ASSEMBLER REFERENCE MANUAL

XEROX

610E00290
December, 1986

Xerox Corporation
Information Systems Division
XDE Technical Services
475 Oakmead Parkway
Sunnyvale, CA 94086

TABLE OF CONTENTS

(This page intentionally blank.)

1. Network components

The Mesa assembler is a program that takes as input a file containing assembly code and outputs an object file that may be loaded directly by the Mesa loader or bound into a larger object file by a linker. Object files all have the same format. This allows object files created from different source languages to be bound together. Programs in one language can thus easily access global variables and procedures written in another language. The assembler provides protection so that the rules of object-oriented programming may be enforced. Programs written in other languages are expected to make frequent use of the extensive libraries written in Mesa (the window system, for instance).

The major goal of the assembler is to be independent of any high-level language. A portable compiler for any language should find the instruction set of the assembler complete enough that porting the compiler is simple. The assembler is intended to be used in the porting of the Berkeley C and Pascal portable compilers to the Xerox Development Environment that runs on the Xerox 8010. If the machine architecture changes, then changes to the ported compilers should be limited to the assembler.

The assembler should be easy for compiler writers to use. That is, the assembly language should be readable and should hide some of the peculiarities of the Mesa machine architecture. The Mesa architecture is described in detail in the *Mesa Processor Principles of Operation* (PrincOps). The assembler instruction set is a simplified version of the PrincOps instruction set. The assembler is, to some extent, machine independent in that it produces code that can run on any PrincOps machine.

The assembler also eases the compiler writer's task by doing a significant amount of optimization on the assembly program so the compiler may produce very naive code,and the performance of the final code will be reasonably good. The assembler performs peephole optimization, crossjumping, unreachable code elimination, and other optimizations. No global program optimizations are performed.

The reader is assumed to be intimately familiar with the PrincOps. PrincOps code for specific programs may be examined by running the Lister on an object file.

(This page intentionally blank.)

The object file that the assembler produces is called a bcd file. The format of bcds rarely changes and is guaranteed to be the same for every bcd in a release. The bcd contains enough information so that the linker and the loader can do their job. The linker's job is resolving references to imports and exports of the modules being linked and combining the code segments of the modules into a single file. The loader's job is resolving any imports and exports of the bcd with any bcds previously loaded and allocating memory space for global frames and code. This section gives some general rules for writing assembly code that creates a valid bcd.

2.1 Sops and pseudoops

The assembler instruction set consists of two types of instructions, sops and pseudoops. Sops are converted to one or more PrincOps instructions (PrincOps instructions are known as "mopcodes"). Sops are instructions like "pop" or "add". Pseudoops contain information needed by the linker and loader as well as information needed to build a symbol table.

2.2 Files

Filenames may be referred to by just their textual name or by their unique id (uid). A uid is a name and version stamp. It is expected that for C and Pascal, version stamps will not be necessary, except where they access Mesa interfaces. In that case the Mesa interfaces do not need to be opened to obtain their stamps. The linker uses version stamps to guarantee consistency across interfaces. It is up to the compiler to choose whether it wants version checking or not. Consistency of version stamps within an assembly program is not checked. If two uids with the same name but different stamps appear in an assembly program, the first stamp seen is used.

2.3 Imports

Imports allow a module to access variables, procedures, and constants in other modules. The **import** pseudoop is a convenience for specifying a version stamp for a particular imported file and is only needed to import from Mesa interfaces. Imported items from Mesa interfaces may be

referred to as "Interface.Item". Imported items from non-Mesa modules may be referred to as "?.Item". If a **lfc, read,** or **write** instruction uses a name that is undefined, then the name is assumed to be imported and a link is generated. Constants are imported just as variables are, but they are stuffed directly into the code and no links are generated for them. Symbols for imported items are always copied to the new object file.

2.4 Exports

Modules may make procedures or variables in their outer scope available for import by other modules by exporting them. Many languages do not have the notion of explicit exports. Such languages need only put an **export all** pseudoop in assembly programs, which automatically exports every procedure and variable in the global scope. To export to a Mesa interface, the pseudoops **exportvar** and **exportproc** are available for selectively exporting particular variables or procedures. If those pseudoops are used, then for each file exported to there must be an **export** pseudoop that gives the total number of items in the definitions file as well as an optional time stamp.

2.5 Stack

For machine independence, an infinite stack should be provided by the assembler so that the compiler need not worry about stack overflow. However, this is quite complicated, and it is more reasonable for the compiler to worry about it. The compiler therefore must know that the actual number of registers is 14, and should save the stack if it overflows. The compiler should keep a stack model to prevent stack underflow or overflow.

2.6 Frames

For local frames, the frame size is specified in the **entry** pseudoop. For global variables, the assembler automatically computes the amount of space necessary. The **gbyte, gword,** and **gblock** pseudoops allocate space for global variables. The maximum size of a local frame or a global frame is 4092 words.

2.7 Block structure and scoping

An assembly program has a block structure similar to a high-level program. There are four types of blocks: the program block, entry blocks, nested blocks, and unnamed blocks. All code between the beginning and end of a block is part of the block, except for code contained in inner blocks. The first

pseudoop in an assembly program must be a **program** pseudoop, and there is only one such program block. Entry blocks declared with the **entry** pseudoop (usually procedures in the outer scope) are callable and have their own local frame. Nested blocks declared with the **nested** pseudoop (usually nested procedures) are callable and have their own local frame. They may also access the local frame of their parent, where the parent is the nearest surrounding block with a frame. Unnamed blocks declared with the **begin** pseudoop are not callable and share the local frame of their parent. Unnamed blocks are used to help build the symbol table and have no effect on the code generated. All types of blocks end with the **end** pseudoop.

The assembler tries not to enforce any unnecessary scoping or pseudoop placement. For instance, global declarations, imports, and exports may be located anywhere in a program with the same effect. Code constants are gathered and put out before the code for the nearest enclosing procedure.

2.8 Addressing

PrincOps instructions do not have addressing modes. There are more instructions than there would be if there were addressing modes. The number of instructions was reduced for the assembler by allowing different addressing modes. It was felt that this would make the sops more mnemonic and uniform for compiler writers.

There are two forms of addressing, immediate and eventual. They have the forms

immediate: sopname.format length expression
eventual: sopname.format length [base offset] indirection

The format is one of **.r**, **.d**, **.f[num:num]**, **.f[]** or empty. **r** specifies that an argument is in floating-point format. **d** specifies that an argument is a double word. **f** specifies that a field within a word is to be operated on; the first number is the offset of the first bit in the field, and the second number is the length in bits of the field. **f[]** specifies that the field descriptor is on the stack and is not a code argument.

The length is one of **l**, **k**, or empty. **l** specifies that one of the arguments is a long pointer. If **l** is not present, short pointers are assumed. **k** specifies that the argument is a link. Generally the **k** is not used.

In immediate addressing, the expression may be an assembly time constant (for instance, in a load immediate instruction). The expression may be the name of an imported variable or procedure (for instance, in an external function call instruction).

In eventual addressing, the base may be either **lf**, **gf**, or **cb**. **lf** is the start of the local frame. **gf** is the start of the global frame. **cb** is the start of the code segment. The offset has the form " + expression", where expression is an assembly time constant. The offset may be empty, in which case it is assumed to be zero.

For offsets from the global frame, the label of the global declaration may be used in the place of the offset. This is a side effect of having the assembler lay out the global frame. For instance,

```
global1: gblock 2
load.d [gf + global1]
```

The gf may be left out entirely, as in,

```
load.d global1
```

This allows assembly code to be written in ignorance of whether the item is defined within the program or externally. The indirection has the form "↑ + expression", where expression is an assembly time constant. The indirection may be empty.

2.9 Code constants

Raw data, such as jump tables, strings, or default values, may be inserted into the code segment with the **words**, **bytes**, and **string** pseudoops. These constants are gathered and output before the code for the nearest enclosing procedure body. However, they may be referred to anywhere in the program.

2.10 Comments

Comments may be put in an assembly program by the special symbol "--". Any characters between the double hyphen and the next carriage return are ignored. Any characters between two double hyphens on the same line are ignored. Any characters between a ≪ and a ≫, including carriage returns, are ignored. Comments may not be nested.

2.11 Identifiers

All identifiers in an assembly program begin with a letter. They may contain alphanumerics, $, and ?. They may be any length. Case is significant.

2.12 Labels

Labels refer to code locations or global frame locations. Labels may not be duplicated in the same block. Arithmetic is allowed

on global frame labels, as if they were an offset from the global frame base, but not on jump labels.

2.13 Constants

Constants may be defined anywhere in the program by a pseudoop of the form "constantname = constantexpression". Complex expressions (those involving an arithmetic operator) may not have forward references to constants, though they may have backward references. The expression may forward reference one level if the constant expression has no operators. Thus

LEGAL

```
b = 3
c = 4
a = b + c

a = b
b = 3
```

ILLEGAL

```
b = 3
a = b + c          -- forward reference in expression
c = 4

a = b
b = c
c = 3              -- forward reference two levels
```

2.14 Expressions

The allowed arithmetic operators are +, -, *, and /. If the dividend mod the divisor is not zero, the answer is floored. No warnings are given for overflow and underflow. Operators are all left associative with standard precedence. Parentheses may be used to get precedence other than left to right. Unary minus is allowed, but binary minus takes precedence.

(This page intentionally blank.)

Every sop and pseudoop must terminate with a carriage return.

```
 1   0 goal       ::= . program ;

 2   1 program    ::= programop eol statementlist

 3   2 statementlist  ::= pseudoop eol statementlist
 4   3              | eol
 5   4              |

 6   5 pseudoop   ::= id = exp
 7   7              | include filename
 8   8              | label entry exp , id
 9   9              | label nested exp , exp , id
10  10              | begin
11  11              | id : gword
12  12              | id : gbyte
13  13              | id : gblock num
14  14              | end
15  15              | source num
16  16              | id : var id , num , num
17  17              | opaque id , filename
18  18              | import filename
19  19              | export filename , num
20  20              | exportvar id , filename , num
21  21              | exportproc id , filename , num
22  43              | typedef
23  44              | export all
24  45              | id : string string
25  46              | id : bytes bytes
26  47              | id : words words
27  48              | align num
28  49              | sop
29  50              | label
30  51              | label sop

31    6  programop       ::= program id , objectStamp ,
creatorStamp , sourceFile , id

32  22 typedef     ::= id : type id
33  23              | id : type enum { enumeratedlist }
34  24              | id : type record [ recordlist ]
35  25              | id : type proc [ recordlist ] returns [ recordlist ]
36  26              | id : type long readonly pointer to id
37  31              | id : type packed array id of id
38  34              | id : type subrange [ num , num ] of id
39  35              | id : type constant id words

40  27 long        ::=
41  28              | long

42  29 readonly    ::=
43  30              | readonly
```

```
44  32 packed      ::=
45  33           | packed

46  36 enumeratedlist ::= enumeratedlist , eitem
47  37           | eitem

48  38 eitem       ::= ( num : id )

49  39 recordlist  ::= recordlist , ritem
50  40           | ritem
51  41           |

52  42 ritem       ::= ( id , id , num , num )

53  52 label       ::= id :

54  53 bytes       ::= bytes exp
55  54           | exp

56  55 words       ::= words exp
57  56           | exp

58  57 filename    ::= uid
59  58           | id

60  59 uid         ::= ( id , stamp )

61  60 sourceFile  ::= ( string , stamp )

62  61 creatorStamp ::= stamp

63  62 objectStamp ::= stamp

64  63 stamp       ::= ( num , num , lnum )
65  64           | ( num , num , num )

66  65 sop         ::= sid effectiveaddr
67  66           | sid . r effectiveaddr
68  67           | sid . d effectiveaddr
69  68           | sid . f [ num : num ] effectiveaddr

70  69 effectiveaddr ::=
71  70           | [ base offset ] indirection
72  71           | length [ base offset ] indirection
73  72           | length exp
74  73           | exp
75  74           | id . id
76  75           | length id . id

77  76 length      ::= l
78  77           | k

79  78 base        ::= lf
80  79           | gf
81  80           | cb

82  81 offset      ::= + exp
83  82           |

84  83 indirection ::= offset
85  84           |

86  85 exp         ::= primary
87  86           | ( exp )
```

```
88  87          |exp '- primary
89  88          |exp + primary
90  89          |exp / primary
91  90          |exp * primary

92  91 primary     ::= num
93  92          |num
94  93          |id
```

(This page intentionally blank.)

An object file contains more information than just the object code. It also contains information about the procedures and block structure of the module, the symbol table, and references to other files besides the one being assembled. The assembler has pseudoops to collect such information. Pseudoops also allow data to be inserted in the code stream. See the grammar for details about particular constructs.

label = exp

The = pseudoop equates the label on the left with the expression on the right. The expression must be a numeric constant. Constants may be forward-referenced one level only, and forward references may not occur in expressions. Example:

```
a = b
b = (12/4) + 8
```

program id , objectStamp , creatorStamp , sourceFile, typelabel

There is only one **program** pseudoop in an assembly code file. It must be the first pseudoop. It specifies the name of the module, a stamp for the object file, and a stamp for the creator of the assembly code file (usually the compiler). The **sourceFile** is the name of the high level language source code for the assembly code, enclosed in quotes. The **typelabel** is the label giving the type of the program. **Stamps** are specified by three numbers enclosed in parentheses, for example (5,6,12345). The first two numbers must be in the range [0..255] and are net and host numbers (these numbers are currently ignored by system programs). The last number is a **LONG CARDINAL** and is a **System.GreenwichMeanTime** that can be obtained from the Time interface. Example:

```
program InsertionSort, (0, 170, 2652982585), (0, 0,
2652981290), ("InsertionSort.mesa", (0, 0, 2652775726))
```

The program pseudoop is the only pseudoop in which time stamps are required. For any other pseudoop, a file may be specified by name without a version stamp and without parentheses. The assembler will do version checking for whatever time stamps are given. It is up to the compiler to choose whether it wants version checking or not.

include filename

The **include** pseudoop causes the assembler to act as if the file were a Mesa interface, so that there is a

dependency between the object file and the include file. Example:

```
include ("stdio.h", (0,0,0))
```

label : entry framesize, typelabel
label : nested framesize, frameoffset, typelabel
begin
end

The **entry** pseudoop should be placed at the start of an entry block (a procedure). **label** is the name of the procedure; the **framesize** is in words; the **typelabel** is the label giving the type of the procedure. The **nested** pseudoop should be placed at the start of a nested block (a nested procedure). The **frameoffset** is the offset of the procedure descriptor in the frame of the parent (see the programming hint on static links below). The **BEGIN** pseudoop should be placed at the start of an unnamed block. Unnamed blocks are different from procedures. They may not be called by using **XFER**, and they do not have frames, so any local frame references use the frame of their parent. The END pseudoop is used to end a block. Procedures may be nested up to seven levels deep, and blocks may be nested arbitrarily deep.Example:

```
P: entry 5
N: nested 5, 6
begin
end
end
end
```

label :gword
label :gbyte
label : gblock num

These pseudoops are for allocating space in the global frame. **gword** and **gbyte** both allocate a word (no packing is implied by **gbyte**). **gblock** allocates num words. The labels may be used to reference the locations. Example:

```
g1:gword
store.d [gf + g1]
```

source location

The **source** pseudoop maintains a mapping between source code and object code. The location is a character position in the source file. The **source** pseudoop is optional and is used by the debugger for setting breakpoints. Wherever there is a **source** pseudoop, a source-level breakpoint may be set. Example:

```
source 800
```

import filename

The **import** pseudoop specifies a version stamp for an imported file. If no stamp is given in the file name, then a null stamp is assumed. Example:

```
import (Exec, (0, 170, 2604877475))
```

export filename, itemcount

The **export** pseudoop should be used for each Mesa definitions file that is exported to. The **itemcount** is the total number of items in the interface.

exportvar offsetLabel , filename , itemNumber

The **exportvar** pseudoop exports a variable. If two modules wish to share a variable, then this pseudoop makes the variable available for another module to import. The variable may be shared through filename, which is a Mesa interface file. If the variable is not shared through an interface file (which is the usual case with languages other than Mesa), then the filename may be replaced by a question mark. The **offsetLabel** is the global declaration of the variable. The item is a number that distinguishes exports to a particular interface. The item numbers should lie between zero and the total number of exports for the interface minus one. (Item numbers imply that the interface must be opened by the compiler in order to export to it. This is intentional.) If the **filename** is a question mark, then the **itemNumber** should be zero. Example:

```
exportvar globalvar, (StripDefs, (
      0, 170, 2604877475)), 3
exportproc entryLabel , filename , itemNumber
```

The **exportproc** pseudoop exports procedures. The procedure is shared through **filename**, which is an interface file. The **entryLabel** is the label of the procedure. The **itemNumber** tells which export to filename this procedure is. **filename** has to be opened to look up the **itemNumber** of **entryLabel**. If **filename** is not a Mesa interface, it should be replaced by a question mark, and the should be zero. Example:

```
exportproc StripComments, (StripDefs, (0, 170,
      2604877475)), 3
export all
```

The **export all** pseudoop is equivalent to "**exportproc** name,?,0" for every global procedure and an "**exportvar** name,?,0" for every global variable. It is expected that the **export all** pseudoop will be used for C and Pascal.

label : string string

The **string** pseudoop places a string in the object code. The characters contained within the quoted string are passed to the output file. Each character is converted to

ASCII and occupies a byte. A backslash may be used to generate certain ASCII characters:

\n, \N, \r, \R	carriage return
\t, \T	tab
\b, \B	backspace
\f, \F	form feed
\l, \L	line feed
\\	backslash
\'	single quote
\"	double quote
\000	null

Any ASCII character can be inserted into the string using a backslash followed by an octal digit, as is the null character above. The **bytes** pseudoop may also be used. The string may be referred to by using the label. For instance label + 0 refers to the first character of the string, label + 1 refers to the second character, and so on, as in

```
ss: string "hi there"
load [cb + (ss + 3)
```

label : bytes num , ...
label : words num , ...

The **bytes** and **words** pseudoops place data into the object code. The numbers following the keyword are separated by commas. For the **bytes** pseudoop, the numbers must lie between 0 and 255. Each number is placed in a consecutive byte. For the **words** pseudoop, the numbers must lie between 0 and 65535. Each number is placed in a consecutive word.

```
jumptable: words 2, 40, 24
```

table num

The **table** pseudoop marks the beginning of a jump table in the code. The pseudoop should be followed by **num** **jumpc** instructions, and finally a single **jumpca** instruction. To construct a jump table, first load the maximum index value (the limit) and then the actual index. Next, for each value in the index range, give a **jumpc** instruction with a target label as the code for the corresponding case arm. Finally, give a **jumpca** to an endcase label. The code offsets should be in a separately defined code constant.

label: var typelabel, size , frameoffset

The **var** pseudoop specifies the types of variables. The **typelabel** points to the type of the variable. The **size** is the size in bits of the variable. The **frameoffset** is the offset in the local frame of the variable in bits. **var** pseudoops are optional but must be used to allow symbolic debugging. Variables may be placed in the scope of a procedure or block by inserting an appropriate **var** pseudoop after the **entry** or **begin** and before the start of an enclosed procedure or block.

label: type label

Defines a named type. The first label becomes an alias of the second within the current block.

label: type enum { (num : id) , ... }

Defines an enumerated type. Each element of the enumerated type has a number and a name.

label: type record [(fieldname, fieldtype, offset, size), ...]

Defines a record type. Each field of the record is enclosed in parentheses, and gives the name of the field, its type, its bit offset within the record, and its bit size.

label: type proc [(itemname, itemtype, offset, size), ...]
returns [(itemname, itemtype, offset, size), ...]

Defines a procedure type. The parameters and return values are in the same format as records.

label: type pointer to label
label: type long pointer to label
label: type readonly pointer to label
label: type long readonly pointer to label

Defines a pointer. The **long** keyword should be used if the pointer is a doubleword. The **readonly** keyword should be used only if the compiler has enforced it.

label: type array indexlabel of componentlabel
label: type packed array indexlabel of componentlabel

Defines an array with the given index type and component type. The index type should be a subrange or a basic type.

label: type subrange [lowerindex, range] of typelabel

Defines a subrange with lowest element and range, which must be numbers (this implies that the compiler know the numbers corresponding to the subrange elements). The type of each element in the subrange is given by the **typelabel**.

label: type constant typelabel, word, word, ...

Defines a constant of the given type containing the given words. The size of the type should match the number of words given. The words of the constant should be laid out according to its type.

label: type union (unimplemented)

Defines a variant type.

type label , filename (unimplemented)

The **type** pseudoop exports opaque types that the module has defined. The **label** is the label of the pseudoop that describes the type to be exported. The **filename** is the object file where the concrete type is actually defined-- in other words, the file where the type has a symbol table entry. The **type** pseudoop is not needed for languages besides Mesa.

5. Programming Hints

This section provides some useful hints about assembly language programming.

5.1 Static links

See Mesa Compiler Internal Documentation, Section 5.2. When you allocate space for a procedure descriptor of a nested procedures, the descriptor must be in a location whose address is 2 MOD 4. This is so that the address of the procedure descriptor is interpreted as an indirect control link by the xfer instruction. For example, say a procedure P contains two nested procedures, R and S. P would have the following code to initialize the procedure descriptors for R and S:

```
lea [gf + 1]          -- must be odd, GF is always 0 MOD 4
lea [cb + R]
store.d [lf + 2]      -- address must be 2 MOD 4, LF is
                      -- always 0 MOD 4
lea [gf + 1]
lea [cb + S]
store.d [lf + 6]      -- address must be 2 MOD 4
```

Note that we load the address GF + 1 so that the control link is interpreted as a procedure descriptor. A procedure call from P to R would look like:

```
lea [lf + 2]          -- get address of descriptor
load 0                -- fill the link
sfc
```

This code loads the address of the procedure descriptor for R and then does an XFER with that address as the destination. The address is treated as an indirect control link. After the XFER is complete, the address is left below the stack. By executing an LK instruction, the address can be recovered. The local frame of P can then be had by subtracting the offset of R's procedure descriptor in P's frame. So the first code in R should be:

```
lk 2
```

This puts the static link to P's frame in local 0. Subsequent access to P's variables can be made with

```
read [lf + 0] + n     -- n is the offset of a variable
```

If R made a call to S, the code would be just slightly different. S has to get the address of R's procedure descriptor from P's frame:

```
load [lf + 0]      -- static link to P's frame
load 2
add                -- get address of descriptor
load 0             -- fill the link
sfc
```

5.2 Parameter passing

The stack has only 14 registers. For expression evalution, all of the registers may be used. However, for procedure calls, a maximum of 12 registers can be used to pass arguments, so that the **xfer** information can be recovered if it is needed. If more than 12 words are needed, the compiler should allocate a frame (using the **af** sop) to pass the parameters in. The callee should copy the arguments into its own frame and free the argument frame (using the **ff** sop).

6. Sops

The table on the following pages lists all of the sops and the legal forms of address for each sop. Many of the sops have only one form of address, while some, such as read and write, have almost all possible forms of address.

OpCode	Name	Meaning	Address and Operand Formats										
			Val	GF	LF	CB	.D	.F	.R	L	K	Ind	#
ACD	Add Cardinal to Double	Stack has 16-bit data1, 32-bit data2. Push data1 + data2.											
ADC	Add Double to Cardinal	Stack has 32-bit data1, 16-bit data2. Push data1 + data2.											
ADD	Addition	Stack has data1, data2. Push data1 + data2.											
							X						
									X				
AF	Allocate Frame	Stack has size. Allocate frame of that size and push 16-bit address of frame.											
AMUL	Multiply without shifting.	Just like MUL except it won't be converted to a shift instruction.											
AND	Logical AND	Stack has data1, data2. Push data1 ∧ data2.											
							X						
BC	Broadcast Condition	Wake up all processes on the condition queue and reschedule if any awakened.											
BITBLT	Bit Block Transfer	Stack has pointer to parameters record. Performs as many operations on rectangular areas in memory. Used mostly in conjunction with bitmap display data.											
BLT	Block Transfer	Block transfer											
						X							
						X				X			
										X			
BLTET	Block Transfer Equality Test	Stack has 32-bit pointer1, 16-bit count, 32-bit pointer2. Compare count words beginning at pointer1 and pointer2; push1 if all words are equal, 0 otherwise.				X				X			
										X			
BNDCK	Bounds Check	Stack has value, limit. IF value ~ IN [0..limit) THEN trap ELSE discard limit.											
										X			

OpCode	Name	Meaning	Address and Operand Formats										
			Val	GF	LF	CB	.D	.F	.R	L	K	Ind	#
BRK	Breakpoint	IF resuming from debugger THEN execute broken opcode ELSE trap.											
CATCH	Catch code follows.	Two byte no-op used by Mesa runtime system. Operand is the catch entry vector position of the catch code.											
CATCHFS	Catch Frame Size	Operand is the frame size for catch code.											
CMP	Compare	Stack has data1, data2. Compare data1 and data2 (signed) and push -1 if data1<data2; 0 if data1 = data2; + 1 if data1>data2.					X						
									X				
DBL	Double	Stack has data. Push data*2.											
							X						
DEC	Decrement	Stack has data. Push data-1.											
DI	Disable Interrupts	Increment the WakeupDisableCounter thus disabling any interrupt processing and process timeouts. Trap if the counter overflows.											
DIS	Discard	Discard the top value on the stack, i.e., decrement the stack pointer by the size of the value.											
							X						
DIV	Signed Division	Stack has data1, data2. Push quotient and remainder from data1/data2; decrement the stack pointer so as to leave the remainder above the stack.											
									X				
							X						
DSK	Dump Stack	Dump the evaluation stack and stack pointer starting. No more than two values above the top of stack need be stored.			X								
DUP	Duplicate	Duplicate the top value on the stack.											
							X						

OpCode	Name	Meaning	Address and Operand Formats										
			Val	GF	LF	CB	.D	.F	.R	L	K	Ind	#
EFC	External Function Call	Fetch link alpha as described in LLKB and XFER to it.	X										
EI	Enable Interrupts	Decrement the WakeupDisableCounter; if the new value is zero, interrupts are now enable. Trap if the counter underflows.											
EXCH	Exchange	Interchange the top two values on the stack.											
							X						
EXDIS	Exchange Discard	Equivalent to the sequence ESCH; DIS.											
FF	Free Frame	Stack has pointer to frame. Put frame in free list.											
FIX	Fix	Stack has 32-bit real. Push 32-bit integer.											
FIXC	Fix to Cardinal	Stack has 32-bit real. Push 16-bit unsigned number.											
FIXI	Fix to Integer.	Stack has 32-bit real. Push 16-bit integer.											
FLOAT	Float	Stack has 32-bit integer. push equivalent floating point.											
GMF	Get Map Flags	Stack has 32-bit virtual page. Push the flags of the indicated virtual page.											
INC	Increment	Stack has data. Push data + 1.											
							X						
IOR	Inclusive OR	Stack has data1, data2. Push data1 $\vee$ data2.											
							X						
KFC	Kernel Function Call	Fetch link from SD[parm] and XFER to it.	X										
JUMP	Unconditional Jump	Unconditional jump to a label.	X										
JUMPE	Jump Equal	Stack has data1, data2. Jump if data1 = data2.	X										
JUMPN	Jump Not Equal	Stack has data1, data2. Jump if data1 $\neq$ data2.	X										

OpCode	Name	Meaning	Address and Operand Formats									
			Val	GF	.LF	CB	.D	.F	.R	L	K	Ind	#
JUMPL	Jump Less Than	Stack has data1, data2. Jump if data1 < data2.	X										
JUMPLE	Jump Less Than or Equal To	Stack has data1, data2. Jump if data1 ≤ data2.	X										
JUMPG	Jump Greater Than	Stack has data1, data2. Jump if data1 > data2.	X										
JUMPGE	Jump Greater Than or Equal To	Stack has data1, data2. Jump if data1 ≥ data2.	X										
UJUMPL	Unsigned Jump Less Than	Stack has data1, data2. Jump if data1 < data2. Unsigned comparison.	X										
UJUMPLE	Unsigned Jump Less Than or Equal To	Stack has data1, data2. Jump if data1 ≤ data2 Unsigned comparison.	X										
UJUMPG	Unsigned Jump Greater Than	Stack has data1, data2. Jump if data1 > data2 Unsigned comparison.	X										
UJUMPGE	Unsigned Jump Greater Than or Equal To	Stack has data1, data2. Jump if data1 ≥ data2. Unsigned comparison.	X										
JUMPC	Jump Case	Jump table entry	X										
JUMPCA	Jump Case Always	Unconditional jump at end of jump table	X										
JUMPRET	No return	The previous instruction cannot return. Used by the optimizers to delete unreachable code.											
LEA	Load Effective Address	Load the memory address of the operand.		X									
					X								
						X							
LFC	Local Function Call	Do the last half of XFER (frame allocation and set new PC) using operand as the PC of the new procedure.	X										
LINT	Lengthen Integer	Stack has 16-bit data. Sign extend data to 32-bits.											

OpCode	Name	Meaning	Address and Operand Formats										
			Val	GF	LF	CB	.D	.F	.R	L	K	Ind	#
LK	Link	Recover word from above the top of stack; store word – operand in LF + 0. For establishing static links.											
LOAD	Load	Push value at given address on top of stack.		X									
				X			X						
													X
							X						X
					X								
					X		X						
											X		
						X							
LP	Lengthen Pointer	Stack has 16-bit pointer. IF pointer = 0 THEN push 0 ELSE push MDS.											
LSK	Load Stack	Load the evaluation stack and stack pointer from given location. No more than two values above the top of stack need be loaded.			X								
ME	Monitor Enter	Stack has 32-bit pointer to monitor lock. IF the monitor is unlocked THEN lock it ELSE enqueue current process on the monitor queue and reschedule.											
MOD	Modulus	Stack has data1, data2. Push data1 MOD data2.					X						
MBP	Make Byte Pointer	Stack has long pointer. Push byte pointer to high byte.											

OpCode	Name	Meaning	Address and Operand Formats										
			Val	GF	LF	CB	.D	.F	.R	L	K	Ind	#
MR	Monitor Reenter	Stack has 32-bit monitor pointer, 32-bit condition pointer. IF monitor locked THEN enqueue on monitor queue and reschedule ELSE {test for aborting and trap if appropriate; lock monitor and proceed}.											
MUL	Multiply	Stack has data1, data2. Push the 32-bit value data1*data2. IF data were 16-bit THEN decrement stack pointer ELSE ignore overflow.											
							X						
									X				
MW	Monitor Wait	Stack has 32-bit monitorPointer, 32-bit conditionPointer, 16-bit timeout. Unlock monitor (and wake up waiting process if any); enqueue current process on condition with timeout value; reschedule. Exceptions: IF condition has wakeup waiting OR process has been aborted THEN current process continues to run.											
MX	Monitor Exit	Stack has 32-bit pointer to monitor lock. Unlock the monitor; IF the queue is not empty, wake up the first waiting process and reschedule.											
NC	Notify Condition	Stack has 32-bit conditionPointer. Wake up the first process on the condition queue and reschedule if awakened.											
NEG	Negate	Stack has data. Push 0-data.											
NILCK	NIL Check	Stack has pointer. IF pointer = 0 THEN trap ELSE leave pointer on stack.											
							X						

OpCode	Name	Meaning	Address and Operand Formats										
			Val	GF	LF	CB	.D	.F	.R	L	K	Ind	#
PI	Port In	Recover source and portAddress from above stack. Store 32-bit zero at portAddress + 0; IF source#0 THEN extend it with a high order zero and store at portAddress + 2 and portAddress + 3.											
PO	Port Out	Stack has portAddress. Store L at portAddress + 0; fetch link from portAddress + 2 and portAddress + 3 and XFER to it with a source of portAddress.											
POR	Port Out Responding	Identical to PO. The distinction between the two is used by a trap handler to decide how to recover from port faults.											
PUT	Put	Stack has data. Store data into effective address; leave data on stack.			X								
					X		X						
PUTS	Put Swapped	Stack has pointer, data. Store data into pointer + operand; leave pointer on stack (but not data).	X										
			X				X						
			X				X			X			
			X					X					
			X							X			
			X					X		X			

OpCode	Name	Meaning	Address and Operand Formats										
			Val	GF	LF	CB	.D	.F	.R	L	K	Ind	#
READ	Read	Load from effective address. If base is not given, then pointer is on top of stack. Field specifier for "READ .F[] L(n)" may be empty, i.e., it is on the stack. The field specifier for reading from the code must be on the stack.	X										
						X							
						X		X					
			X				X						
			X				X			X			
			X					X					
				X								X	
				X			X					X	
				X						X		X	
											X		
							X				X		
					X								
					X		X					X	
					X		X			X		X	
					X			X					
					X							X	
					X			X				X	
					X					X		X	
					X			X		X		X	
READSTR	Read String	Stack has pointer, index. Fetch word from pointer + (index/2) + operand; IF index MOD 2 = 0 THEN push word.high ELSE push word.low.											
										X			
REC	Recover	Recover data item from above the stack (i.e., increment stack pointer by one or two).											
							X						
REM	Remainder	Floating point only. Perform division and leave remainder on stack.							X				
RO	Read Overhead	Stack has pointer. Load from pointer-operand. All access to local and global frame overhead is done through this instruction (and WO) so that the processor could cache this data.											

OpCode	Name	Meaning	Address and Operand Formats										
			Val	GF	LF	CB	.D	.F	.R	L	K	Ind	#
ROTATE	Rotate	Stack has data, count. Rotate data by count MOD 16 bits; left if count is positive; right if count is negative.											
ROUND	Round	Stack has floating point. Rounds to 32-bit integer											
ROUNDC	Round to Cardinal	Stack has floating point. Rounds to 16-bit unsigned number.											
ROUNDI	Round to Integer	Stack has floating point. Rounds to 16-bit signed number.											
RPB	Read byte pointer	Stack has byte pointer. Add offset. Push byte pointed to.											X
SFC	Stack Function Call	Stack has 32-bit controlLink. XFER to controlLink.											
SHIFT	Shift	Stack has data, count. Shift data by count bits; left if count is positive; right if count is negative.											
							X						
STC	Stack Check	If stack pointer # arg then stackerror											X
STORE	Store	Stack has data. Store into effective address.		X									
				X			X						
					X								
					X		X						
SUB	Subtraction	Stack has data1, data2. Push data1-data2.											
							X						
									X				
TRPL	Triple	Stack has data. Push data*3.											
UCMP	Unsigned Compare	Stack has data1, data2. Compare data1 and data2 (unsigned) and push -1 if data1 < data2; 0 if data1 = data2; + 1 if data1 > data2.					X						

OpCode	Name	Meaning	Address and Operand Formats										
			Val	GF	LF	CB	.D	.F	.R	L	K	Ind	#
UDIV	Unsigned Divide	Stack has data1, data2. Push quotient and remainder form data1/data2; decrement the stack pointer so as to leave the remainder above the stack.											
UMOD	Unsigned Modulus	Stack has data1, data2. Push remainder from data1/data2.					X						
WPB	Write byte pointer	Stack has value, byte pointer. Write value to byte.											X
WRITE	Write	Stack has data. Store into effective address. If no base is given, then pointer is below data on the stack.If writing to a field in the local frame through a long pointer, the field specifier may be on the stack (i.e., "WRITE.F[] (LF + n)").	X										
			X				X						
			X				X			X			
			X					X					
				X								X	
				X			X					X	
				X			X			X		X	
				X						X		X	
				X				X				X	
				X				X		X		X	
			X									X	
				X				X					
				X								X	
				X			X					X	
				X			X			X		X	
				X						X		X	
				X				X				X	
				X				X		X		X	
WRITES	Write Swapped	Like write except pointer is above data on stack.	X										
			X				X						
			X				X			X			
			X							X			
			X					X		X			
			X					X					

OpCode	Name	Meaning	Address and Operand Formats										
			Val	GF	LF	CB	.D	.F	.R	L	K	Ind	#
WRITESTRING	Write String	Stack has data, pointer, index: Fetch word from pointer + (index/2) + parameter; IF index MOD 2 = 0 THEN word.high ' data ELSE word.low ' data; Store word at pointer + (index/2) + parameter.											
										X			
XOR	Exclusive OR	Stack has data1, data2. Push data1 ⊕ data2.											