PERQ Micro-Programmer's Guide

Brian Rosen

John P. Strait

This manual provides an introduction to the PERQ micro-programmable CPU.

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

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

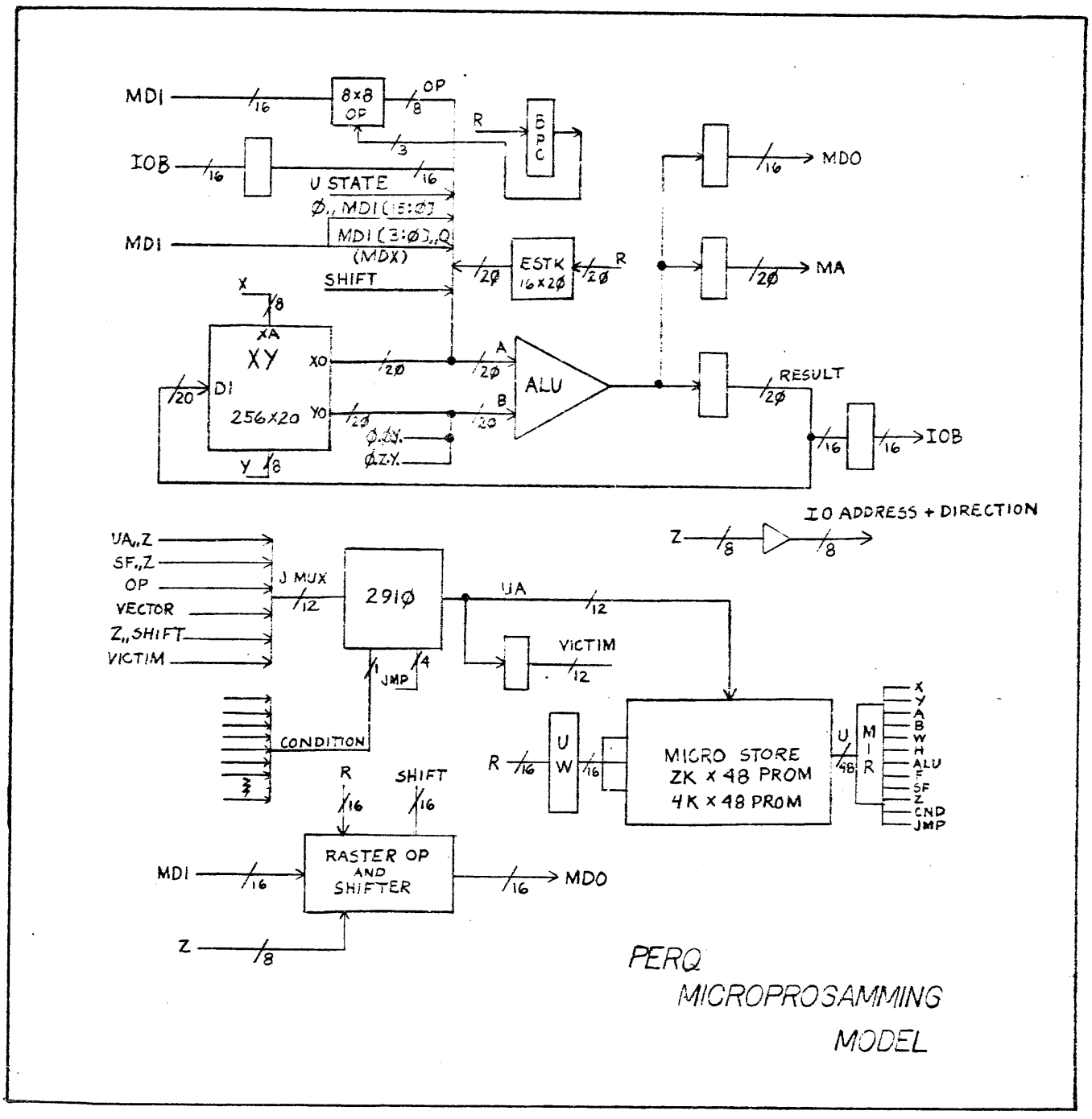
Three Rivers Computer Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

Table of Contents

Introduction to the Hardware

PERQ is implemented with a high-speed microprogrammed processor which has a 170 nanoseconds microcycle time. The microinstruction is 48 bits wide. Most of the data paths in the micro engine are 20 bits wide. The data coming in and out of the processor (IO and Memory data for instance) are 16 bits wide. 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 diagrammed below:



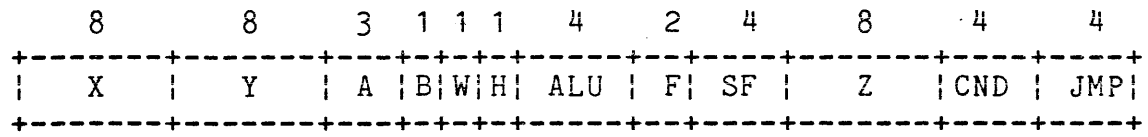
The XY registers (256 registers x 20 bits) form a double ported file of general-purpose registers. The X port outputs are multiplexed with several other sources (the AMUX) to form the A input to the ALU. The Y port outputs, multiplexed with an 8- or 16-bit constant via the BMUX, form the B input to the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

Micro Instruction Format



Field	Width	Use
X	8	Address for X port of XY. Also address used to write XY.
Y	8	Address for Y port of XY. Also low 8 bits of constant.
A	3	AMUX Select: 0 SHIFT 1 NextOp 2 IOD 3 MDI Memory Data Inputs, AMUX[19..16] := 0 AMUX[15..00] := MD[15..00] 4 MDX Memory Data Input extended AMUX[19..16] := MD[03..00] AMUX[15..00] := 0 5 USTATE 6 XY (RAM) 7 ESTK
B	1	BMUX select: 0 = XY[Y], 1 = Constant.
W	1	Write: XY[X] := R if W = 1.
H	1	Hold: If set, do not allow IO devices to access memory. Also used with JMP field to modify address inputs.
ALU	4	ALU function: 0 A 1 B 2 not A 3 not B 4 A and B 5 A and not B 6 A and 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

F 2 Function: Controls usage of SF and Z fields.

F	SF use	Z use
0	Special Func.	Constant/Short Jump
1	Memory Control	Short Jump
2	Special Func.	Shift Control
3	Long Jump	Long Jump

SF 4 Special Function: Upper 4 bits of address for long jump and memory control functions (see F). When used as Special Function:

0	Long Constant
1	ShiftOnR
2	StackReset
3	TOS := (R) (Top Of ESTK)
4	Push (ESTK)
5	Pop (ESTK)
6	CntlRasterOp := (Z)
7	SrcRasterOp := (R)
10	DstRasterOp := (R)
11	WidthRasterOp := (R)
12	LoadOp (OP := MDI)
13	BPC := (R)
14	WCS[15..00] := (R)
15	WCS[31..16] := (R)
16	WCS[47..32] := (R)
17	IOB Function

Z 8 Low 8 bits of Jump Address, high 8 bits of Constant, Shift Control (see F), or IOB address.

CND 4 Condition (what to test during conditional jump):

0	True - always jump
1	False - never jump
2	IntrPend - interrupts pending
3	unused
4	BPC[3] - OP file is empty
5	C19 - no carry out of bit 19 of the ALU
6	Odd - ALU bit 0
7	ByteSign - ALU bit 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
15	Eql - equal to
16	Gtr - greater than
17	Geq - greater than or equal to

JMP 4 Jump Control: See AMD 2910 documentation for further details.

CIA = Current Instruction Address.
NIA = Next Instruction Address.
Addr = SF,,Z (Long) or CIA,,Z (Short).
ZAddr = Z(UpperBits),,0,,Z(LowerBits).
S = internal address register.
CSTK = top of five level call stack.
Push = push CIA+1 onto call stack.
Pop = pop call stack.

Code	Name	Pass	Fail
0	JumpZero	NIA:=0	NIA:=0
1	Call	NIA:=Addr Push	NIA:=CIA+1
2	NextInst/ReviveVictim		
	H = 0	NIA:=OP +ZAddr	NIA:=OP +ZAddr
	H = 1	NIA:=Victim	NIA:=Victim
3	GoTo	NIA:=Addr	NIA:=CIA+1
4	PushLoad	NIA:=CIA+1 Push S:=Addr	NIA:=CIA+1 Push
5	CallS	NIA:=Addr Push	NIA:=S Push
6	Vector/Dispatch		
	H = 0	NIA:=Vector +ZAddr	NIA:=CIA+1
	H = 1	NIA:=Dispatch +ZAddr	NIA:=CIA+1
7	GotoS	NIA:=Addr	NIA:=S
10	RepeatLoop		
	if S <> 0	NIA:=CSTK S:=S-1	NIA:=CSTK S:=S-1
	if S = 0	NIA:=CIA+1 Pop	NIA:=CIA+1 Pop
11	Repeat		
	if S <> 0	NIA:=Addr S:=S-1	NIA:=Addr S:=S-1
	if S = 0	NIA:=CIA+1	NIA:=CIA+1
12	Return	NIA:=CSTK Pop	NIA:=CIA+1

13	JumpPop	NIA:=Addr Pop	NIA:=CIA+1
14	LoadS	NIA:=CIA+1 S:=Addr	NIA:=CIA+1 S:=Addr
15	Loop	NIA:=CSTK Pop	NIA:=CIA+1
16	Next	NIA:=CIA+1	NIA:=CIA+1
17	ThreeWayBranch		
	if S <> 0	NIA:=CIA+1 Pop S:=S-1	NIA:=CSTK S:=S-1
	if S = 0	NIA:=CIA+1 Pop	NIA:=Addr Pop

The Z field is used for many things: as part of a jump address, the upper 8 bits of a constant, Shift Control, and as an IOB address. The F field decodes 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 3 selects a long constant using the Z field. When F <> 2, the Z field is used for a jump address. When SF = 17 and F = 0 or 3, 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, will flag an error if the programmer tries to load two different values into the same microinstruction field.

Constants

Constants can be 8 or 16 bits wide. If B = 1, the B input to the ALU is a constant. If F = 0 or 3 and SF = 0, the Y and Z fields form a 16 bit constant. If SF <> 0 and F <> 0 or 3 then Y is an 8 bit constant.

OldCarry

OldCarry (in ALU functions 15 and 17) is the carry from the immediately preceeding microinstruction, it is used for multiple precision arithmetic.

Condition Codes

All ALU related condition codes test the result of the ALU operation from the previous micro cycle. Thus the normal sequence is to perform an ALU operation and test its result in the next microinstruction. For example, comparison of two registers A and B could be done this way:

```
A - B;
if Gtr Goto(Label);    ! Jumps if A > B
```

All ALU tests with the exception of C19 test the lower 16 bits of the ALU. These are intended for data comparisons. After a subtraction, these condition codes compare the two operands. After other operations, these condition codes compare the 16-bit ALU result against zero.

C19 is designed for unsigned address comparisons. Assuming that A and B are registers containing 20-bit addresses and T is a temporary register, the following code fragments show how to compare A and B.

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

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

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

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

Memory Control

The memory system cycles in 680 ns (exactly 4 microcycles). Microcycles are numbered starting at 0 (t0, t1, t2 and t3). Requests must be made in a particular cycle (which cycle depends on the type of request). If a memory request is made in the wrong cycle, the processor will be suspended until the correct cycle. In some contexts, however, a request made in an improper cycle will be ignored--these contexts are explained below. There are 8 types of memory references, coded into the SF field when F = 1.

SF	Type	Description
16	Fetch	Fetch 1 word from Memory.
17	Store	Store 1 word into Memory
12	Fetch4	Fetch 4 words (0 mod 4 address)
13	Store4	Store 4 words (0 mod 4 address)
10	Fetch4R	Fetch 4 words, transport in reverse order
11	Store4R	Store 4 words, transport in reverse order
14	Fetch2	Fetch 2 words (0 mod 2 address)
15	Store2	Store 2 words (0 mod 2 address)

In the following discussion of the memory controller, the terms "Fetch" and "Store" are used for the memory functions which fetch or store exactly one word. The generic terms "fetch type" and "store type" are used for any type of fetching or storing memory reference.

The address for all memory references comes from R. For all fetch type references, the address (and the request itself) are latched at t3 and data is available from MDI or MDX (A = 3 or 4) at t2. If MDI or MDX is used during a t0 or t1 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 address for a Fetch2 must be even (double-word aligned). If it is odd, the low-order bit of the address is ignored. After a Fetch2, the first word must be read at t2, and the second word must be read at t3.

The address for a Fetch4 must be quad-word aligned. If it is not quad-word aligned, the two low-order bits are ignored. After the Fetch4, the first word must be read at t2, and the succeeding words must be read at t3, t0, and t1.

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 address for a Store2 must be even (double-word aligned). If it is odd, the low-order bit of the address is ignored. The address and Store2 are given in a t3 cycle, and the data is supplied on R in the following t0 and t1.

The address for a Store4 must be quad-word aligned. If it is not quad-word aligned, the two low-order bits are ignored. The address and Store4 are given in a t3 cycle, and the data is supplied in the next four cycles (t0, t1, t2 and t3).

The Fetch4R and Store4R types are identical to the Fetch4 and Store4 references except that word 3 of the quad word is received or sent first and word 0 last. (This is generally only useful for RasterOp so that it can do left to right as well as right to left transfers.)

Here are examples of each type of reference and how they are coded:

```
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. See the high performance rules below.

Combinations of memory references are tricky. There are a few rules which, if followed, will never let you go wrong, but they may preclude some clever twist of microcoding to save some cycles. The simple rules are:

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

2) Never start a memory reference during the four instructions which follow a store type request.

The full rules are complicated, but to achieve high-performance you need to consider them. The following rules define the way the memory controller treats memory requests.

1) After a Fetch or Fetch2 (in t3), any memory reference in t0 or t1 is ignored. A Store specified in the t3 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 t1, but 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.

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 type 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)
MDO := 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.

Opcodes and operands

The OP file contains a 4-word sequence of instruction bytes. A quad-word address is contained in a XY register (UPC), and the 8 bytes pointed to by UPC are loaded into the OP file. The lower 3 bits of the byte address (byte within the quad word) are kept in BPC, a hardware register. BPC addresses the OP file to choose a byte. BPC is actually a 4-bit counter. It is incremented after each a byte is taken out of the OP file by NextInst (JMP=6, H=0) or NextOp (A=1). The 4th bit of BPC (BPC[3]), which is the "overflow" of the counter, is testable via a jump condition and indicates that all bytes in OP have been used.

The NextOp function (A=1) gets the next byte out of the instruction byte stream for use as an operand. The assembler automatically adds an "If BPC[3] GoTo(Refill)" jump clause. If BPC overflows, then control will go to Refill which increments UPC by 4, set BPC to 0, and starts a Fetch4 to the OP file. The special function LoadOp must be executed in the t1 after the Fetch4 to cause the Op file to be loaded with the data coming on MDI. Refill must then jump back to the instruction which needed the byte so that instruction may be re-executed. The instruction which executes NextOp must be capable of being executed twice (once when BPC overflowed and again when it is re-executed after Refill). This precludes instructions such as "R := NextOp + R".

In order for Refill to get back to the instruction which needs to be re-executed, the address of the failed NextOp is saved in a hardware register (Victim) if NextOp is executed when BCP[3] is set. The last instruction in Refill is coded with ReviveVictim (JMP=2, H=1), which sends control back to the "failed" NextOp.

BPC can be set without re-loading the OP file, and so the current quad word can be re-read without fetching it from memory a second time.

The NextInst jump enables a byte of the OP file (which is inverted for NextInst) into the Addr input of the micro-sequencer. The inverted byte is shifted left by 2 bits and OR-ed with ZAddr, sending control to address ZAddr + (OP' * 4). If BPC[3] is true, OP is forced to 377, sending control to location ZAddr, which is another version of Refill. This version of Refill also does the Fetch4 to the OP file, zeroes BPC, increments UPC by 4, and does the LoadOp, but then repeats the NextInst instead of returning via ReviveVictim.

To speed up the execution of Refill, the LoadOp Special Function loads all 4 words via hardware. The LoadOp should be given in the t1 following the Fetch4. The instruction which follows the LoadOp can go back to the NextInst/NextOp since the first byte is quaranteed to be in. The three remaining words arrive and are placed in OP by hardware without further microcode assistance. If BPC is set to a non-zero value (to start reading in the middle of the quad word), the Refill code must wait until the correct byte is in the OP file.

Shift Control

The PERQ shifter can rotate a 16 bit item 0 to 15 places and apply a mask to the shifter outputs. To use the shift hardware, the Z field of the instruction can be coded with the type of shift to be done with the F field set to F = 2. Coding of the shift control uses two 4 bit nibbles (shift control is inverted in the Z field):

Shift Control	Shift
0-17,0	1 bit field starting at bit 0-15
0-16,1	2 bit field starting at bit 0-14
0-15,2	3 bit field starting at bit 0-13
...	
0-2,15	14 bit field starting at bit 0-2
0-1,16	15 bit field starting at bit 0-1
0-17,17	Left shift 0-15
10-17,16	Rotate Right 8-15
10-17,15	Rotate Right 0-7
0-17,17-0	RightShift 0-15

The item to be shifted is placed on R, and the shifted and masked result can be read via SHIFT (A = 0) on the next instruction. The shift control logic keeps the last value 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.

ShiftOnR

The ShiftOnR special function allows a shift function to be a variable. The shift control is obtained from the R bus and thus can be a data item. The usage sequence is: 1) Put the shift control (univerted)
on R and execute ShiftOnR, 2) Put the item to be shifted on R, and 3) Read the shifted result on SHIFT.

Expression Stack

The expression stack is used to evaluate expressions. Items are pushed on the stack by placing them on R and using the Push special function: "TOS := Data, Push". Items can be popped off the stack with the Pop special function. The top of the stack can be written without pushing or popping with the "TOS := Data" special function. The value on the top of the stack can be read at any time from TOS (A = 7). The stack is 16 levels deep. The stack can be reset (no items on the stack) by the StackReset special function. Stack empty can be read as a bit in USTATE.

Input/Output Bus

IOB is the input/output bus for PERQ. The IOB is a 16-bit bi-directional data bus plus a 7-bit address bus. The addresses are supplied on the Z Field. The eighth bit of the Z field indicates the direction of transfer (1=write, 0=read). To read an IO register, set SF = 1 and F = 0 or 3. The IO register is latched in the processor such that a succeeding microinstruction can read it from IOD (A = 2). IO registers can be written by putting a data value on R, putting the appropriate address in Z, and coding the IOB special function (SF = 1, F = 0 or 3).

Jumps

A jump needing an address normally gets it from the Z field. Since Z is only 8 bits wide and the control store is 4K, another 4 bits of address are needed. Short jumps branch to a location on the same 256-word page as the current microinstruction (CIA). To go to an arbitrary location, the F field can specify long jump (F = 3) which uses the SF field for the upper 4 bits of address.

The address for jumps might not come from the Z (and SF). Other sources for jump addresses are: 1) The S register (which is internal to the micro-sequencer), 2) A five level call stack (also internal to the micro-sequencer) which is pushed for a Call and popped for a Return, 3) The current instruction address plus 1, and 4) The Victim register.

There are three jumps which are multi-way branches. The three are: 1) NextInst, which is a 256-way branch based on a byte from the OP file; 2) Dispatch, which is a 16 way (or fewer) branch on the lower 4 bits of the SHIFT outputs; and 3) Vector, which branches to 1 of 8 micro-interrupt service routines. For all of these branches, the Z field of the micro-instruction supplies the other bits of the address. For NextInst, the resulting address is:

```
        0 0 0 0 0 0 0 0
Z Z P P P P P P P P Z Z
7 6 7 6 5 4 3 2 1 0 1 0
```

which results is a 256 way branch with a spacing of 4 instructions. For Dispatch, the address is:

```
Z Z Z Z Z Z S S S S Z Z
7 6 5 4 3 2 3 2 1 0 1 0
```

which also results in a spacing of 4 instructions. The Vector jump uses the outputs of the micro-interrupt priority encoder (V), which determines the highest priority micro-interrupt condition. The address is:

```
Z Z Z Z Z Z - V V V Z Z
7 6 5 4 3 2 0 2 1 0 1 0
```

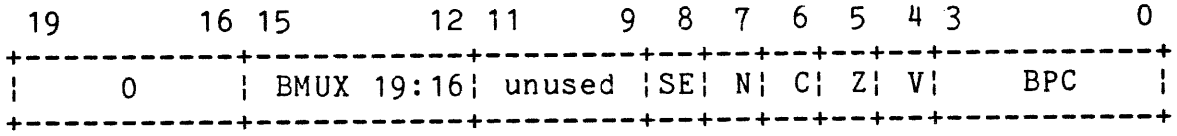
which also has a spacing of four instructions.

Interrupts

The hardware implements a microlevel interrupt which is used to allow the microprocessor to serve IO devices. There are (a maximum of) 8 interrupt requests which are assigned priorities by the hardware. When any of the interrupt requests is asserted, the branch condition IntrPend will succeed. The intended usage of this feature is that at convenient places in the microcode an instruction which has "If IntrPend Call(VecSrv)" is used. If any interrupts are pending, control will pass to VecSrv which should contain a Vector jump to send control to ZAddr + Vector*4 in the control store. The address of the interrupted instruction is on the call stack, and the interrupt microcode can serve the device and return like a subroutine would.

USTATE

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




```
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
             a register (inverted data).  Do it this way:
                not UState(Register), Field(14,4);
                Result := Shift;
                ! Result[3:0] := Register[19:16]
```

Syntax of Micro-Programs

This describes the syntax of Perq micro-programs recognized by the PrqMic assembler. 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.
(     )     - are used for grouping.
  .         - ends a description.
```

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

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

constant        = name ¦ number .

register        = name .

label           = name .

empty   = .


MicroProgram    = {Instruction ';'} 'end' ';' .

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

    Field   = empty
            ¦ Pseudo
            ¦ PascalStyleConstant
            ¦ {Result ':='} ALU
            ¦ Jump
            ¦ Special .

        Pseudo  = 'Define' '(' name ',' constant ')'
                ¦ 'Constant' '(' name ',' constant ')'
                ¦ 'Opcode' '(' constant ')'
                ¦ 'Loc' '(' constant ')'
                ¦ 'Binary'
                ¦ 'Octal'
                ¦ 'Decimal'
                ¦ 'Nop'
                ¦ 'Case' '(' constant ',' constant ')'
                ¦ 'Place' '(' constant ',' constant ')' .

        PascalStyleConstant = name '=' constant .
```

```
Result  = register | SpecialResult .

    SpecialResult   = 'TOS' | 'MA' | 'MDO' | 'BPC'
                    | 'SrcRasterOp' | 'DstRasterOp'
                    | 'WidRasterOp' .

ALU     = ['not'] (Amux | Bmux)
        | Amux Op ['not'] Bmux
        | Amux '+' Bmux ['+' 'OldCarry']
        | Amux '-' Bmux ['-' 'OldCarry']
        | Amux 'amux' Bmux
        | Amux 'bmux` Bmux .

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

    Bmux    = register | constant .

    Op      = 'and' | 'or' | 'xor' | 'nand' | 'nor'
            | 'xnor' .

Jump    = [Test] Goto ['(' Addr ')'] .

    Test    = 'If' Condition .

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

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

    Addr    = label | constant .

Special = Nonary | Unary | Binary .

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

    Unary   = UnaryName '(' constant ')' .

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

```
Binary  = BinaryName '(' constant ',' constant ')' .

BinaryName  = 'Field' .
```

Notes on the Syntax

1) Programs are typed in free format. That is, a single micro-instruction may extend to as many lines as desired. Blank lines and lines consisting only of comments may be inserted anywhere. The exception to this rule is that a new micro-instruction begins with a new line, i.e. you may not place more than one instruction per line.

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

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

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

5) Constants may be defined Pascal style:

```
name = value;
```

This allows including a file which contains constant definitions into both a Pascal program and a micro-program.

6) The following ALU functions are allowed by the syntax description, but do not exist. The assembler disallows them.

```
Amux 'nand' 'not' Bmux
Amux 'nor' 'not' Bmux
Amux 'xor' 'not' Bmux     (equivalent to Amux 'xnor' Bmux)
Amux 'xnor' 'not' Bmux    (equivalent to Amuv 'xor' Bmux)
```

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

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

TOS := BPC := 10;         is invalid.
```

The 'Micro Instruction Format' section shows which fields of a micro-instruction 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 may not be used in a single micro-instruction.

8) 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. This table gives the rules.

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

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

Assembler Commands

There are several commands which are directives to the assembler and placer to perform special actions. Assembler commands are indicated by a '$' in column 1. The entire line is considered to be an assembler command. You may not type other micro-instructions on the same line.

'$Include' FileName

The text in the named file is inserted into the microprogram as though it were present in the original source file. Included files may not be nested. Only the orginal source file may contain an Include command.

'$Title' TitleString

The TitleString is printed on the assembly listing on the first line of every page. 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.

'$NoList'

The listing is turned off until a List command is encountered. This command has an effect only if a listing is requested when the placer is executed.

'$List' The listing is resumed if it was turned off by a NoList command. This command has an effect only if a listing is requested when the placer is executed.

Assembly Instructions

Before a micro-program 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 micro-store locations to those instructions which are not assigned by the micro-programmer.

The following shows how to assemble and place a micro-program. A micro-program source file name has the form <src>.MICRO. <src> is called the root name.

assemble:

```
PrqMic
Root file name? <src>

        -or-

PrqMic <src>
```

place with listing:

```
PrqPlace
Root file name? <src>
List file name? <lst>

        -or-

PrqPlace <src> <lst>
```

place without listing:

```
PrqPlace
Root file name? <src>
List file name?

        -or-

PrqPlace <src>
```

Once a micro-program has been assembled and placed, you can use it in one of these ways:

a. Load it into another Perq with OdtPrq.

b. Load it into the same Perq with the ControlStore module.

c. Write it into a boot file with MakeBoot.

Writable Control Store (WCS) Map

The following provides a map of the Writable Control Store (WCS) and describes the micro code register usage.

When you boot the system, VFY.MICRO runs memory diagnostics. SYSB.MICRO then loads microcode from the MBoot file. At boot time, microcode can only be loaded into WCS addresses 0 through 7377. At the end of the boot, only PERQ.MICRO and IO.MICRO are in the WCS. This leaves 7000 through 7777 for user defined microcode. (You can use 7000 through 7377 for bootable special purpose microcode.)

PERQ.MICRO -PERQ Q-Code interpreter microcode. Temporary registers store state information during Q-Code interpretation. Registers: 3-21, 51-57, 64-67, 370. Temporary registers: 30-50, 61-63, 70-77. Placement: 0-4377.

IO.MICRO - Input/Output microcode. Registers: 200-207, 211-217, 221-233, 235-252, 255-260, 262-266, 276, 277, 327, 373, 374. Temporary registers: 220, 261. Placement: 4400-5777.

LINK.MICRO - 16 bit parallel interface microcode. (Not normally booted into WCS). Registers: 350, 351. Placement: 6744-7400.

SYSB.MICRO - system boot microcode. Registers: 0-64. Placement: 7000-7777.

IOE3.MICRO - Microcode for 3MBaud ETHERNET. This area is available if the system will not use ETHERNET. Registers: 270-274. Placement: part of IO.MICRO.

BOOT.MICRO - Boot microcode. Registers: 0-2, 4, 10, 20, 40, 100, 200, 252, 277, 307, 337, 350, 357, 367, 373, 375, 376, 377. Placement: 0-777.

KRNL.MICRO - PERQ microcode kernel. Registers: 0, 357-377. Placement: 7400-7777.

GOODBY.MICRO - Power down microcode. Placement: 5000-5377.

ETHER10.MICRO - 10MBaud ETHERNET microcode. Registers: 300-321. Placement: 7000-7300

RO.MICRO - Raster-op microcode. Registers: 100-124, 130-142, 370. Placement: part of PERQ.MICRO.

LINE.MICRO - Line drawing microcode. Registers: 100, 101, 103-116. Placement: part of PERQ.MICRO.

VFY.MICRO - Verifies that the hardware seems to work. Registers: 0-16, 300, 301, 370. Placement: 4000-6377.

Quirks

As of 12/1/80, the following quirks are known:

- The Z field is inverted for Shift functions (assembler fixes this).
- The Op file is inverted on NextInst (assembler Opcode does it).
- The Z field is inverted for all addresses (assembler fixes it).
- IOB functions are executed twice if an abort occurs.
- C19 will not be valid if an abort occurs on the test.
- C19 test is inverted sense (i.e. jump if no carry out of bit 19).
- UState[15:12] (the upper BMux bits) are inverted.
- Condition codes are not quite right after double precision adds and subtracts. See the `Condition Codes` section.
- Condition codes are invalid after ReadProduct, ReadQuotient, and ReadVictim.