

**The WM Computer Architectures:  
Military Standard Manual**

Anita K. Jones  
Rohit Wad

Computer Science Report No. TR-90-19  
August 1990

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

# **Abstract**

---

This report is a military standard definition of the instruction set of the WM family of computer architectures. The WM instruction set architecture supports microconcurrency at the instruction level; i.e. it facilitates the execution of several scalar instructions concurrently. Also, WM supports vector processing; that is, it has single instructions that apply the same operation to a collection of data items. Another interesting feature of the WM architectures is streaming -- a mechanism for asynchronous loads and stores of "vector-like" data, that is, data with a known displacement between successive items. 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. The report has been patterned after the 1750 military standard manual.

## Table of Contents

<b>Document Derivation.....</b>	<b>1</b>
<b>1. Scope and Purpose.....</b>	<b>2</b>
1.1 Scope.....	2
1.2 Purpose.....	2
1.3 Applicability.....	2
1.4 Benefits.....	2
<b>2. Referenced Documents.....</b>	<b>2</b>
<b>3. Definitions.....</b>	<b>2</b>
3.1 Address.....	2
3.2 Alignment.....	3
3.3 Arithmetic logic unit (ALU).....	3
3.4 Bit.....	3
3.5 Byte.....	3
3.6 Concurrent operations.....	3
3.7 Condition code .....	3
3.8 Deadlock.....	3
3.9 Device .....	3
3.10 Domain.....	3
3.11 Doubleword.....	3
3.12 Entry.....	3
3.13 First-in-first-out queue (FIFO).....	4
3.14 Floating execution unit (FEU) .....	4
3.15 Floating point register.....	4
3.16 General purpose register.....	4
3.17 Halfword.....	4
3.18 Handler task .....	4
3.19 Input/output (I/O).....	4
3.20 Instruction .....	4
3.21 Integer execution unit (IEU) .....	4
3.22 Instruction fetch unit (IFU).....	4
3.23 Instruction set architecture (ISA).....	4
3.24 Interrupt.....	5
3.25 Load prefetch.....	5
3.26 Memory .....	5
3.27 Micro-concurrency .....	5
3.28 Multi-computer.....	5
3.29 Normal mode.....	5
3.30 Operation code (OPCODE).....	5
3.31 Prefetch.....	5
3.32 Program counter (PC).....	5
3.33 Register.....	5
3.34 Reserved.....	5
3.35 Right.....	5
3.36 Stack.....	5
3.37 Stream.....	5
3.38 Stream mode.....	6
3.39 Streaming.....	6
3.40 Task .....	6

3.41 Typed protection.....	6
3.42 Vector execution unit (VEU).....	6
3.43 Vector register.....	6
3.44 Word.....	6
3.45 Zero register.....	6
<b>4. General Requirements .....</b>	<b>7</b>
4.1 Function units.....	7
4.1.1 Scalar execution units.....	8
4.1.1.1 Data dependency rule .....	9
4.1.2 Vector execution unit.....	9
4.1.3 Instruction fetch unit.....	10
4.1.4 Parameter bypass .....	11
4.1.5 Streaming.....	12
4.1.5.1 Streaming to and from the IEU and FEU.....	12
4.1.5.2 Streaming to and from the VEU.....	13
4.1.6 Special instructions and synchronization.....	13
4.1.7 Deadlock.....	14
4.2 Data formats .....	15
4.2.1 Data alignment.....	15
4.2.2 Data sizes.....	15
4.2.3 Data Types .....	15
4.2.3.1 Boolean values.....	15
4.2.3.2 Signed integer values.....	16
4.2.3.3 Floating point values.....	16
4.3 Instruction formats.....	16
4.3.1 Literals in instructions.....	16
4.3.2 Instruction format notation.....	16
4.3.3 Integer format instructions.....	16
4.3.4 LOAD/STORE format instructions.....	16
4.3.5 Floating point format instructions .....	17
4.3.6 Control format instructions.....	17
4.3.7 Vector format instructions.....	18
4.3.8 Special format instructions.....	18
4.4 Registers and support features .....	18
4.4.1 General registers .....	18
4.4.2 Special registers.....	18
4.4.3 Stack.....	18
4.5 Memory.....	19
4.5.1 Memory reads & writes .....	19
4.6 Operating system support .....	20
4.6.1 Task state.....	20
4.6.2 Protection.....	22
4.6.3 Address mapping.....	23
4.6.4 Initialization of the machine.....	24
4.7 Devices.....	24
4.8 Input/output.....	25
4.9 Traps (exceptions) and interrupts.....	25
4.9.1 Interrupts .....	25
4.9.2 Traps .....	26

4.9.3 Exceptions .....	27
4.9.3.1 Integer exceptions.....	27
4.9.3.2 Load/Store exceptions.....	27
4.9.3.3 Control exceptions .....	27
4.9.3.4 Floating point exceptions.....	28
<b>5. Detailed Requirements.....</b>	<b>29</b>
5.1 Instruction set notation.....	29
5.1.1 Registers.....	29
5.1.1.1 General registers.....	29
5.1.1.1.1 IEU registers .....	29
5.1.1.1.2 FEU registers.....	30
5.1.1.1.3 VEU registers.....	30
5.1.1.2 Implementation dependent registers .....	30
5.1.1.3 Special registers.....	31
5.1.2 Symbols.....	31
5.1.2.1 " $\leftarrow$ ".....	31
5.1.2.1.1 Symbols to the left of $\leftarrow$ .....	31
5.1.2.1.2 Symbols to the right of $\leftarrow$ .....	31
5.1.2.2 " $\leftarrow$ " The assignment operator.....	31
5.1.2.2.1 Symbols to the left of $\leftarrow$ .....	31
5.1.2.2.2 Symbols to the right of $\leftarrow$ .....	32
5.1.2.3 ":" The bit selection operator .....	33
5.1.2.4 "::" The operator-argument operator.....	33
5.1.2.5 "lsl" The arithmetic shift left operator.....	33
5.1.2.6 "asr" The arithmetic shift right operator.....	33
5.1.2.7 "&&" The bitwise and operator.....	33
5.1.2.9 "  " The bitwise or operator.....	33
5.1.2.10 "or" The logical or operator.....	33
5.1.2.11 "EQV" The bitwise equivalence operator .....	33
5.1.2.12 "+, -, -, *, /, /" .....	33
5.1.2.13 "=", <>, <, <=, >=, >" .....	33
5.1.2.14 " $\phi$ ".....	33
5.1.3 Functions.....	34
5.1.4 Operations .....	34
5.1.5 Miscellaneous values.....	35
5.2 Mnemonic conventions.....	35
5.3 Execution semantics .....	35
5.4 Integer Arithmetic and Logical Instructions.....	36
5.4.1 op = +.....	37
5.4.2 op = asl.....	38
5.4.3 op = <.....	39
5.4.4 op = - .....	40
5.4.5 op = -' .....	41
5.4.6 op = and.....	42
5.4.7 op = or.....	43
5.4.8 op = eqv .....	44
5.4.9 op = *.....	45
5.4.10 op = / .....	46
5.4.11 op = /' .....	47

5.4.12 op = =.....	48
5.4.13 op = <>.....	49
5.4.14 op = <=.....	50
5.4.15 op = >=.....	51
5.4.16 op = >.....	52
5.5 Floating Point Instructions.....	53
5.5.1 op = <.....	54
5.5.2 op = +.....	55
5.5.3 op = -.....	56
5.5.4 op = -'.....	57
5.5.5 op = *.....	58
5.5.6 op = /.....	59
5.5.7 op = /'.....	60
5.5.8 op = nop.....	61
5.5.9 op = nop'.....	62
5.5.10 op = =.....	63
5.5.11 op = <>.....	64
5.5.12 op = <=.....	65
5.5.13 op = >.....	66
5.5.14 op = >=.....	67
5.6 Vector Instructions: Integer, Logical and Floating Point.....	68
5.6.1 op = iadd.....	69
5.6.2 op = isub.....	70
5.6.3 op = imul.....	71
5.6.4 op = idiv.....	72
5.6.5 op = iasl.....	73
5.6.6 op = ieql.....	74
5.6.7 op = ineq.....	75
5.6.8 op = igtr.....	76
5.6.9 op = igeq.....	77
5.6.10 op = ilss.....	78
5.6.11 op = ileq.....	79
5.6.12 op = iaddC.....	80
5.6.13 op = iandC.....	81
5.6.14 op = iorC.....	82
5.6.15 op = ieqvC.....	83
5.6.16 op = isubC.....	84
5.6.17 op = imulC.....	85
5.6.18 op = idivC.....	86
5.6.19 op = iaslC.....	87
5.6.20 op = ieqlC.....	88
5.6.21 op = ineqC.....	89
5.6.22 op = igtrC.....	90
5.6.23 op = igeqC.....	91
5.6.24 op = ilssC.....	92
5.6.25 op = ileqC.....	93
5.6.26 op = faddC.....	94
5.6.27 op = fsubC.....	95
5.6.28 op = fmulC.....	96

5.6.29 op = fdivC .....	97
5.6.30 op = fleqC .....	98
5.6.31 op = flssC .....	99
5.6.32 op = fgeqC .....	100
5.6.33 op = fgtrC .....	101
5.6.34 op = fneqC .....	102
5.6.35 op = feqlC .....	103
5.6.36 op = fadd .....	104
5.6.37 op = fsub .....	105
5.6.38 op = fmul .....	106
5.6.39 op = fdiv .....	107
5.6.40 op = feql .....	108
5.6.41 op = fneq .....	109
5.6.42 op = fgtr .....	110
5.6.43 op = fgeq .....	111
5.6.44 op = flss .....	112
5.6.45 op = fleq .....	113
5.7 Load and Store Instructions .....	114
5.8 Control Flow Instructions .....	116
5.9 Special Instructions .....	118
5.9.1 JumpI .....	118
5.9.2 CallI .....	119
5.9.3 EReturn .....	120
5.9.4 Streaming to and from the IEU and FEU .....	121
5.9.5 Streaming to and from the VEU .....	124
5.9.6 ASSERT .....	127
5.9.7 FASSERT .....	128
5.9.8 FLDMOV .....	129
5.9.9 FLDMOVX .....	130
5.9.10 FFB .....	131
5.9.11 CVTIF .....	132
5.9.12 CVTFI .....	133
5.9.13 TIF .....	134
5.9.14 TFI .....	135
5.9.15 TIV .....	136
5.9.16 TIVx .....	137
5.9.17 TFV .....	138
5.9.18 LLH .....	139
5.9.19 SLL .....	140
5.9.20 ReadPCW .....	141
5.9.21 WritePCW .....	142
5.9.22 ConsumeI .....	143
5.9.23 ConsumeF .....	144
5.9.24 SYNCH .....	145
5.9.25 LoadM .....	146
5.9.26 FLoadM .....	147
5.9.27 VLoadM .....	148
5.9.28 StoreM .....	149
5.9.29 FStoreM .....	150

5.9.30 VStoreM.....	151
5.9.31 LoadFifoII, LoadFifoFI, LoadFifoVI .....	152
5.9.32 LoadFifoIO, LoadFifoFO, LoadFifoVO .....	153
5.9.33 StoreFifoII, StoreFifoFI, StoreFifoVI .....	154
5.9.34 StoreFifoIO, StoreFifoFO, StoreFifoVO .....	155
5.9.35 LoadCTX .....	156
5.9.36 StoreCTX.....	157
5.9.37 SwapCTX.....	158
5.9.38 SwapLT .....	159



### **Document Derivation**

This report is a draft of the military standard manual for the WM Computer Architectures. The report is derived from "The WM Computer Architectures: Principles of Operation, Wm. A Wulf" (Computer Science Report No. TR-90-02, University of Virginia).

# 1. Scope and Purpose

## 1.1 Scope

This standard defines the WM instruction set architecture family. It does not define specific implementation details.

## 1.2 Purpose

The purpose of this document is to establish a single architecture family suitable for a spectrum of military applications from embedded signal processors to large-scale, high-performance, general purpose multi-computers.

## 1.3 Applicability

This standard is intended to be used to define only the instruction set architecture of a family of computers. System-unique requirements such as speed, weight, power, additional input/output commands, and environmental operating characteristics are defined in the computer specification for each computer. Application of this standard is not restricted to any particular function or specific hardware implementation or specific family member. This standard is not restricted to implementations of *stand-alone* computers such as a mission computer or fire control computer.

## 1.4 Benefits

The expected benefits of this standard instruction set architecture family are the use and re-use of available support software such as compilers and instruction level simulators and the ability to tailor the architecture and implementation to the specific mission capability. Other benefits may also be achieved such as: (a) reduction in total support software gained by the use of a standard instruction set architecture family for two or more computers in a weapon systems, and (b) software development independent of hardware development.

# 2. Referenced Documents

IEEE Floating Point Standard 754.

# 3. Definitions

## 3.1 Address

An address on the WM architecture is an  $i$ -bit signed value which identifies a location in memory where information is stored. Memory is 8-bit byte addressed. Note that addresses are *signed*; valid addresses lie in the range  $-2^{i-1} \dots (2^{i-1}-1)$ .

### **3.2 Alignment**

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).

### **3.3 Arithmetic logic unit (ALU)**

That portion of hardware in an execution unit in which arithmetic and logical operations are performed.

### **3.4 Bit**

Contraction of binary digit; may be either zero or one. In information theory, a binary digit is equal to one binary decision or the designation of one of two possible values or states of anything used to store or convey information.

### **3.5 Byte**

A group of eight binary digits.

### **3.6 Concurrent operations**

Operations specified by instructions are executed concurrently. The IFU, IEU, FEU and VEU execute instructions in parallel. The IEU and FEU have two pipelined ALUs which perform operations in parallel. Streamed LOAD/STORE operations imply potential concurrent performance of operations.

### **3.7 Condition code**

Scalar execution units can perform relational operations which generate condition codes that are to be consumed by conditional jump and consume instructions.

### **3.8 Deadlock**

The condition in which two or more units cannot execute further because each depends upon an action to be taken by another such unit.

### **3.9 Device**

A "device" is one hardware-understood page type. Device-specific operations are performed by reading and storing bit patterns into memory-mapped device registers in such a page.

### **3.10 Domain**

An addressing 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. Two tasks can share the same address space but may have different access to portions of that space.

### **3.11 Doubleword**

Sixty-four bits.

### **3.12 Entry**

A type of page. It 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. Traps are ECall's on predefined locations (in "page 0").

### **3.13 First-in-first-out queue (FIFO)**

A queue of items such that when the queue is read the value returned is the least recently enqueued item which has not been read and as a side effect it is removed from the queue.

### **3.14 Floating execution unit (FEU)**

That portion of a computer that performs floating point arithmetic and relational instructions.

### **3.15 Floating point register**

A register that may be used for floating point arithmetic and relational operations and general storage of temporary floating point data.

### **3.16 General purpose register**

A register that may be used for integer arithmetic, relational and logical operations, indexing, shifting, and general storage of temporary integer and logical data.

### **3.17 Halfword**

Sixteen bits.

### **3.18 Handler task**

The task which is dispatched to handle, i.e. react to, a specific kind of interrupt.

### **3.19 Input/output (I/O)**

That portion of a computer which interfaces to the external world.

### **3.20 Instruction**

A 32-bit word of program code which tells the WM computer what to do.

### **3.21 Integer execution unit (IEU)**

That portion of a computer that initiates singleton memory reads and writes and performs integer arithmetic, relational and logical instructions.

### **3.22 Instruction fetch unit (IFU)**

That portion of a computer which fetches instructions and dispatches some of them for execution in other units. The IFU executes certain special and control instructions.

### **3.23 Instruction set architecture (ISA)**

The attributes of a digital computer as seen by a machine (assembly) language programmer. ISA includes the processor and input/output instruction sets, their formats, operation codes, and addressing modes; memory management and partitioning if accessible to the machine language programmer; the speed of accessible clocks; interrupt structure; and the manner of use and format of all registers and memory locations that may be directly manipulated or tested by a machine language program. This definition excludes the time or speed of any operation, internal computer partitioning, electrical and physical organization, circuits and components of the computer, manufacturing technology, memory organization, memory cycle time, and memory bus widths.

### **3.24 Interrupt**

A special control signal that suspends the normal flow of the processor operations and allows the processor to respond to a logically unrelated or unpredictable event. An interrupt is essentially a forced context swap.

### **3.25 Load prefetch**

The case in which a load is started well before a memory data being read is needed. The purpose is to reduce the effect of cache misses and long memory access latency.

### **3.26 Memory**

That portion of a computer that holds data and instructions and from which they can be accessed.

### **3.27 Micro-concurrency**

The ability to dispatch multiple operations per cycle.

### **3.28 Multi-computer**

A computer composed of multiple WM computer processors capable of communicating with one another via messages.

### **3.29 Normal mode**

A state of a scalar execution unit FIFO register in which the location of data values currently in the FIFO or next to pass through the FIFO are specified by LOAD/STORE instructions.

### **3.30 Operation code (OPCODE)**

That part of an instruction that defines the machine operation to be performed.

### **3.31 Prefetch**

Because the IFU runs concurrently with the execution units, it may prefetch the next instruction before the execution unit to execute the current instruction has commenced to do so.

### **3.32 Program counter (PC)**

A register in the IFU that holds the address of the next instruction to be fetched.

### **3.33 Register**

A device in an execution unit for the temporary storage of one or more words to facilitate arithmetic, logical, or transfer operations.

### **3.34 Reserved**

Must not be used.

### **3.35 Right**

Permission to perform a specified access or action.

### **3.36 Stack**

A sequence of memory locations in which data may be stored and retrieved on a last-in-first-out (LIFO) basis.

### **3.37 Stream**

A 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.

### **3.38 Stream mode**

A state of an execution unit FIFO register in which the location of data values currently in, or next to pass through, the FIFO are specified by a stream instruction.

### **3.39 Streaming**

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 any of WM's execution units. Data items can then be processed at the speed of the consuming algorithm. Streaming permits many load/store operations to execute concurrently with other instructions.

### **3.40 Task**

A task is a "thread of control". The WM hardware supports a hardware-defined "task control block", TCB, to hold the state of the task when it is not executing.

### **3.41 Typed protection**

Each page is typed; only instructions appropriate to the type are permitted to reference a page. A task must have rights appropriate for the instruction.

### **3.42 Vector execution unit (VEU)**

That portion of a computer that performs vector arithmetic and relational instructions.

### **3.43 Vector register**

An (implementation-dependent size) block of registers that may be used for arithmetic and logical operations and general storage of temporary data.

### **3.44 Word**

Thirty-two bits.

### **3.45 Zero register**

A register that which has the value zero whenever read.

## 4. General Requirements

### 4.1 Function units

WM has three execution units under common control of the instruction fetch unit. The instruction set is partitioned so that each instruction is executed by a particular unit.

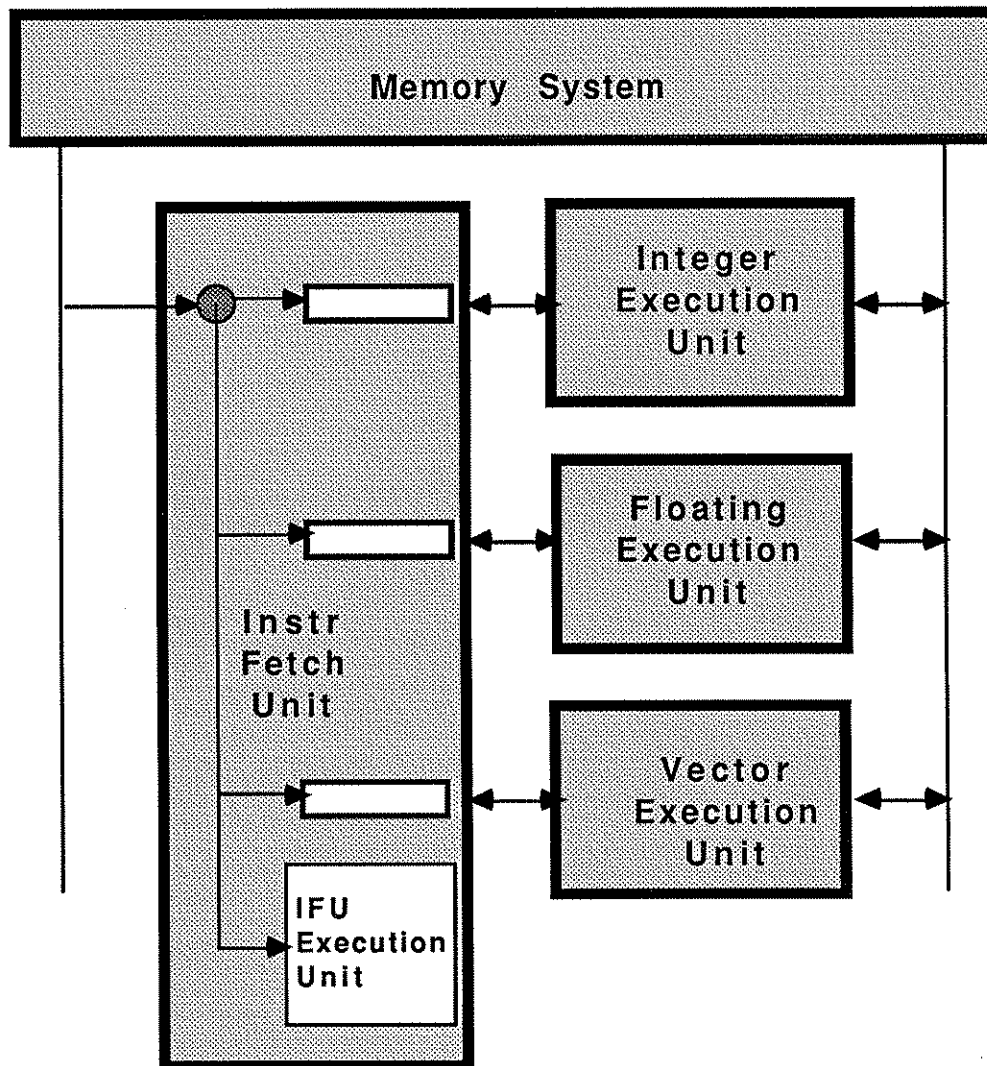


Figure 1: WM System Components

As shown in Figure 1<sup>1</sup>, the IFU can be thought of as enqueueing instructions for execution by each of the other execution units in a set of FIFOs. In addition the IFU

<sup>1</sup>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.

executes certain instructions itself, notably control instructions. The other execution units dequeue instructions and execute them as rapidly as their respective implementations permit.

#### 4.1.1 Scalar execution units

The scalar data manipulation instructions of WM are implemented by the integer and floating point execution units; each such instruction specifies 3 source operands, 2 operators, and a destination register, and evaluate an assignment of the form:

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

The source operand of integer/logical instructions may be the contents of a register, the contents of an input FIFO, 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 Figure 2. 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.

Integer/logical instructions refer to the integer registers; floating point instructions refer to the floating point registers. Conversion instructions refer to one register of each type as appropriate. In the floating point execution unit literals cannot be specified as operands; only floating register operands are permitted.

Relational operators 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 each conditional jump or consume instruction. The number of instructions containing relationals preceding the associated conditional jump or consume instruction must not exceed the size of the condition bit FIFO.



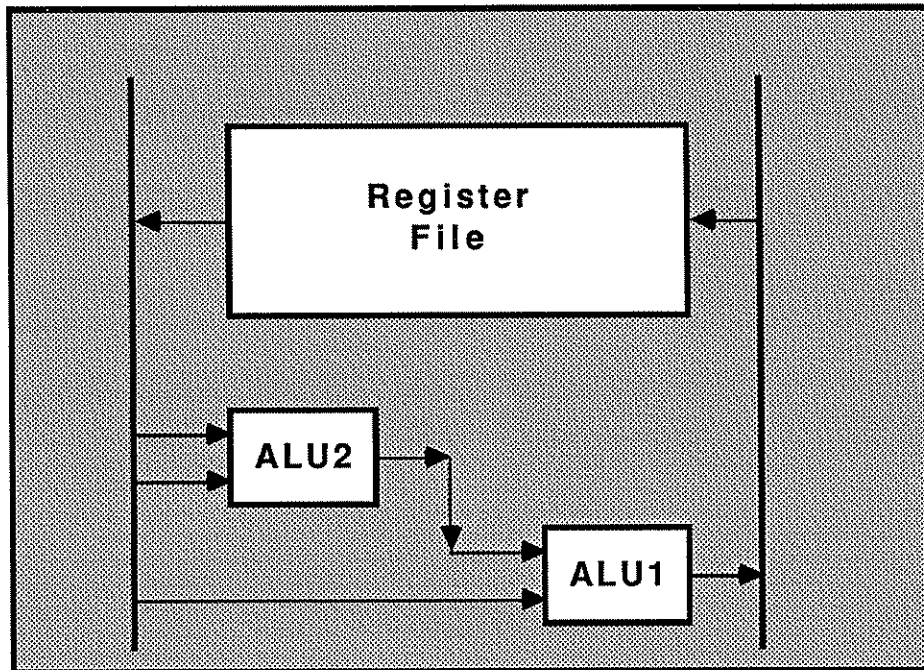


Figure 2: Scalar Execution Unit Structure

#### 4.1.1.1 Data dependency rule

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.

Data dependencies are defined with respect to instructions for the *same* execution unit!

#### 4.1.2 Vector execution unit

The Vector Execution Unit supports integer, logical and floating point operations on "blocks" of  $N$   $y$ -bit items, where  $N$  is an implementation defined parameter.

The vector instructions of WM specify a single operation. The performance of the operation is conditioned on an item-by-item basis by a boolean vector specified by the third source operand. The boolean "mask" determines whether components of the result vector are affected by the operation. In general, the form of a vector instruction is

$$R0 := (R1 \text{ op } R2) \text{ if } R3$$

Each instruction performs the computation

**forall**  $k, 0 \leq k < N, R0_k := \text{if } R3_k \neq 0 \text{ then } (R1_k \text{ op } R2_k) \text{ else } R0_k \text{ fi}$

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.

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.

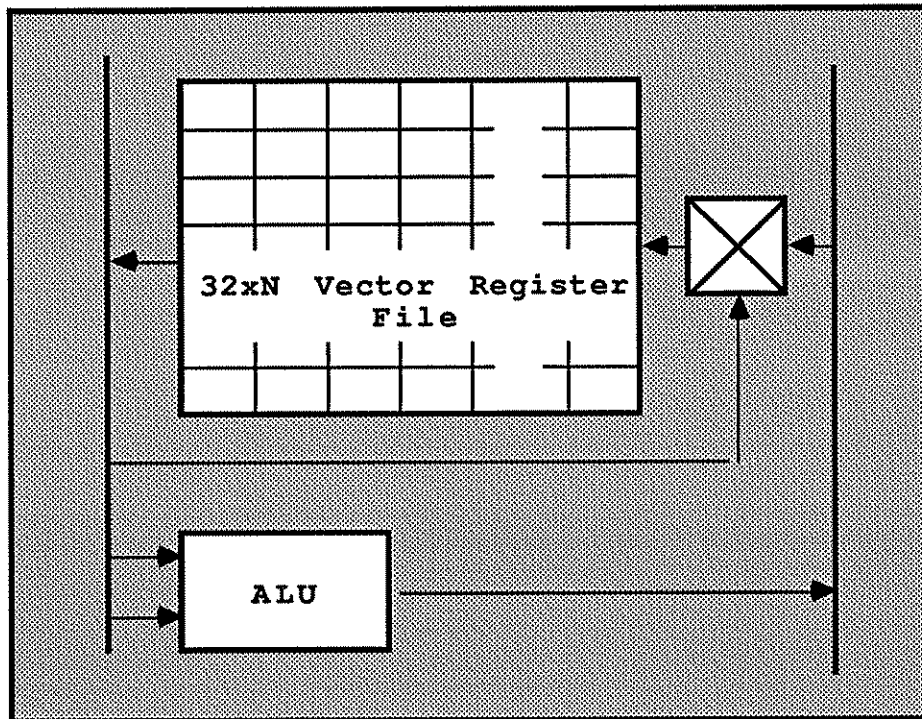


Figure 3: Vector Execution Unit

#### 4.1.3 Instruction fetch unit

The instruction fetch unit fetches sequential instructions from memory based on the value of the Program Counter. Fetched instructions are either executed by the IFU or queued for execution by the one execution unit capable of executing the fetched instruction. The IFU executes selected special instructions and all control instructions. Control instructions replace the Program Counter with a new value, the target address.

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; these support jumps on the on "stream count not zero" for each of the input and output streams.

There are two call instructions: Call and ECall. Call simply stores the current PC in register 4 and jumps to the specified destination. ECall performs the function of a "supervisor call".

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 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 because they do not need to specify a PC-relative address: they are JumpI (Jump Indirect), CallI (Call Indirect) and EReturn (Return from ECall).

#### **4.1.4 Parameter bypass**

Register 1 in each of the scalar execution units is also a FIFO, 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.

Register 1 can hold a short queue of temporary values -- in particular parameters during a subroutine call. The caller enqueues actual parameters, and the called routine dequeues formals.

A call consists of at least:

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

#### 4.1.5 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) 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.

Either register 0 or register 1 in each of the execution units may be used in stream mode. Streaming 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. Normal mode for register 1 is the parameter bypass mode. Stream mode is identical for both registers in all execution units.

When in streaming mode, the first/next data transfer occurs due to a single "start streaming" instruction which 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 normal 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.

##### 4.1.5.1 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<sup>1</sup> (RL2), a stride<sup>2</sup> (RL3), and which FIFO to use (0 or 1).

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

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 normal -- i.e., non-streaming -- mode. A

---

<sup>1</sup> 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.

<sup>2</sup> In bytes.

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.

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.

Streaming instructions may cause Page Fault exceptions. If a Page Fault exception occurs during a memory read, the exception is not raised until an attempt to read register 0 or 1 unsuccessfully. If such an exception occurs during a write of register 0 or 1 (to be written into memory), the exception is raised immediately.

#### **4.1.5.2 Streaming to and from the VEU**

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:

- data is moved in "blocks" of N entities.
- because 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.
- because the VEU supports integer, logical, and floating operations, appropriate streaming operations are provided to do the proper form of operand expansion or contraction.
- because 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".

#### **4.1.6 Special instructions and synchronization**

Responsibility for execution of the special instructions resides in the Instruction Fetch Unit; 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<sup>1</sup>.

The class of special instructions include instructions to

- provide access to special state, to help save and restore the state of the processor and the individual FIFOs efficiently and to perform context loads, stores and swaps.
- convert between the integer and floating numeric data types. Convert instructions reference one register in the integer execution unit and one in the floating execution unit as appropriate. In addition, there exist transfer instructions, which use "bit copy" semantics to transfer between two registers in different execution units (integer, floating and vector). No data conversion is performed except as necessary to expand/contract their representation<sup>2</sup>.
- 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.
- move, with or without sign extension, a field within a word in the IEU. These instructions provide for field extraction (with or without sign extension) and basic shifts.
- find the first (different) bit, i.e. the location of the most significant bit that is different from the sign bit in a value.
- consume one condition code as do conditional jump instructions without dependence on its value.
- read and write the Program Control Word.

The SYNCH instruction 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.

#### **4.1.7 Deadlock**

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. The WM computer will detect a deadlock and trap.

The minimum sizes of the various FIFOs are specified 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.

---

<sup>1</sup> 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.

<sup>2</sup> 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.

## 4.2 Data formats

The instruction set shall support  $i$ -bit fixed point precision,  $f$ -bit floating point single precision, and  $v$ -bit vector (fixed and floating) point precision data in two's complement representation. A member of the family is denoted by three parameters, and is denoted  $WM_{i,f,v}$ . 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:

$$\begin{aligned}i &\in \{16, 32, 64\} \\f &\in \{0, 32, 64\} \\v &\in \{0, 32, 64\}\end{aligned}$$

Data format determines what LOAD and STORE instructions are supported on a particular family member: 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.

### 4.2.1 Data alignment

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. This least significant address bit is ignored when accessing such data elements. Instructions are aligned on word boundaries. Doublewords are aligned on sixty-four bit boundaries.

### 4.2.2 Data sizes

Data elements in memory may be stored in 8-bit byte, 16-bit halfword, 32-bit word, or 64-bit doubleword sizes.

### 4.2.3 Data Types

The WM architecture supports values of several types: boolean values, signed integer values, and floating point values.

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.

#### 4.2.3.1 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 two 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.  $i$ -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.

#### 4.2.3.2 Signed integer values

Arithmetic on 2's-complement  $j$ -bit signed integers with the most significant bit (MSB) as the sign bit 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  $j$ -bit register quantities. Unsigned integers are not supported by the machine. Explicit underflow checking is required when synthesizing unsigned arithmetic with this architecture.

#### 4.2.3.3 Floating point values

Arithmetic is performed using the  $f$ -bit value obeying the IEEE floating point standard.

### 4.3 Instruction formats

Six instruction formats are supported. Each instruction is 32-bits. The operation code consists of bits 4..11 or 8..11 of the instruction.

#### 4.3.1 Literals in instructions

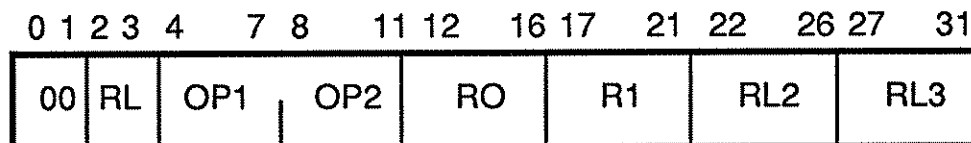
Certain instructions may specify unsigned, 5-bit literals as operands. These literals are the integers 1-32 and are encoded in the obvious way, except that 32 is encoded as zero.

#### 4.3.2 Instruction format notation

The WM ISA definition has five instruction formats.

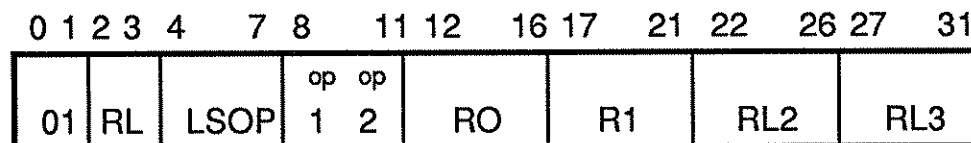
#### 4.3.3 Integer format instructions.

Integer arithmetic and logical instructions are executed by the IEU. The three source specifiers -- R1, RL2 and RL3 -- may name integer registers. RL2 and RL3 may also name 5-bit literals. OP1 is the operation with source inputs R1 and RL2. OP2 is the operations with source inputs consisting of the result of OP1 and RL3. R0 is the destination integer register.



#### 4.3.4 LOAD/STORE format instructions.

Load and Store instructions are executed by the IEU. R0, R1, RL2 and RL3 may name integer registers. RL2 and RL3 may name 5-bit literals. OP1 is the operation with source inputs R1 and RL2. OP2 is the operations with source inputs consisting of the result of OP1 and RL3. R0 is the destination register into which a computed address is stored.





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

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

$R0 := (R1 \text{ op1 } RL2) \text{ op2 } RL3$

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

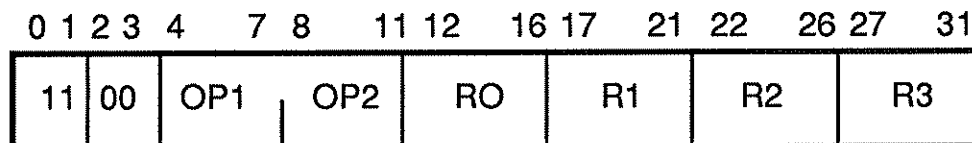
The type/size of the data to be read or written is specified by the LOAD or STORE instruction.

The memory system ensures that certain sequences of load/store operations are performed properly. Loads and stores from one execution unit are not synchronized with those of the other!

There are no LOAD/STORE instructions for the Vector Execution Unit. All memory-VEU transfers are accomplished with streaming instructions.

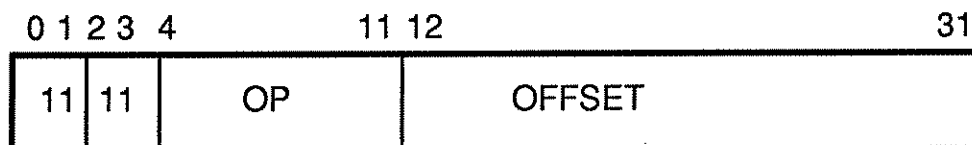
#### 4.3.5 Floating point format instructions

Floating point arithmetic instructions are executed by the FEU. R0, R1, R2 and R3 may name floating registers. OP1 is the operation with source inputs R1 and R2. OP2 is the operations with source inputs consisting of the result of OP1 and R3. R0 is the destination register.



#### 4.3.6 Control format instructions

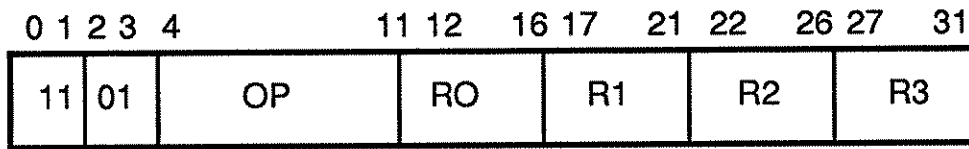
Control instructions are executed by the IFU.



The offset is extended with two least significant zero digits.

#### 4.3.7 Vector format instructions.

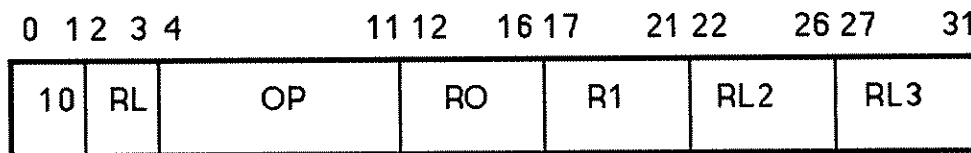
Vector instructions are executed by the VEU.



R0, R1, R2, and R3 are vector registers. R0 is the destination.

#### 4.3.8 Special format instructions

Special instructions are executed by the IFU.



R0, the destination register, and R1 are general registers. RL2 and RL3 are either literals or general registers.

### 4.4 Registers and support features

#### 4.4.1 General registers

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, whenever read, their value is zero.

#### 4.4.2 Special registers

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. Details of the special registers are discussed in section 4.6.

#### 4.4.3 Stack

The WM architecture defines

- Stack Limit as register 2, and
- Stack Index as register 3 of the integer execution unit.

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.

#### 4.4.4 Register convention

The conventions with respect to register usage are:

- r 3     SI
- 
- 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 5     FP (frame-pointer; software convention)
- r 6     HP (exception-handler pointer; software convention)

### 4.5 Memory

#### 4.5.1 Memory reads & writes

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 when the queues are in normal mode.

LOADs and STOREs specify an address. A LOAD is a request to enqueue data from memory into a specified input FIFO, and a STORE is a request to dequeue data from a specified output FIFO and store it to memory.

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

To dequeue data from an input FIFO, an instruction references register 0 as a source operand. To enqueue data in an output FIFO, an instruction specifies register 0 as the destination of a computation. "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.

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 streaming.

Multiple LOAD instructions (with implementation dependent limits) may be executed; the data is enqueued in an input FIFO in the order of the LOAD instructions. Access to register 0 dequeues the next value from the FIFO for use.

LOADs precede access to the read value. STOREs and the production of the data value to be output may occur in either order. The action of writing to memory is taken only when an appropriate pair of instructions have **both** been executed. 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.

The WM architecture defines minimum sizes of the input and output FIFOs; actual sizes are implementation defined. 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  $y$ -bit entries in the vector unit's input FIFO, and
- 1 N-component block of  $y$ -bit entries in the vector unit's input FIFO.

These minimums ensure that any single instruction can execute, even if it names all its source operands and its destination operand as FIFOs.

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*.

## 4.6 Operating system support

### 4.6.1 Task state

A task is a thread of control. 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

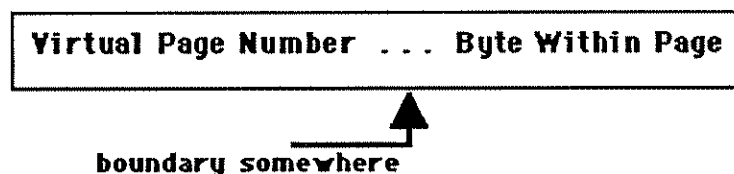
The Program Status Word collects a number of fields that reflect status of the task, such as the run/halt bit, the interrupt enabling bit, the priority and the condition FIFOs. For example, the PSW could include:

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 will resume the previous task.

A task's virtual address space is divided into pages. 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 j-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.

#### 4.6.2 Protection

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<sup>1</sup>.

<sup>1</sup>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".

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

type	rights

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

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

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

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

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

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.

#### 4.6.3 Address mapping

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<sup>1</sup>.

---

<sup>1</sup>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".

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<sup>1</sup>
- 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.

#### 4.6.4 Initialization of the machine

Machine implementation determines initialization.

#### 4.7 Devices

Each device connected to WM must conform to the following 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.
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.

---

<sup>1</sup> The "locked bit" is a software convention; it is, however, checked by hardware when DMA IO transfers are specified. See Section 4.5.



#### 4.8 Input/output

Control of input/output devices is "memory mapped". A portion of the physical address space reserved for "device registers". Unprivileged applications program are permitted to directly access IO devices.

At least three devices are required of all implementations:

- "the CPU", control and status registers for the processor itself. One processor can probe or start/stop another or itself with bit set/reset operations on the appropriate 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.

#### 4.9 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.

##### 4.9.1 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.

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.

#### 4.9.2 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.

### 4.9.3 Exceptions

Exceptions are listed by function unit:

#### 4.9.3.1 Integer 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

#### 4.9.3.2 Load/Store 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.

#### 4.9.3.3 Control 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.

#### **4.9.3.4 Floating point 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.

## 5. Detailed Requirements

### 5.1 Instruction set notation

#### 5.1.1 Registers

##### 5.1.1.1 General registers

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, these writes have no effect.

Registers in the various execution units are given distinct mnemonic 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.

##### 5.1.1.1.1 IEU registers

**r 0** Register r0 refers to an input or an output FIFO depending on the context. Each FIFO consists of an implementation defined number of i-bit values. A 'consume\_count' is associated with each FIFO. A FIFO is in streaming mode if consume\_count is non-zero.

Input FIFO: r0 refers to the input FIFO r0.input when a) r0 appears to the right of the  $\leftarrow$  symbol or b) r0 appears in a value context as in a *if* statement or c) when the input FIFO is explicitly specified.

Output FIFO: r0 refers to the output FIFO r0.output when a) r0 appears to the left of the  $\leftarrow$  symbol or b) when the output FIFO is explicitly specified.

An output FIFO is a FIFO of records. Each record has three fields - a 'value' field which may contain data or address, a 'qualifier' field which qualifies the contents of the 'value' field as DATA or ADDR and a 'size' field which stores the number of bytes to be transferred to memory. The default qualifier is DATA. The size field is meaningful only when the qualifier is ADDR.

**r 1** Register r1 is similar to r0 with the difference that an assignment to the output FIFO r1 results in the value being enqueued in the r1 input FIFO as well.

**r2-r30** These denote i-bit integer registers.

**r31** r31 is a i-bit register with the value 0. Assignments to r31 have no effect on its contents.

#### 5.1.1.1.2 FEU registers

f0-f31 These are similar to their integer counterparts except that a) these are f-bit floating point registers and b) input/output FIFO full/empty are not exception conditions.

#### 5.1.1.1.3 VEU registers

v0 Register v0 refers to an input or an output FIFO depending on the context. Each FIFO contains blocks of elements. Each block contains N v-bit values. A 'tag' is associated with each v-bit value. Both N and the depth of the FIFO are implementation defined. A 'consume\_count' is associated with each FIFO. A FIFO is in streaming mode if consume\_count is non-zero.

Input FIFO: v0 refers to the input FIFO v0 when a) v0 appears to the right of the  $\leftarrow$  symbol or b) v0 appears in a value context as in an *if* instruction or c) when the input FIFO is explicitly specified.

Output FIFO: v0 refers to the output FIFO v0 when a) v0 appears to the left of the  $\leftarrow$  symbol or b) when the output FIFO is explicitly specified.

Like the IEU/FEU output FIFOs, v0 is also a FIFO of records, with the same fields as the IEU/FEU output FIFOs. As mentioned above, v0 has an additional field - the tag field with two possible values - CHANGED and UNCHANGED. The default for the qualifier field is DATA. The tag field is used only when the qualifier is DATA and the size field is used only when the qualifier is ADDR.

v1 Register v1 is similar to v0.

v2-v30 Each of these registers is a block of N elements, each element consists of a v-bit value and the associated tag.

v31 Register v31 is like registers v2-v30, except that 1) the value in the v-bit data field of each element is 0, and 2) assignments to v31 have no effect on this value.

#### 5.1.1.2 Implementation dependent registers

The registers listed here are specified for use in the instruction semantics specification in Chapter V. Other implementations may define different implementation dependent registers.

CC1, CC2 Registers that hold intermediate condition code values. The possible values are TRUE, FALSE and NOT EVAL.

X1, X2 IEU internal registers (at least i-bits).

Y1, Y2 FEU internal registers (at least f-bits).

Z1, Z2 VEU internal registers (at least v-bits).

JC Boolean register used in the specification of control flow instructions.

SM,  
 SCount     Registers used in the specification of stream operations (at least i bits).

SMV        Record used in the specification of vector stream operations : it has two fields: value and tag.

SDec        1-bit boolean register used in the specification of stream operations.

#### 5.1.1.3 Special registers

These are described in Section 4.4.2.

### 5.1.2 Symbols

#### 5.1.2.1 " $\Leftarrow$ "

##### 5.1.2.1.1 Symbols to the left of $\Leftarrow$ :

Output FIFO     The values to the right of the  $\Leftarrow$  symbol are enqueued in the specified fields of the output FIFO.

SMV              The fields of the record get the values of the corresponding fields of the record to the right of the  $\Leftarrow$ .

Register        The value(s) to the right of the  $\Leftarrow$  are assigned to the specified field(s) of the register.

##### 5.1.2.1.2 Symbols to the right of $\Leftarrow$

Output FIFO     The FIFO is dequeued and the record obtained is the value of the symbol.

Register        The value of the contents of the register is the value of the symbol.

#### 5.1.2.2 " $\leftarrow$ ": The assignment operator.

The assignment operator syntax is

$X \leftarrow \text{expression}$  (Assignment to X from expression)

The result of the assignment depends on the symbols that appears to the left (in a name context) and to the right (in a value context) of the  $\leftarrow$  operator.

##### 5.1.2.2.1 Symbols to the left of $\leftarrow$

Input FIFO        An assignment to an input FIFO has the effect of enqueueing the value of the expression to the right of the  $\leftarrow$  operator.

IEU/FEU Output FIFO     If the 'qualifier' field of the record at the head of the FIFO is ADDR, then the FIFO is dequeued, and the value of the expression to the right of the  $\leftarrow$  symbol is written to the memory location given by the 'value' field of the dequeued record.  
 Otherwise, the value of the expression to the right of the  $\leftarrow$  symbol is enqueueing into the 'value' field at the tail of the FIFO, and the corresponding 'qualifier' field is set to DATA.

In either of the above cases, if the FIFO was in streaming mode, `consume_count` is decremented by one.  
 If the output FIFO is `r1` or `f1`, then the assignment to the FIFO results in the value being enqueued in the corresponding input FIFO as well.

VEU output FIFO	<p>If the 'qualifier' field of the record at the head of the FIFO is <code>ADDR</code>, then the FIFO is dequeued. If the 'tag' field is being assigned <code>CHANGED</code>, the value in the value field of the register to the right of the <math>\leftarrow</math> symbol is written to the memory location given by the 'value' field of the dequeued record.          Otherwise the value in the value field of the register to the right of the <math>\leftarrow</math> symbol is enqueued into the 'value' field at the tail of the FIFO, the corresponding 'qualifier' field is set to <code>DATA</code> and the tag field is set as specified.          In either of the above cases, if the FIFO was in streaming mode, <code>consume_count</code> is decremented by one.</p>
<code>r2-r30</code>	The value of the expression to the right of the $\leftarrow$ is assigned to the specified register.
<code>f2-f30, v2-v30</code>	Similar to <code>r2-r30</code>
<code>r31, f31, v31</code>	The assignment does not change the contents of the specified register.
<code>CCi, CCf</code>	An assignment to <code>CCi/CCf</code> enqueues the value of the boolean expression to the right of the $\leftarrow$ operator.
<code>M[addr, size]</code>	<i>size</i> bytes of memory starting at location <i>addr</i> are assigned the value of the expression to the right of the $\leftarrow$ operator. The value of the expression should also be <i>size</i> bytes in length.

#### 5.1.2.2.2 Symbols to the right of $\leftarrow$

Input FIFO	The input FIFO is dequeued and this is the value of the symbol. If the FIFO is in streaming mode, the corresponding <code>consume_count</code> is decremented for each value dequeued. If the FIFO is empty and a value is scheduled to be loaded, the assignment operation blocks until a value is enqueued.
Output FIFO	The output FIFO is dequeued and the value in the 'value' field is the value of the symbol.
<code>r2-r31</code> <code>f2-f31, v2-v31</code>	The value of the symbol is the contents of the specified register. Similar to <code>r2-r31</code>
<code>CCi, CCf</code>	The specified condition code FIFO is dequeued and this is the value of the symbol.



M[addr, size]            The value of *size* bytes of data from the memory starting at location *addr*.

#### 5.1.2.3 ":" : The bit selection operator

reg:i

Selects the *i*<sup>th</sup> bit of the specified register 'reg'.

reg:i-j

Selects *j-i+1* bits of the specified register starting from bit *i*.

#### 5.1.2.4 "::" : The operator-argument operator

opr :: args

Passes the arguments *args* to the operator *opr*.

#### 5.1.2.5 "lsl" : The arithmetic shift left operator

X lsl Y

Shifts the register *X* left by the amount specified by the register/literal *Y*

#### 5.1.2.6 "asr" : The arithmetic shift right operator

X asr Y

Arithmetic shifts register *X* right by the amount specified by the register/literal *Y*.

#### 5.1.2.7 "&&": The bitwise and operator

X && Y

The two registers *X* and *Y* are bitwise and-ed.

#### 5.1.2.8 'and' : The logical and operator

X and Y

The two boolean values *X* and *Y* are logical and-ed and the result is **TRUE** or **FALSE**.

#### 5.1.2.9 "||" The bitwise or operator

X || Y

The registers *X* and *Y* are bitwise or-ed.

#### 5.1.2.10 "or" The logical or operator

X or Y

The two boolean values *X* and *Y* are logical or-ed and the result is **TRUE** or **FALSE**.

#### 5.1.2.11 "EQV" The bitwise equivalence operator

X EQV Y

The registers *X* and *Y* are checked for equivalence bit by bit. For two corresponding bits that are the same, a 1 is produced and for two corresponding bits that are different a 0 is produced.

#### 5.1.2.12 "+, -, '-', \*, /, /"

These have the usual meaning. These operators are overloaded -- they operate on two integer or two floating point operands and produce an integer or floating point representation accordingly.

#### 5.1.2.13 "=", "<>", "<", "<=", ">=", ">"

These have the usual meaning. These relational operators are overloaded -- they operate on two integer or two floating point operands and produce a boolean result.

#### 5.1.2.14 "φ"

The 'don't care' symbol. The value of the symbol is immaterial.

### 5.1.3 Functions

The following functions have been used for describing the semantics of some instructions. They create no side effects -- each takes an argument, and returns a value.

float	Takes a floating point register as argument and returns the floating point number corresponding to the contents of the register.
int_to_float	Takes an integer register as argument and returns the integer in floating point format.
float_to_int	Takes a floating point register as argument and returns the integer corresponding to the floating point register, rounded as specified by a particular machine implementation.
sign_extend	Takes an integer register or literal as argument and returns the sign extended form of this argument or the argument itself depending on the 'sign extension' column of the table for the instruction. Sign extension is performed to <i>i</i> bits, where <i>i</i> is the size of the IEU registers.
relational	Takes an 'op' as argument and returns TRUE if 'op' is a relational operator and FALSE otherwise. =, <>, <, <=, >= and > are the relational ops.
sizeof	Takes an integer or floating point register as argument and returns the size of the register in bits.
qualifier_type	Takes a FIFO as an argument and returns the qualifier field of the element at the head of the FIFO. The FIFO is <i>not</i> dequeued.

### 5.1.4 Operations

The following operations have been used to describe the semantics of instructions. These operations have the side effect of changing the machine state. The general syntax for these operations is

operation-specifier :: arguments

Possible operation specifications:

exception	Takes as an argument the cause of the exception (e.g. Double Stream) and causes a jump to the address specified in 4.9.2. The jump is implemented as an ECall.
initiate stream operation	Takes as arguments a FIFO name, a base address, a count, a stride and the type of streaming operation to be performed (stream_in, stream_out or stop_streaming). It starts the specified stream operation on the specified FIFO. It is described in more detail in the description of stream instructions.

initiate vector  
stream operation      Similar to 'initiate stream operation'. It is described in more detail in the description of vector stream instructions.

### 5.1.5 Miscellaneous values

<b>TRUE, FALSE</b>	The boolean values TRUE and FALSE respectively.
<b>NOT EVAL</b>	A value used for condition code operations. This value is neither TRUE nor FALSE and means 'not evaluated'.
<b>AND</b>	A constant with value 1.
<b>CHANGED, UNCHANGED</b>	Constants used as tags in vector operations. A tag with value 'CHANGED' indicates that the corresponding value was modified and a tag with the value 'UNCHANGED' indicates that the corresponding value was not modified.

### 5.2 Mnemonic conventions

Each instruction has an associated mnemonic convention. e.g. a load instruction might be written as `L8i r5:= (r6 + r7) - r8`.

Integer instructions begin with an 'int', e.g. `int r4 := (r5 + r6) - r0`.

Floating instructions begin with a 'flt', e.g. `flt f4 := (f5 + f6) - f0`.

Vector instructions begin with a 'vec', e.g. `vec v4 := (v5 + v6) if v0`.

### 5.3 Execution semantics

Cycles	Each instruction is composed of a finite sequence of cycles. An instruction description may use any of the following cycles: Cycle 1, Cycle 2, Synch Cycle or the Memory Cycle. Cycle 1 and Cycle 2 terminate when the last operation in these cycles is executed. The Synch Cycle synchronizes the IFU, IEU, FEU and the VEU. The Synch Cycle terminates when all the instructions prior to the instruction executing the Synch Cycle are executed. The Memory Cycle is used in instruction descriptions that require memory references and terminates when the last operation in the cycle is executed.
--------	--

Execution	Each instruction is executed during a finite sequence of cycles. The execution of an instruction is complete when the last cycle in the instruction description terminates. However, for stream instructions, execution of the instruction is complete when the Memory Cycle commences.
-----------	---

Exceptions and Interrupts	If an exception is raised during the execution of a cycle, the succeeding cycle(s) are not executed, and the instruction execution terminates. Interrupts are handled at the end of each instruction. Exceptions and interrupts are discussed in Section 4.9.
---------------------------	---

## 5.4 Integer Arithmetic and Logical Instructions

Mnemonic: int R0 := (R1 op1 RL2) op2 RL3

Format:

0	1	2	3	4	7	8	11	12	16	17	21	22	26	27	31
0 0	RL	OP1	OP2	R0	R1	RL2	RL3								

OP1, OP2		
<u>symbol</u>	<u>encoding</u>	<u>operation</u>
+	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

Description: The operation op1 is performed during cycle 1 with R1 and RL2 as operands. If no exception conditions are generated, operation op2 is performed during cycle 2 with the result of op1 and RL3 as operands. If no exception conditions are generated, the result is written to R0.

Cycle Description:

**Cycle 1:** Cycle 1 for op1

**Cycle 2:** Cycle 2 for op2

## Integer Arithmetic and Logical Instructions

### 5.4.1 op = +

**Cycle 1:**  $X1 \leftarrow R1 + RL2$   
If overflow then PSW:6  $\leftarrow$  1  
CC1  $\leftarrow$  NOT EVAL

**Cycle 2:**  $X2 \leftarrow X1 + RL3$   
CC2  $\leftarrow$  NOT EVAL  
if overflow then  
    if CC1  $\neq$  FALSE then PSW:6  $\leftarrow$  1  
else  
    if CC1  $\neq$  FALSE then R0  $\leftarrow$  X2  
    if relational (op1) then CCi  $\leftarrow$  CC1

## Integer Arithmetic and Logical Instructions

### 5.4.2 op = asl

**Cycle 1:**  $X1 \leftarrow R1 \text{ asl } RL2$   
 $CC1 \leftarrow \text{NOT EVAL}$

**Cycle 2:**  $X2 \leftarrow X1 \text{ asl } RL3$   
 $CC2 \leftarrow \text{NOT EVAL}$   
If  $CC1 \neq \text{FALSE}$  then  $R0 \leftarrow X2$   
If relational (op1) then  $CCi \leftarrow CC1$

## **Integer Arithmetic and Logical Instructions**

### **5.4.3 op = <**

**Cycle 1:**  $X1 \leftarrow R1$   
If  $R1 < R2$  then  $CC1 \leftarrow \text{TRUE}$  else  $CC1 \leftarrow \text{FALSE}$

**Cycle 2:** If  $X1 < R3$  then  $CC2 \leftarrow \text{TRUE}$  else  $CC2 \leftarrow \text{FALSE}$   
If relational (op1) then  
If  $PCW:0 = \text{AND}$  then  $CCi \leftarrow CC1 \text{ and } CC2$  else  $CCi \leftarrow CC1 \text{ or } CC2$   
else  $CCi \leftarrow CC2$   
If  $CCi$  was assigned **TRUE** then  $R0 \leftarrow X1$

## Integer Arithmetic and Logical Instructions

### 5.4.4 op = -

**Cycle 1:**  $X1 \leftarrow R1 - RL2$   
If underflow then  $PSW:7 \leftarrow 1$   
 $CC1 \leftarrow \text{NOT EVAL}$

**Cycle 2:**  $X2 \leftarrow X1 - RL3$   
 $CC2 \leftarrow \text{NOT EVAL}$   
If underflow then  
    If  $CC1 \neq \text{FALSE}$  then  $PSW:7 \leftarrow 1$   
else  
    If  $CC1 \neq \text{FALSE}$  then  $R0 \leftarrow X2$   
    If relational (op1) then  $CCi \leftarrow CC1$



## Integer Arithmetic and Logical Instructions

### 5.4.5 op = '-'

**Cycle 1:**  $X1 \leftarrow RL2 - R1$   
If underflow then  $PSW:7 \leftarrow 1$   
 $CC1 \leftarrow \text{NOT EVAL}$

**Cycle 2:**  $X2 \leftarrow RL3 - X1$   
 $CC2 \leftarrow \text{NOT EVAL}$   
if underflow then  
    if  $CC1 \neq \text{FALSE}$  then  $PSW:7 \leftarrow 1$   
else  
    if  $CC1 \neq \text{FALSE}$  then  $R0 \leftarrow X2$   
    if relational (op1) then  $CCi \leftarrow CC1$

## **Integer Arithmetic and Logical Instructions**

### **5.4.6 op = and**

**Cycle 1:**  $X1 \leftarrow R1 \ \&\& \ RL2$   
 $CC1 \leftarrow \text{NOT EVAL}$

**Cycle 2:**  $X2 \leftarrow X1 \ \&\& \ RL3$   
 $CC2 \leftarrow \text{NOT EVAL}$   
if  $CC1 \neq \text{FALSE}$  then  $R0 \leftarrow X2$   
if relational (op1) then  $CCi \leftarrow CC1$

## **Integer Arithmetic and Logical Instructions**

### **5.4.7 op = or**

**Cycle 1:**  $X1 \leftarrow R1 \parallel RL2$   
 $CC1 \leftarrow \text{NOT EVAL}$

**Cycle 2:**  $X2 \leftarrow X1 \parallel RL3$   
 $CC2 \leftarrow \text{NOT EVAL}$   
If  $CC1 \neq \text{FALSE}$  then  $R0 \leftarrow X2$   
If relational (op1) then  $CCi \leftarrow CC1$

## **Integer Arithmetic and Logical Instructions**

### **5.4.8 op = eqv**

**Cycle 1:**  $X1 \leftarrow R1 \text{ EQV } RL2$   
 $CC1 \leftarrow \text{NOT EVAL}$

**Cycle 2:**  $X2 \leftarrow X1 \text{ EQV } RL3$   
 $CC2 \leftarrow \text{NOT EVAL}$   
**If**  $CC1 \neq \text{FALSE}$  **then**  $R0 \leftarrow X2$   
**If** relational (op1) **then**  $CCi \leftarrow CC1$

## Integer Arithmetic and Logical Instructions

### 5.4.9 op = \*

**Cycle 1:**  $X1 \leftarrow R1 * RL2$   
If overflow then PSW:6  $\leftarrow 1$   
CC1  $\leftarrow$  NOT EVAL

**Cycle 2:**  $X2 \leftarrow X1 * RL3$   
CC2  $\leftarrow$  NOT EVAL  
If overflow then  
    If CC1  $\neq$  FALSE then PSW:6  $\leftarrow 1$   
else  
    If CC1  $\neq$  FALSE then R0  $\leftarrow X2$   
    If relational (op1) then CCi  $\leftarrow$  CC1

## Integer Arithmetic and Logical Instructions

### 5.4.10 op = /

**Cycle 1:** If RL2 = 0 then PSW:4  $\leftarrow$  1 else  
X1  $\leftarrow$  R1 / R2  
If underflow then PSW:7  $\leftarrow$  1  
CC1  $\leftarrow$  NOT EVAL

**Cycle 2:** If RL3 = 0 then  
If CC1  $\neq$  FALSE then PSW:4  $\leftarrow$  1  
else  
X2  $\leftarrow$  X1 / RL3  
CC2  $\leftarrow$  NOT EVAL  
If underflow then  
If CC1  $\neq$  FALSE then PSW:7  $\leftarrow$  1  
else  
If CC1  $\neq$  FALSE then R0  $\leftarrow$  X2  
If relational (op1) then CCi  $\leftarrow$  CC1

## Integer Arithmetic and Logical Instructions

### 5.4.11 op = '/'

**Cycle 1:** If R1 = 0 then PSW:4  $\leftarrow$  1 else  
X1  $\leftarrow$  RL2 / R1  
If underflow then PSW:7  $\leftarrow$  1  
CC1  $\leftarrow$  NOT EVAL

**Cycle 2:** If X1 = 0 then PSW:4  $\leftarrow$  1 else  
X2  $\leftarrow$  RL3 / X1  
CC2  $\leftarrow$  NOT EVAL  
If underflow then  
    If CC1  $\neq$  FALSE then PSW:7  $\leftarrow$  1  
else  
    If CC1  $\neq$  FALSE then R0  $\leftarrow$  X2  
    If relational (op1) then CCi  $\leftarrow$  CC1

## Integer Arithmetic and Logical Instructions

### 5.4.12 op = =

**Cycle 1:**  $X1 \leftarrow R1$   
If  $R1 = RL2$  then  $CC1 \leftarrow \text{TRUE}$  else  $CC1 \leftarrow \text{FALSE}$

**Cycle 2:** If  $X1 = RL3$  then  $CC2 \leftarrow \text{TRUE}$  else  $CC2 \leftarrow \text{FALSE}$   
If relational (op1) then  
    If  $PCW:0 = \text{AND}$  then  $CCi \leftarrow CC1$  and  $CC2$  else  $CCi \leftarrow CC1$  or  $CC2$   
else  $CCi \leftarrow CC2$   
If  $CCi$  was assigned **TRUE** then  $R0 \leftarrow X1$

.....



## **Integer Arithmetic and Logical Instructions**

### **5.4.13 op = <>**

**Cycle 1:**  $X1 \leftarrow R1$   
If  $R1 \neq RL2$  then  $CC1 \leftarrow \text{TRUE}$  else  $CC1 \leftarrow \text{FALSE}$

**Cycle 2:** If  $X1 \neq RL3$  then  $CC2 \leftarrow \text{TRUE}$  else  $CC2 \leftarrow \text{FALSE}$   
If relational (op1) then  
    If  $PCW:0 = \text{AND}$  then  $CCi \leftarrow CC1 \text{ and } CC2$  else  $CCi \leftarrow CC1 \text{ or } CC2$   
else  $CCi \leftarrow CC2$   
If  $CCi$  was assigned **TRUE** then  $R0 \leftarrow X1$

## **Integer Arithmetic and Logical Instructions**

### **5.4.14 op = <=**

**Cycle 1:**  $X1 \leftarrow R1$   
    **if**  $R1 \leq RL2$  **then**  $CC1 \leftarrow \text{TRUE}$  **else**  $CC1 \leftarrow \text{FALSE}$

**Cycle 2:** **if**  $X1 \leq RL3$  **then**  $CC2 \leftarrow \text{TRUE}$  **else**  $CC2 \leftarrow \text{FALSE}$   
    **if** relational (op1) **then**  
        **if**  $PCW:0 = \text{AND}$  **then**  $CCi \leftarrow CC1 \text{ and } CC2$  **else**  $CCi \leftarrow CC1 \text{ or } CC2$   
    **else**  $CCi \leftarrow CC2$   
    **if**  $CCi$  was assigned **TRUE** **then**  $R0 \leftarrow X1$

## **Integer Arithmetic and Logical Instructions**

### **5.4.15 op = >=**

**Cycle 1:**  $X1 \leftarrow R1$   
If  $R1 \geq RL2$  then  $CC1 \leftarrow \text{TRUE}$  else  $CC1 \leftarrow \text{FALSE}$

**Cycle 2:** If  $X1 \geq RL3$  then  $CC2 \leftarrow \text{TRUE}$  else  $CC2 \leftarrow \text{FALSE}$   
If relational (op1) then  
If  $PCW:0 = \text{AND}$  then  $CCi \leftarrow CC1$  and  $CC2$  else  $CCi \leftarrow CC1$  or  $CC2$   
else  $CCi \leftarrow CC2$   
If  $CCi$  was assigned **TRUE** then  $R0 \leftarrow X1$

## **Integer Arithmetic and Logical Instructions**

### **5.4.16 op = >**

**Cycle 1:**  $X1 \leftarrow R1$

**If**  $R1 > RL2$  **then**  $CC1 \leftarrow \text{TRUE}$  **else**  $CC1 \leftarrow \text{FALSE}$

**Cycle 2:** **If**  $X1 > RL3$  **then**  $CC2 \leftarrow \text{TRUE}$  **else**  $CC2 \leftarrow \text{FALSE}$

**If** relational (op1) **then**

**If**  $PCW:0 = \text{AND}$  **then**  $CCi \leftarrow CC1$  **and**  $CC2$  **else**  $CCi \leftarrow CC1$  **or**  $CC2$

**else**  $CCi \leftarrow CC2$

**If**  $CCi$  was assigned **TRUE** **then**  $R0 \leftarrow X1$

## 5.5 Floating Point Instructions

Mnemonic: flt R0 := (R1 op1 R2) op2 R3

Format:

0	1	2	3	4	7	8	11	12	16	17	21	22	26	27	31
1	1	0	0	OP1	OP2	R0	R1	R2	R3						

<u>symbol</u>	<u>OP1, OP2</u>	<u>encoding</u>	<u>operation</u>
+	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

Description: The operation op1 is performed with registers R1 and R2 as operands during cycle 1. If no exception conditions are generated, operation op2 is performed with the result of op1 and register R3 as operands during cycle 2. If no exception conditions are generated, the result is written to R0.

Cycle Description:

**Cycle 1:** Cycle 1 for op1

**Cycle 2:** Cycle 2 for op2

## **Floating Point Instructions**

### **5.5.1 op = <**

**Cycle 1:**  $Y1 \leftarrow R1$   
If  $R1 < R2$  then  $CC1 \leftarrow \text{TRUE}$  else  $CC1 \leftarrow \text{FALSE}$

**Cycle 2:** If  $Y1 < R3$  then  $CC2 \leftarrow \text{TRUE}$  else  $CC2 \leftarrow \text{FALSE}$   
If relational (op1) then  
If  $PCW:0 = \text{AND}$  then  $CCf \leftarrow CC1 \text{ and } CC2$  else  $CCf \leftarrow CC1 \text{ or } CC2$   
else  $CCf \leftarrow CC2$   
If  $CCf$  was assigned **TRUE** then  $R0 \leftarrow Y1$

## **Floating Point Instructions**

### **5.5.2 op = +**

**Cycle 1:**  $Y1 \leftarrow R1 + R2$

**If overflow as defined by IEEE Std. 754 then PSW:8  $\leftarrow$  1**

**CC1  $\leftarrow$  NOT EVAL**

**Cycle 2:**  $Y2 \leftarrow Y1 + R3$

**CC2  $\leftarrow$  NOT EVAL**

**If overflow as defined by the IEEE Std 754 then**

**if CC1  $\neq$  FALSE then PSW:8  $\leftarrow$  1**

**else**

**if CC1  $\neq$  FALSE then R0  $\leftarrow$  Y2**

**if relational (op1) then CCf  $\leftarrow$  CC1**

## **Floating Point Instructions**

### **5.5.3 op = -**

**Cycle 1:**  $Y1 \leftarrow R1 - R2$

if underflow as defined by the IEEE Std. 754 then  $PSW:9 \leftarrow 1$   
 $CC1 \leftarrow \text{NOT EVAL}$

**Cycle 2:**  $Y2 \leftarrow Y1 - R3$

$CC2 \leftarrow \text{NOT EVAL}$

if underflow as defined by the IEEE Std 754 then

if  $CC1 \neq \text{FALSE}$  then  $PSW:9 \leftarrow 1$

else

if  $CC1 \neq \text{FALSE}$  then  $R0 \leftarrow Y2$

if relational (op1) then  $CCf \leftarrow CC1$



## Floating Point Instructions

### 5.5.4 op = '-'

**Cycle 1:**  $Y1 \leftarrow R2 - R1$

If underflow as defined by IEEE Std. 754 then PSW:9  $\leftarrow$  1  
CC1  $\leftarrow$  NOT EVAL

**Cycle 2:**  $Y2 \leftarrow R3 - Y1$

CC2  $\leftarrow$  NOT EVAL

If underflow as defined by the IEEE Std 754 then

If CC1  $\neq$  FALSE then PSW:9  $\leftarrow$  1

else

If CC1  $\neq$  FALSE then R0  $\leftarrow$  Y2

If relational (op1) then CCf  $\leftarrow$  CC1

## Floating Point Instructions

### 5.5.5 op = \*

**Cycle 1:**  $Y1 \leftarrow R1 * R2$   
If overflow as defined by IEEE Std. 754 then  $PSW:8 \leftarrow 1$   
 $CC1 \leftarrow \text{NOT EVAL}$

**Cycle 2:**  $Y2 \leftarrow Y1 * R3$   
 $CC2 \leftarrow \text{NOT EVAL}$   
If overflow as defined by the IEEE Std 754 then  
    if  $CC1 \neq \text{FALSE}$  then  $PSW:8 \leftarrow 1$   
**else**  
    if  $CC1 \neq \text{FALSE}$  then  $R0 \leftarrow Y2$   
    if relational (op1) then  $CCf \leftarrow CC1$

## Floating Point Instructions

### 5.5.6 op = /

**Cycle 1:** If R2 = 0 then PSW:5  $\leftarrow$  1 **else** Y1  $\leftarrow$  R1 / R2  
If underflow as defined by the Std. 754 then PSW:9  $\leftarrow$  1  
CC1  $\leftarrow$  NOT EVAL

**Cycle 2:** If R3 = 0 then  
If CC1  $\neq$  FALSE then PSW:5  $\leftarrow$  1  
**else**  
Y2  $\leftarrow$  Y1 / R3  
CC2  $\leftarrow$  NOT EVAL  
If underflow as defined by the IEEE Std 754 then  
If CC1  $\neq$  FALSE then PSW:9  $\leftarrow$  1  
**else**  
If CC1  $\neq$  FALSE then R0  $\leftarrow$  Y2  
If relational (op1) then CCf  $\leftarrow$  CC1

## Floating Point Instructions

### 5.5.7 op = '/'

**Cycle 1:** If  $R1 = 0$  then  $PSW:5 \leftarrow 1$  else  $Y1 \leftarrow R2 / R1$   
 $CC1 \leftarrow \text{NOT EVAL}$

**Cycle 2:** If  $Y1 = 0$  then  
    If  $CC1 \neq \text{FALSE}$  then  $PSW:5 \leftarrow 1$   
else  
     $Y2 \leftarrow R3 / Y1$   
     $CC2 \leftarrow \text{NOT EVAL}$   
    If underflow as defined by the IEEE Std 754 then  
        If  $CC1 \neq \text{FALSE}$  then  $PSW:9 \leftarrow 1$   
    else  
        If  $CC1 \neq \text{FALSE}$  then  $R0 \leftarrow Y2$   
        If relational (op1) then  $CCf \leftarrow CC1$

## **Floating Point Instructions**

### **5.5.8 op = nop**

**Cycle 1:**  $Y1 \leftarrow R1$   
 $CC1 \leftarrow \text{NOT EVAL}$

**Cycle 2:**  $Y2 \leftarrow Y1$   
 $CC2 \leftarrow \text{NOT EVAL}$   
If  $CC1 \neq \text{FALSE}$  then  $R0 \leftarrow Y2$   
If relational (op1) then  $CCf \leftarrow CC1$

## **Floating Point Instructions**

### **5.5.9 op = nop'**

**Cycle 1:**  $Y1 \leftarrow R2$   
 $CC1 \leftarrow \text{NOT EVAL}$

**Cycle 2:**  $Y2 \leftarrow R3$   
 $CC2 \leftarrow \text{NOT EVAL}$   
**if**  $CC1 \neq \text{FALSE}$  **then**  $R0 \leftarrow Y2$   
**if** relational (op1) **then**  $CCf \leftarrow CC1$

## **Floating Point Instructions**

### **5.5.10 op = =**

**Cycle 1:**  $Y1 \leftarrow R1$   
If  $R1 = R2$  then  $CC1 \leftarrow \text{TRUE}$  else  $CC1 \leftarrow \text{FALSE}$

**Cycle 2:** If  $Y1 = R3$  then  $CC2 \leftarrow \text{TRUE}$  else  $CC2 \leftarrow \text{FALSE}$   
If relational (op1) then  
    If  $PCW:0 = \text{AND}$  then  $CCf \leftarrow CC1$  and  $CC2$  else  $CCf \leftarrow CC1$  or  $CC2$   
else  $CCf \leftarrow CC2$   
If  $CCf$  was assigned **TRUE** then  $R0 \leftarrow Y1$

## **Floating Point Instructions**

**5.5.11 op = <>**

**Cycle 1:**  $Y1 \leftarrow R1$   
If  $R1 \neq R2$  then  $CC1 \leftarrow \text{TRUE}$  else  $CC1 \leftarrow \text{FALSE}$

**Cycle 2:** if  $Y1 \neq R3$  then  $CC2 \leftarrow \text{TRUE}$  else  $CC2 \leftarrow \text{FALSE}$   
If relational (op1) then  
    If  $PCW:0 = \text{AND}$  then  $CCf \leftarrow CC1 \text{ and } CC2$  else  $CCf \leftarrow CC1 \text{ or } CC2$   
else  $CCf \leftarrow CC2$   
If  $CCf$  was assigned **TRUE** then  $R0 \leftarrow Y1$



## **Floating Point Instructions**

### **5.5.12 op = <=**

**Cycle 1:**  $Y1 \leftarrow R1$   
If  $R1 \leq R2$  then  $CC1 \leftarrow \text{TRUE}$  else  $CC1 \leftarrow \text{FALSE}$

**Cycle 2:** If  $Y1 \leq R3$  then  $CC2 \leftarrow \text{TRUE}$  else  $CC2 \leftarrow \text{FALSE}$   
If relational (op1) then  
    If  $PCW:0 = \text{AND}$  then  $CCf \leftarrow CC1$  and  $CC2$  else  $CCf \leftarrow CC1$  or  $CC2$   
else  $CCf \leftarrow CC2$   
If  $CCf$  was assigned **TRUE** then  $R0 \leftarrow Y1$

## **Floating Point Instructions**

### **5.5.13 op = >**

**Cycle 1:**  $Y1 \leftarrow R1$   
    **if**  $R1 > R2$  **then**  $CC1 \leftarrow \text{TRUE}$  **else**  $CC1 \leftarrow \text{FALSE}$

**Cycle 2:** **if**  $Y1 > R3$  **then**  $CC2 \leftarrow \text{TRUE}$  **else**  $CC2 \leftarrow \text{FALSE}$   
    **if** relational (op1) **then**  
        **if**  $PCW:0 = \text{AND}$  **then**  $CCf \leftarrow CC1 \text{ and } CC2$  **else**  $CCf \leftarrow CC1 \text{ or } CC2$   
    **else**  $CCf \leftarrow CC2$   
    **if**  $CCf$  was assigned **TRUE** **then**  $R0 \leftarrow Y1$

## **Floating Point Instructions**

### **5.5.14 op = >=**

**Cycle 1:**  $Y1 \leftarrow R1$   
If  $R1 \geq R2$  then  $CC1 \leftarrow \text{TRUE}$  else  $CC1 \leftarrow \text{FALSE}$

**Cycle 2:** If  $Y1 \geq R3$  then  $CC2 \leftarrow \text{TRUE}$  else  $CC2 \leftarrow \text{FALSE}$   
If relational (op1) then  
    If  $PCW:0 = \text{AND}$  then  $CCf \leftarrow CC1 \text{ and } CC2$  else  $CCf \leftarrow CC1 \text{ or } CC2$   
else  $CCf \leftarrow CC2$   
If  $CCf$  was assigned **TRUE** then  $R0 \leftarrow Y1$

## 5.6 Vector Instructions: Integer, Logical and Floating Point

**Mnemonic:** vec R0 := (R1 op R2) if R3  
vec R0 := R1 op R2

**Format:**

0	1	2	3	4	11	12	16	17	21	22	26	27	31
1	1	0	1	OP				R0	R1	R2	R3		

OP Encoding Table

	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	fneg	fnegC		
1111	igtr	igtrC	fgtr	fgtrC		

**Description:** These are integer, logical and floating point operations on "blocks" of N y-bit items, where N is an implementation defined parameter. The operation op is performed with registers R1 and R2 as operands. For a conditional vector instruction, if no exception condition is generated and if R3 ≠ 0, the result is assigned to R0. For a vector instruction with no conditional, if no exception conditions are generated, the result is assigned to R0. Conceptually, all N component operations are performed simultaneously.

**Cycle Description:**

**Cycle 1:** Cycle 1 for op

**Cycle 2:** Cycle 2 for op

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.1 op = ladd

**Cycle 1:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$   
     $Z1.i.value \leftarrow R1.i.value + R2.i.value$   
    if overflow then  $PSW:14 \leftarrow 1$

**Cycle 2:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$   
     $R0.i.value, tag \leftarrow Z1.i.value, \text{CHANGED}$

## **Vector Instructions: Integer, Logical and Floating Point**

### **5.6.2 op = isub**

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
     $Z1.i.value \leftarrow R1.i.value - R2.i.value$   
    **If underflow then PSW:14  $\leftarrow 1$**

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
     $R0.i.value, \text{tag} \leftarrow Z1.i.value, \text{CHANGED}$

## **Vector Instructions: Integer, Logical and Floating Point**

### **5.6.3 op = imul**

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
     $Z1.i.value \leftarrow R1.i.value * R2.i.value$   
    **if overflow then** PSW:14  $\leftarrow 1$

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
    R0.i.value, tag  $\leftarrow Z1.i.value$ , **CHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.4 op = idiv

**Cycle 1:**  $\forall i$  in  $\{0,1,\dots,N - 1\}$   
    **if** R2.i.value = 0 **then** PSW:14  $\leftarrow$  1 **else**  
        Z1.i.value  $\leftarrow$  R1.i.value / R2.i.value  
    **if** underflow **then** PSW:14  $\leftarrow$  1

**Cycle 2:**  $\forall i$  in  $\{0,1,\dots,N - 1\}$   
    R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**



## Vector Instructions: Integer, Logical and Floating Point

### 5.6.5 op = lasl

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N-1\}$   
Z1.i.value  $\leftarrow$  R1.i.value asl R2.i.value

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N-1\}$   
R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.6 op = leql

**Cycle 1:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

if  $R1.i.value = R2.i.value$  then  $Z1.i.value \leftarrow \text{TRUE}$  else  $Z1.i.value \leftarrow \text{FALSE}$

**Cycle 2:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

$R0.i.value, \text{tag} \leftarrow Z1.i.value, \text{CHANGED}$

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.7 op = ineq

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N-1\}$

**If** R1.i.value  $\neq$  R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE** **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N-1\}$

    R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.8 op = igtr

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
If  $R1.i.value > R2.i.value$  then  $Z1.i.value \leftarrow \text{TRUE}$  else  $Z1.i.value \leftarrow \text{FALSE}$

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
 $R0.i.value, \text{tag} \leftarrow Z1.i.value, \text{CHANGED}$

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.9 op = lgeq

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**If**  $R1.i.value \geq R2.i.value$  **then**  $Z1.i.value \leftarrow \text{TRUE}$  **else**  $Z1.i.value \leftarrow \text{FALSE}$

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

$R0.i.value, \text{tag} \leftarrow Z1.i.value, \text{CHANGED}$

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.10 op = ilss

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**If** R1.i.value < R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE** **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

    R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.11 op = ileq

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
If  $R1.i.value \leq R2.i.value$  then  $Z1.i.value \leftarrow \text{TRUE}$  else  $Z1.i.value \leftarrow \text{FALSE}$

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
 $R0.i.value, \text{tag} \leftarrow Z1.i.value, \text{CHANGED}$

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.12 op = iaddC

**Cycle 1:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

$Z1.i.value \leftarrow R1.i.value + R2.i.value$

If overflow then  $PSW:14 \leftarrow 1$

**Cycle 2:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

If  $R3.i.value \neq \text{FALSE}$  then  $R0.i.value, \text{tag} \leftarrow Z1.i.value, \text{CHANGED}$

else  $R0.i.tag \leftarrow \text{UNCHANGED}$



## Vector Instructions: Integer, Logical and Floating Point

### 5.6.13 op = landC

**Cycle 1:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$   
Z1.i.value  $\leftarrow$  R1.i.value && R2.i.value

**Cycle 2:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$   
If R3.i.value  $\neq$  FALSE then R0.i.value, tag  $\leftarrow$  Z1.i.value, CHANGED  
else R0.i.tag  $\leftarrow$  UNCHANGED

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.14 op = lorC

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N-1\}$   
Z1.i.value  $\leftarrow$  R1.i.value || R2.i.value

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N-1\}$   
If R3.i.value  $\neq$  FALSE then R0.i.value, tag  $\leftarrow$  Z1.i.value, CHANGED  
else R0.i.tag  $\leftarrow$  UNCHANGED

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.15 op = leqvC

**Cycle 1:**  $\forall i$  in  $\{0,1,\dots,N - 1\}$   
Z1.i.value  $\leftarrow$  R1.i.value EQV R2.i.value

**Cycle 2:**  $\forall i$  in  $\{0,1,\dots,N - 1\}$   
If R3.i.value  $\neq$  **FALSE** then R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**  
else R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.16 op = lsubC

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

$Z1.i.value \leftarrow R1.i.value - R2.i.value$

**if** underflow **then** PSW:14  $\leftarrow 1$

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**if** R3.i.value  $\neq$  **FALSE** **then** R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**

**else** R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.17 op = imulC

**Cycle 1:**  $\forall i$  in  $\{0,1,\dots,N - 1\}$

$Z1.i.value \leftarrow R1.i.value * R2.i.value$

If overflow then PSW:14  $\leftarrow$  1

**Cycle 2:**  $\forall i$  in  $\{0,1,\dots,N - 1\}$

If R3.i.value  $\neq$  **FALSE** then R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**

else R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.18 op = idivC

**Cycle 1:**  $\forall i$  in  $\{0,1,\dots,N-1\}$

**if** R2.i.value = 0 **then** PSW:14  $\leftarrow$  1 **else**

        Z1.i.value  $\leftarrow$  R1.i.value / R2.i.value

**if** underflow **then** PSW:14  $\leftarrow$  1

**Cycle 2:**  $\forall i$  in  $\{0,1,\dots,N-1\}$

**if** R3.i.value  $\neq$  **FALSE** **then** R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**

**else** R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.19 op = lslC

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
Z1.i.value  $\leftarrow$  R1.i.value asl R2.i.value

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
If R3.i.value  $\neq$  **FALSE** then R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**  
else R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.20 op = leqIC

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**If** R1.i.value = R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE** **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**If** R3.i.value  $\neq$  **FALSE** **then** R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**  
    **else** R0.i.tag  $\leftarrow$  **UNCHANGED**



## Vector Instructions: Integer, Logical and Floating Point

### 5.6.21 op = ineqC

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N-1\}$

    If  $R1.i.value \neq R2.i.value$  then  $Z1.i.value \leftarrow \text{TRUE}$  else  $Z1.i.value \leftarrow \text{FALSE}$

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N-1\}$

    If  $R3.i.value \neq \text{FALSE}$  then  $R0.i.value, \text{tag} \leftarrow Z1.i.value, \text{CHANGED}$   
    else  $R0.i.tag \leftarrow \text{UNCHANGED}$

## **Vector Instructions: Integer, Logical and Floating Point**

### **5.6.22 op = lgtrC**

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**if** R1.i.value > R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE** **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**if** R3.i.value  $\neq$  **FALSE** **then** R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**  
    **else** R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.23 op = lgeqC

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**If** R1.i.value  $\geq$  R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE** **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**If** R3.i.value  $\neq$  **FALSE** **then** R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**  
    **else** R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.24 op = ilssC

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**If** R1.i.value < R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE** **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**If** R3.i.value  $\neq$  **FALSE** **then** R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**  
    **else** R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

5.6.25 op = lleqC

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N-1\}$

**If** R1.i.value  $\leq$  R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE** **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N-1\}$

**If** R3.i.value  $\neq$  **FALSE** **then** R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**  
    **else** R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.26 op = faddC

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

$Z1.i.value \leftarrow R1.i.value + R2.i.value$

If overflow as defined by IEEE Std. 754 then PSW:14  $\leftarrow 1$

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

If R3.i.value  $\neq 0$  then R0.i.value, tag  $\leftarrow Z1.i.value$ , **CHANGED**

else R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.27 op = fsubC

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

$Z1.i.value \leftarrow R1.i.value - R2.i.value$

**if** underflow as defined by IEEE Std. 754 **then** PSW:14  $\leftarrow 1$

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**if** R3.i.value  $\neq 0$  **then** R0.i.value, tag  $\leftarrow Z1.i.value$ , **CHANGED**

**else** R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.28 op = fmulC

**Cycle 1:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

$Z1.i.value \leftarrow R1.i.value * R2.i.value$

If overflow as defined by IEEE Std. 754 then PSW:14  $\leftarrow 1$

**Cycle 2:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

If R3.i.value  $\neq 0$  then R0.i.value, tag  $\leftarrow Z1.i.value$ , **CHANGED**

else R0.i.tag  $\leftarrow$  **UNCHANGED**



## Vector Instructions: Integer, Logical and Floating Point

### 5.6.29 op = fdivC

**Cycle 1:**  $\forall i$  in  $\{0,1,\dots,N-1\}$

**If** R2.i.value = 0 **then** PSW:14  $\leftarrow$  1 **else**

        Z1.i.value  $\leftarrow$  R1.i.value / R2.i.value

**If** underflow as defined by IEEE Std. 754 **then** PSW:14  $\leftarrow$  1

**Cycle 2:**  $\forall i$  in  $\{0,1,\dots,N-1\}$

**If** R3.i.value  $\neq$  0 **then** R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**

**else** R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.30 op = fleqC

**Cycle 1:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

**if** R1.i.value  $\leq$  R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE**  
    **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

**if** R3.i.value  $\neq 0$  **then** R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**  
    **else** R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

5.6.31 op = flssC

**Cycle 1:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

**If** R1.i.value < R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE**  
    **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

**If** R3.i.value  $\neq$  0 **then** R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**  
    **else** R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

5.6.32 op = fgeqC

**Cycle 1:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

**if** R1.i.value  $\geq$  R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE**  
    **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

**if** R3.i.value  $\neq$  0 **then** R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**  
    **else** R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.33 op = fgtrC

**Cycle 1:**  $\forall i$  in  $\{0,1,\dots,N - 1\}$

**If** R1.i.value > R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE**  
    **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i$  in  $\{0,1,\dots,N - 1\}$

**If** R3.i.value  $\neq$  0 **then** R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**  
    **else** R0.i.tag  $\leftarrow$  **UNCHANGED**

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.34 op = fneqC

**Cycle 1:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

    If  $R1.i.value \neq R2.i.value$  then  $Z1.i.value \leftarrow \text{TRUE}$   
    else  $Z1.i.value \leftarrow \text{FALSE}$

**Cycle 2:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

    If  $R3.i.value \neq 0$  then  $R0.i.value, \text{tag} \leftarrow Z1.i.value, \text{CHANGED}$   
    else  $R0.i.tag \leftarrow \text{UNCHANGED}$

## Vector Instructions: Integer, Logical and Floating Point

5.6.35 op = feqlC

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**if** R1.i.value = R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE**  
    **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**if** R3.i.value  $\neq$  0 **then** R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**  
    **else** R0.i.tag  $\leftarrow$  **UNCHANGED**

## **Vector Instructions: Integer, Logical and Floating Point**

### **5.6.36 op = fadd**

**Cycle 1:**  $\forall i$  in  $\{0,1,\dots,N - 1\}$

$Z1.i.value \leftarrow R1.i.value + R2.i.value$

**if** overflow as defined by IEEE Std. 754 **then** PSW:14  $\leftarrow 1$

**Cycle 2:**  $\forall i$  in  $\{0,1,\dots,N - 1\}$

$R0.i.value, tag \leftarrow Z1.i.value, \text{CHANGED}$



## **Vector Instructions: Integer, Logical and Floating Point**

### **5.6.37 op = fsub**

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

$Z1.i.value \leftarrow R1.i.value - R2.i.value$

**If** underflow as defined by IEEE Std. 754 **then** PSW:14  $\leftarrow 1$

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

$R0.i.value, \text{tag} \leftarrow Z1.i.value, \text{CHANGED}$

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.38 op = fmul

**Cycle 1:**  $\forall i$  in  $\{0,1,\dots,N - 1\}$   
     $Z1.i.value \leftarrow R1.i.value * R2.i.value$   
    If overflow as defined by IEEE Std. 754 then  $PSW:14 \leftarrow 1$

**Cycle 2:**  $\forall i$  in  $\{0,1,\dots,N - 1\}$   
     $R0.i.value, tag \leftarrow Z1.i.value, \text{CHANGED}$

## Vector Instructions: Integer, Logical and Floating Point

### 5.6.39 op = fdiv

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

**If** R2.i.value = 0 **then** PSW:14  $\leftarrow$  1 **else**

        Z1.i.value  $\leftarrow$  R1.i.value / R2.i.value

**If** underflow as defined by IEEE Std. 754 **then** PSW:14  $\leftarrow$  1

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

    R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**

## Vector Instructions: Integer, Logical and Floating Point

5.6.40 op = feql

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
if R1.i.value = R2.i.value then Z1.i.value  $\leftarrow$  TRUE  
else Z1.i.value  $\leftarrow$  FALSE

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
R0.i.value, tag  $\leftarrow$  Z1.i.value, CHANGED

## **Vector Instructions: Integer, Logical and Floating Point**

**5.6.41 op = fneq**

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
    **if** R1.i.value  $\neq$  R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE**  
    **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
    R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**

## **Vector Instructions: Integer, Logical and Floating Point**

**5.6.42 op = fgtr**

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
    **If** R1.i.value > R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE**  
    **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
    R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**

## **Vector Instructions: Integer, Logical and Floating Point**

### **5.6.43 op = fgeq**

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
    **if** R1.i.value  $\geq$  R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE**  
    **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
    R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**

## **Vector Instructions: Integer, Logical and Floating Point**

### **5.6.44 op = flss**

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
    **if** R1.i.value < R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE**  
    **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
    R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**



## **Vector Instructions: Integer, Logical and Floating Point**

**5.6.45 op = fleq**

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
    **If** R1.i.value  $\leq$  R2.i.value **then** Z1.i.value  $\leftarrow$  **TRUE**  
    **else** Z1.i.value  $\leftarrow$  **FALSE**

**Cycle 2:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$   
    R0.i.value, tag  $\leftarrow$  Z1.i.value, **CHANGED**

## 5.7 Load and Store Instructions

Mnemonic: LSOP R0 := (R1 op1 RL2) op2 RL3

Format:

0	1	2	3	4	7	8	11	12	16	17	21	22	26	27	31
1	0	RL	LSOP	OP1	OP2	R0	R1	RL2	RL3						

Description: The LOAD and STORE instructions specify : (1) the address of the data to be read or written, and (2) the size/type of the data (e.g., byte vs. halfword vs. double-precision floating point). The type specified implicitly determines the execution unit involved.

- (1) The address computation is identical to the integer/logical instructions.
- (2) The type/size of the data to be read or written is specified by the LOAD or STORE instruction.

Cycle Description:

### Load Instructions

**Cycle 1:** Cycle 1 for op1

**Cycle 2:** Cycle 2 for op2

**Memory Cycle:** with input FIFO specified by LSOP **do**  
    If FIFO.consume\_count = 0 then FIFO ← sign\_extend (M[X2, size])  
    **else exception::** Load while input streaming

### Store Instructions

**Cycle 1:** Cycle 1 for op1

**Cycle 2:** Cycle 2 for op2

**Memory Cycle:** with output FIFO specified by LSOP **do**  
    If FIFO.consume\_count = 0 then  
        If qualifier\_type (FIFO) = DATA then M[X2, size] ← FIFO  
        **else** FIFO.value, qualifier, size ← X2, ADDR, size  
    **else exception::** Load while output streaming

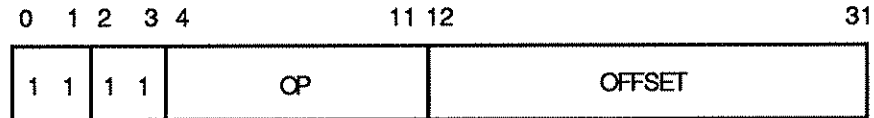
<u>LSOP</u>	<u>FIFO</u>	<u>data_size/type</u>	<u>exec</u> <u>unit</u>	<u>sign</u> <u>extension</u>	<u>LSOP</u> <u>Encoding</u>
<b>Load</b>					
L8i	r0.input	8-bit integer	IEU	no	0000
L8ix	r0.input	8-bit integer	IEU	yes	0100
L16i	r0.input	16-bit integer	IEU	no	0001
L16ix	r0.input	16-bit integer	IEU	yes	0101
L32i	r0.input	32-bit integer	IEU	no	0010
L32ix	r0.input	32-bit integer	IEU	yes	0110
L64i	r0.input	64-bit integer	IEU	n/a	0011
L32f	f0.input	32-bit floating	FEU	n/a	0010
L64f	f0.input	64-bit floating	FEU	n/a	1101
<b>Store</b>					
S8i	ro.output	8-bit integer	IEU	n/a	1000
S16i	ro.output	16-bit integer	IEU	n/a	1001
S32i	ro.output	32-bit integer	IEU	n/a	1010
S64i	ro.output	64-bit integer	IEU	n/a	1011
S32f	fo.output	32-bit floating	FEU	n/a	1110
S64f	fo.output	64-bit floating	FEU	n/a	1111

There are 15 LSOP operations; the other opcode is illegal and will produce an illegal instruction trap if used.

## 5.8 Control Flow Instructions

Mnemonic: JumpOP offset

Format:



<u>JumpOP</u>	<u>OP</u>	<u>Description</u>
Jump	0000 0000:	unconditional Jump
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
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
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
Call	0010 0000:	subroutine Call
ECall	0010 0001:	Entry Call

Description: These instructions replace the Program Counter with a new value, the target address. In all but one case (ECall), 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. Conditional Jumps "consume" a condition bit generated by a relational operation. ECall is an implementation dependent instruction.

Cycle Description:

**Cycle 1:** JC ← **FALSE**  
    **case** JumpOP =  
        Jump: JC ← **TRUE**  
        JumpITn, JumpITy: **if** CCI **then** JC ← **TRUE**  
        JumpIFn, JumpIFy: **if not** CCI **then** JC ← **TRUE**  
        JumpFTn, JumpFTy: **if** CCf **then** JC ← **TRUE**  
        JumpFFn, JumpFFy: **if not** CCf **then** JC ← **TRUE**  
        JNI r0: **if** r0.input.consume\_count ≠ 0 **then** JC ← **TRUE**  
        JNI r1: **if** r1.input.consume\_count ≠ 0 **then** JC ← **TRUE**  
        JNO r0: **if** r0.output.consume\_count ≠ 0 **then** JC ← **TRUE**  
        JNO r1: **if** r1.output.consume\_count ≠ 0 **then** JC ← **TRUE**  
        JNI f0: **if** f0.input.consume\_count ≠ 0 **then** JC ← **TRUE**  
        JNI f1: **if** f1.input.consume\_count ≠ 0 **then** JC ← **TRUE**  
        JNO f0: **if** f0.output.consume\_count ≠ 0 **then** JC ← **TRUE**  
        JNO f1: **if** f1.output.consume\_count ≠ 0 **then** JC ← **TRUE**  
        JNI v0: **if** v0.input.consume\_count ≠ 0 **then** JC ← **TRUE**  
        JNI v1: **if** v1.input.consume\_count ≠ 0 **then** JC ← **TRUE**  
        JNO v0: **if** v0.output.consume\_count ≠ 0 **then** JC ← **TRUE**  
        JNO v1: **if** v1.output.consume\_count ≠ 0 **then** JC ← **TRUE**  
        Call : r4 ← PC  
            JC ← **TRUE**  
        ECall : <Protected Stack Area> ← PTP  
            <Protected Stack Area> ← PC  
            PTP ← <Entry Page PTP Value>  
            JC ← **TRUE**  
    **end case**  
    **if** JC **then** PC ← PC + sign\_extend (offset) lsl 2

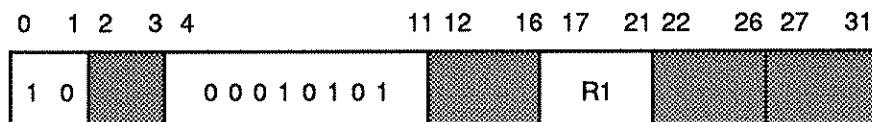
118

## Special Instructions

### 5.9.2 CallI

Mnemonic: CallI R1

Format:



Description: Call Indirect stores the current PC in register 4 and sets the PC to the target address from the register named in the R1 field

Cycle Description:

**Cycle 1:**  $r4 \leftarrow PC$

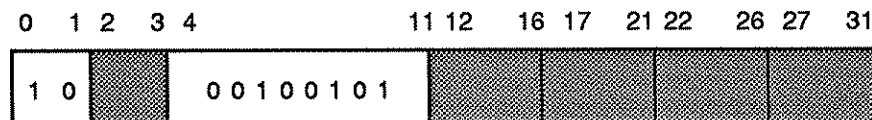
$PC \leftarrow R1$

## Special Instructions

### 5.9.3 EReturn

Mnemonic: EReturn

Format:



Description: EReturn (Entry Return) is the complementary instruction to ECall; it restores the protection table pointer and PC from the protected stack.

Cycle Description:

**Cycle 1:** PC  $\leftarrow$  <Protected Stack Area>  
PTP  $\leftarrow$  <Protected Stack Area>



## Special Instructions

### 5.9.4 Streaming to and from the IEU and FEU

Mnemonic: SOP R0, R1, RL2, RL3

Format:

0	1	2	3	4				11	12		16	17		21	22		26	27		31
1	0		RL				SOP				R0			R1			RL2			RL3

Description: Stream instructions read/write from/to FIFOs. 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 (RL2), a stride (RL3), and which FIFO to use (0 or 1); this last parameter is taken as the least significant bit of the R0 field.

There are five instructions to stop input or output streaming and flush the relevant FIFOs.

Cycle Description:

**Stream in:**

**Cycle 1:**   with input FIFO specified by R0 do  
              if FIFO.consume\_count = 0 then  
                  initiate stream operation :: R0, R1, RL2, RL3, stream\_in  
              else exception:: Double Stream

**Stream out:**

**Cycle 1:**   with output FIFO specified by R0 do  
              if FIFO.consume\_count = 0 then  
                  initiate stream operation :: R0, R1, RL2, RL3, stream\_out  
              else exception:: Double Stream

**Stop Streaming:**

**Cycle 1:**   initiate stream operation ::  $\phi$ ,  $\phi$ ,  $\phi$ ,  $\phi$ , stop\_streaming

## Stream Operations

**Initiate stream operation :: R0, R1, RL2, RL3, stream\_in**

### Memory Cycle:

```
with input FIFO specified by R0 do
  FIFO.consume_count ← RL2
  SM ← R1
  Scount ← RL2
  If Scount > 0 then SDec ← TRUE else SDec ← FALSE
  while Scount ≠ 0
    FIFO ← sign_extend (M[SM, size])
    SM ← SM + RL3
    If SDec then Scount ← Scount - 1
```

**Initiate stream operation :: R0, R1, RL2, RL3, stream\_out**

### Memory Cycle:

```
with output FIFO specified by R0 do
  FIFO.consume_count ← RL2
  SM ← R1
  Scount ← RL2
  If Scount > 0 then SDec ← TRUE else SDec ← FALSE
  while Scount ≠ 0
    If qualifier_type (FIFO) = DATA then M[SM, size] ← FIFO
    else FIFO.value, qualifier, size ← SM, ADDR, size
    SM ← SM + RL3
    If SDec then Scount ← Scount - 1
```

**Initiate stream operation ::  $\phi$ ,  $\phi$ ,  $\phi$ ,  $\phi$ , stop\_streaming**

### Memory Cycle:

```
case op =
StopAll:
  with each input FIFO do
    FIFO.consume_count ← 0
    flush FIFO
  with each output FIFO do
    complete pending memory writes
    FIFO.consume_count ← 0
StopI, StopFI:
  with input FIFO specified by R0 do
    FIFO.consume_count ← 0
    flush FIFO
StopO, StopFO:
  with output FIFO specified by R0 do
    Complete pending memory writes
    FIFO.consume_count ← 0
```

<u>SOP</u>	<u>operation</u>	<u>data_type</u>	<u>exec</u> <u>unit</u>	<u>sign</u> <u>extension</u>	<u>SOP</u> <u>Encoding</u>
<b>Stream In</b>					
Sin8i	load	8-bit integer	IEU	no	00000000
Sin8ix	load	8-bit integer	IEU	yes	01000000
Sin16i	load	16-bit integer	IEU	no	00010000
Sin16ix	load	16-bit integer	IEU	yes	01010000
Sin32i	load	32-bit integer	IEU	no	00100000
Sin32ix	load	32-bit integer	IEU	yes	01100000
Sin64i	load	64-bit integer	IEU	n/a	00110000
Sin32f	load	32-bit floating	FEU	n/a	11000000
Sin64f	load	64-bit floating	FEU	n/a	11010000
<b>Stream out</b>					
Sout8i	store	8-bit integer	IEU	n/a	10000000
Sout16i	store	16-bit integer	IEU	n/a	10010000
Sout32i	store	32-bit integer	IEU	n/a	10100000
Sout64i	store	64-bit integer	IEU	n/a	10110000
Sout32f	store	32-bit floating	FEU	n/a	11100000
Sout64f	store	64-bit floating	FEU	n/a	11110000
<b>Stop streaming</b>					
StopAll	Stop all Streaming operations				01110010
StopII	Stop Integer Input Streaming operations on FIFO specified by R0				00000010
StopIO	Stop Integer Output Streaming operations on FIFO specified by R0				00010010
StopFI	Stop Floating Input Streaming operations on FIFO specified by R0				00100010
StopFO	Stop Floating Output Streaming operations on FIFO specified by R0				00110010
<b>FIFO selection</b>					
If data type = integer then					
if R0 = 1 then FIFO = r0 else FIFO = r1					
else If R0 = 1 then FIFO = f0 else FIFO = f1					

## Special Instructions

### 5.9.5 Streaming to and from the VEU

Mnemonic: VSOP R0, R1, RL2, RL3

Format:

0	1	2	3	4		11	12		16	17		21	22		26	27	31
1	0	RL			VSOP			R0			R1			RL2			RL3

Description: Stream instructions read/write from/to FIFOs. 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 (RL2), a stride (RL3), and the target FIFO. The FIFO is determined by the least significant bit of the R0 field.

There are three instructions to stop input or output streaming and flush the relevant FIFOs.

Cycle Description:

**Stream In:**

**Cycle 1:** with input FIFO specified by R0 do  
    if FIFO.consume\_count = 0 then  
        initiate vector stream operation :: R0, R1, RL2, RL3, stream\_in  
    else exception:: Double Stream

**Stream out:**

**Cycle 1:** with output FIFO specified by R0 do  
    if FIFO.consume\_count = 0 then  
        initiate vector stream operation :: R0, R1, RL2, RL3, stream\_out  
    else exception:: Double Stream

**Stop Streaming:**

**Cycle 1:** initiate vector stream operation ::  $\phi$ ,  $\phi$ ,  $\phi$ ,  $\phi$ , stop\_streaming

## Vector Stream Operations

**initiate vector stream operation:: R0, R1, RL2, RL3, stream\_in**

### Memory Cycle:

```
with input FIFO specified by R0 do
  FIFO.consume_count ← RL2
  SM ← R1
  Scount ← RL2
  If Scount > 0 then SDec ← TRUE else SDec ← FALSE
  while Scount ≠ 0
    FIFO ← sign_extend (M[SM, size])
    SM ← SM + RL3
    If SDec then Scount ← Scount - 1
```

**initiate vector stream operation:: R0, R1, RL2, RL3, stream\_out**

### Memory Cycle:

```
with output FIFO specified by R0 do
  FIFO.consume_count ← RL2
  SM ← R1
  Scount ← RL2
  If Scount > 0 then SDec ← TRUE else SDec ← FALSE
  while Scount ≠ 0
    If qualifer_type (FIFO) = DATA then
      SMV ← FIFO
      If SMV.tag = CHANGED then M[SM, size] ← SMV.value
    else FIFO.value, qualifier, size ← SM, ADDR, size
    SM ← SM + RL3
    If SDec then Scount ← Scount - 1
```

**initiate vector stream operation ::  $\phi$ ,  $\phi$ ,  $\phi$ ,  $\phi$ , stop\_streaming**

### Memory Cycle:

```
case op =
StopAll:
  with each input FIFO do
    FIFO.consume_count ← 0
    flush FIFO
  with each output FIFO do
    complete pending memory writes
    FIFO.consume_count ← 0
StopVI:
  with input FIFO specified by R0 do
    FIFO.consume_count ← 0
    flush input FIFO
StopVO:
  with output FIFO specified by R0 do
    Complete pending memory writes
    FIFO.consume_count ← 0
```

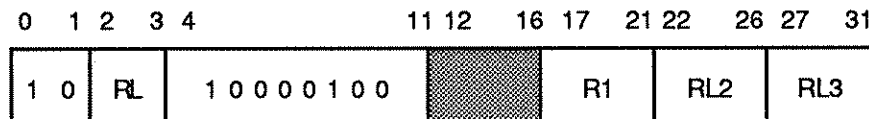
<u>VSOP</u>	<u>operation</u>	<u>data type</u>	<u>exec unit</u>	<u>sign extension</u>	<u>SOP Encoding</u>
<b>Stream In</b>					
VSin8i	load	8-bit integer	VEU	no	00000001
VSin8ix	load	8-bit integer	VEU	yes	01000001
VSin16i	load	16-bit integer	VEU	no	00010001
VSin16ix	load	16-bit integer	VEU	yes	01010001
VSin32i	load	32-bit integer	VEU	no	00100001
VSin32ix	load	32-bit integer	VEU	yes	01100001
VSin64i	load	64-bit integer	VEU	n/a	00110001
VSin32f	load	32-bit floating	VEU	n/a	11000001
VSin64f	load	64-bit floating	VEU	n/a	11010001
VSin1b	load	1-bit boolean	VEU	n/a	00010001
<b>Stream out</b>					
VSout8i	store	8-bit integer	VEU	n/a	10000000
VSout16i	store	16-bit integer	VEU	n/a	10010000
VSout32i	store	32-bit integer	VEU	n/a	10100000
VSout64i	store	64-bit integer	VEU	n/a	10110000
VSout32f	store	32-bit floating	VEU	n/a	11100000
VSout64f	store	64-bit floating	VEU	n/a	11110000
<b>Stop streaming</b>					
StopAll	Stop all Streaming operations				01110010
StopVI	Stop Vector Input Streaming operations on FIFO specified by R0				01000010
StopVO	Stop Vector Output Streaming operations on FIFO specified by R0				01010010

### Special instructions

### 5.9.6 ASSERT

Mnemonic: ASSERT ( $R1 \geq RL2$ )  $\leq RL3$

Format:



**Description:** The ASSERT instruction determines whether the value in integer register R1 is within the bounds specified by RL2 and RL3. If it is not, a hardware Assert Fault is generated.

Cycle Description:

**Cycle 1:** If not  $R1 \geq RL2$  then exception:: Assert Fault

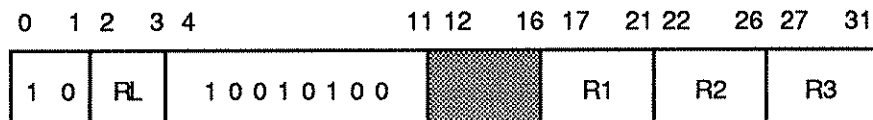
**Cycle 2:** if not  $R1 \leq RL3$  then exception:: Assert Fault

## Special Instructions

### 5.9.7 FASSERT

Mnemonic: FASSERT ( $R1 \geq R2$ )  $\leq R3$

Format:



Description: The FASSERT instruction determines whether the value in floating point register R1 is within the bounds specified by RL2 and RL3. If it is not, a hardware Assert Fault is generated.

Cycle Description:

**Cycle 1:** If not  $R1 \geq R2$  then exception:: Assert Fault

**Cycle 2:** If not  $R1 \leq R3$  then exception:: Assert Fault



## Special Instructions

### 5.9.8 FLDMOV

Mnemonic: FLDMOV R0 := R1, RL2, RL3

Format:

0	1	2	3	4		11	12		16	17		21	22		26	27	31
1 0		RL		1 1 0 0 0 1 0 0				R0		R1		RL2		RL3			

Description: The contents of the register specified by R1 is logically shifted left by RL2 bits, then logically shifted right by RL3 bits and the resulting value is assigned to the register specified by R0.

Cycle Description:

**Cycle 1:**  $X1 \leftarrow R1 \text{ lsl } RL2$

**Cycle 2:**  $R0 \leftarrow X1 \text{ lsr } RL3$

## Special Instructions

### 5.9.9 FLDMOVX

Mnemonic: FLDMOVX R0 := R1, RL2, RL3

Format:

0	1	2	3	4						11	12					16	17					21	22					26	27	31
1	0		RL		1	1	0	1	0	1	0	0		R0		R1		RL2					RL3							

Description: The contents of the register specified by R1 is logically shifted left by RL2 bits, then arithmetically shifted right by RL3 bits and the resulting value is assigned to the register specified by R0.

Cycle Description:

**Cycle 1:** X1  $\leftarrow$  R1 lsl RL2

**Cycle 2:** R0  $\leftarrow$  X1 asr RL3

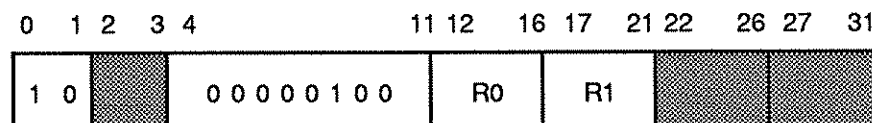


## Special Instructions

### 5.9.11 CVTIF

Mnemonic: CVTIF R0 := R1

Format:



Description: The CVTIF instruction ConVerTs from Integer to Floating. Data conversion is performed; an integer is converted to floating point representation.

R0 specifies a FEU register and R1 specifies an IEU register.

Cycle Description:

#### **Synch Cycle:**

Synchronize the IFU, the IEU, the FEU and the VEU. i.e. further execution of this instruction is delayed until all the instructions prior to the CVTIF instruction have been executed.

#### **Cycle 1:**

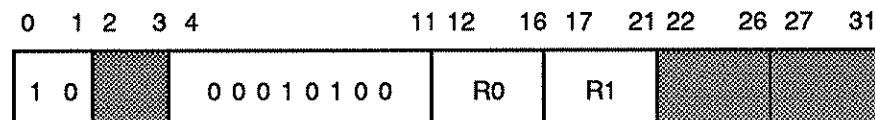
$R0 \leftarrow \text{int\_to\_float}(R1)$

## Special Instructions

### 5.9.12 CVTFI

Mnemonic: CVTFI R0 := R1

Format:



Description: The CVTFI instruction ConVerTs from Floating to Integer. Data conversion is performed; a floating point representation is converted to an integer.

R0 specifies an IEU register and R1 specifies a FEU register.

Cycle Description:

#### **Synch Cycle:**

Synchronize the IFU, the IEU, the FEU and the VEU. i.e. further execution of this instruction is delayed until all the instructions prior to the CVTFI instruction have been executed.

#### **Cycle 1:**

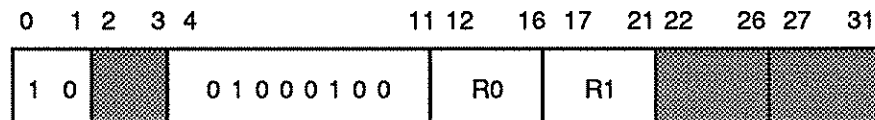
$R0 \leftarrow \text{float\_to\_int} (R1)$

## Special Instructions

### 5.9.13 TIF

Mnemonic: TIF R0 := R1

Format:



Description: TIF transfers an integer from the IEU to the FEU. TIF is a "bit copy" instruction, no data conversion is performed except as necessary to expand/contract the representation.

R1 specifies an IEU register and R0 specifies a FEU register.

Cycle Description:

#### **Synch Cycle:**

Synchronize the IFU, the IEU, the FEU and the VEU. i.e. further execution of this instruction is delayed until all the instructions prior to the TIF instruction have been executed.

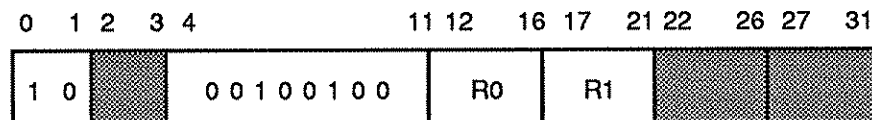
**Cycle 1:** R0  $\leftarrow$  R1

## Special Instructions

### 5.9.14 TFI

Mnemonic: TFI R0 := R1

Format:



Description: TFI transfers a floating point representation from the FEU to the IEU. TFI is a "bit copy" instruction, no data conversion is performed except as necessary to expand/contract the representation.

R0 specifies an IEU register and R1 specifies a FEU register.

Cycle Description:

#### **Synch Cycle:**

Synchronize the IFU, the IEU, the FEU and the VEU. i.e. further execution of this instruction is delayed until all the instructions prior to the TFI instruction have been executed.

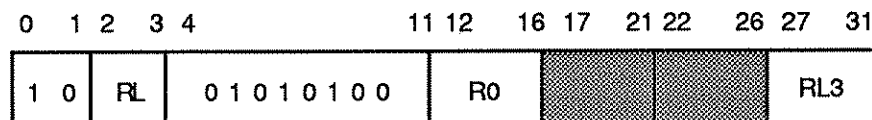
**Cycle 1:** R0 ← R1

## Special Instructions

### 5.9.15 TIV

Mnemonic: TIV R0 := RL3

Format:



Description: TIV transfers an integer from the IEU to the VEU. TIV is a "bit copy" instruction, no data conversion is performed except as necessary to expand/contract the representation. TIV zero extends as necessary.

R0 specifies an VEU register and RL3 specifies an IEU register. The instruction transfers N copies of the contents of the integer register specified by RL3.

Cycle Description:

#### **Synch Cycle:**

Synchronize the IFU, the IEU, the FEU and the VEU. i.e. further execution of this instruction is delayed until all the instructions prior to the TIV instruction have been executed.

**Cycle 1:**  $\forall i \text{ in } \{0,1,\dots,N - 1\}$

R0.i.value, tag  $\leftarrow$  RL3, **CHANGED**

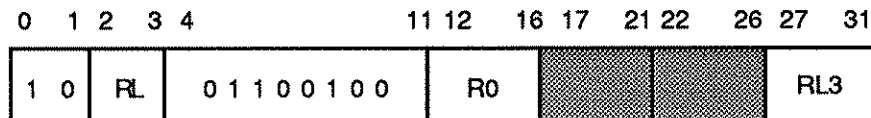


## Special Instructions

### 5.9.16 TIVx

Mnemonic: TIVx R0 := RL3

Format:



Description: TIVx transfers an integer from the IEU to the VEU. TIV is a "bit copy" instruction, no data conversion is performed except as necessary to expand/contract the representation. TIVx performs sign extension.

R0 specifies an VEU register and RL3 specifies an IEU register. The instruction transfers N copies of the contents of the integer register specified by RL3.

Cycle Description:

#### **Synch Cycle:**

Synchronize the IFU, the IEU, the FEU and the VEU. i.e. further execution of this instruction is delayed until all the instructions prior to the TIVx instruction have been executed.

**Cycle 1:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

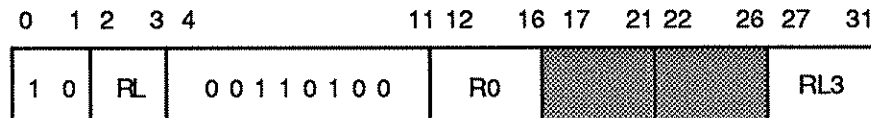
R0.i.value, tag  $\leftarrow$  sign\_extend (RL3), **CHANGED**

## Special Instructions

### 5.9.17 TFV

Mnemonic: TFV R0 := RL3

Format:



Description: TFV transfers an integer from the FEU to the VEU. TFV is a "bit copy" instruction, no data conversion is performed except as necessary to expand/contract the representation.

R0 specifies an VEU register and RL3 specifies a FEU register. The instruction transfers N copies of the contents of the floating point register specified by RL3.

Cycle Description:

#### **Synch Cycle:**

Synchronize the IFU, the IEU, the FEU and the VEU. i.e. further execution of this instruction is delayed until all the instructions prior to the TFV instruction have been executed.

**Cycle 1:**  $\forall i \text{ in } \{0, 1, \dots, N - 1\}$

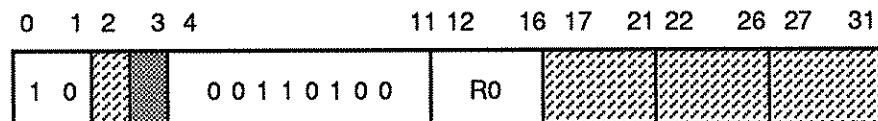
R0.i.value, tag  $\leftarrow$  RL3, **CHANGED**

## Special Instructions

### 5.9.18 LLH

Mnemonic: LLH R0 := 16\_bit\_constant

Format:



[shaded box] = 16 bit constant

Description: LLH assigns the specified 16 bit constant to the destination register. The 16 bit constant is formed by concatenating bit 2 with the R1, RL2 and RL3 fields.

Cycle Description:

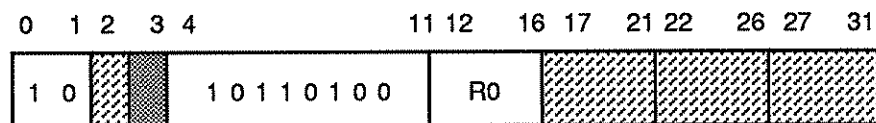
**Cycle 1:** R0  $\leftarrow$  16\_bit\_constant


## Special Instructions

### 5.9.19 SLL

Mnemonic: SLL R0 := 16\_bit\_constant

Format:



 = 16 bit constant

Description: 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. The 16 bit constant is formed by concatenating bit 2 with the R1, RL2 and RL3 fields.

Cycle Description:

**Cycle 1:**  $X1 \leftarrow R0 \ll 16$   
 $X1:16-31 \leftarrow 16\_bit\_constant$

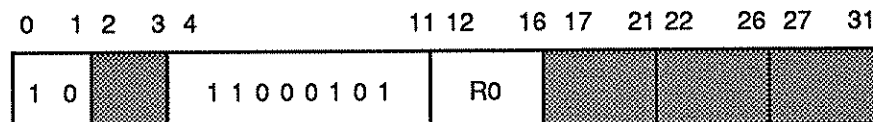
**Cycle 2:**  $R0 \leftarrow X1$

## Special Instructions

### 5.9.20 ReadPCW

Mnemonic: ReadPCW R0

Format:



Description: The contents of PCW are copied to the specified IEU register.

Cycle Description:

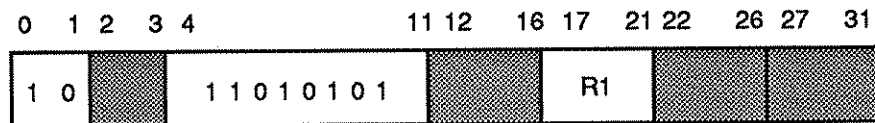
Cycle 1: R0  $\leftarrow$  PCW

## Special Instructions

### 5.9.21 WritePCW

Mnemonic: WritePCW R1

Format:



Description: The contents of the specified IEU register R1 are copied to PCW.

Cycle Description:

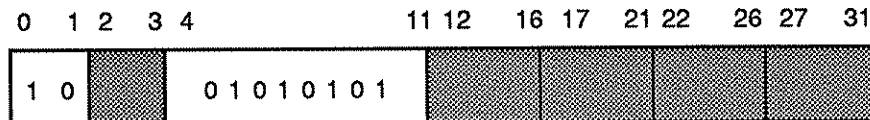
Cycle 1: PCW  $\leftarrow$  R1

## Special Instructions

### 5.9.22 Consumel

Mnemonic: Consumel

Format:



Description: This instruction consumes one integer condition code. The value of the condition consumed is immaterial.

Cycle Description:

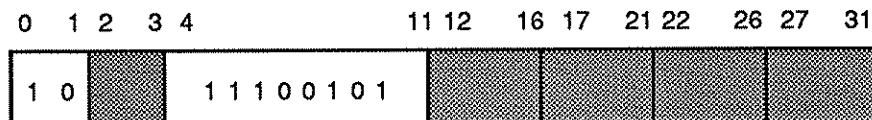
**Cycle 1:** CC1 ← CCi

## Special Instructions

### 5.9.23 ConsumeF

Mnemonic: ConsumeF

Format:



Description: This instruction consumes one floating point condition code. The value of the condition consumed is immaterial.

Cycle Description:

**Cycle 1:** CC1 ← CCf

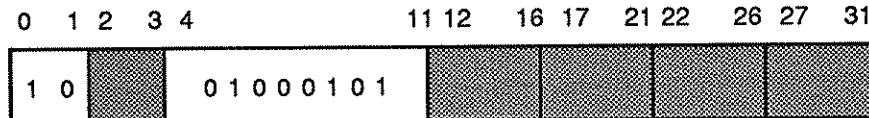


## Special Instructions

### 5.9.24 SYNCH

Mnemonic: SYNCH

Format:



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

An implementation may optimise the way this instruction is executed, but the semantics are as described here.

Cycle Description:

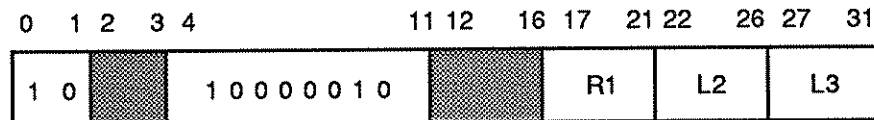
**Synch Cycle:** The execution of this instruction is delayed until all the instructions before the SYNCH instruction have been executed.

## Special Instructions

### 5.9.25 LoadM

Mnemonic: LoadM R1, L2, L3

Format:



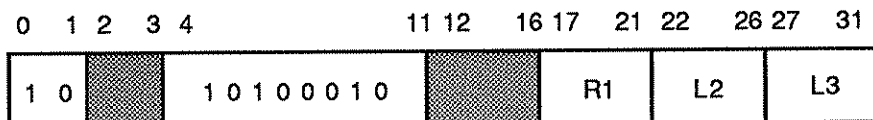
Description: LoadM loads a series of IEU registers, from register number L2 to register number L3, with L3 guaranteed by software to be a greater register number than L2. RL2 and RL3 always specify literals. The storage location is specified by the contents of the register specified by R1.  $1 < L2 \leq L3 \leq 31$ .

Cycle Description:

**Memory Cycle:** The IEU registers from L2 through L3 are assigned the contents of successive memory locations starting from the location specified by the contents of the register denoted by R1. [The locations are  $R1$ ,  $R1 + i/8$ , ...,  $R1 + (L3 - L2)i/8$ , where  $i$  is the size of the IEU registers in bits.]

### 5.9.26 FLoadM

Format:



Cycle Description:

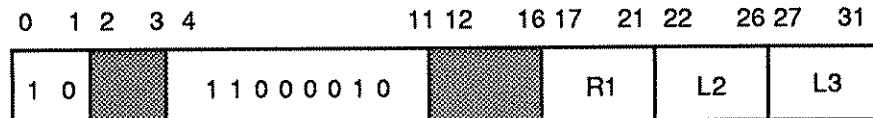
**Memory Cycle:** The FEU registers from L2 through L3 are assigned the contents of successive memory locations starting from the location specified by the contents of the register denoted by R1. [The locations are  $R1$ ,  $R1 + f/8$ , ...,  $R1 + (L3 - L2)f/8$ , where  $f$  is the size of the FEU registers in bits.]

## Special Instructions

### 5.9.27 VLoadM

Mnemonic: VLoadM R1, L2, L3

Format:



Description: VLoadM loads a series of VEU registers, from register number L2 to register number L3, with L3 guaranteed by software to be a greater register number than L2. RL2 and RL3 always specify literals. The storage location is specified by the contents of the register specified by R1.  $1 < L2 \leq L3 \leq 31$ .

Cycle Description:

**Memory Cycle:** The VEU registers from L2 through L3 are assigned the contents of successive memory locations starting from the location specified by the contents of the register denoted by R1. [The locations are R1,  $R1 + Nv/8$ , ...,  $R1 + (L3 - L2)Nv/8$ , where v is the size of the VEU registers in bits and N is the implementation defined depth of the VEU registers.]

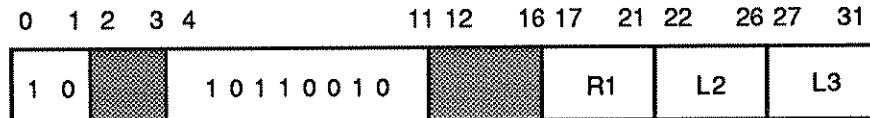


## Special Instructions

### 5.9.29 FStoreM

Mnemonic: FStoreM R1, L2, L3

Format:



Description: FStoreM stores a series of FEU registers, from register number L2 to register number L3, with L3 guaranteed by software to be a greater register number than L2. RL2 and RL3 always specify literals. The storage location is specified by the contents of the register specified by R1.  $1 < L2 \leq L3 \leq 31$ .

Cycle Description:

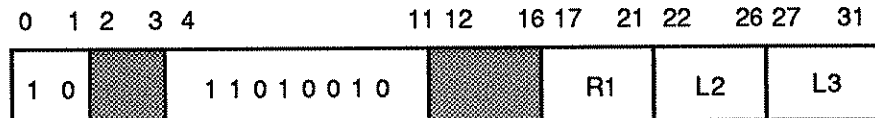
**Memory Cycle:** The contents of FEU registers from L2 through L3 are written to successive memory locations starting from the location specified by the contents of the register denoted by R1. [The locations are  $R1$ ,  $R1 + f/8$ , ...,  $R1 + (L3 - L2)f/8$ , where  $f$  is the size of the FEU registers in bits.]

## Special Instructions

### 5.9.30 VStoreM

Mnemonic: VStoreM R1, L2, L3

Format:



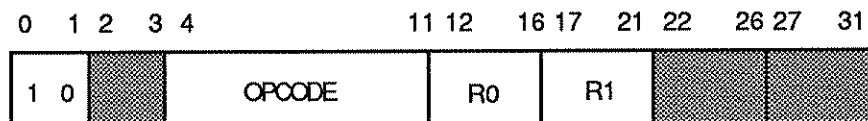
Description: VStoreM stores a series of VEU registers, from register number L2 to register number L3, with L3 guaranteed by software to be a greater register number than L2. RL2 and RL3 always specify literals. The storage location is specified by the contents of the register specified by R1.  $1 < L2 \leq L3 \leq 31$ .

Cycle Description:

**Memory Cycle:** The contents of VEU registers from L2 through L3 are written to successive memory locations starting from the location specified by the contents of the register denoted by R1. [The locations are  $R1$ ,  $R1 + Nv/8$ , ...,  $R1 + (L3 - L2)Nv/8$ , where  $v$  is the size of the VEU registers in bits and  $N$  is the implementation defined depth of the VEU registers.]

### 5.9.31 LoadFifoll, LoadFifoFl, LoadFifoVI

Format:



<u>OPCODE</u>	<u>Instruction</u>
00000011	LoadFifoI
00100011	LoadFifoFI
01000011	LoadFifoVI

Cycle Description:

**Memory Cycle:** The IEU/FEU/VEU input FIFO specified by R0 is loaded with the contents of successive memory locations starting from the location specified in denoted by R1. The number of values loaded is implementation dependent; however, a StoreFifo, LoadFifo pair that specifies the same FIFO and memory location results in leaving the FIFO in the state it was in before the pair of instructions was executed.

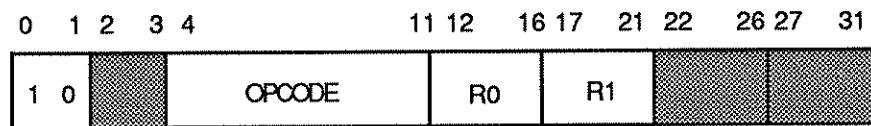


## Special Instructions

### 5.9.32 LoadFifoIO, LoadFifoFO, LoadFifoVO

Mnemonic: LoadFifoIO R0, R1  
LoadFifoFO R0, R1  
LoadFifoVO R0, R1

Format:



<u>OPCODE</u>	<u>Instruction</u>
00010011	LoadFifoIO
00110011	LoadFifoFO
01010011	LoadFifoVO

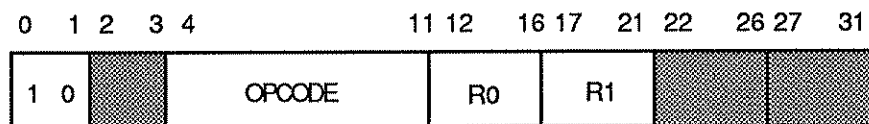
Description: These instructions load the specified FIFO state from the address specified in R1. The amount and format of this information is implementation dependent.

Cycle Description:

**Memory Cycle:** The output FIFO specified by R0 is loaded with the contents of successive memory locations starting from the location specified by the contents of the register denoted by R1. The number and format of values loaded is implementation dependent; however, a StoreFifo, LoadFifo pair that specifies the same FIFO and memory location results in leaving the FIFO in the state it was in before the pair of instructions was executed.

### 5.9.33 StoreFifoll, StoreFifoFI, StoreFifoVI

Format:



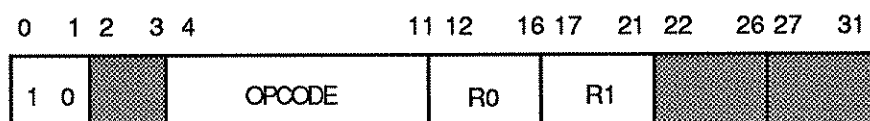
<u>OPCODE</u>	<u>Instruction</u>
10000011	StoreFifoll
10100011	StoreFifoFI
11000011	StoreFifoVI

Cycle Description:

**Memory Cycle:** The contents of the input FIFO specified by R0 are stored to successive memory locations starting from the location specified by the contents of the register denoted by R1. The format in which the contents are stored is implementation dependent; however, a StoreFifo, LoadFifo pair that specifies the same FIFO and memory location results in leaving the FIFO in the state it was in before the pair of instructions was executed.

### 5.9.34 StoreFifoIO, StoreFifoFO, StoreFifoVO

Format:



<u>OPCODE</u>	<u>Instruction</u>
10010011	StoreFifoO
10110011	StoreFifoFO
11010011	StoreFifoVO

Cycle Description:

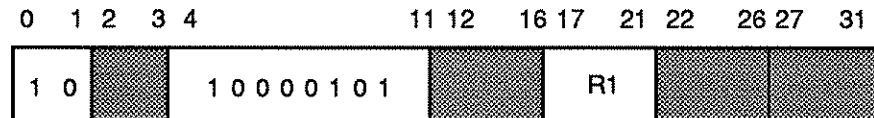
**Memory Cycle:** The contents of the output FIFO specified by R0 are stored to successive memory locations starting from the location specified by the contents of the register denoted by R1. The format in which the contents are stored is implementation dependent; however, a StoreFIFO, LoadFifo pair that specifies the same FIFO and memory location results in leaving the FIFO in the state it was in before the pair of instructions was executed.

## Special Instructions

### 5.9.35 LoadCTX

Mnemonic: LoadCTX R1

Format:



Description: LoadCTX restores context from a block of storage whose address is specified in R1.

Cycle Description:

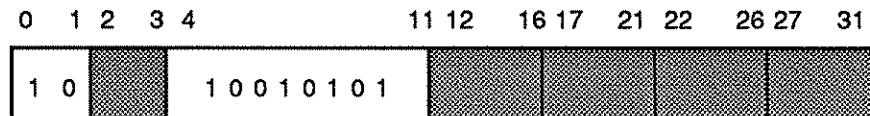
**Memory Cycle:** The set of general registers and special registers are loaded from successive memory locations, starting at the location specified in R1. The number and format of values loaded is implementation dependent.

## Special Instructions

### 5.9.36 StoreCTX

Mnemonic: StoreCTX

Format:



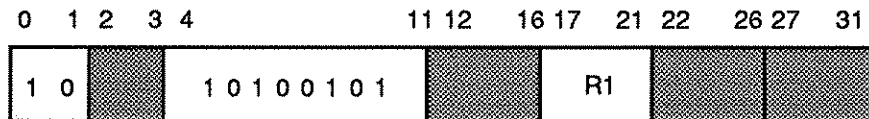
Description: StoreCTX stores the current context to a known location for the current task.

Cycle Description:

**Memory Cycle:** The contents of the set of general registers and special registers are stored to successive memory locations, starting at the a known location for the current task. The format of values stored is implementation dependent.

### 5.9.37 SwapCTX

Format:



Cycle Description:

**Memory Cycle:** The contents of the set of general registers and special registers are stored to successive memory locations, starting at the a known location for the current task. These registers are loaded from successive memory locations, starting at the location specified in R1. The number and format of values stored and loaded is implementation dependent.

### 5.9.38 SwapLT

Format:



Cycle Description:

**Memory Cycle:** The contents of the set of general registers and special registers are stored to successive memory locations, starting at the a known location for the current task. These registers are loaded from successive memory locations, starting at the location specified in the last TCB pointer (LTP). The number and format of values stored and loaded is implementation dependent.