Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Vetle Kristaver Widnes Harnes

Exploring Efficient Accelerator-Core Integration Strategies

A Case Study of BISMO in Chipyard

Master's thesis in Computer Science
Supervisor: Magnus Själander
Co-supervisor: David Metz
June 2023

NTNU
Norwegian University of
Science and Technology

Vetle Kristaver Widnes Harnes

Exploring Efficient Accelerator-Core Integration Strategies

A Case Study of BISMO in Chipyard

Master's thesis in Computer Science
Supervisor: Magnus Själander
Co-supervisor: David Metz
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Preface

This master thesis builds upon my project from the fall of 2023, which ported BISMO from Chisel 2 to Chisel 3. As such, a small part of the background is taken and modified from the project. As this work is not published, this is not referenced in the text.

Acknowledgements

I express my deepest gratitude to my supervisor, Magnus Själander, and my co-supervisor, David Metz, for their guidance and support throughout this Master's thesis. Magnus Själander provided crucial feedback, encouragement, and ideas contributing to the successful completion of this thesis. David Metz offered technical expertise, insights, and debugging assistance on multiple occasions. His expertise on Chisel and the possibilities and limitations of the Chipyard ecosystem helped me avoid many dead ends and made this research possible.

I would also like to thank my study hall neighbor Håkon Harnes for his insight, discussions, and feedback on the thesis. He also made the long days at the study hall more bearable with chess games and his sometimes overly dry sense of humor.

Lastly, I want to thank my girlfriend and my family for their continuous support throughout my studies and this thesis.

Abstract

In recent years, the end of Dennard scaling and the memory wall has stagnated single-core performance. Hardware designers introduced specialized accelerators that excel in specific workloads to circumvent this. These accelerators enable the Central Processing Unit (CPU) to offload work, improving performance and energy efficiency. To do this, the CPU must supervise the accelerator using custom instructions to offload tasks. How these instructions are generated and communicated to the accelerator can drastically affect performance, dictating delay, implementation overhead, and silicon real-estate usage. Because every implementation has different strengths and weaknesses, the choice of instruction generation method needs to consider the accelerator's use case and bandwidth needs while also considering the implementation burden.

This thesis compares three methods for providing instructions for the Bit-Serial Matrix Multiplication Overlay (BISMO), a bit-serial matrix multiplication Field Programmable Gate Arrays (FPGA) accelerator. The first approach uses software-generated instructions with a register-mapped TileLink (TL) interface to provide instructions. The second implementation is a tightly coupled Rocket Custom Coprocessor (RoCC) accelerator featuring custom RISC-V instructions to provide the software-generated instructions to the accelerator. Lastly, a finite-state machine controlled by a single software-generated instruction provides instructions for the accelerator. The key contribution of this thesis lies in its investigation of various instruction generation and communication techniques within the Chipyard ecosystem. As a by-product, this thesis explores the strengths and weaknesses of a tightly coupled RoCC accelerator versus a more loosely coupled Memory-Mapped I/O (MMIO) accelerator.

These approaches were implemented in the Chipyard ecosystem and integrated into a Rocket Chip System on Chip (SoC) to generate instructions using the Rocket Core. Simulations were run on an Alveo U250 FPGA card. The results show that BISMO with an RoCC integration performs very similarly, but RoCC slightly outperforms the TL MMIO integration in some cases. It also shows that by implementing hardware generation, one can speed up the overall performance of the accelerator with a relatively tiny size and communication delay overhead.

Sammendrag

Ytelsen til enkeltkjerner har stagnert de siste årene, som følge av slutten på Dennard skalering og minneveggen. For å omgå dette, har maskinvare designere introdusert spesialiserte akseleratorer som utmerker seg i spesifikke arbeidsbelastninger. Disse akseleratorene lar prosessoren avlaste arbeid, som effektiviserer ytelse og energibruk. Prosessoren styrer akseleratoren ved hjelp av egendefinerte instruksjoner for å kunne avlaste oppgaver. Hvordan disse instruksjonene genereres og kommuniseres til akseleratoren, kan påvirke ytelsen i stor grad, samt bestemme forsinkelse, implementeringskostnad og silikon arealbruk. Siden hver implementasjon har ulike styrker og svakheter, må instruksjonsgenereringsmetoden imøtekomme akseleratorens bruksområde og båndbreddebehov, og samtidig ta hensyn til implementeringsbyrden.

Denne avhandlingen sammenligner tre metoder for generering av instruksjoner for Bit-Serial Matrix Multiplication Overlay (BISMO), en bit-seriell matrise multiplikasjons Field Programmable Gate Arrays (FPGA) akselerator. Den første metoden er å bruke instruksjoner generert i programvare, med et register kartlagt TileLink (TL) grensesnitt som anvendes for kommunikasjon. Den andre metoden er å implementere en tett koblet Rocket Custom Coprocessor (RoCC) akselerator, som inneholder egendefinerte RISC-V instruksjoner for å kommunisere de programvare-genererte instruksjonene til akseleratoren. Den tredje metoden er å anvende en tilstandsmaskin, styrt av én enkelt programvare-generert instruksjon, til å generere instruksjoner for akseleratoren. Hovedbidraget i denne avhandlingen ligger i undersøkelsen av forskjellige instruksjons-genererings- og kommunikasjonsteknikker innen Chipyard-økosystemet. Som et biprodukt, utforsker også denne avhandlingen styrkene og svakhetene til en tett koblet RoCC-akselerator sammenlignet med en mer løst koblet Memory-Mapped I/O (MMIO) akselerator.

Disse variasjonene ble implementert i Chipyard-økosystemet og integrert i Rocket Chip på en integrert krets for å generere instruksjoner ved hjelp av Rocket kjernen. Simuleringer ble kjørt på et Alveo U250 FPGA-kort. Resultatene viser at BISMO med en RoCC-integrasjon presterer ganske likt, men litt bedre enn TL MMIO-integrasjonen i noen tilfeller. Det viser også at ved å implementere maskinvare-generering, kan man øke den samlede ytelsen til akseleratoren, med en relativt liten størrelses- og forsinkelseskostnad.

Contents

Preface . . . i
Acknowledgements . . . ii
Abstract . . . iii
Sammendrag . . . iv

1 Introduction **1**

2 Background **3**
2.1 Chisel . . . 3
2.2 RISC-V . . . 4
2.3 FPGAs . . . 4
2.4 Bit serial matrix multiplication . . . 5
2.5 BISMO . . . 6
2.5.1 Stages . . . 6
2.5.2 Synchronization . . . 7
2.5.3 Instructions . . . 8
2.5.4 Tiles . . . 8
2.6 Chipyard . . . 8
2.6.1 Rocket Chip . . . 9
2.6.2 TileLink . . . 10
2.6.3 RoCC . . . 11
2.7 FireSim . . . 12

3 Implementation **13**
3.1 MMIO . . . 13
3.1.1 Communication interface . . . 14
3.2 RoCC . . . 15
3.2.1 Communication with the Rocket-core . . . 15
3.3 Hardware generation . . . 17
3.3.1 BISMO version two . . . 17
3.3.2 Implementation . . . 19

3.3.3 FSM 22
3.4 Connecting to memory 27

4 Methodology **30**
4.1 Configuration 30
4.2 Verification 32
4.3 Target Hardware 32
4.4 Evaluation 32
4.4.1 Performance data 32
4.4.2 Memory delay 33
4.4.3 Size usage 33
4.4.4 Benchmarks 34
4.5 Proxy kernel 36

5 Results **39**
5.1 Size usage 39
5.2 MMIO delay 40
5.3 Instruction generation overhead 41
5.4 Cycles spent in states – 256 queue 43
5.5 Cycles spent in states – 16 queue 45

6 Discussion **47**
6.1 RoCC 47
6.2 Hardware generation compared to software generation 48
6.3 Configurations 50
6.4 Integration with the Chipyard ecosystem 50
6.5 Weaknesses 51

7 Conclusion **52**
7.1 Future work 52

Bibliography **53**

A Simulations **59**
A.1 Verilator 59
A.2 Running on an FPGA 60

List of Figures

2.1 Compilation from Chisel code to bitstream/VCD. (figure from [1]) 4
2.2 The BISMOv1 architecture. (figure from [2]) 7
2.3 An example Rocket chip configuration. (figure from [3]) 9
2.4 A master and slave node communication with TL. (figure from [4]) 11
2.5 Example of how a RoCC accelerator communicates with the CPU and the L1 cache. 12

3.1 BISMO TL integration with the Rocket Chip. 15
3.2 BISMO RoCC integration with the Rocket core. The dotted lines represent optional features of a RoCC integration that this thesis does not implement. 16
3.3 Three processors generating instructions for the BISMO accelerator. (figure extended from [5]) 18
3.4 The state machine for the fetch command token generation. 24
3.5 The state machine for the fetch instruction generation. 24
3.6 The state machine for the execute command token generation. 25
3.7 The state machine for the execute instruction generation. 26
3.8 The state machine for the result command token generation. 26
3.9 The state machine for the result command generation. 27

4.1 Amount of instructions generated and communicated for each array dimension. . . . 36

5.1 The overhead of generating instructions. 42
5.2 Speed-up from hardware generation using an Finite State Machine (FSM) versus software generation. 43
5.3 The number of cycles spent in the wait and run stage for the different implementations in the different stages with a queue of size 256. 44
5.4 The number of cycles spent in the wait and run stage for the different implementations in the different stages with a queue of size 16. 46

List of Tables

3.1 RoCC custom command fields. 16
3.2 BISMO command token format. 19
3.3 BISMO fetch instruction format. 19
3.4 BISMO execute instruction format. 20
3.5 BISMO result instruction format. 21
3.6 BISMO Finite State Machine (FSM) instruction. 28

4.1 BISMO parameters used for simulations. 31
4.2 Memory request parameters used for simulations. 31
4.3 The u250 FPGA specifications. 32
4.4 BISMO matrix dimensions used for the tests. 35
4.5 RISC-V page table entry [6]. 37
4.6 RISC-V physical page [6]. 38

5.1 How Lookup Table (LUT) components is used in the different implementations . . . 40
5.2 Total Lookup Table (LUT) overhead of the BISMO accelerator in the different implementations. 40
5.3 Delays of loads and store for TL MMIO and RoCC. 41
5.4 Delays of loads and store in RoCC and TL MMIO in nanoseconds. 41

A.1 The three configurations. 59

Listings

3.1 Register mapping two registers, one read and one write. The first field defines the size of the register, while the second defines the register it is mapped to. 14
3.2 Hardware for-loop construct, where iter, i, j, total_iters, i_max, and j_max is registers, bram is memory and output is the output from the Bundle. 23
3.3 Connecting a TL node to the front bus. 27
3.4 AXI4 to TL conversion with the tools used. 28
4.1 Chipyard configuration with BISMO used in simulations. 31

Acronyms

ASIC	Application-Specific Integrated Circuit
AXI	Advanced Extensible Interface
BISMO	Bit-Serial Matrix Multiplication Overlay
BRAM	Block Random Access Memory
CLB	Configurable Logic Block
CPU	Central Processing Unit
DMA	Direct Memory Access
DPA	Dot Product Array
DPU	Dot Product Unit
DRAM	Dynamic Random-Access Memory
FIRRTL	Flexible Internal Representation for Register Transfer Language (RTL)
FPGA	Field Programmable Gate Arrays
FSM	Finite State Machine
HDL	Hardware Description Language
HLS	High-Level Synthesis
ISA	Instruction Set Architecture
LHS	Left-Hand Side
LUT	Lookup Table
MLP	Memory-Level Parallelism

MMIO	Memory-Mapped I/O
NTNU	Norwegian University of Science and Technology
OPs	Operations per second
PK	Proxy Kernel
PTW	Page Table Walker
QNN	Quantized Neural Networks
RHS	Right-Hand Side
RoCC	Rocket Custom Coprocessor
RTL	Register Transfer Language
SoC	System on Chip
TL	TileLink
TLB	Translation lookaside buffer

Chapter 1

Introduction

In recent years, the end of Dennard scaling [7] and the imminent end of Moore's law [8] have posed significant challenges for the hardware industry. The ever-growing demand for computational power has been increasingly difficult to meet through conventional scaling of transistor sizes, leading to a paradigm shift towards the development of specialized hardware accelerators [9]. These accelerators offer energy-efficient and high-performance solutions by tailoring the hardware to specific application domains, tackling the limitations posed by the stagnation of single-core performance.

Artificial intelligence has experienced remarkable advances, thanks in part to the success of accelerators [10]. Parallel execution of matrix multiplications and large datasets have rendered performance and energy efficiency paramount. As one of the most trending research subjects, AI witnessed the publication of approximately 335,000 papers in 2021 alone [11]. Applications such as ChatGPT [12] have attracted significant public and scientific attention.

Quantized Neural Networks (QNN)s have gained significant attention in the artificial intelligence community, particularly for their efficiency in resource-constrained environments such as edge devices and embedded systems. By employing lower bit-width representations for weights and activations, QNNs substantially reduce both memory footprint and computational complexity, making them ideal for low-power applications without compromising performance [13]. Bit-serial matrix multiplication is a technique that can be leveraged by QNNs, enabling efficient computation of matrix products with reduced hardware resources. This approach serializes the input operands and processes them bit by bit, resulting in simpler and smaller hardware structures [14].

Bit-Serial Matrix Multiplication Overlay (BISMO) [5, 2] is a bit-serial matrix multiplication accelerator developed by Norwegian University of Science and Technology (NTNU) in 2018. Bit-serial operations enable BISMO to control the desired precision in software while performing matrix multiplications, the main mathematical operation in convolutional neural networks. BISMO has yielded promising results when benchmarked on Field Programmable Gate Arrays (FPGA)s. BISMO was ported to Chisel 3 in autumn 2022 and relies on software to generate its instructions. Yaman et al. had previously optimized this version of BISMO and found that software instruction generation is a bottleneck and proposed High-Level Synthesis (HLS) as a solution. This yielded great results and indicated that software-generated instructions for a Memory-Mapped I/O (MMIO) accelerator might not be optimal. This optimized version is written in Chisel 2 and is incompatible with the Chisel 3 ecosystem.

Chipyard [15] is an open-source framework for designing, verifying, and evaluating SoC designs. Chipyard provides a comprehensive, integrated platform that streamlines the design and verification process by offering a unified environment for various tools and methodologies. It enables designers to rapidly prototype and test different accelerator configurations, reducing development time and ensuring the resulting designs meet the desired specifications. In conjunction with Chipyard, FireSim [16], an FPGA-accelerated cycle-accurate simulation platform, has been developed to enable fast and accurate simulation of complex SoC designs. Using FireSim, designers can efficiently verify and validate their SoC designs, mitigating the risk of functional or performance issues in the final implementation.

The main contribution of this thesis is the comparison of the BISMO accelerator with software-generated instructions, using a tightly coupled Rocket Custom Coprocessor (RoCC) and a loosely coupled TL accelerator with a hardware generation approach. The advantages and disadvantages of MMIO and tightly coupled RoCC integrations in Chipyard will be discussed as a byproduct. Another contribution of this thesis is the integration of BISMO into the Chipyard ecosystem, which makes advanced simulations and research more accessible. The result shows little difference between the RoCC and MMIO integrations for a software generation approach. This indicates that the flexibility of the MMIO accelerator puts it a cut above RoCC, especially for larger workloads. When looking at hardware versus software generation, the performance difference was mostly minuscule, with hardware generation being the superior option for most, if not all, workloads.

In this master thesis, the content is organized into six chapters. The background section provides essential context and pertinent literature, establishing a solid foundation for the study. The implementation section details the BISMO implementations in Chipyard, elaborating on the system's development process and key aspects. The methodology section outlines the research design, data collection, and analytical procedures used to ensure the reliability and validity of the study findings. The result section presents the empirical results obtained from the research, offering a systematic data account. The discussion section analyzes the results, interpreting the findings and exploring their implications. The conclusion section will summarize the findings and contributions and present future work.

Chapter 2

Background

This chapter will present the main concepts behind BISMO, Chipyard, Firesim, and some of the tools that enable them. First, some of the tools and their motivation will be presented. Then bit-serial matrix multiplication and BISMO will be explained. Lastly, the relevant parts of the Chipyard ecosystem and Firesim will be introduced.

This master's thesis builds upon my project from the fall of 2023. As such, a small part of this section is taken and modified from the project. As this work is not published, this is not referenced in the text.

2.1 Chisel

A Hardware Description Language (HDL) is a programming language used to describe digital and analog circuits. HDLs was developed as a response to the increasing complexity of circuit technology in the 1970s, as designers wanted high-level digital logic without being tied to a specific technology [17]. The two primary HDLs are Verilog and VHDL. Verilog and VHDL were initially developed as simulation languages and later applied to hardware development. Since these languages were not initially developed for hardware development, they contain many unsynthesizable constructs, which can introduce ambiguity.

Chisel [18] is a HDL developed by UC Berkley in 2012. The idea behind Chisel is to take advantage of the powerful abstraction facilities in modern-day programming languages to make it easier to reuse components and increase productivity. Chisel is embedded in Scala. It is not directly involved in simulation but is used to generate Verilog that can be simulated using other tools. This enables the generation of high-speed C++-based cycle-accurate software simulators. Chisel also makes it easy to generate low-level Verilog that can be mapped to FPGAs or standard ASIC for synthesis. Chisel compiles down to Flexible Internal Representation for RTL (FIRRTL) [19], simplifying, verifying, transforming, and emitting the circuit.

Verilator [20] is an open-source software tool that converts Verilog code into a cycle-accurate behavioral model. Verilator ignores everything between clock edges and is capable of high-speed simulations. Since Verilator is a cycle-based simulator, it cannot track exact circuit timings or replace all features of other event-based simulators [21]. An overview of the Chisel pipeline can be found in figure 2.1.

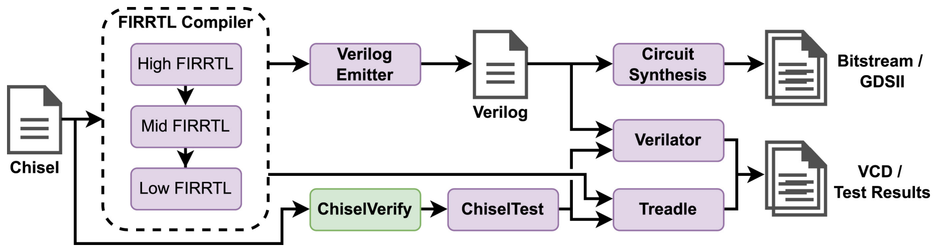

Figure 2.1: Compilation from Chisel code to bitstream/VCD. (figure from [1])

2.2 RISC-V

RISC-V is an open-source Instruction Set Architecture (ISA) developed at UC Berkley in 2016 [22]. An ISA is part of the abstract computer model and interfaces hardware and software. It defines the different operations the processor can execute and how they are executed. It is the only way that a user can interact with hardware. Although there exist other open-source ISAs, like SPARC [23] and OpenRISC [24], Waterman [22] found that all are severely lacking compared to what one could expect from a modern-day ISA. Other more complete ISA's like ARMv8 [25] and x86 [26] are proprietary, which means you need a license to modify or implement it [27]. RISC-V aims to create an open-source ISA that can be used on everything from embedded platforms to state-of-the-art processors. RISC-V has recently gained a lot of traction, with more than 3,100 members ranging from large companies to academic institutions [28] and significant investments from companies such as Intel [29]. The Chipyard ecosystem and its generators are based on the RISC-V ISA.

2.3 FPGAs

Field Programmable Gate Arrays (FPGA)s enable you to implement digital circuits and systems by configuring the chip's programmable logic blocks and routing resources. FPGAs are highly flexible and can be reprogrammed multiple times, enabling the implementation of various designs and functions on a single chip. This flexibility makes FPGAs an ideal solution for low to medium-volume productions, where time to market is critical, and enables easy design updates and changes without fabricating a new chip. FPGAs also offers a higher level of customization than Application-Specific Integrated Circuit (ASIC), which cannot be reconfigured and is designed for a specific application. Additionally, FPGAs can be used for prototyping and testing before committing to a final ASIC design, reducing the risk of errors and improving the quality of the final product [30].

2.4 Bit serial matrix multiplication

In 2006 researchers at Berkeley University identified 13 computational patterns that represent classes of tasks commonly found in computational applications [31]. The general idea behind these dwarves is that focusing on these common patterns makes it possible to create optimized software and hardware configurations that can effectively handle various applications. Dense Linear Algebra, and by extension Matrix multiplication, is one of the thirteen Berkley dwarves, which makes it an excellent avenue to explore for performance.

Umuroglu and Jahre [5] showed that matrix multiplications could be expressed as a weighted sum of large numbers of binary matrix multiplications. This enables computing parallel matrix multiplications in a bit-serial fashion while overcoming the typically poor performance and high latency. The bit-serial matrix multiplication algorithm can be seen in algorithm 1.

Algorithm 1 Bit serial matrix multiplication

```
Input: m × k l-bit matrix L, k × n r-bit matrix R
Output: P = L · R
  for $i \leftarrow 0...l-1$ do
    for $j \leftarrow 0...p-1$ do
      $sgnL \leftarrow (i == l-1\ ?\ -1 : 1)$
      $sgnL \leftarrow (j == p-1\ ?\ -1 : 1)$
      $weigth = sgnL \cdot sgnR \cdot 2^{i+j}$
      Binary matrix multiplication between $L^{[i]}$ and $R^{[j]}$
      for $r \leftarrow 0...m-1$ do
        for $c \leftarrow 0...n-1$ do
          for $d \leftarrow 0...k-1$ do
            $P_{rc} = P_{rc} + weight * (L^{[i]}_{rd} \cdot R^{j}_{dc})$
```

As shown in equation (2.1), a matrice can be expressed as n additions of shifted matrices, where n is the precision of the value with the highest precision requirement. In this case, the most significant value, 3, can be represented as 2 bits, which requires two shifted additions.

$$\begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix} = 2^1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + 2^0 \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \tag{2.1}$$

As an extension, the multiplication of two matrices, L and R, can be expressed as a weighted sum of products between binary matrices. This can be seen in equation (2.2), where two 2 bits matrices L and R, are multiplied. In this example, when L and R are raised to a power, it denotes the bit shift of the matrix. L is what will be refered to as the Left-Hand Side (LHS) matrix, while R is the

Right-Hand Side (RHS) matrix.

$$\begin{aligned} P &= L \cdot R \\ &= (2^1 L^{[1]} + 2^0 L^{[0]}) \cdot (2^1 R^{[1]} + 2^0 R^{[0]}) \\ &= 2^2 L^{[1]} \cdot R^{[1]} + 2^1 L^{[1]} \cdot R^{[0]} + 2^1 L^{[0]} \cdot R^{[1]} + 2^0 L^{[0]} \cdot R^{[0]} \end{aligned} \tag{2.2}$$

2.5 BISMO

Deep neural networks have proven well suited for image recognition [32, 33, 34], self-driving cars [35], processing tasks, among other things. While deep neural networks yield good performance results, their size prevents them from being widely used. As a response, QNNs were developed to address the model size. QNNs aim to reduce models' size while sacrificing as little performance as possible. Park et al. [36] found that the model size can be significantly reduced by reducing bit precision in intermediate layers, barely losing any precision, and improving energy efficiency. While promising, existing hardware and FPGA based matrix multiplication accelerators usually operate with fixed precision [37, 38, 39], making them unable to leverage the frugality of a bit-serial approach.

Bit-Serial Matrix Multiplication Overlay (BISMO) is a variable precision bit-serial matrix multiplication FPGA accelerator developed by Umuroglu et al. [2]. BISMO is a reprogrammable matrix multiplication accelerator that seeks to leverage the fact that many computations use variable bit precision in different phases of the application. BISMO currently has two implementations, which will be referred to as BISMOv1 [5] and BISMOv2 [2]. BISMOv1 is the original implementation of BISMO and uses software-generated instruction to operate the different stages. BISMOv2 is an optimized version of BISMOv1, with an improved Dot Product Unit (DPU) and uses HLS with a single instruction to generate instructions for the different stages. BISMOv2 is currently written in Chisel 2, which makes it incompatible with the Chipyard ecosystem.

2.5.1 Stages

BISMO is divided into three main stages, fetch, execute, and result, as seen in figure 2.2.

Fetch stage

The fetch stage performs fetches from the main memory to provide data for the execute stage. The Fetch stage contains a Direct Memory Access (DMA) engine, a route generator, and a linear array interconnect [5]. The fetch stage uses a modified Advanced Extensible Interface (AXI) interface to fetch data from the main memory, where the route generator sends read requests and decides where the data should be written.

Execute stage

The execute stage utilizes an array of DPU's to perform the matrix multiplication on data stored in the matrix buffers. Each DPU receives a configurable number of bits from the LHS and RHS matrix buffers. The DPUs in the same row of the data processing matrix receive the same data from the LHS buffer, while the DPUs in the same column receive the same data from the RHS buffer. A software-controllable sequence generator determines which data are read from the matrix buffers. The execution stage can be easily scaled by adjusting the number of rows and columns in the Dot Product Array (DPA).

The DPU pipeline computes a partial result in the dot product operation between a row and column of two-bit matrices. Single-bit multiplications are performed using a multi-bit logic AND operation, and the summation is obtained through a pop count of the result. A unit of left shift implements the weight of the calculation. It can be optionally negated, with both operations being controlled by software. The partial results are accumulated and stored in a register.

Result stage

The result stage writes the results generated by the execution stage to the main memory, using a stream writer with a downsizer and a DMA engine for resizing and striding. This setup allows for overlapping computations and data transfers between stages, enabling independent operation and scalability to match a platform's memory write bandwidth.

2.5.2 Synchronization

Since BISMO consists of three individual stages that perform memory operations with unpredictable timings, it needs synchronization. Synchronization lets stages communicate their current state. These states are running, waiting for data, or finished operating. Synchronization instructions coordinate two separate pipeline stages through signal and wait for instructions. The signal instruction generates a token for the relevant synchronization FIFO while the wait instruction pauses until a token is obtained. The synchronization FIFO is designated exclusively for the execution stage in both the fetch and result stages. Consequently, the execution stage maintains two synchronization FIFOs to align with the fetch or result stage, as seen in figure 2.2. The tokens themselves do not hold specific information, granting programmers the flexibility to assign meaning to each synchronization, such as denoting a matrix buffer's full or empty state.

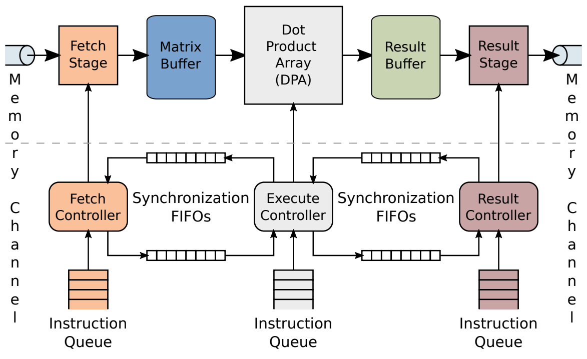

Figure 2.2: The BISMOv1 architecture. (figure from [2])

2.5.3 Instructions

Every operation that BISMO performs needs instructions to tell it where to get the data, what to do with it, and where to put it. These instructions need to be generated in some way. The original BISMO paper used software-generated instructions to run the accelerator. These instructions were generated by the Central Processing Unit (CPU) and communicated with the accelerator through the interconnect system. Although it is not mentioned in the original BISMO paper [5] or in the article on the optimized version of the architecture [2], the inclusion of instruction generation through HLS implies that this generation scheme induced a bottleneck through either bandwidth issues, the CPU being able to generate instructions fast enough, or the CPU being unable to perform other tasks due to cycles spent generating instructions.

Since the accelerators need to receive instructions through the interconnect system and are consumed at different speeds during execution, it uses an internal queue to store the instructions. This lets the processor fill the queues with instructions during downtime, improving performance.

2.5.4 Tiles

The different stages in BISMO operate on data segments of different sizes. These segments will be referred to as tiles in this thesis. The tiles in BISMO are divided into three levels, the L0, L1, and L2 tiles. The L2 tile is the largest tile, which is what a single fetch instruction fetches into its memory. Each L2 tile is divided into L1 tiles, the tile size that the execute stage can read into its memory. L1 tiles are divided into L0 tiles, the size of the tiles the DPA uses for operation.

2.6 Chipyard

Chipyard [15] is an open-source framework that simplifies designing, simulating, and testing custom hardware accelerators, processors, and SoCs. Developed by researchers at Berkeley University, Chipyard provides a robust ecosystem that integrates various hardware generators, software tools, and libraries, enabling developers and researchers to create and evaluate custom hardware designs.

One of the key components of Chipyard is the Rocket Chip generator [3], a highly configurable and parameterizable RISC-V processor generator. The Rocket Chip generator can create various processor designs, from simple single-core configurations to more complex multicore systems. This flexibility enables users to explore design trade-offs such as power consumption, performance, and area usage.

Chipyard also includes software simulation and hardware emulation support through tools like FireSim, enabling users to simulate their designs on FPGAs at near-cycle-accurate speeds. This capability enables rapid design space exploration and early validation and testing of custom hardware designs before fabrication.

2.6.1 Rocket Chip

The Rocket Chip generator [3] is a sophisticated open-source hardware design framework developed by the University of California, Berkeley, as part of the RISC-V project. The Rocket Chip generator is primarily intended to facilitate the rapid development and customization of RISC-V based SoC designs, enabling researchers and industry professionals to create tailored solutions for various applications. The Rocket Chip can be seen in figure 2.3.

The Rocket Chip generator is made up of several components, including the Rocket core, a high-performance, in-order, single-issue RISC-V processor, and the TL coherent interconnect protocol, which is responsible for connecting multiple processor cores, caches, and memory controllers within the SoC. The generator also includes various peripherals, such as UART, SPI, and GPIO controllers, as well as debug and trace infrastructure. The Rocket Chip generator can produce a wide range of SoC designs optimized for performance, area, or energy efficiency by parameterizing these components and enabling users to configure their attributes.

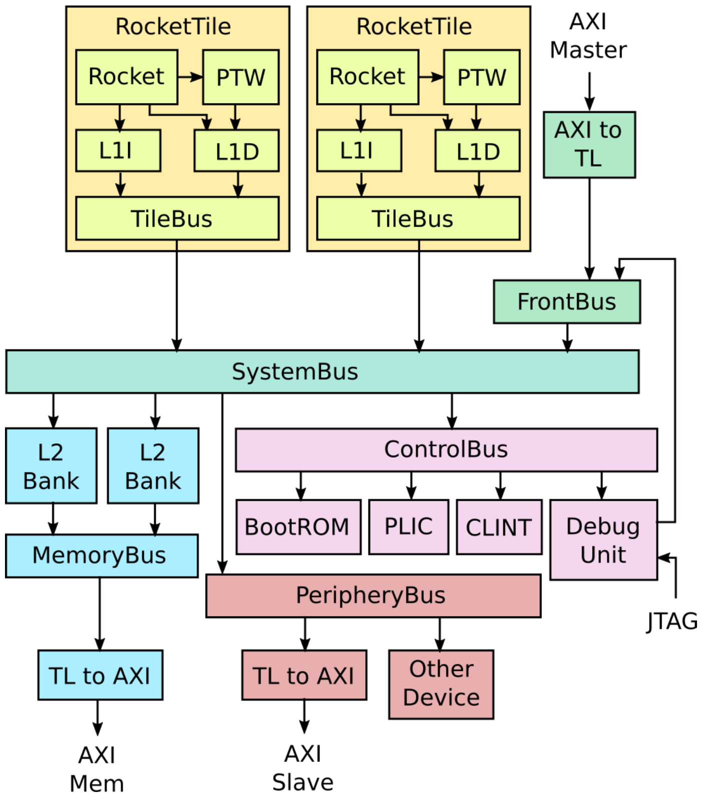

Figure 2.3: An example Rocket chip configuration. (figure from [3])

2.6.2 TileLink

TileLink (TL) [4] is an open-source, cache-coherent, and scalable on-chip communication protocol between processing elements and memory in chip design. SiFive developed it to address the challenges of designing complex SoC with multiple processing cores, accelerators, and memory controllers. In TL, each processing element or memory controller is represented as a tile connected to a network of other tiles through channels. The channels transmit messages between the tiles, representing each as a transaction.

There are four types of TL transactions:

1. Get: A Get transaction is sent by a processing element to request data from memory.
2. Put: A Put transaction is sent by a processing element to store data in memory.
3. Acquire: An Acquire transaction is sent by a processing element to request ownership of a cache line in memory.
4. Release: A Release transaction is sent by a processing element to release ownership of a cache line in memory.

TL consists of five channels, A, B, C, D, and E, where channels B, C, and E are used for TL-C only, while channels A and D are mandatory. Since this thesis does not use TL-C, these three channels will not be explained. These channels use a decoupled interface to communicate the state of current transmission. A typical master-slave communication can be seen in figure 2.4.

Channel A requests memory operations and flows from the master to the slave interface. A request operation is sent to a specific address range. It can be a read or a write operation, depending on what the `code` bitfields contains.

Channel D is for the response of a memory operation and flows from the slave interface to the master. Channel D carries the response from another operation. In the case of a memory read, it has the data and some additional metadata information about the operation. In the case of a write, it only carries the metadata.

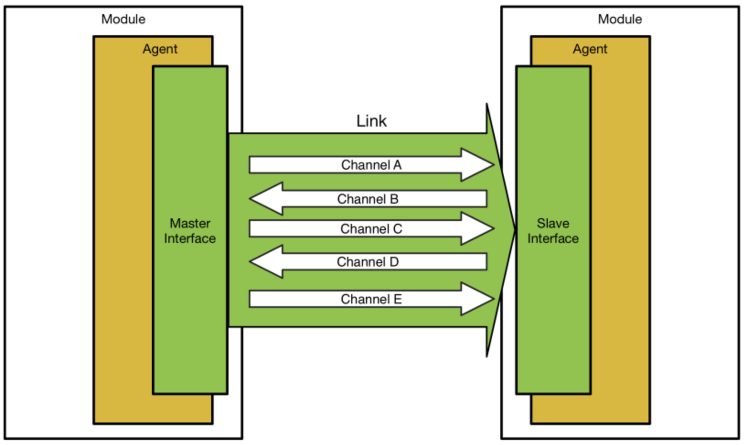

Figure 2.4: A master and slave node communication with TL. (figure from [4])

2.6.3 RoCC

The Rocket Custom Coprocessor (RoCC) interface in Chipyard enables the integration of custom accelerators. It is a communication channel between the Rocket CPU and the custom accelerator, enabling them to work together seamlessly. Developers can use the RoCC interface to create and integrate application-specific accelerators in their Chipyard designs.

The RoCC interface operates with custom instructions explicitly defined for the accelerator. These instructions provide a standardized way for the Rocket CPU to communicate and control the accelerator. The RoCC interface includes signal connections and handshake protocols to facilitate data transfer and instructions between the Rocket processor and the accelerator. The accelerator can offload specialized tasks, such as encryption [40], image processing, and machine learning [41], from the Rocket CPU, improving overall system performance.

One of the key benefits of the RoCC interface is its flexibility and extensibility. It enables developers to design and integrate various custom accelerators tailored to their application needs. The RoCC interface abstracts away the complexities of the CPU internals, providing a standardized interface for accelerator integration. Designers can instead focus on developing the accelerator's functionality and performance without worrying about low-level implementation details. The RoCC interface and the Chipyard framework provide a powerful platform to explore and evaluate custom accelerators, promoting innovation and customization in RISC-V-based SoC designs. An example of how a simple accelerator would be integrated with the RoCC interface can be seen in figure 2.5.

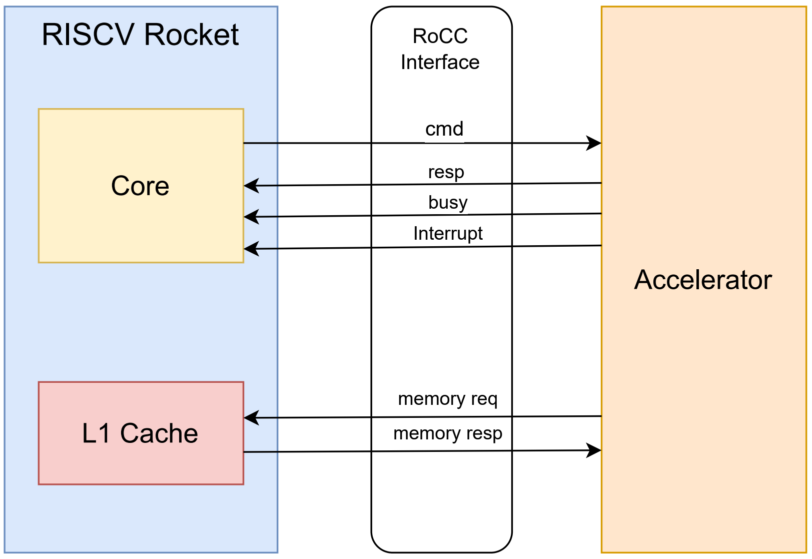

Figure 2.5: Example of how a RoCC accelerator communicates with the CPU and the L1 cache.

2.7 FireSim

Firesim [16] is an open-source, FPGA-accelerated cycle-accurate hardware simulation platform. It lets developers simulate and prototype custom hardware designs, including processors, networks, and accelerators, on a large scale using cloud-based FPGAs. FireSim is particularly useful for research and development in computer architecture and systems, enabling rapid prototyping and evaluation of new designs.

FireSim simulates hardware by translating target designs in a hardware description language such as Chisel or Verilog into a cycle-accurate FPGA model. This model can be programmed on an FPGA, emulating the target design with high fidelity. This FPGA-based approach offers high performance and cycle-accurate simulation, making it a powerful tool for exploring hardware design.

FireSim uses a token-based system to ensure memory timings in FPGA simulations. In a token-based memory system, requests and responses are passed as tokens between the memory subsystem and the simulated core. Each token represents a memory operation unit, such as a read or a write. The core and the memory subsystem exchange the tokens in a manner that accurately models the target memory system's timing characteristics.

The token-based system enables FireSim to simulate complex memory systems with realistic timing behavior. By adjusting the number of tokens and the timing at which they are exchanged, FireSim can model different memory subsystems, from simple single-cycle memories to more complex cache hierarchies and memory controllers. This flexibility is crucial in accurately simulating hardware designs that rely on specific memory timing characteristics.

Chapter 3

Implementation

This chapter presents three different integrations of the BISMO accelerator with Chipyard. Section 3.1 presents an integration that utilizes an MMIO interface, which uses TL for communication between the processor and the accelerator. Section 3.2 presents an integration that employs Chipyard's RoCC interface, enabling BISMO to be tightly integrated with the processor pipeline. Section 3.3 presents the instruction patterns of BISMO and how these are leveraged to construct a Finite State Machine (FSM) to explore a hardware-based approach. Section 3.4 presents how communication with main memory was handled in all three integrations. All the work and forks are hosted on Github[1][2][3].

3.1 MMIO

One way of coupling an accelerator with an SoC is through a loosely coupled MMIO device. Loosely coupled accelerators reside outside the core, using the interconnect system for communication. Employing this loose connection yields advantages such as ease of integration and portability. The accelerator residing outside the core also enables it to use a larger area, with the penalty of a longer communication delay and taking up bandwidth in the interconnect system. Chipyard employs the open-standard TL protocol for MMIO mapping. This is achieved by mapping the accelerator to an address and communicating with the accelerator through reads and writes on the peripheral bus. The Chipyard ecosystem has integrated register-mapped MMIO accelerators like the NVDLA deep learning accelerator [41] and the Greatest Common Denominator accelerator, which serves as examples of how this can be performed. The original BISMOv1 accelerator used an MMIO interface for communication but with an AXI interface. The TL implementation can hence be considered highly similar, if not identical, in performance to the original, thus serving as a good baseline.

[1]https://github.com/Vetleh/chipyard
[2]https://github.com/Vetleh/BISMO_Chisel_3
[3]https://github.com/Vetleh/rocket-chip

3.1.1 Communication interface

The BISMO architecture has two primary interfaces that need to be integrated with Chipyard. BISMO needs to receive control signals in the form of instructions from the Rocket Chip and be able to read/write data from/to the main memory. How communication to main memory is implemented will be discussed in section 3.4.

Control signals

The control signals must be exposed to the user so that they can control accelerator operations and states. It also enables performance data collection to benchmark, troubleshoot, and locate avenues for improvement in the architecture. MMIO accelerators in Chipyard have two main ways of achieving this, register mapping with AXI or TL. Since TL is the memory protocol used by the Rocket core and other Chipyard generators, using this over the AXI interface offers a more seamless integration with the ecosystem since it does not require an intermediate converter layer to function.

Exposing an MMIO accelerator to the CPU can be done using the built-in TLRegisterRouter widgets, which remove the details of the handling of interconnect protocols and provide a convenient interface for specifying memory-mapped registers [42]. An example of how register mapping can be achieved on an instantiated TL node can be seen in listing 3.1.

```
node.regmap(
  0x00 -> Seq(
    RegField.r(1, is_ready)
  ),
  0x04 -> Seq(
    RegField.w(1, start)
  )
)
```

Listing 3.1 Register mapping two registers, one read and one write. The first field defines the size of the register, while the second defines the register it is mapped to.

A TL register node and a device must be instantiated to create a memory mapping for the accelerator. On creation, the node takes an address, where it will reside in physical memory and a mask that describes the size of the address space. The address `0x1000` and the mask `0xfff` were used for this. This makes the accelerator take up the addresses from `0x1000 - 0x1fff`. Then, all the different register sizes and offset addresses must be defined and connected to the accelerator. Depending on the needs, the registers can be implemented as read, write, or read/write, and must be aligned to 4 bytes. Once all desired communication registers are connected, the created node can be connected to the peripheral bus, which enables it to communicate with the rocket chip. BISMO integrated into the Rocket Chip with TL can be seen in figure 3.1.

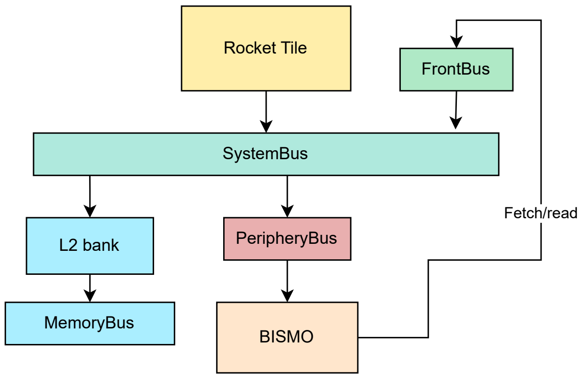

Figure 3.1: BISMO TL integration with the Rocket Chip.

3.2 RoCC

Another way to connect an accelerator to the Rocket core is through the RoCC interface. RoCC accelerators are tightly integrated into the processor pipeline. They can extend the capabilities of the core by offloading specific computations or tasks to the custom accelerator, improving performance and power efficiency. This tight coupling enables lower latencies, less software overhead, and no bandwidth usage on the cross-bar network. This comes at the cost of a more complex design, less scalability, and less flexibility since a custom software toolchain is required for RoCC, while MMIO can use the standard toolchain with driver support [43].

A tightly coupled RoCC accelerator also exposes the accelerator to the rocket cores' features. This includes access to the L1 cache, the page table walker, and two TL output nodes. One of the TL nodes connects to a tile-local arbiter along with the backside of the L1I cache. On the contrary, the other is directly connected to the L1-L2 crossbar [44]. Using the page table walker, the accelerator can work directly with virtual addresses supplied by the processor. Working with virtual addresses allows for easier integration with other hardware while removing the issue of allocating continuous physical memory. While this is a significant advantage, the translation from physical to virtual memory gives additional latency, and an efficient design would most likely require caching addresses. Therefore, this would damage performance and make it less comparable to the other implementations, so it was not implemented.

3.2.1 Communication with the Rocket-core

The RoCC interface permits a tight coupling with the Rocket core by utilizing a custom protocol and non-standard ISA instructions reserved within the RISC-V ISA encoding space. Each core can have up to four different tightly coupled RoCC accelerators that reside close to the core. The communication happens through custom opcodes on the form

```
customX rd, rs1, rs2, funct
```

The size and meaning of each can be seen in table 3.1. `customX` will be a number in the range that defines which of the up to four accelerators the custom instruction belongs. It is also supported for a single accelerator to use more than one of the `customX` fields. This would, however, limit the number of other accelerators that can be used on the same core. The `rd`, `rs1`, `rs2` defines the register destination and the two register sources for the operation, while the `funct` field allows for differentiating the different instructions. However, these fields can be applied in any way that is needed. They need not retain their original meaning unless they want to follow the RISC-V instruction set specification [45].

Name	size	Description
rs1	64	The value of source register 1.
rs2	64	The value of source register 2.
rd	5	Destination register.
funct	7	Type of instruction.

Table 3.1: RoCC custom command fields.

In order to conduct communication between the rocket core and the accelerator, a register mapping approach was chosen. This was considered the most logical option because it reduced the amount of existing software that needed to be changed due to its similarities with the TL MMIO implementation. Creating complicated RoCC instructions would yield few advantages and does not make much sense without modifying the existing BISMO interface. A vector of registers is initialized in Chisel with `Reg(Vec(outer.n, UInt(xLen.W)))`, which then can be indexed. In this case, `outer.n` is the number of registers needed, and `xLen` is the size of the `rs1` field. BISMO tightly coupled with the Rocket CPU pipeline can be seen in figure 3.2.

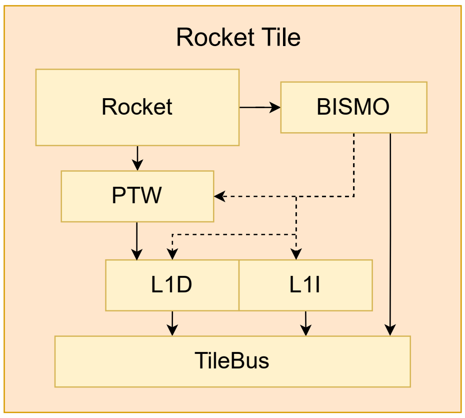

Figure 3.2: BISMO RoCC integration with the Rocket core. The dotted lines represent optional features of a RoCC integration that this thesis does not implement.

3.3 Hardware generation

BISMOv1 uses software to generate instructions for the three stages. This means that the CPU needs to generate instructions and send them through the interconnect system to the accelerator. If the accelerator can perform more instructions than the interconnect system can support, this could cause a bottleneck, making it memory-bound. However, if the processor cannot generate the instructions fast enough, it could also create a bottleneck, making it compute-bound. These problems can arise in some configurations but not others, making it hard to assess the accelerator's performance in different implementations properly. A solution to this could be generating instructions in hardware.

3.3.1 BISMO version two

When Yaman et al. [2] looked at potential avenues of improvement in the BISMO architecture, instruction generation was identified as a candidate. The proposed solution was to use HLS [46] to generate the hardware that generates the instructions. Although this solution yielded promising results, HLS can be considered a suboptimal solution for two reasons.

Firstly, using multiple hardware generation tools in a single project can make the project hard to maintain. Chisel already has a steep learning curve, and due to the low number of users, it can lack beginner-friendly documentation and examples. Combining the low-level hardware-focused approach of Chisel with the more algorithmic approach of HLS can also cause confusion and suboptimal solutions.

Secondly, effectively generating hardware with an algorithmic approach is complex and requires a hardware mindset. Edwards [47] identified five main software-to-hardware challenges when using HLS tools.

1. The software executes sequentially while the hardware runs concurrently. This can create issues when mapping sequentially generated algorithms onto concurrent hardware.

2. Software implicitly manages timing through instruction sequences. On the contrary, synchronous hardware must handle timing constraints and synchronize clock cycle-level operations.

3. The software uses fixed word length, while the hardware is optimized according to the task.

4. Software memory models use a unified address space for storing data. In an FPGA, local variables are kept in registers in distinct memory blocks with individual address spaces. This makes pointers less practical and dynamic memory allocation challenging.

5. In software, communication is typical through shared memory, while in FPGAs, it involves building suitable hardware for everything from stream processing and token passing to using dedicated FIFOs for flow control.

These challenges make it difficult to know how the implementations map to hardware. It is easy to fall into a software mindset when applying HLS, which can significantly affect the size and performance of the synthesized hardware [48].

Three implementations were considered to respond to the potential pitfalls of HLS. The first was connecting a small RoCC processor to each stage that could generate the instructions, which can be seen in figure 3.3. The second implementation considered involved using a single processor to generate instructions for all stages. The last was in the form of a FSM. Implementing multiple processors would offer advantages since it is more flexible, user-configurable, and can be given custom instructions. There were some issues with this option.

Designing a minimal processor with only the features needed for the instruction generation of each stage would be very time-consuming and challenging. It would require a complete rewrite of all the tests and would also have to meet timing constraints. Using the existing Rocket CPU to do this as a proof of concept that could be benchmarked was considered a solution. The issue with using the Rocket CPU is that Chipyard represents Rocket CPUs as a tile. Since a Rocket CPU used for instruction generation would also be represented as a tile, it would require a tile within a tile, and it was unclear whether this is even possible. Since this complicated implementation could potentially lead to a dead end, it was scraped. There is also the issue that the instructions to be generated are relatively simple. Removing all the unnecessary processor constructs could result in a more complicated pipelined state machine. Thus, using a FSM for instruction generation seemed like the most realistic and easy-to-implement solution, although representing nested loops in Chisel can be complex. While creating a finite state machine with Chisel does not guarantee an optimal solution, it gives a more hardware-based approach which can be easier to look into for further avenues of optimization.

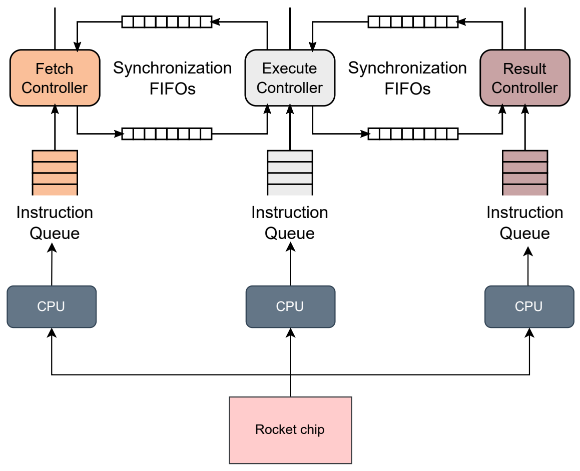

Figure 3.3: Three processors generating instructions for the BISMO accelerator. (figure extended from [5])

3.3.2 Implementation

In order to design a FSM that can generate instructions, it is essential to understand what the BISMO instructions look like, in what order the fetch, execute, and result stages process the matrices, and how the hardware uses the instructions.

Command tokens

Command tokens control the different synchronization channels and the execution of the accelerator itself. A command token uses a decoupled interface consisting of two fields: the opcode, the type of command to be run, and the synchronization channel, which tells which channel synchronization should be performed. The format of a command token can be seen in table 3.2.

Name	Bit width	Description
Valid	1	Tells when the data in the decoupled interface is valid.
Opcode	2	Opcode for the command to run.
Synchronization channel	0-1	Synchronization channel to use.

Table 3.2: BISMO command token format.

Fetch instructions

A fetch instruction consists of eight different signals, which can be seen in table 3.3. The fetch instruction tells the fetch stage which Dynamic Random-Access Memory (DRAM) addresses it is supposed to fetch the data from and to which Block Random Access Memory (BRAM) address this data should be written.

Name	Bit width	Description
Valid	1	Tells when the data in the decoupled interface is valid.
DRAM base address	64	Base address for all fetch groups.
DRAM block size bytes	32	Size of each block read from DRAM.
DRAM block offset bytes	32	Byte offset to the start of next block in DRAM.
DRAM block count	32	Number of blocks to fetch for each group.
Tiles per row	16	Number of writes before going to the next BRAM.
BRAM address base	64	Base BRAM address to start from for writes.
BRAM id start	4	ID of BRAM to start from.
BRAM id range	4	ID range of BRAM to end.
Instruction size total	249	Total amount of bits in a single instruction.

Table 3.3: BISMO fetch instruction format.

The fetch stage fetches every combination of the LHS and RHS tiles. To synchronize with the execute stage, four command tokens must also be generated for every fetch instruction. First, it needs to acquire the execute buffer in order to validate that the execute stage is done operating on the tiles it will use for the write. Then it must generate two-run instructions for the LHS- and RHS-L2 tiles it will fetch. Lastly, a command token will be generated that tells the execute stage that it has finished writing the data to BRAM. The entire flow of the instructions and command tokens for the fetch stage can be seen in algorithm 2.

Algorithm 2 Fetch stage instructions

```
for LHS L2 tile ← LHS L2 tiles per matrix do
    for RHS L2 tile ← RHS L2 tiles per matrix do
        for Common L2 tile ← Common L2 tiles per matrix do
            Get execute buffer
            Push LHS L2 tile fetch instruction
            Push run command
            Push RHS L2 tile fetch instruction
            Push run command
            Return execute buffer
```

Execute instructions

Execute instructions consist of eight different signals, as seen in table 3.4. The execute instructions tell the execute stage where the tiles it will operate on reside in BRAM, how many tiles it should operate on, where the data should be written to, and other metadata.

Name	Bit width	Description
Valid	1	Tells when the data in the decoupled interface is valid
LHS offset	32	Start offset for LHS tiles.
RHS offset	32	Start offset for RHS tiles.
Number of tiles	32	Number of L0 tiles to execute.
Shift amount	5	The number of left shifts.
Negate	1	Negate during accumulation.
Clear before first acc	1	Clear accumulators prior to first accumulation.
Write en	1	Write to result memory at the end of the current execution.
Write address	2	ID range of BRAM to end at.
Instruction size total	107	Total amount of bits in a single instruction.

Table 3.4: BISMO execute instruction format.

When having fetched an L2 tile, the execute stage performs on all the smaller tiles in the fetched LHS and RHS tiles. The hardware then goes through all the combinations of the LHS and RHS L1 tiles and creates an execute instruction for each. For every L2 tile fetched, a command token has to be generated to acquire the fetch buffer, and then when finished, another one has to be generated to return the fetch buffer to synchronize with the fetch stage. At the end of a stripe, the execute stage must first acquire the new result buffer, push the instruction and command token, then yield the result buffer. This can be seen in algorithm 3.

Algorithm 3 Execute stage instructions

```
for LHS L2 tile ← LHS L2 tiles per matrix do
    for RHS L2 tile ← RHS L2 tiles per matrix do
        for Common L2 tile ← Common L2 tiles per matrix do
            Get fetch buffer
            for LHS L1 tile ← LHS L1 tiles per matrix do
                for RHS L1 tile ← RHS L1 tiles per matrix do
                    if end of stripe then
                        Get result buffer
                    end if
                    Push execute run command
                    if end of stripe then
                        Return result buffer
                    end if
            Return fetch buffer
```

Result instructions

Result instructions consist of five different signals, as seen in table 3.5. The instructions tell the result stage where the data that have been finished in the execute stage is located in BRAM and where it should be stored in DRAM. It also supports a wait instruction, performed at the end, to track when all the writes have finished.

Name	Bit width	Description
Valid	1	Tells when the data in the decoupled interface is valid.
DRAM base	64	The start of the DRAM region the data will be written to.
DRAM skip	64	Defines the size of the steps.
Wait complete	1	Wait for completing all writes (no new DRAM writes generated).
Wait complete bytes	32	Number of bytes to wait for completion.
Result memory address	1	Result memory address to read from.
Instruction size total	163	Total amount of bits in a single instruction.

Table 3.5: BISMO result instruction format.

At the end of every stripe operated on, the result stage must write all the calculated combinations of L2 tiles to the main memory. This is done by acquiring the execute buffer, creating a result instruction, pushing an execute command, and returning the execute buffer. This can be seen in algorithm 4.

Algorithm 4 Result stage instruction generation

```
for LHS L2 tile ← LHS L2 tiles per matrix do
    for RHS L2 tile ← RHS L2 tiles per matrix do
        for Common L2 tile ← Common L2 tiles per matrix do
            for LHS L1 tile ← LHS L1 tiles per matrix do
                for RHS L1 tile ← RHS L1 tiles per matrix do
                    if end of stripe then
                        Get result buffer
                    end if
                    Push RHS L2 tile fetch instruction
                    Push execute run command
                    if end of stripe then
                        Get execute buffer
                        Push result instruction
                        Push result run command
                        Return execute buffer
                    end if
            Return fetch buffer
Push wait result instruction
Push result run command
```

3.3.3 FSM

Six separate state machines were created to generate all the instructions and command tokens logically while meeting the timing requirements in synthesis. Three generate the command tokens for the different stages, while the others generate the instructions. The decision to separate the design into so many different state machines came from the realization that instructions and commands can be generated independently since the accelerator uses queues. This makes generating them individually yield more concise and easier-to-manage code and implementation logic. Since the complexity of the different generations also varies a lot, some pipelining would, in theory, have to be done to meet timing requirements. While this implementation just clocked down the design instead of pipelining it, having a large state machine where only some generators would have to be pipelined would be challenging to manage and a time sink with little to no real gain. The FSM implementation uses an RoCC integration to communicate with the Rocket CPU.

Nested for loops

As the iteration tracking logic required for instruction generation is a for-loop, this can be expressed in two ways. This can be done as handshakes between different counters or having multiple counters dependent on each other through muxes. The second approach was chosen, which uses a sequence of computations based on conditions encapsulated within a when control flow statement. Each condition effectively serves as a checkpoint to decide whether a specific variable should be updated, thus emulating the behavior of for-loop constructs in high-level programming languages. An example can be seen in listing 3.2.

```
when(iter < total_iters){
  output := bram(i)(j)
  // Increment total iteration tracker
  iter := iter + 1.U
  // For-loop logic
  i := i + 1.U
  when(i < i_max) {
    i := 0.U
    j := j + 1.U
    when(j < j_max) {
      j := 0.U
    }
  }
}
```

Listing 3.2 Hardware for-loop construct, where iter, i, j, total_iters, i_max, and j_max is registers, bram is memory and output is the output from the Bundle.

This pattern offers a combination of the low-level control found in traditional hardware description languages and the higher-level abstraction provided by a software-like syntax. Each loop iteration is conditioned on a set of variables, representing a form of the loop counter, meeting specific criteria - to match a particular value. As the loop progresses, these variables are systematically updated, resetting to an initial value when a certain threshold is reached. This way, the code executes a complex, multi-level looping construct in a hardware-friendly manner, demonstrating the power of Chisel's high-level constructs for handling sophisticated control flow patterns.

Fetch command token generator

The fetch command token generation uses the common L2 tiles, LHS L2 tiles, and RHS L2 tiles in the matrices and multiplies them to determine the iteration space. Then, for each iteration, it generates four command tokens in order. Firstly, it generates a synchronize token to get the execute buffer. Next, it generates two-run tokens, one to fetch the LHS L2 tile and one to fetch the RHS L2 tile. Then it generates a put token to the execute buffer, signaling that the fetch is complete. The state machine for this can be seen in figure 3.4.

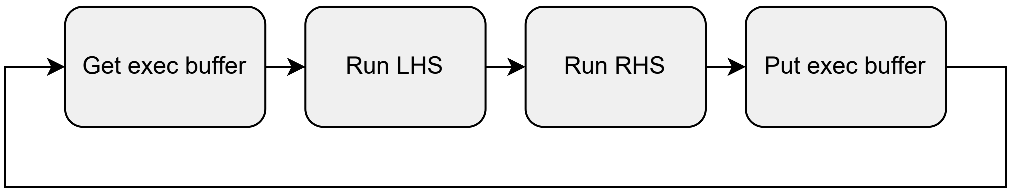

Figure 3.4: The state machine for the fetch command token generation.

Fetch instruction generator

The fetch instruction generator calculates the iteration space like fetch command generator. Each iteration generates two instructions, one for fetching an LHS tile and one for fetching an RHS tile. Which of these is currently being generated is tracked with an internal state. The fetch instruction generator outputs eight different signals, which can be seen in table 3.3.

The instruction generator takes in the start of the LHS and RHS matrices in memory, bytes per L2 tile, the amount of L0 tiles per L1 tile, and the number of bytes in the DPA common dimension and uses this in order to calculate where in memory to fetch from, the size of the blocks and the number of blocks to fetch. It also checks if `block size bytes = block offset bytes`, as the fetch request can be merged, merging multiple instructions into one large instruction. The state machine has to track the current offset of the LHS and RHS matrix tiles being read and also to which BRAM region it should be written. A basic overview of the FSM can be seen in figure 3.5.

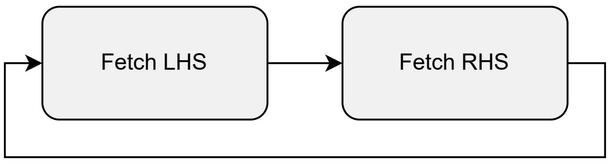

Figure 3.5: The state machine for the fetch instruction generation.

Execute command token generator

From a design point of view, the execute command token generator is the most complex. This is because it has to synchronize with the fetch stage after each tile has been executed, creating nested loops with multiple different states for each iteration which is difficult to express in Chisel. The state machine can be seen in figure 3.6. It calculates the iteration space using common L2 tiles, LHS L2 tiles, RHS L2 tiles, LHS L1 tiles, and RHS L1 tiles that it receives from the instruction. First, the generator must check if a new stripe has been fetched and, if so, acquire the input matrix buffers. Then it needs to check if it is at the end of a stripe. If so, it must acquire the result buffer, run the execute command, and then give away the result and fetch the buffer. It generates an execute command token if it is not at the end of a stripe.

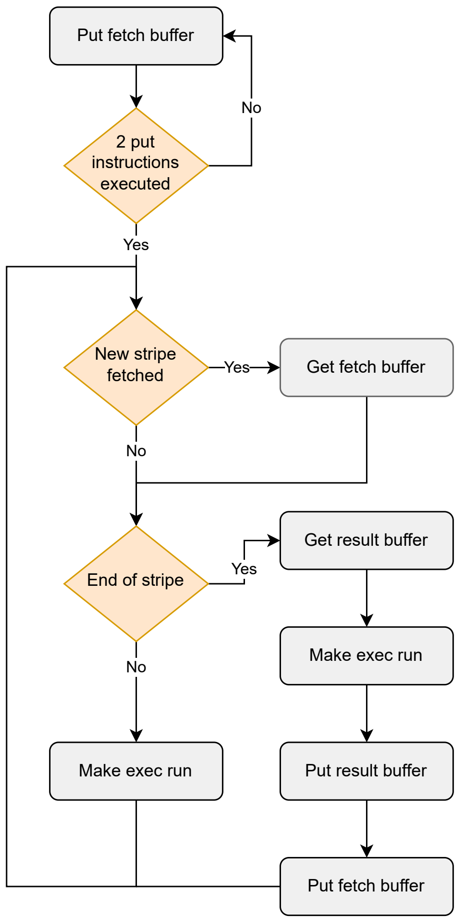

Figure 3.6: The state machine for the execute command token generation.

Execute instruction generator

The execute instruction generator calculates the iteration space similarly to the execute command generator. The generator generates a single instruction for every iteration, as seen in figure 3.7. The execute instruction generator outputs eight different signals, which can be seen in table 3.4.

The execute instruction generator needs to calculate in which BRAM region the fetch stage placed the data and to which region it should write the data for the result stage. To fetch the correct tiles for execution, it needs to calculate the LHS and RHS offset in BRAM, which it does with iteration tracking logic.

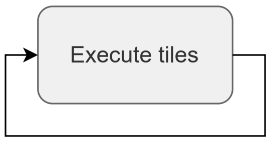

Figure 3.7: The state machine for the execute instruction generation.

Result command token generator

The result command token generators iteration space is calculated by multiplying LHS L2 tiles, RHS L2 tiles, LHS L1 per L2 tile, and RHS L1 tiles per L2 tile. First, two put command tokens are issued to fill the synchronization buffers. After this, the command tokens are generated as follows. It generates a command token to get the execution stage buffer, ensuring it has finished. Then it creates a run command token, which makes the result stage write to memory. When the write has finished, the execute buffer returns to the execution stage, implying it is done. If it has finished all the iterations, it will generate a run command token for the wait instruction, which waits until all the writes to memory have finished. The state machine can be seen in figure 3.8.

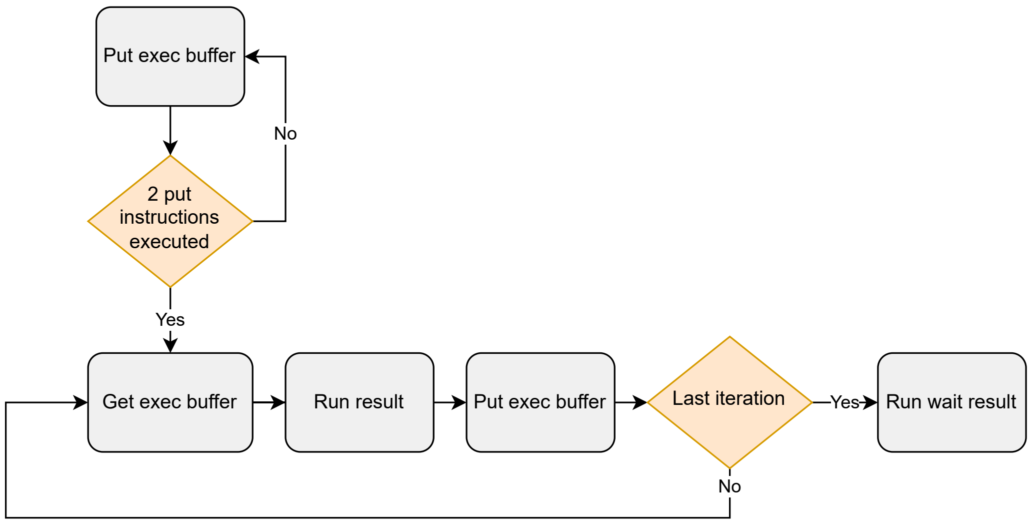

Figure 3.8: The state machine for the result command token generation.

Result Instructions

The result instruction generator calculates the iteration space similarly to the result command generator. For every iteration, the generator only generates a single instruction, as can be seen in figure 3.9

The result instruction generator uses iteration tracking logic to determine which BRAM region the execute stages for the L2 tile were written. It then calculates the DRAM address where the result L2 tile should be stored by tracking the current LHS and RHS tiles it is working on and using this to calculate the offset from the start of the result matrix.

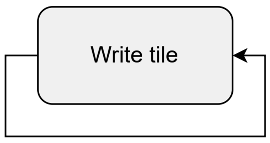

Figure 3.9: The state machine for the result command generation.

New Instruction

The presented state machine uses only a single instruction with a decoupled interface. It does not have a specific signal to reset the generators individually. However, it relies on the top-level reset to reset the accelerator after each use. The new instructions can be seen in table 3.6.

3.4 Connecting to memory

Since BISMO is operating on large matrices stored in memory, it needs a way to issue reads and writes to the main memory. This means that it needs to be connected to the front bus. Since BISMO uses an AXI interface and the rocket chip uses TL, two options were considered, namely, rewriting the BISMO interface to TL or using a converter. Since the AXI to TL converter seemingly had little overhead, rewriting the BISMO interface would make little sense concerning this thesis because it would most likely not influence the results in any way.

The first step to connect the BISMO accelerator to the front bus is creating a TL identity node. The TL identity node has inputs and outputs and connects the inputs and outputs [49]. This makes it an intermediate layer between the AXI interface and the front bus. This node is then connected to the front bus. The way this is done can be seen in listing 3.3.

```
fbus.coupleFrom("bismo_dbb") { bus =>
      (bus := TLBuffer(
        BufferParams(p(BISMOFrontBusExtraBuffers))
      ) := tl_identity_node)
```

Listing 3.3 Connecting a TL node to the front bus.

Seven intermediate tools must be used to convert the AXI node to a TL node. This can be seen in listing 3.4.

Name	Bit width	Description
Valid	1	Tells when the data in the decoupled interface is valid.
DRAM base LHS	64	The base address of the LHS matrix.
DRAM base RHS	64	The base address of the RHS matrix.
DRAM block offset bytes	32	Byte offset to the start of next block in DRAM.
DRAM block size bytes	32	Size of each block read from DRAM.
DRAM block count	32	Number of blocks to fetch for each group.
Tiles per row	16	Number of writes before going to the next BRAM.
NumTiles	32	Number of L0 tiles to execute.
Shift amount	5	The number of left shifts.
Negate	1	Negate during accumulation.
Dram base	64	Start of the DRAM region, the result data will be written.
Dram skip	64	Defines the size of the steps.
Wait complete bytes	64	The number of bytes the result stage writes to memory.
LHS L0 per L1	64	Amount of L0 tiles per L1 tile in the LHS.
RHS L0 per L1	64	Amount of L0 tiles per L1 tile in the RHS.
LHS L1 per L2	32	Amount of L1 tiles per L2 tile in the LHS.
RHS L1 per L2	32	Amount of L1 tiles per L2 tile in the RHS.
LHS L2 per matrix	32	Amount of L2 tiles in the LHS matrix.
RHS L2 per matrix	32	Amount of L2 tiles in the RHS matrix.
ZL2 per matrix	32	Amount of common tiles in the RHS and LHS matrix.
LHS bytes per L2	64	Bytes per L2 tile in the LHS matrix.
RHS bytes per L2	64	Bytes per L2 tile in the RHS matrix.
Nrows a	32	The number of rows in the LHS matrix.
DPA z bytes	64	Amount of bytes that fit in the common DPA dimensions.
Instruction size total	983	Total amount of bits in a single instruction.

Table 3.6: BISMO FSM instruction.

```
(tl_identity_node
:= TLBuffer()
:= TLWidthWidget(dataBits / 8)
:= AXI4ToTL(numTlTxns = numTlTxns)
:= AXI4UserYanker(capMaxFlight = Some(numTlTxns))
:= AXI4Fragmenter()
:= AXI4IdIndexer(idBits = idBits)
:= AXI4Buffer()
:= axi_node)
```

Listing 3.4 AXI4 to TL conversion with the tools used.

An issue that arose during implementation was poor converter performance. This resulted from the lack of tracking logic for out-of-order responses, which meant that the TLFIFOFixer widget had to be used. The TLFIFOFixer ensures that responses return in FIFO order, meaning that only one outstanding request can be issued at a time. This removes memory parallelism. This issue had been discussed in the chipyard Google groups [50] for the NVDLA.

The author modifies an existing design, incorporating the concept of a new type of memory system, referred to as ReorderRAM, that stores transaction metadata. This system has a capacity determined by the maximum concurrent transactions and the total number of unique identifiers. A corresponding system of single-bit registers is also maintained to validate each ReorderRAM entry. Furthermore, two counters are kept to manage read requests and responses.

The process of assigning source fields in the new design remains the same as in the original model. Upon receiving a read request from an identifier, its corresponding counter increments. The identifier of the request then gets updated to the current counter value. As a read response comes in, its identifier is compared to the response counter of its corresponding identifier. If they align, the response is processed normally. However, if the response is out of sequence, it is stored in the ReorderRAM. Future in-sequence responses trigger checks in the ReorderRAM for the subsequent response. Data is pulled from the ReorderRAM and sent until an invalid entry is encountered; at this point, regular response handling is resumed.

A downside of this implementation is that it is separate from the official Chipyard release. Some time into this thesis, it also became clear that the implementation does not comply with the standard AXI protocol. It could cause problems if multiple devices work on the same data [51]. However, this does not affect the BISMO implementation for this thesis, making it a good band-aid fix.

Chapter 4

Methodology

This chapter presents the methodology used to benchmark and collect results for the three implementations. Section 4.1 will present the configuration used for the simulations. Section 4.2 will talk about how verification of the design was handled. Section 4.3 will present the target hardware used for the simulation. Section 4.4 will present how the different implementations were evaluated and what the different metrics mean. Section 4.5 will discuss how memory was handled with the RISC-V Proxy Kernel (PK) and some modifications that had to be made. The proxy kernel fork used in this thesis can be found on Github[1].

4.1 Configuration

BISMO comes with many reconfigurability options, most notably the LHS, RHS, and common dimensions. These were chosen based on what performed best in the original BISMO paper [5]. The command queue entries are set to 16 or 256. Using two vastly different queue sizes can expose different bottlenecks. It can show that the interconnect system is too slow or that the accelerator is consuming the queue faster than the CPU deduces that the queue is being emptied and pushes more instructions. This would mean that communication delay is the issue. The configuration can be seen in table 4.1.

[1]`https://github.com/Vetleh/riscv-pk`

Name	Description	value
DPA LHS dimensions	The unique dimension of the LHS matrix.	8
DPA RHS dimensions	The unique dimension of the RHS matrix.	8
DPA common dimensions	The common dimension between the LHS and RHS matrix.	256
LHS entries per memory	Number of L0 tiles that can be stored on-chip for LHS matrices.	1024
RHS entries per memory	Number of L0 tiles that can be stored on-chip for RHS matrices.	1024
BRAM pipeline before	Levels of registers on address input of each tile memory BRAM.	1
BRAM pipeline after	Levels of registers on each tile memory BRAM data output.	1
Accumulator width	Accumulator width in bits.	32
Max shift steps	Maximum number of shift steps.	16
Command queue entries	Size of the instruction queues.	16/256
Shifter	Defines if the shifter should be instantiated.	false

Table 4.1: BISMO parameters used for simulations.

The memory request parameters were chosen based on what the TL interface in Chipyard supports. In this case, the essential parameters are the address and data width. The configuration can be seen in table 4.2.

Name	Description	value
Address width	Width of memory addresses.	48
Data width	Width of reads/writes.	64
Channel ID width	Width of channel ID.	32
Metadata width	Width of metadata (cache, protection, etc.).	1
Same ID In Order	Whether requests with the same ID return in-order, like in AXI.	true

Table 4.2: Memory request parameters used for simulations.

The Chipyard configuration used for the tests utilized a single large-core Rocket core. Configuration instantiation can be seen in listing 4.1.

```
class BISMORocketConfig
    extends Config(
      new bismo.WithBISMOAccel() ++
        new WithNBigCores(1) ++
        new Chipyard.config.AbstractConfig
    )
```

Listing 4.1 Chipyard configuration with BISMO used in simulations.

4.2 Verification

While simulating an FPGA is faster, it also provides fewer debugging options, and the required bitstream can take time to generate. Verilator simulations were used to solve this during the development process to validate and test the design. This becomes helpful when developing complex FPGA systems, as it enables easier identification of logical and timing problems at the software level before moving onto hardware. Verilator enables waveforms and print debugging with a short build time. This makes it efficient when applying and debugging small changes to the design without the overhead of building a bitstream. Verilator simulation on a general-purpose computer is much slower than on FPGAs, making it inferior when testing software changes.

4.3 Target Hardware

Firesim will target the Xilinx u250 data center accelerator card for the simulations. The u250 is an FPGA that can simulate custom Chipyard RTL designs at speeds up to 300MHz. The designs of this thesis will simulate at 50MHz. This is a significant speed-up from simulations on general-purpose computers, which run in kilohertz. These FPGAs are situated on the IDUN/EPIC cluster [52], which is a research cluster of tier 2 [53] at NTNU. The specifications of the Xilinx u250 can be seen in table 4.3. The bitstream for the FPGA is built using Vivado v.2022.2.

Type	Value
Peak INT8 TOPs	33.3
DDR Memory Bandwidth	77GB/s
Internal SRAM Bandwidth	38TB/s
Lookup Table (LUT)s	1,341,000

Table 4.3: The u250 FPGA specifications.

4.4 Evaluation

This section presents the evaluation methods for all the data collected and the tests used. Cycle counts of the different states, performances, and sizes will be used to evaluate the different designs.

4.4.1 Performance data

Relevant data must be collected to assess the strengths and weaknesses of different implementations.

Command state

Command state describes how the stages spend the cycles that the accelerator is running a workload. There are four different states:

1. Get: the number of cycles the stage waits for an instruction from the processor.

2. Run: the number of cycles the stage is running.
3. Send: the number of cycles the stage spends sending synchronization tokens.
4. Receive: the number of cycles the stage spends on receiving synchronization tokens.

Each stage will always exist in one of these states, and counters will be used to keep track of them. A stage machine keeps track of the current state. It uses the opcode of the instructions to track which operation is being performed. The state machine's default state waits for a command before using the command opcode to deduce the next state. Then it will be in that state until the operation is done, at which point it will return to the get state.

Performance

Performance is measured in Operations per second (OPs). The OPs per cycle for the entire run will be benchmarked. This is calculated as follows:

$$2 * LHSDim * RHSDim * CommonDim/cycletime \tag{4.1}$$

Where LHSDim is the dimensions of the left-hand side matrix, RHSDim is the dimension for the right-hand side matrix, CommonDim is the dimension that the matrices have in common, and cycle time is given by `1000 / Clock frequency (MHZ)`. The clock frequency is calculated using an internal counter that increments every cycle and a one-second delay to read the counter after one second has passed. Since the BISMO accelerator is clocked at 3.2GHz, which is an unrealistic speed, the OPs will be incomparable to other work; hence these should only be reviewed concerning the similarly clocked implementations in this thesis. While just dividing the OPs by `(3.2GHz / expected BISMO frequency)` would yield a more realistic result, it would also be inaccurate due to the Rocket Chip and interconnect system also being clocked down to a slower speed than usual.

4.4.2 Memory delay

The difference in communication delay for the RoCC implementation and TL implementation will be calculated by first sending a request that enables BISMOs internal cycle counter, reading the cycle counter, making a read or write request, and then rereading the cycle count. Then the cycles read will be the time it took for the request to complete, which should be a good reference. This ignores factors like contention on the peripheral bus, but benchmarking this would only be feasible and logical if the BISMO accelerator is implemented with other accelerators in a configuration that makes sense for an SoC and would be way out of the scope of this thesis.

4.4.3 Size usage

In order to estimate the size usage of the different implementations, the bitstreams will be analyzed. First, a baseline will be calculated using the Rocket Chip without the BISMO accelerator. Then each implementation size will be calculated. The size calculation is done with the output from the bitstream generation, which Vivado outputs. The critical size for this thesis is the Configurable Logic Block (CLB) usage, which typically consists of essential elements such as Lookup Table (LUT)s, flip-flops, and multiplexers.

A LUTs forms the fundamental building block of an FPGA, capable of implementing any N Boolean variables logic function, essentially acting as a truth table for varying input combinations. In hardware terms, a LUT can be conceptualized as a group of memory cells connected to multiplexers, with the LUT inputs serving as selector bits on the multiplexer, deciding the outcome at any given moment. This dual functionality enables a LUT to be utilized as a function computation engine and a data storage element [54].

The size usages that will be used for size estimation are the following:

- CLB LUTs* represents the total number of Look-Up Tables used in the FPGA.
- LUT as Logic indicates the LUTs used for implementing combinational logic.
- LUT as memory shows the LUTs used as memory, which could be used to implement small memories, FIFOs, or shift registers. ...
  - LUT as Distributed RAM is a subset of LUT as Memory, specifically showing LUTs used as distributed RAM.
  - LUT as Shift Register is another subset of LUT as Memory, specifically showing LUTs used as shift registers.
- CLB Registers defines the total amount of CLB registers used.
  - Register as Flip Flop is a subset of CLB registers and defines the number of flip flop registers.
  - Register as Latch is a subset of CLB registers and defines the number of latch registers used.
- Amount of CARRY8 components used in the implementations.
- Amount of F7, F8 and F9 Muxes used in the implementations.

4.4.4 Benchmarks

The benchmarks are performed with matrices of different dimensions. The test takes two randomly generated matrices of a given size, generates the instructions for the accelerator, and checks that the result is correct. The different configurations used for the tests can be seen in table 4.4. The tests first generate all the required instructions. Then it pushes all required instructions into the queues if they have enough space to fit them. Once all the instructions are pushed, it waits for the result command queue to empty before turning off the accelerator. Cycle counts in the different states, and the accelerator in general will be measured in two different manners. Afterward, the accelerator is reset with a top-level reset signal before the next test is run. The first test measures the number of cycles to push the instruction. This will start the cycle counter after the instructions are generated. The second test measures the time it takes to generate and push the instructions. This will start the cycle counter after the instructions are pushed. The number of instructions that each of these configurations requires in the test can be seen in figure 4.1.

LHS dim	RHS dim	Common dim
8	8	131072
8	8	65536
8	8	32768
16	16	65536
16	32	32768
32	16	32768
32	32	16384
16	16	262144
16	32	262144
32	16	262144
32	32	262144
8	8	524288
8	8	1048576
8	16	524288
8	16	1048576
8	32	524288
8	32	1048576
16	8	524288
16	8	1048576
16	16	524288
16	16	1048576
16	32	524288
16	32	1048576
32	8	524288
32	8	1048576
32	16	524288
32	16	1048576
32	32	524288
32	32	1048576

Table 4.4: BISMO matrix dimensions used for the tests.

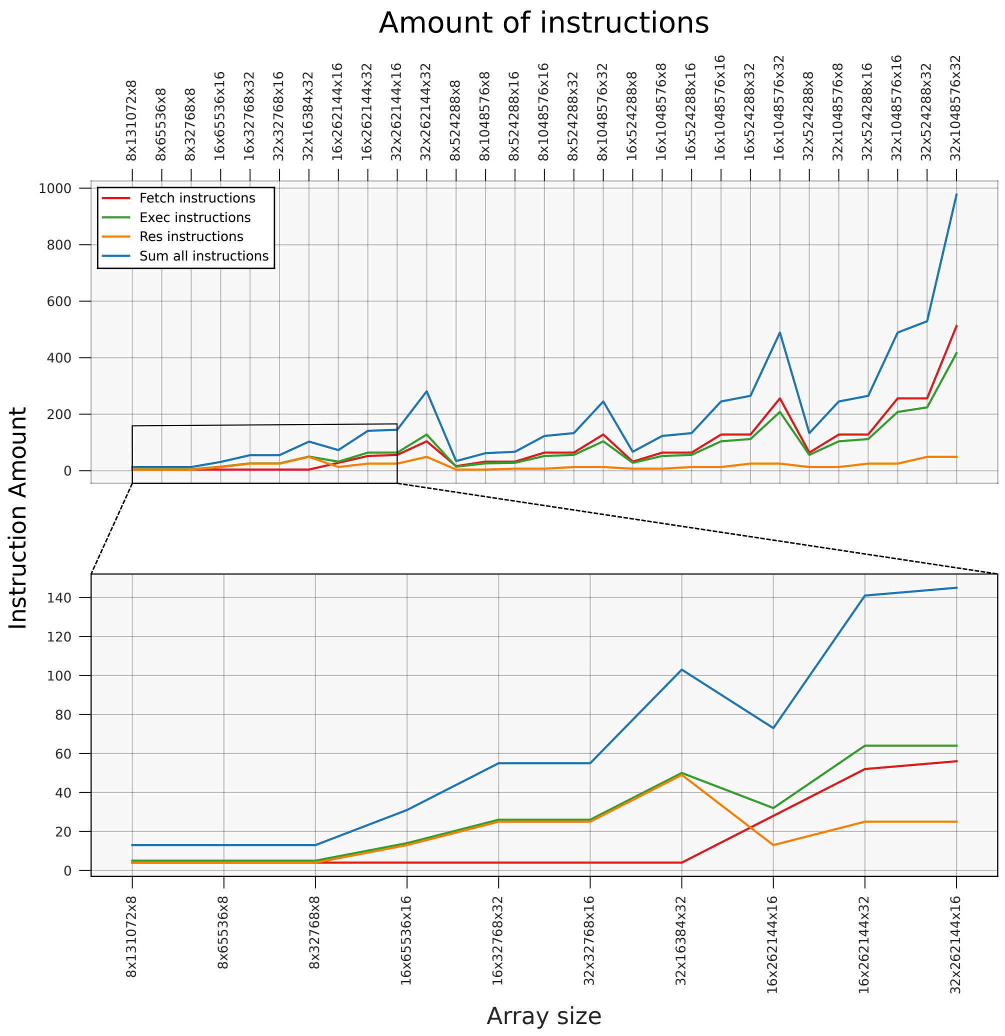

Figure 4.1: Amount of instructions generated and communicated for each array dimension.

4.5 Proxy kernel

The RISC-V PK [55] is a minimalistic platform for executing applications capable of accommodating RISC-V ELF binaries that are statically linked. The PK enables applications not designed for baremetal mode to run without an operating system. It supports RISC-V implementations with limited I/O by proxying I/O-related calls to the host computer. For Verilator simulations, the Linux system it runs on serves as the host computer, and for simulation on the u250 FPGA, the nodes use an Intel Xeon Gold 6226 and serves as the host system. The path of the binary application must be supplied as an argument to initiate the PK. After that, it initializes the system, which involves setting up exception handlers and virtual memory. Then, it reaches out to the host to retrieve the binary, loads it into memory, and finally starts the application with the given arguments. Using the proxy kernel for simulation was chosen over bare-metal and Linux for several reasons.

Since the existing tests and drivers for the accelerator are written in C++, which uses memory allocations and other memory features, a total rewrite of the existing tests would be required, and a custom memory allocator to handle memory would have to be added. This would be time-consuming while presenting verification issues, as verifying that the hardware implementation is correct can only be done with correct software tests and vice versa. Machine learning is also the primary use case of BISMO since it is geared towards matrix operations with variable precision. This makes having support for C++ code and a proper memory management implementation desirable, as it can enable benchmarking actual QNN workloads in the future.

A full-fledged Linux implementation would offer great flexibility, a realistic environment, and many other features. The problem with Linux is that MMIO accelerators in Chipyard do not have access to the Page Table Walker (PTW). Since BISMO's fetch- and control stages use physical addresses to access memory, Linux would require modifications to the kernel to guarantee a static physical address and the ability to get the physical address from a virtual address. There would also have to be written a driver for the accelerator, making this a tremendous task while contributing little to the questions this thesis wishes to answer.

The PK uses virtual memory to map to physical memory, requiring virtual-to-physical mapping. Usually, memory regions can be located and mapped to virtual memory by accessing `/dev/mem` on the device's file system. One cannot access the simulation file system when using the PK with Chipyard, as the files and syscalls are routed through the host device. There is no native way in the PK to map a specific physical memory region to a virtual region, which made communicating with the MMIO register mappings impossible. Solving this required a custom system call that maps a continuous physical region to a virtual region. The system call returned the virtual address of the mapping. It enables communication by combining the virtual address with the offset of the registers. This was done by performing a page table walk to get the page table entry of the virtual address. The entry of the page table in an Sv48 virtual space like Chipyard consists of 64 bits, as seen in table 4.5. The reserved field can then be set to 0 by anding a bitmask before right-shifting the page table entry with 10 to be left with the physical page table entry. The result can then be left-shifted to get the physical page on the format seen in table 4.6 before adding the offset to get the physical address in memory.

64 54	53 37	36 28	27 19	18 10	9 8	1	1	1	1	1	1	1	1
Reserved	PPN[3]	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	17	9	9	9	2	1	1	1	1	1	1	1	1

Table 4.5: RISC-V page table entry [6].

55 39	38 30	29 21	20 12	11 0
PPN[3]	PPN[2]	PPN[1]	PPN[0]	offset
10	17	9	9	9

Table 4.6: RISC-V physical page [6].

Since the BISMO accelerator cannot access the Translation lookaside buffer (TLB), it fetches and writes directly to physical memory through the front bus. This bare-metal property implies that the accelerator needs the physical addresses of the matrices and where the result should be stored. These addresses are sent from the CPU generating the instructions, which operate on virtual memory. To do this with the PK, a custom system call was implemented that walks the virtual page table with a virtual address, checks if the physical memory is contiguous, and returns the start of the physical memory region. An issue that arose with this implementation was that securing continuous physical memory, which the accelerator would need, is difficult, if not impossible, without rewriting the memory management system of the proxy kernel or just removing the ability for the proxy kernel to deallocate memory. The first option would, in all cases, be the most optimal since one can reserve a specific memory region just for BISMO. Since BISMO only needs to allocate three matrices with continuous memory, this could be handled by reserving a region and dividing it into three equal parts, then using one part for each matrix. However, the option of turning off memory deallocation was chosen due to the limited time. Suppose that the proxy kernel is unable to deallocate memory. In that case, there will be no available pages from the start of the memory to the end of the last allocated page; hence, all the allocated memory will be continuous. This comes with the drawback of the memory eventually being filled up.

Chapter 5

Results

This section presents the results of the synthesis and simulations. Why the results look like they do will also be briefly discussed, creating some context. Section 5.1 presents the size usage of the different implementations and compares it to the baseline. Section 5.2 presents and compares the delay from the CPU to the accelerator using MMIO and RoCC. Section 5.3 presents the overhead of generating instructions for the FSM and the software approach. Section 5.4 and section 5.5 present the cycle counts of the command states using a queue size of 256 and 16, respectively.

5.1 Size usage

The size usage of the three implementations and the baseline, the Rocket Chip without the BISMO accelerator, can be seen in table 5.1. The different implementations evaluated are the default implementation, which is the Rocket Chip without the BISMO accelerator, the loosely coupled MMIO implementation, the tightly coupled RoCC implementation, and the FSM implemented with a loosely coupled RoCC interface. Table 5.1 show that the MMIO and RoCC software generated implementations use about the same amount of LUTs, with only a $\approx 1.2\%$ difference. Subtracting the baseline from the MMIO, RoCC, and RoCC FSM defines the use of the implementation of the BISMO accelerator itself without the rest of the Rocket Chip, which can be seen in table 5.2.

The overhead of the FSM can thus be calculated by subtracting the RoCC implementation from it, which results in $62599 - 55710 = 6889$. This yields a silicone overhead of $\approx 12.3\%$. The original BISMO paper [5] has a definition of LUT usage, which can be seen in equation (5.1) and equation (5.2).

$$LUT_{\text{total}} = LUT_{\text{base}} + LUT_{\text{array}} \tag{5.1}$$

$$LUT_{\text{array}} = D_{\text{m}} \cdot D_{\text{n}} \cdot (LUT_{\text{DPU}} \cdot LUT_{\text{res}}) \tag{5.2}$$

Where D_m and D_n are the dimensions of the DPA, since the FSM implementation does not depend on any of these factors, a larger DPA would lessen the relative overhead of the implementation.

Site Type	Default	MMIO	RoCC	RoCC FSM
CLB LUTs	52753	107195	108463	115352
LUT as Logic	48634	99317	100257	107226
LUT as Memory	4119	7878	8206	8126
LUT as Distributed RAM	4112	7860	8188	8108
LUT as Shift Register	7	18	18	18
CLB Registers	29945	91621	93998	93772
Register as Flip Flop	29945	91621	93998	93772
Register as Latch	0	0	0	0
CARRY8	817	1899	1902	2896
F7 Muxes	1982	6394	7051	7063
F8 Muxes	202	2379	2582	2560
F9 Muxes	0	0	0	0

Table 5.1: How LUT components is used in the different implementations

Implementation	CLB LUT usage
MMIO	54442
RoCC	55710
RoCC FSM	62599

Table 5.2: Total LUT overhead of the BISMO accelerator in the different implementations.

The opposite would also be true for a slimmer DPA. The actual size of the DPA would depend on the workloads for which the accelerator will be used. However, the configuration used in this thesis yielded the best results in the original paper [5], and it would be fair to assume this is the most optimal. The configuration size was, however, scaled further in the optimized version of BISMO [2], which could point to limited hardware resources being the reason it was not made larger in the original paper. This could indicate a larger size yielding better performance, meaning the relative overhead of 12.5% would be reduced.

5.2 MMIO delay

Since the CPU for the MMIO and the RoCC accelerators generate instructions at the same rate, the number of cycles a memory load and store to the accelerator takes will be the only potential difference between them. Testing the time it takes to perform a load and a store reflects the difference in communication delay, shown in table 5.3. The difference in the communication delay between MMIO and RoCC comes from the coupling and hence the distance. The tighter coupling of the RoCC means the accelerator is integrated into the processor pipeline and hence a lot closer in proximity to the core. Since delay scales approximately linearly with length [56], this leads to a significantly longer delay for the looser coupling.

	MMIO	**RoCC**
Load	39 cycles	14 cycles
Store	39 cycles	11 cycles

Table 5.3: Delays of loads and store for TL MMIO and RoCC.

Since FireSim simulates the design at 3.2GHz, the delays of the different implementations can be calculated by taking `cycles / cycles per nanosecond`. The results can be seen in table 5.4.

	MMIO	**RoCC**	**RoCC speedup**
Load	12.12 ns	4.38 ns	3.59x
Store	12.12 ns	3.44 ns	3.52x

Table 5.4: Delays of loads and store in RoCC and TL MMIO in nanoseconds.

This delay indicates that the difference in delay between an MMIO and RoCC accelerator is relatively large and especially matters if the workload of the accelerator is small, which would make the delay an even more significant factor. However, it may not always be detrimental for larger workloads since memory parallelism enables the processor to issue multiple requests simultaneously, and the delay in clock cycles is relatively low compared to the workload. The delay would mostly be detrimental for the MMIO integration in cases where the read and write responses are highly dependent on each other and happen often.

5.3 Instruction generation overhead

This section will present the overhead of instruction generation with a software-based versus a hardware approach by comparing the RoCC FSM hardware generation with the RoCC software generation. Here, the cycle count is started before the instructions are generated. This makes the cycle count reflect the time it takes to generate, push, and execute all the instructions.

In figure 5.1, the total operations per second can be seen. Note that the y-axis scale factor is 10^7. The data show a massive startup overhead to generate the software instructions. Interestingly, the two graphs follow each other with an offset as the matrices size increases. Since the number of instructions and the size of the instructions are the same for the FSM approach, this could mean that the extra cycle count comes from system calls to the host from the PK when working with virtual memory and calculating the instruction parameters, which starts to become a more significant issue as the array sizes increase. This is likely the cause of much of the initial overhead, as generating the instructions is relatively simple but requires memory allocation. If this were circumvented, one would still expect the difference to be significant, as the instruction generated for the FSM is not much larger than one instruction for each of the stages combined, and the accelerator usually requires way more than one instruction per stage. This is a downside of using the PK as memory operations and system calls become hard to evaluate, and in this case, it is impossible to deduce the actual overhead of instruction generation.

Figure 5.2 shows the speedup as a multiplier for the RoCC FSM. This shows that the speed-up starts extremely high, then starts to trend downward before it flattens out between 10x and 3x. The results are extremely volatile, showing a 300x to 3x speed-up range. It is also important to note that the FSM required the implementation to be clocked down to 10 MHz, which means that it, in practice, would be slower than software generation. Although the results are not entirely reliable, they speak for hardware generation since the number of cycles consumed in the CPU is much greater in software generation. However, this depends on how crucial the performance of other processes running on the CPU, if any. If this is unimportant, one could argue that one would still be better served with a lower performance CPU and hardware generation.

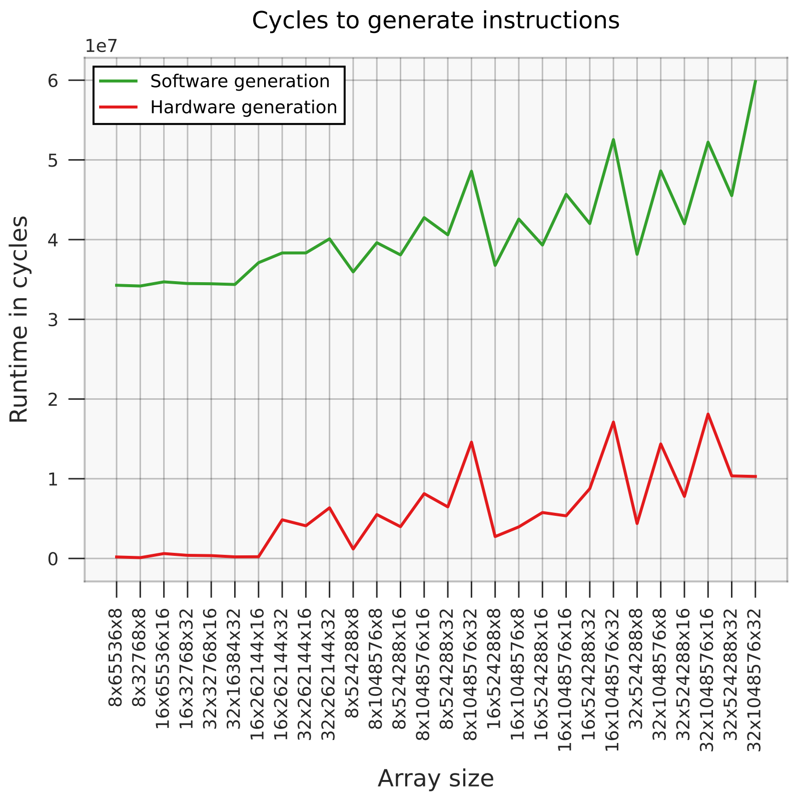

Figure 5.1: The overhead of generating instructions.

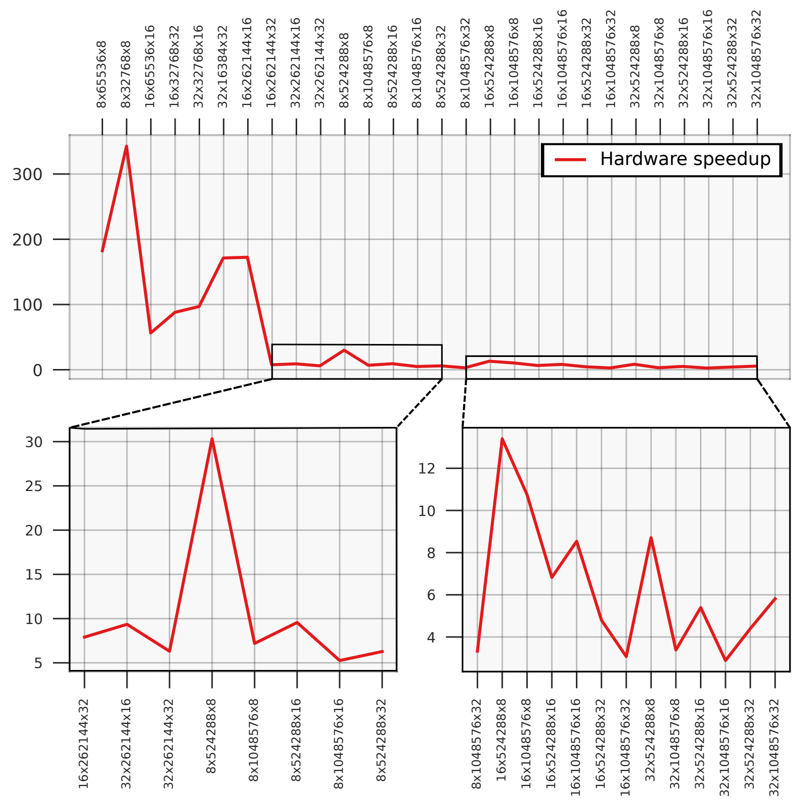

Figure 5.2: Speed-up from hardware generation using an FSM versus software generation.

5.4 Cycles spent in states – 256 queue

This section presents the results from the three implementations with an instruction queue size of 256. A queue of size 256 was used in order to assess the performance when the queue is large enough to have a buffer larger than the number of instructions consumed for execution on a single tile.

The results of figure 5.3 show that the FSM implementation waits significantly longer for instructions than the MMIO and RoCC implementations, which wait about equally long. However, the time spent in the csRun state of all three implementations is equal, which is to be expected since they are connected to the main memory similarly. The first thing to notice in figure 5.3 is the relationship between the run and the wait for command states. When the matrices get more extensive, a significant portion of the time is spent in the run state, which means that, for larger workloads, the time it takes to receive the instructions dwarves compared to the runtime, especially the runtime of the fetch and execute stage. For the result stage, this does not hold since it can send out the requests without having to be idle while waiting for it to finish before it reaches the last request.

Another aspect is that most of the runtime is spent in the fetch stage, which spends ≈ 10 times as much time in the run state as in the execution stage. The result stage barely takes up any time, making fetching the main bottleneck of the implementation.

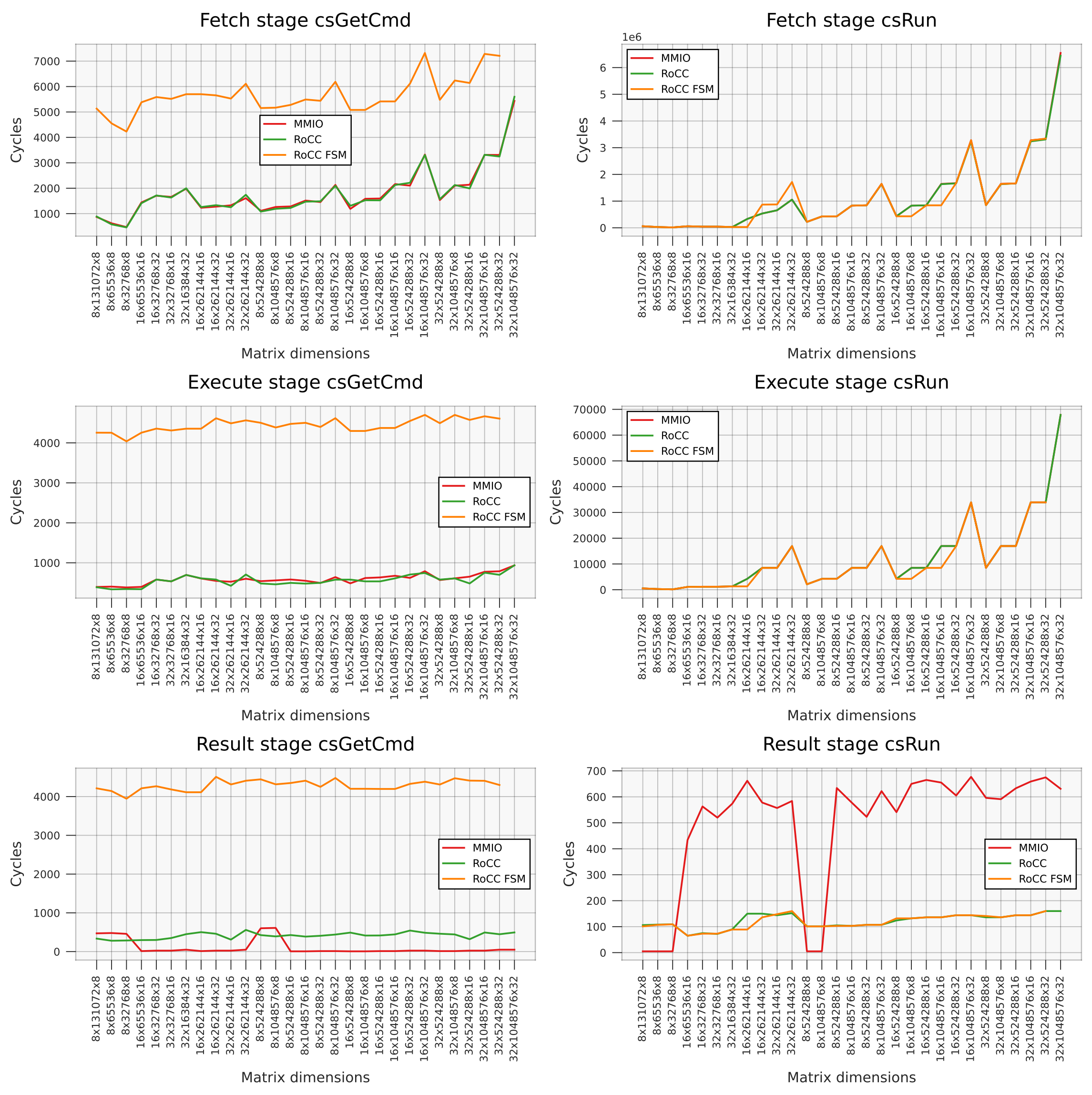

Figure 5.3: The number of cycles spent in the wait and run stage for the different implementations in the different stages with a queue of size 256.

The MMIO and RoCC software generation have very similar performance metrics, which makes sense considering that they have about the same delay to memory and communication with CPU. An area where the benchmarks differ is in the result stage, where the wait for command and run states differ significantly from the RoCC software generation implementation. The results are also very sporadic at this stage. This is most likely due to some issues with the state machine that tracks the current state of the stage. This is a logical conclusion to draw since the sum of the cycles in the csGetCmd state and the cycles in the csRun stage are equal when comparing the implementation of RoCC software generation with the implementation of MMIO software generation.

The last interesting finding is that the implementation of RoCC FSM integration has much more cycles in the csGetCmd state. This can be attributed to the larger instruction and the fact that the state machine does not start before the instruction is sent due to its decoupled interface. In the case of the two other approaches, the small fetch instruction will be sent first, enabling it to start fetching data quickly, which is the operation that takes the longest.

5.5 Cycles spent in states – 16 queue

This section presents the results of using a small queue of size 16 for the tests. A queue size of 16 is used to assess if the communication delay can be an issue since a small queue can potentially empty before the processor can deduce that it is being drained and push more instructions.

The results of figure 5.4 show that the implementations perform similarly to those with a larger queue size, except for a specific interval. In this interval, the LHS and RHS dimensions are large compared to the common dimensions in the matrices, which are relatively small. This means that the fetch stage performs smaller fetches for each fetch instruction, while the number of instructions is high due to the relatively large LHS and RHS dimensions. The fetch stage is the main bottleneck of the implementation; therefore, when it finishes quickly, it increases the number of instructions consumed per cycle in all stages, making instruction communication the bottleneck. On the other hand, the FSM can handle this since it can generate an instruction every other cycle for the accelerator. The result stage performs better in this interval but it is insignificant since the improvement is about 40 cycles. However, this is unexpected since they perform the same workloads and use the same memory implementation. The RoCC and MMIO implementations perform very similarly, and both struggle to keep up in the same interval. From figure 5.4, one can see that the MMIO implementation struggles more, most likely due to a longer communication delay.

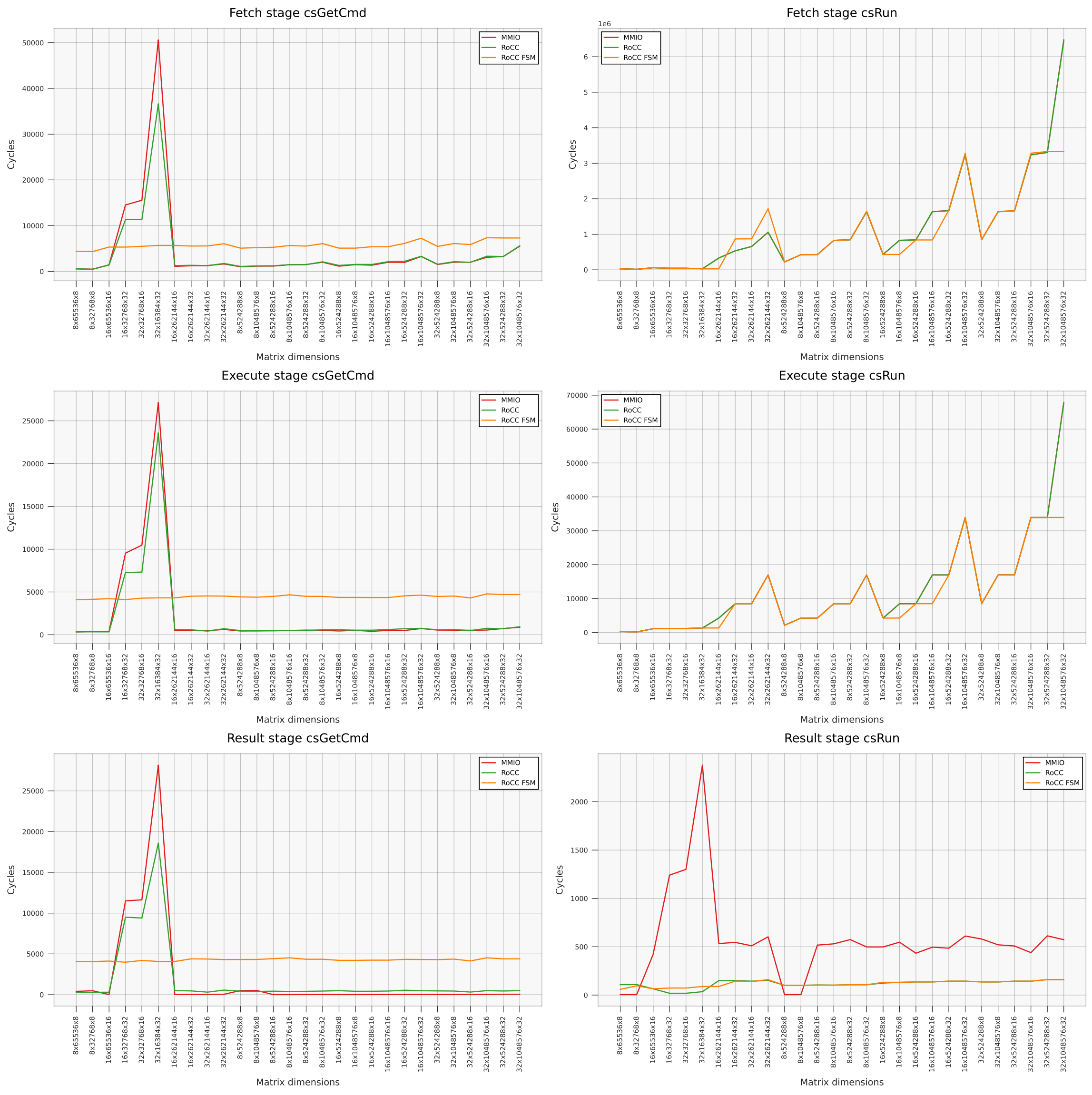

Figure 5.4: The number of cycles spent in the wait and run stage for the different implementations in the different stages with a queue of size 16.

Chapter 6

Discussion

This chapter aims to place the results in a broader context, looking at how the methodology affected the results, what the results mean, and how this can be applied. Section 6.1 discusses the success of the implementation of RoCC software generation compared to the baseline approach of a TL MMIO accelerator. Section 6.2 looks at the hardware approach and discusses the strengths and weaknesses of this compared to the original software approach. Section 6.3 will discuss which configuration seems the most reasonable. Section 6.4 then discusses the integration with the Chipyard ecosystem and looks at how portable and easy the current implementation is to integrate with other Chipyard SoC designs. Lastly, section 6.5 will discuss the weaknesses of the methodology and implementations used in this thesis and how this affects the results.

6.1 RoCC

As mentioned in chapter 5, the performance results of the RoCC integration were very similar if not identical to the baseline TL MMIO integration with a larger queue size, both in performance and in size usage. This shows that the accelerator can push the instructions needed as long as the queues are large enough to have a buffer. On a smaller queue, it has problems with some sizes that can be seen in figure 5.4. When the common dimension becomes small, and the RHS and LHS dimensions become large, the L2 tiles that are fetched become large. As seen in figure 4.1, this results in fewer fetch instructions than others since they can fetch large tiles. Large tiles mean that the execute stage will have a much bigger workload to execute each time it receives the data, which makes it consume instructions rapidly. Due to the delay from the accelerator to the CPU, the accelerator will have consumed and emptied the queue faster than the accelerator can notice that the queue is no longer full and push more instructions. This explains why the TL MMIO integration performs worse than the RoCC integration since it has a noticeably longer delay, as can be seen in table 5.4. This means there are situations where the shorter communication time of the RoCC yields better performance since Memory-Level Parallelism (MLP) cannot be leveraged.

An important factor not tested in this thesis, and a significant advantage of RoCC, is contention on the main and peripheral bus. While the implementation does not suffer performance-wise, the current setup is simple, with a basic in-order processor without other accelerators. When dealing with more complicated SoC designs with multiple accelerators that also run applications in parallel, consuming unnecessary bandwidth on the interconnect buses can degrade the performance of BISMO and other processes and integrations running.

A tightly coupled accelerator also enables the accelerator to be integrated into the SoC pipeline. As mentioned in section 3.2, this gives access to the L1 cache and the PTW, among other things. Although this provides advantages such as access to virtual memory and data caching, it can also have some unwanted side-effects [57]. Firstly, caching can enable faster reads and writes, but cache pollution can compromise other applications running on the SoC. This can be solved by tweaking the cache protocols, but this would make the integration less portable, and finding an optimal setup when one has many moving parts can be daunting. Secondly, tightly coupled accelerators can have an issue with timing because it is normal for accelerators to run at the same clock frequency or a relative clock frequency to the core itself. With loose coupling, this is not necessary. This could result in the accelerator not running at its optimal frequency, diminishing the returns of the tight coupling. Lastly, having a tightly coupled accelerator can pose size requirements since it needs to fit in the CPU pipeline. Depending on the workload of BISMO, it could be desirable to aggressively scale the amount of DPUs or the memory in BISMO. When taking up silicone close to the cores, the resources are limited and more precious, and deciding what it should be used on is a science. This would make configurations very domain-specific and decrease portability. The L1 cache will most likely improve the performance of BISMO since it reuses tiles. This is not certain, and as it has yet to be implemented and tested, it could induce extra overhead with no real gain.

When comparing the tightly coupled RoCC integration with the loosely coupled MMIO integration of BISMO, it is also essential to consider the use case of the accelerator and how it functions. BISMO's likely use case would be a stand-alone inference accelerator. In that case, the ability to use multiple accelerators and aggressively scale the size for the workload would most likely yield the best performance. Using larger memory sizes and maybe even a cache in BISMO could also remove the potential advantages of an L1 cache. MMIO accelerator could also enable clocking down or even turning off the CPU while it runs, saving energy, although this would require large queues or hardware generation. This would not be possible with a RoCC accelerator. This points to BISMO being best suited as an MMIO accelerator, as the workload is also large, and the shorter communication time would usually be irrelevant for performance.

6.2 Hardware generation compared to software generation

The results regarding hardware versus software generation indicate two things. The hardware generation is more consistent and can always generate instructions fast enough, but the more complicated instructions require a longer delay. These effects were expected due to the tighter integration and the fact that the logic required is offloaded to hardware, increasing instruction size. However, the results for the time it took to generate the instructions themselves yielded different results than expected. Software generation yielded a significantly longer generation time. However, one would expect the software generation overhead to increase linearly with the number of instructions generated, which it did not.

In contrast, the instruction generation for the FSM should be constant, no matter the implementation size. This did not hold up either. The instruction for the hardware and software integration increases close to identically with the different array sizes. As this makes little sense, it is likely a side-effect of using the PK to handle memory. Since the proxy kernel induces a significant overhead on system calls and memory management, it most likely skews the benchmarks, making it difficult to evaluate correctly. This could also mean that the initial difference between the two implementations could be from the software generation having different constructs. This makes it hard to evaluate how much slower it would be in a regular application, but the instruction generation overhead when using the PK seems significant. One factor to consider is that when applying the accelerator for inference, one will most likely encounter the exact sizes repeatedly. In this case, the instructions can be stored and reused, making the initial generation of instructions insignificant when doing many iterations. This would come at the cost of using memory to store instructions. However, as seen in figure 4.1, the amount of instructions needed is small compared to the size of the matrices operated upon, which would make this memory usage insignificant, especially for large matrices.

The instruction to the hardware generator is not optimized and is passed as a single extensive instruction. Parts of the instruction could likely be interpreted from the configuration of BISMO, which would remove some initial delays. The instruction is also a single instruction with a decoupled interface. All stages' values must be communicated before any generation starts. A way to solve this could be to divide the instructions required for the generators into multiple decoupled interfaces and push the instruction for the fetch op and instruction generation first since the other two stages depending on the data from the fetch stage. This would hopefully make the initial delay of the hardware generation similar to that of the software generation, giving hardware generation an advantage since it can use smaller queues while constantly generating instructions fast enough. This was expected and is likely why most accelerators have one instruction interpreted in hardware.

Size analysis showed a $\approx 12.3\%$ LUT overhead. While this is a significant overhead, the implementation has not been designed with LUT usage in mind, meaning that there most likely is some low-hanging fruit in terms of LUT optimization. Since the generator does not depend on the implementation size, the relative overhead would diminish as the DPA increase in size. As the hardware and software generation implementation performed very similarly, assessing whether the increased overhead is worth it is difficult. One factor to consider in this case is that it further decoupled the accelerator from the processor. This makes it easier to assess the value of the accelerator since the time it would take to generate the instructions would be negligent, and it would, in the MMIO case, remove a lot of the pressure on the interconnect buses.

6.3 Configurations

This thesis has examined two aspects of coupling the BISMO accelerator with the Rocket Chip: instruction generation and communication. Since these two aspects can be combined into four configurations, it begs the question of which is the best suited. The results point towards hardware generation being the best for performance while reducing software requirements. There is little to warrant software generation other than giving the user more control of the accelerator, but in this case, it offers more of a burden than a distinct advantage. It also seems like there is little to justify an RoCC accelerator from the results. However, this thesis did not extensively explore all the advantages of the RoCC integration, like access to the L1 cache. What could be a great benefit of using RoCC is the access to the PTW, which would make it easier to integrate and use the accelerator as one would not have to deal with the memory management to secure continuous physical memory and send the physical memory address to the accelerator. Besides that, the TL MMIO interface offers much more flexibility while imposing only a minor communication penalty relative to the workload it executes. This configuration was not implemented, so more research would be required on both RoCC and MMIO integrations with hardware generation to give a definitive answer.

One could also consider hybrid hardware and software generation in different stages. One implementation that could be considered is only generating the instructions for the fetch stage in the software. Since the fetch stage needs to read data from main memory, which inherently has some delay, it will most likely never consume instructions faster than the CPU can provide them, as long as the queue is large enough to have a buffer. The fetch instructions are the most complex instructions to calculate, and this would also make it easier to meet timing in the hardware generators. There is also a question of whether the instructions cause the performance detriments that manifested in some cases or if it is the command tokens. As multiple command tokens are usually required for each stage per execution, generating these in hardware while generating the instructions in software could solve the performance issues and alleviate some of the pressure on the main bus. Since command tokens do not require complicated calculations, it would also drastically reduce silicone real-estate usage while reducing, if not outright removing, the need for pipelining. Moving the heavier generation to the processor would consume a lot of CPU cycles, making this trade-off hard to gauge, especially in embedded devices where timing is critical.

6.4 Integration with the Chipyard ecosystem

A critical contribution of this thesis was the integration of BISMO into Chipyard and Firesim. While the integration has been successful, it also comes with a few caveats. When implementing BISMO, both the proxy kernel and the AXI4ToTL tools in Chipyard had to be modified. While this works well for this specific implementation, this severely limits portability. It will be difficult for other projects to integrate and benchmark the BISMO accelerators with their design, which removes one of the significant advantages of the Chipyard ecosystem. During this thesis, the AXI4ToTL modification was also proven to not comply with the AXI standard, potentially creating undefined behavior. There is much to be desired regarding a streamlined and portable implementation, but it is still a significant step in the right direction and will hopefully enable further research.

6.5 Weaknesses

This thesis builds on earlier work, which made integration with the Chipyard ecosystem difficult and introduced many moving parts, making it difficult to gauge the performance of the implementations properly.

One issue with the current implementation is the tests themselves. When BISMO was initially developed, the focus of the paper [5] was the peak performance of the implementation, and it ignored the instruction generation aspect. This means that the instruction generation in the C++ tests is most likely not optimized, and this was not a focus of this thesis either. This entails that timings for the software-generated instructions could likely be optimized further, but to which extent is unknown. When pushing the instructions, the tests also check if the queue is full every time it pushes an instruction, which takes cycles, and remove MLP which could affect results. A better solution could be to check the available free entries in the current queue and push as many instructions as possible. The tests also did not look into cases where the number of instructions relative to the queue size became large. In this case, the queue size could be further reduced, but this would still not be realistic as some stages execute many instructions, then wait before it starts to execute again. If the queue size is small, this would nearly always yield bad results, but realistically there could be a specific queue size concerning memory where it would never have to wait for instruction.

The accelerator is also clocked to the same speed as the core, 3.2GHz, while it would run way slower in reality, maybe pushing 500MHz in an ASIC. This is an unrealistic speed for the accelerator and makes it consume instructions and data much faster than it would in an actual implementation. This could indicate that the CPU sometimes not pushing instruction fast enough is just a byproduct of the accelerator running too fast relative to the memory system and the CPU. 3.2 GHz is also optimistic for an in-order core. A more realistic clock speed for the CPU would be 1-2 GHz, making the accelerator run at 2-4x slower frequency than the core in a more realistic configuration.

Chapter 7

Conclusion

In this thesis, the BISMO accelerator was integrated into Chipyard in three ways to assess how its instructions can be communicated and generated. BISMO has also been successfully integrated with Chipyard, enabling further research.

The results in chapter 5 show that the extra delay in communicating all the instructions from the CPU is not significant in most cases unless the tiles fetched become large, at which point the RoCC implementation performs slightly better. The hardware generation implementation does not have this issue but suffers a longer initial delay due to the larger instruction format. However, this is offset by spending fewer CPU cycles generating instructions, making it faster in practice but comes at the cost of a higher LUT usage. What is the optimal BISMO configuration is difficult to say. Due to many moving parts and limited time, this thesis did not extensively explore all the advantages of RoCC or integrate any configuration into a more complicated design. However, from a portability, flexibility, and efficiency standpoint, a TL MMIO implementation with hardware generation would probably make the most sense, even though it was not implemented and benchmarked in this thesis.

This thesis sought to successfully implement BISMO with Chipyard, which it did, but with a few caveats. Hopefully, this will enable future work and research, which could fill the gaps in this thesis.

7.1 Future work

This section will present some avenues of improvement and future research for the experimental setup and the BISMO accelerator in itself.

A potential advantage of an RoCC accelerator is its access to the L1 cache. It could be interesting to connect BISMO to an L1 cache since it reuses data due to the nature of matrix multiplication. The FSM is also a potential avenue for further research. The FSM managed to generate instructions for the accelerator successfully, but the complexity of the state machine limited the clock speed of the

SoC. This could be resolved by pipelining some generators to meet the timing. As the accelerator has been able to clock at 200MHz, this would be a good target clock frequency. This is a substantial speedup and may not be feasible; hence, the instruction generators may be required to be clocked down relative to the accelerator. Another avenue for research is looking into clocking down BISMO relative to the Rocket core. The Rocket core and BISMO simulate at 3.2GHz, an unrealistic speed for BISMO. Clocking it to a realistic frequency would make the results comparable to other matrix multiplication accelerators and give a more definitive answer of what is optimal when experimenting with different configurations.

When improving BISMO [2], a new and improved DPU was implemented along with burst requests. The improved DPU and burst requests are currently written in Chisel 2 and must be ported to Chisel 3 to be integrated and benchmarked in Chipyard. The improved DPU traverses the tiles differently, meaning the software and hardware generation logic must be modified to accommodate it. Another thing is that the BISMO accelerator uses an AXI interface to communicate with memory. Although a conversion layer has proved sufficient, converting to an TL interface would make it easier to integrate, eliminate some overhead, and no longer depend on Chipyard conversion tools, which proved to have some difficulties.

As mentioned in the section 3.3, the FSM was implemented as a more accessible alternative to a processor for instruction generation. While this is acceptable, testing a custom processor to generate instructions could provide valuable insight and make the accelerators more flexible and customizable.

Bibliography

[1] A. Dobis, K. Laeufer, H. J. Damsgaard, T. Petersen, K. J. H. Rasmussen, E. Tolotto, S. T. Andersen, R. Lin, and M. Schoeberl, “Verification of Chisel Hardware Designs with ChiselVerify,” *Microprocessors and Microsystems*, vol. 96, p. 104737, 2023.

[2] Y. Umuroglu, D. Conficconi, L. Rasnayake, T. Preusser, and M. Sjalander, “Optimizing Bit-Serial Matrix Multiplication for Reconfigurable Computing,” *ACM Transactions on Reconfigurable Technology and Systems*, 2019.

[3] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, “The rocket chip generator,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, 2016.

[4] SiFive, “SiFive TileLink Specication,” May 2022, accessed 7. May 2023. [Online]. Available: https://starfivetech.com/uploads/tilelink_spec_1.8.1.pdf

[5] Y. Umuroglu, L. Rasnayake, and M. Sjalander, “BISMO: A Scalable Bit-Serial Matrix Multiplication Overlay for Reconfigurable Computing,” in *Field Programmable Logic and Applications (FPL), 2018 28th International Conference on*, ser. FPL ’18, 2018.

[6] A. Waterman and K. Asanovi ́c, “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20190608-Base-Ratified,” Mar. 2019, accessed 27. Apr. 2023. [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf

[7] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, “Design of ion-implanted mosfet’s with very small physical dimensions,” *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[8] G. E. Moore, “Cramming more components onto integrated circuits,” *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[9] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini, “A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives,” *Journal of Systems Architecture*, vol. 129, p. 102561, 2022.

[10] J. Misra and I. Saha, “Artificial neural networks in hardware: A survey of two decades of progress,” *Neurocomputing*, vol. 74, no. 1, pp. 239–255, 2010, artificial Brains.

[11] D. Zhang, N. Maslej, E. Brynjolfsson, J. Etchemendy, T. Lyons, J. Manyika, H. Ngo, J. C. Niebles, M. Sellitto, E. Sakhaee, Y. Shoham, J. Clark, and R. Perrault, “The AI Index 2022 Annual Report,” 2022.

[12] Y. Liu, T. Han, S. Ma, J. Zhang, Y. Yang, J. Tian, H. He, A. Li, M. He, Z. Liu, Z. Wu, D. Zhu, X. Li, N. Qiang, D. Shen, T. Liu, and B. Ge, “Summary of ChatGPT/GPT-4 Research and Perspective Towards the Future of Large Language Models,” 2023.

[13] Y. Guo, “A survey on methods and theories of quantized neural networks,” *arXiv preprint arXiv:1808.04752*, 2018.

[14] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, “Finn: A framework for fast, scalable binarized neural network inference,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 65–74.

[15] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs,” *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[16] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic, “FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 29–42.

[17] M. Ciletti, *Advanced Digital Design with the Verilog HDL*, ser. Prentice Hall Xilinx design series. Prentice Hall, 2011.

[18] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, “Chisel: Constructing hardware in a Scala embedded language,” in *DAC Design Automation Conference 2012*, 2012, pp. 1212–1221.

[19] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, “Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations,” in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 209–216.

[20] W. Snyder, “Verilator and systemperl,” in *North American SystemC Users’ Group, Design Automation Conference*, 2004.

[21] N. Kremeris. (2021) Verilator Pt.1: Introduction. (last visited: 2022-12-03). [Online]. Available:

https://itsembedded.com/dhd/verilator_1/

[22] A. Waterman, "Design of the RISC-V Instruction Set Architecture," Ph.D. dissertation, EECS Department, University of California, Berkeley, Jan 2016.

[23] S. I. Inc., "The SPARC Architecture Manual version 8," Dec. 2021, accessed 12. May 2023. [Online]. Available: https://www.gaisler.com/doc/sparcv8.pdf

[24] OpenRISC Community, "OpenRISC - OpenRISC," May 2022, accessed 12. May 2023. [Online]. Available: https://openrisc.io

[25] Arm, "Arm Architecture Reference Manual for A-profile architecture," *Arm*, Apr. 2023.

[26] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals," May 2023, accessed 12. May 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[27] M. D. Hill, D. Christie, D. Patterson, J. J. Yi, D. Chiou, and R. Sendag, "Proprietary versus Open Instruction Sets," *IEEE Micro*, vol. 36, no. 4, pp. 58–68, 2016.

[28] S. Chen, "A chip design that changes everything: 10 Breakthrough Technologies 2023," *MIT Technology Review*, Jan. 2023.

[29] I. Corporation, "Intel Launches $1 Billion Fund to Build a Foundry Innovation Ecosystem," May 2023, accessed 19. May 2023. [Online]. Available: https://www.intc.com/news-events/press-releases/detail/1525/intel-launches-1-billion-fund-to-build-a-foundry

[30] U. Farooq, Z. Marrakchi, and H. Mehrez, *FPGA Architectures: An Overview*. New York, NY: Springer New York, 2012, pp. 7–48.

[31] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," 2006.

[32] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[33] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*. Springer, 2016, pp. 21–37.

[34] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image style transfer using convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2414–2423.

[35] C. Badue, R. Guidolini, R. V. Carneiro, P. Azevedo, V. B. Cardoso, A. Forechi, L. Jesus,

R. Berriel, T. M. Paixao, F. Mutz *et al.*, "Self-driving cars: A survey," *Expert Systems with Applications*, vol. 165, p. 113816, 2021.

[36] E. Park, J. Ahn, and S. Yoo, "Weighted-entropy-based quantization for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5456–5464.

[37] T. Adiono, A. Putra, N. Sutisna, I. Syafalni, and R. Mulyawan, "Low Latency YOLOv3-Tiny Accelerator for Low-Cost FPGA Using General Matrix Multiplication Principle," *IEEE Access*, vol. 9, pp. 141 890–141 913, 2021.

[38] P. Gorlani, T. Kenter, and C. Plessl, "OpenCL Implementation of Cannon's Matrix Multiplication Algorithm on Intel Stratix 10 FPGAs," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 99–107.

[39] A. Ahmad and M. A. Pasha, "Optimizing Hardware Accelerated General Matrix-Matrix Multiplication for CNNs on FPGAs," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 11, pp. 2692–2696, 2020.

[40] C. Schmidt and A. Izraelevitz, "A fast parameterized SHA3 accelerator," in *tech. rep.* EECS Department, University of California, 2015.

[41] "NVIDIA Deep Learning Accelerator," Sep. 2019, accessed 1. May 2023. [Online]. Available: http://nvdla.org

[42] "6.7. MMIO Peripherals — Chipyard 1.9.0 documentation," Mar. 2023, accessed 1. May 2023. [Online]. Available: https://chipyard.readthedocs.io/en/stable/Customization/MMIO-Peripherals.html

[43] B. A. Research, "6.5. RoCC vs MMIO — Chipyard 1.9.0 documentation," Mar. 2023, accessed 20. May 2023. [Online]. Available: https://chipyard.readthedocs.io/en/stable/Customization/RoCC-or-MMIO.html

[44] "6.6. Adding a RoCC Accelerator — Chipyard 1.9.0 documentation," Mar. 2023, accessed 5. Jun. 2023. [Online]. Available: https://chipyard.readthedocs.io/en/stable/Customization/RoCC-Accelerators.html

[45] RISC-V, "Specifications – RISC-V International," May 2023, accessed 20. May 2023. [Online]. Available: https://riscv.org/technical/specifications

[46] Xilinx, "High Level Design," May 2023, accessed 20. May 2023. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/high-level-design.html

[47] S. Edwards, "The challenges of synthesizing hardware from c-like languages," *IEEE Design & Test of Computers*, vol. 23, no. 5, pp. 375–386, 2006.

[48] D. G. Bailey, "The advantages and limitations of high level synthesis for FPGA based image

processing," in *Proceedings of the 9th International Conference on Distributed Smart Cameras*, 2015, pp. 134–139.

[49] B. A. Research, "9.1. TileLink Node Types — Chipyard 1.8.1 documentation," Oct. 2022, accessed 20. May 2023. [Online]. Available: https://chipyard.readthedocs.io/en/1.8.1/TileLink-Diplomacy-Reference/NodeTypes.html

[50] Tynan McAuley, "TileLink questions and NVDLA performance," May 2023, accessed 22. May 2023]. [Online]. Available: https://groups.google.com/g/chipyard/c/dGCYZZB12is

[51] Chipsalliance, "Rocket-Chip," May 2023, accessed 23. May 2023. [Online]. Available: https://github.com/chipsalliance/rocket-chip/pull/2773

[52] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, "EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure," 2022.

[53] M. Guest, "The scientific case for high performance computing in europe 2012-2020," Technical report, FP7 Project PRACE RI-261557, Tech. Rep., 2014.

[54] Xilinx, "LUT," Aug. 2018, accessed 22. May 2023. [Online]. Available: https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/yeo1504034293627.html

[55] RISC-V Software, "RISCV Proxy Kernel and Boot Loader," May 2023, accessed 22. May 2023. [Online]. Available: https://github.com/riscv-software-src/riscv-pk

[56] V. Prasad and M. Desai, "Interconnect delay minimization using a novel pre-mid-post buffer strategy," in *16th International Conference on VLSI Design, 2003. Proceedings.*, 2003, pp. 417–422.

[57] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "An analysis of accelerator coupling in heterogeneous architectures," in *Proceedings of the 52Nd Annual Design Automation Conference*, 2015, pp. 1–6.

Appendix A

Simulations

Three different BISMO configurations in Chipyard exist, which can be seen in table A.1.

Name	Description
BISMOTLRocketConfig	TL configuration with software-generated instructions.
BISMORoCCRocketConfig	RoCC configuration with software-generated instructions.
BISMORoCCHWRocketConfig	RoCC configuration with hardware-generated instructions.

Table A.1: The three configurations.

A.1 Verilator

To run the Verilator simulation, one must first set up the Chipyard repository. The setup guide for this can be found at:
https://chipyard.readthedocs.io/en/stable/Chipyard-Basics/Initial-Repo-Setup.html
Instead of cloning from the official Chipyard repo, do the following:

```
git clone https://github.com/Vetleh/chipyard
git pull
git checkout 1.8.1-bismo-v1 // sw generation, 1.8.1-bismo-v1-hw for hw generation
./build-setup.sh riscv-tools
```

Once the repository is set up, the configuration has to be built. This can be done with the following commands:

```
cd sims/verilator


// Makes TL software generation
```

```
make CONFIG=BISMOTLRocketConfig debug

// Make RoCC software generation
make CONFIG=BISMORoCCRocketConfig debug

// Make RoCC hardware generation
make CONFIG=BISMORoCCHWRocketConfig debug
```

Where the CONFIG argument specifies which BISMO implementation one wishes to build. The debug is optional and enables waveforms. Once this is done, the statically linked C++ implementation binary can be built, and the simulation can run. Start in the root Chipyard folder, then type in the following command.

```
cd generators/bismo-config/src/main/tests/
make rocc-sw-debug
```

The correct test folder has to be accessed, and the debug argument should be there if the implementation was built with the debug flag; if not, emit it. The results will now be placed into the output folder.

A.2 Running on an FPGA

To run this design at an FPGA depends on the FPGA and toolchains used. If you have access to NTNU Idun FPGAs, the simulations can be run on the FPGAs following the following tutorial for the FPGA you wish to use on the wiki:
https://github.com/EECS-NTNU/chipyard/wiki

Below is an example of building, flashing, and running a config, where `CONFIG_NAME` should be replaced by the config name and `BUILD_NAME` is the name of the built bitstream. Absolute paths have to be used.

```
// Build bitstream.
make driver fpga PLATFORM=alveo ALVEO_PLATFORM=u250
↪  PLATFORM_CONFIG=BaseF1Config1Mem_F50MHz TARGET_CONFIG=CONFIG_NAME
↪  JAVA_HEAP_SIZE=16G

// Flash bitstream.
fpga-util.py -f d8 -b
↪  /chipyard/sims/firesim/sim/generated-src/alveo/BUILD_NAME/u250/vivado_proj/firesim.bit

// Run binary on FPGA.
```

```
./FireSim-alveo +permissive +mm_relaxFunctionalModel_0=0 +mm_writeMaxReqs_0=10
↪  +mm_readMaxReqs_0=10 +mm_writeLatency_0=30 +mm_readLatency_0=30 +slotid=d8
↪  +permissive-off /chipyard/toolchains/riscv-tools/riscv-pk/build/pk
↪  /path/to/binary
```