COMMERCIAL-IN CONFIDENCE

PERQ Microprogrammers Guide

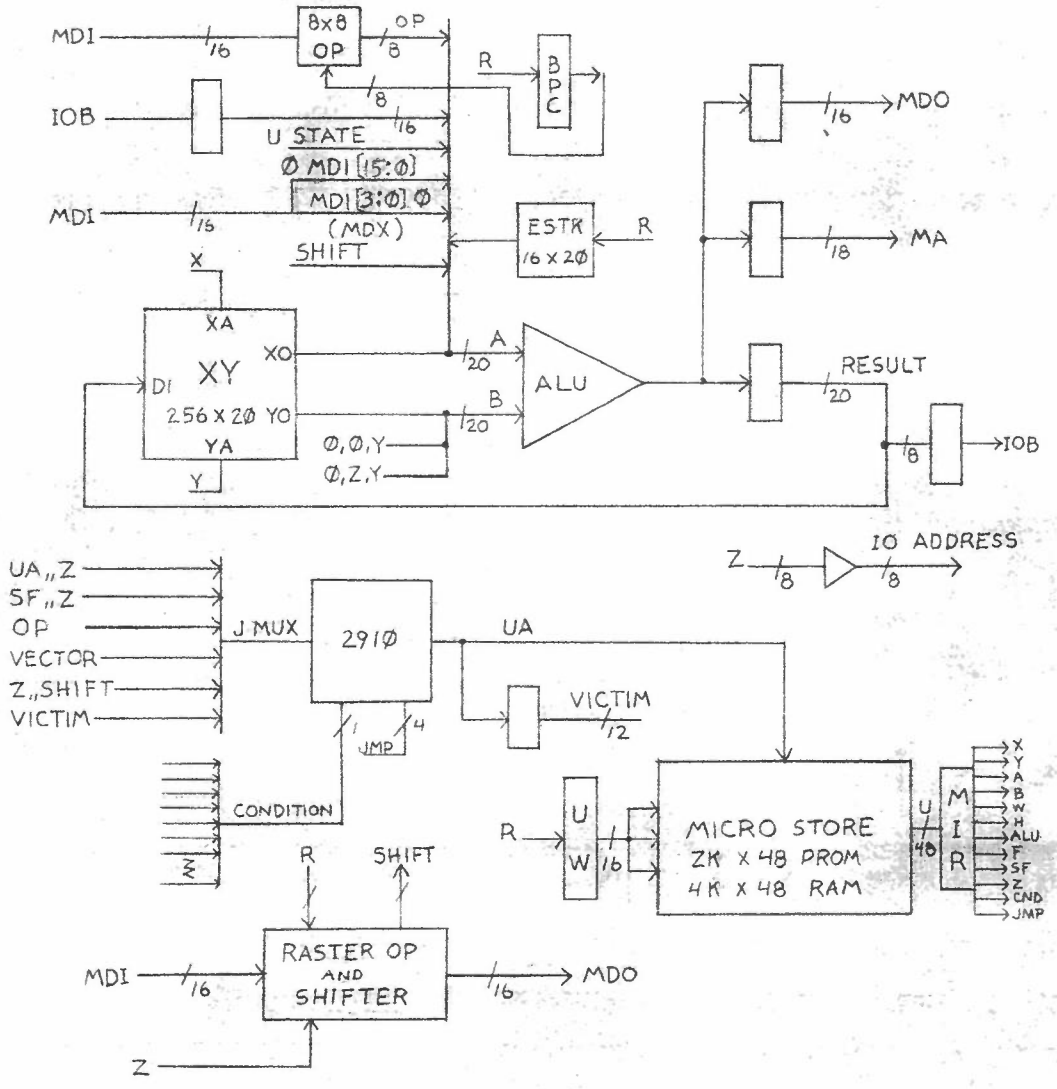
Brian Rosen

Three Rivers Computer Corporation

November 12, 1979

Three Rivers Computer Corporation
160 North Craig Street
Pittsburgh, PA 15213
(412) 621-6250

The PERQ is implemented with a high speed microprogrammed processor capable of executing a microinstruction in 170 ns. The microinstruction is 48 bits wide. The width of most of the data paths in the micro engine is 20 bits. The data coming in and out of the processor (IO and Memory data for instance) is 16 bits. The extra 4 bits allow the microprogrammed processor to calculate real addresses in a 1 megaword addressing space. The assumption is that real addresses are kept in a doubleword in memory but calculations on addresses can be single precision within the processor. The PASCAL programmer never sees the 20 bit paths. The major data paths are diagrammed below:



PERQ MICRO PROGRAMMING MODEL

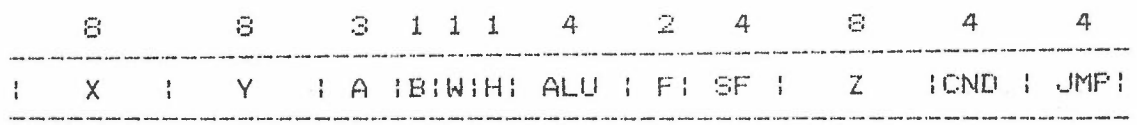
The XY registers (256 registers x 20 bits) form a double ported file of general purpose registers. The X port outputs are multiplexed with several other sources (the AMUX) to form the A input to the ALU. The Y port outputs, multiplexed with an 8 or 16 bit CONSTant via the BMUX forms the B input to the ALU. The ALU outputs (R) are fed back to the XY registers as well as the MemoryData output and MemoryAddress registers. Memory Data coming from the memory is sent to the ALU via the AMUX. A 16 bit I/O Bus (IOB) 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 (OP). OP is written 16 bits at a time from the MD inputs. The output of OP is 8 bits wide and is read via AMUX and can be sent to the micro-addressing section for opcode dispatch. The read port of OP 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.

The Micro Instruction:



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 A' 3 B' 4 A and B 5 A and B' 6 A nand B 7 A or B 8 A or B' 9 A nor B 10 A xor B 11 A xnor B 12 A + B 13 A + B + OldCarry 14 A - B 15 A - B - OldCarry

F 2 Function, controls usage of SF and Z fields

E	SE 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 Func:

0	NOP
1	ShiftOnR
2	StackReset
3	TOS := (R) (Top Of ESTK)
4	Push (ESTK)
5	Pop (ESTK)
6	Cnt1RasterOp := (R)
7	SrcRasterOp := (R)
8	DstRasterOp := (R)
9	WidthRasterOp := (R)
10	LoadOp (OP := MDI)
11	BPC := (R)
12	WCS[15..00] := (R)
13	WCS[31..16] := (R)
14	WCS[47..32] := (R)
15	IOB Function

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

CND 4 Condition. What to test for conditional jump.

0	True (always jump)
1	False (never jump)
2	BPC[3] - if set, need to refill OP
3	C19 - carry out of msb of data path
4	InterruptsPending
5	A[0] - odd/even
6	A[7] - byte sign
7	A[15] - sign bit
10	Eql
11	Neq
12	Gtr
13	Geq
14	Lss
15	Leq
16	Carry (out of bit 15)
17	Overflow (of bit 15)

JMP 4 Jump Control. See 2910 documentation for further details. CIA = Current Instruction Address
NIA = Next Instruction Address
Addr = SF,,Z (Long) or CIA,,Z (Short)
ZFill = Z(UpperBits),,0,,Z(LowerBits)

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 +ZFILL	NIA:=OP +ZFILL
	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 +ZFill	NIA:=CIA+1
	H = 1	NIA:=Dispatch +ZFill	NIA:=CIA+1
7	GotoS	NIA:=Addr	NIA:=S
8	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
9	Repeat		
	if S <> 0	NIA:=Addr S:=S-1	NIA:=Addr S:=S-1
	if S = 0	NIA:=CIA+1 Pop	NIA:=CIA+1 Pop
10	Return	NIA:=CSTK Pop	NIA:=CIA+1
11	JumpPop	NIA:=Addr Pop	NIA:=CIA+1
12	LoadS	NIA:=CIA+1 S:=Addr	NIA:=CIA+1 S:=Addr
13	Loop	NIA:=CSTK Pop	NIA:=CIA+1

14	Next	NIA:=CIA+1	NIA:=CIA+1
15	ThreeWayBranch		
	if S <> 0	NIA:=CIA+1	NIA:=CSTK
		Pop	
		S:=S-1	S:=S-1
	if S = 0	NIA:=CIA+1	NIA:=Addr
		Pop	Pop

Constants can be 8 or 16 bits. If BMUX = 1, F = 0 and SF = 0, the Y and Z fields form a 16 bit constant. If BMUX = 1 and either F or SF <> 0 then Y is an 8 bit constant.

OldCarry (in ALU functions 13 and 15) is the carry from the immediately preceeding microinstruction, it is used for multiple precision arithmetic.

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, when SF and F = 0, the constant will be 16 bits (B = 1). If B <> 1 then the Z field is free, and can be used for Jump or IOB addresses. When F = 3, 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. There is nothing that prevents the processor from using the Z field for both a constant (if F andSF = 0 and B = 1) and a Jump address in the same instruction. This also applies to Z used as an IOB address. The assembler will flag questionable Z field usages.

Memory Control. The memory system cycles in 680 ns (exactly 4 microcycles). Microcycles are numbered starting at 0 (t0, t1, t2 and t3, plus t4, t5 and t6 which overlap the following t0, t1 and t2 respectively). Requests must be made on a particular cycle (which cycle depends on the type of request). If a memory request (Fetch or Store) is made on the wrong cycle, the processor will be suspended until the correct cycle. 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)

The address for all memory references comes from R. On a Fetch type reference, the address (and the request itself) are latched at t0 and data is available as an AMUX source at t3. If AMUX = 3 or 4 (MDI or MDX) during a t1 or t2 following a fetch type memory reference, the processor is suspended until t3. The memory data will remain available until the next memory reference's t2 (exception: see the note about IO memory references). It is possible to read the (same) data several times, but not after the next reference reaches t2.

For Fetch4 references, the first word is available at t3, and the succeeding words arrive at t4, t5 and t6. In this case, the processor must read MD during t3 - t6; the data does not wait for you to read it. However, attempting to read the first word during t1 or t2 will cause suspension.

For a Store reference, the address and store command is given in the t-1 cycle (t3 of preceeding cycle) and the data to be written is supplied (on Result) in the t0 cycle following the Store command.

In a Store4 type reference, the command and address is given in cycle t0, and the data is supplied in the next four cycles (t1, t2, t3 and t4). Fetch4 and Store4 types of references use two memory cycles (1.2 usec) since another memory request is not allowed during the t4 cycle (a t0, at least potentially) because the microinstruction cannot specify both an address and a data word in the same microword. (This is not quite true during RasterOp, where the hardware can send data through the RasterOp data paths at the same time the processor is generating addresses.)

The Fetch4R and Store4R types are identical to the Fetch4 and Store4 references except that word 3 of the quad word is received/sent from/to the memory 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 (TheAddr, TheData, QuadAddr, and Data0-3 are XY registers):

```
FetchOne:       MA := TheAddr, Fetch;    (t0)
                ...                      (t1)
                ...                      (t2)
                TheData := MDI;          (t3)

FetchFour:      MA := QuadAddr, Fetch4;  (t0)
                ...                      (t1)
                ...                      (t2)
                Data0 := MDI;            (t3)
                Data1 := MDI;            (t4)
                Data2 := MDI;            (t5)
                Data3 := MDI;            (t6)

StoreOne:       MA := TheAddr, Store;    (t7 = t3 = t-1)
                MDO := TheData;          (t0)
                ....

StoreFour:      MA := QuadAddr, Store4;  (t0)
                MDO := Data0;            (t1)
                MDO := Data1;            (t2)
                MDO := Data2;            (t3)
                MDO := Data3;            (t4)
```

The IO system can request memory cycles at any time. The memory system gives priority to the IO system so that if the processor and the IO system request cycles, the IO will get it. The Hold bit, if set, locks out IO requests while it is set. Since the IO system can get a memory cycle any time Hold is not set, there is no guarantee that the MDI data will be valid following the t2 after a fetch, even if the processor doesn't start a new cycle.

Opcodes and operands. The OP register file contains a 4 word sequence of instruction bytes. The intended usage of the OP file and the BPC register that addresses it is as follows. The quad word address of the current instruction is contained in a XY register (IPC), and the 8 bytes pointed to by IPC are stored in OP. The lower 3 bits of the IPC (which byte in a quad word) is kept in BPC, a hardware register. BPC addresses OP to choose a byte. BPC is actually a 4 bit counter. It is incremented whenever a byte is taken out of OP by NextInst (JMP=6, H=0) or NextOp (AMUX=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 (AMUX=1) gets the next byte out of the instruction byte stream for use as an operand. It is coded with an "If BPC[3] GoTo(Refill)" Jump clause. If BPC is overflowed, then control will go to Refill which increments IPC by 8 bytes and starts a Fetch4 to OP. The special function LoadOp must be executed in the t2 of the fetch to cause the Op file to be loaded with the data coming on MDI. Refill must then get back to the instruction which needed the byte. This instruction must be re-executed. The instruction which executes NextOp must be capable of being executed twice (once when BPC was overflowed, and once 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) whenever 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.

Jump and Call Q-codes calculate IPC and BPC and load OP with the appropriate quadword. They can optimize execution by checking to see if the right quadword is already in OP (new IPC = old IPC) and just load BPC.

The NextInst JMP enables OP (which is inverted) into the "Addr" input of the microinstruction sequencer shifted left by 2 bits, and ored with ZFill, sending control to address ZFill + (OP' * 4). If BPC[3] is true, OP is forced to 255, sending control to location ZFill+0, which is another version of Refill. This Refill also does the Fetch4 to OP, zeroes BPC, increments IPC, and does the LoadOp, but then repeats the instruction dispatch instead of returning via Victim.

In order to speed up the execution of Refill, the LoadOp Special Function loads all 4 words via hardware. The LoadOp should be given in the t3 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 OP by hardware without further microcode assistance. This doesn't work with Q-code Jumps since the low bits of the target for the Jump are not guaranteed to be 0.

Shift Control allows the microprogram to use the shift/mask hardware. The 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 Z field uses two 4 bit nibbles:

Z Field	Shift
0-15,0	1 bit field starting at bit 0-15
0-14,1	2 bit field starting at bit 0-14
0-13,2	3 bit field starting at bit 0-13
...	
0-2,13	14 bit field starting at bit 0-2
0-1,14	15 bit field starting at bit 0-1
0-15,15	Left shift 0 - 15
8-15,14	Rotate 8 - 15
8-15,13	Rotate 0 - 7
0-15,15-0	RightShift 0 - 15

The item to be shifted is placed on R, and the shifted and masked result can be read via AMUX = 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.

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 would be 1) put the shift control item on R and execute ShiftOnR, 2) put the item to be shifted on R, and 3) read the shifted result on SHIFT.

ESIK is used to evaluate expressions. Items to be pushed on the stack are placed on R with Special Function Push. Items can be popped off the stack with special function Pop. The Top Of the Stack can be written without pushing or popping with the TOS:= special function. TOS can be read at any time with := TOS (AMUX = 7). The stack is 16 levels deep (15 pushes) The stack can be reset (no items on the stack) by the StackReset SF. Stack empty and full can be read as condition bits in USTATE.

IOB is the Input Output bus for PERQ. The IOB is a 16 bit bidirectional bus plus a 7 bit address bus. The Addresses are supplied on the Z bus. The eighth bit indicates the direction of transfer 1=write, 0=read. There are two ways to read an IO register. One way takes a single cycle, and is used when the logic in the device can decode its address very rapidly. It is coded with an IO address in the Z field, and A=2, SF=1 with F=0 or 3. Some devices may not respond fast enough, and may require restricting ALU functions to logical (no carry) operations.

The other way to read an IO register is to code the SF=1, F=0 or 3 without coding A=2. In this case, the IO register is latched in the processor such that a succeeding microinstruction can read it with A=2 (no special function needed). IO registers can be written by putting the appropriate address in Z and coding the IOB special function (SF=1, F=0 or 3).

Consult documentation on individual I/O devices to determine which form of IOB should be used.

Short/Long Jumps. Any instruction needing an "Addr" as part of its JMP field normally gets it from the Z field. Since Z is only 8 bits long, 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 is 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 Addr for Jumps might not come from the Z (and SF); several JMP codes cause the Addr inputs to the microsequencer to come from other sources. There are three cases in which the address is a multi-way branch. The three cases are: NextInst dispatch, where a 256 way dispatch based on the opcode is done, the Dispatch JMP, which causes a 16 way (or fewer) dispatch on the lower 4 bits of the SHIFT outputs, and Vector dispatch 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. For instruction dispatch, 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 between entry points. The Dispatch JMP branches on a field selected via the shifter of up to 4 bits. 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
```

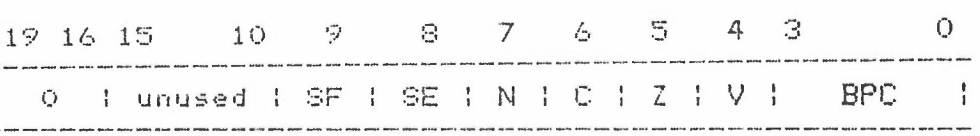
The Vector JMP dispatches on the outputs of the microinterrupt priority encoder (V), which determines the highest priority micro-interrupt condition. The address is:

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

As previously mentioned, ReviveVictim enables the Victim register into the address input of the microsequencer.

Interrupts. The hardware implements a microlevel interrupt which is used to allow the microprocessor to help IO devices. There are (a maximum of) 8 interrupt requests which are assigned priorities by the hardware into a 3 bit Vector. When any of the interrupt requests is asserted, the Branch condition InterruptsPending will succeed. The intended usage of this feature is that at convenient places in the microcode an instruction which has "If InterruptsPending Call(VecSrv)" is used. If any interrupts are pending, control will pass to VecSrv which would contain a Vector Jump field which send control to Vector*4 in the control store, with the Interrupted CIA on the stack. The Interrupt microcode can service a device, and return like a subroutine would.

USTATE. The USTATE register contains various interesting items, packed in a single word. The USTATE register (AMUX=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)
SF - ESTK Full (inverted data -- 0 = full)

Quicks. As of 10/14/79, the hardware has several shortcomings caused by some unfortunate data inversions. Most of the inversions are fixed by the assembler, some are up to the microprogrammer to fix. The inversions are:

- The IO bus WRITE data is inverted (microprogrammer beware)
- The Z field is inverted for Shift functions (Assembler does it)
- The Op file is inverted on NextInst (Assembler Opcode does it)
- The Z field is inverted for all addresses (Assembler does it)
- The A[0], A[7] and A[15] jumps are inverted sense (MP beware)
- USTATE is inverted (MP beware)
- BPC:= must have inverted data (MP beware);