The Saturn Microarchitecture Manual

Jerry Zhao
Daniel Grubb
Miles Rusch
Tianrui Wei
Kevin Anderson
Borivoje Nikolic
Krste Asanović

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-215
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-215.html

December 14, 2024

Acknowledgement

Research was partially funded by SLICE Lab industrial sponsors and affiliates, and by the NSF CCRI ENS Chipyard Award #2016662. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

The Saturn Microarchitecture Manual

Jerry Zhao, Daniel Grubb, Miles Rusch, Tianrui Wei,
Kevin Anderson, Borivoje Nikolic, Krste Asanovic
CS Division, EECS Department, University of California, Berkeley
{jzh|dpgrubb|miles.rusch|tianruiwei|kevinand|bora|krste}@berkeley.edu

November 1, 2024

Contents

1 Introduction

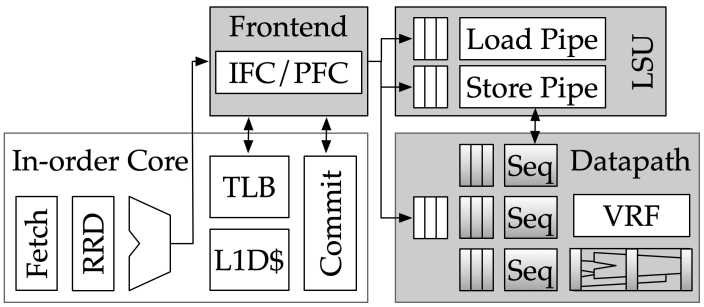

Figure 1: A high-level overview of the Saturn Vector Unit

This manual describes the Saturn Vector Unit, a parameterized and extensible vector microarchitecture executing the RISC-V vector extension. Saturn was developed to address an academic need for a representative, compliant, and flexible generator of RISC-V vector units targeting deployment in domain-specialized cores. Saturn is implemented as a parameterized Chisel RTL generator, enabling a range of possible Saturn configurations across many target deployment scenarios. This document discusses the microarchitectural details of all Saturn components.

- Section 2 describes the motivation for Saturn and compares Saturn's design approach to those of existing data-parallel microarchitecture archetypes
- Section 3 discusses the system organization of Saturn
- Section 4 describes the microarchitecture of Saturn's vector frontend unit
- Section 5 describes the microarchitecture of Saturn's vector load-store unit
- Section 6 describes the microarchitecture of Saturn's datapath and vector instruction sequencers
- Section 7 provides guidance on writing efficient vector code for Saturn
- Section 8 discusses the historical context of Saturn within past academic vector units

1.1 Objectives

Saturn was developed with the following objectives:

- Provide a **representative baseline** implementation of the RISC-V Vector specification
- Support **full compliance** with the complete RVV specification, including support for virtual memory and precise faults
- Target **ASIC implementations**, rather than FPGA deployments
- Be sufficiently **parameterized** to support configurations across a wide power/performance/area design space
- Demonstrate efficient scheduling of vector operations on a microarchitecture with a **short hardware vector length**
- Implement a **SIMD-style** microarchitecture, comparable to existing SIMD datapaths in DSP microarchitectures
- Integrate with existing **efficient area-compact scalar cores**, rather than high-IPC general-purpose cores
- Support **extensibility** with custom vector instructions, functional units, and accelerators that leverage the baseline capability in the standard RVV ISA
- Target deployment as part of a **DSP core** or similarly domain-specialized core, instead of general-purpose systems

2 Background

This chapter presents a background discussion of deployment scenarios for data-parallel systems and the dominant architectures in the commercial space for such systems. Saturn, as an implementation of a modern scalable vector ISA targeting deployment in specialized cores, fills an underexplored niche in the space of open-source data-parallel microarchitectures. A comparison of Saturn's microarchitecture philosophy to alternative vector approaches is discussed.

2.1 Data-parallel Workloads

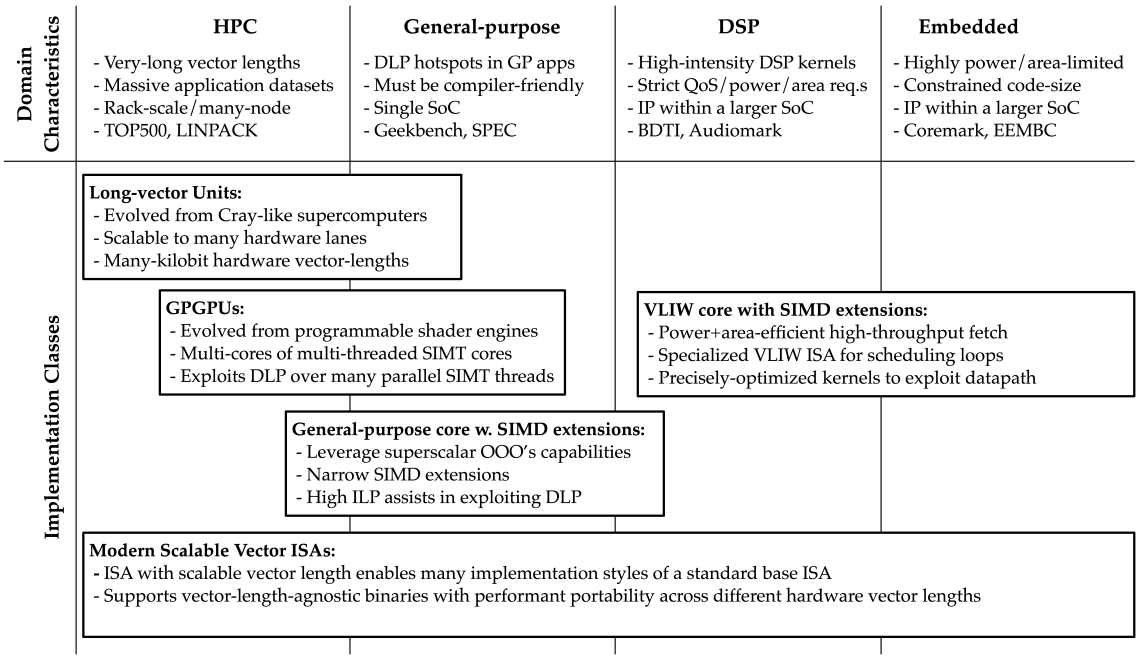

Figure 2: Taxonomy of data-parallel domains and their dominant architectures

We broadly categorize the range of deployment scenarios and workloads for programmable data-parallel systems into four domains. These categories are differentiated by power/area constraints and workload characteristics. Figure 2 outlines the four deployment domains where DLP-enabled architectures are critical, and the salient architectural and microarchitectural archetypes that have come to dominate each domain.

HPC, scientific computing, and ML training workloads are typically characterized by very long application vector lengths, exhibiting a high degree of data-level parallelism. Systems targeting these workloads are generally designed and deployed at datacenter scale, with available

data-level-parallelism exploited across the many processors in a multi-node cluster. While single-core efficiency and performance remain important for this domain, scalability towards multi-core, multi-node systems is of more critical concern.

General-purpose server, desktop, and mobile workloads intersperse data-level-parallel codes with sequential, control-flow heavy non-data-parallel blocks. The implications of Amdahl's Law dictate the design of these systems; exploiting data-level parallelism must not come at the expense of single-thread performance or thread-level-parallelism. Compared to the other domains, compiler and programmer-friendly ISAs are particularly important for these systems, as they must remain performant on the greatest diversity of workloads.

Digital signal processing (DSP) and similar domain-specialized workloads exploit fundamentally data-parallel algorithms. These workloads are often offloaded in modern SoCs to specialized "DSP cores" optimized towards power and area-efficient exploitation of data-level parallelism, with less consideration towards general-purpose performance. "DSP cores" usually run optimized kernels written by programmers at a low level to maximize efficiency. Large SoCs typically integrate many such "DSP cores" across various design points to provide optimized compute for different subsystems.

Applications executing on embedded devices face extreme constraints around power, area, and code size. Exploiting data-level parallelism can be critical for these workloads as well, but might come as a secondary concern compared to power and area. As these cores often provide a self-contained memory system, code and data size are more critical architectural considerations.

2.2 Data-parallel Implementation Archetypes

The requirements of the common data-parallel workload domains have induced the commercial space to converge towards several fundamental architectural and microarchitectural archetypes.

2.2.1 Long-vector Implementations

Long-vector ISAs and their **many-lane vector-unit** implementations trace their heritage most clearly to the early Cray-style vector units. Modern incarnations of such systems primarily target the HPC and datacenter space, where long application vector lengths map cleanly into high-capacity vector register files.

Microarchitecturally, these implementations distribute the vector register file, functional units, and memory access ports across many vector lanes, with scalability towards many lanes being a key requirement. A single core of such a vector unit might have upwards of 32 parallel vector lanes, with a hardware vector length of over 16384 bits.

Notably, a lane-oriented microarchitecture with distributed memory ports and a high-radix memory interconnect enables high performance on codes which require high address generation

throughput. Indexed loads and stores can leverage the distributed memory access ports to maintain high memory utilization, while register-register scatters and gathers can leverage a complex yet scalable cross-lane interconnect.

The NEC Aurora [1] is a modern example of a commercial long-vector microarchitecture, while Hwacha [2], Ara [3], and Vitruvius [4] are examples of academic long-vector implementations.

2.2.2 SIMT GP-GPUs

GP-GPUs execute a specialized **SIMT** ISA across a multi-core of multi-threaded processors. GP-GPUs evolved from the need for increasingly programmable shader pipelines in graphics workloads. Programming models for shaders developed into general-purpose SIMT frameworks like CUDA and OpenCL. GPU implementations similarly became more general-purpose, eventually maturing into modern GP-GPU microarchitectures.

Notably, the multi-core SIMT archetype in GP-GPUs naturally adapts to a range of deployment domains. By scaling the number of multi-threaded cores, GP-GPU microarchitectures can target integrated GPUs in general-purpose SoCs, graphics accelerator cards for desktop/consumer systems, as well as interconnected compute cards for datacenter-scale HPC systems. The rise of GP-GPUs is a contributing factor to the decline of specialized long-vector microarchitectures for datacenter HPC systems.

In the commercial space, NVIDIA and AMD dominate the GP-GPU market. Their HPC and datacenter products, while originally derived from consumer GPU architectures, have since diverged towards a more differentiated, yet still multi-core-SIMT microarchitecture.

Within the general-purpose mobile consumer space, AMD, Intel, Apple, Qualcomm, and Huawei all deploy embedded GP-GPU systems within their SoCs. These vendors all provide developer SDKs for supporting general purpose applications on their GP-GPU microarchitectures.

2.2.3 General-purpose ISAs with SIMD

General-purpose ISAs with SIMD extensions enable exploiting data-level parallelism in a microarchitecture already optimized towards scalar IPC. SIMD extensions evolved from subword-SIMD ISAs designed to pack a small fixed-length vector of sub-register-width elements within a single scalar register. Subword-SIMD is a cheap way to integrate limited forms of data-parallel compute, particularly for multimedia applications, into a general-purpose processor, without requiring substantial redesign of complex microarchitectural components. Modern SIMD ISAs have generally moved beyond subword-SIMD (except in the embedded space) towards wider SIMD registers.

Microarchitectures of general-purpose cores perform instruction scheduling of SIMD instructions using similar mechanisms as they do for scalar instructions. Existing features like superscalar issue, out-of-order execution, and register renaming can be reused to maintain high instruction

throughput into the SIMD functional units. This paradigm has largely held even in modern general-purpose-SIMD cores, which feature much wider SIMD registers than scalar registers for improved performance on data-parallel workloads.

Practically all widely deployed commercial general-purpose cores ship with SIMD extensions. Intel and AMD out-of-order cores support some form of the SSE or AVX extensions [5], while ARM's A-profile [6] architecture requires NEON.

2.2.4 VLIW ISAs with SIMD

VLIW ISAs with SIMD extensions have traditionally dominated the space of DSP and similar domain-specialized cores, as well as embedded cores. For these deployment scenarios, power and area constraints preclude the deployment of costly microarchitectural features towards high instruction throughput in a general-purpose ISA, such as speculative execution and register renaming. The structure of DSP kernels presents a good fit for the VLIW paradigm, since regular loops, rather than branch-dense control code, are easier to optimize on a VLIW microarchitecture.

Furthermore, DSP applications often require more regularly behaved memory systems to achieve strict QoS requirements, leading to DSP cores often integrating low-or-fixed-latency software-managed scratchpad memories, rather than caches with dynamic, unpredictable behavior. Applications and microarchitectures which prefer statically predictable memory systems are especially well-suited for VLIW ISAs.

However, VLIW-based ISAs are notoriously difficult to program compared to general-purpose ISAs or vector ISAs. Performant VLIW code can also suffer from issues such as large static code size due to the need for extensive static scheduling and software-pipelined loops. Nonetheless, specialized VLIW ISAs provide a microarchitecturally simple and efficient programmer-visible mechanism for maintaining high instruction throughput into SIMD functional units.

Cadence, Synopsys, CEVA, and Qualcomm all ship commercial VLIW DSPs with SIMD extensions. Cadence, Synopsys, and CEVA cores are IP products typically integrated into a customer's SoC as an embedded core, while Qualcomm's Hexagon DSP Cores are integrated throughout their SoC line to provide DSP compute.

2.2.5 Scalable Vector ISAs

In contrast to the above patterns, modern **scalable vector ISAs** aspire to provide a common unified ISA that can support a range of microarchitectural implementation styles, supporting long-vector Cray-like machines, general-purpose out-of-order machines with vector extensions, specialized DSP cores with vector extensions, as well as ultra-compact embedded vector units. The dominant examples of such ISAs include ARM's proprietary SVE [7] and MVE [8] extensions, as well as the open RISC-V Vector extension [9].

Existing academic implementations of RVV have broadly targeted the HPC and general-purpose deployment scenarios with long-vector or out-of-order-core-integrated microarchitectures. Compared to most prior academic implementations, Saturn targets DSP and domain-specialized cores, and represents a class of designs we call *"short-vector"*.

Existing open-source "short-vector" implementations like Spatz [10], Vicuna [11], and RISCV-2 [12] require either register-renaming or constricted memory systems. RISCV2 requires register-renaming, while Spatz and Vicuna assume a low-latency memory system. Vicuna further requires a global stall to adapt to variable memory latencies. Unlike these designs, Saturn demonstrates that a "short-vector" design can achieve high performance and efficiency without any architectural or microarchitectural sacrifices.

2.3 The RISC-V Vector ISA

The RISC-V Vector ISA is the standard extension in RISC-V for exploiting data-level parallelism. A full discussion of the ISA design can be found in its specification [9]. This section highlights several properties of RVV that pose notable challenges to implementation or distinguish it from other vector ISAs.

2.3.1 Dynamic Vector Type and Length

Stripmine loops in RVV use `vset` instructions to dynamically adjust vector configuration state in the body of the loops. These instructions set the dynamic `vl` vector length register in addition to the `vtype` register, which sets the element width, register grouping, and mask/tail agnosticity for subsequent operations.

A naive implementation of RVV might treat the `vtype` as a single system-wide register, owing to its effect on the behavior of many components of the vector datapath. However, such an approach would substantially degrade performance, as `vset` is used in the inner loops of vector kernels to effect stripmining or to enable mixed-precision kernels.

As a result, performant implementations must maintain copies of the `vtype` and `vl` registers, instead of maintaining a single global status. Since neither `vtype` nor `vl` require many bits to encode, this state can be renamed into a control bundle that propagates alongside each vector instruction in the datapath.

Furthermore, since `vtype` and `vl` affect the generation of precise faults by vector memory instructions, it is insufficient to update these registers only at commit, since precise faults must be generated ahead of commit. Doing so would introduce an interlock between a `vset` and a subsequent vector memory operation, which must stall until the `vset` commits before using the updated `vtype`/`vl` to check for precise faults. Instead, performant scalar core implementations should bypass updates of `vtype` and `vset` to an early stage in the pipeline.

2.3.2 Memory Disambiguation

RVV mandates that vector memory operations appear to execute in instruction order with respect to **all** other instructions on the same hart, including scalar memory instructions. While an alternative ISA design may have relaxed this ordering requirement, such an approach would necessitate costly and precise programmer-inserted fences to enforce scalar-vector memory ordering.

This requirement for scalar-vector memory disambiguation poses a challenge to decoupled post-commit vector unit implementations, in which vector loads and stores might run behind scalar loads and stores. Stalling scalar loads and stores until the vector loads and stores drain could have costly implications on kernels which naturally would benefit from overlapping scalar and vector memory operations. For instance, in an input-stationary matrix multiplication, the inner loop streams across scalar loads of an input tile and vector loads and stores of the output. This kernel would naturally require efficient scalar-vector memory disambiguation.

Performant implementations should allow concurrent execution of scalar and vector memory operations by performing precise early-stage memory disambiguation of vector memory accesses.

2.3.3 Precise Faults

RVV mandates precise faults for vector memory operations. Faulting vector loads and stores must execute up until the element which causes the fault, report the element index that generated the fault, and block commit of any younger scalar or vector instructions. This implies that implementations must check for precise faults ahead of commit.

However, offloading address generation entirely ahead of-commit would have significant negative performance consequences, as this would stall unrelated scalar instructions even in the common case where instructions do not fault. Performant implementations should expediently commit vector memory instructions in the common case where they do not fault, and only interlock the scalar core in the uncommon case where a fault is present.

2.3.4 Vector Register Grouping

The `LMUL` (length multiplier) register grouping field of `vtype` enables grouping consecutive vector registers into a single longer vector register. In addition to enabling mixed-precision operations, this feature allows kernels that do not induce vector register pressure to access an effectively longer hardware vector length. Generally, performance programmers for RISC-V will use this feature to reduce the dynamic instruction count of their loops and potentially improve the utilization of hardware compute resources. For example, vector `memcpy` induces no register pressure and can trivially set a high `LMUL` to reduce dynamic instruction count. Since higher `LMUL` settings will unroll instructions in hardware, `LMUL` also reduces static code size byreducing the need for unrolling loops in in software.

Thus, implementations should not penalize code which uses high `LMUL` to reduce instruction fetch pressure. The general intuition around vector code should be to use the highest `LMUL` setting while avoiding register spills. Implementations should should strive to support this intuition.

One tempting approach to implementing register grouping behavior would be to crack `LMUL > 1` instructions early in the pipeline and implement the backend instruction scheduling around `LMUL = 1`. While this strategy is straightforward to implement as it simplifies the instruction scheduling, it may cause performance issues due to pressure on datapath scheduling resources from the many micro-ops generated by cracked high-`LMUL` instructions. Alternatively, the addition of queueing resources to reduce this pressure would add significant area and power overhead.

2.3.5 Segmented Memory Instructions

Segmented memory instructions enable a "transpose" of an "array-of-structs" data representation in memory into a "struct-of-arrays" in consecutive vector registers. Such instructions, while very complex behaviorally, are fundamental to many algorithms and datatypes. For instance, complex numbers and image pixel data are conventionally stored in memory as "arrays-of-structs".

The instructions are critical for repacking data in memory into element-wise format for vector instructions. Compared to other vector or SIMD ISAs, RVV provides few facilities for register-register repacking, instead relying on segmented memory instructions to perform "on-the-fly" repacking between memory and registers.

Given the importance of these instructions, performant RVV implementations should not impose an excess performance overhead from their execution. Vector codes which use these memory operations should perform no worse than the equivalent code which explicitly transforms the data over many vector instructions.

2.4 Short-Vector Execution

Saturn's instruction scheduling mechanism differentiates it from the relevant comparable archetypes for data-parallel microarchitectures. Fundamentally, Saturn relies on efficient dynamic scheduling of short-chime short-vectors, without relying on costly register renaming. When `LMUL` is short (1 or 2), vector chimes may be only 2-4 cycles long, requiring higher throughput scheduling than a long-chime machine.

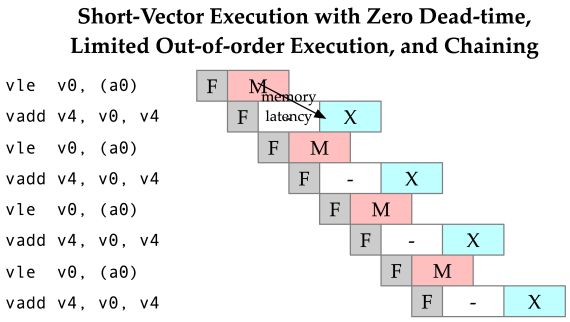

Figure 3: Pipeline diagram of instruction execution with short vector lengths, zero dead time, limited out-of-order execution, and chaining. Instructions in the X and M pipelines can execute out-of-order w.r.t. each other.

Figure 3 depicts a simplified pipeline visualization of a short vector loop, consisting of a load and dependent arithmetic instruction, executing on a simplified Saturn-like short-vector datapath. In this example, each vector chime is 2 cycles.

A short-vector machine should fully saturate both the arithmetic and memory pipelines with such short vector lengths and chimes. Instruction throughput requirements are moderate, but can still be fulfilled with a modest superscalar in-order scalar core. Notably, some degree of out-of-order execution, beyond just chaining, is necessary to enable saturating both the memory and arithmetic pipelines.

Figure 4 highlights the importance of zero dead time for short-vector microarchitectures. Unlike in Figure 3, the machine in this example has 1-cycle dead time for each functional unit, perhaps due to an inefficiency in freeing structural resources as instructions are sequenced. A single cycle of dead time in a short-chime machine substantially degrades the utilizations of the datapaths, as the dead time cannot be amortized.

Figure 5 highlights the importance of limited out-of-order execution for short-vector microarchitectures like Saturn. Unlike in Figure 3, the machine in this example requires the instructions to enter the datapath in-order. Requiring strict in-order execution would substantially degrade performance for suboptimally scheduled vector code. Despite the same zero-cycle dead time, the restriction on in-order execution prevents the machine from aggressively issuing instructions to hide

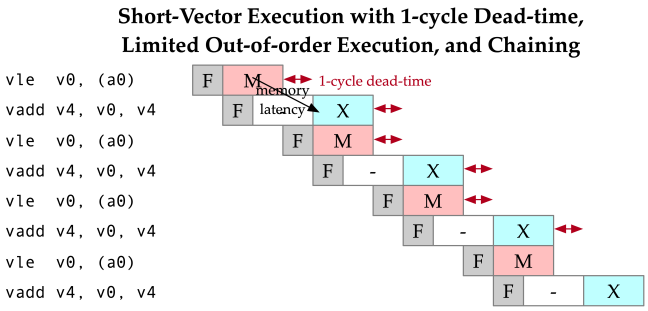

Figure 4: Pipeline diagram of instruction execution with short vector lengths, 1-cycle dead time, limited out-of-order execution, and chaining. Instructions in the X and M pipelines can execute out-of-order w.r.t. each other.

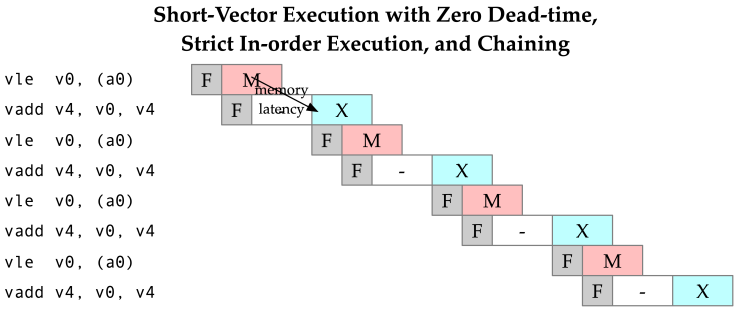

Figure 5: Pipeline diagram of instruction execution with short vector lengths, zero-cycle dead time, strict in-order execution, and chaining. Instructions in the X and M pipelines must begin execution in-order.

the latency in the M pipe.

2.4.1 Compared to Long-Vector Units

Long-vector microarchitectures feature very-long-vector-lengths distributed across many parallel vector lanes. Such implementations typically store their vector register files in dense lane-distributed SRAM banks.

Given the very long vector lengths, vector instructions are executed in a deeply temporal manner, even across many parallel vector lanes. Thus, instruction throughput is less critical for maintaining high utilization of functional units. Instead, long-vector microarchitectures typically derive efficiency and high utilization by amortizing overheads over fewer long-chime inflight instructions.

Figure 6 shows an example pipeline diagram of a vector loop in a deeply temporal long-vector machine. Unlike in the short-vector example in Figure 3, Instruction throughput requirements are minimal and strict in-order execution is sufficient for maintaining high utilization of the datapaths.

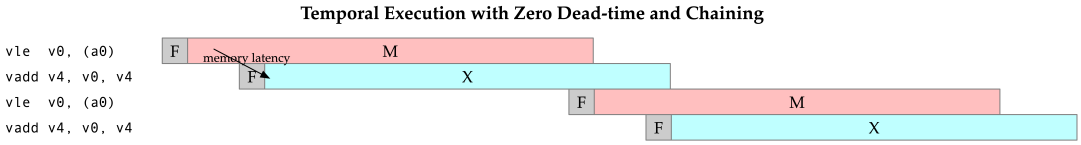

Figure 6: Pipeline diagram of instruction execution in a deeply-temporal long-vector machine with zero dead time.

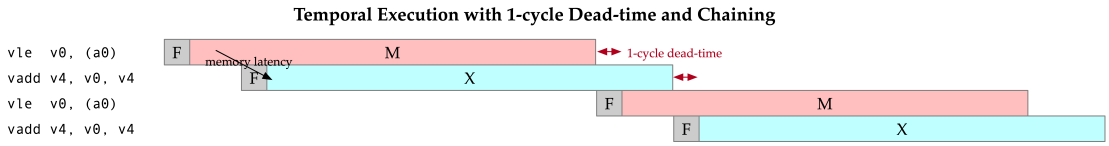

Figure 7: Pipeline diagram of instruction execution in a deeply-temporal long-vector machine with 1-cycle dead time.

Figure 7 highlights how dead time in a deeply-temporal vector-unit is amortized over many cycles of temporal execution per instruction. This contrasts with the Saturn-like short-vector machine, in which short chimes cannot hide dead time.

For DSP deployments, the long-vectors paradigm is particularly ill-suited when compared to short-vectors Saturn-like cores.

- Many DSP applications feature short and/or widely varying application vector lengths. This makes it difficult for long-vector machines to effectively utilize their resources as it precludes deep temporal execution. Short-vector machines can achieve higher utilization on these shorter application vector lengths.
- Short-vector machines use an inherently lower capacity vector register file, which has positive implications in terms of silicon area and power consumption.
- Short-vector machines can still reduce IPC requirements and dynamic instruction counts on applications with long vector lengths by leveraging the register grouping capabilities of modern vector ISAs.

2.4.2 Compared to General-purpose SIMD Cores

SIMD datapaths in general-purpose cores are typically deeply integrated into the scalar instruction execution pipeline. In these designs, existing capabilities for out-of-order execution, speculative execution, superscalar fetch, and register renaming are leveraged to maximize SIMD datapath utilization. While these features are costly in power and area, they are fundamental necessary

components of modern general-purpose cores, and thus are also leveraged when executing SIMD code.

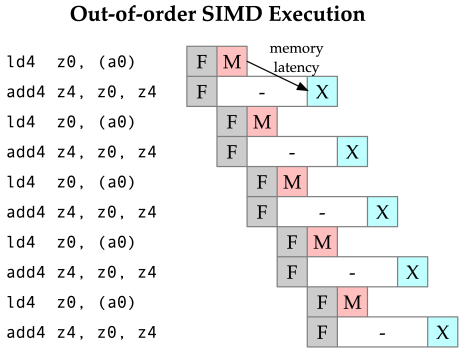

Figure 8: Pipeline diagram of instruction execution in an out-of-order SIMD machine.

Figure 8 depicts an example pipeline diagram of a SIMD loop in an out-of-order core with SIMD extensions. Notably, as these machines are typically designed with single-chime instruction execution, high instruction throughput is necessary to maintain high utilization of multiple datapaths. Furthermore, register renaming is required to enable execution past the WAW and WAR hazards in this example loop.

Unlike these cores, a Saturn-like short-vector design does not rely on costly features like high-throughput instruction fetch, out-of-order execution, speculative execution, or register-renaming. Efficient scheduling of short-chime vector instructions with a limited capability for out-of-order execution is sufficient for maintaining datapath utilization on memory workloads, even with a minimal in-order scalar core.

2.4.3 Compared to VLIW + SIMD DSP Cores

VLIW cores with SIMD extensions also rely on high-throughput instruction fetch for performance through VLIW instruction encodings. VLIW instructions enable low-cost, low-complexity superscalar fetch and provide the programmer with precise control over instruction scheduling.

Figure 9 depicts an example software-pipelined loop executing on a VLIW-SIMD machine. While VLIW instruction encodings effectively provide high operation throughput to the datapath, leveraging them typically requires precisely optimized static scheduling, such as with extensive loop unrolling and software-pipelining. Notably, the verbose prologue and epilogue sections of such loops inflate the code size beyond what a normal vector or scalar loop would require.

Unlike SIMD datapaths in VLIW cores, a Saturn-like short-vector design does not require the high instruction throughput of VLIW fetch. A short-vector machine's more aggressive capability for

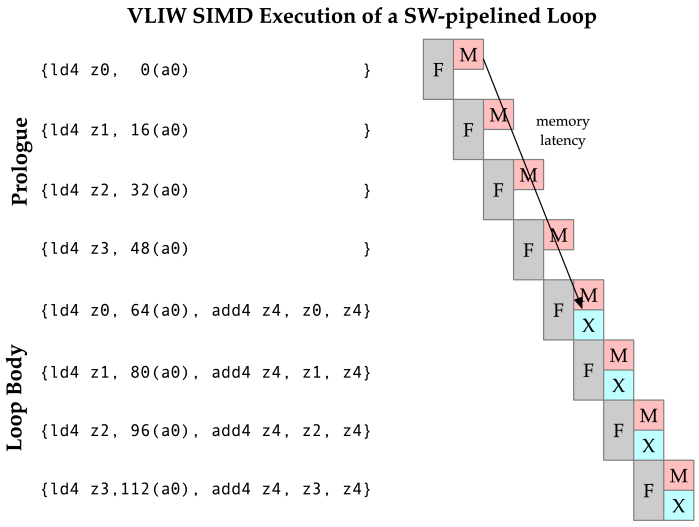

Figure 9: Pipeline diagram of instruction execution in a VLIW SIMD machine, executing a software-pipelined loop.

dynamic instruction scheduling also diminishes the need for precisely scheduled microarchitecture-aware code, compared to VLIW-SIMD designs.

3 System Organization

Saturn is implemented using the Chisel hardware description language [13], and is intended to integrate into existing Chisel-based cores as part of the open-source Chipyard [14] SoC framework. The generator is a parameterized Scala program that uses the Chisel embedded DSL to generate a synthesizable RTL circuit given a user-defined configuration.

3.1 Organization

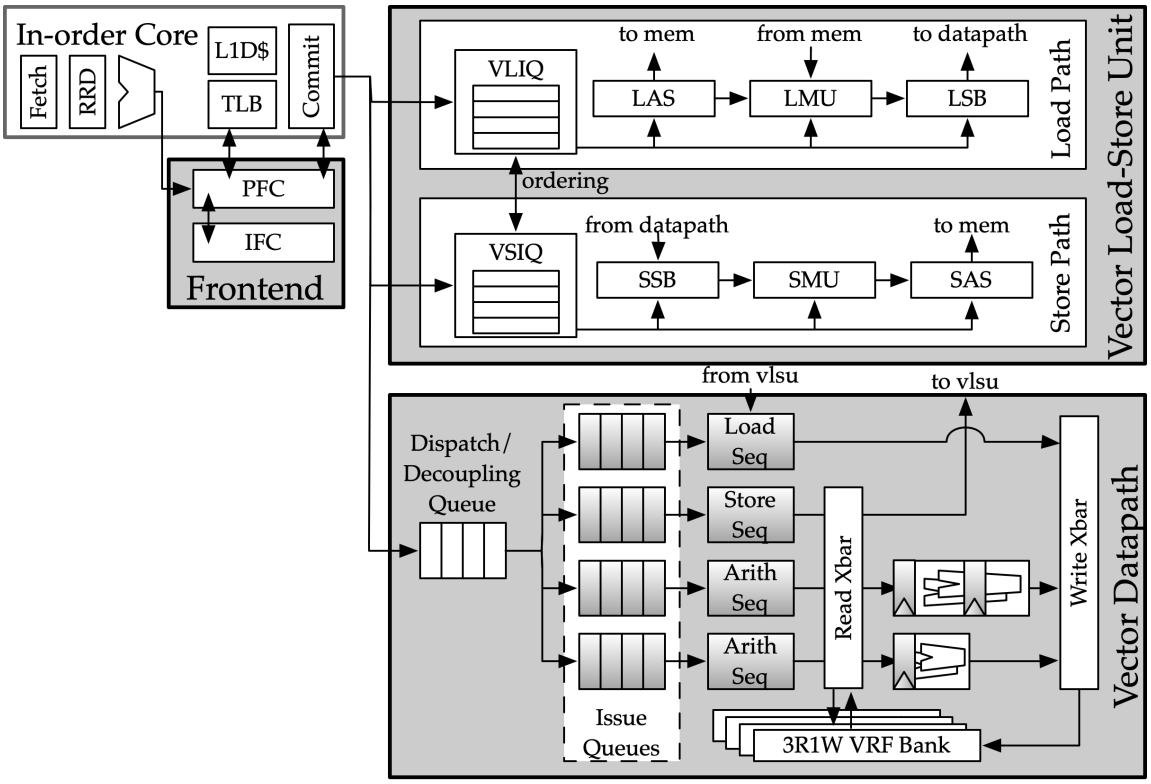

Figure 10: Overview of Saturn's Microarchitecture

Saturn is organized into three main components, as shown in Figure 10.

The **Vector Frontend (VFU)** integrates into the pipeline of the host RISC-V core. In in-order cores, the stages of the VFU align with the pre-commit pipeline stages of the host core. All vector instructions are passed into the vector frontend. The VFU performs early decode of vector instructions and checks them for the possibility of generating a fault by performing address translation of vector memory address operands. Instructions that do not fault are passed to the

other vector components once they have also passed the commit stage of the scalar core.

The **Vector Load-Store Unit (VLSU)** performs vector address generation and memory access. Inflight vector memory instructions are tracked in the vector load-instruction-queue (VLIQ) and store-instruction-queue (VSIQ). The load/store paths within the VLSU execute independently and communicate with the VU through load-response and store-data ports.

The **Vector Datapath (VU)** contains instruction issue queues (VIQs), vector sequencers (VXS/VLS/VSS), the vector register file (VRF), and the SIMD arithmetic functional units. The functional units (VFUs) are arranged in execution unit clusters (VEUs), where each VEU is fed by one sequencer. The sequencers schedule register read/write and issue operations into the VEUs, while interlocking on structural and data hazards. The VU is organized as a unified structure with a SIMD datapath, instead of distributing the VRF and VEUs across vector lanes. This approach is better suited for compact designs, where scalability to ultra-wide datapaths is less of a concern.

3.2 Key Principles

Several key principles lay the foundation for Saturn's microarchitecture.

Saturn relies on **post-commit execution** of all vector instructions. That is, the VLSU and VU only receive committed instructions from the VFU. Physical addresses are passed from the VFU into the VLSU to ensure that the VLSU will never fault on memory accesses. This simplifies the microarchitecture of the VLSU and VU, as all operations in these units are effectively non-speculative. To enforce precise faults ahead of commit, Saturn's VFU implements an aggressive, but precise mechanism that validates instructions are free-of-fault and reports any faults precisely if present.

Saturn's microarchitecture is designed around **in-order execution** with many latency-insensitive interfaces. Notably, the load and store paths are designed as pipelines of blocks with latency-insensitive decoupled interfaces. The load-response and store-data interfaces into the VU are also latency-insensitive decoupled interfaces. Within the VU, each execution path (load, store, or arithmetic) executes instructions in-order. The in-order execution of the load/store paths aligns with the in-order load-response and store-data ports.

Saturn is organized as a **decoupled access-execute (DAE)** [15] architecture, where the VLSU acts as the "access processor" and the VU acts as the "execute processor". Shallow instruction queues in the VU act as "decoupling" queues, enabling the VLSU's load-path to run many instructions ahead of the VU. Similarly, the VLSU's store path can run many cycles behind the VU through the decoupling enabled by the VSIQ. This approach can tolerate high memory latencies with minimal hardware cost.

Saturn still supports a limited, but sufficient capability for **out-of-order execution**. The load, store, and execute paths in the VU execute independently, dynamically stalling for structural and

data hazards without requiring full in-order execution. Allowing dynamic "slip" between these paths naturally implies out-of-order execution. To track data hazards, all vector instructions in the VU and VLSU are tagged with a "vector age tag (VAT)". The VATs are eagerly allocated and freed, and referenced in the machine wherever the relative age of two instructions is ambiguous.

Saturn is designed around two key parameters. `VLEN` **and** `DLEN`. `VLEN` is the vector length of each register file, as defined in the architecture specification. `DLEN` is a micro-architectural detail that describes the datapath width for each of the SIMD-stype datapaths in Saturn. Specifically, the load pipe, store pipe, and SIMD arithmetic pipes are all designed to fulfill `DLEN` bits per cycle, regardless of element width. Future versions of Saturn may allow a narrower memory interface width (`MLEN`) than `DLEN`.

To decode vector instructions, the Saturn generator implements a **decode-database**-driven methodology for vector decode. The Saturn generator tabularly describes a concise list of all vector control signals for all vector instructions. Within the generator of the VU, control signals are extracted from the pipeline stages using a generator-time query into the instruction listings. The results from this query are used to construct a smaller decode table for only the relevant signals, which is passed to a logic minimizer in Chisel that generates the actual decode circuit. This approach reduces the prevalence of hand-designed decode circuits in the VU and provides a centralized location in the generator where control signal settings can be referenced.

4 Frontend

Vector instructions are dispatched to the VFU at an early stage in the host core's pipeline. These instructions must be dispatched along with updated values of `vtype` and `vl`, to control the fault-checking mechanisms. The **pipelined fault checker (PFC)** and **iterative fault checker (IFC)** units perform memory translation for vector memory instructions, and either validate the absence of a memory access-related fault or generate a precise fault ahead of the commit point in the host core. Instructions validated to be free of faults are dispatched to the VLSU and VU.

4.1 Core Integration

Saturn currently supports integration into either Rocket or Shuttle as the host scalar RISC-V core. The host cores are responsible for decoding vector instructions that must be dispatched to the VFU, in addition to updating and supplying the `vtype` and `vl` CSRs to these instructions.

Rocket and Shuttle expose nearly identical interfaces to the vector unit. The scalar pipeline dispatches vector instructions to the VFU pipeline at the X stage, immediately after scalar operands are read from the scalar register files. The scalar pipeline also must provide these instructions with up-to-date values of `vtype`, `vl`, and `vstart`. The M stage exposes a port through which the VFU can access the host core's TLB and MMU. The WB stage contains interfaces for synchronizing committing or faulting vector instructions with the scalar core's instruction scheme and stalling scalar instruction commit if necessary. The WB stage also exposes interfaces for modifying `vstart` and `vconfig` as vector instructions commit.

4.1.1 Rocket

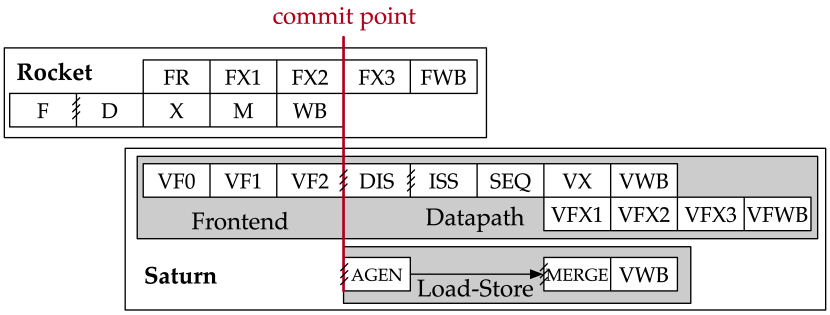

Figure 11: Rocket's pipeline stages with Saturn attached. Hatched lines indicate FIFO queues between pipeline stages.

Rocket is a 5-stage in-order single-issue core. As shown in Figure 11, the Saturn VFU integrates into the execute, memory, and write-back stages of Rocket, where the write-back stage is the commit

point. At the write-back stage, vector instructions which cannot retire due to ongoing activity in the IFC can kill all younger instructions in the earlier pipeline stages, and request the core to re-fetch and replay the instruction at the next PC. Vector instructions which cannot pass the PFC due to a TLB miss or a lack of capacity in the backend instruction queues are replayed through Rocket's existing replay mechanisms.

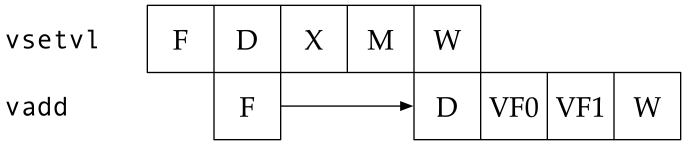

Figure 12: `vset`-induced bubble in Rocket.

Rocket does not maintain a speculative copy of the `vtype` and `vl` CSRs at the decode stage, so a data hazard can interlock the decode stage whenever a vector instruction proceeds a `vset` instruction. As shown in Figure 12, a `vset` will always induce a 2-cycle bubble on a proceeding vector instruction. The effect of this is most noticeable in short-chime mixed-precision vector code, in which `vset` instructions are frequent.

4.1.2 Shuttle

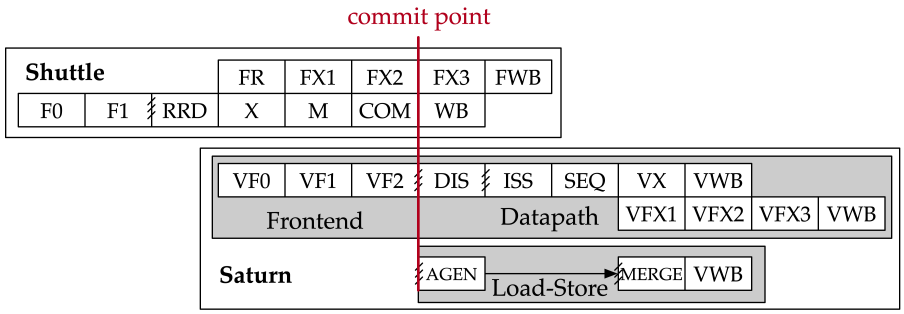

Figure 13: Shuttle's pipeline stages with Saturn attached. Hatched lines indicate FIFO queues between pipeline stages.

Shuttle is a 7-stage in-order superscalar core, typically configured as 2-issue or 3-issue. The Saturn VFU integrates into the execute, memory, and commit stages of Shuttle.

Only one of the execution pipes in Shuttle can dispatch into the VFU, but any of the pipes can execute a `vset` operation. However, during steady-state operation, Shuttle can dynamically construct instruction packets at the decode stage to maximize instruction throughput given structural hazards by stalling partial instruction packets.

Similar to Rocket, vector instructions that cannot retire at the commit stage will kill younger instructions in the pipeline, and request a refetch and replay of the subsequent instruction.

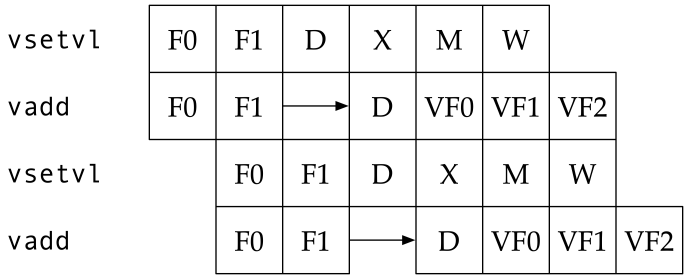

Figure 14: Shuttle dual-issue with forwarding of `vset`

Unlike Rocket, Shuttle implements a bypass network for `vset` instructions modifying `vtype` or `vl`. Vector instructions following a `vset` instruction do not need to stall, as the `vtype` and `vl` operands can be accessed through the bypass network. However, a vector instruction cannot proceed in the same instruction packet as a `vset`; it must proceed on the next cycle instead. Figure 14 shows how Shuttle can dynamically stall a partial instruction packet with the `vadd` to issue it with a younger `vset` on the next cycle. This example also depicts how stalling the `vadd` maintains 2 IPC through Shuttle, and 1 IPC into the vector unit.

4.2 Memory Translation and Faults

After entering the VFU, vector instructions first proceed through the pipelined fault checker (PFC). Instructions that the PFC cannot conservatively guarantee to be free of faults are issued to the IFC. Instructions that pass the PFC successfully can then be dispatched to the VU and VLSU after they pass the commit point.

Since vector instructions may be speculative ahead of the commit point, any vector instruction flushed by the scalar core is also flushed from the VFU. The PFC/IFC design pattern achieves the goal of making common case vector instructions fast, through the PFC, while preserving correct precise fault behavior for all vector instructions through the IFC.

The PFC and IFC share access to a single TLB port in the VFU. This TLB port would typically access the scalar core's TLB. Future modifications to Saturn could supply a dedicated vector TLB instead.

4.2.1 Pipelined Fault Checker (PFC)

The Pipelined Fault Checker is designed to handle common vector instructions without stalling the pipeline at 1 IPC. Vector instructions fall into one of the following categories:

- **Single-page** vector instructions include arithmetic and vector memory instructions for which the extent of the access can be bound to one physical page, at most. This includes unit-strided vector loads and stores that do not cross pages, as well as physically addressed accesses that access a large contiguous physical region. These are the most common vector instructions and need to proceed at high throughput through the VFU.
- **Multi-page** vector instructions are memory instructions for which the extent of the instruction's memory access can be easily determined, but the range crosses pages. These are somewhat common vector instructions, and must not incur a substantial penalty.
- **Iterative** vector instructions include masked, indexed, or strided memory instructions that might access arbitrarily many pages. These instructions would fundamentally be performance-bound by the single-ported TLB, so the VFU can process these instructions iteratively.

In stage-0 (VF0), the PFC establishes which category a vector instruction belongs to. Note that this does not require memory translation, and can be quickly determined from the instruction opcode, base address offset, and current settings of `vtype` and `vl`.

Single-page instructions execute down the PFC pipeline with no stalls. In stage-1 (VF1), the accessed page is checked through the TLB port. In stage-2 (VF2), misses in the TLB flush the PFC, forcing the VFU to request a replay of the vector instruction. This mirrors how the host in-order core handles scalar TLB misses through a replay mechanism.

If the VF2 TLB response indicates an access or page fault, retirement of the instruction is blocked, and the instruction is issued to the IFC to determine if it faults. This is done because masked vector memory elements that access invalid addresses do not generate faults. The IFC maintains the capability to access the vector register file for the mask operand, if present.

Page-crossing instructions incur multiple cycles of occupancy in the PFC. The VF1 stage computes the number of elements within the first page, then updates `vstart` and requests a replay from the scalar core at the same PC. The replayed instruction will repeatedly see a non-zero `vstart`, compute an updated base address at the next page, and request a replay if the remaining elements cross pages, until all the pages have been checked. In the VF2 stage, the PFC will set the `vstart` and `vl` signals for the vector instructions dispatched into the VU and VLSU to correctly set the partial execution of such instructions. This behavior cracks page-crossing contiguous loads and stores into single-page operations.

4.2.2 Iterative Fault Checker (IFC)

Iterative instructions cannot be conservatively bound by the PFC. Instead, these instructions perform a no-op through the PFC and are issued to the IFC. Unlike the PFC, which operates page-by-page, the IFC executes element-by-element, requesting index and mask values from the VU for

indexed and masked vector operations. The IFC generates a unique address for each element in the vector access, checks the TLB, and dispatches the element operation for that instruction to the VU and VLSU only if no fault is found. Upon a fault, the precise element index of the access that generates the fault is known, and all accesses preceding the faulting element would have been dispatched to the VU and VLSU.

The IFC accesses the TLB through the same port as the PFC, with arbitration set to prioritize the IFC, as the IFC will always track older instructions than the PFC. The IFC also can access the VRF through the VU to fetch index or mask data.

4.2.3 Towards Performant Indexed or Strided Accesses

Lane-organized long-vector machines often implement high-radix memory systems with many independent memory access ports, enabling high-throughput address generation and good performance on indexed or strided accesses. In contrast, DSP cores, whether vector or SIMD, typically are integrated into a more conventional memory system designed around contiguous block accesses. Saturn's VLSU is designed towards deployment as a DSP system, and thus fundamentally has limited performance on indexed or strided accesses, as it can only generate one element's address per cycle for these instructions.

Thus, the design of the IFC, which can only performe one-element-per-cycle translation, matches the throughput of the VLSU. Improving the throughput of the IFC does yield some benefits even if the VLSU's address generation is fundamentally restricted. Younger vector instructions after an indexed or strided access can be aggressively issued to the backend, allowing the backend to overlap their execution with other elementwise loads or stores.

To reduce the cases that fall back to the IFC, the PFC can be extended to consider memory region granularities that are larger than a page. For instance, superpages in virtual memory can exceed the maximum possible extent of an constrained-index-width indexed access or a bounded-stride strided access. Similarly, when memory translation is disabled, the extent of contiguous physical memory serves as a large bound as well.

4.3 Scalar-Vector Memory Disambiguation

Vector memory instructions architecturally appear to execute sequentially with the scalar loads and stores generated by the same hart. Scalar stores cannot execute while there is a pending older vector load or store to that same address. Scalar loads cannot execute while there is a pending older vector load to that same address, as doing so could violate the same-address load-load ordering axiom in RVWMO since the vector and scalar loads access different paths into the memory system. Section 5.3 discusses the mechanisms for vector-vector memory disambiguation.

The S2 stage of the PFC also receives the physical address of the current in-flight scalar load or

store about to commit in the host scalar core's W stage. This address is checked against the older inflight loads and stores in the VLIQ and VSIQ in the VLSU. On a match, a replay for the younger scalar load or store is requested.

The VLSU does not access or probe the scalar store buffers. To avoid RAW or WAW memory hazards against scalar stores in a scalar store buffer, the PFC stalls the dispatch of vector instructions in the S2 stage until the scalar store buffer is empty. We observe that this requirement has minimal impact on most vector codes, as scalar stores are rare in stripmined loops.

4.4 Interface to VU and VLSU

The micro-op presented to the VU and VLSU contains the instruction bits, scalar operands, and current `vtype`/`vstart`/`vl` settings for this instruction. For memory operations, this bundle also provides the physical page index of the accessed page for this instruction, since the PFC and IFC crack vector memory instructions into single-page accesses. For segmented instructions where a segment crosses a page, `segstart` and `segend` bits are additionally included in the bundle, to indicate which slice of a segment resides in the current page.

5 Vector Load-store Unit

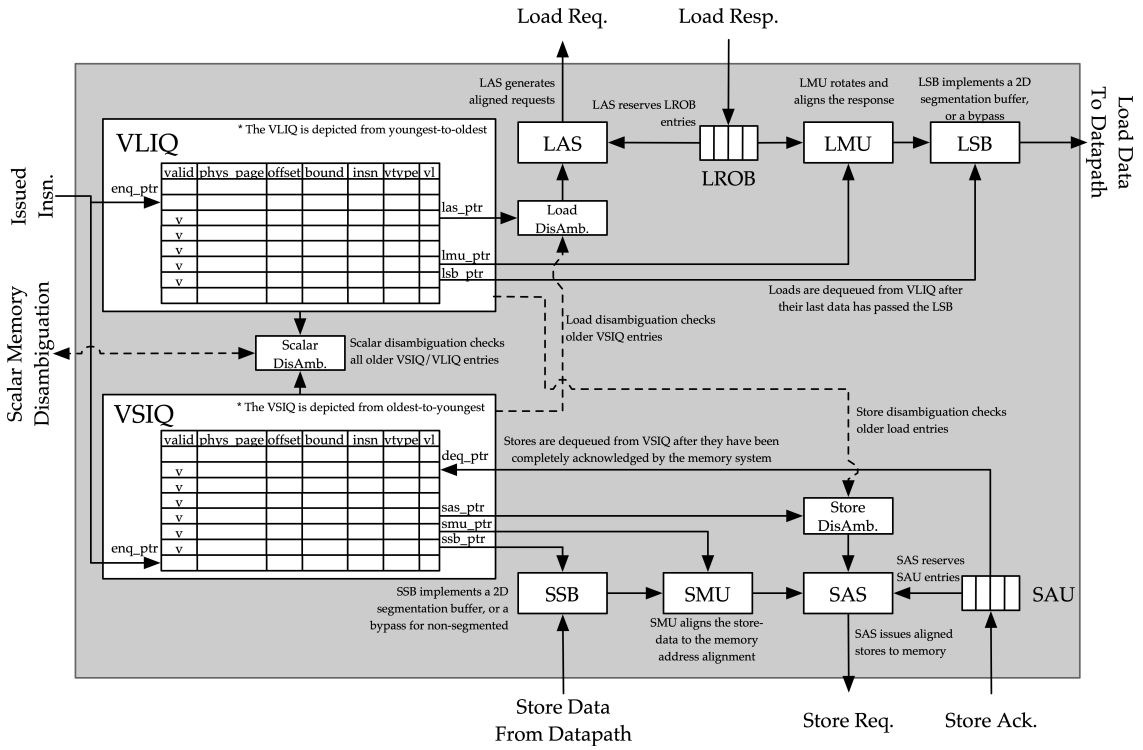

Figure 15: Vector Load-Store Unit Diagram

The Vector Load-Store Unit (VLSU) contains independent paths for loads and stores and is decoupled from the vector datapath (VU). The load path performs load address generation and reformats load responses into load writebacks. The store path performs store address generation and reformats vector register data into store requests. Each path is designed to saturate a block memory interface that is `DLEN` bits wide.

The load and store paths are each designed as an in-order pipeline of units with latency-insensitive interfaces. The control signals for each unit are driven by pointers into the load/store instruction queues (VLIQ/VSIQ). Since independent pointers for each unit are maintained, each unit can operate on the same, or different instructions within the queues. After an instruction has finished execution through the load or store path, its entry in the instruction queue is freed.

The load path consists of the following stages:

- Load Address Sequencer (LAS)
- Load Reordering Buffer (LROB)

- Load Response Merger (LMU)
- Load Segment Buffer (LSB)

The store path consists of the following stages:

- Store Segment Buffer (SSB)
- Store Data Merger (SMU)
- Store Address Sequencer (SAS)
- Store Acknowledgement Unit (SAU)

The store path is very similar to the reverse of the load path. Notably, the SAS is identical to the LAS, while the SMU is identical to the LMU.

The VLSU additionally contains a separate address sequencer for accessing a specialized memory region for high throughput scatter and gather - the Scatter-Gather Address Sequencer (SGAS). Indexed loads and stores that access a physical region managed by the scatter-gather memory use the SGAS for address sequencing, instead of the LAS or SAS, respectively.

5.1 Memory System

Saturn is designed for integration into a memory system that maintains coherence between VLSU accesses and scalar accesses into a scalar L1. The VLSU exposes a generic load/store decoupled request and response interface. This is typically converted into a coherent TileLink interface within either Rocket or Shuttle, but could be extended in the future to expose an incoherent AXI port instead.

One approach for the VLSU would be to direct all vector memory accesses into the scalar L1. While simple, such a design would induce frequent structural hazards and require a specialized host core with a specialized L1 data cache.

While the Saturn integration with Rocket does support this approach, the standard and preferred mechanism is to provide a vector-specific memory port that bypasses the L1 and accesses coherent backing memory. Figure 16 depicts such a memory system.

Saturn configurations with high `DLEN` would generally require higher memory bandwidth. However, scaling up the system-level interconnect to meet Saturn's bandwidth demands may be prohibitively costly. Instead, the preferred approach for high-`DLEN` Saturn configs is to integrate a high-bandwidth local TCM (tightly-coupled memory), which software should treat as a software-managed cache for vector accesses. This TCM should be tile-local and globally addressable, but not necessarily cacheable. Figure 17 depicts a Saturn configuration with a high-bandwidth TCM, but a reduced-bandwidth system interconnect.

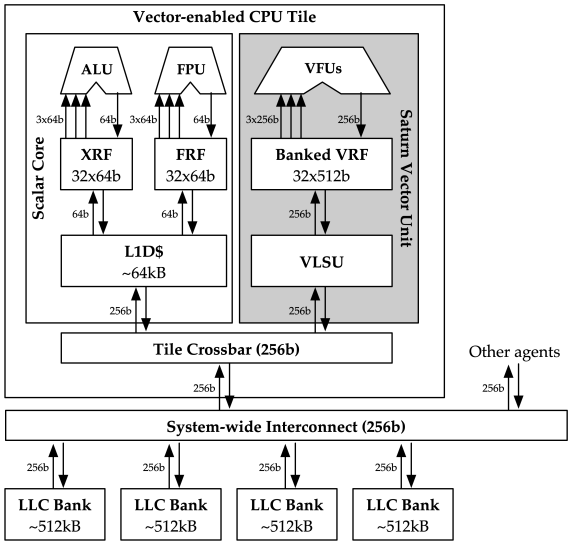

Figure 16: Example default configuration of the Saturn memory system

Saturn's integration with Shuttle supports these tile-local TCMs.

Saturn also supports integration into a system with a specialized "scatter-gather memory" (SGTCM). Unlike the standard memory interface, which supports one address per cycle for loads and one address per cycle for stores, the SGTCM interface presents an array of parallel byte-wide ports. The SGTCM is intended to be implemented as a specialized non-cacheable core-local memory.

Figure 18 depicts how the Saturn VLSU can be parameterized to bypass the block memory port to access a specialized address-generation engine for a deeply-banked scatter-gather memory. Saturn's integration with Shuttle supports a byte-wise banked SGTCM.

5.2 Inflight Instruction Queues

Upon dispatch from the VFU into the VLSU, a vector memory instruction is written into either the load instruction queue (VLIQ) or store instruction queue (VSIQ).

Each entry in this queue contains the base offset, physical page index, and stride, as well as a bit-mask of older vector loads or stores. The VFU guarantees that all instructions dispatched into the VLSU have been cracked into operations with single-page access extents. As a result, the base offset and stride are stored as 12 bits of page offset, while a physical page index provides the upper bits of the address. Each entry additionally contains the `vstart`, `vl`, `segstart`, and `segend` settings of this instruction, along with all the fields for addressing mode, element width, index width, and mask control.

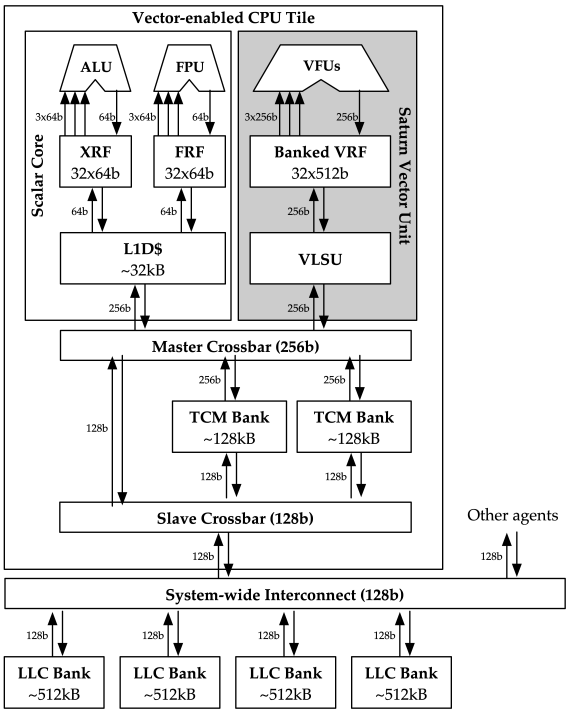

Figure 17: Example Saturn memory system with high-bandwidth local TCM (Tightly-coupled memory)

The entry also contains a bound (extent) for the instruction's memory access within its accessed page. This is derived from the base offset, stride, `vl`, and addressing mode settings of the instruction, but is encoded directly within the entry to enable fast disambiguation checks. Instructions with indexed accesses are marked conservatively as accessing the entire page.

Memory disambiguation checks are performed using a CAM-like structure over all the entries in the VLIQ or VSIQ to find the entries accessing the same page. The base and extent of a given access can be checked against the base and extent of the access in the entry to determine if overlap exists. Both vector-vector and vector-scalar ordering checks use this CAM.

The address sequencer and segment buffer units derive their control signals from pointers into the inflight instruction queues. For long-chime instructions (when `LMUL` is high), these pointers can reference the same instruction, enabling a single instruction to occupy all the units in the load or store path. For short-chime instructions (when `LMUL` is low), these pointers can refer to different instructions, enabling many simultaneous inflight instructions for short-chime code in a high-latency memory system.

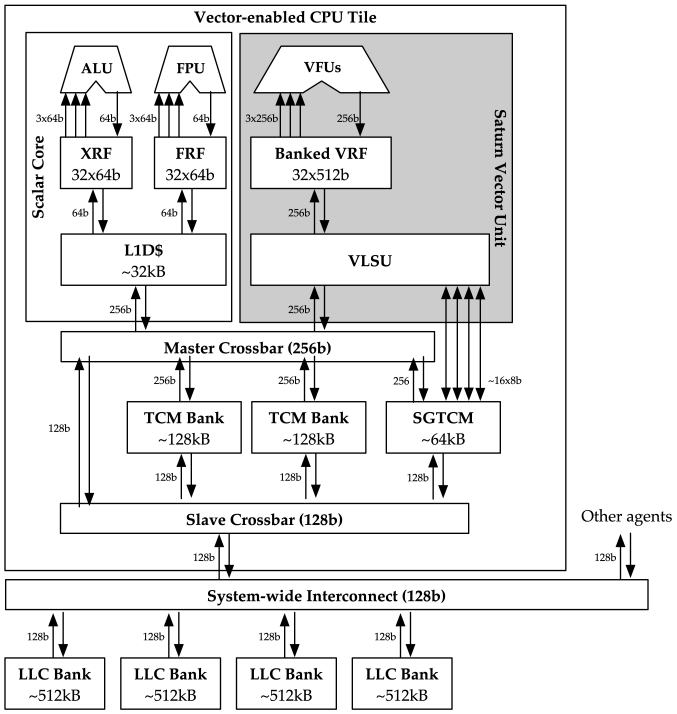

Figure 18: Example Saturn memory system with high-bandwidth local TCM and scatter-gather TCM (SGTCM)

5.3 Vector-Vector Memory Disambiguation

Saturn is responsible for stalling vector or scalar requests if an older vector or scalar request has not been made visible to the coherent memory system, and would cause a violation of the memory model if the younger request was allowed to proceed.

- A younger scalar load must stall until all older vector stores to the same address have been issued and acknowledged.
- A younger scalar store must stall until all older vector loads to the same address have been completed and all older vector stores to the same address have been issued and acknowledged
- A vector load or store cannot begin execution while there are pending older scalar stores in the scalar store buffer
- A younger vector load cannot issue requests while there are pending older vector stores to the same address

- A younger vector store cannot issue requests while there are pending older vector loads to the same address

Section 4.3 discusses how scalar-vector memory disambiguation is managed, while this section discusses the vector-vector cases. Vector memory disambiguation is performed when instructions enter either the LAS or SAS. An instruction cannot sequence address generation operations through either address sequencer until it has cross-checked the appropriate sequencer for older instructions with overlapping access extents.

The cross-check compares the to-be-address-sequenced instruction's extent with the extent of older instructions in the relevant inflight queue with a CAM-like structure. Since the VLIQ and VSIQ do not need to be deep, as they hold complete vector instructions, this CAM structure has tolerable cost.

Address disambiguation based on the precise extents of vector accesses, as opposed to a coarse cache-line or page-based granularity, requires a more costly CAM-like circuit. While a static-granularity address CAM can just compare the upper bits of the physical address, determine overlap by precise extents require comparators on the base and bounds of each access, where the width of each comparator is the page offset in Saturn's case. This costly circuit is necessary to avoid stalls on false-positive conflicts between consecutive contiguous vector loads and stores.

5.4 Memory Response Ordering

The VLSU does not assume that the vector memory interface will respond to requests in order. This further necessitates the implementation of a load-reordering buffer (LROB). Saturn supports a LROB with as many buffer entries as possible inflight requests. Saturn additionally supports implementing the LROB with fewer buffer entries than possible inflight requests, for use in scenarios where the outer memory system generally preserves response order, but is not guaranteed to. In this configuration, the LROB will replay loads when the LROB's buffers overflow, preserving an in-order response stream into the LMU.

The store acknowledgment unit similarly tracks the outstanding store acknowledgments. Once the final acknowledgment for a given store has arrived, that store can be freed from the VSIQ.

If integrated into a memory system that preserves load response ordering, the LROB can be omitted. Future improvements to Saturn can add this optimization.

5.5 Address Sequencing

The Address Sequencers (LAS/SAS) generate requests for all memory instructions except for indexed accesses into the SGM. The address sequencers emit requests aligned to the width of the memory interface. The sequencer can proceed with an instruction once cleared by address disambiguation.

The address sequencers effectively iterate over two nested loops. The outer loop iterates over the element index while the inner loop iterates over a "segment index" within a segment for segmented accesses. An index port and mask port provide a stream of indices/masks generated by the VU for indexed and masked operations.

Unit-strided (segmented and non-segmented) accesses do not execute the inner loop, and iterate the outer loop by the number of elements requested by the next request. These requests saturate the available memory bandwidth. Masked unit-strided loads ignore the mask settings, instead applying the mask when performing a masked write into the VRF in the VU. Masked unit-strided stores receive a byte mask from the SMU along with the data bytes, and use this to construct the store request.

Strided and indexed non-segmented accesses do not execute the inner loop, and iterate the outer loop by a single element per cycle. A mask is generated to select the active bytes within the access for the requested element. These accesses use the mask port if set by the instruction, and omit generating the request if the element is masked off.

Strided and indexed segmented accesses execute both the outer and inner loop. The inner loop iterates by the number of elements within a segment available within the next segment, while the outer loop iterates by segment index. These access the mask port if set by the instruction, and omit generating the request if the segment is masked off. Generally, these can saturate the memory bandwidth when the size of one segment is larger than the memory width.

The sequencers will stall if the memory interface is not ready or if there are no more tags to track outstanding memory accesses. When the last request for an instruction has been sent to the memory system, the pointer into the VLIQ/VSIQ for the instruction is incremented, and the next instruction to undergo address sequencing can proceed.

5.6 Merge Units

The merge units (LMU/SMU) are general-purpose circuits that correct for misalignment of the memory system response data before the next step in the load or store paths. These can be considered a generalized form of a rotation buffer, decoupling the alignment and extent of input data from the alignment and extent of output data, and preserving latency-insensitive decoupled interfaces. Microarchitecturally, the merge unit includes two additional byte-shifters compared to a standard rotation buffer. One additional shifter enables partial dequeues of buffered data, while the other allows the buffer to "compact" many partial packets into a full response packet.

The merge units have FIFO semantics, where the enqueue into the unit specifies a base and extent of active data within the wide input vector. The merge unit rotates away the inactive bytes, compacting the active bytes into contiguous storage. The dequeue shifts the buffered data into position based on the requested base and extent. A bypass path from the enqueue to the dequeue enables full-throughput continuous dataflow for misaligned contiguous accesses.

For the LMU, the push base and extent (head and tail) are set by the address offset associated with the original memory request. For block-contiguous accesses, only the first and last beat of a single access would encode a non-aligned head or tail, respectively. For element-indexed or strided accesses where each memory request contains only a single valid element, the push head and tail denote the start and end byte of the active element. In this way, the LMU serves two roles, either rotating block-contiguous accesses or compressing indexed on strided accesses. This behavior decouples load-response alignment from writeback; regardless of addressing mode or alignment, the LMU generates aligned `DLEN`-wide contiguous bytes for writeback in the VU.

For segmented loads, the LMU serves an additional purpose; it enables decoupling of the load write-back scheduling performed by the datapath from the segment restructuring performed by the segment buffer. That is, the segment buffer does not necessarily proceed at `DLEN` bits per cycle for all segment sizes. Depending on the segment size, the segment buffer may request a sub-`DLEN` slice of bytes, which the LMU will provide once available.

The SMU operates as the reversed path of the LMU. The push head and tail of the SMU are usually aligned, except for the last element group when `vl` is misaligned. For segmented stores, the push head and tail may be set by the store segment buffer, instead of the store datapath. The pop head and tail are driven by the address alignment generated by the SAS. Notably, the SMU additionally tracks a byte-wise mask bit for masked stores, such that the mask can be applied to the generated store request.

5.7 Segment Buffers

For segmented accesses to proceed with high throughput, the LSB and SSB must "buffer" a sufficient number of responses to "transpose" a set of segments into a set of vector writebacks, or a set of

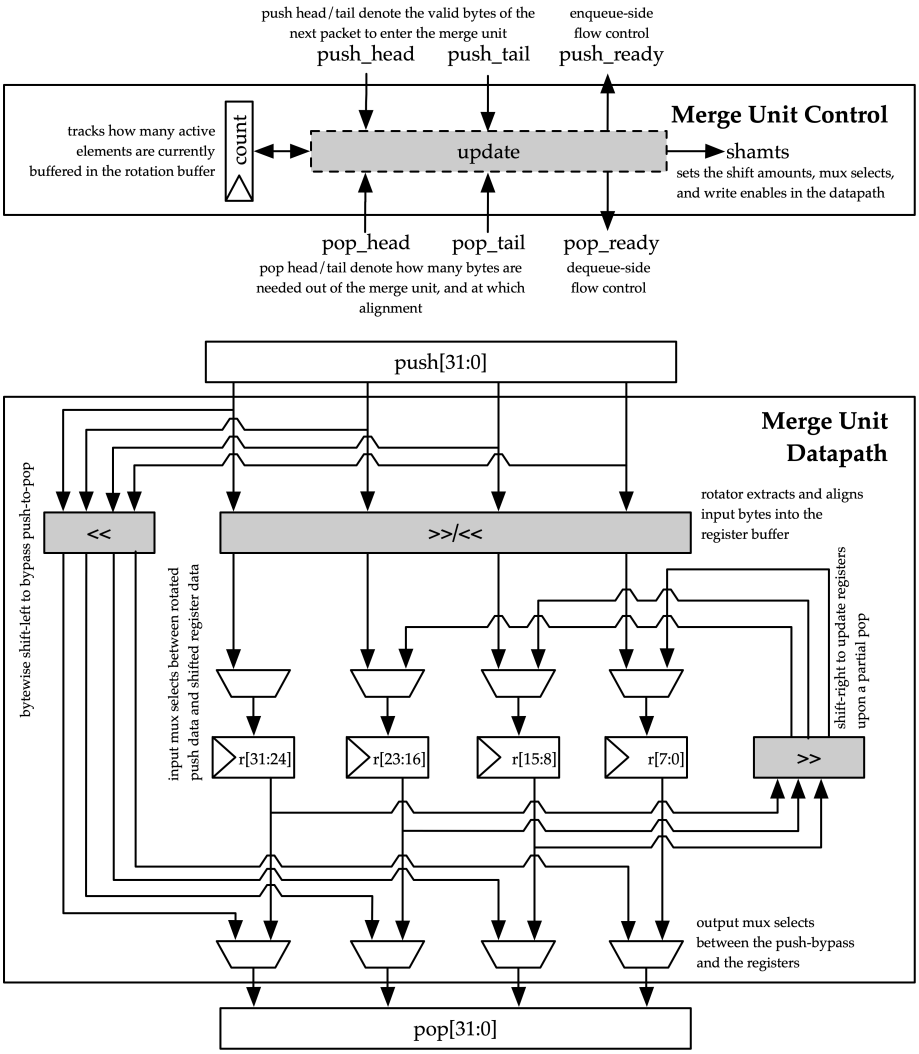

Figure 19: Control and datapath of the merge units. The merge unit can be considered a generalized rotation buffer, where the enqueue and dequeue sides are each latency-insensitive interfaces requesting an update (either a push or pop) of some segment of contiguous valid data into or out of the merge buffer.

vector store-data into a set of segments. Non-segmented accesses bypass the segment buffer units entirely.

Each segment buffer is implemented as a double-buffered 2D array of flops. The double-buffering enables full-rate segmented accesses. For instance, in the LSB, one half is filled by load responses while the other is drained by load writeback.

Each segment buffer is 8 rows deep to support up to 8 fields in a segment, as required by the specification. Each segment buffer is `DLEN` bits wide to buffer entire element groups of writeback data.

Load responses from the LMU write columns into the LSB, while the LSB emits rows into the load writeback port to the VU. Store data from the VU writes rows into the SSB, while the SSB emits columns into the SMU.

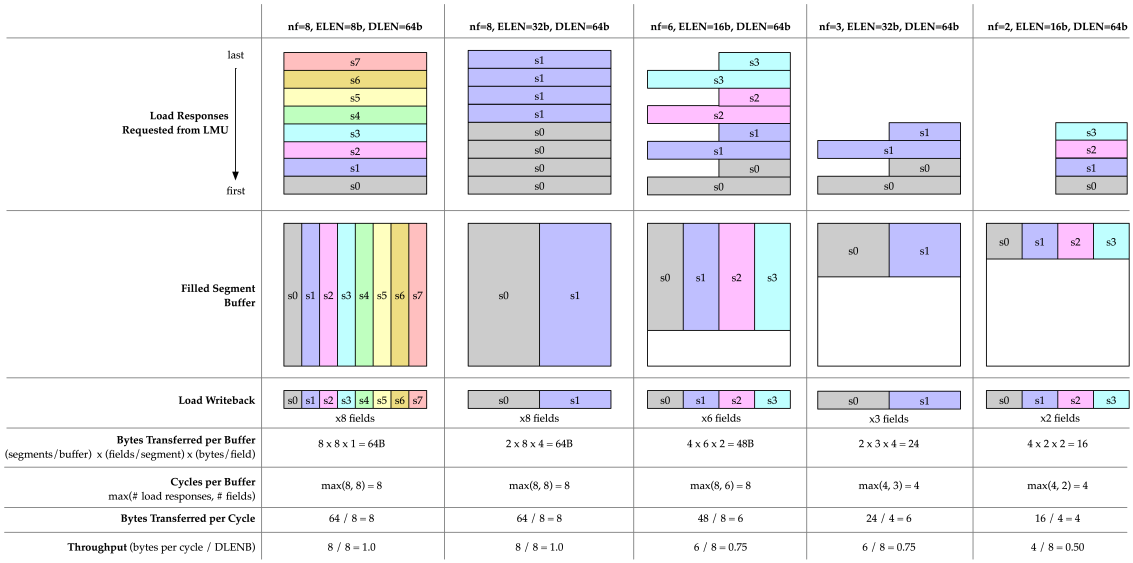

Figure 20: Table depicting behavior, storage layout, and throughput of the double-buffered LSB for varying NF/ELEN on a DLEN=64b configuration.

Figure 20 depicts how the LSB requests aligned segments from the LMU, stores them in a 2D segment buffer array, and presents a stream of aligned write-back data to the datapath. Notably, some configurations of `NF` and `ELEN` result in sub-optimal throughput, underutilizing the memory system. However, segmented loads and stores will always be more performant than the equivalent sequence of non-segmented loads and stores used in their place.

Some optimizations to improve the throughput of the power-of-two `NF` instructions have yet-to-be implemented.

6 Vector Backend

The vector datapath (VU) handles all vector instruction scheduling, computation, and register file access. The VU is organized around independent "sequencing paths", which issue instructions independently to a per-sequencer vector execution unit (VXU).

The datapath is organized around a "vector element group" as the base unit of register access and compute. An element group is a fixed-width `DLEN`-bits-wide contiguous slice of the architectural register file space. `DLEN` guides the parameterization of most critical components in the backend. The register file, load-path, store-path, and most functional units are designed to process `DLEN` bits per cycle (or 1 element group/cycle), regardless of element width (`ELEN`).

6.1 Issue Queues

Vector instructions entering the VU are immediately assigned a unique age tag and enqueued into an in-order dispatch queue (VDQ). The age tag is used to disambiguate age when determining data hazards in the out-of-order sequencing stage. This means the age tag is wide enough to assign a unique tag to every inflight operation in the backend. Age tags are recovered when instructions complete execution.

The VDQ provides a low-cost decoupling queue between the VU and the VLSU. The dequeue side of the VDQ is connected to the vector issue queues (VIQs). Each VIQ issues instructions in-order into its corresponding instruction sequencer. Execution across the VIQs may be out-of-order, as the VIQs may naturally "slip" against each other.

Each sequencing path has an independent issue queue. The load issue queue only needs to track a destination address and mask enable, while the store issue queue only needs to track a source address and mask enable. The arithmetic issue queue(s) must track up to three source operands along with the destination and mask enable. The special issue queue, used for index generation, register gathers, slides, compress, and reductions, tracks only a source operand.

There are six currently supported configuration options for the issue queue and sequencer topology.

- The **unified** configuration features a single sequencer for all arithmetic operations fed by a VIQ. This configuration has the lowest hardware cost, but cannot simultaneously saturate integer and floating-point functional units.
- The **shared** configuration features separate integer and floating-point sequencers fed by a common VIQ. Instructions are issued from the shared VIQ to either the integer or floating-point sequencer. This configuration can simultaneously saturate integer and floating-point execution units with well-scheduled code, requiring the interleaving of integer and floating-point operations.

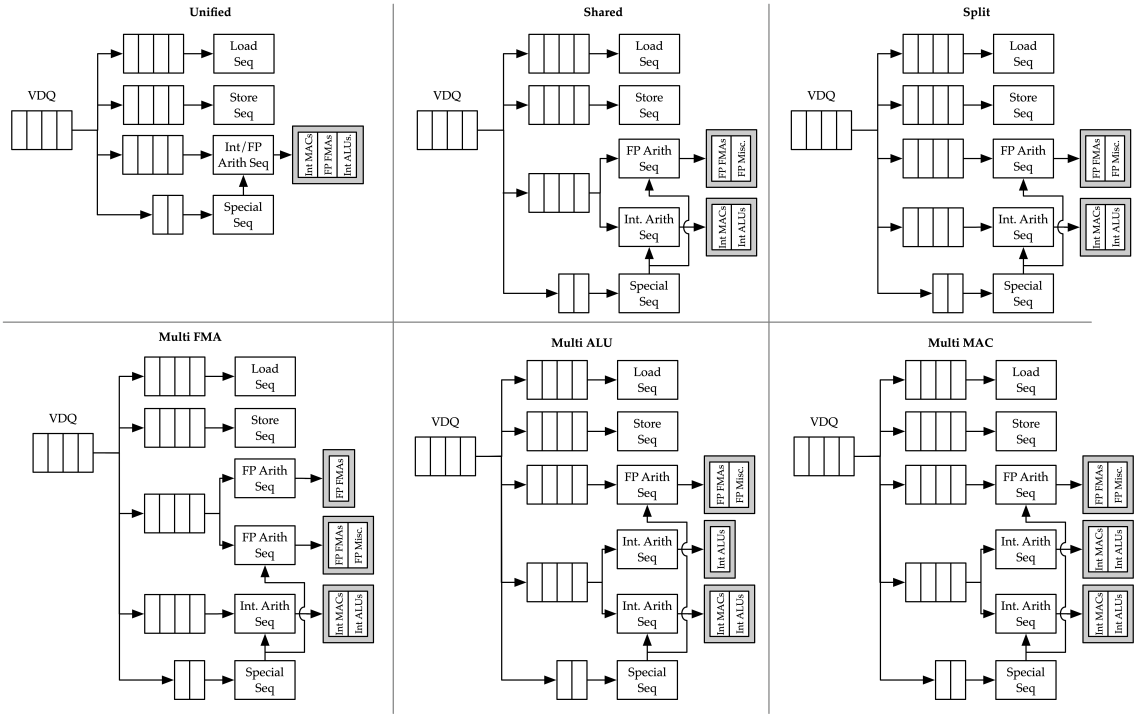

Figure 21: Issue Queue Configurations

- The **split** configuration features separate integer and floating-point sequencers each fed by an independent issue queue. This configuration is the most performant and the most flexible, as it can dynamically schedule out-of-order across integer and floating-point instructions. It incurs additional area costs owing to the separate floating-point issue queue.
- The **multi-ALU** configuration adds an additional integer sequencer and functional units to enable dual-issue of integer ALU operations
- The **multi-MAC** and **multi-FMA** configurations add additional integer or floating point sequencers along with functional units to enable dual-issue of integer or floating-point multiply-accumulate operations, respectively. These configurations increase the ratio of arithmetic throughput to memory bandwidthm, increasing the maximum performance of the machine for kernels with sufficient reuse.

6.2 Operation Sequencers

The instruction sequencers convert a vector instruction into a sequence of operations that execute down the functional unit datapaths, one operation per cycle. The sequencers advertise the requested

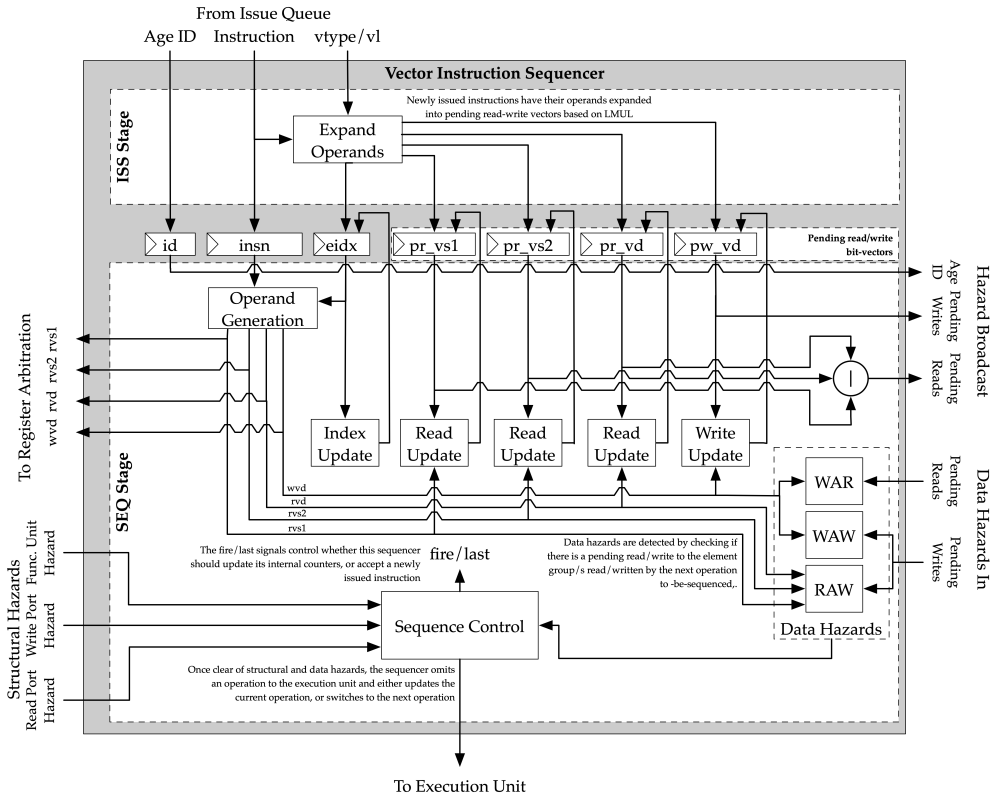

Figure 22: Vector Instruction Sequencer

register file read and write addresses for the next operation, as well as the age tag for the currently sequenced instruction. If there are no structural hazards from non-pipelined functional units or register file ports and there are no data hazards against older vector instructions, a sequencer will issue an operation and update its internal state. An instruction will depart a sequencer along with the last operation it sequences, eliminating dead time between successive vector instructions.

Notably, the sequencers enact "fire-and-forget" operation issue. Once an operation is issued by a sequencer, it is guaranteed to be free of further structural or data hazards as it proceeds down the pipelined functional unit datapaths. This eliminates the need for costly operand or result queues and obviates back-pressure in the functional unit pipelines.

6.2.1 Load/Store Sequencers

The load-store sequencers (VLS/VSS) are the simplest, as they only track one vector operand or destination.

The VLS sequences load writebacks into the VRF. The VLS will stall if the decoupled load response port from the VLSU has not presented a requested element group of write-back data. Since the VLSU's load path and the load-issue path are both in-order, issued operations from the VLS pop the next element group from the decoupled load-response port and write into the VRF.

The VSS behaves similarly to the VLS, except it sequences element groups of data into the decoupled store-data port. The store data-port can de-assert ready to stall the store sequencer.

Both the VLS and VSS handle the additional complexity segmented operations, which write a set of consecutive vector registers. To align with the data order expected in the segment buffers in the VLSU, the sequencers execute two nested loops to handle these instructions. The outer loop iterates over element group index, just as in normal vector instructions, while the inner loop iterates over the number of fields in the segmented instruction.

6.2.2 Execute Sequencer

The execute sequencers (VXSs) sequence all arithmetic operations. They track up to three register operands, with up to four reads and one write per operation (for a masked FMA). Each VXS issues to a single vector execution unit (VXU). A VXU is a collection of vector functional units, which may or may not be pipelined. The VXSs will stall operation execution if the requested functional unit within its VXU is unavailable.

6.2.3 Special Sequencer

The special sequencer (VPS) handles four classes of special-case instructions which enqueue a special sequencing operation into this sequencer, while the main instruction control is still consumed by either the VLS, VSS, or VXS. Using the VPS as an independent sequencer divides the read operand hazard tracking from the destination operand hazard tracking, enabling chaining naturally for both source and destination operands even if they are consumed at different rates.

- For **indexed memory instructions**, the VLSU's address-sequencing unit needs to process indices fetched from the register file. These instructions enter the VPS in addition to the VLS or VSS. The out-of-order sequencing support enables the VPS to run ahead of the VLS or VSS, feeding indices to the VLSU while older loads or stores might still be inflight.
- For **slides**, the VPS sequences reads for the source operand. DLEN-wide read-data is fed to a rotation circuit and buffer to align them for writeback, which is sequenced by a VXS. This enables slides to proceed at DLEN bits/cycle.
- For **compress** and **register gather**, the VPS sequences element-wise reads for the source operand. Elements enter an element buffer, which is accessed by the VXS when it sequences element-wise writebacks.

- For **reductions**, the VPS maintains a DLEN-wide accumulation buffer. The VPS performs the initial read of element 0 to populate the accumulation buffer. Once available, the VPS provides the accumulation buffer for the VXSs to access, and stalls the VXS for long-latency accumulates.

6.3 Hazards

Due to the out-of-order execution across the different sequences, RAW, WAW and WAR hazards are all possible. Furthermore, supporting vector chaining implies that these hazards should be resolved at sub-vector-register granularity. Since Saturn is designed around `DLEN`-wide element groups as the base throughput of the machine, Saturn resolves data hazards at `DLEN` granularity. The scheduling mechanism precisely tracks which element groups an instruction or operation has yet to read-or-write to interlock the sequencers.

In Saturn, the "out-of-order instruction window" includes all instructions in the issue queues (but not the VDQ), the instructions currently in progress within the sequencers, and any operations which have not yet completed execution in the functional unit datapaths. Each instruction in this window must advertise a precise set of element groups that it will read from or write to in future cycles, along with its age tag.

- Instructions in the issue queues already contain their operand specifiers. Since these instructions have not yet been sequenced, a conservative bound on the element groups to be accessed can be easily computed using the `LMUL` and the base register operand.
- The sequencers track a precise bit-mask of element groups that the currently-sequenced instruction may still access. For regular vector instructions that access their operands sequentially, the sequencers can clear these bits with each issued operation. For irregular vector instructions, the sequencers can conservatively leave these bits set.
- Operations inflight in the functional units that have yet to write back track a single element group of the write destination.

The advertised information across the out-of-order window is aggregated into a pending-read and pending-write one-hot vector for each sequencer. These one-hot vectors each contain one element for each architectural element group in the VRF, which is 32xVLEN/DLEN (a typical total value being 64). These one-hot vectors are constructed using an age filter based on the age tag of the current instruction in the sequencer and the age of each operation in the out-of-order window. The age filter restricts the pending-read and pending-write vectors to only pending reads and writes from instructions older than the currently sequenced instruction.

In some cases, the relative age is unambiguous, so no age filter is needed. Instructions in the sequencer are inherently older than instructions from the feeding issue queue for that sequencer, so

no age filter is needed. Sequenced operations in the VXUs are inherently the oldest writes to any element group, so no age filter is needed for these either.

Each sequencer computes the element groups that will be accessed or written to by the next operation to be issued, and determines if a pending older read or write to those element groups would induce a RAW, WAR or WAR hazard. If there is no data hazard and there is no structural hazard, the operation can be issued, with the sequencer incrementing its internal element index counter, or draining the instruction.

For vector instructions with regular consecutive access patterns, the last issued operation that accesses some element group can clear the sequencer's internal bit-mask of pending reads and/or writes. This frees younger vector instructions in other sequencers to chain off this element group as soon as possible.

6.4 Vector Register File

The VRF is organized as a multi-ported banked array of flops. The architectural register file is striped across the banks by element group index. Neighboring element groups reside in neighboring banks. Each bank contains 3 read ports and 1 write port, to fulfill the minimum read requirements of a three-input fused-multiply-add. The generator supports generating VRFs with 1, 2, or 4 banks. Configurations that expect to keep multiple integer sequencers utilized simultaneously will prefer more banks to meet the increased read bandwidth requirements.

As an optimization, Saturn implements per-bank single-entry fall-through write buffers, effectively emulating a 3R2W memory with a 3R1W memory. Write bank conflicts between pipelined writebacks and load writebacks can result in a performance penalty. The write buffers allow two writes into a single bank, with the second write delayed onto the next cycle. Due to the bank striping of the vector register file, the delayed write is unlikely to induce further bank conflicts.

A shadow copy of vector mask register `v0` is maintained in a 1R1W memory to avoid provisioning an extra read port for the bulk banked register file.

A read crossbar connects the issue port of the sequencers to the register file read ports. The read crossbar resolves structural hazards during the read stage and stalls the sequencers if necessary. The read stage also arbitrates for access to the write ports across multiple fixed-latency execution paths.

6.5 Functional Units

Each execution unit is composed of some set of functional units. Operations are issued to functional units along with their vector operands. Functional units can either be pipelined or iterative.

Pipelined functional units have fixed execution latencies, so once an operation is issued, it will execute without stalling. The sequencing mechanism checks for future write port conflicts on the

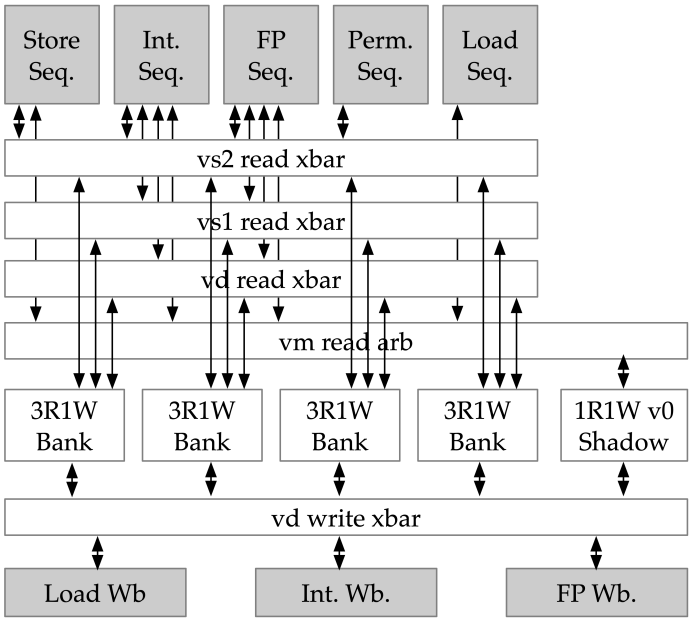

Figure 23: Banked vector register file for a 4-bank configuration

target VRF bank across inflight and simultaneously sequenced operations to ensure that the next sequenced operation will not induce a structural hazard on the write port in the future. If a conflict is detected, the younger operation will be stalled and will likely start executing the very next cycle.

Iterative functional units have variable execution latencies or contain expensive hardware such that it is desirable to execute at a rate of just one element per cycle. Once an iterative functional unit has completed its operation on a given element, it will arbitrate for the target VRF write port, write the result, then assert readiness to accept a new operation from the sequencer.

Table 1 conveys the default design of all the functional units in Saturn, including their supported instructions, microarchitectures, and latencies. Many of these parameters can be adjusted in the generator; the default settings are shown.

Name	Instructions Supported	Micro-Arch.	Structure	Notes
`IntPipe`	Integer add/sub Integer max/min	Multi-element-width adders and comparators	2-stage pipe	Only saturating adds write back in stage 2, all others write back in stage 1
`ShiftPipe`	Shifts	SIMD array of barrel-shifters	2-stage pipe	
`BitwisePipe`	Bitwise insns.	Bitwise array	1-stage pipe	
`BitmanipPipe`	Zvbb insns.	Multi-element-width priority encoders	2-stage pipe	
`PrefixUnit`	Prefix-like insns. (popc/first/etc.) Scalar write-backs	Prefix-sum circuit w. accumulator	Stateful 1-stage pipe	
`IntDivide`	Integer divides Opt. multiplies	Iterative FSM	Iterative elem-wise	Can be configured to support multiplies in min-area configurations
`MulPipe`	Integer multiply	Single elem-wise multiplier SIMD-array of multipliers	3-stage pipe	Build the elem-wise multiplier in min-area configurations
`PermuteUnit`	Slides, gathers, compress	Minimal logic	1-stage pipe	Manages write-backs for register-permutations
`FMAPipe`	FP multiplies, adds	SIMD array of FMAs Port to host CPU's FPU	4-stage pipe	Share the host core's FPU in min-area configurations
`FPDivSqrtUnit`	FP Divide, Square root	Iterative unit	Iterative elem-wise	
`FPConvPipe`	FP convert, compare	SIMD array of FP units	2-stage pipe	

Table 1: Saturn functional unit configurations

7 Programming Guide

Performant vector code on Saturn is generally straightforward to implement and requires less microarchitectural fine-tuning than other vector or SIMD systems. Still, some general guidelines for vector programming apply. This section is divided into discussions on microarchitecturally-relevant areas of interest when writing vector code.

Saturn's implementation is built around the machine datapath width, `DLEN`. The fundamental occupancy of one vector instruction, or chime length, in a given Saturn implementation is `VLEN/DLEN`. In most implementations, this will be 2, although Saturn can generate implementations with a chime length of 1 or greater. Using the `LMUL` parameter allows the programmer to extend the chime length to `LMUL x VLEN/DLEN`, which increases the effective occupancy of each vector instruction.

Generally speaking, efficient vector code should maximize the utilization of the functional units. Saturn efficiently implements vector register grouping, and thus programmers should generally use the largest possible `LMUL` that applications allow without causing excessive vector register spilling. Maximizing `LMUL` decreases dynamic instruction count and increases chime length, avoiding underutilization due to insufficient instruction throughput.

7.1 Optimizing Towards Chaining

Leveraging vector chaining is critical for achieving high utilization of the functional units. Saturn implements vector chaining at `DLEN`, or element-group granularity, through the vector register file (direct functional unit to functional unit bypasses are not implemented). Chaining enables a dependent vector instruction to begin sequencing operations as soon as its parent instruction writes back its first element group, assuming no other structural or data hazards.

Chaining only occurs between instructions occupying different sequencers, as the sequencers are single-issue in-order. All sequencers can chain between each other. For example, arithmetic operations may chain off of memory operations and vice-versa. When using a configuration with multiple execute sequencers, independent arithmetic operations may chain off each other and overlap execution. Optimized vector code should seek to load-balance instructions across the sequencers such that chaining might occur.

7.2 Optimizing Around Issue Limitations

To maximize the utilization of all functional units, the issue structure of the machine should be considered. The issue queue structure and depths are parameterized in Saturn and may differ across different configurations.

In the "Unified" structure, all vector arithmetic instructions, whether integer or floating-point,

execute from the same sequencer, so no parallel arithmetic execution is possible. In the "Shared" and "Split" structures, independent sequencers drive the integer and floating-point execution units, so parallel execution is possible under certain conditions. In the "Shared" configuration, since the sequencers share a single in-order issue queue, interleaving vector instructions across the integer and floating-point paths is critical. In the "Split" configuration, each sequencer has its own issue queue, reducing the importance of software-scheduled load balancing.

When the issue queue sizes are shallow, the issue queues may fill up and back-pressure the decoupling queue, and consequently, the scalar core's frontend. Even modest configurations of Saturn should contain enough buffering within the issue and decoupling queues to avoid back-pressure for well-load-balanced code.

7.3 Optimizing Around the Scalar Core

Saturn is designed to be performant even when integrated with a compact, single-issue, in-order scalar core. Specific details related to Rocket and Shuttle can be found in Section 4. When writing loops, scalar bookkeeping instructions should be scheduled to overlap with the execution of the vector instructions.

Saturn configurations with short chime lengths are preferably integrated with Shuttle, as short-chime instructions generally require higher frontend instruction throughput. When operating on long vector lengths with long chimes, the performance difference between Rocket and Shuttle diminishes. As discussed in Section 4, the handling of `vset` instructions varies across different scalar cores. This may induce pipeline bubbles and should be considered when writing kernels.

Due to the post-commit execution of vector instructions, vector instructions that write scalar registers should be used sparingly. As the supported scalar cores do not implement speculative execution, RAW dependencies on vector-produced scalar results will cause the scalar pipeline to stall. These should be avoided in inner loops.

7.4 Optimizing Around Pipeline Latencies

A single vector instruction requires `VLEN/DLEN` (the chime length) cycles of occupancy on most `DLEN`-wide functional unit datapaths. When this chime length is less than the pipeline depth of the relevant functional unit, the proceeding instruction will stall due to a data dependency on the yet-to-be-produced writeback data. This situation is rare due to the support for chaining, but might still appear in a sequence of low-`LMUL` floating point instructions.

To saturate the FMA units in this scenario, either a longer `LMUL` should be used, or independent FMAs must be scheduled back-to-back. Generally, performant code should use the highest `LMUL` possible that avoids vector register spilling.

7.5 Optimizing Segmented Memory Accesses

Saturn's implementation strives to implement complex instructions efficiently. In particular, the segmented loads and stores that reformat memory into vector registers are implemented to be able to fully utilize the wide memory port. As a general rule, using such complex instructions should never induce a performance penalty over an equivalent longer sequence of simpler instructions. Thus, programmers should use complex instructions whenever possible, even if the kernel could be equivalently implemented by a longer sequence of simpler instructions.

7.6 Optimizing Reductions

Currently, Saturn cannot maintain full utilization for element-wise reductions due to an insufficient number of accumulator registers to track intermediate values. Classic techniques for vectorizing reduction operations should be utilized where possible. For example, applicable reduction loops should be expressed as a sequence of element-wise operations before a final reduction across all elements in the tail code.

8 History

8.1 Prior Vector Implementations

Saturn draws from a long line of prior vector units.

8.1.1 Torrent-0

T-0 [16] was an early demonstration of the principles of instruction scheduling in short-vector microarchitectures. Torrent demonstrated a decoupled-access-execute-style microarchitecture with efficient scheduling of short-chime vector instructions. However, Torrent relied on fixed memory and pipeline latencies for instruction scheduling, relying on a global stall to handle memory stalls.

Many ideas presented in Torrent have been refined toward implementations of a modern vector ISA like RVV. Precise faults, segmented memory operations, register grouping, and mixed-precision instructions are modern innovations in vector architectures that have significant implications on vector microarchitectures and instruction scheduling.

8.1.2 Hwacha

Hwacha [2] was a decoupled-fetch vector unit designed to support scalability to many-lane designs. Microarchitecturally, Hwacha is a classic "long-vector" machine, with deeply temporal vector execution distributed across parallel independent vector lanes. Each vector lane contained an independent instruction sequencer, independent functional units, a slice of the register file, and an independent memory port. Like Saturn, Hwacha was implemented in Chisel and could integrate with Rocket as a post-commit decoupled unit.

Notably, Hwacha executes a unique "decoupled vector fetch" ISA. In Hwacha's ISA, vector code is encoded in "vector fetch blocks". These vector fetch blocks are independently fetched and executed by a limited frontend in the vector unit, instead of being embedded in the same sequential instruction stream as scalar code. Programmers are expected to describe stripmine loops as these vector fetch blocks, and dispatch pointers to these blocks to the vector unit via "vector fetch" instructions. Hwacha's ISA also does not require scala-vector memory ordering or support precise faults, simplfying Hwacha's microarchitecture for address translation and disambiguation.

The component of Hwacha's microarchitecture most relevant to modern scalable vector ISAs is the instruction sequencer. However, Hwacha's instruction sequencer is designed for the needs of a deeply temporal long-vector implementation, rather than a short-chime vector unit like Saturn. Hwacha's sequencer imposes a small structural dead time overhead as instructions do not drain from the sequencer until their final element has been written back fully. This results in high sequencer entry pressure for short-chime vector code blocks. Compared to Hwacha's sequencer,

Saturn's sequencers aggressively drain short-chime instructions without requiring long occupancies in limited structural resources.

8.1.3 LEM

LEM [17] was an experimental microcoded RVV vector unit designed around an early variant of the Shuttle superscalar core. Some ideas around instruction decode in LEM inspired the vector decode generators in Saturn. LEM was the first Chisel implementation of an RVV-compliant vector unit.

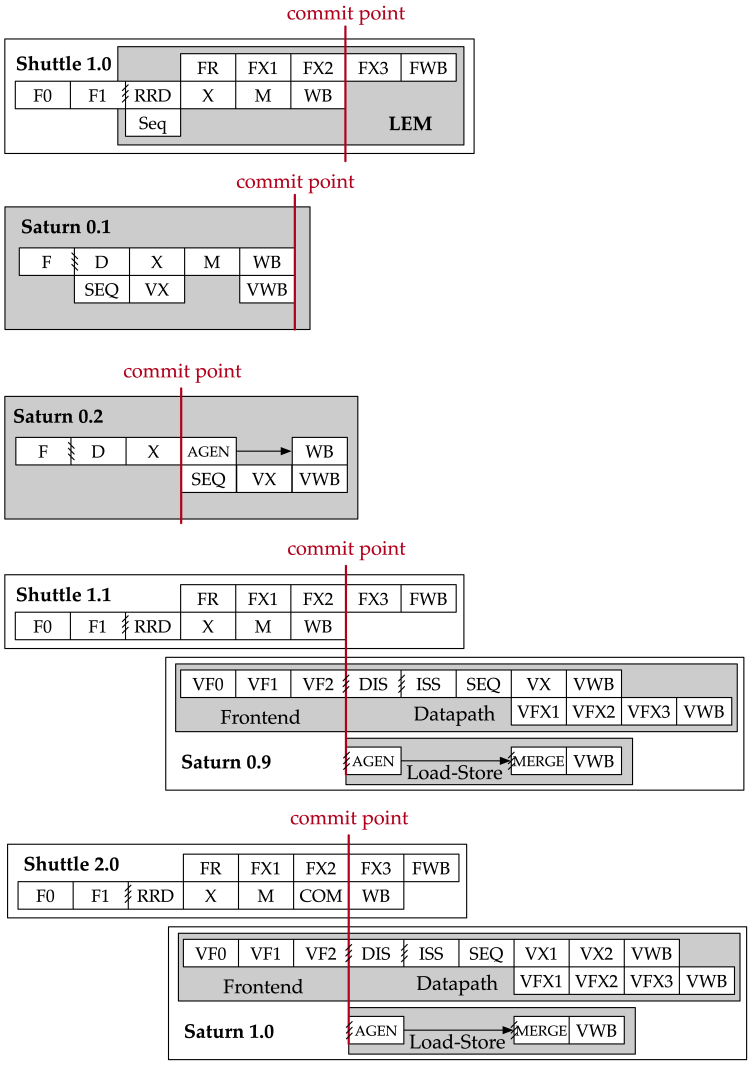

Figure 24: Evolution of Saturn towards Saturn 1.0

8.2 Timeline

The Saturn project began in the fall of 2021 with a vision to develop a standard base vector implementation for hosting specialized accelerators. Early iterations of Saturn explored different design approaches to compact vector implementations, including shallow-pipeline systems, merged scalar-vector implementations, and partial implementations of the full RISC-V Vector ISA.

Saturn 0.1 implemented an aggressive ahead-of-commit vector pipeline, with a custom scalar core implementation. Saturn 0.1 had unstable and unpredictable performance characteristics due to the complex set of interlocks needed to enable parallel vector and scalar execution. Specifically, allowing for variable-chime instruction execution significantly increased the complexity of the tightly integrated vector unit due to the implications on instruction occupancies of shared scalar/vector pipeline structures. Only a small subset of vector instructions were supported on this implementation.

Saturn 0.2 implemented a post-commit vector unit behind a custom-designed superscalar core with a very shallow pipeline and no support for virtual memory. A shared scalar-vector load-store-unit executed both scalar and vector memory operations post-commit, with a specialized high-bandwidth non-blocking L1 data cache fulfilling both scalar and vector memory requests. A very shallow three-stage pipeline pre-commit attempted to mitigate high scalar-load-use penalties, but this precluded support for virtual memory. Saturn 0.2 did implement the first version of the instruction sequencing mechanism used in Saturn 1.0.

Saturn 0.9 began development in the fall of 2023 with a complete rewrite over prior versions. The desire to support complete RVV alongside a fully-featured scalar core was motivated by the observation that accelerator architects generally expect mature software stacks and well-behaved core implementations for scalar control code. Furthermore, learnings from Saturn 0.1 and 0.2 suggested that a "three-stage" load path, with ahead-of-commit fault checking, early post-commit load-address-generation, and late post-commit load write-back, would adequately address performance requirements while remaining compliant to the ISA.

The decoupled-access-execute architecture with an aggressive yet precise vector frontend became the dominant design paradigm driving the rest of Saturn 0.9's microarchitecture. The implementation of Saturn focused on developing a robust precise fault mechanism, load-store path, and dynamic out-of-order instruction issue in that order. Once the microarchitecture around dynamic vector scheduling with chaining was refined, implementation of the functional units proceeded expediently. After the first vector integer ALU was implemented, the rest of the vector instructions were brought up in three months.

At the time of writing, Saturn 0.9 has been taped out twice. In spring 2024, a conservative multi-core Saturn configuration attached to Rocket was taped out out as part of Berkeley's undergraduate tapeout course. Concurrently, a more aggressive heterogeneous clustered multi-core of Saturns attached to Shuttle cores was taped out in a research SoC.

Saturn 1.0 was developed alongside Shuttle 2.0, leveraging physical design experience from the prior tapeouts. Saturn 1.0 contained many critical path and area optimizations and was released in the fall of 2024 along with this microarchitecture manual.

8.3 Contributions

Jerry Zhao led the Saturn project and implemented the instruction scheduling mechanism, load-store unit, and scalar-core integration. Daniel Grubb led the implementation of the SIMD floating-point units and the Saturn physical implementation and contributed to the core subsystems of the vector backend. Miles Rusch implemented several variants of SIMD integer multipliers. Tianrui Wei assisted with early versions of Saturn and performed key work that would evolve into the 1.0 version.

Kevin He, Nico Casteneda, and Mihai Tudor led a tapeout of a conservative Saturn with Rocket implementation as part of Berkeley's undergraduate tapeout course. Kevin Anderson, Daniel Grubb, and Vikram Jain led a research tapeout of a DSP-optimized chip with heterogeneous clusters of Saturns.

Many people have contributed to projects that have led to Saturn. John Fang developed LEM, exploring the challenges of generating efficient vector decode structures. Albert Ou's thesis work highlighted use cases for Saturn-like vector units in accelerating DSP kernels.

Lastly, the wisdom of former graduate students who have developed vector units during their tenures has proven instrumental to the development of Saturn. Conversations with Albert Ou, Colin Schmidt, Andrew Waterman and Chris Batten have all been insightful.

8.4 Funding

Research was partially funded by SLICE Lab industrial sponsors and affiliates, and by the NSF CCRI ENS Chipyard Award #2016662. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] "SX-Aurora TSUBASA Architecture Guide Revision 1.1," 2018. [Online]. Available: https://sxauroratsubasa.sakura.ne.jp/documents/guide/pdfs/Aurora_ISA_guide.pdf

[2] Y. Lee, C. Schmidt, A. Ou, A. Waterman, and K. Asanović, "The hwacha vector-fetch architecture manual, version 3.8.1," Tech. Rep. UCB/EECS-2015-262, Dec. 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-262.html

[3] M. Perotti, M. Cavalcante, R. Andri, L. Cavigelli, and L. Benini, "Ara2: Exploring Single- and Multi-Core Vector Processing with an Efficient RVV1.0 Compliant Open-Source Processor," Nov. 2023. [Online]. Available: http://arxiv.org/abs/2311.07493

[4] F. Minervini, O. Palomar, O. Unsal, E. Reggiani, J. Quiroga, J. Marimon, C. Rojas, R. Figueras, A. Ruiz, A. Gonzalez, J. Mendoza, I. Vargas, C. Hernandez, J. Cabre, L. Khoirunisya, M. Bouhali, J. Pavon, F. Moll, M. Olivieri, M. Kovac, M. Kovac, L. Dragic, M. Valero, and A. Cristal, "Vitruvius+: An Area-Efficient RISC-V Decoupled Vector Coprocessor for High Performance Computing Applications," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 2, pp. 28:1–28:25, Mar. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3575861

[5] C. S. Anderson, J. Zhang, and M. Cornea, "Enhanced Vector Math Support on the IntelAVX-512 Architecture," in *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*. Amherst, MA, USA: IEEE, Jun. 2018, pp. 120–124. [Online]. Available: https://ieeexplore.ieee.org/document/8464794/

[6] "Arm Architecture Reference Manual for A-profile architecture." [Online]. Available: https://developer.arm.com/documentation/ddi0487/ka/

[7] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, Mar. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7924233/

[8] "Armv8-M Architecture Reference Manual," 2015. [Online]. Available: https://developer.arm.com/documentation/ddi0553/bx/?lang=en

[9] "RISC-V "V" Vector Extension," Sep. 2021. [Online]. Available: https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf

[10] M. Cavalcante, D. Wüthrich, M. Perotti, S. Riedel, and L. Benini, "Spatz: A Compact Vector Processing Unit for High-Performance and Energy-Efficient Shared-L1 Clusters," in

Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design, ser. ICCAD '22. New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 1–9. [Online]. Available: https://dl.acm.org/doi/10.1145/3508352.3549367

[11] M. Platzer and P. Puschner, "Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation," pp. 18 pages, 831 915 bytes, 2021. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2021.1

[12] K. Patsidis, C. Nicopoulos, G. C. Sirakoulis, and G. Dimitrakopoulos, "RISC-V2: A Scalable RISC-V Vector Processor," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, Oct. 2020, pp. 1–5. [Online]. Available: https://ieeexplore.ieee.org/document/9181071

[13] 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 *DAC Design Automation Conference 2012*, Jun. 2012, pp. 1212–1221. [Online]. Available: https://ieeexplore.ieee.org/document/6241660

[14] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9099108

[15] J. E. Smith, "Decoupled access/execute computer architectures," *ACM SIGARCH Computer Architecture News*, vol. 10, no. 3, pp. 112–119, Apr. 1982. [Online]. Available: https://dl.acm.org/doi/10.1145/1067649.801719

[16] K. Asanović, "Vector microprocessors," Ph.D. dissertation, EECS Department, University of California, Berkeley, 1998. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/6404.html

[17] Z. Fang, "LEM: A configurable RISC-v vector unit based on parameterized microcode expander," no. EECS-2022-150, 2022. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-150.pdf