WRAITH: A Resource-Efficient Dataflow Accelerator

ECE 427 Final Report

Prakhar Gupta
prakhar7@illinois.edu

Ingi Helgason
ingih2@illinois.edu

Pradyun Narkadamilli
pradyun2@illinois.edu

Sam Ruggerio
samuelr6@illinois.edu

***Abstract*—This work presents WRAITH, a compute substrate that attempts to exploit available TLP and DLP in a power-efficient manner by colocating a set of "virtual" RISC-V processors and a dataflow accelerator on a shared datapath.**

I. Project Summary

A. Main Features

WRAITH contains a compute substrate consisting of 16 Processing Elements (PEs), arranged as a 4x4 2D mesh. 8 of these PEs are configured with 32-bit multipliers and 8 are configured with 32-bit ALUs (supporting most standard RISC-V ALU operations). Each PE can support up to 14 unique actions. A 2x2 subcluster PEs share a register file, with banked writes and all-way reads. There are two "scratchpad" banks, each 2KB, which hold the input and output results of kernel data respectively.

Two 5-stage RISC-V cores, implementing RV32IM are located near the compute substrate. Each core has a pipelined, 2KB, direct-mapped, shared I/D-cache. These cores utilize the compute mesh interface to perform multiplication. Provided the mesh is configured to handle RISC-V multiplications, this action can be performed even if the compute mesh is currently processing a kernel. These RISC-V cores share a 6-cycle 32-bit divider to fully implement the M specification.

All elements of WRAITH are managed by the Mesh and Memory Management Unit (MMMU). The MMMU handles a 32-bit tri-state bus from an off-chip controller and responds to memory requests from the RISC-V cores. When the off-chip controller is ready to utilize the mesh, it utilizes a cooperative protocol to receive the kernel for the mesh, along with the kernel input data itself. The MMMU also handles CSR read and write requests to inspect the functional state of the chip.

WRAITH fits onto a $1mm^2$ square tile using a 65nm process. It can support clock speeds up to 500MHz from an external clock, with some test features limited to 200MHz.

B. High-Level Architecture

The WRAITH architecture is depicted in Figure 1. At the center is a CGRA-like mesh of processing elements (PEs). Typical CGRA architectures tend to follow the ADRES [1] scheme, where each PE has a separate PC and executes its own program stored in a private configuration RAM. Care must be taken in the configuration to ensure cycle-by-cycle correctness during execution.

By contrast, WRAITH adopts a reactionary model for its mesh. Rather than operating on a sequential program, PEs instead generate responses to packets received from any of their neighboring nodes. The protocol for doing so is described in section V-B. Both execution and configuration tasks for a PE are initiated by sending it a packet, meaning WRAITH's mesh does not need any explicit control logic.

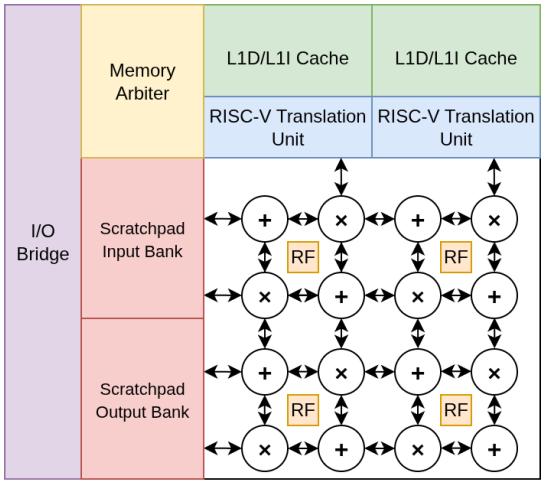

Fig. 1. WRAITH Architectural Diagram

Control logic is instead managed by two controller peripherals, the MMMU and a Scratchpad Memory controller. These blocks work together to format input data and requests into a format ingestible by the ingress ports of the mesh.

Below, we give more detailed architectural descriptions of the various components utilized by WRAITH.

Processing Elements: Each processing element is a 3-stage pipeline which takes an input packet from a 2-depth FIFO, performs an operation, and generates an output packet to a 2-depth FIFO. See Figure 2 for a high level diagram.

The PE supports basic forwarding to mitigate data hazards, and has stall controls if the destination egress FIFO is full during operation. The multiplier supports 32-bit signed and unsigned multiplication, compliant with the RISC-V M specification. The ALU supports all standard RV32I operations except for `slt` and `srl`.

Each PE is connected to up to 4 adjacent PEs, with boundary PEs being connected to either the input/output scratchpad banks or the RISC-V cores for multiplication. The layout is a checkerboard of ALU and Multiplier PEs.

RISC-V Translation Units: WRAITH is equipped with two RISC-V "Translation Units" (RVTU). Each RVTU is a stan-

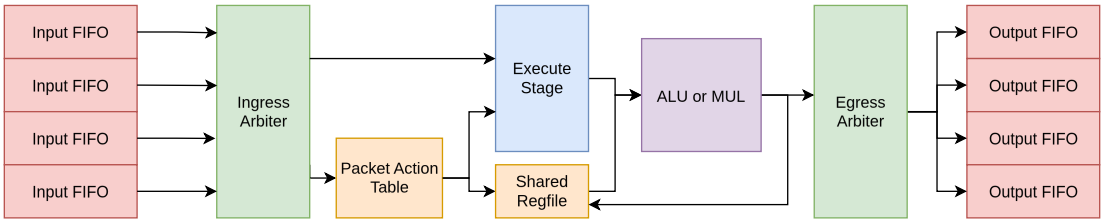

Fig. 2. Processing Element Pipeline

dard 5-stage pipeline core implementing RV32IM. However, in the Execute stage, a multiplication μOP is translated into two packets compliant with our mesh architecture. These packets are injected into the mesh and the RVTU awaits a response. This is our main proof of concept to show viable reuse of the compute mesh during low utilization. For completeness of the M specification, we included a 6-cycle 32-bit divider that is shared between the two RVTUs. See Figure 3 for the modified execute stage and shared cache changes.

Each RVTU has a pipelined, 2KB, direct-mapped, shared I/D-cache. These caches do not support triggered cache-flushes, as it is expected that this is done in software due to their small size. The user is expected to keep the memory sections of each RVTU's program separate to avoid collision, as caches are not coherent.

Each RVTU also has a dedicated IO reset pin, and a halt CSR to indicate program completion. The reset pin also serves as a disable pin of sorts, as the RVTU will not emit any requests to any shared resources if it is held high.

Scratchpad Banks: WRAITH contains two 2KB banks which handle input and output kernel data between the off-chip controller and the compute mesh. The off-chip begins a request by setting up CSRs to tell the scratchpad controller which PE to send and expect data from, how many packets are being set, and what PID should be used when sending the packets into the mesh. Kernels can operate on up to 512 32-bit words.

By default, the scratchpad controller is responsible for adding the PID to the input data, which determines whether the mesh is being configured or utilized. There is an alternative mode to reduce the bit-depth of the packet payload and include mixed-PIDs in the 32-bit word.

Once the scratchpad controller has received the correct number of packets, the off-chip controller can trigger a writeback to flush the output scratchpad bank. Provided that the kernel does not require pre-loaded constants each run, once the mesh is programmed once, the off-chip controller can send multiple cycles of data in succession.

Mesh and Memory Management Unit: The MMMU handles the memory requests from the RVTUs, the scratchpad banks, and the off-chip controller in a cooperative protocol. The MMMU controls a 32-bit tristate bus, polling specific pins to receive requests from off-chip. The MMMU contains an arbiter which manages bus access contention between the RISC-V cluster, the CSR file and the scratchpad banks.

Additionally, the MMMU manages two sets of CSR register files: one written to by on-chip and read by off-chip, and one read by on-chip and written to by off-chip. CSR requests are given priority over kernel operations, which is given priority over RVTU requests.

C. Physical Design

WRAITH utilizes a $1mm^2$ tile using TSMC's 65nm process. There is a single clock domain which drives both the IO ring and internal modules. WRAITH supports clock speeds up to 500MHz, with some testing features limited to 200MHz. The standard IO ring allots a $820 \times 820\mu m$ core area. We were able to expand our core area to $844 \times 844\mu m$ with four $34 \times 34\mu m$ triangular corners blocked for place and route. This increased our usable area to $710024\mu m^2$ from $672400\mu m^2$ Before filling, we utilize $547602\mu m^2$ core area, resulting in 77% density. A general view of our physical design can be seen in Figure 4, while the triangular blockage we used can be seen in Figure 5.

Our estimated power usage at a 500MHz clock, with a 1.0V core voltage:

- 158.4mW Internal Power
- 122.9mW Switching Power
- 255.2mW Leakage Power[1]

Our IO utilization is as follows, for a total of 52 pins:

- 32 pins for the inout tri-state bus
- 2 pins for RVTU resets
- 6 pins for Scratchpad Bank "fallback" mode
- 3 pins for inspecting the IO Bridge FSM state
- 1 pin for power indication
- 2 pins for clock & reset
- 6 pins for power & ground pairs.

D. Design Choices

Reactive Mesh Architecture. CGRAs which adhere to the ADRES architecture [1] have a configuration RAM in each processing element to determine what operation to do next. Although this allows for more complex data orchestration paradigms, it would have been infeasible to implement in an area-efficient manner within the time constraints. Our reactionary model vastly simplifies implementation details and routing overhead, albeit at the cost of scheduling flexibility.

[1]This does not include leakage induced by our register file IP blocks, as the analysis reports 42W of leakage, which is erroneous.

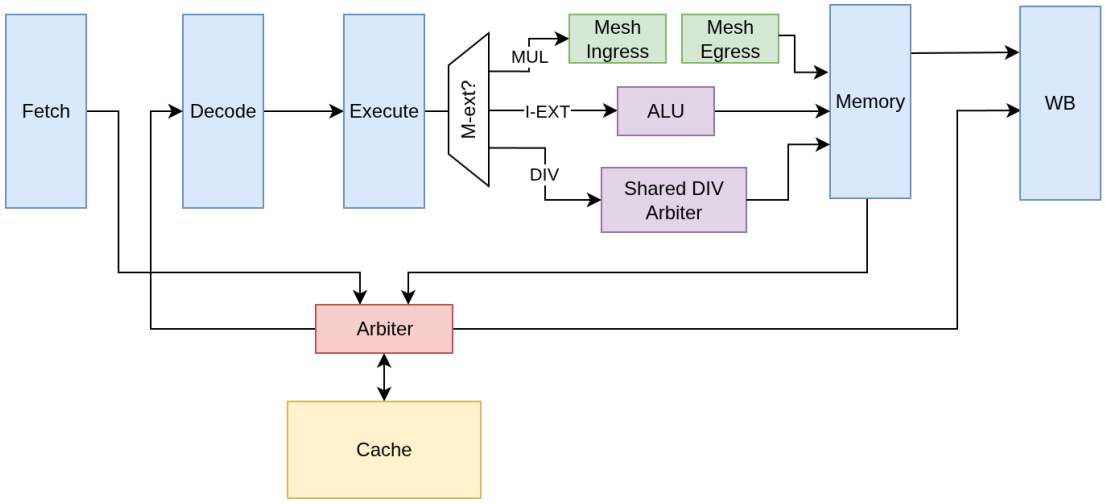

Fig. 3. RISC-V Translation Unit

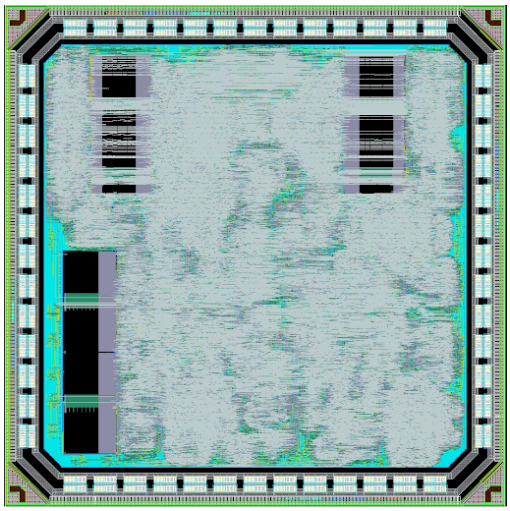

Fig. 4. WRAITH Chip physical layout, with metal layers 7-9 hidden.

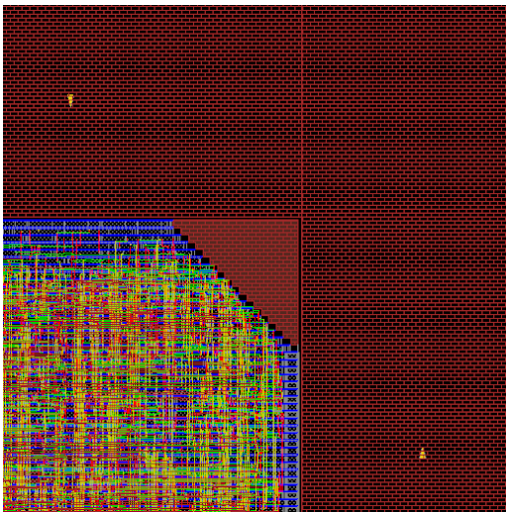

Fig. 5. A closeup of the corner blockage which permits expanded core area.

Packet Translation. One possible method to reuse multipliers in the mesh would be a top-level configuration switch on a processing element dedicating it to the CPU, effectively removing it from the mesh. We decided that it would be more flexible to translate multiplication instructions into mesh packets. This consumes 5 of the PAT entries within the corresponding PE, however the alternative would be to completely remove it, so we feel that this was a better middle ground.

RVTU Caches. We chose a shared I/D cache with 128 entries of 128 bits each due to size constraints and minimum supported configurations of the IP generator. We wouldn't be able to make a reasonably smaller cache for a dedicated instruction cache without running into routing issues. 128 entries is the minimum number of entries one can generate using ARM's IP generators.

Split Scratchpad Banks. Common scratchpad architectures in literature for CGRAs is usually one unified scratchpad cache which handles both input and output data. Due to the reactive nature of our mesh and wanting to simplify the memory bus protocol as much as possible, we opted to split our scratchpad banks into dedicated input and output banks. This simplified the Scratchpad to Mesh interconnect without needing large crossbars to support entry from any PE.

32-bit Bus. We realized that with CSRs, we needed very few IO pins to configure our chip. This allowed us to make use of a full 32-bit bus which gives substantial speed improvements to loading kernel data and handling RISC-V cachelines.

E. Verification & Test Strategy

We verified our design using block-level test benches, and a sophisticated top level test bench which completely simulates an off-chip controller to configure and run kernels on the mesh and run programs on the RVTUs.

We have a PE test bench which verified the configuration and functional output of each PE, PE subcluster, and mesh over

the course of several test cases. We have an RVTU test bench which verifies the RVTUs with a RVFI interface and Spike, along with the shared divider and mesh interface. Finally, our top level test bench configures and runs a mesh, whilst handling memory requests from both RVTUs simultaneously, with RVFI and Spike verification.

The top level testbench has several asynchronous tasks, that snoop the bus and react to requests made by the RVTUs. It also has a contention mechanism for all the potential off-chip drivers (CSR Write, SPM Write, Cacheline Response), so as to verify pseudo-random order of accesses. The top level test bench also has a Post-PnR version which runs on the compiled netlist of our design. This catches timing violations as well as functional correctness after synthesis and Place and Route.

We have reserved some IO pins for DFT features of our chip during bring-up. Notably, we have 3 pins which connect directly to the MMMU's IO Bridge to inspect the current state of the controller FSM. We also have 6 pins to stream packets in and out of the compute mesh, bypassing the scratchpad banks or IO bridge in the event of unforeseen failure. This is a rudimentary protocol which takes in a packet over 36 clock cycles and writes it out over 36 clock cycles. We also have a single pin to indicate successful power delivery. These control pins were false pathed in analysis, but we estimate we can utilize them at speeds up to 200MHz without timing violations.

F. Initial Proposal Differences

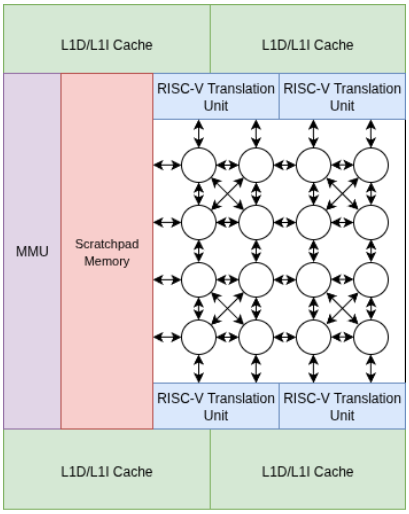

Fig. 6. Initial WRAITH Architectural Diagram

Our initial proposal diagram can be seen in Figure 6. We had to make a few simplifying changes due to area constraints. The largest one was our reduction of four cores to two, due to the limitations we had generating smaller caches and generally lack of total available area. We split our scratchpad banks to reduce crossbar area and simplify the logic between the mesh and the IO bridge.

We also simplified our mesh topology, initially each 2x2 subcluster had an all-to-all connection, but we reduced it down to be entirely 2D. In part, this change was done due to moving from having small, private register files within each PE to a shared regfile among a subcluster. We also had 4-depth FIFOs connecting each PE, but consumed far too much area, so we reduced it to a 2-depth PE.

We also simplified the intended use of the mesh during RVTU execution, limiting it down to just multiplication. In the future, we hope that with larger area we can support more RISC-V operations within the mesh with a lower latency.

Not all changes were simplifications however, we were able to increase our IO bus from 16-bit to 32-bit, as we moved to using CSRs for most control features. We also added support for varying PIDs to support generation of non-linear kernel routing.

Additionally, we were initially planning clock speeds of 100-200MHz, but were luckily able to achieve 500MHz across the entire chip.

II. Post-Silicon Validation Plan

We expect the entire group to be involved with Post-Silicon validation, to varying degrees. Our validation plan is straightforward, due to the main interface with WRAITH being a single 32-bit bus, resets, and some DFT signals. We will port our Post-PnR test bench to an FPGA to act as an off-chip controller. We will design a simple PCB to put our packaged chip onto, with all signal wires connecting to the FPGA'S IO, except for our LED indication pin and power.

We may also implement the IO bridge and portions of the compute mesh on a secondary FPGA to perform some initial testing prior to our chip's delivery date in late February. During this time, we will also design and program additional kernels to test on our compute mesh.

Once we have the chips on a PCB, we will begin with low-speed testing to verify the DFT features are working as intended and we are receiving correct results from the mesh and RVTUs. We will repeat these tests at higher clock speeds until we reach our analyzed 500MHz or detect some failure. We expect testing to be done in mid-April.

III. Individual Contributions

- **Prakhar**
  - Scratchpad Memory/Controller RTL & Verification
  - IO Bridge Communication & CSR definitions
  - PE Mesh Verification
  - Full-System & Post-PnR Verification
- **Ingi**
  - MMMU Bridge and Arbiter RTL & Verification
  - PD/PnR Flow, Scripting, Tooling
  - LVS Scripting, Closure & Signoff
- **Pradyun**
  - RVTU Cluster RTL & Verification
  - PE/Mesh Architecture, Golden Model & RTL
  - MMMU Bridge RTL
  - Full-System & Post-PnR Verification
  - Kernel design & Bitstream generation

- **Sam**
  - PE/Mesh RTL & Verification
  - RVTU Cache RTL
  - PD/PnR Flow, Closure & Signoff
  - Miscellaneous tooling & local tool execution

IV. Major Challenges

A. Post-PNR Testing

Our verification methodology definitely required significant engineering effort during the Post-PNR phase. While the behavorial testbench was fairly stable it did not account for metastability and timing path delays in a design back-annotated with the SDF timing annotations. This led to several bugs that were time-consuming to solve, and made debugging difficult, as it was impossible to know if the simulation was failing due to issues with the design, or the testbench. Additionally, the use of synthesis

1) `wait`: The systemverilog `wait` keyword evaluates *before* NBAs for every time event in the VCS simulator. As such, the wait would evaluate on "stale" signals, leading to the testbench following an incorrect flow. Hardcoding a sanity check with a fixed delay as a "hotfix" did not work either due to metastability in FSM state registers. As such it took quite some effort to determine a correct alternative.

2) Thread Starvation: The Post-PNR testbench utilized a multitude of `fork join` tasks. Some of these were cacheline handlers, which would be invoked at the start of the sim, and would snoop/drive the bus to service RVTU cacheline requests, and process cacheline flushes, while contending with other tasks for the bus. Due to direct-mapped nature of the RVTU caches - most of the bus traffic was for cacheline requests. The tasks for this, would always win arbitration for the bus. As such, the other tasks that checked mesh completion by polling the CSRs would be starved of the bus, significantly increasing the simulation runtime. As such, adding a `wait` of a random duration between 1 - 64 cycles to the main loop of the cacheline request handler task, enabled a more fair utilization of the memory bus.

B. ARM SRAM Generation/Import

The SRAM generator had some rough edges, and we consistently experienced issues when importing into Cadence Virtuoso. This necessitated repeated SPICE netlist corrections, which were done through a mixture of processing scripts and manual adjustments where necessary. We had to re-write an existing `replace.py` script due to its compatibility with the Regfile-type Physical IP, which we chose due to its smaller sense amplifier and increased synergy with our design properties.

C. Kernel Mapping and Development

One of our greatest limitations, partially due to scheduling, was the difficulty of modeling and mapping a variety of kernels to our architecture. These limitations are primarily imposed by data permanence issues that can come with subsequent packets overwriting old data in the register file and the single-input-stream model imposed by a unified SPM.

While we cannot support concurrent kernel execution, data permanence is mitigated via a modulo register file scheduling scheme. As our architecture supports varying PIDs, in the kernel configuration and data encoding we rotate PIDs for data bitstreams (for example, cycling through 8 distinct PIDs). By doing so, we can ensure adjacent packets access and write to different register file indices without any clobbering concerns. The exact number of PIDs required to cycle through varies based on the maximum effective pipeline depth of an expressed kernel, and the age of the intermediate results it requires. In practice, we have found success with naively implementing windowing schemes (like for convolution of any 3-wide filter) or imposing relevant constraints on the kernel to reduce required data lifetimes (1D convolution with an 8-wide filter, but only if it has real roots).

V. Documentation

This section discusses the design and properties of the major components of WRAITH. The high-level architecture is discussed in Section 1.B, and a diagram of these components are shown in Figure 1.

A. Memory Bus

As previously discussed, WRAITH requires a large amount of memory transfer between itself and the off-chip components of its surrounding system. Our design reuses a 32-bit wide tristate bus for handshake and data, in order to avoid using dedicated "in" and "out" buses. This allows an almost 1 word/cycle data rate for both input and output operations, despite the limited number of signal I/O pins available (48).

The communication protocol used is somewhat inspired by the PCI "Legacy" bus protocol, where only the essential signals are carried over (namely: `CLK`, and a subset of `AD[31:0]`'s signel address and numerous data lines). Other key features, such as the `REQ#` and `GNT#` signals, are functionally reproduced by requiring a tighter degree of coupling/cooperative interaction. For instance, device requests (for memory read/write) are broadcast using the unified address-data lines. Arbitration between requests initiated by WRAITH and the off-chip host occurs according to a predefined format for bus requests: as such, no explicit `grant` signal is required, as devices assume the request priority order and shall not contend for access to the tristate bus.

1) Limitations/Challenges: Before discussing our implementation of this specification, we would like to note here that this design possess some consequences: some which we anticipated, and others which were not initially clear. First and foremost, our interface is inherently limited to the assumption that the bus possesses no more than two bus controllers; as such, this memory bus places limitations on the systems in which WRAITH can be used. The introduction of certain control overheads along with the high clock rate of the signaling environment (due to the tri-state nature of the

bus) also impose some restrictions on the nature of the testing environment.

Finally, the usage of a single data line led us to constrain memory transfers to being contiguous rather than in the form of several temporally-distinct "packets" (as can be accomplished in other protocols such as AXI4-Stream). As such, the "pinned memory requirement" typical of certain processor-accelerator integrated systems obtains a somewhat stronger form, wherein memory must be coalesced in a buffer with constant/predictable lookup in order to program the WRAITH device. Note that this issue can be somewhat sidestepped by the usage of multiple programming phases, at the cost of increased runtime consumed by programming/configuration tasks.

Unforeseen physical design considerations led us to make RTL modifications late into the design phase, in order to account for the potential for shorts on bus acquisition/release not noticed during protocol design. The revised protocol is presented here, which we believe does not suffer from (or minimally, is significantly less prone to) shorting/bus contention issues. As such, the authors do not necessarily recommend the re-creation of this system, instead offering it up for study.

Regardless, the bus protocol is functional and verified (both behaviorally and post-PnR), and does accomplish its intended goal of providing a multi-master bus with minimal control pin overhead. The following section will describe the protocol we settled on.

2) Memory Bus Protocol: For subsequent discussion, *on-chip* shall refer to any request initiated by the WRAITH chip, while *off-chip* shall refer to any request initiated by the off-chip host.

The WRAITH system, together with its host core, require the implementation of 8 distinct message types, which can be seen in Table I.

TABLE I
WRAITH MEMORY BUS MESSAGE TYPES

Message Type	Description	Code
Requests initiated by off-chip host		
Scratchpad Memory Write	Write to scratchpad memory to provide kernel configuration or data. Length determined by CSR value.	2
CSR Write	Write a value to a WRAITH control and status register.	5
CSR Read Request	Request to read a WRAITH control and status register value.	3
Requests initiated by WRAITH		
Scratchpad Memory Write-back	Write full kernel output to off-chip host. Length configured by CSR value.	1
Cacheline Read Request	Request to read cacheline contents from off-chip host memory/cache.	8
Cacheline Writeback	Write back dirty cacheline contents modified by WRAITH.	6
Response types		
CSR Read Response	Initiated by WRAITH. Contains value of last requested CSR.	7
Cacheline Read Response	Initiated by off-chip host. Contains previously requested cacheline contents.	4

The flow between states is determined entirely by the message type. In order to reduce logic depth around tristate drivers, WRAITH currently exhibits a static priority for off-chip requests; however, the logic described is fully functional if code or message type is used to perform message selection. The states required in order to handle these two message types (that is, the state machine associated with WRAITH's memory bridge) is summarized in Figure 7.

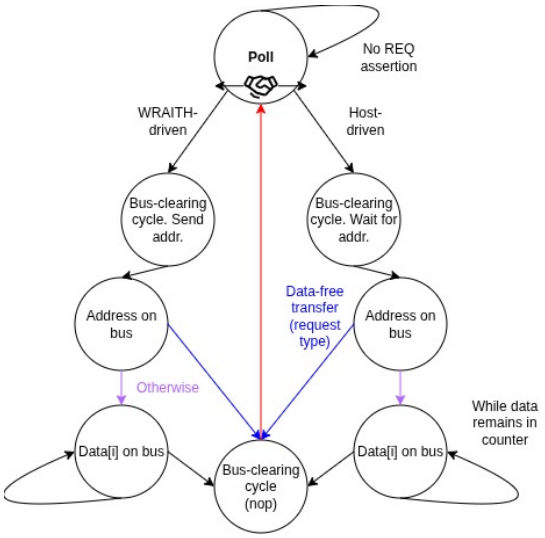

Fig. 7. WRAITH: Memory bus states

With regards to WRAITH-driven data, the on-chip logic used to determine which requests/address/data to send (that is, after the grant is obtained) is best discussed elsewhere. However, we note here that the memory controller performs arbitration between the different sources and types of on-chip requests, such that WRAITH presents only one request during any given "poll" cycle.

The protocol described above is ultimately used to provide the interface pictured in Figure 8 to on-chip components, which is the interface exposed to the on-chip caches, as well as to the scratchpad memory controller.

B. WIMP

WRAITH's mesh implements the **WRAITH Interconnect Mesh Protocol (WIMP)**. WIMP dictates packet format, configuration/response format and routing control for the mesh and its peripherals.

All packets in WRAITH are single-beat to maintain consistency and simplify implementation details. A packet is comprised of two fields:

1) A 4-bit Packet ID (or PID)
2) A 32-bit immediate value

All WRAITH PEs implement a register-immediate scheme – that is to say, any operation is done purely by taking one register and the mesh immediate as inputs to the FU. The FU output is then used as the immediate for an output packet.

By looking at the PID, WRAITH is able to determine what operation to perform and how to generate a response packet. This is done by indexing into a PE Action Table (or PAT) with

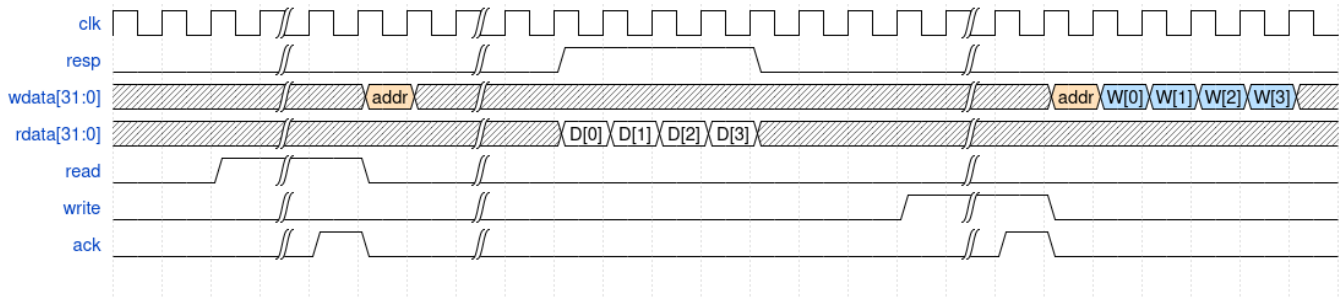

Fig. 8. On-chip Interface

the PID. The format of a single PAT entry is shown in Table II, with a graphical representation in figure 9.

TABLE II
WIMP PAT ENTRY FORMAT

Field Name	Description	Bitwidth
`response_pid`	What PID to tag the output packet with	4
`dest`	Which direction to send the packet to	3
`src`	Index into the RF for input value	5
`rd`	Index into the RF to write output	5
`rf_we`	Write enable to register file port	1
`imm_we`	Write the packet immediate directly to the RF	1
`src_imm`	Interpret src as an immediate	1
`fu_op`	Function select for instantiated FU	3

It should be noted that despite WRAITH only having 4 possible output directions, `dest` is made 3-bit to accommodate an additional `SINK` direction. This allows some operations, like a load-immediate from an RVTU or spacer packet, to terminate with a side effect but no spurious output packet.

As routing logic is encoded by the PAT configuration but writing the PAT configuration is determined by routed packets, there is a bit of a catch-22 on initial reset to program the mesh. To avoid this, a bypass routing mechanism is implemented specifically for PIDs 0 and 1. Each PE in the mesh is assigned a coordinate from 0 to 15. This bypass network expects packets to be inputted at the top left node, passes a packet to the right until in the correct column, then downwards until the destination is reached.

PID 0 is always allocated for writing an entry into the PAT. Its packet immediates are formatted as shown in Figure 9, and will always spawn a response of PID 0.

Some kernels require hard-coded constants in the register file to operate correctly. Thus PID 1 is allocated for a "constant load" operation, and formats its packet immediates as shown in Figure 9. These packets are also routed via the bypass network.

C. Processing Elements

There are a total of 16 Processing Elements within WRAITH. 8 support standard ALU operations, while the other 8 support multiplication. Each 2×2 subcluster of PEs share a 32-entry register file. Each PE can write to 8 entries in the register file, but can read from all 32 entries. As seen in Figure 2, each PE is connected to up to 4 ingress FIFOs, and outputs packets to egress FIFOs, all of which are 2-depth. The Ingress Arbiter is a simple round robin arbiter to dequeue a packet from one of the available ingress FIFOs.

This packet is then used to supply the 32-bit immediate to the execute stage, and index into the Packet Action Table, as described in Section I-B. The next stage of the processing element utilizes the functional unit, whether it be an ALU or multiplier. The output is then registered into the 3rd stage, which is pushed into an egress FIFO.

If a FIFO is full, the PE will stall until it becomes available to push through. It will not backfill or process other packets which don't depend on the full FIFO. It will however, support forwarding over the register file if the PE is reading and writing to the same entry.

The operations for the ALU are as follows, where a is the packet immediate and b is the register value:

Option	Operation
0	$a + b$
1	$a << b$
2	$a >>> b$
3	$a - b$
4	$a \oplus b$
5	b
6	$a \mid b$
7	$a \,\&\, b$

TABLE III
PE ALU OPERATIONS

The ALU is given an identity mode to act as a pass-through value for b. This is used for windowing operations in some kernels (like convolution) in conjunction with the PAT's `imm_we` option. The multiplier supports standard RISC-V format, `mul, mulh, mulsu, mulhu`, as well as the identity function. Two multiplier PEs are connected to the RVTUs, to provide direct handling of the RISC-V multiplication instructions. Two PEs are connected to the input scratchpad bank, and 2 PEs are connected to the output scratchpad bank.

D. RISC-V Cluster

WRAITH has a RISC-V cluster, depicted in figure 10, consisting of two RISC-V translation units (RVTUS).

These are for the most part standard RISC-V cores that support the RV32IM ISA, and are built to fit a typical 5-stage pipelining scheme with static branch pessimism and forwarding to mitigate data hazards. To simplify logic design

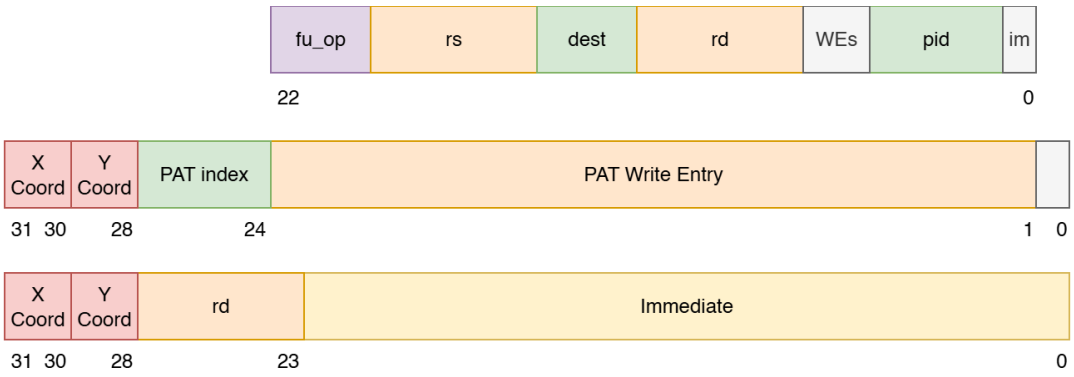

Fig. 9. *Above:* PAT Write Entry Format. *Middle:* Configuration Packet. *Lower:* Constant-Load Packet.

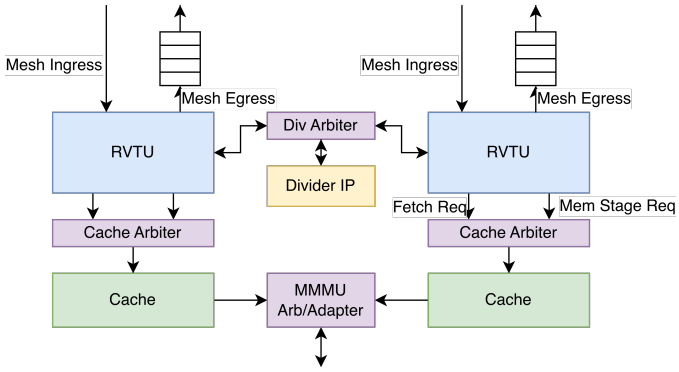

Fig. 10. RISC-V Cluster Diagram

and verification efforts, a ”global stall” approach is taken, where the pipeline can only progress once all pipeline stalls have been resolved. This includes cache access stalls, multi-cycle execute operations and bubble insertion on consecutive memory instructions.

Notably, RVTUs do not inherently contain the logic to compute M-extension instructions – rather, they dispatch this logic to other shared resources via the interfaces depicted in figure 3.

For divide support, the RISC-V cluster comes with a shared 6-stage sequential divider. Divider access is managed by a round-robin arbiter, which provides a port to each pipeline. This arbiter accepts divide operands and an operation select from each pipeline's execute stage, and broadcasts the completion status of the divider IP once the operation has completed. To comply with global stall semantics, the arbiter is aware of when the pipeline associated with an outstanding request is stalling, and will maintain the result and valid signals on its output until this stall is lowered. Furthermore, to ensure correctness of operation, dependencies on outstanding load instructions are monitored to guarantee that both inputs are ready and broadcasted on forwarding paths (as applicable) before initiating a request to the arbiter.

Multiplication is performed by co-opting MUL processing elements in the mesh via a set of dedicated mesh ingress/egress ports connected to the RVTU's execute stage. To conform with convention for ingress ports, the RVTU instantiates a 2-deep FIFO to enqueue ingress packets into before consuming them.

For each multiplication instruction, an RVTU will generate two packets to the mesh egress packet – one to transfer its first operand into one of the MUL PE's register slots, and another to initiate the actual multiply instruction. This store operation is typically implemented as a multiply-by-1, where the 1 is a constant stored in the PE register file. This is done to facilitate a register-register instruction within the mesh's register-immediate ISA. The MUL PEs response is sent to the RVTU's ingress FIFO, where it is then dequeued and interpreted as an operation result once all other pipeline stalls have resolved.

To simplify logic, RVTU requests simply WIMP packets with no special properties – thus multiply support must explicitly be configured in the mesh's kernel programming. During normal operation, this also means that if the RVTU is expected to run a program that has multiplication instructions, its reset pin should be asserted until programming completes.

To facilitate sharing of the RVTU cache, a cache arbiter (CA) is attached to the fetch and memory stages of the pipeline. Priority is statically given to the memory stage, and it is assumed that there is always implicitly a request from the fetch stage. As both stages must have their requests fulfilled for the pipeline stall to go down, memory stage responses are buffered in a register stage. This register stage is tagged with a valid flag raised on cache response and lowered on the first cycle the pipeline stall drops. While this valid flag is asserted, the arbiter will ignore the request interface of the memory stage.

Both cache arbiters are then fed into a second-stage round-robin arbiter that manages access to the MMMU. This second-level arbiter accepts and services CA requests at the cache line granularity, but includes a burst adapter to convert CA requests to the MMMU's on-chip memory protocol.

E. Processing Mesh Bitstream

Mesh bitstreams are typically encoded as three distinct sequences.

1) Configuration: The contents of each PE's PATs, laid out linearly and tagged with the PE and PAT index they are associated with.
2) Constants: Any constants required by the program that are larger than 5 bits. These are laid out the same way as the configuration bitstream, albeit with a wider "line index" (for the register file). Register indices must match the register file bank allocation of the stated processing element, as this bitstream facilitates a series of register-write operations.
3) Data: The input data for the mesh to operate on. Can be expressed as a sequence of 32-bit values, or as a sequence of 28-bit values tagged with a 4-bit PID specifier. The interpretation is determined by the `PID_SEL` CSR.

As our system adheres to register-immediate operation, and input data is delivered via the immediate lines on the mesh, we are unable to provide constants greater than 5 bits intrinsically as part of the configuration bitstream. Thus the constant bitstream preloads register indices unused by the kernel with any required values for access at runtime. In practice, the constant bistream has been used to preload the mesh's register files with filter constants when performing 1-D convolution.

Each bitstream is sent over separate bus transactions. To simplify SPM and bridge handling, PIDs 0 and 1 were allocated for handling the configuration and constant bistream. The appropriate PID is indicated via CSR settings.

Typical execution semantics dictate programming the mesh with the configuration and constant bitstreams once, then delivering an arbitrary number of data bitstreams for execution. The mesh can be reprogrammed at any time (assuming no outstanding data transaction). It should be noted that if the chip is not being reset before programming, one must take care to either not overwrite dedicated RVTU PAT and register file entries in the new kernel, or assert the RVTU resets prior to reprogramming. There is no explicit restriction in place to enforce constant registers not being overwritten – if desired, this must be done by the kernel expressed in the configuration bitstream.

VI. Testing Instructions

A. Generating Mesh Bitstream

At the moment, all kernels must be hand-written – due to the novel nature of our mesh and the additional constraints imposed by RVTU configuration, there is no pre-existing compiler framework compatible with our chip. While a number of dummy kernels were flashed onto the mesh to test functionality, at the moment we do all full-system functional testing with a Conv3 kernel (convolution of some input data stream with a 3-wide filter). This is done with integers at the moment, but floating-point constants can be supported via software in the future.

To generate the bitstreams for Conv3, run `cartographer/conv3_mapper.py`. This will dump a data bitstream into `out.dat`, a constant bitstream into `out.const`, and a configuration bitstream into `out.cfg`. A "golden output" will also be printed to the terminal to verify runtime output against. A pre-generated set of Conv3 kernel bitstreams is available in `chip/hvl/top-verif/mem_files`.

B. RVTU Simulation

The RVTUs are tested via several unit testbenches, the most relevant being `rvtu_cluster_sim`. This testbench instantiates the full RISC-V cluster, instantiates two auto-configured MUL PEs, and has a memory model that emulates the timing of the MMMMU arbiter.

To run this testbench, go to `chip/sim` and run `make run_vcs_cluster_sim`. This target takes `PROG0` and `PROG1` arguments that configure which program to run on each core. This can be a C/C++, assembly or compiled program. A small assortment of provided testcases is available in `chip/sim/testcode`, with the Makefile defaulted to run Coremark on Core 0 and `needle.cpp` (Rodinia's Needleman-Wunsch benchmark) on Core 1.

Do note that while using pre-compiled programs is supported, the linker script uses different static-mapped memory regions for each core to make a unified memory model more feasible. Using pre-compiled memory images for a core they it is not intended is very likely to quickly incur runtime errors.

Each core is harnessed with an RVFI monitor for runtime correctness-checking, but will also dump commit logs to `chip/sim/vcs/{core0.log, core1.log}`. These can be diffed against Spike by running `make spike_im PROG=<PROGRAM>` to generate a golden log. The ELFs required to do so are dumped in `chip/sim/memgen` when running the main simulation target. The ELF's name will be appended with the core number it was compiled for, and you will likely run into Spike mismatches if using the wrong ELF file.

C. Behavioral Full-System Verification

Note: For brevity, all paths listed in this section are relative to the `/chip` subdirectory.

After navigating to the `sim` directory, run `make vcs/top_tb` and `make run_vcs_top_tb`. Running so converts 2 RVTU programs in the `$PROG[0,1]` paths to memory maps. Do note that these paths are set by default in the Makefile.

The testbench also ingests the files in `chip/hvl/top-verif/mem_files/` directory, namely `[cfg,cnst,data]_mem.hex`. These files represent the bitstreams for the configuration, constant and data streams for the Mesh.

This simulation also consumes the files in the `memgen` folder, which contains a memory map for the RVTU memory space (generated by compiling the program files mentioned earlier).

Lastly, the outputs of the behavioral sim can be observed in `vcs/simulation.log`. Additionally, the output memory maps (SPM WB data stream, final RVTU memory map) can

be found in `vcs/spm_wb.dmp` and `rvtu_wb.mem` in the `hvl/top-verif/mem_files/` directory.

The SPM Writeback can be diffed against a golden output in the `hvl` directory. Similarly, the RVTU memory transaction logs can be diffed against logs in the groupdir (namely, `/groups/ece427-group0/rvtu_memlogs/`).

D. Post-PNR Full-System Verification

After navigating to the `chip/sim` directory, run `make vcs/post_pnr_tb` and `make run_vcs_post_pnr_tb`

After navigating to the `chip/sim` directory, run `make vcs/top_tb` and `make run_vcs_top_tb`.

This testbench uses the same variables/paths as the Behavioral FSV testbench, with an additional path to the SDF file used for back annotation. This SDF file is expect to be in the `synth/pnrout/` directory with the name `top.pnr.sdf`.

VII. Acknowledgements

We would be remiss without thanking all the people and structures that helped us complete this project.

Thank you to Milos Becvar, our mentor, for providing valuable guidance and sanity-checks during an admittedly frantic development cycle.

Thank you to Stanley Wu and Jerry Wang for offering all manner of support throughout this project. Your experience was invaluable in avoiding common pitfalls and making a feasible design.

Finally, thank you to Apple, the University of Illinois and Professor Dong Kai Wang for giving us this opportunity. Taping out a chip as university students is an extremely rare opportunity and helped us build invaluable experience pursuing future careers in this field.

References

[1] B. Mei, M. Berekovic, and J-Y. Mignolet. “ADRES & DRESC: Architecture and Compiler for Coarse-GrainReconfigurable Processors”. In: *Fine- and Coarse-Grain Reconfigurable Computing*. Ed. by Stamatis Vassiliadis and Dimitrios Soudris. Dordrecht: Springer Netherlands, 2007, pp. 255–297. ISBN: 978-1-4020-6505-7. DOI: 10.1007/978-1-4020-6505-7_6. URL: https://doi.org/10.1007/978-1-4020-6505-7_6.