

# Creating a RISC-V heterogeneous SoC

Progress report

**Matthew Knight**

Department of Computer Science

University of Warwick

# 1 Introduction

The aim of this project is to design a heterogeneous system on a chip (SoC) using the RISC-V open standard ISA[18][7]. 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[2] 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[12], 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 its speed over the efficiency core.

## 1.2 Related work

### 1.2.1 A RISC-V Heterogeneous SoC for Embedded Devices[16]

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.

### 1.2.2 Muntjac multicore RV64 processor[9]

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[14] 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.

### 1.3 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[5] 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: Objectives for the project

## 2 Research

### 2.1 Learning Chisel

As the HDL that is used in RocketChip[3], the underlying SoC generator that this project utilises, being able to write and understand Chisel to at least a basic extent is necessary. The Chisel Learning Journey<sup>1</sup> is a step-by-step guide to quickly becoming effective at writing Chisel and producing effective code. This has been incredibly useful in understanding the vivado-risc-v[15] project and RocketChip generator, though has required a large amount of time. I have not yet fully completed the guide, but have achieved a basic proficiency with Scala that has made creating new SoC designs possible, even if no complex topics have been covered.

### 2.2 Existing Project - vivado-risc-v[15]

The documentation for the existing project is minimal. There is adequate to generate existing designs and implement on an FPGA after a few hours of reading and compilation, but little exists after this. This is somewhat expected - the project is open-source, but primarily for usage by the maintainers. A lot of the time spent has been in understanding the existing project, as well as creating a development environment that matches what is required.

#### 2.2.1 RocketChip

Much time has also been spent in understanding the RocketChip generator[3]. As the SoC generator used by this project, understanding it's capabilities is key to effectively using it. The documentation is very sparse and mostly points to that of the sub-repositories which are used extensively in the generator. Code analysis of the RocketChip generator has been the most effective way to learn about the generator, but this has not been entirely successful. The code is often uncommented and assumes a great deal of knowledge about CPU design, as

---

<sup>1</sup><https://github.com/Intensivate/learning-journey/wiki>

well as the RISC-V ISA, making reading it very difficult. Information such as register locations, interrupt handling, etc, is not readily available and has posed issues to progress on the project.

### 2.3 RISC-V ISA and Assembly

There is excellent documentation on the RISC-V ISA, hosted by the RISC-V foundation. There are 2 documents, the unprivileged specification[18] and privileged specification[7]. The unprivileged specification details instructions that can be executed in U-mode, or user mode - when user programs are executing. The privileged specification details instructions that are executed in M-mode, or machine mode and are intended to be called by the OS or bare-metal software. The documents describe all instructions and registers on RISC-V cores, but do not contain examples or context for many of them. In addition to this, there is huge variance between cores - parts of the RISC-V ISA are optional and are implementation specific, describing what behaviour should be seen as opposed to how much of it is done.

There are also few sources for writing RISC-V Assembly code. Assembly is necessary for understanding the SoC initial startup code, as well as being used often in RISC-V bare-metal software. Assembly is also the simplest way to begin writing instructions that can test and verify heterogeneous designs once implemented on the FPGA, as it is highly unlikely existing software will directly work on the new design without changes. The RISC-V specifications provide details on assembly instructions, though without examples or context.

The RISC-V ISA is an open standard - this differs from open source significantly. An open standard is a formal description of the interface and behaviours exhibited by the hardware. Open source allows the design to be freely available, with most licenses also allowing modification and redistribution. This means that designs of RISC-V cores can be proprietary and contain changes that extend the ISA, but must implement the ISA in order to conform to the standard.

## 3 Progress

### 3.1 Sourcing an FPGA

It was quickly identified that a larger FPGA was required in order to achieve the full project aims. The current Nexys A7-100T[6] has 15,850 slices, each of which contain 4 LUTs for a total of 63,400. This is inadequate, as the SoC requires 10,800 LUTs + 27,500 LUTs per "big" RV64 rocketcore, preventing more than 1 "big" RV64 rocketcore being implemented on the FPGA.

The other option for a larger core is the Sonic BOOM RV64 cores. These have a much larger LUT requirement, with the "medium" BOOM[4] RV64 core using 148,500 LUTs, which is clearly not possible to implement on the current FPGA.

Alternative boards are the Nexys Video and Genesys 2. The Nexys Video contains 134,600 LUTs and would allow for multiple "big" rocketcores to be implemented, but would be unlikely to fit any BOOM cores. The Genesys 2 contains 203,800 LUTs and would allow for a BOOM and "big" rocketcore to be implemented at once. Due to monetary reasons, the Nexys Video was chosen as the FPGA to be sourced. The board is currently out of stock from University sources, so no progress has been made towards purchasing it.

### 3.2 SoC Design

An initial SoC design has been created, consisting of a large and small core. This is using the rocketchip[3] RV64 cores, and are generated using a Chisel description of the processor, with the large core being the "big" rocketcore and the small core the "small" rocketcore. The difference between these are changes to the instruction set extensions implemented and the size of the instruction and data caches. This change reduces the core footprint and LUT utilisation on the FPGA significantly, from an estimated 27500 LUTs to only 7600 LUTs. This is a significant reduction, though comes with the cost of reduced functionality in the smaller core.

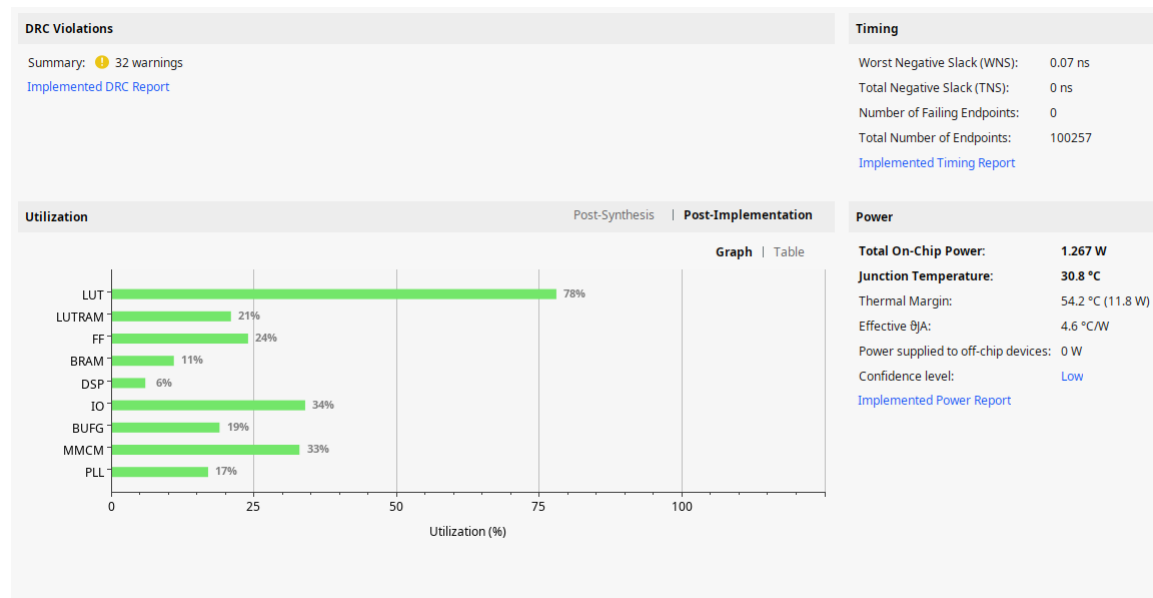


Figure 2: Hardware utilisation of SoC with 1 "big" core

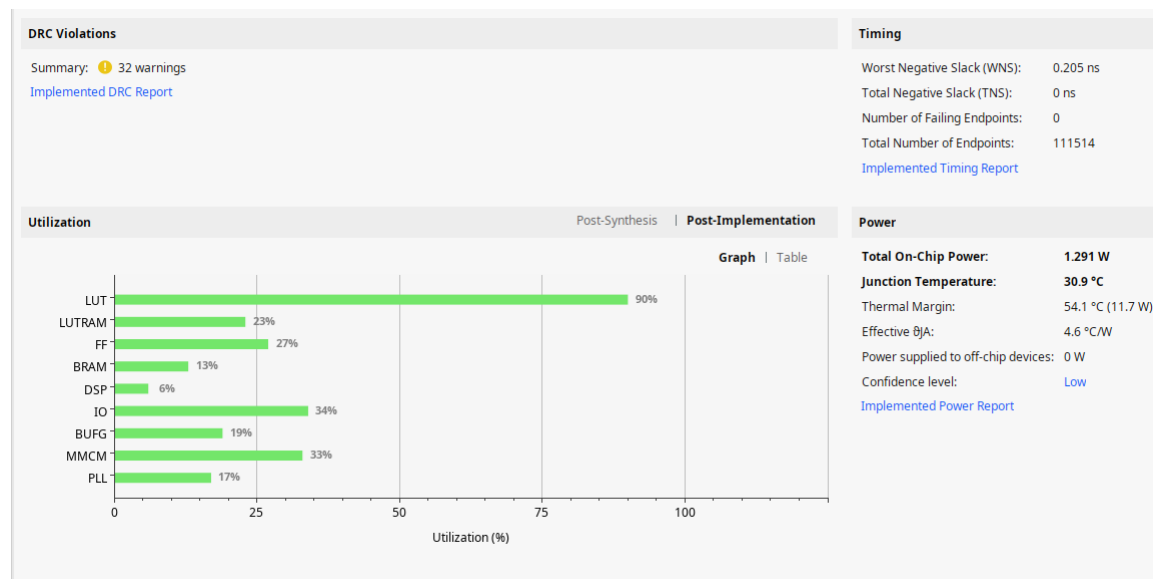


Figure 3: Hardware utilisation of SoC with 1 "big" core and 1 "small" core



### 3.2.1 Core Designs

The "big" core implements the RV64IMAFDZicsrZifenciC[18] instruction set extensions. The IMAFDZicsrZifenci are commonly shortened to G, and are instructions for integer multiplication and division, atomic instructions, single-precision floating-point arithmetic, double-precision floating-point arithmetic, control and status registers and instruction-fetch fence. The final extension, C, is for compressed instructions, allowing the width of an instruction to be just 16 bits instead of the standard 32 for certain instructions.

The "small" core reduces this to RV64IAZicsrZifenciC[18], removing multiplication and both types of floating-point arithmetic. This reduces the capability of the core significantly, and means programs compiled for the "big" core that utilise the extensions unique to the "big" core will not run on the "small" core. This does not follow the original project idea exactly, as the cores were to be equal/very similar in ability to execute instructions, but dissimilar in total processing power due to other factors like pipelining, cache, superscalar, etc. However, this has been unavoidable due to the inability to source a new FPGA in the time frame.

Currently, these cores are the standard designs included in RocketChip[3]. However, there are future plans to modify the cores to be better suited to the tasks to be done on the SoC and the size of the available FPGA.

## 3.3 Software

### 3.3.1 Linux

Debian Linux was successfully run on a design implementing a single "big" RV64 rocketcore, and was used for tasks like internet messaging using IRC[10] and basic text editing. The FPGA was connected to using Minicom[17], a USB serial CLI tool.

IMAGINE THERES A SCREENSHOT OF A LINUX TERMINAL HERE

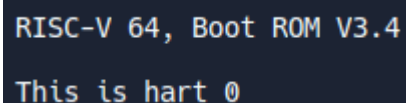
The "big" RV64 rocketcore is the minimum required to run mainstream Linux, as smaller designs forgo the memory-management unit (MMU). When present,

the MMU is responsible for the transfer for data between the registers and memory, as well as ensuring only valid/safe memory addresses are used. When an MMU is not present, there are no checks on the memory that a program is accessing. This is a massive security issue if allowed in the OS, especially in a mainstream OS that could be attacked by malware. As such, mainstream Linux is not compatible with a processor without an MMU[11] and therefore not with any processor containing a core smaller than the RV64 "big" rocketcore. As the Nexys A7 FPGA supports a single "big" rocketcore, Debian Linux[13] was successfully run on an SoC design containing a single "big" rocketcore. This demonstrates that a processor containing cores equal or larger than the "big" rocketcore would be able to run mainstream linux.

The MMU requirement blocks the objective of running the RISC-V Debian Linux distribution on a custom heterogenous SoC, as the larger Nexys Video FPGA has not yet been sourced and will not be for several months. As such, the aims for software to run on the produced SoC have been reduced to bare-metal, with the stretch objective of embedded linux in the future. Embedded linux would not require an MMU[11], but would provide an extremely small amount of features when compared to Debian Linux, and have stringent constraints on the software running.

### 3.3.2 Bare-metal software

Bare-metal code has successfully been written, compiled and run on various SoC designs implemented on the FPGA. A design containing a single "big" rocketcore was implemented on the FPGA, and an SD card inserted into the board containing an ELF executable that was then run by the SoC after boot. The code outputted text over a serial connection that was then read by the host computer, and identified the 'hart' (hardware thread) of the core.

A screenshot of a terminal window with a dark background and light-colored text. The text is displayed in two lines: "RISC-V 64, Boot ROM V3.4" on the first line and "This is hart 0" on the second line.

```
RISC-V 64, Boot ROM V3.4
This is hart 0
```

Figure 4: Serial output of bare-metal program

This was followed by attempting to run on the heterogeneous design previously mentioned. The only serial output received only identified a hart of 0, indicating only a single core was running the ELF executable. Investigation of the default bootrom, assembly code that sets up the core and starts the user executable, revealed that only hart's with an id of 0 started running user code - else they were trapped in a loop of waiting for an inter-processor interrupt to indicate they should execute code in a region of memory stored in a local register. The solution for this is to generate an interrupt and load the register with the location of the ELF, or to adjust the bootrom so that harts always branch to user code. In-line assembly in the C code to generate the interrupt and load the register is currently being worked on.

The current bare-metal software has all been written in C, and then cross-compiled using GCC for RISC-V. Rust is also a candidate language for bare-metal RISC-V programming[1], and comes with certain benefits that improve usability, such as good type checking and memory safety. Changing from using C to Rust is currently being considered, and will be tested.

## 4 Project management

The progress made so far has been mixed - some objectives have been achieved much faster than was expected, while other objectives have taken a significantly longer amount of time than expected. This has been tracked effectively using the Trello<sup>2</sup> board, and allowed for the project supervisor to be updated frequently as to how the project has been progressing. Project meetings have occurred most weeks and have been useful in informing the project supervisor of progress, as well as getting advice for any issues that arose during the week.

The initial design of the SoC took only a single week, far under the initial estimate. This was mostly due to underestimating the capabilities of the RocketChip generator. Verifying and testing the design is ongoing, as difficulty has been had in creating software that will run on the SoC after implementation on the FPGA. There has been a much smaller amount of time available for the

---

<sup>2</sup><https://trello.com/>

project than planned for, and has lead to unsatisfactory progress. To rectify this, the scope has been adjusted to better fit what is possible in the remaining time, and to fit the constraints of what is known to be possible now more is known of the subject area. In addition, much more work will be complete over the December period. Several weeks were allocated to be free or have minimal work complete, but will now be used to achieve progress that should've been made previously.

Due to the issues with obtaining an FPGA that would allow mainstream Linux to be tested, the project objectives have changed so the embedded Linux is the target instead. The objective for running embedded Linux has also been moved to 'could' in figure 1, indicating that there is a medium likelihood this objective will not be completed.

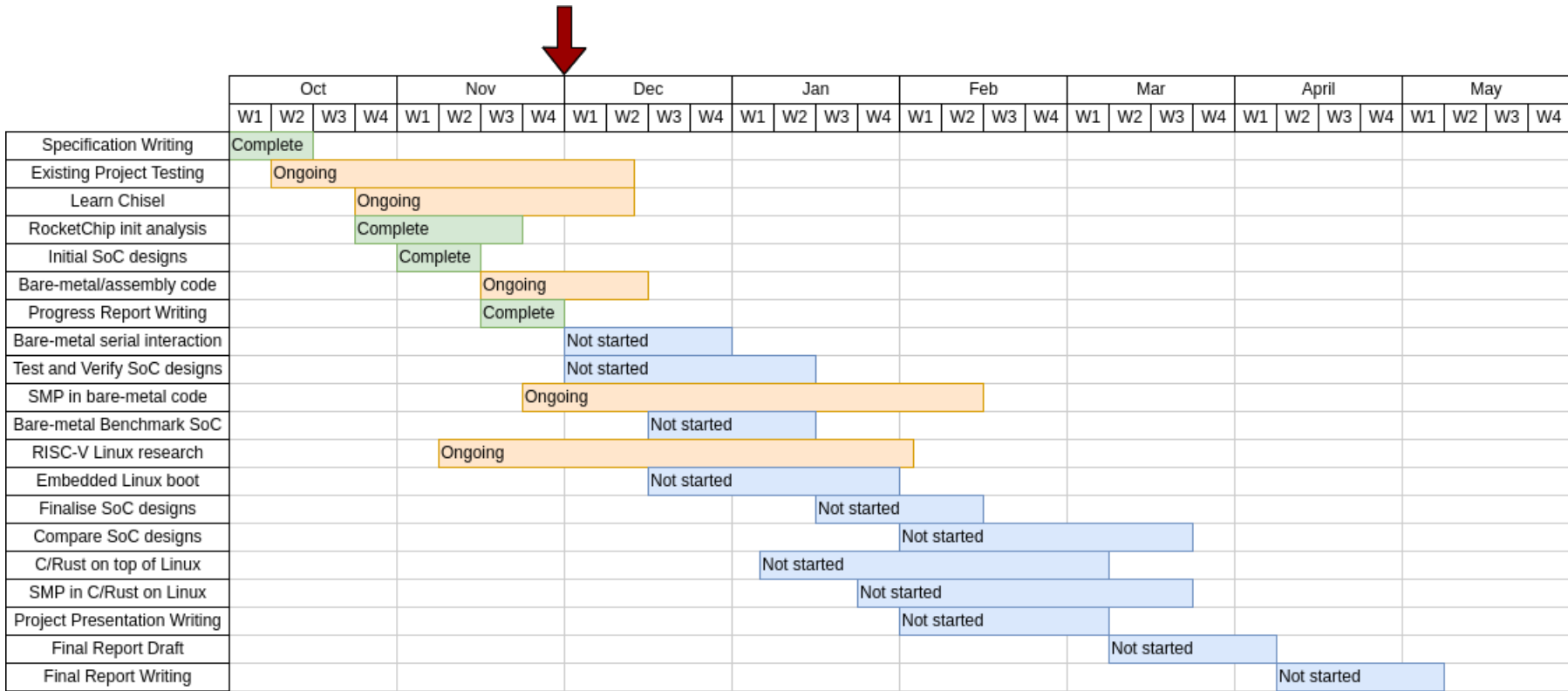


Figure 5: Gantt chart for progress tracking

The Gantt[8] chart in figure 5 shows the progress that has been made so far, as well as expected tasks over the next few weeks. There is a significant amount of time between the presentation and the final report due date, and this has been left mostly empty. This is to accommodate any issues that arise during the project, and should allow time for more to be done between the presentation and report submission if necessary.

## References

- [1] D. H. AleÅ; Katona, Vadim Kaushan. riscv - rust. <https://docs.rs/riscv/0.10.0/riscv/>.
- [2] Apple. Apple unveils m2. <https://www.apple.com/uk/newsroom/2022/06/apple-unveils-m2-with-breakthrough-performance-and-capabilities/>, 2022.
- [3] K. AsanoviÄ, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.
- [4] C. Celio, D. A. Patterson, and K. AsanoviÄ. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical Report UCB/EECS-2015-167, EECS Department, University of California, Berkeley, Jun 2015.
- [5] R. B. Dai Clegg. *Case Method Fast-Track: A RAD Approach*. Addison-Wesley, 1994.
- [6] Digilent. Nexys a7 reference manual. <https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual>.

- [7] K. A. Editors Andrew Waterman and D. . John Hauser, RISC-V International. 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>.
- [8] H. L. Gantt. Work, wages and profit. *Engineering Magazine*, 1910.
- [9] X. Guo, D. Bates, R. Mullins, and A. Bradbury. Muntjac multicore RV64 processor: introduction and microarchitectural guide. Technical Report UCAM-CL-TR-972, University of Cambridge, Computer Laboratory, June 2022.
- [10] D. R. J. Oikarienen. Internet relay chat protocol. <https://datatracker.ietf.org/doc/html/rfc1459#section-4.1>, 1993.
- [11] T. kernel development community. Concepts overview - memory management. <https://www.kernel.org/doc/html/latest/admin-guide/mm/concepts.html>.
- [12] A. Ltd. big.little processing. <https://web.archive.org/web/20121022055646/http://www.arm.com/products/processors/technologies/bigLITTLEprocessing.php>, 2011.
- [13] K. M. Manuel A. Fernandez Montecelo. Risc-v - debian. <https://wiki.debian.org/RISC-V>, 2021.
- [14] SiFive. Sifive tilelink specification. [https://sifive.cdn.prismic.io/sifive/7bef6f5c-ed3a-4712-866a-1a2e0c6b7b13\\_tilelink\\_spec\\_1.8.1.pdf](https://sifive.cdn.prismic.io/sifive/7bef6f5c-ed3a-4712-866a-1a2e0c6b7b13_tilelink_spec_1.8.1.pdf).
- [15] E. Tarassov. Xilinx vivado block designs for fpga risc-v soc running debian linux distro. <https://github.com/eugene-tarassov/vivado-risc-v>, 2020.
- [16] L. Valente, M. Sinigaglia, Y. Tortorella, D. Rossi, and L. Benini. 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>.

- [17] M. van Smoorenburg. Minicom. <https://salsa.debian.org/minicom-team/minicom>, <1999.
- [18] E. A. Waterman and D. . Krste Asanovic, RISC-V Foundation. 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>.