The Educational RISC-V Microprocessor Wildcat

Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark
Kgs. Lyngby, Denmark
Email: masca@dtu.dk

***Abstract*—Today, in teaching computer architecture, we use the RISC-V instruction set to explain the basics of a microprocessor. Initially, we introduce a conceptual single-cycle implementation of a RISC-V microprocessor, followed by a detailed presentation of a pipelined implementation of RISC-V. We can support this topic in teaching with executable implementations of RISC-V.**

This paper presents two implementations of the RISC-V instruction set architecture: (1) an instruction set simulator to explore the RISC-V instructions set and (2) a pipelined implementation of RISC-V. We name this processor Wildcat, after a nice hiking area close to where RISC-V was developed, at the University of California, Berkeley. Wildcat is available under the BSD 2-Clause License at https://github.com/schoeberl/wildcat

***Index Terms*—Computer architecture education, RISC-V, open-source**

I. Introduction

Andrew Waterman defined the RISC-V instruction set architecture (ISA) in his PhD thesis [13] at the University of California, Berkeley (UCB), supervised by Krste Asanovic and Dave Patterson. Waterman distilled the core principles of a RISC ISA drawn from three decades of RISC architectures, including MIPS, SPARC, and Alpha. An important aspect of the RISC-V ISA is that it is available in open source. Therefore, most computer architecture teaching has switched to RISC-V in the last years. RISC-V is an ISA definition; it does not define an implementation. The "V" stands for the fifth RISC project at UCB and also indicates that vector instructions are a part of the standard.

This paper presents Wildcat, a family of RISC-V processor implementations. We present an instruction set simulator to enable students to familiarize themself with the RISC-V instruction set architecture (ISA), concrete the quite small set of integer instructions (RV32I).

To teach the pipelining of a microprocessor, we present a 3-stage RISC-V pipeline consisting of: instruction fetch (IF), instruction decode (ID), and execute (EX). ID includes register read and EX includes memory access (load and store). Although many textbooks present a 5-stages pipeline, we think that a 3-stages pipeline is actually a better and easier approach to teach pipelining: (1) the principles can be shown in three stages, (2) the design requires forwarding, but only form one stage, (3) it avoids the load-use hazard and the need for detection that hazard and the resulting stalling. While single cycle and 2-stages cannot be practically implemented, a 3-stage pipeline can be implemented in an FPGA (or ASIC).

The Wildcat project aims to provide educational examples of a RISC-V simulator and a pipeline implementation. Therefore, the code prioritizes readability and avoids performance or size optimizations that could introduce unnecessary complexity.

For this design, we use Chisel [2], [10], a hardware construction language embedded in Scala. Chisel is a more modern language than VHDL or Verilog, resulting in less distraction from archaic syntax and leading to better readable code. From our experiences in teaching digital electronics in VHDL and later in Chisel, we see that students, who have a programming background in Java, getting quicker productive in Chisel than in VHDL. Furthermore, editor support for Chisel (as it is practically Scala), is better than for VHDL or Verilog.

II. A Simulator

Before designing any non-trivial hardware, and a pipelined processor is non-trivial, it makes sense to write a simulator for that hardware. The simulation does not need to be cycle accurate. In our course on computer architecture students need to write a RISC-V ISA simulator as their final project. Writing a simulator is a good preparation for an elective course of implementing a RISC-V processor in an FPGA.

When I attended Andrew Waterman's tPhD defense, I decided to learn more about RISC-V and to code an instruction set simulator. As I already have switched all my hardware activities to Chisel at that time, I wrote the simulator in Scala. This has two benefits: (1) we can co-simulate the hardware design and the ISA simulator in the same Java virtual machine; and (2) we can share constants between the simulator and the hardware. The ISA simulator is around 300 lines of readable code.

Another option towards a working processor pipeline is a single cycle implementation. As textbooks often start with a single-cycle implementation, we also implemented a base version of RISC-V in Chisel as a single cycle circuit. The instruction memory is implemented as ROM, loading the program at simulation or hardware generation time. For the data memory we use asynchronous RAM (a `Mem` in Chisel). Due to the use of asynchronous memory this design is not useful to implement in hardware, but it can serve as ISA simulator, closer to a real implementation. As we are aiming for reusable code, we have put common functionality, such es decoding, immediate generation, and the ALU code, into functions (in the original 3-stages pipeline). Due to this reuse, the single-cycle implementation is just around 65 lines of code.

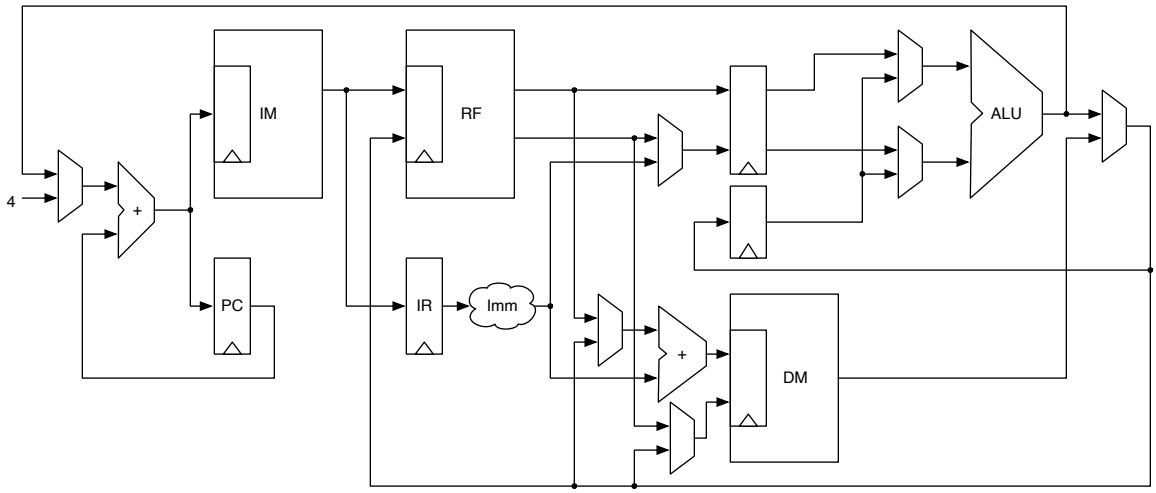

Fig. 1. The 3-stage Wildcat processor pipeline (simplified, omitting control and decoded signals).

III. A Simple Pipeline

Figure 1 shows the simple 3-stages pipeline of Wildcat. Instruction memory (IM), the register file (RF), and the data memory (DM) are implemented in synchronous RAM. Therefore, the input registers of those three memories are part of the pipeline registers.

The first stage (IF) fetches the instruction from the instruction memory. the instruction is stored in the instruction register (IR). However, the two 5-bit register addresses are fed directly from the output of the instruction memory into the (registered) input of the RF.

The second stage (ID) decodes the instruction, generates the immediate field, and selects between register value and the immediate value. Furthermore, that stage computes the address for the memory access, by adding the immediate to the address from the register.

The third stage (EX) performs either an ALU operation or a memory operation. The multiplexer at the output of EX selects wether an ALU result or a memory load will be written into the RF. The output of the EX stage is also registered in the ID/EX pipeline register to forward either the ALU result or the load result. Note that we need only one forwarding path for the ALU. Furthermore, as the memory access is in the same stage as the ALU operation, there is no load-use hazard and no need to stall the pipeline, simplifying the understanding of of a pipelined processor implementation. The output of the EX stage is also forwarded to the input of the memory.

The three stages pipeline is simple to explain, simple to implement, and surprising efficient. As longer pipelines need more forwarding paths into the ALU, they also need larger multiplexers in front of the ALU input. At least in FPGAs, those multiplexers and the needed wires introduces a longer critical path then in the 3-stage pipeline, resulting in a slightly lower maximum clock frequency.

We are not arguing against longer pipelines in general. However, for teaching pipelined processor design a 3-stages pipeline is sufficient. This organization can also be implemented in an FPGA. The repository contains also 4- and 5-stages implementations for comparison.

IV. Testing

Testing (sometimes called verification) of hardware designs is at least as important as testing of software projects. Testing of a processor is actually simpler than a hardware design that needs test vectors to drive a simulation. A processor can simple execute test programs.

The RISC-V test suit[1] contains an extensive list of assembler programs. To run those tests a considerable amount of instructions need already be correctly implemented.

However, when starting to develop a simulator or a pipeline of a processor one would like to run tests even at a very early development stage. Therefore, we provide very simple assembler programs for debugging and testing for our computer architecture lab.[2] That repository also contains the expected output of the register file after termination of the simulation for each example program.

A. Self-contained Tests

Self-contained tests are tests that end with a well known outcome, e.g., that one register shall contain a known number. In the standard RISC-V tests, a passed test ends with 1 in register x28, after executing an `ecall` instruction.

Our simulator and the hardware implementation include tests that execute the RISC-V tests for the RV32I specification.

B. Co-simulation

Another option for testing a processor is co-simulation. Again, co-simulation is relative straight forward as the only state that needs to be compared is the content of the register file. Any error in the pipeline or in the memory subsystem will at some point emerge in the register file.

[1] https://github.com/riscv-software-src/riscv-tests
[2] https://github.com/schoeberl/risc-v-lab

TABLE I
WILDCAT RESULTS IN THREE DIFFERENT TECHNOLOGIES

Wildcat	fmax (MHz)	LC	Register	RAM bits
Altera/Intel Cyclon IV	86.2 MHz	1,756	379	2,048
AMD Artix 7	112.3 MHz	1,270	303	0
SkyWater130	81.2 MHz	429 x 432 μmm^2		

We can execute co-simulation and compare results at every instruction or, similar to the simple tests, compare the content of the register file at the end of the test execution. For simplicity we implemented the comparison after exiting the simulator and the hardware simulation. If there are discrepancies between those two simulations, a trace file for the register contents can be generated to find the spot where the results diverged.

V. EVALUATION

Alhough the main focus of Wildcat on being an educational RISC-V implementation, we still want to be able to implement Wildcat in an FPGA or ASIC.

A. Reading Complexity

For the implementations of the different versions of Wildcat, we aimed at sharing as much code as possible, without obfuscating the code. We defined all constants in Scala land, to be used in the simulator and with type conversion in the Chisel hardware implantation.

We are also able to share a considerable amount of Chisel code, implemented in functions, between the versions of Wildcat: from the single-cycle version up to a 5-stages implementation.

B. Synthesis Results

We evaluate the 3-stage pipeline of Wildcat in two FPGA families and for an ASIC using the open-source SkyWater130 PDK (130 nm process node) and the open-source OpenLane 2 design flow. As we do not (yet) have memory compilers available in open-source for the SkyWater130 process, we synthesize the pipeline alone and leaving out the instruction and data memory. Those memories are connected with memory busses. For the evaluation within an FPGA we use a preloaded instructions scratchpad memory and a data scratchpad memory.

For the FPGA results we use synchronous on-chip memory (BRAM) for the RF. In the ASIC design we implement the RF in discrete flip-flops.

The two FPGAs are: (1) the a bit date Cyclone IV FPGA (from former Altera, then Intel, now Altera again) and (2) the Artix 7 (from former Xilinx now AMD). A logic cell (LC) in the Cyclone IV contains a 4-bit lookup table (LUT) and a flip-flop. The Artix 7 is a more recent FPGA using 6-bit LUTs. For each LUT the Artix includes two flip-flops. We constrain both designs to an unreachable 200 MHz to push the synthesis tools to optimize for maximum clock frequency.

For the ASIC flow we use OpenLane2 and let the tool decide on the needed area. We report that area in the table. The ASIC flow with OpenLane will not finish when the timing constraint will not be met. Therefore, we set it to a frequency of 50 MHz.

We report maximum clock frequency for the slowest timing voltage/temperature point. We report the ASIC size from the report after floor planning (in report 12 `floorplan`). For the ASIC we can find the slack after placement and routing in report 51 (`stapostpnr`).

Table I shows the synthesis results with maximum clock frequency and resource usage in LCs, register, and on-chip RAM bits. For the Cyclone IV we see the expected usage of 2048 RAM bits for the register file. One set of 32x32 bits uses 1024 bits and to support two read ports we need two memories. The Artix FPGA can use LUTs for small memories and Vivado decided to not use BRAM resources. Therefore, we do not see any RAM bits used for the Artix.

For ASIC flow we report the size in chip area. To set the area usage of around 0.18 mm2 in context: one can get a 10 mm2 chip on a multi-project waver from eFabless for $ 10.000.

As we assume that the flip-flop based RF needs the most amount of chip area, we explored to synthesis the RF alone, resulting in an area of around 320 x 320 μmm^2. Therefore, for this simple 3-stage pipeline, the RF dominates the resource usage as is consumes about 55 % of the processor area. We tried to use a latch based RF design, but without any further tweaks in the OpenLane2 design flow, yosys stopped with more than 900 errors. To improve this situation we really need an open-source memory compiler. We are aware of the OpenRAM [5] project, but have not been able to generate memory from that project.

If we are able to scale down the area of the RF, we plan to submit Wildcat to the next Tiny Tapeout [12] shuttle. We also plan to use Tiny Tapeout in our new chip design course. Therefore, the experience on getting a RISC-V pipeline into a reasonable number of Tiny Tapeout tiles is a preparation for this course.

Out of curiosity we also synthesized the single cycle implementation of Wildcat with a blinking LED program into the Artix FPGA. Most of the hardware was optimized away and the maximum clock frequency reported was around 11 MHz. Nevertheless, we configured the FPGA and the LEDs have been blinking.

C. Source Access

Wildcat is available in open-source on GitHub: https://github.com/schoeberl/wildcat. All tests can be run with:

```
make test
```

Several other `Makefile` targets are available to generate Verilog code and to synthesis with Vivado, Quartus, or OpenLane.

VI. FUTURE WORK AND TEACHING

I have been teaching computer architecture for more than 15 years following the classic textbooks [8], [7]. When teaching the pipeline of a RISC processor I used, without questioning, the classic 5-stages organization. With my recent work on Wildcat

I will reconsider which organization I will present initially. In future teaching I will first present a 3-stages pipeline to explain the principles. I will use the implementation of Wildcat to illustrate that this is a reasonable and valid organization. Later, we can talk about other organizations, e.g., the classic 5-stages pipeline or longer organizations.

For the computer architecture course at DTU we teach the students RISC-V assembler programming and they have to implement as final project a RISC-V ISA simulator. For this lab we have prepared material, such as small test programs in open-source on GitHub: https://github.com/schoeberl/cae-lab. We have used those simple test programs also with our Wildcat implementation as simulation and as pipeline.

As a followup to the computer architecture course we have established a new course on designing and implementing a RISC-V pipeline in an FPGA. Again, all material is available in open-source on GitHub: https://github.com/schoeberl/risc-v-lab. This repository contains further tests, some from the Ripes project [9] and simplified versions of the RISC-V tests. In this three weeks course students shall implement a RISC-V from scratch in an FPGA. To provide an example we can use Wildcat and the design of a 3-stage pipeline.

I am teaching Digital Electronics 2 for second semester students. There we use Chisel [2] as design language. In that course I use the Chisel book [10].[3] The final chapter describes the design of a simple processor, called Leros [11]. The Leros implementation is a simple state machine with datapath. We plan to add another chapter on the implementation of a pipelined RISC-V processor, and Wildcat will be the example implementation. However, both processors are not the topic of the second semester introduction course, but make good ending chapters in a book on digital design.

All together we are moving our digital design education to use open-source IPs and open-source tools, such as OpenLane. Furthermore, we develop an new chip design course where we have planned to provide all teaching material in open-source at: https://github.com/os-chip-design/chip-design-intro

VII. Related Work

One of the first hardware implementation of RISC-V was called Rocket [1]. Rocket is implemented in Chisel and represents a 5-stage in-order scalar pipeline. For teaching purposes a group of PhD students developed the Sodor family of RISC-V processors. Sodor[4] is available as single-stage, 2-stage, 3-stage, up to a classic 5-stage version. Sodor is written in Chisel. However, some of the code is quite advanced Scala code, which is not an easy read for a beginners of computer architecture and Chisel. Furthermore, the Sodor project is no longer self-contained. It needs the Chipyard SoC generator, which itself, has an elaborate setup. In contrast to Sodor, Wildcat is available in standalone, and we have put effort into producing readable code.

YARVI (Yet Another RISC-V Implementation) [5] by Tommy Thorn was probably the first RISC-V implementation that could be synthesized into an FPGA (originally released 2014). The current version of YARI, called YARVI2, is a 8-stage pipeline with an effort on branch prediction. In future work we will take inspiration in YARVI to extend Wildcat to longer pipelines and add branch prediction. The main challenge in those extensions is to reuse code, without duplication, and without disturbing readability.

PULPino [4] is a 32-bit RISC-V microcontroller system developed at ETH Zurich. It is written in SystemVerilog in a conservative style (e.g., not using structures but individual signals.) We appreciate that project but consider it too complex for education or research.

Ibex [3][6] is a two stage pipeline with an additional clock cycle for memory access. Therefore, similar in the design as our Wildcat project. The pipeline can be extended with a write back stage.

PicoRV32[7] is a RISC-V implementation optimized for a small size and a high clocking frequency but not for execution speed. The implementation is sequential, where instructions take between 3 and 14 clock cycles. The single Verilog file picorv32.v is about 3000 lines of Verilog code, not an easy reading. We may add a sequential version of Wildcat to our family of processors, if this is still an interesting design point (we found one textbook containing a sequential design of a RISC-V microprocessor [6]).

Morten, a former student at DTU, developed Ripes [9], a graphical simulator for different configurations of a RISC-V pipeline. We use Ripes to educate students in computer architecture.

There exist many RISC-V implementations both in open-source and commercial closed source. Exploring all the open-source designs would probably be an educational exercise, resulting in a long survey paper.

VIII. Conclusion

For teaching computer architecture and digital design good readable reference implementations of RISC-V are a helpful tool. We have implemented several versions of RISC-V: an instruction set simulator, a single cycle version, a 3-stage pipeline, 4-, and 5-stage versions. The aim of this project is to provide easy readable code that can be used as examples in teaching.

Acknowledgment

I would like to thank Tommy Thorn for the ongoing, inspiring, and enjoyable discussions of RISC pipeline organizations.

[3]available in open-source at: https://github.com/schoeberl/chisel-book
[4]https://github.com/ucb-bar/riscv-sodor
[5]https://github.com/tommythorn/yarvi
[6]https://github.com/lowRISC/ibex
[7]https://github.com/YosysHQ/picorv32

REFERENCES

[1] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In *The 49th Annual Design Automation Conference (DAC 2012)*, pages 1216–1225, San Francisco, CA, USA, June 2012. ACM.

[3] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8.

[4] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[5] Matthew R. Guthaus, James E. Stine, Samira Ataei, Brian Chen, Bin Wu, and Mehedi Sarwar. Openram: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6.

[6] S. Harris and D. Harris. *Digital Design and Computer Architecture, RISC-V Edition*. Elsevier Science, 2021.

[7] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[8] David A. Patterson and John L. Hennessy. *Computer Organization and Design, RISC-V Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2020.

[9] Morten Borup Petersen. Ripes: A visual computer architecture simulator. In *2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE)*, 2021.

[10] Martin Schoeberl. *Digital Design with Chisel*. Kindle Direct Publishing, 2019. available at https://github.com/schoeberl/chisel-book.

[11] Martin Schoeberl and Morten Borup Petersen. Leros: The return of the accumulator machine. In Martin Schoeberl, Thilo Pionteck, Sascha Uhrig, Jürgen Brehm, and Christian Hochberger, editors, *Architecture of Computing Systems - ARCS 2019 - 32nd International Conference, Proceedings*, pages 115–127. Springer, May 2019.

[12] Matt Venn. Tiny tapeout: A shared silicon tape out platform accessible to everyone. *IEEE Solid-State Circuits Magazine*, 16(2):20–29, 2024.

[13] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016.