May 2021

Tightly Coupling the PicoRV32 RISC-V Processor with Custom Logic Accelerators via a Generic Interface

Dillon Todd
Clemson University, dtodd225@gmail.com

Tightly Coupling the PicoRV32 RISC-V Processor with Custom Logic Accelerators via a Generic Interface

---

A Thesis
Presented to
the Graduate School of
Clemson University

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

---

by
Dillon Wesley Todd
May 2021

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

Abstract

FPGAs facilitate the implementation of custom logic to accelerate specific compputing tasks. This custom logic can be paired with a traditional processor which will offload computationally intensive tasks to the custom logic for faster execution. This pairing may take the form of loosely-coupled co-processors or tightly-coupled, application-specific logic integrated into the processor and utilizing extensions to the instruction set architecture. Both models have disadvantages: the former incurs area overhead and latency as a result of the communication between processor and co-processor, while the specificity of the latter can complicate the development of both the custom logic and custom instructions.

The Tightly Integrated Generic RISC-V Accelerator (TIGRA) interface addresses these issues with an interface for custom logic that avoids latency by providing direct access to the processor's registers and whose generic nature simplifies the modification of custom logic and instructions. In this work, the TIGRA interface is implemented on the PicoRV32, a simple, synthesizable RISC-V processor. Three different custom logic test cases with corresponding custom instructions are implemented to test TIGRA: an AES-128 encryption core, PACoGen hardware for performing posit arithmetic operations, and logic for handling multiplication instructions. Simulation and synthesis results reveal an absence of any additional latency in the execution of instructions using TIGRA, as well as minimal area overhead.

Dedication

This work is dedicated to my parents, who inspired me to study engineering and to pursue a master's degree, and who have supported me the whole way.

Acknowledgments

This work would not have been possible without help from Brad Green, Theresa Lê, and Dr. Smith, for which I am deeply grateful. I must also thank the faculty, staff, and friends from both Clemson University and Baylor University who have taught and supported me over the last six years.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Hardware accelerators and co-processors offer performance and energy-efficiency benefits for System-on-Chips (SoCs) by offloading computationally-intensive tasks from the processor to custom-designed logic capable of completing the task more efficiently [1] [2]. FPGAs (field programmable gate arrays) are an effective means of implementing hardware accelerators due to the parallelism and application specific designs they enable. For this very purpose, Intel incorporated FPGAs in the packages of their Intel® Xeon® Scalable Processor family of devices. To streamline communication between the processor and FPGA accelerator, these devices utilize PCIe (Peripheral Component Interconnect Express) and Intel's UPI (Ultra Path Interconnect) ports [3]. Such protocols can achieve low latency communication, but the best results when integrating an accelerator with a processor, in terms of both performance and overhead, can be achieved with an accelerator tightly integrated into the processor's pipeline [4], such that the connection introduces no latency at all.

Several developments in recent years have facilitated the design of such SoCs. One is the increasing availability of programmable logic in FPGAs [5], which can be used to quickly develop, modify, and implement hardware designs. Another is the emergence of RISC-V, a free, open-standard instruction set architecture (ISA) that can be easily extended with custom instructions capable of utilizing an accelerator. The use of programmable logic, RISC-V, and various IP (intellectual property) cores can decrease the time and cost of developing a specialized and efficient SoC.

A remaining hurdle in utilizing hardware accelerators, however, is the task of integrating

them with the processor. Solutions often require application-specific modifications to the processor's design, which limit its ability to accommodate changes in the custom logic. This work thus presents TIGRA, a Tightly Integrated, Generic RISC-V Accelerator interface, as a means of providing a custom logic accelerator with access to the necessary internal signals of the processor, regardless of the custom logic's functionality and with minimal overhead. To demonstrate TIGRA's performance, it is used to integrate three different custom logic test cases with the PicoRV32 RISC-V processor and shown to execute the associated custom instructions without any added latency.

1.2 RISC-V

RISC-V is a free and open ISA that originated from the Parallel Computing Lab of the University of California at Berkeley. Its creators promote it as an alternative to proprietary ISAs, such as those of ARM and Intel, the licenses for which can require significant time and money to obtain. They envision a processor industry that benefits from standards, competition, innovation, and sharing of ideas based on an open ISA, similar to what has been done in the software domain with Linux [6].

The RISC-V ISA is designed with lessons from its predecessors and the modern demands for processors in mind, aiming to remain simple, versatile, and stable [6]. There exist four versions of the ISA, differentiated by the number and width of integer registers they support, and a variety of optional extensions to add functionality, such as instructions for multiplication, division, and floating point values. The decision to keep the base integer ISA, standard extensions, and other extensions separate in this manner allows implementations to be tailored to their application and maximize energy efficiency by only devoting resources to needed functionality [7].

Naming conventions are specified to represent the version and the extensions in an implementation of the RISC-V ISA. For example, the name of an implementation using the 32-bit base integer ISA and the normal total of 32 registers will begin with "RV32I" and have letters appended according the extensions used. If multiplication ("M") and compressed instruction ("C") extensions are included, the name becomes "RV32IMC". The RV32E base integer ISA behaves similarly, but reduces the total number of registers to 16, while RV64I and RV128I represent versions of the ISA with register widths of 64 and 128 bits respectively [7].

The instructions of RISC-V do not differ greatly from those of typical ISAs, as a large ma-

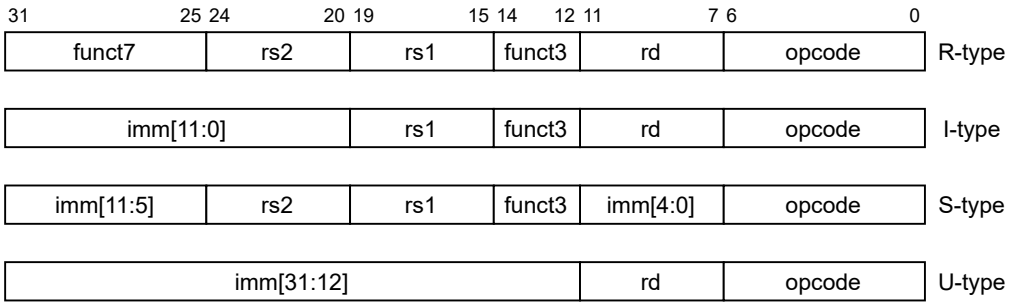

Figure 1.1: Formats of the four core instruction types in RISC-V.

jority of them have historical precedents [8]. Most of these instructions fall into one of four core formats: R-, I-, S-, and U-type instructions, which are shown in Fig. 1.1. It can be seen that the locations of the various fields in the instructions are maintained between types when possible to simplify the task of decoding. For every type, the seven least significant bits are occupied by the opcode, which, together with the minor opcode fields of $funct3$ and $funct7$ when applicable, identifies the specific instruction. The fields $rs1$ and $rs2$ identify the registers containing the instruction operands and rd represents the register to which the result will be written. Values in an immediate field (imm) can have various uses, such as one of the operands in the I-type add immediate ($addi$) instruction or as the offset from a memory address in the S-type store word (sw) or I-type load word (lw) instructions [7].

An important characteristic of RISC-V is the way it allows for non-standard extensions to the ISA. Encoding spaces of various lengths are left available in the 32-bit instruction format, meaning new instructions can be designed to utilize opcodes not used by any existing instructions. In particular, four major opcodes, each representing a 25-bit encoding space, are specifically reserved for custom extensions in the RISC-V specification. This flexibility aligns with RISC-V's goal of facilitating application-specific designs and makes it a good fit for projects like this one, which aim to integrate custom accelerators and associated instructions into a processor.

1.3 PicoRV32

The PicoRV32 is a simple, synthesizable RISC-V processor designed to serve as an auxiliary processor in FPGA designs. It implements the RV32I instruction set and includes a variety of parameters that allow configuration for the RV32E instruction set, inclusion of the "M" and

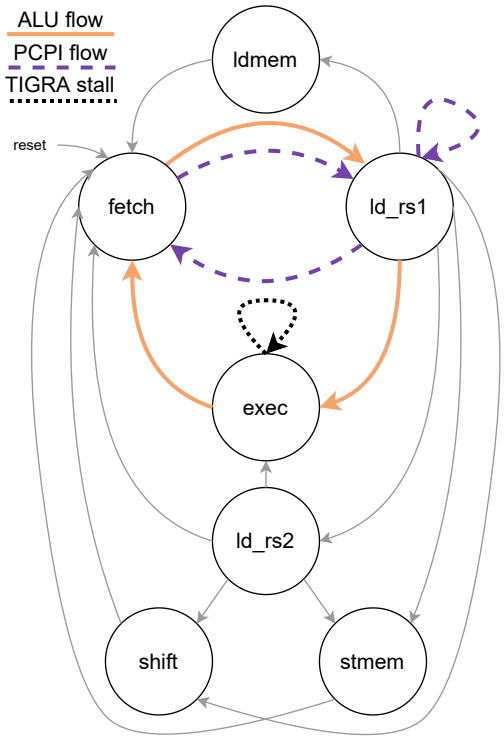

Figure 1.2: PicoRV32 state machine diagram, highlighting the typical ALU flow, PCPI flow, and the stall in *exec* added with TIGRA.

"C" RISC-V extensions, and other functionality. One significant feature is the Pico Co-Processor Interface (PCPI), which can be enabled to allow coupling of an FPGA co-processor or accelerator with the PicoRV32. Included in the Verilog description are modules that utilize PCPI to implement the RISC-V multiply and divide instructions as examples of how the interface can be used to add custom logic and instructions to the processor. Since TIGRA aims to accomplish similar tasks in a different manner, PCPI serves as a useful comparison.

As shown in the analysis in [5] and the synthesis results of [9], the design of PicoRV32 targets a small footprint and high maximum frequency. Consequently, it lacks features that would increase its complexity, such as a multi-stage pipeline. However, while not pipelined in the sense that different stages of multiple instructions execute simultaneously, the processor does feature a state machine that divides the execution of each instruction into stages. In all, there are eight possible states: $fetch$, ld_rs1, ld_rs2, $exec$, $ldmem$, $stmem$, $shift$, and $trap$. Fig. 1.2 illustrates the processor's flow through these states. By default, the $ENABLE\_REGS\_DUALPORT$ parameter of the PicoRV32 will be enabled, allowing the processor to read two values from the register file

simultaneously and eliminating any need for the ld_rs2 state. The *trap* state, meanwhile, exists to let the processor handle unrecognized instructions. All six other states may be used during normal operation.

The flow of a normal ALU instruction such as an add, highlighted in orange in Fig. 1.2, is as follows. An instruction is first read in from a memory location corresponding to the current program counter and appears on the 32-bit *mem_rdata* input of the PicoRV32's native memory interface. This first step occurs on the first cycle of the $fetch$ stage. On the next cycle, the processor begins the decoding process by parsing the major opcode and the registers from the instruction according to the RISC-V instruction formats, then setting the *decoder_trigger* signal. Based on the decoded major opcode, the specific instruction is next identified according to the minor opcode fields $funct3$ and $funct7$. Each instruction recognized by the PicoRV32 has a corresponding 1-bit flag to signal whether it matches the current instruction. One of these flags is set at the start of the next cycle, when the processor transitions from $fetch$ to ld_rs1. During ld_rs1, the instruction operands are loaded from the register file according to the register numbers previously decoded. Additionally, one of several 1-bit registers representing the possible ALU operations is set according to the instruction flag. In the case of an add instruction, for example, the register $is\_lui\_auipc\_jal\_jalr\_addi\_add\_sub$ gets set high to indicate that the sum/difference result from the ALU should be used. This register is set on the transition to the *exec* state, during which the ALU operations occur and the appropriate result is written to the alu_out register. As the processor transitions back to the $fetch$ state, the result is latched into alu_out_q. In addition to fetching the next instruction and beginning the decode process as previously described, the PicoRV32 performs its write back operations for the previous instruction in this state, writing data to the register file from either alu_out_q or reg_out.

In all, this process takes two cycles in the $fetch$ state, one in ld_rs1, and one in *exec*, and represents the flow of primary interest for this project. The flow is somewhat different, however, in the cases of load and store instructions. Rather than *exec*, these instructions send the PicoRV32 to the *ldmem* and *stmem* states respectively, where it remains for five clock cycles. During that period, the processor reads and begins decoding the next instruction with similar timing to the case previously described, but does not return to $fetch$ until values have been read from or written to memory, which requires a couple more clock cycles. Logic associated with *decoder_trigger* and other signals ensures correct timing during this process. As will be discussed in Section 3.1.1, this logic would prove relevant to the task of integrating custom logic with the PicoRV32 without adding

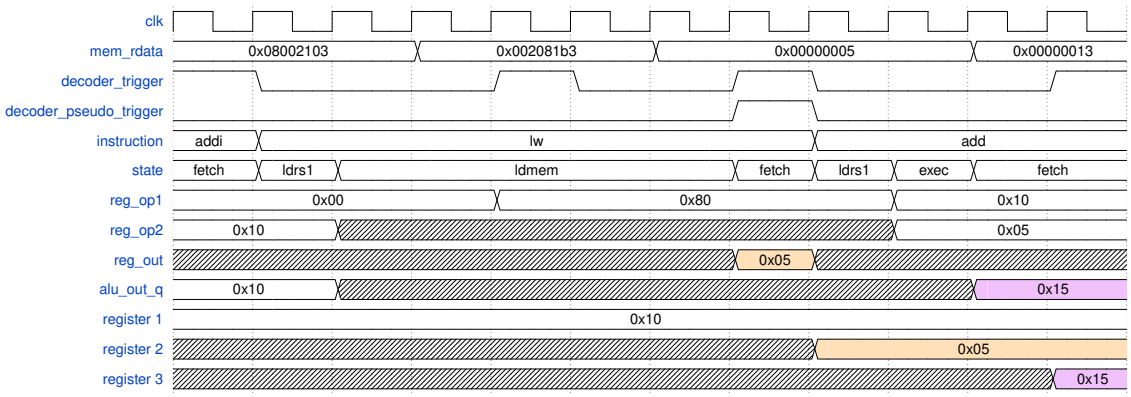

Figure 1.3: Timing diagram for PicoRV32's execution of *lw* and *add* instructions. The *lw* loads a value of 0x05 into register 2, and the *add* writes the sum of registers 1 and 2 to register 3.

latency.

Fig. 1.3 illustrates the PicoRV32's execution of a *lw* instruction followed by an *add*, including the flow through states for each case. It can be seen that the *lw* stores the value of 0x05 in register 2. During the *add*, the addition operands that appear on *reg_op*1 and *reg_op*2 are then drawn from registers 1 and 2 respectively, and the result is stored in register 3. The colored boxes highlight the paths each operation's results take to reach their destination registers. Also visible is the way *mem_rdata* can contain both instructions and other data. During the *ldmem* stage in the diagram, the *mem_rdata* value of 0x002081b3 represents the subsequent *add* instruction, but the next value of 0x05 is the data to be loaded into register 2. The signal *decoder_pseudo_trigger* goes high during this latter period, indicating the presence of non-instruction data on *mem_rdata*. This represents the case in which the instruction data must somehow be latched so as to remain available after the completion of the *lw*.

1.3.1 Pico Co-Processor Interface (PCPI)

The PCPI interface of the PicoRV32 features four outputs (*pcpi_valid*, *pcpi_insn*, *pcpi_rs*1, and *pcpi_rs*2) and four inputs (*pcpi_wr*, *pcpi_rd*, *pcpi_wait*, and *pcpi_ready*). The outputs *pcpi_rs*1 and *pcpi_rs*2 are always assigned the operands of the current instruction. The output *pcpi_valid* indicates valid data on these operand signals and may serve as a start signal for the co-processor. The output *pcpi_insn* receives the entire 32-bit instruction when PCPI is enabled and the *decoder_trigger* is set, allowing the co-processor to do further decoding of the instruction according to the logic it

implements. For example, in the provided multiplication core, the $funct3$ field of the instruction is used to determine which bits of the result to return, based on the multiplication instructions defined in the "M" extension of the RISC-V instruction set.

The PCPI inputs are used to indicate the co-processor's state of completion. During execution of the instruction, *pcpi_wait* is set high to stall the PicoRV32 until the co-processor writes its result to *pcpi_rd* and sets *pcpi_ready* high. On the PicoRV32's end, the state machine will remain in ld_rs1 while stalling, then return to $fetch$ for write back, as shown by the purple, dashed line in Fig. 1.2. A *pcpi_timeout* register will count down cycles from 15 if *pcpi_wait* is not set, limiting the time the processor will stall without response from the co-processor before proceeding to the trap state. Notably, the PicoRV32 does not recognize instructions meant for the co-processor based on their opcode, instead relying on the PCPI parameters and signals to determine whether an unrecognized instruction is a valid PCPI instruction or just illegal.

As indicated in Fig. 1.2, TIGRA is designed to integrate accelerators into the processor pipeline in a way that more closely resembles the behavior of a typical ALU or execution unit than PCPI allows. Comparisons of the implementation and performance of TIGRA and PCPI are provided in Sections 3.4, 4.3, and 4.4.

1.4 Outline

Chapter 2 discusses some related work on the topics of using custom co-processors and accelerators and extending instruction set architectures. Insights from these works inform the designs of the TIGRA interface and the various test cases described in Chapter 3. These test cases represent implementations of custom logic for AES encryption, posit arithmetic, and multiplication. Simulation results for the custom instructions implemented in each case are presented in Chapter 4, as well as a comparison of TIGRA with PCPI. Chapter 5 contains concluding remarks and possible future work to build off of what has been done with TIGRA and the PicoRV32.

Chapter 2

Related Work

The authors of [1] discuss the trend toward focus on energy efficiency in modern SoCs, which informed the design of RISC-V itself, as seen in [6]. With this in mind, they present a RISC-V hardware accelerator for digital signal processing applications that aims to minimize energy consumption by performing operations in a shorter time than would be achievable through software execution. They integrate their accelerator with the Rocket Chip [10] via the Rocket Custom Coprocessor (RoCC) interface. The use of RoCC allows for the addition of custom instructions without modifying the Rocket core, but results in the accelerator being decoupled from the core in a manner that can add latency to the execution of those instructions, as found in [11]. A more ideal method of integrating a co-processor would use a similarly generic interface, but would tightly couple the processor and co-processor in a way that adds no latency between the two.

The authors of [4] demonstrate two different approaches to the task of supplementing a processor with custom hardware on an FPGA. In their case, the custom logic performs AES encryption. One approach uses a loosely-coupled co-processor meant to represent the more common technique for offloading computationally-intensive routines from the processor. The co-processor in this case is connected via an Avalon Switch Fabric System Bus. A similar interface is utilized in [12], which presents a framework for embedding an accelerator in the available FPGA logic of a network switching hub. As seen in the results of [4], such an interface introduces latency in communication between core and accelerator, resulting in instruction executions slower than what would be seen if the custom logic were part of the processor pipeline.

The second approach in [4], meant to be representative of more recent trends, features

tightly-coupled custom logic utilizing a custom extension to the ISA. The authors report significant speedup from the tightly-coupled implementation relative to the co-processor approach, which they attribute to the elimination of communication overhead found in the latter. Their results illustrate a key idea motivating this work, which is the potential efficiency benefits offered by tightly-coupled custom logic accelerators, but their solution is specific to the AES application. Their tightly-coupled co-processor never receives the instruction itself, meaning all decoding must be accomplished by the processor and must be tailored specifically to the implemented AES instructions. The paper also predates RISC-V, which has made the ISA extension a more accessible solution, as explored in this work with the PicoRV32. The custom AES instructions discussed in Section 3.2 bear significant resemblance to those designed in [4], though their solution requires a greater total number of clock cycles to perform encryption.

In presenting a design for accelerators for algorithms secure against quantum computer attacks ("post-quantum cryptography"), [13] also discusses the distinction between tightly-coupled and loosely-coupled accelerators. The authors point out limitations of previous designs for similar accelerators, such as the communication and area overhead, high resource usage, and inflexibility. Their tightly-coupled solution, called "RISQ-V", extends the RISC-V ISA with what they call the "PQ" extension, utilizing the R-type format for their custom instructions. The logic for executing the custom instructions is integrated into the processor in the form of a pair of new components inserted into the pipeline. These components have direct access to the decoder output and the processor's registers, just like a traditional ALU, but are designed to execute specific cryptography operations more efficiently. While the design aims to reuse processor's existing logic in some cases, it also adds logic to the decoder to provide control signals to the accelerators that identify the various possible operations. The result is an interface that, while more flexible than those of related works, is still specific to post-quantum cryptography operations.

Another example of a tightly-coupled RISC-V accelerator is provided in [2], wherein the authors implement an out-of-order floating point unit for their processor. In their design, custom instructions are only partially decoded by the existing decode logic of the processor, then fully decoded in the new co-processor, thus simplifying the interface between the two. However, the co-processor still depends on the decoder to parse the instruction according the unique instruction format utilized by floating point instructions, limiting the interface's applicability. Both this floating point co-processor and the cryptography accelerator from [13] represent custom logic with application-

specific interfaces in their respective processors. [13] portrays the decision between tightly- and loosely-coupled accelerators as a tradeoff between efficiency and ease of integration. The aim of the TIGRA interface presented in Section 3.1 of this paper is to simplify the task of integrating custom logic in a RISC-V processor, thereby minimizing the difficulty of implementing custom instructions while retaining the efficiency and performance benefits.

Chapter 3

Research Design and Methods

3.1 TIGRA

The Tightly Integrated Generic RISC-V Accelerator (TIGRA) interface is designed to facilitate the integration of FPGA custom logic with a RISC-V processor's pipeline in a way that mimics the operation of a traditional ALU, while not depending on the specifics of the processor's or custom logic's architecture. It allows the custom logic to stall the processor for an arbitrary number of cycles, so operations of any length can be implemented, but adds no additional latency to the process. This integration requires the use of pre-existing signals in the processor to provide and receive data, but also a few synchronization signals to ensure custom instructions execute with the proper timing. In all, TIGRA features seven signals: processor outputs cl_insn, cl_rs1, cl_rs2, cl_mem_valid, and $cl\_latch\_next\_insn$, and processor inputs $cl\_outData$ and cl_valid. These seven signals are listed in Table 3.1.

Custom instructions for use with TIGRA follow the RISC-V R-type format, meaning they include two operand registers, a destination register, a major opcode, and two minor opcodes. Use of this format allows reuse of the processor's existing logic to decode the registers in the custom instruction just as with any other instruction. The operand values retrieved by the processor from its registers are passed to the custom logic through cl_rs1 and cl_rs2. Further decoding of the custom instruction is left up to the custom logic, which receives the instruction from the processor via cl_insn. The custom logic can then use the opcode fields of the instruction to differentiate between the various instructions it implements. Once the operations for the appropriate instruction

Table 3.1: TIGRA Interface Signals

Signal Name	**I/O**	**Function**
cl_insn	Output	Instruction data read from the processor's memory and passed to the custom logic
cl_rs1	Output	First register operand decoded by the processor and passed to the custom logic
cl_rs2	Output	Second register operand decoded by the processor and passed to the custom logic
cl_mem_valid	Output	Indicates that cl_rs1 and cl_rs2 are ready for execution of the current instruction
cl_latch_next_insn	Output	Indicates the current value on cl_insn is a valid instruction and should be latched by custom logic
cl_outData	Input	Result data from custom logic to be written to destination register specified by the instruction
cl_valid	Input	Indicates to the processor that cl_outData is ready to be written to register file

complete, the custom logic writes the result to *cl_outData*, from which the processor can write back to its registers. To ensure the custom instructions execute with the correct values at the correct times, the synchronization signals *cl_latch_next_insn*, *cl_mem_valid*, and *cl_valid* are used to indicate the readiness of the custom instruction, instruction operands, and result respectively. Both the processor and custom logic should include mechanisms to stall while the received synchronization signals are low and set the output synchronization signals high when the corresponding values are ready.

Although all TIGRA instructions follow the R-type format, the use of custom logic allows flexibility in how the instruction can be used, as demonstrated in Section 3.2. In some cases, for example, the user may not want a custom instruction to write an output back to the processor registers. This behavior can be achieved by ensuring the rd field of the instruction contains 00000 and the custom logic outputs a value of 0 for that instruction. The processor will the write 0 to the $x0$, register, which by convention always contains 0. A similar technique is described in the canonical encoding of a NOP given in the RISC-V specification [7]. Conversely, there may be cases when the register operands are not necessary for execution of the custom instruction. In these cases, the values in the $rs1$ and $rs2$ fields of the instruction are arbitrary, though the convention when testing TIGRA was to set both to 00000.

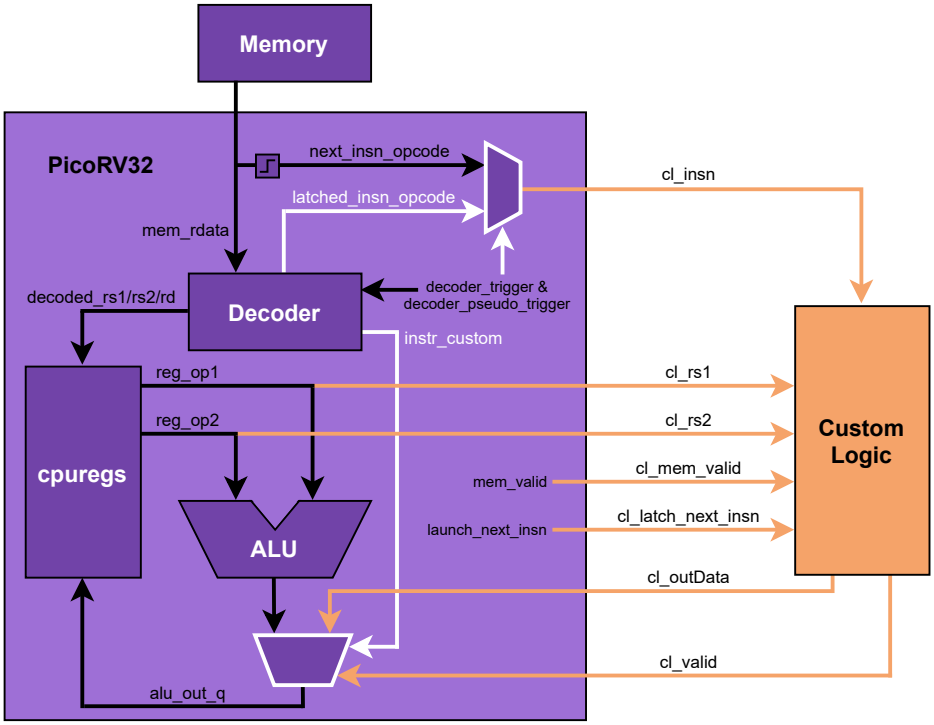

Figure 3.1: PicoRV32 coupled with custom logic through TIGRA interface. Orange signals represent the TIGRA interface, while white signals and multiplexers represent logic added to the PicoRV32.

3.1.1 Adapting PicoRV32 to TIGRA

One goal in designing TIGRA was to minimize overhead by reusing existing signals in the processor as much as possible. However, to keep TIGRA generic, some modifications must be made to the processor to correctly utilize the interface.

For the PicoRV32, the seven TIGRA signals are added as inputs and outputs of the processor. These are illustrated in Fig. 3.1, where the orange lines represent the TIGRA interface, black lines are internal signals of the PicoRV32, and white lines are additions to the processor logic to accommodate TIGRA. The simplest signals to incorporate are cl_rs1, cl_rs2, cl_mem_valid, and $cl\_latch\_next\_insn$, which are simply assigned the values of existing signals reg_op1, reg_op2, mem_valid, and $launch\_next\_insn$ respectively. The first three utilize the existing decoding logic of the processor for retrieving operands from the register file, while $launch\_next\_insn$ uses the processor's current state (e.g., $fetch$) and $decoder\_trigger$ to indicate when the contents of mem_rdata represents the next instruction as opposed to other read data. Understanding and using $launch\_next\_insn$ solved issues initially experienced with store and load instructions, whose

execution utilizes *mem_rdata* both ways, as described in Section 1.3.

Integration of the three remaining TIGRA signals is slightly more complex, and necessitates some adjustments to the PicoRV32's existing design. The signal *cl_insn*, for one, can not simply receive an existing signal in the PicoRV32 due to the way the processor handles load and store instructions. The signal *next_insn_opcode* typically contains the next instruction, obtained during the decode process, but holds non-instruction data during the latter cycles of a load or store. To address this, an additional register, named *latched_insn_opcode*, is created to latch the value of *next_insn_opcode* when it holds an actual instruction. Decode logic already exists to identify this scenario, so *latched_insn_opcode* receives its value at that location in the code. Another added line then writes to *cl_insn* either *latched_insn_opcode* or *next_insn_opcode*, determining the appropriate source based on the processor's *decoder_trigger* and *decoder_pseudo_trigger* signals. These possible sources for *cl_insn* are illustrated by the multiplexer that outputs *cl_insn* in Fig. 3.1. Notably, *cl_insn* does actually receive non-instruction data from *next_insn_opcode* at certain points, but the timing of *cl_latch_next_insn* allows the custom logic to latch the correct instruction before that occurs.

Though they do require adjustments to the PicoRV32, the two processor inputs of TIGRA, *cl_valid* and *cl_outData*, are simple to integrate. The signal *alu_out_q*, used for latching the ALU output to be written back to the register file, originally received the output of the ALU on every clock cycle. Logic is added to optionally latch the value of *cl_outData* into *alu_out_q* instead when a custom instruction has just completed, signalled by *cl_valid* and one or both of the new signals *instr_custom* and *instr_custom_q*, as shown by the multiplexer output *alu_out_q* in Fig. 3.1.

The signal *instr_custom* is added to PicoRV32 as a flag for custom instructions, similar to the existing ones for each regular instruction (e.g., *instr_add*, *instr_lw*). This 1-bit signal is set to 1 upon decoding of the major opcode 0101011. This specific opcode is reserved in the RISC-V specification for custom extensions to the ISA [7], so it is used for all custom instructions for TIGRA. The signal *instr_custom* thus serves as an indicator for a recognized instruction, and can be used to avoid sending the processor to the *trap* state, as well as for modifying logic relevant to the use of custom instructions. It also furthers the goal of treating custom instructions as much like regular ALU instructions as possible, in contrast to PCPI, which has no such identifier for its instructions, as discussed in Section 1.3.1. Another 1-bit signal, *instr_custom_q* is created to latch the value of *instr_custom*, which becomes necessary when the custom instruction stalls, since *instr_custom* may

be cleared as the next instruction is decoded.

It was determined that the processor should stall in the *exec* state of the PicoRV32 during the execution of custom instructions, rather than in ld_rs1 like PCPI. Following a sequence of $fetch$, ld_rs1, *exec* allows the TIGRA dataflow to closely resemble that of regular ALU instructions, as seen in Fig. 1.2, but necessitates several changes to the logic in *exec*. These include two *if* statements, the first of which simply sets *instr_custom_q* if *instr_custom* is set, effectively making *instr_custom_q* an indicator for stalling instructions. The second checks both of those flags and *cl_valid* to determine whether the current instruction is a regular instruction, completed custom instruction, or stalling custom instruction. If stalling, nothing is done and the processor remains in the *exec* state. Otherwise, the subsequent logic allows a regular instruction to proceed as normal back to the $fetch$ state, but some additional logic is included for completed custom instructions. For the cases in which a custom instruction stalled, indicated by *instr_custom_q*, another signal is added, called *do_decoder_trigger*. When set, this signal sets *decoder_trigger* on the following cycle, ensuring the decoding of the next instruction begins at the appropriate time. *do_decoder_trigger* is necessary in the case of stalling instructions, since the variable-length stall does not follow the tight timing constraints of regular instructions, and would thus introduce additional latency if not for this correction. After setting *do_decoder_trigger* and clearing *instr_custom_q* the processor then returns to $fetch$ for write back and decoding of the next instruction as usual.

The end result of these modifications is a version of the PicoRV32 processor capable of providing a custom logic co-processor with access to its registers via the TIGRA interface, and of handling correctly formatted custom instructions with no extra latency, regardless of the operations they represent. In fact, custom instructions requiring no stall, such as simply reading a value from the custom logic, can execute in exactly as many clock cycles as any regular ALU instruction. To test and demonstrate the functionality of TIGRA, three different custom logic use cases are implemented, representing a variety of useful operations. These are discussed in the following sections.

3.2 AES

The Advanced Encryption Standard (AES) is a symmetric key encryption algorithm widely used to ensure secure communication in a variety of fields, such as internet of things (IoT) applications [14][15]. It was adopted in 2000 by the National Institute of Standards and Technology (NIST)

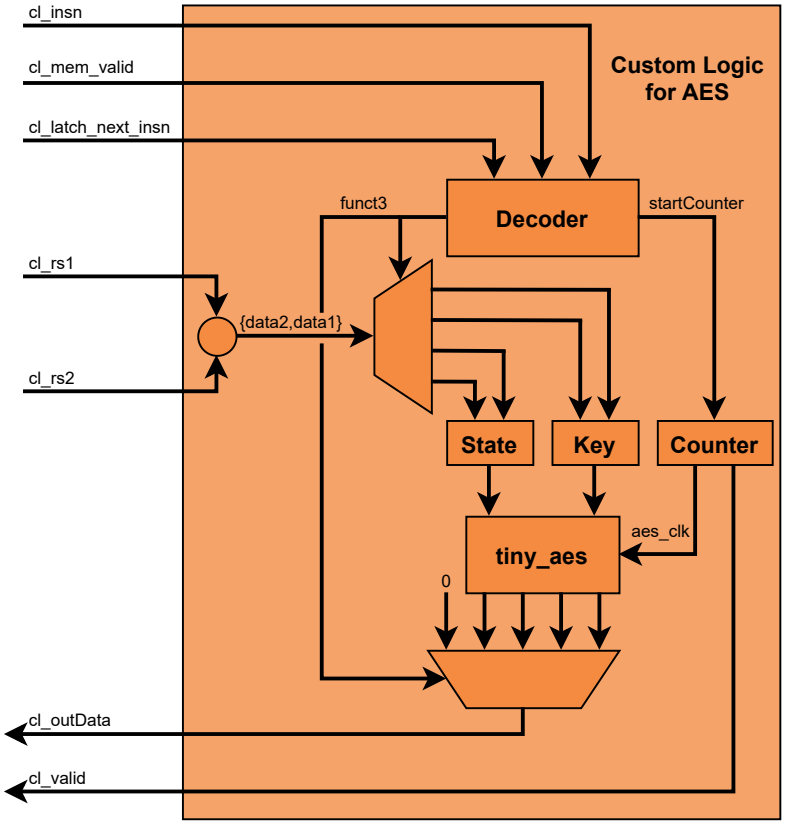

Figure 3.2: TIGRA-compatible custom logic for implementing AES encryption with the PicoRV32.

[16] and has since been included in IEEE standards such as those for Bluetooth, Wireless LAN, and ZigBee [14]. The algorithm utilizes an encryption key input and ten rounds of operations to transform the 128-bit input "state" into the encrypted or decrypted result of equal length. Each round of the algorithm features four steps, referred to as $SubBytes$, $ShiftRows$, $MixColumns$, and $AddRoundKey$, each of which treats the state as a 4x4 matrix of bytes to be manipulated. Three different versions of the AES specification exist, differentiated by their key lengths of 128, 192, or 256 bits [16].

The specifics of this process, however, are not relevant for this project. The use here of AES instead serves as a demonstration of how TIGRA can be used to easily implement custom instructions and logic with a RISC-V processor. Based on an understanding of the inputs, output, and timing requirements of a pre-made AES core, simple custom logic is designed to interface with the PicoRV32, allowing it to encrypt data from its registers and store the result with a short series of instructions.

Table 3.2: Custom Instructions for AES

Opcode	Custom Instruction	Function
000	wr_key_lo	Write lower half of key to custom logic
001	wr_key_hi	Write upper half of key to custom logic
010	wr_state_lo	Write lower half of state to custom logic
011	wr_state_hi	Write upper half of state to custom logic and trigger encryption of the state
100	rd_res_lo	Read least-significant quarter of AES output (bits 31:0) from custom logic and store in rd
101	rd_res_midlo	Read second-least-significant quarter of AES output (bits 63:32) from custom logic and store in rd
110	rd_res_midhi	Read second-most-significant quarter of AES output (bits 95:64) from custom logic and store in rd
111	rd_res_hi	Read most-significant quarter of AES output (bits 127:96) from custom logic and store in rd

Specifically, the AES-128 version of the tiny_aes core from opencores.org [17] is utilized for the test case. This core performs encryption or decryption of the state in 21 clock cycles and includes inputs for the clock, state, and key, and an output for the result. The core is free running, so the operation begins on the first clock cycle that both the desired key and state are present in their respective registers. To make tiny_aes compatible with TIGRA and the PicoRV32, it must be wrapped with custom logic to maintain its inputs and outputs and control its timing.

An immediately apparent challenge in connecting the AES core to the PicoRV32 is the difference in width between the 128-bit key, state and result and the 32-bit registers of the processor. To address this, the custom logic is designed to implement eight custom instructions, differentiated by their $funct3$ minor opcodes, that allow the PicoRV32 to read and write pieces of the AES values in sizes it can handle. The first four instructions each write either the upper or lower half of the key or state from the register file to the appropriate section of the 128-bit state or key register in the custom logic. Each of the last four instructions allows the processor to read a quarter of the AES result into one of its registers. The custom logic to accomplish these tasks is illustrated in Fig. 3.2.

The custom logic first obtains the instruction from the cl_insn TIGRA input and latches it into a register according to $latch\_next\_insn$. From that register, the logic decodes the instruction by checking for the custom major opcode (0101011) and a 1 on the cl_mem_valid input, then performs some actions according the minor opcode. For the first four $funct3$ values (000 - 011), either the key or state register receives the concatenated values of the operand inputs cl_rs2 and cl_rs1. In this way, the processor can utilize the two registers specified by an instruction not as operands, but

as two halves of a 64-bit value, representing half of the 128-bit key or state. Since the encryption result can't be computed until the entire state and key are available, these first four instructions also cause a 0 to be written to *cl_outData* and depend on the instruction's *rd* field containing 000 to ensure no change is made to the PicoRV32's register file, as described in Section 3.1. These first four instructions are named *wr_key_lo*, *wr_key_hi*, *wr_state_lo* and *wr_state_hi* according to their functionality, as seen in Table 3.2. To minimize the number of instructions required for AES, *wr_state_hi* has the additional effect of triggering the encryption process, meaning it must execute only after the desired key and lower half of the state are ready.

Reading the 128-bit result of the AES operation similarly requires four instructions, since only one register's-worth of data can be read at a time by the PicoRV32. These latter four instructions are named *rd_res_lo*, *rd_res_midlo*, *rd_res_midhi*, and *rd_res_hi* and correspond to $funct3$ values of 100 to 111. Each causes a different 32-bit quarter of the result register to be written to *cl_outData*, from which the PicoRV32 can read it and store it in the register designated by the *rd* field of the instruction. The $rs1$ and $rs2$ fields are arbitrary for these four instructions and ignored by the custom logic, though the convention followed was to always set these instruction fields to 0.

To complete the actual encryption process, two modules are instantiated in the custom logic: the AES-128 core and a simple counter module. Since the tiny_aes core lacks any mechanism for controlling its start or indicating the validity of its output, the counter module is used to accomplish both, ensuring proper timing when executing instructions from the PicoRV32. Upon decoding the *wr_state_hi* instruction, the logic sets high the *startCounter* input to the counter module that causes its internal count register to be cleared and begin incrementing on each clock cycle. While the count is below 20, the counter output, which is assigned to the *cl_valid* TIGRA output and gates the clock provided to the AES module, remains 0. After 21 cycles, the counter output returns to 1, causing *aes_clk* and the AES module to halt and indicating via *cl_valid* that the encryption of the state is complete. During this 21-cycle process, the PicoRV32 is held in the *exec* state of the *wr_key_hi* instruction, waiting for *cl_valid* to be set. The AES result can then be read with no delay by subsequent instructions.

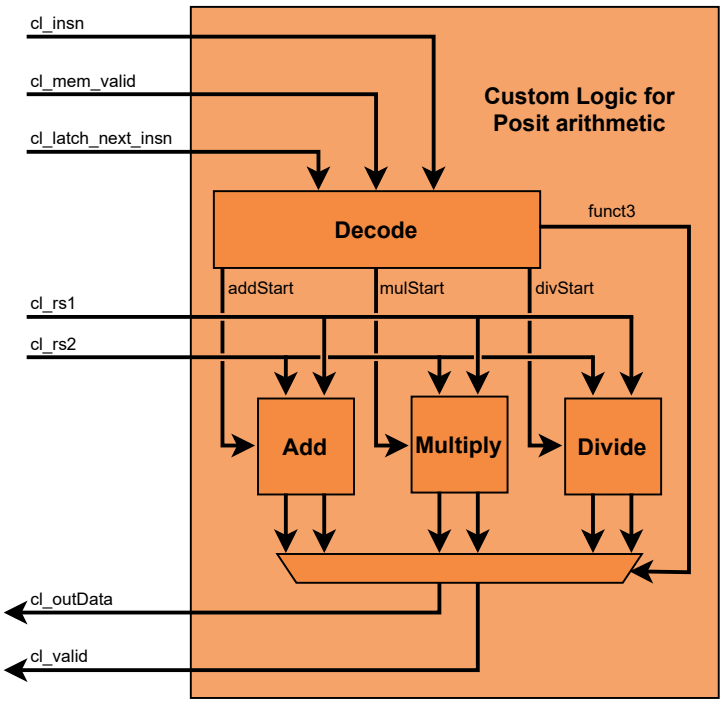

Figure 3.3: TIGRA-compatible custom logic for implementing posit arithmetic with the PicoRV32.

3.3 Posit Arithmetic

Posits are a proposed replacement for floating point values, designed to offer superior accuracy at a lower resource cost [18]. Posit numbers can be expressed with a variable number of bits, where a greater width offers greater precision, and devote a varying number of those bits to the *exponent* field, which determines the possible magnitude of the number.

While some processors contain specific execution units for floating point arithmetic (FPUs), posits currently have far less hardware support. The Posit Arithmetic Core Generator (PACoGen) aims to address this by providing means to generate HDL code for posit adder, multiplier, and divider hardware [19]. For the second TIGRA test case, custom logic and instructions are designed to utilize PACoGen to support posit arithmetic on the PicoRV32. The PACoGen modules are configured to operate on 32-bit posits with 6-bit exponent fields, referred to here as (32,6) posit format, to align with the width of the PicoRV32's registers and match the architectures described in [18].

In contrast to the free-running tiny_aes core, the posit adder, multiplier, and divider each include a "start" input and "done" output, which simplifies the task of ensuring proper timing when designing the custom logic. The primary function of the custom logic, then, is to trigger the correct

Table 3.3: Custom Instructions for Posit Arithmetic

Opcode	Custom Instruction	Function
000	posit_add	Add posits in rs1 and rs2 and store result in rd
001	posit_mult	Multiply posits in rs1 and rs2 and store result in rd
010	posit_div	Divide posit in rs1 by posit in rs2 and store result in rd

operation and output the corresponding result based on the received instruction. This process is illustrated in Fig. 3.3.

Since the PACoGen hardware represents three different operations to perform on pairs of 32-bit data, three custom instructions are implemented that, from the processor's perspective, function similarly to their integer add, multiply and divide counterparts. These custom instructions are listed in Table 3.3. Each corresponds to one of the three PACoGen modules instantiated in the custom logic, labeled as Add, Multiply, and Divide in Fig. 3.3, which each receive cl_rs1 and cl_rs2 as operands, and a start signal provided by the decoding logic.

Similar to the AES custom logic, the instruction received from the processor on cl_insn is latched when $cl\_latch\_next\_insn$ goes high. One of three internal signals, $addStart$, $mulStart$, and $divStart$, can then be set when cl_mem_valid goes high if the instruction contains the custom major opcode and 000, 001, or 010 in the $funct3$ field. Each start signal is mapped to the input of its respective operation's module.

Each module has two outputs mapped to internal signals in the custom logic: one for the result value and one to signal completion of the operation. Based on the latched instruction, the logic multiplexes one pair of outputs to $cl\_outData$ and cl_valid, allowing the processor to exit the *exec* stage where it had stalled and write the result back to its register file. When the instruction does not represent a custom instruction or the $funct3$ field does not contain one of the recognized minor opcodes, $cl\_outData$ simply receives 0 and the cl_valid is kept high to avoid any stall.

3.4 Multiplication

The final case tested is an implementation of the multiplication instructions included in the RISC-V ISA's "M" extension. This test case was chosen specifically because the Verilog description of the PicoRV32 includes a "pcpi_mul" core that can optionally be enabled to support the "M" extension, allowing a direct comparison between the performances of PCPI and TIGRA.

The pcpi_mul core implements the four instructions $mul, mulh, mulhs$, and $mulhsu$, all

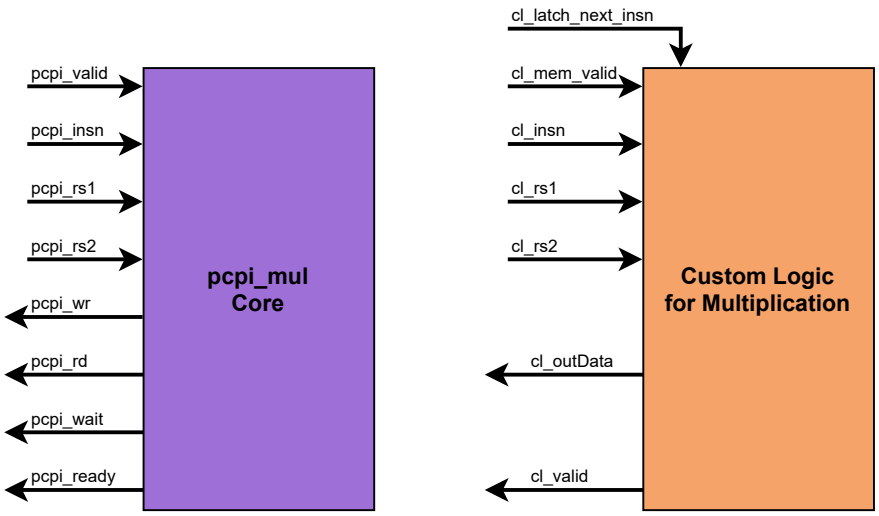

Figure 3.4: Comparison of the inputs and outputs for pcpi_mul core and custom logic multiplier. Signals on the same horizontal line are analogous.

of which entail the same multiplication operations, differing only in which portion of the result is returned and whether the operands are treated as signed or unsigned [7]. To make the two implementations as similar as possible, the logic of pcpi_mul is directly copied into the custom logic module to be used with TIGRA, where only a few changes must be made to adapt it to the new interface.

The left half of Fig. 3.4 depicts the PCPI signals utilized by the pcpi_mul core, while the right half shows the TIGRA interface used for the custom logic implementation. Most of the PCPI signals have direct TIGRA counterparts, such as *pcpi_valid* serving the same purpose as *cl_mem_valid*. These analogous signals appear on the same horizontal line.

The first difference in the TIGRA implementation involves the use of *cl_latch_next_insn*. This is necessary to ensure the *cl_insn* value latched and decoded by the custom logic is an actual instruction, as opposed to the other values seen during *lw* and *sw* instructions. pcpi_mul does not require an equivalent to *cl_latch_next_insn* since the PicoRV32 already includes logic to ensure *pcpi_insn* only receives instructions. Additionally, the decoding logic in the TIGRA version checks for the same custom opcode utilized in the other test cases, rather than the 0110011 opcode given for "M" instructions in the RISC-V specification. This is done to prevent the PicoRV32 from going to the trap state, since it does not recognize the "M" opcode and the processor-side logic for TIGRA does not include similar mechanisms to PCPI for stalling on an unrecognized instruction.

Table 3.4: Custom Instructions for Multiplication

Opcode	**Custom Instruction**	**Function**
000	mul	Multiply rs1 and rs2 and write lowest 32 bits of result to rd
001	mulh	Multiply rs1 and rs2 and write upper 32 bits of result to rd
010	mulhsu	Multiply signed rs1 and unsigned rs2 and write upper 32 bits of result to rd
011	mulhu	Multiply unsigned rs1 and unsigned rs2 and write upper 32 bits of result to rd

Accommodating the "M" opcodes would only require simple changes to the processor logic, though, similar to the ones described in Section 3.1.1 to recognize the custom opcode for TIGRA instructions.

The two other differences between multiplier implementations are the omissions of equivalents to *pcpi_wr* and *pcpi_wait* in the TIGRA version. The uses of *pcpi_wr* and *pcpi_ready* in pcpi_mul are identical, so *cl_valid* is sufficient to indicate completion of the multiplication process. Since the processor-side logic for TIGRA lacks a timeout mechanism, there's no use for an equivalent to *pcpi_wait*. Because the value of *pcpi_wait* is used in the logic for internal signals of pcpi_mul, the signal is changed from an output to an internal register in the TIGRA version.

Since the custom logic for multiplication otherwise matches that of pcpi_mul, its design implements the same four instructions, differentiated by their $funct3$ fields and listed in Table 3.4.

Chapter 4

Results

To test the designs for TIGRA and the custom logic use cases, Vivado 2020.1 is utilized to create testbenches representing specific instruction sequences and memory values, and then simulate the results to confirm they match expectations. Each instruction sequence includes the custom instructions implemented for the test case, as well as regular RISC-V instructions included before, after, and in some cases in between to confirm the custom instructions introduce no latency or unintended behavior. The sequences begin at the first memory location and end with a loop consisting of NOPs followed a *jal* instruction to ensure the processor only reads and attempts to execute the intended instruction data. Other values, to be loaded into the PicoRV32's registers via *lw* instructions, are included at memory locations beyond the last instruction.

4.1 AES

Among the files included with tiny_aes was a short testbench for encrypting a few states and checking whether the core returned the expected result. To confirm the custom logic design produces this same result given the same inputs, known state and key values are written to memory locations in the testbench for PicoRV32 with the AES custom logic. Four *lw*s near the start of the instruction sequence load the state and key into the PicoRV32's registers. The first four custom instructions are included next to copy the state and key into the custom logic's registers. This process can be seen in Figs. 4.1 and 4.2, where sections of the key or state appear first on cl_rs1 and cl_rs2 at the start of the *exec* stage after being decoded in ld_rs1. On the next clock cycle, the concatenated version

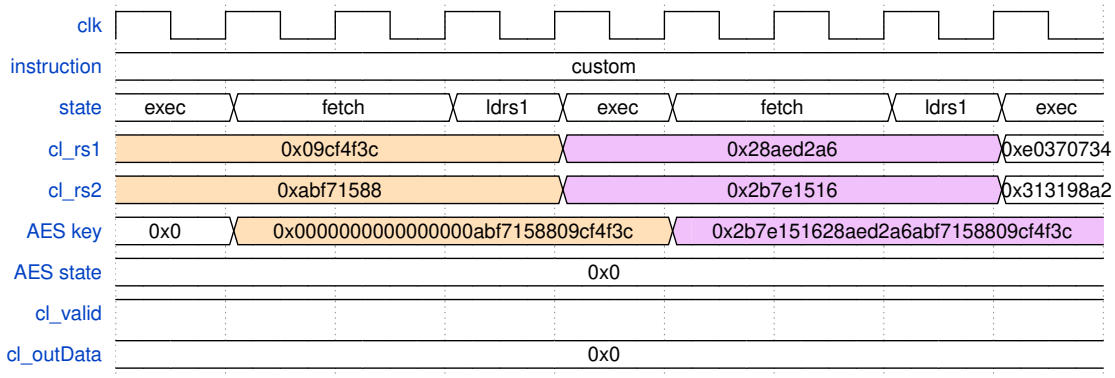

Figure 4.1: Simulation results for *wr_key_lo* and *wr_key_hi*.

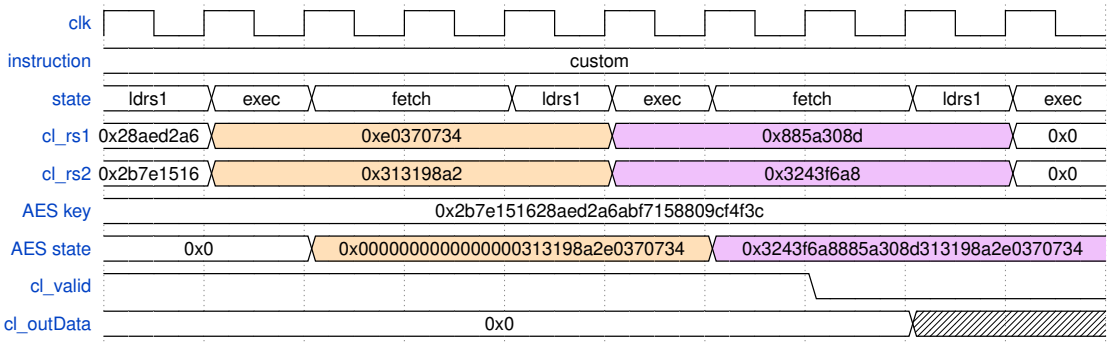

Figure 4.2: Simulation results for *wr_state_lo* and *wr_state_hi*. The latter also triggers the encryption process, indicated by the changes on *cl_valid* and *cl_outData*

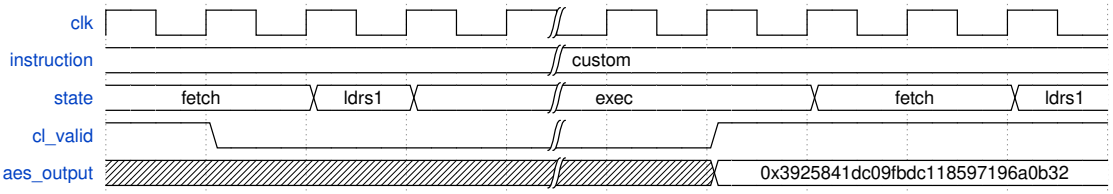

Figure 4.3: Simulation results for the AES encryption process triggered by the *wr_state_hi* custom instruction. The break in the diagram represents 16 cycles, during which none of the observed signals change.

of these values is written to either the key or state register. The value output to *cl_outData* is 0 for all these instructions, according to the custom logic's design described in Section 3.2.

Each sequence of *fetch*, *ld_rs*1, *exec* states in Figs. 4.1 and 4.2 represents the execution of one instruction. The four cycles required for each of the first three instructions match the timing of any regular ALU instruction and indicate that they do not introduce any new latency. The last of the four custom instructions shown (on the right half of Fig. 4.2) is *wr_state_hi*, which also

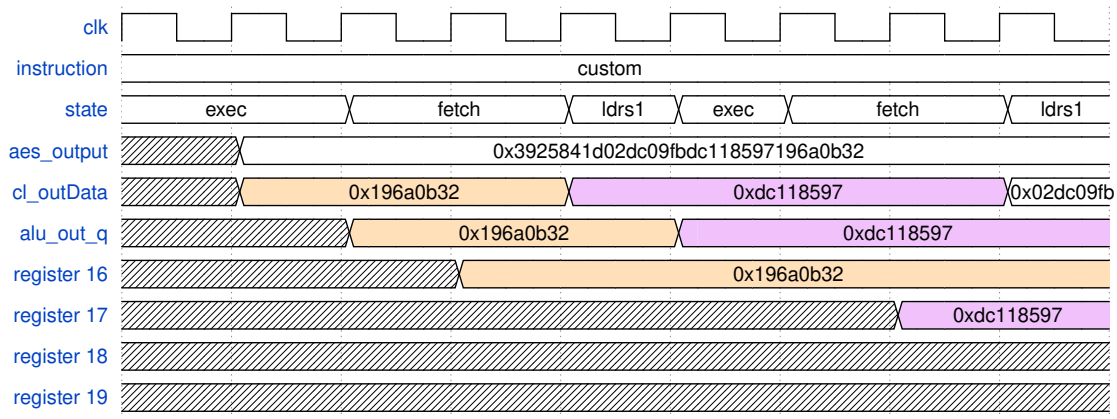

Figure 4.4: Simulation results for *rd_res_lo* and *rd_res_midlo*.

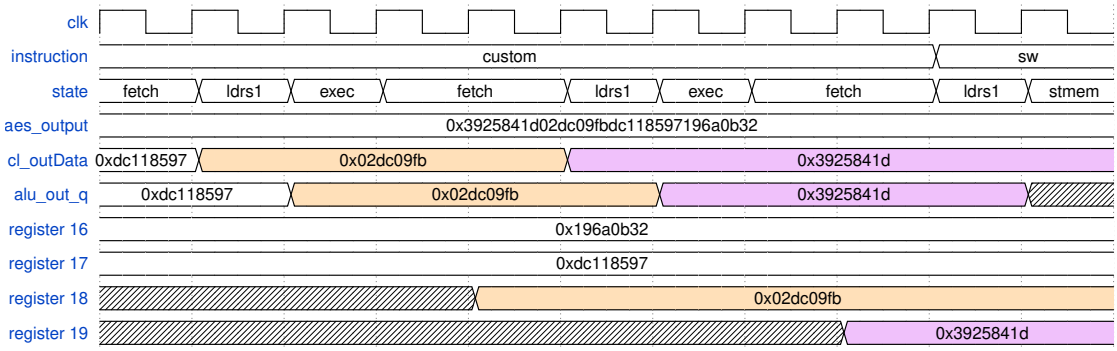

Figure 4.5: Simulation results for *rd_res_midhi* and *rd_res_hi*.

triggers the encryption process. This can be seen in the transition of *cl_valid* from high to low, and the change of *cl_outData* from 0 to an undefined value. The rest of this instruction's execution, including the 20-cycle stall in *exec*, is shown in Fig. 4.3. After the required number of cycles for tiny_aes, the encrypted result appears in the custom logic's internal signal *aes_output*, from which it will be written to *cl_outData*. The value seen there (0x3925...) matches the expected result for the given state and key, confirming correct operation of AES in the custom logic. The transition from *exec* to a normal, 2-cycle $fetch$ on the first cycle after *cl_valid* goes high indicates that the stall successfully completes, introducing no latency beyond the number of cycles required for encryption.

Following the instructions responsible for writing the key and state and triggering the encryption process, the other four custom instructions are included in the sequence to read the AES result back into the PIC32's registers. Figs. 4.4 and 4.5 depict the execution of these instructions, with *alu_out_q* receiving a new quarter of the result every four cycles and being written to the

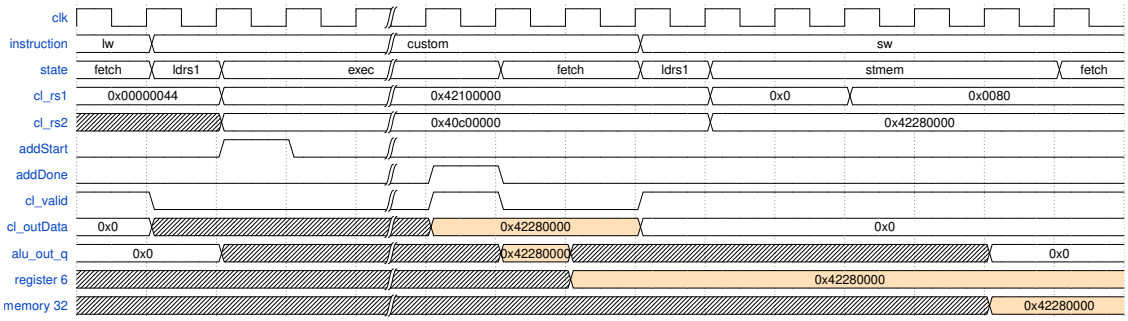

Figure 4.6: Simulation results for posit addition custom instruction, followed by a *sw* instruction that writes the result to memory. The break in the diagram represents 2 cycles, during which none of the observed signals change

register file during $fetch$. Though not shown, cl_valid remains high during all four instructions, indicating the absence of any stalls. Consequently, these instructions execute with no additional latency relative to a regular ALU instruction, as in Fig. 4.1.

4.2 Posit Arithmetic

Testing the behavior of the posit arithmetic custom logic also requires known values. The decimal values 18 and 3 are translated into their (32,6) posit representations of 0x4210_0000 and 0x40C0_0000 respectively to serve as the operands for addition and multiplication. 25 and 5 are similarly translated into 0x4248_0000 and 0x4120_0000 for use in the division operation. These 32-bit values are then placed in memory locations in the testbench. The instruction sequence is written to load posits into PicoRV32 registers with lw, then execute the custom add, multiply, and divide instructions to produce the posit equivalents of 21, 54, and 5 respectively.

Execution of the add instructions can be seen in Fig. 4.6. The internal signals $addStart$ and $addDone$ are included to indicate the start and end of the operation on the inputs, which appear on cl_rs1 and cl_rs2. The custom logic assigns $addDone$ directly to cl_valid in this case, based on the decoding of the instruction. On the same cycle that $addDone$ goes high, the expected result value of 0x4228_0000 (posit representation of 21) appears on $cl\_outData$, from which it propagates to alu_out_q and its destination register in the following two cycles.

The PicoRV32 stalls in the $exec$ stage while the posit addition takes place, ending only when cl_valid returns high. This results in an $exec$ with a duration of six cycles in this case. To confirm that the stalling custom instruction introduces no additional latency, such as a $fetch$ stage

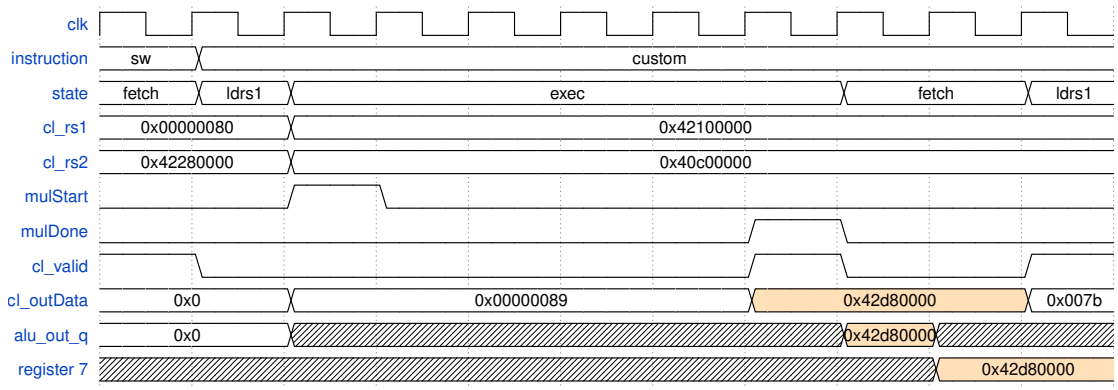

Figure 4.7: Simulation results for posit multiplication custom instruction.

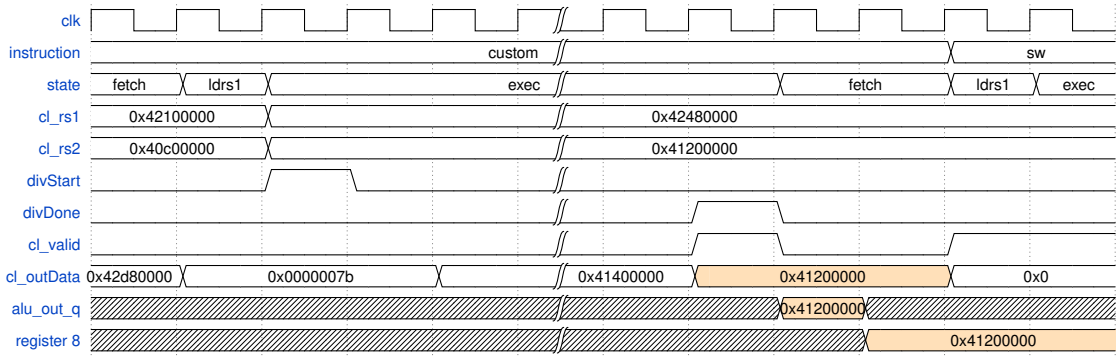

Figure 4.8: Simulation results for posit division custom instruction. The break in the waveforms represents 7 cycles, during which none of the observed signals change.

lasting more than two cycles, a *sw* instruction is included in the sequence between the custom add and multiply instructions. This *sw* can be seen in the right half of Fig. 4.6, where it writes the addition result to memory, executing with the expected timing. This confirms that the modifications to the PicoRV32 for triggering the decoding process after a stall function properly. *cl_valid* and *cl_outData*'s respective default values of 1 and 0 can also be seen during the execution of *sw*.

Execution of the posit multiply instruction, shown in Fig. 4.7, greatly resembles that of the addition operation. This time, the *mulStart* and *mulDone* signals and the multiplication result are utilized, according to the decoded instruction, and the result written to *cl_outData* is 0x42d8_0000 (posit representation of 54). The timing, however, is identical, with the result appearing in its destination register eight cycles after the arithmetic operation began.

The final custom instruction in the sequence is posit division. As indicated in [19], the division operation requires a greater number of cycles than the other two, and this is visible in the

simulation results of Fig. 4.8. A different pair of operands are utilized for this operation to confirm the expected behavior of the divison module, since leaving *cl_rs*1 and *cl_rs*2 unchanged from the previous instruction would cause the quotient to appear earlier than normal, though *divDone* would go high at the same time. The break in the timing diagram represents represents seven cycles, during which none of the observed signals change. At the end of a 13-cycle *exec* stage, *divDone* and *cl_valid* go high, allowing the result 0x4120_0000 (posit representation of 5) to be written to *alu_out_q*. This successful operation further confirms the functionality of the custom logic and the modified PicoRV32's ability to handle variable-length stalls.

Together, these results demonstrate how the TIGRA design described in Sections 3.1 and 3.1.1 and relatively simple custom logic are used to incorporate PACoGen posit arithmetic hardware and corresponding custom instructions into the PicoRV32 pipeline with no latency beyond the cycles required for the posit operations themselves. More broadly, the fact that the processor and the TIGRA interface required no changes between the AES and posit test cases, despite the differences in how the custom instructions were used, demonstrates TIGRA's flexibility and potential to simplify the design of custom logic and instructions.

4.3 Multiplication

To compare the PCPI and TIGRA implementations of co-processors to support RISC-V multiplication instructions, testbenches are created for each version. The first instantiates an unmodified PicoRV32 with the parameters set to enable PCPI and multiplication. The second instantiates the TIGRA-adapted PicoRV32 and the custom logic for multiplication based on the pcpi_mul core. Each includes the same sequence of instructions, differing only in the opcode used for the *mul* instruction, according to the design differences described in Section 3.4. To check whether the custom logic successfully uses *cl_latch_next_insn* to latch instructions, the *mul* is preceded in the instruction sequence with a *lw*, such that *cl_insn* contains data the custom logic should ignore by the time the multiplication instruction begins execution.

Fig. 4.9 depicts the simulation results of a *mul* instruction using pcpi_mul. It can be seen that the processor stalls in the *ld_rs*1 state until the product of *pcpi_rs*1 and *pcpi_rs*2 appears on *pcpi_rd* and *pcpi_ready* goes high. The break in the waveforms represents 33 cycles, for a total of 40 cycles in the instruction's execution. This matches the latency predicted in [20]. PCPI results are

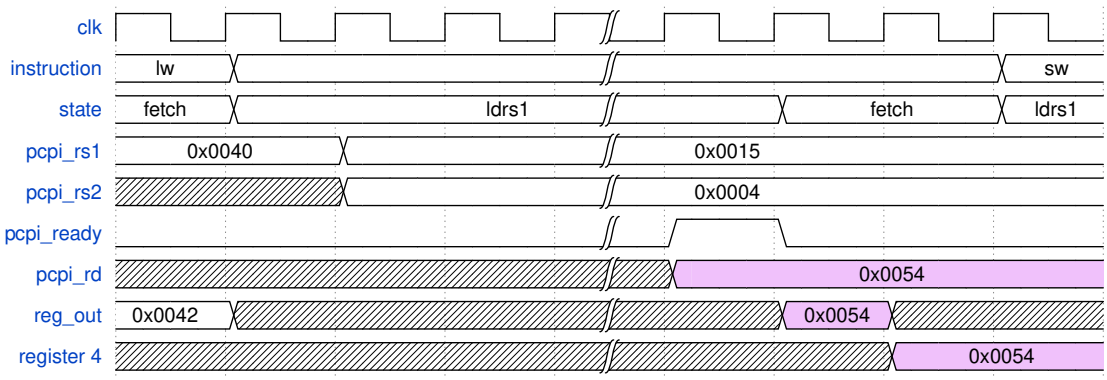

Figure 4.9: Execution of multiplication instruction using the pcpi_mul core.

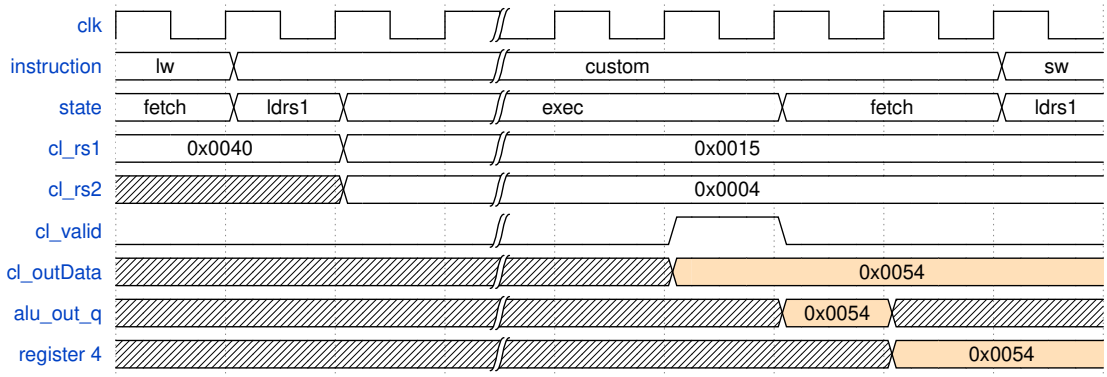

Figure 4.10: Execution of custom multiplication instruction using TIGRA-compatible custom logic.

written to *reg_out* rather than *alu_out_q* results, but its value is written to the destination register in a similar manner. The *Instruction* signal is blank for the duration of *mul* because the PicoRV32 does not contain flags to uniquely identify PCPI instructions.

The simulation results for the TIGRA custom logic multiplication are shown in Fig. 4.10. Though the signal names differ, it can be observed that the overall process greatly resembles that of pcpi_mul. The stall occurs in the *exec* state rather than *ld_rs*1, and the *Instruction* signal displays "custom" according to the modifications made to adapt the processor to TIGRA. The break in this diagram represents 33 cycles as well, indicating the multiplication executes in 40 cycles, just as in the PCPI equivalent. Thus, TIGRA introduces no additional latency and the custom logic successfully latches the appropriate instruction data, stalls properly, and produces the correct result without utilizing the PCPI logic built into the PicoRV32.

Table 4.1: Resource utilization by PCPI and TIGRA implementations of multiplier custom logic

Synthesized Modules	CLB LUTs	CLB registers
PicoRV32 base	886	574
PicoRV32_TIGRA	938	617
PicoRV32 + PCPI and co-processor	1170	853
PicoRV32_TIGRA + multiply CL	1146	857

4.4 Resource Utilization

To further evaluate the efficiency of TIGRA, Vivado is used to synthesize several designs for the Xilinx XCVU9P-FLGB2104-2-I FPGA. The synthesis results provide information about the device resources utilized by each design. The original, unmodified PicoRV32 core is shown in Table 4.1 to require 886 configurable logic block (CLB) lookup tables (LUTs) and 574 CLB registers. The core adapted to fit the TIGRA interface yields corresponding values of 938 and 617 respectively, indicating a relatively small increase in logic utilization. The two other synthesized designs are the base PicoRV32 with PCPI and the pcpi_mul core enabled, and the modified core with the TIGRA-compatible custom logic multiplier. As discussed in Section 3.4, these two designs implement the same logic for executing RISC-V multiplication instructions, so their logic utilization results provide useful insight into the area overhead required by TIGRA relative to an existing co-processor interface. As seen in Table 4.1, the PCPI design utilizes 1170 CLB LUTs and 853 CLB registers, while The TIGRA design requires 1146 and 857 respectively. These TIGRA results represent a 2.05% decrease in the number of CLB LUTs and just a 0.47% increase in number of CLB registers utilized. This favorable comparison can be attributed to the extra logic required by the PCPI design for catching illegal instructions and returning the processor to the $fetch$ state directly from ld_rs1, which is omitted in the TIGRA version of the PicoRV32. The results demonstrate the feasibility of implementing TIGRA in hardware, and that doing so does not come with a penalty of excessive area overhead.

Chapter 5

Conclusions and Discussion

This work presents and demonstrates the functionality of the TIGRA interface with the PicoRV32 RISC-V processor. It shows how the processor was adapted to fit the interface, allowing an accelerator to access the registers and necessary signals of the processor regardless of the accelerator's function. Three different custom logic test cases, including a total of 15 custom instructions, demonstrate the relative simplicity of implementing new or existing logic designs as efficient accelerators with TIGRA. Simulation results for the custom instructions show how TIGRA integrates an accelerator into the pipeline like an ALU or other execution unit to produce the correct results while adding no latency in communication between the processor and accelerator. The generic interface, the direct access to the processor's registers, and the support for arbitrarily-long stalls in the execution of custom instructions allow great flexibility in the design of custom logic. Logic utilization reports reveal a low area overhead required by TIGRA, and a favorable comparison with the Pico Co-Processor Interface provided with the PicoRV32. TIGRA thus simplifies the integration of custom accelerators and implementation of custom instructions without sacrificing performance in terms of clock cycles required for instruction execution or demanding disproportionate FPGA resources.

5.1 Future Work

The TIGRA-enabled PicoRV32 coupled with the custom logic designs described in this work can be implemented on hardware, such as Amazon Web Services' EC2 FPGAs, to further confirm and

quantify TIGRA's performance. With the work on PicoRV32 serving as a proof of concept, future work can implement TIGRA in more complex RISC-V processors and potentially establish standard practices for how to do so while accounting for variations in processor architecture. Related literature indicates significant interest in the Rocket Core in particular, making it a good candidate for testing and legitimizing TIGRA. More custom logic use cases can be explored, including more creative ways to utilize TIGRA's flexibility in decoding and executing custom instructions. Work could also be done to support additional custom instruction formats with TIGRA, such as instructions containing an immediate field.

Bibliography

[1] 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.

[2] 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.

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

[4] 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.

[5] R. Höller, D. Haselberger, D. Ballek, P. Rössler, M. Krapfenbauer, and M. Linauer. Open-source risc-v processor ip cores for fpgas — overview and evaluation. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–6, 2019.

[6] Krste Asanović and David A. Patterson. Instruction sets should be free: The case for risc-v. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, Aug 2014.

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

[8] Tony Chen and David A. Patterson. Risc-v geneology. Technical Report UCB/EECS-2016-6, EECS Department, University of California, Berkeley, Jan 2016.

[9] E. Matthews, Z. Aguila, and L. Shannon. Evaluating the performance efficiency of a soft-processor, variable-length, parallel-execution-unit architecture for fpgas using the risc-v isa. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8, 2018.

[10] 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.

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

[12] C. Tsuruta, T. Kaneda, N. Nishikawa, and H. Amano. Accelerator-in-switch: A framework for tightly coupled switching hub and an accelerator with FPGA. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2017.

[13] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. Risq-v: Tightly coupled risc-v accelerators for post-quantum cryptography. Cryptology ePrint Archive, Report 2020/446, 2020. `https://eprint.iacr.org/2020/446`.

[14] S. M. Noor and E. B. John. Resource shared galois field computation for energy efficient aes/crc in iot applications. *IEEE Transactions on Sustainable Computing*, 4(4):340–348, 2019.

[15] 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.

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

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

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

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

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