

Burroughs

**Reference
Manual**

Volume 2

**A-Series
System
Architecture**

Copyright © 1984, Burroughs Corporation, Detroit, Michigan 48232

Priced Item
Printed in U.S.A.
April 1984

5014954

Burroughs cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded using the Remarks form at the back of the manual, or may be addressed directly to Corporate Documentation-West, Burroughs Corporation, 1300 John Reed Court, City of Industry, California 91745, U.S.A.

LIST OF EFFECTIVE PAGES

Page	Issue
Title	Original
ii	Original
iii	Original
iv	Blank
v thru xix	Original
xx	Blank
xxi thru xxii	Original
1-1 thru 1-22	Original
2-1 thru 2-15	Original
2-16	Blank
3-1 thru 3-101	Original
3-102	Blank
4-1 thru 4-22	Original
A-1 thru A-11	Original
A-12	Blank
B-1 thru B-83	Original
B-84	Blank
C-1 thru C-19	Original
C-20	Blank

TABLE OF CONTENTS

Section	Title	Page
	INTRODUCTION	xxi
1	DATA STRUCTURES	1-1
	General Information	1-1
	Even-Tag Words	1-2
	Operands (Single-and Double-Precision)	1-2
	Numeric Operands	1-5
	Boolean Operands	1-5
	Tag-4 Word	1-6
	Tag-6 Word (Uninitialized Datum)	1-6
	Odd-Tag Words	1-7
	Program Code Words	1-7
	Segments	1-8
	Descriptors	1-8
	Data Segment Descriptor	1-8
	Code Segment Descriptor	1-10
	Stack Segments	1-11
	Paged Segments	1-11
	References	1-12
	Address Couples	1-12
	Fixed-Fence Address Couples	1-12
	Variable-Fence Address Couples	1-12
	Lexical Links	1-13
	IRW (Indirect Reference Word)	1-13
	NIRW (Normal Indirect Reference Word)	1-13
	SIRW (Stuffed Indirect Reference Word)	1-13
	IndexedDD (Indexed Data Descriptor)	1-14
	PCW (Program Control Word)	1-16
	Stack Linkage Words	1-16
	MSCW (Mark Stack Control Word)	1-17
	RCW (Return Control Word)	1-18
	TSCW (Top of Stack Control Word)	1-19
	Interlocks	1-20
	Tags 8-15	1-21
2	STACK CONCEPT AND PROCESSOR STATE	2-1
	General Information	2-1
	Stacks	2-1
	Code Segment Dictionaries	2-1
	Addressing Granularity	2-1
	Program Addressing Environment	2-2
	Memory Addressing	2-3
	Expression Stack	2-7
	Executable Code Streams	2-8
	General Boolean Accumulators	2-10
	Miscellaneous Processor State	2-10
	Processor State Component Sizes	2-11
	Addressing Environment State:	2-11
	Memory Addressing State:	2-11

TABLE OF CONTENTS (Cont)

Section	Title	Page
2	Expression Stack State:	2-12
(Cont)	Code Stream Pointer:	2-12
	Execution State Attributes:	2-12
	General Boolean Accumulators:	2-12
	Miscellaneous State:	2-12
	System Control	2-13
	Programming Restrictions Due to Hidden State	2-15
3	OPERATOR SET AND COMMON ACTIONS	3-1
	General Information	3-1
	Operators and Code Streams	3-1
	Primary, Variant and Edit Operators	3-2
	Common Actions : common action	3-2
	Initial and Restart State	3-2
	Checks and Interrupts	3-3
	Expression Stack Control	3-3
	Top-of-Stack Push Operations	3-3
	Top-of-Stack Pop Operations	3-4
	Descriptor Interpretation	3-4
	Computational Operators	3-5
	Numeric Operand Interpretation	3-5
	Representable Operand Formats	3-6
	Single-Precision Operand Values	3-6
	Double-Precision Operand Values	3-6
	Automatic Arithmetic Functions	3-6
	Numeric-Interpretation Operators	3-7
	Arithmetic Operators	3-7
	ADD (add)	3-7
	SUBT (subtract)	3-7
	MULT (multiply)	3-7
	MULX (extended multiply)	3-8
	DIVD (divide)	3-8
	IDIV (integer divide)	3-8
	RDIV (remainder divide)	3-8
	NORM (normalize)	3-8
	AMIN and AMAX (arithmetic minimum and maximum)	3-9
	Relational Operators	3-9
	LESS (less than)	3-9
	LSEQ (less than or equal to)	3-9
	EQL (equal to)	3-9
	NEQL (not equal to)	3-9
	GREQ (greater than or equal to)	3-9
	GRTR (greater than)	3-9
	Range Test Operators	3-9
	RNGT (range test)	3-10
	DRNT (dynamic range test)	3-10
	Numeric Type-Transfer Operators	3-10
	NTIA (integerize truncated)	3-11

TABLE OF CONTENTS (Cont)

Section	Title	Page
3 (Cont)	NTGR (integerize rounded)	3-11
	SNGL (set to single-precision rounded)	3-11
	SNGT (set to single-precision truncated)	3-11
	NTTD (integerize double-precision truncated)	3-11
	NTGD (integerize double-precision rounded)	3-12
	aISX (integer subset exception action)	3-12
	Scale Left	3-12
	SCLF (scale left)	3-13
	DSLX (dynamic scale left)	3-13
	Scale Right	3-13
	SCRS (scale right save)	3-14
	DSRS (dynamic scale right save)	3-14
	SCRT (scale right truncate)	3-14
	DSRT (dynamic scale right truncate)	3-14
	SCRR (scale right rounded)	3-15
	DSRR (dynamic scale right rounded)	3-15
	SCRF (scale right final)	3-15
	DSRF (dynamic scale right final)	3-15
	Binary to Decimal Conversion	3-16
	BCD (binary convert to decimal)	3-16
	DBCD (dynamic binary convert to decimal)	3-17
	Bit Vector Interpretation	3-17
	Logical Operators	3-17
	LNOT (logical not)	3-17
	LAND (logical and)	3-17
	LOR (logical or)	3-18
	LEQV (logical equivalence)	3-18
	Relational Operator	3-18
	SAME (logical equality)	3-18
	Literal Operators	3-18
	ZERO (insert literal zero)	3-18
	ONE (insert literal one)	3-18
	LT8 (insert 8 bit literal)	3-18
	LT16 (insert 16 bit literal)	3-18
	LT48 (insert 48 bit literal)	3-19
	Bit-Vector Type-Transfer Operators	3-19
	STAG (set tag)	3-19
	XTND (set to double-precision)	3-20
	JOIN (set two singles to double)	3-20
	SPLT (set double to two singles)	3-20
	Evaluate Word Structure Operators	3-21
	RTAG (read tag)	3-21
	CBON (count binary ones)	3-21
	LOG2 (leading one test)	3-21
	Word Manipulation Operators	3-21
BSET (bit set)	3-22	
DBST (dynamic bit set)	3-22	

TABLE OF CONTENTS (Cont)

Section	Title	Page
3 (Cont)	BRST (bit reset)	3-22
	DBRS (dynamic bit reset)	3-22
	ISOL (field isolate)	3-23
	DISO (dynamic field isolate)	3-23
	INSR (field insert)	3-23
	DINS (dynamic field insert)	3-24
	FLTR (field transfer)	3-24
	DFTR (dynamic field transfer)	3-24
	CHSN (change sign)	3-25
	Linear Index-Function Operator	3-25
	OCRX (occurs index)	3-25
	Reference Generation and Evaluation Operators	3-26
	Double Precision	3-26
	Stack references	3-27
	Lexical Link Evaluation	3-27
	aLXLK (evaluate lexical link)	3-27
	Lexical Chains	3-27
	aLXCH (traverse lexical chain)	3-27
	Address-Couple Evaluation	3-27
	Evaluation of References	3-28
	Address Couple Parameters	3-28
	NIRWs	3-28
	SIRWs	3-28
	IndexedWordDDs	3-29
	PCWs	3-29
	IRW Chains	3-29
	Reference Chains	3-30
	Reference Generation Operators	3-31
	NAMC (name call)	3-31
	LNMC (long name call)	3-31
	STFF (stuff)	3-32
	INDX (index)	3-32
	INXA (index by means of address-couple parameter)	3-33
	MPCW (make PCW)	3-34
	Read Evaluation Operators	3-34
	aFOP (Fetch Operand)	3-34
	aCPY (fetch copy descriptor)	3-34
	VALC (value call)	3-35
	LVLC (long value call)	3-36
	NXLV (index and load value)	3-36
	NXVA (index and load value by means of address-couple parameter)	3-37
	NXLN (index and load name)	3-37
	EVAL (evaluate)	3-38
	LOAD (load)	3-39
	LODT (load transparent)	3-39
	Store Evaluation Operators	3-41
	Normal Store Operators	3-41
	STOD (store delete)	3-42

TABLE OF CONTENTS (Cont)

Section	Title	Page
3	STON (store non-delete)	3-42
(Cont)	STAD and STAN (store delete/non-delete by means of address-couple)	3-42
	Overwrite Operators	3-43
	OVRD (overwrite delete)	3-43
	OVRN (overwrite non-delete)	3-43
	RDLK (read and lock)	3-44
	Interlock Operators	3-45
	LOK (lock interlock)	3-47
	UNLK (unlock interlock)	3-47
	LOKC (conditional lock interlock)	3-48
	LKID (read interlock status)	3-48
	Processor State Operators	3-48
	Code Stream Pointer Distribution	3-48
	aPRCW (distribute PCW/RCW code-stream pointer)	3-48
	Branching Operators	3-49
	Static Branches	3-49
	BRUN (branch unconditional)	3-49
	BRTR and BRFL (branch true and branch false)	3-49
	Dynamic Branches	3-50
	DBUN (dynamic branch unconditional)	3-50
	DBTR and DBFL (dynamic branch true and dynamic branch false)	3-51
	Stack Structure Operators	3-51
	Display Update	3-51
	Procedure Entry Operators	3-51
	MKST (mark stack)	3-52
	MKSN (mark-stack bound to name-call)	3-53
	IMKS (insert mark stack)	3-54
	ENTR (enter)	3-54
	Completing the MSCW	3-55
	Constructing the RCW	3-55
	Initializing the Processor State	3-56
	aACCE (accidental entry)	3-56
	aINTE (interrupt entry)	3-57
	Procedure Exit Operators	3-59
	EXIT (exit)	3-59
	RETN (return)	3-61
	Stack Environment Operator	3-62
	MVST (move to stack)	3-62
	Deactivating the Current Stack	3-62
	Changing the Addressing Environment and Identifying the Destination Stack	3-62
	Restoring Destination Stack State	3-63
	Updating the Lexical Environment State	3-63
	Top-of-Stack Operators	3-64
	DLET (delete top-of-stack)	3-64
	EXCH (exchange top-of-stack)	3-64
	DUPL (duplicate top-of-stack)	3-65
	RSUP (rotate stack up)	3-65

TABLE OF CONTENTS (Cont)

Section	Title	Page
3	RSDN (rotate stack down)	3-65
(Cont)	Processor-State Manipulation Operators	3-65
	Read State Operators	3-66
	RTFF (read true-false flip-flop)	3-66
	RSNR (read SNR)	3-66
	WHOI (read processor id)	3-66
	WATI (read machine identification)	3-66
	RTOD (read time of day clock)	3-67
	RPRR (read processor register)	3-67
	RIPS (read internal processor state)	3-67
	Set State Operators	3-68
	SXSX (set external sign flip-flop)	3-68
	EEXI (enable external interrupts)	3-68
	DEXI (disable external interrupts)	3-68
	SINT (set interval timer)	3-68
	WTOG (write time of day clock)	3-68
	SPRR (set processor register)	3-69
	RUNI (indicate running)	3-70
	WIPS (write internal processor state)	3-70
	ZIC (zero Interrupt_Count)	3-70
	Read and Set State Operator	3-70
	ROFF (read and reset overflow flip-flop)	3-70
	Data Array Operators	3-71
	Searching Operators	3-71
	LLLU (linked list lookup)	3-71
	SRCH (masked search for equal)	3-72
	Pointer Operators	3-73
	element_size conventions	3-73
	Length Argument	3-74
	Source Argument	3-74
	Short-Source Operators	3-74
	Destination Argument	3-75
	Source1 and Source2 Arguments	3-75
	Overlapping Source and Destination	3-75
	Update Of Pointer-Operator Arguments	3-77
	Unconditional Character-Transfer Operators	3-78
	Character-Relational Operators	3-78
	Scan Operators	3-79
	Transfer Operators	3-80
	Character-Sequence Compare Operators	3-81
	Character Set-Membership Operators	3-82
	Scan Operators	3-82
	Transfer Operators	3-83
	Character-Sequence Extraction Operator	3-83
	Character Translate Operator	3-84
	Decimal-Character-Sequence Operators	3-85
	Pack Operators	3-86

TABLE OF CONTENTS (Cont)

Section	Title	Page
3 (Cont)	Unpack Operators	3-87
	Unpack-Unsigned Operators	3-87
	Unpack-Signed Operators	3-88
	Input-Convert Operators	3-89
	Word-Transfer Operators	3-90
	Word-Transfer-Protected Operators	3-90
	Word-Transfer-Overwrite Operators	3-90
	Primitive Display Operator	3-91
	SHOW (primitive display)	3-91
	Edit Operators	3-91
	Enter-Edit Operators	3-92
	Table edit-mode	3-92
	Single edit-mode	3-92
	Enter-Table-Edit Operators	3-92
	Enter-Single-Edit Operators	3-94
	Edit-Mode Operators	3-95
	Character Skip Operators	3-95
	Skip Forward	3-95
	Skip Reverse	3-95
	Character Insert Operators	3-96
	INSU (insert unconditional)	3-96
	INSC (insert conditional)	3-96
	INOP (insert overpunch)	3-96
	INSG (insert display sign)	3-96
	ENDF (end float)	3-97
	Character Move Operators	3-97
	MCHR (move characters)	3-97
	MVNU (move numeric)	3-97
	MINS (move with insert)	3-98
	MFLT (move with float)	3-98
	Miscellaneous Edit Operators	3-99
	RSTF (reset float flip-flop)	3-99
	ENDE (end edit)	3-99
	External Communication Operators	3-99
	CUIO (communicate with Universal I/O)	3-99
	SCNI/ SCNO (scan in/out) IDLE (idle until interrupt)	3-99
	PAUS (pause until interrupt)	3-99
	REMC (read external memory control)	3-99
	WEMC (write external memory control)	3-100
	Miscellaneous Operators	3-100
NOOP (no operation)	3-100	
DLAY (delay)	3-100	
PUSH (push working stack onto activation record)	3-100	
STOP (unconditional processor halt)	3-101	
HALT (conditional processor halt)	3-101	
NVLD (invalid operator)	3-101	
ASRT (assert)	3-101	

TABLE OF CONTENTS (Cont)

Section	Title	Page
3 (Cont)	VARI (introduce variant operator)	3-101
4	INTERRUPTS	4-1
	General Information	4-1
	Interrupt Parameters	4-1
	Interrupt ID Parameter	4-1
	Resumption Conditions	4-6
	P2 parameter	4-7
	Superhalt	4-7
	Interrupt Definition	4-8
	Operator Dependent Interrupts	4-8
	MCP Service	4-10
	Presence Bit	4-10
	Paged Array	4-10
	Binding Request	4-11
	Stack Overflow	4-12
	Block Exit	4-12
	Locking and Unlocking	4-12
	Error Reporting	4-12
	Invalid Operator	4-13
	Undefined Operator	4-13
	Invalid Stack Argument	4-13
	Invalid Argument Value	4-14
	Invalid Code Parameter	4-14
	Invalid Reference	4-14
	Invalid Reference Chain	4-14
	Invalid Object	4-16
	Invalid Index	4-16
	Memory Protect	4-17
	Divide by Zero	4-17
	Exponent-Overflow	4-17
	Exponent-Underflow	4-18
	Precision Loss	4-18
	Integer-Overflow	4-18
	Stack-Underflow	4-19
	Stack Structure Error	4-19
	Code Segment Error	4-20
	Invalid Program Word	4-20
	Page Structure Error	4-20
	False Assertion	4-21
	Alarm Interrupts	4-21
	Invalid Address	4-21
	Uncorrectable Memory Error	4-21
	Loop Timer	4-22
	Hardware Error	4-22
	External Interrupts	4-22
A	OPERATOR SET	A-1
	General Information	A-1

TABLE OF CONTENTS (Cont)

Section	Title	Page
B	OPERATOR REFERENCE SUMMARIES	B-1
	General Information	B-1
	The Code-Stream Encoding Of The Operator	B-1
	Clients	B-1
	Stack State Transformation	B-1
	Interrupts That May Be Generated	B-1
	Symbols Used In This Appendix	B-2
	Operator and Common Action Listing	B-3
	aACCE	B-3
	aCPY	B-3
	aFOP	B-4
	aINTE	B-4
	aISX	B-5
	aLXCH	B-5
	aLXLK	B-6
	aPRCW	B-6
	ADD	B-7
	AMAX	B-7
	AMIN	B-7
	ASRT	B-8
	BCD	B-8
	BRFL	B-8
	BRTR	B-9
	BRST	B-9
	BRUN	B-9
	BSET	B-9
	CBON	B-9
	CEQD	B-10
	CEQU	B-10
	CGED	B-10
	CGEU	B-11
	CGTD	B-11
	CGTU	B-11
	CHSN	B-11
	CLED	B-11
	CLEU	B-11
	CLSD	B-12
	CLSU	B-12
	CNED	B-12
	CNEU	B-12
	CUIO	B-12
	DBCD	B-13
	DBFL	B-13
	DBRS	B-13
	DBST	B-14
	DBTR	B-14
	DBUN	B-14

TABLE OF CONTENTS (Cont)

Section	Title	Page
B (Cont)	DEXI	B-14
	DFTR	B-15
	DINS	B-15
	DISO	B-16
	DIVD	B-16
	DLAY	B-16
	DLET	B-17
	DRNT	B-17
	DSLFL	B-17
	DSRF	B-18
	DSRR	B-18
	DSRS	B-18
	DSRT	B-19
	DUPL	B-19
	EEXI	B-19
	ENDE	B-19
	ENDF	B-20
	ENTR	B-21
	EQUL	B-22
	EVAL	B-22
	EXCH	B-22
	EXIT	B-23
	EXPU	B-24
	EXSD	B-24
	EXSU	B-25
	FLTR	B-25
	GREQ	B-25
	GRTR	B-25
	HALT	B-26
	ICLD	B-26
	ICRD	B-26
	ICUD	B-26
	ICVD	B-27
	ICVU	B-27
	IDIV	B-27
	IDLE	B-27
	IMKS	B-28
	INDX	B-29
	INOP	B-30
	INSC	B-30
	INSG	B-31
	INSR	B-31
	INSU	B-31
	INXA	B-32
	ISOL	B-32
	JOIN	B-33
	LAND	B-33

TABLE OF CONTENTS (Cont)

Section	Title	Page
B (Cont)	LEQV	B-33
	LESS	B-33
	LKID	B-34
	LLLU	B-34
	LNMC	B-35
	LNOT	B-35
	LOAD	B-35
	LODT	B-36
	LOG2	B-36
	LOK	B-36
	LOKC	B-37
	LOR	B-37
	LSEQ	B-37
	LT8	B-37
	LT16	B-37
	LT48	B-38
	LVLC	B-38
	MCHR	B-38
	MFLT	B-38
	MINS	B-39
	MKSN	B-39
	MKST	B-39
	MPCW	B-40
	MULT	B-40
	MULX	B-40
	MVNU	B-41
	MVST	B-42
	NAMC	B-43
	NEQL	B-43
	NOOP	B-43
	NORM	B-43
	NTGD	B-44
	NTGR	B-44
	NTIA	B-44
	NTTD	B-44
	NVLD	B-45
	NXLN	B-45
	NXLV	B-46
	NXVA	B-47
	OCRX	B-47
	ONE	B-48
	OVRD	B-48
	OVRN	B-48
	PACD	B-48
	PACU	B-49
	PAUS	B-49
	PKLD	B-49

TABLE OF CONTENTS (Cont)

Section	Title	Page
B (Cont)	PKRD	B-49
	PKUD	B-50
	PUSH	B-50
	RDIV	B-50
	RDLK	B-51
	REMC	B-51
	RETN	B-52
	RIPS	B-52
	RNGT	B-53
	ROFF	B-53
	RPRR	B-53
	RSDN	B-54
	RSNR	B-54
	RSTF	B-54
	RSUP	B-54
	RTAG	B-55
	RTFF	B-55
	RTOD	B-55
	RUNI	B-55
	SAME	B-56
	SCLF	B-56
	SCRF	B-56
	SCRR	B-57
	SCRS	B-57
	SCRT	B-57
	SEQD	B-58
	SEQU	B-58
	SFDC	B-58
	SFSC	B-59
	SGED	B-59
	SGEU	B-59
	SGTD	B-59
	SGTU	B-59
	SHOW	B-60
	SINT	B-60
	SISO	B-61
	SLED	B-61
	SLEU	B-61
	SLSD	B-62
	SLSU	B-62
	SNED	B-62
	SNEU	B-62
	SNGL	B-62
	SNGT	B-63
	SPLT	B-63
	SPRR	B-64
	SRCH	B-64
	SRDC	B-64

TABLE OF CONTENTS (Cont)

Section	Title	Page
B (Cont)	SRSC	B-65
	STAD	B-65
	STAG	B-66
	STAN	B-66
	STFF	B-66
	STOD	B-67
	STON	B-68
	STOP	B-68
	SUBT	B-68
	SWFD	B-69
	SWFU	B-69
	SWTD	B-69
	SWTU	B-70
	SXSN	B-70
	TEED	B-70
	TEEU	B-71
	TEQD	B-71
	TEQU	B-72
	TGED	B-72
	TGEU	B-72
	TGTD	B-72
	TGTU	B-72
	TLED	B-72
	TLEU	B-73
	TLSD	B-73
	TLSU	B-73
	TNED	B-73
	TNEU	B-73
	TRNS	B-74
	TUND	B-75
	TUNU	B-75
	TWFD	B-76
	TWFU	B-76
	TWOD	B-77
	TWOU	B-77
	TWSD	B-78
	TWSU	B-78
	TWTD	B-79
	TWTU	B-79
	UNLK	B-79
UPLD	B-79	
UPLU	B-79	
UPRD	B-79	
UPRU	B-80	
UPUD	B-80	
UPUU	B-80	
USND	B-81	
USNU	B-81	

TABLE OF CONTENTS (Cont)

Section	Title	Page
B (Cont)	VALC	B-81
	VARI	B-81
	WATI	B-82
	WEMC	B-82
	WHOI	B-82
	WIPS	B-82
	WTOD	B-83
	XTND	B-83
	ZERO	B-83
	ZIC	B-83
C	OPERATOR DEPENDENT INTERRUPT REFERENCE SUMMARIES	C-1
	General Information	C-1
	Binding Request	C-2
	Block Exit	C-2
	Code Segment Error	C-2
	Divide by Zero	C-2
	Exponent Overflow	C-2
	Exponent Underflow	C-3
	False Assertion	C-3
	Integer Overflow	C-3
	Invalid Argument Value	C-5
	Invalid Code Parameter	C-6
	Invalid Index	C-6
	Invalid Object	C-8
	Invalid Operator	C-9
	Invalid Reference	C-9
	Invalid Reference Chain	C-10
	Invalid Stack Argument	C-10
	Locking	C-13
	Memory Protect	C-13
	Paged Array	C-14
	Page Structure Error	C-15
	Precision Loss	C-16
	Presence Bit	C-16
	Stack Overflow	C-17
	Stack Structure Error	C-18
	Stack Underflow	C-18
	Undefined Operator	C-19
	Unlocking	C-19

LIST OF ILLUSTRATIONS

Figure	Title	Page
1-1	Word Format	1-1
1-2	Single Precision Operand Format	1-3
1-3	Double Precision Operand Format	1-4
1-4	Boolean Operand Format	1-6
1-5	Tag-4 Word Format	1-7
1-6	Tag-6 Word Format	1-7
1-7	Program Code Word Format	1-8
1-8	Data Descriptor Format	1-9
1-9	Code Segment Descriptor Format	1-11
1-10	Normal Indirect Reference Word Format	1-13
1-11	SIRW Word Format	1-14
1-12	Indexed Word Data Descriptor Format	1-15
1-13	Indexed Character Data Descriptor (Pointer) Format	1-16
1-14	Program Control Word Format	1-17
1-15	Mark Stack Control Word (MSCW) Format	1-18
1-16	Return Control word (RCW) Format	1-19
1-17	Top Of Stack Control Word (TSCW) Format	1-20
1-18	Interlock Control word Format	1-21
2-1	Addressing environment example	2-4
2-2	Memory Environment Mapping	2-5
2-3	Topmost Activation Record Example	2-8
2-4	Processor Code Stream Pointer	2-9
4-1	P-1 Operator Dependent Interrupt (ODI) ID Parameter Format	4-2
4-2	P-1 Alarm Interrupt ID Parameter Format	4-3
4-3	P-1 External Interrupt ID Parameter Format	4-4

LIST OF TABLES

Table	Title	Page
1-1	Address Couple Fence Decoding	1-12
A-1	Operators, Alphabetical List	A-1
A-2	Operators, Numerical List	A-6

INTRODUCTION

This manual describes and defines an architecture used in Burroughs Corporation data processing system products. Products that include architecture described in this manual are essentially compatible with each other, including four generations of prior system products. It is intended that this tradition extend to systems developed in the future.

"Essentially compatible", as used in this manual, means that programs written to process on one system also process on other systems sharing the same architectural design. Any reprogramming or adaptation to process a program on a similar system architecture will be of a minor nature. Programs originally written for execution on a prior generation system may require adaptation to account for present-day peripheral devices, which did not exist when the prior system architecture was designed. This is also true when adapting a present-day program to execute on a prior system design architecture.

This architecture is designed for use in systems with different hardware characteristics, called "implementations" in this manual. System implementations may differ in the manner or method of handling internal operations or reporting system status information. When "implementation" is used in this manual, it implies a possible variance between systems with different hardware characteristics. "Implementation" variances may explain why particular programs execute differently on systems that use this common architecture design.

This manual is designed to be used as the second volume of a two-volume System Reference Manual, for all systems that utilize this common architecture design. The first volume in a System Reference Manual set describes the characteristics of the hardware used in the system, and identifies the system designation. This second volume then describes the common operating system concepts and requirements.

This document is organized as four sections, followed by three reference appendixes. Sections are numbered and appendixes listed alphabetically.

Section 1: Data Structures

This section describes the data structures and formats used in this architecture. All structures and formats, including system control structures and formats, are given.

Section 2: Stack Concept and Processor State

This section describes the concepts and operating characteristics of a stack. The stack links the hardware and software of a system together, to initiate Activation Records of a program or process upon the system. Processor state (the system status required by the architecture) and memory addressing environment of the architecture are also described in this section.

Section 3: Operator Set and Common Actions

This section defines all operators in the architecture repertoire. Common Actions, which are general functions of the common architecture, are also described.

Section 4: Interrupts

Interrupts, generated by the architecture to document and define events and errors, are defined and described in this section.

Appendix A: Operator Code Lists

This appendix lists all operators of the architecture, in alphabetical order and by numeric-code value. It also identifies the mnemonic terms that distinguish operator functions. The system operating Mode for each operator code is specified.

Appendix B: Operator Reference Summaries

This appendix lists operator mnemonics in alphabetic order. For each operator, the changes to the top of the stack resulting from execution of the operator are given. In addition, all interrupts that can be generated during execution of an operator are identified. For each interrupt that is listed, the most probable cause is given.

Appendix C: Interrupt Reference Summaries

This appendix itemizes all Operator Dependent Interrupts (ODI), in alphabetic order. For each ODI, operators (listed in alphabetical order) that generate that ODI, along with the most probable cause for the interrupt, is given.

From time-to-time this manual digresses to provide pragmatic commentary. Pragmatic commentaries describe practical aspects and as such may represent interruptions of technical subject descriptions. The following convention is used to inform a reader of the start of pragmatic discussion. Pragmatic discussions terminate at new topic headings, which change the subject. The following is an example of a pragmatic commentary:

Pragmatic Notes

Pragmatic Notes Subject : Data Types

This architecture supports a number of data types that can be uniquely distinguished from each other by their structure; all such data types are defined in this section. This architecture supports additional data types that are distinguished by context; some of these are defined in this section, while the remainder are defined along with the applicable operators.

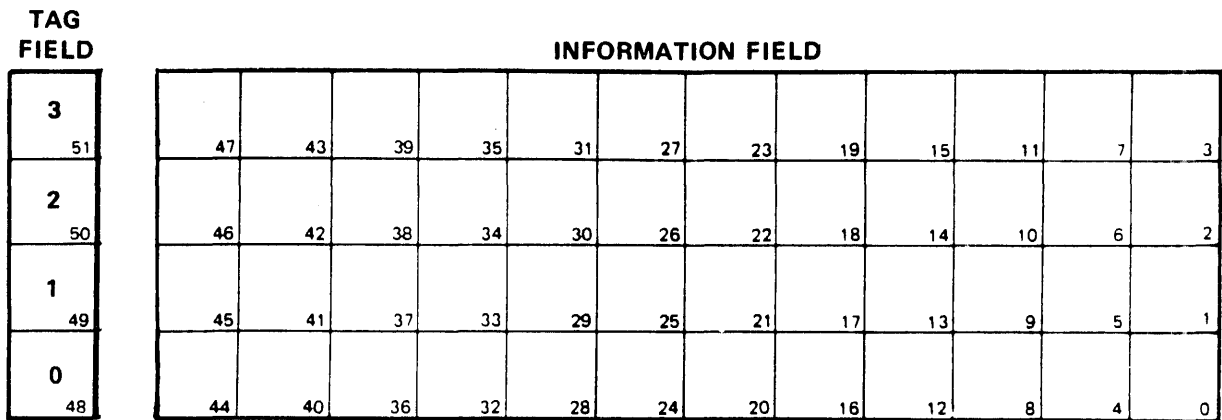
SECTION 1 DATA STRUCTURES

GENERAL INFORMATION

Words are the fundamental unit of data. A word consists of a tag field and an information field. Figure 1-1 shows the structure of a word and identifies the fields and bits within the word structure.

A tag field consists of four binary bits. The value of the tag field bits provides the general interpretation of data contained in the word information field. There are 16 different tag field values possible, but all possible values are not currently used. Some tag field values define words that have variable interpretations, but in these cases, information field bits further define the particular interpretation that applies to that word.

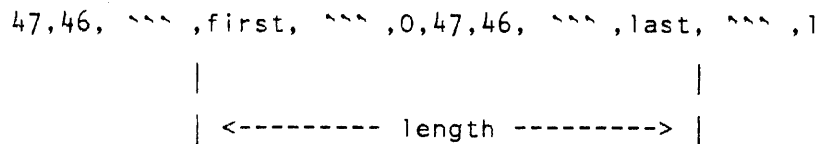
Words have 48 information field bits. The information field bits are numbered 47 down to zero, from the high-order bit down to the low-order bit. Within a word, the 48 bits are subdivided into smaller bit fields. A smaller field within the information field of a word is denoted [first:length]. First is the bit number of the high-order bit in the field ($first \leq 47$), and length is the field length in bits ($length \leq 48$). Fields are often given names, such as "stack_number" for "[47:12]". A field of an object, or the class formed by applying the field specification to a class of objects, is denoted by suffixing the field name or specification to the object or type name with an interposed dot: x.[46:1] or SIRW.stack_number.



MV5353

Figure 1-1. Word Format

When necessary, fields are wrapped around from the lowest-order bit to the highest-order bit. If $length > first + 1$ then $length - (first + 1)$ bits are concatenated starting from the highest-order bit (47). The following sequence illustrates the field [first:length] in the case where $length > first + 1$:



Word types are distinguished by tag value and frequently by additional type bits in the word. This section defines the data type name, tag and type bit identification, field interpretation, and semantics of each word type.

In the remainder of the document, word types will be referred to by type name. Because the data type double-precision consists of two words, the term "item" is used (instead of "word") to refer to an entity whose type may be double-precision.

EVEN-TAG WORDS

Words with even tag values serve primarily as computation arguments, rather than as reference arguments or control structures.

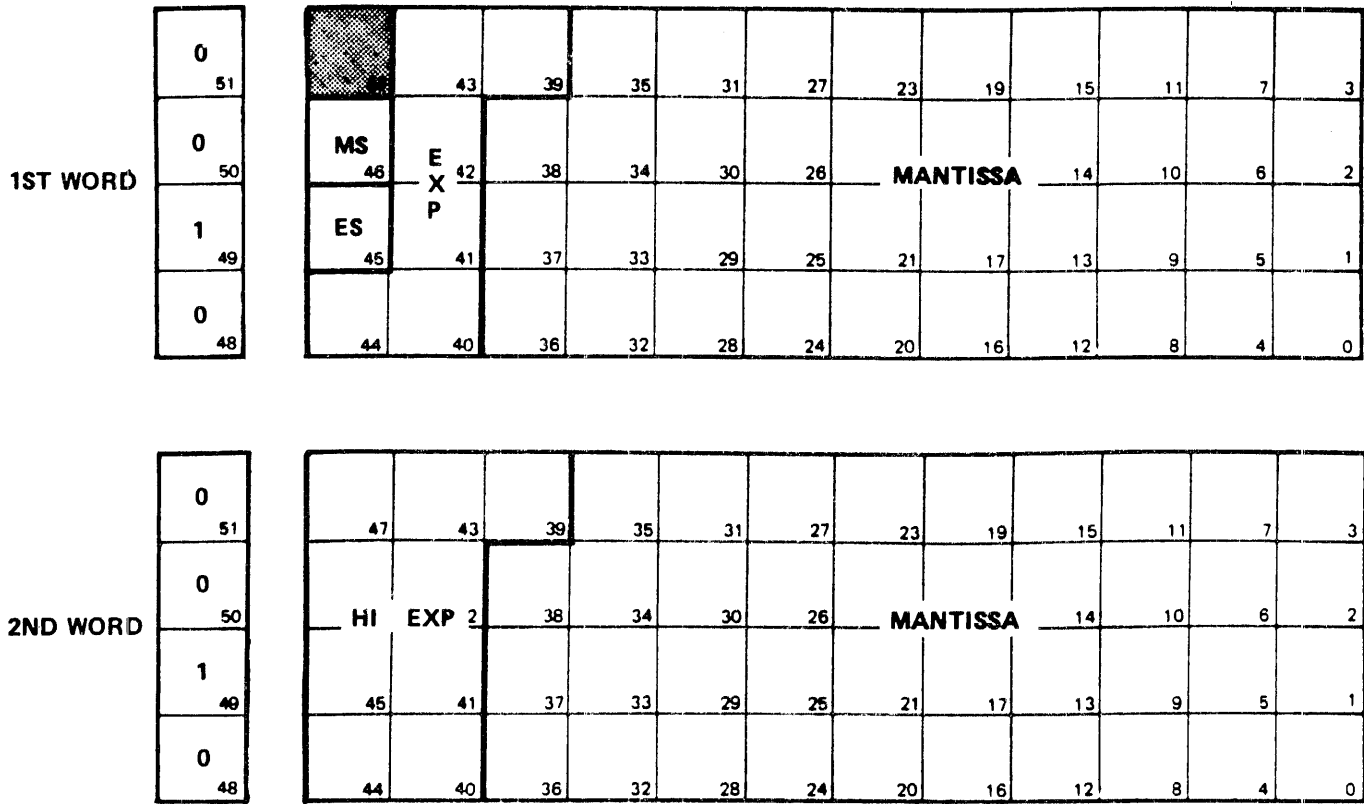
An important aspect of such words is that they can be stored over in memory by normal store (as opposed to overwrite) operations, whereas all odd tagged words are protected from normal writes.

Operands (Single-and Double-Precision)

The great majority of data items dealt with by programs are operands, of which there are two types. A single-precision operand is a single word with a tag of 0 (see figure 1-2). A double-precision operand is a pair of consecutive words, both with a tag of 2 (the "first" word is always the word at the lower memory address whenever the operand is stored in memory). Figure 1-3 shows a double-precision operand.

Throughout this document, the term operand is used solely to refer to the type union single and double-precision.

Neither operand type has a unique interpretation applied to it. Operators, according to their function, apply different interpretations. For example, operands are interpreted as numeric values, bit vectors, and character sequences by arithmetic, word manipulation, and pointer operators, respectively. Numeric and Boolean operands are generated and interpreted by a wide variety of operators, and are therefore defined here. Section 3 defines additional operand interpretations with the operator groups that apply them.



MV5355

1st word:

mant__sign	[46: 1]	Mantissa sign (0 = positive, 1 = negative)
exp__sign	[45: 1]	Exponent sign (0 = positive and 1 = negative)
exponent	[44: 6]	The low-order 6 bits of the exponent
mantissa	[38:39]	The integral portion of the mantissa

2nd word:

hi__order__exp	[47: 9]	The high-order 9 bits of the exponent
mantissa	[38:39]	The fractional portion of the mantissa

Figure 1-3. Double Precision Operand Format

In processor state, a double-precision operand is treated as a 96-bit operand with a single 4-bit tag equal to 2. When a tag-2 item is pushed onto the stack, the high-order and low-order words are written in that order, both with tags of 2. When a tag-2 word is popped from the expression stack as an argument, the next word is also popped from the stack and the two words joined to form the double-precision item; the word higher in the stack is taken as the low-order half of the double. If the second word popped does not have tag = 2, the action is undefined.

Numeric Operands

Many operators interpret operands as numeric values. The structure of the numeric data is defined in this section; details of numeric interpretation are found with the operator descriptions in Numeric Operand Interpretation.

A single-precision floating-point operand is represented as a word with the following fields:

	[47: 1]	Not used
mant__sign	[46: 1]	Mantissa sign (0 = positive, 1 = negative)
exp__sign	[45: 1]	Exponent sign (0 = positive, 1 = negative)
exponent	[44: 6]	The power of eight by which the mantissa is scaled
mantissa	[38:39]	The integer magnitude of the number before scaling

A single-precision floating-point operand with exponent = 0 is used as the canonical representation of a single-precision integer; an operand in this form is called a single__integer.

The form "k-bit integer" (where k is an integer in {1 to 39}) is used to specify an operator output value in which field [47:48-k] contains zero. The same term is used to specify an operator input value in which fields [46:1] and [44:45-k] contain zero, or field [44:45] contains zero. (Bits 47 and 45 are insignificant in integer representations; bit 46 is insignificant if the mantissa value is zero.)

A double-precision floating-point operand is represented as two words with the following fields:

First word:

	[47: 1]	Not used
mant__sign	[46: 1]	Mantissa sign (0 = positive, 1 = negative)
exp__sign	[45: 1]	Exponent sign (0 = positive and 1 = negative)
exponent	[44: 6]	The low-order 6 bits of the exponent
mantissa	[38:39]	The integral portion of the mantissa

Second word:

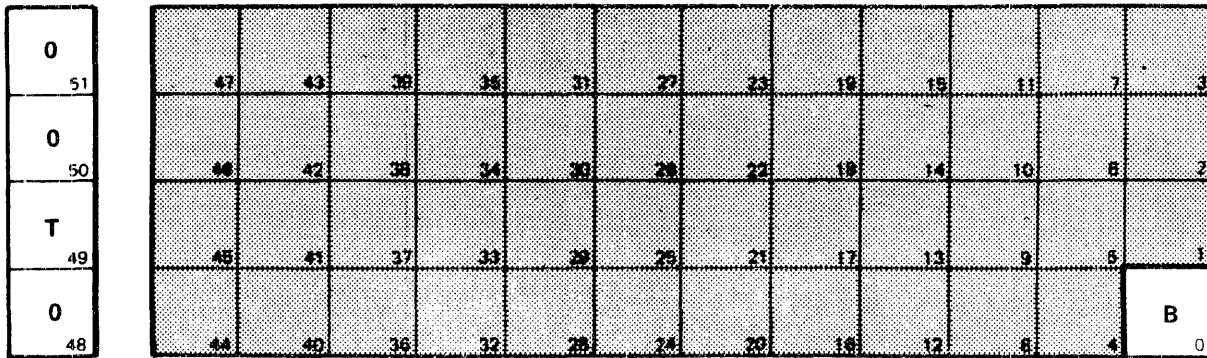
hi__order__exp	[47: 9]	The high-order 9 bits of the exponent
mantissa	[38:39]	The fractional portion of the mantissa

A double-precision floating-point operand with exponent = 13 is used as the canonical representation of a double-precision integer; an operand in this form is called a double__integer.

Boolean Operands

Figure 1-4 shows the word format of a Boolean operand. Some operators generate and other operators consume operands interpreted as Boolean values. This architecture represents the Boolean values TRUE and FALSE as binary 1 and 0, respectively. The terms True and False are used to specify Boolean values: In the specification of a stack output from an operator, True and False are defined as the 1-bit integer values 1 and 0, respectively. In the specification of a stack argument for an operator, True and False are defined as operands with bit [0:1] equal to 1 or 0, respectively. Boolean interpretation ignores field [47:47] of any operand and the second word of a double-precision operand.

Boolean operands are generated by the relational operators, among others. Operands are interpreted as Boolean values by the branch operators and the ASRT (assert) operator. The logical operators do not interpret Boolean values; rather, they perform Boolean arithmetic upon all the bits of an item, in parallel.



MV5356

tag (0: single-precision,
 2: double-precision (2nd word ignored))

[0: 1] Boolean value (0 = false, 1 = true)

Figure 1-4. Boolean Operand Format

Tag-4 Word

Tag-4 words are data words, but the only interpretation applied to them is as a 48-bit vector by a class of computational operators. Figure 1-5 shows the format of a tag-4 word.

Tag-4 words cannot normally be fetched to the expression stack as operands, and they are not valid arguments for arithmetic computational operators. However, they may be stored over by normal store operators.

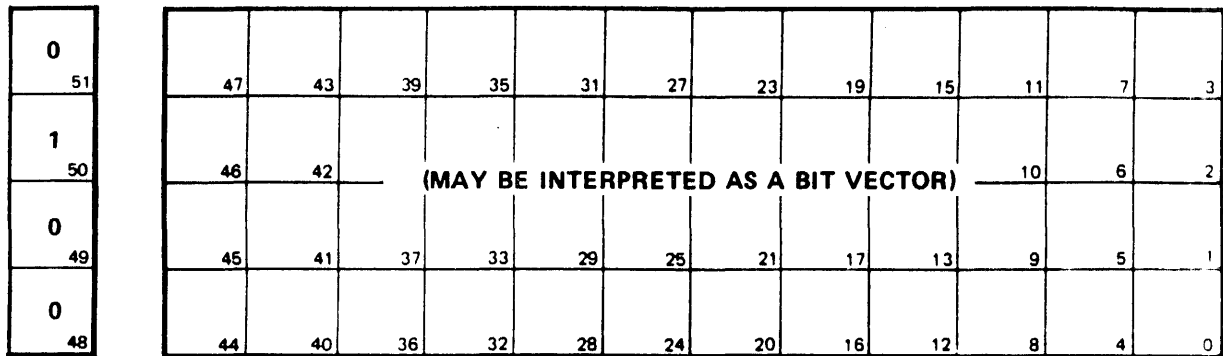
This architecture does not exploit the tag value 4; the value is being held in reserve for application in future levels of this architecture. Software uses some configurations of tag-4 data as flags for various purposes.

Tag-6 Word (Uninitialized Datum)

Figure 1-6 shows the format of a tag-6 word. "Uninitialized datum" and "tag-6 word" are synonymous type names for a word whose tag is 6.

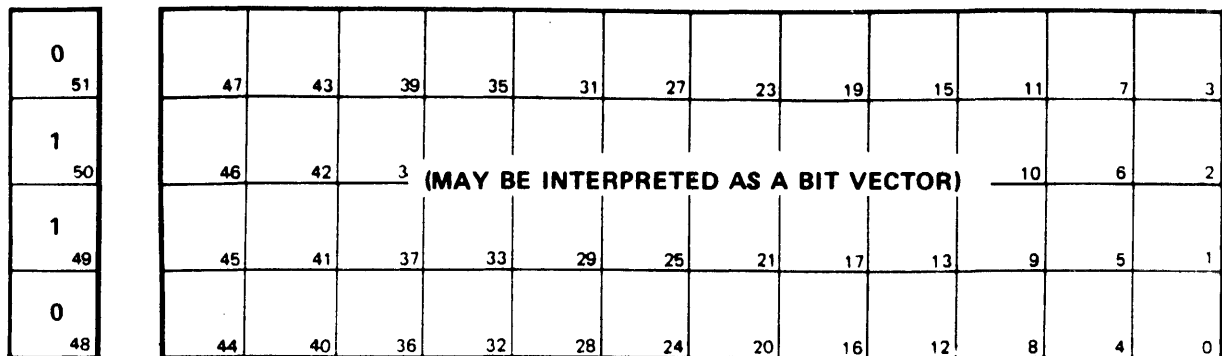
Tag-6 words are data words, but the only interpretation applied to them is as a 48-bit vector by a class of computational operators.

Tag-6 words cannot normally be fetched to the expression stack as operands, and they are not valid arguments for arithmetic computational operators. However, they may be stored over by normal store operators.



MV5357

Figure 1-5. Tag-4 Word Format



MV5358

Figure 1-6. Tag-6 Word Format

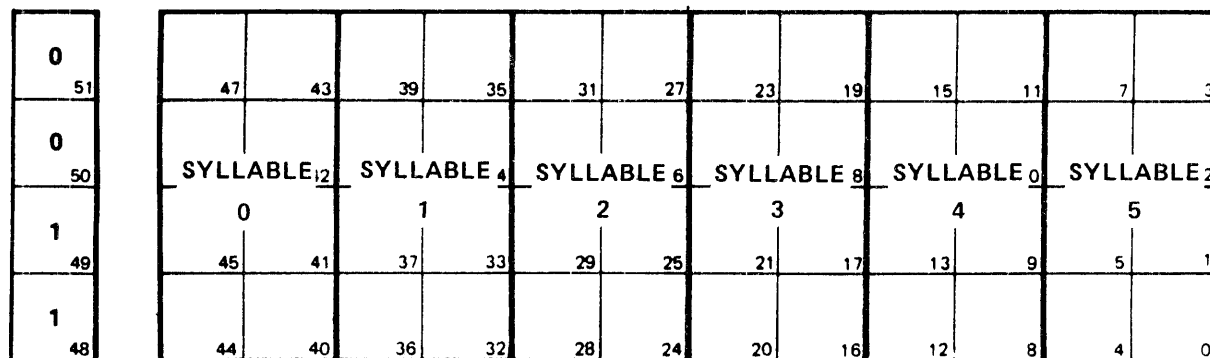
This architecture defines minimal semantics for tag-6 words. One utility is implied by the type name: a tag-6 word can be used as the initial value of a variable; an operator expecting an operand or descriptor will generate an interrupt, but a normal store operator can be used to assign an operand value to the variable. It is conventional for software to use the value zero with tag = 6 for this purpose; other tag-6 values are used by software to create distinctive flags for various purposes.

ODD-TAG WORDS

Program code words, program and data control words, and memory address reference words have odd tag field values. Words with odd tag values are protected against accidental destruction (by overwriting) while they are present in actual memory (but not in virtual memory). This architecture uses odd tag words to implement program control and direction over user programs, by means of a Master Control Program (MCP). Control functions such as the manual system initialization process and the interrupt control mechanism use odd tag words to implement their functions.

PROGRAM CODE WORDS

Variable length operator sequences are stored in arrays of program code words called code segments. Each program code-word contains six 8-bit containers called syllables, numbered zero to five from high-order to low-order. Figure 1-7 shows the format of a program code-word. The mapping of program codes into code words is defined in section 3.



MV5359

Figure 1-7. Program Code Word Format

SEGMENTS

A group of memory words may be associated together to form a "segment". This document refers to two classes of segments, "virtual" and "actual". Segments may contain either data or code. Virtual segments are defined by special objects called Data Segment Descriptors (DDs) and Code Segment Descriptors (CSDs). An actual segment is a contiguous group of memory words, and is defined by an unpagged Data Segment Descriptor or a Code Segment Descriptor.

Virtual data segments may be unpagged (represented by one actual segment of arbitrary length) or pagged (subdivided into fixed-size actual segments, with a possibly shorter last page). Code segments are always unpagged.

Pragmatic Notes

Virtual/Actual Segments

In this architecture, both virtual and actual segments are defined by descriptors. This useful distinction exists between pagged and unpagged data descriptors. Except in this context, the adjectives are seldom used.

DESCRIPTORS

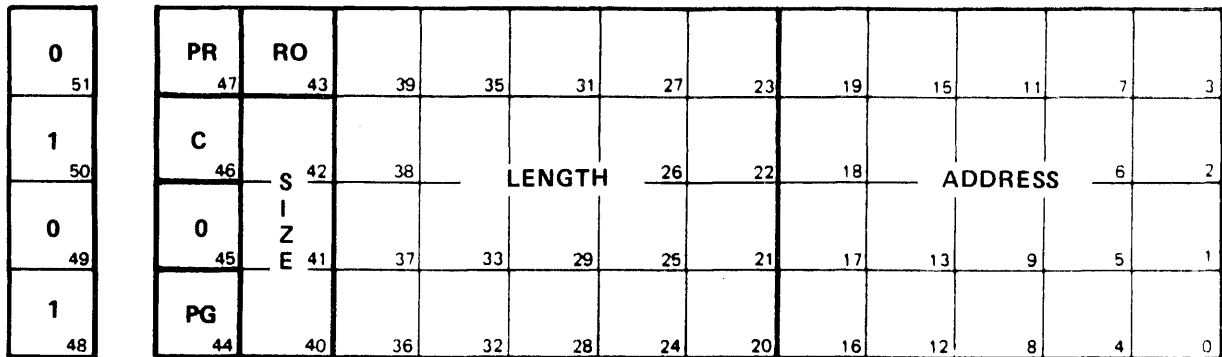
Memory is organized into variable-size segments that are either data segments or program code segments. Data segments are used to implement a program's virtual memory segments and to contain data structures such as stacks and Segment Dictionaries. Program code segments are used to contain the operator sequences of a program. Both data segments and code segments are described by descriptors. Data-segment descriptors and code-segment descriptors are described in their respective subsections.

Data Segment Descriptor

A virtual data-segment is an array of elements, where an element of the array is a single word, a double word pair, or a "sub-word" character requiring 4 or 8 bits. Data-segment descriptor (DD) is the word type that describe data segments. The tag of a DD is 5. Figure 1-8 shows the DD word format.

Memory management of the data-segment utilizes a present bit, an address field, and a copy bit, which distinguishes two classes of DDs: original and copy descriptors. Copies may be either indexed or unindexed. The word "copy" is usually omitted in describing an indexed descriptor, because any indexed DD is a copy.

If the array is present, the address field of a descriptor contains the base memory location of the data-segment. The important distinction between original and copy descriptors exists for absent arrays: the address field of an absent original contains a software-encoded value; the address field of an absent copy contains the nominal address of an original descriptor for the array.



MV5360

present	[47: 1]	Present bit (0 = absent, 1 = present)
copy	[46: 1]	Copy bit (0 = original, 1 = copy)
indexed	[45: 1]	Indexed bit (0: unindexed)
paged	[44: 1]	Paged bit (0 = non-paged, 1 = paged)
read_only	[43: 1]	Read-only bit (0 = read/write, 1 = read-only)
element_size	[42: 3]	The type of array element (0 = single precision, 1 = double-precision, 2 = hex, 4 = EBCDIC, 3,5,6,7 are invalid)
length	[39:20]	The number of elements in the array
address	[19:20]	present: nominal address of the base word of the data-segment; absent copy: nominal address of the associated original descriptor; absent original: encoded by software

Figure 1-8. Data Descriptor Format

The element_size field of the DD specifies the type of array element: single-precision, double-precision, EBCDIC (8-bit), and hex (4-bit). The terms word descriptor and WordDD are used for descriptors whose element_size values are single or double-precision; the terms character descriptor and CharDD are used for descriptors whose element_size values are EBCDIC or hex. The terms SingleDD and DoubleDD are used for WordDDs with element_size single-and double-precision, respectively.

The read_only bit in a DD can be set to prevent use of the DD for write access to the data.

Indexed DDs are actually references, which are described later in this section.

Unindexed DDs have a length field and a paged indicator. The length field contains the number of elements in the array; the number of words in the array may be deduced from element_size and length.

If the paged bit is 0, there is no distinction between the "virtual" segment and an "actual" segment; when the DD is present it describes a single area of contiguous nominal memory addresses. If the paged bit is 1, the virtual-segment is paged, in which case it consists of a number of pages each `page_size` words long (the last page may be shorter). In a present paged descriptor the address field contains the memory address of the first word of a page-table segment, which contains descriptors for the individual pages. These page descriptors are original single-word unpaged DDs. When a paged descriptor is indexed, another level of indexing is performed so that the resulting indexed descriptor references the specified element of the specified page (see the `INDX` operator for a discussion of the indexing of paged descriptors).

Generally, there is one original DD for a segment, and copies are created by most of the operators that fetch DDs to the top of the stack. However, functional operator definition does not require precisely one original DD for each array. Furthermore, an original DD may be brought to the top of the stack without being transformed into a copy (but only the `LODT` and `RDLK` operators perform this action).

Operators that access data through descriptors depend on the following assumptions (the operators produce undefined results if the assumptions are not true):

1. The number of memory words occupied by a single unpaged segment is enough to hold all of the array elements of any unindexed descriptor referencing the array. That is, letting L = length from the data descriptor, W = number of words, and E = element size, then

for E = single, $W = L$;
for E = double, $W = 2*L$;
for E = EBCDIC, $W = (L + 5) \text{ DIV } 6$;
for E = hex, $W = (L + 11) \text{ DIV } 12$.

2. The words directly before and after the actual segment have odd tags.

The two ways of determining the boundaries of data segments (the length in the unindexed descriptor and the odd-tagged words at the actual segment boundaries) are used by the operator set as follows:

1. The indexing operators use the unindexed descriptor length.
2. The operators that fetch and store data through indexed descriptors (other than the pointer operators) make no check at all (the previous index operation's check is trusted).
3. Set membership tables and translation tables are not checked.
4. The word transfer overwrite operators make no check.
5. The character reverse-skip edit operators can check the index in the pointer.
6. All other pointer operators use the odd-tagged boundary words.

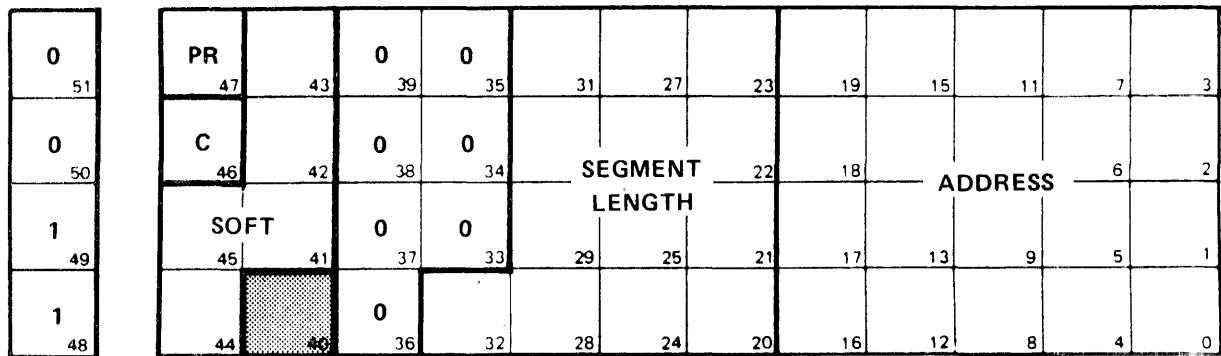
This specification allows for the following inconsistency: an unindexed descriptor with an element-size of hex or EBCDIC may not be indexed beyond the length specified in the descriptor, but data may be accessed beyond this limit, up to the next word boundary, by pointer operators.

Code Segment Descriptor

A code-segment is an array of program code words referenced by a code-segment descriptor. Figure 1-9 shows the format of the code-segment descriptor. The tag value of a code-segment descriptor is 3.

Memory management utilizes a present bit, copy bit, and the address field. The interpretation of these fields is the same as for a data descriptor. (For a present code-segment, the address field contains the base memory location of the segment. For an absent code-segment, the address field in an original contains a software-encoded value, while the address field in an absent copy points to an original.) Copy code-segment descriptors are generated and used only as an interrupt parameter, when an attempt is made to execute code from an absent segment.

The `seg_length` field contains the number of code words in the segment. Code segments may not be paged.



MV5361

present	[47: 1]	Present bit (0 = absent, 1 = present)
copy	[46: 1]	Copy bit (0 = original, 1 = copy)
	[45: 5]	Reserved for Software
	[39: 7]	Must be zero
<code>seg_length</code>	[32:13]	The number of code words in the segment
address	[19:20]	present: nominal address of the base word of the data-segment; absent copy: nominal address of the associated original descriptor; absent original: encoded by software

Figure 1-9. Code Segment Descriptor Format

Stack Segments

A stack is a particular use of an actual segment, used to define program environments and maintain processing history. Stacks are referred to by stack numbers; a stack number is an index on a data descriptor called the Stack-Vector Descriptor (SVD). The SVD is a present unpagged unindexed SingleDD; it defines an actual segment that contains a stack descriptor for each stack in the system. A stack descriptor is an unpagged unindexed SingleDD. The allowable range of stack numbers is {0 to $\min(4095, \text{SVD.length}-1)$ }. The SVD is located at a nominal address calculated as $D[0] + 2$; that is, its address-couple is (0,2).

Paged Segments

A virtual-segment is associated with one or more actual segments. To an unpagged virtual-segment there corresponds exactly one actual segment; in this case the virtual-actual distinction can be considered redundant, and the adjective is often omitted.

A paged virtual-segment is represented by several actual segments, called pages. The Data Segment Descriptor is marked "paged"; it defines an actual segment called a "page table" containing one Data Segment Descriptor for each page. Each page DD is marked "unpaged" and "original"; it defines the actual segment for that page. All pages are page__size words long, except the last page in a virtual-segment, which may be shorter.

REFERENCES

There are several reference data types: 1) normal and 2) stuffed indirect reference words (NIRWs and SIRWs), which point to locations in activation records, 3) indexed data descriptors (IndexedDDs), which point to individual elements of data segments, and 4) program control words (PCWs), which provide code stream pointers and initial execution state values. Reference data types are described in the following paragraphs.

Address Couples

An address couple is a pair of indexes (Lambda, Delta) that reference a word in the current addressing environment: Lambda specifies a lexical level in the current addressing environment, and Delta is the offset to the referenced location from the base of the activation record at level Lambda. Note that the location referenced by an address-couple may vary according to the addressing environment at the time of its interpretation, depending on the value of D[LL] and on the values of the MSCWs in the lexical chain.

Fixed-Fence Address Couples

Address couples in Normal Indirect Reference Words (NIRWs) and in several operators are encoded in 16 bits with a 4-bit lambda value in field [15:4] and a 12-bit delta value in field [11:12].

Variable-Fence Address Couples

Address couples in NAMC and VALC operators are encoded in 14 bits with a "variable fence" between Lambda and Delta. Taking advantage of the fact that Lambda must be less than or equal to LL, the number of high-order bits interpreted as the Lambda value varies with the value of LL at evaluation time. The remaining low-order bits are interpreted as the Delta value. Table 1-1 gives explicit ranges.

Table 1-1. Address Couple Fence Decoding

LL range	Bits left of fence	Lambda range	Delta range
{0 to 3}	2	{0 to LL}	{0 to 2**12 - 1}
{4 to 7}	3	{0 to LL}	{0 to 2**11 - 1}
{8 to 15}	4	{0 to LL}	{0 to 2**10 - 1}

The Lambda value is the reverse of the bits to the left of the fence, and the Delta value is taken from the bits to the right of the fence. Following are examples of address-couple interpretation. Each pair is the same address-couple representation, but notice the effect of the dynamic fence, indicated by the colon (:):

- 1 a) at LL = 2, 10:000000010011 ≥ (1,19)
- b) at LL = 13, 1000:0000010011 ≥ (1,19)

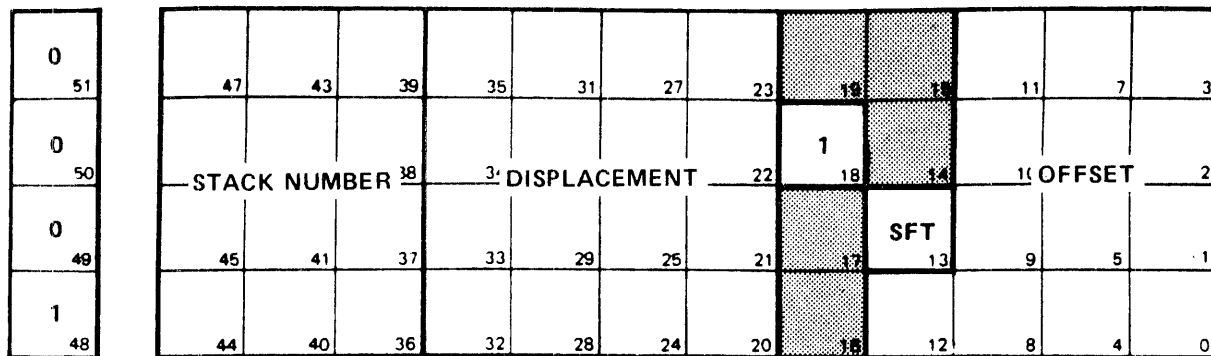
- 2 a) at LL = 5, 101:00001000000 ≥ (5,64)
- b) at LL = 3, 10:100001000000 ≥ (1,2112)

An SIRW has three fields: `stack__number`, `displacement`, and `offset`. The memory location referenced by an SIRW is computed by the following function:

$$\text{BaseAddress}(\text{stack_number}) + \text{displacement} + \text{offset},$$

where `BaseAddress(stack__number)` is the address of the base of the stack whose number is contained in the `stack__number` field. `BaseAddress + displacement` yields the address of the base word of an activation record, and `offset` is the index of the referenced location relative to that base.

Note that `stack__number` and `displacement` constitute a Lexical Link, and `offset` corresponds to `NIRW.delta`.



MV5363

- | | | |
|----------------------------|---------|--|
| <code>stack__number</code> | [47:12] | The identification of the stack containing the referenced location |
| <code>displacement</code> | [35:16] | The displacement from the base of the stack to the base of the activation record |
| | [18: 1] | 1: denotes SIRW |
| | [13: 1] | Reserved for software use |
| <code>offset</code> | [12:13] | The offset from the base of the activation record to the referenced location |

Figure 1-11. SIRW Word Format

IndexedDD (Indexed Data Descriptor)

IndexedDDs reference an individual element of a data-segment. The interpretation of an IndexedDD is a variation of the interpretation of a DD (it is an indexed copy DD). The tag of an IndexedDD is 5, and its Indexed bit is 1.

An IndexedDD's `read__only`, `element__size`, `present` and `address` field are interpreted identically to those of an unindexed copy DD. (An IndexedDD must be a copy, and it cannot be paged.) An indexed word descriptor is called an IndexedWordDD or (more specifically) an IndexedSingleDD or IndexedDoubleDD (see Figure 1-12). An indexed character descriptor is called a Pointer (see Figure 1-13).

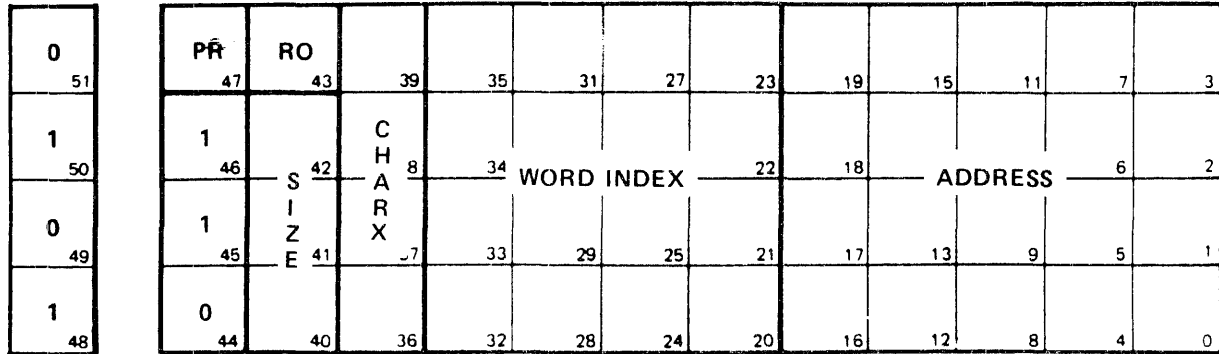
The interpretation of the index to the referenced element depends on `element__size`. For IndexedWordDDs, the index field is the word index from the base of the array or page to the single-precision word or to the first word of the double-precision word pair. For Pointers, the index field consists of two subfields: `word__index`, the index from the base of the array or page to the word containing the referenced character, and `char__index`, the character index within the word. For EBCDIC Pointers, `char__index` must be in the range {0 to 5}, and for hex Pointers, it must be in the range {0 to 11}. `Char__index 0` is the highest-order character in the word.

0		PR	RO												
51		47	43	39	35	31	27	23	19	15	11	7	3		
1		1	0												
50		46	42	38	3	INDEX	26	22	18	ADDRESS			6	2	
0		1	0												
49		45	41	37	33	29	25	21	17	13	9	5	1		
1		0	D												
48		44	40	36	32	28	24	20	16	12	8	4	0		

MV5364

present	[47: 1]	Present bit (0 = absent, 1 = present)
copy	[46: 1]	Copy bit (1: [indexed] copy)
indexed	[45: 1]	Indexed bit (1: indexed)
	[44: 1]	Must be zero: the effect of a 1 in this bit is undefined.
read_only	[43: 1]	Read-only bit (0 = read/write, 1 = read-only)
element_size	[42: 3]	The type of array element (0 = single-precision; 1 = double-precision). (2,4 denote Pointer; 3,5,6,7 are invalid.)
index	[39:20]	The word index from the base of the array to the referenced item
address	[19:20]	present: the nominal address of the base word of the array; absent: the nominal address of the associated original data descriptor

Figure 1-12. Indexed Word Data Descriptor Format



MV5365

- | | | |
|--------------|---------|---|
| present | [47: 1] | Present bit (0 = absent, 1 = present) |
| copy | [46: 1] | Copy bit (1: [indexed] copy) |
| indexed | [45: 1] | Indexed bit (1: indexed) |
| | [44: 1] | Must be zero: the effect of a 1 in this bit is undefined. |
| read_only | [43: 1] | Read-only bit (0 = read/write, 1 = read-only) |
| element_size | [42: 3] | The type of array element (2 = hex, 4 = EBCDIC). (0,1 denote IndexedWordDD; 3,5,6,7 are invalid) |
| char_index | [39: 4] | The index within the word of the referenced character |
| word_index | [35:16] | The index from the base of the array to the word containing the referenced character |
| address | [19:20] | present: the nominal memory address of the base word of the array; absent: the nominal memory address of the associated original data descriptor. |

Figure 1-13. Indexed Character Data Descriptor (Pointer) Format

PCW (Program Control Word)

A program control word (PCW) contains the initial code-stream pointer and execution state values associated with an activation record in the program. A PCW is the means by which the execution state is established for an activation record when it is entered (when it becomes the topmost activation record). The tag of a PCW is 7. Figure 1-14 shows the format of a PCW word.

The PCW code-stream pointer consists of the fields sdll, sdi, pwi, and psi. The PCW lex_level field indicates the lexical level at which the activation record is to run. The control_state attribute specifies execution in normal or control state.

STACK LINKAGE WORDS

There are three data types utilized for stack linkage. An MSCW (mark stack control word) and an RCW (return control word) are the two words that contain stack linkage values for an activation record in the addressing environment. A TSCW (top-of-stack control word) is used to preserve processor state in an inactive stack (a stack to which no processor is bound).



MV5367

stack__number	[47:12]	Identifies the stack containing the activation record to which the lexical link points
displacement	[35:16]	The displacement from the base of the stack__number stack to the base of the activation record to which the lexical link points
entered	[18: 1]	The entered bit (0 = inactive, 1 = entered)
lex__level	[17: 4]	The lexical level at which the activation record runs
history__link	[13:14]	The displacement down the stack from the MSCW to the base of the prior MSCW

Figure 1-15. Mark Stack Control Word (MSCW) Format

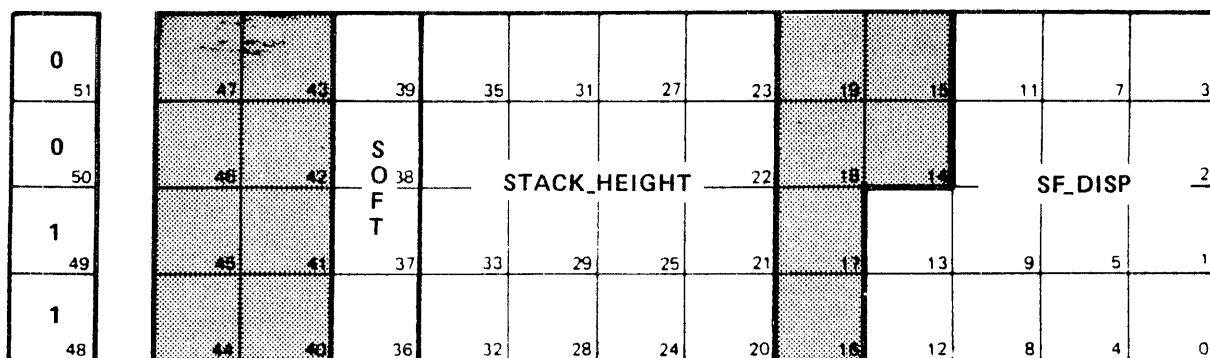
RCW (Return Control Word)

An RCW is stored at the base location plus one of an activation record, immediately above the MSCW. The RCW is associated with MSCW.history__link and preserves code-stream pointer and execution state to be restored when the activation record is exited and execution is resumed in the prior topmost activation record on the historical chain. The tag of an RCW is 3. Figure 1-16 shows the format of a RCW.

The RCW code-stream pointer consists of the fields sdll, sdi, pwi, and psi. Preserved execution state consists of control__state (the CS Boolean), the processor state Booleans defined in "General Boolean Accumulators", and lex__level (the lexical level of the prior topmost activation record on the historical chain defined by MSCW.history__link). A restart indicator in the RCW may condition restart state for the first operator in the designated code-stream.

The block__exit bit indicates whether or not an interrupt is to be generated when the activation record is deallocated. This bit is always initialized to 0 by the enter operators. It can be set to 1 by software, in which case it must be reset to 0 by software before an EXIT or RETN operator can deallocate the activation record associated with this RCW.

The exit__opt field can be used to retain two bits of state at procedure entry to enable an optimization at procedure exit; see the definitions of ENTR and EXIT.



MV5369

	[39: 4]	Reserved for software use
stack_height	[35:16]	Displacement S-BOSR
SF_disp	[13:14]	Displacement S-F

Figure 1-17. Top Of Stack Control Word (TSCW) Format

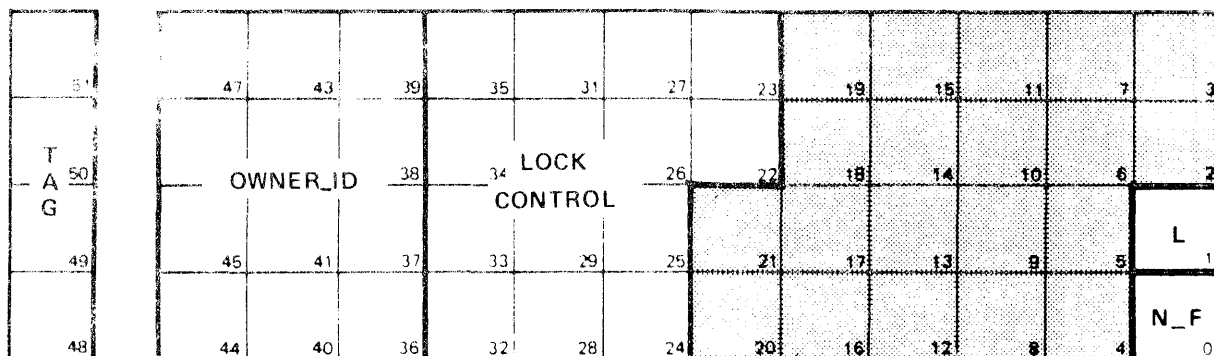
INTERLOCKS

The data type interlock and its associated operators provide a mechanism for processes to effect mutual exclusion of code regions. The operators accept either 0 or 3 in the tag of an interlock, but always set the tag to 3. Figure 1-18 shows an Interlock word format.

Each valid interlock status is named and defined in the following table, which also characterizes the contents of the owner_id and lock_control fields.

:Free				
:Locked__Uncontended				
:Busy				
:Locked__Contended	locked__	not__free__	owner__id	lock__control
state	bit	bit		
Free	0	0	0	arbitrary
Locked__Uncontended	1	1	owner stack number	arbitrary
Busy	0	1	busy stack number	0
Locked__Contended	0	1	owner stack number	non-zero

The combination locked__bit = 1 and not__free__bit = 0 is undefined and is not generated by the interlock operators. In this architecture, operators do not create Locked__Contended interlocks, but they distinguish the Locked__Contended from the Busy state.



MV5370

tag		accepted: 0 or 3, created: 3
owner__id	[47:12]	Stack number of lock owner (or busy contender)
lock__control	[35:14]	Reserved for software use
locked__bit	[1: 1]	(0: state is not Locked__Uncontended, 1: state is Locked__Uncontended)
not__free__bit	[0: 1]	(0: state is "Free", 1: state is not "Free")

Figure 1-18. Interlock Control word Format

TAGS 8-15

This architecture is defined with 4-bit tags, but only tag values 0-7 are used to define data types. Tag values 8-15 are reserved for later Levels of this architecture specification.

The architecture specified in this document permits an implementation to handle the high tag values in either of the following ways:

1. Ignore the fourth bit (implement 3-bit tags).
2. Treat tags 8-15 as defining arbitrary bit-vector data types.

If option 1 is chosen, the entire document is to be read as specifying 3-bit tags and 51-bit words.

If option 2 is chosen, the following conventions apply:

1. Words with tag = 8, 10, 12, or 14 are treated as are those with tag = 6. That is, such words can be fetched with the LOAD operator, stored with the overwrite operators, and stored over with normal store operators; they may be used as computational but not arithmetic arguments.
2. Words with tag = 9, 11, 13, or 15 are treated as are those with tag = 3. That is, only the LODT and RDLK operators can fetch them and only the overwrite and transfer-words-overwrite operators can store or store over them. If on the expression stack, they may be used as stack arguments for those operators that accept an argument of "any" type.

(Of course, these words are not treated as synonyms for words with tag = 6 or tag = 3; each different tag value is unique to such operators as RTAG, SAME, and SRCH, and in such assertions as "the tag of an RCW must be 3".)

This section contains the only discussion of tag values 8-15 in the document. Section 3 and the appendixes make no reference to such tag values.

Pragmatic Notes

Software Conventions for the Fourth Tag Bit

Given the two options defined in this section, and given the specification of the remaining operators, the following software convention should avoid any confusion of the four-bit tags (with the possible exception of tag-transfer input/output): always force tag bit 3 = 1 in the mask argument of the SRCH operator; force tag bit 3 = 0 in all other contexts.

SECTION 2

STACK CONCEPT AND PROCESSOR STATE

GENERAL INFORMATION

This section discusses concepts associated with the process model implemented by the architecture. It is intended primarily to introduce the stack structure and control mechanism required by the operator set described in section 3 of this manual.

The hardware processor separates program functions into operators and operands. Program controller logic directs the fetching and execution of operator codes. Stack controller logic directs activity in the stack mechanism. Built-in synchronization circuits are required in the hardware of the system, to synchronize the operations of the program and stack controller mechanisms. The stack concept implemented in the system provides features necessary for automatic interrupt handling control logic, reentrant code programming techniques, and virtual memory operations.

Stack control functions include "common actions", which are described in detail in section 3 of this manual. This section describes the structure and linkages within the stack. Processor control logic is briefly described because system initialization functions utilize stack structure and require the automatic synchronization that exists between the stack and program controllers.

STACKS

The machine is oriented around the concept of a segmented memory and specially treated segments called stacks. The processor uses an expression stack; most operators take their arguments from the top of the stack and leave their results on top of the stack.

A stack can be considered the instantaneous state of a process. The stack contains a historical record of all procedures (blocks) that have been entered and not yet exited. A system can utilize many stacks, with a processor being assigned to one stack at a time (thus providing multiprogramming). A system can be equipped with more than one processor (thus providing multiprocessing); at any moment, each processor is bound to a different stack.

The data addressing space of the executing process is mapped into its stack, other stacks linked to it, and data segments referenced by descriptors contained in its stack structure.

CODE SEGMENT DICTIONARIES

Executable code is contained in segments defined by descriptors that occur in special segments called Code Segment Dictionaries. A code-segment dictionary can be considered the instantaneous state of a program.

ADDRESSING GRANULARITY

The unit of addressing is the word, a memory unit comprising 48 data bits and a 4-bit tag. Operators exist that can deal with parts of words, but memory is addressed and accessed in whole words. The term "item" is more general than "word"; an item may contain more than one word.

PROGRAM ADDRESSING ENVIRONMENT

The addressing environment of the executing code-stream consists of a set of local addressing spaces contained within stacks. These are called activation records (referred to as lexical regions elsewhere), and each consists of a set of variables addressed by an index relative to the base of the activation record. An activation record can be considered the instantaneous state of a procedure or block.

Activation records are managed by use of two linked lists: the historical chain and the current lexical chain. Both links are contained in a structure called a Mark Stack Control Word (MSCW), located at the base of the activation record; links to an activation record always address the base word.

The historical chain is a chronologically ordered list that consists of History Links connecting the MSCW of each activation record to that of its initiating activation record. An historical chain pointer to the most recently created MSCW is all that is required to access any activation record in the stack.

Activation records are created on the top of the stack by a sequence of operations:

1. A "mark stack" operation defines the base location for the incipient activation record, creating a new MSCW at the head of the historical chain.
2. A reference to the code for the new procedure is placed on the stack, followed by any parameters for the procedure.
3. An "enter" operation is performed. The new activation record is now linked into the lexical chain; the addressing environment and code-stream for the new procedure are established.

Prior to the "enter" operation, the MSCW is marked "inactive"; a historical linkage but not a lexical linkage exists and the incipient activation record is actually part of the expression stack of the initiating process. After step 3, the MSCW is marked "entered" and the new activation record formally exists.

(The activation record at the head of the lexical chain is deleted from the stack by the "exit" operation; the addressing environment and code stream for the historically prior activation record are reinstated.)

A History Link is represented as an integer displacement from an MSCW to its immediate predecessor MSCW in the stack.

The Lexical Link of an activation record points to the base of the immediately global addressing space, which is defined in terms of the static program structure as follows: if B0 and B1 are blocks in the program, B0 immediately contains B1, and activation records AR0 and AR1 correspond to B0 and B1, then AR0 is the immediately global addressing space of AR1.

The current addressing environment is the set of activation records addressed by the lexical chain whose head is the activation record bound to the executing code-stream. This activation record is called the topmost activation record and is the activation record that contains the first entered MSCW on the historical chain. The position of an activation record in the lexical chain defines its lexical level. The lexical level of the topmost activation record is defined to be LL; there are LL+1 activation records in the current environment. Lexical level 0 defines the end of the chain and denotes the most global addressing space. A lexical chain pointer to the topmost activation record is required for accessing the current environment.

A Lexical Link is a (stack number, displacement) couple. The stack number is an index that uniquely identifies a stack; the displacement is a relative position within the stack of the base of the activation record.

History Links always point to an MSCW in the same stack, but Lexical Links may point to an activation record in another stack. Therefore, an addressing environment may be mapped into a tree structure.

A general reference to an item in the current environment takes the form of a (Lambda, Delta) address couple, where Lambda is a lexical level and Delta is an offset to the referenced item from the base of the activation record at level Lambda. Address couples are the means of addressing locations in the current environment.

Processor management of the activation records in the stack utilizes the following registers:

- F: The nominal address of the most recent MSCW in the stack. F defines the head of the historical chain: all activation records and incipient activation records for the process are accessible by following History Links from F.
- LL: The lexical level of the topmost activation record in the current addressing environment – the level at which the processor is running. LL is always in the range $0 \leq LL \leq 15$.
- D[LL]: The D[LL] nominal address of the MSCW at the base of the topmost activation record. D[LL] defines the head of the lexical chain: all activation records in the current addressing environment are accessible by following Lexical Links from D[LL].
- D[0]: The nominal address of the MSCW at the base of the most global activation record.

The stack-vector descriptor is located at address-couple (0,2); the interrupt entry is defined at address-couple (0,3). The stack-vector descriptor and the interrupt entry can be located relative to D[0], at nominal addresses D[0]+2 and D[0]+3, respectively, even when the lexical chain from D[LL] is invalid. Operators that redefine the lexical chain can change the D[0] value.

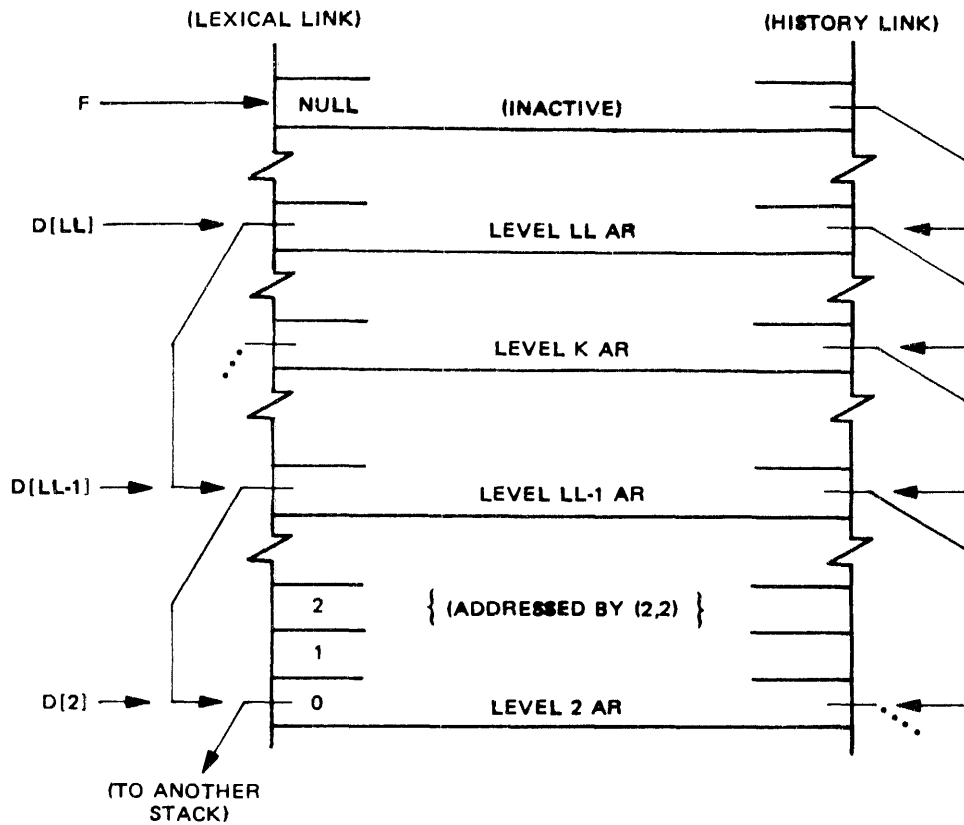
Addressing may be optimized by defining an array of "display" registers maintained such that: :display registers

- D[i]: The D[i] nominal address of the MSCW at the base of the activation record at level i in the current addressing environment, for i in {0 to LL}

Figure 2-1 shows an addressing environment example. Note that the lexical link from the level 2 activation record is to another stack; there could also be a fork in the stack-structure tree above the level 2 activation record. The activation record shown between the level LL and level LL-1 activation records is not linked into the current environment; it is shown as level k. Depending upon the lexical linkage, k might be equal to LL or greater or less.

MEMORY ADDRESSING

A process executing on a hardware processor (or on an extension of such a process into the I/O subsystem) has a program address space of 2^{20} words. Each of these words has a 20-bit nominal address ranging from 0 to $2^{20}-1$. It is this nominal address that occurs in descriptors and state registers; it is often called simply the memory address.



MV5371

Figure 2-1. Addressing environment example

A system may have more than 2^{*20} words of physical memory, in which case the processor can address only part of the whole at any one time. There is a mechanism for mapping the 2^{*20} contiguous nominal addresses into a larger physical memory. The mapping is specified in sections, called environment components; the collection of such components available to a process constitute its environment. The whole of the larger memory may be used by defining several different environments. Each environment can be identified by an environment number ranging from zero to some upper limit; the complete name of each program address is thus the couple

NOTE

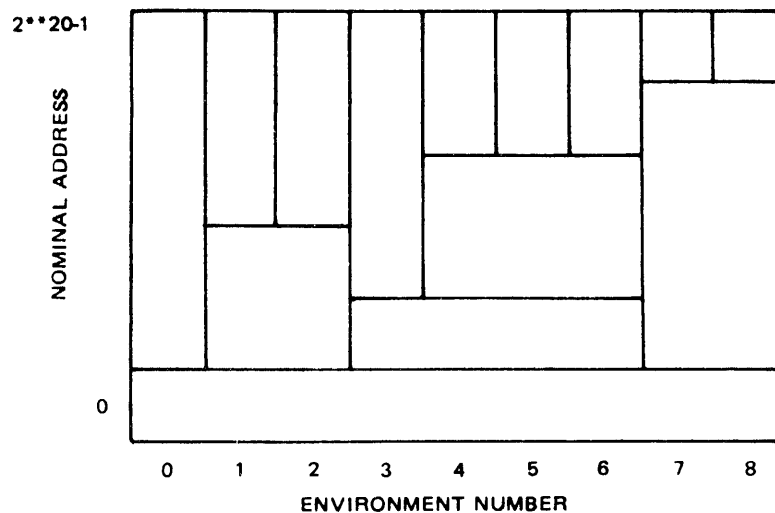
The memory addressing "environment" and the "program address" couple just defined refer only to the memory addressing mechanism discussed in this section. These terms and concepts should not be confused with the "addressing environment" and "address-couple" defined in the previous section and used throughout the document.

The memory available to a given system may be thought of as being addressed by a single continuum of addresses ranging from zero to some upper limit. There is then some implementation defined mapping from this continuum of addresses to the physical addressing mechanism provided by a particular implementation. Note that the continuum may have no explicit representation in either hardware or software, but is used to separate the concerns of nominal (program) address space management from implementation-dependent physical addressing mechanisms. This architecture is concerned with the first

mapping, from nominal address to continuum; the further mapping to physical mechanisms is not specified. "Holes" (subranges of addresses not present or not available) in any of these three levels have no effect on the model.

An environment component is a contiguous subrange of the nominal address space that is mapped onto a contiguous subrange (of the same size) of the continuum. Thus, each program addressing environment is composed of one or more environment components, separated by "fences". Through the mapping, environment components from several different environments may be mapped onto the same subrange of the continuum, creating an "alias" situation where an element of the continuum is addressed by several "names" – (environment number, nominal address) couples.

In order to discuss restrictions on the generality of mapping structures, two models are used to illustrate the ways components from different environments may be identified in the mapping into the continuum. The first is a 2-dimensional diagram (figure 2-2), where the horizontal dimension represents the range of environment numbers present in a system at some time, and the vertical dimension represents the range of nominal addresses (0 to $2^{*}20 - 1$). The horizontal lines represent fences separating environment components within an environment, and the enclosed rectangles represent components that may be shared among different environments. An example diagram appears below.



MV5372

Figure 2-2. Memory Environment Mapping

The second model is a graph, where the nodes represent components in the continuum, and the directed edges connect components to neighboring components at the next higher nominal-address subrange within the same environment. That is, for nodes A and B, the edge

$$A \rightarrow B$$

defines the relation that A and B are in the same environment, B has higher nominal address than A, and there is no other node (component) between A and B. These "adjacent" nodes need not contain the immediate succeeding address; that is, there may be holes in an environment. A path from a node representing the component containing nominal address zero to some terminal (no departing edges) node represents a complete environment.

(The two models are, of course, equivalent: the nodes of the graph correspond to the boxes in the diagram; the edges of the graph correspond to the fences separating the boxes.)

In terms of these models, the following axioms state the requirements of every implementation:

1. All boxes are rectangles.
2. The graph is a single-rooted tree.
3. The mapping preserves order and contiguity of addresses within each component. That is, if x , $x+1$, and y are all addresses within the same environment component, and m is the mapping from the environment into the continuum, then:

$$\begin{aligned} \text{i) } & x < y \iff m(x) < m(y) \\ \text{ii) } & m(x) + 1 = m(x+1) . \end{aligned}$$

4. "Aliasing" of addresses is restricted so that a continuum location has the same nominal address in all environments that share it, and the same continuum address does not occur twice in the same environment: If $e1$ and $e2$ are environment numbers, $a1$ and $a2$ nominal addresses, and M the mapping into the continuum, then:

$$\begin{aligned} \text{iii) } & M(e1,a1) = M(e2,a2) \text{ and } e1 \neq e2 \rightarrow a1 = a2 \\ \text{iv) } & a1 \neq a2 \rightarrow M(e1,a1) \neq M(e1,a2) \end{aligned}$$

A valid implementation may reverse the order of the addresses; that is, reverse the meaning of the edges in the graph and reverse the labeling of the vertical dimension in the diagram.

A valid implementation may impose any combination of the following restrictions:

1. The common component (root of the tree) must be at the low order addresses.
2. Fences may occur only at particular nominal addresses. That is, the size of components may be quantized.
3. The number of fences is limited to some upper bound defined by the implementation. (This limit may be zero, so that the scheme effectively reduces to a traditional single-component memory, with zero the only valid environment number.)
4. Any or all fences may be forced to identical locations in all environments.
5. The fan-out (departing edge count) at each node must be identical to the fan-out for other nodes at the same level (depth) in the tree.
6. All environments must be complete; that is, all environments must be the full $2^{*}20$ words long. This requirement does not mean that there can be no holes in the actual memory space; some addresses in some environments may be unusable.
7. The $D[0]$ value must be in the common component.
8. The stack-vector must reside in the common component.
9. A data or code segment must be entirely contained within one component. The following restrictions are corollaries of this one, except that they also restrict operations by means of a descriptor that spans multiple segments (such as the "M" descriptor, with address 0 and length $2^{*}20-1$, which spans all but one word of an entire environment.)
10. All memory accesses made by a single invocation of a data array operator (via a single descriptor) must occur within the same component, except that the first access of LLLU may be in a different component from the second and subsequent accesses.
11. When a descriptor is both indexed and evaluated in the same operator (as in NXLV, NXVA, NXLN), the base address and the sum (base address + index) must be in the same component.
12. Both word of a double-precision item must be in the same component.

If an implementation selects any of restrictions, 7 through 12, and the restriction is violated, the results are undefined.

The association between an environment component and a component of the continuum is keyed from the Environment Number Register (ENR), which holds the environment number currently in use. The mapping is set up using implementation-defined operations; ENR is loaded by the move-stack (MVST) and set-processor-register (SPRR) operators.

Memory address mapping uses the following register:

ENR: The environment number of the current process

EXPRESSION STACK

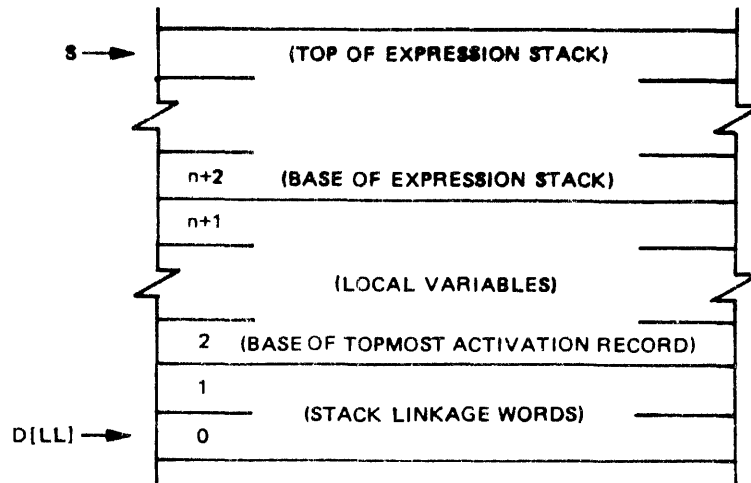
Operator definition assumes the existence of an expression stack. Initial arguments are taken from it, and results are pushed onto it. The expression stack and current addressing environment concepts are merged by treating the topmost activation record as the expression stack.

Variables local to the activation record are initialized by execution of operators that push items onto the expression stack followed by a PUSH operator, which appends the expression stack onto the top of the topmost activation record. This "stack building code" is usually the first operator sequence executed following completion of entry into the activation record. Procedure parameters are treated similar to local variables. They are initialized by execution of operators (just prior to the ENTR operator) that push items onto the expression stack. The ENTR operator, among other functions, appends the expression stack onto the newly created topmost activation record. (See also the description of the PUSH operator in Miscellaneous Operators and the ENTR operator in Processor State Operators.)

The stack that contains the expression stack and topmost activation record is identified by an integer value called Stack Number. The base and limit of the stack are obtained from the stack descriptor (see also Stack Segments and Stack References). There may be activation records in the stack below the topmost one.

The term "expression stack" properly describes that portion of the stack from the top of the topmost activation record to the top of the stack. This architecture does not fully define the boundary between the activation record and the expression stack: a PUSH or ENTR is required to allow items from the expression stack to become addressable as part of the activation record, but items from the activation record as well as from the expression stack can be consumed as top-of-stack arguments.

Figure 2-3 shows a typical configuration of the topmost activation record after completion of stack building code and subsequent operator execution.



MV5373

Figure 2-3. Topmost Activation Record Example

Processor management of the expression stack utilizes the following registers:

- SNR:** The stack number of the stack to which the processor is currently bound.
- S:** The nominal address corresponding to the top word in the expression stack.

The following optimization registers define the boundaries of the stack:

- BOSR:** The nominal base address of the stack containing the expression stack.
- LOSR:** The nominal limit address of the expression stack.

Pragmatic Notes

Top-of-Stack Registers

This architecture does not specify top-of-stack registers, but it does permit an implementation to use an arbitrary number of processor registers to optimize access to top-of-stack values. An implementation satisfies this architecture specification if the ENTR and PUSH operators cause the contents of any top-of-stack optimization registers to be written to memory. More elaborate optimization is possible, by treating the top-of-stack registers as holding cached values for the corresponding memory words. The S register defines the top-of-stack address as though all stack values were in memory.

EXECUTABLE CODE STREAMS

Variable length operator sequences are stored in arrays of program code words called code segments. Each program code-word contains six 8-bit containers called syllables. (The mapping of operators into syllables is specified in the Section 3 and Appendix B of this manual.)

Each code-segment is referenced indirectly by a descriptor, called a code-segment descriptor (see Code Segment Descriptor). Code-segment descriptors for a program are collected in an array called a Code Segment Dictionary.

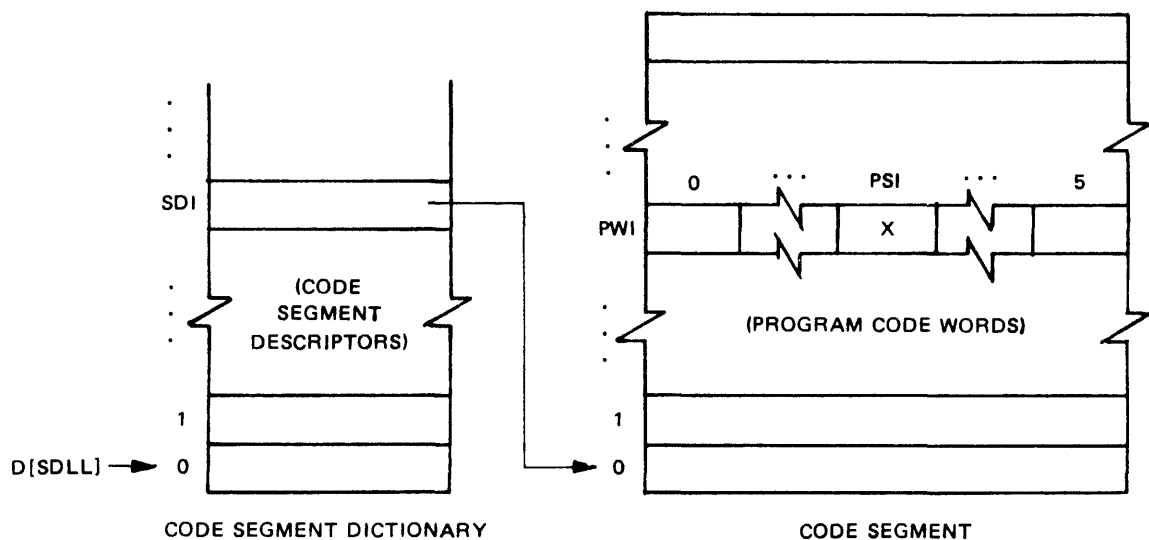
The term "code-stream pointer" is used to describe a reference to the entry point of an operator sequence in a code-segment. A code-stream pointer consists of the following components:

An address-couple (SDLL, SDI) references the code-segment descriptor. SDLL is the Code Segment Dictionary lexical level (it is usually the case that a user program Code Segment Dictionary is the level 1 activation record in its addressing environment, and the operating system Code Segment Dictionary is at level 0). SDI is the Code Segment Dictionary index to the code-segment descriptor relative to the base of the specified Code Segment Dictionary. The entry point in the code-segment is indicated by PWI, the program word index relative to the base of the code-segment, and PSI, the program syllable index within that word.

The processor code-stream pointer consists of the following component registers:

- SDLL: The lexical level (0 or 1) at which the current Code Segment Dictionary is addressed.
- SDI: The index in the Code Segment Dictionary to the current code-segment descriptor.
- PWI: the index in the code-segment to the code-word containing the next operator.
- PSI: The index in the code-word to the next operator syllable.

Figure 2-4 illustrates the processor code-stream pointer.



MV5374

Figure 2-4. Processor Code Stream Pointer

Processor state also includes a Boolean attribute of the executing code-stream:

- CS: If CS is set (control state), maskable external interrupts are disabled. If it is reset (normal state), they are enabled and may occur between operator executions.

GENERAL BOOLEAN ACCUMULATORS

Processor state includes several Boolean accumulators that are used by several operator groups. Their use and definition are discussed in section 3.

TFFF: The true false flip-flop.
OFFF: The overflow flip-flop.
EXTF: The external sign flip-flop.
FLTF: The float flip-flop.

MISCELLANEOUS PROCESSOR STATE

Processor state includes the following:

page__size: Data segments may be subdivided into fixed-size pages. Page__size is the length in words of such pages; its value is a constant in a level of the architecture, for Level Alpha page__size = 256.

Halt: If the Halt Boolean is true, processor execution will stop upon execution of a HALT (conditional processor halt). If it is false, a HALT is treated as a NOOP (no operation).

Interrupt__Count: A counter incremented once at each interrupt attempt; the counter may be set to zero by the ZIC operator. If Interrupt__Count is incremented beyond 3, the processor superhalts.

TOD: The time of day clock, with values in 2.4-microsecond units.

Running__Indicator: The running indicator is a Boolean that is set true by the RUNI operator and set false automatically by the processor if an interval of four seconds elapses since RUNI invocation. If the indicator is reset, a Run Timeout (unmasked external) interrupt is generated.

Interval__Timer: The Interval__Timer is armed and set by the SINT operator and decremented at intervals of 512 microseconds. If the timer counts to zero or is specifically set to zero, an Interval Timer (external) interrupt is generated. Any external interrupt causes the Interval__Timer to be disarmed.

proc: The processor identification state, composed of:

proc__id: The processor identification number

serial__number: The system serial number

factory__release__level: The Engineering Release Lever(ERL)

field__modification__level: The field rework Level

E-mode__level: The architecture level:
4 "01" Alpha

E-mode__features: (reserved)

machine__type:	Machine series type id: 4 "02" B5900 4 "03" A9 4 "05" B7900
page__size__indicator:	page size = 256
microcode__version:	microcode version indicator

Pragmatic Notes

Processor Identification State

All the processor identification state is constant: Proc__id uniquely identifies individual processors in a multiprocessor system; it is typically established when the system is installed. Microcode__version is a feature of processors implemented with loadable control stores; it is supplied by the microcode itself. Unit__id may be used to provide such data as serial number and manufacturing or modification level. The manufacturing organizations determine the structure and content of unit__id and the mechanism by which it is supplied; these matters are not specified in this manual. E-mode__level, E__mode__features, page__size, and page__size__indicator can be provided either by hardware or microcode, depending upon the implementation.

PROCESSOR STATE COMPONENT SIZES

This subsection summarizes characteristics of processor state components and gives their "container size". Processor state described here is also described elsewhere in this manual, where processor state affects particular system functions. The initial values of processor state components, required to start a system into operation, are given subsequently in the System Control subsection.

The "container size" for each component is the number of bits required to contain the maximum allowable value of the component (components containing a single Boolean value require one bit). A correct architecture implementation is required to use the full container sizes for all components except the environment number (ENR). For ENR, an implementation may use any container size from 0 to 12 bits. SPRR must invoke aISX action if an attempt is made to store a value too large for the container. MVST must generate an Invalid Argument Value interrupt if the ENR value in its argument is too large, except that if the ENR width is zero, MVST may be defined as having a 12-bit integer argument (stack number only) with aISX action.

Addressing Environment State:

F	(20 bits):	The nominal address of the most recently created MSCW in the stack.
LL	(4 bits):	The lexical level of the topmost activation record in the current addressing environment – the level at which the processor is running.
D[LL]	(20 bits):	The nominal address of the MSCW at the base of the topmost activation record.
D[0]	(20 bits):	The nominal address of the MSCW at the base of the most global activation record.

Memory Addressing State:

ENR	(0-12 bits):	The environment number used to map nominal addresses into elements of the memory address continuum.
-----	--------------	---

Expression Stack State:

SNR	(12 bits):	The stack__number of the stack containing the expression stack.
S	(20 bits):	The nominal address of the top word in the expression stack.
BOSR	(20 bits):	The nominal base address of the stack containing the expression stack.
LOSR	(20 bits):	The nominal limit address of the expression stack.

Code Stream Pointer:

SDLL	(1 bit):	The lexical level at which the current Code-Segment Dictionary is addressed.
SDI	(13 bits):	The index in the Code-Segment dictionary to the current code-segment descriptor.
PWI	(13 bits):	The index in the code-segment to the code-word containing the next operator.
PSI	(3 bits):	The index in the code-word to the next operator syllable.

Execution State Attributes:

CS	(Boolean):	While CS is true (control state), maskable external interrupts are disabled. When it is false (normal state), they are enabled and may occur between operator executions.
----	------------	---

General Boolean Accumulators:

TFFF	(Boolean):	The true false flip-flop.
OFFF	(Boolean):	The overflow flip-flop.
EXTF	(Boolean):	The external sign flip-flop.
FLTF	(Boolean):	The float flip-flop.

Miscellaneous State:

Halt (Boolean):	If Halt is true, processor execution will stop upon execution of a HALT (conditional processor halt). If it is false, a HALT is treated as a NOOP (no operation). The state of the Boolean is set by an agency external to the architecture processor (ultimately, by a human operator).
Interrupt__Count (2 bits):	Interrupt-entry counter.
TOD (36 bits):	The time-of-day clock, which is incremented once every 2.4 microseconds whenever the system is functional (even when the processor is halted.)
Running__Indicator (Boolean):	The running indicator. The effect on the running indicator of halting the processor is implementation-defined.

Interval__Timer (11 bits): The interval timer, with values in 512-microsecond units. The effect on the interval__timer of halting the processor is implementation-defined.

(Boolean): "Interval__Timer is armed" state.

proc (95 bits): The processor identification, composed of:

proc__id	(3 bits).
unit__id	(32 bits).
E-mode__level	(4 bits).
E-mode__features	(4 bits).
machine__type	(8 bits).
page__size__indicator	(4 bits).
microcode__version	(40 bits).

Processor identification state is presented here to standardize its data formats and naming-conventions. However, the uses of processor identification state, including the methods of acquiring, updating, and accessing its values, are implementation-defined. Any errors that may be caused by processor state data-handling operations are also implementation-defined. (See the descriptions of the RIPS, WIPS, REMC, and WEMC operators in section 3, especially the pragmatic note to the RIPS operator description.)

SYSTEM CONTROL

The following are system control state used by systems.

HALT A Halt may be initiated by the human operator by means of the Maintenance Subsystem interface, by the processor by means of the STOP or HALT operator or by an external interface signal. The processor will not initiate any more operators and will stop when the currently executing operator or operators are completed.

The MLIP is made aware that the processor is halted and responds by behaving as if the Suspend all Queues flag were true. The maintenance processor will enforce a minimum two 2. second delay after the Halt occurs before a Continue, a Clear, or a Start will be recognized. During this delay the MLIP may continue to run, allowing I/Os other than Test Waits to complete.

Pragmatic Notes

Timer Functions in Halted Processors

This architecture does not fully specify the behavior of the Running__Indicator and the Interval__Timer when a processor is halted and continued, because these matters do not concern normal operation. However, it should be noted that diagnostic use of the HALT and STOP operators is inconvenient if the processor immediately interrupts when continued; this is especially true for run__timeout, which software may treat as an error.

- CONTINUE** After a Halt the processor may be restarted by a Continue. Execution will resume at the operator that would have been initiated had the system not been halted.
- CLEAR** The Clear function may be executed by the human operator by means of the Maintenance Subsystem or by an external interface signal. If the system is not halted, a Halt operation is first performed. The internal processor state is set so as to permit normal execution, including clearing the superhalt counter.
- In response to a Clear, the MLIP will reset the Suspend all Queues flag, clear its reference to the Error IOCB, and broadcast a "master clear" on the MLI PORTS. It will not answer either processor or DLP requests until the master clear handshake is complete.
- START** The Start function initializes the processor state and begins processor execution. It may be initiated by the human operator by means of the Maintenance Subsystem or by an external interface. Start has effect only if the processor is in a halted state; it performs a Clear operation, except that a Halt is not first done. Start then initializes the following state:

F	0
LL	0
D[LL]	0 (or value of a parameter from external interface)
SNR	0
S	4"4000"
BOSR	0
LOSR	0
ENR	0
SDLL	0
SDI	4
PWI	0
PSI	0
CS	0 (False)

After the state is initialized, control is transferred to code in memory by simulating an interrupt. (Current implementations use Invalid Address as the interrupt literal).

If start is initiated by an external interface, D[0] is set to the value specified by the external agency.

NOTE

The system maintenance processor provides a means to load a code file to memory address 0 prior to a manual Start operation.

PROGRAMMING RESTRICTIONS DUE TO HIDDEN STATE

In addition to explicit architecture state, which is accessible directly through various operators, a processor maintains other state to facilitate efficient execution. For the most part, this hidden state is not described in this document. However, some restrictions on software are necessary to ensure that hidden processor state is consistent with visible processor state (especially memory contents).

This architecture defines enter, exit, and branch operators as the only mechanisms to alter the sequential execution of the code-stream. The result is undefined if the program changes a code-segment descriptor or the contents of the code-segment while any processor is executing code from the referenced segment. (The processor is free to capture the code-segment base address and limit, one or more words of code, and so forth.) The result is also undefined of changing D[0] or ENR by means of the SPRR or MVST operator when the code-stream pointer would designate different code in the new and old addressing environments.

The effect of changing the stack descriptor for an active stack (a stack to which an active processor is bound) is undefined.

The effect of changing the stack-vector descriptor is undefined until D[0] is subsequently assigned a value by SPRR, ENTR, EXIT, RETN, or MVST (which can change the continuum element associated with the nominal address in D[0]). An implementation may so restrict the nominal address mapping and D[0] values that D[0] maps to the same continuum element in all environments, in which case the implementation may capture the stack-vector descriptor whenever D[0] is altered. An implementation may further restrict the nominal address mapping and stack-vector address so that the continuum elements in the stack-vector are the same in all environments, in which case the mapping of the stack-vector base address onto the continuum may be captured whenever D[0] is altered. If an implementation includes such restriction, and the value in D[0] or the address in the stack-vector descriptor violates the restriction, the result is undefined.

If a nonpresent copy descriptor is brought to the expression stack and then modified in the present, copy, or address field, the modified descriptor must be explicitly returned to memory prior to being used as the reference input to an operator. (The memory write can be effected by an overwrite operation, an explicit PUSH, or an implicit push via ENTR.) In other words, the result is undefined if a descriptor is brought to the expression stack with copy = 1 and present = 0, the present or copy or address field is modified, and then the descriptor is consumed as a reference.

The effect is undefined of changing the lexical linkage for any activation record currently in the addressing environment of a processor.

The result is undefined of any fetch or store operation in the current stack (whether by means of an address-couple, SIRW, DD, or absolute-address) above the current upper-bound for stack addressing. That bound is moved upward to the S setting by an explicit PUSH, the implicit push of an enter operator, or a move-stack operator. It is moved downward by an EXIT operator, or by a DLET operator that moves S below the most recent PUSH setting.

SECTION 3

OPERATOR SET AND COMMON ACTIONS

GENERAL INFORMATION

This section defines the architecture operator set and common actions. Operators and common actions are presented in functional category order. General information about each functional category is given before the specific functions within the category are described.

Operators usually execute upon operands present in the stack structure described in section 2 of this manual. When the result of an operator is data, that data is the most recent entity in the expression stack, at the top of the activation record. Changes to the expression stack that result from the normal conclusion of an operator sequence are given as part of the operator description. Changes to the expression stack that result from abnormal conclusion of an operator sequence are described in section 4 of this manual, along with other system interrupts.

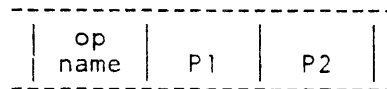
Operators and Code Streams

An operator is composed of an opcode and up to four parameters. Opcodes are typically one syllable, and parameters, if any, are in the syllables following the opcode. Opcode and parameter mapping into syllables varies; operator formats are explicitly specified in this section and in Appendix C. (The term parameter is used in operator descriptions to describe items from the code-stream; the term argument is used for items from the stack.)

A code-stream is considered to be a sequence of syllables fetched without regard to word boundaries. The two cases where word boundaries are relevant are discussed separately with the operators LT48 (insert 48-bit literal) and MPCW (make PCW).

In diagrams specifying opcode and parameter interpretation, the operator name is used to represent its opcode value. (Opcode values are specified in the Operator Encoding and Operator Reference Information appendices). Vertical bars (|) denote syllable boundaries, and dotted vertical lines (:) denote parameter boundaries not corresponding to syllable boundaries. Where relevant, a word boundary is denoted by a double vertical bar (||).

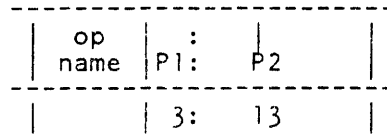
The following example diagram shows a 3-syllable operator, including two single-syllable parameters.



The next diagram shows a 3-syllable operator, two syllables of which are a single parameter.



The final diagram shows a 3-syllable operator including two parameters that are mapped into two syllables. P1 is 3 bits and P2 is 13 bits.



Primary, Variant and Edit Operators

There may be several interpretations of an opcode syllable, depending upon context.

Primary opcodes are represented in a single syllable.

Variant opcodes are represented by two syllables, of which the first is the primary operator VARI.

Edit opcodes are found in special tables (specified as an argument to an Enter Table Edit operator, TEED or TEEU) or in the code-stream immediately following an Enter Single Edit operator (EXSD, EXSU, or EXPU).

Common Actions : common action

The concept of a "common action" is used in this document for a function that is common to several operators. Common actions are defined to effect economy (by reducing repetition), to improve rigor, and to provide a convenient reference for citation. Common actions are given three-and four-letter names like operators, but with a prefix of "a", as in "aPRCW". It should be emphasized that a common action specification is a rhetorical device used to specify operators; it is not a constituent of the system architecture.

Initial and Restart State

Some Primary and Variant operators can be entered in either of two states, initial or restart. By default, all operators begin in initial state. In some cases, when the execution of an operator must be interrupted, it may be necessary to resume in some other way, for example, with a different stack configuration or different assumptions about some system state. For these operators, a restart state is defined.

If an operator invokes interrupt entry (aINTE) or accidental entry (aACCE), it may cause RCW.rs to be set to 1, so that the operator will be resumed in restart state following the interruption. When EXIT or RETN finds RCW.rs = 1, the next operator executed is begun in restart state.

Those cases that require the use of the restart mechanism are specified; other cases may use restart as an implementation option (see, for example, the pragmatic note under NXLV). A given operator may have at most one restart state different from its normal initial state. The semantics of restart state are defined for each specified instance of its use.

In general, the information implied by the use of restart state must be preserved until the operator is completed: once an operator has been resumed in restart state, any subsequent interruption and re-summation of the same operation must use restart state.

Checks and Interrupts

Throughout this section, checks are defined to verify argument types, consistency of data, bounds on indexes, integrity of structures, and other related topics. The checks are generally stated in the form "if (some condition) then (some interrupt) is generated." Interrupts are defined generally and specifically in section 4; it may suffice for now to say that interrupt generation causes the current operator (usually) to abort its current function, and to cause a designated operating-system procedure to be invoked. Not all the checks specified here are required of every implementation; some are defined as "optional" in Appendix C. Not all the checks applicable to the operator are mentioned in every operator description; many are described generally for a class of operators. All the interrupts applicable to each operator are specified in Appendix C.

Expression Stack Control

Most operators require items from the top of the expression stack, and leave their result(s) on top of the stack. Stack items required by operators are called arguments. They are normally consumed; that is, they are used and deleted from the stack. To avoid excessive repetition, deletion of arguments is assumed for all operators, unless explicitly noted.

Top-of-Stack Push Operations

Operators that produce top-of-stack results must "push" them, in order, onto the expression stack. If the item being pushed is a double-precision operand, the first word is pushed below the second, both with tag = 2.

The top of the expression stack has a nominal address defined to be S ; the proper values for S are in the range $(D[LL]+2)$ -to- $(LOSR-1)$, where $LOSR$ is maintained equal to $DD.address + DD.length$ for the stack descriptor. Whenever a word is pushed onto the expression stack, S is incremented by one and that address is assigned to the pushed word. If, as a result of the push, $S = LOSR$, a formal Stack-Overflow condition exists.

A Stack-Overflow interrupt is required only when data are written to the expression stack in memory; an implementation is free to keep some of the top-of-stack words in local processor state (registers). It must be noted that S is defined in the architecture as the address corresponding to the top word in the expression stack; if, at a given moment, a processor has captured k top-of-stack words in local state, the address of the top word actually in memory is then $S_m = S - k$. Although S_m is not defined as architecture state, it may be substituted for S in the definition of stack overflow: an implementation may define the Stack-Overflow condition to be $S_m = LOSR$.

Note that Stack-Overflow is detected only when a push completes with the top-of-stack address exactly equal to $LOSR$. If a Stack-Overflow condition occurs on pushing the first word of a double-precision item, the second word is pushed before the interrupt is generated.

If a Stack-Overflow is detected while $Interrupt_Count > 0$, the interrupt generation is deferred until $Interrupt_Count$ is set to zero (by the ZIC operator).

The memory used by a stack is not strictly limited to the actual segment defined by the stack DD. If Stack-Overflow is detected, the stack segment will be overrun for the following reasons:

1. The word whose push is detected as an overflow is stored as the first word past the end of the defined actual segment.
2. The operator that detects the stack overflow may complete, pushing one or more additional words onto the stack.
3. The Stack-Overflow interrupt generation pushes 4 words onto the stack.
4. Any top-of-stack words held in optimization registers can be pushed into memory.
5. Software can push some additional words onto the stack in the process of handling the interrupt.

The total number of words pushed into memory for the first four reasons cannot exceed 50 in the worst case.

Pragmatic Notes

Stack overflow will overrun the declared stack

For Stack-Overflow interrupt generation and handling to complete without overwriting a critical value in memory, the memory allocation for the stack must be larger than the value in the length field of the stack descriptor.

Top-of-Stack Pop Operations

Any operator that requires a stack argument must "pop" it from the expression stack. If the word at the top-of-stack has tag = 2, the word below is also popped; if this word does not have tag = 2, the result is undefined. The top and next words are taken as the second and first words of a double-precision operand. (Note that because argument items may be either single or double words, multiple arguments must be accessed by popping them in order, from the topmost down.)

The expression stack utilizes the set of locations whose nominal addresses are above $D[LL]+2$. Note that the topmost activation record stack linkage words at $D[LL]$ and $D[LL]+1$ are excluded. If $S < D[LL]+2$ and an operator attempts to use an expression stack argument, a Stack-Underflow interrupt is generated; this checking is required for all operators that utilize stack arguments.

Descriptor Interpretation

Most of the data and all the code in a system architecture reside in memory segments accessed by means of data and code-segment descriptors. The address of a particular memory word is computed by adding an index to the base address of an actual segment. For data, the index and the reference to the base of the actual segment are combined into a single item, an IndexedDD; for code, the segment base reference is in the CSD and the index is derived from the code-stream-pointer component PWI.

The word referenced by an IndexedDD is located as follows: the nominal address of the word is calculated by adding the index value from the descriptor to the base address of the segment. If the descriptor is marked present, its address field contains the base address; otherwise that field contains the nominal address of the original DD, whose address field contains the base address if the segment is present. If the IndexedDD and the referenced original DD are both marked absent, a Presence Bit interrupt is generated. If the referent of the absent copy DD is not an original DD, an Invalid Object interrupt is generated.

Code addressing is similar to data addressing, except that the CSD provides only the segment-base reference; the index is in PWI. (Note that when edit-mode code is executed as a result of an enter-table-edit operator, the code is referenced by an IndexedDD.) Code is never referenced through a copy CSD, so the indirection described for data reference through an absent IndexedDD does not apply; an absent CSD causes a Presence Bit interrupt to be generated.

COMPUTATIONAL OPERATORS

Operators in this group are loosely termed computational operators because they take arguments directly from the stack and leave some form of result on top of the stack.

Computational operators do not evaluate references; their arguments must be items on the stack initially. Required parametric values may be static code parameters or dynamic stack arguments.

Numeric Operand Interpretation

Computational operators act on single- or double-precision operands interpreted as integers or floating-point numbers. Binary computational operators require two operands to be present on the stack and unary computational operators require one. Single-precision and double-precision operands are defined in section 1 of this manual.

In the following discussions, the symbols $+$, $-$, $*$, and $/$ are used to denote respectively the add, subtract, multiply, and divide arithmetic functions, and the *** symbol denotes the exponentiation function. Other symbols and combinations of symbols represent implied arithmetic functions, as follow:

Symbol	Meaning
$a = b$	a Equal To b
$a < b$	a Less Than b
$a > b$	a Greater Than b
$a \leq b$	a Less Than Equal to b
$a \geq b$	a Greater Than Equal to b
$a \neq b$	a Not Equal to b
$\{a,b,c\}$	Set a through c, including b
$a \rightarrow b$	a Mapped Into Set b
$ a $	a(absolute)
S n	n is Single-precision
D n	n is Double-precision
R n	n Rounded
T n	n Truncated
N n	n Normalized
I n	n Integerized
RS n	n Single-precision, Rounded
RD n	n Double-precision, Rounded
TS n	n Single-precision, Truncated
TD n	n Double-precision, Truncated
Ri n	n Absolute-value, Rounded-to-Integer
NS	Normalized, Single-precision
ND	Normalized, Double-precision
RaI n	n Algebraic-value, Rounded-to-Integer
Ti n	n Absolute-value, Truncated-to-Integer
RId	Rounded-to-Integer, Double-precision
TId	Truncated-to-Integer, Double-precision

Representable Operand Formats

Operand formats depend upon context and purpose. Generally, operands are of type INTEGER or type REAL. An INTEGER is a value which does not require an exponent part. A REAL is any value that requires an exponent part or contains a decimal-point (octal-point).

INTEGER values are usually expressed as single-precision operands. However, INTEGER values are also expressed as double-precision operands because of arithmetic function logic. This logic requires that whenever an input parameter to an arithmetic function is a double-precision operand the result of the function must also be expressed as a double-precision operand.

REAL values are expressed as either single-precision or double-precision operands. These values are floating-point expressions which require an exponent part or a value containing a component less than unity (a fractional-value).

Single-Precision Operand Values

Single-precision operands can contain any value in the range: $-549,755,813,887$ with an exponent of -64 (decimal), through $+549,755,813,887$ with an exponent of $+64$ (decimal).

Double-Precision Operand Values

Double-precision operands contain values in the range: $1.55083668571006866684511$ with an exponent of -29580 decimal through $1.94882838205028079124466$ with an exponent value of $+29580$ decimal.

Automatic Arithmetic Functions

Certain arithmetic functions are automatically performed by the system. These are rounding, truncation, integerization, normalization, and the conversion of operands to single-or double-precision. Some of these functions are also implemented as unique operator codes, and thus may be executed as part of user-program options.

Particular arithmetic operators predefine the formats of resultant operands. The architecture computes an arithmetic function resultant value and then adjusts the value to conform to the predefined result operand format. This methodology requires that rounding and truncation be used to fit resultant values into the fixed operand formats. Errors while an arithmetic operator is in process may be due to a wrong value result or to a Loss-Of-Precision that occurred from forcing a resultant to fit into a predefined operand format.

The architecture must be able to determine the nature of computational operation errors and to categorize errors by defining whether an error is a Loss-Of-Precision, Integer-Overflow, Exponent-Overflow, or an Exponent-Underflow error. Interrupts are described in section 4 of this manual.

Normalization is a computational function that removes leading-zeroes from an operand by adjusting the value of its exponent. Normalization is used by the architecture to facilitate arithmetic logic circuitry. The alignment of mantissa values, through normalization, enhances the efficiency of arithmetic operations. If the requirement to normalize an operand cannot be performed due to the limited size of the exponent field, a Loss-Of-Precision error is detected.

Integerization is a computational process that adjusts the mantissa of an operand until it is in integer format. Integer format was described previously in this section. If an operand cannot be adjusted so that it is in integer format (because of the size of its exponent field) an Integer-Overflow error is detected.

Numeric-Interpretation Operators

The operators in the following groups interpret operands numerically as a primary part of their function; numeric interpretation also occurs as some part of the function of operators in other groups.

Arithmetic Operators

Arithmetic operators require either one or two operands on top of the stack. If the items are not operands, an Invalid Stack Argument interrupt is generated.

Binary operators will generate a single-precision result if both operands are single-precision and a double-precision result if either or both operands are double-precision. Where required, single-precision is extended to double-precision prior to the operation by appending a second word of all zeros. Note that the numeric value of the operand is not changed.

For example, in the architecture,

```
1 * 8**(-63) MULT 2 → 2 * 8**(-63)
1 * 8**(-63) DIVD (1/2) → 2 * 8**(-63)
1 * 8**(-63) MULT 2.5 → 3*8**(-63) with precision loss
1 * 8**(-63) DIVD 4 → 0 with precision loss
```

These examples illustrate another difference: this architecture produces the proper result (0 for Exponent Underflow; unnormalized small number for precision loss); where predecessor systems depend upon software to replace the stack result with a zero of the appropriate type.

ADD (add)

ADD requires two operands on top of the stack. The numeric values of the two operands are algebraically added and rounded, and the result is left on top of the stack.

Exponent-Underflow or Overflow is never generated by ADD. If both x and y are single__integers and the absolute value of the sum is less than 2^{*39} , then the result is a single__integer.

SUBT (subtract)

SUBT requires two operands on top of the stack. The numeric value of the top item is algebraically subtracted from the numeric value of the second item and rounded, and the result is left on top of the stack.

Exponent-Underflow is never generated by SUBT. If both x and y are single__integers and the absolute value of the difference is less than 2^{*39} , then the result is a single__integer.

MULT (multiply)

MULT requires two operands on top of the stack. The numeric values of the two operands are algebraically multiplied and rounded, and the result is left on top of the stack.

If both x and y are single__integers and the absolute value of the product is less than 2^{*39} , then the result is a single__integer.

If the result of the rounding function causes the exponent value to be too large to fit in the operand format exponent field, an Exponent-Overflow interrupt is detected. If rounding causes the exponent value to be too small to fit in the operand format exponent field, an Exponent-Underflow interrupt is detected.

MULX (extended multiply)

MULX requires two operands on top of the stack. Any single-precision operand is extended to double-precision before the numeric values are algebraically multiplied and rounded. The double-precision result is left on top of the stack.

DIVD (divide)

DIVD requires two operands on top of the stack. The numeric value of the second item is algebraically divided by the numeric value of the top item and rounded, and the result is left on top of the stack. If the divisor (top-of-stack operand) equals zero, a Divide by Zero interrupt is generated.

If the result of the rounding function causes the exponent value to be too large to fit in the operand format exponent field, an Exponent-Overflow interrupt is detected. If rounding causes the exponent value to be too small to fit in the operand format exponent field, an Exponent-Underflow interrupt is detected.

IDIV (integer divide)

IDIV requires two operands on top of the stack. The numeric value of the second item is algebraically divided by the numeric value of the top item. The fractional part of the floating point quotient is discarded, and the integer part is left on top of the stack in canonical integer representation.

If the divisor (top-of-stack operand) equals zero, a Divide by Zero interrupt is generated. If the truncation function results in an exponent value too large to fit in the operand format exponent field space, an Integer-Overflow interrupt is generated.

RDIV (remainder divide)

RDIV requires two operands on top of the stack. The numeric value of the second item is divided by the numeric value of the top item. The integer quotient with remainder is generated but only the remainder is left on top of the stack. The sign of the result is the same as the sign of the second item (the dividend).

If the divisor (top-of-stack operand) equals zero, a Divide by Zero interrupt is generated. Neither Exponent-Overflow nor Exponent-Underflow can be generated by RDIV. However, Integer-Overflow is generated whenever IDIV would generate Integer-Overflow for the same arguments.

NORM (normalize)

NORM requires an operand on the top-of-stack; otherwise an Invalid Stack Argument interrupt is generated. If the operand is single-precision, it is converted to normalized single-precision representation. If the operand is double-precision, it is converted to normalized double-precision representation.

If the result of the rounding function causes the exponent value to be too large to fit in the operand format exponent field, an Exponent-Overflow interrupt is detected. If rounding causes the exponent value to be too small to fit in the operand format exponent field, an Exponent-Underflow interrupt is detected.

AMIN and AMAX (arithmetic minimum and maximum)

AMIN (and AMAX) require two operands on top of the stack. The numeric values of the two operands are compared and the arithmetically lesser (or greater) of the two operand values is left as the result on top of the stack. If one of the input operands is single-precision and the other input operand is double-precision, the single-precision operand is extended before the comparison, and a double-precision result is generated. If both of the inputs are single-integers, then so is the result.

Relational Operators

The relational operators all require two operands on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. The numeric value of the second item is algebraically compared to the numeric value of the top item, and a Boolean result is left on top of the stack. The form of the Boolean results True and False is defined in Boolean Operands.

LESS (less than)

LESS leaves a True result if the second from top-of-stack operand is arithmetically less than the top operand and a False result otherwise.

LSEQ (less than or equal to)

LSEQ leaves a True result if the second from top-of-stack operand is arithmetically less than or equal to the top operand and a False result otherwise.

EQUL (equal to)

EQUL leaves a True result if the second from top-of-stack operand is arithmetically equal to the top operand and a False result otherwise.

NEQL (not equal to)

NEQL leaves a True result if the second from top-of-stack operand is arithmetically not equal to the top operand and a False result otherwise.

GREQ (greater than or equal to)

GREQ leaves a True result if the second from top-of-stack operand is arithmetically greater than or equal to the top operand and a False result otherwise.

GRTR (greater than)

GRTR leaves a True result if the second from top-of-stack operand is arithmetically greater than the top operand and a False result otherwise.

Range Test Operators

The range test operators compute the result of a double arithmetic inequality, $L \leq X \leq H$. The value of X is a stack argument and is left on the stack along with the Boolean result.

RNGT (range test)

The RNGT operator includes the values of L and H as two eight-bit parameters.



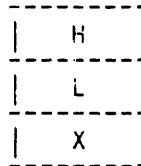
RNGT requires one operand on top of the stack (X); otherwise an Invalid Stack Argument interrupt is generated.

Two operands are produced as a result of RNGT. The topmost result is the Boolean result of the arithmetic inequality (that is, the result is True if the inequality is True, and it is False if the inequality is False).

The other (bottom-most) result is an identical copy of the input, X.

DRNT (dynamic range test)

The DRNT operator is identical to the RNGT operator except that the values of L and H are stack arguments instead of code parameters.



DRNT requires three operands on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. As in RNGT, two operands are produced.

Numeric Type-Transfer Operators

The following type transformations may be invoked on the top-of-stack item by operators defined in this group.

- NTIA, NTGR: Convert operand numeric value to single__integer.
- NTGD, NTTD: Convert operand numeric value to double__integer.
- SNGT, SNGL: Convert operand numeric value to single-precision.
- SNGT: Convert WordDD to SingleDD.

The following type transformations, among others, may be invoked on the top-of-stack item by operators defined in Bit-Vector Type-Transfer Operators.

- XTND: Convert single-precision operand to double-precision, or convert WordDD to DoubleDD.

For the following type transfer operators, the top-of-stack operand is denoted x.

NTIA (integerize truncated)

NTIA requires an operand on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. The operand is converted to `single__integer` representation by truncation and the result is left on top of the stack. If the truncation function results in a value too large to fit in a `single__integer` word format an Integer-Overflow interrupt is generated.

NTGR (integerize rounded)

NTGR requires an operand on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. The operand is converted to `single__integer` representation by rounding and the result is left on top of the stack. If the rounding function results in a value too large to fit in a `single__integer` word format an Integer-Overflow interrupt is generated.

SNGL (set to single-precision rounded)

SNGL requires an operand on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. The operand is converted to normalized single-precision representation and is left on top of the stack.

If the rounding function results in an exponent value too large to fit in a `single__integer` operand format, an Exponent-Overflow interrupt is detected. If the result of the normalization function or the rounding function results in an exponent value too small to fit in a `single__integer` operand format, an Exponent-Underflow interrupt is detected.

SNGT (set to single-precision truncated)

SNGT requires an operand or `WordDD` on top of the stack; otherwise an Invalid Stack Argument interrupt is generated.

If the argument is an operand, it is converted to normalized single-precision representation and left on top of the stack.

If the truncation function results in an exponent value too large to fit in a `single__integer` operand format, an Exponent-Overflow interrupt is detected. If the result of the normalization function or the truncation function results in an exponent value too small to fit in a `single__integer` operand format, an Exponent-Underflow interrupt is detected.

If the argument is a `SingleDD`, it is left on the stack unchanged. If the argument is an unindexed `DoubleDD` with length $\geq 2^{*}19$, an Invalid Argument Value interrupt is generated. Otherwise, if the argument is a `DoubleDD`, it is left on the stack as a `SingleDD`: the `element__size` is set to single-precision, and if the `DoubleDD` is unindexed, its length field is multiplied by 2.

NTTD (integerize double-precision truncated)

NTTD requires an operand on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. The operand is converted with truncation to `double__integer` representation, and the result is left on top of the stack.

If the truncation function results in an exponent value too large to fit in a `double__integer` operand format, an Exponent-Overflow interrupt is detected. If the result of the normalization function or the truncation function results in an exponent value too small to fit in a `double__integer` operand format, an Exponent-Underflow interrupt is detected.

NTGD (integerize double-precision rounded)

NTGD requires an operand on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. The operand is converted with rounding to double__integer representation and the result is left on top of the stack.

If the rounding function results in an exponent value too large to fit in a double__integer operand format, an Exponent-Overflow interrupt is detected.

aISX (integer subset exception action)

aISX is a common action invoked by operators that require an argument to be within an integer subset, if the argument is not a k-bit integer (where k is determined by the invoking operator).

If the argument is not an operand, an Invalid Stack Argument interrupt is generated.

If the argument is an operand but not a single__integer, the implementation may be defined to integerize the operand (as in NTGR); if the operand cannot be integerized, an Integer-Overflow interrupt is generated. If the integerized operand is a k-bit integer, the invoking operator proceeds to use the integerized value in place of the original argument. If the integerized argument is not a k-bit integer, or the implementation does not perform integerization, the action in the next paragraph is performed.

If the argument is an operand but not a k-bit integer, either an Invalid Argument Value or an Invalid Stack Argument interrupt is generated, as implementation-defined.

The implementation options may be defined separately for each invocation of aISX.

Pragmatic Notes

Minimal Specification Constraint for Low-Level Operators

The aISX action is defined to avoid over-specifying the error action to be taken in the implementation of certain operators that are "low-level". This means that their use is normally restricted to operating-system software.

The preferred implementation of aISX is to generate an interrupt for any argument that is not a k-bit integer, rather than to integerize; this mechanism will catch the most software errors. The flexibility is available so an implementation can "borrow logic" from other operators if some economy is thereby effected.

Scale Left

Scale left operators perform multiplication of an operand on top of the stack by 10 raised to a power specified by a scale factor. The scale factor may be a dynamic argument or a static parameter.

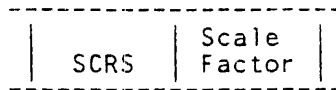
The item to be scaled must be an operand; otherwise an Invalid Stack Argument interrupt is generated. If the operand is not an integer, it is integerized with the RID function; if it cannot be integerized, an Integer-Overflow interrupt is generated.

If the scale factor is a dynamic argument, it must be an operand; otherwise an Invalid Stack Argument interrupt is generated. It is integerized with rounding if required, and if it cannot be integerized, an Integer-Overflow interrupt is generated, and if the result is a valid integer but not in the range 0-to-12, an Invalid Argument Value interrupt is generated.

The remainder is a single-precision operand interpreted as a left-justified decimal (hex) sequence. The number of decimal digits in the remainder is equal to the scale factor, and each digit is in the range hex "0" to hex "9". The values of the rightmost 12-<scale factor> digits are undefined. The remainder is the unsigned result of dividing the absolute value of the integerized argument by a power of ten.

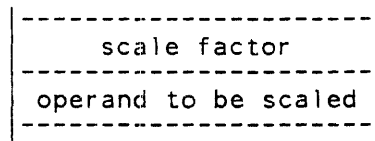
SCRS (scale right save)

The argument to be scaled is integerized using the RId function. SCRS leaves the quotient on top of the stack and the remainder second from top of the stack. The correct sign of the entire result is left in the sign bit of the quotient, even if the quotient itself is zero. The operand to be scaled is required on top of the stack, and the scale factor is a parameter:



DSRS (dynamic scale right save)

The operation is the same as SCRS, but the scale factor is required on top of the stack above the operand to be scaled: only the quotient is left on top of the stack.



SCRT (scale right truncate)

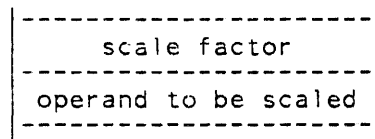
The argument to be scaled is integerized using the TId function. Only the quotient is left on top of the stack. (The operation effectively applies the Ti function to the quotient $x/10^{**n}$.)

The operand to be scaled is required on the stack, and the scale factor is a parameter:



DSRT (dynamic scale right truncate)

The operation is the same as SCRT, but the scale factor is required on top of the stack above the operand to be scaled:



SCRR (scale right rounded)

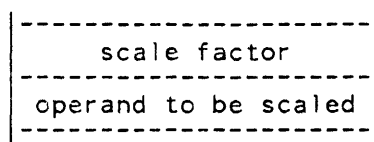
The argument to be scaled is integerized, using the RId function if the scale factor is zero and the TId function otherwise. Only the quotient is left on top of the stack. If the most significant digit of the remainder is greater than or equal to five, the magnitude of the quotient is increased by one. (The operation effectively applies the generic Ri function to the quotient $x/10^{**n}$.)

The operand to be scaled is required on top of the stack, and the scale factor is a parameter:



DSRR (dynamic scale right rounded)

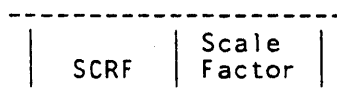
The operation is the same as SCRR, but the scale factor is required on top of the stack above the operand to be scaled:



SCRF (scale right final)

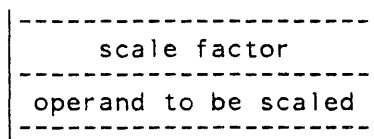
The argument to be scaled is integerized using the RId function. Only the remainder is left on top of the stack. EXTf (external sign flip-flop) is set to 1 if the mant__sign of the operand to be scaled is minus and to 0 otherwise. OFFf (overflow flip-flop) is set to 1 if the quotient is any non-zero value; OFFf remains unchanged if the quotient is zero.

The operand to be scaled is required on top of the stack, and the scale factor is a parameter:



DSRF (dynamic scale right final)

The operation is the same as SCRF, but the scale factor is required on top of the stack above the operand to be scaled:



Binary to Decimal Conversion

The binary-to-decimal conversion operators are variations of the scale-right-final operators, in that the result is a decimal digit sequence representing the remainder of division by a power of ten. There are two operators, a static form (BCD) with the number of digits specified as a code parameter, and a dynamic form (DBCD) with the number of digits supplied as a stack argument.

The number to be converted is a stack argument that must be an operand; otherwise an Invalid Stack Argument interrupt is generated. If necessary, that argument is integerized with rounding (using the RId function); if the operand cannot be integerized, an Integer-Overflow interrupt is generated. This integerized argument is referred to below as B (the binary integer).

$|B| \bmod 10^{**}N$ is converted to a sequence of N decimal digits, where N is provided as a code parameter or a stack argument. The result is left justified in an operand, which is single-precision if N is 12 or less and double-precision if N is in the range 13 to 24. The contents of the operand beyond the N-digit sequence are undefined.

EXTF is set to:	false (positive)	if $B \geq 0$,
	true (negative)	if $B < 0$ and $ B \bmod 10^{**}N > 0$,
	undefined state	if $B < 0$ and $ B \bmod 10^{**}N = 0$.
OFFF is set to:	true (overflow)	if $ B \geq 10^{**}N$.
	unchanged	if $ B < 10^{**}N$.

Pragmatic Notes

Binary-to-decimal operators relate to scale-right operators

The binary-to-decimal operators differ from the scale-right operators in two ways:

- The binary-to-decimal operators do not set EXTF true if the binary argument is -0 .
- The binary-to-decimal operators accept $N > 12$. For $12 < n \leq 24$, the following code sequences produce the same result operand:

Static n		Dynamic n	
			LT8 12
			SUBT
BCD n	SCRS n-12	DBCD	DSRS
	SCRF 12		SCRF 12
	EXCH		EXCH
	JOIN		JOIN

BCD (binary convert to decimal)

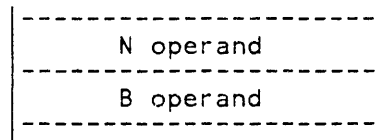
The B operand is the only stack argument; N is a code parameter:

(variant) BCD number of
digits, N

If $N > 24$, an Invalid Code Parameter interrupt is generated.

DBCD (dynamic binary convert to decimal)

Two stack arguments are required; if either is not an operand, an Invalid Stack Argument interrupt is generated.



The topmost argument is integerized with rounding (RaI function), if necessary, to produce N; if the argument cannot be integerized, an Integer-Overflow interrupt is generated. If N is not in the range 0-to-24, an Invalid Argument Value interrupt is generated. The second argument provides B.

Bit Vector Interpretation

This group of operators provides various functions. Those operators that act on stack items either interpret the items as bit vectors or deal with the word as a whole. In general, there are few restrictions on the type of stack items that will be acted upon.

Logical Operators

Logical operators require one or two top-of-stack items; they may be of any type. The items are interpreted as 48-bit vectors, unless one or both are double-precision items. In that case they are interpreted as 96-bit vectors, and if only one of the two items is double precision, the other is extended with 48 zero bits (whether the item is an operand or not).

The logical operation is applied in parallel to each bit of the vectors, and the result is left on top of the stack. For the unary LNOT operator, the tag of the result is the same as the tag of the top of stack item. For the binary logical operators the result is double-precision if either argument is double-precision; otherwise the tag of the result is the tag of the second from top item.

The four logical operations are illustrated here in binary notation:

```

      NOT   01 =   10
0011 AND 0101 = 0001
0011 OR  0101 = 0111
0011 EQV 0101 = 1001

```

LNOT (logical not)

LNOT requires a single top-of-stack item. All bits of the vector are complemented, and its tag remains unchanged.

LAND (logical and)

LAND requires two top-of-stack items. The logical AND of the two bit vectors is left on top of the stack.

LOR (logical or)

LOR requires two top-of-stack items. The logical OR of the two bit vectors is left on top of the stack.

LEQV (logical equivalence)

LEQV requires two top-of-stack items. The logical EQV (equivalence) of the two bit vectors is left on top of the stack.

Relational Operator

SAME (logical equality)

SAME requires two top-of-stack items. They are interpreted as 52-bit vectors (including tag bits). If all corresponding bits of the two vectors have the same value, a True result is left on top of the stack; otherwise a False result is left. If both items are double-precision, the bit vector interpretation includes the second words. Note that if only one item is double-precision, the result is necessarily false.

Literal Operators

Literal operators place a single-precision constant on top of the stack. They do not use any initial top-of-stack items.

ZERO (insert literal zero)

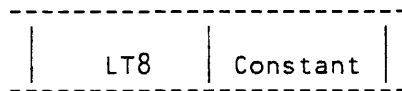
ZERO leaves on top of the stack a single-precision word with all bits initialized to zero.

ONE (insert literal one)

ONE leaves on top of the stack a 1-bit integer equal to 1.

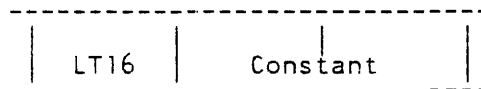
LT8 (insert 8 bit literal)

LT8 leaves on top of the stack an 8-bit integer that is a copy of its one-syllable parameter.



LT16 (insert 16 bit literal)

LT16 leaves on top of the stack a 16-bit integer that is a copy of its two-syllable parameter.



XTND (set to double-precision)

XTND requires an operand or a WordDD on top of the stack; otherwise an Invalid Stack Argument interrupt is generated.

If the argument is a double-precision operand, it is left on the stack unchanged. If it is a single-precision operand, it is converted to double-precision representation by appending a second word whose fields are initialized to zero; the double-precision result is left on the stack. Note that its numeric value is not changed.

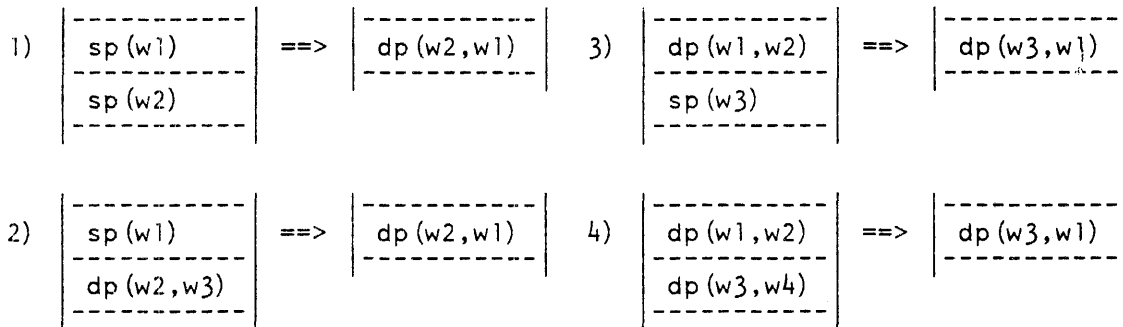
If the argument is a DoubleDD, it is left on the stack unchanged. If the argument is a SingleDD, it is left on the stack as a DoubleDD: its element__size is set to double-precision, and if the SingleDD is unindexed, its length field is divided by 2; any remainder is discarded.

This architecture XTND disallows CharDDs, whereas B6800 XTND looks only at bit 40, the "double-precision bit", of a data descriptor. The value of bit 40 is zero in the encoding of EBCDIC or hex element__size values, and by setting it to 1, B6800 XTND generates an invalid encoding. Furthermore, the DD length is improperly divided by 2 in this case.

JOIN (set two singles to double)

JOIN requires two operands on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. A double-precision item is constructed from the two operands, and the result is left on top of the stack.

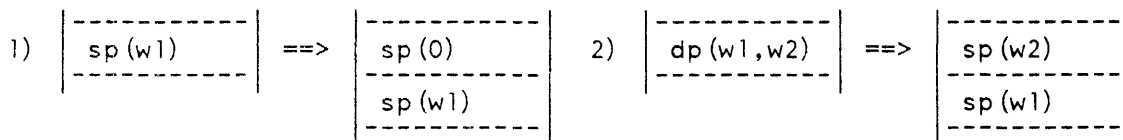
The first and second words of the double-precision result are taken from the first words of the second and top operands respectively. The following possibilities arise from combinations of single and double-precision operands:



SPLT (set double to two singles)

SPLT requires an operand on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. Two single-precision items are constructed from the operand and left on top of the stack.

If the operand is single-precision, it is left on the stack and a single precision zero is pushed on the stack above it. If the operand is double-precision, its two words are converted to two single-precision items. The first word is pushed on the stack first, and the second word is left on top of the stack.



Evaluate Word Structure Operators

RTAG (read tag)

RTAG requires one item on top of the stack, and its result is a 4-bit integer whose value is the tag of the item.

CBON (count binary ones)

CBON requires an operand on top of the stack; otherwise an Invalid Stack Argument interrupt is generated. The number of binary-ones present in the operand are counted. If the operand is double-precision, all 96-bits are examined. CBON leaves a 7-bit integer value, which is the number of binary-ones counted, on top of the stack.

LOG2 (leading one test)

LOG2 requires one item on top of the stack, and then replaces it with a 6-bit integer value. The integer contains the bit-number of the leading (most-significant) binary-one bit in the stack item. If all bits in the item are binary-zeroes, LOG2 leaves an integer zero on the stack; otherwise the integer contains the (number +1 of the) highest-order binary-one bit in the item. Only the first word (upper-half) of a double-precision stack item is examined.

Word Manipulation Operators

Word manipulation operators provide the capability to alter any "partial field" of a word in the stack called the destination, in some cases based on a field of another word in the stack called the source. The following operations are provided:

BSET, DBST:	Set a single destination bit.
BRST, DBRS:	Reset a single destination bit.
ISOL, DISO:	Create a destination whose low-order field is set from a field of the source.
INSR, DINS:	Set a field of the destination from the low-order field of the source.
FLTR, DFTR:	Set a field of the destination from a field of the source.
CHSN:	Complement the "sign" bit (bit 46) of the destination.

Source items may be of any type. Except for CHSN, destination items may be any type. The altered destination item is left on the top of the stack. If the source is a double-precision item, the field is taken from its first word, and the second word is discarded. If the destination is a double-precision item, the bit or field altered is in its first word, and the second word is retained unchanged in the double-precision result. The following terms are used for bit/field specifications:

Db:	The destination bit to be set or reset, or the high-order bit of the destination field.
Sb:	The high-order bit of the source field.
Len:	The length of both the source and destination fields.

There are static and dynamic operators corresponding to several of the operations. The static operators take Db, Sb, and Len specifications, as required, from code parameters; the dynamic operators take them from stack arguments.

If dynamic Db, Sb, and Len specification items are not operands, an Invalid Stack Argument interrupt is generated. They are integerized with rounding, if required, and if they cannot be integerized, an Integer-Overflow interrupt is generated. All Db and Sb values must be in the range {0 to 47}, and Len values must be in {0 to 48}. Static operators generate an Invalid Code Parameter interrupt if any of these values are invalid, and dynamic operators generate an Invalid Argument Value interrupt if any are invalid.

The effect of word manipulation operators will be shown as an assignment to a field of the destination word. The remainder of the destination word is not changed. Note that Len = 0 is a valid specification of a null field; in this case the destination will not be altered at all.

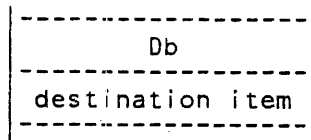
BSET (bit set)

BSET sets a single destination bit: destination.[Db:1] := 1. The destination is the only required top-of-stack item, and Db is specified by a parameter:



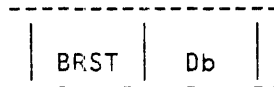
DBST (dynamic bit set)

DBST sets a single destination bit: destination.[Db:1] := 1. The required initial stack state includes Db:



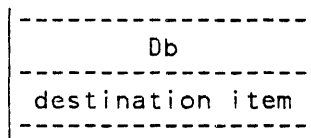
BRST (bit reset)

BRST resets a single destination bit: destination.[Db:1] := 0. The destination is the only required top-of-stack item, and Db is specified by a parameter:



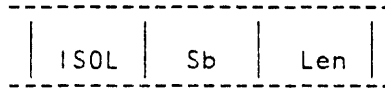
DBRS (dynamic bit reset)

DBRS resets a single destination bit: destination.[Db:1] := 0. The required initial stack state includes Db:



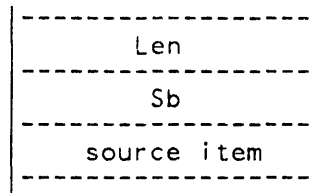
ISOL (field isolate)

ISOL creates a single-precision destination word initialized to zero, and then sets its low-order field from a field of the source: `destination := 0; destination.[Len-1:Len] := source.[Sb:Len]`. The source is the only required top-of-stack item, and Sb and Len are specified by parameters:



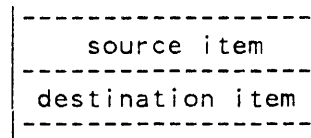
DISO (dynamic field isolate)

DISO creates a single-precision destination word initialized to zero, and then sets its low-order field from a field of the source: `destination := 0; destination.[Len-1:Len] := source.[Sb:Len]`. The required initial stack state includes Len and Sb:

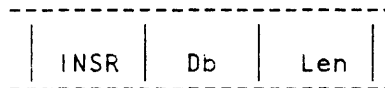


INSR (field insert)

INSR sets a field of the destination from the low-order field of the source: `destination.[Db:Len] := source.[Len-1:Len]`. The required initial stack state includes only the source and destination:

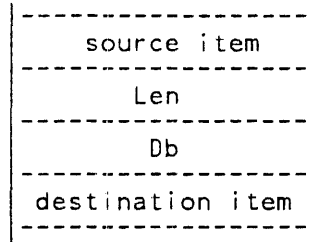


Values for Db and Len are specified by parameters:



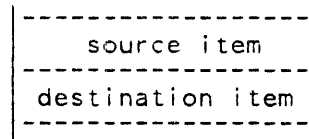
DINS (dynamic field insert)

DINS sets a field of the destination from the low-order field of the source: `destination.[Db:Len] := source.[Len-1:Len]`. The required initial stack state includes Len and Db (but note that for DINS, the source item is required on top of the stack):

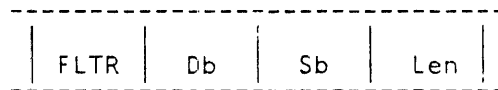


FLTR (field transfer)

FLTR sets a field of the destination from a field of the source: `destination.[Db:Len] := source.[Sb:Len]`. The required initial stack state includes only the source and destination:

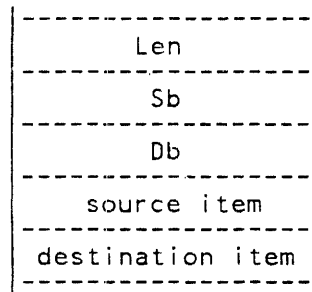


Values for Db, Sb, and Len are specified by parameters:



DFTR (dynamic field transfer)

DFTR sets a field of the destination from a field of the source: `destination.[Db:Len] := source.[Sb:Len]`. The required initial stack state includes Len, Sb and Db:



CHSN (change sign)

The CHSN operator requires an operand on top of the stack; otherwise, an Invalid Stack Argument interrupt is generated. CHSN complements a single destination bit: `destination.[46:1] := NOT destination.[46:1]`

Linear Index-Function Operator

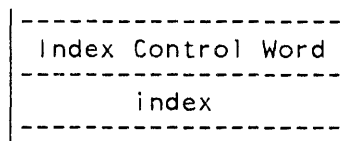
OCRX (occurs index)

OCRX computes a linear integer function of an integer index, with bounds checking. The function is defined as

$$\text{RelativeIndex}(\text{offset}, \text{length}, \text{index}) = \text{offset} + (\text{index}-1)*\text{width};$$

where index must be in the range {1 to limit}.

OCRX leaves on top of the stack the result of the RelativeIndex function applied to values derived from two arguments:



The Index Control Word (ICW) must be a single-precision operand. It contains three fields: :index control word (ICW)

ICW__width	[47:16]	The width coefficient
ICW__limit	[31:16]	the upper bound for the index
ICW__offset	[15:16]	the offset coefficient

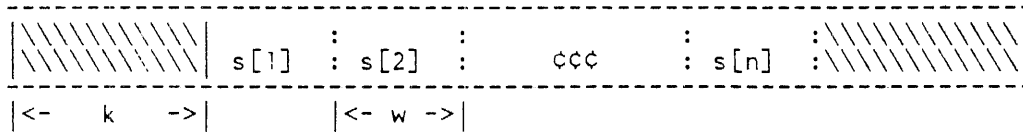
If the ICW is not a single-precision operand or if the index is not an operand, an Invalid Stack Argument interrupt is generated. The index argument is integerized with rounding if required; if it cannot be integerized, an Integer-Overflow interrupt is generated.

If the index is not in the range {1 to ICW__limit}, an Invalid Index interrupt is generated. Otherwise, OCRX leaves on top of the stack a single__integer whose value is RelativeIndex (ICW__offset, ICW__width, index).

Pragmatic Notes

OCRX is an indexing-computing function

OCRX is intended for computing indexing functions to sub-records within a linear record structure. For example, assume that a record contains a sequence of sub-records s[1] to s[n]; each sub-record contains w elements of the base record type, and the first sub-record begins k elements into the record:



The index for $s[i]$ relative to the base of the record can be computed by the `RelativeIndex` function of the `OCRX` operator, with i as the index and an ICW composed with:

```
ICW__width = w
ICW__limit = n
ICW__offset = k
```

Note that `ICW__limit` and index are in the same units (ordinal index of the sub-record); `ICW__width` and `ICW__offset` are in the same arbitrary units (typically characters or words); the `RelativeIndex` function transforms from the first set of units to the second.

REFERENCE GENERATION AND EVALUATION OPERATORS

The common feature of this operator group is the generation and evaluation of references and chains of references. The group consists of reference generation operators (generators) and operators that evaluate references in order to read a target item onto the stack (read evaluators) or store an item from the stack into a target location (store evaluators).

Basic evaluation of a reference consists of calculating the nominal memory address to which it refers. Read evaluation consists of fetching the contents of the referenced location; store evaluation consists of writing the contents of that location.

Double Precision

References to double-precision operands refer to the first word. The need for a second word may be indicated by the tag of the first word or by the reference (`IndexedDoubleDD`). The second word is in the next higher memory location. (Note that this architecture, like predecessor implementations, does not permit a double-precision operand to be split between pages of a virtual-segment.)

Pragmatic Notes

Beware Aliasing Address Couples or IRWs with `IndexedWordDDs`

Care must be exercised in any situation in which an address-couple parameter or IRW might reference the same location as an `IndexedWordDD`, if there can be any conflict as to operand precision. Such a situation might arise if a descriptor is pointed into part of a stack (creating an "in-stack array"), and the same locations are also referenced by address-couple. Read evaluation operators such as `VALC` and `LOAD` distinguish single-from double-precision operands by the tag of the operand, if it is addressed directly by an address-couple parameter or an IRW; the determination is made from the `element__size` of an `IndexedWordDD`, if such a descriptor is the last element in the reference chain. Similarly, normal store operators require the store operand to match the target type, which is determined from the tag of the target word (address-couple parameter or IRW) or by the `element__size` (`IndexedWordDD`).

Stack references

A stack is accessed by its stack number as follows. The stack-vector descriptor at address-couple (0,2) is indexed by the stack number. If the stack number is not in the range {0 to SVD.length-1}, an Invalid Index interrupt is generated; otherwise the stack descriptor is fetched. If either the word accessed as the stack-vector descriptor or the word accessed as the stack descriptor is not an unpagged unindexed SingleDD, an Invalid Object interrupt is generated. If the stack descriptor is an absent original DD, a Presence Bit interrupt is generated. The SVD must be a present original DD, and the stack descriptor must not be an absent copy DD; the effect of violating these restrictions is undefined.

Lexical Link Evaluation

A lexical link is a (stack number, displacement) couple; it specifies the base of an activation record. Basic evaluation of a lexical link consists of computing the nominal base address of the activation record by adding the displacement to the base address of the referenced stack. If the stack number equals the contents of SNR, the base address is found in BOSR; otherwise, it is obtained from the stack descriptor, as described in Stack References.

aLXLK (evaluate lexical link)

The common action aLXLK is defined to perform the evaluation defined in this section.

Lexical Chains

The addressing environment of a process is a list of activation records: the base address of the topmost record is recorded in D[LL]; the MSCW at that address contains a lexical link to the record for level LL-1, and so on to level zero.

If an implementation maintains display registers, it is necessary to traverse all or part of the lexical chain whenever the topmost activation record changes, as occurs in procedure entry/exit and move-stack operators (see Display Update). If an implementation does not have a full set of display registers, it may be necessary to traverse part of the lexical chain in order to find the activation record for a lexical level less than LL.

During lexical chain traversal, a Stack Structure Error is generated if the word addressed by a lexical link is not an entered MSCW, or if the MSCW of the activation record for lexical level *i* does not contain *i* in the lex_level field.

aLXCH (traverse lexical chain)

The common action aLXCH is defined to perform lexical chain traversal and consistency checking defined in this section.

Address-Couple Evaluation

Address couples occur in operator parameters and in Normal Indirect Reference Words (NIRWs). Those in parameters occur in either fixed-or variable-fence forms; NIRWs contain fixed-fence address couples. A name-call operator uses its address-couple parameter to construct an NIRW on the stack, where it will become an initial reference for a subsequent operator. Other operators use their address-couple parameter as their own initial reference.

Basic evaluation of an address-couple (Lambda,Delta) consists of calculating the nominal address of the referenced word, by adding Delta to the base address of the activation record whose lexical level is specified by Lambda. If Lambda = LL, the activation record base is in D[LL]. If not, the activation record base address can be read from a display register, if such registers are implemented, or found by traversing the lexical chain beginning at D[LL] (common action aLXCH). Note that since Lambda specifies an activation record in the current addressing environment, the result of address-couple evaluation may vary according to the environment.

When an address-couple is evaluated, Lambda must be less than or equal to LL, and for Lambda = LL, the address of the referenced stack location must be less than or equal to the address of the top-of-stack (S); otherwise, an Invalid Reference interrupt is generated. Furthermore, for Lambda = LL, referencing data that has not been explicitly pushed with an ENTR or PUSH operator is an undefined operation.

Evaluation of References

Read and store evaluators share the general capability to process a chain of references in order to locate some target item. Reference chains may be composed of address-couple parameters, IRWs (NIRWs and SIRWs), IndexedWordDDs, and PCWs.

Definition of valid target items and allowable reference chains depends on the function of the particular operator, but evaluation of each element of a reference chain and of IRW chains is common to the operator group. The following sections define evaluation of each reference form, IRW chain evaluation, and the notation used for each operator to specify allowable reference chains and valid target items.

Address Couple Parameters

Name-call operators use the parameter to construct an NIRW. Other operators evaluate the parameter to determine the corresponding nominal address, which is then typically used for read or store access.

NIRWs

The STFF operator transforms an NIRW into an SIRW that references the same location. Other operators evaluate the NIRW address-couple to determine the corresponding nominal address, which is then typically used for read or store access.

NIRWs may be initial references only.

SIRWs

Basic evaluation of an SIRW consists of calculating the nominal address of the referenced word: SIRW.offset is added to the address derived from the Lexical Link (SIRW.stack__number, SIRW.displacement) by the common action aLXLK. (The ENTR operator uses the Lexical Link address as well as the sum.)

The result of evaluation of an SIRW is constant regardless of the current addressing environment.

No validity check is performed on the sizes of the displacement and offset fields during SIRW evaluation. SIRWs are created from NIRWs, and the NIRW components are verified at that time.

Pragmatic Notes

SIRW as a parameter reference

The SIRW is a reference to an item in an activation record in a stack; it is context-independent in that its interpretation does not depend upon the current addressing environment (apart from the D[0] environment, which determines the location of the stack-vector descriptor, which defines the stacks). The principal application of SIRWs is to pass reference parameters from one addressing environment to another. IndexedWordDDs serve as references to items in segments other than stacks.

IndexedWordDDs

Basic evaluation of a data descriptor as a reference is possible only for an IndexedWordDD referring to a present segment. The evaluation consists of calculating the nominal address of the referenced word (see Descriptor Interpretation).

In cases where the target of an IndexedWordDD is an operand, the element type of the operand is determined by the `element_size` field of the IndexedWordDD (the last if a sequence of IndexedWordDDs was evaluated). The `element_size` value of single-or double-precision overrides the tag of the target operand. All operators that evaluate IndexedWordDDs, with the exception of LODT and the overwrite operators, obey this convention.

An IndexedSingleDD may be used to reference words of any type appropriate to the referencing operator. An IndexedDoubleDD may be used only to reference an operand (for read evaluation) or an even-tagged word (for normal store evaluation).

PCWs

In the context of reference chain evaluation, evaluation of a PCW consists of an "accidental" procedure entry. The PCW is assumed to point to a function with no parameters, whose returned value will be either the target of the chain or another valid reference. The net result of the accidental entry is that the place of the PCW in the reference chain is taken by the item returned by the function.

The accidental entry is accomplished by the `aACCE` action, defined in Procedure entry operators. It is assumed that the function will terminate with a RETN (return) operator that leaves an item on top of the stack. If it does not, the item on top of the stack will be used incorrectly, as if it were such a result.

When the operator resumes, the result of the function is treated as the target or next reference. If it is not valid in the context of the operator, an Invalid Stack Argument interrupt, rather than an Invalid Reference Chain interrupt, is generated.

IRW Chains

Throughout this group of operators, those that evaluate multiple references will evaluate a sequence of one or more IRWs wherever a single IRW may be evaluated. Because NIRWs may occur as initial references only, these chains consist of an optional NIRW referencing a chain of SIRWs. Chains of IRWs that may optionally contain the initial NIRW are referred to as "IRW chains"; chains of IRWs that may not contain an NIRW are referred to as "SIRW chains". (It is often convenient to regard an address-couple parameter, as well as an NIRW, as the head of an "IRW chain".)

Evaluation of the IRW chain consists of successive IRW evaluations, starting with the head of the chain, until IRW read evaluation does not produce an IRW.

Reference Chains

Operators that evaluate reference chains start from an initial reference and apply successive reference read evaluation, according to a set of chaining rules, until a target item is produced. For each such operator, the set of initial references and targets is specified using the following notation:

```
<Initial Reference> ::= {set of reference items}
<target> ::= {set of target items}
```

Chaining rules are specified by showing the valid evaluation results for each reference form that may be a part of the chain. The form of such specification is:

```
reference → evaluation results,
```

Where "→" indicates read evaluation of the reference as defined in the preceding sections. Evaluation results can include reference forms or an Initial Reference, any of which is subsequently evaluated, or a target. Evaluation of the chain will continue until a target is encountered or until reference evaluation produces an item that is not a valid result. If chain evaluation terminates with an invalid item, an interrupt is generated. With two exceptions, the interrupt is Invalid Reference Chain: a Binding Request interrupt is generated if the target is a DD with `element_size = 7`; an Invalid Stack Argument interrupt is generated if the result of PCW evaluation (aACCE) is unacceptable.

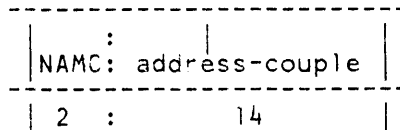
Chaining rule notation is illustrated by the following example. (Note that IndexedWordDD is used for the union of IndexedSingleDD or IndexedDoubleDD, which may be listed separately in the expansion.)

```
<Initial Reference> ::= {NIRW, SIRW chain, IndexedWordDD}
<target> ::= {operand}
NIRW → SIRW chain
      IndexedWordDD
      PCW
      <target>
SIRW chain → IndexedWordDD
             PCW
             <target>
IndexedSingleDD → IndexedWordDD
                  <target>
IndexedDoubleDD → <target>
PCW → SIRW chain
      IndexedWordDD
```

Reference Generation Operators

NAMC (name call)

The NAMC operator transforms an address-couple in the code-stream into an NIRW. NAMC is a 2-syllable operator with a special structure, a 2-bit opcode and a 14-bit variable-fence address-couple:



The NAMC operator converts the variable-fence address-couple parameter into a fixed-fence address-couple and leaves it in an NIRW on top of the stack. Name-call operators need not interpret the address-couple; however, the tests on Lambda and Delta defined for address-couple evaluation may be applied and an Invalid Reference interrupt generated if either component is out of range.

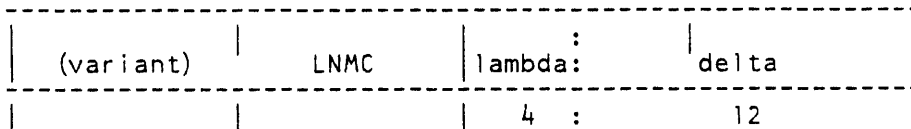
Pragmatic Notes

NAMC is sensitive to lexical level

The transformation from fixed-to variable-fence address-couple is made with the variable fence set according to LL at the time the NAMC operator is executed. Because NIRWs are intended for immediate consumption (within the same block activation), there is no loss of generality due to this "premature binding" of the fence.

LNMC (long name call)

LNMC is equivalent to NAMC, except that its parameter is a fixed-fence rather than a variable-fence address-couple. LNMC is a 4 syllable operator whose appearance in the code-stream is:



Pragmatic Notes

LNMC provides full-range Delta at any LL

Because it has a fixed-fence address-couple, LNMC can be used to construct address couples with Delta as large as $2^{12} - 1$ at any lexical level. NAMC can address the full range of Delta values only when $LL \leftarrow 3$.

STFF (stuff)

STFF converts the NIRW on top of the stack into an SIRW. If STFF encounters an SIRW on top of the stack, it terminates leaving the SIRW. If the top-of-stack item is not an NIRW or SIRW, an Invalid Stack Argument interrupt is generated.

The (Lambda,Delta) address-couple in the NIRW is interpreted as described in Address-Couple Evaluation. If $\text{Lambda} > \text{LL}$, or $\text{Lambda} = \text{LL}$ and $\text{address}(\text{stack location}) > \text{S}$, an Invalid Reference interrupt is generated. Otherwise the SIRW is constructed to point to the word in the stack addressed by (Lambda,Delta). The displacement field is set to the stack-relative offset to the MSCW for the activation record referenced by the NIRW; if that offset exceeds $2 \times 16 - 1$, a Stack Structure Error interrupt is generated. The stack__number field is set to the stack number containing this activation record. The offset field is set to the value of delta from the NIRW. The unused fields are set to zero.

Pragmatic Notes

STFF Algorithm

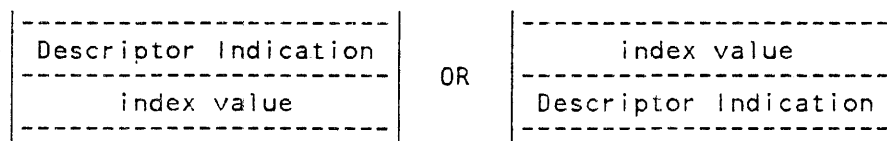
The following algorithm produces the Lexical Link corresponding to the Lambda component of an address-couple: If $\text{Lambda} = \text{LL}$, the Lexical Link is (SNR, D[LL]-BOSR); otherwise, the Lexical Link is contained in the MSCW for the activation record at lexical level $\text{Lambda} + 1$ in the current addressing environment (at D[Lambda+1] if that display register is implemented).

INDX (index)

INDX applies an integer index to an unindexed DD and leaves on top of the stack an IndexedDD pointing to the specified element. If the DD is a WordDD, the result is an IndexedWordDD, and if it is a CharDD, the result is a Pointer. An unindexed copy DD may be on the stack initially, or an unindexed original or copy DD may be addressed by an IRW chain.

<Descriptor Indication>	::= {unindexed copy WordDD, unindexed copy CharDD, initial reference}
<Initial Reference>	::= IRW chain
<target>	::= {unindexed WordDD, unindexed CharDD}
IRW chain	→ <target>

INDX requires the Descriptor Indication and an operand index value on top of the stack in either order:



If the index value is not an operand, an Invalid Stack Argument interrupt is generated. If the Descriptor Indication is a copy DD with `element__size = 7`, a Binding Request interrupt is generated; otherwise, if the Descriptor Indication is not an unindexed copy WordDD or CharDD or the head of an IRW chain, an Invalid Stack Argument interrupt is generated.

The index value is integerized with rounding (by using the RaI function defined for the NTGR operator) if required. If it cannot be integerized, an Integer-Overflow interrupt is generated. If the resulting integer is not in the range {0 to DD.length-1}, an Invalid Index interrupt is generated. (Both DD.length and the index value are assumed to be in DD.element__size units).

An IndexedWordDD or a Pointer is constructed according to the value of DD.element__size. For the different element__size values, the effective index values are derived from the index argument as follows:

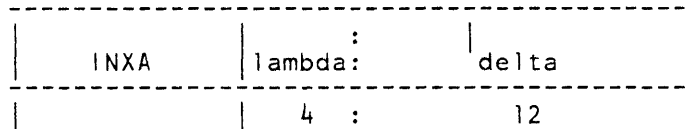
0 (single-precision):	WI = index
1 (double-precision):	WI = index * 2
2 (hex):	WI = index DIV 12; CI = index MOD 12
4 (EBCDIC)	WI = index DIV 6; CI = index MOD 6

If the unindexed DD is unpagged, an indexed DD is constructed from it as follows: If the designated descriptor is an original DD, aCPY action is invoked to produce a copy DD. The present, copy, read__only, element__size, and address fields are copied from the unindexed copy DD. The indexed bit is set to 1. For a WordDD, an IndexedWordDD is constructed by setting the index field to WI; if WI exceeds $2^{*}20-1$, an Invalid Index interrupt is generated. For a CharDD, a Pointer is constructed by setting the word__index field to WI and the char__index field to CI; if WI exceeds $2^{*}16-1$, an Invalid Index interrupt is generated.

If the data-segment is paged (the unindexed DD has paged = 1), INDX resolves the paging by performing another level of indexing. The paged descriptor is indexed by WI DIV page__size, and the referenced word is accessed as a page descriptor. If it is an original unpagged SingleDD, the indexing operation proceeds; otherwise a Page Structure Error interrupt is generated. An indexed DD is created as follows: A copy of the page DD is fetched, using common action aCPY. The indexed bit is set to 1. The element__size and read__only fields are copied from the initial (paged) DD; the present and address fields are retained from the aCPY copy of the page DD. For a WordDD, an IndexedWordDD is constructed by setting index to WI MOD page__size. For a CharDD, a Pointer is constructed by setting word__index to WI MOD page__size and char__index to CI.

INXA (index by means of address-couple parameter)

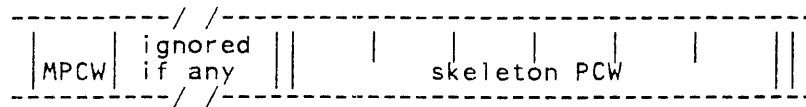
INXA is a 3-syllable operator containing a fixed-fence address-couple; the code-stream appearance is:



INXA is functionally equivalent to a name-call operator, containing the same address-couple, followed immediately by the INDX operator.

MPCW (make PCW)

MPCW is a literal operator that constructs a PCW at the top of the stack from a six-syllable parameter in the code-stream. The parameter is taken from the first code-word following the MPCW opcode. "Padding" syllables, if any, from the opcode to the end of the word containing the opcode are ignored. The PCW is constructed from the parameter by setting the tag to 7 and putting the value of SNR into the stack_number field.



The parameter is assumed to be a valid PCW skeleton. It is inserted at the top of the stack and tagged as a PCW.

Read Evaluation Operators

aFOP (Fetch Operand)

The common action aFOP is invoked by operators that evaluate a reference to fetch an operand value (as opposed to LODT and RDLK, which fetch only one word, not a whole operand in the double-precision case). The action can be described as a procedure with one formal parameter, the reference to be evaluated, and three possible results:

1. An operand value.
2. A non-operand value.
3. An Invalid Object interrupt: odd-tagged second word.

If the referenced word does not have a tag of 0 or 2, it is a non-operand value.

If the reference is an IRW and the referenced word has tag = 0, or the reference is an IndexedSingleDD and the referenced word has tag = 0 or 2, the referenced word is fetched and the tag set to zero (if necessary) to form an operand value.

If the reference is an IRW and the fetched word has tag = 2, its successor word (at the next higher nominal memory address) is fetched. If that word has a tag other than two, an Invalid Object interrupt is generated; otherwise, the referenced word and its successor are joined as the first and second words of a double-precision operand to form an operand value.

If the reference is an IndexedDoubleDD, the physical successor word of the referenced word is fetched. If the successor word has an odd tag, an Invalid Object interrupt is generated; otherwise, the referenced word and its successor are joined as first and words of a double-precision operand (with the tag set to two) to form an operand value.

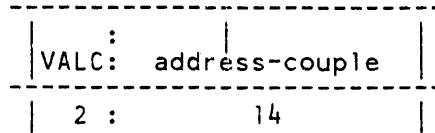
aCPY (fetch copy descriptor)

The common action aCPY is invoked by any operator that accesses a descriptor (by evaluating a reference) and places a copy of that descriptor on the stack. (The exceptions are LODT and RDLK, which make a facsimile of the original descriptor rather than a "copy" of it.)

A duplicate of the referenced descriptor is brought to the stack to become the copy. (If it is already a copy, no transformation is necessary.) If the descriptor is an absent original, the address field is replaced by the nominal memory address at which the original was located; otherwise the address is retained unchanged. The copy bit is set to 1.

VALC (value call)

The VALC operator evaluates a reference chain whose head is an address-couple parameter; the target must be an operand, which is left on top of the stack. VALC is a 2-syllable operator with a special structure, a 2-bit opcode and a 14-bit variable-fence address-couple:



Reference chain evaluation performed by VALC is:

< Initial Reference >	::=	{address-couple}
< target >	::=	{operand}
address-couple	→	SIRW chain IndexedWordDD < target >
SIRW chain	→	IndexedWordDD PCW < target >
IndexedSingleDD	→	IndexedWordDD < target >
IndexedDoubleDD	→	< target >
PCW	→	SIRW chain IndexedWordDD < target >

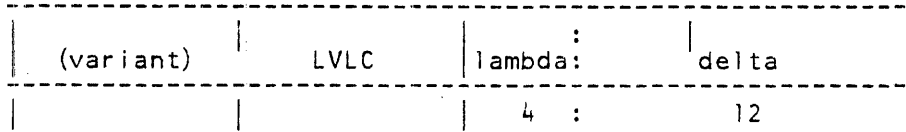
Reference evaluation is performed by invocation of the common action aFOP: an operand value is left on the stack as the result of VALC; any non-operand value is examined as an element of the reference chain. If reference evaluation produces a DD with element_size = 7, a Binding Request interrupt is generated.

If a PCW must be evaluated, accidental entry is performed by invoking the common action aACCE. The RCW.rs bit is set in the new activation record; when resumed in restart state, VALC ignores its code parameter and consumes a stack argument as the target of the PCW. If this argument is not a valid reference or target, an Invalid Stack Argument interrupt is generated.

Otherwise, if reference evaluation produces an item that is not a valid result according to the above chain evaluation rules, an Invalid Reference Chain interrupt is generated.

LVLC (long value call)

LVLC is equivalent to VALC, except that its parameter is a fixed-fence rather than a variable-fence address-couple. LVLC is a 4-syllable operator whose appearance in the code-stream is:



Pragmatic Notes

LVLC provides full-range Delta at any LL

Because it has a fixed-fence address-couple, LVLC can be used to construct address couples with Delta as large as $2^{12} - 1$ at any lexical level. VALC can address the full range of Delta values only when $LL \leq 3$.

The LVLC operator, new in this architecture, is identical to the VALC operator except that its parameter is a fixed-fence address-couple.

NXLV (index and load value)

NXLV performs an INDX (index) operation to produce an IndexedWordDD and then evaluates the IndexedWordDD to fetch an operand.

The required initial stack state is the same as that for INDX except that the DD must be a WordDD:

<Descriptor Indication>	::=	{unindexed copy word DD, initial reference}
<Initial Reference>	::=	IRW chain
<index target>	::=	unindexed WordDD
IRW chain	→	<index target>

If the index operation is unsuccessful, an interrupt is generated as specified for INDX. If the <index target> is successfully indexed, the ultimate target is fetched by read evaluation of the Indexed-WordDD:

IndexedWordDD → operand

The IndexedWordDD is evaluated by invoking the common action aFOP: an operand value is left on the stack as the result of NXLV; any non-operand value causes an Invalid Object interrupt to be generated.

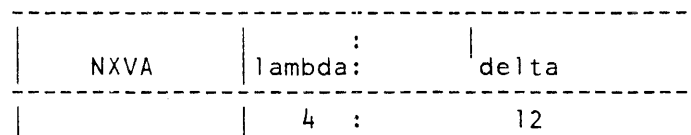
Pragmatic Notes

Presence Bit interrupt may use Restart

When NXLV or NXLN generates a Presence Bit interrupt, the stack configuration and RCW can be constructed either to repeat the operator in initial state, with two arguments, or to repeat the operator in restart state, with the already-indexed descriptor as the only argument. (In the initial-state case, the Descriptor Indication argument could be the original argument, or it could be the unindexed copy descriptor after any IRW chain evaluation.)

NXVA (index and load value by means of address-couple parameter)

NXVA is a 3-syllable operator containing a fixed-fence address-couple; the code-stream appearance is:



NXVA is functionally equivalent to a name-call operator, containing the same address-couple, followed immediately by the NXLV operator.

NXLN (index and load name)

NXLN performs an INDX (index) operation to produce an IndexedSingleDD and then evaluates the IndexedSingleDD to fetch an unindexed DD; a copy of that DD is left on top of the stack.

The required initial stack state is the same as that for INDX except that the descriptor indication or index target must be a SingleDD.

<Descriptor Indication>	::=	{unindexed copy SingleDD, initial reference}
<Initial Reference>	::=	IRW chain
<index target>	::=	unindexed SingleDD
IRW chain	→	<index target>

If the index operation is unsuccessful, an interrupt is generated as specified for INDX. If the <index target> is successfully indexed, the ultimate target is fetched by read evaluation of the IndexedSingleDD:

IndexedSingleDD → unindexed DD

If the target is not an unindexed DD, an Invalid Object interrupt is generated.

The target is fetched as a copy (with aCPY action) and left on the top of the stack. (The NXLN operator does not examine the paged, read_only, element_size, length/index, or address field of a target DD.)

EVAL (evaluate)

The purpose of EVAL is to evaluate a reference chain in order to locate some target and then leave on top of the stack the reference whose evaluation produced the target.

Reference chain evaluation performed by EVAL is:

<Initial Reference>	::=	{NIRW, SIRW chain, IndexedWordDD}
<target>	::=	{even-tag word, unindexed DD, IndexedDD with element_size > 1}
NIRW	→	IndexedWordDD SIRW chain PCW <target>
SIRW chain	→	IndexedWordDD PCW <target>
IndexedWordDD	→	* no evaluation – see below *
PCW	→	IndexedWordDD SIRW chain

If a target is located, the reference whose evaluation produced the target is left on top of the stack as the result. If an IndexedWordDD is encountered, it is left as the result without being evaluated. In effect, an IndexedWordDD is treated as if it had been evaluated and a target had been the result.

If a PCW must be evaluated, accidental entry is performed by invoking the common action aACCE. If the result of the function is not a valid reference, an Invalid Stack Argument interrupt is generated.

Otherwise, if reference evaluation produces an item that is not a valid result according to the above chain evaluation rules, an Invalid Reference Chain interrupt is generated.

LOAD (load)

LOAD performs a single evaluation of the Initial Reference, and if the result is a target, it is left on top of the stack:

<Initial Reference>	::=	{NIRW, SIRW, IndexedWordDD}
<target>	::=	{operand, tag-4 word, tag-6 word, SIRW, any data descriptor}
NIRW	→	<target>
SIRW	→	<target>
IndexedSingleDD	→	<target>
IndexedDoubleDD	→	operand (target)

If the item on top of the stack is not an Initial Reference, an Invalid Stack Argument interrupt is generated.

Reference evaluation is performed by invocation of the common action aFOP: an operand value is left on the stack as the result of LOAD; any non-operand value is examined as a possible target. If the Initial Reference is an IndexedDoubleDD and the target is not an operand, an Invalid Object interrupt is generated.

If the target is a DD, it is fetched as a copy (with aCPY action) and left on the top of the stack. (The LOAD operator does not examine the indexed, paged, read__only, element__size, length/index, or address field of a target DD.)

In all other cases, if the referenced item is a valid <target>, it is left on top of the stack without conversion; otherwise, an Invalid Object interrupt is generated.

LODT (load transparent)

LODT performs a single evaluation of the Initial Reference and leaves the result on top of the stack, with no restriction placed on the type of the result:

<Initial Reference>	::=	{NIRW, SIRW, IndexedSingleDD, 20-bit integer address}
<target>	::=	{any item}
NIRW	→	<target>
SIRW	→	<target>
IndexedSingleDD	→	<target>
Integer	→	<target>

If the argument is not an Initial Reference, one of the following actions is performed, depending upon the type and range of the argument and on the implementation-defined handling of an invalid operand argument:

not operand:	Generate Invalid Stack Argument interrupt.
operand not single__integer:	Generate Invalid Stack Argument or Invalid Argument Value interrupt, or integerize (which may generate Integer Overflow) and test/use resulting integer.
single__integer not in {0 to $2^{*}20 - 1$ }:	Generate Invalid Stack Argument or Invalid Argument Value or Invalid Address interrupt.

If the Initial Reference is an IRW or an IndexedSingleDD, it is evaluated normally. If it is a 20-bit integer, it is interpreted as a nominal memory address from which the target is fetched.

The addressed target word is left on top of the stack. If its tag is 2, the second word of the item left on top of the stack is zero. (Apart from the handling of tag = 2, the LODT operator does not examine or alter the target word in any way.)

Pragmatic Notes

Improper operand-address action is flexible

The error action if the LODT argument is an operand but not a 20-bit integer is specified like the aISX action (q.v.), with the additional option of generating Invalid Address interrupt. This specification permits an implementation to treat integers greater than this architecture address width the same as integers that are within that limit but exceed the implementation memory path. As for aISX, the preferred implementation is to interrupt rather than to integerize a noninteger argument.

Pragmatic Notes

Invalid Address used as ODI

The LODT operator may generate the Invalid Address (alarm) interrupt in ODI fashion, by detecting that the operand argument is improper, as well as in alarm fashion when a memory fetch fails.

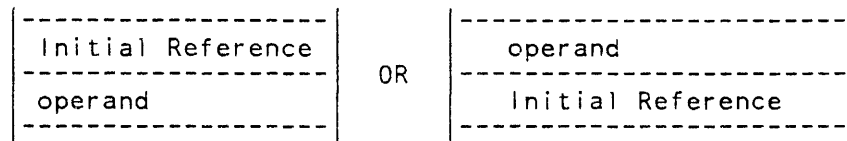
Store Evaluation Operators

Normal Store Operators

Normal store operators evaluate a reference chain in order to store an operand from the stack (the store object) into a target location. Reference chain evaluation performed by store operators is:

<code><Initial Reference></code>	<code>::=</code>	<code>{NIRW, SIRW chain, IndexedWordDD}</code>
<code><target></code>	<code>::=</code>	even-tagged word
NIRW	<code>→</code>	SIRW chain Indexed WordDD PCW <code><target></code>
SIRW chain	<code>→</code>	IndexedWordDD PCW <code><target></code>
IndexedSingleDD	<code>→</code>	IndexedWordDD <code><target></code>
IndexedDoubleDD	<code>→</code>	<code><target></code>
PCW	<code>→</code>	SIRW chain Indexed WordDD

For STOD and STON, the Initial Reference and the operand are required on top of the stack, in either order:



If the top-of-stack item is not an Initial Reference or an operand, or if the top-of-stack item is an Initial Reference and the second item is not an operand, or if the top-of-stack item is an operand and the second item is not an Initial Reference, an Invalid Stack Argument interrupt is generated.

If any reference evaluation produces an odd-tag item other than an IRW, DD, or PCW, or if an IndexedWordDD is marked `read_only`, a Memory Protect interrupt is generated. If any reference evaluation produces a DD with `element_size = 7`, a Binding Request interrupt is generated.

If a PCW must be evaluated, accidental entry is performed by invoking the common action `aACCE`. STOD or STON deletes the Initial Reference argument and then invokes `aACCE`, so that the result of the accidental-entry procedure becomes a new Initial Reference; these operators require no restart state. For STAD or STAN, the `RCW.rs` bit is set in the new activation record; when resumed in restart state, STAD or STAN ignores its code parameter and is functionally equivalent to STOD or STON, respectively. The chaining rules for a PCW successor are enforced partly by the store operators (which generate an Invalid Stack Argument interrupt if the Initial Reference is not a valid reference) and partly by the RETN operator (which generates an Invalid Stack Argument interrupt if its argument is an NIRW).

If reference evaluation produces an item otherwise not a valid result according to the above chain evaluation rules, an Invalid Reference Chain interrupt is generated.

The normal store operators perform additional type checking; if any of the following situations occur, an Invalid Object interrupt is generated:

1. The operand is single-precision and:
 - 1) The final reference is an IndexedDoubleDD, or tag = 2.
2. The operand is double-precision and:
 - 1) The final reference is an IndexedSingleDD, or
 - 2) The final reference is an IRW and the referenced word has tag = 0.

If the operand is single-precision, it is stored at the target location, with a tag of zero.

If the operand is double-precision, the successor of the target location (at the next higher nominal memory address) is examined. If the successor location contains an odd-tagged word, a Memory Protect interrupt is generated; otherwise, the first and second words of the operand are written into the target location and its successor, respectively; both words have tag = 2. If an interrupt is generated while a double-precision value is being stored, the first half of the item may or may not have been stored, depending upon the implementation.

STOD (store delete)

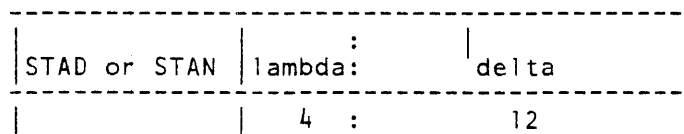
A normal store operation is performed. Both the Initial Reference and the operand are deleted from the stack.

STON (store non-delete)

A normal store operation is performed. The operand is left unchanged on top of the stack, and the Initial Reference is deleted.

STAD and STAN (store delete/non-delete by means of address-couple)

STAD and STAN are 3-syllable operators that contain a fixed-fence address-couple; the code-stream appearance is:



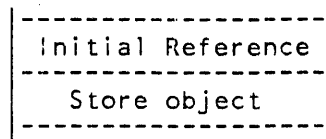
The STAD (or STAN) operator is functionally equivalent to a name-call operator, containing the same address-couple, followed immediately by the STOD (or STON) operator, except that PCW evaluation requires the use of restart state (as specified above).

Overwrite Operators

Overwrite operators perform a single evaluation of the Initial Reference and store some item from the stack (the store object) into the resultant target location. There are no restrictions on either the store object or the initial contents of the target location; the "→" notation indicates store evaluation for these operators, which do not examine the target contents (except for RDLK).

<Initial Reference>	::=	{NIRW, SIRW, IndexedSingleDD}
<target>	::=	{any item}
NIRW	→	<target>
SIRW	→	<target>
IndexedSingleDD	→	<target>

The topmost stack item must be an Initial Reference; otherwise an Invalid Stack argument is generated. The second stack item is the Store object. If the Initial Reference is an IndexedSingleDD marked read_only, a Memory Protect interrupt is generated.



Overwrite operators store single words and are oblivious to double-precision: If the store object is double-precision, only the first word is stored, with its tag of 2. If the reference addresses a tag-2 word, only that word is overwritten; its successor is unchanged. Note that each case can produce an unpaired double-precision word in memory.

OVRD (overwrite delete)

An overwrite operation is performed. Both the Initial Reference and the store object are deleted from the stack.

OVRN (overwrite non-delete)

An overwrite operation is performed. The store object is left unchanged on top of the stack, and the Initial Reference is deleted.

RDLK (read and lock)

RDLK is identical to OVRD, except that the word previously occupying the target location is left as the stack result. If the result has tag = 2, the second word of the double-precision result is set to zero.

In any implementation, the following requirements must be met by all processors and by any other processing elements (such as I/O or communications processors) that share the system memory and use the RDLK convention for synchronization.

1. RDLK reads the former contents and writes the new contents of the referenced word as an indivisible operation; no other processor can access the referenced word between the read and write steps of a RDLK operator.
2. Store operations for each processor effectively occur in order. That is, if a code-stream on one processor initiates stores into location A and then into location B, any processor that finds the new value at B (with RDLK) and subsequently examines A must find the new value at A.
3. No data-fetch memory operation that follows the RDLK in the dynamic code-stream of a processor may take place until after the RDLK operation.

Statements 2 and 3 amount to restrictions on the ability of an implementation to reorder the manipulating of independent pieces of state (such as the contents of different memory locations).

Pragmatic Notes

RDLK locking protocols

RDLK provides a primitive locking mechanism for multiprocessor systems. The following illustration of a locking protocol is useful to point out why each of the three requirements above is necessary:

Consider two storage locations, x and y. Assume that y is to be used as a lock protecting the contents of x; 0 and 1 denote unlocked and locked, respectively. The following program is being executed simultaneously in multiple processors:

- a: lock y: store 1 into y by means of RDLK until the result is not 1;
- b: fetch value from x;
- c: perform function on value;
- d: store new value into x;
- e: unlock y: store 0 into y.

(Steps b through d constitute a "critical region" that is to be executed on at most one processor at a time.)

Requirement 1 provides that exactly one processor will complete step a. Let us call this processor number 1, and say that it has completed step 1a. The other processor(s) will remain in step a, looping, until the completion of step 1e.

Requirement 2 provides that step 1d is completed before step 1e.

After step 1e, another processor can complete step a; let us call that processor number 2 and say that it has completed step 2a. Requirement 3 provides that step 2b does not begin until step 2a has successfully completed.

The ordering constraints combine to assure that the value fetched in step 2b is the one that was stored in step 1d.

Interlock Operators

Operators in this group are defined together because they all manipulate the interlock data type and therefore share particular characteristics. An interlock can be associated with any desired program state. By "locking" that interlock (executing LOK) prior to entering a critical region and "unlocking" it (executing UNLK) upon leaving the region, a process can be assured of exclusive access to the associated state (assuming that all other processes observe the same locking convention).

The LOK and UNLK operators perform the lock and unlock operations, respectively. If more than one process is contending for the same interlock, these operators generate special service interrupts to permit the operating system to resolve the contention. The LOKC operator performs a conditional lock operation, avoiding the interrupt if the lock operation fails. When a LOK or LOKC operator is successfully completed, the process that initiated the operator is allowed to proceed and is recorded as the "owner" of the interlock. The UNLK operator removes the current ownership, permitting another process to complete a LOK or LOKC and acquire ownership. LOK, UNLK, and LOKC are special store reference-evaluation operators.

The LKID operator reads the state of an interlock, reporting the owner (if any) of the interlock. LKID is a special read reference-evaluation operator.

Interlock operators have a single stack argument, an Initial Reference that must directly reference the interlock. They evaluate the Initial Reference and perform the required manipulation of the target interlock.

<Initial Reference>	::=	{NIRW, SIRW, IndexedSingleDD}
<target>	::=	interlock
NIRW	→	<target>
SIRW	→	<target>
IndexedSingleDD	→	<target>

The topmost stack item must be an Initial Reference; otherwise an Invalid Stack Argument interrupt is generated. Except for LKID, if the Initial Reference is an IndexedSingleDD marked read_only, a Memory Protect interrupt is generated. If the target does not have tag = 3 or tag = 0, an Invalid Object interrupt is generated. This check is optional; if the interrupt occurs, the contents of the referenced interlock are undefined.

Pragmatic Notes

Interlock Function Compatibility with other Architectures

The interlock mechanism has been specified for Level Alpha in such a way that the functions of the interlock operators can be emulated by sequences of this architecture code using operators that are common to this and prior implementations. Thus, a system can simultaneously run programs that use the new operators and programs that emulate their functions, perhaps interacting with the same interlock objects. However, programs compiled specifically to run on processors at Level Alpha or higher should use only the interlock operators, with the semantics outlined above, and should not be dependent upon the detailed internal structure of the interlock objects. It should be possible for later architectures to redefine the mechanism and the data structure, while retaining the operator semantics.

The operators in this group can effect the following net status transition upon an interlock:

Free to Locked__Uncontended (by LOK and LOKC):

The owner__id field is set to the SNR value, the lock__control field is copied from the prior interlock value, the locked__bit and not__free__bit are both set to 1, and the tag is set to 3.

Locked__Uncontended to Free (by UNLK):

The lock__control field is copied from the prior interlock value, the tag is set to 3, and the remainder of the word is set to zero.

The operators in this group can effect the following temporary status transition upon an interlock; the prior value must be retained internally by the processor during the operation:

Free, Locked__Uncontended, Locked__Contended, or Busy to Busy:

The owner__id field is set to the SNR value, the not__free__bit is set to 1, the tag is set to 3, and the remainder of the word is set to zero.

Use of the Busy status is an implementation option for the operators and for software. The change to Busy status must be accomplished indivisibly; the operators are subject to the same constraints defined for the RDLK operator.

If an operator (or software routine) sets an interlock to Busy and the prior status of the interlock was already Busy, the effect on the interlock is to preserve Busy status but perhaps to change the owner__id (contender) stack number. In this case, the prior value must be discarded. The transition to Busy may be repeated; such "buzzing" of the interlock may be continued indefinitely, until a non-Busy prior status is found (or the operator is aborted by a Loop Timer interrupt). (Successive accesses to a Busy interlock may require separation by an implementation-dependent delay, to avoid starvation of other memory requestors in the system – including the processor expected to un-Busy the interlock.)

If the transition to Busy discovers a prior status that is not Busy, the operator must restore the prior value, either unchanged or modified to perform a valid status transition. The restoration is accomplished with a simple write operation (with the semantics of OVRD rather than RDLK).

The LOK, LOKC, and UNLK operators may examine and (if appropriate) modify the interlock value in a single, indivisible operation, if such operation is possible in the implementation. Alternatively, these operators may effect the transition to Busy status, and then proceed:

1. If the prior status is Free (for LOK or LOKC) or Locked__Uncontended (for UNLK), the prior value is modified to the opposite status and restored to the interlock, thereby completing the operator.
2. If the prior status is Locked__Uncontended (for LOK or LOKC) or Free (for UNLK) or Locked-Contended, the unmodified prior value is restored to the interlock; the LOK or UNLK operator generates a Locking or Unlocking interrupt, respectively, or the LOKC operator reports failure to effect locking.
3. If the prior value is Busy, there are two possibilities:
 - 1) The operator can immediately generate an interrupt (LOK, UNLK) or report failure (LOKC). This action is the same as 2, above, except that the prior value is not restored to the interlock.
 - 2) The operator can continue to set the lock Busy, until the prior value is found to satisfy the predicate for situation 1 or 2, above.

When a Locking or Unlocking interrupt is generated, the reference to the interlock is passed as the interrupt P2; if the interlock reference is an NIRW, it is converted to an SIRW.

Pragmatic Notes

Implementation Options for Interlock Operators

Several implementation options are specified for these operators. If a hardware design permits atomic read-examine-modify-write operations, that mechanism may provide the most effective implementation. For other designs, the Busy status is available. The decision to generate interrupt (or indicate LOKC failure) can be taken immediately (after a single "RDLK" exchange), or the operator can "Buzz" the interlock until non-Busy status is discovered. The latter approach may lead to a slight reduction in the number of interrupts to be handled. LKID requires either buzzing or continual reading of the interlock: reading may reduce memory interference, but buzzing may permit sharing of mechanism between LKID and the other three interlock operators.

Pragmatic Notes

Interlock software conventions

The LOK and UNLK operators are defined with the following assumptions with regard to the software:

The interrupt procedure must "buzz" the interlock (using RDLK to exchange a Busy value into the interlock) until a non-Busy prior value is found.

If the Locking interrupt routine finds a Free value, it emulates the LOK action and exits.

If the Unlocking interrupt routine finds a Locked__Uncontended value, it emulates the UNLK action and exits.

If the Locking interrupt routine finds a Locked__Uncontended or a Locked__Contended value, it constructs a Locked__Contended interlock and links the contending process into a wait list. (The lock__control field is available for this purpose.) The owner__id value is preserved.

If the Unlocking interrupt routine finds a Locked__Contended value, it moves one process from the waiting to the ready list and constructs a Locked__Contended interlock (if there are remaining waiters) or a Locked__Uncontended interlock (otherwise). The owner__id of the interlock is set to the stack number of the readied process.

If the Unlocking interrupt routine finds a Free value, an error exists in the locking protocol of the program.

The owner__id value in a Busy interlock is not significant to the interlocking algorithms, but may be of diagnostic value.

LOK (lock interlock)

If the target interlock has not__free__bit = 0, the interlock status is changed to Locked__Uncontended; otherwise, a Locking interrupt is generated.

UNLK (unlock interlock)

If the target interlock has locked__bit = 1, the interlock status is changed to Free; otherwise, an Unlocking interrupt is generated.

LOKC (conditional lock interlock)

If the target interlock has `not__free__bit = 0`, the interlock status is changed to `Locked__Uncontended` and a True result is left on the stack; otherwise, a False (failure) result is left on the stack.

LKID (read interlock status)

While the target interlock status is `Busy` (`interlock.not__free__bit = 1`, `interlock.locked__bit = 0`, and `interlock.lock__control = 0`), the operator waits. When the target interlock is not `Busy`, the value of its `owner__id` field is left on the stack as a 12-bit integer.

If the result of LKID is zero, the interrogated lock was `Free`. Otherwise, the interlock was `Locked__Uncontended` or `Locked__Contended`; LKID reports the stack number of the process currently "owning" the interlock.

At implementation option, the LKID operator may examine the interlock nondestructively (by read operations) or it may effect the transition to `Busy` status and examine (and restore) the prior value. The operator must wait until the interlock status is not `Busy` (or until being aborted by a Loop Timer interrupt).

PROCESSOR STATE OPERATORS

This section deals with operators that interact with processor state, primarily the state of the currently executing code-stream and the state of the stack in which the processor is running.

Code Stream Pointer Distribution

The processor code-stream pointer is initialized by the distribution of PCW or RCW code-stream pointer components according to the following steps:

1. SDLL and SDI are set from the `sdll` and `sdi` fields of the PCW or RCW, respectively, and the referenced code-segment descriptor (CSD) is fetched by evaluating (SDLL,SDI) as an address-couple (see Address Couple Evaluation). (Note that this address-couple is evaluated in the new environment, in the case of procedure entry or exit.) If the tag of the code-segment descriptor is not 3, a Code Segment Error interrupt is generated.
2. The `pwi` and `psi` values are verified as follows. If `pwi` is not in the range $\{0 \text{ to } \text{CSD.seg_length} - 1\}$, an Invalid Index interrupt is generated, and if `psi` is not in the range $\{0 \text{ to } 5\}$, an Invalid Argument Value interrupt is generated; otherwise PWI and PSI are set from the respective field values. (These tests are optional; they may be performed on both PCW and RCW, on just the PCW, or on neither.)
3. If the CSD is present, `CSD.address` is the nominal base address of the new code-segment; the processor is conditioned to execute next from the new code-segment. If the CSD is absent, a Presence Bit interrupt is generated. Note that the code-stream pointer distribution is still completed in this case. The RCW constructed for the interrupt contains the pointer just distributed, and exit from the interrupt will complete the intended distribution from the then-present CSD.

aPRCW (distribute PCW/RCW code-stream pointer)

The common action aPRCW accomplishes the distribution of a code-stream pointer from a PCW or RCW, as described in this section.

Branching Operators

Branching operators provide for altering the processor's code-stream pointer component. They may change the point of execution in the current code-segment or establish a new code-segment with an initial point of execution.

Branches may be conditional or unconditional. Conditional branches alter the code-stream pointer or continue sequential execution depending upon the Boolean interpretation of an item in the stack.

In a conditional branch, if the branch target is not valid but the branch condition is not met, it is implementation-dependent whether the sequential execution continues or an interrupt is generated to report the invalid target.

Branching operators are classified as static or dynamic branches, as specified in the following two subsections.

Static Branches

Static branches are always to a point within the current code-segment. That point is indicated by a 2-syllable parameter. Op name stands for the particular static branch operator encoding:

op name	op_ psi	op_ pwi
	3 :	13

The high-order 3 bits of the parameter are interpreted as the new PSI value, and the low-order 13 as the new PWI value. If op__pwi is not in the range {0 to CSD.seg__length-1}, where CSD is the current code-segment descriptor, an Invalid Index interrupt is generated. If op__psi is greater than 5, an Invalid Code Parameter interrupt is generated.

BRUN (branch unconditional)

Processor registers PSI and PWI are set from the parameter, and the processor is conditioned to execute next the operator at that point in the current code-segment.

BRTR and BRFL (branch true and branch false)

The top-of-stack operand must be an operand; otherwise, an Invalid Stack Argument interrupt is generated. The top-of-stack item is interpreted as a Boolean value. If BRTR finds it to be True or BRFL finds it to be False, processor registers PSI and PWI are set from the parameter and the processor is conditioned to execute next the operator at that point in the current code-segment; otherwise, sequential execution continues.

Dynamic Branches

Dynamic branches take their code-stream pointer values from a branch destination item on top of the stack. They may branch to a computed point within the current code segment or to a point in an arbitrary code-segment.

Branching within the current code-segment is indicated if the branch destination item is an operand. It is integerized with rounding, if required, to produce a 14-bit integer. If the operand cannot be integerized, an Integer-Overflow interrupt is generated; if the integerized operand is not a 14-bit integer, either an Invalid Argument Value or an Invalid Index interrupt is generated, at implementation option. A 14-bit integer is interpreted as the code-segment index in units of half-words (3 syllables):

<code>dyn__pwi</code>	[13:13]	The new PWI value.
<code>alignment</code>	[0: 1]	The alignment bit (0 = word boundary, 1 = half word boundary)

The new PSI value is 0 (the word boundary) if the alignment bit is 0 and 3 (the half-word boundary) if the alignment bit is 1. If `dyn__pwi` is not in the range {0 to `CSD.seg__length-1`}, where `CSD` is the current code-segment descriptor, an Invalid Index interrupt is generated.

Branching to a point in an arbitrary code-segment is indicated if the branch destination item is a PCW, or an NIRW to a PCW. If `PCW.sdll` is different from the current value of `SDLL`, an Invalid Argument Value interrupt is optionally generated. The PCW code-stream pointer is distributed by invoking the common action `aPCW`, as specified in Code Stream Pointer Distribution. `PCW.control__state` is ignored.

If the top-of-stack item is not an NIRW, PCW, or operand, an Invalid Stack Argument interrupt is generated. If NIRW evaluation does not produce a PCW, an Invalid Object interrupt is generated, and if `PCW.lex__level` is not equal to `LL`, an Invalid Argument Value interrupt is generated.

All validity checking of the branch destination is optional for conditional branches. The test that `dyn__pwi` is in {0 to `CSD.seg__length-1`} is optional.

Pragmatic Notes

Dynamic branch destination operand checking

An implementation can combine the two tests on the operand value, by generating an Invalid Index interrupt if the integerized value is not in the range {0 to $2*(CSD.seg_length) - 1$ }. If an implementation opts not to apply the `seg__length` check, then Invalid Argument Value is the preferable interrupt to report that the integer value is not a 14-bit integer.

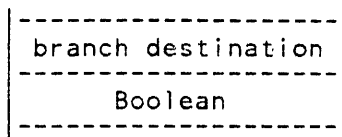
DBUN (dynamic branch unconditional)

A branch destination item is required on top of the stack. If it is an operand, the branch is within the current code-segment. If it is a PCW, or an NIRW to a PCW, the branch is to an arbitrary code-segment.

DBTR and DBFL (dynamic branch true and dynamic branch false)

The second from top-of-stack operand must be an operand; otherwise, an Invalid Stack Argument interrupt is generated. That operand is interpreted as a Boolean value. If DBTR finds it to be True or DBFL finds it to be False, a branch is executed using the top-of-stack branch destination item exactly as in DBUN. Otherwise, sequential execution continues.

The required initial stack state is:



Stack Structure Operators

Stack structure operators provide for procedure entry and exit and for changing the processor's site of activity by establishing a new running stack. They are involved in setting, saving, and restoring processor state components and linkage of activation records in the stack, both historical and lexical.

Display Update

If an implementation maintains Display registers, their contents must be maintained so that $D[i]$ contains the base address of the activation record for lexical level i , for all i in the range $\{0 \text{ to } LL\}$. Whenever the topmost activation record is changed, this invariance must be re-established, by traversing the lexical chain beginning at the base of the newly selected activation record (see common action aLXCH).

Pragmatic Notes

Display update early termination

Because the display registers model the lexical chains, and because of the tree structure implicit in the lexical chain definition, the following optimization is always possible: The lexical chain traversal and display update action is a loop through decreasing lexical levels i ; the loop can be terminated when 1) the new value for $D[i]$ is the same as the current value, and 2) i is less than the previous value of LL (prior to the value of LL being changed by the operator invoking display update).

Procedure Entry Operators

In general, executing a procedure call requires a code sequence that performs the following steps:

1. Execution of MKST (mark stack) initializes the MSCW at the base of the incipient activation record, linking it at the head of the historical chain;
2. A reference to the PCW for the procedure is pushed onto the stack (in the location the RCW will subsequently occupy);
3. Parameters to the procedure, if any, are built by operators that push items onto the stack (executed in the caller's environment);
4. Execution of ENTR (enter) completes the stack linkage in the MSCW and RCW, creating a new topmost activation record, while saving the caller's environment and instating the procedure's environment;
5. Stack building code initializes the procedure's local variables, and the procedure body is executed.

MKST (mark stack)

MKST builds an inactive MSCW on top of the stack and inserts it at the head of the historical chain. If the displacement between the new MSCW location and the base of the current stack ($S+1 - \text{BOSR}$) is outside the range $\{1 \text{ to } 2^{16}-1\}$, and the old F-BOSR displacement is in that range, a Stack Structure Error interrupt is generated; P2 is the calculated displacement value. The difference between the new MSCW location and the current value of the F register is computed as $S+1 - F$. If that difference is outside the range $\{1 \text{ to } 2^{14}-1\}$, a Stack Structure Error interrupt is generated; P2 is the calculated difference. These tests are optional for explicit MKST and IMKS operators and for MKST implied by accidental entry (aACCE); neither test may be applied to MKST implied by interrupt (aINTE).

If the calculated difference is within bounds, an MSCW is constructed with the `history__link` field set to the difference and all other fields set to zero, including the entered bit, thus marking the MSCW inactive. F is set to point to the new MSCW on top of the stack.

Pragmatic Notes

F -- BOSR limited to $2^{16}-1$

The restriction $F - \text{BOSR} < 216$ is applied to MKST so that a program cannot generate an activation record to which a lexical link cannot point. This restriction is unnecessarily harsh, in that a problem will arise only if an address-couple in this new activation record is used to generate a lexical link (by STFF or ENTR, including aINTE or aACCE). However, it is the intent of this architecture that stacks be limited to 2^{16} words, so the check in MKST is legitimate. (Field-width checks are also defined for STFF, and thus for ENTR, but they detect an improper activation record somewhat after the fact.) If the limit is enforced in MKST, the checks defined for STFF and ENTR become redundant.

The test is not applied to aINTE, and it is applied only to the first activation record above the limit, so that an operating-system interrupt procedure can invoke other procedures to deal with the error situation.

MKSN (mark-stack bound to name-call)

The operator is functionally equivalent to the MKST operator, subject to the following rules:

1. The immediate next operator in the code-stream must be NAMC (name call); otherwise, an Undefined Operator interrupt is generated. The check is optional; if an interrupt is generated, the RCW designates the MKSN operator.
2. The address-couple in the NAMC must be the initial reference for the corresponding ENTR.
3. With two exceptions, the addressing environment, reference chain, PCW, and CSD must remain the same when the corresponding ENTR is executed as when the MKSN was executed. The exceptions are:
 - 1) The reference chain does not yield a PCW. As a result of the ensuing interrupt (e.g. Binding Request), the reference chain, PCW, and CSD are subject to modification.
 - 2) Another mark-stack operation occurs, either explicitly or via an interrupt. In this case, code-segment status (absent or present at a specific address) is subject to change (but the addressing environment, reference chain, and PCW must not change).
4. Any interrupt that ENTR would generate by evaluation of the reference chain may, at implementation option, be generated instead by the MKSN operator. (In the case of an Invalid Reference Chain or Binding Request interrupt, ODI_subtype bit [12:1] is set, as though the interrupt had been from ENTR.) Any interrupt capable of interpretation as a service request must have a resumption condition of Repeat-IR; return from the interrupt procedure will repeat the MKSN operator.
5. Between the execution of the MKSN and the subsequent execution of the corresponding ENTR, the result of a LODT on the MSCW or the word above it in the stack is implementation-defined; the result of any other operation on those words is undefined. The MSCW must remain valid (tag = 3, entered = 0, history_link valid), but the other fields in the MSCW and the entire word and tag of the word above the MSCW may be defined to pass any state from MKSN to ENTR.

The observance of these rules in software is mandatory. Their enforcement in an E-mode implementation is optional. Undetected violations of the rules can lead to undefined results.

Pragmatic Notes

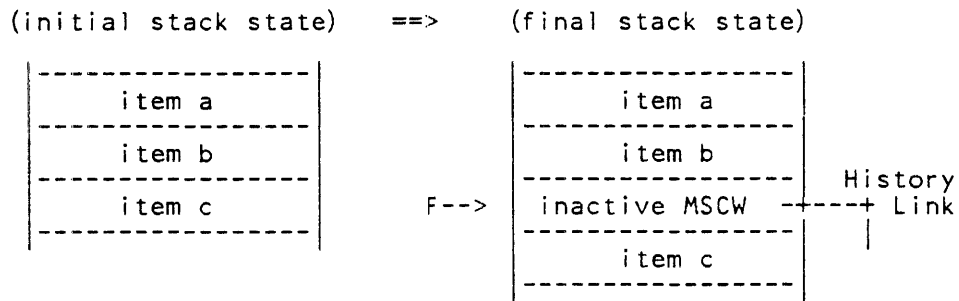
MKSN pragmatics

A correct implementation of MKSN need only perform the MKST operation followed by a NAMC operation. However, it may be an optimization for a processor to begin some of the work of ENTR in the MKSN operator. Rule 1 requires the presence of the NAMC; an implementation may treat the NAMC as a parameter of the MKSN. Rule 2 enables early examination of the reference chain. Rule 3 forbids interference from operators executed as part of the parameter-passing code. Rule 4 may make it simpler for MKSN and other invocations of ENTR to share common mechanisms. Rule 5 permits an implementation to "poison" the NIRW at F+1 to help enforce rule 3; for example, the NIRW can be given a tag other than 1. Of course, ENTR must accept the poisoned NIRW. Rule 5 also permits the F and F+1 words to contain other state being transmitted from MKSN to ENTR; one example is to put a lexical link to the PCW environment into the MSCW and the target PCW (appropriately poisoned) above it.

IMKS (insert mark stack)

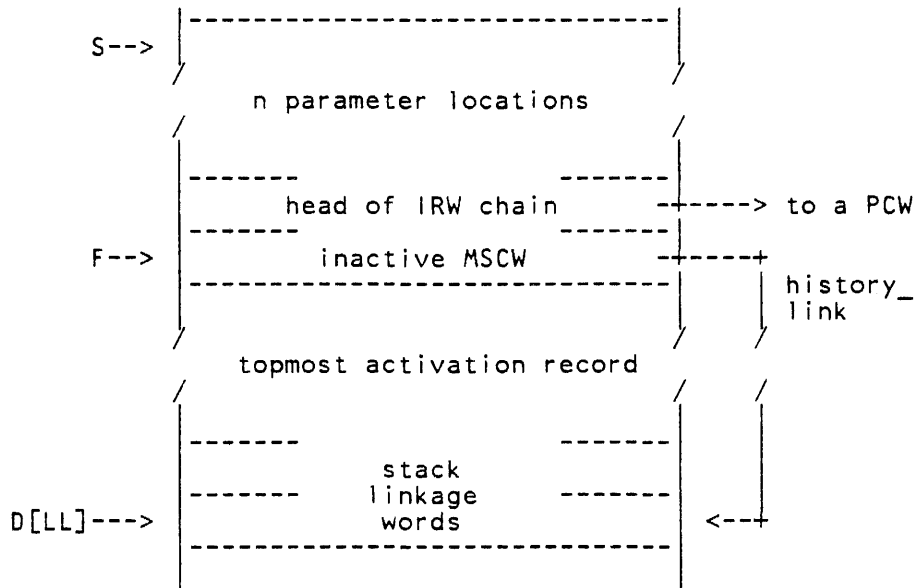
IMKS builds an inactive MSCW exactly as does MKST (mark stack), except that the new MSCW is inserted "underneath" the two top-of-stack items. IMKS produces the effect of having saved the top two stack items, deleted them from the stack, invoked MKST, and then pushed the two items back onto the stack. If there are less than two items in the expression stack at the start of an IMKS operation, a Stack-Underflow interrupt is generated.

The following diagram illustrates the stack state transformation produced by IMKS:



ENTR (enter)

The initial stack state for ENTR assumes prior execution of MKST (or IMKS). An inactive MSCW is required at the stack location addressed by F, and the head of an IRW chain to a PCW is required at the F + 1 stack location. The following diagram illustrates the initial stack state (there may be other inactive MSCWs on the historical chain between F and D[LL]):



If the item addressed by F does not have a tag of 3 or its entered bit is 1 (indicating an entered MSCW), or if the top-of-stack address is less than or equal to F, a Stack Structure Error interrupt is generated. If the F+1 stack location is not the head of an IRW chain, an Invalid Stack Argument interrupt is generated. If IRW chain evaluation produces a DD with element_size = 7, a Binding Request interrupt is generated. Otherwise, if IRW chain evaluation does not produce a PCW, an Invalid Reference Chain interrupt is generated (see IRW chains for a definition of IRW chain evaluation). If PCW.invalid_ll = 1, an Invalid Argument Value interrupt is generated.

ENTR consists of the following functional tasks:

1. Complete the MSCW, inserting it at the head of the appropriate lexical chain.
2. Construct an RCW to save the current processor code-stream pointer and Boolean accumulators.
3. Initialize processor state for the procedure being entered, including code-stream pointer and addressing environment.

Completing the MSCW

If PCW.lex_level > 0, the activation record containing the PCW is the immediately global addressing space (global AR) of the procedure being entered. ENTR forms a Lexical Link (stack_number, displacement) to address the base of the global AR; this Lexical Link is inserted into the MSCW to complete the lexical chain that defines the addressing environment of the new procedure. (Note that if PCW.lex_level = 0, there is no global AR; in this case, the value of the Lexical Link is undefined.)

The global AR is identified by the form of reference to the PCW (the final reference if an IRW chain is evaluated). If the reference is an NIRW, the global AR is the activation record at level NIRW.Lambda in the addressing environment at invocation of ENTR: a Lexical Link to that AR is constructed by implicit invocation of the STFF operator. If NIRW.lambda is unequal to PCW.lex_level - 1, an Invalid Argument Value interrupt is generated. If the reference is an SIRW, its Lexical Link (stack_number, displacement) points directly to the global AR. If the word at the base of the global AR is not an entered MSCW, a Stack Structure Error interrupt is generated; if its lex_level value is unequal to PCW.lex_level - 1, an Invalid Argument Value interrupt is generated.

PCW.lex_level is copied into MSCW.lex_level, MSCW.entered is set to 1, and the completed MSCW is stored back at the F stack location. Note that MSCW.history_link is not altered by ENTR.

Constructing the RCW

Processor state values stored in the RCW are the Boolean accumulators (TFFF, OFFF, EXTF, and FLTF), the processor code stream pointer (SDLL, SDI, PSI, PWI), CS (control state), and LL. For explicit ENTR, the code-stream pointer designates the syllable following ENTR. For accidental entry (ENTR invoked by aACCE), the code-stream pointer designates the operator invoking aACCE. For interrupt entry (ENTR invoked by aINTE), the code-stream pointer is determined by the specific interrupt situation.

The restart indicator RCW.rs is set to 0 by an explicit ENTR; it is set to 1 only in some of the interrupt (aINTE) and accidental (aACCE) entry cases for which the entire entry sequence (MKST...ENTR) is performed together. (For these implicit invocations, the value of rs is determined by the invoking operator.) The only cases of accidental entry that must set the rs bit are for value call, STAD and STAN. The interrupt cases that must set the rs bit are noted in the descriptions of the operators and interrupts. Implementations may define further uses of rs.

An implementation may use the `exit__opt` field in the RCW to enable optimizations in the procedure exit operators. Any implementation must be such that `exit__opt = 0` means that no optimization is to be performed: software assignment of 0 to the `exit__opt` field in an RCW is valid, and is required in any activation record whose environment might be changed by explicit alteration of stack linkages.

The RCW is stored at the `F+1` stack location.

Initializing the Processor State

`LL` is set from `PCW.lex__level` and `D[LL]` is set from `F` to address the base of the activation record. The processor CS Boolean is set from `PCW.control__state`.

Any applicable display registers are updated to reflect the new addressing environment: If the PCW reference was an NIRW, no update is necessary, since the global AR of the new activation record is already in the addressing environment. If the PCW reference was an SIRW, display update begins at `D[LL-1]`, the new global AR.

The expression stack is appended to the new activation record, making any procedure parameters accessible.

The processor code-stream pointer state is initialized from the PCW by invocation of the common action `aPRCW`, as discussed in Code Stream Pointer Distribution.

Pragmatic Notes

Pragmatics of `exit__opt`

There are several conditions that can be noticed at procedure entry and used to optimize procedure exit. The applicability of a particular optimization depends on the processor implementation. In the following, the prefix Caller refers to a value prior to ENTR or subsequent to exit; Callee refers to a value subsequent to ENTR and prior to exit.

If $D[\text{CallerSDLL}] = D[\text{CalleeSDLL}]$, the Code Dictionary activation record did not change, so code-stream pointer distribution from the RCW can proceed prior to display update. Note that this equality holds trivially if procedure entry was by means of an address-couple with $\text{Lambda} \geq \text{CallerSDLL}$.

For entry by means of an address-couple with $\text{Lambda} < \text{CallerLL}$, EXIT or RETN needs to restore only display registers `D[i]` for

$$\text{CalleeLL} \leq i \leq \text{CallerLL}.$$

If $\text{Lambda} = \text{CallerLL}$, not even `D[CallerLL]` is changed at exit.

`aACCE` (accidental entry)

The common action `aACCE` is invoked by some reference-chain evaluation operators to "evaluate" a PCW. The action is automatic invocation of the parameterless procedure (function) defined by the PCW; it is defined as three steps:

1. Invoke MKST.
2. Place the reference to the PCW on the stack (at `Mem[F+1]`), and
3. Invoke ENTR.

The action is subject to the interrupts of MKST and ENTR, although some error situations, such as no unentered MSCW at Mem[F], are prevented by the close coupling of MKST and ENTR, and any potential Binding Request or Invalid Reference Chain situation would have been handled already by the invoking operator.

The accidental entry and subsequent return leave the processor executing the operator that initiated the accidental entry. (Unlike explicit ENTR, which builds an RCW referencing the successor operator, aACCE points the RCW at the invoking operator.) The invoking operator provides the value for RCW.rs. In particular, value-call operators, STAD, and STAN set rs to 1 to cause re-entry in restart state.

aINTE (interrupt entry)

The common action aINTE is invoked to generate an interrupt, which is implemented as an entry to an MCP procedure whose PCW, or an SIRW chain thereto, is located by the fixed address-couple (0,3). Two parameter words are provided to the procedure.

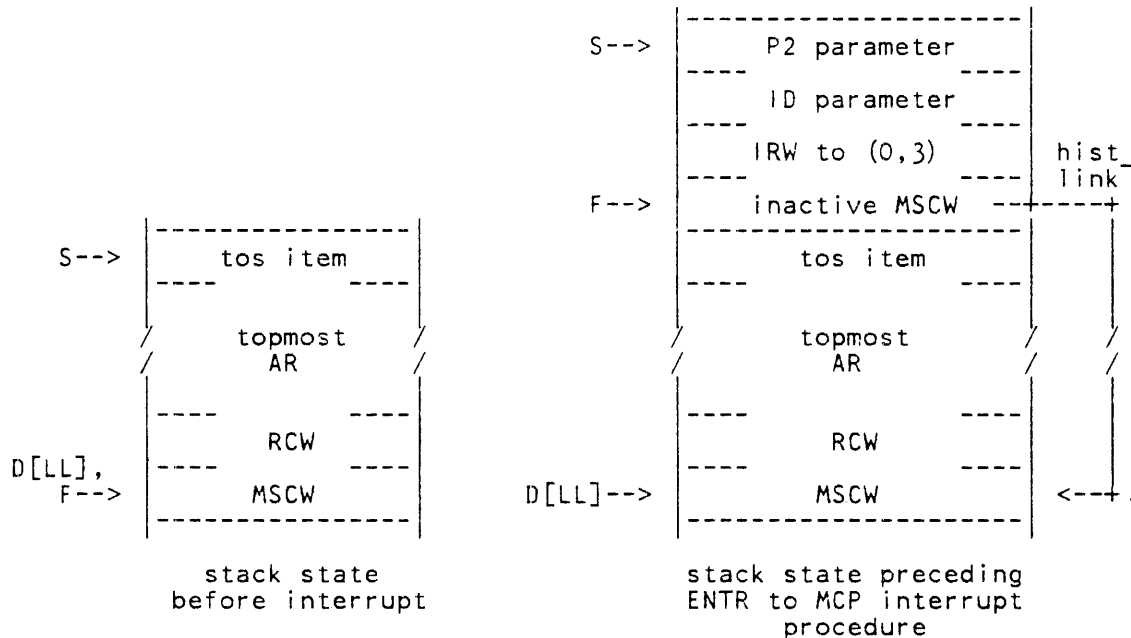
The action can be defined as four steps:

1. Invoke MKST.
2. Place an NIRW to (0,3) on the stack (at Mem[F + 1]).
3. Place the two parameter words on the stack.
4. Invoke ENTR.

The aINTE action may be invoked by operators (operator-dependent interrupts), between operators (external interrupts), or spontaneously at any time (alarm interrupts). The invoking mechanism determines the contents of the parameters and of the resulting RCW.

The action is subject to the interrupts of MKST and ENTR, although some error situations, such as no unentered MSCW at Mem[F], are prevented by the close coupling of MKST and ENTR. If an interrupt is generated during the aINTE action, the stack will contain the four words inserted by aINTE: the inactive MSCW, interrupt reference, and two parameter words. Note that if interrupt entry generates another interrupt (Invalid Reference Chain, Binding Request, or Invalid Argument Value) because the interrupt reference is not usable, the new interrupt will surely fail for the same reason; successive recursive interrupts will cause the processor to halt as described in Superhalt.

The following diagram illustrates the stack state transformation produced by the interrupt entry sequence (F is shown pointing to the same activation record as D[LL] in the initial state, but that is not required):



Stack state transformation produced by interrupt entry

The effect is undefined of a Presence Bit or Stack Structure Error interrupt during a display update, because the processor cannot know whether or not D[0] would be altered, thus redefining (0,3). Software must avoid any move (such as ENTR, EXIT/RETN, or MVST) into an activation record whose lexical chain traverses an absent stack.

Pragmatic Notes

Interrupt with unusable lexical chain

Given that a display update has failed (or a lexical chain is in error), it is reasonable to assert that D[0] has not (yet) moved and can be used to locate (0,3). Note, however, that if the word at D[0]+3 is a PCW and the lexical chain cannot be traversed from D[LL] to D[0], the architecture does not fully define the immediate global environment for the interrupt procedure: the base of that environment is at D[0], but the number of the containing stack is not known. It has been suggested that a processor could attempt to find the stack number of the level-0 environment by reading the stack_number field of the MSCW at D[1], the MSCW at D[0], or the PCW at D[0]+3. The first option fails if it is the D[1] to D[0] link that is bad; the other two would require a software convention, because E-mode places no requirement on stack_number in a level-0 MSCW or a PCW. Such an implementation-defined extension of E-mode Level Alpha could be appropriate to improve robustness in error handling. It is also appropriate to superhalt in such cases.

Procedure Exit Operators

There are two operators for deleting an activation record and returning execution to the prior topmost activation record. RETN (return) assumes termination of a function and leaves the top-of-stack item as a result, whereas EXIT assumes termination of a procedure and does not leave a result.

EXIT (exit)

EXIT deletes the topmost activation record from the stack and restores processor state for the prior topmost activation record. The base location of the topmost activation record is addressed by D[LL], and the prior topmost activation record is identified by the first entered MSCW on the historical chain whose head is D[LL].

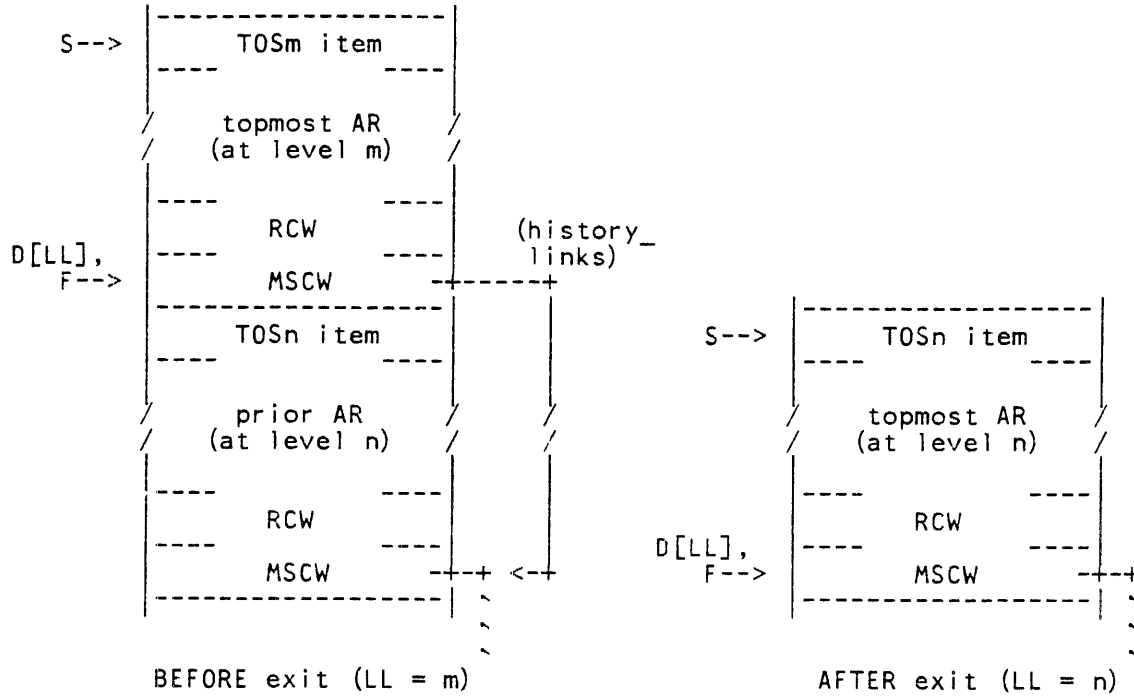
If the tag of the D[LL] or D[LL]+1 item is not 3, a Stack Structure Error interrupt is generated. If the RCW addressed by D[LL]+1 has block__exit = 1, a Block Exit interrupt is generated. Otherwise, the base of the prior topmost activation record is located by following the historical chain from D[LL] until the first entered MSCW is encountered. If a history__link is evaluated and found to be zero, or to point to a location less than or equal to BOSR, or if the tag of a stack item addressed by a history__link is not 3, or if the lex__level field of the first entered MSCW is not equal to the lex__level field of the RCW in the initial topmost activation record, a Stack Structure Error interrupt is generated.

The topmost activation record is deleted from the stack by setting the top-of-stack pointer, S, to D[LL]-1. F is reset to address the first MSCW on the historical chain whose head is D[LL], whether or not it is entered. LL is set from the value saved in the RCW. D[LL] is reset to address the base of the prior topmost activation record. Remaining processor state is reset by distributing values saved in the RCW at the initial D[LL]+1 stack location. The Boolean processor accumulators (TFFF, OFFF, EXTF, FLTF) and CS (control state) are reset from their saved values in the RCW. Any applicable display registers are updated to reflect the new addressing environment: the lexical chain is traversed beginning at D[LL].

The processor code-stream pointer is initialized from the RCW by invoking the common action aPRCW, as discussed in Code Stream Pointer Distribution. If RCW.rs = 1, the processor is conditioned to execute in restart state the operator addressed by the new code-stream pointer.

Unless specific optimization information is recorded by ENTR in the RCW.exit__opt field, the RCW.sdll component cannot be interpreted until the environment change (and any display update) is complete.

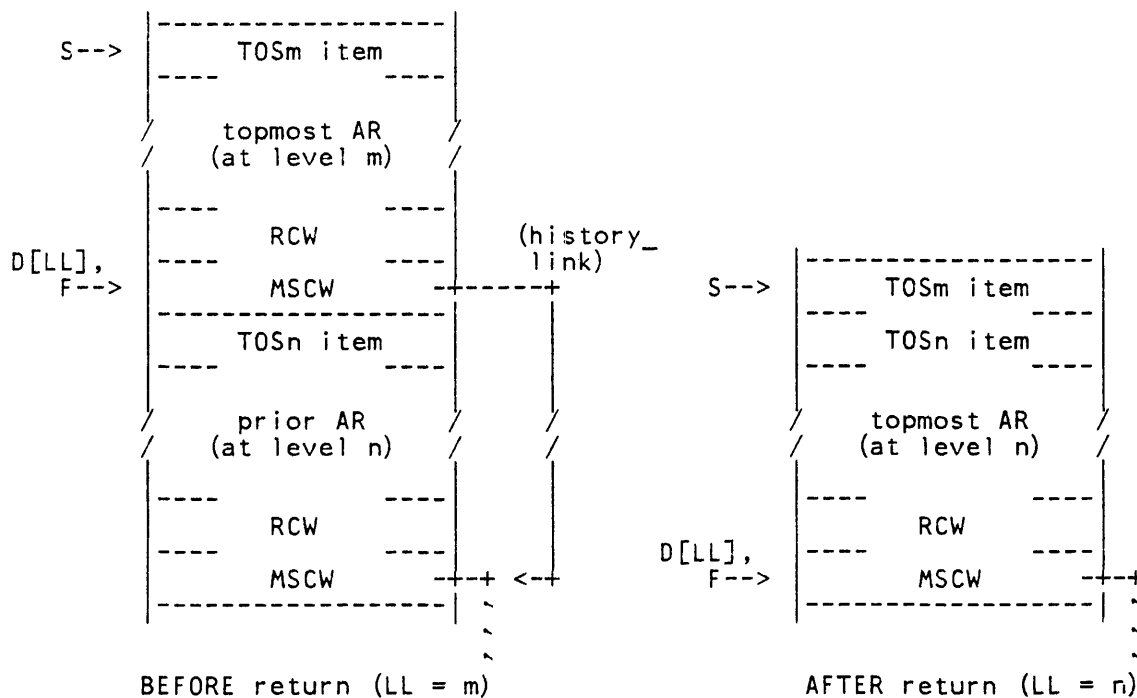
The following diagram illustrates the stack state transformation produced by EXIT. In the initial and final states, F is shown pointing to the same activation record addressed by D[LL], but no such coincidence is required. Note that LL is typically different before and after the EXIT.



RETN (return)

RETN is exactly the same as EXIT, except that it assumes that the terminated activation record is a function and that the initial top-of-stack item is to be the result of the function. RETN therefore retains the initial top-of-stack item and pushes it back onto the top of the stack after the topmost activation record is deleted. If the top-of-stack item is an NIRW, an Invalid Stack Argument interrupt is generated.

The following diagram illustrates the stack state transformation produced by RETN. In the initial and final states, F is shown pointing to the same activation record addressed by D[LL], but no such coincidence is required. Note that LL is typically different before and after the RETN.



Stack Environment Operator

MVST (move to stack)

MVST changes the processor's site of activity by deactivating the current stack and activating a destination stack. A new memory addressing environment is also specified. A Top of Stack Control Word (TSCW) stored at the base location of an inactive stack is used to save the height of the stack and a link to the start of the historical chain. These two fields within the TSCW are sufficient to activate the stack.

MVST requires a single-precision operand on top of the stack to specify the stack number of the destination stack and the new environment number; otherwise an Invalid Stack Argument interrupt is generated. The operand contains two fields:

- [23:12] destination environment number
- [11:12] destination stack number

MVST consists of the following functional tasks:

1. Deactivate the current stack by writing a TSCW at its base.
2. Change addressing environment and identify the destination stack.
3. Restore processor stack state.
4. Update lexical environment state.

Deactivating the Current Stack

If S-BOSR is outside the range $\{1 \text{ to } 2^{*}16 - 1\}$, or S-F is outside the range $\{1 \text{ to } 2^{*}14 - 1\}$, a Stack Structure Error interrupt is generated (both tests are optional). Otherwise, MVST builds a TSCW by setting the stack height field to the value S-BOSR, setting the SF__disp field to the value S-F, setting the tag to 3, and setting the rest of the word to zero. The TSCW is stored at the base of the stack (addressed by BOSR).

Changing the Addressing Environment and Identifying the Destination Stack

If the destination environment number does not exceed the container size for ENR, it is loaded into the register; otherwise an Invalid Argument Value interrupt is generated. (If an implementation does not provide multiple environments, the container size for ENR is zero and the only value that may be assigned to ENR is zero; in this case, the test is optional. Another option when the ENR container size is zero is to define the MVST argument as a 12-bit integer, with aISX invocation for violation.) When ENR is loaded, the addressing environment is changed; the new stack may be in a different memory component from the old one.

SNR is set from the destination stack number. The stack descriptor identified by the destination stack number is fetched as specified by the "Stack References" section. If the stack number is not valid, an Invalid Index interrupt is generated. If the stack descriptor is marked not present, a Presence Bit interrupt is generated.

If an interrupt is detected during step 2 or 3, the processor is not running on a valid stack. Therefore, instead of generating the interrupt, the processor immediately generates a superhalt condition. (If a superhalt occurs, the Interrupt__Count value is undefined; otherwise, the Interrupt__Count value is unchanged by MVST.)

Restoring Destination Stack State

If the TSCW in the destination stack does not have tag = 3, a Stack Structure Error interrupt is generated. The following processor registers are loaded in the specified order, or its equivalent:

1. BOSR \leftarrow Stack descriptor.address
2. LOSR \leftarrow Stack descriptor.length + BOSR
3. S \leftarrow TSCW.stack __height + BOSR
4. F \leftarrow S - TSCW.SF__disp.

If the computed value of F is less than or equal to BOSR, a Stack Structure Error is generated.

The proc__id value (as a 3-bit integer) is stored at the base word of the destination stack.

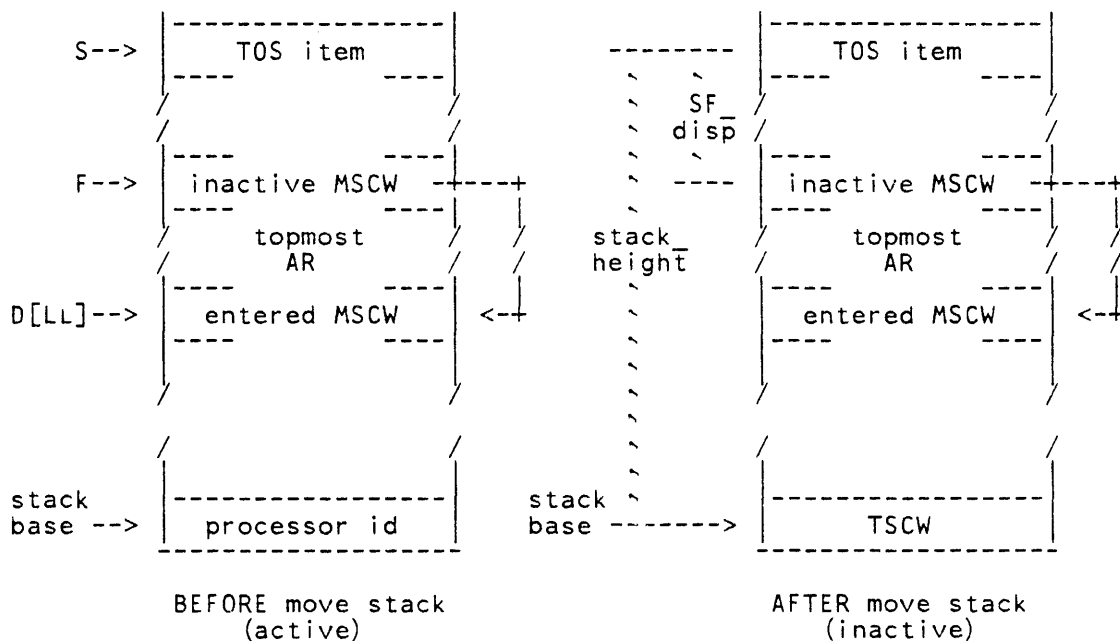
Updating the Lexical Environment State

D[LL] is set to point to the first entered MSCW (MSCW.entered = 1) on the historical chain whose head is F. If, in following the historical chain, a history__link is encountered that points to a location less than or equal to BOSR, or if the tag of a stack item addressed by a history__link is not 3, or if the lex__level field of the first entered MSCW is not equal to LL, a Stack Structure Error interrupt is generated.

Any other appropriate display registers are updated to reflect the new address environment: the lexical chain is traversed beginning at D[LL].

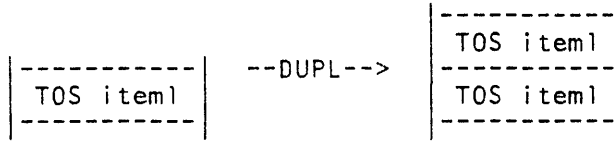
If an interrupt is detected during 4. , the interrupt RCW points to the operator following MVST.

The following example illustrates the current stack transformation produced by MVST after the destination stack number has been deleted from the stack. The transformation of the destination stack is essentially the inverse of that of the current stack. An inactive MSCW between S and D[LL] is illustrated but atypical.



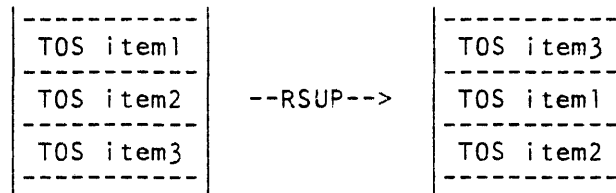
DUPL (duplicate top-of-stack)

DUPL requires one item on top of the stack, creates an exact duplicate of it, and leaves both items on top of the stack:



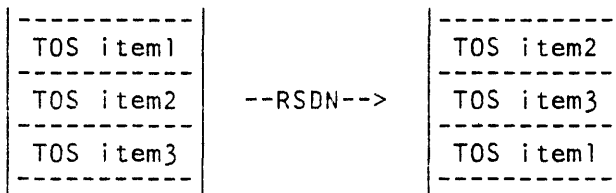
RSUP (rotate stack up)

RSUP requires three top-of-stack items. The third from top item is "rotated up" to become the top of stack item:



RSDN (rotate stack down)

RSDN requires three top-of-stack items. The top item is "rotated down" to become the third from top-of-stack item:



Processor-State Manipulation Operators

These operators are classified as read state, set state, or read and set state functions. The operators in each class are described in the following paragraphs.

Read state operators place processor state register values on top of the stack and mark the stack state items valid. Set state operators bring values to the top of the stack and set processor state to match the values brought to the top of the stack. Read and set state operators are similar to read state operators, except that the processor state is reset at the conclusion of the operation.

Read State Operators

RTFF (read true-false flip-flop)

RTFF leaves on top of the stack the value of TFFF as a Boolean operand, True or False.

RSNR (read SNR)

RSNR leaves on top of the stack a 12-bit integer containing the SNR value.

Pragmatic Notes

RSNR is equivalent to LT8 53, RPRR

The RSNR operator has the same function as RPRR with a stack argument of 53. RSNR provides a migration path away from the use of RPRR except in limited contexts of low-level code.

WHOI (read processor id)

WHOI leaves on top of the stack a 3-bit integer containing the processor identification number, `proc__id`.

WATI (read machine identification)

WATI leaves on top of the stack a double-precision operand containing information about the level of implementation. It consists of the `proc` state (except for `proc__id`), formatted into the following fields:

First Word:

[47:32] `unit__id` or 0
[15: 4] `E-mode__level`
[11: 4] `E-mode__features` (default: 0)
[7: 8] `machine__type`

Second Word:

[47: 4] `page__size__indicator`
[43: 4] 0 (reserved)
[39:40] `microcode__version` or 0

Implementations return zero in fields for which the defined data are not relevant.

Pragmatic Notes

Format of `microcode__version` is defined by software

This architecture defines `microcode__version` as a 40-bit value to be returned by the WATI operator by implementations that use field-loadable microcode. The content of this field is not of functional concern to operators of this architecture; it is a matter of convention between the software that creates the value (the microcode compiler) and the software that reads it (the operating system and any program to which the information is made available). The following example convention is used for A9 systems, when operating software at the mark 3.4.740 release level:

- [39:08] `mark__level`: The "mark level" of the release (34).
- [31:16] `cycle`: The generation cycle of the release (740).
- [15:16] `creation`: The creation date, computed by means of the following equation:

$$(\text{year} - 1970) * 1000 + \text{day_of_year}$$

RTOD (read time of day clock)

RTOD leaves on top of the stack a 36-bit integer containing the value of the time of day clock. The range of values is {0 to $2^{36}-1$ } in units of 2.4 microseconds.

RPRR (read processor register)

RPRR requires one 6-bit integer stack argument; otherwise Integer Subset Exception action (aISX) is invoked. The argument is interpreted as a processor register identification (register id), and the result left on top of the stack is the value of the specified register. This result value is a k-bit integer, where k is the width of the target register.

Readable processor registers are associated with register ids that are a subset of integers in the range {0 to 63}. If the register id is not a valid value, an Invalid Argument Value interrupt is generated. See the table under SPRR for the valid register IDs and widths.

The value reported for the S register is the address of the top-of-stack item after the RPRR argument has been consumed.

RIPS (read internal processor state)

The RIPS operator is provided to read implementation-defined processor state. RIPS accepts a single-precision argument and leaves a single-precision value. The implementation must specify the allowable argument values, any validity checking, the form and meaning of the output values, and the semantics of the operator, including any interrupt generation.

Pragmatic Notes

Implementation-defined low-level operators

The RIPS, WIPS, REMC, and WEMC operators are defined as to opcode and stack argument/result number and type, but not semantically. They exist to facilitate access to machine state at a level of abstraction below that of the architecture functional definition. It is within the spirit of this specification for an implementation to use such operators to perform diagnostic, maintenance, initialization, or configuration functions, for example. It is contrary to that spirit to use such operators to extend the architecture functionality at the level of abstraction of this specification. RIPS is a new operator in this architecture (and the B7900).

Set State Operators

SXSN (set external sign flip-flop)

SXSN requires an operand on top of the stack; otherwise, an Invalid Stack Argument interrupt is generated. SXSN sets EXTF (external sign flip-flop) to the value of bit 46 of the top-of-stack item. The operand is left unchanged on the stack.

EEXI (enable external interrupts)

EEXI conditions the processor to respond to external interrupts and resets the processor CS Boolean to 0 (normal state). If any external interrupt is pending when EEXI is executed (in control state), an external interrupt occurs immediately following the EEXI operator, even if the immediate successor operator is DEXI.

DEXI (disable external interrupts)

DEXI conditions the processor to ignore all masked external interrupts and sets the processor CS Boolean to 1 (control state).

SINT (set interval timer)

SINT arms the interval timer and sets it from an operand on top of the stack. If the item on top-of-stack is not an 11-bit integer, Integer Subset Exception action (aISX) is invoked. Otherwise, the Interval_Timer is set to the specified value and armed.

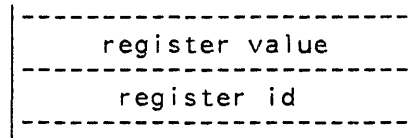
WTOD (write time of day clock)

WTOD sets the time of day clock from an operand on top of the stack. If the item on top of the stack is not a 36-bit integer, Integer Subset Exception action (aISX) is invoked.

SPRR (set processor register)

SPRR assigns a value to a register; it requires two stack arguments. The second argument is the register id; it must be a 6-bit integer or else Integer Subset Exception action (aISX) is invoked. The top argument is the register value; it must be a k-bit integer (where k is the width of the destination register), or else Integer Subset Exception action (aISX) is invoked. If the top argument fits within the k-bit width specified in this architecture, but is too large for the container actually implemented, an Invalid Argument Value is generated; see the "Processor State" appendix. The contents of the specified register are set to the register value.

The required initial stack state is:



Settable processor registers are associated with register ids that are a subset of integers in the range {0 to 63}. If the register id is not a valid value, an Invalid Argument Value interrupt is generated.

The following table specifies the decimal register id encodings, register names, validity for RPRR and SPRR, and register widths (in bits).

Register ID	Register Name	RPRR	SPRR	Width
0	D[0]	yes	yes	20
1 to LL-1		invalid	invalid	
LL	D[LL]	yes	yes	20
LL+1 to 35		invalid	invalid	
36	LOSR	yes	yes	20
37	BOSR	yes	yes	20
38	F	yes	yes	20
39-51		invalid	invalid	
52	S	yes	yes	20
53	SNR	yes	yes	12
54-56		invalid	invalid	
57	(reserved)	invalid	invalid	
58	ENR	yes	yes	0-12
59-63		invalid	invalid	

Assignment of a value to D[LL] does not invoke immediate display update. After the assignment, access by means of a global address-couple (with lambda < LL) is undefined unless the new D[LL] value addresses a proper MSCW that links to the same immediate global activation record as did the old value.

If the register id designates S, the new register value remains in S at the completion of the SPRR operator (the arguments to SPRR are consumed before the register assignment is made).

Pragmatic Notes

Beware manipulating S

Because S is the address of the top-of-stack, expressions and assignments involving S can generate counterintuitive results. For example, a statement to increment S could be written "S:= S+1" and compiled in a straightforward way to produce the following architecture code:

```
LT8      52    % Register id for SPRR
LT8      52    % Register id for RPRR
RPRR
ONE
ADD
SPRR
```

Because the SPRR register id is already on the expression stack, RPRR returns a value one higher than the top-of-stack address prior to execution of this statement. Therefore, the net effect of the statement is to increment S by two, not one.

RUNI (indicate running)

RUNI sets the Running_Indicator.

WIPS (write internal processor state)

The WIPS operator is provided to write implementation-defined processor state. WIPS accepts two single-precision arguments and leaves no result. The implementation must specify the allowable argument values, any validity checking, and the semantics of the operator, including any interrupt generation.

Pragmatic Notes

Implementation-defined low-level operators See note under RIPS.

ZIC (zero Interrupt_Count)

The ZIC operator sets the Interrupt_Count to zero.

Read and Set State Operator

ROFF (read and reset overflow flip-flop)

ROFF leaves on top of the stack the Boolean value of OFFF (overflow flip-flop) and then unconditionally resets OFFF to false.

DATA ARRAY OPERATORS

Operators in this group perform functions on arrays specified by data descriptor stack arguments. The functions applied generally consist of sequential processing of one or more arrays of word or character elements. Termination occurs when an element length has been exhausted or when some condition is satisfied.

Data in one of the argument arrays may be modified, or the arrays may be processed in order to produce a result, or both actions may occur. Results are indicated by items left on top of the stack or by the setting of one or more processor Boolean accumulators.

Array operators typically accept IndexedDDs as arguments or accept unindexed DDs and index them. For an operator to access the data, the actual segment must be present; see Descriptor Interpretation.

Searching Operators

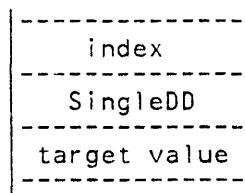
There are two searching operators. LLLU (linked list lookup) searches an explicitly linked list for the first element whose data component is greater than or equal to a target value, and SRCH (masked search for equal) searches an implicitly ordered list (backwards) for the first word that is bitwise equal to a target value after both the word and target value have been masked.

LLLU (linked list lookup)

LLLU processes an array as an explicitly linked list and applies the following interpretation to each word in the array (ignoring the tag):

LLLU__data	[47:28]	The atomic data component
LLLU__link	[19:20]	The link component (an index from the base of the array to the next element in the list)

LLLU requires an initial index, an unindexed unpagged copy SingleDD, and a target value on top of the stack:



The index and target value must be operands and are integerized with rounding if required. If either the index or the target value is not an operand or the second from top-of-stack item is not an unpagged unindexed copy SingleDD, an Invalid Stack Argument interrupt is generated. If the index or target value cannot be integerized, an Integer-Overflow interrupt is generated. If the SingleDD is not marked present, a Presence-Bit interrupt is generated. Whenever an index value is used to fetch a word from the list, if it is not in the range $\{0 \text{ to } \text{DD.length} - 1\}$, an Invalid Index interrupt is generated.

The initial index is applied to the SingleDD, and the first word of the list is fetched. Starting with that word, LLLU applies the following iterative loop.

If the link component equals zero, a `single__integer -1` is left on top of the stack to signal failure. If the link is non-zero and the data component is greater than or equal to the absolute value of the target value, the operator terminates; otherwise the link value becomes the new index and the iteration is repeated. If termination occurs in the first iteration, the initial index is returned on the stack as a 20-bit integer. If termination occurs on a subsequent iteration, the index used on the previous iteration is returned as a 20-bit integer.

Pragmatic Notes

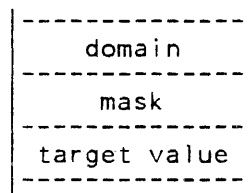
LLLU Pragmatics

For convenient use of the operator, the list should be constructed so that the potential targets are found in the second through the penultimate elements; then the stack result is the index of the element whose link points to the target element. If the target inequality is satisfied on the first element, the result is ambiguous, there being no prior element. If the target inequality is satisfied on the last (`link = 0`) element, the failure result is returned.

SRCH (masked search for equal)

SRCH scans an array called the "domain" (an actual segment), from an indexed starting point back towards the base, for a word that matches a target value in selected bits. Any or all of the 52 bits (word and tag) may be matched. The result is a `single__integer`: the index of the matching word, or `-1` if the search fails.

SRCH requires three arguments on top of the stack: the search domain (a `SingleDD`), the mask, and the target value:



The domain argument must be an `IndexedSingleDD` or an unindexed unindexed copy `SingleDD`; otherwise an `Invalid Stack Argument` interrupt is generated.

If the domain is an unindexed descriptor with non-zero length, it is indexed by its length-1 to form an `IndexedSingleDD`. If it is an unindexed `DD` with zero length, no search occurs and the failure result (`-1`) is left on the stack.

Both the mask and target value are bit vectors of length 52. The target value is logically ANDed with the mask before it is used for comparison. The second word of any double-precision target value or mask is ignored.

The word referenced by the `IndexedSingleDD` is logically ANDed with the mask and compared to the (masked) target value. If a matching word is found, its index is left on the stack as the SRCH result. If there is no match and the actual segment index is non-zero, the index is decremented by one and the search continues. If there is no match and the index is zero, the actual segment is exhausted: the failure result is left on the stack.

If the actual segment is a page, the SRCH result index is relative to the page, not the array (virtual-segment).

Pointer Operators

Pointer operators deal with sequences of word or character elements in data arrays. There are pointer operators for scanning, transferring, comparing, and editing the element sequences in various ways; various operators deal with only a source or a destination sequence, with both a source and a destination sequence, or with two source sequences. Sequential processing terminates when an element length is exhausted or, in some cases, when a condition is satisfied before the length is exhausted. Some pointer operators, called enter-edit operators, serve to invoke one or more instances of another class of pointer operators, called edit-mode operators.

Typical pointer operators require initial stack arguments that specify the length, the source element sequence, and the destination element sequence. A source element sequence can be contained in an operand or referenced by an indexed data descriptor; a destination sequence is always referenced by an indexed data descriptor. A source or destination descriptor defines the first element of the target sequence.

Some operators (pack, input convert, string isolate) read a source and generate a result operand on the stack; these stack results are not defined to be destinations.

The term "pointer operator" reflects the fact that the source or destination descriptor is usually a Pointer (indexed character descriptor). In fact, indexed word descriptors are also acceptable, although they are usually coerced to Pointers.

If an EBCDIC (or hex) Pointer has a `char__index` value outside the range {0 to 5} (or {0 to 11}), an Invalid Argument Value interrupt is generated.

For transfer-words operations, the tag of each source word is transferred to the destination. For all other pointer operations, the tag of each destination word is preserved.

`element__size` conventions

The element size for either the source or destination sequence can be specified by the `element__size` in the Pointers, inferred by default, or fixed by the operator. Only the translate operator can manipulate source and destination sequences of different element sizes. The word-transfer operators always operate on single words, but accept indexed descriptors of any valid `element__size`. The unpack operators require a source operand and treat it as hex.

For all pointer operators except word-transfer, the following element-size adjustments are made: If there are both source and destination arguments, and only one is a Pointer, the `element__size` of the Pointer is applied to the other argument. If there is no Pointer argument, then any source or destination is assumed to be EBCDIC. When a character `element__size` is applied to a source or destination word descriptor, that argument effectively becomes a Pointer; if the index is greater than $2^{16}-1$, an Invalid Index interrupt is generated.

For word-transfer operators, the `element__size` specified in the source and destination descriptors is unaltered, but the operation deals with single words. If a Pointer is used as source or destination, and the `char__index` is non-zero, the descriptor is adjusted by setting `char__index` to zero and incrementing `word__index`; if the resulting `word__index` exceeds $2^{16}-1$, an Invalid Index interrupt is generated.

For all pointer operators except word-transfer and translate, an Invalid Stack Argument interrupt is generated if the source and destination arguments are both Pointers and the `element__size` values are not equal.

Length Argument

When a pointer operator is invoked in initial state, the specifications in the following paragraphs apply. If a pointer operator is resumed in restart state, the length argument must be a 20-bit integer, or else the action is undefined. (In some restart situations, a length of zero is significant.)

The length argument must be an operand; otherwise an Invalid Stack Argument interrupt is generated. It is integerized with rounding if required. If it cannot be integerized, an Integer-Overflow interrupt is generated. If the integer value exceeds $2^{20} - 1$, an Invalid Argument Value interrupt is generated.

A negative length value is equivalent to zero. If $\text{length} \leq 0$, all pointer operators terminate without accessing any source element or transferring any destination element; no Paged Array interrupt is generated; service or error interrupts based upon the other arguments may be generated or not, at implementation option. (The enter-single-edit operators do not themselves terminate for zero-length input; rather, any edit operators that require a length do so.)

A Paged Array interrupt is not generated after the length has been exhausted, or by compare-delete operators after a mismatch has been detected, or by scan or conditional transfer operators after a source character has failed to satisfy the condition.

Source Argument

The source argument must be an operand or an IndexedDD of any valid element__size; otherwise an Invalid Stack Argument interrupt is generated.

A source operand is interpreted according to element__size conventions defined above as an EBCDIC sequence of 6 (or 12) characters or a hex sequence of 12 (or 24) characters, for a single-or double-precision operand, respectively. The operand is logically concatenated with itself as required to form an indefinite length sequence.

Except for word transfer overwrite, all pointer operations generate a Paged Array interrupt if an odd-tagged word is read by means of a source pointer.

Short-Source Operators

The string-isolate, pack, and input-convert operators are special in that the source sequence is short (never more than 25 characters) and the result is left on the stack as an operand rather than being moved to a destination sequence. Some of these operators interpret one character or one zone field as a sign.

If a short-source operator is executed in initial state and then generates a Paged Array interrupt after the sign or any data character has been read, the RCW.rs bit is set to 1 and the stack is updated to the restart configuration: the updated length, the updated source and the partial result are put on the stack. The length is above the source; the relative position and form of the partial result are implementation-defined. The effect of resuming the operator in restart state is undefined if the content or top-of-stack position of the partial-result item has been altered.

It is never necessary to set RCW.rs to 1 when the Paged Array interrupt is generated while attempting to fetch the first character in the source sequence. For operators that interpret the first character as a sign, it is necessary that $\text{RCW.rs} = 0$ when the interrupt occurs fetching the first character.

Because the source character sequences for these operators are short, so the operation can legitimately cross only one page boundary, and because these operators have no destination that might be subject to enlargement, it is not necessary to have a repeat resumption condition when the operator is begun in restart state (having already encountered one page boundary). The stack configuration and resumption condition is implementation-defined if a short-source operator is executed in restart state and then generates a Paged Array interrupt.

Destination Argument

The destination argument must be an IndexedDD of any valid element__size; otherwise an Invalid Stack Argument interrupt is generated.

Except for word transfer overwrite, all pointer operations generate a Paged Array interrupt if either a read or write access is attempted to an odd-tagged destination word.

An operator is said to "require a destination" if a destination stack argument is specified, except for the Enter Single Edit operators; these operators "require a destination" if the subsequent edit operator is a move, insert, or end-float operator.

If a destination pointer is marked read__only, operators that require a destination generate a Memory Protect interrupt; the interrupt is optional if the initial length ≤ 0 .

Source1 and Source2 Arguments

The compare operators process two sources, rather than a source and a destination. Source2 and Source1 are treated as defined above for Source and Destination, respectively; the compare operators are said not to "require a destination".

Overlapping Source and Destination

A source and destination are said to overlap if both arguments are IndexedDDs into the same segment and the displacement (index difference) between them is less than the effective length (number of elements transferred).

The effect of an overlapped unconditional word or character-transfer depends upon the direction and magnitude of the displacement. In the following, D represents the displacement expressed as destination element index minus the corresponding source element index. L represents the transfer length. N is 0, 8, or 16 for word, 8-bit, or 4-bit elements, respectively.

- $D \leq -L$: No overlap
- $-L < D \leq 0$: Destination sequence overwrites $L+D$ source elements.
- $0 < D < N$: Destination and source contents are undefined.
- $N \leq D < L$: D source elements are repeated throughout destination.
- $L \leq D$: No overlap

Conditional (character-relational or set-membership) transfer operators are subject to the same constraints as unconditional transfers, but note that when $D > 0$ the only opportunity for conditional termination is within the first D elements. L in the foregoing specification is the number of elements actually transferred.

For translate involving 4-bit characters, the displacement d is reckoned in 4-bit characters from the initial source to the initial destination. For translation of L characters, the overlap cases and effects are as follows (where $/2$ indicates halving with truncation):

4-to 4-bit:	$-L < d \leq 0$: $L+d$ source elements overwritten $0 < d < L$: Undefined
4-to 8-bit:	$-2L < d \leq 1 - L$: $\max(L, 2L+D)$ source elements overwritten $1-L < d < L$: Undefined
8-to 4-bit:	$-L < d < 0$: $(L+d+1)/2$ source elements overwritten $0 \leq d < L$: Source sequence overwritten $L \leq d < 2L$: Undefined
8-to 8-bit:	$-L < D \leq 0$: $L+D$ source elements overwritten $0 < D < L$: Undefined

For edit operators, the source and destination overlap if $-L_d < D < L_s$, where L_d and L_s are the destination and source length, respectively. (Note that for data-transferring edit operators, $L_d \geq L_s$.) The result of overlapped editing is undefined if $D > L_s - L_d$. For a table-edit sequence, each group of consecutive edit operators (other than skips) is to be considered a unit for the application of this test. Overlap considerations do not apply directly to skip operators, but skip operators in an edit-table change the initial pointer displacements for subsequent operators, so one operator's destination element might become another operator's source element.

Pragmatic Notes

Overlap Pragmatics

For conditional and unconditional transfer operators, the three overlap cases are:

Destination first: move the data "down" (toward lower addresses)

Destination equals source: effective no-op

Source first: "smear" destination with repetitions of the source

Smearing occurs when destination elements become subsequent source elements; smearing works for word transfers or for character transfers beyond a minimum displacement, but not for the translate or edit operators.

Translation differs from simple transfer in that the transferred characters are modified (so smearing is not defined), and the source and destination element sizes may differ. Except for translate in place to the same or smaller element size, overlapping translate operations must be performed with great care.

Edit operators also can modify the transferred characters, and can transfer more characters to the destination than from the source. Overlapping edit operations is recommended only for simple editing in place, as in using MINS to suppress leading zeros.

Update Of Pointer-Operator Arguments

Most pointer operators occur in both "delete" and "update" variations. The "delete" forms consume all of their stack arguments and do not leave updated results on the stack (although they may leave other results on the stack). The "update" forms leave on top of the stack an updated reference(s) to the source and destination (if applicable), and the length (if termination is possible before the length is exhausted). (The SISO and SHOW operators have only a delete form; the TRNS and EXPU operators have only an update form.)

A pointer operator can be interrupted (for example, at a page boundary); in which case the length, source, and destination are updated to the point of the interrupt. Both update and delete operators are subject to such interruption; the stack arguments are configured for resuming the operator (in initial or restart state, depending upon the situation).

An updated length result is a 20-bit integer indicating the number of elements remaining to be processed at termination or interrupt. It is produced by update operators that may terminate before the length is exhausted, or by operators that are interrupted and can be resumed. If the initial length is negative, the updated length is zero.

An updated source operand is the original operand circularly rotated left such that the left-justified element is the next element that would have been processed if termination or interrupt had not occurred.

A source or destination descriptor is updated as an indexed descriptor that references the next source or destination element that would have been processed had termination or interrupt not occurred. The updated descriptor reflects any adjustments made according to the `element_size` conventions defined above: Word-transfer operators may adjust the `char_index` and `word_index` values; all other operators change any indexed word descriptor into a Pointer.

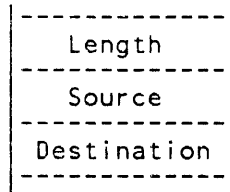
If the initial length ≤ 0 for an update operator, or if update is caused by an interrupt prior to transfer of any data, the input arguments left on the stack may or may not be modified. For example, a length < 0 may have been replaced by 0, and `element_size` changes may have been effected.

The field-width limits in a descriptor are $2^{*}16 - 1$ for the `word_index` field in a Pointer and $2^{*}20 - 1$ for the `index` field in an indexed word descriptor. If the word index value to be updated into a descriptor exceeds the limit, an Invalid Index interrupt is generated. If the update was being done to report another interrupt, the Invalid Index is reported instead. An implementation may generate the interrupt at any point in the sequence processing where the word index would exceed the limit.

Unconditional Character-Transfer Operators

Unconditional character-transfer operators transfer hex or EBCDIC characters from the source to the destination. The number of characters transferred is specified by the length. TFFF is left in an undefined state.

The required initial stack state is:



The following operator leaves no results on the stack:

TUND (transfer characters unconditional delete)

The following operator leaves the updated source on top of the the stack and the updated destination second from top of the stack:

TUNU (transfer characters unconditional update)

Character-Relational Operators

Character-relational operators sequentially apply a relational comparison of each source character to a delimiter character supplied by a stack argument until the length is exhausted or a relation fails. TFFF indicates the cause of termination: it is reset to 0 if a relation fails and set to 1 if the length is exhausted (all source characters satisfy the relation).

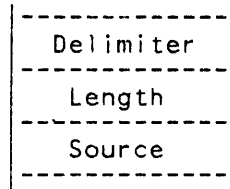
The delimiter argument must be a single-precision operand; otherwise an Invalid Stack Argument interrupt is generated. It is interpreted as a single right-justified character (EBCDIC or hex according to the effective element size of the source); all bits in the delimiter word except those in the delimiter character itself are ignored.

The binary value of each source character is compared to the binary value of the delimiter character. The operator names specify the relation of source character to delimiter that must hold for the operation to continue. For example, the SLSU operator scans across source characters less than the delimiter.

Scan Operators

Character-relational scan operators sequentially compare each source character to the delimiter character as defined above.

The required initial stack state is:



The following operators leave no results on the stack:

- SGTD (scan while greater delete)
- SGED (scan while greater than or equal delete)
- SEQD (scan while equal delete)
- SNED (scan while not equal delete)
- SLED (scan while less than or equal delete)
- SLSD (scan while less than delete)

The following operators leave the updated length on top of the stack and the updated source second from top of the stack:

- SGTU (scan while greater update)
- SGEU (scan while greater than or equal update)
- SEQU (scan while equal update)
- SNEU (scan while not equal update)
- SLEU (scan while less than or equal update)
- SLSU (scan while less than update)

Transfer Operators

Character-relational transfer operators sequentially compare each source character to the delimiter character as defined above. Each source character that satisfies the relation is transferred to the destination sequence.

The required initial stack state is:

Delimiter
Length
Source
Destination

The following operators leave no results on the stack:

- TGTD (transfer while greater delete)
- TGED (transfer while greater than or equal delete)
- TEQD (transfer while equal delete)
- TNED (transfer while not equal delete)
- TLED (transfer while less than or equal delete)
- TLSD (transfer while less than delete)

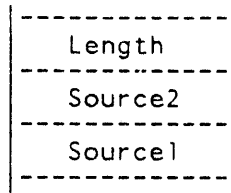
The following operators leave the updated length on top of the stack, the updated source second from top of the stack, and the updated destination third from top of the stack:

- TGTU (transfer while greater update)
- TGEU (transfer while greater than or equal update)
- TEQU (transfer while equal update)
- TNEU (transfer while not equal update)
- TLEU (transfer while less than or equal update) TLSU (transfer while less than update)

Character-Sequence Compare Operators

Character-sequence compare operators apply a relational comparison of the source1 sequence to the source2 sequence. TFFF is set to 1 if the relation is satisfied and reset to 0 if the relation fails.

The required initial stack state is:



The binary values of each corresponding source1 and source2 character are compared. The two sequences are equal if and only if each source1 character is equal to the corresponding source2 character for the specified length (or the initial length is zero). Source1 is strictly less (greater) than source2 if and only if for the first (left-most) pair of unequal characters, the source1 character is strictly less (greater) than the source2 character.

The following operators terminate when the actual relation is determined. No result is left on the stack.

- CGTD (compare characters greater delete)
- CGED (compare characters greater than or equal delete)
- CEQD (compare characters equal delete)
- CNED (compare characters not equal delete)
- CLED (compare characters less than or equal delete)
- CLSD (compare characters less than delete)

The following operators terminate only when the length is exhausted. If a Paged Array interrupt is taken after the relation (TFFF state) has been determined, RCW.rs is set to 1, so that TFFF is not modified when the operator is resumed. They leave the updated source on top of the stack and the updated destination second from top of the stack. The updated Pointers reference the first character after the end of the sequence as determined by the length:

- CGTU (compare characters greater update)
- CGEU (compare characters greater than or equal update)
- CEQU (compare characters equal update)
- CNEU (compare characters not equal update)
- CLEU (compare characters less than or equal update)
- CLSU (compare characters less than update)

Character Set-Membership Operators

Character set-membership operators test source characters for membership in a character set supplied by a stack argument. The relations applied consist of inclusion and exclusion, and source characters are sequentially tested until the relation fails or the length is exhausted. TFFF indicates the cause of termination: it is reset to 0 if a relation fails and set to 1 if the length is exhausted (all source characters satisfy the membership criterion).

The character set argument must be an IndexedSingleDD; otherwise an Invalid Stack Argument interrupt is generated. This IndexedSingleDD locates the first word of the character set. The actual segment addressed by the IndexedSingle DD must be long enough to contain the referenced word and, for EBCDIC source, the next seven words. This requirement is not directly enforced, but if an odd-tagged word is encountered in the set table, a Memory Protect interrupt is generated.

The character set is interpreted as a bit vector indexed by the source character. If the selected bit is 1, the character is included in the set; otherwise it is excluded from the set. The bit is located by the address equation:

$$\text{Mem}[\text{set. address} + \text{set. index} + \text{WordIndex}(c)]. [\text{BitIndex}(c):1]$$

WordIndex and BitIndex are computed from the binary representation of the source character (c) as follows:

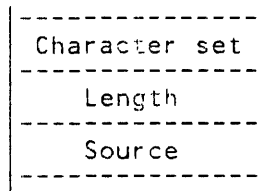
EBCDIC#WordIndex	value of 3 high-order bits
#BitIndex	31 - (value of 5 low-order bits)
hex #WordIndex	0
#BitIndex	31 - (4 bit value)

In the following operator names, the relation "while source included in set" is called "while true", and "while excluded from set" is called "while false".

Scan Operators

Character set-membership scan operators apply the sequential membership test of each source character to the character set as defined above.

The required initial stack state is:



The following operators leave no results on the stack:

SWTD (scan while true delete)
SWFD (scan while false delete)

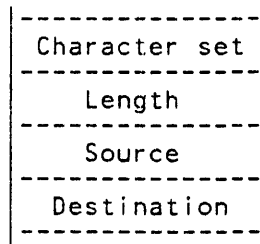
The following operators leave the updated length on top of the stack and the updated source second from top of the stack:

SWTU (scan while true update)
SWFU (scan while false update)

Transfer Operators

Character set-membership transfer operators apply the sequential membership test of each source character to the character set as defined above. Each source character that satisfies the membership criterion is transferred to the destination sequence.

The required initial stack state is:



The following operators leave no results on the stack:

TWTD (transfer while true delete)
TWFD (transfer while false delete)

The following operators leave the updated length on top of the stack, the updated source second from top of the stack, and the updated destination third from the top of the stack:

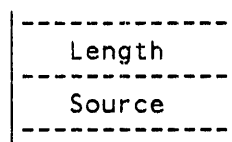
TWTU (transfer while true update)
TWFU (transfer while false update)

Character-Sequence Extraction Operator

SISO (string isolate)

SISO extracts a character sequence from the source, creates an operand containing the extracted sequence right-justified with leading zero-fill (if required), and leaves the operand on top of the stack. The length specifies the number of characters in the extracted sequence.

The required initial stack state is:



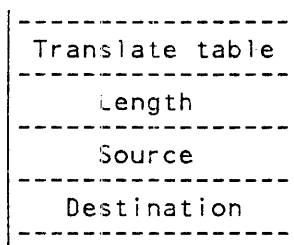
The result may be a single- or double-precision operand depending on the length and the source character type. If the source is EBCDIC, the result is single for length ≤ 6 and double for {7 to 12}. If the source is hex, the result is single for length ≤ 12 and double for {13 to 24}. An Invalid Argument Value interrupt is generated if the source is EBCDIC and length > 12 or if the source is hex and length > 24 .

Character Translate Operator

TRNS (translate)

TRNS sequentially accesses characters from the source sequence, maps each character into a specified character set, and stores the translated character into the destination sequence. The character set mapping is indicated by a translate table argument.

The required initial stack state is:



The translate table must be an IndexedSingleDD; otherwise an Invalid Stack Argument interrupt is generated. This IndexedSingleDD locates the first word of the translate table. The actual segment addressed by the IndexedSingleDD must be long enough to contain the referenced word and the next 3 or 63 words for a hex or EBCDIC source, respectively. This requirement is not directly enforced, but if an odd-tagged word is encountered in the translate table, a Memory Protect interrupt is generated.

The translate table is interpreted as an array of words, each containing 4 right-justified 8 bit characters. It is indexed by the source character, and the selected 8 bit character is stored into an EBCDIC destination, or the 4 low-order bits of the character are stored into a hex destination. The character is located by the address equation:

$$\text{Mem}[\text{table. address} + \text{table. index} + \text{WordIndex}(c)]. [\text{FieldIndex}(c):8]$$

WordIndex and FieldIndex are computed from the binary representation of the source character (c) as follows:

EBCDIC	WordIndex	= value of 6 high-order bits
	FieldIndex	= 31 - 8*(value of 2 low order bits)
hex	WordIndex	= value of 2 high-order bits
	FieldIndex	= 31 - 8*(value of 2 low order bits)

TRNS leaves the updated source on top of the stack and the updated destination second from top of the stack.

Decimal-Character-Sequence Operators

Decimal-character-sequence operators interpret hex or EBCDIC sequences as decimal sequences, and provide conversion functions among various decimal representations. (Hex-sequence representations of decimal data are often called Binary Coded Decimal, BCD.)

A decimal digit is represented as a four-bit binary integer in the range {0 to 9}; a digit sequence is an unsigned sequence of decimal digits. (A value in the range {hex "A" to hex "F"} is a "nondigit".) Digit sequences can be represented as operand values or in hex or EBCDIC character sequences.

In an operand, a sequence of n digits is represented as a sequence of adjacent 4-bit fields, right-or left-justified according to the operator. Up to 12 or 24 digits can be contained in a single-or double-precision operand, respectively; the second word of a double holds the low-order digits.

The hex representation of a digit sequence is as a hex sequence of the corresponding digit values. The EBCDIC representation of a digit sequence is as an EBCDIC sequence in which the numeric field (low-order four bits) of each character contains a digit value. The high-order four bits are called the zone field; zone fields are significant in some operators, but they do not form part of the digit sequence.

A signed decimal integer is represented as a digit sequence and a sign value. Hex "D" represents a negative sign; any other 4-bit value represents a positive sign. The sign may be placed at either the left (high-order) or right (low-order) end of the sequence. A signed sequence of n digits is represented in hex as a sequence of $n + 1$ characters; the sign is the leftmost or rightmost character. A signed n -digit sequence is represented in EBCDIC as a sequence of n characters; the sign occupies the zone field of the leftmost or rightmost character. (Operand decimal sequences are always unsigned; EXTf can be used to hold the sign.)

There are three groups of decimal digit-sequence pointer operators. (See also the arithmetic operators BCD and DBCD, which produce a digit sequence from a binary integer.) The pack operators transform a hex or EBCDIC source sequence into a right-justified operand digit sequence. The unpack operators transform a left-justified operand digit sequence into a hex or EBCDIC destination sequence. The input-convert operators are similar to pack, but the integer value of the source sequence is transformed to binary representation.

Two pairs of operators, PACD/PACU and ICVD/ICVU, treat a hex source sequence as either unsigned or left-signed, depending upon the value of the first character: a nondigit value is taken as a sign; a digit value is taken as a digit. The sign is not counted in the length.

Pragmatic Notes

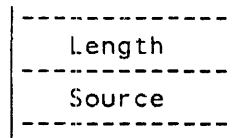
"Old" and "new" decimal-sequence operators

The operators PACD, PACU, UABD, UABU, USND, USNU, ICVD, and ICVU are a set of "old" operators (introduced on the B6500). The operators PKUD, PKLD, PKRD, UPUD, UPUU, UPLD, UPLU, UPRD, UPRU, ICUD, ICLD, and ICRD constitute a set of "new" operators. UPUD and UPUU are UABD and UABU renamed; the others were introduced into this architecture. The new operators provide a complete set of unambiguously unsigned, left-signed and right-signed options. The old operators provide only left hex sign and right EBCDIC zone sign, and the PACx and ICVx operators are data-driven with respect to the presence or absence of a sign character in a hex sequence.

Pack Operators

Pack operators perform a conversion from the source EBCDIC or hex decimal sequence to a decimal operand containing the corresponding digit sequence right-justified with leading zero-fill. The operand is left as a result on the stack. Nondigits in a hex source sequence (other than a sign character) or in the numeric field of an EBCDIC source sequence are transferred unmodified to the operand sequence.

The required initial stack state is:



The result is a single-precision operand if length ≤ 12 and double-precision for {13 to 24}. If length > 24 , an Invalid Argument Value interrupt is generated.

The following operators leave the decimal result on top of the stack.

- PKUD (pack unsigned delete)
- PKLD (pack left-signed delete)
- PKRD (pack right-signed delete)
- PACD (pack delete)

The following operator leaves the updated source on top of the stack and the decimal result second from top of the stack.

- PACU (pack update)

PKUD leaves EXTF and TFFF in undefined states. All other pack operators set both EXTF and TFFF: true = negative and false = positive or unsigned; if the length argument is ≤ 0 , EXTF and TFFF are reset (false).

If length > 0 and the source is hex and there is a sign, the number of characters read from a hex sequence is one greater than the length value; a sign is always present for PKLD and PKRD; and never present for PKUD; a sign is present for PACx when the leftmost hex character is a nondigit. If length ≤ 0 , no source characters are read and the digit-sequence result is zero.

Following are the sign locations for these operators:

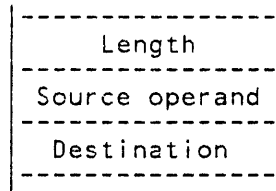
PKUD:	none
PKLD, EBCDIC:	zone of leftmost character
PKLD, hex:	leftmost character
PKRD, EBCDIC:	zone of rightmost character
PKRD, hex:	rightmost character
PACx, EBCDIC:	zone of rightmost character
PACx, hex:	leftmost character if nondigit, else none

If the PKRD operator is resumed in restart state with a hex source, the operator continues even if length = 0 (in which case the sign character is yet to be fetched). If a left-signed operator is resumed in restart state, the sign has already been determined; this concern applies to PACD or PACU with a hex source and to PKLD with any source.

Unpack Operators

Unpack operators interpret the source operand as a left-justified digit sequence and store the corresponding hex or EBCDIC decimal sequence into the destination. Nondigits in the operand sequence are transferred unmodified to the hex characters or the numeric field of the EBCDIC characters in the destination sequence.

The required initial stack state is:



The element__size convention for unpack is that the source operand is unconditionally treated as hex; if the destination is an IndexedWordDD, it is changed to an EBCDIC Pointer. If the source is not an operand, an Invalid Stack Argument interrupt is generated. If length > 24, an Invalid Argument Value interrupt is generated.

Unpack-Unsigned Operators

Unpack-unsigned operators store the destination decimal sequence without sign. For an EBCDIC destination, the zone field of each character is set to hex"F". For a hex destination, the digit sequence is stored with no sign character.

The following operator leaves no results on the stack:

UPUD (unpack unsigned delete)

The following operator leaves the updated source operand on top of the stack and the updated destination second from top of the stack:

UPUU (unpack unsigned update)

Unpack-Signed Operators

Unpack-signed operators store the destination decimal sequence with a sign; the sign is determined by EXTf (external sign flip-flop), where true = negative and false = positive.

Hex "C" and "D" are used as the positive and negative sign characters respectively. For an EBCDIC destination, the sign is inserted into the zone field of the rightmost or leftmost character, depending upon the operator, and all other zone fields are set to hex "F". For a hex destination and length > 0, the sign is inserted as the leftmost or rightmost character, depending upon the operator; length + 1 hex characters are transmitted to the destination sequence. If length ≤ 0, no characters are transmitted to the destination. The operator mnemonics and names are listed below with the location of the sign for hex and EBCDIC sequences.

If a Paged Array interrupt is generated after the sign is inserted, by a USNx operator with a hex destination or by a UPLx operator, the RCW.rs bit is set to 1. When resumed in restart state, these operators ignore the sign (becoming, in effect, an unpack-unsigned operator.)

If a Paged Array interrupt is generated by a UPRx operator in attempting to store the sign character into a hex destination, the RCW.rs bit is set and the length argument is updated to zero. When resumed in restart state with a length = 0 and a hex destination, these operators proceed to store the sign. These operators do not require the use of restart state except for the hex length = 0 case.

The following operators leave no results on the stack:

	HEX	EBCDIC
UPLD (unpack left-signed delete)	left	left
UPRD (unpack right-signed delete)	right	right
USND (unpack signed delete)	left	right

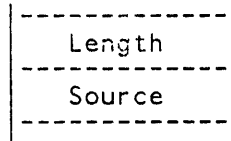
The following operators leave the updated source operand on top of the stack and the updated destination second from top of the stack:

	HEX	EBCDIC
UPLU (unpack left-signed update)	left	left
UPRU (unpack right-signed update)	right	right
USNU (unpack signed update)	left	right

Input-Convert Operators

Input-convert operators perform a conversion from the source EBCDIC or hex decimal sequence to a numeric operand containing the signed integer value of the corresponding digit sequence. The operand is left as a result on the stack. TFFF and EXTF are left in undefined states.

The required initial stack state is:



If the length > 23, an Invalid Argument Value interrupt is generated. If the integer absolute value of the source decimal sequence is less than 8**13, a single__integer is produced; otherwise a double__integer is produced.

The following operators leave the integer result on top of the stack.

- ICUD (input convert unsigned delete)
- ICLD (input convert left-signed delete)
- ICRD (input convert right-signed delete)
- ICVD (input convert delete)

The following operator leaves the updated source on top of the stack and the integer result second from top of the stack.

- ICVU (input convert update)

If length > 0 and the source is hex and there is a sign, the number of characters read from a hex sequence is one greater than the length value. A sign is always present for ICLD, ICRD, and never present for ICUD; a sign is present for ICVx when the leftmost hexadecimal character is a nondigit. If length ≤ 0, no source characters are read and the binary integer result is positive zero.

The sign locations for these operators are the same as for the corresponding pack operators:

- | | |
|---------------|---|
| ICUD: | none |
| ICLD, EBCDIC: | zone of leftmost character |
| ICLD, hex: | leftmost character |
| ICRD, EBCDIC: | zone of rightmost character |
| ICRD, hex: | rightmost character |
| ICVx, EBCDIC: | zone of rightmost character |
| ICVx, hex: | leftmost character if nondigit, else none |

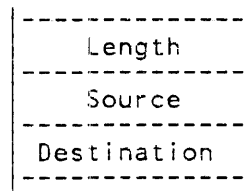
If any character in the source decimal sequence is not a decimal digit (see Decimal character-sequence operators), the result value is undefined, except that a sequence of all hex "F" characters is equivalent to a sequence of nines.

If the ICRD operator is resumed in restart state with a hex source, the operator continues even if length = 0 (in which case the sign character is yet to be fetched). If a left-signed operator is resumed in restart state, the sign has already been determined; this concern applies to ICVD or ICVU with a hex source and to ICLD with any source.

Word-Transfer Operators

Word-transfer operators transfer word elements from the source to the destination. The number of words is specified by the length. Source tags are transferred.

The required initial stack state is:



A source operand is interpreted as a word or pair of words logically concatenated with itself indefinitely.

Source and destination Pointers, if not already word-aligned, are advanced to the next word boundary (see `element__size` conventions under Pointer Operations).

Word-Transfer-Protected Operators

A word transfer operation is performed as defined above.

The following operator leaves no results on the stack:

TWSD (transfer words delete)

The following operator leaves the updated source on top of the stack and the updated destination second from top of the stack:

TWSU (transfer words update)

Word-Transfer-Overwrite Operators

A word-transfer operation is performed as defined above. Source words are transferred to the destination regardless of tag value (a Paged Array interrupt cannot occur).

The following operator leaves no results on the stack:

TWOD (transfer words overwrite delete)

The following operator leaves the updated source on top of the stack and the updated destination second from top of the stack:

TWOU (transfer words overwrite update)

Primitive Display Operator

SHOW (primitive display)

The SHOW operator displays a sequence of characters on an external device (subject to implementation restrictions) without using the normal input/output system. SHOW requires two arguments, length (on top) and source. If the source is a descriptor, it must be an EBCDIC Pointer or an IndexedWordDD (which is coerced to an EBCDIC pointer); a hex pointer causes an Invalid Stack Argument interrupt to be generated.

The operator causes min(length,implementation bound) characters to be transmitted from the source to a visible display; any excess characters are ignored.

An upper bound on length is implementation-defined; acceptable values are 0 or ≥ 24 . Any implementation that has no display mechanism will define the bound as zero; the SHOW operator can then be implemented as equivalent to DLET twice (any type checking on the arguments is then optional).

The SHOW source must be entirely contained within one actual segment. If an odd-tagged source word is encountered, a Memory Protect interrupt is generated.

A display of any length, including zero, entirely removes any prior message. Each display persists until replaced by a subsequent display or destroyed by human action or some implementation-defined occurrence. (For example, an implementation may share display facilities between primitive display and normal Operator-Display-Terminal function, in which case ODT output can overwrite primitive output.) If separate processors simultaneously attempt primitive displays on a multiprocessor system, the effect is undefined. (At implementation option, there may be separate or shared display facilities.)

The characters to be displayed are represented in EBCDIC. The following 44 characters, plus space, must be displayed with recognizable graphics:

ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789 ,./+ -= ()

The display of any other EBCDIC non-control character is implementation-dependent. The effect of EBCDIC control characters is undefined.

Pragmatic Notes

SHOW operator is for low-level code

The SHOW operator is applicable to very low-level code, such as bootstraps, operating-system initialization, and diagnostic procedures. Depending upon the implementation, the SHOW operator may be quite slow; it is not intended for routine use on a running system. (It is, of course, fast enough to avoid Loop Timer interrupts.) The SHOW source may not include a page boundary.

Edit Operators

Edit-mode operators can be considered sub-operators invoked by a special class of pointer operators, the enter-edit operators. Most edit operators process source or destination characters sequentially until a length is exhausted.

Enter-Edit Operators

There are two modes in which edit operators are executed; each is initiated by an "enter edit" operator. The enter edit operator provides the source and destination. It may specify update, which causes a reference to the destination and source (if applicable) to be left on top of the stack at termination of edit-mode.

Table edit-mode

A sequence of edit operators is executed until terminated by ENDE (end edit). Each acts on the source and destination supplied by the table enter edit operator, and length is a parameter for each edit operator requiring it. If update is specified, the updated source and destination are left on the stack by ENDE.

Each edit operator that uses the source/destination updates it internally at termination, so that a group of edit operators may sequentially process source/destination characters. Character-skip operators may advance or back up the source/destination to alter the normal sequential processing.

Single edit-mode

A single edit operator acts on the source and destination supplied by the enter single edit operator. Length is also supplied as a stack argument at entry, whether or not it is required by the edit operator. If update is specified, the updated source and destination are left on the stack at termination of the edit operator.

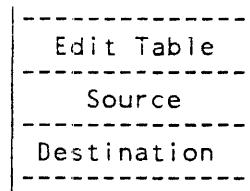
If a particular operator does not use the source or the destination argument, any tests on the argument type are optional for that operator for the unused argument(s). An unused argument may be modified by update action; for example, an IndexedWordDD may be changed to a Pointer.

All enter-edit operators except EXPU set FLTF = 0 when executed in initial state; all leave FLTF unchanged when resumed in restart state.

Enter-Table-Edit Operators

Enter-table-edit operators supply the source and destination sequences, and a reference to the sequence, or table, of edit operators to be executed. Each edit operator acts on the source or destination supplied at entry, and length is a parameter for each edit operator requiring it.

The required initial stack state is:



If the edit-table argument is not an IndexedDD, an Invalid Stack Argument interrupt is generated. The data descriptor is interpreted as usual, except that the element_size field is ignored and the index field is subdivided as follows:

	[39: 1]	zero
esi	[38: 3]	Edit table syllable index of the first edit operator
	[35: 3]	zero
ewi	[32:13]	Edit table word index of the word containing the first edit operator

If field [39:1] or [35:3] is non-zero, the results are undefined. If the esi field is not in the range {0 to 5}, an Invalid Argument Value interrupt is generated. If the descriptor is not an indexed DD, an Invalid Stack Argument interrupt is generated. Otherwise, edit operators (and their parameters) are fetched from the edit-table, starting from the esi syllable of the ewi word, until completion of an ENDE (end edit) operator. The normal code stream is then resumed with the operator following the enter table-edit operator.

If execution is attempted of an edit-table word that does not have a tag of zero, an Invalid Program Word interrupt is generated. If an ENDE is not encountered before the table array page is exhausted, the odd-tagged word that is required to follow the page causes an Invalid Program Word interrupt.

In the case of a Paged Array interrupt, an operator executed in table-edit-mode must invoke restart action if the FLTF state is true or if the operator has traversed one or more characters or if the interrupt occurred transferring data to the destination. To invoke restart action, the operator sets RCW.rs to 1 and updates the stack to the restart configuration of the enter-table-edit operator: the updated length is on the stack in addition to the the updated table descriptor, updated source pointer, and the updated destination pointer. The updated length is either the topmost or the second argument, as specified by the implementation. The length argument (as a 20-bit integer) must always be present in restart state; otherwise the result is undefined. If the interrupted operator has no length parameter, the length argument value is 1. the updated table descriptor points to the edit operator that generated the interrupt. (Once an enter-table-edit operator has been resumed in restart state, any subsequent interruption and resumption of the operation of that same edit operator must use restart state.)

If the edit-table word index to be updated into a descriptor exceeds $2^{13} - 1$, an Invalid Index interrupt is generated. If the update was being done to report another interrupt, the Invalid Index is reported instead. An implementation may generate the interrupt at any point in the sequence processing when table code is being executed from a word whose index exceeds the limit.

For the following operator, the edit-mode terminator ENDE leaves no results on the stack:

TEED (table enter edit delete)

For the following operator, the edit-mode terminator ENDE leaves the updated source on top of the stack and the updated destination second from top of the stack:

TEEU (table enter edit update)

Pragmatic Notes

Table-edit restart

It may be simplest always to use restart mode for a page boundary interrupt in a table-edit sequence. Restart state is required in three cases:

1. The interrupted edit operator has transferred one or more characters, so the updated length in a stack argument must be used instead of the code parameter.
2. The FLTF state is 1, so FLTF must not be reset upon resuming the operator.
3. The interrupt occurred transferring data to the destination, so the length is required as either the top or second argument (as specified by the ODI_subtype field in the interrupt ID parameter). The length argument indicates to software the amount by which the destination segment must be extended to complete the edit operator. If an interrupt occurs on an edit operator that lacks a length parameter (INSG, INOP, ENDF), the effective length is 1.

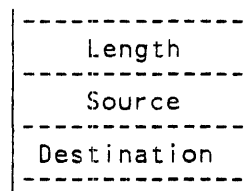
Enter-Single-Edit Operators

- EXSU (execute single edit operator update)
- EXSD (execute single edit operator delete)
- EXPU (execute single edit operator, single pointer update)

Enter-single-edit operators supply the destination sequence, (sometimes) the source sequence, and the length for the edit operator that follows it in the code-stream. Each argument must be on the stack and must meet type restrictions, although it may not be required by the edit operator.

All edit operators requiring length terminate immediately if it is zero.

The EXSD and EXSU operators require length, source, and destination on top of the stack:

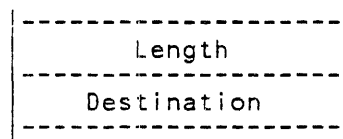


For EXSD, the subsequent edit operator leaves no results on the stack.

For EXSU, the subsequent edit operator leaves the updated source on top of the stack and the updated destination second from top of the stack.

In the case of a Paged Array interrupt, any operator executed by EXSD or EXSU sets RCW.rs to 1 if FLTF = 1. If an EXSx operator is resumed in restart state, any subsequent interruption and resumption of the same operation must use restart state.

The EXPU operator requires length and destination on top of the stack. No source is provided; if the subsequent edit operator is one that generally requires a source, an Undefined Operator interrupt is generated.



The `element_size` convention applied is that a single or double-precision destination descriptor is changed to an EBCDIC Pointer.

The subsequent edit operator leaves the updated destination Pointer on top of the stack. (No delete form of single-pointer enter edit operator is provided.)

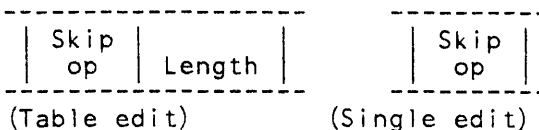
Edit-Mode Operators

The following subsections define the operators that are executed in edit-mode (under the control of an enter-edit operator). (These operators are sometimes called "edit micro-operators".)

Character Skip Operators

Character skip operators advance or back up the source or destination sequence. Length indicates the number of characters to be skipped (a negative length argument is treated as zero).

Length is a parameter for table-edit-mode only:



Skip Forward

- SFSC (skip forward source characters)
- SFDC (skip forward destination characters)

Character skip forward operators advance the source or destination sequence. A source operand is circularly rotated left by length characters. A Pointer is incremented by length characters. Each word in the array from the initial to the final point is accessed, and a Paged Array interrupt is generated if a word has an odd tag. If the operator SFSC is entered by the EXPU operator, an Undefined Operator interrupt is generated.

Skip Reverse

- SRSC (skip reverse source characters)
- SRDC (skip reverse destination characters)

Character skip reverse operators back up the source or destination sequence. A source operand is circularly rotated right by length characters. A Pointer is decremented by length characters; if the resultant `word_index` is less than zero, a Paged Array interrupt is generated. Alternatively, each word addressed by the decrementing index (but not the initial word if the initial character index = 0) is accessed, and a Paged Array interrupt is generated if any of these words has an odd tag. If a Paged Array interrupt is generated, the updated version of the pointer that caused the fault has a word index of 0 and a character index of 0. If the operator SRSC is entered by the EXPU operator, an Undefined Operator interrupt is generated.

Character Insert Operators

Character insert operators store a character or a sequence of characters into the destination sequence, in some cases conditionally based on the value of FLTF (float flip-flop) and EXTF (external sign flip-flop). Each character is a parameter, except for a fixed sign character.

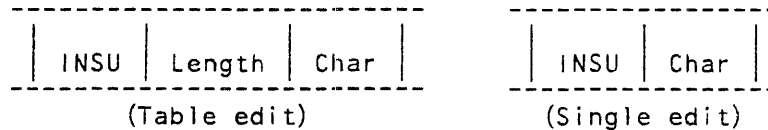
If the destination is marked read_only, a Memory Protect interrupt is generated.

Several insert operators do not allow a hex destination. Those that do store only the numeric field of a parameter character.

INSU (insert unconditional)

INSU stores a sequence composed of length repetitions of the parameter character (Char) into the destination.

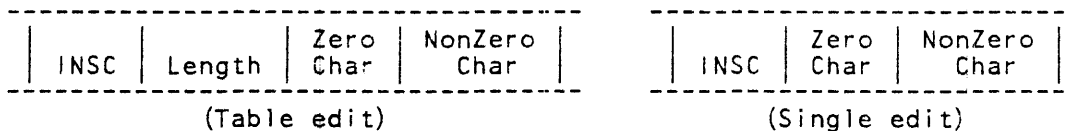
Length is a parameter for table-edit-mode only:



INSC (insert conditional)

INSC stores a sequence composed of length repetitions of a selected parameter character into the destination. If FLTF = 0, ZeroChar is selected; if FLTF = 1, NonZeroChar is selected.

Length is a parameter for table-edit-mode only:



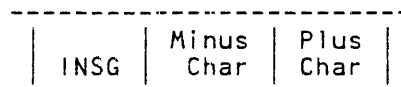
INOP (insert overpunch)

INOP stores hex "D" into the zone field of the destination character if EXTF = 1; the destination character is not altered if EXTF = 0. Note that in either case the destination Pointer is advanced 1 character. If the destination element_size is hex, an Invalid Stack Argument interrupt is generated.

INSG (insert display sign)

INSG stores MinusChar into the destination if EXTF = 1, and stores PlusChar if EXTF = 0. If the destination element_size is hex, an Invalid Stack Argument interrupt is generated.

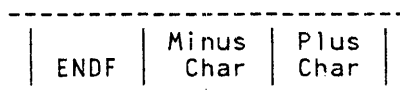
MinusChar and PlusChar are parameters:



ENDF (end float)

If FLTF = 0, ENDF stores a selected parameter character into the destination; MinusChar is selected if EXTF = 1, and PlusChar is selected if EXTF = 0. If FLTF = 1, no character is stored, and the destination Pointer is not advanced. FLTF is unconditionally reset to zero.

MinusChar and PlusChar are parameters:



Character Move Operators

Character move operators transfer characters from source to destination with editing. Some move operators conditionally store into the destination a sequence of repeated parameter characters based on the value of FLTF (float flip-flop), the source character, and EXTF (external sign flip-flop).

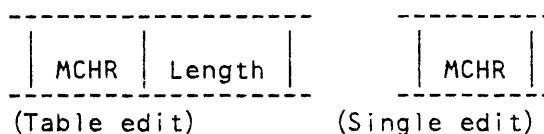
If the operator is entered by the EXPU operator, an Undefined Operator interrupt is given.

If the destination is marked read_only, a Memory Protect interrupt is generated.

If the destination element_size is hex, only the numeric field of a parameter character is stored.

MCHR (move characters)

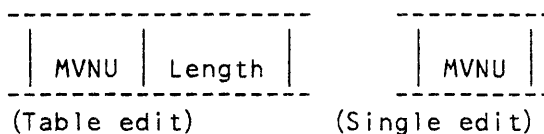
MCHR transfers length characters from the source to the destination. Length is a parameter for table-edit-mode only:



MVNU (move numeric)

For an EBCDIC source and destination, MVNU transfers length numeric fields from the source to the destination, setting each zone field to hex "F". For a hex source and destination, MVNU transfers length hex characters (in this case MVNU is identical to MCHR).

Length is a parameter for table-edit-mode only:



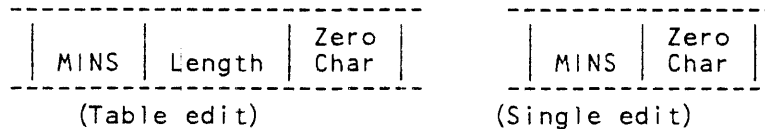
MINS (move with insert)

MINS performs a leading zero suppression function from the source to the destination for length source characters. In the following definition, the "source numeric field" is the numeric field of an EBCDIC character or the entire hex character.

While FLTF = 0 and the value of the source numeric field is zero, the ZeroChar parameter is transferred to the destination. If the value of the source numeric field is nonzero, FLTF is set to 1, and the source numeric field is transferred as described in the next paragraph.

While FLTF = 1, the source numeric field is transferred to the destination and the zone field of an EBCDIC destination character is set to hex"F".

Length is a parameter for table-edit-mode only:



MFLT (move with float)

MFLT performs a signed leading zero suppression function from the source to the destination for length source characters. MFLT is functionally equivalent to MINS (move with insert) except for conditional insertion of a sign character into the destination sequence.

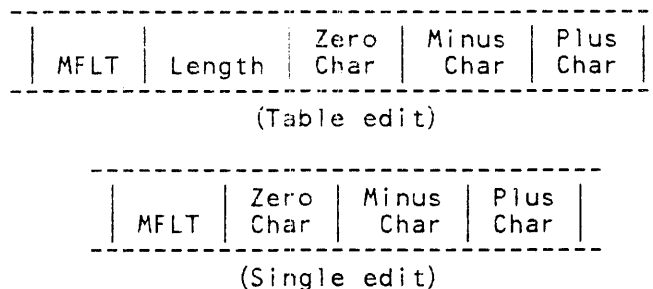
While FLTF = 0 and the value of the source numeric field is zero, the ZeroChar parameter is transferred to the destination.

If FLTF = 0 and the value of the source numeric field is nonzero, the PlusChar (if EXTF = 0) or the MinusChar (if EXTF = 1) is inserted in the destination sequence, FLTF is set to 1, and the source numeric field is transferred as defined for MINS.

While FLTF = 1, the source numeric field is transferred to the destination, as in MINS.

Note that the number of characters stored into the destination sequence may be length + 1. Length characters are stored only if FLTF is initially 0 and for length characters, all source numeric fields are zero, or if FLTF is initially 1.

Length is a parameter for table-edit-mode only:



Miscellaneous Edit Operators

RSTF (reset float flip-flop)

RSTF unconditionally resets FLTF (float flip-flop) to 0.

ENDE (end edit)

ENDE terminates table-edit-mode. If update was enabled by the enter edit operator, ENDE leaves the updated source on top of the stack and the updated destination second from top of the stack.

EXTERNAL COMMUNICATION OPERATORS

CUIO (communicate with Universal I/O)

CUIO requires an Input/Output Control Block (IOCB) data descriptor on top of the stack and passes the address field of the descriptor to the Message Level Interface Port (MLIP). An IOCB descriptor must be an unpagged unindexed present copy SingleDD. The first word of the referenced IOCB array must be a single-precision operand containing an IOCB mark, hex"10CB", in the field [47:16].

If the top-of-stack item is not a valid IOCB descriptor, an Invalid Stack Argument interrupt is generated. If the first word of the IOCB array is not a single-precision operand with a valid IOCB mark, an Invalid Argument Value interrupt is generated. Otherwise, the address field of the descriptor is transmitted to the MLIP, and CUIO terminates when the MLIP acknowledges receiving the address.

A detailed description of I/O operation is contained in the second volume of this manual (the unique System Reference Manual of a host system that uses this architecture).

SCNI/ SCNO (scan in/out)

IDLE (idle until interrupt)

IDLE loops internally until an external interrupt signal is present. At that time, it invokes the interrupt procedure and terminates. The CS flip-flop is not examined or altered. The interrupt RCW designates the operator following the IDLE.

PAUS (pause until interrupt)

PAUS loops internally until an external interrupt signal is present, at which time the operator terminates normally.

The PAUS and IDLE operators differ in that the IDLE operator causes the external interrupt to occur, regardless of control state. The interrupt occurs immediately after a PAUS if CS is false or the interrupt is not masked by CS; otherwise the interrupt remains pending.

REMC (read external memory control)

The REMC operator is provided to read implementation-defined state in devices connected to the processor. A "memory control" is typical of such a device. REMC accepts a single-precision argument and leaves a single-precision value. The implementation must specify the allowable argument values, any validity checking, the form and meaning of the output values, and the semantics of the operator, including any interrupt generation.

Pragmatic Notes

Implementation-defined low-level operators See the note under RIPS.

WEMC (write external memory control)

The WEMC operator is provided to write implementation-defined state in devices connected to the processor. A "memory control" is typical of such a device. WEMC accepts two single-precision arguments and leaves no result. The implementation must specify the allowable argument values, any validity checking, and the semantics of the operator, including any interrupt generation.

Pragmatic Notes

Implementation-defined low-level operators See note under RIPS.

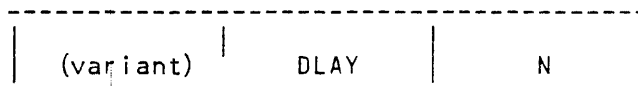
MISCELLANEOUS OPERATORS

NOOP (no operation)

No action is performed.

DLAY (delay)

The DLAY operator has a one-syllable code parameter.



DLAY does nothing for $N + 1$ intervals of time T , where N is the parameter value and T is implementation-defined. The main purpose of the DLAY operator is to occupy one processor long enough for other processor(s) to effect memory access to a shared data word; T should be chosen to suit this purpose.

The DLAY operator is not intended for accurate timings; an error of plus or minus $\max(2, N/5) * T$ is acceptable.

Pragmatic Notes

DLAY pragmatics

In typical implementations, T should be about the duration of an operator (such as RDLK) for which access to main memory is required. On implementations that overlap execution of multiple operators, it may be desirable for DLAY to synchronize the processor so that other operators are also not accessing memory.

PUSH (push working stack onto activation record)

The PUSH operator makes all items on the expression stack addressable as part of the topmost activation record. (See also Expression Stack.)

STOP (unconditional processor halt)

STOP causes the processor execution to halt in an orderly way, so that execution can be resumed by an external action.

HALT (conditional processor halt)

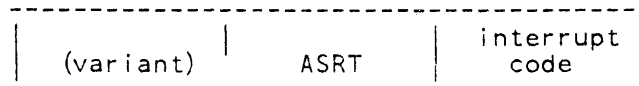
If the processor Halt Boolean is false, HALT is equivalent to NOOP. If the Halt Boolean is true and the HALT operator is executed in variant-mode, the operator is equivalent to STOP. If the Halt Boolean is true and the HALT operator is executed in edit-mode, processor execution halts; the resulting processor state and the ability to continue execution are implementation-defined.

NVLD (invalid operator)

An Invalid Operator interrupt is unconditionally generated.

ASRT (assert)

The ASRT operator has a one-syllable code parameter.



The ASRT operator requires one operand on the stack; otherwise an Invalid Stack Argument interrupt is generated.

The stack argument is interpreted as a Boolean value. If it is True, no further action is taken. If it is False, a False Assertion interrupt is generated with the "interrupt code" parameter passed as an 8-bit integer as the P2 parameter.

VARI (introduce variant operator)

The VARI operator may be considered a primary operator that causes the next code syllable to be interpreted as a variant operator. The two operator syllables are tightly bound, in that no external interrupt can occur between the VARI and the introduced operator, and any RCW that designates a variant operator must point to the VARI.

SECTION 4 INTERRUPTS

GENERAL INFORMATION

An interrupt is an automatic invocation of an operating-system procedure; the mechanism is defined in Section 3 as the common action aINTE. Exit from the MCP interrupt procedure, when practical, returns execution to the interrupted code-stream.

Interrupts are divided into three classes:

- | | |
|----------|---|
| ODI | An Operator Dependent Interrupt is invoked directly by the current operator to request an MCP service required by the operator or to report a programming or operator fault |
| Alarm | An Alarm interrupt is triggered by hardware fault detection during operator execution |
| External | An External interrupt is invoked between operators to report events that are independent of the executing code-stream. |

Appendix C of this manual summarizes Operator Dependent Interrupts and lists operators that invoke each interrupt. In addition, Appendix C also gives the principal condition or state that causes an operator sequence to invoke each interrupt. Operator functions and sequences are defined in Section 3 of this manual.

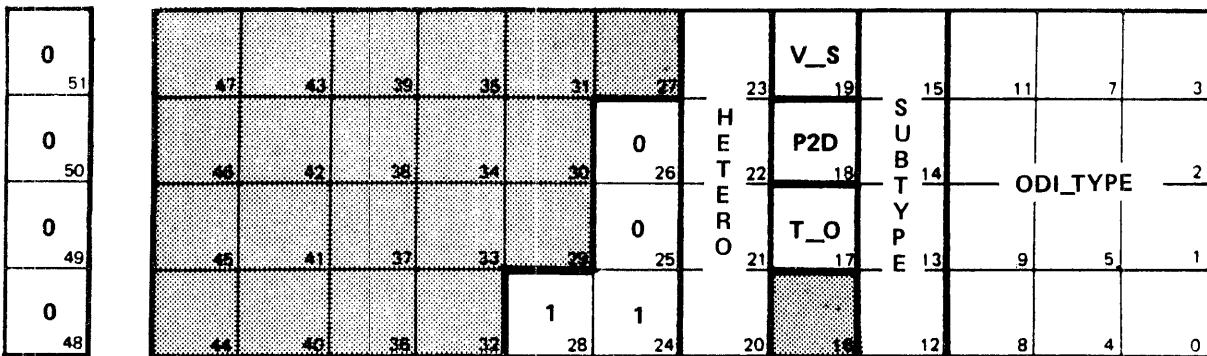
For External interrupts, the RCW created by ENTR (and stored at the F + 1 stack location) will point to the next operator in the current code-stream; for Alarm interrupts, it will point to the operator that was executing when the fault was detected. For most Operator Dependent interrupts, the RCW points to the operator that generated the interrupt. In some cases, the RCW points to the operator immediately following that which detected the interrupt, or indicates the new destination if the interrupt occurred in distributing a code-stream pointer. Note that in single edit-mode, the executing operator is considered to be the enter single edit operator, not the edit operator. Similarly, for variant-mode operators, the RCW points at the VARI operator.

Interrupt Parameters

Information passed to the MCP interrupt procedure is contained in two parameter items in the stack. The first is a single-precision operand interpreted as an interrupt identification literal (ID). The second item, called the P2 parameter, varies according to the nature of the interrupt.

Interrupt ID Parameter

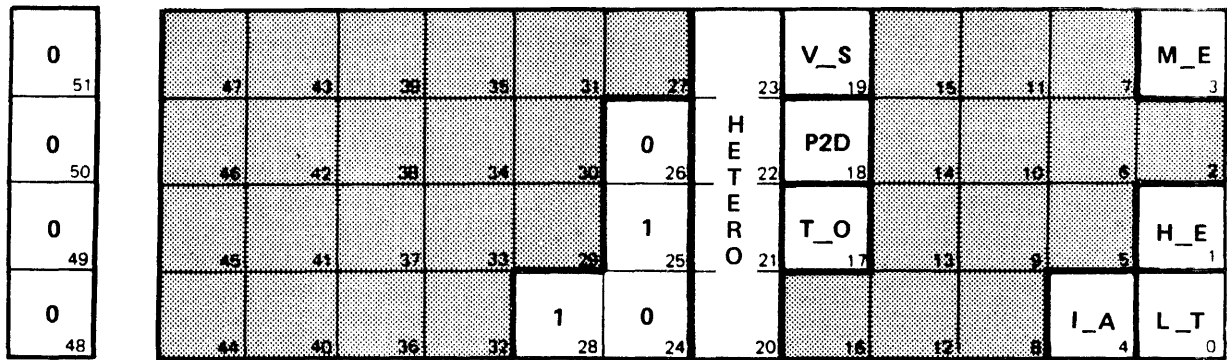
The first interrupt parameter is the single-precision interrupt identification literal (ID parameter). Figures 4-1 through 4-3 show the different formats of this interrupt parameter word. For all interrupts, the int_class field (ID.[26:3]) indicates the class of interrupt with values {1 = Operator Dependent, 2 = Alarm, 4 = External}. Note that {0,3,5,6,7} are invalid.



MV5375

- E-mode__bit [28: 1] Constant value 1
- int__class [26: 3] Binary 001: Operator Dependent
- hetero [23: 4] Reserved for use by heterogeneous systems
- valid__state [19: 1] (1: valid; 0: invalid)
- P2__double [18: 1] (0: P2.tag value was retained, 1: P2.tag was 2)
- this__op [17: 1] (0: code-stream pointer advanced or moved, 1: RCW → interrupted operator)
- ODI__subtype [15: 4] Qualifiers for particular ODI interrupts
- ODI__type [11:12] ODI type number

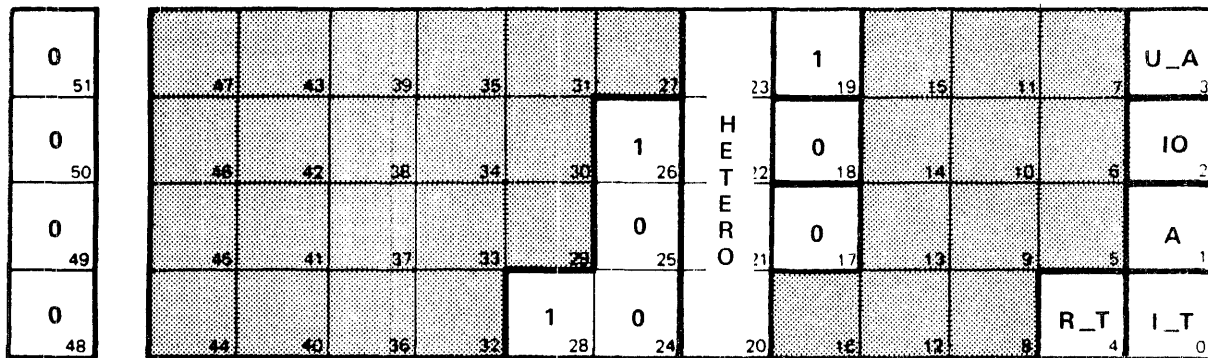
Figure 4-1. P-1 Operator Dependent Interrupt (ODI) ID Parameter Format



MV5376

E-mode__bit	[28: 1]	Constant value 1
int__class	[26: 3]	Binary 010: Alarm
hetero	[23: 4]	Reserved for use by heterogeneous systems
valid__state	[19: 1]	(1: valid; 0: invalid)
P2__double	[18: 1]	(0: P2.tag value was retained, 1: P2.tag was 2)
this__op	[17: 1]	(0: code-stream pointer advanced or moved, 1: RCW → interrupted operator)
invalid__addr	[4: 1]	1 = Invalid Address
mem__error	[3: 1]	1 = Uncorrectable Memory Error
hardware__error	[1: 1]	1 = Hardware Error
loop__timer	[0: 1]	1 = Loop Timer

Figure 4-2. P-1 Alarm Interrupt ID Parameter Format



MV5377

- E-mode__bit [28: 1] Constant value 1
- int__class [26: 3] Binary 100: External
- hetero [23: 4] Reserved for use by heterogeneous systems
- valid__state [19: 1] (1: valid)
- P2__double [18: 1] (0: P2.tag value was retained)
- this__op [17: 1] (0: code-stream pointer advanced or moved)
- run__timeout [4: 1] 1 = Running Timeout
- unmasked__attn [3: 1] 1 = Unmasked Attention
- IO [2: 1] 1 = I/O Finished
- attn [1: 1] 1 = Attention
- int__timer [0: 1] 1 = Interval Timer

Figure 4-3. P-1 External Interrupt ID Parameter Format

Fields in [47:32] are common to the interrupt ID for all interrupt classes and are defined below. Fields in [15:16] depend on the `int__class` value and are defined in Section 1 for each class of interrupts.

Interrupt ID (tag = 0)

Field Name	Bits	Meaning or Usage
	[47:19]	Reserved for future use
<code>E-mode__bit</code>	[28: 1]	Constant value 1
	[27: 1]	Reserved for future use
<code>int__class</code>	[26: 3]	1 = Operator Dependent 2 = Alarm 4 = External
<code>hetero</code>	[23: 4]	Reserved for implementation-defined use by heterogeneous systems
<code>valid__state</code>	[19: 1]	Indicates validity of state for re-entry to the code-stream (1: valid; 0: invalid)
<code>P2__double</code>	[18: 1]	If 1, the P2 parameter is a single-precision first word of a item. If 0, the tag of the P2 parameter correctly indicates its type.
<code>this__op</code>	[17: 1]	If 1, the RCW points to the interrupted operator. If 0, the code-stream pointer has been advanced or moved.
	[16: 1]	Reserved
	[15:16]	Dependent on <code>int__class</code>

The `E-mode__bit` serves to distinguish interrupts on this architecture from those of preceding implementations that may share software. (If the `E-mode__bit` is 1, the processor is capable of executing the WATI operator.)

If `valid__state` = 1, the global system state and the top-of-stack configuration are proper for initiation of the operator referenced by the interrupt RCW. If `valid__state` = 0, the global system state or the top-of-stack configuration may not be consistent; the operating system may not EXIT back to the interrupted environment. In this case the state of the stack immediately below the interrupt MSCW is implementation-defined.

If `this__op` = 1, the interrupt RCW references the operator that was interrupted. (If that operator was a variant operator, the RCW points to the VARI; if it was an edit-mode operator, the RCW points to the TEED, TEEU, EXSD, EXSU or EXPU operator.) If `this__op` = 0, the RCW records a code-stream pointer that has been advanced or moved. If the interrupt is Operator Dependent or an Alarm, it was generated (or enabled) by or during the previous operator execution. That operator may not be the physical predecessor in the code-segment; it may have been a branch or subroutine operator that moved the code-stream pointer.

`ID.P2__double` is used to indicate that the P2 parameter had a tag = 2. (See Interrupt P2 Parameter in this section.)

The couple (valid__state,this__op) is subject to the following interpretations:

- (0,0) The interrupted code-stream may not be resumed; the interrupt was generated by a prior operator or between operators, not by the operator addressed by the RCW.
- (0,1) The interrupted code-stream may not be resumed; the RCW addresses the operator that was interrupted.
- (1,0) The code-stream may be resumed with the operator referenced by the RCW, which did not generate the interrupt. The interrupt may have been generated between operators (external), or by the previous operator (operator-dependent), which has consumed its stack inputs, produced its normal stack outputs, and effected its normal changes on system state.
- (1,1) The code-stream may be resumed at the operator that was interrupted. (RCW.rs is 0 or 1, depending on the stack configuration for initial or restart state at the beginning sequence of the operator.)

Resumption Conditions

The triple (ID.valid__state,ID.this__op,RCW.rs) defines the "resumption condition" for an interrupt. Frequently specified resumption conditions have names, as follows (x means "either 0 or 1 as implementation-defined"):

- (0,x,x) Defunct
- (1,0,0) Continue
- (1,1,x) Repeat-IR
- (1,1,0) Repeat-Initial
- (1,1,1) Repeat-Restart

The term Continue-Next is used as an abbreviation for the specification that the resumption condition is Continue and the interrupt RCW references the operator that follows the interrupted operator in the code-segment. The Continue-Next condition is not uniquely encoded, but can be inferred when Continue condition is reported for particular interrupt types.

Repeat-IR specifies "either Repeat-Initial or Repeat-Restart as implementation-defined". The term is used when Restart is not required, but may be used at implementation option.

The resumption condition is specified for each interrupt. The specification may be applied to the entire class, to an interrupt type, or to a particular instance of interrupt generation.

NOTE

The specification of a Repeat-Initial condition means the state is consistent with re-entering the operator "at the top," not that the operator inputs are unchanged. For example, a reference-evaluation operator may have consumed part of a reference chain, or a pointer operator may have performed part of its function and updated its arguments.

Pragmatic Notes

Valid__state permits operator retry action

For operator-dependent error interrupts and alarm interrupts, the `valid__state` bit informs the MCP whether operator retry is feasible. In general, the extent to which operators may be retried depends on the implementation. The specification requires that `valid__state` be 0 unless retry is proper; it usually does not specify which error situations may permit retry.

One practical implementation technique is to define a "retry" bit that is set true at the beginning of each operator and set false whenever an operator changes state in any way that invalidates retry; error interrupt generation then transcribes "retry" to `ID.valid__estate`.

P2 parameter

The second interrupt parameter, the P2 parameter, varies according to the specific interrupt type. For a given interrupt it may be an item of fixed or varying type, or it may contain no information.

Interrupts involving descriptors often present special concerns for P2. The phrase "P2 is a copy of the descriptor" is used to mean that if a DD is encountered, aCPY action is used to fetch a copy DD as P2; this situation is typical of an interrupt generated by an operator that is attempting to fetch a word from memory and interpret it as a descriptor. For Presence Bit interrupts generated in code-stream pointer distribution, P2 is an aCPY copy of the absent CSD; this is the only context in which a copy CSD occurs. In some error reports, a descriptor may occur in memory or on the stack simply as a word type not recognized in the context; in such cases an unchanged duplicate of an original descriptor may be reported as P2.

Whenever the item to be reported as P2 is double-precision (`tag = 2`), the first word of the item is reported with `tag = 0` and `P2__double = 1` is reported in the ID parameter. (Correct software operation requires that the interrupt mechanism pass a fixed number of parameter words to the interrupt procedure, so a double-precision tag in P2 is suppressed.)

Interrupt Definition, in this section, specifies interpretation individually for all interrupts in which P2 is meaningful. Where the P2 parameter contains no information, it is not explicitly specified.

Superhalt

A superhalt condition exists when the processor cannot continue to process operators, either in the current code-stream or by interrupting to another code-stream. When a superhalt condition exists, the processor halts in a state from which normal continuation is not possible.

Superhalt conditions can be generated internally by implementation (a microprocessor failure), by an operator (that is, when MVST encounters an interruptable condition during the interval that the processor has no stack environment), or by detection of an interrupt loop.

An interrupt loop is detected by means of the `Interrupt__Count` register value. The register is incremented by one at the beginning of interrupt entry; if it is incremented from 3, a superhalt condition exists. The `Interrupt__Count` register is not automatically decremented; it can be set to zero by the ZIC operator. (An interrupt loop could arise if, for example, the ENTR operator invoked by the interrupt entry sequence itself generated an interrupt and the ENTR invoked by that interrupt generated an interrupt, ad infinitum.)

INTERRUPT DEFINITION

Operator Dependent Interrupts

Operator Dependent Interrupts are invoked directly by the current operator to request an MCP service required by the operator or to report a programming or operator fault.

The Operator Dependent ID parameter identifies the type of interrupt, and indicates by the `valid__state` and `this__op` bits the status of the code-stream pointer in the interrupt RCW (see Resumption Condition). Operator Dependent ID (`int__class = 1`).

ODI__subtype	[15: 4]	The subtype of Operator Dependent Interrupt, defined as required for each value of ODI__type.
ODI__type	[11:12]	The type of Operator Dependent Interrupt, where 0 = Presence Bit S 1 = Paged Array SE
	[12:1]	1: page referenced by P2 was being written
	[13:1]	1: length is word count 0: length is character count
	[14:1]	1: length is at F-2 0: length is at F-1
	[15:1]	1: operator was skip reverse ([14:2] is valid only if [12:1] = 1) 2 = Stack-Overflow Se 3 = Invalid Operator Es 4 = Undefined Operator Es 5 = Invalid Stack Argument E 6 = Invalid Argument Value E 7 = Invalid Code Parameter E 8 = Invalid Reference E 9 = Invalid Reference Chain Es [12:1] = 1: operator was ENTR 10 = Invalid Index Es 11 = Memory Protect E 12 = Divide by Zero E# 13 = Exponent-Underflow Ew# 14 = Exponent-Overflow E# 15 = Integer-Overflow E# 16 = Stack-Underflow E 17 = << unused >> 18 = Stack Structure Error E 19 = Code Segment Error E 20 = Invalid Program Word E 21 = << unused >> 22 = Invalid Object E

- 23 = Page Structure Error E
- 24 = Block Exit S
- 25 = Binding Request S
- [12:1] = 1: operator was ENTR
- 26 = Precision Loss Ew#
- 27 = False Assertion E
- 28 = Locking S
- 29 = Unlocking S

Values of specific ODI__subtype bits are defined for specific ODI__type cases; when not specified, ODI__subtype values are undefined.

Operator Dependent Interrupts are defined in two classes: those that (usually) request an MCP service, and those that (usually) report error conditions arising from programming or operator faults. The next two subsections define these classes.

Pragmatic Notes

ODI Classification

As a suffix to the ODI__type definition, the table above shows the classification of each interrupt, according to the following legend. In general, interrupts classified as "E" have implementation-defined resumption conditions and cannot, therefore, be treated as service requests. The other interrupts generally have resumption conditions specified in this architecture. Most "E#" and "Ew#" cases have Continue-Next specifications, with the operator result on the stack, but that result is reasonable only in the "Ew#" case.

- S: Service
- Se: Service, sometimes treated as error
- SE: Service by definition, but likely to be Error
- E: Error
- Es: Error, but capable of being interpreted as Service
- E#: Error: numeric result is out of range and unusable
- Ew#: Error warning: numeric result is out of range but usable

When an operator reports several interrupts, there is generally no requirement that one interrupt take precedence, other than that imposed by the operator function. When processing actions are functionally sequential, interrupts generated by the earlier take precedence; otherwise, interrupts generated by any part of the operator may be reported. (For example, if some item B is meaningful only when item A is interpreted in a particular way, an error in that interpretation of A must take precedence over an error detected in B.) If a conditional action of the operator is not performed, interrupts that might have been generated by it are not required. (For example, if the interpretation of A is such that B is not significant, then any errors that might have been detected in B need not be reported.)

Pragmatic Notes

ODI__subtype provides operator context

ODI__subtype values are defined for certain ODI__type values to allow software to determine the interrupt context without the need to locate the interrupted operator by evaluation of the RCW code-stream pointer.

MCP Service

The interrupts defined in this section usually constitute requests for an MCP service that is an extension of the hardware operators. In some situations, especially limiting cases, no service can be provided and the interrupt must be treated as an error.

Presence Bit

A Presence Bit interrupt is used by operators to gain access to a data array or program code-segment that is not present in memory. A data array (or a stack) is accessed through a DD. A program code-segment is accessed through a CSD. A Presence Bit interrupt is generated when access is required under the following conditions:

Access is required through use of an original DD that is absent ($DD.present = 0$ and $DD.copy = 0$). An absent copy DD is treated as a reference to the original. If $DD.present = 0$ and $DD.copy = 1$, $Mem[DD.address]$ is accessed; the Presence Bit interrupt occurs only if that original DD is absent. (If no original DD is found, an Invalid Object interrupt is generated, rather than the Presence Bit interrupt.)

Access to code is required by means of an absent CSD ($CSD.present = 0$).

NOTE

A Presence Bit interrupt is not generated if the descriptor is an absent copy, but the associated original descriptor is present. In that case, the address field of the associated original is used to make the required access.

In the case of an access through a data descriptor, including a stack descriptor, the P2 parameter is a copy of the DD. The resumption condition is Repeat: after the segment has been made present and the original DD changed, an exit from the interrupt procedure will repeat the operator that generated the interrupt.

In the case of an access through a code-segment descriptor, the P2 parameter is a copy of the CSD. The resumption condition is Continue and the interrupt RCW contains the new code-stream pointer. After the code-segment has been made present and the original CSD changed, an exit from the interrupt procedure will complete the enter, exit, or dynamic branch into the intended code-segment.

Paged Array

A Paged Array interrupt is used by pointer operators to indicate an attempted access beyond the end of the array or page. Pointer operators that access a data array sequentially rely on the following assumptions:

1. The elements of an array page are operands.
2. The words directly before and after an array page have odd tags.

Pointer operators generally perform sequential processing of data arrays. A Paged Array interrupt is generated by these operators when an odd-tagged item is read from an array or a store is attempted into an array word containing an odd-tagged item. If the MCP determines that the odd-tagged word marks the end of the virtual array, an error condition exists, and the operator can be resumed only if the MCP enlarges the segment and modifies the descriptors accordingly. If the odd-tagged word marks the end of an actual segment (array page) but not the end of the virtual-segment, the MCP can adjust the pointer on the stack and return from the interrupt procedure to resume the operator on the next page of the array.

For operators traversing the sequence in the forward direction, bit-15 in the ODI__subtype field of the interrupt parameter is 0, and P2 is an IndexedDD denoting the first character where access was attempted within a word with an odd tag; normally, this is the first character outside the actual segment. For skip reverse operators, bit-15 in ID.ODI subtype is 1 and P2 is normally an IndexedDD with index 0, denoting the beginning of the actual segment in which the skip occurred; if an odd-tag word occurs within the segment, the index of P2 is implementation-dependent. The MCP is required to replace the first copy of the same IndexedDD below the interrupt MSCW by an IndexedDD correctly referencing the next array element.

Three bits in ODI__subtype indicate whether the descriptor in P2 is the destination of a transfer, whether the transfer is in words or characters, and the location of the updated length argument; Bits 13 and 14 are significant only if bit 12 = 1. A fourth bit indicates the direction of traversal of the character sequence.

- [12:1] Transfer destination indicator
(0: P2 is source pointer or operator does not transfer data;
1: P2 is destination pointer and operator transfers data.)
- [13:1] Element size indicator (0: characters; 1: words)
- [14:1] Length position indicator (0: in Mem[F-1];
- [15:1] Direction indicator (0: forward; 1: reverse)

The resumption condition for Paged Array interrupt is specified in Appendix C; it is usually a form of Repeat.

Pragmatic Notes

ODI__subtype supports destination expansion

ODI__subtype bit 12 is defined for Paged Array interrupt so that software can recognize attempted data transfer past the segment end. Software has the option of expanding the segment and resuming the interrupted operator. The amount of expansion required to complete the operation can be determined from the updated length, the P2 element__size, and the indicator in ODI__subtype bit 13. The stack location of the updated length is indicated in bit 14.

Binding Request

A Binding Request interrupt is generated when a reference chain evaluation (for any operator but EVAL) produces a DD with element__size = 7. The interrupt is also generated when an indexing operator encounters a copy DD with element__size = 7 as a Descriptor Indication.

The requested service is to replace the original DD with an appropriate item according to software convention. (If the DD is an absent original, all fields but present, copy, and element__size are subject to software interpretation.)

The P2 parameter is a copy of that descriptor. (If that descriptor is an original DD, the copy is created by aCPY action). If the interrupted operator is ENTR, bit 12 of the interrupt ID parameter is set to 1.

The resumption condition is Repeat-Initial or Repeat-Restart according to whether the interrupted operator began in initial or restart state.

Stack Overflow

Stack-Overflow indicates that a push onto the expression stack has caused the stack size to equal its limit. The interrupt is a request to the MCP to extend the array in memory for the stack. All operators may be resumed subsequent to MCP Stack-Overflow processing, under the assumption that the stack size has been extended.

One of the following conditions must be present to restart or retry an operation that encounters a Stack-Overflow interrupt:

1. The operator must complete before generating the interrupt. In this case the resumption condition is Continue.
2. The operator must detect any possible stack overflow before altering its inputs or any permanent state. In this case, the operator may be retried, and the resumption condition is Repeat.

Resumption condition specifications for either of these two interrupt situations is implementation-dependent, within the constraints listed.

While classified as an Operator Dependent Interrupt, Stack-Overflow is not necessarily reported by an operator that causes growth in the number of words on the expression stack. Because the Stack-Overflow condition may be defined in terms of memory words, and because a processor may retain some top-of-stack words in local state, the Stack-Overflow condition may be detected when words that are already formally on the expression stack are moved from local state to memory. Note, too, that an operator can pop a word from the expression stack, causing S to be decremented to LOSR-1, and then push a result onto the stack; thus, an operator with no more stack results than arguments can generate a Stack-Overflow interrupt.

Block Exit

The EXIT and RETN operators generate a Block Exit interrupt when an attempt is made to deallocate an activation record that has `RCW.block__exit = 1`.

The resumption condition is Repeat-Initial.

Locking and Unlocking

The Locking and Unlocking interrupts are generated by the LOK and UNLK operators, respectively, when operating-system service is required to resolve an interlock contention. P2 contains a reference (SIRW or IndexedSingleDD) to the interlock.

The resumption condition is Continue-Next.

Error Reporting

This set of interrupts reports error conditions arising from programming, compiler or operator faults. (In some cases, the MCP may take corrective measures or otherwise remove the error situation and resume the code-stream, in which case the interrupt was effectively a service request. Such action is possible, of course, only with Repeat and Continue resumption conditions, when `ID.valid__state = 1`.)

Some error reporting interrupts are "optional." That is, some valid implementations of this specification may not check for these error conditions. On such an implementation, the results of an operation producing an undetected error condition are undefined. Appendix C indicates which error conditions are optional.

Pragmatic Notes

Optional checks

The principles upon which some checks are made optional are these:

1. Given a correct implementation of operating system and user-language compiler, there is no way to verify that the check is or is not made.
2. The likelihood of a devastating failure being avoided by including the check is deemed small.

A compelling reason for omitting a consistency check is that some optimization has made it unnecessary for the relevant state to be accessed. In other cases, an implementation can be made faster by ignoring some checks on state that are unlikely to be wrong or innocuous if wrong. The general recommendation is that an implementation include as many checks as practical, especially tag checks on any words that must be accessed anyway.

Invalid Operator

An Invalid Operator interrupt is unconditionally generated by execution of NVLD (invalid operator). No other operator generates this interrupt.

The resumption condition is Repeat-Initial.

Undefined Operator

An Undefined Operator interrupt is generated due to the attempted execution of an operator whose encoding is not valid in the context. Valid operator encodings are found in Appendixes A and B. Primary and Variant encodings are expected in the normal succession of operators in the code-stream, whether accessed sequentially or subsequent to branch or subroutine operators. Edit encodings are expected in the code-stream following an enter single-edit operator or in a table designated by an enter table-edit operator; edit operators requiring a destination are undefined following the EXPU operator. Only a NAMC operator is defined following a MКСN operator.

If an edit operator was expected, the resumption condition is Defunct-Here: the RCW points to the enter-edit operator, and for table edit, the table pointer is updated to point to the offending code-syllable. If an operator other than NAMC follows MКСN (and the implementation enforces the restriction), the resumption condition is Defunct; it is implementation-defined whether the RCW points to the MКСN, or to the next operator in the code-stream sequence.

Otherwise, the resumption condition is Continue-Next, and P2 is an operand containing the following information:

[47:39]	zero
[8: 1]	1 if a variant operator was expected
[7: 8]	The unrecognized operator syllable

Invalid Stack Argument

An Invalid Stack Argument interrupt indicates an invalid initial stack state for an operator. This interrupt is generated by any operator that places data type restrictions on its dynamic stack arguments if one or more items on top of the stack do not have the required type(s). Argument type restrictions are in terms of data types defined in Supported Data Types, of this section, according to tag value and, in some cases, additional type bits within the word.

For all Invalid Stack Argument interrupts, the stack item that violates type restriction is the P2 parameter. If two or more items are of incorrect type, only one is the P2 parameter. If the incorrect item is double-precision, the first word is given as a single-precision P2 parameter operand, and ID.P2__double = 1. If the incorrect item is an original DD, it is given as P2 without modification.

The resumption conditions are implementation-defined.

Invalid Argument Value

An Invalid Argument Value interrupt indicates that the data type of a dynamic stack argument is correct, but its value is not within a valid range. This interrupt is generated if an operand argument (interpreted as an integer) produces an invalid value, or if a field of a structured data type item has an undefined or invalid value.

The stack item having an invalid value is the P2 parameter. If that item is double-precision, its first word is given as a single-precision P2 parameter operand, and ID.P2__double = 1.

The resumption conditions are implementation-defined.

Invalid Code Parameter

An Invalid Code Parameter interrupt indicates that a code-stream parameter has an invalid value. This interrupt is generated if a parameter is interpreted as an integer and produces a value greater than the maximum valid value, or if the value of the parameter does not meet other constraints imposed by the operator. The invalid value is given as the P2 parameter in the form of a single__integer.

The resumption conditions are implementation-defined.

Invalid Reference

An Invalid Reference interrupt indicates an attempted evaluation of an invalid address-couple reference to an item in the current addressing environment. This interrupt is generated during evaluation of a NIRW or an address-couple parameter under the following conditions:

1. The Lambda (lexical level) component is greater than LL (the lexical level at which the processor is running).
2. For Lambda = LL, the address of the referenced stack location is greater than the top-of-stack address.

If the invalid reference is a NIRW, the NIRW is given as the P2 parameter. If the invalid reference is an address-couple parameter, the P2 parameter is a single-precision operand whose low-order field is the address-couple. A fixed-fence address-couple is transferred to P2 without modification; a variable-fence address-couple may be given as P2 without modification or, after translation to fixed-fence format, as defined by the implementation.

The resumption conditions are implementation-defined.

Invalid Reference Chain

An Invalid Reference chain interrupt indicates that a reference evaluation produced an unexpected result. This interrupt is generated by operators that evaluate reference chains, when the evaluation of an address-couple parameter, NIRW, SIRW, or IndexedWordDD produces an item that is neither a valid reference in the chain nor a valid target item that terminates the chain. The definitions of valid reference chains and valid target items vary according to operator function.

The invalid reference evaluation result is the P2 parameter. If that item is a double-precision operand, the first word is passed as a single-precision operand with ID.P2__double = 1. If that item is a DD, the P2 value is a copy fetched by aCPY action. (The reference evaluation result is never the initial reference; an incorrect initial reference causes an Invalid Reference or Invalid Stack Argument interrupt.) If the interrupted operator is ENTR, bit 12 of the interrupt ID parameter is set to 1.

The resumption conditions are Repeat-Restart if the operator began in restart state, or Repeat-IR otherwise. When the initial-or restart-state specification of the operator requires an initial reference to the chain in question, a valid reference is left on the stack; this may be the original reference (unless PCW evaluation has occurred) or one of its successors.

Pragmatic Notes

Restart states for Invalid Reference Chain interrupts

Typically, an invalid reference chain is detected when some valid reference (in the operator context) points to some item that is not valid. The last valid reference is left on the stack as the argument to resume the operator, and the erroneous item is the P2 parameter. The typical resumption condition is Repeat-Initial. If the reference chain did not include a PCW (accidental entry), it is also permissible to leave the initial reference as the resumption stack argument. In the particular case that the initial reference is part of the operator (a VALC operator for instance), the choices are to use Repeat-Initial and restart the chain from the beginning, or to use Repeat-Restart and provide the reference argument on the stack. (Of course, if a PCW has already been evaluated, the Repeat-Restart condition must be used; this is an example of "once in restart, always in restart.")

Pragmatic Notes

Interrupts related to reference evaluation

The Invalid Reference Chain interrupt is generated in situations in which a reference does not point to a valid item. It is generated only by operators that evaluate potential reference chains. When the chaining rules for such an operator are violated, there are three possible interpretations from a programmer's viewpoint, but these are not generally distinguishable by the processor:

The unexpected item was an improper next reference in the chain.

The unexpected item was an improper final target.

The unexpected item was the accidental target of a valid-appearing, but misdirected reference.

The first possibility can be excluded in any context that does not permit reference chaining. In these cases, the Invalid Object interrupt is generated (although the third possibility still exists).

If an initial reference is unacceptable as to type, an Invalid Stack Argument interrupt is generated. Address-couple initial references are also subject to Invalid Reference interrupts.

If PCW evaluation invokes an accidental-entry procedure that returns an unsatisfactory value, the resumed operator produces an Invalid Stack Argument, rather than an Invalid Reference Chain interrupt. This situation is the same as that of an invalid initial reference for the restarted operator.

If the invalid reference evaluation result in an Invalid Reference Chain situation is a DD with element__size = 7, a Binding Request interrupt is generated instead.

Invalid Object

An Invalid Object interrupt is generated if a single reference (rather than a reference chain) evaluation is to be performed and the target object does not satisfy the operator requirements.

The interrupt is also generated in certain special cases:

A double-precision operand is to be fetched, but the second word has an incorrect tag.

The stack-vector descriptor is not an unpagged, original SingleDD.

The operand type to be stored (single-or double-precision) does not fit the target.

A stack descriptor is not an unpagged, unindexed Single DD.

The address field of an absent copy DD does not designate an original DD.

The invalid object word is the P2 parameter. If that word has a tag of 2, it is passed as a single-precision operand with `ID.P2__double = 1`. If that word is a DD, the P2 value is a copy fetched by `aCPY` action.

The resumption conditions are implementation-defined.

Invalid Index

An Invalid Index interrupt is generated if an integer value used to index an array of elements is not within a valid index range for that array. Invalid Index conditions may exist for indexing data descriptors, code-segment descriptors, the stack-vector descriptor, or (by the `OCRX` operator) linear indexing functions. Invalid Index interrupts can also be generated when an index value exceeds the field width in a descriptor. The P2 parameter varies depending on the type of array and form of index, as noted in the following cases:

Data Descriptor (DD)

Invalid Index is generated when indexing an unindexed DD if the index value is not in the range $\{0 \text{ to } \text{DD.length}-1\}$ or if, in indexing an unpagged DD or updating an indexed DD, the computed word index is not in the range $\{0 \text{ to } 2^{**}W - 1\}$ (where W is 20 for IndexedWordDDs, 16 for pointers, and 13 for edit table operators). Indexing operators pass a copy of the unindexed DD as the P2 parameter. Pointer operators pass a copy of the IndexedDD as the P2 parameter, with the word index field containing the computed index modulo $2^{**}W$.

Code Segment Descriptor (CSD)

Invalid Index is generated by branching operators if the program-word index component is not in the range $\{0 \text{ to } \text{CSD.seg_length}-1\}$. If the new code-stream pointer is specified by a PCW (dynamic branches, `ENTR`), the PCW is passed as the P2 parameter. If branching within the current code-segment is indicated by a top-of-stack operand (dynamic branches), P2 is the operand, and if indicated by a parameter (static branches), P2 is a single-precision operand, where the low-order field is the 16-bit parameter.

Stack-Vector Descriptor (SVD)

Invalid Index is generated during stack accessing if the `stack__number` is not in the range {0 to `SVD.length-1`}. An indexed copy of the SVD, where the index field is the invalid `stack__number`, is the P2 parameter.

Linear record structure

Invalid Index is generated by the OCRX operator if the sequence index operand is not in the range {1 to `ICW.ICW__limit`}. P2 is implementation defined.

The resumption condition is Repeat-IR for `INDX`, `INXA`, `NXLV`, `NXVA`, and `NXLN`, when the index is not in {0 to `DD.length-1`}, and Repeat-IR for OCRX when the index is not in {1 to `ICW.ICW__limit`}. The resumption condition is implementation-defined in all other cases.

Memory Protect

A Memory Protect interrupt indicates an invalid attempt to write into a memory location, or improper access to a memory-protected word. It is generated under the following conditions:

1. A write is attempted by store, overwrite, pointer, or edit operators, where the memory location is referenced by an IndexedDD marked `read__only`. The IndexedDD is the P2 parameter.
2. For store operators, a tag = 3 item is encountered in evaluating a reference chain, or the second-word location for a double-precision item contains an odd-tagged word. The tag = 3 or odd-tagged item is the P2 parameter.
3. An odd tagged word is unexpectedly encountered in a set or translate table by a pointer operator or in the source by a show operator. The IndexedDD referencing this word is the P2 parameter.

The resumption conditions are implementation-defined.

Divide by Zero

A Divide by Zero interrupt is generated by arithmetic divide operators if the numeric interpretation of the top-of-stack operand (the divisor) yields a value of zero.

The P2 parameter is the dividend.

The resumption condition is Continue-Next; the result value is implementation-dependent.

Exponent-Overflow

An Exponent-Overflow interrupt is generated by arithmetic and numeric type transfer operators if the result of a rounding or truncation function is an exponent value too large to fit in the exponent field of the operand format.

The resumption condition is Continue-Next. For `SNGL` or `SNGT`, the result type is single-precision. For binary operators, the result value is single- or double-precision as determined by the input argument types. The result magnitude is the largest representable in that type; the result sign is determined by the input argument(s).

Exponent-Underflow

An Exponent-Underflow interrupt is generated by arithmetic and numeric type transfer operators if the result of a rounding, truncation, or normalization function is an exponent value too small to fit in the exponent field of the operand format.

The resumption condition is Continue-Next. The result type is single-precision for SNGL or SNGT, as determined by the input argument types(s) for other operators. The result value is zero. The interrupt procedure can simply exit, if the underflow is tolerable.

Precision Loss

A Precision Loss interrupt is generated by arithmetic operators if the result of a rounding function results in a Loss-Of-Precision.

When Precision Loss is reported, the unnormalized (imprecise) result of the operation is left on the stack and the resumption condition is Continue-Next. The interrupt procedure can simply exit, if the Loss-Of-Precision is tolerable.

Integer-Overflow

An Integer-Overflow interrupt indicates that an operand required to have an integer value cannot be represented as an integer. This interrupt is generated if the integer numeric value of the operand, after truncation or rounding if necessary, is not in the range $\{-2^{39}+1$ to $2^{39}-1\}$ for single-or $\{-2^{78}+1$ to $2^{78}-1\}$ for double-precision.

The operand is the P2 parameter. If it is double-precision, the first word is used as a single-precision P2 parameter operand, with ID.P2_double = 1. For the following operators, the resumption condition is Continue-Next and the result on the stack has the specified type and representation, with indeterminate value:

Operator	Type	Representation
NTIA, NTGR	single-precision	integer
NTGD, NTTD	double-precision	integer
IDIV	per inputs	integer
RDIV	per inputs	any

For the following operators, if the interrupt occurs while integerizing the argument to be scaled or converted, the resumption condition is Continue-Next and the result on the stack has the specified type and representation, with indeterminate value:

Operator	Type	Representation
SCLF, DSLF	double-precision	integer
SCRR, DSRR, SCRT, DSRT	double-precision	integer
SCRF, DSRF	single-precision	decimal-digit sequence
SCRS, DSRS	(tos)double-precision	decimal-digit sequence
	(2nd)single-precision	decimal-digit sequence
BCD, DBCD	per N	decimal-digit sequence

In all other cases of Integer-Overflow, resumption conditions are implementation-defined.

Pragmatic Notes

Integer-Overflow Pragmatics

Integer-Overflow interrupt permits continuation of the code-stream when the argument being integerized is the primary input to an operator whose output is an arithmetic function of that input. The code-stream cannot be resumed when the argument is, for instance, an index, a length, or a scale factor. The type and representation of the results are those that would occur in the limit with inputs that result in very large, but not overflowing, magnitudes. Note that a single- or double-precision zero is an acceptable result in each case.

Stack-Underflow

Stack-Underflow indicates that an operator attempted to pop an argument from an empty expression stack. The expression stack is the set of locations whose addresses are in the range $\{D[LL]+2 \text{ to } LOSR\}$. A Stack-Underflow interrupt is generated if the address of the top-of-stack is less than $D[LL]+2$ when a pop is attempted.

Resumption conditions are implementation-defined.

Stack Structure Error

A Stack Structure Error interrupt indicates an invalid condition in the stack linkage structures used to control procedure entry, procedure exit, and move-to-stack operations. The item presented as the P2 parameter depends on the error condition, as noted.

The following notation is used:

Stack[i] = Contents at index i in the current stack.
Mem[a] = Contents of memory word at address a.
HistLink = Stack index computed from a history link.
LexLink = Address computed from a lexical link.

The operators ENTR (including aACCE and aINTE), EXIT, RETN, MVST, and aLXCH generate Stack Structure Error interrupts under any of the following conditions.

(Stack[HistLink] \neg = MSCW) or (Mem[LexLink] \neg = entered MSCW) or
(ENTR: Mem[F] = inactive MSCW) or
(EXIT,RETN: Mem[D[LL]+1] \neg = RCW) or
(aLXCH: Mem[LexLink to level i].lex_level \neg = i)
(MVST: Stack[0] \neg = TSCW): P2 = invalid word.

(EXIT,RETN: RCW.ll \neg = MSCW.ll) or (MVST: LL \neg = MSCW.ll), for the first entered MSCW on the historical chain whose head is the history link corresponding to the RCW or derived from the TSCW: P2 = RCW or TSCW.

(HistLink \leq 0 : P2 = MSCW containing the history link)
(EXIT,RETN,MVST: history_link = 0 in inactive MSCW: P2 is is MSCW)
(MVST: Computed F address \leq BOSR: P2 = F address)
(ENTR: S \leq F: P2 = S).

The common action aLXLK generates the interrupt when the referent is not an entered MSCW, or when the MSCW for level *i* does not have *i* in the `lex_level` field. P2 is the incorrect word.

The operators MKST (including aACCE, but not aINTE) and IMKS can optionally generate a Stack Structure Error interrupt if $S + 1 - F$ exceeds $2^{14} - 1$ or $S + 1 - BOSR$ exceeds $2^{16} - 1$. P2 is the erroneous value.

The STFF and ENTR operators (including aACCE, but not aINTE) can optionally generate a Stack Structure Error interrupt if the displacement value in the Lexical Link corresponding to the address-couple exceeds $2^{16} - 1$. P2 is the displacement value.

Type checking of a stack linkage word (MSCW) occurs whenever the word must be accessed; the check is always optional if the access is optional. Operators that update display registers may traverse part, but not necessarily all, of the lexical chain for the new environment; these operators are ENTR (including aACCE and aINTE), EXIT, RETN and MVST (see aLXCH). Operators that evaluate an NIRW (or PCW/RCW) may need to traverse part of the lexical chain if the implementation does not include a complete set of display registers (see aLXLK).

Resumption conditions are implementation-defined.

Code Segment Error

A Code Segment Error interrupt indicates that in distributing a PCW or RCW code-stream pointer, an invalid code-segment descriptor is accessed. This interrupt is generated if the item accessed at address-couple (`sdll,sdi`) is not a tag-3 word, where `sdll` and `sdi` are components of a RCW or PCW code-stream pointer. The invalid word is the P2 parameter.

The resumption condition is as follows: `ID.valid_state` assumes an implementation-defined value, `ID.this_op = 0`, and the interrupt RCW contains the new code-stream pointer.

Invalid Program Word

An Invalid Program Word interrupt indicates that a word accessed from the current code-segment is not a Program Code Word. It is generated in table-edit-mode if the word tag is not 0. In all other modes it is generated if the tag is not 3.

The invalid word is the P2 parameter.

For non-table code, the resumption condition is as follows: `ID.valid_state` assumes an implementation-defined value. `ID.this_op = 0` if the invalid word contained the first syllable of a branch target, or `ID.this_op = 1` otherwise. The interrupt RCW references the first syllable of the operator that contains a syllable in the invalid word.

For edit-table code, the resumption condition is implementation-defined, but the stack configuration must be defined to contain an updated table pointer referencing the edit operator that encountered the invalid word.

Page Structure Error

A Page Structure Error interrupt is generated when an attempt to index a paged array encounters a page descriptor which is not an unpagged, original SingleDD. P2 is a copy of the erroneous page descriptor. The resumption condition is implementation-defined.

False Assertion

The False Assertion interrupt is generated only by the ASRT operator when the stack argument is False. The one-syllable code parameter is presented as an 8-bit integer value in the P2 parameter .

The resumption condition is Continue-Next.

Alarm Interrupts

Alarm interrupts are triggered by hardware fault detection, and the RCW created by the interrupt entry will point to the operator that was executing when the fault was detected.

The Alarm ID parameter identifies the type of interrupt and indicates whether or not the interrupted operator may be retried. If the stack state at the time of the interrupt is still consistent with the required initial state for the operator, and no global system state has been irreparably altered, the resumption condition is Repeat-IR; otherwise it is Defunct. More than one fault condition may be reported by a single Alarm interrupt.

Alarm ID (int__class = 2)

	[15:11]	Reserved
int__type	[4: 5]	The type of Alarm interrupt composed of: invalid__addr [4: 1] = 1 = Invalid Address mem__error [3: 1] = 1 = Uncorrectable Memory Error hardware__error [1: 1] = 1 = Hardware Error loop__timer [0: 1] = 1 = Loop timer

Invalid Address

This interrupt indicates an attempt to address a word of memory that does not exist on the system. P2 contains the address.

Uncorrectable Memory Error

The P2 parameter identifies the memory address and the nature of the error. Single-bit read data errors are corrected by hardware and are not reported by an interrupt, unless correction is disabled.

P2 parameter:

	[47: 7]	The memory error field composed of: mem__single__bit[43: 1] = 1 = Single-Bit Read data Error mem__multi__bit [42: 1] = 1 = Multiple-Bit Read data Error
syndrome	[39: 8]	addr__PE [41: 1] = 1 = Address-Parity Error Reserved for implementation definition address [31:32] = The implementation-defined memory address for the memory operation

Loop Timer

This interrupt indicates an effectively infinite loop by an operator. It is triggered by expiration of a timer whose interval is sufficient for valid execution of any operator. A Loop Timer interrupt indicates an operator fault, with two possible exceptions:

1. Reference chain evaluation is nonterminating if the chain loops.
2. The LLLU (linked list lookup) operator may encounter a data-driven, nonterminating loop.

The following operators are not subject to the Loop Timer interrupt; HALT (when the Halt Boolean is TRUE), STOP, IDLE, and PAUS.

Hardware Error

This interrupt indicates a hardware-detected error that is uncorrectable. The P2 parameter is implementation-defined.

External Interrupts

External interrupts are invoked between operators to report events that are independent of the executing code-stream.

The External ID parameter identifies the type of interrupt. More than one external event may be reported by a single External interrupt. The resumption condition is Continue; the interrupt RCW references the next operator in the interrupted code-stream.

External ID (int__class = 4)

	[15:11]	Reserved
int__type	[4: 5]	The type of External Interrupt composed of: run__timeout [4: 1] = 1 = Running Timeout unmasked__attn [3: 1] = 1 = Unmasked Attention IO [2: 1] = 1 = I/O Finished attn [1: 1] = 1 = Attention int__timer [0: 1] = 1 = Interval Timer

External interrupts are masked by the CS (control state) flip-flop, except for Unmasked Attention and Running Timeout. External interrupt cannot occur between a VARI and the subsequent variant operator syllable, or between an enter-single-edit operator and the subsequent edit-mode operator.

APPENDIX A OPERATOR SET

GENERAL INFORMATION

This appendix contains two tables, which list the operators described in section 3 of this manual. The common actions described in section 3 are not included in the tables.

Table A-1 lists operators in alphabetic order according to the formal description of the operation. For each operator, the corresponding mnemonic and hexadecimal code-string value are given.

Table A-2 lists operators in hexadecimal code-string value order, in Mode sequence. All Primary-Mode operators are listed, followed by all Variant-Mode operators, followed by all Edit-Mode operators. For each operator hexadecimal code, the corresponding formal description name and mnemonic are given.

These two tables contain the same data, collated in different ways. Thus, two different approaches can be used to obtain corresponding data about any operator in the Operator Set repertoire.

Table A-1. Operators, Alphabetical List

Operator Name	Mnemonic	Hexidecimal
ADD	ADD	80
ARITHMETIC MAXIMUM	AMAX	958A
ARITHMETIC MINIMUM	AMIN	9588
ASSERT	ASRT	9580
BINARY CONVERT TO DECIMAL	BCD	9577
BIT RESET	BRST	9E
BIT SET	BSET	96
BRANCH FALSE	BRFL	A0
BRANCH TRUE	BRTR	A1
BRANCH UNCONDITIONAL	BRUN	A2
CHANGE SIGN BIT	CHSN	8E
COMPARE CHARACTERS EQUAL DELETE	CEQD	F4
COMPARE CHARACTERS EQUAL UPDATE	CEQU	FC
COMPARE CHARACTERS GREATER OR EQUAL DELETE	CGED	F1
COMPARE CHARACTERS GREATER OR EQUAL UPDATE	CGEU	F9
COMPARE CHARACTERS GREATER DELETE	CGTD	F2
COMPARE CHARACTERS GREATER UPDATE	CGTU	FA
COMPARE CHARACTERS LESS OR EQUAL DELETE	CLED	F3
COMPARE CHARACTERS LESS OR EQUAL UPDATE	EU	FB
COMPARE CHARACTERS LESS DELETE	CLSD	F0
COMPARE CHARACTERS LESS UPDATE	CLSU	F8
COMPARE CHARACTERS NOT EQUAL DELETE	CNED	F5
COMPARE CHARACTERS NOT EQUAL UPDATE	CNEU	FD
CONDITIONAL LOCK INTERLOCK	LOKC	95B1
CONDITIONAL PROCESSOR HALT	HALT	95DF
COUNT BINARY ONES	CBON	95BB
COMMUNICATE WITH UNIVERSAL I/O	CUIO	954C

Table A-1. Operators, Alphabetical List (Cont)

Operator Name	Mnemonic	Hexidecimal
DELAY	DLAY	95F6
DELETE TOP-OF-STACK	DLET	B5
DISABLE EXTERNAL INTERRUPT	DEXI	9547
DIVIDE	DIVD	83
DUPLICATE TOP-OF-STACK	DUPL	B7
DYNAMIC BINARY CONVERT TO DECIMAL	DBCD	957F
DYNAMIC BIT RESET	DBRS	9F
DYNAMIC BIT SET	DBST	97
DYNAMIC BRANCH FALSE	DBFL	A8
DYNAMIC BRANCH TRUE	DBTR	A9
DYNAMIC BRANCH UNCONDITIONAL	DBUN	AA
DYNAMIC FIELD INSERT	DINS	9D
DYNAMIC FIELD ISOLATE	DISO	9B
DYNAMIC FIELD TRANSFER	DFTR	99
DYNAMIC RANGE TEST	DRNT	9583
DYNAMIC SCALE LEFT	DSLFL	C1
DYNAMIC SCALE RIGHT FINAL	DSRF	C7
DYNAMIC SCALE RIGHT ROUND	DSRR	C9
DYNAMIC SCALE RIGHT SAVE	DSRS	C5
DYNAMIC SCALE RIGHT TRUNCATE	DSRT	C3
ENABLE EXTERNAL INTERRUPTS	EEXI	9546
END EDIT (Edit-Mode)	ENDE	DE
END FLOAT (Edit Mode)	ENDF	D5
ENTER	ENTR	AB
EQUAL	EQL	8C
EVALUATE	EVAL	AC
EXCHANGE	EXCH	B6
EXECUTE SINGLE MICRO, SINGLE POINTER UPDATE	EXPU	DD
EXECUTE SINGLE MICRO DELETE	EXSD	D2
EXECUTE SINGLE MICRO UPDATE	EXSU	DA
EXIT	EXIT	A3
FIELD INSERT	INSR	9C
FIELD ISOLATE	ISOL	9A
FIELD TRANSFER	FLTR	98
GREATER THAN	GRTR	8A
GREATER THAN OR EQUAL	GREQ	89
IDLE UNTIL INTERRUPT	IDLE	9544
INDEX	INDX	A6
INDEX AND LOAD NAME	NXLN	A5
INDEX AND LOAD VALUE	NXLV	AD
INDEX AND LOAD VALUE VIA ADDRESS COUPLE	NXVA	EF
INDEX VIA ADDRESS COUPLE	INXA	E7
INPUT CONVERT DELETE	ICVD	CA
INPUT CONVERT LEFT-SIGNED DELETE	ICLD	9575
INPUT CONVERT RIGHT-SIGNED DELETE	ICRD	9576
INPUT CONVERT UNSIGNED DELETE	ICUD	A4
INPUT CONVERT UPDATE	ICVU	CB
INTRODUCE VARIANT OPERATOR	VARI	95
INSERT CONDITIONAL (Edit-Mode)	INSC	DD

Table A-1. Operators, Alphabetical List (Cont)

Operator Name	Mnemonic	Hexidecimal
INSERT DISPLAY SIGN (Edit-Mode)	INSG	D9
INSERT MARK STACK	IMKS	CF
INSERT OVERPUNCH (Edit-Mode)	INOP	D8
INSERT UNCONDITIONAL (Edit-Mode)	INSU	DC
INTEGER DIVIDE	IDIV	84
INTEGERIZE DOUBLE-PRECISION ROUNDED	NTGD	9587
INTEGERIZE DOUBLE-PRECISION TRUNCATED	NTTD	9586
INTERGERIZE ROUNDED	NTGR	87
INTERGERIZE TRUNCATED	NTIA	86
INVALID OPERATOR	NVLD	FF
INVALID OPERATOR	NVLD	95FF
LEADING ONE TEST	LOG2	958B
LINKED LIST LOOKUP	LLLU	95BD
LESS THAN	LESS	88
LESS THAN OR EQUAL	LSEQ	8B
LITERAL CALL ONE	ONE	B1
LITERAL CALL ZERO	ZERO	B0
LITERAL CALL 8-BITS	LT8	B2
LITERAL CALL 16-BITS	LT16	B3
LITERAL CALL 48-BITS	LT48	BE
LOAD	LOAD	BD
LOAD TRANSPARENT	LODT	BC
LOAD TRANSPARENT	LODT	95BC
LOCK INTERLOCK	LOK	95B0
LOGICAL AND	LAND	90
LOGICAL EQUAL	SAME	94
LOGICAL EQUALITY	LEQV	93
LOGICAL NEGATE	LNOT	92
LOGICAL OR	LOR	91
LONG NAME CALL	LNMC	F6
LONG VALUE CALL	LVLC	F7
MAKE PROGRAM CONTROL WORD	MPCW	BF
MARK STACK BOUND TO NAME CALL	MKSN	DF
MARK STACK	MKST	AE
MASKED SEARCH FOR EQUAL	SRCH	95BE
MOVE CHARACTERS (Edit-Mode)	MCHR	D7
MOVE NUMERIC UNCONDITIONAL (Edit-Mode)	MVNU	D6
MOVE TO STACK	MVST	95AF
MOVE WITH FLOAT (Edit-Mode)	MFLT	D1
MOVE WITH INSERT (Edit-Mode)	MINS	D0
MULTIPLY	MULT	82
MULTIPLY EXTENDED	MULX	8F
NAME CALL	NAMC	40 to 7F
NO OPERATION	NOOP	FE
NO OPERATION	NOOP	95FE
NORMALIZE	NORM	958E
NOT EQUAL	NEQL	8D
OCCURS INDEX	OCRX	9585

Table A-1. Operators, Alphabetical List (Cont)

Operator Name	Mnemonic	Hexidecimal
OVERWRITE DELETE	OVRD	BA
OVERWRITE NON-DELETE	OV RN	BB
PACK DELETE	PACD	D1
PACK LEFT-SIGNED	PKLD	9573
PACK RIGHT-SIGNED	PKRD	9574
PACK UNSIGNED	PKUD	9572
PACK UPDATE	PACU	D9
PAUSE UNTIL INTERRUPT	PAUS	9584
PRIMITIVE DISPLAY	SHOW	95DE
PUSH DOWN STACK REGISTERS	PUSH	B4
RANGE TEST	RNGT	9582
READ AND CLEAR OVERFLOW FLIP-FLOP	ROFF	D7
READ EXTERNAL MEMORY CONTROL	REMC	9592
READ INTERLOCK STATUS	LKID	95B3
READ INTERNAL PROCESSOR STATE	RIPS	9598
READ MACHINE IDENTIFICATION	WATI	95A4
READ PROCESSOR IDENTIFICATION	WHOI	954E
READ PROCESSOR REGISTER	RPRR	95B8
READ STACK NUMBER	RSNR	9581
READ TAG FIELD	RTAG	95B5
READ TIME-OF-DAY CLOCK	RTOD	95A7
READ TRUE/FALSE FLIP-FLOP	RTFF	DE
READ WITH LOCK	RDLK	95BA
REMAINDER DIVIDE	RDIV	85
RESET FLOAT (Edit-Mode)	RSTF	D4
RETURN	RETN	A7
ROTATE STACK DOWN	RSDN	95B7
ROTATE STACK UP	RSUP	95B6
RUNNING INDICATOR	RUNI	9541
SCALE LEFT	SCLF	C0
SCALE RIGHT FINAL	SCR F	C6
SCALE RIGHT ROUNDED	SCRR	C8
SCALE RIGHT SAVE	SCRS	C4
SCALE RIGHT TRUNCATE	SCRT	C2
SCAN WHILE EQUAL DELETE	SEQD	954F
SCAN WHILE EQUAL UPDATE	SEQU	95FC
SCAN WHILE FALSE DELETE	SWFD	95D4
SCAN WHILE FALSE UPDATE	SWFU	95DC
SCAN WHILE GREATER OR EQUAL DELETE	SGED	95F1
SCAN WHILE GREATER OR EQUAL UPDATE	SGEU	95F9
SCAN WHILE GREATER DELETE	SGTD	95F2
SCAN WHILE GREATER UPDATE	SGTU	95FA
SCAN WHILE LESS OR EQUAL DELETE	SLED	95F3
SCAN WHILE LESS OR EQUAL UPDATE	SLEU	95FB
SCAN WHILE LESS DELETE	SLSD	95F0
SCAN WHILE LESS UPDATE	SLSU	95F8
SCAN WHILE NOT EQUAL DELETE	SNED	95F5
SCAN WHILE NOT EQUAL UPDATE	SNEU	95FD
SCAN WHILE TRUE DELETE	SWTD	95D5

Table A-1. Operators, Alphabetical List (Cont)

Operator Name	Mnemonic	Hexidecimal
SCAN WHILE TRUE UPDATE	SWTU	95DD
SET DOUBLE TO TWO SINGLES	SPLT	9543
SET EXTERNAL SIGN FLIP-FLOP	SXSN	D6
SET INTERVAL TIMER	SINT	9545
SET PROCESSOR REGISTER	SPRR	95B9
SET TAG FIELD	STAG	95B4
SET TO DOUBLE-PRECISION	XTND	CE
SET TO SINGLE-PRECISION ROUNDED	SNGL	CD
SET TO SINGLE-PRECISION TRUNCATED	SNGT	CC
SET TWO SINGLES TO DOUBLE	JOIN	9542
SKIP FORWARD DESTINATION CHARACTERS (Edit-Mode)	SFDC	DA
SKIP FORWARD SOURCE CHARACTERS (Edit-Mode)	SFSC	D2
SKIP REVERSE DESTINATION CHARACTERS (Edit-Mode)	SRDC	DB
SKIP REVERSE SOURCE CHARACTERS (Edit-Mode)	SRSC	D3
STORE DELETE	STOD	B8
STORE DELETE VIA ADDRESS COUPLE	STAD	F6
STORE NON-DELETE	STON	B9
STORE NON-DELETE VIA ADDRESS COUPLE	STAN	F7
STRING ISOLATE	SISO	D5
STUFF ENVIRONMENT	STFF	AF
SUBTRACT	SUBT	81
TABLE ENTER EDIT DELETE	TSSD	D0
TABLE ENTER EDIT UPDATE	TEEU	D8
TRANSFER CHARACTERS UNCONDITIONAL DELETE	TUND	E6
TRANSFER CHARACTERS UNCONDITIONAL UPDATE	TUNU	EE
TRANSFER WHILE EQUAL DELETE	TEQD	E4
TRANSFER WHILE EQUAL UPDATE	TEQU	EC
TRANSFER WHILE FALSE DELETE	TWFD	95D2
TRANSFER WHILE FALSE UPDATE	TWFU	95DA
TRANSFER WHILE GREATER OR EQUAL DELETE	TGED	E1
TRANSFER WHILE GREATER OR EQUAL UPDATE	TGEU	E9
TRANSFER WHILE GREATER DELETE	TGTD	E2
TRANSFER WHILE GREATER UPDATE	TGTU	EA
TRANSFER WHILE LESS OR EQUAL DELETE	TLED	E3
TRANSFER WHILE LESS OR EQUAL UPDATE	TLEU	EB
TRANSFER WHILE LESS DELETE	TLSD	E0
TRANSFER WHILE LESS UPDATE	TLSU	E8
TRANSFER WHILE NOT EQUAL DELETE	TNED	E5
TRANSFER WHILE NOT EQUAL UPDATE	TNEU	ED
TRANSFER WHILE TRUE DELETE	TWTD	95D3
TRANSFER WHILE TRUE UPDATE	TWTU	95DB
TRANSFER WORDS OVERWRITE DELETE	TWOD	D4
TRANSFER WORDS OVERWRITE UPDATE	TWOU	DC
TRANSFER WORDS DELETE	TWSD	D3
TRANSFER WORDS UPDATE	TWSU	DB
TRANSLATE	TRNS	95D7
UNCONDITIONAL PROCESSOR HALT	STOP	95BF
UNLOCK INTERLOCK	UNLK	95B2
UNPACK LEFT-SIGNED DELETE	UPLD	9570

Table A-1. Operators, Alphabetical List (Cont)

Operator Name	Mnemonic	Hexidecimal
UNPACK LEFT-SIGNED UPDATE	UPLU	9578
UNPACK RIGHT-SIGNED DELETE	UPRD	9571
UNPACK RIGHT-SIGNED UPDATE	UPRU	9579
UNPACK UNSIGNED DELETE	UABD	95D1
UNPACK UNSIGNED UPDATE	UABU	95D9
UNPACK SIGNED DELETE	USND	95D0
UNPACKED SIGNED UPDATE	USNU	95D8
VALUE CALL	VALC	00 to 3F
WRITE EXTERNAL MEMORY CONTROL	WEMC	9593
WRITE INTERNAL PROCESSOR STATE	WIPS	9599
WRITE TIME-OF-DAY	WTOD	9549
ZERO INTERRUPT_COUNT	ZIC	9540

Table A-2. Operators, Numerical List

Hexidecimal	Operator name	Mnemonic
PRIMARY MODE		
00 thru 3F	VALUE CALL	VALC
40 thru 7F	NAME CALL	NAMC
80	ADD	ADD
81	SUBTRACT	SUBT
82	MULTIPLY	MULT
83	DIVIDE	DIVD
84	INTEGER DIVIDE	IDIV
85	REMAINDER DIVIDE	RDIV
86	INTEGERIZE TRUNCATE	NTIA
87	INTEGERIZE ROUNDED	NTGR
88	LESS THAN	LESS
89	GREATER THAN OR EQUAL	GREQ
8A	GREATER THAN	GRTR
8B	LESS THAN OR EQUAL	LSEQ
8C	EQUAL	EQL
8D	NOT EQUAL	NEQL
8E	CHANGE SIGN BIT	CHSN
8F	EXTENDED MULTIPLY	MULX
90	LOGICAL AND	LAND
91	LOGICAL OR	LOR
92	LOGICAL NEGATE	LNOT
93	LOGICAL EQUALITY	LEQV
94	LOGICAL EQUAL	SAME
95	INTRODUCE VARIANT OPERATOR	VARI
96	BIT SET	BSET
97	DYNAMIC BIT SET	DBST
98	FIELD TRANSFER	FLTR
99	DYNAMIC FIELD TRANSFER	DFTR
9A	FIELD ISOLATE	ISOL

Table A-2. Operators, Numerical List (Cont)

Hexidecimal	Operator name	Mnemonic
PRIMARY MODE		
9B	DYNAMIC FIELD ISOLATE	DISO
9C	FIELD INSERT	INSR
9D	DYNAMIC FIELD INSERT	DINS
9E	BIT RESET	BRST
9F	DYNAMIC BIT RESET	DBRS
A0	BRANCH FALSE	BRFL
A1	BRANCH TRUE	BRTR
A2	BRANCH UNCONDITIONAL	BRUN
A3	EXIT	EXIT
A4	INPUT CONVERT UNSIGNED DELETE	ICUD
A5	INDEX AND LOAD NAME	NXLN
A6	INDEX	INDX
A7	RETURN	RETN
A8	DYNAMIC BRANCH FALSE	DBFL
A9	DYNAMIC BRANCH TRUE	DBTR
AA	DYNAMIC BRANCH UNCONDITIONAL	DBUN
AB	ENTER	ENTR
AC	EVALUATE DESCRIPTOR	EVAL
AD	INDEX AND LOAD VALUE	NXLV
AE	MARK STACK	MKST
AF	STUFF ENVIRONMENT	STFF
B0	LITERAL CALL ZERO	ZERO
B1	LITERAL CALL ONE	ONE
B2	LITERAL CALL 8-BITS	LT8
B3	LITERAL CALL 16-BITS	LT16
B4	PUSH DOWN STACK REGISTERS	PUSH
B5	DELETE TOP-OF-STACK	DLET
B6	EXCHANGE	EXCH
B7	DUPLICATE TOP-OF-STACK	DUPL
B8	STORE DELETE	STOD
B9	STORE NON-DELETE	STON
BA	OVERWRITE DELETE	OVRD
BB	OVERWRITE NON-DELETE	OVRN
BC	LOAD TRANSPARENT	LODT
BD	LOAD	LOAD
BE	LITERAL CALL 48-BITS	LT48
BF	MAKE PROGRAM CONTROL WORD	MPCW
C0	SCALE LEFT	SCLF
C1	DYNAMIC SCALE LEFT	DSLFL
C2	SCALE RIGHT TRUNCATE	SCRT
C3	DYNAMIC SCALE RIGHT TRUNCATE	DSRT
C4	SCALE RIGHT SAVE	SCRS
C5	DYNAMIC SCALE RIGHT SAVE	DSRS
C6	SCALE RIGHT FINAL	SCRF
C7	DYNAMIC SCALE RIGHT FINAL	DSRF
C8	SCALE RIGHT ROUNDED	SCRR
C9	DYNAMIC SCALE RIGHT ROUNDED	DSRR
CA	INPUT CONVERT DELETE	ICVD

Table A-2. Operators, Numerical List (Cont)

Hexidecimal	Operator name	Mnemonic
PRIMARY MODE		
CB	INPUT CONVERT UPDATE	ICVU
CC	SET TO SINGLE-PRECISION TRUNCATED	SNGT
CD	SET TO SINGLE-PRECISION ROUNDED	SNGL
CE	SET TO DOUBLE-PRECISION	XTND
CF	INSERT MARK STACK	IMKS
D0	TABLE ENTER EDIT DELETE	TEED
D1	PACK DESTRUCTIVE	PACD
D2	EXECUTE SINGLE MICRO DELETE	EXSD
D3	TRANSFER WORDS DESTRUCTIVE	TWSD
D4	TRANSFER WORDS OVERWRITE DELETE	TWOD
D5	STRING ISOLATE	SISO
D6	SET EXTERNAL SIGN FLIP-FLOP	SXSN
D7	READ AND CLEAR OVERFLOW FLIP-FLOP	ROFF
D8	TABLE ENTER EDIT UPDATE	TEEU
D9	PACK UPDATE	PACU
DA	EXECUTE SINGLE MICRO UPDATE	EXSU
DB	TRANSFER WORDS UPDATE	TWSU
DC	TRANSFER WORDS OVERWRITE UPDATE	TWOU
DD	EXECUTE SINGLE MICRO SINGLE POINTER UPDATE	EXPU
DE	READ TRUE/FALSE FLIP-FLOP	TRFF
DF	MARK STACK BOUND TO NAME CALL	MKSN
E0	TRANSFER WHILE LESS DELETE	TLSD
E1	TRANSFER WHILE GREATER OR EQUAL DELETE	TGED
E2	TRANSFER WHILE GREATER DELETE	TGTD
E3	TRANSFER WHILE LESS OR EQUAL DELETE	TLED
E4	TRANSFER WHILE EQUAL DELETE	TEQD
E5	TRANSFER WHILE NOT EQUAL DELETE	TNED
E6	TRANSFER CHARACTERS UNCONDITIONAL DELETE	TUND
E7	INDEX VIA ADDRESS COUPLE	INXA
E8	TRANSFER WHILE LESS UPDATE	TLSU
E9	TRANSFER WHILE GREATER OR EQUAL UPDATE	TGEU
EA	TRANSFER WHILE GREATER UPDATE	TGTU
EB	TRANSFER WHILE LESS OR EQUAL UPDATE	TLEU
EC	TRANSFER WHILE EQUAL UPDATE	TEQU
ED	TRANSFER WHILE NOT EQUAL UPDATE	TNEU
EE	TRANSFER CHARACTERS UNCONDITIONAL UPDATE	TUNU
EF	INDEX AND LOAD VALUE VIA ADDRESS COUPLE	NXVA
F0	COMPARE CHARACTERS LESS DELETE	CLSD
F1	COMPARE CHARACTERS GREATER OR EQUAL DELETE	CGED
F2	COMPARE CHARACTERS GREATER DELETE	CGTD
F3	COMPARE CHARACTERS LESS OR EQUAL DELETE	CLED
F4	COMPARE CHARACTERS EQUAL DELETE	CEQD

Table A-2. Operators, Numerical List (Cont)

Hexidecimal	Operator name	Mnemonic
PRIMARY MODE		
F5	COMPARE CHARACTERS NOT EQUAL DELETE	CNED
F6	STORE DELETE VIA ADDRESS COUPLE	STAD
F7	STORE NON-DELETE VIA ADDRESS COUPLE	STAN
F8	COMPARE CHARACTERS LESS UPDATE	CLSU
F9	COMPARE CHARACTERS GREATER OR EQUAL UPDATE	CGEU
FA	COMPARE CHARACTERS GREATER UPDATE	CGTU
FB	COMPARE CHARACTERS LESS OR EQUAL UPDATE	CLEU FC
FD	COMPARE CHARACTERS NOT EQUAL UPDATE	CNEU
FE	NO OPERATION	NOOP
FF	INVALID OPERATOR	NVLD
VARIANT MODE		
9540	ZERO INTERRUPT_COUNT	ZIC
9541	RUNNING INDICATOR	RUNI
9542	SET TWO SINGLES TO DOUBLE	JOIN
9543	SET DOUBLE TO TWO SINGLES	SPLT
9544	IDLE UNTIL INTERRUPT	IDLE
9545	SET INTERVAL TIMER	SINT
9546	ENABLE EXTERNAL INTERRUPTS	EEXI
9547	DISABLE EXTERNAL INTERRUPTS	DEXI
9549	WRITE TIME-OF-DAY	WTOD
954C	COMMUNICATE WITH UNIVERSAL I/O	CUIO
954E	READ PROCESSOR IDENTIFICATION	WHOI
9570	UNPACK LEFT-SIGNED DELETE	UPLD
9571	UNPACK RIGHT-SIGNED DELETE	UPRD
9572	PACK UNSIGNED	PKUD
9573	PACK LEFT-SIGNED	PKLD
9574	PACK RIGHT-SIGNED	PKRD
9575	INPUT CONVERT LEFT-SIGNED DELETE	ICLD
9576	INPUT CONVERT RIGHT-SIGNED DELETE	ICRD
9577	BINARY CONVERT TO DECIMAL	BCD
9578	UNPACK LEFT-SIGNED UPDATE	UPLU
9579	UNPACK RIGHT-SIGNED UPDATE	UPRU
957F	DYNAMIC BINARY CONVERT TO DECIMAL	DBCDD
9580	ASSERT	ASRT
9581	READ STACK NUMBER	RSNR
9582	RANGE TEST	RNGT
9583	DYNAMIC RANGE TEST	DRNT
9584	PAUSE UNTIL INTERRUPT	PAUS
9585	OCCURS INDEX	OCRX
9586	INTEGERIZE, DOUBLE-PRECISION, TRUNCATED	NTTD
9587	INTEGERIZE, DOUBLE-PRECISION, ROUNDED	NTGD
9588	ARITHMETIC MINIMUM	AMIN
958A	ARITHMETIC MAXIMUM	AMAX

Table A-2. Operators, Numerical List (Cont)

Hexidecimal	Operator name	Mnemonic
VARIANT MODE		
958B	LEADING ONE TEST	LOG2
958C	LONG NAME CALL	LNMC
958D	LONG VALUE CALL	LVLC
958E	NORMALIZE	NORM
9592	READ EXTERNAL MEMORY CONTROL	REMC
9593	WRITE EXTERNAL MEMORY CONTROL	WEMC
9598	Read INTERNAL PROCESSOR STATE	RIPS
9599	WRITE INTERNAL PROCESSOR STATE	WIPS
95A4	WHAT MACHINE IDENTIFICATION	WATI
95A7	READ TIME-OF-DAY	RTOD
95AF	MOVE TO STACK	MVST
95B0	LOCK INTERLOCK	LOK
95B1	CONDITIONAL LOCK INTERLOCK	LOKC
95B2	UNLOCK INTERLOCK	UNLK
95B3	READ INTERLOCK STATUS	LKID
95B4	SET TAG FIELD	STAG
95B5	READ TAG FIELD	RTAG
95B6	ROTATE STACK UP	RSUP
95B7	ROTATE STACK DOWN	RSDN
95B8	READ PROCESSOR REGISTER	RPRR
95B9	SET PROCESOR REGISTER	SPRR
95BA	READ WITH LOCK	RDLK
95BB	COUNT BINARY ONES	CBON
95BC	LOAD TRANSPARENT	LODT
95BD	LINKED LIST LOOK-UP	LLU
95BE	MASKED SEARCH FOR EQUAL	SRCH
95BF	UNCONDITIONAL PROCESSOR HALT	STOP
95D0	UNPACK SIGNED DELETE	USND
95D1	UNPACK UNSIGNED DELETE	UPUD
95D2	TRANSFER WHILE FALSE DELETE	TWFD
95D3	TRANSFER WHILE TRUE DELETE	TWTD
95D4	SCAN WHILE FALSE DELETE	SWFD
95D5	SCAN WHILE TRUE DELETE	SWTD
95D7	TRANSLATE	TRNS
95D8	UNPACK SIGNED UPDATE	USNU
95D9	UNPACK UNSIGNED UPDATE	UPUU
95DA	TRANSFER WHILE FALSE UPDATE	TWFU
95DB	TRANSFER WHILE TRUE UPDATE	TWTU
95DC	SCAN WHILE FALSE UPDATE	SWFU
95DD	SCAN WHILE TRUE UPDATE	SWTU
95DE	PRIMITIVE DISPLAY	SHOW
95DF	CONDITIONAL HALT	HALT
95F0	SCAN WHILE LESS DELETE	SLSD
95F1	SCAN WHILE GREATER OR EQUAL DELETE	SGED
95F2	SCAN WHILE GREATER DELETE	SGTD
95F3	SCAN WHILE LESS OR EQUAL DELETE	SLED
95F4	SCAN WHILE EQUAL DELETE	SEQD

Table A-2. Operators, Numerical List

Hexidecimal	Operator name	Mnemonic
VARIANT MODE		
95F5	SCAN WHILE NOT EQUAL DELETE	SNED
95F6	DELAY	DLAY
95F8	SCAN WHILE LESS UPDATE	SLSU
95F9	SCAN WHILE GREATER OR EQUAL UPDATE	SGEU
95FA	SCAN WHILE GREATER UPDATE	SGTU
95FB	SCAN WHILE LESS OR EQUAL UPDATE	SLEU
95FC	SCAN WHILE EQUAL UPDATE SEQU	
95FD	SCAN WHILE NOT EQUAL UPDATE	SNEU
95FE	NO OPERATION	NOOP
95FF	INVALID OPERATOR	NVLD
 EDIT MODE		
D0	MOVE WITH INSERT	MINS
D1	MOVE WITH FLOAT	MFLT
D2	SKIP FORWARD SOURCE CHARACTERS	SFSC
D3	SKIP REVERSE SOURCE CHARACTERS	SRSC
D4	RESET FLOAT	RSTF
D5	END FLOAT	ENDF
D6	MOVE NUMERIC UNCONDITIONAL	MVNU
D7	MOVE CHARACTERS	MCHR
D8	INSERT OVERPUNCH	INOP
D9	INSERT DISPLAY SIGN	INSG
DA	SKIP FORWARD DESTINATION CHARACTERS	SFDC
DB	SKIP REVERSE DESTINATION CHARACTERS	SRDC
DC	INSERT UNCONDITIONAL	INSU
DD	INSERT CONDITIONAL	INSC
DE	END EDIT	ENDE
DF	CONDITIONAL PROCESSOR HALT	HALT

APPENDIX B

OPERATOR REFERENCE SUMMARIES

GENERAL INFORMATION

Operators and common actions are listed alphabetically, and for each operator, the following information is given.

The Code-Stream Encoding Of The Operator

The number of code-stream syllables required by the operator is shown, and the operator encoding is defined to be a sequence composed of the opcode literal and, optionally, one or more parameters. The opcode literal is shown as a hexadecimal-string (such as "A3"), or, in some cases, as a binary value (such as 01). Parameters are specified as name:number-of-bits (such as op__psi:3). If the operator is not a primary-mode operator, the interpretation mode is shown following the operator encoding.

Clients

For most common actions, the invoking operators are listed.

Stack State Transformation

Stack state transformations are shown by diagrams, which illustrate inputs and outputs in order from top-of-stack downward:

Input items → Output items

The effect of the stack state transformation is that all inputs are consumed and all outputs are created. "Null →" and "→ Null" indicate that the operator has no inputs or outputs, respectively. There is an implicit invariant: the item that was on the stack below the lowest input (if any) remains on the stack below the lowest output (if any), unaffected by the stack state transformation of the operator. The input items are shown for the initial state of the operator; some operators have additional stack input arguments in restart state.

Initial stack items are denoted id: type(interpretation), where id and interpretation are optional. Type may be a data type as defined in section 1 of this manual, "Any" indicating no type restriction, or "*" indicating a required set of types as defined under Invalid Stack Argument interrupt. Id, if included, is a distinguishing name indicating how the item is to be used and establishing a reference for the final stack state. Interpretation is explicitly included if multiple interpretations of the type are possible.

Final stack items may be denoted type(interpretation), id, id', or as a set of types. Type(interpretation) is defined for initial stack items. Id and id' indicate the initial stack item of that name, the latter case having a modified value.

Interrupts That May Be Generated

Where relevant, a brief statement of the conditions under which the interrupt may occur is included. See Appendix C for explanation of the condition notation used and definition of nonstandard terms and abbreviations.

Symbols Used In This Appendix

Certain symbols used in this appendix represent status conditions, computation results, and comparison requirements. Other symbols that represent subset ranges, and subset units are also used throughout this appendix. These are as follows:

Symbol	Meaning
$\{a,b,c\}$	The set including items a, b, and c. All item relationships are proper. Item magnitude is not implied by the listing order.
$[m:n]$	The set of bits starting with m, the most-significant bit, that extends downward for n bits (including m).
$[A + 1]$	The value of A, incremented +1.
$a \rightarrow b$	A linkage from position a that results in position b, or the value of b that results from use of a. If b is a set, a maps into the set b.
$a \nrightarrow \text{in } \{b,c,d\}$	a not present in set $\{b,c,d\}$.
$a \nrightarrow b$	b not linked or obtained by use of a.
$a < b$	a is less than b.
$a > b$	a is greater than b.
$a \nrightarrow = b$	a is not equal to b.
$a \leq b$	a is less than equal to b.
$a \geq b$	a is equal to or greater than b.
$a \nrightarrow \geq b$	a is not equal to or greater than b.
<code>element__size</code>	<code>SIRW.Lexical__Link</code> The <code>Lexical__Link</code> field in a SIRW word. <code>element__size</code> bit.

OPERATOR AND COMMON ACTION LISTING

The following is a listing of all operators and common actions.

aACCE

Name:	accidental entry
Encoding:	none (common action)
Clients:	EVAL LVLC STAD STAN STOD STON VALC
Stack State transformation:	See the functional definition of aACCE in section 3
Interrupts:	
Inv Arg Value:	Mem[F + 1] = NIRW directly to PCW and PCW.ll > 0 and PCW.ll - 1 \neg = NIRW.lambda or PCW.ll \neg in {0, MSCW.ll + 1} or PCW.invalid_ll \neg = 0
Stack-Overflow	(new F) - (old F) \neg in {1 to 2**14 - 1} (MKST)
Stack Structure:	or (new F) - BOSR \neg in {0 to 2**16 - 1} \div or new displacement \neg in {1 to 2**16 - 1} \div or SIRW.lexical_link \neg \geq entered MSCW

Also see aLXCH - display update; aPRCW - code-stream pointer distribution.

aCPY

Name:	fetch copy descriptor
Encoding:	none (common action)
Clients:	INDX INXA LOAD NXLN: generation of Presence Bit, Invalid Object, Invalid Reference Chain, and Binding Request interrupts
Stack state transformation:	not applicable
Interrupts:	none

aFOP

Name: fetch operand value
 Encoding: none (common action)
 Clients: LOAD NXLV NXVA LVLC VALC
 Stack state transformation: not applicable
 Interrupts:
 Inv Object: second-word of double (obtained by means of IRW) has tag $\neg = 2$
 or second-word of double (obtained by means of IndexedDD) has odd tag

aINTE

Name: interrupt entry
 Encoding: none (common action)
 Clients: all operators and actions that generate interrupts
 Stack state transformation: See the functional definition of aINTE in section 3
 Interrupts:
 Binding Request: IRW chain \rightarrow DD with element__size = 7
 Inv Arg Value: Mem[F + 1] = NIRW directly to PCW and PCW.ll >
 and PCW.ll - 1 $\neg =$ NIRW.lambda
 or PCW.ll \neg in {0, MSCW.ll + 1}
 or PCW.invalid__ll $\neg = 0$
 Inv Ref Chain: IRW chain = \rightarrow
 (PCW, or DD with element__size = 7)
 Inv Stack Arg: M[F + 1] (from interrupt__reference) not IRW
 Stack-Overflow Stack Structure: $S \leq F$ or SIRW.lexical__link $\neg \geq$ entered
 MSCW

Also see aLXCH – display update; NIRW evaluation; aLXLK – SIRW evaluation; aPRCW – code-stream pointer distribution.

aISX

Name: integer subset exception
 Encoding: none (common action)
 Clients: MVST RPRR SINT SPRR WTOD
 Stack state transformation: not applicable
 Interrupts:
 Int Overflow: argument not sp integer (*)
 Inv Arg Value: argument not k-bit integer (*)
 Inv Stack Arg: argument not operand or not k-bit integer

(* Int Overflow and Inv Arg Value are alternatives for some cases of Inv Stack Arg: see action definition.)

aLXCH

Name: traverse lexical chain
 Encoding: none (common action)
 Clients: address-couple parameter evaluation:
 INXA NXVA LVLC MKSN-NAMC STAD
 STAN VALC NIRW
 address-couple evaluation: aINTE DBFL DBTR DBUN ENTR EVAL
 INDX LKID LOAD LODT LOK LOKC
 NXLN NXLV OVRD OVRN RDLK STFF
 STOD STON UNLK (sdll,sdi) address-couple
 evaluation: aPRCW
 display update: aACCE aINTE ENTR EXIT MVST RETN
 Stack state transformation: not applicable
 Interrupts:
 Stack Structure: For i in levels traversed:
 Mem[LexLink to level i] \neg = entered MSCW
 or MSCW.lex__level \neg = i

Also see aLXLK (MSCW.stack__number, MSCW.displacement).

aLXLK

Name: evaluate lexical link
 Encoding: none (common action)
 Clients: MSCW lexical link evaluation: aLXCH
 SIRW evaluation: aINTE ENTR EVAL INDX INXA LKID
 LOAD LODT LOK LOKC LVLC MKSN-
 NAMC NXLN NXLV NXVA OVRD OVRN
 RDLK STAD STAN STOD STON VALC
 UNLK
 Stack state transformation: not applicable
 Interrupts:
 Inv Index: stack__number \neg in {0 to SVD.length-1}
 Inv Object: Mem[AbsentCopyDD.address] not original DD
 or stack-vector descriptor not unpagged original
 SingleDD or stack descriptor not unpagged
 unindexed SingleDD
 Presence Bit: stack descriptor

aPRCW

Name: distribute PCW/RCW code-stream pointer
 Encoding: none (common action)
 Clients: aACCE aINTE DBFL DBTR DBUN ENTR
 EXIT RETN
 Stack state transformation: not applicable
 Interrupts:
 Code Seg Error
 Inv Arg Value: (PCW or RCW).psi \neg in {0 to 5}
 Inv Index: (PCW or RCW).pwi \neg in {0 to
 CSD.seg__length-1}
 Presence Bit: code-segment

ADD

Name: add
Encoding: 1 syllable ("80")

Stack state transformation: $\left| \begin{array}{c} \text{opnd (numer ic)} \\ \text{opnd (numer ic)} \end{array} \right| \Rightarrow \left| \text{opnd (numer ic)} \right|$

Interrupts:
Exp Overflow: $R(x+y) = \text{exponent value too big}$
Inv Stack Arg: TOS not opnd or TOS2 not opnd Stack-Underflow

AMAX

Name: arithmetic maximum
Encoding: 2 syllables ("958A") Variant

Stack state transformation: $\left| \begin{array}{c} \text{opnd (numer ic)} \\ \text{opnd (numer ic)} \end{array} \right| \Rightarrow \left| \text{opnd (numer ic)} \right|$

Interrupts:
Inv Stack Arg: TOS not opnd or TOS2 not opnd
Stack-Underflow

AMIN

Name: arithmetic minimum
Encoding: 2 syllables ("9588") Variant
Otherwise see AMAX.

ASRT

Name: assert

Encoding: 3 syllables ("9580", interrupt_code:8) Variant

Stack state transformation: $\left| \begin{array}{c} \text{opnd (Boolean)} \end{array} \right| \Rightarrow \text{Null}$

Interrupts:

Inv Stack Arg: TOS not operand

False Assertion: opnd NOT Boolean True

Stack-Underflow

BCD

Name: binary convert to decimal

Encoding: 3 syllables ("9577", N:8) Variant

Stack state transformation: $\left| \begin{array}{c} \text{opnd (numeric)} \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{opnd (BCD)} \end{array} \right|$

Interrupts:

Int Overflow: TOS not dp integer

Inv Code Param: $N > 24$

Inv Stack Arg: TOS not opnd

Stack-Overflow

Stack-Underflow

BRFL

Name: branch false

Encoding: 3 syllables ("A0", op__psi:3, op__pwi:13)

Stack state transformation: $\left| \begin{array}{c} \text{opnd (Boolean)} \end{array} \right| \Rightarrow \text{Null}$

Interrupts:

Inv Code Param: $\text{op_psi} > 5$

Inv Index: $\text{op_pwi} \notin \{0 \text{ to } \text{CSD.seg_length}-1\}$

Inv Stack Arg: TOS not operand

Stack-Underflow

BRTR

Name: branch true
 Encoding: 3 syllables ("A1", op__psi:3, op__pwi:13)
 Otherwise see BRFL.

BRST

Name: bit reset
 Encoding: 2 syllables ("9E", Db:8)
 Stack state transformation:

dest: any (bit-vector)	==>	dest'
------------------------	-----	-------

 Interrupts:
 Inv Code Param: Db > 47
 Stack-Underflow

BRUN

Name: branch unconditional
 Encoding: 3 syllable ("A2", op__psi:3, op__pwi:13)
 State stack transformation: none
 Interrupts:
 Inv Code Param: op__psi > 5
 Inv Index: op__pwi \neg in {0 to CSD.seg__length-1}

BSET

Name: bit set
 Encoding: 2 syllables ("96", Db:8)
 Otherwise see BRST.

CBON

Name: count binary ones
 Encoding: 2 syllables ("95BB") Variant
 Stack state transformation:

opnd (bit-vector)	==>	7-bit integer
-------------------	-----	---------------

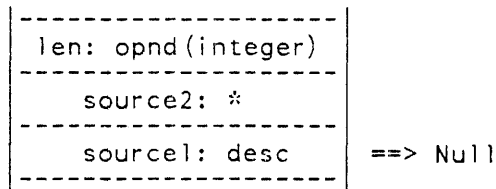
 Interrupts:
 Inv Stack Arg: TOS not opnd
 Stack-Underflow

CEQD

Name: compare characters equal delete

Encoding: 1 syllable ("F4")

Stack state transformation:



Interrupts:

Int Overflow: len not sp integer

Inv Arg Value: source1/source2 Pointer.char__index out of range or len > 2**20-1

Inv Index: source1'/source2' word index \neg in {0 to 2**16-1}

Inv Object: Mem[AbsentCopyDD.address] not original DD

Inv Stack Arg: len not opnd
or source1 not {IndexedDD, opnd}
or source2 not {IndexedDD, opnd}
or (source1 = EBCDIC(hex) and source2 = hex(EBCDIC))

Paged Array: source1 or source2 Pointer

Presence Bit: source1 or source2 Pointer

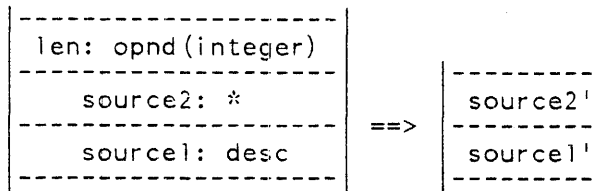
Stack-Underflow

CEQU

Name: compare characters equal update

Encoding: 1 syllable ("FC")

Stack state transformation:



Interrupts: same as CEQD.

CGED

Name: compare characters greater or equal delete

Encoding: 1 syllable ("F1")
Otherwise see CEQD.

CGEU

Name: compare characters greater or equal update
Encoding: 1 syllable ("F9")
Otherwise see CEQU.

CGTD

Name: compare characters greater delete
Encoding: 1 syllable ("F2")
Otherwise see CEQD.

CGTU

Name: compare characters greater update
Encoding: 1 syllable ("FA")
Otherwise see CEQU.

CHSN

Name: change sign
Encoding: 1 syllable ("8E")

Stack state transformation: $\left| \begin{array}{c} \text{num: opnd (numeric)} \\ \hline \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{num}' \\ \hline \end{array} \right|$

Interrupts:
Inv Stack Arg: num not opnd
Stack-Underflow

CLED

Name: compare characters less or equal delete
Encoding: 1 syllable ("F3")
Otherwise see CEQD.

CLEU

Name: compare characters less or equal update
Encoding: 1 syllable ("FB")
Otherwise see CEQU.

CLSD

Name: compare characters less delete
Encoding: 1 syllable ("F0")
Otherwise see CEQD.

CLSU

Name: compare characters less update
Encoding: 1 syllable ("F8")
Otherwise see CEQU.

CNED

Name: compare characters not equal delete
Encoding: 1 syllable ("F5")
Otherwise see CEQD.

CNEU

Name: compare characters not equal update
Encoding: 1 syllable ("FD")
Otherwise see CEQU.

CUIO

Name: communicate with Universal I/O
Encoding: 2 syllables ("954C") Variant

Stack state
transformation:

----- iocb: SingleDD -----	==> Null
----------------------------------	----------

Interrupts:

Inv Arg Value: Mem[iocb].[47:16] \neg = hex"10CB"

Inv Stack Arg: iocb not present unpagged unindexed copy
SingleDD

Stack-Underflow

DBCD

Name: dynamic binary convert to decimal

Encoding: 2 syllables ("957F") Variant

Stack state transformation:

N: opnd(integer)	B: opnd(numeric)	==>	opnd(BCD)
------------------	------------------	-----	-----------

Interrupts:

Int Overflow: N not sp integer or B not dp integer

Inv Arg Value: N \neg in {0 to 24}

Inv Stack Arg: N not opnd or B not opnd
Stack-Underflow

DBFL

Name: dynamic branch false

Encoding: 1 syllable ("A8")

Stack state transformation:

branch-dest: *	opnd(Boolean)	==>	Null
----------------	---------------	-----	------

Interrupts: same as DBUN, plus:

Inv Stack Arg: opnd(Boolean) not operand

DBRS

Name: dynamic bit reset

Encoding: 1 syllable ("9F")

Stack state transformation:

db: opnd(integer)	dest: any(bit-vector)	==>	dest'
-------------------	-----------------------	-----	-------

Interrupts:

Int Overflow: db not sp integer

Inv Arg Value: db \neg in {0 to 47}

Inv Stack Arg: db not opnd
Stack-Underflow

DBST

Name: dynamic bit set
 Encoding: 1 syllable ("97")
 Otherwise see DBRS.

DBTR

Name: dynamic branch true
 Encoding: 1 syllable ("A9")
 Otherwise see DBFL.

DBUN

Name: dynamic branch unconditional
 Encoding: 1 syllable ("AA")

Stack state transformation:

branch-dest: *	==>	Null
----------------	-----	------

Interrupts:
 Int Overflow: branch-dest opnd not sp integer
 Inv Arg Value: PCW.ll \neg = LL
 or branch-dest opnd \neg in {0 to 2**14-1}
 (optionally reportable as Invalid Index) or
 PCW.sdll \neg = SDLL
 Inv Index: branch-dest opnd.dyn__pwi
 \neg in {0 to CSD.seg__length-1}
 Inv Object: NIRW = \rightarrow PCW
 Inv Reference: NIRW
 Inv Stack Arg: branch-dest not {opnd,PCW,NIRW}
 Stack-Underflow

Also see aLXCH – NIRW evaluation; aPCW – code-stream pointer distribution.

DEXI

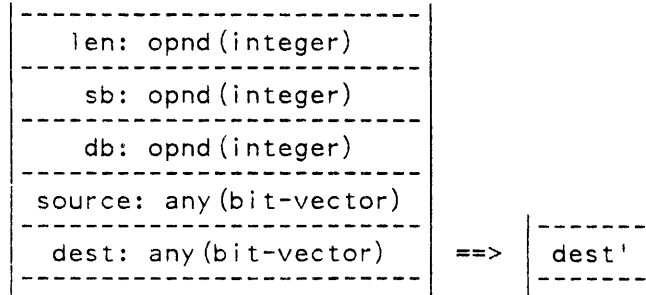
Name: disable external interrupts
 Encoding: 2 syllables ("9547") Variant
 Stack state transformation: none
 Interrupts: none

DFTR

Name: dynamic field transfer

Encoding: 1 syllable ("99")

Stack state transformation:



Interrupts:

Int Overflow: len or sb or db not sp integer

Inv Arg Value: len \neg in {0 to 48} or sb \neg in /0 to 47} or db \neg in {0 to 47}

Inv Stack Arg: len not opnd
or sb not opnd
or db not opnd

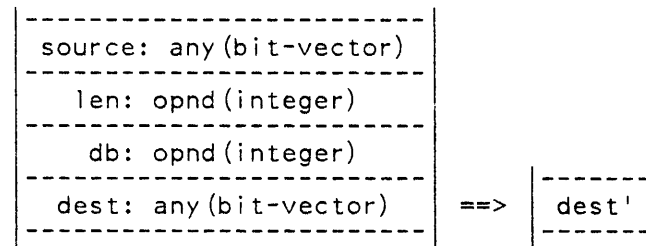
Stack-Underflow

DINS

Name: dynamic field insert

Encoding: 1 syllable ("9D")

Stack state transformation:



Interrupts:

Int Overflow: len or db not sp integer

Inv Arg Value: len \neg in {0 to 48} or db \neg in {0 to 47}

Inv Stack Arg: len not opnd or db not opnd

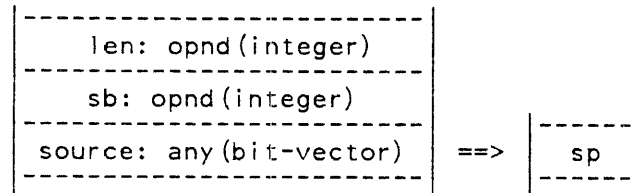
Stack-Underflow

DISO

Name: dynamic field isolate

Encoding: 1 syllable ("9B")

Stack state transformation:



Interrupts:

Int Overflow: len or sb not sp integer

Inv Arg Value: len \neg in {0 to 48} or sb \neg in {0 to 47}

Inv Stack Arg: len not opnd or sb not opnd

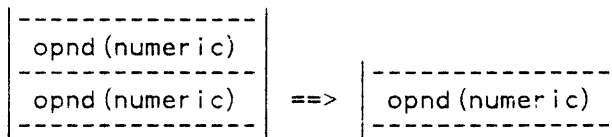
Stack-Underflow

DIVD

Name: divide

Encoding: 1 syllable ("83")

Stack state transformation:



Interrupts:

Divide by Zero: TOS opnd = 0

Exp Overflow: R(x/y) = exponent value too big

Exp Underflow: R(x/y) = exponent value too small

Inv Stack Arg: TOS not opnd or TOS2 not opnd

Precision Loss: R(x/y) \neg = R*(x/y)

Stack-Underflow

DLAY

Name: delay

Encoding: 3 syllables ("95F6", N:8) Variant

Stack state transformation: none

Interrupts: none

DLET

Name: delete top-of-stack

Encoding: 1 syllable ("B5")

Stack state transformation: $\left| \begin{array}{c} \text{---} \\ \text{any} \\ \text{---} \end{array} \right| \Rightarrow \text{Null}$

Interrupts:

Stack-Underflow

DRNT

Name: dynamic range test

Encoding: 2 syllables ("9583") Variant

Stack state transformation: $\left| \begin{array}{c} \text{---} \\ \text{H: opnd (numeric)} \\ \text{---} \\ \text{L: opnd (numeric)} \\ \text{---} \\ \text{X: opnd (numeric)} \\ \text{---} \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{---} \\ \text{opnd (Boolean)} \\ \text{---} \\ \text{X} \\ \text{---} \end{array} \right|$

Interrupts:

Inv Stack Arg: H not operand or L not operand or X not operand

Stack-Underflow

DSL F

Name: dynamic scale left

Encoding: 1 syllable ("C1")

Stack state transformation: $\left| \begin{array}{c} \text{---} \\ \text{sf: opnd (integer)} \\ \text{---} \\ \text{opnd (numeric)} \\ \text{---} \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{---} \\ \text{opnd (integer)} \\ \text{---} \end{array} \right|$

Interrupts:

Int Overflow: sf not sp integer or TOS2 not dp integer

Inv Arg Value: sf not in {0 to 12}

Inv Stack Arg: sf not opnd or TOS2 not opnd

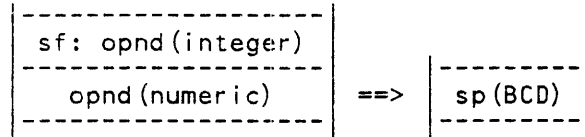
Stack-Underflow

DSRF

Name: dynamic scale right final

Encoding: 1 syllable ("C7")

Stack state transformation:



Interrupts:

Int Overflow: sf not sp integer or TOS2 not dp integer

Inv Arg Value: sf \neg in {0 to 12}

Inv Stack Arg: sf not opnd or TOS2 not opnd

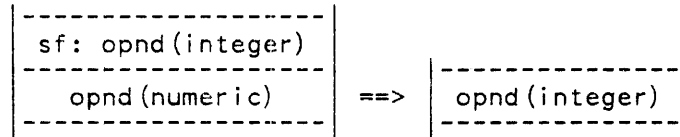
Stack-Underflow

DSRR

Name: dynamic scale right rounded

Encoding: 1 syllable ("C9")

Stack state transformation:



Interrupts:

Int Overflow: sf not sp integer or TOS2 not dp integer

Inv Arg Value: sf \neg in {0 to 12}

Inv Stack Arg: sf not opnd or TOS2 not opnd

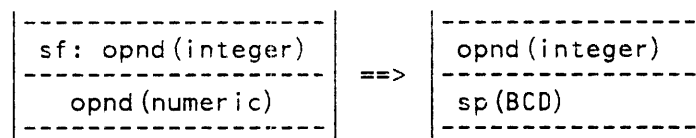
Stack-Underflow

DSRS

Name: dynamic scale right save

Encoding: 1 syllable ("C5")

Stack state transformation:



Interrupts:

Int Overflow: sf not sp integer or TOS2 not dp integer

Inv Arg Value: sf \neg in {0 to 12}

Inv Stack Arg: sf not opnd or TOS2 not opnd

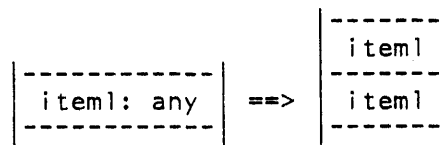
Stack-Underflow

DSRT

Name: dynamic scale right truncate
Encoding: 1 syllable ("C3")
Otherwise see DSRR.

DUPL

Name: duplicate top-of-stack
Encoding: 1 syllable ("B7")
Stack state transformation:



Interrupts:
Stack-Overflow
Stack-Underflow

EEXI

Name: enable external interrupts
Encoding: 2 syllables ("9546") Variant
Stack state transformation: none
Interrupts: none

ENDE

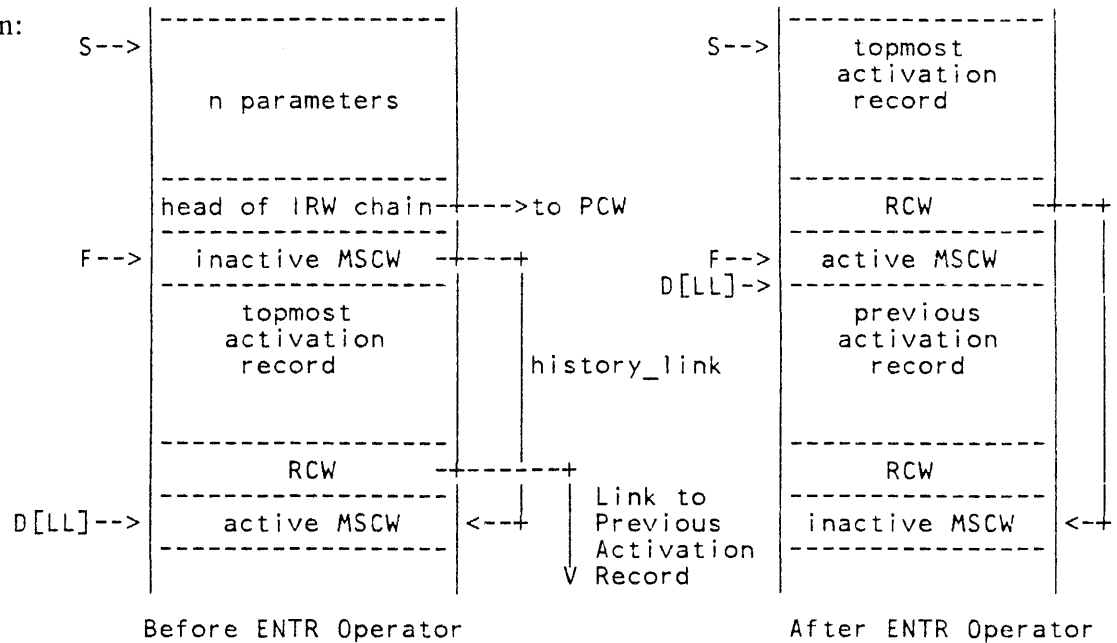
Name: end edit
Encoding: 1 syllable ("DE") Edit
Stack state transformation: none, except to establish final stack state for TEEU
Interrupts:
Inv Index: source' or dest' word index \neg in $\{0 \text{ to } 2^{**}16 - 1\}$

ENDF

Name:	end float
Encoding:	3 syllables ("D5", MinusChar:8, PlusChar:8) Edit
Stack state transformation:	none
Interrupts:	
Inv Index:	dest' word index \neg in $\{0 \text{ to } 2^{**}16-1\}$
Memory Protect:	read_only dest pointer
Paged Array:	dest pointer
Stack-Overflow:	If table-edit, update for Paged Array interrupt

ENTR

Name: enter
Encoding: 1 syllable ("AB")
Stack state transformation:



Interrupts:
Binding Request: IRW chain \rightarrow DD with element_size = 7
Inv Arg Value: Mem[F+1] = NIRW directly to PCW and PCW \div and PCW.ll - 1 \neg = NIRW.lambda or PCW.ll \neg in {0, MSCW.ll+1} or PCW.invalid_ll \neg = 0
Inv Reference: NIRW
Inv Ref Chain: IRW chain = \rightarrow (PCW, or DD with element_size = 7)
Inv Stack Arg: Mem[F+1] not IRW
Stack Structure: $S \leq F$ or Mem[F] \neg = inactive MSCW or SIRW.lexical_link $\neg \geq$ entered MSCW or new displacement \neg in {1 to 2**16-1}

Also see aLXCH - NIRW evaluation; display update; aLXLK - SIRW evaluation; aPRCW - code-stream pointer distribution.

EQUL

Name: equal to
 Encoding: 1 syllable ("8C")
 Stack state transformation:

opnd (numeric)	==>	sp (Boolean)
opnd (numeric)		

 Interrupts:
 Inv Stack Arg: TOS not opnd or TOS2 not opnd
 Stack-Underflow

EVAL

Name: evaluate
 Encoding: 1 syllable ("AC")
 Stack state transformation:

ref: *	==>	{IRW, IndexedWordDD}

 Interrupts:
 Inv Reference: NIRW
 Inv Ref Chain: see functional definition in Section 3
 Inv Stack Arg: ref not {IRW,IndexedWordDD}
 Stack-Underflow

Also see aACCE – IRW → PCW; aLXCH – NIRW evaluation; aLXLK – SIRW evaluation.

EXCH

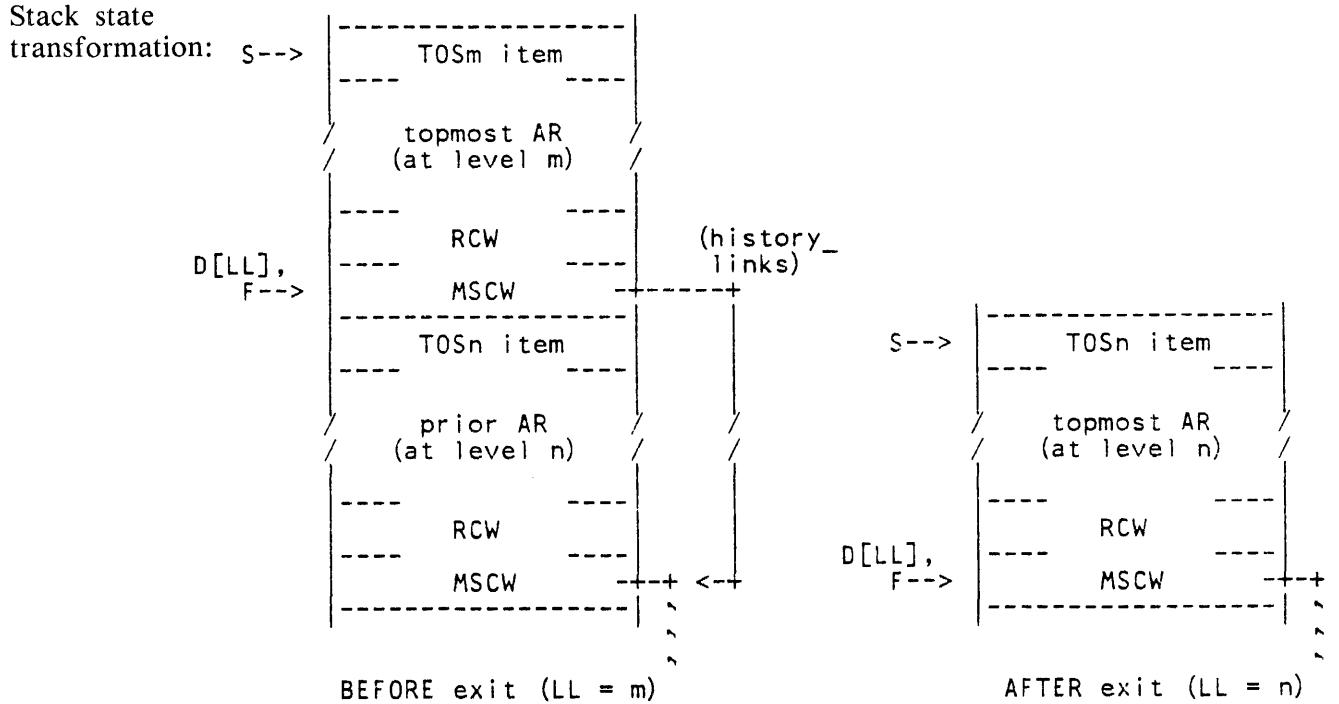
Name: exchange top-of-stack
 Encoding: 1 syllable ("B6")
 Stack state transformation:

item1: any	==>	item2
item2: any		item1

 Interrupts:
 Stack-Underflow

EXIT

Name: exit
Encoding: 1 syllable ("A3")



Interrupts:

Block Exit: RCW.block__exit = 1

Stack Structure:

- Mem[D[LL]+1] \rightarrow RCW
- or Mem[D[LL]] \rightarrow entered MSCW
- or MSCW.history__link = 0
- or HistLink \leq BOSR
- or Stack[HistLink] \rightarrow MSCW
- or RCW.ll \rightarrow MSCW.ll

(First entered MSCW on historical chain)

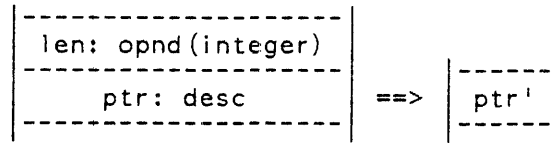
Also see aLXCH – display update; aPRCW – code-stream pointer distribution.

EXPU

Name: execute single edit operator, single pointer update

Encoding: 1 syllable ("DD")

Stack state transformation:



Interrupts:

Int Overflow: len not sp integer

Inv Arg Value: dest Pointer.char__index out of range
or len > 2**20-1

Inv Object: Mem[AbsentCopyDD.address] not original DD

Inv Stack Arg: len not opnd or ptr not IndexedDD

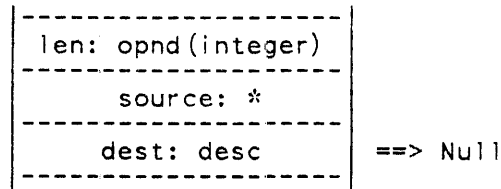
Presence Bit: ptr
Stack-Underflow

EXSD

Name: execute single edit operator delete

Encoding: 1 syllable ("D2")

Stack state transformation:



Interrupts:

Int Overflow: len not sp integer

Inv Arg Value: source/dest Pointer.char__index out of range
or len > 2**20-1

Inv Object: Mem[AbsentCopyDD.address] not original DD

Inv Stack Arg: len not opnd or source not
{IndexedDD,opnd}
or dest not IndexedDD

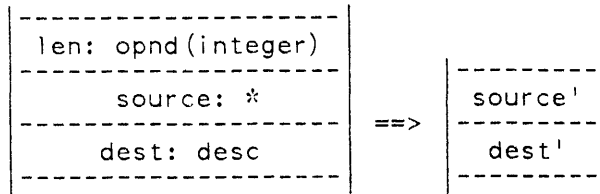
Presence Bit: source or dest pointer
Stack-Underflow

EXSU

Name: execute single edit operator update

Encoding: 1 syllable ("DA")

Stack state transformation:



NOTE

Note: final stack state produced at completion of subsequent edit micro-operator.

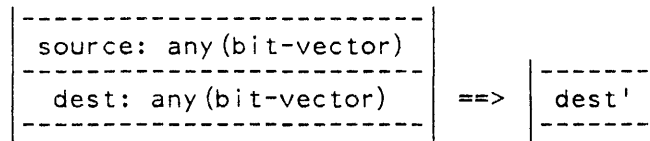
Interrupts: same as EXSD.

FLTR

Name: field transfer

Encoding: 4 syllables ("98", Db:8, Sb:8, Len:8)

Stack state transformation:



Interrupts:

Inv Code Param: Db > 47 or Sb > 47 or Len > 48
Stack-Underflow

GREQ

Name: greater than or equal to

Encoding: 1 syllable ("89")
Otherwise see EQU.

GRTR

Name: greater than

Encoding: 1 syllable ("8A")
Otherwise see EQU.

HALT

Name: conditional processor halt
 Encoding: 2 syllables ("95DF") Variant
 1 syllable ("DF") Edit
 Stack state transformation: none
 Interrupts: none

ICLD

Name: input convert left-signed delete
 Encoding: 2 syllables ("9575") Variant
 Otherwise see ICUD.

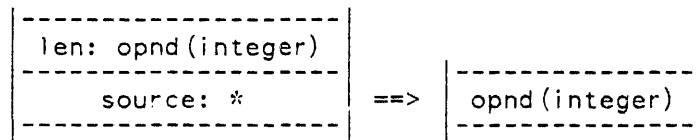
ICRD

Name: input convert right-signed delete
 Encoding: 2 syllables ("9576") Variant
 Otherwise see ICUD.

ICUD

Name: input convert unsigned delete
 Encoding: 1 syllable ("A4")

Stack state transformation:



Result is sp if $ABS(integer) < 8^{**}13$, dp otherwise.

Interrupts:
 Int Overflow: len not sp integer
 Inv Arg Value: len > 23
 or source Pointer.char__index out of range
 Inv Index: source' word index \neg in {0 to $2^{**}16-1$ }
 Inv Object: Mem[AbsentCopyDD.address] not original DD
 Inv Stack Arg: len not opnd or source not {IndexedDD, opnd}
 Paged Array: source pointer
 Presence Bit: source pointer
 Stack-Overflow: update for Paged Array interrupt
 Stack-Underflow

ICVD

Name: input convert delete
Encoding: 1 syllable ("CA")
Otherwise see ICUD.

ICVU

Name: input convert update
Encoding: 1 syllable ("CB")

Stack state transformation:

len: opnd(integer)	==>	source'
source: *		opnd(integer)

NOTE

Result is sp if $ABS(integer) < 8**13$, dp otherwise.

Interrupts: same as ICUD.

IDIV

Name: integer divide
Encoding: 1 syllable ("84")

Stack state transformation:

opnd(numeric)	==>	opnd(integer)
opnd(numeric)		

Interrupts:

Divide by Zero: TOS opnd = 0
Int Overflow: result not sp or dp integer
(Result type depends on argument types.)
Inv Stack Arg: TOS not opnd or TOS2 not opnd
Stack-Underflow:

IDLE

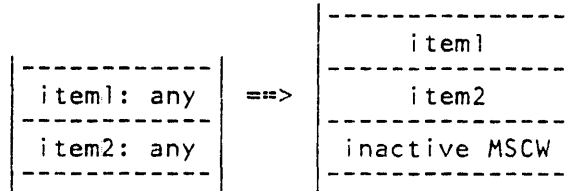
Name: idle until interrupt
Encoding: 2 syllables ("9544") Variant
Stack state transformation: none
Interrupts: An external interrupt occurs at the end of the operator.

IMKS

Name: insert mark stack

Encoding: 1 syllable ("CF")

Stack state
transformation:



Interrupts: same as MKST, plus:
Stack-Underflow

INDX

Name:	index													
Encoding:	1 syllable ("A6")													
Stack state transformation:	<table border="0" style="margin-left: 20px;"> <tr> <td style="border: 1px dashed black; padding: 5px;">desc-ind: *</td> <td rowspan="2" style="vertical-align: middle; padding: 0 10px;">==></td> <td style="border: 1px dashed black; padding: 5px;">IndexedDD</td> </tr> <tr> <td style="border: 1px dashed black; padding: 5px;">index: opnd(integer)</td> <td style="border: 1px dashed black; padding: 5px;">IndexedDD</td> </tr> <tr> <td colspan="3" style="text-align: center; padding: 5px 0;">OR</td> </tr> <tr> <td style="border: 1px dashed black; padding: 5px;">index: opnd(integer)</td> <td rowspan="2" style="vertical-align: middle; padding: 0 10px;">==></td> <td style="border: 1px dashed black; padding: 5px;">IndexedDD</td> </tr> <tr> <td style="border: 1px dashed black; padding: 5px;">desc-ind: *</td> <td style="border: 1px dashed black; padding: 5px;">IndexedDD</td> </tr> </table>	desc-ind: *	==>	IndexedDD	index: opnd(integer)	IndexedDD	OR			index: opnd(integer)	==>	IndexedDD	desc-ind: *	IndexedDD
desc-ind: *	==>	IndexedDD												
index: opnd(integer)		IndexedDD												
OR														
index: opnd(integer)	==>	IndexedDD												
desc-ind: *		IndexedDD												
Interrupts:														
Binding Request:	IRW chain → DD with element_size = 7 or desc-ind is copy DD with element_size = 7													
Int Overflow:	index not sp integer													
Inv Index:	index \neg in {0 to DD.length-1} or (unpaged CharDD and word index \neg in {0 to 2**16-1}) or (unpaged DoubleDD and (doubled) word index \neg in {0 to 2**20-1})													
Inv Object:	Mem[AbsentCopyDD.address] not original DD													
Inv Reference:	NIRW													
Inv Ref Chain:	IRW chain = → (unindexed WordDD, unindexed CharDD, or DD with element_size = 7)													
Inv Stack Arg:	desc-ind not {unindexed copy WordDD, unindexed copy CharDD,IRW} or index not opnd													
Presence Bit:	page table													
Page Struct Err:	paged DD [page index] = → unpaged original SingleDD													
Stack-Underflow														

Also see aLXCH – NIRW evaluation; aLXLK – SIRW evaluation.

INOP

Name: insert overpunch
Encoding: 1 syllable ("D8") Edit
Stack state transformation: none
Interrupts:
Inv Index: dest' word index \neg in {0 to 2**16-1}
Inv Stack Arg: dest.element_size = hex
Memory Protect: read_only dest pointer
Paged Array: dest pointer
Stack-Overflow: If table-edit, update for Paged Array interrupt.

INSC

Name: insert conditional
Encoding: 3 syllables ("DD", ZeroChar:8, NonZeroChar:8) Edit
4 syllables ("DD", Length:8, ZeroChar:8, NonZeroChar:8) Table edit
Stack state transformation: none
Interrupts:
Inv Index: dest' word index \neg in {0 to 2**16-1}
Memory Protect: read_only dest pointer
Paged Array: dest pointer
Stack-Overflow: If table-edit, update for Paged Array interrupt.

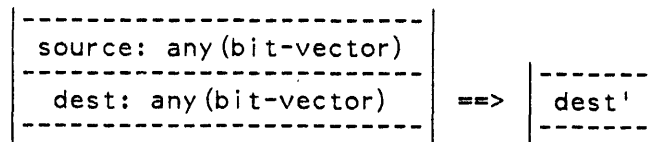
INSG

Name: insert display sign
 Encoding: 3 syllables ("D9", MinusChar:8, PlusChar:8) Edit
 Stack state transformation: none
 Interrupts:
 Inv Index: dest' word index \neg in {0 to 2**16-1}
 Inv Stack Arg: dest.element_size = hex
 Memory Protect: read_only dest pointer
 Paged Array: dest pointer
 Stack-Overflow: If table-edit, update for Paged Array interrupt.

INSR

Name: field insert
 Encoding: 3 syllables ("9C", Db:8, Len:8)

Stack state transformation:



Interrupts:
 Inv Code Param: Db > 47 or Len > 48
 Stack-Underflow

INSU

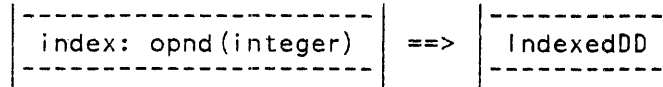
Name: insert unconditional
 Encoding: 2 syllables ("DC", Char:8) Edit
 3 syllables ("DC", Length:8, Char:8) Table Edit
 Otherwise see INSC.

INXA

Name: index, by means of address-couple parameter

Encoding: 3 syllables ("E7", lambda:4, delta:12)

Stack state transformation:



Interrupts:

Binding Request: IRW chain \rightarrow DD with element_size = 7

Int Overflow: index not sp integer

Inv Index: index \neg in {0 to DD.length-1} or
(unpaged CharDD and word index \neg in {0 to 2**16-1})
or (unpaged DoubleDD and (doubled) word index \neg in {0 to 2**20-1})

Inv Object: Mem[AbsentCopyDD.address] not original DD

Inv Reference: address-couple parameter

Inv Ref Chain: IRW chain = \rightarrow (unindexed WordDD,
unindexed CharDD,
or DD with element_size = 7)

Inv Stack Arg: index not opnd

Presence Bit: page table

Page Struct Err: paged DD [page index] = \rightarrow unpaged original SingleDD

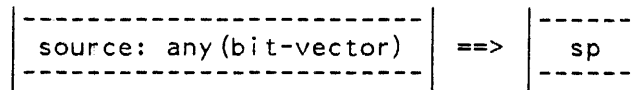
Stack-Underflow

ISOL

Name: field isolate

Encoding: 3 syllables ("9A", Sb:8, Len:8)

Stack state transformation:



Interrupts:

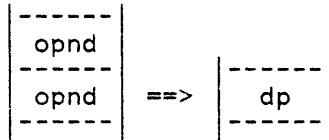
Inv Code Param: Sb > 47 or Len > 48

Stack-Underflow

JOIN

Name: set two singles to double
Encoding: 2 syllables ("9542") Variant

Stack state transformation:

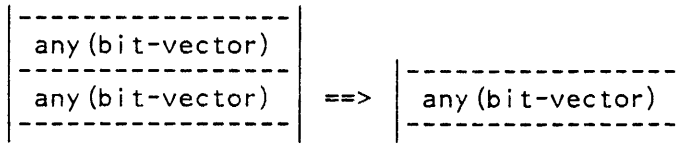


Interrupts:
Inv Stack Arg: TOS not opnd or TOS2 not opnd
Stack-Underflow

LAND

Name: logical and
Encoding: 1 syllable ("90")

Stack state transformation:



Interrupts:
Stack-Underflow

LEQV

Name: logical equivalence
Encoding: 1 syllable ("93")
Otherwise see LAND.

LESS

Name: less than
Encoding: 1 syllable ("88")
Otherwise see EQU.

LKID

Name: read interlock status
 Encoding: 2 syllables ("95B3") Variant
 Stack state transformation: $\left| \begin{array}{c} \text{ref: *} \\ \text{---} \\ \text{---} \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{sp (12-bit integer)} \\ \text{---} \\ \text{---} \end{array} \right|$
 Interrupts:
 Inv Object: ref = \rightarrow word with tag in {0,3}
 or Mem[AbsentCopyDD.address] not original DD
 Inv Reference: NIRW
 Inv Stack Arg: ref not {IRW, IndexedSingleDD}
 Presence Bit: IndexedSingleDD
 Stack-Underflow

Also see aLXCH – NIRW evaluation; aLXLK – SIRW evaluation.

LLLU

Name: linked list lookup
 Encoding: 2 syllables ("95BD") Variant
 Stack state transformation: $\left| \begin{array}{c} \text{index: opnd(integer)} \\ \text{---} \\ \text{list: SingleDD} \\ \text{---} \\ \text{targ: opnd(integer)} \\ \text{---} \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{sp(integer)} \\ \text{---} \\ \text{---} \end{array} \right|$
 Interrupts:
 Int Overflow: index or targ not sp integer
 Inv Index: any index value \neg in {0 to DD.length-1}
 Inv Object: Mem[AbsentCopyDD.address] not original DD
 Inv Stack Arg: index not opnd or targ not opnd or list not unpagged unindexed SingleDD
 Presence Bit: SingleDD
 Stack-Underflow

LNMC

Name: Long name call
 Encoding: 4 syllables ("958C", lambda:4, delta:12)
 Variant
 Stack state transformation: Null ==> $\left| \begin{array}{c} \text{---} \\ \text{NIRW} \\ \text{---} \end{array} \right|$
 Interrupts:
 Inv Reference: address-couple parameter
 Stack-Overflow

LNOT

Name: logical not
 Encoding: 1 syllable ("92")
 Stack state transformation: $\left| \begin{array}{c} \text{---} \\ \text{item: any (bit-vector)} \\ \text{---} \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{---} \\ \text{item'} \\ \text{---} \end{array} \right|$
 Interrupts:
 Stack-Underflow

LOAD

Name: load
 Encoding: 1 syllable ("BD")
 Stack state transformation: $\left| \begin{array}{c} \text{---} \\ \text{ref: *} \\ \text{---} \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{---} \\ \text{target} \\ \text{---} \end{array} \right|$

NOTE

Target = {opnd, tag 4 word, unit opnd, SIRW, desc}

Interrupts:
 Inv Object: reference = $\rightarrow >$ {SIRW, DD, even-tag word}
 or IndexedDoubleDD = \rightarrow operand
 or Mem[AbsentCopyDD.address] not original DD
 Inv Reference: NIRW
 Inv Stack Arg: ref not {IRW, IndexedWordDD}
 Presence Bit: IndexedWordDD
 Stack-Underflow

Also see aLXCH – NIRW evaluation; aLXLK – SIRW evaluation.

LODT

Name: load transparent
 Encoding: 1 syllable ("BC")
 2 syllables ("95BC") Variant

Stack state transformation: $\left[\begin{array}{c} \text{ref: *} \end{array} \right] \Rightarrow \left[\begin{array}{c} \text{any} \end{array} \right]$

Interrupts:
 Int Overflow: ref opnd not a sp integer (*)
 Inv Address: ref integer \neg in $\{0 \text{ to } 2^{**}20-1\}$ (*)
 Inv Arg Value: ref opnd \neg in $\{0 \text{ to } 2^{**}20-1\}$ (*)
 Inv Object: Mem[AbsentCopyDD.address] not original DD
 Inv Reference: NIRW
 Inv Stack Arg: ref not {IRW,IndexedSingleDD,20-bit integer}
 Presence Bit: IndexedSingleDD
 Stack-Underflow

Also see aLXCH – NIRW evaluation; aLXLK – SIRW evaluation.

NOTE

(*) Int Overflow, Inv Address and Inv Arg Value are alternatives for some cases of Invalid Stack Argument; see operator definition in section 3.

LOG2

Name: leading one test
 Encoding: 2 syllables ("958B") Variant

Stack state transformation: $\left[\begin{array}{c} \text{any (bit-vector)} \end{array} \right] \Rightarrow \left[\begin{array}{c} \text{sp (integer)} \end{array} \right]$

Interrupts:
 Stack-Underflow

LOK

Name: lock interlock
 Encoding: 2 syllables ("95B0") Variant

Stack state transformation: $\left[\begin{array}{c} \text{ref: *} \end{array} \right] \Rightarrow \text{Null}$

Interrupts: same as LOKC, plus:
 Locking: interlock status not Free

LOKC

Name: conditional lock interlock
 Encoding: 2 syllables ("95B1") Variant
 Stack state transformation: $\left| \begin{array}{c} \text{---} \\ \text{ref: *} \\ \text{---} \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{---} \\ \text{sp (Boolean)} \\ \text{---} \end{array} \right|$
 Interrupts: same as LKID, plus:
 Memory Protect: read_only IndexedSingleDD

LOR

Name: logical or
 Encoding: 1 syllable ("91")
 Otherwise see LAND.

LSEQ

Name: less than or equal to
 Encoding: 1 syllable ("8B")
 Otherwise see EQUL.

LT8

Name: insert 8-bit literal
 Encoding: 2 syllables ("B2", constant:8)
 Stack state transformation: Null $\Rightarrow \left| \begin{array}{c} \text{---} \\ \text{sp} \\ \text{---} \end{array} \right|$
 Interrupts: Stack-Overflow

LT16

Name: insert 16-bit literal
 Encoding: 3 syllables ("B3", constant:16)
 Otherwise see LT8.

LT48

Name: insert 48-bit literal
Encoding: 7 to 12 syllables ("BE", ..., constant:48)

NOTE

Constant starts on word boundary; otherwise see LT8.

LVLC

Name: Long value call
Encoding: 4 syllables ("958D", lambda:4, delta:12)
Variant
Otherwise see VALC.

MCHR

Name: move characters
Encoding: 1 syllable ("D7") Edit
2 syllables ("D7", Length:8) Table Edit
Stack state transformation: none
Interrupts:
Inv Index: source' or dest' word index \neg in {0 to $2^{**}16 - 1$ }
Memory Protect: read_only dest pointer
Paged Array: source or dest pointer
Stack-Overflow: If table-edit, update for Paged Array interrupt.
Undefined Op: move operator follows EXPU

MFLT

Name: move with float
Encoding: 4 syllables ("D1", ZeroChar:8, MinusChar:8, PlusChar:8) Edit
5 syllables ("D1", Length:8, ZeroChar:8, MinusChar:8, PlusChar:8) Table Edit
Otherwise see MCHR.

MINS

Name: move with insert
 Encoding: 2 syllables ("D0", ZeroChar:8) Edit
 3 syllables ("D0", Length:8, ZeroChar:8)
 Table Edit
 Otherwise see MCHR.

MKSN

Name: mark-stack bound to name-call
 Encoding: 1 syllable ("DF")
 Stack state transformation: same as MKST (may be compiled with subsequent NAMC)
 Interrupts: same as MKST.

If an optimization of the MKSN-NAMC pair is implemented, the following interrupts can also be generated:

Inv Reference: address-couple from NAMC
 Undefined Op: next operator not NAMC

Such an implementation may also be defined to generate the following interrupts in anticipation of the forthcoming ENTR:

Binding Request: IRW chain \rightarrow DD with element_size = 7
 Inv Ref Chain: IRW chain $\neg \rightarrow$ (PCW, or DD with element_size = 7)

Also see aLXCH – NIRW evaluation; aLXLK – SIRW evaluation.

MKST

Name: mark stack
 Encoding: 1 syllable ("AE")
 Stack state transformation: Null \Rightarrow $\left| \begin{array}{c} \text{-----} \\ \text{inactive MSCW} \\ \text{-----} \end{array} \right|$
 Interrupts:
 Stack-Overflow
 Stack Structure: (new F) – (old F) \neg in {1 to 2**14–1}
 or (new F) – BOSR \neg in {0 to 2**16–1}

MPCW

Name: make PCW
 Encoding: 7 to 12 syllables ("BF", ..., SkeletonPCW:48)

NOTE

SkeletonPCW starts on word boundary

Stack state transformation: Null ==> $\left| \begin{array}{c} \text{-----} \\ \text{PCW} \\ \text{-----} \end{array} \right|$

Interrupts:
Stack-Overflow

MULT

Name: multiply
 Encoding: 1 syllable ("82")

Stack state transformation: $\left| \begin{array}{c} \text{-----} \\ \text{opnd (numeric)} \\ \text{-----} \\ \text{-----} \\ \text{opnd (numeric)} \\ \text{-----} \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{-----} \\ \text{opnd (numeric)} \\ \text{-----} \end{array} \right|$

Interrupts:
 Exp Overflow: $R(x*y) = \text{exponent value too big}$
 Exp Underflow: $R(x*y) = \text{exponent value too small}$
 Precision Loss: $R(x*y) \neg = R*(x*y)$
 Inv Stack Arg: TOS not opnd or TOS2 not opnd
 Stack-Underflow

MULX

Name: extended multiply
 Encoding: 1 syllable

Stack state transformation: $\left| \begin{array}{c} \text{-----} \\ \text{opnd (numeric)} \\ \text{-----} \\ \text{-----} \\ \text{opnd (numeric)} \\ \text{-----} \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{-----} \\ \text{dp (numeric)} \\ \text{-----} \end{array} \right|$

Interrupts: same as MULT

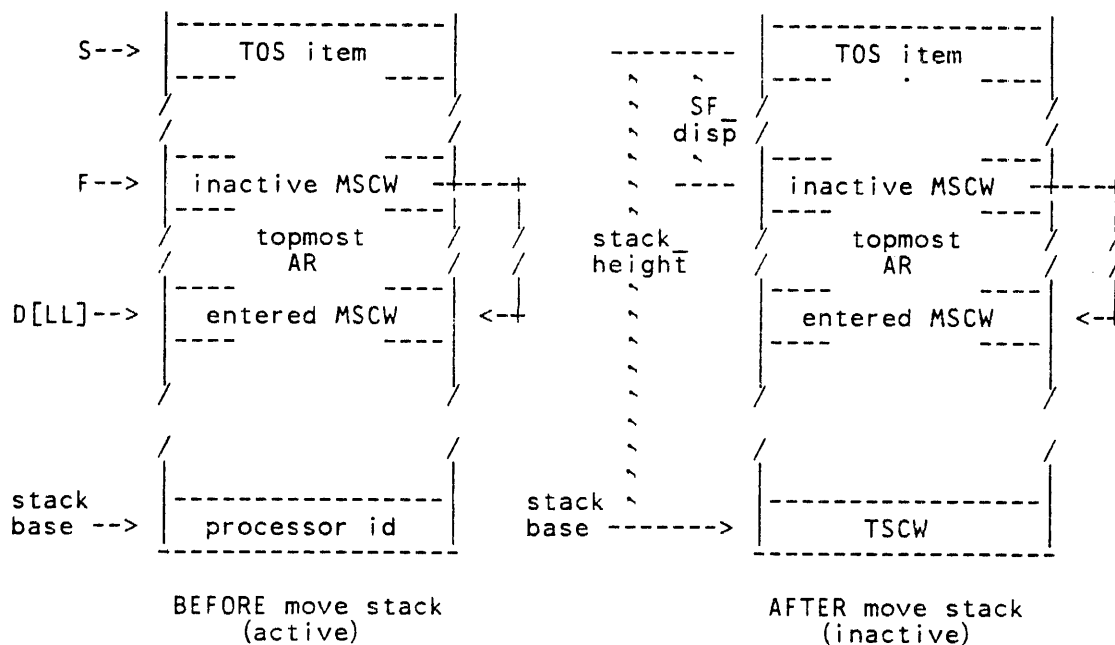
MVNU

Name: move numeric unconditional
Encoding: 1 syllable ("D6") Edit
2 syllables ("D6", Length:8) Table Edit
Otherwise see MCHR.

MVST

Name: move to stack
Encoding: 2 syllables ("95AF") Variant

Stack state transformation



Interrupts:

Inv Arg Value: ENR value too large for container
Inv Index: SNR value \neg in $\{0 \text{ to } \text{SVD.length}-1\}$
Inv Object: Mem[AbsentCopyDD.address] not original DD
or stack-vector descriptor not unpagged original SingleDD
or stack descriptor not unpagged unindexed SingleDD

Inv Stack Arg: TOS not single-precision operand

Presence Bit: destination stack descriptor

Stack Structure: Stack [stack base] \neg = TSCW
or computed $F \leftarrow \text{BOSR}$
or S-BOSR \neg in $\{1 \text{ to } 2^{**16}-1\}$
or S-F: \neg in $\{1 \text{ to } 2^{**14}-1\}$
or Stack [HistLink] \neg = MSCW
or LL \neg = MSCW.ll
(First entered MSCW on historical chain.)

Stack-Underflow: (option if ENR container-size is 0)

Also see aISX – argument not 12-bit integer (option if ENR container-size = 0); aLXCH – display update.

NAMC

Name: name call
 Encoding: 2 syllables (binary 01:2, AddressCouple:14)
 Stack state transformation: Null ==> $\left| \begin{array}{c} \text{---} \\ \text{NIRW} \\ \text{---} \end{array} \right|$
 Interrupts:
 Inv Reference: address-couple parameter
 Stack-Overflow

NEQL

Name: not equal to
 Encoding: 1 syllable ("8D")
 Otherwise see EQU.

NOOP

Name: no operation
 Encoding: 1 syllable ("FE")
 2 syllables ("95FE") Variant
 Stack state transformation: none

NORM

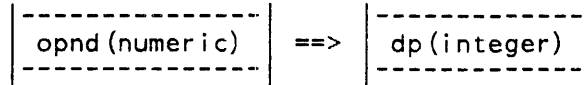
Name: normalize
 Encoding: 2 syllables ("958E") Variant
 Stack state transformation: $\left| \begin{array}{c} \text{---} \\ \text{num:opnd (numeric)} \\ \text{---} \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{---} \\ \text{num} \\ \text{---} \end{array} \right|$
 Interrupts:
 Inv Stack Arg: num not opnd
 Exp Underflow: N(x) = exponent value too small
 Stack-Underflow

NTGD

Name: integerize double-precision rounded

Encoding: 2 syllables ("9587") Variant

Stack state transformation:



Interrupts:

Int Overflow: TOS opnd not dp integer

Inv Stack Arg: TOS not opnd

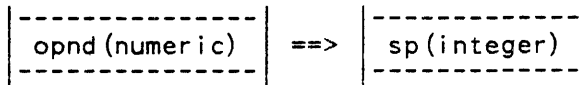
Stack-Underflow

NTGR

Name: integerize rounded

Encoding: 1 syllable ("87")

Stack state transformation:



Interrupts:

Int Overflow: TOS opnd not sp integer

Inv Stack Arg: TOS not opnd

Stack-Underflow

NTIA

Name: integerize truncated

Encoding: 1 syllable ("86")
Otherwise see NTGR.

NTTD

Name: integerize double-precision truncated

Encoding: 2 syllables ("9586") Variant
Otherwise see NTGD.

NVLD

Name: invalid operator
 Encoding: 1 syllable ("FF")
 2 syllables ("95FF") Variant
 Stack state transformation: none
 Interrupts:
 Invalid Operator

NXLN

Name: index and load name
 Encoding: 1 syllable ("A5")

Stack state transformation:

desc-ind: *	==>	unindexed copy DD
index: opnd(integer)	OR	unindexed copy DD
index: opnd(integer)	==>	unindexed copy DD
desc-ind: *		

Interrupts:
 Binding Request: IRW chain \rightarrow DD with element_size = 7
 or desc-ind is copy DD with element_size = 7
 Int Overflow: index not sp integer
 Inv Index: index \neg in {0 to DD.length-1}
 or (unpaged DoubleDD and (doubled) word
 index \neg in {0 to 2**20-1})
 Inv Object: SingleDD [index] = \rightarrow unindexed DD
 or Mem[AbsentCopyDD.address] not original DD
 Inv Reference: NIRW
 Inv Ref Chain: IRW chain = \rightarrow (unindexed SingleDD,
 or DD with element_size = 7)
 Inv Stack Arg: desc-ind not {unindexed copy SingleDD,IRW}
 or index not opnd
 Presence Bit: page table or indexed SingleDD
 Page Struct Err: pagged SingleDD [page index] = \rightarrow unpaged
 original SingleDD
 Stack-Underflow

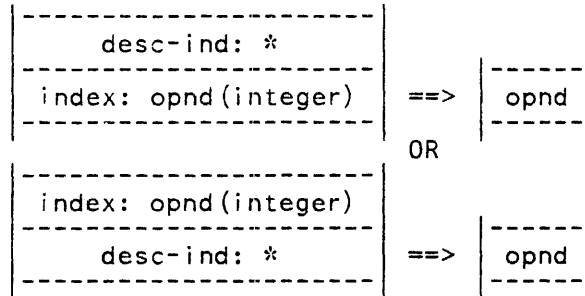
Also see aLXCH – NIRW evaluation; aLXLK – SIRW evaluation.

NXLV

Name: index and load value

Encoding: 1 syllable ("AD")

Stack state transformation:



Interrupts:

Binding Request: IRW chain → DD with element_size = 7
or desc-ind is copy DD with element_size = 7

Int Overflow: index not sp integer

Inv Index: index ∉ in {0 to DD.length-1}
or (unpaged DoubleDD and (doubled) word
index ∉ in {0 to 2**20-1})

Inv Object: WordDD [index] = → opnd
or Mem[AbsentCopyDD.address] not original
DD

Inv Reference: NIRW

Inv Ref Chain: IRW chain = → (unindexed WordDD,
or DD with element_size = 7)

Inv Stack Arg: desc-ind not {unindexed copy WordDD,IRW}
or index not opnd

Page Struct Err: pagged WordDD [page index] = → unpaged
original SingleDD

Presence Bit: Page table or indexed WordDD
Stack-Underflow

Also see aLXCH – NIRW evaluation; aLXLK – SIRW evaluation.

NXVA

Name:	index and load value by means of address-couple parameter
Encoding:	3 syllables ("EF", lambda:4, delta:12)
Stack State transformation:	$\left[\begin{array}{c} \text{index: opnd(integer)} \end{array} \right] \Rightarrow \left[\begin{array}{c} \text{opnd} \end{array} \right]$
Interrupts:	
Binding Request:	IRW chain \rightarrow DD with element_size = 7
Int Overflow:	index not sp integer
Inv Index:	index \neg in {0 to DD.length-1} or (unpaged DoubleDD and (doubled) word index \neg in {0 to 2**20-1})
Inv Object:	WordDD [index] = \rightarrow opnd or Mem[AbsentCopyDD.address] not original DD
Inv Reference:	address-couple parameter
Inv Ref Chain:	IRW chain = \rightarrow (unindexed WordDD, or DD with element_size = 7)
Inv Stack Arg:	index not opnd
Page Struct Err:	paged WordDD [page index] = \rightarrow unpaged original SingleDD
Presence Bit: Stack-Underflow	page table or indexed WordDD

Also see aLXCH – address-couple parameter evaluation; aLXLK – SIRW evaluation.

OCRX

Name:	occurs index
Encoding:	2 syllables ("9585") Variant
Stack state transformation:	$\left[\begin{array}{c} \text{sp(ICW)} \\ \text{opnd(integer)} \end{array} \right] \Rightarrow \left[\begin{array}{c} \text{sp(integer)} \end{array} \right]$
Interrupts:	
Int Overflow:	TOS2 opnd not sp integer
Inv Index:	TOS2 opnd \neg in {1 to ICW.ICW_limit}
Inv Stack Arg: Stack-Underflow	TOS not sp or TOS2 not opnd

ONE

Name: insert literal one
 Encoding: 1 syllable ("B1")
 Otherwise see LT8.

OVRD

Name: overwrite delete
 Encoding: 1 syllable ("BA")

Stack state transformation:

ref: *	==>	Null
object: any		

Interrupts:
 Inv Object: Mem[AbsentCopyDD.address] not original DD
 Inv Reference: NIRW
 Inv Stack Arg: ref not {IRW,IndexedSingleDD}
 Memory Protect: read_only IndexedSingleDD
 Presence Bit: IndexedSingleDD
 Stack-Underflow

Also see aLXCH – NIRW evaluation; aLXLK – SIRW evaluation.

OVRN

Name: overwrite non-delete
 Encoding: 1 syllable ("BB")

Stack state transformation:

ref: *	==>	object
object: any		

Interrupts: same as OVRD

PACD

Name: pack delete
 Encoding: 1 syllable ("D1")
 Otherwise see PKUD.

PACU

Name: pack update
Encoding: 1 syllable ("D9")

Stack state transformation:

len: opnd (integer)	==>	source'
source: *		opnd (hex-sequence)

NOTE
Result is sp if len in {0 to 12} and dp if len in {13 to 24}

Interrupts: same as PKUD

PAUS

Name: pause until interrupt
Encoding: 2 syllables ("9584") Variant
Stack state transformation: none
Interrupts: none

PKLD

Name: pack left-signed
Encoding: 2 syllables ("9573") Variant
Otherwise see PKUD.

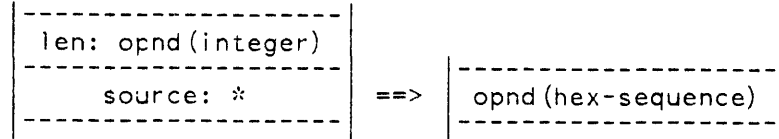
PKRD

Name: pack right-signed
Encoding: 2 syllables ("9574") Variant
Otherwise see PKUD.

PKUD

Name: pack unsigned
Encoding: 2 syllables ("9572") Variant

Stack state transformation:



NOTE

Result is sp if len in {0 to 12} and dp if len in {13 to 24}.

Interrupts:
 Int Overflow: len not sp integer
 Inv Arg Value: len > 24
 or source pointer.char__index out of range
 Inv Index: source' word index \neg in {0 to 2**16-1}
 Inv Object: Mem[AbsentCopyDD.address] not original DD
 Inv Stack Arg: len not opnd or source not {IndexedDD,opnd
 Paged Array: source pointer
 Presence Bit: source pointer
 Stack-Overflow: Update for Paged Array interrupt.
 Stack-Underflow

PUSH

Name: push working stack onto activation record
Encoding: 1 syllable ("B4")

Stack state transformation: Expression stack is pushed onto topmost activation record.

Interrupts:
Stack-Overflow

RDIV

Name: remainder divide
Encoding: 1 syllable ("85")
Otherwise see IDIV.

RDLK

Name: read lock
Encoding: 2 syllables ("95BA") Variant

Stack state transformation:

ref: *	==>	prior contents: any
--------	-----	---------------------

Interrupts: same as OVRD

REMC

Name: read external memory control
Encoding: 2 syllables ("9592") Variant

Stack state transformation:

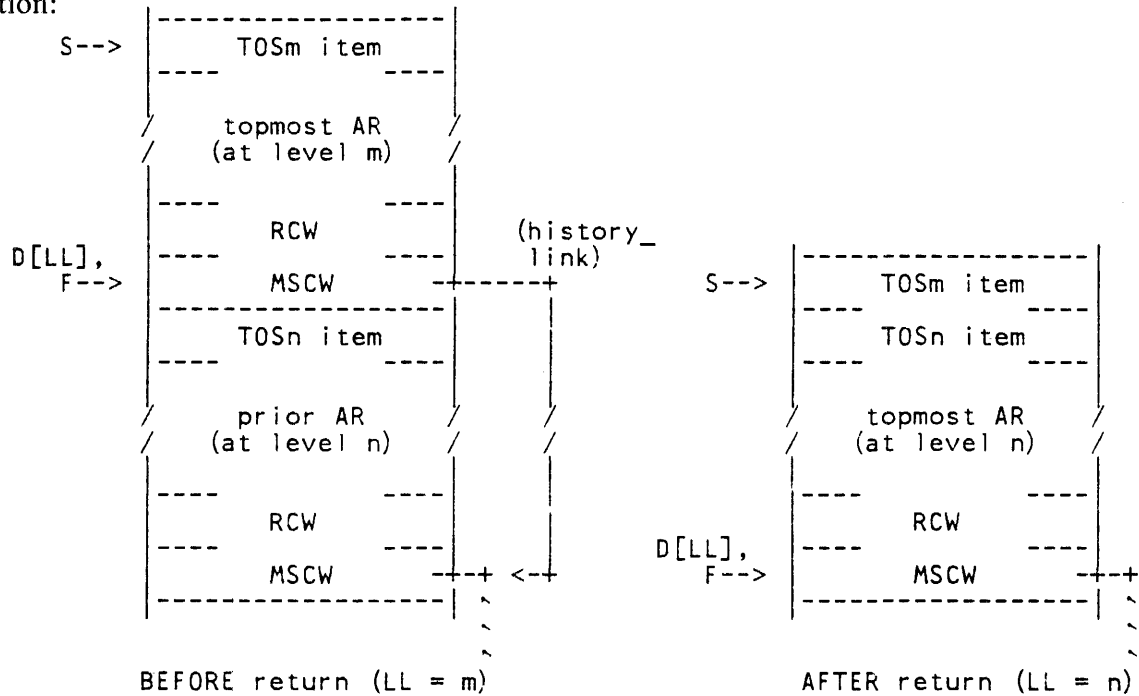
sp	==>	sp
----	-----	----

Interrupts:
Stack-Underflow
(Others are
implementation-defined.)

RETN

Name: return
Encoding: 1 syllable ("A7")

Stack state transformation:

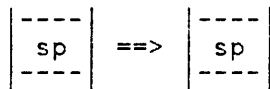


Interrupts: same as EXIT, plus:
Inv Stack Arg: TOS is NIRW
Stack-Underflow

RIPS

Name: read internal processor state
Encoding: 2 syllables ("9598") Variant

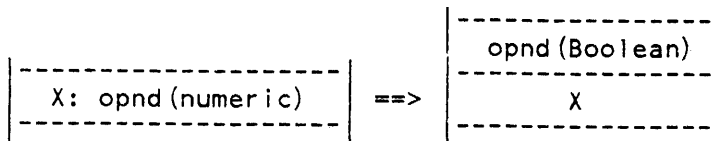
Stack state transformation:



Interrupts:
Stack-Underflow
(Others are implementation-defined.)

RNGT

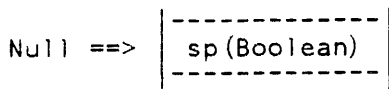
Name: range test
 Encoding: 4 syllables ("9582", l:8, h:8) Variant
 Stack state transformation:



Interrupts:
 Inv Stack Arg: TOS not operand
 Stack-Overflow
 Stack-Underflow

ROFF

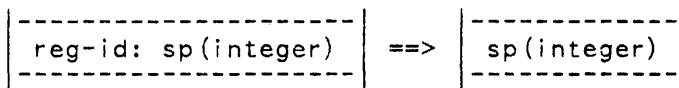
Name: read and reset overflow flip-flop
 Encoding: 1 syllable ("D7")
 Stack state transformation: Null ==>



Interrupts:
 Stack-Overflow

RPRR

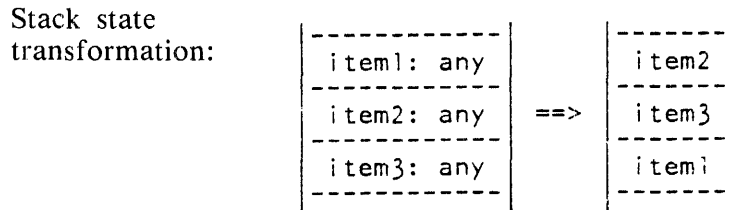
Name: read processor register
 Encoding: 2 syllables ("95B8") Variant
 Stack state transformation:



Interrupts:
 Inv Arg Value: reg-id not in {0, LL, 36-38, 52-53, 58}
 Stack-Underflow
 Also see aISX: reg-id not 6-bit integer.

RSDN

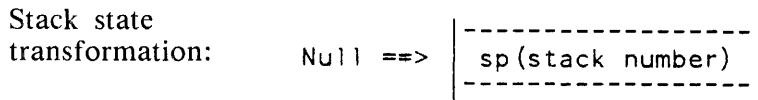
Name: rotate stack down
Encoding: 2 syllables ("95B7") Variant



Interrupts:
Stack-Underflow

RSNR

Name: read stack number
Encoding: 2 syllables ("9581") Variant



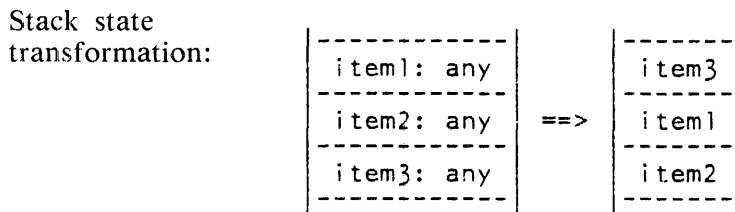
Interrupts:
Stack-Overflow

RSTF

Name: reset float flip-flop
Encoding: 1 syllable ("D4") Edit
Stack state transformation: none

RSUP

Name: rotate stack up
Encoding: 2 syllables ("95B6") Variant



Interrupts:
Stack-Underflow

RTAG

Name: read tag
Encoding: 2 syllables ("95B5") Variant
Stack state transformation: $\left| \begin{array}{c} \text{---} \\ \text{any} \\ \text{---} \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{---} \\ \text{4-bit integer} \\ \text{---} \end{array} \right|$
Interrupts:
Stack-Underflow

RTFF

Name: read true-false flip-flop
Encoding: 1 syllable ("DE")
Stack state transformation: $\text{Null} \Rightarrow \left| \begin{array}{c} \text{---} \\ \text{sp(Boolean)} \\ \text{---} \end{array} \right|$
Interrupts:
Stack-Overflow

RTOD

Name: read time of day clock
Encoding: 2 syllables ("95A7") Variant
Stack state transformation: $\text{Null} \Rightarrow \left| \begin{array}{c} \text{---} \\ \text{sp(integer)} \\ \text{---} \end{array} \right|$
Interrupts:
Stack-Overflow

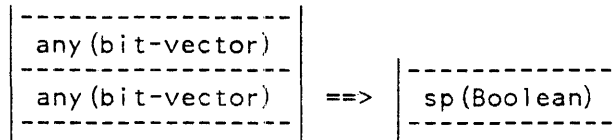
RUNI

Name: indicate running
Encoding: 2 syllables ("9541") Variant
Stack state transformation: none
Interrupts: none

SAME

Name: logical equality
Encoding: 1 syllable ("94")

Stack state transformation:

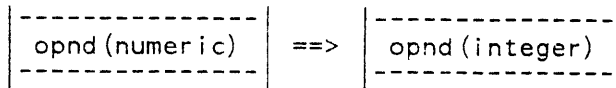


Interrupts:
Stack-Underflow

SCLF

Name: scale left
Encoding: 2 syllables ("C0", ScaleFactor:8)

Stack state transformation:



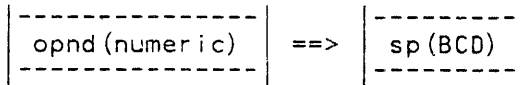
Interrupts:

Int Overflow: TOS not dp integer
Inv Code Param: ScaleFactor > 12
Inv Stack Arg: TOS not opnd
Stack-Underflow

SCRF

Name: scale right final
Encoding: 2 syllables ("C6", ScaleFactor:8)

Stack state transformation:



Interrupts:

Int Overflow: TOS not dp integer
Inv Code Param: ScaleFactor > 12
Inv Stack Arg: TOS not opnd
Stack-Underflow

SCRR

Name: scale right rounded
Encoding: 2 syllables ("C8", ScaleFactor:8)

Stack state transformation: $\left| \begin{array}{c} \text{opnd (numeric)} \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{opnd (integer)} \end{array} \right|$

Interrupts: same as SCRF

SCRS

Name: scale right save
Encoding: 2 syllables ("C4", ScaleFactor:8)

Stack state transformation: $\left| \begin{array}{c} \text{opnd (numeric)} \end{array} \right| \Rightarrow \left| \begin{array}{c} \text{opnd (integer)} \\ \text{sp (BCD)} \end{array} \right|$

Interrupts: same as SCRF, plus:
Stack-Overflow

SCRT

Name: scale right truncate
Encoding: 2 syllables ("C2", ScaleFactor:8)
Otherwise see SCRR.

SEQD

Name: scan while equal delete
 Encoding: 2 syllables ("95F4") Variant

Stack state transformation:

delim: sp(char)	==> Null
len: opnd(integer)	
source: *	

Interrupts:

Int Overflow: len not sp integer

Inv Arg Value: source Pointer.char__index out of range
 or len > 2**20-1

Inv Index: source' word index \neg in {0 to 2**16-1}

Inv Object: Mem[AbsentCopyDD.address] not original DD

Inv Stack Arg: delim not sp or len not opnd or source not {IndexedDD,opnd}

Paged Array: source pointer

Presence Bit: source pointer

Stack-Underflow

SEQU

Name: scan while equal update
 Encoding: 2 syllables ("95FC") Variant

Stack state transformation:

delim: sp(char)	==>	len'
len: opnd(integer)		source'
source: *		

Interrupts: same as SEQD

SFDC

Name: skip forward destination characters
 Encoding: 1 syllable ("DA") Edit
 2 syllables ("DA", Length:8) Table Edit

State stack transformation: none

Interrupts: same as SRDC, plus:

Inv Index: dest' word index \neg in {0 to 2**16-1}

SFSC

Name: skip forward source characters
Encoding: 1 syllable ("D2") Edit
2 syllables ("D2", Length:8) Table Edit
Stack state transformation: none
Interrupts: same as SRSC, plus:
Inv Index: source' word index \neg in {0 to $2^{16}-1$ }

SGED

Name: scan while greater or equal delete
Encoding: 2 syllables ("95F1") Variant
Otherwise see SEQD.

SGEU

Name: scan while greater or equal update
Encoding: 2 syllables ("95F9") Variant
Otherwise see SEQU.

SGTD

Name: scan while greater delete
Encoding: 2 syllables ("95F2") Variant
Otherwise see SEQD.

SGTU

Name: scan while greater update
Encoding: 2 syllables ("95FA") Variant
Otherwise see SEQU.

SHOW

Name: primitive display

Encoding: 2 syllables ("95DE") Variant

Stack state transformation:

len: opnd(integer)	==> Null
source: *	

Interrupts:

Int Overflow: len not sp integer

Inv Arg Value: source Pointer.char__index \neg in {0 to 5}
or len > 2**20-1

Inv Index: source' word index \neg in {0 to 2**16-1}

Inv Object: Mem[AbsentCopyDD.address] not original DD

Inv Stack Arg: len not opnd or source not {EBCDIC
pointer,IndexedWordDD,opnd}

Memory Protect: odd-tagged word in source

Presence Bit: source pointer

Stack-Underflow

SINT

Name: set interval timer

Encoding: 2 syllables ("9545") Variant

Stack state transformation:

time: sp(integer)	==> Null

Interrupts:

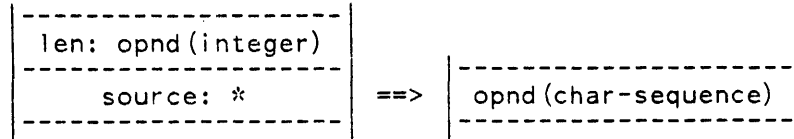
Stack-Underflow

Also see aISX: time not 11-bit integer.

SISO

Name: string isolate
Encoding: 1 syllable ("D5")

Stack state transformation:



Result type is NOTE

source	sp if len in	dp if len in
EBCDIC	{0 to 6}	{7 to 12}
hex	{0 to 12}	{13 to 24}

Interrupts:

Int Overflow: len not sp integer

Inv Arg Value: (source = EBCDIC and len > 12)
or (source = hex and len > 24)
or source pointer.char_index out of range

Inv Index: Source' word index \neg in {0 to 2**16-1}

Inv Object: Mem[AbsentCopyDD.address] not original DD

Inv Stack Arg: len not opnd or source not {IndexedDD,opnd}

Paged Array: source pointer

Presence Bit: source pointer

Stack-Overflow: Update for Paged Array interrupt.
Stack-Underflow

SLED

Name: scan while less or equal delete

Encoding: 2 syllables ("95F3") Variant
Otherwise see SEQD.

SLEU

Name: scan while less or equal update

Encoding: 2 syllables ("95FB") Variant
Otherwise see SEQU.

SLSD

Name: scan while less delete
 Encoding: 2 syllables ("95F0") Variant
 Otherwise see SEQD.

SLSU

Name: scan while less update
 Encoding: 2 syllables ("95F8") Variant
 Otherwise see SEQU.

SNED

Name: scan while not equal delete
 Encoding: 2 syllables ("95F5") Variant
 Otherwise see SEQD.

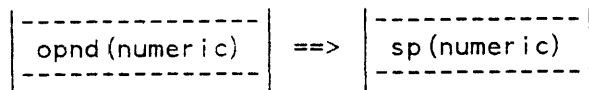
SNEU

Name: scan while not equal update
 Encoding: 2 syllables ("95FD") Variant
 Otherwise see SEQU.

SNGL

Name: set to single-precision rounded
 Encoding: 1 syllable

Stack state transformation:



Interrupts:

Exp Overflow: RS(ND(x)) = exponent value too large
 Exp Underflow: N(x) = exponent value too small
 or RS(ND(x)) = exponent value too small
 or NS(RS(ND(x))) = exponent value too

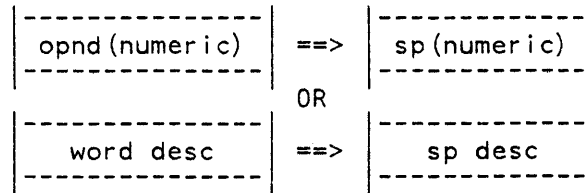
Inv Stack Arg: TOS not opnd
 Stack-Underflow

SNGT

Name: set to single-precision truncated

Encoding: 1 syllable ("CC")

Stack state transformation:



Interrupts:

Exp Overflow: TS(ND(x)) = exponent value too big

Exp Underflow: N(x) = exponent value too small
or TS(ND(x)) = exponent value too small
or NS(TS(ND(x))) = exponent value too small

Inv Arg Value: TOS = unindexed DoubleDD and length
> 2**19 - 1

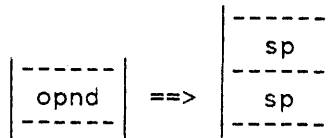
Inv Stack Arg: TOS \cap in {opnd, WordDD}
Stack-Underflow

SPLT

Name: set double to two singles

Encoding: 2 syllables ("9543") Variant

Stack state transformation:



Interrupts:

Inv Stack Arg: TOS not opnd
Stack-Overflow
Stack-Underflow

SPRR

Name: set processor register

Encoding: 2 syllables ("95B9") Variant

Stack state transformation:

reg-val: sp(integer)
reg-id: sp(integer)

 ==> Null

Interrupts:

Inv Arg Value: reg-id not in {valid values}
See SPRR definition.

Stack-Underflow

Also see aISX: reg-id not 6-bit integer or
reg-val not (register-width)-bit integer.

SRCH

Name: masked search for equal

Encoding: 2 syllables ("95BE") Variant

Stack state transformation:

domain: SingleDD
mask: any(bit-vector)
targ: any(bit-vector)

 ==>

sp(integer)

Interrupts:

Inv Object: Mem[AbsentCopyDD.address] not original DD

Inv Stack Arg: domain \neg in {unpaged copy SingleDD, IndexedSingleDD}

Presence Bit: SingleDD

Stack-Underflow

SRDC

Name: skip reverse destination characters

Encoding: 1 syllable ("DB") Edit
2 syllable ("DB", Length:8) Table Edit

Stack state transformation: none

Interrupts:

Paged Array: dest Pointer

Stack-Overflow: If table-edit, update for Paged Array interrupt.

SRSC

Name: skip reverse source characters
 Encoding: 1 syllable ("D3") Edit
 2 syllables ("D3", Length:8) Table Edit
 Stack state transformation: none
 Interrupts:
 Paged Array: source pointer
 Stack-Overflow: I table-edit, update for Paged Array interrupt.
 Undefined Op: skip-source follows EXPU

STAD

Name: store delete by means of an address-couple
 Encoding: 3 syllables ("F6", lambda:4, delta:12)
 Stack state transformation:

object: operand	==>	Null
-----------------	-----	------

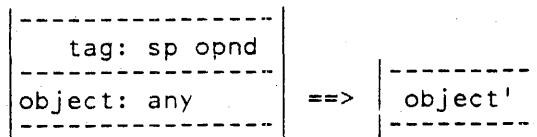
 Interrupts:
 Binding Request: IRW chain → DD with element_size = 7
 Inv Object: object operand type does not match store target
 or Mem[AbsentCopyDD.address] not original DD
 Inv Reference: address-couple parameter
 Inv Ref Chain: See functional definition in section 3.
 Inv Stack Arg: (initial state) TOS not opnd
 (restart state) TOS not {IRW, IndexedWordDD}
 or TOS2 not opnd
 Memory Protect: read_only IndexedWordDD
 or reference chain → tag-3 item
 or dp second word location is an odd-tagged item
 Presence Bit: IndexedWordDD
 Stack-Underflow

Also see aACCE – ref chain ≥ PCW; aLXCH – address-couple parameter evaluation; aLXLK – RIRW evaluation.

STAG

Name: set tag
Encoding: 2 syllables ("95B4") Variant

Stack state transformation:



Interrupts:

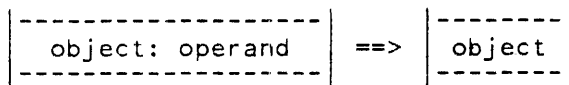
Inv Stack Arg: tag not opnd
Stack-Underflow

STAN

Name: store non-delete by means of address-couple parameter

Encoding: 3 syllables ("F7", lambda:4, delta:12)

Stack state transformation:



Interrupts: same as STAD

STFF

Name: stuff
Encoding: 1 syllable ("AF")

Stack state transformation:



Interrupts:

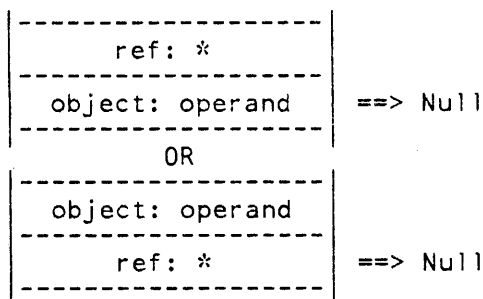
Inv Reference: NIRW
Inv Stack Arg: TOS not IRW
Stack Structure: new displacement \neg in $\{1 \text{ to } 2^{**}16-1\}$
Stack-Underflow

Also see aLXCH – NIRW evaluation.

STOD

Name: store delete
Encoding: 1 syllable ("B8")

Stack state transformation:



Interrupts:

Binding Request: IRW chain → DD with element_size = 7

Inv Object: object operand type does not match store target
or Mem[AbsentCopyDD.address] not original DD

Inv Reference: NIRW

Inv Ref Chain: See functional definition in Section 3.

Inv Stack Arg: TOS not {IRW, IndexedWordDD, opnd}
or (TOS in {IRW, IndexedWordDD} and TOS2 not opnd)
or (TOS is opnd and TOS2 not {IRW, IndexedWordDD})

Memory Protect: read_only IndexedWordDD
or reference chain → tag-3 item
or dp second word location is an odd-tagged item

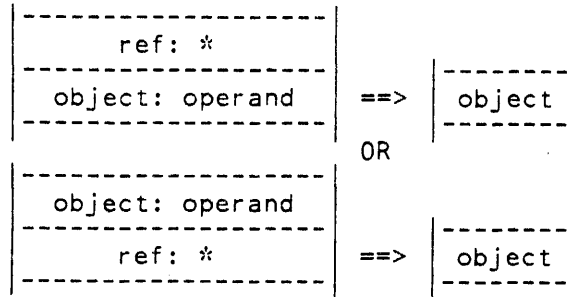
Presence Bit: IndexedWordDD
Stack-Underflow

Also see aACCE – ref chain ≥ PCW; aLXCH – NIRW evaluation; aLXLK – SIRW evaluation.

STON

Name: store non-delete
Encoding: 1 syllable ("B9")

Stack state transformation:



Interrupts: same as STOD

STOP

Name: unconditional processor halt
Encoding: 2 syllables ("95BF") Variant

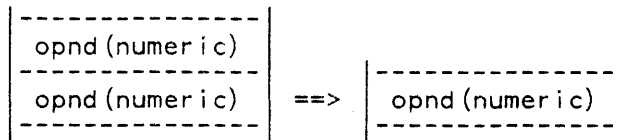
Stack state transformation: none

Interrupts: none

SUBT

Name: subtract
Encoding: 1 syllable ("81")

Stack state transformation:



Interrupts:

Exp Overflow: R(x-y) = exponent value too big

Inv Stack Arg: TOS not opnd or TOS2 not opnd
Stack-Underflow

SWFD

Name: scan while false delete
Encoding: 2 syllables ("95D4") Variant

Stack state transformation:

set: word desc

len: opnd(integer)

source: *

==> Null

Interrupts:

Int Overflow: len not sp integer

Inv Arg Value: source Pointer.char__index out of range
or len > 2**20-1

Inv Index: source' word index \neg in {0 to 2**16-1}

Inv Object: Mem[AbsentCopyDD.address] not original DD

Inv Stack Arg: set not IndexedSingleDD
or len not opnd
or source not {IndexedDD,opnd}

Memory Protect: odd-tag in set

Paged Array: source pointer

Presence Bit: set word desc or source pointer

Stack-Underflow

SWFU

Name: scan while false update
Encoding: 2 syllables ("95DC") Variant

Stack state transformation:

set: word desc

len: opnd(integer)

source: *

==>

len'

source'

Interrupts: same as SWFD

SWTD

Name: scan while true delete
Encoding: 2 syllables ("95D5") Variant
Otherwise see SWFD.

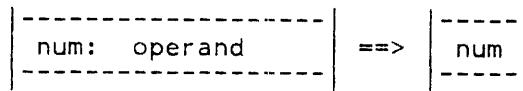
SWTU

Name: scan while true update
 Encoding: 2 syllables ("95DD") Variant
 Otherwise see SWFU.

SXSN

Name: set external sign flip-flop
 Encoding: 1 syllable ("D6")

Stack state transformation:

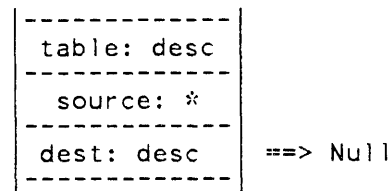


Interrupts:
 Inv Stack Arg: num not opnd
 Stack-Underflow

TEED

Name: table enter edit delete
 Encoding: 1 syllable ("D0")

Stack state transformation:



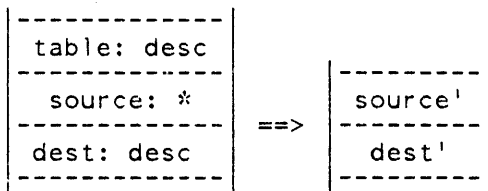
Interrupts:
 Inv Arg Value: table.esi \neg in {0 to 5}
 or source/dest Pointer.char_index out of range
 or len > 2**20-1 (restart state)
 Inv Index: table' word index \neg in {0 to 2**13-1}
 Inv Object: Mem[AbsentCopyDD.address] not original DD
 Inv Stack Arg: table not IndexedDD or source not {IndexedDD,opnd} or dest not IndexedDD
 Presence Bit: micro-op table pointer
 or source or dest pointer
 Stack-Underflow

TEEU

Name: table enter edit update

Encoding: 1 syllable ("D8")

Stack state transformation:



NOTE

Final stack state is produced by ENDE.

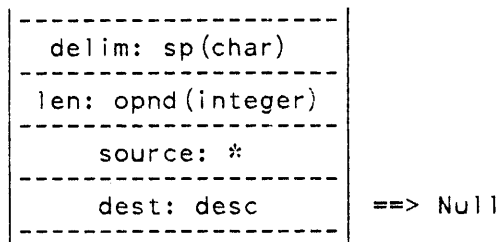
Interrupts: same as TEED.

TEQD

Name: transfer while equal delete

Encoding: 1 syllable ("E4")

Stack state transformation:



Interrupts:

Int Overflow: len not sp integer

Inv Arg Value: source/dest Pointer.char_index out of range
or len > 2**20-1

Inv Index: source' or dest' word index \neg in
{0 to 2**16-1}

Inv Object: Mem[AbsentCopyDD.address] not original DD

Inv Stack Arg: delim not sp or len not opnd
or source not {IndexedDD,opnd}
or dest not IndexedDD
or (source = EBCDIC(hex) and dest = hex(EBCDIC))

Memory Protect: read_only dest pointer

Paged Array: source or dest pointer

Presence Bit: source or dest pointer

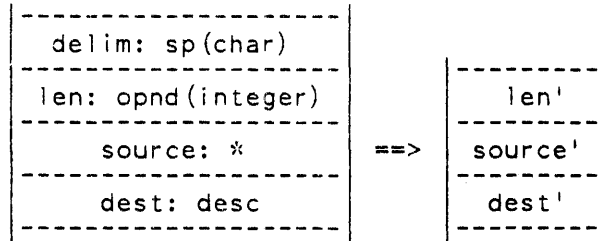
Stack-Underflow

TEQU

Name: transfer while equal update

Encoding: 1 syllable ("EC")

Stack state transformation:



Interrupts: same as TEQD

TGED

Name: transfer while greater or equal delete

Encoding: 1 syllable ("E1")
Otherwise see TEQD.

TGEU

Name: transfer while greater or equal update

Encoding: 1 syllable ("E9")
Otherwise see TEQU.

TGTD

Name: transfer while greater delete

Encoding: 1 syllable ("E2")
Otherwise see TEQD.

TGTU

Name: transfer while greater update

Encoding: 1 syllable ("EA")
Otherwise see TEQU.

TLED

Name: transfer while less or equal delete

Encoding: 1 syllable ("E3")
Otherwise see TEQD.

TLEU

Name: transfer while less or equal update
Encoding: 1 syllable ("EB")
Otherwise see TEQU.

TLSD

Name: transfer while less delete
Encoding: 1 syllable ("E0")
Otherwise see TEQD.

TLSU

Name: transfer while less update
Encoding: 1 syllable ("E8")
Otherwise see TEQU.

TNED

Name: transfer while not equal delete
Encoding: 1 syllable ("E5")
Otherwise see TEQD.

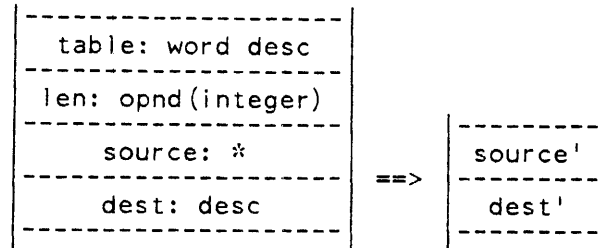
TNEU

Name: transfer while not equal update
Encoding: 1 syllable ("ED")
Otherwise see TEQU.

TRNS

Name: translate
Encoding: 2 syllables ("95D7") Variant

Stack state transformation:



Interrupts:

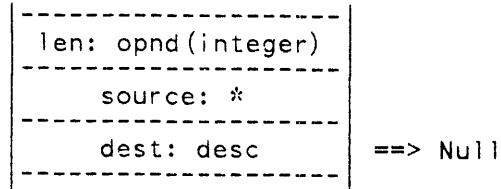
Int Overflow: len not sp integer
 Inv Arg Value: source/dest Pointer.char__index out of range
 or len > 2**20-1
 Inv Index: source' or dest' word index \neg in {0 to 2**16-1}
 Inv Object: Mem[AbsentCopyDD.address] not original DD
 Inv Stack Arg: table not IndexedSingleDD or len not opnd
 or source not {IndexedDD,opnd}
 or dest not IndexedDD
 Memory Protect: read_only dest pointer
 or odd-tag in table
 Paged Array: source or dest pointer
 Presence Bit: table word desc or source or dest pointer
 Stack-Underflow

TUND

Name: transfer characters unconditional delete

Encoding: 1 syllable ("E6")

Stack state transformation:



Interrupts:

Int Overflow: len not sp integer

Inv Arg Value: source/dest Pointer.char__index out of range
or len > 2**20-1

Inv Index: source' or dest' word index \neg in {0 to 2**16-1}

Inv Object: Mem[AbsentCopyDD.address] not original DD

Inv Stack Arg: len not opnd or source not {IndexedDD,opnd}
or dest not IndexedDD
or (source = EBCDIC(hex) and dest = hex(EBCDIC))

Memory Protect: read__only dest pointer

Paged Array: source or dest pointer

Presence Bit: source or dest pointer

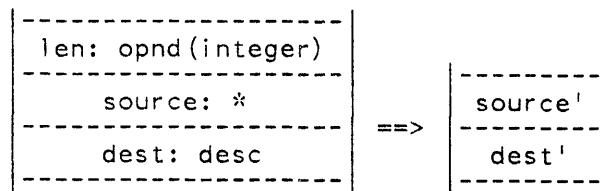
Stack-Underflow

TUNU

Name: transfer characters unconditional update

Encoding: 1 syllable ("EE")

Stack state transformation:

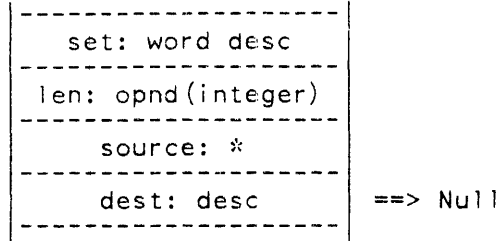


Interrupts: same as TUND

TWFD

Name: transfer while false delete
 Encoding: 2 syllables ("95D2") Variant

Stack state transformation:



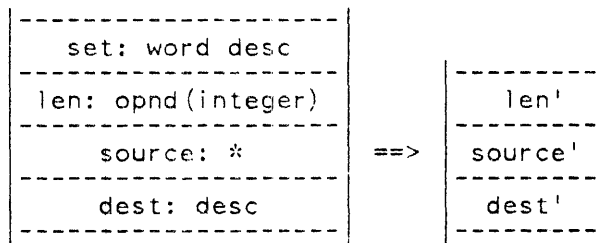
Interrupts:

Int Overflow: len not sp integer
 Inv Arg Value: source/dest Pointer.char_index out of range or len > 2**20-1
 Inv Index: source' or dest' word index \neg in {0 to 2**16-1}
 Inv Object: Mem[AbsentCopyDD.address] not original DD
 Inv Stack Arg: set not IndexedSingleDD or len not opnd or source not {IndexedDD,opnd} or dest not IndexedDD or (source = EBCDIC(hex) and dest = hex(EBCDIC))
 Memory Protect: read_only dest pointer or odd-tag in set
 Paged Array: source or dest pointer
 Presence Bit: set word desc or source or dest pointer
 Stack-Underflow

TWFDU

Name: transfer while false update
 Encoding: 2 syllables ("95DA") Variant

Stack state transformation:



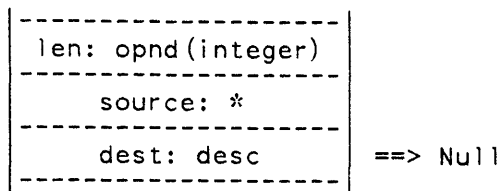
Interrupts: same as TWFD

TWOD

Name: transfer words overwrite delete

Encoding: 1 syllable ("D4")

Stack state transformation:



Interrupts:

Int Overflow: len not sp integer

Inv Arg Value: source/dest Pointer.char_index out of range
or len > 2**20-1

Inv Index: source' or dest' word index \neg in {0 to 2**k-1}, where k = 20 for IndexedWordDD
or 16 for pointer or source or destination
pointer has word_index = 2**16-1 and char_index > 0

Inv Object: Mem[AbsentCopyDD.address] not original DD

Inv Stack Arg: len not opnd or source not {IndexedDD,opnd}
or dest not IndexedDD

Memory Protect: read_only dest IndexedDD

Presence Bit: source or dest IndexedDD

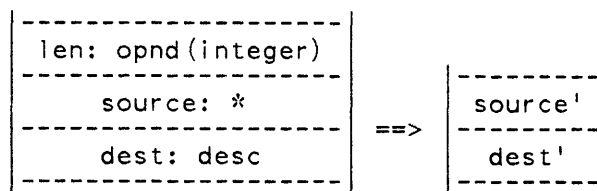
Stack-Underflow

TWOU

Name: transfer words overwrite update

Encoding: 1 syllable ("DC")

Stack state transformation:



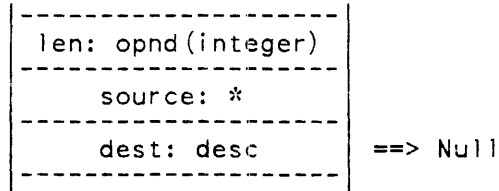
Interrupts: same as TWOD

TWSD

Name: transfer words delete

Encoding: 1 syllable ("D3")

Stack state transformation:



Interrupts:

Int Overflow: len not sp integer

Inv Arg Value: source/dest Pointer.char__index out of range
or len > 2**20-1

Inv Index: source' or dest' index \neg in {0 to 2**k-1},
where k = 20 for IndexedWordDD, or 16
for pointer or source, or destination pointer
has word__index = 2**16-1 and char__index
> 0

Inv Object: Mem[AbsentCopyDD.address] not original DD

Inv Stack Arg: len not opnd or source not
{IndexedDD,opnd}
or dest not IndexedDD

Memory Protect: read__only dest IndexedDD

Paged Array: source or dest IndexedDD

Presence Bit: source or dest IndexedDD

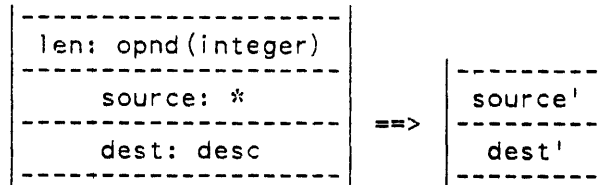
Stack-Underflow

TWSU

Name: transfer words update

Encoding: 1 syllable ("DB")

Stack state transformation:



Interrupts: same as TWSD

TWTD

Name: transfer while true delete
Encoding: 2 syllables ("95D3") Variant
Otherwise see TWFD.

TWTU

Name: transfer while true update
Encoding: 2 syllables ("95DB") Variant
Otherwise see TWFU.

UNLK

Name: unlock interlock
Encoding: 2 syllables ("95B2") Variant

Stack state
transformation:

-----		ref: *		==> Null

Interrupts: same as LOKC, plus:
Unlocking: interlock status not Locked_Uncontended

UPLD

Name: unpack left-signed delete
Encoding: 2 syllables ("9570") Variant
Otherwise see UPUD.

UPLU

Name: unpack left-signed update
Encoding: 2 syllables ("9578") Variant
Otherwise see UPUU.

UPRD

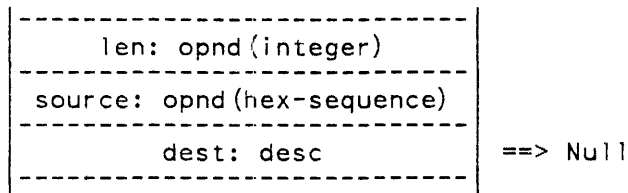
Name: unpack right-signed delete
Encoding: 2 syllables ("9571") Variant
Otherwise see UPUD.

UPRU

Name: unpack right-signed update
 Encoding: 2 syllables ("9579") Variant
 Otherwise see UPUU.

UPUD

Name: unpack unsigned delete
 Encoding: 2 syllables ("95D1") Variant
 Stack state transformation:

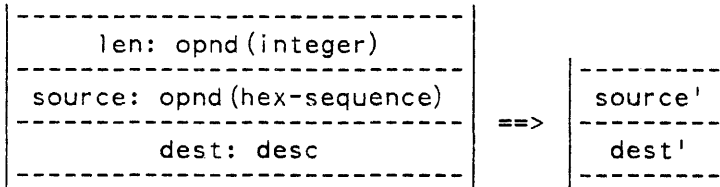


Interrupts:

Int Overflow: len not sp integer
 Inv Arg Value: len > 24
 or dest pointer.char__size out of range
 Inv Index: dest' word index \neg in {0 to 2**16-1}
 Inv Object: Mem[AbsentCopyDD.address] not original DD
 Inv Stack Arg: len not opnd, or source not opnd, or
 dest not IndexedDD
 Memory Protect: read_only dest pointer
 Paged Array: dest pointer
 Presence Bit: dest pointer
 Stack-Underflow

UPUU

Name: unpack unsigned update
 Encoding: 2 syllables ("95D9") Variant
 Stack state transformation:



Interrupts: same as UPUD

USND

Name: unpack signed delete
Encoding: 2 syllables ("95D0") Variant
Otherwise see UPUD.

USNU

Name: unpack signed update
Encoding: 2 syllables ("95D8") Variant
Otherwise see UPUU.

VALC

Name: value call
Encoding: 2 syllables (binary 00:2, AddressCouple:14)

Stack state
transformation:

Null ==>

----- opnd -----

Interrupts:

Binding Request: IRW chain → DD with element_size = 7
Inv Object: Mem[AbsentCopyDD.address] not original DD
Inv Reference: lambda > ll
Inv Ref Chain: See functional definition in section 3.
Inv Stack Arg: restart TOS not {SIRW, IndexedWordDD,
operand}
Presence Bit: IndexedWordDD
Stack-Overflow:

Also see aACCE – ref chain ≥ PCW; aLXCH – address-couple parameter evaluation; aLXLK

VARI

Name: introduce variant operator
Encoding: 1 syllable ("95")
Stack state
transformation: none
Interrupts: none

WATI

Name: read machine identification

Encoding: 2 syllables ("95A4") Variant

Stack state transformation: Null ==>

dp(machine id)

Interrupts:
Stack-Overflow

WEMC

Name: write external memory control

Encoding: 2 syllables ("9593") Variant

Stack state transformation:

sp
sp

 ==> Null

Interrupts:
Stack-Underflow (Others are implementation-defined.)

WHOI

Name: read processor identification

Encoding: 2 syllables ("954E") Variant

Stack state transformation: Null ==>

sp(proc id)

Interrupts:
Stack-Overflow

WIPS

Name: write internal processor state

Encoding: 2 syllables ("9599") Variant

Stack state transformation:

sp
sp

 ==> Null

Interrupts:
Stack-Underflow (Others are implementation-defined.)

WTOD

Name: write time-of-day clock
Encoding: 2 syllables ("9549") Variant

Stack state transformation:

time: sp(integer)	==>	Null
-------------------	-----	------

Interrupts:
Stack-Overflow:

Also see aISX: time not 36-bit integer.

XTND

Name: set to double-precision
Encoding: 1 syllable ("CE")

Stack state transformation:

opnd(numeric)	==>	dp(numeric)
OR		
word desc	==>	dp desc

Interrupts:
Inv Stack Arg: TOS not {opnd,word desc}
Stack-Underflow

ZERO

Name: insert literal zero
Encoding: 1 syllable ("B0")
Otherwise see LT8.

ZIC

Name: zero Interrupt_Count
Encoding: 2 syllables ("9540") Variant
Stack state transformation: none
Interrupts: none

APPENDIX C

OPERATOR DEPENDENT INTERRUPT REFERENCE SUMMARIES

GENERAL INFORMATION

Operator Dependent Interrupts are listed alphabetically. For each interrupt, the operators and functions that cause the interrupt are listed alphabetically. Where relevant, brief statements of various conditions under which an operator or function generates an interrupt are included. In some cases, operators and functions which generate a particular interrupt are grouped into classes.

The association of Interrupt conditions with a class of operators, individual operator, or function, is indicated by use of ellipses (series of periods or dots). If a class, operator, or function, has multiple conditions itemized, no precedence or priority is implied, all conditions listed are itemized as though they are parallel to each other. If a common condition generates an interrupt during multiple classes, operators, or functions, the condition is itemized under each class, operator, or function, for each interrupt type.

The Itemization of Conditions under which interrupts are generated contain the following nonstandard terms and abbreviations.

Address(name)	the nominal memory address associated with "name".
C	resumption condition is Continue.
Dest	destination array specification for pointer operators.
Dest' word index	computed word-index of an updated destination pointer.
Dp	double-precision.
HistLink	the stack index computed from a history link.
LexLink	the address computed from a lexical link.
ll	lex___level.
Mem[Ref]	The item stored in the memory location addressed by Ref (Ref may be, for instance, a descriptor, IRW, or nominal address.)
Opnd	operand.
OPT	OPTIONAL (interrupt condition detection is implementation option.)
R-I	resumption condition is Repeat-Initial.
R-R	resumption condition is Repeat-Restart.
R.*	resumption condition is Repeat-Initial or Repeat-Restart, according to whether the operator began in initial or restart state, respectively.
Source	source array specification for pointer operators.
Source digit	non-signed hexadecimal character or a 4-bit EBCDIC numeric digit.
Source' word index	computed word-index of an updated source pointer.
Sp	single-precision.

Stack[i]	The item stored at index-i in the current stack.
SVD	stack-vector descriptor.
TOS	Top-of-stack item.
TOSi	i-th item from the top of the stack (TOS1 = TOS).
Word (as modifier of descriptor)	descriptor.element__size is sp or dp.
TooBig	The exponent-value is too large to fit the available exponent-field container (space).
TooSmall	The exponent-value is too small to fit the available exponent-field container (space).

Unless restart state is specified explicitly, the interrupt descriptions that follow apply to the initial state of the operator. For string-isolate, pack, input-convert, and enter-table-edit operators, the presence of an additional argument in restart state changes the TOSi positions of some other arguments.

Binding Request

The following operators generate a Binding Request interrupt if a descriptor with an element__size of 7 is encountered in a reference chain:

aINTE ENTR INDX INXA LVLC NXLN NXLV NXVA STAD STAN STOD STON VALC
MKS-NAMC (optional implementation, anticipating ENTR)

The following operators generate a Binding Request interrupt if a copy descriptor with an element__size of 7 is encountered as a Descriptor Indication argument:

INDX NXLN NXLV

Block Exit

EXIT,RETN RCW.block__exit = 1

Code Segment Error

The common action aPCW can generate Code Segment Error interrupts while distributing a new code-sequence pointer, if the location referenced by the PCW (RCW, for EXIT and RETN) does not contain a code-segment descriptor (with tag = 3). The test is OPTIONAL.

Divide by Zero

The following operators generate a Divide by Zero interrupt if the numeric interpretation of the top-of-stack operand (divisor) is zero:

DIVD IDIV RDIV

Exponent Overflow

The following operators generate an Exponent-Overflow interrupt if the result of a rounding or truncation function is TooBig. For binary operators, the second from top-of-stack operand is denoted x and the top operand y; for unary operators, the top-of-stack operand is denoted x.

ADD	$R(x + y) = \text{TooBig}$
DIVD	$R(x/y) = \text{TooBig}$
MULT,MULX	$R(x*y) = \text{TooBig}$
SNGL	$RS(ND(x)) = \text{TooBig}$
SNGT	$TS(ND(x)) = \text{TooBig}$
SUBT	$R(x-y) = \text{TooBig}$

Exponent Underflow

The following operators generate an Exponent-Underflow interrupt if the result of a rounding, truncation, or normalization function is TooSmall. For binary operators, the second from top-of-stack operand is denoted x and the top operand y; for unary operators, the top-of-stack operand is denoted x.

DIVD	$R(x/y) = \text{TooSmall}$
MULT,MULX	$R(x*y) = \text{TooSmall}$
NORM	$N(x) = \text{TooSmall}$
SNGL	$N(x) = \text{TooSmall}$ or $RS(ND(x)) = \text{TooSmall}$ or $NS(RS(ND(x))) = \text{TooSmall}$
SNGT	$N(x) = \text{TooSmall}$ or $TS(ND(x)) = \text{TooSmall}$ or $NS(TS(ND(x))) = \text{TooSmall}$

False Assertion

A False Assertion interrupt is generated only by the ASRT (assert) operator when its stack input is False. The operator's single-parameter syllable is the P2 parameter.

Integer Overflow

Conditions under which Integer-Overflow interrupts are generated are denoted as "stack item" not "type" integer (not representable as an integer), where "type" is sp or dp.

aISX	argument not sp integer (See aISX summary.)
BCD	TOS opnd not dp integer
CEQD,CEQU,CGED,CGEU, CGTD,CGTU,CLED,CLEU, CLSD,CLSU,CNED,CNEU	TOS opnd not sp integer
DBCD	TOS opnd not sp integer or TOS2 opnd not dp integer
DBFL,DBTR	TOS opnd not sp integer
DBRS,DBST	TOS opnd not sp integer
DBUN	TOS opnd not sp integer
DFTR	TOS or TOS2 or TOS3 opnd not sp integer
DINS	TOS2 or TOS3 opnd not sp integer
DISO	TOS or TOS2 opnd not sp integer

DSLF, DSRF,DSRR,DSRS,DSRT	TOS opnd not sp integer or TOS2 opnd not dp integer
EXPU,EXSD,EXSU	TOS opnd not sp integer
ICLD,ICRD,ICUD, ICVD,ICVU	TOS opnd not sp integer
IDIV	result not sp or dp integer (Result type depends on argument types.)
INDX	index (TOS or TOS2) opnd not sp integer
INXA	TOS opnd not sp integer
LLLU	TOS or TOS3 opnd not sp integer
LODT	argument not sp integer (see LODT summary)
NTGD,NTTD	TOS opnd not dp integer
NTGR,NTIA	TOS opnd not sp integer
NXLN,NXLV	index (TOS or TOS2) opnd not sp integer
NXVA	TOS opnd not sp integer
OCRX	TOS2 opnd not sp integer
PACD,PACU,	
PKLD,PKRD,PKUD	TOS opnd not sp integer
RDIV	same conditions as IDIV – result would be Integer-Overflow
SCLF, SCRF,SCRR,SCRS,SCRT	TOS opnd not dp integer
SEQD,SEQU,SGED,SGEU, SGTD,SGTU,SLED,SLEU, SLSD,SLSU,SNED,SNEU	TOS2 opnd not sp integer
SHOW	TOS not sp integer
SISO	TOS opnd not sp integer
SWFD,SWFU,SWTD,SWTU	TOS2 opnd not sp integer
TEQD,TEQU,TGED,TGEU, TGTD,TGTU,TLED,TLEU, TLSD,TLU,SNED,TNEU	TOS2 opnd not sp integer
TRNS	TOS2 opnd not sp integer
TUND,TUNU	TOS opnd not sp integer
TWFD,TWFU,TWTD,TWTU	TOS2 opnd not sp integer
TWOD,TWOU,TWSD,TWSU	TOS opnd not sp integer
UPLD,UPLU,UPRD,UPRU, UPUD,UPUU,USND,USNU	TOS opnd not sp integer

Invalid Address is an Alarm interrupt, but it can be generated in one operator-dependent context; when the LODT argument is an integer, but not within the range of a nominal memory address. An implementation may be defined to generate an Invalid Argument Value or Invalid Stack Argument interrupt in the same circumstance.

LODT TOS integer \neg in {0 to $2^{**}20-1$ }

Invalid Argument Value

Conditions under which Invalid Argument Value interrupts are generated are denoted as $\langle \text{argument} \rangle$ relational expression or $\langle \text{argument} \rangle \neg \text{in } \{\text{valid range}\}$, where $\langle \text{argument} \rangle$ may be a $\langle \text{type name} \rangle . \text{field}$ or a stack-item specification.

aACCE,aINTE	(Mem[F+1] = NIRW directly to PCW and PCW.ll > 0 and PCW.ll - 1 \neg = NIRW.lambda) (OPTIONAL) or PCW.ll - 1 \neg = MSCW.ll (OPTIONAL) or PCW.invalid_ll \neg = 0
aISX	argument not k-bit integer (see aISX summary)
aPRCW	(PCW or RCW).psi \neg in {0 to 5} (OPTIONAL)
CUIO	Mem[TOS descriptor].[47:16] \neg = hex"10CB"
DBCD	TOS opnd \neg in {0 to 24}
DBFL,DBTR,DBUN	PCW.ll \neg = LL (OPTIONAL) or branch-dest opnd \neg in {0 to 2**16-1} (optionally reportable as Invalid Index) or PCW.sdll \neg = SDLL (OPTIONAL)
DBRS,DBST	TOS opnd \neg in {0 to 47}
DFTR	TOS opnd \neg in {0 to 48} or TOS2 opnd \neg in {0 to 47} or TOS3 opnd \neg in {0 to 47}
DINS	TOS2 opnd \neg in {0 to 48} or TOS3 opnd \neg in {0 to 47}
DISO	TOS opnd \neg in {0 to 48} or TOS2 opnd \neg in {0 to 47}
DSLRF, DSRF,DSRR,DSRS,DSRT ENTR	TOS opnd \neg in {0 to 12} (Mem[F+1] = NIRW directly to PCW and PCW.ll > 0 and PCW.ll - 1 \neg = NIRW.lambda) (OPTIONAL) or PCW.ll \neg in {0, MSCW.ll+1} (OPTIONAL) or PCW.invalid_ll \neg = 0
ICLD,ICRD,ICUD, ICVD,ICVU	TOS opnd > 23
LODT	argument not k-bit integer (see LODT summary)
MVST	ENR value too large for container (OPTIONAL if container size = 0)
PACD,PACU, PKLD,PKRD,PKUD	TOS opnd > 24
RPRR	TOS not in {0, LL, 36-38, 52-53, 58}
SISO	(source = EBCDIC and TOS opnd > 12) or (source = hex and TOS opnd > 24)
SNGT	TOS = unindexed DoubleDD and length > 2**19-1

SPRR	TOS1 not in {0, LL, 36-38, 52-53, 58}
TEED,TEEU	table descriptor.esi \neg in {0 to 5} (OPTIONAL)
UPLD,UPLU,UPRD,UPRU, UPUD,UPUU,USND,USNU	TOS opnd > 24

All pointer operators can generate the interrupt if the char_index field of a source or destination pointer is not in the proper range ({0 to 5} for EBCDIC, {0 to 11} for hexadecimal). The checks are OPTIONAL.

CEQD CEQU CGED CGEU CGTD CGTU CLED CLEU CLSD CLSU CNED CNEU
EXPU EXSU EXSD
ICLD ICRD ICUD ICVD ICVU
PKLD PKRD PKUD PACD PACU
SEQD SEQU SGED SGEU SGTD SGTU SLED SLEU SLSD SLSU SNED SNEU
SHOW
SWFD SWFU SWTD SWTU
TEED TEEU
TEQD TEQU TGED TGEU TGTD TGTU TLED TLEU TLSD TLSU TNED TNEU
TRNS
TUND TUNU
TWFD TWFU TWTD TWTU
TWOD TWOU
TWSL TWSU
UPLD UPLU UPRD UPRU UPUD UPUU USND USNU

All pointer operators can generate the interrupt if the length argument exceeds $2^{*}20+1$. The check is OPTIONAL for the SHOW operator and applies to TEED and TEEU only in restart state. (Some pointer operators have a more restrictive limit, specifically string-isolate, input-convert, pack and un-pack.)

Invalid Code Parameter

Conditions under which Invalid Code Parameter interrupts are generated are denoted as parameter name > maximum valid value; all these tests are OPTIONAL.

BCD	N > 24
BRFL,BRTR	op_psi > 5
BRST,BSET	Db > 47
BRUN	op_psi > 5
FLTR	Db > 47 or Sb > 47 or Len > 48
INSR	Db > 47 or Len > 48
ISOL	Sb > 47 or Len > 48
SCLF, SCRF,SCRR,SCRS,SCRT	ScaleFactor > 12

Invalid Index

Conditions under which Invalid Index interrupts are generated are denoted "index value" \neg in {valid range}.

NOTE

Tests for word index $< 2^{**16}$ or 2^{**20} for pointer updates are shown for both delete and update forms of the pointer operators. Only the update versions can generate the interrupt at normal termination, but both versions can generate the interrupt if update is required in mid-operator, such as for another interrupt.

aLXLK	stack__number \neg in {0 to SVD.length-1} (OPTIONAL)
aPRCW	(PCW or RCW).pwi \neg in {0 to CSD.seg__length-1} (OPTIONAL)
BRFL,BRTR,BRUN	op__pwi (param.) \neg in {0 to CSD.seg__length- 1} (OPTIONAL)
CEQD,CEQU,CGED,CGEU, CGTD,CGTU,CLED,CLEU, CLSD,CLSU,CNED,CNEU	source1'/source2' word index \neg in {0 to $2^{**16}-1$ }
DBFL,DBTR,DBUN	TOS opnd.dyn pwi \neg in {0 to CSD.seg__length-1} (OPTIONAL)
ENDE	source' or dest' word index \neg in {0 to $2^{**16}-1$ }
ENDF	dest' word index \neg in {0 to $2^{**16}-1$ }
ICLD,ICRD,ICUD, ICVD,ICVU	source' word index \neg in {0 to $2^{**16}-1$ }
INDX	index (TOS or TOS2 opnd) \neg in {0 to DD.length-1} or (unpaged CharDD and word index \neg in {0 to $2^{**16}-1$ }) or (unpaged DoubleDD and (doubled) word index \neg in {0 to $2^{**20}-1$ })
INOP,INSC,INSG,INSU	dest' word index \neg in {0 to $2^{**16}-1$ }
INXA	TOS opnd \neg in {0 to DD.length-1} or (unpaged CharDD and word index \neg in {0 to $2^{**16}-1$ }) or (unpaged DoubleDD and (doubled) word index \neg in {0 to $2^{**20}-1$ })
LLLU	any index value \neg in {0 to DD.length-1}
MCHR,MFLT,MINS,MVNU	source' or dest' word index \neg in {0 to $2^{**16}-1$ }
MVST	Stack number not in {0 to SVD.length-1} (OPTIONAL)
NXLN,NXLV	index (TOS or TOS2 opnd) \neg in {0 to DD.length-1} or (unpaged DoubleDD and (doubled) word index \neg in {0 to $2^{**20}-1$ })
NXVA	TOS opnd \neg in {0 to DD.length-1} or (unpaged DoubleDD and (doubled) word index \neg in {0 to $2^{**20}-1$ })
OCRX	TOS2 opnd \neg in {1 to TOS.ICW__limit}

PACD,PACU, PKLD,PKRD,PKUD	source' word index \neg in {0 to $2^{**}16-1$ }
SEQD,SEQU,SGED,SGEU, SGTD,SGTU,SLED,SLEU, SLSD,SLSU,SNED,SNEU	source' word index \neg in {0 to $2^{**}16-1$ }
SFDC,SRDC	dest' word index \neg in {0 to $2^{**}16-1$ }
SFSC,SRSC	source' word index \neg in {0 to $2^{**}16-1$ }
SHOW	source' word index \neg in {0 to $2^{**}16-1$ }
SISO	source' word index \neg in {0 to $2^{**}16-1$ }
SWFD,SWFU,SWTD,SWTU	source' word index \neg in {0 to $2^{**}16-1$ }
TEED,TEEU	table' word index \neg in {0 to $2^{**}13-1$ }
TEQD,TEQU,TGED,TGEU, TGTD,TGTU,TLED,TLEU, TLSD,TLSU, TNED, TNEU	source' or dest' word index \neg in {0 to $2^{**}16-1$ }
TRNS	source' or dest' word index \neg in {0 to $2^{**}16-1$ }
TUND,TUNU	source' or dest' word index \neg in {0 to $2^{**}16-1$ }
TWFD,TWFU,TWTD,TWTU	source' or dest' word index \neg in {0 to $2^{**}16-1$ }
TWOD,TWOU,TWSD,TWSU	source' or dest' word index \neg in {0 to $2^{**}k-1$ }, where $k = 20$ for IndexedWordDD, 16 for pointer or source, or destination pointer has <code>word_index = $2^{**}16-1$</code> and <code>char_index > 0</code>
UPLD,UPLU,UPRD,UPRU, UPUD,UPUU,USND,USNU	dest' word index \neg in {0 to $2^{**}16-1$ }

Invalid Object

Invalid Object interrupts are indicated with the expression "reference $\neg \rightarrow$ valid target", where applicable, or noted with a short error condition statement.

aFOP	second word of dp (accessed by means of IRW) has tag $\neg = 2$ or (accessed by means of IndexedDD) has odd tag
aLXLK	stack-vector descriptor not unpagged original SingleDD (OPTIONAL) or stack descriptor not unpagged unindexed SingleDD (OPTIONAL)
DBFL,DBTR,DBUN	NIRW $\neg \rightarrow$ PCW
LKID	ref $\neg \rightarrow$ word with tag in {0,3} (OPTIONAL)
LOAD	ref $\neg \rightarrow$ (SIRW, DD, even-tag word) or IndexedDoubleDD $\neg \rightarrow$ operand

MVST	stack-vector descriptor not unpagged original SingleDD (OPTIONAL) or stack descriptor not unpagged unindexed SingleDD (OPTIONAL)
LOK,LOKC	ref $\neg \rightarrow$ word with tag in {0,3} (OPTIONAL)
NXLN	SingleDD [index] $\neg \rightarrow$ unindexed DD
NXLV,NXVA	WordDD [index] $\neg \rightarrow$ opnd
STAD,STAN,STOD,STON	Operand type does not match store target.
UNLK	ref $\neg \rightarrow$ word with tag in {0,3} (OPTIONAL)

In addition, all operators that generate Presence Bit interrupts on a data descriptor can, instead, generate Invalid Object interrupts if an absent copy descriptor does not refer to an original DD:

Mem[AbsentCopyDD. address] not original DD

NOTE

Because the effect of an absent stack-vector descriptor or an absent copy stack descriptor is undefined, an interrupt in this situation is OPTIONAL for aLXLK and MVST.

Invalid Operator

An Invalid Operator interrupt is generated only by NVLD (invalid operator). NVLD is encoded in both primary and variant modes.

Invalid Reference

An Invalid Reference interrupt may be generated by evaluation of an NIRW or an address-couple parameter. In the case of NAMC and LNMC, the interrupt can be generated in examining the address-couple without evaluation. The tests are OPTIONAL.

DBFL,DBTR,DBUN	NIRW
ENTR	NIRW
EVAL	NIRW
INDX	NIRW
INXA	address-couple parameter
LKID	NIRW
LOAD,LODT	NIRW
LNMC	address-couple parameter
LVLC	address-couple parameter
LOK,LOKC	NIRW
MKSN-NAMC	address-couple parameter (optional implementation)
NAMC	address-couple parameter
NXLN,NXLV	NIRW
NXVA	address-couple parameter
OVRD,OVRN	NIRW
RDLK	NIRW
STFF	NIRW
STAD,STAN	address-couple parameter
STOD,STON	NIRW
UNLK	NIRW
VALC	address-couple parameter

Invalid Reference Chain

Conditions under which Invalid Reference Chain interrupts are generated are noted by indicating invalid reference chains as "reference chain $\neg \rightarrow$ valid reference or target", where feasible. Operators that evaluate general reference chains are marked "***". Refer to operator chaining rules as defined in section 3.

aINTE	IRW chain $\neg \rightarrow$ (PCW or (DD with element__size = 7))
ENTR	IRW chain $\neg \rightarrow$ (PCW or (DD with element__size = 7))
EVAL	**
INDX,INXA	IRW chain $\neg \rightarrow$ (unindexed WordDD, unindexed CharDD, or (DD with element__size = 7))
LVLC	**
MKSN-NAMC	(optional implementation, anticipating ENTR) IRW chain $\neg \rightarrow$ (PCW or (DD with element__size = 7))
NXLN	IRW chain $\neg \rightarrow$ (unindexed SingleDD or (DD with element__size = 7))
NXLV,NXVA	IRW chain $\neg \rightarrow$ (unindexed WordDD or (DD with element__size = 7))
STAD,STAN,STOD,STON	**
VALC	**

Invalid Stack Argument

Conditions under which Invalid Stack Argument interrupts are generated are denoted as "stack item" not "required type." "Required type" is a data type or set of types defined in Section 1 of this manual.

aISX	Argument not k-bit integer
ADD	TOS not opnd or TOS2 not opnd
AMIN,AMAX	TOD not opnd or TOS2 not opnd
ASRT	TOS not opnd
BCD	TOS not opnd
BRFL,BRTR	TOS not opnd
CBON	TOS not opnd
CEQD,CEQU,CGED,CGEU, CGTD,CGTU,CLED,CLEU, CLSD,CLSU,CNED,CNEU	TOS not opnd or TOS2 not {IndexedDD,opnd} or TOS3 not IndexedDD or TOS2 = EBCDIC(hex) and TOS3 = hex(EBCDIC)

CHSN	TOS not opnd
CUIO	TOS not present unpagged unindexed copy SingleDD
DBCD	TOS not opnd or TOS2 not opnd
DBFL,DBTR	TOS not {opnd,PCW,NIRW} TOS2 not opnd
DBRS,DBST	TOS not opnd
DBUN	TOS not {opnd,PCW,NIRW}
DFTR	TOS not opnd or TOS2 not opnd or TOS3 not opnd
DINS	TOS2 not opnd or TOS3 not opnd
DISO	TOS not opnd or TOS2 not opnd
DIVD	TOS not opnd or TOS2 not opnd
DRNT	TOS not opnd or TOS2 not opnd or TOS3 not opnd
DSLFL,DSRF,DSRR, DSRs,DSRT	TOS not opnd or TOS2 not opnd
ENTR	Mem[F + 1] not IRW
EQUL	TOS not opnd or TOS2 not opnd
EVAL	TOS not {IRW,IndexedWordDD}
EXPU	TOS not opnd or TOS2 not IndexedDD
EXSD,EXSU	TOS not opnd or TOS2 not {IndexedDD,opnd} or TOS3 not IndexedDD
GREQ,GRTR	TOS not opnd or TOS2 not opnd
ICLD,ICRD,ICUD, ICVD,ICVU	TOS not opnd or TOS2 not {IndexedDD,opnd}
IDIV	TOS not opnd or TOS2 not opnd
INDX	(TOS not {unindexed copy WordDD, unindexed copy CharDD,IRW} or TOS2 not opnd) and (TOS not opnd or TOS2 not {unindexed copy WordDD, unindexed copy CharDD,IRW})
INXA	TOS not opnd
INOP	dest.element__size = hex
INSG	dest.element__size = hex
JOIN	TOS not opnd or TOS2 not opnd
LESS	TOS not opnd or TOS2 not opnd
LKID	TOS not {IRW, IndexedSingleDD}
LLLU	TOS or TOS3 not opnd or TOS2 not unpagged unindexed copy SingleDD
LOAD	TOS not {IRW,IndexedWordDD}

LODT	TOS not {IRW,IndexedSingleDD,20-bit integer}
LOK,LOKC	TOS not {IRW, IndexedSingleDD}
LSEQ	TOS not opnd or TOS2 not opnd
LVLC (restart)	TOS not {SIRW, IndexedWordDD, operand}
MCHR,MFLT,MINS,MVNU	source = EBCDIC(hex) and dest = hex (EBCDIC)
MULT,MULX	TOS not opnd or TOS2 not opnd
MVST	TOS not single-precision operand
NEQL	TOS not opnd or TOS2 not opnd
NORM	TOS not opnd
NTGD,NTGR,NTIA,NTTD	TOS not opnd
NXLV	(TOS not {unindexed copy WordDD,IRW} or TOS2 not opnd) and (TOS not opnd or TOS2 not {unindexed copy WordDD,IRW})
NXVA	TOS not opnd
OCRX	TOS not sp or TOS2 not opnd
OVRD,OVRN	TOS not {IRW,IndexedSingleDD}
PACD,PACU,	
PKLD,PKRD,PKUD	TOS not opnd or TOS2 not {IndexedDD,opnd}
RDIV	TOS not opnd or TOS2 not opnd
RDLK	TOS not {IRW,IndexedSingleDD}
RETN	TOS = NIRW
RNGT	TOS not opnd
SCLF	TOS not opnd
SCRF,SCRR,SCRS,SCRT	TOS not opnd
SEQD,SEQU,SGED,SGEU, SGTD,SGTU,SLED,SLEU, SLSD,SLSU,SNED,SNEU	TOS not sp or TOS2 not opnd or TOS3 not {IndexedDD,opnd}
SHOW	TOS not opnd or TOS2 not {EBCDIC pointer,IndexedWordDD,opnd}
SISO	TOS not opnd or TOS2 not {IndexedDD,opnd}
SNGL	TOS not opnd
SNGT	TOS not {opnd,word descriptor}
SPLT	TOS not opnd
SRCH	TOS \neg in {unpaged unindexed copy SingleDD, IndexedSingleDD}
STAG	TOS not sp
STFF	TOS not IRW

STAD,STAN (initial)	TOS not opnd
STAD,STAN (restart)	TOS not {IRW, IndexedWordDD} or TOS not opnd
STOD,STON	TOS not {IRW, IndexedWordDD, opnd} or (TOS in {IRW, IndexedWordDD} and TOS2 not opnd) (TOS is opnd and TOS2 not {IRW,IndexedWordDD})
SUBT	TOS not opnd or TOS2 not opnd
SWFD,SWFU,SWTD,SWTU	TOS not IndexedSingleDD or TOS2 not opnd or TOS3 not {IndexedDD,opnd}
SXSN	TOS not opnd
TEED,TEEU	TOS not IndexedDD or TOS2 not {IndexedDD, opnd} or TOS3 not IndexedDD
TEQD,TEQU,TGED,TGEU, TGTD,TGTU,TLED,TLEU, TLSD,TLSU,TNED,TNEU	TOS not sp or TOS2 not opnd or TOS3 not {IndexedDD,opnd} or TOS4 not IndexedDD or TOS3 = EBCDIC(hex) and TOS4 = hex(EBCDIC)
TRNS	TOS not IndexedSingleDD or TOS2 not opnd or TOS3 not {IndexedDD,opnd} or TOS4 not IndexedDD
TUND,TUNU	TOS not opnd or TOS2 not {IndexedDD,opnd} or TOS3 not IndexedDD or TOS2 = EBCDIC(hex) and TOS3 = hex(EBCDIC)
TWFD,TWFU,TWTD,TWTU	TOS not IndexedSingleDD or TOS2 not opnd or TOS3 not {IndexedDD,opnd} or TOS4 not IndexedDD or TOS3 = EBCDIC(hex) and TOS4 = hex(EBCDIC)
TWOD,TWOU,TWSD,TWSU	TOS not opnd or TOS2 not {IndexedDD,opnd} or TOS3 not IndexedDD
UPLD,UPLU,UPRD,UPRU, UPUD,UPUU,USND,USNU	TOS not opnd or TOS2 not opnd TOS3 not IndexedDD
UNLK	TOS not {IRW, IndexedSingleDD}
VALC (restart)	TOS not {SIRW, IndexedWordDD, operand}
XTND	TOS not {opnd,word descriptor}

Locking

The Locking interrupt is generated by the LOK operator when the target interlock status is not Free.

Memory Protect

Most conditions under which Memory Protect interrupts are generated are noted by referencing a read_only descriptor through which a write is attempted. Others are noted by encountering an odd-tagged word in a set or translate table.

ENDF	read__only destination pointer
INOP,INSC,INSG,INSU	read__only destination pointer
LOK,LOKC	read__only IndexedSingleDD
MCHR,MFLT,MINS,MVNU	read__only destination pointer
OVRD,OVRN	read__only IndexedSingleDD
RDLK	read__only IndexedSingleDD
SHOW	odd-tag word in source
STAD,STAN,STOD,STON	reference chain → tag-3 item or dp second- word location is an odd-tagged item or read__only IndexedWordDD
SWFD,SWFU,SWTD,SWTU	odd-tag in set
TEQD,TEQU,TGED,TGEU, TGTD,TGTU,TLED,TLEU, TLSD,TLSU,TNED,TNEU	read__only destination pointer
TRNS	read__only destination pointer or odd-tag in table
TUND,TUNU	read__only destination pointer
TWFD,TWFU,TWTD,TWTU	read__only destination pointer or odd-tag in set
TWOD,TWOU	read-only destination pointer
TWSD,TWSU	read__only destination pointer
UPLD,UPLU,UPRD,UPRU, UPUD,UPUU,USND,USNU	read__only destination pointer
UNLK	read__only IndexedSingleDD

Paged Array

Conditions under which Paged Array interrupts are generated are noted by naming the descriptor for the array that may be Paged.

Resumption condition is implementation-defined for:

restart-state string-isolate, pack, and input-convert operators.

Resumption condition is Repeat-Initial for:

compare update operators before the relation is known.
pack and unpack operators with left-sign not yet fetched.
UPLD and UPLU operators prior to storing any character.
USND and USNU operators (in hexadecimal) prior to storing any character.

Resumption condition is Repeat-Restart for:

compare operators after the relation is known.

UPLD and UPLU operators after the sign is stored.

USND and UPRU operators after the hexadecimal sign is stored.

UPRD and UPRU operators attempting to store a hexadecimal sign.

string-isolate, pack, or input-convert operators after any character has been fetched from the source.

edit operators initiated by TEED or TEEU operators, (except for an interrupt on source segment at the start of an edit operator with FLTF = 0.)

edit operators initiated by EXSD or EXSU with FLTF = 1.

Resumption condition is Repeat-IR in all other cases.

CEQD,CEQU,CGED,CGEU, CGTD,CGTU,CLED,CLEU, CLSD,CLSU,CNED,CNEU	source1 or source2 pointer
ENDF	dest pointer
ICLD,ICRD,ICUD, ICVD,ICVU	source pointer
INOP,INSC,INSG,INSU	dest pointer
MCHR,MFLT,MINS,MVNU	source or dest pointer
PACD,PACU, PKLD,PKRD,PKUD	source pointer (R-R)
SEQD,SEQU,SGED,SGEU, SGTD,SGTU,SLED,SLEU, SLSD,SLSU,SNED,SNEU	source pointer
SFDC,SRDC	dest pointer
SFSC,SRSC	source pointer
SISO	source pointer
SWFD,SWFU,SWTD,SWTU	source pointer
TEQD,TEQU,TGED,TGEU, TGTD,TGTU,TLED,TLEU, TLSD,TLSU,TNED,TNEU	source or dest pointer
TRNS	source or dest pointer
TUND,TUNU	source or dest pointer
TWFD,TWFU,TWTD,TWTU	source or dest pointer
TWSD,TWSU	source or dest pointer
UPLD,UPLU,UPRD,UPRU, UPUD,UPUU,USND,USNU	dest pointer (R-I, except R-R for signed after sign stored)

Page Structure Error

The indexing operators can generate a Page Structure Error interrupt when indexing a paged DD.

INDX,INXA	paged DD [page index] \rightarrow unpagged original SingleDD
-----------	---

NXLN	paged SingleDD [page index] $\neg \rightarrow$ unpaged original SingleDD
NXLV,NXVA	paged WordDD [page index] $\neg \rightarrow$ unpaged original SingleDD

Precision Loss

The following operators generate a Precision Loss interrupt whenever rounding is possible (no Exponent-Underflow), but precision must be lost to achieve an exponent within range.

DIVD	$R(x/y) \neg = R^*(x/y)$
MULT	$R(x*y) \neg = R^*(x*y)$
MULX	$R(x*y) \neg = R^*(x*y)$

Presence Bit

Conditions under which Presence Bit interrupts are generated are noted by naming the descriptor through which access is required or the structure that is absent.

Resumption conditions are Repeat-IR, except as specified in the following table.

aLXLK	stack descriptor
aPRCW	code-segment descriptor (C)
CEQD,CEQU,CGED,CGEU, CGTD,CGTU,CLED,CLEU, CLSD,CLSU,CNED,CNEU	source1 or source2 pointer (R-* for CxxU)
EXSD,EXSU	source or dest pointer (R-* if FLTF = 1)
EXPU	dest pointer
ICLD,ICRD,ICUD, ICVD,ICVU	source pointer (R-*)
INDX,INXA	page table
LKID	IndexedSingleDD
LLLU	SingleDD
LOAD	IndexedWordDD
LOK,LOKC	IndexedSingleDD
LODT	IndexedSingleDD
LVLC	IndexedWordDD (R-*)
MVST	destination stack descriptor
NXLN	page table or indexed SingleDD
NXLV,NXVA	page table or indexed WordDD
OVRD,OVRN	IndexedSingleDD
PACD,PACU, PKLD,PKRD,PKUD	source pointer (R-*)
RDLK	IndexedSingleDD
SEQD,SEQU,SGED,SGEU, SGTD,SGTU,SLED,SLEU, SLSD,SLSU,SNED,SNEU	source pointer

SHOW	source pointer
SISO	source pointer (R-*)
SRCH	SingleDD
STAD,STAN	IndexedWordDD (R-*)
STOD,STON	IndexedWordDD
SWFD,SWFU,SWTD,SWTU	set word descriptor or source pointer
TEED,TEEU	edit-table descriptor or (R-*) source or dest pointer (R-*)
TEQD,TEQU,TGED,TGEU, TGTD,TGTU,TLED,TLEU, TLSD,TLSU,TNED,TNEU	source or dest pointer
TRNS	table word descriptor or source or dest pointer
TUND,TUNU	source or dest pointer
TWFD,TWFO,TWTD,TWTU	set word descriptor or source or dest pointer
TWOD,TWOU,TWSD,TWSU	source or dest pointer
UPLD,UPLU,UPRD,UPRU	dest pointer (R-*)
UPUD,UPUU	dest pointer
USND,USNU	dest pointer (R-*)
UNLK	IndexedSingleDD
VALC	IndexedWordDD (R-*)

Stack Overflow

Stack-Overflow interrupt is generated when a word is pushed onto the expression stack with the top-of-stack address equal to the stack limit (LOSR).

The following operators and common actions have more stack outputs than inputs, and so can lead to expression-stack growth:

aACCE aINTE DUPL IMKS LT8 LT16 LT48 LVLC MKSN MKST MPCW NAMC ONE RNGT
ROFF RSNR RTFF RTOD SCRS SPLT VALC WATI WHOI ZERO

The following operators can have stack outputs that, while equal in number to the stack arguments, occupy more words:

BCD LOAD LODT NTGD NTTD NXVA SCLF XTND

Some operators have more arguments in restart state than in initial state. The following operators can detect a stack overflow while updating the stack, prior to generating a Paged Array interrupt. The Stack-Overflow interrupt will be generated after the initial one.

ICLD ICRD ICUD ICVD ICVU PKLD PKRD PKUD PACD PACU SISO

When initiated by TEED or TEEU:

ENDF INOP INSC INSG INSU MCHR MFLT MINS MVNU SFDC SFSC SRDC SRSC

Stack-Overflow is not necessarily reported by the operator that causes the stack build-up.

Stack Structure Error

aACCE	new displacement \neg in {1 to $2^{**}16-1$ } (OPTIONAL) or $S+1 - F \neg$ in {1 to $2^{**}14-1$ } (OPTIONAL) (MKST) or SIRW.lexical_link $\neg \geq$ entered MSCW (OPTIONAL)
aINTE	SIRW.lexical_link $\neg \geq$ entered MSCW (OPTIONAL)
aLXCH	For i in levels traversed: Mem[LexLink to level i] $\neg =$ entered MSCW (OPT) or MSCW.lex_level $\neg =$ i (OPTIONAL)
ENTR	$S \leq F$ or Mem[F] $\neg =$ inactive MSCW or SIRW.lexical_link $\neg \geq$ entered MSCW (OPTIONAL) or new displacement \neg in {1 to $2^{**}16-1$ } (OPTIONAL)
EXIT,RETN	Mem[D[LL]] $\neg =$ entered MSCW or Mem[D[LL]+1] $\neg =$ RCW or MSCW.history_link = 0 (OPTIONAL) or HistLink \leq BOSR or Stack[HistLink] $\neg =$ MSCW or RCW.ll $\neg =$ MSCW.ll (OPTIONAL) (First entered MSCW on historical chain)
MKSN,MKST,IMKS	OPTIONAL, not allowed for interrupts: (new F) - (old F) \neg in {1 to $2^{**}14-1$ } or (new F) - BOSR \neg in {0 to $2^{**}16-1$ }
MVST	computed $F \leq$ BOSR or HistLink \leq BOSR or S-BOSR \neg in {1 to $2^{**}16-1$ } (OPTIONAL) or S-F \neg in {1 to $2^{**}14-1$ } (OPTIONAL) or Stack[stack base] $\neg =$ TSCW or Stack [HistLink] $\neg =$ MSCW or MSCW.ll $\neg =$ LL (OPTIONAL) (First entered MSCW on historical chain)
STFF	new displacement \neg in {1 to $2^{**}16-1$ }

Stack Underflow

Stack-Underflow may be generated by any operator that requires stack arguments. If n argument words are required ($n \geq 1$) and the address of the top-of-stack address at operator entry is less than $D[LL]+n+1$, a Stack-Underflow condition exists.

Since most operators require stack arguments, the following lists only those operators that do NOT generate a Stack-Underflow interrupt:

aACCE aCPY aINTE aISX
BRUN DLAY DEXI
EEXI ENDE ENDF ENTR EXIT HALT IDLE INOP INSC INSG
INSU LNMC LT8

LT16 LT48 LVLC MCHR MFLT MINS
MKSN MKST MPCW MVNU NAMC NOOP NVLD ONE PAUS
PUSH ROFF
RSTF RTFF RTOD RUNI SFDC SFSC SRDC SRSC VALC WATI
WHOI ZERO
ZIC

Undefined Operator

All operator encodings that are undefined for the current interpretation mode cause an Undefined Operator interrupt. Defined operators are identified in Appendixes A and B; all operators not identified in these appendixes are undefined.

When MKSN-NAMC optimization is implemented, the interrupt is generated if the operator following MKSN is not NAMC.

The interrupt is generated when the EXPU operator is used to execute an edit operator that requires a source: one of four move operators (MINS, MFLT, MVNU, MCHR) or two skip source operators (SFSC, SRSC).

Unlocking

The Unlocking interrupt is generated by the UNLK operator when the target interlock status is not Locked.

