Project Proposal: “Design, Verification and Synthesis of a RISC-V RV32IC Processor”

Matt Young
s4697249
m.young2@uqconnect.edu.au

October 2023

Contents

1 Executive summary

1. For my BCompSc Honours thesis, I want to design, verify, and implement on an FPGA, a simple 32-bit RISC-V CPU core. The CPU will be entirely designed and built by myself, from scratch.
2. The design will be multi-cycle and non-pipelined to ensure that I can complete it in only a year. However, it is designed flexibly to allow for future improvements.
3. Design, verification and logic synthesis will all be done using free and open-source tools like Yosys, Verilator and Nextpnr.
4. I also already own the Lattice ECP5-85K FPGA that will be used, so I don't need any funding or budget for this project - other than possibly access to electronics test equipment like a logic analyser.
5. I believe I can achieve the above goals as I have experience with SystemVerilog and FPGAs already, I own all the tools required, and have been researching on this topic since about 2021.
6. There are some extension tasks such as adding some speculation, a CLZ instruction and a pipeline that I can attempt if time permits.
7. Your involvement as a supervisor would be much appreciated!

2 Introduction

This document contains the initial proposal for my thesis project, "Design, Verification and Synthesis of a RISC-V RV32IC Processor". Please note that this is only meant to explain the aim and goals of the project, not to be an official UQ thesis proposal.

2.1 Background

RISC-V is a popular, open-source, royalty-free, extensible instruction set architecture (ISA). It was originally designed by the University of California, Berkeley in 2015, but has since been managed by RISC-V International. Unlike ISAs such as ARM or x86, RISC-V is a fully open standard, meaning it can be implemented by a lowly computer science undergraduate without the risk of being sued.

Furthermore, RISC-V enjoys support from popular compiler toolchains, software vendors and the wider semiconductor industry. Many large companies such as NVIDIA, Western Digital, Seagate, Alibaba and Intel have taped out RISC-V processors that are currently deployed in real products. There are also numerous RISC-V startups like SiFive, Tenstorrent, Esperanto and more who design new and novel processors.

Existing RISC-V processors can happily boot Linux, compile code with Clang and GCC, and even run complex applications like Firefox. Overall, RISC-V's wide industry adoption and open specification makes it a very attractive ISA to design a research processor around.

2.2 This project

The goal of my thesis is to design, verify and synthesise a simple 32-bit RISC-V CPU core (specifically one which implements the RV32IC ISA - which will be covered shortly). The implementation platform will be the Lattice ECP5-85K FPGA, which I already own via the OrangeCrab devkit. [1] The implementation language will be SystemVerilog, as detailed further below.

This project has the codename "Jelly", in reference to the jellyfish, which is apparently one of the simplest and most efficient animals, which are similar goals to this CPU.

3 Microarchitecture design

Building a CPU from scratch is a complex task, especially for a first-timer, and one year is not an especially long time. To ensure that I can complete the project to a very high standard, Jelly's microarchitecture is relatively simple as far as industry RISC-V processors go. However, although it may be slow, it will still be fully functional,

[1] An alternative implementation platform is the GateMate A1 FPGA and its evaluation board, which may be used if implementation difficulties arise with clocking the Lattice. This board is supported by the same open source tool flow (Yosys).

and designed in such a way as to allow for flexibility and future upgrades (such as the extension tasks detailed later in this paper).

The processor will be designed with reference to “Digital Design and Computer Architecture, RISC-V Edition” by Harris & Harris [1]. This is essentially the seminal book on introductory RISC-V microarchitecture design, and will have many good references for my own design. However, my design will **not** be directly lifted from this book: I am only using it as a reference, and my design will diverge in significant ways from the textbook implementation. Additionally, the textbook only implements a very small subset of RISC-V instructions.

3.1 ISA

The exact architecture that Jelly aims to implement is `RV32IC_Zicsr_Zicntr`. This implements the base RISC-V integer instruction set, RV32I, the “C” Compressed Instruction extension, control and status registers (CSR) and performance counters. In total, RV32I includes 47 instructions, the “C” extension includes an additional 16 instructions, the Zicsr extensions adds 6. So, in total, I would need to implement and verify 69 instructions.

The reason the “C” extension was added is that it’s only some relatively simple extra logic in the decoder, and can significantly increase code density, leading to faster program execution. The Zicntr extension is necessary for onboard performance metrics (see the “Benchmarking” section), and Zicsr is a dependency of Zicntr so is required for this design.

3.2 Pipeline

Unlike most processors, Jelly is *not* pipelined (although this is an extension task). As stated above, the goal for Jelly is to be as simple as possible, and although a simple 4-stage RISC pipeline is well documented and considered simple, it may increase complexity and verification difficulty due to data hazards.

Instead, Jelly will perform the fetch, decode and execute operations on separate cycles. The completed instruction flow may look something similar to this:

Fetch (cycle 1)

Instruction memory is clocked, bringing the instruction at PC from IMEM to chip registers. Although Jelly’s IMEM is backed by FPGA BRAM, it still takes at least one cycle to read the data in this BRAM.

Decode (cycle 2)

The instruction is decoded by combinatorial logic into its operands. This will probably involve decoding the instruction to ALU, LSU (load store unit) and BU (branch unit) operands. If the instruction failed to decode in this stage, the illegal instruction flag will be set, and Jelly will eventually jump to the fault handler (which exists at a fixed address in RAM).

Execute (cycle 3)

In this stage, the ALU is run. Jelly will most likely also contain a barrel shifter, to make sure that we can execute any shift instruction on one clock cycle. This is mainly done for ease of implementation, but may also have performance benefits as well. If an instruction was not able to complete in one cycle, the execute stage will be re-run until the instruction has completed. If the instruction is a load-store instruction, then the load-store unit is asked to begin reading or writing to RAM. If the instruction specifically was a load instruction, then execute may need to be re-run to read the value from RAM. If the instruction was a branch, then the branch target is also computed by the ALU in this stage.

Writeback (cycle 4)

The results from the ALU are written back into the register file, and PC is set to the correct address based on flags and information set previously (e.g. jumps, branches, traps, etc).

3.3 Memory and privilege

The CPU will only implement the RISC-V Unprivileged ISA, as Jelly effectively shares its processing class with microcontrollers. To that end, there are currently no plans to add an MMU to the Jelly design.

Jelly implements a von Neumann architecture. Both the data memory and instruction memory will be backed by ECP5 block RAM (BRAM), which Lattice calls “sysMEM”. The exact size of this will depend on what sort of design resources the CPU takes, but I'm targeting at least 64 KiB of RAM and approximately 128 KiB of program memory. Theoretically we can target significantly more.

Although Jelly's instruction memory is backed by BRAM during execution, the program is first copied from an external Winbond SPI flash module, to allow for simpler runtime programming without re-flashing the FPGA itself. This copy operation will be implemented as a separate CPU hardware peripheral that runs before boot, being completely invisible and inaccessible to the end-user. All it will do is use quad SPI fast-read mode to read the first 128 KiB from the flash chip using Winbond's protocol, which should take no more than a few milliseconds after the FPGA boot process completes.

The exact memory map of the processor will be determined at a later stage, as it depends on the final details of the microarchitecture and particularly FPGA resource utilisation. Nonetheless, the memory map will most likely be similar to that outlined in Harris & Harris, with the addition of an interrupt vector table.

3.4 Interrupts

I don't have massive plans in terms of the interrupt controller - just something that works enough to get UART in and out, and to handle system traps like an illegal instruction. This is somewhat of a departure from the microcontroller design specification, however, it's worth noting that Jelly is not capable of being an actual MCU as it's missing other peripherals like GPIO. There are currently no plans for any sort of nested interrupt controller at all.

Jelly will most likely implement an interrupt system similar to the AVR. At the beginning of the program, an interrupt vector table will list jump addresses for specific types of interrupts, which the processor will jump to if an ISR or fault occurs.

3.5 Boot process

1. Power is sent to the FPGA, triggering its internal reset circuitry
2. The FPGA loads Jelly's RTL from SPI flash mounted to OrangeCrab board
3. Jelly RTL begins executing
4. Jelly initialises a PLL oscillator and waits for it to stabilise if necessary. Jelly will use this oscillator as its internal clock generator.
5. SPI module copies first 128 KiB from Winbond SPI flash into BRAM
6. CPU state is reset
7. PC is set to the code segment base address and the CPU begins executing instructions from there

3.6 Other microarchitecture details

The remaining details, such as what units the processor has and how it's organised on a lower level, will be decided while the thesis is in progress, as this will likely be an iterative process. Additionally, the microarchitecture described above is not set in stone, and may change during the course of development.

4 Tools and resources

4.1 HDL

Jelly will be implemented using the **SystemVerilog** hardware description language (HDL). This is because I already have familiarity with it, having started around November 2022, and it makes describing complex designs easier than plain Verilog or VHDL.

In addition, I have spent some months working on "Slingshot", a SystemVerilog Language Server Protocol (LSP). [2] This provides editor completion and diagnostics services, making SV a very productive language for me to edit in. Slingshot was specifically developed in anticipation of this thesis.

[2] This tool may possibly be capable of being a thesis in its own right, if that is preferred over the main CPU design project described in this document.

4.2 Simulation, verification and synthesis

For simulation and verification, I will depend exclusively on open source tools. These are perfectly fine for my use-case, and most importantly, free. The specific tools are:

- **Verilator** [2]: An open-source, fast, cycle-accurate SystemVerilog simulator that works by transpiling SV to C++. It will be used extensively for verification. The only downside is it doesn't support verification methodologies like UVM, which is acceptable because I will be verifying the processor using a different method (see below).
- **Yosys** [3]: An open-source EDA synthesis tool. Yosys is capable of synthesising SystemVerilog to a netlist, which will be used for FPGA implementation.
- **nextpnr** [3]: An open-source place and route tool. Nextpnr has support for Lattice ECP5 FPGAs, and will be run after Yosys to place and route the synthesised netlist.
- **Catch2**: An open-source unit testing framework for C++. This will be used to create basic unit tests for the SystemVerilog code after Verilator has translated it to C++.

5 Verification plan

Ensuring that Jelly conforms to the RISC-V specification is arguably the most important task here. Jelly aims to be 100% compliant with the RISC-V RV32 specification, even in obscure edge cases, should they exist. To that end, I plan to verify Jelly in the following way, in order:

5.1 Verilator unit tests

Simple unit tests will be performed for each SystemVerilog module, for example the ALU. These will be written in C++ and use Verilator as the simulation platform. The unit tests themselves will use the Catch2 library. This is a good way to run an initial sanity check of the design, but is not sufficient for verifying an entire CPU. It's also useful during development to make sure that the device is functioning as intended.

5.2 RISC-V compliance test

In this test, the CPU will be run through the RISC-V Architecture Test SIG provided suite of test vectors. This will be a good initial test to prove that the design is functional, and should be very easy to use. These test suites are capable of auto-checking themselves, which will be very useful for detecting issues in the SystemVerilog code.

The RISC-V test suite that will be used is available from [4].

5.3 Random instruction generation

In this process, a tool called FORCE-RISCV [5] will be used to randomly generate combinations of valid RISC-V instructions. Each of these instructions will then be run through the Jelly CPU, which we aim to verify, and the official Spike [6] RISC-V simulator, which we know is valid. We will compare the register state before and after each instruction, and make sure that Jelly is exactly identical to Spike.

This will be run repeatedly for a gruelling 12 hour period, and all inconsistencies against Spike will be treated as bugs and fixed.

Note that, based on some initial research, force-riscv appears to be a bit stubborn and difficult to use, so either may be skipped in favour of LibFuzzer (below) or using a custom, simpler RISC-V random instruction generator (although this may blow out the scope of the project too much).

6 Benchmarking

The main benchmark I intend to perform on Jelly is CoreMark and Dhrystone. For Dhrystone, I'll measure DMIPS, or Dhrystone million instructions per second, which is commonly used as a rough proxy for CPU performance.

Once synthesised for the FPGA, I will measure the total CoreMark score and CoreMark/MHz, and DMIPS, and compare against similar processors. In simulation, I will compare against an "ideal" 100 MHz clock speed, to gauge how the processor might perform on an ideal ASIC tapeout.

Unfortunately, I cannot guarantee a particular clock frequency as a deliverable, because this is extremely challenging to estimate at this early stage in the project. A reasonable goal might be 10-30 MHz, and then I can feel accomplished if (hopefully, when) this goal is met and exceeded.

The overall purpose of benchmarking the CPU is to compare it against existing industry designs, and to understand the processor's strengths and weaknesses. For example, it will probably have a small footprint, but Jelly's in-order, multi-cycle design means it will not perform extremely well compared to modern, speculative CPUs.

7 Possible extension tasks

This section contains improvements that I will add to Jelly if time permits. They are ordered in the order that I would attempt them (simplest first).

Note: These are only suggestions for extension tasks. I may not attempt any of these at all, and I almost certainly won't attempt all of them.

7.1 Implement "speculative pre-fetching"

Being a non-pipelined processor, Jelly fetches instructions on one cycle and executes them on the next. In this task, I'll implement a technique I've called "speculative pre-fetching" to potentially eliminate the fetch *and* decode cycle. In industry, this is usually called "instruction prefetch" or "cache prefetch".

The instruction memory and instruction fetcher will be modified to read up to two instructions simultaneously (note this implies that the BRAM is at least dual-port, which the Lattice ECP5 does indeed support). The first instruction fetched is the actual instruction at $PC, but the new *second* instruction fetched is whichever instruction the CPU believes it will most likely execute next. In the simple case of sequential code, this is just $PC+4. I also plan to modify the instruction decoder to simultaneously decode two instructions at a time. This means that if the pre-fetcher is correct, two entire cycles can be skipped.

The skipping mechanism would run on the Writeback stage of the "pipeline". When the pre-fetcher runs on the Fetch stage, it will leave a note indicating where it expects $PC to point after Execute. On the Writeback stage, if the current position of $PC matches the guess, then the CPU skips straight to Execute (since we already have Fetched and Decoded the next instruction). Note this means we cannot pre-fetch two instructions in a row. Since we jump straight to Execute, there aren't any cycles remaining to speculate about where the $PC will change to. This is somewhat problematic, and we will need to benchmark whether or not this actually makes the CPU noticeably faster.

This approach will easily work in sequential code, but would always be wrong for branches. To fix this, we could implement a simple branch predictor (as is another extension task below). However, the branch prediction data structure may be complicated to implement, as each branch instruction would have to be "tagged" with an expected taken/not taken bit.

7.2 Add a fast "CLZ" instruction

In this task, we'll implement a part of the RISC-V Bit Manipulation instruction set, the Count Leading Zeroes (CLZ) instruction. This specific implementation will execute in a single cycle. CLZ is very useful for various types of algorithms and cryptography, and is a good example of how specialised hardware can be orders of magnitude faster than software.

The implementation of this would have to be single-cycle, and would be most likely backed by either some type of balanced tree or binary search algorithm, as detailed here in [7].

7.3 Add "M" and "D" extensions

In this task, I'll add the RISC-V Multiply and Divide extensions. Of these, the "M" extension is the most important because multiply is usually more commonly used than divide.

It is currently undecided how this will be implemented and what type of performance to be expected from it. The goal is ideally to at least beat the software implementation of multiply. This may involve running the multiplier circuit at a faster clock speed that the main processor.

7.4 Implement a simple RISC pipeline

In this project, I would significantly overhaul Jelly from being a naive processor into a fully pipelined one, which opens the door for many further optimisations.

I plan to use the classic RISC pipeline, as it is the easiest to implement and the most well documented. The stages for this pipeline are: fetch, decode, execute, write-back. We could also look into implementing a shorter or longer pipeline.

This will involve a large amount of work, because we will need to re-verify the entire processor to make sure that it handles hazards correctly. I plan to handle hazards simply by stalling, as this is an in-order machine.

As fun as this is, I would only really attempt this if I complete and verify the Jelly design significantly *ahead* of schedule (e.g. 6 months ahead of schedule), as it would be a lot of work, and I would have to abandon it if the deadline approached.

7.5 Add a simple branch predictor

With the pipeline created, conditional branches may cause problems with stalling. Most modern processors use a *branch predictor*, a circuit that attempts to guess which way a branch will go.

The algorithm I plan to use is a 2-bit saturating counter (as documented on Wikipedia). This is one of the most simple branch predictors, but should actually show a performance increase.

7.6 SkyWater 130 nm VLSI tapeout using OpenLane

This is the most challenging subproject, as it is almost another paper in and of itself. It is very, very unlikely that this will actually be done. However, in this project, I'd tape out Jelly into an ASIC using open source tools. I would target SkyWater Technologies' 130 nm process node, using Google's free PDK. The synthesis tool would be a mix of Yosys for synthesis and OpenLane for place and route. This would be extremely complex to verify, as while open source tools *can* run a full tapeout, they don't have good power and thermal analysis tools. In addition, we would need to instantiate PDK-specific SRAM cells for the data memory of the CPU. Having this actually manufactured would be too expensive (and be somewhat of a waste for a processor as simple as Jelly), but the experience gained taping out the processor might be beneficial.

8 References

[1] Sarah L. Harris and David Harris. "7 - Microarchitecture". In: *Digital Design and Computer Architecture*. Ed. by Sarah L. Harris and David Harris. Morgan Kaufmann, 2022, pp. 392–497. ISBN: 978-0-12-820064-3. DOI: https://doi.org/10.1016/B978-0-12-820064-3.00007-6. URL: https://www.sciencedirect.com/science/article/pii/B9780128200643000076.

[2] Wilson Snyder. *Verilator, open-source SystemVerilog simulator and lint system*. https://www.veripool.org/verilator/. 2023.

[3] David Shah et al. *Yosys+nextpnr: an Open Source Framework from Verilog to Bitstream for Commercial FPGAs*. 2019. arXiv: 1903.10407 [cs.DC].

[4] Neela Gala and Marc Karasek. *RISC-V Architecture Test SIG*. https://github.com/riscv-non-isa/riscv-arch-test. 2023.

[5] Futurewei Technologies Inc. *FORCE-RISCV*. https://github.com/openhwgroup/force-riscv. 2022.

[6] RISC-V International. *Spike, a RISC-V ISA Simulator*. https://github.com/riscv-software-src/riscv-isa-sim. 2023.

[7] TEMLIB. *Verilog - Synthesize High Speed Leading Zero Count*. Electrical Engineering Stack Exchange. URL: https://electronics.stackexchange.com/q/196992.