THE WM COMPUTER ARCHITECTURE DEFINITION AND RATIONALE
Version 2

Wm. A. Wulf

Computer Science Report No. TR-88-22
February 22, 1989

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

The WM Computer Architecture Definition and Rationale Version 2

Wm. A. Wulf

Computer Science Department
University of Virginia
Charlottesville, Va.

22 February, 1989

Table of Contents

Abstract 1
Acknowledgements 1
Chapter 1. General Features 2
Chapter 2. Logical Machine Structure 5
2.1. Addresses & Data 5
2.2. Integer and Floating Point Execution Units 7
2.3. Memory Reads & Writes 9
2.5 IO 14
Chapter 3. Instruction Descriptions 16
3.1. Integer Arithmetic & Logical Instructions 16
3.2. Load & Store Instructions 18
3.3. Control Flow Instructions 21
3.4. Floating Point Instructions 23
3.5. Special Instructions 25
Chapter 4. Task, Program, and System Framework 32
4.1. Task State 34
4.2. Protection Table 36
4.3. Map Table 37
4.4. Traps (Exceptions) and Interrupts 38
4.5. Devices 40
Appendix A. Summary of Restrictions on Valid Software 41
Appendix B. 43
B.1. Integer Format and Operation Encodings 43
B.2. Load/Store Format and Operation Encodings 43
B.3. Floating Point Format and Encodings 44
B.4. Control Format and Operation Encodings 44
B.5. Special Format and Operation Encodings 45
Appendix C. WM Assembly Language 47
C.1. Lexical Structure 47
C.2. Instructions 48
Appendix D. Streaming Versus Vectorizing 52
D.1. The Issue 52
D.2. An Example, Matrix Multiply 53
D.3. Another Example, an IIR Filter 53
D.4. Another Example, FFT Inner Loop 54
D.5. Other Applications of Streaming 56
Appendix E. Rationale Behind Chosen Operation Sets 57
E.1. Integer & Logical 57
E.2. Floating Point 58
E.3. Load/Store Instructions 58
E.4. Control Instructions 58
E.5. Special Instructions 59
Appendix F. Constructing WM Software Suggestions and Rationale 62
F.1. Calling Sequence and Register Conventions 62
F.2. Ada Exceptions 63
F.3. Parameter Passing 64

F.4. Loop Control.....64
F.5. The Display.....65
F.6. Absolute Addresses.....65

Abstract

This report is a preliminary definition of, and rationale for, the WM computer architecture. It is not quite a rigorous architectural definition, nor a thoroughly motivated tutorial. Rather it is something in between. Its purpose is to put a stake in the ground; we expect there will be changes as we implement the machine, but this is the point from which the implementation will proceed. Thus we attempt here to capture not only the current design, but something of the philosophy and rationale behind the decisions in the design -- so that, as we go forward, we can retain some consistency of vision.

It can be said that there is very little in the WM design that has not appeared in some other computer previously -- just as it can be said that Frank Lloyd Wright used the same old doors and windows as his predecessors. Architecture is not about the components of a design, it is about their composition. Specifically, it is about how a particular composition affects a very complex objective function.

In the case of computer architecture the objective function has never been formulated, and, indeed, many of the arguments about the desirability of brand X vs. brand Y are really arguments about the relative importance of aspects of the objective function. In the case of WM, at least, we tried to maximize concerns as diverse as

- performance: cycle time (critical paths), multiple operations per cycle, realistic peak vs. sustainable performance, ...
- implementability: testability, pin counts, memory bandwidth, available chip densities, ...
- specificity: precise architectural definition without over-constraining implementations
- compilability: availability and efficacy of algorithms, ...
- system issues: real-time applications, OS overheads in multi-user systems, use in embedded applications, use in multi-computer systems, ...
- applicability: don't favor one class of computation over another
- algorithms: eschew need for algorithm (re)invention

The WM architecture should be evaluated in terms of how well the composition of its components affects these objectives.

Acknowledgements

A number of people have made contributions to the design of WM. Professor Charles Hitchcock of Dartmouth has been involved from the beginning, and is responsible for a number of the machine's unique features. Professor Anita Jones, has been a patient sounding board, and has developed some of the WM software. Jim Aylor, Alan Batson, Jack Davidson, Barry Johnson, Anita Jones, Peter Kester, Sunil Pamidi, Max Salinas and Tim Sigmon are the core of the WM team at Virginia. Ivan Sutherland and Bob Sproull asked the right, sometimes embarrassing, questions at just the right time.

WAW

Chapter 1. General Features

WM is a computer architecture designed to support rapid execution and compact representation of programs written in high-level languages. It is a combination of innovative computer architecture and fast, modern machine features. Its elements support each other quite synergistically. The net result is an architecture with (a) roughly the hardware complexity associated with RISC machines, (b) roughly the code densities associated with CISC machines, and (c) roughly the performance associated with "mini-supercomputers" -- and all this is achieved with almost no increase in compiler complexity.

The most important additions to computer performance are found in three features:

Concurrent ALUs — A single machine instructions may specify two integer/logical or two floating point operations; each such instruction may operate on 2, 3, or 4 operands. The integer and floating point units are separated so that an integer and a floating point instruction can be dispatched simultaneously. Moreover, the integer and floating point units are each implemented by pipelined ALUs. In principle, therefore, WM can dispatch 4 arithmetic operations per cycle.

Streaming — A mechanism is provided for asynchronous accesses of "vector-like" data, that is, data with a known displacement between successive items. A single instruction can be executed to cause a "stream" of such data items to be delivered to WM's execution units, which can then be processed at the speed of the consuming algorithm. This scheme is similar to, but more general than that found in "vector" supercomputers, and gives the effect of executing

as many as 8 load/store instructions per cycle *in addition to* the 4 arithmetic operations per cycle.

Fast Conditionals — A new approach to conditional jumps is taken that allows most such operations to take NO execution time -- that is, their execution time can be completely overlapped with that of other instructions.

The net effect of these three features is a peak performance 13 RISC-like operations per cycle. Other important architecture features that further enhance the performance and applicability of WM include:

Micro-concurrence — The ability to dispatch multiple operations per cycle is significantly enhanced by a careful decoupling of independent execution units, and synchronizing their actions through FIFOs.

32-bit Instructions — All instructions are 32-bits. This aids in numerous ways, such as by expanding the relative jump distance, providing quicker decoding, eliminating awkward page-fault situations, and simplifying the prefetch requirements.

Load Prefetch — WM is a load/store machine, but loads can be started well before a memory data item is needed. This reduces the effect of a long memory access latency.

Large Register Set — 64 register names are provided -- of which 27 are integer/logical "general" registers, 29 are floating-point registers, and 8 are functionally specific (see section 2.3 concerning the WM memory/processor interface).

Parameter Passing — A parameter passing mechanism is incorporated that is faster and more convenient than previous methods.

Ada Influence — WM was designed with the needs of high-level languages, as well as the capabilities of modern optimizing compilers, in mind. Of particular interest was rapid execution of Ada programs -- both because of the growing importance of Ada and because it provided a good "stress test" on the success of this aspect of the design. In particular, constructs that are awkward on other modern computers, such as proper loop control, exceptions, constraint checking, and tasking, are smoothly supported on WM.

Operating System Support — WM provides a number of features to support operating systems; the general philosophy being to move as much functionality as possible *safely* into ordinary,

unprivileged "user level" code. The result is that it is possible to

- directly handle IO devices from user programs
- have user programs act as a scheduler for a set of other user tasks
- build a compartmented, multi-level military security system easily

Each of these provides the possibility of significant reductions in operating system overheads.

Chapter 2. Logical Machine Structure

This chapter describes the basic framework of the WM machine, such as its data types, logical memory interface, and implied stack structure. Those familiar with modern computer architectures may wish only to skim the material on data and addressing. However, there are several unusual aspects of the WM design covered in the sections "Execution Units" and "Memory Reads & Writes"; these should be read carefully.

2.1. Addresses & Data

2.1.1. Addresses

Addresses on the WM architecture are 32-bit values and memory is 8-bit byte addressed. WM systems are "paged", and addresses are translated and protected as described in Chapter 4.

2.1.2. Data Sizes

Data elements in memory may be stored in 8-bit byte, 16-bit halfword, 32-bit word, or 64-bit doubleword sizes. Data elements are assumed to be aligned. For example, addresses of halfword data elements are assumed to have a zero least significant bit, thus specifying a halfword boundary. In fact, this least significant address bit is ignored when accessing such data elements. All instructions are 32-bits in length, and the Program Counter (PC) always specifies a word-aligned address. Default instruction sequencing is linear and increasing (i.e., the execution of the instruction at address XXX+4 follows the execution of the instruction at address XXX).

WM is a "load/store architecture", that is, data manipulation can only be performed on (between) registers. Separate instructions are required to move data between the registers and memory. There are 32 general register names that may be specified in an instruction -- however integer and floating point registers are distinct, providing 64 total register names. As an aid in computation, register 31 in both the integer and floating point unit is defined to be identically zero. Although it is possible to write to this register, references to register 31 evaluate to zero. In addition, four register names in the integer unit and two in the floating point unit are reserved for special meanings, as will be discussed in section 2.3.

2.1.3. Data Types

These data types are supported by the WM architecture:

Boolean Values
: No explicit instructions exist to support operations on boolean values. However, the available operations were created with such support in mind. In particular, any bit in a register may be set, or selected in one instruction, and any bit may be cleared or tested in 2 instructions. These macro functions are synthesized by the proper operation combination. Vectors of boolean values may be loaded from and stored to memory as bytes, halfwords, words, or doublewords. 32-bit boolean vectors may be logically manipulated with register/register instructions. Shorter boolean fields may also be extracted from larger vectors with a single instruction.

Signed Integer Values
: Arithmetic on 2's-complement 32-bit signed integers is supported by individual operations on this machine. While signed integers may be loaded and stored as words, halfwords, or bytes, all integer arithmetic is performed on 32-bit register quantities. Unsigned integers are not supported by the machine; they are viewed as a subset of the signed integers (as Ada defines NATURAL). Explicit underflow checking is required when synthesizing unsigned arithmetic with this architecture.

Floating Point Values
: Both single- and double-precision floating point formats of the IEEE floating point standard are supported. The terms "float" and "double" in this document refer to single- and double-precision floating point data and/or operations, respectively.

2.1.4. Data Numbering

The current architecture description is "big-endian", although a simple rewrite could make it "little-endian" without substantive change. It seems more important that the description be consistent, which we have tried to do.

Bits within bytes, halfwords, words, and doublewords are numbered from left to right starting with 0. The lefthand side is the most significant. Bytes within larger entities, such as words, are also numbered from left to right starting with 0. The last byte (number 3) in word 327 is just before the first byte (number 0) in word 328.

2.2. Integer and Floating Point Execution Units

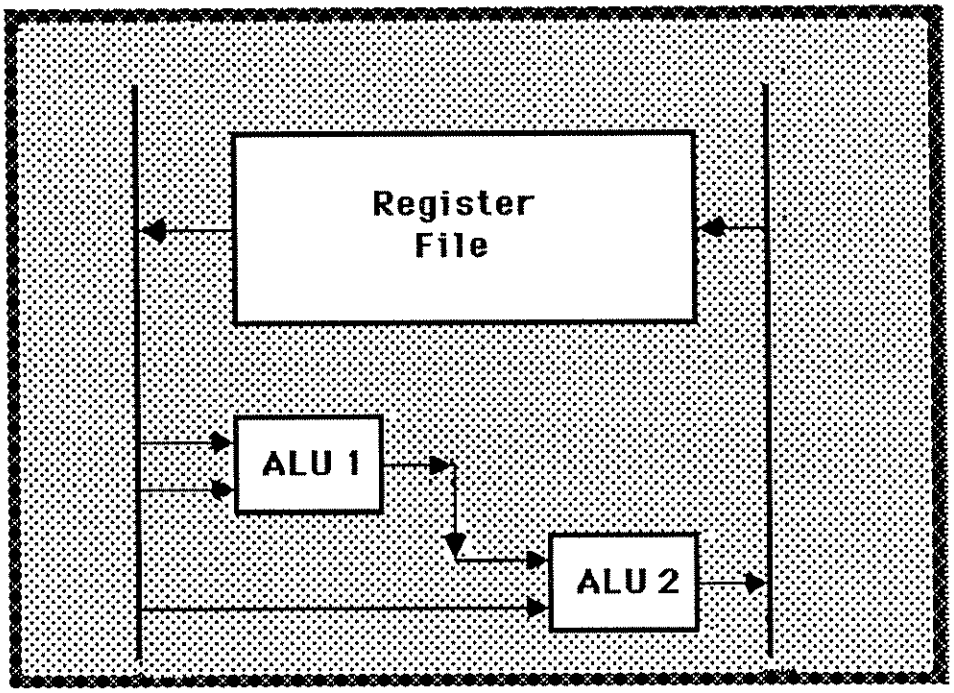
The data manipulation instructions of WM specify 3 source operands, 2 operators, and a destination register, and evaluate an assignment of the form:

R0 := (R1 op1 R2) op2 R3

The source operand of integer/logical instructions may be the contents of a register, the contents of an input FIFO (see the next section), or unsigned literals; the destination may be either a register or an output FIFO. The source operands of a floating point instruction may be either a register or an input FIFO, and the destination may be either a register or an output FIFO (floating literals are not supported as source operands).

The execution units of WM are implemented as a pair of pipelined ALUs, as shown in the following figure. In general, while the second (outer, op2) operation of one instruction is being executed in ALU2, the first (inner, op1) operation of the successor instruction is being executed in ALU1.

The integer/logical registers are distinct from the floating point registers. Integer/logical instructions refer to the integer registers; floating point instructions refer to the floating point registers. Conversion instructions (e.g., "convert integer to floating") refer to one register of each type as appropriate. Integer instructions are used for address computation, and to specify data transfers between memory and registers of both types.



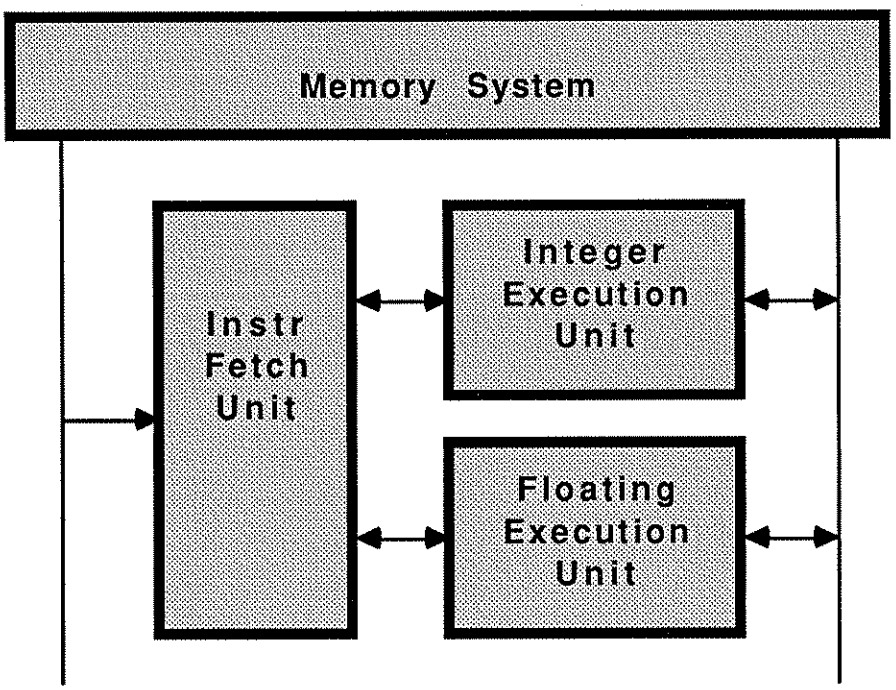
The pipelined structure of the WM execution units induces the data dependency rule:

> **The result of an instruction is not available as an operand of the inner operation of the following instruction. The value of an inner operand is specifically independent of the effect of the previous instruction.**

Valid programs must obey this rule. Clever programs will exploit it.

Note that since the integer execution unit and the floating point execution unit are decoupled, data dependencies can arise which are not immediately obvious. Consider an integer instruction followed by a series of floating point instructions which are in turn followed by another integer instruction. Depending on the sizes of the instruction FIFOs and the speed of the instruction fetch unit, the two integer instructions may have a data dependency.

Since integer/logical and floating point operations as well as data are distinct, the proper conceptual model of WM is one with two execution units (of the general form shown above) under common control of the "instruction fetch unit". Note that the independence of the integer and floating point units permits concurrent execution under most circumstances. A more precise definition of when concurrence is not allowed will be discussed later.



2.3. Memory Reads & Writes

The method of accessing memory in WM is somewhat unusual:

1. WM interposes FIFOs ("first in, first out queues") between the register set and the memory. Load and store instructions are operations on these FIFOs.

2. In most machines, memory reads and writes require specifying two quantities: a register name and a memory address -- e.g., "load the contents of location 42 into register 13". In WM, by contrast, loads and stores specify *only an address*, because
   - a load is a request to enqueue data from memory into an input FIFO for register 0, and
   - a store is a request to dequeue data from an output FIFO for register 0 and store it to memory.

   In both cases, only the memory address is required.

3. To dequeue a datum from an input FIFO, an instruction need only reference register 0 as a source operand. To enqueue a datum in an output FIFO, an instruction need only specify register 0 as the destination of a computation.

To read a value from memory, a LOAD instruction is first used to compute a memory address. This is passed to the memory system and the data transfer is initiated. Multiple load instructions may be executed; the data is read in the order of the LOAD instructions and queued in an input FIFO. The execution units access the queued data by referencing register 0. When register 0 is read as part of a computation, the next value is taken out of the FIFO and used.

To write a value to memory, that value may first be written (or computed) into register 0; as a consequence it is enqueued in an output FIFO. Then, at some later time, a STORE instruction may be issued which computes a memory address. Once both actions have been completed, a copy of the value that was written into register 0 is written to memory at the computed address. Because "register 0" is a FIFO, several values can be computed into it before a STORE instruction is executed.

The write to a memory location could have been accomplished in the other order -- the STORE instruction could have been executed before the instruction that computes the value to be stored; the action of writing to memory is taken only when an appropriate pair of instructions have **both** been executed. And again, several STORE instructions could have been executed before the first value to be stored is computed into register 0; the addresses are queued until the value to be stored is computed.

Note that register name 0 specifies different things when read and when written. Reading register 0 takes a value from the memory input FIFO. Writing register 0 prepares a value to be written to memory.

Finally, note that the LOAD and STORE instructions themselves are executed on the integer/logical unit and the addresses they compute are directed to the proper FIFO (i.e., the address computed by the "load floating" instruction is directed to the floating point FIFO).

The size of the input and output FIFOs are implementation defined, although the architecture requires at least:

- 3 32-bit entries in the integer unit's input FIFOs
- 1 32-bit entry in the integer unit's output FIFO
- 3 64-bit entries in the floating unit's input FIFOs, and
- 1 64-bit entry in the floating unit's output FIFO.

These minimums are adequate to ensure that a single instruction can execute, even if it names all its source operands and it's destination operand as FIFOs. Generally, implementations should provide more entries in each of the FIFOs in order to allow software to perform better.

For any particular implementation, hence specific FIFO sizes, it is possible to construct a program that will deadlock -- for example, by trying to enqueue more than a particular FIFO can hold. Such programs are *invalid*. See appendix A for a more complete discussion of invalid programs.

2.3.1 Parameter Bypass

Register 1 is also a FIFO, albeit with somewhat different properties than that of register 0. Specifically, a value stored (computed) into register 1 is immediately enqueued in the register 1 input FIFO. As with register 0, items are dequeued from the register 1 input FIFO simply by using register 1 as a source operand.

Thus, register 1 is a short buffer that might be used, for example, to hold a short queue of temporary values. The primary intent of this mechanism, however, is to hold parameters during a subroutine call -- the caller enqueues actual parameters, and the called routine dequeues formals. See Appendix F for a discussion of the suggested WM calling sequence.

2.3.2 Streaming

The memory system also supports the reading and writing of "streams" of data. A data stream is an linear sequence of memory items, all of the same size and type, that start at a known address and are spaced a constant distance (stride) from each other.

Either register 0 or register 1 may be used in stream mode; that is, each of these registers supports two modes of operation, normal and streaming mode respectively. Normal mode for register 0 is the LOAD/STORE mode described at the beginning of section 2.3. Normal mode for register 1 is the parameter bypass mode described in section 2.3.1. Stream mode is identical for both registers, and is described here.

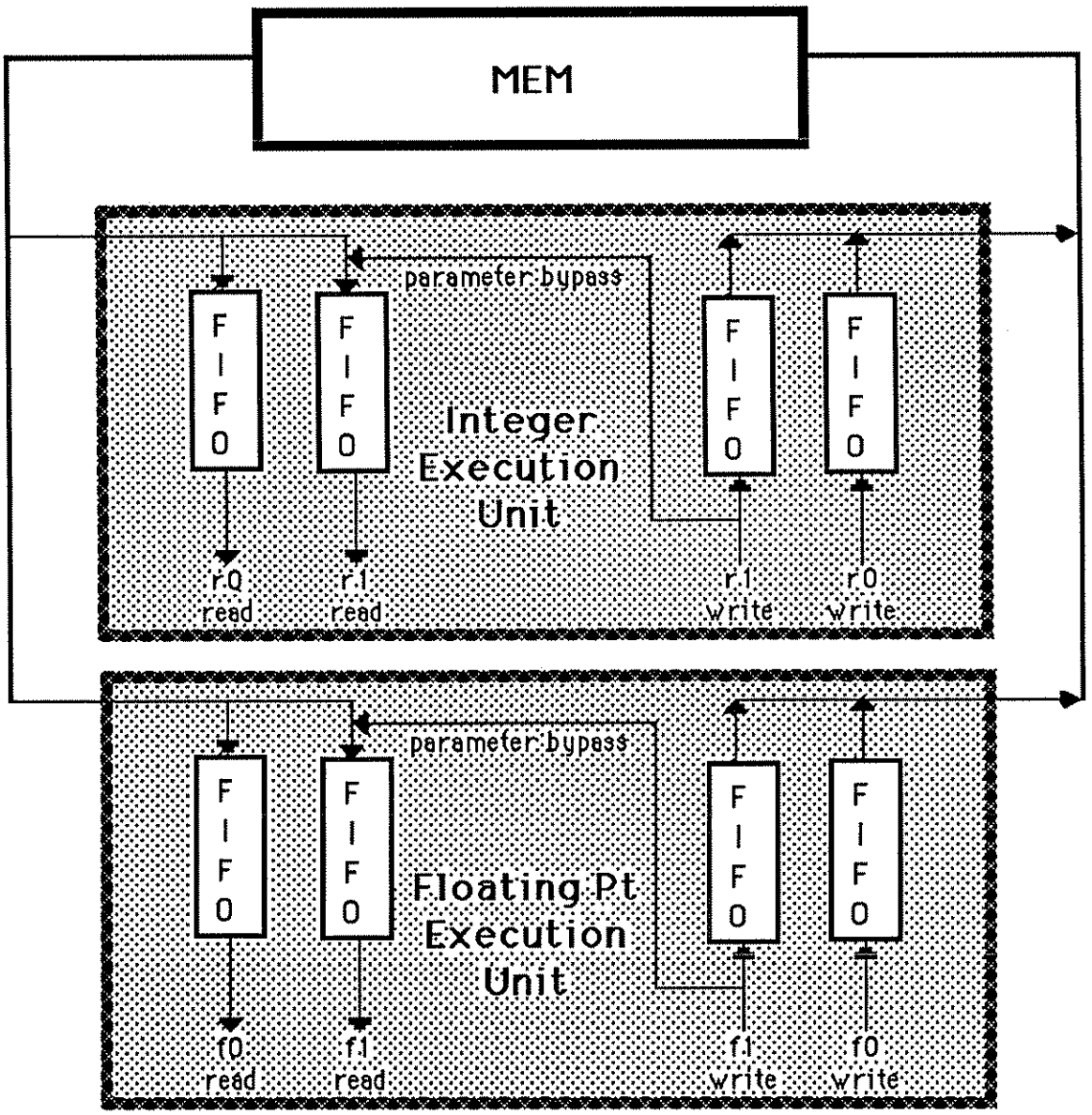
When in streaming mode, no load or store instructions need to be issued in order to initiate the next data transfer; a single "start streaming" instruction initiates the transfer of the entire stream. Asynchronous "stream control units" compute the addresses of the "next" data item. and initiate the transfer.

When streaming, data is removed from the input FIFOs in the same manner as in non-streaming mode -- that is, by instructions that reference register 0 or register 1. Similarly, by designating register 0 or register 1 as the destination of an instruction, data is inserted into the output FIFO (same as the normal mode for register 0 but different from register 1's normal mode.) If streaming is performed only with register 0, programs that exploit streaming are functionally identical to those that do not, except that no LOAD/STORE instructions appear in the streaming programs. If register 1 is involved in a stream, however, the parameter bypass capability is not available.

Streaming provides functionality and performance similar to that of high speed "vector" architectures, but

- it is more general in the sense that it can do everything that a vector unit can, plus a great deal more. Computations done by vector instructions are typically "small" -- e.g., simple component-by-component adds or multiplies. Computations on streamed data are, by contrast, arbitrary software combinations of WM's "scalar" instructions. In particular, this allows streaming of computations involving recurrences (which cannot be

vectorized) and reductions (which are supported on only some vector machines).
- it requires no additional hardware (e.g., no vector registers, no separate ALUs, etc.).
- much simpler and more complete compiler algorithms can be used to detect streaming opportunities than to detect vectorization opportunities.



.2.4. Stack Structure

The WM architecture has Stack Limit and Stack Index registers defined to be registers 2 and 3 of the integer execution unit, respectively. The Stack Limit register is guaranteed by software to lie on a page boundary, thus having its lower bits be zero accordingly. (The page size is implementation-dependent, so the number of zeroed lower bits is not specified.) The Stack Index contains an integer such that the address of the top of stack is computed as follows:

TOS := SL + SI

The Stack Index normally has a negative value. The stack grows towards the positive addresses, and a transition from negative to positive Stack Index is the overflow condition. This condition is checked by hardware whenever the Stack Index is written; an exception is generated if it is met. The Stack Limit may only be written by programs with proper privileges. No push or pops are supported, nor needed, on this machine. The stack is not for expression evaluation, but merely for procedure interface. As such, being able to adjust the effective top of stack is sufficient, given that most parameters are passed via the input FIFO associated with register 1.)

The hardware support for the stack, together with the general instruction set facilities of WM, encourage an overall stack layout such as shown in the figure below below. The various areas of the stack are:

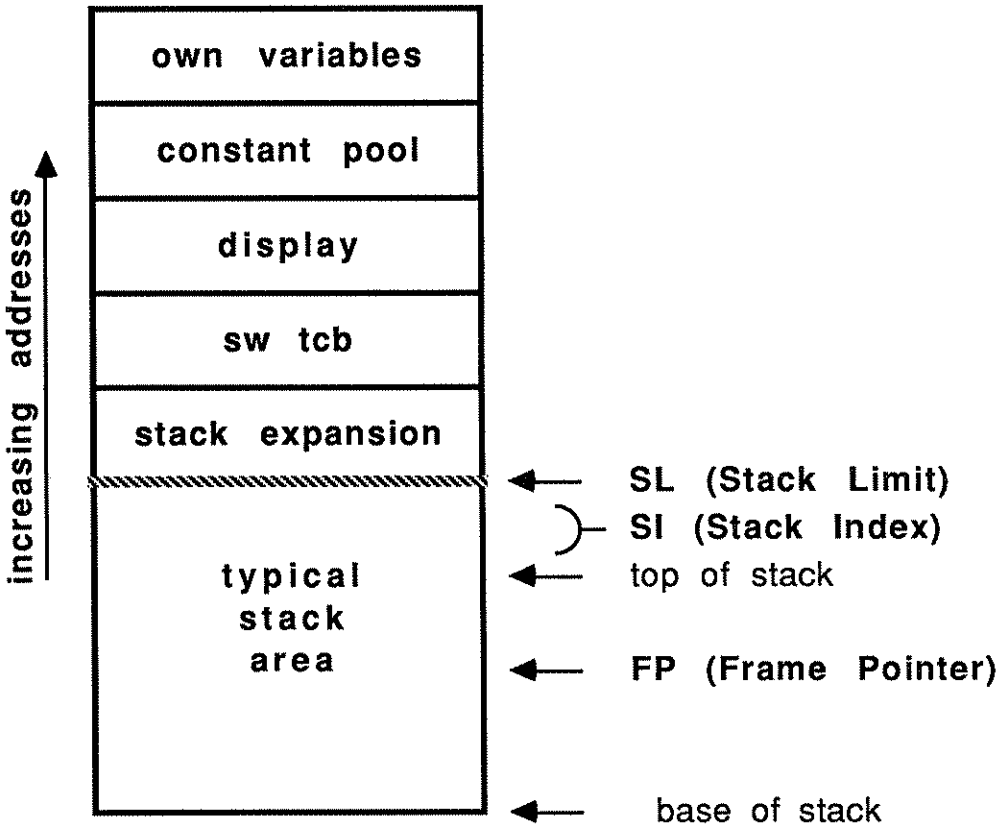
- stack expansion area: this area is normally unused -- but provides temporary stack expansion while, for example, a exception is taken and the program stack was near its limit (this implies that exception handlers must be careful about the amount of stack they use, but allows graceful handling of otherwise awkward situations).

- software task control block: this is a software-defined area for recording task status. There is also a hardware-defined task save area that has special protection associated with it (see chapter 4 for further detail.).

- display: this area is, by software convention, used to hold the display (note, the display is somewhat different than the usual; see Appendix F).

- constant pool: this area, by software convention, is used for storing constants larger than those that can be synthesized in normal instructions.

- own variables: this area is, again, by software convention, used for statically allocated variables and the constant pool for the task.



Note that, in general, Ada requires a "cactus stack"; only one branch of this stack is shown here. In general, the display elements may point into lower branches of the stack.

Besides registers 0, 1, 2, 3 and 31, all other register names specify truly general purpose registers. Note, however, that the hardware uses some of the general registers (e.g., the PC is stored in register 4 by a CALL instruction), and that software may impose additional conventions -- thus not all 27 general registers are available for computation. Appendix F discusses this in more detail.

Other aspects of the machine state, such as the Program Counter (PC), the Cycle Counter (CC), the Program Control Word (PCW), and the Program Status Word (PSW) cannot be directly accessed by instruction (other than certain bits of the PSW which may be set as a side effect of another instruction -- e.g., the condition code); these registers are only (re)set as a consequence of a context-swap.

2.5 IO

Control of input/output devices is "memory mapped"; that is, there is a portion of the physical address space reserved for "device registers". This, together with the address translation and protection mechanism discussed in Chapter 4, provides great flexibility -- it even allows one to safely permit unprivileged applications programs to directly access IO devices, thus eliminating operating system overhead from them.

At least three devices are required of all implementations:

- "the CPU", control and status registers for the processor itself. Chapter 4 contains more details, but, for example this allows one processor to start/stop another, allows diagnostic probing of the processor, and replaces a number of instructions (such as HALT) with bit set/reset operations on this device register.

- one or more "timers", which are 32-bit counters that decrement each 100ns and, if enabled, interrupt when they become negative (but continue counting until reset), and,

- a "calendar" which is a 64-bit counter that is incremented each 100ns, runs continuously when power is enabled, and will interrupt when it overflows.

Chapter 3. Instruction Descriptions

The WM instructions are grouped by function and decoding into the following five categories:

- Integer Arithmetic & Logical
- Load & Store
- Control Flow
- Floating Point
- Special

The following sections describe each set of instructions in some detail.

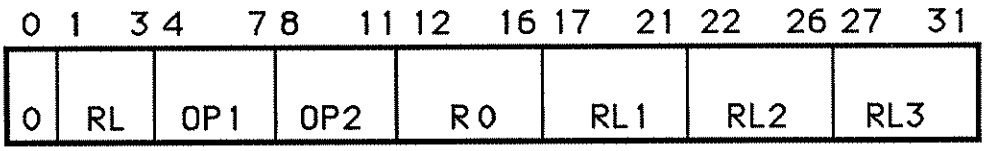
3.1. Integer Arithmetic & Logical Instructions

The integer arithmetic and logical instructions all take the form[1]:

R0 := (RL1 op1 RL2) op2 RL3

There are three source specifiers (RL1, RL2, and RL3), two operation specifiers (op1 and op2), and a destination specifier, R0. R0, RL1, RL2, and RL3 may specify one of 32 register names. Alternatively, RL1, RL2, and RL3 may specify a 5-bit literal (which is zero-extended to 32-bits); the RL field determines this interpretation. The operands are either 32-bit signed integers or a vector of 32 boolean values, depending on the operation specified. The instruction format is:

[1] Here and in the following, we observe the convention that capital "R" denotes a register field of an instruction, and a lower case "r" denotes a specific register. "RL" denotes an instruction field that may be either a register or a literal.



Bits 1, 2, and 3 specify if RL1, RL2, and RL3, respectively, are 5-bit literals (bit equal 1) or register names (bit equal 0). The operation fields, op1 and op2, are encoded identically. The 16 functions they can specify are:

+	addition
-	subtraction
- '	reverse subtraction: (a-b)==(b-'a)
and	bitwise AND
or	bitwise OR
eqv	bitwise EQUIVALENCE
*	multiplication
/	division
/'	reverse division: (a/b)==(b/'a)
asl	arithmetically shift (to the left) the left operand by the amount of the right operand (performs scaling)
=	equal
<>	not equal
<	less than
<=	less than or equal
>=	greater than or equal
>	greater than

Note that there are no unary (monadic) operators in this collection; this is because the monadic operations can be synthesized from the dyadic ones and literals. Specifically,

-x == (0-x), and
not (x) == (x eqv 0)

The last six operations are the relationals. They produce their left operand as a result. They also produce a boolean value. If two relationals exist in the same instruction, their boolean values are either AND'd or OR'd together, determined by a bit in the Program Control Word, and written to the conditional bit. Otherwise, the single boolean value sets a condition bit in the PSW[1]. In either case, if the boolean result is False, then the condition bit is set, but the instruction's register write and exception conditions are nullified. As will be discussed later, valid programs must adhere to the rule that exactly one instruction with relational operations is executed (dynamically) for each conditional jump instruction executed.

Such relationals support general conditional branching for loops and if-then-else constructs found in high-level languages. Their semantics also allow for efficient Ada

[1]Note, there is a small FIFO of condition bits in the PSW.

looping and range checking. An Ada loop may use Integer'Last as its upper bound. In such cases, a comparison of the bound and the loop counter must be made before the counter is incremented. Otherwise, an overflow would result, which it shouldn't. The instructions:

```
r10 := Integer'Last
r8 := (r8 < r10) + 1
```

provides this function efficiently. Assuming the PCW indicates that relational should be OR'd, Ada range checking can be accomplished in a single instruction. The instruction:

```
r8 := (r8 > 12) < 1
JumpIT  OutOfRange
```

checks to see if the value in register 8 is in the range [1..12], since the result of the two relationals is OR'd together.

3.1.1. Exceptions

The following arithmetic conditions result in exceptions unless masked off in the PSW:

- Input FIFO 0/1 Empty: an attempt to read r0 or r1 was made when no value is present in the FIFO, nor is any value scheduled to be loaded.

- Output FIFO 0/1 Full: an attempt to write r0 or r1 was made when the associated output FIFO was already full, and no value is scheduled to be stored.

- Overflow/Underflow: an arithmetic operation overflowed or underflowed

- Divide by 0: an attempt to divide by zero was made

3.2. Load & Store Instructions

The LOAD and STORE instructions specify two things: (1) the address of the data to be read or written, and (2) the size/type of the data (e.g., byte vs. halfword vs. double-precision floating point).

(1) The address computation is formally and semantically identical to the assignments of the integer/logical instructions:

```
R0 := (R1 op1 RL2) op2 RL3
```

The only differences are that the set of operators is smaller, R1 cannot be a literal, and, more importantly, the result of the computation is sent to the memory system *in addition to being sent to the destination register.* The permitted operations are:

+	addition
-	subtraction
*	multiplication
asl	arithmetical shift left

The definition of each of these is the same as it's integer/logical counterpart.[1]

(2) The type/size of the data to be read or written is specified by the LOAD or STORE instruction.

On LOADs:

- word, halfword, and byte requests (possibly sign-extended) are sent to the input FIFO associated with register 0 in the integer execution unit.

- floating and double requests are sent to the input FIFO associated with register 0 in the floating point execution unit.

On STOREs

- word, halfword, and byte requests (possibly checked for range violations) are sent to the output FIFO associated with register 0 in the integer execution unit.

- floating and double requests are sent to the output FIFO associated with register 0 in the floating point execution unit.

Once a load instruction has been issued, a following instruction may attempt to read the associated input FIFO (by referencing register 0). This instruction will not be issued until a value from memory is available. In fact, a single instruction may reference register 0 three times as a source operand, and thus read up to three values from the input FIFO, so long as a LOAD has been performed for each such reference.

The memory system is responsible for ensuring that certain sequences of load/store operations are performed properly, as specified below. However, these sequences are guaranteed to function properly *only* with respect to the IEU and FEU separately; loads and stores from one execution unit are not synchronized with those of the other!

```
STORE (to address X)
LOAD (from address X)
```

loads the same value as was stored. Conversely, the sequence:

```
LOAD (from address X)
STORE (to address X)
```

reads the "previous value" of location X. Finally, the sequence:

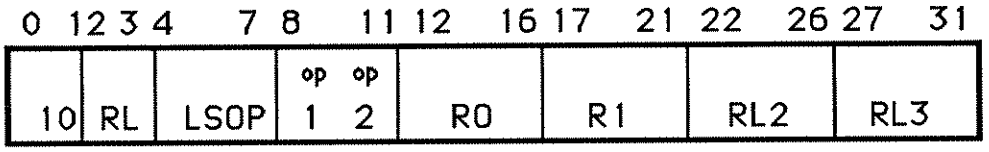
```
r0 := Y
STORE (to address X)
r0 := Z
STORE (to address X)
```

must result with the value Z at location X in memory.

---

[1] Note: prepending these operator codes with "00" produces the integer/logical encodings of the same operators.

The format of load and store instructions is:



Bits 2 and 3 specify whether RL2 and RL3, respectively, are 5-bit literals (bit equal 1) or register names (bit equal 0). R1 must name a register. The LSOP field encodes the load/store function, which may be one of:

LD	Load Double
LF	Load Floating
LW	Load Word
LH	Load Halfword zero extended
LB	Load Byte zero extended
LHX	Load Halfword sign-eXtended
LBX	Load Byte sign-eXtended
SD	Store Double
SF	Store Floating
SW	Store Word
SH	Store Halfword
SB	Store Byte
SHA	Store Halfword with Arithmetic range check
SBA	Store Byte with Arithmetic range check

The last two store instructions, SBA and SHA, verify that the value to be stored is arithmetically representable in the storage unit into which it will be stored; e.g., SBA checks that the value is in the range -128..127. The SB and SH instructions, by contrast, simply store the least significant 8 and 16 bits of the value, respectively.

There are 14 LSOP operations; the other two opcodes are illegal and will produce an illegal instruction trap if used.

Other instructions that are related to the loads and stores, but are discussed in the Special Instruction section, include:

- Context Save and Restore
- Streaming Instructions

3.2.1. Exceptions

The following exceptions may occur as a result of a load or store instruction:

- Input FIFO 0/1 Empty: as per integer instructions when used as a source operand in the address calculation.

- Input FIFO 0 Full: an attempt was made to perform a load when the input FIFO was already full, or will be full after some pending loads complete.

- Output FIFO 0 Empty: an attempt has been made to perform a store when the output FIFO is empty and no further address can be buffered.

- Output FIFO 0/1 Full: as per integer instructions when specified as the destination register for the address calculation.

- Overflow/Underflow: an arithmetic operation overflowed or underflowed

- Illegal Access: an attempt to read, write, or execute from/to a memory location without proper access privilege (see Chapter 4 for a more complete discussion).

- Load While Input Streaming: a load instruction while register 0 is in input streaming mode.

- Store While Output Streaming: a store instruction while register 0 is in output streaming mode.

3.3. Control Flow Instructions

The set of control flow instructions include Jumps and Calls. These instructions replace the Program Counter with a new value, the target address. In all but one case, this is a PC-relative address, and is formed by concatenating two zeros to the bottom of the sign-extended offset and adding this value to the current Program Counter[1].

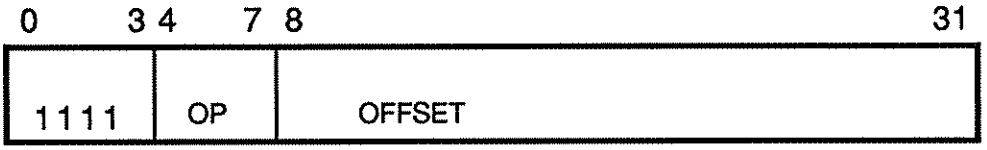
There are four conditional jumps associated with the two condition FIFOs: "Jump True" and "Jump False" for each of the integer and floating conditions. Conditional Jumps "consume" a condition bit generated by a relational operation. Valid programs must guarantee that exactly one instruction containing a relational operation is executed for each conditional instruction.

There are eight conditional jumps associated with the streaming facility of the machine (see below); these support jumps on the on "stream count not zero" for each of the input and output streams.

There are two call instructions: Call and ECall. Call simply stores the current PC in register 4 and jumps to the specified destination. ECall performs the function of a "supervisor call", and will be discussed in detail in Chapter 4. The format of control instructions is:

---

[1]The exception is ECall, whose target address is determined indirectly from the PC-relative address described above.



The opcode may specify one of 15 operations:

Jump	the basic unconditional jump.
JumpIT	jumps only if the Integer condition bit is True.
JumpIF	jumps only if the Integer condition bit is False.
JumpFT	jumps only if the Floating condition bit is True.
JumpFF	jumps only if the Floating condition bit is False.
JNI r0	"Jump on Stream Count Not Zero; Integer Input FIFO 0".
JNI r1	"Jump on Stream Count Not Zero; Integer Input FIFO 1".
JNO r0	"Jump on Stream Count Not Zero; Integer Output FIFO 0".
JNO r1	"Jump on Stream Count Not Zero; Integer Output FIFO 1".
JNI f0	"Jump on Stream Count Not Zero; Floating Input FIFO 0".
JNI f1	"Jump on Stream Count Not Zero; Floating Input FIFO 1".
JNO f0	"Jump on Stream Count Not Zero; Floating Output FIFO 0".
JNO f1	"Jump on Stream Count Not Zero; Floating Output FIFO 1".
Call	Store PC of the next instruction in r4; jump to specified address.
ECall	Entry Call; see below

Note that there are 15 operations in this class; 1111 is not a valid control flow opcode. In particular, an instruction that is all ones will trap as an undefined instruction!

ECall provides the functionality of "supervisor call" in other architectures; it has three effects:

(1) it changes the protection table pointer to that contained in the entry page (note, the map table pointer is not changed),
(2) it jumps indirectly through the specified PC-relative location, and
(3) it saves the prior protection table pointer and program counter in a special protected stack area.

The rationale for each of these actions will become clearer in Chapter 4. Note, however, that the address specified in by the PC-relative target address must be that of an "Entry Page", and that the task executing the ECall must have "call rights" to this page.

Three instructions that affect control flow are encoded among the "special" instructions since they do not need to specify a PC-relative address (see section 3.5 for the format of special instructions). Nonetheless, we will discuss them here; they are Jumpl (Jump Indirect), Calll (Call Indirect) and EReturn (Return from ECall).

Jumpl	Jump Indirect. An unconditional jump that gets its target address from the register named in the R1

field. This instruction can be used for "case table" jumps, to branch more than 2**24 instructions, and to return from procedure calls.

CallI — Call Indirect stores the current PC in register 4 and gets its target address from the register named in the R1 field.

EReturn — Entry Return. This is the complementary instruction to ECall; it restores the protection table pointer and PC from the protected stack.

3.3.1. Exceptions

The following exceptions may be raised as the result of a control flow instruction:

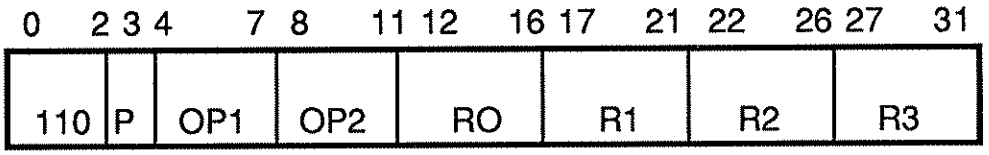
- Condition Unset: A JumpIT(JumpFT) or JumpIF(JumpFF) instruction is being executed, and the condition bit has not been set (and is not in the process of being set) by a previous relational operator.
- Protection Violation: An attempt was made to transfer control to a page without proper access privileges (see Chapter 4 for a more complete discussion).
- Page Fault: In a virtual memory system, an attempt to execute from a virtual address that does not exist in physical memory.

3.4. Floating Point Instructions

The floating point instructions are quite similar to the integer/logical instructions. Their general computation is:

R0 := (R1 op1 R2) op2 R3

Notice, however, that literals cannot be specified; only register operands are permitted -- these are always the "floating registers". The instruction format is:



The "p" bit specifies if the instruction's operands are single-precision (= 0) or double-precision (= 1). These values are expressed in IEEE floating point standard format. Pairs of registers are used when double-precision is specified. In such cases, even/odd pairs are specified. For example, both f4 and f5 specify the value (f4 catenate

f5). When either register f0 or f1 is specified in a register read, the corresponding FIFO has two word values read from it and catenated to form a 64-bit value. The first word read is on the most significant end (i.e., the "left").

The op1 and op2 fields are encoded identically. They may specify any of the following 14 operations:

+	addition
-	subtraction
- '	reverse subtraction
*	multiplication
/	division
/ '	reverse division
nop	no operation; returns its left operand
nop'	no operation; returns its right operand
=	equal
<>	not equal
<=	less than or equal
<	less than
>	greater than
>=	greater than or equal

Note that monadic operations can be synthesized by using f31 which is defined to be identically zero.

```
-x == (f31 - x)
```

There are, however, two monadic operations included in the floating point set -- this is due to the presence of encoding space and the realization that in most implementations, the nop operation will execute in less time than its synthesized counterpart.

The last six operations are the relationals. They produce their left operand as a result. They also produce a boolean value. If two relationals exist in the same instruction, their boolean values are either AND'd or OR'd together and written to the conditional bit under control of a PCW bit. Otherwise, the single boolean value sets the condition bit. In either case, if the boolean result is False, then the instruction's register write and exception conditions are nullified. Software must guarantee that exactly one instruction with relational operations is specified before a conditional jump.

3.4.1. Exceptions

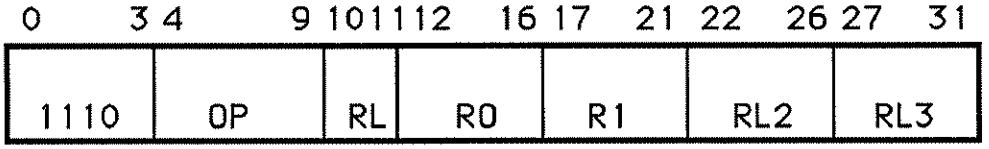
The following arithmetic conditions result in exceptions unless masked off in the PCW:

- Overflow/Underflow: as per integer instructions.
- Divide by 0: as per integer instructions.
- Condition FIFO overflow: As per the integer instructions.

Note that input/output FIFO empty/full are not exception conditions for the floating point instructions as they were for the integer and load/store instructions. This allows the integer and floating point units to proceed asynchronously preparing/consuming addresses and data -- but does require a more global detection of erroneous (deadlocked) programs.

3.5. Special Instructions

Only a few types of instructions are encoded in this category. Each type will be discussed separately, however the common format for these instructions is:



3.5.1. Streaming

The WM computer architecture supports a feature called "streaming". Streaming is a method of loading and storing structured data elements without having to do explicit address computations. It assumes a vector of data elements are present, or are to be created, in memory, and that they are a constant stride (number of bytes) apart from each other. Stream instructions are used to read/write such vectors from/to FIFOs.

There are 14 instructions that initiate streaming operations. These are analogous to the 14 types of loads and stores. They may specify data as doublewords, floating, words, halfwords (zero- or sign-extended/arithmetically checked), or bytes (zero- or sign-extended/arithmetically checked). The operands specify a base address (R1), a count[1] (RL2), a stride[2] (RL3), and which FIFO to use (0 or 1); this last parameter is taken as the least significant bit of the R0 field.

Finally, there are five instructions to stop streaming operations. These instructions stop input or output streaming and flush the relevant FIFOs.

To summarize the operations, they are:

SinD	Stream In Doublewords (to the floating execution unit)
SinF	Stream in Floating (to the floating execution unit)
SinW	Stream In Words (to the integer execution unit)
SinH	Stream In Halfwords (to the integer execution unit)
SinB	Stream In Bytes (to the integer execution unit)
SinHX	Stream In Halfwords sign-eXtended (to the integer execution unit)

[1] A count of -1 is defined to be an infinitely long stream. That is, the stream will continue until a stop streaming instruction is performed

[2] In bytes.

SinBX	Stream In Bytes sign-eXtended (to the integer execution unit)
SoutD	Stream Out Doublewords (from the floating execution unit)
SoutF	Stream Out Floating (from the floating execution unit)
SoutW	Stream Out Words (from the integer execution unit)
SoutH	Stream Out Halfwords (from the integer execution unit)
SoutB	Stream Out Bytes (from the integer execution unit)
SoutHA	Stream Out Halfwords with Arithmetic range checking
SoutBA	Stream Out Bytes with Arithmetic range checking
StopAll	Stop all Streaming operations
StopII	Stop Integer Input Streaming operations on FIFO specified by R0
StopIO	Stop Integer Output Streaming operations on FIFO specified by R0
StopFI	Stop Floating Input Streaming operations on FIFO specified by R0
StopFO	Stop Floating Output Streaming operations on FIFO specified by R0

An example of a valid assembly instruction is:

```
SinW    r1, r9, 22, 16  -- FIFO, address, count, stride
```

This instruction would cause a vector of 22 words, whose base address is contained in r9, and which are displaced 16 bytes from each other, to be streamed to FIFO "r1" in the IEU.

A stop instruction applied to an output FIFO will complete pending memory writes (where data is available), reset the stream count, remove any extra addresses which have been calculated and restore the FIFO to non-streaming mode. A stop instruction applied to an input FIFO will take the counterpart action, discarding all data currently in the FIFO.

Only one input stream and one output stream per FIFO may coexist. This imposes a maximum of eight (four input and four output) simultaneous streams.

An input FIFO is considered to be in streaming mode until all of its data has been consumed or until the stream is halted by a stop streaming instruction. An output FIFO is considered to be in streaming mode until all data has been written to it or until the stream is halted by a stop streaming instruction. An exception is raised if a start streaming instruction is executed while the specified FIFO is in streaming mode. An exception is also raised if a LOAD/STORE instruction is executed for a FIFO in streaming mode.

Note that unlike LOAD/STORE instructions, consistency is not guaranteed between input and output streams. More specifically, when streaming both in and out of the same locations, the memory system has no responsibility of maintaining the order between memory reads and writes. For recurrences, the addition of a few registers to hold temporary values solves this problem.

Streaming instructions may cause Page Fault exceptions. If made during a memory read, the exception is not raised until an attempt to read register 0 or 1 unsuccessfully.

If made during a write of register 0 or 1 (to be written into memory), the exception is raised immediately.

All start streaming instructions require implicit synchronization with the Integer Execution Unit. Stop integer streaming instructions also require implicit synchronization with the IEU while stop floating streaming instructions require synchronization with both execution units.

3.5.2. State Manipulation

There are a few instructions to provide access to special state and to help save and restore state efficiently.

```
LoadM       R1, L2, L3
FLoadM      R1, L2, L3
StoreM      R1, L2, L3
FStoreM     R1, L2, L3
```

These instructions load and store a series of registers, from register number L2 to register number L3, with L3 guaranteed by software to be a greater register number than L2. Note separate instructions specify IEU and FEU registers. The RL field is ignored by this instruction and literals are always specified by RL2 and RL3. The storage location involved is specified by R1.

```
LoadFifoII      R0, R1
LoadFifoIO      R0, R1
StoreFifoII     R0, R1
StoreFifoIO     R0, R1
LoadFifoFI      R0, R1
LoadFifoFO      R0, R1
StoreFifoFI     R0, R1
StoreFifoFO     R0, R1
```

These instructions load and store the specified FIFO state from/to the address specified in R1. The amount and format of this information is implementation dependent. Again, separate instructions specify IEU and FEU input and output FIFOs.

The above floating load and store instructions are implicitly synchronized with both execution units while their integer counterparts are only synchronized with the IEU.

```
LoadCTX     R1
StoreCTX
SwapCTX     R1
SwapLT
```

These instructions perform context load, store, and swap. LoadCTX restores context from the a block of storage whose address is specified in R1. StoreCTX stores the current context at a known location for the current task. SwapCTX combines the previous two

operations. SwapLT, "swap to last task", is identical to SwapCTX, except that the address of the new TCB is implicitly the "last TCP pointer" (see Chapter 4).

The LoadCTX and SwapCTX instructions verify that R1 contains the address of a "TCB page" (see Chapter 4). Every piece of vital CPU state is loaded or stored by these instructions. Consequently, if context is saved and later restored, then the next instruction is executed as if no context save/restore had occurred.

All context switching instructions require implicit synchronization of the entire machine (see the SYNCH instruction in section 3.5.6).

3.5.3. Type Conversion

Instructions exist to convert between the three internal numeric data types. The opcodes and their corresponding instructions are:

```
CVTIF   R0 := R1
CVTID   R0 := R1
CVTFI   R0 := R1
CVTFD   R0 := R1
CVTDI   R0 := R1
CVTDF   R0 := R1
```

There interpretation is the obvious one, ConVerT from Integer, Floating, or Double, to another data type. Note that these instructions reference one register in the integer execution unit and one in the floating execution unit as appropriate. In addition, two instructions are included for transferring data unmodified between the two units:

```
TIF     R0 := R1
TFI     R0 := R1
```

These have the obvious "Transfer from Integer to Floating", and "Transfer from Floating to Integer" interpretations[1].

Note: although in general the IEU and FEU operate asynchronously, they are synchronized for the convert and transfer instructions -- that is, the basic sequential semantics of program execution are preserved.

3.5.4. Constraint Checking

There are two instructions for checking arithmetic constraints:

```
ASSERT      (R1 >= RL2) <= RL3
FASSERT     (R1 >= R2) <= R3
```

---

[1] Aside from the obvious "bit hacking" these instructions allow, they may also be used to get more streams to one of the executions units if the other has them free.

The ASSERT and FASSERT instructions determine if a value is within certain bounds. If it is not, a hardware RANGE trap is generated. Unlike the integer and floating point relationals, the two boolean values are AND'd together by these instructions and no condition code is enqueued. Note that it is possible to check only a single bound, for example,

```
ASSERT  (r8 >= r8) <= 13    to check only the upper bound, and
ASSERT  (r8 >= 13) <= r8    to check only the lower bound.
```

3.5.5. Field Manipulation

Fields can be moved within a word with the following instructions:

```
FLDMOV   R0 := R1, RL2, RL3      -- FieLD MOVe
FLDMOVX  R0 := R1, RL2, RL3      -- FieLD MOVe (sign-eXtended)
```

These may be thought of as being implemented using more primitive arithmetic and logical shifts[1]:

```
FLDMOV   R0 := (R1 lsl RL2) lsr RL3
FLDMOVX  R0 := (R1 lsl RL2) asr RL3
```

They may be used to perform several functions, including:

field extraction — an arbitrary signed or unsigned field may be taken out of a word and placed in an aligned, signed or unsigned, format. For example, "FLDMOVX r5 := r7,8,24" takes a signed byte from byte 1 of register 7 and transforms it into a signed 32-bit integer in register 5.

basic shifts — lsl, lsr, and asr are the basic functions within these instructions. By using the appropriate zero literal, each can be synthesized. For example, "FLDMOV r5 := r7,0,7" performs a 7-bit logical shift right. Since asl is provided in the basic instruction set, all basic shifts are represented.

Shift amounts of greater than 32 perform the same function as a shift by 32, namely the clearing (or setting if asr of a negative number) of the result. Field insert is not explicitly supported in the instruction set since it can be expressed in three instructions with the existing instruction set.

Finally, the "find first (different) bit" is provided:

```
FFB    R0 := R1
```

---

[1] "lsr" is "logical shift right"; "lsl" is logical shift left; "asr" is arithmetic shift right.

This instruction finds the first bit that is different from the sign bit, starting from the left. It returns the bit number of the bit found. If all the bits in the 32-bit word are the same, then the value 0 is returned. Taking this instruction as a function and applying it to some shorter examples, we get:

3 = FFB(000101001)
1 = FFB(010000010)
4 = FFB(111100101)
0 = FFB(111111111)

3.5.6. Constant Generation

There are two instructions which can be used to generate large constants:

```
LLH   R0 := <16 bit constant>
LUH   R0 := <16 bit constant>
```

Load Lower Halfword forms a 32 bit constant from the given 16 bits and stores the result in the register specified by the R0 field. Load Upper Halfword OR's the given constant into the upper 16 bits of the register specified by the R0 field. A 32 bit constant can therefor be generated in two instructions: LLH followed by LUH.

The 16 bit constants are formed by the concatenating the low order RL bit (bit 11) with the R1, R2 and R3 fields.

3.5.7. Condition Code Consumption

The instructions

```
ConsumeF
ConsumeI
```

consume one condition code as do conditional jump instructions. However, the value of the code consumed is immaterial.

3.5.8. PCW Access

The instructions

```
ReadPCW     R0
WritePCW    R1
```

allow the programmer to read and write the Program Control Word.

3.5.9. Pipeline Synchronization

The instruction

SYNCH

causes the processor to synchronize the IFU, IEU and FEU. In effect, it will inhibit instruction dispatch until a consistent, "as though the instructions were really executed sequentially" state is reached. See section E.5.6.

Certain special instructions, such as those performing context switching and type conversion, require implicit synchronization of the machine. These instructions can be thought of as always having an implied SYNCH instruction preceding them.

Other special instructions, as noted previously, require implicit synchronization of only one of the execution units with the IFU. Synchronization, in these cases, can be accomplished by inhibiting instruction dispatch to the appropriate execution unit until the contents of that unit's instruction buffer have been consumed.

Chapter 4. Task, Program, and System Framework

This chapter is concerned with that collection of architectural issues that support an "operating system" -- issues such as memory mapping, protection, control of input-output devices, interrupts and traps, system calls, and the hardware notion of "process" or "task".

WM approaches these issues in a manner that allows many of the functions normally associated with an operating system to be safely delegated to applications programs, thus eliminating unnecessary overhead from these functions and, perhaps more importantly, eliminating inefficiencies due to circumlocutions when the operating system provides the wrong facility. It is possible, for example, for an important data-base application to be safely allocated its own disk; this allows the data-base system to (1) avoid expensive operating system calls for data transfers, and (2) format the disk in a manner appropriate to the application.

The same WM facilities that allow operating system functions to be safely handled by applications, are also powerful structuring tools for building more conventional operating systems. Using the WM facilities it is possible to construct a variety of novel, secure, and highly efficient operating environments -- including an efficient multi-level military security system. At the same time, the WM hardware does not favor one style of operating system over another, and does not impose a performance penalty on simple systems; it is possible to implement a traditional system such as UNIX *very* efficiently.

Before starting into the details, we'll briefly introduce, and hopefully motivate, the concepts to be discussed:

Tasks: As usually defined, a task is a "thread of control". The WM hardware understands the notion of a task, and supports a hardware-defined "task

control block", TCB, to hold the state of the task when it is not executing. There are instructions to save, restore and "swap to" tasks; for purposes of these instructions, tasks are named by the address of their TCBs.

An "interrupt" is essentially a forced context swap; in general, different sources of interrupts may specify different tasks as their "handler tasks".

Domains: A task executes in an addressing domain. The domain consists of a flat, paged address space; each page in this space has two independent properties: (1) address translation information, and (2) typed protection information; these are defined by a "map table" and "protection table" respectively. Pointers to these tables are part of the task state in the TCB. These properties are separated so that, for example, two tasks can share the same address space but have different access to portions of that space.

The concept of a typed protection system is especially powerful. Each page is typed; only instructions appropriate to the type are permitted to reference a page, and the task must have rights appropriate for the instruction. For example, TCB is a type. The instruction to perform a context swap may be executed only on this type of page -- and then only if the task has "swap rights" to that particular page.

Entries: An "entry" is another type of page, and is a generalization of the "trap vector" of some other architectures. An "entry call", ECall, instruction may reference (only) an entry page and requires "call rights"; if executed, it has three effects:

(1) it changes the protection table pointer to that contained in the entry page (note, the map table pointer is not changed),
(2) it transfers indirectly through the location specified in the ECall, and
(3) it saves the prior protection table pointer and program counter in a special protected stack area in the TCB.

ECall provides the functionality of the "supervisor call" or "system call" instruction of other architectures, but does not make a rigid N-level distinction between "user" and "system". Instead, a particular operating system can provide a variety of, possibly nested, entry pages with greater, lesser, or merely different accesses. Entries in an entry page are pairs of words specifying a protection table pointer and the location where control is to be transferred.

The "EReturn" instruction unstacks and restores the protection state and PC of the caller.

Traps are merely ECall's on predefined locations (in "page 0").

Devices: A "device" is yet another hardware understood page type, and corresponds to the memory-mapped device registers in many architectures. Device-specific operations are performed by storing bit patterns into these locations (registers). Device status, and sometimes data, is accessed by reading these locations.

A goal of the WM design was minimal operating system overhead in critical applications -- such as accessing sensor data in embedded systems, and interprocess communication in multi-computer complexes. A key to achieving this is the ability of an application program to directly control a device, and do so safely. The ability to map devices into an application's address space provides the mechanism to achieve the goal.

The following sections discuss each of the above items in more detail.

4.1. Task State

Whenever a task is saved or restored, all of its processor state is transferred to or from its hardware-defined Task Control Block. This is an area in memory with room for:

(1) State visible to the program
- integer and floating point registers.
- Program Counter, PC.
- Program Control Word, PCW.
- Program Status Word, PSW.
- Cycle Counter, CC.
- Last TCB Pointer, LTP.
- Protection Table Pointer, PTP.
- Map Table Pointer, MTP.

(2) State visible only indirectly by the program
- input/output FIFO state.
- streaming state.
- other implementation-defined state

In general, the amount and description of the state is implementation-dependent. Only the TCB format for the state visible to the program is defined. Some of the architecturally-defined state is discussed below.

The PCW and PSW are two architecturally-defined CPU device registers. Implementations may add other registers (e.g., to control hardware diagnostics).

The Program Control Word collects a number of fields whose values affect the execution of a task, such as the bit which indicates whether the results of two relational operators in an instruction are AND'd or OR'd as well as the bits that enable/disable certain traps. The PCW consists of:

Bit#	Meaning
0	AND/OR relationals (AND == 1)
- -	Exceptions Enabled:
1	Assert
2	Integer Divide By Zero
3	Floating Divide by Zero
4	Integer Overflow
5	Floating Overflow
6	Integer Underflow
7	Floating Underflow
8	Cycle Counter Overflow
9	Raise Address
10	Raise Call
11	Raise Jump

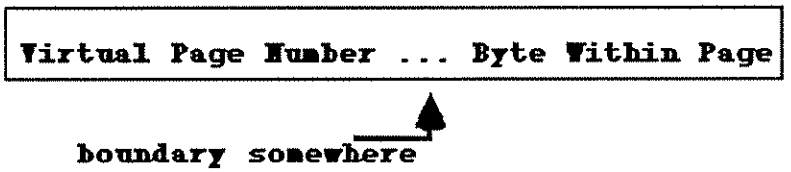
The Program Status Word collects a number of fields that reflect status of the task, such as the run/halt bit, the interrupt enabling bit, the priority and the condition FIFOs. For example, the PSW could include:

Bit #	Meaning
0	Run/Halt (run == 1)
1	Interrupts Enabled
2:5	Priority[0:3]
6:8	Integer Condition FIFO Bits
9:10	integer Condition FIFO Depth
11:13	Floating Condition FIFO Bits
14:15	Floating Condition FIFO Depth

The Cycle Counter is a 32-bit register that is incremented by one every cycle that the task executes. It may overflow (once every ~200 seconds with a 50ns cycle time), in which case an exception may be raised.

The Last TCB Pointer, LTP, in general points to a TCB. When an interrupt occurs, a forced context swap is performed and the LTP of the *new* task is set to point to the TCB of the task that was running at the time of the interrupt. Thus, in the case of nested interrupts, the LTPs form a chained "stack" of the suspended handlers; the SwapLT instruction (see Sec. 3.5.2) will resume the previous task.

The MTP and PTP are discussed in the next two subsections. Note first, however, that a task's virtual address space is divided into "pages" in the conventional manner. An address is divided into two parts, the virtual page number, and the byte within page address. The boundary between these parts is implementation-dependent, as is the structure of the tables (one-level, two-level, etc.). Pages must, however, be at least 512 bytes.



Assume N bits of virtual page number and 32-N bits that specify the byte within the page. Associated with every virtual page number is a protection table and map table entry, as described below.

4.2. Protection Table

Each task has a Protection Table that defines its memory access rights on a page-by-page basis. The Protection Table Pointer (PTP) in the TCB is either null (zero), or defines the base of this table and virtual address page numbers are used to index into it. If the PTP is null no type or rights checking is performed, otherwise protection is checked as specified below[1].

A Protection Entry is a byte, with the following format:

The first four bits define the page type. This field is interpreted as:

0000	Memory
0001	TCB
0010	Entry
0011	Device
0100-0111	reserved for hardware
1000-1111	reserved for software

Only the first four are hardware defined. Accesses to pages with reserved protection types raise a memory protection exception.

An access to "Memory" pages may either be reads, writes, or executes. The rights bits are R, W, and X, and determine if such operations are allowed, or if they result in memory protection exceptions.

Accesses to a TCB page may be reads, writes, or context save/restore/swap; the protection bits are correspondingly, R, W, and S. Note that saving context is not a privileged operation.

---

[1]The PTP may be null because protection is not implemented on a certain model of WM. In addition, however, PTP is null when the processor is first "booted" -- this corresponds to the "most privileged state".

Accesses to an Entry page may be reads, writes, or ECalls; the protection bits are correspondingly, R, W, and C.

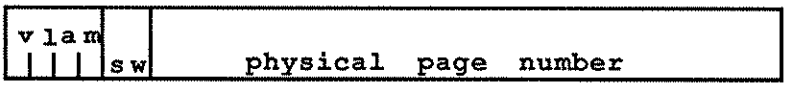
Accesses to Device pages may be only reads and writes, and the corresponding rights bits are R and W.

It should be noted that W-rights (write-rights) to TCB and Entry pages are *very* potent rights since they allow changing PTPs; normally these rights would be granted only to the most trusted portion of a system.

4.3. Map Table

Each task has a Map Table that defines its virtual-to-physical address translation. The Map Table Pointer (MTP) in the TCB is either null (zero), or defines the base of this table and virtual address page numbers are used to index into it. If the MTP is null, no translation is performed; otherwise translation proceeds as specified below[1].

Map table entries have the following format:



and their bits are interpreted as follows:

0	Valid - this page exists in physical memory
1	Locked --this page is locked into memory[2]
2	Accessed - this page has been read
3	Modified - this page has been written
4:5	Software usable/defined
6:31	Physical Page Number - 26 bits

The 26-bit physical page number is catenated with the Byte Within Page field to form the physical address. This limits the physical memory (without bank-switching) to an address space of 26 plus size(Byte Within Page) bits. This can be more than 32 bits.

Note: the WM architecture does not define the number of levels in the structure of the Map or Protection tables.

---

[1] The MTP may be null because virtual memory is not implemented on a certain model of WM. In addition, however, MTP is null when the processor is first "booted" -- this corresponds to the "unmapped processor state".

[2] The "locked bit" is a software convention; it is, however, checked by hardware when DMA IO transfers are specified. See Section 4.5.

4.4. Traps (Exceptions) and Interrupts

Non-programmed control flow changes can occur through two types of events:

interrupts — these are asynchronous with respect to instruction execution and may not be associated with the currently executing task.

traps — these are hardware-defined and are the direct result of an instruction just executed.

Interrupts are implemented as context-swaps to a handler task; traps are implemented as ECall's to handler entries. The terms "trap" and "exception" are used interchangeably.

4.4.1. Interrupts

Interrupts are best viewed as communication (messages) from asynchronous cooperating processes that happen to be implemented in hardware -- and as such, the task mechanism is the proper one for handling them. Thus, the effect of an interrupt is almost identical to a SwapCTX instruction; the only difference is that, on interrupts, the LTP (last TCB pointer) of the new task is set to point to the TCB of the task that was running at the time of the interrupt. A SwapCTX or SwapLT instruction does not set this register, and the LTP is loaded from the new TCB, just like all its other state.

The minor difference in behavior of interrupts and the SwapCTX/SwapLT instructions provides the functionality of "stacking" nested interrupts -- but leaves software free to use the LTP in clever ways (such as for a "run queue").

Note that each device capable of interrupting the processor must retain one or more addresses of the TCBs for the handlers of the interrupts it generates, and present this address to the processor along with the priority of the interrupt.

An interrupt (context swap) will be performed to the handler task if the priority of the interrupt is higher than that of the processor, and indeed, is the highest of all outstanding interrupts.

4.4.2. Traps

The page zero of a program's virtual memory (starting at address 0) must contain an Entry Page. A trap is implemented as an ECall on a hardware-understood location within this page. The hardware-defined locations are:

Except. #	Exception
0	(reserved)
4	(reserved)
8	Load While Input Streaming
12	Store While Output Streaming
16	Input FIFO Full
20	Input FIFO Empty
24	Condition FIFO Full
28	Condition FIFO Empty
32	(reserved)
36	(reserved)
40	Undefined Instruction
44	Assert Fault
48	(reserved)
52	(reserved)
56	Divide By Zero
60	Arithmetic Overflow/Underflow
64	(reserved)
68	(reserved)
72	Memory Protection Violation
76	Stack Index Negative
80	Jump on Stream Count while not streaming
84	(reserved)
88	Cycle Counter Overflow
92	Raise Address
96	(reserved)
100	(reserved)
104	Raise Call
108	Raise Jump
112	(reserved)
116	(reserved)
120	Page Fault
124	(reserved)

NOTE: THIS TABLE IS NOT CORRECT
CONSIDER IT AN EXAMPLE FOR NOW

The exceptions are ordered. If an instruction produces more than one exception, the one that vectors to the lowest memory location is selected. The other exceptions related to that instruction are nullified. An exception handling routine may itself cause an exception.

The EReturn instruction is used to return from an exception, just as from an ECall.

4.4.3. Initialization of the Machine

<<To be determined>>

4.5. Devices

Any of a wide variety of devices may be connected to WM and, mostly, each will be idiosyncratic with respect to its definition of its own control registers; each device, however, must conform to a few conventions:

1. The device must "know" the TCB address to which it is to interrupt. This may be wired-in for certain devices, or may be a settable register -- the latter being the preferred approach.

2. DMA devices must use "the zero-th register", the zero-th location relative to the device page, as the memory address register; non-DMA devices are advised not to use this location at all. The memory translation hardware recognizes stores into the zero-th location of device pages, and assumes the value to be stored is a virtual address; it then

    - verifies that the specified page is both valid and locked, and
    - stores the translated (physical) address rather than the virtual one.

3. DMA transfers may not cross a page boundary, thus the maximum size block that can be transferred is a page.

Appendix A. Summary of Restrictions on Valid Software

Programs to be executed by this architecture must have certain properties to guarantee correct operation. These conditions are summarized here, and some further ones added. Also, some recommendations about program structure are given.

Recommendations

- It is strongly encouraged that LOAD instructions be scheduled as early in the program code as possible.

- It is strongly encouraged that instructions containing relational operators be scheduled as early in the program code as possible.

Requirements

- A register being written is available as the outer operand in the next instruction and as an inner operand in the instruction after that. If specified as the inner operand in the instruction after it is to be written, the value before it was written is used.

- All data items must be aligned. Lower-order bits that should be zero will not be checked nor used.

- The L2 specified by a LoadM or a StoreM instruction must be less than the L3 specified.

- No LOAD (STORE) may be performed on an input (output) FIFO that is streaming.

- Instructions containing relational operations must be properly paired (dynamically) with conditional jumps that consume them. The number of instructions containing relationals preceding the associated conditional jump must not exceed the size of the condition bit FIFO.

- Certain sequences of operations may lead to a deadlock situation (each of the IFU, IEU and FEU unable to make progress). Such programs are invalid (see discussion below).

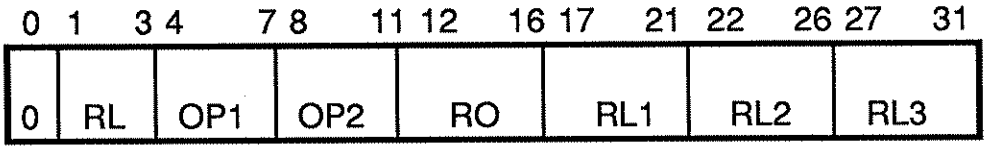
Since the various FIFOs that couple WM's components are finite, deadlocks are possible -- for example a simple sequence of LOAD instructions longer than the input FIFO will deadlock. Although the hardware will detect a deadlock and trap, the compiler (or assembly language programmer) is responsible for ensuring that they do not occur.

The minimum sizes of the various FIFOs are specified in such a way that it is always possible to construct a valid WM program. For example, the minimum size of the input FIFOs are 3 so that, at worst, an instruction requiring 3 source operands from memory can be emitted, and consume, its operands without blocking. A conservative compiler algorithm can delay loading FIFOs as long as possible and consume FIFO operands as soon as possible; it can be shown that such an algorithm always works.

This conservative algorithm will not produce optimal performance; an algorithm that does will be discussed in a separate report.

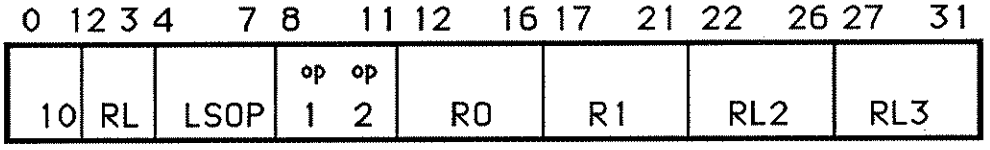
Appendix B.
Instruction Formats and Encodings

B.1. Integer Format and Operation Encodings



+	0010:	addition
-	0000:	subtraction
- '	0100:	reverse subtraction
*	0001:	multiplication
/	1000:	division
/ '	1100:	reverse division
asl	0011:	arithmetically shift left
eqv	0101:	bitwise EQUIVALENCE
or	0110:	bitwise OR
and	0111:	bitwise AND
=	1010:	equal
<>	1110:	not equal
<	1011:	less than
<=	1101:	less than or equal
>=	1001:	greater than or equal
>	1111:	greater than

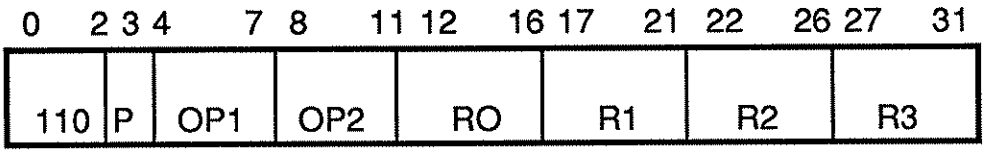
B.2. Load/Store Format and Operation Encodings



LB	0000:	Load Byte
LBX	0001:	Load Byte sign-eXtended
LH	0010:	Load Halfword
LHX	0011:	Load Halfword sign-eXtended
LD	0101:	Load Double
LW	0110:	Load Word

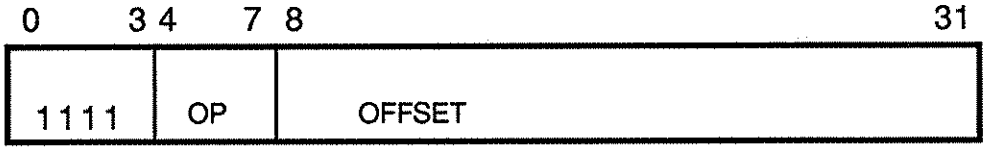
LF	0111:	Load Floating
SB	1000:	Store Byte
SBA	1001:	Store Byte with Arithmetic range check
SH	1010:	Store Halfword
SHA	1011:	Store Halfword with Arithmetic range check
SD	1101:	Store Double
SW	1110:	Store Word
SF	1111:	Store Floating

B.3. Floating Point Format and Encodings



+	0010:	addition
-	0000:	subtraction
- '	0100:	reverse subtraction
*	0001:	multiplication
/	1000:	division
/ '	1100:	reverse division
nop	0011:	pass the left operand
nop'	0111:	pass the right operand
tbd	0110:	to be determined
tbd	0101:	to be determined
=	1010:	equal
<>	1110:	not equal
<	1011:	less than
<=	1101:	less than or equal
>=	1001:	greater than or equal
>	1111:	greater than

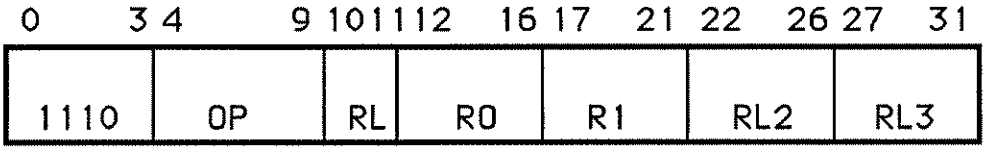
B.4. Control Format and Operation Encodings



Jump	1000:	unconditional Jump
JumpIT	1001:	Jump if Integer condition bit is True
JumpIF	1010:	Jump if Integer condition bit is False
JumpFT	1011:	Jump if Floating condition bit is True
JumpFF	1100:	Jump if Floating condition bit is False

JNI r0	0000:	Jump on stream count Not zero; Input FIFO r0
JNI r1	0001:	Jump on stream count Not zero; Input FIFO r1
JNO r0	0010:	Jump on stream count Not zero; Output FIFO r0
JNO r1	0011:	Jump on stream count Not zero; Output FIFO r1
JNI f0	0100:	Jump on stream count Not zero; Input FIFO f0
JNI f1	0101:	Jump on stream count Not zero; Input FIFO f1
JNO f0	0110:	Jump on stream count Not zero; Output FIFO f0
JNO f1	0111:	Jump on stream count Not zero; Output FIFO f1
Call	1101:	subroutine Call
ECall	1110:	Entry Call

B.5. Special Format and Operation Encodings



SinB	00 0000:	Stream in Bytes
SinBX	00 0100:	Stream in Bytes sign-eXtended
SinH	00 1000:	Stream in Halfwords
SinHX	00 1100:	Stream in Halfwords sign-eXtended
SinD	01 0100:	Stream in Doublewords
SinW	01 1000:	Stream in Words
SinF	01 1100:	Stream in Floating
SoutB	10 0000:	Stream out Bytes
SoutBA	10 0100:	Stream out Bytes with Arithmetic range checking
SoutH	10 1000:	Stream out Halfwords
SoutHA	10 1100:	Stream out Halfwords with Arithmetic range checking
SoutD	11 0100:	Stream out Doublewords
SoutW	11 1000:	Stream out Words
SoutF	11 1100:	Stream out Floating
StopAll	00 0001:	Stop All streaming operations
StopII	00 0101:	Stop Integer Input streaming on FIFO specified by R0
StopIO	00 1001:	Stop Integer Output streaming on FIFO specified by R0
StopFI	00 1101:	Stop Floating Input streaming on FIFO specified by R0
StopFO	01 0001:	Stop Floating Output streaming on FIFO specified by R0
LoadM	00 0010:	integer Load Multiple registers
FLoadM	00 0110:	Floating Load Multiple registers
StoreM	00 1010:	Store Multiple integer registers
FStoreM	00 1110:	Floating Store Multiple registers
LoadFifoII	01 0010:	Load Integer Input Fifo state

LoadFifoIO	11	0010:	Load Integer Output Fifo state
StoreFifoII	01	1010:	Store Integer Input Fifo state
StoreFifoIO	11	1010:	Store Integer Output Fifo state
LoadFifoFI	01	0110:	Load Floating Input Fifo state
LoadFifoFO	11	0110:	Load Floating Output Fifo state
StoreFifoFI	01	1110:	Store Floating Input Fifo state
StoreFifoFO	11	1110:	Store Floating Output Fifo state
LoadCTX	10	0010:	Load ConTeXt
StoreCTX	10	1010:	Store ConTeXt
SwapCTX	10	0110:	Swap ConTeXt
SwapLT	10	1110:	Swap to Last Task
CVTIF	00	0011:	ConVerT Integer to Floating
CVTID	00	0111:	ConVerT Integer to Double
CVTFI	00	1011:	ConVerT Floating to Integer
CVTFD	00	1111:	ConVerT Floating to Double
CVTDI	01	0011:	ConVerT Double to Integer
CVTDF	01	0111:	ConVerT Double to Floating
TIF	01	1011:	Transfer Integer to Floating
TFI	01	1111:	Transfer Floating to Integer
ASSERT	10	0011:	integer ASSERTion
FASSERT	10	0111:	Floating ASSERTion
FLDMOV	10	1011:	FieLD MOVe
FLDMOVX	10	1111:	FieLD MOVe sign-eXtended
FFB	11	1101:	Find First different Bit
SYNCH	11	1111:	SYNCHronize IFU, IEU, and FEU
JumpI	11	0011:	Jump Indirect
CallI	11	0101:	Call Indirect
EReturn	11	0111:	Entry call Return
LLH	11	1001:	Load Lower Halfword
LUH	11	1011:	Load Upper Halfword
ConsumeI	11	0000:	Consume Integer condition code
ConsumeF	11	0001:	Consume Floating condition code
ReadPCW	10	1001:	Read Program Control Word
WritePCW	10	1101:	Write Program Control Word

Appendix C. WM Assembly Language

There are two varieties of assembler that might be written for the WM machine. The first accepts fully defined instructions, such as a compiler might produce, and converts them to machine code quite rotely. The second type of assembler is geared toward human readers and writers of WM code. The assembler would expand abbreviated instructions, check validity of statements, and rearrange the result to minimize NOPs and optimize conditional jumps. What we define here is somewhere between these extremes -- but probably closer to the first. We expect that the current specification will be a proper subset of the more humane assembler specification.

C.1. Lexical Structure

A WM assembly "module" consists of one or more blank lines, instructions, or directives; it must begin with the directive

.module <identifier>

and must be terminated by the directive

.end

Instructions and directives must be wholly contained on a single "line", and either may be "labeled". Instructions or directives are labeled by prefixing them with

<identifier>:

Comments begin with the characters "--" and continue to the end of the line on which they appear.

Identifiers, which are used for labels, obey the Ada syntax, except that the special symbols ".", "$", "%", and "?" may be used anywhere that a letter is permitted in Ada. Letters appearing in an identifier are case-insensitive.

The only reserved identifiers are: (1) the instruction mnemonics, (2) "r0", "r1", ..., "r31", "f0", ..., "f31" (which are used to name the registers and FIFOs), and (3) the directive names mentioned later.

Numbers, both integer and floating point, also obey the Ada syntax. Specifically, based numbers are written

<base>#<value>#

where <base> is the base (in decimal) and must be in the range 2..16. The letters "a".."f" may be used in the <value> to represent the digits 10..15 when the base exceeds 10.

C.2. Instructions

Standard integer and logical instructions are written in the form:

```
int   R0 := (RL1 op1 RL2) op2 RL3
```

":=" is always the assignment symbol and the parentheses are required. Various abbreviations are allowed and appropriate NOPs[1] will be inserted by the assembler; for example

```
int   R0 := (RL2) op2 RL3      -- op1 defaults to a nop
int   R0 := (RL1 op1 RL2)      -- op2 defaults to a nop
int   R0 := RL3                -- both are nops, RL3 is the outer operand
int   R0 := (RL2)              -- both are nops, RL2 is an inner operand
```

The op's must be recognized as valid ones Simple expressions involving only constants and the machine's integer/logical op set may be included in the place of a literal. Such expressions must evaluate to a valid literal (0..31) and may include square brackets for parentheses.

As a convenience, statements may also be written in the commuted form

```
int   R0 := RL3 op2 (RL1 op1 RL2)
```

In the event that op2 is non-commutative and its "reverse form" exists, the assembler will make the proper substitution. That is, one may write

```
int    r8 :=  r7 - (r4 asl 7)
```

and achieve the same effect as

```
int    r8 := (r4 asl 7) -' r7
```

Finally, since integer instructions always begin with an assignment to an integer register (r0..r31), the prefix "int" may be dropped. The following are all valid integer/logical instructions:

---

[1] In the case of the integer instructions, OR with literal zero is a fine NOP.

```
int     r8 := (r4 asl 7) - r7
int     r5 := r8 /' (r10 <>8#15#)
        r14 := 31
        r21 := (#2#1111 and r12) = 15
```

Floating point instructions are a minor variation on this. The differences are:

- they are prefixed with the word "float" or "double" ("float" can be dropped)
- literals cannot be used
- register specifiers in double-precision operations specify register pairs

The following are valid floating point instructions:

```
float   f7 := f12 - (f9 + f5)
double  f14 := (f24 /' f16)
        f11 := (f14 * f0) + f6
```

Load and store instructions are similar to integer/logical instructions. The differences are:

- only +, -, *, and asl are valid operators
- the instruction is preceded by an instruction code, such as LW (load word)

The following are valid load or store instructions:

```
LBX     r31 := (r7 asl 2) + r12
SD      r4 := (r0 + r1) - 14
```

Jumps and calls also begin by specifying an instruction code and continue with a target specification. The PC-relative target may be specified with a simple label combined in an expression with constants. Jumpl specifies a register which contains its target's address. The following are all valid control flow instructions:

```
JUMP        exit
JumpIF      case+7
Call        DiskHandler
jumpi       r21
```

The last type of instruction that can be specified are the special instructions; these start with instruction codes and have a comma separated list of operands (in the order R0,...,R3); only as many operands as are required for a particular instruction need be written.

C.3. Directives

Directives specify assembly-time information of various kinds; each is discussed separately.

C.3.1. Modules

A module is a linkable-unit. As noted in the introduction, a module begins with

.module <identifier>

and ends with

.end

A module must be wholly contained within a single file, however a file may contain several modules. Module definitions may not be nested.

C.3.2. Sections

Instructions or data may be placed in any one of a number of "sections". Any code or data following the directive

. section <identifier>

will be placed in the named section until another ".section" directive is encountered. Each named section is distinct. Loader directives must be supplied to control the relocation of sections into the virtual address space. Sections, however, always will begin on a page boundary.

C.3.3. Alignment

The directive

.align <number>

will force enough zero bytes to be inserted so that the following instructions or data will be inserted at a byte address that is a multiple of <number>; thus ".align 2" forces half-word alignment and ".align 8" forces double-word alignment.

C.3.4. Data Definitions

Four directives are provided for allocating (static) data space and initializing it (each of the "list of" items below refers to a comma-separated list):

.block	<integer number>
.byte	<list of <integer number> or <character>>
.half	<list of <integer number>>
.word	<list of <integer number>>
.float	<list of <floating number>>
.double	<list of <floating number>>
.string	<list of <string>>

where

".block"	allocates the specified number of bytes without initializing it.
".byte"	allocates a sequence of bytes and initializes them to the specified values.

".half" allocates a sequence of half-words, and initializes them to the specified values.

".word" allocates a sequence of words, and initializes them to the specified values.

".float" allocates a sequence of words, and initializes them to the specified values.

".double" allocates a sequence of double-words, and initializes them to the specified values.

".string" allocates a sequence of variable sized chunks, where the length of the chunks is that of the specified strings, and initializes them to the specified values.

C.3.5 Equivalence

The ".equ" directive is used to assign assembly-time values to labels; specifically,

<label>: .equ <assembly time expression>

will assign the value of the expression to the label.

C.3.6. Global and External Labels

Labels are, by default, local to the current module. Two directives are provided to override the default:

.global <identifier list>

and

.external <identifier list>

Identifiers appearing in a ".global" directive must be defined in the current module. These identifiers become visible to the linker and may appear in a ".external" directive of another module.

Identifiers appearing in a ".external" directive must NOT be defined in the current module. They must, however, be defined in another module and appear in a ".global" directive of that module.

Appendix D. Streaming Versus Vectorizing

D.1. The Issue

Only certain classes of algorithms may be vectorized (fitted to the vector instructions of a supercomputer). Certain manipulations to make a computation "fit" a vector operation on a computer are often necessary. Many vector computers use finite-sized vector registers, like the Cray architectures. Data sets must be broken up into segments of this size. Further, very recently computed values, that are further down in the pipeline, cannot be retrieved for the computation of "later" values. Other machines use memory-to-memory vector operations, in which case a similar "threshold", related to the write-to-memory/read-from-memory pipeline length, exists.

Two simple pieces of code sum up the basic problem with vectorizing:

```
FOR i IN 1..1000 LOOP
     a(i) := a(i+1) * 2;
END LOOP;
```

```
FOR i IN 1..1000 LOOP
     a(i) := a(i-1) * 2;
END LOOP;
```

The first example can be vectorized -- the second cannot. In the second, a value to be computed depends on the value just computed. In a pipelined machine that vectorizes, this value is caught in the torrential current of the pipeline.

Both examples can, however, be streamed. The code for each, on the WM architecture, is shown below (note, here and later we use notations such as "a(2)" to represent an address; in practice such addresses will have to be computed, as SL- or FP-relative for example).

```
  LLH      r9 := 1000
  SinW     r0, a(2), r9, 4

  SoutW    r0, a(1), r9, 4
LOOP:int   r0 := (r0 + 0) asl 1
  JNi      r0  LOOP
```

```
  LLH      r9 := 1000
  LW       a(0)
  int      r7 := r0
  SoutW    r0, a(1), r9, 4
LOOP:int   r7 := (r7 + 0) asl 1
  int      r0 := r7
  JNo      r0 LOOP
```

Note that the second example contains one additional instruction to hold the "state" between iterations -- that is, the array element from the prior iteration. In general, streaming loop bodies involving recurrences will use N registers, where N is the number of elements involved in the recurrence.

D.2. An Example, Matrix Multiply

A more practical example of streaming is found in the inner loop of matrix multiply -- which is similar to several other matrix and vector computations. The Ada form of the loop is:

```
for k in 1..N loop
    t := t + a(i,k)*b(k,j);
end loop;
```

Which translates trivially into the WM code

```
        SinW    r0, a(i,1), N, 4
        SinW    r1, b(1,j), N, 4*N,
LOOP:   int     t := (r0*r1) + t
        JNi     r0 LOOP
```

It is interesting to compare this code with that of a vector machine. Note, in particular, that although there are two instructions in this inner loop, the jump takes zero time -- so a new partial sum is formed each cycle. This performs as well as any vector machine, even one with a built-in "dot product" operation. Moreover, because the vectors can be of arbitrary, run-time determined, size, no "strip mining" need be done, and there are no awkward boundary conditions.

D.3. Another Example, an IIR Filter

Another interesting example is that of an infinite impulse response filter. It cannot be vectorized because it involves recurrences as illustrated above. With a little transformation the WM architecture performs well on it.

```
FOR i IN 3..500 LOOP
    a(i) := (b(i) + a(i-1)*k1 + a(i-2)*k2) / 2;
END LOOP;
```

The corresponding WM code is:

```
        LW      a(2)
        LW      a(1)
        LLH     r5 := 498
        SinW    r0, b(3), r5, 4         -- Stream Word In FIFO 0
        SoutW   r0, a(3), r5, 4         -- Stream Out FIFO 0
                r7 := k1
                r8 := k2
                r10 := r0               -- a(i-1) := a(2)
                r11 := r0               -- a(i-2) := a(1)
LOOP:           r9 := (r10*r7) + r0     -- b(i) + a(i-1)*k1
                r9 := (r11*r8)+r9       -- + a(i-2)*k2
                r11 := r10              -- a(i-2) := a(i-1)
                r10 := r9 / 2           -- a(i-1) := a(i)
                r0 :=  r10              -- a(i) :=
        JNi r0  LOOP                    -- loop if not done
```

For comparison purposes, it is worth noting that this inner loop involves 10 RISC-like operations in 5 cycles.

D.4. Another Example, FFT Inner Loop

To explore another example of how code can be streamed, consider a fast fourier transform. We will not explain the FFT algorithm since it is the form of the loop that is important. The inner loop from a particular Ada version uses single dimensional vectors w, z and e of complex numbers with a real part (rp) and an imaginary part (ip):

```
LOOP
    w(i+k).rp := z(i).rp+z(m+i).rp;
    w(i+k).ip := z(i).ip+z(m+i).ip;
    w(i+j).rp := e(k+1).rp*(z(i).rp-z(i+m).rp)
                   -e(k+1).ip*(z(i).ip-z(i+m).ip);
    w(i+j).ip := e(k+1).rp*(z(i).ip-z(i+m).ip)
                   +e(k+1).ip*(z(i).rp-z(i+m).rp);
    i := i+1;
    EXIT WHEN i > j;
END LOOP;
```

This loop iterates over i with j, k and m held constant. The resulting simplified loop after constant folding is rewritten as:

```
LOOP
    wk(i).rp := z(i).rp + zm(i).rp;
    wk(i).ip := z(i).ip + zm(i).ip;
    wj(i).rp := ek.rp * (z(i).rp - zm(i).rp)
                -  ek.ip * (z(i).ip - zm(i).ip);
    wj(i).ip := ek.rp * (z(i).ip - zm(i).ip)
                +  ek.ip * (z(i).rp - zm(i).rp);
    i := i+1;
    EXIT WHEN i > j;
END LOOP;
```

Eliminating redundant reads, and introducing symbolic register names, we get:

```
LOOP
    zrp = z(i).rp;
    zmrp = zm(i).rp;
    zip = z(i).ip;
    zmip = zm(i).ip;
    wk(i).rp := zrp + zmrp;
    wk(i).ip := zip + zmip;
    wj(i).rp := ek.rp * (zrp - zmrp)
                -  ek.ip * (zip - zmip);
    wj(i).ip := ek.rp * (zip - zmip)
                +  ek.ip * (zrp - zmrp);
    i := i+1;
    EXIT WHEN i > j;
END LOOP;
```

An optimizing compiler would go further, eliminating the two common subexpressions:

```
LOOP
    zrp = z(i).rp;
    zmrp = zm(i).rp;
    zip = z(i).ip;
    zmip = zm(i).ip;
    wk(i).rp := zrp + zmrp;
    wk(i).ip := zip + zmip;
    zrd := zrp - zmrp;
    zid := zip - zmip;
    wj(i).rp := ek.rp * zrd  -  ek.ip * zid;
    wj(i).ip := ek.rp * zid  +  ek.ip * zrd
    i := i+1;
    EXIT WHEN i > j;
END LOOP
```

On the WM machine, this example can be streamed. Assuming register rj holds the count to control looping, assembly code for this example, using symbolic register names, is:

```
        SinF     f0, z(i).rp, rj, 4
        SinF     f1, zm(i).rp, rj, 4
        SoutF    f0, wk(i).rp, rj, 4
        SoutF    f1, wj(i).rp, rj, 4
loop:   zrp :=   f0;
        zmrp :=  f1;
        zip :=   f0;
        zmip :=  f1;
        f0 :=    zrp + zmrp;
        f0 :=    zip + zmip;
        ft1 :=   (zrp - zmrp) * ekrp;
        ft2 :=   (zip - zmip) * ekip;
        f1 :=    (ft1) - ft2;
        ft1 :=   (zip - zmip) * ekrp;
        ft2 :=   (zrp - zmrp) ekip;
        f1 :=    (ft1) + ft2;
        JNI f0   LOOP
```

Not counting the rj variable, 8 registers are used. This inner loop would execute 12 cycles, performing a total of 22 operations, per iteration.

This is the quality of code that could be expected of an optimizing compiler that knows how to create streaming code, but without heroic effort. Further hand-tuning is possible, with the following assembly code as a result:

```
      SinF    f0, z(i).rp, rj, 4
      SinF    f1, z(i+m).rp rj, 4
      SoutF   f0, w(i+k).rp, rj, 4
      SoutF   f1, w(i+j).rp, rj, 4
loop: zmrp := f1;
      zrm :=  (f0) - zmrp;        -- (zrp-zrmp)
      zmip := f1;
      zim :=  (f0) - zmip;        -- (zip-zimp)
      f0 :=   (zrm + zmrp) + zmrp--  = zrp+zrmp
      f0 :=   (zim + zmip) + zmip--  = zip+zimp
      ft :=   ekrp * zrm;
      f1 :=   (ekip * zim) -' ft;
      ft :=   ekrp * zim
      f1 :=   (ekip * zrm) + ft;
      JNI f0  LOOP
```

Only 7 general purpose registers are used here, and there are 10 cycles per iteration. Even though it appears that extraneous computation is made in order to reduce the instruction count (and loop latency) by 2, a total of 22 operations are still performed.

D.5. Other Applications of Streaming

Hopefully it should be obvious that streaming is useful for more than just numerical/vector processing. For example,

- string processing: string comparison, finding a substring, etc.
- sorting: for example, a merge-sort can be done in Log(N) streaming operations (using the two output streams).
- decimal arithmetic: (COBOL implementors, you'll love it!)

Ada causes real problems for vector machines, especially in connection with exceptions. Because there is no mechanism for inserting constraint checks into vector operations, and because machine checks, (such as overflow), are generally not "raised" by the hardware until after the vector operation is completed -- ugly, large, and non-optimal vector code must be generated. No such problem exists for streaming, however. Since the "vector" operation is programmed, it can include appropriate ASSERTs where required.

Appendix E. Rationale Behind Chosen Operation Sets

E.1. Integer & Logical

This set was chosen to contain the most frequently needed operations that could be combined with elegance. With the exceptions of asl, the operators are commutative. This was a goal, so that a code generator would be able to encode both (A op (B op C)) and ((A op B) op C) in single instructions. It is believed that code density benefits as well. The two exceptions were made following thoughtful justification (given below).

The chosen 16 operations, and their rationale, follow.

+	Addition is the most frequently needed operation. This is the only way is synthesized, short of painfully long boolean operations.
-	Subtraction also is quite common.
-'	Reverse subtract exists so that Rd := (R1 op R2) -' R3 could be performed. Without it, this computation would require two instructions.
and	A common boolean operator, useful also for bit clearing and testing.
or	A common boolean operator, useful also for bit setting.
eqv	This is is a non-traditional choice and represents considerable exploration among options. Besides AND and OR, the boolean operations of most concern are NOT and XOR. (These also happen to be those specified in Ada.) XOR is relatively uncommon, but

provides hard to synthesize functionality. The EQV (equivalence) function easily synthesizes NOT Z as 0 EQV Z. It also provides XOR in two quick steps. Because encoding space was tight, it seemed that EQV provided the required functionality in a single operator.

*,/,/' These operators are included because of their importance.

asl This function is vital for efficient scaling by a power of two. Since this is so common, (especially in address arithmetic), it was included. Its commutated form (asl') is not included, however. This is partially due to lack of encoding space.

<>,=,<,<=,>=,> If two relationals appear in the same instruction, their condition bits are AND'd or OR'd as indicated by a bit in the PCW. This allows Ada's "in <range>" operation to be performed in a single instruction when the relationals are being OR'd.

The "result" of a relational is its left operand. Since the relationals can be "commuted" by substituting the appropriate operator (e.g., ">" => "<=") this may seem arbitrary. However, using this convention allows the result of the inner computation by op1 to be the result of the instruction.

Other shifts are not explicitly included, but field extract (signed and unsigned) can synthesize any shift desired. Also not included were abs, rem, rem', mod, and mod'.

E.2. Floating Point

These operations are the basic arithmetics and relationals. The only "unusual" choices involve the two monadic operations. There is room for two additional operations, but they have not yet been selected.

E.3. Load/Store Instructions

Data is manipulated in WM's registers as full 32-bit (or 64-bit floating point) quantities; it is therefore necessary to provide a full complement of load/store operations to map smaller byte and halfword data to/from memory.

The four integer operations provided as part of the load/store operations are statistically the most frequent operations used in performing address arithmetic.

E.4. Control Instructions

The rationale for the various control instructions should be fairly obvious. Note, however, that the existence of both true and false conditional branches is closely related

to the choice of the relational operators in the integer and floating point instructions. Specifically, the "reverse" form of the relationals (ones that would produce their right operand) were not included since, for example, "a > b" could be transformed into "b<=a" so long as the sense of the branch was also inverted.

The Jumpl, jump indirect, instruction was included primarily for case-table and subroutine-return jumps; it can, however, be used as well when the distance of a jump may exceed 2^{26} bytes.

As noted several times, the ECall and EReturn instructions provide the functionality of "supervisor call" on other architectures. By setting up a single entry page, and associating an "all rights to everything" protection table with that entry, one gets the same effect as a "user/kernel mode" system at about the same cost. By setting up a series of nested, each more privileged than the last, entries, one gets the effect of a "ring" system. The most general use of entries, however, will involve non-hierarchical protection, as for example, in a compartmentalized military security system.

E.5. Special Instructions

The special instructions will be discussed in groups: streaming, state manipulation, type conversion, constraint checking, field manipulation, and SYNCH.

E.5.1. Streaming Instructions

Streaming was included in the machine to achieve several related effects:

- eliminate load and store instructions from loop bodies,
- eliminate loop counting instructions from loop bodies,
- allow the memory system to exploit the regularity of the memory reference pattern

All of these apply, of course, just in the case of vector-like data; however this form of data is sufficiently common to be of substantial interest. It is interesting to speculate whether, given the streaming mechanism, it would be worthwhile to reorganize data structures to exploit it -- e.g., use a polish representation of syntax trees so that linear traversals are possible.

The 14 "start streaming" instructions mirror the 14 load/store instructions and are included for the same reason. The "stop streaming" instructions are included primarily to permit indefinite-length streaming operations. Consider, for example, a loop to find the first instance of the character "z" in a string; the string can be streamed through a compare-and-count loop, but once the position of the "z" has been determined, the streaming stops.

E.5.2. State Manipulation Instructions

WM has a lot of state. It also has a fairly sophisticated notion of a task, and a rich set of state-manipulation instructions. A larger state speeds processing of individual programs at the expense of increased context-switching overhead between programs; we

opted to accept the additional context-switch overhead, but to ameliorate it by providing instructions for saving and restoring state as rapidly as possible.

The purpose of most of these instructions should be obvious. Note, however, those for saving and restoring FIFO state. In a sense, these are the obvious analogs of those for saving and restoring registers; there are two important differences, however:

- register save/restore could be done with explicit load/store instructions. FIFO save/restore cannot because portions of the state are not accessible, and
- at certain points, e.g. at entry to a trap handler, the state of the FIFOs is not known

E.5.3. Type Conversion Instructions

The rationale for these instructions is the obvious one.

E.5.4. Constraint Checking Instructions

The only difference between these instructions and the obvious range-checking integer and floating-point instructions, is that they trap rather than setting the condition code. This provides a certain consistency between hardware- and software-detected range violations.

E.5.5. Field Manipulation Instructions

These instruction provide both field extraction and shift functions. Given that there wasn't adequate encoding space in the integer instruction format for these shifts, and that they are relatively less common, this appeared to be a reasonable solution.

E.5.6. The SYNCH Instruction

In general, WM has what have been called "imprecise interrupts", which is actually a misnomer -- "imprecise traps" would be more accurate. In any case, because of the potential for asynchronous execution of the IEU, FEU, and IFU, at the time of a trap from one unit (say a floating-divide-by-zero) the other units may be executing in quite another portion of the program; there is no guarantee of a simple mapping of the processor state to a place in the user's program. Often this does not matter, since

- the trap is a fatal error, or
- the trap can be handled locally without such a mapping (e.g., page faults)

However, in the case it *does* matter, the SYNCH instruction can be used to force synchronization. It can, for example, be used by Ada programs to ensure that programmer-visible side-effects are synchronous with constraint checks. Liberal use of this instruction may, however, have an adverse effect on performance -- possibly a significant one -- since it blocks execution of multiple instructions per cycle.

E.6. Instructions Specifically *Not* Included

A few instructions typically found in other machines are not present in WM. Trivial examples of this are NOP and HALT.

There are no integer/logical NOPs; they can be synthesized. There do exist floating NOPs. Note that both the normal and reverse form exist in order to permit one operator assignments in the face of the data dependency rule[1].

Note that integer (floating) NOPs affect only the IEU (FEU) -- not the whole machine, and in principle may not consume any extra cycles. Thus the are useful for simple expressions (0 or 1 operators) or data dependency synchronization. They are *not* useful for delaying the machine. If that's what your goal is, use SYNCH.

There is no machine HALT instruction. However, the machine itself is an accessible device. The machine can be halted by writing the appropriate bit in the Program Status Word. The other aspects of machine state are set in the same fashion.

There are no operations to support unsigned arithmetic or (often related) multi-precision arithmetic (e.g. operations such as "add with carry"). These were omitted in part because there wasn't enough encoding space for a complete set, but also because they are logically unnecessary and often provided (only) for historical reasons.

Since unsigned operations are often used for address arithmetic, WM's addresses are signed. That is, each of the signed integers $-2^{-31}..+(2^{31}-1)$ names a byte in memory. Among other things, this eliminates the schizophrenia in most machines regarding signed arithmetic (e.g. indexing is usually (and incorrectly) a signed operation).

It should also be noted that there are no instructions to support memory-based synchronization, e.g., "test-and-set". This functionality will be provided by specialized devices accessed through "device pages". There are several reasons for this decision. First, it simplifies the memory system since there is no need to support "read-modify-write" operations. Since WM's memory system will be strained just to supply data and instructions at the rate that the processor consumes them, this simplification is important. Second, it provides the opportunity to create high-performance application-specific synchronization and/or communication mechanisms. For a critical real-time application, for example, one could implement an entire message system in hardware.

---

[1] Note: (a nop' b) nop c does not compute the same value as (c nop a) nop' b due to the data dependency rule.

Appendix F. Constructing WM Software Suggestions and Rationale

F.1. Calling Sequence and Register Conventions

The following is a proposed set of register conventions and calling sequence. Consider it a strawman to test feasibility, and not the final/official one.

The conventions with respect to register usage are:

- r 0 — input integer FIFO; always assumed empty at calls
- r 1 — input integer FIFO; contains 1st N parameters on calls, and the result(s) on returns
- f 0 — input floating FIFO; always assumed empty at calls
- f 1 — input floating FIFO; contains 1st N parameters on calls and the result(s) on returns
- r 2 — SL; not modifiable
- r 3 — SI
- r 4 — PC saved here by the call instruction
- r 5 — FP (frame-pointer; software convention)
- r 6 — HP (exception-handler pointer; software convention)
- r 3 1 — identically zero; writes to this register have no effect.
- f 3 1 — identically zero; writes to this register have no effect.

This leaves r8-r30 and f2-f30 available for any use the compiler wants to make of them; 52 unassigned registers is LOTS. Note that with this many registers, it probably makes sense to have a software convention that certain (high numbered) registers are not saved/restored across a call; this can save significant call overhead in some cases; such a convention is not defined here.

A call consists of:

```
r1 := p1   -- 1st parameter
..
r1 := pn   -- Nth parameter
call subr  -- implicitly,  r4 := PC
```

A routine body consists of the following (assuming that a display does not need to be saved):

```
subr:       r7 :=   (SL+SI)          -- address of save area (TOS)
       StoreM r7, r3, r#             -- r# = highest reg used in this rtn
            SI :=   (k1 + k2) + SI   -- k1 = #regs to save,
                                     -- k2 = # stack locals
            r5 :=   (SL+SI) - k2     -- set the frame pointer
            r6 :=   errXit           -- exception unwind
            <...>                    -- the actual body
            r7 :=   (SL+SI) - [k1+k2]
       LoadM  r7, r3, r#             -- restore the regs, incl caller's FP
       Jumpl  r 4                    -- normal return to the caller
errXit:     r7 :=   (SL+SI) - [k1+k2]
       LoadM  r7, r3, r#             -- restore the regs, incl caller's FP
       Jumpl  r6                     -- return to the caller's handler
```

Note: the issue of exception handling are discussed in the next section.

Also Note: The code at "errXit" can be shared between subroutines under some circumstances. Notably, with the large register set of WM, we expect relatively few cases where locals are allocated to the stack. In this case, all routines saving N registers will have the same epilogue and only one copy is required.

F.2. Ada Exceptions

Ada requires nested exception handlers; they are statically nested within a subroutine and dynamically nested as routines are called. Several schemes are well known for handling this structure, but one seems most natural for the WM architecture.

Register r6, by software convention, always holds the address of the current handler. It is saved/restored by the normal prologue/epilogue of subroutines, and either points to a specific handler or the "unwind" handler for the current routine ("errXit" in the previous discussion of the calling conventions). Thus,

- routine prologs are responsible for setting r6 to its default value (i.e., the current routine's "unwind" handler,
- entry to a block with an exception handler is responsible for setting r6 to the address of the handler; note that this is a static, compile-time known address,
- exit from a block with an exception handler is responsible for setting r6 to the address of the enclosing handler (or unwind handler if there is no enclosing block with a handler); again, this is a static, compile-time known address.

This scheme implies two instructions of "overhead" per block (that has a handler) and avoids any "searching" overhead when an exception is raised. It also allows full optimization to be applied within the block in contrast to schemes involving a "map".

Note that hardware-detected exceptions, such as overflow and divide-by-zero, will "trap" to the run-time system. Except for this, however, they can be handled just as a software-defined exception since r6 will point to the correct handler.

F.3. Parameter Passing

It is proposed that FIFO 1 should be used to hold parameters passed between procedures. This is a significantly different mechanism than those traditionally used. The simplest method used is to pass parameters of the procedure via a stack in memory. In most cases this results in 2N memory accesses for N parameters passed, N to write the parameters onto the stack and another N to access them for computation.

Parameters may be passed in registers on machines with a sufficient number of them. In such cases, the compiler enforces a convention about which registers will be maintained for parameters between procedures. If a number of procedure calls are made, then the parameter-passing registers must be saved. This is to make room for the next set of parameters for the procedure to be called. A procedure call can cost up to 2N memory references with this scheme as well.

A more recent parameter passing mechanism is the use of overlapping multiple register sets. With such a scheme, both incoming and outgoing parameters are "seen" by a procedure. Hence, 2N registers are filled just for passing N parameters. Such multiple register schemes are also deemed less-attractive; in recent machine implementations the large register arrays slow down basic machine cycle times.

Passing parameters via a FIFO, as proposed for this architecture, offers advantages over these other methods. Computed parameters need not displace existing register values and parameters used once need not consume a register space. A procedure must empty its incoming parameters before calling another procedure, but this overhead never appears for "leaf" procedures.

F.4. Loop Control

The semantics of the Ada 'for' statement poses a problem for many computer architectures; happily it is a non-problem for WM. Consider

```
N: integer;
...
for i in 1..N loop
  <body>
end loop;
```

In such a case, it is possible that N assumes the value integer'last, (the largest positive number). The value of i must be compared to N before being incremented at the bottom of the loop -- if it were incremented first, there would be an overflow an raise an exception; this is not Ada. This consideration precludes the use of the "add one and branch" class of instructions on most computers. On WM, however, the instruction

```
i := (i < N) + 1
```

has exactly the right semantics.

F.5. The Display

It was noted earlier that a natural use of the area above the Stack Limit register was for stack expansion, software TCB, etc. In effect, the SL acts as a natural base register for the static storage of the task, e.g., the constant pool, own variables, and display. The "display" mentioned here is perhaps a bit richer than the word typically connotes. Specifically, it is suggested that it contains two parts:

- the procedure-frame display, as is common. This portion needs to be updated as part of the normal procedure prolog/epilog in order to support "up-level" addressing of locals and formals of enclosing procedure declarations.
- the SL display. This portion would be created (only) at task creation, would contain the SL values for statically enclosing tasks, and can be used to access the static storage of those inclusion tasks.

F.6. Absolute Addresses

As is obvious from a quick scan of the WM instruction formats (Appendix B), there is no way to include a full 32-bit address in an instruction, and hence, no convenient way to use absolute addresses. If such addresses are desired, they will have to be placed in the constant pool and loaded (SL-relative) into a register before being used.

The preferred style of addressing on WM, however, is to avoid absolute addresses wherever possible. Specifically,

- the large, 24-bit, jump displacements should be adequate for all programs of practical concern,
- task-specific static variables can be addressed SL-relative, and
- the SL-display discussed in the previous section can be used for inherited static variables of parent tasks.

Programs which utilize these facilities, can be easily made "position independent".