Clemson University

TigerPrints

All Dissertations | Dissertations

5-2022

The Development of TIGRA: A Zero Latency Interface For Accelerator Communication in RISC-V Processors

Wesley Brad Green
wbgreen@g.clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations

Part of the Computer and Systems Architecture Commons

Recommended Citation

Green, Wesley Brad, "The Development of TIGRA: A Zero Latency Interface For Accelerator Communication in RISC-V Processors" (2022). *All Dissertations*. 2982.
https://tigerprints.clemson.edu/all_dissertations/2982

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

The Development of TIGRA: A Zero Latency Interface for Accelerator Communication in RISC-V Processors

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Engineering

---

by
Wesley Brad Green
May 2022

Accepted by:
Dr. Melissa C. Smith, Committee Chair
Dr. Jon C. Calhoun
Dr. Walt Ligon
Dr. Rong Ge

Abstract

Field programmable gate arrays (FPGA) give developers the ability to design application specific hardware by means of software, providing a method of accelerating algorithms with higher power efficiency when compared to CPU or GPU accelerated applications. FPGA accelerated applications tend to follow either a loosely coupled or tightly coupled design. Loosely coupled designs often use OpenCL to utilize the FPGA as an accelerator much like a GPU, which provides a simplifed design flow with the trade-off of increased overhead and latency due to bus communication. Tightly coupled designs modify an existing CPU to introduce instruction set extensions to provide a minimal latency accelerator at the cost of higher programming effort to include the custom design.

This dissertation details the design of the Tightly Integrated, Generic RISC-V Accelerator (TIGRA) interface which provides the benefits of both loosely and tightly coupled accelerator designs. TIGRA enabled designs incur zero latency with a simple-to-use interface that reduces programming effort when implementing custom logic within a processor. This dissertation shows the incorporation of TIGRA into the simple PicoRV32 processor, the highly customizable Rocket Chip generator, and the FPGA optimized Taiga processor. Each processor design is tested with AES 128-bit encryption and posit arithmetic to demonstrate TIGRA functionality.

After a one time programming cost to incorporate a TIGRA interface into an existing processor, new functional units can be added with up to a 75% reduction in the lines of code required when compared to non-TIGRA enabled designs. Additionally, each functional unit created is co-compatible with each processor as the TIGRA interface remains constant between each design. The results prove that using the TIGRA interface introduces no latency and is capable of incorporating existing custom logic designs without modification for all three processors tested. When compared to the PicoRV32 coprocessor interface (PCPI), TIGRA coupled designs complete one clock cycle faster. Similarly, TIGRA outperforms the Rocket Chip custom coprocessor (RoCC) interface by

an average of 6.875 clock cycles per instruction. The Taiga processor's decoupled execution units allow for instructions to execute concurrently and uses a tag management system that is similar to out-of-order processors. The inclusion of the TIGRA interface within this processor abstracts the tag management from the user and demonstrates that the TIGRA interface can be applied to out-of-order processors.

When coupled with partial reconfiguration, the flexibility and modularity of TIGRA drastically increases. By creating a reprogrammable region for the custom logic connected via TIGRA, users can swap out the connected design at runtime to customize the processor for a given application. Further, partial reconfiguration allows users to only compile the custom logic design as opposed to the entire CPU, resulting in an 18.1% average reduction of compilation during the design process in the case studies. Paired with the programming effort saved by using TIGRA, partial reconfiguration improves the "time to design" and "test new functionality" timelines for a processor.

Dedication

I dedicate this Dissertation to my grandfather, Frank Minozzi. The man who taught me the meaning of hard work while never forgetting to have a little bit of fun.

Acknowledgments

I would first like to acknowledge Dr. Melissa Smith, who has pushed and supported me for too many long years. Without her guidance, mentorship, and friendship I would not have pushed through and completed my PhD.

I want to thank my family and friends who have helped me through some of the toughest times of my life. They all stood by me at the lowest points and helped to lift me up when it was needed the most.

I want to thank my partner, Caitie B. O'Donnell, who helped kick my butt into gear, motivate me, and support me as I pushed toward the completion of this work.

I would specifically like to thank (and congratulate) Dillon Todd and Theresa Lê, who's Master's Theses helped bring the TIGRA concept to life!

Table of Contents

Title Page i

Abstract ii

Dedication iv

Acknowledgments v

List of Tables viii

List of Figures ix

List of Listings xii

1 Introduction 1

2 Related Work 3
- 2.1 AES 3
- 2.2 Posits 3
- 2.3 Loosely Coupled Accelerators 4
- 2.4 Tightly Coupled Accelerators 5
- 2.5 Partial Reconfiguration 5
- 2.6 Summary 6

3 Research Design and Methods 7
- 3.1 TIGRA Design 7
- 3.2 Custom Logic Design and Usage 13
- 3.3 Summary 15

4 Case Studies 17
- 4.1 PicoRV32 17
- 4.2 Rocket Chip Generator 21
- 4.3 Taiga Processor 24
- 4.4 AES 128-bit Custom Logic 27
- 4.5 Posit Arithmetic Custom Logic 28
- 4.6 Built-in Functions Custom Logic 30
- 4.7 Comparing the Programming Effort 32
- 4.8 Summary 33

5 Simulation Results 35
- 5.1 PicoRV32 Simulation Results 35
- 5.2 PicoRV32 Hardware Results 39

5.3 Rocket Chip Results . . . 41
5.4 Taiga Processor Results . . . 46
5.5 Summary . . . 51

6 Partial Reconfiguration . . . 53
6.1 Taiga Partial Reconfiguration . . . 54
6.2 Summary . . . 57

7 Conclusions and Future Work . . . 58

Appendices . . . 61
A Code Listings . . . 62
B Extended Timing Diagrams . . . 76
C Raw Data from Simulations . . . 78

Bibliography . . . 81

List of Tables

4.1 Custom instructions for AES 28
4.2 Custom instructions for arbitrary Taiga design 32
4.3 Comparison of the lines of code edited for each design in Chapter 4. 32

5.1 Latency of TIGRA enabled Rocket compared to RoCC when executing AES instructions. Each value counts the number of clock cycles between instruction fetch and when the result is stored in a register. 47

6.1 Synthesis and Implementation timing results of the full Taiga processor with each test case connected via TIGRA and the same designs using Xilinx DFX. 55
6.2 Comparison of synthesis and implementation times of a complete design compared to the same design using Xilinx DFX. 56

List of Figures

3.1 The main RISC-V instruction type opcodes: Register, Immediate, Upper immediate, and Store. 8
3.2 A processor coupled with custom logic via the TIGRA interface. The signals connecting the Custom Logic to the processor block (shown in orange) are required to handle communication with the user designed hardware. The multiplexer and the signal connecting it to the decoder (shown in white) represent the minimal logic added to the processor to ensure TIGRA instructions follow the standard ALU flow. Clock and reset signals are not shown. 10
3.3 An example four-stage CPU pipeline, showing how to set each TIGRA control signal in the host processor and the custom hardware design. 11
3.4 A flowchart detailing the generalized process to setup a TIGRA interface on an existing processor. The upper orange highlighted area represents the process required when adding any new execution unit to an existing CPU, and the lower purple area represents the unique effort required to generalize the new execution unit for TIGRA. 12
3.5 An example custom logic setup implementing many accelerators. The signals external to the custom logic block represent the TIGRA interface. The necessary instruction decoder(s) and other logic is not shown to maintain generality and keep the design readable. 14

4.1 The PicoRV32 state diagram, emphasising the PCPI and TIGRA flows. The boxes represent the value and timing of TIGRA control signals during execution. 19
4.2 The AWS F1 HDK hierarchy when connected with a full PicoRV32 with TIGRA enabled and connected to custom logic. 20
4.3 Example execution stall of a TIGRA instruction in the Rocket pipeline. The purple highlighted instruction (TIGRA) stalls in the execute stage of the pipeline, causing later instructions (Insn4 and Insn5 highlighted in orange) to stall in decode and fetch. 23
4.4 Execution of a RoCC instruction in the Rocket Chip pipeline. As shown, custom hardware connected to a RoCC interface begins execution as the instruction exits the main Rocket Chip pipeline. 25
4.5 Taiga's general pipeline flow showing the necessary additions to include a TIGRA interface. The added unit is shown separate from the Taiga execution units to represent that each operates independently of each other. Each vertical stage is separated by a queue to facilitate the decoupled operation and ensure instructions complete in order. 26
4.6 AES-128 Bit custom logic block diagram. 29
4.7 Posit arithmetic custom logic block diagrams for use with PicoRV32 or hardware compilation (a) and for use with the Rocket Chip generator and Taiga simulators (b). 30
4.8 Rocket ALU custom logic block diagram. 31

5.1 PicoRV32 with TIGRA and AES: The second two TIGRA instructions which store the AES state in the custom logic and complete the encryption. The break represents 15 clock cycles during which none of the shown signals change. 36

5.2 PicoRV32 with TIGRA and AES: The first two reads from the AES custom logic, showing the end of the stall from the last write. 37
5.3 PicoRV32 with TIGRA and Posit Arithmetic: Posit addition showing the data flow from the custom logic all the way through register write-back. 37
5.4 PicoRV32 with TIGRA and Posit Arithmetic: Posit multiplication showing the data flow from the custom logic all the way through register write-back. 38
5.5 PicoRV32 with TIGRA and Posit Arithmetic: Posit division showing the data flow from the custom logic all the way through register write-back. The break represents 6 clock cycles during which none of the shown signals change. 38
5.6 PicoRV32 with TIGRA and multiplication: Multiplication, the break represents 30 clock cycles. 40
5.7 PicoRV32 with PCPI and multiplication: Multiplication, the break represents 31 clock cycles. 40
5.8 Rocket Chip with AES via TIGRA: The first three AES writes. 42
5.9 Rocket Chip with TIGRA: The remaining instructions to complete AES encryption in a Rocket Chip generated processor. The break in the diagram represents 18 clock cycles during which none of the signals shown change. 43
5.10 Rocket Chip with TIGRA: Posit addition of 18 and 3, $0x42100000$ and $0x40C00000$, providing the resultant value 21, $0x42280000$. 44
5.11 Rocket Chip with TIGRA and Rocket Chip ALU. Results from two XOR instructions executing within the instruction verification environment provided by the RISC-V Tools. 45
5.12 Rocket Chip with RoCC and AES. The first AES write instruction that writes the lowest 32-bits of the AES key to the custom logic. The break represents 7 clock cycles during which none of the shown signals change. 45
5.13 Rocket Chip with RoCC and AES. The first two AES read instructions that return the lowest 64-bits of the encrypted result. 46
5.14 Simulation of the first two TIGRA instructions when using custom logic with AES. These instructions both complete within one clock cycle. 48
5.15 Simulation showing the latter five TIGRA instructions executing on the Taiga processor with AES hardware connected. The break represents 18 clock cycles during which none of the shown signals change. 49
5.16 Simulation of two consecutive *tigra_0* instructions with Posit addition connected via the TIGRA interface on the Taiga processor. Each break represents 2 clock cycles during which none of the shown signals change. 50
5.17 Simulation of the TIGRA instructions 4-7 when using custom logic with arbitrary return values and stalls. These instructions signal complete at varying times even though the data is immediately available. 50
5.18 Simulation of the first 7 multiplications to test the multiplication custom logic. . . . 51

6.1 Abstract block diagram for partial reconfiguration. The Custom Logic block is the reconfigurable region in orange. The PR logic block, shown in white, represents the extra logic required to partially reprogram the FPGA. 53
6.2 Screenshot showing a portion of the XCVU9P device, including the custom logic block which is designated as the partially reconfigurable region. 55

1 Simulation of the first two TIGRA instructions in PicoRV32 with AES via TIGRA. These instructions both complete within one clock cycle. 76
2 Simulation of the last two TIGRA instructions in PicoRV32 with AES via TIGRA. These instructions all complete within one clock cycle. 76

3 Simulation of TIGRA instructions 2 and 3 when using custom logic with AES. Instruction 2 completes within one clock cycle, while instruction 3 initiates a stall and completes the encryption. 77
4 Simulation of the first four TIGRA instructions when using custom logic with arbitrary return values and stalls. These instructions all complete within one clock cycle. 77
5 Original timing diagram showing six of the eight instructions for custom logic with arbitrary return values and stall lengths. This diagram shows the difficulty of reading content in this format compared to the transcribed waveforms shown in the paper. . 79
6 Original timing diagram showing the first 7 TIGRA multiplication instructions. This diagram shows the compiler reordering commands for the code in Listing 7. Here you can see the processor issue 5 TIGRA instructions consecutively, while the code issues at most 3 consecutively. 80

Listings

1 The code required to add a TIGRA enabled PicoRV32 processor to an AWS F1 wrapper with posit custom logic. 62
2 The instructions used to test the Posit custom logic within the PicoRV32 with TIGRA. 66
3 The instructions used to test the AES custom logic within the PicoRV32 with TIGRA. 67
4 The instructions used to test multiplication within the PicoRV32 with TIGRA. . . . 68
5 The instructions used to test multiplication within the PicoRV32 using PCPI. 68
6 C code used to test AES encryption in the Rocket Chip and Taiga processors via the TIGRA interface. 69
7 C code used to test multiplication in the Taiga processor via the TIGRA interface. . 70
8 C code used to test the arbitrary custom logic in the Taiga processor via the TIGRA interface. 72
9 Module declaration of the TIGRA custom logic in the Taiga processor using a System Verilog interface and struct to simplify design. 75
10 Flattened module declaration of the TIGRA custom logic for use with Dynamic Function eXchange. 75
11 Binary object dump of AES test code for the Rocket Chip processor. This shows the instructions required to complete the code in Listing 6 and demonstrates the instruction reordering completed by the compiler. 78

Chapter 1

Introduction

Field-programmable gate arrays (FPGAs) give developers the ability to design application specific hardware by means of software. This enables developers to create custom implementations of existing algorithms that execute with very little latency and with higher power efficiency than the algorithm on CPUs or even GPUs, as shown in [35], [11] and [29]. As a result of this capability, FPGAs uniquely fill a need to combat growing power requirements in high performance computing (HPC) environments [55]. Developers are also designing application specific accelerators, such as Google's TPU [30] and NVIDIA's tensor cores [39] for machine learning, in an attempt to ensure HPC applications continue to scale over time. FPGAs are often used to prototype novel designs before building an application specific integrated circuit (ASIC) such as in [18] where the authors build an FPGA-based accelerator to process data captured by the Square Kilometre Array (SKA) telescope with the intention of taping out to an ASIC. The authors in [36] create a real-time power, temperature and aging monitor system (eTAPMon) used to help predict the ASIC characteristics from FPGA-based prototypes of multi-processor systems-on-chips (MPSoCs) designs.

With the advent of OpenCL for FPGAs [21][53], developers are given a set of tools to simplify FPGA programming that enable acceleration of existing algorithms. OpenCL enables the host computer to use an FPGA as a loosely coupled accelerator, much like when using a general purpose graphics processing unit (GPGPU) for application development. As a loosely coupled accelerator, the device communicates to the CPU over a bus which introduces latency. Further, loosely coupled devices tend to operate on very coarse-grained chunks of an existing application in an attempt to minimize the communication time required and latency introduced.

To minimize the latency introduced for applications that require fine-grained computation or to introduce improvements and extensions to an existing architecture, many developers use FPGAs to create tightly coupled accelerators [33][3][42]. These tightly coupled devices require the developer modify the existing CPU pipeline to incorporate the custom logic in the FPGA into design, significantly increasing the programming effort required. Existing within the CPU pipeline, these designs incur very little or no latency penalty and can allow very fine-grained execution. However, without fabricating the corresponding hardware from a tightly coupled design, these architectural improvements are only available for use through simulation or a soft-core CPU running within an FPGA.

Due to the restrictions on tightly coupled accelerators, FPGAs are typically only used for algorithm acceleration within the loosely coupled OpenCL model. While developers may design custom hardware that can complete a design with very low latency, most existing mechanisms do not leverage FPGA custom logic co-processors in real world applications without synthesizing and running the entire processor and custom design on an FPGA. This drawback, coupled with the release of the Intel Xeon Scalable processors [43], motivate the research described in this work.

This dissertation introduces TIGRA [13], a tightly integrated generic RISC-V accelerator interface that provides the benefits of a tightly coupled accelerator without requiring modification of the underlying CPU architecture. With TIGRA, the researcher aims to provide an architecture modification that will allow developers to fully leverage FPGA custom designs on a CPU without the overhead of communication via a standard bus. TIGRA reduces the required programming effort to incorporate fine-grained, tightly coupled accelerators to existing architectures and has implications for many common HPC algorithms, such as encryption or compression. The interface also provides developers with a mechanism for quickly testing proposed extensions to an architecture, such as built in posit arithmetic or improved hardware for existing functionality.

The remainder of this dissertation is organized as follows: Chapter 2 discusses existing work for different FPGA based accelerators, Chapter 3 provides the design of TIGRA and experiments proving the functionality, Chapter 4 explains the design of each case study used to prove TIGRA, Chapter 5 discusses the results and implications of the use of TIGRA, Chapter 6 shows the benefits of using partial reconfiguration, and Chapter 7 concludes the paper with a discussion of future work and summarized results.

Chapter 2

Related Work

In this section, we discuss the test cases used in Chapter 4 and the motivation behind these choices. We will also describe many different loosely and tightly coupled FPGA accelerators and the improvements they provide over existing architectures and pure software implementations.

2.1 AES

With the rising usage of cloud computing and data transfers via the network, data security concerns have grown, pushing the need for fast encryption in datacenters. Rijndael, the Advanced Encryption Standard (AES) algorithm, is one of the most widely used algorithms for data security in high performance computing [12]. As FPGAs have become more widely used, developers have been targeting these reprogrammable devices in an attempt to include ultra low latency encryption hardware in existing designs. In early 2004, the authors of [15] achieve a 21.54 Gb/s AES design with a clock frequency of only 168.3 Mhz. A more recent design achieves a throughput of 54.52 Gb/s on FPGA hardware using AES-192 [2]. Both of these designs were executed on FPGAs without a host computer, restricting their usage in existing applications or within an HPC environment.

2.2 Posits

Posit numbers are a rising data format intended to operate as a drop in replacement for the floating point number system [14]. Posits are the hardware friendly evolution of Type II Unums,

a number format designed to improve upon the floating point system. This newer number system has proven to provide a higher dynamic range, with higher accuracy at a smaller data size than the existing floating point standard. Developers have begun testing posits in deep learning applications, which rely on floating point calculations to achieve high accuracy inference results. The authors in [10] show that an 8-bit posit implementation of Deep Neural Networks achieve the same classification results as the same network running on 32-bit floating point architectures. Further, [37] demonstrates a 51% reduction in power with the same comparison, showing the capabilities of posits. In [26], [25], and [27], the authors create a posit arithmetic generator for FPGAs that is used in Section 4.5 to enable Posit arithmetic via the TIGRA interface for each tested processor.

2.3 Loosely Coupled Accelerators

Loosely coupled accelerators, devices that communicate over a bus to a host computer like a GPU, are very commonly used in HPC applications. OpenCL for FPGAs [21][53] enables the use of application specific hardware as a loosely coupled accelerator, and many developers have begun creating designs to leverage usage of these lower power devices for algorithm acceleration. [46] introduces an OpenCL based AES accelerator that achieves a throughput of 392.173 MB/s. Many other designs leverage the Rocket Custom Coprocessor (ROCC) interface from the Rocket Chip generator [7] to enable loosely coupled accelerators within the RISC-V ISA. The RoCC interface allows developers to more easily include custom logic within a Rocket Chip processor, and the interface also provides access to the floating point unit (FPU) and memory within the processor.

In [56], the authors create a low power YOLO network accelerator via RoCC that achieves 6.36 GOps/W. [9] introduces a DSP accelerator using the RoCC interface that achieves up to 100 times speedup over existing software based algorithms. The authors in [20] create "Centrifuge" to leverage high-level synthesis (HLS) and Fire Sim [32] and enable developers to create the interface for RoCC accelerators without the need to modify RTL directly. And the authors in [47] create *Picos*, a hardware based task-scheduler, and *Phentos*, a full Rocket Chip ecosystem utilizing *Picos*, resulting in a 308 times improvement compared to base designs without the hardware task-scheduler. While these results are promising, loosely coupled accelerators introduce overhead due to the communication latency incurred. [41] and the results in Section 5.3.4 explore the latency incurred when using the RoCC interface.

2.4 Tightly Coupled Accelerators

Tightly coupled accelerators aim to remove the communication overhead incurred from loosely coupled designs, but often require much more programming effort to implement the accelerator. The developers of [24] create a tightly coupled AES system that achieves a 35% performance increase when compared to a loosely coupled implementation using an Avalon interface. In [3], a tightly coupled H.265/HEVC deblocking filter achieves an 11% performance increase when compared to the standard software implementation. Tightly coupled accelerators are also often used to implement extensions to an existing instruction set. The authors of [42] propose an out-of-order floating point extension and [33] introduces ten more bit-manipulation instructions to the RISC-V ISA. In [38], the authors propose a posit extension to RISC-V. In the described work, the authors replace the existing FPU in the Rocket Chip generator with custom logic for posit arithmetic and explain the difficulty in designing custom instructions and adding to an existing Rocket Chip processor. The authors in [31] develop the RISC-V Architecture Extension for the Number Theoretic Transform (RANTT) which incorporates a task scheduler and does not require the use of custom instructions. The RANTT design achieves up to 6x performance over a pure software based design, but pure hardware accelerator implementations out-perform this implementation.

2.5 Partial Reconfiguration

As development tools mature, more researchers have begun to utilize partial reconfiguration (PR) in designs to allow for increased functionality of designs and reduce the area and power requirements when working with FPGAs. The authors of [16] develop a software defined radio (SDR) design that allows users to swap networking protocols within the FPGA at runtime to achieve a reduction of 76.71% power usage compared to a design that implements all required protocols when not using partial reconfiguration. The authors improve upon their own design by optimizing partitioning in [17] to further reduce power and area usage of five different communication protocols. In [1], the authors evaluate using partial reconfiguration within the RoCC interface by implementing the AES and DES encryption algorithms to achieve a speedup of up to 249.91 times when compared to the software implementation. This design enables the user to choose which encryption is required at runtime. PR is also used in video shot boundary detection to allow users to swap between different algorithms in [48], but the authors note the difficulty in improving the partial reconfiguration design

due to limitations within the FPGA development software.

2.6 Summary

The works described here all provide improvements upon existing software based designs. The loosely coupled designs provide ease of programmability through HLS tools but incur a latency penalty due to bus communication. Conversely, tightly coupled designs incur no latency overhead and provide fine-grained control of a design but require much more effort to modify an existing processor. The TIGRA interface provides the ease of programmability given by loosely coupled designs while incurring no latency penalty similar to a tightly coupled design. Further, leveraging partial reconfigurability with the TIGRA interface improves the portability and modularity of custom logic in designs utilizing FPGAs for the custom logic.

Chapter 3

Research Design and Methods

In this chapter, we discuss the proposed design and usage of the Tightly Integrated Generic RISC-V Accelerator (TIGRA) interface for a generic processor, how to integrate TIGRA into an existing design, the forward-looking goal for TIGRA, and how to use the interface in a development environment.

3.1 TIGRA Design

TIGRA is a tightly integrated, generic RISC-V accelerator interface that aims to bridge the gap between loosely and tightly coupled accelerators. This interface provides a simple interconnect between FPGA custom logic and a CPU pipeline that reduces the programming effort required to incorporate custom hardware with existing processors with no latency. The initial addition of a TIGRA interface incurs a one time programming cost, but reduces the programming effort to add any future custom logic to the processor. The intended design allows for the connected custom logic to exist separately from the CPU, but receive the benefits of an integrated hardware element or instruction set modification.

In this section we describe the design choices made for TIGRA, the minimal signal requirements for the interface, and design guidelines for developing and using TIGRA. Here, we target developers aiming to incorporate a TIGRA interface into RISC-V processors while maintaining simplicity, speed, and flexibility for hardware designs.

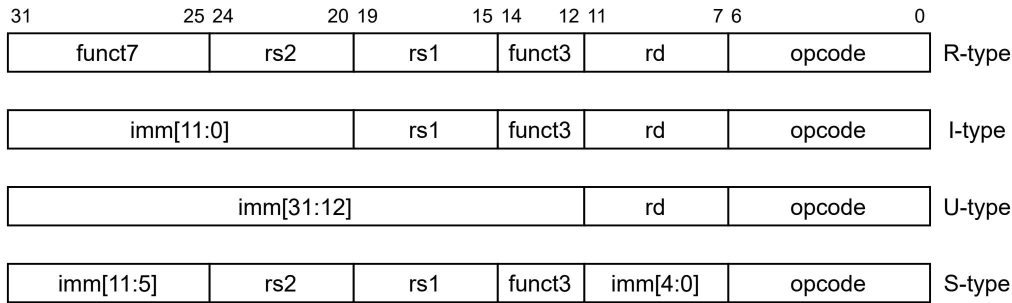


Figure 3.1: The main RISC-V instruction type opcodes: Register, Immediate, Upper immediate, and Store.

3.1.1 RISC-V ISA

RISC-V is an open-source ISA developed in an effort to provide a standard that drives innovation and enables easy customization and design of different processors [23]. The authors at RISC-V International maintain a database of many available designs for developers to download and test new extensions or designs. The availability and customizability of RISC-V CPUs motivates the choice to model TIGRA for use within this architecture, and the different instruction designs help motivate the interface design. Further, while TIGRA is motivated by RISC-V, the concepts presented are applicable to any existing architecture.

RISC-V is designed with 4 main instruction types, shown in Figure 3.1[50]. As shown, R-type instructions require two source registers and write a result to a destination register. I-type instructions are used for instructions that require immediate input values, such as add immediate or load word instructions, and use one source register and a 12-bit immediate value. As I-type instructions only allow for 12-bit values and most data-types are at minimum 32-bits wide, the RISC-V authors needed to create the U-type instructions. These instructions load the upper 20 bits of a register from an immediate value within the instruction. Used together, I and U-type instructions load a register with a 32-bit value. S-type instructions are used for stores, using one of the source registers as the base memory address and the other as the data to be written to memory. The immediate fields of the S-type instructions are used together as a memory offset. The *opcode* field of each type shown specifies the major opcode of the instruction. The $funct3$ and $funct7$ fields, when present, represent 3 and 7-bit minor opcodes, respectively, to further distinguish between instructions. The $rs1$, $rs2$, and rd fields point to locations within the processor register file to specify where to read from and write to as required by each instruction.

We choose to model TIGRA using the R-type instruction, as this type provides the most generality and flexibility with data movement between custom logic and the processor. When creating TIGRA designs, we will use the major opcodes 0001011 and 0101011 as these are reserved by the RISC-V authors for custom instructions and guaranteed to not be used by future ISA extensions. As R-type instructions contain both $funct7$ and $funct3$ fields, this enables the creation of 1024 unique instructions per major opcode. Therefore, only one major opcode needs to be used for TIGRA custom instructions.

3.1.2 Interface Requirements

We define a minimum of seven required signals in TIGRA to complete the interface between a processor and an FPGA based custom logic accelerator. Following the format of R-type instructions, two data inputs and one data output are required to meet all specifications. The signals cl_rs1 and cl_rs2 of Figure 3.2 correspond to the $rs1$ and $rs2$ fields, respectively, of an R-type instruction. Specifically, the rsX signals should pass the corresponding values from the register file of the CPU to TIGRA, the same data that would be passed to the ALU of the processor. The data output of the custom logic, $cl_outData$, corresponds to the rd field of the instruction and will ultimately be written back to this location in the register file. These three signals follow the data-width of registers within the implemented processor.

To synchronize with the processor without requiring queues or incurring additional latency, TIGRA uses two main signals, cl_mem_valid and cl_valid. The CPU must send logic high via cl_mem_valid to signify the CPU has retrieved the values of $rs1$ and $rs2$ from the register file and that this data exists and is valid on the corresponding signals, cl_rs1 and cl_rs2, to the custom logic. This allows the custom logic to latch the incoming data and begin execution on the next clock cycle. The custom logic, in turn, sends logic high via cl_valid to the processor to signify the data on $cl_outData$ is valid and the CPU can latch this on the next clock cycle. The custom logic sends a logic low via cl_valid to provide backpressure to the CPU and generate stalls necessary during execution for multi-clock cycle hardware.

We designed TIGRA with the intention of complex custom logic that may perform more than one operation. To maintain generality, we require the CPU send the full instruction via cl_insn to the custom logic. This enables usage of the $funct7$ and $funct3$ fields of the opcode, allowing the custom logic to decode the intended functionality and compute the correct result. For many

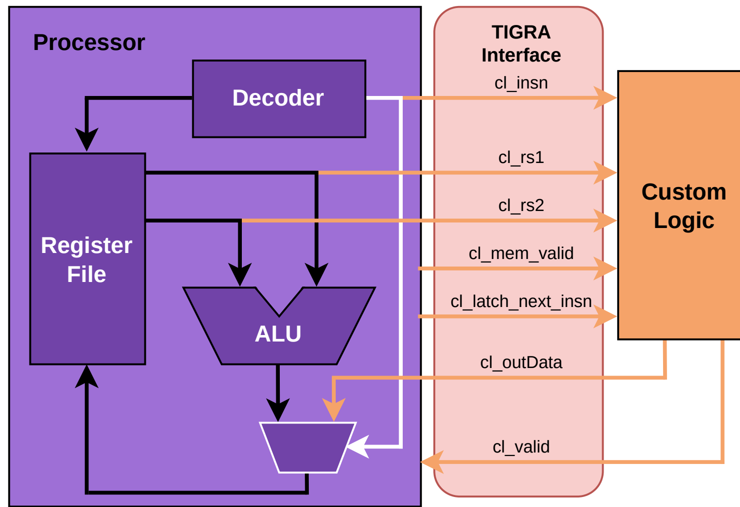


Figure 3.2: A processor coupled with custom logic via the TIGRA interface. The signals connecting the Custom Logic to the processor block (shown in orange) are required to handle communication with the user designed hardware. The multiplexer and the signal connecting it to the decoder (shown in white) represent the minimal logic added to the processor to ensure TIGRA instructions follow the standard ALU flow. Clock and reset signals are not shown.

processors, the next instruction may be decoded before the currently executing instruction completes operation. This motivates the need for the *cl_latch_next_insn* signal, used by the processor to send a logic high when the currently executing instruction switches within the processor pipeline.

3.1.3 Processor Modifications

Figure 3.2 shows the minimum additions to a processor to create the TIGRA interface. As we intend custom logic connected via TIGRA to act as a tightly coupled accelerator, we require designs follow the standard ALU flow within the pipeline. The processor sends the current instruction and register data over the corresponding signals described in Section 3.1.2. It should also multiplex the data from *cl_outData* with the output of the ALU, using the correct instruction to determine which data to send for write back to the register file.

Not shown in the block diagram is the logic required to handle the synchronization signals. Every processor design needs to handle synchronization logic differently, but the TIGRA interface must remain constant and general across all designs. Figure 3.3 shows the synchronization of a simple, four-stage CPU pipeline, similar to that discussed in Section 4.1. The developer integrating TIGRA in a design is responsible for the following:

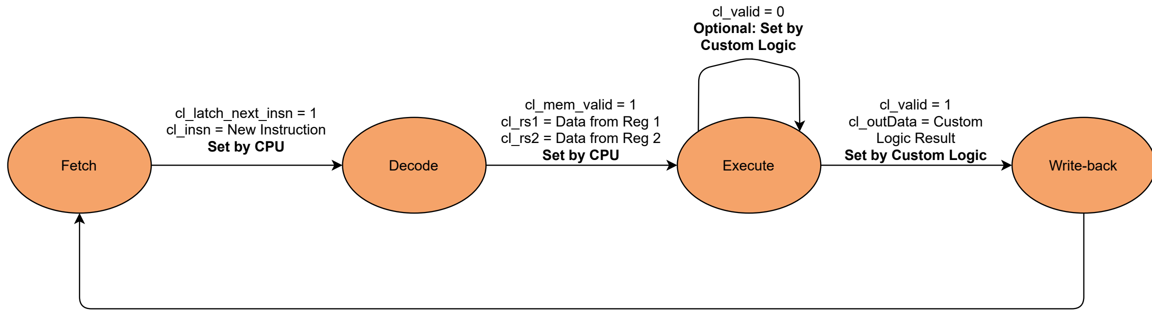


Figure 3.3: An example four-stage CPU pipeline, showing how to set each TIGRA control signal in the host processor and the custom hardware design.

- Ensuring the processor stalls at the correct stages of the CPU when the custom logic sets cl_valid to a logic low.
- Sending active-high on cl_mem_valid when the data on cl_rs1 and cl_rs2 is guaranteed to have valid data from the register file.
- Sending active-high on $cl_latch_next_insn$ when the currently executing instruction changes.
- Ensuring all required signals are available outside of the CPU, allowing custom logic to exist outside of the design to maintain the modularity of the TIGRA interface.

Figure 3.4 provides a simplified and generalized flowchart to explain the process of adding a TIGRA interface to an existing processor. Much of the effort required to introduce the generic TIGRA interface matches the effort required to add any new execution unit to a processor, as shown by the light orange highlighted area in Figure 3.4. These upper steps explain the effort required to modify the existing processor for any new functionality, starting with understanding the existing processor functionality to determine if additions must be made to the base CPU logic. Every processor will require different edits to include custom functionality, with some requiring the developer to add a mechanism to allow the processor to stall while others already include example execution units that stall.

After adding and verifying the new execution unit, developers must generalize the logic and instantiate the TIGRA interface defined in Figure 3.2. This required work is unique to adding TIGRA and is represented by the purple highlighted box of Figure 3.4. Within the TIGRA execution unit, developers must handle all synchronization and communication with the processor to maintain

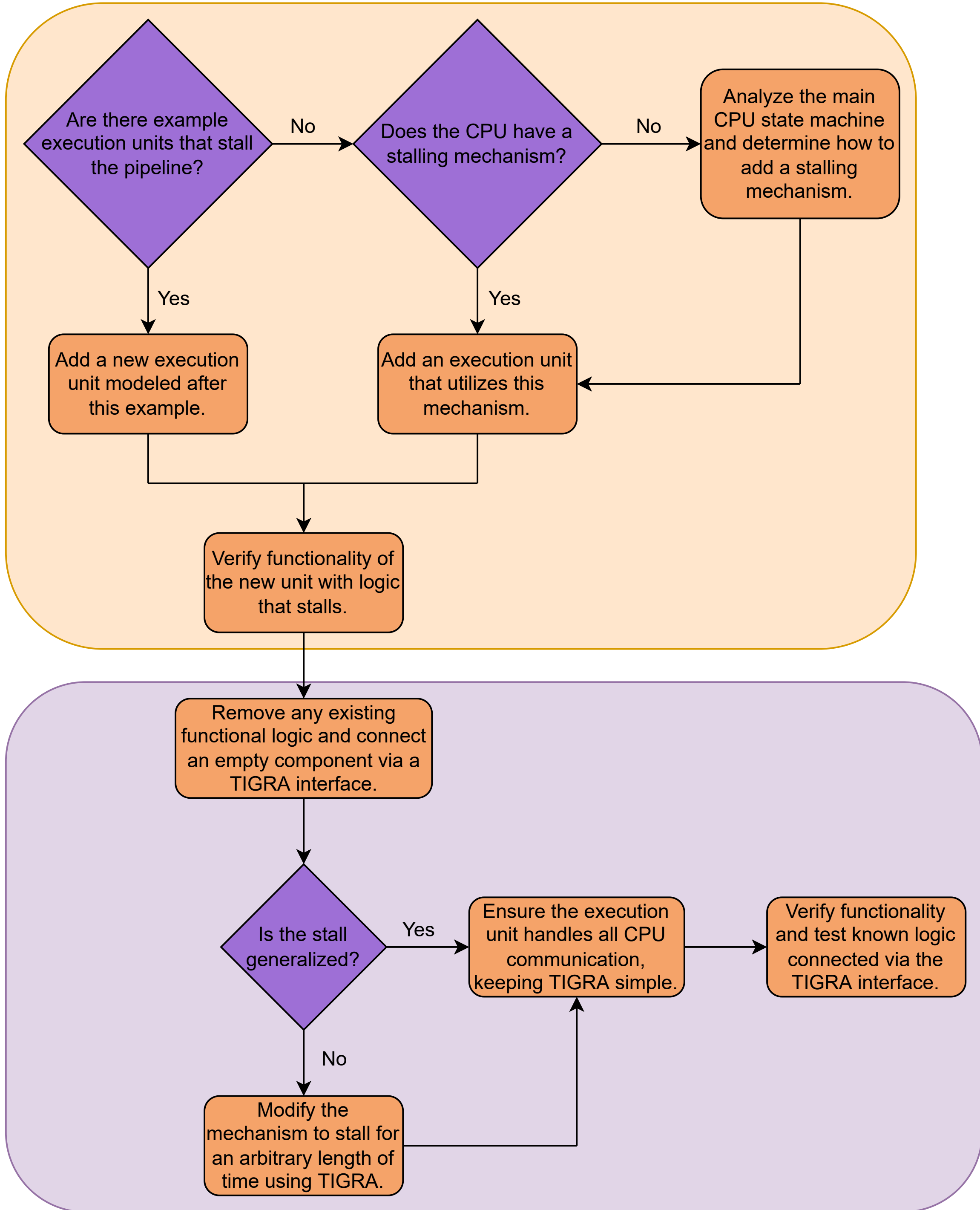


Figure 3.4: A flowchart detailing the generalized process to setup a TIGRA interface on an existing processor. The upper orange highlighted area represents the process required when adding any new execution unit to an existing CPU, and the lower purple area represents the unique effort required to generalize the new execution unit for TIGRA.

the simplicity and generality of the TIGRA interface. Once completed, no further modifications are required to the processor to introduce new custom logic.

3.2 Custom Logic Design and Usage

In this section, we describe how to develop custom logic to interface with TIGRA and provide insight into using these custom logic instructions within a normal application. This section targets developers aiming to create an FPGA based accelerator on a TIGRA enabled processor.

3.2.1 Developing Custom Logic

When developing a custom accelerator for use with TIGRA, users can focus on creating hardware that meets their requirements. TIGRA is designed to work with any accelerator with minimal effort and overhead, allowing developers the freedom to design their hardware and test within any simulation environment. This enables the use of existing custom hardware, such as Tiny_AES [19] or PACoGen Posits [28] discussed in Chapter 4. Once a given accelerator has been verified via simulation, the user must create a wrapper to enable communication via the TIGRA interface. Custom logic developers should be familiar with the functionality of TIGRA described in 3.1.2 and are responsible for the following:

- Connecting their design to the system clock and reset, if necessary.
- Properly decoding the instruction from *cl_insn*, latching if necessary on a logic high value from *cl_latch_next_insn*, to ensure the custom logic accelerator is aware when a TIGRA instruction is currently in the execute stage of the pipeline.
  - For some designs, such as un-clocked hardware that completes quickly, latching the next instruction may not be necessary as long as the correct answer can be provided on the next clock edge. These designs should always set *cl_valid* to a logic-high, ensuring there are no unnecessary stalls in the CPU pipeline.
  - This puts the onus of choosing minor opcodes for each function within the custom logic on the designer at this stage of development.
- Properly handling the *cl_mem_valid* signal to only latch the input data values when this signal is logic-high if required for the given custom hardware.

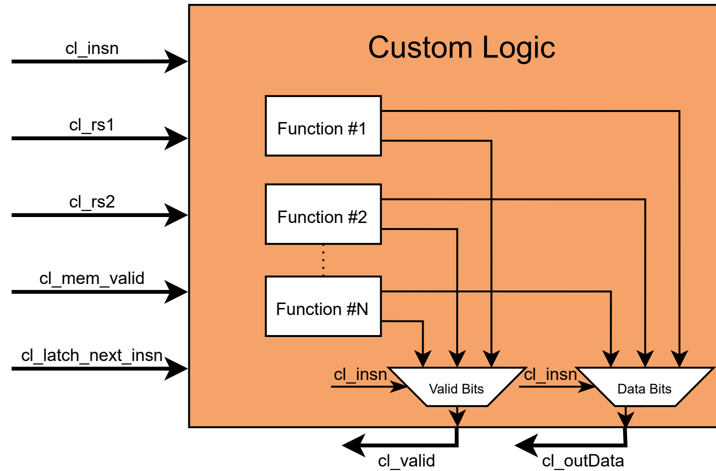


Figure 3.5: An example custom logic setup implementing many accelerators. The signals external to the custom logic block represent the TIGRA interface. The necessary instruction decoder(s) and other logic is not shown to maintain generality and keep the design readable.

- Setting *cl_valid* to a logic-high when the custom logic has successfully output the correct data on *cl_outData*, allowing the CPU to release any stalls and latch the result on the next clock edge.

We have designed TIGRA to ensure that no overhead is incurred when connecting a custom logic accelerator to the interface. Developers must be careful when designing the wrapper to not introduce latency to their design. They should only use latches when necessary, such as with multi-cycle designs or hardware that implements more than one functionality. The TIGRA wrapper should be kept as lightweight as possible to maximize the possible clock frequency within the custom logic.

While the design practices listed above dictate guidelines for a singular custom logic accelerator, developers may include multiple accelerators or functions within their custom logic design. Shown in Figure 3.5, the only additions required are multiplexers to ensure the correct valid and data bits are sent back to the processor and a decoder to parse the incoming instruction, if necessary. The diagram shows *cl_insn* as the select line for the multiplexers, but this may be changed to the internal latched instruction for designs that execute over many clock cycles. To maintain generality, developers may pass all signals to each individual accelerator and handle data latching and processing within each accelerator or create a decoder at the top level to handle proper instruction decoding and latching.

As the TIGRA interface is processor agnostic, custom logic designs connected via this in-

terface are completely portable. The generality of the TIGRA interface enables developers to create a design once and use the same custom logic on any TIGRA enabled processor. The portability of custom logic designs is further explained in Chapter 4.

3.2.2 Using a TIGRA Instruction

After the custom logic has been wrapped and connected to the TIGRA interface, the device is ready for use in an application. Devices connected via TIGRA are applicable for testing and usage in both simulation and hardware based environments, if the hardware exits. Bare-metal applications, or those used in simulation within design tools, will require developers write instruction level code. Example instruction calls for the test cases used to test the design in Section 4.1 can be seen in Appendix A. In these cases, developers must use the major TIGRA instruction opcode and supply the correct $funct7$ and $funct3$ fields to ensure the intended instruction is executed.

When working with a more complex processor such as those generated by Rocket Chip, bare-metal execution may not be possible. For these cases, modification to the RISC-V toolchain may be required to enable inline assembly usage of TIGRA instructions. [34] provides a detailed guide for adding TIGRA instructions to the RISC-V ISA in Appendix D. This enables the RISC-V compiler to understand inline calls to TIGRA and allows developers to include these instructions within standard C code. Further, the RISC-V toolchain provides the necessary utilities to generate the instruction level code described previously, allowing developers to write C code and receive the necessary code for simulation environments.

3.3 Summary

Completing the one-time TIGRA integration ensures that no further modification to the underlying processor architecture is required, abstracting the process from end-users. This initial, one-time programming effort requires in-depth knowledge of the chosen CPU to ensure no side effects are incurred during development. Connecting via TIGRA and moving the custom logic to outside the CPU provides many benefits:

- Future developers need not focus on modifying the instruction pipeline, thus simplifying custom logic integration and reducing the required programming effort to incorporate custom logic accelerators to a CPU.

- Leveraging partial reconfiguration on FPGAs, developers testing designs in simulation only need to recompile the custom logic portion of the design, speeding up design time.
- The custom logic may be swapped out more easily in simulation and real hardware via partial reconfiguration, mimicking the flow used by OpenCL and other loosely coupled accelerators.
- TIGRA connected custom logic designs can be used on any TIGRA enabled processor.
- The custom logic will perform as a tightly coupled accelerator with fine-grained execution, zero added latency, and enables more efficient testing of future instruction set extensions.
- Developers can create C-based algorithms with inline assembly to use TIGRA instructions within an existing algorithm.

Chapter 4

Case Studies

The case studies of this work implement TIGRA in the simple PicoRV32 [51] processor, the Rocket Chip [7] processor generator, and the Taiga [40] processor. Each of these designs were tested with the same two applications: AES 128-bit encryption and posit arithmetic. The test cases chosen to verify TIGRA are meant to simulate real use cases of the interface in an HPC or hardware design environment. The AES 128-bit test case uses Tiny AES [19] and represents an FPGA based hardware accelerator that can compute an iteration of the algorithm in a fraction of the time of software based designs. The posit test case uses the PACoGen posit arithmetic core generator [28] and represents a custom instruction set extension based in FPGA hardware. And finally, we also test on a built-in functionality as a baseline comparison for TIGRA, demonstrating that TIGRA introduces no added latency and provides the same functionality as a built-in design. In this chapter, we discuss the implementation of TIGRA within these processors and provide hardware based verification of the PicoRV32 design on Amazon Web Services (AWS) F1 cloud services with Xilinx XCVU9P-FLGB2104-2-i FPGAs.

4.1 PicoRV32

[49] implements TIGRA within the PicoRV32 processor from [51]. PicoRV32 is an in-order, non-pipelined processor based on the RISC-V instruction set architecture that aims to reduce area and increase possible clock speeds to serve as an auxiliary CPU for FPGA and ASIC designs. This processor comes equipped with an optional Pico Co-processor Interface (PCPI) that acts as a tightly

coupled accelerator to enable the multiplication and division instructions from the RISC-V ISA. This interface is required to enable the "M" extension for RISC-V which enables integer multiplication and divison. The PCPI interface acts as a direct comparison to the generic TIGRA interface, and will help verify that TIGRA introduces no latency to designs.

4.1.1 PicoRV32: TIGRA Integration

The simple design of PicoRV32 facilitated a very straightforward integration of TIGRA. Figure 4.1 represents the state diagram of PicoRV32, with TIGRA following the ALU flow represented as the bold orange line in the diagram. PicoRV32 fetches the next instruction during the $fetch$ state of the diagram, and once decoded will set the $cl_latch_next_insn$ signal of the TIGRA interface to logic high. This will enable connected custom logic to latch the instruction that is beginning to execute on the CPU. When PicoRV32 reaches the ld_rs1 state, it begins fetching data from the register file and sets cl_mem_valid to a logic high. This allows the custom logic to latch the incoming data, if necessary, on the transition to the $exec$ state by guaranteeing valid data on cl_rs1 and cl_rs2 signals of the TIGRA interface.

During the $exec$ state, if the custom logic requires more than one clock cycle to complete, the design sets cl_valid to logic low and cause a stall in the PicoRV32 pipeline. Once custom logic finishes execution and can guarantee data on $cl_outData$ is correct, it sets the cl_valid signal which ends the stall in the CPU on the next clock edge. The CPU transitions back to fetch, during which write back occurs and the cycle begins again. PicoRV32 spends 2 clock cycles in fetch, one for write back and the other for fetch, meaning this CPU executes in four stages. Developers can abstract the decisions made to integrate TIGRA in this design to other four stage CPUs.

The PicoRV32 with TIGRA design requires a few extra signals to handle proper communication between the CPU and connected custom logic and ensure stalls occur only when necessary and within the execute stage of the pipeline. We edited the CPU decoder to understand custom instructions and verify these instructions follow the same flow as the built in ALU. As PicoRV32 is a simple processor that is capable of running without an operating system, we chose not to edit the RISC-V compiler to include TIGRA instructions and supplied the instructions for each test manually. Overall, including TIGRA into PicoRV32 required the modification or addition of 32 lines of code in the main source file for the processor.

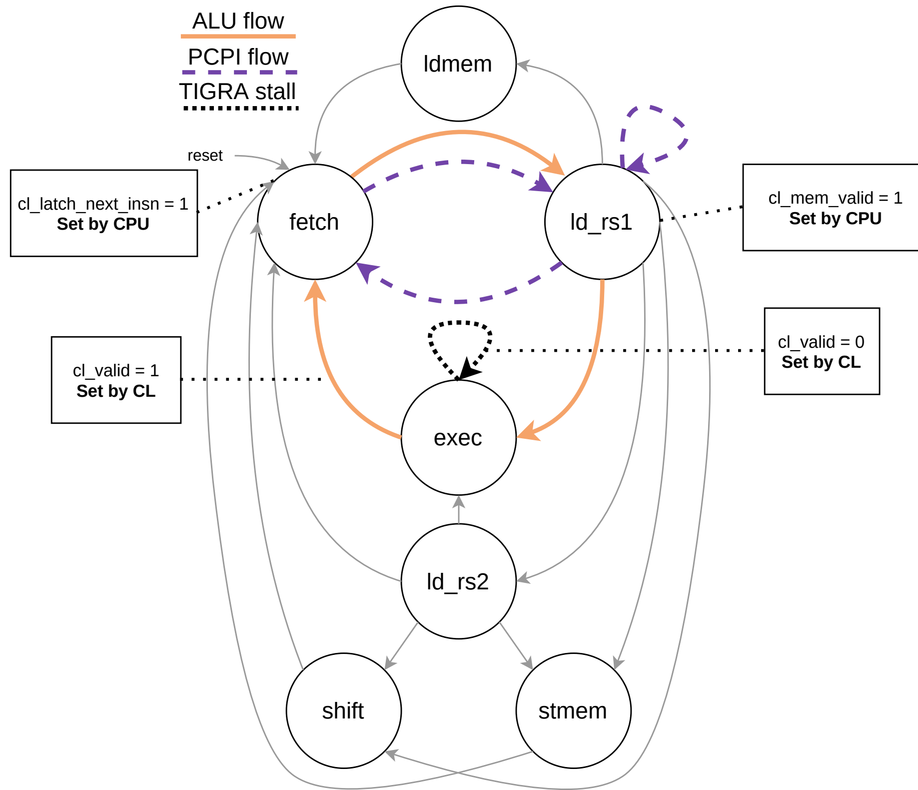


Figure 4.1: The PicoRV32 state diagram, emphasising the PCPI and TIGRA flows. The boxes represent the value and timing of TIGRA control signals during execution.

4.1.2 Hardware Verification with AWS

PicoRV32 allows for bare-metal execution of the processor, allowing developers to directly supply instructions and enable hardware verification of the device. We implemented the PicoRV32 processor with and without TIGRA on AWS F1 FPGAs utilizing the AWS HDK [4]. The AWS HDK allows users to implement hardware based designs on FPGAs and communicate with them via a PCIe shell interface. The AWS F1 shell handles partial reconfiguration of the available FPGA logic and all communication to and from the device. Utilizing C function calls provided by the AWS HDK, $fpga_pci_peek$ and $fpga_pci_poke$, users can read from and write to defined memory spaces within a custom logic design during runtime.

Figure 4.2 represents the layout of an AWS F1 node when connected with the designed wrapper for PicoRV32. As shown, the AWS wrapper requires 3 main components: memory, the PicoRV32 processor, and the custom logic under test. Listing 1 demonstrates the mapping of each of these components for the posit test case. This listing begins on line 217 as the signal declarations are

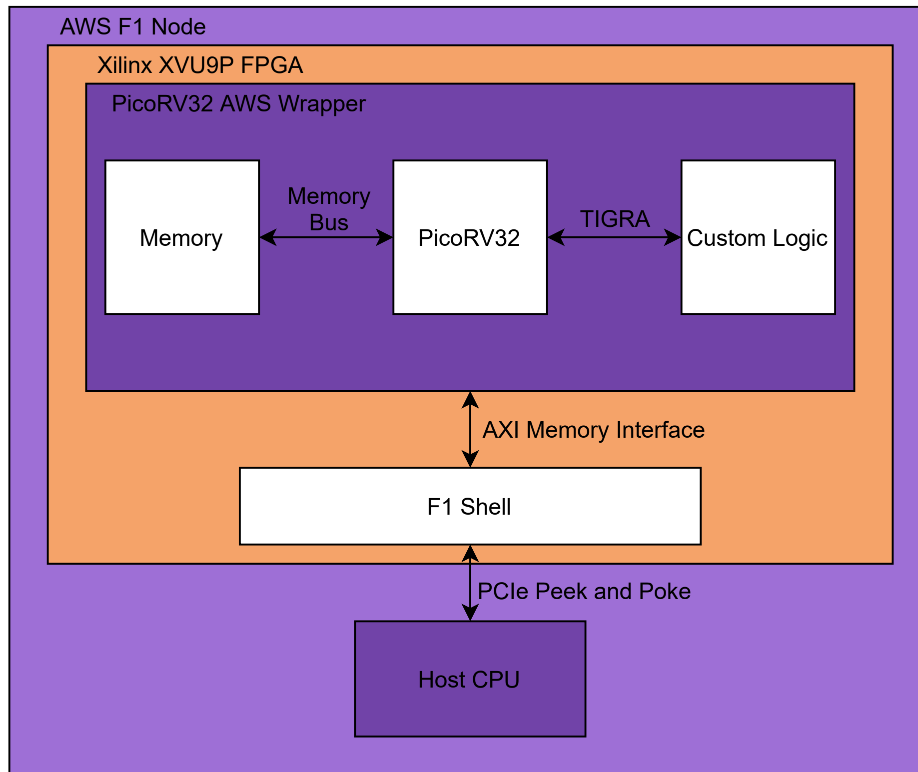


Figure 4.2: The AWS F1 HDK hierarchy when connected with a full PicoRV32 with TIGRA enabled and connected to custom logic.

omitted for brevity. Examples from [4] provide the basic write (poke) and read (peek) functionality, and this code is demonstrated in lines 217-281. For these designs, we do not utilize the poke function and therefore do not modify the write request and response blocks shown on lines 217-244. However, we do intend to read from memory using peek, and simply need to provide the memory locations to return when the host computer requests data from a valid address. The required modifications are shown on lines 276-278, where $BASE_REG_ADDR$ is a user defined value and $memory$ is the Verilog reg array used to store instructions and program data.

Lines 287-308 of the listing show mapping the PicoRV32 processor with TIGRA, which remains the same for any custom logic accelerator. As each accelerator connects via TIGRA to the processor, lines 310-320 should also remain constant except for the device instantiated. The remainder of the code shows the synchronization required with the PicoRV32 memory interface and the reg array that acts as the data storage for the device. Lines 326-354 initialize the instruction memory and initial long-term storage values, while lines 355-386 represent the minimal synchronization

required to write or read data from memory. The instructions on lines 347-350 represent a *NOP* loop to prevent accessing invalid memory locations. This is necessary as designs loaded onto an F1 FPGA begin execution immediately and continue until manually cleared by the user.

This design is replicated with for use with AES-128 bit encryption, multiplication over the TIGRA interface, and multiplication using PCPI. To switch the custom logic, we only need modify the component instantiation on lines 310-320 of Listing 1. For the PCPI multiplication, we switch out the PicoRV32 core with the unmodified version and remove the custom logic from the design. Each test case utilizes a unique set of instructions, found in Appendix A. Listing 3 shows the instructions required to complete one block of AES-128 encryption, and Listings 4 and 5 show the changes made to implement multiplication with TIGRA and PCPI, respectively. The multiplication programs differ only on line 4, where the TIGRA implementation uses the custom instruction while the PCPI version uses the standard RISC-V multiplication instruction.

4.2 Rocket Chip Generator

[34] implements TIGRA within the Rocket Chip generator from [7]. Rocket Chip is a parameterizable design generator built on Chisel that allows users to select between in or out-of-order processors (Rocket or BOOM), enable different RISC-V extensions or a RoCC accelerator, and generate synthesizable RTL that can boot Linux. Rocket Chip generated processors are designed with a standard 5-stage RISC-V pipeline which include fetch, decode, execute, memory, and writeback stages. These cores allow for pipelined execution with up to 5 instructions executing concurrently, each in a different processor stage. The developers include a Simple Custom Instruction Extension (SCIE) interface that allows the integration of custom logic directly into a Rocket Chip generated processor. However, SCIE instructions only support single clock cycle instructions as these designs do not communicate with the stalling mechanisms in the CPU.

The developers of the Rocket Chip generator package the cycle accurate simulator, Verilator [44], to test modifications to the processor. Verilator is capable of simulating the boot process, executing compiled executables from the RISC-V compiler, and emitting the waveform traces of all signals during an execution run. Using GTKWave [8] or similar wave viewer, developers can view the output waveform traces to verify signals update as expected, view registers to validate processor writeback, and check simulation specific traces or debug values to help troubleshoot design issues.

We rely on the Verilator output when testing TIGRA to ensure the processor modifications function as expected.

4.2.1 Rocket Chip: TIGRA Integration

Inclusion of TIGRA aims to solve the drawbacks of SCIE and RoCC, giving developers the ability to easily create new hardware that executes in an arbitrary number of clock cycles without adding latency to the design. We modeled the initial design of TIGRA after the SCIE pipelined interface, but we added backpressure mechanisms to allow any connected custom hardware to stall the processor pipeline as necessary. We modified 5 files within the Rocket Chip source code to ensure the generator maps RISC-V *custom-0* and *custom-1* instructions to TIGRA, stalls the processor pipeline when the *cl_valid* output is logic low, and allows users to enable the interface as a parameter during configuration. Most of the changes centered on ensuring the processor will stall correctly for multi-cycle instructions and did not introduce latency for simple hardware that executes in one cycle or less.

As shown in Figure 4.3, the CPU sets the *cl_mem_valid* signal to a logic high when a TIGRA instruction enters the execute stage of the pipeline, this guarantees data on the cl_rs1 and cl_rs2 lines are valid and is the earliest this data is available in the Rocket processor. The custom logic will in turn latch the incoming data, if necessary, on the next clock edge. The *cl_latch_next_insn* signal is not required here as Rocket Chip generated processors maintain a register that contains the value of the instruction occupying each stage of the pipeline. We simply supply this on the *cl_insn* signal and lock *cl_latch_next_insn* to a logic high and the instruction value will not change during a stall. The custom logic then sets *cl_valid* to a logic low, if required, and finally sets the same signal to a logic high once *cl_outData* contains a valid result. This initiates a stall for multi-clock cycle instructions and clears it once completed.

Stalling in Rocket requires stalling more than a single instruction, as shown by the shaded boxes in Figure 4.3. This is handled by a series of latched signals that update on every edge of the clock, beginning in the decode stage of the pipeline. The currently executing instruction may initiate a stall in decode to prevent instructions from attempting to move to execute during the stall. This stall is also directly applied to the fetch stage, which prevents increments to the program counter to ensure no future instructions are lost. On the next rising clock edge, instructions occupying the memory and write back stages advance and a stall is initiated in the now empty memory stage.

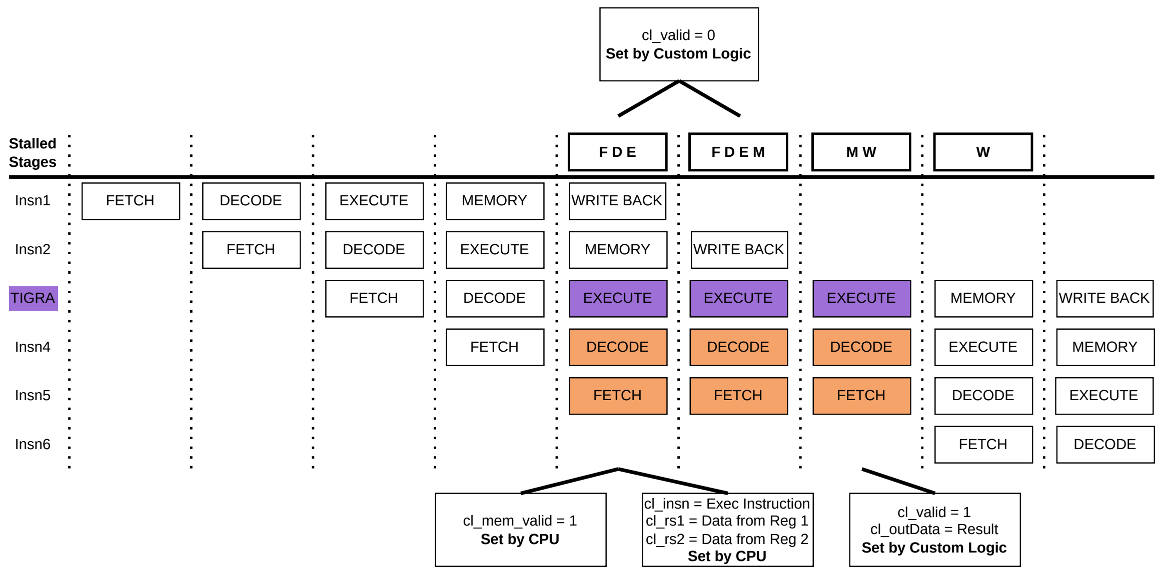


Figure 4.3: Example execution stall of a TIGRA instruction in the Rocket pipeline. The purple highlighted instruction (TIGRA) stalls in the execute stage of the pipeline, causing later instructions (Insn4 and Insn5 highlighted in orange) to stall in decode and fetch.

Next, the instruction in the write back stage completes and this stage is stalled to prevent erroneous writes back to the register file. Once the custom logic sets *cl_valid* to a logic high, the stalls are lifted in the fetch, decode, and execute stages immediately. The stalls in memory and writeback are lifted as new instructions occupy these stages of the pipeline.

As Rocket Chip designs are capable of booting Linux, designers can modify the RISC-V compiler to understand custom instruction opcodes, which allows users to write C programs using inline assembly to test designs. This feature requires the developer to create the opcode masks and matching strings for each individual instruction, modify 2 more files in the RISC-V compiler by adding 3 lines of code per custom instruction, and recompile the tools to ensure these instructions are understood by future programs. In total, adding a TIGRA interface with 16 instructions to the Rocket Chip source in Chisel and the RISC-V compiler required the addition or modification of 180 lines of code to meet the specifications. Each new instruction introduces, at minimum, 4 new lines of code and requires a recompile of the RISC-V toolchain.

These totals are counted with no custom logic connected to the TIGRA interface and can serve as an estimate for the minimum work required to introduce a tightly coupled accelerator or introduce a new instruction set to Rocket Chip generated processors. Further, developers seeking to modify the Rocket Chip generator will be required to understand the processor pipeline and make

equivalent modifications in the Chisel source code.

4.2.2 Rocket Custom Coprocessor (RoCC)

Developers generating Rocket Chip processors have the ability to include a coprocessor interface, the Rocket Custom Coprocessor (RoCC). RoCC aims to provide designers with a built-in interface that allows for custom hardware integration without requiring modifications to the connected CPU. This interface provides connections to the Floating Point Unit (FPU) and processor memory which enables connected designs to utilize existing processor resources, when available, to provide more flexibility within connected custom hardware. When compared with the TIGRA interface, the RoCC interface provides more functionality at the cost of requiring users to handle more processor communication when connecting a custom logic design.

RoCC instructions do not follow the standard instruction flow of other Rocket Chip functions. As shown in Figure 4.4, custom hardware connected via a RoCC interface begin execution as the calling instruction exits the writeback stage of the Rocket Chip pipeline. When occupying the different stages of the pipeline, RoCC instructions set up data and prepare dedicated signals for custom hardware execution. These instructions utilize a series of signals to provide synchronization with the main CPU, allowing for arbitrary execution times. Once complete, the custom logic signals that data is valid and ready for writeback. The resultant data is written to the CPU registers when the writeback interface is available, or the CPU may stall to allow writeback in the case of data dependencies or if an upcoming instruction requires usage of the RoCC connected hardware. A further analysis of data movement is shown in Section 5.3.4.

4.3 Taiga Processor

The Taiga processor [40] was developed for optimized usage within an FPGA, aiming to reduce resource utilization and improve maximum clock speed in hardware. Similar to Rocket Chip generated designs, Taiga allows users to customize available RISC-V ISA extensions, cache sizes, and multiple other parameters. This processor is also fully synthesizable and capable of booting Linux and running compiled RISC-V programs. Taiga utilizes only 26% of the FPGA resources with nearly double the clock frequency of a similarly configured Rocket Chip CPU when synthesized for a Xilinx FPGA.

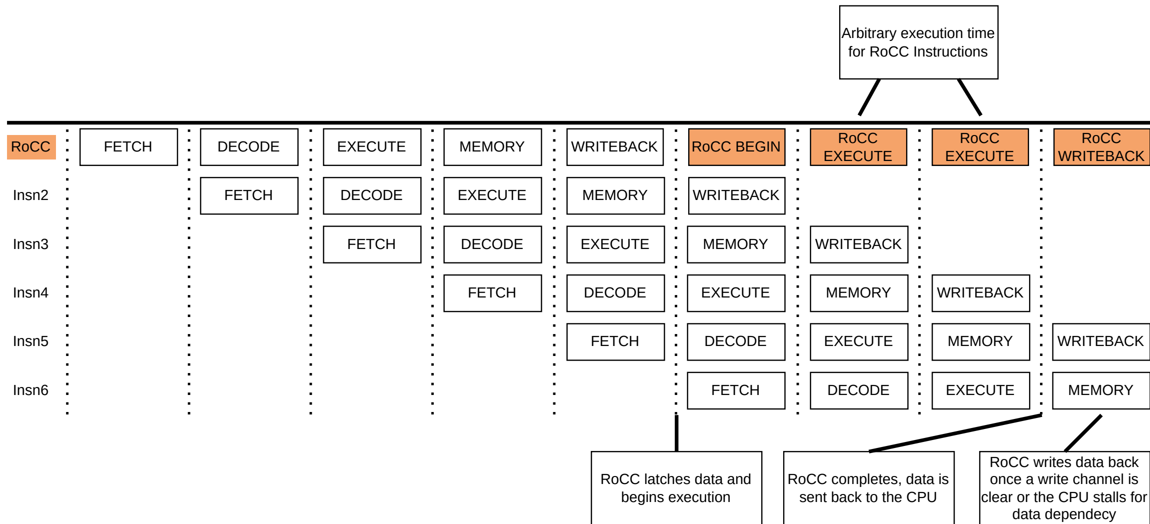


Figure 4.4: Execution of a RoCC instruction in the Rocket Chip pipeline. As shown, custom hardware connected to a RoCC interface begins execution as the instruction exits the main Rocket Chip pipeline.

Taiga processors utilize a four-stage RISC-V pipeline that merges the memory and writeback stages of a standard five-stage design into a single load/store stage as shown in Figure 4.5. The design decouples each of the RISC-V pipeline stages via queues and uses parallel execution units that help mask the latency of multi-clock cycle instructions, requiring extra hardware to ensure instructions write back in the correct order. The authors created a tag management system with a priority queue that prevents out-of-order writeback and only stalls the CPU when a requested execution unit is busy or data dependencies force the CPU to wait for the appropriate write back. Taiga's decoupled design facilitates customization and does not require developers to modify the instruction pipeline to incorporate hardware that takes more than one clock cycle to stall.

4.3.1 Taiga Processor: TIGRA Integration

Modifying Taiga to include a TIGRA interface required the addition of a new execution unit to build the logic to communicate with the base processor and create the interface. New Taiga execution units require developers to manage a tag to ensure the return data is associated with the appropriate instruction during write-back, and these units must handshake with the write-back unit to prevent unnecessary processor stalls. To avoid adding signals to the user-side of TIGRA, we handled all of the tag management and handshaking within the new TIGRA execution unit to

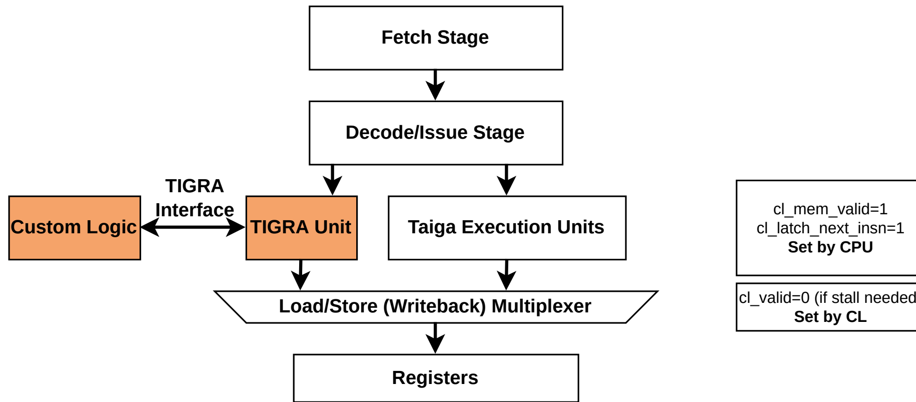


Figure 4.5: Taiga's general pipeline flow showing the necessary additions to include a TIGRA interface. The added unit is shown separate from the Taiga execution units to represent that each operates independently of each other. Each vertical stage is separated by a queue to facilitate the decoupled operation and ensure instructions complete in order.

maintain simplicity when adding custom hardware via the interface. We followed the multiplication unit's design as an example to build the new execution unit, which required further modification of 7 more files to allow the processor to issue RISC-V *custom-0* and *custom-1* instructions to the TIGRA execution unit and generate the necessary write-back units and hardware during synthesis or simulation. The result allows users to optionally include a TIGRA interface during configuration, maintaining the decoupled and customizable design of the Taiga processor. Figure 4.5 demonstrates the additions to the processor, where each vertical arrow represents an instruction queue. Taiga sets both the *cl_mem_valid* and *cl_latch_next_insn* signals to logic high when a TIGRA instruction enters the execute stage of the pipeline as this is when the CPU sends the data to the TIGRA execution unit. The instruction must be latched here as new instructions will continue to be issued even if the custom logic requires a stall.

The hardware may set *cl_valid* to a logic low to stall in the Load/Store stage until completed, and the Taiga processor will stall the pipeline if an upcoming instruction requires an execution unit that is busy or waiting to writeback due to the hardware stall. The writeback multiplexer requires a handshake with each execution unit in order to verify data is returned in the proper order and allow the execution unit to accept new inputs.

Enabling TIGRA instructions in the RISC-V compiler required modification of 2 more files, 3 lines of code per custom instruction, before rebuilding the tool-chain to allow the compiler to understand inline assembly for TIGRA instructions. Verilator can then simulate the boot process,

run a program, and save the output waveform for verification. Adding the TIGRA interface with 16 instructions as a configuration option within the Taiga processor required the addition or modification of 147 lines of code total. As with Rocket, these totals are counted with no custom logic connected to the TIGRA interface and can serve as an estimate for the minimum work required to introduce a tightly coupled accelerator or introduce a new instruction set to a Taiga processor. While the decoupled design eases the programming effort to introduce new execution units to the processor, new developers still need to learn the write-back handshake protocol and tag management to ensure new designs work properly. Each additional instruction introduces at least 4 lines of code and each execution unit will require the developer to modify multiple files within the Taiga source code.

4.4 AES 128-bit Custom Logic

Tiny AES [19] provides very small and quick AES encryption on 128-bit, 192-bit, and 256-bit data. We choose to operate in the 128-bit encryption as a proof as both processors described in this chapter are restricted to 32-bit registers, meaning there is no built in support for larger data. Because of this, we must develop the custom logic in a way that let's us transfer and work with the required 128-bit state, key, and result data-width. To do this, we take advantage of the R-type instruction that allows transfer of two registers at a time as input when writing data to the custom logic. To write the state and key, 2 instructions each are required to write 64 bits at a time of each value. As R-type instructions only include a single destination register, we can only read 32 bits per instruction from the result, dictating that we must include 4 instructions to read 32-bit chunks from the resulting data into the CPU registers. These minimal 8 instructions require 3 bits to encode each with a unique instruction value, which fits perfectly within the $funct3$ field of R-type instructions as shown in Table 4.1. To keep logic minimal, we set the custom logic to begin encryption once the full state and key are received from the connected CPU.

Figure 4.6 shows the block diagram of the AES custom logic. The decoder receives the CPU instruction to decode the appropriate functionality and starts the counter when the last bits of the state are written to it's 128-bit register via the $tigra_3$ instruction. The demultiplexer on the left selects which part of the state or key to latch the incoming data for the first four instructions and sets the $startCounter$ signal to logic high once all data is latched. The counter begins on this clock

Table 4.1: Custom instructions for AES

funct3	Instruction	Function
000	tigra_0	Store lower half of key
001	tigra_1	Store upper half of key
010	tigra_2	Store lower half of state
011	tigra_3	Store upper half of state and start encryption
100	tigra_4	Return AES output[31:0]
101	tigra_5	Return AES output[63:32]
110	tigra_6	Return AES output[95:64]
111	tigra_7	Return AES output[127:96]

cycle, setting *cl_valid* to logic low until the tiny_aes core completes the encryption 21 clock cycles later. This will cause the connected processor to stall until until the encrypted result is available. The larger multiplexer sends either the appropriate 32-bit quarter of the AES output or a string of all zeroes to the *cl_outData* signal.

This hardware connects to the TIGRA interface on the PicoRV32, Rocket Chip, and Taiga processors without any modification. Building this custom logic required the addition of only 92 lines of code, the bulk of which are simple assignments within Verilog case statements to latch the incoming data or provide the correct output when reading the encrypted result. The tiny_aes core was unmodified and used as a design component within the designed wrapper.

4.5 Posit Arithmetic Custom Logic

PACoGen includes 3 built in functions for posit<32,6> encoded numbers, addition, multiplication, and division. Each of these implementations is fully pipelined and takes 5, 6, and 12 cycles to compute the result, respectively. This design facilitates use with TIGRA and requires very little additional logic, as shown in Figure 4.7a. The custom logic wrapper decodes the input instruction and sends the start signal to the correct function on a logic high from *cl_mem_valid*. The added multiplexer uses the $funct3$ field of the instruction to determine which function's data and valid bit to pass through the TIGRA interface via *cl_outData* and *cl_valid*.

The design in Figure 4.7a is unusable for simulation purposes for the Rocket Chip and Taiga processors as the PACoGen cores for multiply and divide use Xilinx specific constructs to optimize multiplication operations. As both of these processors utilize the cycle accurate Verilator simulator, they are unable to compile and test designs when using the posit multiply and divide units. We

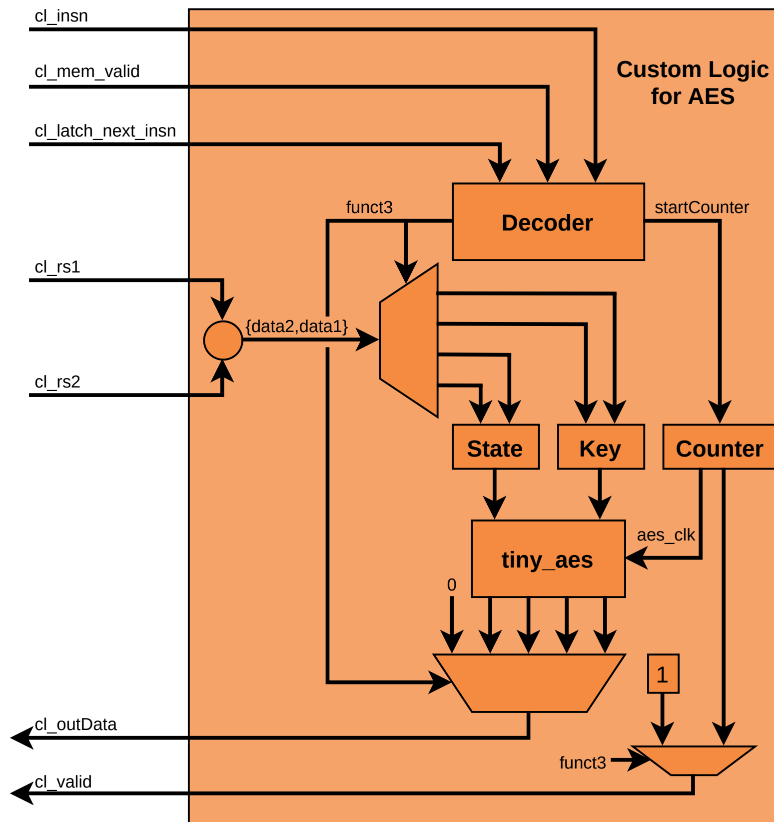
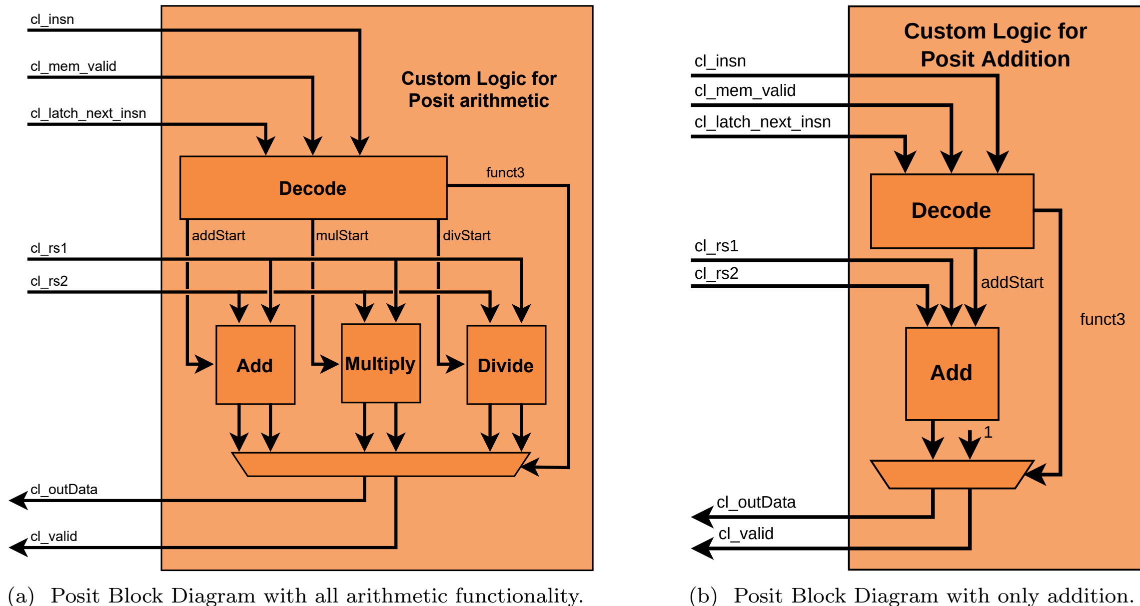


Figure 4.6: AES-128 Bit custom logic block diagram.



(a) Posit Block Diagram with all arithmetic functionality.

(b) Posit Block Diagram with only addition.

Figure 4.7: Posit arithmetic custom logic block diagrams for use with PicoRV32 or hardware compilation (a) and for use with the Rocket Chip generator and Taiga simulators (b).

modified the design of the custom logic as shown in Figure 4.7b by removing the offending multiply and divide units. The decoder logic remains unchanged and we modified the multiplexer to output a default value of logic high on *cl_valid* for the unused instructions.

The design including addition, multiplication, and division functionality required 80 additional lines of code to connect to the TIGRA interface, while the addition only wrapper required only 48 additional lines of code. Both designs utilize the unmodified source directly from [28] as components within the designed wrappers to communicate via the TIGRA interface.

4.6 Built-in Functions Custom Logic

For PicoRV32, multiplication is provided as a co-processor via the PCPI interface. The co-processor signals are very similar to TIGRA, requiring no modification to exist within TIGRA and is implemented by simply creating an instance of the multiplication and connecting it to the TIGRA signals. This enables multiplication within TIGRA and provides a direct test case against a known tightly coupled extension that completes the same operation.

For Rocket, we implement the functionality of many standard ALU functions: *ADD*, *XOR*,

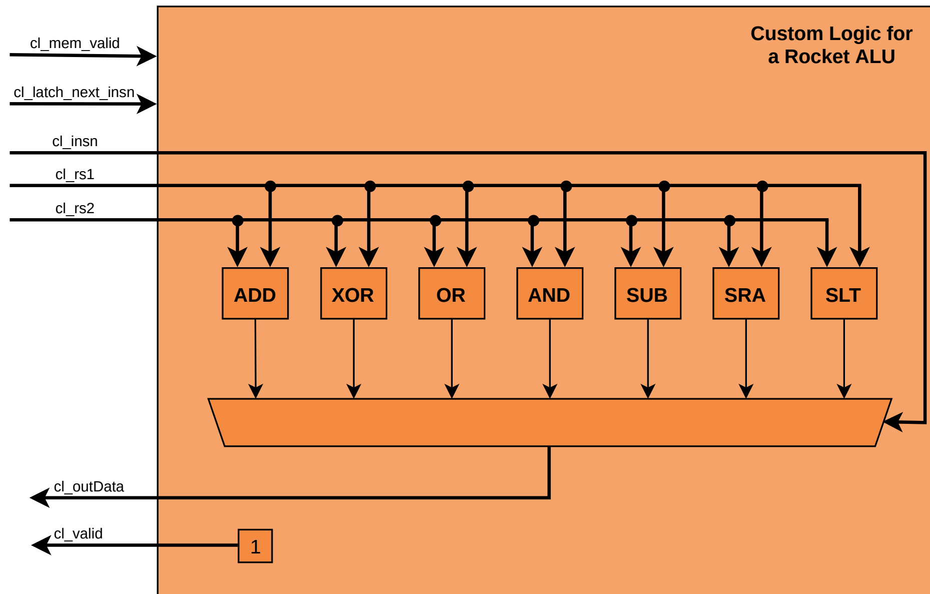


Figure 4.8: Rocket ALU custom logic block diagram.

OR, AND, SUB, SRA, and SLT. Each of these instructions complete in one clock cycle, making inclusion within TIGRA very simple. When the instructions complete in one clock cycle, there is no need to latch the instruction or the data, and cl_valid will always output a logic high. Figure 4.8 shows these design choices. Both cl_mem_valid and $cl_latch_next_insn$ are not used within the custom logic. Each functionality of the ALU receives both input operands and a multiplexer decodes cl_insn and outputs the correct result on $cl_outData$. The CPU will handle movement of data as necessary, making this inclusion straightforward.

In Taiga, the researcher created two smaller designs, one to test against the built in multiplier and another to test multiple instructions with arbitrary stall lengths and return values. The Taiga multiplier was copied into a custom logic wrapper to connect with the TIGRA interface and give a direct comparison against existing functionality. The other design returns known values for 8 different instructions and each requires an arbitrary amount of time to complete to verify Taiga stalls correctly when executing multiple concurrent instructions with varying timings. Table 4.2 shows the return value and execution length of each implemented TIGRA instruction.

Table 4.2: Custom instructions for arbitrary Taiga design

Instruction	Return Value	Stall Cycles
tigra_0	0	0
tigra_1	1	0
tigra_2	2	0
tigra_3	3	0
tigra_4	4	2
tigra_5	5	2
tigra_6	6	1
tigra_7	7	1

Table 4.3: Comparison of the lines of code edited for each design in Chapter 4.

Processor	**Lines Edited for TIGRA**
PicoRV32	32
Rocket Chip	180
Taiga	147
Custom Logic	**Lines Edited for Wrapper**
AES 128-bit	92
Posit Arithmetic - Full	80
Posit Arithmetic - Add Only	48
RoCC with AES 128-bit	107
Application Specific Taiga Unit	**Lines Edited for Taiga**
AES 128-bit	224
Posit Addition	189

4.7 Comparing the Programming Effort

Adding any tightly coupled accelerator or instruction set extension to each processor requires modification of the source code of the CPU to enable understanding of custom instructions, ensure multi-clock cycle instructions do not add any unnecessary latency, and verify the added functionality does not interfere with any of the default functionality of the CPU. Integrating the TIGRA interface to each processor requires developers complete the same modifications as described in Section 3.1.3, but the interface is left open and made generic for connecting with any custom logic design, as shown in Figure 3.4. Table 4.3 shows the number of lines edited to include the TIGRA processor to each of the processors and to write each custom logic wrapper use in testing via the TIGRA interface.

The lines edited for TIGRA indicate the final code product after including the interface with no connected logic. This programming effort represents a one-time cost required to modify the base processor and include the TIGRA interface. The lines edited for the wrapper detail the code required to connect an existing custom logic design via the TIGRA interface without modification to

the functionality of the downloaded source. As described in Section 3.1.3, adding a TIGRA interface to an existing processor is a similar process to adding any custom functionality. However, the TIGRA execution unit requires additional logic to ensure the interface remains simple and general for usage with connecting custom logic. By adding the total effort to add Taiga and include AES 128-bit and Posit addition from Table 4.3, 239 and 195 lines of code are required in total. When compared to the application specific designs, the full TIGRA with connected custom logic requires 15 and 6 additional lines of code for AES and Posit addition respectively. These extra lines of code are attributed to the logic required in the TIGRA execution unit to handle processor communication.

As a comparison, Table 4.3 also shows the number of lines required to add application specific execution units for AES 128-bit encryption and Posit Addition to the Taiga processor using the same unmodified sources described in Sections 4.4 and 4.5. These application specific execution units required the modification or addition of 224 and 189 lines of code, respectively, compared to the 92 and 48 required to create the custom logic wrapper to connect these designs via the TIGRA interface. By using TIGRA, adding AES 128-bit encryption and Posit addition to the Taiga processor required 41.1% and 25.4% less modified or added code when compared to adding the custom instructions directly.

The table also includes the lines of code required to include AES 128-bit encryption via the RoCC interface on Rocket Chip processors. The RoCC connected design requires 15 extra lines of code when compared to the equivalent TIGRA connected design. These extra lines of code are attributed to the extra complexity included with the RoCC interface, which allows users to communicate to the Rocket Chip FPU or directly access memory. The simplicity of the TIGRA interface allows users to create less logic to connect a custom logic design within an existing processor.

4.8 Summary

This chapter describes how TIGRA affects the instruction pipeline of the PicoRV32, Rocket Chip generator, and Taiga processors. We demonstrate during which stage of the instruction pipeline all TIGRA synchronization signals are set to enable custom logic integration to each of the generated processors. We also create custom logic for the different described test cases, detailing the necessary additions to existing cores to ensure proper functionality with the TIGRA interface. Each custom logic design explained is compatible with every TIGRA enabled processor and does not require any

modification due to the portability provided by the TIGRA interface.

Overall, while the initial TIGRA incorporation requires similar programming effort when compared to application specific custom logic, this is only a one-time cost incurred when integrating the interface. All future custom designs require significantly less code to incorporate and use a much more simple interface that does not require any further modifications to the CPU. This initial, one-time cost consumes most of the effort as developers must understand the intricacies of a given CPU and make modifications as necessary. The TIGRA interface removes the need to understand the inner workings of a processor and allows future additions to utilize less code to add new functionality. TIGRA provides a level of abstraction that eases the programming effort for all future designs.

Chapter 5

Simulation Results

In this chapter, we present the results of testing the TIGRA interface with the PicoRV32, Rocket Chip generator, and Taiga processors. We discuss the data from simulation for all processors, as well as initial FPGA hardware results on Amazon Web Services (AWS) F1 instances with the PicoRV32 processor. All timing diagrams are transcribed into a more readable format, as screenshots within the simulator do not represent the data very clearly as shown in Appendix C and Figures 5 and 6.

5.1 PicoRV32 Simulation Results

Similar to the design described in Section 4.1.2, each of the following test cases uses the simple PicoRV32 memory interface and instructions provided in Appendix A. The simulations are performed with Xilinx Vivado 2020.1 and target the XCVU9P-flgb2104-2-i FPGA for synthesis information.

5.1.1 AES

Figures 5.1 and 5.2 show encryption via custom logic connected via TIGRA to the PicoRV32 processor. The first figure demonstrates the final two writes into the custom logic, which write the 128-bit state value and initiate the execution stall. As expected, the first write shown completes in one clock cycle, and returns $0x0$ via *cl_outData* as all AES write instructions have no return value. Further, the CPU sets *cl_mem_valid* to high for one clock cycle, representing valid and

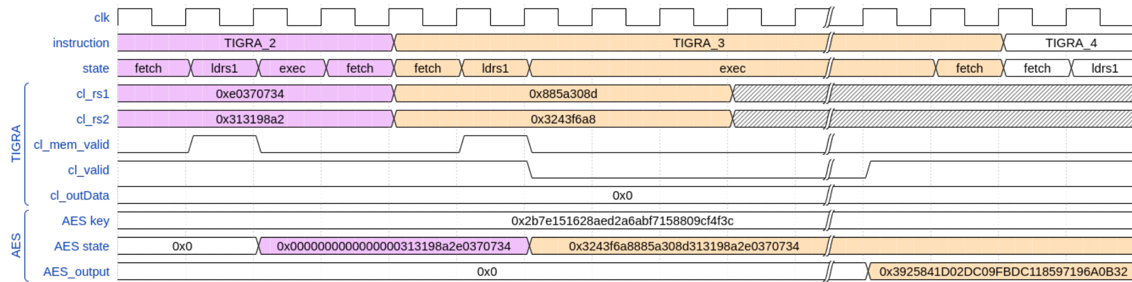


Figure 5.1: PicoRV32 with TIGRA and AES: The second two TIGRA instructions which store the AES state in the custom logic and complete the encryption. The break represents 15 clock cycles during which none of the shown signals change.

correct data on the cl_rs1 and cl_rs2 registers which is then latched into the custom logic on the next clock cycle. The instructions are color coded to demonstrate which instruction is responsible for changes within the custom logic, and we see the $TIGRA_2$ instruction write the lowest 64-bits of the state value and $TIGRA_3$ latch the upper 64-bits of the state and initiate a stall in the execute stage. The $TIGRA_3$ instruction sets cl_valid to a logic low, stalling the processor until encryption completes 21 clock cycles later. The $TIGRA_0$ and $TIGRA_1$ instructions can be seen in Appendix B as their execution mimics that shown in Figure 5.1.

Figure 5.2 shows the first two reads from the AES result, storing the lowest 64-bits of aes_output. The values from $cl_outData$ are color coordinated to clearly represent the flow of data from the custom logic and through the CPU. As TIGRA aims to follow the ALU flow as closely as possible, output data from the interface is latched into the native alu_out_q signal when cl_valid is high and the processor moves out of the execute stage of the pipeline. This signal is used by the PicoRV32 processor's ALU to store results and prepare for writes to the CPU registers, and this data flow is shown clearly in the timing diagram. The next clock cycle writes data back to the register used in the instruction, as specified on line 19 of Listing 3. The $TIGRA_5$ instruction then reads the next 32-bits of the AES result and writes this into the appropriate register. The remaining reads are not shown here as they follow the exact flow, and they can be found in Appendix B.

As shown, only one of the AES instruction generates a stall in the execution pipeline. This instruction stalls for 21 clock cycles in $exec$, spends 1 clock cycle in ld_rs1, and 2 clock cycles in $fetch$ for the initial decode and the final write back. Every other instruction spends a total of 4 clock cycles from decode to write back, with no added latency introduce between instruction execution. I total, PicoRV32 with TIGRA encrypts one 128-bit block via AES in a total of 52 clock

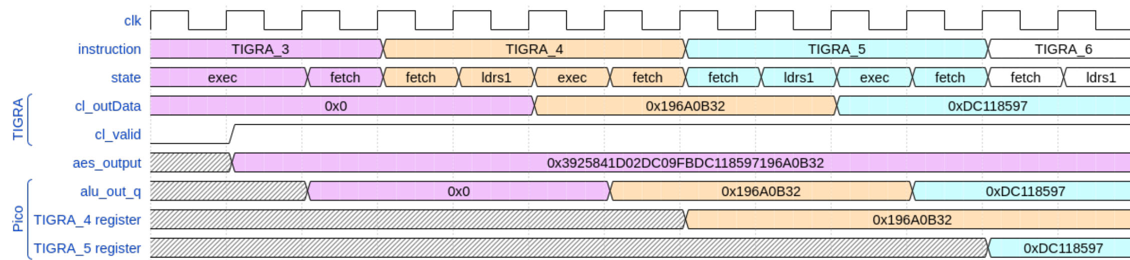


Figure 5.2: PicoRV32 with TIGRA and AES: The first two reads from the AES custom logic, showing the end of the stall from the last write.

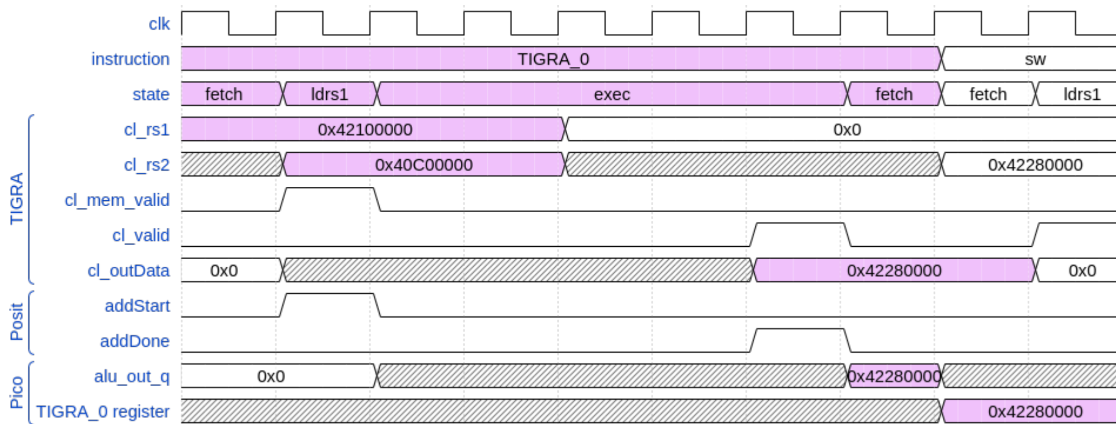


Figure 5.3: PicoRV32 with TIGRA and Posit Arithmetic: Posit addition showing the data flow from the custom logic all the way through register write-back.

cycles. By comparison, efficient computation of one 128-bit block in software on RISC-V requires on average 645 clock cycles to complete [45]. This exhibits the usefulness of using the TIGRA interface to connect custom logic into the CPU, as this simple design exhibits a 12.4 times speedup when encrypting 128-bit data with AES.

5.1.2 Posit Arithmetic

The results of posit addition, multiplication, and division are represented via figures 5.3, 5.4, and 5.5 respectively. For each diagram, the completed result is highlighted first on *cl_outData*, then on *alu_out_q*, and finally on the corresponding register. Posit addition and division each complete in the number of clock cycles expected as discussed in Section 4.5. Multiplication, however, completes in only four cycles instead of the six clock cycles detailed by the developer. Further inspection of the code in [28] shows the multiplication case may be mislabeled and does only require four clock cycles to complete.

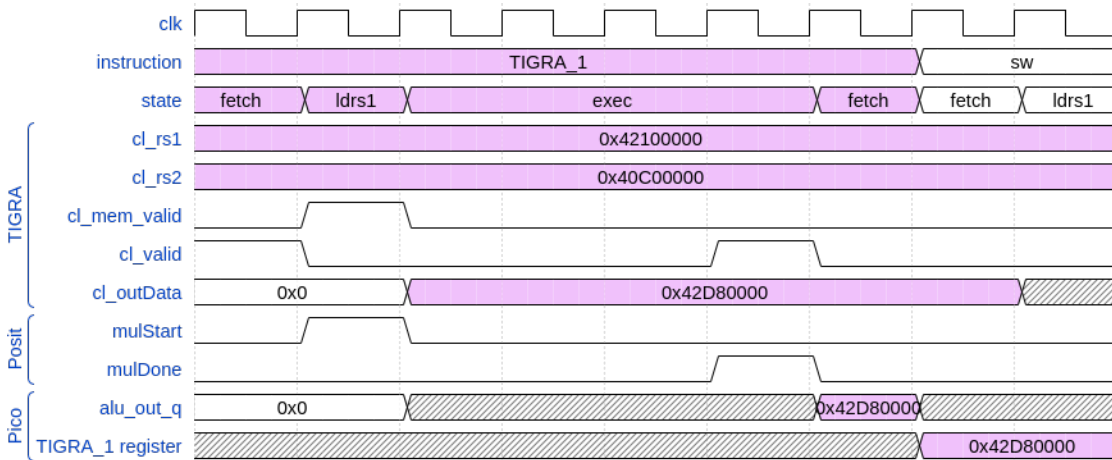


Figure 5.4: PicoRV32 with TIGRA and Posit Arithmetic: Posit multiplication showing the data flow from the custom logic all the way through register write-back.

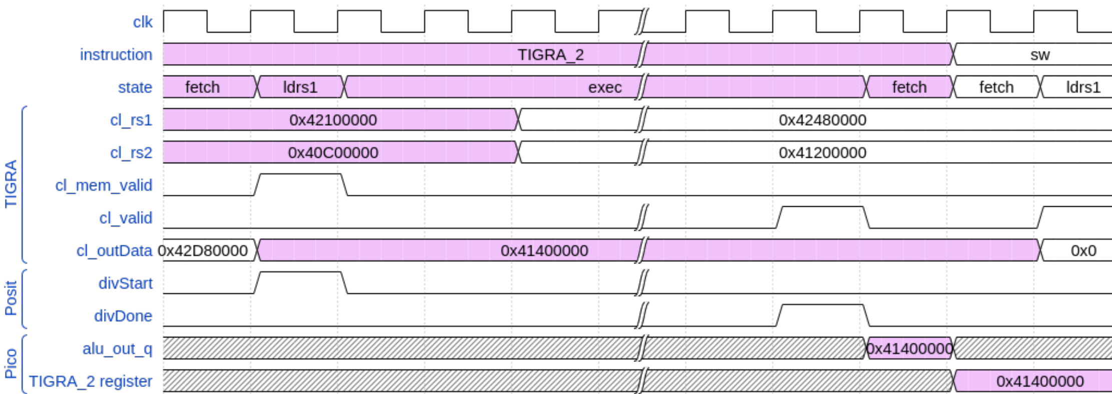


Figure 5.5: PicoRV32 with TIGRA and Posit Arithmetic: Posit division showing the data flow from the custom logic all the way through register write-back. The break represents 6 clock cycles during which none of the shown signals change.

The results for figures 5.4 and 5.5 appear on *cl_outData* well before each sets its respective done and *cl_valid* bit. This can be attributed to the fully-pipelined design of the PACoGen posit cores used and the choice of using the same values for testing each instruction. By design, each posit core runs freely and synchronizes results using the *opStart* and *opDone* bits shown in each figure, replacing *op* with the operation shown.. The logic-high value is simply passed through a shift register for the requisite number of clock cycles before writing a logic high on the output. Therefore, as the input values do not change between instruction calls, the result has already been computed by the PACoGen core, but the done signals are not set until the result can be guaranteed. Overall, these tests show a functional posit extension to RISC-V without requiring any modifications to the PACoGen code, validating the use of the TIGRA interface for the design.

5.1.3 Multiplication

The final simulation test case for PicoRV32 compares TIGRA with the packaged co-processor interface, PCPI. Figures 5.6 and 5.7 show these results. Both diagrams achieve the same result, $0x0054$ or 84 in decimal, and successfully write data into register file. However, using TIGRA with the same multiplication logic produces the result 1 clock cycle quicker than when used with PCPI. Figure 5.6 shows the *mul_start* signal, which is built into the multiplier logic for PicoRV32, is driven high 2 clock cycles after the $fetch$ state, compared to 3 in Figure 5.7 when using PCPI. Both designs complete the multiplication in the same time from *mul_start* to *cl_valid* and *pcpi_ready*, respectively. The extra clock cycle of latency can be attributed to extra synchronization required as PCPI does not follow the standard ALU execution flow. This further demonstrates the benefits of using the TIGRA interface to connect custom logic accelerators.

5.2 PicoRV32 Hardware Results

We test hardware functionality on the AWS F1 cloud service, which hosts Xilinx XCVU9P-flgb2104-2-i FPGAs. The current iteration of the design only allows for a simple $fpga_pci_peek$ into a user defined memory space. This returns a single value that can be compared with results from simulation, and does not allow for any complex testing of the device. As such, visual results are currently omitted until this design is improved.

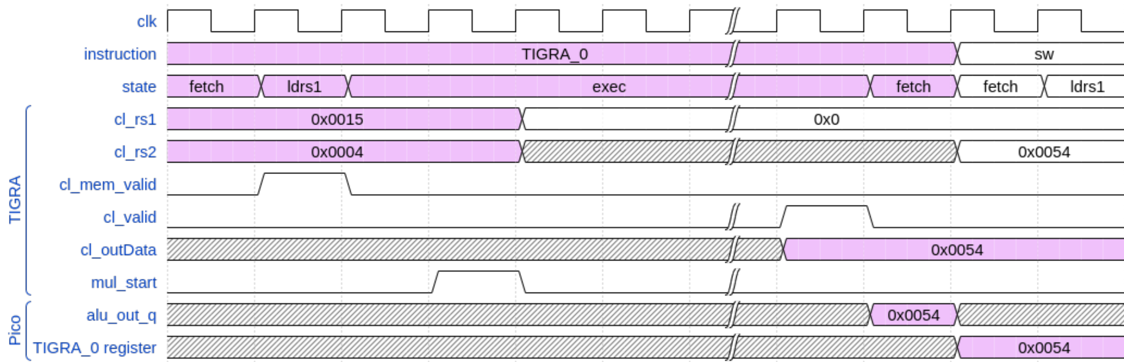


Figure 5.6: PicoRV32 with TIGRA and multiplication: Multiplication, the break represents 30 clock cycles.

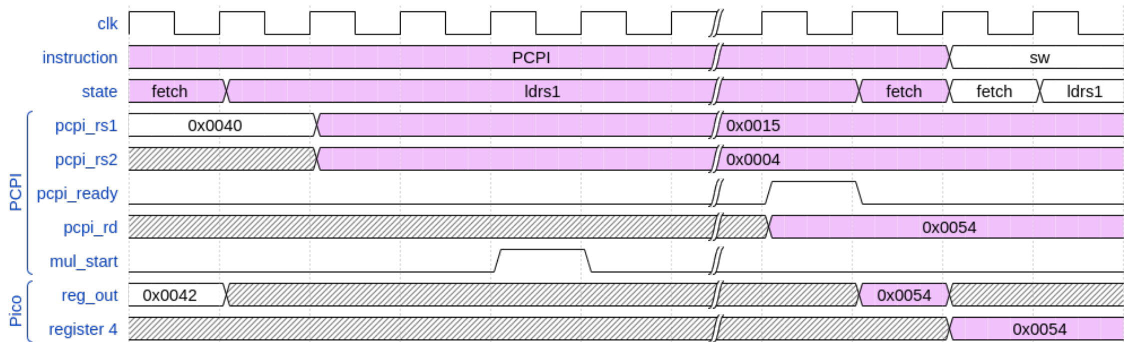


Figure 5.7: PicoRV32 with PCPI and multiplication: Multiplication, the break represents 31 clock cycles.

5.3 Rocket Chip Results

Developers cannot easily perform hardware verification of Rocket Chip generated processors but are provided a robust set of simulation utilities to test all designs. [34] details the necessary steps to test Rocket Chip with TIGRA using the packaged Verilator simulator and receive the results shown in this section. The timing diagrams in this section are color coded to help represent data flow through the Rocket Chip processor and represent which instructions are responsible for changes in the custom logic and within the processor.

5.3.1 AES

To test the Rocket AES design, we compiled a simple C program, shown in Listing 6 that initializes the AES key and state values and then calls all 8 TIGRA instructions consecutively to complete encryption and store the result. Figure 5.8 shows the two writes to the AES key and the first write to the AES state, all instructions that do not require a stall. The instructions perform the operations listed in Table 4.1 and we coordinate the data flow for each instruction by color within the figure. For each instruction entering the execute stage of the pipeline, the Rocket Chip processor sets the cl_mem_valid signal to high to indicate data may be latched on the next clock edge. The first instruction, $TIGRA_0$ latches data into the lower 64 bits of the key register and is arbitrarily set to return the value 1. The CPU latches this return value into the mem_reg_wdata signal as the instruction moves to the memory stage, the same signal used by the Rocket Chip ALU to save these results. The data is then copied into the wb_reg_wdata signal and finally into a CPU register as the instruction moves to the writeback stage and then finally retires. The remaining two instructions, $TIGRA_1$ and $TIGRA_2$ latch the upper 64 bits of the key and lower 64 bits of the state, respectively, and follow the same data flow.

While these first three instructions are programmed to execute consecutively, as shown in Listing 6, the compiler reorders instructions based on data dependencies. Listing 11 shows the order of instructions after compilation, with data setup surrounding the first three TIGRA instructions to prepare the values used for the AES key and state. This reordering explains the gap between the initial writes in Figure 5.8.

Figure 5.9 shows TIGRA instructions 3-7. The first instruction on the diagram writes the upper 64 bits of the AES state and initiates a stall in the Rocket Chip processor to complete the

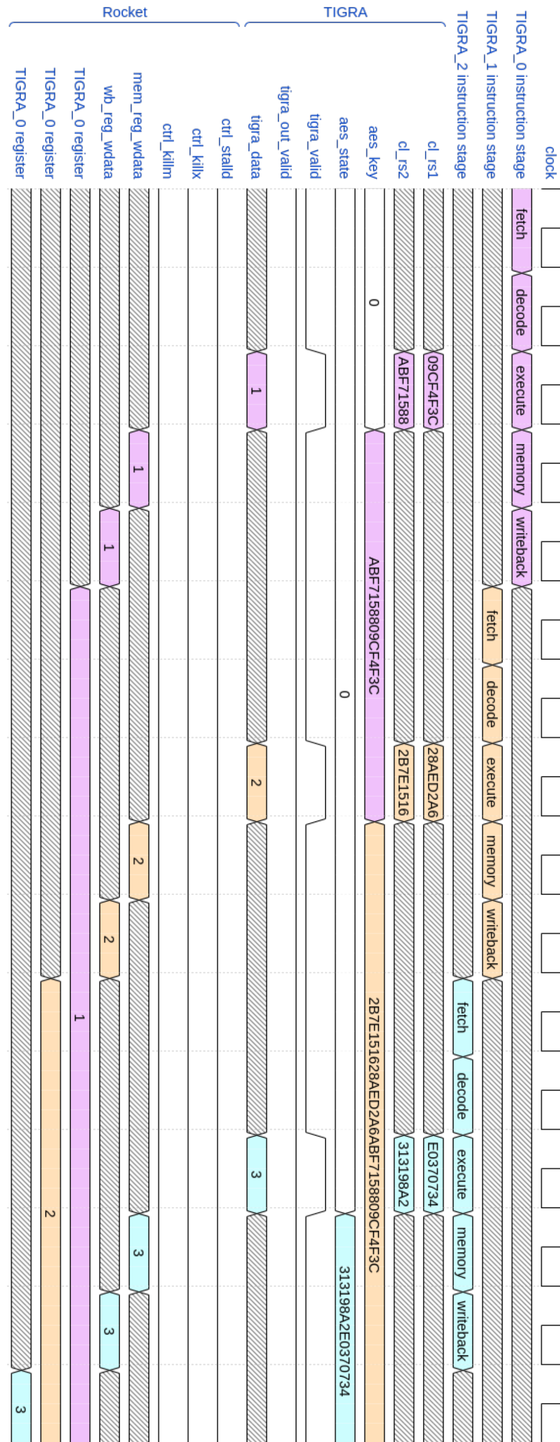


Figure 5.8: Rocket Chip with AES via TIGRA: The first three AES writes.

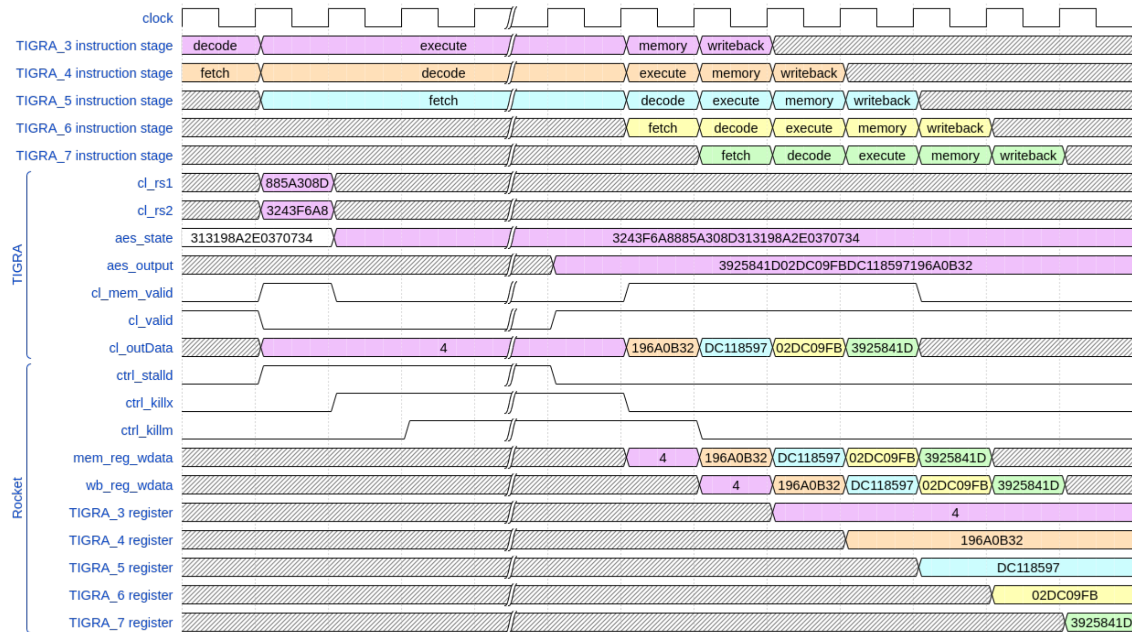


Figure 5.9: Rocket Chip with TIGRA: The remaining instructions to complete AES encryption in a Rocket Chip generated processor. The break in the diagram represents 18 clock cycles during which none of the signals shown change.

encryption. The custom logic stalls the CPU for 22 total clock cycles by setting cl_valid to a logic low. This value is immediately copied to the $ctrl_stalld$ signal, which stalls the $TIGRA_4$ and $TIGRA_5$ instructions in decode and fetch, respectively. In the subsequent two clock cycles, $ctrl_killx$ and $ctrl_killm$ are set to logic high to stall in the memory and writeback stages of the pipeline as they are vacated. Once encryption completes, the custom logic sets the cl_valid signal and the internal Rocket Chip control signals are cleared in reverse order. The processor then continues as normal and TIGRA instructions 4-7 read the encrypted result into CPU registers.

5.3.2 Posit Addition

Figure 5.10 shows the full execution of a posit addition instruction, $TIGRA_0$. Mirroring the stall shown for AES, this instruction pushes $ctrl_stalld$ high, followed by $ctrl_killx$ and then $ctrl_killm$. Once complete, the custom logic sends the data from add_result within the addition PACoGen core to $cl_outData$ to be latched by the Rocket Chip processor as the instruction moves to the memory stage. The stalls are then lifted to enable the CPU to resume normal execution. The output data is copied to mem_reg_wdata as the instruction enters the memory stage. This is then

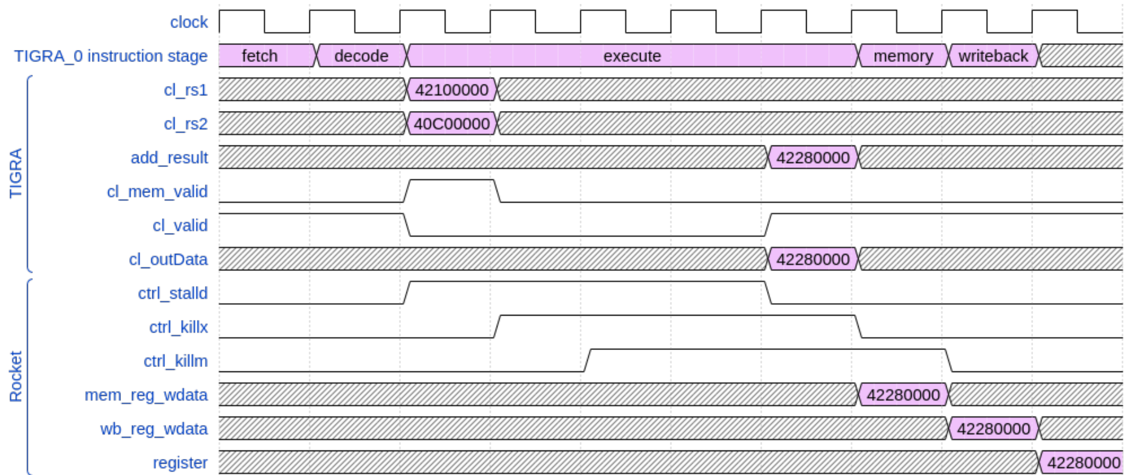


Figure 5.10: Rocket Chip with TIGRA: Posit addition of 18 and 3, $0x42100000$ and $0x40C00000$, providing the resultant value 21, $0x42280000$.

written to wb_reg_wdata for write back into the register file as the instruction enters the last stage of the pipeline and finally is latched into a register as the instruction retires.

5.3.3 Rocket ALU

The final Rocket test case implements a series of single clock cycle instructions as defined in Section 4.2.1. Figure 5.11 shows the results from the XOR instruction using the RISC-V tool's instruction verification capabilities. When using the instruction verification libraries, chosen instructions are executed very and the expected results are compared with those returned by the given instruction. The chosen commands will not execute consecutively due to this comparison, explaining the break in the diagram. Figure 5.11 shows the output appearing on $cl_outData$ immediately within the execute stage, as expected, before the instruction moves on to the next stage of the pipeline. The results are copied through mem_reg_wdata and wb_reg_wdata before being copied into a register as the instruction retires. We do not show the other functions from Figure 4.8 as their execution is the same, simply with different results.

5.3.4 RoCC Results and Comparison to TIGRA

Figure 5.12 shows the first AES instruction executing within the rocket custom coprocessor. Accelerators and designs connected with this interface do not begin executing until after the instruction exits the writeback stage of the normal Rocket Chip pipeline, as shown by the $rocc_cmd_valid$

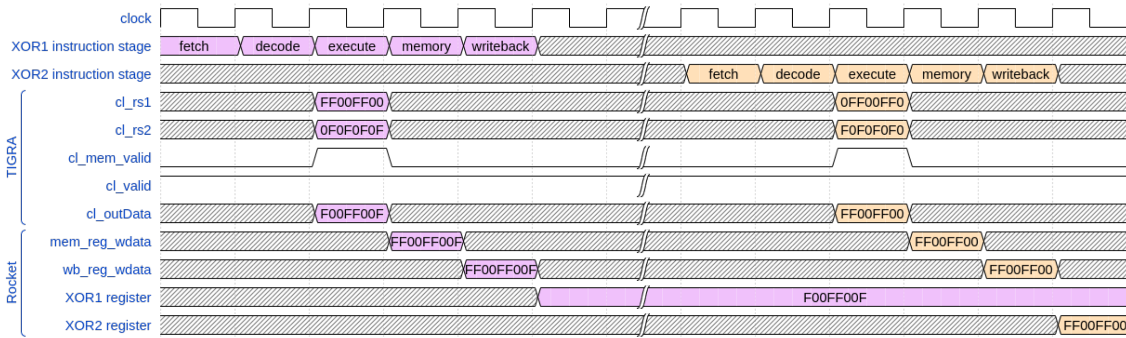


Figure 5.11: Rocket Chip with TIGRA and Rocket Chip ALU. Results from two XOR instructions executing within the instruction verification environment provided by the RISC-V Tools.

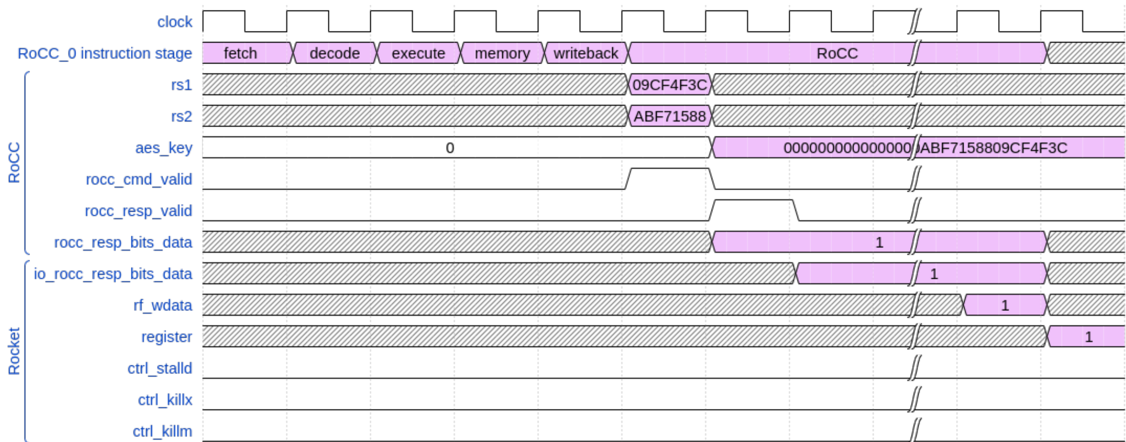


Figure 5.12: Rocket Chip with RoCC and AES. The first AES write instruction that writes the lowest 32-bits of the AES key to the custom logic. The break represents 7 clock cycles during which none of the shown signals change.

signal going to a logic high as the instruction leaves the writeback stage. This guarantees that any RoCC enabled instruction incurs at least a 4 clock cycle penalty after being fetched. To further this, RoCC instructions can not write back to the register file at any predictable time depending on the compiler optimizations and reordering, as the RoCC interface must wait for the writeback bus to clear or stall the processor to force a write. This wait introduces a variable latency for each instruction executed through the RoCC interface and is represented in Figure 5.12. The result of the $RoCC_0$ instruction shown is latched to a register 15 cycles after the fetch stage, or 10 clock cycles after the custom logic returned a high value on the $rocc_resp_valid$ signal. This figure also shows that RoCC uses different signals to write data back to the register file, $io_rocc_resp_bits_data$, which will require overhead in logic to route responses correctly.

RoCC instructions behave more predictably when compiler reordering allows them to exe-

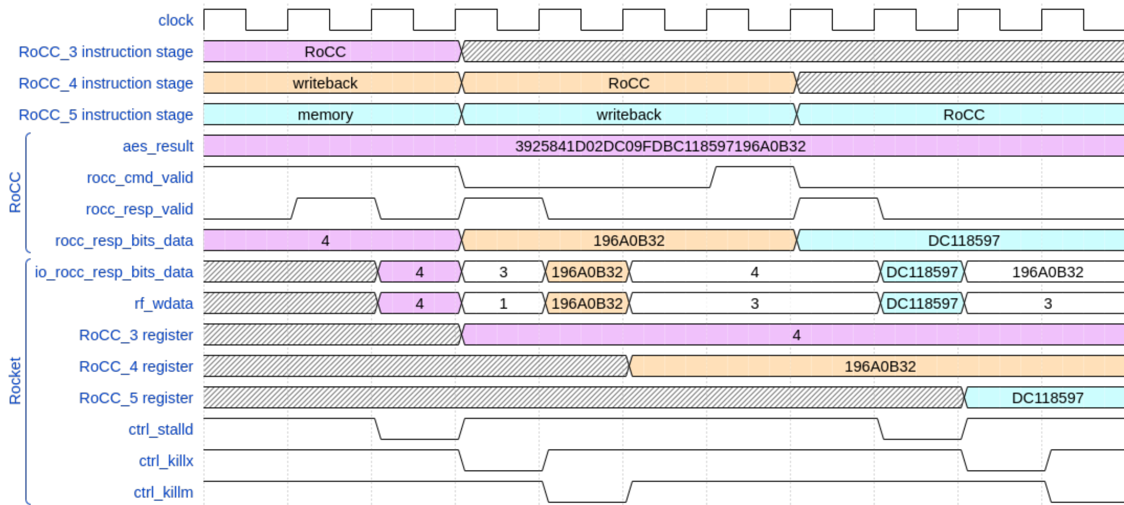


Figure 5.13: Rocket Chip with RoCC and AES. The first two AES read instructions that return the lowest 64-bits of the encrypted result.

cute consecutively, which removes the contention when RoCC attempts to write back to the register file. Figure 5.13 shows two AES read instructions near the end of the C program that execute in sequence after the *Rocc_3* instruction which completed the encryption. With no remaining instructions available to mask the latency of the interface, each invocation of RoCC stalls the Rocket Chip processor pipeline, adding additional overhead for subsequent calls to the attached accelerator. RoCC instructions fetched before, but executing after a multi-clock cycle stall, such as *RoCC_4* and *RoCC_5* suffer the largest penalty as they stall in the Rocket Chip pipeline while the *RoCC_3* instruction executes.

Table 5.1 shows the comparison of RoCC instructions against the TIGRA equivalents. Instructions 3, 4, and 5 require the longest time for both the RoCC and TIGRA interfaces, as both designs stall the Rocket Chip processor when encrypting the data in instruction 3. Each corresponding RoCC instruction incurred extra overhead when compared to the TIGRA equivalent. For one iteration of AES-128, RoCC instructions added 6.875 clock cycles per instruction on average when compared to the corresponding TIGRA implementation.

5.4 Taiga Processor Results

Similar to the Rocket Chip designs, the Taiga modifications are tested using the packaged Verilator simulator to receive the results shown in this section. The timing diagrams are color coded

Table 5.1: Latency of TIGRA enabled Rocket compared to RoCC when executing AES instructions. Each value counts the number of clock cycles between instruction fetch and when the result is stored in a register.

AES Instr.	Latency TIGRA	Latency RoCC	Difference
0	4	15	11
1	4	12	8
2	4	7	3
3	26	29	3
4	26	30	4
5	26	33	7
6	4	12	8
7	4	15	11

to help represent data flow through the processor and represent which instructions are responsible for changes in the custom logic and within the processor.

5.4.1 AES

We tested AES encryption on the Taiga processor using the same C program described in Section 5.3. Figure 5.14 shows the results of the first two AES instructions writing the full AES key to the custom logic. Each of these instructions complete in one clock cycle, and return arbitrary values to follow data flow through the Taiga processor. As shown, the custom logic maintains a logic high on *cl_valid*, and the TIGRA execution unit sets *tigra_unit_done* signal to high during the *load/store* stage of the pipeline. The Taiga processor's writeback multiplexer acknowledges this instruction as complete by signaling high on the *load_store_ack* signal, and data is placed on the *retiring_data* signal before being written to a register as the instruction retires. Similar to the Rocket Chip processor, these instructions do not execute consecutively due to compiler reordering.

Figure 5.15 shows the results of the last five TIGRA instructions described in Table 4.1 executing within the Taiga processor with the AES custom hardware attached. As AES encryption begins in the *load/store* stage of the $TIGRA_3$ instruction, Taiga sets the *tr_unit_stall* signal as high until the custom logic outputs a high output valid signal. This signal shows the processor has stalled as the execution unit needed for the next instruction is unavailable, preventing new TIGRA instructions from entering the custom hardware and preventing new instructions from continuing out of the fetch stage until the current instruction completes. Further, the timing diagram shows that the Taiga processor stalls the currently executing instruction in the *load/store* stage of the pipeline until data is available for writeback. This stall occurs in the *load/store* stage to allow other instructions

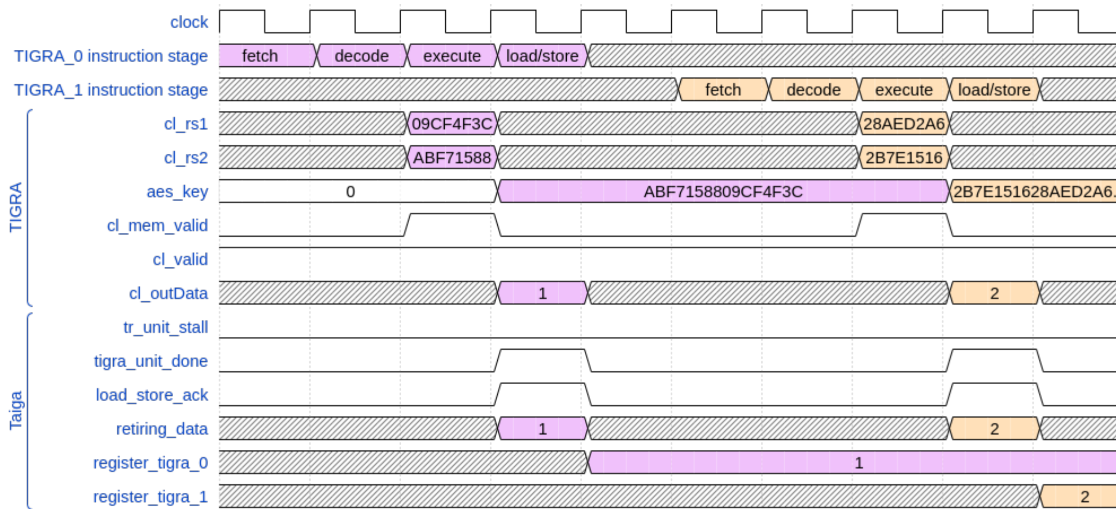


Figure 5.14: Simulation of the first two TIGRA instructions when using custom logic with AES. These instructions both complete within one clock cycle.

to enter available execution units, and the writeback multiplexer will stall the CPU if necessary. Once the TIGRA execution unit sets *tigra_unit_done* to logic high, Taiga's writeback multiplexer immediately responds with an acknowledgement on the *load_store_ack* signal and supplies the correct output to the *retiring_data* signal. The processor writes the result to the appropriate register on the next clock cycle, following the same flow as built-in Taiga execution units.

The remaining instructions complete as the Taiga processor issues new requests to the TIGRA execution unit, and all instructions complete consecutively following the same pathway as the $TIGRA_3$ instruction. It is important to note that both the $TIGRA_6$ and $TIGRA_7$ instructions are listed as in the *fetch* stage of the pipeline while the processor stalls to complete the encryption. Taiga's decoupled design allows up to 8 in flight instructions by default. The processor fetches upcoming instructions and schedules them to complete, even when current instructions require more than one clock cycle to complete or will use the same execution unit of a former instruction. This design allows multiple TIGRA instructions to occupy the *fetch* stage, but only one instruction may occupy all subsequent stages as each uses the same hardware.

5.4.2 Posit Addition

Figure 5.16 shows the completion of posit addition 18 and 3, *0x42100000* and *0x40C00000* respectively, followed by the addition of 21 and 3, with *0x42280000* representing 21. As there

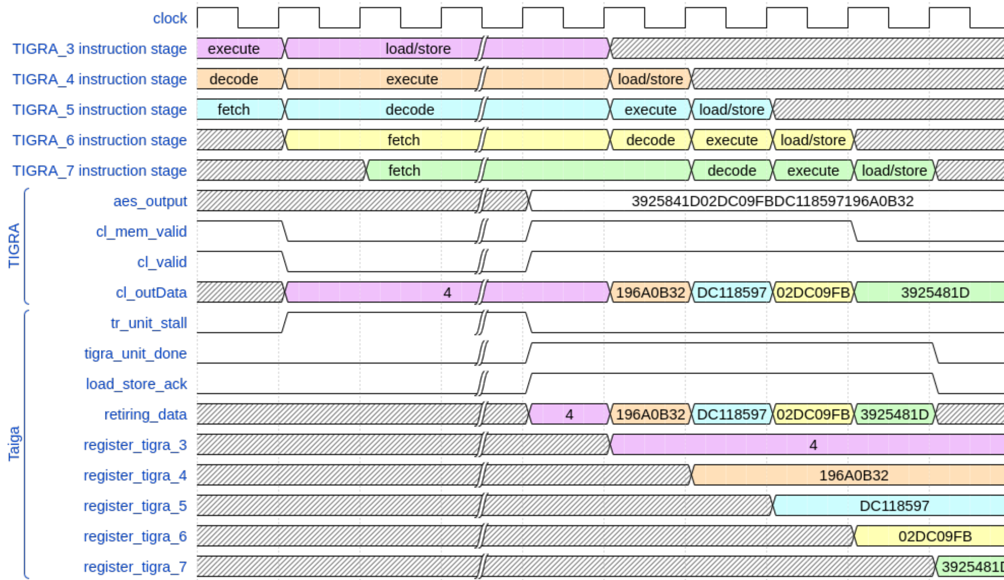


Figure 5.15: Simulation showing the latter five TIGRA instructions executing on the Taiga processor with AES hardware connected. The break represents 18 clock cycles during which none of the shown signals change.

are no other instructions left in the program, the Taiga processor does not set the *tr_unit_stall* signal. The host CPU recognizes the explicit data dependencies defined by the compiler and the consecutive TIGRA instructions do not cause an unnecessary stall in the pipeline. The AES example sets *tr_unit_stall* as multiple instructions were fetched that could not enter the *decode* or *execute* stages of the Taiga pipeline until the previous instructions completed. Both these results and those shown by the AES example prove that the TIGRA interface adds no latency to the Taiga processor and the processor stalls only when neeeded.

5.4.3 Taiga Testing

The instructions shown in Table 4.2 are completed via a simple C program that calls all 8 TIGRA instructions with inline assembly as shown in Listing 8. Figure 5.17 shows the latter 4 instructions, those that require arbitrary stalls, completing in the Taiga processor. Each instruction immediately supplies the return value, but this value is not latched by the processor until the custom logic sets *cl_valid* to a logic high. *cl_rs*1 and *cl_rs*2 are not shown on the timing diagram here, as the custom logic ignores these inputs and sets the outputs based on the input instruction. The C program provided, however, does set the inputs to each instruction by overwriting the *temp*0 and

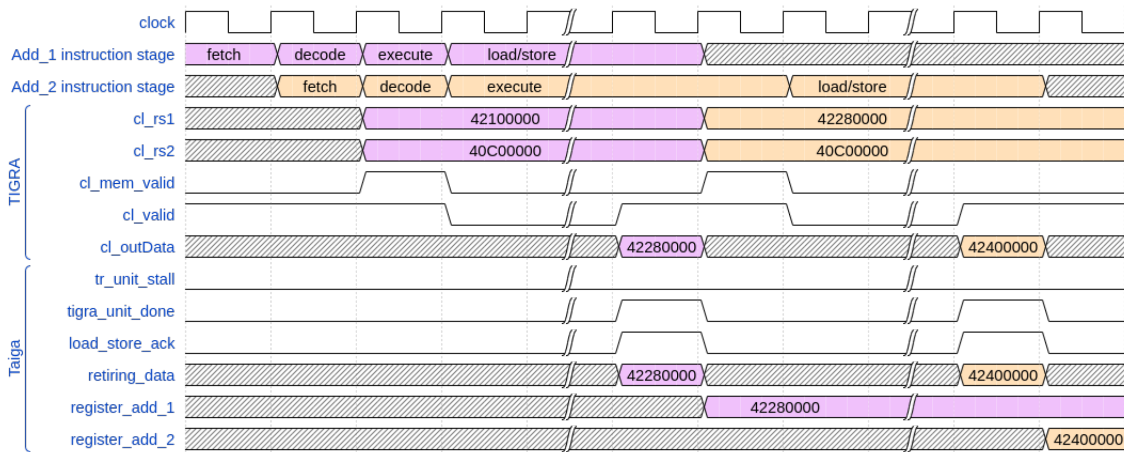


Figure 5.16: Simulation of two consecutive *tigra_ 0* instructions with Posit addition connected via the TIGRA interface on the Taiga processor. Each break represents 2 clock cycles during which none of the shown signals change.

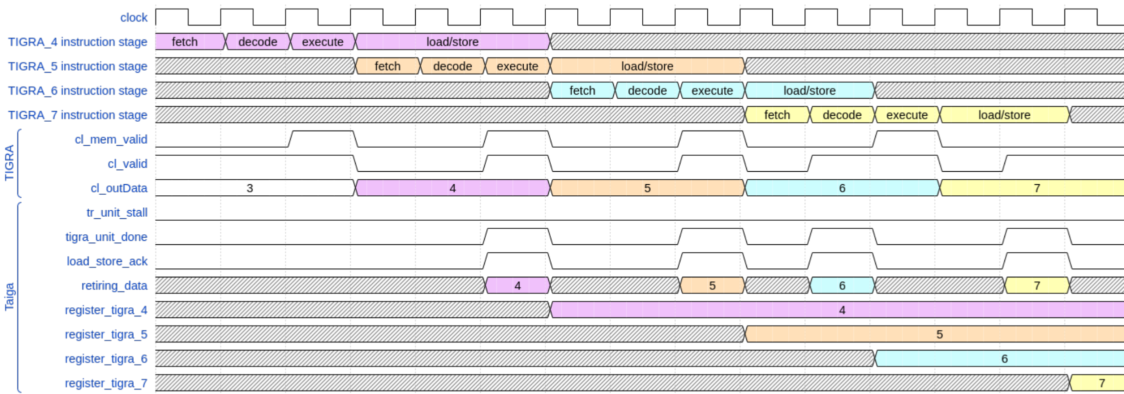


Figure 5.17: Simulation of the TIGRA instructions 4-7 when using custom logic with arbitrary return values and stalls. These instructions signal complete at varying times even though the data is immediately available.

*temp*1 values shown in Listing 8 immediately before calling the next instruction. This extraneous overwrite causes the TIGRA instructions to be fetched 2 clock cycles apart, the gap shown in the timing diagram. As none of the arbitrary delays exceed 2 clock cycles, this custom logic does not cause the Taiga processor to stall due to the parallel execution unit design.

Listing 7 provides the code used to test multiplication within the Taiga processor via the TIGRA interface. Figure 5.18 shows the first 7 multiplications performed by the custom logic in this design. Each multiplication immediately provides a result when the instruction enters the *load/store* stage of the pipeline, and the results are written through the processor pipeline as expected. This timing diagram further shows 5 TIGRA instructions executing concurrently, while the test code calls

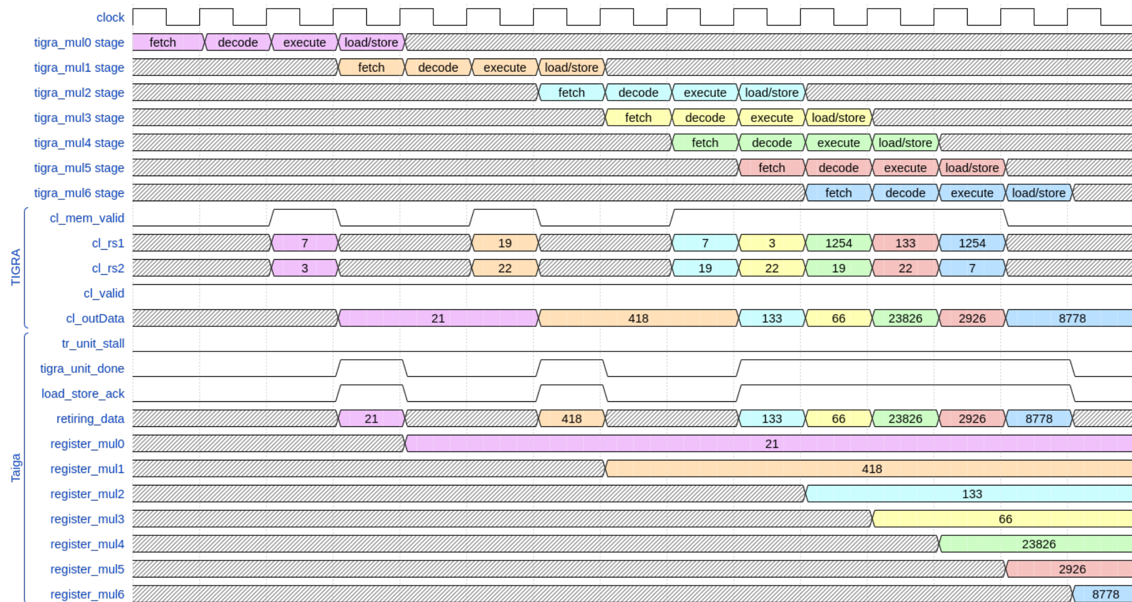


Figure 5.18: Simulation of the first 7 multiplications to test the multiplication custom logic.

at most 3 TIGRA instructions in a row. As the results of the computations on lines 45 and 54 of Listing 7 are not required until line 89, the compiler reordered the instructions to more efficiently use processor resources when computing multiple sums. Figure 6 in Appendix C shows the waveform capture of the execution of this program.

It may be noted that multiplication in TIGRA does not require 2 clock cycles when compared to the default Taiga implementation. The Taiga multiplier uses a clock cycle to sign extend the inputs to provide a correct result for any number supplied. The multiplication design implemented within TIGRA removes this extra clock cycle to complete multiplication immediately upon latching the input data from cl_rs1 and cl_rs2

5.5 Summary

The results from this section show that incorporating TIGRA with a CPU can allow users to enable different functionality through usage of custom logic. In the PicoRV32, Rocket Chip, and Taiga processors, we successfully demonstrate that TIGRA introduces zero latency when using custom logic accelerators and enables developers to build custom instruction extensions that operate in the same pipeline as the ALU of the CPU. We also show that interfaces that require extra synchronization logic or diverge from the standard ALU execution flow, such as PCPI or ROCC,

may incur latency penalties similar to loosely coupled accelerators.

Chapter 6

Partial Reconfiguration

Partial Reconfiguration is the process of compiling and reprogramming a subset of an FPGA design instead of completing the process on the entire project. This introduces modularity to a design by allowing developers to create multiple compatible custom modules and dynamically reprogram the reconfigurable region at runtime while the base logic continues to function. Partial reconfiguration requires extra logic to perform the reprogramming of only specific regions of the FPGA as a trade-off for the flexibility provided. Figure 6.1 shows a high level block diagram of the partial reconfiguration design using the TIGRA interface. Partial reconfiguration also allows developers to recompile each module separately while leaving the main processor unchanged.

Both Intel's Partial Reconfiguration (PR) [22] and AMD/Xilinx's Dynamic Function eXchange (DFX) [6] supply tools to enable partial reconfiguration within designs. In this chapter, we discuss how DFX is leveraged to enhance the functionality of the TIGRA interface.

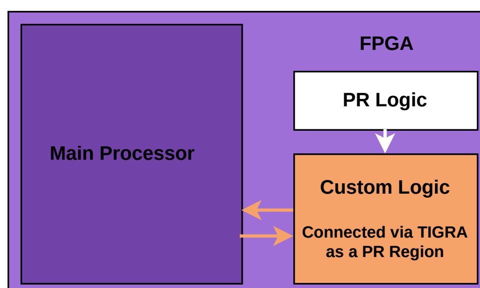


Figure 6.1: Abstract block diagram for partial reconfiguration. The Custom Logic block is the reconfigurable region in orange. The PR logic block, shown in white, represents the extra logic required to partially reprogram the FPGA.

6.1 Taiga Partial Reconfiguration

Setting up Partial Reconfiguration with DFX requires some modifications to the original design in order to be compatible with the partial reconfiguration process. The original design of TIGRA utilized System Verilog interfaces and structures to simplify design of the TIGRA execution unit and mimic existing execution units within Taiga. Listing 9 shows the module declaration, interface declaration, and *tigra_inputs_t* struct definition. Interfaces are not compatible with DFX, requiring that we specify each signal individually or "flatten" the module declaration as shown in Listing 10. Further, when building Taiga for usage on an FPGA, the authors provide scripts to automatically set up the required memory and wrappers to run on the Xilinx ZedBoard. The process automatically packages the Taiga CPU as a block diagram design, meaning the design is static and therefore incompatible with DFX. We then needed to enable DFX on the CPU before packaging as a block diagram, removing the capability of using the automated scripts to generate the full processor system for use on the Xilinx ZedBoard platform. As we are no longer using the automated scripts, we changed the target device to the Ultrascale+ XCVU9P-FLGB2104-2-i FPGA to provide more FPGA resources when setting up partial reconfiguration, allowing larger custom logic designs to exist in the reprogrammable region.

We first enabled partial reconfiguration on the TIGRA interface within the Taiga processor using AMD/Xilinx's DFX within Vivado. We then follow the guides in [54] and [52] to enable DFX for the attached custom logic and created a profile for each function described in Chapter 4. When enabling partial reconfiguration within a design, developers must select a region within the FPGA to designate for partial reconfiguration. This restricts the FPGA resources such as memory, look-up tables (LUTs), and slices for usage only within custom modules for partial reconfiguration. Figure 6.2 shows a small portion of the XCVU9P device, with the custom logic block highlighted and programmed with the arbitrary design described in Section 4.6. This region covers over 50,000 LUTs, 100,000 registers, and over 200 RAM blocks which supplies more than enough resources for the designs tested in this paper.

6.1.1 Results

One of the goals for implementing partial reconfiguration is to reduce compilation time when modifying and testing designs. Each child design for the reconfigurable region is compiled separately,

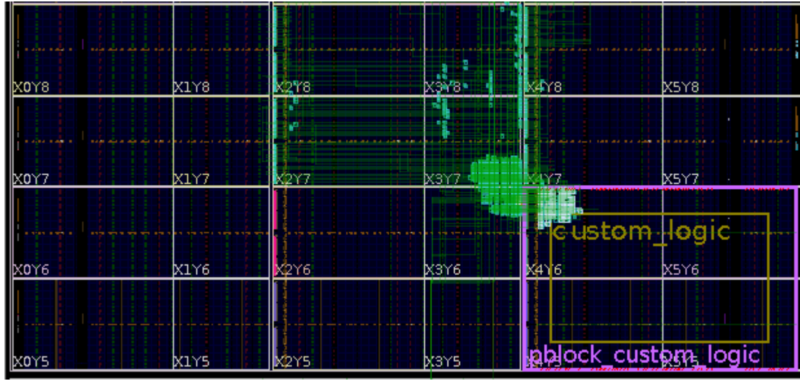


Figure 6.2: Screenshot showing a portion of the XCVU9P device, including the custom logic block which is designated as the partially reconfigurable region.

Table 6.1: Synthesis and Implementation timing results of the full Taiga processor with each test case connected via TIGRA and the same designs using Xilinx DFX.

	Synthesis Time (s)	**Implementation Time (s)**
Complete Arbitrary	56.6	327
Complete AES	73.0	346.2
Complete Multiply	56.8	318.8
Complete Posit	61.0	323.8
Base Design	96.6	
Parent Module (Arbitrary)	46.0	317.6
Child (AES)	64.0	289.2
Child (Multiply)	48.2	262.6
Child (Posit Addition)	50.6	258.4

or out-of-context, meaning smaller designs are placed and routed for each module. Out-of-context compilation also ensures that no optimizations occur when between the CPU and the custom logic, which could remove unused signals or attempt to re-route some designs outside of the locked region.

To test DFX with TIGRA, we ran synthesis and implementation on the Taiga processor with AES connected via TIGRA with the design using partial reconfiguration. Synthesis is the process of generating the hardware required to complete a given design, and implementation is the process of placing and routing the synthesized design on the designated FPGA. Table 6.1 shows the average timing results of 5 runs of synthesis and implementation on the Taiga processor for the complete designs of all test cases and the partial reconfiguration results. The "Base Design" refers to the non-reprogrammable region of the design and does not have an implementation time as this is completed with the parent of the reconfigurable region.

The parent design is the first module and all children re-use parts of the implementation

Table 6.2: Comparison of synthesis and implementation times of a complete design compared to the same design using Xilinx DFX.

Synthesis		
	Speedup	**Percentage Time Saved**
Arbitrary	1.23	18.7%
AES	1.14	12.3%
Multiply	1.18	15.1%
Posit	1.21	17.0%
Implementation		
	Speedup	**Percentage Time Saved**
Arbitrary	1.03	2.3%
AES	1.20	16.5%
Multiply	1.21	17.6%
Posit	1.25	20.2%

to further reduce the time of these stages of the compilation process. Specifically, the parent implementation run creates the FPGA routing and logic required to implement partial reconfiguration and this is locked into the overall design for re-use when building more custom modules. The parent design is usually chosen as the most complicated design that will be implemented within the reprogrammable region, as this module will require the most logic and routing. However, a bug in the Xilinx Vivado software [5] causes the implementation stage to fail and prevents manual selection of the parent and child modules. The fix forces the software to automatically select the arbitrary test design described in Section 4.6 and Table 4.2 as the parent implementation, which can affect the implementation time of the child designs which are more complicated, such as the AES custom logic.

Table 6.2 shows the speedup gained in the synthesis and implementation stages of compilation for each test case. As expected, each design's synthesis run completes in less time than the respective run for an entire compilation when switching out the custom logic. The implementation run of the arbitrary design only saves 2.3% of the time compared to the complete design, but this places and routes the locked logic required for partial reconfiguration. On average, partial reconfiguration saves 15.78% of the time for each synthesis run and 14.15% of the time for each implementation run. If we remove the parent module from the implementation average, this result changes to 18.1% of the time saved on average for each new child module designed.

6.2 Summary

This section demonstrates the capabilities of using partial reconfiguration with TIGRA to speed up the hardware design process for custom logic accelerators or instruction set extensions. The results in Table 6.2 show up to 20.2% time saved in compilation to add new instructions to a processor. Developers using a TIGRA enabled processor can compile modules with any custom logic design and reprogram only the reserved region of the FPGA at runtime, improving the flexibility of the processor. The modularity introduced by using the TIGRA interface allows for smaller custom logic designs as only the hardware needed at a given time is used, which can reduce area usage and further improve maximum clock speed and reduce the power required during execution.

Chapter 7

Conclusions and Future Work

This dissertation aims to bridge the gap between loosely coupled and tightly coupled accelerators and simplify the process of adding custom logic to existing processors. By giving developers the ability to quickly swap out and more easily create tightly coupled accelerators, designs can benefit from a zero latency interface and with partial reconfiguration can be chosen and switched at run-time. Incorporating the TIGRA interface with an existing RISC-V processor requires a non-recurring programming effort, as described in Section 4.7, to ensure all pre-existing processor functionality remains consistent and confirm that the TIGRA interface does not introduce overhead for connected custom logic designs. All further usage of the interface requires no modifications of the underlying CPU architecture or instruction pipeline, and developers can focus on building custom logic and then communicate via the simplified TIGRA interface. Additionally, all custom logic designs are completely portable across different TIGRA enabled processors and require no modification to connect to a new CPU.

Chapter 3 provides insight into incorporating TIGRA into an existing processor pipeline, designing custom logic for use within TIGRA, and how to use the custom instructions with the RISC-V compiler. Chapter 4 gives implementation details of three very different RISC-V CPU's that were TIGRA enabled as proof of concept designs. The simple PicoRV32 processor with TIGRA was used to prove that TIGRA adds no latency overhead when comparing to the built-in coprocessor interface, PCPI. Adding TIGRA to the Rocket Chip generator demonstrates that the TIGRA interface works on fully pipelined processors, can be implemented in the Chisel environment, and reduces latency when compared to the Rocket Chip Custom Coprocessor Interface, RoCC. Custom designs connected

via RoCC have more functionality, FPU and memory access, when compared to the TIGRA interface, but this added functionality adds latency and programming effort penalties. The modifications in the Taiga processor, which utilizes decoupled execution units, show promise toward the inclusion of TIGRA in an out-of-order processor, such as the Berkeley Out-of-Order Machine [57]. Taiga required that the TIGRA design manage tags and return data in the correct order similar to out-of-order processors, all while maintaining the simplicity of the user facing interface. Future work will explore the inclusion of TIGRA in out-of-order processors.

Coupled with Chapter 5, we show that TIGRA can work on architectures with differing complexity and pipeline lengths to provide many benefits in the fields of HPC and hardware design by removing the latency associated with hardware based accelerators. TIGRA can further be extended to allow multiple instances of custom logic and execute them concurrently, by managing currently executing instructions through the addition of tags to the interface. While adding tags and managing multiple in flight instructions may require more initial work when adding TIGRA to a processor, the user side development of the custom logic will require minimal additional programming effort to return the appropriate tag as each instruction retires from the custom logic.

Further, building TIGRA with Partial Reconfiguration shows the capabilities of using the interface in a real-world processor. Based on the case studies, developers can save, on average, 18.1% of the compilation time during the design process to more quickly prototype hardware-based accelerators or instruction set extensions. These modifications can then be applied at runtime by reprogramming the custom logic region connected via the TIGRA interface, providing further flexibility within a TIGRA enabled processor by allowing the user to swap between execution units while the processor continues to run. PR designs also improve the modularity of designs and may reduce resource usage of connected custom logic, which can lead to reduced power usage of running designs and improve overall clock-speed of the custom logic.

Chapters 4 and 5 show the design and results of encryption and posit arithmetic connected via TIGRA as two future looking extensions for RISC-V processors, but any custom design can be connected via the interface. Further work is being completed to test machine learning inference via custom logic connected by the TIGRA interface. Further, Figure 3.5 shows that many accelerators can be used within the TIGRA interface by adding minimal logic to decode the input instructions and select the correct output to return to the CPU. These capabilities demonstrate the potential of using the TIGRA interface for multiple applications and in differing environments. Developers can

couple all necessary logic for an application in one custom logic design, and partial reconfiguration can allow users to swap the connected hardware at runtime to customize the processor to any given application.

Overall, adding the TIGRA interface to a processor has been shown to reduce the programming effort by reducing the lines of code edited or added to include an accelerator by up to 75% in the case studies. The simplified interface allows developers to include existing custom logic, such as the PACoGen posit arithmetic cores or AES encryption hardware, with minimal extra design when using existing logic. Further, the TIGRA interface is shown to execute 6.875 clock cycles per instruction faster than the Rocket Custom Coprocessor, a commonly used interface to include custom hardware into the Rocket Chip processor. The TIGRA interface works well with the parameterizable designs of the Rocket Chip generator and Taiga processors, working as an extension on these processors that may be enabled during compilation time. This dissertation proves the benefits of including a TIGRA interface on future processors to improve the modularity and ease of customizability of RISC-V processors without any latency overhead.

Appendices

Appendix A Code Listings

This appendix shows some of the code required to implement TIGRA on different processors or test the designs described in Chapter 4. These listings provide supplemental information to describe how the work in this dissertation was carried out.

```
217 // Write Request
218 logic        wr_active;
219 logic [31:0] wr_addr;
220
221 always_ff @(posedge clk_main_a0)
222   if (!rst_main_n_sync) begin
223      wr_active <= 0;
224      wr_addr   <= 0;
225   end
226   else begin
227      wr_active <=  wr_active && bvalid  && bready ? 1'b0     :
228                   ~wr_active && awvalid           ? 1'b1     :
229                                                     wr_active;
230      wr_addr <= awvalid && ~wr_active ? awaddr : wr_addr     ;
231   end
232
233 assign awready = ~wr_active;
234 assign wready  =  wr_active && wvalid;
235
236 // Write Response
237 always_ff @(posedge clk_main_a0)
238   if (!rst_main_n_sync)
239     bvalid <= 0;
240   else
241     bvalid <=  bvalid &&  bready           ? 1'b0  :
242                          ~bvalid && wready ? 1'b1  :
243                                              bvalid;
244 assign bresp = 0;
245
246 // Read Request
247 always_ff @(posedge clk_main_a0)
248    if (!rst_main_n_sync) begin
249       arvalid_q <= 0;
250       araddr_q  <= 0;
251    end
252    else begin
253       arvalid_q <= arvalid;
254       araddr_q  <= arvalid ? araddr : araddr_q;
255    end
```

```
256
257 assign arready = !arvalid_q && !rvalid;
258
259 // Read Response
260 always_ff @(posedge clk_main_a0)
261    if (!rst_main_n_sync)
262    begin
263       rvalid <= 0;
264       rdata  <= 0;
265       rresp  <= 0;
266    end
267    else if (rvalid && rready)
268    begin
269       rvalid <= 0;
270       rdata  <= 0;
271       rresp  <= 0;
272    end
273    else if (arvalid_q)
274    begin
275       rvalid <= 1;
276       rdata  <= (araddr_q == `BASE_REG_ADDR             ) ? memory[32]:
277                 (araddr_q == (`BASE_REG_ADDR+4)         ) ? {memory[33]            }:
278                 (araddr_q == (`BASE_REG_ADDR+8)         ) ? {memory[34]            }:
279                                                        `UNIMPLEMENTED_REG_VALUE       ;
280       rresp  <= 0;
281    end
282
283 //-------------------------------------------------
284 // PicoRV32 with POSIT CL
285 //-------------------------------------------------
286
287 picorv32_TIGRA #(
288 ) uut (
289         .clk                    (clk_main_a0            ),
290         .resetn                 (rst_main_n_sync        ),
291         .trap                   (trap                   ),
292         .mem_valid              (mem_valid              ),
293         .mem_instr              (mem_instr              ),
294         .mem_ready              (mem_ready              ),
295         .mem_addr               (mem_addr               ),
296         .mem_wdata              (mem_wdata              ),
297         .mem_wstrb              (mem_wstrb              ),
298         .mem_rdata              (mem_rdata              ),
299
300         //TIGRA Interface
301         .cl_insn                (cl_insn                ),
```

```
302         .cl_rs1                 (cl_rs1                 ),
303         .cl_rs2                 (cl_rs2                 ),
304         .cl_mem_valid           (cl_mem_valid           ),
305         .cl_latch_next_insn     (cl_latch_next_insn     ),
306         .cl_outData             (cl_outData             ),
307         .cl_valid               (cl_valid               )
308 );
309 
310 pico_cl_posits pico_posits(
311         .clk                    (clk_main_a0            ),
312         .resetn                 (rst_main_n_sync        ),
313         .insn_from_pico         (cl_insn                ),
314         .data1                  (cl_rs1                 ),
315         .data2                  (cl_rs2                 ),
316         .mem_valid_from_pico    (cl_mem_valid           ),
317         .latch_next_insn        (cl_latch_next_insn     ),
318         .outData                (cl_outData             ),
319         .out_valid              (cl_valid               )
320 );
321 
322 
323 always_ff @(posedge clk_main_a0) begin
324         mem_ready <= 0;
325 
326         if (!rst_main_n_sync) begin                                     // Reset
327                 memory[0]  = 32'b000000010010_00000_000_00001_0010011;  // addi x1,18
328                 memory[1]  = 32'b000000000011_00000_000_00010_0010011;  // addi x2,3
329                 memory[2]  = 32'b0000000_00010_00001_000_00011_0110011; // add x3,x1,x2
330 
331                 //Load from memory[16], posit value 18
332                 memory[3]  = 32'b000001000000_00000_010_00100_0000011;  // lw x4,64(x0)
333                 //Load from memory[17], posit value 3
334                 memory[4]  = 32'b000001000100_00000_010_00101_0000011;  // lw x5,68(x0)
335                 memory[5]  = 32'b0000000_00101_00100_000_00110_0101011; // posit add x6,x4,x5
336 
337                 //Store sum in memory[32]
338                 memory[6]  = 32'b0000100_00110_00000_010_00000_0100011; // sw x6,128(x0)
339                 memory[7]  = 32'b0000000_00101_00100_001_00111_0101011; // posit mult.  x7,x4,x5
340                 memory[8]  = 32'b0000000_00101_00100_010_01000_0101011; // posit divide x8,x4,x5
341 
342                 //Store product in memory[33]
343                 memory[9]  = 32'b0000100_00111_00000_010_00100_0100011; // sw x7,132(x0)
344                 //Store quotient in memory[34]
345                 memory[10] = 32'b0000100_01000_00000_010_01000_0100011; // sw x8,136(x0)
346 
347                 memory[11] = 32'b000000000000_00000_000_00000_0010011;  // addi x0,0 (nop)
```

```
348                 memory[12] = 32'b000000000000_00000_000_00000_0010011;  // addi x0,0 (nop)
349                 memory[12] = 32'b000000000000_00000_000_00000_0010011;  // addi x0,0 (nop)
350                 memory[13] = 32'h ff5ff06f;                             // jal back to first nop
351
352                 memory[16] = 32'b01000010000100000000000000000000;      // 18 as posit<32,6>
353                 memory[17] = 32'b01000000110000000000000000000000;      // 3 as posit<32,6>
354         end
355         else if (mem_valid && !mem_ready) begin
356                 if(mem_addr < 1024) begin
357                         mem_ready <= 1;
358                         mem_rdata <= memory[mem_addr >> 2];
359
360                         if(mem_wstrb[0]) memory[mem_addr >> 2][ 7: 0] <= mem_wdata[ 7: 0];
361                         if(mem_wstrb[1]) memory[mem_addr >> 2][15: 8] <= mem_wdata[15: 8];
362                         if(mem_wstrb[2]) memory[mem_addr >> 2][23:16] <= mem_wdata[23:16];
363                         if(mem_wstrb[3]) memory[mem_addr >> 2][31:24] <= mem_wdata[31:24];
364                 end
365         end
366 end
```

Listing 1: The code required to add a TIGRA enabled PicoRV32 processor to an AWS F1 wrapper with posit custom logic.

```
memory[0]  = 32'b000000010010_00000_000_00001_0010011;  //addi x1,18
memory[1]  = 32'b000000000011_00000_000_00010_0010011;  //addi x2,3
memory[2]  = 32'b0000000_00010_00001_000_00011_0110011; //add x3,x1,x2
memory[3]  = 32'b000001000000_00000_010_00100_0000011;  //lw x4,64(x0) (x4 = mem[16])
memory[4]  = 32'b000001000100_00000_010_00101_0000011;  //lw x5,68(x0) (x5 = mem[17])

memory[5]  = 32'b0000000_00101_00100_000_00110_0101011; //posit add x6,x4,x5

//Store posit sum in mem[32]
memory[6]  = 32'b0000100_00110_00000_010_00000_0100011; //sw x6,128(x0)  (mem[32] = x6)
memory[7]  = 32'b0000000_00101_00100_001_00111_0101011; //posit multiply x7,x4,x5
memory[8]  = 32'b0000000_00101_00100_010_01000_0101011; //posit divide x8,x4,x5

//Store posit product in mem[33]
memory[9]  = 32'b0000100_00111_00000_010_00100_0100011; //sw x7,132(x0)  (mem[33] = x7)
//Store posit quotient in mem[34]
memory[10] = 32'b0000100_01000_00000_010_01000_0100011; //sw x8,136(x0)  (mem[34] = x8)

memory[11] = 32'b000000000000_00000_000_00000_0010011;  //addi x0,0 (nop)
memory[12] = 32'b000000000000_00000_000_00000_0010011;  //addi x0,0 (nop)
memory[12] = 32'b000000000000_00000_000_00000_0010011;  //addi x0,0 (nop)
memory[13] = 32'h ff5ff06f;                             //jal back to first nop

memory[16] = 32'b01000010000100000000000000000000;      //18 in posit<32,6> format
memory[17] = 32'b01000000110000000000000000000000;      //3 in posit<32,6> format
```

Listing 2: The instructions used to test the Posit custom logic within the PicoRV32 with TIGRA.

```
memory[ 0] = 32'b000000101010_00000_000_00001_0010011;  //addi x1,42
memory[ 1] = 32'b000000010001_00000_000_00010_0010011;  //addi x2,17
memory[ 2] = 32'b0000000_00010_00001_000_00010_0110011; //add x2,x1,x2
memory[ 3] = 32'b000010000000_00000_010_01000_0000011;  //lw x8,128(x0) (x8 = mem[32])
memory[ 4] = 32'b000010000100_00000_010_01001_0000011;  //lw x9,132(x0)
memory[ 5] = 32'b000010001000_00000_010_01010_0000011;  //lw x10,136(x0)
memory[ 6] = 32'b000010001100_00000_010_01011_0000011;  //lw x11,140(x0)
memory[ 7] = 32'b000010010000_00000_010_01100_0000011;  //lw x12,144(x0)
memory[ 8] = 32'b000010010100_00000_010_01101_0000011;  //lw x13,148(x0)
memory[ 9] = 32'b000010011000_00000_010_01110_0000011;  //lw x14,152(x0)
memory[10] = 32'b000010011100_00000_010_01111_0000011;  //lw x15,156(x0)

memory[11] = 32'b000010100000_00000_000_00011_0010011;  //addi x3,160

memory[12] = 32'b0000000_01001_01000_000_00000_0101011; //wr_key_lo x8,x9
memory[13] = 32'b0000000_01011_01010_001_00000_0101011; //wr_key_hi x10,x11
memory[14] = 32'b0000000_01101_01100_010_00000_0101011; //wr_state_lo x12,x13
memory[15] = 32'b0000000_01111_01110_011_00000_0101011; //wr_state_hi x14,x15
memory[16] = 32'b0000000_00000_00000_100_10000_0101011; //rd_res_lo x16
memory[17] = 32'b0000000_00010_00001_000_00010_0110011; //add x2,x1,x2
memory[18] = 32'b0000000_00000_00000_101_10001_0101011; //rd_res_midlo x17
memory[19] = 32'b0000000_00000_00000_110_10010_0101011; //rd_res_midhi x18
memory[20] = 32'b0000000_00000_00000_111_10011_0101011; //rd_res_hi x19

memory[21] = 32'b0000000_10000_00011_010_00000_0100011; //sw x16,0(x3)
memory[22] = 32'b0000000_10001_00011_010_00100_0100011; //sw x17,4(x3)
memory[23] = 32'b0000000_10010_00011_010_01000_0100011; //sw x16,8(x3)
memory[24] = 32'b0000000_10011_00011_010_01100_0100011; //sw x19,12(x3)

memory[25] = 32'b000000000000_00000_000_00000_0010011;  //addi x0,0 (nop)
memory[26] = 32'b000000000000_00000_000_00000_0010011;  //addi x0,0 (nop)
memory[27] = 32'b000000000000_00000_000_00000_0010011;  //addi x0,0 (nop)
memory[28] = 32'h ff5ff06f;                             //jal back to first nop

memory[32] = 32'h09cf4f3c;     //key lo
memory[33] = 32'habf71588;     //key midlo
memory[34] = 32'h28aed2a6;     //key midhi
memory[35] = 32'h2b7e1516;     //key hi
memory[36] = 32'he0370734;     //state lo
memory[37] = 32'h313198a2;     //state midlo
memory[38] = 32'h885a308d;     //state midhi
memory[39] = 32'h3243f6a8;     //state hi
```

Listing 3: The instructions used to test the AES custom logic within the PicoRV32 with TIGRA.

```
memory[0] = 32'b000000010101_00000_000_00001_0010011;  //addi x1,21
memory[1] = 32'b000000000101_00000_000_00010_0010011;  //addi x2,5
memory[2] = 32'b0000000_00010_00001_000_00011_0110011; //add x3,x1,x2
memory[3] = 32'b0000001_00010_00001_000_00100_0101011; //mul x4,x1,x2

memory[4]  = 32'b0000010_00100_00000_010_00000_0100011;//sw x4,64(x0)  (mem[16] = x4)
memory[5] = 32'b0000000_00011_00001_000_00011_0110011; //add x3,x1,x3
memory[6] = 32'b0000000_00011_00010_000_00011_0110011; //add x3,x2,x3

memory[7] = 32'b000000000000_00000_000_00000_0010011;  //nop (addi x0,0)
memory[8] = 32'b000000000000_00000_000_00000_0010011;  //nop (addi x0,0)
memory[9] = 32'b000000000000_00000_000_00000_0010011;  //nop (addi x0,0)
memory[10] = 32'h ff5ff06f;                            //jal back to first nop
```

Listing 4: The instructions used to test multiplication within the PicoRV32 with TIGRA.

```
memory[0] = 32'b000000010101_00000_000_00001_0010011;  //addi x1,21
memory[1] = 32'b000000000101_00000_000_00010_0010011;  //addi x2,5
memory[2] = 32'b0000000_00010_00001_000_00011_0110011; //add x3,x1,x2
memory[3] = 32'b0000001_00010_00001_000_00100_0110011; //mul x4,x1,x2  21*5 = 105

memory[4]  = 32'b0000010_00100_00000_010_00000_0100011;//sw x4,64(x0)  (mem[16] = x4)
memory[5] = 32'b0000000_00011_00001_000_00011_0110011; //add x3,x1,x3
memory[6] = 32'b0000000_00011_00010_000_00011_0110011; //add x3,x2,x3

memory[7] = 32'b000000000000_00000_000_00000_0010011;  //nop (addi x0,0)
memory[8] = 32'b000000000000_00000_000_00000_0010011;  //nop (addi x0,0)
memory[9] = 32'b000000000000_00000_000_00000_0010011;  //nop (addi x0,0)
memory[10] = 32'h ff5ff06f;                            //jal back to first nop
```

Listing 5: The instructions used to test multiplication within the PicoRV32 using PCPI.

```
int main( int argc, char* argv[] )
{
    unsigned int key0 = 0x09cf4f3c;
    unsigned int key1 = 0xabf71588;
    unsigned int key2 = 0x28aed2a6;
    unsigned int key3 = 0x2b7e1516;
    unsigned int state0 = 0xe0370734;
    unsigned int state1 = 0x313198a2;
    unsigned int state2 = 0x885a308d;
    unsigned int state3 = 0x3243f6a8;

    int tig0, tig1, tig2, tig3, tig4, tig5, tig6, tig7;
    asm volatile ( "tigra1_0 %0, %1, %2;\n"
                    :"=r" (tig0)
                    :"r" (key0), "r" (key1)
                    :
                );
    asm volatile ( "tigra1_1 %0, %1, %2;\n"
                    :"=r" (tig1)
                    :"r" (key2), "r" (key3)
                    :
                );
    asm volatile ( "tigra1_2 %0, %1, %2;\n"
                    :"=r" (tig2)
                    :"r" (state0), "r" (state1)
                    :
                );
    asm volatile ( "tigra1_3 %0, %1, %2;\n"
                    :"=r" (tig3)
                    :"r" (state2), "r" (state3)
                    :
                );
    asm volatile ( "tigra1_4 %0, %1, %2;\n"
                    :"=r" (tig4)
                    :"r" (tig0), "r" (tig1)
                    :
                );
    asm volatile ( "tigra1_5 %0, %1, %2;\n"
                    :"=r" (tig5)
                    :"r" (tig2), "r" (tig3)
                    :
                );
    asm volatile ( "tigra1_6 %0, %1, %2;\n"
                    :"=r" (tig6)
                    :"r" (tig4), "r" (tig5)
                    :
```

```
47                  );
48     asm volatile ( "tigra1_7 %0, %1, %2;\n"
49                     :"=r" (tig7)
50                     :"r" (tig6), "r" (tig0)
51                     :
52                  );
53
54
55   return 0;
56 }
```

Listing 6: C code used to test AES encryption in the Rocket Chip and Taiga processors via the TIGRA interface.

```
1  int main(void) {
2      //Platform Initialization
3      platform_init ();
4
5      //Records cycle and instruction counts
6      start_profiling ();
7
8      //Set up variables
9      unsigned int key0 = 7;
10     unsigned int key1 = 3;
11     unsigned int key2 = 19;
12     unsigned int key3 = 22;
13     unsigned int res0;
14     unsigned int res1;
15     unsigned int res2;
16     unsigned int res3;
17
18     //Perform first multiplication: tigra_mul0
19     asm volatile ( "tigra0 %0, %1, %2;\n"
20         :"=r" (res0)
21         :"r" (key0), "r" (key1)
22         :
23      );
24
25     //Multiply using the result of the TIGRA instruction to ensure
26     //the operation is not optimized out by the compiler.
27     unsigned int temp0 = res0 * key0;
28
29     //tigra_mul1
30     asm volatile ( "tigra0 %0, %1, %2;\n"
31         :"=r" (res1)
32         :"r" (key2), "r" (key3)
```

```
33          :
34      );
35
36     unsigned int temp1 = res1 * key1;
37
38     //tigra_mul2
39     asm volatile ( "tigra0 %0, %1, %2;\n"
40         :"=r" (res2)
41         :"r" (key0), "r" (key2)
42         :
43      );
44
45     unsigned int temp2 = res2 * key2;
46
47     //tigra_mul3
48     asm volatile ( "tigra0 %0, %1, %2;\n"
49         :"=r" (res3)
50         :"r" (key1), "r" (key3)
51         :
52      );
53
54     unsigned int temp3 = res3 * key3;
55
56     //tigra_mul4
57     asm volatile ( "tigra0 %0, %1, %2;\n"
58         :"=r" (res0)
59         :"r" (temp1), "r" (key2)
60         :
61      );
62
63     //tigra_mul5
64     asm volatile ( "tigra0 %0, %1, %2;\n"
65         :"=r" (res1)
66         :"r" (res2), "r" (key3)
67         :
68      );
69
70     //tigra_mul6
71     asm volatile ( "tigra0 %0, %1, %2;\n"
72         :"=r" (res2)
73         :"r" (temp1), "r" (key0)
74         :
75      );
76
77     unsigned int temp4 = res2 * key1;
78
```

```
79      //tigra_mul7
80      asm volatile ( "tigra0 %0, %1, %2;\n"
81          :"=r" (res3)
82          :"r" (temp4), "r" (key2)
83          :
84       );
85
86      unsigned int temp6 = res0 * 5;
87      unsigned int temp5 = res1 * 7;
88
89      unsigned int all_temp = temp0+temp1+temp2+temp3+temp4+temp5+temp6;
90
91      unsigned int tigra_tmp = 2;
92      //tigra_mul8
93      asm volatile ( "tigra0 %0, %1, %2;\n"
94          :"=r" (tigra_tmp)
95          :"r" (all_temp), "r" (tigra_tmp)
96          :
97       );
98
99      unsigned int testing;
100
101     /tigra_mul9
102     asm volatile ( "tigra0 %0, %1, %2;\n"
103         :"=r" (testing)
104         :"r" (tigra_tmp), "r" (key1)
105         :
106      );
107
108
109     //Records cycle and instruction counts
110     //Prints summary stats for the application
111     end_profiling ();
112
113     return 0;
```

Listing 7: C code used to test multiplication in the Taiga processor via the TIGRA interface.

```
1 int main(void) {
2     int temp0 = 1;
3     int temp1 = 3;
4     int tig0, tig1, tig2, tig3, tig4, tig5, tig6, tig7;
5
6     asm volatile ( "tigra0 %0, %1, %2;\n"
7                     :"=r" (tig0)
8                     :"r" (temp0), "r" (temp1)
```

```
9                      :
10                 );
11
12     temp0 = 2;
13     temp1 = 6;
14     asm volatile ( "tigra1 %0, %1, %2;\n"
15                    :"=r" (tig1)
16                    :"r" (temp0), "r" (temp1)
17                    :
18                 );
19
20     temp0 = 4;
21     temp1 = 12;
22     asm volatile ( "tigra2 %0, %1, %2;\n"
23                    :"=r" (tig2)
24                    :"r" (temp0), "r" (temp1)
25                    :
26                 );
27
28     temp0 = 8;
29     temp1 = 24;
30     asm volatile ( "tigra3 %0, %1, %2;\n"
31                    :"=r" (tig3)
32                    :"r" (temp0), "r" (temp1)
33                    :
34                 );
35
36     temp0 = 16;
37     temp1 = 48;
38     asm volatile ( "tigra4 %0, %1, %2;\n"
39                    :"=r" (tig4)
40                    :"r" (temp0), "r" (temp1)
41                    :
42                 );
43
44     temp0 = 32;
45     temp1 = 96;
46     asm volatile ( "tigra5 %0, %1, %2;\n"
47                    :"=r" (tig5)
48                    :"r" (temp0), "r" (temp1)
49                    :
50                 );
51
52     temp0 = 64;
53     temp1 = 192;
54     asm volatile ( "tigra6 %0, %1, %2;\n"
```

```
55                     :"=r" (tig6)
56                     :"r" (temp0), "r" (temp1)
57                     :
58                 );
59 
60     temp0 = 128;
61     temp1 = 384;
62     asm volatile ( "tigra7 %0, %1, %2;\n"
63                     :"=r" (tig7)
64                     :"r" (temp0), "r" (temp1)
65                     :
66                 );
67     return 0;
68 }
```

Listing 8: C code used to test the arbitrary custom logic in the Taiga processor via the TIGRA interface.

```
module tigra_cl(
        input logic clk,
        input logic rst,
        tigra_interface.tigra_side tigra
);

interface tigra_interface;
    tigra_inputs_t tigra_inputs;
    logic latch_next_insn;
    logic tigra_valid;
    logic mem_valid;
    logic [XLEN-1:0] tigra_out;

    modport tigra_side(input tigra_inputs, latch_next_insn, mem_valid,
                       output tigra_valid, tigra_out);
    modport taiga_side(input tigra_valid, tigra_out, output tigra_inputs,
                       latch_next_insn, mem_valid);

endinterface

typedef struct packed{
        logic [XLEN-1:0] rs1;
        logic [XLEN-1:0] rs2;
        logic [31:0] instruction;
} tigra_inputs_t;
```

Listing 9: Module declaration of the TIGRA custom logic in the Taiga processor using a System Verilog interface and struct to simplify design.

```
module tigra_cl(
    input logic clk,
    input logic rst,
    input tigra_inputs_t tigra_inputs,
    input logic latch_next_insn,
    output logic tigra_valid,
    input logic mem_valid,
    output logic [31:0] tigra_out
);
```

Listing 10: Flattened module declaration of the TIGRA custom logic for use with Dynamic Function eXchange.

Appendix B Extended Timing Diagrams

This appendix includes extra timing diagrams not included within the main body of the dissertation. These images are not required to explain the benefits of using the TIGRA interface, but do provide the full implementation for each design described in Chapter 5.

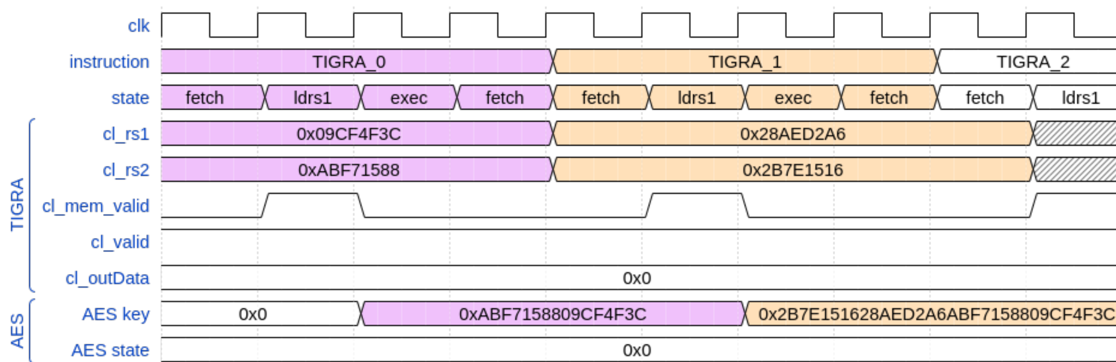


Figure 1: Simulation of the first two TIGRA instructions in PicoRV32 with AES via TIGRA. These instructions both complete within one clock cycle.

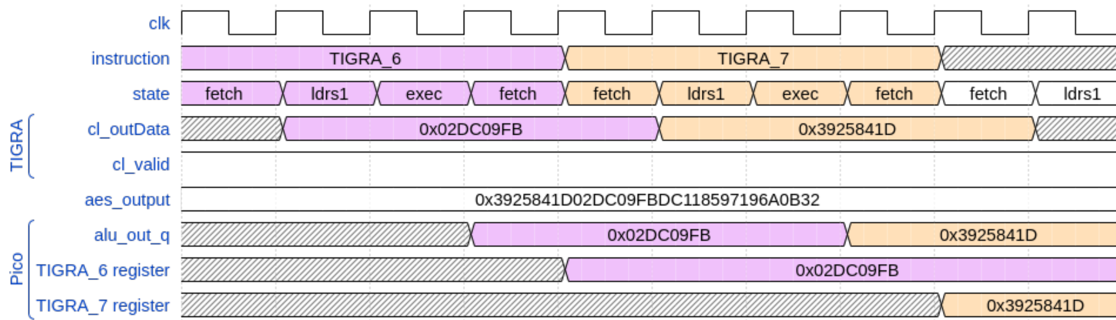


Figure 2: Simulation of the last two TIGRA instructions in PicoRV32 with AES via TIGRA. These instructions all complete within one clock cycle.

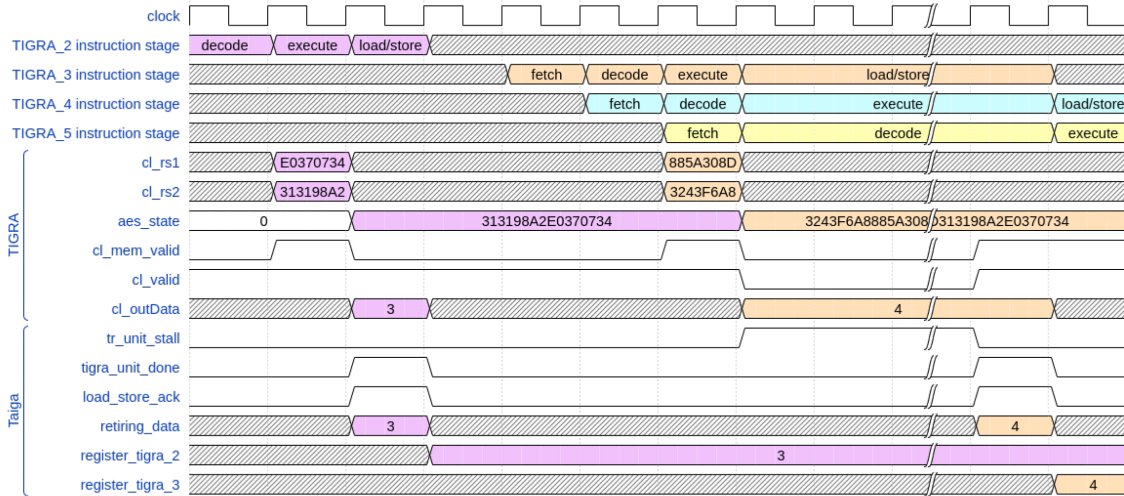


Figure 3: Simulation of TIGRA instructions 2 and 3 when using custom logic with AES. Instruction 2 completes within one clock cycle, while instruction 3 initiates a stall and completes the encryption.

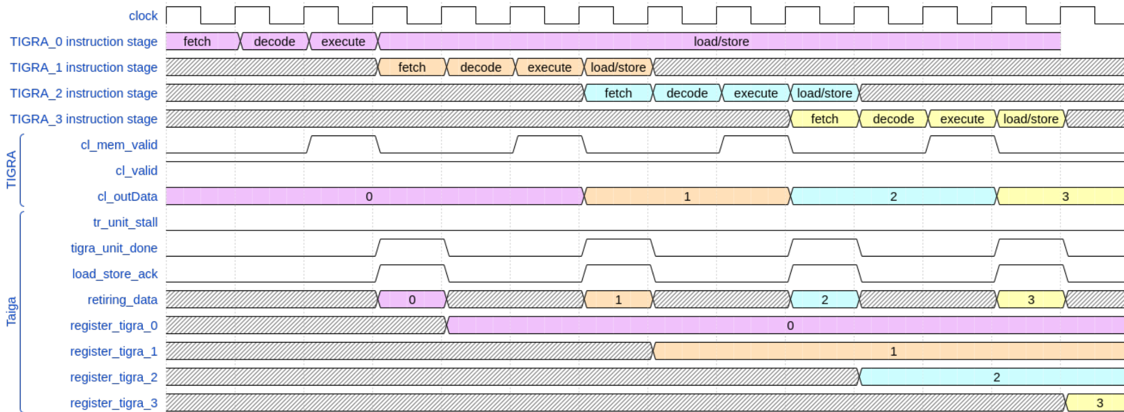


Figure 4: Simulation of the first four TIGRA instructions when using custom logic with arbitrary return values and stalls. These instructions all complete within one clock cycle.

Appendix C   Raw Data from Simulations

The information in this section shows some of the raw data obtained when testing the different processors. This data is provided to back up claims made by the researcher in different sections of this dissertation.

```
1  80002898:     09cf57b7      lui     a5,0x9cf5
2  8000289c:     abf716b7      lui     a3,0xabf71
3  800028a0:     f3c78793      addi    a5,a5,-196 # 9cf4f3c <buflen.2800+0x9cf4efc>
4  800028a4:     58868693      addi    a3,a3,1416 # abf71588 <__global_pointer$+0x2bf6e17c>
5  800028a8:     00d786ab      tigra1_0        a3,a5,a3
6  800028ac:     28aed7b7      lui     a5,0x28aed
7  800028b0:     2b7e1737      lui     a4,0x2b7e1
8  800028b4:     2a678793      addi    a5,a5,678 # 28aed2a6 <buflen.2800+0x28aed266>
9  800028b8:     51670713      addi    a4,a4,1302 # 2b7e1516 <buflen.2800+0x2b7e14d6>
10 800028bc:     00e797ab      tigra1_1        a5,a5,a4
11 800028c0:     e0370737      lui     a4,0xe0370
12 800028c4:     3131a637      lui     a2,0x3131a
13 800028c8:     73470713      addi    a4,a4,1844 # e0370734 <__global_pointer$+0x6036d328>
14 800028cc:     8a260613      addi    a2,a2,-1886 # 313198a2 <buflen.2800+0x31319862>
15 800028d0:     00c7272b      tigra1_2        a4,a4,a2
16 800028d4:     885a3637      lui     a2,0x885a3
17 800028d8:     3243f5b7      lui     a1,0x3243f
18 800028dc:     08d60613      addi    a2,a2,141 # 885a308d <__global_pointer$+0x859fc81>
19 800028e0:     6a858593      addi    a1,a1,1704 # 3243f6a8 <buflen.2800+0x3243f668>
20 800028e4:     00b6362b      tigra1_3        a2,a2,a1
21 800028e8:     00f6c7ab      tigra1_4        a5,a3,a5
22 800028ec:     00c7572b      tigra1_5        a4,a4,a2
23 800028f0:     00e7e7ab      tigra1_6        a5,a5,a4
24 800028f4:     00d7f7ab      tigra1_7        a5,a5,a3
```

Listing 11: Binary object dump of AES test code for the Rocket Chip processor. This shows the instructions required to complete the code in Listing 6 and demonstrates the instruction reordering completed by the compiler.

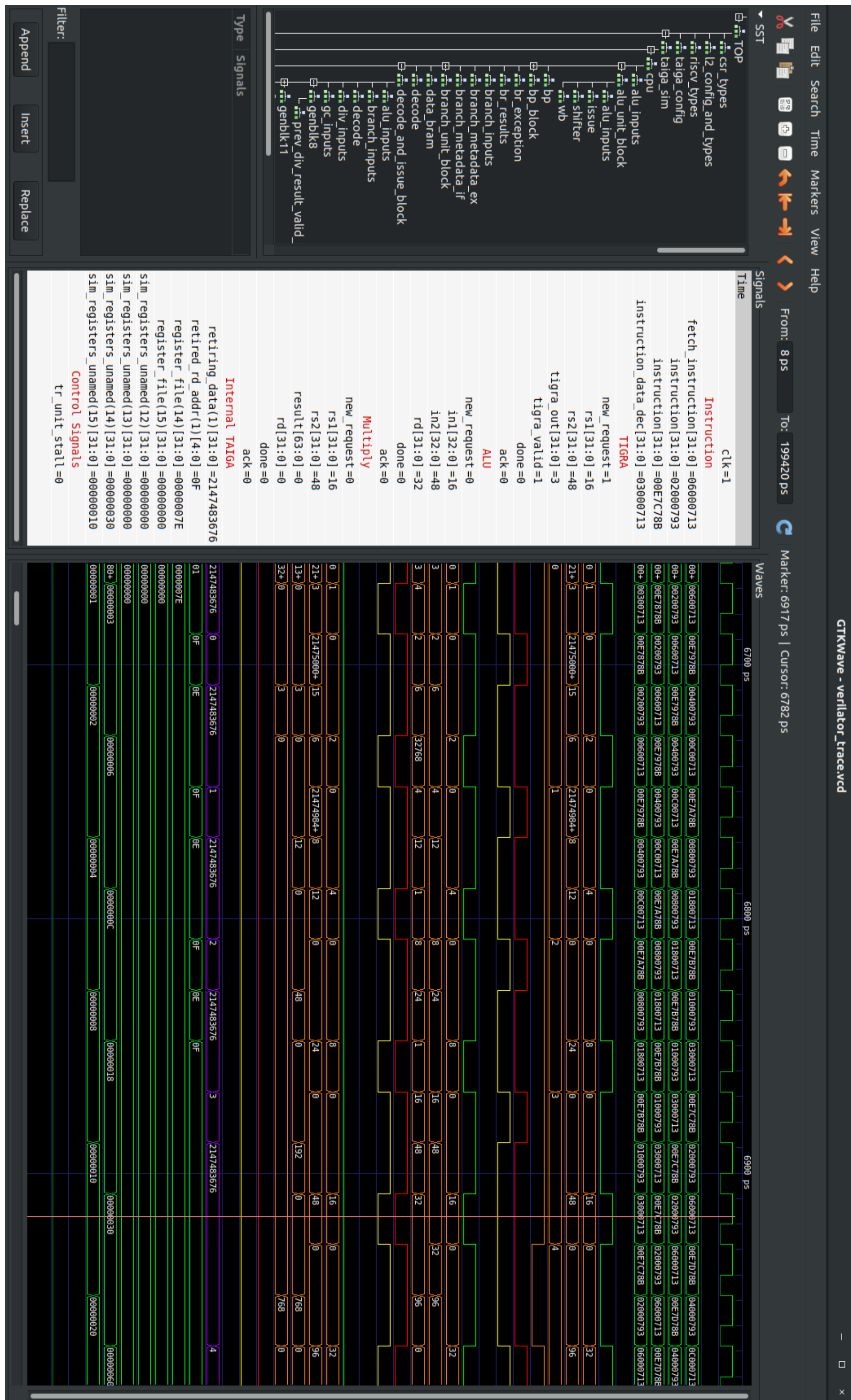


Figure 5: Original timing diagram showing six of the eight instructions for custom logic with arbitrary return values and stall lengths. This diagram shows the difficulty of reading content in this format compared to the transcribed waveforms shown in the paper.

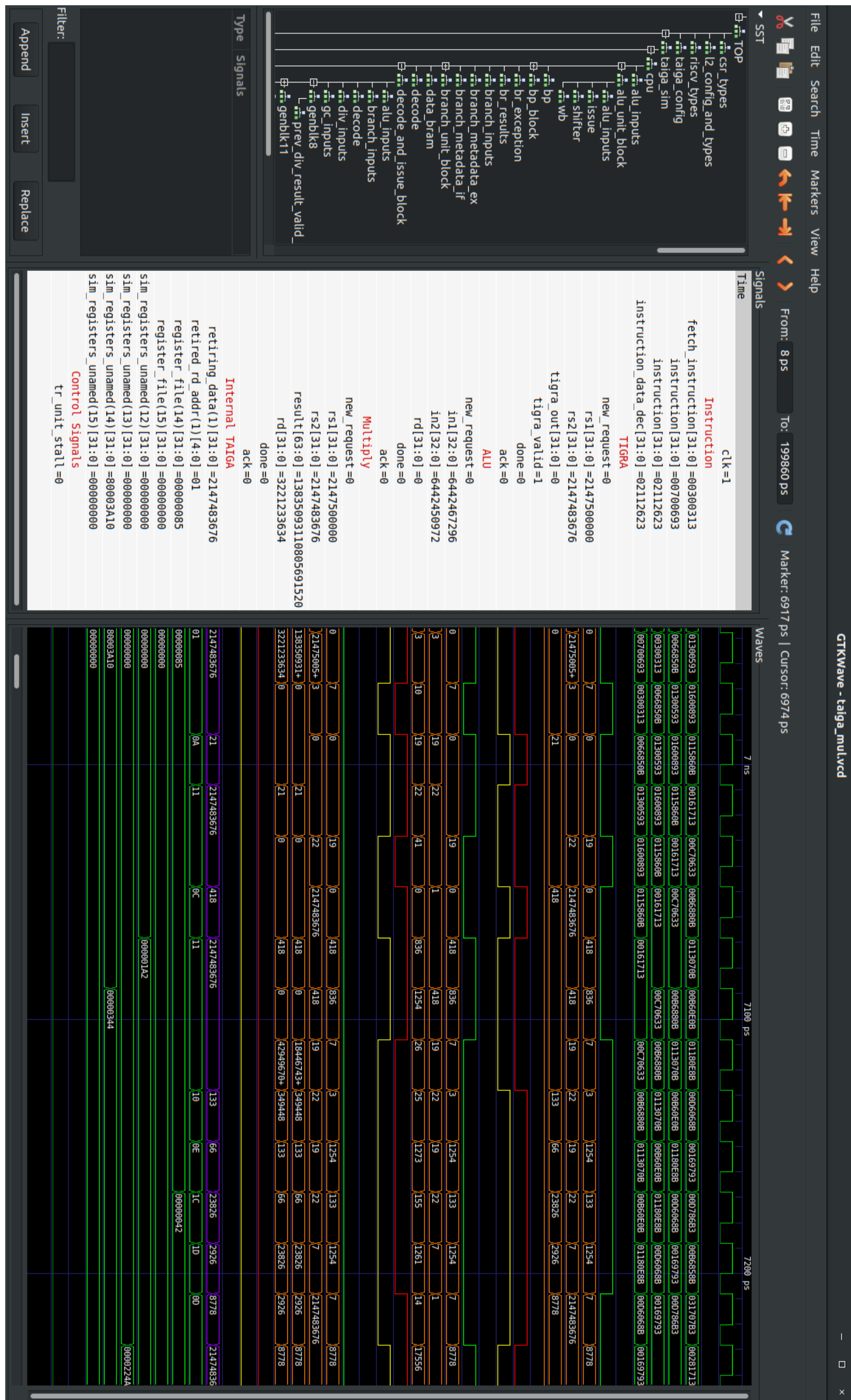


Figure 6: Original timing diagram showing the first 7 TIGRA multiplication instructions. This diagram shows the compiler reordering commands for the code in Listing 7. Here you can see the processor issue 5 TIGRA instructions consecutively, while the code issues at most 3 consecutively.

Bibliography

[1] Jairo Walber Abdala Castro and Aurelio Morales-Villanueva. Exploring dynamic partial reconfiguration in a tightly-coupled coprocessor attached to a risc-v soft-processor on a fpga. In *2021 IEEE XXVIII International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*, pages 1–4, 2021.

[2] M. S. Abdul-Karim, K. H. Rahouma, and K. Nasr. High Throughput and Fully Pipelined FPGA Implementation of AES-192 Algorithm. In *2020 International Conference on Innovative Trends in Communication and Computer Engineering (ITCE)*, pages 137–142, 2020.

[3] M. Alizadeh and M. Sharifkhani. Extending RISC- V ISA for Accelerating the H.265/HEVC Deblocking Filter. In *2018 8th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 126–129, 2018.

[4] Amazon Web Services. AWS EC2 FPGA Development Kit. `https://github.com/aws/aws-fpga`, 2020.

[5] AMD/Xilinx. 76386 - vivado 2019.1 - [drc vivado - dfx - inbb-3 black box error within dfx project mode. `https://support.xilinx.com/s/article/76386?language=en_US`, 2021.

[6] AMD/Xilinx. High level design features - dynamic function exchange. `https://www.xilinx.com/products/design-tools/vivado/high-level-design.html#dfx`, 2022.

[7] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[8] Anthony Bybell. Welcome to gtkwave. `http://gtkwave.sourceforge.net/`, Sep 2021.

[9] L. Calicchia, V. Ciotoli, G. C. Cardarilli, L. di Nunzio, R. Fazzolari, A. Nannarelli, and M. Re. Digital Signal Processing Accelerator for RISC-V. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 703–706, 2019.

[10] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi. Deep Positron: A Deep Neural Network Using the Posit Number System. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1421–1426, 2019.

[11] Doris Chen and Deshanand Singh. Fractal video compression in opencl: An evaluation of cpus, gpus, and fpgas as acceleration platforms. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 297–304, 2013.

[12] Joan Daemen and Vincent Rijmen. *The design of Rijndael: the wide trail strategy explained.* Springer, 2001.

[13] Brad Green, Dillon Todd, Jon C. Calhoun, and Melissa C. Smith. Tigra: A tightly integrated generic risc-v accelerator interface. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 779–782, 2021.

[14] Gustafson and Yonemoto. Beating Floating Point at Its Own Game: Posit Arithmetic. *Supercomput. Front. Innov.: Int. J.*, 4(2):71–86, June 2017.

[15] A. Hodjat and I. Verbauwhede. A 21.54 gbits/s fully pipelined aes processor on fpga. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 308–309, 2004.

[16] Sherif Hosny, Eslam Elnader, Mostafa Gamal, Abdelrhman Hussien, Ahmed H. Khalil, and Hassan Mostafa. A software defined radio transceiver based on dynamic partial reconfiguration. In *2018 New Generation of CAS (NGCAS)*, pages 158–161, 2018.

[17] Sherif Hosny, Eslam Elnader, Mostafa Gamal, Abdelrhman Hussien, and Hassan Mostafa. Multi-partitioned software defined radio transceiver based on dynamic partial reconfiguration. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2020.

[18] Junjie Hou, Yongxin Zhu, Sen Du, Shijin Song, and Yuefeng Song. Fpga-based scale-out prototyping of degridding algorithm for accelerating square kilometre array telescope data processing. *IEEE Access*, 8:15586–15597, 2020.

[19] Homer Hsing. tiny_aes. `https://opencores.org/projects/tiny_aes`, 2013.

[20] Qijing Huang, Christopher Yarp, Sagar Karandikar, Nathan Pemberton, Benjamin Brock, Liang Ma, Guohao Dai, Robert Quitt, Krste Asanovic, and John Wawrzynek. Centrifuge: Evaluating full-system hls-generated heterogenous-accelerator socs using fpga-acceleration. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019.

[21] Intel. Intel FPGA SDK for OpenCL. `https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/support.html`, 2021.

[22] Intel. Partial reconfiguration. urlhttps://www.intel.com/content/www/us/en/software/programmable/quartus-prime/partial-reconfiguration.html, 2022.

[23] RISC-V International. RISC-V International, 2021.

[24] A. Irwansyah, V. P. Nambiar, and M. Khalil-Hani. An AES Tightly Coupled Hardware Accelerator in an FPGA-based Embedded Processor Core. In *2009 International Conference on Computer Engineering and Technology*, volume 2, pages 521–525, 2009.

[25] M. K. Jaiswal and H. K. . So. Universal number posit arithmetic generator on FPGA. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1159–1162, 2018.

[26] M. K. Jaiswal and H. K. So. Architecture Generator for Type-3 Unum Posit Adder/Subtractor. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

[27] M. K. Jaiswal and H. K. So. PACoGen: A Hardware Posit Arithmetic Core Generator. *IEEE Access*, 7:74586–74601, 2019.

[28] Manish Kumar Jaiswal. PACoGen: Posit Arithmetic Core Generator, 2019.

[29] Zheming Jin and Hal Finkel. Power and performance tradeoff of a floating-point intensive kernel on opencl fpga platform. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 716–720, 2018.

[30] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[31] Emre Karabulut and Aydin Aysu. Rantt: A risc-v architecture extension for the number theoretic transform. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 26–32, 2020.

[32] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42, 2018.

[33] B. Koppelmann, P. Adelt, W. Mueller, and C. Scheytt. Risc-v extensions for bit manipulation instructions. In *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 41–48, 2019.

[34] Theresa T. Lê. A tightly integrated generic instruction risc-v accelerator (tigra) for the rocket core. Master's thesis, Clemson University, 2021.

[35] Shuai Li, Yukui Luo, Kuangyuan Sun, Nandakishor Yadav, and Kyuwon Ken Choi. A novel fpga accelerator design for real-time and ultra-low power deep convolutional neural networks compared with titan x gpu. *IEEE Access*, 8:105455–105471, 2020.

[36] Alexandra Listl, Daniel Mueller-Gritschneder, Fabian Kluge, and Ulf Schlichtmann. Emulation of an asic power, temperature and aging monitor system for fpga prototyping. In *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, pages 220–225, 2018.

[37] Jinming Lu, Chao Fang, Mingyang Xu, Jun Lin, and Zhongfeng Wang. Evaluations on deep neural networks training using posit number system. *IEEE Transactions on Computers*, 70(2):174–187, 2021.

[38] Arunkumar M V, Ganesh Bhairathi, and Harshal Hayatnagarkar. Perc: Posit enhanced rocket chip. In *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*, 05 2020.

[39] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. Nvidia tensor core programmability, performance amp; precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531, 2018.

[40] Eric Matthews and Lesley Shannon. Taiga: A new risc-v soft-processor framework enabling high performance cpu architectural features. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2017.

[41] Davide Pala. Design and programming of a coprocessor for a RISC-V architecture, 2017.

[42] V. Patil, A. Raveendran, P. M. Sobha, A. David Selvakumar, and D. Vivian. Out of order floating point coprocessor for RISC V ISA. In *2015 19th International Symposium on VLSI Design and Test*, pages 1–7, 2015.

[43] Thomas M Schulte and Steve Leibson. Intel® FPGAs Accelerate Intel® Xeon® Scalable Processors in Servers and High-End Embedded Systems. Technical report, Intel, 2019.

[44] Wilson Snyder. Welcome to verilator. `https://www.veripool.org/verilator/`, 2022.

[45] Ko Stoffelen. *Efficient Cryptography on the RISC-V Architecture*, pages 323–340. 09 2019.

[46] Tsubasa Takaki, Yang Li, Kazuo Sakiyama, Shoei Nashimoto, Daisuke Suzuki, and Takeshi Sugawara. An optimized implementation of aes-gcm for fpga acceleration using high-level synthesis. In *2020 IEEE 9th Global Conference on Consumer Electronics (GCCE)*, pages 176–180, 2020.

[47] Xubin Tan, Jaume Bosch, Miquel Vidal, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, and Mateo Valero. General purpose task-dependence management hardware for task-based dataflow programming models. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 244–253, 2017.

[48] Balkis Tej, Marwa Hannachi, and Abdessalem Ben Abdelali. Partial dynamic reconfiguration for efficient adaptive implementation of a video shot boundary detection system. In *2020 4th International Conference on Advanced Systems and Emergent Technologies (IC_ASET)*, pages 327–331, 2020.

[49] Dillon W. Todd. Tightly coupling the picorv32 risc-v processor with custom logic accelerators via a generic interface. Master's thesis, Clemson University, 2021.

[50] Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA, December 2019.

[51] Clifford Wolf. PicoRV32 - A Size-Optimized RISC-V CPU. `https://github.com/cliffordwolf/picorv32`, 2019.

[52] Xilinx. Vivado design suite tutorial - dynamic function exchange. Technical Report UG947, 2020 [Online].

[53] Xilinx. Vitis Unified Software Platform. `https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html#documentation`, 2021.

[54] Xilinx. Vivado design suite user guide - dynamic function exchange. Technical Report UG909, 2022 [Online].

[55] O. Yanovskaya, M. Yanovsky, and V. Kharchenko. The concept of green cloud infrastructure based on distributed computing and hardware accelerator within fpga as a service. In *Proceedings of IEEE East-West Design Test Symposium (EWDTS 2014)*, pages 1–4, 2014.

[56] G. Zhang, K. Zhao, B. Wu, Y. Sun, L. Sun, and F. Liang. A RISC-V based hardware accelerator designed for Yolo object detection system. In *2019 IEEE International Conference of Intelligent Applied Systems on Engineering (ICIASE)*, pages 9–11, 2019.

[57] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. May 2020.