The Hwacha Vector-Fetch Architecture Manual, Version 3.8.1

Yunsup Lee
Colin Schmidt
Albert Ou
Andrew Waterman
Krste Asanovi

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-262
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-262.html

December 19, 2015

The Hwacha Vector-Fetch Architecture Manual
Version 3.8.1

Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, Krste Asanović
CS Division, EECS Department, University of California, Berkeley
{yunsup|colins|aou|waterman|krste}@eecs.berkeley.edu

December 19, 2015

Contents

1 Introduction

This work-in-progress document outlines the fourth version of the Hwacha vector-fetch architecture. Inspired by traditional vector machines from the 1970s and 1980s such as the Cray-1 [13], and lessons learned from our previous vector-thread architectures such as Scale [9, 7] and Maven [5, 10, 12], the Hwacha architecture is designed to provide high performance at low power/energy for a wide range of applications while being a favorable compiler target. The main feature of the Hwacha architecture is exploiting a high degree of decoupling between vector data access and vector execution. Traditional vector architectures are known to provide some degree of decoupling, however, the Hwacha architecture pushes the limits of decoupling even further by exposing a *vector-fetch* assembly programming model that hoists out all vector instructions into a separate *vector-fetch block*.

We first introduce the Hwacha vector-fetch assembly programming model, the abstract low-level software interface that gives programmers or compiler writers an idea of how code executes on the Hwacha machine, and also discuss key architectural features—in particular, how Hwacha contrasts with other data-parallel architectures, including packed SIMD, SIMT, and traditional vector. We then present the Hwacha instruction set architecture (ISA), and discuss some design decisions.

In conjunction with this document, we have published two other documents that describe the microarchitecture and preliminary evaluation results. All documents are versioned 3.8.1 as they describe a snapshot of the current work that is in progress towards version 4.

2 Data-Parallel Assembly Programming Models

The Hwacha assembly programming model is best explained by contrast with other, popular data-parallel assembly programming models. As a running example, we use a conditionalized SAXPY kernel, CSAXPY. Figure 1 shows CSAXPY expressed in C as both a vectorizable loop and as a SPMD kernel. CSAXPY takes as input an array of conditions, a scalar **a**, and vectors **x** and **y**; it computes **y += ax** for the elements for which the condition is true.

```
void csaxpy(size_t n, bool cond[], float a, float x[], float y[])
{
  for (size_t i = 0; i < n; ++i)
    if (cond[i])
      y[i] = a*x[i] + y[i];
}
```

(a) vectorizable loop

```
void csaxpy_spmd(size_t n, bool cond[], float a, float x[], float y[])
{
  if (tid < n)
    if (cond[tid])
      y[tid] = a*x[tid] + y[tid];
}
```

(b) SPMD kernel

Figure 1: Conditional SAXPY kernel written in C. The SPMD kernel launch code for (b) is omitted for brevity.

2.1 Packed SIMD Assembly Programming Model

Figure 2 shows CSAXPY kernel mapped to a hypothetical packed SIMD architecture, similar to Intel's SSE and AVX extensions. This SIMD architecture has 128-bit registers, each partitioned into four 32-bit fields. As with other packed SIMD machines, ours cannot mix scalar and vector operands, so the code begins by filling a SIMD register with copies of **a**. To map a long vector computation to this architecture, the compiler generates a *stripmine loop*, each iteration of which processes one four-element vector. In this example, the stripmine loop consists of a load from the conditions vector, which in turn is used to set a predicate register. The next four instructions, which correspond to the body of the *if*-statement in Figure 1(a), are masked by the predicate register[1]. Finally, the address registers are incremented by the SIMD width, and the stripmine loop is repeated until the computation is finished—almost. Since the loop handles four elements at a time, extra code

[1]We treat packed SIMD architectures generously by assuming the support of full predication. This feature is quite uncommon. Intel's AVX architecture, for example, only supports predication as of 2015, and then only in its Xeon line of server processors.

```
1  csaxpy_simd:
2       slli    a0, a0, 2
3       add     a0, a0, a3
4       vsplat4 vv0, a2
5  stripmine_loop:
6       vlb4    vv1, (a1)
7       vcmpez4 vp0, vv1
8  !vp0 vlw4    vv1, (a3)
9  !vp0 vlw4    vv2, (a4)
10 !vp0 vfma4   vv1, vv0, vv1, vv2
11 !vp0 vsw4    vv1, (a4)
12      addi    a1, a1, 4
13      addi    a3, a3, 16
14      addi    a4, a4, 16
15      bleu    a3, a0, stripmine_loop
16 # handle edge cases
17 # when (n % 4) != 0 ...
18      ret
```

Figure 2: CSAXPY kernel mapped to the packed SIMD assembly programming model. In all pseudo-assembly examples presented in this section, `a0` holds variable `n`, `a1` holds pointer `cond`, `a2` holds scalar `a`, `a3` holds pointer `x`, and `a4` holds pointer `y`.

is needed to handle up to three *fringe* elements. For brevity, we omitted this code; in this case, it suffices to duplicate the loop body, predicating all of the instructions on whether their index is less than **n**.

The most important drawback to packed SIMD architectures lurks in the assembly code: the SIMD width is expressly encoded in the instruction opcodes and memory addressing code. When the architects of such an ISA wish to increase performance by widening the vectors, they must add a new set of instructions to process these vectors. This consumes substantial opcode space: for example, Intel's newest AVX instructions are as long as 11 bytes. Worse, application code cannot automatically leverage the widened vectors. In order to take advantage of them, application code must be recompiled. Conversely, code compiled for wider SIMD registers fails to execute on older machines with narrower ones. As we later show, this complexity is merely an artifact of poor design.

2.2 SIMT Assembly Programming Model

Figure 3 shows the same code mapped to a hypothetical SIMT architecture, akin to an NVIDIA GPU. The SIMT architecture exposes the data-parallel execution resources as multiple threads of execution; each thread executes one element of the vector. One inefficiency of this approach is immediately evident: the first action each thread takes is to determine whether it is within bounds, so that it can conditionally perform no useful work. Another inefficiency results from the duplication

```
1  csaxpy_simt:
2    mv    t0, tid
3    bgeu  t0, a0, skip
4    add   t1, a1, t0
5    lb    t1, (t1)
6    beqz  t1, skip
7    slli  t0, t0, 2
8    add   a3, a3, t0
9    add   a4, a4, t0
10   lw    t1, (a3)
11   lw    t2, (a4)
12   fma   t0, a2, t1, t2
13   sw    t0, (a4)
14 skip:
15   stop
```

Figure 3: CSAXPY kernel mapped to the SIMT assembly programming model.

of scalar computation: despite the unit-stride access pattern, each thread explicitly computes its own addresses. (The SIMD architecture, in contrast, amortized this work over the SIMD width.) Moreover, massive replication of scalar operands reduces the effective utilization of register file resources: each thread has its own copy of the three array base addresses and the scalar **a**. This represents a threefold increase over the fundamental architectural state.

2.3 Traditional Vector Assembly Programming Model

Packed SIMD and SIMT architectures have a disjoint set of drawbacks: the main limitation of the former is the static encoding of the vector length, whereas the primary drawback of the latter is the lack of scalar processing. One can imagine an architecture that has the scalar support of the former and the dynamism of the latter. In fact, it has existed for over 40 years, in the form of the traditional vector machine, embodied by the Cray-1. The key feature of this architecture is the *vector length register* (VLR), which represents the number of vector elements that will be processed by the vector instructions, up to the hardware vector length (HVL). Software manipulates the VLR by requesting a certain application vector length (AVL); the vector unit responds with the smaller of the AVL and the HVL. As with packed SIMD architectures, a stripmine loop iterates until the application vector has been completely processed. But, as Figure 4 shows, the difference lies in the manipulation of the VLR at the head of every loop iteration. The primary benefits of this architecture follow directly from this code generation strategy. Most importantly, the scalar software is completely oblivious to the hardware vector length: the same code executes correctly and with maximal efficiency on machines with any HVL. Second, there is no fringe code: on the final trip through the loop, the VLR is simply set to the length of the fringe.

```
1  csaxpy_tvec:
2  stripmine_loop:
3       vsetvl  t0, a0
4       vlb     vv0, (a1)
5       vcmpez  vp0, vv0
6  !vp0 vlw     vv0, (a3)
7  !vp0 vlw     vv1, (a4)
8  !vp0 vfma    vv0, vv0, a2, vv1
9  !vp0 vsw     vv0, (a4)
10      add     a1, a1, t0
11      slli    t1, t0, 2
12      add     a3, a3, t1
13      add     a4, a4, t1
14      sub     a0, a0, t0
15      bnez    a0, stripmine_loop
16      ret
```

Figure 4: CSAXPY kernel mapped to the traditional vector assembly programming model.

The advantages of traditional vector architectures over the SIMT approach are owed to the coupled scalar control processor. There is only one copy of the array pointers and of the scalar **a**. The address computation instructions execute only once per stripmine loop iteration, rather than once per element, effectively amortizing their cost by a factor of the HVL.

3 Hwacha Vector-Fetch Assembly Programming Model

The Hwacha architecture builds on the traditional vector architecture, with a key difference: the vector operations have been hoisted out of the stripmine loop and placed in their own *vector fetch block*. This allows the scalar control processor to send only a program counter to the vector processor. The control processor then completes the stripmining loop faster and is able to continue doing useful work, while the vector processor is independently executing the vector instructions.

Figure 5 shows the Hwacha user-visible register state. Like the traditional vector machine, Hwacha has vector data registers (`vv`*0–255*) and vector predicate registers (`vp`*0–15*), but it also has two flavors of scalar registers. These are the *shared* registers (`vs`*0–63*, `vs`*0* is hardwired to constant 0), which can be read and written within a vector fetch block, and *address* registers (`va`*0–31*), which are read-only within a vector fetch block. This distinction supports non-speculative access/execute

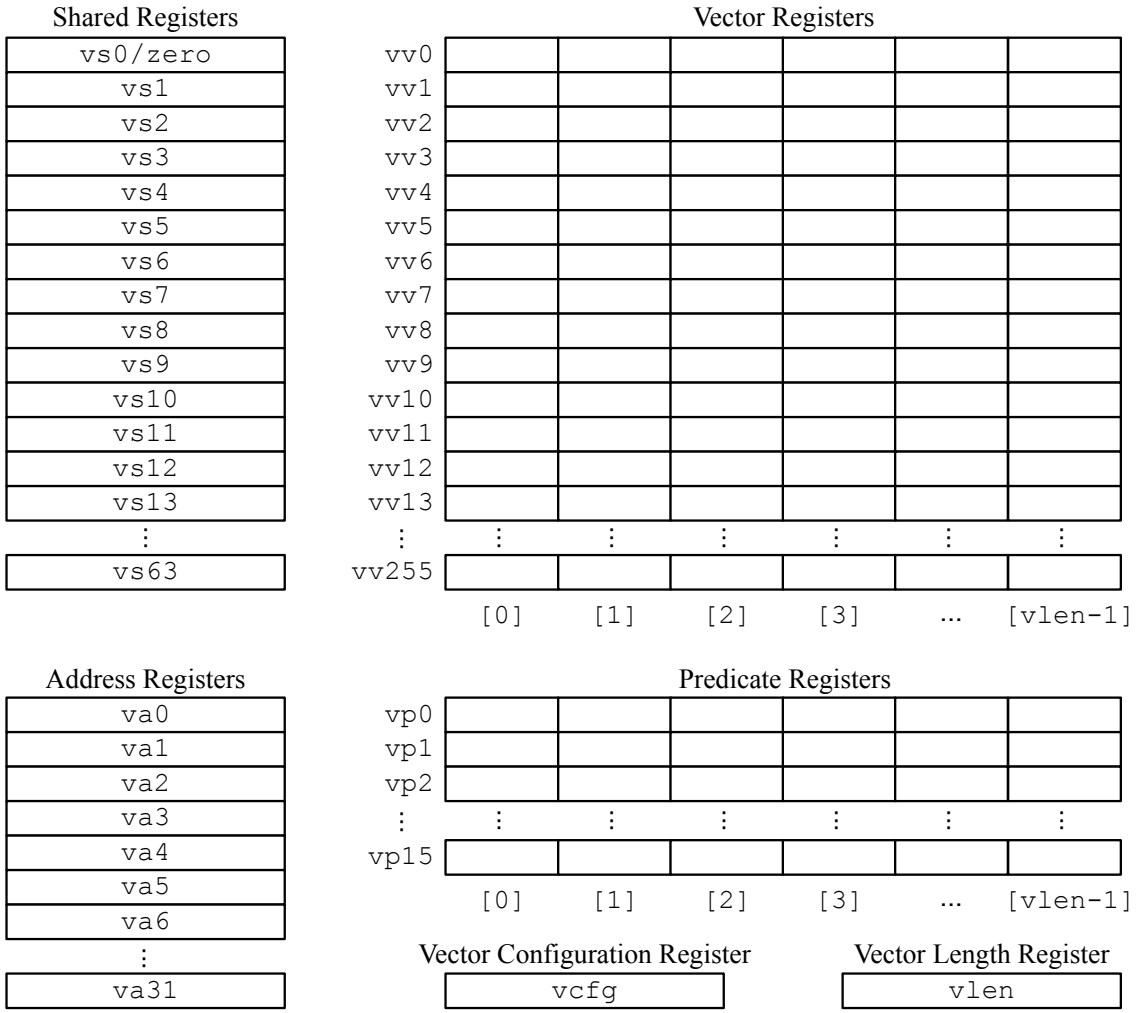


Figure 5: Hwacha user-visible register state.

```
1  csaxpy_control_thread:
2    vsetcfg ...
3    vmcs vs1, a2
4  stripmine_loop:
5    vsetvl  t0, a0
6    vmca    va0, a1
7    vmca    va1, a3
8    vmca    va2, a4
9    vf      csaxpy_worker_thread
10   add     a1, a1, t0
11   slli    t1, t0, 2
12   add     a3, a3, t1
13   add     a4, a4, t1
14   sub     a0, a0, t0
15   bnez    a0, stripmine_loop
16   ret
17
18 csaxpy_worker_thread:
19       vlb     vv0, (va0)
20       vcmpez  vp0, vv0
21 !vp0 vlw      vv0, (va1)
22 !vp0 vlw      vv1, (va2)
23 !vp0 vfma     vv0, vv0, vs1, vv1
24 !vp0 vsw      vv0, (va2)
25       vstop
```

Figure 6: CSAXPY kernel mapped to the Hwacha assembly programming model.

decoupling and is further described in the Hwacha microarchitecture manual.

Vector data and shared registers may hold 8-, 16-, 32-, and 64-bit integer values and half-, single-, and double-precision floating-point values. Vector predicate registers are 1-bit wide, and hold boolean values that mask vector operations. Address registers hold 64-bit pointer values, and serve as the base and stride of unit-strided and strided vector memory instructions.

In addition, the vector configuration register `vcfg`, which keeps the configuration state of the vector unit, and the vector length register `vlen`, which stores the maximum hardware vector length, are also visible to the user. The configuration state is described in Section 5.1.

The maximum hardware vector length is configurable based on how many registers of each type a program uses. Regardless of how many registers are used, a hardware vector length of 8 is guaranteed. The vector length register (`vlen`) can be set to zero, in such case, all vector instructions will not be executed.

Figure 6 shows the CSAXPY code for the Hwacha machine. The structure of the stripmine loop in the control thread (line 1–16) is similar to the traditional vector code, however, all vector

operations in the stripmine loop have been hoisted out into its own worker thread (line 18–25). The control thread first executes the `vsetcfg` instruction (line 2), which adjusts the maximum hardware vector length taking the register usage into account. `vmcs` (line 3) moves the value of a scalar register from the control thread to a `vs` register. The stripmine loop sets the vector length with a `vsetvl` instruction (line 5), moves the array pointers to the vector unit with `vmca` instructions (line 6–8), then executes a vector fetch (`vf`) instruction (line 9) causing the Hwacha unit to execute the vector fetch block. The code in the vector fetch block is equivalent to the vector code in Figure 4, with the addition of a `vstop` instruction, signifying the end of the block.

For the CSAXPY example, factoring the vector code out of the stripmine loop reduces the scalar instruction count by 15%, but as the stripmine loop gets complicated with more vector instructions, the fraction of saved scalar instruction fetches per stripmine loop due to the Hwacha assembly programming model will increase. This lets the control thread run ahead further, enabling a higher degree of access/execute decoupling. Consult the Hwacha microarchitecture manual for more details.

4 Hwacha Instruction Set Architecture

The Hwacha ISA is a load-store architecture; to perform an operation, operands must be read and written to registers. The Hwacha ISA defines instructions for both the control thread and the worker thread. The control thread runs on the control processor and is responsible for configuring the vector accelerator, kicking off work to the vector accelerator, and interacting with the operating system. The worker thread runs on the vector accelerator, which executes vector, vector-scalar, and scalar instructions that carry out the actual computation by operating on registers and moving data between registers and the main memory.

The current practice is to assign a distinct opcode for each supported data type, as is the convention in general-purpose ISAs. For example, there are separate instructions `VFMADD.D`, `VFMADD.S`, and `VFMADD.H` to denote double/single/half-precision fused multiply-adds, respectively.

In future iterations, we may consider adopting a *polymorphic instruction set*. The input/output precisions of an operation would instead be determined by the source and destination register specifiers in conjunction with the register width configuration from `vcfg`. Orthogonality in the instruction set could therefore be achieved without excessive consumption of opcode space or encoding complexity. However, these advantages must be weighed against the vector registers losing the ability to hold values narrower than their configured width, which may constrain register reuse by a compiler.

5 Control Thread Instructions

The control thread instructions are 32 bits in length and are a greenfield extension in the CUSTOM* major opcode space (i.e., overlaid on top of normal RISC-V instructions). There are 3 types of control thread instructions — 1) vector configuration instructions, which configures the vector accelerator, 2) vector fetch instruction, which sends a sequence of work thread instructions to the vector accelerator, 3) vector move instructions, which move scalar values from the control processor to the vector accelerator. These instructions follow the standard RoCC instruction format below.

31 25	24 20	19 15	14	13	12	11 7	6 0
funct7	rs2	rs1	xd	xs1	xs2	rd	opcode
7	5	5	1	1	1	5	7
roccinst[6:0]	src2	src1				dest	*custom-0/1/2/3*

5.1 Vector Configuration Instructions

31 25	24 20	19 15	14	13	12	11 7	6 0
funct7	rs2	rs1	xd	xs1	xs2	rd	opcode
7	5	5	1	1	1	5	7
imm[11:5]	imm[4:0]	src1	0	1	0	VSETCFG	*custom-0*
VSETVL	00000	src1	1	1	0	rd	*custom-0*
VGETCFG	00000	00000	1	0	0	rd	*custom-0*
VGETVL	00000	00000	1	0	0	rd	*custom-0*
VUNCFG	00000	00000	0	0	0	00000	*custom-0*

VSETCFG configures the vector unit with a 64 bit constant built with the top 52 bits of the source register *rs1* combined with the 12 bit immediate value at the lowest 12 bits. The 64 bit configuration register `vcfg` layout follows.

63 32	31 23	22 14	13 9	8 0
0	#v16	#v32	#pred	#v64
32	9	9	5	9

The assembler will take a VSETCFG *#v64, #pred2:0*, or VSETCFG *#v64, #pred, #v32, #v16*, which will also generate sequence of instructions to build the corresponding 64 bit configuration object. *#v64* and *#pred* denotes the number of 64-bit vector and predicate registers used in the program respectively. For further optimization, the number of 32-, 16-bit vector registers may be specified in the *#v32*, and *#v16* fields respectively. These values can range from 0 to 256 for the number of v registers and 0 to 16 for the number of p registers. Once the vector unit is reconfigured, the maximum hardware vector length may be adjusted, and the vector length register is reset to 0.

VSETVL sets the vector length register by taking the minimum of the requested vector length in register *rs1* and the maximum hardware vector length, and writes the set vector length to register *rd*.

VGETCFG writes the content of the vector configuration register (`vcfg`) to register *rd*. Similarly, VGETVL writes the content of the vector length register (`vlen`) to register *rd*.

VUNCFG clears the vector configuration register and sets the vector unit as unused. The vector unit must be configured before any subsequent vector instructions are issued. Before any subsequent vector instructions are issued the vector unit must be configured. The vector unit begins operation in this unconfigured state. If the machine has not been configured since reset or the most recent `vuncfg`, then for any control thread instruction other than `vsetcfg` will trigger an accelerator disabled exception.

5.2 Vector Fetch Instruction

31 25	24 20	19 15	14	13	12	11 7	6 0
funct7	rs2	rs1	xd	xs1	xs2	rd	opcode
7	5	5	1	1	1	5	7
imm[11:5]	10000	src1	0	1	0	imm[4:0]	*custom-1*

The VF instruction executes a block of vector instructions that resides at the target address, which is obtained by adding the 12-bit signed immediate to register *rs1*.

5.3 Vector Move Instructions

31 25	24 20	19 15	14	13	12	11 7	6 0
funct7	rs2	rs1	xd	xs1	xs2	rd	opcode
7	5	5	1	1	1	5	7
VMCS	00srdhi	src1	0	1	0	srdlo	*custom-1*
VMCA	00000	src1	0	1	0	ard	*custom-1*

VMCS (Vector Move Control to Scalar) moves the content of register *rs1* in the control processor to a vector shared (`vs`) register $\{srdhi, srdlo\}$. VMCA (Vector Move Control to Address) moves the content of register *rs1* in the control processor to a vector address (`va`) register *ard*.

6 Worker Thread Instructions

The worker thread instructions are 64 bits in length and respects the variable-length RISC-V encoding. The worker thread instructions are grouped into a separate vector-fetch block. The first instruction in a vector-fetch block is pointed to by a vector-fetch instruction, which lives in the control thread's instruction stream.

In the Hwacha ISA, there are 4 core instruction formats (VJ/VU/VI/VR/VR4), as shown in Figure 7. All are a fixed 64 bits in length and must be aligned on a eight-byte boundary in memory. An instruction address misaligned exception is generated if the `pc` is not eight-byte aligned on an instruction fetch.

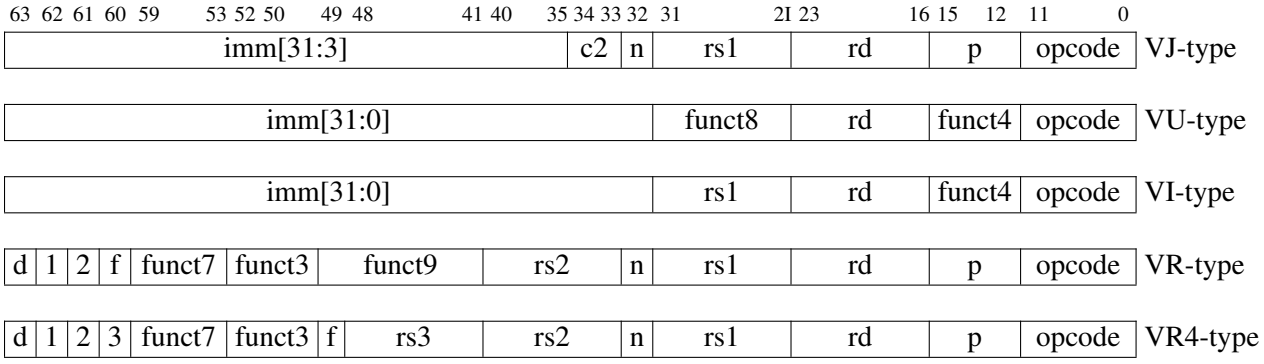


Figure 7: Hwacha instruction formats.

Note, the ISA keeps the source (*rs1* and *rs2*) and destination (*rd*) registers and the predicate (*p*) at the same position in all formats to simplify decoding. Immediates are left aligned.

The *p* field designates a predicate register that masks the instruction. The *n* flag (bit 32) negates the condition of the appointed predicate.

When the *d* flag at bit 63 is set, register *rd* is interpreted as a vector register (`vd`). When it is cleared, register *rd* is interpreted as a shared register (`vs`). Similarly, the *1* flag (bit 62), the *2* flag (bit 61), and the *3* flag (bit 60) indicates whether *rs1*, *rs2*, and *rs3* refers to a vector register or a shared register respectively. An instruction with all source and destination operands marked as shared registers are considered a scalar instruction (`@s` in the assembly), and therefore does not need to execute in vector fashion. Instructions that do not use predicates are given the `@all` prefix in assembly.

6.1 Vector Strided, Strided-Segment Memory Instructions

Unit-stride, constant-stride, unit-stride-segmented, constant-stride-segmented vector load and store instructions transfer values between vector registers and memory. Unit-stride vector memory instructions transfer vectors whose elements are held in contiguous locations in memory. Constant-stride vector memory instructions transfer vectors whose elements are held in memory addresses

that form an arithmetic progression. Segmented vector memory instructions transfer values between consecutive vector registers and memory. These instructions use the VR-type format.

63	62	61	60 48	47 45	44 42	41	40 33	32	31 24	23 16	15 12	11 0
s/v	*s/v*	*s/v*	—	seglen	w	st?	as2	n	as1	vd	p	opcode
1	1	1	13	3	3	1	8	1	8	8	4	12
1	0	0	0	seglen	w	st?	stride	n	base	dest	pred	VLD
1	0	0	0	seglen	w	st?	stride	n	base	dest	pred	VST

The vector predicate register *p* in conjunction with the *n* flag masks the vector memory instruction. Vector load instructions copy a vector of values from memory to vector register *rd*. Vector store instructions copy a vector values in vector register *rd* to memory. The base address for the memory transfers are taken from the address register *as1*. The non-segmented memory operations are a degenerate case where *seglen* is set to zero.

The *seglen* field indicates the number of segments (i.e., consecutive vector registers performing the vector memory operation). The *w* field indicates width of the transfer (byte, halfword, word, doubleword) and whether to sign-extend or zero-extend load results. The *st?* field indicates whether addresses are unit-strided or constant-strided. Address register *as2* holds the stride for the constant-stride vector memory operations.

6.1.1 Format

```
@[!]p vl{b,h,w,d,bu,hu,wu} vd, as1
@[!]p vlst{b,h,w,d,bu,hu,wu} vd, as1, as2
@[!]p vlseg{b,h,w,d,bu,hu,wu} vd, as1, seglen
@[!]p vlsegst{b,h,w,d,bu,hu,wu} vd, as1, as2, seglen
@[!]p vs{b,h,w,d} vd, as1
@[!]p vsst{b,h,w,d} vd, as1, as2
@[!]p vsseg{b,h,w,d} vd, as1, seglen
@[!]p vssegst{b,h,w,d} vd, as1, as2, seglen
```

6.1.2 Vector Load Operation

```
if (unit-strided) stride = elsize;
else stride = areg[as2]; // constant-strided

for (int i=0; i<vl; ++i)
  if ([!]preg[p][i])
    for (int j=0; j<seglen+1; j++)
      vreg[vd+j][i] = mem[areg[as1] + (i*(seglen+1)+j)*stride];
```

6.1.3 Vector Store Operation

```
if (unit-strided) stride = elsize;
else stride = areg[as2]; // constant-strided

for (int i=0; i<vl; ++i)
  if ([!]preg[p][i])
    for (int j=0; j<seglen+1; j++)
      mem[areg[base] + (i*(seglen+1)+j)*stride] = vreg[vd+j][i];
```

6.2 Vector Indexed Memory Instructions

Vector indexed load and store instructions transfer vectors whose elements are located at offsets from a base address, with the offsets specified by the values of an index vector. The effective address of each element is the base address plus the sign-extended offset register. These instructions use the VR-type format.

63	62	61	60 48	47 45	44 42	41	40 33	32	31 24	23 16	15 12	11 0
s/v	s/v	s/v	—	seglen	w	—	vs2	n	ss1	vd	p	opcode
1	1	1	13	3	3	1	8	1	8	8	4	12
1	0	1	0	seglen	w	0	index	n	base	dest	pred	VLD
1	0	1	0	seglen	w	0	index	n	base	dest	pred	VST

All the fields except the index field have the same meaning to the fields of vector memory instructions in Section 6.1. The offset from the base address is taken from vector register *vs2*.

6.2.1 Format

```
@[!]p vlx{b,h,w,d,bu,hu,wu} vd, ss1, vs2
@[!]p vlsegx{b,h,w,d,bu,hu,wu} vd, ss1, vs2, seglen
@[!]p vsx{b,h,w,d} vd, ss1, vs2
@[!]p vssegx{b,h,w,d} vd, ss1, vs2, seglen
```

6.2.2 Vector Indexed Load Operation

```
for (int i=0; i<vl; ++i)
  if ([!]preg[p][i])
    for (int j=0; j<seglen+1; j++)
      vreg[vd+j][i] = mem[sreg[base] + vreg[vs2][i] + j*elsize];
```

6.2.3 Vector Indexed Store Operation

```
for (int i=0; i<vl; ++i)
```

```
if ([!]preg[p][i])
  for (int j=0; j<seglen+1; j++)
    mem[sreg[base] + vreg[vs2][i] + j*elsize] = vreg[vd+j][i];
```

6.3 Vector Atomic Memory Instructions

The vector atomic memory operation (AMO) instructions perform a vector of read-modify-write operations. These instructions use the VR-type format.

63	62	61	60 41	40 33	32	31 24	23 16	15 12	11 0
s/v	*s/v*	*s/v*	funct20	rs2	n	rs1	vd	p	opcode
1	1	1	13	8	1	8	8	4	12
1	s1	s2	operation,ordering	src	n	addr	dest	pred	VAMO

Similar to AMO operations defined by the RISC-V ISA, these instructions atomically load data value(s) from address(es) in register *rs1* (either the shared register *ss1* or vector register *vs1* indicated by the *s1* flag), place the original value into vector register *vd*, apply a binary operator to the loaded value(s) and the value(s) in register *rs2* (either from the shared register *ss2* or vector register *vs2* indicated by the *s2* flag), then store the result back to the address(es) in register *rs1*. Both register specifies *rs1* and *rs2* refer to vector registers *vs1* and *vs2* when the *s1* flag and the *s2* flag are set respectively. If *s1* and *s2* are both clear the AMO operation will be preformed vector length times on the same address. If only one of *s1* and *s2* are scalars then either the same address or the same data will be used, respectively, for all operations.

These vector AMO instructions optionally provide release consistency semantics with the *aq* bit and *rl* bit. For more detail, please consult the "A" standard extension chapter of the RISC-V user-level ISA specification.

6.3.1 Format

```
@[!]p vamo{swap,add,xor,and,or,min,max,minu,maxu}.{w,d}.{vv,vs,sv,ss} vd, rs1, rs2
```

6.3.2 Operation

```
for (int i=0; i<vl; ++i)
  if ([!]preg[p][i])
    atomic {
      temp = mem[s1 ? vreg[rs1][i] : sreg[rs1]];
      mem[s1 ? vreg[rs1][i] : sreg[rs1]] = amoop(temp, s2 ? vreg[rs2][i] : sreg[rs2]);
      vreg[vd][i] = temp;
    }
```

6.4 Vector Integer Arithmetic Instructions

Vector integer arithmetic instructions typically take two input operands and produce one output operand. The only exceptions are the vector element index instruction (`veidx`), which takes one input operand and returns the element index left shifted by the value from the input operand. These instructions are encoded in the VR-type format.

63	62	61	60 … 41	40 … 33	32	31 … 24	23 … 16	15 … 12	11 … 0
s/v	s/v	s/v	funct20	rs2	n	rs1	rd	p	opcode
1	1	1	13	8	1	8	8	4	12
d	s1	s2	operation	src2	n	src1	dest	pred	VOP

The vector predicate register *p* in conjunction with the *n* flag masks the vector instruction. Flags *d*, *s1*, and *s2* indicate whether registers *rd*, *rs1*, and *rs2* should be interpreted as vector registers (if set) or shared registers (if cleared). When all source and destination operands are marked as shared registers, the instruction is decoded as a scalar instruction. For this case, the *p* register should be set to zero and the *n* field should be cleared, otherwise, an illegal instruction exception will be raised.

6.4.1 Format

```
@[!]p viop.{vv,vs,sv} rd, rs1, rs2
@s viop.ss sd, ss1, ss2
```

6.4.2 Operation

```
for (int i=0; i<vl; ++i)
  if ([!]preg[p][i])
    (d ? vreg[rd][i] : sreg[rd]) =
      iop(s1 ? vreg[rs1][i] : sreg[rs1],
          s2 ? vreg[rs2][i] : sreg[rs2]); // for insts with 2 inputs
```

6.5 Vector Reduction Instructions

The vector first instruction (`vfirst`) returns the first value of a vector register that is not masked off under the predicate, in a shared register. If the predicate register is entirely clear the result is zero. This can be used, in conjunction with a vector compare instruction, to iterate through the distinct values in a vector register.

63	62	61	60 … 41	40 … 33	32	31 … 24	23 … 16	15 … 12	11 … 0
s/v	s/v	s/v	funct20	—	n	vs1	sd	p	opcode
1	1	1	13	8	1	8	8	4	12
0	1	0	VFIRST	0	n	vsrc	sdest	pred	VOP

6.5.1 Format

```
@[!]p vfirst sd, vs1
```

6.5.2 Operation

```
sred[sd] = 0;
for (int i=0; i<vl; ++i)
  if ([!]preg[p][i]) {
    sreg[sd] = vreg[vs1][i]
    break
  }
```

6.6 Vector Floating-Point Arithmetic Instructions

Vector floating-point arithmetic instructions take one, two, or three input operands and produce one output operand. These instructions are encoded in either the VR-type format or the VR4-type format depending on the number of input operands.

63	62	61	60 57	56 53	52 50	49 41	40 33	32	31 24	23 16	15 12	11 0
s/v	s/v	s/v	—	fmt	rm	funct	rs2	n	rs1	rd	p	opcode
1	1	1	4	4	3	9	8	1	8	8	4	12
d	s1	s2	0	fmt	rm	FADD/FSUB	src2	n	src1	dest	pred	VOP-FP
d	s1	s2	0	fmt	rm	FMUL/FDIV	src2	n	src1	dest	pred	VOP-FP
d	s1	s2	0	fmt	rm	FMIN/FMAX	src2	n	src1	dest	pred	VOP-FP
d	s1	s2	0	fmt	rm	FSQRT	0	n	src1	dest	pred	VOP-FP

63	62	61	60	59 57	56 53	52 50	49	48 41	40 33	32	31 24	23 16	15 12	11 0
s/v	s/v	s/v	s/v	—	fmt	rm	—	rs3	rs2	n	rs1	rd	p	opcode
1	1	1	1	3	4	3	1	8	8	1	8	8	4	12
d	s1	s2	s3	0	fmt	rm	0	src3	src2	n	src1	dest	pred	VF[N]MADD
d	s1	s2	s3	0	fmt	rm	0	src3	src2	n	src1	dest	pred	VF[N]MSUB

The instruction fields are similar to those of integer arithmetic instructions shown in Section 6.4. The precision of the input operand(s) and the output operand, and the rounding mode of the operation are all specified in the *fmt* and *rm* fields respectively. If an operation has a single *fmt* specifier rather than two, this specifier will be treated as both the input and output *fmt*. For the one-input floating-point operation, and two-input floating-point operations, the operation type is encoded in the *funct* field under the same *opcode* space VOP-FP. For three-input floating-point operations, the operation type is encoded in *opcode* space VFMADD, VFNMADD, VFMSUB, VFNMSUB.

Widening-precision floating-point add, subtract, multiply and fused-multiply-add operations that produce higher-precision results are simply denoted with different input precision (*fmti*) and output precision (*fmto*) marked in the *fmt* field.

Floating point exceptions generated by the worker thread are accrued in the control thread's `fflags` register. Once the control thread reads this register it sees the accrued exceptions in program order. The worker thread is unable to read the exceptions.

Only the control thread is able to set the dynamic rounding mode. The value of the dynamic rounding mode is inherited by the worker thread as the vector fetch instruction is issued. The worker thread is able to use static and dynamic rounding mode instructions, but is unable to change the dynamic rounding mode.

6.6.1 Format

```
@[!]p vfop.fmto.fmti.v rd, rs1, {rm}
@[!]p vfop.fmto.fmti.{vv,vs,sv} rd, rs1, rs2, {rm}
@[!]p vfop.fmto.fmti.{vvv,vvs,vsv,svv,vss,svs,ssv} rd, rs1, rs2, rs3, {rm}
@s vfop.fmto.fmti.s sd, ss1, {rm}
@s vfop.fmto.fmti.ss sd, ss1, ss2, {rm}
@s vfop.fmto.fmti.sss sd, ss1, ss2, ss3, {rm}
```

6.6.2 Operation

```
for (int i=0; i<vl; ++i)
  if ([!]preg[p][i])
    (d ? vreg[rd][i] : sreg[rd]) =
      fop(fmto, fmti, // output and input precisions
          s1 ? vreg[rs1][i] : sreg[rs1],
          s2 ? vreg[rs2][i] : sreg[rs2],  // for insts with 2 inputs
          s3 ? vreg[rs3][i] : sreg[rs3]); // for insts with 3 inputs
```

6.7 Vector Compare Instructions

Vector predicate compare instructions compares two registers and stores the resulting flag into a predicate register.

63	62	61	60 53	52 41	40 33	32	31 24	23 20	19 16	15 12	11 0
s/v	s/v	s/v	—	funct	rs2	n	rs1	—	pd	p	opcode
1	1	1	8	12	8	1	8	4	4	4	12
1	s1	s2	0	EQ/LT[U]	src2	n	src1	0	pdest	pred	VOP

The vector predicate register *p* in conjunction with the *n* flag masks the vector compare instruction. The compare function is chosen in the *funct20* field. Instructions that produce opposite results for equal, less than, less than unsigned comparisons are omitted, as all instructions can take the negated predicate condition as an input.

In addition to integer comparisons, there is also a set of floating point comparisons, which operate similarly.

63	62	61	60 57	56 53	52 41	40 33	32	31 24	23 20	19 16	15 12	11 0
s/v	s/v	s/v	—	fmt	funct	rs2	n	rs1	—	pd	p	opcode
1	1	1	4	4	12	8	1	8	4	4	4	12
1	s1	s2	0	fmt	FEQ/LT/LE	src2	n	src1	0	pdest	pred	VOP

6.7.1 Format

```
@[!]p vcmp{eq,lt,ltu}.{vv,vs,sv} pd, rs1, rs2
@[!]p vcmpf{eq,lt,le}.{d,s,h}.{vv,vs,sv} pd, rs1, rs2
```

6.7.2 Operation

```
for (int i=0; i<vl; ++i)
  if ([!]preg[p][i])
    preg[pd][i] = cmp(s1 ? vreg[rs1][i] : sreg[rs1],
                      s2 ? vreg[rs2][i] : sreg[rs2]);
```

6.8 Vector Predicate Memory Instructions

Vector predicate memory instructions spill and refill contents of the predicate register to and from memory. These instructions use the VR-type format.

63	62	61	60 50	49	48 41	40 33	32	31 24	23 20	19 16	15 12	11 0
s/v	s/v	s/v	—	p	—	—	—	as1	—	pd	—	opcode
1	1	1	11	1	8	8	1	8	4	4	4	12
1	0	0	0	1	0	0	0	base	0	pdest	0	VLD
1	0	0	0	1	0	0	0	base	0	psrc	0	VST

These instructions are not predicated, meaning that they always execute fully. The vector predicate load instruction restores the predicate mask from memory, while the store instruction spills the content of the predicate mask to memory. In both cases, the base address for the memory transfers are taken from the address register *as1*.

6.8.1 Format

```
@all vpl pd, as1
@all vps pd, as1
```

6.8.2 Vector Predicate Load Operation

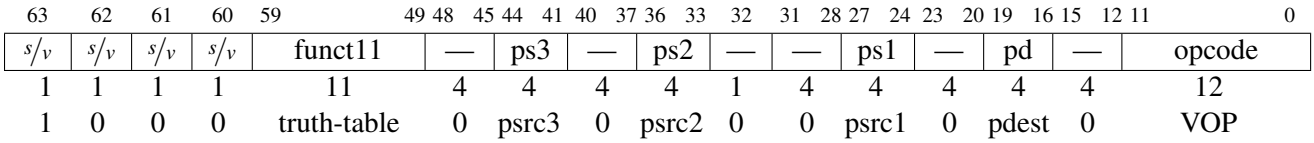
```
for (int i=0; i<vl; ++i)
  preg[pd][i] = mem[areg[as1]+i] & 0x1;
```

6.8.3 Vector Predicate Store Operation

```
for (int i=0; i<vl; ++i)
  mem[areg[as1]+i] = (sign-extend-to-byte)preg[pd][i];
```

6.9 Vector Predicate Arithmetic Instructions

Vector predicate arithmetic instructions perform a logic operation on three input predicates and writes the result to a predicate register.



These instructions are not predicated, meaning that they always execute fully. To be generic, the logic operation performed on the input predicates is expressed as an 8 entry truth table in the lower 8 bits of the *funct11* field. A couple of well-known logic operations are defined as an assembler pseudo-instructions.

6.9.1 Format

```
@all vpop pd, ps1, ps2, ps3, truth-table
@all vp{clear,set} pd
@all vp{xorxor,xoror,xorand} pd, ps1, ps2, ps3
@all vp{orxor,oror,orand} pd, ps1, ps2, ps3
@all vp{andxor,andor,andand} pd, ps1, ps2, ps3
```

6.9.2 Operation

```
for (int i=0; i<vl; ++i)
  preg[pd][i] = op(preg[ps1][i], preg[ps2][i], preg[ps3][i]);
```

6.10 Scalar Memory Instructions

Scalar memory instructions transfer values between shared registers and memory.

63	62	61	60 49	48	47 45	44 41	40 33	32	31 24	23 16	15 12	11 0
s/v	*s/v*	*s/v*	—	a/s?	—	w	ss2	n	rs1	sd	p	opcode
1	1	1	12	1	3	4	8	1	8	8	4	12
0	0	0	0	a/s?	0	w	0	n	base	dest	pred	VLD
0	0	0	0	a/s?	0	w	src	n	base	0	pred	VST

Scalar loads and stores and encoded the same as vector loads and stores just with all scalar/vector flags cleared. Scalar load instructions copy the value from memory to the shared register *sd*. Scalar store instructions copy values in shared register *ss2* to memory. The effective byte address is obtained from either address register *as1* or shared register *ss1*. The *a/s?* field picks which register to use to calculate the effective address, and the *w* field describes the width of the operation.

6.10.1 Format

```
@s vla{b,h,w,d,bu,hu,wu} sd, as1
@s vls{b,h,w,d,bu,hu,wu} sd, ss1
@s vsa{b,h,w,d} as1, ss2
@s vss{b,h,w,d} ss1, ss2
```

6.10.2 Scalar Load Operation

```
if (use aregs for address) reg = areg;
else reg = sreg; // use sregs for address

sreg[sd] = mem[reg[rs1]];
```

6.10.3 Scalar Store Operation

```
if (use aregs for address) reg = areg;
else reg = sreg; // use sregs for address

mem[reg[rs1]] = sreg[ss2];
```

6.11 Scalar Instructions

In addition to the scalar instructions as a result of marking all input and output operands as shared registers, some scalar instructions are encoded as register-immediate operations using the VU-type and the VI-type format.

63 32	31 24	23 16	15 12	11 0
imm	ss1	sd	funct4	opcode
32	8	8	4	12
imm[31:0]	src1	dest	ADDI/SLTI[U]	VOP-IMM
imm[31:0]	src1	dest	ANDI/ORI/XORI	VOP-IMM

Similar to ADDI, SLTI, ANDI, ORI, and XORI on RISC-V, the operation is done between the sign-extended 32-bit immediate and the value of shared register *ss1*, and the result is placed in shared register *sd*. Arithmetic overflow is ignored and the result is simply the low 64-bits of the result.

63 38	37 32	31 24	23 16	15 12	11 0
imm	imm	ss1	sd	funct	opcode
26	6	8	8	4	12
26{0}	shamt	src1	dest	SLLI	VOP-IMM
26{0}	shamt	src1	dest	SRLI	VOP-IMM
1,25{0}	shamt	src1	dest	SRAI	VOP-IMM

The same holds for SLLI, SLRI, and SRAI operations. The 6-bit *shamt* value is held in the lower bits of the 32-bit immediate.

63 32	31 24	23 16	15 12	11 0
imm	funct	sd	funct	opcode
32	8	8	4	12
imm[31:0]	operation	dest	funct	VLUI
imm[31:0]	operation	dest	funct	VAUIPC

The VLUI instruction loads the 32-bit immediate into the upper 32 bits of the shared register *sd* and fills the lower 32 bits with all zeros. A sequence of VLUI and VADDI instructions can be used to generate 64 bit constants, for integer and floating point values.

The VAUIPC instruction forms a 64-bit constant by shifting the 32-bit immediate to the upper 32 bits and filling in the lowest 32 bits with zeros, then adding the current `pc` to this value and stores the result into the destination register *sd*.

6.11.1 Format

```
@s sop sd, imm
@s sop sd, ss1, imm
```

6.11.2 Operation

```
sreg[sd] = op(sign-extend(imm),
              sreg[ss1]) // for insts with a register input
```

6.12 Control Flow Instructions

There are three types of control flow instructions: stop, fence, and consensual jumps. The immediate for control flow instructions are reduced to 29 bits to allow for any 8-byte aligned address within 32 bits to be generated as the displacement.

63 35	34 33	32	31 24	23 16	15 12	11 0
—	—	—	—	—	—	opcode
29	2	1	8	8	4	12
STOP/FENCE	0	0	0	0	0000	VCTRL

The VSTOP instruction marks the end of the current vector-fetch block instruction, and the VFENCE instruction orders all memory accesses before and after the instruction.

63 35	34 33	32	31 24	23 16	15 12	11 0
imm	c2	n	ss1	sd	p	opcode
29	2	1	8	8	4	12
imm[31:3]	ALL	neg?	0	dest	pred	VCJAL
imm[31:3]	ANY	neg?	0	dest	pred	VCJAL
imm[31:3]	ALL	neg?	src	dest	pred	VCJALR
imm[31:3]	ANY	neg?	src	dest	pred	VCJALR

The vector consensual jump and link (VCJAL) instruction and the vector consensual jump and link register (VCJALR) instruction both use the VJ-type format where the 29-bit immediate encodes a signed offset in multiple of 8 bytes. VCJAL stores the address of the instruction following the jump (`vpc+8`) into shared register *sd*, and jumps to the target address (computed by sign-extending the immediate and adding it to the `vpc`) when the consensual condition is met (ALL bits in the predicate register *p* are set, or ANY bit in the predicate register *p* is set).

The indirect consensual jump instruction VCJALR calculates the target address by adding the sign-extended immediate to the shared register *ss1*, then setting the lowest 3 bit of the result to zero. Similar to VCJAL, when the consensual condition is met, the address of the instruction following the jump is stored into shared register *sd*, and the jump is taken.

6.12.1 Format

```
@all vstop
@all vfence
@[!]p vcjal.{all,any} sd, imm
@[!]p vcjalr.{all,any} sd, ss1, imm
```

6.12.2 Stop Operation

Halt execution on the current vector-fetch block, and move on to the next vector-fetch block.

6.12.3 Fence Operation

Order all memory accesses before and after the fence instruction.

6.12.4 Consensual Jump Operation

```
if (vcjal) base = vpc;
else base = sreg[ss1]; // vcjalr

if (met_consensual_cond([!]preg[p])) {
  sreg[sd] = vpc + 8;
  vpc = base + imm;
}
```

6.13 Exceptions

The worker thread only generates two types of exceptions, misaligned exceptions, and illegal instruction exceptions. If the address of a load or store is not aligned with its data width then a misaligned load or store exception is generated. Additionally vector worker thread instructions are aligned to 8 bytes and will generate an instruction misaligned exception otherwise.

Illegal instruction exceptions are generated for several reasons. For arithmetic instructions setting the destination to a shared register with any of the source operands as a vector register generates and illegal instruction exception. In addition, for floating point instructions if the register is configured to a smaller precision than the instructions opcode dictates an illegal instruction exception. If an instruction attempts to access a vector register or predicate register with a number larger than the currently configured number of vector or predicate registers, then an illegal instruction exception is generated. This includes the case where a segmented load or store would generate a reference to a vector register larger than the current configuration.

7 Listing of Hwacha Worker Thread Instructions

This section presents opcode maps and worker thread instruction listings for Hwacha. Table 1 shows a map of the major opcodes for Hwacha. We try to mimic RISC-V's major opcode mapping as much as possible. Note that the lowest 7 bits of a Hwacha worker thread instruction (`inst[6:0]`) are tied to `0_111_111` in order to follow the 64-bit RISC-V instruction encoding.

inst[9:7] / inst[11:10]	000	001	010	011	100	101	110	111
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	*reserved*
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	*reserved*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*custom-5*	*custom-2/rv128*	*reserved*
11	CTRL	JALR	*custom-4*	JAL	SYSTEM	*custom-6*	*custom-3/rv128*	*reserved*

Table 1: RISC-V base opcode map, inst[1:0]=11

The actual bit mapping for all Hwacha worker thread instruction follows.

63	62	61	60	59 53	52 50	49 48 45 44 41 40	38 37 36 35 34 33	32	31 29	28 27 24 23	20 19 16 15	12 11 0	
d	1	2	f	funct7	funct3	funct9	vs2	n	vs1	vd	p	opcode	VR-type

Hwacha Vector Load/Store Instructions (Unit-Stride, Unit-Stride-Segmented)

1	0	0	0	0000000	000	000000000	00000000	n	000	as1	vd	p	101100111111	@[!]p VLB vd,as1
1	0	0	0	0000000	000	000000010	00000000	n	000	as1	vd	p	101100111111	@[!]p VLH vd,as1
1	0	0	0	0000000	000	000000100	00000000	n	000	as1	vd	p	101100111111	@[!]p VLW vd,as1
1	0	0	0	0000000	000	000000110	00000000	n	000	as1	vd	p	101100111111	@[!]p VLD vd,as1
1	0	0	0	0000000	000	000001000	00000000	n	000	as1	vd	p	101100111111	@[!]p VLBU vd,as1
1	0	0	0	0000000	000	000001010	00000000	n	000	as1	vd	p	101100111111	@[!]p VLHU vd,as1
1	0	0	0	0000000	000	000001100	00000000	n	000	as1	vd	p	101100111111	@[!]p VLWU vd,as1
1	0	0	0	0000000	000	000000000	00000000	n	000	as1	vd	p	111100111111	@[!]p VSB vd,as1
1	0	0	0	0000000	000	000000010	00000000	n	000	as1	vd	p	111100111111	@[!]p VSH vd,as1
1	0	0	0	0000000	000	000000100	00000000	n	000	as1	vd	p	111100111111	@[!]p VSW vd,as1
1	0	0	0	0000000	000	000000110	00000000	n	000	as1	vd	p	111100111111	@[!]p VSD vd,as1
1	0	0	0	0000000	seglen	000000000	00000000	n	000	as1	vd	p	101100111111	@[!]p VLSEGB vd,as1,seglen
1	0	0	0	0000000	seglen	000000010	00000000	n	000	as1	vd	p	101100111111	@[!]p VLSEGH vd,as1,seglen
1	0	0	0	0000000	seglen	000000100	00000000	n	000	as1	vd	p	101100111111	@[!]p VLSEGW vd,as1,seglen
1	0	0	0	0000000	seglen	000000110	00000000	n	000	as1	vd	p	101100111111	@[!]p VLSEGD vd,as1,seglen
1	0	0	0	0000000	seglen	000001000	00000000	n	000	as1	vd	p	101100111111	@[!]p VLSEGBU vd,as1,seglen
1	0	0	0	0000000	seglen	000001010	00000000	n	000	as1	vd	p	101100111111	@[!]p VLSEGHU vd,as1,seglen
1	0	0	0	0000000	seglen	000001100	00000000	n	000	as1	vd	p	101100111111	@[!]p VLSEGWU vd,as1,seglen
1	0	0	0	0000000	seglen	000000000	00000000	n	000	as1	vd	p	111100111111	@[!]p VSSEGB vd,as1,seglen
1	0	0	0	0000000	seglen	000000010	00000000	n	000	as1	vd	p	111100111111	@[!]p VSSEGH vd,as1,seglen
1	0	0	0	0000000	seglen	000000100	00000000	n	000	as1	vd	p	111100111111	@[!]p VSSEGW vd,as1,seglen
1	0	0	0	0000000	seglen	000000110	00000000	n	000	as1	vd	p	111100111111	@[!]p VSSEGD vd,as1,seglen

63	62	61	60	59 53	52 50	49 48 45 44 41 40 38	37 36 35 34 33	32	31 29	28 27 24 23	20 19 16 15	12 11 0	
d	1	2	f	funct7	funct3	funct9	vs2	n	vs1	vd	p	opcode	VR-type

Hwacha Vector Load/Store Instructions (Constant-Stride, Constant-Stride-Segmented)

1	0	0	0	0000000	000	000000001	000	as2	n	000	as1	vd	p	101100111111	@[!]p VLSTB vd,as1,as2
1	0	0	0	0000000	000	000000011	000	as2	n	000	as1	vd	p	101100111111	@[!]p VLSTH vd,as1,as2
1	0	0	0	0000000	000	000000101	000	as2	n	000	as1	vd	p	101100111111	@[!]p VLSTW vd,as1,as2
1	0	0	0	0000000	000	000000111	000	as2	n	000	as1	vd	p	101100111111	@[!]p VLSTD vd,as1,as2
1	0	0	0	0000000	000	000001001	000	as2	n	000	as1	vd	p	101100111111	@[!]p VLSTBU vd,as1,as2
1	0	0	0	0000000	000	000001011	000	as2	n	000	as1	vd	p	101100111111	@[!]p VLSTHU vd,as1,as2
1	0	0	0	0000000	000	000001101	000	as2	n	000	as1	vd	p	101100111111	@[!]p VLSTWU vd,as1,as2
1	0	0	0	0000000	000	000000001	000	as2	n	000	as1	vd	p	111100111111	@[!]p VSSTB vd,as1,as2
1	0	0	0	0000000	000	000000011	000	as2	n	000	as1	vd	p	111100111111	@[!]p VSSTH vd,as1,as2
1	0	0	0	0000000	000	000000101	000	as2	n	000	as1	vd	p	111100111111	@[!]p VSSTW vd,as1,as2
1	0	0	0	0000000	000	000000111	000	as2	n	000	as1	vd	p	111100111111	@[!]p VSSTD vd,as1,as2
1	0	0	0	0000000	seglen	000000001	000	as2	n	000	as1	vd	p	101100111111	@[!]p VLSEGSTB vd,as1,as2,seglen
1	0	0	0	0000000	seglen	000000011	000	as2	n	000	as1	vd	p	101100111111	@[!]p VLSEGSTH vd,as1,as2,seglen
1	0	0	0	0000000	seglen	000000101	000	as2	n	000	as1	vd	p	101100111111	@[!]p VLSEGSTW vd,as1,as2,seglen
1	0	0	0	0000000	seglen	000000111	000	as2	n	000	as1	vd	p	101100111111	@[!]p VLSEGSTD vd,as1,as2,seglen
1	0	0	0	0000000	seglen	000001001	000	as2	n	000	as1	vd	p	101100111111	@[!]p VLSEGSTBU vd,as1,as2,seglen
1	0	0	0	0000000	seglen	000001011	000	as2	n	000	as1	vd	p	101100111111	@[!]p VLSEGSTHU vd,as1,as2,seglen
1	0	0	0	0000000	seglen	000001101	000	as2	n	000	as1	vd	p	101100111111	@[!]p VLSEGSTWU vd,as1,as2,seglen
1	0	0	0	0000000	seglen	000000001	000	as2	n	000	as1	vd	p	111100111111	@[!]p VSSEGSTB vd,as1,as2,seglen
1	0	0	0	0000000	seglen	000000011	000	as2	n	000	as1	vd	p	111100111111	@[!]p VSSEGSTH vd,as1,as2,seglen
1	0	0	0	0000000	seglen	000000101	000	as2	n	000	as1	vd	p	111100111111	@[!]p VSSEGSTW vd,as1,as2,seglen
1	0	0	0	0000000	seglen	000000111	000	as2	n	000	as1	vd	p	111100111111	@[!]p VSSEGSTD vd,as1,as2,seglen

63	62	61	60	59 53	52 50	49 48 45 44 41 40	3837 36 35 34 33	32	31 2928 27 24 23	20 19 16 15	12 11 0	
d	1	2	f	funct7	funct3	funct9	vs2	n	vs1	vd	p	opcode	VR-type

Hwacha Vector Indexed Load/Store Instructions

1	0	1	0	0000000	000	000000000	vs2	n	ss1	vd	p	101100111111	@[!]p VLXB vd,ss1,vs2
1	0	1	0	0000000	000	000000010	vs2	n	ss1	vd	p	101100111111	@[!]p VLXH vd,ss1,vs2
1	0	1	0	0000000	000	000000100	vs2	n	ss1	vd	p	101100111111	@[!]p VLXW vd,ss1,vs2
1	0	1	0	0000000	000	000000110	vs2	n	ss1	vd	p	101100111111	@[!]p VLXD vd,ss1,vs2
1	0	1	0	0000000	000	000001000	vs2	n	ss1	vd	p	101100111111	@[!]p VLXBU vd,ss1,vs2
1	0	1	0	0000000	000	000001010	vs2	n	ss1	vd	p	101100111111	@[!]p VLXHU vd,ss1,vs2
1	0	1	0	0000000	000	000001100	vs2	n	ss1	vd	p	101100111111	@[!]p VLXWU vd,ss1,vs2
1	0	1	0	0000000	000	000000000	vs2	n	ss1	vd	p	111100111111	@[!]p VSXB vd,ss1,vs2
1	0	1	0	0000000	000	000000010	vs2	n	ss1	vd	p	111100111111	@[!]p VSXH vd,ss1,vs2
1	0	1	0	0000000	000	000000100	vs2	n	ss1	vd	p	111100111111	@[!]p VSXW vd,ss1,vs2
1	0	1	0	0000000	000	000000110	vs2	n	ss1	vd	p	111100111111	@[!]p VSXD vd,ss1,vs2
1	0	1	0	0000000	000	000000000	vs2	n	ss1	vd	p	101100111111	@[!]p VLSEGXB vd,ss1,vs2,seglen
1	0	1	0	0000000	000	000000010	vs2	n	ss1	vd	p	101100111111	@[!]p VLSEGXH vd,ss1,vs2,seglen
1	0	1	0	0000000	000	000000100	vs2	n	ss1	vd	p	101100111111	@[!]p VLSEGXW vd,ss1,vs2,seglen
1	0	1	0	0000000	000	000000110	vs2	n	ss1	vd	p	101100111111	@[!]p VLSEGXD vd,ss1,vs2,seglen
1	0	1	0	0000000	000	000001000	vs2	n	ss1	vd	p	101100111111	@[!]p VLSEGXBU vd,ss1,vs2,seglen
1	0	1	0	0000000	000	000001010	vs2	n	ss1	vd	p	101100111111	@[!]p VLSEGXHU vd,ss1,vs2,seglen
1	0	1	0	0000000	000	000001100	vs2	n	ss1	vd	p	101100111111	@[!]p VLSEGXWU vd,ss1,vs2,seglen
1	0	1	0	0000000	000	000000000	vs2	n	ss1	vd	p	111100111111	@[!]p VSSEGXB vd,ss1,vs2,seglen
1	0	1	0	0000000	000	000000010	vs2	n	ss1	vd	p	111100111111	@[!]p VSSEGXH vd,ss1,vs2,seglen
1	0	1	0	0000000	000	000000100	vs2	n	ss1	vd	p	111100111111	@[!]p VSSEGXW vd,ss1,vs2,seglen

63	62	61	60	59 53	52 50	49 48 45 44 41 40	3837 36 35 34 33	32	31 2928 27	24 23 20 19	16 15 12 11	0	
d	1	2	f	funct7	funct3	funct9	vs2	n	vs1	vd	p	opcode	VR-type

Hwacha Vector Atomic Memory Instructions

1	s1	s2	0	00001	aq	rl	010	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOSWAP.W vd,rs1,rs2
1	s1	s2	0	00000	aq	rl	010	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOADD.W vd,rs1,rs2
1	s1	s2	0	00100	aq	rl	010	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOXOR.W vd,rs1,rs2
1	s1	s2	0	01100	aq	rl	010	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOAND.W vd,rs1,rs2
1	s1	s2	0	01000	aq	rl	010	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOOR.W vd,rs1,rs2
1	s1	s2	0	10000	aq	rl	010	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOMIN.W vd,rs1,rs2
1	s1	s2	0	10100	aq	rl	010	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOMAX.W vd,rs1,rs2
1	s1	s2	0	11000	aq	rl	010	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOMINU.W vd,rs1,rs2
1	s1	s2	0	11100	aq	rl	010	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOMAXU.W vd,rs1,rs2
1	s1	s2	0	00001	aq	rl	011	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOSWAP.D vd,rs1,rs2
1	s1	s2	0	00000	aq	rl	011	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOADD.D vd,rs1,rs2
1	s1	s2	0	00100	aq	rl	011	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOXOR.D vd,rs1,rs2
1	s1	s2	0	01100	aq	rl	011	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOAND.D vd,rs1,rs2
1	s1	s2	0	01000	aq	rl	011	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOOR.D vd,rs1,rs2
1	s1	s2	0	10000	aq	rl	011	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOMIN.D vd,rs1,rs2
1	s1	s2	0	10100	aq	rl	011	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOMAX.D vd,rs1,rs2
1	s1	s2	0	11000	aq	rl	011	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOMINU.D vd,rs1,rs2
1	s1	s2	0	11100	aq	rl	011	000000000	rs2	n	rs1	vd	p	010110111111	@[!]p VAMOMAXU.D vd,rs1,rs2

63	62	61	60	59 53	52 50	49 48 45 44 41 40	38 37 36 35 34 33 32	31 29	28 27 24 23	20 19 16 15	12 11 0	
d	1	2	f	funct7	funct3	funct9	vs2	n	vs1	vd	p	opcode	VR-type

Hwacha Vector Integer Arithmetic Instructions

1	s1	0	0	0000010	110	000000000	00000000	n	rs1	vd	p	011000111111	@[!]p VEIDX vd,rs1
0	1	0	0	0000011	110	000000000	00000000	n	vs1	sd	p	011000111111	@[!]p VFIRST sd,vs1
d	s1	s2	0	0000000	000	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VADD rd,rs1,rs2
d	s1	s2	0	1000000	000	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VADDU rd,rs1,rs2
d	s1	s2	0	0100000	000	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VSUB rd,rs1,rs2
d	s1	s2	0	0000000	001	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VSLL rd,rs1,rs2
d	s1	s2	0	0000000	010	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VSLT rd,rs1,rs2
d	s1	s2	0	0000000	011	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VSLTU rd,rs1,rs2
d	s1	s2	0	0000000	100	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VXOR rd,rs1,rs2
d	s1	s2	0	0000000	101	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VSRL rd,rs1,rs2
d	s1	s2	0	0100000	101	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VSRA rd,rs1,rs2
d	s1	s2	0	0000000	110	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VOR rd,rs1,rs2
d	s1	s2	0	0000000	111	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VAND rd,rs1,rs2
d	s1	s2	0	0000000	000	000000000	rs2	n	rs1	rd	p	011100111111	@[!]p VADDW rd,rs1,rs2
d	s1	s2	0	0100000	000	000000000	rs2	n	rs1	rd	p	011100111111	@[!]p VSUBW rd,rs1,rs2
d	s1	s2	0	0000000	001	000000000	rs2	n	rs1	rd	p	011100111111	@[!]p VSLLW rd,rs1,rs2
d	s1	s2	0	0000000	101	000000000	rs2	n	rs1	rd	p	011100111111	@[!]p VSRLW rd,rs1,rs2
d	s1	s2	0	0100000	101	000000000	rs2	n	rs1	rd	p	011100111111	@[!]p VSRAW rd,rs1,rs2
d	s1	s2	0	0000001	000	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VMUL rd,rs1,rs2
d	s1	s2	0	0000001	001	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VMULH rd,rs1,rs2
d	s1	s2	0	0000001	010	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VMULHSU rd,rs1,rs2
d	s1	s2	0	0000001	011	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VMULHU rd,rs1,rs2
d	s1	s2	0	0000001	100	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VDIV rd,rs1,rs2
d	s1	s2	0	0000001	101	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VDIVU rd,rs1,rs2
d	s1	s2	0	0000001	110	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VREM rd,rs1,rs2
d	s1	s2	0	0000001	111	000000000	rs2	n	rs1	rd	p	011000111111	@[!]p VREMU rd,rs1,rs2
d	s1	s2	0	0000001	000	000000000	rs2	n	rs1	rd	p	011100111111	@[!]p VMULW rd,rs1,rs2
d	s1	s2	0	0000001	100	000000000	rs2	n	rs1	rd	p	011100111111	@[!]p VDIVW rd,rs1,rs2
d	s1	s2	0	0000001	101	000000000	rs2	n	rs1	rd	p	011100111111	@[!]p VDIVUW rd,rs1,rs2
d	s1	s2	0	0000001	110	000000000	rs2	n	rs1	rd	p	011100111111	@[!]p VREMW rd,rs1,rs2
d	s1	s2	0	0000001	111	000000000	rs2	n	rs1	rd	p	011100111111	@[!]p VREMUW rd,rs1,rs2

63	62	61	60	59 53	52 50	49 48 45 44 41 40	3837 36 35 34 33	32	31 2928 27	24 23	20 19	16 15	12 11 0	
d	1	2	f	funct7	funct3	funct9	vs2	n	vs1	vd	p		opcode	VR-type
d	1	2	3	funct7	funct3	f vs3	vs2	n	vs1	vd	p		opcode	VR4-type

Hwacha Vector Floating-Point Arithmetic Instructions (Double-Precision)

d	s1	s2	s3	0000101	rm	0	rs3	rs2	n	rs1	rd	p	100000111111	@[!]p VFMADD.D rd,rs1,rs2,rs3,rm
d	s1	s2	s3	0000101	rm	0	rs3	rs2	n	rs1	rd	p	100010111111	@[!]p VFMSUB.D rd,rs1,rs2,rs3,rm
d	s1	s2	s3	0000101	rm	0	rs3	rs2	n	rs1	rd	p	100100111111	@[!]p VFNMSUB.D rd,rs1,rs2,rs3,rm
d	s1	s2	s3	0000101	rm	0	rs3	rs2	n	rs1	rd	p	100110111111	@[!]p VFNMADD.D rd,rs1,rs2,rs3,rm
d	s1	s2	0	0000101	rm	000000000		rs2	n	rs1	rd	p	101000111111	@[!]p VFADD.D rd,rs1,rs2,rm
d	s1	s2	0	0000101	rm	000000001		rs2	n	rs1	rd	p	101000111111	@[!]p VFSUB.D rd,rs1,rs2,rm
d	s1	s2	0	0000101	rm	000000010		rs2	n	rs1	rd	p	101000111111	@[!]p VFMUL.D rd,rs1,rs2,rm
d	s1	s2	0	0000101	rm	000000011		rs2	n	rs1	rd	p	101000111111	@[!]p VFDIV.D rd,rs1,rs2,rm
d	s1	0	0	0000101	rm	000001011		00000000	n	rs1	rd	p	101000111111	@[!]p VFSQRT.D rd,rs1,rm
d	s1	s2	0	0000101	000	000000100		rs2	n	rs1	rd	p	101000111111	@[!]p VFSGNJ.D rd,rs1,rs2
d	s1	s2	0	0000101	001	000000100		rs2	n	rs1	rd	p	101000111111	@[!]p VFSGNJN.D rd,rs1,rs2
d	s1	s2	0	0000101	010	000000100		rs2	n	rs1	rd	p	101000111111	@[!]p VFSGNJX.D rd,rs1,rs2
d	s1	s2	0	0000101	000	000000101		rs2	n	rs1	rd	p	101000111111	@[!]p VFMIN.D rd,rs1,rs2
d	s1	s2	0	0000101	001	000000101		rs2	n	rs1	rd	p	101000111111	@[!]p VFMAX.D rd,rs1,rs2
d	s1	0	0	0000100	rm	000001000		00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.D.S rd,rs1,rm
d	s1	0	0	0000110	rm	000001000		00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.D.H rd,rs1,rm
d	s1	0	0	0000101	001	000011100		00000000	n	rs1	rd	p	101000111111	@[!]p VFCLASS.D rd,rs1

Hwacha Vector Floating-Point Arithmetic Instructions (Single-Precision)

d	s1	s2	s3	0000000	rm	0	rs3	rs2	n	rs1	rd	p	100000111111	@[!]p VFMADD.S rd,rs1,rs2,rs3,rm
d	s1	s2	s3	0000000	rm	0	rs3	rs2	n	rs1	rd	p	100010111111	@[!]p VFMSUB.S rd,rs1,rs2,rs3,rm
d	s1	s2	s3	0000000	rm	0	rs3	rs2	n	rs1	rd	p	100100111111	@[!]p VFNMSUB.S rd,rs1,rs2,rs3,rm
d	s1	s2	s3	0000000	rm	0	rs3	rs2	n	rs1	rd	p	100110111111	@[!]p VFNMADD.S rd,rs1,rs2,rs3,rm
d	s1	s2	0	0000000	rm	000000000		rs2	n	rs1	rd	p	101000111111	@[!]p VFADD.S rd,rs1,rs2,rm
d	s1	s2	0	0000000	rm	000000001		rs2	n	rs1	rd	p	101000111111	@[!]p VFSUB.S rd,rs1,rs2,rm
d	s1	s2	0	0000000	rm	000000010		rs2	n	rs1	rd	p	101000111111	@[!]p VFMUL.S rd,rs1,rs2,rm
d	s1	s2	0	0000000	rm	000000011		rs2	n	rs1	rd	p	101000111111	@[!]p VFDIV.S rd,rs1,rs2,rm
d	s1	0	0	0000000	rm	000001011		00000000	n	rs1	rd	p	101000111111	@[!]p VFSQRT.S rd,rs1,rm
d	s1	s2	0	0000000	000	000000100		rs2	n	rs1	rd	p	101000111111	@[!]p VFSGNJ.S rd,rs1,rs2
d	s1	s2	0	0000000	001	000000100		rs2	n	rs1	rd	p	101000111111	@[!]p VFSGNJN.S rd,rs1,rs2
d	s1	s2	0	0000000	010	000000100		rs2	n	rs1	rd	p	101000111111	@[!]p VFSGNJX.S rd,rs1,rs2
d	s1	s2	0	0000000	000	000000101		rs2	n	rs1	rd	p	101000111111	@[!]p VFMIN.S rd,rs1,rs2
d	s1	s2	0	0000000	001	000000101		rs2	n	rs1	rd	p	101000111111	@[!]p VFMAX.S rd,rs1,rs2
d	s1	0	0	0000001	rm	000001000		00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.S.D rd,rs1,rm
d	s1	0	0	0000010	rm	000001000		00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.S.H rd,rs1,rm
d	s1	0	0	0000000	001	000011100		00000000	n	rs1	rd	p	101000111111	@[!]p VFCLASS.S rd,rs1

63	62	61	60	59 53	52 50	49 48	45 44 41 40	38 37 36 35 34 33	32	31 28	27 24	23 20	19 16	15 12	11 0	
d	1	2	f	funct7	funct3	funct9		vs2	n	vs1	vd	p			opcode	VR-type
d	1	2	3	funct7	funct3	f	vs3	vs2	n	vs1	vd	p			opcode	VR4-type

Hwacha Vector Floating-Point Arithmetic Instructions (Half-Precision)

d	s1	s2	s3	0001010	rm	0	rs3	rs2	n	rs1	rd	p	100000111111	@[!]p VFMADD.H rd,rs1,rs2,rs3,rm
d	s1	s2	s3	0001010	rm	0	rs3	rs2	n	rs1	rd	p	100010111111	@[!]p VFMSUB.H rd,rs1,rs2,rs3,rm
d	s1	s2	s3	0001010	rm	0	rs3	rs2	n	rs1	rd	p	100100111111	@[!]p VFNMSUB.H rd,rs1,rs2,rs3,rm
d	s1	s2	s3	0001010	rm	0	rs3	rs2	n	rs1	rd	p	100110111111	@[!]p VFNMADD.H rd,rs1,rs2,rs3,rm
d	s1	s2	0	0001010	rm	000000000		rs2	n	rs1	rd	p	101000111111	@[!]p VFADD.H rd,rs1,rs2,rm
d	s1	s2	0	0001010	rm	000000001		rs2	n	rs1	rd	p	101000111111	@[!]p VFSUB.H rd,rs1,rs2,rm
d	s1	s2	0	0001010	rm	000000010		rs2	n	rs1	rd	p	101000111111	@[!]p VFMUL.H rd,rs1,rs2,rm
d	s1	s2	0	0001010	rm	000000011		rs2	n	rs1	rd	p	101000111111	@[!]p VFDIV.H rd,rs1,rs2,rm
d	s1	0	0	0001010	rm	000001011		00000000	n	rs1	rd	p	101000111111	@[!]p VFSQRT.H rd,rs1,rm
d	s1	s2	0	0001010	000	000000100		rs2	n	rs1	rd	p	101000111111	@[!]p VFSGNJ.H rd,rs1,rs2
d	s1	s2	0	0001010	001	000000100		rs2	n	rs1	rd	p	101000111111	@[!]p VFSGNJN.H rd,rs1,rs2
d	s1	s2	0	0001010	010	000000100		rs2	n	rs1	rd	p	101000111111	@[!]p VFSGNJX.H rd,rs1,rs2
d	s1	s2	0	0001010	000	000000101		rs2	n	rs1	rd	p	101000111111	@[!]p VFMIN.H rd,rs1,rs2
d	s1	s2	0	0001010	001	000000101		rs2	n	rs1	rd	p	101000111111	@[!]p VFMAX.H rd,rs1,rs2
d	s1	0	0	0001001	rm	000001000		00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.H.D rd,rs1,rm
d	s1	0	0	0001000	rm	000001000		00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.H.S rd,rs1,rm
d	s1	0	0	0001010	001	000011100		00000000	n	rs1	rd	p	101000111111	@[!]p VFCLASS.H rd,rs1

Hwacha Vector Floating-Point Arithmetic Instructions (Widening-Precision)

d	s1	s2	0	0000110	rm	000000000		rs2	n	rs1	rd	p	101000111111	@[!]p VFADD.S.H rd,rs1,rs2,rm
d	s1	s2	0	0000110	rm	000000001		rs2	n	rs1	rd	p	101000111111	@[!]p VFSUB.S.H rd,rs1,rs2,rm
d	s1	s2	0	0000110	rm	000000010		rs2	n	rs1	rd	p	101000111111	@[!]p VFMUL.S.H rd,rs1,rs2,rm
d	s1	s2	s3	0000010	rm	0	rs3	rs2	n	rs1	rd	p	100000111111	@[!]p VFMADD.D.H rd,rs1,rs2,rs3,rm
d	s1	s2	s3	0000010	rm	0	rs3	rs2	n	rs1	rd	p	100010111111	@[!]p VFMSUB.D.H rd,rs1,rs2,rs3,rm
d	s1	s2	s3	0000010	rm	0	rs3	rs2	n	rs1	rd	p	100100111111	@[!]p VFNMSUB.D.H rd,rs1,rs2,rs3,rm
d	s1	s2	s3	0000010	rm	0	rs3	rs2	n	rs1	rd	p	100110111111	@[!]p VFNMADD.D.H rd,rs1,rs2,rs3,rm
d	s1	s2	0	0000010	rm	000000000		rs2	n	rs1	rd	p	101000111111	@[!]p VFADD.D.H rd,rs1,rs2,rm
d	s1	s2	0	0000010	rm	000000001		rs2	n	rs1	rd	p	101000111111	@[!]p VFSUB.D.H rd,rs1,rs2,rm
d	s1	s2	0	0000010	rm	000000010		rs2	n	rs1	rd	p	101000111111	@[!]p VFMUL.D.H rd,rs1,rs2,rm
d	s1	s2	0	0000001	rm	000000000		rs2	n	rs1	rd	p	101000111111	@[!]p VFADD.D.S rd,rs1,rs2,rm
d	s1	s2	0	0000001	rm	000000001		rs2	n	rs1	rd	p	101000111111	@[!]p VFSUB.D.S rd,rs1,rs2,rm
d	s1	s2	0	0000001	rm	000000010		rs2	n	rs1	rd	p	101000111111	@[!]p VFMUL.D.S rd,rs1,rs2,rm

63	62	61	60	59 53	52 50	49 48 45 44 41 40	3837 36 35 34 33	32	31 2928 27 24 23	20 19 16 15	12 11 0	
d	1	2	f	funct7	funct3	funct9	vs2	n	vs1	vd	p	opcode	VR-type

Hwacha Vector Floating-Point to/from Integer Conversion Instructions

d	s1	0	0	0000001	rm	000011000	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.W.D rd,rs1,rm
d	s1	0	0	0000101	rm	000011000	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.WU.D rd,rs1,rm
d	s1	0	0	0001001	rm	000011000	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.L.D rd,rs1,rm
d	s1	0	0	0001101	rm	000011000	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.LU.D rd,rs1,rm
d	s1	0	0	0000001	rm	000011010	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.D.W rd,rs1,rm
d	s1	0	0	0000101	rm	000011010	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.D.WU rd,rs1,rm
d	s1	0	0	0001001	rm	000011010	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.D.L rd,rs1,rm
d	s1	0	0	0001101	rm	000011010	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.D.LU rd,rs1,rm
d	s1	0	0	0000000	rm	000011000	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.W.S rd,rs1,rm
d	s1	0	0	0000100	rm	000011000	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.WU.S rd,rs1,rm
d	s1	0	0	0001000	rm	000011000	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.L.S rd,rs1,rm
d	s1	0	0	0001100	rm	000011000	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.LU.S rd,rs1,rm
d	s1	0	0	0000000	rm	000011010	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.S.W rd,rs1,rm
d	s1	0	0	0000100	rm	000011010	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.S.WU rd,rs1,rm
d	s1	0	0	0001000	rm	000011010	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.S.L rd,rs1,rm
d	s1	0	0	0001100	rm	000011010	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.S.LU rd,rs1,rm
d	s1	0	0	0000010	rm	000011000	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.W.H rd,rs1,rm
d	s1	0	0	0000110	rm	000011000	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.WU.H rd,rs1,rm
d	s1	0	0	0001010	rm	000011000	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.L.H rd,rs1,rm
d	s1	0	0	0001110	rm	000011000	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.LU.H rd,rs1,rm
d	s1	0	0	0000010	rm	000011010	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.H.W rd,rs1,rm
d	s1	0	0	0000110	rm	000011010	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.H.WU rd,rs1,rm
d	s1	0	0	0001010	rm	000011010	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.H.L rd,rs1,rm
d	s1	0	0	0001110	rm	000011010	00000000	n	rs1	rd	p	101000111111	@[!]p VFCVT.H.LU rd,rs1,rm

63	62	61	60	59 53	52 50	49 48 45 44 41	40 38 37 36 35 34 33	32	31 29 28 27 24	23 20	19 16	15 12	11 0	
d	1	2	f	funct7	funct3	funct9	vs2	n	vs1	vd		p	opcode	VR-type
d	1	2	3	funct7	funct3	f vs3	vs2	n	vs1	vd		p	opcode	VR4-type

Hwacha Vector Compare Instructions

1	s1	s2	0	0000000	000	100000000	rs2	n	rs1	0000	pd	p	011000111111	@[!]p VCMPEQ pd,rs1,rs2
1	s1	s2	0	0000000	100	100000000	rs2	n	rs1	0000	pd	p	011000111111	@[!]p VCMPLT pd,rs1,rs2
1	s1	s2	0	0000000	110	100000000	rs2	n	rs1	0000	pd	p	011000111111	@[!]p VCMPLTU pd,rs1,rs2
1	s1	s2	0	0000101	010	000010100	rs2	n	rs1	0000	pd	p	011000111111	@[!]p VCMPFEQ.D pd,rs1,rs2
1	s1	s2	0	0000101	001	000010100	rs2	n	rs1	0000	pd	p	011000111111	@[!]p VCMPFLT.D pd,rs1,rs2
1	s1	s2	0	0000101	000	000010100	rs2	n	rs1	0000	pd	p	011000111111	@[!]p VCMPFLE.D pd,rs1,rs2
1	s1	s2	0	0000000	010	000010100	rs2	n	rs1	0000	pd	p	011000111111	@[!]p VCMPFEQ.S pd,rs1,rs2
1	s1	s2	0	0000000	001	000010100	rs2	n	rs1	0000	pd	p	011000111111	@[!]p VCMPFLT.S pd,rs1,rs2
1	s1	s2	0	0000000	000	000010100	rs2	n	rs1	0000	pd	p	011000111111	@[!]p VCMPFLE.S pd,rs1,rs2
1	s1	s2	0	0001010	010	000010100	rs2	n	rs1	0000	pd	p	011000111111	@[!]p VCMPFEQ.H pd,rs1,rs2
1	s1	s2	0	0001010	001	000010100	rs2	n	rs1	0000	pd	p	011000111111	@[!]p VCMPFLT.H pd,rs1,rs2
1	s1	s2	0	0001010	000	000010100	rs2	n	rs1	0000	pd	p	011000111111	@[!]p VCMPFLE.H pd,rs1,rs2

Hwacha Vector Predicate Memory Instructions

1	0	0	0	0000000	000	100000000	00000000	0	000	as1	0000	pd	0000	101100111111	@all VPL pd,as1
1	0	0	0	0000000	000	100000000	00000000	0	000	as1	0000	pd	0000	111100111111	@all VPS pd,as1

Hwacha Vector Predicate Arithmetic Instructions

1	0	0	0	01	op	1	0000	ps3	0000	ps2	0	0000	ps1	0000	pd	0000	011000111111	@all VPOP pd,ps1,ps2,ps3,op
1	0	0	0	01	00000000	1	0000	0000	0000	0000	0	0000	0000	0000	pd	0000	011000111111	@all VPCLEAR pd
1	0	0	0	01	11111111	1	0000	0000	0000	0000	0	0000	0000	0000	pd	0000	011000111111	@all VPSET pd
1	0	0	0	01	01101001	1	0000	ps3	0000	ps2	0	0000	ps1	0000	pd	0000	011000111111	@all VPXORXOR pd,ps1,ps2,ps3
1	0	0	0	01	01111101	1	0000	ps3	0000	ps2	0	0000	ps1	0000	pd	0000	011000111111	@all VPXOROR pd,ps1,ps2,ps3
1	0	0	0	01	00010100	1	0000	ps3	0000	ps2	0	0000	ps1	0000	pd	0000	011000111111	@all VPXORAND pd,ps1,ps2,ps3
1	0	0	0	01	01101010	1	0000	ps3	0000	ps2	0	0000	ps1	0000	pd	0000	011000111111	@all VPORXOR pd,ps1,ps2,ps3
1	0	0	0	01	01111111	1	0000	ps3	0000	ps2	0	0000	ps1	0000	pd	0000	011000111111	@all VPOROR pd,ps1,ps2,ps3
1	0	0	0	01	00010101	1	0000	ps3	0000	ps2	0	0000	ps1	0000	pd	0000	011000111111	@all VPORAND pd,ps1,ps2,ps3
1	0	0	0	01	01010110	1	0000	ps3	0000	ps2	0	0000	ps1	0000	pd	0000	011000111111	@all VPANDXOR pd,ps1,ps2,ps3
1	0	0	0	01	01010111	1	0000	ps3	0000	ps2	0	0000	ps1	0000	pd	0000	011000111111	@all VPANDOR pd,ps1,ps2,ps3
1	0	0	0	01	00000001	1	0000	ps3	0000	ps2	0	0000	ps1	0000	pd	0000	011000111111	@all VPANDAND pd,ps1,ps2,ps3

63	62	61	60	59 53	52 50	49 48 45 44 41 40	38 37 36 35 34 33	32	31 29	28 27 24 23	20 19 16 15	12 11 0	
d	1	2	f	funct7	funct3	funct9	vs2	n	vs1	vd	p	opcode	VR-type

Hwacha Scalar Load/Store Instructions

0	0	0	0	0000000	000	010000000	00000000	0	000	as1	sd	0000	101100111111	@s VLAB sd,as1
0	0	0	0	0000000	000	010000010	00000000	0	000	as1	sd	0000	101100111111	@s VLAH sd,as1
0	0	0	0	0000000	000	010000100	00000000	0	000	as1	sd	0000	101100111111	@s VLAW sd,as1
0	0	0	0	0000000	000	010000110	00000000	0	000	as1	sd	0000	101100111111	@s VLAD sd,as1
0	0	0	0	0000000	000	010001000	00000000	0	000	as1	sd	0000	101100111111	@s VLABU sd,as1
0	0	0	0	0000000	000	010001010	00000000	0	000	as1	sd	0000	101100111111	@s VLAHU sd,as1
0	0	0	0	0000000	000	010001100	00000000	0	000	as1	sd	0000	101100111111	@s VLAWU sd,as1
0	0	0	0	0000000	000	010000000	ss2	0	000	as1	00000000	0000	111100111111	@s VSAB as1,ss2
0	0	0	0	0000000	000	010000010	ss2	0	000	as1	00000000	0000	111100111111	@s VSAH as1,ss2
0	0	0	0	0000000	000	010000100	ss2	0	000	as1	00000000	0000	111100111111	@s VSAW as1,ss2
0	0	0	0	0000000	000	010000110	ss2	0	000	as1	00000000	0000	111100111111	@s VSAD as1,ss2
0	0	0	0	0000000	000	000000000	00000000	0	ss1		sd	0000	101100111111	@s VLSB sd,ss1
0	0	0	0	0000000	000	000000010	00000000	0	ss1		sd	0000	101100111111	@s VLSH sd,ss1
0	0	0	0	0000000	000	000000100	00000000	0	ss1		sd	0000	101100111111	@s VLSW sd,ss1
0	0	0	0	0000000	000	000000110	00000000	0	ss1		sd	0000	101100111111	@s VLSD sd,ss1
0	0	0	0	0000000	000	000001000	00000000	0	ss1		sd	0000	101100111111	@s VLSBU sd,ss1
0	0	0	0	0000000	000	000001010	00000000	0	ss1		sd	0000	101100111111	@s VLSHU sd,ss1
0	0	0	0	0000000	000	000001100	00000000	0	ss1		sd	0000	101100111111	@s VLSWU sd,ss1
0	0	0	0	0000000	000	000000000	ss2	0	ss1		00000000	0000	111100111111	@s VSSB ss1,ss2
0	0	0	0	0000000	000	000000010	ss2	0	ss1		00000000	0000	111100111111	@s VSSH ss1,ss2
0	0	0	0	0000000	000	000000100	ss2	0	ss1		00000000	0000	111100111111	@s VSSW ss1,ss2
0	0	0	0	0000000	000	000000110	ss2	0	ss1		00000000	0000	111100111111	@s VSSD ss1,ss2

63 62 61 60 59 53 52 50 49 48	45 44	41 40 38	37 36 35 34 33	32	31 29 28 27	24 23	20 19	16 15	12	11 0	
imm[31:3]			c2	n	vs1	vd		p	opcode		VJ-type
imm[31:0]					funct8	vd		funct4	opcode		VU-type
imm[31:0]					vs1	vd		funct4	opcode		VI-type

Hwacha Scalar Arithemtic Instructions

imm[31:0]			00000000	sd	0000	011010111111	@s VLUI sd,imm
imm[31:0]			00000000	sd	0000	001010111111	@s VAUIPC sd,imm
imm[31:0]			ss1	sd	0000	001000111111	@s VADDI sd,ss1,imm
imm[31:0]			ss1	sd	0010	001000111111	@s VSLTI sd,ss1,imm
imm[31:0]			ss1	sd	0011	001000111111	@s VSLTIU sd,ss1,imm
imm[31:0]			ss1	sd	0100	001000111111	@s VXORI sd,ss1,imm
imm[31:0]			ss1	sd	0110	001000111111	@s VORI sd,ss1,imm
imm[31:0]			ss1	sd	0111	001000111111	@s VANDI sd,ss1,imm
00000000000000000000000000000000	shamt		ss1	sd	0001	001000111111	@s VSLLI sd,ss1,shamt
00000000000000000000000000000000	shamt		ss1	sd	0101	001000111111	@s VSRLI sd,ss1,shamt
00000100000000000000000000000000	shamt		ss1	sd	0101	001000111111	@s VSRAI sd,ss1,shamt
imm[31:0]			ss1	sd	0000	001100111111	@s VADDIW sd,ss1,imm
00000000000000000000000000000000	shamt		ss1	sd	0001	001100111111	@s VSLLIW sd,ss1,shamt
00000000000000000000000000000000	shamt		ss1	sd	0101	001100111111	@s VSRLIW sd,ss1,shamt
00000100000000000000000000000000	shamt		ss1	sd	0101	001100111111	@s VSRAIW sd,ss1,shamt

Hwacha Control Flow Instructions

00000000_00000000_00000000_00000	00	0	00000000	00000000	0000	110000111111	@all VSTOP
00000000_00100000_00000000_00000	00	0	00000000	prev succ	0000	110000111111	@all VFENCE
imm[31:3]	c2	n	00000000	sd	p	110110111111	@[!]p VCJAL sd,imm
imm[31:3]	c2	n	ss1	sd	p	110010111111	@[!]p VCJALR sd,ss1,imm

8 History

The Hwacha vector-fetch architecture builds on several earlier projects: T0, Scale, Maven, and three early versions of Hwacha. This section outlines the lineage and history of vector machines that influenced the current vector machine design.

8.1 Lineage

The T0 (Torrent-0) vector microprocessor project at UC Berkeley and ICSI begun in 1992 with Krste Asanović as the lead architect and RTL designer, and Brian Kingsbury and Bertrand Irrisou as main VLSI implementers [3, 4, 1, 2]. T0 was a vector processor based on the MIPS-II ISA, implemented in Hewlett-Packard's CMOS26G 1.0 um CMOS process with 2 metal layers. The resulting $16.75 \times 16.75\ mm^2$ chip operated at a maximum frequency of 45 MHz at 5 V and consumed less than 12 W. The T0 vector machine had a custom-designed 5-read-3-write register file that contained 16 vector registers, each holding 32×32-bit elements, split across 8 vector lanes each with two dynamically reconfigurable vector arithmetic pipelines, and interfaced with external SRAM as main memory.

The Scale (Software-Controlled Architecture for Low Energy) vector-thread architecture project at MIT begun in 2000 with Ronny Krashinsky and Christopher Batten as lead architects [9, 6, 7, 8]. The Scale vector processor had an SMIPS control processor (SMIPS stands for Scale MIPS, a subset of MIPS without the branch delay slot), and a vector-thread unit with 4 lanes each with 4 clusters that featured different types of execution resources. The instructions that ran on the vector-thread unit were grouped into an atomic instruction block (AIB), where all the instructions in an AIB were executed before moving on to the next element in the vector. Scale included a do-across network for efficient cross-element communication, which were particularly useful when vectorizing loops with loop-carried dependencies. Scale implemented a cache refill/access decoupling scheme to hide memory latency. Scale did not support floating-point operations in hardware. The Scale chip was implemented in TSMC CL018G 180 nm CMOS process with 6 metal layers. The resulting $23.14\ mm^2$ chip operated at a maximum frequency of 260 MHz at 1.8 V consuming 0.4–1.1 W, depending on the workload.

The Maven (Malleable Array of Vector-thread ENgines) vector-thread architecture project at UC Berkeley and MIT begun in 2007 with Christopher Batten as the lead architect, and Yunsup Lee and Rimas Avizienis as main implementers of the Maven vector-thread unit and scalar units respectively [5, 11, 10, 12]. The goal of the Maven project was to explore the programmability and efficiency of a wide range of data-parallel accelerators including the MIMD, traditional vector, and the newly proposed Maven vector-thread architecture. The group designed a flexible RTL framework that instantiated all designs, which were then pushed through the VLSI flow to obtain accurate area, performance, and power/energy numbers to compare. The Maven design was never

fabricated, however, many VLSI layouts were produced using the TSMC 65GPLUS 65 nm CMOS process with 9 metal layers. The size of the resulting accelerator were around 4–6.3 mm^2, and operated at a maximum frequency of 680–763 MHz at 1 V consuming 111–331 mW. The Maven vector-thread unit was configurable to have 1, 2, or 4 lanes, each with a vector register file that was optionally banked 1, 2, or 4 ways. Vector arithmetic instructions were packed into separate vector-fetch blocks to let the vector memory instructions run ahead and prefetch the needed vector data. Branch instructions were allowed in vector-fetch blocks to support kernels with irregular control flow, and were implicitly handled by the hardware (meaning that the hardware was responsible for the bookkeeping the divergence state). Explicit predication was not supported. Floating-point operations were supported by the hardware.

8.2 Previous versions of Hwacha

The Hwacha project started right after the Maven project in 2011. The first version of Hwacha had a similar assembly programming model as Maven, where the vector memory instructions were kept in the control processor's instruction stream, and the vector arithmetic instructions were hoisted out into a separate vector-fetch block. Branches were not allowed in vector-fetch blocks, and predication was not supported. Conditional move instructions were the only way to write data-dependent execution. The Hwacha project used the 64-bit RISC-V ISA [15] as its base ISA, moving away from the Maven ISA. The vector microarchitecture changed significantly, where the vector lane was mainly redesigned to work with a banked vector register file that was split into 8 banks of area-efficient 1-read-1-write SRAM macros. The RTL was written from scratch in Verilog.

The second version of Hwacha was rewritten from scratch in an early version of Chisel, and mainly added support for virtual memory and restartable exceptions [14]. The vector runahead unit was rearchitected to prefetch vector data into the nearest cache as a result.

The third version of Hwacha was rewritten from scratch in Chisel. This rewrite happened after we had learned better ways of expressing hardware designs in Chisel. The control logic and the vector memory unit were mostly rewritten in a cleaner way. Starting from this version, the vector unit talked to the L2 cache directly rather than the L1 data cache.

8.3 Current version of Hwacha

The fourth version is the one explained in this document. This version of the Hwacha machine has been rewritten from scratch in Chisel yet again. Since the third version, the vector lane has been rearchitected to use four 128-bit SRAM macros opposed to eight 64-bit macros that were used in the previous versions. The assembly programming model has changed, and now all vector instructions are hoisted out into a vector-fetch block. This version supports full predication on all

vector instructions. Consensual branches, reduction operations, and variable latency operations such as floating-point divide and square root, integer divide and remainder operations are also supported.

8.4 Collaboration

Many people have made contributions to the Hwacha project. The Hwacha vector accelerator was developed by Yunsup Lee, Albert Ou, Colin Schmidt, Sagar Karandikar, Krste Asanović, and others from 2011 through 2015. As the lead architect of Hwacha, Yunsup Lee directed the development and evaluation of the architecture, microarchitecture, RTL, compiler, verification framework, microbenchmarks, and application kernels. Albert Ou was primarily responsible for the RTL implementation of the Vector Memory Unit and mixed-precision extensions. Colin Schmidt took the lead on the definition of the Hwacha ISA, RTL implementation of the scalar unit, C++ functional ISA simulator, vector torture test generator, Hwacha extensions to the GNU toolchain port, and the OpenCL compiler and benchmark suite. Sagar Karandikar took the lead on the bar-crawl tool for design-space exploration, VLSI floorplanning, RTL implementation of the Vector Runahead Unit, ARM Mali-T628 MP6 GPU evaluation, and the assembly microbenchmark suite. Palmer Dabbelt took the lead on the physical design flow and post-PAR gate-level simulation in the 28 nm process technology. Henry Cook took the lead on the RTL implementation of the uncore components, including the L2 cache and the TileLink cache coherence protocol. Howard Mao took the lead on dual LPDDR3 memory channel support and provided critical fixes for the outer memory system. Andrew Waterman took the lead on the definition of the RISC-V ISA, the RISC-V GNU toolchain port, and the RTL implementation of the Rocket core. Andrew also helped to define the Hwacha ISA. John Hauser took the lead on developing the hardware floating-point units. Many others contributed to the surrounding infrastructure, such as the Rocket Chip SoC generator. Huy Vo, Stephen Twigg, and Quan Nguyen contributed to older versions of Hwacha. Finally, Krste Asanović was integral in all aspects of the project.

8.5 Funding

The Hwacha project has been partially funded by the following sponsors.

- **Par Lab**: Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support came from Par Lab affiliates: Nokia, NVIDIA, Oracle, and Samsung.
- **Silicon Photonics**: DARPA POEM program, Award HR0011-11-C-0100.
- **ASPIRE Lab**: DARPA PERFECT program, Award HR0011-12-2-0016. The Center for Future Architectures Research (C-FAR), a STARnet center funded by the Semiconductor Re-

search Corporation. Additional support came from ASPIRE Lab industrial sponsors and affiliates: Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung.

- **NVIDIA graduate fellowship**

Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

References

[1] K. Asanović. Torrent Architecture Manual. Technical report, EECS Department, University of California, Berkeley, Dec 1996.

[2] K. Asanović. Vector Microprocessors. PhD Thesis, EECS Department, University of California, Berkeley, 1998.

[3] K. Asanović, J. Beck, B. Irissou, B. Kingsbury, N. Morgan, , and J. Wawrzynek. The T0 Vector Microprocessor. *Proceedings Hot Chips VII*, Aug 1995.

[4] K. Asanović, B. Kingsbury, B. Irissou, J. Beck, , and J. Wawrzynek. T0: A Single-Chip Vector Microprocessor with Reconfigurable Pipelines. *22nd European Solid-State Circuits Conference (ESSCIRC-1996)*, Sep 1996.

[5] C. Batten. Simplified Vector-Thread Architectures for Flexible and Efficient Data-Parallel Accelerators. PhD Thesis, MIT, 2010.

[6] C. Batten, R. Krashinsky, S. Gerding, and K. Asanović. Cache Refill/Access Decoupling for Vector Machines. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2004.

[7] R. Krashinsky. Vector-Thread Architecture and Implementation. PhD Thesis, MIT, 2007.

[8] R. Krashinsky, C. Batten, and K. Asanović. Implementing the Scale Vector-Thread Processor. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 13(3), Jul 2008.

[9] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The Vector-Thread Architecture. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[10] Y. Lee. Efficient VLSI Implementations of Vector-Thread Architectures. MS Thesis, UC Berkeley, 2011.

[11] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2011.

[12] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the Tradeoffs Between Programmability and Efficiency in Data-Parallel Accelerators. *ACM Trans. Comput. Syst.*, 31(3):6:1–6:38, Aug 2013.

[13] R. M. Russell. The Cray-1 Computer System. *Communications of the ACM*, 21(1):63–72, Jan 1978.

[14] H. Vo, Y. Lee, A. Waterman, and K. Asanović. A Case for OS-Friendly Hardware Accelerators. *Workshop on the Interaction between Operating System and Computer Architecture (WIVOSCA), at the International Symposium on Computer Architecture (ISCA)*, Jun 2013.

[15] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.