IMPLEMENTATION OF A CPU CONTROLLER FOR DYNAMIC BINARY TRANSLATION

BY

SOUMIL GUPTA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Computer Engineering
in the Grainger College of Engineering of the
University of Illinois Urbana-Champaign, Spring 2025

Urbana, Illinois

Adviser:

Professor Nam-Sung Kim

ABSTRACT

As modern compute workloads grow more demanding, hardware accelerators integrated with CPUs are valuable for improving the execution speed and energy efficiency of highly parallelizable workloads. However, writing code to run on these accelerators generally requires specialized compilers or synthesis tools before execution. To address this, Wang *et al.* [1] introduced the Microarchitecture Extensions for Spatial Architecture (MESA) controller, a hardware block that continuously monitors the CPU instruction stream and transparently offloads suitable traces to a configurable Coarse-Grained Reconfigurable Array (CGRA) for acceleration. However, the current MESA controller is constrained to a fixed mapping algorithm, hence limiting its adaptability to different CGRAs in commercial applications without costly redesigns per CGRA architecture. This thesis aims to extend the MESA controller's capabilities by developing a programmable version capable of efficiently mapping workloads to a wide range of CGRAs. The enhanced controller will support a custom instruction set, allowing it to execute mapping algorithms tailored to the specific configuration of the CGRA.

The revised MESA controller is implemented in SystemVerilog and verified through hardware and software simulation methods. The outcome of this thesis is a verified hardware design of a programmable version of the MESA controller as an IP that can be integrated into different System-on-Chip (SoC)s with varying CGRA designs.

SUBJECT KEYWORDS

Dynamic Binary Translation, MESA Controller, Coarse-Grained Reconfigurable Arrays, Spatial Dataflow Graphs, Hardware Accelerators, Heterogeneous Systems, Instruction Set Architecture, RTL Design, RTL Verification, Reconfigurable Computing, Parallel Programming, Computer Architecture

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to Professor Nam Sung Kim and Professor Dong Kai Wang, for their unwavering support, invaluable guidance, and deep encouragement throughout my research journey. Their expertise and insights have been instrumental in shaping this thesis and my growth as a researcher. This thesis is built upon the original paper on the MESA controller by Professor Dong Kai Wang, and his insight has been invaluable in the development of this programmable version of the MESA controller.

I extend my sincere thanks to Tianchen (Jerry) Wang, and the Future Architecture and System Technology for Scalable Computing (FAST) lab for their constructive feedback and assistance throughout the project. The vibrant research environment fostered within the group has been a constant source of inspiration.

My appreciation also goes to the Department of Electrical and Computer Engineering at the University of Illinois Urbana-Champaign for providing the resources and opportunities that made this research possible.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

AMAT Average Memory Access Time.

AXI Advanced eXtensible Interface.

CGRA Coarse-Grained Reconfigurable Array.

CPU Central Processing Unit.

DBT Dynamic Binary Translation.

DFG Dataflow Graph.

DUT Design Under Test.

GCT Graph Configuration Table.

GPU Graphical Processing Unit.

GRF Graph Register File.

ILP Instruction-Level Parallelism.

IR Intermediate Representation.

ISA Instruction Set Architecture.

LDFG Logical Dataflow Graph.

MESA Microarchitecture Extensions for Spatial Architecture.

PE Processing Element.

SDFG Spatial Dataflow Graph.

SoC System-on-Chip.

CHAPTER 1

INTRODUCTION

1.1 Motivation & Background

Applications in everyday life, from gaming to running machine-learning models, are all hampered by the limitations of conventional CPU architectures. Traditional CPUs are latency-oriented devices, meaning they are optimized for minimizing delay in sequential instruction execution. Thus, they are not best suited for workloads with a high degree of data or task-level parallelism [2]. As a result, specialized hardware accelerators such as GPUs and TPUs have gained prominence for their significantly higher throughput and energy efficiency on target workloads. However, incorporating such accelerators presents numerous challenges, tempering the promise of democratizing major technologies such as IoT/edge computing, AI/ML, autonomous systems, and data analytics through hardware acceleration. Developers need to adopt specialized programming models (CUDA, OpenCL, OpenMP, etc.), utilize specialized libraries (cdDNN, TensorFlow, etc.), or rely on complex compiler toolchains [1, 3, 4, 5, 6, 7].

To address this inconvenience, Dynamic Binary Translation (DBT) has emerged as a promising alternative solution. The DBT system dynamically identifies 'hot' regions in assembly code, converts them into an Intermediate Representation (IR) that captures control and data dependencies, and uses this IR to program an accelerator [8].

An architecture that can take advantage of DBT for code optimization is CGRAs. CGRAs consist of resource tiles connected together and configured to mirror Instruction-Level Parallelism (ILP) within programs. These tiles include Processing Element (PE)s, memory blocks, and interconnect switches. The method to model and reconfigure a CGRA is through Logical Dataflow Graph (LDFG)s [9, 10, 1]. These graphs express computations

in terms of operations and dataflow graphs, allowing dynamic mapping to the hardware based on available resources and routing paths. CGRAs can be reconfigured for LDFGs of different user-defined code blocks at runtime. In contrast, mainstream, commercial accelerators have their resources fixed ahead of runtime to serve specific code functions [1].

MESA is a hardware controller that can perform DBT integrated onto a SoC that monitors instruction traces during CPU execution, builds a latency-optimized spatial model of the program code, and offloads it to the CGRA [1].

Integrating MESA into a SoC with CGRA, memory modules, and CPU cores removes the need for specialized compilers, languages, or toolchains to build code-specific accelerator fabrics [1]. The goal of a SoC with CPU cores, a MESA controller, and a CGRA includes:

1. Maximal utilization of spatial accelerator resources
2. Offloading of online dynamic spatial mapping
3. Energy-efficiency and high performance
4. Hardware scheduling between CPU cores
5. Iteration-level parallelism and instruction-level parallelism
6. No required modifications to software to enable compatibility with hardware.

A significant limitation of the existing MESA controller design was its hard-coded algorithm for building a LDFG, which is specific to a set CGRA architecture (number, organization, and connectivity of resource tiles). For advancing CGRAs with a partner MESA controller into commercial heterogeneous SoC designs, the MESA controller must be adaptable for any CGRA architecture. This thesis extends the MESA controller's flexibility by writing and verifying a programmable version with a custom ISA and five-stage pipeline capable of executing mapping algorithms unique to different CGRAs. This programmable design will avoid the time-consuming and costly process of redesigning the MESA controller for each SoC with a different CGRA, thus further advancing the utility of the MESA controller and, by extension, the democratization of CGRA-based heterogeneous systems.

The scope of this thesis was the development of the core pipeline of a programmable MESA controller. The controller is designed to be part of a larger project involving the development and bringup of a proof-of-concept SoC with Berkeley RISCV BOOM Cores [11], an adaptation of the Stanford Amber CGRA [9], and the custom MESA controller as described in this thesis. This project was in-part funded and part of a larger joint undertaking by Samsung Advanced Institute of Technology, KAIST University, and the Future Architecture and System Technology for Scalable Computing (FAST) Lab at the University of Illinois Urbana-Champaign.

1.2 Amber CGRA

The Amber CGRA, developed at Stanford, leverages the agile hardware design flow proposed by Bahr *et al.* [9]. This project uses the Amber CGRA because of the reduced complexity involved in generating an Amber CGRA.

First, the type of application a developer wants to accelerate is built using Halide, a . Then, additional custom DSLs called 'PEak,' 'Lake,' and 'Canal' use the Halide code to generate PEs, memory, and interconnects, ultimately generating the HDL Verilog of the CGRA. Building multiple DSLs aims to obtain what Rick et. al. coined "a single source of truth " [9].

The paper extends Halide's built-in scheduling primitives to specify parts of user applications to accelerate. These annotations annotate program regions with hints on memory hierarchy and where to find parallelism exploits that subsequent tools in this design flow can leverage. The designer can indicate which application components will be offloaded to the CGRA. The Halide code is compiled into an internal IR, which organizes the computation into loops and array operations. This IR is further transformed into CoreIR—a standardized, LLVM-inspired intermediate representation for hardware—which provides the user's program in a format that can be mapped into PE and memory tiles. PEak and Lake DSLs respectively optimize cost for memory mapping and kernel mapping. PEak generates PEs based on the application's needs, ensuring that the right operations are available for acceleration. Once PEak has specified the PEs, they are ready for integration into the CGRA. In parallel, Lake configures the memory units that will store data for the computations, defining how memory is structured and ensuring

that data flows efficiently to the PEs with minimal delays. Canal then generates the interconnect network that routes data between PEs and memory. It optimizes the routing paths, minimizing latency and ensuring seamless communication between components [9].

1.3 Related Works

Next, this review will assess the works pertaining to the formulation of the original MESA paper. Instead of focusing on the overall importance of hardware accelerators and various innovations in democratizing their incorporation in computing applications, this review narrows down and assesses core inspiration and previous advances that the MESA paper improves and combines, as it is a more relevant exercise to the already-established project goal.

MESA builds on the foundation of existing works, which focus on improving performance through more efficient code execution through specialized hardware and accelerators. Due to the inefficiencies inherent in Von Neumann architecture and the breakdown of Dennard scaling, the focus for improving compute performance has switched from device-level optimizations to software-level optimizations [1]. Prior hardware architectures like DynaSpam and DORA have attempted to tackle similar challenges, though these approaches were often limited in performance improvements and were more complex to implement. The appeal of MESA is that it requires no modifications to CPU architecture, software, compiler, or a configurable accelerator to be used.

DynaSpAM, introduced by Liu *et al.* [12], tackles the limitations of dynamic mapping for spatial architectures by repurposing idle resources inside an out-of-order (OoO) CPU core. The framework exploits the core's reorder buffer, register file, and issue logic to schedule computation and then uses dynamic binary translation (DBT) to offload hot code regions to a target CGRA, thereby boosting accelerator utilization with minimal extra hardware. In comparison, MESA offers more capability because it is not restricted to any specific CGRA interconnect configuration. Nevertheless, DynaSpAM provides a good boost to performance while minimizing additional area and power overheads, and its benchmarks were compared to the original MESA's.

The architecture uses the OOO CPU's branch predictor, store-set method for RAW (Read-after-Write) & WAR (Write-after-Read) dependencies, and ROB (reorder buffer) for in-order commits to build a program mapping to a 1-D CGRA. First, it performs trace detection to determine code segments to accelerate. It uses a cache to store sequences of instructions that repeat and a history buffer to track the outcomes of recent branches. If the same branch repeats its outcome more than once, the corresponding code segment is marked as a "hot trace." Next is trace mapping, which first requires determining the following sequence of instructions that can practically be mapped. The code is fetched, decoded, and renamed like any other instruction, but the trace is mapped to the hardware at the dispatch stage. DynaSpAM generates a resource-aware configuration for this trace, which can be stored in a cache. This configuration is indexed by the trace's address and branch outcomes and can be reused whenever the trace reappears, saving time on repeated operations. Then, there is trace offloading, where the system can offload the execution of the trace to the spatial fabric (the CGRA), bypassing the OoO pipeline. The operands from the trace execution can be sent back to the OoO pipeline if needed. Control and memory dependencies are managed using speculation, and the ROB ensures that instructions still commit in the correct order, maintaining consistency. DynaSpAM more closely follows MESA's philosophy of providing an end-to-end hardware solution for best utilizing hardware accelerator resources without involving software developers or hardware developers for application-specific IP redesigns. DORA (**D**ynamic **O**ptimizer for **R**econfigurable **A**rchitectures), introduced by Watkins *et al.* [8], takes a software-centric path for using DBT to pair CGRAs with conventional CPU workloads.

DORA uses a helper core to monitor execution and perform DBT, whereas MESA takes a hardware-based approach with a latency-driven algorithm for spatial mapping. MESA offers a more prudent solution because DORA has a longer configuration time, measured in milliseconds, given its software implementation. MESA achieves sub-microsecond configuration times due to its hardware implementation. Regardless, DORA is another useful technology with which to compare MESA benchmarks and provides insight into how to integrate CGRAs. DORA, in this case, offers substantive energy and power benefits in comparison to compilers and works for 2-D CGRAs. With its software approach, it offers a more effective mapping strategy than Dy-

naSpAM. DORA works by adding a lower-power microcontroller connected to the CPU, rather than modifying the CPU pipeline itself as DynaSpAM does. This approach allows a global view of the CPU architectural state and simplifies CGRA integration while lowering overall power consumption. DORA's DBT works by first identifying hot regions, then generating configuration information and optimizing execution, and then stores optimized code in a cache which can be invoked when the same code segment is used in the future. DORA's DBT optimizations include excluding induction registers, applying loop unrolling, identifying accumulators that affect pipelining, and using store forwarding. Post-mapping techniques involve monitoring specific registers to detect nearing hot traces and optimizing register usage by detecting constants, evaluating inefficient loop unrolling, and stopping register monitoring when no longer needed to save power. [8]

Building on the spatial-dataflow insights of Plasticine by Prabhakar *et al.* [13] and the EDGE-based TRIPS processor evaluated by Gebhart *et al.* [14], Nowatzki *et al.* [15] proposes a general dataflow framework aiming to answer the relevant question: "How could domain-specific hardware efficiency be achieved without domain-specific hardware solutions?" They first identified three properties of accelerator programs—high computational intensity, simple control patterns and dependencies, and simple streaming memory access and reuse patterns. Their system offered a dataflow graph just like MESA to represent repeated and pipelined computations, instruction streaming (forwarding), and a scratchpad address space for efficient sharing of data. However, their solution provided much power and area overheads at the cost of making a system generalized for various workloads and programs. The paper also goes over more solutions, such as SIMD, SIMT, and Vector Threads, which have been widely used in hardware and software architectures but present notable challenges. SIMD uses fixed-length vector abstractions, which require large register files and struggle with efficient data reuse. SIMT also encounters inefficiencies in execution. Vector Threads try to do the same operation across multiple elements using simple instructions, but suffer from poor core pipeline utilization. The MESA architecture is a spatial-dataflow approach, which is inefficient with memory addresses. The paper introduced three categories that assessed each of these options: reducing per-instruction power and resource access costs, minimizing the cost of memory addressing and communication, and improving execution resource utilization. [15]

Ultimately, the existing MESA architecture provides a good solution for simplifying the effective incorporation of hardware accelerator resources for programs with large amounts of parallelizable workloads. The assessment of additional papers further confirms the promise of the MESA controller in minimizing power and area costs for maximum performance. The next step is to make the MESA controller more flexible to work for the variety of CGRAs out there, something that many other researchers have been considering with their innovations.

CHAPTER 2

PIPELINED MESA CONTROLLER OPERATION

2.1 Operation

The controller's functionality is broken down into the following steps [1]:

1. Monitor code execution on CPU core and build Spatial Dataflow Graph (SDFG) model of data dependencies between instructions.

2. Build LDFG model of code snippet from SDFG, by optimizing each instruction's expected latency on the CGRA.

3. Map LDFG to CGRA tiles for code execution.

4. Interrupt code execution on CPU and move CPU execution to different process

5. Once CGRA finishes code execution, transfer post-execution architectural state to CPU and resume process execution on CPU.

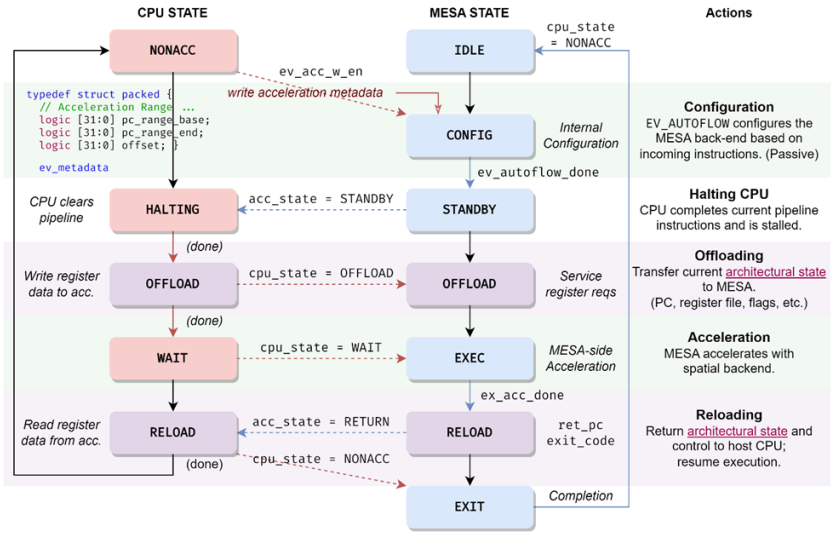

Figure 2.1: In-depth transition diagram between phases of MESA operation

In a dataflow graph (for example see 2.2), each CPU instruction is considered a node, and dependencies are stored with edges to other nodes. The edges also have weights, representing the average number of cycles it takes for data to be transferred from the output of one instruction (the source node) to the input of the next instruction (the destination node). The total latency for each instruction is determined by the latency of the instruction itself and the maximum data arrival time from its predecessors. The overall latency for a sequence of instructions is defined by the instruction on the critical path, which has the longest completion time. The total weight of a path is the sum of the latencies of the instructions and the data transfers between them. Data transfer latencies are calculated based on the Manhattan distance between the nodes, with shorter distances (such as between immediate neighbors) having lower latency. For load/store instructions, latencies are tracked through performance counters, allowing the calculation of Average Memory Access Time (AMAT) for each instruction. These latency values are stored for each type of instruction to inform future optimizations. The LDFG tracks the order in which instructions should be executed in a pipelined approach. The LDFG is then converted into an SDFG, building a 2D representation from a 1D representation of the instructions and their dependencies. The SDFG enables out-of-order execution and identifies the critical path of the instructions. This graph optimizes the placement of instructions along the CGRA's processing elements due to data transfer latency and potential bottlenecks. DBT occurs when CPU instructions are passed to the MESA controller and put in the form of the LDFG.

In MESA's code region detection, a trace-like cache which is a non-intrusive instruction monitoring system is added to the core frontend to identify loops small enough to be handled by the accelerator. Once detected, the system checks for unsupported instructions (like branches, jumps, or I/O operations) and evaluates the loop's computational requirements, memory operations, and iteration count through branch prediction and PC tracking. MESA may not be effective for loops that rarely repeat. If an instruction writes to the instruction cache within the code region, it is stored in the trace cache to avoid redundant fetching and decoding of loop instructions. In terms of memory access, the assumption is that AMAT varies per instruction, and this is measured using performance counters. If a store commits to the address of a younger load that executed first, the store value propagates through the

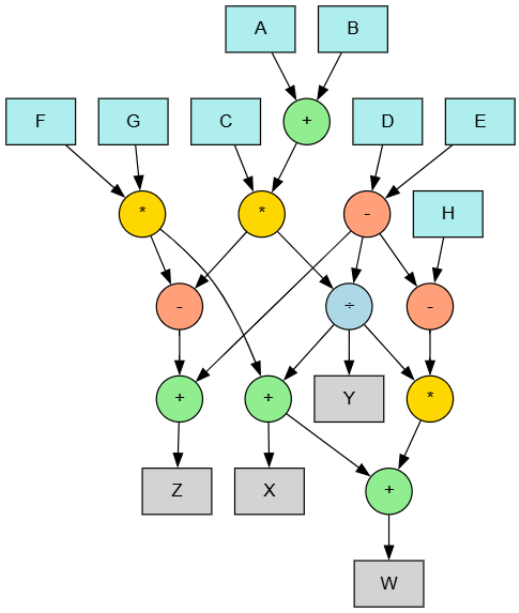

Figure 2.2: A basic dataflow graph example.

DFG as if the load is complete and load-store pairs with the same address form direct forwarding paths in the DFG. The accelerator configuration involves MESA processing the SDFG, sending control bits to the accelerator, and performing the virtual-to-physical mapping of operations to PEs in the CGRA. Time-multiplexing of PEs may occur, and remapping happens depending on the code region's needs. SDFG instances can be duplicated on the CGRA if the loop has minimal dependencies.

After the CPU finishes the current loop iteration, control is transferred to the accelerator before the CPU context-switches to another task. Since the DFG is strictly acyclic, nested loops are not supported, and loops travel in a feedforward manner. If a loop branch occurs, the PEs for skipped instructions are disabled, and backward jumps are not allowed. A table tracking AMAT for each instruction updates calculations every time a code region mapped to the CGRA finishes, allowing for the continual updating of the SDFG as needed.

Given the power of the MESA controller when a specific spatial mapping algorithm is applied, the next logical step is to make the MESA controller programmable to work for any CGRA. Thus, this thesis project is aimed to producing a programmable MESA controller that can perform the referred mapping but to the interconnect structure of various CGRAs. This will

enable the capability of the MESA controller to become more appealing for developers attempting to build a SoC with acceleration involved but utilizing different accelerator designs.

2.2 Instruction Set Architecture (ISA)

The instructions the programmable MESA controller can execute are encoded in a 54-bit format that includes a 6-bit opcode and 16-bits for each source and destination register/field. The instructions have a larger bit-length than well-known ISAs such as RISCV due to the nature of the register data encoding. The MESA controller contains multiple register files, and each register file holds data types with multiple attributes, resulting in a more complex scheme for encoding addresses within the controller. As such, addresses are in 16-bit format in the form of {rtype, index, field}.

MESA Instruction

Bits 53-48: Opcode
Bits 47-32: Destination register
Bits 31-16: Source register 1
Bits 15-0: Source register 2

MESA Register Addressing

Bits 15-13: Rtype *Which MESA register file*
Bits 12-4: Addr *Which address within reg. file*
Bits 3-0: Field *Which field at address within reg. file*

Keep in mind that this multi-part addressing means that one instruction can operate on data across multiple register files at once. The set of instructions for MESA to support this level of versatility are listed below. These instructions were chosen for their possible use in a program that builds an LDFG and SDFG.

Table 2.1: MESA Instruction Set

Opcode	**Operands**	**Description**
add	dst, src1, src2	Arithmetic addition.
addi	dst, src1, imm	
sub	dst, src1, src2	Arithmetic subtraction.
subi	dst, src1, imm	
mv	dst, src	Move contents from source to destination.
mvi	dst, imm (32-bit)	Move immediate value to destination.
ldi (32-bit)	dst, src1, offset	Indirect load (from AXI bus) into *dst*.
ldid (64-bit)	dst, src1, offset	Same as *ldi*, but 64-bit.
sti (32-bit)	src, src1, offset	Indirect store (to AXI bus), *src* holds data.
stid (64-bit)	src, src1, offset	Same as *sti*, but 64-bit.
and	dst, src1, src2	Logical AND.
andi	dst, src1, imm	
or	dst, src1, src2	Logical OR.
ori	dst, src1, imm	
xor	dst, src1, src2	Logical XOR.
xori	dst, src1, imm	
srl/sra/sll	dst, src1, imm	Shift left/right (logical and arithmetic).
rename	dst (gct), src1 (gct), src2	Renames a RISC-V instruction in GCT by replacing operands.
rename.clear		Clears all entries of the Rename Tables.
vmap	src1, vmap_reg_idx	The vMap instruction has multiple variants.
vmap.set	dst, vmap_reg_idx	Sets a register of the vMap unit.
vmap.get		Gets a register of the vMap unit.
vmap.map		Maps a GCT instruction into the SDFG.
vmap.clear		Clears the vMap free matrices.
beq/bez/bge/blt	src1, src2, offset	Conditional branch.
jmp	target	Jump to another instruction.

2.3 Pipeline Stages

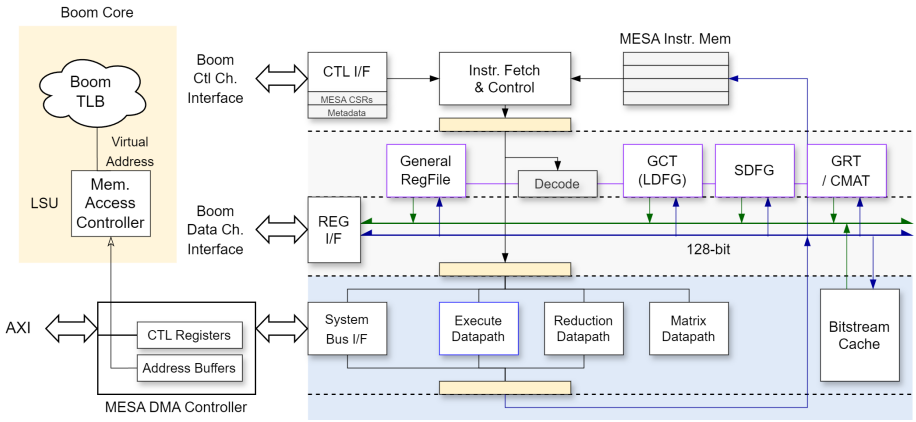

Figure 2.3: Pipelined MESA Controller

The pipeline is responsible for executing instructions that carry out the logic of each of the actions specified in 2.1. The core pipeline consists of fetch, decode, issue, execute, and writeback stages.MESA also has support for flushes and stalls to handle structural, data, and control hazards.

The fetch stage straightforwardly retrieves instructions from MESA instruction memory for execution. In the next cycle, the retrieved instruction is decoded based on the MESA ISA.

Based on the decoded instruction, the issue stage determines what data will be needed for the instruction execution, and retrieves that data from memory or one of the multiple MESA register files. There are three major register files in the controller. The Graph Configuration Table (GCT) stores the SDFG of the CPU code execution. During the MESA config state (see 2.1, CPU instructions are fed into the MESA pipeline (see 2.3), and the GCT logic checks the CPU instruction's validity and whether a loop exists in the code based on the frequency of that specific CPU instruction. The LDFG stores a spatial dataflow graph of the CPU instructions as previously described.

The fourth stage of the pipeline is the execution stage, where there is an ALU to support all opcodes in 2.1. This also includes Advanced eXtensible Interface (AXI) I/O based functionality. The fifth stage is a writeback stage to write data back to our register files and update the pipeline on branch logic commits.

There are three kinds of hazards to handle in a five-stage pipeline. The first is data-dependence hazards. This hazard occurs when a younger instruction is expecting data that is not yet available as it is still being produced by an older instruction still in the pipeline. This can be in the form of a RAW error, where a younger instruction tries to read data that an older one has not written yet, or a WAR error, where a younger instruction wants to write data that an older instruction has to read later. Logic placed at each stage of the pipeline detects for such issues and acts accordingly. These hazards are handled in this pipeline via stalling, where stages before a problematic stage remain still and do not propogate their results forward onto the next stage. These 'problematic' stages are ones that require multiple cycles to complete sometimes, such as execute and issue stages, which either will take multiple cycles to complete an operation that involves the main bus, or reading from a register file or other memory source.

Control hazards occur when a branch or jump needs to occur, but instructions from the wrong sequence of instructions are in the pipeline. BEZ, BEQ, BGE, BLT, JMP, JMPR, JAL are all control instructions that can or will result in a branch in the instruction logic. After the execute stage, the new program counter (PC) is known for instructions after the control instruction is executed, and that PC does not align with the set of instructions being fetched in the fetch stage, the pipeline stages before the execute are flushed (replaced by 'NOPs') and the fetch stage redirects its PC in instruction memory. Typically, a branch predictor can reduce the amount of times this hazard occurs, but one has not yet been implemented in this programmable controller.

Structural hazards occur when there is not enough of a resource for all active instructions at once. For example, multiple transactions could be occuring on one of the controller external interfaces, and so each transaction on the writeback stage to the bus has to be serialized. The same situation also applies to the CSR component, and a stall on the issue stage occurs to avoid a structural fight for the same register bits. There is also transparency incorporated in the register files, so any reads and writes to the same address in a register file on the same cycle can be directly fed over.

CHAPTER 3

RTL VERIFICATION

While writing RTL code, one must continuously verify as they write to ensure no problems arise down the road. For the RTL development process, multiple tools were used, including ModelSim from Intel, AMD Vivado, Verilator, and Chipyard for simulation and compilation of SystemVerilog code.

3.1 Unit-Testing Modules

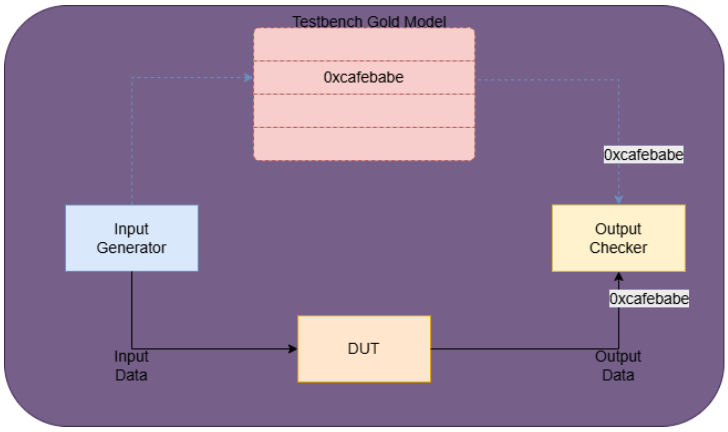

Figure 3.1: Example for verifying register file. Golden unsynthesizable register file called testbench memory gets compared to MESA's Design-Under-Test (DUT) Regfile

The verification of the MESA controller's pipeline was conducted through a unit-testing approach that focused on the key data source modules such as the Spatial Dataflow Graph (SDFG) register file, the Global Control Table (GCT), and the Global Register Table (GRT). Each module was tested individually using a dedicated testbench that comprised three main components: an input generator, the device under test (DUT), and an output checker with a reference model.

The input generator produced a set of test vectors, both deterministic and randomized, to cover the full operational range of the module interfaces. The DUT processed these inputs based on the current register architecture and instruction encoding. The outputs were then compared to those of a pre-verified reference model using an automated comparison mechanism to identify any deviations. See 3.2.

Randomized input patterns were applied to ensure coverage of edge cases and boundary conditions. Each identified error led to a review of the signal behavior and adjustments to the RTL implementation. This iterative process was repeated to verify that each module met its functional specifications under varying conditions.

Throughout the development cycle, the MESA controller changed I/O interfaces, data type definitions, and register configurations. The testbenches were updated accordingly to reflect these modifications. Simulation tools, such as ModelSim, were used to verify the waveforms and output consistency after each design update, ensuring the test infrastructure correctly captured the design changes.

One particular module that required a different framework for verification was a custom AXI module for communication between the MESA controller and CGRA. This line of communication is important for sending the CGRA bitstreams of data on how to configure its resources for the mapped program and sending the translated CPU trace to be accelerated. The module intercepts load and store operations from the MESA controller and converts them into corresponding AXI4 transactions. This conversion involves generating proper AXI signals such as ARVALID, AWVALID, WVALID, RVALID, and BVALID, ensuring that all read and write operations conform to the AXI protocol. An AMD Xilinx Vivado project was established to verify this module, which integrated the MESA controller source files, a custom AXI master module, and an AXI slave IP selected from the Vivado IP Catalog. The AXI slave IP was configured with customized channel widths to match the requirements of the CGRA node. In the project, the MESA controller was instantiated alongside the AXI interface module. Load and store instructions, generated by programs loaded from .lst files into the controller's instruction memory, were executed through the controller pipeline.

During simulation with Vivado Flow Simulator, the custom AXI master converted these instructions into a series of AXI transactions. The test-

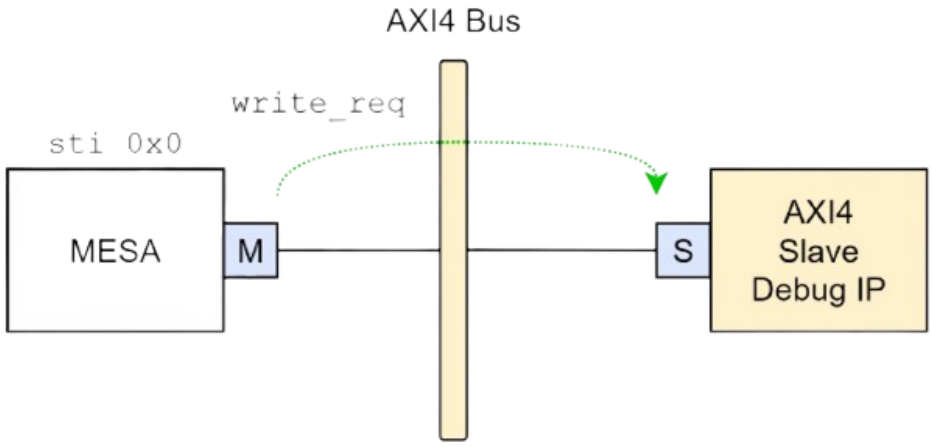

Figure 3.2: AXI Verification Model

bench included additional logic to capture and display detailed information of the AXI signals. This allowed for real-time monitoring of the entire transaction process: from address phase (ARVALID and AWVALID signals) to data phases (WVALID and RVALID) and finally acknowledgement via the BVALID signal.

3.2 Golden Model

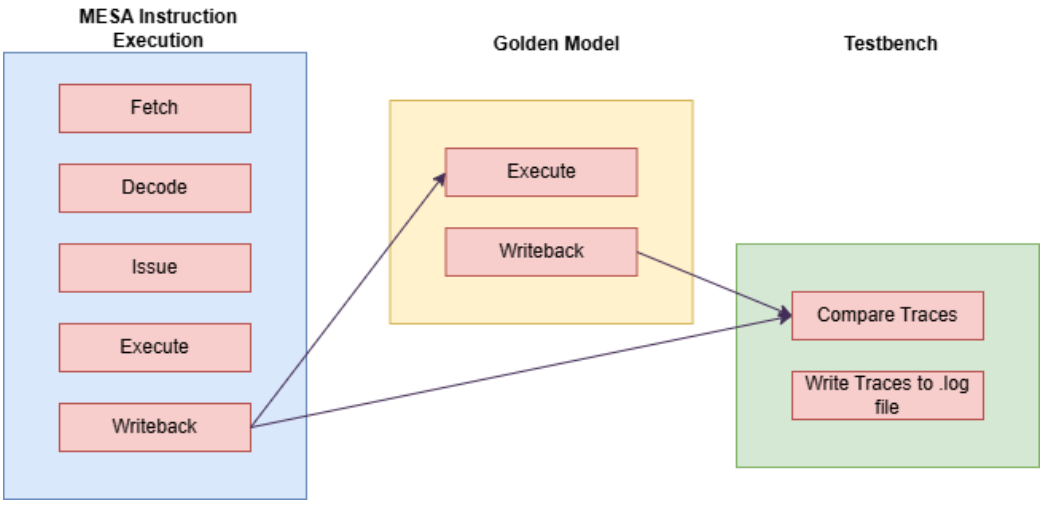

Figure 3.3: Test Assembly Execution and Verification Model

Inspired by the Spike verification framework for RISC-V processors riscv-isa-sim, a "spike-like" verification framework for the custom MESA ISA was implemented. The framework surrounds a testbench that generates randomized MESA assembly instructions with coverage and constraints, which were fed into a golden model of the MESA controller and the RTL model of the official MESA controller, which was the Design Under Test (DUT).

The golden model was written in C++-style SystemVerilog, representing every architectural data source used for graph storage as a register file. A dedicated testbench was written to validate the golden model, with randomized constrained instructions that met coverage. Both the golden model and DUT's signal traces for an instruction's execution (formalized in a MESA formal interface (MFI) equivalent to Spike's RVFI (Risc-V Formal Equivalent) were saved into '.log' files that were then compared. The complete verification flow is illustrated in 3.3.

3.3 Assembly Test Programs

The last part of the verification process is to ensure that complete MESA assembly programs for the target purpose work as intended on the DUT. An assembler was written in Python to take a text file of MESA assembly instructions and compile them into binary, which can be directly loaded onto the MESA instruction memory to be executed. A complete SoC model was built to compile in both Synopsys DC and Chipyard. This model was then fed multiple MESA assembly programs, and the traces of each instruction were manually validated and compared against a golden model.

REFERENCES

[1] D. K. Wang et al., "Mesa: Microarchitecture extensions for spatial architecture generation," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*. Orlando, FL, USA: ACM, jun 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589084 pp. 1–14.

[2] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. Salt Lake City, Utah, USA: ACM, 2014, pp. 269–284.

[3] "CUDA toolkit," 2025, NVIDIA, accessed: 2025-04-19. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[4] "OpenCL: Open computing language," 2025, The Khronos Group, accessed: 2025-04-19. [Online]. Available: https://www.khronos.org/opencl/

[5] "OpenMP api," 2025, OpenMP Architecture Review Board, accessed: 2025-04-19. [Online]. Available: https://www.openmp.org/

[6] NVIDIA, "cuDNN: Cuda deep neural network library," 2025, accessed: 2025-04-19. [Online]. Available: https://developer.nvidia.com/cudnn

[7] "TensorFlow: An end-to-end open source machine learning platform," 2025, TensorFlow, accessed: 2025-04-19. [Online]. Available: https://www.tensorflow.org/

[8] M. A. Watkins, T. Nowatzki, and A. Carno, "Software transparent dynamic binary translation for coarse-grain reconfigurable architectures," in *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. New York, NY, USA: IEEE, 2016. [Online]. Available: https://doi.org/10.1109/HPCA.2016.7446060 pp. 138–150.

[9] R. Bahr et al., "Creating an agile hardware design flow," Stanford University, Technical Report, 2020, unpublished technical report.

[10] J. Melchert, K. Feng, C. Donovick, R. Daly, C. Barrett, M. Horowitz, P. Hanrahan, and P. Raina, "Automated design space exploration of cgra processing element architectures using frequent subgraph analysis," 2021. [Online]. Available: https://arxiv.org/abs/2104.14155

[11] The Regents of the University of California, "The berkeley out-of-order machine (boom)," 2019, accessed: 2025-04-23. [Online]. Available: https://docs.boom-core.org/en/latest/sections/intro-overview/boom.html

[12] F. Liu, H. Ahn, S. R. Beard, T. Oh, and D. I. August, "Dynaspam: Dynamic spatial architecture mapping using out-of-order instruction schedules," in *Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. Portland, OR, USA: IEEE, 2015. [Online]. Available: https://doi.org/10.1145/2749469.2750414 pp. 541–553.

[13] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 389–402.

[14] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley, "An evaluation of the trips computer system," in *Proceedings of the 14th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2009, pp. 1–12.

[15] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," *SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 416–429, jun 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080255

[16] E. Gunadi and M. H. Lipasti, "Crib: Consolidated rename, issue, and bypass," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 23–32.

[17] E. Rotem, Y. Mandelblat, V. Basin, E. Weissmann, A. Gihon, R. Chabukswar, R. Fenger, and M. Gupta, "Alder lake architecture," in *2021 IEEE Hot Chips 33 Symposium (HCS)*. New York, NY, USA: IEEE, 2021, pp. 1–23.

[18] J. Srouji, "Introducing m1 Pro and m1 Max: the most powerful chips apple has ever built," Oct. 2021, Apple Inc., accessed: 2022-10-19. [Online]. Available: https://www.apple.com/newsroom/2021/10/introducing-m1-pro-and-m1-max-the-most-powerful-chips-apple-has-ever-built

[19] V. Govindaraju, C. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, Sep. 2012.

[20] D. K. Wang and N. S. Kim, "DiAG: A dataflow-inspired architecture for general-purpose processors," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 93–106.

[21] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann, "SNAFU: An ultra-low-power, energy-minimal cgra-generation framework and architecture," in *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain: IEEE, 2021, pp. 1027–1040.

[22] N. Serafin, S. Ghosh, H. Desai, N. Beckmann, and B. Lucia, "Pipestitch: An energy-minimal dataflow architecture with lightweight threads," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1409–1422.

[23] G. Gobieski, S. Ghosh, M. Heule, T. Mowry, T. Nowatzki, N. Beckmann, and B. Lucia, "Riptide: A programmable, energy-minimal dataflow compiler and architecture," in *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*, 2022, pp. 546–564.

[24] riscv-software-src, "riscv-isa-sim: Spike, a risc-v isa simulator," Dec. 2021, accessed: 2025-04-23. [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim

[25] F. Elsabbagh, B. Tine, P. Roshan, E. Lyons, E. Kim, D. E. Shim, L. Zhu, S. K. Lim, and H. Kim, "Vortex: Opencl compatible risc-v gpgpu," 2020. [Online]. Available: https://arxiv.org/abs/2002.12151