PRODUCT SPECIFICATION

REV LTR	REVISION ISSUE DATE	APPROVED BY	REVISIONS
D	7/17/78	J. Hale	1-3 Added QSSW to list of Privileged Instructions. Changes for MARK VIII.0 Release

TABLE OF CONTENTS

INTRODUCTION

Every Fortran S-language program has a base register and a limit register. The area between the addresses in these registers is used as data space only. Code segments are stored by the MCP at any available memory location. The data space includes a non-overlayable area, which contains the evaluation stack, and another area containing overlayable data. The overlayable area may contain more than one data segment.

Various parameters required by the MCP and interpreter are stored beyond the limit register in the run structure nucleus.

A typical Fortran program layout in memory is shown in Figure 1-1.

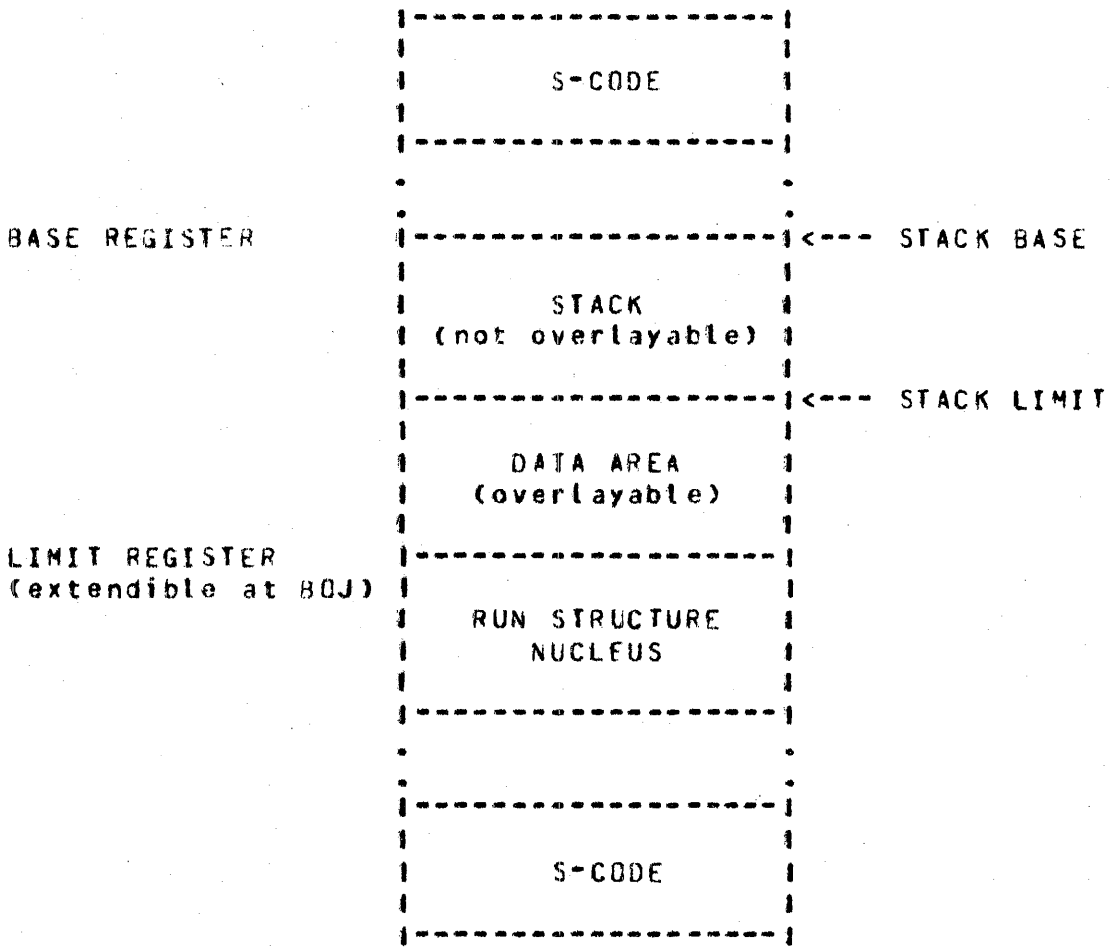


Figure 1-1: Typical Memory Layout

DEFINITIONS AND ABBREVIATIONS

The following definitions and abbreviations are used throughout this specification:

Mark-Stack Word (MSW)	A 36-bit word in the stack containing the 18-bit base-relative value of the previous return control register, concatenated with the base-relative value of the previous mark-stack register.
Mark-Stack Register (MSR)	Contains the stack address of the latest MSW on the stack.
Item	Refers to logical entities on the stack. An item consists of one or two words and has a type (e.g., address, integer, real or double precision) associated with it.
Next-Instruction Pointer (NIP)	Points to the next instruction to be executed.
Return Control Word (RCW)	Contains the segment and displacement of the instruction following the current subprogram call. The value is a 10-bit segment number concatenated with an 18-bit displacement. These 28 bits are right-justified in the 36-bit word.
Return Control Register (RCR)	Points to the stack word containing the latest RCW.
TOP-OF-STACK (TOS)	Represents the item most recently placed on the stack.
TOP-OF-STACK-MINUS-ONE (TOS-1)	Represents the second most recently placed item.
Word	Is 36 bits long and has no intrinsic "logical" meaning.

PRIVILEGED INSTRUCTIONS

The following S-instructions are privileged and may be generated only by an I/O compiler. They are intended for use in writing instrinsics. Data input to them is not necessarily checked for validity; it is the Fortran programmer's responsibility to give valid parameters.

COMM
CSRP
LDB
LDCR
LODB
QDS
QSSW
STC
STOP
WID
XTRF

STRUCTURE

PROGRAM PARAMETERS

Parameters unique to Fortran program execution are to be found in the scratchpad area in the run structure nucleus. They are listed in Table 2-1.

Parameter	Significance
Stack Size	Original size allowed for stack
Stack Length	Current remaining stack length
Stack Address	Current stack address
Mark-Stack Register	Address of latest MSW
Return Control Register	Address of latest RCW

Table 2-1: Unique Fortran Parameters

FORTRAN ADDRESSING

Addressing breaks down into three basic types -- code segments, data segments and a stack -- with items in the stack being code, data or addresses.

Address references to the stack may be:

Implied — Generally TOS and TOS-1 (e.g., ADD, SUBTRACT).

Direct — Positive or negative, relative to the RCW (this type appears in the code) or base-relative (this type appears in the stack itself and refers to items in the stack).

A reference to a data segment address consists of a base and a displacement. The base is an index into the data dictionary and locates the segment containing the desired data. The displacement is a word index into the segment.

If a reference to a code segment address is in a segment other than that containing the desired code segment, the address is a base and a displacement; the base is an index into the code dictionary and the displacement is the bit location relative to the beginning of the segment. If the reference is to the same segment which contains the reference, the address is the displacement from the beginning of the segment.

Code and data segment references use the MCP's virtual memory management to place the segments in memory. The dictionaries record the memory address of each segment.

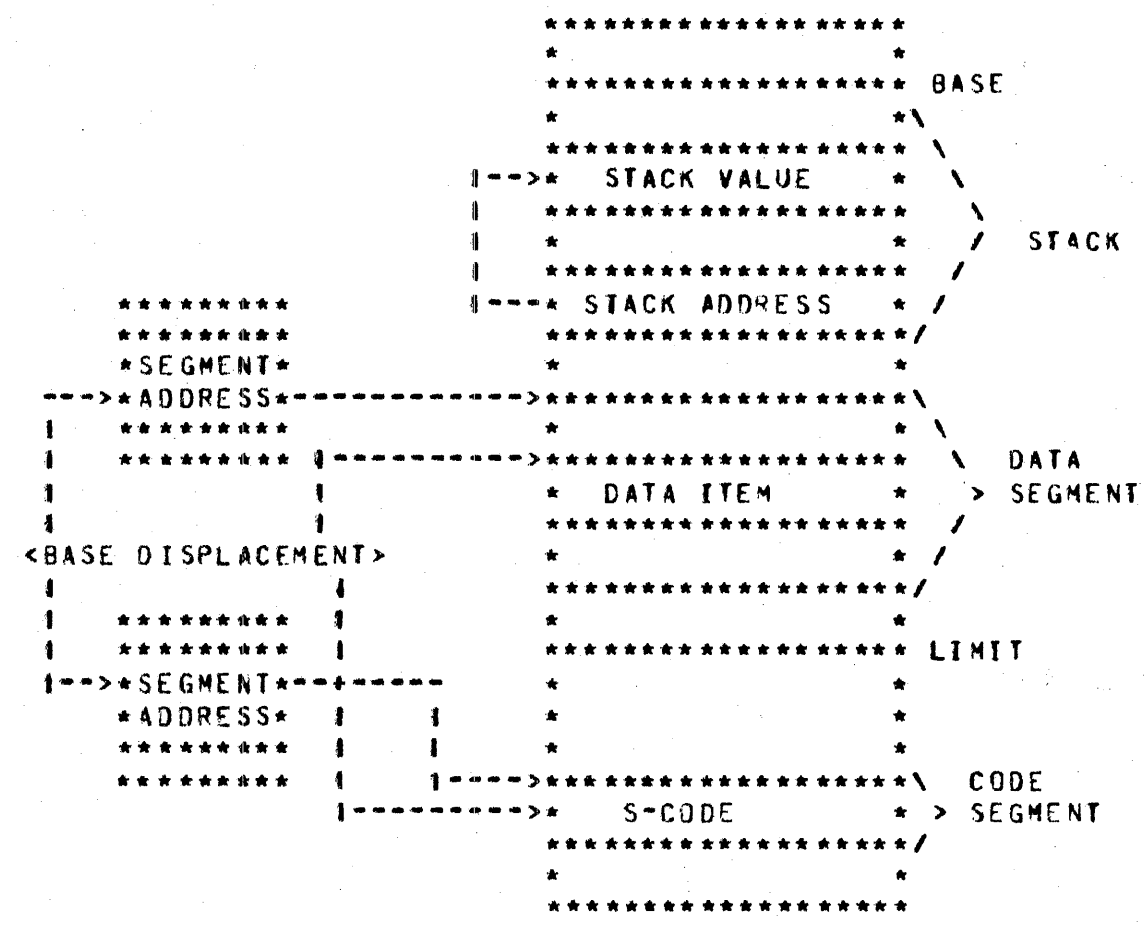


Figure 2-1: Addressing Methods

FORTRAN STACK

The Fortran S-machine includes one evaluation stack, which is used for expression evaluation, subroutine linkage and subroutine parameter passing.

The stack is in static memory but does not necessarily begin at the base register. It grows "up" from the base register end of memory toward the limit register end of memory.

Each word of the stack is 36 bits long. Entries on the stack may be one word long (e.g., integer and real values or simple addresses) or two words (e.g., array addresses or double precision values). Double-word items are pushed to the stack by first pushing the low-order 36-bit word followed by the high-order 36-bit word; this insures that the top word of the stack item contains the type bits which indicate whether a second word is on the stack for that item.

An important consequence of this arrangement is that double-word items appear to be backwards in memory. This is important when reading Fortran dumps produced by the dump analyzer. For example, if the integer 3 is pushed onto the stack, followed by the double precision 1.D+0, then by 1.0, they would logically appear in the stack as:

TOS	501800000	(real 1.0)
TOS-1	001800000	(upper bits of double precision value)
TOS-2	000000000	(lower bits of double precision value)
TOS-3	000000003	(integer 3)

However, a hexadecimal memory dump would show these in the stack as:

000000003 000000000 001800000 501800000

Note that the high-order bit of the double precision value <u>appears</u> in memory to be adjacent to the low-order bit of the bottom 36 bits.

<u>SUBROUTINE LINKAGE</u>

In addition to the stack being used for expression evaluation, it is also used for parameter passing and subprogram linkage information. Invoking a subprogram is a four-part process:

1. Push a mark-stack word (MSW) onto the stack.
2. Load parameters to the stack.
3. Push a return control word (RCW) to the stack.
4. Transfer to the new location.

Two registers are maintained in the S-machine to help a subprogram to return to the program from which it was called. One, the mark-stack register (MSR), is a pointer into the evaluation stack.

It points to the location of the mark-stack word (MSW), (described below) and it points to the place to cut the stack back to when exiting from the current subprogram.

The other return control register (RCR), is a pointer into the evaluation stack where a return control word (RCW) is located. The code address in the RCW tells where control should transfer to when exiting from the current subprogram.

Since there are only one MSR and one RCR in the S-machine, their contents must be saved on each call to a subprogram and restored on each return from a subprogram. Their values are saved in the MSW, which contains two 18-bit, base-relative pointers. The left half is the previous RCR and the right half is the previous MSR.

The return control register, in addition to pointing to an RCW, also serves as a reference pointer in the stack. When a Fortran subprogram references a parameter, the parameter is found by referencing a word in the stack, relative to the current RCR. The compiler, when referencing a parameter, will set the stack bit in the S-instruction which causes the interpreter to reference the Nth word down from the current RCW toward the current MSW (N is found in the address field of the S-instruction). In addition, the compiler can generate code to reference a data item that is N items "up" from the current RCR. This is useful for local data items which need not be maintained over successive calls to the subprogram and for temporary storage of intermediate results when evaluating expressions.

PARAMETER PASSING

Parameters may be passed by value or by address. The value of a parameter is accessed directly or indirectly from the stack by addressing a word relative to the RCR. When loading the value of a parameter, the interpreter checks the type bits of the designated stack item. If the type bits indicate that the item is a value (type bits ¬= binary 10), that value is loaded, with type bits set according to the rules for value loading. If the type bits indicate that the stack item is an address (type bits = binary 10), the data address is decoded, the data segment made present (if necessary) and the desired value loaded.

STACK OVERFLOW-UNDERFLOW MANAGEMENT

If stack overflow occurs, it will cause an abnormal termination. Since the compiler is assumed to generate correct code, stack underflow cannot occur; therefore it is not checked.

ARRAY BOUNDS

A Fortran array reference (ALA, LA, ALV or CAD) will contain a base address, a minimum subscript and a maximum subscript. The value found on the TOS will be the actual subscript and will be checked to insure that it is greater than or equal to the minimum subscript and less than or equal to the maximum subscript values given in the instruction. The minimum and maximum subscripts will be 18-bit numbers in twos complement. The maximum number of array elements will therefore be 2**17 - 1.

The base address will be adjusted by the compiler (binder) so that before being adjusted by the subscript at run time, it may not (and typically will not) reference data in the array. In particular, the subscript will be adjusted backwards by one to allow for the Fortran convention of arrays beginning at element one rather than zero. Further variation may result from optimizing array references of the form A(I+C), where C is some integer constant. The base address of the array will be adjusted by the compiler to allow for the constant C.

If a subscript is real, it is truncated to an integer.

If the type bits in the operator indicate double precision (type bits = binary 11), the subscript is doubled before being used to adjust the base address.

FORMATS

STACK FORMAT

Stack items can be S-code addresses, data addresses or values. With the exception of the mark-stack and return control addresses, the first seven bits of a stack item have meanings as shown in Figure 3-1.

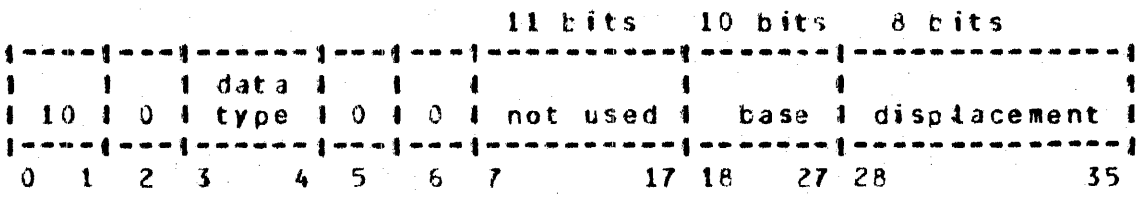
Figure 3-1: Stack Item Prefix

The two-bit stack type flag is 00 for an integer value, 01 for a real value, 10 for an address or 11 for a double precision value. The code flag is 1 only when the item is an S-code address. The two-bit data type flag is meaningful for values only. The array flag is 1 only when the item is an array. The stack-relative flag is meaningful for data addresses only.

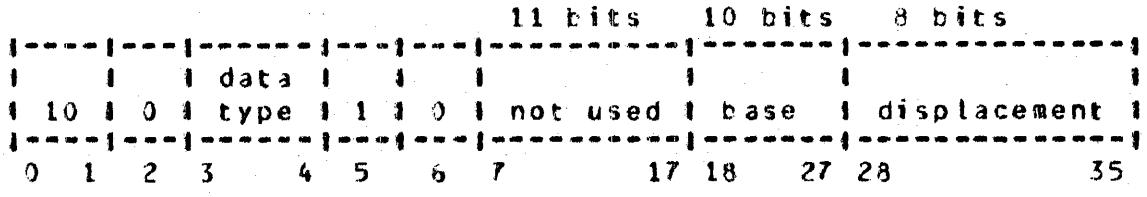
Address Formats

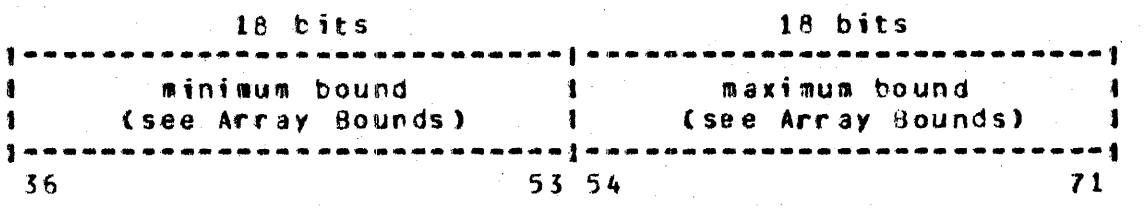
Address-generating operators are LA, ALA, CAD, CSRP, MKS, SMKS, and CALL. The six types of addresses are simple variable addresses, array item addresses, stack-relative addresses, code addresses, mark-stack addresses and return control addresses. Each address occupyies 36 or 72 bits. With the exceptions of mark-stack and return control addresses, the first two bits are 10 and the next five indicate code, type, array and stack-relative values.

Simple Variable Address

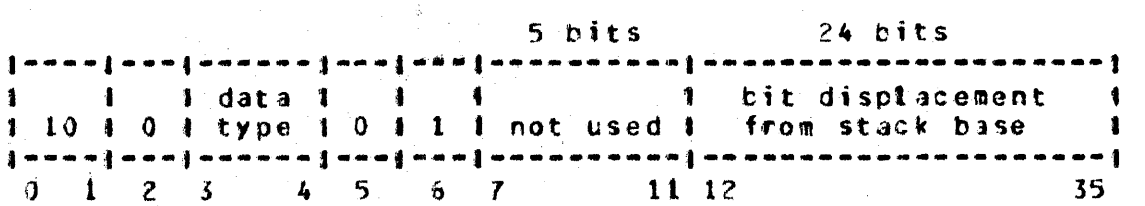


Array Address

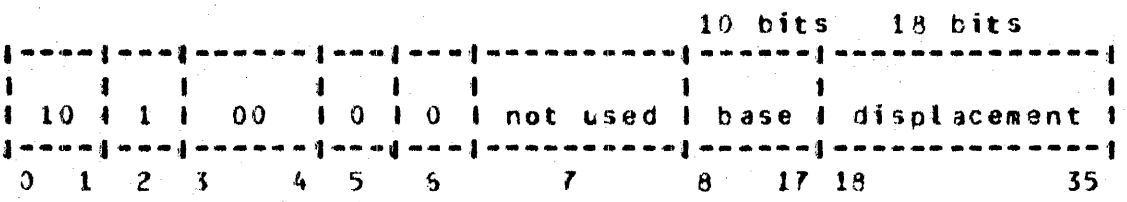




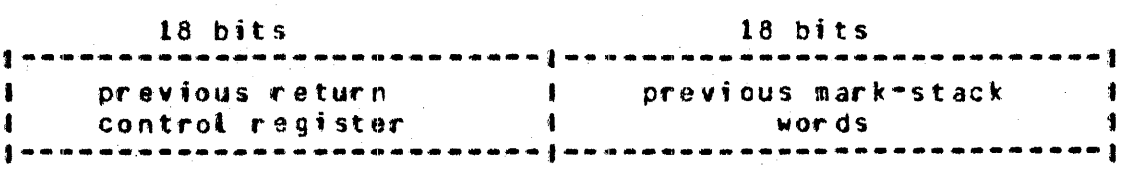
Stack-Relative Address



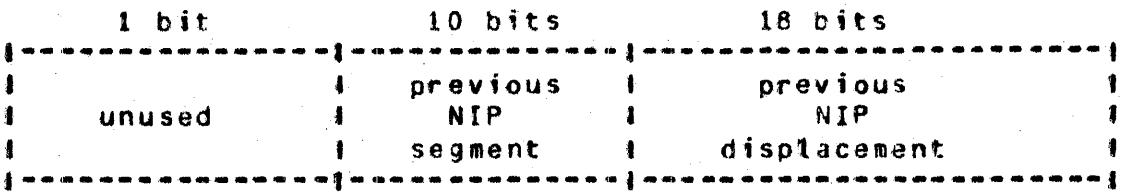
Code Address



Mark-Stack Address



Return Control Address



VALUE FORMATS

Of the four types of value formats, three can be found either on the stack or in a data segment. These three are integer, real and double precision. The fourth type, found only in data segments, is called "uninitialized". Bits 3 and 4, the data type field, are 00 for integer or logical variables, 01 for real variables or 11 for double precision (10 is undefined).

Integers and the fraction portion of non-integers are stored in sign-magnitude notation. Exponents are in "excess 256" notation; that is, an exponent is positive by the amount that it is greater than 256 or negative by the amount that it is less than 256. For example, an exponent of 259 becomes an "excess 256" exponent of +3. The exponent is the power to which two is to be raised. The normalized fraction is a number less than one and greater than or equal to 0.5, except in the case where a fraction equals zero. The fraction part of all nonzero floating point numbers must be

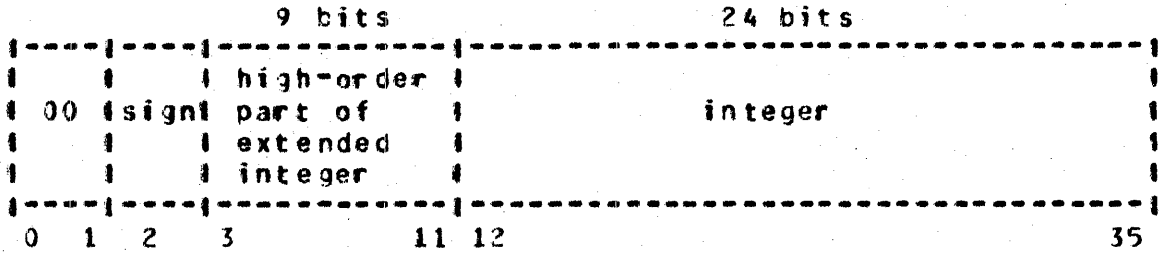
normalized; otherwise, the value is undefined.

Negative numbers are represented with a sign bit of 1, positive with a sign bit of 0. Zero is represented by all zeros except in the type field.

An exponent equal to zero and a nonzero fraction (e.g., N**0.2) will result in an exponent underflow, divide by zero, and/or undefined results as the result of an arithmetic operation. An exponent that is nonzero with a zero fraction is undefined and will cause undefined results in an arithmetic operation.

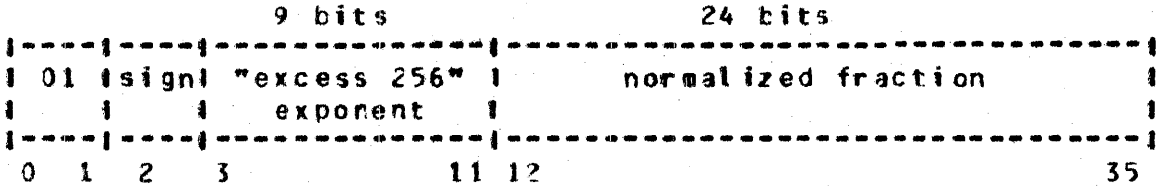
The second word of a double-precision, floating-point number is treated merely as a 36-bit extension to the right of the fraction part of the first word of the number. The first word of a double precision number looks just like a single-precision, floating-point number, except that the type bits are 11.

Integer

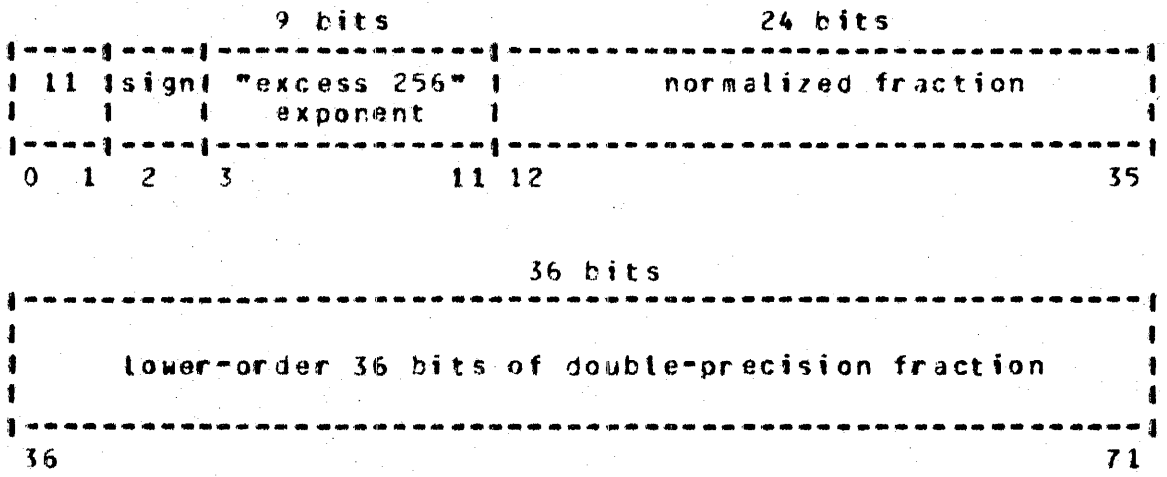


An extended integer is indicated by a nonzero high-order part.

Real



Double Precision



Uninitialized

Uninitialized data have type field 10. The remainder of the format is undefined. Uninitialized data will never be loaded to the stack but will be detected by a "LV" or "ALV" S-instruction.

S-INSTRUCTION FORMATS

An S-instruction consists of an operation code (S-operator) which may be followed by a descriptor and/or other operands. There are two S-operator lengths: three bits for the most frequently used instructions and nine bits for all others. All nine-bit S-operators begin with bits 111. Many S-instructions operate on items from the stack, while others have one or more operands appended to the instruction itself. Detailed information on operands is given in the "S-INSTRUCTIONS" section with descriptions of each affected S-instruction.

OPERAND FORMATS

Table 3-1a lists all possible kinds of operands and the number of bits for each occurrence. The six bits of a descriptor are listed in Table 3-1b.

Operand	No. of Bits
Flag	1
Type indicator	2
Parameter type	4
Descriptor	6
No. of operands	8
Small positive integer or segment no.	10
Code address or array bound (max. or min.)	18
Literal	36

Table 3-1a: Operands List

Meaning	No. of Bits
Code flag	1
Type field	2
Array flag	1
Stack field	1
Unused	1

Table 3-1b: Descriptor Bits

S-INSTRUCTIONS

LISTING BY INSTRUCTION

Instruction Name	Mnemonic	Operation Code
ABSOLUTE VALUE	ABS	111 101 011
ADD	ADD	111 010 110
AND	AND	111 001 001
ARITHMETIC IF	AIF	111 000 110
ARRAY LOAD ADDRESS	ALA	010
ARRAY LOAD VALUE	ALV	011
BRANCH FALSE	BRFL	111 000 101
CHECK TYPE	CKT	111 110 100
COMMUNICATE	COMM	111 001 100
COMPARE PARAMETER ATTRIBUTES	CPA	111 011 011
COMPLEMENT	NOT	111 011 110
COMPUTED GO TO	CGO	111 001 000
CONSTRUCT ARRAY DESCRIPTOR	CAD	111 100 110
CONSTRUCT STACK RELATIVE POINTER	CSRP	111 101 000
DECREMENT-MULTIPLY-ADD PARAMETER	DMAP	111 011 101
DEPRESS THE STACK TO MEMORY	DPRS	111 100 101
DIVIDE	DIV	111 011 001
DOUBLE	DBLE	111 101 010
DS JOB	QDS	111 110 110
DUPLICATE-ONE	DUP1	111 101 100
DUPLICATE-TWO	DUP2	111 101 101
DYNAMIC BRANCH	DBCH	111 110 000
EQUAL	EQ	111 010 000
EXCHANGE	XCH	111 100 001
EXTRACT FIELD	XTRF	111 011 111
FIX	FIX	111 101 001
FLOAT	FLOT	111 000 000
GO TO	GOTO	111 000 111
GREATER THAN	GT	111 010 011
GREATER THAN OR EQUAL	GE	111 010 101
HARDWARE MONITOR	HMON	111 100 100

(Table 4-1 is continued on next page)

Instruction Name	Mnemonic	Operation Code
LESS THAN	LT	111 010 010
LESS THAN OR EQUAL	LE	111 010 100
LOAD ADDRESS	LA	000
LOAD SMALL INTEGER	LSI	110
LOAD BITS	LODB	111 011 010
LOAD BYTE	LDB	111 001 110
LOAD COMMUNICATE REPLY	LDCR	111 100 000
LOAD SINGLE PRECISION LITERAL	LIT	101
LOAD VALUE	LV	001
MAKE TWO	MAK2	111 100 111
MARK STACK	MKS	111 000 100
MAXIMUM	MAX	111 101 111
MINIMUM	MIN	111 101 110
MOVE	MOVE	111 110 011
MULTIPLY	MUL	111 011 000
MULTIPLY-LITERAL-ADD	MLA	111 011 100
NEGATE	NEG	111 100 010
NOT EQUAL	NE	111 010 001
OR	OR	111 001 010
RETURN	RTN	111 000 011
RETURN VALUE AND EXIT	RTNV	111 000 010
SENSE SWITCH	SSW	111 110 101
SPECIAL MARK STACK	SMKS	111 100 011
STATEMENT FUNCTION LINKAGE	SF	111 001 011
STORE BITS	STOB	111 001 111
STORE CHARACTERS	STC	111 110 001
STORE DESTRUCTIVE	STD	100
STORE NONDESTRUCTIVE	STND	111 001 101
SUBROUTINE CALL	CALL	111 000 001
SUBTRACT	SUB	111 010 111
WRITE INTEGER DIGITS	WID	111 110 010

Table 4-1: S-Instructions Listed by Instruction Name

LISTING BY MNEMONIC

Instruction Name	Mnemonic	Operation Code
ABSOLUTE VALUE	ABS	111 101 011
ADD	ADD	111 010 110
ARITHMETIC IF	AIF	111 000 110
ARRAY LOAD ADDRESS	ALA	010
ARRAY LOAD VALUE	ALV	011
AND	AND	111 001 001
BRANCH FALSE	BRFL	111 000 101
CONSTRUCT ARRAY DESCRIPTOR	CAD	111 100 110
SUBROUTINE CALL	CALL	111 000 001
CHECK TYPE	CKT	111 110 100
COMMUNICATE	COMM	111 001 100
COMPUTED GO TO	CGO	111 001 000
COMPARE PARAMETER ATTRIBUTES	CPA	111 011 011
CONSTRUCT STACK RELATIVE POINTER	CSRP	111 101 000
DYNAMIC BRANCH	DBCH	111 110 000
DOUBLE	DBLE	111 101 010
DIVIDE	DIV	111 011 001
DECREMENT-MULTIPLY-ADD PARAMETER	DMAP	111 011 101
DEPRESS THE STACK TO MEMORY	DPRS	111 100 101
DUPLICATE-ONE	DUP1	111 101 100
DUPLICATE-TWO	DUP2	111 101 101
EQUAL	EQ	111 010 000
FIX	FIX	111 101 001
FLOAT	FLOT	111 000 000
GREATER THAN OR EQUAL	GE	111 010 101
GO TO	GOTO	111 000 111

(Table 4-2 is continued on the next page)

Instruction Name	Mnemonic	Operation Code
GREATER THAN	GT	111 010 011
HARDWARE MONITOR	HMON	111 100 100
LOAD ADDRESS	LA	000
LOAD BYTE	LDB	111 001 110
LOAD COMMUNICATE REPLY	LDCR	111 100 000
LESS THAN OR EQUAL	LE	111 010 100
LOAD SINGLE PRECISION LITERAL	LIT	101
LOAD SMALL INTEGER	LSI	110
LESS THAN	LT	111 010 010
LOAD VALUE	LV	001
MAKE TWO	MAK2	111 100 111
MAXIMUM	MAX	111 101 111
MINIMUM	MIN	111 101 110
MARK STACK	MKS	111 000 100
MULTIPLY-LITERAL-ADD	MLA	111 011 100
MOVE	MOVE	111 110 011
MULTIPLY	MUL	111 011 000
NEGATE	NEG	111 100 010
COMPLEMENT	NOT	111 011 110
OR	OR	111 001 010
OS JOB	QOS	111 110 110
RETURN	RTN	111 000 011
RETURN VALUE AND EXIT	RTNV	111 000 010
STATEMENT FUNCTION LINKAGE	SF	111 001 011
SENSE SWITCH	SSW	111 110 101
STORE CHARACTERS	STC	111 110 001
STORE DESTRUCTIVE	STD	100
STORE BITS	STOB	111 001 111
SUBTRACT	SUB	111 010 111
WRITE INTEGER DIGITS	WID	111 110 010
EXCHANGE	XCH	111 100 001
EXTRACT FIELD	XTRF	111 011 111

Table 4-2: S*Instructions Listed by Mnemonics

LISTING BY OPERATION-CODE VALUE

Three-Bit Operation Codes

Octal Value	Binary Value	Mnemonic	Instruction Name
0	000	LA	LOAD ADDRESS
1	001	LV	LOAD VALUE
2	010	ALA	ARRAY LOAD ADDRESS
3	011	ALV	ARRAY LOAD VALUE
4	100	STD	STORE DESTRUCTIVE
5	101	LIT	SINGLE PRECISION LITERAL
6	110	LSI	INTEGER LITERAL

Table 4-3a: S-Instructions Listed by Op Codes (Three Bits)

Nine-Bit Operation Codes

Octal Value	Binary Value	Mnemonic	Instruction Name
700	111 000 000	FLOT	FLOAT
701	111 000 001	CALL	SUBROUTINE CALL
702	111 000 010	RTNV	RETURN VALUE AND EXIT
703	111 000 011	RTN	RETURN
704	111 000 100	MKS	MARK STACK
705	111 000 101	BRFL	BRANCH FALSE
706	111 000 110	AIF	ARITHMETIC IF
707	111 000 111	GOTO	GO TO
710	111 001 000	CGO	COMPUTED GO TO
711	111 001 001	AND	AND
712	111 001 010	OR	OR
713	111 001 011	SF	STATEMENT FUNCTION LINKAGE
714	111 001 100	COMM	COMMUNICATE
715	111 001 101	STND	STORE NONDESTRUCTIVE
716	111 001 110	LDB	LOAD BYTE
717	111 001 111	STOB	STORE BITS
720	111 010 000	EQ	EQUAL
721	111 010 001	NE	NOT EQUAL

(Table 4-3b is continued on next page)

Octal Value	Binary Value	Mnemonic	Instruction Name
722	111 010 010	LT	LESS THAN
723	111 010 011	GT	GREATER THAN
724	111 010 100	LE	LESS THAN OR EQUAL
725	111 010 101	GE	GREATER THAN OR EQUAL
726	111 010 100	ADD	ADD
727	111 010 111	SUB	SUBTRACT
730	111 011 000	MUL	MULTIPLY
731	111 011 001	DIV	DIVIDE
732	111 011 010	LODB	LOAD BITS
733	111 011 011	CPA	COMPARE PARAMETER ATTRIBUTES
734	111 011 100	MLA	MULTIPLY-LITERAL-ADD
735	111 011 101	DMAP	DECREMENT-MULTIPLY-ADD-PARAMETER
736	111 011 110	NOT	COMPLEMENT
737	111 011 111	XTRF	EXTRACT FIELD
740	111 100 000	LDCR	LOAD COMMUNICATE REPLY
741	111 100 001	XCH	EXCHANGE
742	111 100 010	NEG	NEGATE
743	111 100 011	SMKS	SPECIAL MARK STACK CALL
744	111 100 100	HMON	HARDWARE MONITOR
745	111 100 101	DPRS	DEPRESS THE STACK TO MEMORY
746	111 100 110	CAD	CONSTRUCT ARRAY DESCRIPTOR
747	111 100 111	MAK2	MAKE TWO
750	111 101 000	CSRP	CONSTRUCT STACK RELATIVE POINTER
751	111 101 001	FIX	FIX
752	111 101 010	DBLE	DOUBLE
753	111 101 011	ABS	ABSOLUTE VALUE
754	111 101 100	DUP1	DUPLICATE-ONE
755	111 101 101	DUP2	DUPLICATE-TWO
756	111 101 110	MIN	MINIMUM
757	111 101 111	MAX	MAXIMUM
760	111 110 000	DBCH	DYNAMIC BRANCH
761	111 110 001	STC	STORE CHARACTER
762	111 110 010	WID	WRITE INTEGER DIGITS
763	111 110 011	MOVE	MOVE
764	111 110 100	CKT	CHECK-TYPE
765	111 110 101	SSW	SENSE SWITCH
766	111 110 110	QDS	DS JOB

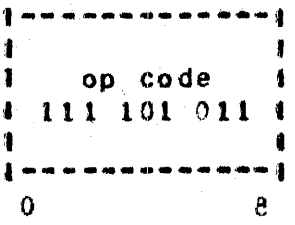
Table 4-3b: S-Instructions Listed by Op Codes (Nine Bits)

DESCRIPTIONS

Arithmetic Operators

ABSOLUTE VALUE (ABS)

Format:



ABS turns off the sign bit for the value on the top of the stack.

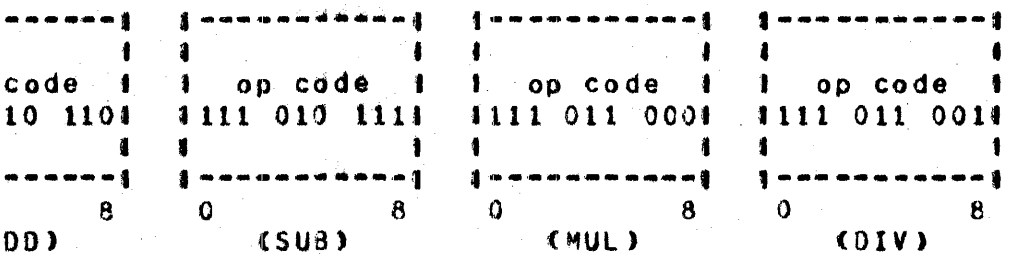
ADD (ADD)

SUBTRACT (SUB)

MULTIPLY (MUL)

DIVIDE (DIV)

Format:



Each of these four functions pops two items off the stack and performs the indicated operation. The mode of the operands may be mixed, with the results described in Table 4-4. The result of the operation is pushed onto the stack. SUB subtracts TOS from TOS-1. DIV divides TOS into TOS-1. Attempting to divide by zero causes abnormal termination.

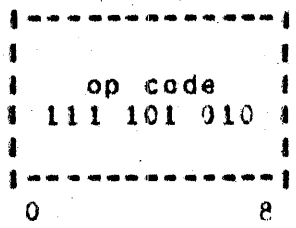
	second operand type		
first operand type	integer	real	double precision
integer	integer*	real	double precision
real	real	real	double precision
double precision	double prec.	double prec.	double precision

*integer division yields a truncated result.

Table 4-4: Modes of Arithmetic Results

DOUBLE (DBLE)

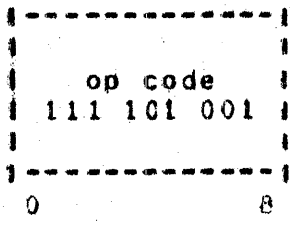
Format:



DBLE converts the item at TOS from integer or real to double precision and replaces the original item with the converted item.

FIX (FIX)

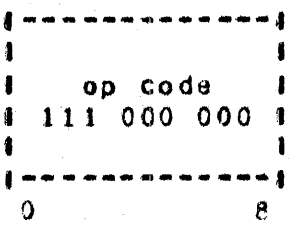
Format:



FIX replaces the item at TOS with a (possibly extended) integer value. The fraction part of a real or double precision item is truncated, not rounded.

FLOAT (FLOAT)

Format:

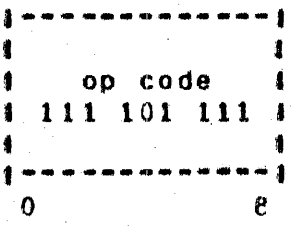


FLOAT examines the type field of the item at TOS. If it is real, the item is left unchanged. If an integer, the item is converted to real; some loss of precision is possible if the integer was an extended integer, although rounding will be performed after normalization.

If the item is double precision, it is rounded and truncated to real.

MAXIMUM (MAX)

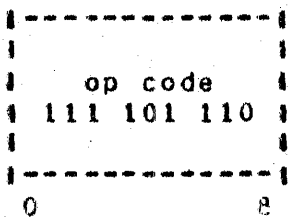
Format:



MAX pops the top two items off the stack and pushes the greater of them back onto the stack. The two items need not be of the same type.

MINIMUM (MIN)

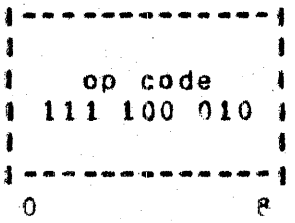
Format:



MIN pops the top two items off the stack and pushes the minimum of them back onto the stack. The types of the two items need not be the same.

NEGATE (NEG)

Format:

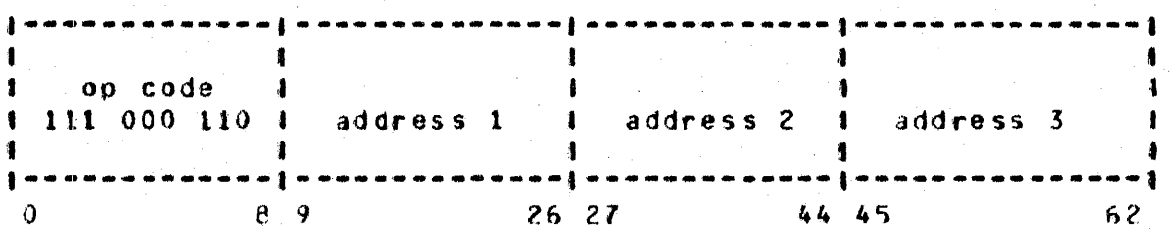


NEG complements the sign bit of the top item on the stack.

Branching Operators

ARITHMETIC IF (AIF)

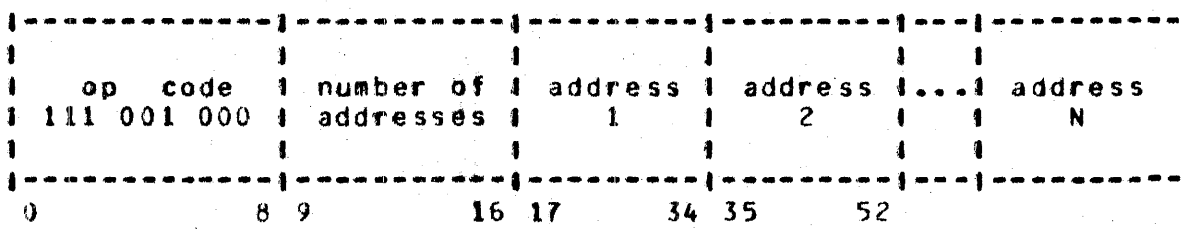
Format:



AIF pops the stack. If the popped word is negative, the first code address is added to the current-segment base and this location is used to fetch the next executable S-instruction. If the popped word is zero, the second address is used. If positive, the third address is used.

COMPUTED GO TO (CGO)

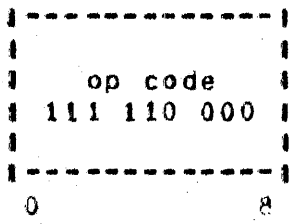
Format:



The CGO instruction expects an integer on TOS, which it compares with the number of addresses following the operation code. If the stack integer is greater than that number or less than one, control transfers to the instruction following the CGO instruction. Otherwise, the integer is used as an index into the address list in the CGO instruction (note that indexing starts with 1, not 0). The address in the list is a segment-relative address which becomes the next instruction pointer. A maximum of 255 addresses may be specified. If the TOS is not an integer, then CGO forces it to integer.

DYNAMIC BRANCH (DBCH)

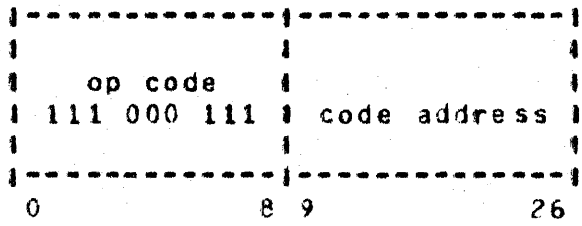
Format:



DBCH expects a code address. The address is popped and execution continues at TOS. If the segment field is zero, the address is a bit displacement relative to the start of the current code segment.

GOTO (GOTO <CODE ADDRESS>)

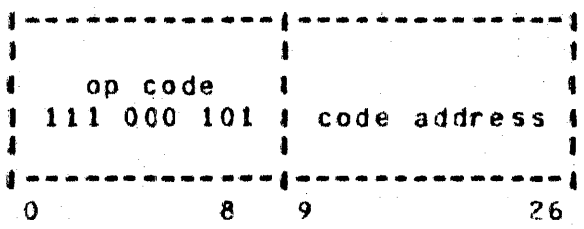
Format:



GOTO obtains the program base-relative code address, converts it to an absolute address and places it in the next instruction pointer.

BRANCH FALSE (BRFL)

Format:

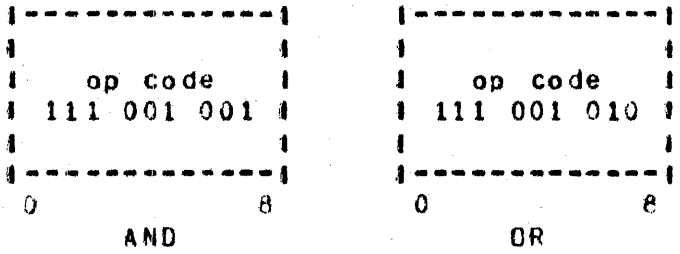


BRFL pops the stack and examines the least significant bit of the value. If the bit is zero, the next instruction pointer is set to the absolute address resulting from conversion of the program base-relative code address passed in the instruction. Otherwise, the next instruction pointer is increased by the length of the BRFL instruction.

Logical Operators

AND (AND) and OR (OR)

Format:

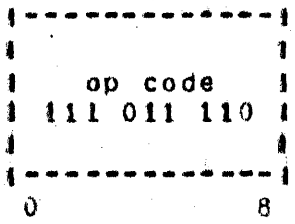


Either of these operations pops two value operands from the stack. If the operands are both single-word length or are both double-word length, the indicated operation is performed on all bits except the type bits of both operands. The type is set to double precision if either operand is double precision, to real if either is real, otherwise to integer.

If one operand is single-word and the other is double-word, the single-word operand is assumed to have an all-zero second word. A bitwise "logical and" or "logical or" is then performed on the two operands and the result is pushed onto the stack.

COMPLEMENT (NOT)

Format:



NOT pops the stack, checks the type and performs ones complement on all bits indicated, except for the type itself (e.g., if the descriptor indicates double precision, 72 bits minus the descriptor are complemented). The result is pushed onto the stack.

Relational Operators

EQUAL (EQ)

NOT EQUAL (NE)

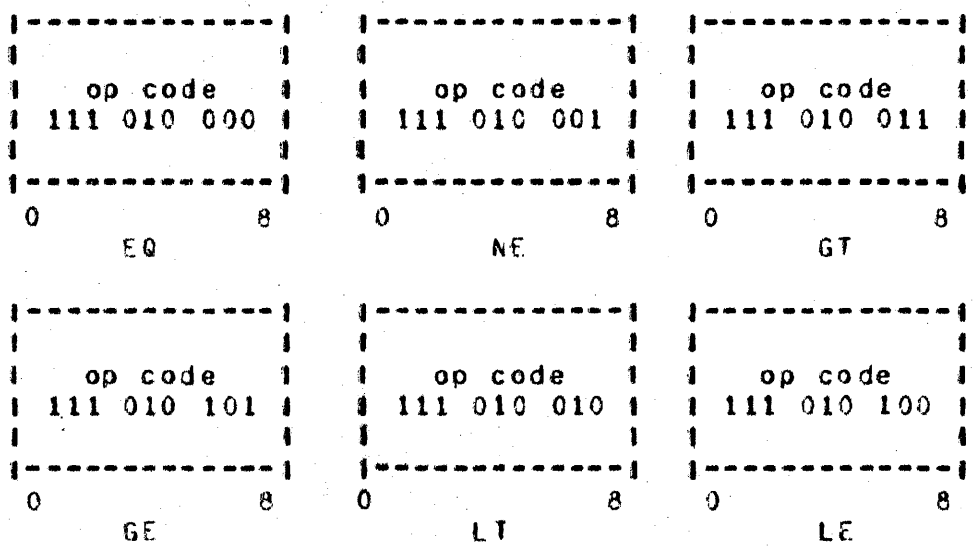
GREATER THAN (GT)

GREATER THAN OR EQUAL (GE)

LESS THAN (LT)

LESS THAN OR EQUAL (LE)

Format:



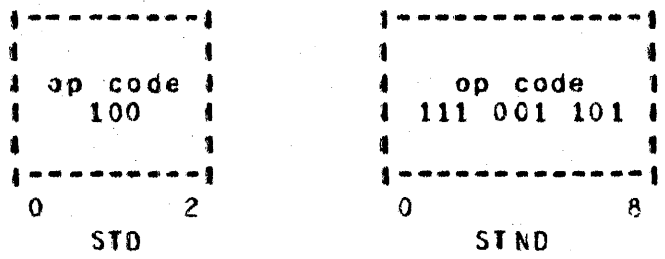
Each of these comparators pops two items off the stack and performs the indicated comparison. If the result is true, an integer 1 is pushed onto the stack, otherwise an integer 0 is pushed. The operation is performed as <TOS-1><relation><TOS>.

Store Operators

STORE DESTRUCTIVE (STD)

STORE NONDESTRUCTIVE (STND)

Format:



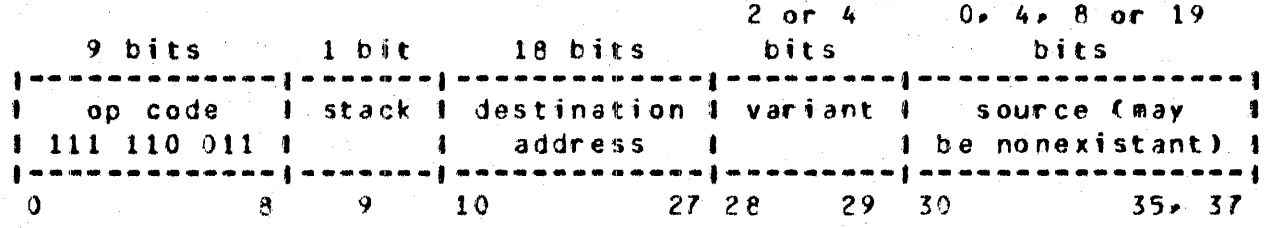
Both STD and STND expect the top word of the stack to be an address and the word below to be a value. The value is stored at the address with restrictions placed by the descriptor, according to Table 4-5. The address is popped from the evaluation stack. With STD only, the value is also popped.

Address	Value	Rule
Integer	Integer	Assign
Integer	Real	Truncate to an integer and assign
Integer	Double Precision	Truncate to an integer and assign
Real	Integer	Float and assign
Real	Real	Assign
Real	Double Precision	Round to 24 bits and assign most significant part
Double Prec.	Integer	Extend to double prec. and assign
Double Prec.	Real	Extend to double prec. and assign
Double Prec.	Double Precision	Assign

Table 4-5: Assignment Rules for STD and STND

MOVE (MOVE)

Format:



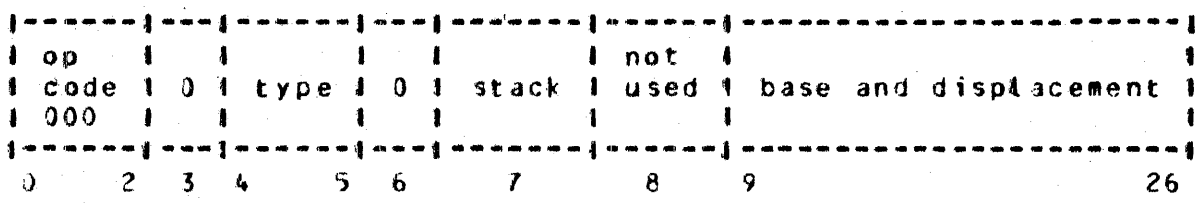
Bits 28 and 29	Bits 30 and 31	Interpretation
00	00	no source field; source implied to be 0
00	01	no source field; source implied to be 1
00	10	source field is four-bit literal in bits 32...35
00	11	undefined
01	--	no source field; source implied to be TOS
10	--	source field is 19 bits; bit 30 is a stack Boolean and bits 31...48 are a segment-displacement address
11	--	source field is bits 30...37; it is a word displacement in the same segment as the destination address

Type conversion is never performed. The type field in the destination will be set the same as in the source. For variant 00, type is set to integer in the destination.

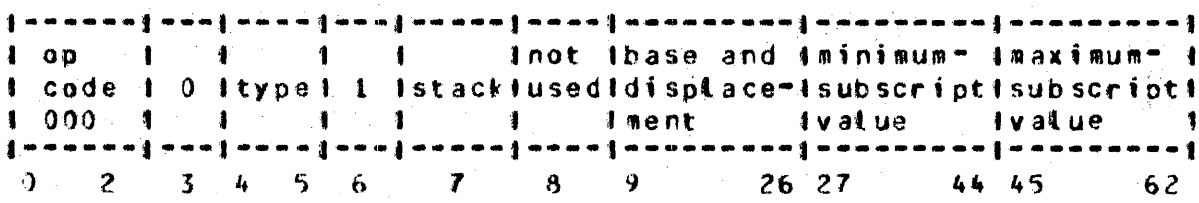
Load Operators

LOAD ADDRESS (LA)

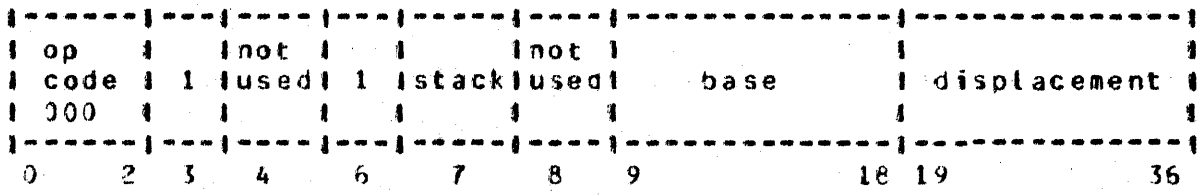
Format 1:



Format 2:



Format 3:



If the stack bit is zero, then the descriptor bits, base and displacement are used in creating either a simple variable address (format 1), array address (format 2), or code address (format 3) at TOS.

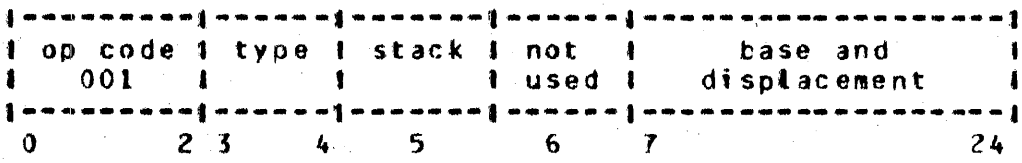
If the stack bit is 1, the base and displacement form a word index relative to the return control register. The index is positive if the leftmost bit of the base is zero and negative if one. The index is multiplied by 36 and added to the RCR to form a pointer into the stack. If the item pointed to is an address, it is copied into TOS. If it is a value, a stack-relative

address using the pointer is placed at TOS.

In either case, the TOS pointer is bumped appropriately.

LOAD VALUE (LV)

Format:



If the stack bit is 0, LV fetches the value at the indicated base-displacement address and puts it on the stack. If the type field is 10, LV loads the value with the type field as it appears in memory; otherwise, it forces the type field equal to that in the S-instruction.

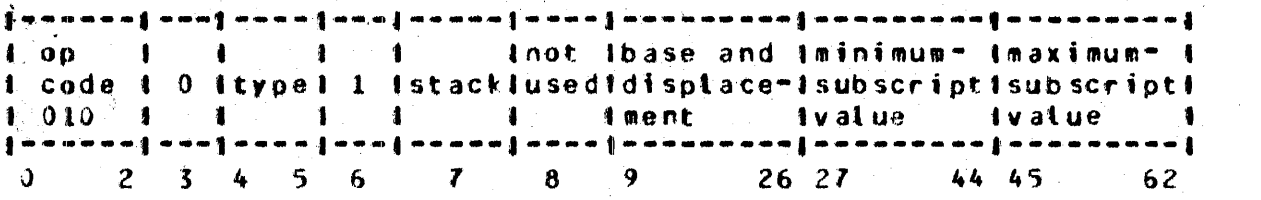
If the stack bit is 1, the low-order 17 bits of the base and displacement are taken as an offset from the current return control register. This value is multiplied by 36. If the high-order bit is set, this multiplication result is then subtracted from (otherwise it is added to) the current RCR. If the item thus located in the stack is a value, then LV loads it to the top of the stack. If the item found was an address, LV uses the found address to locate the data value and loads the found value to the top of the stack. If the type field in the S-instruction is 10, LV loads the value with that type field as found in the value; otherwise, it sets the type field equal to that in the S-instruction.

If the type field in the value does match the type field in the S-instruction, conversion is not performed although the type field is modified as described above.

The program is terminated with "uninitialized data" if the type field on the value is 10.

ARRAY LOAD ADDRESS (ALA)

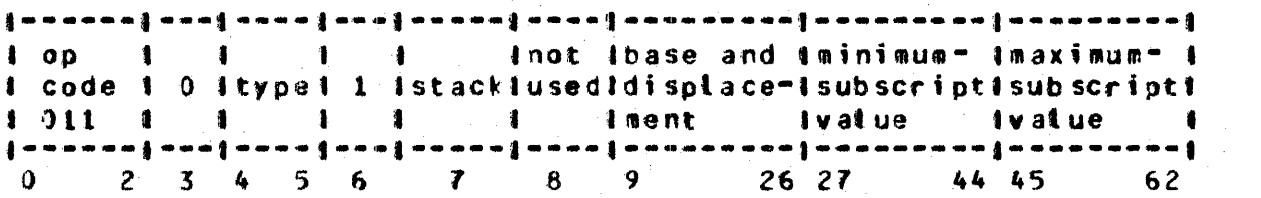
Format:



ALA places a simple variable address at TOS. The address is formed as for LA, except that TOS has a subscript used in modifying the base address as described in Section 1. If the stack bit is 1, the minimum and maximum bounds do not appear in the code but appear as part of the address in the stack which is pointed to by the base and displacement.

ARRAY LOAD VALUE (ALV)

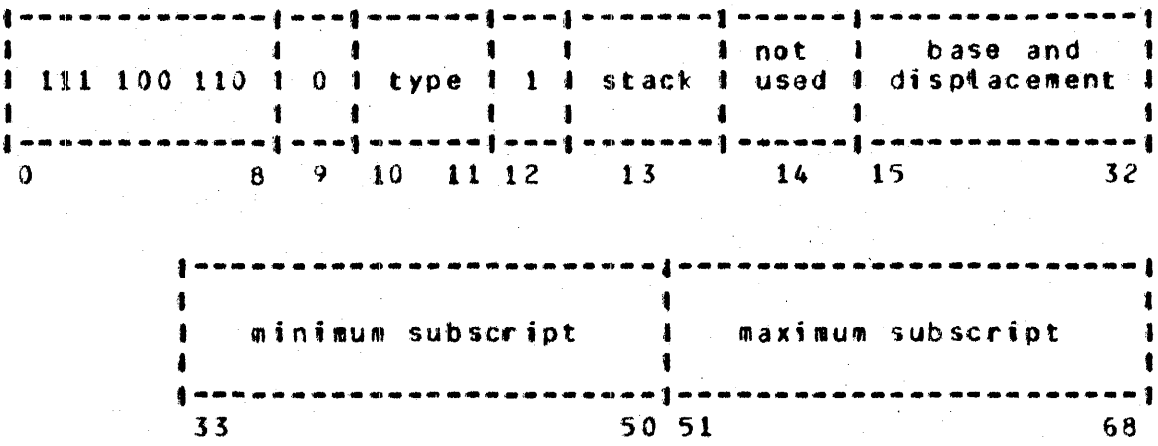
Format:



ALV places a value at TOS by performing an ALA but placing the addressed value at TOS instead of the address. The type field of the loaded value is set to the type specified in the operator, unless the S-instruction's type field is 10; in this case, it is set like the type field of the data item in memory. The program is terminated with "uninitialized data" if the type field of the item in memory is 10.

CONSTRUCT ARRAY DESCRIPTOR (CAD)

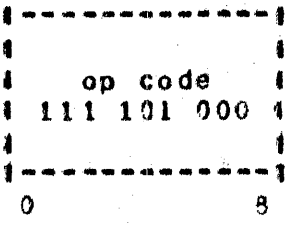
Format:



Using the item on TOS, CAD modifies the base, displacement, minimum and maximum subscript, then places an array descriptor on the stack (after popping the top item). If the stack bit = 1, CAD uses the array descriptor in the stack referenced. In this case, minimum subscript and maximum subscript are not given in the S-instruction. See Section 1 for description of base and array bound formats.

CONSTRUCT STACK RELATIVE POINTER (CSRP)

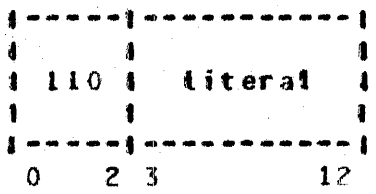
Format:



CSRP uses the TOS as a word pointer to construct a stack-relative address description. The TOS item indicates the number of words from the return control register to point to. A positive index is toward the stack base, a negative index is away from the base.

LOAD SMALL INTEGER (LSI)

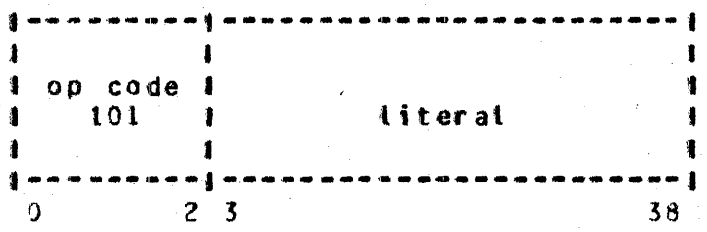
Format:



LSI uses the 10-bit literal as the rightmost 10 bits of the mantissa of a 36-bit integer. The integer formed is assumed positive.

LOAD SINGLE PRECISION LITERAL (LIT)

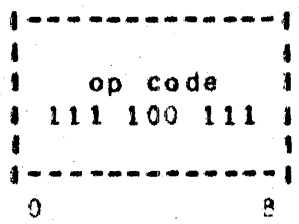
Format:



LIT places the 36-bit literal on TOS and increments the TOS pointer.

MAK TWO (MAK2)

Format:



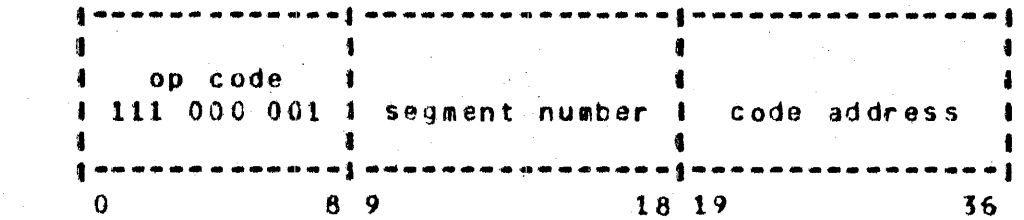
MAK2 makes the item at TOS occupy two words in the stack. If the item at TOS is not a double-word item (i.e. is integer or real value or a non-array address), then MAK2 copies the item forward in the stack and makes its former position in the stack all zeros.

This is used in passing parameters, where the formal parameter may be a scalar (occupying one word in the stack) or an array where the actual parameter was a subscripted variable (could occupy one word) or an array.

Subroutine Call Operators

SUBROUTINE CALL (CALL)

Format:

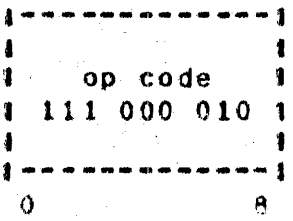


If the segment number is not zero, CALL multiplies the segment number by 80 to get an index into the segment dictionary. If the needed segment is not present, it issues a COMMUNICATE to the MCP to fetch it. When the segment is present, the code address is added to the absolute base address of the new segment, to get the next-instruction pointer. The mark-stack register is then moved to the return control register.

If the segment number is zero, CALL multiplies the code address by 36 and subtracts the product from the return control register to get an index into stack. It then gets the segment number and code address from the stack (see LOAD ADDRESS for the format) and performs the actions described above for a nonzero segment number.

RETURN VALUE AND EXIT (RTNV)

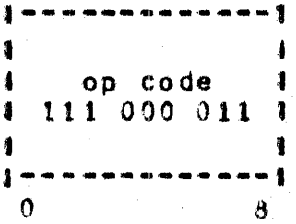
Format:



RTNV pops the TOS and saves the value which is on the TOS. A RETURN is performed and the saved value is pushed onto the stack.

RETURN (RTN)

Format:

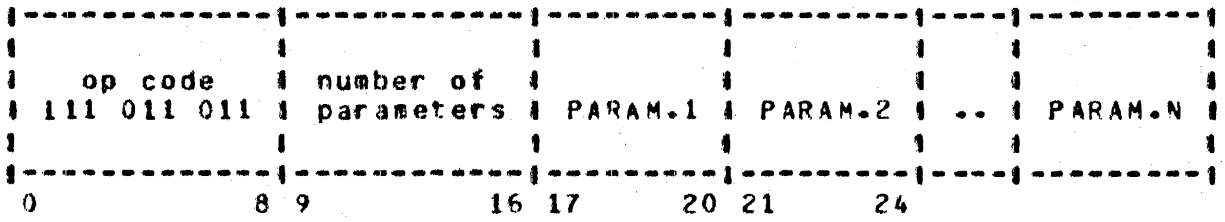


The RETURN instruction uses the return control register to get the segment-relative address from the stack, brings in the proper segment then sets the instruction address register (next-instruction pointer) to the absolute address of the next instruction. The TOS pointer is set to the contents of the mark-stack register, the MSR is loaded with the contents of the rightmost 18 bits of the TOS and the RCR is loaded with the leftmost 18 bits of the TOS.

COMPARE PARAMETER ATTRIBUTES (CPA)

Syntax: CPA <no. of parameters> <PARAM.1> <PARAM.2> ... <PARAM.N>

Format:



The CPA instruction insures that the parameters in the stack agree in number and type with those expected by the called subprogram. Starting with the operand after the mark-stack word, CPA checks the descriptors of all parameters passed on the stack against the types in the CPA instruction. If a non-match occurs or the number of parameters passed in the instruction disagrees with the number of parameters between the mark-stack and return control word, then a run-time error occurs.

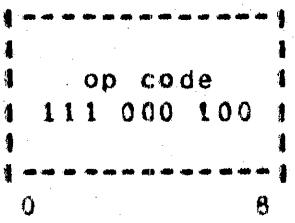
Each parameter is assumed to occupy two words in the stack.

If an operand on the stack begins with a binary 10, (i.e., operand is an address), then the next four bits of the stack operand are compared with the parameter type found in the CPA instruction. If the parameter is an array and the CPA array bit is zero, then the parameters are considered matching and the array bit in the parameter description is set to zero. This facilitates passing subscripted arrays to scalars.

When the code flag (see Figure 3-1) is on in both the CPA and the parameter, special checking occurs. If the type bits in a parameter descriptor are binary 10, the parameter is considered to match; in all other cases, the descriptor bits must match those of the CPA. A maximum of 255 parameter types may be specified.

MARK STACK (MKS)

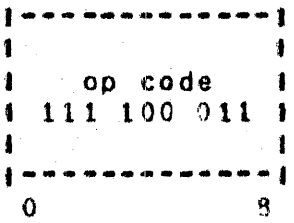
Format:



MKS concatenates the rightmost 18 bits of the return control register with the rightmost 18 bits of the mark-stack register and pushes the result onto the stack. The mark-stack register is set to the TOS pointer and the TOS pointer is incremented.

SPECIAL MARK STACK (SMKS)

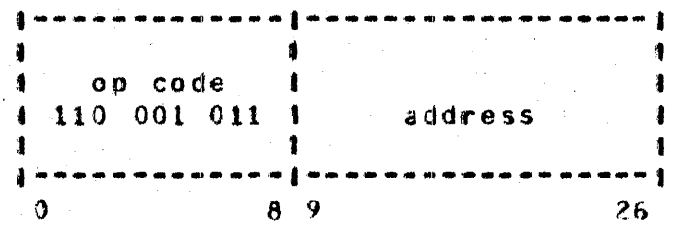
Format:



SMKS is used for invoking exponentiation and complex arithmetic routines. It expects two arguments on the stack. SMKS moves them up and inserts an MKS operation.

STATEMENT FUNCTION LINKAGE (SF)

Format:



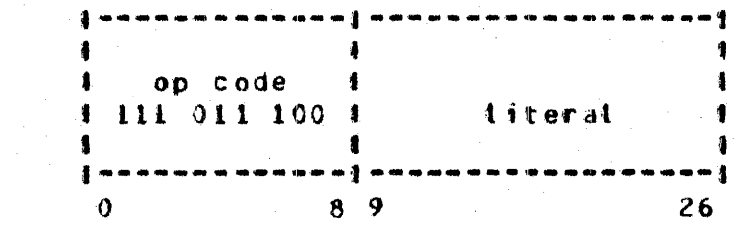
SF sets up linkage to a statement function, then transfers control. The statement function is not external; the code address is a bit displacement from the program base.

This is the same as call, except that the code address is current. The segment address is base-relative.

Miscellaneous

MULTIPLY BY LITERAL AND ADD (MLA)

Format:



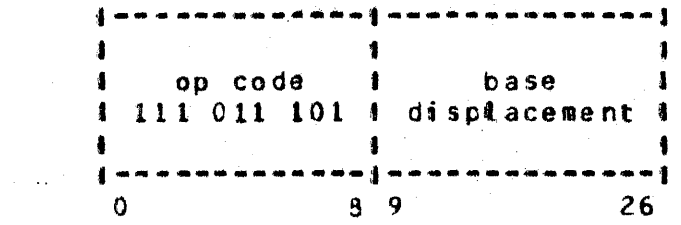
MLA is used to form an index into an array. For an array A dimensioned A(D1, D2, D3,...D) and referred to by A(S1, S2, S3,...S), the general index is calculated by:

(S1 - 1) + D1(S2 - 1) + D1*D2(S3 - 1) +...+ D1*D2*D3...DN(SN - 1).

D1 through DN are emitted as literals following the MLA instruction. The MLA instruction multiplies the element on the TOS by the literal passed in the instruction. The stack is popped and the result created above is added to the new TOS. The total is pushed onto the stack.

DECREMENT-MULTIPLY-ADD PARAMETER (DMAP)

Format:



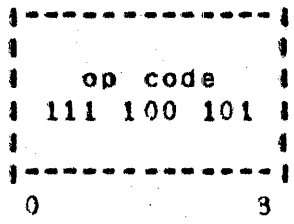
DMAP is a specialized instruction used when variable dimensions have been passed to a subprogram. The cumulative dimension information cannot be given to the interpreter via a literal (as in MLA). This information is calculated and stored relative to the latest RCW.

DMAP decrements (by 1) the value on the TOS and examines the contents of the stack word to which the given base-displacement points. If this word is a base-displacement address of a simple variable, then the value of this variable is placed on the stack. If the word is a value, then it is placed on the stack. The value of the word pointed to by the base-displacement is then multiplied by the decremented variable. The stack is popped and the value on the TOS is added to the current total. This sum is placed on the TOS.

Note: The value loaded by the base-displacement must not be double precision.

DEPRESS THE STACK TO MEMORY (DPRS)

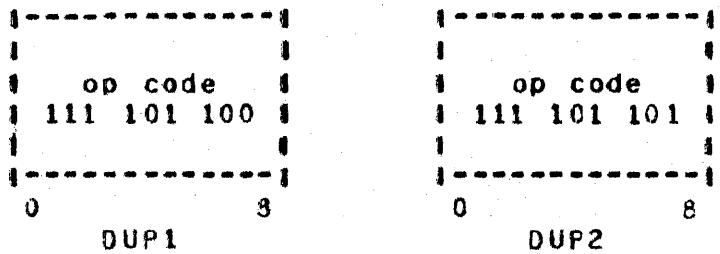
Format:



The DEPRESS operator insures that any occupied TOS registers are pushed down into memory.

DUPLICATE-ONE (DUP1) and DUPLICATE-TWO (DUP2)

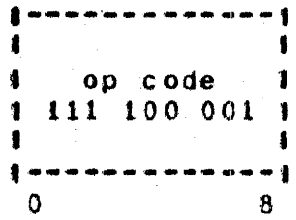
Format:



DUP1 (or DUP2) pushes onto the stack one (or two) value(s) or address descriptor(s) which exactly duplicate(s) the one (or two) item(s) currently at the top. Any combinations of single-word or double-word item(s) may be duplicated.

EXCHANGE (XCH)

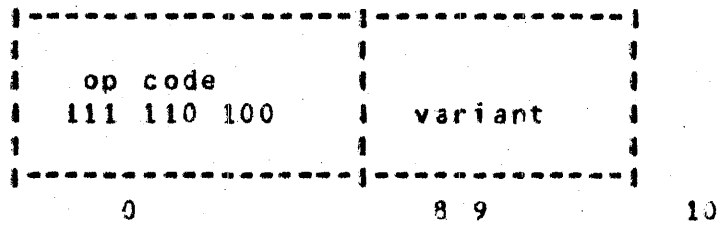
Format:



XCH exchanges the top two items on the stack. These may be single-word or double-word stack items.

CHECK TYPE (CKT)

Format:

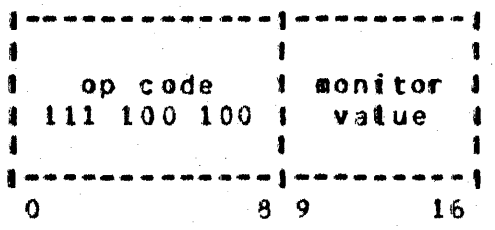


Variant	Type
00	Integer function
01	Real function
10	Subroutine
11	Double precision function

CKT requires that the first single-word item below the RCW be an integer value. The value is popped from the stack and replaced by the RCW. The value is compared with the type in the operator and, if they are not equal, the program is halted and an "INVALID SUBPROGRAM TYPE" message is generated.

HARDWARE MONITOR (HMON)

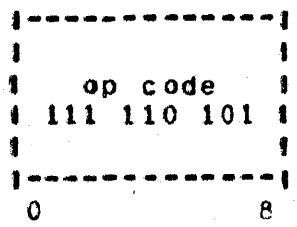
Format:



HMON outputs the eight-bit monitor value to the hardware monitor.

SENSE SWITCH (SSW)

Format:



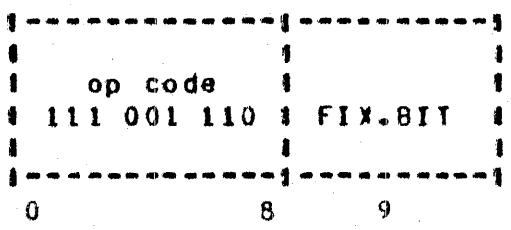
SSW loads program switches 1 through 8 (not 0 or 9) from the run structure nucleus, as a 32-bit literal, and pushes this onto the stack.

DESCRIPTIONS FOR I/O COMPILER ONLY

All instructions in this section are available in the I/O compiler only, not in the standard compiler.

COMMUNICATE (COMM <FIX.BIT>)

Format:

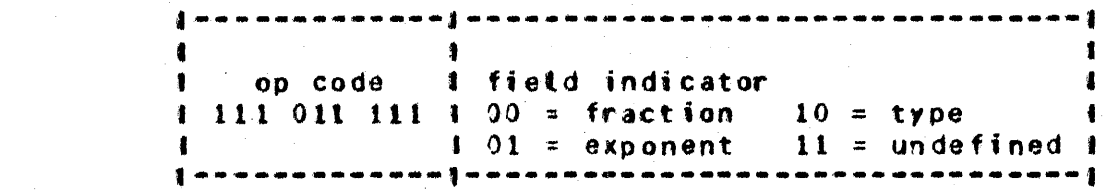


COMMUNICATE loads the 48-bit RS.COMMUNICATE.MSG.PTR area of the run structure nucleus with an SOL-type descriptor indicating a length of 120 bits and the absolute address of the message to be communicated before giving up control to the MCP.

The TOS must be the Fortran data address of the message. The address must be a simple (one-word) address, not an array address descriptor. If FIX.BIT = 1, then the 24-bit field at the message address + 72 bits (CT.2) is a right-justified, 18-bit, segment-relative data address (base followed by displacement). This address must be converted to a base-relative address and the indicated data segment must be made present before control is transferred to the MCP.

EXTRACT FIELD (XTRF)

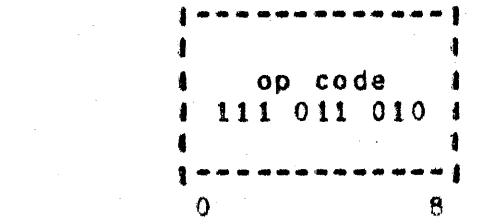
Format:



XTRF extracts either the type, exponent or fraction (from a real value only) field from a single- or double-word item found on the TOS. The resultant value replaces the original value on the stack in integer format. The field to be isolated is indicated by the two-bit value of the field indicator.

LOAD BITS (LODB)

Format:



LODB loads 0 to 33 bits from a data area to the TOS, in the form of an integer. It expects the following items on the stack:

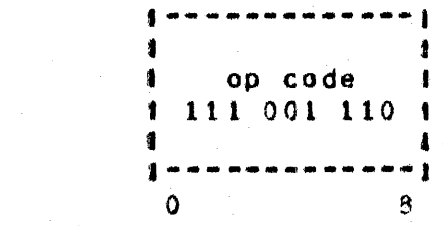
TOS An address which may be an array address.

TOS-1 The address modifier, which is added to the above address as a bit displacement into the field. The modifier may be negative.

TOS-2 The length in bits (must be less than or equal to 33), to be loaded to the evaluation stack.

LOAD BYTE (LDB)

Format:



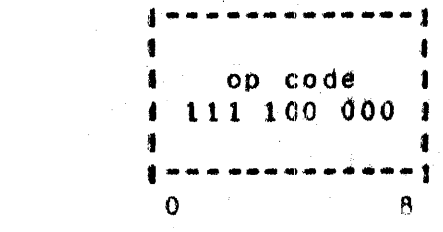
LDB loads eight bits from a data area to the TOS in the form of an integer. It expects the following items on the stack:

TOS — An address which may be an array address. Currently, this address may not be the address of a stack-relative, double-precision item, unless TOS-1 is less than 28.

TOS-1 — The address modifier, which is added to the above address as a bit displacement into the field. The modifier may be negative.

LOAD COMMUNICATE REPLY (LDCR)

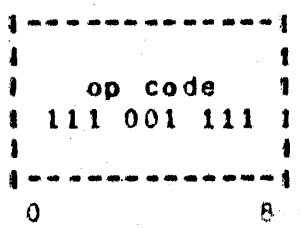
Format:



LDCR loads the low-order 36 bits of the run structure reply area to the TOS.

STORE BITS (STOB)

Format:



STOB expects four parameters on the stack:

TOS — An address (base) of a data area. The address may be an array address which requires bounds checking, but it may not be the address of a stack-relative, double-precision item.

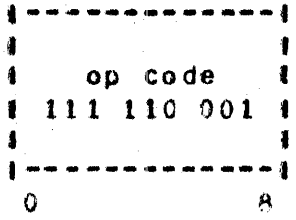
TOS-1 — An integer value which serves as an address modifier in bits.

TOS-2 — An integer value which tells how many bits (from the right) are to be stored.

TOS-3 — A value, part of which is to be stored into the program's data area. The value may be either single or double precision.

STORE CHARACTERS (STC)

Format:



STC expects four items on the stack:

TOS The base address of a storage field.

TOS-1 A bit displacement into the field.

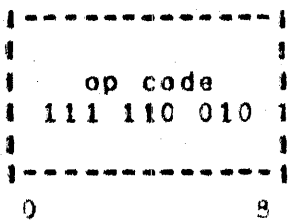
TOS-2 The length (in characters) of the field to be filled.

TOS-3 A single- or double-word item containing the characters to store.

If the length of the field to be filled is greater than the number of characters given, the given characters are sorted cyclically and repeatedly until the field is filled. In a single-word source (four characters), the characters will be left-justified in the low-order 32 bits. In a double-length source word (eight characters), the first four characters will be as in a single-length item and the second four will be left-justified in the low-order 32 bits of the second 36 bits.

WRITE INTEGER DIGITS (WID)

Format:



WID expects four items on the stack:

TOS The base address of a storage field.

TOS-1 A bit displacement into the field from the base.

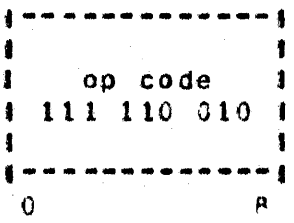
TOS-2 The length (in characters) of the field to be filled.

TOS-3 A (possibly extended) integer.

It converts TOS-3 from binary to EBCDIC display characters and write the characters into the designated field, right-justified, and with zero fill.

DS JOB (QDS)

Format:



QDS expects three items on the stack:

TOS Base-displacement data address pointing to "DS" or "DP" error message.

TOS-1 Message length in bytes

TOS-2 COMMUNICATE verb (always 29)

QDS causes a COMMUNICATE verb to be issued, causing the job to be discontinued and the specified error message to be displayed.

INDEX