The WM Family of Computer Architectures

Wm. A. Wulf
Charles Hitchcock

Computer Science Report No. TR-90-05
March 1990

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract number N00014-89-J1699.

The WM Family of Computer Architectures

Wm. A. Wulf[1]
Charles Hitchcock[2]

Abstract

This paper describes the computational aspects of a family of superscalar architectures that, for comparable area and technology, significantly outperform RISC architectures without requiring heroic compiler technology. Although the WM instruction set is apparently sequential, in fact its semantics have been carefully designed to permit concurrent execution by a collection of hardware processes that are synchronized through first-in-first-out queues, FIFOs.

Introduction

This paper outlines a family of architectures, collectively called WM, which achieve significantly greater performance than RISC designs -- often by a factor of four or more. WM is what has been called a "superscalar" architecture [Jouppi89], that is, it achieves its performance by executing more than one instruction concurrently. Unlike other superscalar designs, WM ameliorates the common problems encountered in achieving sustained concurrent execution of instructions by unusual approaches to the semantics of its basic instruction set. The resulting charms of the design are the simplicity of the mechanisms used and the synergy of their interaction, its modest demands on compiler technology, its broad spectrum of applicability, and, most of all, the fact that the complexity of an implementation is comparable to that of RISC machines.

WM is a family of architectures in the sense that it is parameterized by the size (width) of the arithmetic operations supported by a member of the family. $WM_{<n,m>}$ supports integer arithmetic of width n-bits and floating point arithmetic of width m-bits directly in hardware. Thus, for example, $WM_{<16,0>}$ is a 16-bit (integer only) microcomputer, while $WM_{<64,64>}$ is likely to be a high performance machine with a large 64-bit virtual address space and 64-bit floating point arithmetic.

The following sections briefly describe the WM architecture, focusing on its computational features; support for the operating system aspects of the design is left for another paper. The full definition of WM may be found in [Wulf90]. Then we provide a few examples and use these to provide an intuitive feel for the machine's performance. Finally we conclude with some general remarks on the design.

[1] National Science Foundation (on leave from The University of Virginia)

[2] Dartmouth College

General Information

The core of WM is a relatively conventional, 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 (r0-r31) and 32 floating-point registers (f0-f31). One register in each of the integer/logical and floating-point sets (r31 and f31, also called rZ and fZ) are "always zero"; stores into these registers do not affect their value. A few other registers have special purposes that will be discussed later.

Members of the WM family support 8-, 16-, 32-, and 64-bit integer and logical and 32- and 64-bit floating point data types in memory. As noted above, the size (width) of the registers, and hence arithmetic operations, is a parameter of the architecture; for $WM_{<n,m>}$ n and m are constrained so that n must be 16, 32, or 64, and m must be 0, 32, or 64. Memory data types are widened as necessary when loaded from memory and shortened as necessary when stored. Memory types wider than the register width of a particular family member are not supported.

All WM instructions are exactly 32-bits, and must be word-aligned; there are five defined instruction formats, as shown in Figure 1. Instructions are executed by one of the three, unsynchronized, but coupled 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 (possibly in cooperation with one, or both, of the other execution units). An instruction may be dispatched to each of these components on each cycle.

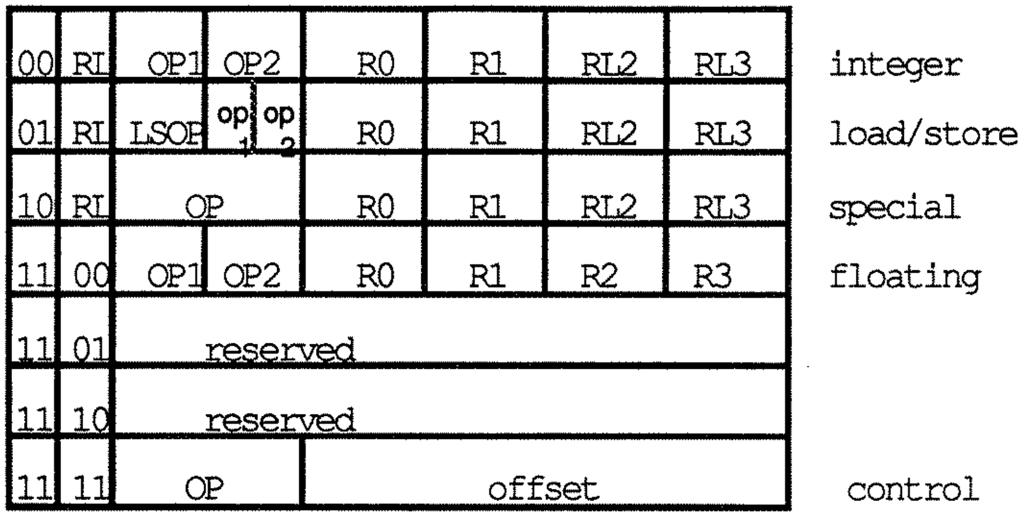
00 RI OP1 OP2 R0 R1 RL2 RL3 integer
01 RI LSOP op 1 op 2 R0 R1 RL2 RL3 load/store
10 RI OP R0 R1 RL2 RL3 special
11 00 OP1 OP2 R0 R1 R2 R3 floating
11 01 reserved
11 10 reserved
11 11 OP offset control

Figure 1: WM Instruction Formats

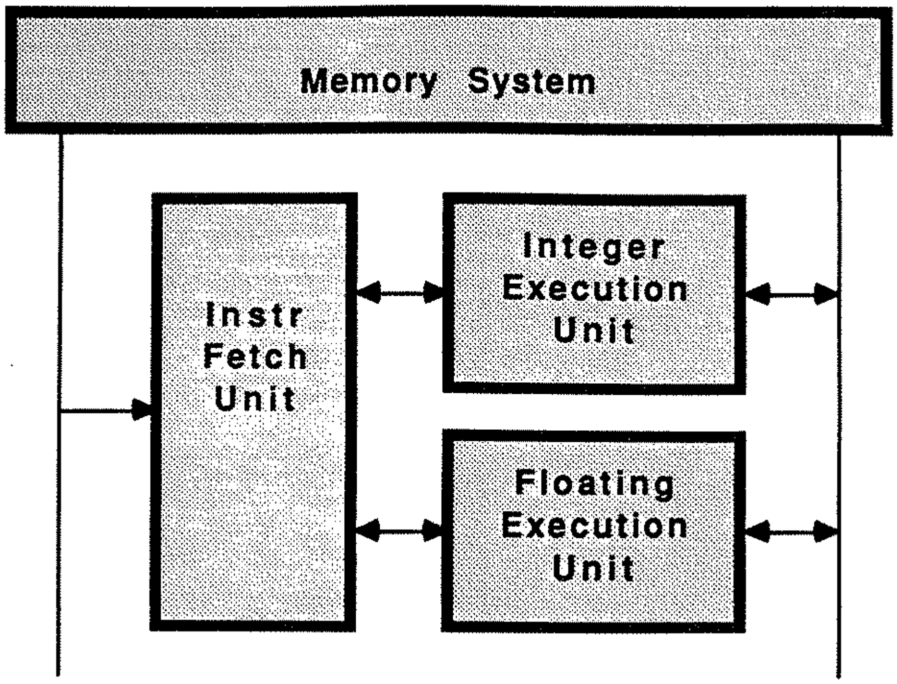

Figure 2: The IFU/IEU/FEU Decomposition

The semantics of the WM instruction set are conceptually sequential; the instruction at location *(I)* is executed before the instruction at location *(I+4)*. However, the design has been partitioned to permit greater concurrency than is suggested by this statement. The principal state shared between primary memory, 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 individually for the three units, 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.

To support the concurrent execution of instructions, the Instruction Fetch Unit (IFU) has the conceptual structure suggested by Figure 3. Instructions are fetched sequentially from memory and are either enqueued for execution by the IEU, enqueued for execution by the FEU, or are executed by the IFU itself. In some cases synchronization with one or more of the execution units may be required -- as in an instruction to convert an integer value (in an integer register) to a floating point value (in a floating point register). In such cases the hardware may have to stall until the relevant queues have been drained. In the vast majority of cases, however, instructions are dequeued by the IEU, FEU, and IFU at a rate determined independently by their respective implementations and computations.

Concurrent execution of operations from a conceptually sequential instruction stream is the key to the performance of the *implementation* of many computers; it is the essence, for example, of the notion of pipelining. The degree of concurrency that can be achieved, however, is limited by architectural aspects of those computers -- notably conditional branches and load/store operations, whose outcome must be known before subsequent operations can be performed. The key aspect of the WM design is a collection of mechanisms that permit the inherent concurrency of an application to be exploited. These mechanisms are the primary subject of the following sections.

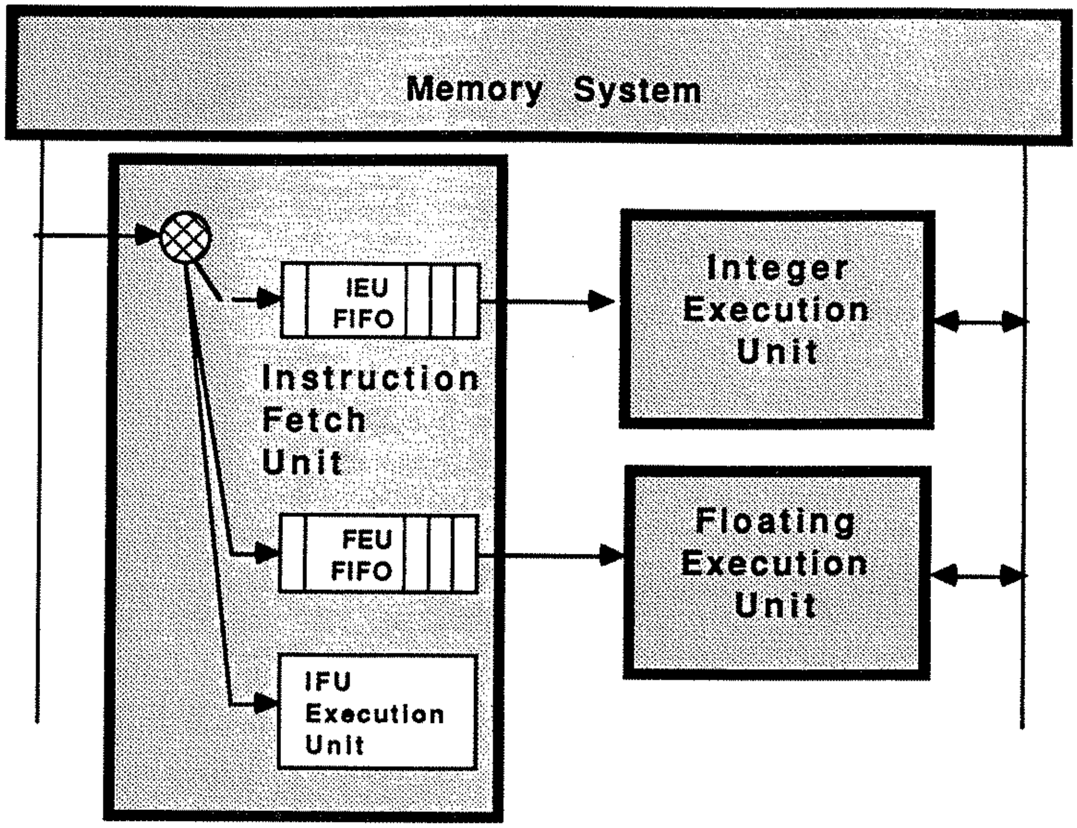

Figure 3: Conceptual Instruction Fetch Unit

The Integer Instructions

The integer instructions evaluate the assignment:

$$R0 := (O1 \textbf{ op1 } O2) \textbf{ op2 } O3$$

That is, they perform two operations on three source operands and place the result in a destination register. The first source operand, O1, must be the contents of a register; the other two operands, O2 and O3, may be either the contents of one of the integer registers or 5-bit unsigned literals from the instruction word. Referring back to Figure 1, note the 2-bit field RL -- it specifies whether the contents of the operand fields RL2 and RL3 name registers or literals.

Figure 4 illustrates the conceptual model of the IEU. The machine is pipelined so that an instruction can be dispatched to the IEU each cycle: while **op2**, the outer operation of one instruction is being executed in ALU2, **op1**, the inner operation of the next instruction is being executed in ALU1. This gives rise to the data-dependency rule:

> The result of one instruction is not available as an operand of the inner operation (**op1**) of the next instruction. Moreover, the inner operands are guaranteed to have the values extant prior to the execution of the previous instruction.

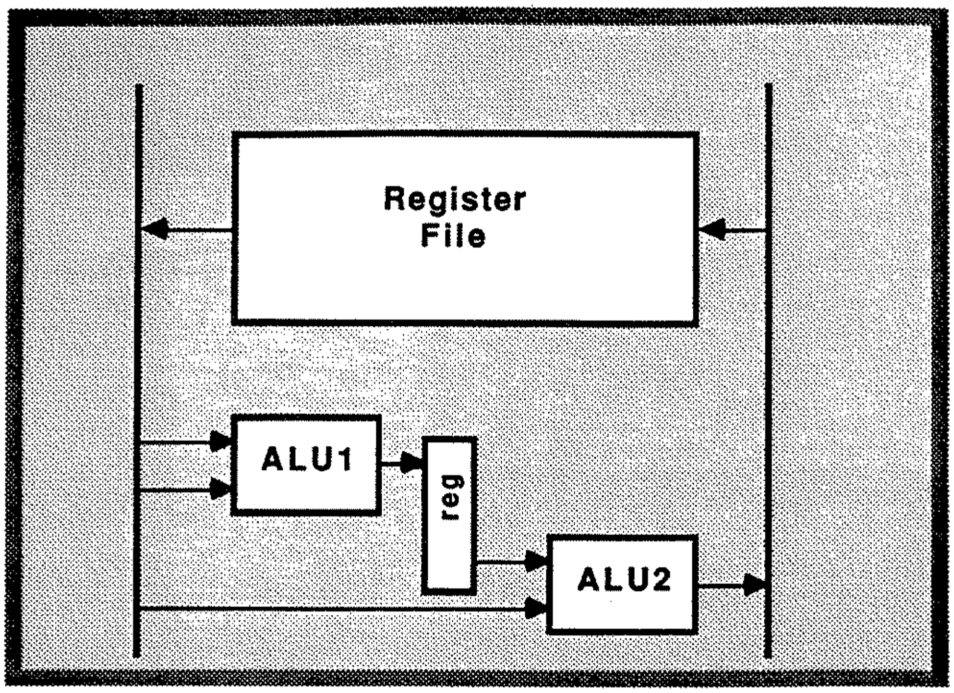

Figure 4: The IEU Conceptual Model

There are 16 operation codes for each of **op1** and **op2**, as listed in Table 1, and with the few exceptions noted below, they are what one would expect. The major non-commutative operators, subtract and divide, have "reverse" forms (denoted with primes, e.g., -' is reverse subtract) to facilitate encoding arbitrary arithmetic expressions from the source program. No unary operations are included since they can be synthesized from binary operations with zero (literal zero or rZ). The boolean "equivalence" operator was

operation	explanation
+	add
-	subtract
-'	reverse subtract
*	multiply
/	divide
/'	reverse divide
<<	left shift (arithmetic)
and	bitwise "and"
or	bitwise "or"
eqv	bitwise "equivalence"
=	equal
<>	not equal
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Table 1: Integer Operations

included rather than the more common "exclusive or" since it can be used to implement complementation (not x == x eqv 0).

There are six relational operators; these operators have four 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 overlap the execution of conditional jumps with that of other instructions.

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

r10 := (r10 < r11) + 1

compares the value in r10 to the value in r11 and generates a condition code bit, increments the value of r10, and stores this new value back into r10. This is the "compare-and-add" portion of the the typical "compare-add-and-branch" loop control paradigm.

If two relationals occur in the same instruction, the resulting condition code is the logical "and" or "or" of the two -- which depends upon a program settable bit in the Program Control Word. This permits, for example, range checking in a single instruction.

If the condition code resulting from an instruction containing one or more relationals is false, the store into the destination register is not performed, nor are any exceptions resulting from the operations raised. Thus, the previous example

r10 := (r10 < r11) + 1

will increment r10 only if its previous value is less than the value in r11.

Once more, remember that the precision of the arithmetic performed is a parameter of the architecture; generally programs written for $WM_{<n,m>}$ will not execute properly on $WM_{<p,q>}$ if n is not equal to p or m is not equal to q.

The Floating Point Instructions

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

F0 := (O1 **op1** O2) **op2** O3

Note that no literals are supported for the floating point instructions (there is no RL field for them; see Figure 1).

The floating point operations are also what one would expect: floating add, subtract, etc. As with the integer instructions 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; thus Figure 4 is the proper conceptual model for both the IEU and FEU. The precision of the arithmetic supported is, as for the integer instructions, a parameter of the architecture.

The Control Instructions

The control instructions perform PC-relative jumps, and contain a 20-bit signed displacement. 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^{20} instructions and wish to jump from beginning to end.

The most interesting control instructions are the conditional jumps, since they are a major inhibitor of concurrency in traditional architectures; 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 (single bit) condition-code value; conditional jumps "consume" one. As with all producer-consumer systems, we can synchronize the two participants by FIFOs -- one for the integer CC's and one for the floating CC's. This means that the IFU is free to execute a conditional jump as soon as a condition code value becomes available, i.e., the relevant CC FIFO is non-empty. Thus, if the compiler separates instructions containing relational operators from the corresponding conditional jump by at least one instruction, the execution of the the jump can be overlapped with that of other instructions. Indeed, with sufficient separation, the conditional jump can be executed well ahead of intervening computational instructions, thus facilitating "pre-fetching" instructions from the target of the jump.

Several things should be noted about this scheme. First, the above rule isn't really a restriction on real programs; a compiler won't generate a relational unless it intends to test the resulting condition. Second, the problem of moving relationals is similar to that of filling "branch shadows" in "delayed branch" or "branch with execute" schemes. Third, in practice the ability of the compiler to perform such motions is significantly enhanced by the fact that other operators do not set the condition codes. Finally, the existence of conditional branches has been one of the most serious impediments to significant mico-concurrency [Rise72]; this scheme all but eliminates this problem, as will be illustrated by later examples.

The Load/Store Instructions

Although similar mechanisms have been used before [Smith87], the method of performing loads and stores is another unusual aspect of the WM design, and is another source of its ability to execute several instructions concurrently. The scheme has several important differences from conventional load/store operations:

- 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 inserted into the output

FIFO by specifying register zero as the destination of a data-manipulation instruction.

- In WM, only the memory address is specified in the load/store instructions. The target of a load (source of a store) is implicitly one of the FIFOs.
- WM has no "addressing modes" per se; rather, addresses are computed using a subset of the normal integer operations.

As noted above, load/store instructions specify (only) an address. They do so by performing a computation syntactically and semantically identical to the integer instructions. That is, they compute the assignment

$$R0 := (O1\ \mathbf{op1}\ O2)\ \mathbf{op2}\ O3$$

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. They are add, subtract, left shift, and multiply -- which are the most common operations in addressing expressions.

The operand O1 must be a register, and the operands O2 and O3 may be either a register or a literal, just as for the integer instructions. The data paths, ALUs, and data dependency rule are the same; there is no special "address generation" unit in the architecture. Only during the assignment to the destination register 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 the register specified by R0, 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 determines the type of load or store to be done, e.g., load an 8-bit byte vs. load a 32-bit word. This also implicitly determines which FIFO is involved -- those in the IEU (integer load/stores) or FEU (floating load/stores). The options for LSOP's are given in Table 2.

LSOP	explanation
L8i	load 8-bit integer (no sign extension)
L8ix	load 8-bit integer (with sign extension)
L16i	load 16-bit integer (no sign extension)
L16ix	load 16-bit integer (with sign extension)
L32i	load 32-bit integer (no sign extension)
L32ix	load 32-bit integer (with sign extension)
L64i	load 64-bit integer
L32f	load 32-bit floating point (with extension)
L64f	load 64-bit floating point
S8i	store 8-bit integer (low bits only)
S16i	store 16-bit integer (low bits only)
S32i	store 32-bit integer (low bits only)
S64i	store 64-bit integer
S32f	store 32-bit floating point
S64f	store 64-bit floating point

Table 2: Load/Store Operations

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 the register specified by R0, the set also includes a generalized form of "auto-increment".

An extreme example of the use of WM's load/store instructions is illustrated by the code to perform the computation "A = B+C+D", which is

```
 L32i rZ := <address computation for B>    -- enqueue 32-bit integer from location B
 L32i rZ := <address computation for C>    -- enqueue 32-bit integer from location C
 L32i rZ := <address computation for D>    -- enqueue 32-bit integer from location D
 S32i rZ := <address computation for A>    -- enqueue address of 32-bit integer, A
      r0  :=  (r0+r0)+r0                   -- dequeue source operands, and
                                           - -   perform the arithmetic, and
                                           - -   enqueue the result to be stored, and
                                           - -   trigger the actual store
```

A request to load a value smaller than the arithmetic width of a member of the WM family will cause the resulting data item to be expanded to the proper width, e.g., on $WM_{<32,32>}$ a L8i will cause the named 8-bit byte to be padded with 24 leading zeros and L8ix will cause the byte to be sign extended. Loads or stores specifying a data item wider than that supported by a family member are illegal and will raise an exception.

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. Both of these properties help to overcome another major impediment to micro-concurrency in conventional architectures. A later example demonstrates the importance of this feature.

The WM *architecture* requires that the input FIFOs must contain at least three elements, and the output FIFOs must have at least one element; this is sufficient to guarantee that an instruction of the form

```
r0 := (r0 op1 r0) op2 r0
```

can always be executed. An *implementation*, however, may have more elements per FIFO and doing so will generally lead to better performance by allowing greater freedom to mask memory latency. A program is invalid if it presumes more FIFO elements than are present on a given implementation; programs presuming more elements than are present will deadlock. In general, compilers will need to be parameterized by the FIFO sizes in order to take the fullest advantage of the hardware.

The 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 include the instructions that control an important feature of WM, "streaming" that will be discussed below.

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

```
Sin32i        fifo, base, count, stride
```

initiates "input streaming mode", in this case, a sequence of 32-bit integer loads. The effect of this instruction is similar to that of "count" loads being 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.

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 (r0 and r1) and the FEU (f0 and f1). Therefore, the "start streaming" operations name which FIFO is to be used, as shown above.

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". Strictly, jNZ performs the jump if the number of operands dequeued *or scheduled to be dequeued* is less than the number specified by the "start streaming" operation.

A simple example of the use of streaming is vector assignment -- copying a specified number of (32 bit) words from one area of memory to another; assuming that r6 and r7 contain the addresses to the source and destination locations respectively, and r8 contains the number of words to be copied, then the code for this operation is:

```
      Sin32i  r0, r6, r8, 4     -- (1) start streaming in
      Sout32i r0, r7, r8, 4     -- (2) start streaming out
   L: r0 := r0                  -- (3) copy from input to output stream
      jNZ r0  L                 -- (4) loop if not done copying
```

The essence of this example is lines (3) and (4). After initializing the streaming operations, line (3) merely dequeues one operand from the input FIFO and enqueues it in the output FIFO; the load and store operations are asynchronously performed by the stream control units. Line (4) tests whether all of the data items have been processed, and can be overlapped with the execution of line (3). Thus one memory-to-memory copy can be dispatched each cycle (assuming an adequately fast memory system).

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, makes the load/store operations "asynchronous processes", and signals predictable reference patterns that can be exploited by the memory system. Unlike vector machines, however, 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 the FIFOs as source/destination operands.

Examples and Rationale

Let's consider several versions of a "dot product" example to illustrate the basic operation of the machine -- and, specifically, to illustrate its micro-concurrency potential. 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:

```
        r4 == i                     f4 == X
        r5 == N
        r6 == address of A(0)
        r7 == address of B(0)
```

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, the assembler uses 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 :=  fZ                  -- (2) initialize X to zero
Loop: L32f   rZ := (r4<<2) + r6  -- (3) load A(i) into FIFO f0; discard the address
      L32f   rZ := (r4<<2) + r7  -- (4) load B(i) into FIFO f0; discard the address
      f4 :=  (f0*f0) + f4        -- (5) compute the partial sum
      r4 :=  (r4+1) < r5         -- (6) increment and test i
      JiT    Loop                -- (7) loop if not done
```

First, let's explain each line of the example.

(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 and three operands; as a notational convenience they needn't be written and the assembler inserts appropriate "nops".

(2) This assigns the floating point zero value to f4 from the "always zero" register, fZ.

(3) This is a load-floating (L32f) instruction that computes the address of A(i) by using "scaled indexing". Since the address is not needed later, it is stored into the "always zero" 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.

(5) This is the actual computation of X = A(i)*B(i) + X. Note that there are 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 left operand -- the incremented value of r4 ("i") in this case. Note that, because of the rule governing stores and exceptions in instructions containing relationals, "i" will have the value N on exit and an overflow will not occur even if N is the largest positive integer.

(7) This is the branch ("Jump on integer 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 the authors -- but it is pervasive. Obviously, some number of expressions of this form "occur naturally" in any program, especially below the level most apparent to the high-level language programmer. In addition, however, many common idioms have this form. For example, "multiply-and-add" is the central computation in many scientific computations, for example:
- dot product,
- Horner's rule (polynomial evaluation), and
- LU decomposition,

But it is not just scientific computations that have this form, there are quite a number of common "system-y" idioms of this form as well; for example:
- addressing expressions (double indexing, scaled indexing, etc.),
- field extraction (e.g., shift-mask for unsigned extraction),
- increment-and-test,
- range checking, and
- stack-adjustment with limit-testing,

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 can be dispatched and executed at RISC machine rates.

Second, note the effect of decoupling of the integer and floating point units, as previously described. The first execution of line (5), the actual dot product computation, will likely be delayed some amount of time until the values of A(1) and B(1) 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, a number of floating point operations may be enqueued for execution, but the integer portion of the loop can continue execution and can happily be many iterations ahead of the floating-point portion. In addition to allowing concurrency, this means that the probability of a cache miss delaying subsequent floating point operations is reduced.

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 :=  fZ                      -- (2) initialize X to zero
Loop: L32f   rZ := (r4<<2) + r6      -- (3) load A(i) into FIFO f0
      r4 :=  (r4+1) < r5             -- (6) increment and test i
      L32f   rZ := (r4<<2) + r7      -- (4) load B(i) into FIFO f0
      f4 :=  (f0*f0) + f4            -- (5) compute next term
      JiT    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

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

and, because it obeys the data-dependency rule, successive instances of it can be dispatched on each cycle of the FEU.

Example 3: A Streamed Implementation

Consider a PTTM ("pedal to the metal") implementation of floating-point, as one might use for a machine intended 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 predictability of a vector-like reference pattern. In such cases, as well as many others, streaming is just what the user ordered.

Assuming slightly different initial register values:

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

The streaming version of dot product is:

```
          f4 :=  fZ                   -- (1) initialize X to zero
          Sin32f      f0, r6, r5, 4   -- (2) start streaming A(i)'s to f0
          Sin32f      f1, r7, r5, 4   -- (3) start streaming B(i)'s to f1
    Loop: f4 :=  (f0*f1)  +  f4       -- (4) do the actual computation
          jNZ f0  Loop                -- (5) jump on f0 count not zero
```

This version of dot product is similar to the vector move example given earlier except, of course, that two input FIFOs are used and the computation in the loop is more interesting. Note specifically that because execution of the jNZ can be overlapped with the arithmetic operations, the performance of the 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". Note, however, that WM (or other scalar machines) do not obviously benefit from a deeply pipelined floating point unit, so the dispatch rate may be slower than for vector machines.

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, with the same caveat as above, 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 (in this case X(i) is defined in terms of X(i-1), which was computed on the immediately previous iteration). Recurrences pose no problem for non-vector machines, including WM, but WM can achieve vector-like performance on them. Assuming the following register assignments:

```
r 3                == the address of Z(2)
r 4                == the address of Y(2)
r 5                == the address of X(1)
r 6                == N-1
r 7                == 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

```
        L32f         r 5                    -- (1) enqueue request for X(1)
        Sin32f       f0, r3, r6, 4 -- (2) start streaming in the Z's
        Sin32f       f1, r4, r6, 4 -- (3) start streaming in the Y's
        Sout32f      f0, r5, r7, 4 -- (4) prepare to stream out the X's
        f3 :=        f0                     -- (5) initialize f3 with X(1)
        FNOP                                -- (6) pause for the data-dependency
Loop:   f3 :=        (f1-f3)  *  f0 -- (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
```

Two things are worth special mention in this example. First, at line (6) we inserted a floating "nop", FNOP, to ensure that the proper value of f3 is available as the inner operand to the first execution of line (7); the FNOP could be "fZ := (fZ+fZ)+fZ", for example. Since there are no other floating point operations prior to entry to the loop, there is no way to move the "f3 := f0" that will not make it adjacent to the instruction at "Loop".

Second, we are preserving the invariant that at the head of the loop f3 contains the value of X(i-1); this value is both used in computing the new X(i) and is the value stored on that iteration. In the loop, 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. Since X(i-1) is stored on each iteration, one final store is done outside the loop to store X(N).

This example is typical of the code compiled for loops involving recurrences -- it involves one register for each variable involved in the recurrence. These registers need to be initialized before the loop, and stored after the loop. A number of register-to-register copies are done inside the loop to move the variables involved into the "right place", but 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 compile time technique for detecting the potential for streaming is a simple variant of that used to detect strength reduction; specifically it does not require data-dependency analysis.

Example 5: UNIX String Compare

To illustrate that the utility of streaming is not limited to traditional numeric computations, consider the following implementation of the UNIX string comparison utility. The algorithm is:

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

Assuming that the PCW has been set to "or" the results of two relationals per instruction, and the following register assignments,

```
r4    ==    2**31-1 (the largest positive integer)
r5    ==    S1 (address of 1st string)
r6    ==    S2 (address of 2nd string)
r10   ==    temporary (holds *S2 on exit from the loop)
r12   ==    counter (holds index of characters being scanned)
```

The code, omitting the procedure entry/exit sequences, is shown below. This code uses one instruction not discussed previously, "StopAll". This instruction merely stops all streaming and clears the FIFOs of any data that might be there from the streaming operation(s). It is useful in situations such as this one where the loop termination condition is a property of the data being streamed.

```
      r12 :=   0                 -- (1) set counter to zero
      Sin8i    r0, 1, r4, r5     -- (2) stream in string pointed to by S1
      Sin8i    r1, 1, r4, r6     -- (3) stream in string pointed to by S2
Loop: r10 :=   (r0 = 0) <> r1    -- (4) (*S1 = 0) OR (*S1 <> *S2)
      r12 :=   r12+1             -- (5) up the counter
      JiF      Loop              -- (6) loop if above condition is false
      StopAll                    -- (7) stop all streaming
      LB       rZ := (r6-1)+r12  -- (8) load *S2
      r10 :=   (r10 - r0)        -- (9) compute return value, (*S1-*S2)
                                 - -   >> note, *S1 in r10 from line (4)
```

The heart of this code is lines (4)-(6). Line (4) serves two functions; first it tests for the end condition, and second when the end is reached it captures one of the string characters, namely *S1, in r10. Since it would be more expensive to keep both characters on each iteration, we instead keep the index of the characters being tested in r12, and reload *S2 when the end of the loop is reached. Note that the increment of r12 in the loop is, in a sense, "free". That is, if the test and jump were adjacent the jump could not be overlapped.

Preliminary Performance Indication

At best comparing the performance of computers fairly is a difficult and application dependant process; comparing architectures rather than implementations adds another dimension to the difficulty and uncertainty of the task. Therefore we have chosen to defer an in-depth analysis of WM performance to another paper in order to deal completely with the issue. Nonetheless, some indication of performance relative to other architectures is desirable.

To provide this indication we have hand-coded the three examples presented above (dot product, LLL5, and string compare) in MIPS-X [Chow86] in a careful, "no holds barred" manner. MIPS-X was chosen because it is an especially clean, prototypical RISC design. "No holds barred" hand coding was done to avoid comparing compilers rather than architectures.

We then made two unrealistic assumptions, namely that all operations take a single cycle and that there are no cache misses. Although unrealistic, these assumptions allow us to compare the best performance that one could expect from any implementation of the architectures. The results are shown in Table 3, below.

The examples in this paper, of course, were chosen to illustrate the features of WM; it's not surprising that it performs well on them. To illustrate the other extreme we also coded an example that is very unfavorable to WM, namely a search through a linked list of records for that record with the largest value of an integer field. This example is the worst we know of for WM; it cannot be streamed, it has no floating point operations to overlap with integer ones, and it cannot use two operations in any of its instructions. The results for this example are also shown in Table 3, under the assumption that half of the records scanned will have a value greater than that seen previously. The code for all examples is contained in an appendix.

	cycles/loop		
Example	MIPS-X	WM	ratio
dot product (streamed)	7	1	7
LLL5	9	2	4.5
string compare	6.75	2	3.38
linked-list search	7	4.5	1.55

Table 3

The fractional (6.75) result for the MIPS-X coding of string comparison arises from the fact that, since MIPS-X does not support byte loads, word loads and shift-mask operations are done to extract the string characters. Only one load per 4 bytes was done, and no shifting is required for one of the characters. Thus 6.75 is the average per byte. The fractional result for WM on the linked-list search arises from averaging the paths; on MIPS-X, a clever use of "squashed" branches made both paths have identical lengths.

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 can be viewed as supporting a software-like collection of processes synchronized through a set of message buffers (FIFOs).

Any architecture imposes a loose limit on the performance of implementations of itself -- the limit is a "loose" one because one may use various well-known implementation tricks to exceed the limit in special cases. Ignoring implementation tricks for the moment, the limit for RISC architectures 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, such as vector machines, SIMD machines, and VLIW machines. WM can be viewed as 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?

(3) what are the hardware costs to achieve these performance levels?

The theoretical peak performance of WM is 13 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 three operands from streamed FIFOs (therefore executing three 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 most 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.

The observed concurrency in WM programs is higher than that reported in other studies of realistic superscalar designs [e.g., Jouppi89 and Smith89]. The primary reason is that these studies make a number of implicit assumptions about the semantics of instruction sets and their interpretation. WM eliminates many such assumptions (as many as we are aware of), for example by placing the restriction on the production and consumption of condition code bits. The net effect is that the data and control dependencies in a WM program more closely model those inherent in the original algorithm.

Within reason, conserving chip 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), and that a single chip implementation will be feasible shortly. Some care was expended on the bandwidth (pins) between components, however, 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 authors' 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 non-

[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).

vectorizable portions because they can still be streamed, they can still use 2 operations/instruction, etc.

In retrospect the 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.

Acknowledgements

Professors Anita Jones, Jim Aylor, Jack Davidson, Andrew Grimshaw, and Barry Johnson, and graduate students Ricky Benitez, Peter Kester, Steve McCalla, Sunil Pamidi, and Max Salinas are the core of the WM team at The University of Virginia and have made many valuable contributions to the WM design, as well as to the simulator and compiler implementations. Ivan Sutherland and Bob Sproull asked the right, sometimes embarrassing, questions at just the right times. This work was supported in part by DARPA, under contract number N00014-89-J1699 through the Office of Naval Research.

References

[Chow86] Chow, P., "MIPS-X Instruction Set and Programmer's Manual", Technical Report No. CSL-86-289, Stanford University, May 1986.

[Jouppi89] Jouppi, N. P., and Wall, D. W., "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines", Third International Conference on Architectural Support for Programming Languages and Operating Systems, April 1989.

[Rise72] Riseman, E. M., "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Transactions on Computers, Vol. C-21, December 1972.

[Smith87] Smith, J. E., et. al., "The ZS-1 Central Processor", Second International Conference on Architectural Support for Programming Languages and Operating Systems, October 1987.

[Smith89] Smith, M. D., Johnson, M., and Horowitz, M. A., "Limits on Multiple Instruction Issue", Third International Conference on Architectural Support for Programming Languages and Operating Systems, April 1989.

[Wulf90] Wulf, W. A., "The WM Computer Architectures; Principles of Operation", Technical Report No. TR-90-2, University of Virginia, January 1990.

Appendix -- Code Examples

The following is the code for the five example programs discussed in the section on "Preliminary Performance Indication". Only the inner loop code is shown; appropriate register (or other) initialization is presumed to have been done. The MIPS-X code is not in MIPS assembler format, but is intended to be readable.

As with any such hand-coded examples, there may be better codings that the authors were authors were insufficiently clever to invent. Our best efforts are included here so that the reader may judge them.

Dot Product

Source Code

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

WM Assembly Language

```
    Loop: f4 :=  (f0*f1) + f4 -- (4) do the actual computation
          jNZ f0 Loop         -- (5) jump on f0 count not zero
```

MIPS Assembly Language

```
    L:    ldf f1 := 0(r1)     -- get A(i)
          ldf f2 := 0(r2)     -- get B(i)
          addi r1 := r1+4     -- incr ptr to A(i)
          addi r2 := r2+4     -- incr ptr to B(i)
          bleg r1<=r3, L      -- test end; note next 2 instr in shadow
          fmul f1 := f1*f2    --   A(i)*B(i)
          fadd f3 := f3+f1    --  x  :=  x+(A(i)*B(i)
```

LLL5

Source Code

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

WM Assembly Language

```
    Loop: f3 :=  (f1-f3) * f0    -- (7) compute X(i)
          f0 :=  (f3)            -- (8) store previous X, X(i-1)
          jNZ f0 Loop            -- (9) loop if not done
```

MIPS Assembly Language

```
    L:    ldf f1 := 0(r1)     -- load Z(i)
          ldf f2 := 0(r2)     -- load Y(i)
          adi r1 := r1+4      -- next pointer to Z(i)
          adi r2 := r2+4      -- next pointer to Y(i)
          stf 0(r3) := f3     -- store previous X(i)
          adi r3 := r3+4      -- next pointer to X(i)
          bleq r1 <= r4, L    -- again, next 2 are in shadow
          fsub f2 := f1-f2
          fmul f3 := f1*f2
```

String Compare

Source Code

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

WM Assembly Language

```
Loop:  r10 := (r0 = 0) <> r1   -- (4) (*S1 = 0) OR (*S1 <> *S2)
       r12 := r12+1            -- (5) up the counter
       JiF     Loop            -- (6) loop if above condition is false
```

MIPS Assembly Language

```
L:     ld  r1 := 0(r2)        -- get next WORD of bytes from *S1
       ld  r3 := 0(r4)        -- get next WORD of bytes from *S2
La:    adi r2 := r2+4
       adi r4 := r4+4
b1a:   rot r5 := r1 << r6     -- rotate 1 byte
       and r5 := r5 and r7    -- mask off low byte; get *S1
       bne r5 <> 0, b1b       -- test end condition; note next 2 NOT sq'd
       rot r8 := r3 << r6
       and r8 := r8 and r7    -- mask off low byte; get *S2
       jump done              -- exit loop if *S1 == 0
b1b:   bne r5 == r8, done     -- note, squash next two if no jump
b2a:   rot r5 := r1 << r9     -- rotate 2 bytes
       and r5 := r5 and r7    -- mask off low byte; get *S1
        bne r5 <> 0, b2b      -- test end condition; note next 2 NOT sq'd
       rot r8 := r3 << r9
       and r8 := r8 and r7    -- mask off low byte; get *S2
       jump done              -- exit loop if *S1 == 0
b2b:   bne r5 == r8, done     -- note, squash next two if no jump
b3a    rot r5 := r1 << r10    -- rotate 3 bytes
       and r5 := r5 and r7    -- mask off low byte; get *S1
       bne r5 <> 0, b3b       -- test end condition; note next 2 NOT sq'd
       rot r8 := r3 << r10
       and r8 := r8 and r7    -- mask off low byte; get *S2
       jump done              -- exit loop if *S1 == 0
b3b:   bne r5 == r8, done     -- note, squash next two if no jump
b4a:   and r5 := r1 and r11   -- mask off low byte; get *S1
       bne r5 <> 0, b4b       -- test end condition; note next 2 NOT sq'd
       nop
       and r8 := r3 and r7    -- mask off low byte; get *S2
       jump done              -- exit loop if *S1 == 0
b4b:   beq r5 == r8, La       -- yea, try next word of bytes
       ld  r1 := 0(r2)        -- get next WORD of bytes from *S1
       ld  r3 := 0(r4)        -- get next WORD of bytes from *S2
done: ...
```

Linked-List Search

Source Code

```
for(; rp!=0; rp=*rp) { if (*(rp+4) > m) {rpm=rp; rm=*(rp+4);}
```

WM Assembly Language

```
L:      L32i    (rp+4)       --   (1)     load data value
        L32i    (rp)         --   (2)     load next pointer
        rm :=   (r0 > rm)    --   (3)     test if new value is larger; save it if so
        rp :=   (r0 <> 0)    --   (4)     test if next pointer is nil; save it if not
        JiT     T            --   (5)     test if new value was larger;
        rpm := (rp)          --   (6)     save current record pointer;
T:      JiT     L            --   (7)     loop if new pointer non-nil
                             -- Note that in line (6) we exploit the data
                             -- dependency rule to get the value of rp
                             -- prior to the assignment in line (4)
```

MIPS Assembly Language

```
L:      ld      r2 := 4(r3)  --   (1)     load data value
L2:     ld      r1 := 0(r3)  --  (2)      load next pointer
        bgtsq   r2, r4, nb   --   (3)     squash next two if not larger
        mv      r5 := r3     --   (4)     save pointer to record with largest value
        mv      r4 := r1     --   (5)     save largest value
nb:     bnesq   r1, 0, L2    --  (6)      loop if next pointer non-nil
        mv      r3 := r1     --   (7)     move pointer to proper register
        ld      r2 := 4(r3)  --   (8)     load next data value
                             -- Note that the branch in line (3) is the opposite
                             -- of what one might expect; it jumps if the
                             -- current record value is larger than any seen so
                             -- far. This allows the next two instructions to be
                             -- executed in this case.
```