WM Machine Architecture Measurement An Initial Analysis

Wm. A. Wulf and Anita K. Jones

November 17, 1988

1 Introduction

The objective is to analyze the performance of the WM architecture, and to compare it to that of other machines. Architecture and implementation cannot be cleanly separated for such purposes. However, our approach is to characterize the work accomplished in the execution of an instruction summing the costs of primitive unit operations performed in the course of executing the instruction. By characterizing instructions from widely different architectures in terms of the same primitive unit operations, they can be compared.

In this initial analysis, we have selected a small set of benchmarks and coded them in C and in WM assembly language. We measured their performance ("wall clock ticks" to complete the benchmark) using the WM simulator. We also determined, by hand counts, the unit operations performed, weighted by the cost per unit operation type. We then computed the "units of work accomplished per wall-clock tick". When this number is greater than one, it indicates the degree of concurrency of execution of the unit operations.

2 Initial Benchmarks

First and foremost the benchmarks described below were selected because they were at hand and small, so that hand coding was a reasonable task.

There is some degree of balance in the benchmarks. They are representative of several different styles of programming and application.

The benchmarks include:

- Whetstone excerpts: Three key routines. Representative of scientific calculation. Nasty in terms of data dependencies.
- Infinite Impulse Filter: Recurrence in two levels. Representative scientific function.
- Dot Product: Workhorse mathematics function. Streamable. Very tight loop.
- Lawrence Livermore Loop Number 5: Recurrence. Tight loop. Representative of some scientific calculations.
- Fast Fourier Transform Inner Loop: Frequently used mathematics function. Streamable.
- Unix string manipulation: String copy and string compare. Representative of systems code and text processing code.
- Saxpy: Excerpted from Linpack. Tight loop. Input vectors have differing strides.
- Link list scan: Loop controlled by link field accesses in contrast to predictable vector accesses. Representative of search algorithms. Representative of Operating Systems applications.

Code for the benchmarks is shown in the Appendix. The code being measured is typically found between two simulator TIMER commands.

3 Characterizing Execution

To characterize the function performed by an instruction, we define unit operations, UO's. The unit instruction is one of: load value from memory; store value to memory; or a single arithmetic, logical or control operation with register values or small literals as arguments. For a given technology, we assume that the execution time of a unit operation can be specified as an

integral multiple of some basic time unit – "the tick"; moreover, we assume that this execution time is independent of its location/use in a program or of the values of its operands. If X is a unit operation, then t(x) is it's execution time. No time is ascribed to instruction fetch and decode.

The effect of executing an instruction from any architecture can be described by a sequence of unit operations. For example, the VAX instruction addw3 x, (r2)+, r3 can be described by

```
load value from memory
increment register value
add two register values
```

Further, the "work accomplished" by this instruction can be characterized as the sum of the execution costs of its constituent UO's

```
t(load) + t(add) + t(add) .
```

Note that in general the time required to execute this instruction will not be equal to the work accomplished; it may be greater (because, for example, instruction decode is serial with actual execution of the instruction or operands are not available) or less (typically because there is some concurrency in the execution of the component UO's).

4 Analysis

4.1 Rules for Counting UO's on WM

Counting rules include:

- All jumps have the same function and cost.
- In non-streaming mode, references to FIFO-backed registers count zero. In streaming mode, references to f0, r0, r1, and f1 count as two operations, load and address increment.
- In multiple operator instructions, only the operators doing useful work are counted.
- If an instruction is a simple register to register move, then (it is assumed to be necessary to the algorithm implementation and) the count is one.

4.2 Cycles per Operation

The assumptions about the number of cycles to perform an operation correspond to the current Stellar machine implementation. The Operation Time values were specified by Todd Basche and are:

- 12 ticks: floating divide; float reverse divide
- 10 ticks: integer divide; integer reverse divide
- 4 ticks: float multiply
- 2 ticks: all float operators except divides, multiply and nop; integer multiply
- 1 tick: jumps; loads; stores; nops; all operators not named above

4.3 Benchmark Analysis

The following table shows performance measures of the inner loop (typically) of a set of benchmarks. The most important result is the right hand column which shows the amount of work per tick.

To compute work per tick for each benchmark, the code being measured is inspected to determine the amount of UO work done in one iteration of the loop. This work count is annotated on the benchmark code in the Appendix; the total count appears in the second column below. The amount of work is the sum of the number of ticks to perform each of the set of unit operations. For each unit operation the tick cost is given above. The actual simulator file defining operation costs is in the Appendix.

The third column gives the tick count for the measured code as determined by the WM simulator. Except for Whetstone derivative benchmarks each benchmark involved loops which were run for many (20 to 100) iterations so that concurrency of multiple iterations was possible. The simulator given tick count is conservative in that all instructions were completed and final stores were complete before the final tally was made. This definition of what to measure is conservative in that the last instructions in a loop may overlap with processing which follows in actual programs.

WM Performance using "Stellar Operation Times"

	UO Work	Simulated Time	Work/Tick
FFT	49	29	1.69
IIR	22	16	1.37
DOT	11	4	2.75
LLL5	13	6	2.16
WHET-pa	51	40	1.27
WHET-p0	22	14	1.57
WHET-p3	31	29	1.06
STRCPY	6	2	3.00
STRCMP	8	3	2.66
SAXPY	13	4	3.25
LINKLIST	8	5	1.60

5 Notes on the Benchmarks

Simulated time was measured in two ways. For the whetstone derivative benchmarks, the code is packaged as subroutines. So it did not seem appropriate to measure that code as though in a loop. Consequently the measurements reported here for whetpa, whetp0 and whetp3 are measured from the tick in which the first instruction of the code is processed through that tick in which the last portion of the last instruction is completed. (Note that whetpa has 8 loop iterations in it, by definition.) Any overlap of subroutine entry and return is not taken into consideration. Consequently, these measurements are conservative.

WHETp0 is an example in which the FEU waits on the IEU.

The remaining benchmarks all involve loops. These were measured in looping code that did permit one iteration to overlap execution with others. Typically, 100 iterations were measured. The code executed in most loops is not data dependent. Exceptions include LINKLIST. In this case three measurements were taken. In the best case each time through the loop, the list element was not interesting and therefore not processed. In the worst

case, the list element was always interesting and always processed. Happily the average case measurements fall in between best and worst.

STRCMP is interesting in that execution of the first loop instruction is started in the IEU every three ticks. The IEU is held up one tick, awaiting the condition code from the previous test to be processed by the IFU.

6 The WM Simulator

The simulator has some assumptions that affect its timing reported under "simulated time" in the measurements above. Several of interest are discussed below.

Memory is "one tick fast". If execution of a load/store operation is completed during one tick in the IEU, then the data is available in the next tick. This is reasonable if one assumes perfect and fast caching. But, it is a simplification to be aware of.

The IFU is infinitely fast. It is capable of performing any of its operations for which data is ready and buffering space is available, unless the operation necessitates synchronization with other function units. It takes a tick for a condition code value to get from an EU to the IFU

Benchmarks written in C and WM code

These benchmarks are included here. Most provide corresponding C or Fortran code in the comments to document the computation being performed.

```
                -- FFT Inner Loop
                --       (taken from the WM manual)

    synch
    LW r9 := j                      -- r10 == j
    r10 := r0
    lf r9 := ekrp
    f10 := f0
    lf r9 := ekip
    f11 := f0

    r3 := z                         -- r3 == address of z
    r4 := zm                        -- r4 == address of zm
    r5 := wk                        -- r5 == address of wk
    r6 := wj                        -- r6 == address of wj

    SinF f0, r3, r10, 8             -- stream z(i).rp's into f0 FIFO
    SinF f1, r4, r10, 8             -- stream zm(i).rp's into f1 FIFO
    SoutF f0, r5, r10, 8            -- stream wk(i).rp's into f0 FIFO output
    SoutF f1, r6, r10, 8            -- stream wj(i).rp's into f1 FIFO output
    TIMER clear, start
                                WORK
L:  f3 := (f0 nop f9) nop f9    2   --    zrp := f0
    f4 := (f1 nop f9) nop f9    2   --    zmrp := f1
    f5 := (f0 nop f9) nop f9    2   --    zip := f0
    f6 := (f1 nop f9) nop f9    2   --    zmip := f1
    f0 := (f3 + f4) nop f8      4   --    f0 := zrp + zmrp
    f0 := (f5 + f6) nop f8      4   --    f0 := zip + zmip
    f7 := (f3 - f4) * f10       6   --    ft1 := (zrp - zmrp) * ekrp
    f8 := (f5 - f6) * f11       6   --    ft2 := (zip - zmip) * ekip
    f1 := (f7 nop f8) - f8      4   --    f1 := (ft1) - ft2
    f7 := (f5 - f6) * f10       6   --    ft1 := (zip - zmip) * ekrp
    f8 := (f3 - f4) * f11       6   --    ft2 := (zrp - zmrp) * ekip
    f1 := (f7 nop f8) - f8      4   --    f1 := (ft1) - ft2
    JNIf0 L                     1   -- loop if not done
                               ---
    TIMER stop print           49

    .section data
j:  .word 4                         -- Count control, MUST be even=2*# loops
ekrp: .float 1.0
ekip: .float 2.0
z:  .float 1.0,0.0                  -- z vector; zeros are ip placeholder
    .float 2.0,0.0,3.0,0.0,4.0,0.0,5.0,0.0,6.0,0.0
zm:  .float 11.0,0.0                -- zm vector
    .float 12.0,0.0,13.0,0.0,14.0,0.0,15.0,0.0,16.0,0.0
wk:  .float 1.0,0.0                 -- wk vector
    .float 0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0
wj:  .float 1.0,0.0                 -- wj vector
    .float 0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0
```

```
-- INFINITE IMPULSE RESPONSE FILTER    Nb. recurrences
--      FOR i in 3 .. N LOOP
--              a(i) := (b(i) + a(i-1)* k1 + a(i-2)* k2) /2;
--      END LOOP;
--                      or
--  for(i=3; i<=N; i++) a[i] = (b[i] + a[i-1]*k1 + a[i-2]*k2) / 2;

    synch                           -- ensure data section reg, r8, is loaded

    lw   N                          -- r5 == N    loop control
    r5 := r0
    lw   k1
    r6 := r0                        -- r6 ==  k1
    lw   k2
    r12 := A
    r13 := B
    r4 := r0                        -- r4 ==  k2
    r12 := (r12 + 12)               -- address of A[3]
    r13 := (r13 + 12)               -- address of B[3]

    lw  r9 := (r12 - 4)             -- load a(2)
    lw  r9 := (r12 - 8)             -- load a(1)

    SinW  r0, r13, r5, 4            -- Stream Word in FIFO 0
    SoutW r0, r12, r5, 4            -- Stream out FIFO 0
    r10 := r0                       -- a(i-1) := a(2)
    r11 := r0                       -- a(i-2) := a(1)
    TIMER clear start

lp: r9 := (r10 * r6) + r0     5     -- b(i) + a(i-1) * k1
    r9 := (r11 * r4) + r9     3     -- + a(i-2) * k2
    r11 := r10                1     -- a(i-2) := a(i-1)
    r10 := (r9 / 2)           10    -- a(i-1) := a(i)
    r0  := r10                2     -- a(i) :=
    jnir0 lp                  1     -- loop if not done
                              ---
    TIMER stop print          22

    .section data
N:  .word 5                         -- Loop control variable
k1: .word 1
k2: .word 1

A:  .word 2,3,4,5,6,7,8,9,10        -- A vector

B:  .word 2,3,4,5,6,7,8,9,10        -- B vector
```

```
                -- STREAMED  DOT  PRODUCT
                --  sum = 0.0;
                --  for (i=0; i<N; i++) sum += a[i] * b[i];

    synch                       -- ensure data section reg, r8, is loaded

    lw  N                       -- r5 == N    loop control
    r5 := r0
    r6 := A                     -- r6 == address of A(0)
    r7 := B                     -- r7 == address of B(0)
    f4 := (f4 - f4)             -- init sum to 0.0
    SinF f0,r6,r5,4             -- stream A(i)'s into f0 FIFO
    SinF f1,r7,r5,4             -- stream B(i)'s into f1 FIFO
    TIMER clear start

l:  f4 :=(f0 * f1) + f4   10    -- do it!
    jnif0 l                1    -- loop if not done
                           --
    TIMER stop print
    sf r9 := RES                -- store result
    f0 := f4

    .section data
N:  .word 5                     -- Loop control variable
RES: .float 0.0                 -- site to store result
A:  .float 1.0                  -- A vector of length N+1
    .float 1.0,1.0,1.0,1.0,1.0
B:  .float 1.0                  -- B vector of length N+1
    .float 1.0,1.0,1.0,1.0,1.0
```

```
                -- Lawrence Livermore Loop # 5
                --      (tri-diagonal elimination)
                --   DO 5 i=2,N
                --5  X(i) = Z(i) * (Y(i) - X(i-1))
                --            or
                --   for (i= 2; i<n; i++)
                --      x[i] = z[i] * (y[i] - x[i-1]);


    synch                       -- ensure that the stack ptr is loaded
    r3 := Z                     -- r3 == address of Z
    r4 := Y                     -- r4 == address of Y
    r5 := X                     -- r5 == address of X
    LW  N                       -- r7 == N
    r7 := r0
    r4 := (r4 + 8)              -- r4 == address of Y(2)
    r3 := (r3 + 8)              -- r3 == address of Z(2)
    r5 := (r5 + 4)              -- r5 == address of X(1)
    r6 := (r7 - 1)              -- r6 == N-1
                                -- f0 (input) == stream of Z(i)s
                                -- f1 (input) == stream of Y(i)s
                                -- f0 (output) == resulting stream of X(i-1)s
                                -- f3 == X(i-1)
    LF (r5)                     -- enque request for X(1)
    SinF f0, r3, r6, 4          -- stream Z(i)'s into f0 FIFO
    SinF f1, r4, r6, 4          -- stream Y(i)'s into f1 FIFO
    SoutF f0, r5, r7, 4         -- prep to stream X(i)'s into f0 FIFO output
    f3 := f0                    -- init f3 with X(1)
    Nop                         -- pause for data dependency rule
    TIMER start clear

L:  f3 := (f1-f3) * f0          -- compute X(i)
    f0 := (f3)                  -- store PREVIOUS X, X(i-1)
    JNIf0 L                     -- loop if not done

    TIMER print
    f0 := f3                    -- store last X

    .section data
N:  .word 5                     -- Loop control variable
X:  .float 1.0                  -- X vector of length N+1
    .float 2.0,3.0,4.0,5.0,6.0
Y:  .float 1.0                  -- Y vector of length N+1
    .float 2.0,4.0,6.0,8.0,10.0
Z:  .float 1.0                  -- Z vector of length N+1
    .float 2.0,4.0,6.0,8.0,10.0
--newX:  .float 1.0             -- expected X vector AFTER computation
--     .float 2.0,8.0,-4.0,96.0,-860.0
```

10
2
1
13

```
--              Whetstone subroutine pa
--
-- pa(e)
--   float e[];
--   {
--     int j;
--     j = 0;
--     a1: e[1] = (e[1]+e[2]+e[3]-e[4])*t;
--         e[2] = (e[1]+e[2]-e[3]+e[4])*t;
--         e[3] = (e[1]-e[2]+e[3]+e[4])*t;
--         e[4] = (e[1]+e[2]+e[3]+e[4])*t;
--     j += 1;
--     if ((j-8)<0) goto a1 else goto a2;
--     a2: ;
--   }

      r5 := 0                     -- J == 0
      LF  T
      r6 := Eaddr                 -- r6 == address of E
      f5 := f0                    -- f5 == T
      f7 := (f7-f7)               -- f7 == 0.0
      LF  (r6 + 4)                -- get the e[]'s
      LF  (r6 + 8)
      LF  (r6 + 12)
      LF  (r6 + 16)
      f11 := f0                   -- e[1]
      f12 := f0                   -- e[2]
      f13 := f0                   -- e[3]
      f14 := f0                   -- e[4]
      TIMER clear start
                                  WORK
loop: SF  (r6 + 4)                  2
      SF  (r6 + 8)                  2
      SF  (r6 + 12)                 2
      SF  (r6 + 16)                 2
      r5  := (r5+1) < 8             2   -- test loop termination
      f6  := (f11+f12) + f13        4
      f7  := f7                     0   -- floating Nop
      f11 := (f6-f14) * f5          6
      f0  := (f5 nop f5) @' f11     0   -- store e[1]; note data dep
      f6  := (f11+f12) - f13        4
      f7  := f7                     0   -- Nop
      f12 := (f6+f14) * f5          6
      f0  := (f5 nop f5) @' f12     0   -- store e[2]
      f6  := (f11-f12) + f13        4
      f7  := f7                     0   -- Nop
      f13 := (f6+f14) * f5          6
      f0  := (f5 nop f5) @' f13     0   -- store e[3]
      f6  := (f12-f11) + f13        4
      f7  := f7                     0   -- Nop
      f14 := (f6+f14) * f5          6
      f0  := (f5 nop f5) @' f14     0   -- store e[4]
      JumpIT loop                   1
                                   ---
      TIMER stop print             51
      .section data
T:      .float 1.0
Eaddr:  .float 1.0,2.0,3.0,4.0,5.0
```

```
--              Whetstone subroutine p0
--
-- p0()
--   {
--     e1[j] = e1[k];    -- Note we do not know whether j,k, and l might
--     e1[k] = e1[l];            have the same values, but the hardware
--     e1[l] = e1[j];            is supposed to sort this out.
--   }

      TIMER  clear, start

      r10 := E1addr                 -- r10 == e1 address     1
      LW  J                                                  1
      LW  K                                                  1
      LW  L                                                  1
      r4 := r0                      -- r4  == J              0
      r5 := r0                      -- r5  == K              0
      r6 := r0                      -- r6  == L              0
      LF (r5 ^ 2) + r10 -- e[k]                              3
      SF (r4 ^ 2) + r10 -- e[j]                              3
      f0 := f0                                               2
      LF (r6 ^ 2) + r10 -- e[l]                              3
      SF (r5 ^ 2) + r10 -- e[k]                              3
      f0 := f0                                               0
      LF (r4 ^ 2) + r10 -- e[j]                              3
      SF (r6 ^ 2) + r10 -- e[l]                              3
      f0 := f0                                               0

      TIMER stop print                                      22

      .section data
J:      .word 1
K:      .word 1
L:      .word 1
E1addr: .word 1,2,3,4,5
```

```
--              Whetstone subroutine p3
--
-- p3(x,y,z)
--   float *x, *y, *z;
--   {
--     float x1, y1;
--     x1 = *x;
--     y1 = *y;
--     x1 = t*(x1+y1);     -- this and next two instructions are known
--     y1 = t*(x1+y1);     --     to be "hard" on the Stellar machine
--     *z = (x1+y1)/t2;    --     because of the data dependencies
--   }

      Synch                          -- to get stack pointer loaded
      TIMER clear start

      LF  Y                          1
      LF  T                          1
      LF  X                          1
      LF  T2                         1
      SF  Z                          1
      f6 := f0                       0  -- f6 == *Y
      f7 := f0                       0  -- f7 == T
      f5 := (f0+f6)*f7               6  -- f5 == (*X+Y1)*T
      f9 := (f9 nop f9) @' f9        0  -- Note f5 req'd as inner op of next instr
      f6 := (f5+f6)*f7               6  -- f6 == (X1+Y1)*T
      f9 := (f9 nop f9) @' f9        0  -- Note f5 req'd as inner op of next instr
      f0 := (f5+f6)/f0              14  -- *Z := (X1+Y1)/T2
                                    31
      TIMER stop print

      .section data
X:    .float 1.0
Y:    .float 1.0
Z:    .float 1.0
T2:   .float 1.0
T:    .float 1.0
```

```
--              Unix string copy
--
-- strcpy(s1,s2)
--   char *s1, *s2;
--   { char *s = s1;
--     while (*s1++ = *s2++);
--     return (s);
--   }

      r11 := (31 ^ 20)          -- get large count
      r5 := s1                  -- base address of s1
      r6 := s2                  -- base address of s2
      SinB  r0, r5, r11, 1
      SoutB r0, r6, r11, 1
      TIMER clear start

loop: r0 := (r0 <> 0)           -- copy and test for null terminator
      JumpIT loop               -- loop if

      TIMER stop print
      r0 := 0                   -- write the terminator
      StopAll

      .section data
s1:   .byte 1,2,3,4,5,0               -- each byte specification will be
                                      --    word aligned; careful
s2:   .word 0,0,0
```

5
1
6

```
--              Unix string compare
--
-- strcmp(s1,s2)
--   char *s1, *s2;
--   {
--     for (; *s1 == *s2; s1++, s2++) if (!*s1) break;
--     return (*s1-*s2);
--   }

      r11 := (31 ^ 20)          -- get large count
      r5  := s1                 -- base address of s1
      r6  := s2                 -- base address of s2
      SinB  r0, r5, r11, 1
      SinB  r1, r6, r11, 1
      r12 := 0                  -- count for computing result
      TIMER clear start

loop: r10 := (r0 = 0) <> r1     -- is *s1 null? OR is *s1<>*s2?
      r12 := (r12 + 1)          -- count chars compared
      JumpIF loop

      TIMER stop print
      StopAll
      LB r7 := (r6 - 1) + r12   -- reload *s2[count-1]
      r10 := (r10 - r0)

      .section data
--s1:   .byte 1,2,3,0           -- s1 == s2
--s2:   .byte 1,2,3,0

--s1: .byte 0                   -- s1 is null
--s2: .byte 5,5,5,0

s1: .byte 3,4,5,6,7,0          -- s1 lexically greater
s2: .byte 3,4,5,1,0
```

```
            --  saxpy from LINPACK sources
            --      constant times a vector plus a vector
            --  unequal increments
            --  Note: incx and incy are increment parameters
            --  ix=1
            --  iy=1
            --      Do 30 i=1,n
            --              DY(iy) =  DY(iy) + DA*DX(ix)
            --              ix=ix+incx
            --              iy=iy+incy
            --      30 continue


    synch                        -- ensure that the stack ptr is loaded
    LF  DA                       -- f3 == DA
    LW  N                        -- r9 == N
    LW  INCX                     -- r8 == INCX
    LW  INCY                     -- r7 == INCY
    r3 := DY                     -- r3 == address of DY
    r4 := DX                     -- r4 == address of DX
    f3 := f0
    r9 := r0
    r8 := (r0*4)                 -- r8 == INCX * 4
    r7 := (r0*4)                 -- r7 == INCY * 4
                                 -- f0 (input) == stream of DY(i)s
                                 -- f1 (input) == stream of DX(i)s
                                 -- f0 (output) == resulting stream of DY(i)'s
    SinF f0, r3, r9, r7          -- stream DY(i)'s into f0 FIFO
    SinF f1, r4, r9, r8          -- stream DX(i)'s into f1 FIFO
    SoutF f0, r3, r9, r7         -- prep to stream DY(i)'s into f0 FIFO output
    TIMER start clear

L:  f0 := (f1*f3) + f0       12  -- DY(i) = DY(i) + DA*DX(i)
    JNIf0 L                   1  -- loop if not done
                             13
    TIMER stop print
    .section data
N:  .word 10                     -- Loop control variable
INCX: .word 1                    -- DX index increment
INCY: .word 2                    -- DY index increment
DA: .float 3.0
DX:  .float 1.0                  -- DX vector  10 some big
    .float 2.0,4.0,6.0,8.0,10.0,10.0,8.0,6.0,4.0,2.0
DY:  .float 1.0                  -- DY vector  80 some big
    .float 2.0,3.0,4.0,5.0,6.0,6.0,5.0,4.0,3.0,2.0
    .float 2.0,3.0,4.0,5.0,6.0,6.0,5.0,4.0,3.0,2.0
    .float 2.0,3.0,4.0,5.0,6.0,6.0,5.0,4.0,3.0,2.0
    .float 2.0,3.0,4.0,5.0,6.0,6.0,5.0,4.0,3.0,2.0
    .float 2.0,3.0,4.0,5.0,6.0,6.0,5.0,4.0,3.0,2.0
    .float 2.0,3.0,4.0,5.0,6.0,6.0,5.0,4.0,3.0,2.0
    .float 2.0,3.0,4.0,5.0,6.0,6.0,5.0,4.0,3.0,2.0
    .float 2.0,3.0,4.0,5.0,6.0,6.0,5.0,4.0,3.0,2.0
    .float 2.0,3.0,4.0
```

```
--    Scan down a linked list
--              Locate list element with certain property
--              Property here is highest value in a field
--
      r13 := 0                      -- rec. addr with highest priority value
      r12 := 0                      -- highest priority seen
      LW  n
      r6  := r0                     -- how many links to generate
      r10 := a                      -- addr of first recond
      r9  := a + 8                  -- addr of second record
      SoutW r1,r10,r6,8             -- set link field for n records
init:     r1 := r9
          r9 := (8) + r9
          jnor1 init

      LW r10 := a                   -- prime r0 with address of first rec
      LW r9  := a + 4               --          and its priority
        TIMER clear start
   WORK
lp:   1 r11:=r10                    b  w  -- save the current record pointer
      1 r10 := (r0 <> 0)            b  w  -- find the next link, if any
      1 r12 := (r0 > r12)           b  w  -- test the priority of curent
      1 JumpIF  xit                 |  |  -- exit if no next record
      1 LW (r10)                    |  |  -- start read of next record link
      2 LW (r10+4)                  |  |  -- start read of next priority
      1 JumpIF  lp                  b  w  -- loop if not higher priority
      1 r13 := r11                  -  w  -- save current pointer if highest
      1 Jump    lp                  8  w  -- loop
xit:  1 JumpIF  xit2                -  -  -- test priorty of last record
      1 r13 := r11                     10 -- save current pointer if highest
xit2: -
      12
        TIMER stop print

        .section data               -- base is a dummy record
n:     .word 19
-- Best case
--a:    .word 0,10
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1
--      .word 0,1

-- Worst case
--a:    .word 0,1
--      .word 0,2
--      .word 0,3
--      .word 0,4
--      .word 0,5
--      .word 0,6
--      .word 0,7
```

Annotation

b : executed in each iteration of best case data

w : executed in each iteration of worst case data

```
--        .word 0,8
--        .word 0,9
--        .word 0,10
--        .word 0,11
--        .word 0,12
--        .word 0,13
--        .word 0,14
--        .word 0,15
--        .word 0,16
--        .word 0,17
--        .word 0,18
--        .word 0,19
--        .word 0,20

-- Average case
a:        .word 0,1                -- all records have 2 fields
          .word 0,1                --    link w/ 0 == no next field
          .word 0,2                --    priority alway > 0
          .word 0,1
          .word 0,3
          .word 0,1
          .word 0,4
          .word 0,5
          .word 0,6
          .word 0,1                -- 20 record; 19 non null links
          .word 0,7                -- 10 succesive "higher" priorities
          .word 0,1                --    are encountered
          .word 0,1
          .word 0,8
          .word 0,1
          .word 0,9
          .word 0,1
          .word 0,1
          .word 0,1
          .word 0,15               --Note link field of last rec must stay zero
```

	Work	Simulated Time	Work/Tick
bestcase	8+*	5	1.6
worstcase	10+	6	1.67
average	9+	5.5	1.64

+ represents the addition 2 UO's performed as loop is exited

```
extern long iTimes[];
#define INTtimes    { 1, 1, 1, 1, 1, 1, 2, 1,10, 1, 1, 1,10, 1, 1, 1}
/*                    |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__greater than
**                    |  |  |  |  |  |  |  |  |  |  |  |  |  |__AND
**                    |  |  |  |  |  |  |  |  |  |  |  |  |__less than or equal
**                    |  |  |  |  |  |  |  |  |  |  |  |__reverse divide
**                    |  |  |  |  |  |  |  |  |  |  |__less than
**                    |  |  |  |  |  |  |  |  |  |__OR
**                    |  |  |  |  |  |  |  |  |__greater than or equal
**                    |  |  |  |  |  |  |  |__divide
**                    |  |  |  |  |  |  |__equal
**                    |  |  |  |  |  |__multiply
**                    |  |  |  |  |__equivalence
**                    |  |  |  |__reverse subtract
**                    |  |  |__reverse arithmetic shift left
**                    |  |__add
**                    |__conx
**                    |__subtract
*/

extern long fTimes[];
#define FLTtimes    { 2, 2, 2, 2, 2, 2, 4, 2,12, 2, 1, 2,12, 2, 1, 2 }
/*                    |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__greater than
**                    |  |  |  |  |  |  |  |  |  |  |  |  |  |__reverse NOP
**                    |  |  |  |  |  |  |  |  |  |  |  |  |__less than or equal
**                    |  |  |  |  |  |  |  |  |  |  |  |__reverse divide
**                    |  |  |  |  |  |  |  |  |  |  |__less than
**                    |  |  |  |  |  |  |  |  |  |__NOP
**                    |  |  |  |  |  |  |  |  |__
**                    |  |  |  |  |  |  |  |__divide
**                    |  |  |  |  |  |  |__equal
**                    |  |  |  |  |  |__ multiply
**                    |  |  |  |  |__
**                    |  |  |  |__reverse subtract
**                    |  |  |__greater than or equal
**                    |  |__add
**                    |__
**                    |__subtract
*/

extern long iTimes[16], fTimes[16];
```

Stellar Single Precision Operation Times

The WM Computer Architecture

Wm. A. Wulf
Pittsburgh, Pa.

27 December 1987

Introduction

The time to complete an application can be expressed as the product N·C·T, where N is the number of instructions executed, C is the number of cycles per instruction, and T is the time per cycle. The important lesson of the RISC architectures was that, by simplifying the instruction set and thus potentially *increasing* N, the other two terms are disproportionately decreased and overall execution time is reduced.

This paper outlines an architecture, called WM[1], which retains the RISC philosophy of "performance through simplicity". However, WM achieves significantly greater performance than RISC designs; it does so by:

(1) reducing N, while retaining the cycle time of a RISC machine.
(2) reducing C to *less than one*.

At first, these may seem as non-intuitive as the advantages of RISC machines did in 1975. However, for example, N can be reduced by at least two techniques: (1) an instruction can specify two or more independent actions -- each of which has RISC-like simplicity and speed, and (2) a single instruction can spawn a sequence of asynchronous actions -- each of which would otherwise have required a RISC-like instruction to be executed. C can be reduced to less than one by dispatching more than one instruction per cycle. The combined effect in WM can be quite dramatic -- as much as a 4-10 performance advantage over RISC designs.

Designing an architecture that is faster than the current generation of RISC chips isn't especially hard -- witness the plethora of (mini)supercomputer designs that have emerged recently. The charms of the WM design, however, are the simplicity of the mechanisms used, its obvious suitability for single-chip implementation, and its broad spectrum of applicability.

---

[1] Application for patent protection has been made for portions of the material described here.

The following sections briefly describe various aspects of the WM architecture. We then provide a few examples and use these to provide an intuitive feel for the machine's performance. Finally we make some general remarks on the design.

Description of the Architecture

General Information

WM is a 32-bit, byte addressed machine. It is a "load/store" architecture: arithmetic and logical operations can only be performed between registers. There are 32 integer/logical registers and 32 floating-point registers; a few of these registers have special purposes as will be discussed later.

All WM instructions are exactly 32-bits, and must be word-aligned; there are five instruction formats, as shown in Figure 1. Instructions are executed by one of the three, asynchronous components shown in Figure 2: the Instruction Fetch Unit (IFU), the Integer Execution Unit (IEU), and Floating Execution Unit (FEU). Integer and Load/Store instructions are executed by the IEU; floating point instructions are executed by the FEU; control and special instructions are executed by the IFU. An instruction may be dispatched to each of these components on each cycle.

The semantics of the WM instruction set are sequential; instruction *I* is executed before instruction *(I+1)*. However, the design has been partitioned to permit greater concurrency than is suggested by this statement. The principal state shared between the IFU, IEU, and FEU is a set of FIFOs ("first-in-first-out" queues); interactions between the components are implicitly synchronized through these FIFOs. Although strict sequentiality of instruction execution must be maintained for the components individually, no such requirement exists globally -- in principle the IFU, IEU and FEU can be executing instructions from quite distinct portions of the program concurrently. The relative progress of the components is governed by their relative speeds and by intrinsic data dependencies, not merely by PC-value. This independence between components significantly increases the potential for multiple instruction executions per cycle.

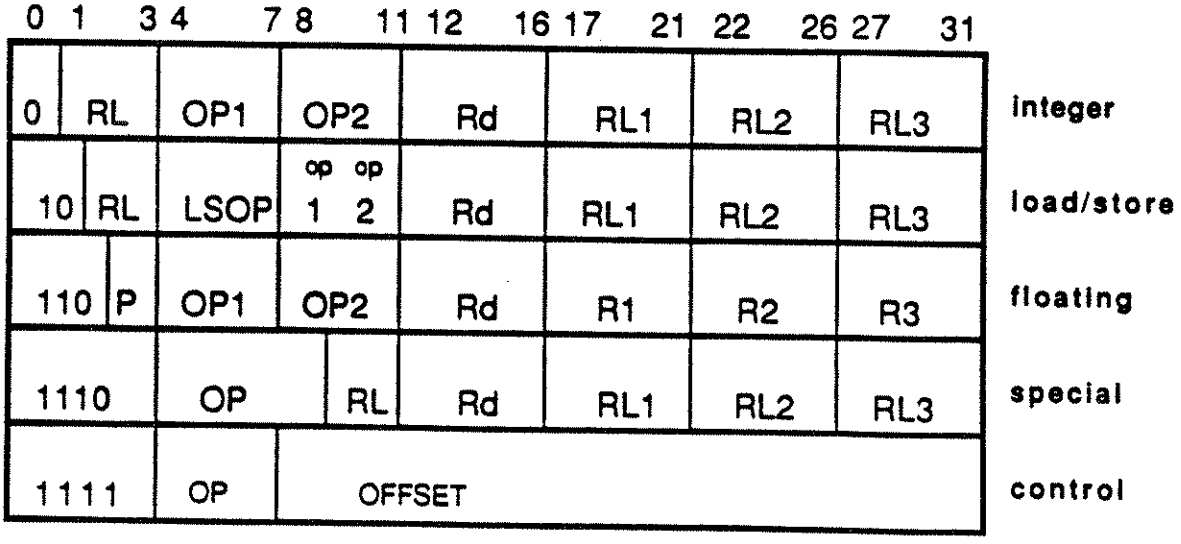

Figure 1: The WM Instruction Formats

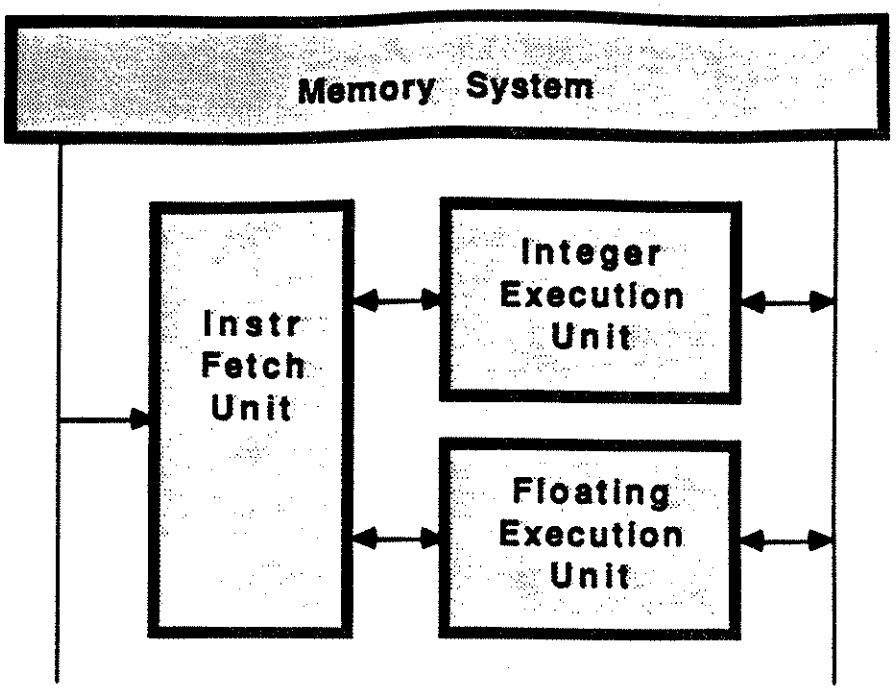

Figure 2: The IFU/IEU/FEU Decomposition

The Integer Instructions

The integer instructions evaluate the assignment:

Rd := O1 **op1** (O2 **op2** O3)

That is, they perform two operations on three source operands and place the result in a destination register. The source operands, O1-O3, may be either the contents of one of the integer registers or 5-bit unsigned literals from the instruction word[1].

Figure 3 illustrates the conceptual model of the IEU. The machine is pipelined so that an instruction can be dispatched to the IEU each cycle: while **op1**, the outer operation of one instruction is being executed in ALU1, **op2**, the inner operation of the next instruction is being executed in ALU2. This gives rise to the data-dependency rule:

> The result of one instruction is not available as an operand of the inner operation of the next instruction.

[1]The RLi fields are the operand specifiers and the RL field determines whether RLi is the name of the register holding Oi or is the literal Oi itself.

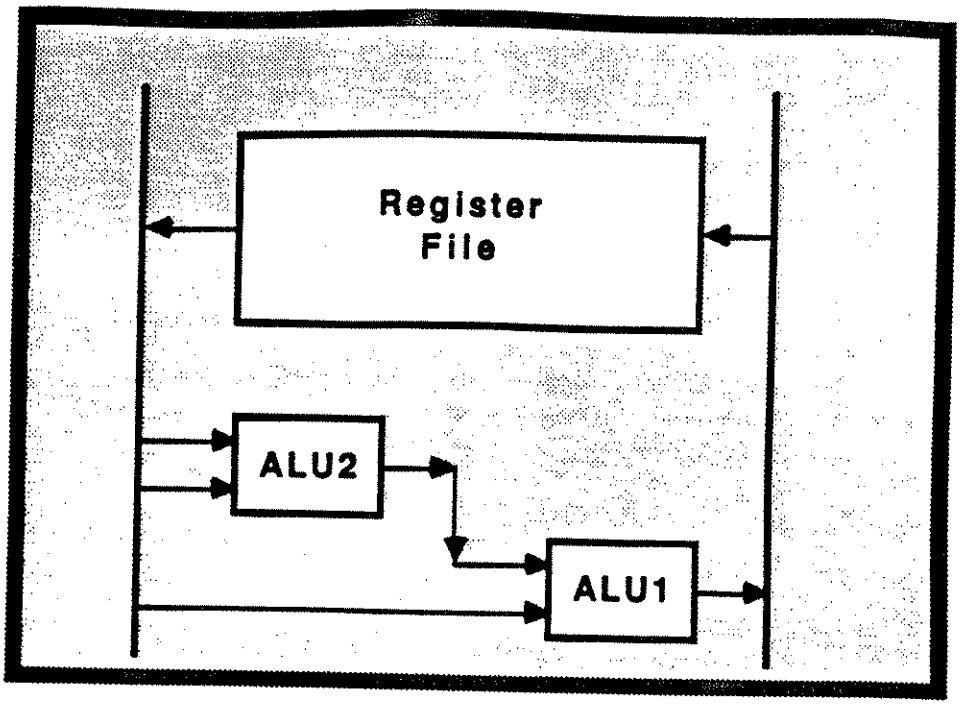

Figure 3: The IEU Conceptual Model

There are 16 operation codes for each of **op1** and **op2**, and they are mostly what one would expect -- add, subtract, and, or, etc.[1] In addition to these expected operators, there are a set of relationals: equal, greater-than, etc. The relational operators have three properties:

They are the *only* operations that set the condition code; the non-relational operators, e.g., add, do *not* ! We will see later that this is half of what permits WM to execute conditional jumps in *zero* time.

They produce their right operand as a result; the value of "(r11 < r10)" is the value contained in r10. Thus, an instruction such as

```
r10 := r11 > (r10 + 1)
```

both increments r10 and compares the incremented value to the value in r11. This is the "add-and-compare" portion of the the typical "add-compare-and-branch" loop control paradigm.

If two relationals occur in the same instruction, the resulting condition code is the logical "or" of the two. This permits, for example, range checking in a single instruction.

[1] In principal the operations available in the two ALUs need not be the same, but for the purposes of this paper we will assume they are. As one might expect, a good deal of cleverness was expended to determine just the right set of operators.

The Floating Point Instructions

The floating point instructions are similar to the integer instructions and also perform the computation

$$Fd := O1 \ \mathbf{op1} \ (O2 \ \mathbf{op2} \ O3)^1$$

The "p" bit of the instruction (Figure 1), determines whether this computation is performed in single or double precision. No literals are allowed.

The floating point operations are also pretty much what one would expect. Specifically, there are a set of relational operators with the same semantics as their integer counterparts. The conceptual model of the FEU is identical to that of the IEU, and it too is pipelined and has the same data dependency rule.

The Control Instructions

The control instructions perform PC-relative jumps, and contain a 24-bit displacement[2]. The "interesting" instructions are the conditional jumps -- for which we impose the following restriction on valid programs:

> Instructions containing relational operators must be dynamically paired one-for-one with conditional jump instructions.

As a consequence of this rule we can model this portion of the machine as a producer-consumer system. Instructions containing relational operators "produce" a condition-code value; conditional jumps "consume" one. As with all producer-consumer systems, we can synchronize the two participants by a FIFO. This means that the IFU is free to execute the conditional jump as soon as a condition code value becomes available, i.e., the FIFO is non-empty. Thus, if the compiler separates instructions containing relational operators from the corresponding conditional jump by at least one instruction[3], the execution of the the jump can be completely overlapped with that of other instructions and hence takes "zero time".

Note that, in practice, the ability of the compiler to move relational operators is significantly enhanced by the fact that other operators do not set the condition codes!

---

[1] To avoid confusion, the integer registers are referred to as r0-r31 and the floating point registers are referred to as f0-f31.

[2] Since all WM instructions are the same size and word-aligned, this displacement is padded with two trailing zeros before being added to the PC. An unconditional "jump indirect" is provided if you really have a program larger than 2^{24} instructions and wish to jump from beginning to end.

[3] This is identical to the problem of filling "branch shadows" in "delayed branch" or "branch with execute" schemes.

The Load/Store Instructions

The method of performing loads and stores is one of the more unusual aspects of the WM design, and is another source of its performance advantage. The scheme has three important differences from the conventional:

1. On most machines, a load (or store) specifies two quantities: a memory address and a register name -- i.e., "load the contents of location 42 into register 13", or "store the contents of register 3 into location 511".

   In WM, only the memory address is specified in the load/store instructions. The target of a load (source of a store) is implicitly "register zero" [1].

2. On most machines there are a set of "addressing modes", albeit simple ones for RISC machines, that are used to compute memory addresses as part of the load/store operation.

   WM achieves this effect with the standard integer operators that are part of the load/store instruction.

3. On most machines, the effect of a load/store is conceptually "immediate".

   WM interposes FIFOs between the registers and memory.

   - A "load" instruction is a request to enqueue data from a specific memory location into an input FIFO; data is dequeued from this FIFO by referencing register zero as a source operand of a data-manipulation instruction. Several load instructions may be executed, and consequently data enqueued, before being referenced and dequeued.
   - A "store" instruction is a request to dequeue data from an output FIFO and store it into a specific memory location. The data is enqueued in an output FIFO by specifying register zero as the destination of a data-manipulation instruction.

Load/store instructions specify an address. They do so by performing a computation semantically identical to the integer instructions. That is, they compute the assignment

Rd := O1 **op1** (O2 **op2** O3)

The initial execution of these instructions is identical to that of integer instructions. The operations, **op1** and **op2**, are a subset of the integer operations, and, when padded with leading zeros, are even encoded the same. The data paths, ALUs, and data dependency rule are the same; there is no special "address generation" unit in the hardware. Only during the assignment to Rd is there a difference between the load/store and integer instructions -- and it is in the form of an additional action. Concurrent with the assignment to Rd, the result of "O1 **op1** (O2 **op2** O3)" and the LSOP (Fig. 1) are sent to the memory system.

The value of "O1 **op1** (O2 **op2** O3)" is the address of the value to be loaded or stored. Note: it is *not* the value to be loaded or stored -- it is the *address* of the value.

The LSOP field determines the type of load or store to be done -- e.g., load-byte vs. load-word. This also implicitly determines whether the IEU (integer load/stores) or FEU(floating load/stores) is involved.

---

[1] This will be either r0 or f0, depending on whether an integer or floating load/store was done,

A typical sequence of events for loading/using a memory operand is as follows. First, a load instruction is executed. This provides an address, the kind of data to be loaded (word vs byte, sign-extended or not), and the FIFO to be used (integer vs floating point). The memory system now operates asynchronously to fetch the data, and at some later time inserts the data at the end of the appropriate input FIFO. In the mean time, the execution units are proceeding asynchronously. They may or may not have attempted to dequeue this data by referencing register zero as a source operand; if so, they will be blocked until the data is available. Eventually, however, an execution unit will reference register zero and the data will be dequeued.

The sequence of events possible for storing data is similar, but has the added possibility that the assignment to r0 (i.e., the computation of the value to be stored) and the store instruction can occur in either order. It is the presence of a <value, address> pair that triggers the actual store operation.

Note that the "addressing modes" of WM are just the set of two-operator, three operand integer expressions. This set conveniently includes most of the familiar addressing modes of CISC machines, e.g., scaled indexing. Because of the assignment to Rd, the set also includes a generalized form of "auto-increment".

The FIFO interface to memory encourages the compiler to move load operations to earlier points in the program, and hence to mask latency resulting from a cache miss. More importantly, however, it decouples the integer "address generation" portion of a load/store from the use or generation of the data. Thus, for example, the integer and floating point units of the machine can proceed asynchronously in a data-flow-like, "data availability" driven manner.

Special Instructions and "Streaming"

The "special" instructions include all those conventional instructions not covered in the previous discussion -- e.g. those for interacting with the operating system, for supporting subroutine linkages, etc. We will not discuss these. However, the specials also include the instructions that control an important feature of WM -- "streaming".

Streaming is simply a mechanism for causing a sequence of values to be loaded from, or stored to memory. An instruction such as

```
SinW    stride, count, base
```

initiates "streaming mode", in this case, a sequence of "word loads". The effect of this instruction is just as though "count" loads were executed, the first computing "base" as the address from which the load is to be done, and successive ones incrementing this address by "stride". Just as with the load instructions, the data from those memory locations is enqueued in an input FIFO and computational instructions access (and dequeue) that data by referencing register zero as a source operand.

The similarity of streaming to "vector load/store" operations should be apparent, and the rationale for its existence is similar -- it reduces the number of instructions that must be executed in loop bodies, and signals predictable reference patterns that can be exploited by the memory system. Note that WM has no vector instructions; streaming obviates the need for them. Moreover, unlike vector machines, the rate at which the operands are loaded or stored is determined by the rate at which they are consumed/produced by computational instructions that reference register zero as source/destination.

Streaming, as we shall see later, is extremely powerful, and it is useful to have at least two input and two output streams for each of the IEU and the FEU. Therefore, the "start streaming" operations name which FIFO is to be used; thus the format we shall use is

SinW fifo, stride, count, base

It is also useful to add a set of conditional jump instructions that can test whether or not a streaming operation is complete -- jNZ, "Jump on Stream Count Non Zero". Examples appearing later should motivate these decisions.

Examples and Rationale

Let's consider several versions of a "dot product" example. The algorithm is, of course:

```
      X = 0.0
      DO 10 i=1,N
10    X = X + A(i)*B(i)
```

In order to focus on the essentials of the example, let's suppose that the following registers have been assigned and the corresponding values assigned to them are:

f3 == 0.0	f4 == X
r4 == i	r5 == N
r6 == address of A(0)	r7 == address of B(0)
r8 == "trash"	

Remember, r0-r31 are the integer/logical registers in the IEU and f0-f31 are the floating point registers in the FEU. In coding for WM, we use this to distinguish integer and floating point instructions since they are otherwise similar in form. Thus,

```
r4 := r5 + (r6 - r7)      -- is an integer instruction, but
f4 := f5 + (f6 - f7)      -- is a floating point instruction.
```

Example 1: The Obvious Code

The most obvious , although not the best performing, WM code for the dot product loop is

```
        r4 := 1              -- (1) initialize i to 1
        f4 := f3             -- (2) initialize X to zero
Loop:   LF   r8 := r6 + r4*4 -- (3) load A(i) into FIFO f0
        LF   r8 := r7 + r4*4 -- (4) load B(i) into FIFO f0
        f4 := f4 + (f0*f0)   -- (5) compute next term
        r4 := r5<=(r4+1)     -- (6) increment and test i
        jT   Loop            -- (7) loop if not done
```

First, let's walk through the example explaining each line.

(1) This instruction copies the literal value one to register r4 (recall, integer instructions may contain 5-bit unsigned literal values). In reality, of course, this instruction contains two operations; as a notational convenience they needn't be written and the assembler inserts appropriate "nops".
(2) This merely copies the floating point zero value from register f3 to register f4. Note that, again, the two operations are not used in this instruction.
(3) This is a load-floating (LF) operation and computes the address of A(i) by using "scaled indexing"; since the address is not needed later, it is stored into the "trash" register. As a result of this instruction, the value of A(i) will, at some later time, be enqueued in the floating point input FIFO corresponding to f0.
(4) This is the analogous load of B(i). Note that at some later time, the value of B(i) will be enqueued after that of A(i) in the floating point input FIFO, f0. Note that we used "*" in these instructions for exposition; a shift is obviously better.
(5) This is the actual computation of X = X + A(i)*B(i). Note the two references to f0 -- these dequeue the values of A(i) and B(i) that were enqueued by the previous load instructions.
(6) This is the increment and test of i. Recall that the value of a relational is it's right operand -- the incremented value of r4 in this case.
(7) This is the branch ("jump on True") back to the head of the loop. It branches just in the case that the condition code set in (6) is "true".

Before modifying the example for better performance, it's worth noting several things. First, notice the use of two operations per instruction in the address computations, the dot-product computation itself, and the increment-and-test. The frequency of the "O1 **op1** (O2 **op2** O3)" pattern in real programs was a great surprise to this author -- but it is pervasive. Obviously, some number of expressions of this form "occur naturally" in any program. In addition, however, many common idioms have this form. For example, "multiply-and-add" is the central computation in many scientific computations:

- dot product,
- Horner's rule (polynomial evaluation),
- LU decomposition,
- etc.

Similarly, there are quite a number of "system-y" idioms of this form:

- addressing expressions (double indexing, scaled indexing, etc.),
- field extraction (shift-mask for unsigned extraction, shift-shift for signed extraction),
- increment-and-test,
- range checking,
- stack-adjustment with limit-testing,
- etc.

In a series of experiments compiling a variety of benchmarks, the number of WM instructions in inner loops was actually *smaller* than the number of VAX instructions -- the use of two-operations per instruction was the primary reason. Though surprising, this is one of the reasons for WM's performance advantage -- to complete an application it executes a number of instructions comparable to a CISC machine, but instructions are dispatched and executed at RISC machine rates.

Second, note the decoupling of the integer and floating point units. The first execution of line (5), the actual dot product computation, will likely be delayed some amount of time until the values of A(i) and B(i) are loaded. This does not, however, need to delay the execution of any of the integer or control instructions! Specifically, the remainder of the loop body can be executed and the next set of loads can be initiated. Depending on the latency of the cache/memory-system and the relative speed of

integer and floating point operations, the integer portion of the loop can happily be many iterations ahead of the floating-point portion.

Example 2: Improved Code

The example can be improved by noting that placing the increment-and-test immediately next to the conditional jump was a bad idea; it prevented the jump from being executed concurrently with the other instructions. It would be better to move the relational earlier in the loop.

It would be tempting to think that the "fix" is simply to interchange lines (5) and (6), but that won't help -- and the reason lets us emphasize a point. On each "cycle", WM can dispatch an integer instruction, dispatch a floating point instruction, and execute a branch[1]. Specifically, lines (5) and (6) can be dispatched at the same time, and hence interchanging them doesn't increase the "distance" between the increment-and-test and the conditional branch. A somewhat trickier transformation, but one that is easy for the compiler, is shown below:

```
        r4 := 1                 -- (1) initialize i to 1
        f4 := f3                -- (2) initialize X to zero
Loop:   LF  r8 := r6 + r4*4     -- (3) load A(i) into FIFO f0
        r4 := r5<=(r4+1)        -- (4) increment and test i
        LF  r8 := r7 + r4*4     -- (5) load B(i) into FIFO f0
        f4 := f4 + (f0*f0)      -- (6) compute next term
        jT  Loop                -- (7) loop if not done
```

By moving the increment-and-test over two instructions, including one of the loads, we have achieved the requisite separation. It might appear, however, that we have introduced a bug by incrementing r4 before the second load. Not so. Recall the data dependency rule says that the result of one operation is not available until the *outer* operation of the following instruction. Thus, the fetch of r4 in line (5) gets the previous, un-incremented value of i.

In this version, the integer unit executes 3 instructions, the floating-point unit executes one instruction (which overlaps with some or all of the integer instructions), and the IFU is able to completely overlap the execution of the branch instruction.

The exact performance of this example will depend upon the ratio of integer and floating-point operation times, but for many implementations, floating multiply will take at least three integer-operation times. For such implementations, the machine is completely limited by the floating-multiply time; everything else is completely overlapped! In particular, note that line (6) is it's own successor with respect to the floating-point unit and, because it obeys the data-dependency rule, successive instances of it can be dispatched on each cycle if the FEU.

Example 3: A Streamed Implementation

Consider a PTTM[2] implementation of floating-point, as one might use for scientific computations. For such an implementation, even the "improved" code above may be memory limited -- partly because of the load instructions themselves, and partly because the memory system is unable to exploit the

---

[1] In principle, then, WM can dispatch 5 operations each cycle since each of the integer and floating-point instructions can specify two operations.

[2] PTTM == "Pedal To The Metal"

predictability of a vector-like reference pattern. In such cases, as well as many others, streaming is just what the architect ordered.

Assuming slightly different initial register values:

```
f3 == 0.0
f4 == X
r5 == N
r6 == address of A(1)
r7 == address of B(1)
```

The streaming version of dot product is:

```
        f4 := f3                -- (1) initialize X to zero
SinF    f0, 4, r5, r6           -- (2) start streaming A(i)'s to f0
SinF    f1, 4, r5, r7           -- (3) start streaming B(i)'s to f1
Loop:   f4 := f4 + (f0*f1)      -- (4) do the actual computation
jNZ f0  Loop                    -- (5) jump on f0 count not zero
```

The two new operators here are "SinF" and "jNZ f0". "SinF f0,4,r5,r6" starts streaming in floating point values to the FIFO corresponding to f0; the stride is 4 (bytes), the count, N, is contained in r5, and the address if the first item, A(1), is contained in r6. "jNZ f0 Loop" jumps to L if the count of items not yet consumed by the loop is not zero.

Because execution of the jNZ can be overlapped with the arithmetic operations, the performance of this loop is limited only by the speed of the floating multiplier and the bandwidth of the memory system. In general, streaming mode allows WM to perform vector operations as fast as the functional units can handle the operands; in this sense it is capable of "vector performance".

Streaming is, however, more general than vectoring since the "vector operation" can be any programmed sequence of operations -- including ones involving reductions, recurrences, complex conditionals, arbitrary length vectors, etc., each of which is a problem for vector machines. In this sense WM is capable of "super vector performance". The dot-product example illustrates a reduction; the following examples will illustrate recurrences and conditionals.

Example 4: Handling Recurrences

Consider the fifth "Livermore loop", which is tri-diagonal elimination below the diagonal:

```
      DO 5 i=2,N
5     X(i)  =  Z(i)*(Y(i)-X(i-1))
```

A loop such as this cannot be vectorized because it contains a "recurrence" -- a value that is computed in one iteration and used in a subsequent one. Recurrences pose no problem for non-vector machines, including WM -- but WM can achieve vector-like performance on them. Assuming the following register assignments:

```
r3              == the address of Z(2)
r4              == the address of Y(2)
r5              == the address of X(1)
r6              == N-1
r7              == N
f0  (input)     == the stream of Z(i)'s
f1  (input)     == the stream of Y(i)'s
f0  (output)    == the resulting stream of X(i-1)'s
f3              == X(i-1)
```

The code for LLL-5 is

```
        LF      r5                  -- (1) enqueue request for X(1)
        SinF    f0, 4, r6, r3       -- (2) start streaming in the Z's
        SinF    f1, 4, r6, r4       -- (3) start streaming in the Y's
        SoutF   f0, 4, r7, r5       -- (4) prepare to stream out the X's
                f3 := f0            -- (5) initialize f3 with X(1)
        Nop                         -- (6) pause to observe the data-dependency rule
Loop:           f3 := f0*(f1-f3)    -- (7) compute X(i)
                f0 := f3            -- (8) store previous X, X(i-1)
        jNZ f0  Loop                -- (9) loop if not done
                f0 := f3            -- (10) store the last X
```

Note two things. First, at line (6) we inserted a Nop to ensure that the proper value of f3 is available to the first execution of line (7); we could have interchanged lines (4) and (5) to achieve the same effect -- but it seemed pedagogically better this way. Second, we have used the same technique as in the "improved" scalar code for dot product -- because of the data-dependency rule, line (8) causes the value in f3 *before* the computation in line (7) to be stored.

This example is typical of the code compiled for loops involving recurrences -- it involves one register for each variable involved in the recurrence, and a number of register-to-register copies to move the variables involved into the "right place". This copy code can be eliminated by unrolling the loop a number of times equal to the number of variables involved in the recurrence and "renaming" the registers on each such unrolling. The effect of this unrolling and renaming is illustrated by the following example.

Example 5: Bubble Sort

To illustrate that the utility of streaming is not limited to traditional numeric computations, as well as to exhibit the effect of loop unrolling, consider the following implementation of bubble sort[1]. Since the resulting code will be a bit tricky, we'll use a simplified version of the all-too-familiar algorithm. Using C source for grins this time, the algorithm is:

```
for (i=2; i<=n; i++)
  for (j=i; j<=n; j++)
    if (A[j-1] > A[j])  {t := A[j]; A[j] := A[j-1]; A[j-1} := t;}
```

[1] Yes, we know bubble sort is not the best kind. Yes, we know that the outer loop can be aborted if no interchanges are performed on one iteration. We suggest that the reader consider the implementation of some of these alternatives -- especially the logarithmic sorts; use of WM's two input and two output FIFO's can be quite effective!

To keep the program simpler and focus on the basic loop, we'll assume i>2 and the following initial conditions:

```
r4              == A[i-1]
r5              == A[i]
r0 (input)      set up to stream in A[i+1], A[i+2], ...
r0 (output)     set up to stream out A[i-1], A[i], ...
```

Then, with unrolling, other traditional scalar compilation algorithms, and only a little smarts about the data-dependency rule, the inner loop becomes:

```
        r4 := (r5 < r4)         -- set the 1st CC value
Loop:   jT     L5               -- is A[j-1] ≥ A[j]  ?
L4:     ---------------         -- this is the unrolled case of r4 < r5
        r4 := (r5<r0)           -- start reloading r4 and set CC for the next iteration
        r0 := r4                -- store r4, the smaller item on this iteration
        jNZ r0       Loop       -- loop if not done
        j      Xit              -- exit if input is empty
L5:     ---------------         -- this is the unrolled case of r4 >= r5
        r5 := (r4<r0)           -- start reloading r5 and set CC for the next iteration
        r0 := r5                -- store r5, the smaller item on this iteration
        jNZ r0 Loop             -- loop if not done

Xit:    ---------------         -- store the last two values in proper order
L45:    jT     L54
        r0 := r4
        r0 := r5
        j      Done
L54     r0 := r5
        r0 := r4
Done:
```

Note that the roles of r4 and r5 are interchanged in the chunks of code labeled L4 and L5. This renaming eliminates the interchange code, and the resulting code uses only two cycles per iteration.

Summary and Commentary

At one level, WM is a conventional von Neuman machine; specifically, its instructions have classic sequential semantics. WM is also unconventional in a number of respects: 2 operations/instruction, explicit relational operators, load/store operations through FIFOs, separate IEU, FEU, and IFU defined at the architectural level, and streaming. Together, these mechanisms provide a significant performance advantage at a small cost in additional hardware.

Any architecture imposes a loose limit on the performance of implementations of itself[1]. For RISC machines, that limit is one relatively simple instruction per cycle; the only question is how fast is the cycle time. There are many ways to try to exceed the RISC limit, e.g. vector machines, SIMD ma-

[1] The limit is a "loose" one because one may use various implementation tricks, e.g. "prefetch", to exceed the limit. Because such tricks are more-or-less applicable to all architectures, we ignore them for this discussion.

chines, VLIW machines, etc. WM is yet another in this list. For these machines, there are several relevant questions:

(1) what is the peak performance?
(2) what fraction of the peak performance is attainable due to
- inherent problem structure (e.g., recurrences can't be vectorized)?
- compiler limitations?

The theoretical peak performance of WM is something like 11 RISCy operations per cycle. This theoretical peak would be attained if on each cycle we could dispatch an integer instruction, a floating point instruction and a branch, and both the integer and floating instructions (a) used both operations, (b) read two operands from streamed FIFOs (therefore executing two implicit load operations), and (c) set their result into a streamed FIFO (therefore executing an implicit store operation). No realistic program will sustain this peak performance.

A definitive answer to the question of attainable performance awaits further experimentation. However, over a fairly broad set of benchmark fragments, 4-5 RISC-like instructions per cycle has been common. This is significant for several reasons. First, unlike other approaches, WM's performance does not result from, and hence does not depend upon, the applicability of a single mechanism (e.g., vectors); hence, the performance spans an assortment of applications. Second, the performance of the WM design does not depend on heroic compiler techniques; quite conventional scalar optimizations are enough. Third, the hardware implications of the design are modest.

Within reason, conserving area was not an explicit goal in the WM design -- witness the assumption of two fully functional integer ALUs, two fully functional floating ALUs, and eight FIFOs. On the other hand, it seems clear that these assumptions are modest compared to other high performance approaches, and that WM can be implemented with a small chip set (one plus floating point). Some care was expended on the bandwidth (pins) between components because, even as more area becomes available, PTTM implementations may still favor separate floating point units.

The reader may have detected a certain ambivalence in the foregoing discussion of performance: should WM be compared to a workstation-like, single-chip RISC design, or should it be compared to one or more of the many (mini)supercomputer designs?

The author's original goal was to design a 100+ MIPS[1], single-chip machine for "embedded" and "general purpose" applications written in high level languages -- notably Ada. In that sense, the appropriate comparison is to RISC machines, where the preliminary data suggests at least a 4-to-1 performance advantage, and that the goal is easily reachable. As the design evolved, however, the nature of the mechanisms (e.g. streaming) suggested comparison with supercomputer designs. Here the preliminary data is less clear, and more sensitive to implementation issues. Given "comparable" PTTM implementations, WM probably loses on the vectorizable portions of the code because the floating-point operations cannot be as deeply pipelined; it probably wins on the the non-vectorizable portions because they can still be streamed, they can still use 2 operations/instruction, etc..

In retrospect my ambivalence feels "just right". The distinction between "embedded", "general purpose", and "scientific" computation is an artificial one. It was created by the existence of computers that were better at one class of computation than another, *not* because the distinction exists crisply in real applications. WM is an example of a simple architecture that spans a broader spectrum of these classifications than previous machines.

---

[1]Whatever that means, but intuitively roughly 100 times a VAX11/780 executing "real" programs (whatever *that* means, but intuitively something closer to Linpak or Grep than Dhrystone or Puzzle).

Acknowledgements

Professor Charles Hitchcock, now at Dartmouth, participated in the initial design of WM, and contributed to a number of the ideas reported here. The author first saw the use of a FIFO interface to memory in the design of V32 by Bill Dietz and Lee Szewerenko. Anita Jones has assisted in development of WM software, and provided valuable critique of this paper. Many others, notably Ivan Sutherland and Bob Sproull, have contributed by asking the right embarrassing questions at the right time. Deep thanks to all!