The Advantages of RISC-V and Custom Hardware

Aaron Dorney

Final Year Project - BSc in Computer Science

Supervisor: Professor John Morrison

Department of Computer Science
University College Cork

April 25, 2022

1 Abstract

The open source software community has become extremely prevalent in the field of computing as it promotes collaboration and the sharing of information. The proprietary nature of instruction sets means that hardware designs can not be openly shared. This also means that the amount of of specialised hardware available is limited as it is only offered by vendors with off the shelf solutions. There are a number of fields and businesses that could greatly benefit from more varied custom hardware designs such as data centers that require hardware that provides both low power consumption and low execution times. RISC-V is an open source instruction set that aims to promote open source hardware in both industry and academia. The aim of this project was to investigate the tools for designing custom RISC-V hardware as well the potential benefits that hardware acceleration can provide to certain workloads.

2 Declaration of Originality

Signed: Aaron Dorney

Date: 25/4/22

3 Acknowledgements

I'd firstly like to thank my supervisor Professor John Morrison for taking on my project and for his immense help on deciding the direction the project. His feedback throughout the course of the project was crucial in helping to ensure its goals were met in a timely fashion. I'd also like to thank some of my fellow students for providing feedback whilst discussing my project and for proposing potential topics to explore.

Contents

1 Abstract **i**

2 Declaration of Originality **ii**

3 Acknowledgements **iii**

1 Introduction **1**

1.1 Limitations of Proprietary ISAs 1

1.1.1 A Lack of Open Source Contribution 1

1.1.2 Data centres 1

1.1.3 Heterogeneous SoCs 2

1.1.4 Domain Specific Processors 2

1.2 RISC-V 3

1.3 Goals 3

1.4 Approach 3

1.5 Project Structure 4

2 Tools and Relevant Background **4**

2.1 CISC vs RISC 4

2.2 The RISC-V Architecture 5

2.3 Chipyard[7] 6

2.4 Hardware Description 6
2.5 RTL Simulation 7
2.6 Rocket Core 7
2.7 Custom Opcodes 8
2.8 GCC 9
2.9 GTKWave 9

3 Design Workflow **10**

4 Implementation **14**

4.1 PopCount 14
4.1.1 Analysis 14
4.1.2 Implementation 14
4.1.3 Evaluation 15
4.2 Fast Inverse Square Root 16
4.2.1 Analysis 16
4.2.2 Implementation 17
4.2.3 Evaluation 17
4.3 Regular Expressions 20
4.3.1 Analysis 20
4.3.2 Implementation 22
4.3.3 Evaluation 24

4.4 Vector Operations 26

4.4.1 Analysis 26

4.4.2 Implementation 29

4.4.3 Evaluation 34

5 Evaluation **37**

6 Conclusions **38**

6.1 Reflection 38

6.1.1 Issues 38

6.1.2 Insights 40

6.2 Future Work 41

6.2.1 Peripherals 41

6.2.2 FPGA 41

6.2.3 Further Regular Expression Work 42

6.2.4 SiFive 42

6.2.5 ROCC Modification 42

6.2.6 Vector-SIMD instructions 42

List of Figures

1 Encoding of an R-type instruction 8

2 Basic skeleton for RoCC class 10

3 Logic for a basic accelerator 11

4 An SoC configuration 12

5 A bare metal C benchmark 12

6 Results of a benchmark 13

7 Waveform output from a benchmark 13

8 Graph of *popcount* benchmark results 15

9 Graph of fast inverse benchmark results (Note: graph uses log scale) 18

10 Graph of fast inverse benchmark results with no pipelining optimisations 19

11 State machine for regular expression parser 22

12 Results of regular expression benchmark 24

13 Illustration of a SIMD operation 26

14 Illustration of a vector processor operation 27

15 State machine for matrix multiplication accelerator with limited resources 29

16 State machine for matrix multiplication accelerator with a large amount of internal memory 30

17 State machine for matrix multiplication accelerator that uses transposition 32

18 State machine for matrix multiplication accelerator that more closely resembles a vector processor 33

19 Matrix multiplication accelerator results for 2-16 (Note: graph uses log scale) 34

20 Matrix multiplication accelerator results for 16-64 (Note: graph uses log scale) 36

1 Introduction

1.1 Limitations of Proprietary ISAs

1.1.1 A Lack of Open Source Contribution

Over the past few years the open source software community has seen immense success with many companies both contributing to and making use of open source projects. Making a project open source allows for rapid development and improvement as it allows for anyone to contribute both work and feedback to the project. While open source software has become widely used and accepted, open source hardware has seen less success. The majority of instruction sets are proprietary, meaning any designs implementing a proprietary architecture such as x86 or Arm cannot openly disclose their design. This serves as a significant roadblock to the creation of an open source hardware community.

1.1.2 Data centres

The power consumption of data centres has become a serious concern in recent years as it leads to higher operation costs and a larger carbon footprint. Many believe that data centers will begin transitioning to a more heterogeneous architecture in the future that makes use of specialised accelerators and CPUs that are tailored to a specific task[1]. Ideally these specialised cores and accelerators would also prioritise low power consumption. A potential roadblock to this approach is a lack of diversity in terms of specialised hardware as most hardware vendors only provide general purpose off the shelf solutions. If the synthesis and design of custom hardware were more widely available it would increase the variety of available hardware and ideally allow data centres to reduce both power consumption and execution times.

1.1.3 Heterogeneous SoCs

SoCs (System on Chip) involve grouping all the resources of a computer such as the CPU, GPU and memory into a single chip. SoCs are most commonly used in smartphones as they require multiple specialised units to be placed within a limited amount of space. SoCs have also seen use in personal computers with the release of Apples' M1 chip. Since SoCs consist of numerous individual units it is not always possible for a manufacturer to build an SoC with a single ISA (Instruction Set Architecture). As a result manufacturers have to source components from multiple companies, with their own seperate licensing agreements. This can lead to a number of design complications as it requires multiple devices using different ISAs to communicate effectively. Using a single, extensible ISA would simplify communication between different units while also simplifying the overall design process.

1.1.4 Domain Specific Processors

Reducing the number of transistors on a chip now faces challenges at the atomic level and many believe that Moore's law has come to an end[2]. As a result alternative approaches must be taken to increase the compute power of our devices. One such alternative is domain specific processors[3]. The idea behind a domain specific processor is to create a processor tailored to a specific task or field. An example of this approach would be Google's tensor processing units that are designed for handling AI workloads. While creating a domain specific processor can provide significant performance improvements, designing one can be a significant undertaking. This is due to the fact that domain specific processors require a specialised instruction set. As a result companies have to design their own instruction set from scratch which requires a significant amount of time and resources. The ability to create a domain specific processor using a predefined instruction set that is small but extensible would drastically simplify the design process in certain cases

1.2 RISC-V

RISC-V is an open source ISA that was developed at the University of California Berkeley and has continued to grow in popularity since its inception due to its potential for both academic and commercial use. The number of languages and operating systems supporting the architecture continues to grow and there are even consumer level RISC-V microcontrollers available. Since RISC-V is open source it allows hardware developers to share their designs freely and there are already multiple open source RISC-V CPU designs such as the Rocket core[4] and BOOM[5] core (Berkeley Out of Order Machine). The ISA is extensible and even provides four opcodes specifically for custom instructions. This allows designers to adapt their instruction set to suit a specific need.

1.3 Goals

The primary motivation for this project was to investigate the RISC-V ISA as well as the ecosystem of hardware design tools surrounding it. Additionally this project aimed to explore the potential benefits that custom hardware can provide in terms of potential speedup while also illustrating that tailoring hardware to a specific task can have multiple potential approaches with varying economies of scale.

1.4 Approach

In order to investigate the RISC-V ISA and tools while also illustrating the performance benefits of custom hardware, the project was divided into four topics that can benefit from hardware acceleration. Two topics were selected for a custom instruction and two more were selected for an accelerator. Custom instructions are fast and have minimal communication overhead but are limited in terms of versatility. Accelerators on the other hand can handle complex and large workloads but incur a communication overhead with the CPU.

The purpose of these designs was to highlight certain aspects of the hardware design process while also showing the potential speedup that custom hardware can provide to specific workloads. As such these designs were created to serve as a proof of concept rather than a fully fledged product with drivers and features such as error detection.

1.5 Project Structure

Chapter 2 will cover the tools used for the project as well as some relevant background on certain topics. Chapter 3 provides a brief showcase of the workflow involved with the project through the use of a basic accelerator design. Chapter 4 is broken into four empirical studies, based on specific topics. For each topic an analysis of the subject will be provided, which will be followed by the implementation of one or more hardware designs centred around the underlying subject. The design(s) will then be evaluated using a benchmark, the results of which will then be discussed. Chapters 5 and 6 will conclude the report with an overall evaluation of the body of work followed by a discussion on the outcomes of the project as well as some potential topics for future work.

2 Tools and Relevant Background

2.1 CISC vs RISC

Originally it was standard for instruction sets to use complex instructions that took multiple cycles to complete. This design approach is what would now be referred to as a CISC (Complex Instruction Set Computer) design. In 1980 a team at IBM led by the researcher John Cocke completed work on the first computer using a RISC (Reduced Instruction Set Computer) design known as the IBM 801. The motivation behind the design was due to observations made by Cocke in a previous project in which the instruction set being used was reduced by removing instructions that were seldom used and that took a large number of cycles to complete. It was found that this led to a significant speedup in execution time as breaking the complex operations of the removed instructions into multiple simple instructions allowed for more compiler optimisations to take place.

A key difference between the RISC and CISC design philosophies is where reductions in execution times are achieved. In a CISC architecture hardware is the primary means of reducing execution times which comes in the form of specialised instructions. A RISC architecture on the other hand relies more on software to achieve improved performance which comes in the form of more efficient code and compiler optimisations.

The smaller instruction set of a RISC design also acts as both a strength and a weakness. The smaller instruction set provides many benefits such as more potential for compiler optimisations and a reduction in the learning curve for designers to become accustomed to the architecture. The overall design philosophy of RISC also prevents instruction set bloat in which an instruction set has countless instructions that are seldom used and which most people do not even know the purpose of. This smaller instruction set has one key drawback which is program size. A program written in a RISC instruction set will typically be considerably larger than one written in a CISC instruction set as a series of operations in the RISC program could be described by a single CISC instruction. This can be a significant concern in scenarios where the available RAM in a system is limited.

2.2 The RISC-V Architecture

The RISC-V ISA has a number of extensions and ISA bases(e.g., 64 bit and 32 bit) which are classed as either open or ratified. Open extensions are subject to change however ratified extensions are frozen. This ensures stability which allows designers to begin using an extension or base without worrying about sudden changes hindering the design process. This also ensures that the base ISA and its extensions don't continually grow and eventually become more akin to a CISC architecture.

One of the primary appeals of RISC-V as an ISA is its support for extensibility. This allows for architecture designs that make full use of the benefits of a RISC architecture while also incorporating elements of CISC design in the form of custom instructions. This results in an instruction set that primarily uses simple instructions that reap the benefits of compiler optimisations but also provides more complex instructions that no amount of optimisation can compete with.

Since the instruction set is open source it also provides a number of security advantages. Since the details of most ISA's aren't openly disclosed, this means that hardware designers have to rely on security by obscurity. This operates on the basis that if a security vulnerability exists in an architecture, it is unlikely to be found if no one knows the specifics of the architecture. This approach has not seen much success with new hardware based attacks being discovered on a regular basis such as the spectre attack[6]. Openly disclosing designs would potentially allow for security flaws to be discovered far more quickly and potentially even be resolved during the design process rather than with a hasty software patch.

2.3 Chipyard[7]

Chipyard is a framework for SoC design developed by UC Berkeley. It consists of a number of tools that aid in the design, simulation and testing of an SoC. It also contains a number of open source accelerators and processors such as the previosuly mentioned Berkeley Out Of Order Machine (BOOM) or the NVIDIA Deep Learning Accelerator (NVDLA).

2.4 Hardware Description

Hardware description languages are a means of programmatically describing hardware. These languages provide a much needed layer of abstraction for hardware design as it allows one to describe numerous gates and logic circuits in a single line. The most commonly used hardware description languages are VHDL and Verilog. The majority of this project was written using a hardware description language called Chisel.

Chisel is a hardware description language embedded in Scala. Its main appeal is that it makes use of the object oriented and functional programming paradigms of Scala. An additional set of tools called the FIRRTL compiler can then be used to transpile Chisel into synthesizable Verilog and produce an RTL simulator. This makes Chisel a more viable language as most hardware synthesis tools support Verilog.

2.5 RTL Simulation

An RTL (Register Transfer Level) simulator provides a gate level simulation of a hardware design. This means that the simulator provides cycle accurate hardware simulation but does not take space, power or thermal factors into account. Since these simulators are a gate level simulation they do not provide any abstraction from the hardware and are only designed to run bare metal RISC-V binaries. Alternatively a tool called the proxy kernel can be used as an intermediary between an RTL simulator and a RISC-V binary designed to run on a kernel. This tool was rarely used during the project however as it added an unwanted layer of abstraction from the hardware.

2.6 Rocket Core

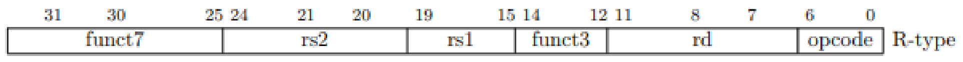
The Rocket core is an open source, in-order processor designed in Chisel that uses the 64 bit RISC-V instruction set. The core also makes use of an architecture known as the modified Harvard architecture which divides the L1 cache of each core into an instruction cache and a data cache. This allows for both an instruction and data to be fetched by the core in a single cycle. All of the designs for this project made use of a Rocket core due to its support for both extensions to the instruction set and accelerators.

The Rocket core supports the RoCC (Rocket Custom Coprocessor) interface which allows for accelerators to be added to an SoC with ease. The RoCC interface allows the programmer to interface with other units such as the MMU to read from or write to memory or the FPU to perform floating point operations. This means that a RoCC accelerator can perform complex operations but also incurs a communication overhead with the CPU. Furthermore the CPUs' execution pipeline stalls completely when waiting for an accelerator response which can also have an impact on execution performance.

Instruction set extensions were created using the SCIE (Simple Custom Instruction Extension) which the Rocket core supports. There is little to no documentation on the SCIE but it appears to either add an instruction to the ALU or create a small unit specifically for executing custom instructions. While the RoCC interface provides versatility at the cost of communication overheads, the SCIE provides little communication overhead at the cost of versatility. This is because the SCIE can only be used for simple operations and cannot interact with other units. The default implementation of the SCIE was written in Verilog and although they could have been written in Chisel, any SCIE instructions for the project were written in Verilog to provide some exposure to another hardware description language.

2.7 Custom Opcodes

The RISC-V ISA contains four opcodes that can be used for customisation. Instructions using these opcodes can be used to execute an extension of the instruction set or communicate with an accelerator. For the sake of clarity any use of the term "*custom instruction*" in this report will be referring to an extension of the ISA rather than the use of an accelerator.



source: RISC-V instruction set manual[8]

Figure 1: Encoding of an R-type instruction

The RISC-V ISA has multiple instruction formats to suit certain operations such as immediate instructions or load instructions. The only format used in this project was the R format which takes two source registers and a destination register. Additionally, R format instructions are encoded with two *funct* values which can be used to dictate the specific operation to be carried out. In the case of RoCC accelerators, the three *funct3* bits are used to specify which of the source and destination fields contain relevant information.

2.8 GCC

GCC provides a RISC-V compiler which was used to compile benchmarks written in C and run them on RTL simulators. The compiler was primarily used to create bare metal RISC-V binaries but was also used to generate assembly dump files to determine what instructions a binary would be using. This was done to ensure any custom instructions were encoded correctly when debugging and to determine whether or not baseline tests were using certain beneficial instructions (e.g., the square root instruction) for a particular benchmark.

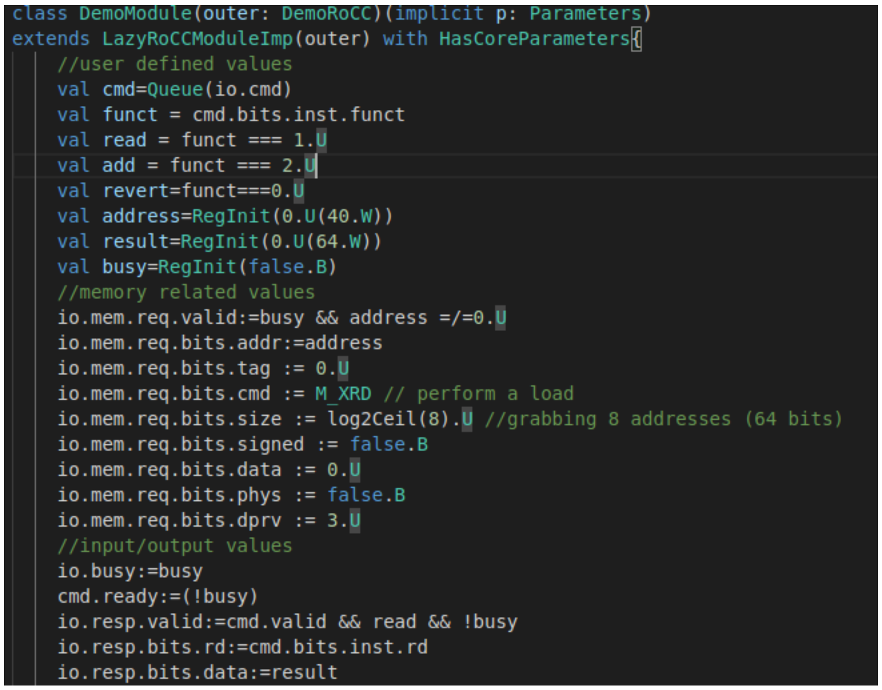
2.9 GTKWave

GTKWave is an application for viewing waveforms. The RTL simulators generated for this project can be used to generate a waveform file after executing a binary. These waveforms contain all the values and signals in the design and how they changed each cycle. These waveforms were essential for debugging design problems as they could be used to identify where something went wrong in execution and what values or registers were causing the issue.

3 Design Workflow

The following will be a brief overview of the workflow involved with creating a very basic accelerator that requests the value at an address and returns that value incremented by one. The design of a custom instruction has a similar workflow but using a different set of classes.

When creating an accelerator the first step is to write a module class that captures its functionality.



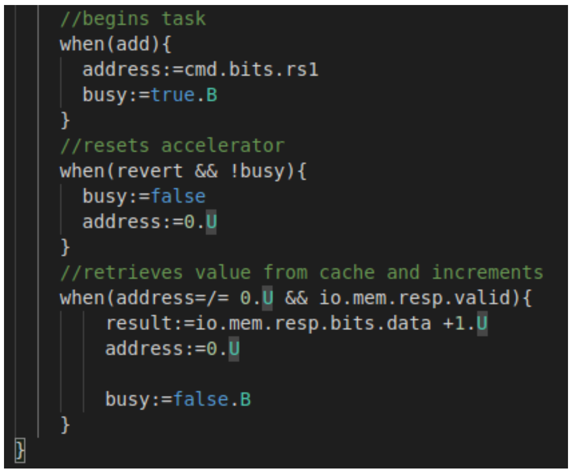
```
class DemoModule(outer: DemoRoCC)(implicit p: Parameters)
extends LazyRoCCModuleImp(outer) with HasCoreParameters{
    //user defined values
    val cmd=Queue(io.cmd)
    val funct = cmd.bits.inst.funct
    val read = funct === 1.U
    val add = funct === 2.U
    val revert=funct===0.U
    val address=RegInit(0.U(40.W))
    val result=RegInit(0.U(64.W))
    val busy=RegInit(false.B)
    //memory related values
    io.mem.req.valid:=busy && address =/=0.U
    io.mem.req.bits.addr:=address
    io.mem.req.bits.tag := 0.U
    io.mem.req.bits.cmd := M_XRD // perform a load
    io.mem.req.bits.size := log2Ceil(8).U //grabbing 8 addresses (64 bits)
    io.mem.req.bits.signed := false.B
    io.mem.req.bits.data := 0.U
    io.mem.req.bits.phys := false.B
    io.mem.req.bits.dprv := 3.U
    //input/output values
    io.busy:=busy
    cmd.ready:=(!busy)
    io.resp.valid:=cmd.valid && read && !busy
    io.resp.bits.rd:=cmd.bits.inst.rd
    io.resp.bits.data:=result
```

Figure 2: Basic skeleton for RoCC class

The first section in figure 2 contains user defined values. These consist of wires and registers that will dictate how the accelerator operates. The *cmd* value sets up a queue for incoming requests and the *funct* values are a sequence of bits in the encoded instruction that can be used to dictate the instructions' purpose.

The next section contains wires that are used for memory operations. Whenever a memory request becomes valid, it will be sent. The memory request interface requires specific information about the target address such as the number of bytes to request, whether the address is physical or virtual and whether the value being requested is signed or unsigned. The interface also supports write operations which can be performed by changing the *cmd* wire and placing the value to write in the data wire.

The bottom set of wires dictate when to process the next request and when to return a result. In this case the accelerator will process the next request provided the accelerator is not busy and currently executing a task. If the device receives a read request then it will copy the contents of the result register to an output register that was specified in the request.

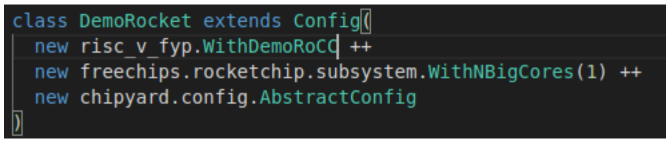


```
    //begins task
    when(add){
      address:=cmd.bits.rs1
      busy:=true.B
    }
    //resets accelerator
    when(revert && !busy){
      busy:=false
      address:=0.U
    }
    //retrieves value from cache and increments
    when(address=/= 0.U && io.mem.resp.valid){
        result:=io.mem.resp.bits.data +1.U
        address:=0.U

        busy:=false.B
    }
}
```

Figure 3: Logic for a basic accelerator

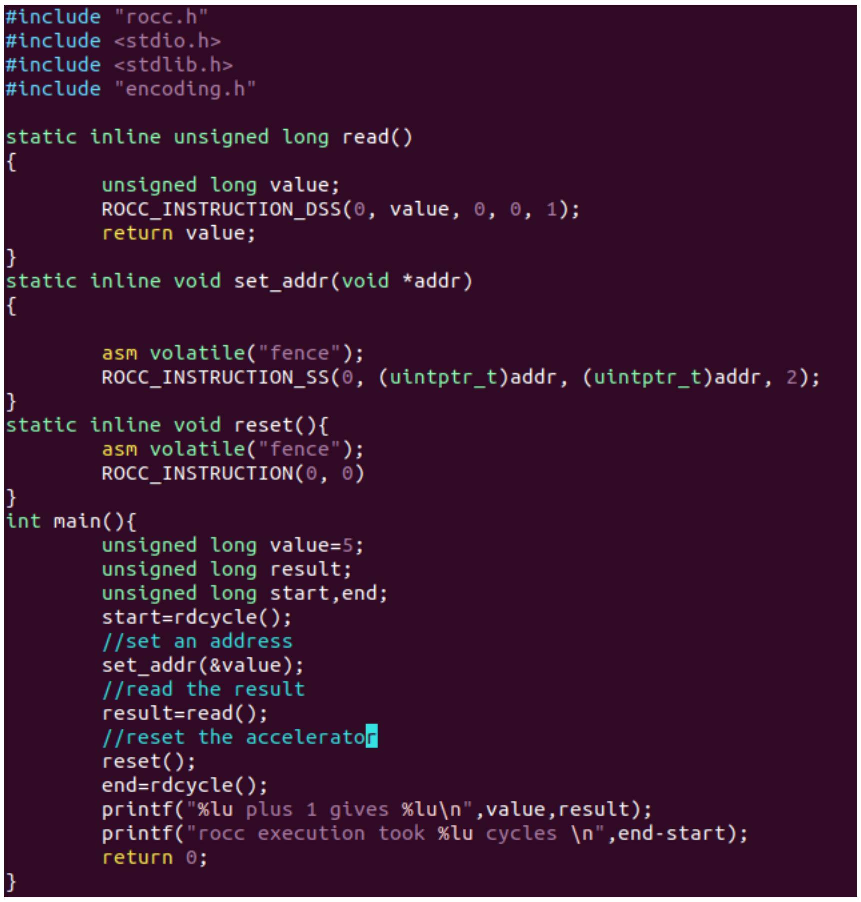
Figure 3 shows the actual logic for the accelerator. Once the accelerator receives an instruction with the *funct* value for add it will extract an address value from the instruction and store it in the *address* register. The accelerator then sets itself to busy which will block any incoming requests until it finishes its task. A cache request is then sent to retrieve the contents of the address and once a response is received, the desired output is placed in the result register. The busy register is then set to false and the result will be returned once an instruction with the *funct* for read is received. Finally an instruction with the *funct* for revert is used to ensure the *funct* wire is set low again which prevents the accelerator from attempting a new task.



```
class DemoRocket extends Config(
  new risc_v_fyp.WithDemoRoCC ++
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig
)
```

Figure 4: An SoC configuration

Once the design is complete, the module class is encapsulated in a series of additional classes and finally placed in a config file. A makefile provided by Chipyard is then used to turn this config into an RTL simulator.



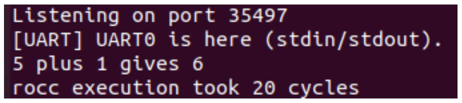
```
#include "rocc.h"
#include <stdio.h>
#include <stdlib.h>
#include "encoding.h"

static inline unsigned long read()
{
        unsigned long value;
        ROCC_INSTRUCTION_DSS(0, value, 0, 0, 1);
        return value;
}
static inline void set_addr(void *addr)
{

        asm volatile("fence");
        ROCC_INSTRUCTION_SS(0, (uintptr_t)addr, (uintptr_t)addr, 2);
}
static inline void reset(){
        asm volatile("fence");
        ROCC_INSTRUCTION(0, 0)
}
int main(){
        unsigned long value=5;
        unsigned long result;
        unsigned long start,end;
        start=rdcycle();
        //set an address
        set_addr(&value);
        //read the result
        result=read();
        //reset the accelerator
        reset();
        end=rdcycle();
        printf("%lu plus 1 gives %lu\n",value,result);
        printf("rocc execution took %lu cycles \n",end-start);
        return 0;
}
```

Figure 5: A bare metal C benchmark

Next a bare metal test is written in C to test the accelerators' functionality. Header files provided by Chipyard are used to measure cycle counts and also provide macros for producing the inline assembly for a custom instruction.



```
Listening on port 35497
[UART] UART0 is here (stdin/stdout).
5 plus 1 gives 6
rocc execution took 20 cycles
```

Figure 6: Results of a benchmark

Finally the C file is compiled into a *.riscv* binary and passed to the simulator which runs the binary and outputs the results of the test.

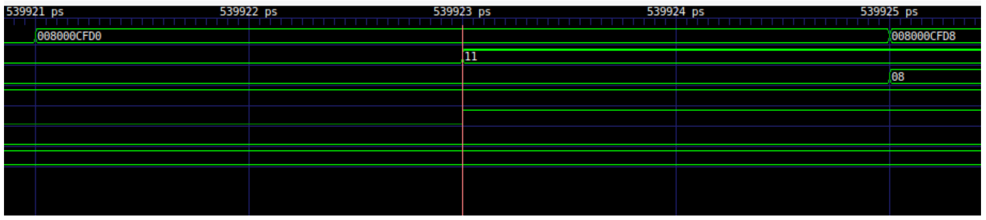


Figure 7: Waveform output from a benchmark

Should the benchmark not behave as intended, additional flags can be used with the simulator to generate a waveform file. This file can then be used to display the value of wires and registers on a given clock cycle.

4 Implementation

4.1 PopCount

4.1.1 Analysis

Popcount is an instruction that has numerous uses but is not present in the base RISC-V instruction set or any official extensions. The instruction simply counts the number of ones in a binary sequence. It has numerous uses such as performing matrix multiplication on binary matrices or evaluating hamming codes for error correction in networking.

4.1.2 Implementation

The rocket chip codebase provides a file to place SCIE implementations which will then be used in default configurations. This proved quite cumbersome as testing multiple designs would involve cutting and pasting source code into this file before beginning RTL synthesis. Since Chisel is an object oriented language, attempts were made to extend the class hierarchy of the rocket chip codebase, to allow for custom SCIE implementations to be passed to the top level constructor which would allow for separate configs to be created using different SCIE implementations. These attempts proved fruitless as the Rocket chip class hierarchy was quite rigid and used a lot of constants that did not support lazy evaluation. As a result of this the only solution was to copy multiple classes from the rocket chip codebase into the project and alter them slightly to accept an SCIE implementation as input for a config. While it may have been cleaner to create a fork of the rocket chip repository implementing these changes, forcing a clean copy of Chipyard to use this fork instead of the main branch would be far more cumbersome than simply using the adapted files.

The design for the *popcount* instruction was quite simple and involved iterating over the value of the register and checking if each bit was zero or one. A RoCC accelerator was also created to observe the performance differences between a custom instruction and a custom accelerator.

4.1.3 Evaluation

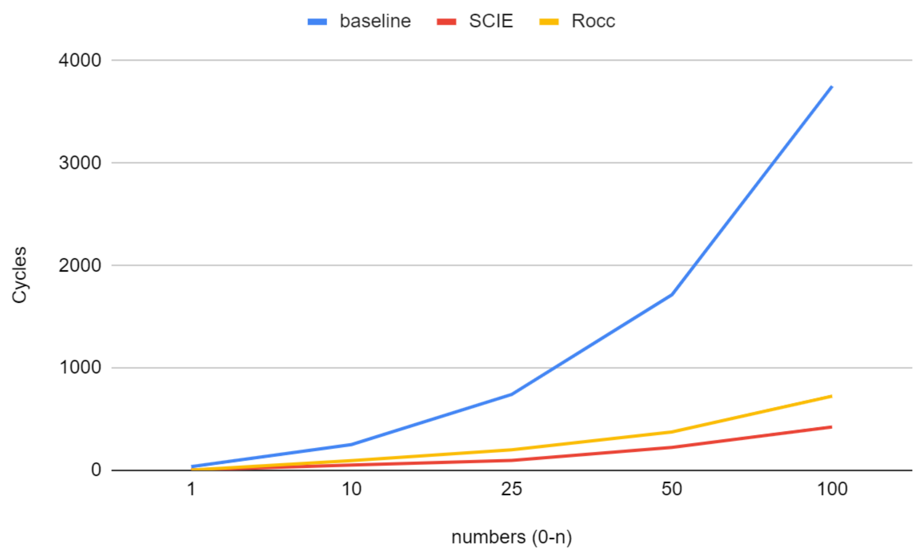


Figure 8: Graph of *popcount* benchmark results

A benchmark was written to count the number of set bits for a range of numbers. As the range size increased the performance difference between the baseline software implementation and the two hardware implementations became more apparent. Additionally the communication overhead incurred by the accelerator became more apparent as its performance began to diverge from the custom instruction.

4.2 Fast Inverse Square Root

4.2.1 Analysis

Vector normalisation plays an extremely important role in 3D graphics as it is required to simulate lighting. One of the most computationally intensive aspects of vector normalisation involves calculating the inverse square root of a number. This simple computation needs to be calculated millions of times per second to implement 3D physics in games. During the early 1990s this was a serious bottleneck for games developers as compute power was limited and the performance of floating point operations was even more so.

Quake 3 arena was a multiplayer first person shooter released in 1999 and was praised for its game design and performance at the time of release. In 2005 ID Software, the developers of the game, open sourced the source code for the game's engine. An algorithm in the source code quickly gained notoriety and was dubbed the "fast inverse square root" algorithm.

The algorithms' genius lies in the fact that it eliminates the most computationally intensive aspect of calculating an inverse square root; division. Floating point division is considerably slower than multiplication and the regular method for calculating the inverse square root consists entirely of division operations. This is because calculating the square root of a number requires multiple rounds of division.

While the mathematics behind the algorithm is quite interesting it also requires knowledge of bit manipulation, IEEE floating point notation, address pointers and a number of mathematical concepts. The simplest explanation of the algorithm is that it uses bit manipulation on the numbers' exponent (the power it is raised to) to calculate an estimation for the inverse square root of the number. Then something called Newton's method is applied to the estimation which improves the estimations' accuracy.

It should also be noted that nowadays vector normalisation for graphics is typically offloaded to a GPU or SIMD instruction as it typically involves computing the inverse square root of multiple numbers and benefits greatly from parallelisation. That being said, scalar instructions can still be useful when alternatives like a GPU or SIMD instruction are not available. Furthermore the fast inverse square root algorithm was used to circumvent the limitations of hardware at the time of its inception and observing how it performed as a hardware implementation was an area of interest.

4.2.2 Implementation

The initial intention was to implement the entire algorithm but this was later changed to just the operations on the exponent. This was because verilog has no native support for floating point arithmetic meaning the custom instruction would need to implement floating point subtraction and multiplication to perform Newton's method. This would drastically increase the instructions' complexity and also seemed like it would have very little benefit to performance as the required operations already existed in the FPU.

4.2.3 Evaluation

When using benchmarks to verify the functionality of the accelerator, it became apparent that the compiler did not have printf support for floats in bare metal tests. Initially the aforementioned proxy kernel was used to ensure the outputs of the instruction were correct. This proved quite tedious as the proxy kernel drastically increased the running time of benchmarks which slowed down the debugging process. The benchmarks were then changed to print results as unsigned longs and run without the proxy kernel. These results were then verified with online tools by converting the unsigned longs to binary and converting the binary to floating point notation.

Once the custom instruction had been implemented three benchmarks were written to calculate the inverse square roots of a given range. The first benchmark made use of the square root instruction. The second benchmark implemented the software version of fast inverse square root and the third benchmark used the hardware implementation instead. Originally a fourth benchmark was used that did not make use of the square root instruction and instead used other instructions to achieve the same result. This benchmark was removed as it had drastically worse performance than all the other benchmarks. Furthermore a square root instruction is a common instruction in any device that supports floating point operations which made further use of the benchmark seem redundant.

The benchmark using only the square root instruction had drastically worse performance than the other two benchmarks. The hardware implementation of fast inverse was typically only a few cycles faster than the software implementation in any given test which was unexpected.

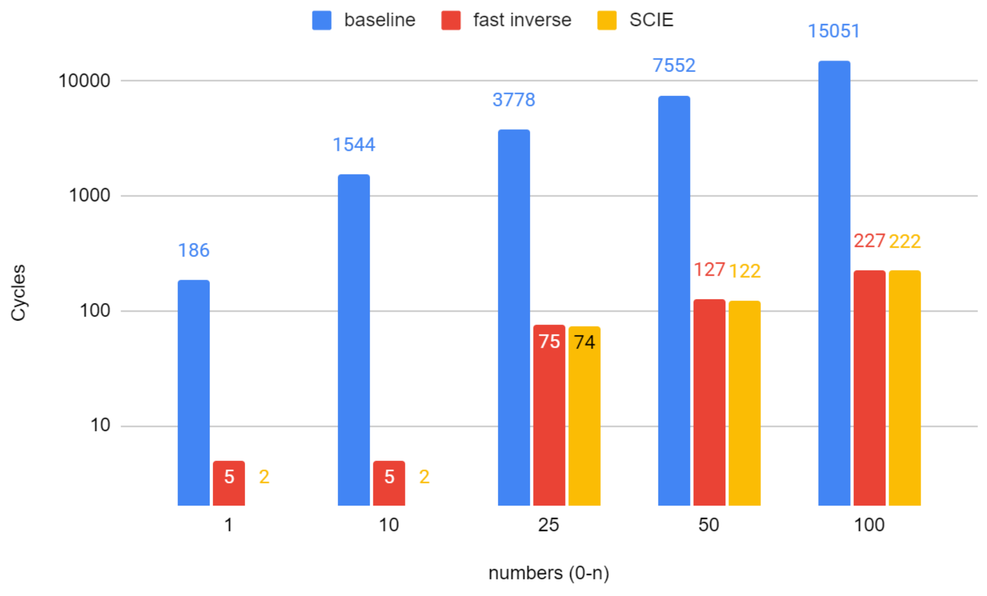


Figure 9: Graph of fast inverse benchmark results (Note: graph uses log scale)

Upon inspecting the results it became apparent that the compiler was influencing the results of the benchmarks. Some benchmarks with very small ranges had unexpected results such as one task and ten tasks taking the same number of cycles to compute. It appeared that the compiler was reordering instructions outside of the functions measuring cycle counts to improve *pipelining*.

The process of retrieving an instruction and executing it in a CPU is referred to as the *execution pipeline*. This pipeline is broken into multiple stages (five in the case of a Rocket core) such as fetching an instruction or using an execution unit. Instruction piplining involves passing a new instruction into the first stage of the pipeline when the instruction ahead of it moves on to the second stage. Certain units (including the SCIE) can also be broken down into a pipeline if it has distinct stages. Ideally the execution pipeline is full at all times which allows for an instruction to be performed every cycle. Since certain instructions depend on the outcomes of previous instructions, the pipeline can not always be kept full but a compiler can rearrange the order of instructions to ensure the pipeline remains as full as possible at all times.

The presence of pipelining optimisations explained the similar performance between the hardware and software implementations as the fast inverse algorithm would benefit greatly from pipelining due to the simplicity of its operations. It appeared that the hardware implementation was also benefiting from pipelining but to a lesser extent as it consisted of fewer, but more complex stages to pipeline. Attempts were made using fence instructions to limit pipelining optimisations to within the confines of the cycle measuring functions to gain more accurate results but this typically led to the optimisations being disabled completely.

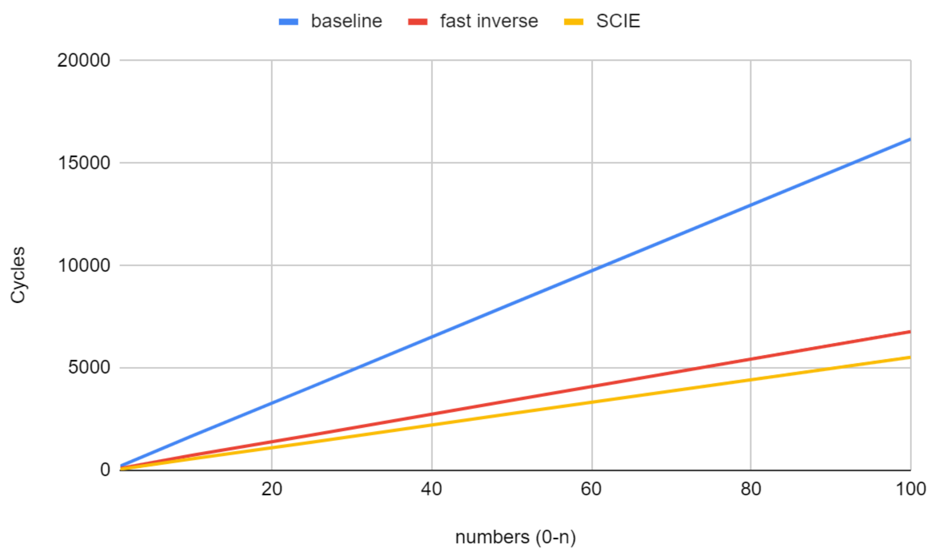


Figure 10: Graph of fast inverse benchmark results with no pipelining optimisations

The benchmarks were run again but using the volatile keyword to disable pipelining and other compiler optimisations. This reduced the performance gap between the baseline benchmark and the two fast inverse implementations. The disabling of pipelining optimisations also widened the gap in performance between the software and hardware implementations.

Although the hardware implementation outperformed the software implementation when pipelining optimisations were removed, the general use case of the algorithm made this redundant. This is because in the use case for fast inverse, the algorithm/instruction would be called numerous times in quick succession which would allow for further pipelining to take place. The lack of a tangible improvement when using a custom instruction was disappointing however it highlighted both the performance improvements compilers offer as well as the importance of writing software with hardware in mind. This is because every stage of the fast inverse algorithm corresponds to a set of simple instructions that benefit from pipelining. These results also showcased one of the primary appeals of a RISC architecture which is the fact that multiple, simple operations can sometimes be optimised to compete with or even outperform a dedicated instruction for the same task.

4.3 Regular Expressions

4.3.1 Analysis

Regular expressions are a useful means of defining a pattern to search for in a string. They have numerous uses such as defining syntax for a programming language or ensuring user input adheres to a certain format but they are also frequently used to extract important information from unstructured data. In this scenario regular expressions can have significant performance issues if the data to parse is large and contains a number of potential, but incomplete matches.

The most common cause of performance issues with regular expressions is backtracking. When a regular expression parser finds a potential match the characters ahead of it must be checked to determine if it is in fact a match. If this check fails the parser must revert to its initial state of when the potential match was found and continue parsing from there. An input with a high number of potential, but incomplete matches can lead to a significant increase in execution time especially when there are multiple stages of backtracking involved.

Designing an accelerator to handle any valid regular expression poses a number of challenges. Firstly the device would require a limit to the depth of backtracking it can support as each level would require an additional register to store a state that can be reverted to. This would lead to a complex design as it would need to support all regex characters, allow for character escaping and have a large number of registers for backtracking that in most cases would go unused. A potential solution to this would be the use of an FPGA which could be configured with a lightweight design, tailored to a specific regular expression. In the event that the expression needs to be changed, the FPGA could simply be reconfigured to adopt the new expression.

4.3.2 Implementation

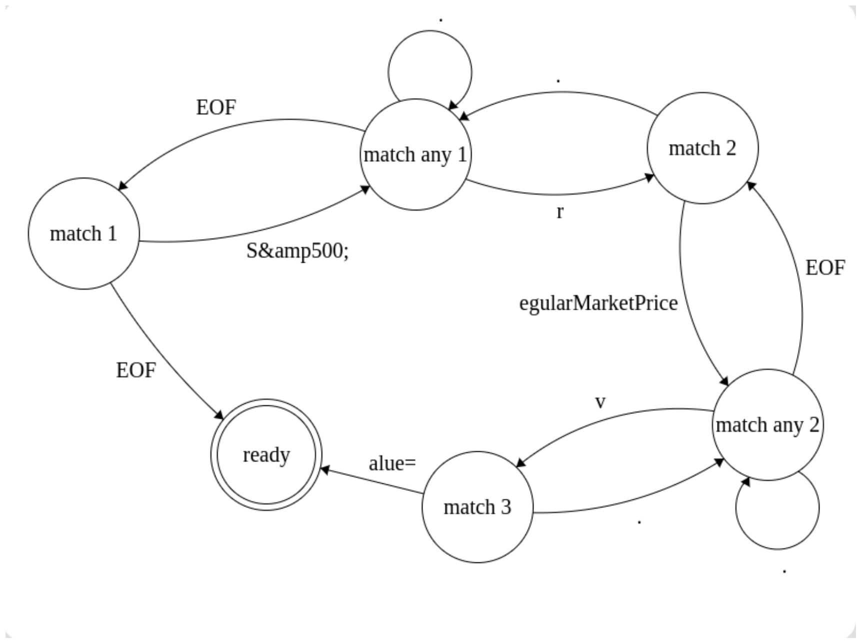


Figure 11: State machine for regular expression parser

A section of HTML from a financial website was used for an example design and the following regular expression was created:

```
S&amp;P 500\".*?regularMarketPrice.*?value=\"
```

The goal of the expression was to locate the price of a financial index and it contained two stages where significant backtracking may occur (denoted by ".*?"). These two stages involved matching any character, any number of times non-greedily, meaning the expression would search for the first match rather than the largest. These segments would continue to the end of the input string, in search of a match before backtracking. Before designing an accelerator, the expression was converted to a finite state machine, which served as a blueprint for the design

The design for the accelerator would read in the entire input string and store it in internal memory. Ideally any accelerator designs would have enough internal memory to store the entire input string. Scenarios with larger inputs such as large files may require a different design which stores one segment of the file at a time. This would provide a performance improvement for local backtracking but would result in far more cache requests and a more complex design.

An issue that significantly slowed development on this task involved benchmarks producing an error that would occur after the accelerator made a cache request. Significant time was spent ruling out potential points of failure such as the encoding of the custom instruction, the validity of the address being requested and whether or not the compiler treated chars as signed or unsigned. The cause of this issue was eventually found to be a single wire for cache requests which had been unchanged from example implementations provided by Chipyard. Comparing successful benchmarks with failing ones showed that when the wire was set to zero, requests would fail and when set to three cache requests would be successful. Attempts were made to ascertain the purpose of this wire but comprehensive documentation on the RoCC accelerator was difficult to find let alone information on a single value in the interface. The wire was then hard coded to the value three and benchmarks began producing the expected results.

Towards the end of the project and long after work on this topic was completed, a potential explanation for the wires' purpose was found. RISC-V has a privileged instruction set which implements different *privilege* modes that dictate how certain aspects of the hardware can be used. This can be used for security purposes such as preventing access to physical memory or in conjunction with higher level OS security checks such as access permissions. The value three specifies machine mode which essentially provides full permissions. The cache request wire that was causing issues appears to be for specifying the cache requests' permissions and the default implementation simply set this value to the permissions of the accelerator request. In the initial design for this task, the permissions of the accelerator request weren't captured for the entirety of execution, meaning the wire reset to zero and resulted in an invalid cache request. With this information in mind the design was altered slightly to keep the wire specifying the accelerator requests' permissions high during execution. This addition only added an extra cycle to execution times. Although the cause of the problem was eventually found, the amount of time spent searching for an explanation illustrated one of the primary issues faced during the project which was a lack of sufficient documentation.

An additional aspect of this design to note is the fact that characters are initially read into main memory one value at a time. The default bus size for a cache request is 64 bits meaning a more optimal approach would be to fetch eight characters at a time. Unfortunately Chisel's has very little functionality regarding bit indexing which makes converting a single 64 bit value to eight 8 bit values quite difficult. This restriction on the design was unfortunate however the majority of the accelerators' execution workload was time spent parsing the data rather than time spent reading it in.

4.3.3 Evaluation

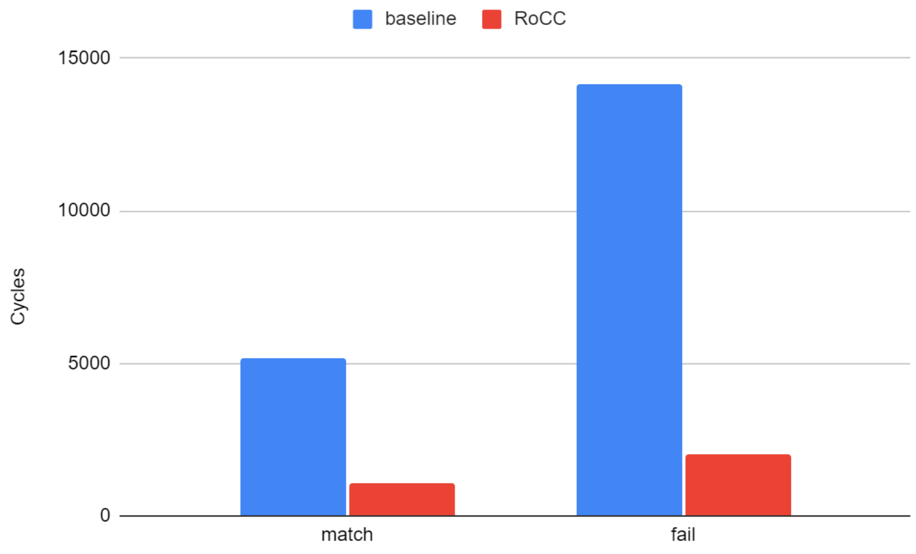


Figure 12: Results of regular expression benchmark

Since gcc did not appear to support the standard regex library for C, a baseline function for parsing the regular expression had to be written using only the standard C library. Since the regex library may provide more optimised search strategies, the baseline results should be treated with a degree of scepticism.

When the input string contained a match, the hardware accelerator outperformed the baseline considerably. This was likely due to the accelerators' use of internal memory and its reduced overhead in terms of instruction decoding.

When the input only contained partial matches, the gap between the baseline and accelerator widened significantly. With no successful matches, the amount of backtracking in both implementations greatly increased. In the software implementation this resulted in far more instructions to decode as well as more cache misses as large backtracks would result in the cache being dumped and replaced with the values from the initial state. The accelerator suffered from neither of these issues as it had no extra instructions to decode and already had the characters to backtrack to stored in internal memory.

4.4 Vector Operations

4.4.1 Analysis

Vector operations feature heavily in numerous fields of computing from artificial intelligence to gaming as they are a useful way to operate on large pieces of information. Due to the computationally intensive nature of matrix operations, specialised hardware is a necessity for computing certain tasks within an acceptable time frame. In the current market there are only two options available to consumers and businesses that want to perform operations on large vectors. One can either use a processor with SIMD instructions or use a GPU.

For those unfamiliar with SIMD it stands for single instruction multiple data and involves the use of special vector registers which can hold multiple pieces of information. When performing an operation with two vector registers, an element from each register is passed into an ALU to perform an operation. Adding ALUs to the processor allows for parallelization in which multiple vector elements are passed into multiple dedicated ALUs. GPUs operate in a similar fashion however on a much larger scale, using numerous specialised cores that divide up a task and run in parallel.

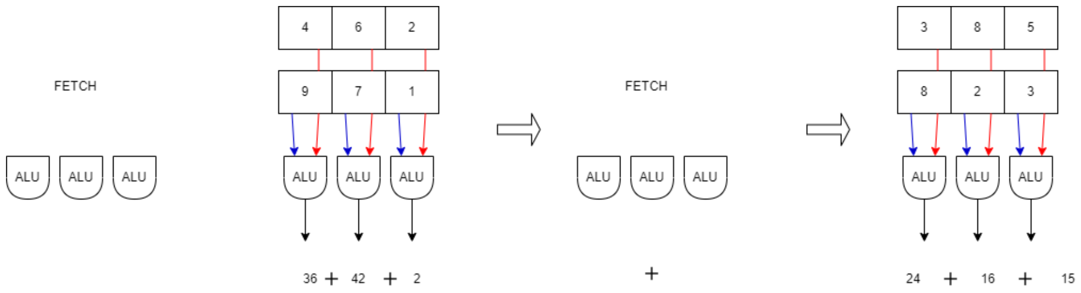


Figure 13: Illustration of a SIMD operation

SIMD instructions prove to be a very useful addition to general purpose CPUs when a GPU is not available however they do come at a cost. SIMD instructions need dedicated instructions to specify things like the type of data being operated on and the size of the data being operated on. This can lead to a massive instruction set when one is trying to accommodate all kinds of operations for all data types.

Another of SIMDs' weaknesses is the size of its vector registers. Each element in the vector register requires a dedicated ALU which limits the size of the register. While the parallelization provided by these ALUs offers a huge performance boost, the limited vector register sizes require large tasks to be broken up into small chunks. As a result a considerable amount of time is spent moving information into and out of the register. When working on large amounts of data this overhead becomes significant.

To avoid a bloated instruction set (arguably the main aim of a RISC ISA) RISC-V designers are tending towards a design that died out in the 1980s called vector processors[9]. These coprocessors address instruction set bloat through the use of special registers called CSRs (Control State Register). These registers dictate how the coprocessor acts and their values can be changed with a simple instruction. For example, instead of having an instruction for 16, 32 and 64 bit versions of the same operation, one can simply load one of those values into a CSR which dictates the size of information it is working on.

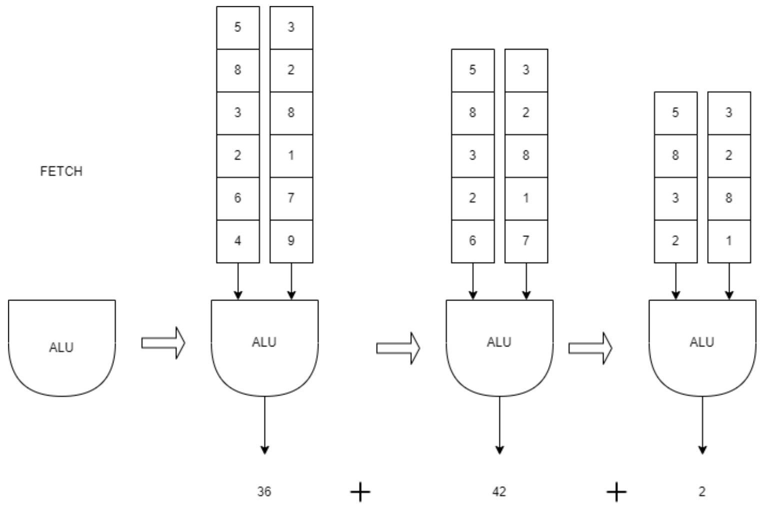


Figure 14: Illustration of a vector processor operation

Vector processors also address the overhead of small vector registers by having much larger ones. Instead of performing a number of small parallelised operations, a vector processor performs one large sequential task. At the start of an operation a vector processor will load as much information as possible into large vector registers. The vector processor then begins looping over elements sequentially and performing the operation. This may seem counterintuitive however it ensures that the ALU is in almost constant use. This contrasts with SIMD where multiple elements are processed in a single cycle however each ALU then goes unused while waiting for the next set of elements to be loaded.

4.4.2 Implementation

4.4.2.1 Limited Resource Design

The first design was primarily to become familiar with the memory request/response interface using a simple example, however it was also created out of interest in observing the performance benefits of an extremely small, cost effective accelerator. The first step in the design phase was creating a finite state machine to represent the accelerator.

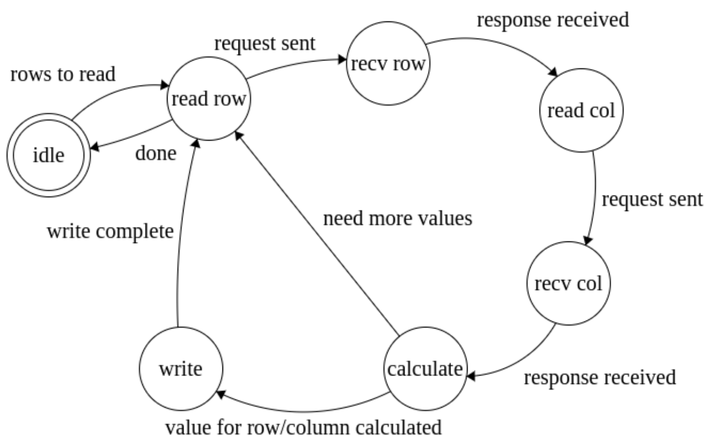


Figure 15: State machine for matrix multiplication accelerator with limited resources

As illustrated in figure 15 the accelerator fetches an element from a row of the first matrix and a column of the second matrix. Once these values are retrieved they are multiplied and added to a partial sum. The next elements are then retrieved and this continues until the end of a row/column is reached. The final sum is then written to memory. This is repeated with each column in the second matrix and once the sum for the last column is reached, the process is repeated for the next row in the first matrix.

While this design may seem no different from an iterative software approach, it avoids the overhead of sending and decoding numerous instructions.

4.4.2.2 Resource Abundant Design

The next design went in the opposite direction in terms of available resources. While the first design had no internal memory and went element by element, this design stored the entire second matrix as well as a row from the first matrix.

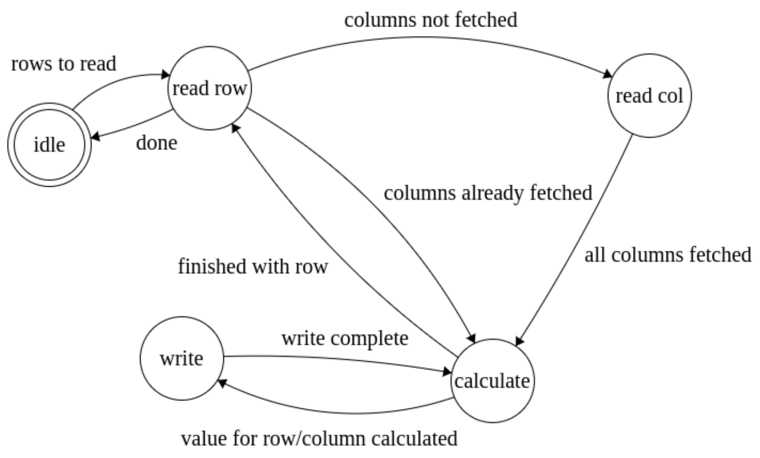


Figure 16: State machine for matrix multiplication accelerator with a large amount of internal memory

This design fetches the first row followed by the entire second matrix. The products for the row and each column are calculated and written to memory. The next row is then fetched and this process repeats until the value for the last row and last column is calculated.

While this design is not entirely practical it does serve to illustrate the performance benefits of local memory in accelerators. Furthermore with a good driver this design could still have its merits by breaking a large matrix into chunks and passing them to the accelerator.

4.4.2.3 Transposition Design

After reviewing the first design it became clear that cache misses were a significant issue. On large matrices each request for the next value in a column would incur a cache miss. In response a new design was created that transposed the second matrix at the beginning. Transposing a matrix simply involves turning its rows into columns and its columns into rows. This incurs a significant computational cost at the beginning of an operation but greatly improves cache performance for operations on large matrices.

The reason matrix transposition improves cache performance is because of how a matrix is stored in main memory. When performing matrix multiplication intermediate values are calculated by taking a value from a row in one matrix and a value from a column in the second matrix. When a row value is stored in the cache, the row values after it are also stored as part of a cache line. This means that when the next row value is needed it is already in the cache. While row values are adjacent in memory, the distance between each column value is the length of an entire row. If the matrix has large row lengths when a column value is loaded into the cache then it is unlikely that the next column value will already be cached. Transposing the matrix means that these column values become adjacent in memory and that each cache line will store multiple column values.

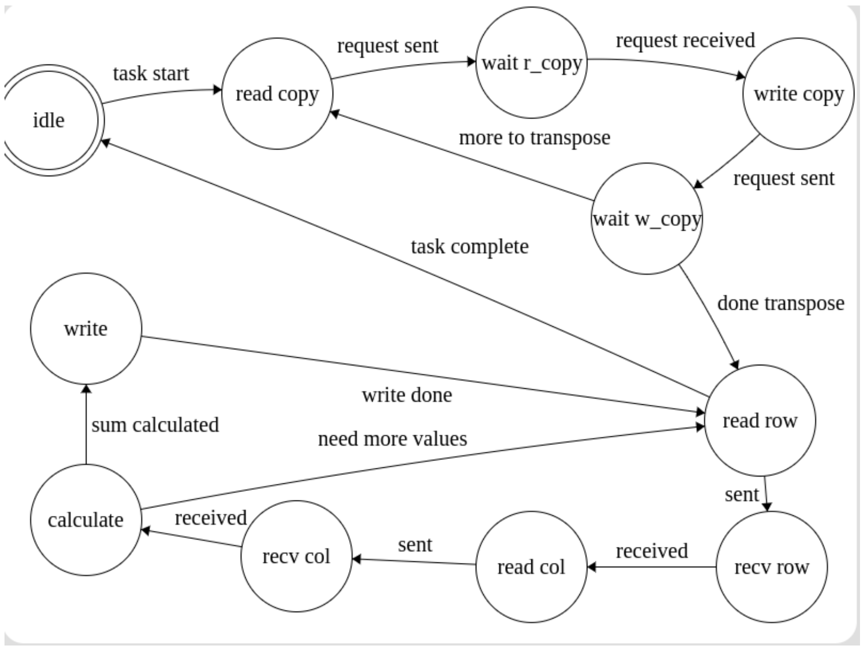


Figure 17: State machine for matrix multiplication accelerator that uses transposition

This design is similar to the first design however at the beginning, a transpose of the second matrix is written to memory. Instead of getting the dot product of rows and columns, the accelerator proceeds to calculate the dot product of rows in the first matrix and rows in the transposed matrix to achieve the same result. This design would be ideal in situations where there is an abundant amount of main memory but the affordability of the accelerator is still a significant concern.

4.4.2.4 Vector Processor Design

The final design was intended to be the most practical and to incorporate findings from previous designs.

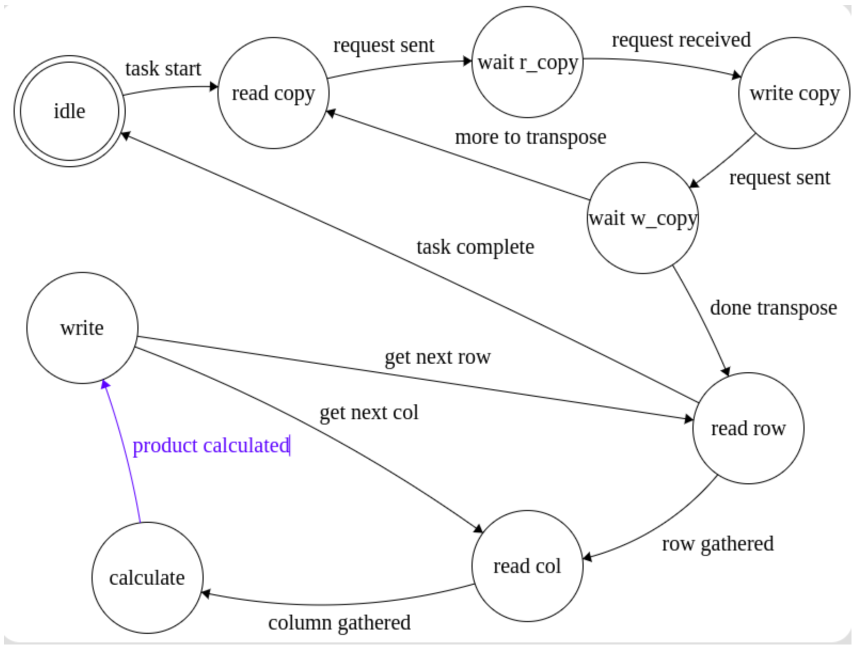


Figure 18: State machine for matrix multiplication accelerator that more closely resembles a vector processor

This design incorporates the matrix transposition from the previous design and consists of 2 vector registers that each hold a row and a column. As a result columns have to be loaded into local memory multiple times. The design was intended to adhere more to the standard architecture of vector processors and offer a middle ground between the lack of memory in the first and third design and the impractical abundance of memory in the second design.

4.4.3 Evaluation

Benchmark Results (cycles taken)						
matrix size	baseline	baseline transposed	transpose design	no memory	vector unit	memory abundant
2	140	146	169	141	155	117
4	1077	1250	665	553	457	247
8	7063	7812	4281	3833	2281	1043
16	53190	55491	31829	29768	14523	6235
32	416809	432071	257992	250054	112089	46276
64	6889363	3816257	2456415	5563609	1288366	378425

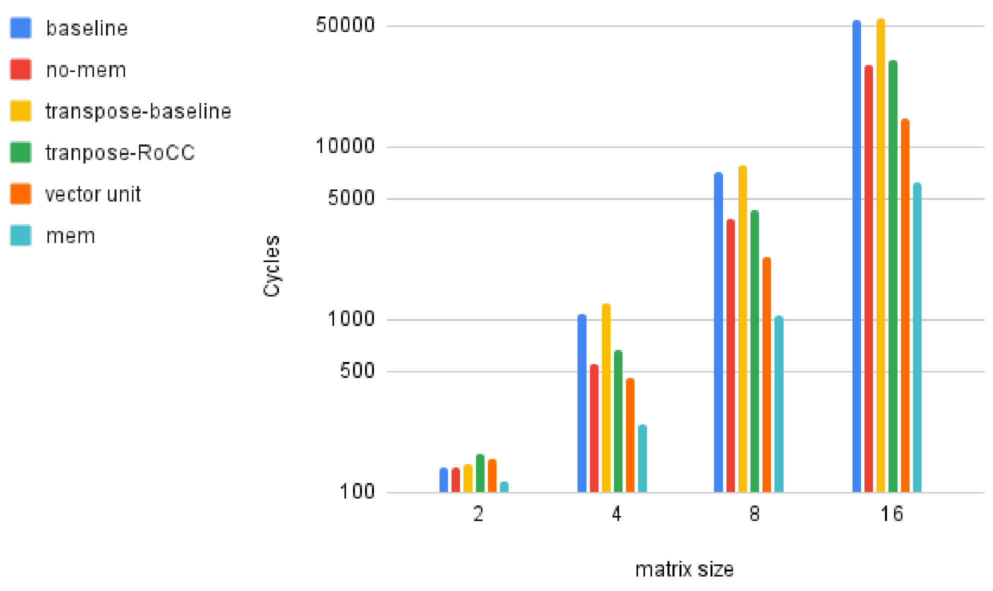


Figure 19: Matrix multiplication accelerator results for 2-16 (Note: graph uses log scale)

Benchmarks were written for testing each design using square matrices of 64 bit unsigned longs. A baseline benchmark performing basic matrix multiplication was written along with a benchmark that transposed one of the matrices in a similar fashion to the final two designs. Benchmarking stopped at matrices of size sixtyfour as baseline benchmarks began taking a considerable ammount of time to complete past that point. Additionally any designs using internal memory had a maximum matrix size of sixtyfour meaning tests involving larger matrix sizes would require breaking the matrix up into smaller pieces.

In benchmarks involving small matrices, the design using no internal memory or transposition outperformed the design using transposition. Similarly the baseline benchmark performed slightly better than the software implementation using transposition. This is due to the computational overhead of transposing the matrix at the beginning of the benchmark. The cache line size in the Rocket core is 512 bits meaning it can store eight 64 bit numbers. This means that for matrices smaller than eight, a request for a column value would result in the next column value being cached. Additionally for matrix sizes up to and including eight, entire rows would be cached, which would have limited the number of cache misses. Although the number of cache misses would increase for matrix sizes of eight and sixteen, it appeared that the computational overhead of transposition was still greater than the performance cost that these cache misses incurred.

The design using no internal memory or transposition performed quite well against the baseline benchmark in tests using small matrices. This gap in performance showed the computational overhead of decoding instructions. Additionally the vector unit based design and memory abundant design considerably outperformed the other designs due to their use of internal memory.

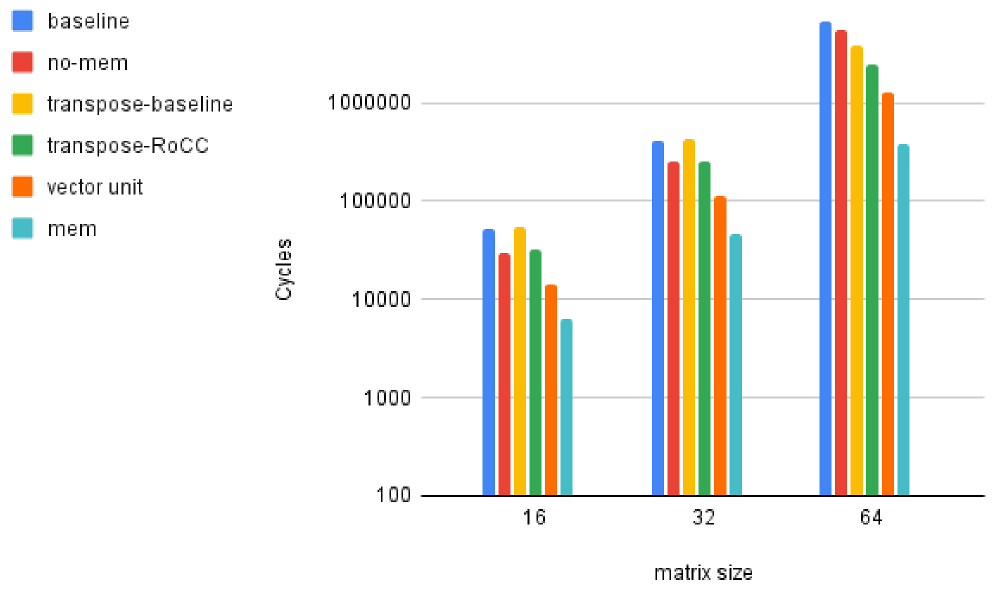


Figure 20: Matrix multiplication accelerator results for 16-64 (Note: graph uses log scale)

As matrix size increased the gap in performance between the implementations that had no transposition and the ones that did began to reduce. Once the matrix size increased to sixtyfour, the impact of cache misses became more apparent. The gap in performance between the baseline benchmark and the design using no memory or transposition reduced significantly as the overhead of cache misses began to far exceed the overhead of instruction decoding. Additionally the baseline using transposition and the initial transposition design had a significant gap in performance, meaning the overhead of decoding instructions still had a considerable impact on performance. The vector unit and memory abundant designs continued to have the best performance as matrix size increased

The most promising results were from the vector processor based design as it achieved a considerable reduction in execution time while also only using a moderate amount of internal memory. As previously mentioned Chisel and the RoCC interface had seemingly no support for fetching multiple values in a single cache request. By default the size of this bus is 64 bits meaning it would have had no impact on these benchmarks as they used matrices of unsigned longs (also 64 bits). Regardless of this a more efficient Vector processor design would involve the use of a much larger bus that allows multiple elements to be requested in order to expedite the process of loading information into the accelerator. This would also play to the strengths of the vector processor design as its primary use case would be workloads with large datasets.

5 Evaluation

The *popcount* instruction showed a significant performance improvement over the base ISA and highlighted the extensibility of the RISC-V instruction set. The fast inverse square root instruction on the other hand had disappointing results in terms of performance but still provided valuable insight on the benefits of writing software that takes full advantage of its underlying hardware.

The accelerator for a regular expression showed that complex workloads can use very lightweight designs when the accelerator is tailored to a specific task while still providing significant speedup. This task also highlighted some of the challenges of hardware design such as serialising recursive concepts and isolating the source of an issue in a design.

Exploring vector operations proved to be one of the more informative aspects of the project as it involved the most work. Working on multiple designs for matrix multiplication showcased some of the many economies of scale involved in hardware design such as the amount of available internal memory or opportunities to improve cache performance at the cost of additional resources.

At the beginning of the project benchmarks were intended to include power estimations to show the additional power costs incurred by each design. Unfortunately the tools provided by Chipyard for FPGA bitstream generation required a licence for special FPGA synthesis software. Additionally the open source tools that were used in tandem with the synthesis software had quite sparse documentation. As a result this area was not investigated further. While the primary means of saving power with custom hardware would be a reduction in the number of active units required to handle the same workload, power estimations may have been able to highlight certain designs that were not viable. This is because high power usage from the device may also result in an increase in temperature, which in turn would increase the energy costs of cooling measures. While not pursuing this further was disappointing it did highlight the need for open source synthesis tools as requiring enterprise level software to fully test a hardware design may limit interest among the open source community.

6 Conclusions

6.1 Reflection

This project provided a significant amount of experience and insight into the ecosystem of low level hardware. In the initial phases of the project time was mostly spent becoming familiar with certain tools, languages and concepts while also working on simple hardware designs. As the project progressed however, more complex and ambitious hardware designs became achievable which greatly illustrated the body of knowledge that was acquired throughout the course of the project. While the project was successful in achieving its aims, its scope was reduced by a number of issues faced throughout the course of the project.

6.1.1 Issues

6.1.1.1 Learning Curve

The project featured a large number of tools, languages and frameworks that posed a significant challenge to become accustomed with. A general familiarity with Scala syntax was required to properly navigate the codebase and write designs in Chisel. Properly navigating the codebase also required a knowledge of certain Scala features such as lazy evaluation and implicit values. Working with Chipyard also required knowledge of the Scala project structure and build files as the project was maintained as a git submodule. This made resuming work from a fresh copy of Chipyard less cumbersome as it simply required a git command and a few adjustments to some top level project files. While Chisel was quite accommodating to learn it did require learning design patterns and becoming accustomed to the concept of operations taking place on a cycle by cycle basis rather than sequentially. Becoming familiar with this concept involved debugging a number of invalid designs in which a register was a cycle ahead or behind of its intended value.

6.1.1.2 Documentation

The biggest hindrance to the scope and progress of the project was a lack of documentation. A number of tools and interfaces that were integral to the project had either poorly written documentation or no documentation whatsoever. The SCIE for example which was used to design custom instructions has no official documentation and is only briefly mentioned in a few presentations and articles. The documentation for the RoCC accelerator on the other hand is quite sparse and makes a number of assumptions about the reader's prior knowledge. Thankfully the Chipyard codebase has a number of example accelerators that were instrumental in becoming familiar with the interface. These examples however are mentioned nowhere in the documentation and were only discovered whilst exploring the codebase. This discovery of useful information in the codebase that is not discussed in the documentation was a frequent occurrence. Initially benchmarks used hand written inline assembly to call custom instructions or interact with an accelerator however a header file was later discovered with useful macros for achieving the same goal. This lack of sufficient documentation considerably reduced the scope of the project as significant time was spent searching for solutions to issues that could easily be resolved with a few lines of explanation.

6.1.1.3 Development Time

Another factor that led to a reduction in the initial scope of the project was a lack of understanding of the significant amount of time it takes to create and validate a design. Typically building a design would take up to twenty minutes or more using eight cores and eight gigabytes of RAM. This also required all other applications to be closed during RTL synthesis which meant further work could not be performed while waiting. Once a simulator was created a benchmark would be run and instructed to create a waveform dump file and time out in the event of a stall. Waiting for a time out to occur could take up to fifteen minutes for certain benchmarks. Inspecting the waveform file also took a significant amount of time as the values of wires would have to be observed cycle by cycle until the cause of an issue was found. An attempt to fix this issue would begin this design process again which was quite time consuming especially when the cause of an issue was ambiguous and required multiple attempts to resolve. While this process did hinder the scope of the project it did provide insight into the significant work involved in synthesising and debugging a hardware design.

6.1.2 Insights

6.1.2.1 Hardware Desing

Despite the issues faced during the project it was still a rewarding and informative experience. The project provided exposure to numerous design concepts such as vector processors and the performance impact of caches and internal memory. The project also provided experience with numerous languages, frameworks and tools such as Scala, Chisel, Verilog and GTKwave. The experience gained regarding low level hardware in general will be indispensable in future endeavours as knowing how software interacts with and leverages its underlying hardware proves useful when writing both high level and low level programs.

6.1.2.2 C experience

Writing benchmarks provided valuable experience with C and its ecosystem while also highlighting some of the many reasons it is the most prevalent low level programming language. Since benchmarks were written to be run bare metal the project also emphasised just how versatile the gcc compiler is as particular flags were used to disable high level features and create a bare metal binary. Writing benchmarks also required the use of preprocessor directives which allowed for benchmarks to quickly be altered and recompiled and helped immensely in expediting the testing process.

6.1.2.3 A Growing Community

While researching topics for the project, or simply looking for information to resolve an issue, it became clear that RISC-V has a growing community and has resulted in numerous open source core designs being developed and released. Even throughout the course of this project support for the RISC-V ISA has grown such as Intel Foundry Services supporting the fabrication of RISC-V chips[10]. This shows that the RISC-V ISA will likely have a considerable impact on the hardware design industry in the coming years and that the experience gained with the ISA will likely prove invaluable in any future endeavours. Regardless of RISC-V's future, the project provided experience with and knowledge on numerous low level hardware concepts that are useful in multiple fields.

6.2 Future Work

6.2.1 Peripherals

Chipyard provides interfaces for the design of peripherals which were an area of interest while working on the project. Due to the limited timeframe of the project and the significant potential learning curve involved with becoming familiar with the interface, this area was not investigated further. A potential future endeavour would be the design of a packet filter.

6.2.2 FPGA

As stated previously, some of the FPGA synthesis tools required a commercial licence to use. The simulation of custom designs was of great interest when working on the project however this issue along with the time required to become familiar with synthesis tools resulted in the topic not being pursued further. Purchasing an FPGA and acquiring a software licence to investigate this topic further would be of great interest. Furthemore, examining the power consumption of different designs would also be of interest.

6.2.3 Further Regular Expression Work

While only one accelerator was designed for regular expressions, further investigation into the area would be of interest. A particular design that may be promising would be an accelerator designed for a networking context where information becomes available in small instalments. The design challenge with this would involve periodically updating the address space that the accelerator is reading from and stalling the accelerator when it requires more information to be received.

6.2.4 SiFive

Another area of interest would be using a RISC-V microcontroller such as the one provided by SiFive. Running linux on a RISC-V microcontroller and developing applications would provide more insight into how the instruction set performs when operating beneath multiple layers of abstraction such as a kernel or a high level language.

6.2.5 ROCC Modification

While the RoCC interface and Chisel in general were extremely interesting to work with, a key drawback was a lack of support for bit indexing and extracting multiple values from a single cache request. This would be an interesting topic to pursue and could perhaps be achieved through further familiarisation with Chisels' many classes and inbuilt functions. Alternatively Chisel has support for classes whose inputs are passed to a Verilog implementation which has much better support for bit indexing operations.

6.2.6 Vector-SIMD instructions

Learning about the architecture of vector processors and the advantages they have over SIMD was quite interesting and further work around this topic would be of considerable interest. In particular vector-SIMD instructions are quite interesting as they involve incorporating the parallelization of SIMD into a vector unit[11]. This architecture (as well as the regular vector processor architecture) can also implement pipelining which could provide significant performance improvements and would be interesting to explore.

References

[1] Timothy Prickett Morgan. “Covering all the compute bases in a heterogeneous datacenter”. In: *The Next Platform* (Jan. 2020). URL: `https://www.nextplatform.com/2020/01/15/covering-all-the-compute-bases-in-a-heterogeneous-datacenter/`.

[2] John Loeffler. “No more transistors: The end of Moore’s law”. In: *Interesting Engineering* (Feb. 2022). URL: `https://interestingengineering.com/transistors-moores-law#:~:text=Why%20Is%20Moore's%20Law%20in%20Trouble%3F,-Source%3A%20Intel&amp;text=The%20problem%20with%20Moore's%20Law,do%20to%20make%20them%20smaller.`.

[3] Vipin Jain. “Council post: The era of domain-specific processors”. In: *Forbes* (Apr. 2020). URL: `https://www.forbes.com/sites/forbestechcouncil/2020/04/17/the-era-of-domain-specific-processors/?sh=1445f615792f`.

[4] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[5] Jerry Zhao et al. “SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine”. In: (May 2020).

[6] Paul Kocher et al. “Spectre attacks: Exploiting speculative execution”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1–19.

[7] Alon Amid et al. “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs”. In: *IEEE Micro* 40.4 (2020), pp. 10–21. ISSN: 1937-4143. DOI: `10.1109/MM.2020.2996616`.

[8] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1*. Tech. rep. UCB/EECS-2016-118. EECS Department, University of California, Berkeley, May 2016. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html`.

[9] Erik Engheim. "RISC-V vector instructions vs arm and x86 SIMD". In: *Medium* (Jan. 2021). URL: `https://medium.com/swlh/risc-v-vector-instructions-vs-arm-and-x86-simd-8c9b17963a31`.

[10] Karl Freund. "Intel creates $1B innovation fund to grow RISC-V market (and attract new foundry customers)". In: *Forbes* (Feb. 2022). URL: `https://www.forbes.com/sites/karlfreund/2022/02/07/intel-creates-1b-innovation-fund-to-grow-risc-v-market-and-attract-new-foundry-customers/?sh=6fa921fd16aa`.

[11] Erik Engheim. "Advantages of RISC-V vector processing over x86 SIMD". In: *Medium* (Apr. 2022). URL: `https://itnext.io/advantages-of-risc-v-vector-processing-over-x86-simd-c1b72f3a3e82`.