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

Document Derivation.....................................................................................1
1. Scope and Purpose...................................................................................2
  1.1 Scope.....................................................................................................2
  1.2 Purpose..................................................................................................2
  1.3 Applicability..........................................................................................2
  1.4 Benefits..................................................................................................2
2. Referenced Documents..............................................................................2
3. Definitions..................................................................................................2
  3.1 Address..................................................................................................3
  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

4. General Requirements ......................................................... 7

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
| 5.4.12 op = = | 4.8 |
| 5.4.13 op = <= | 4.9 |
| 5.4.14 op = <= | 5.0 |
| 5.4.15 op = >= | 5.1 |
| 5.4.16 op = > | 5.2 |

### 5.5 Floating Point Instructions
- 5.5.1 op = < | 5.3 |
- 5.5.2 op = + | 5.4 |
- 5.5.3 op = - | 5.5 |
- 5.5.4 op = * | 5.6 |
- 5.5.5 op = / | 5.7 |
- 5.5.6 op = / | 5.8 |
- 5.5.7 op = / | 5.9 |
- 5.5.8 op = nop | 6.0 |
- 5.5.9 op = nop | 6.1 |
- 5.5.10 op = | 6.2 |
- 5.5.11 op = <> | 6.3 |
- 5.5.12 op = <= | 6.4 |
- 5.5.13 op = > | 6.5 |
- 5.5.14 op = >= | 6.6 |

### 5.6 Vector Instructions: Integer, Logical and Floating Point
- 5.6.1 op = iadd | 6.7 |
- 5.6.2 op = isub | 6.8 |
- 5.6.3 op = imul | 6.9 |
- 5.6.4 op = idiv | 7.0 |
- 5.6.5 op = iasl | 7.1 |
- 5.6.6 op = ieql | 7.2 |
- 5.6.7 op = ineq | 7.3 |
- 5.6.8 op = igtr | 7.4 |
- 5.6.9 op = igeq | 7.5 |
- 5.6.10 op = ilss | 7.6 |
- 5.6.11 op = ileq | 7.7 |
- 5.6.12 op = iaddC | 7.8 |
- 5.6.13 op = landC | 7.9 |
- 5.6.14 op = iorC | 8.0 |
- 5.6.15 op = ineqC | 8.1 |
- 5.6.16 op = isubC | 8.2 |
- 5.6.17 op = imulC | 8.3 |
- 5.6.18 op = idivC | 8.4 |
- 5.6.19 op = iaslC | 8.5 |
- 5.6.20 op = ieqlC | 8.6 |
- 5.6.21 op = ineqC | 8.7 |
- 5.6.22 op = igtrC | 8.8 |
- 5.6.23 op = igeqC | 8.9 |
- 5.6.24 op = ilssC | 9.0 |
- 5.6.25 op = ileqC | 9.1 |
- 5.6.26 op = faddC | 9.2 |
- 5.6.27 op = fsubC | 9.3 |
- 5.6.28 op = fmulC | 9.4 |
- 5.6.29 op = fnmsub | 9.5 |
- 5.6.30 op = fnmsub | 9.6 |
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.9.30 VStoreM</td>
<td>151</td>
</tr>
<tr>
<td>5.9.31 LoadFifo, LoadFifoFI, LoadFifoVI</td>
<td>152</td>
</tr>
<tr>
<td>5.9.32 LoadFifoO, LoadFifoFO, LoadFifoVO</td>
<td>153</td>
</tr>
<tr>
<td>5.9.33 StoreFifo, StoreFifoFI, StoreFifoVI</td>
<td>154</td>
</tr>
<tr>
<td>5.9.34 StoreFifoO, StoreFifoFO, StoreFifoVO</td>
<td>155</td>
</tr>
<tr>
<td>5.9.35 LoadCTX</td>
<td>156</td>
</tr>
<tr>
<td>5.9.36 StoreCTX</td>
<td>157</td>
</tr>
<tr>
<td>5.9.37 SwapCTX</td>
<td>158</td>
</tr>
<tr>
<td>5.9.38 SwapLT</td>
<td>159</td>
</tr>
</tbody>
</table>
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 $-2^{i-1} ... (2^{i-1}-1)$. 
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 reduces 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\(^1\), the IFU can be thought of as enqueuing instructions for execution by each of the other execution units in a set of FIFOs. In addition the IFU

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

\[ R0 := (R1 \ op1 \ R2) \ op2 \ R3 \]

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, \( \text{op2} \)) operation of one instruction is being executed in ALU2, the first (inner, \( \text{op1} \)) 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 relational operators 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 relational operators preceding the associated conditional jump or consume instruction must not exceed the size of the condition bit FIFO.
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

\[
\text{forall } k, 0 \leq k < N, \ R0_k := \begin{cases} \text{if } R3_k \neq 0 & \text{then } (R1_k \text{ op } R2_k) \text{ else } R0_k \end{cases}
\]

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.

\[\text{Figure 3: Vector Execution Unit}\]

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 EC all. Call simply stores the current PC in register 4 and jumps to the specified destination. EC all 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 EC all 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 J umpI (Jump Indirect), CallI (Call Indirect) and ER e turn (Return from EC all).

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:

\[
\begin{align*}
  r1 & := p1 & \text{-- 1st parameter} \\
  & \vdots \\
  r1 & := pn & \text{-- Nth parameter} \\
  \text{call} & \text{ subr} & \text{-- implicitly, } \ r4 & := \text{PC}
\end{align*}
\]
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 count¹ (RL2), a stride² (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

¹ A count of -1 is defined to be an infinitely long stream. That is, the stream will continue until a stop streaming instruction is performed.
² In bytes.
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\(^1\).

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\(^2\).
- determine if a value is within certain bounds. If it is not, a hardware Assert Fault is generated. Unlike the integer and floating point relational, 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.

---

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

\(^2\) 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 y-bit vector (fixed and floating) point precision data in twos complement representation. A member of the family is denoted by three parameters, and is denoted WM_{i,f,y}. 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:

\[ i \in \{16, 32, 64\} \]
\[ f \in \{0, 32, 64\} \]
\[ y \in \{0, 32, 64\} \]

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 (MSF) 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 \( j \)-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.

\[
\begin{array}{ccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 7 & 8 & 11 & 12 & 16 & 17 & 21 & 22 & 26 & 27 & 31 \\
00 & RL & OP1 & OP2 & RO & R1 & RL2 & RL3 \\
\end{array}
\]

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.

\[
\begin{array}{ccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 7 & 8 & 11 & 12 & 16 & 17 & 21 & 22 & 26 & 27 & 31 \\
01 & RL & LSOP & op & op & RO & R1 & RL2 & RL3 \\
\end{array}
\]
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:

\[ R0 := (R1 \ \textbf{op1} \ \text{RL2}) \ \textbf{op2} \ \text{RL3} \]

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.

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 7 & 8 & 11 & 12 & 16 & 17 & 21 & 22 & 26 & 27 & 31 \\
11 & 00 & \text{OP1} & \text{OP2} & \text{RO} & \text{R1} & \text{R2} & \text{R3} \\
\end{array}
\]

### 4.3.6 Control format instructions

Control instructions are executed by the IFU.

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 11 & 12 & 31 \\
11 & 11 & \text{OP} & \text{OFFSET} \\
\end{array}
\]

The offset is extended till two least significant zero digits.
4.3.7 Vector format instructions.
Vector instructions are executed by the VEU.

<table>
<thead>
<tr>
<th>0 1 2 3 4</th>
<th>11 12 16 17 21 22 26 27 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 01</td>
<td>OP RO R1 R2 R3</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>0 1 2 3 4</th>
<th>11 12 16 17 21 22 26 27 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 RL</td>
<td>OP RO R1 RL2 RL3</td>
</tr>
</tbody>
</table>

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 7-bit entries in the integer unit's input FIFOs
- 1 7-bit entry in the integer unit's output FIFO
- 3 7-bit entries in the floating unit's input FIFOs
- 1 7-bit entry in the floating unit's output FIFO
- 3 N-component blocks of γ-bit entries in the vector unit's input FIFO, and
- 1 N-component block of γ-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:

<table>
<thead>
<tr>
<th>Bit#</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AND/OR relationsals (AND == 1)</td>
</tr>
<tr>
<td>-</td>
<td>Exceptions Enabled (enabled == 1):</td>
</tr>
<tr>
<td>1</td>
<td>Attempted Stack Limit Modification</td>
</tr>
<tr>
<td>2</td>
<td>Stack Index Negative</td>
</tr>
<tr>
<td>3</td>
<td>Assert Fault</td>
</tr>
<tr>
<td>4</td>
<td>Integer Divide By Zero</td>
</tr>
<tr>
<td>5</td>
<td>Floating Divide By Zero</td>
</tr>
<tr>
<td>6</td>
<td>Integer Arithmetic Overflow</td>
</tr>
<tr>
<td>7</td>
<td>Integer Arithmetic Underflow</td>
</tr>
<tr>
<td>8</td>
<td>Floating Arithmetic Overflow</td>
</tr>
<tr>
<td>9</td>
<td>Floating Arithmetic Underflow</td>
</tr>
<tr>
<td>10</td>
<td>Cycle Counter Overflow</td>
</tr>
<tr>
<td>11</td>
<td>Raise Address</td>
</tr>
<tr>
<td>12</td>
<td>Raise Call</td>
</tr>
<tr>
<td>13</td>
<td>Raise Jump</td>
</tr>
</tbody>
</table>

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.

```
Virtual Page Number ... Byte Within Page
```

The boundary is somewhere

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\(^1\).

---
\(^1\)The PTP may be null because protection is not implemented on a certain model of WM. In addition, however, PTP is null when the processor is first "booted" -- this corresponds to the "most privileged state".
A Protection Entry is a byte, with the following format:

<table>
<thead>
<tr>
<th>type</th>
<th>rights</th>
</tr>
</thead>
</table>

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Memory</td>
</tr>
<tr>
<td>0001</td>
<td>TCB</td>
</tr>
<tr>
<td>0010</td>
<td>Entry</td>
</tr>
<tr>
<td>0011</td>
<td>Device</td>
</tr>
<tr>
<td>0100-0111</td>
<td>reserved for hardware</td>
</tr>
<tr>
<td>1000-1111</td>
<td>reserved for software</td>
</tr>
</tbody>
</table>

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\(^1\).

---

\(^1\)The MTP may be null because virtual memory is not implemented on a certain model of WM. In addition, however, MTP is null when the processor is first "booted" -- this corresponds to the "unmapped processor state".
Map table entries have the following format:

```
  V I A M  |  SW |
              physical page number
```

and their bits are interpreted as follows:

- 0  Valid - this page exists in physical memory
- 1  Locked --this page is locked into memory
- 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.

---

1 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:

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

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:

\[ r_0, ..., r_{31} \] refer to the integer/logical registers in the IEU,
\[ f_0, ..., f_{31} \] refer to the floating point registers in the FEU, and
\[ v_0, ..., v_{31} \] refer to the registers in the VEU.

In addition \( r_Z, f_Z, \) and \( v_Z \) 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. IEU registers

\[ r_0 \] Register \( r_0 \) 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.

\textbf{Input FIFO:} \( r_0 \) refers to the input FIFO \( r_0.input \) when a) \( r_0 \) appears to the right of the \( \leftarrow \) symbol or b) \( r_0 \) appears in a value context as in a \textit{if} statement or c) when the input FIFO is explicitly specified.

\textbf{Output FIFO:} \( r_0 \) refers to the output FIFO \( r_0.output \) when a) \( r_0 \) 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 \( r_1 \) is similar to \( r_0 \) with the difference that an assignment to the output FIFO \( r_1 \) results in the value being enqueued in the \( r_1 \) input FIFO as well.

\[ r_2 - r_{30} \] These denote i-bit integer registers.

\[ r_{31} \] \( r_{31} \) is a i-bit register with the value 0. Assignments to \( r_{31} \) have no effect on its contents.
5.1.1.1.2 FEU registers

f0-131 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 ← 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 ← 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 "⇐"

5.1.2.1.1 Symbols to the left of ⇐:

Output FIFO The values to the right of the ⇐ 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 ⇐.

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

5.1.2.1.2 Symbols to the right of ⇐

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 "←": The assignment operator.
The assignment operator syntax is

X ← 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 ← operator.

5.1.2.2.1 Symbols to the left of ←

Input FIFO An assignment to an input FIFO has the effect of enqueuing the value of the expression to the right of the ← 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 ← 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 ← symbol is enqueued 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 is 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**

If the 'qualifier' field of the record at the head of the FIFO is ADDR, then the FIFO is dequeued. If the 'tag' field is being assigned CHANGED, the value in the value field of the register to the right of the ← 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 ← symbol is enqueued into the 'value' field at the tail of the FIFO, the corresponding 'qualifier' field is set to DATA and the tag field is set as specified.
In either of the above cases, if the FIFO was is streaming mode, consume_count is decremented by one.

**r2 · r30**
The value of the expression to the right of the ← is assigned to the specified register.

**f2-f30, v2-v30**
Similar to r2-r30

**r31, f31, v31**
The assignment does not change the contents of the specified register.

**CCI, CCf**
An assignment to CCI/CCf enqueues the value of the boolean expression to the right of the ← operator.

**M[addr, size]**
size bytes of memory starting at location addr are assigned the value of the expression to the right of the ← operator. The value of the expression should also be size bytes in length.

### 5.1.2.2.2 Symbols to the right of ←

**Input FIFO**
The input FIFO is dequeued and this is the value of the symbol. If the FIFO is in streaming mode, the corresponding consume_count 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.

**r2 · r31**
The value of the symbol is the contents of the specified register.

**f2-f31, v2-v31**
Similar to r2-r31

**CCI, CCf**
The specified condition code FIFO is dequeued and this is the value of the symbol.
The value of \textit{size} bytes of data from the memory starting at location \textit{addr}.

\textbf{5.1.2.3} "::" : The bit selection operator

\texttt{reg;i}

Selects the \textit{i}th bit of the specified register 'reg'.

\texttt{reg;i-j}

Selects \textit{j-i+1} bits of the specified register starting from bit \textit{i}.

\textbf{5.1.2.4} ":=" : The operator-argument operator

\texttt{opr :: args}

Passes the arguments \texttt{args} to the operator \texttt{opr}.

\textbf{5.1.2.5} ":sl" : The arithmetic shift left operator

\texttt{X Isl Y}

Shifts the register \texttt{X} left by the amount specified by the register/literal \texttt{Y}.

\textbf{5.1.2.6} ":asr" : The arithmetic shift right operator

\texttt{X asr Y}

Arithmetic shifts register \texttt{X} right by the amount specified by the register/literal \texttt{Y}.

\textbf{5.1.2.7} ":&\&": The bitwise and operator

\texttt{X &\& Y}

The two registers \texttt{X} and \texttt{Y} are bitwise and-ed.

\textbf{5.1.2.8} 'and' : The logical and operator

\texttt{X and Y}

The two boolean values \texttt{X} and \texttt{Y} are logical and-ed and the result is \texttt{TRUE} or \texttt{FALSE}.

\textbf{5.1.2.9} ":||" : The bitwise or operator

\texttt{X || Y}

The registers \texttt{X} and \texttt{Y} are bitwise or-ed.

\textbf{5.1.2.10} ":or" : The logical or operator

\texttt{X or Y}

The two boolean values \texttt{X} and \texttt{Y} are logical or-ed and the result is \texttt{TRUE} or \texttt{FALSE}.

\textbf{5.1.2.11} "EQV" : The bitwise equivalence operator

\texttt{X EQV Y}

The registers \texttt{X} and \texttt{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.

\textbf{5.1.2.12} ":+, -, \cdot, \ast, /, \div"\footnote{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.}

\textbf{5.1.2.13} ":=, \lt, \lt, \le, \ge, \gt"\footnote{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.}

\textbf{5.1.2.14} "\phi"

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 bits, where 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 not 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

TRUE, FALSE  The boolean values TRUE and FALSE respectively.

NOT EVAL  A value used for condition code operations. This value is neither TRUE nor FALSE and means 'not evaluated'.

AND  A constant with value 1.

CHANGED, UNCHANGED  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: \( \text{int } R0 := (R1 \text{ op1 RL2}) \text{ op2 RL3} \)

Format:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 0 | 0 | RL | OP1 | OP2 | R0 | R1 | RL2 | RL3 |

<table>
<thead>
<tr>
<th>symbol</th>
<th>encoding</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0010</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>0000</td>
<td>subtraction</td>
</tr>
<tr>
<td>'</td>
<td>0100</td>
<td>reverse subtraction</td>
</tr>
<tr>
<td>*</td>
<td>0001</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>1000</td>
<td>division</td>
</tr>
<tr>
<td>′</td>
<td>1100</td>
<td>reverse division</td>
</tr>
<tr>
<td>asl</td>
<td>0011</td>
<td>arithmetically shift left</td>
</tr>
<tr>
<td>eqv</td>
<td>0101</td>
<td>bitwise EQUIVALENCE</td>
</tr>
<tr>
<td>or</td>
<td>0110</td>
<td>bitwise OR</td>
</tr>
<tr>
<td>and</td>
<td>0111</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>=</td>
<td>1010</td>
<td>equal</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>1110</td>
<td>not equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>1011</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>1101</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;=</td>
<td>1001</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>1111</td>
<td>greater than</td>
</tr>
</tbody>
</table>

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: \( X_1 \leftarrow R_1 \text{ asl } RL_2 \)
\( CC_1 \leftarrow \text{ NOT EVAL} \)

Cycle 2: \( X_2 \leftarrow X_1 \text{ asl } RL_3 \)
\( CC_2 \leftarrow \text{ NOT EVAL} \)
If \( CC_1 \neq \text{ FALSE} \) then \( R_0 \leftarrow X_2 \)
If relational (op1) then \( CC_i \leftarrow CC_1 \)
Integer Arithmetic and Logical Instructions

5.4.3 \( \text{op} = \lt \) 

**Cycle 1:** \( X_1 \leftarrow R_1 \)
   \[
   \text{If } R_1 < \text{RL2 then } C_{C1} \leftarrow \text{TRUE else } C_{C1} \leftarrow \text{FALSE}
   \]

**Cycle 2:** if \( X_1 < \text{RL3} \) then \( C_{C2} \leftarrow \text{TRUE else } C_{C2} \leftarrow \text{FALSE} \)
   \[
   \text{if relational (op1) then}
   \]
   \[
   \text{if PCW:0 = AND then } C_{Ci} \leftarrow C_{C1} \text{ and } C_{C2} \text{ else } C_{Ci} \leftarrow C_{C1} \text{ or } C_{C2}
   \]
   \[
   \text{else } C_{Ci} \leftarrow C_{C2}
   \]
   \[
   \text{if } C_{Ci} \text{ was assigned TRUE then } R_0 \leftarrow X_1
   \]
Integer Arithmetic and Logical Instructions

5.4.4 \texttt{op} = -

\textbf{Cycle 1:} $X_1 \leftarrow R_1 - RL_2$
\hspace{1em} if underflow then $PSW:7 \leftarrow 1$
\hspace{1em} $CC_1 \leftarrow \text{NOT EVAL}$

\textbf{Cycle 2:} $X_2 \leftarrow X_1 - RL_3$
\hspace{1em} $CC_2 \leftarrow \text{NOT EVAL}$
\hspace{1em} if underflow then
\hspace{2em} if $CC_1 \neq \text{FALSE}$ then $PSW:7 \leftarrow 1$
\hspace{2em} else
\hspace{3em} if $CC_1 \neq \text{FALSE}$ then $R_0 \leftarrow X_2$
\hspace{3em} if relational (op1) then $CC_i \leftarrow CC_1$
Integer Arithmetic and Logical Instructions

5.4.5 op = `-`

**Cycle 1:** \(X_1 \leftarrow RL2 - R1\)
- If underflow then \(PSW:7 \leftarrow 1\)
- \(CC1 \leftarrow NOT EVAL\)

**Cycle 2:** \(X_2 \leftarrow RL3 - X_1\)
- \(CC2 \leftarrow NOT EVAL\)
- 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

5.4.6 op = and

Cycle 1: X1 ← R1 && RL2
      CC1 ← NOT EVAL

Cycle 2: X2 ← X1 && RL3
      CC2 ← NOT EVAL
      if CC1 = FALSE then R0 ← X2
      if relational (op1) then CCi ← CC1
5.4.7 \texttt{op} = \texttt{or}

\textbf{Cycle 1:} \hspace{1em} X1 \leftarrow R1 \parallel RL2
\hspace{1em} CC1 \leftarrow \text{NOT EVAL}

\textbf{Cycle 2:} \hspace{1em} X2 \leftarrow X1 \parallel RL3
\hspace{1em} CC2 \leftarrow \text{NOT EVAL}
\hspace{1em} \text{if CC1} \neq \text{FALSE then } R0 \leftarrow X2
\hspace{1em} \text{if relational (op1) then } CC1 \leftarrow CC1
Integer Arithmetic and Logical Instructions

5.4.8 op = eqv

**Cycle 1:** \( X_1 \leftarrow R_1 \text{ EQV RL2} \)
\( CC_1 \leftarrow \text{ NOT EVAL} \)

**Cycle 2:** \( X_2 \leftarrow X_1 \text{ EQV RL3} \)
\( CC_2 \leftarrow \text{ NOT EVAL} \)
\( \text{If } CC_1 \neq \text{ FALSE then } R_0 \leftarrow X_2 \)
\( \text{If relational (op1) then } CC_1 \leftarrow CC_1 \)
Integer Arithmetic and Logical Instructions

5.4.9 op = *

Cycle 1: X1 ← R1 * RL2
    if overflow then PSW:8 ← 1
    CC1 ← NOT EVAL

Cycle 2: X2 ← X1 * RL3
    CC2 ← NOT EVAL
    if overflow then
        if CC1 ≠ FALSE then PSW:6 ← 1
    else
        if CC1 ≠ FALSE then R0 ← X2
        if relational (op1) then CCi ← CC1
Integer Arithmetic and Logical Instructions

5.4.10 \( \text{op} = / \)

**Cycle 1:** If \( \text{RL2} = 0 \) then \( \text{PSW:4} \leftarrow 1 \) else
- \( X1 \leftarrow \text{R1} / \text{R2} \)
- If underflow then \( \text{PSW:7} \leftarrow 1 \)
- \( \text{CC1} \leftarrow \text{NOT EVAL} \)

**Cycle 2:** If \( \text{RL3} = 0 \) then
- If \( \text{CC1} \neq \text{FALSE} \) then \( \text{PSW:4} \leftarrow 1 \)
  else
  - \( X2 \leftarrow X1 / \text{RL3} \)
  - \( \text{CC2} \leftarrow \text{NOT EVAL} \)
  - If underflow then
    - If \( \text{CC1} \neq \text{FALSE} \) then \( \text{PSW:7} \leftarrow 1 \)
  - else
    - If \( \text{CC1} \neq \text{FALSE} \) then \( R0 \leftarrow X2 \)
    - If relational (op1) then \( \text{CCI} \leftarrow \text{CC1} \)
5.4.11 \( \text{op} = '/' \)

**Cycle 1:** If \( R_1 = 0 \) then \( \text{PSW:4} \leftarrow 1 \) else

\[ X_1 \leftarrow RL2 / R1 \]

if underflow then \( \text{PSW:7} \leftarrow 1 \)

\( \text{CC1} \leftarrow \text{NOT EVAL} \)

**Cycle 2:** If \( X_1 = 0 \) then \( \text{PSW:4} \leftarrow 1 \) else

\[ X_2 \leftarrow RL3 / X1 \]

\( \text{CC2} \leftarrow \text{NOT EVAL} \)

if underflow then

if \( \text{CC1} \neq \text{FALSE} \) then \( \text{PSW:7} \leftarrow 1 \)

else

if \( \text{CC1} \neq \text{FALSE} \) then \( R_0 \leftarrow X_2 \)

if relational (op1) then \( \text{CCi} \leftarrow \text{CC1} \)
Integer Arithmetic and Logical Instructions

5.4.12 \texttt{op} = =

\textbf{Cycle 1:} \(X_1 \leftarrow R_1\)
   \hspace{1em} If \(R_1 = RL_2\) then \(CC_1 \leftarrow \text{TRUE}\) else \(CC_1 \leftarrow \text{FALSE}\)

\textbf{Cycle 2:} If \(X_1 = RL_3\) then \(CC_2 \leftarrow \text{TRUE}\) else \(CC_2 \leftarrow \text{FALSE}\)
   \hspace{1em} if relational (op1) then
   \hspace{2em} if PCW:0 = \texttt{AND} then \(CC_i \leftarrow CC_1\) and \(CC_2\) else \(CC_i \leftarrow CC_1\) or \(CC_2\)
   \hspace{2em} else \(CC_i \leftarrow CC_2\)
   \hspace{1em} if \(CC_i\) was assigned \text{TRUE} then \(R_0 \leftarrow X_1\)
Integer Arithmetic and Logical Instructions

5.4.13 op = <->

**Cycle 1:** X1 ← R1
    If R1 ≠ RL2 then CC1 ← TRUE else CC1 ← FALSE

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

5.4.15 op = >=

Cycle 1: X1 ← R1
   If R1 ≥ RL2 then CC1 ← TRUE else CC1 ← FALSE

Cycle 2: If X1 ≥ RL3 then CC2 ← TRUE else CC2 ← FALSE
   if relational (op1) then
      if PCW.0 = AND then CCi ← CC1 and CC2 else CCi ← CC1 or CC2
   else CCi ← CC2
   if CCi was assigned TRUE then R0 ← X1
Integer Arithmetic and Logical Instructions

5.4.16 $op = >$

Cycle 1: $X1 \leftarrow R1$
   
   If $R1 > RL2$ then $CC1 \leftarrow TRUE$ else $CC1 \leftarrow FALSE$

Cycle 2: If $X1 > RL3$ then $CC2 \leftarrow TRUE$ else $CC2 \leftarrow FALSE$
   
   If relational $(op1)$ then
   
   If $PCW:0 = AND$ 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:

<table>
<thead>
<tr>
<th>0 1 2 3 4 7 8 11 12 16 17 21 22 26 27 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>symbol</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>+</td>
<td>0010</td>
</tr>
<tr>
<td>subtraction</td>
<td>-</td>
<td>0000</td>
</tr>
<tr>
<td>reverse subtraction</td>
<td>-'</td>
<td>0100</td>
</tr>
<tr>
<td>multiplication</td>
<td>*</td>
<td>0001</td>
</tr>
<tr>
<td>division</td>
<td>/</td>
<td>1000</td>
</tr>
<tr>
<td>reverse division</td>
<td>/'</td>
<td>1100</td>
</tr>
<tr>
<td>pass the left operand</td>
<td>nop</td>
<td>0011</td>
</tr>
<tr>
<td>pass the right operand</td>
<td>nop'</td>
<td>0111</td>
</tr>
<tr>
<td>reserved</td>
<td>0110</td>
<td>reserved</td>
</tr>
<tr>
<td>reserved</td>
<td>0101</td>
<td>reserved</td>
</tr>
<tr>
<td>equal</td>
<td>=</td>
<td>1010</td>
</tr>
<tr>
<td>not equal</td>
<td>&lt;&gt;</td>
<td>1110</td>
</tr>
<tr>
<td>less than</td>
<td>&lt;</td>
<td>1011</td>
</tr>
<tr>
<td>less than or equal</td>
<td>&lt;=</td>
<td>1101</td>
</tr>
<tr>
<td>greater than or equal</td>
<td>&gt;=</td>
<td>1001</td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
<td>1111</td>
</tr>
</tbody>
</table>

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:** $Y_1 \leftarrow R_1$
- If $R_1 < R_2$ then $CC_1 \leftarrow$ TRUE else $CC_1 \leftarrow$ FALSE

**Cycle 2:** If $Y_1 < R_3$ then $CC_2 \leftarrow$ TRUE else $CC_2 \leftarrow$ FALSE
- If relational (op1) then
  - If $PCW:0 = \text{AND}$ then $CCf \leftarrow CC_1 \text{ and } CC_2$ else $CCf \leftarrow CC_1 \text{ or } CC_2$
  - else $CCf \leftarrow CC_2$
- If $CCf$ was assigned TRUE then $R_0 \leftarrow Y_1$
Floating Point Instructions

5.5.2 op = +

Cycle 1: \( Y_1 \leftarrow R_1 + R_2 \)
- If overflow as defined by IEEE Std. 754 then PSW:8 \( \leftarrow 1 \)
- \( CC1 \leftarrow NOT EVAL \)

Cycle 2: \( Y_2 \leftarrow Y_1 + R_3 \)
- \( CC2 \leftarrow NOT EVAL \)
  - 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 \( R_0 \leftarrow Y_2 \)
    - If relational (op1) then \( CCf \leftarrow CC1 \)
Floating Point Instructions

5.5.3 op = -

Cycle 1: \( Y_1 \leftarrow R_1 - R_2 \)
\[
\text{if underflow as defined by the IEEE Std. 754 then PSW:9} \leftarrow 1
\]
\( C_{C1} \leftarrow \text{NOT EVAL} \)

Cycle 2: \( Y_2 \leftarrow Y_1 - R_3 \)
\( C_{C2} \leftarrow \text{NOT EVAL} \)
\[
\text{if underflow as defined by the IEEE Std 754 then}
\]
\[
\text{if } C_{C1} \neq \text{FALSE then PSW:9} \leftarrow 1
\]

\text{else}
\[
\text{if } C_{C1} \neq \text{FALSE then } R_0 \leftarrow Y_2
\]
\[
\text{if relational (op1) then } C_{Cf} \leftarrow C_{C1}
\]
Floating Point Instructions

5.5.4 \( \text{op} = -' \)

Cycle 1: \( Y_1 \leftarrow R_2 - R_1 \)
- If underflow as defined by IEEE Std. 754 then PSW:9 \( \leftarrow 1 \)
- \( CC1 \leftarrow \text{NOT EVAL} \)

Cycle 2: \( Y_2 \leftarrow R_3 - Y_1 \)
- \( CC2 \leftarrow \text{NOT EVAL} \)
- 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 \( R_0 \leftarrow Y_2 \)
    - If relational (op1) then \( CCf \leftarrow CC1 \)
Floating Point Instructions

5.5.5 op = *

Cycle 1: Y1 ← R1 * R2
   If overflow as defined by IEEE Std. 754 then PSW:8 ← 1
   CC1 ← NOT EVAL

Cycle 2: Y2 ← Y1 * R3
   CC2 ← NOT EVAL
   If overflow as defined by the IEEE Std 754 then
   if CC1 ≠ FALSE then PSW:8 ← 1
   else
   if CC1 ≠ FALSE then R0 ← Y2
   if relational (op1) then CCf ← CC1
Floating Point Instructions

5.5.6 \textbf{op} = \div

\textbf{Cycle 1}: \textbf{if} \text{R2} = 0 \textbf{then} \text{PSW:5} \leftarrow 1 \textbf{else} \text{Y1} \leftarrow \text{R1} / \text{R2}
\text{if} \text{underflow as defined by the Std. 754 then} \text{PSW:9} \leftarrow 1
\text{CC1} \leftarrow \text{NOT EVAL}

\textbf{Cycle 2}: \textbf{if} \text{R3} = 0 \textbf{then}
\hspace{1em} \textbf{if} \text{CC1} \neq \text{FALSE} \textbf{then} \text{PSW:5} \leftarrow 1
\textbf{else}
\hspace{2em} \text{Y2} \leftarrow \text{Y1} / \text{R3}
\hspace{2em} \text{CC2} \leftarrow \text{NOT EVAL}
\hspace{2em} \text{if} \text{underflow as defined by the IEEE Std 754 then}
\hspace{3em} \text{if} \text{CC1} \neq \text{FALSE} \textbf{then} \text{PSW:9} \leftarrow 1
\textbf{else}
\hspace{4em} \textbf{if} \text{CC1} \neq \text{FALSE} \textbf{then} \text{R0} \leftarrow \text{Y2}
\hspace{4em} \text{if relational (op1) then} \text{CC1} \leftarrow \text{CC1}
Floating Point Instructions

5.5.7 op = /

Cycle 1: if R1 = 0 then PSW:5 ← 1 else Y1 ← R2 / R1
   CC1 ← NOT EVAL

Cycle 2: if Y1 = 0 then
   if CC1 ≠ FALSE then PSW:5 ← 1
   else
      Y2 ← R3 / Y1
      CC2 ← NOT EVAL
      if underflow as defined by the IEEE Std 754 then
         if CC1 ≠ FALSE then PSW:9 ← 1
      else
         if CC1 ≠ FALSE then R0 ← Y2
         If relational (op1) then CCF ← CC1
Floating Point Instructions

5.5.8 \texttt{op} = \texttt{nop}

\textbf{Cycle 1:} Y1 \leftarrow R1  \\
\quad \text{CC1} \leftarrow \text{NOT EVAL}

\textbf{Cycle 2:} Y2 \leftarrow Y1  \\
\quad \text{CC2} \leftarrow \text{NOT EVAL}  \\
\quad \text{if CC1 \neq FALSE then } R0 \leftarrow Y2  \\
\quad \text{if relational (op1) then } \text{CCf} \leftarrow \text{CC1}
Floating Point Instructions

5.5.9 op = nop'

Cycle 1: Y1 ← R2
CC1 ← NOT EVAL

Cycle 2: Y2 ← R3
CC2 ← NOT EVAL
if CC1 ≠ FALSE then R0 ← Y2
if relational (op1) then CC1 ← CC1
Floating Point Instructions

5.5.10 \textbf{op} = =

\textbf{Cycle 1}: \textit{Y1} \leftarrow \textit{R1}
\begin{itemize}
\item If \textit{R1} = \textit{R2} then \textit{CC1} \leftarrow \text{TRUE} else \textit{CC1} \leftarrow \text{FALSE}
\end{itemize}

\textbf{Cycle 2}: if \textit{Y1} = \textit{R3} then \textit{CC2} \leftarrow \text{TRUE} else \textit{CC2} \leftarrow \text{FALSE}
\begin{itemize}
\item if relational (\textit{op1}) then
  \begin{itemize}
  \item if PCW:0 = \text{AND} then \textit{CCf} \leftarrow \text{CC1 and CC2} else \textit{CCf} \leftarrow \text{CC1 or CC2}
  \item else \textit{CCf} \leftarrow \text{CC2}
  \end{itemize}
\item if \textit{CCf} was assigned \text{TRUE} then \textit{R0} \leftarrow \text{Y1}
\end{itemize}
Floating Point Instructions

5.5.11 op = <>

Cycle 1: Y1 ← R1
     If R1 ≠ R2 then CC1 ← TRUE else CC1 ← FALSE

Cycle 2: if Y1 ≠ R3 then CC2 ← TRUE else CC2 ← FALSE
     if relational (op1) then
         if PCW:0 = AND then CCf ← CC1 and CC2 else CCf ← CC1 or CC2
     else CCf ← CC2
     if CCf was assigned TRUE then R0 ← Y1
Floating Point Instructions

5.5.12 op = <=

Cycle 1: Y1 ← R1
    If R1 ≤ R2 then CC1 ← TRUE else CC1 ← FALSE

Cycle 2: If Y1 ≤ R3 then CC2 ← TRUE else CC2 ← FALSE
    if relational (op1) then
        If PCW.0 = AND then CCf ← CC1 and CC2 else CCf ← CC1 or CC2
    else CCf ← CC2
    if CCf was assigned TRUE then R0 ← 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 = 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

5.5.14  op = >=

Cycle 1: Y1 ← R1
   If R1 ≥ R2 then CC1 ← TRUE else CC1 ← FALSE

Cycle 2: if Y1 ≥ R3 then CC2 ← TRUE else CC2 ← FALSE
   if relational (op1) then
      if PCW.0 = AND then CCf ← CC1 and CC2 else CCf ← CC1 or CC2
   else CCf ← CC2
   if CCf was assigned TRUE then R0 ← Y1
5.6 Vector Instructions: Integer, Logical and Floating Point

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

**Format:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>OP</th>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
</table>

**OP Encoding Table**

<table>
<thead>
<tr>
<th></th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
</tr>
</thead>
</table>
| 0000 | isub | isubC | fsub | fsubC | VCVTF | VCVTF
| 0001 | imul | imulC | fmul | fmulC | VCVTF | VCVTF
| 0010 | iadd | iaddC | fadd | faddC |
| 0011 | iasl | iaslC |       |       |       |       |
| 0100 |     |      |       |       |       |       |
| 0101 | ieqv | ieqvC |       |       |       |       |
| 0110 | lor  | lorC  |       |       |       |       |
| 0111 | land | landC |       |       |       |       |
| 1000 | idiv | idivC | fdiv | fdivC |
| 1001 | lse  | lseC  | fseq | fseqC |
| 1010 | leql | leqC  | feq  | feqC  |
| 1011 | llss | llssC | flss | flssC |
| 1100 |     |      |       |       |       |       |
| 1101 | lileq | lileqC | fleq | fleqC |
| 1110 | lineq | lineqC | fneq | fneqC |
| 1111 | lgtr | lgtrC | fgtr | fgtrC |

**Description:** These are integer, logical and floating point operations on "blocks" of N x-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 \texttt{op = iadd}

\textbf{Cycle 1: }\forall i \in \{0, 1, \ldots, N - 1\}
\begin{align*}
Z1.i.\text{value} & \leftarrow R1.i.\text{value} + R2.i.\text{value} \\
\text{if overflow then } & PSW:14 \leftarrow 1
\end{align*}

\textbf{Cycle 2: }\forall i \in \{0, 1, \ldots, N - 1\}
\begin{align*}
R0.i.\text{value}, & \text{ tag } \leftarrow Z1.i.\text{value}, \text{ CHANGED}
\end{align*}
Vector Instructions: Integer, Logical and Floating Point

5.6.2 \( \text{op} = \text{isub} \)

**Cycle 1:** \( \forall i \in \{0, 1, \ldots, N - 1\} \)
\[
Z1.i.value \leftarrow R1.i.value - R2.i.value \\
\text{If underflow then PSW:14} \leftarrow 1
\]

**Cycle 2:** \( \forall i \in \{0, 1, \ldots, N - 1\} \)
\[
R0.i.value, \text{tag} \leftarrow Z1.i.value, \text{CHANGED}
\]
Vector Instructions: Integer, Logical and Floating Point

5.6.3 $\text{op} = \text{imul}$

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

**Cycle 2:** $\forall i \in \{0, 1, ..., N - 1\}$
- $R0.i.value, \text{tag} \leftarrow Z1.i.value, \text{CHANGED}$
5.6.4 \( \text{op} = \text{idiv} \)

**Cycle 1:** \( \forall i \in \{0, 1, \ldots, N - 1\} \)
- If \( \text{R2.i.value} = 0 \) then \( \text{PSW:14} \leftarrow 1 \) else
  - \( \text{Z1.i.value} \leftarrow \text{R1.i.value} / \text{R2.i.value} \)
  - If underflow then \( \text{PSW:14} \leftarrow 1 \)

**Cycle 2:** \( \forall i \in \{0, 1, \ldots, N - 1\} \)
- \( \text{R0.i.value}, \text{tag} \leftarrow \text{Z1.i.value}, \text{CHANGED} \)
Vector Instructions: Integer, Logical and Floating Point

5.6.5 op = lsl

**Cycle 1:** ∀ i in {0,1,...,N - 1}
\[ Z1.i.value \leftarrow R1.i.value \text{ asl } R2.i.value \]

**Cycle 2:** ∀ i in {0,1,...,N - 1}
\[ R0.i.value, \text{ tag} \leftarrow Z1.i.value, \text{ CHANGED} \]
Vector Instructions: Integer, Logical and Floating Point

5.6.6 \texttt{op = leql}\n
\textbf{Cycle 1}: \forall i \in \{0,1,...,N - 1\} \\
\hspace{1em} \text{if} \ R1.i.value = R2.i.value \text{ then } Z1.i.value \leftarrow \text{TRUE} \text{ else } Z1.i.value \leftarrow \text{FALSE}

\textbf{Cycle 2}: \forall i \in \{0,1,...,N - 1\} \\
\hspace{1em} 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: \( \forall i \in \{0, 1, \ldots, N - 1\} \)
If \( R1_i.value \neq R2_i.value \) then \( Z1_i.value \leftarrow TRUE \) else \( Z1_i.value \leftarrow FALSE \)

Cycle 2: \( \forall i \in \{0, 1, \ldots, N - 1\} \)
\( R0_i.value, \text{tag} \leftarrow Z1_i.value, \text{CHANGED} \)
Vector Instructions: Integer, Logical and Floating Point

5.6.8 op = lgtr

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

**Cycle 2:** \( \forall i \in \{0,1,...,N-1\} \)
\( R0.i.value, tag \leftarrow Z1.i.value, \text{CHANGED} \)
Vector Instructions: Integer, Logical and Floating Point

5.6.9 op = lgeq

**Cycle 1:** ∀ i in \{0,1,...,N - 1\}
    if R1.i.value ≥ R2.i.value then Z1.i.value ← TRUE else Z1.i.value ← FALSE

**Cycle 2:** ∀ i in \{0,1,...,N - 1\}
    R0.i.value, tag ← Z1.i.value, CHANGED
Vector Instructions: Integer, Logical and Floating Point

5.6.10 \texttt{op = liss}

\textbf{Cycle 1:} \texttt{\forall i \in \{0,1,...,N - 1\}}
\hspace{1em}
\texttt{If R1.i.value < R2.i.value then Z1.i.value \leftarrow TRUE else Z1.i.value \leftarrow FALSE}

\textbf{Cycle 2:} \texttt{\forall i \in \{0,1,...,N - 1\}}
\hspace{1em}
\texttt{R0.i.value, tag \leftarrow Z1.i.value, CHANGED}
Vector Instructions: Integer, Logical and Floating Point

5.6.11 op = ileq

**Cycle 1:** \( \forall i \in \{0,1,...,N - 1\} \)

\[
\text{If } R1.i.\text{value} \leq R2.i.\text{value} \text{ then } Z1.i.\text{value} \leftarrow \text{TRUE else } Z1.i.\text{value} \leftarrow \text{FALSE}
\]

**Cycle 2:** \( \forall i \in \{0,1,...,N - 1\} \)

\[
R0.i.\text{value}, \text{tag} \leftarrow Z1.i.\text{value}, \text{CHANGED}
\]
Vector Instructions: Integer, Logical and Floating Point

5.6.12 \texttt{op} = \texttt{iaddC}

\textbf{Cycle 1:} \forall i \in \{0,1,...,N-1\}
\begin{itemize}
  \item \texttt{Z1.i.value} $\leftarrow \texttt{R1.i.value} + \texttt{R2.i.value}$
  \item \textbf{if overflow} then \texttt{PSW.14} $\leftarrow 1$
\end{itemize}

\textbf{Cycle 2:} \forall i \in \{0,1,...,N-1\}
\begin{itemize}
  \item \textbf{if} \texttt{R3.i.value} $\neq \texttt{FALSE}$ \textbf{then} \texttt{R0.i.value}, \texttt{tag} $\leftarrow \texttt{Z1.i.value}$, \texttt{CHANGED}
  \item \textbf{else} \texttt{R0.i.tag} $\leftarrow \texttt{UNCHANGED}$
\end{itemize}
Vector Instructions: Integer, Logical and Floating Point

5.6.13 op = iandC

**Cycle 1:** \( \forall i \in \{0, 1, \ldots, N - 1\} \)
\[ Z1.i.value \leftarrow R1.i.value \&\& R2.i.value \]

**Cycle 2:** \( \forall i \in \{0, 1, \ldots, N - 1\} \)
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

5.6.14  \texttt{op = lorC}

\textbf{Cycle 1:} \( \forall i \in \{0,1,\ldots,N - 1\} \)
\[ Z1.i.value \leftarrow R1.i.value \parallel R2.i.value \]

\textbf{Cycle 2:} \( \forall i \in \{0,1,\ldots,N - 1\} \)
\begin{align*}
\text{if } R3.i.value \neq \text{FALSE} \text{ then } &R0.i.value, \ tag \leftarrow Z1.i.value, \ \text{CHANGED} \\
\text{else } &R0.i.tag \leftarrow \text{UNCHANGED}
\end{align*}
Vector Instructions: Integer, Logical and Floating Point

5.6.15 \( op = \text{ieqvC} \)

**Cycle 1:** \( \forall i \in \{0,1,...,N - 1\} \)
\[ Z1.i.value \leftarrow R1.i.value \text{ EQV } R2.i.value \]

**Cycle 2:** \( \forall i \in \{0,1,...,N - 1\} \)
- 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

5.6.16 \text{op} = \text{lsubC}

\textbf{Cycle 1:} \forall i \in \{0,1,...,N - 1\}
\begin{align*}
Z1.i\text{.value} & \leftarrow R1.i\text{.value} - R2.i\text{.value} \\
\text{if underflow then } & PSW:14 \leftarrow 1
\end{align*}

\textbf{Cycle 2:} \forall i \in \{0,1,...,N - 1\}
\begin{align*}
\text{If } R3.i\text{.value} \neq \text{FALSE then } & R0.i\text{.value}, \text{tag} \leftarrow Z1.i\text{.value}, \text{CHANGED} \\
\text{else } & R0.i\text{.tag} \leftarrow \text{UNCHANGED}
\end{align*}
Vector Instructions: Integer, Logical and Floating Point

5.6.17 op = imulC

**Cycle 1: \( \forall i \in \{0,1,...,N - 1\} \)**
\[ Z1.i.value \leftarrow R1.i.value \times R2.i.value \]
if overflow then PSW:14 \( \leftarrow 1 \)

**Cycle 2: \( \forall i \in \{0,1,...,N - 1\} \)**
if \( R3.i.value \neq FALSE \) 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

5.6.18 op = idivC

Cycle 1: ∀ i in \{0,1,...,N - 1\}
    if R2.i.value = 0 then PSW:14 ← 1 else
    Z1.i.value ← R1.i.value / R2.i.value
    if underflow then PSW:14 ← 1

Cycle 2: ∀ i in \{0,1,...,N - 1\}
    if R3.i.value ≠ FALSE then R0.i.value, tag ← Z1.i.value, CHANGED
    else R0.i.tag ← UNCHANGED
Vector Instructions: Integer, Logical and Floating Point

5.6.19 op = iaslC

**Cycle 1:** \( \forall i \in \{0,1,...,N - 1\} \)
\[ Z_{1,i}.value \leftarrow R_{1,i}.value \text{ asl } R_{2,i}.value \]

**Cycle 2:** \( \forall i \in \{0,1,...,N - 1\} \)
- If \( R_{3,i}.value \neq \text{FALSE} \) then \( R_{0,i}.value, \text{ tag} \leftarrow Z_{1,i}.value, \text{ CHANGED} \)
- else \( R_{0,i}.tag \leftarrow \text{UNCHANGED} \)
Vector Instructions: Integer, Logical and Floating Point

5.6.20 op = ieqlC

Cycle 1: ∀ i in {0,1,...,N - 1}
     If R1.i.value = R2.i.value then Z1.i.value ← TRUE else Z1.i.value ← FALSE

Cycle 2: ∀ i in {0,1,...,N - 1}
     If R3.i.value ≠ FALSE then R0.i.value, tag ← Z1.i.value, CHANGED
     else R0.i.tag ← UNCHANGED
Vector Instructions: Integer, Logical and Floating Point

5.6.21 op = ineqC

Cycle 1: ∀ i in {0, 1, ..., N - 1}
   If R1.i.value ≠ R2.i.value then Z1.i.value ← TRUE else Z1.i.value ← FALSE

Cycle 2: ∀ i in {0, 1, ..., N - 1}
   If R3.i.value ≠ FALSE then R0.i.value, tag ← Z1.i.value, CHANGED
   else R0.i.tag ← UNCHANGED
Vector Instructions: Integer, Logical and Floating Point

5.6.22  op = igtrC

Cycle 1: \( \forall i \in \{0,1,...N-1\} \)
if \( R1.i.value > R2.i.value \) then \( Z1.i.value \leftarrow \text{TRUE} \) else \( Z1.i.value \leftarrow \text{FALSE} \)

Cycle 2: \( \forall i \in \{0,1,...N-1\} \)
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

5.6.23 \( op = \text{ lgeqC} \)

**Cycle 1:** \( \forall \ i \ in \{0,1,...,N - 1\} \)

If \( R1.i \cdot \text{value} \geq R2.i \cdot \text{value} \) then \( Z1.i \cdot \text{value} \leftarrow \text{TRUE} \) else \( Z1.i \cdot \text{value} \leftarrow \text{FALSE} \)

**Cycle 2:** \( \forall \ i \ in \{0,1,...,N - 1\} \)

If \( R3.i \cdot \text{value} \neq \text{FALSE} \) then \( R0.i \cdot \text{value}, \ tag \leftarrow Z1.i \cdot \text{value}, \ \text{CHANGED} \)
else \( R0.i \cdot \tag \leftarrow \text{UNCHANGED} \)
Vector Instructions: Integer, Logical and Floating Point

5.6.24 op = lissC

Cycle 1: ∀ i in \{0,1,...N - 1\}
   If R1.i.value < R2.i.value then Z1.i.value ← TRUE else Z1.i.value ← FALSE

Cycle 2: ∀ i in \{0,1,...N - 1\}
   If R3.i.value ≠ FALSE then R0.i.value, tag ← Z1.i.value, CHANGED
   else R0.i.tag ← UNCHANGED
Vector Instructions: Integer, Logical and Floating Point

5.6.25 \textit{op} = \textit{ileqC}

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

\textbf{Cycle 2: }\forall i \in \{0,1,...,N-1\}
\hspace{1em} \text{If } R3.i.value \neq \text{FALSE} \text{ then } R0.i.value, \text{ tag } \leftarrow Z1.i.value, \text{ CHANGED}
\hspace{1em} \text{else } R0.i.tag \leftarrow \text{UNCHANGED}
Vector Instructions: Integer, Logical and Floating Point

5.6.26 \( \text{op} = \text{faddC} \)

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

**Cycle 2:** \( \forall i \in \{0, 1, \ldots, N - 1\} \)
- 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

5.6.27 \( \text{op} = \text{fsubC} \)

**Cycle 1:** \( \forall i \in \{0, 1, \ldots, N - 1\} \)

\[ Z1.i.\text{value} \leftarrow R1.i.\text{value} - R2.i.\text{value} \]

if underflow as defined by IEEE Std. 754 then PSW.14 \( \leftarrow 1 \)

**Cycle 2:** \( \forall i \in \{0, 1, \ldots, N - 1\} \)

if \( R3.i.\text{value} \neq 0 \) then \( R0.i.\text{value}, \text{tag} \leftarrow Z1.i.\text{value}, \text{CHANGED} \)

else \( R0.i.\text{tag} \leftarrow \text{UNCHANGED} \)
Vector Instructions: Integer, Logical and Floating Point

5.6.28 op = fmulC

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

**Cycle 2:** \( \forall i \in \{0, 1, \ldots, N - 1\} \)
\[
\text{If } R3.i.value \neq 0 \text{ then } R0.i.value, \text{ tag } \leftarrow Z1.i.value, \text{ CHANGED}
\]
else \(R0.i.tag \leftarrow \text{UNCHANGED}\)
Vector Instructions: Integer, Logical and Floating Point

5.6.29 op = fdivC

Cycle 1: \( \forall i \in \{0, 1, \ldots, N - 1\} \)
- 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: \( \forall i \in \{0, 1, \ldots, N - 1\} \)
- 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

5.6.30 $\text{op} = \text{fleqC}$

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

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

5.6.31 \text{op} = \text{flssC}

\textbf{Cycle 1:} \forall i \in \{0, 1, \ldots, \text{N} - 1\}
\begin{align*}
&\text{If } R1.i.value < R2.i.value \text{ then } Z1.i.value \leftarrow \text{TRUE} \\
&\text{else } Z1.i.value \leftarrow \text{FALSE}
\end{align*}

\textbf{Cycle 2:} \forall i \in \{0, 1, \ldots, \text{N} - 1\}
\begin{align*}
&\text{If } R3.i.value \neq 0 \text{ then } R0.i.value, \text{ tag} \leftarrow Z1.i.value, \text{ CHANGED} \\
&\text{else } R0.i.tag \leftarrow \text{UNCHANGED}
\end{align*}
Vector Instructions: Integer, Logical and Floating Point

5.6.32 op = fgeqC

**Cycle 1:** ∀ i in {0, 1,...N - 1}
- If R1.i.value ≥ R2.i.value then Z1.i.value ← TRUE
- else Z1.i.value ← FALSE

**Cycle 2:** ∀ i in {0, 1,...N - 1}
- If R3.i.value ≠ 0 then R0.i.value, tag ← Z1.i.value, CHANGED
- else R0.i.tag ← UNCHANGED
5.6.33 \texttt{op} = \texttt{fgtrC}

\textbf{Cycle 1:} \forall i \in \{0,1,\ldots,N-1\}
  \begin{align*}
    \text{if } R1.i.value > R2.i.value \text{ then } & Z1.i.value \leftarrow \text{TRUE} \\
    \text{else } & Z1.i.value \leftarrow \text{FALSE}
  \end{align*}

\textbf{Cycle 2:} \forall i \in \{0,1,\ldots,N-1\}
  \begin{align*}
    \text{if } R3.i.value \neq 0 \text{ then } & R0.i.value, \text{ tag } \leftarrow Z1.i.value, \text{ CHANGED} \\
    \text{else } & R0.i.tag \leftarrow \text{UNCHANGED}
  \end{align*}
Vector Instructions: Integer, Logical and Floating Point

5.6.34 \text{op} = \text{fneqC}

\begin{enumerate}
\item \textbf{Cycle 1:} \forall i \in \{0,1,...,N - 1\}
    \begin{enumerate}
    \item If R1.i.value \neq R2.i.value then \text{Z1.i.value} \leftarrow \text{TRUE}
    \item else \text{Z1.i.value} \leftarrow \text{FALSE}
    \end{enumerate}
\item \textbf{Cycle 2:} \forall i \in \{0,1,...,N - 1\}
    \begin{enumerate}
    \item if R3.i.value \neq 0 then R0.i.value, tag \leftarrow Z1.i.value, \text{CHANGED}
    \item else R0.i.tag \leftarrow \text{UNCHANGED}
    \end{enumerate}
\end{enumerate}
Vector Instructions: Integer, Logical and Floating Point

5.6.35 \textbf{op = feglC}

\textbf{Cycle 1:} \( \forall \ i \ in \ \{0,1,\ldots,N - 1\} \)
\begin{enumerate}
\item \textbf{if} R1.i.value = R2.i.value \textbf{then} Z1.i.value \leftarrow \textbf{TRUE}
\item \textbf{else} Z1.i.value \leftarrow \textbf{FALSE}
\end{enumerate}

\textbf{Cycle 2:} \( \forall \ i \ in \ \{0,1,\ldots,N - 1\} \)
\begin{enumerate}
\item \textbf{if} R3.i.value \neq 0 \textbf{then} R0.i.value, tag \leftarrow Z1.i.value, \textbf{CHANGED}
\item \textbf{else} R0.i.tag \leftarrow \textbf{UNCHANGED}
\end{enumerate}
Vector Instructions: Integer, Logical and Floating Point

5.6.36 \( \text{op} = \text{fadd} \)

**Cycle 1:** \( \forall i \in \{0,1,...,N - 1\} \)

\[ Z1.i.\text{value} \leftarrow R1.i.\text{value} + R2.i.\text{value} \]

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

**Cycle 2:** \( \forall i \in \{0,1,...,N - 1\} \)

\[ R0.i.\text{value}, \text{tag} \leftarrow Z1.i.\text{value}, \text{CHANGED} \]
Vector Instructions: Integer, Logical and Floating Point

5.6.37 op = fsub

**Cycle 1:** \( \forall i \in \{0,1,...,N - 1\} \)

\[ Z1.i.value \leftarrow R1.i.value - R2.i.value \]

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

**Cycle 2:** \( \forall i \in \{0,1,...,N - 1\} \)

\[ R0.i.value, tag \leftarrow Z1.i.value, CHANGED \]
Vector Instructions: Integer, Logical and Floating Point

5.6.38 \texttt{op = fmul}

\textbf{Cycle 1: }\forall i \in \{0, 1, \ldots, N - 1\}
\[ Z1.i.value \leftarrow R1.i.value \times R2.i.value \]
\text{If overflow as defined by IEEE Std. 754 then PSW:14} \leftarrow 1

\textbf{Cycle 2: }\forall i \in \{0, 1, \ldots, N - 1\}
\[ R0.i.value, \text{ tag} \leftarrow Z1.i.value, \text{ CHANGED} \]
Vector Instructions: Integer, Logical and Floating Point

5.6.39 \texttt{op} = \texttt{fdiv}

\textbf{Cycle 1:} \( \forall i \in \{0,1,\ldots, N - 1\} \)
- 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 \( PSW:14 \leftarrow 1 \)

\textbf{Cycle 2:} \( \forall i \in \{0,1,\ldots, N - 1\} \)
- \( R0.i.value, \, tag \leftarrow Z1.i.value, \, \text{CHANGED} \)
Vector Instructions: Integer, Logical and Floating Point

5.6.40 \texttt{op = feql}

\textbf{Cycle 1:} \( \forall i \in \{0, 1, \ldots, N - 1\} \)
  \begin{align*}
    & \text{if } R1.i.value = R2.i.value \text{ then } Z1.i.value \leftarrow \text{TRUE} \\
    & \text{else } Z1.i.value \leftarrow \text{FALSE}
  \end{align*}

\textbf{Cycle 2:} \( \forall i \in \{0, 1, \ldots, N - 1\} \)
  \begin{align*}
    & R0.i.value, \text{ tag } \leftarrow Z1.i.value, \text{ CHANGED}
  \end{align*}
Vector Instructions: Integer, Logical and Floating Point

5.6.41 op = fneq

**Cycle 1:** ∀ i in \( \{0, 1, \ldots, N - 1\} \)
  - if \( R1.i \).value ≠ \( R2.i \).value then \( Z1.i \).value ← TRUE
  - else \( Z1.i \).value ← FALSE

**Cycle 2:** ∀ i in \( \{0, 1, \ldots, N - 1\} \)
  - \( R0.i \).value, tag ← \( Z1.i \).value, CHANGED
**Vector Instructions: Integer, Logical and Floating Point**

5.6.42 \( \text{op} = \text{fgtr} \)

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

**Cycle 2:** \( \forall \ i \ \text{in} \ \{0, 1, ..., N - 1\} \)
   - \( R0.i.value, \ \text{tag} \leftarrow Z1.i.value, \text{CHANGED} \)
Vector Instructions: Integer, Logical and Floating Point

5.6.43 op = fgeq

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

**Cycle 2:** $\forall i \in \{0, 1, ..., N - 1\}$
- $R0.i.value, \ tag \leftarrow Z1.i.value, \ CHANGED$
Vector Instructions: Integer, Logical and Floating Point

5.6.44 \textbf{op} = \texttt{flss}

\textbf{Cycle 1}: \( \forall i \in \{0, 1, \ldots, N - 1\} \)
\hspace{1em} If \( R1.i.value < R2.i.value \) then \( Z1.i.value \leftarrow \text{TRUE} \)
\hspace{1em} else \( Z1.i.value \leftarrow \text{FALSE} \)

\textbf{Cycle 2}: \( \forall i \in \{0, 1, \ldots, N - 1\} \)
\hspace{1em} \( R0.i.value, \text{tag} \leftarrow Z1.i.value, \text{CHANGED} \)
Vector Instructions: Integer, Logical and Floating Point

5.6.45 op = fleq

Cycle 1: ∀ i in {0, 1, ..., N - 1}
   If R1.i.value ≤ R2.i.value then Z1.i.value ← TRUE
   else Z1.i.value ← FALSE

Cycle 2: ∀ i in {0, 1, ..., N - 1}
   R0.i.value, tag ← Z1.i.value, CHANGED
5.7 Load and Store Instructions

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

Format:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>11</th>
<th>12</th>
<th>16</th>
<th>17</th>
<th>21</th>
<th>22</th>
<th>26</th>
<th>27</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>RL</td>
<td>LSOP</td>
<td>OP1</td>
<td>OP2</td>
<td>R0</td>
<td>R1</td>
<td>RL2</td>
<td>RL3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
<table>
<thead>
<tr>
<th>LSOP</th>
<th>FIFO</th>
<th>data size/type</th>
<th>exec</th>
<th>sign</th>
<th>extension</th>
<th>LSOP</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L8i</td>
<td>r0.input</td>
<td>8-bit integer</td>
<td>IEU</td>
<td>no</td>
<td></td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>L8ix</td>
<td>r0.input</td>
<td>8-bit integer</td>
<td>IEU</td>
<td>yes</td>
<td></td>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>L16i</td>
<td>r0.input</td>
<td>16-bit integer</td>
<td>IEU</td>
<td>no</td>
<td></td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>L16ix</td>
<td>r0.input</td>
<td>16-bit integer</td>
<td>IEU</td>
<td>yes</td>
<td></td>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>L32i</td>
<td>r0.input</td>
<td>32-bit integer</td>
<td>IEU</td>
<td>no</td>
<td></td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>L32ix</td>
<td>r0.input</td>
<td>32-bit integer</td>
<td>IEU</td>
<td>yes</td>
<td></td>
<td>0110</td>
<td></td>
</tr>
<tr>
<td>L64i</td>
<td>r0.input</td>
<td>64-bit integer</td>
<td>IEU</td>
<td>n/a</td>
<td></td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>L64f</td>
<td>f0.input</td>
<td>32-bit floating</td>
<td>FEU</td>
<td>n/a</td>
<td></td>
<td>0010</td>
<td></td>
</tr>
</tbody>
</table>

| Store |       |                |      |      |           |      |          |
| 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:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>OP</th>
<th>OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump</td>
<td>0000</td>
<td>0000: unconditional Jump</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JumpITy</td>
<td>0000</td>
<td>1101: Jump if Integer condition bit is True</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JumpIfy</td>
<td>0000</td>
<td>1100: Jump if Integer condition bit is False</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JumpFTy</td>
<td>0000</td>
<td>1111: Jump if Floating condition bit is True</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JumpFFy</td>
<td>0000</td>
<td>1110: Jump if Floating condition bit is False</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JumpITn</td>
<td>0000</td>
<td>1001: Jump if Integer condition bit is True</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JumpIFn</td>
<td>0000</td>
<td>1000: Jump if Integer condition bit is False</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JumpFTn</td>
<td>0000</td>
<td>1011: Jump if Floating condition bit is True</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JumpFFn</td>
<td>0000</td>
<td>1010: Jump if Floating condition bit is False</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNI r0</td>
<td>0001</td>
<td>0000: Jump on stream count Not zero; Input FIFO r0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNI r1</td>
<td>0001</td>
<td>0001: Jump on stream count Not zero; Input FIFO r1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNO r0</td>
<td>0001</td>
<td>0010: Jump on stream count Not zero; Output FIFO r0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNO r1</td>
<td>0001</td>
<td>0011: Jump on stream count Not zero; Output FIFO r1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNI f0</td>
<td>0001</td>
<td>0100: Jump on stream count Not zero; Input FIFO f0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNI f1</td>
<td>0001</td>
<td>0101: Jump on stream count Not zero; Input FIFO f1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNO f0</td>
<td>0001</td>
<td>0110: Jump on stream count Not zero; Output FIFO f0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNO f1</td>
<td>0001</td>
<td>0111: Jump on stream count Not zero; Output FIFO f1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNI v0</td>
<td>0001</td>
<td>1000: Jump on stream count Not zero; Input FIFO v0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNI v1</td>
<td>0001</td>
<td>1001: Jump on stream count Not zero; Input FIFO v1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNO v0</td>
<td>0001</td>
<td>1010: Jump on stream count Not zero; Output FIFO v0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNO v1</td>
<td>0001</td>
<td>1011: Jump on stream count Not zero; Output FIFO v1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call</td>
<td>0010</td>
<td>0000: subroutine Call</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECall</td>
<td>0010</td>
<td>0001: Entry Call</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

    JumpFn, JumpFy: If CCI then JC ← TRUE
    JumpFm, JumpFy: 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) Isl 2
5.9 Special Instructions

5.9.1 JumpI

Mnemonic: JumpI R1

Format:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>11</th>
<th>12</th>
<th>16</th>
<th>17</th>
<th>21</th>
<th>22</th>
<th>26</th>
<th>27</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description: This instruction is an unconditional jump to a target address designated by the register named in the R1 field.

Cycle Description:

Cycle 1: PC ← R1
Special Instructions

5.9.2 Calll

**Mnemonic:** Calll R1

**Format:**

```
  0 1 2 3 4  11 12 16 17 21 22 26 27 31
  1 0 0 0 1 0 1 0 1  R1
```

**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 ← PC

PC ← R1
Special Instructions

5.9.3 EReturn

Mnemonic: EReturn

Format:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>11</th>
<th>12</th>
<th>16</th>
<th>17</th>
<th>21</th>
<th>22</th>
<th>26</th>
<th>27</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 ← <Protected Stack Area>
   PTP ← <Protected Stack Area>
Special Instructions

5.9.4 Streaming to and from the IEU and FEU

Mnemonic: SOP R0, R1, RL2, RL3

Format:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RL</td>
<td>SOP</td>
<td>R0</td>
<td>R1</td>
<td>RL2</td>
<td>RL3</td>
<td></td>
</tr>
</tbody>
</table>

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 :: φ, φ, φ, φ, 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
  StopII, StopFl:
    with input FIFO specified by R0 do
      FIFO.consume_count ← 0
      flush FIFO
  StopIO, StopFO:
    with output FIFO specified by R0 do
      complete pending memory writes
      FIFO.consume_count ← 0
<table>
<thead>
<tr>
<th>SOP</th>
<th>operation</th>
<th>data_type</th>
<th>exec</th>
<th>sign</th>
<th>extension</th>
<th>SOP</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream In</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sin8i</td>
<td>load</td>
<td>8-bit integer</td>
<td>IEU</td>
<td>no</td>
<td></td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>Sin8ix</td>
<td>load</td>
<td>8-bit integer</td>
<td>IEU</td>
<td>yes</td>
<td></td>
<td>01000000</td>
<td></td>
</tr>
<tr>
<td>Sin16i</td>
<td>load</td>
<td>16-bit integer</td>
<td>IEU</td>
<td>no</td>
<td></td>
<td>00010000</td>
<td></td>
</tr>
<tr>
<td>Sin16ix</td>
<td>load</td>
<td>16-bit integer</td>
<td>IEU</td>
<td>yes</td>
<td></td>
<td>01010000</td>
<td></td>
</tr>
<tr>
<td>Sin32i</td>
<td>load</td>
<td>32-bit integer</td>
<td>IEU</td>
<td>no</td>
<td></td>
<td>00100000</td>
<td></td>
</tr>
<tr>
<td>Sin32ix</td>
<td>load</td>
<td>32-bit integer</td>
<td>IEU</td>
<td>yes</td>
<td></td>
<td>01100000</td>
<td></td>
</tr>
<tr>
<td>Sin64i</td>
<td>load</td>
<td>64-bit integer</td>
<td>IEU</td>
<td>n/a</td>
<td></td>
<td>00110000</td>
<td></td>
</tr>
<tr>
<td>Sin64f</td>
<td>load</td>
<td>64-bit floating</td>
<td>FEU</td>
<td>n/a</td>
<td></td>
<td>11000000</td>
<td></td>
</tr>
<tr>
<td>Stream out</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sout8i</td>
<td>store</td>
<td>8-bit integer</td>
<td>IEU</td>
<td>n/a</td>
<td></td>
<td>10000000</td>
<td></td>
</tr>
<tr>
<td>Sout16i</td>
<td>store</td>
<td>16-bit integer</td>
<td>IEU</td>
<td>n/a</td>
<td></td>
<td>10010000</td>
<td></td>
</tr>
<tr>
<td>Sout32i</td>
<td>store</td>
<td>32-bit integer</td>
<td>IEU</td>
<td>n/a</td>
<td></td>
<td>10100000</td>
<td></td>
</tr>
<tr>
<td>Sout64i</td>
<td>store</td>
<td>64-bit integer</td>
<td>IEU</td>
<td>n/a</td>
<td></td>
<td>10110000</td>
<td></td>
</tr>
<tr>
<td>Sout32f</td>
<td>store</td>
<td>32-bit floating</td>
<td>FEU</td>
<td>n/a</td>
<td></td>
<td>11100000</td>
<td></td>
</tr>
<tr>
<td>Sout64f</td>
<td>store</td>
<td>64-bit floating</td>
<td>FEU</td>
<td>n/a</td>
<td></td>
<td>11110000</td>
<td></td>
</tr>
<tr>
<td>Stop streaming</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StopAll</td>
<td>Stop all Streaming operations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01110010</td>
<td></td>
</tr>
<tr>
<td>StopI</td>
<td>Stop Integer Input Streaming operations on FIFO specified by R0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00000010</td>
<td></td>
</tr>
<tr>
<td>StopIO</td>
<td>Stop Integer Output Streaming operations on FIFO specified by R0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00010010</td>
<td></td>
</tr>
<tr>
<td>StopFI</td>
<td>Stop Floating Input Streaming operations on FIFO specified by R0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00100010</td>
<td></td>
</tr>
<tr>
<td>StopFO</td>
<td>Stop Floating Output Streaming operations on FIFO specified by R0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00110010</td>
<td></td>
</tr>
<tr>
<td>FIFO selection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If data type = integer then</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if R0 = 1 then FIFO = r0 else FIFO = r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else if R0 = 1 then FIFO = f0 else FIFO = f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Special Instructions

5.9.5 Streaming to and from the VEU

Mnemonic: VSOP R0, R1, RL2, RL3

Format:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>11</th>
<th>12</th>
<th>16</th>
<th>17</th>
<th>21</th>
<th>22</th>
<th>26</th>
<th>27</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>RL</td>
<td>VSOP</td>
<td>R0</td>
<td>R1</td>
<td>RL2</td>
<td>RL3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 :: φ, φ, φ, 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 qualifier_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:: φ, φ, φ, φ, 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
<table>
<thead>
<tr>
<th>VSOP</th>
<th>operation</th>
<th>data_type</th>
<th>exec unit</th>
<th>sign extension</th>
<th>SOP Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream In</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSIn8i</td>
<td>load</td>
<td>8-bit integer</td>
<td>VEU</td>
<td>no</td>
<td>000000001</td>
</tr>
<tr>
<td>VSIn8ix</td>
<td>load</td>
<td>8-bit integer</td>
<td>VEU</td>
<td>yes</td>
<td>01000001</td>
</tr>
<tr>
<td>VSIn16i</td>
<td>load</td>
<td>16-bit integer</td>
<td>VEU</td>
<td>no</td>
<td>000100001</td>
</tr>
<tr>
<td>VSIn16ix</td>
<td>load</td>
<td>16-bit integer</td>
<td>VEU</td>
<td>yes</td>
<td>01010001</td>
</tr>
<tr>
<td>VSIn32i</td>
<td>load</td>
<td>32-bit integer</td>
<td>VEU</td>
<td>no</td>
<td>00100001</td>
</tr>
<tr>
<td>VSIn32ix</td>
<td>load</td>
<td>32-bit integer</td>
<td>VEU</td>
<td>yes</td>
<td>01100001</td>
</tr>
<tr>
<td>VSIn64i</td>
<td>load</td>
<td>64-bit integer</td>
<td>VEU</td>
<td>n/a</td>
<td>00110001</td>
</tr>
<tr>
<td>VSIn64f</td>
<td>load</td>
<td>64-bit floating</td>
<td>VEU</td>
<td>n/a</td>
<td>11000001</td>
</tr>
<tr>
<td>VSIn8f</td>
<td>load</td>
<td>64-bit floating</td>
<td>VEU</td>
<td>n/a</td>
<td>11010001</td>
</tr>
<tr>
<td>VSIn1b</td>
<td>load</td>
<td>1-bit boolean</td>
<td>VEU</td>
<td>n/a</td>
<td>00010001</td>
</tr>
</tbody>
</table>

| Stream out |
| 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     |

Stop streaming
- StopAll: Stop all Streaming operations
- StopVI: Stop Vector Input Streaming operations on FIFO specified by R0
- StopVO: Stop Vector Output Streaming operations on FIFO specified by R0
Special Instructions

5.9.6 ASSERT

Mnemonic: ASSERT (R1 ≥ RL2) ≤ RL3

Format:

\[
\begin{array}{cccccccccc}
  & 0 & 1 & 2 & 3 & 4 & 11 & 12 & 16 & 17 & 21 & 22 & 26 & 27 & 31 \\
 1 & 0 & RL & 1 & 0 & 0 & 0 & 1 & 0 & 0 & R1 & RL2 & RL3 \\
\end{array}
\]

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 ≥ RL2 then exception:: Assert Fault

Cycle 2: if not R1 ≤ RL3 then exception:: Assert Fault
Special Instructions

5.9.7 FASSERT

**Mnemonic:** FASSERT (R1 ≥ R2) ≤ R3

**Format:**

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

**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 ≥ R2 then exception:: Assert Fault
- **Cycle 2:** If not R1 ≤ R3 then exception:: Assert Fault
Special Instructions

5.9.8 FLDMOV

Mnemonic: FLDMOV R0 := R1, RL2, RL3

Format:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>11</th>
<th>12</th>
<th>16</th>
<th>17</th>
<th>21</th>
<th>22</th>
<th>26</th>
<th>27</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>RL</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>R0</td>
<td>R1</td>
<td>RL2</td>
<td>RL3</td>
</tr>
</tbody>
</table>

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 ← R1 lsl RL2

Cycle 2: R0 ← X1 lsr RL3
Special Instructions

5.9.9 FLDMOVX

Mnemonic: FLDMOVX R0 := R1, RL2, RL3

Format:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>11</th>
<th>12</th>
<th>16</th>
<th>17</th>
<th>21</th>
<th>22</th>
<th>26</th>
<th>27</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td>RL</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R0</td>
<td>R1</td>
<td>RL2</td>
<td>RL3</td>
<td></td>
</tr>
</tbody>
</table>

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 ← R1 isl RL2

Cycle 2: R0 ← X1 asr RL3
Special Instructions

5.9.10 FFB

Mnemonic: FFB R0 := R1

Format:

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 11 & 12 & 16 & 17 & 21 & 22 & 26 & 27 & 31 \\
1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & R0 & R1 \\
\end{array}
\]

Description: FFB finds the first bit that is different from the sign bit in the R1 value, starting from the left. It stores into R0 the bit number of the bit found. If all the bits in the word are the same, the value 0 is stored.

Cycle Description:

Cycle 1: \( X1 \leftarrow 0 \)
\[
i \leftarrow 1 \\
\text{while } i \leq \text{sizeof}(R1) - 1 \text{ and } X1 = 0 \\
\quad \text{if } R1:i \neq R1:0 \text{ then } X1 \leftarrow i \\
\quad i \leftarrow i + 1
\]

Cycle 2: \( R0 \leftarrow X1 \)
Special Instructions

5.9.11 CVTIF

Mnemonic: CVTIF R0 := R1

Format:

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

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 ← int_to_float (R1)
Special Instructions

5.9.12 CVTFI

Mnemonic: CVTFI R0 := R1

Format:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>11</th>
<th>12</th>
<th>16</th>
<th>17</th>
<th>21</th>
<th>22</th>
<th>26</th>
<th>27</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>R0</td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 ← float_to_int (R1)
Special Instructions

5.9.13 TIF

Mnemonic: TIF R0 := R1

Format:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>11</th>
<th>12</th>
<th>16</th>
<th>17</th>
<th>21</th>
<th>22</th>
<th>26</th>
<th>27</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>R0</td>
<td>R1</td>
<td></td>
</tr>
</tbody>
</table>

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 ← R1
Special Instructions

5.9.14 TFI

Mnemonic: TFI R0 := R1

Format:

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

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:

\[
\begin{array}{cccccccccccc}
1 & 0 & \text{RL} & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & \text{R0} & \text{RL3}
\end{array}
\]

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: \( \forall i \in \{0, 1, \ldots, N - 1\} \)
\[ \text{R0.i.value, tag} \leftarrow \text{RL3, CHANGED} \]
Special Instructions

5.9.16 TIVx

Mnemonic: TIVx R0 := RL3

Format:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>11</th>
<th>12</th>
<th>16</th>
<th>17</th>
<th>21</th>
<th>22</th>
<th>26</th>
<th>27</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>RL</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>R0</td>
<td></td>
<td>RL3</td>
</tr>
</tbody>
</table>

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: ∀ i in {0,1,...,N - 1}
R0[i].value, tag ← sign_extend (RL3), CHANGED
Special Instructions

5.9.17   TFV

Mnemonic: TFV R0 := RL3

Format:

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11  4  3  2  1  0

   1 0
   RL 00110100 R0   RL3
```

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: ∀ i in {0, 1, ..., N - 1}
          \( R0.i.value, tag \leftarrow RL3, CHANGED \)
Special Instructions

5.9.18 LLH

Mnemonic: LLH R0 := 16_bit_constant

Format:

```
 0 1 2 3 4 11 12 16 17 21 22 26 27 31
1 0 0 0 1 1 0 1 0 0 R0
```

= 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 ← 16_bit_constant
Special Instructions

5.9.19  SLL

Mnemonic: SLL R0 := 16_bit_constant

Format:

```
  0  1  2  3  4  11 12  16  17  21 22  26 27  31
```

```
0 1  0 1 1 0 1 0 0  R0
```

\[ \text{= 16 bit constant} \]

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 lsl 16

\[ X1_{16-31} \leftarrow \text{16_bit_constant} \]

Cycle 2: R0 \( \leftarrow \) X1
Special Instructions

5.9.20 ReadPCW

Mnemonic: ReadPCW R0

Format:

```
0 1 2 3 4 11 12 16 17 21 22 26 27 31
  1 0 1 1 0 0 0 1 0 1       R0
```

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

Cycle Description:

Cycle 1: R0 ← PCW
Special Instructions

5.9.21 WritePCW

Mnemonic: WritePCW R1

Format:

```
0 1 2 3 4 11 12 16 17 21 22 26 27 31
1 0 1 1 0 1 0 1 0 1 R1
```

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

Cycle Description:

Cycle 1: PCW ← R1
Special Instructions

5.9.22 Consumel

Mnemonic: Consumel

Format:

```
 0 1 2 3 4 11 12 16 17 21 22 26 27 31
 1 0 0 1 0 1 0 1 1
```

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

Cycle Description:

Cycle 1: \( CC_1 \leftarrow CC_i \)
Special Instructions

5.9.23 ConsumeF

**Mnemonic:** ConsumeF

**Format:**

```
   0 1 2 3 4 11 12 16 17 21 22 26 27 31
   1 0 1 1 1 0 0 1 0 1
```

**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:

```
0 1 2 3 4 11 12 16 17 21 22 26 27 31
1 0 0 1 0 0 0 1 0 1
```

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:

```
0 1 2 3 4 11 12 16 17 21 22 26 27 31
1 0 1 0 0 0 0 0 1 0  R1  L2  L3
```

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.]
Special Instructions

5.9.26 FLoadM

**Mnemonic:** FLoadM R1, L2, L3

**Format:**

```
0 1 2 3 4 11 12 16 17 21 22 26 27 31
1 0 1 0 1 0 0 0 1 0 R1 L2 L3
```

**Description:** FLoadM loads 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. RLR2 and RLR3 always specify literals. The storage location is specified by the contents of the register specified by R1. 1 \( \leq \) L2 \( \leq \) L3 \( \leq \) 31.

**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:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>11</th>
<th>12</th>
<th>16</th>
<th>17</th>
<th>21</th>
<th>22</th>
<th>26</th>
<th>27</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>R1</td>
<td>L2</td>
<td>L3</td>
</tr>
</tbody>
</table>

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 StoreM

Mnemonic: StoreM R1, L2, L3

Format:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>11</th>
<th>12</th>
<th>16</th>
<th>17</th>
<th>21</th>
<th>22</th>
<th>26</th>
<th>27</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>R1</td>
<td>L2</td>
<td>L3</td>
<td></td>
</tr>
</tbody>
</table>

Description: StoreM stores 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 ≤ L3 ≤ 31.

Cycle Description:

Memory Cycle: The contents of IEU 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 + i/8, ..., R1 + (L3 - L2)i/8, where i is the size of the IEU registers in bits.]
Special Instructions

5.9.29 FStoreM

Mnemonic: FStoreM R1, L2, L3

Format:

```
  0 1 2 3 4  11 12 16 17 21 22 26 27 31
  1 0 1 0 1 0 1 0  R1   L2   L3
```

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 ≤ L3 ≤ 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:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |

R1  L2  L3

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 ≤ L3 ≤ 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.]
Special Instructions

5.9.31 LoadFifoII, LoadFifoFl, LoadFifoVI

Mnemonic: LoadFifoII R0, R1
LoadFifoFl R0, R1
LoadFifoVI R0, R1

Format:

<table>
<thead>
<tr>
<th>0 1 2 3 4</th>
<th>11 12 16 17 21 22 26 27 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>OPCODE</td>
</tr>
<tr>
<td></td>
<td>R0</td>
</tr>
<tr>
<td></td>
<td>R1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000011</td>
<td>LoadFifoII</td>
</tr>
<tr>
<td>00100011</td>
<td>LoadFifoFl</td>
</tr>
<tr>
<td>01000011</td>
<td>LoadFifoVI</td>
</tr>
</tbody>
</table>

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 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:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>11</th>
<th>12</th>
<th>16</th>
<th>17</th>
<th>21</th>
<th>22</th>
<th>26</th>
<th>27</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>OPCODE</td>
<td>R0</td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010011</td>
<td>LoadFifoIO</td>
</tr>
<tr>
<td>00110011</td>
<td>LoadFifoFO</td>
</tr>
<tr>
<td>01010011</td>
<td>LoadFifoVO</td>
</tr>
</tbody>
</table>

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.
Special Instructions

5.9.33 StoreFifoII, StoreFifoFI, StoreFifoVI

Mnemonic: StoreFifoII R0, R1
            StoreFifoFI R0, R1
            StoreFifoVI R0, R1

Format:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>11</th>
<th>12</th>
<th>16</th>
<th>17</th>
<th>21</th>
<th>22</th>
<th>26</th>
<th>27</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>OPCODE</td>
<td>R0</td>
<td></td>
<td></td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000011</td>
<td>StoreFifoII</td>
</tr>
<tr>
<td>10100011</td>
<td>StoreFifoFI</td>
</tr>
<tr>
<td>11000011</td>
<td>StoreFifoVI</td>
</tr>
</tbody>
</table>

Description: These instructions store the specified FIFO state to the address specified by R1. The amount and format of this information is implementation dependent.

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.
Special Instructions

5.9.34 StoreFifoO, StoreFifoFO, StoreFifoVO

Mnemonic: StoreFifoO R0, R1
           StoreFifoFO R0, R1
           StoreFifoVO R0, R1

Format:

```
 0 1 2 3 4 11 12 16 17 21 22 26 27 31
 1 0   OPCODE   R0   R1
```

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>10010011</td>
<td>StoreFifoO</td>
</tr>
<tr>
<td>10110011</td>
<td>StoreFifoFO</td>
</tr>
<tr>
<td>11010011</td>
<td>StoreFifoVO</td>
</tr>
</tbody>
</table>

Description: These instructions store the specified output FIFO state to the address specified by R1. The amount and format of this information is implementation dependent.

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:

<table>
<thead>
<tr>
<th>0 1 2 3 4</th>
<th>11 12 16 17 21 22 26 27 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 0 1 0 1</td>
<td>R1</td>
</tr>
</tbody>
</table>

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:

```
  0 1 2 3 4 11 12 16 17 21 22 26 27 31
   1 0 1 0 0 1 0 1 0 1
```

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.
Special Instructions

5.9.37 SwapCTX

Mnemonic: SwapCTX R1

Format:

```
0  1  2  3  4  11 12  16 17  21 22  26 27  31
   1 0 1 0 1 0 0 1 0 1  R1
```

Description: SwapCTX swaps the current context with the context block at memory location specified in R1. SwapCTX combines the LoadCTX and the StoreCTX instructions.

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.
Special Instructions

5.9.38 SwapLT

Mnemonic: SwapLT

Format:

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>12</td>
<td>16</td>
<td>17</td>
<td>21</td>
<td>22</td>
<td>26</td>
<td>27</td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description: SwapLT is identical to SwapCTX except that it swaps the current context with the context of the last 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. 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.