The WM Computer Architectures
Principles of Operation

Wm. A. Wulf

Computer Science Department
University of Virginia
Charlottesville, Va.

January 1990

Abstract

This report is a definition of, and partial rationale for, the WM family of computer architectures.

The primary objective behind the WM design was concurrency -- concurrency at several levels. Since no single style of concurrency is suitable for all applications, several styles coexist and complement each other in the design. First, WM supports micro-concurrency at the instruction level; that is, it facilitates the execution of several scalar instructions concurrently. Second, WM supports vector processing; that is, it has single instructions that apply the same operation to a collection of data items. Finally, WM was designed to be a node in a multiprocessor, a "multi-computer", and facilities are provided for extremely low overhead communication and coordination among cooperating processors.

A secondary, but important objective of WM was to permit a wider variety of implementations than is normally considered in machine designs. To this end, WM was designed as a "family of architectures" parameterized by the size of the data manipulated by a family member. This approach permits members of the WM family to span the spectrum from 16-bit, integer only mini-computers through 64-bit supercomputers.

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 family of computer architectures 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". Members of the family differ from each other in the size (width) of their data manipulation operations, as well as in the type of operations supported -- members of the family span the spectrum from 16-bit, integer only microcomputers to vector supercomputers with 64-bit virtual addresses.

The notion of a "family of architectures" is, to our knowledge, a new one. The more familiar notion is that of a family of computers with a common architecture; such a family is a collection of implementations that execute the same instruction set. By way of contrast, members of the WM family implement conceptually similar, but not necessarily identical instruction semantics, and not all instructions are supported on all family members.

> Arithmetic operations are examples of the "similar but not identical" instructions: all members of the WM family support an "integer add" operation, but their semantics differ; on the 16-bit versions of the family this is a 16-bit add, while on 64-bit versions it is a 64-bit add.
>
> Load and store instructions are examples of ones not supported on all family members: on 32-bit versions of the machine, 8-, 16-, and 32-bit integer data types are supported in memory, and operations are provided to load these data types into the registers. On a 16-bit version of the family, only 8- and 16-bit integer data types are supported, and the operation to load a 32-bit integer is illegal.

The most important contributions of the WM family are related to their performance, which principly arise from three features:

Concurrent Operations	A primary objective of the WM designs was to allow many operations to occur concurrently. This has several aspects. First, the control, integer, floating point, and vector units are separated so that an instruction can be dispatched to each simultaneously. Second, many instructions may specify two operations, and the ALUs that implement these operations are pipelined so that two operations to be dispatched per cycle. Third, many instructions specify a large set of operations; this is obviously true of the vector instructions, but is also true of "streamed" load/store operations, discussed below. Finally, WM provides facilities that enhance its use as a node in a multiprocessor.
Streaming	A mechanism is provided for asynchronous loads and stores 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 facility applies to WM's scalar as well as its vector execution units, and has the effect of potentially executing many load/store operations concurrent with the execution of other instructions.
Fast Conditionals	A new approach to conditional jumps is taken that allows their execution to be completely overlapped with that of other instructions and the "prefetch" of subsequent instructions to begin well ahead of the need for them.

The net effect of these three features is a peak performance of many operations per cycle; the exact number depends upon parameters of the architecture and implementation, as will be described later. Other important architecture features that further enhance the performance and applicability of WM include:

Micro-concurrency	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 cache misses and a long memory access latency.

Large Register Set

In the largest members of the WM family, 96 register names are provided -- of which 27 are integer/logical "general" registers, 29 are floating-point registers, 29 are vector registers, and 11 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
- communicate with other processors in a large multicomputer

Each of these provides the possibility of significant reductions in operating system overheads. A primary objective of the philosophy was to support the use of WM as nodes in a large "multi-computer", where performance of applications depends critically upon such overheads.

Chapter 2. Logical Machine Structure

This chapter describes the basic framework of the WM family, 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.

As noted in the last Chapter, WM is a family of computer architectures. A member of the family in denoted by three parameters, and is denoted $WM_{\underline{i},\underline{f},\underline{v}}$. The parameters denote the size, and implicitly the existence, of the integer, floating, and vector data manipulation operations of the family member. The parameters are constrained such that:

$$\underline{i} \in \{16, 32, 64\}$$
$$\underline{f} \in \{0, 32, 64\}$$
$$\underline{v} \in \{0, 32, 64\}$$

Thus, $WM_{16,0,0}$ is a 16-bit, integer-only architecture -- such as one might use for micro-computer (controller) applications. $WM_{32,32,0}$ is an architecture with 32-bit integer and floating point operations, but no vector operations -- such as one might use for workstation applications. $WM_{64,64,64}$ is an architecture with 64-bit integer (and virtual addresses), floating point and vector operations -- such as one might use for supercomputer applications.

As noted in the last Chapter, (machine language) programs that utilize the features of more capable members of the WM family are invalid for less capable members of the family. In many cases, such as using a vector instruction on a machine without a vector unit, will be caught by the hardware (i.e., an exception will be raised). In other cases, such as a program that depends upon 32-bit addressing may appear to execute properly on less capable machines, such as ones with only a 16-bit integer unit, or may raise an apparently unrelated exception such as overflow. In general, programs should be compiled for the member of the family on which they are intended to execute.

To simplify the discussion in the following sections, we will often refer explicitly or implicitly to the "WM architecture", actually meaning a member of the architectural family, and generally a member with all of the integer, floating, and vector capabilities. The reader should be able to easily infer the implications for less capable members of the family.

2.1. Addresses & Data

2.1.1. Addresses

Addresses on the WM architecture are $\underline{i}$-bit signed values and memory is 8-bit byte addressed. Note that addresses are *signed*; valid addresses lie in the range -2^{31} .. $(2^{31}-1)$. 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).

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 an integer register may be set, tested, or selected in one instruction, and any bit may be cleared 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. $\underline{i}$-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 $\underline{i}$-bit signed integers is supported by individual operations. While, on appropriate family members, signed integers may be loaded and stored as doublewords, words, halfwords, or bytes, all integer arithmetic is performed on $\underline{i}$-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 — Arithmetic is performed using the $\underline{f}$-bit IEEE floating point standard.

2.1.4. Data Numbering

The architecture is "big-endian". 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.

2.1.5. Register Names

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, floating point, and vector registers are distinct, providing 96 total register names. As an aid in computation, register 31 in all three units are defined to be identically zero. Although it is possible to write to these registers, stores have no effect. In addition, four register names in the integer unit and two in the floating point and vector units are reserved for special meanings, as will be discussed in sections 2.3 and 2.4.

For purposes of this manual, the registers in the various execution units are given distinct nemonic names:

r0, ..., r31	refer to the integer/logical registers in the IEU,
f0, ..., f31	refer to the floating point registers in the FEU, and
v0, ..., v31	refer to the registers in the VEU.

In addition rZ, fZ, and vZ refer to register 31 (the "always zero" register) in the IEU, FEU, and VEU respectively. Capital "R" is used to denote the contents of a register field of an instruction, independent of the type of instruction.

2.1.6. Literals in Instructions

Certain instructions may specify unsigned, 5-bit literals as operands. These literals are the integers 1-32 and are encoded in the obvious way, except that 32 is encoded as zero. Since r31 is "always zero", in effect all the literals in 0..32 may be represented in these instructions.

2.2. The Execution Units

As noted earlier, a primary objective of the WM design was to support concurrency at several levels, including concurrent execution of several instructions. To support this, the instruction set has been partitioned so that the proper conceptual model of WM is one with three execution units under common control of the "instruction fetch unit", IFU.

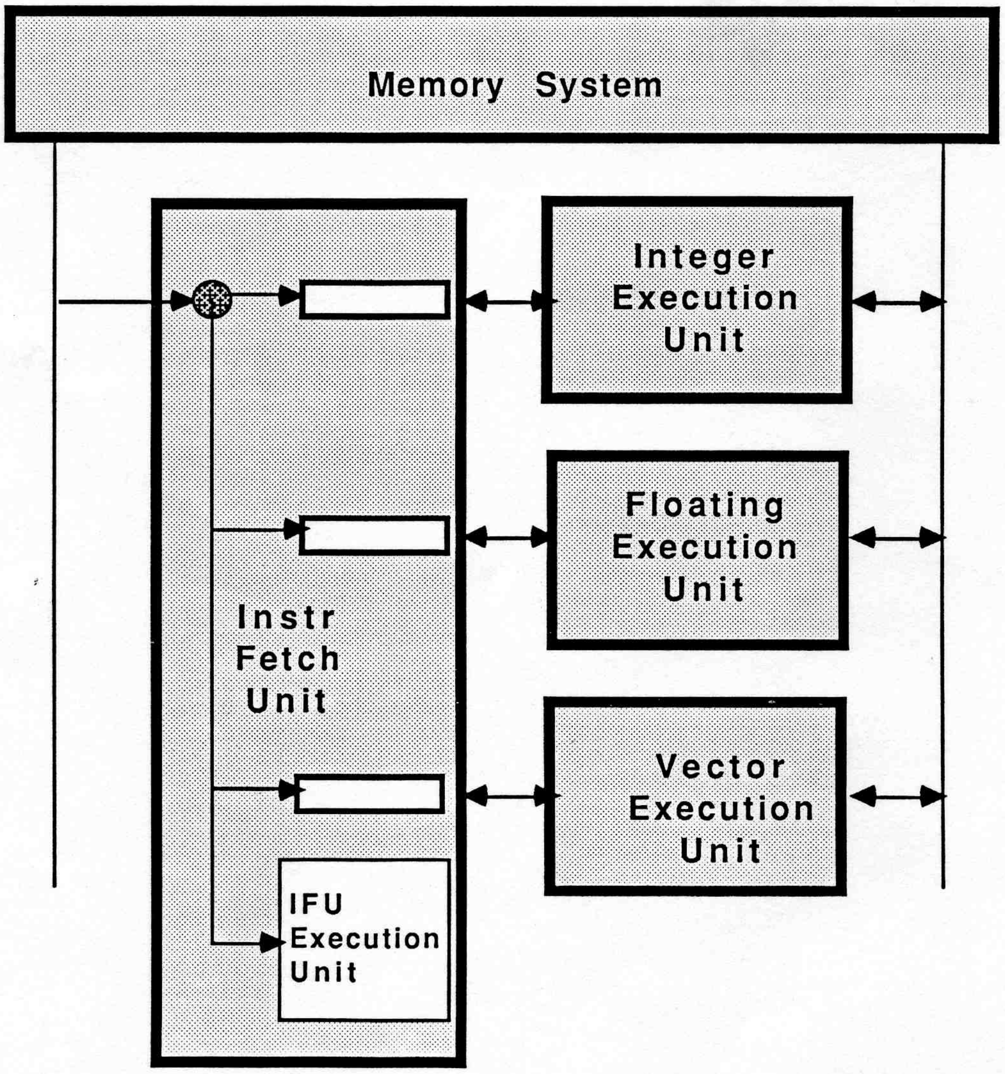


Figure 1: WM System Components

As shown in figure 1[1], at least conceptually, the IFU can be thought of as enqueuing instructions for execution by each of the other execution units in a set of FIFOs, as well as executing certain instructions itself (notably control instructions). The other execution units dequeue instructions and execute them as rapidly as their respective implementations permit. Thus, on any given cycle as many as four instructions may be dispatched.

Depending on the relative speed of the implementations of the various execution units, the actual execution of instructions may be "out of order" with respect to the original program text; as will be seen, however, that the semantics of the instruction set is such that the the semantics of the program are preserved under such reorderings.

2.2.1. The Scalar Execution Units

The scalar data manipulation instructions of WM are implemented by the Integer and Floating Point Execution Units; they specify 3 source operands, 2 operators, and a destination register, and evaluate an assignment of the form:

$$R0 := (R1 \ \underline{\mathbf{op1}} \ R2) \ \underline{\mathbf{op2}} \ R3$$

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 an unsigned literal; 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, other than zero, are not supported as source operands).

The integer and floating point 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. Thus one instruction (two operations) can be dispatched to each of the scalar execution units each cycle.

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.

[1]This figure and the several that follow it are intended to provide an intuitive, model implementation to explicate the semantics of the WM instruction set. Actual implementations may or, more likely, may not have a similar structure.

The pipelined structure of the WM scalar 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 *for the same execution unit.* 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 data dependencies are defined with respect to instructions for the *same* execution unit! Thus, there can be data dependencies that are not immediately obvious from a cursory examination of the source code. Consider an integer instruction followed by a long series of floating point instructions, followed by another integer instruction. The two integer instructions are adjacent with respect to execution by the IEU, and thus have a data dependency. As suggested by the figure in Section 2.2, the second integer instruction will be enqueued and executed immediately after the first; the textually intervening floating instructions have no impact on the execution of the integer instructions.

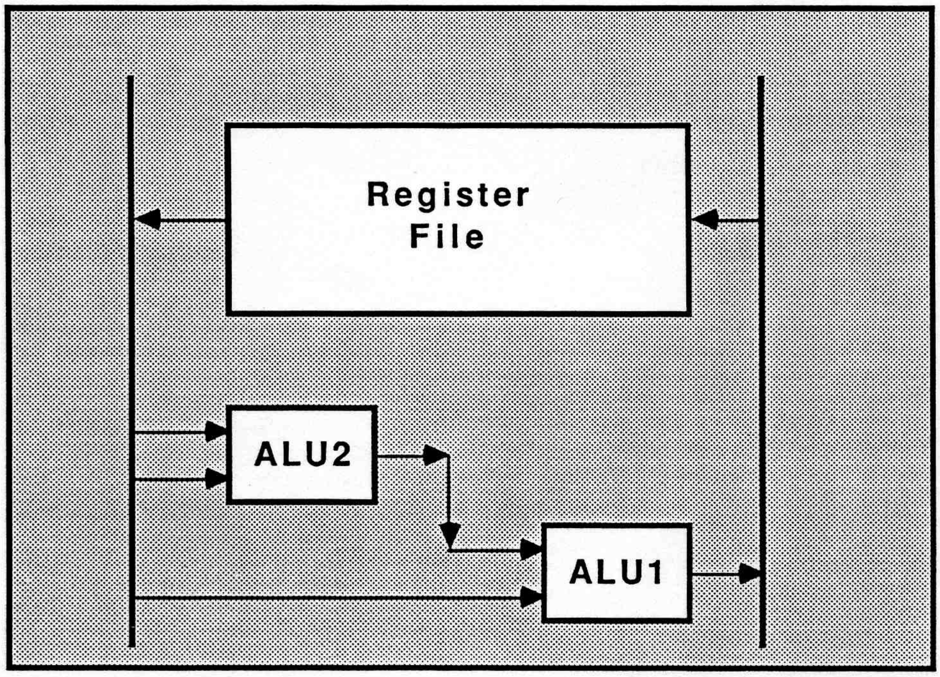


Figure 2: Scalar Execution Unit Structure

2.2.1. The Vector Execution Unit

The vector instructions of WM contain only one operation, but may specify a "mask" of boolean values that determine whether components of the result vector are affected by the operation. In general, the form of a vector instruction is

R0 := (R1 <u>op</u> R2) <u>if</u> R3

Each vector register contains N <u>v</u>-bit components, where N is a parameter of the implementation; each instruction performs the computation

<u>forall</u> k, 0≤k<N, $R0_k$:= **<u>if</u>** $R3_k \neq 0$ **<u>then</u>** ($R1_k$ **<u>op</u>** $R2_k$) **<u>else</u>** $R0_k$ **<u>fi</u>** conditional execution

At least conceptually all of these operations are performed simultaneously; an implementation may choose to perform them serially (as with a single pipelined ALU), but this is not visible to the program.

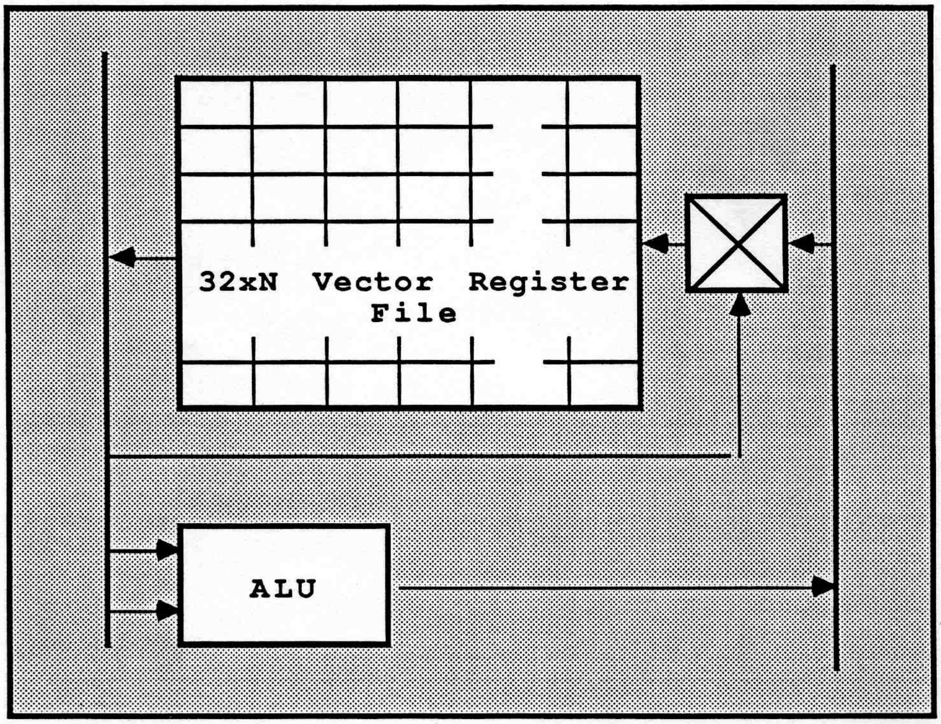


Figure 3: Vector Execution Unit

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 sets and the memory. Load and store instructions are operations on these FIFOs, and are executed by the IEU.

   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, and

- a store is a request to dequeue data from an output FIFO and store it to memory.

In both cases, only the memory address is required.

2. Data manipulation instructions (executed by the IEU, FEU, or VEU), which can only name registers (or literals) as operands, use "register 0" to name the input and output FIFOs.

   To dequeue data from an input FIFO, an instruction need only reference register 0 as a source operand. To enqueue data in an output FIFO, an instruction need only specify register 0 as the destination of a computation. Note that "register 0" is interpreted differently when used as a source and destination operand; as a source operand it refers to an input FIFO of the execution unit, and as a destination operand it refers to an output FIFO of the execution unit.

3. LOAD and STORE instructions are executed by the Integer Execution Unit, but may imply that the data to be loaded or stored is destined for either the Integer or Floating execution Unit FIFOs; memory operations for the Vector Execution Unit are handled by another mechanism (streaming) that is discussed later in this section.

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 enqueued in an input FIFO in the order of the LOAD instructions. 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 dequeued from 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 also may be accomplished in the other order -- the STORE instruction may be 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.

The LOAD and STORE instructions are executed by the integer/logical unit. The addresses they compute are directed to the FIFO in the proper execution unit; thus, for example, the address computed by a "load floating" instruction is directed to the FIFO in the FEU, and the data read as a result will be enqueued in that FIFO.

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

- 3 i-bit entries in the integer unit's input FIFOs
- 1 i-bit entry in the integer unit's output FIFO
- 3 f-bit entries in the floating unit's input FIFOs,
- 1 f-bit entry in the floating unit's output FIFO,
- 3 N-component blocks of v-bit entries in the vector unit's input FIFO, and
- 1 N-component block of v-bit entries in the vector unit's input FIFO.

These minimums are adequate to ensure that any 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 in each of the execution units is also a FIFO, albeit with somewhat different properties than that of register 0. Specifically, a value stored (computed) into the register 1 output FIFO is immediately enqueued in the register 1 input FIFO. As with register 0, items are dequeued simply by using register 1 as a source operand.

Thus, register 1 is a 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 interface 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 in each of the execution units may be used in stream mode, and indeed, this is the only mode for the VEU; each of these registers supports two modes of operation in the IEU and FEU, 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 in all execution units, 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(s) 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 vector load/store operations, but

- it is more general in the sense that it applies to the scalar execution units as well as the vector execution unit. Computations on streamed data are arbitrary software combinations of WM's "scalar" or "block" (vector) instructions. In particular, this allows streaming of computations involving recurrences (which cannot be vectorized) in the IEU and FEU.
- much simpler and more complete compiler algorithms can be used to detect streaming opportunities than to detect vectorization opportunities.

Streaming to/from the VEU is similar to that to/from the IEU or FEU, except that:

- data is moved in "blocks" of N entities, and
- there are boundary conditions when the stream count is not a multiple of N.

When the stream count, that is the number of entities to be streamed, is not a multiple of N, there will be some number $n \leq N$ of entities that do not participate in the "last" load/store operation performed by the streaming hardware. An implementation of WM must treat this odd-sized block as though it were full from the VEU's perspective, but only load/store the valid elements.

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 by the architecture.) 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 figure 4. 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).

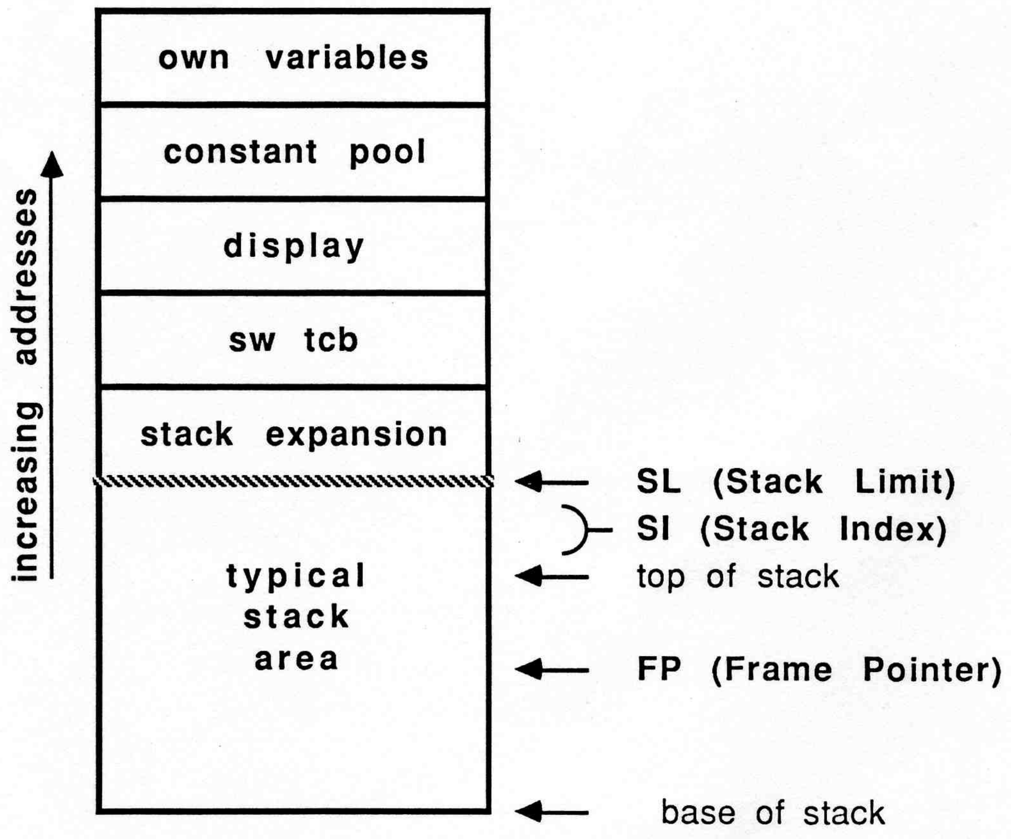


Figure 4: Suggested Stack Structure

- 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., condition codes); 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 six categories:

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

The following sections describe each set of instructions in some detail. It should be noted, however, that these different classes of instructions are executed by different components of WM. Specifically, refering back to Figure 1 in Section 2.2,

- the integer/logical and load/store instructions are executed by the IEU,
- the floating instructions are executed by the FEU,
- the vector instructions are executed by the VEU, and
- the the control and special instructions are executed under control of the IFU.

In general, an instruction of each type may be dispatched to its respective unit each cycle.

3.1. Integer Arithmetic & Logical Instructions

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

R0 := (R1 <u>op1</u> RL2) <u>op2</u> RL3

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

There are three source specifiers (R1, RL2, and RL3), two operation specifiers (op1 and op2), and a destination specifier, R0. R0, R1, RL2, and RL3 may specify one of 32 register names. Alternatively, RL2, and RL3 may specify a 5-bit literal[1]; the RL field determines this interpretation. The operands are either i-bit signed integers or a vector of i boolean values, depending on the operation specified. The instruction format is:

0 1	2 3	4 7	8 11	12 16	17 21	22 26	27 31
00	RL	OP1	OP2	RO	R1	RL2	RL3

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 relationals and have somewhat unusual properties. They produce *two* results: their left operand as a numeric result, and a boolean ("condition code") 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. In either case, if the boolean result is False, then the condition bit is set, but the

[1]As explained in section 2.1.6, the possible literals are integers in the range 1..32, encoded in the obvious way except that 32 is encoded as zero. Such literals are zero-extended to i bits.

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. In particular, note that because relationals produce their left operand as their result, instructions such as

```
r8 := (r8 < r10) + 1
```

have the effect of both testing the value in r8, and incrementing that value. This is the "increment and test" portion of the familiar "increment, test and jump" needed for loop control.

The semantics of the relationals also allow for efficient Ada 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 relationals 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 (Data Capacity Exceeded): 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. doubleprecision floating point). The type specified implicitly determines the execution unit involved.

(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 and 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. Thus, for example, a request to load a 64-bit floating point value is sent to the FIFO associated with register 0 in the FEU.

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:

0 1	2 3	4 7	8 11	12 16	17 21	22 26	27 31
01	RL	LSOP	op 1 op 2	RO	R1	RL2	RL3

As with the integer instructions, the RL field specifies 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:

op	operation	data size/type	exec unit	sign extension
L8i	load	8-bit integer	IEU	no
L8ix	load	8-bit integer	IEU	yes
L16i	load	16-bit integer	IEU	no
L16ix	load	16-bit integer	IEU	yes
L32i	load	32-bit integer	IEU	no
L32ix	load	32-bit integer	IEU	yes
L64i	load	64-bit integer	IEU	n/a
L32f	load	32-bit floating	FEU	n/a
L64f	load	64-bit floating	FEU	n/a
S8i	store	8-bit integer	IEU	n/a
S16i	store	16-bit integer	IEU	n/a
S32i	store	32-bit integer	IEU	n/a
S64i	store	64-bit integer	IEU	n/a
S32f	store	32-bit floating	FEU	n/a
S64f	store	64-bit floating	FEU	n/a

There are 15 LSOP operations; the other opcode is 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

Note, in particular, that there are no load/store instructions for the Vector Execution Unit. All memory-VEU transfers are done with 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 Full (Address Capacity Exceeded): 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 (Data Capacity Exceeded): as per integer instructions when specified as the destination register for the address calculation.
- Overflow/Underflow: an arithmetic operation overflowed or underflowed.
- Memory Protection Violation: 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 eight conditional jumps associated with the two condition FIFOs: "Jump True" and "Jump False" for each of the integer and floating conditions; each jump may predict whether the jump will be taken or not. 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 jump.

There are twelve conditional jumps associated with the streaming facility of the machine (see Section 3.6.1); these support jumps on the on "stream count not zero" for each of the input and output streams.

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

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:

0 1	2 3	4 11	12 31
11	01	OP	OFFSET

The opcode may specify one of the following operations:

Jump	the basic unconditional jump.
JumpITy	jumps only if the Integer condition bit is True; assume jump will be taken.
JumpIFy	jumps only if the Integer condition bit is False; assume jump will be taken.
JumpFTy	jumps only if the Floating condition bit is True; assume jump will be taken.
JumpFFy	jumps only if the Floating condition bit is False; assume jump will be taken.
JumpITn	jumps only if the Integer condition bit is True; assume jump will *not* be taken.
JumpIFn	jumps only if the Integer condition bit is False; assume jump will *not* be taken.
JumpFTn	jumps only if the Floating condition bit is True; assume jump will *not* be taken.
JumpFFn	jumps only if the Floating condition bit is False; assume jump will *not* be taken.
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".
JNI v0	"Jump on Stream Count Not Zero; Vector Input FIFO 0".
JNI v1	"Jump on Stream Count Not Zero; Vector Input FIFO 1".
JNO v0	"Jump on Stream Count Not Zero; Vector Output FIFO 0".
JNO v1	"Jump on Stream Count Not Zero; Vector Output FIFO 1".
Call	Store PC of the next instruction in r4; jump to specified address.
ECall	Entry Call; see below

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 3.6 for the format of special instructions). Nonetheless, we will discuss them here; they are JumpI (Jump Indirect), CallI (Call Indirect) and EReturn (Return from ECall).

JumpI	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 in order to branch more than 2^{20} 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 FIFO Empty: 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.

- Memory 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 <u>**op1**</u> R2) <u>**op2**</u> R3

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

0 1	2 3	4 7	8 11	12 16	17 21	22 26	27 31
11	00	OP1	OP2	RO	R1	R2	R3

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 minus 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 -- the "nop's"; this is due to the fact that in some implementations, the nop operation will execute in less time than its synthesized counterpart.

The last six operations are 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. Vector Instructions

As noted earlier, in general the vector instructions perform a single operation, but the item-by-item operation may be conditioned by a boolean vector contained in a third vector register.

R0 := (R1 op R2) If R3

In reality, there are two forms of each vector operation -- conditional or not; for simplicity, only the unconditional form of the vector operations are defined below. Vector instructions have the format:

0 1	2 3	4 11	12 16	17 21	22 26	27 31
11	01	OP	RO	R1	R2	R3

The Vector Execution Unit supports integer, logical and floating point operations on "blocks" of N v-bit items, where N is an implementation defined parameter. Conceptually, all N component operations are performed simultaneously.

3.5.1 Logical Vector Operations

The logical vector operations are:

operation	meaning
and	bit-wise and
or	bit-wise or
eqv	bit-wise equivalence

3.5.2 Integer Vector Operations

The integer vector operations are:

operation	meaning
iadd	integer add
isub	integer subtract

imul	integer multiply
idiv	integer divide
iasl	integer arithmetic shift left
ieql	integer "equality"
ineq	integer "inequality"
igtr	integer "greater than"
igeq	integer "greater than or equal to"
ilss	integer "less than"
ileq	integer "less than or equal to"

The vector relational operations are different from their counterparts for the IEU and FEU; they do not produce a condition code. Rather they produce a vector of boolean values in the specified destination register -- such a vector may, for example, be used to control a subsequent conditional vector operation. Thus, the loop:

```
      DO 10,  i=1, M
  10  IF (A(i) .EQ. 0) B(i) = B(i) + 1
```

may be coded as follows (see Section 3.6.1.b for a definition of streaming for the VEU and Section 3.6.3 for a definition of transfers from the IEU to the VEU):

```
      VSin32i     v0, r5, r4, 4 -- assuming r5 is base address of A, r4 holds M
      VSin32i     v1, r6, r4, 4 -- assuming r6 is base address of B
      VSout32i    v1, r6, r4, 4 -- assuming r6 is base address of B
      TIV         v2 := 0       -- set all elements of v2 to integer zero
      TIV         v3 := 1       -- set all elements of v3 to integer one
   L: v4 := (v0 ieql v2)        -- determine whether A(i)'s are zero
      v1 := (v1 iadd v3) if v4  -- increment appropriate B(i)'s
      JNI v0, L                 -- loop if not done
```

3.5.3 Floating Point Vector Operations

The floating point vector operations are:

operation	meaning
fadd	floating add
fsub	floating subtract
fmul	floating multiply
fdiv	floating divide
feql	floating "equality"
fneq	floating "inequality"
fgtr	floating "greater than"
fgeq	floating "greater than or equal to"
flss	floating "less than"
fleq	floating "less than or equal to"

The floating point relational operations are similar to the integer ones in the VEU; they produce a boolean vector in the destination register.

3.6.1 Other Vector Operations

There are a few other vector operations, namely:

operation	meaning
vCVTIF	vector convert integer to floating
vCVTFI	vector convert floating to integer

3.6. 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:

0 1	2 3	4 11	12 16	17 21	22 26	27 31
10	RL	OP	RO	R1	RL2	RL3

Responsibility for execution of the special instructions resides in the Instruction Fetch Unit, IFU; in reality, however, one or more of the other execution units may be involved. When more than one execution unit is involved, the IFU must ensure that the proper synchronization of the other units occurs so that sequential semantics are enforced[1]. For example, type conversion between integer values and floating point values in the IEU and FEU registers respectively, are special instructions. One such instruction is:

CVTIF R0 := R1

This instruction converts the integer value in the (integer) register specified by R1 to a floating point value in the (floating) register specified by R0. This instruction must be executed as though all preceding integer and floating point instructions had completed -- and specifically any instructions that depend upon, or set, the proper values in the registers specified by R0 and R1 must have completed.

3.6.1. 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 for each element. 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)

[1] In general this may imply waiting for all previous instructions to complete and inhibiting all subsequent instructions until the special instruction has completed. In practice, however, many relatively simple optimizations can be detected by an implementation.

apart from each other. Stream instructions are used to read/write such vectors from/to FIFOs. Streaming is conceptually identical for the IEU, FEU, and VEU, but the implications with respect to the VEU are slightly different and will be discussed separately.

3.6.1.a Streaming to and from the IEU and FEU

There are 15 instructions that initiate streaming operations to the IEU and FEU. These are analogous to the 15 types of loads and stores. They specify data as integer or floating point and size of the data items. The operands of streaming operations 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:

op	operation	data type	exec unit	sign extension
Sin8i	load	8-bit integer	IEU	no
Sin8ix	load	8-bit integer	IEU	yes
Sin16i	load	16-bit integer	IEU	no
Sin16ix	load	16-bit integer	IEU	yes
Sin32i	load	32-bit integer	IEU	no
Sin32ix	load	32-bit integer	IEU	yes
Sin64i	load	64-bit integer	IEU	n/a
Sin32f	load	32-bit floating	FEU	n/a
Sin64f	load	64-bit floating	FEU	n/a
Sout8i	store	8-bit integer	IEU	n/a
Sout16i	store	16-bit integer	IEU	n/a
Sout32i	store	32-bit integer	IEU	n/a
Sout64i	store	64-bit integer	IEU	n/a
Sout32f	store	32-bit floating	FEU	n/a
Sout64f	store	64-bit floating	FEU	n/a

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

[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.

An example of a valid assembly instruction is:

```
Sin32i    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 for the integer and floating point units.

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.

3.6.1.b Streaming to and from the VEU

As noted above, streaming to and from the VEU is conceptually similar to streaming to and from the IEU and FEU; however, it differs in a few details:

- since there are no LOAD or STORE instructions for the VEU, there is only one "mode" for the VEU FIFOs. Note specifically that v1 cannot be used as a parameter bypass.
- since the VEU supports integer, logical, and floating operations, appropriate streaming operations are provided to do the proper form of operand expansion or contraction.

- since the operations of the VEU may be controlled by 1-bit (boolean) control vectors, the ability to stream such vectors is provided.

Vector streaming occurs in "blocks" of N items, where N is the implementation-defined number of items per vector register. In the event that the stream count is not a multiple of N the "last block" of items read or written will contain less than N items. On input the block will be padded with suitable values, and any addressing violations resulting from attempting to access these invalid values will be suppressed. Similarly, on output, only the valid items will be written to memory, and no inappropriate addressing violations will be raised. The implication of these rules is that the program does not need to worry about the "boundary conditions"; examples such as that shown in Section 3.5.2 will work properly even if the vector length is not a multiple of N.

The VEU streaming operations are:

op	operation	data type	exec unit	sign extension
VSin1b	load	1-bit boolean	VEU	no
VSin8i	load	8-bit integer	VEU	no
VSin8ix	load	8-bit integer	VEU	yes
VSin16i	load	16-bit integer	VEU	no
VSin16ix	load	16-bit integer	VEU	yes
VSin32i	load	32-bit integer	VEU	no
VSin32ix	load	32-bit integer	VEU	yes
VSin64i	load	64-bit integer	VEU	n/a
VSin32f	load	32-bit floating	VEU	n/a
VSin64f	load	64-bit floating	VEU	n/a
VSout1b	store	1-bit boolean	VEU	n/a
VSout8i	store	8-bit integer	VEU	n/a
VSout16i	store	16-bit integer	VEU	n/a
VSout32i	store	32-bit integer	VEU	n/a
VSout64i	store	64-bit integer	VEU	n/a
VSout32f	store	32-bit floating	VEU	n/a
VSout64f	store	64-bit floating	VEU	n/a

StopAll	Stop all Streaming operations (same instruction as discussed above)
StopVI	Stop Vector Input Streaming operations on FIFO specified by R0
StopVO	Stop Vector Output Streaming operations on FIFO specified by R0

The following program, to copy a vector from one location to another, increments register r9 each time through the loop. After the loop terminates, the value r8/r9 will give the value of the implemenation determined block size N if r8 is a multiple of N.

```
-- r8 = count of words to copy
-- r6(r7) = address of source(sink) location
        r9 := r31
        VSin32i v0, r6, r8, 4
        VSout32i     v0, r7, r8, 4
L:      v0 := v0
        r9 := r9 + 1
        JNIv0    L
```

3.6.2. 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
VLoadM	R1, L2, L3
StoreM	R1, L2, L3
FStoreM	R1, L2, L3
VStoreM	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. Separate instructions specify IEU, FEU and VEU 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
LoadFifoVI	R0, R1
LoadFifoVO	R0, R1
StoreFifoVI	R0, R1
StoreFifoVO	R0, R1

These instructions load and store the specified (R0) FIFO state from/to the address specified in R1. The amount and format of this information is implementation dependent. Again, separate instructions specify IEU, FEU, and VEU input and output FIFOs.

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.

3.6.3. Type Conversion

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

```
CVTIF  R0 := RL3
CVTFI  R0 := RL3
```

There interpretation is the obvious one, ConVerT from Integer,or Floating, 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 scalar execution units:

```
TIF    R0 := RL3
TFI    R0 := RL3
```

These have the obvious "Transfer from Integer to Floating", and "Transfer from Floating to Integer" interpretations; these are "bit copy" instructions, no data conversion is performed except as necessary to expand/contract their representation[1].

Two instructions are also provided to move data from the IEU or FEU to the VEU:

```
TIV    R0 := RL3
TIVx   R0 := RL3
TFV    R0 := RL3
```

These instructions transfer N copies of the integer or floating point register specified in RL3; all items in the destination vector register receive identical values. Thus, for example,

```
TIV    v5 := 1
```

will initialize all items in vector register v5 to (integer) one. TIV zero extends as necessary while TIVx performs sign extension.

3.6.4. Constraint Checking

There are two instructions for checking arithmetic constraints in the IEU and FEU:

```
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 Assert Fault 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.
```

3.6.5. Field Manipulation

Fields can be moved within a word in the IEU 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 i perform the same function as a shift by i, 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 for the IEU:

```
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 in the R1 value, starting from the left. It stores into R0 the bit number of the bit found. If all the bits in the word are the same, the value 0 is stored. Taking this instruction as a function and applying it to some short examples, we get:

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

3.6.6. Constant Generation

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

```
LLH    R0 := <16 bit constant>       -- Load Lower Half
SLL    R0 := <16 bit constant>       -- Shift and Load Lower
```

LLH assigns a 16-bit constant to the destination register. SLL logically shifts the destination register left by 16 bits and then assigns the 16-bit constant from the instruction to the low order 16-bits of the destination register. In both cases the 16 bit constants are formed by the concatenating the low order RL bit (bit 11) with the R1, RL2 and RL3 fields.

Constants of 32-bits may be formed by a LLH, SLL pair; 64-bit constants may be formed by a LLH followed by three SLL's. If necessary these constants can be moved to the FEU or VEU with "transfer" instructions (see Section 3.6.3).

3.6.7. 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. They are semantically identical to a "conditional jump to the next instruction", but may be optimized by an implementation.

3.6.8. PCW Access

The instructions

```
ReadPCW     R0
WritePCW    R1
```

allow the programmer to read and write the Program Control Word (see Chapter 4). The register specified by these instruction is taken from the IEU register set.

3.6.9. Pipeline Synchronization

The instruction

```
SYNCH
```

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

Typically the use of SYNCH is not required for the correct execution of programs; there are, however, circumstances where it is useful. It can be used, for example, at "break points" during debugging to ensure that the programmer is observing a program state that corresponds to a defined point in the source program. It can also be used at statement boundaries in Ada to ensure proper ordering of exceptions when the compiler cannot prove that those exceptions will not be raised.

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 and allowed to perform direct i/o operations on that 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 a 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" to that page; 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 32-bit 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 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, floating point and vector 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 (enabled == 1):
1	Attempted Stack Limit Modification
2	Stack Index Negative
3	Assert Fault
4	Integer Divide By Zero
5	Floating Divide By Zero
6	Integer Arithmetic Overflow
7	Integer Arithmetic Underflow
8	Floating Arithmetic Overflow
9	Floating Arithmetic Underflow
10	Cycle Counter Overflow
11	Raise Address
12	Raise Call
13	Raise Jump

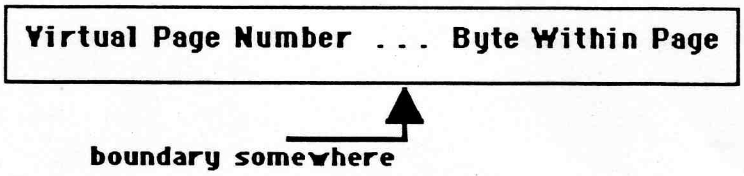
The Program Status Word collects a number of fields that reflect status of the task, such as the run/halt bit, the interrupt enabling bit, the priority and the condition FIFOs. For example, the PSW could include:

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. 3.6.2) 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.

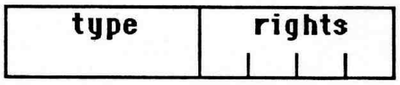


Assume K bits of virtual page number and i-K bits that specify the byte within the page. Associated with every virtual page number is a protection table and map table entry, as described below.

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 the physical address of 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.

[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 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.

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

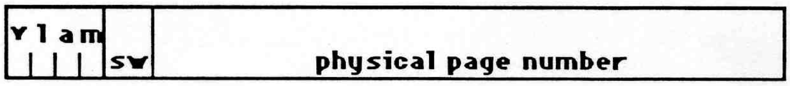
Accesses to Device pages may be only reads and writes, and the corresponding rights bits are R and W.

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 the physical address of 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.

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:

Location	Exception
0	(reserved)
8	Load While Input Streaming
16	Store While Output Streaming
24	Input FIFO Full
32	Input FIFO Empty
40	Output FIFO Full (Data Capacity Exceeded)
48	Output FIFO Full (Address Capacity Exceeded)
56	Condition FIFO Full
64	Condition FIFO Empty
72	(reserved)
80	Undefined Instruction
88	Memory Protection Violation
96	Attempted Stack Limit Modification
104	Stack Index Negative
112	Jump On Stream Count while not streaming
120	Double Stream
128	(reserved)
136	Assert Fault
144	Integer Divide By Zero
152	Floating Divide By Zero
160	Integer Arithmetic Overflow
168	Integer Arithmetic Underflow
176	Floating Arithmetic Overflow
184	Floating Arithmetic Underflow
192	(reserved)
200	Cycle Counter Overflow
208	Raise Address
216	Raise Call
224	Raise Jump
232	(reserved)
240	Page Fault
248	(reserved)

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

Machine implementation determines initialization.

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 physical 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. Also, some recommendations about program structure to improve performance are given.

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 or consume instructions. The number of instructions containing relationals preceding the associated conditional jump or consume instruction 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.

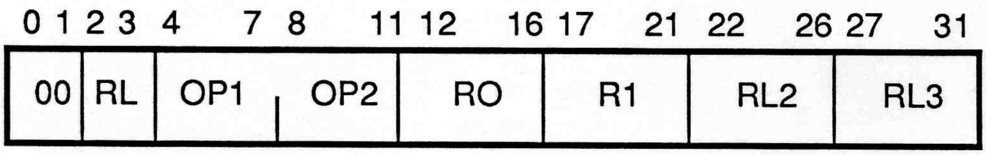
Recommendations

- 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.

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

0 1	2 3	4 7	8 11	12 16	17 21	22 26	27 31
01	RL	LSOP	op 1 op 2	RO	R1	RL2	RL3

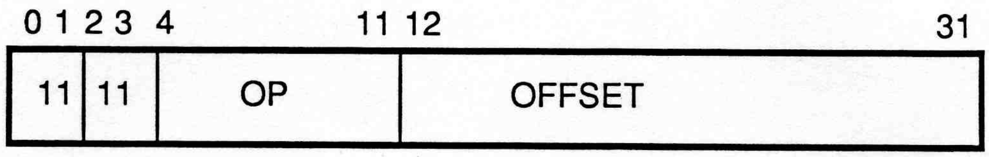
L8i	0000:	Load Byte
L16i	0001:	Load Halfword
L32i	0010:	Load Word
L64i	0011:	Load Doubleword
L8ix	0100:	Load Byte sign-eXtended
L16ix	0101:	Load Halfword sign-eXtended
L32ix	0110:	Load Word sign-eXtended
S8i	1000:	Store Byte
S16i	1001:	Store Halfword
S32i	1010:	Store Word
S64i	1011:	Store Doubleword
L32f	1100:	Load Floating
L64f	1101:	Load Doubleword Floating
S32f	1110:	Store Floating
S64f	1111:	Store Doubleword Floating

B.3. Floating Point Format and Encodings

0 1	2 3	4 7	8 11	12 16	17 21	22 26	27 31
11	00	OP1	OP2	RO	R1	R2	R3

+	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
	0110:	reserved
	0101:	reserved
=	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	0000 0000:	unconditional Jump

-- jumps that predict the branch <u>will</u> be taken

JumpITy	0000 1101:	Jump if Integer condition bit is True
JumpIFy	0000 1100	Jump if Integer condition bit is False
JumpFTy	0000 1111:	Jump if Floating condition bit is True
JumpFFy	0000 1110:	Jump if Floating condition bit is False

-- jumps that predict the branch will <u>not</u> be taken

JumpITn	0000 1001:	Jump if Integer condition bit is True
JumpIFn	0000 1000	Jump if Integer condition bit is False
JumpFTn	0000 1011:	Jump if Floating condition bit is True
JumpFFn	0000 1010:	Jump if Floating condition bit is False

-- jumps on stream count non-zero

JNI r0	0001 0000:	Jump on stream count Not zero; Input FIFO r0
JNI r1	0001 0001:	Jump on stream count Not zero; Input FIFO r1
JNO r0	0001 0010:	Jump on stream count Not zero; Output FIFO r0
JNO r1	0001 0011:	Jump on stream count Not zero; Output FIFO r1
JNI f0	0001 0100:	Jump on stream count Not zero; Input FIFO f0
JNI f1	0001 0101:	Jump on stream count Not zero; Input FIFO f1
JNO f0	0001 0110:	Jump on stream count Not zero; Output FIFO f0
JNO f1	0001 0111:	Jump on stream count Not zero; Output FIFO f1
JNI v0	0001 1000:	Jump on stream count Not zero; Input FIFO v0
JNI v1	0001 1001:	Jump on stream count Not zero; Input FIFO v1
JNO v0	0001 1010:	Jump on stream count Not zero; Output FIFO v0
JNO v1	0001 1011:	Jump on stream count Not zero; Output FIFO v1

-- special jump instructions

Call	0010 0000:	subroutine Call
ECall	0010 0001:	Entry Call

B.5. Vector Operation Encodings

0 1	2 3	4 11	12 16	17 21	22 26	27 31
11	01	OP	RO	R1	R2	R3

Note: The suffix 'C' on the following operation codes denotes the conditional form of the operation.

	0000	0001	0010	0011	0100	0101
0000	isub	isubC	fsub	fsubC	VCVTIF	VCVTIFC
0001	imul	imulC	fmul	fmulC	VCVTFI	VCVTFIC
0010	iadd	iaddC	fadd	faddC		
0011	iasl	iaslC				
0100						
0101	ieqv	ieqvC				
0110	ior	iorC				
0111	iand	iandC				
1000	idiv	idivC	fdiv	fdivC		
1001	igeq	igeqC	fgeq	fgeqC		
1010	ieql	ieqlC	feql	feqlC		
1011	ilss	ilssC	flss	flssC		
1100						
1101	ileq	ileqC	fleq	fleqC		
1110	ineq	ineqC	fneq	fnegC		
1111	igtr	igtrC	fgtr	fgtrC		

B.6. Special Format and Operation Encodings

0 1	2 3	4 11	12 16	17 21	22 26	27 31
10	RL	OP	RO	R1	RL2	RL3

	0000	0001	0010	0011	0100	0101
0000	Sin8i	VSin8i	StopII	LoadFifoII	CVTIF	JumpI
0001	Sin16i	VSin16i	StopIO	LoadFifoIO	CVTFI	CallI
0010	Sin32i	VSin32i	StopFI	LoadFifoFI	TFI	EReturn
0010	Sin64i	VSin64i	StopFO	LoadFifoFO	TFV	
0100	Sin8x	VSin8x	StopVI	LoadFifoVI	TIF	SYNCH
0101	Sin16x	VSin8x	StopVO	LoadFifoVO	TIV	ConsumeI
0110	Sin32x	VSin32x			TIVx	ConsumeF
0111		VSin1b	StopAll			
1000	Sout8i	VSout8i	LoadM	StoreFifoII	Assert	LoadCTX
1001	Sout16i	VSout16i	StoreM	StoreFifoIO	FAssert	StoreCTX
1010	Sout32i	VSout32i	FLoadM	StoreFifoFI	LLH	SwapCTX
1011	Sout64i	VSout64i	FStoreM	StoreFifoFO	SSL	SwapLT
1100	Sin32f	VSin32f	VLoadM	StoreFifoVI	FLDMOV	ReadPCW
1101	Sin64f	VSin64f	VStoreM	StoreFifoVO	FLDMOVX	WritePCW
1110	Sout32f	VSout32f			FFB	
1111	Sout64f	VSout64f				

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) the register names "r0", "r1", ..., "r31", "f0", ..., "f31", "v0", ... , "v31" , rZ, fZ, and vZ, 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

Integer/logical, floating point, and vector instructions are written in the form:

```
int    R0 := (R1 op1 RL2) op2 RL3,
flt    R0 := (R1 op1 R2) op2 R3, and
vec    R0 := (R1 op R2) if R3
```

respectively. ":=" is always the assignment symbol and the parentheses are required. Various abbreviations are allowed as defined below.

(1) Since the type of an instruction may be inferred from the register names used, the prefix "int", "flt", and "vec" may be dropped.

(2) Right-hand-side expressions may be abbreviated, and appropriate NOPs[1] will be inserted by the assembler; for example

```
int R0 := (RL2) op2 RL3       -- op1 defaults to a nop
int R0 := (R1 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
```

(3) Simple expressions involving only constants and the machine's integer/logical operations 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.

(4) 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)
```

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

and achieve the same effect as

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

(5) The literal zero, "0", may be used instead of the name of the "always zero" register (r31 or rZ, etc.).

(6) The left-hand-side of an assignment, i.e., "R0 :=", may be omitted; in such cases the assembler will insert an assignment to the proper "always zero" register. Thus, for example, a relational whose only intended effect is to generate a condition code may be written as

```
        int        (R1 relop RL2)
```

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 L32i.

Specifically, all of the abbreviations listed above are valid. Thus, the following are valid load/store instructions:

```
L8ix   r31 :=      (r7 asl 2) + r12
S64f   r4 :=  (r0 + r1) - 14
L32i          (r5 asl 2) + r3
L32f          (r6)
```

Jumps and calls also begin by specifying an instruction code and continue with a target specification. The branch-prediction suffix may be deleted; in such cases the assembler will assume that the branch will be taken. 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
JumpIFn  case+7
JumpFT   somewhere
Call     DiskHandler
jumpi    r21
```

The special instructions start with instruction code, 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. Rationale Behind Chosen Operation Sets

D.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 integer/logical 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 one exception was made following thoughtful justification (given below).

The chosen 16 integer/logical 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 a single instruction since A XOR B is NOT(A EQV B), or (A EQV B) EQV 0. 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 commuted 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'.

D.2. Floating Point and Vector Instructions

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

The vector operations are also basic arithmetic and relational operations. They have been defined as "block" operations; that is, conceptually all elements are evaluated simultaneously and independently. This permits an implementation to use one or more pipelines. It also permits "chaining" of operations.

The streaming of operations to/from the vector unit, and specifically the handling of vectors whose size is not a multiple of N, provides automatic "strip mining" of vector loops.

D.3. Load/Store Instructions

Data is manipulated in WM's registers as 16-, 32- or 64-bit 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.

D.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. Branch prediction was included to assist prefetching the target instruction.

The JumpI, 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^{22}-1 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.

D.5. Special Instructions

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

D.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 "start streaming" instructions mirror the 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.

D.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

D.5.3. Type Conversion Instructions

The rationale for these instructions is the obvious one.

D.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.

D.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.

D.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.

D.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-

[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.

specific synchronization and/or communication mechanisms. For a critical real-time application, for example, one could implement an entire message system in hardware.

Appendix E. Constructing WM Software Suggestions and Rationale

E.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
v 0	input vector FIFO; always assumed empty at calls
v 1	input vector FIFO; always assumed empty at calls
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.
v 3 1	identically zero; writes to this register have no effect.

This leaves r7-r30, f2-f30, and v2-v30 available for any use the compiler wants to make of them. Note that with this many registers, it probably makes sense to have a software convention that certain 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.

E.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.

E.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.

E.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.

E.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.

E.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 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 or constructed using LLH and SLH.

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

- the large, 20-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".