Enabling Vector Load and Store Instructions on HammerBlade Architecture

Robert Ramstad

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science

University of Washington

2024

Committee:

Michael Taylor

Jacques Rudell

Scott Huack

Program Authorized to Offer Degree:
Electrical and Computer Engineering

University of Washington

Abstract

Enabling Vector Load and Store Instructions on HammberBlade Architecture

Robert Ramstad

Chair of the Supervisory Committee:

Michael Taylor

Electrical and Computer Engineering

Traditionally, computer architecture has been dominated by overly complex instruction sets that created a "solution" to every problem by adding another instruction. If these complex instructions sets are one side of a coin, the Reduced Instruction Set Computer (RISC)-V [105] architecture is the other. RISC-V processors consist of 47 base instructions. Having such a low amount of instructions is both the biggest strength and the biggest weakness of the new era of RISC-V processors. Currently, there is a remarkable lack of high performance RISC-V processors. The Hammerblade[39] architecture is one of the few. The main difference between Hammerblade and other RISC-V processors is its leveraging of parallel computer architecture as a multi-core system. However, while Hammerblade consists of potentially thousands of cores, it does not perform any data-level parallelism.

The primary intent of this thesis is to further understand how to increase the local memory throughput by extending the RISC-V core to include Single Instruction Multiple Data (SIMD) loads and stores. This will add the capability to locally load and store four words of data on top of the existing singular word loads and stores. A single Vanilla RISC-V core[40] can now have the potential of a 4x speedup in loads and stores. This will allow Hammerblade to not only leverage the parallelism of the manycore architecture, but also the data-level parallelism on each individual core.

Acknowledgement

I would like to thank my advisor Professor Michael Taylor, and BSG member Tommy Jung, for their help.

Declaration

I, Mr. Robert Ramstad hereby declare that this thesis is the record of authentic work carried out during the academic year 2023 - 2024 and has not been submitted to any other University or Institute towards the award of any degree.

BSG Acknowledgement

Portions of this work were partially supported by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement numbers FA8650-18-2-7863 and FA8650-18-2-7856; NSF grants SaTC-1563767, SaTC-1565446. This work intersects and leverages research and infrastructure created by the members of the Bespoke Silicon Group, spanning across accelerators ([16, 28, 109, 106, 65, 64, 111, 12, 103, 76, 34, 81, 5, 77, 33, 35, 57, 102]), ASIC Clouds ([98, 107, 96, 85, 59, 58, 62, 88]), open source hardware ([97, 87, 26]) RISC-V ([68, 75, 74, 24, 110, 2, 101, 53, 73, 63, 20, 72]), Network-on-Chips ([55, 69, 112, 92, 60]), security ([19, 45, 44, 4]), benchmark suites ([99, 61, 10]), dark silicon ([84, 83, 84, 89, 14, 36, 33, 102]), multicore ([68, 42, 41, 43, 86, 92, 91, 95, 82, 60, 90, 93, 94, 104]), compiler tools ([48, 30, 49, 32, 51, 50, 31, 108, 1, 8]) and FPGAs ([113, 47, 10]).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Even with the RISC-V open source revolution[27] still in its infancy, the world of hardware design has been drastically changed. Traditionally instruction set architectures (ISA's) are hidden behind closed doors and require potentially millions of dollars to get permission to use[7], the RISC-V ISA offers a new approach. Currently there is a lack of powerful RISC-V cores, not due to limitations of the ISA itself, but limitations of the designs available.

The HammerBlade Manycore[75] is a processing behemoth. Simply put, this chip is a modern spectacle of parallel computing. Each core is made up of compute tiles connected via a mesh-like network on chip (NoC)[22]. The software running on HammerBlade is written in HammerBlade CUDA-lite allowing for dispatching of workloads to subsets of the core. All of this is compiled by the open source GNU Compiler Collection (GCC).

While the HammerBlade Manycore takes advantage of parallel computing via multiple cores, it currently does not take advantage of any parallel processing in the individual cores. This is a huge missed opportunity, as each and every one of the cores can be optimized to load and store much quicker.

This thesis will summarize my work while I address the problem proposed above by investigating potential solutions to vectorizing memory access in HammerBlade. This will only be for float loads and stores that occur locally within a singular RISC-V core. While most of my work was concentrated on the hardware changes, I also worked within GCC to add custom instructions and rewrote multiple benchmarks using said instructions.

1.1 Chapter 1.1. Thesis Overview

The goal of this thesis is to provide a survey of modern parallel computer architecture, my proposed architecture changes, and future work to be done.

Chapter 2. Literature Review

This chapter goes over relevant literature and background information required to fully understand the task at hand. It will primarily focus on ISA design, vectorization techniques, and HammerBlade specific architecture and programming.

Chapter 3. RISC-V Vanilla Core Design Overview

All changes made to the existing RISC-V Vanilla Core are outlined and described. First going over the intended use of the added instructions, followed by the changes required to make it happen.

Chapter 4. Verification and Benchmarks

The scope of this thesis does not include full compiler support in the GCC toolchain, however some work must be done for the system to understand the added instructions. This chapter covers all these changes and why they must be made for proper testing of SIMD instructions. In addition, it describes how the core reacts to a variety of benchmarks in the HammerBlade benchmark suite, quantifying the gains achieved from the changes made.

Chapter 5. Cost Analysis

This chapter focuses on the cost of implementing the augmented core. The power and area costs are analyzed to further understand the cost of modification.

Chapter 6. Conclusions and Future Works

The final chapter talks about where the logical next steps could be to further advance this research. It also reflects on the issues and limitations of the current implementation.

Appendix. Python Code for Power Analysis

The appendix contains the code related to calculating the net zero power breakpoint.

Chapter 2

Literature Review and Background

2.1 HammerBlade

The HammerBlade Manycore is a highly parallel processor that is designed with open source hardware in mind. The basic format of this architecture is a tile array connected by a mesh network. The Celerity chip (2017)[3] was one of the first fully taped out versions of the HammerBlade architecture. Celerity is a 511 core system built in 16nm which achieved a world record in RISC-V and Coremark performance. This chip is also supported by the RISC-V Rocket Core [6] and a binarized neural network accelerator for real time computer vision applications.

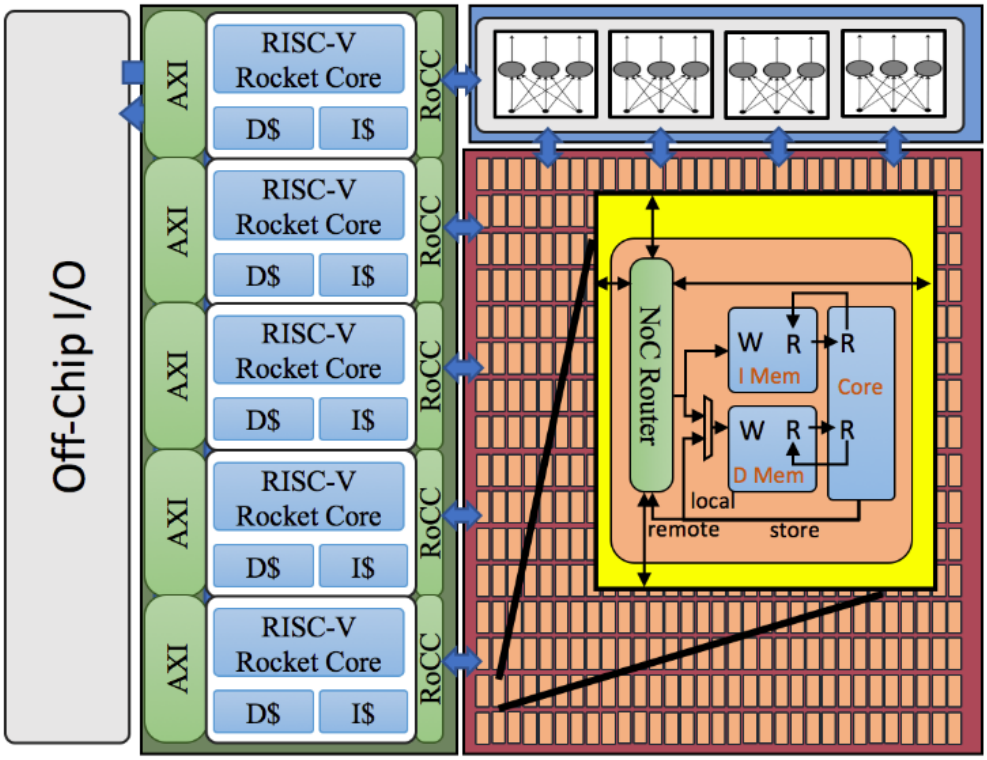

Figure 2.1: Architecture of Celerity Chip [75]

2.1.1 Architecture

In the simplest terms, the HammerBlade architecture is an array of interconnected tiles by a mesh network on chip (NoC). Each tile can serve a different function, either as a throughput optimized Vanilla RISC-V cores[67], or any custom accelerator. To support these tiles is victim cache (Vcache) tiles to coordinate with off-chip DRAM (HBM2).

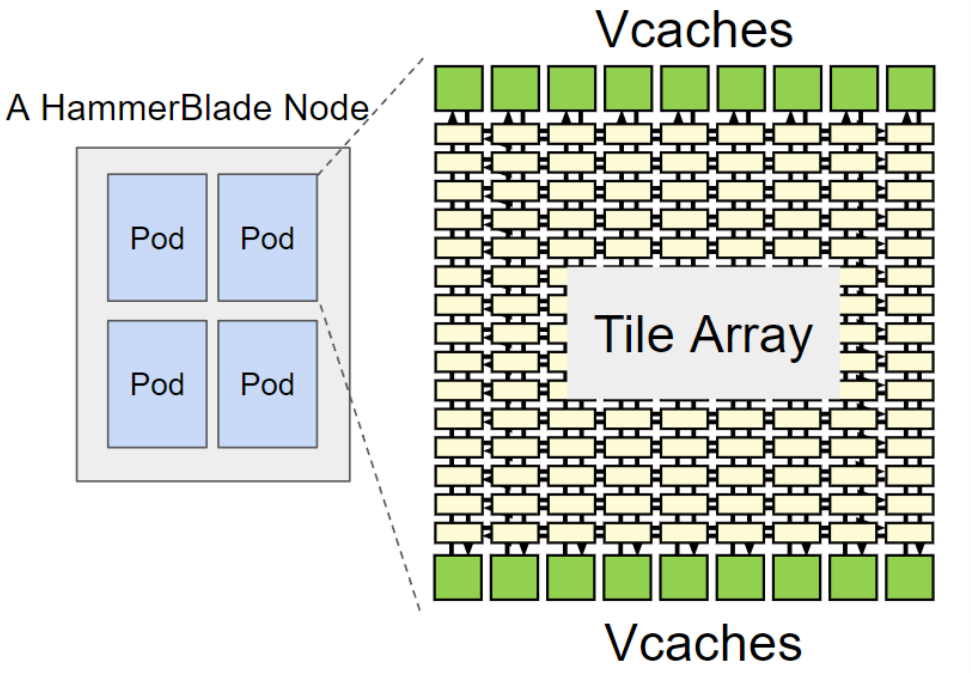

Figure 2.2: HammerBlade Node and Tile Layout [21]

2.1.2 On-Chip Network

The HammerBlade architecture makes use of a Ruche Network[22] which is a 2-d mesh with additional parallel links. This network has two separate networks, the forward and reverse network. The forward network deals with remote load and store requests while the reverse network deals with all responses. The Ruche network is dimension-order routed, meaning that it is either X-then-Y in the case of the forward network, or Y-then-X for the reverse network. Dimension-order routing prevents deadlock and also removes path diversity [70].

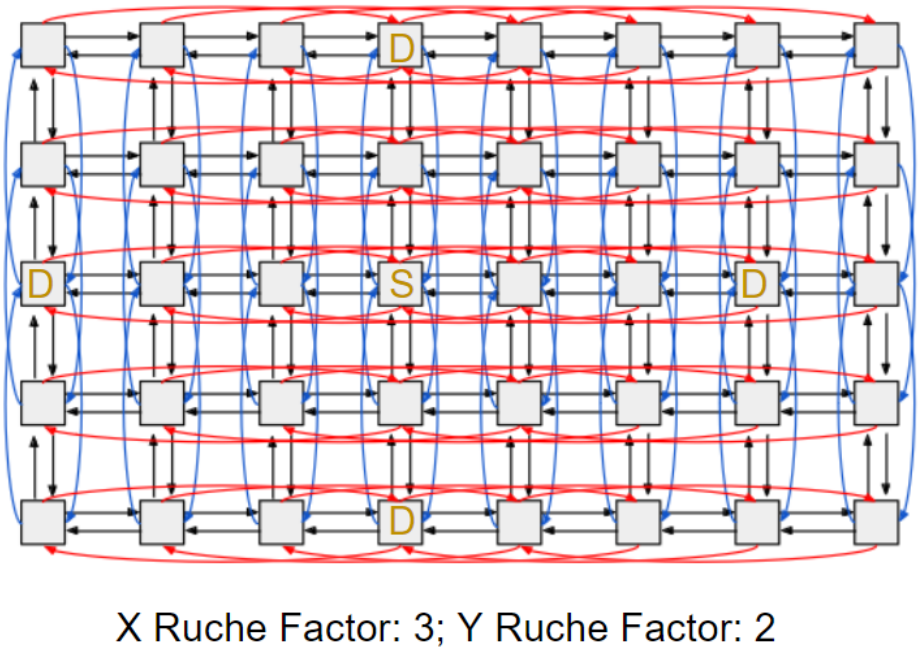

Figure 2.3: Ruche Network [21]

2.1.3 Cache Tiles and Memory

As seen in Figure 2.2, each tile array is supported by rows of victim caches (Vcaches). These Vcaches communicate with memory controllers that manage various types of high bandwidth memory [52].

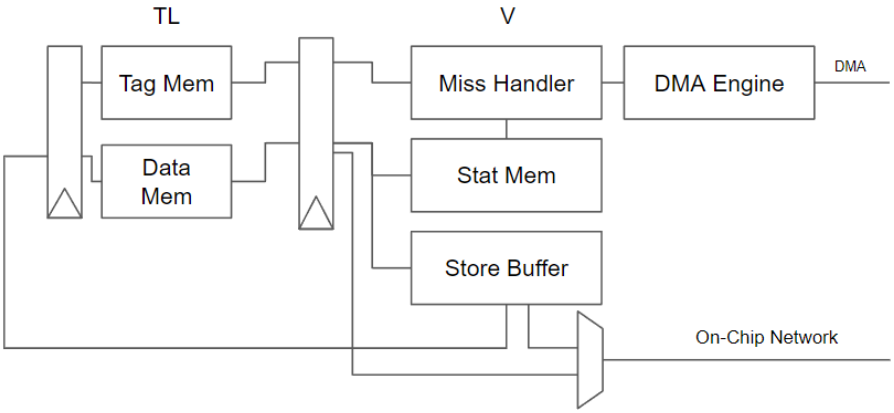

Figure 2.4: Cache Tile Logic [37]

The Vcache follows a 2-stage blocking cache, tag lookup (TL) and verify (V). The Vcache will handle one miss at a time and those misses will stall the cache pipeline. Vcaches are 8-way set associative[54]. To communicate with the Vcaches, compute tiles will send request packets out to the network which are dealt with and responded to by the respective Vcache. The off-chip DRAM address space is interleaved between the Vcache tiles to balance requests among the Vcache array.

2.1.4 HammerBlade Software

The HammerBlade software architecture follows the CUDA-lite model[100]. This is a derivative model of the CUDA model popularized by NVIDIA[25]. The main idea behind this programming model is to treat the GPU as a separate workhorse. The host dispatches instructions to portions of the GPU which subsequently computes and responds with the relevant data. On HammerBlade this is done via tile groups.

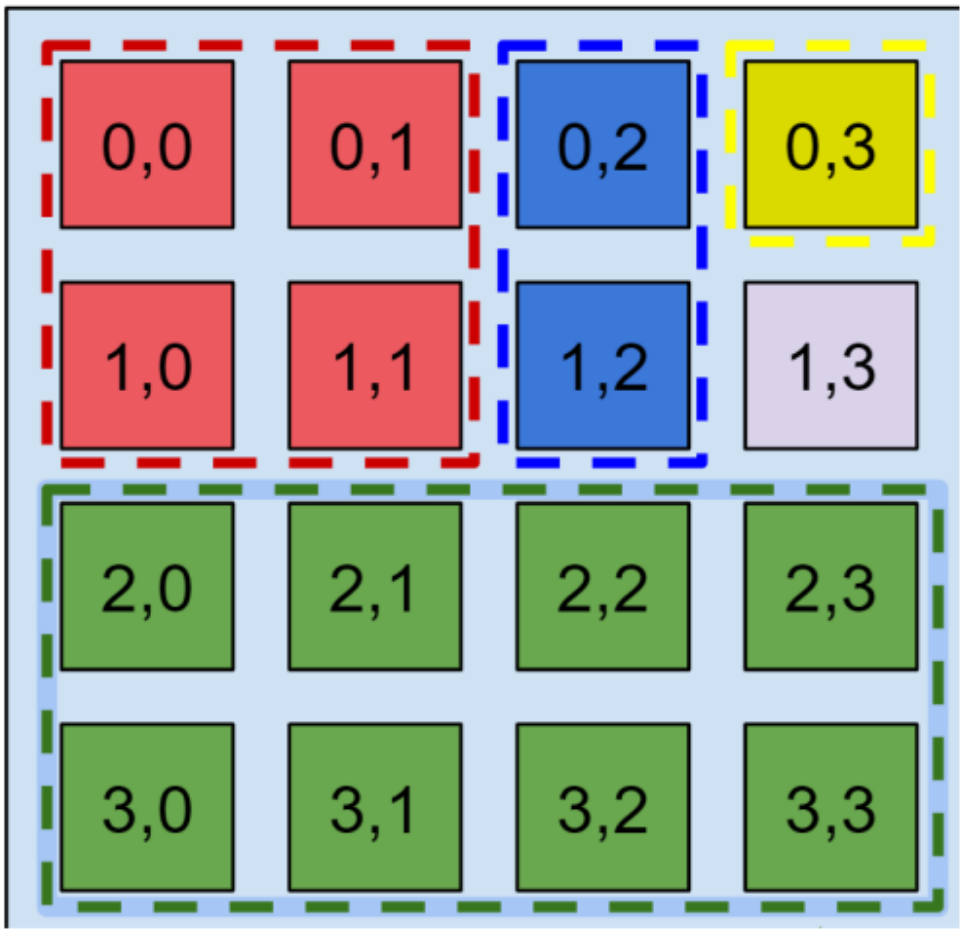

Figure 2.5: Tile Group Layout of SPMD Software [15]

Each of these groups collectively execute a kernel in parallel, share local memory, and are synchronized using barriers. This allows the desired code to be split into manageable slices and dispatched to tile groups. Each individual tile has access to its current tile group dimensions, their position within said group, and their portion of the entire work[15].

2.1.5 HammerBlade Compute Tiles

While the compute tiles on the HammerBlade system can be any specialized accelerator, the general purpose Vanilla Core will be the primarily focus. Similarly, it is where most of this thesis' work will be done. The Vanilla Core is a 5-stage in-order RISC-V core. The standard Vanilla Core implements the RV32IMF(A) ISA. This thesis proposes the addition of two new vector-like instructions to this base ISA. The Vanilla Core has a 3-stage ALU, and 4KB i-cache (IMEM) and local DMEM. The Vanilla Core supports non-blocking remote loads which allows it to have pending memory operations (in this case remote loads) while still allowing for other instructions to complete. These tiles also include a network router, to send and receive data from the network itself.

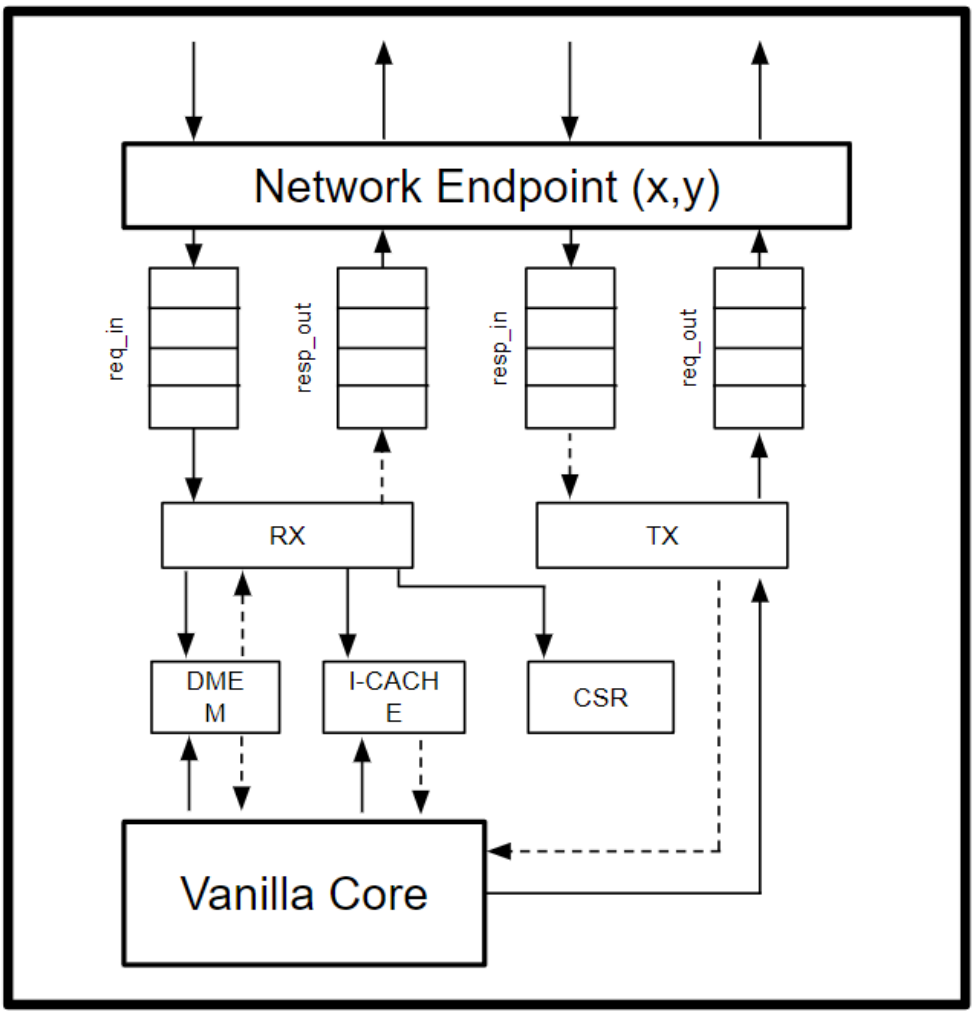

Figure 2.6: Compute Tile Layout [37]

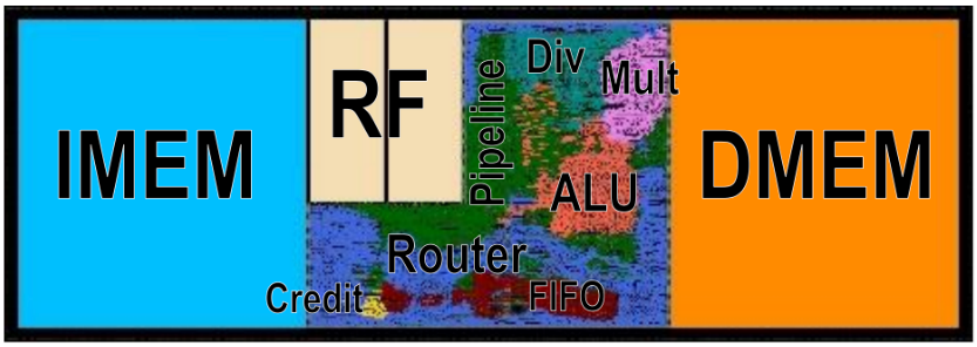

Figure 2.7: Tile Floor Plan in TSMC 16nm [21]

2.2 Vanilla Core

The Vanilla Core is the workhorse and most basic building block of the HammerBlade chip. While any type of accelerator is accepted, The Vanilla Core is the standard compute tile used in the HammerBlade architecture.

2.2.1 Instruction Set Architecture

The Vanilla Core is a 5-stage partially in-order RISC-V processor that implements the RV32IMF(A) ISA. Rather than being optimized for individual core performance, it is optimized for aggregated performance. The Vanilla Core is silicon proven with both 180nm and 16nm tapeout and a prototype manycore chip with 16x31 cores achieved a record CoreMark score of 812,350[37].

2.2.2 Programming Model

The Vanilla Core operates in a SPMD (Single Program Multiple Data) [23] setting when used in the HammerBlade architecture. Each Vanilla Core is computing the same program on a different subset of the data that is dispatched via the host tile. Each Vanilla Core operates in a 32-bit virtual address, called the Endpoint Virtual Address (EVA) allowing for in-group memory sharing.

Resource	Address Format
Local DMEM	0000_0000_0000_0000_0001_????_????_????
Global	01yy_yyyy_xxxx_xx??_????_????_????_????
In-Group	001y_yyyy_xxxx_xx??_????_????_????_????
L2/DRAM	1xx?_????_????_????_????_????_????_????

Figure 2.8: EVA Mapping in HammerBlade Architecture [37]

- Local DMEM: Accesses the local 4KB SRAM on an individual tile.
- Global: Accesses remote endpoints using x,y coordinates.
- In-Group: Accesses the tiles via tile group origin. This is set via configurable registers on each Vanilla Core.
- L2/DRAM: Each address of the off-chip DRAM corresponds to an address in each L2 cache located on the border of the tile array. The x-coordinate and virtual bank address provide a physical address in the off-chip DRAM.

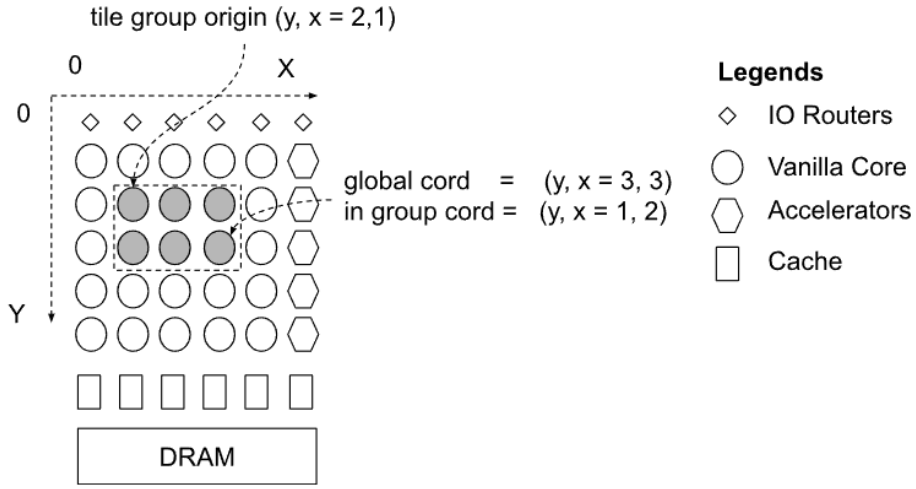

Figure 2.9: Vanilla Core Tile Group Coordinates [37]

2.2.3 Pipeline Architecture

The Vanilla Core follows the standard 5-stage pipeline (IF,ID,EXE,MEM,WB). However, there are a few unique quirks to the Vanilla Core. After ID, the pipeline splits into two paths EXE and FP_EXE (Floating point execute). The EXE path has the LSU, ALU, FP_INT, and IDIV units while the FP_EXE has a 2-stage FPU and an FDIV/FSQRT unit.

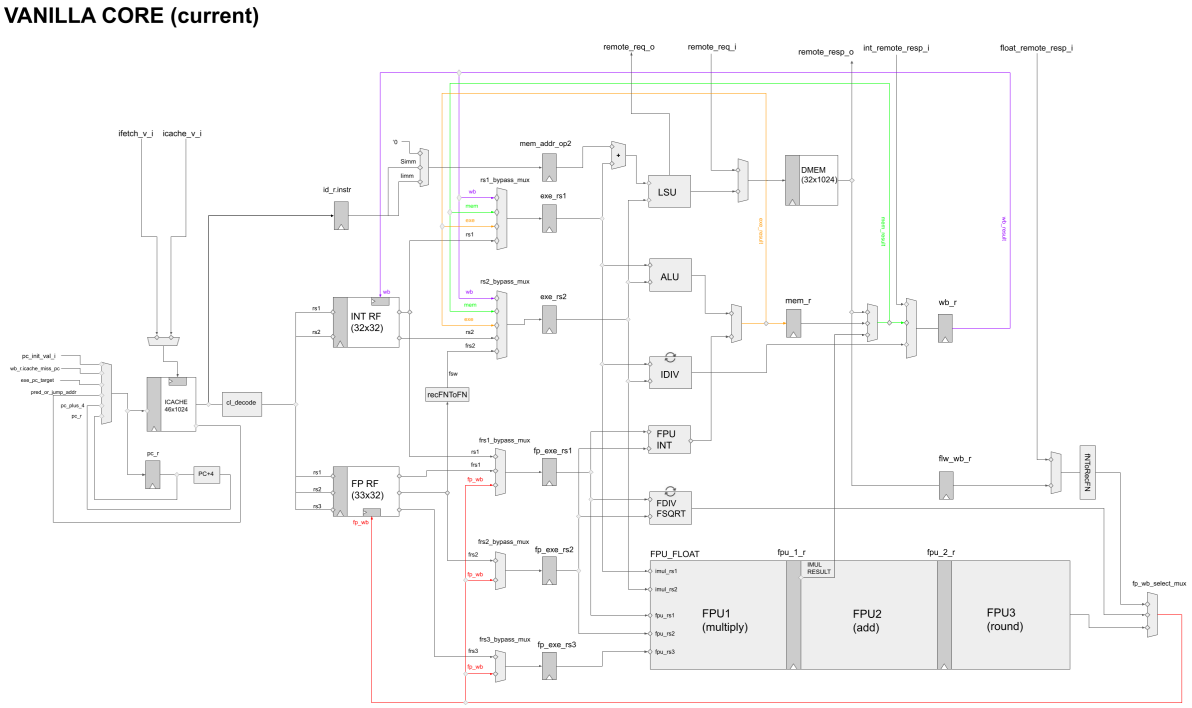

Figure 2.10: The Architecture of RISC-V Vanilla Core

2.3 Single Instruction Multiple Data

The bulk of this thesis analyzes the implementation of Single Instruction Multiple Data (SIMD) memory instructions into the aforementioned Vanilla Core. The key understanding with SIMD is that many computation heavy algorithms rely on a series of consecutive loads and stores before and after computing the workload. This is commonly a considerable percentage of the overall instructions executed by each individual tile.

2.3.1 Basics of SIMD

The basic idea behind SIMD instructions is that sequenced loads and stores in consecutive memory locations can be leveraged to convert a series load/store instructions into one larger load/store instruction. This SIMD instruction would load/store 4x the amount of data in the same amount of time a normal load/store would take to complete. In RISC-V architecture, this can require minimal hardware additions [80]. This is often done by something called a "vector unit", essentially treating the string of data as a vector and storing the entire vector at once.

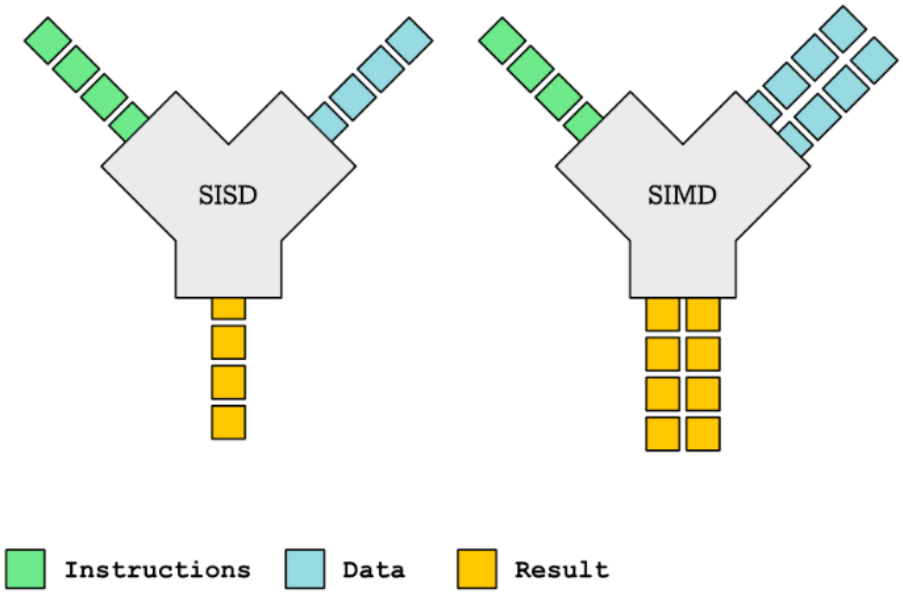

Figure 2.11: Goal of SIMD Architecture [11]

The general idea behind SIMD is that it is often desirable to have the capabilities to do the same computation on seperate sets of data in parallel.

2.3.2 RISC-V "V" Extension

There is currently a vector extension for the RISC-V ISA that has been implemented in a variety of processors. One of note is the RISC-V2 [66], a scalable RISC-V processor with a full vector processing unit (VPU) and up to 16 lanes for vector instructions. While it achieved impressive speeds, the hardware costs are incredibly high. In the case of the RISC-V2 the 16-lane Vector core required over 10x average power and 6x area[66]. This highlights the expensive hardware overhead required to fully implement the V extension for the RISC-V ISA.

Chapter 3

Implementation

3.1 Design Overview

The primary goal behind this instruction extension on the Vanilla Core is twofold. First, to increase the speed at which each individual tile can compute its individual workload. And second, to ensure that this speed does not come at the cost of power or area. With 2048 cores in the current HammerBlade design, a drastic increase in power or area could be heavily detrimental to the performance of the design. As mentioned earlier, the Vanilla Core is optimized for computing in parallel with other Vanilla Cores rather than only individual computation. The desired change is to add local SIMD float loads and local SIMD float stores. An important aspect of the proposed design is to keep the existing logic intact as much as possible, this means adding parallel buses for SIMD data and adding logic only in required locations to ensure proper data movement throughout the core.

The high-level changes to the core are as follows

- Decode (ID) Stage: replaced float register file with SIMD float register file
- Execute (EXE) Stage: added SIMD output from register file, Altered LSU to support SIMD data
- Memory (MEM) Stage: DMEM changed to 4x width, Float scoreboard changed
- Write Back (WB) Stage: Added logic to support SIMD enable signal to register file

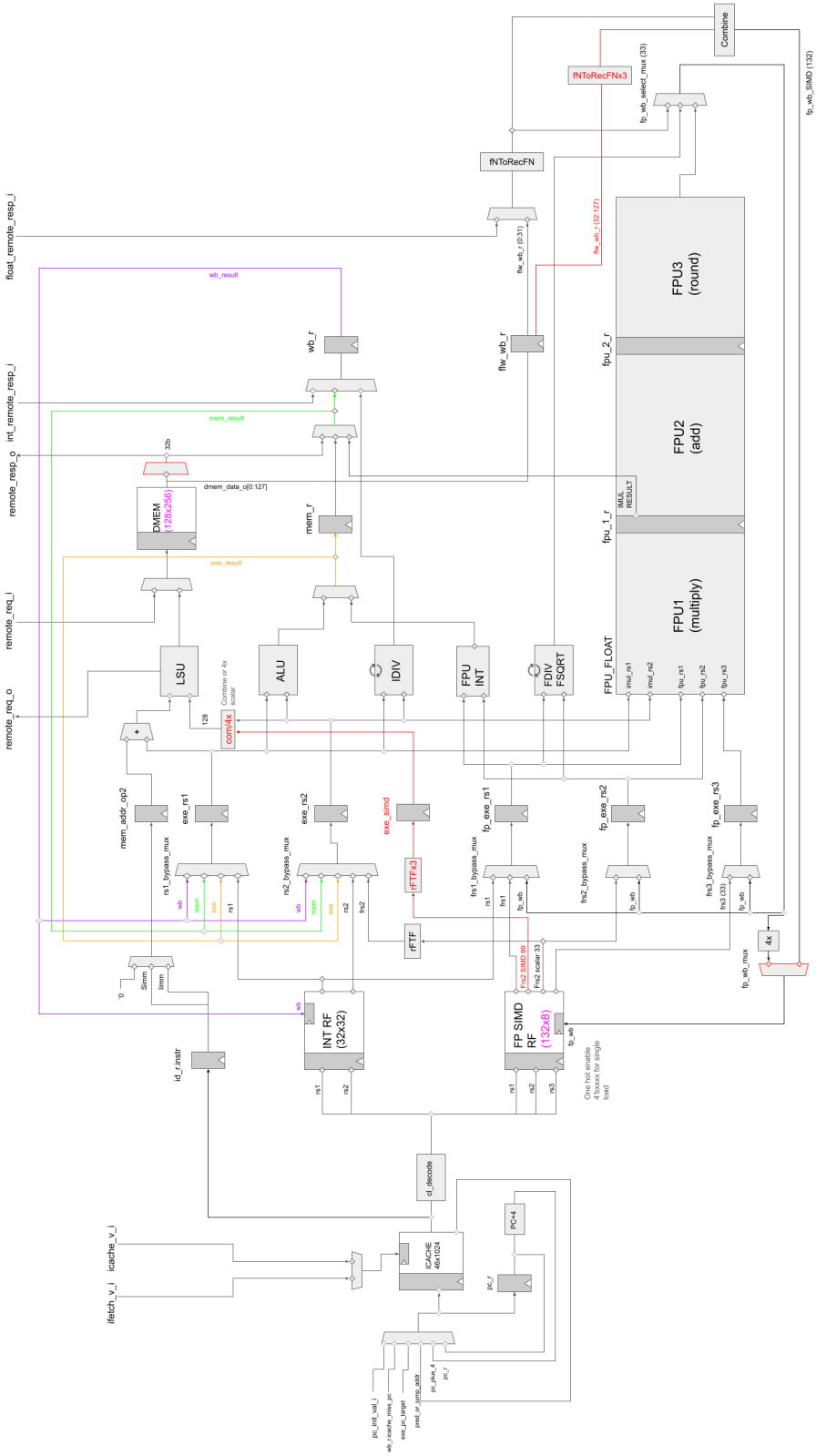

Figure 3.1: SIMD Vanilla Core (Changes in red)

3.2 Instructions and Uses

The instructions must be described in order to understand how the modifications will support the implementation. Two instructions will be added to the existing Vanilla Core: Float Load SIMD (FLS) and Float Store SIMD (FSS). The opcodes follow the instructions single word counterparts FLW and FSW respectively.

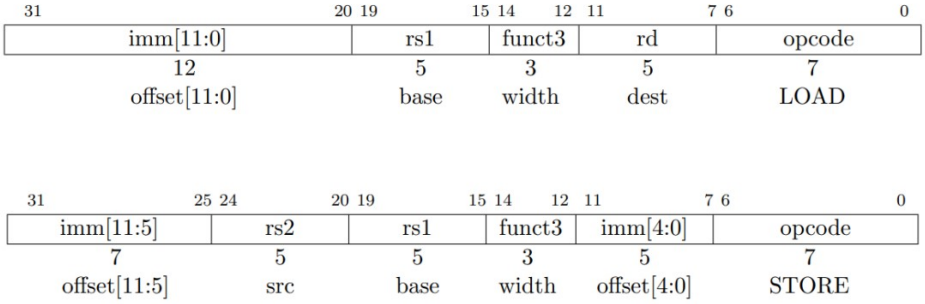

Figure 3.2: RV32 opcode structure for load and store instructions

Float Load SIMD

fls register_grp, address

The SIMD load instruction will load 4 words from local memory given by address into the register group given by register_grp.

Float Store SIMD

fss register_grp, address

The SIMD store instruction will store 4 words from the register group given by register_grp into the DMEM address given by address.

3.3 Register File

In order to support the execution of 4-word wide loads and stores, the float register file must be changed accordingly. It's important to note that in this implementation the SIMD instructions can only be called on registers that are a multiple of 4 (0,4,8,etc). The function of the register file largely stays the same, but now the SIMD data is output alongside the original read data.

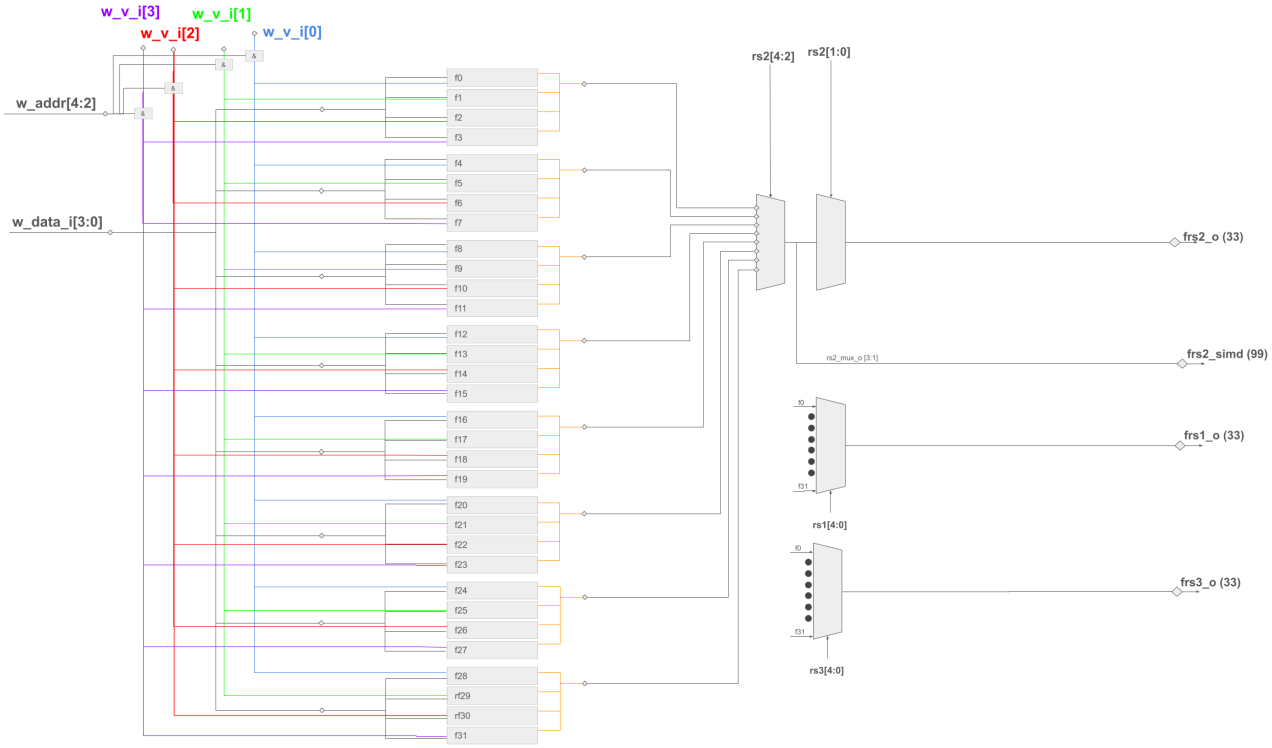

Figure 3.3: SIMD Register File Schematic

Note that the registers are grouped into groups of 4. This allows the requested data to be outputted using only the input address and the SIMD signal. This is what is a considered a banked register due to the grouping of the registers (banking). This heavily decreases the area cost by limiting unnecessary multiplexers being added.

3.3.1 Store Instruction

The semantics of the normal FSW (float store word) is as follows: when valid read(r_v_i) is turned on, the register file outputs the data from the individual register decided by the read address(r_addr_i). This data is then sent through the pipeline to the EXE stage.

The new semantics of FSS (float store SIMD) is the exact same as before, however now the registers are broken into groups of 4. When valid read is turned on, not only is the read address register outputted, but also the data from registers 2-4 of the register group which contains the read address.

When FSW is called, only the read address data is required. However when FSS is called, both the read address data and the data from that register group are required.

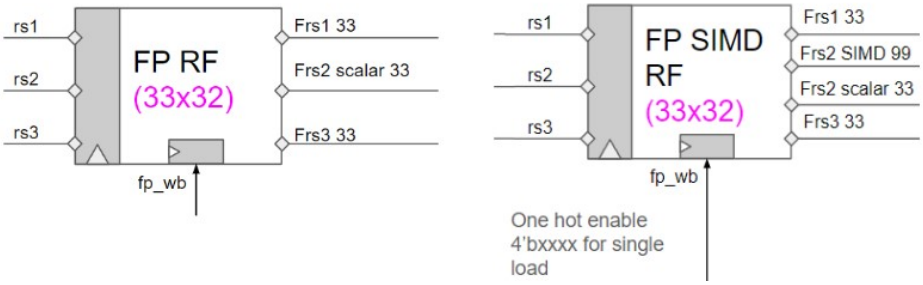

Figure 3.4: Standard vs SIMD regfile outputs

Shown below are example store instructions to further understand the SIMD register file function. The new register file SIMD output allows the existing logic to stay intact, while also enabling the SIMD data to follow in parallel to the MEM stage.

f0: data = 10	**fsw**, addr= 1 frs2 = 20 frs2_simd= 20,30,40	**fss**, read_addr = 1 Not allowed, read_addr must be mult of 4
f1: data = 20		
f2: data = 30	**fsw**, read_addr = 0 frs2 = 10 frs2_simd= 20,30,40	**fss**, read_addr = 0 frs2 = 10 frs2_simd= 20,30,40
f3: data = 40		

Figure 3.5: Store Example

3.3.2 Load Instruction

The semantics of the normal FLW (float load word) is as follows: when write valid (w_v_i_) is turned on, the data at the input port is stored into the write address (w_addr_i).

The SIMD core FLW instruction acts slightly differently. The input address corresponds to the register groups rather than the individual registers. To access the individual registers write valid (w_v_i_) is converted to a 4-bit enable signal. Each bit corresponds to the register within the group that is being written to.

The new FLS instruction acts in the exact same as the SIMD core FLW instruction, however the 4-bit enable signal is all 1's (4'b1111) meaning the entire register group is written to. Again, the SIMD instruction is only available on registers that are a multiple of 4.

3.3.3 System Verilog Module

The following code is the module used for the SIMD register file.

```
`include "bsg_defines.sv"

module simd_regfile
  #(`BSG_INV_PARAM(width_p)
    , `BSG_INV_PARAM(els_p)

    , localparam addr_width_lp=`BSG_SAFE_CLOG2(els_p)
  )
  (
    input clk_i
    , input reset_i

    , input [3:0] w_v_i //one hot write enable
    , input [addr_width_lp-1:0] w_addr_i
    , input [3:0][width_p-1:0] w_data_i //width = 132

    , input [2:0] r_v_i
    , input [2:0][addr_width_lp-1:0] r_addr_i
    , output logic [2:0][width_p-1:0] r_data_o //carries frs1,2,3
    , output logic [2:0][width_p-1:0] rs2_simd_data_o //NEW OUTPUT, width 99
  );

  wire unused = reset_i;

  logic [2:0][addr_width_lp-1:0] r_addr_r;


  always_ff @ (posedge clk_i)
    for (integer i = 0; i < 3; i++)
      if (r_v_i[i]) r_addr_r[i] <= r_addr_i[i];


  logic [3:0][width_p-1:0] mem_r [(els_p>>2)-1:0];

  assign r_data_o[0] = mem_r[r_addr_r[0][4:2]][r_addr_r[0][1:0]];
```

```
  assign r_data_o[1] = mem_r[r_addr_r[1][4:2]][r_addr_r[1][1:0]];
  assign r_data_o[2] = mem_r[r_addr_r[2][4:2]][r_addr_r[2][1:0]];

  for (genvar i = 0; i < 3; i++)
     assign rs2_simd_data_o[i] = mem_r[r_addr_r[1][4:2]][i+1];

  always_ff @ (posedge clk_i) begin
    for (integer i = 0; i < 4; i++) begin
      if (w_v_i[i]) begin
        mem_r[w_addr_i[4:2]][i] <= w_data_i[i];
      end
    end
  end

endmodule

`BSG_ABSTRACT_MODULE(simd_regfile)
```

3.4 EXE Stage

To support the SIMD data coming from the register file the EXE stage is modified to include said data in the pipeline. The load store unit (LSU) is also modified to support the SIMD data.

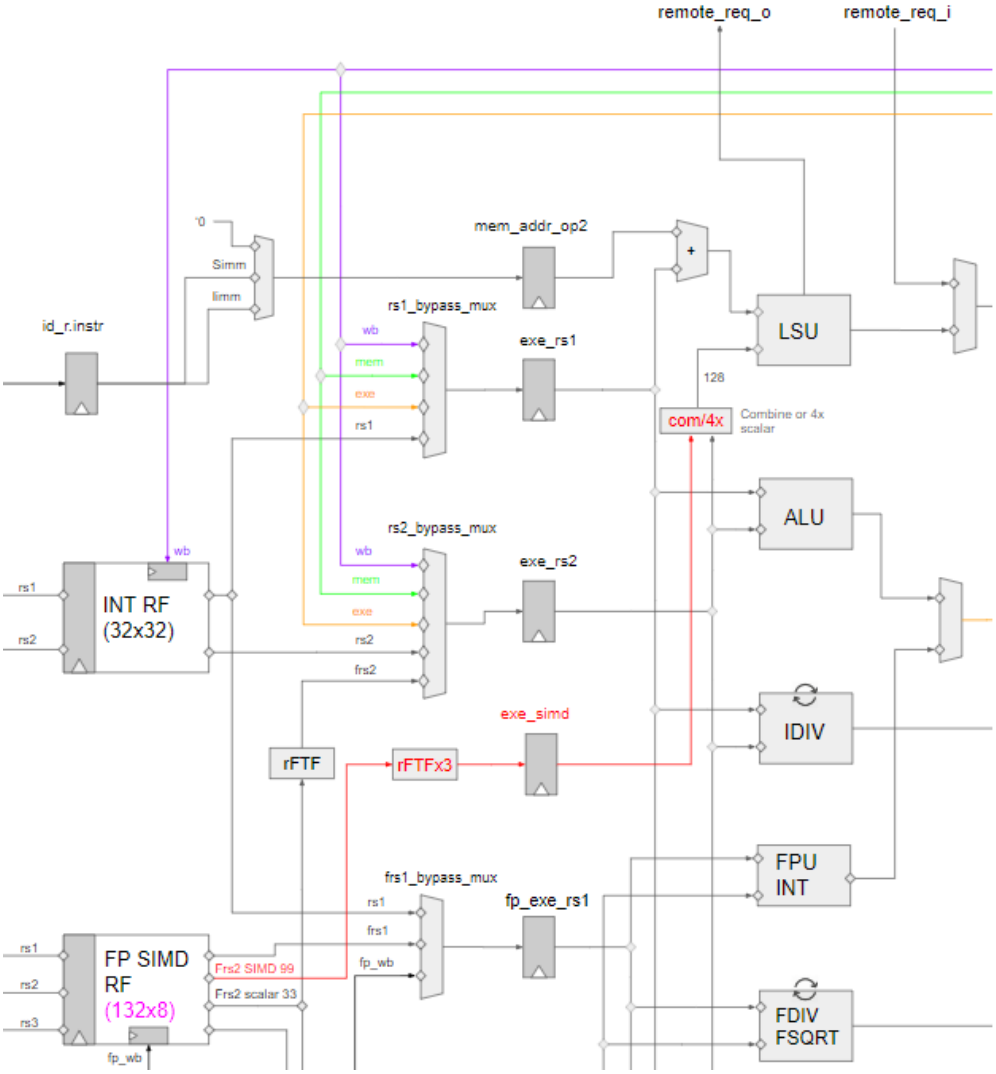

Figure 3.6: EXE Pipeline with SIMD data in red

3.4.1 Load-Store Unit

The Load-Store Unit (LSU) is a combinational logic element in the EXE stage of the Vanilla Core. It is responsible for address generation, address checking, and handshaking logic with the pipeline and network. The LSU checks each memory address and determines whether it's a local or remote address. Local instructions take precedence over remote instructions. A Local Load Buffer Figure 3.7 ensures that a remote load does not overwrite the DMEM output.

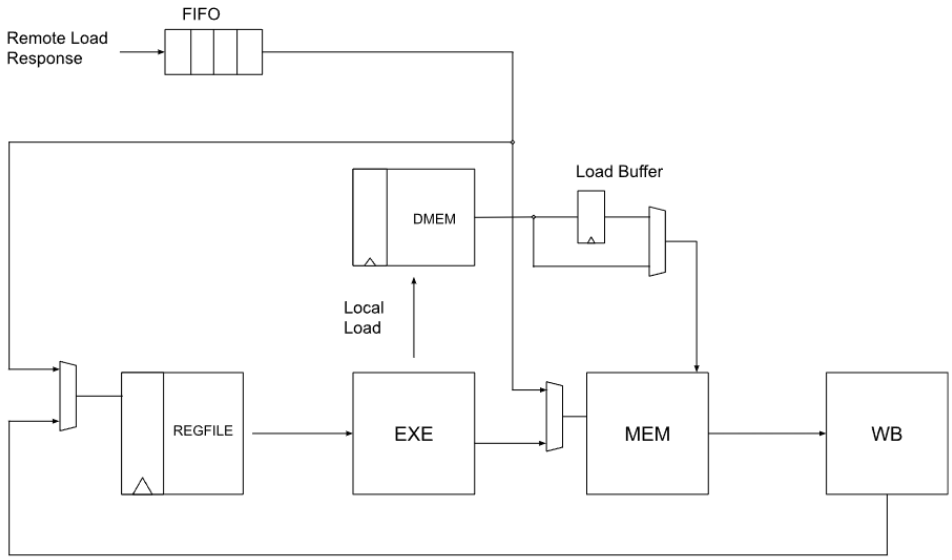

Figure 3.7: Local Load Buffer Schematic [37]

The main changes made to the LSU support the masking of the data being sent to the DMEM. This ensures that FSW instructions properly send data to the correct location, and that FSS instructions send all expected data.

The semantics of the LSU related to local loads/stores are as follows: If it is a SIMD store (FSS), the store data is set to the SIMD data and the mask is set to all 1s. For a singular word store (FSW), the store data is replicated 4x and the mask depends on the last two digits of the DMEM write address. This allows the store data to be written to the proper word-sized DMEM address using the last two bits of the address to mask the input data. Below are the SIMD related portions of the LSU that determine the data and mask being sent to the DMEM.

```
always_comb begin
    if (exe_decode_i.is_simd_op) begin
      store_simd_data = {{exe_rs2_simd_i},{store_data}};
      store_simd_mask = {{4{store_mask}}};
    end
```

```
  else begin
    store_simd_data = {{4{store_data}}};
    unique casez (dmem_addr_o[1:0])
      2'b00: begin
        store_simd_mask = {{3{4'b0000}}, {store_mask}}; //only store [0]
      end
      2'b01: begin
        store_simd_mask = {{2{4'b0000}}, {store_mask}, {1{4'b0000}}}; //only store [1] element
      end
      2'b10: begin
        store_simd_mask = {{1{4'b0000}}, {store_mask}, {2{4'b0000}}}; //only store [2] element
      end
      2'b11: begin
        store_simd_mask = {{store_mask},{3{4'b0000}}}; // only store [3] element
      end
      default: begin
        store_simd_mask = 16'b0000000000000000;
      end
    endcase
  end
end
```

3.4.2 recFNToFN Module

The recFNToFN module is the implementation of the Berkeley HardFloat module [13] to conform the data to the IEEE 2001 Verilog standard for floating points[71]. The data is stored in the float register file according to the IEEE standard for Floating-Point Arithmetic and must be converted before it is sent to DMEM. This is one of the first major downsides to the SIMD extension, as to ensure a SIMD store can occur in one cycle all the applicable data must be converted which requires 4 recFNToFN modules compared to the previous one.

3.5 MEM Stage

The MEM stage was largely kept the same but the DMEM needed to be altered to support four wide loads. Similarly, the dependencies on the float scoreboard needed to be updated to ensure accurate SIMD stores.

3.5.1 DMEM

The DMEM is altered to be 4x wider and have 4x less depth in order to support four wide loads and stores while also staying at a size of 4KB. To ensure that single wide loads are properly dealt with, a mux is required at the output of the DMEM for the desired single word. The key to understand here is that the DMEM address sent throughout the core is 9 bits, but only 7 are required in the DMEM module itself. This is due to the fact that the address width is essentially abstracted away from the pipeline, and the remaining last two bits of the original DMEM address are instead used on an output mux.

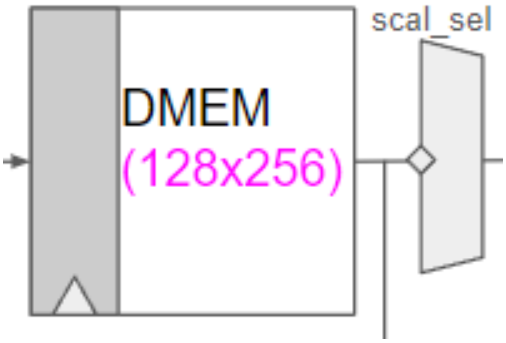

Figure 3.8: DMEM Ouput Mux

3.5.2 Float Scoreboard

The scoreboard is a tool the core uses to ensure that there are registers that are neither reserved nor dependent. Oftentimes, SIMD stores are done after a series of remote loads which take more time to complete than local loads. Currently, a register address is "scored" when there is a remote load placed to that location. Subsequently, a store is not allowed from that register until the score has been cleared. The SIMD stores, unlike the single word stores, are reliant on 4 registers being clear which is reflected in the code below taken from the SIMD core's scoreboard module.

```
always_comb begin
   if(is_simd_op) begin
     rs_depend_on_sb_simd = (scoreboard_r[src_id_i[1] + 1] & op_reads_rf_i[1]) |
         (scoreboard_r[src_id_i[1] + 2] & op_reads_rf_i[1]) | (scoreboard_r[src_id_i[1] + 3] &
         op_reads_rf_i[1]);
      end
   else begin
     rs_depend_on_sb_simd = 1'b0;
   end
 end
```

3.6 WB Stage

The strategy of the WB stage is largely similar to the EXE stage. The pipeline is altered to support the SIMD data, and 3 fNToRecFN must be added to convert the data to the floating point standard before it is stored.

3.6.1 Pipeline changes

To ensure that the data is stored properly to the float register file the original write enable must be converted to a 4-bit one-hot enable signal. The enable signal is 4'b1111 if it is a SIMD operation, or the one-hot encoded float register file write address. Below is the write back logic taken from the Vanilla Core System Verilog file.

```
always_comb begin
  float_rf_wen_li = 4'b0;
  float_rf_wdata_li = '0;

  if (flw_wb_ctrl_r.is_simd_op & ~float_remote_load_resp_v_i) begin
    float_rf_wdata_li = flw_recoded_data;
  end
  else begin
    float_rf_wdata_li = {4{float_rf_wdata}};
  end


  if (float_rf_wen & flw_wb_ctrl_r.is_simd_op & ~float_remote_load_resp_v_i) begin
    float_rf_wen_li = 4'b1111;
  end

  else if (float_rf_wen) begin
    unique casez (float_rf_waddr[1:0])
      2'b00: begin
        float_rf_wen_li = 4'b0001; //0001
      end
      2'b01: begin
        float_rf_wen_li = 4'b0010; //0010
      end
      2'b10: begin
        float_rf_wen_li = 4'b0100; //0100
      end
      2'b11: begin
        float_rf_wen_li = 4'b1000; //0001
      end
      endcase
    end
end
```

Chapter 4

Verification and Benchmarks

A large part of this thesis was ensuring that the SIMD Vanilla Core was properly executing both single word load/stores and SIMD load/stores. The work would be largely uninteresting without speedup analysis on common benchmarks comparing the SIMD Vanilla Core to the standard Vanilla Core.

4.1 Verification

The verification done on the SIMD Vanilla Core was done using the open source bsg_manycore repository[18] owned and maintained by the Bespoke Silicon Group at University of Washington. The verification suite is organized using a tiered system where levels 0 and 1 are verifying the execution of single instructions while levels 2 and 3 implement HammerBlade specific instruction loops (i.e remote loads) and the coremark benchmark suite[29]. To fully verify the core, all levels must be successfully completed.

4.1.1 Basic Verification

To verify successful execution of that standard Vanilla Core instruction set, the core was verified using all 3 levels of the bsg_manycore verification suite. The Vanilla Core uses a SPMD program loader to get relevant packets from the core to ensure it reaches the correct states (BSG_FINISH packets). This packet also gives simulation time stamps to ensure that there is no performance degradation between the SIMD and standard Vanilla Core.

To ensure that the core was not losing performance, the VCS simulation report was compared between the SIMD and standard Vanilla Core. Below is the bsg_barrier VCS simulation report compared between the SIMD and standard Vanilla Core.

	Standard Core	SIMD Core
Runtime(ps)	580841000	580841000

Table 4.1: bsg_barrier simulation results

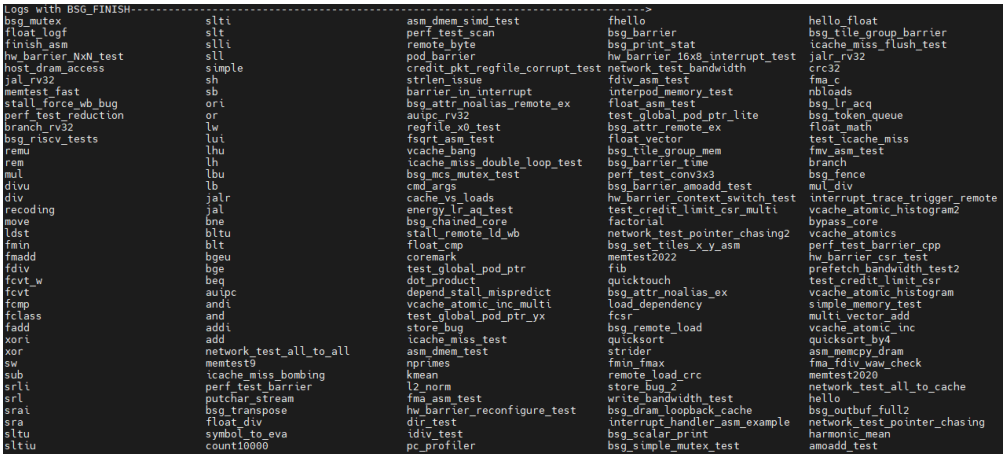

Figure 4.1: All Successfully Completed SPMD Programs

4.1.2 Vanilla Core Compiler

The HammerBlade architecture makes use of the Gnu Compiler Collection (GCC)[79], an open source RISC-V compiler. GCC is responsible for turning C/C++ code into the assembly that runs directly on the core itself. To support the SIMD instructions the GNU-GCC toolchain must be altered to include SIMD load (FLS) and SIMD store (FSS). This is done by creating the associated match and mask of FLS and FSS instructions, and adding them to the toolchain. I used Hadi R. Sandid's tutorial regarding this topic to successfully add the instructions to the toolchain[78]. The SIMD instructions follow the same format as the standard FLW and FSW instructions and are largely the same outside of changed control logic. Below is the related code to the declaration of the FLS and FSS instructions and related FLW/FSW instructions.

```
`define RV32_FLW_S `RV32_Itype(`RV32_LOAD_FP, 3'b010)
`define RV32_FSW_S `RV32_Stype(`RV32_STORE_FP, 3'b010)

`define RV32_FLS_S `RV32_Itype(`RV32_LOAD_FP, 3'b101)
`define RV32_FSS_S `RV32_Stype(`RV32_STORE_FP, 3'b101)
```

4.1.3 Current Compiler Limitations

There are a few important caveats to understand with the addition of the FLS and FSS instructions to the Vanilla Core. First, the instructions can only be called on register addresses that are multiples of four. Second, the desired data must be in consecutive register addresses. As GCC converts the C/C++ code to assembly, it allocates registers accordingly; however it does not consider the use of consecutive registers. This eliminates the possibility of SIMD instructions being properly compiled by the current compiler[1], so a different approach to verifying the SIMD instructions must be taken.

[1]There is currently a concerted effort by members of The Bespoke Silicon Group to alter the HammerBlade compiler to support the SIMD hardware extension however it was not complete at this time[9]

4.1.4 SIMD Verification

Because of the current compilers limitations, the standard C/C++ code cannot be used to properly simulate the Vanilla Core. Instead, assembly code is used to send SIMD instructions to the Vanilla Core. Two of the bsg_manycore SPMD verification benchmarks were chosen to verify proper SIMD instruction execution: asm_dmem_test and asm_memcpy_dram. Both benchmarks ultimately do the same thing, which is write 1024 pieces of data to memory and then read said data. The difference being that asm_dmem_test writes to the local 4KB DMEM (exclusively local load/stores), while asm_memcpy_dram writes to the modeled off-chip DRAM (local and remote load/stores).

4.2 Benchmarking

Once verification of the SIMD Vanilla Core was complete, the next step was to ensure that the speedup provided outweighs the costs of the hardware changes. There were three benchmarks used to evaluate the speedup of the SIMD core.

- **asm_dmem_test**: SPMD program that copies 1024 pieces of data to and from local DMEM
- **asm_memcpy_dram**: SPMD program that copies 1024 pieces of data to and from remote off-chip DRAM
- **sgemm**: Single precision general matrix multiplication running on the HammerBlade benchmark suite [17]

4.2.1 asm_dmem_test

This benchmark was altered by unrolling the write and read loops by a factor of 4, and replacing the SW and LW instructions with the float SIMD alternatives. This required a moderate increase in overhead due to the fact the standard loop was using integer loads, while the SIMD loop uses float loads requiring conversion from integers to floating point. Below is the write and read loop code from the standard and SIMD program.

Listing 4.1: Standard loop

```
dmem_to_dram_loop:
 // write to DMEM
li t0, 0
li t1, 0
li t2, N
write_loop:
  sw t1, 0(t0)
  addi t1, t1, 1
  addi t0, t0, 4
  bne t1, t2, write_loop

// read from DMEM
li t0, 0
li t1, 0
li t2, N
read_loop:
  lw t3, 0(t0)
  bne t3, t1, fail
  addi t1, t1, 1
  addi t0, t0, 4
  bne t1, t2, read_loop
```

Listing 4.2: SIMD loop

```
li t0, 0
li t1, 0
addi t2, t1, 1
addi t3, t2, 1
addi t4, t3, 1
li t5, N
init_loop:
  fcvt.s.w f0, t1 // = 0
  fcvt.s.w f1, t2 // = 1
  fcvt.s.w f2, t3 // = 2
  fcvt.s.w f3, t4 // = 3

  fss f0, 0(t0)

  addi t1, t1, 4
  addi t2, t2, 4
  addi t3, t3, 4
  addi t4, t4, 4

  addi t0, t0, 16

  bne t1, t5, init_loop

// read from DMEM
li t0, 0
li t1, 0
li t2, N
read_loop:
  fls f0, 0(t0)
  fcvt.w.s t3, f0
  bne t3, t1, fail
  addi t1, t1, 4
  addi t0, t0, 16
  bne t1, t2, read_loop
```

This code modification using the new instructions allows the 4 word-sized load/store to be replaced with a singular SIMD load/store. The speedup via VCS Simulation report is seen below.

Standard Core	SIMD Core	Speedup
10781000ps	5420000ps	1.989

Table 4.2: asm_dmem_test report

The calculations seen below approximate the change in loop instructions in the SIMD core. Via these calculations a speedup of 2.117 is seen, in-line with what is expected.

SIMD:

$$256 \text{ loops} \times 11 \text{ instructions} + 256 \text{ loops} \times 6 \text{ instructions} = 4352 \text{ total loop instructions} \quad (4.1)$$

Standard:

$$1024 \text{ loops} \times 4 \text{ instructions} + 1024 \text{ loops} \times 5 \text{ instructions} = 9216 \text{ total loop instructions} \quad (4.2)$$

4.2.2 asm_memcpy_test

With asm_dmem_test, the reads and writes are from DMEM, and as a result, asm_dmem_test can use SIMD operations. Conversely, asm_memcpy_test also reads and writes from DRAM and in these cases SIMD operations cannot be used. Read and writes from DRAM are what the Vanilla Core considers a "remote" operation and it receives the data from the network rather than the local scratchpad DMEM. Below is an example of converting the dmem_to_dram loop in asm_memcpy_dram where 4 FLW instructions can be replaced with a single FLS instruction. Note that the subsequent 32 fsw instructions are not viable for SIMD due to the fact they are storing to off-chip DRAM.

Listing 4.3: Standard loop

```
dmem_to_dram_loop:
  flw f0, 0(t0)
  flw f1, 4(t0)
  flw f2, 8(t0)
  flw f3, 12(t0)
  flw f4, 16(t0)
  flw f5, 20(t0)
  flw f6, 24(t0)
  flw f7, 28(t0)
  flw f8, 32(t0)
  flw f9, 36(t0)
  flw f10, 40(t0)
  flw f11, 44(t0)
  flw f12, 48(t0)
  flw f13, 52(t0)
  flw f14, 56(t0)
  flw f15, 60(t0)
  flw f16, 64(t0)
  flw f17, 68(t0)
  flw f18, 72(t0)
  flw f19, 76(t0)
  flw f20, 80(t0)
  flw f21, 84(t0)
  flw f22, 88(t0)
  flw f23, 92(t0)
  flw f24, 96(t0)
  flw f25, 100(t0)
  flw f26, 104(t0)
  flw f27, 108(t0)
  flw f28, 112(t0)
  flw f29, 116(t0)
  flw f30, 120(t0)
  flw f31, 124(t0)
  fsw f0, 0(t1)
  fsw f1, 4(t1)
  fsw f2, 8(t1)
  fsw f3, 12(t1)
  fsw f4, 16(t1)
  fsw f5, 20(t1)
  fsw f6, 24(t1)
  fsw f7, 28(t1)
  fsw f8, 32(t1)
  fsw f9, 36(t1)
```

Listing 4.4: SIMD loop

```
dmem_to_dram_loop_SIMD:
fls f0, 0(t0)
fls f4, 16(t0)
fls f8, 32(t0)
fls f12, 48(t0)
fls f16, 64(t0)
fls f20, 80(t0)
fls f24, 96(t0)
fls f28, 112(t0)
fsw f0, 0(t1)
fsw f1, 4(t1)
fsw f2, 8(t1)
fsw f3, 12(t1)
fsw f4, 16(t1)
fsw f5, 20(t1)
fsw f6, 24(t1)
fsw f7, 28(t1)
fsw f8, 32(t1)
fsw f9, 36(t1)
fsw f10, 40(t1)
fsw f11, 44(t1)
fsw f12, 48(t1)
fsw f13, 52(t1)
fsw f14, 56(t1)
fsw f15, 60(t1)
fsw f16, 64(t1)
fsw f17, 68(t1)
fsw f18, 72(t1)
fsw f19, 76(t1)
fsw f20, 80(t1)
fsw f21, 84(t1)
fsw f22, 88(t1)
fsw f23, 92(t1)
fsw f24, 96(t1)
fsw f25, 100(t1)
fsw f26, 104(t1)
fsw f27, 108(t1)
fsw f28, 112(t1)
fsw f29, 116(t1)
fsw f30, 120(t1)
fsw f31, 124(t1)
```

Listing 4.5: Standard loop cont.

```
fsw f10, 40(t1)
fsw f11, 44(t1)
fsw f12, 48(t1)
fsw f13, 52(t1)
fsw f14, 56(t1)
fsw f15, 60(t1)
fsw f16, 64(t1)
fsw f17, 68(t1)
fsw f18, 72(t1)
fsw f19, 76(t1)
fsw f20, 80(t1)
fsw f21, 84(t1)
fsw f22, 88(t1)
fsw f23, 92(t1)
fsw f24, 96(t1)
fsw f25, 100(t1)
fsw f26, 104(t1)
fsw f27, 108(t1)
fsw f28, 112(t1)
fsw f29, 116(t1)
fsw f30, 120(t1)
fsw f31, 124(t1)
```

In asm_memcpy_dram there are 19798 total instructions and 2048 of them are local loads/-store that are converted to SIMD load/store instructions. The 2048 FLW/FSW instructions are converted into 512 FLS/FSS instructions. Using Amdahl's law and assuming the speedup is 4x, the expected speedup is 1.081. However the table below shows a different story.

Standard Core	SIMD Core	Speedup
17603000ps	16493000ps	1.06730127933

Table 4.3: asm_memcpy_dram results

Almost 2% of the overall speedup is missing. The Vanilla Core's handling of remote loads must be further understood to find the unexpected slowdown. The semantics of a remote load is that the core sends out a remote request, and "scores" the desired register address of remote load data. The SIMD Vanilla Core supports non-blocking loads but incoming remote loads are not promised to be in order via the on-chip response network. Consequently, there are situations where SIMD store instructions are waiting for an out of order remote load. This will stall the upcoming SIMD store instruction until the remote load is completed. Figure 4.2 shows a sequence of 8 SIMD stores after a series of remote loads. The float dependency wire indicates that for the next cycle the core must stall due to the fact the data to be stored is not ready yet via remote load.

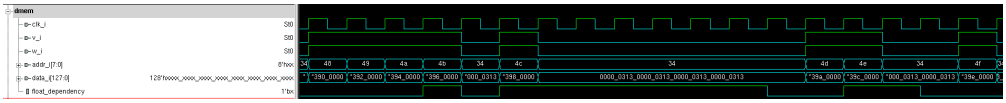

Figure 4.2: Remote Load Delay for SIMD Store

The added stalls incurred by remote loads means that the speedup of SIMD stores after remote loads is 2x rather than 4x (and can be even lower depending on order of remote loads). The average speedup of SIMD instructions will now be 3x due to the fact SIMD loads are 4x and SIMD stores are 2x. Using these values, Amdahl's law results in a speedup of 1.0714 which is in line with the speedup measured.

4.2.3 SGEMM

The Single precision General Matrix Multiplication (SGEMM) is a common benchmark that is implemented in the hb_hammerbench[46] suite by The Bespoke Silicon Group. It simulates HammerBlade architecture using the CUDA-lite[100] programming model previously described. This benchmark relies heavily on local FLW and FSW instructions making it an attractive recipient of a SIMD makeover. SGEMM can be ran using a variety of settings, but all benchmarking was done using SGEMM 512,2 (512x512 matrix multiplication, 2 passes).

Instruction	Standard Core	SIMD Core
local flw	151519232	67108864
local fsw	50331648	0
local fls	0	21102592
local fss	0	12582912

Table 4.4: Instruction count for SGEMM

One difficulty with the SGEMM benchmark (and any hb_hammerbench benchmark) is that the CUDA-lite model is explicitly written in C/C++. While it was trivial to alter the assembly (asm) benchmarks, the CUDA-lite code required a separate avenue of attack. Due to the current compilers limitations within the current HammerBlade toolchain, inline assembly would be troublesome with how registers are allocated. The most efficient way to add SIMD instructions to this benchmark is by object dumping the existing kernel (code that runs on each tile) and rewriting portions of it in separate assembly functions and linking the functions in C. Below is an example of a rewritten portion of the SGEMM Kernel.

kernel.cpp:

```
// load the psum
    bsg_unroll(SUB_DIM)
    for (int py = 0; py < SUB_DIM; py++) {
      bsg_unroll(SUB_DIM)
      for (int px = 0; px < SUB_DIM; px++) {
        psum[py][px] = sub_block_out[(py*BLOCK_DIM)+px];
      }
    }
```

SIMDkernel.cpp:

```
// load the psum
load_psum_s(sub_block_out);
```

The disassembly from these code snippets is as follows:

kernel.dis:

```
        psum[py][px] = sub_block_out[(py*BLOCK_DIM)+px];
    c4c:       0007a687              flw    fa3,0(a5) # 100000 <_bsg_elf_vcache_size+0x80000>
    c50:       0047a787              flw    fa5,4(a5)
    c54:       0087a607              flw    fa2,8(a5)
    c58:       00c7a707              flw    fa4,12(a5)
    c5c:       0407a587              flw    fa1,64(a5)
    c60:       0447a507              flw    fa0,68(a5)
    c64:       0487a007              flw    ft0,72(a5)
    c68:       04c7a087              flw    ft1,76(a5)
    c6c:       0807a887              flw    fa7,128(a5)
    c70:       0847ae07              flw    ft8,132(a5)
    c74:       0887ae87              flw    ft9,136(a5)
    c78:       08c7af07              flw    ft10,140(a5)
    c7c:       0c07af87              flw    ft11,192(a5)
    c80:       0c47a407              flw    fs0,196(a5)
    c84:       0c87a307              flw    ft6,200(a5)
    c88:       0cc7a807              flw    fa6,204(a5)
```

SIMDkernel.dis:

```
// load the psum

00000a8c <load_psum_s>:
#include "bsg_manycore_asm.h"
.text
.global load_psum_s
load_psum_s:
   fls f0, 0(a0)
 a8c:   00055007              fls    ft0,0(a0)
   fls f4, 64(a0)
 a90:   04055207              fls    ft4,64(a0)
   fls f8, 128(a0)
 a94:   08055407              fls    fs0,128(a0)
   fls f12, 192(a0)
 a98:   0c055607              fls    fa2,192(a0)
```

There are 5 separate linked assembly functions in the SIMD SGEMM kernel, allowing 84410368 FLW instructions to be converted into 21102592 FLS instructions, and 50331648 FSW instructions into 12582912 FSS instructions. This accounts for 23.53% of all instructions. Using Amdahl's law and understanding that not all of the 4x speedup will be experienced from stores following remote loads, a speedup of 1.13 is achieved.

Standard Core	SIMD Core	Speedup
6613918 cycles	6337718 cycles	1.04358035495

Table 4.5: SGEMM Speedup

In the core utilization statistics Table 4.6 there is suddenly an increase in ADDI, JAL, and JALR instructions. This is because when the assembly functions are called, the compiler must first calculate each operand, then JAL to the function location embedded into the assembly, and finally JALR back to the code from the function call. This accounts for 7.75% lost speedup. If this "lost" speedup is added to the simulation speedup the speedup is 1.12, which is what is expected.

Instruction	SIMD Count	SIMD Percent	Standard Count	Standard Percent
local flw	67108864	13.10%	151519232	26.47 %
local fsw	0	0.00 %	50331648	8.79 %
local fls	21102592	4.12%	0	0.00 %
local fss	12582912	2.46%	0	0.00 %
addi	69826664	13.63%	55443879	9.68 %
jal	11550720	2.25%	0	0.00 %
jalr	11550720	2.25%	0	0.00 %
fmadd	268435456	52.39%	268435456	46.89 %
local ld	682791	0.13%	18113	0.00 %
local st	690345	0.13%	530476	0.09 %
bne	10862434	2.12%	10862420	1.90 %
blt	1043	0.00%	1048	0.00 %
bge	3072	0.00%	3072	0.00 %
slli	1050149	0.20%	2501	0.00 %
add	2256511	0.44%	1271473	0.22 %
sub	1158	0.00%	2048	0.00 %
lui	595265	0.12%	7242	0.00 %
or	384	0.00%	384	0.00 %
and	384	0.00%	384	0.00 %
fence	128	0.00%	128	0.00 %
barsend	128	0.00%	128	0.00 %
barrecv	128	0.00%	128	0.00 %
remote st global	128	0.00%	128	0.00 %
remote flw dram	2097152	0.41%	2097152	0.37 %
remote seq flw dram	31457280	6.14%	31457280	5.49 %
remote fsw dram	524288	0.10%	524288	0.09 %

Table 4.6: SGEMM Core Utilization

Chapter 5

Cost Analysis

To fully understand the impact of the Vanilla Core proposed modifications, the power and energy characteristics must also be analyzed. As seen in the Tile Floor Plan in TSMC 16nm 2.7 the IMEM, DMEM, and register file take up 62% of the overall tile area[15]. Consequently, the IMEM and DMEM will be the primary focus regardings the area and power changes.

5.1 Area

Increasing the DMEM width while decreasing depth creates a 19.8% increase in overall DMEM Area [38]. Considering that the DMEM is one of the largest contributors to overall tile area, this is somewhat undesirable. Not only does the overall area increase, but the increased width causes the Y length to be almost four times longer than the X length which will be a limiter, especially if looking to optimize the tile for area.

Using a banked SIMD DMEM system (two DMEM units acting as one) requires more area than both the single-unit DMEM and standard DMEM, however it gives us further power optimizations.

	SIMD DMEM depth:256, width:128	Standard DMEM depth:1024, width:32	Banked DMEM depth:256, width:64
X length (microns)	36.377	53.045	72.754
Y length (microns)	143.616	82.176	82.176
Area (microns^2)	5224.319232	4359.02592	5978.632704

Table 5.1: DMEM Area

5.2 Power

The second aspect of the cost analysis is to understand the change in power after implementing SIMD modifications. To do this two primary power sources will be considered: The IMEM and DMEM. The IMEM is the 4KB instruction cache and the DMEM is the local 4KB scratchpad used for all local loads/stores.

	SIMD DMEM	Standard DMEM	IMEM
Read Power(mA)	4.11E-03	2.21E-03	2.83E-03
Write Power(mA)	5.72E-03	2.82E-03	3.58E-03
Read Power per bit(mA)	3.21E-05	6.92E-05	n/a

Table 5.2: DMEM & IMEM read/write power

While the overall power per read and write is significantly higher in the SIMD DMEM, it is actually more efficient in reading per bit. Both the IMEM and DMEM are static random-access memory (SRAM). The SRAM energy in the HammerBlade tile takes about 35% of the total energy to execute one scalar load/store[56] which implies that a decrease or increase in power will heavily affect the power characteristics of a full HammerBlade array. Given this knowledge, there will be a "breakeven point" where if a program can convert a percentage of its load/stores into SIMD loads/stores it will save energy rather than use more energy.

To accurately calculate this the total power used by single-word load/stores and SIMD load/stores must be understood. This is done by summing up the read and write power from IMEM, and read power for loads or write power for stores. Shown in Table 5.3 are the total power values for individual load/stores and 4x load/stores.

	SIMD Core	Standard Core
FLW(mA)	10.52E-03	8.62E-03
FSW(mA)	12.13E-03	9.23E-03
4xFLW(FLS) (mA)	10.52E-03	34.48E-03
4xFSW(FSS) (mA)	12.13E-03	36.92E-03

Table 5.3: Instruction Power from IMEM DMEM

Using these values it can be computed that as long as 24.1% of load instructions and 31.9% of store instructions are able to be converted to SIMD, power will reach net zero. Note that these are load/store exclusive, meaning that if 24.1% of load instructions are converted that would cover the power cost of all remaining load instructions to a net zero; the same is true for stores. The related code for calculating this can be found in the appendix. Using SGEMM 512 as an example, out of the 201850886 total load/store instructions 134742022 were able to be converted to SIMD instructions-66.75%. Using the data from the Table 5.3, 6.9E05 mA of power is saved.

5.2.1 Banked DMEM alternative

To decrease the cost of single word load and store instructions, the SIMD DMEM can be "banked". The DMEM is split into two 64 wide 256 deep identical DMEM units rather than the single 128 wide 256 deep DMEM unit. This is an alternative approach to the existing SIMD DMEM and is not supported by the current Vanilla Core SIMD modifications.

	SIMD DMEM	Standard DMEM	Banked DMEM
Read Power(mA)	4.11E-03	2.21E-03	2.32E-03
Write Power(mA)	5.72E-03	2.82E-03	3.12E-03
Read Power per bit(mA)	3.21E-055	6.92E-05	3.63E-05

Table 5.4: Various DMEM configuration read/write power

The banked DMEM would only turn on the respective unit that contains the location required for a single word load/store. For SIMD instructions, both DMEM units would be turned on. As seen in Table 5.4, our single word instructions would incur a power cost that is significantly lower than the larger single-unit SIMD DMEM. However, the power cost of SIMD instructions would be substantially larger than the single-unit SIMD DMEM due to the need to power on both DMEM units.

	SIMD Core	Standard Core	Banked DMEM Core
FLW(mA)	10.52E-03	8.62E-03	8.73E-03
FSW(mA)	12.13E-03	9.23E-03	9.53E-03
4xFLW(FLS) (mA)	10.52E-03	34.48E-03	17.46E-03
4xFSW(FSS) (mA)	12.13E-03	36.92E-03	19.06E-03

Table 5.5: Instruction Power from IMEM DMEM

Using a banked DMEM, we would only need to convert 6.3% of load instructions and 2.6% of store instructions to reach our power breakpoint. In SGEMM 512, the banked DMEM would save 5.76514E05 mA of power. This is less power saved than with the single-unit SIMD DMEM but is more robust against increased power consumption from single word instructions. The two main downsides to using a banked DMEM are that the DMEM address logic will need to be changed to support SIMD load/stores accessing both units and that there is a slight area increase compared to the single-unit SIMD DMEM (5978 and 5224 microns^2 respectively).

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The HammerBlade manycore architecture relies on high-throughput workhorse compute tiles, primarily via the RISC-V Vanilla Core. This thesis proposes a low-weight extension to the Vanilla Core that can not only improve the throughput of the SIMD Core compute tile but also decrease its power profile. First, the hardware was extended to support two instructions: FLS and FSS. This was done by creating parallel lanes along the standard FLW and FSW instructions and required control logic to support the widened data bus. Next, GCC was altered to allow for the use of the FLS and FSS instructions through custom assembly files. Multiple SPMD and HammerBlade benchmarks were altered and analyzed to understand the performance gained by the SIMD extension. Lastly, the power and area of the proposed changes were analyzed to further understand the cost of this modification.

This modification adds increased throughput for applications with high data-level parallelism with the potential for large power savings depending on the percentage of the application that can be converted to SIMD instructions. The primary drawback of this design is the increased area. With that in mind this SIMD core may not become the standard compute tile for the HammerBlade architecture, but can instead blend in seamlessly as a "soft" accelerator depending on the intended use of the HammerBlade system.

6.2 Future Work

While largely successful, there are a number of directions this work can go in the future. Listed below are a few ways that this work can be furthered and refined.

- **SIMD core area and switching power**: One largely unexplored area of this thesis is the area and power cost of the proposed changes outside of the SRAM. This may impact the efficacy of the modified SIMD design
- **Read-enable SRAM**: To further reduce power, different portions of the SRAM could be powered on and off depending on the type of instruction being executed on SRAM.
- **Compiler support**: The benchmarks chosen gave promising results, but can be fully investigated once the compiler modification is complete.

- **Striding**: Most common algorithms, if not storing to adjacent memory addresses, are storing to common intervals throughout memory. With the current design this is not feasible, and likely would require multiple DMEMs on chip to support.
- **Place and route**: This design was not fully pushed through the RTL flow. Completing this would further the understanding of the power and area costs of the proposed design.
- **Vector register**: Many current SIMD processors contain a vector register that is solely responsible for vector instructions (FLS, FSS). This was not in the scope of this thesis, and I expect the area required for another vector register to outweigh the potential gain in a manycore system.

These potential modifications would further the understanding of how SIMD instructions could transform the HammerBlade Manycore architecture.

Chapter 7

Appendix: Python Code for Power Analysis

7.1 Python Code for Power Analysis

To properly calculate the ratio of instructions required to "break even" power-wise was calculated using this simple python script:

```
from decimal import Decimal

def TestValues(totalInstructions):
    totalInstructions = int(totalInstructions)
    minStore = 0
    minLoad = 0
    calcStore = -1
    calcLoad = -1
    for num in range(totalInstructions):
        if minStore == 0:
            calcStore = CalcStore(num, totalInstructions - num)
            if  calcStore > 0:
                minStore = num
        if minLoad == 0:
            calcLoad = CalcLoad(num, totalInstructions - num)
            if calcLoad > 0:
                minLoad = num

    print("-----")
    print("total instructions: " + str(totalInstructions))
    print("minimum simd stores for positive delta : " + str(format_percentage(minStore /
        totalInstructions)))
    print("minimum simd loads for positive delta : " + str(format_percentage(minLoad/
        totalInstructions)))
    print("-----")

def CalcStore(simd, normal):
    return Decimal(Decimal("5.6025") * simd - Decimal("3.45") * normal)

def CalcLoad(simd, normal):
    return Decimal(Decimal("6.1075") * simd - Decimal("3.26") * normal)

def format_percentage(decimal_number, decimal_places=2):
    percentage = "{:.{}%}".format(decimal_number, decimal_places)
    return percentage
```

```
def main():
    choice = input("enter total instructions: ")
    TestValues(choice)


if __name__ == "__main__":
    main()
```

Bibliography

[1] A. Agarwal et al. "The RAW compiler project". In: *Proceedings of the Second SUIF Compiler Workshop*. 1997, pp. 21–23.

[2] Tutu Ajayi et al. "Celerity: An Open Source RISC-V Tiered Accelerator Fabric". In: *HOTCHIPS*. Aug. 2017.

[3] Tutu Ajayi et al. "Celerity: An open source RISC-V tiered accelerator fabric". In: *Symp. on High Performance Chips (Hot Chips)*. 2017.

[4] Alric Althoff et al. "Hiding Intermittant Information Leakage with Architectural Support for Blinking". In: *International Symposium on Computer Architecture (ISCA)*. 2018.

[5] Manish Arora et al. "Reducing the Energy Cost of Irregular Code Bases in Soft Processor Systems". In: *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2011.

[6] Krste Asanovic et al. "The rocket chip generator". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016), pp. 6–2.

[7] Krste Asanović and David A Patterson. "Instruction sets should be free: The case for risc-v". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[8] Saahil Athrij. "Vectorizing the Hamerblade Compiler". MA thesis. University of Washington, 2024.

[9] Sahil Athrij. *Vectorizing Memory Access on HammerBlade Architecture*. Master Thesis. 2024.

[10] Jonathan Babb et al. "The Raw Benchmark Suite: Computation Structures for General Purpose Computing". In: *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 1997.

[11] Marcel Baracchi-Frei et al. "Real-Time GNSS Software Receiver: Challenges, Status, and Perspectives". In: *GPS World* 20.9 (2009), pp. 40–47.

[12] B. Beresini, S. Ricketts, and M.B. Taylor. "Unifying manycore and FPGA processing with the RUSH architecture". In: *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*. 2011, pp. 22–28.

[13] Berkeley. *HardFloat Release 1: Verilog Modules*. `http://www.jhauser.us/arithmetic/HardFloat-1/doc/HardFloat-Verilog.html`. 2019.

[14] Vikram Bhatt et al. " Sichrome: Mobile web browsing in Hardware to save Energy ". In: *Dark Silicon Workshop, ISCA*. 2012.

[15] Max Ruttenberg Borna Ehsani. *HammerBlade*. 2023.

[16] Ajay Brahmakshatriya et al. "Taming the Zoo: A Unified Graph Compiler Framework for Novel Architectures". In: *ISCA*. 2021.

[17] $bsg_bladerunnerrepository$. `https://github.com/bespoke-silicon-group/bsg_bladerunner`. Accessed: 2024-01-05.

[18] $bsg_manycorerepository$. `https://github.com/bespoke-silicon-group/bsg_manycore`. Accessed: 2024-01-05.

[19] Sadullah Canakci et al. "DirectFuzz: Automated Test Generation for RTL Designs using Directed Graybox Fuzzing". In: *DAC*. 2021.

[20] Yuan-Mao Chueh. "A Complete Open Source Network Stack For BlackParrot". MA thesis. University of Washington, 2022.

[21] Yuan-Mao Chueh. *CSE549 HammerBlade*. 2023.

[22] Scott Davidson Dai Cheol Jung et al. *Ruche Networks: Wire-Maximal, No-Fuss NoCs*. 2020.

[23] Frederica Darema. "The spmd model: Past, present and future". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 8th European PVM/MPI Users' Group Meeting Santorini/Thera, Greece, September 23–26, 2001 Proceedings 8*. Springer. 2001, pp. 1–1.

[24] Scott Davidson et al. "The Celerity Open-Source 511-core RISC-V Tiered Accelerator Fabric". In: *Micro, IEEE* (Mar. 2018).

[25] Ramandeep Singh Dehal et al. "Gpu computing revolution: Cuda". In: *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. IEEE. 2018, pp. 197–201.

[26] Hadi Esmaeilzadeh and Michael Bedford Taylor. "Open Source Hardware: Stone Soups and Not Stone Satues, Please". In: *SIGARCH Computer Architecture Today*. Dec. 2017.

[27] Hadi Esmaeilzadeh and Michael Bedford Taylor. "Open Source Hardware: Stone Soups and Not Stone Satues, Please". In: *SIGARCH Computer Architecture Today* (2017).

[28] Emily Furst. "Code Generation and Optimization of Graph Programs on a Manycore Architecture". PhD thesis. University of Washington, 2021.

[29] Shay Gal-On and Markus Levy. "Exploring coremark a benchmark maximizing simplicity and efficacy". In: *The Embedded Microprocessor Benchmark Consortium* (2012).

[30] S. Garcia et al. "The Kremlin Oracle for Sequential Code Parallelization". In: *Micro, IEEE* 32.4 (July 2012), pp. 42–53.

[31] Saturnino Garcia et al. "Bridging the Parallelization Gap: Automating Parallelism Discovery and Planning". In: *USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*. 2010.

[32] Saturnino Garcia et al. "Kremlin: Rethinking and Rebooting gprof for the Multicore Age". In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2011.

[33] Nathan Goulding et al. "GreenDroid: A Mobile Application Processor for a Future of Dark Silicon". In: *HOTCHIPS*. 2010.

[34] N. Goulding-Hotta et al. "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future". In: *Micro, IEEE* (Mar. 2011), pp. 86–95.

[35] Nathan Goulding-Hotta. "Specialization as a Candle in the Dark Silicon Regime". PhD thesis. University of California, San Diego, 2020.

[36] Nathan Goulding-Hotta et al. "GreenDroid: An Architecture for the Dark Silicon Age". In: *Asia and South Pacific Design Automation Conference (ASPDAC)*. 2012.

[37] Bespoke Silicon Group. *BaseJump RISC-V Vanilla Core*. `https://docs.google.com/document/d/1zni4og639nGNEIWdaTYTTcOiM3fz0aEma2l2QX5lztw`. 2020.

[38] Bespoke Silicon Group. $GF14_SRAM_DATASHEET$. 2024.

[39] Bespoke Silicon Group. *HammerBlade Manycore Overview*. `https://docs.google.com/document/d/1wpdx0FykCyIAL3VdJEBz0tK-aQyChW0TKdHfbIXQJQI`. 2020.

[40] Bespoke Silicon Group. *Vanilla Core*. `https://github.com/bespoke-silicon-group/bsg_manycore/tree/master/v/vanilla_bean`. 2024.

[41] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. "DR-SNUCA: An Energy-Scalable Dynamically Partitioned Cache". In: *International Conference on Computer Design (ICCD)*. 2013.

[42] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. "QualityTime: A Simple Online Technique for Quantifying Multicore Execution Efficiency". In: *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014.

[43] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. "Time Cube: A Manycore Embedded Processor with Interference-Agnostic Progress Tracking". In: *International Conference On Embedded Computer Systems: Architectures, Modeling And Simulation (SAMOS)*. 2013.

[44] Byron Hawkins, Brian Demsky, and Michael Bedford Taylor. "A Runtime Approach to Security and Privacy". In: *European Security and Privacy*. 2016.

[45] Byron Hawkins, Brian Demsky, and Michael Bedford Taylor. " BlackBox: Lightweight Security Monitoring for COTS Binaries". In: *Code Generation and Optimization*. 2016.

[46] $hb_hammerbenchrepository$. `https://github.com/bespoke-silicon-group/hb_hammerbench`. Accessed: 2024-04-10.

[47] Hu et al. "FPGA Global Routing Architecture Optimization Using a Multicommodity Flow Approach ". In: *ICCD*. 2007.

[48] Donghwan Jeon, Saturnino Garcia, and Michael Bedford Taylor. "Skadu: Efficient Vector Shadow Memories for Poly-scopic Program Analysis". In: *Conference on Code Generation and Optimization (CGO)*. 2013.

[49] Donghwan Jeon et al. "Kismet: Parallel Speedup Estimates for Serial Programs". In: *Conference on Object-Oriented Programming, Systems, Language and Applications (OOPSLA)*. 2011.

[50] Donghwan Jeon et al. "Kremlin: Like gprof, but for Parallelization". In: *Principles and Practice of Parallel Programming (PPoPP)*. 2011.

[51] Donghwan Jeon et al. "Parkour: Parallel Speedup Estimates from Serial Code". In: *USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*. 2011.

[52] Hongshin Jun et al. "Hbm (high bandwidth memory) dram technology and architecture". In: *2017 IEEE International Memory Workshop (IMW)*. IEEE. 2017, pp. 1–4.

[53] Dai Cheol Jung. "Caches for Complex Open Source System-on-Chip Designs". MA thesis. University of Washington, 2019.

[54] Dai Cheol Jung. "Caches for Complex Open Source System-on-Chip Designs". PhD thesis. 2019.

[55] Dai Cheol Jung et al. "Ruche Networks: Wire-Maximal, No-Fuss NoCs". In: *NOCS*. 2020.

[56] Tommy Jung. personal communication. 2024.

[57] Moein Khazraee. "Reducing the development cost of customized cloud infrastructure". PhD thesis. University of California, San Diego, 2020.

[58] Moein Khazraee et al. "Moonwalk: NRE Optimization in ASIC Clouds or, accelerators will use old silicon". In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2017.

[59] Moein Khazraee et al. "Specializing a Planet's Computation: ASIC Clouds". In: *IEEE Micro* (May 2017).

[60] Jason Kim et al. "Energy Characterization of a Tiled Architecture Processor with On-Chip Networks". In: *International Symposium on Low Power Electronics and Design (ISLPED)*. Aug. 2003.

[61] Sravanthi Kota Venkata et al. "SD-VBS: The San Diego Vision Benchmark Suite". In: *IEEE International Symposium on Workload Characterization (IISWC)*. 2009.

[62] Ikuo Magaki et al. "ASIC Clouds: Specializing the Datacenter". In: *International Symposium on Computer Architecture (ISCA)*. 2016.

[63] Sripathi Muralitharan. "TinyParrot: An Integration-Optimized Linux-Capable Host Multicore". MA thesis. University of Washington, 2021.

[64] S. Pal et al. "A 7.3 M Output Non-Zeros/J Sparse Matrix-Matrix Multiplication Accelerator using Memory Reconfiguration in 40 nm". In: *Symposium on VLSI Circuits*. 2019, pp. C150–C151.

[65] D. Park et al. “A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix–Matrix Multiplication Accelerator”. In: *IEEE Journal of Solid-State Circuits* (Apr. 2020), pp. 933–944.

[66] Kariofyllis Patsidis et al. “RISC-V 2: a scalable RISC-V vector processor”. In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2020, pp. 1–5.

[67] David Patterson and Andrew Waterman. *The RISC-V Reader: an open architecture Atlas*. Strawberry Canyon, 2017.

[68] D. Petrisko et al. “BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs”. In: *IEEE Micro* (July 2020), pp. 93–102.

[69] Daniel Petrisko et al. “NoC Symbiosis”. In: *NOCS*. 2020.

[70] MM Hafizur Rahman, Asadullah Shah, and Yasushi Inoguchi. “A deadlock-free dimension order routing for hierarchical 3d-mesh network”. In: *2012 International Conference on Computer & Information Science (ICCIS)*. Vol. 2. IEEE. 2012, pp. 563–568.

[71] Molleti Rajani and P Narayana Murty. “Verilog implementation of double precision floating point division using vedic paravartya sutra”. In: *2015 IEEE International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)*. 2015, pp. 253–256. DOI: 10.1109/ICRCICN.2015.7434245.

[72] Robert ”Max” Ramstad. “Enabling Vector Load and Store instructions on HammerBlade Architecture”. MA thesis. University of Washington, 2024.

[73] Shashank Vijaya Ranga. “ParrotPiton and ZynqParrot: FPGA Enablements for the BlackParrot RISC-V Processor”. MA thesis. University of Washington, 2021.

[74] A. Rovinski et al. “A 1.4 GHz 695 Giga Risc-V Inst/s 496-Core Manycore Processor With Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS”. In: *2019 Symposium on VLSI Circuits*. 2019, pp. C30–C31.

[75] A. Rovinski et al. “Evaluating Celerity: A 16-nm 695 Giga-RISC-V Instructions/s Manycore Processor With Synthesizable PLL”. In: *IEEE Solid-State Circuits Letters* 2.12 (2019), pp. 289–292.

[76] Jack Sampson et al. “An Evaluation of Selective Depipelining for FPGA-based Energy-Reducing Irregular Code Coprocessors”. In: *Conference on Field Programmable Logic and Applications (FPL)*. 2011.

[77] Jack Sampson et al. “Efficient Complex Operators for Irregular Codes”. In: *High Performance Computing Architecture (HPCA)*. 2011.

[78] Hadi R. Sandid. *Adding Custom Instructions to the RISC-V GNU-GCC toolchain*. https://hsandid.github.io/posts/risc-v-custom-instruction/. Accessed: 2024-04-04.

[79] Richard Stallman et al. *The GNU project*. 1998.

[80] Sarah M. Al-sudany, Ahmed S. Al-Araji, and Bassam M. Saeed. “FPGA based MIPS Pipeline Processor with SIMD Architecture”. In: 2020. URL: https://api.semanticscholar.org/CorpusID:247243867.

[81] Steven Swanson and Michael Taylor. “GreenDroid: Exploring the next evolution for smartphone application processors”. In: *IEEE Communications Magazine*. Mar. 2011.

[82] MB Taylor et al. “The Raw processor-a scalable 32-bit fabric for embedded and general purpose computing”. In: *Proceedings of Hot Chips XIII*. 2001.

[83] Michael Taylor. “A Landscape of the New Dark Silicon Design Regime”. In: *Micro, IEEE* (Sept. 2013).

[84] Michael Taylor. “A Landscape of the New Dark Silicon Design Regime”. In: *Design Automation and Test in Europe*. Apr. 2014.

[85] Michael Taylor. “The Evolution of Bitcoin Hardware”. In: *Computer, IEEE* (Sept. 2017).

[86] Michael Taylor. “Tiled Microprocessors”. PhD thesis. Massachusetts Institute of Technology, 2007.

[87] Michael B. Taylor. "BaseJump STL: SystemVerilog needs a Standard Template Library for Hardware Design". In: *Design Automation Conference*. June 2018.

[88] Michael B. Taylor. "Bitcoin and the Age of Bespoke Silicon". In: *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 2013.

[89] Michael B. Taylor. "Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse". In: *Design Automation Conference (DAC)*. 2012.

[90] Michael B. Taylor et al. "A 16-issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network". In: *IEEE International Solid-State Circuits Conference (ISSCC)*. Feb. 2003.

[91] Michael B. Taylor et al. "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams". In: *International Symposium on Computer Architecture (ISCA)*. June 2004.

[92] Michael B. Taylor et al. "Scalar Operand Networks". In: *IEEE Transactions on Parallel and Distributed Systems*. Feb. 2005.

[93] Michael B. Taylor et al. "Scalar Operand Networks". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. Feb. 2005.

[94] Michael B. Taylor et al. "Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures". In: *International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2003.

[95] Michael B. Taylor et al. "The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs". In: *IEEE Micro*. Mar. 2002.

[96] Michael Bedford Taylor. "Geocomputers and the Commercial Borg". In: *SIGARCH Computer Architecture Today*. Dec. 2017.

[97] Michael Bedford Taylor. "Your agile open source HW stinks (because it is not a system)". In: *ICCAD*. 2020.

[98] Michael Bedford Taylor et al. "ASIC Clouds: Specializing the Datacenter for Planet-Scale Applications". In: *CACM* (2020), pp. 103–109.

[99] Shelby Thomas et al. "CortexSuite: A Synthetic Brain Benchmark Suite". In: *International Symposium on Workload Characterization (IISWC)*. Oct. 2014.

[100] Sain-Zee Ueng et al. "CUDA-lite: Reducing GPU programming complexity". In: *Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31-August 2, 2008, Revised Selected Papers 21*. Springer. 2008, pp. 1–15.

[101] Luis Vega and Michael Bedford Taylor. " RV-IOV: Tethering RISC-V Processors via Scalable I/O Virtualization ". In: *CARRV*. 2017.

[102] Ganesh Venkatesh et al. "Conservation cores: reducing the energy of mature computations". In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2010.

[103] Ganesh Venkatesh et al. "QsCores: Configurable Co-processors to Trade Dark Silicon for Energy Efficiency in a Scalable Manner". In: *International Symposium on Microarchitecture (MICRO)*. 2011.

[104] Elliot Waingold et al. "Baring it all to Software: Raw Machines". In: *IEEE Computer*. Sept. 1997.

[105] Andrew Waterman et al. "The risc-v instruction set manual, volume i: Base user-level isa". In: *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* 116 (2011), pp. 1–32.

[106] Chenhao Xie et al. "Q-VR: System-Level Design for Future Mobile Collaborative Virtual Reality". In: *ASPLOS*. 2021.

[107] Shaolin Xie et al. "Extreme Datacenter Specialization for Planet-Scale Computing: ASIC Clouds". In: *ACM Sigops Operating System Review*. 2018.

[108] Karen Zee et al. "Runtime Checking for Program Verification". In: *RV*. 2007.

[109] Xingyao Zhang et al. "η-LSTM: Co-Designing Highly-Efficient Large LSTM Training via Exploiting Memory-Saving and Architectural Design Opportunities". In: *ISCA*. 2021.

[110] Ritchie Zhao et al. "Celerity: An Open Source RISC-V Tiered Accelerator Fabric". In: *7th RISC-V Workshop*. 2017.

[111] Qiaoshi Zheng et al. "Exploring Energy Scalability in Coprocessor-Dominated Architectures for Dark Silicon". In: *Transactions on Embedded Computing Systems (TECS)* (Mar. 2014).

[112] Yi Zhu et al. "Advancing supercomputer performance through interconnection topology synthesis". In: *International Conference on Computer-Aided Design (ICCAD)*. 2008, pp. 555–558.

[113] Yi Zhu et al. "Energy and Switch Area Optimizations for FPGA Global Routing Architectures". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)*. Jan. 2009.