Design and Analysis of a Heterogeneous RISC-V SoC on FPGAs

Matthew Knight
Department of Computer Science
University of Warwick

Supervised by Dr Eduardo Wächter

30 April 2023

Abstract

This project aims to design and analyse heterogeneous RISC-V SoCs implemented in FPGAs. Many devices have varying performance needs during use, and always aim to minimise power usage. Heterogeneous architectures aim to provide a solution that delivers high performance and high efficiency by combining highly efficient CPU cores (S cores) and highly powerful CPU cores (B cores) in a single processor. When performing background, low priority tasks the S cores are used and the B cores are idle/off, resulting in low power usage. When there are demanding, high priority tasks the B and S cores are used, resulting in higher performance than if just the S cores existed.

The CPU cores implement the RISC-V ISA, an open-standard ISA with support for Linux. There is minimal research into general purpose RISC-V processors implementing a heterogeneous architecture, so the project will perform some novel research. The S and B cores will be designed using the RocketChip generator, that generates HDL for a configurable RISC-V SoC.

The designed heterogeneous SoC has been be benchmarked and compared to homogenous designs in terms of size, power draw and performance. Based on these metrics, heterogeneous RISC-V SoCs implemented in FPGAs have limited benefits over homogenous, with a B and S processor providing small decreases in power and area usage for significantly less performance compared to a two B processor, but giving much increased performance for small power and area increases compared to a two S processor.

Contents

Chapter 1

Introduction

The aim of this project is to design a heterogeneous system on a chip (SoC) using the RISC-V open standard ISA[**?**][**?**]. Once designed, the SoC should be implementable on an FPGA and be able to execute bare-metal C code and assembly with SMP, allowing all cores to execute code at the same time.

1.1 Motivation

Designing a processor is a complex challenge. They need to be power efficient to reduce the cost of running the system, as well as providing adequate processing power when required. Often one is sacrificed to improve the other: for instance, energy efficiency is often reduced due in designs of large processors containing multiple cores. Due to the increased size and complexity, more power has to be supplied to achieve the same performance as a smaller processor.

A heterogeneous CPU attempts to solve this issue by combining dissimilar core designs; often a powerful core, or p-core, and an efficient core, or e-core. When only light processing is required, the p-core can be effectively shutdown and the e-core will do all processing, resulting in less power used. For heavy processing, the p-core is then used to increase peak performance. Depending on the exact implementation, the p-core can be used either individually or in tandem with the e-core, but both result in greater performance than just the e-core.

This has been used extensively in many mobile devices, and increasingly in larger devices like laptops. The Apple M2 processor[**?**] is one such example, implementing 4 high-performance cores and 4 energy-efficient cores. These are arranged in a hybrid configuration similar to ARM DynamIQ and big.LITTLE[**?**], allowing cores to dynamically be assigned work that best fits them. In Apple's design, both cores are capable of executing the same code, with the performance core benefitting from much increased cache and other proprietary changes to increase it's speed over the efficiency core.

1.2 Objectives

The overall aim is design a heterogeneous SoC containing two types of RISC-V core that can execute bare-metal C code and assembly with SMP. Objectives have been labelled according to the MoSCoW method[**?**] to indicate their importance and expected completion.

1. Design a heterogeneous RISC-V SoC containing 2 dissimilar cores, each capable of executing at least the RV64 instruction set (Must).
2. Execute assembly instructions on both cores when implemented in an SoC on an FPGA (Must).
3. Execute bare-metal code on both cores when implemented in an SoC on an FPGA (Should).
4. Measure the performance of the SoC for comparison against single-core designs. (Should)
5. Run embedded Linux on the SoC and connect to it via serial or SSH (Could).
6. Allow processes to execute on both cores inside of embedded Linux (Could).
7. Intelligently select which core the process will run on, depending on factors such as process priority and resource usage (Won't).

Figure 1.1: Objectives for the project

Chapter 2

Background and Research

2.1 RISC-V

2.1.1 ISAs and RISC vs CISC

An ISA is an instruction set architecture, and is the formal definition for what an abstract model of a CPU. The ISA defines the instructions, registers, communication standards and other features of a CPU in order to allow a person with the ISA to design a physical CPU that would implement the abstract model. CPUs that implement the same ISA (or supersets of an ISA) are able to execute the same code, meaning programs written for a CPU can be run on a different CPU if they share the same ISA. This is how a single compiled program is able to be run on so many different systems, and is incredibly important for modern devices, where there is a huge range of processors available.

Many modern ISAs are based on older ISAs that have been extended to allow for new instructions, registers, communication protocols, etc, while maintaining support for previous ISAs. This is typically called CISC (Complex Instruction Set Computers), where there are a very large amount of instructions the processor can execute. The backwards compatibility can be very useful, and having explicit instructions for many tasks can mean the tasks are completed more efficiently than if multiple smaller instructions were executed. Having large instructions can also reduce program sizes, due to the increased operations per instruction, which is very beneficial, especially in small systems.

There are several issues with CISC however, such as increased complexity in the CPU as the amount of instructions increases, leading to rising costs of design and manufacture, and can force the physical size of the CPU to be larger to accommodate the extra logic space for larger instructions. More complex designs will also make future development harder if backwards compatibility is to be maintained.

The alternative to CISC is RISC (Reduced Instruction Set Computer). RISC aims to reduce the number of instructions the CPU requires in order to keep the design as simple as possible. This allows the design to be done much more easily, as well as allowing future extensions to be easier. Reduced complexity can also allow for greater optimisations in what instructions are in the ISA, with the aim of achieving greater performance than CISC by executing more instructions at much greater speed. There are drawbacks however, with increased program size and possible reduction in speed of execution for multiple operations that could've been completed in a single instruction using CISC.

2.1.2 The RISC-V ISA

RISC-V is a new, modern, open-standard ISA that follows RISC concepts. The ISA does not specify a single instruction set, but multiple by having three different width instruction sets with different base instructions. 32-bit RISC-V (RV32I) is focussed on embedded and personal systems, 64-bit RISC-V (RV64I) for personal and server systems and 128-bit RISC-V (RV128I) for server and high-performance compute systems. RV32 and RV64 have embedded versions, RV32E and RV64E, that have reduced registers and instructions.

Extensions

At base, these implement only integer addition/subtraction. The ISA contains extensions that add more functionality to the CPUs, such as support for multiplication/division, floating point operations IEEE-754, compressed instructions, atomics, etc. An instruction set implementing these extensions is referred to by adding the initial of the extension to the name, so a 64-bit CPU with integer addition/subtraction, multiplication/division and compressed instructions would be 'RV64IMC'. Custom extensions can also be written, allowing for a great deal of customisation in RISC-V CPU designs.

There are 30 official extensions to the base instruction sets. The most popular are listed below.

M Integer multiplication and division

A Atomic instructions

F Single-precision floating point

D Double-precision floating point

Zicsr Control and Status Register (CSR)

Zifencei Instruction-Fetch Fence

G Short for `IMAFDZicsrZifencei` as a 'general-purpose' CPU, e.g. RV64G

C Compressed instructions, 16-bit instead of 32/64

Others Quad-precision floating point, bit manipulation, vector, misaligned atomics, etc

2.2 Processor Design

Processor performance can be measured in multiple ways, with the main metrics being speed at executing tasks, cost of manufacturing, physical size and energy consumption. These are implicitly linked - a processor with larger physical size will likely draw a larger amount of energy, contain more logic and so be faster at executing tasks and cost more to manufacture. The aim of processor design is to maximise the speed of task execution, while minimising the rest. The balance between these is what gives rise to different processor designs, as a mobile device is much more constrained in power usage than a large server and will require a different processor.

2.2.1 Power reduction in processors

Maximising task execution while minimising energy consumption can be balanced in a multitude of ways. Power consumption in CPUs can be split into two sections: switching power and leakage power. Switching power is the power used by CMOS (transistor) gates as they change states, and varies on CPU activity. Leakage power is the power lost through slow leakage current flow through transistors in the 'off' state, and has increased as transistor sizes decrease and the boundary that must be crossed reduces.

Dynamic Voltage Frequency Scaling (DVFS)

CPUs contain clocks, which synchronises the components inside the CPU as they change state and execute instructions. Increasing the clock speed will result in an increase to the number of instructions executed per second, directly increasing the performance of the CPU. However, this directly increases the amount of switching power the CPU draws, given by equation 2.1.

$$P_{switching} = \alpha \cdot C \cdot V^2 \cdot f \alpha = activityfactor, proportionoftransistorsthatswitcheverycycleC = capaci \tag{2.1}$$

In addition to this, increased clock speeds often require increased voltage in order to increase the current flow through transistors to activate them in the reduced time frame from shorter clock periods. Therefore, reducing the clock speed and the voltage will result in a quadratic reduction in switching power for a linear reduction in performance and if low performance is needed, the power usage can be significantly reduced. CPUs implementing dynamic voltage frequency scaling are able to adjust their voltage and frequency during runtime dependant on the current tasks being run. This allows them to dramatically reduce power consumption during periods of low CPU utilisation, while still being able to provide good performance with high CPU utilisation.

However, this makes some requirements of the CPU. There must be software that allows the CPU to estimate it's current utilisation, and this has to be run frequently in order to have a fast response and increase clock speed when a demanding task is run. This can take up valuable CPU time, and requires some form of scheduling to repeatedly switch to and from this task. The CPU must also implement hardware that allows it to change the clock speed and voltage, increasing the physical size of the CPU and increasing manufacturing costs.

Clock Gating

Clock gating is another technique for reducing power consumption in a CPU. When a section of the CPU is unused for a number of cycles, the clock signal to that section is removed. This disconnect reduces the switching power, as no transistor switching will occur in that section of the CPU without the clock, as well as reducing the capacitance seen by the clock generator.

2.2.2 Heterogeneous SoC Designs

Another solution for power reduction in processors is heterogeneous designs. Multicore processors contain multiple CPUs in order to increase the potential performance in parallel computing tasks. A heterogeneous design contains multiple, different CPU designs instead of all CPUs being copies of each other as in homogeneous designs.

By having multiple different types of CPU, the processor can increase efficiency and reduce power consumption by intelligently selecting which CPU to run a task on. For example, a processor containing a small (S) CPU and a big (B) CPU could be running in a mobile phone. For the vast majority of the time, the mobile phone is not in use and only runs tasks such as checking for texts, emails, etc. These tasks can be run on the S CPU, with the B CPU fully disabled/clock gated, reducing the power consumption to that of the S CPU + B CPU leakage current. This power consumption should be less than that of the B CPU performing the tasks, in order for the processor to actually provide efficiency increases over a B CPU homogeneous design.

When the mobile device is actively in use, like video playback or mobile games, the B CPU would be used to provide greater performance than the S CPU, or both used for parallel tasks. This allows a heterogeneous S+B design to provide better energy efficiency and better performance than a homogeneous design with one B CPU.

There are some drawbacks to heterogeneous designs. In order to properly utilise the differences between the CPUs, a scheduler must be customised for the exact processor as other designs with different CPUs will need to switch which task is run where at different stages. A heterogeneous system with a medium CPU and a big CPU will be able to use the smaller CPU for more intensive tasks than a heterogeneous system with a small CPU and a big CPU, thus requiring a scheduler with different parameters.

Heterogeneous designs may also have differences in ISA. This can be beneficial, allowing an S CPU to be even smaller by removing features like vector support, but can also lead to many issues when writing software. If there are differences in the ISA and a program must be able to run on both S and B CPUs, the program has to be compiled for a subset of the available instructions. This can prevent features of the large CPU from being fully leveraged and reduce the increased performance that moving from the S CPU to the B CPU would provide. As such, general purpose heterogeneous systems that switch processes between S and B CPUs prefer them to have the same ISA, but differ in micro-architecture (how the ISA is implemented: frequency, pipelines, cache sizes, etc) to allow one compiler for both CPUs

The physical size and complexity of the processor will also be increased compared to a single CPU design. This increases the cost of design and manufacturing, another drawback compared to homogeneous designs.

Accelerator/Co-processor Heterogeneous Designs

Some heterogeneous designs do not attempt to pair types of general purpose CPUs, but instead have a host CPU type and accelerator CPU type. The host CPU explicitly schedules tasks for the accelerator CPU, the latter of which is typically optimised for a certain task(s) to increase the performance and efficiency in completing it. This format is typically used for systems with highly specific workloads, for example a digital camera may require a general-purpose CPU to run the OS, and have accelerator CPUs for image and video processing.

It follows that accelerator heterogeneous designs are a better choice when it is known exactly what work the SoC will be doing, such as embedded systems. For more general purpose systems where the use-case or end-user of the system will determine the type of work done, accelerator designs are a lot less suitable due to the increased performance that having another general purpose CPU would give in cases where the accelerator is not applicable.

2.3 FPGAs

FPGAs (Field Programmable Gate Arrays) are flexible logic chips that can implement hardware designs that would usually be manufactured as ASICs (Application-Specific Integrated Circuits). FPGAs are often used due to their reprogrammability and low start-up cost: while ASICs can be cheaper to mass produce, FPGAs are cheaper when a very limited amount are being produced, for example in prototyping. FPGAs can also be reprogrammed, and reused for multiple projects. However, an FPGA is far slower than an ASIC performing the same task as it is not 'hardwired', and become much more expensive than ASICs when mass-production of the chip is required.

2.3.1 Flexible logic blocks

Flexible logic blocks are what allow FPGAs to implement arbitrary logic functions. These consist of look-up tables, multiplexers, flip-flops and some small arithmetic logic, such as a 4-bit adder.

LUT An n-input LUT is a 2^n bit memory which can be used to store the truth table for an n-input logic function, thereby functioning as a combinational logic gate(s).

Multiplexer Multiplexers can be used to switch the output between the LUTs, flip-flops and arithmetic logic.

Flip-flops Used together with the LUTs to implement clocked logic functions.

Arithmetic logic Dependant on the FPGA, typically shared between a few logic blocks.

LUTs in modern FPGAs are typically around 6-inputs and so can store 64 combinations. For logic functions with more inputs than a single LUT, multiple LUTs can be combined and implement complex functions.

2.3.2 Flexible routing

FPGAs need to be able to connect different resources inside the chip to create the correct datapaths. This is achieved using large grids of wires between groups of logic blocks (sometimes called slices). The grids connect at switch boxes, which connects tracks to create data pathways between components. Latency in the signals sent are increased as the length of wire and amount of switch boxes increases, so minimising this is an important part of the process when a design is being mapped to an FPGA.

2.3.3 Flexible IO

FPGAs often have IO pins connected to logic that allows them to be programmed, implementing different IO protocols as required.

2.3.4 Hard modules

While the flexible logic can be used to implement any logic function, the FPGA implementation will be slower than a dedicated hardware module. To increase performance in FPGAs, hard modules are embedded in the flexible logic to increase the performance when doing common tasks, like integer arithmetic or signal processing.

Memory is often added to increase performance when large amounts of data must be stored and processed by the FPGA, as well as reduce the amount of

LUTs used as data storage instead of logic functions. Block RAMs are embedded inside the flexible logic to reduce the physical distance between them, decreasing latency and increasing potential clock speed. As well as BRAMs, FIFOs and IO buffers are commonly used.

DSP (Digital Signal Processing) blocks are also embedded within the flexible logic blocks. These contain hardwired ALUs, and increase FPGA performance of arithmetic based tasks.

2.3.5 FPGA programming and HDLs

FPGAs implement logic defined by a HDL (Hardware Description Language). HDLs are similar to conventional coding languages in how they are written, but describe the structure and behaviour of a digital circuit at RTL (Register Transfer Level). There are also differences in the "compilation" of a HDL. Instead of being reduced to bytecode specific to an ISA, the HDL goes through the process of synthesis, where the formal definition is simplified into a design in terms of logic gates. The generated design is a netlist, effectively a list of the logic gates and the connections between them. The next stage in the process is place and route, which is specific to the FPGA being used. The netlist is mapped to resources inside the FPGA, such as LUTs and DSPs, with wires then routed between the resources to make the desired data paths. This stage requires optimisation, as the placement of the resources affects the distance of wire and latency, so changing the placement can result in better (or worse) maximum clock speeds and signal integrity. The result of place and route is a final design that can be implemented on an FPGA by generating a bitstream, a file that is loaded by the FPGA and configures the resources and switchboxes to implement the design.

2.4 Related work

2.4.1 A RISC-V Heterogeneous SoC for Embedded Devices[?]

This project is ongoing, and presents work designing a RV64 (RISC-V 64-bit) host core that offloads tasks to a PMCA (Programmable Many Core Accelerator) made from RV32 (RISC-V 32-bit) cores, which implement extensions for machine learning and discrete signal processing. The suggested use-case for the SoC is in IoT applications and programmable embedded devices. The host core is Linux compatible, and offers a full OS that acts as a platform for programs that run on the PMCA. The use of a large RV64 core to allow a full Linux OS to run on the SoC provides a huge amount of flexibility to the programmer, as the OS implements features like CLI, memory virtualisation, networking and more that allow programs to be written much more generally than embedded software running without an OS. However, this usage of a full Linux OS could be considered excessive for the use-case. An embedded Linux OS would have a lower overhead due to the reduced services it offers, which is very beneficial in an embedded environment where efficiency is highly important. Unfortunately, no data is provided about the processing power or energy usage of the design.

2.4.2 Muntjac multicore RV64 processor[?]

Muntjac is an SoC generator comprising of multiple components, that can be used to produce a Linux capable SoC. There is only one type of core in the system, RV64, but the SoC can be multicore. The project report is dedicated to the design of the core and cache as opposed to usage in any devices, as the purpose is to provide an easily understood and extensible platform for specialised designs. This is excellently done - the project is very well presented and uses multiple open-standards like TileLink[**?**] to increase the ease of working with it. It would be possible for this project to be extended for a heterogeneous SoC design, but it appears that this has not yet been done in any public works that extend the project.

Chapter 3

Project Management

3.1 Methodology

The design and implementation of this project is expected to be experimentation based, as there are a huge number of factors to consider in the hardware design process and the final design will be based on how these factors affect performance. The hardware implementation and software tests design and implementation will be dependant on this design stage, and an agile methodology will be used to allow for changes in the objectives as the design experimentation progresses.

Chapter 4

Tools and Technologies

4.1 FPGA & Vivado

The FPGA development board used by this project is the Nexys-A7-100t. This development board uses the XC7A100T FPGA chip, with 128MiB of external RAM, a large array of IO ports and onboard peripherals. The FPGA chip contains 63400 LUTs, 126800 flip-flops, 2860Kb of block RAM and 240 DSP slices. This should be adequate for the design of a small SoC, as explained in 4.2, but there is risk that this won't be the case. As such, an alternative FPGA development board has been identified as the Nexys-Video. This uses the XC7A200T FPGA chip, with 134600 LUTs, 269200 flip-flops, 13Mb of block RAM and 740 DSP slices, and will definitely be adequate for our use case.

Connection to the FPGA development board will be made using USB serial and minicom[**?**], a basic serial interaction terminal program.

Once HDL has been generated for the SoC, we need to generate the bitstream to implement it on the FPGA. We will be using Vivado for this, a software suite for design, simulation, analysis, synthesis and implementation of digital circuits using HDL. Vivado has been chosen because of our familiarity with the software, and the availability of the software to us.

4.2 RocketChip

The aim of the project is to design a fully functioning RISC-V SoC that could potentially run a full OS, and attempting this from nothing would take much longer than the project timeline and is beyond the current skills of the author. Therefore, an SoC generator will be used, where we can create configuration files and semi-custom designs using high-level descriptions that have the full HDL created by software. We need a full SoC to be generated, not just CPUs, to include IO controllers, cache and other necessary components for a full computing system. Due to how niche RISC-V currently is, there is only one suitable candidate: RocketChip[**?**].

The RocketChip generator is a collection of smaller sub-generators that create parameterised components. These components fit together to create the full SoC via standard interfaces, allowing any design implementing the interfaces to be easily exchanged for another in the design. The most important to this project is the core, and RocketChip can generate three core types: RocketCore, BOOM and Z-scale Core. RocketCore and BOOM are both parameterisable, allowing a semi-custom design to be created. The default BOOM core (RV64GC) utilises around 148500 LUTs for a single core, and so cannot be used with the available FPGA. A default RocketCore (RV64GC) uses approximately 38300 LUTs for a single core, and 27500 for each default core after, giving 65800 LUTs necessary for a dual core design. This is greater than the number of LUTs in the FPGA, and so we will have to design cores smaller than these defaults. Exact details of the parameterisation options available in RocketCore are in the **??** chapter.

CPUs instantiated in a RocketChip SoC are inside of 'tiles' following the TileLink architecture. This contains them inside standard interface sections, and allows them to be added to them same bus with cache coherency both within and without the tiles.

4.3 vivado-risc-v

This projects extends an existing repository vivado-risc-v[**?**]. The vivado-risc-v repository is a collection of other tools and resources, along with additional scripts and sources, that can generate HDL (hardware description language) for RISC-V based SoCs, link them to hardware resources on specific FPGAs using Vivado and create the bitstream to program the FPGA. The repository also has scripts to generate a bootable external memory device, such as an SD card, with OpenSBI, U-Boot, Linux kernel and Debian OS. This allows a functional RISC-V SoC to be instantiated on an FPGA, boot into Debian Linux and be used as a general-purpose computer, albeit without GUI support. The Nexys-A-100t is directly supported by vivado-risc-v, so no changes had to be made to add support for the chosen FPGA board.

The main use of vivado-risc-v is board support and Vivado project generation. The HDL for the design is generated by the external tools, for example RocketChip to generate the SoC, sifive-cache to add more cache and testchipip for clocks, and the vivado-risc-v scripts add these to a new Vivado project setup for the FPGA board in use. vivado-risc-v also contains board-specific drivers for ethernet, SD card readers, serial connections and the device tree structure data for each board. These are necessary to connect the implemented SoC on the FPGA to the rest of the hardware on the development board.

4.4 Chisel

RocketChip is written in Chisel, a Scala-based HDL.

4.5 Bare-metal code

Once the SoC is implemented, the project aims to analyse the performance against homogeneous designs. As such, a testing suite needs to be developed to measure various performance metrics, running as bare-metal code. Bare-metal involves running as the only program on the system other than the bootloader - this means there is no OS to manage memory, threads, files and others. The code must also be written in a language than can be directly compiled to machine or byte-code, eliminating language such as Python or Java which are interpreted or run on a virtual machine. The languages with most support for RISC-V bare-metal are C/C++ and Rust, as these are often used for embedded programming. Another option is to write RISC-V assembly directly. This would make compilation much simpler, but prevents the use of libraries and is much harder to write complex programs - though it is likely the testing suite will be quite simple in order to accurately measure individual characteristics of the SoC.

Rust High-level language with speed comparable to C, memory/thread safety and excellent type system. Very detailed compiler messages and has a lot of features that would not be utilised in this usecase. Relatively easy to compile to RISC-V byte code, with crates (libraries) for low-level RISC-V programming.

C++ High-level language also with speed comparable to C. Also has many features that wouldn't be used for our purpose, similarly easy to compile to RISC-V byte code and has libraries for low-level RISC-V programming.

C Low-level language, very fast. Has easy direct memory manipulation, easy to compile to RISC-V byte code and has similar libraries to C++ for low-level RISC-V programming.

Assembly Byte code, will be very hard to write complex programs in but will be very easy to compile and debug in simulators by stepping through instructions individually.

C and Assembly are the chosen languages for the bare-metal code, due to its ease of compilation and how they can be easily mixed together. Assembly will be used to manually start the cores, jump to memory locations, etc, and C will be used to write the tests. This combination makes design and implementation of the tests easiest for the author.

Bare-metal programming will also require statically linked libraries. This means any libraries used must be compiled alongside the program and are part of the output byte code. Typical library usage is dynamically linked, where the compiled program references code in libraries that are identified when the program begins, and are loaded then. This is much harder to achieve in bare-metal, and reduces the portability of the final output.

4.6 Linux

Some of the software for the design and implementation of the SoC is only available on Linux. As such, a Linux install needs to be setup on the author's computer. We first attempted an install using windows subsystem for Linux, allowing Linux to be run semi-natively inside of Windows. This did not function as expected however, limiting the RAM available to 8GB and requiring interactions between Windows and Linux software, as well as reducing the overall performance due to extra overhead from windows. This made a large impact when doing synthesis and place and route, which are very computationally heavy tasks. We therefore installed Linux natively, solving the compatibility issues and increasing performance.

4.7 Git

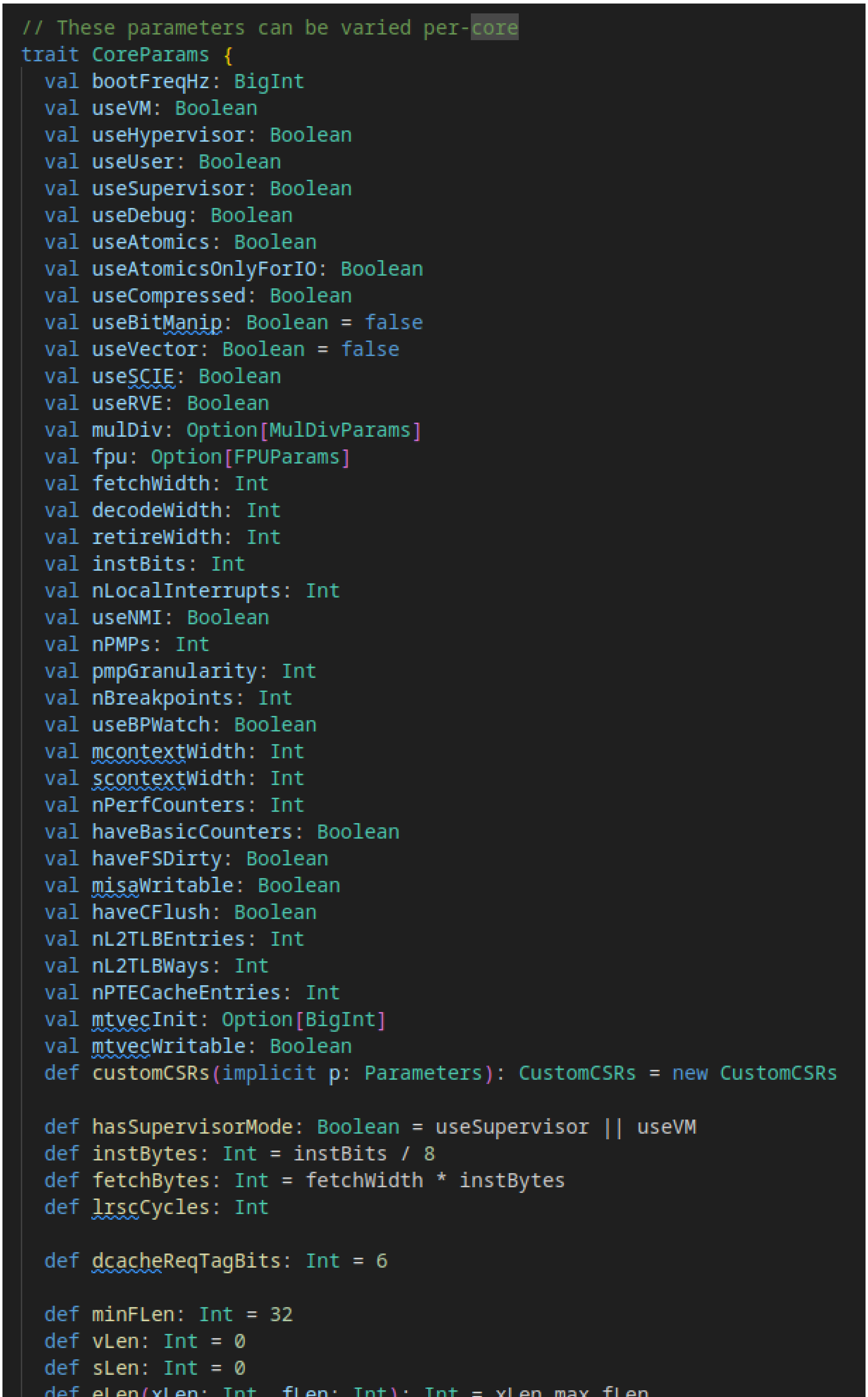
Git will be used for version control of the project. Version control is highly necessary, storing historical versions of the project and allowing multiple branches of the current state to test new features in isolation of other developments. GitHub is used as a remote repository, a backup of the project. This will also allow development to be synchronised across multiple devices, beneficial as work will be completed by the author in a multitude of places.

Chapter 5

Hardware Design

We decided to design and implement two unique RocketCore CPUs for the heterogeneous SoC. These will be in two separate tiles with cache coherency. The aim will be for each core to implement the RV64GC ISA, the 'general-purpose' RISC-V ISA, as well as an MMU (Memory Management Unit) in order to run a full Debian Linux OS, if only with terminal interaction. The following code snippets show the parameterisation options available for the RocketCore CPU and the tiles they lie within.

The micro-architecture specification of RocketCore is not publicised. This means implementation details of the core are not documented, and makes understanding how the core functions a difficult task. For example, to identify what changes occur when `useVM` is enabled instead of disabled, we must search through the source code (10000+ lines of Chisel[]) to find where the value is used and what it is used for - what other variables are impacted, how they change, etc. The following sections identify and discuss what customisation is available, with parameters common between all cores specified and parameters that will be varied during testing identified.

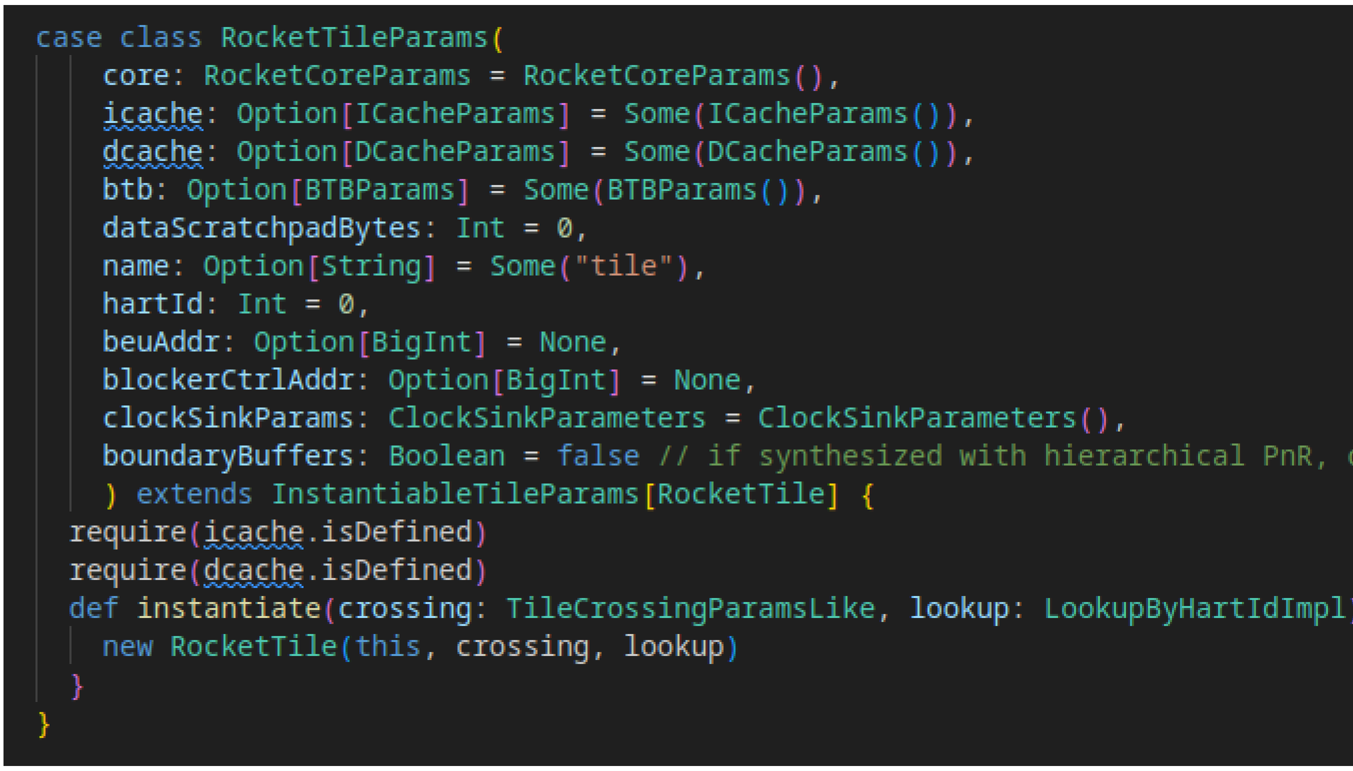


```scala
// These parameters can be varied per-core
trait CoreParams {
  val bootFreqHz: BigInt
  val useVM: Boolean
  val useHypervisor: Boolean
  val useUser: Boolean
  val useSupervisor: Boolean
  val useDebug: Boolean
  val useAtomics: Boolean
  val useAtomicsOnlyForIO: Boolean
  val useCompressed: Boolean
  val useBitManip: Boolean = false
  val useVector: Boolean = false
  val useSCIE: Boolean
  val useRVE: Boolean
  val mulDiv: Option[MulDivParams]
  val fpu: Option[FPUParams]
  val fetchWidth: Int
  val decodeWidth: Int
  val retireWidth: Int
  val instBits: Int
  val nLocalInterrupts: Int
  val useNMI: Boolean
  val nPMPs: Int
  val pmpGranularity: Int
  val nBreakpoints: Int
  val useBPWatch: Boolean
  val mcontextWidth: Int
  val scontextWidth: Int
  val nPerfCounters: Int
  val haveBasicCounters: Boolean
  val haveFSDirty: Boolean
  val misaWritable: Boolean
  val haveCFlush: Boolean
  val nL2TLBEntries: Int
  val nL2TLBWays: Int
  val nPTECacheEntries: Int
  val mtvecInit: Option[BigInt]
  val mtvecWritable: Boolean
  def customCSRs(implicit p: Parameters): CustomCSRs = new CustomCSRs

  def hasSupervisorMode: Boolean = useSupervisor || useVM
  def instBytes: Int = instBits / 8
  def fetchBytes: Int = fetchWidth * instBytes
  def lrscCycles: Int

  def dcacheReqTagBits: Int = 6

  def minFLen: Int = 32
  def vLen: Int = 0
  def sLen: Int = 0
  def eLen(xLen: Int, fLen: Int): Int = xLen max fLen
```

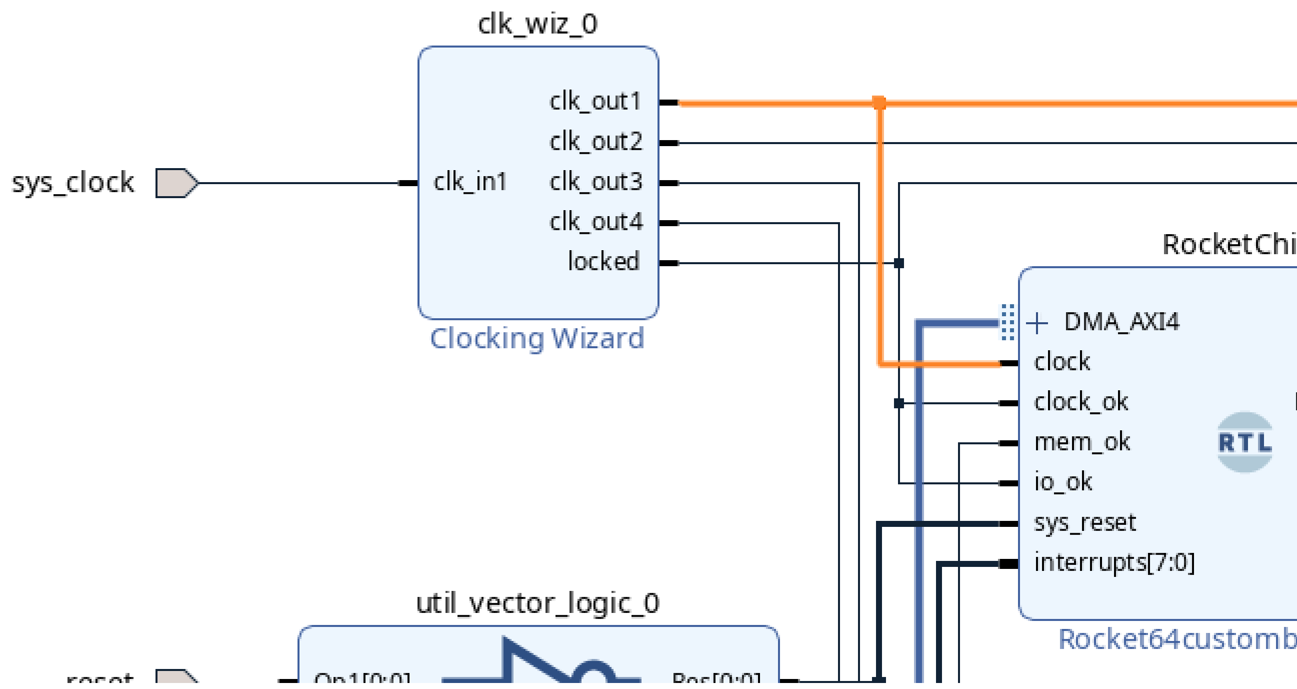


```
case class RocketTileParams(
    core: RocketCoreParams = RocketCoreParams(),
    icache: Option[ICacheParams] = Some(ICacheParams()),
    dcache: Option[DCacheParams] = Some(DCacheParams()),
    btb: Option[BTBParams] = Some(BTBParams()),
    dataScratchpadBytes: Int = 0,
    name: Option[String] = Some("tile"),
    hartId: Int = 0,
    beuAddr: Option[BigInt] = None,
    blockerCtrlAddr: Option[BigInt] = None,
    clockSinkParams: ClockSinkParameters = ClockSinkParameters(),
    boundaryBuffers: Boolean = false // if synthesized with hierarchical PnR,
    ) extends InstantiableTileParams[RocketTile] {
  require(icache.isDefined)
  require(dcache.isDefined)
  def instantiate(crossing: TileCrossingParamsLike, lookup: LookupByHartIdImpl
    new RocketTile(this, crossing, lookup)
  }
}
```

5.1 Constant elements in RocketCore

5.1.1 Core Frequency - `bootFreqHz`

The `bootFreqHz` core variable does not impact the actual frequency of the core, but instead is added to the device tree structure, so that an operating system is able to read the boot frequency of the CPU. The actual frequency of the cores are defined by clocks generated in the Vivado project. These can be edited, with supported frequencies of 160, 125, 100, 80, 62.5, 50, 40, 31.25, 25, and 20 MHz. The clock manager uses phase locked loop and counters to divide the central clock from the FPGA crystal oscillator. This reduces the available frequencies to those that can be formed from that central clock using the dividers. The Nexys-A7-100t has a crystal capable of generating up to 50 MHz, and is shared between all cores in the RocketChip SoC. This prevents individual clock frequencies for cores, one of the significant changes between big and small cores in typical heterogeneous systems. This will severely limit



the actual performance difference between cores, as well as the power consumption difference. This also prevents frequency scaling as the RocketChip SoC does not have control of the clock generator, another key part of modern power saving measures. These are limitations of FPGAs as a platform and the SoC generators used in this project.

5.1.2 ALU

The integer ALU is not customisable within RocketCore. This results in all RocketCore CPUs having the same integer additions per cycle, and as we cannot vary the frequency between cores in the FPGA, all our designs will have the same integer operations per second.

5.1.3

5.2 MMU

The `useVM` option for core parameters controls whether an MMU (Memory Management Unit) is instantiated inside the core and virtual memory is used. MMU is enables the use of Virtual Memory, an abstraction of physical addresses to logical addresses. This increases the security and stability of a system, by preventing processes from accessing the memory space of other processes. Stopping reads to another processes' memory space ensures sensitive data currently held in memory by a process cannot be accessed by another process. Stopping writes to the memory space prevents the corruption of another processes' data, that could otherwise lead to user data loss, the process stopping unexpectedly or entire system crash depending on the importance of the memory corrupted process. Virtual memory can also allow more logical memory than there is physical memory to be allocated to programs.

The MMU in RocketCore is non-parameterisable - it is either instantiated inside the CPU or it isn't. We have chosen to enable virtual memory and have an MMU instantiated inside the cores. Support for the Linux kernel and Debian requires an MMU, hence it is needed in the cores.

5.3 User/Supervisor modes

The `useUser` and `useSupervisor` options control the addition of hardware privilege levels. All RISC-V CPUs have the machine privilege level (M-mode). Code run in M-mode is able to execute any instruction, including those that would result in the CPU becoming trapped permanently. M-mode is intended to be used for managing secure execution done in lower privilege levels, completely trusted code that must be run on M-mode for system setup or for embedded systems where M-mode is the only privilege level implemented.

User mode (U-mode) can be added to the CPU to provide a secure environment where code can be executed safely, and wouldn't be able to cause serious damage to the system. The CPU keeps track of state at prevents this, but current privilege level is not visible to software as this would become a virtualisation hole - a way for a program to attempt to escape or see outside it's current virtualisation level. Privileged instructions cannot be run in U-mode, preventing access to registers such as `mtvec`, keeping the address of the machine trap vector and mode. U-mode is typically implemented for secure embedded systems, where a level of privilege is required to ensure the system does not fail when application code is run.

Supervisor mode (S-mode) can also be added. This privilege level fits between U-mode and M-mode, adding many S-mode registers that only M-mode had previously, such as `SPP` for the previous privilege mode or `stvec` for the supervisor trap vector address and mode. S-mode is utilised most often by OSs that want to be able to run applications on top of themselves and keep functioning, so must prevent such applications from being able to maliciously or accidentally interfere with it's operation.

As the SoC designed by this project will never be put into a production environment, we can disable user and supervisor mode options. However a requirement for virtual memory and implementing the MMU in RocketCore is the inclusion of supervisor mode, and so the the extensions will be implemented regardless of the options. While the U and S-modes are implemented, we have chosen to execute all code in M-mode. This removes hardware security from the system, but the only code being run will be either written by the author or from trusted open-source projects. In addition to this, the system is completely isolated and would be unable to cause any damage if compromised, only connected to another system via serial. There are also sections of the code that must be run in M-mode, like fetching from performance monitoring registers, and it is easier to execute fully in M-mode than to write traps and handlers in assembly to create M-mode code to fetch data from the registers separately.

5.4 Hypervisor mode

`useHypervisor` option enables the hypervisor mode extension. Hypervisor mode extends supervisor mode, adding additional virtualisation. S-mode becomes HS-mode, where a hypervisor program or hosting-capable OS runs, and changes some registers typically accessed from `Sxxx` to `Hxxx`, such as `SIE` to `HIE`. A virtualisation bit is added and set when the CPU is executing a guest OS, and this changes the HS-mode and U-mode to VS-mode and VU-mode. An additional layer of address translation is enabled when this occurs, as well as the registers accessed in VS-mode returning to the `Sxxx` versions.

This mode is used for systems where multiple OSs may be running concurrently, such as servers or workstation PCs. This project does not aim to design an SoC to be used in this context and we can safely disable this option.

5.5 Debug mode

Debug mode has been preliminarily proposed as an extension to the RISC-V ISA. The `useDebug` option controls the implementation of this extension. When enabled, this generates hardware in the core to enable debugger control of the core, as well as a debugger module in the SoC. The changes to the core include addition of CSR registers, writable only by an external debugger interacting with the debug module, that can halt execution, force jumps, and other control functions. The debug module allows an external debugger to get the contents of core data registers, cache, RAM, and CSRs as needed, enabling someone testing hardware to inspect the status of the core during program execution. The debug module also provides the capability to 'step through' the instructions, a very useful feature when identifying the exact instruction where a program deviates from expected execution.

We have enabled this option during the design phase, as the debug features it provides is very useful when designing the test software. For the actual core under test the option was disabled, to reduce the FPGA usage for comparisons and maximise the available resources for additional processing hardware.

5.6 Atomics

The `useAtomics` and `useAtomicsOnlyForIO` options enable parts of the Atomic RISC-V extension. The atomic extension implements instructions allowing for read-modify-write to memory and IO while keeping memory consistency between multiple RISC-V harts utilising the same memory/IO.

Base RISC-V ISA has a relaxed memory model, allowing loads and stores to take place in any order if unspecified. The atomic extension implements load-reserved and store-conditional instructions that can apply additional ordering constraints on memory/IO operations, and ensure data is valid during operations performed on it. There are two bits used to specify acquire and release ordering requirements for this RISC-V hart: when the acquire bit is set, no memory operation instructions following the acquire-set instruction can take place before the acquire-set operation; when the release bit is set, the release-set instruction must take place after any previously issued memory operations. The combination of these bits allow for sequential ordering of memory operations for a RISC-V hart can be asserted. These instructions are primarily utilised for out-of-order CPUs, where the ordering of issued instructions does not necessarily match the order of instruction execution, leading to race conditions without atomically defined instructions.

The atomics also implement instructions for multicore synchronisation. Atomic write-modify instructions for memory perform changes that occur uninterrupted with a single instruction. These also implement the acquire and release bits, and allow for swapping, addition, bitwise operations and maximum/minimum instructions to be completed on data held in shared memory without other harts interefering.

As the system we are designing is multicore, we enabled the `useAtomics` option. Not allowing simultaneous memory access in a multicore system would cause a severe bottleneck in most cases and is the only other solution to ensure memory validity in this case.

5.7 Compressed instructions

Enabling the compressed instruction set extension is enabled using the `useCompressed` option. The extension adds support for 16-bit versions of common RISC-V instructions and are utilised whenever possible to reduce the code-size when compiled. In order to use 16-bit instructions instead of 32-bit, the data embedded in the instruction must be able to be stored in a reduced amount of bits than provided in 32-bit: for instance, when performing integer additions where a source and destination register are specified, if the registers are the same then the instruction can likely be compressed. The compressed extension affects the base ISA (RV64I) as well as the single-precision and double-precision extensions, though this must be supported in the implementation of these instead of the compressed extension.

With compressed enabled, the code-size can typically be reduced by 25-30%, with 50-60% of 32-bit instructions being replaced with 16-bit counterparts. This is a huge reduction, and is especially useful in embedded systems where storage space can be extremely limited. We enabled this option to add compressed instruction compatibility to the design, and allowing the code-size reductions that it provides.

5.8 Bit manipulation and Vector

`useBitManip` and `useVector` options exist in the `CoreParams` class used for defining core designs. However, RocketCore is not advertised as implementing these extensions and so their function was unknown. Examining the Rocket-Chip code that utilises these options reveals a partial implementation of the vector extension, where CSR registers are added for control, as well as logic to update dirty bits and other status bits. This is the only vector code implemented however - there are no registers or other logic defined for the vector extension. Instructions have been defined in Chisel for the vector operations, but there is no decode logic defined for the instructions and so they cannot be used. The `useBitManip` option only adds the bit manipulation extension letter to the ISA string held in the CSR, making no other changes. As such, these options have been left disabled in the design as they serve no functional purpose.

5.9 SCIE

Custom instructions support is enabled within RocketCore by enabling the `useSCIE` option (Si-Fice Custom Instruction Extension). This enables support for custom instructions, producing a custom instruction logic interface. Connected custom instruction logic is supplied with the current instruction, data from two registers and can output register data and a dispatch code. As we are not writing any custom instructions for this project, the extension is unnecessary and we have not enabled it in the designs.

5.10 RVE

`useRVE` changes the base ISA to RV32E instead of RV32I when enabled. RV32E supports the same instructions and extensions as RV32I, but removes the top 16 general purpose registers, limiting the core to `x0` to `x15`. This can reduce the area used by a RV32 core by up to 25% when no other extensions are enabled. As our design targets the RV64 architecture, this option is incompatible and disabled.

5.11 Integer Multiplier

The integer multiplier unit in RocketCore is customised using a set of parameters set in `MulDivParams` and passed to the `MulDiv` class for hardware creation. The parameters for this vary between the design scenarios, as seen in

5.11.1 Unrolling

The multiplier unit allows for the unrolling of both multiplier and division instructions. Unrolling controls how much of the multiplication operation is completed in a single clock cycle. Without unrolling, multiplier latency is 64 cycles in RV64 architecture, with a single bit computed per cycle. Increasing the unrolling factor decreases the latency in a ratio of `64/unroll` for the multiplier and `1 + 64/unroll` for divider. The inverse relation between unroll factor and latency means increasing the factor can provides dramatic decreases for small factors, but diminishing returns are achieved as the unroll factor increases. An issue with increasing the unroll factor is the constraints placed on the clock speed. The increased amount of computation performed each cycle increases the minimum required time period to finish and may violate the setup time if the clock speed is not adjusted accordingly.

Unrolling multiplication and division operations result in unequal changes to timing constraints - each individual division operation takes longer than a multiplication operation. To maintain high clockspeed, division operations cannot be unrolled as much as multiplication operations.

5.11.2 Early out

The multiplier unit also has an option for early output of results when possible. A mask is applied to the input and the early output bit is set when the result can be read. Early output can result in a large latency decrease depending on the situation, as small integer multiplications will result in smaller outputs that will likely not require all integer bits and can make use of the early out. The minimum latencies for multiplication and division is reduced to 2 cycles and 3 cycles in a best case scenario.

Granularity for the early out can also be set for the division operations.

5.12 Floating Point Unit

The floating point unit (FPU) can also be customised in a limited way, using the `FPUParams` class passed to the FPU generator. Setting this class to `None` instead of instantiating it results in no FPU initialised, and the ISA lacks F or D extension support. FPUs are much more complicated than integer operation units, with the RocketCore implementation including subunits for integer to FP conversion and vice-versa, pipelines and register file. Use of the FPU is discussed more in .

5.12.1 Floating point lengths

The minimum float length (`minFLen` in the `FPUParams class`) can be changed to support half-precision floating points of 16 bits, and the maximum float length `fLen` changed to support double-precision floating points. The FPU in RocketCore does not support quad-precision floating points of 128-bits, though this is not widely used in CPUs.

5.12.2 FMA Latency

The RocketCore FPU supports fuse multiply-add operations, where multiplication and addition are performed with a single operation. These are pipelined instructions for RocketCore, and the latency can be specified at instantiation. The cycle latency choice is bounded, allowing 1 to 3 cycles. Similar to the multiplier unrolling, decreased latency can come at the cost of increased clock period, decreasing clock frequency, but may overall be an improvement in FPU performance.

5.12.3 Squareroot

The `divSqrt` option enables a dedicated square root function in the FPU, allowing the square root of numbers to be found faster than regular division. This is a useful function due to the increased use of square roots in modern CAD tools and 3D graphics processing, where slow square root can bottleneck a processor[**?**].

5.13 Local Interrupts

Additional local interrupts can be added to the RocketCore using `nLocalInterrupts` option. These additional interrupts are added to the CSRs and are accessible from all implemented modes. No additional local interrupts are necessary for this design, as there are already multiple available within the standard CSRs.

5.14 Non-maskable interrupt

The use of non-maskable interrupts can be disabled in RocketCore, meaning it would technically not meet RISC-V specification when disabled. However, enabling/disabling this option has no functional difference for software as NMI is used exclusively for hardware error conditions and cause an immediate jump to the NMI vector in M-mode to address the issue. NMIs have been disabled in our implementation as we do not foresee a use for them in the project.

5.15 Hardware Breakpoints

Hardware breakpoints can be added to the core for use with debuggers with the `nBreakpoints` option. This adds hardware resources to set breakpoints in running code, using triggers on address/data/memory operations. For testing purposes, we've included a single hardware breakpoint in each core, though actual usage of the hardware breakpoints is not anticipated due to the existence of software breakpoints in the RISC-V specification, as well as the debug module that can already be used to stop and step through sections of programs.

5.16 Physical Memory Protection

A custom amount of physical memory protection regions can be defined in RocketCore with `nPMPs`. PMP is used to limit or increase the physical address space available to software, to increase security and reduce faults. This primarily applies to U and S-mode software, where regions of memory are typically blocked off when using an OS or similar host software and security is required. PMPs can also optionally be configured to apply to M-mode accesses. U and S-mode are not implemented in this project and we will not require security for this design, but future programs utilising the designs might. We elected to include 8 PMP regions, as this was the amount in a default big core.

5.17 Performance Counters

5.18 Cache flush

RocketCore implements a custom instruction for fully flushing the data cache, enabled with `haveCFlush`. The instruction has been disabled for our designs.

5.19 Writable ISA

RocketCore allows writing to the register holding ISA details, which is automatically configured and written to in the design. This has been disabled for our designs as the automatic ISA is accurate.

5.20 L2 TLB

An L2 TLB can be configured on-CPU in RocketCore. Due to the size constraints on our designs, we have not implemented this for our cores.

5.21 Page Table Entry Cache

We've configured our designs with an 8 entry PTE cache. Code being run on the design will be very small and likely not span more than a single page, but the design should be suitable for future programs that may take more space, and so would benefit from a larger PTE cache.

5.22 Machine Trap Vector

The machine trap vector is configurable in RocketCore. The startup value can be set, as well as preventing the trap vector from being modified. Our designs init with a trap vector of `0x0` and the trap vector is writable - the boot loader sets the trap vector on startup, so setting a default value is not required.

5.23 Fast loading

RocketCore allows data to be loaded directly to the core when fetching from memory when enabled via the `fastLoadWord` and `fastLoadByte` options. Typically, a cache miss occurs and data is fetched from memory into cache, and then loaded from cache to the core. Enabling these options add bypasses, allowing data to be loaded directly from memory into the core. Having both of these options enabled is allowed, but only one is ever used - `loadFastByte` takes priority during compilation to RTL. Our designs use `loadFastWord`, allowing a full word to bypass the cache instead of just the first byte.

5.24 Branch Prediction CSR

RocketCore has the option to add a custom branch prediction CSR. This contains two bits, used for requesting to completely flush the branch target buffer and to set the branch prediction CSR to static. We disabled this option in our design.,

5.25 Clock Gating

5.26 Instruction Cache (L1)

The instruction cache on the core is customised using the `ICacheParams` class.

`nSets` Amount of sets in the cache

`nWays` Amount of ways in each set

`rowBits` Appears to have no function in `ICache`

`nTLBSets` Amount of sets in the cache TLB

`nTLBWays` Amount of ways in each TLB set

`nTLBBasePageSectors` How many sectors to divide each page into for easier memory loading

`nTLBSuperpages` Amount of superpages - much larger page than normal, providing larger TLB coverage

`tagECC` ECC for tags in cache

`dataECC` ECC for cache data

`itimAddr` Instruction Tightly Integrated Memory base address and enables ITIM if not `None`

`prefetch` Enables pre-fetching, getting next cache line in advance

`blockBytes` Size of each cache line

`latency` Latency of a fetch instruction, 1 or 2 cycles. Decreased latency limits clock frequency

`fetchBytes` Bytes fetched by CPU for each cycle

ECC and ITIM have been disabled in our core designs to keep the cores as simple possible and reduce possible factors that could influence performance.

5.27 Data Cache (L1)

The data cache on the core is customised using the `DCacheParams` class.

`nSets` Amount of sets in the cache

`nWays` Amount of ways in each set

`rowBits` Appears to have no function in `DCache`

`subWordBits` Amount of bits in each subword in the cache word

`replacementPolicy` Replacement policy - valid options are random, least recently used and pseudo least recently used

`nTLBSets` Amount of sets in the cache TLB

`nTLBWays` Amount of ways in each TLB set

`nTLBBasePageSectors` How many sectors to divide each page into for easier memory loading

`nTLBSuperpages` Amount of superpages - much larger page than normal, providing larger TLB coverage

`tagECC` ECC for tags in cache

`dataECC` ECC for cache data

`dataECCBytes` Bytes used for the tagECC

`nMSHRs` Number of miss status holding registers, tracking outstanding cache misses

`nSDQ` Store Data Queue buffers storing data from execute unit

`nRPQ` Queue config option for MSHRs

`nMMIOs` Memory mapped IO cache entries

`blockBytes` Bytes per data cache line

`separateUncachedResp` Uncertain function

`acquireBeforeRelease` Uncertain function

`pipelineWayMux` Uncertain function

`clockGate` Enable clock gating in the data cache

`scratch` Enable scratchpad in data cache and define size

To reduce complexity of the design and keep performance affecting factors to a controllable amount, the options for `nSDQ`, `nRPQ`, `nMMIOs`, `separateUncachedResp`, `acquireBeforeRelease` and `pipelineWayMux` have all been left as set in the RocketCore example big core. Scratchpad memory has been disabled in all caches, as full data cache is implemented. ECC options have also been disabled in the cores, removing unnecessary features like those for fault tolerant systems. The clock gating option has also been disabled - attempting synthesis with this enabled resulted in missing connections, and implementation on the FPGA was not possible.

5.28 Branch Target Buffer and Branch History Table

A BTB and BHT can also be created by RocketCore. The BTB is a fully-associative cache, mapping instruction addresses to predicted PC contents. The RockerCore implementation has parameterisable amount of entries, match bits, pages, return address stacks and out-of-order updates.

`nEntries` BTB entries

`nMatchBits` Instruction address matching bits

`nPages` BTB pages

`nRAS` Number of return address stacks

`updatesOutOfOrder` Enable updating BTB out of order

The BTB contains the BHT, tracking the history of whether a branch is taken or not. The BHT has a parameterisable amount of entries, counter length (1 or 2 bits), history length and history bits.

`nEntries` Pattern entries in the BHT

`Counter length` Number of prediction bits

`historyLength` History length

`historyBits` Number of bits in each history entry

5.29 Final Design Scenario Limitations

Both custom cores implement the RV64IMAZicsrZifenceiC ISA, with MMU. Experimentation with FPUs was attempted, and different cores successfully synthesised and implemented on the FPGA with various FPU configurations. However, attempting to synthesis a dual core design was unsuccessful - the amount of LUTs required for such a design was always greater than the amount available on the FPGA, no matter the reductions made in other options like cache.

Cores with different ISAs were considered as a solution to this approach. We first verified that this was possible, and successfully implemented a dual core SoC with a large FPU-enabled core and a small FPU-disabled core. While a functional hardware design, the software design for this SoC would be much harder due to the differing ISAs. Running the Linux kernel would also be more complicated: a custom kernel can be compiled to use soft-float (running floating point operations using integer ALU and registers) instead of FPU hardware, this would prevent the FPU actually being used by Linux and programs run within the OS, thus rendering the FPU in the big core pointless.

Another option considered was to share the FPU between cores, but research into this showed RocketChip is not capable of such designs. As such, the final design has not been able to include an FPU and we dropped support for the F and D ISA extensions.

5.30 Final Design Scenario - Custom Big Core

5.30.1 Multiplier

The multiplier in the big core is unrolled by a factor of 16. This results in a minimum cycle latency of 4 cycles (assuming no early out), much fewer than 64 cycles with no unrolling. Multiplication early out is also enabled, giving an actual minimum cycle latency of 2 cycles. Division has been unrolled by a factor of 4, bringing minimum cycle latency to 16 cycles. Further increases in division unrolling incurred failing time constraints, preventing further unrolling without decreasing the clock frequency. We decided that compromising on clock frequency would bring greater reductions in real-world performance than increasing division performance would, though if only measuring in cycles this would only be a benefit to the CPU. Division early out is enabled and gives a minimum cycle latency of 3 cycles when applicable.

5.30.2 Instruction Cache

The instruction cache contains 64 sets of 4 ways, giving 256 lines of 64 bytes and 16KB of cache total. This is a reasonable amount of instruction cache for a small core, and decreases the amount of capacity cache misses that occur. A TLB for the instruction cache has also been implemented, with 2 sets of 32 ways for 64 entries. The TLB uses 4 sectors per base page and 4 super pages to reduce excess swapping of large amounts of data between memory and storage.

5.30.3 Data Cache

The data cache is the same 64 sets of 4 ways for a total of 16KB. The TLB is also the same size and configuration as instruction cache. The number of MSHR has been set to 1, providing a single miss status handling register. This should be adequate, and is the same amount as in the example big RocketCore.

5.30.4 BTB and BHT

A large BTB has been instantiated in the big core, with 28 entries, 14 match bits, 6 pages and 6 return address stacks. This is a standard large BTB for RocketCore, and should provide reasonable performance for our use. The BTB also contains the BHT, which has been implemented with the example big RocketCore values: 512 entries, counter length of 1, history length of 8 and 3 history bits.

5.30.5 Code Listing

```
class WithNHetBigCores(n: Int, overrideIdOffset: Option[Int] = None)
    extends Config((site, here, up) => {
    //smaller big core for hetero on small FPGA, RV64
    case RocketTilesKey => {
        val prev = up(RocketTilesKey, site)
        val idOffset = overrideIdOffset.getOrElse(prev.size)
        val big = RocketTileParams(
        core = RocketCoreParams(
            bootFreqHz = 0,
            useVM = true, // MMU enabled
            useUser = false,  // User/Super/Hyper disabled
            useSupervisor = false,
            useHypervisor = false,
            useDebug = true,
            useAtomics = true,
            useAtomicsOnlyForIO = false,
            useCompressed = true,
            useRVE = false,  // Non-embedded
            useSCIE = false,  // No custom instructions
            nLocalInterrupts = 0,
            useNMI = false,
            nBreakpoints = 1,
            useBPWatch = false,
            mcontextWidth = 0,
            scontextWidth = 0,
            nPMPs = 8,
            nPerfCounters = 16,
            haveBasicCounters = true,
            haveCFlush = false,
            misaWritable = true,
            nL2TLBEntries = 0,
            nL2TLBWays = 1,
            nPTECacheEntries = 8,
            mtvecInit = Some(BigInt(0)),
            mtvecWritable = true,
            fastLoadWord = true,
            fastLoadByte = false,
            branchPredictionModeCSR = false,
            clockGate = false,
            mvendorid = 0, // 0 means non-commercial implementation
            mimpid = 0x20181004, // release date in BCD
            fpu = None,
            mulDiv = Some(MulDivParams(
            mulUnroll = 16,
            divUnroll = 4,
            mulEarlyOut = true,
            divEarlyOut = true))),
        btb = Some(BTBParams( // Large BTB
            nEntries = 28,
            nMatchBits = 14,
            nPages = 6,
            nRAS = 6,
            bhtParams = Some(BHTParams()), // default BHT
            updatesOutOfOrder = false)),
        dcache = Some(DCacheParams( // Large D-cache
            rowBits = site(SystemBusKey).beatBits,
            nSets = 64,
            nWays = 4,
            nTLBSets = 2,
            nTLBWays = 32,
            nMSHRs = 1,
            clockGate = false,
            blockBytes = site(CacheBlockBytes))),
        icache = Some(ICacheParams( // Large I-cache
            rowBits = site(SystemBusKey).beatBits,
            nSets = 64,
            nWays = 4,
            nTLBSets = 2,
            nTLBWays = 32,
            prefetch = true,
            blockBytes = site(CacheBlockBytes))))
        List.tabulate(n)(i => big.copy(hartId = i + idOffset)) ++ prev
    }
})
```

5.31 Final Design Scenario - Custom Small Core

5.31.1 Multiplier

The multiplier in the small core hasn't been unrolled, giving a minimum cycle latency of 64. This has been done to reduce area and power usage - each unroll adds another bit multiplier logic section, so moving from no unrolling to a factor 2 unrolling would double the amount of logic between the stages, factor 4 unroll giving a 4x increase, etc. The early out option has also been disabled, meaning there can be no reduction in multiplier cycle times, but again reducing the multiplier logic in the core. Division similarly has no unrolling and no early out, and has a minimum cycle latency of 64.

5.31.2 Instruction Cache

The instruction cache contains 64 sets of 1 way, giving 64 lines of 64 bytes and 4KB of cache total. This is 1/4 of that afforded to the big core and will result in a significant increase in cache misses, but will also provide a decrease in the power and area usage by the core. A TLB for the instruction cache has also been implemented, with 1 sets of 16 ways for 16 entries. The TLB uses 4 sectors per base page and 4 super pages, same as the big core.

5.31.3 Data Cache

The data cache is the same 64 sets of 1 ways for a total of 4KB. The TLB is also the same size and configuration as instruction cache. The number of MSHR has been set to 0, removing them entirely.

5.31.4 BTB and BHT

The BTB and BHT have been entirely removed from the core. Removing the ability to predict branches entirely will result in a large decrease in performance, especially in cases where there is a large amount of branching instructions. However, this core is designed for minimising area and power usage, not performance, so the performance cost is justified by the decreases in area and power usage.

5.31.5 Code Listing

```
class WithNHetSmallCores(n: Int, overrideIdOffset: Option[Int] = None)
    extends Config((site, here, up) => {
    //Smaller small core for hetero on small FPGA
    case RocketTilesKey => {
        val prev = up(RocketTilesKey, site)
        val idOffset = overrideIdOffset.getOrElse(prev.size)
        val small = RocketTileParams(
        core = RocketCoreParams(
            bootFreqHz = 0,
            useVM = true, // MMU enabled
            useUser = false,  // User/Super/Hyper disabled
            useSupervisor = false,
            useHypervisor = false,
            useDebug = true,
            useAtomics = true,
            useAtomicsOnlyForIO = false,
            useCompressed = true,
            useRVE = false,  // Non-embedded
            useSCIE = false,  // No custom instructions
            nLocalInterrupts = 0,
            useNMI = false,
            nBreakpoints = 1,
            useBPWatch = false,
            mcontextWidth = 0,
            scontextWidth = 0,
            nPMPs = 8,
            nPerfCounters = 16,
            haveBasicCounters = true,
            haveCFlush = false,
            misaWritable = true,
            nL2TLBEntries = 0,
            nL2TLBWays = 1,
            nPTECacheEntries = 8,
            mtvecInit = Some(BigInt(0)),
            mtvecWritable = true,
            fastLoadWord = true,
            fastLoadByte = false,
            branchPredictionModeCSR = false,
            clockGate = false,
            mvendorid = 0, // 0 means non-commercial implementation
            mimpid = 0x20181004, // release date in BCD
```

```
            fpu = None,
            mulDiv = Some(MulDivParams( //small mul
            mulUnroll = 1,
            mulEarlyOut = false,
            divEarlyOut = false
            ))),
        btb = None, //no branch prediction
        dcache = Some(DCacheParams( //reduced D-cache
            rowBits = site(SystemBusKey).beatBits,
            nSets = 64,
            nWays = 1,
            nTLBSets = 1,
            nTLBWays = 16,
            nMSHRs = 0,
            clockGate = false,
            blockBytes = site(CacheBlockBytes))),
        icache = Some(ICacheParams( //reduced I-cache
            rowBits = site(SystemBusKey).beatBits,
            nSets = 64,
            nWays = 1,
            nTLBSets = 1,
            nTLBWays = 16,
            blockBytes = site(CacheBlockBytes))))
        List.tabulate(n)(i => small.copy(hartId = i + idOffset)) ++ prev
    }
})
```

Chapter 6

Software Design

6.1 Bootloader

6.2 Linker Scripts

6.3 Benchmark Design

Several benchmarks have been designed to measure the performance of the CPU. As the main differences between the big and small cores are in the multiplier, cache and branching, we will assess performance in these areas. To achieve this, four benchmark programs have been written: addition, matrix multiplication, memory stress and a mixed branching benchmark.

Measurement of performance will be done in cycles instead of seconds. This decision has been made as the frequency of all cores is the same in the FPGA and so measuring the difference between cycles will be equivalent to measuring the actual time taken for the programs to run.

Each benchmark starts by identifying the current CPU through the hart register and starting the other hart if the system is multicore and core is hart `0`. The current cycle count held in the `mcycle` register is read, benchmark started and the cycle register read again, with the difference between the two reads being the amount of cycles the benchmark took.

The benchmarks are also customisable with varying array sizes and iterations. 100 iterations of each benchmark are completed by default - this is to reduce the amount of impact run to run variance has on the final results.

6.3.1 Addition

The addition benchmark has been added to assess the addition performance differences. The integer addition unit is the same between all RocketCores, but performance is still expected to vary some amount due to differences in branching, cache, etc.

The addition benchmark performs a loop of additions on two 32-bit integer arrays, storing the result in a final 32-bit integer array. Additions are performed on elements in the arrays in an order similar to how matrix multiplication is performed. This is to reduce the memory fetching overhead, reusing array elements for multiple additions and forcing the cycle count to depend more on the arithmetic logic.

```c
void addition_iter(uint32_t *in_a, uint32_t *in_b, uint32_t *res) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            res[i] = 0;
            for (int k = 0; k < SIZE; k++) {
                res[i] += in_a[i * SIZE + k] + in_b[k * SIZE + j];
            }
        }
    }
}
```

6.3.2 Matrix Multiplication

The matrix multiplication benchmarks measures the cycles to compute the multiplication of two 32-bit integer matrices. Multiplication should dominate the amount of cycles taken to execute and provide a reasonable comparison between CPUs.

```
void matrix_bench(uint32_t *in_a, uint32_t *in_b, uint32_t *res) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            res[i] = 0;
            for (int k = 0; k < SIZE; k++) {
                res[i] += in_a[i * SIZE + k] * in_b[k * SIZE + j];
            }
        }
    }
}
```

6.3.3 IO Benchmark

The IO benchmark repeatedly requests data from opposite ends of the input arrays and sets the result array. By fetching from opposite ends, the program aims to increase the amount of cache misses as the array data should be stored contiguously in element order.

```
void io_iter(uint32_t *in_a, uint32_t *in_b, uint32_t *res) {
    for (int i = 0; i < SIZE; i++) {
        res[i] = 0;
        res[i] = in_a[i];
        res[i] = in_b[i];
        res[i] = in_a[SIZE-i-1];
        res[i] = in_b[SIZE-i-1]; //increase cache misses
    }
}
```

6.3.4 Mixed Benchmark

The mixed benchmark is designed to perform computations, data fetching and branches in a way representative of real-world applications.

```
void mixed_bench(uint32_t *in_a, uint32_t *in_b, uint32_t *res, uint32_t sel) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            sel = 1 - sel;
            res[i] = 0;
            if (sel) {
                for (int k = 0; k < SIZE; k++) {
                    res[i] += in_a[i * SIZE + k] * in_b[k * SIZE + j];
                }
            }
            else {
                for (int k = 0; k < SIZE; k++) {
                    res[i] += in_a[i * SIZE + k] + in_b[k * SIZE + j];
                }
            }
        }
    }
}
```

6.3.5 title

6.4 Software Verification

Software verification was completed using the Spike RISC-V emulator. Spike is a functional model of a RISC-V processor, with options for number of harts (hardware threads, RISC-V representation of logical cores), and ISA. Spike supports all major ratified extensions, as well as some still in the proposed stage. As our final hardware implementation uses RV64IMAZicsrZifenceiC, we can set Spike to simulate the same core type when invoking from the command line.

Chapter 7

Analysis

Comparison between homogeneous and heterogeneous designs will be made by benchmarking individual SoCs. In the case of multicore SoCs, the benchmark programs are run in tandem

Bibliography

[] Apple, . Apple unveils m2. `https://www.apple.com/uk/newsroom/2022/06/apple-unveils-m2-with-breakthrough-performance-and-capabilities/`, 2022.

[] AsanoviÄ, Krste & Avizienis, Rimas & Bachrach, Jonathan & Beamer, Scott & Biancolin, David & Celio, Christopher & Cook, Henry & Dabbelt, Daniel & Hauser, John & Izraelevitz, Adam & Karandikar, Sagar & Keller, Ben & Kim, Donggyu & Koenig, John & Lee, Yunsup & Love, Eric & Maas, Martin & Magyar, Albert & Mao, Howard & Moreto, Miquel & Ou, Albert & Patterson, David A. & Richards, Brian & Schmidt, Colin & Twigg, Stephen & Vo, Huy & Waterman, Andrew. The rocket chip generator. Technical Report 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`.

[] Dai Clegg, Richard Barker. *Case Method Fast-Track: A RAD Approach*. Addison-Wesley, 1994. ISBN 978-0-201-62432-8.

[] Editors Andrew Waterman, Krste Asanovic & RISC-V InternationalJohn Hauser, December 2021. The risc-v instruction set manual, volume ii: Privileged architecture, document version 20211203. `https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf`.

[] Guo, Xuan & Bates, Daniel & Mullins, Robert & Bradbury, Alex. Muntjac multicore RV64 processor: introduction and microarchitectural guide. Technical Report UCAM-CL-TR-972, University of Cambridge, Computer

Laboratory, June 2022. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-972.pdf.

[] Ltd., ARM. big.little processing. https://web.archive.org/web/20121022055646/http://www.arm.com/products/processors/technologies/bigLITTLEprocessing.php, 2011.

[] Priya, K Hema. Design and optimization of floating point division and square root using minimal device latency. *IOP Conference Series: Materials Science and Engineering*, 2021.

[] SiFive, . Sifive tilelink specification. https://sifive.cdn.prismic.io/sifive/7bef6f5c-ed3a-4712-866a-1a2e0c6b7b13_tilelink_spec_1.8.1.pdf.

[] Tarassov, Eugene. Xilinx vivado block designs for fpga risc-v soc running debian linux distro. https://github.com/eugene-tarassov/vivado-risc-v, 2020.

[] Valente, Luca & Sinigaglia, Mattia & Tortorella, Yvan & Rossi, Davide & Benini, Luca. A risc-v heterogeneous soc for embedded devices. https://open-src-soc.org/2022-05/media/posters/4th-RISC-V-Meeting-2022-05-03-Luca-Valente-poster-abstract.pdf.

[] van Smoorenburg, Miquel. Minicom. https://salsa.debian.org/minicom-team/minicom, <1999.

[] Waterman, Editors Andrew & RISC-V FoundationKrste Asanovic, December 2019. The risc-v instruction set manual, volume i: User-level isa, document version 20191213. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.