A Tightly Integrated Generic Instruction RISC-V Accelerator (TIGRA) for the Rocket Core

Theresa Thao Le  
*Clemson University*, t.le0193@gmail.com

Follow this and additional works at: [https://tigerprints.clemson.edu/all_theses](https://tigerprints.clemson.edu/all_theses)

**Recommended Citation**

Le, Theresa Thao, "A Tightly Integrated Generic Instruction RISC-V Accelerator (TIGRA) for the Rocket Core" (2021). *All Theses*. 3595.  
[https://tigerprints.clemson.edu/all_theses/3595](https://tigerprints.clemson.edu/all_theses/3595)

This Thesis is brought to you for free and open access by the Theses at TigerPrints. It has been accepted for inclusion in All Theses by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.
A Tightly Integrated Generic Instruction RISC-V Accelerator (TIGRA) for the Rocket Core

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

by
Theresa T. Lê
August 2021

Accepted by:
Dr. Melissa Smith, Committee Chair
Dr. Jon C. Calhoun
Dr. Walter Ligon
Abstract

Custom accelerators are largely beneficial for compute intensive applications such as data encryption or floating point arithmetic. These accelerators allow for a very specific task to be offloaded to its own unit so that the rest of the pipeline is not overwhelmed by these complicated instructions. To further achieve speed, a custom accelerator can be offloaded to an FPGA while still being on the same die as the CPU. Intel had announced this new technology back in 2014 and recently at the end of 2020, AMD released a patent application describing a similar approach.

In this thesis, we present a tightly coupled accelerator for the Rocket core, a RISC-V core that was developed at the University of California, Berkeley. This accelerator allows the user to develop their own custom logic that is part of the five stage pipeline but is abstracted away from execution units. This tightly coupled accelerator allows the user custom R-type instructions in the RISC-V ISA to use for their own applications.

We test the generic accelerator with the following three test applications: AES, posit addition, and the Rocket core’s ALU. All three applications execute without any additional latency and stalls the pipeline appropriately for instructions that execute in more than one clock cycle.
Dedication

I dedicate this thesis to my parents who with their fierce support and steadfast encouragement inspired me to continue to challenge myself in new ways in all aspects of my life. To my brother, who quickly became my first cheerleader, my first opponent, and my ally; you have stood by me through every decision in life. And finally, to my partner who never once stopped being the good man in a storm. Thank you for taking this giant leap with me.
Acknowledgments

Thank you to Dr. Smith, my committee chair, for helping guide my work these past two years. Your patience and advice have been invaluable to me both inside and outside the classroom. Thank you also to Drs. Calhoun and Ligon for your time and support by being part of my committee.

This work would not have been possible without Brad Green or Dillon Todd. Thank you for the camaraderie, for the late night debugging sessions, and for being my sounding board. I would not have been able to make it through without you both.

Thank you to the students of the FCTL lab, my friends at Clemson University, and the faculty and staff of the ECE department for your advice and support.

And finally, thank you to Dr. Cristinel Ababei of Marquette University for planting the seed long ago for me to pursue graduate school.
# Table of Contents

Title Page ................................................................. i  
Abstract ................................................................. ii  
Dedication ................................................................. iii  
Acknowledgments ........................................................ iv  
List of Tables ............................................................. vii  
List of Figures ............................................................ viii  
List of Listings ............................................................ ix  

1 Introduction ............................................................. 1  
  1.1 Motivation .......................................................... 1  
  1.2 Contributions ..................................................... 2  

2 Background ............................................................. 3  
  2.1 RISC-V ISA ......................................................... 3  
    2.1.1 Instruction Types ........................................... 4  
    2.1.2 RISC-V Extensions ......................................... 4  
  2.2 Rocket Core Architecture ....................................... 5  
    2.2.1 RoCC Interface ............................................. 6  

3 Related Work .......................................................... 8  
  3.1 Decoupled Accelerators ......................................... 8  
  3.2 Tightly Coupled Accelerators .................................. 9  
  3.3 Summary .......................................................... 9  

4 Research Design and Methods ...................................... 10  
  4.1 Design of TIGRA ................................................ 10  
  4.2 Modification in Chisel ......................................... 12  
    4.2.1 Rocket Core Configuration ............................... 12  
    4.2.2 Code Generation ......................................... 12  
    4.2.3 Netlist Viewer ............................................ 13  
    4.2.4 TIGRA Implementation in Scala ......................... 14  
    4.2.5 Configuration ............................................ 14  
    4.2.6 Integrating TIGRA in the Rocket Core .................. 15  
    4.2.7 Regenerating Code ....................................... 17  
  4.3 Modification in Verilog ........................................ 17  
  4.4 Summary ........................................................ 20
# Verification

5.1 Emulator Binary ................................................................. 21
5.2 RISC-V Benchmark Tests ....................................................... 22
5.3 Test Cases ................................................................. 22
  5.3.1 AES ............................................................... 22
  5.3.2 Posit Addition ...................................................... 29
  5.3.3 ALU ............................................................... 32
5.4 Summary .......................................................... 36

# Conclusions and Future Work ............................................. 38

6.1 Future Work ............................................................ 38
6.2 Conclusion ............................................................ 39

### Appendices .......................................................... 40

A Code Listings .............................................................. 41
B The Rocket Chip Generator Repository ................................ 69
  B.1 Check Out Repositories ............................................ 69
  B.2 Run Emulation ....................................................... 71
  B.3 Generate Waveforms ................................................ 71
  B.4 Generate Verilog Files ............................................. 72
C Create a Vivado Project ................................................... 74
  C.1 New Project Wizard .................................................. 74
  C.2 Troubleshooting ...................................................... 77
D Adding New Instructions to the RISC-V ISA ....................... 80
  D.1 Update the ISA ....................................................... 80
  D.2 Add the new instruction to the Spike ISA Simulator ........... 84

### Bibliography .......................................................... 86
List of Tables

5.1 The TIGRA instructions for AES ........................................... 24
5.2 The TIGRA instructions for AluTigra ..................................... 34
List of Figures

2.1 The different instruction types of the RV32I base ISA [1] ............................................. 4
2.2 Base opcodes for the RISC-V ISA [1] .............................................................................. 5
2.3 The location of the RoCC Interface in the Rocket pipeline [2] ..................................... 7
4.1 A RISC-V R-type instruction format [1] ................................................................. 11
4.2 An example Rocket system generated from the Rocket generator [3] .................... 13
4.3 The TIGRA interface as shown in Vivado ................................................................. 18
4.4 The high level design of TIGRA in the Rocket chip .................................................. 18
4.5 The high level design of TIGRA outside of the Rocket core at the system level ...... 19
4.6 A portion of the TIGRA interface shown in the Netlist Viewer in Vivado ............ 20
5.1 A high level diagram of tiny_aes implemented with TIGRA ................................. 24
5.2 Simulation results for tigra1_0 and tigra1_1 showing writes to each half of the key ............................................................................................................................. 27
5.3 Simulation results for tigra1_2 and tigra1_3 showing writes to each half of the state and triggering AES ............................................................................................... 27
5.4 Simulation results for tigra1_3 showing the stall in the pipeline and the resulting output data .................................................................................................................. 28
5.5 Simulation results for tigra1_4 and tigra1_5 showing reads of the bottom half of the AES output .................................................................................................................. 28
5.6 Simulation results for tigra1_6 and tigra1_7 showing reads of the top half of the AES output .................................................................................................................. 29
5.7 A high level diagram of PositAdder implemented with TIGRA ................................. 30
5.8 Simulation results for the beginning of posit addition, tigra0_0 ............................... 32
5.9 The completion of posit addition including the write back to memory .................. 32
5.10 A high level diagram of AluTigra implemented with TIGRA ................................. 33
5.11 Simulation results for the TIGRA implemented version of the ADD instruction ... 35
5.12 Simulation results for the TIGRA implemented version of the XOR instruction .... 35
5.13 Simulation results for the TIGRA implemented version of the OR instruction ...... 35
5.14 Simulation results for the TIGRA implemented version of the AND instruction .... 36
5.15 Simulation results for the TIGRA implemented version of the SUB instruction .... 36
5.16 Simulation results for the TIGRA implemented version of the SRA instruction .... 36
5.17 Simulation results for the TIGRA implemented version of the SLT instruction .... 36

1 Select Create Project ........................................................................................................... 74
2 Project name and location details .................................................................................... 75
3 Project type ....................................................................................................................... 75
4 Add files to the project .................................................................................................... 76
5 Source files for ExampleProj ........................................................................................ 77
6 Update to SystemVerilog for SimDTM.v ...................................................................... 78
7 Error caused by missing EICG_Wrapper.v file ............................................................. 79
List of Listings

1 The partial Scala file for the TIGRA decoder used in the interface ............... 41
2 The remaining implementation of the TIGRA interface .................................. 42
3 Updating the Instruction Decode unit with the TIGRA boolean and decode unit .. 43
4 Instantiation of the TIGRA decoder in RocketCore.scala ................................. 44
5 Instantiation of TIGRA in RocketCore.scala .................................................. 44
6 Updating the memory stage of the pipeline with TIGRA signals ....................... 45
7 Setting the writeback register of the Rocket core ......................................... 46
8 Stalling the pipeline if TIGRA has not completed ....................................... 46
9 Portions of the generated Verilog that shows new inputs and outputs for RocketCore 47
10 Portions of the generated Verilog that shows new inputs and outputs for RocketTile 47
11 Portions of the generated Verilog that shows new inputs and outputs for TilePRCIDomain 48
12 New wires created at the ExampleRocketSystem level and wired to the new TIGRA instance .............................................................. 48
13 The assembly code used to test AES .............................................................. 49
14 The commands needed to generate the AES test binary and dump the instructions to a text file ................................................................. 50
15 A portion of the dump file that shows the first four AES instructions that write the state and key values ................................................................. 50
16 Four AES instructions that complete reading the AES result shown in the dump file 51
17 A portion of the output file that shows how a TIGRA instruction is tested .......... 52
18 The assembly code used to test posit addition ............................................. 53
19 The commands needed to generate the posit addition test binary and dump the instructions to a text file ................................................................. 53
20 A portion of the dump file that shows the test cases for posits ....................... 54
21 A portion of the posits output file that shows how posit addition is tested .......... 55
22 The original ALU implemented in Scala showing the size of the instruction encoding and the valid instructions ......................................................... 56
23 A portion of the assembly code used to test AluTigra ................................... 57
24 The second half of the assembly code used to test AluTigra ............................ 58
25 The commands needed to generate the AluTigra test binary and dump the instructions to a text file ................................................................. 59
26 The partial dump file for testing AluTigra ADD instructions with original RISC-V instructions ................................................................. 59
27 The partial dump file for testing AluTigra ADD and the original RISC-V XOR instruction ................................................................. 60
28 The partial dump file for testing AluTigra XOR and the original RISC-V OR instruction 61
29 The partial dump file for testing AluTigra OR and the original RISC-V OR instruction 62
30 The partial dump file for testing AluTigra OR and the original RISC-V AND instruction 63
31 The partial dump file for testing AluTigra AND and the original RISC-V SUB instruction ................................................................. 64
The partial dump file for testing *AluTigra* SUB and the original RISC-V SUB instruction 65
The partial dump file for testing *AluTigra* SRA and the original RISC-V SRA instruction 66
The partial dump file for testing *AluTigra* SRA and the original RISC-V SLT instruction 67
The partial dump file for testing *AluTigra* SLT instruction . . . . . . . . . . . . . . . . . . . 68
Chapter 1

Introduction

1.1 Motivation

In 2018, Intel released the Intel® Xeon® Scalable Processor that integrates a CPU and an FPGA on the same die for tightly coupled acceleration [4]. The integrated FPGA is the Intel® Arria® 10 and while it has its own cache, it shares memory with the CPU over the Intel® Ultra Path Interconnect (UPI) bus. The CPU and the FPGA are tightly coupled making the data sharing very fast; however, there is an inherent latency when communicating between the two because of the bus. To achieve better performance, an accelerator should be tightly integrated with the CPU so that no latency is observed.

The difficulty with integrating such a tightly coupled accelerator is that a hardware designer needs to know the details of the core which with they are working in order to best exploit the hardware to achieve the best performance. Furthermore, tightly coupled accelerators can be very specific to the application for which they are designed making it difficult to modify if the designer would like to test out many applications with the same core. With loosely coupled accelerators, flexibility is offered because of an intermediate communication protocol, such as UPI, but again, we lose performance due to the latency of adding another interface.

In this research, we implement a tightly coupled interface that is generic enough so that multiple applications switch through this interface with little configuration. This interface does not add any additional latency to the original execution of the core. We use the open source ISA, RISC-V, and implement this interface on a 32-bit processor for simplicity.
1.2 Contributions

We present TIGRA, the Tightly Integrated Generic RISC-V Accelerator. This interface allows hardware designers to quickly integrate their custom logic into a RISC-V core without knowing the specifics of the core with which they are working. We use the University of California, Berkeley’s Rocket Chip generator and implement TIGRA in the Rocket core [5]. We test R-type instructions that execute in one clock cycle and R-type instructions that execute in multiple clock cycles. We develop TIGRA instructions as part of the RISC-V ISA and use these instructions to test our interface. We use TIGRA to test three applications, AES, posit addition, and the Rocket core’s native ALU, to show that our interface does not add any additional latency to our execution regardless of application or instruction length.

We organize this work into the following chapters. Chapter 2 details the background information about the RISC-V ISA and the Rocket Chip generator. Chapter 3 discusses related work, specifically loosely coupled and tightly coupled accelerators. We explain the design of TIGRA in Chapter 4. We discuss our verification strategy and results in Chapter 5. Finally, we discuss our future work in Chapter 6.
Chapter 2

Background

In this chapter, we introduce some key concepts of the RISC-V Instruction Set Architecture (ISA). We also discuss the technical details of the Rocket Core Architecture and its many applications, specifically for generating chips.

2.1 RISC-V ISA

The RISC-V ISA is a completely free and open ISA originally developed at UC Berkeley to aid in computer architecture research and education [1]. There are two primary base integer variants, called RV32I and RV64I which provides 32-bit or 64-bit address spaces respectively. The “I” in the base integer ISA stands for integer and contains integer computational instructions, integer loads, integer stores, and control-flow instructions. There is also room for future improvement for a RV128I variant which would support a 128-bit address space. There are a few more extensions to the ISA including the “M” extension for multiply and divide, the “A” extension for atomic read, modify and write memory instructions, the “F” extension for floating-point instructions, the “D” extension for double-precision floating-point instructions, and finally the “C” extension for compressed, narrower 16-bit forms of common instructions [1]. The IMAFD extensions come together to make the “G” or general extension. The ISAs are then called the RV32G for the 32-bit version or RV64G for the 64-bit version.
2.1.1 Instruction Types

There are four core instruction formats for the base RV32I ISA and RV64I ISA. These four types are R, I, S, and U. Two more variants, B and J are also included based on the handling of immediates as shown in Figure 2.1 [1].

R-type instructions are register-register instructions where rs1 and rs2 are source registers and rd is the destination register. Operations are done using the two source registers and the result is stored in the destination register. I-type instructions are register-immediate instructions where an immediate value, imm(x), is used instead of the rs2 register. These immediate values are either small data types or partial data. U-type instructions are another form of immediate instructions and are used for loading upper immediate values to build 32-bit constants or to add the upper immediate value to the program counter (pc).

Control transfer instructions use both the J-type instructions for unconditional jumps and the B-type instructions for conditional branches. Finally, load and store instructions are the only instructions that access memory. Loads use the I-type instructions and stores use the S-type instructions.

2.1.2 RISC-V Extensions

With all these different instruction types, the RISC-V ISA has a rich portfolio of instructions that do a wide variety of tasks for general-purpose computing. The complete list of instructions and their functionality is found in [1], Chapter 25. The ISA also allows for modification so that users
can add in new extensions as they see fit. These extensions are highly specialized for a specific application and are implemented using the custom-0 and custom-1 bits that are available for this purpose. Further modifications are made using the custom-2 and custom-3 bits that will eventually be used for the RV128I base instructions, however, for this work, only the custom-0 and custom-1 bits are used to implement generic instructions.

<table>
<thead>
<tr>
<th>inst[4:2]</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111 (≥ 32b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 LOAD</td>
<td>LOAD-FP</td>
<td>custom-0</td>
<td>MISC-MEM</td>
<td>OP-IMM</td>
<td>AUIPC</td>
<td>OP-IMM-32</td>
<td>48b</td>
<td></td>
</tr>
<tr>
<td>01 STORE</td>
<td>STORE-FP</td>
<td>custom-1</td>
<td>AMO</td>
<td>OP</td>
<td>LUI</td>
<td>OP-32</td>
<td>64b</td>
<td></td>
</tr>
<tr>
<td>10 MADD</td>
<td>MSUB</td>
<td>NMSUB</td>
<td>NMADD</td>
<td>OP-FP</td>
<td>reserved</td>
<td>custom-2/rev128</td>
<td>48b</td>
<td></td>
</tr>
<tr>
<td>11 BRANCH</td>
<td>JALR</td>
<td>reserved</td>
<td>JAL</td>
<td>SYSTEM</td>
<td>reserved</td>
<td>custom-3/rev128</td>
<td>≥ 80b</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.2: Base opcodes for the RISC-V ISA [1]

2.2 Rocket Core Architecture

For this work, we use the open-source, RISC-V core named Rocket. Rocket Chip is a System on Chip (SoC) generator that was originally developed by the University of California, Berkeley and is now supported by SiFive [3]. The Rocket Chip generator is written in Chisel, which is a hardware construction language embedded in Scala [6]. Chisel is used to take full advantage of Scala properties such as object oriented programming and functional programming. The Rocket Chip generator is a Scala program that invokes the Chisel compiler in order to generate the RTL in Verilog to describe a complete SoC [7]. It allows the designer a plug and play environment so they can swap out substantial design components such as cache sizes or an optional floating point unit (FPU) [5]. It consists of six sub components:

1. Core Generator
2. Cache Generator
3. Rocket Custom Coprocessor Interface (RoCC) compatible coprocessor generator
4. Tile Generator
5. TileLink Generator
6. Peripherals
The Rocket Chip Generator allows for two different cores to be used: the Rocket core which is a five-stage in-order scalar core or the Berkeley Out-of-Order (BOOM) core. Both cores support the RV64G ISA and the Rocket core also supports the RV32G ISA. The cache generator allows for a family of caches with configurable sizes, associativities and replacement policies to be created. The RoCC coprocessor generator will be explained in more detail in Section 2.2.1, but in summary, it allows an interface for designers to create their own loosely coupled accelerators. The Tile and TileLink Generator creates cache-coherent tiles and networks for these agents as well as the associated cache controllers. Finally, the peripherals are generators for AMBA-compatible buses [5].

For this work, a very simple configuration is selected. The main key components of this configuration include, a 32-bit Rocket Core, a multiply and divide unit, a 16 KB data cache, and a 4 KB instruction cache.

### 2.2.1 RoCC Interface

The Rocket Custom Coprocessor (RoCC) interface allows the user to create their own decoupled custom coprocessor for specific applications [5]. The Rocket core and the attached coprocessor communicate through *command* and *response* interfaces. These commands are stored into queues and wait until either the accelerator or the core is ready to receive the data. These commands are instructions executed by the Rocket core and can include values in up to two integer registers (e.g. *rs1* and *rs2* for R-type instructions). Commands can also only write into one register. The coprocessor shares the Rocket core’s data cache and page table walker and is allowed to interrupt the core so these coprocessors are constructed to use a page-based virtual memory system.

The RoCC interface aims to solve a similar problem as the TIGRA interface. The difference between these two interfaces is the latency. [2] performed a latency study that showed that reading from the RoCC interface always incurred a penalty of 4-5 clock cycles. To show this latency, first, [2] retrieves the resultant data from the coprocessor and stores it in a register. Immediately after, [2] uses it in an *addi* instruction, which is an instruction that is part of the RISC-V ISA. In addition to using the retrieved data immediately, [2] interleaves a *nop* instruction in an effort to hide the latency. The idea behind using a number of *nop* instructions is to show some other computation that has no dependence on the data from the coprocessor. Regardless of how many *nop* instructions are used, there is always a latency penalty of 4-5 clock cycles.

One potential reason why this latency penalty exists is because of the location of the RoCC
interface in the Rocket pipeline as seen in Figure 2.3. The RoCC interface first receives custom instructions in its command router in the WB stage of the pipeline [2]. From there, the command router then selects the custom coprocessor to forward the custom instruction. This is an extra step that is needed before the instruction is processed. The other reason why the latency penalty exists is because of how the core handles the write-back of data that is coming from the RoCC interface [2]. The core preempts the writing of the data from the interface in favor of the write in the register file of the instruction that is currently in the WB stage. This means that data coming back from the RoCC interface needs to wait until a “non-writing” instruction is present in the WB stage.

The TIGRA interface does not incur a latency penalty. Because it is a tightly coupled accelerator interface, the designer uses their own custom instructions that uses their custom logic as if it is part of the original pipeline. We do not use any additional steps to write data back from the interface. The pipeline simply stalls until the instruction completes its execution.
Chapter 3

Related Work

In this chapter, we explore a few other designs that are similar to the TIGRA interface. These designs include decoupled accelerators and tightly coupled accelerators. The decoupled accelerators shown here have a latency penalty to their design while the tightly coupled accelerators are very specific to their applications and do not offer room for flexibility.

3.1 Decoupled Accelerators

Decoupled accelerators have an interface that separates the accelerator from the core architecture. The value in having one interface is that it allows users to easily implement many different accelerators or co-processors that interact with the core. The RoCC interface is exactly this and has been used for many different applications. [2] uses the RoCC interface to create a block-cipher co-processor which performs an AES-128 encryption in 45 clock cycles, meaning the co-processor takes in a 128-bit key and correctly encrypts data. The entire operation of the coprocessor including the reading and writing through the RoCC interface takes 48 clock cycles, showing the latency penalty that occurs when using this interface. These results were verified via simulation.

[8] develops an accelerator for digital signal processing applications that performs addition, multiplication, and linear combination. [8] shows that the energy consumption of the accelerator was reduced by 40-50× with respect to the software execution and achieves a speed-up of up to 100× over the software implementation depending on the image size. Similarly, [9] creates an accelerator with the RoCC interface for decimal computation and yields about a 2.3× speedup with their
software-hardware co-design implementation.

3.2 Tightly Coupled Accelerators

Tightly coupled accelerators are advantageous because they do not introduce any additional latency. Designers can make very custom changes to best exploit the hardware for their application. However, the disadvantage of using a tightly coupled accelerator is that because these accelerators are application specific, they are not easily swapped out like the decoupled accelerators can be. Some tightly coupled accelerators may require specific modifications to the core pipeline that may not be needed for other applications. This is not very flexible as it puts a heavy burden on the designer to understand both the core and the application thoroughly to make the necessary modifications.

[10] implements and integrates a posit processing unit into the Rocket chip generator. This PPU replaces the FPU and successfully supports the single precision and double precision floating point extensions. [11] develops ten bit manipulation instructions in an accelerator named BitALU. The BitALU is another component in the execute stage of the pipeline, similar to the ALU. It does not impact the critical path of the Rocket core. Similarly, [12] creates five new instructions for accelerating the HEVC Deblocking Filter and achieves an 11.6% improvement. In this work, instead of creating an entirely new unit like [10] and [11], [12] implements a new 128-bit register file in addition to the new instructions to accommodate the large data size needed to accelerate the algorithm. The new instructions replace the frequently used sequences and functions of the algorithm and uses the new register file to achieve the 11.6% improvement.

3.3 Summary

Decoupled accelerators offer flexibility but add overhead to the overall execution time of the application. Tightly coupled accelerators eliminate this overhead; however, their design requires in-depth knowledge of the RISC-V ISA and the RISC-V core in order for it to be implemented properly. TIGRA is a generic interface that allows for tightly coupled accelerators to be implemented only requiring the designer to have detailed knowledge about their application.
Chapter 4

Research Design and Methods

This chapter explores the design methodology behind implementing TIGRA. We first explore the high level design of TIGRA and how the interface works with the existing Rocket core pipeline. Next, we modify code in Chisel to first generate the Verilog code for the core. We then further modify code in Verilog to complete the design of the core. Finally, in order to test the custom instructions that are used with the TIGRA interface, we update the RISC-V GNU toolchain. All code listings are in Appendix A.

4.1 Design of TIGRA

This section provides a high-level design of the TIGRA interface. This section details the signal names and their purposes, the RISC-V instruction type used with TIGRA, and how to encode the instruction.

TIGRA is an interface that allows the designer to create their own tightly coupled interface using generic instructions. As mentioned in Chapter 3, a tightly coupled accelerator is advantageous because the designer makes very specific changes to best exploit the hardware; however, this makes the design inflexible. Tightly coupled accelerators are frequently intended for one application; therefore, the developed instructions are specific for only that application and cannot be reused. The advantage of using TIGRA is that the interface allows for synchronization with the core pipeline, and the instructions used to interface with the accelerator are generic enough that they follow the same format as a RISC-V, R-type instruction.
The TIGRA interface has seven signals that the designer must use to interface between the core and their custom logic. These signals are: `insn`, `rs1`, `rs2`, `rd`, `out_valid`, and `valid`. There are also clock and reset signals.

The `insn` signal is the currently executing instruction. It follows an R-type instruction format; `rs1` and `rs2` are the decoded register values from the instruction and `rd` is the decoded destination register. We see in Figure 4.1 the value for the `rs1` register stored in bits 19-15, the value for the `rs2` register stored in bits 24-20, and the value stored for the `rd` register stored in bits 11-7. Before the accelerator processes the custom instruction, the `valid` bit must be high to indicate there are no issues from earlier stages such as a stall in the pipeline or an invalid decoded instruction. The `valid` signal states that the data on `rs1` and `rs2` are the correct data and ready to be processed.

Once the custom logic finishes its work, the `out_valid` signal goes high to show that the data is correct and ready to be written back. At this point, the write-back stage of the pipeline takes the data computed by the accelerator and store it in the `rd` register.

There may be instances where the designer develops an instruction that does not need to write back to the destination register. An example of this instruction would be a store instruction. When this occurs, the behavior of the instruction is similar to a NOP instruction where the destination register is set to register 0. Alternatively, the designer can decide to place a unique value in the destination register to verify the instruction executed correctly. Once the instruction finishes its execution, then the actual value in the `rd` register is compared to the expected value to verify that the custom logic behaves correctly. This method is used for one of the test applications, AES, detailed in Section 5.3.1.

Instructions that use the TIGRA interface have the major opcode `0b0001011` or `0b0101011`. These major opcodes are reserved for custom instructions and guaranteed to not have any issues with further RISC-V instruction sets such as 128-bit instructions as noted in Figure 2.2. The `funct3` and `funct7` fields are available for the designer to differentiate different functionality within the custom logic.
4.2 Modification in Chisel

This section details the modifications that we make in Chisel to implement TIGRA in Rocket. Note that we use SCALA to reference the file path, ROCKETCHIP/src/main/scala/ in the rocket-chip repository, which contains the scala source code. The SCALA/rocket subfolder contains all of the files that are pertinent to the core. The surrounding subfolders are other units that support the core such as SCALA/amba or SCALA/jtag. ROCKETCHIP refers to the file path where the rocket-chip repository is downloaded. Details on how to download source code for the Rocket core are found in Appendix B.

4.2.1 Rocket Core Configuration

Chisel is a hardware construction language that generates the RTL to implement the Rocket chip. We first download the repository for the Rocket Chip Generator, compile the code, and run emulation on it to ensure that all RISC-V tests pass properly before any modifications can be made. The Rocket chip is flexible and the designer selects from a number of pre-made configurations specified in SCALA/system/Configs.scala to generate the Verilog code to compile. We select TinyConfig which implements a TinyCore, a 32-bit processor with no floating point unit, no virtual memory, and no branch target buffer unit. The TinyCore has a multiply and divide unit, a 16 Kb data cache and a 4 Kb instruction cache. We select this configuration because it is the simplest. Further details on downloading code through emulation are found in Appendix B.

4.2.2 Code Generation

We then identify the files to modify in order to implement TIGRA. After generating the Verilog code, we see that the main Verilog file for TinyConfig is already over 160,000 lines of code; therefore, we write the code for TIGRA in Chisel first and take advantage of the generator handling the details that are abstracted out by writing in a higher-level language. Rather than meticulously reading through Chisel or Verilog we view the architecture through FPGA tools, specifically a Netlist Viewer. For this work, we use Xilinx’s Vivado Design Suite. Instructions on how to create a project are found in Appendix C.
4.2.3 Netlist Viewer

Upon inspection of the Netlist Viewer in Vivado, we find that the Rocket core is a sub-unit four levels deep from the top-most level unit. The top-most level unit is called `ExampleRocketSystem`. The entire system includes many parts of an SoC (System on Chip) including a memory system, an MMIO (Memory-Mapped Input/Output) subsystem, and a DMA (Direct Memory Access) subsystem. The `ExampleRocketSystem` instantiates a `TilePRCIDomain` unit. This unit implements the translation lookaside buffers (TLBs) and synchronization units. This level is where a `RocketTile` is placed. `RocketTile` instantiates a `RocketCore` which contains the code we modified to implement TIGRA. Alongside the core is a PTW (Page Table Walker), an L1 instruction cache, and an L1 data cache. Figure 4.2 shows an example Rocket system.
4.2.4 TIGRA Implementation in Scala

One construct of interest in the Rocket core is the SCIE which stands for Simple Custom Instruction Extension. This construct was partially created but never fully implemented by the creators of Rocket. The SCIE allows the designer to use two source registers and one destination register for a custom instruction using the custom0 or custom1 opcodes. It has a separate decoder so that the designer specifies a pipelined or unpipelined version of their instruction. However, it appears incomplete because no mechanism exists for stalling the processor if the instruction takes multiple clock cycles to finish executing before writing back to memory. When using the SCIE in the Rocket core, we import the scie package from SCALA/scie. We model our implementation after the SCIE and create the SCALA/tigra directory and the Tigra.scala file for this interface.

Listing 1 shows the TIGRA decoder portion of the interface. Line 1 creates a new package for our interface called tigra and lines 3-5 import necessary Chisel packages to implement the interface. We first declare a Tigra object in lines 7-9 and only allow certain opcodes for the instruction. As previously described in Section 4.1, only the major opcodes 0b0001011 or 0b0101011 will be decoded as TIGRA instructions. We also specify a default iLen which refers to the instruction length. This value can be either 32 or 64 depending on if the designer wants to use RV32 or RV64, respectively.

Lines 12-38 create a decoder for TIGRA which is needed in the instruction decode phase of the core. The decoder takes in the instruction and if it matches the TIGRA opcode, will set the pipelined output to 1. The TigraDecoder extends, or inherits, from the BlackBox class. This extension will create a new file named TigraDecoder.v as shown in line 20, and will populate it with the contents of the setInline() function in lines 22-36.

Listing 2 details the remaining implementation of the TIGRA interface. Lines 40-49 create a new class named TigraInterface and sets up the inputs and outputs as described in Section 4.1. Finally, the last class in lines 51-90 generate a new Verilog file named Tigra.v and writes out the inputs and outputs the designer can use to insert their custom logic. The code that will be populated in the Tigra.v file is example code that is intended to be replaced.

4.2.5 Configuration

Similar to how TinyConfig is a pre-made Rocket core configuration, we develop a new configuration specific for TIGRA. We start by making a parameter, useTIGRA, the designer toggles to
use the TIGRA interface. We add this parameter to the CoreParams structure in SCALA/tile/Core.scala and initially set it to false to prevent the Rocket core from instantiating TIGRA for other configurations that do not use it. We create a new Core configuration in SCALA/subsystem/Configs.scala and model it after the TinyCore. We set the useTIGRA parameter to true, enabling the TIGRA interface, and name the new core TigraCore. We then modify SCALA/system/Configs.scala to create a new configuration named TigraConfig, modeled after TinyConfig which we used previously.

4.2.6 Integrating TIGRA in the Rocket Core

4.2.6.1 Decode Stage

We instantiate the TIGRA interface in the Rocket core and begin by making changes to the instruction decode stage of the pipeline. This change is made in SCALA/rocket/IDecode.scala. Listing 3 shows a portion of the control signals structure, IntCtrlSigs and the addition of a boolean named tigra. These control signals are used in later stages in the Rocket core to properly route the instructions to the correct functional units for execution. The tigra boolean is set when a TIGRA instruction has been fetched properly from the instruction buffer. We create a TigraDecode class that sets up other control signals necessary for the instruction to execute properly. Listing 3 shows how the configuration is modeled after the SCIEDecode class.

We next instantiate TIGRA in the Rocket core. This instantiation takes place inside SCALA/rocket/RocketCore.scala. We first import the TIGRA package and continue modeling the modifications after the SCIE. Listing 4 shows the new TigraDecode class in the decode_table so that when an instruction is fetched, it is properly decoded as a TIGRA instruction. We create a new module named id_tigra_decoder which will be created if the useTIGRA parameter is set to true. This decoder is the new module that we created in Listing 1.

4.2.6.2 Execute Stage

In the execute stage, we modify what controls the valid input to TIGRA, shown in Listing 5. TIGRA is valid if the following three conditions are met:

- the valid bit from the execute register, ex_reg_valid, is high
- a TIGRA instruction has been decoded
• the useTIGRA parameter is set to true

The ex_reg_valid signal is high when the instruction is valid, there are no interrupts, there are no stalls in the pipeline, there are no instruction replays, and there are no branch mispredictions. A TIGRA instruction is decoded when the instruction matches the correct major opcode and the useTIGRA parameter is set to true. The user sets the useTIGRA parameter to true to use the TIGRA interface or uses the pre-defined configuration, TigraConfig, referenced in Section 4.2.5. Once these conditions are met, the custom logic executes based upon insn, rs1, and rs2.

4.2.6.3 Memory Stage

After the execute stage of the pipeline, we enter the memory stage where we need to set up additional signals to properly write back the results from TIGRA in the next stage. Listing 6 shows the additional code that was added to the existing memory stage. First, we create new mem_tigra signals that is set to the respective execute stage’s signals. One signal is mem_tigra_pipelined which indicates that the TIGRA instruction is now going through the memory stage. The other signal is mem_tigra_out_valid which is set by the TIGRA interface to show if the instruction has completed and the output data is valid. These signals are needed in the writeback stage to determine if TIGRA data should be written back to the register file.

The next signal we update is the mem_reg_wdata signal. This signal holds the data that will be written back to a register in the register file. We only want data to be written back from TIGRA when the TIGRA instruction has completed. Otherwise, the data that should be written back should come from the ALU. The TIGRA output data comes from tigra.io.rd and setting it to mem_reg_wdata here ensures that data is written correctly for a single clock cycle instruction. For a multi clock cycle instruction, mem_reg_wdata is set outside of the elsewhen block. If TIGRA needs to stall, then ex_pc_valid will drop to 0 after the first clock cycle and remain 0 until the TIGRA instruction has completed. We do this additional write to mem_reg_wdata because the first write to this register for a multi clock cycle TIGRA instruction is not guaranteed to be correct.

Writing to this register outside of the elsewhen block ensures that the completed result is stored correctly so that it will be written back to a register in the register file. The mem_reg_wdata and mem_tigra_out_valid signals are updated only when the TIGRA instruction has completed.
4.2.6.4 Writeback Stage

Listing 7 shows two modifications in the writeback stage of the pipeline to accommodate TIGRA. The first modification is to the \texttt{wb\_reg\_valid} signal where it will also be set high if the TIGRA instruction has completed executing, noted by \texttt{mem\_tigra\_out\_valid}, and the current instruction in the memory stage is a TIGRA instruction, noted by \texttt{mem\_ctrl\_tigra}. When \texttt{wb\_reg\_valid} is high, it will trigger a register write and the data that is in \texttt{wb\_reg\_wdata} will be written to a register. \texttt{wb\_reg\_wdata} will get its data from \texttt{mem\_reg\_wdata} when the current instruction in the memory stage is a TIGRA instruction and the output is valid. Similar to the memory stage, this is done to ensure that the completed result from TIGRA is written to the register file.

Finally, we stall the pipeline in the decode stage if TIGRA has not completed yet. This stall is done in the decode stage to prevent additional instructions to be sent to the execute stage while the current instruction in the execute stage is a TIGRA instruction. While this stall will block future instructions from executing, it allows currently executing instructions in the memory stage or the writeback stage to complete. Stalling in the decode stage stalls future stages in the pipeline. We set an extra condition for the \texttt{ctrl\_stallid} value shown in Listing 8.

4.2.7 Regenerating Code

We regenerate the Verilog code following the steps in Appendix B, Section B.4. We view the Rocket core in the Netlist Viewer of Vivado and see that TIGRA was created successfully inside the Rocket core. With the majority of the work completed in Chisel, we make modifications in Verilog to lift the TIGRA interface outside of the core and make additional ports as needed to handle the logic for the inputs and outputs. Figure 4.3 shows the TIGRA interface. Figure 4.4 shows the high level design of where the TIGRA interface fits on the Rocket chip.

4.3 Modification in Verilog

To finish the design of the TIGRA interface, we pull this interface out of the Rocket core and place it onto the \texttt{ExampleRocketSystem}. Figure 4.5 shows a high-level diagram of where this interface is placed.

Listing 9 shows the first modification we make at the \texttt{RocketCore} level. We remove the instantiation of the TIGRA interface and create new input and output ports for the Rocket core.
Figure 4.3: The TIGRA interface as shown in Vivado

Figure 4.4: The high level design of TIGRA in the Rocket chip
Figure 4.5: The high level design of TIGRA outside of the Rocket core at the system level definition. The inputs for the TIGRA interface become outputs for the Rocket core, and the outputs become the inputs. These ports are created and retain the same naming convention as the TIGRA interface but prefixed with `tigra_core` to show how the ports relate to each other at the next level. The original signal names were prefixed with `tigra_` and these signals are replaced with the new signals.

We repeat this process in `RocketTile` where we create new input and output ports. These ports are prefixed with `core_domain_tile`. We map the `tigra_core` signals to the `core_domain_tile` signals. This modification is shown in Listing 10. We similarly do the same for the `TilePRCIDomain` unit where we map the `core_domain_tile` to `tile_prci` signals, shown in Listing 11. We want the TIGRA interface to be in the `ExampleRocketSystem` therefore we instantiate the interface here and create wires to link it to `TilePRCIDomain`. These wires are prefixed with `prci_top`. We map the `tile_prci` signals to the `prci_top` signals. We map the `prci_top` signals to the TIGRA instance to complete the interface. Listing 12 shows this final modification.
Figure 4.6: A portion of the TIGRA interface shown in the Netlist Viewer in Vivado

4.4 Summary

We identify and make modifications in Chisel in the decode unit and the Rocket core to implement TIGRA. We develop a new package to implement the TIGRA interface and integrate it into the Rocket code base. After generating the Verilog files from the Chisel files, we manually lift the TIGRA interface out of the Rocket core, implementing extra input and output ports as necessary. Finally, we modify the configuration files to make switching between different Rocket configurations easier.
Chapter 5

Verification

This chapter details how we verify the correctness of TIGRA. TIGRA is considered to be correct if it does not add any additional latency in executing single cycle or multi-cycle instructions. TIGRA instructions must perform correctly and accurately. TIGRA’s value is in the hardware designer’s ability to quickly swap out different applications to use with the interface. In this chapter, we first discuss the original RISC-V tests that must pass with the new modifications. We next discuss the different test cases used to verify TIGRA. We then discuss how to create new RISC-V tests to test our applications properly. Finally, we discuss the results of our test cases.

5.1 Emulator Binary

Before the Verilog modifications are made, it is useful to run emulation with TIGRA in the Rocket core to ensure that this new unit does not break any existing functionality. Running emulation can be done by following the steps in Appendix B.2. This step will take all the scala files to generate a new emulator binary and run all the RISC-V benchmark tests. These benchmark tests test each RISC-V instruction for correct behavior. After modifications are made in Verilog to lift TIGRA out of the core, we can simply make run again in the $ROCKETCHIP/emulator folder and the modified Verilog files will be used to create the new emulator binary. This command will also rerun all RISC-V benchmark tests.
5.2 RISC-V Benchmark Tests

The RISC-V benchmark tests are part of rocket-tools repository and are created as part of the tool setup detailed in Appendix B.1.1. It is recommended that all tests pass whenever any changes are made to the Rocket core. These tests simply test the functionality of each instruction to ensure its correctness. The tests are divided into folders based upon the which ISA is being tested: RV32I or RV64I. The tests that we are most interested in are in the rv64ui folder. A large amount of the R-type instructions are saved here, and we can model our TIGRA instructions after these tests. Each test file tests one instruction therefore we can leverage the testing setup for our tests. The rv64ui tests test the RV64I ISA at the user-level and only contain integer instructions [13]. There is also a rv32ui folder that is a subset of the rv64ui tests but for the RV32I ISA. Before we can begin writing tests, we must add our TIGRA instructions to the RISC-V ISA. These instructions can be found in Appendix D.

5.3 Test Cases

Three different test cases verify TIGRA successfully: AES-128, posit addition, and the Rocket core’s native ALU. Posit multiplication and division are not tested here because both units use code proprietary to Xilinx which we do not have access to because we use Verilator to verify TIGRA. We highlight single cycle instructions with AES and how multiple instructions are needed to fully encrypt 128 bits. We then show posit addition, a multi-cycle instruction where TIGRA successfully stalls the pipeline to allow for the full execution of the addition. Finally, we show how the Rocket core’s native ALU behaves identically to the TIGRA implemented ALU and adds no additional latency.

5.3.1 AES

5.3.1.1 Background

AES stands for Advanced Encryption Standard and was designed by Belgian designers Joan Daemen and Vincent Rijmen [14]. The algorithm is a block cipher algorithm that takes in a 128-bit plaintext value, named state, as its input and uses a 128-bit key to encrypt the data. The key sizes can be 128, 192 or 256 bits long and is what is used to differentiate between the AES variants. For
this work, we use AES-128. The state goes through multiple rounds of four steps: SubBytes where each byte of the state array is replaced with another from a lookup table; ShiftRows where the rows of the state are shifted over a certain number of bytes; MixColumns which is a mixing operation that operates on the columns of the state; and finally AddRoundKey where each byte of the state is XOR'd with a byte of the round key [14]. The round key is derived based upon the input key and the key schedule. The output after the encryption is a 128-bit value. AES is a symmetric key algorithm meaning that the same key is used for both encryption and decryption.

5.3.1.2 Implementing AES with TIGRA

Understanding the specifics of AES is not necessary in order to implement it with TIGRA. We need to know the inputs, outputs and expected values. From there, we can creatively use TIGRA instructions to best facilitate AES.

We use tiny_aes from [15] as our custom logic to integrate with TIGRA. TigraConfig is a 32-bit processor and we have two inputs, state and key that are 128 bits wide. To load these inputs, we break up each value into four parts that are 32 bits wide. For the state value, we load state[31:0] into the rs1 register and state[63:32] into the rs2 register. This can be done in one instruction, tigra1_0. To load the upper 64 bits, we can do the same thing with a second instruction, tigra1_1. Recall from Appendix D, that the TIGRA instructions are labeled, tigra<custom-#>_<funct3 value>_<funct7 value>. For just loading the state value, the value placed in the rd register will not be useful therefore we can assign a unique value to each instruction’s rd register simply to see that the instruction completed correctly.

Similar to how the state values are loaded to tiny_aes, we can load key values with instructions tigra1_2 and tigra1_3. When tigra1_3 starts, however, tiny_aes will start the encryption. The tiny_aes core will constantly encrypt using the state and key values, therefore we gate the clock coming into the core so that it only begins the encryption when tigra1_3 starts. The core does not have any additional outputs to notify the user when encryption has completed, however [15] states that AES-128 takes 21 clock cycles for the core to output the encrypted state value. Therefore, we create a counter that counts to 21 and sets TIGRA’s out_valid to 1 after it has completed counting. The out_valid signal is set to 0 while the encryption is ongoing.

After tiny_aes has completed encryption, we need to read back the data. Again, the output data will be 128 bits therefore we break up the read into four instructions. Assuming that out_data
Table 5.1: The TIGRA instructions for AES

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Minor Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tigra1.0</td>
<td>000</td>
<td>Write key[63:0]</td>
</tr>
<tr>
<td>tigra1.1</td>
<td>001</td>
<td>Write key[127:64]</td>
</tr>
<tr>
<td>tigra1.2</td>
<td>010</td>
<td>Write state[63:0]</td>
</tr>
<tr>
<td>tigra1.3</td>
<td>011</td>
<td>Write state[127:64] and begin AES encryption</td>
</tr>
<tr>
<td>tigra1.4</td>
<td>100</td>
<td>Read AES output, out_data[31:0]</td>
</tr>
<tr>
<td>tigra1.5</td>
<td>101</td>
<td>Read AES output, out_data[63:32]</td>
</tr>
<tr>
<td>tigra1.6</td>
<td>110</td>
<td>Read AES output, out_data[95:64]</td>
</tr>
<tr>
<td>tigra1.7</td>
<td>111</td>
<td>Read AES output, out_data[96:127]</td>
</tr>
</tbody>
</table>

For these four instructions, the `rs1` and `rs2` fields will not be used. Table 5.1 shows a summary of the AES instructions. Figure 5.1 shows the completed design of tiny_aes implemented with TIGRA.

5.3.1.3 Test Design

As mentioned in Section 5.2, to test AES, we leverage the riscv-tests that are part of the rocket-tools directory. The entry code for initializing the execution environment as well as the linker script to correctly map the entry section and the main program’s code are part of the rocket-tools repository. We also leverage the testing macros that are written to test the validity of each instruction. Listing 13 shows a specific macro that we use to test AES, TEST_RR_OP. This
macro has five arguments. The first argument is simply a number meant to identify a given test case. The second argument is the name of the instruction to be tested. The third argument is the expected rd value and finally, the fourth and fifth arguments are the rs1 and rs2 values respectively.

To test AES, we want to check that for a given state and key, we get the correct encrypted value out. The tiny_aes source code also came with known values as part of the testbench therefore we use these values to test TIGRA’s implementation of AES. The instructions tigra10 and tigra11 load in the key value, and we expect some value in the rd register. The instructions tigra12 and tigra13 load in the state value and will also have some value in the rd register. Note that tigra13 will start the execution of AES which takes 21 clock cycles while all the previous instructions take one clock cycle. Even though tigra13 will take significantly longer to execute, we expect that the stall in the pipeline will still result in the correct value being placed in the rd register when it has completed execution.

After the completion of the loads and the execution of AES, we read back the result from tiny_aes. We write expected values for AES in the rd register for the next four instructions. Because we are simply reading the result, the rs1 and rs2 registers are not used therefore their values do not matter. Listing 13 shows one full test scenario; however, a second test scenario is executed but omitted from this listing. If all instructions return the expected values, then the entire test will complete with a “PASSED” flag. Otherwise if at any given time an instruction does not return the expected value, then the test will exit early and complete with a “FAILED” flag.

After the test code has been written, we run the two commands in Listing 14 to generate the test binary and dump the instructions to a text file. These two commands are modeled after the commands that are already used to generate all the other test binaries and can be found in the Makefile for riscv-tests. We modify the command and use the 32 bit RISC-V GCC compiler instead of the 64 bit version. We specify the general 32 bit architecture as well. We use the ILP32 ABI which is the standard integer calling convention only; hardware floating-point calling convention is not used [16]. We run the objdump command on the AES test binary to see what exactly will be tested. This step is optional but can be helpful in debugging the actual output against the expected output.

Listing 15 shows test.2 which is the TEST RR OP macro expanded on line 33 in Listing 13. We see that the program counter is at 0x800000fc and that the first instruction is a LUI or “load upper immediate”. This instruction is encoded as 0x09cf50b7. Immediately following this
instruction is a ADDI or “add immediate”. These two instructions together load the lower 32 bits of the state value, 0x9cf4f3c, into the ra register. The next two instructions load the next 32 bits of the state value into the sp register. We then execute the tigra1_0 instruction and set the value of the rd register which in this case is a4. We load the expected value for tigra1_0 by using the LI or “load immediate” instruction. This expected value is placed into the t2 register. Recall that the value in the rd register does not matter for these load instructions but a unique value is placed there to check for proper execution. The gp register is updated and the BNE or “branch not equal” instruction is called next. If the a4 register and the t2 register do not match each other, then we branch to the fail portion of the assembly where we exit from the test. If the two values in the registers match, then we continue with the program execution and move onto the next test case. All eight of the test cases follow this sequence of instructions.

We use the emulator binary we created before to test this AES binary. We run the following command in ROCKETCHIP/emulator:

```
./emulator-freechips.rocketchip.system-freechips.rocketchip.system.TigraConfig-debug
  +max-cycles=10000000 +verbose -voutput/rv32ui-p-aes.vcd output/rv32ui-p-aes 3>&1 1>&2 2>&3 |
  /home/ttle/Research/rocket-tools/bin/spike-dasm > output/rv32ui-p-aes.out
```

This command will call the emulator binary for TigraConfig and time out after a certain number of cycles. With the verbose flag, we can generate additional output files that help with debugging including a .vcd file that will be used with gtkwave to view the waveform and a *.out file which is a detailed text file that shows the cycle by cycle dump of the writeback stage of the pipeline. We run the emulator on the AES binary, rv32ui-p-aes, which we copied over from the rocket-tools directory into ROCKETCHIP/emulator/output.

5.3.1.4 Test Results

After we run the test, we see in Listing 17 that the entire test passed. This means that all actual values match expected values. We continue to view the waveform and inspect each instruction.

Figure 5.2 shows the execution of tigra1_0 and tigra1_1. The rs1 value and the rs2 value set the bottom half of the key when tigra1_0 executes. Similarly, tigra1_1 sets the top half of the key. The break between tigra1_0 and tigra1_1 represents five clock cycles. During this time, the pipeline loads the expected value, compares it against the actual value in rd, and continues to load
Figure 5.2: Simulation results for tigra1.0 and tigra1.1 showing writes to each half of the key the next values into rs1 and rs2 for tigra1.1 because the actual value matches the expected value.

Figure 5.3 shows the execution tigra1.2 and the beginning of tigra1.3. tigra1.2 sets the bottom half of state. tigra1.3 sets the top half of state and starts AES. Immediately when tigra1.3 begins, out_valid drops to 0 and ctrl_stalld goes high. On the next clock cycle, ctrl_killx goes high because of the stall in the decode stage. This shows that we do not have a new, valid instruction decoded yet. Similarly, ctrl_killm follows the next clock cycle and goes high to show that we do not have a valid instruction for the memory stage yet.

Figure 5.3: Simulation results for tigra1.2 and tigra1.3 showing writes to each half of the state and triggering AES

Figure 5.4 shows the completion of tigra1.3. The break shows sixteen clock cycles where tigra1.3 executes and no signal values change. Once encryption finishes, data_out has the encrypted data and out_valid goes high. This change triggers to ctrl_stalld to go low again and on
the next clock cycle, the next instruction will be loaded into the execute stage. \texttt{ctrl.killx} will go low on the next clock cycle showing that the instruction loaded in the execution stage can now execute. Similarly, \texttt{ctrl.killm} will follow and go low indicating that the memory stage has a valid instruction.

Figure 5.4: Simulation results for \texttt{tigra1.3} showing the stall in the pipeline and the resulting output data

We read the result from AES using \texttt{tigra1.4} through \texttt{tigra1.5}. Figure 5.5 shows the execution of \texttt{tigra1.4} and \texttt{tigra1.5}. The instructions execute in one clock cycle and place a quarter of \texttt{data.out} into \texttt{rd}. The \texttt{tigra1.4} and \texttt{tigra1.5} instructions together read the bottom half of the data. The break between \texttt{tigra1.4} and \texttt{tigra1.5} represents five clock cycles where the pipeline again does its comparison between the expected and actual values.

Figure 5.5: Simulation results for \texttt{tigra1.4} and \texttt{tigra1.5} showing reads of the bottom half of the AES output
5.3.2 Posit Addition

5.3.2.1 Background

Posits are designed to be an alternative to the IEEE 754 floating-point standard [17]. They have a larger dynamic range and higher accuracy when compared to floats and take less circuitry than an IEEE float FPU [17]. [10] developed a drop in replacement for the Rocket core’s FPU and successfully modified the RISC-V ISA test suite to use posit arithmetic on their Posit Processing Unit (PPU). For this work, we use PACoGen, one of the first open source HDL for posit arithmetic, and integrate it with TIGRA [18]. We specifically use the posit addition core with TIGRA.

5.3.2.2 Implementing Posit Addition with TIGRA

We use the pipelined version of the posit addition unit that is supplied with PACoGen. We name this unit \textit{PositAdder} and it takes in a 32-bit posit with a 6-bit exponent size. Our \textit{TigraCore} is a 32-bit processor therefore there is little we need to change to integrate the \textit{PositAdder}. The \textit{PositAdder} has four inputs: \textit{clk}, \textit{in1}, \textit{in2}, and \textit{start}. We set their inputs to TIGRA’s \textit{clock}, \textit{rs1} and \textit{rs2} signals respectively. The \textit{start} signal will require decoding. We take in the \textit{insn} signal from TIGRA and determine if it matches the TIGRA instruction we designate for posit addition, \textit{tigra0.0}. This instruction means we are using \textit{custom0} and the \textit{funct3} value is 0. If the decoded instruction matches \textit{tigra0.0} and the \textit{valid} bit is high, then we can set the \textit{start} signal to high. The wire that accomplishes this is named \textit{add_start}.

The \textit{PositAdder} has two output signals: \textit{out} and \textit{done}. The \textit{out} signal is the result of the
Figure 5.7: A high level diagram of PositAdder implemented with TIGRA

posit addition. We set that signal to TIGRA’s rd output through the wire named add_res. The done signal indicates that the PositAdder has completed execution. It is originally set to 0 and will be set to 1 when it finishes. We set this done signal to TIGRA’s out_valid signal through the wire named add_done. Figure 5.7 shows the completed modifications needed to integrate PositAdder with TIGRA.

5.3.2.3 Test Design

We design the posit addition test the same way we design the AES test described in Section 5.3.1.3. Listing 18 shows the different ways we use the TEST RR OP macro. We first test two known RISC-V instructions to ensure that our modifications in TIGRA do not have any effect on the ISA. We next test tigra0.0 and use known posit representations for our rs1 and rs2 values. We select 0x42100000 which represents the decimal value 18 for rs1 and 0x40C00000 which represents the decimal value 3 for rs2. We expect the sum to be the decimal value 21, which is represented as 0x42280000. After we execute this instruction, we test one more known RISC-V instruction again simply to ensure that TIGRA does not affect the rest of the ISA.

Listing 19 shows the commands we use to generate the test binary and dump the instructions to a text file. Listing 20 shows the test cases for the mul and add, our two known instructions. Posit
addition is tested in test_4 where it follows the same sequence of instructions that have been used for all TEST_RR_OP macros. We first load values into the rs1 and rs2 registers. We then execute tigra0_0 and store the result in the a4 register. We then load the expected value into register t2 and compare the expected value to the actual value. If the two values in the registers match, then we continue with the program execution, otherwise we will branch to the fail portion of the assembly.

We use our emulator binary we created for TigraConfig and run it on the posit addition test binary, rv32ui-p-posit. This command is the same as before for AES and will create a .vcd file to view the waveform and .out file to view the cycle by cycle dump of the writeback stage.

5.3.2.4 Test Results

We view the .out file that we generate and see in Listing 21 that all tests pass. In the output file, we see that there are two tigra0_0 instructions written to the file. This is because posit addition takes multiple clock cycles to finish executing therefore the stall causes two writes in the writeback stage for this instruction. This can happen, but it’s important to note that we save the correct answer by the end of the execution. We see this occur in the waveform shown in Figure 5.8. At the beginning of execution, we save some value into the rd register but because out_valid is low, we know this value to be incorrect. We see at the end of the execution that the correct value is in rd. This value, highlighted in orange, is 0x4228000 which is the posit representation for the decimal number 21. The break during tigra0_0’s execution is a two clock cycle break where no signal values are changing.

Figure 5.8 also shows the add_start and add_done signals. As we expect, add_start goes high to begin the execution of PositAdder. While the PositAdder computes the sum, out_valid is low which triggers ctrl_stalld to go high preventing future instructions from executing. One clock cycle later, ctrl_killx goes high and ctrl_killm follows in the next clock cycle. Once the addition has completed, add_done goes high which immediately triggers out_valid to go high as well.

Figure 5.9 shows the completion of posit addition. While the load upper immediate (LUI) instruction is in the execute stage, the rd register’s value moves in mem_reg_wdata in preparation for the writeback stage. In the next clock cycle, we write the value to the appropriate register in the register file.

31
5.3.3 ALU

5.3.3.1 Background

The arithmetic logic unit (ALU) is our last test case. It is responsible for performing arithmetic operations such as addition and subtraction as well as bitwise operations such as AND or OR operations. The Rocket core already has a predefined ALU written in Chisel. We generate the Verilog code when we generate the rest of the Rocket chip, as explained in Section 4.2.2 and use the Verilog to make our own TIGRA version of the ALU, named \textit{AluTigra}. We do this comparison against an already developed module to ensure that TIGRA does not add any additional latency to its execution.
5.3.3.2  Implementing the ALU with TIGRA

When we generate the Verilog code for the ALU, we find that the \textit{io\_fn} input is what differentiates the instructions. Listing 22 shows the number of bits needed to decode the instruction and the list of instructions. We pick seven instructions to implement with \textit{AluTigra}. These instructions are ADD, XOR, OR, AND, SUB, SRA (shift right arithmetic), and SLT (set less than). For SRA, the value placed in the \textit{rd} register is the value in \textit{rs1} shifted right by the number of bits specified in the \textit{rs2} register. The sign bit will be replicated to fill in the leftmost bits. For SLT, the \textit{rd} register is set to 1 if the value in \textit{rs1} is less than the value in \textit{rs2}. Otherwise, it will be set to 0. We pick these seven instructions because the RISC-V test suite already contains test cases for these instruction. All other instructions are either repetitive or they do not exist in the test suite.

In order to use the instructions with \textit{AluTigra}, we need to further decode the instruction. Previously for AES and posit addition, we used a major opcode and the minor opcode, \textit{funct3} to differentiate between instructions. The \textit{funct3} opcode is only three bits wide, therefore we use the additional minor opcode, \textit{funct7} and decode these bits to determine which instruction \textit{AluTigra} will execute. We use \textit{custom0} and set \textit{funct3} to the value four. We vary the \textit{funct7} field to match what is shown in Listing 22 for each instruction. Table 5.2 shows how the TIGRA instructions match with the ALU instructions.

![Figure 5.10: A high level diagram of AluTigra implemented with TIGRA](image)

33
Table 5.2: The TIGRA instructions for AluTigra

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Minor Opcode</th>
<th>RISC-V Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>tigra0_4</td>
<td>0000</td>
<td>ADD</td>
</tr>
<tr>
<td>tigra0_4_04</td>
<td>0100</td>
<td>XOR</td>
</tr>
<tr>
<td>tigra0_4_06</td>
<td>0110</td>
<td>OR</td>
</tr>
<tr>
<td>tigra0_4_07</td>
<td>0111</td>
<td>AND</td>
</tr>
<tr>
<td>tigra0_4_10</td>
<td>1010</td>
<td>SUB</td>
</tr>
<tr>
<td>tigra0_4_11</td>
<td>1011</td>
<td>SRA</td>
</tr>
<tr>
<td>tigra0_4_12</td>
<td>1100</td>
<td>SLT</td>
</tr>
</tbody>
</table>

We decode the instruction to determine when AluTigra will execute. AluTigra can only execute when the valid bit is high, custom0 is part of the major opcode, and the funct3 field is set to four. We use the bottom four bits of the funct7 field as the input value for io_fn. These bits are bits 28 to 25 of the insn word. We set TIGRA’s rs2 and rs1 to AluTigra’s io_in2 and io_in1 respectively. AluTigra has two outputs, io_out and io_adder_out. Upon inspection in the Rocket core, we find that io_out is used to write back data to memory. We set this value to our rd register through the wire named alu_out_data. All the ALU instructions take one clock cycle to execute therefore out_valid is set to high. Figure 5.10 shows the high level diagram of how we implement AluTigra.

5.3.3.3 Test Design

We test AluTigra the same way we have tested AES and posit addition. There is no need to write any new test cases; the ALU instructions already have their own test suite therefore we use three test cases from each instruction to test with our instructions. Listings 23 and 24 show how we first test the RISC-V instructions and then test our TIGRA instructions.

Listing 25 shows the commands we use to build our test binary, rv32ui-p-alutigra. Listings 26 through 35 show our dump file with our new TIGRA instructions. Our tests follow the same pattern of loading in values for rs1 and rs2, executing either the RISC-V instruction or the TIGRA instruction, then comparing the expected value with the actual value. If at any time the two values do not match, we expect to branch to the fail portion of the assembly and exit the test.

5.3.3.4 Test Results

We run the test and find in the output file that all tests pass. This means that not only do the original RISC-V instructions produce the expected values but our TIGRA instructions with
the same functionality also produce the same expected values. Figures 5.11 through 5.17 show the
waveforms that were produced for each instruction. In between each instruction, we indicate a
break. During this break, the pipeline is finishing out the test case by loading in the expected value,
comparing it with the actual, and loading in the next rs2 and rs1 values for the next instruction. We
observe expected behavior for all the AluTigra instructions and note that each instruction takes one
clock cycle to complete. The RISC-V equivalent instructions also take one clock cycle to complete
therefore there is no change in latency when the ALU is implemented with TIGRA.

Figure 5.11: Simulation results for the TIGRA implemented version of the ADD instruction

Figure 5.12: Simulation results for the TIGRA implemented version of the XOR instruction

Figure 5.13: Simulation results for the TIGRA implemented version of the OR instruction
5.4 Summary

We test TIGRA on three applications each with unique characteristics. For AES, we find that TIGRA can successfully execute both single cycle instructions and multi clock cycle instructions. We use multiple instructions in order to accommodate the 128 bit inputs and outputs we need to properly encrypt data using our 32-bit processor. We test TIGRA with posit addition, a multi clock cycle instruction, and we observe a proper pipeline stall with the correct answer stored properly in
the register file. Finally, we test TIGRA with the Rocket core’s ALU and note that no added latency was added by using our interface. All three applications had minor modifications to properly decode the TIGRA instructions for successful execution of the custom logic.
Chapter 6

Conclusions and Future Work

6.1 Future Work

There are a number of items that we can implement to further extend this work. The first item that would be most beneficial is new instruction types. For this work, we implement R-type instructions only. Other instruction types include I, U, or S instructions, as explained in Section 2.1.1. These instructions can be implemented and will require further modifications inside the Rocket core. Once this work is completed for the 32-bit core we created, we can expand to 64-bit cores allowing for larger instructions to execute.

Another item to expand on is implementing more TIGRA instructions. Currently, the only TIGRA instructions that are implemented in the ISA are the ones that we use to test our applications. There are \(2^{11}\) possible instructions when taking into account the major and minor opcodes. These instructions can be implemented as an extension to the RISC-V ISA and will allow a hardware designer flexibility when selecting instructions for their design. Furthermore, the naming scheme of the TIGRA instructions is not descriptive enough for a given application. The TIGRA instructions are labeled as \(tigra<custom-\#>_.<funct3\ value>._.<funct7\ value>\) as noted in Appendix D. While these instructions are generic and can easily be reusable for other use cases, reading through the assembly will not provide any details about the application. Therefore, some kind of aliasing technique where the hardware designer can create their own name for their instructions but still apply it to the TIGRA instructions would help make the assembly code be more meaningful.

We can also further extend this work on other RISC-V processors. [19] developed the same
design for TIGRA for the PicoRV32 processor with similar results where no additional latency was added. By extending this work to other RISC-V processors, we show how versatile TIGRA is with different implementations. This will give hardware designers a variety of cores to select from depending on the tradeoffs of each.

6.2 Conclusion

We develop TIGRA for the Rocket core and handle both single cycle and multi cycle instructions. We properly stall the pipeline and write the correct result by the time the TIGRA instruction finishes executing. We create TIGRA instructions so that the RISC-V assembler will recognize our custom instructions. We successfully implement three custom logic applications and show that no additional latency is added due to the interface. With this generic interface, a hardware designer simply needs to understand their application to integrate it into the Rocket core with TIGRA. The designer no longer needs to know low-level details of the Rocket core, saving them time when testing their custom logic. This interface will allow for even more rapid prototyping enabling future designs to be realized more quickly.
Appendices
Appendix A  Code Listings

Listing 1: The partial Scala file for the TIGRA decoder used in the interface
class TigraInterface(xLen: Int) extends Bundle {
    val clock = Input(Clock())
    val valid = Input(Bool())
    val insn = Input(UInt(Tigra.iLen.W))
    val rs1 = Input(UInt(xLen.W))
    val rs2 = Input(UInt(xLen.W))
    val reset = Input(Reset())
    val rd = Output(UInt(xLen.W))
    val out_valid = Output(Bool())
}

class Tigra(xLen: Int) extends BlackBox(Map("XLEN" -> xLen)) with HasBlackBoxInline {
    val io = IO(new TigraInterface(xLen))

    setInline("Tigra.v",
        s""
            |module Tigra #(parameter XLEN = 32) (input clock,
            |    input valid,
            |    input [${Tigra.iLen-1}:0] insn,
            |    input [XLEN-1:0] rs1,
            |    input [XLEN-1:0] rs2,
            |    input reset,
            |    output [XLEN-1:0] rd,
            |    output out_valid);
            |
            |// Placeholder code
            |reg[XLEN-1:0] temp;
            |reg    temp_valid;
            |always @(posedge clock) begin
            |    if (valid) begin
            |        temp <= rs1 + rs2;
            |        temp_valid <= 1;
            |    end
            |    else begin
            |        temp <= temp;
            |        temp_valid <= 0;
            |    end
            |end
            |
            |assign rd = temp;
            |assign out_valid = temp_valid;
            |endmodule
       ***.stripMargin)
    }

Listing 2: The remaining implementation of the TIGRA interface
class IntCtrlSigs extends Bundle {
    val legal = Bool()
    val fp = Bool()
    val rocc = Bool()
    val branch = Bool()
    val jal = Bool()
    val jalr = Bool()
    val rxs2 = Bool()
    val rxs1 = Bool()
    val scie = Bool()
    val tigra = Bool()
    val sel_alu2 = Bits(width = A2_X.getWidth)
    val sel_alu1 = Bits(width = A1_X.getWidth)
    val sel_imm = Bits(width = IMM_X.getWidth)
}

class SCIEDecode(implicit val p: Parameters) extends DecodeConstants {
    val table: Array[(BitPat, List[BitPat])] = Array(
        // scie
        // |
        // scie
        // |
        SCIE.opcode-> List(Y,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X, DW_XPR,FN_X, N,M_X,
                        N,N,N,N,Y,CSR.N,N,N,N,N)
    )
}

class TigraDecode(implicit val p: Parameters) extends DecodeConstants {
    val table: Array[(BitPat, List[BitPat])] = Array(
        // tigra
        // |
        // tigra
        // |
        Tigra.opcode-> List(Y,N,N,N,Y,Y,N,A2_ZERO,A1_RS1, IMM_X, DW_XPR,FN_X, N,M_X,
                        N,N,N,N,Y,CSR.N,N,N,N,N)
    )
}

Listing 3: Updating the Instruction Decode unit with the TIGRA boolean and decode unit
val decode_table = {
  require(!usingRocC || !rocketParams.useSCIE)
  (if (usingMulDiv) new MDecode(pipelinedMul) ++: (xLen > 32).option(new M64Decode(pipelinedMul)).toSeq else Nil) ++:
  (if (usingAtomics) new ADecode ++: (xLen > 32).option(new A64Decode).toSeq else Nil) ++:
  (if (fLen >= 32) new FDecode ++: (xLen > 32).option(new F64Decode).toSeq else Nil) ++:
  (if (fLen >= 64) new DDecode ++: (xLen > 32).option(new D64Decode).toSeq else Nil) ++:
  (if (minFLen == 16) new HDecode ++: (xLen > 32).option(new H64Decode).toSeq ++: (fLen >= 64).option(new HDecode).toSeq else Nil) ++:
  (usingRocC.option(new RocCDecode)) ++:
  (rocketParams.useSCIE.option(new SCIE Decode)) ++:
  (rocketParams.useTIGRA.option(new TIGRA Decode)) ++:
  (if (xLen == 32) new I32Decode else new I64Decode) ++:
  (usingVM.option(new SVMDecode)) ++:
  (usingSupervisor.option(new SDecode)) ++:
  (usingDebug.option(new DebugDecode)) ++:
  (usingNMI.option(new NMIDecode)) ++:
  Seq(new FenceIDecode(tile.dcache.flushOnFenceI)) ++:
  coreParams.haveCFlush.option(new CFlushDecode(tile.dcache.canSupportCFlushLine)) ++:
  Seq(new IDecode) flatMap(_.table)
}

val id_tigra_decoder = if (!rocketParams.useTIGRA) Wire(new TigraDecoderInterface) else {
  val td = Module(new Tigra)
  td.io.insn := id_raw_inst(0)
  td.io
}
}

Listing 4: Instantiation of the TIGRA decoder in RocketCore.scala

/* New TIGRA interface added */
val tigra = Module(new Tigra(xLen))
tigra.io.clock := Module.clock
tigra.io.valid := ex_reg_valid && ex_tigra_pipelined
tigra.io.insn := ex_reg_inst
val tigra.rs1 := ex_rs(0)
tigra.rs2 := ex_rs(1)
tigra.reset := reset

Listing 5: Instantiation of TIGRA in RocketCore.scala
when (mem_reg_valid && mem_reg_flush_pipe) {
  mem_reg_sfence := false
}.elsewhen (ex_pc_valid) {
  mem_ctrl := ex_ctrl
  mem_scie_unpipelined := ex_scie_unpipelined
  mem_scie_pipelined := ex_scie_pipelined
  mem_tigra_pipelined := ex_tigra_pipelined
  mem_tigra_out_valid := tigra.io.out_valid
  mem_rvc := ex_reg_rvc
  mem_load := ex_ctrl.mem && isRead(ex_ctrl.mem_cmd)
  mem_store := ex_ctrl.mem && isWrite(ex_ctrl.mem_cmd)
  mem_sfence := ex_sfence
  mem_btb_resp := ex_reg_btb_resp
  mem_flush_pipe := ex_reg_flush_pipe
  mem_slow_bypass := ex_slow_bypass
  mem_reg := ex_reg
  mem_reg_cause := ex_cause
  mem_reg_inst := ex_reg_inst
  mem_reg_raw_inst := ex_reg_raw_inst
  mem_reg_mem_size := ex_reg_mem_size
  mem_reg_pc := ex_reg_pc
  mem_reg_wdata := Mux(ex_tigra_pipelined && tigra.io.out_valid, tigra.io.rd, alu.io.out)
  mem_br_taken := alu.io.cmp_out
}

when (ex_ctrl.rx2 && (ex_ctrl.mem || ex_ctrl.rocc || ex_sfence)) {
  val size = Mux(ex_ctrl.rocc, log2Ceil(xLen/8).U, ex_reg_mem_size)
  mem_reg := new StoreGen(size, 0.U, ex_rs(1), coreDataBytes).data
}

when (ex_ctrl.jalr && csr.io.status.debug) {
  // flush If on D-mode JALR to effect uncached fetch without DL flush
  mem_ctrl.fence_i := true
  mem_flush_pipe := true
}

}.elsewhen(ex_tigra_pipelined && tigra.io.out_valid) {
  mem_reg_wdata := tigra.io.out_valid
  mem_tigra_out_valid := tigra.io.out_valid
}

Listing 6: Updating the memory stage of the pipeline with TIGRA signals
Listing 7: Setting the writeback register of the Rocket core

Listing 8: Stalling the pipeline if TIGRA has not completed
Listing 9: Portions of the generated Verilog that shows new inputs and outputs for RocketCore

Listing 10: Portions of the generated Verilog that shows new inputs and outputs for RocketTile
/* Create new inputs/outputs for TilePRCIDomain */
output tile_prci_clock,
output tile_prci_valid,
output [31:0] tile_prci_insn,
output [31:0] tile_prci_rs1,
output [31:0] tile_prci_rs2,
output tile_prci_reset,
input [31:0] tile_prci_rd,
input tile_prci_out_valid

/* Set new inputs/outputs of TilePRCIDomain */
.tile_prci_clock(prci_top_clock),
.tile_prci_valid(prci_top_valid),
.tile_prci_insn(prci_top_insn),
.tile_prci_rs1(prci_top_rs1),
.tile_prci_rs2(prci_top_rs2),
.tile_prci_reset(prci_top_reset),
.tile_prci_rd(prci_top_rd),
.tile_prci_out_valid(prci_top_out_valid)

Listing 11: Portions of the generated Verilog that shows new inputs and outputs for TilePRCIDomain

/* New wires for TIGRA at ExampleRocketSystem level */
wire prci_top_clock;
wire prci_top_valid;
wire [31:0] prci_top_insn;
wire [31:0] prci_top_rs1;
wire [31:0] prci_top_rs2;
wire prci_top_reset;
wire [31:0] prci_top_rd;
wire prci_top_out_valid;

/* Instantiate TIGRA at ExampleRocketSystem level */
Tigra #(.XLEN(32)) tigra ( // @[RocketCore.scala 427:22]
          .clock(prci_top_clock),
          .valid(prci_top_valid),
          .insn(prci_top_insn),
          .rs1(prci_top_rs1),
          .rs2(prci_top_rs2),
          .reset(prci_top_reset),
          .rd(prci_top_rd),
          .out_valid(prci_top_out_valid)
          );

Listing 12: New wires created at the ExampleRocketSystem level and wired to the new TIGRA instance
#include "riscv_test.h"
#include "test_macros.h"

RVTEST_RV64U
RVTEST_CODE_BEGIN

#-------------------------------------------------------------
# Test AES encryption using TIGRA instructions. Instructions
# are encoded as the following:
#-------------------------------------------------------------
# tigra1_0 -- write key [63:0]
# tigra1_1 -- write key [127:64]
# tigra1_2 -- write state [63:0]
# tigra1_3 -- write state [127:64] and begin AES encryption
# tigra1_4 -- read result [31:0]
# tigra1_5 -- read result [63:32]
# tigra1_6 -- read result [95:64]
# tigra1_7 -- read result [127:96]

# First test scenario
# Note: rd values are set to a unique value for key/state insns
# and are not meaningful
#-------------------------------------------------------------

TEST_RR_OP( 2, tigra1_0, 0x0000000A, 0x09cf4f3c, 0xabf71588 );
TEST_RR_OP( 3, tigra1_1, 0x0000000B, 0x28aed2a6, 0x2b7e1516 );
TEST_RR_OP( 4, tigra1_2, 0x0000000C, 0xe0370734, 0x313198a2 );
TEST_RR_OP( 5, tigra1_3, 0x0000000D, 0x885a308d, 0x3243f6a8 );
TEST_RR_OP( 6, tigra1_4, 0x196a0b32, 0x00000001, 0x00000000 );
TEST_RR_OP( 7, tigra1_5, 0xdc118597, 0x00000002, 0x00000001 );
TEST_RR_OP( 8, tigra1_6, 0x02dc09fb, 0x00000008, 0x00000007 );
TEST_RR_OP( 9, tigra1_7, 0x3925841d, 0x22222222, 0x11111111 );

# Second test scenario with different state and key values
#-------------------------------------------------------------
TEST_RR_OP(10, tigra1_0, 0x0000000A, 0x0c0d0e0f, 0x08090a0b );
TEST_RR_OP(11, tigra1_1, 0x0000000B, 0x04050607, 0x00010203 );
TEST_RR_OP(12, tigra1_2, 0x0000000C, 0xccddeeff, 0x8899aabb );
TEST_RR_OP(13, tigra1_3, 0x0000000D, 0x64566777, 0x00011223 );
TEST_RR_OP(14, tigra1_4, 0x70b4c55a, 0x00000001, 0x00000000 );
TEST_RR_OP(15, tigra1_5, 0xd8c2b780, 0x00000002, 0x00000001 );
TEST_RR_OP(16, tigra1_6, 0x6a7b0430, 0x00000008, 0x00000007 );
TEST_RR_OP(17, tigra1_7, 0x69c4e0d8, 0x22222222, 0x11111111 );

TEST_PASSFAIL
RVTEST_CODE_END

.data
RVTEST_DATA_BEGIN
TEST_DATA
RVTEST_DATA_END

Listing 13: The assembly code used to test AES
Listing 14: The commands needed to generate the AES test binary and dump the instructions to a text file

```bash
```

```bash
riscv32-unknown-elf-objdump --disassemble-all --disassemble-zeroes --section=.text --section=.text.startup --section=.text.init --section=.data rv32ui-p-aes > rv32ui-p-aes.dump
```

Listing 15: A portion of the dump file that shows the first four AES instructions that write the state and key values
Listing 16: Four AES instructions that complete reading the AES result shown in the dump file
Listing 17: A portion of the output file that shows how a TIGRA instruction is tested
#include "riscv_test.h"
#include "test_macros.h"

RVTEST_RV64U
RVTEST_CODE_BEGIN

#-------------------------------------------------------------
# Test posit arithmetic
#-------------------------------------------------------------
# rd rs1 rs2
TEST_RR_OP( 2, mul, 0x06260060, 0x00005678, 0x00001234 );
TEST_RR_OP( 3, add, 0x00000007, 0x00000005, 0x00000002 );

# Posit Addition
TEST_RR_OP( 4, tigra0_0, 0x42280000, 0x42100000, 0x40c00000 );

# Continue testing other RISC-V instructions
TEST_RR_OP( 5, mul, 0x00000015, 0x00000003, 0x00000007 );

TEST_PASSFAIL

RVTEST_CODE_END

.data
RVTEST_DATA_BEGIN

TEST_DATA

RVTEST_DATA_END

Listing 18: The assembly code used to test posit addition

riscv32-unknown-elf-gcc --march=rv32g --mabi=ilp32 --static --mmodel=medany --fvisibility=hidden --nostdlib
rv32ui-p-posits

riscv32-unknown-elf-objdump --disassemble-all --disassemble-zeroes --section=.text
--section=.text.startup --section=.text.init --section=.data rv32ui-p-posits >
rv32ui-p-posits.dump

Listing 19: The commands needed to generate the posit addition test binary and dump the instructions to a text file
Listing 20: A portion of the dump file that shows the test cases for posits
Listing 21: A portion of the posits output file that shows how posit addition is tested
object ALU
{
  val SZ_ALU_FN = 4
  def FN_X = BitPat("b????")
  def FN_ADD = UInt(0)
  def FN_SL = UInt(1)
  def FN_SEQ = UInt(2)
  def FN_SNE = UInt(3)
  def FN_XOR = UInt(4)
  def FN_SR = UInt(5)
  def FN_OR = UInt(6)
  def FN_AND = UInt(7)
  def FN_SUB = UInt(10)
  def FN_SRA = UInt(11)
  def FN_SLT = UInt(12)
  def FN_SGE = UInt(13)
  def FN_SLTU = UInt(14)
  def FN_SGEU = UInt(15)
}

class ALU(implicit p: Parameters) extends CoreModule()(p) {
  val io = new Bundle {
    val dw = Bits(INPUT, SZ_DW)
    val fn = Bits(INPUT, SZ_ALU_FN)
    val in2 = UInt(INPUT, xLen)
    val in1 = UInt(INPUT, xLen)
    val out = UInt(OUTPUT, xLen)
    val adder_out = UInt(OUTPUT, xLen)
    val cmp_out = Bool(OUTPUT)
  }
}

Listing 22: The original ALU implemented in Scala showing the size of the instruction encoding and the valid instructions
RVTEST_RV64U
RVTEST_CODE_BEGIN

#-------------------------------------------------------------
# Arithmetic tests
#-------------------------------------------------------------

# Test original ADD instruction
TEST_RR_OP( 10, add, 0x00000000, 0x00000000, 0x00000000 );
TEST_RR_OP( 20, add, 0x00000002, 0x00000001, 0x00000001 );
TEST_RR_OP( 30, add, 0x0000000a, 0x00000003, 0x00000007 );

# Test TIGRA ADD instruction
TEST_RR_OP( 40, tigra0_4, 0x00000000, 0x00000000, 0x00000000 );
TEST_RR_OP( 50, tigra0_4, 0x0000000a, 0x00000003, 0x00000007 );
TEST_RR_OP( 60, tigra0_4, 0x00000002, 0x00000001, 0x00000001 );

# Test original XOR instruction
TEST_RR_OP( 70, xor, 0xf00ff00f, 0xff00ff00, 0x0f0f0f0f );
TEST_RR_OP( 80, xor, 0xff00ff00, 0x0ff00ff0, 0xf0f0f0f0 );
TEST_RR_OP( 90, xor, 0x0ff00ff0, 0x00ff00ff, 0x0f0f0f0f );

# Test TIGRA XOR instruction
TEST_RR_OP( 100, tigra0_4_04, 0xf00ff00f, 0xff00ff00, 0x0f0f0f0f );
TEST_RR_OP( 110, tigra0_4_04, 0xff00ff00, 0x0ff00ff0, 0xf0f0f0f0 );
TEST_RR_OP( 120, tigra0_4_04, 0x0ff00ff0, 0x00ff00ff, 0x0f0f0f0f );

# Test original OR instruction
TEST_RR_OP( 130, or, 0x0f000f00, 0xff00ff00, 0x0f0f0f0f );
TEST_RR_OP( 140, or, 0xff00ff00, 0x0ff00ff0, 0xf0f0f0f0 );
TEST_RR_OP( 150, or, 0x0ff00ff0, 0x00ff00ff, 0x0f0f0f0f );

# Test TIGRA OR instruction
TEST_RR_OP( 160, tigra0_4_06, 0xff00ff0f, 0xff00ff00, 0x0f0f0f0f );
TEST_RR_OP( 170, tigra0_4_06, 0xff00ff00, 0x0ff00ff0, 0xf0f0f0f0 );
TEST_RR_OP( 180, tigra0_4_06, 0x0ff00ff0, 0x00ff00ff, 0x0f0f0f0f );

# Test original AND instruction
TEST_RR_OP( 190, and, 0x0f000f00, 0xff00ff00, 0x0f0f0f0f );
TEST_RR_OP( 200, and, 0x00f000f0, 0xff00ff00, 0xf0f0f0f0 );
TEST_RR_OP( 210, and, 0x000f000f, 0x0ff00ff0, 0x0f0f0f0f );

# Test TIGRA AND instruction
TEST_RR_OP( 220, tigra0_4_07, 0x0f000f00, 0xff00ff00, 0x0f0f0f0f );
TEST_RR_OP( 230, tigra0_4_07, 0x00f000f0, 0xff00ff00, 0x0f0f0f0f );
TEST_RR_OP( 240, tigra0_4_07, 0x000f000f, 0x0ff00ff0, 0x0f0f0f0f );

Listing 23: A portion of the assembly code used to test AluTigra
# Test original SUB instruction
TEST_RR_OP( 250, sub, 0x00000000, 0x00000001, 0x00000001 );
TEST_RR_OP( 260, sub, 0xffffffffc, 0x00000003, 0x00000007 );
TEST_RR_OP( 270, sub, 0x00008000, 0x00000000, 0xffffffff8000 );

# Test TIGRA SUB instruction
TEST_RR_OP( 280, tigra0_4_10, 0x00000000, 0x00000001, 0x00000001 );
TEST_RR_OP( 290, tigra0_4_10, 0xffffffffc, 0x00000003, 0x00000007 );
TEST_RR_OP( 300, tigra0_4_10, 0x00008000, 0x00000000, 0xffffffff8000 );

# Test original SRA instruction
TEST_RR_OP( 310, sra, 0x7fffffff, 0x7fffffff, 0 );
TEST_RR_OP( 320, sra, 0x3fffffff, 0x7fffffff, 1 );
TEST_RR_OP( 330, sra, 0x00ffffff, 0x7fffffff, 7 );

# Test TIGRA SRA instruction
TEST_RR_OP( 340, tigra0_4_11, 0x7fffffff, 0x7fffffff, 0 );
TEST_RR_OP( 350, tigra0_4_11, 0x3fffffff, 0x7fffffff, 1 );
TEST_RR_OP( 360, tigra0_4_11, 0x00ffffff, 0x7fffffff, 7 );

# Test original SLT instruction
TEST_RR_OP( 370, slt, 0, 0x00000001, 0x00000001 );
TEST_RR_OP( 380, slt, 1, 0x00000003, 0x00000007 );
TEST_RR_OP( 390, slt, 0, 0x00000007, 0x00000003 );

# Test TIGRA SLT instruction
TEST_RR_OP( 400, tigra0_4_12, 0, 0x00000001, 0x00000001 );
TEST_RR_OP( 410, tigra0_4_12, 1, 0x00000003, 0x00000007 );
TEST_RR_OP( 420, tigra0_4_12, 0, 0x00000007, 0x00000003 );

TEST_PASSFAIL

RVTEST_CODE_END

.data
RVTEST_DATA_BEGIN
TEST_DATA
RVTEST_DATA_END

Listing 24: The second half of the assembly code used to test AluTigra
Listing 25: The commands needed to generate the *AluTigra* test binary and dump the instructions to a text file

```plaintext
riscv32-unknown-elf-gcc -march=rv32g -mabi=ilp32 -static -mcmodel=medany -fvisibility=hidden -nostdlib
   rv32ui-p-alutigra
riscv32-unknown-elf-objdump --disassemble-all --disassemble-zeroes --section=.text
   --section=.text.startup --section=.text.init --section=.data rv32ui-p-alutigra >
   rv32ui-p-alutigra.dump
```

Listing 26: The partial dump file for testing *AluTigra* ADD instructions with original RISC-V instructions

```
79 800000fc <test_10>:
80 800000fc: 00000093 li ra,0
81 80000100: 00000113 li sp,0
82 80000104: 00208733 add a4,ra,sp
83 80000108: 00000393 li t2,0
84 8000010c: 00a00193 li gp,10
85 80000110: 4e771463 bne a4,t2,800005f8 <fail>
86
87 80000114 <test_20>:
88 80000114: 00100093 li ra,1
89 80000118: 00100113 li sp,1
90 8000011c: 00208733 add a4,ra,sp
91 80000120: 00200393 li t2,2
92 80000124: 01400193 li gp,20
93 80000128: 4c771863 bne a4,t2,800005f8 <fail>
94
95 8000012c <test_30>:
96 8000012c: 00300093 li ra,3
97 80000130: 00700113 li sp,7
98 80000134: 00208733 add a4,ra,sp
99 80000138: 00a00393 li t2,10
100 8000013c: 01e00193 li gp,30
101 80000140: 4a771c63 bne a4,t2,800005f8 <fail>
102
103 80000144 <test_40>:
104 80000144: 00000093 li ra,0
105 80000148: 00000113 li sp,0
106 8000014c: 020c70b tigra0_4 a4,ra,sp
107 80000150: 00000393 li t2,0
108 80000154: 02800193 li gp,40
109 80000158: 4a771063 bne a4,t2,800005f8 <fail>
```

59
Listing 27: The partial dump file for testing AluTigra ADD and the original RISC-V XOR instruction
Listing 28: The partial dump file for testing AluTigra XOR and the original RISC-V OR instruction
Listing 29: The partial dump file for testing AluTigra OR and the original RISC-V OR instruction
Listing 30: The partial dump file for testing AbuTigra OR and the original RISC-V AND instruction
Listing 31: The partial dump file for testing *AluTigra* AND and the original RISC-V SUB instruction
Listing 32: The partial dump file for testing *AluTigra* SUB and the original RISC-V SUB instruction
Listing 33: The partial dump file for testing AluTigra SRA and the original RISC-V SRA instruction
Listing 34: The partial dump file for testing AluTigra SRA and the original RISC-V SLT instruction
Listing 35: The partial dump file for testing *AluTigra* SLT instruction
Appendix B  The Rocket Chip Generator Repository

This Appendix is an abridged version of the instructions for downloading the Rocket Chip Generator repository and developing the Verilog code needed to implement the Rocket core on an FPGA. Official documentation for Rocket can be found on the README page of the Rocket Chip Generator repository¹.

B.1 Check Out Repositories

B.1.1 Rocket Tools

The first step is to create a folder for the working directory and checking out the two git repositories needed to build Rocket. The first repository is a collection of all software tools needed to build Rocket: https://github.com/freechipsproject/rocket-tools. After cloning the repository, the submodules need to be updated.

```
$ git clone https://github.com/freechipsproject/rocket-tools
$ cd rocket-tools
$ git submodule update --init --recursive
```

Before building the rocket-tools repo, the RISCV environment variable needs to be set. This environment variable needs to point to the RISC-V toolchain, which is the rocket-tools directory.

```
$ export RISCV=/path/to/install/riscv/toolchain
```

The MAKEFLAGS variable can be set if there are N cores on the host system.

```
$ export MAKEFLAGS="$MAKEFLAGS -jN"
```

The build.sh script will compile the Rocket tools. The build-rv32ima.sh script must also be run if the RV32 ISA will be used. These two steps will need to be run again if the version is updated and the tools need to be recompiled.

¹https://github.com/chipsalliance/rocket-chip
B.1.2 Dependencies

There may be a few dependencies needed to properly install Rocket tools. These dependencies are listed on the Rocket tools README\(^2\) and are for Ubuntu or Fedora. The following packages are needed for the `yum` package manager:

- libmpc-devel
- mpfr-devel
- gmp-devel
- libusb-devel
- zlib-devel
- pkgconfig
- expat-devel
- flex-devel

B.1.3 Rocket Chip

The second repository contains all the source code for the Rocket Chip Generator:


$ git clone https://github.com/ucb-bar/rocket-chip.git

The Rocket Chip repository also has a number of submodules that need to be updated before the project can be built.

$ cd rocket-chip
$ git submodule update --init

The `$ROCKETCHIP` variable can be set to point to the `rocket-chip` repository. This variable will be used to describe the commands in the next sections.

\(^2\)https://github.com/chipsalliance/rocket-tools/blob/master/README.md
## B.2 Run Emulation

To build the project, navigate to `$ROCKETCHIP/emulator` and make run. The `make` command will generate C++ code for Verilator, a cycle-accurate emulator. It will then compile the emulator and all RISC-V assembly tests and benchmarks. Finally, all tests and benchmarks are run on the emulator. All generated files are in `$ROCKETCHIP/emulator/generated-src`. It is strongly recommended that all these tests pass regardless of what modifications are made to the RISC-V ISA or the Rocket Chip.

```bash
$ cd $ROCKETCHIP/emulator
$ make run
```

## B.3 Generate Waveforms

Waveforms can be generated for each test or benchmark for more in-depth debugging. If Synopsys VCS is installed on the host system, running the command `make run-debug` in `$ROCKETCHIP/emulator` can be used to create `.vpd` files for the waveforms.

```bash
$ cd $ROCKETCHIP/emulator
$ make run-debug
```

If Synopsys VCS is not installed, then the error `vcs: '+lint=all,noVCDE,noDNGS,noUI' is not a vcs command. See 'vcs help'.` will print out on the console and `make run-debug` will stop. Fortunately there is a way to create `.vcd` files that can be used with the free software, GTKWave\(^3\) in order to view waveforms:

\(^3\)http://gtkwave.sourceforge.net/
1. Navigate to `$ROCKETCHIP/src/main/scala/system` and use a text editor to open up the file, `RocketTestSuite.scala`.

   ```bash
   $ cd $ROCKETCHIP/src/main/scala/system
   $ emacs RocketTestSuite.scala  # edit the file
   ```

2. Search inside the file for the text `.vpd` and replace it with `.vcd`. There should have been five changes total.

3. Navigate back to the `$ROCKETCHIP/emulator` folder and rerun `make run-debug` to generate the waveforms. **Note** that a large number of files will be generated and will take up about 50 GB.

   ```bash
   $ cd $ROCKETCHIP/emulator
   $ make run-debug
   ```

### B.4 Generate Verilog Files

Navigate to `$ROCKETCHIP/vsim` and run `make verilog`. This command will generate synthesizable Verilog that can be used with FPGA tools. The files will be saved in `$ROCKETCHIP/vsim/generated-src`.

   ```bash
   $ cd $ROCKETCHIP/vsim
   $ make verilog
   ```

#### B.4.1 Change Configurations

When running the commands to run the emulator or to generate the Verilog files, the default Rocket configuration is used. These configurations are specified in `$ROCKETCHIP/src/main/scala/system/Configs.scala`. The default configuration is referenced in this file as `DefaultConfig` and inherits some of its elements from `BaseConfig`. The user can customize Rocket by changing cache sizes or cache associativity, for example. The modifications can be made by updating this file with the unique configuration. Different configurations can be used in emulation or code generation by using the `CONFIGS` flag. For the purposes of this thesis, the `TinyConfig` configuration was used for simplicity. It is the basis for the `TigraConfig` configuration.
$ cd $ROCKETCHIP/vsim
$ make verilog CONFIG=TinyConfig # to generate Verilog files
$ cd $ROCKETCHIP/emulator
$ make run-debug CONFIG=TinyConfig # to generate waveforms for debugging
Appendix C  Create a Vivado Project

There are a number of FPGA tools that can be used to import the Verilog files that were
generated by the Rocket Chip Generator. The tool used for this thesis was Xilinx’s Vivado Design
Suite. The steps in this Appendix detail how to import the Verilog files, fix any small issues, and
successfully elaborate the design.

C.1 New Project Wizard

1. Open up Vivado and click on Create Project under the Quick Start menu. Refer to Figure
1.

   ![Create Project](image)

   Figure 1: Select Create Project

2. Click Next on the next screen to progress through the New Project Wizard.

3. Give the project a meaningful name in the Project name box and select its location. Check
the box, Create project subdirectory so that all the files will remain in one folder. Click
Next. Refer to Figure 2.
4. Select **RTL Project** and uncheck the box, **Do not specify sources at this time**. Click **Next**. Refer to Figure 3.

Figure 2: Project name and location details

Figure 3: Project type
5. Begin adding files to the project by clicking on Add Files. The files needed are in $ROCKETCHIP/vsim/generated-src. The files will be prefixed with the configuration specified when they were generated. For example, for the DefaultSmallConfig files are:

- freechips.rocketchip.system.DefaultSmallConfig.behav_srams.v
- freechips.rocketchip.system.DefaultSmallConfig.v

There is also a folder that was generated that is prefixed with the configuration. Open that folder and include all those auxiliary files.

6. Review the files added. There is a checkbox named, Copy sources into project. Checking this box will make a local copy of these files in the project directory. Unchecking this box will make this project be a bit more flexible if any changes are made either in Chisel or Verilog. If changes will be made frequently, it is recommended that this box remain unchecked to avoid having to manually update the files for every change. Once all the files look correct, click Next. Refer to Figure 4.

![Add Sources](image_url)

Figure 4: Add files to the project

76
7. If constraints are needed, add the files on the next page. Otherwise, click Next.

8. A Xilinx part or board will need to be added to the project. This will vary depending on what is available. For this thesis, the xcvu9p-flgb2104-2-i was selected as this is the FPGA that is used for AWS. Search for the part number and select the correct part. Click Next.

9. The Summary page will appear. Review everything to make sure the project is in order. If not, navigate back to a step by clicking the Back button. Click Finish to finish creating the project.

C.2 Troubleshooting

C.2.1 Syntax Error Files

After initially creating the project, there will be a file in the Syntax Error Files folder in the Sources window. The elaborated design will not open until this file has been fixed.

1. Click the arrow to the left of Syntax Error Files to see which file is causing the issue. It will likely be SimDTM.v. Note that if there are more files in this folder, the errors will need to be manually indentified and corrected. Refer to Figure 5.

2. To correct SimDTM.v, click on the file once. The window below the Source window, Source

![Figure 5: Source files for ExampleProj](image)

2. To correct SimDTM.v, click on the file once. The window below the Source window, Source
File Properties, will change to reflect the file properties. Click the triple dots by Type to change from Verilog to SystemVerilog. Refer to Figure 6.

![Source File Properties](image)

Figure 6: Update to SystemVerilog for SimDTM.v

3. The sources will automatically update after the file properties are changed. There will no longer be a Syntax Error Files folder once all of the erroneous files have been fixed.

### C.2.2 Open Elaborated Design Errors

Most of the files needed to create a project should have been generated or copied over during the Generate Verilog Files step in Appendix B.4. One file that could have been missed is the EICG_Wrapper.v and will become apparent during RTL Analysis. An error may look like the one in Figure 7. This file is found in $ROCKETCHIP/src/main/resources/vsrc. Once this file is added to the project, RTL Analysis should work successfully.
Figure 7: Error caused by missing EICG.Wrapper.v file
Appendix D  Adding New Instructions to the RISC-V ISA

This appendix is a modified version of the instructions on how to add custom instructions to the RISC-V ISA found in Nitish Srivastava’s blog post. 4

D.1 Update the ISA

If rocket-tools has not already been downloaded, follow the steps in Appendix B.1.1 to build the toolchain. For TIGRA instructions, we will mimic already existing R-type instructions, such as an ADD instruction.

1. Open up rocket-tools/riscv-opcodes/opcodes. This file contains the list of instructions, their opcodes and instruction bits. At the end of the opcodes file, add the TIGRA instructions.

```
# Custom instructions for TIGRA
#  funct7  funct3  custom-0
tigra0_0  rd  rs1  rs2  31..25=0  14..12=0  6..2=0x02  1..0=3
tigra0_1  rd  rs1  rs2  31..25=0  14..12=1  6..2=0x02  1..0=3
....
#  funct7  funct3  custom-1
  tigra1_0  rd  rs1  rs2  31..25=0  14..12=0  6..2=0xA  1..0=3
  tigra1_1  rd  rs1  rs2  31..25=0  14..12=1  6..2=0xA  1..0=3
  tigra1_2  rd  rs1  rs2  31..25=0  14..12=2  6..2=0xA  1..0=3
  ....
  tigra0_4_01  rd  rs1  rs2  31..25=0x01  14..12=4  6..2=0x02  1..0=3
  tigra0_4_02  rd  rs1  rs2  31..25=0x02  14..12=4  6..2=0x02  1..0=3
```

Bits 1-0 are high to indicate that this is an instruction. We can only use either the custom-0 or custom-1 bit as part of the major opcode. For custom-0 we set bits 6 to 2 to 0x02 and for custom-1 we set bits 6 to 2 to 0xA. Refer to Figure 2.2 for more detail on major opcodes. The funct3 field is three bits wide and is used as a minor opcode. We can add eight instructions using just the funct3 fields for each major opcode. If needed, we can add additional instructions using the other minor opcode field, funct7. The TIGRA instructions are labeled as tigra<custom-#>_<funct3 value>_<funct7 value>.

---

4https://nitish2112.github.io/post/adding-instruction-riscv/
2. Run the following command to generate a new file with the new instructions.

```
$ cat opcodes-pseudo opcodes opcodes-rvc opcodes-rvc-pseudo opcodes-custom | ./parse-opcodes -c > ~/temp.h
```

3. Open up ~/temp.h and there will be a `#define MATCH` and a `#define MASK` for every TIGRA instruction. We see that the MASK constants are the same for all TIGRA instructions therefore we make one general `#define MASK_TIGRA`. Copy and paste these defines into riscv-opc.h in rocket-tools/riscv-gnu-toolchain/riscv-binutils-gdb/include(opcode/).

```
#define MATCH_TIGRA0_0 0xb
#define MASK_TIGRA 0xfe00707f
#define MATCH_TIGRA0_1 0x100b
#define MATCH_TIGRA0_2 0x200b
....
```

4. We also see in ~/temp.h a DECLARE_INSN macro for every TIGRA instruction. Update these instructions to use the new MASK and copy and paste these lines into riscv-opc.h in rocket-tools/riscv-gnu-toolchain/riscv-binutils-gdb/include(opcode/).

```
DECLARE_INSN(tigra0_0, MATCH_TIGRA0_0, MASK_TIGRA)
DECLARE_INSN(tigra0_1, MATCH_TIGRA0_1, MASK_TIGRA)
DECLARE_INSN(tigra0_2, MATCH_TIGRA0_2, MASK_TIGRA)
....
```

5. Update rocket-tools/riscv-gnu-toolchain/riscv-binutils-gdb/opcodes/riscv-opc.c to include the new instructions. Add the following line for every TIGRA instruction in the assignment of `const struct riscv_opcode riscv_opcodes[]`.

```
const struct riscv_opcode riscv_opcodes[] =
{
    /* name, isa, operands, match, mask, match_func, pinfo. */
    {"unimp", "C", "", 0, 0xffffU, match_opcode, 0 },
    ....
    {"wfi", "I", "", MATCH_WFI, MASK_WFI, match_opcode, 0 },
    ....
};
```
The TIGRA instructions will belong to the "I" or integer ISA. The operands are "d, s, t" referring to the destination and two source registers that it will use.

6. Recompile `rocket-tools` and the TIGRA instructions will be part of the new ISA.
7. To test if the assembler can use the new TIGRA instruction, write a short program in C and name it testIns.c.

```c
#include <stdio.h>
int main()
{
    /* code */
    int a, b, c, d, z;
    a = 0x09cf4f3c; // key lo
    b = 0xabf71588; // key midlo
    c = 0x28aed2a6; // key midhi
    d = 0x2b7e1516; // key hi

    asm volatile
    ( "tigra0_0 %[z], %[x], %[y]\n" : [z] "=r" (z)
         : [x] "r" (a), [y] "r" (b)
    );

    asm volatile
    ( "tigra0_1 %[z], %[w], %[v]\n" : [z] "=r" (z)
         : [w] "r" (c), [v] "r" (d)
    );

    return 0;
}
```
8. Compile it using the modified RISC-V compiler.

```bash
$ riscv64-unknown-elf-gcc testIns.c -o testIns
```

9. Do an `objdump` on the binary and open up `testIns.dump` to see if the instructions are actually being used.

```bash
$ riscv64-unknown-elf-objdump -dC testIns > testIns.dump
```

### D.2 Add the new instruction to the Spike ISA Simulator

Spike is a RISC-V ISA simulator and implements a functional model of one more RISC-V harts. The files that will be modified are similar to what was done previously when updating the toolchain.

1. Open `rocket-tools/riscv-isa-sim/riscv/encoding.h` and add the lines that came from `/temp.h`.

   ```c
   #define MATCH_TIGRA0_0 0xb
   #define MASK_TIGRA 0xfe00707f
   #define MATCH_TIGRA0_1 0x100b
   #define MATCH_TIGRA0_2 0x200b
   ....
   DECLARE_INSN(tigra0_0, MATCH_TIGRA0_0, MASK_TIGRA)
   DECLARE_INSN(tigra0_1, MATCH_TIGRA0_1, MASK_TIGRA)
   DECLARE_INSN(tigra0_2, MATCH_TIGRA0_2, MASK_TIGRA)
   ....
   ```

2. Create a header file in `rocket-tools/riscv-isa-sim/riscv/insns/` for every TIGRA instruction. The file should be named `<new_instruction_name>.h`. Leave the file blank so that the instruction is generic for any application. The importance of this step is that a header file exists for every TIGRA instruction.

3. Add the header files to `rocket-tools/riscv-isa-sim/riscv/riscv.mk.in`

---

riscv_insn_list = \ 
  ...
  tigra0_0 \ 
  tigra0_1 \ 
  ...

4. In rocket-tools/riscv-isa-sim/spike_main/disasm.cc, define the TIGRA instructions as R-type instructions.

  ...
  DEFINE_RTYPE(tigra0_0);
  DEFINE_RTYPE(tigra0_1);
  ...

5. Rebuild rocket-tools and TIGRA instructions will be added to Spike.
Bibliography


