DISTRIBUTION LIST

B1800/B1700 SOFTWARE PRODUCT SPECIFICATIONS

DETROIT

J. Peterson - Prod. Mgmt.
H. Woodrow - Int'l Prod Mgmt
C. Kunkelmann - BMG
J. McClintock - CSG
Dir., Pgmg. - SSG
M. Dowers - Corp. TIO
D. Hill - TC, BM, & SS

U.S. AND EUROPE

J. H. Pedersen (Plymouth)
W. E. Feeser (Austin)
J. Berta (Downingtown)
G. Smolnik (Paoli)
M. E. Ryan (Tredyffrin)
T. Yama - F&SSG (McLean)
J. Poterack - F&SSG (McLean)
J. Sallone - F&SSG (McLean)
M. Smith - F&SSG (McLean)
R. Sutton - F&SSG (McLean)
L. DeBartelo - WADC (Irvine)
R. Cole (Pasadena)
H. M. Townsend (Pasadena)
N. Cass - Pat. Atty. (Pasadena)
S. Samman (Mission Viejo)
J. Lowe (Mission Viejo)
J. C. Allan (Glenrothes)
W. McKee (Cumbernauld)
B. Higgins (Livingston)
Mgr, NPSGrp (Ruislip)
E. Norton (Middlesex)
J. Cazanove (Villers)
J. C. Wery (Liege)
R. Bouvier (Liege)
G. LeBlanc (Liege)
C. J. Tooth - SSG (London)
J. Dreystadt (Wayne)
K. Iwasawa - Prod. Manager (Tokyo)
S. Walsh - 2 (TIO Cumbernauld)

SANTA BARBARA PLANT

R. Shobe
R. Bauerle
A. van der Linden - 12
J. Alajoki - 2

Distribution list current as of 4/20/82

2233 2860

Burroughs Corporation

COMPUTER SYSTEMS GROUP
SANTA BARBARA PLANT

PASCAL S-MACHINE

PRODUCT SPECIFICATION

REV LTR	REVISION ISSUE DATE	APPROVED BY	REVISIONS
A	5/11/82	R Smith	Original Issue -- Mark 11.0 Release

Burroughs Corporation
Computer Systems Group
Santa Barbara Plant

TABLE OF CONTENTS

INTRODUCTION 1-1
RELATED DOCUMENTS 1-1
REGISTERS 2-1
DATA STRUCTURES 3-1
MEMORY ORGANIZATION 3-1
STACK FORMAT 3-2
Variables 3-2
Expressions 3-3
Linkage Information 3-3
HEAP FORMAT 3-3
CONSTANT POOL 3-4
CODE SEGMENTATION 4-1
OPERATORS 5-1
LOAD ADDRESS OPERATORS 5-1
Load Constant Address 5-1
Load Local Address 5-2
Load Global Address 5-2
Load Lexic Level Address 5-3
LOAD WORD OPERATORS 5-3
Load Constant Word 5-3
Load Local Word 5-4
Load Global Word 5-4
Load Lexic Level Word 5-5
Load Indirect Word 5-5
Load Unsigned Field 5-6
Load Signed Field 5-6
LOAD NON-WORD OPERATORS 5-6
Push Structure 5-6
Push Real 5-7
Push Real Constant 5-7
Push Set 5-7
ADDRESS MODIFICATION OPERATORS 5-8
Field 5-8
Index 5-8
STORE WORD OPERATORS 5-9
Store Local Word 5-9
Store Global Word 5-9
Store Lexic Level Word 5-10
Store Indirect Word 5-10
STORE NON-WORD OPERATORS 5-10
Store Real 5-10
Store Set 5-11
Store Tag 5-11
Store Unsigned Field 5-12
Store Signed Field 5-12
Copy Structure 5-12
VALIDATION OPERATORS 5-13
Pointer 5-13

Variant 5-13
Range 5-13
NUMERIC COMPUTATION 5-14
Negate Word 5-14
Absolute Value Word 5-14
Successor Word 5-14
Predecessor Word 5-15
Increment Word 5-15
Decrement Word 5-15
Add Word 5-16
Subtract Word 5-16
Multiply Word 5-16
Divide Word 5-17
Modulo Word 5-17
Square Word 5-17
Negate Real 5-18
Absolute Value Real 5-18
Add Real 5-18
Subtract Real 5-19
Multiply Real 5-19
Divide Real 5-19
Square Real 5-20
BOOLEAN COMPUTATION OPERATORS 5-20
Not 5-20
And Word 5-20
Or Word 5-21
SET COMPUTATION OPERATORS 5-21
Union Set 5-21
Intersection Set 5-21
Set Difference 5-22
Build Set 5-22
Build Set Range 5-23
RELATIONAL OPERATORS 5-23
Equal Word 5-23
Not Equal Word 5-24
Less Than Word 5-24
Not Less Than Word 5-24
Greater Than Word 5-25
Not Greater Than Word 5-25
Odd 5-25
Equal Real 5-26
Not Equal Real 5-26
Less Than Real 5-26
Not Less Real 5-27
Greater Than Real 5-27
Not Greater Than Real 5-27
Equal Structure 5-28
Not Equal Stucture 5-28
Less Than Structure 5-28
Not Less Than Structure 5-29
Greater Than Structure 5-29
Not Greater Than Structure 5-29
Equal Set 5-30
Not Equal Set 5-30

Not Less Than Set 5-30
Not Greater Than Set 5-31
In Set 5-31
CONVERSION OPERATORS 5-31
Truncate Real 5-31
Round 5-32
Convert Word to Real 5-32
Convert Word2 to Real 5-32
CONTROL OPERATORS 5-33
Jump 5-33
False Jump 5-33
Case Jump 5-33
Call 5-34
Call Segment 5-34
Enter Procedure 5-35
Exit Procedure 5-35
Exit Function 5-36
Enter Program 5-36
Exit Program 5-36
Call Standard Routine 5-37
Function Value 5-37
MISCELLANEOUS OPERATORS 5-37
Pop 5-37
Push Stack 5-38
New Line 5-38
New 5-38
New Initialized 5-39
Initialize Variables 5-39
Construct File Descriptor 5-39
Text 5-40
STANDARD ROUTINES 6-1
FILE HANDLING ROUTINES 6-1
Get 6-1
Get Text 6-1
Put 6-2
Put Text 6-2
Update 6-2
Position 6-2
End Of File 6-3
End Of Line 6-3
Reset 6-3
Reset Text 6-3
Rewrite 6-4
Rewrite Text 6-4
Read Character 6-4
Read Integer 6-4
Write Character 6-5
Write String 6-5
Write Integer 6-5
Read Line 6-6
Write Line 6-6
Page 6-6
Retain 6-6
Remove 6-7

Reassign 6-7
Relinguish 6-7
Remain 6-7
Write Boolean 6-8
NON-FILE ROUTINES 6-8
Mark 6-8
Release 6-8
Expo 6-9
Clock 6-9
Date 6-9
Time 6-9
Abort 6-10
Communicate 6-10
Reinstate Message 6-10
Display 6-10
APPENDIX A - OPERATORS IN ALPHABETICAL ORDER A-1
APPENDIX B - OPERATORS IN OP CODE ORDER B-1

Burroughs Corporation
Computer Systems Group
Santa Barbara Plant

INTRODUCTION

This document describes the virtual machine used to implement B1000 Pascal. The first sections of the document present the overall structure of the virtual machine. These are followed by a detailed description of each of the operators. The virtual machine is also called the S-machine and the operators S-ops. Here "S" is for soft or software.

The Pascal S-machine is realized on the B1000 through an interpreter. Like SDL, Cobol, and Fortran, Pascal has its own interpreter. The interpreter is written in MIL, the B1000 Micro Implementation Language.

The Pascal virtual machine is a stack machine. Storage is composed of a single stack and a heap plus read-only areas for the virtual code and the constant pool.

The stack and the heap start at opposite ends of memory and grow toward each other. The stack is composed of activation records, one for every procedure which is active. Each activation record consists of a return linkage area, a parameter storage area, a local variable storage area, and an expression evaluation stack. The heap is the collection of cells pointed to by Pascal pointer variables. It is initially empty, but cells may be allocated by the Pascal NEW procedure.

RELATED DOCUMENTS

B1000 Pascal P.S. 2233 2852

Burroughs Corporation
Computer Systems Group
Santa Barbara Plant

REGISTERS

B.R — A scratchpad register which contains the absolute memory address of the base of the memory that is allocated to this program. This scratchpad has the same value as the base register, BR. The value can be used in computations easier in a scratchpad register. A base relative address is an offset from this value.

B — The base relative address of the base of the local variables. B is a copy of the display register associated with the current lexic level.

CONST — The base relative address of the base of the constant pool. CONST is not a scratchpad register. It is a value in memory. It is located at an offset of 144 bits or 6 words from G, the global base register.

CURRENT.LINE — The source line number which corresponds to the code being executed.

DISPLAY — A group of 7 scratchpad registers which are treated as an array of registers. The registers are indexed from 1 to 7 and correspond to the base address of the variables of the active procedures at lexic levels 1 through 7. The G register serves as DISPLAY [0] and the B register is a copy of DISPLAY [LL], where LL is the current lexic level.

G — The base relative address of the base of the global varibles. The G register can be thought of as being the zeroth element of the display array. For a particular program its value does not change like the other display registers, thus it is not treated internally as part of the display array.

HEAPTOP — The base relative address of the next available location at the top of the heap.

LL — The number of the current lexic level. Lexic levels can range from 0 to 7.

NIP — The next instruction pointer. The NIP is an absolute address that points to the location within a code segment where the next instruction is to be read.

SP — The base relative address of the next available location at the top of the stack.

STACK.TOP — Logically the 24 bit item which is the top of the stack. The top of the stack is maintained in a register to cut down on memory accesses. This register is managed entirely by the code generated by the compiler. If STACK.TOP is full and a new item needs to be pushed onto the stack, then the compiler generates code that first pushes STACK.TOP into the memory portion of the stack before loading STACK.TOP with the new value.

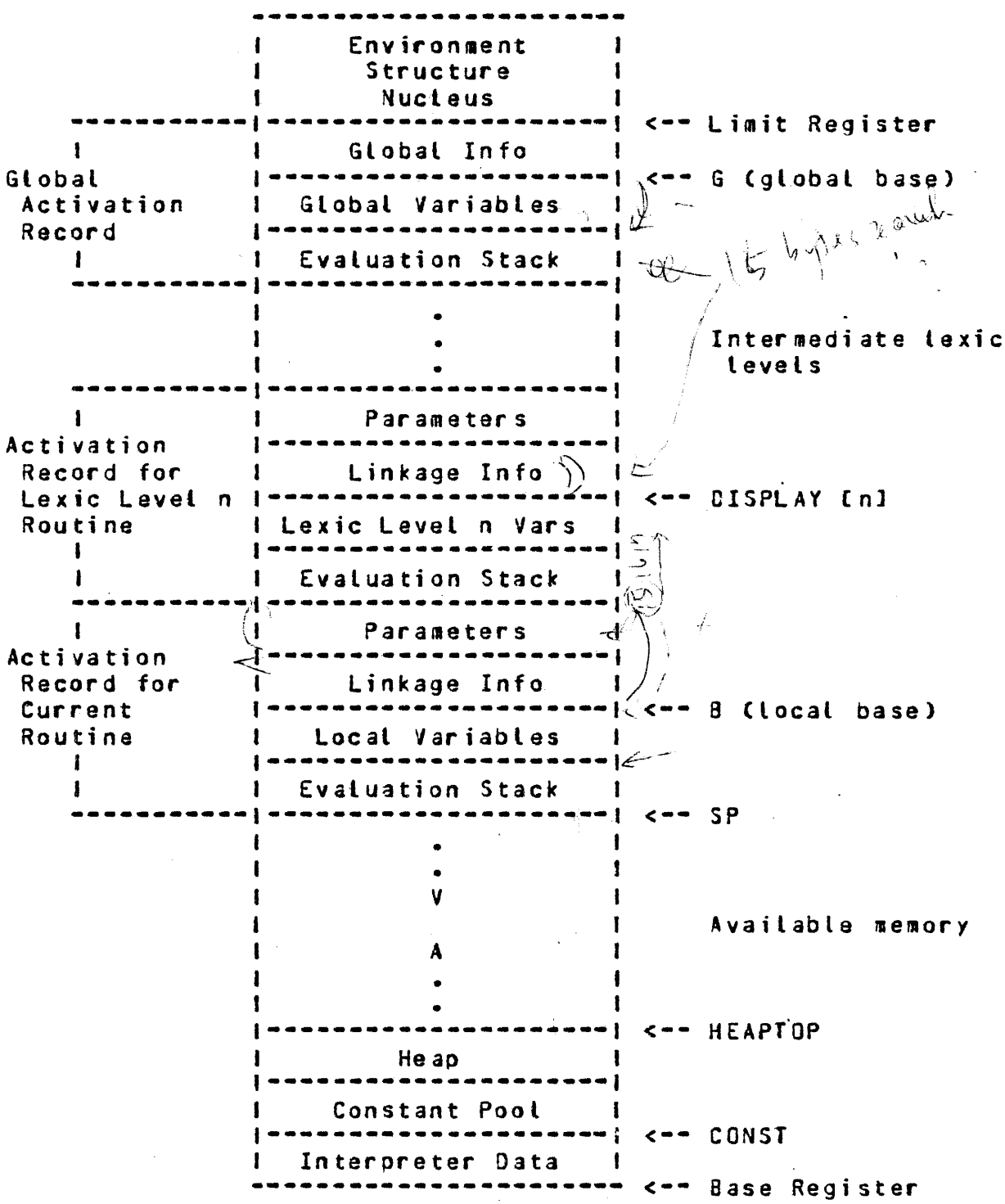
managed entirely

stack.TOP

Burroughs Corporation
Computer Systems Group
Santa Barbara Plant

DATA STRUCTURES

MEMORY ORGANIZAIION



STACK FORMAT

There are basically three different kinds of items that make up the stack. These are data or variables, expressions which includes parameters, and linkage information. The formats for each of these items is described below.

Variables

The different kinds of variables and the amount of memory allocated are listed below.

real	48 bits total	
	14 bits	: filler
	1 bit	: sign (1 is negative)
	9 bits	: binary exponent in excess 256 notation
	24 bits	: normalized mantissa
other scalars	24 bits	: two's complement signed integer value
pointer types	24 bits	: unsigned base relative bit address
set types	256 bits	: i in s means bit (i) = 1
file types	first two fields only present for text files	
	16 bits	: bit offset of last character in buffer
	16 bits	: bit offset of current character in buffer
	16 bits	: bit length of buffer
	1 bit	: defined boolean
	1 bit	: end of file boolean
	6 bits	: file number
	n bits	: file buffer variable (address of file points here)

record and array types
amount of space required rounded up to an integral number of bytes

fields inside packed structures

scalars other than reals	: number of bits required for value if signed then two's complement
sets	: bits = largest value of base type
records and arrays	: exact amount of space required

Expressions

When variables are loaded as expressions onto the evaluation stack the size on the evaluation stack is not necessarily the same as the size of the variable. In particular, any fields that are smaller than 24 bits are loaded as 24 bit quantities onto the stack. The following types of items appear as expressions.

word	: 24 bits	(integers, scalar, and address values)
real	: 48 bits	(floating point values)
field	: 24 bits	(packed integer and scalar values)
set	: 256 bits	(set values, possibly packed)
structure	: rounded up to nearest byte	(array and record values)

Linkage Information

The procedure linkage information is a structure with the following fields. Part of the structure is built by the CALL operator and the rest is built by the ENTER operator.

return displacement	: 24 bits	(CALL)
return segment number	: 16 bits	(CALL)
current lexic level	: 8 bits	(ENTER)
local base (display [ll])	: 24 bits	(ENTER)
stack pointer restore value	: 24 bits	(ENTER)
procedure line number	: 24 bits	(ENTER)

HEAP FORMAT

The heap is a stack-like structure composed of data items of various sizes. The items are pointed to by Pascal pointer types. New items are allocated at the top of the heap by the procedure NEW. The heap is cut back to a previous point by the procedure RELEASE. Previously allocated items that lie above the release point are no longer valid.

CONSTANT POOL

The constant pool is a fixed area of memory within a program's run structure which contains all string and set constants. The contents of the pool are set up as a data segment by the compiler and the operating system initializes the base of memory with this data segment at the beginning of the program. The constants are referenced by offset from the CONST register. Data in the constant pool is never modified.

The first few bytes of the data segment that contains the constants are reserved. This area is used as data space for interpreter variables. The constants start just after this area.

Burroughs Corporation
Computer Systems Group
Santa Barbara Plant

CODE SEGMENTATION

The Pascal object code is broken into segments by the compiler. These segments of code are brought in from disk and placed at available areas in memory by the operating system at the request of the interpreter. The code does not reside between the base and limit registers of a program.

The code in a Pascal code file is always on byte boundaries. Addresses can be referenced by byte offsets. A code address is a 24 bit number with the first 10 bits being the segment number and the last 14 bits the byte offset into the segment. Pascal code files have a one level segment dictionary. The maximum number of segments in a Pascal code file is 1024. The maximum size of each segment is 16384 bytes or 131072 bits.

The compiler automatically segments the code file. A particular procedure always resides within a single segment. Calls and exits are the only operators that require segment switching.

segment ? segment size

Call exit

OPERATORS

Operators whose mnemonic ends in 1, 2, or 3 have three different formats. The 1, 2, or 3 refers to the number of bytes that the first operand has.

Some operators have twin operators whose mnemonic begins with a "p". These twin operators perform the same function, but they do one extra thing. They first push the STACK.TOP register into the memory portion of the stack. The "p" stands for push.

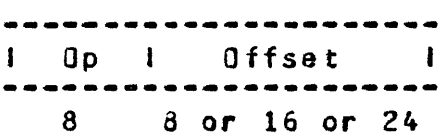
In general, parameters named "length" are lengths in bits. Parameters named "offset", "displacement", or "size" are in bytes. The operator description will indicate bits or bytes. Parameters that are in bytes are usually converted to bits by the interpreter before being used.

The format of each instruction is shown as a diagram in the following sections. The number under each field in the diagrams is the length in bits of that field.

LOAD ADDRESS OPERAIORS

Load Constant Address

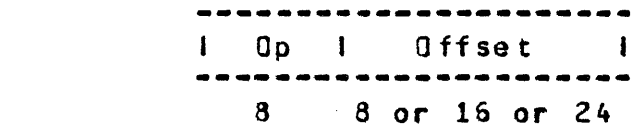
(ldcaddr1 = 1)
(ldcaddr2 = 2)
(ldcaddr3 = 3)
(pcaddr1 = 4)
(pcaddr2 = 5)
(pcaddr3 = 6)



The address of a constant at <Offset> bytes from the base of the constant pool is pushed onto the stack as a word. The address is computed by adding <Offset> to CONST. The pcaddr operators first push STACK.TOP into the memory portion of the stack.

Load Local Address

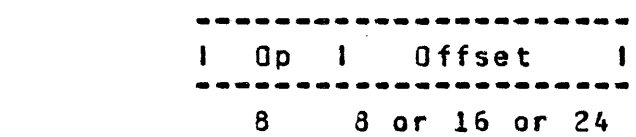
(ldladdr1 = 7)
(ldladdr2 = 8)
(ldladdr3 = 9)
(pladdr1 = 10)
(pladdr2 = 11)
(pladdr3 = 12)



The address of a variable at <Offset> bytes from the base of the the local variables is pushed onto the stack as a word. The address is computed by adding <Offset> to B. The pladdr operators first push STACK.TOP into the memory portion of the stack.

Load Global Address

(ldgaddr1 = 13)
(ldgaddr2 = 14)
(ldgaddr3 = 15)
(pgaddr1 = 16)
(pgaddr2 = 17)
(pgaddr3 = 18)



The address of a variable at <Offset> bytes from the base of the the global variables is pushed onto the stack as a word. The address is computed by adding <Offset> to G. The pgaddr operators first push STACK.TOP into the memory portion of the stack.

Load Lexic Level Address

(ldlladdr1 = 19)
(ldlladdr2 = 20)
(ldlladdr3 = 21)
(plladdr1 = 22)
(plladdr2 = 23)
(plladdr3 = 24)

Op	Offset	Lexic Level
8	8 or 16 or 24	8

The address of a variable at <Offset> bytes from the base of the the <Lexic Level> variables is pushed onto the stack as a word. The address is computed by adding <Offset> to DISPLAY [Lexic Level]. The plladdr operators first push STACK.TOP into the memory portion of the stack.

LOAD WORD OPERATORS

Load Constant Word

(ldcword1 = 25)
(ldcword2 = 26)
(ldcword3 = 27)
(pcword1 = 28)
(pcword2 = 29)
(pcword3 = 30)

Op	Value
8	8 or 16 or 24

<Value> is pushed onto the stack as a word. The sign is extended if necessary. The pcword operators first push STACK.TOP into the memory portion of the stack.

Load Local Word

```
(ldlword1 = 31)
(ldlword2 = 32)
(ldlword3 = 33)
(plword1  = 34)
(plword2  = 35)
(plword3  = 36)
```

Op	Offset
8	8 or 16 or 24

The value of a variable at <Offset> bytes from the base of the the local variables is pushed onto the stack as a word. The address is computed by adding <Offset> to B. The plword operators first push STACK.TOP into the memory portion of the stack.

Load Global Word

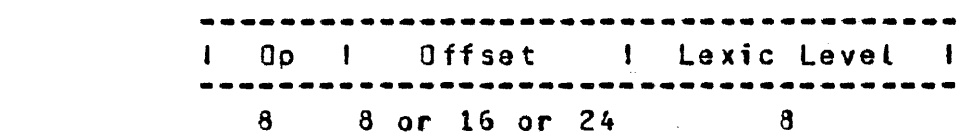
```
(ldgword1 = 37)
(ldgword2 = 38)
(ldgword3 = 39)
(pgword1  = 40)
(pgword2  = 41)
(pgword3  = 42)
```

Op	Offset
8	8 or 16 or 24

The value of a variable at <Offset> bytes from the base of the the global variables is pushed onto the stack as a word. The address is computed by adding <Offset> to G. The pgword operators first push STACK.TOP into the memory portion of the stack.

Load Lexic Level Word

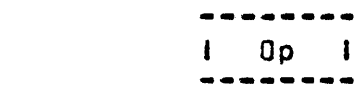
(ldllword1 = 43)
(ldllword2 = 44)
(ldllword3 = 45)
(pllword1 = 46)
(pllword2 = 47)
(pllword3 = 48)



The value of a variable at <Offset> bytes from the base of the the <Lexic Level> variables is pushed onto the stack as a word. The address is computed by adding <Offset> to DISPLAY [Lexic Level]. The pllword operators first push STACK.TOP into the memory portion of the stack.

Load Indirect Word

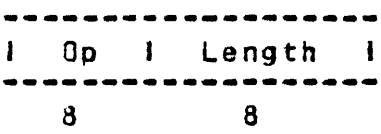
(loadind = 49)



The word on the top of the stack is the address of a variable. This address is popped off the stack and the variable it points to is pushed onto the stack as a word.

Load Unsigned Field

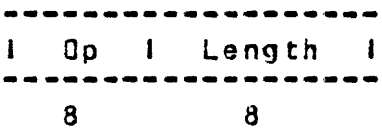
(loadufield = 50)



The word on the top of the stack is the address of a field within a variable. This address is popped off the stack and the field it points to for <Length> bits is pushed onto the stack as a word. The field is treated as unsigned.

Load Signed Field

(loadsfield = 51)

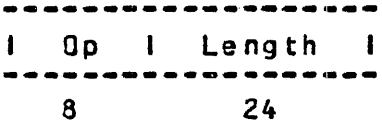


The word on the top of the stack is the address of a field within a variable. This address is popped off the stack and the field it points to for <Length> bits is pushed onto the stack as a word. The field is sign extended on the stack.

LOAD NON-WORD OPERATORS

Push Structure

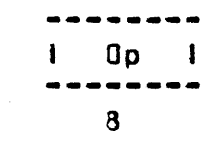
(pushstruct = 52)



The word on the top of the stack is the address of a structured variable. This address is popped off the stack and the data it points to for <Length> bits is pushed onto the stack. The data on the stack is padded such that the size is an integral number of bytes.

Push Real

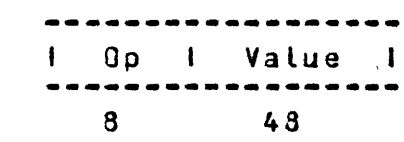
(pushreal = 53)



The word on the top of the stack is the address of a real variable. This address is popped off the stack and the real variable it points to is pushed onto the stack as a two word real value.

Push Real Constant

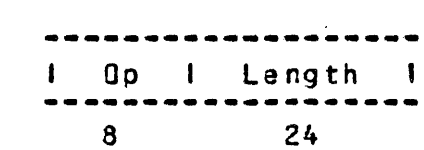
(pushrealconst = 142)



<Value> is pushed onto the stack as a two word real value.

Push Set

(pushset = 54)



The word on the top of the stack is the address of a set. This address is popped off the stack and the set it points to for <Length> bits is pushed onto the stack as a set. The set on the stack is padded with enough zeroes to make the resulting length 256 bits.

ADDRESS MODIFICATION OPERATORS

Field

(field1 = 55)
(field2 = 56)
(field3 = 57)

Op	Offset
8	8 or 16 or 24

The word on the top of the stack is an address. This address is incremented by <Offset> bits.

Index

(index = 58)

bit length

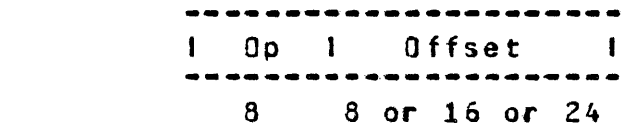
Op	Min	Max - Min	Element Length
8	24	24	24

The word on the top of the stack is an index into an array. The word at top-1 is the address of the beginning of the array. The index is popped from the stack and normalized to 0 by subtracting <Min>. If the resulting index is less than 0 or greater than <Max - Min> then an index out of range error is reported. If there is no error, then the normalized index is multipled by the <Element Length> (bits) and the address on the top of the stack is incremented by this amount.

STORE WORD OPERATORS

Store Local Word

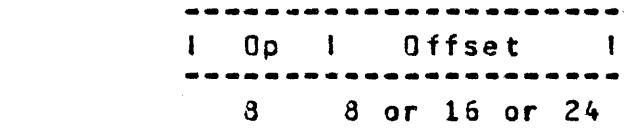
(stlword1 = 63)
(stlword2 = 64)
(stlword3 = 65)



The word on the top of the stack is popped and stored as a full word at the address computed by adding <Offset> bytes to the local variable base B.

Store Global Word

(stgword1 = 66)
(stgword2 = 67)
(stgword3 = 68)



The word on the top of the stack is popped and stored as a full word at the address computed by adding <Offset> bytes to the global variable base G.

Store Lexic Level Word

(stllword1 = 69)
(stllword2 = 70)
(stllword3 = 71)

Op	Offset	Lexic Level
8	8 or 16 or 24	8

The word on the top of the stack is popped and stored as a full word at the address computed by adding <Offset> bytes to DISPLAY [Lexic Level].

Store Indirect Word

(storeind = 62)

Op
8

The word on the top of the stack is stored as a full word at the address indicated by the top-1 word. Both words are popped from the stack.

STORE NON-WORD OPERATORS

Store Real

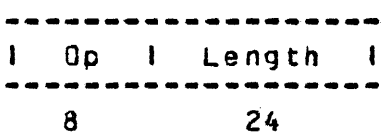
(storereal = 72)

Op
8

The real value on the top two of the stack is stored at the address indicated by the top-1 word. The real value and the address are popped from the stack.

Store Set

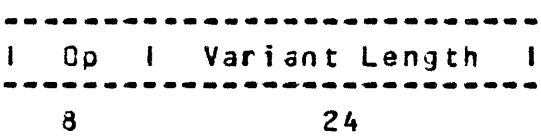
(storeset = 73)



The top 256 bits of the stack is a set. This set is stored for <Length> bits at the address indicated by the word on the stack below the set. If any members of the set being stored are outside the range of the set being stored into, then a value out of range error is reported. Both the set and the address are popped from the stack.

Store Tag

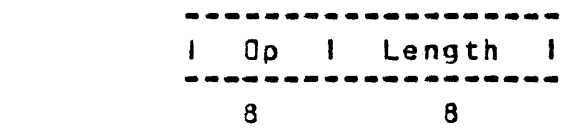
(storetag = 74)



The top of the stack is a tag value. The top-1 item is the address of the tag field in which to store the value. The tag is stored into the field pointed to by the address for a length of 5 bits. The variant part of the record immediately following the tag field is set to undefined for a length of <Variant Length> bits.

Store Unsigned Field

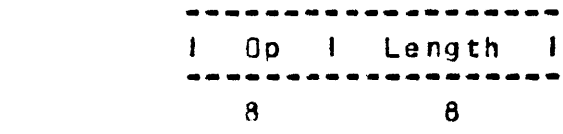
(storeufield = 75)



The word on the top of the stack is stored at the field pointed to by the address at top-1 for <Length> bits. The value is treated as unsigned. Both items are popped from the stack. If data is lost when stored into the field then a value out of range is reported.

Store Signed Field

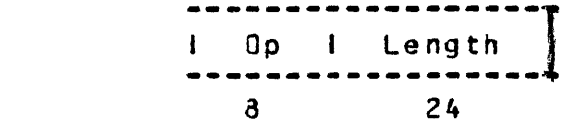
(storesfield = 76)



The word on the top of the stack is stored at the field pointed to by the address at top-1 for <Length> bits. The value is treated as signed. Both items are popped from the stack. If data is lost when stored into the field then a value out of range is reported.

Copy Structure

(copystruct = 78)

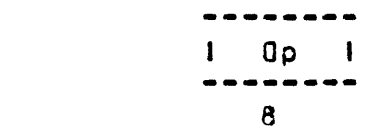


The structure pointed to by the address on the top of the stack is copied to the structure pointed to by the address at top-1. <Length> bits are copied. Both addresses are popped from the stack.

VALIDATION OPERATORS

Pointer

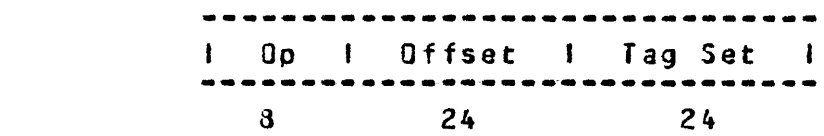
(pointer = 59)



The word on the top of the stack is an address. If it is equal to the value reserved for a NIL pointer then a pointer error is reported. The address is left on the stack.

Variant

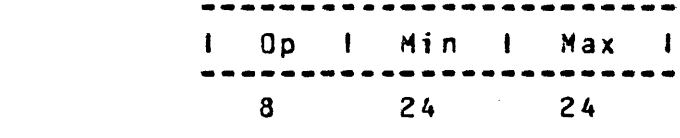
(variant = 60)



The word on the top of the stack is the address of a variant record. <Offset> is added to this address to give the address of the tag field for the variant. If the value of the tag is not a member of <Tag Set> then a variant error is reported. The original address is left on the stack.

Range

(range = 61)

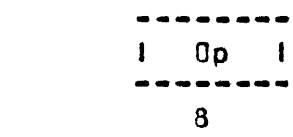


If the value on the top of the stack is less than <Min> or greater than <Max> then a value out of range error is reported. The value is left on the stack.

NUMERIC COMPUTATION

Negate Word

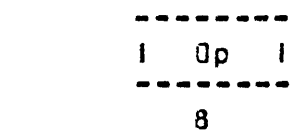
(negword = 86)



The word on the top of the stack is popped and its negated (unary minus) value is pushed.

Absolute Value Word

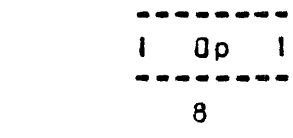
(absword = 161)



The word on the top of the of the stack is popped and its absolute value is pushed.

Successor Word

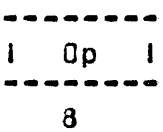
(succword = 163)



The word on the top of the stack is incremented by 1.

Predecessor Word

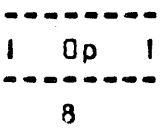
(predword = 164)



The word on the top of the stack is decremented by 1.

Increment Word

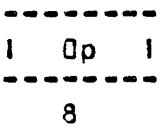
(incrword = 153)



The word on the top of the stack is an address. The word it points to is incremented by 1. The address is popped from the stack.

Decrement Word

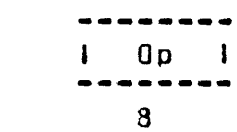
(decrword = 154)



The word on the top of the stack is an address. The word it points to is decremented by 1. The address is popped from the stack.

Add Word

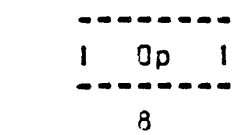
(addword = 88)



The top two words are popped and their sum is pushed. Integer overflow is checked for.

Subtract Word

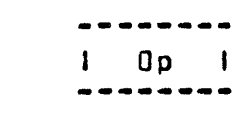
(subword = 90)



The top two words are popped and their difference is pushed. The top word is subtracted from top-1. Integer overflow is checked for.

Multiply Word

(mulword = 93)



The top two words are popped and their product is pushed. Integer overflow is checked for.

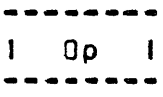
Divide Word

(divword = 95)

The top two words are popped and their quotient is pushed. The top-1 word is divided by the top word. If the top word is equal to 0 then divide by zero is reported.

Modulo Word

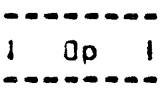
(modword = 97)



The top two words are popped and their remainder is pushed. The top-1 word is divided by the top word. If the top word is equal to 0 then divide by zero is reported.

Square Word

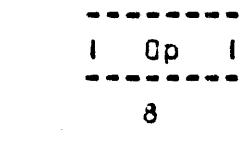
(squareword = 144)



The word on the top of the stack is replaced by its value squared. Integer overflow is checked for.

Negate Real

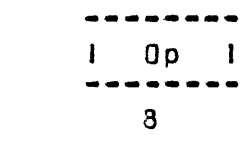
(negreal = 87)



The real on the top of the stack is popped and its negated (unary minus) value is pushed.

Absolute Value Real

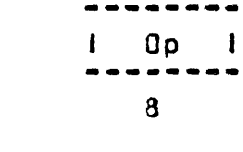
(absreal = 162)



The real on the top of the of the stack is popped and its absolute value is pushed.

Add Real

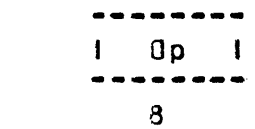
(addreal = 89)



The top two reals are popped and their sum is pushed. Real overflow and underflow are checked for.

Subtract Real

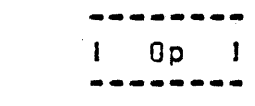
(subreal = 91)



The top two reals are popped and their difference is pushed. The top real is subtracted from top-1. Real overflow and underflow are checked for.

Multiply Real

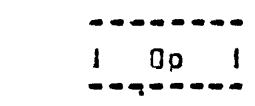
(mulreal = 94)



The top two reals are popped and their product is pushed. Real overflow and underflow are checked for.

Divide Real

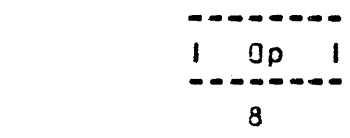
(divreal = 96)



The top two reals are popped and their quotient is pushed. The top-1 real is divided by the top word. If the top word is equal to 0 then divide by zero is reported. Real overflow and underflow are checked for.

Square Real

(squarereal = 145)

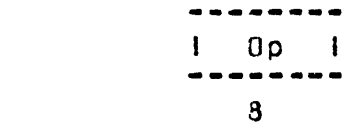


The real on the top of the stack is replaced by its value squared. Real overflow and underflow are checked for.

BOOLEAN COMPUTATION OPERATORS

Not

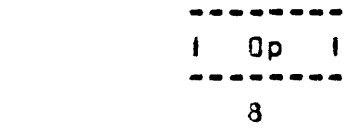
(not = 81)



The boolean on the top of the stack is complemented.

And Word

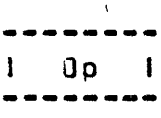
(andword = 82)



The two booleans on the top of the stack are popped and are anded together. The result is pushed.

Or Word

(orword = 84)

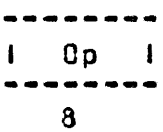


The two booleans on the top of the stack are popped and are ored together. The result is pushed.

SET COMPUTATION OPERATORS

Union Set

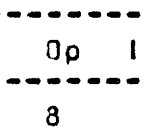
(union = 85)



The top two items on the stack are sets. They are both popped and ored together producing their union. The resulting set is pushed back onto the stack. All of the sets are 256 bits long.

Intersection Set

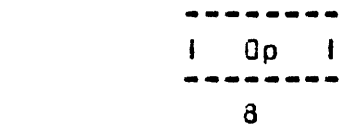
(intersection = 83)



The top two items on the stack are sets. They are both popped and anded together producing their intersection. The resulting set is pushed back onto the stack. All of the sets are 256 bits long.

Set Difference

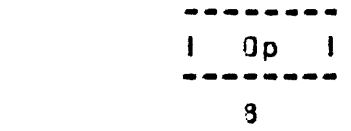
(setdiff = 92)



The top two items on the stack are sets. They are both popped and the top set is subtracted from the top-1 set. This difference is pushed back onto the stack. All of the sets are 256 bits long. The difference of two sets A - B is the members of A that are not in B. This operation is performed by complementing the set B and anding this with the set A.

Build Set

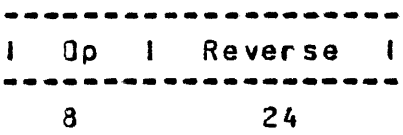
(buildset = 98)



The top word on the stack is an integer between 0 and 255. The top-1 item is a set. The integer is popped from the stack and if it is not in the correct range a value out of range is reported. The set is modified such that it contains the member corresponding to the integer.

Build Set Range

(buildsetrange = 143)

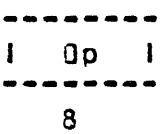


The top two words on the stack are the minimum and maximum of a range of members to be placed in a set. The set is the item at top-2. If either of the words are out of the range 0 to 255 then value out of range is reported. If <Reverse> is 0, then the top word is the maximum and top-1 is the minimum. If <Reverse> is 1, then the top word is the minimum and top-1 is the maximum. The words are popped from the stack and the set is modified such that it contains the members between minimum and maximum inclusive.

RELATIONAL OPERATORS

Equal Word

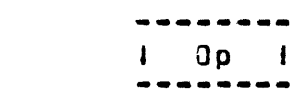
(eqword = 101)



The two words on the top of the stack are popped. If the top-1 word is equal to top then true is pushed onto the stack, otherwise false is pushed.

Not Equal Word

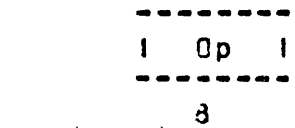
(neword = 104)



The two words on the top of the stack are popped. If the top-1 word is not equal to top then true is pushed onto the stack, otherwise false is pushed.

Less Than Word

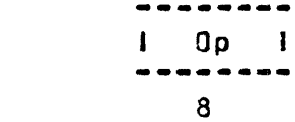
(lsword = 100)



The two words on the top of the stack are popped. If the top-1 word is less than top then true is pushed onto the stack, otherwise false is pushed.

Not Less Than Word

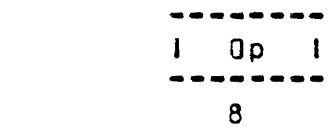
(nlword = 103)



The two words on the top of the stack are popped. If the top-1 word is greater than or equal to top then true is pushed onto the stack, otherwise false is pushed.

Greater Than Word

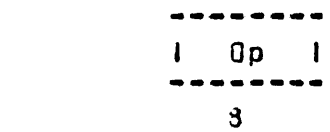
(grword = 102)



The two words on the top of the stack are popped. If the top-1 word is greater than top then true is pushed onto the stack, otherwise false is pushed.

Not Greater Than Word

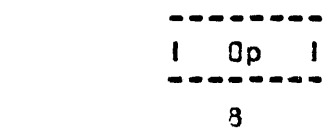
(ngword = 105)



The two words on the top of the stack are popped. If the top-1 word is less than or equal to top then true is pushed onto the stack, otherwise false is pushed.

Odd

(odd = 140)



The word on the top of the stack is popped. If the low order bit of the word is 1, then true is pushed, otherwise false is pushed.

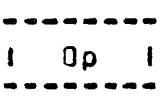
Equal Real

(eqreal = 107)

The two reals on the top of the stack are popped. If the top-1 real is equal to top then true is pushed onto the stack, otherwise false is pushed.

Not Equal Real

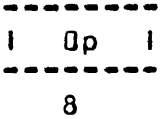
(nereal = 110)



The two reals on the top of the stack are popped. If the top-1 real is not equal to top then true is pushed onto the stack, otherwise false is pushed.

Less Than Real

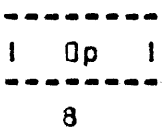
(lsreal = 106)



The two reals on the top of the stack are popped. If the top-1 real is less than top then true is pushed onto the stack, otherwise false is pushed.

Not Less Real

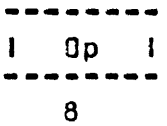
(nlreal = 109)



The two reals on the top of the stack are popped. If the top-1 real is greater than or equal to top then true is pushed onto the stack, otherwise false is pushed.

Greater Than Real

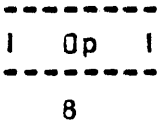
(grreal = 108)



The two reals on the top of the stack are popped. If the top-1 real is greater than top then true is pushed onto the stack, otherwise false is pushed.

Not Greater Than Real

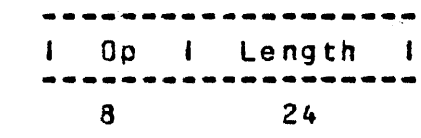
(ngreal = 111)



The two reals on the top of the stack are popped. If the top-1 real is less than or equal to top then true is pushed onto the stack, otherwise false is pushed.

Equal Structure

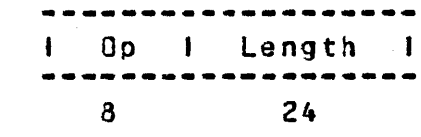
(eqstruct = 117)



The top two words are addresses of two structures. The addresses are popped and the structures are compared for <Length> bits. If the structure pointed to by top-1 is equal to the structure pointed to by top then true is pushed onto the stack, otherwise false is pushed.

Not Equal Stucture

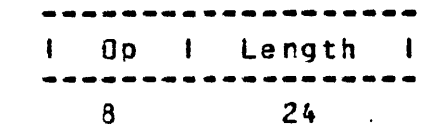
(nestruct = 120)



The top two words are addresses of two structures. The addresses are popped and the structures are compared for <Length> bits. If the structure pointed to by top-1 is not equal to the structure pointed to by top then true is pushed onto the stack, otherwise false is pushed.

Less Than Structure

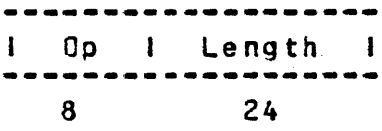
(lsstruct = 116)



The top two words are addresses of two structures. The addresses are popped and the structures are compared for <Length> bits. If the structure pointed to by top-1 is less than the structure pointed to by top then true is pushed onto the stack, otherwise false is pushed.

Not Less Than Structure

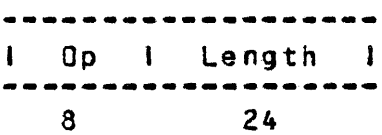
(nlstruct = 119)



The top two words are addresses of two structures. The addresses are popped and the structures are compared for <Length> bits. If the structure pointed to by top-1 is greater than or equal to the structure pointed to by top then true is pushed onto the stack, otherwise false is pushed.

Greater Than Structure

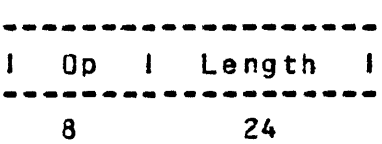
(grstruct = 118)



The top two words are addresses of two structures. The addresses are popped and the structures are compared for <Length> bits. If the structure pointed to by top-1 is greater than the structure pointed to by top then true is pushed onto the stack, otherwise false is pushed.

Not Greater Than Structure

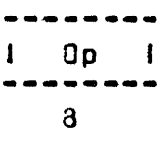
(ngstruct = 121)



The top two words are addresses of two structures. The addresses are popped and the structures are compared for <Length> bits. If the structure pointed to by top-1 is less than or equal to the structure pointed to by top then true is pushed onto the stack, otherwise false is pushed.

Equal Set

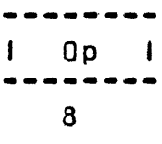
(eqset = 112)



The two sets on the top of the stack are popped and compared. If they are equal then true is pushed, otherwise false is pushed. The sets are both 256 bits long.

Not Equal Set

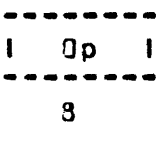
(neset = 114)



The two sets on the top of the stack are popped and compared. If they are not equal then true is pushed, otherwise false is pushed. The sets are both 256 bits long.

Not Less Than Set

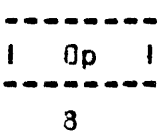
(nlset = 113)



The two sets on the top of the stack are popped and compared. If the top-1 set contains the top set then true is pushed, otherwise false is pushed. The sets are both 256 bits long.

Not Greater Than Set

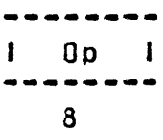
(ngset = 115)



The two sets on the top of the stack are popped and compared. If the top-1 set is a subset of the top set then true is pushed, otherwise false is pusned. The sets are both 256 bits long.

In Set

(inset = 99)

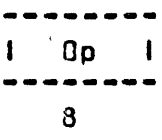


The top of the stack is a set. If the word at top-1 is a member of that set then true is pushed, otherwise false is pushed. Both the set and the word are popped before the result is pushed.

CONVERSION OPERATORS

Truncate Real

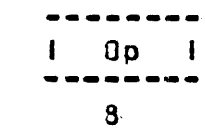
(truncreal = 160)



The real on the top of the stack is popped and an integer representing the truncated real is pushed. Integer overflow is checked for.

Round

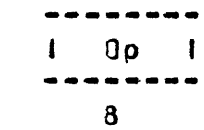
(round = 146)



The real on the top of the stack is popped and an integer representing the rounded real is pushed. Integer overflow is checked for.

Convert Word to Real

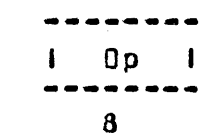
(convword = 165)



The word on the top of the stack is popped and its value is pushed as a real number.

Convert Word2 to Real

(convword2 = 141)

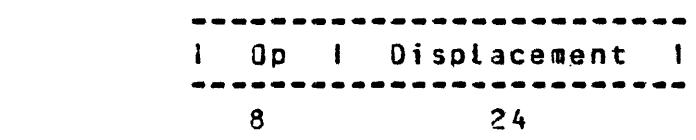


A real is on the top of the stack and a word is at top-1. The real number is popped and saved. The word is popped and converted to a real which is then pushed. The real that was saved is then pushed back onto the stack.

CONTROL OPERATORS

Jump

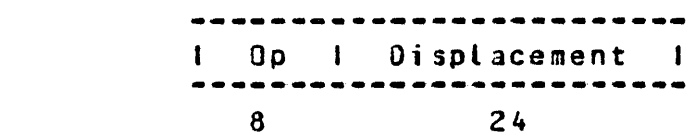
(jump = 123)



<Displacement> bytes is added to NIP. The NIP is pointing just past the jump op code field before being undated. A jump can only branch inside a code segment.

False Jump

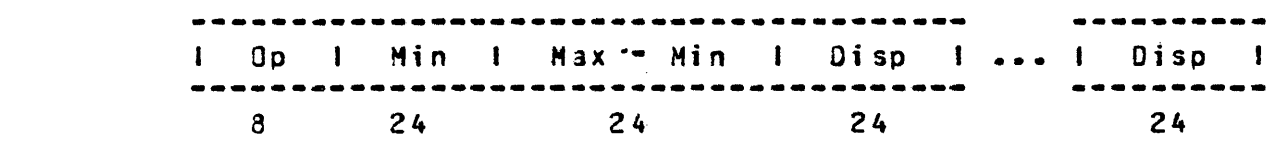
(falsejump = 124)



The boolean on top of the stack is popped. If it is false then <Displacement> is added to NIP. If the value is true, execution continues with the op following the falsejump.

Case Jump

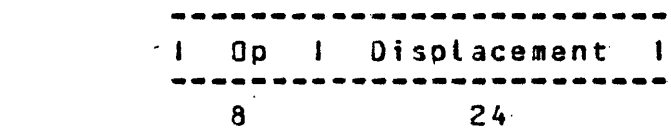
(casejump = 125)



The word on the top of the stack is popped. The word is normalized to 0 by subtracting <Min>. If the result is negative or greater than <Max - Min> then a case error is reported. The normalized word is used as an index into the table of <Disp>s. The corresponding <Disp> is read and if 0 then a case error is reported. If the <Disp> is not 0, then it is added as bytes to NIP. NIP is pointing just after <Disp> before it is modified.

Call

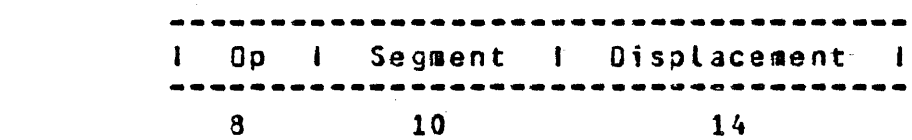
(call = 128)
(pushcall = 127)



The return displacement and segment number are pushed onto the stack. NIP, which points just after the call op field, is modified by <Displacement> bytes and execution continues there. The pushcall operator first pushes STACK.TOP into the memory portion of the stack.

Call Segment

(callseg = 139)
(pushcallseg = 138)



The return displacement and segment number are pushed onto the stack. This op transfers control to another code segment. is the segment number and <Displacement> is the byte offset into that segment. If the segment is not in memory, the operating system is requested to bring it in. The pushcallseg operator first pushes STACK.TOP into the memory portion of the stack.

Enter Procedure

(enter = 130)

Op	Stack Size	Parm Size	Line	LL	Var Size
8	24	24	24	24	24

This is the first op of every procedure. It constructs a new activation record for that procedure. If there are not at least <Stack Size> bytes of memory available between SP and HEAPTOP then a stack limit error is reported. The lexic level of the calling routine, local base (B) of the calling routine, SP restore value, and <Line> are pushed onto the stack to complete the linkage information. <Parm Size> is used to compute the SP restore value. <LL> is used to set the new lexic level. B is updated to the base of the new local variables. SP is set to B + <Var Size> bytes or just beyond the variables for the evaluation stack area.

Exit Procedure

(exit = 132)

Op
8

The display register which corresponds to the lexic level being exited is restored to its value before the call. This value is found by searching through the activation records until the first record with the same lexic level is found. The line number, B, and NIP are restored by retrieving values from the linkage area that was built when the routine being exited was called. SP is set back to its restore value. Execution continues at the point following the call. It is possible that a segment change will be made by the exit.

Exit Function

(functionexit = 131)

Op
8

The same actions that the exit performs are also done. Also, the function result is copied to the new top of the stack.

Enter Program

(enterprog = 133)

Op	Line	Stack Size	Var Size
8	24	24	24

The global activation record is constructed similar to the way the enterproc op does.

Exit Program

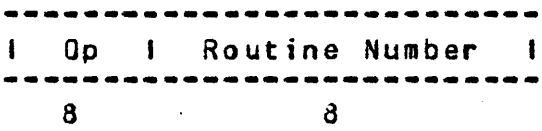
(exitprog = 134)

Op
8

The program is terminated and control is given to the operating system.

Call Standard Routine

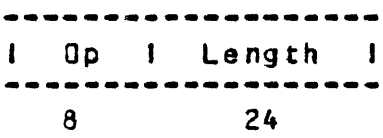
(callstd = 135)



A standard routine indicated by <Routine Number> is called. See Section 6 on Standard Routines.

Function Value

(funcvalue = 122)

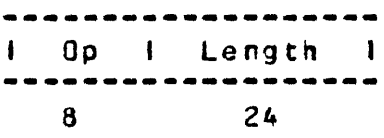


An item of <Length> bits is pushed onto the stack. This area is to be used inside the function for its result.

MISCELLANEOUS OPERATORS

Pop

(pop = 148)



<Length> bits are popped from the stack.

Push Stack

(pushstack = 149)

Op
8

STACK.TOP is pushed into the memory portion of the stack.

New Line

(newline1 = 150)
(newline2 = 151)
(newline3 = 152)

Op	Line Number
8	8 or 16 or 24

CURRENT.LINE is set to <Line Number>.

New

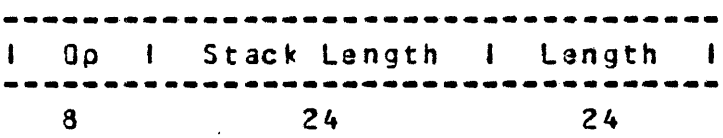
(new = 79)

Op	Stack Length	Length
8	24	24

The word on the top of the stack is the address of a pointer variable. A new cell on the heap of <Length> bits is allocated if there is sufficient space available between the heap and the stack. <Stack Length> is used to determine if there is enough space. If there is not, then an error is reported. The address is popped from the stack.

New Initialized

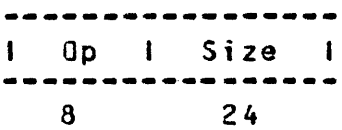
(newinit = 80)



This op performs the same as new except that the new cell is set to undefined.

Initialize Variables

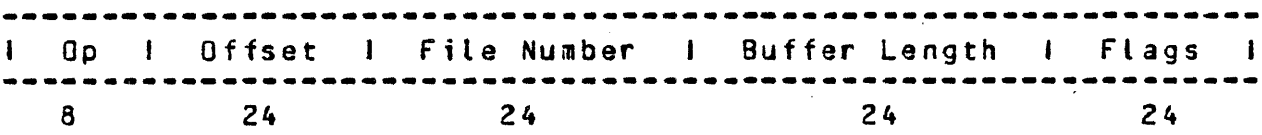
(initvar = 126)



The addressable storage of the local activation record is set to undefined for <Size> bytes.

Construct File Descriptor

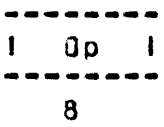
(cfdesc = 136)



A file descriptor is constructed at <Offset> bytes from B. <File Number> and <Buffer Length> (bits) are used to initialize the buffer. <Flags> are various booleans that tell what kind of file is being handled. Bit 0 is not used. Bit 1 means a text file. Bit 2 means the file "input". Bit 3 means the file "output". The other bits in <Flags> are not used.

Text

(text = 137)



The word on the top of the stack is the address of a text file descriptor. The current character offset field in the descriptor is read and added to the original address. This resulting address replaces the original address on the top of the stack.

STANDARD ROUTINES

The following are similar to the operators described in Section 5. The difference is that each is invoked by the "callstd" operator instead of being a specific operator. The value given for each standard routine is the <Routine Number> field of the callstd operator.

FILE HANDLING ROUTINES

Get

(get = 0)

The word on the top of the stack is the address of a non-text file descriptor. It is popped. If the file is not defined or is at end of file then an error is reported. The next record in the file is read into the buffer portion of the file descriptor.

Get Text

(gettext = 1)

The word on the top of the stack is the address of a text file descriptor. It is popped. If the file is not defined or is at end of file then an error is reported. If the current character offset field in the file descriptor is equal to the last character offset, then the next record is read into the file buffer and the current character offset is set to 0. Otherwise, the current character offset is incremented to point to the next character in the current buffer.

Put

(put = 2)

The word on the top of the stack is the address of a non-text file descriptor. It is popped. If the file is not defined or is not at the end of file then an error is reported. The file buffer is written at the next sequential position in the file.

Put Text

(puttext = 3)

The word on the top of the stack is the address of a text file descriptor. It is popped. If the file is not defined or is not at the end of file then an error is reported. The current character offset is incremented to the next position in file buffer. If it is equal to the last position in the buffer then the buffer is written as the next record in the file.

Update (Deleted)

(update = 4)

The word on the top of the stack is the address of a non-text file descriptor. It is popped. If the file is not defined or is at the end of file then an error is reported. The buffer is written at the current position in the file and the next record is then read into the buffer.

Position (Replaced by seek)

(position = 5)

The word on the top of the stack is the address of a non-text file descriptor. The top-1 word is the record number (0 relative) in the file to be positioned to. Both are popped. If the file is not defined then an error is reported. The file is positioned to the record number and that record is read into the file buffer.

End Of File

(eof = 6)

The word on the top of the stack is the address of a text or non-text file. It is popped. If the file is not defined then an error is reported. If the file is at end of file, then true is pushed onto the stack, otherwise false is pushed.

End Of Line

(eoln = 7)

The word on the top of the stack is the address of a text file. It is popped. If the file is not defined then an error is reported. If the current character offset is equal to the last character offset, then true is pushed, otherwise false is pushed.

Reset

(reset = 8)

The word on the top of the stack is the address of a non-text file. It is popped. If the file is already open, then it is closed. The file is opened input/output if the hardware type is disk, otherwise it is opened input. The first record is read.

Reset Text

(resettext = 9)

The word on the top of the stack is the address of a text file. It is popped. If the file is already open, then it is closed. The file is opened input/output if the hardware type is disk, otherwise it is opened input. The first record is read.

Rewrite

(rewrite = 10)

The word on the top of the stack is the address of a non-text file. It is popped. If the file is already open, then it is closed. The file is opened new/input/output if the hardware type is disk, otherwise it is opened new/output.

Rewrite Text

(rewritetext = 11)

The word on the top of the stack is the address of a text file. It is popped. If the file is already open, then it is closed. The file is opened new/input/output if the hardware type is disk, otherwise it is opened new/output.

Read Character

(readc = 12)

The word on the top of the stack is the address of a text file. The word at top-1 is the address of a character variable. They are both popped. The current character in the file buffer is moved to the character variable. A get operation is then performed.

Read Integer

(readi = 13)

The word on the top of the stack is the address of a text file. The word at top-1 is the address of an integer variable. They are both popped. An integer is read from the file and converted to binary and stored in the integer variable. Various errors are detected such as invalid integer format, integer overflow, and end of file.

Write Character

(writec = 14)

The word on the top of the stack is the address of a text file. The word at top-1 is the minimum number of characters to be written. The word at top-2 is the integer value of a character to be written. They are all popped. If the file is not defined or is not at end of file then an error is reported. The character is written in the file buffer preceded by enough blanks to equal minimum width. If this many characters would cause the buffer to overflow, then an error is reported.

Write String

(writes = 15)

The word on the top of the stack is the address of a text file. The word at top-1 is the length of the string to be written. The word at top-2 is the minimum number of characters to be written. The word at top-3 is the address of the string of characters to be written. They are all popped. If the file is not defined or is not at end of file then an error is reported. The character string is written to the file buffer preceded by enough blanks to equal minimum width. If this many characters would cause the buffer to overflow, then an error is reported.

Write Integer

(writei = 16)

The word on the top of the stack is the address of a text file. The word at top-1 is the minimum number of characters to be written. The word at top-2 is the integer value of an integer to be written. They are all popped. If the file is not defined or is not at end of file then an error is reported. The is converted to character form and is written to the file preceded by enough blanks to equal minimum width. If this many characters would cause the buffer to overflow, then an error is reported.

Write Boolean

Read Line

(readln = 17)

The word on the top of the stack is the address of a text file. It is popped. If the file is not defined or is at end of file then an error is reported. The next record of the file is read into the file buffer area.

Write Line

(writeln = 18)

The word on the top of the stack is the address of a text file. It is popped. If the file is not defined or is not at end of file then an error is reported. The file buffer is written as the next record in the file.

Page

(page = 19)

The word on the top of the stack is the address of a text file. It is popped. If the file is not defined or is not at end of file then an error is reported. A page eject operation is written to the file.

Retain

(retain = 20)

The word on the top of the stack is the address of a text or non-text file. It is popped. If the file is not defined then an error is reported. The file is closed and locked into the directory.

Remove

(remove = 21)

The word on the top of the stack is the address of a text or non-text file. It is popped. If the file is not defined then an error is reported. The file is closed and purged.

Reassign

(reassign = 22)

The word on the top of the stack is the address of a text or non-text file. The word at top-1 is the length in bytes of a string that is to be the new external file name. All three items are popped. If the file is open then an error is reported. The external file name is changed to the new name. The string is assumed to be in the proper format for a file title. The string is passed directly to the MCP via the change attribute communicate.

Relinquish

(relinguisn = 23)

The word on the top of the stack is the address of a text or non-text file. It is popped. The file is closed release. This is the default remove operation for internal files.

Remain

(remain = 24)

The word on the top of the stack is the address of a text or non-text file. It is popped. The file is closed and locked into the directory. This is the default retain operation for external files.

Write Boolean

(writeb = 25)

The word on the top of the stack is the address of a text file. The word at top-1 is the minimum number of characters to be written. The word at top-2 is a boolean value. They are all popped. If the file is not defined or is not at end of file then an error is reported. If the boolean value is true then the string "TRUE" is written, otherwise "FALSE" is written. Either string is preceded by enough blanks to equal minimum width. If this many characters would cause the buffer to overflow, then an error is reported.

NON-FILE ROUTINES

Mark

(mark = 26)

The word on the top of the stack is the address of a pointer variable. The word is popped and the pointer is set to the value in HEAPTOP.

Release

(release = 27)

The word on the top of the stack is the address to which the heap is to be cut back. This word is popped and HEAPTOP is set to its value.

Expo

(expo = 28)

The real on the top of the stack is replaced by its exponent part. The exponent is left on the stack as a word.

Clock

(clock = 29)

The number of tenths of seconds since the beginning of this program is pushed onto the stack as a word.

Date

(date = 30)

The word on the top of the stack is the address of an 8 character string. It is popped and the current date is stored in this string in the format mm/dd/yy.

Time

(time = 31)

The word on the top of the stack is the address of a 10 character string. It is popped and the current time is stored in this string in the format hh:mm:ss.t.

Abort

(abort = 32)

The program is immediately aborted.

Communicate

(communicate = 33)

The word on the top of the stack is the length of the structure to be passed to the communicate. The word at top-1 is the address of that structure. They are both popped and a communicate is done using this structure as the parameter.

Reinstate Message

(reinstatemessage = 34)

The reinstate message field in the run structure nucleus is read and pushed onto the stack as a word.

Display

(display = 35)

The word on the top of the stack is the length in bytes of the string to be displayed. The word at top-1 is the address of that string. They are both popped. The string is then displayed on the ODT.

APPENDIX A - OPERATORS IN ALPHABETICAL ORDER

Op Mnemonic	Op Name	Op Code
absreal	Absolute Value Real	162
absword	Absolute Value Word	161
addreal	Add Real	89
addword	Add Word	88
andword	And Word	82
buildset	Build Set	98
buildsetrange	Build Set Range	143
call	Call	128
callseg	Call Segment	139
callstd	Call Standard Routine	135
casejump	Case Jump	125
cfdesc	Construct File Descriptor	136
convword	Convert Word To Real	165
convword2	Convert Word2 To Real	141
copystruct	Copy Structure	78
decrword	Decrement Word	154
divreal	Divide Real	96
divword	Divide Word	95
enter	Enter Procedure	130
enterprog	Enter Program	133
eqreal	Equal Real	107
eqset	Equal Set	112
eqstruct	Equal Structure	117
eqword	Equal Word	101
exit	Exit	132
exitprog	Exit Program	134
falsejump	False Jump	124
field1	Field	55
field2	Field	56
field3	Field	57
functionexit	Function Exit	131
funcvalue	Function Value	122
grreal	Greater Than Real	108
grstruct	Greater Than Structure	118
grword	Greater Than Word	102
incrword	Increment Word	153
index	Index	58
initvar	Initialize Variables	126
inset	In Set	99
intersection	Intersection Set	83

jump	Jump	123
ldcaddr1	Load Constant Address	1
ldcaddr2	Load Constant Address	2
ldcaddr3	Load Constant Address	3
ldcword1	Load Constant Word	25
ldcword2	Load Constant Word	26
ldcword3	Load Constant Word	27
lgaddr1	Load Global Address	13
lgaddr2	Load Global Address	14
lgaddr3	Load Global Address	15
ldgword1	Load Global Word	37
ldgword2	Load Global Word	38
ldgword3	Load Global Word	39
ldladdr1	Load Local Address	7
ldladdr2	Load Local Address	8
ldladdr3	Load Local Address	9
ldlladdr1	Load Lexic Level Address	19
ldlladdr2	Load Lexic Level Address	20
ldlladdr3	Load Lexic Level Address	21
ldllword1	Load Lexic Level Word	43
ldllword2	Load Lexic Level Word	44
ldllword3	Load Lexic Level Word	45
ldlword1	Load Local Word	31
ldlword2	Load Local Word	32
ldlword3	Load Local Word	33
loadind	Load Indirect Word	49
loadsfield	Load Signed Field	51
loadufield	Load Unsigned Field	50
lsreal	Less Than Real	106
lsstruct	Less Than Structure	116
lsword	Less Than Word	100
modword	Modulo Word	97
mulreal	Multiply Real	94
mulword	Multiply Word	93
negreal	Negate Real	87
negword	Negate Word	86
nereal	Not Equal Real	110
neset	Not Equal Set	114
nestruct	Not Equal Structure	120
new	New	79
newinit	New Initialized	80
newline1	New Line	150
newline2	New Line	151
newline3	New Line	152
neword	Not Equal Word	104
ngreal	Not Greater Than Real	111
ngset	Not Greater Than Set	115
ngstruct	Not Greater Than Structure	121
ngword	Not Greater Than Word	105
nlreal	Not Less Than Real	109
nlset	Not Less Than Set	113

nlstruct	Not Less Than Structure	119
nlword	Not Less Than Word	103
not	Not	81
odd	Odd	140
orword	Or Word	84
pcaddr1	Push Constant Address	4
pcaddr2	Push Constant Address	5
pcaddr3	Push Constant Address	6
pcword1	Push Constant Word	28
pcword2	Push Constant Word	29
pcword3	Push Constant Word	30
pgaddr1	Push Global Address	16
pgaddr2	Push Global Address	17
pgaddr3	Push Global Address	18
pgword1	Push Global Word	40
pgword2	Push Global Word	41
pgword3	Push Global Word	42
pladdr1	Push Local Address	10
pladdr2	Push Local Address	11
pladdr3	Push Local Address	12
plladdr1	Push Lexic Level Address	22
plladdr2	Push Lexic Level Address	23
plladdr3	Push Lexic Level Address	24
pllword1	Push Lexic Level Word	46
pllword2	Push Lexic Level Word	47
pllword3	Push Lexic Level Word	48
plword1	Push Local Word	34
plword2	Push Local Word	35
plword3	Push Local Word	36
pointer	Pointer	59
pop	Pop	147
predword	Predecessor Word	164
pushcall	Push Call	127
pushcallseg	Push Call Segment	138
pushreal	Push Real	53
pushrealconst	Push Real Constant	142
pushset	Push Set	54
pushstack	Push Stack	149
pushstruct	Push Structure	52
range	Range	61
round	Round	146
setdiff	Set Difference	92
squarereal	Square Real	145
squareword	Square Word	144
stgword1	Store Global Word	66
stgword2	Store Global Word	67
stgword3	Store Global Word	68
stllword1	Store Lexic Level Word	69
stllword2	Store Lexic Level Word	70
stllword3	Store Lexic Level Word	71

stlword1	Store Local Word	63
stlword2	Store Local Word	64
stlword3	Store Local Word	65
storeind	Store Indirect Word	62
storereal	Store Real	72
storeset	Store Set	73
storesfield	Store Signed Field	76
storetag	Store Tag	74
storeufield	Store Unsigned Field	75
subreal	Subtract Real	91
subword	Subtract Word	90
succword	Successor Word	163
text	Text	137
truncreal	Truncate Real	160
union	Union Set	85
variant	Variant	60

APPENDIX B - OPERATORS IN OP CODE ORDER

Op Mnemonic	Op Name	Op Code
ldcaddr1	Load Constant Address	1
ldcaddr2	Load Constant Address	2
ldcaddr3	Load Constant Address	3
pcaddr1	Push Constant Address	4
pcaddr2	Push Constant Address	5
pcaddr3	Push Constant Address	6
ldladdr1	Load Local Address	7
ldladdr2	Load Local Address	8
ldladdr3	Load Local Address	9
pladdr1	Push Local Address	10
pladdr2	Push Local Address	11
pladdr3	Push Local Address	12
lgaddr1	Load Global Address	13
lgaddr2	Load Global Address	14
lgaddr3	Load Global Address	15
pgaddr1	Push Global Address	16
pgaddr2	Push Global Address	17
pgaddr3	Push Global Address	18
ldlladdr1	Load Lexic Level Address	19
ldlladdr2	Load Lexic Level Address	20
ldlladdr3	Load Lexic Level Address	21
plladdr1	Push Lexic Level Address	22
plladdr2	Push Lexic Level Address	23
plladdr3	Push Lexic Level Address	24
ldcword1	Load Constant Word	25
ldcword2	Load Constant Word	26
ldcword3	Load Constant Word	27
pcword1	Push Constant Word	28
pcword2	Push Constant Word	29
pcword3	Push Constant Word	30
ldlword1	Load Local Word	31
ldlword2	Load Local Word	32
ldlword3	Load Local Word	33
plword1	Push Local Word	34
plword2	Push Local Word	35
plword3	Push Local Word	36
ldgword1	Load Global Word	37
ldgword2	Load Global Word	38
ldgword3	Load Global Word	39
pgword1	Push Global Word	40
pgword2	Push Global Word	41
pgword3	Push Global Word	42
ldllword1	Load Lexic Level Word	43
ldllword2	Load Lexic Level Word	44
ldllword3	Load Lexic Level Word	45
pllword1	Push Lexic Level Word	46
pllword2	Push Lexic Level Word	47
pllword3	Push Lexic Level Word	48

loadind	Load Indirect Word	49
loadufield	Load Unsigned Field	50
loadsfield	Load Signed Field	51
pushstruct	Push Structure	52
pushreal	Push Real	53
pushset	Push Set	54
field1	Field	55
field2	Field	56
field3	Field	57
index	Index	58
pointer	Pointer	59
variant	Variant	60
range	Range	61
storeind	Store Indirect Word	62
stlword1	Store Local Word	63
stlword2	Store Local Word	64
stlword3	Store Local Word	65
stgword1	Store Global Word	66
stgword2	Store Global Word	67
stgword3	Store Global Word	68
stllword1	Store Lexic Level Word	69
stllword2	Store Lexic Level Word	70
stllword3	Store Lexic Level Word	71
storereal	Store Real	72
storeset	Store Set	73
storetag	Store Tag	74
storeufield	Store Unsigned Field	75
storesfield	Store Signed Field	76
copystruct	Copy Structure	78
new	New	79
newinit	New Initialized	80
not	Not	81
andword	And Word	82
intersection	Intersection Set	83
orword	Or Word	84
union	Union Set	85
negword	Negate Word	86
negreal	Negate Real	87
addword	Add Word	88
addreal	Add Real	89
subword	Subtract Word	90
subreal	Subtract Real	91
setdiff	Set Difference	92
mulword	Multiply Word	93
mulreal	Multiply Real	94
divword	Divide Word	95
divreal	Divide Real	96
modword	Modulo Word	97
buildset	Build Set	98
inset	In Set	99
lsword	Less Than Word	100
eqword	Equal Word	101
grword	Greater Than Word	102
nlword	Not Less Than Word	103

neword	Not Equal Word	104
ngword	Not Greater Than Word	105
lsreal	Less Than Real	106
eqreal	Equal Real	107
grreal	Greater Than Real	108
nlreal	Not Less Than Real	109
nereal	Not Equal Real	110
ngreal	Not Greater Than Real	111
eqset	Equal Set	112
nlset	Not Less Than Set	113
neset	Not Equal Set	114
ngset	Not Greater Than Set	115
lsstruct	Less Than Structure	116
eqstruct	Equal Structure	117
grstruct	Greater Than Structure	118
nlstruct	Not Less Than Structure	119
nestruct	Not Equal Structure	120
ngstruct	Not Greater Than Structure	121
funcvalue	Function Value	122
jump	Jump	123
falsejump	False Jump	124
casejump	Case Jump	125
initvar	Initialize Variables	126
pushcall	Push Call	127
call	Call	128
enter	Enter Procedure	130
functionexit	Function Exit	131
exit	Exit	132
enterprog	Enter Program	133
exitprog	Exit Program	134
callstd	Call Standard Routine	135
cfdesc	Construct File Descriptor	136
text	Text	137
pushcallseg	Push Call Segment	138
callseg	Call Segment	139
odd	Odd	140
convword2	Convert Word2 To Real	141
pushrealconst	Push Real Constant	142
buildsetrange	Build Set Range	143
squareword	Square Word	144
squarereal	Square Real	145
round	Round	146
pop	Pop	147
pushstack	Push Stack	149
newline1	New Line	150
newline2	New Line	151
newline3	New Line	152
incrword	Increment Word	153
decrword	Decrement Word	154
truncreal	Truncate Real	160
absword	Absolute Value Word	161
absreal	Absolute Value Real	162
succword	Successor Word	163
predword	Predecessor Word	164

convword Convert Word To Real 165

INDEX

Abort 6-10
Absolute Value Real 5-18
Absolute Value Word 5-14
Absreal 5-18
Absword 5-14
Add Real 5-18
Add Word 5-16
Addreal 5-18
ADDRESS MODIFICATION OPERATORS 5-8
Addword 5-16
And Word 5-20
Andword 5-20
APPENDIX A - OPERATORS IN ALPHABETICAL ORDER A-1
APPENDIX B - OPERATORS IN OP CODE ORDER B-1

BOOLEAN COMPUTATION OPERATORS 5-20
Build Set 5-22
Build Set Range 5-23
Buildset 5-22
Buildsetrange 5-23

Call 5-34
Call Segment 5-34
Call Standard Routine 5-37
Callseg 5-34
Callstd 5-37
Case Jump 5-33
Casejump 5-33
Cfdesc 5-39
Clock 6-9
CODE SEGMENTATION 4-1
Communicate 6-10
CONSTANT POOL 3-4
Construct File Descriptor 5-39
CONTROL OPERATORS 5-33
CONVERSION OPERATORS 5-31
Convert Word to Real 5-32
Convert Word2 to Real 5-32
Convword 5-32
Convword2 5-32
Copy Structure 5-12
Copystruct 5-12

DATA STRUCTURES 3-1
Date 6-9
Decrement Word 5-15
Decrword 5-15

Display 6-10
Divide Real 5-19
Divide Word 5-17
Divreal 5-19
Divword 5-17

End Of File 6-3
End Of Line 6-3
Enter 5-35
Enter Procedure 5-35
Enter Program 5-36
Enterprog 5-36
Eof 6-3
Eoln 6-3
Eqreal 5-26
Eqset 5-30
Eqstruct 5-28
Equal Real 5-26
Equal Set 5-30
Equal Structure 5-28
Equal Word 5-23
Eqword 5-23
Exit 5-35
Exit Function 5-36
Exit Procedure 5-35
Exit Program 5-36
Exitprog 5-36
Expo 6-9
Expressions 3-3

False Jump 5-33
Falsejump 5-33
Field 5-8
Field1 5-8
Field2 5-8
Field3 5-8
FILE HANDLING ROUTINES 6-1
Function Value 5-37
Functionexit 5-36
Funcvalue 5-37

Get 6-1
Get Text 6-1
Gettext 6-1
Greater Than Real 5-27
Greater Than Structure 5-29
Greater Than Word 5-25
Grreal 5-27
Grstruct 5-29
Grword 5-25

HEAP FORMAT 3-3

In Set 5-31

Increment Word 5-15
Incrword 5-15
Index 5-8
Initialize Variables 5-39
Initvar 5-39
Inset 5-31
Intersection 5-21
Intersection Set 5-21
INTRODUCTION 1-1

Jump 5-33

Ldcaddr1 5-1
Ldcaddr2 5-1
Ldcaddr3 5-1
Ldcword1 5-3
Ldcword2 5-3
Ldcword3 5-3
Ldgaddr1 5-2
Ldgaddr2 5-2
Ldgaddr3 5-2
Ldgword1 5-4
Ldgword2 5-4
Ldgword3 5-4
Ldladdr1 5-2
Ldladdr2 5-2
Ldladdr3 5-2
Ldlladdr1 5-3
Ldlladdr2 5-3
Ldlladdr3 5-3
Ldllword1 5-5
Ldllword2 5-5
Ldllword3 5-5
Ldlword1 5-4
Ldlword2 5-4
Ldlword3 5-4
Less Than Real 5-26
Less Than Structure 5-28
Less Than Word 5-24
Linkage Information 3-3
LOAD ADDRESS OPERATORS 5-1
Load Constant Address 5-1
Load Constant Word 5-3
Load Global Address 5-2
Load Global Word 5-4
Load Indirect Word 5-5
Load Lexic Level Address 5-3
Load Lexic Level Word 5-5
Load Local Address 5-2
Load Local Word 5-4
LOAD NON-WORD OPERATORS 5-6
Load Signed Field 5-6
Load Unsigned Field 5-6
LOAD WORD OPERATORS 5-3

Loadind 5-5
Loadsfield 5-6
Loadufield 5-6
Lsreal 5-26
Lsstruct 5-28
Lsword 5-24

Mark 6-8
MEMORY ORGANIZATION 3-1
MISCELLANEOUS OPERATORS 5-37
Modulo Word 5-17
Modword 5-17
Mulreal 5-19
Multiply Real 5-19
Multiply Word 5-16
Mulword 5-16

Negate Real 5-18
Negate Word 5-14
Negreal 5-18
Negword 5-14
Nereal 5-26
Neset 5-30
Nestruct 5-28
New 5-38
New Initialized 5-39
New Line 5-38
Newinit 5-39
Newline1 5-38
Newline2 5-38
Newline3 5-38
Neword 5-24
Ngreal 5-27
Ngset 5-31
Ngstruct 5-29
Ngword 5-25
Nlreal 5-27
Nlset 5-30
Nlstruct 5-29
Nlword 5-24
NON-FILE ROUTINES 6-8
Not 5-20
Not Equal Real 5-26
Not Equal Set 5-30
Not Equal Stucture 5-28
Not Equal Word 5-24
Not Greater Than Real 5-27
Not Greater Than Set 5-31
Not Greater Than Structure 5-29
Not Greater Than Word 5-25
Not Less Real 5-27
Not Less Than Set 5-30
Not Less Than Structure 5-29
Not Less Than Word 5-24

NUMERIC COMPUTATION 5-14

Odd 5-25
OPERATORS 5-1
Or Word 5-21
Orword 5-21

Page 6-6
Pcaddr1 5-1
Pcaddr2 5-1
Pcaddr3 5-1
Pcword1 5-3
Pcword2 5-3
Pcword3 5-3
Pgaddr1 5-2
Pgaddr2 5-2
Pgaddr3 5-2
Pgword1 5-4
Pgword2 5-4
Pgword3 5-4
Pladdr1 5-2
Pladdr2 5-2
Pladdr3 5-2
Plladdr1 5-3
Plladdr2 5-3
Plladdr3 5-3
Pllword1 5-5
Pllword2 5-5
Pllword3 5-5
Plword1 5-4
Plword2 5-4
Plword3 5-4
Pointer 5-13
Pop 5-37
Position 6-2
Predecessor Word 5-15
Predword 5-15
Push Real 5-7
Push Real Constant 5-7
Push Set 5-7
Push Stack 5-38
Push Structure 5-6
Pushcall 5-34
Pushcallseg 5-34
Pushreal 5-7
Pushrealconst 5-7
Pushset 5-7
Pushstack 5-38
Pushstruct 5-6
Put 6-2
Put Text 6-2
Puttext 6-2

Range 5-13

Read Character 6-4
Read Integer 6-4
Read Line 6-6
Readc 6-4
Readi 6-4
Readln 6-6
Reassign 6-7
REGISTERS 2-1
Reinstate Message 6-10
Reinstatemessage 6-10
RELATED DOCUMENTS 1-1
RELATIONAL OPERATORS 5-23
Release 6-8
Relinguish 6-7
Remain 6-7
Remove 6-7
Reset 6-3
Reset Text 6-3
Resettext 6-3
Retain 6-6
Rewrite 6-4
Rewrite Text 6-4
Rewritetext 6-4
Round 5-32

SET COMPUTATION OPERATORS 5-21
Set Difference 5-22
Setdiff 5-22
Square Real 5-20
Square Word 5-17
Squarereal 5-20
Squareword 5-17
STACK FORMAT 3-2
STANDARD ROUTINES 6-1
Stgword1 5-9
Stgword2 5-9
Stgword3 5-9
Stllword1 5-10
Stllword2 5-10
Stllword3 5-10
Stlword1 5-9
Stlword2 5-9
Stlword3 5-9
Store Global Word 5-9
Store Indirect Word 5-10
Store Lexic Level Word 5-10
Store Local Word 5-9
STORE NON-WORD OPERATORS 5-10
Store Real 5-10
Store Set 5-11
Store Signed Field 5-12
Store Tag 5-11
Store Unsigned Field 5-12
STORE WORD OPERATORS 5-9

Storeind 5-10
Storereal 5-10
Storeset 5-11
Storesfield 5-12
Storetag 5-11
Storeufield 5-12
Subreal 5-19
Subtract Real 5-19
Subtract Word 5-16
Subword 5-16
Successor Word 5-14
Succword 5-14

Text 5-40
Time 6-9
Truncate Real 5-31
Truncreal 5-31

Union 5-21
Union Set 5-21
Update 6-2

VALIDATION OPERATORS 5-13
Variables 3-2
Variant 5-13

Write Boolean 6-8
Write Character 6-5
Write Integer 6-5
Write Line 6-6
Write String 6-5
Writeb 6-8
Writec 6-5
Writei 6-5
Writeln 6-6
Writes 6-5