Writing Custom Microcode for the iPSC/VX

David S. Scott
Intel Scientific Computers

INTRODUCTION

This technical report provides an introduction to writing custom microcode for the iPSC/VX. It is not a complete description of all of the functionality of the vector board. It does provide an overview of the techniques needed for writing microcode.

There are several possible reasons for writing custom microcode. The obvious reason is to obtain good performance for a computation which cannot take advantage of the existing vector library. Sequential operations, sequences of short vector calculations, and to a lesser extent, sparse (indirect addressed) calculations all fall into this category. For complex computations, it may be worthwhile to write microcode even if the operation does vectorize well. This is because the vector board computes faster than it touches memory. Performance can be improved by reusing operands for several operations while they are in the registers, instead of having to reread them from memory. Also, microcode can often take better advantage of the limited fast static memory available than can source code. Finally, reducing a time consuming operation to a single microcode call may make it easier to use the node processor simultaneously for some other activity. In particular, it may be easier to overlap communication with vector board computation.

Four different example codes will be discussed. A standard SAXPY routine will be discussed first. It will be used to illustrate the basic procedure for writing microcode. A routine which computes the inner product of a sparse vector with a dense vector shows how indirect addressing is implemented. A pseudo-random number generator shows how a non-vector (recursive) application can be implemented. A small dense matrix-vector product routine shows how a sequence of short vector operations can be efficiently overlapped.

MICROCODE BASICS

Microcode routines are written in an assembly language. They are assembled and linked into a microcode library. This library is used as part of the application (on the iPSC/2) and is loaded into the control store of the vector board when the node program is loaded. A standard microcode library is linked unless a different one is specified in the link command. iSC has provided an interface to the standard library which is called VECLIB. VECLIB is accessed by FORTRAN or C function and subroutine calls. To access a custom microcode routine, it is necessary to write an interface routine. How to write an interface routine is documented in an appendix.

The vector board has four functional units.

The microsequencer controls the instruction stream including counter operations. There are four 16-bit counters and a stack.

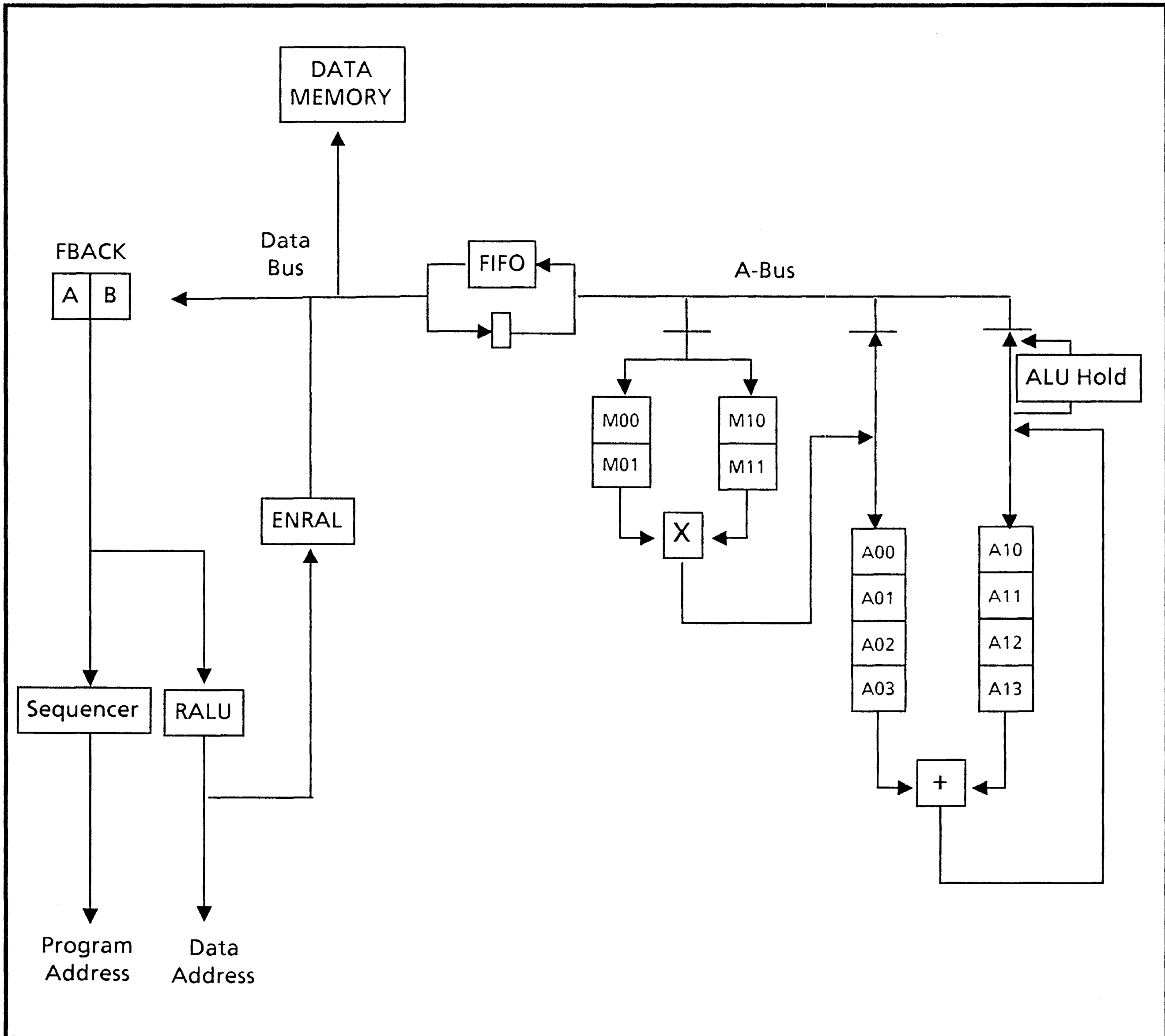
The register arithmetic unit does integer address calculations. There are thirty-two 32-bit address registers. Some of these are reserved for use by the operating system.

The multiplier does integer multiplies, single precision multiplies, and double precision multiplies. It has four 32-bit registers in two banks of two. The registers are paired to make 64-bit registers. A multiply always uses one operand from each bank.

The adder (or ALU) does a variety of unary and binary arithmetic and logical operations. It has eight 32-bit registers in two banks of four. The registers are paired to make 64-bit registers. A binary operator always uses one operand from each bank.

The output of the multiplier can be chained to the left side of the ALU. The output of the ALU can be chained to the right side of the ALU. There is a two deep FIFO on the path from the arithmetic units to memory where results can be stored until the destination address is computed. The data paths are as follows:

DATA PATHS



On each microcode instruction it is possible to do each of the following:

1. Do one address calculation.
2. Fetch or store one number (32- or 64-bit).
3. Do one sequencer operation (push, pop, write, decrement, jump).

4. Store one operand in the multiplier.
5. Store one operand in the left side of the ALU (either from memory or a multiplier result).
6. Store one operand in the right side of the ALU (either from memory or an ALU result).
7. Start one multiply. (Except that no multiply can be started on the first two cycles after a double precision multiply is started.)
8. Start one ALU operation.
9. Store one arithmetic result in the FIFO for later store to memory.
10. Use one ten bit constant in an address calculation or jump instruction.

The native cycle time for the vector board is 100 ns. Since one multiply and one add can be started in one cycle, this gives a maximum megaflop rate of 20 (in single precision). However, most of the memory on the vector board is not fast enough to keep up with the arithmetic. There is a small amount of static memory (currently 16 Kbytes of which 5 Kbytes can be used for user data storage; another 4K bytes is available as scratch space inside of microcode) from which either 32 or 64 bits can be accessed in 100 ns. The rest of the memory is dynamic and 32 bits requires 200 ns per access and 64 bits requires 250 ns per access. Keeping track of these differential cycle times is not required when microcoding. A cycle with a slow memory reference just takes longer than 100 ns. Time can be saved by reading (or writing) two 32-bit data values as one 64-bit data value provided that the two values are sequential in memory (stride 1) and that the first one has an even address.

Many operations on the board are pipelined. A fetch from memory becomes available for loading exactly two cycles after the read. All ALU operations, and 32-bit multiplies finish in three cycles. A (32-bit) integer multiplication produces a 64-bit result which must be loaded as a double precision quantity into a 64-bit register. The 32-bit result is then available in the even register. The result may not be loaded into the FIFO. A 64-bit multiply finishes in five cycles. Furthermore, after starting a 64-bit multiply, it is illegal to start another multiply until 3 cycles later. That is, a double precision multiply can be started every three cycles.

The most important constraint of the architecture is that the multiplier input port is on the same data path as the result path to the FIFO (the A-Bus). Thus it is impossible to load an input from memory into the multiplier or ALU on the same cycle as a result is stored in the FIFO.

There are also two multiplier chip bugs which must be avoided. A multiply must not be started exactly four cycles after a previous double precision multiply was started (three is OK, as is five or more). Also, an operand must not be stored into M0x four cycles after a double precision multiply is started.

MICROCODE METHODOLOGY

The first step in writing microcode is identifying the task which is to be microcoded and then identifying the inner loop. The next step is to write out the inner loop in straight line code. For example, consider the vector operation (in single precision)

$$y = a*x + y$$

where x and y are vectors and a is a scalar. This operation is performed by the veclib routine SAXPY. Assuming that the scalar **a** is already stored in the multiplier, the shortest straight line code which can perform this operation is:

1. read next x
2. wait
3. store x in the multiplier, compute a*x
4. read next y
5. wait
6. store ax in the ALU, store y in the ALU, add
7. wait
8. wait
9. store ax + y in the FIFO
10. store ax + y in memory

This code performs two floating point operations (flops) in 10 instructions. Assuming that the three memory touches are all in dynamic memory this is 2 flops in 1300 nanoseconds (~1.53 megaflops). If the data were in static memory it would be 2 flops in 1000 nanoseconds or 2 megaflops. To achieve a higher computation rate it is necessary to roll up the loop so that several pairs of data are being handled simultaneously (software pipeline). Since there are three memory touches, it is impossible to roll the loop tighter than three instructions. However, a straight roll up of the loop given above fails since instruction 1 and 4 would both map onto instruction 1 and they are both memory fetches. This can be avoided by moving the read of y up to step 3. It is also necessary to delay the store of the result to memory until the cycle which is not reading x or y. Consider the following layout (the rows are instructions and the columns indicate the pipelining).

1.	Read x			
2.		y into ALU,		store result in memory
3.	x into multiplier, compute a*x, read y,	store ax in ALU, compute ax + y	store ax + y in FIFO	

This is close, but it fails because the store into the FIFO is simultaneous with the arrival of x. To avoid this conflict it is necessary to delay the addition so that the result appears on cycle 1 (when nothing is arriving from memory). This yields the following.

1. Read x,	compute a*x	store ax in ALU compute y + ax	store result in FIFO
2.	y into ALU,		store result in memory
3. x into multiplier, read y,			

This loop is now correct. The next step to producing correct code is to write the loop in correct syntax including loop control.

The result is as follows:

```
SAXPY1:             /*  label for loop */

R3 = R3 + R4,       /*  address calculation for x */
x = MEM,            /*  fetch x from memory */
A00 = PROD -> w,    /*  store product in ALU register (left) */
w = M00 .*S. M10,   /*  start multiply */
z = A00 .+S. A11,   /*  start add */
FIFO = ALUR -> z,   /*  store result in FIFO */
DCCNTR C0;          /*  decrement loop counter */

R1 = R1 + R2,       /*  address calculation for result */
MEM = z,            /*  store result */
A11 = y;            /*  store y in ALU register (right) */

R5 = R5 + R6,       /*  address calculation to read y */
y = MEM,            /*  fetch y from memory */
M00 = x,            /*  store x in multiplier register (left) */
RDFIFO,             /*  enable FIFO on memory bus */
JDR /SIGN SAXPY1;   /*  Jump to SAXPY1 if C0 is negative */
```

The explanation of the syntax follows.

MICROCODE SYNTAX

A microcode instruction is written in a micro-assembly language. If the statement is labeled, the label appears first followed by a colon. Each operational part of an instruction is separated by commas and the instruction is terminated by a semicolon. The ordering of the operational parts is irrelevant. All characters between "/*" and "*/" are treated as comments.

ARITHMETIC

Arithmetic instructions have the form

```
var = operand1   operator   operand2
```

where

var is a user defined variable which has no semantic content. However, it is recommended that the variable be declared to be the type of the result.

operand1 is a left side register

operand2 is a right side register

operator is an allowed operator.

The multiplier registers are M00 and M01 on the left side and M10 and M11 on the right side. 64-bit register names are M00 and M10. A 32-bit multiply may be started on every cycle. No multiply can be started for two cycles after a 64-bit multiply is started. Multiplication is commutative but the use of the registers is not. The left side registers can be loaded and used on the same cycle. The right side registers must be loaded the cycle before use. Furthermore, the left side registers must not be modified on the first cycle after use (32 bits) or for the first two cycles after use (64 bits). The M10 must not be modified on the first cycle after a double precision multiply.

All operators start and end with a period. The allowed operations are

```
d  =  M00  .*D.  M10
s  =  M00  .*S.  M10
s  =  M01  .*S.  M10
s  =  M00  .*S.  M11
s  =  M01  .*S.  M11
i  =  M00  .*I.  M10
i  =  M01  .*I.  M10
i  =  M00  .*I.  M11
```

The internal spaces are NECESSARY. Note that one of the register pairs for integer multiplication is not allowed. This is a microcode constraint since only 3 bits are used to code the multiplier operation and so only eight distinct operations can be coded.

There are many ALU operations. The simplest are .+I., .+S., and .+D. The eight 32-bit ALU registers are A00, A01, A02, A03 on the left side and A10, A11, A12, A13 on the right side. They are paired to make 64-bit registers A00, A02, (left) and A10, A12 (right). All binary ALU operators use one register from each side. All ALU registers can be loaded and used in the same cycle, but the even ones (A00, A02, A10, and A12) must not be modified the cycle **after** they are used.

Some of the more esoteric operations change meaning based on whether a modifier bit is set. In addition to increasing the total number of operands available, this also makes it possible to code two different operations with the same microcode. This is useful for saving space for microcode libraries, but it is of little value for custom code.

Storing Results in Registers

The basic syntax is

```
reg  =  source  ->  var
```

where register is any of the twelve arithmetic registers or FIFO, source is either ALUR, MULT, or MEM and var is a declared variable of the appropriate type, which determines whether 32 or 64 bits are stored. The MEM source is the default and need not be explicitly written; in the special case of storage from memory the form is

```
reg = var
```

Address Calculations.

There are 32 address registers Rx where x is 0-31. The allowed forms of address calculation are

```
Rx = constant
Rx = Ry
Rx = Rx + constant
Rx = Ry + constant
Rx = Rx + Ry
Rx = Rx - Ry
Rx = Rx /2
```

Where constant is an explicit constant (≦ 10 bits). Some of the higher numbered registers have reserved values. Is is safest to use only R0-R15. By convention:

R1 is the starting address of the first operand (destination)
R2 is the stride of R1 (zero for scalar)

R3 is the starting address of the second operand
R4 is the stride of R3 (zero for scalar)

R5 is the starting address of the third operand
R6 is the stride of R5 (zero for scalar)

R7 is the starting address of the fourth operand
R8 is the stride of R7 (zero for scalar)

These registers are initialized in the prolog using data provided by the interface routine.

The syntax for using the address calculation to fetch from memory is

```
var = MEM
```

The syntax for storing to memory is

```
MEM = var
```

except that an additional command

```
RDFIFO
```

is needed on the cycle AFTER the address calculation and the MEM = var. This is because the actual store to memory is done the cycle after the address calculation and RDFIFO is needed to enable the FIFO onto the memory bus. It cannot be done automatically since the address ALU (RALU) provides an alternative source of data for storing to memory.

SAXPY EXAMPLE

In our SAXPY example, we have the problem that y is both an input and an output variable. In the call to SAXPY, y is the first operand but by the time the inner loop is reached, both R1 and R5 are pointing to y.

The full layout of the SAXPY loop indicating the software pipeline is:

```
SAXPY1:
R3 = R3 + R4,  w = M00 .*S. M10,  A00 = PROD -> w,  FIFO = ALUR -> z,
x = MEM,                          z = A00 .+S. A11,
DCCNTR C0;

               A11 = y,                             R1 = R1 + R2,
                                                    MEM = z;

R5 = R5 + R6,                                       RDFIFO,
y = MEM,
M00 = x,
JDR /DIGN SAXPY1;
```

Note that using A10 to store y would not work since the new y is stored only one cycle after the old y is used. Note also that there would be NO error message generated by the assembler if A10 had been used because there is no interline error checking.

Given a correct inner loop, it is necessary to add the appropriate start-up and tail. The tail is straightforward. At the end of the last time through the inner loop, the last operands have only made it through the first column of the software pipeline (and the next to last operands have made it through the second column, and so on). It is necessary to provide additional code after the main loop that carries the computation to the end. For the saxpy code, there are three remaining columns. This is accomplished by replicating the inner loop three times and then deleting the unneeded operations.

It is possible to write a start-up similar to the tail. That is, the first column, followed by the first two columns, etc. However, this approach requires that special code be written for $n \leqq 3$, since the calculation should not then reach the main loop.

To avoid this problem, the start-up is accomplished differently. The vector processor has a hardware write delay, WDEL, which can be used to suppress writes to memory. This allows the inner loop to be the start up. The only requirements are that:

1. WDEL is set to the correct integer
2. R1 is backed up enough times
3. All operations in the later columns have legal data each time through the loop.

The complete code is as follows.

```
name         SAXPY                      /* required documentation */
vers         1.0

#DEFINE      V$AXPY_SP    0xb4          /* declare literal number */

defcmd       P6, V$AXPY_SP
```

This declares the microcode number and which prolog (see the appendix) is used. and also simultaneously indicates the labeled start of executable code.

```
float           u, w, x, y, z               /*declare variables*/
public          V$AXPY_SP                   /*make start label
                                            public*/
extern          SZERO                       /*address of 0.0*/


SECT            PM_FUNC                     /*begin executable code*/

V$AXPY_SP:

R5 = R5,        a = MEM;                    /*read scalar a*/
R5 = R1;

R6 = R2,        M10 = a;                    /*put a in multiplier*/

PAUSE,          WDEL = 3;

R0 = SZERO, z = MEM;                        /*Read zero for dummy
                                            arguments.*/

R1 = R1 - R2;                               /*back up output pointer
                                            (first)*/

R1 = R1 - R2,                               /*back up output pointer
                                            (second)*/
M00 = z, A00 = z, A11 = z,                  /*store zeros */
w = M00 .*S. M10,                           /*start dummy multiply*/
z = A00 .+S. A11;                           /*start dummy add*/

R1 = R1 - R2;                               /*back up output pointer
                                            (third)*/

R0 = SZERO,  y = MEM;                       /*read dummy y*/

/*MAIN LOOP*/


SAXPY1:
R3 = R3 + R4, w = M00 .*S. M10, A00 = PROD -> w, FIFO = ALUR -> z,
x = MEM,                        z = A00 .+S. A11,
DCCNTR C0;

              A11 = y,                                R1 = R1 + R2,
                                                      MEM = z;
```

```
R5 = R5 + R6                                                RDFIFO,
y = MEM,
M00 = x,
JDR /DIGN SAXPY1;

/* tail */
                                                            /*first*/
              w = M00 .*S. M10,  A00 = PROD -> w, FIFO = ALUR -> z,
                                 z = A00 .+S. A11;

              A11 = y,                                      R1 = R1 + R2,
                                                            MEM = z;

                                                            RDFIFO;

                                                            /*second*/
                                 A00 = PROD -> w, FIFO = ALUR -> z,
                                 z = A00 .+S. A11;

                                                            R1 = R1 + R2,
                                                            MEM = z;

                                                            RDFIFO;

                                                            /*third*/
                                                            FIFO = ALUR -> z;

                                                            R1 = R1 + R2,
                                                            MEM = z;

                                                            RDFIFO;

RTN;
END
```

Out of dynamic memory, this SAXPY computes two flops in 600 ns of 3.33 Mflops. If x is in static memory then it is 2 flops in 500 nanoseconds, which is 4 Mflops. If y is in fast memory, it is 2/.400 or 5 Mflops. Finally, if both operands are in fast memory, it is 6.6 Mflops. The code can be made even faster if the vectors have stride one. This is because two single precision numbers can be read or written as one double precision number, provided it is to or from an EVEN address. (Double precision numbers must be properly aligned on the vector board.)

The following code implements the inner loop of SAXPY provided strides are 1 and provided the starting address of both x and y start on an even address. In general, it is the interface routine which would check these conditions. In fact, the interface routine could also check for both odd addresses and do a single calculation on the 387 and then call the vector routine with even addresses. It would also be possible to write code for x and y out of phase.

To do a double read, one just does d = MEM for a double precision variable d. A double store is harder. The two numbers to be stored must come out of the ALU on successive cycles. The special word ALUHOLD is used on the first cycle and then FIFO = ALUR -> C is used on the second cycle where C is declared COMPLEX.

x and y are declared double.
z is declared complex.
R2 is equal to 2.

As written this loop will fail if n is odd since it will do an even number of elements anyway. It will also fail if the starting addresses of x or y are NOT even. A double read of an odd address does not generate an error--it just generates an incorrect value */ result

```
SAXPY2:
R3 = R3 + R2,                          A01 = PROD -> u2, FIFO = ALUR ->z,
x = MEM,        u2 = M01 .*S. M10, w2 = A01 .+S. A11,
DCCNTR C0;

DCCNTR C0,      A10 = y,                                 R1 = R1 + R2,
                                                         MEM = z;

R5 = R5 + R2,  A00 = PROD -> u1,  ALUHOLD,               RDFIFO,
y = MEM,        w1 = A00 .+S. A10,
M00 = x,
u1 = M00 .*S. M10,
JDR /SIGN SAXPY2;
```

The inner loop of this routine runs at 5.33, 6.66, 8.88 or 13.33 Mflops, depending on whether neither, x, y, or both are in fast memory. The peak rate for the vector board is 20 Mflops obtained for double read inner products where both operands are in fast memory.

RANDOM NUMBER GENERATOR

We now look at a non-vectorizable application. The best way to compute pseudo-random numbers in the range (0,1) is to compute pseudo-random positive integers and scale them. The integers can be computed by the recurrence

$$n_{i+1} \equiv a * n_i + c \quad \mathrm{mod}\ 2^{31}$$

The constants c and a must be chosen carefully (see [Knuth...]). This is NOT a vector operation but it can be microcoded reasonably efficiently on the vector processor.

The integer multiply and integer add each take 3 cycles to complete. Fortunately, the modulo operation need not be done before the next multiply is started. thus, six cycles are needed for the inner loop. Only one cycle will be a memory touch so that the inner loop will take either 700 nanoseconds (single precision) or 750 nanoseconds (double precision). A schematic of the inner loop in single precision is as follows:

```
1.   a * n                  clear-sign-bit

2.                                             scale

3 .

4.   (a  *  n)  +  c

5.                          convert-to-float   write
```

6 .

The modulo operation is equivalent to clearing the sign bit. This can be done by the ALU as a bitwise logical-and with the mask 7FFFFFFF. The convert-to-float is an ALU operation so it must be delayed to cycle 5 since (axn) + c is also an ALU operation.

The complete inner loop is as follows:

```
         SVRAND0:
1.       M00 = ALUR -> n,       A11  =  ALUR  -> n,
         an = M00 .*I. M10,     p  =  A01  .LAND.  A11;

2.                                                  M01  =  ALUR  ->  f,
                                                    sf  =  M01  .*S.  M11;

3.       DCCNTR  C0;

4.       A00 = PROD -> an,      A12  =  ALUR  -> p
         n  =  A00  . + I.  A10;

5.                              f = .SFLTDB. A12,   R1  =  R1  +  R2,
                                                    MEM  =  sf,
                                                    FIFO  =  PROD  ->  sf;

6.       JDR /SIGN SVRAND0,                         RDFIFO;
```

The variable **an** must be declared double to store the 64-bit product. This assumes that the constant **a** is in M10, the constant **c** is in A10, the constant 7fffffff is in A01, and the scale factor $1/2^{31}$ (=4.656612e-10) is in M11. This is a new problem. Literals (constants) in microcode instructions are limited to 10 bits. Therefore, these constants cannot be contained in the microcode itself. They must be somewhere in data memory. On the other hand, these constants are read only once, so that there is no great need for them to be in fast memory. The way to accomplish this is to put the constants in a labeled location of slow memory. Then put the label in a (differently) labeled part of fast memory. The fast label will be small enough to be a literal in a microcode instruction. In this code (R1, R2) is the output vector, R3 is the address of the input integer scalar. As is traditional with random number generators, the final integer is written back to R3.

The software pipeline is three deep, so a write delay of two is needed and R1 has to be explicitly backed up twice. Finally, it is necessary to get a value from data into an R register. This is accomplished by the sequence (for example)

```
R1 = R1,                 i = MEM;          read integer

ENFDB;                                     enables path to RALU

R3 = FBACK;                                write to R register
```

This results in the value pointed to by R1 being written into R3.

```
/*>EH
Edit History
```

```
                Created:            Sept. 86
                Programmer:         David Scott

                Modified:
                Programmer:
                Reason:
*/
/*
>**
VORTEX MODULE: SRAND.VA             Integer input and single precision
                                    output

    OPERATION:  Random number function

    FUNCTION:   V$RAND_SP

WITH:          C0   32766 + N, where N is the size of the output
                    vector
               R1   address of the result vector v1
               R2   stride
               R3   address of the integer seed
               R4   zero

RETURNS WITH:       C0, C1, R1 modified
               R2, R3 unchanged
ALGORITHM:
                Pseudo random integers generated using

                J(i+1) = A*J(i) + C mod (2**31 -1)

                which are then scaled to the interval (0,1). Integer
                overflow may cause results to be negative and so the
                intermediate result is masked to turn off the sign
                bit.

A and C were chosen according to suggestions by Knuth

A = 843314861

C = 453816693

S = 4.656612e-10                    the scale factor (= 1/2^31)

M = 7FFFFFF    the sign mask
*/

#include VORDEF

                name SRAND
                vers 1.0

        defcmd C4, V$RAND_SP
```

```
#define _V$RAND_SP 0x1df

                public V$RAND_SP
                extern SPONE

        SECT SDM_COEFF
        even
DURANSP:
        dcl DURANDC
        SECT DM_TABLE
        even
DURANDC:
        dcl 0x3243f6ad
        dcl 0x1b0cb175
        dcl 0x2fffffd2
        dcl 0x7fffffff

                sect PM_FUNC
                int a, c, m, n, p
                float s, sf
                double an

V$RAND_SP:
        R0 = DURANSP, n = MEM;  /* read constants */
        enfdb;

        R0 = fback, a = MEM;

        R0 = R0 + 1, c = MEM;

        R0 = R0 + 1, s = MEM,
        M10 = a;                /* multiplicative factor A */

        R0 = R0 + 1, m = MEM,
        A10 = c;                /* Additive term C */

        R1 = R1 - R2,
        M11 = s,                /* scale factor S */
        A02 = s;                /* dummy number */

        R1 = R1 - R2,
        A01 = m;                /* overflow mask */

        R3 = R3, n = MEM;
        cont;

        A00 = n,                /* first seed */
        n = .LPASSA. A00;

        R1 = R1 - R2;

        y = .LPASSA. A02;

/* main loop: 6 cycles */
```

```
/*1------------------------------------------------------------*/

SRAND0:

M00 = ALUR -> n,         A11 = ALUR -> n
 an = M00 .*I. M10,      p = A01 .LAND. A11;

/*2------------------------------------------------------------*/

                                                M01 = ALUR -> y,
                                                sf = M01 .*S. M11;

/*3------------------------------------------------------------*/

DCCNTR C0;

/*4------------------------------------------------------------*/

A00 = PROD -> an,        A12 = ALUR -> p,
n = A00 .+I. A10;

/*5------------------------------------------------------------*/

                         s = .SFLTDB. A12,      R1 = R1 + R2,
                                                MEM = sf,
                                                FIFO = PROD -> sf;

/*6------------------------------------------------------------*/

JDR /SIGN SRAND0,                               RDFIFO;

/*-------------------------------------------------------------*/

                         A11 = ALUR -> n,
                         p = A01 .LAND. A11;

/*2------------------------------------------------------------*/

                                                M01 = ALUR -> y,
                                                sf = M01 .*S. M11;

/*3------------------------------------------------------------*/

cont;

/*4------------------------------------------------------------*/

                         A12 = ALUR -> p;

/*5------------------------------------------------------------*/

                         s = .SFLTDB. A12,      R1 = R1 + R2,
                                                MEM = sf,
```

```
                                                FIFO = PROD -> sf;
/*6----------------------------------------------------------------*/
                                                RDFIFO;
/*1----------------------------------------------------------------*/
cont;
/*2----------------------------------------------------------------*/
                                                M01 = ALUR -> y,
                                                sf = M01 .*S. M11;
/*3----------------------------------------------------------------*/
cont;
/*4----------------------------------------------------------------*/
cont;
/*5----------------------------------------------------------------*/
                                                R1 = R1 + R2,
                                                MEM = sf,
                                                FIFO = PROD -> sf;
/*6----------------------------------------------------------------*/

/* store seed */
        RDFIFO,
        n = .LPASSB. A12;

        cont;

        cont;

        R3 = R3, MEM = n,
        FIFO = ALUR -> n;

        RDFIFO, RTN;
        END
```

SPARSE DAXPY

We now look at computing the inner product of a dense vector y with a sparse vector x. The non-zero elements of x are stored sequentially and an integer index vector IND which gives the index of each element of x. In FORTRAN

```
      S = 0
      DO 10 I = 1, M
```

```
              S = S + X(I)*Y(IND(I))
10          CONTINUE
```

This requires reading integer indices into R registers. Unlike the random number generator, these indices are offsets to a base address. The following code is legal, but is dangerous.

```
R7 = R7 + R8, i = MEM;

ENFDB;

R11 = R0 + FBACK;
```

Where R0 is the base address. The danger is that this instruction truncates FBACK to 10 bits. If this is ok, then it can be used to save cycles.

However, in this example we will assume that 10 bits is not enough; that is, the indirect address may be larger than 1023, in which case the only safe construct is

```
Rx = FBACK
```

which stores a full 32 bits. We will assume that the computation is done in double precision.

The inner loop requires 6 cycles:

```
R7 = R7 + r8,          M10 = y,                A00 = PROD -> Z,
i = MEM,                                       s = A00 .+.D. A10;

ENFDB,                 M00 = x,
R0 = R3,               z = M00 .*D. M10;

R11 = FBACK;

R0 = R0 + R11;                                 A10 = ALUR -> s;

R0 = R0 + R11,
y = MEM;

R5 = R5 + R6,
x = MEM;
```

The double addition of R11 is due to the fact that double precision variables occupy two words. This code also assumes that R3 is actually two less than the start address of the dense vector y. This is because integer indices start at 1, but offsets start at zero.

One cycle could be saved if the elements of IND were multiplied by 2. Two more cycles could be saved if IND were overwritten by absolute pointers to the appropriate elements of y. Then the construct

```
F11 = FBACK,
y = MEM;
```

could be used. This works only if IND was not used for any other purpose.

MATRIX TRANSPOSE

As a final example, we look at a non-in-place transpose of a rectangular double precision (or complex) matrix.

The input is assumed to be N x M and the N is the length which is stored in C0 by the interface routine. Because of the doubly nested loop, it is necessary to remember the initial values of R1, R3 and C0. R1 and R3 are saved in other R registers. C0 is pushed onto the stack. It is important that no extra counter value be left on the stack (that is, all pushed values must be popped). Like all vector pointers, R1 and R3 are backed up by the interface routine. That is:

```
R1 = R1 - R2
R3 = R3 - R4
```

Because of the double nesting, it is also necessary to back up R1 and R3 in the other direction, that is:

```
R1 = R1 - 2
R3 = R3 - 2
```

The outer loop is of length M. It is necessary to store this into a C register. The jump instruction jumps if the C register is negative. To preserve the semantics, it is necessary to add 32766 to M and store this in C2. This overflows and makes C2 very negative. Each decrement makes it more negative until it overflows again and becomes very positive. This terminates the loop. The value 32766 is stored in register R27.

```
#include /usr/ipsc/lib/vordef.vh

                      name DTRANS
                      vers 1.0

             defcmd P6, V$TRANS_DP

#define _V$TRANS_DP  0x1e7

                      public V$TRANS_DP


                      sect PM_FUNC

                      double x1, x2
                      int i
/*
```

This routine Transposes a NxM double precision matrix A into a matrix B. The fortran equivalent of A and B are

```
         Double Precision A(lda, M), B(ldb, N)
         do 10 i = 1, M
            do 10 j = 1, N
10             B(i,j) = A(j,i)
```

Because of memory pipelining, two values of A are read and then they are written to B. Special care must be taken to make the code correct when N is odd. The special instruction JTWO is used. This skips one instruction. It is equivalent to JDR with a label on the second instruction except that JTWO doesn't need the label. This leaves the literal field available for other use. For example the register calculation could have been R3 = R3 + 2 which would have used the literal field. */

```
/*
Register Usage

Input
        R1      b pointer
        R2      leading dimension of b in words (ldb*2)
        R3      a pointer
        R4      leading dimension of a in words (lda*2)
        R5      M
        R6      zero

(NOTE: the prolog routine P6 backs up R1, R3, and R5--that is
R1 = R1 - R2
R3 = R3 - R4
R5 = R5 - R6
This prepares for the striding through the vector.  It is the only
reason that R6 has to be zero.)

Used
        R7      used to remember R1
        R8      2 (stride)
        R9      used to remember R3
Input
        C0      N
Used
        C2      M

*/

V$TRANS_DP:

        R5 = R5, i = MEM;                       Set C2 to M (+32766)
        ENFDB;
        R0 = FBACK;
        PAUSE, R0 = R0 + R27, ENRAL, PFBRAL;
        cont;
        PAUSE, HOLDB, ENFDB;
        WRCNTR C2 FBACK;

        R8 = 2;                                 Stride

        R7 = R1;                                Save R1
        R7 = R7 - R8;                           Back up
        R9 = R3;                                Save R3
        R9 = R9 - R8;                           Back up
```

```
DOUT:

        PSCNTR C0;                              Save C0

        R1 = R7;                                Reset R1
        R1 = R1 + R8;                           Next Row
        R7 = R1;                                Save R1

        R3 = R9;                                Reset R3
        R3 = R3 + R4;                           Next Column
        R9 = R3;                                Save R3

DIN:            R3 = R3 + R8,                   Read first
                x1 = MEM,
                RDFIFO,
                DCCNTR C0;                      First decrement

                R3 = R3 + R8,                   Read second
                x2 = MEM,
                JTWO /SIGN;                     Skip on NOT end

                FIFO = x1, JDR ODD;             Odd end

                R1 = R1 + R2,                   First write
                MEM = x1,
                FIFO = x1,
                DCCNTR C0;                      Second decrement

                R1 = R1 + R2,                   Second write
                MEM = x2,
                FIFO = x2,
                RDFIFO,
                JDR /SIGN DIN;
EVEN:   RDFIFO,                                 Even End
        DCCNTR C2;

        PPCNTR C0;                              Recover C0

        JDR /SIGN DOUT;

        RTN;

ODD:    R1 = R1 + R2,                           Odd End
        MEM = x1,                               Last write
        DCCNTR C2;

        PPCNTR C0,                              Recover C0
        RDFIFO;

        JDR /SIGN DOUT;

        RTN;
        END
```

Appendix A.

The following list gives the most important restrictions and conditions for writing microcode.

1. Memory fetches take two cycles. That is, they arrive on the A-bus for storage in arithmetic registers two cycles after the address calculation.
2. ALU operations take three cycles.
3. 32-bit multiplies take three cycles.
4. 64-bit multiplies take five cycles.
5. No multiply can be started on cycles 1, 2, or 4 after a 64-bit multiply is started (3 is OK).
6. No M-register can be loaded on the cycle after a 64-bit multiply is started.
7. Any 32-bit M-register can be loaded and used on the same cycle.
8. M00 can be loaded and used for a 64-bit multiply in one cycle but M10 must be loaded at least one cycle early.
9. Any 32-bit A register can be loaded and used on the same cycle.
10. Even A registers (A00, A02, A10, A12) must not be loaded on the cycle after they are used in an ALU operation.
11. Integer multiplies produce 64-bit results and cannot go directly to the FIFO.
12. Only one of the three sources for the memory bus can be used in one cycle. That is, only one of the following three possibilities:

    a. ENRAL on the previous cycle.

    b. = MEM on the previous cycle.

    c. RDFIFO on this cycle.

    If two are used, a parity error will result.

Appendix B. How to write interface routines

ROGER!!