PERQ Systems Corporation

ACCENT
MICROPROGRAMMING MANUAL

December 7, 1984

This manual is for use with Accent Release S6.

2600 Liberty Avenue
P. O. Box 2600
Pittsburgh, PA 15230
(412) 355-0900

Accent is a trademark of Carnegie-Mellon University.

Accent and many of its subsystems and support programs were originally developed by the CMU Computer Science Department as part of its Spice Project.

The information in this document is subject to change without notice and should not be construed as a commitment by PERQ Systems Corporation. The company assumes no responsibility for any errors that may appear in this document.

PERQ Systems Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

PERQ, PERQ2, LINQ, and Qnix are trademarks of PERQ Systems Corporation.

ACCENT

MICROPROGRAMMING MANUAL

PREFACE

This manual contains documentation on microprogramming with Accent, the new operating system for the PERQ workstation. Accent was developed jointly by PERQ Systems Corporation and the Spice Project in the Computer Science Department at Carnegie-Mellon University. "Spice" is an acronym for Scientific Personal Integrated Computing Environment.

Before you use this manual you should be familiar with the material in the *Accent User's Manual* and the *Accent Programming Manual*.

Other manuals for the Accent operating system are:

- Accent Microprogramming Manual
- Accent Languages Manual
- Accent Lisp Manual
- Accent Qnix Manual (forthcoming in a future release)
- Accent System Administration Manual

Throughout the manuals the term "PERQ" refers to all models of the PERQ workstation unless stated otherwise. When a distinction is made between the PERQ workstation and the PERQ2 workstation, the term "PERQ2" refers to both Model LN-3000 and LN-3500.

The following symbols have been used throughout the Accent manuals:

< >	Material that is to be replaced by symbols or text as explained in the accompanying text. Do not type the angle brackets. Example: <filename> indicates that you should type the name of your file.
[ ]	Optional feature. Do not type the square brackets.
{ }	0 to n repetitions of an optional item. Do not type the braces.
CAPITALS	Literal, to be reproduced exactly as shown (although it may be reproduced in upper-case or lower-case). Example: <filename.CMD> indicates that the filename must contain the extension .cmd.
|	"Or"--choice between the items shown on either side of the symbol.
CTRL	Control key
ESC, INS	Escape key (labeled as ACC ESC or INS on various models)
DEL	Delete key (labeled as REJ DEL on some models)
HELP	Help key
LF	Linefeed Key
RETURN	Carriage return
italics	Input to be typed by the user

ACCENT MICROPROGRAMMING

TABLE OF CONTENTS

MICROPROGRAMMER'S REFERENCE

December 7, 1984

2600 Liberty Avenue
P. O. Box 2600
Pittsburgh, PA 15230
(412) 355-0900

Table of Contents Page

List of Figures

List of Tables

1. Introduction

The PERQ workstation features a microprogrammable CPU. The PERQ workstation microprogramming language is described in this document, along with identification of implementation-dependent behavior. The document assumes that the reader is unfamiliar with the PERQ workstation but has some prior experience with horizontally programmed microengines.

There are three available PERQ CPUs: the PERQ1 CPU, the PERQ1a CPU, and the PERQ1b CPU. Some references to the PERQ1a call it the 16K ControlStore CPU. The PERQ1b is a 24 bit CPU. The PERQ1a and PERQ1b CPUs have all the features of the PERQ1 CPU, with additional features listed below.

The PERQ1 CPU has the following features:

A high-speed microprogrammed processor with a 170 nanosecond microcycle time and 48-bit microinstructions.

20-bit wide data paths: 16 data bits (bits 0..15), and four bits that are used to calculate real addresses.

A 1 megaword addressing space.

The following additional features are available in the PERQ1a CPU:

16K writable control store.

A 14 bit computable Goto with the address coming from the processor shift output.

Single precision multiply step and divide step hardware.

A base register for addressing the X and Y registers.

A readable victim latch.

Ability to use a long constant in a microinstruction which pushes the expression stack.

The PERQ1b CPU includes all of the features of the PERQ1a, plus the following additional features:

X, Y, Estack, Madr, MDX, and ALU are extended to be 24 bits wide.

Adds a new AMux source called Upper to read X[23:16].

Changes the C19 test to C23 (see Section 3.4.1.4).

The syntax is described with a meta-language called EBNF (extended BNF). The following meta-symbols are used.

' ' - surround literal text.

| - separate alternatives.

[ ] - surround optional parts.

{ } - surround parts which may be repeated zero or more times.

( ) - group items.

. - end a description.

1.1. Format

Type programs in free format; a single micro-instruction can extend to as many lines as desired. Blank lines and lines consisting only of comments can be inserted anywhere. The exception to this rule is that a new micro-instruction must begin with a new line; you cannot have more than one instruction per line.

1.2. Names

Names can be any length, but only 10 characters are significant when two names are compared.

1.3. Comments

Indicate comments by an exclamation mark (!). The remainder of the line following the exclamation mark is ignored. Comments can also be enclosed in braces Pascal style: '{' and '}' or '(*' and '*)'.

1.4. Constant Expressions

Expressions are allowed in most places where numeric values are required. In a few instances, only numbers or named constants are allowed and expressions are not. The syntax of constant expressions mimics that of Pascal. Expressions consist of operators and operands with certain precedence rules. Parentheses are used for controlling the order of evaluation.

As in Pascal, operators fall into one of three precedence classes: Multiplying operators have the highest priority, adding operators are next, and relational operators have the lowest priority. The multiplying operators are:

*	Signed integer multiply
div	Signed integer divide
mod	Signed integer remainder

and	Bitwise logical product
nand	Inverted bitwise logical product
lsh	Left shift
rsh	Right shift
rot	Right rotate

Lsh, rsh, and rot shift their lefthand operands the number of bits specified by their righthand operands.

The adding operators are:

+	Signed integer sum
-	Signed integer difference
or	Bitwise logical sum
nor	Inverted bitwise logical sum

The relational operators are:

=	Signed integer equal-to
<>	Signed integer not-equal-to
<	Signed integer less-than
>	Signed integer greater-than
<=	Signed integer less-than-or-equal-to
>=	Signed integer greater-than-or-equal-to
xor	Exclusive-or
xnor	Inverted exclusive-or

Xor and xnor are considered to be relational operators because they perform bitwise equality and inequality operations. The integer comparison operators return 0 for false and 1 for true.

In addition to the three precedence classes, three unary operators are available:

+	Unary integer identity
-	Unary integer negation
not	Unary bitwise complement

The unary identity and negation fall between the adding operators and the multiplying operators in precedence. The unary complement is higher priority than the multiplying operators.

Constant expressions are computed using 16-bit arithmetic and no overflow checks are applied (with the exception of a check for division by zero). In order to eliminate certain syntactic ambiguities, constant expressions must often be surrounded by parentheses. For example, the expression

```
not 1
```

could be interpreted as an ALU expression or a constant expression. For example,

```
R := not 1;    ! complement 1 at execution time
R := (not 1);  ! complement 1 at assembly time
```

The first form is preferable for constants that are less than 255 because the assembler can use a short constant, whereas the second form requires a long constant. This can be used to create a 16-bit value with a short constant:

```
R := not (not 177400);  ! mask 177400 is formed
                        ! by the short constant 377
```

1.5. Syntax

Note that Chapter 6 also contains a pictorial representation of the EBNF syntax given below.

```
name = letter {letter | digit} .

number = ['2#' | '8#' | '10#' | '#' ] digit {digit} .

constant  = name | number .

register = ['%'] name .
         | '%' constant .
         | '%' '(' ConstExpr ')' .

label = name .

empty = .


microprogram = {Instruction ';'} 'End' ';' .

ConstExpr = ConstSimpleExpr ConstRelOp ConstSimpleExpr .

ConstRelOp =    '=' | '<>' | '<' | '>'
            |'<=' | '>=' | 'Xor' | 'Xnor'.

ConstSimpleExpr = ['+' | '-'] ConstTerm
                      {ConstAddOp ConstTerm}.

ConstAddOp = '+' | '-' | 'Or' | 'Nor' .

ConstTerm = ConstFactor {ConstMulOp ConstFactor} .

ConstMulOp =     '*'  | 'Div' | 'Mod' | 'And'
           | 'Nand' | 'Lsh' | 'Rsh' | 'Rot' .

ConstFactor = {'Not'} (constant | '(' ConstExpr ')') .

Instruction = {label ':'} Phrase {',' Phrase} .

  Phrase = empty
         | Pseudo
         | PascalStyleConstant
         | {Result ':='} ALU
         | Jump
         | Special .

    Pseudo = 'Define' '(' ['%'] name ',' ConstExpr ')'
           | 'Constant' '(' name ',' ConstExpr ')'
           | 'Opcode' '(' [ConstExpr ','] ConstExpr ')'
           | 'Loc' '(' ConstExpr ')'
           | 'Binary'
           | 'Octal'
```

```
        | 'Decimal'
        | 'Nop'
        | 'Case' '(' ConstExpr ',' ConstExpr ')'
        | 'Place' '(' ConstExpr ',' ConstExpr ')' .

PascalStyleConstant = name '=' ConstExpr .

Result = register
        | 'TOS' | 'MA' | 'MDO' | 'BPC'
        | 'SrcRasterOp' | 'DstRasterOp'
        | 'WidRasterOp' | 'MQ' | 'RBase" .

ALU = ['not'] (AMux | BMux)
        | AMux Op ['not'] BMux
        | AMux '+' BMux ['+' 'OldCarry']
        | AMux '-' BMux ['-' 'OldCarry']
        | AMux 'AMUX' BMux
        | AMux 'BMUX' BMux
        | 'MQ'
        | 'Victim' .

AMux = register | 'Shift'' | 'NextOp' | 'IOD'
        | 'MDI' | 'MDX'  | 'TOS'
        | 'UState' ['(' register ')']
        | 'Upper' ['(' register ')'] .

BMux = register | constant | '(' ConstExpr ')' .

Op = 'And' | 'Or' | 'Xor' | 'Nand' | 'Nor' | 'Xnor' .

Jump = ['If' Condition] Directive ['(' Target ')'] .

Condition = 'True' |  'False'   | 'BPC[3]'  | 'C19'
        | 'C23'    | 'IntrPend' | 'Odd'     | 'ByteSign'
        | 'Eql'    | 'Neq'      | 'Gtr'     | 'Geq'
        | 'Lss'    | 'Leq'      | 'Carry'   | 'OverFlow'.

Directive = 'Goto' | 'Call' | 'Return' | 'Next' | 'JumpZero'
        | 'LoadS' | 'GotoS' | 'CallS' | 'NextInst'
        | 'ReviveVictim' | 'PushLoad' | 'Vector' | 'Dispatch'
        | 'RepeatLoop' | 'Repeat' | 'JumpPop' | 'LeapPop'
        | 'Loop' | 'ThreeWayBranch' .

Target = label | constant | 'Shift' .

Special = Nonary | Unary | Binary .

Nonary=  'WCSlow' | 'WCSmid' | 'WCShi' | 'LoadOp' | 'Hold'
        | 'StackReset' | 'Push' | 'Pop' | 'Fetch' | 'Fetch2'
        | 'Fetch4' | 'Fetch4R' | 'Store' | 'Store2'
        | 'Store4' | 'Store4R' | 'ShiftOnR'
        | 'MultiplyStep' | 'DivideStep' .

Unary = UnaryName '(' ConstExpr ')' .
```

```
UnaryName = 'LeftShift' | 'RightShift' | 'Rotate'
          | 'IOB'  | 'CntlRasterOp' .
Binary = BinaryName '(' ConstExpr ',' ConstExpr ')' .
BinaryName  = 'Field' .
```

1.6. Notes on the Syntax

Numeric constants preceded by a '#' are octal constants.

Constants can be defined Pascal style:

name = value;

This allows including a file that contains constant definitions into both a Pascal program and a microprogram.

The syntax allows constructions that are semantically incorrect. In other words, there are many combinations of actions that cannot be represented in a single instruction. For example,

TOS := MA := 10; is valid, but

TOS := BPC := 10; is invalid.

Section 2.3 shows which fields of a microinstruction are used by a particular action. The rule is that a certain field may be used only once. Thus since 'TOS :=' and 'BPC :=' both use the SF (special function) field, they both cannot be used in a single micro-instruction.

Some features of the hardware are specific to the PERQ1, PERQ1a, or PERQ1b CPU. The assembler reflects this by restricting usage of these features:

- The percent sign that signals the use of the base register may only be used on the PERQ1a or PERQ1b.

- MQ, RBase, Victim, LeapPop, Goto(Shift), MultiplyStep, and DivideStep may only be used on the PERQ1a or PERQ1b.
- 'Upper' may only be used on the PERQ1b.

Some goto types do not allow tests (are unconditional), and for some the test is optional. Similarly, some do not allow addresses, some require them, and for some the address field is optional. For PERQ1a and PERQ1b, some goto types do not allow Shift to be used as the address. Table 1 specifies the rules.

1.7. PERQ Workstation Hardware Architecture

The PERQ workstation is implemented with a high-speed microprogrammed processor with a 170 nanoseconds microcycle time. A microinstruction is 48 bits wide. Most of the data paths in the micro engine are 20 bits wide (24 bits wide for PERQ1b CPUs). The data coming in and out of the processor (for example, IO and Memory data) are 16 bits wide (20 bits for PERQ1b). The extra 4 bits allow the microprogrammed processor to calculate real addresses in a 1 megaword addressing space. The assumption is that virtual addresses are kept in a doubleword in memory but calculations on addresses can be single precision within the processor. The programmer of the virtual machine never sees the 20 bit paths.

The major data paths are shown in Figure 1.

The XY registers (256 registers x 20 bits--24 bits for PERQ1b) form a dual ported file of general-purpose registers. The X port outputs are multiplexed with several other sources (the AMux) to form the A input to the ALU. The Y port outputs, multiplexed with an 8- or 16-bit constant via the BMux, form the B input to the ALU. The ALU outputs (R) are fed back to the XY registers as well as the memory data output and memory address registers.

Table 1: Goto Specifications

req - required, opt - optional, <blank> - not allowed

Goto type	test	address	Shift
Goto	opt	req	opt
Call	opt	req	opt
Return	opt		
Next			
JumpZero			
LoadS		req	opt
GotoS	opt	opt	opt
CallS	opt	opt	opt
NextInst	req		
ReviveVictim			
PushLoad	opt	req	opt
Vector	opt	req	
Dispatch	opt	req	
RepeatLoop			
Repeat		req	opt
JumpPop	opt	req	opt
LeapPop*	opt	req	opt
Loop	opt		
ThreeWayBranch	opt	req	

(*PERQ1a and PERQ1b only)

Memory data coming from the memory is sent to the ALU via the AMux. The IO bus (IOB) connects the CPU to IO devices. It consists of an 8-bit address (IOA) which is driven from a microword field and a 16-bit birectional data bus (IOD) which is read via AMux and written from R.

Opcodes and operands that are part of the instruction byte stream are buffered in a special 8 x 8 RAM (the Op file). The Op file is loaded 16 bits at a time from the memory data inputs. The output

Figure 1: PERQ System Block Diagram

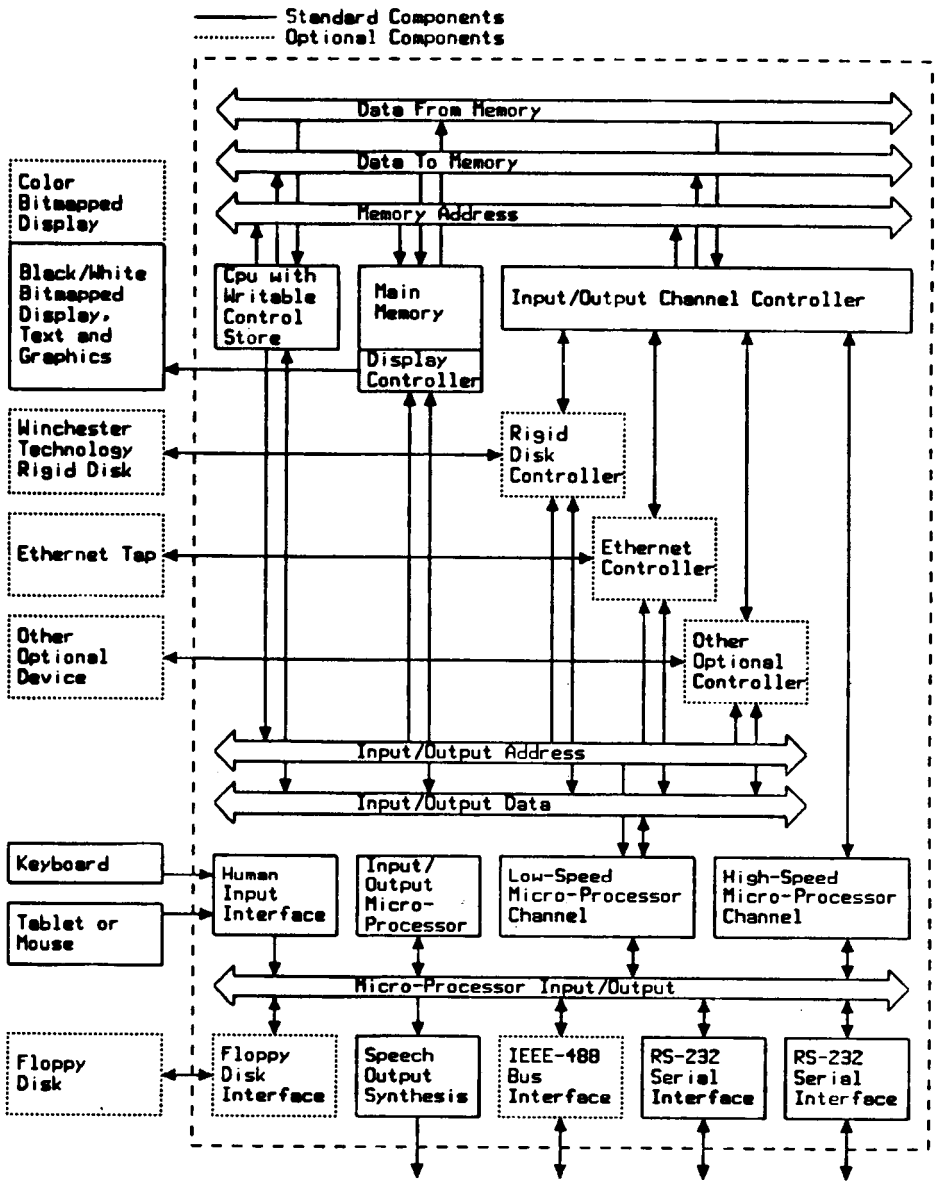
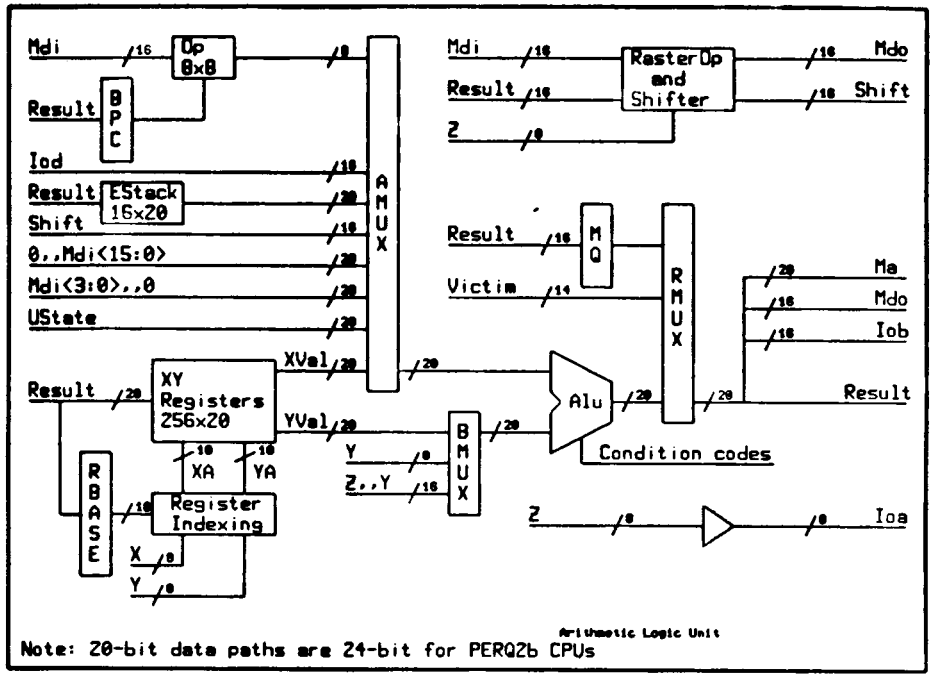
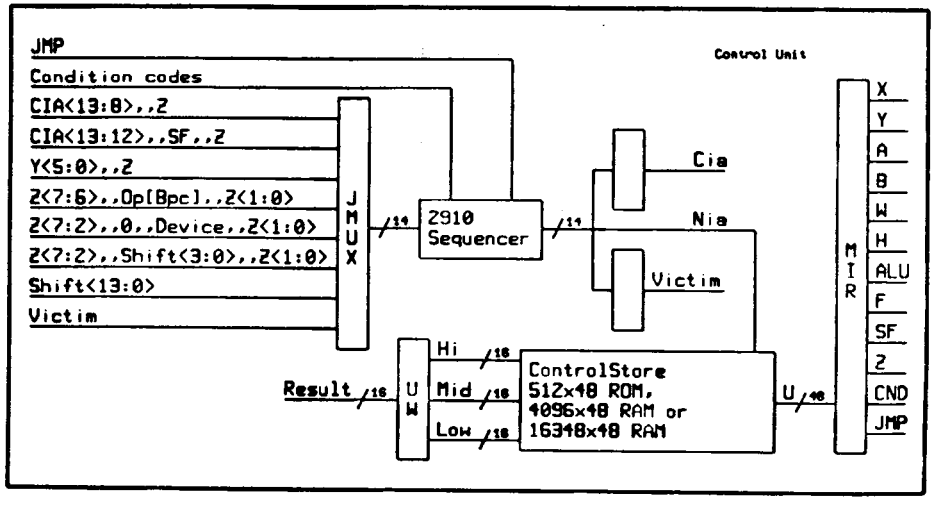


Figure 2: Arithmetic Logic Unit and Control Unit





of the Op file is 8 bits wide and can be read via AMux or can be sent to the micro-addressing section for opcode dispatch. The read port of the Op file is addressed by the 3-bit BPC (Byte Program Counter).

A shift matrix (Shift), which is part of the special hardware provided for the RasterOp operator, can be accessed by loading an item to be shifted via the R bus, and reading the shifted result on AMux.

A 16-level push down stack (EStk) is written from R and read on AMux. The stack is used by the Q-code interpreter to evaluate expressions. BPC and the microstate condition codes can be read as the Micro State Register (UState) via AMux.

2. Microinstructions

Each microinstruction is a collection of 12 fields with a total of 48 bits, executed by the machine in 170 ns. Each of these 170 nanosecond intervals is referred to as a microcycle.

An "instruction" is a sequence of "Phrases." The microassembler produces microinstructions from one or more of these instructions. Each instruction can be associated with zero or more labels. Instructions are separated by semicolons (";").

NOTE: Some microinstructions, notably those produced by fetch and store type instructions, must be executed at specific microcycles, and they influence several following microcycles.

The general syntax of a microinstruction is:

```
Instruction = {label ':'} Phrase {',' Phrase} .

Phrase = empty
       | Pseudo
       | PascalStyleConstant
       | {Result ':='} ALU
       | Jump
       | Special .
```

2.1. Labels

A label is a name that is prefixed to an instruction for use as a branching target. The syntax of a label is:

```
label = name .
```

Instructions can have simple or compound labels (see Sections 2.1.1 and 2.1.2).

2.1.1. Simple labels

A simple label is a name separated from its associated instruction with a colon. This name can then be used as a target in a branch or jump instruction. A name is formed from a letter followed by any number of letters or digits. The microassembler recognizes only the first ten letters or digits of a name; any more are ignored.

2.1.2. Compound labels

Compound labels are formed by concatenating label-colon pairs to the front of an instruction. Thus, each instruction can have several names. For example, the construction

```
foobar:zeetix:looptarget:  A + B + OldCarry;
```

allows the microinstruction "A + B + OldCarry" to be named "foobar", "zeetix", or "looptarget".

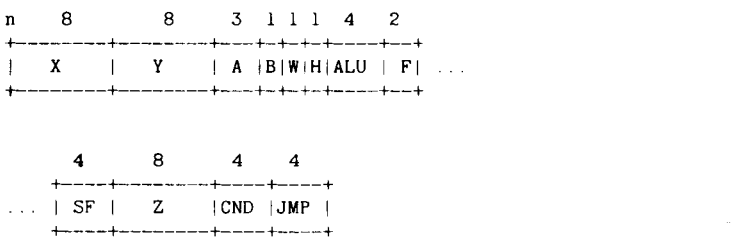
2.2. Phrases

A phrase is the syntactic building block used to build instructions. An instruction is formed from a concatenation of phrases, with each phrase separated by a comma (','). The assembler recognizes six types of phrases. Chapter 2.1 describes the six types in detail. The syntax is as follows:

```
Phrase = empty
       | Pseudo
       | PascalStyleConstant
       | {Result ':='} ALU
       | Jump
       | Special .
```

2.3. Microinstruction Format

Each 48-bit microinstruction is composed of 12 fields, as shown below.



The following is the way that microinstructions are written into microstore using the WCSHi, WCSMid, and WCSLow directives (see Section 3.5.1.1). Note the order that the microinstruction fields are contained in the three words:

WCS High X, Y

WCS Mid CND, JMP, B, H, F, SF

WCS Low Z, A, ALU.1, W, ALU.0, ALU.3, ALU.2

Note that when writing words into the microstore, all bits are complemented, except for:

Z.7..0 for Jumps, Dispatch, Vector, NextInst, LeftShift, RightShift, Rotate, and Field.

SF.3..0 for Long Jumps.

Z.6..0 for IOB addresses--note that Z.7 *is* inverted in this case.

A brief description of each field is presented in Sections 2.3.1 through 2.3.12. The microassembler produces microinstructions based upon the instructions described in detail in Chapter 3.

2.3.1. X (bits 47..40)

The X-field contains the address for the X port of the XY register file. This same field is used to reference a given register in the XY file for a register write operation.

2.3.2. Y (bits 39..32)

The Y-field contains the address for the Y port of the XY register file. It can also be used as the low order byte of a constant.

2.3.3. A (bits 31..29)

The A-field contains the select lines used to drive the AMux. The A-field is encoded as in the table below:

A Field	Selects
0	Shifter output
1	NextOp (Opfile[BPC])
2	IOD (IO Data bus)
3	MDI (Memory Data inputs). See Section 3.3.2.1.
4	MDX (Memory Data input, extended). See Section 3.3.2.1.
5	Microstate register (UState) or Upper.
6	XY register at location specified by the X field.
7	Top of the Expression Stack.

2.3.4. B (bit 28)

The B-field contains the BMux select line. If B is set, the BMux selects a constant. Otherwise, BMux selects the register specified by the Y-field contents.

2.3.5. W (bit 27)

The Write bit, when set, causes the contents of R to be written into the register specified in the X field. When reset, no XY registers are modified.

2.3.6. H (bit 26)

The Hold bit, when set, prevents IO devices from accessing memory. It is also used with the JMP field (see Section 2.3.12) to modify address inputs.

2.3.7. ALU (bits 25..22)

The ALU field encodes the function used by the ALU to combine the A and B inputs to the ALU. It is encoded as in the table below.

ALU Field (Octal)	ALU Function
0	A
1	B
2	Not A
3	Not B
4	A And B
5	A And Not B
6	A Nand B
7	A Or B

10	A Or Not B
11	A Nor B
12	A Xor B
13	A Xnor B
14	A + B
15	A + B + OldCarry
16	A - B
17	A - B - OldCarry

OldCarry is the carry from the immediately preceeding microinstruction and is used for multiple precision arithmetic.

2.3.8. F (bits 21..20)

The Function field controls the interpretation of the SF and Z field contents. The F field is encoded as in the table below:

Function	SF Use	Z use
For the PERQ1 CPU:		
0	Special Function	Constant/Short Jump
1	Memory Control	Short Jump
2	Special Function	Shift Control
3	Long Jump	Long Jump

For the PERQ1a and PERQ1b CPUs:

0	Special Function	Constant/Short Jump
1	Memory Control	Short Jump and extended Special Func.
2	Special Func.	Shift Control
3	Long Jump	Long Jump

2.3.9. SF (bits 19..16)

While the function field (see Section 2.3.8) selects Memory Control, the Special Function bits contain the memory control bits. When the function field selects Long Jump, the Special Function field contains the four high-order memory address bits. Otherwise, the Special Function field selects following functions as in the table below:

SF (Octal)	FUNCTION
0	LongConstant
1	ShiftOnR
2	StackReset (clear the EStk)
3	TOS := (R) (Top of EStk)
4	Push (the EStk)
5	Pop (the EStk)
6	CntlRasterOp := (Z)
7	SrcRasterOp := (R)

```
10      DstRasterOp := (R)

11      WidthRasterOp := (R)

12      LoadOp (OP := MDI)

13      BPC := (R)

14      WCS[15..0] := (R)

15      WCS[31..16] := (R)

16      WCS[47..32] := (R)

17      IOB Function
```

When used as Memory Control:

```
10      Fetch4R  Fetch 4 words in reverse order

11      Store4R  Store 4 words in reverse order

12      Fetch4   Fetch 4 words

13      Store4   Store 4 words

14      Fetch2   Fetch 2 words

15      Store2   Store 2 words

16      Fetch    Fetch 1 word from memory

17      Store    Store 1 word into memory
```

For PERQ1a and PERQ1b CPUs when used as Extended Special Function:

```
0       (R) := Victim Latch
```

```
1       Multiply step or divide step

2       Load multiplier or dividend

3       Load base register

4       (R) := product or quotient

5       Push long constant (EStk)

6       2910 address inputs := Shift

7       Leap address generation
```

2.3.10. Z (bits 15..8)

The Z-field contains the low eight bits of a jump address, the high eight bits of a Constant, shift control, or an IOB address, depending on the state of the Function (F) and Special Function (SF) fields.

The encodings of the F field do not necessarily enforce restrictions on the use of the Z field, they merely enable some of them. In particular, B = 1, SF = 0, and F = 0 or 2 selects a long constant using the Z field. For the PERQ1a and PERQ1b, F = 1 and SF = 5 also selects a long constant using the Z field. In the PERQ1, long constants and special functions may not be used in the same microinstruction. The PERQ1a and PERQ1b have a special function which pushes the EStk and selects a long constant in the same instruction. The programmer need not select this special function explicitly--the assembler takes care of it. When F <> 2, the Z field is used for a jump address. When SF = 17 and F = 0 or 2, the Z field is used for an IOB address. When F = 2, the Z field is loaded into the Shift Control register. These are the only specific actions taken by the hardware that affect the usage of the Z field. The hardware does nothing to prevent the Z field from being used for several things at once. For example, it

could be used for a long constant and a jump address at the same time, or it could be used as an IO address and a jump address at the same time. The assembler, however, flags an error if the programmer tries to load two different values into the same microinstruction field.

2.3.11. Cnd (bits 7..4)

The Condition (Cnd) field determines what to test during a conditional jump. They are encoded as in the table below:

Cnd (Octal)	Test
0	True - always jump
1	False - never jump
2	IntrPend - interrupts pending
3	Spare (unused)
4	BPC[3] - Op File is empty
5	C19 - no carry out of bit 19 of the ALU (R[19]) for PERQ1a C23 - no carry out of bit 23 of the ALU (R[23]) for PERQ1b
6	Odd - ALU bit 0 (R[0])
7	ByteSign - ALU bit 7 (R[7])
10	Neq - Not equal to
11	Leq - Less than or equal to
12	Lss - Less than

```
13      Overflow - 16 Bit overflow in the ALU

14      Carry - carry out of bit 15 of the
        ALU (R[15])

15      Eql - Equal to

16      Gtr - Greater than

17      Geq - Greater than or equal to
```

2.3.12. JMP (bits 3..0)

The Jmp field is described in detail in Chapter 6 of the Advanced Micro Devices document *Bipolar Microprocessor Logic and Interfaces Data Book*; phrases which use it are described in detail in Section 3.4.

For the PERQ1 CPU, the Jmp field is encoded as in the table below, using the following symbols:

```
CIA         =     Current Instruction Address.

NIA         =     Next Instruction Address.

Addr        =     CIA[11:8],,Z[7:0]  (Short)

                  or SF[3:0],,Z[7:0]  (Long).

S           =     Internal Address Register.

CStk        =     Top of 5-Level Call Stack.

OpCode      =     Z[7:6],,(not OpFile[BPC])[7:0],,

                  Z[1:0].
```

```
Vector      =     Z[7:2],,0,,(not
                  Device)[2:0],,Z[1:0].

Dispatch    =     Z[7:2],,(not Shift)[3:0],,Z[1:0].

Push        =     Push CIA+1 onto call stack

Pop         =     Pop call stack
```

CODE(Octal)/

	NAME	PASS	FAIL
0		NIA := 0	NIA := 0
	JumpZero	Clear CStk	Clear CStk
1		NIA := Addr	NIA := CIA + 1
	Call	Push CStk	
2			
	NextInst	NIA := OpCode	NIA := OpCode
	(H=0)		
	ReviveVictim	NIA := Victim	NIA := Victim
	(H=1)		
3		NIA := Addr	NIA := CIA + 1
	Goto		
4		NIA := CIA + 1	NIA := CIA + 1
	PushLoad	Push CStk	Push Cstk
		S := Addr	
5		NIA := Addr	NIA := S
	CallS	Push CStk	Push CStk
6			
	Vector	NIA := Vector	NIA := CIA + 1
	(H=0)		

```
  Dispatch          NIA := Dispatch        NIA := CIA + 1
  (H=1)
7                   NIA := Addr            NIA := S
  GotoS
10
  RepeatLoop        NIA := CStk            NIA := CStk
  S <> 0            S   := S - 1           S   := S - 1
  S = 0             NIA := CIA + 1         NIA := CIA + 1
                    Pop CStk               Pop CStk
11
  Repeat            NIA := Addr            NIA := Addr
  S <> 0            S   := S - 1           S   := S - 1
  S = 0             NIA := CIA + 1         NIA := CIA + 1
12                  NIA := CStk            NIA := CIA + 1
  Return            Pop CStk
13                  NIA := Addr            NIA := CIA + 1
  JumpPop           Pop CStk
14                  NIA := CIA + 1         NIA := CIA + 1
  LoadS             S   := Addr            S   := Addr
15                  NIA := CIA + 1         NIA := CStk
  Loop              Pop CStk
16
  Next              NIA := CIA + 1         NIA := CIA + 1
17
  ThreeWayBranch
  S <> 0            NIA := CIA + 1         NIA := CStk
                    Pop CStk
                    I - S =:   S           I - S =:   S
```

```
S = 0          NIA := CIA + 1          NIA := Addr
               Pop CStk                Pop CStk
```

For the PERQ1a and PERQ1b CPUs, the JMP field is encoded as in the table below, using the following symbols:

```
CIA        = Current Instruction Address.

NIA        = Next Instruction Address.

CBank      = Current 4K microstore Bank.

Addr       = CBank[1:0],,CIA[11:8],,Z[7:0](Short) or
             CBank[1:0],,SF[3:0],,Z[7:0] (Long) or
             Y[5:0],,Z[7:0]                (Leap).

S          = Internal Register/Counter.

CStk       = Top of 5-level call stack.

OpCode     = Z[7:6],,(not OpFile[Bpc])[7:0],,Z[1:0].

Vector     = Z[7:2],,0,,(not device)[2:0],,Z[1:0].

Dispatch   = Z[7:2],,(not Shift)[3:0],,Z[1:0].

Push       = Push CIA + 1 onto call stack.

Pop        = Pop call stack.

(lo)       = Bits [11:0].

(hi)       = Bits [13:12].
```

CODE(Octal)/

	NAME	PASS	FAIL
0		NIA := 0	NIA := 0
	JumpZero	Clear CStk	Clear CStk
1		NIA:=Addr	NIA:=CIA+1
	Call	Push CStk	
2			
	NextInst	NIA(lo):=OpCode	NIA(lo):=OpCode
	(H=0)	NIA(hi):=CBank	NIA(hi):=CBank
	ReviveVictim	NIA(lo):=Victim	NIA(hi):=Victim
	(H=1)	NIA(hi):=CBank	NIA(lo):=CBank
3		NIA:=Addr	NIA(lo):=CIA+1
	Goto		NIA(hi):=CBank
4		NIA(lo):=CIA+1	NIA(lo):=CIA+1
	PushLoad	NIA(hi):=CBank	NIA(hi):=CBank
		Push CStk(lo)	Push CStk(lo)
		S:=Addr	
5		NIA:=Addr	NIA:=S
	CallS	Push CStk	Push CStk
6			
	Vector	NIA(lo):=Vector	NIA:=CIA+1
	(H=0)	NIA(hi):=CBank	NIA(hi):=CBank
	Dispatch	NIA(lo):=Dispatch	NIA:=CIA+1
	(H=1)	NIA(hi):=CBank	NIA(hi):=CBank
7			
	GotoS	NIA:=Addr	NIA:=S

```
10
  RepeatLoop
   if S(lo) <> 0   NIA(lo):=CStk          NIA(lo):=CStk
                   NIA(hi):=CBank         NIA(hi):=CBank
                   S(lo):=S(lo)-1         S(lo):=S(lo)-1
   if S(lo) = 0    NIA(lo):=CIA+1         NIA(lo):=CIA+1
                   NIA(hi):=CBank         NIA(hi):=CBank
                   Pop CStk(lo)           Pop CStk(lo)
11
  Repeat
   if S(lo) <> 0   NIA(lo):=Addr          NIA(lo):=Addr
                   NIA(hi):=CBank         NIA(hi):=CBank
                   S(lo):=S(lo)-1         S(lo):=S(lo)-1
   if S(lo) = 0    NIA(lo):=CIA+1         NIA(lo):=CIA+1
                   NIA(hi):=CBank         NIA(hi):=CBank
12                 NIA:=CStk              NIA(lo):=CIA+1
  Return                                  NIA(hi):=CBank
                   Pop CStk
13
  JumpPop          NIA:=Addr              NIA(lo):=CIA+1
  (H=0)                                   NIA(hi):=CBank
                   Pop CStk(lo)
  LeapPop          NIA:=Addr              NIA(lo):=CIA+1
  (H=1)                                   NIA(hi):=CBank
                   Pop CStk
14                 NIA(lo):=CIA+1         NIA(lo):=CIA+1
  LoadS            NIA(hi):=CBank         NIA(hi):=CBank
                   S:=Addr                S:=Addr
15                 NIA(lo):=CIA+1         NIA(lo):=CStk
  Loop             NIA(hi):=CBank         NIA(hi):=CBank
                   Pop CStk(lo)
```

```
16                  NIA(lo):=CIA+1          NIA(lo):=CIA+1
  Next              NIA(hi):=CBank          NIA(hi):=CBank
17
  ThreeWayBranch
   if S(lo) <> 0    NIA(lo):=CIA+1          NIA(lo):=CStk
                    NIA(hi):=CBank          NIA(hi):=CBank
                    Pop CStk(lo)
                    S(lo):=S(lo)-1          S(lo):=S(lo)-1
   if S(lo) = 0     NIA(lo):=CIA+1          NIA(lo):=Addr
                    NIA(hi):=CBank          NIA(hi):=CBank
                    Pop CStk(lo)            Pop CStk(lo)
```

3. Phrases

This chapter describes each of the phrases recognized by the microassembler.

The general syntax of a phrase is as follows:

```
Phrase = empty
       | Pseudo
       | PascalStyleConstant
       | {Result ':='} ALU
       | Jump
       | Special .
```

3.1. Pseudo Phrases

A "Pseudo" is a phrase which is used by the microassembler to alter its own state. Its syntax is:

```
Pseudo = 'Define' '(' ['%'] name ',' ConstExpr ')'
       | 'Constant' '(' name ',' ConstExpr ')'
       | 'Opcode' '(' [ConstExpr ','] ConstExpr ')'
       | 'Loc' '(' ConstExpr ')'
       | 'Binary'
       | 'Octal'
       | 'Decimal'
       | 'Nop'
       | 'Case' '(' ConstExpr ',' ConstExpr ')'
       | 'Place' '(' ConstExpr ',' ConstExpr ')' .
```

Sections 3.1.1 through 3.1.10 describe the defined Pseudos.

3.1.1. Define

Associates a name with a certain register number. The ConstExpr must be in the range 0..255.

For example, the instruction:

```
Define(aRegister, #10);
```

informs the assembler that the name aRegister refers to register number #10. When the $PERQ1a and the $Base assembler options are used, the percent sign (%) is required for registers with numbers less than #100.

3.1.2. Constant

Associates a name with a numeric constant. For example:

```
Constant(Fourteen, 2*7);
```

informs the assembler that the name Fourteen should be synonymous with the constant 14 (decimal).

3.1.3. Opcode

Assembles the current instruction into the location specified by the following formula:

```
Opcode(Op)       ==>(Op Xor #377) lsh 2
Opcode(Base, Op) ==>(Op Xor #377) lsh 2 + Base
```

For example, the instruction:

```
Opcode(1), Tos := Tos + 1;
```

assembles into the location:

```
(1 Xor #377) lsh 2
```

which evaluates to:

```
#376 lsh 2
```

which evaluates to:

```
1770
```

This computation matches the hardware for the NextInst jump so that statements containing

```
Opcode(JumpTableBase, OpcodeNumber), ...
```

can be used in conjunction with a jump of the form

```
NextInst(JumpTableBase)
```

3.1.4. Loc

Assembles the current instruction into the location specified by ConstExpr.

3.1.5. Binary

Makes 2 the default base for numeric constants.

3.1.6. Octal

Makes 8 the default base for numeric constants.

3.1.7. Decimal

Makes 10 the default base for numeric constants.

3.1.8. Nop

A placeholder. Assembles a No-Op instruction, used for synchronization between the processor and the memory / IO systems.

In response to a NOP, the microassembler builds a MicroInstruction with the X, Y, B, W, H, ALU, and CN fields equal to zero, and the A field equal to 6 (select XY[X]). The

JMP, SF, and Z fields are used to encode a jump to the following instruction (in the source code).

NOTE: The condition codes are not preserved during a Nop instruction because the hardware executes an ALU operation during every instruction, including Nop.

3.1.9. Case

Assembles the next instruction into the location specified by the following formula:

```
Case(Val)       ==> (Val Xor #17) lsh 2
Case(Base, Val) ==> (Val Xor #17) lsh 2 + Base
```

For example, the instruction

```
Case(#100, 1), Tos := Tos + 1;
```

assembles into the location

```
(1 Xor #17) lsh 2 + #100
```

which evaluates to:

```
#16 lsh 2 + #100
```

which evaluates to:

```
#170
```

This computation matches the hardware for the Dispatch jump so that statements containing

```
Case(JumpTableBase, CaseNumber), ...
```

can be used in conjunction with a jump of the form

```
Dispatch(JumpTableBase)
```

3.1.10. Place

Makes a range of locations available for assembly. The statement

Place(A, B)

specifies the range A to B (inclusive). Several Place statements can be used, in which case the union of all ranges can be used. This function is used by the PRQPLACE.

"Place" allows the microstore to be partitioned, so that explicit subranges of it can be loaded without risk of inadvertant damage to the remainder. For example, a range of the microstore could be defined, using Place, as an overlay area. Attempts to load microinstructions outside the range specified in the Place (through the "LOC" directive, for example) cause the microassembler to flag an error.

3.2. Pascal Style Constants

The Pascal style constant definition:

name = ConstExpr

is functionally identical to:

Constant(name, ConstExpr).

This construction allows a constant definition file to be included by both a Pascal program and a microprogram. The instruction causes all references to the name to be interpreted as constants.

This is useful in defining entities such as register assignments or parameters which must be used by both Pascal programs and PERQ workstation microprograms.

For example, suppose a file labeled "AFile" contained the following statements:

```
RegLoad     = 1002;
StartVal    = 1023;
StopVal     = 3049;
```

and a Pascal program and a PERQ workstation microprogram looked like:

```
PASCAL PROGRAM          PERQ MICROPROGRAM
...                     ...
Const                   $Include Afile
{$Include AFile}          ...
...
```

The constants "RegLoad", "StartVal", and "StopVal" would be available to both the Pascal program and the microprogram, and a change to their value (in AFile) is passed into both.

The syntax for a Pascal Style Constant is:

```
PascalStyleConstant = name '=' ConstExpr .
```

WARNING: The default base in Pascal is always Decimal, but the default base in microcode is usually Octal (but may be changed). Thus it is safest to use declarations of the form:

```
name = #number
```

which is interpreted as Octal in both Pascal and microcode. Alternatively, the microprogram could contain:

```
Decimal;            ! change default base to 10
$Include AFile
Octal;              ! change default base back to 8
```

3.3. {Result ':='} ALU

Sections 3.3.1 and 3.3.2 describe the primitives that form the constructions of Section 3.3.3. Section 3.3.1 describes the optional result assignment locations and Section 3.3.2 describes Amux, Bmux, the operators that drive the ALU, and the OldCarry bit. Section 3.3.3 describes the permissible constructions that can be formed from Amux, Bmux, the operators, and the OldCarry bit.

The 20-Bit AMux output (AMux[19:0]--AMux[23..0] for PERQ1b) and the 20-Bit BMux output (BMux[19:0]--BMux[23..0] for PERQ1b) form the A and B inputs to the ALU. The ALU output is latched in the R register (R[19:0]--R[23..0] for PERQ1b).

The syntax is:

```
Result = register
    | 'TOS' | 'MA' | 'MDO' | 'BPC'
        | 'SrcRasterOp' | 'DstRasterOp'
        | 'WidRasterOp' | 'MQ' | 'RBase" .
```

3.3.1. {Result ':='}

The ALU output can be written into zero or more Result locations, described in Sections 3.3.1.1 through 3.3.1.8 below.

The assignment construction stores the full or partial result of the ALU (R[19:0] or R[23:0] for PERQ1b) in the designated target location. In addition to simple assignments (which store R in one location), compound assignments (which store R in several locations) can be constructed, subject to the constraint that each microinstruction field can only be assigned *once*.

NOTE: The syntax allows multiple assignment of the same ALU output in the same microinstruction. Since each field of the

microinstruction can only be assigned once per microinstruction (see Section 2.3), care should be exercised in multiple assignments to prevent conflicts. The microassembler dissallows such conflicts.

Simple Assignments (result := ALU) copy the bits in R to the location specified by result, bit for bit. If the result location is less than 20 (24 for PERQ1b) bits wide (BPC, for example), the assignment maps only those bits of R which also appear in result. For example:

```
BPC := IOD;
```

copies the four low-order bits of the IO data bus into the four-bit BPC register.

Compound Assignments (a := b := ... := ALU) copy the bits in R to each location specified so long as each result location can be specified without a conflicting use of any microinstruction field. For example:

```
Define(Foo, #15), Foo := TOS := MA := 10, Fetch;
```

is valid (TOS uses the SF and F fields, MA uses no fields), while the instruction:

```
TOS := BPC := 10;
```

is invalid (TOS uses the SF field in a way which conflicts with the BPC use of the same field). The microassembler disallows these invalid combinations.

3.3.1.1. Register

Register directs the contents of R to Register. Register must be a previously defined register name (see Section 3.1.1).

A base register facility is available on the PERQ1a and PERQ1b CPUs. Some suggested uses of this facility are:

1. Saving and restoring register values in a loop.
2. Establishing context dependent registers in a way that they may co-exist with a set of globally accessable registers.
3. Using the registers as a deep stack.

The use of the base register is controlled by bits 6 and 7 of the X and Y fields of the microinstruction. For register numbers 0 through 77, the base register value is ORed with the X or Y field to form the register address. Register numbers 100 through 377 are not modified. Thus if the base register is loaded with 0, the PERQ1a and PERQ1b register addressing mechanism is compatible with that of the PERQ1.

The base register is loaded from R with the "RBase := " special function. The value loaded into the base register is inverted when it is loaded. Thus to load a value, V, into the base register, use:

"RBase := not V".

The base register is cleared when the boot button is pressed, permitting normal register access during boot sequences.

The assembler requires the programmer to explicitly indicate whether a register is affected by the base register. This is required both at the point of definition and at the point of use. Registers whose numbers are less than 100 must be prefixed by a

percent sign (%).

If the base register is set to zero, or is not loaded after booting, register addressing on the PERQ1a and PERQ1b is compatible with the PERQ1. In this case, the $NOBASE assembler option may be used so that the assembler will not require the percent sign prefix.

This instruction uses the X and W fields (X := address, W := 1).

3.3.1.2. TOS

TOS is a reference to the top of the expression stack. Each assignment to TOS replaces the previous contents of the expression stack with the current ALU output.

A reference to TOS uses the F and SF fields.

3.3.1.3. MA

MA is the memory address register. MA is latched from R during fetch or store type microinstructions. An assignment to the MA register does not occupy any microinstruction fields, but is helpful as a mnemonic aid.

NOTE: The MA register is latched from R during *each* fetch or Store type microinstruction. It is therefore a good programming practice to reflect this in an assignment to it, making its contents (for the fetch or store) explicit.

3.3.1.4. MDO

MDO is the memory data output register. Each assignment to MDO directs the low-order 16 bits of the ALU output (R[15:0]) to MDO[15:0]. Like the MA register, an assignment to MDO does not occupy any fields, but is helpful as a mnemonic aid.

NOTE: The MDO register is latched from R during the data transfer portion of each store type instruction. It is therefore a good programming practice to reflect this in an assignment to it, making its contents explicit.

3.3.1.5. BPC

The four-bit Byte Program Counter (BPC). An assignment to BPC loads R[3:0] (the four low-order bits of R) into the BPC.

This instruction uses the F and SF fields (F:=0, SF:=8#13).

3.3.1.6. SrcRasterOp

SrcRasterOp is a control register in the raster-op / shifter hardware. An assignment to SrcRasterOp loads R into the SrcRasterOp register.

This instruction uses the F and SF fields (F:=0, SF:=8#7).

3.3.1.7. DstRasterOp

DstRasterOp is a control register in the raster-op / shifter hardware. An assignment to DstRasterOp loads R into the DstRasterOp register.

This instruction uses the F and SF fields (F:=0, SF:=8#10).

3.3.1.8. WidRasterOp

WidRasterOp is a control register in the raster-op/shifter hardware. An assignment to WidRasterOp loads R into the WidRasterOp register.

This instruction uses the F and SF fields (F:=0, SF:=8#11).

3.3.1.9. RBase

The PERQ1a and PERQ1b provide a base register facility. See Section 3.3.1.1 for a description of how the base register affects register addressing. An assignment to RBase loads the complement of R[7:0] (the low order 8 bits) into the base register.

This instruction uses the F and SF fields (F:=1, SF:=3).

3.3.1.10. MQ

The PERQ1a and PERQ1b provide hardware to assist integer multiply and divide. See Sections 3.5.1.16 and 3.5.1.17 for a description of the multiply/divide hardware. As assignment to MQ loads R into the multiplier/quotient register.

This instruction uses the F and SF fields (F:=1, SF:=2).

3.3.2. ALU

The syntax is

```
Amux ::= register
       | 'Shift'
       | 'NextOp'
       | 'IOD'
       | 'MDI'
       | 'MDX'
       | 'TOS'
       | 'UState' [ '(' register ')' ]
       | 'Upper' [ '(' register ')' ]
```

```
Bmux ::= register | constant | '(' ConstExpr ')'

Op   ::= 'and'
       | 'or'
       | 'Xor'
       | 'nand'
       | 'nor'
       | 'Xnor'
```

3.3.2.1. AMux

The microassembler uses the following primitives to control the AMux. The 20 AMux output bits (AMux[19..0]) form the A input to the ALU.

```
AMux = register | 'Shift' | 'NextOp' | 'IOD'
     | 'MDI' | 'MDX' | 'TOS'
     | 'UState' ['(' register ')']
     | 'Upper' ['(' register ')'] .
```

Register directs the contents of the register addressed by the X field to the AMux output. Register must be a previously defined register name (see Section 3.1.1). If part of an assignment construction, the register specified in AMux must be the same as the register being written. For example, the statement

```
Foo := Foo + Bar;
```

works, while the statement

```
Foo := Bar + Zee;
```

does not.

Shift directs the output of the shifter onto the AMux output lines. This output is the shifted value of whatever was on R in the last executed microinstruction.

NOTE: For meaningful results, the shift control function must have been specified during a prior microinstruction using a LeftShift (Section 3.5.2.1), RightShift (Section 3.5.2.2), Rotate

(Section 3.5.2.3) or ShiftOnR (Section 3.5.1.15) special function.

NextOp directs the contents of the OP File byte currently pointed to by the BPC onto the AMux output lines, and BPC is incremented by one.

NOTE: If BPC overflows, the microinstruction eventually executes again. This precludes instructions such as "Foo := NextOp + Foo".

The microassembler automatically adds an "IF BPC[3] Goto(Refill)" jump clause to any instruction containing a NextOp. Thus, if BPC overflows, control passes to Refill. The instructions at 'refill' must increment UPC by four (pointing to the next quadword), set BPC to 0 (clearing the overflow), and start a Fetch4 to the Op File (loading the next quadword). The special function LoadOp (see Section 3.5.1.2) must be executed in the t1 after the Fetch4 to cause the Op file to be loaded with the data coming in on MDI. Refill must then jump back to the instruction which needed the byte so that the instruction may be re-executed. This is easily accomplished with a 'ReviveVictim' jump directive (see Section 3.4.2.10).

The PERQ1a and PERQ1b can read the victim register directly (see Section 3.3.3.4).

WARNING: When a NextOp executes with BPC[3] set, the current address is loaded into the Victim register and locked. No new value is loaded until the Victim register is cleared by the ReviveVictim jump or by reading the Victim register directly.

IOD directs the contents of the IO databus to the 16 low bits of the AMux output. Bits 19..16 (23..16 for PERQ1b) are cleared.

MDI extracts a memory word; the contents of the Memory Data Bus are directed to the low-order 16 bits of the AMux output (AMux[15:0]). The remaining high order bits (AMux[19:16]--AMux[23:16] for PERQ1b) of the output are cleared.

MDI is valid only during the four cycles which immediately follow a fetch-type instruction. Section 5.2 contains details of memory control timing.

MDX extracts a memory extension; the low order 4 bits (8 for PERQ1b) of the Memory Data Bus (MD[3:0]; MD[7:0] for PERQ1b) are directed to AMux[19:16] (AMux[23:16] for PERQ1b). Bits AMux[15:0]

are cleared. MDX is used in conjunction with MDI to obtain a full 20 bit (24 bit for PERQ1b) physical address.

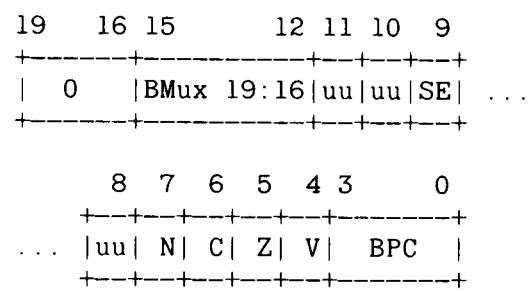
For example, a 20 bit physical address at memory address FOO can be placed in register BAR as follows:

```
MA:= FOO, Fetch2; ! initiate the Fetch
                  ! sequence
BAR := MDI;       ! get the first word
BAR := BAR Or MDX;! get the rest
```

TOS directs the contents of the top of the Expression stack to the AMux output.

UState [(Register_Name)] directs the contents of the microstate register to the AMux output on all CPUs. If Register_Name is specified, UState[15:12] contains the inverted four bits 19:16 of the selected register (using the Y field and BMux). If Register_Name is not specified, UState[15:12] contains Bmux[19:16] (inverted).

The UState register contains various interesting items packed in a single word. The UState register (A=5) looks like:



uu	Unused
BPC	Byte Program Counter
N	Negative (ALU result < 0)
Z	Zero (ALU result = 0)
C	Carry (ALU carry out of bit 15)
V	Overflow (ALU overflow occured)
SE	EStk Empty (inverted data -- 0 = empty)
BMux 19:16	Upper 4 Bits of BMux, used to read bits 19:16 of a register (inverted data).

NOTE: UState[15:12] contain the *inverted* register or BMux bits on PERQ1 and PERQ1a. They can be complemented again (using the NOT function, Section 3.3.1) and a field qualifier can provide a mask. For example,

```
Not UState(aRegister), Field(14,4);
aResult := shift;
```

Upper [(Register_Name)], on PERQ1b only, supplies the upper 8 bits of BMux[23:16] on Upper[7:0] inverted. Ustate[15:12] are always 0 for PERQ1b.

3.3.2.2. BMux

The microassembler uses the following primitives to control the BMux. For PERQ1 and PERQ1a, the 20 BMux output bits (BMux[19..0]) form the B input to the ALU. For PERQ1b, BMux[23:0] form the B input.

The syntax is

BMux ::= register | constant | '(' ConstExpr ')'

Register directs the contents of Register to the BMux output. The register name must have been previously defined (see Section 3.1.1).

Constant directs a one or two byte constant onto the low order 8 or 16 bits of the BMux output (BMux[7:0] or BMux[15:0]). The high-order 4 bits of the BMux output (BMux[19:16]--BMux[23:20] for PERQ1b) are cleared. The microassembler determines the required length of the constant (one or two bytes) and uses the F and SF fields accordingly.

Constant may be either implicit (through the use of a predefined constant name; see Section 3.1.2) or literal.

3.3.2.3. Operators

The microassembler recognizes several operators. These keywords are used by the microassembler to determine which ALU functions are to be enabled.

The syntax is:

Op = 'And' | 'Or' | 'Xor' | 'Nand' | 'Nor' | 'Xnor'.

All of these instructions use the ALU field of the microinstruction.

And - R gets the bitwise logical And of AMux and BMux.

Or - R gets the bitwise logical Or of AMux and BMux.

Xor - R gets the bitwise logical Xor of AMux and BMux.

Nand - R gets the inverted bitwise logical And of AMux and BMux.

Nor - R gets the inverted bitwise logical Or of AMux and BMux.

Xnor - R gets the inverted bitwise logical Xor of AMux and BMux.

3.3.2.4. OldCarry bit

The OldCarry bit, used in several ALU constructions, contains the carry (or borrow) from bit 16 of the immediately preceding microinstruction, and is used to perform multiple precision arithmetic operations.

OldCarry contains a carry bit if the immediately preceding

microinstruction was an addition and a borrow bit if the immediately preceding microinstruction was a substraction.

3.3.3. Constructions

This section presents valid ALU constructions. They are grouped into constructions that use one operand, constructions that imply logical operations between two operands, constructions that cause arithmetic operations among two operands (sometimes including the oldcarry bit for multiple precision arithmetic), and several special constructions used primarily for diagnostic purposes.

For each of these instructions, result, AMux, BMux, the operator, and OldCarry is specified using the syntax defined in the previous sections.

Each of these constructions uses the ALU field of the microinstruction in addition to the fields used by the elements of the construction.

The syntax is

```
ALU = ['Not'] (AMux | BMux)
    | Amux Op ['Not'] BMux
    | AMux '+' BMux ['+' 'OldCarry']
    | AMux '-' BMux ['-' 'OldCarry']
    | AMux 'AMUX' BMux
    | AMux 'BMUX' BMux
    | 'MQ'
    | 'Victim' .
```

3.3.3.1. Single operand constructions

Single operand instructions extract either AMux or BMux. They may be preceded by the keyword Not, causing the bitwise inversion of the operand.

The syntax is

['Not'] (Amux | Bmux)

Amux directs the AMux outputs to R[19:0] (R[23:0] for PERQ1b). When preceded by the keyword Not, the sense of R[19:0] (R[23:0]) is inverted. Amux is specified using one of the forms described in Section 3.3.2.1.

Bmux directs the BMux outputs to R[19:0] (R[23:0]). When preceded by the keyword Not, the sense of R[19:0] (R[23:0]) is inverted. Bmux is specified using one of the forms described in Section 3.3.2.2.

3.3.3.2. Double operand logical constructions

These constructions are used to form logical combinations of the AMux and BMux outputs.

The syntax is

AMux Op ['Not'] BMux

AMux Operator BMux uses one of the operators described in Section 3.3.2.3, and combines the AMux outputs with the BMux outputs, placing the result in R[19:0] (R[23:0] for PERQ1b).

AMux Operator Not BMux combines the Amux outputs (specified in Section 3.3.2.1) with the INVERTED BMux outputs (specified in Section 3.3.2.2), placing the result in R[19:0] (R[23:0]).

NOTE: The following constructions are not implemented in the ALU:

```
AMux 'Nand' 'Not' BMux
AMux 'Nor'  'Not' BMux
AMux 'Xor'  'Not' BMux
AMux 'Xnor' 'Not' BMux
```

The microassembler disallows these four constructions.

3.3.3.3. Double operand arithmetic constructions

These constructions are used to form arithmetic combinations of the AMux and BMux outputs.

The syntax is:

```
  AMux '+' BMux ['+' 'OldCarry']
| AMux '-' BMux ['-' 'OldCarry']
  ...
```

AMux + BMux [+ OldCarry] forms the 20-bit binary sum of AMux and BMux (24-bit for PERQ1b). The OldCarry bit is the carry from AMux[15] and BMux[15]. Thus, for multiple precision arithmetic, the appropriate sequence is:

```
1. AMux + BMux ;          ! Low order word
2. AMux + BMux + OldCarry; ! High-order word,
```

To minimize overhead during normal arithmetic operations from memory, the condition codes are set based upon the low-order 16 bits of the result (R[15:0]). Thus, while the high-order bits of AMux and BMux do participate in the addition, and the result is maintained in R[19:16] (R[23:16] for PERQ1b), they *do not* participate in the state of the condition codes. This affects the outcome of conditional branches (see Section 3.4).

AMux - BMux [- OldCarry] forms the 20-bit binary difference of

AMux and BMux (24-bit for PERQ1b). The OldCarry bit is the borrow from AMux[15] and BMux[15]. Thus, for multiple precision arithmetic, the appropriate sequence is:

```
1. AMux - BMux ;            ! Low order word
2. AMux - BMux - OldCarry;  ! High-order word,
```

To minimize overhead during normal arithmetic operations from memory, the condition codes are set based upon the low-order 16 bits of the result (R[15:0]). Thus, while the high-order bits of AMux and BMux do participate in the subtraction, and the result is maintained in R[19:16] (R[23:16] for PERQ1b), they *do not* participate in the state of the condition codes. This affects the outcome of conditional branches (see Section 3.4).

The special operators AMUX and BMUX permit the state of both the AMux outputs and the BMux outputs to be specified while only one or the other is being directed to R[19:0] (R[23:0] for PERQ1b).

In certain situations, primarily diagnostic, this is a necessary addition to the machine's functionality. For example, an ALU diagnostic might use these operators to identify BMux bits that are inappropriately coupled to the ALU output, by setting both AMux and BMux outputs to a known value and checking that the ALU generates the correct output.

The syntax is:

```
    AMux 'AMUX' BMux
  | BMux 'BMUX' BMux
```

AMux AMUX BMux causes the AMux outputs to be directed to R[19:0] (R[23:0]), while simultaneously putting a known value on the BMux outputs.

AMux BMUX BMux causes the BMux outputs to be directed to R[19:0] (R[23:0]), while simultaneously putting a known value on the AMux outputs.

3.3.3.4. Special constructions

The PERQ1a and PERQ1b allow two more constructions to read the MQ (multiplier/quotient) and Victim registers. These are used as though they were ALU operations. For example:

```
Result := MQ;
```

reads the MQ register and assigns its value to the register named Result. The Victim register (see Section 3.3.2.1) is read in a similar way. Reading the Victim register clears it an makes it possible to load a new Victim value. The Victim register is defined only from the time a NextOp is executed with BPC[3] set until it is read or used in a ReviveVictim jump. Therefore the Victim register may only be read once.

WARNING: Reading MQ and Victim does not actually use the ALU and thus no computation may be performed with the value. Reasonable actions are to assign the value to a register or send it to the shifter. Note that since the value does not pass through the ALU, condition codes are invalid in the cycle that follows. This affects the outcome of conditional branches (see Section 3.4).

3.4. Jump

Jump microinstructions are used to alter the sequential control flow of a microprogram. A Jump microinstruction is constructed from an optional condition, a directive, and a target address.

A jump requiring an address normally gets it from the Z field. However, since Z is only eight bits wide, and the controlstore requires a 12 bit address, another four bits of address are needed.

The microassembler derives these other four bits based upon the target of the jump. Short jumps branch to a location on the same 256-word page as the current microinstruction (CIA). In order to branch to an arbitrary location in the control store, the F field specifies a long jump (F=3), which uses the SF field for the upper four bits of address.

The PERQ1a and PERQ1b have a writable controlstore that is expandable in multiples of 4K up to 16K instructions. These 4K multiples are referred to as banks of the controlstore. The PERQ1a and PERQ1b microinstruction address paths have been expanded to provide 2 more address bits. Hardware has been added to provide 2910-like functions for the high order bits. This approach has numerous drawbacks because the 2910 is not an expandable bit slice. These problems are twofold. First, it is not possible to expand the 12-bit counter on the 2910 since it does not provide a carry out. Secondly, instructions such as 'RepeatLoop' and 'ThreeWayBranch' manipulate the control stack depending on the state of the counter. The state of the counter is not available externally to the chip. This can lead to the two stacks getting "out of sync" with each other.

The upper 2 and the lower 12 bits of the microaddress are treated differently. Sequencer instructions are different for the upper and lower bits. The jump control table shows the differences between them.

In addition to Short and Long jumps, there is a level of address generation called Leap that is capable of addressing the entire 16K controlstore. A 2-bit bank register supplies the current bank address during Long and Short jumps. When Leap is done, however, the 14-bit address is formed from concatenation of the Y and Z fields, with the Y field supplying the most significant byte. The assembler generates leap addresses (by setting F = 1 and SF = 7) when the target address of a jump is in another bank.

The assembler restricts a single assembly to one bank. This means leaps are required only for constant addresses in another bank.

The following notes are provided as an aid to reading the jump control table for PERQ1a and PERQ1b CPUs:

1. Incrementing or decrementing the microinstruction program counter and the S register does not affect the upper 2 bits. Thus these will not cross bank boundaries.

2. The low 12 bits of S are not available to the circuitry controlling the upper 2 bits of S and the upper 2 bits of the microinstruction program counter. Thus jumps that depend on whether S has reached zero will not cross bank boundaries, and are allowed to push or pop only the lower 12 bits of the CStk.

3. Since Repeat, RepeatLoop, Loop, and ThreeWayBranch only pop the lower 12 bits of the CStk, PushLoad only pushes the lower 12 bits of the CStk. This means that the short version of JumpPop must be used to exit such a loop.

4. Since Call and CallS push both parts of the CStk, the long version of JumpPop (called LeapPop) must be used to clear the call stack.

5. NextInst, Vector, and Dispatch jumps may not be used across bank boundaries.

The microassembler provides minimal support for Leaping jumps. A single assembly must fit entirely inside a 4K bank of the controlstore. This means that Leaps are allowed only with constant addresses or with the Goto(Shift) described in Section

3.4.3.3. The assembler, therefore, always knows when to generate a Leap.

Since the JumpPop type has two variants to control whether to pop the upper stack, a new jump type is necessary: LeapPop. This is a JumpPop that pops the upper stack.

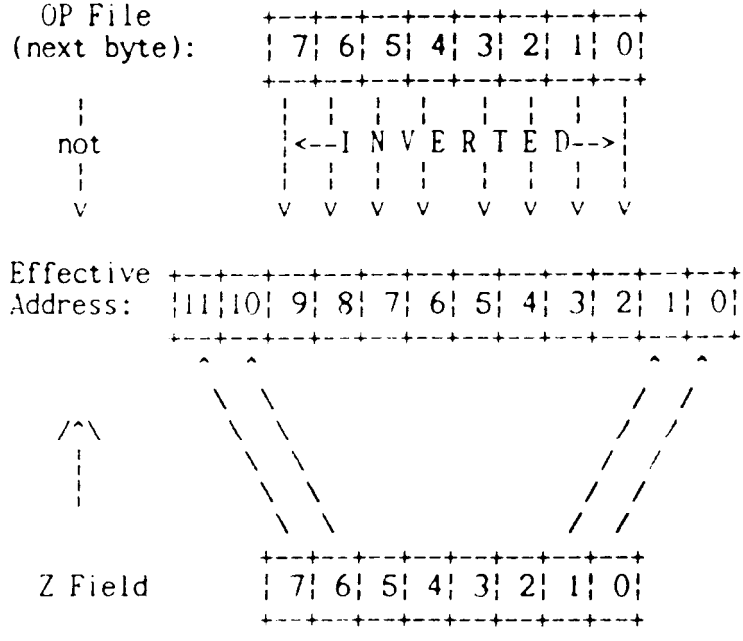
For all three CPUs, the address for jumps might not come from the Z, SF, or Y fields.

In addition to control-store addresses that come from the Z and SF fields, a jump address can also come from the following sources:

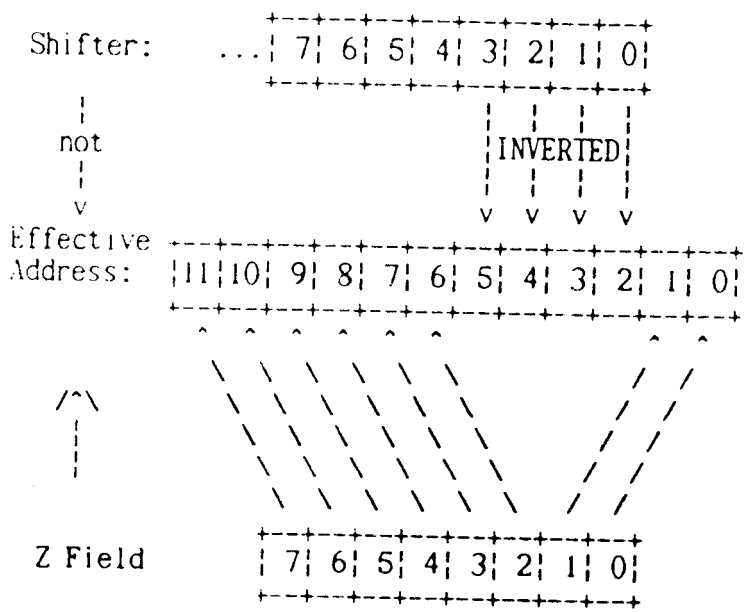
SOURCE	DESCRIPTION
S register	Internal to the microsequencer
Call Stack	Five-level stack internal to the microsequencer
Current Address + 1	Uses full 12-bit Address
Victim Register	Hardware register that contains the address of the most recent 'failed' NextOp.
Processor Shifter	For PERQ1a and PERQ1b.

There are three jumps which are multi-way branches. For each of these three jumps, the Z field provides the address bits needed to complete the 12-bit control-store effective address. The details of address generation for these three multi-way branches are as follows:

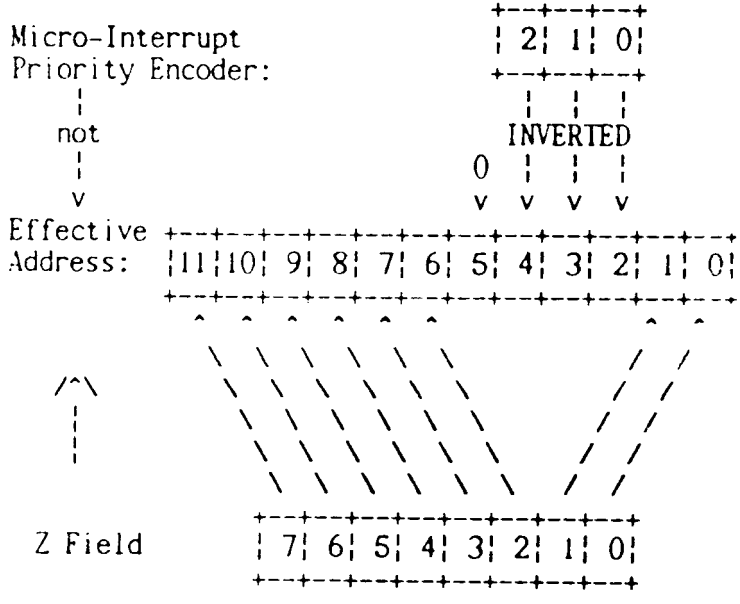
DIRECTIVE	DESCRIPTION
NextInst	256-way branch, based on the next byte from the Op file. The next byte provides bits 9..2 of the effective address (inverted); bits 11..10 come from Z[7:6] and bits 1..0 come from Z[1:0]. This results in a 256-way branch with a spacing of four instructions.



DIRECTIVE	DESCRIPTION
Dispatch	16-way branch, based on the low-order four bits of the Shifter. Bits 5..2 come from the shifter (inverted); the remainder come from Z. This results in a 16 way branch with a spacing of four instructions.



DIRECTIVE	DESCRIPTION
Vector	8-way branch, based on the three micro-interrupt priority encoder bits (the V bits). Bits 4..2 come from the priority encoder (inverted), bit 5 is always zero, and the remaining bits come from Z. This results in an eight way branch with a spacing of four microinstructions.



The syntax is:

```
Jump = ['If' Condition] Directive ['(' Target ')'] .

Condition = 'True' | 'False' | 'BPC[3]' | 'C19'
          | 'IntrPend' | 'Odd' | 'ByteSign'
          | 'Eql' | 'Neq' | 'Gtr' | 'Geq'
          | 'Lss' | 'Leq' | 'Carry' | 'OverFlow'.

Directive = 'Goto' | 'Call' | 'Return' | 'Next'
    | 'JumpZero' | 'LoadS' | 'GotoS' | 'CallS'
    | 'NextInst' | 'ReviveVictim' | 'PushLoad'
    | 'Vector' | 'Dispatch' | 'RepeatLoop'
    | 'Repeat' | 'JumpPop' | 'LeapPop' | 'Loop'
    | 'ThreeWayBranch' .

Target = label | constant | 'Shift' .
```

3.4.1. Jump conditions

Conditions are used to control whether or not a given jump is taken. Some jump directives do not allow conditions, while for the remainder the condition is optional. Section 3.4.2 presents in more detail which directives do or do not allow condition execution. The possible jump conditions are described in 3.4.1.1-3.4.1.12.

All ALU related condition codes test the ALU operation from the previous microcycle. Thus, the normal sequence is to perform an ALU operation and then test its operands in the next microinstruction. For example, comparison of two registers A and B is accomplished in this way:

```
A - B;
if Gtr Goto(Label);    ! Jumps if A > B; another ALU
                       ! may be performed here
```

All ALU tests, with the exception of C19 (C23) test the lower 16

bits of the ALU. They are intended for data comparisons. After addition, these condition codes compare the signed result against zero. If overflow has occurred, this will be accounted for; i.e., adding two numbers with the same sign *always* yields a result with the same sign even if overflow occurs (see Section 3.4.1.15). After a subtraction, these condition codes compare the two operands as 16-bit signed numbers. After other operations, these condition codes compare the 16-bit signed ALU result against zero. (See Section 3.4.1.15.) Consider the following example:

```
A := A - B;
if Gtr call(x);    ! calls x if 16-bit signed A +
                   ! 16-bit signed B > 0
A;
if Gtr call(y);    ! calls y if 16-bit signed A > 0
```

In this example, x is called if 16-bit signed A is greater than 16-bit signed B, and y is called if the 16-bit signed result of the subtraction is greater than 0. These two tests do not necessarily act the same for the same values of A and B. For example, if A was #077777 and B was #177777, then x would be called but y would not. This is because A, which is positive, is greater than B, which is negative, but the result of the subtraction, when treated as a 16-bit signed number (#100000), is not greater than 0; the 16th bit of the result will be a 1, indicating that it is negative and therefore not greater than 0. Considering an addition example:

```
A := A + B;
if Gtr call(x);    ! calls x if 16-bit signed A >
                   ! 16-bit signed B
A;
if Gtr call(y);    ! calls y if 16-bit signed A > 0
```

X is called if 16-bit signed A plus 16-bit signed B is greater than 0, and y is called if the 16-bit signed result of the addition is greater than 0. If, in this case, A was #077777 and B was #000001, x would be called but y would not. X is called because the result of an addition of two numbers of the same sign is

considered to have the same sign; A and B are both positive, so their sum is considered positive. Y is not called because the actual result of the addition (#100000), considered as a 16-bit signed number, is negative and therefore not greater than 0. This situation holds true of the following tests: Eql, Neq, Gtr, Geq, Lss, and Leq.

The condition codes are not entirely sensible after a double precision add or subtract:

```
ALower - BLower;
AUpper - BUpper - OldCarry;
if Lss Goto(AIsLessThanB);
```

The Z condition code flag considers only the result of the upper precision operation, not the result of the lower precision operation. This means that Eql, Neq, Leq, and Gtr condition codes (which use the Z flag) are not valid after a double precision add or subtract unless the low order 16 bits of the result are zero. Only Lss, Geq, Carry, and Overflow reflect the result of the entire double precision operation. The following subroutine may be used to compare the two double precision numbers:

```
DblCmp: ALower - BLower;
        ALower - BLower - OldCarry,
            if Eql Return;    ! if ALower = BLower, the
                              ! condition codes are good
            not 0, if Lss Return;    ! A < B
            1, Return;               ! A > B
```

After calling DblCmp you may use the Lss, Leq, Eql, Neq, Geq, and Gtr condition codes. Carry and Overflow, however, are not valid.

C19 is designed for unsigned address comparisons. After an addition, it contains the *inverted* carry from bit 19; after a subtraction, it contains the borrow from bit 19.

Assuming that A and B are registers containing 20-bit addresses and T is a temporary register, the following code fragments show how C19 may be used to compare A and B.

```
A - 1;                  ! Jumps if A = 0;
if C19 Goto(label);     ! doesn't jump if A <> 0

T := A;
T := T - B;
T - 1;                  ! Jumps if A = B;
if C19 Goto(label);     ! doesn't jump if A <> B

A - B;                  ! Jumps if A < B;
if C19 Goto(label);     ! doesn't jump if A >= B

B - A;                  ! Jumps if A > B;
if C19 Goto(label);     ! doesn't jump if A <= B
```

The syntax is

condition ::= 'If true' | 'If false'
| 'If BPC[3]' | 'If C19'
| 'If IntrPend' | 'If Odd'
| 'If ByteSign' | 'If Eql'
| 'If Neq' | 'If Gtr'
| 'If Geq' | 'If Lss'
| 'If Leq' | 'If Carry'
| 'If Overflow

On PERQ1b, C19 is replaced by C23, and the above comparison works for 24-bit addresses.

3.4.1.1. True

Always Pass. Equivalent to no conditional.

3.4.1.2. False

Never Pass.

3.4.1.3. BPC[3]

Pass if BPC overflow has occurred (BPC[3] = 1). This implies that the Op file is empty.

3.4.1.4. C19 (C23)

Pass if the inverted carry from ALU bit 19 of the last microinstruction was set. The sense of C19 is inverted, so if C19 is reset, a Pass occurs and if C19 is set, a Fail occurs.

On PERQ1b, C23 checks ALU bit 23 and passes as for C19.

3.4.1.5. IntrPend

Pass if any device is requesting an interrupt.

3.4.1.6. Odd

Pass if ALU bit 0 of the last microinstruction was set.

3.4.1.7. ByteSign

Pass if ALU bit 7 of the last microinstruction was set.

3.4.1.8. Eql

Pass if equal to zero. After a subtraction, this tests whether the two operands were equal. Otherwise, this tests whether the result was equal to zero. Eql is a 16-bit test. Refer to Section 3.4.1 for information about how to properly use this jump condition.

3.4.1.9. Neq

Pass if not equal to zero. After a subtraction, this tests whether the two operands were unequal. Otherwise, this tests whether the result was unequal to zero. Neq is a 16-bit test. Refer to Section 3.4.1 for information about how to properly use this jump condition.

3.4.1.10. Gtr

Pass if greater than zero. After a subtraction, this tests whether the A-input was greater than the B-input. Otherwise, this tests whether the result was greater than zero. Gtr is a 16-bit test. Refer to Section 3.4.1 for information about how to properly use this jump condition.

3.4.1.11. Geq

Pass if greater than or equal to zero, whether or not an overflow occurred. After a subtraction, this tests whether the A-input was greater than or equal to the B-input. Otherwise, this tests whether the result was greater than or equal to zero. Geq is a 16-bit test. Refer to Section 3.4.1 for information about how to properly use this jump condition.

3.4.1.12. Lss

Pass if less than zero. After a subtraction, this tests whether the A-input was less than the B-input. Otherwise, this tests whether the result was less than zero. Lss is a 16-bit test. Refer to Section 3.4.1 for information about how to properly use this jump condition.

3.4.1.13. Leq

Pass if less than or equal to zero. After a subtraction, this tests whether the A-input was less than or equal to the B-input. Otherwise, this tests whether the result was less than or equal to zero. Leq is a 16-bit test. Refer to Section 3.4.1 for information about how to properly use this jump condition.

3.4.1.14. Carry

Pass if carry bit from bit 15 of the previous instruction was set. The ALU performs subtraction using two's complement arithmetic, and so the carry bit after a subtraction is a two's complement carry.

3.4.1.15. Overflow

Pass if a 16-bit overflow occurred in the previous microinstruction. Overflow is set under two conditions, as follows:

Sign of A, B Inputs	Operation	Result Sign
SAME	Addition	Different
DIFFERENT	Subtraction	Different from A

3.4.2. Jump directives

Each directive specifies an action to take. Some directives do not allow a conditional to be specified (see Section 3.4 for details on specifying the conditional). Some directives require a target to be specified, for some a target is optional, and for some the target is implied by the directive and cannot be explicitly specified. Targets are described in more detail in Section 3.4.3.

The description for each directive includes whether or not a

conditional and/or target can be supplied.

Further information about details of the jump instructions is available in the documentation for the AMD 2910 microsequencer (the *Bipolar Micrprocessor Logic and Interface Data Book*, from Advanced Micro Systems).

The syntax is

```
Directive = 'Goto' | 'Call' | 'Return' | 'Next'
    | 'JumpZero' | 'LoadS' | 'GotoS' | 'CallS'
    | 'NextInst' | 'ReviveVictim' | 'PushLoad'
    | 'Vector' | 'Dispatch' | 'RepeatLoop'
    | 'Repeat' | 'JumpPop' | 'LeapPop' | 'Loop'
    | 'ThreeWayBranch' .
```

The jump directives are described in Sections 3.4.2.1-3.4.2.18.

3.4.2.1. Goto

If the optional condition succeeds, or if no condition is supplied, branches to the specified target. If the optional condition fails, execution continues at the current instruction address plus one.

The condition is optional; the target is required.

3.4.2.2. Call

If the optional condition succeeds, or if no condition is supplied, pushes the current instruction address plus one onto the call stack and branches to the target. If the optional condition fails, execution continues at the current instruction address plus one.

The condition is optional; the target is required.

3.4.2.3. Return

If the optional condition succeeds, or if no condition is supplied, branches to the value of the top of the call stack and pops the call stack. If the optional condition fails, execution continues at the current instruction address plus one.

The condition is optional; the target cannot be specified.

3.4.2.4. Next

Execution continues at the current instruction address plus one.

The condition cannot be specified; the target cannot be specified.

3.4.2.5. JumpZero

Jumps to address zero in the control store.

The condition cannot be specified; the target cannot be specified.

3.4.2.6. LoadS

Load the target into the S register. The S register is an address register internal to the microsequencer.

The condition cannot be specified; the target is required.

3.4.2.7. GotoS

There are three cases. If the optional condition succeeds and the optional target is supplied, the hardware branches to the target. If no target is supplied, the hardware branches to the next microinstruction in the source. If the condition fails or if no condition is supplied, execution continues at the microinstruction at S.

The condition is optional if there is no target and required if there is a target. Unlike other jumps, the default condition is False

which uses the value of the S register. The target is optional.

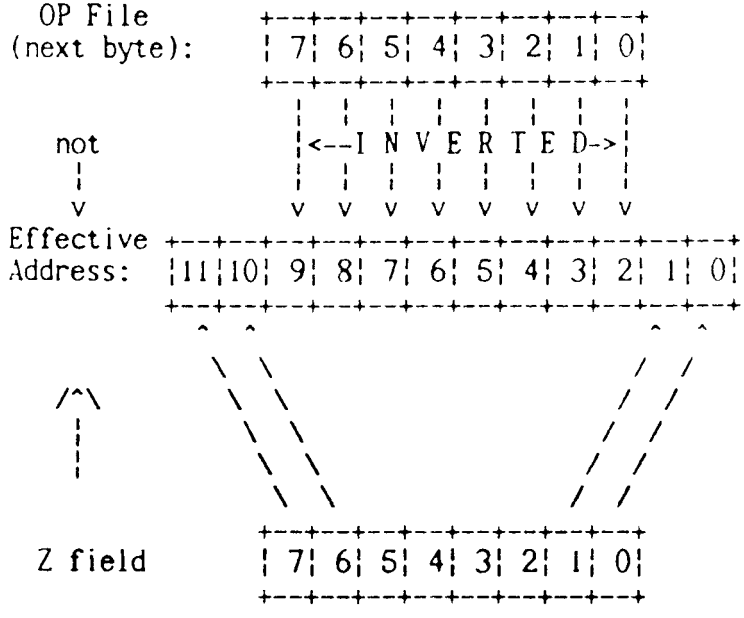
3.4.2.8. CallS

If the optional condition succeeds the hardware branches to the target and pushes the current instruction address plus one onto the call stack. If the condition fails, or if no condition is supplied, the hardware pushes the current instruction address plus one onto the call stack and execution continues at control store address S.

The condition is optional if there is no target and required if there is a target. Unlike other jumps, the default condition is False which uses the value of the S register. The target is optional.

3.4.2.9. NextInst

A 256-way branch, based on the next byte from the Op file. The next byte provides bits 9..2 of the effective address (inverted); bits 11..10 come from Z[7:6] and bits bits 1..0 come from Z[1:0]. This results in a 256-way branch with a spacing of four instructions.



If the Op file is empty (BPC[3] = 1) then the OP File bits are forced to #377. Since these bits are inverted in the effective address, control passes to the base address generated from the Z bits (effective address bits 9..2 are zero). It is necessary to place a special refill function at this address ('Zaddr') which fetches four new bytes, increments UPC by four, performs a LOADOP (see Section 3.5.1.2), and repeats the NEXTINST. The return from this special refill should be via NextInst.

The Z field is derived from the target by the assembler (see Section 3.4.3).

The condition cannot be specified; the target is required.

3.4.2.10. ReviveVictim

Control branches to the value of Victim register. The victim register contains the address of the most recent microinstruction that included a NextOp while BPC[3] was set.

NOTE: The contents of the victim register are defined only from the time the nextOp is executed (with BPC[3] set) to the first execution of ReviveVictim. Therefore, ReviveVictim can be executed only once after a failed NextOp.

The condition cannot be specified; the target cannot be specified.

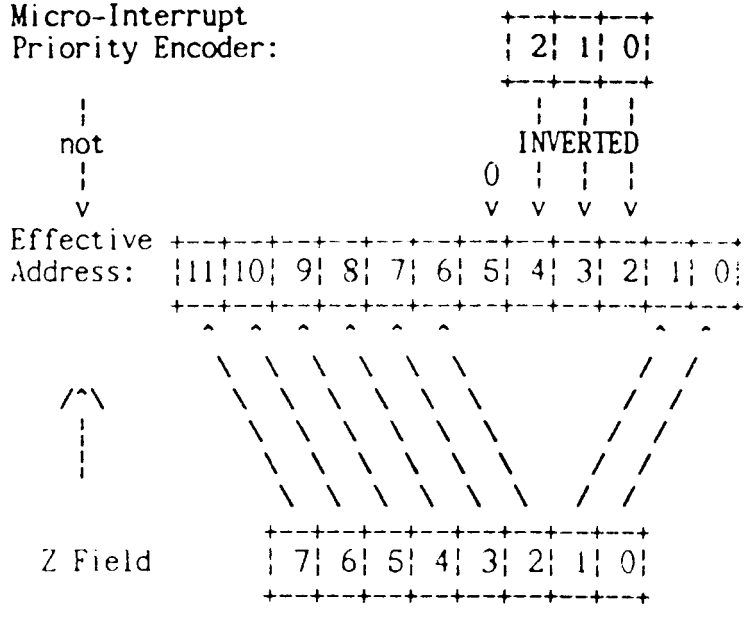
3.4.2.11. PushLoad

If the optional condition succeeds or if no condition is supplied, execution continues at the current instruction address plus one, the current instruction address plus one is pushed onto the call stack, and the target is stored in S. If the condition fails, the current instruction address plus one is pushed onto the call stack and execution continues at that address.

The condition is optional; the target is required.

3.4.2.12. Vector

An up-to-eight way branch, based on the three micro-interrupt priority encoder bits (the V bits). Bits 4..2 come from the priority encoder (inverted), bit 5 is always zero, and the remaining bits come from Z. This results in an eight way branch with a spacing of four microinstructions.

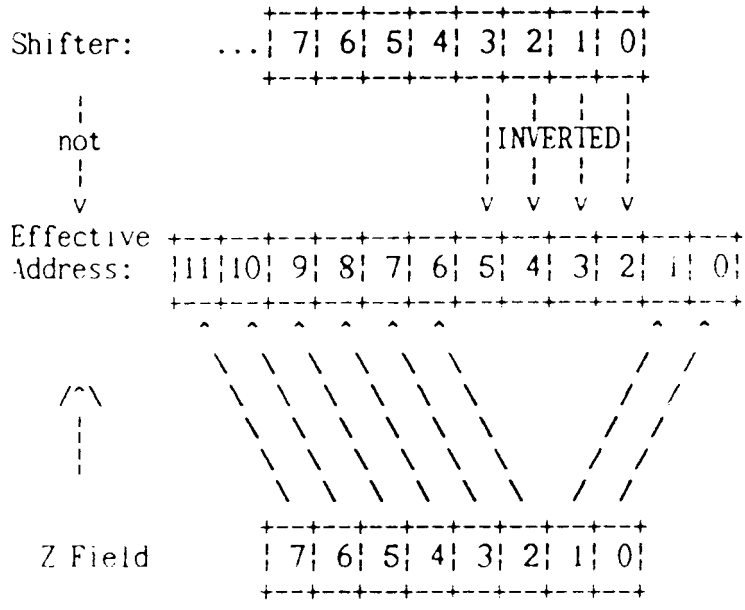


The microassembler derives the Z field from the target (see Section 3.4.3).

If the optional condition succeeds, or if no condition is supplied, execution continues at the effective address. If the condition fails, execution continues at the current instruction address plus one. The condition is optional; the target is required.

3.4.2.13. Dispatch

A 16 Way branch, based on the low-order four bits of the Shifter. Bits 5..2 come from the shifter (inverted); the remainder come from Z. This results in a 16 way branch with a spacing of four instructions.



The Z field is derived from the target by the microassembler (see Section 3.4.3 for details).

If the optional condition succeeds, or if no condition is supplied, execution continues at the effective address. If the condition

fails, execution continues at the current instruction address plus one.

The condition is optional; the target is required.

3.4.2.14. RepeatLoop

If S is non-zero, execution continues at the instruction whose address is at the top of the call stack and S is decremented by one.

If S is zero, execution continues at the current instruction address plus one and the call stack is popped.

The conditional cannot be specified; the target cannot be specified.

3.4.2.15. Repeat

If S is non-zero, execution continues at the target address and S is decremented by one.

If S is zero, execution continues at the current instruction address plus one.

The condition cannot be specified; the target is required.

3.4.2.16. JumpPop and LeapPop

If the optional condition succeeds, or if no condition is supplied, execution continues at the target address and the call stack is popped. If the condition fails, execution continues at the current instruction address plus one.

LeapPop is used on the PERQ1a and PERQ1b CPUs to pop both the upper and lower call stacks. LeapPop is encoded by the assembler as a JumpPop with the Hold bit set. See Section 3.4

for a description of the upper and lower call stacks of the PERQ1a and PERQ1b.

The condition is optional; the target is required.

3.4.2.17. Loop

If the optional condition succeeds, or if no condition is supplied, execution continues at the current instruction address plus one, and the call stack is popped. If the condition fails, then execution continues at the address specified by the top of the call stack.

The condition is optional; the target cannot be specified.

3.4.2.18. ThreeWayBranch

If the optional condition succeeds, or no condition is supplied, then execution continues at the current instruction address plus one. The call stack is popped, and if S is non-zero, S is decremented.

If the condition is supplied and fails, then if S is non-zero, execution continues at the address specified by the top of the call stack and S is decremented. If S is zero, execution continues at the target address and the call stack is popped.

The condition is optional; the target is required.

3.4.3. Targets

Targets are used by the microassembler to derive the appropriate contents for the F, SF, Z fields. The microassembler determines whether or not a jump is long or short and sets the F and SF fields accordingly.

The syntax is

target = label | constant | 'Shift' .

3.4.3.1. Label

A label is formed from a letter followed by an arbitrary number of letters or digits. The microassembler will use, as the target address of the jump, the control store address of the instruction labelled with the corresponding name (see Section 2.1 for details of microinstruction labelling).

3.4.3.2. Constant

A constant is either an explicit constant constructed from a sequence of digits (optionally preceded by a radix indicator) or it may be a previously defined name (see Section 3.1.2).

3.4.3.3. Goto(Shift)

On the PERQ1a and PERQ1b, the shifter outputs may be used as the address by typing the word "Shift" where a jump address is allowed. The shifter may be used with the following jump types: Call, Goto, PushLoad, CallS, GotoS, Repeat, JumpPop, LeapPop, and LoadS. The following example jumps to an address in a register named Addr.

```
Addr, RightShift(0);            ! run address through shifter
Goto(Shift);                    ! jump
```

The following example performs an n-way dispatch (n <= 4096) to address Addr + N * 4.

```
N, LeftShift(2);                ! multiply N by 4 by shifting
Shift + Addr, RightShift(0);    ! add base and send to shifter
Goto(Shift);                    ! jump
```

3.5. Special Functions

Memory references, Writable Control Store (WCS) references, certain RasterOp control functions, and several housekeeping operators are handled by the microassembler as special functions.

Special functions requiring no arguments (Nonary functions) are described in Section 3.5.1, special functions requiring one

argument are described in Section 3.5.2, and special functions requiring two arguments are described in Section 3.5.3.

The syntax is:

```
Special = Nonary | Unary | Binary .

Nonary = 'WCSlow' | 'WCSmid' | 'WCShi'
       | 'LoadOp' | 'Hold' | 'StackReset' | 'Push'
       | 'Pop' | 'Fetch' | 'Fetch2' | 'Fetch4'
       | 'Fetch4R' | 'Store' | 'Store2'
       | 'Store4' | 'Store4R' | 'ShiftOnR'
       | 'MultiplyStep' | 'DivideStep' .

Unary = UnaryName '(' ConstExpr ')' .

UnaryName = 'LeftShift' | 'RightShift' | 'Rotate'
       | 'IOB' | 'CntlRasterOp' .

Binary = BinaryName '('ConstExpr',' ConstExpr')'.

BinaryName  = 'Field' .
```

3.5.1. Nonary

Nonary functions require no arguments. It should be noted that the memory reference functions (the fetch-type and store-type functions) require that specific timing constraints be satisfied. These constraints are detailed in Section 5.2. Other constraints or specific timing requirements are detailed in the relevant section.

3.5.1.1. WCS directives

The WCS directives are used to write locations within the writeable control store. The microinstruction word is 48 bits long, so each WCS word is written as three 16-bit words, using WCSLow, WCSMid and WCSHi. In order to write a location of the controlstore, the address which is to be written should be placed in the S register with a LoadS instruction. The WCSLow, WCSMid, and WCSHi functions are executed in three successive microinstructions together with the jump and condition codes "if True GotoS." During each one of these WCS functions, 16 bits of a microinstruction is written with the value of the ALU result from the previous microinstruction. It is important that the jump and condition codes be written correctly, as it is this part of the microinstruction that supplies the address.

Each instruction that contains a WCSLow, WCSMid, or WCSHi takes 340 nanoseconds to complete because it is executed twice. This precludes instructions which change their own initial conditions such as

```
R := R + 1, WCSlow
```

The following example writes a microinstruction with the values in three registers. The example does not show values being loaded into the registers, but this must be done prior to the execution of the WCSLow, WCSMid, and WCSHi functions.

```
   Define(LowWord, 100);
   Define(MidWord, 101);
   Define(HighWord, 102);

   LowWord, LoadS(MicroAddress);
   MidWord,  WCSLow, if True GotoS(A);
A: HighWord, WCSMid, if True GotoS(B);
B:           WCSHi,  if True GotoS(C);
C:
```

Note that since each directive takes two microcycles to complete, a total of at least 6 microcycles is needed to write any location in the writeable control store.

The WCSLow directive causes the contents of R to be written into the low order 16 bits of a location in the Writeable Control Store (WCS[15:0]).

NOTE: WCSLow requires two microcycles to complete.

The WCSMid directive causes the contents of R to be written into the middle 16 bits of a location in the Writeable Control Store (WCS[31:16]).

NOTE: WCSMid requires two microcycles to complete.

The WCSHi directive causes the contents of R to be written into the high order 16 bits of a location in the Writeable Control Store (WCS[47:32]).

NOTE: WCSHi requires two microcycles to complete.

3.5.1.2. LoadOp

The PERQ workstation provides special hardware to fill the Op file (in order to improve the performance of refill). After a Fetch4, a LoadOp specified in the t1 immediately following causes the hardware to load four words into the Op file without further microcode assistance.

The LoadOp must be given in the t1 immediately following the Fetch4. The instruction which follows the LoadOp can go back to the NextInst/NextOp since the first byte is guaranteed to be in. The three remaining words arrive and are placed in the Op file

without microcode assistance.

NOTE: If BPC is non-zero (to start reading in the middle of the quadWord), the refill code must wait until the correct byte is in the Op file.

3.5.1.3. Hold

The Hold function is used to inhibit IO devices from accessing memory. The IO system can request memory cycles at any time. The memory system gives priority to the IO system so that if both the processor and the IO system make memory requests, the IO is served first while the processor is delayed. IO requests are locked out during the execution of a microinstruction with the hold function set. To be effective, hold must be set in a t2. This is necessary only while doing overlapped memory references (see Sections 3.5.1.9 through 3.5.1.15).

The hold function must not be set all the time; an IO reference must be permitted at least once every three memory cycles.

3.5.1.4. StackReset

The StackReset special function empties the expression stack. The StackEmpty (SE) bit in the microstatus word (UState[8]) is asserted (to the zero state) when this function is executed, and the DDS is incremented.

NOTE: The result of operations other than Push that reference the stack while the stack is empty are implementation-dependent and should be considered undefined. Figure 3.5.1.6 gives several examples of manipulating the expression stack.

3.5.1.5. Push

The Push special function increments the expression stack pointer. Data is placed on the stack through an assignment using the Push special function, as follows:

```
TOS := Data, Push;    ! Pushes "Data" on the stack.
```

A Push into the StackTop (the 16th push) causes the stack empty bit to become set (to the zero state). This indicates that subsequent pushes (without intervening pops) produce undefined results.

NOTE: The result of a Push while the stack is full is implementation-dependent and should be considered undefined. Figure 3.5.1.6 gives several examples of manipulating the expression stack.

3.5.1.6. Pop

The Pop special function decrements the expression stack pointer. One or more pops while the stack is empty (with no intervening pushes) produces undefined results.

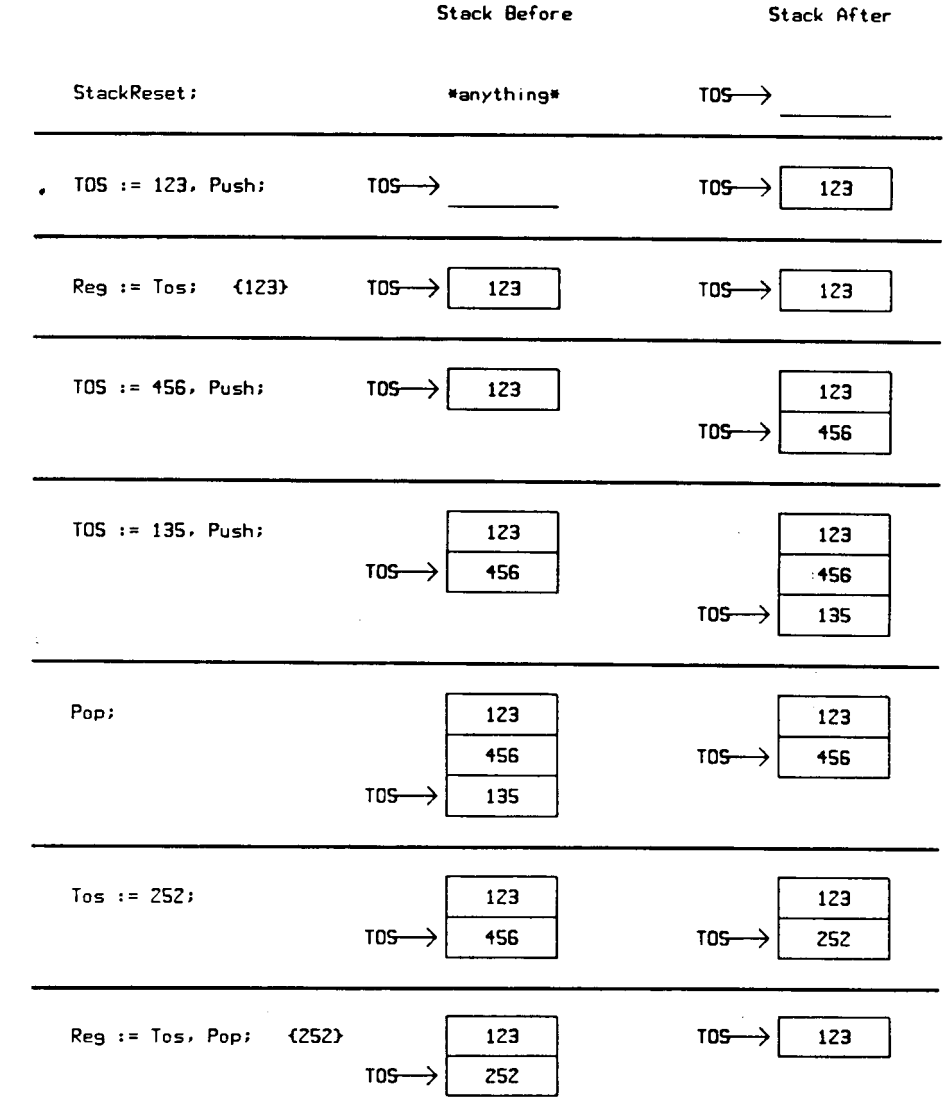
NOTE: The result of Pop while the stack is empty is implementation dependent and should be considered undefined. Figure 3.5.1.6 gives several examples of manipulating the expression stack.

3.5.1.7. Fetch

The Fetch special function initiates a one word memory data read sequence. The memory address for a Fetch is latched from the contents of the R register at the time that the Fetch is recognized (t3).

The data from a fetch is available from MDI or MDX at the next t2. If MDI or MDX is used during the intervening t0 and t1, the

Figure 3: Examples of Expression Stack Functions



processor is suspended until t2.

Any address may be used with a Fetch, and the resulting data word may be read in any cycle from t2 until the following t1 (inclusive).

3.5.1.8. Fetch2

The Fetch2 special function initiates a double word memory data read sequence. The memory address for a Fetch2 is latched from the contents of the R register at the time that the Fetch2 is recognized (t3).

The first data word returned from a Fetch2 (word 0 of the double word) must be read at the next t2; the second word (word 1 of the double word) must be read at the next t3. The state of MDI and MDX is undefined in succeeding cycles. If MDI or MDX is used during the intervening t0 and t1 (after a Fetch2 at t3) the processor is suspended until t2.

The low-order address bit of a Fetch2 request is ignored, so that a Fetch2 is always double-word aligned (rounded down).

3.5.1.9. Fetch4

The Fetch4 special function initiates a quad-word memory data read sequence. The memory address for a Fetch4 is latched from the contents of the R register at the time that the Fetch4 is recognized (t3).

The first data word returned from a Fetch4 (word 0 of the quad word) must be read at the next t2; the second word (word 1 of the quad word) must be read at the next t3, the third (word 2 of the quad word) at t0, and the last (word 4 of the quad word) at t1. The state of MDI and MDX is undefined in succeeding cycles.

The two low-order address bits of a Fetch4 request are ignored, so that a Fetch4 is always quad-word aligned (rounded down).

3.5.1.10. Fetch4R

The Fetch4R special function initiates a quad-word memory data read sequence. The Fetch4R is exactly like the Fetch4, with the exception that the quad word is returned in reverse order. Thus, the high-order word is received first and the low-order word last. The memory address for a Fetch4R is latched from the contents of the R register at the time that the Fetch4R is recognized (t3).

The first data word returned from a Fetch4R (word 3 of the quad word) must be read at the next t2; the second word (word 2 of the quad word) must be read at the next t3, the third (word 1 of the quad word) at t0, and the last (word 0 of the quad word) at t1. The state of MDI and MDX is undefined in succeeding cycles.

The two low-order address bits of a Fetch4R request are ignored, so that a Fetch4 is always quad-word aligned (rounded down).

3.5.1.11. Store

The Store special function initiates a one word memory data write sequence. The memory address for a Store is latched from the contents of the R register at the time that the Store is recognized (t2).

Any address may be used with a Store. The data for a Store is supplied on R in the following t3.

3.5.1.12. Store2

The Store2 special function initiates a double word memory data write sequence. The memory address for a Store2 is latched from the contents of the R register at the time that the Store2 is recognized (t3).

The low-order bit of the address for a Store2 is ignored, so a Store2 always writes to a double-word aligned location (rounded down). The first data word for a Store2 (word 0 of the double word) is supplied on R during the next t0 and the second data word (word 1 of the double word) is supplied on R during the following t1.

3.5.1.13. Store4

The Store4 special function initiates a quad-word memory data write sequence. The memory address for a Store4 is latched from the contents of the R register at the time that the Store4 is recognized (t3).

The two low-order bits of the address for a Store4 are ignored, so a Store4 always writes to a quad-word aligned location (rounded down). The first data word for a Store4 (word 0 of the quad-word) is supplied on R during the next t0, the second data word (word 1 of the quad-word) is supplied on R during the following t1, the third data word (word 2 of the quad-word) is supplied on R during the following t2, and the last data word (word 3 of the quad-word) is supplied on R during the following t3.

3.5.1.14. Store4R

The Store4R special function initiates a quad-word memory data write sequence. The Store4R is exactly like the Store4, with the exception that the quad word is written in reverse order. Thus, the high-order word is written first and the low-order word last. The memory address for a Store4R is latched from the contents of the R register at the time that the Store4R is recognized (t3).

The two low-order bits of the address for a Store4R are ignored, so a Store4R always writes to a quad-word aligned location (rounded down). The first data word for a Store4R (word 3 of the quad-word) is supplied on R during the next t0, the second data word (word 2 of the quad-word) is supplied on R during the following t1, the third data word (word 1 of the quad-word) is supplied on R during the following t2, and the last data word (word 0 of the quad-word) is supplied on R during the following t3.

3.5.1.15. ShiftOnR

The ShiftOnR special function, by obtaining the shift control from the R register, allows a shift function to be a variable.

The shifter hardware can either rotate a 16-bit item 0 to 15 places (rightward), can right justify an arbitrarily positioned 0 to 15 bit subfield of an item, or it can right or left shift a 16-bit item 0 to 15 places.

The shifter hardware is driven by two four bit nibbles, encoded as follows (R has a 16 bit item to be shifted):

a) Mask Operations:

```
m :=  Mask Width (bits, one origin)
      1 <= m <= 16
i :=  Index of low-order bit of Mask
      0 <= i <= 15
```

```
      For i <= 16 - m,
                                i          m
         Result = {(R/2 ) and (2  -1)}

             [R is shifted right by i bits; then all
              but the low-order m bits are zeroed]

      Low-order Nibble  := m-1
      High-order Nibble := i

b)    LeftShift Operations:

      j :=  Number of places to
              shift, 0 <= j <= 15

      Low-order Nibble  := #17
      High-order Nibble := j

c)    Rotate Operations:

      j :=  Number of places to
              rotate, 0 <= j <= 15

      For 0 <= j <= 7,

         Low-order Nibble  := #15
         High-order Nibble :=  j + 8

      For 8 <= j <= 15,

         Low-order Nibble  := #16
         High-order Nibble :=  j

d)    RightShift Operations

      n := Number of places to RightShift,
           0 <= n <= 15

      Low-order Nibble  := 15 - n
      High-order Nibble := n
```

The usage sequence for the ShiftOnR special function is:

1. Put the shift control byte on R and execute ShiftOnR.
2. Put the item to be shifted on R.
3. Read the shifted result on shift. The shifter control logic always keeps the last shift control function loaded so that the shifter can shift a succession of words without respecifying the shift control function. The shift outputs always have the shifted value of what was last on R.

For example, to shift the data contained in a register called ShiftItem by an amount specified as a shift control byte in the register called ShiftCount and then store the shift result in the register ShiftResult use:

```
ShiftCount, ShiftOnR;   ! load the variable shift
                        ! control byte

ShiftItem;              ! load the data to be shifted
                        ! and shift it

ShiftResult:= Shift;    ! store the shifter results
```

3.5.1.16. MultiplyStep

The PERQ1a and PERQ1b CPUs have hardware to support single precision multiplication and division. The multiply / divide hardware consists of a 16-bit shift register called MQ and a control circuit. The function of the multiply / divide hardware (off, signed multiply, unsigned multiply, unsigned divide) is controlled by 2 bits in the WidRasterOp register:

WidRasterOp<7:6>	Operation
0	Off
1	Unsigned Divide
2	Unsigned Multiply
3	Signed Multiply

Signed divides are accomplished by remembering the signs of the dividend and divisor. The divide is performed on the absolute values of the dividend and divisor. The resulting quotient and remainder are negated if necessary.

At each step of a multiplication, the multiplicand is added to the partial product if the least significant bit of the multiplier is set. The partial product and the multiplier are then shifted to the right. Sixteen steps are needed to compute the full product.

- The MQ register is initially loaded with the multiplier by the "MQ :=" special function. This may be done before or after the multiply hardware is enabled.
- The ALU add and subtract functions (without OldCarry) perform normally if the low order bit of MQ is set, but if it is not set, the ALU passes the AMux value through unchanged. Thus, the adding or subtracting of the multiplicand is performed only if the corresponding bit of the multiplier is set.
- A Rotate(1) shift function should be used. When Shift is used as the AMux source, Shift<15> contains Result<15> Xor Overflow from the previous instruction for signed multiply or contains the Carry from the previous instruction for unsigned multiply. The remaining bits of Shift have their normal values.
- When the MultiplyStep special function is executed, the MQ register is shifted to the right and Result<0>

from the current instruction is shifted into MQ<15>. Thus, the low precision word of the product is shifted into MQ.

- The upper precision word of the product is left in an XY register.
- The low precision word of the product is read by using the ":= MQ" special function which forces MQ onto R. This value does not pass through the ALU, thus no computation is possible during this cycle. Reasonable actions are:

  1) Write the value of MQ into a register; or

  2) Send MQ to the processor shifter.

  Note that since the value does not pass through the ALU, condition codes are invalid in the cycle which follows reading MQ.

The following example performs signed multiplication of two single precision numbers yielding a double precision product. Note that the last MultiplyStep is performed with a subtract rather than an add. This is because the multiplier is interpreted as:

```
MQ<0> * 2^0 + MQ<1> * 2^1 + ...
    + MQ<14> * 2^14 - MQ<15> * 2^15
```

```
Constant(OffMultiply, 0);
Constant(OffDivide, 0);
Constant(UnSignedDivide, 100);
Constant(UnSignedMultiply, 200);
Constant(SignedMultiply, 300);

Define(Multiplier, 200);
Define(Multiplicand, 201);
Define(ProductLow, 202);
Define(ProductHigh, 203);

Rotate(1);                      ! shifter must rotate right 1
MQ := Multiplier;               ! load the multiplier
WidRasterOp := SignedMultiply;  ! set signed multiply
ProductHigh := 0,               ! partial product to shifter
```

```
        PushLoad(10#14);     ! push .+1, set S to 10#14
Shift + Multiplicand, MultiplyStep, RepeatLoop; ! 10#15 steps
Shift - Multiplicand, MultiplyStep; ! 10#16th step for sign bit
ProductHigh := Shift;        ! read upper precision product
ProductLow := MQ;            ! read lower precision product
WidRasterOp := OffMultiply;  ! turn off multiply hardware
```

During unsigned multiply all multiply steps use addition. This is because we interpret the multiplier as:

```
MQ<0> * 2^0 + MQ<1> * 2^1 + ...
    + MQ<14> * 2^14 + MQ<15> * 2^15
```

```
ProductHigh := 0,            ! partial product to shifter
            PushLoad(10#15);! push .+1, set S to 10#15
Shift + Multiplicand, MultiplyStep, RepeatLoop; ! 10#16 steps
ProductHigh := Shift;        ! read upper precision product
ProductLow := MQ;            ! read lower precision product
```

3.5.1.17. DivideStep

The multiply / divide hardware can also be used to do unsigned division by a non-restoring division algorithm. At each step, the divisor is subtracted (or added) from the partial remainder and a new bit of the dividend is shifted left into the partial remainder. Rather than restoring the partial remainder after subtracting (or adding) too much, the divisor is added (or subtracted) on the next step.

- The MQ register is initially loaded with the dividend by the "MQ :=" special function. This may be done before or after the divide hardware is enabled.
- The ALU subtract function (without OldCarry) performs normally if Result<15> of the previous instruction was not set, but if it was set, an add (without OldCarry) is performed instead. Thus the subtracting or adding of the divisor is controlled by the sign bit of the previous ALU result.
- A Rotate(10#15) shift function should be used. When Shift is used as the AMux source, Shift<0>

contains the value that MQ<15> had at the beginning of the previous instruction. Thus the dividend is shifted into the AMux from the right.

- When the DivideStep special function is executed, the MQ register is shifted to the left and the complement of Result<15> from the current instruction is shifted into MQ<0>. Thus the quotient is shifted into MQ.

- The quotient is read 16 bits at a time by the ":= MQ" special function which forces the lower precision quotient onto R. This value does not pass through the ALU, thus no computation is possible during this cycle. Reasonable actions are:

  1) Write the value of MQ into a register; or

  2) Send MQ to the processor shifter.

  Note that since the value does not pass through the ALU, condition codes are invalid in the cycle which follows reading MQ.

- The remainder is left in an XY register. Since a non-restoring algorithm is used, the loop may terminate with a negative remainder. In this case, the divisor is added back into the remainder.

The following example performs a single precision divide of a single precision dividend and a single precision divisor yielding a single precision quotient and a single precision remainder.

```
Constant(OffMultiply, 0);
Constant(OffDivide, 0);
Constant(UnSignedDivide, 100);
Constant(UnSignedMultiply, 200);
Constant(SignedMultiply, 300);

Define(Dividend, 200);
Define(Divisor, 202);
Define(Quotient, 203);
Define(QuotientSign, 204);
```

```
    Define(RemainderSign, 205);
    Define(Remainder, 206);

    Tos := 0, Push;              ! 0 for two's complementing
    RemainderSign := Dividend, RightShift(0),
                                 ! set sign of Mod
    QuotientSign := Shift Xor Divisor, ! set sign of Div
                if Geq Goto(A);
    Dividend := Tos - Dividend;  ! abs value of dividend
    Divisor;

 A: if Gtr Goto(B);              ! if divisor >= 0
    Divisor := Tos - Divisor;    ! abs value of divisor

 B: Rotate(10#15);               ! shifter must rotate left 1
    MQ := Dividend;              ! load dividend
    WidRasterOp := UnSignedDivide; ! set unsigned divide
    LoadS(10#15);                ! S := 10#15
    Remainder := 0,              ! initialize remainder
                    DivideStep;  ! get started

 C: Remainder := Shift - Divisor,
                 DivideStep, Repeat(C);! 10#16 steps
    Remainder;  ! must pass through ALU again- want to
                ! test the sign bit (not do a compare)
    WidRasterOp := OffDivide,        ! turn off divide hardware
                if Geq Goto(D);      ! if remainder >= 0
    Remainder := Remainder + Divisor;! correct remainder

 D: Quotient := MQ;              ! read quotient
    QuotientSign;
    RemainderSign, if Geq Goto(F); ! if quotient should be >= 0
    Quotient := Tos - Quotient;  ! set negative quotient
    RemainderSign;

 F: if Geq Goto(Done);          ! if remainder should be >= 0
    Remainder := Tos - Remainder; ! set negative remainder
Done:
    Pop;                         ! restore stack
```

3.5.2. Unary

Unary special functions require one argument to be specified (in the "Constant" phrase) along with the function keyword.

3.5.2.1. LeftShift

The LeftShift function performs a logical left shift of the contents of the R register zero to 16 places.

The shifter control logic always keeps the latest shift control function loaded so that a succession of words can be shifted without respecifying the function. Figure 3.5.3 demonstrates Shifter operations.

3.5.2.2. RightShift

The RightShift function performs a logical right shift of the contents of the R register zero to 16 places.

The shifter control logic always keeps the latest shift control function loaded so that a succession of words can be shifted without respecifying the function. Figure 3.5.3 demonstrates Shifter operations.

3.5.2.3. Rotate

The Rotate function performs a logical rotate (right) of the contents of the R register zero to 16 places. Positive arguments from zero to 16 rotate R rightwards; negative arguments from 0 to -16 rotate R leftwards.

The shifter control logic always keeps the latest shift control function loaded so that a succession of words can be shifted without respecifying the function. Figure 3.5.3 demonstrates Shifter operations.

3.5.2.4. IOB

The IOB special function supplies an IO bus address. The high-order bit of the argument indicates the direction of transfer (IOB[7] = 1 for write, IOB[7] = 0 for read).

3.5.2.5. CntlRasterOp

The CntlRasterOp function loads an argument into the internal control register of the RasterOp hardware.

3.5.3. Binary

Binary functions require two arguments. At this time, "Field" is the only binary function implemented by the microassembler.

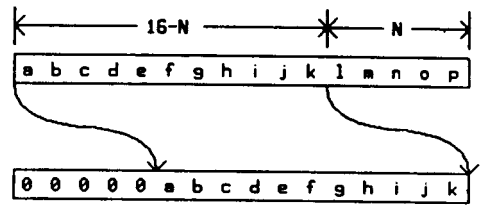
Field extracts a subfield of R. The first argument is the bit index of the low order (right-most) bit of the field; the second argument is the width of the field.

Figure 3.5.3 portrays the operation of the Shifter for the Field function.

Figure 4: Shifter Operations

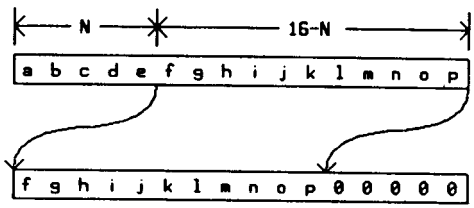
RightShift(N)

shift right N bits, filling on the left with zero bits.



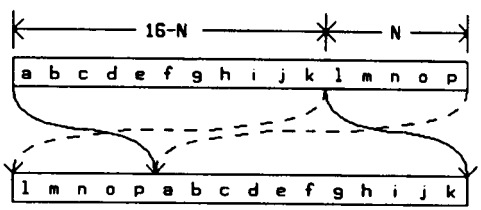
LeftShift(N)

shift left N bits, filling on the right with zero bits.



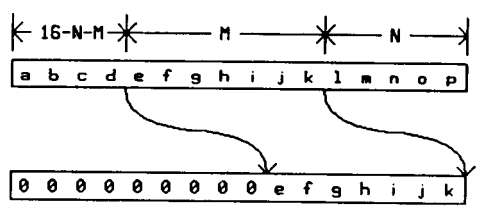
Rotate(N)

rotate right N bits.



Field(N, M)

extract a field starting at bit N, M bits wide.



4. Microassembler User's Guide

Section 4.1 describes the command level interface to the microassembler and placer, while Section 4.2 describes the microassembler directives within a source file that are recognized by the microassembler.

4.1. Microassembler Commands

Before a microprogram can be run it must be assembled with PrqMic and then placed with PrqPlace. PrqMic translates the program into binary machine language, and PrqPlace assigns physical microstore locations to those instructions which are not assigned by the microprogrammer.

This section shows how to assemble and place a microprogram. A microprogram source file name has the form <src>.micro. <src> is called the "root name".

Once a microprogram has been assembled and placed, it can be used in one of the following ways:

1. Load it into the same PERQ workstation with the ControlStore module.
2. Write it into a boot file with MakeBoot.

4.1.1. Assemble

To assemble a microprogram, type "PrqMic" or "PrqMic <src>". If the root name is not supplied, PrqMic will prompt for it as follows:

"Root file name?"

4.1.2. Place

To place a microprogram, type

"PrqPlace"

or

"PrqPlace <src> [~<lst>]".

If <src> is not supplied, PrqPlace will prompt for it. <lst> is the filename of the listing, if desired. To place without a listing, type carriage return in response to the list filename prompt (if no arguments were supplied with the command line) or type the command line with <src> supplied but no <lst>.

4.2. Microassembler Directives

There are several command lines that are directives to the microassembler and placer to perform special actions. These are indicated by the presence of a dollar sign ($) in column 1 of the command line. The entire line is considered to be a microassembler command, and thus other microinstructions cannot be present on the same line.

4.2.1. Include

This directive inserts text from a file into the microprogram as though it were present in the original source file. The syntax is

'$Include' filename

Included files cannot be nested, and so only the original source file may contain an include command.

4.2.2. Title

This directive prints a title string on the first line of every page of the assembly listing. The syntax is

$Title <TitleString>

The first Title command sets the main title, which is printed at the left of each page. Each subsequent title command sets the subtitle, which is printed at the right of each page.

4.2.3. NoList

This directive turns off listing. The syntax is

$NoList

The listing is turned off until a List command (see Section 4.2.4) is encountered. This command has an effect only if a listing has been requested when the placer is executed (see Section 4.1).

4.2.4. List

This directive resumes listing. The syntax is

$List

The listing is resumed if it was turned off by a NoList (see Section 4.2.3) command. This command has an effect only if a listing has been requested when the placer is executed (see Section 4.1).

4.2.5. PERQ1

The assembler will not recognize PERQ1a or PERQ1b features. This is the default. The syntax is

$PERQ1

4.2.6. PERQ1a

The assembler recognizes PERQ1a features only. $PERQ1 is the default. The syntax is

$PERQ1a

4.2.7. PERQ24

The assembler recognizes PERQ1a and PERQ1b features. $PERQ1 is the default. The syntax is

$PERQ24

Note that the $PERQ1, $PERQ1a, and $PERQ24 directives are mutually exclusive.

4.2.8. Base

Registers with numbers less than 100 must be declared and used with a percent sign (%) appended to their names. The percent sign indicates that the register is one to which the base register is applied. This directive is valid only in PERQ1a and PERQ24 mode and is the default. The syntax is

$Base

4.2.9. NoBase

The percent sign is not required for registers with numbers less than 100. This is used to maintain compatibility with the PERQ1. When used, this option forces the programmer to be certain that the base register contains 0. This directive is valid only in PERQ1a and PERQ24 mode. The syntax is

$NoBase

5. Additional Information

5.1. Quirks and Oddities

The following quirks are known:

1. The Z field is inverted for shift functions (assembler fixes this).
2. The Op file is inverted on NextInst (assembler Opcode does it).
3. The Z field is inverted for all jump addresses (assembler fixes it).
4. IOB functions are executed twice if an abort occurs.
5. C19 (C23) will not be valid if an abort occurs on the test.
6. C19 (C23) test is inverted sense (i.e. jump if no carry out of bit 19 or bit 23).
7. UState 15:12 (the upper BMux bits) are inverted.
8. Condition codes are not quite right after double precision adds and subtracts. See Section 2.3.11.
9. The SF bits are inverted for JMP addresses (assembler fixes this).
10. Condition codes are invalid after reading MQ or Victim.
11. RBase must be loaded with inverted data.
12. Instructions containing the ALU operation A + B + OldCarry or A - B - OldCarry will not produce

correct results if they are aborted.

13. Constant expressions may not be used as jump targets. The assembler should allow this but does not.

14. On the PERQ1a and PERQ1b, when the S register is used strictly as a 12-bit counter by RepeatLoop, Repeat, or ThreeWayBranch, its upper 2 bits are ignored. In upper banks of the controlstore the assembler will generate leap jumps for LoadS and PushLoad instructions even though current bank jumps would be acceptable.

15. Z field bits 6..0 are inverted for IOB addresses (assembler fixes this).

5.2. Memory Function Timing Information

The memory system cycles in 680 nanoseconds (exactly four microcycles). Microcycles are numbered starting at 0, and denoted 't0', 't1', 't2', and 't3'. Requests must be made in a particular cycle (depending on the type of the request). If a memory request is made in the wrong cycle, the processor will be suspended until the correct cycle, or, in some cases, the improper request will be ignored altogether. In the discussions which follow, 'fetch' or 'store' refer to a memory function which fetches or stores exactly one word. The generic terms 'fetch type' and 'store type' refer to any fetching or storing reference.

There are eight types of memory references, coded into the SF field while F = 1. They are encoded as in the following table:

SF (Octal)	Type	Description
10	Fetch4R	Fetch four words, transport in reverse order

11	Store4R	Store four words, transport in reverse order
12	Fetch4	Fetch four words
13	Store4	Store four words
14	Fetch2	Fetch two words
15	Store2	Store two words
16	Fetch	Fetch one word from memory
17	Store	Store one word into memory

The address for all memory references comes from R. For all fetch type references, the address (and the request itself) is latched at t3 and data is available from MDI or MDX at the following t2. If MDI or MDX is used during a t0 or t1 immediately following a fetch type memory reference, the processor is suspended until t2.

Any address may be used with a Fetch, and the memory word may be read during any cycle from t2 until the following t1.

The low-order bit of the address for a Fetch2 is ignored, so that a Fetch2 is always double-word aligned. After a Fetch2, the first word must be read at t2, and the second word must be read at t3.

The two low-order bits of the address for a Fetch4 or Fetch4R are ignored, so that a Fetch4 or Fetch4R is always quad-word aligned. After a Fetch4 or Fetch4R, the first word must be read at t2, the second at t3, the third at the next t0, and the last at the

next t1. Fetch4R returns word3 of the quad-word first, then word2, word1, and word0. This word reversal (from the fetch4 sequence) is primarily useful for RasterOp so that it can do left to right as well as right to left transfers.

Any address may be used with a Store. The address and Store command are given in a t2 cycle and the data to be written is supplied on R in the following t3.

The low-order bit of the address for a Store2 is ignored, so that a Store2 is always double-word aligned. The address and Store2 are given in a t3 cycle, and the data is supplied on R during the following t0 and t1.

The two low-order bits of the address for a Store4 or Store4R are ignored, so that a Store4 or Store4R is always quad-word aligned. The address and Store4 are given in a t3 cycle, and the data is supplied in the four following cycles (t0, t1, t2, t3). Store4R stores word3 of the quad-word first, then word2, word1, and word0. This word reversal (from the Store4 sequence) is primarily useful for RasterOp so that it can do left to right as well as right to left transfers.

The following are examples of each type of reference and their code:

```
Fetch:      MA := Addr, Fetch;      (t3)
            ...                     (t0)
            ...                     (t1)
            Data := MDI;            (t2)

Fetch2:     MA := Addr, Fetch2;     (t3)
            ...                     (t0)
            ...                     (t1)
            Data0 := MDI;           (t2)
            Data1 := MDI;           (t3)

Fetch4:     MA := Addr, Fetch4;     (t3)
            ...                     (t0)
            ...                     (t1)
            Data0 := MDI;           (t2)
```

```
                Data1 := MDI;           (t3)
                Data2 := MDI;           (t0)
                Data3 := MDI;           (t1)

Fetch4R:        MA := Addr, Fetch4R;    (t3)
                ...                     (t0)
                ...                     (t1)
                Data3 := MDI;           (t2)
                Data2 := MDI;           (t3)
                Data1 := MDI;           (t0)
                Data0 := MDI;           (t1)

Store:          MA := Addr, Store;      (t2)
                MDO := Data;            (t3)

Store2:         MA := Addr, Store2;     (t3)
                MDO := Data0;           (t0)
                MDO := Data1;           (t1)

Store4:         MA := Addr, Store4;     (t3)
                MDO := Data0;           (t0)
                MDO := Data1;           (t1)
                MDO := Data2;           (t2)
                MDO := Data3;           (t3)

Store4R:        MA := Addr, Store4R;    (t3)
                MDO := Data3;           (t0)
                MDO := Data2;           (t1)
                MDO := Data1;           (t2)
                MDO := Data0;           (t3)
```

The IO system can request memory cycles at any time. The memory system gives priority to the IO system so that if both the processor and the IO system make memory requests, the IO is served first while the processor is delayed. The Hold bit, if set, locks out IO requests while it is set. To be effective, Hold must be asserted in a t2. This is necessary only when doing overlapped memory references.

In some contexts, a request made in an improper cycle will be ignored as follows:

1. After a Fetch or Fetch2 (in t3), any memory reference in t0 or t1 is ignored. A Store specified in the t2 will start immediately, but all others will abort until the correct time.

2. Fetch4 and Fetch4R follow the rules for Fetch and Fetch2 with the exception that a Store4 (in the same direction--forward or reverse) can be specified in t0. This is only used for RasterOp.

3. After a Store (in t2), any memory reference in t3 or t0 is ignored. References started in t1 are aborted until the correct cycle.

4. After a Store2, Store4 or Store4R (in t3), any memory reference in t0 through t3 is ignored. Memory references started in t0 are aborted until the correct cycle.

5. To be effective, Hold must be asserted in a t2. You must be careful about aborts caused by using MDI in the wrong cycle--you may be aborted past the t2, causing the Hold to be ignored. You may not specify Hold too often--you must allow an IO reference at least once in every 3 memory cycles.

6. After a Fetch, MDI is valid from t2 through the following t1 (four full cycles). For Fetch2, Fetch4, and Fetch4R, each MDI is valid for a single microcycle.

These six constraints can be simplified into the following two simple rules. These two rules, if followed, will never cause a problem, but they may preclude certain performance optimizations permitted under rigorous application of the above six constraints. The simple guidelines are:

- Never start a memory reference after a fetch type reference until you have taken all the data.

- Never start a memory reference during the four microinstructions which follow a store type request.

Following these rules, we can construct many interesting overlapped memory requests. Note that in the following examples, Hold is always asserted in a t2. A Fetch ... Store sequence is an exception--you need not use Hold, but it doesn't hurt performance, so we assert it for consistency.

Indirect fetches:

```
MA := Addr, Fetch;              (t3)
instruction or Nop;             (t0) must be explicit
instruction or Nop;             (t1) must be explicit
MA := MDI, Fetch<n>, Hold;      (t2,t3) any type of
                                        fetch
   ...
Data := MDI;                    (t2)
   ...
```

Hold is asserted in t2 so that IO requests do not pre-empt the processor. The instruction "MA := MDI, Fetch<n>, Hold;" first tries to execute in t2, but is aborted until t3 because it contains a fetch. The MDI is still valid because MDI is valid from t2 to the following t1 after a Fetch.

```
MA := Addr, Fetch;      (t3)
instruction or Nop;     (t0)  must be explicit
instruction or Nop;     (t1)  must be explicit
instruction, Hold;      (t2)
MA := MDI, Fetch<n>;    (t3)  any type of fetch
   ...
Data := MDI;            (t2)
   ...
```

Again, Hold is asserted in t2. Note that this differs from the previous example in that the Hold and Fetch<n> are not done in the same instruction. These two examples show that for indirect fetches, the two fetches may be separated by two or three other instructions.

Indirect stores:

```
MA := Addr, Fetch;              (t3)
instruction or Nop;             (t0)  must be explicit
instruction or Nop;             (t1)  must be explicit
MA := MDI, Store, Hold;         (t2)
MDO := Data;                    (t3)
```

In this case, the MDI, the Store, and the Hold all execute in t2.

```
MA := Addr, Fetch2;             (t3)
instruction or Nop;             (t0)  must be explicit
instruction or Nop;             (t1)  must be explicit
MA := MDI, Store, Hold;         (t2)
MDO := MDI;                     (t3)
```

In this case, the first fetched word is used as an address, and the second is used as data to be stored.

```
MA := Addr, Fetch;              (t3)
instruction or Nop;             (t0)  must be explicit
instruction or Nop;             (t1)  must be explicit
MA := MDI, Store<n>, Hold;      (t2,t3) any except
                                        Store
MDO := Data;                    (t0)
          ...
```

Hold is asserted in t2 so that IO requests do not pre-empt the processor. The instruction "MA := MDI, Store<n>, Hold;" first tries to execute in t2, but is aborted until t3 because it contains a store. The MDI is still valid because MDI is valid from t2 to the following t1 after a Fetch.

```
MA := Addr, Fetch;          (t3)
instruction or Nop;         (t0)  must be explicit
instruction or Nop;         (t1)  must be explicit
instruction, Hold;          (t2)
MA := MDI, Store<n>;        (t3)  any except Store
MDO := Data;                (t0)
   ...
```

Again, Hold is asserted in t2. Note that this differs from the

previous example in that the Hold and Store<n> are not done in the same instruction. These two examples show that for indirect stores, the Fetch and the Store<n> may be separated by two or three other instructions.

Copy operations:

```
MA := Addr1, Fetch;              (t3)
instruction or Nop;              (t0)  must be explicit
instruction or Nop;              (t1)  must be explicit
MA := Addr2, Store, Hold;        (t2)
MD0 := MDI;                      (t3)
```

A word is copied from one memory location to another. Unfortunately, two or four word copies are not possible because the times when data must be read and written are different for the fetches and stores.

6. Representations of EBNF Syntax

Figure 5: Syntax Representation 1

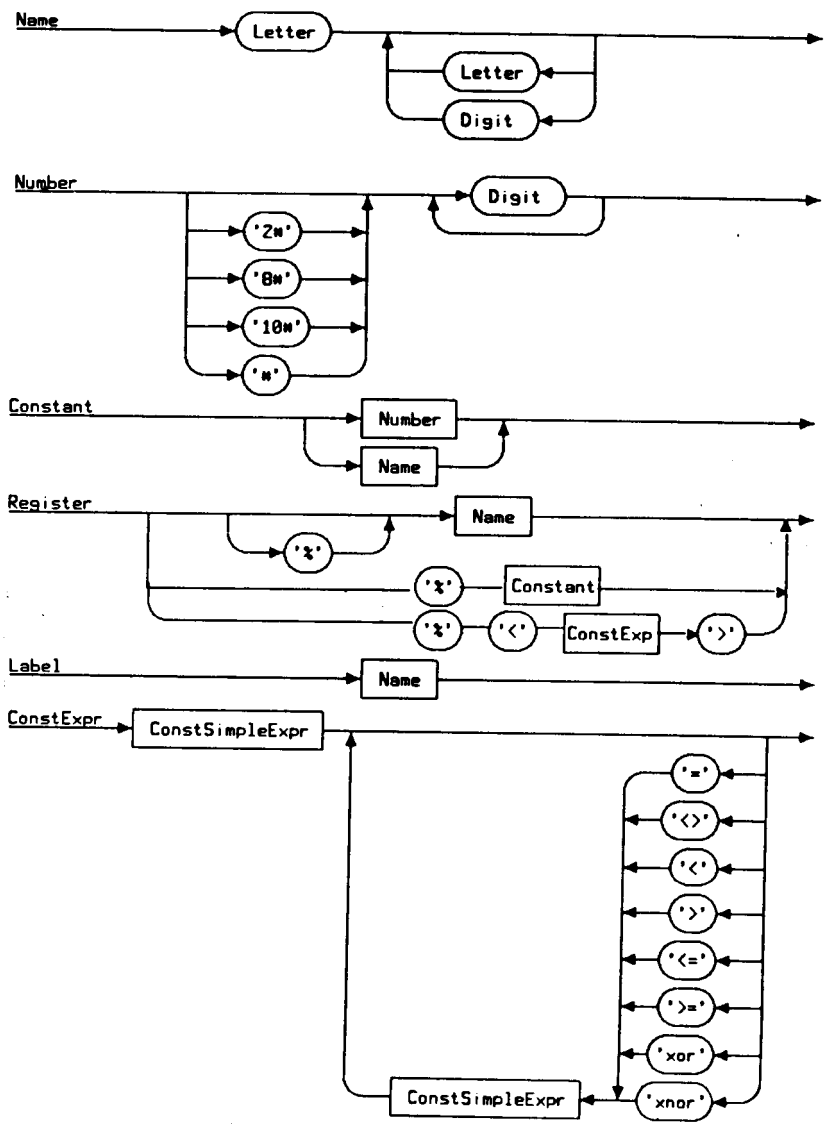


Figure 6: Syntax Representation 2

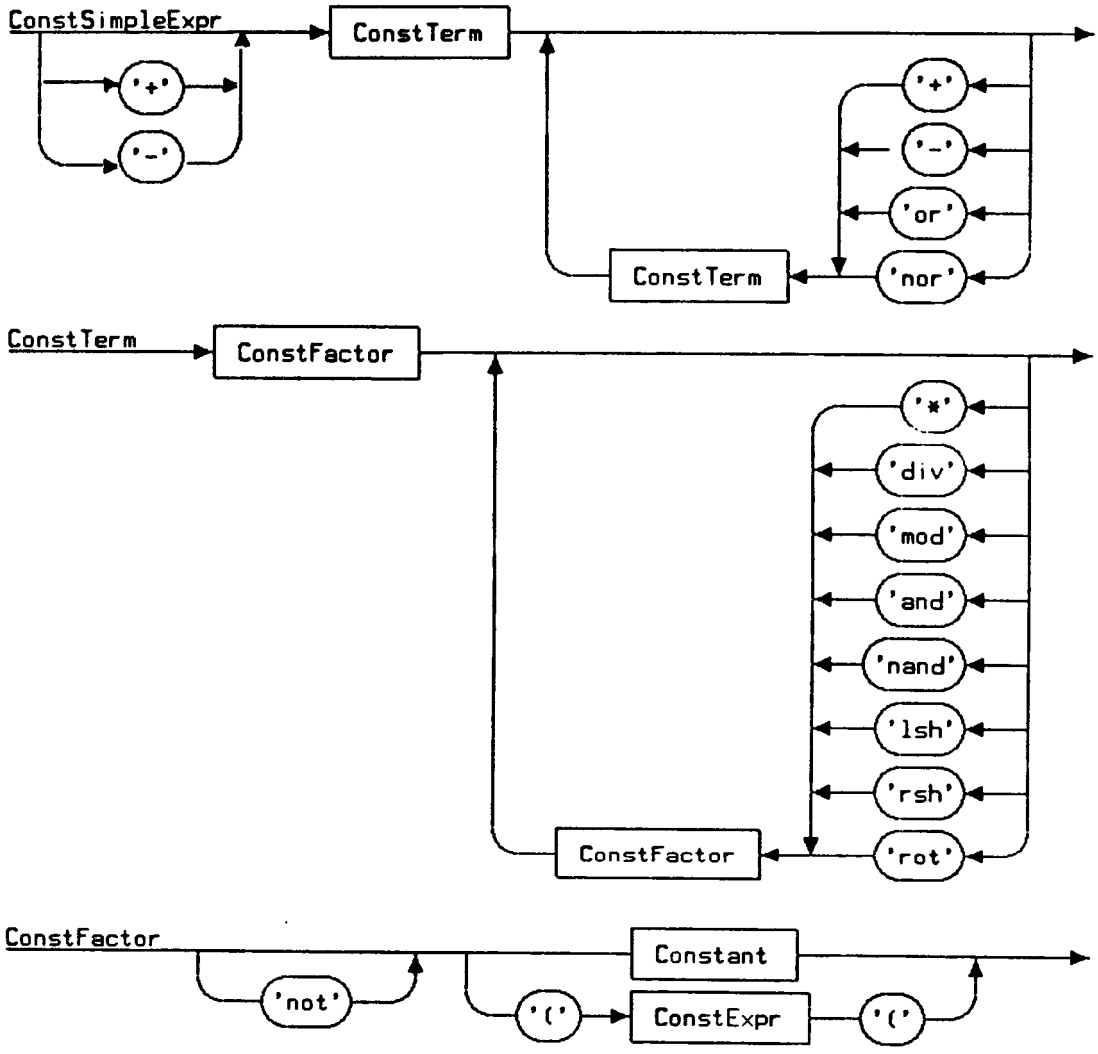


Figure 7: Syntax Representation 3

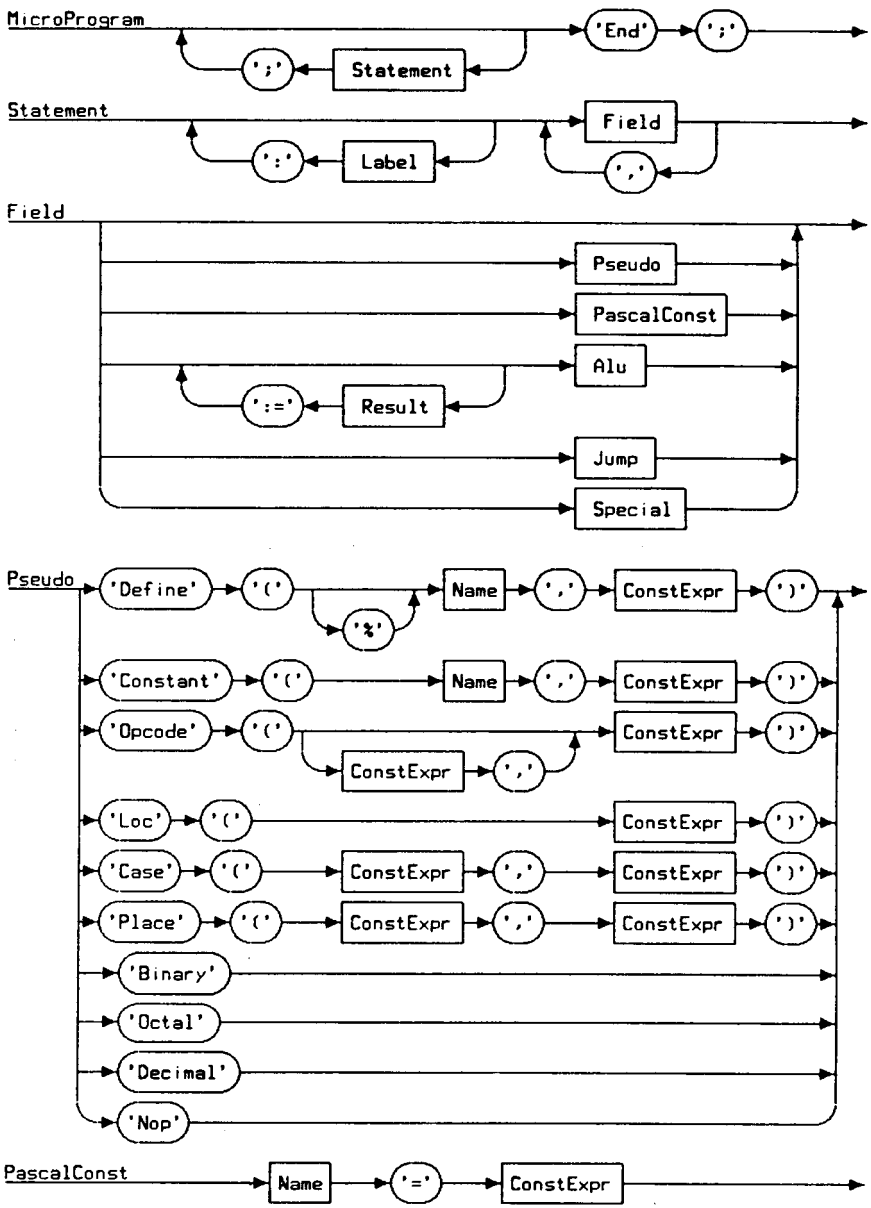


Figure 8: Syntax Representation 4

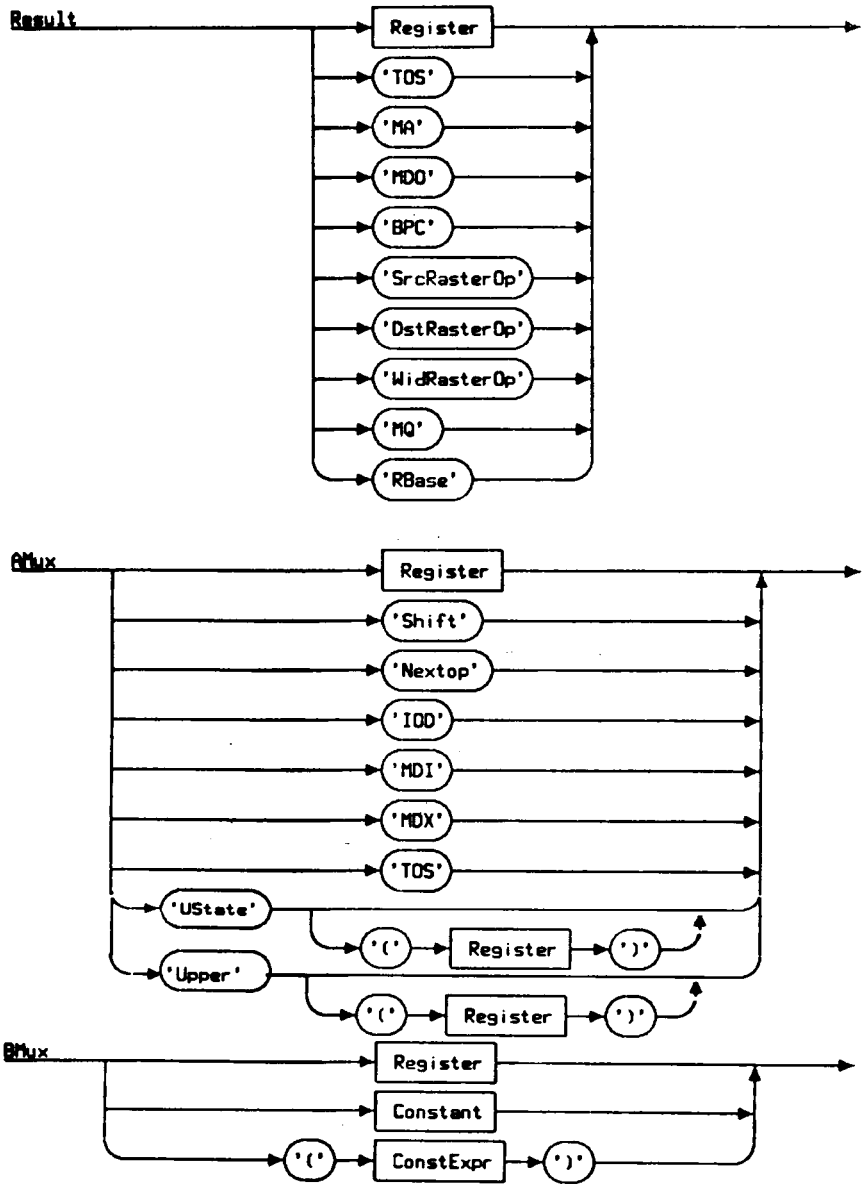


Figure 9: Syntax Representation 5

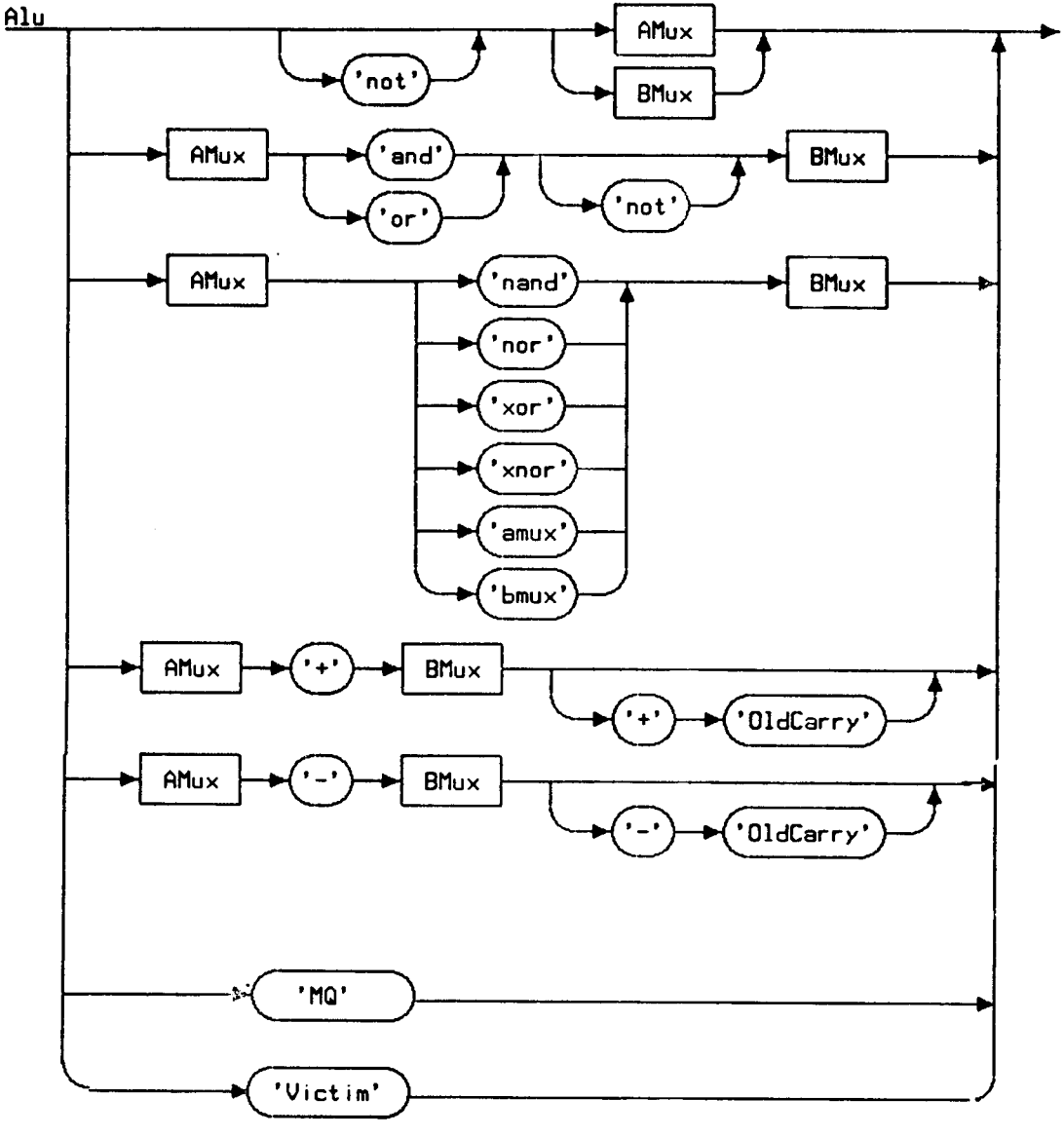
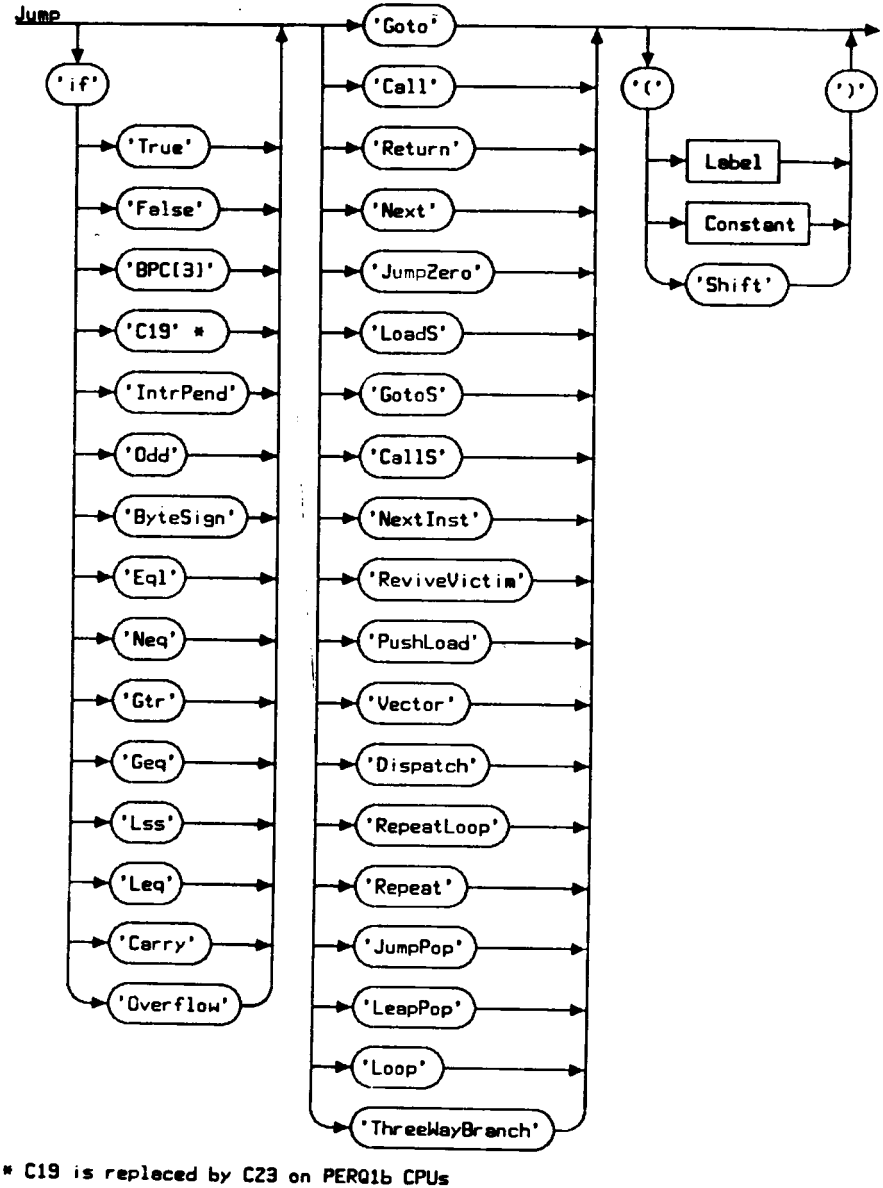


Figure 10: Syntax Representation 6



* C19 is replaced by C23 on PERQ1b CPUs

Figure 11: Syntax Representation 7

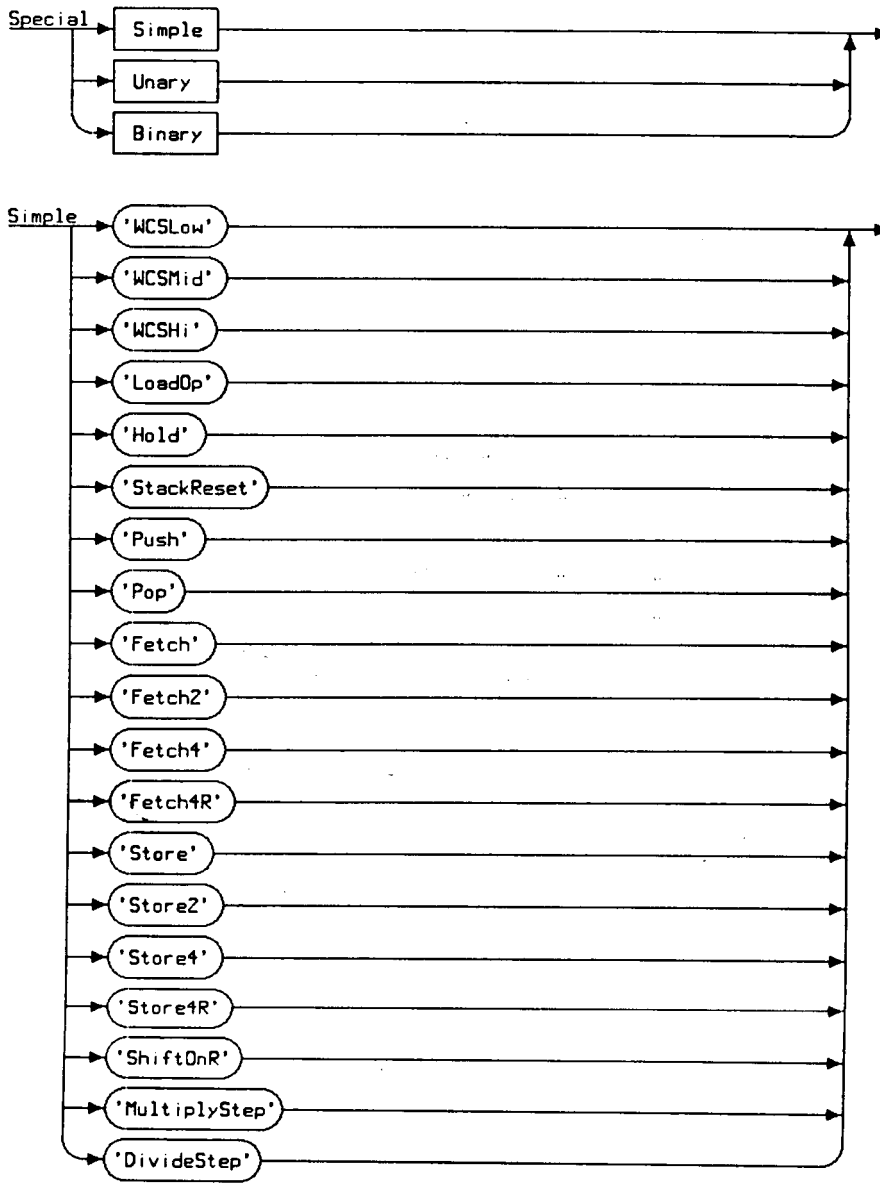
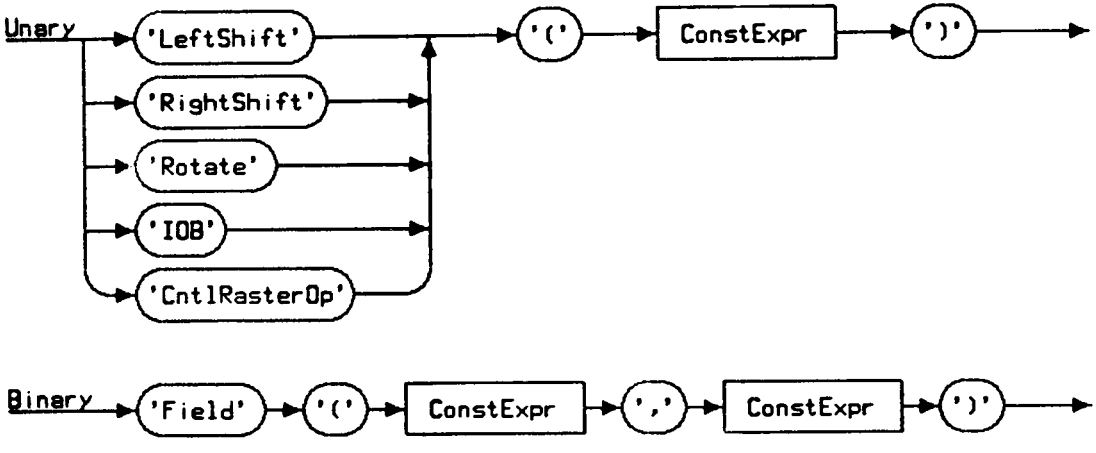


Figure 12: Syntax Representation 8



PASCAL / C
MACHINE REFERENCE

December 7, 1984

2600 Liberty Avenue
P. O. Box 2600
Pittsburgh, PA 15230
(412) 355-0900

Table of Contents Page

1. Q-Machine Architecture

1.1. Definitions

Segment - The Pascal system is based on segments. Each separately compiled module is made into a segment. Segments have a maximum size of 64K words. When a segment is created, a portion of process virtual address space is allocated. The high order word of this virtual address is interpreted as the segment number.

MStack - Memory Stack. A data segment which contains the user run-time stack.

EStack - Expression Stack. A 16 level expression evaluation stack (internal to the PERQ workstation processor).

MTOS - Top of MStack. MTOS refers to the virtual address of the top of the memory stack. (MTOS) denotes the item on the top of the MStack.

ETOS - Top of EStack. (ETOS) denotes the item on the top of the EStack.

Activation Record - Stack segment fragment for a single routine containing local variables, parameters, function result, temporaries (anonymous variables), other housekeeping values (Activation Control Block - defined below), and a copy of the EStack at the time the activation record is created.

CB - Code Base (register). High order 16 bits of the virtual

address of the current code segment.

SB - Stack Base (register). High order 16 bits of the virtual address of the current stack segment.

PC - Program Counter (register). Low order 16 bits of the virtual address of the current instruction. The virtual address CB,,PC is the full virtual address of the current instruction.

GDB - Global Data Block. A GDB contains the global variables for a particular module. GDBs always begin on a double-word boundary.

ISN - Internal Segment Number (compiler-generated).

SSN - System Segment Number (system-generated). High order 16 bits of the virtual address of a segment. Note that System Segment 0 is reserved and may never be used.

LL - Lexical Level. Note: the Lexical Level of the main body of a process is always 0.

RN - Routine Number (register). RN contains the ordinal number of the current routine. Note: RN must lie in the range 0 to 255.

CS - Code Segment (register). See Code Base.

SS - Stack Segment (register). See Stack Base.

PS - Parameter Size. PS is the number of words in an activation record that are used for parameters.

RPS - Result + Parameter Size. This is the number of words in an activation record that are used for function result and

parameters.

LTS - Local + Temporary Size. LTS is the number of words in an activation record that are used for locals and temporaries (anonymous variables). (Note: the LTS of a main program body is always forced to 0.)

AP - Activation Pointer (register). AP contains the low order 16 bits of the virtual address of the current activation record. SB,,AP gives the full virtual address.

DL - Dynamic Link. This is the AP of the caller, represented as an offset from SB.

SL - Static Link. This is the AP of the surrounding routine, represented as an offset from SB.

TP - Top Pointer (register). TP contains the low order 16 bits of the virtual address of the top of the run-time MStack. SB,,TP is the full virtual address.

TL - Top Link. TP of the caller, represented as an offset from SB.

GP - Global Pointer (register). Low order 16 bits of the virtual address of the GDB for the current code segment. SB,,GP gives the full virtual address.

GL - Global Link. GP of the caller, represented as an offset from SB.

LP - Local Pointer (register). Low order 16 bits of the virtual address of the current activation record. When the LP is stored in an Activation Control Block (ACB), it is represented as an offset from SB. Unlike other values in the

ACB, the LP value is the current value of the Local Pointer, not some previous value.

XGP - eXternal Global Pointer. Pointer to another code segment's GDB, represented as an offset from SB.

XST - eXternal Segment Table. For a given program module, the XST translates ISNs to SSNs and XGPs.

RS - Return Segment. RS is the CS of the caller.

RA - Return Address. PC of the caller, represented as an offset from CB.

RR - Return Routine. RN of the caller.

RD - Routine Dictionary. Each code segment contains a routine dictionary which is indexed by RN. For each routine, the routine dictionary gives the lexical level (LL), entry address, exit address, parameter size (PS), result + parameter size (RPS), and local + temporary size (LTS).

ACB - Activation Control Block. The ACB contains housekeeping values in the activation record. It contains the SL, LP, DL, GL, RS, RA, RR and EP. In the ACB, the DL, GL, RS, RA, and RR are the AP, GP, CS, PC, and RN of the caller, respectively. The SL is the AP of the routine that surrounds the current one. The LP in the ACB is the current local pointer.

EEB - Exception Enable Block - Each EEB enables a single exception by associating an exception with a handler. A (possibly empty) list of EEBs is associated with each activation record in the stack.

Enabling an Exception - Associating a certain exception handler with a certain exception.

EP - Exception Pointer. The address (as an offset from SB) of a list of nodes that describe which exceptions are enabled in a certain routine.

ER - Exception Routine Number. The routine number of an exception.

ES - Exception Segment Number. The segment number of an exception.

Exception - An error or unusual occurrence in the execution of a routine or program.

Exception Handler - A procedure to be executed when a certain exception is raised.

HR - Handler Routine Number. The routine number of an exception handler.

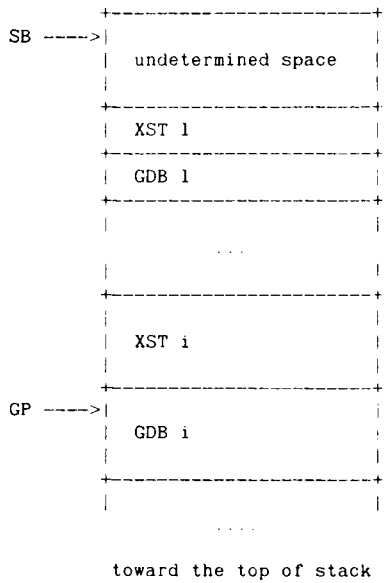
NE - Next Exception. The address (as an offset from SB) of the next in a list of nodes that describe which exceptions are enabled in a certain routine.

Raising an Exception - Asserting a certain exception.

1.2. Language Memory Organization

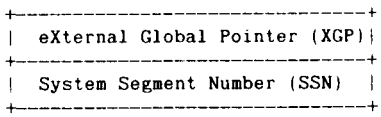
1.2.1. Global data

At the global level, there is a Global Data Block (GDB) and an eXternal Segment Table (XST) associated with each code segment in a process. For a particular program module, the GDB contains the global variables, and the XST translates internal (compiler-generated) segment numbers (ISNs) to actual system segment numbers (SSNs) and eXternal Global Pointers (XGPs). To simplify the system, a single pointer is devoted to reference both the current GDB and XST. This Global Pointer (GP) points to the lowest address in the GDB.



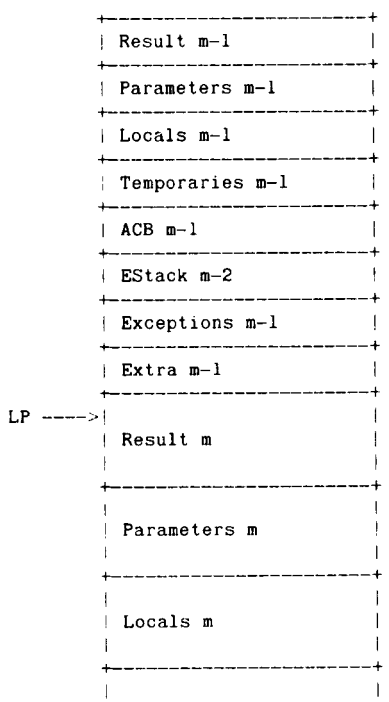
The XST for each segment is indexed by the internal segment numbers (ISNs). The entry is at GP - 2*ISN (Note: There is no entry for ISN 0; ISN 0 always refers to the current segment). Each entry contains the offset from stack base (SB) of an external data block (XGP) and the actual system segment number (SSN) of the external segment. The XGP values are set by the linker, and

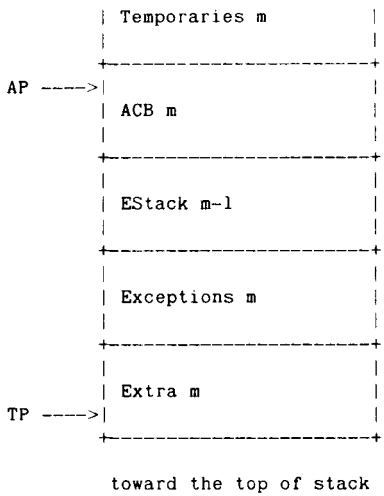
the SSN values are set by the loader.



1.2.2. Local data

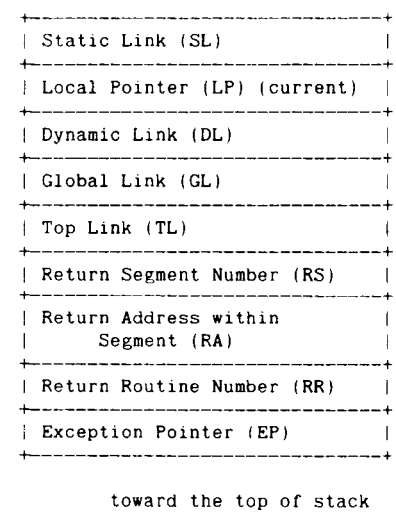
At the local level, there is an activation record consisting of local variables, function result, parameters, temporaries (anonymous variables), the Activation Control Block (ACB), the previous EStack, exception enable blocks, and extra values that the routine may push and pop from the run-time stack. Three pointers are used to access and keep track of this information: the top-of-stack pointer (TP), the current-activation pointer (AP), and the local-variables pointer (LP).





The function result, parameters, locals and temporaries are located by an offset from LP.

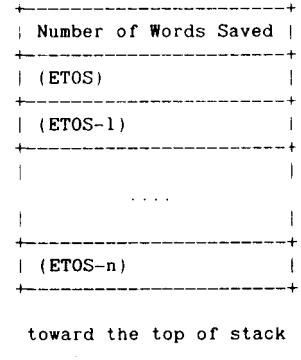
Each ACB has the following form:



The values in the ACB are the AP of the surrounding routine

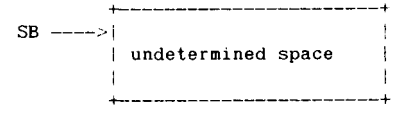
(SL), the current (not previous) LP, the AP of the caller (DL), the GP of the caller (GL), the TP of the caller (TL), the SSN of the caller (RS), the program counter (PC) of the caller (RA), the RN of the caller (RR) and a pointer to the current exception enable records (EP). Note: When previous pointer values are saved in the ACB they are called links: SL, DL, GL, TL. Because the current (not previous) LP and EP are stored in the ACB, they are called pointers, not links. The static link is not used for main programs and top-level routines. It is zero for these routines and is therefore a means of detecting top-level routines. The dynamic link is zero for the first routine on the stack (the one at the base of the stack). This is used to detect the end of the stack during stack searches.

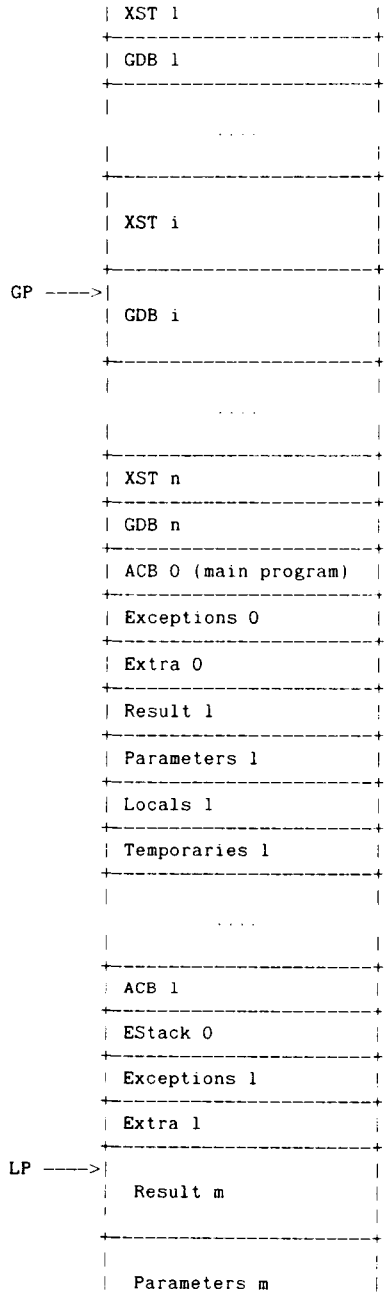
The EStack image immediately follows the ACB and looks like this:

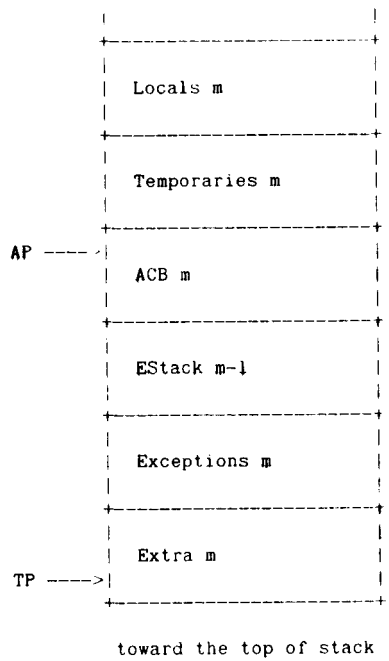


1.2.3. Run-time stack organization

The following is an outline of the stack for a process of n segments, executing the mth routine call, which is in the ith segment:




XST 1
GDB 1
XST i
GP
GDB i
XST n
GDB n
ACB 0 (main program)
Exceptions 0
Extra 0
Result 1
Parameters 1
Locals 1
Temporaries 1
ACB 1
EStack 0
Exceptions 1
Extra 1
LP
Result m
Parameters m



1.2.4. Code Segment Organization

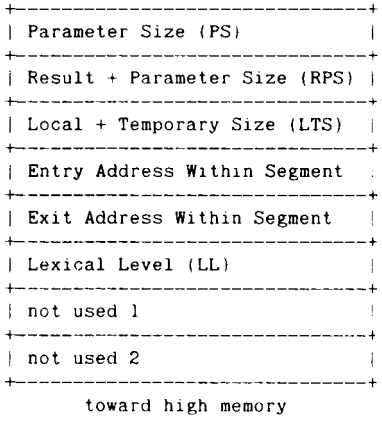
A code segment contains the code for all routines in a segment and a routine dictionary that contains vital information about each of these routines.

The first word of every code segment is the offset from the base of the segment to the first word of the routine dictionary. The second word contains the number of routines that are defined in the segment. These two words are followed by the actual code that comprise the routines. Finally, the code is followed by the routine dictionary.

The code is padded with 0 to 3 words of zero (by the compiler) so that the routine dictionary is aligned on a quad-word boundary. This is possible since the compiler knows that the base of the segment is also aligned on a quad-word boundary. It should also

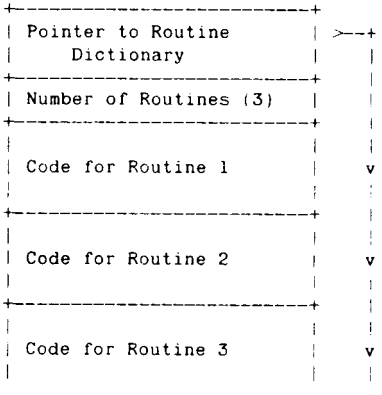
be noted that each entry in the dictionary is exactly 2 quad-words long (8 words). The routine dictionary is indexed by (Base Address of Dictionary)+8*RN.

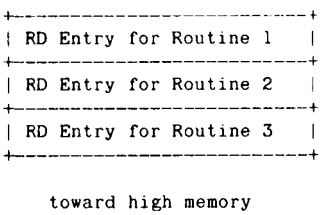
Each entry has the following form:



The Entry and Exit Addresses are the offsets from code base (CB) to the beginning of the routine and the beginning of the "terminate code" of the routine.

The following is a sample of a code segment containing 3 routines:





1.3. Error Handling and Fault Conditions

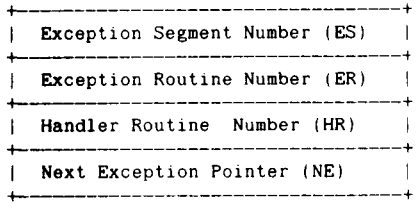
Error processing is done through an exception handling mechanism. The syntax of exception and handler is described in the document "PERQ Pascal Extensions" in the *Accent Languages Manual*. The QCodes used to enable and raise exceptions are described this document in Section 4.4, Routine Calls and Returns.

An exception is declared in a way similar to a procedure declaration. Therefore it is convenient to assign a routine number at the point of definition of an exception. There is a corresponding entry in the routine dictionary to describe this exception. This entry describes a procedure with the correct number of parameters along with no locals or temporaries. The system segment number of the module, containing the exception, and its routine number uniquely identify the exception.

An exception handler is simply a procedure--handlers may not return a value. An exception handler is enabled by declaring it inside a routine. This outer routine is called the enabler of the handler. The code segment numbers and global pointers of the enabler and handler are the same. The static link of the handler is the same as the activation pointer of the enabler. Thus an exception handler is uniquely identified by the ACB of the enabler and the routine number of the handler.

An exception enable record consists of the definition of the exception to be handled (ER and ES), the routine number of the

handler (HR), and a link to the next exception enable record (NE). ER and ES are negative for a handler of all exceptions.



When a routine is called, the exception pointer (EP) within the new ACB is set to zero to indicate that there are no exception handlers. If exception handlers are declared within the routine, the compiler generates appropriate QCodes to add those handlers to the routine's exception list. When a routine is exited, the exception records are popped from the run-time stack along with the rest of the activation record.

When an exception is raised, the QCode interpreter microcode calls the procedure RaiseP in the module Except. This routine searches back through the run-time stack to find the most recent routine which contains a handler for that exception. Once such a candidate is found, the stack is searched again to determine if that handler is already active. If it is, the search for a candidate continues. This implementation ensures that while a particular instance of a handler is active, it will not be activated again. A recursive routine may contain a handler, and there may be several instances of the same handler. In this case, each handler is activated separately. Note that the handlers for NormMsg and EmergMsg (defined in EXCEPT) are reinvoked even if the handler is active.

Determining if a handler is active by searching the stack is not the

best method. If we assume that the depths of handler activation records are related to the number of active handlers, then the total stack search time for raising an exception is related to the square of the number of active handlers of that exception. This should rarely be burdensome because it is not expected that there will be more than one or two active handlers for a given exception. Recursive routines that pass exceptions up the call stack are pathological cases.

When an unused enable for the exception is found, the associated handler is called. The handler has the option of exiting to the routine which enabled the exception (via a Goto) or of returning to the point where the exception was raised (by falling off the end of the procedure).

If no handler is found, the default handler is called. The search order for a certain routine's exception list is the reverse of the order in which they were enabled. If a handler of all exceptions is declared within a certain routine, the compiler enables it first.

A handler of all exceptions is called in a special way. When it is called, the parameters that were passed when the exception was raised have already been pushed onto the stack. A new activation record for the handler is built above those parameters. Such a handler for all exceptions must have four words of parameters: the segment and routine numbers of the exception that was raised, a pointer to the first word of the original parameters, and a pointer to the first word after the original parameters. The two pointers are represented as integer offsets from the base of the stack.

2. Instruction Format

Instructions on the Q-machine are one byte long followed by zero to four parameters. Parameters are either a signed byte (B : range -128 to 127), an unsigned byte (UB : range 0 to 255) or a word (W). Words need not be word aligned (unless specified). The low byte is first in the instruction byte stream.

Any exceptions to these formats are noted with the instructions where they occur.

3. Pointers

There are five different types of pointers, defined as follows:

Word Pointer: A 32-bit virtual address.

Byte Pointer: A 32-bit virtual address to the base of the array in (ETOS-1), (ETOS-2) and a byte offset into the array in (ETOS).

String Pointer: Same as a byte pointer.

Packed Field Pointer: A 32-bit virtual address to the base of the word the field is in, in (ETOS-1), (ETOS-2). A one word field descriptor in (ETOS).

Field Descriptor:

Bits 0-3: The field width (in bits) minus 1

Bits 4-7: The rightmost bit of the field.

Pascal Pointer: Obtained by declaring a variable as a pointer to another data type (i.e., var I: ^Integer;). (TOS-1) is the system segment number that contains the datum. (TOS) is the offset from the segment base to the datum. This data structure is a full 32 bit virtual address - the same as a word pointer.

Implementation Note: Stacks grow from low addresses to high addresses (i.e., if the address of TOS is 10 then the address of TOS-1 is 9 -- not 11).

4. QCode Descriptions

This section provides a detailed description for each QCode. The QCode descriptions appear categorically. The following lists the QCodes and equates each with its respective QCode number. Thus, if you only know the QCode number, you can use the list to equate the number to the named QCode.

Note that some QCodes are Extended Opcodes. The Extended Opcodes provide for additional instructions beyond the decoding limitation of a single byte opcode field. All extended opcodes are two-byte instructions where the first byte is EXOP and the second byte defines the function of the instruction.

```
QCode OpCode Definitions:

LDC0      = 0;
LDC1      = 1;
LDC2      = 2;
LDC3      = 3;
LDC4      = 4;
LDC5      = 5;
LDC6      = 6;
LDC7      = 7;
LDC8      = 8;
LDC9      = 9;
LDC10     = 10;
LDC11     = 11;
LDC12     = 12;
LDC13     = 13;
LDC14     = 14;
LDC15     = 15;
LDCM0     = 16;
LDCB      = 17;
LDCW      = 18;
LDCN      = 19;
LDLC1     = 20;
LDLCM1    = 21;
LDLCB     = 22;
LDDC      = 23;
LSA       = 24;
QAND      = 25;
QOR       = 26;
QNOT      = 27;
```

```
QXOR      = 28;
EQUI      = 29;
NEQI      = 30;
LEQI      = 31;
LESI      = 32;
GEQI      = 33;
GTRI      = 34;
ABI       = 35;
ADI       = 36;
NGI       = 37;
SBI       = 38;
MPI       = 39;
DVI       = 40;
MODI      = 41;
CHK       = 42;
ROTSHI    = 43;
nCVTLI    = 44;
nCVTIL    = 45;
nADL      = 46;
nSBL      = 47;
nEQULONG  = 48;
nNEQLONG  = 49;
nLEQLONG  = 50;
nLESLONG  = 51;
nGEQLONG  = 52;
nGTRLONG  = 53;
EQNIL     = 54;
EXCH      = 55;
EXCH2     = 56;
REPL      = 57;
REPL2     = 58;
MMS       = 59;
MMS2      = 60;
MES       = 61;
MES2      = 62;
EXOP      = 63;
LLLB      = 64;
LLLW      = 65;
SLLB      = 66;
SLLW      = 67;
LILB      = 68;
LILW      = 69;
SILB      = 70;
SILW      = 71;
LOLB      = 72;
LOLW      = 73;
SOLB      = 74;
SOLW      = 75;
LGLB      = 76;
LGLW      = 77;
SGLB      = 78;
SGLW      = 79;
LDLB      = 80;
LDLW      = 81;
LDLO      = 82;
LDL1      = 83;
```

```
LDL2        = 84;
LDL3        = 85;
LDL4        = 86;
LDL5        = 87;
LDL6        = 88;
LDL7        = 89;
LDL8        = 90;
LDL9        = 91;
LDL10       = 92;
LDL11       = 93;
LDL12       = 94;
LDL13       = 95;
LDL14       = 96;
LDL15       = 97;
LLAB        = 98;
LLAW        = 99;
STLB        = 100;
STLW        = 101;
STL0        = 102;
STL1        = 103;
STL2        = 104;
STL3        = 105;
STL4        = 106;
STL5        = 107;
STL6        = 108;
STL7        = 109;
LDIB        = 110;
LDIW        = 111;
LIAB        = 112;
LIAW        = 113;
STIB        = 114;
STIW        = 115;
LD0B        = 116;
LD0W        = 117;
LOAB        = 118;
LOAW        = 119;
ST0B        = 120;
ST0W        = 121;
LDGB        = 122;
LDGW        = 123;
LGAB        = 124;
LGAW        = 125;
STGB        = 126;
STGW        = 127;
LLLLB       = 128;
SLR0        = 129;
SLR1        = 130;
SLR2        = 131;
SLR3        = 132;
LLR0        = 133;
LLR1        = 134;
LLR2        = 135;
LLR3        = 136;
SIR0        = 137;
SIR1        = 138;
SIR2        = 139;
```

```
SIR3      = 140;
LIR0      = 141;
LIR1      = 142;
LIR2      = 143;
LIR3      = 144;
IXAB      = 145;
IXAW      = 146;
IXA1      = 147;
IXA2      = 148;
IXA3      = 149;
IXA4      = 150;
INCB      = 151;
INCW      = 152;
INDB      = 153;
INDW      = 154;
IND0      = 155;
IND1      = 156;
IND2      = 157;
IND3      = 158;
IND4      = 159;
IND5      = 160;
IND6      = 161;
IND7      = 162;
STIND     = 163;
LBLB      = 164;
LBL0      = 165;
STDW      = 166;
LDLIND    = 167;
LDB       = 168;
LDBLong   = 169;
LDBIND    = 170;
STB       = 171;
STBLong   = 172;
STBIND    = 173;
LDCH      = 174;
STCH      = 175;
IXP       = 176;
IXPLong   = 177;
LDP       = 178;
STPF      = 179;
SAS       = 180;
LXAB      = 181;
LXAW      = 182;
LXA2      = 183;
LXA3      = 184;
LXA4      = 185;
LBAR0B    = 186;
LBAR0W    = 187;
LBAR1B    = 188;
LBAR1W    = 189;
LBAR2B    = 190;
LBAR2W    = 191;
LBAR3B    = 192;
LBAR3W    = 193;
LBIR0B    = 194;
LBIR1B    = 195;
```

```
LBIR2B    = 196;
LBIR3B    = 197;
LBLR0B    = 198;
LBLR0W    = 199;
LBLR1B    = 200;
LBLR1W    = 201;
LBLR2B    = 202;
LBLR2W    = 203;
LBLR3B    = 204;
LBLR3W    = 205;
IXAR0B    = 206;
IXAR0W    = 207;
IXAR1B    = 208;
IXAR1W    = 209;
IXAR2B    = 210;
IXAR2W    = 211;
IXAR3B    = 212;
IXAR3W    = 213;
LXAR0B    = 214;
LXAR0W    = 215;
LXAR1B    = 216;
LXAR1W    = 217;
LXAR2B    = 218;
LXAR2W    = 219;
LXAR3B    = 220;
LXAR3W    = 221;
XJP       = 222;
JMPB      = 223;
JMPW      = 224;
JFB       = 225;
JFW       = 226;
JTB       = 227;
JTW       = 228;
JEQB      = 229;
JEQW      = 230;
JNEB      = 231;
JNEW      = 232;
LDRET1    = 233;
LDRET2    = 234;
CCALL     = 235;
CENTER    = 236;
CRET      = 237;
CALLXB    = 238;
CALLL     = 239;
CALLV     = 240;
RET       = 241;
EXITT     = 242;
EXGO      = 243;
ATPB      = 244;
ATPW      = 245;
LDAP      = 246;
LDTP      = 247;
EVENT     = 248;
WCS       = 249;
JCS       = 250;
GOTOOVL   = 251;
```

```
KOPS        = 252;
NOOP        = 253;
BREAK       = 254;
REFILLOP    = 255;


{
{ Spice Kernel Operations
{       Second byte of KOPS QCode

KSVRETURN           = 0;
KCURPROCESS         = 1;
KHASH               = 2;
KSETSOFT            = 3;
KCLOCK              = 4;
KGETCB              = 5;
KSETVMTABLES        = 6;
KSVCALL             = 9;
KADDTOQUEUE         = 10;
KREMOVEFROMQUEUE    = 11;
KWAKEUP             = 12;
KSLEEP              = 13;
KUNBLOCK            = 14;
KBLOCK              = 15;
KLOCKSVC            = 16;
KUNLOCKSVC          = 17;
KSEARCH             = 18;
KVPENTER            = 19;
KVPREMOVE           = 20;
KSEARCHADDR         = 21;
KSEARCHPV           = 22;
KCOPYPAGE           = 23;


{
{ Extended Opcodes:
{       Second byte of the EXOP QCode.
{

0 thru 2 are undefined
NGL             = 3;
4 is undefined
MPL             = 5;
DVL             = 6;
MODL            = 7;
ABL             = 8;
9 thru 14 are undefined
LBITS           = 15;
LBNOT           = 16;
LBAND           = 17;
LBOR            = 18;
LBXOR           = 19;
exCHKLong       = 20;
exTNCRI         = 21;
exFLTIR         = 22;
exADR           = 23;
```

```
exNGR          = 24;
exSBR          = 25;
exMPR          = 26;
exDVR          = 27;
exRNDRI        = 28;
exABR          = 29;
exEQUReal      = 30;
exNEQReal      = 31;
exLEQReal      = 32;
exLESReal      = 33;
exGEQReal      = 34;
exGTRReal      = 35;
exTNCQL        = 36;
exFLTLQ        = 37;
exADDQ         = 38;
exNEGQ         = 39;
exSUBQ         = 40;
exMULQ         = 41;
exDIVQ         = 42;
exRNDQL        = 43;
exABSQ         = 44;
exEQUQ         = 45;
exNEQQ         = 46;
exLEQQ         = 47;
exLESQ         = 48;
exGEQQ         = 49;
exGTRQ         = 50;
exTNCRL        = 51;
exFLTLR        = 52;
exCVQR         = 53;
exCVRQ         = 54;
exRNDRL        = 55;
exLDQ          = 56;
exSTQ          = 57;
exEXCHQ        = 58;
exPERMD        = 59;
exJLK          = 60;
exJMS          = 61;
exEQUStr       = 62;
exNEQStr       = 63;
exLEQStr       = 64;
exLESStr       = 65;
exGEQStr       = 66;
exGTRStr       = 67;
exEQUByt       = 68;
exNEQByt       = 69;
exLEQByt       = 70;
exLESByt       = 71;
exGEQByt       = 72;
exGTRByt       = 73;
exEQUPowr      = 74;
exNEQPowr      = 75;
exLEQPowr      = 76;
exSGS          = 77;
exGEQPowr      = 78;
exSRS          = 79;
```

```
exINN          = 80;
exUNI          = 81;
exQINT         = 82;
exDIF          = 83;
exADJ          = 84;
exEQUWord      = 85;
exNEQWord      = 86;
exRASTOP       = 87;
exSTRROP       = 88;
exLINE         = 89;
exMVBB         = 90;
exMVBW         = 91;
exMOVB         = 92;
exMOVW         = 93;
exSTMW         = 94;
exLDMC         = 95;
exLDMW         = 96;
exSETEXC       = 97;
exENABLE       = 98;
exQRAISE       = 99;
exZEROMEM      = 100;
exINCDDS       = 101;
exSTRTIO       = 102;
exLVRD         = 103;
exLBIR0W       = 104;
exLBIR1W       = 105;
exLBIR2W       = 106;
exLBIR3W       = 107;
exLBQR0B       = 108;
exLBQR0W       = 109;
exLBQR1B       = 110;
exLBQR1W       = 111;
exLBQR2B       = 112;
exLBQR2W       = 113;
exLBQR3B       = 114;
exLBQR3W       = 115;
exIXAR01       = 116;
exIXAR02       = 117;
exIXAR03       = 118;
exIXAR04       = 119;
exIXAR11       = 120;
exIXAR12       = 121;
exIXAR13       = 122;
exIXAR14       = 123;
exIXAR21       = 124;
exIXAR22       = 125;
exIXAR23       = 126;
exIXAR24       = 127;
exIXAR31       = 128;
exIXAR32       = 129;
exIXAR33       = 130;
exIXAR34       = 131;
exLXAR01       = 132;
exLXAR02       = 133;
exLXAR03       = 134;
exLXAR04       = 135;
```

```
exLXAR11          136;
exLXAR12          137;
exLXAR13        = 138;
exLXAR14        = 139;
exLXAR21        = 140;
exLXAR22        = 141;
exLXAR23        = 142;
exLXAR24        = 143;
exLXAR31        = 144;
exLXAR32        = 145;
exLXAR33        = 146;
exLXAR34        = 147;
exPop1          = 148;
exPop2          = 149;
exNDSTLB        = 150;
exNDSTLW        = 151;
exNDSLLB        = 152;
exNDSLLW        = 153;
exNDSTGB        = 154;
exNDSTGW        = 155;
exNDSGLB        = 156;
exNDSGLW        = 157;
exNDSIR0        = 158;
exNDSIR1        = 159;
exNDSIR2        = 160;
exNDSIR3        = 161;
exNDSLR0        = 162;
exNDSLR1        = 163;
exNDSLR2        = 164;
exNDSLR3        = 165;
exMathMODI      = 166;
exMathMODL      = 167;
168 thru 238 are undefined
SvRet1          = 239;
240 is undefined
SvRet2          = 241;
242 is undefined
CVTCL           = 243;
244 is undefined
CVTCI           = 245;
LOPSHI          = 246;
DECREG3         = 247;
DECREG2         = 248;
DECREG1         = 249;
DECREG0         = 250;
INCREG3         = 251;
INCREG2         = 252;
INCREG1         = 253;
INCREG0         = 254;
EXREFILLOP      = 255;
```

4.1. Variable Fetching, Indexing, Storing and Transferring

4.1.1. Loads and stores of one word

4.1.1.1. Constant one word loads

LDC0..15 0-15

Load Word Constant. Pushes the value (0..15), with high byte zero, onto the EStack.

LDCMO 16

Load Constant -1. Pushes the value -1 onto the EStack.

LDCB B 17

Load Constant Byte. Pushes the next byte on the EStack, sign extended to a word.

LDCW W 18

Load Constant Word. Pushes the next word on the EStack.

4.1.1.2. Local one word loads and stores

LDL0..15 82-97

Short Load Local Word. LDLx fetches the word with offset x in the current activation record and pushes it onto the EStack.

LDLB UB 80

Load Local Word / Byte Offset. Fetches the word with offset UB in the current activation record and pushes it on the EStack.

LDLW W 81

Load Local Word / Word Offset. Fetches the word with offset W in the current activation record and pushes it on the EStack.

LLAB UB 98

Load Local Address / Byte Offset. Pushes a word pointer to the word with offset UB in the current activation record on EStack.

LLAW W 99

Load Local Address / Word Offset. Pushes a word pointer to the word with offset W in the current activation record on EStack.

STL0..7 102-109

Short Store Local Word. Store (ETOS) into word with offset x in the current activation record.

STLB UB 100

Store Local Word / Byte Offset. Store (ETOS) into word with offset UB in the current activation record.

STLW W 101

Store Local Word / Word Offset. Store (ETOS) into word with offset W in the current activation record.

Implementation Note: The address of the first local (offset 0) is contained in the Local Pointer register (LP). The address of the Nth local is computed as (LP) + N.

exNDSTLB UB 150

Non-Destructive Store Local Word / Byte Offset. Identical to STLB except that the EStack is NOT popped.

Note that this is an extended operation.

exNDSTLW W 151

Non-Destructive Store Local Word / Word Offset.

Identical to STLW except that the EStack is NOT popped.

Note that this is an extended operation.

4.1.1.3. Own one word loads and stores

LDOB UB 116

Load Own Word / Byte Offset. Loads the value of the own variable with the offset in the second byte onto the EStack.

LDOW W 117

Load Own Word / Word Offset. Loads the value of the own variable at the offset in the next two bytes onto the EStack.

LOAB UB 118

Load Own Address / Byte Offset. Loads the virtual address of the Own variable with the specified byte offset onto the EStack.

LOAW W 119

Load Own Address / Word Offset. Loads the virtual address of the Own variable with the specified word offset onto the EStack.

STOB UB 120

Store Own Word / Byte Offset. Stores (ETOS) into the word with offset UB in the current Global Data Block (GDB).

STOW W 121

Store Own Word / Word Offset. Stores (ETOS) into the word with offset W in the current Global Data Block (GDB).

Implementation Note: The address of the first own (offset 0) is contained in the Global Pointer register (GP). The address of the Nth own is computed as (GP)+N.

4.1.1.4. Global one word loads and stores

LDGB UB1,UB2 122
Load Global Word / Byte Offset. Loads the word with offset UB2 in the Global Data Block (GDB) for program segment UB1 onto EStack.

LDGW UB,W 123
Load Global Word / Word Offset. Same as LDGB except a full word offset is used.

LGAB UB1,UB2 124
Load Global Address / Byte Offset. Pushes the virtual address of a global variable onto the EStack. UB2 specifies the byte offset of the global variable in segment UB1.

LGAW UB,W 125
Load Global Address / Word Offset. Same as LGAB except a full word offset is used.

STGB UB1,UB2 126
Store Global Word / Byte Offset. Stores (ETOS) in word with offset UB2 in the Global Data Block (GDB) for program segment UB1.

STGW UB,W 127
Store Global Word / Word Offset. Same as STGB except a full word offset is used.

Implementation Note: Self-relative pointers to the

Global Data Blocks (GDB) for each externally referenced segment are contained in the External Segment Table (XST), pointed to by the Global Pointer (GP). The address of the first global (offset 0) in the designated GDB is computed as GP - 2 * ISN, where ISN (Internal Segment Number) is the program segment number specified in the load or store instruction. The Nth global is addressed by the base address (computes as above) plus N.

exNDSTGB UB1,UB2 154

Non-Destructive Store Global Word / Byte Offset. Identical to STGB except that the EStack is NOT popped.

Note that this is an extended operation.

exNDSTGW UB, W 155

Non-Destructive Store Global Word / Word Offset. Identical to STGW except that the EStack is NOT popped.

Note that this is an extended operation.

4.1.1.5. Intermediate one word loads and stores

LDIB UB1,UB2 110

Load Intermediate Word / Byte Offset. UB1 indicates the number of static links to traverse to find the activation record to use. UB2 is the offset within the activation record of the desired word. The word is pushed on the EStack.

LDIW UB,W 111

Load Intermediate Word / Word Offset. Same as LDIB except a word offset is used.

LIAB UB1,UB2 112

Load Intermediate Address / Byte Offset. Loads the virtual address of an intermediate variable. UB1 is the number of static links to be traversed and UB2 is the byte offset in that activation record.

LIAW UB,W 113

Load Intermediate Address / Word Offset. Loads the virtual address of an intermediate variable. UB1 is the number of static links to be traversed and W is the word offset in that activation record.

STIB UB1,UB2 114

Store Intermediate Word / Byte Offset. Stores (ETOS) in memory (address determined as in LDIB).

STIW UB,W 115

Store Intermediate Word / Word Offset. Stores (ETOS) in memory (address determined as in LDIW).

Implementation Note: The Activation Pointer register (AP) contains the address of the current Activation Control Block (ACB). Within the ACB is the Static Link (SL) to the previous ACB. To compute the address of the first intermediate word of the desired level, traverse the Static Links to the correct ACB. Within the ACB is the Local Pointer (LP) for that activation record.

4.1.1.6. Indirect one word loads and stores

STIND 163

Store Indirect. (ETOS) is stored into the word pointed to by the virtual address (ETOS-1), (ETOS-2).

LDIND 173

Load Indirect. Word pointed to by virtual address (ETOS), (ETOS-1) is pushed on EStack.

4.1.2. Loads and stores of multiple words

4.1.2.1. Constant long loads

LDCN 19

Load Constant Nil. Pushes the value of NIL (e.g. a long 0) onto the EStack.

LDLC1 20

Pushes the long constant 1 onto (ETOS),(ETOS-1).

LDLCM1 21

Pushes the long constant -1 onto (ETOS),(ETOS-1).

LDLCB SB 22

Load long constant byte. Load SSB onto the EStack as a long, ie push two words onto the EStack.

LDDC <block> 23

Load Double Word Constant. <block> is a double word constant. Load the constant onto EStack. The bytes in <block> are in the order:

```
most-significant-word, low byte
most-significant-word, high byte
least-significant-word, low byte
least-significant-word, high byte
```

or

```
B2 B3 B0 B1
```

if the bytes in a double word are numbered 3 2 1 0.

4.1.2.2. Indirect long loads and stores

LBLB UB 164

Load Based Long / Byte Index. Indexes the virtual base address (ETOS), (ETOS-1), by UB words, and pushes the double word pointed to by the result on EStack.

LBL0 165

Load Based Long / 0 Index. (ETOS), (ETOS-1) is the virtual base address of a double word. The double word is pushed onto EStack.

STDW 166

Store Double Word. (ETOS),(ETOS-1) is a double word and (ETOS-2), (ETOS-3) is the virtual address of a double word block of memory. The double word is popped from EStack into the double word pointed to by (ETOS-2), (ETOS-3).

LDLIND 167

Load Long Indexed. (ETOS), (ETOS-1) contain the base address of an array of longs. (ETOS-2), (ETOS-3) contains an index into that array. The index is multiplied by two, added into the base address, and the long word at that computed address is fetched into (ETOS), (ETOS-1).

4.1.2.3. Indirect quad word loads and stores

exLDQ 56

Load Quad Word. (ETOS), (ETOS-1) is the virtual base address of a quad word. The quad word is pushed onto EStack. (ETOS) contains the lowest-addressed word and (ETOS-3) the highest-addressed word.

Note that this is an extended operation.

exSTQ 57

Store Quad Word. (ETOS)..(ETOS-3) is a quad word and (ETOS-4), (ETOS-5) is the virtual address of a quad word block of memory. The quad word is popped from EStack into the quad word pointed to by (ETOS-4), (ETOS-5). (ETOS) is stored into the lowest-addressed word and (ETOS-3) into the highest-addressed word.

Note that this is an extended operation.

4.1.2.4. Long loads and stores

LGLB UB1,UB2 76

Load the global long in segment UB1 at offset UB2. Two words are pushed onto the EStack.

LGLW UB1,W 77

Same as LGLB except that the offset in the segment is a word offset.

LLLB UB 64

Load Long Local Byte. Load the Local variable at byte offset UB as a long.

LLLW W 65

Load Long Local Byte. Load the Local variable at

word offset W as a long.

LILB UB1,UB2 68
Load Intermediate at nesting level UB1 and byte offset UB2 as a long variable.

LILW UB1,W 69
Load Intermediate at nesting level UB1 and word offset UB2 as a long variable.

LOLB UB 72
Load the own variable at byte offset UB as a long.

LOLW W 73
Load the own variable at word offset W as a long.

SLLB UB 66
Store (ETOS), (ETOS-1) in the local variable at byte offset UB.

SLLW W 67
Store (ETOS), (ETOS-1) in the local variable at word offset W.

SILB UB1,UB2 70
Store (ETOS), (ETOS-1) in the variable at nesting depth UB1 and byte offset UB2.

SILW UB1,W 71
Store (ETOS), (ETOS-1) in the variable at nesting depth UB1 and word offset W.

SOLB UB 74
Store (ETOS), (ETOS-1) in the own variable at byte offset UB.

SOLW W 75
Store (ETOS), (ETOS-1) in the own variable at word offset W.

SGLB UB1,UB2 78

Store (ETOS), (ETOS-1) in the global long in segment UB1 at offset UB2.

SGLW UB1,W 79

Same as SGLB except that the offset in the segment is a word offset.

LLLLB UB 128

Load aligned long local byte. Load the local variable at byte offset UB as a long value where UB is doubleword aligned (e.g., is an even number).

exNDSLLB UB 152

Non-Destructive Store Local Long / Byte Offset. Identical to SLLB except that the EStack is NOT popped.

Note that this is an extended operation.

exNDSLLW W 153

Non-Destructive Store Local Long / Word Offset. Identical to SLLW except that the EStack is NOT popped.

Note that this is an extended operation.

exNDSGLB UB1,UB2\156

Non-Destructive Store Global Long / Byte Offset. Identical to SGLB except that the EStack is NOT popped.

Note that this is an extended operation.

exNDSGLW UB1,W 157

Non-Destructive Store Global Long / Word Offset. Identical to SGLB except that the EStack is NOT

popped.

Note that this is an extended operation.

4.1.2.5. Multiple word loads and stores

exLDMC UB,BLOCK 95

Load Multiple Word Constant. UB is the number of words to load, and BLOCK is a block of UB words, in reverse word order. Load the block onto the MStack.

Note that this is an extended operation.

exLDMW 96

Load Multiple words. (ETOS-1), (ETOS-2) is the virtual address of the beginning of a block of (ETOS) words. Push the block onto the MStack, in reverse order.

Note that this is an extended operation.

exSTMW 94

Store Multiple Words. The MStack contains a block of (ETOS) words, (ETOS-1), (ETOS-2) is the virtual address of a similar block. Pop the block from MStack to the destination block, in reverse order.

Note that this is an extended operation.

4.1.3. Byte arrays

Note: A byte pointer is loaded onto the stack with a LLA, LOA or LGA of the base address of the array followed by the computation of the offset.

LDB 168

Load Byte. Push the byte (after zeroing the high byte) at byte-offset (ETOS) in virtual address (ETOS-1), (ETOS-2).

LDBLong 169

Load Byte / Long Index. Push the byte (after zeroing the high byte) at byte-offset (ETOS), (ETOS-1) in virtual address (ETOS-2), (ETOS-3).

LDBIND 170

Load Byte Indirect. (ETOS),(ETOS-1) contains the virtual address of the word containing the desired byte shifted left one bit position and concatenated with a 0 or a 1 bit to denote pushing the low-order or the high-order byte of that word. That byte (after zeroing the high byte) is pushed onto the EStack.

STB 171

Store Byte. Store the low byte of (ETOS) into the byte at byte-offset (ETOS-1) in virtual address (ETOS-2), (ETOS-3).

STBLong 172

Store Byte / Long Index. Store the low byte of (ETOS) into the byte at byte-offset (ETOS-1), (ETOS-2) in virtual address (ETOS-3), (ETOS-4).

STBIND 173

Store Byte Indirect. (ETOS-1),(ETOS-2) contains the virtual address of the word that is to contain the byte

shifted left one bit position and concatenated with a 0 or a 1 bit to denote storing into the low-order or the high-order byte of that word. The low-order byte of (ETOS) is placed into the desired byte position.

exMVBB UB 90

Move Bytes / Byte Counter. (ETOS) is the byte-offset into virtual address (ETOS-1),(ETOS-2) of the source block, UB bytes are moved into the block starting at byte-offset (ETOS-3) in virtual address (ETOS-4), (ETOS-5).

Note that this is an extended operation.

exMVBW 91

Move Bytes / Word Counter. Same as MVBB except (ETOS-1), (ETOS-2), (ETOS-3) is the source byte pointer, (ETOS-4), (ETOS-5), (ETOS-6) is the destination byte pointer, and (ETOS) is the number of bytes to transfer.

Note that this is an extended operation.

4.1.4. Strings

LSA UB,<chars> 24

Load String Address. UB is the length of the string constant <chars>. A word pointer is pushed on EStack (the virtual address of UB is pushed. UB is word aligned.

SAS 180

String Assign. (ETOS-1),(ETOS-2), (ETOS-3) is the source string pointer (a byte pointer), and (ETOS-4),(ETOS-5),(ETOS-6) is the destination string byte pointer. (ETOS) is the declared length of the destination. The length of the source and

destination are compared, and if the source string is longer than the destination, a run-time error occurs. Otherwise all bytes of source containing valid information are transferred to the destination string.

LDCH 174

Load Character. (ETOS-1),(ETOS-2) is a string pointer. (ETOS) is checked to insure that it lies within the dynamic length of the string. If so, the character pointed to by (ETOS-1),(ETOS-2) is pushed; otherwise, a run-time error occurs.

STCH 175

Store Character. (ETOS-2),(ETOS-3) is a string pointer. The character (ETOS) is put in at offset (ETOS-1) in the string. If (ETOS-1) is greater than the dynamic length of the string a runtime error occurs.

4.1.5. Record and array indexing and assignment

exMOVB UB 92

Move Words / Byte Counter. (ETOS), (ETOS-1) is the virtual address of a block of UB words, and (ETOS-2), (ETOS-3) is the virtual address of a similar block. The block pointed to by (ETOS),(ETOS-1) is transferred to the block pointed to by (ETOS-2), (ETOS-3).

Note that this is an extended operation.

exMOVW 93

Move Words / Word Counter. Same as MOVB except (ETOS-1),(ETOS-2) is the source pointer, (ETOS-3), (ETOS-4) is the destination pointer, and (ETOS) is the number of words to be transferred.

Note that this is an extended operation.

IND0-7 155-162

Short Index and Load Word. INDx indexes the virtual address (ETOS), (ETOS-1) by x words, and pushes the word pointed to by the result on EStack. (Note: LDIND is synonymous with IND0).

INDB UB 153

Static Index and Load Word / Byte Index. Indexes the virtual address (ETOS), (ETOS-1) by UB words, and pushes the word pointed to by the result on EStack.

INDW W 154

Static Index and Load Word / Word Index. Same as INDB except a full word index is used.

INCB UB 151

Increment Field Pointer / Byte Index. The virtual address (ETOS), (ETOS-1) is indexed by UB words and the resultant pointer is pushed on EStack.

INCW W 152

Increment Field Pointer / Word Index. Same as INCB except a full word index is used.

Note: INCB and INCW are equivalent to add UB or W to (ETOS).

IXAB UB 145

Index Array / Byte Array Size. (ETOS) is an integer index, (ETOS-1),(ETOS-2) is the virtual address of the base of the array, and UB is the size (in words) of an array element. The virtual address of the first word of the indexed element is pushed on EStack.

IXAW 146

Index Array / Word Array Size. Same as IXAB except (ETOS-1) is the integer index, (ETOS-2), (ETOS-3) is the virtual address of the base of the array, and (ETOS) is the size (in words) of an array element.

IXA1..4 147-150

Index Array / Short Array Size. Same as IXAB except array element sizes are fixed at 1-4.

LXAB UB 181

Index Array / Byte Array Size. (ETOS,ETOS-1) is a long index, (ETOS-2),(ETOS-3) is the virtual address of the base of the array, and UB is the size (in words) of an array element. The virtual address of the first word of the indexed element is pushed on EStack.

LXAW 182

Index Array / Word Array Size. Same as LXAB except (ETOS-1,ETOS-2) is the long index, (ETOS-3), (ETOS-4) is the virtual address of the base of the array, and (ETOS) is the size (in words) of an array element.

LXA2..4 183-185

Index Array / Short Array Size. Same as LXAB except array element sizes are fixed at 2-4.

IXP UB 176

Index Packed Array. (ETOS) is an integer index, and (ETOS-1), (ETOS-2) is the virtual address of the base of the array. Bits 4-7 of UB contain the number of elements per word minus 1, and bits 0-3 contain the field width (in bits) minus 1. Compute and push a packed field pointer.

IXPLong UB 177

Index Packed Array / Long. (ETOS,ETOS-1) is a long index, and (ETOS-2), (ETOS-3) is the virtual address of the base of the array. Bits 4-7 of UB contain the number of elements per word minus 1, and bits 0-3 contain the field width (in bits) minus 1. Compute and push a packed field pointer.

LDP 178

Load a Packed Field. Push the field described by the packed field pointer (ETOS),(ETOS-1),(ETOS-2) on EStack.

STPF 179

Store into Packed Field. Store (ETOS) in the field described by the packed field pointer (ETOS-1),(ETOS-2),(ETOS-3).

4.2. Top of Stack Arithmetic and Comparisons

4.2.1. Logical

ROTSHI UB 43

Rotate / Shift. (ETOS-1) is the argument to be rotated or shifted, and (ETOS) is the distance to rotate or shift. If UB is 1 then a right rotate occurs, and if UB is 0 then a shift occurs. The direction of the shift is determined from (ETOS); If (ETOS) >= 0 then a left shift occurs; otherwise, a right shift. (ETOS) must be in the range from -15 to +15.

QAND 25

Logical Add. AND (ETOS) into (ETOS-1).

QOR 26

Logical Or. OR (ETOS) into (ETOS-1).

QNOT 27

Logical Not. Take one's complement of (ETOS).

QXOR 28
Logical Exclusive-Or. XOR (ETOS) into (ETOS-1).

4.2.2. Integer

ABI 35
Absolute Value of Integer. Take absolute value of (ETOS). Result is undefined if (ETOS) is initially -32768.

ADI 36
Add Integers. Add (ETOS) and (ETOS-1).

NGI 37
Negate Integer. Take the twos complement of (ETOS).

SBI 38
Subtract Integers. Subtract (ETOS) from (ETOS-1).

MPI 39
Multiply Integers. Multiply (ETOS) and (ETOS-1). This instruction may cause overflow if the result is larger than 16 bits.

DVI 40
Divide Integers. Divide (ETOS-1) by (ETOS) and push quotient (as defined by K. Jensen and N.Wirth, *Pascal User Manual and Report*, Springer Verlog, New York, 1974). If (ETOS) is zero, a run-time error occurs.

MODI 41
Modulo Integers. MODI is actualy the remainder operation (as defined by K. Jensen and N.Wirth, *Pascal User Manual and Report*, Springer Verlog, New York, 1974). Denote A as (ETOS-1) and B as (ETOS). The result of the MODI operation is an

Integer such that (A rem B) has the SIGN OF A and an absolute value less than the absolute value of B; in addition this result must satisfy the relation:

(A rem B) = A - B * N

for some integral value of N. This result is pushed onto EStack. If (ETOS) is zero, a run-time error occurs.

CHK 42

Check Against Subrange Bounds. Insure that (ETOS-1) <= (ETOS-2) <= (ETOS), leaving (ETOS-2) on top of the stack. If conditions are not met a run-time error occurs.

EQUI 29

Integer Equality Test. Push true (1) if (ETOS-1) = (ETOS); otherwise, push false (0).

NEQI 30

Integer Inequality Test. Push true if (ETOS-1) <> (ETOS); otherwise, push false.

LEQI 31

Integer Less Than or Equal Test. Push true if (ETOS-1) <= (ETOS); otherwise, push false.

LESI 32

Integer Less Than Test. Push true if (ETOS-1) < (ETOS); otherwise, push false.

GEQI 33

Integer Greater Than or Equal Test. Push true if (ETOS-1) >= (ETOS); otherwise, push false.

GTRI 34

Integer Greater Than Test. Push true if (ETOS-1) >

(ETOS); otherwise, push false.

exMathMODI 166

Mathematical Integer Modulus. Denote A as (ETOS-1) and B as (ETOS). The result of the exMathMODI operation is an Integer such that (A mod B) has the SIGN OF B and an absolute value less than the absolute value of B; in addition this result must satisfy the relation:

(A mod B) = A - B * N

for some integral value of N. This result is pushed onto EStack.

Note that this is an extended operation.

4.2.3. Long

nADL 46

Add Longs. Add (ETOS),(ETOS-1) and (ETOS-2),(ETOS-3).

nSBL 47

Subtract Longs. Subtract (ETOS),(ETOS-1) from (ETOS-2),(ETOS-3).

nCVTLI 44

Convert long to integer. (ETOS), (ETOS-1) is converted into an integer and left on the stack.

nCVTIL 45

Convert Integer to long. (ETOS) is converted to the corresponding long. Includes sign extensions.

NGL 3

Negate Long. Negates long value in (ETOS), (ETOS-1).

Note that this is an extended operation.

MPL 5

Multiplies two long values. Multiples (ETOS), (ETOS-1) by (ETOS-2), (ETOS-3). Note that this is an extended operation.

DVL 6

Divides two long values. Divides (ETOS-2), (ETOS-3) by (ETOS), (ETOS-1) and pushes the long quotient onto the EStack. If (ETOS), (ETOS-1) is zero, a run-time error occurs.

Note that this is an extended operation.

MODL 7

Modulo Longs. MODL is actualy the remainder operation (as defined by K. Jensen and N.Wirth, *Pascal User Manual and Report*, Springer Verlog, New York, 1974). Denote A as (ETOS-2),(ETOS-3) and B as (ETOS),(ETOS-1). The result of the MODL operation is a Long such that (A rem B) has the SIGN OF A and an absolute value less than the absolute value of B; in addition this result must satisfy the relation:

(A rem B) = A - B * N

for some integral value of N. This result is pushed onto EStack.

Note that this is an extended operation.

ABL 8

Absolute value of a long value.

Note that this is an extended operation.

nEQULong 48

Long Equality Test. Push true if (ETOS-2),(ETOS-3) = (ETOS),(ETOS-1); otherwise, push false.

nNEQLong 49

Long Inequality Test. Push true if (ETOS-2),(ETOS-3) <> (ETOS),(ETOS-1); otherwise, push false.

nLEQLong 50

Long Less Than or Equal Test. Push true if (ETOS-2),(ETOS-3) <= (ETOS),(ETOS-1); otherwise, push false.

nLESLong 51

Long Less Than Test. Push true if (ETOS-2),(ETOS-3) < (ETOS),(ETOS-1); otherwise, push false.

nGEQLong 52

Long Greater Than or Equal Test. Push true if (ETOS-2),(ETOS-3) >= (ETOS),(ETOS-1); otherwise, push false.

nGTRLong 53

Long Greater Than Test. Push true if (ETOS-2),(ETOS-3) > (ETOS),(ETOS-1); otherwise, push false.

EQNIL 54

Equal to NIL. Push true if (ETOS),(ETOS-1) is equal to NIL; otherwise, push false.

LBITS 15

Count Bits in Long. Sets (ETOS) to the count of the number of bits asserted in the long in (ETOS),

(ETOS-1).

Note that this is an extended operation.

LBNOT 16

Logical Not Long. Take one's complement of (ETOS),(ETOS-1).

Note that this is an extended operation.

LBAND 17

Logical AND Long. AND (ETOS),(ETOS-1) into (ETOS-2),(ETOS-3)

Note that this is an extended operation.

LBOR 18

Logical OR Long. OR (ETOS),(ETOS-1) into (ETOS-2),(ETOS-3)

Note that this is an extended operation.

LBXOR 19

Logical Exclusive OR Long. XOR (ETOS),(ETOS-1) into (ETOS-2),(ETOS-3)

Note that this is an extended operation.

LOPSHI 246

Logical Shift/Long. (ETOS-1), (ETOS-2) is the argument to be shifted, and (ETOS) is the distance to shift. If (ETOS) >= 0 then a left shift occurs; otherwise a right shift. (ETOS) must be in the range from -31 to +31.

Note that this is an extended operation.

exCHKLong 20

Check Long Against Subrange Bounds. Insure that (ETOS-2),(ETOS-3) <= (ETOS-4),(ETOS-5) <= (ETOS),(ETOS-1); leaving (ETOS-4),ETOS-5) on top of the stack. If conditions are not met a run-time error occurs.

Note that this is an extended operation.

exMathMODL 167

Mathematical Long Modulus. Denote A as (ETOS-2), (ETOS-3) and B as (ETOS), (ETOS-1). The result of the exMathMODL operation is a Long such that (A mod B) has the SIGN OF B and an absolute value less than the absolute value of B; in addition this result must satisfy the relation:

(A mod B) = A - B * N

for some integral value of N. This result is pushed onto EStack.

Note that this is an extended operation.

4.2.4. Single Precision Real operations

This section describes arithmetic operations on single precision floating point (32 bit) values. In general, (ETOS) is the low-order word and (ETOS-1) is the high-order word of the value. When two floating point values are involved, (ETOS-2) is the low-order word of the second real and (ETOS-3) is the high-order word.

All over / underflows cause a run-time error. Division by zero (0) also causes a run-time error.

exGEQReal 34

>= of two single precision real values.

Note that this is an extended operation.

exGTRReal 35

> of two single precision real values.

Note that this is an extended operation.

exTNCRL 51

The single precision real (ETOS),(ETOS-1) is truncated (as defined by K. Jensen and N.Wirth, *Pascal User Manual and Report*, Springer Verlog, New York, 1974), converted to a long, and pushed onto EStack.

Note that this is an extended operation.

exFLTLR 52

The long (ETOS),(ETOS-1) is converted to a single precision floating-point number and pushed onto EStack.

Note that this is an extended operation.

exRNDRL 55

The single precision real (ETOS),(ETOS-1) is rounded (as defined by K. Jensen and N.Wirth, *Pascal User Manual and Report*, Springer Verlog, New York, 1974), truncated and converted to a long, and pushed onto EStack.

Note that this is an extended operation.

4.2.5. Double Precision Real operations

This section describes arithmetic operations on double precision floating point (64 bit) values. In general, (ETOS) is the lowest-order word and (ETOS-3) is the highest-order word of the value. When two floating point values are involved, (ETOS-4) is the lowest-order word of the second real and (ETOS-7) is the highest-order word.

All over / underflows cause a run-time error. Division by zero (0) also causes a run-time error.

exTNCQL 36
The real (ETOS)..(ETOS-3) is truncated (as defined by K. Jensen and N.Wirth, *Pascal User Manual and Report*, Springer Verlog, New York, 1974), converted to a long, and pushed onto EStack.

Note that this is an extended operation.

exFLTLQ 37
The long (ETOS),(ETOS-1) is converted to a floating-point number and pushed onto EStack.

Note that this is an extended operation.

exADDQ 38
Add (ETOS)..(ETOS-3) and (ETOS-4)..(ETOS-7).

Note that this is an extended operation.

exNEGQ 39
Negate the real (ETOS)..(ETOS-3).

Note that this is an extended operation.

exSUBQ 40

Subtract (ETOS)..(ETOS-3) from (ETOS-4)..(ETOS-7).

Note that this is an extended operation.

exMULQ 41

Multiply (ETOS)..(ETOS-3) and (ETOS-4)..(ETOS-7).

Note that this is an extended operation.

exDIVQ 42

Divide (ETOS)..(ETOS-3) by (ETOS-4)..(ETOS-7).

Note that this is an extended operation.

exRNDQL 43

The real (ETOS)..(ETOS-3) is rounded (as defined by K. Jensen and N.Wirth, *Pascal User Manual and Report*, Springer Verlog, New York, 1974), truncated and converted to a long, and pushed onto the EStack.

Note that this is an extended operation.

exABSQ 44

Take the absolute value of the real (ETOS)..(ETOS-3).

Note that this is an extended operation.

exEQUQ 45

= of two double precision real values. (ETOS) = true or false.

Note that this is an extended operation.

exNEQQ 46

<> of two double precision real values.

Note that this is an extended operation.

exLEQQ 47
<= of two double precision real values.

Note that this is an extended operation.

exLESQ 48
< of two double precision real values.

Note that this is an extended operation.

exGEQQ 49
>= of two double precision real values.

Note that this is an extended operation.

exGTRQ 50
> of two double precision real values.

Note that this is an extended operation.

exCVQR 53
The double precision real (ETOS)..(ETOS-3) is converted to the nearest single precision real, and pushed onto EStack.

Note that this is an extended operation.

exCVRQ 54
The single precision real (ETOS),(ETOS-1) is converted to the nearest double precision real, and pushed onto EStack.

Note that this is an extended operation.

4.2.6. Sets

exADJ UB 84

Adjust Set. The set on the top of the MStack is forced to occupy UB words, either by expansion or compression, and its length word is popped from EStack.

Note that this is an extended operation.

exSGS 77

Build Singleton Set. The integer (ETOS) is checked to insure that 0 <= (ETOS) <= 4,095, the set [(ETOS)] is pushed on MStack, and the size of the set is pushed on EStack. If (ETOS) is out of range, the null set is pushed (a zero is pushed on EStack, the MStack is not altered).

Note that this is an extended operation.

exSRS 79

Build SubRange Set. The integers (ETOS) and (ETOS-1) are checked as in SGS, the set [(ETOS-1)..(ETOS)] is pushed onto MStack, and the size of the set is pushed on EStack. (The null set is pushed if (ETOS-1) > (ETOS) or either is out of range).

Note that this is an extended operation.

exINN 80

Set Membership. See if integer (ETOS) is in set contained on the top of MStack, and with length (ETOS-1), pushing TRUE or FALSE on EStack.

Note that this is an extended operation.

exUNI 81

Set Union. The union of the two sets contained on the top of MStack (with sizes (ETOS) and (ETOS-1)) is pushed on MStack, and the length of the result on EStack.

Note that this is an extended operation.

exQINT 82

Set Intersection. The intersection of the two sets contained on the top of MStack (with sizes (ETOS) and (ETOS-1)) is pushed on MStack, and the length of the result on EStack.

Note that this is an extended operation.

exDIF 83

Set Difference. The difference of the two sets contained on the top of MStack, and sizes (ETOS) and (ETOS-1) is pushed on MStack, and the length of the result on EStack.

Note that this is an extended operation.

exEQUPOWR 74

Set =,

Note that this is an extended operation.

exNEQPOWR 75

Set <>,

Note that this is an extended operation.

exLEQPOWR 76

Set <= (subset of),

Note that this is an extended operation.

exGEQPOWR 78
and Set >= (superset of) comparisons of the two sets on top of EStack, with sizes (ETOS) and (ETOS-1).

Note that this is an extended operation.

4.2.7. Strings

exEQUSTR 62
String =,

Note that this is an extended operation.

exNEQSTR 63
String <>,

Note that this is an extended operation.

exLEQSTR 64
String <=,

Note that this is an extended operation.

exLESSTR 65
String <,

Note that this is an extended operation.

exGEQSTR 66
String >=,

Note that this is an extended operation.

exGTRSTR 67
and > String comparisons. The string pointed to by string pointer (ETOS-3),(ETOS-4),(ETOS-5) is

lexicographically compared to the string pointed to by string pointer (ETOS),(ETOS-1),(ETOS-2).

Note that this is an extended operation.

4.2.8. Byte arrays

exEQUBYT UB 68
Byte Array =,

Note that this is an extended operation.

exNEQBYT UB 69
Byte Array <>,

Note that this is an extended operation.

exLEQBYT UB 70
Byte Array <=,

Note that this is an extended operation.

exLESBYT UB 71
Byte Array <,

Note that this is an extended operation.

exGEQBYT UB 72
Byte Array >=,

Note that this is an extended operation.

exGTRBYT UB 73
and Byte Array > comparisons. <=, <, >=, and > are only emitted for packed arrays of characters. Byte array 1 is compared to byte array 2, and the result is pushed onto the EStack.

If the argument UB is non-zero, then UB is the size of the arrays. (ETOS),(ETOS-1),(ETOS-2) is a byte pointer to byte array 2.
(ETOS-3),(ETOS-4),(ETOS-5) is a byte pointer to byte array 1.

If the argument UB is zero, then (ETOS) is the size of the arrays. (ETOS-1),(ETOS-2),(ETOS-3) is a byte pointer to byte array 2.
(ETOS-4),(ETOS-5),(ETOS-6) is a byte pointer to byte array 1.

Note that this is an extended operation.

4.2.9. Array and record comparisons

exEQUWORD UB 85

Word or multiword structure =

Note that this is an extended operation.

exNEQWORD UB 86

and word or multiword structure <> comparisons. Word array 1 is compared with word array 2, and the result is pushed onto the EStack.

If the argument UB is non-zero, then UB is the size of the arrays. (ETOS),(ETOS-1) is a word pointer to array 2, and (ETOS-2),(ETOS-3) is a word pointer to array 1.

If the argument UB is zero, then (ETOS) is the size of the arrays. (ETOS-1),(ETOS-2) is the word pointer to array 2, and (ETOS-3),(ETOS-4) is the word pointer to array 1.

Note that this is an extended operation.

4.3. Jumps

JMPB B 223
Unconditional Jump / Byte Offset. B is added to the IPC. Negative values of B cause backward jumps.

JMPW W 224
Unconditional Jump / Word Offset. W is added to the IPC. Negative values of W cause backward jumps.

JFB B 225
False Jump / Byte Offset. Jump (as in JMPB) if (ETOS) is false.

JFW W 226
False Jump / Word Offset. Jump (as in JMPW) if (ETOS) is false.

JTB B 227
True Jump / Byte Offset. Jump (as in JMPB) if (ETOS) is true.

JTW W 228
True Jump / Word Offset. Jump (as in JMPW) if (ETOS) is true.

JEQB B 229
Equal Jump / Byte Offset. Jump (as in JMPB) if integer (ETOS) equals (ETOS-1).

JEQW W 230
Equal Jump / Word Offset. Jump (as in JMPW) if integer (ETOS) equals (ETOS-1).

JNEB B 231
Not Equal Jump / Byte Offset. Jump (as in JMPB) if integer (ETOS) is not equal to (ETOS-1).

JNEW W 232

Not Equal Jump / Word Offset. Jump (as in JMPW) if integer (ETOS) is not equal to (ETOS-1).

XJP W1,W2,W3,<Case Table> 222

Case Jump. W1 is word-aligned, and is the minimum index of the table. W2 is the maximum index. W3 is the offset to the code to be executed if the case specified has no entry in the case table. The case table is W2 - W1 + 1 words long and contains offsets to the code to be executed for each case.

If (ETOS), the actual index, is not in the range W1..W2 then W3 is added to PC. Otherwise, (ETOS) - W1 is used as an index into the case table and the index entry is added to PC.

4.4. Routine Calls and Returns

Note: There can be at most 256 routines in a segment.

CALLL UB 239

Call Routine. Call routine UB, which is in the current segment.

CALLXB UB1,UB2 238

Call External Routine / Byte Segment. UB1 is the internal segment number (ISN) which contains the routine numbered UB2 to be called.

CCALL 235

Call a C procedure. (ETOS), (ETOS-1) is the address of the procedure to be called. (ETOS-2) is the number of parameters of the procedure. Remove the procedure address from the EStack and jump to it. Push the address of the next word after the instruction onto (ETOS),(ETOS-1). Leaves the number of

parameters in (ETOS-2).

Note: the return address pushed is always word-aligned. A NOP instruction must follow the CCALL if it does not end on a word boundary.

CENTER 236

Enter a C procedure. (ETOS), (ETOS-1) define the virtual word address of the return point. (ETOS-2) is the size of the parameters being passed into the C procedure. This is used to save the required space on the stack before pushing the ACB.

CRET 237

Return from a C procedure.

LDRET1 233

Load a one word return value from a C procedure onto the expression stack.

LDRET2 234

Load a two word return value from a C procedure onto the EStack.

SVRET1 239

Save a one-word C procedure return value.

Note that this is an extended operation.

SVRET2 241

Save a two-word C procedure return value.

Note that this is an extended operation.

WARNING: The C instructions: CCALL, CENTER, CRET, LDRET1, LDRET2, SVRET1, and SVRET2 must not be used from within a Pascal program.

exLVRD W,UB1,UB2 103

Load Variable Routine Descriptor. This Q-Code pushes a Variable Routine Descriptor on the EStack for the routine UB1 in segment ISN W, at lexical level UB2. The following values (which comprise a variable routine descriptor) are pushed: (ETOS) = System Segment Number (SSN); (ETOS-1) = Global Pointer, represented as an offset from SB; (ETOS-2) = Routine Number; and (ETOS-3) = Static Link (determined as if a call were actually performed to the routine here).

Note that this is an extended operation.

CALLV 240

Call Variable Routine. The EStack elements (ETOS) --- (ETOS-3) are a variable routine descriptor (as described above in LVRD).

RET 241

Return from Routine. Return from the current routine. If the routine was a function, the function value is left on the top of the MStack. Since the first word of a code segment is not code, but an offset to the routine dictionary, the return is performed to the exit code of that routine, if the RA which is being returned to is 0. (This proves useful for the EXIT and EXGO Q-Codes described below).

EXITT W,UB 242

Exit from Routine. Exit from all routines up to and including the most recent invocation of the routine UB in ISN W. This is accomplished by setting the RAs in all the ACBs to 0, from the most recent through and including the first ACB which was created from an invocation of the routine to be exited,

and jumping to the exit code of the current routine.

EXGO W1,UB,W2 243

Exit and Goto. Exit from all routines up to, but not including, routine UB in ISN W1, and then jump to the instruction with offset W2 from CB. The implementation is similar to EXIT, except the last RA modified is loaded with W2.

exENABLE W,UB1,UB2 98

Enable Exception Handler. W and UB1 are the internal segment and routine numbers, respectively, of the exception being enabled. UB2 is the routine number of the handler. A new exception enable record is pushed (quad word aligned) onto the MStack and linked into the routine's current exception list.

Note that this is an extended operation.

exQRAISE W1,UB,W2 99

Raise Exception. W1 and UB1 are the internal segment and routine numbers, respectively, of the exception to be raised. W2 is the number of words of parameters that have already been pushed onto the memory stack. The exception is raised.

Note that this is an extended operation.

exSETEXC 97

Set-up Exceptions. Initializes the exception mechanism by supplying the micro-code the code segment (CS) and global pointer (GP) of the Pascal routine utilized to process a RAISE of an exception. This instruction should be issued once per process as part of its standard initialization sequence. The code

segment and global pointer are set from the segment in which SETEXC is executed.

Note that this is an extended operation.

exJLK UW 60

Jump and Link. Push the current PC onto the memory stack, as an unsigned offset from the base of the current code segment (CS), and jump to the offset UW within the current code segment (CS).

Note that this form of call does not set up any of the run-time frame described above.

Note that this is an extended operation.

exJMS 61

Jump on Memory Stack. Pop a 16-bit unsigned word from the memory stack. Treat the value popped as an offset within the current code segment (CS). Jump to that location.

Note that this is an extended operation.

4.5. Kernel Opcodes for Accent

KOPS UB 252

Kernel Operations. The Accent Kernel Operation, UB, is executed. All Accent Kernel operations are two-byte order codes. They all must begin with this opcode.

Specific Kernel operations are described below. In general users should NOT issue these instructions. They are intended for internal use by the Kernel itself. Users should utilize the routines available in

the library modules AccentUser and AccCall to obtain services from the Kernel (which issues these instructions in a safe way) rather than issuing these instructions directly. Most of these Kernel operations require the issuing process to be in Supervisor mode.

KSEARCH 18

Search the VMTable for an inuse entry for the process number (ETOS-4). Searches in the range of addresses starting with (ETOS-2),(ETOS-3) and ending with (ETOS),(ETOS-1). (ETOS-5) is the first entry to search; (ETOS-6) is the last entry to search. Returns index in (ETOS).

KUNLOCKSVC 17

If the supervisor is locked for this process, then let in the next process waiting to enter the supervisor, or if none, then just mark supervisor as not busy.

KLOCKSVC 16

The supervisor is locked for the process or the process is blocked till the supervisor is not busy. Returns true on the EStack if the lock was effective; false otherwise.

KBLOCK 15

Place the current process on the block queue (ETOS). The process is put on the front of the queue if (ETOS-1) is non-zero and on the back if it is zero.

KUNBLOCK 14

Unblock the process in (ETOS). The process is placed either on the potentially runnable queue or on its priority queue. Pre-emptive scheduling is performed if the process has a higher priority than the current process.

KSLEEP 13

The process in (ETOS-2) is put on the sleep queue for (ETOS-3) ticks. (ETOS-4) and (ETOS-5) define the sleep ID which is used by the WAKEUP primitive.

(ETOS) and (ETOS-1) define a pointer to a boolean. If the pointer is non-NIL, and the boolean is FALSE, it is set TRUE, and the process continues executing. If the boolean does not exist, or is TRUE, the process sleeps as described.

KWAKEUP 12

The sleep queue is scanned and all processes [if (ETOS-4) is true, one otherwise] matching the sleep id in (ETOS-2), (ETOS-3) are unblocked. (ETOS) and (ETOS-1) define a pointer to a boolean. The boolean (if the pointer is non-nil) is reset if the instruction does not actually wake up any processes.

KREMOVFROMQUEUE 11

The process (ETOS) is removed from whichever queue it is on.

KADDTOQUEUE 10

Add the process (ETOS) to the queue (ETOS-1). If (ETOS-2) is true then add to the front of the queue, otherwise to the rear.

KSVCALL 9

Supervisor call of type (ETOS). The parameters to the call are in the virtual address (ETOS-1), (ETOS-2). The parameters are placed in the process context for the supervisor and the process changes state. Only one process can make an SVCall at a time.

KSETVMTABLES 6

Used to size the system tables to match the memory size. (ETOS) is the virtual memory hash mask. (ETOS-1),(ETOS-2) is the physical address of the new VMLink table. (ETOS-3), (ETOS-4) is the physical address of the new VMTable. (ETOS-5),(ETOS-6) is the address of the new VMPhysTable.

KGETCB 5

A fast way to get data from a circular buffer. (ETOS) is the Z80 device number. Returns -1 in (ETOS) if successful, 0 otherwise. (ETOS-1) has the 16 bit item from the buffer (garbage if ETOS=0). If the device number designates the pointing device, 3 words are returned on the EStack. (ETOS-2) contains the *x* position, (ETOS-1) contains the *y* position, and (ETOS) contains the encoded state of the buttons. This is *not* the same as the position returned by the window manager.

KCLOCK 4

Return the current clock time. (ETOS) has the LSW and (ETOS-1) the MSW of the clock time.

KSETSOFT 3

Sets the software interrupt (message pending) bit in the PCB handle for the (ETOS) process. If (ETOS-1) is true, then this is an emergency message; else a normal message.

KHASH 2

Hashes a virtual address and a process number into an index into the hash tables. (ETOS) is the process number, (ETOS-1), (ETOS-2) forms the virtual address. Returns the index of the entry containing the address on (ETOS), or VPNIL if not in the table.

KCURPROCESS 1

Return in (ETOS) the process ID of the current process.

KSVRETURN 0

Performs a return from the supervisor state. The opcode has a number of side effects on the state of the operating system. The opcode performs a context swap into the user state.

KVPENTER 19

Adds a virtual address to the Kernel's virtual to physical address mapping table.

On entry to this Kernel operation (ETOS),(ETOS-1) contains the virtual address; (ETOS-2) contains the process number; (ETOS-3),(ETOS-4) contains the physical address of PVList; (ETOS-5),(ETOS-6) contains the physical address of VPFree table; and (ETOS-7) contains the physical page index;

If the entry was successfully entered, (ETOS) is 1 and (ETOS-1) is the index of an entry that had to be moved, or VPNIL if no entry was moved.

If the entry was not entered, (ETOS) contains 0, and (ETOS-1) is undefined.

KVPREMOVE 20

Removes a virtual address from the Kernel's virtual to physical address mapping table.

On entry to this Kernel operation (ETOS),(ETOS-1) contains the virtual address; (ETOS-2) contains the process number; (ETOS-3),(ETOS-4) contains the physical address of PVList; and (ETOS-5),(ETOS-6)

contains the physical address of VPFree table.

This Kernel operation will change (ETOS) to be the index of the entry that was moved, if any, otherwise (ETOS) will contain VPNIL.

KSEARCHADDR 21

Searches the VP table for entries in a range of addresses. Searches by address order, rather than by scanning entire table.

On entry to this Kernel operation (ETOS) contains the number of pages to search; (ETOS-1) contains the process number; and (ETOS-2),(ETOS-3) contains the start of the address range.

This Kernel operation will change the EStack as follows: (ETOS) will contain the number of pages remaining to scan, *including* the page found or 0 if address range exhausted; (ETOS-1) will contain the VP table index of the page found; and (ETOS-2),(ETOS-3) will contain the address found.

KSEARCHPV 22

Searches PV table for entry which can be made free. Finds least used page and may (if flag is true) reset used bits on VP and PV entries when the end is reached.

On entry to this Kernel operation (ETOS) contains the starting index; (ETOS-1) contains the ending index + 1; and (ETOS-2) contains a flag indicating that we are searching for any page and not just completely unmapped pages. If this flag is set we will reset used bits and counts on pages when the ending index is reached.

This Kernel operation will set (ETOS) to contain the index which can be made free or the ending index.

KCOPYPAGE 23

Copies a page of data from one page to another. If (ETOS-5) is true, it will zero destination page instead (source page is then ignored). The physical address of the source page is in (ETOS),(ETOS-1). The physical address of the destination pages is in (ETOS-2),(ETOS-3). (ETOS-4) contains the number of words to copy / zero. Note that the physical addresses must be quad word aligned and that the count must be a multiple of four words.

4.6. Register Operations

These instructions utilize the four working registers. The working registers are 32 bit registers (pairs of PERQ XY-registers).

LLR0..3 133-136

Load Long Register. Push the contents of the specified register pair onto the EStack.

SLR0..3 129-132

Store Long Register. Store (ETOS),(ETOS-1) into the specified register.

LIR0..3 141-144

Load Integer Register. Push the contents of the least-significant word of the specified register pair onto the EStack.

SIR0..3 137-140

Store Integer Register. Store (ETOS) into the least-significant word the specified register.

exNDSLR0..3 162-165

Non-Destructive Store Long Register. Identical to

SLR0..3 except that the EStack is NOT popped.

Note that this is an extended operation.

exNDSIR0..3 158-161
Non-Destructive Store Integer Register. Identical to SIR0..3 except that the EStack is NOT popped.

Note that this is an extended operation.

INCREG0..3 254-251
Increment one of the four working registers.

Note that these are extended operations.

DECREG0..3 250-247
Decrement one of the four working registers.

Note that these are extended operations.

LBIR0..3B UB 194-197
Load Based Integer, Register plus Byte offset. The word contained at the virtual address obtained by adding the contents of the specified register to UB is pushed on to the EStack.

exLBIR0..3W W 104-107
Load Based Integer, Register plus Word offset. The word contained at the virtual address obtained by adding the contents of the specified register to W is pushed on to the EStack.

Note that these are extended operations.

LBLR0..3B UB 198,200,202,204
Load Based Long, Register plus Byte offset. The double word contained at the virtual address obtained

by adding the contents of the specified register to UB is pushed on to the EStack.

LBLR0..3W W 199,201,203,205
Load Based Long, Register plus Word offset. The double word contained at the virtual address obtained by adding the contents of the specified register to W is pushed on to the EStack.

exLBQR0..3B UB 108,110,112,114
Load Based Quad, Register plus Byte offset. The quad word contained at the virtual address obtained by adding the contents of the specified register to UB is pushed on to the EStack.

Note that these are extended operations.

exLBQR0..3W W 109,111,113,115
Load Based Quad, Register plus Word offset. The quad word contained at the virtual address obtained by adding the contents of the specified register to W is pushed on to the EStack.

Note that these are extended operations.

LBAR0..3B UB 186,188,190,192
Load Based Address, Register plus Byte offset. The virtual address obtained by adding the contents of the specified register to UB is pushed on to the EStack.

LBAR0..3W W 187,189,191,193
Load Based Address, Register plus Word offset. The virtual address obtained by adding the contents of the specified register to W is pushed on to the EStack.

IXAR0..3B UB 206,208,210,212
Integer Index Register/Byte Array Size. (ETOS) is

an integer index, the register contains the virtual address of the base of the array, and UB is the size (in words) of an array element. The virtual address of the first word of the indexed element is pushed on EStack.

IXAR0..3W 207,209,211,213

Integer Index Register / Word Array Size. Same as IXAR0..3B except (ETOS-1) is the integer index, the register contains the virtual address of the base of the array, and (ETOS) is the size (in words) of an array element.

exIXAR0..3 / 1..4 116-131

Integer Index Register / Short Array Size. Same as IXAR0..3B except array element sizes are fixed at 1-4.

Note that these are extended operations.

LXAR0..3B UB 214,216,218,220

Long Index Register / Byte Array Size. (ETOS),(ETOS-1) is a long index, the register contains the virtual address of the base of the array, and UB is the size (in words) of an array element. The virtual address of the first word of the indexed element is pushed on EStack.

LXAR0..3W 215,217,219,221

Long Index Register / Word Array Size. Same as LXAR0..3B except (ETOS-1),(ETOS-2) is the long index, the register contains the virtual address of the base of the array, and (ETOS) is the size (in words) of an array element.

exLXAR0..3 / 1..4 132-147

Long Index Register / Short Array Size. Same as

LXAR0..3B except array element sizes are fixed at 1-4.

Note that these are extended operations.

4.7. Systems Programs Support Procedures

BREAK 254

Breakpoint QCode. Causes a Qcode level breakpoint to the microcode kernel (KRNL).

NOOP 253

No-Operation.

REPL 57

Replicate. Replicate (ETOS).

REPL2 58

Replicate Two. Replicate two top-of-EStack words (i.e., first push original (ETOS-1), then push original (ETOS)).

exPOP1 148

Pop EStack Once. The contents of (ETOS) are popped and discarded.

Note that this is an extended operation.

exPOP2 149

Pop EStack Twice. The contents of (ETOS),(ETOS-1) are popped and discarded.

Note that this is an extended operation.

MMS 59

Move to Memory Stack. Push (ETOS) onto MTOS (16-bit transfer).

MES 61

Move to Expression Stack. Push (MTOS) onto ETOS (16-bit transfer - top 4 bits are zeroed).

MMS2 60

Move Double to Memory Stack. Transfer the top two words from the EStack to the MStack. The order is reversed; old (ETOS) is (MTOS-1), (ETOS-1) is (MTOS).

MES2 62

Move Double to Expression Stack. Transfer the top two words from the MStack to the EStack. The order is reversed; old (MTOS) is (ETOS-1), (MTOS-1) is (ETOS).

exRASTOP 87

RasterOp. RasterOp is a special QCode which is used to manipulate blocks of memory of arbitrary sizes. It is especially useful for creating and modifying displays on the screen. RasterOp modifies a rectangular area (called the "destination") of arbitrary size (to the bit). The picture drawn into this rectangle is computed as a function of the previous contents of the destination and the contents of another rectangle of the same size called the "source." The functions performed to combine the two pictures are described below.

RasterOp can be used on memory other than that used for the screen bitmap. There are two parameters that specify the areas of memory to be used for the source and destination: a pointer to the start of the memory block, and the length (in words) of scanlines in the block. A scanline is one of the elements that cross the block. On the portrait screen, for example, a scanline is one of the horizontal lines with a length of 48

words. Within these regions, the positions of the source and destination rectangles are given as offsets from the pointer. Thus position (0,0) would be at the upper left corner of the region, and, for the screen, (767, 1023) would be the lower right.

The EStack must be arranged in the following order for RASTER-OP:

(ETOS-12) Function
(ETOS-11) Width
(ETOS-10) Height
(ETOS-9) Destination-X-Position
(ETOS-8) Destination-Y-Position
(ETOS-7) Destination-Area-Line-Length
(ETOS-6) Destination-Memory-Pointer (Must be a *physical* address) (ETOS-5)
(ETOS-4) Source-X-Position
(ETOS-3) Source-Y-Position
(ETOS-2) Source-Area-Line-Length
(ETOS-1), Source-Memory-Pointer (Must be a *physical* address) (ETOS)

The values on the stack are defined below:

"Function" defines how the source and the destination are to be combined to create the final picture stored at the destination. The RasterOp functions are as follows (Src represents the source and Dst the destination):

Function	Name	Action
0	RRpl	Dst gets Src
1	RNot	Dst gets NOT Src
2	RAnd	Dst gets Dst AND Src
3	RAndNot	Dst gets Dst AND NOT Src
4	ROr	Dst gets Dst OR Src
5	ROrNot	Dst gets Dst OR NOT Src
6	RXor	Dst gets Dst XOR Src
7	RXNor	Dst gets Dst XNOR Src

"Width" specifies the size in the horizontal ("x") direction of the source and destination rectangles (in bits).

"Height" specifies the size in the vertical ("y") direction of the source and destination rectangles (in scan lines).

"Destination-X-Position" is the bit offset of the left side of the destination rectangle. The value is offset from Destination-Memory-Pointer (see below).

"Destination-Y-Position" is the scan-line offset of the top of the destination rectangle. The value is offset from Destination-Memory-Pointer (see below).

"Destination-Area-Line-Length" is the number of words which comprise a line in the destination region (hence defining the region's width). The appropriate value to use when operating on the screen is 48. in the destination region (hence defining the region's width). The appropriate value to use when operating on the portrait screen is 48; and on the landscape screen, 80. The specified value must be a multiple of four (4).

"Destination-Memory-Pointer" is the virtual address of the top left corner of the destination region. This pointer MUST be quad-word aligned, however.

"Source-X-Position" is the bit offset of the left side of the source rectangle. The value is offset from Source-Memory-Pointer (see below).

"Source-Y-Position" is the scan-line offset of the top of the source rectangle. The value is offset from Source-Memory-Pointer (see below).

"Source-Area-Line-Length" is the number of words which comprise a line in the source region (hence defining the region's width). The appropriate value to use when operating on the screen is 48. The specified value must be a multiple of four (4) and within the range 4 through 48.

"Source-Memory-Pointer" is the virtual address of the top left corner of the source region. This pointer MUST be quad-word aligned, however.

Note that this is an extended operation.

A runtime error occurs if the process does not have physical memory privileges enabled.

exSTRROP 88

String Raster-Op. StrRop puts characters from a byte array onto the screen. It draws until a character count is exhausted, a screen width is exhausted, an end of page is encountered on the byte string, or a control character is reached.

On entry to exSTRROP the EStack should contain the following:

(ETOS) = Byte offset from the beginning of the byte array. (ETOS-1,2) = Virtual address of the byte array. (ETOS-3) = Maximum byte offset + 1. (ETOS-4,5) = Font table physical address. (ETOS-6) = Maximum X-coordinate + 1. (ETOS-7,8) = Destination physical base address. (ETOS-9) = Destination Y-coordinate. (ETOS-10) = Destination X-coordinate. (ETOS-11) = packed record

```
Func: (low 3 bits) Raster-op
      function;
Scan: (bits 15..3) Words per
      scan line (0 means 60)
      end.
```

On exit from exSTRROP the EStack will contain:

(ETOS) = Current X-Coordinate. (ie, where to start next time) (ETOS-1) = Next byte offset. (ETOS-2) = Termination condition:

```
0 - Character count exhausted.
1 - Screen width exhausted.
2 - Control character encountered, or
3 - Page fault encountered.
```

Note that this is an extended operation.

A run-time error occurs if the process does not have physical memory privileges enabled.

exLINE 89

(ETOS) Address of area on which line is drawn (a physical
(ETOS-1) address)
(ETOS-2) Scan line width in words (must be a multiple of 4)

(ETOS-3) *x* coordinate of start of line
(ETOS-4) *y* coordinate of start of line
(ETOS-5) *x* coordinate of end of line
(ETOS-6) *y* coordinate of end of line
(ETOS-7) Drawing style:
0: draw line (set points to 1) 1: erase line (set points to 0) 2: invert line (complement points)

Note that this is an extended operation.

A run-time error occurs if the process does not have physical memory privileges enabled.

exSTRTIO 102

Perform an IO operation. (ETOS) is the channel on which to start IO. Other parameters may be required for the specific device. See the ID microcode for details.

Note that this is an extended operation.

A run-time error occurs if the process does not have physical memory privileges enabled.

EXCH 55

Exchange. (ETOS) and (ETOS-1) are swapped.

EXCH2 56

Exchange Double. The pair (ETOS) and (ETOS-1) are swapped with the pair (ETOS-2) and (ETOS-3).

exEXCHQ 58

Exchange Quad. The quad word in (ETOS)..(ETOS-3) is swapped with the quad word (ETOS-4)..(ETOS-7).

Note that this is an extended operation.

exPERMD 59

Permute Double. The double word in (ETOS),(ETOS-1) is swapped with the quad word (ETOS-2)..(ETOS-5).

Note that this is an extended operation.

LDTP 247

Load Top Pointer (plus 1). Pushes the virtual address of Top Pointer (TP) plus 1 onto EStack.

LDAP 246

Load Activation Pointer. The virtual address of the current activation pointer is pushed onto the EStack.

ATPB SB 244

Add to Top Pointer/Byte Value. Adds SB to TP.

ATPW 245

Add to Top Pointer/Word Value. Adds (ETOS) to TP.

WCS 249

Write Control Store. A control store word is written from information on the EStack. (ETOS) is the address (with bytes exchanged) in the control store to which the word will be written. (ETOS-1) is the value to be written into the high-order third, (ETOS-2) is the value to be written into the middle third and (ETOS-3) is to be written into the low-order third.

JCS 250

Jump to a Location in the Control Store. Control is transferred to the control store address (with bytes exchanged) given in (ETOS). A routine called with JCS should exit with a NextInst(0) jump.

EVENT 248

EVENT is used for a Q-code level event registration. It generates a TrapEvent fault, which is handled by the Pager / Scheduler process. (ETOS) contains the event code and (ETOS-1),(ETOS-2) contain a pointer to the event parameter block.

GOTOOVL 251

Goto Micro-code Overlay. (ETOS) is the entry descriptor for the overlaid micro-code routine to be entered; its low order 4 bits are the entry number and its remaining high order bits are the overlay number.

REFILLOP 255

Refill the OpFile. This instruction causes execution to proceed from the beginning of the next quad-word.

exZEROMEM 100

Zero Memory. Zeroes (ETOS) words beginning at the virtual address in (ETOS-1),(ETOS-2).

Note that this is an extended operation.

exINCDDS 101

Increment Diagnostic Display. The value of the diagnostic display is incremented and the contents of the EStack is checked. If the EStack is not empty, a runtime error is generated.

Note that this is an extended operation.

EXOP UB 63

Extended Opcodes. The first byte of each extended instruction. UB defines the function to be performed.

CVTCI 245

Convert the signed character in (ETOS) to a 16-bit

integer in (ETOS).

Note that this is an extended operation.

CVTCL 243

Convert the signed character in (ETOS) to a 32-bit long in (ETOS), (ETOS-1).

Note that this is an extended operation.

APPENDIX A
REGISTER ALLOCATION

December 7, 1984

2600 Liberty Avenue
P. O. Box 2600
Pittsburgh, PA 15230
(412) 355-0900

Accent is a trademark of Carnegie-Mellon University.

Accent and may of its subsystems and support programs were originally developed by the CMU Computer Science Department as part of its Spice Project.

The information in this document is subject to change without notice and should not be construed as a commitment by PERQ Systems Corporation. The company assumes no responsibility for any errors that may appear in this document.

PERQ Systems Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

PERQ, PERQ2, LINQ, and Qnix are trademarks of PERQ Systems Corporation.

REGISTER ALLOCATION

All numbers are in Octal.

0 .. 77 Language Interpreter Registers

Used by each language interpreter. A subset of these registers (0.. <NumRegs>-1) is saved and restored on process swap. NumRegs is part of the microkernel register set. It must be set to a multiple of 4.

{Warning - the language interpreters may change NumRegs at unpredictable moments}

100 .. 137 Language Interpreter Registers

These registers are not saved across process swap.

140 .. 177 Unused - reserved for User Microcode

These registers are not used or changed by any of the Accent microcode. They are available for user microcode use. However, they are not saved across process swap.

200 .. 277 IO Microcode Registers

300 .. 377 MicroKernel and Virtual Address Translation Registers

APPENDIX B
MICROSTORE ALLOCATION

December 7, 1984

2600 Liberty Avenue
P. O. Box 2600
Pittsburgh, PA 15230
(412) 355-0900

Accent is a trademark of Carnegie-Mellon University.

Accent and may of its subsystems and support programs were originally developed by the CMU Computer Science Department as part of its Spice Project.

The information in this document is subject to change without notice and should not be construed as a commitment by PERQ Systems Corporation. The company assumes no responsibility for any errors that may appear in this document.

PERQ Systems Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

PERQ, PERQ2, LINQ, and Qnix are trademarks of PERQ Systems Corporation.

MICROSTORE ALLOCATION

All numbers are in Octal.

Bank 0:

0 .. 3377 Pascal Interpreter

Perq.Bin

3400 .. 4177
Reserved

4200 .. 7377
IO System and Microkernel

Microkernel.Bin

7400 .. 7777
Overlay Area for Microcode

These overlays are built into the Accent Boot file. There are two overlays in use: one holds the system initialization microcode, and the other the Pascal Interpreter microcode to handle Sets.

KernelInit.Bin (system initialization)

Sets.Bin (set microcode)

Bank 1:

10000 .. 14777
Pascal Interpreter, Bank 1

Perq.High.Bin

15000 .. 15777
VmVp.Bin

Virtual memory microcode

16000 .. 17777
Reserved

Banks 2 and 3:

20000 .. 37777
Lisp Interpreter
or user microcode*

* It is suggested that user microcode be placed in Bank 3 (30000 .. 37777) first before using Bank 2.

INDEX

KEY TO ABBREVIATIONS - MICROPROGRAMMER'S MANUAL

MC - Microprogrammer's Reference
PC - Pascal / C Machine Reference
RG - Register Allocation
MS - Microstore Allocation