# The Hwacha Microarchitecture Manual, Version 3.8.1

*Yunsup Lee*
*Albert Ou*
*Colin Schmidt*
*Sagar Karandikar*
*Howard Mao*
*Krste Asanović*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-263
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-263.html

December 19, 2015

# The Hwacha Microarchitecture Manual
# Version 3.8.1

Yunsup Lee, Albert Ou, Colin Schmidt, Sagar Karandikar, Howard Mao, Krste Asanović
CS Division, EECS Department, University of California, Berkeley
{yunsup|aou|colins|sagark|zhemao|krste}@eecs.berkeley.edu

December 19, 2015

# Contents

# 1 Introduction

This work-in-progress document outlines the Hwacha decoupled vector-fetch microarchitecture in detail. We first discuss how we modified the open-source Rocket Chip SoC generator to provide a system framework comparable to commercially available data-parallel accelerators. We exploit the generator's RTL libraries, including an in-order core implementing the RISC-V instruction set, multiple levels of coherent caches, and a standardized accelerator interface we used to attached the Hwacha vector accelerator. The vector accelerator executes the Hwacha instruction set architecture described in the Hwacha vector-fetch architecture manual. We present the overall machine organization, then describe the details of the vector frontend and the scalar unit, vector execution unit (VXU), vector memory unit (VMU), and the vector runahead unit (VRU).

## 2 System Architecture

Figure 1 illustrates the overall system architecture of the Hwacha vector microprocessor. We use the open-source Rocket Chip SoC generator to elaborate our design [5]. The generator consists of highly parameterized RTL libraries written in Chisel [1]. In this section, we discuss the salient capabilities of the generator that allows us to integrate the Hwacha vector accelerator productively within a modern SoC environment while being efficient and providing a simple assembly programming model.

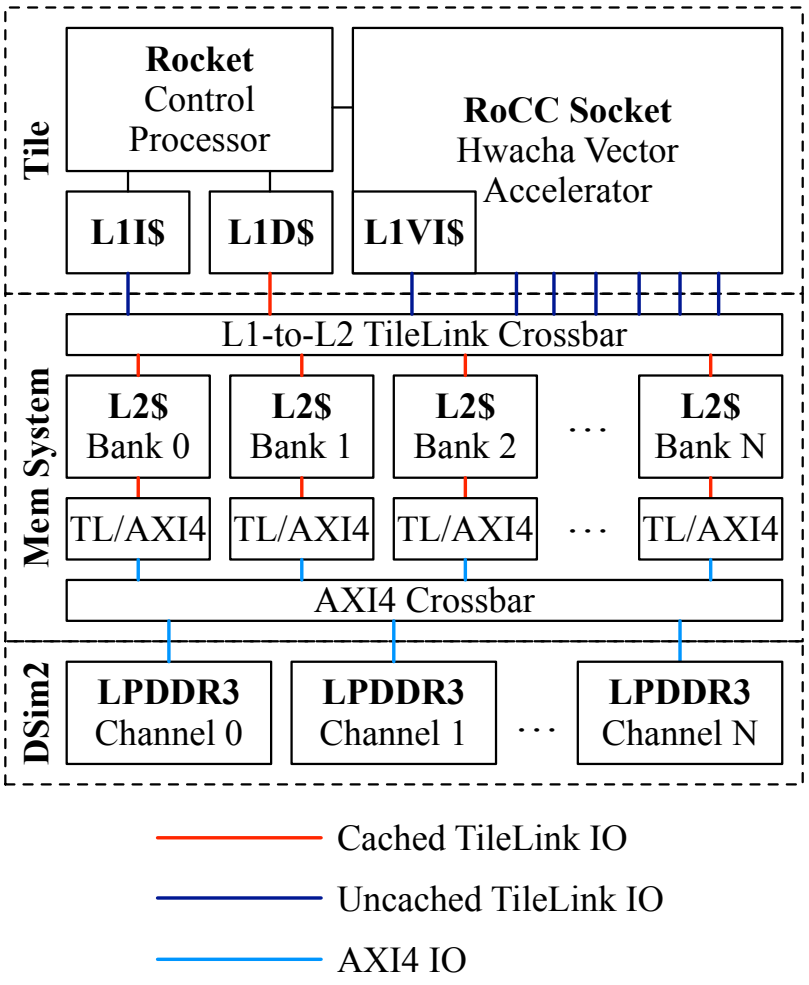


**Figure 1: System architecture provided by the Rocket Chip SoC generator.**

A tile consists of a Rocket control processor and a RoCC (Rocket Custom Coprocessor) socket. Rocket is a five-stage in-order RISC-V scalar core that talks to its private blocking L1 instruction cache and non-blocking L1 data cache [5]. The RoCC socket provides a standardized interface for issuing commands to a custom accelerator, as well as an interface to the memory system. The

Hwacha decoupled-vector accelerator, alongside its blocking vector instruction cache, is designed to fit within the RoCC socket. The control thread and the worker thread of the Hwacha assembly programming model (consult the Hwacha vector-fetch architecture manual) are mapped to the Rocket control processor and Hwacha vector accelerator respectively.

The shared L2 cache is banked, set-associative, and fully inclusive of the L1 caches. Addresses are interleaved at cache line granularity across banks. The tile and L2 cache banks are connected through an on-chip network that implements the TileLink cache coherence protocol [3]. There are two flavors of TileLink IO: cached and uncached. The cached TileLink interface is used by clients that create private copies of cache blocks such as the L1 data cache and L2 cache banks. These blocks are kept coherent throughout the memory system. The uncached TileLink interface is used for clients that do not keep private copies, such as the vector unit. Note, instruction caches use uncached TileLink IO, as the cache-coherence protocol does not keep the content of those caches coherent with respect to the data stream. The L2 cache banks are coherence master endpoints that implement a cache-coherence protocol, which is selected during elaboration. There is also an option to accelerate the protocol using directory bits that live in the L2 cache tag array.

TileLink offers several capabilities in support of the vector accelerator. *Cache coherence* between the L1 data cache and the vector accelerator preserves the shared memory abstraction between the control processor and vector accelerator. This keeps the assembly programming model simple. There is no need to keep two separate address spaces: one for host memory and the other for accelerator's target memory. When the vector accelerator makes a read request to a cache line, the L2 cache bank looks up the directory bit to quickly determine whether the cache line resides in the L1 data cache. If so, the L2 cache bank will then take appropriate steps to guarantee that the L1 data cache does not hold any dirty data. The mechanism the L2 cache bank uses to meet the guarantee depends on the cache-coherence protocol. If the cache line has an exclusive state, the L2 cache bank will send a message to the L1 data cache requesting the line be downgraded to shared. If the cache line is already in a shared state, no action is needed. When the vector accelerator makes a write request, the L2 will check whether the cache line is in a shared or exclusive state and send a message to the L1 asking it to drop the line, if necessary. *Sub-cache-block accesses* to words within a cache line reduce memory bandwidth for vector gathers and scatters. *Data prefetch requests*, which the L2 efficiently merges with subsequent sub-block accesses, help the system overlap data transfer and computation to achieve better bandwidth utilization. *Atomic memory operations*, which are performed by ALUs inside each L2 cache bank, offload the work of reduction computations.

The refill ports of the L2 cache banks are connected to a bank of cached TileLink IO to AXI4 converters. The AXI4 interfaces are then routed to the appropriate LPDDR3 channels through the AXI4 crossbars. The LPDDR3 memory channels are implemented in the testbench, where the DRAM timing is simulated using DRAMSim2 [6].

The memory system parameters, such as the cache size, associativity, number of L2 cache banks

and memory channels, and cache-coherence protocol, are set from a configuration object during elaboration. The configuration object also holds design parameters for the Rocket control processor and the Hwacha vector accelerator.

The Hwacha vector accelerator is influenced by several system-level decisions inherent to the Rocket Chip SoC generator. In mapping the control thread to the Rocket scalar core, we exploit vector-fetch decoupling to push the limits of in-order processors. The unified and coherent virtual address space enables restartable exceptions for vector instructions. By connecting the accelerators to the L2 cache instead of the L1 data cache, we have traded off longer average access latency for substantially higher bandwidth to the cache. However, this decision makes memory access coalescing a more important design feature for the accelerator. Exploiting these built-in SoC generator features allowed us to substantially improve the capabilities of the Hwacha vector accelerator while simultaneously making it simple to apply as much parameter tuning as possible to balance the memory system and vector accelerator designs.

# 3 Hwacha Machine Organization

Hwacha combines ideas from access/execute decoupling [7], decoupled vector architectures [4], and cache refill/access decoupling [2], applying them to work within a cache-coherent memory system without any risk of deadlocking. Extensive decoupling enables the microarchitecture to effectively tolerate long and variable memory latencies with an in-order design.

Figure 2 presents the high-level anatomy of the vector accelerator. Hwacha is situated as a discrete coprocessor with its own independent frontend. This vector-fetch decoupling relieves the control processor to resolve address calculations for upcoming vector fetch blocks, among other bookkeeping actions, well in advance of the accelerator.

Hwacha consists of one or more replicated vector lanes assisted by a scalar unit. Internally, the lane is bifurcated into two major components: the Vector Execution Unit (VXU), which encompasses the vector and predicate register files and the functional units, and the Vector Memory Unit (VMU), which coordinates data movement between the VXU and the memory system.

Hwacha also features a Vector Runahead Unit (VRU) that exploits the inherent regularity of constant-stride vector memory accesses for aggressive yet extremely accurate prefetching. Unlike out-of-order cores with SIMD that rely on reorder buffers and GPUs which rely on multithreading, the Hwacha architecture is particularly amenable to prefetching without requiring a large amount of state.

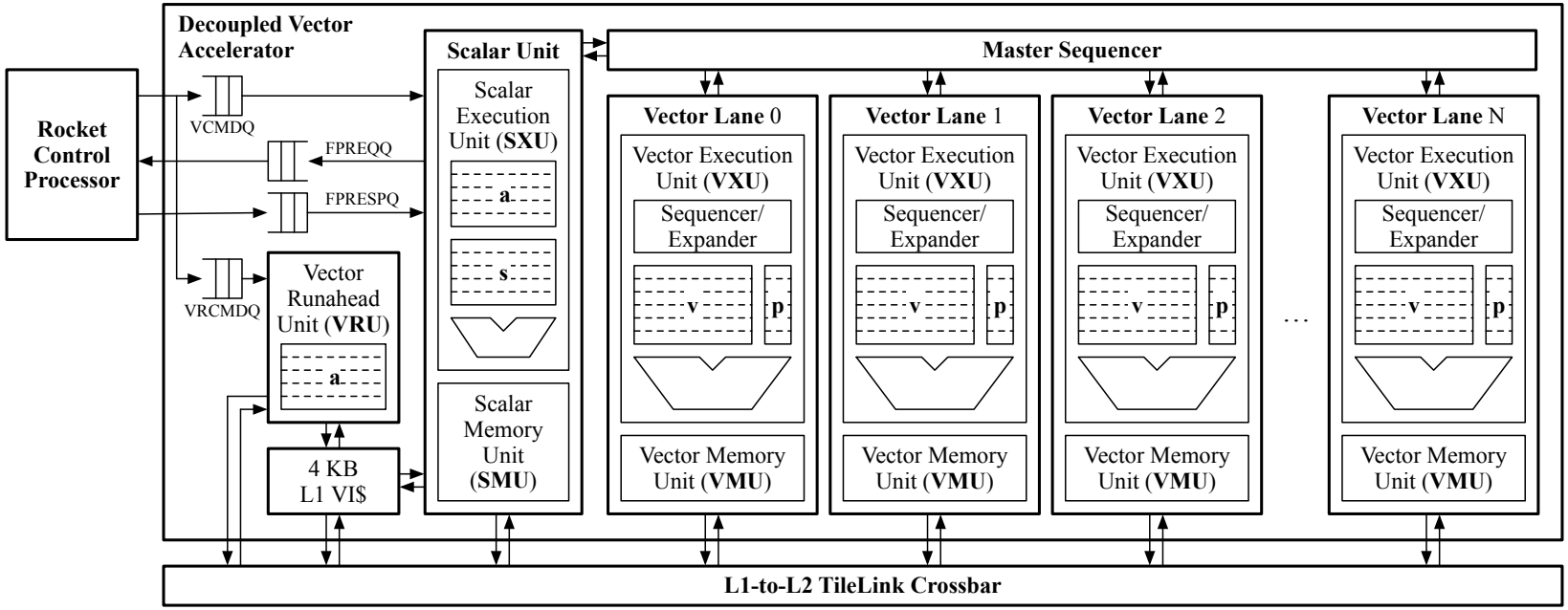


**Figure 2: Block Diagram of the Decoupled Vector Accelerator.** VCMDQ = vector command queue, VRCMDQ = vector runahead command queue, FPREQQ = floating-point request queue, FPRESPQ = floating-point response queue.

# 4 RoCC Frontend and Scalar Unit

Control thread instructions arrive through the Vector Command Queue (VCMDQ). Upon encountering a `vf` command, the scalar unit begins fetching at the accompanying PC from the 4 KB two-way set-associative vector instruction cache, continuing until it reaches a `vstop` in the vector-fetch block.

The scalar unit includes the address and shared register files and possesses a fairly conventional single-issue, in-order, four-stage pipeline. It handles purely scalar computation, loads, and stores, as well as the resolution of consensual branches and reductions resulting from the vector lanes. The FPU is shared with the Rocket control processor. At the decode stage, vector instructions are steered to the lanes along with any scalar operands.

# 5 Vector Execution Unit

The VXU, depicted in Figure 3, is broadly organized around four banks. Each contains a 256x128b 1R1W 8T SRAM that forms a portion of the vector register file (VRF), alongside a 256x2b 3R1W predicate register file (PRF). Also private to each bank are a local integer ALU and PLU. A crossbar connects the banks to the long-latency functional units, grouped into clusters whose members share the same operand, predicate, and result lines.

Vector instructions are issued into the sequencer, which monitors the progress of every active operation within that particular lane. The master sequencer, shared among all lanes, holds the common dependency information and other static state. Execution is managed in "strips" that complete eight 64 b elements worth of work, corresponding to one pass through the banks. The sequencer acts as an out-of-order, albeit non-speculative, issue window: hazards are continuously examined for each operation; when clear for the next strip, an age-based arbitration scheme determines which ready operation to send to the expander.

The expander converts a sequencer operation into its constituent micro-ops (μops), low-level control signals that directly drive the lane datapath. These are inserted into shift registers with the displacement of read and write μops coinciding exactly with the functional unit latency.

The μops iterate through the elements as they sequentially traverse the banks cycle by cycle. As demonstrated by the bank execution example in Figure 4, this stall-free systolic schedule sustains $n$ operands per cycle to the shared functional units after an initial $n$-cycle latency. Variable-latency functional units instead deposit results into per-bank queues for decoupled writes, and the sequencer monitors retirement asynchronously. Vector chaining arises naturally from interleaving μops belonging to different operations.

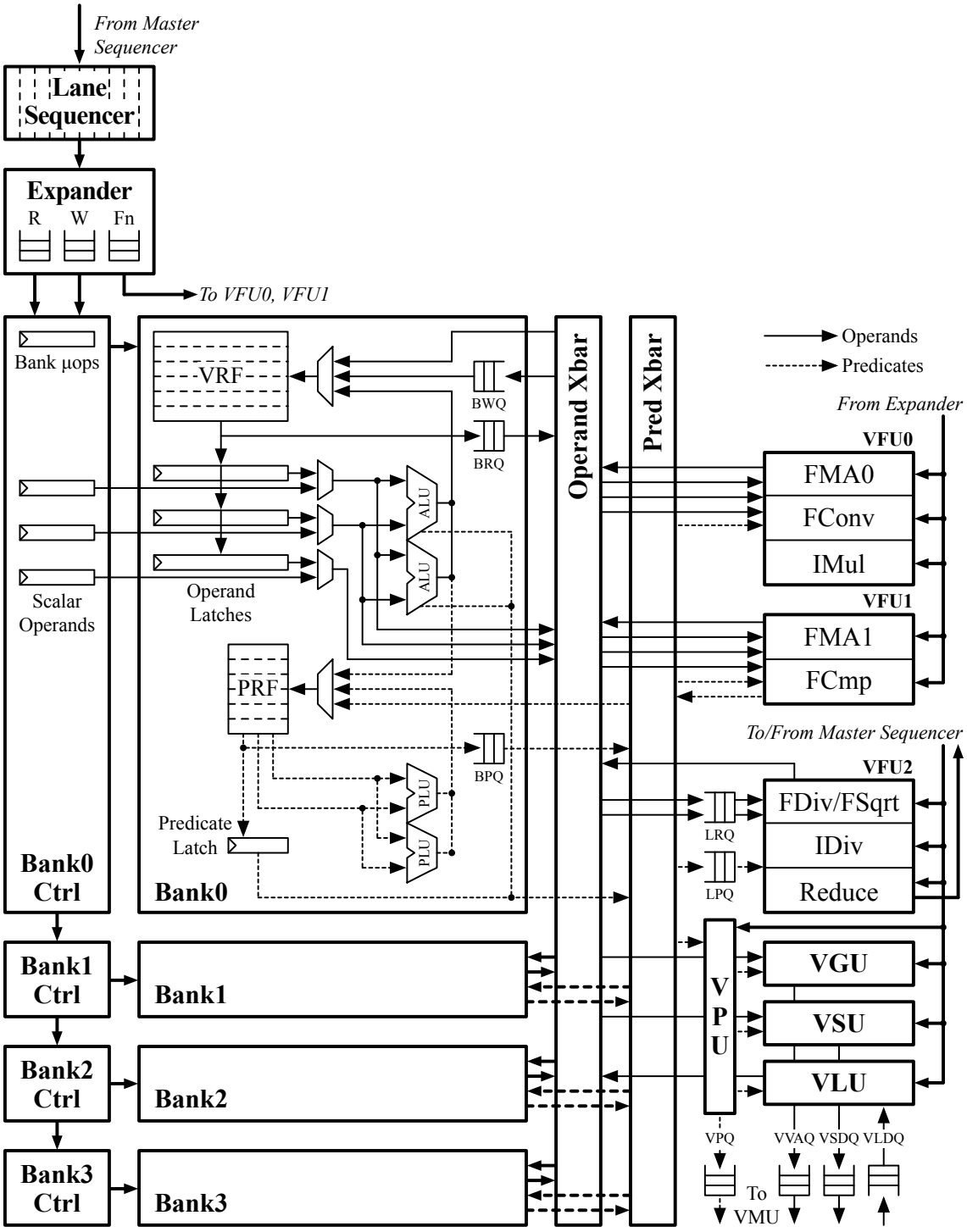


**Figure 3: Block Diagram of Vector Execution Unit (VXU).** VRF = vector register file, PRF = predicate register file, ALU = arithmetic logic unit, PLU = predicate logic unit, BRQ = bank operand read queue, BWQ = bank operand write queue, BPQ = bank predicate read queue, LRQ = lane operand read queue, LPQ = lane predicate read queue, VFU = vector functional unit, FP = floating-point, FMA = FP fused multiply add unit, FConv = FP conversion unit, FCmp = FP compare unit, FDiv/FSqrt = FP divide/square-root unit, IMul = integer multiply unit, IDiv = integer divide unit, VPU = vector predicate unit, VGU = vector address generation unit, VSU = vector store-data unit, VLU = vector load-data unit, VPQ = vector predicate queue, VVAQ = vector virtual address queue, VSDQ = vector store-data queue, VLDQ = vector load-data queue.

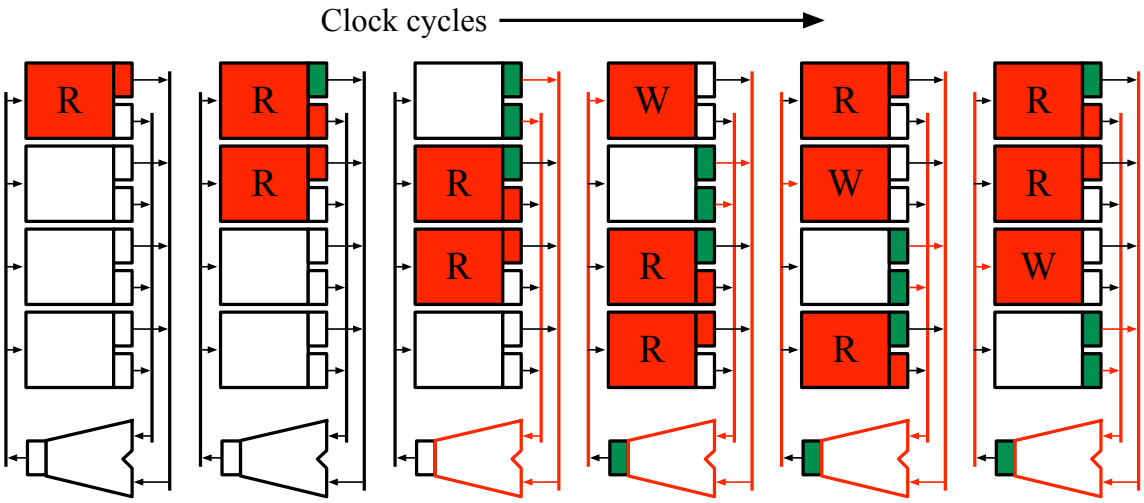


**Figure 4: Systolic Bank Execution Diagram.** In this example, after a 2-cycle initial startup latency, the banked register file is effectively able to read out 2 operands per cycle.

# 6 Vector Memory Unit

The per-lane VMUs are each equipped with a 128 b interface to the shared L2 cache. This arrangement delivers high memory bandwidth, albeit with a trade-off of increased latency that is overcome by decoupling the VMU from the rest of the vector unit. Figure 5 outlines the organization of the VMU.

As a memory operation is issued to the lane, the VMU command queue is populated with the operation type, vector length, base address, and stride. Address generation for constant-stride accesses proceeds without VXU involvement. For indexed operations such as gathers, scatters, and AMOs, the Vector Generation Unit (VGU) reads offsets from the VRF into the Vector Virtual Address Queue (VVAQ). Virtual addresses are then translated and deposited into the Vector Physical Address Queue (VPAQ), and the progress is reported to the VXU. The departure of requests is regulated by the lane sequencer to facilitate restartable exceptions.

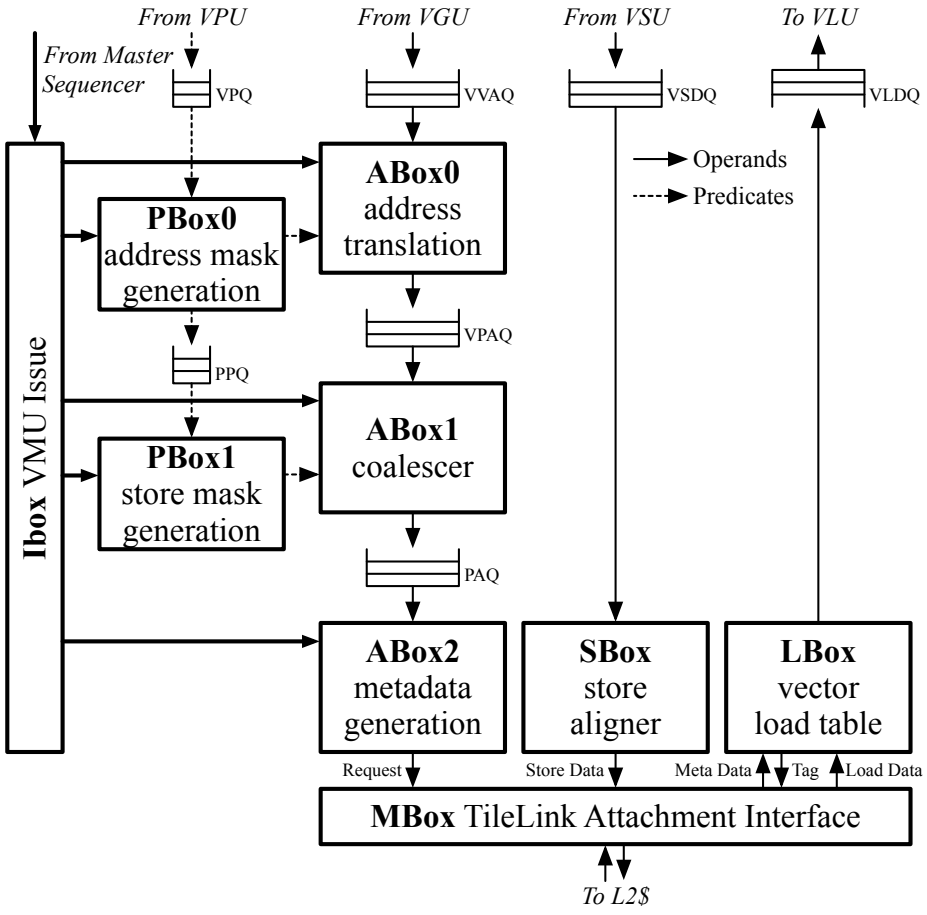


**Figure 5: Block Diagram of Vector Memory Unit (VMU).** Consult Figure 3 for VPU, VGU, VSU, VLU, VPQ, VVAQ, VSDQ, VLDQ. VPAQ = vector physical address queue, PPQ = pipe predicate queue, PAQ = pipe address queue.

The address pipeline is assisted by a separate predicate pipeline. Predicates must be examined to determine whether a page fault is genuine, and are used to derive the store masks. The VMU supports limited density-time skipping given power-of-2 runs of false predicates.

Unit strides represent a very common case for which the VMU is specifically optimized. The initial address generation and translation occur at a page granularity to circumvent predicate latency and accelerate the lane sequencer check. To more fully utilize the available memory bandwidth, adjacent elements are coalesced into a single request prior to dispatch. The VMU correctly handles edge cases with base addresses not 128 b-aligned and irregular vector lengths not a multiple of the packing density [8].

The Vector Store Unit (VSU) multiplexes elements read from the VRF banks into the Vector Store Data Queue (VSDQ). An aligner module following the VSDQ shifts the entries appropriately for scatters and unit-stride stores with non-ideal alignment.

In reverse, the Vector Load Unit (VLU) routes data from the Vector Load Data Queue (VLDQ) to their respective banks. As the memory system may arbitrarily order responses, two VLU optimizations become crucial. The first is an opportunistic writeback mechanism that permits the VRF to accept elements out of sequence; this reduces latency and area compared to a reorder buffer. The VLU is also able to simultaneously manage multiple operations to avoid artificial throttling of successive loads by the VMU.

# 7 Vector Runahead Unit

The Vector Runahead Unit (VRU), shown in Figure 6, takes advantage of the decoupled nature of the Hwacha architecture to hide memory latency and maximize functional unit utilization. Unlike out-of-order machines with SIMD that rely on the reorder buffer for decoupling and GPUs which rely on multithreading, the Hwacha design is particularly amenable to prefetching without relying on large amounts of state.

The VRU has a separate vector runahead command queue (VRCMDQ) between Hwacha and the Rocket control processor. It receives the current vector length from `vsetvl` commands as well as addressing information from `vmca` commands, which it stores in an internal copy of the vector address register file. Upon receiving a `vf` command, the VRU fetches instructions from Hwacha's L1 vector instruction cache and decodes unit-strided load and store instructions. Using the previously collected address information along with the vector length, the VRU issues prefetching

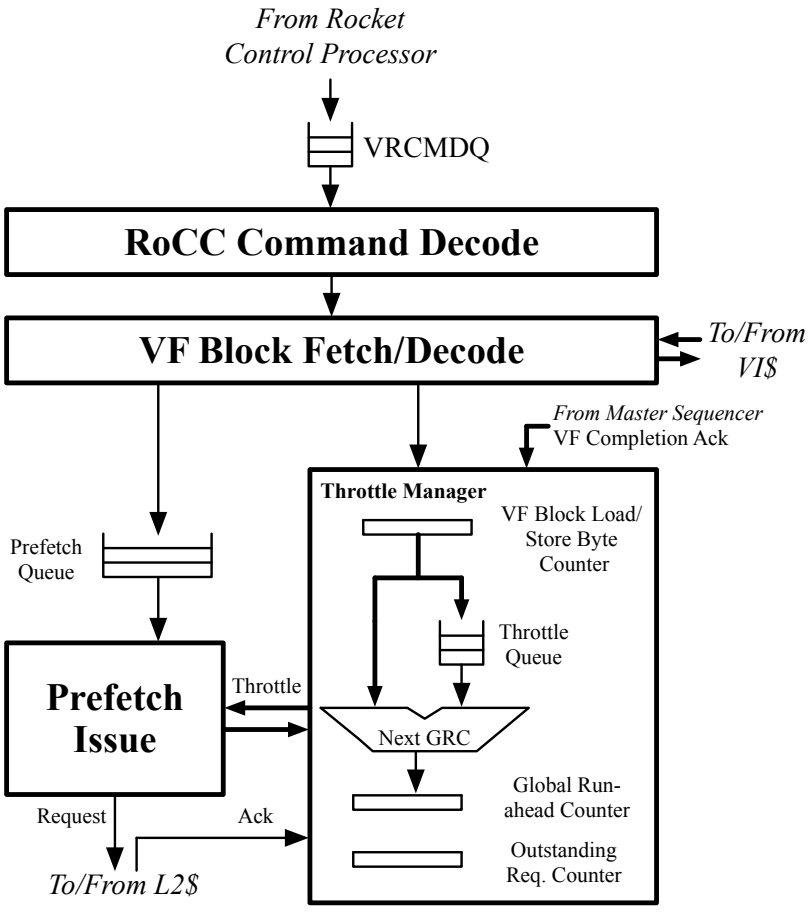


**Figure 6: Block Diagram of Vector Runahead Unit (VRU).** VRCMDQ = vector runahead command queue, VF = vector fetch, VI$ = vector instruction cache, GRC = global runahead counter.

commands directly to the L2, in anticipation of loads and stores issued by the vector lanes. Unlike in other machines, these prefetches are in most cases non-speculative. Since the address registers and vector length cannot be changed by the worker thread, the VRU will be certain what data is being fetched at each vector load and store instruction.

Efficiently using L2 tracking resources and managing the runahead distance are critical to balancing latency-hiding with allowing the rest of Hwacha to make forward-progress at a reasonable pace. We limit the VRU to using at most one-third of the outstanding access trackers in the L2 cache, since in the unit strided case, the VRU's prefetch blocks are twice as large as the execution unit's loads and stores.

In managing the runahead-distance of the VRU, the controller must avoid two extremes. A VRU that runs too close to real-time execution risks invoking a performance penalty. This penalty arises not only from the obvious inability to hide latency, but also because the VRU wastes L2 tracking resources and creates a hotspot around one bank of the L2 cache. A VRU that runs too far ahead of real-time execution has the potential to remove items from the L2 that are in-use or that have been prefetched but not yet used.

To prevent the VRU from running too close to the execution units, we ignore a small number of vector fetch blocks at startup. We observe that sacrificing the prefetch of the loads and stores from one or two initial vector fetch blocks greatly increases the ability of the VRU to runahead in a steady state. To prevent the VRU from running too far ahead of the execution units, we implement a throttling scheme that counts the total number of bytes of loads and stores that the VRU has decoded but that have not yet been encountered by the execution units. In a vector processor like Hwacha, this scheme is hindered by conditional execution of loads and stores in vector fetch blocks using predication. Our scheme ensures that the counts in the VRU's throttle mechanism are synchronized at the end of each vector fetch block, regardless of the presence of unexecuted loads and stores due to predication and consensual branches. In our scheme, the VRU maintains a queue containing individual load/store byte counters for each vector fetch block that the VRU has seen, but that has not been acknowledged by the vector lanes. A global counter is also incremented by this per-block count of bytes whenever the VRU finishes decoding a vector fetch block. When the vector lanes complete the execution of a vector fetch block, an acknowledgement is sent to the VRU, which pops an entry off of the load/store byte count queue and decrements the global load/store byte counter by the appropriate amount. This global counter is then used to throttle the runahead distance of the VRU.

# 8 Multilane Configuration

Hwacha is parameterized to support any power-of-2 number of identical lanes. Although the master sequencer issues operations to all lanes synchronously, each lane executes entirely decoupled from one another.

To achieve more uniform load-balancing, elements of a vector are striped across the lanes by a runtime-configurable multiple of the sequencer strip size (the "lane stride"), as shown in Figure 7. This also simplifies the base calculation for memory operations of arbitrary constant stride, enabling the VMU to reuse the existing address generation datapath as a short iterative multiplier. The striping does introduce gaps in the unit-stride operations performed by an individual VMU, but the VMU issue unit can readily compensate by decomposing the vector into its contiguous segments, while the rest of the VMU remains oblivious. Unfavorable alignment, however, incurs a modest waste of bandwidth as adjacent lanes request the same cache line at these segment boundaries. Figure 8 provides an example of such a situation.

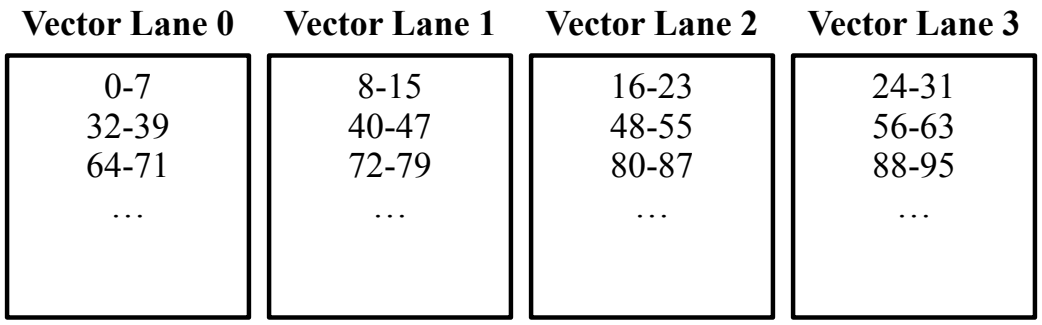


**Figure 7: Mapping of elements across a four-lane machine.**

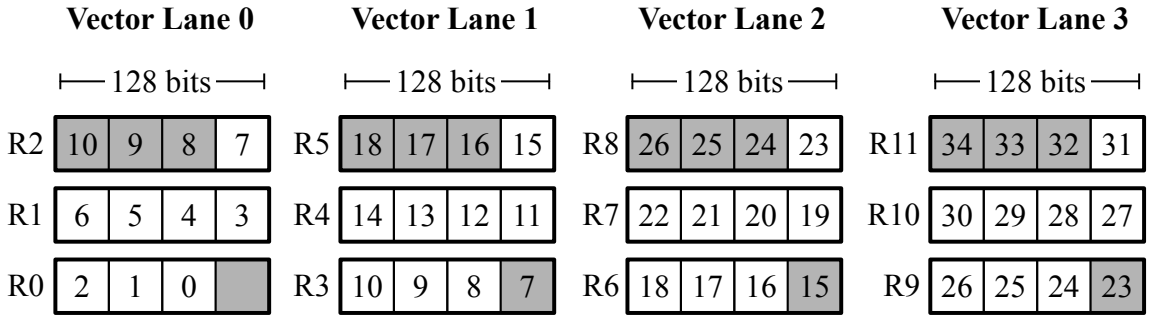


**Figure 8: Example of redundant memory requests by adjacent lanes.** These occur when the base address of a unit-strided vector in memory is not aligned at the memory interface width (128 bits)—in this case, `0x???????4`. Each block represents a 128 b TileLink beat containing four 32 b elements. Shaded cells indicate portions of a request ignored by each lane. Note that R2 overlaps with R3, R5 with R6, etc.

# 9 Design Space

Table 1 lists a relevant subset of Chisel configuration parameters that can be adjusted to tune the Hwacha design at elaboration time.

**Table 1: Hwacha tunable parameters and default values.**

| Parameter | Description | Default Value |
|---|---|---|
| HwachaNLanes | Number of vector lanes | 1 |
| HwachaNSeqEntries | Number of sequencer entries | 8 |
| HwachaStagesALU | Number of ALU pipeline stages | 1 |
| HwachaStagesPLU | Number of PLU pipeline stages | 0 |
| HwachaStagesIMul | Number of IMul pipeline stages | 3 |
| HwachaStagesDFMA | Number of double-precision FMA pipeline stages | 4 |
| HwachaStagesSFMA | Number of single-precision FMA pipeline stages | 3 |
| HwachaStagesHFMa | Number of half-precision FMA pipeline stages | 3 |
| HwachaStagesFConv | Number of FConv pipeline stages | 2 |
| HwachaStagesFCmp | Number of FCmp pipeline stages | 1 |
| HwachaNVVAQEntries | Number of VVAQ entries | 4 |
| HwachaNVPAQEntries | Number of VPAQ entries | 24 |
| HwachaNVSDQEntries | Number of VSDQ entries | 4 |
| HwachaNVLDQEntries | Number of VLDQ entries | 4 |
| HwachaNVLTEntries | Number of Vector Load Table entries | 64 |
| HwachaNDTLB | Number of data TLB entries | 8 |
| HwachaNPTLB | Number of prefetch TLB entries | 2 |
| HwachaLocalScalarFPU | Instantiate separate FPU for scalar unit | False |
| HwachaBuildVRU | Instantiate VRU | True |
| HwachaConfMixedPrec | Enable Mixed Precision | False |

# 10 History

The detailed project history is described in the history section of the Hwacha vector-fetch architecture manual.

## 10.1 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 Research 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] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, June 2012.

[2] C. Batten, R. Krashinsky, S. Gerding, and K. Asanović. Cache Refill/Access Decoupling for Vector Machines. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 331–342, Washington, DC, USA, 2004. IEEE Computer Society.

[3] H. M. Cook, A. S. Waterman, and Y. Lee. TileLink Cache Coherence Protocol Implementation, 2015.

[4] R. Espasa and M. Valero. Decoupled vector architectures. In *High-Performance Computer Architecture, 1996. Proceedings., Second International Symposium on*, pages 281–290, Feb 1996.

[5] Y. Lee, A. Waterman, R. Avižienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović. A 45nm 1.3GHz 16.7 Double-Precision GFLOPS/W RISC-V Processor with Vector Accelerators. In *2014 European Solid-State Circuits Conference (ESSCIRC-2014)*, Venice, Italy, Sept. 2014.

[6] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters*, 10(1):16–19, Jan. 2011.

[7] J. E. Smith. Decoupled Access/Execute Computer Architectures. *ACM Trans. Comput. Syst.*, 2(4):289–308, Nov. 1984.

[8] J. E. Smith, G. Faanes, and R. A. Sugumar. Vector instruction set support for conditional operations. In A. D. Berenbaum and J. S. Emer, editors, *27th International Symposium on Computer Architecture*, pages 260–269. IEEE Computer Society, 2000.