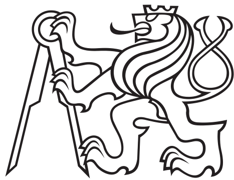
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Measurement

Pipelined RISC-V processor design in VHDL for education and FPGA demonstration

Damir Gruncl

Field of study: Open Informatics
Subfield: Computer Engineering
May 2024

Acknowledgements

I thank my supervisor Ing. Píša for sparking my interest in low-level computing and giving me the opportunity to work on this project, my father Ing. Michal Gruncl for always being ready to provide advice and explanations, and Eduard Lavus for an initial prototype which my work extends.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 3. May 2024

ctuthesis t1606152353

Abstract

RISC-V architecture has gained significant momentum in past years, and our Computer Architecture classes switched to the standard 5-stage pipeline RISC-V model and our QtRvSim simulator in the year 2022. This project creates a matching VHDL design to help smoothly transfer from simulation to real digital circuit design. It allows signal recording during the simulation, connecting peripherals to the CPU core, and running it on FPGA hardware. The goal of the design is not an optimally balanced pipeline but a working illustrative implementation matching the educational model.

We also demonstrate how this simple processor can be used in industrial-grade control of PMSM electrical motor with 40 kHz Park, Clarke forward and inverse transformations.

Keywords:
RISC-V,mzapo,FPGA,QtRvSim,PMSM

Abstrakt

Architektura RISC-V získává v posledních letech na popularit a naše kurzy po íta ové architektury p ešly v roce 2022 na standardní p tistup ový pipeline model RISC-V a náš simulátor QtRvSim. Tento projekt vytvá í odpovídající návrh VHDL, který umož uje hladce p ejít ze simulace do návrhu reálného digitálního obvodu. Umož uje nahrávání signál b hem simulace, p ipojení periferií k jádru procesoru a jeho spušt ní na FPGA. Cílem návrhu není optimální pipeline, ale funk ní názorná implementace odpovídající výukovému modelu.

Ukážeme také, jak lze tento jednoduchý procesor použít v pr myslovém ízení PMSM elektromotoru s dop ednou a inverzní Parkovou a Clarkovou transformací na frekvenci 40 kHz.

Klíčová slova:
RISC-V,mzapo,FPGA,QtRvSim,PMSM

Překlad názvu: Z et zený návrh RISC-V procesoru ve VHDL ur ený pro výuku a p edvedení na FPGA

Contents

Glossary 3

1 Introduction 5
1.1 RISC-V . . . 6
1.2 Project goals . . . 6
1.3 Other RISCV cores . . . 7
1.3.1 neorv32 . . . 7
1.3.2 River CPU . . . 7
1.3.3 RPU . . . 8
1.3.4 FemtoRV . . . 8
1.3.5 Evensgn CPU . . . 8

2 Tools used 9
2.1 VHDL . . . 9
2.2 GHDL . . . 9
2.3 MicroZed APO . . . 9
2.4 Xilinx Vivado 2018 . . . 10
2.5 Yosys Open Synthesis Suite . . . 11
2.6 ICE-V Wireless . . . 11

3 Design 13
3.1 Design Requirements . . . 13
3.2 Technical Requirements . . . 13
3.3 QtRvSim . . . 14

4 Core 15
4.1 Modularity . . . 15
4.2 Pipeline . . . 16
4.2.1 Singlecycle . . . 16
4.3 Hazards . . . 17
4.3.1 Forwarding . . . 18
4.3.2 Stall . . . 18
4.3.3 Flush . . . 18
4.3.4 Fresh . . . 18
4.4 Traps . . . 19
4.5 Frame . . . 19
4.5.1 Free Spaces . . . 19
4.6 Unaligned Memory Handling . . . 21
4.7 Debugger . . . 21
4.8 M extension . . . 22
4.8.1 Multiplication . . . 22
4.8.2 Division and Modulo . . . 22

5 Uncore 24
5.1 Memory . . . 24
5.1.1 Debug Memory . . . 26
5.2 AXI . . . 26
5.3 Interrupt Controller . . . 26
5.3.1 IRQ IP signals . . . 28
5.4 Controller . . . 28
5.5 Configuration . . . 28

6 Software support 29
6.1 Vivado Implementation . . . 29
6.2 ICE-V Implementation . . . 30
6.3 Compiling for Rvapo . . . 30
6.4 Test Suite . . . 31
6.5 Debugger . . . 31

7 Motor control 33
7.1 Commutation . . . 33
7.2 Control . . . 34
7.3 Hardware . . . 34
7.4 FPGA Interface - pmsm_3pmdrv 34
7.5 PXMC . . . 35
7.6 Controller Implementation . . . 36
7.7 Commutator Implementation . . . 37
7.8 Coprocessor . . . 38
7.8.1 Tracer . . . 39

8 Conclusion 40
8.1 Vivado Integration . . . 40
8.2 Further Development Options . . . 41
8.2.1 Address Space . . . 42
8.3 Limitations of Technology . . . 42

Bibliography 43

Appendix A – List of attachments 45

Appendix B – Core Documentation 46
8.4 General Records . . . 47
8.5 Main Interconnect . . . 47
8.6 Program Counter . . . 49
8.7 Fetch . . . 50
8.8 Decode . . . 50
8.9 Execute . . . 51
8.10 Memory . . . 52
8.11 Writeback . . . 54
8.12 Combined Memory . . . 55
8.13 Register File . . . 56
8.14 Hazard Unit . . . 56
8.15 Branch Unit . . . 58
8.16 Arithmetic-logical Unit . . . 59
8.16.1 Optimised ALU . . . 59
8.17 M Calculator . . . 60
8.18 Bitmanip . . . 61

Appendix C – motor control hardware schematic 62

Appendix D – mzapo board schematic 65

Figures

2.1 Block schema of mzapo board . . 10
2.2 Block schema of ICE-V Wireless board. 12

3.1 Schematic of processor simulated by QtRvSim. 14

4.1 pipeline tasks 17
4.2 Horizontal lines represent clock edges. 17
4.3 accelerating division of 8 by 3 . . 23

5.1 System 24
5.2 Interrupt handling. Arrows indicate PC progress. 27

6.1 UI of RVapo debugger 32

7.1 Motor control with pxmc library. 36
7.2 Graph of commutation transformations. 38
7.3 Current on motor coils as calculated by coprocessor. X axis shows relative IRC values. Load 25 W. 38

8.1 Processor core diagram 46
8.2 Byte access register. 53
8.3 Motor HW schematic 64
8.4 mz-apo board schematic 66

Tables

ctuthesis t1606152353

MASTER'S THESIS ASSIGNMENT

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

I. Personal and study details

Student's name:	**Gruncl Damir**	Personal ID number:	**491848**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Measurement**		
Study program:	**Open Informatics**		
Specialisation:	**Computer Engineering**		

II. Master's thesis details

Master's thesis title in English:

RISC-V Pipelined CPU VHDL Design for Education, FPGA Demonstrators and Applications

Master's thesis title in Czech:

Z et zený návrh RISC-V procesoru ve VHDL ur ený pro výuku a demostraci v etn jeho aplikací na FPGA

Guidelines:

RISC-V architecture has gained significant momentum in recent years, and our Computer Architecture classes switched to the standard 5-stage pipeline RISC-V model and our QtRvSim simulator in the 2022 year. The matching VHDL design helps a deeper understanding of CPU principles. It allows signal recording during the simulation. The CPU can be combined with simple peripherals and run on FPGA hardware. The goal of the design is not an optimally balanced pipeline but a match to the educational model.

1) Study previous work on RISC-V design utilizing VHDL structures to represent interstage connections
2) Adapt the design for FPGA hardware (as coprocessor on Xilinx Zynq system and possibly on other solutions, Lattice iCE40, nanoXplore, and Polar Fire)
3) Add modules that allow access to peripherals from the core and access SoC system memory as AXI master, which can connect to peripherals on MZ_APO design as well
4) Use a designed coprocessor for control and communication applications
5) Document your achievements

Bibliography / sources:

[1] Patterson, D. A., and J. L.: Computer Organization and Design RISC-V Edition, The Hardware Software Interface 2nd ed. Morgan Kaufman, 2021, ISBN: 9780128203316

[2] Waterman, A., Asanovic, K.: The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, CS Division, EECS Department, University of California, Berkeley, 2020 https://riscv.org/technical/specifications/

[3] Dupák, J.; Píša, P.; Štepanovský, M.; Ko í, K. QtRVSim – RISC-V Simulator for Computer Architectures Classes In: embedded world Conference 2022. Haar: WEKA FACHMEDIEN GmbH, 2022. p. 775-778. ISBN 978-3-645-50194-1.

Name and workplace of master's thesis supervisor:

Ing. Pavel Píša, Ph.D. Department of Control Engineering FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **05.02.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until:
by the end of summer semester 2024/2025

Ing. Pavel Píša, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

© VUT v Praze, Design: VUT v Praze, VIC

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt

Student's signature

© VUT v Praze, Design: VUT v Praze, VIC

Glossary

- **Entity** is an block in VHDL with defined inputs, outputs, state and transfer function. It has a similar role to classes in object oriented languages.
- **IP** A bounded subsystem with defined interface, which can be reused between multiple designs. It is a shortcut for Intellectual Property, referring to its purpose as a finished piece of work that can be sold and easily integrated into any project.

  In Vivado, it refers to block in its graphical designer. It is assumed that an engineer will decompose their task into subtasks, implement each of these using one IP and then interconnect these in the graphical designer in much the same way that chips are interconnected on a printed circuit board.
- **Trap** Trap occurs when normal code execution is halted, either because of error or because a special instruction called for it.
- **clz** Count leading zeros. Mathematical operation or rv32_Zbb instruction depending on context.
- **LUT** Look Up Table, one of the main primitives found in FPGAs and digital logic in general, used for implementing logical functions.
- **BRAM** Block RAM, in this text it shall refer to physical RAM modules on a FPGA (as opposed to memories assembled from other primitives).
- **ISA** Instruction set architecture. Essentially it is a standardised interface that processor exposes to programs, which have to be compiled for it.
- **RISC** Reduced instruction set computer. Describes computers with only "simple" instructions. There is no precise definition; in this work it should be understood to mean load/store architecture.
- **load/store architecture** ISA where memory accessing instructions only read and write data with no further operations.
- **IF (F)** Instruction fetch stage. The one-letter name is used in source code.
- **ID (D)** Instruction decode stage.
- **EX (E)** Execution stage.
- **MEM (M)** Memory access stage.

- **WB (W)** Write back stage - write result to registers.
- **NOP** No OPeration instruction.

Chapter 1
Introduction

There is a tendency in software engineering to aim for high levels of abstraction. Starting with operating systems and continuing with high-level programming languages, virtual machines, frameworks and hardware abstraction libraries, a situation has been reached where in many use cases, there is no need to know what hardware is down under all the layers of virtualisation. Instead, engineers prefer to work with a simple standard interface to speed up development and increase compatibility.

However, there remain cases where such knowledge is useful; the hardware is still there and software can hide its properties but not change them. One such case is high performance computing. Despite increasing memory sizes and benchmark scores, RAM is still slow and cache covers a minuscule fraction of it, conditional jumps (particularly jumps on variable addresses, such as in virtual method tables) can drive performance down by an order of magnitude, but also increase it when predictors fit the data well, and parallel computing adds many new and exciting pitfalls like race conditions and virtual cache thrashing. Compilers can do a lot, but to get maximum out of a computer it is necessary to understand both the hardware and software.

Furthermore, there are niche cases where general-purpose hardware does not fit due to high requirements on timing and data volumes, and specialized hardware like DSP and FPGAs are used.

Finally, someone also has to develop and optimize the hardware, and the low level software interacting with it, and patch the occasional security vulnerabilites caused by overly aggressive optimizations, such as Meltdown and Spectre.

For these reasons, there is still a need for at least some engineers to understand how and why computers work. A problem that arises is that, unlike with software where source code can be naturally accessed, and debuggers and instrumentation tools are readily accessible, hardware is usually a black box. Processors might have some performance counters and other tools to see beyond what the standard ISA interface that it exposes to the programmer, but one still can not see nearly as much as in software. And even if it was possible, modern CPUs are so complex that it would take many months of study to gain enlightement.

Therefore simplified educational designs and simulators have to be used as introduction to computer architectures. In ages long past, CTU has used the MipsIt graphical simulator, but it has grown outdated much like the MIPS architecture itself, and so work on homebrewn simulator has begun, first on MIPS architecture, but later switching to RISC-V. Detailed description and history of development can be found in the thesis of Ing. Dupák[2].

This simulator is already used for teaching and can therefore be considered

complete, even if more features may be added. But the simulator is not an actual processor, rather it is an schematic depicting what a processor would do, and so for advanced study, work has started on a matching HDL design.

Much like the simulator, it has been a gradual progress, starting with mapo, a MIPS-based skeleton of a CPU written by Pavel Píša to accompany his lectures. This initial skeleton of structured signals interconnecting CPU stages has been used as a model by Eduard Lavuš when he started design of new RISC-V CPU as an exercise for the Advanced Computer Architectures course. He wrote blocks from which a CPU matching the educational design could be assembled in either singlecycle or pipelined version. His work is continued in this project.

1.1 RISC-V

RISC-V is an open standard instruction set architecture. Technically, it relies on concepts already tested in established RISC architectures - 32 registers, small base instruction set, load/store design of memory access, register-based ALU instructions, stack-based program flow. But unlike unlike preceding commercially successful CPU architectures, it is provided under open source licenses, while still being suitable for practical use in wide range of applications from embedded systems to high performance computers. Thought it has a small share in the market dominated by x86-64 and ARM, on-silicon RISC-V processors are already in production for years, from small systems-on-chip to multicore performance models capable of running standard GNU/Linux operating system, and some of the largest players in digital computing, like Intel, Google and Nvidia[18] are involved in its development.

The architecture is designed to be flexible, composed of numerous small subsets that can be freely mixed. Those that are relevant to this work are listed.

- **I** Basic instruction set su cient for a working processor. Contains jumps, addition, subtraction, logical operations, memory access, breakpoint and supervisor call.
- **M** Multiplication, division and modulo.
- **Zbb** Bit manipulation. Contains instructions like bit rotate, count leading zeros and logical operations with negate.

The full name of chosen instruction set is then RV prefix, bus width and list of instruction subsets, like so: `RV32IM_Zbb`. Underscores are used a separator for multiletter subset name.

1.2 Project goals

This project aims to create a HDL design of a RISC-V processor that matches QtRvSim[1] educational simulator. This means that it must be composed of same blocks, with same interconnection signals, including names. It favours simplicity and clarity of design over speed. It shall be able to run both in a logic design simulator and on the education and development kits also used by the university, which have an ARM processor capable of running Linux and a FPGA module.

At the same time, the designed CPU should demonstrate more than the bare minimum necessary to create a Turing-complete machine, since writing some CPU

that works at least in simulator is a task simple enough that it is standard part of advanced computer architecture courses at CTU. Therefore, as a secondary goal application in hardware control shall be explored - PMSM motor control was chosen as an example - and such peripherals will be added to the design that it shall be capable of use as a general microcontroller.

1.3 Other RISCV cores

In this section we look at some other open-source RISC-V cores, starting from large systems on chip and ending with simple student creations, evaluating their structure and methods of implementation.

1.3.1 neorv32

Neorv32 is a design written in VHDL meant as out of the box SoC[4] with extensive documentation. Features include on-chip debugging with OpenOCD, various communication buses like SPI and UART, interrupts, memory caching, true random number generator, DMA controller, support for FreeRTOS and more. Many features are optional; inclusion is controlled by generics. Runs at 130-150 MHz with inorder, single-issue execution.

Extensive user support is also provided, with setups for various Intel, Xilinx and Lattice FPGAs and libraries with functions for high level hardware access.

It has a two stage pipeline - fetch and execute, implemented as VHDL processes within a single entity designated as control unit, which has over 2500 lines of code, several dozen processes and about hundred signals. Some blocks are in separate entites, like alu, bus, register file and instruction decompressor, but there is no clear separation of stages. Signals are mostly organised in records, but some remain separate.

Overall, it is a design meant to be used either as-is, or by advanced users. Understanding and modifying its internals is impossible for beginners.

1.3.2 River CPU

A SoC written in VHDL, with an AXI bus to connect memory and optionally other peripherals like Ethernet, GPIO and UART[5]. Has advanced features like floating point unit, branch prediction, queue for memory access and support for Zephyr real-time OS.

Design is traditional five stages, but in a single cycle, no registers between stages. One entity per stage and central interconnect. Major blocks like register file, FPU, control status registers are extracted into entites of their own. Makes an e ort at using records to simplify interconnection, but entity ports still use only plain logic vectors, to which record signals are assigned one-by-one, which rather defeats the purpose.

Though a less ambitious project than neorv321.3.1, it is still massive and complicated, unsuitable for learning.

1.3.3 RPU

Is a simple RISCV CPU written in VHDL as part of a blog post series on designing a RISCV core[6]. Though a simpler design than the previous two, it can run non-trivial software like Zephyr RTOS or DOOM. However, the git repository contains only HDL code and nothing else, so actually running it would take considerable amount of work.

It has components organised by tasks: memory control, control unit, decoder, alu, registers, etc. It has a five-stage pipeline implemented by means of a state automaton in the control unit, which sends signals to manage the processor.

The top level component contains both interconnection of entities and about hundred lines worth of control logic, doing a wild variety of tasks from selecting PC to reporting unaligned memory accesses, which cause exception. No records are used, only logic vectors.

1.3.4 FemtoRV

Minimalistic processor meant for teaching[27], written in Verilog. It is simple indeed, with less than 1000 lines of source code, most of it in one file separated into blocks with explanatory comments. There is also a series of tutorials describing the design process, and implementations on multiple cheap development boards with sample programs are provided. It is certainly suitable for a beginner - in fact, too suitable, since there are only a singlecycle version, but computer architecture courses at CTU include pipelining, so we need a more advanced design.

However, there is one definite point of interest in this core - in its basic form it requires less than 1000 LUTs, so it is a good study on optimalisation. Some of its ideas were adopted by this work8.16.

1.3.5 Evensgn CPU

A university project for a course of computer architecture written in Verilog, supporting the RV32I instruction set. It is included as a representative of the better end of what a student might create. It has the traditional five-stage pipeline, one module per stage and interconnect to tie them together, then an external memory.

It is similar in design to RVapo: a simple processor that follows the educational models and has little of the complexity of SoC-like designs. However, being written in Verilog, it lacks the elegance of VHDL records. Also, pipeline stages are created by declaring outputs of stage entities as reg, so the stage registers are parts of stage entites, which is less modular than RVapo where inclusion of these registers depends on interconnect.

Chapter 2
Tools used

This is a digital logic design project, for which the following tools are necessary: a HDL language, a simulator and a FPGA with a synthesis tool. The choice of these tools is more restricted than it would be with a software project targeting an operating system, since on an FPGA there is no layer of abstraction between hardware and user code. Instead the synthetised design, created by the user, runs directly on the hardware. Thus the decision was largely dictated by what hardware was already in use at the university, with additional preference for free software. Here they will be briefly introduced.

2.1 VHDL

One of the oldest hardware definition languages, created in 1984. Despite newer alternatives like Migen o ering better support for generic code writing and more C-like syntax, VHDL along with similarly aged Verilog remains common; in fact, these two languages are the only ones directly supported by synthesis tools for both Zynq-7000 and ice40 chips used in this project. This is because modern high-level hardware definition languages are converted to older, low level ones which are then given to synthesis tools, which then do not need to be modified. However, most projects and courses at CTU, incluing Eduard Lavuš's code this work is based on, use Verilog or VHDL directly, and so too does this project.

2.2 GHDL

Free software simulator and translator for VHDL. It converts VHDL source code into a native executable, allowing fast repeated simulation. The generated program is capable of accessing the host filesystem, which is convenient for development of processors, as it is possible to easily change memory content without resynthetising. Its VHDL processing capability is also used as a plugin in the Yosys6.2 synthesis suite, where it replaces its default Verilog frontend.

2.3 MicroZed APO

An educational kit[30][29] which was designed by the PiKRON company by Petr Porazil at the university's request for use in various courses related to embedded systems, such as such as Real-time Systems Programming and Computer Archi-

tectures, as well as other project not directly related to teaching like CAN bus implementation[31] and motor control.

The educational kit is based on MicroZed single board computer equipped by AMD XilinX Zynq 7Z010 chip, which has a dual-core Cortex A9 processor and a FPGA with 28k logic cells and 240kB dual-port RAM. The system is equipped with 1 GB of DDR RAM. This board is then attached to educational kit baseboard, which adds simple peripherals like LEDs, small LCD, PMOD connectors, etc.

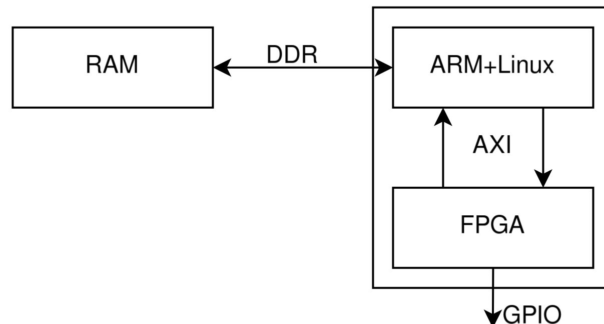


Figure 2.1: Block schema of mzapo board

Typically the board is then used with an OS on the ARM hard cores (or Processing System in Xilinx's terms) – Linux, VxWorks or RTEMS are used for various applications – connected to the FPGA using AXI bus, which is built like a memory interface, that is, every transaction has data and address. Thus FPGA and ARM naturally become memory-mapped peripherals to each other, and hardware interaction such as CAN communication, motor control, etc. can be done on the FPGA quickly and with precise timing, exposing only a high level interface to the OS.

2.4 Xilinx Vivado 2018

Vivado is the vendor-provided IDE for Xilinx FPGAs. The user provides HDL entities, called IP in Vivado – in our case the RVapo processor and several peripherals – which are then interconnected using a graphical tool in a form of visual programming to create a so-called block design. From this design is generated HDL code realizing IP interconnection and configuration of integrated processors and peripherals in the silicon, and this code is finally synthetised. It is very much a closed system prone to mysterious issues and unhelpful error messages, and it is di cult to look inside the process of converting the graphical design to HDL code and find out what went wrong when that happens. This, along with its commercial nature, is the reason its use in this project has been kept to a minimum by ignoring its simulation capability in favor of GHDL.

On the other hand though, an indisputable advantage of Vivado in combination with the MicroZed Linux+FPGA board is how easy it is to communicate with it. With IPs provided by Xilinx, it takes only a few clicks to map registers or RAM block into the memory space of the ARM through AXI bus and then view their content using a Linux program capable of accessing physical memory. This way, one has a reliable and simple way to get large amount of information out of the FPGA or to create a minimal working design to start with.

This is in contrast to boards like ICE-V Wireless or Altera DE2, where I have

frequently run into the problem of design working in simulation but not on the board. The bug was usually a simple mistake in interaction with some hardware that did not occur in simulation due to imperfect emulation of in testbeds, but finding such is really di cult when there is no way to look inside the FPGA without writing it oneself.

2.5 Yosys Open Synthesis Suite

Yosys OSS[15] is a set of free, open-source tools for programming FPGAs. Unlike monolithic, closed tools like Vivado, it is modular, separated into distinct programs:

- Yosys - a RTL synthesis framework with integrated Verilog and option to use other languages through modules, like GHDL for VHDL.
- nextpnr - a general place and route tool that decides how to implement the RTL code in the available hardware, which superseded the earlier arachne-pnr. Also includes graphical viewer of its results. Its main output is .asc text file, which is packed into a binary format of the FPGA by a di erent tool.
- Target FPGA support - multiple projects that integrate support of specific hardware, mostly Lattice. These modify the general nextpnr to make it aware of the target hardware and provide a separate executable to convert its output into the correct binary format.

Getting started with this suite is somewhat harder than with Vivado. Instead of arranging a design in the GUI tool and clicking "Generate bitstream", one has to write a build script to run the HDL files through the correct sequence of terminal–based tools. In the educational context of this work this is not necessarily a bad thing, because it provides insight into what the "Generate bitstream" button actually does.
Also, the omission of GUI design tools – or indeed any sort of IDE – means that the user has full control over what hardware definitions are fed into the synthesis suite, so many errors that I've encountered with Vivado simply can not happen.

2.6 ICE-V Wireless

A single board computer designed for applications in wireless networks[13], cheap and with low power consumption. It has the Espressif ESP32-C3 RISC-V processor and a Lattice iCE40UP5k-SG48 FPGA. This FPGA has the advantage of open-source toolchain support having been reverse-engineered as part of project Icestorm. Unfortunately it is so small that the pipelined version of RVapo did not fit onto it, and the single-cycle one took 4531 out of 5280 LUTs. Since a single-cycle design is impossible to use on hardware due to the usual memory delay of one clock cycle, a rather crude interconnect had to be written which wastes processing time by waiting for memory.

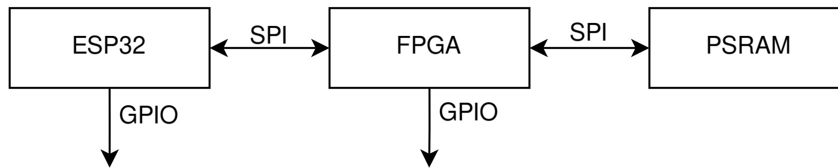


Figure 2.2: Block schema of ICE-V Wireless board.

The platform is also rather di cult to work with; the Icestorm open source toolchain is poorly documented, and there being no ready-made blocks nor an OS like in the Vivado/MicroZed APO system, one has to write their design from ground up. Also the only way to access the FPGA is through the Espressif and a SPI bus, no ready-made memory mapped peripherals. Instead custom communication logic has to be implemented.

Chapter 3

Design

3.1 Design Requirements

The CPU must match QtRvSim as closely as possible, down to same signal names. This being an educational design, it should be made simple and clean, to show how computers work without the obfuscation inevitably caused by optimization. Therefore, it has the standard five stages: fetch, decode, execute, memory, writeback, and a simple memory with no caches or bus. There are also no interrupts.

At the same time, we want to demonstrate not only a basic CPU but also some more advanced features that would make it more practically useful in the same code, so modularity and flexibility are important. This would also make further development easier. Some features that are used in practice and could be implemented in a demonstration design are:

- Debugger - it is enough to inject user-provided instructions and read their result.
- Hardware control - the mzapo kit used at the university provides a variety of peripherals that only need to be connected to the core.
- Memory mapped peripherals - achieved by separating memory from core and inserting a multiplexer.

3.2 Technical Requirements

The CPU must be syntethisable on a FPGA used by the university, meaning the Zynq based education and development kits, and must be compatible with technologies already in use at the university in other projects, which means being written in VHDL and packaged as IP for Vivado.

It must be able to load new programs without resynthetising; in simulator this means simply reading them from file, while on FPGA there is no ready for use method, so it has to be designed and implemented. An AXI bus master shall be provided to communicate with mzapo on-board peripherals, as well as the on-chip ARM. Finally, it must be possible to run compiled programs, not only handwritten ones. This means full support of at least rv32i instruction set, which is the basic unit of RISC-V and conversion of executables into appropriate format.

3.3 QtRvSim

The simulator is not a part of this project, but the CPU design should match its schematic, which is therefore included in Figure3.1.

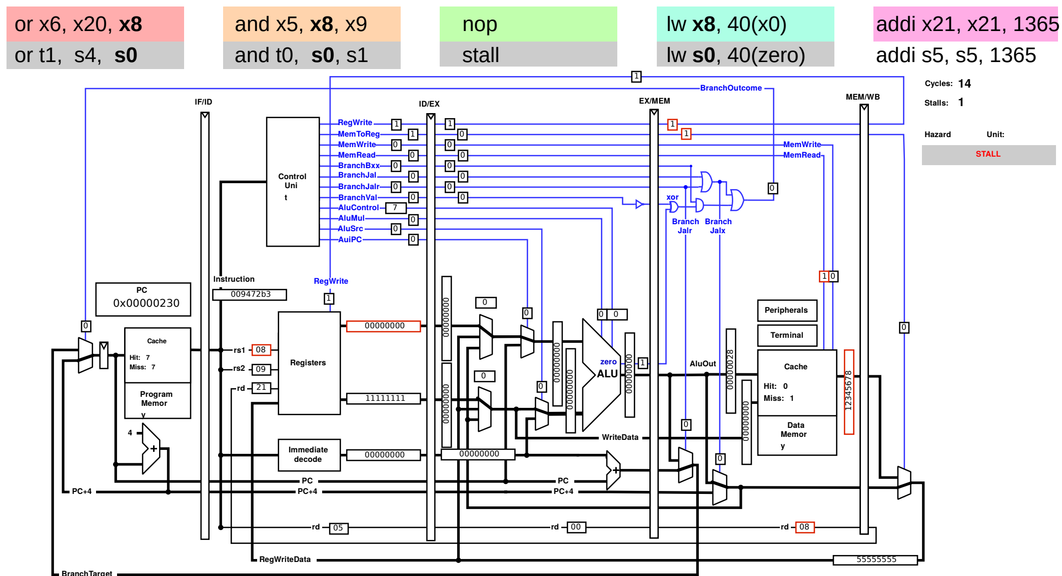


Figure 3.1: Schematic of processor simulated by QtRvSim.

Chapter 4
Core

The core is the part that executes instructions. It is based on a RISC-V CPU written by another student, Eduard Lavus[3] in VHDL. It does nothing more than process a sequence of instructions in order, with no interrupts or speculation. The only deviance from the standard instruction set is that 0x00000000 is considered NOP, to make implementation simpler.

Memory is placed outside the core to support memory mapped peripherals and various hardware memories that may be available on FPGA. Core only has ports for it.

The description of principles of the core design follows. A detailed reference manual with decomposition to HDL entities and description of signals is included later in the chapter8.3.

4.1 Modularity

In order to make the desing modular, each stage of the pipeline has one main entity, which is a black box as far a rest of the design is concerned, and then there is a single top level entity that interconnects them. Therefore changing the implementation of a stage is as easy as linking a di erent implementation of its entity, as demonstrated with unaligned memory handling 4.6. In some stages certain tasks are extracted into a subentities: hazard unit, ALU, branch unit, and register file. This also makes it easier to potentially move them elsewhere.

Signals between entities are always arranged in a single VHDL record. Thus it is not a problem to add even signals that are only used in some configurations – they will not be visible in the entity source code, and optimalization routines of synthesis tools will just delete them. Then the interconnecting entity and these records form a sort of frame of a processor supporting certain capabilities, but the entities that "fill" this frame do not necessarily have to implement all of them. This is demonstrated with the M extension; when it is disabled using the appropriate generic, its wiring remains in the frame, but it is not connected to any actual logic.

So we follow the encapsulation principles of OOP, where the implementation of stages can easily change without a ecting the rest of the CPU, so long as a contract is fulfilled – that is, the entity for that stage implements its part of processing instruction in the required set.

With the task of instruction processing being separated into stages and each stage encapsulated into an entity, it is also possible to reuse them in di erent designs. A simple use case of this reuse is creating pipelined and singlecycle version of the processor by merely switching interconnects; a more ambitious option would be to

assemble a superscalar processor from these components.

4.2 Pipeline

The pipeline has classic five stages of instruction processing: fetch(IF), decode(ID), execute(EX), memory(MEM), writeback(WB). There is also PC, which too can be considered a stage, although it is not part of execution proper.

Describing this design as pipelined is very accurate, because what it really does it take the program counter and pipe it through the stages on a single execution path, where it is gradually transformed into the result which falls out of writeback at the end.

Figure 4.1 shows how are tasks assigned to pipeline stages. Since FPGAs usually implement memories using synchronous hardware macros where read result is available on next clock edge, their internal output registers can act as stage registers of pipeline. This feature is enabled by clk_delayed signals that are part of memory port; otherwise memory with combinatorial circuit-like behavior on read side is assumed.

This is simple with instruction memory, which just needs to read one aligned word, but more complex with data memory, since the RISC-V memory read instructions allows adressing any byte, reading word, halfword or byte, with sign extension or without. Hardly any memory supports all these options, and if memory is asynchronous and its output register is also stage register, then the M stage will never see the memory output and will not be able to edit it accordingly. This could be solved by stalling, but for performace's sake, we instead implemented this in the interconnect.

Branch instructions require a 32-bit addition of PC and immediate in parallel with a comparer for conditional branches, then a multiplexer to choose the final address. In terms of complexity this is comparable with the ALU workload, which also contains addition (in parallel with other arihmetic or logical operations) and then a multiplexer, so it should be possible to move branch unit to EX stage, but QtRvSim has jumps and branches realised in MEM stage and so RVapo must also.

Registers only write on clock edge – hence the register drawn in Reg write.

4.2.1 Singlecycle

A single cycle version of the processor is created by using an interconnect that connects the stages immediately without a register, except between PC and Fetch. Because every instruction is executed in one clock cycle, hazards are ignored.

This naturally does not work with synchronous memory which has output registers, and so the default singlecycle version can not be e ciently synthetised.

For ice40, which did not have enough space for the pipelined version, a special interconnect was written where each instruction actually takes four cycles, as illustrated in Figure 4.2.

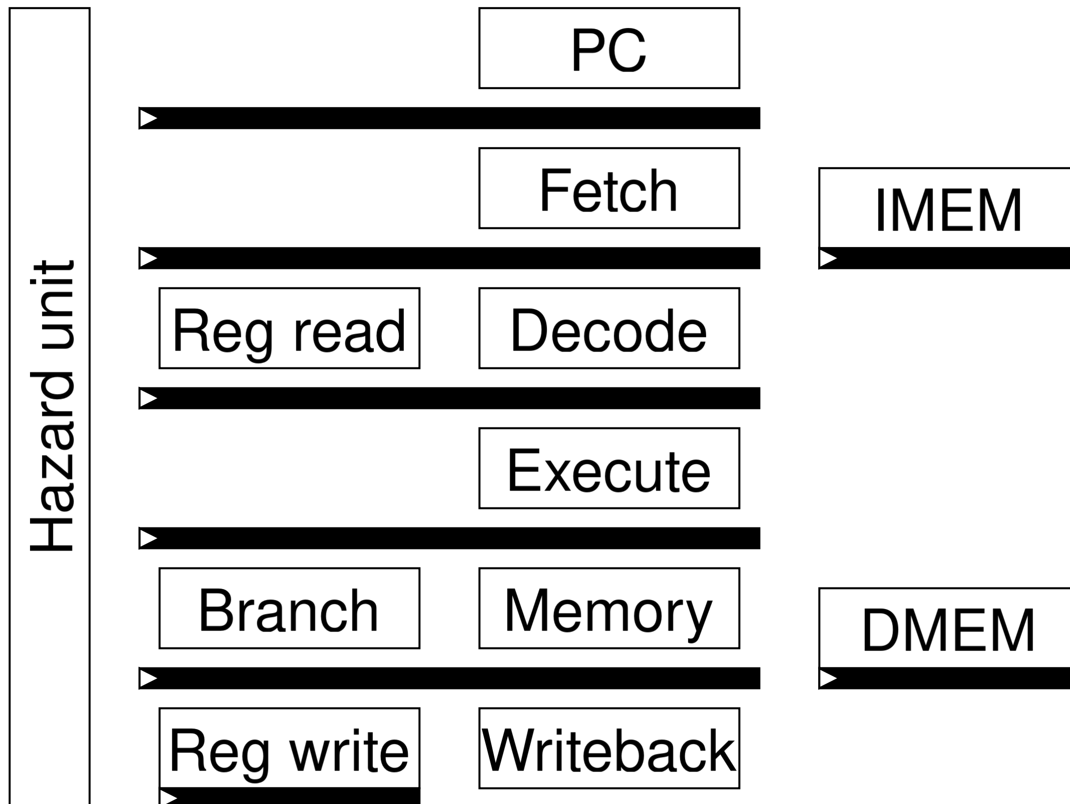


Figure 4.1: pipeline tasks

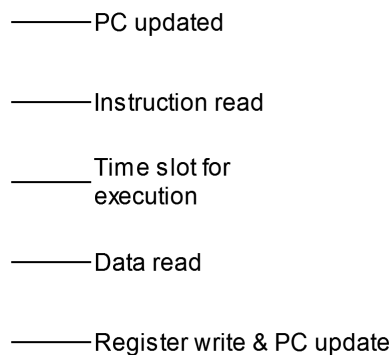


Figure 4.2: Horizontal lines represent clock edges.

Multicycle operations are impossible in all single-cycle implementations, for they require stalling and this in turn requires at least a minimal hazard unit, which is in contrast with the single-cycle processor's purpose as the basic example which will be expanded into something actually useful in the pipelined version.

4.3 Hazards

A hazard is any situation that occurs as a result of issuing new instructions that depend on previous ones which are not yet completed. They can be resolved by careful ordering of instructions or inserting NOPs, but this requires a customised

compiler and makes executables nonportable, so it is rarely used in practice.

The RISC-V standard does not make any restrictions on instruction order, so the processor has to solve these problems internally. The RVapo solution is that each stage sends signals to a special entity called the hazard unit, which then sends out signals indicating how to fix the problem by stalling, forwarding, or flushing. It is up to the stages to give correct signals, there is no means of tracing the progress of instruction through the pipeline, and therefore no method of detecting deadlock or dropped instructions due to an error. There is also no formal commit event. MEM is e ectively the commit stage, if it finishes without exception, then e ect of the instruction can never be undone and data is accessible by all following instructions (sometimes by forward from WB). But when will that happen? Eventually. Some instructions, like division or AXI bus operations take unpredictable cycle count to execute, and there is no way to find out when they are complete – the stall signals only say "whichever instruction is in this stage can now leave it".

4.3.1 Forwarding

There are forwards from MEM and WB to EX, implemented by sending read register addresses to EX and destination register address all the way to WB. This way it easy to check if correct value for one of the source registers is in pipeline.

The register file is also capable of bypassing, that is when a register is written and read at the same time, the written value is read. This is necessary because new values are actually written and become available at the rising clock edge, so when an instruction in ID reads a value that is being written (from WB), without bypassing the old value will be given. When that instruction reaches EXE, the value will already be in the register file and can not be forwarded from later in the pipeline.

4.3.2 Stall

It is impossible to forward when ALU – that is, EXE stage – requires a value that was read by a directly preceding memory instruction, so one stall cycle is inserted by the hazard unit in such cases. Forwarding would be possible when a value read from memory is required by a directly succeeding instruction but only in the MEM stage, not in EXE stage. Forwarding to MEM only makes sense on the data to be written to memory, but no special logic is given for this case, so loading data and immediately saving it will result in an unnecessary stall.

Stages IF,EX and MEM may also initiate a stall of arbitrary length.

4.3.3 Flush

Occurs only on jump, since the core always assumes the "not taken" outcome. ID,EX,MEM is flushed then on the cycle directly following the jump, which is executed in MEM stage – the instruction that was in EXE at moment of jump being executed would be in MEM at the moment of flush taking e ect.

4.3.4 Fresh

When a stage initates a stall, it is always because of a multicycle operation implemented by a state automaton, which holds a stall signal until it reaches final state. The question of reset arises then. With only one stage capable of such long

operations, it is fine to reset automatically on reaching final stage, since end of the stall means pipeline moving and new instructon entering the automaton.

However, when multiple stages start a long operation at the same time, then when one finishes, the other may still be stalling, and then the finished automaton would restart the same operation, raising another stall. This causes the core to deadlock until both automatons happen to reach final state at the same time.

The solution is a 'fresh' signal, which is raised by interconnect for one clock cycle after the pipeline moved, indicating that the internal state of the stage in question should be reset. It is implemented as a simple negation of appropriate stall signals.

4.4 Traps

Traps are always raised in the Decode stage, either by ECALL and EBREAK instructions or when instruction can not be decoded. They must reach the MEM stage before being acknowledged, as previous stages may be flushed on a jump. When a trap signal reaches MEM, it is sent through the hazard unit to interconnect, which halts the pipeline and raises a trapped signal to the outside world. The pipeline can be restarted using debugger4.7. An external device, such as the ARM processor or a custom device5.3, can be used to handle exceptions if the trapped signal was connected to its external interrupt lines.

4.5 Frame

It has been explained4.1 that the RVapo core is designed for high modularity. However, some parts of the design are fixed. Their implementation can be changed, but the contract can not. Those are listed here.

- Five stages of pipeline with the records between them for all supported features.
- Program counter control (autoincrements and branch).
- Harvard memory structure, meaning memory ports in IF and MEM.
- Stalling logic for all stages where multicycle operations are allowed – IF,EX,MEM – such that the stage needs only raise the 'stall' and 'fresh' signals.
- Decoding and executing RV32I set.
- Hazards resolution.
- At most one instruction is issued per clock cycle and executed in order. Though with the input and output of each stage being one VHDL record (and hazard signals), a superscalar design should be possible primarily by changing the interconnect, with the stage entities being reused.

4.5.1 Free Spaces

Among the fixed logic of the 'frame', some places are reserved for extensions, adding additional functionality with little to no modifications to the rest of the design.

Control of Program Flow

The branch result, using information from EXE, is decided in the branch unit entity, whose decision is then fed to PC entity, which selects next PC accordingly. To do this the branch unit requires two pieces of information: signals to decide the branch and target of the potential branch. This is not computed by the entity itself to reuse logic.

Currently, the only decision signal is the zero flag from ALU, which should always be enough since branching is a binary decision, so it is only a matter of signaling ALU to execute the correct operation. In the current instruction set, conditional branches depend on either equality or greater than, which are implemented by subtraction to test equality and SLT for less/greater compare (SLT instruction can be expressed in C as: (a<b)?1:0), so the zero signal fits both cases naturally.

The potential branch targets are more of a problem because ALU can execute only one operation at once. When the branch is not conditional, it can compute the target, but when it is conditional, then ALU only does the comparison and the target has to be computed externally.

Therefore adding new branch instruction is possible, but more exotic ones (read: conditional non-PC relative) would require some more logic to get the target.

The fetching of instructions and actually updating PC to the next value is in RvapoFetch entity. This both avoids a combinatorial loop and allows to intercept the next instruction, which is used ie. to implement debugger4.7 by feeding in instructions from the outside and not using the new PC value given by StageProgramCounter.

Decoding

The decoding of instructions happens entirely inside the fully combinatorial StageDecode entity. Using generics or by swapping the entity, supported ISA can be easily changed.

Input: instruction word and PC

Output: signals to control the other blocks, which may go unused

Execution

All arithmetic-logical instructions are executed in the EXE stage, but the StageExecute entity actually only adds PC and immediate. Actual computations on data happen in the ALU entity, which itself only supports rv32i instructions, and inside it, generics can activate generate blocks to instantiate and connect entities that compute extension instructions.

Addition of PC and immediate is separated from ALU, because conditional jump instruction need comparison and addition, but the ALU is specified to do one operation per cycle on at most two operands. Rewiring it to execute two operations at once in this special case is possible, since comparison (SLT or subtract**??**) and addition use di erent parts of the logic circuit, and it would be more e cient, but it was decided to prioritize clean and simple design.

In case of instructions that can not be executed in one cycle, the EXE stage may raise a stall. For a large instruction set, the (de)multiplexing itself may create too large delays; this would be best solved by creating a 'fast track' for the basic instructions which would be executed as normal, and raising a one cycle stall on any other.

It is also possible to replace ALU entirely; this project o ers an alternative ALU optimized for space with ideas taken from[19].

Input: two operands and control signals. Again, some control signals may be unused.

Output: result of ALU operation and stall requests

Memory Control

Memory interaction is in the RvapoMemory entity. The rest of the processor does not see or care how it is done, and thus support can be easily inserted for various memories. This project actually provides two implementations of this stage. More details in section4.6.

Input: ALU result (doubles as the address for memory instructions), write data, control signals

Output: Read data

Actual Memory

No actual memory device is inside the core, it only has ports to which any memory or communication bus can be connected. A stall signal is used to allow arbitrary delay on memory access. See 5.1.

4.6 Unaligned Memory Handling

Eduard Lavuš's original design assumed a memory that could address individual bytes with no regard for alignment. That's fine in a simulator, even beneficial because it is trivial, but a hardware memory often requires word-aligned addressing.

This is not a major issue since in C standard[8], pointers not aligned to referenced type are undefined behaviour, so in a program that only uses the int data type (or better yet, int32_t), everything will be 4B aligned. Thus, many a program would work anyway, but it is not a correct implementation and can cause mysterious errors. For example, the Rust reference says

> The size of a value is always a multiple of its alignment ... Most primitives are generally aligned to their size, although this is platform-specific behavior. In particular, on x86 u64 and f64 are only aligned to 32 bits.

meaning that a Rust program using only 32bit integers is not guaranteed to be 4B-aligned.

Therefore, a new memory stage that separates unaligned accesses in two was written, described here8.2, although the original one is also available, to be used for simple demonstration or when FPGA real estate is scarce, as is the case on ice40.

4.7 Debugger

To support a debugger, a CPU must allow extracting information about the program state – that is program counter, register, and memory content – without changing that state, but also arbitrarily changing it if so commanded.

Rvapo implements this primarily in the Fetch stage, which can pause the CPU by not incrementing program counter and issuing NOP instead of the currently read

instruction, so that it is not executed repeatedly. The pipeline is not stalled in this state, though, so by injecting instructions from outside the processor state can be changed. Only PC autoincrement is blocked, but jumps still work.

Interconnect then displays result of every instruction on a special output port, so program state can be both changed and examined by injecting the correct instructions. A slight problem is that it as explained in the hazards4.3 section, is not indicated in any way when an instruction is completed. So the correct result of injected instruction will almost always appear within a hundred cycles, but it is not guaranteed.

The interconnect can also internally return the core to running state when trapped, allowing debugger to examine its state.

4.8 M extension

The reason why multiplication and division are not part of the standard instruction set despite being among the most common arithmetic operations is their high complexity. Therefore their implementation requires more attention than the ALU instructions from the RV32I set, for which it is enough to use the correct VHDL operator.

4.8.1 Multiplication

The Zynq-7000 FPGA features pipelined X DSP slices capable of full 18x18-bit multiplication followed by 36 bit accumulation at as much as 762MHz. Assembling a 32x32 bit multiplication from these blocks requires three steps, which should be pipelined and has been implemented as such, but it has been found that at the 100MHz that we run RVapo at, Vivado is capable of synthetising the multiplication operator as a one-cycle operation, which is now used.

The explicit, pipelined multiplicator remains in the git repository only as a curiosity; it is not needed but might be on di erent hardware.

4.8.2 Division and Modulo

These two operations are implemented using the same algorithm, which computes both at once. It is repeated subtraction, but not of the raw divisor. Instead the divisor is multiplied with the largest power of two such that it remains lesser than the dividend, and that number is subtracted. Figure 4.3 shows an example of this method.

The count leading zeroes operation is used to implement this: if the divisor has n more leading zeroes than the dividend and n>1, then

$$divisor * 2^{(n-1)}$$

can be subtracted (and (n-1) added to quotient). Otherwise only the raw divisor is subtracted. When dividend becomes smaller than divisor, then it is remainder and both divison and modulo is calculated. This way count of subtractions is logarithmical rather than linear w.r.t the dividend size.

Normal	Shifted
0b1000	0b1000
-0b0011	-0b0110
0b0101	0b0010
-0b0011	
0b0010	

Figure 4.3: accelerating division of 8 by 3

Chapter 5

Uncore

Uncore are peripherals of the core that are not directly involved in executing instruction. In designs that are intended for purposes beyond educational, these are usually connected to the core by a bus. This design has no bus, just a port for memory. Any other peripherals can be inserted in between. Figure 5.1 displays a system with multiple peripherals set up. Arrow direction indicates master-slave relationship.

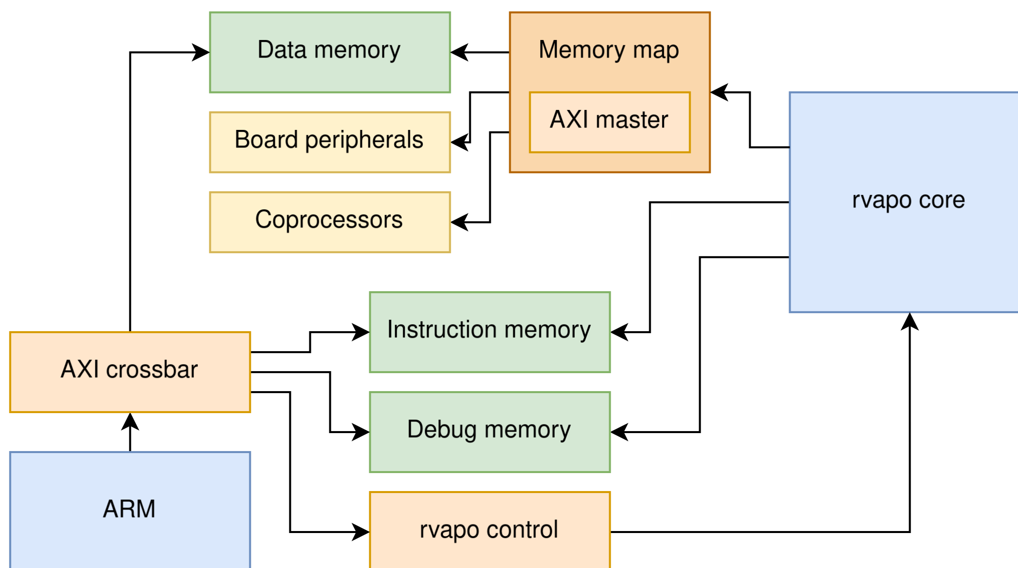


Figure 5.1: System

5.1 Memory

The primary hardware used here are the RAM macros present on FPGAs. Usually, these have the following interface, with one cycle delay on read:

general FPGA RAM interface	
Name	Description
clk	Clock pulse.
address	Address to be read or written.
data in	Data to write.
byte enable	Which bytes to write. All zero means to read.
data out	Data to read.

The core has ports like that, but it adds a signal for stalling and for one clock cycle delay to support various memory devices. Harvard architecture is implied with two completely separate ports for instruction and data memories so that instruction and data can be read in the same cycle easily, but it is possible to connect one memory to both with some arbitration. A third memory can be added for debugging information.

The embedded BRAMs in programmable logic subsystem of the Zynq have two independent ports, which would allow using one actual memory for instruction and data, but we prefer connecting the second port to onboard ARM through AXI and mapping it to its address space. This allows both the RISC-V CPU and the ARM with Linux OS to access the memory directly, allowing easy loading of programs and reading results. All necessary IPs are provided by Xilinx, so it takes minimal e ort to set up.

To respect the read result availability delayed one clock after read issue, the interstage connections allow data read from memory to bypass the interstage registers. Thus one-cycle delay can be handled without stalling.

This is important because stalling on instruction memory would mean at least one stall per instruction unless a cache was added, and that would slow down and complicate the design beyond original scope.

To summarise, three read modes that can be switched at runtime are supported on the core's memory ports:

- asynchronous, where memory completes read operations immediately. Used only in simulation.
- one cycle delay, where operations are completed on the next clock cycle, with data forwarded across stage registers, thus avoiding stalls. Used on the Zynq FPGA.
- arbitrary delay, where the whole CPU is stalled as long as the memory operation takes. Used for memory-mapped peripherals.

For a more advanced design, we want to map some other hardware to memory, so that it can be accessed with regular lw/sw instructions:

- Data memory
- Instruction memory
- AXI master
- Mathematical coprocessors
- Option to add more arbitrary devices.

This is achieved using a memory mapper, a general-purpose IP that can be connected to the core's ports and acts as a rudimentary single-master memory bus – meaning that the same mapper instance can not be used for both instruction and data.

5.1.1 Debug Memory

Simulators should behave exactly like hardware, so debugging on FPGA should not be necessary. However, there were some issues with connecting other IPs during development, and so in addition to data and instruction memory, a third memory port can be added to the core, through which one word of information can be saved per clock cycle. There is an additional port that can be used to select what is saved, which can be accessed from the ARM using memory-mapped registers in the controller5.4.

5.2 AXI

The Advanced eXtensible Interface is a communication protocol designed by ARM for internal communication on chips. Being that the Zynq-7000 chip contains an ARM CPU, the AXI bus is used as the primary means of communication between its components; the ARM chip as well as many Xilinx-provided IPs use it exclusively.

It is a master-slave, point-to-point bus with five channels: read address, read data, write address, write data, and write response. The read response is included in the read data channel. The response channels contain various communication metadata. Each channel contains a ready and valid signal and transaction occurs when these are raised by the receiver and sender respectively at the same clock edge.

This is just a brief overview; in full it is a very complicated protocol with some forty signals, and the documentation[14] has over 300 pages. Fortunately, Vivado can generate general-purpose AXI masters and slaves. An AXI master generated by Vivado has one main entity, which can be split into two parts: one that controls the bus signals, and a state automaton that commands the other part. After minor modification of the automaton, such AXI master has been integrated into this project, specifically into the `MemoryMap` entity.

Thus, the AXI bus is mapped to RVapo memory and accessed with lw/sw instructions, which restricts communication to one word (4B) at a time. Bursts, which allow fast transfers of continuous data, are not implemented, as it would require direct memory access between RVapo RAM and the AXI address space, which in turn would require the MemoryMap to implement a real data bus with its own register-configured controller rather than the glorified multiplexer it is now.

5.3 Interrupt Controller

In design requirements3.1, it has been explicitly stated that interrupts should not be implemented to preserve simplicity, but I have realised that the debugger4.7 can be used by an external IP to implement interrupts with no changes to the RVapo core. This does not conform to the o cial RISC-V specification[20], which requires the use of special control registers to handle interrupts, but it is simple, and that is more important for our purposes.

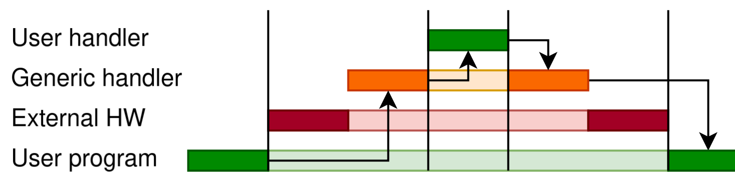


Figure 5.2: Interrupt handling. Arrows indicate PC progress.

We start describing figure 5.2 from the top.

The user interrupt handler should be any ordinary C method that is registered as a handler for a given interrupt number. All registers should therefore be saved to memory before it is called and then restored once it handler exits. To register the method as an interrupt, its address should be written in the interrupt table, which should be placed at a fixed location either by hand or a linker script.

The generic interrupt handler naturally does just that: allocates space on stack, saves all registers, reads handler address from a table, the location of which is hardcoded in it, using an index that must be provided as a parameter in register A0, calls the user handler, then reads registers from memory and deallocates the stack after user handler returns. Two problems arise:

- **Calling** The interrupt number has to be passed to the handler, and return address has to be saved. This requires the use of at least one register before the generic handler has had a chance to save it (for the interrupt number, the return address could be saved with `sw ra,-4(sp); jalr handler`).
- **Return** The return has to be implemented via a jump to register, and the value of the target register has to be restored after the return. The solution is to use ebreak instead and have the HW controller take care of things by injecting instructions from the debugger.

These are solved by the hardware handler, which uses the debugger4.7 to inject instructions that are not in memory and execute them without modifying the program counter. Upon detecting a positive level on an interrupt line, this handler does the following:

- Pause the CPU and wait a few cycles for the pipeline to empty so that no jump can flush the instructions it issues (there is no way of tracing the progress of instructions through pipeline4.3).
- Save A0 and RA below the stack pointer.
- Load A0 with o set to interrupt table (interrupt number*word size) and RA with upper 20bits of generic handler address, which must be provided to it on a port, using something like the controller registers5.4.
- Jump and link to RA, with the lower part of the generic handler address used as immediate.
- Wait for ebreak signal from CPU, which indicates that the generic handler is done.
- Untrap the CPU 4.4.

- Jump to RA, which at this point will contain the program counter at time of CPU being paused at the beginning of this sequence, which was saved to RA in step 5.
- Restore the value of RA which it previously saved below the stack pointer.

Note that the HW handler does not allocate stack for the data it saves, nor does it restore A0. Anything that can be done by the generic handler is left to it.

The generic handler address could be sent to the CPU as a 20-bit immediate in the JAL instruction and it would work in most cases, but it would require adding an input signal to the hardware handler so that it could read PC. As for the "most cases", the problem would arise when trying to read the instructions through AXI bus – physical memory allocated in the ARM RAM is usually at address 0x3xxx xxxx, which is well beyond the 20bit range.

There is no interrupt enable register, but it could easily be added by mapping this peripheral to the RVapo memory.

5.3.1 IRQ IP signals

InterruptController		
Type	Name	Description
in	clk	Clock signal.
in	rst	Reset signal.
out [31:0]	instr	Instruction sent to processor.
out	pause	Pause signal of debugger.
out	issue	Issue signal of debugger.
in [3:0]	irq	Interrupt request vector. Bit 0 has the highest priority.
in [31:0]	irq_address	Address of generic interrupt handler.
in [5:0]	trap	Trap signal for processor.

5.4 Controller

A module that contains sixteen 4B registers mapped to ARM address space through the AXI bus. These registers are then connected to RVapo through plain signals, providing a simple method of communication. Its primary use is to reset the processor after loading a new program and controlling the debugger.

5.5 Configuration

Because the CPU has several variable or optional components (single cycle/pipelined, debug memory port, unaligned access handling), multiple valid combinations can be built. These are configured using four generics:

- simple - 1 to use single-cycle design, 0 for pipelined.
- debug - whether to generate debug memory control.
- unalign - enables splitting of unaligned memory acesses into two aligned ones.
- enable M - whether M (multiplication) ISA extension should be active.

Chapter 6

Software support

This chapter will describe the support infrastructure of the project – that which is required to make use of the HDL code but is not a part of it.

6.1 Vivado Implementation

The Zynq bitstream was built using Vivado 2018.1. As mentioned in its introduction2.4, Vivado is not easy to work with for beginners. The first problem is that unlike with common software development environments, the project is not just a single configuration file such as Makefile or CMakeLists. Therefore it can not simply be uploaded into a version control system. Instead, Vivado o ers an option to write a script that can regenerate the project using the constraints file, block design wrappers, and IP repositories. This process is unreliable, with the script usually containing invalid paths. Project `README.md` describes the correct paths in detail.

Another major problem is the connecting hardware pins of the chip. In theory, it is enough to write a constraint file that will bind hardware pins/ports to named logic vectors and then create corresponding logical pins in the block design, but this was found to be unreliable. Furthermore, it was considered better at CTU to have one general-purpose constraint file mapping pins to logic vectors corresponding to hardware connectors on the board that would be used in all projects, and then do project-specific mapping in HDL. Therefore a custom Vivado project was created where the block design is wrapped in a VHDL file named `top.vhd`, where block design virtual pins are mapped to the logic vectors that the standard constraints file `microzed_apo-rev1.xdc` maps to actual hardware pins.

At least two IPs are necessary for a minimal design: the processor itself in `rvapo/core` folder and the controller in `integration/vivado/IP`. The following optional peripherals are provided:

- Memory mapping unit with AXI master
- Interrupt controller
- Motor controller
- Mathematical coprocessor for fixed point sin, cos, and division.

The Vivado project provided already includes them in its IP repository and contains a block design where all optional IPs are connected correctly, so it should work out of the box. Instructions for use are in readme file.

6.2 ICE-V Implementation

As described in the introduction of the yosys suite, it contains no IDE. To create bitstream, one just writes a script that feeds the HDL files to the appropriate CLI tools.

A major disadvantage of this suite is poor documentation. In particular, it was di cult to find out how exactly the memories work; the tools could not infer the BRAM macros and attempted to create a memory from LUTs, which did not fit onto the chip, so the macros had to be hardcoded.

This then created problems with initializing the memory without resynthetising; the i cepack tool should allow this just by modifying the extant .asc file, but all the documentation I could find, in the form of one StackOverflow answer[22] by an apparent contributor to the project, did not mention the correct format of memory content to be provided, just assumed the use of Verilog $readmemh function initialising an inferred memory. Hardcoded macros, of course, use di erent data format, so making it work means reading source codes and lots of trial and error.

That answer also says that Yosys would have to store information on how is the inferred memory broken up into physical RAM cells, but Yosys itself only contains Verilog support and GHDL is used for VHDL, so it is unclear whether this is even possible. For now, the ice40 integration has to be resynthetised every time memory content changes.

6.3 Compiling for Rvapo

The CPU can execute any program with RV32IM instruction set. Eduard Lavuš originally included build scripts and a manual for compiling Rust programs; these can still be found in old git commits and should work just fine. However, the currently maintained version uses C/C++ with riscv-gcc[11], because it is part of many CTU software-related courses, and most students will already be familiar with it.

The only requirement RVapo has for the executable format is that the first word in memory will be stack position/memory size and the second will be initial program counter.

To achieve this, a custom build system was borrowed from materials for the ACA course. It contains a linker script, which writes the requisite two words, then at a constant address given by the linker script the .text._start, .text, .main, and .data sections are placed in that order. The existence of _start section is guaranteed by the use of a custom crt0, named ctr0local.S, which also initializes the global pointer.

For convenience, a Makefile is provided in the /software/c folder, producing four outputs:

- Human-readable instructions of the executable, created using the GNU binutils objdump.
- rdwrmem compatible format, created from the .elf with GNU binutils objcopy and then hexdump.
- text format readable by the simulated memory, created from the .elf with GNU binutils objcopy and then hexdump.
- VHDL file initializing ice40 memory.

6.4 Test Suite

The pipelined design with debugger, register bypassing and ability to do multicycle operations in IF, EX and MEM o ers plenty of opportunities for obscure errors that only arise on certain sequences of instructions. The FPGA implementation also has to work correctly with hardware. These problems are implementation-dependent, and so a set of custom tests has been written. These primarily target the bugs that most frequently occurred during development: hazards and AXI issues. Hazards and other issues in computation logic can be tested well enough in GHDL simulation using `/scripts/tests.sh`. The simulation testbed only contains a dummy AXI slave; for more thorough testing of the bus, the suite has been adapted to run on the Zynq board using `scripts/fpgatest.sh`, where it accesses the ARM RAM.

- **exe_stalls.c** Tests stalling of the EX stage when it uses data that can not be forwarded trivially, that is from memory or previous long ALU operation.
- **im_stalls.c** Tests situations when instruction read stalls.
- **jump_hazards.c** Test jumps after and before various operations.
- **mem_unalign_stalls.c** Various memory access sequences.
- **mem_unalign_axi.c** Access to memory outside BRAM (AXI assumed).
- **M.c** Tests the RV32M extension.
- **clz.c** Tests the count leading zeroes operation. Originally a subtask of division, but being part of the Zbb extension, it was included as a full instruction too.
- **fnc.c** Tests the mathematical coprocessor.
- **alu_smoke.c** Simple test of addition, shifts, and comparison. Written to test the space-optimized ALU implementation.
- **qsort.c** Sorts a short array using quicksort. Used as a test of stack operations.
- **irq.c** Test interrupts and global variables (that means mainly loading global pointer).

The simulation testbed rvapo_tb_mmap.vhd is also capable of inserting debug pauses at regular intervals or stepping through the whole program in debug mode. This way, it is possible to verify that the debug instruction injector4.7 does not break anything.

The test suite takes less than a minute to run; it is therefore also suitable as a smoke test after introducing new features.

6.5 Debugger

The hardware debugger4.7 on the RVapo processor is controlled by two 32b signals: instruction and its result, as well as three one-bit signals: pause, issue, and step. These can be easily mapped into the memory of the ARM using the rvapo_ctrl IP, but controlling them using rdwrmem or another method of accessing physical memory addresses is impractical.

Therefore a debugging program was developed. It runs in either plain terminal mode or with a text-based GUI shown in Figure 6.1 created by the ncurses[12] library, which is a common part of Unix-based operating systems. It can read and write registers and data memory, view program counter, trap status, and currently fetched instruction. For execution control, it has the standard next instruction, pause, and continue commands. When a trap is detected, attempting to issue a debug instruction will force recovery as explained in 8.5, allowing the user to examine the state of the processor. Trap can be raised using the ebreak instruction, which thus acts just like a breakpoint in common debuggers.

Figure 6.1: UI of RVapo debugger

Chapter 7

Motor control

Several projects aimed at controlling motors have been completed at CTU. In this project, we explore the application of the RVapo CPU for PMSM motor control. Since motor control is not the primary goal of this project, we will only briefly describe the problem and adaptation of existing solutions to the RVapo CPU. More detailed descriptions can be found for example in bachelors thesis of Ing. Prudek[17], which was the main source for the following lines.

7.1 Commutation

Commutation is, in layman's terms, the act of rotating the magnetic field inside a motor to produce motion. Since the opposite poles of a magnetic field attract, the rotor will spin until they are aligned, or forever if some mechanism rotates the fields so that they never meet.

Some motors have internal mechanical commutators that will cause them to spin automatically with a steady power supply. These power one of the coils on the rotor, usually using "brushes", which power whichever of the rotor coils is in contact with them. As the rotor spins, di erent coils inside it are powered, so the magnetic fields of the rotor and stator never align. Brush wear is a common cause of faults.

Motors without mechanical commutators require a controller to update the current on stator coils as the permanent magnet in the rotor spins to maintain motion. This is more complicated but removes the problem of brush wear and allows for precise user control, for the force generated by a coil is controlled by the current and voltage on it, and the sum of these force vectors gives the magnetic field of the stator, which should be either perpendicular to that of the rotor for maximum motion e ciency or aligned to hold position. In our case, the current is controlled using PWM voltage regulation. This naturally produces a discretized signal, which should approximate the ideal current curves as closely as possible to ensure precise and e cient motion. Figure 7.2 shows the PWM duty cycles for a constant-speed spin.

To gain an idea of how fast the commutation algorithm should be, we can take the number of positions recongnised by IRC – 4096 in our case – and multiply it by desired rotations per seconds and the number of motors to control. This will give the update rate required for maximum possible accuracy. Though it is not necessary to update the voltages for every possible position of the motor, it still is clear that refresh rates on the order of 10kHz may be required for some applications. Latency is also an issue. The time between sampling the motor state and applying the control signal should be as close to zero as possible.

7.2 Control

Commutation merely rotates the motor; a higher level controller is necessary to control direction, speed, and optionally momentum. To gain any degree of accuracy, this requires taking feedback from the motor and running a control algorithm such as a PID controller, which takes desired and actual values and produces a control action signal.

7.3 Hardware

The 3p-motor-driver-1 board8.3, designed by PiKRON, is used to connect a three-coil BLDC motor to the mzapo board through a 40-pin parallel connector. The motor itself is powered from a separate DC supply through MOSFETs realized power stage.

It provides access to the following hardware:

- An IRC with two channels, which gives relative position of the motor axis.
- Three digital Hall sensors, which give absolute position of the motor axis with 60° resolution.
- Three amplifiers for the incoming PWM signal.
- Three analog Hall sensors measuring actual current on motor coils. Their output is fed to a 12-bit ADC, from which values can be read using SPI bus.

The ADC interfacing to the processor system is realized in FPGA by a SPI bus state automaton which provides cumulative rolling sums of samples for each individual ADC, not immediate value. This can be used to easily gain averages current over time to filter out anomalous values, but the commutator we use is fast enough to iterate as fast as the ADC can convert values, so it still uses immediate values by subtracting previous value from current.

7.4 FPGA Interface - pmsm_3pmdrv

The motor driver board has an accompanying VHDL entity, pmsm_3pmdrv, which generates PWM from a 14-bit counter, communicates with the ADC on the SPI bus, counts IRC pulses, and provides results as registers in an AXI slave, allowing any device with AXI master to control the motor. In this project this entity is mapped to both ARM and RVapo CPUs, thereby allowing a change of distribution of motor control tasks using software only.

All three PWMs are generated from single counter with autoreaload value 5000 on a 100MHz clock for frequency of 20kHz. ADC is on a SPI bus on 2 MHz clock synchronous to the master clock, and values are read in an overlapping pattern with 15 clock cycles per value read. Therefore all three channels can be read $44444.\overline{4}$ times per second, or $2.\overline{2}$ times in a PWM period.

In order to synchronise output of control (PWM) with the input (ADC), it was decided to read all three channels twice and then wait for PWM reload. A minor implementation issue with this is that the aforementioned 15 clock cycles per value read apply only on continuous reads. Waiting for PWM reload means having six additional cycles to start the overlapping pattern. Single burst of three value then

takes not 45 cycles, but 51. However, with 2 MHz SPI clock and 20 kHz PWM, 100 SPI clock cycles are available per PWM period.

Therefore, to read all three channels twice in the PWM period, a burst of six values is actually read, which takes 96 cycles.

The Hall sensors measuring current are right next to each other on the PCB, resulting in certain crosstalk. Bc. Jakub Janoušek was able to has developed callibration method to decompose and compensate this crosstalk in a frame of his diploma thesis project titled Open-source Motion Control on Mid-range and Small FPGAs. The obtained calibration matrix is used by is used by a coprocessor7.7 to correct data before providing them to motor controller.

The following table describes the provided registers. All are 32-bit.

root: pmsm_3pmdrv1_t		
O set	Name	Description
0x08	irc_pos	Actual IRC value.
0x0C	irc_idx_pos	IRC value at time of mark?TODO
0x10	pwm1	PWM 1 duty, bit 31 is disable, 30 is enable.
0x14	pwm2	PWM 2 duty, bit 31 is disable, 30 is enable.
0x18	pwm3	PWM 3 duty, bit 31 is disable, 30 is enable.
0x20[11:0]	ADC status	ADC conversions count.
0x20[12]	ADC status	1 indicates the second triplet of reads in PWM period.
0x20[18:16]	HAL	Digital HAL sensors output.
0x24	adc1	Value of ADC 1. Cumulative.
0x28	adc2	Value of ADC 2. Cumulative.
0x2C	adc3	Value of ADC 3. Cumulative.

7.5 PXMC

PXMC[26] is a general library for motor control written by PiKRON. It contains a command parser and a PID controller suitable for controlling various motors – steppers, DC, and brushless DC with or without coprocessors. To make use of it, an adapter program has to be written to provide communication between the library and the motor. The way to do this is to initialise the main control structure of the library, pxmc_state, with information about the motor, controller parameters, and callbacks. Then create a thread where these callbacks, listed henceforth, are called. The main file for implementing all this is customarily called appl_pxmc.c.

- **pxms_do_inp** This method should take input from the motor. Position is required, current optional.
- **pxms_do_con** This should run an iteration of the controller. pxmc_pid_con, a part of the PXMC library, is the standard choice.
- **pxms_do_out** Take the output of the controller, which is desired "energy" (current or voltage depending to system setup) on a coil perpendicular to the rotor, recalculate it to format compatible with available hardware, and send it to the motor.
- **pxms_do_ap2hw** Set the new absolute position of the motor. Hardware does not always provide it, so there is a software emulation in the form of o set to get absolute position from relative.

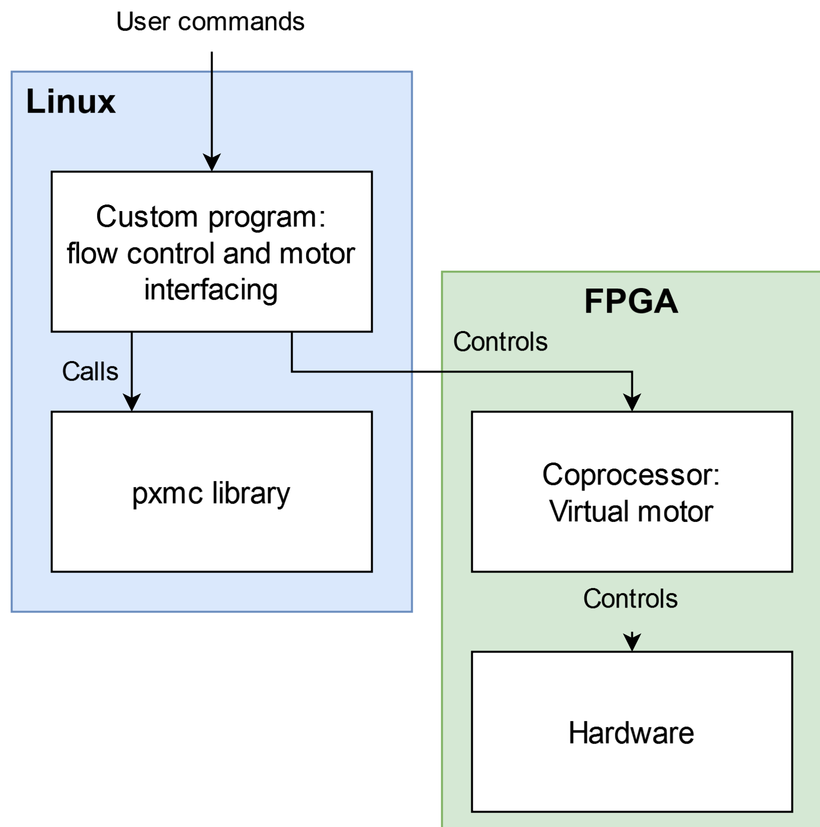


Figure 7.1: Motor control with pxmc library.

7.6 Controller Implementation

The controller is a pxmc-based program, running on top of Linux on the ARM part of the mzapo board. Instead of writing it from scratch, code from two existing projects was reused:

- **pxmc-linux**[24], which is a demonstration program for controlling motors through a command line interface. It runs on the mzapo board using only the ARM processor.
- **lx-rocon**[25], which is a program for motor control in industrial applications. It uses a coprocessor to provide the virtual coil.

This project takes the pxmc-linux program, replaces the code in between pxmc and hardware with that from lx-rocon, and edits the hardware access to fit the RVapo processor and pmsm_3pmdrv peripheral.

Its inputs are absolute position from digital Hall sensors, relative position from IRC, and current through two virtual coils, one perpendicular to the motor axis (denoted as quadrature or Q) and the other in line with it (denoted as direct or D). The output is desired PWM duty cycle on a virtual coil that is orthogonal to the rotor.

The program is meant to run at 1kHz. This is achieved by using clock_nanosleep with TIMER_ABSTIME, so that it sleeps until clock reaches an absolute time, instead of the default sleeping for a given period. The absolute time to sleep until

is then incremented by a millisecond after each iteration of the control loop. To improve timing properties of the operating system, the program is locked in RAM and run with SCHED_FIFO and priority 60, making it a real-time task that will preempt any normal programs running under default scheduling.

7.7 Commutator Implementation

The commutation algorithm provides the aforementioned7.6 virtual coil. It does so using Clark and Park transformations[17] to convert orthogonal coil PWM duty given by control algorithm to PWM on the three actual coils and convert measured currents on these coils to currents on two virtual coils, one orthogonal and one aligned with the rotor.

This algorithm can be a part of the adapter of pxmc library or it can run on a coprocessor. In the lx-rocon project, it runs on the FPGA on tumbl, a custom softcore CPU with a MIPS-like architecture[28] with a memory-mapped peripheral for fast fixed point goniometrical functions and division.

The tumbl CPU was replaced by RVapo as part of demonstrating its capability. The commutating program was copied with only small changes to initialisation. It is capable of controlling multiple motors, referred to as axes, simultaneously.

Since the coprocessor can actually run faster than the analog-digital converter, and therefore repeat computation on the same data, there is a synchronisation block in its program, which waits until the ADC conversions count has changed. This is also important because the ADC is cumulative and the actual current is gained by subtracting previous value from the actual, so not waiting for next measurement would result in false readings of zero current. The program tracks the minimal amount of wait cycles at synchronisation; as long as this number is nonzero, it can be said that the speed of coprocessor is su cient for precise control, since at that point the ADC is the bottleneck.

As described in 7.4, the PWM is generated with frequency 20kHz, and the commutation algorithm is synchronised to it, although without synchronisation it can run at 110kHz, meaning that depending on overhead it might be able to commutate four or five motors, which the program supports. Such speed is owed to highly optimised firmware using fixed point numbers, very few conditional branches or expensive mathematical operations like division.

Faster operation to control more motors at once can be achieved by optimizing the AXI bus, which is used to access the `pmsm_3pmdrv` entity. The default design uses complicated network of AXI interconnects to allow both processors to access all available peripherals and memories, which results in latency of ±50 cycles. Reducing these to a necessary minimum increases commutator iterations to 160 000 per second.

Figure 7.2 shows inputs/outputs of the commutation.

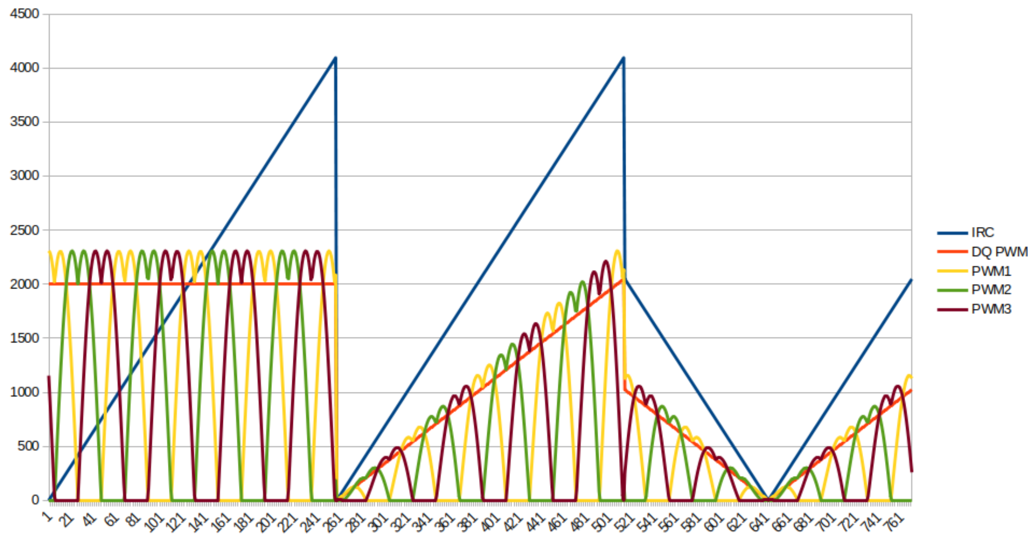


Figure 7.2: Graph of commutation transformations.

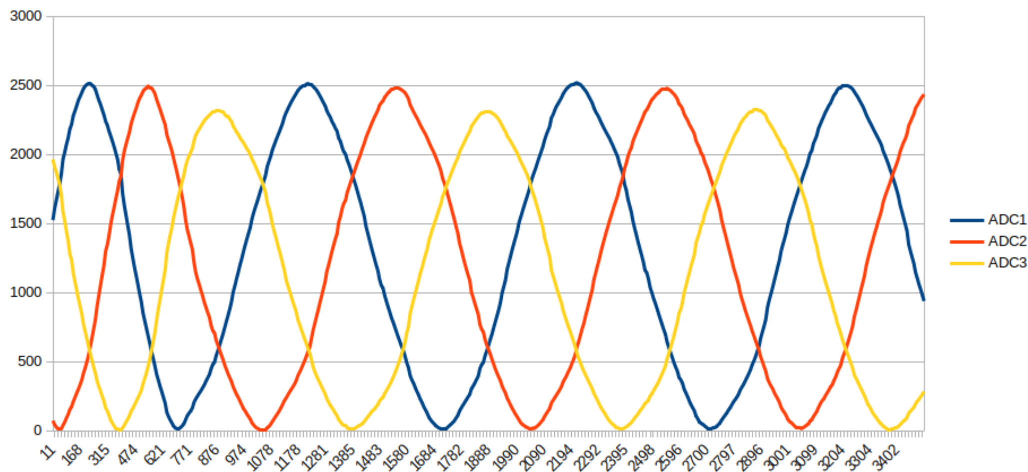


Figure 7.3: Current on motor coils as calculated by coprocessor. X axis shows relative IRC values. Load 25 W.

7.8 Coprocessor

The coprocessor state is in the pxmcc_data_t data structure. This structure contains both internal state, configuration and interface; here only the configuration and interface parts of it will be documented. A custom linker script puts its address in the data memory at address 0x10. Since RVapo memory is mapped to ARM adress space, all that is needed to control the coprocessor is the address of its data memory. All the variables have the size of one word, which is 4B.

root: pxmcc_data_t		
Type	Name	Description
pxmcc_common_data_t	common	structure for data that is common to all axes
pxmcc_axis_data_t	axis	array of axis-specific data
pxmcc_curadc_data_t	curadc	array for ADC data, three elements per axis

axis data: pxmcc_axis_data_t	
Name	**Description**
ccflg	set 0 disables commutation
mode	Type of motor. 0 for BLDC
pwm_dq	Desired PWM duty at virtual coil.
cur_dq	Current at the virtual coil.
ptirc	IRC pulses in phase table.
cur_d_cum	[12:31] rolling cumulative current sum, [0:4] rolling sample counter
cur_q_cum	same as cur_d_cum but in Q
inp_info	Pointer to position source (IRC in our case).
out_info	Index into curadc array for this axis.
pwmtx_info	Pointers to PWM control registers offset by a constant base address.

adc data: pxmcc_curadc_data_t	
Name	**Description**
cur_val	ADC increment between last two iterations.
siroladc_offs	Offset to calibrate zero.
siroladc_last	Last value read from ADC, raw.

common data: pxmcc_common_data_t	
Name	Description
fwversion	Firmware version.
pwm_cycle	Autoreload value of PWM counter.
act_idle	Last amount of wait cycles at ADC sync.
min_idle	Shortest amount of wait cycles at ADC sync.
rx_done_sqn	Number of last processed ADC measurement.

7.8.1 Tracer

To help with development, a simple C program is provided in the /scripts/tracer folder which is capable of recording data from the coprocessor interface as well as memory-mapped FPGA hardware and storing them in a .csv file. It has two main modes of operation: passive recording triggered by sufficiently high PWM values (no point in recording when idle), and injecting mock IRC values to simulate motion of a motor that does not need to exist. Other than that, despite being a program in form, it is meant more as a script, to be edited on the fly without setting any fixed interface. Yes, C is a scripting language when gcc is just a few keystrokes away.

Chapter 8
Conclusion

The result of the work is a working RVapo softcore processor matching the QtRvSim diagram with a simple singlecycle version and a full-featured pipelined version, which share most of their source code. The design is modular and flexible as required, which is demonstrated by its optional features.

The pipelined version has also been implemented on the Zynq-7000 FPGA, where it can be used as a simple but working microcontroller, with memory mapped peripherals, debugger and interrupt controller. It can also be used as a coprocessor for the larger ARM CPU, which has been demonstrated on the motor control example. It also leaves space on the FPGA to add other devices.

Implementation on ICE-V board is also possible, but due to space constraints, only singlecycle version can be used, and still the chip is almost entirely filled. Changing initial embedded memory blocks content also requires resynthetising, which is particularly inconvenient for a processor. Overall using the RVapo core on this board can not be recommended until additional work is invested into mapping adders and other units to the iCE-40 provided DSP blocks, because Yosys could not infer this automatically. Finally, the di culties in accessing the FPGA make it less suitable for beginner's exploration than the Zynq.

8.1 Vivado Integration

A pipelined version of the RVapo CPU takes 3200 out of 17600 LUTs on the Zynq-7000 FPGA, so plenty of space is left on the chip; in fact, the AXI logic takes up more resources than the entire processor. Still, the full system utilizes less than 60% of the FPGA.

Vivado, the synthesis tools o ered by Xilinx2.4, o ers a hiearchical breakdown of resource utilization which could in theory be used to determine which components of the CPU take up most space, but many entities are elided, probably as a result of being merged during optimization, so it is impossible to get any practically useful result without thorough study of the synthetised design, which was not done because speed is not a priority of this project.

As for timing, it runs at 100MHz, even though Vivado claims that timing analysis has failed. It appears that letting Vivado generate unpipelined 32x32 multiplication on 18x18 multiplication macros confuses its timing analyzer, as all the failing paths are around the multiplier (which works perfectly).

Xilinx BRAM IPs were used for memory, one each for data and instructions, at their maximum size of 8kB for a total of 16kB of memory, although compilers will only recongnise it as 8kB, as both memories share address space.

8.2 Further Development Options

Adding too many features could betray the original purpose of being a simple educational design, so one would have to be careful to take advantage of its modular design without modifying the original frame 4.1 much.

This makes the implementation of superscalar execution an idea of questionable merit. Were such a project to be undertaken, it would be better to make it a di erent processor that only uses some RVapo components.

Instead, one thing that could be readily added are cache, which can be a fully external component requiring no modification to the core. The FPGA BRAM could readily serve as a fast directly mapped cache, and the already functioning AXI master could be used to make the DDR RAM the slow main memory (such access usually has latency in excess of 10 cycles while the BRAM takes only one cycle). Caches can be optionally enabled in QtRvSim, so it is within project scope.

Branch prediction or decision acceleration (the branches are decided in MEM to make design cleaner, but it can be done earlier) would not be of much use with such a short pipeline, but it could be done: in the currently supported instruction set, jump instructions can be quickly distinguished by their opcode, which is always at same position in the instruction word, so there is free time even in decode stage for some computation.

A breakdown of options by jump type follows:

- **JAL** - not conditional and PC relative, so there is actually no need for predicting. The jump could be moved definitely from MEM to EX, probably even ID considering that the workload is just addition of constant to PC and multiplexers on opcode and next PC, which is less than E stage does. The reason why it is in Memory stage is because it it so in QtRvSim and also this way all jumps are handled in the same entity.

- **JALR** - similar to JAL, but requires register read which happens in ID, and after that an immediate needs to be added to the target register, so this one would most likely have to be in EX.

- **Conditional jumps** - current instruction address in IF, so alongside the instruction a directly mapped cache of taken/not taken could be read, and then use or ignore its result in ID depending on whether the instruction is a jump, resulting in a workload same as JAL, just with an additional condition. Then in MEM stage, the flush and new PC would be switched based on combination of predicted and actual decision, rather than just the decision, which would only require a di erent branch unit. For the cache a BRAM could be used, which has 8kB, therefore space for 8192 mappings.

- **Universal** - Instead of trying to take advantage of properties of particular jump instruction, it would be possible to implement general branch target and history bu er and read them in parallel with instruction memory. Then the final resolution in MEM stage would update the predictor state. This would give predictions immediately, with no wasted time on success, but possibly lower succes rate. Implementing the various approaches and comparing their performance would be a fitting expansion of this project.

8.2.1 Address Space

Data and instruction memories are physically separated in this design, but they share address space. This is wasteful, because gcc can not work with this and will only be able to use half of the actually present memory. It would be better if the memories had di erent addresses; a linker script could then easily place the .text and .data sections in the correct memory. Since the size of the BRAMs on Zynq is always power of two, truncating the address would be enough to adjust it to hardware reality.

8.3 Limitations of Technology

This core was designed to be modular, and indeed its components can be exchanged easily, but we are limited by the wiring between them, and their interfaces. It is of course possible to not use some signals, but there is no way to easily change flow of data through the modules. A good example of this is in the ALU 8.16, where a design decision suboptimal from hardware resource usage viewpoint was made in the interest of simplicity, but now there is no way to easily maintain two version of the design: a simple demonstration one and a more serious for advanced study or even practical applications such a motor control discussed in7.

Similar problem would occur if branch prediction, discussed in previous section8.2, was to be implemented, as it would require a lot of new wiring to route the branch result to program counter earlier.

A more modern HDL langugage with some advanced polymorphic features or other way of designing more modifiable interfaces, like generate statements but for ports, might therefore be better of this project, particularly if various cache and branch prediction methods were to be implemented and compared.

Bibliography

[1] Dupák, J.; Píša, P.; Štepanovský, M.; Ko í, K. QtRVSim, "RISC-V Simulator for Computer Architectures Classes", *embedded world Conference 2022* [Online] https://comparch.edu.cvut.cz/publications/ewC2022-Dupak-Pisa-Stepanovsky-QtRvSim.pdf

[2] Dupák, J.; Píša, P. https://dspace.cvut.cz/bitstream/handle/10467/94446/F3-BP-2021-Dupak-Jakub-thesis.pdf

[3] E. Lavuš, "RISC-V processor designed in VHDL to match QtRvSim", [Online] https://gitlab.fel.cvut.cz/b4m35pap/rvapo-vhdl

[4] stnolting, "SoC built around a RISC-V CPU", [Online] https://github.com/stnolting/neorv32

[5] sergeykhbr, "SoC built around a RISC-V CPU", [Online] https://github.com/sergeykhbr/riscv_vhdl

[6] Colin Riley, "Basic RISC-V CPU implementation in VHDL", [Online] https://github.com/Domipheus/RPU

[7] evensgn, "RISC-V CPU: course project", [Online] https://github.com/Evensgn/RISC-V-CPU

[8] "The C11 standard", [Online] https://port70.net/ñsz/c/c11/n1570.html#6.3.2.3p7

[9] riscv.org, "Learn RISC-V" [Online] https://github.com/riscv/learn

[10] https://github.com/iammituraj/pequeno_riscv

[11] riscv.org,"GCC for risc-v" https://github.com/riscv-collab/riscv-gnu-toolchain

[12] "ncurses", https://invisible-island.net/ncurses/ncurses.html

[13] "ICE-V Wireless", https://github.com/ICE-V-Wireless/ICE-V-Wireless

[14] ARM Holdings, "AMBA AXI Protocol Specification", [Online] https://developer.arm.com/documentation/ihi0022/k/?lang=en

[15] "OSS CAD Suite", https://github.com/YosysHQ/oss-cad-suite-build

[16] The Rust Reference, [Online] https://doc.rust-lang.org/reference/type-layout.html

[17] Prudek Martin; Píša, P., "Brushless motor control with Raspberry Pi board and Linux", [Online] https://dspace.cvut.cz/bitstream/handle/10467/62036/F3-BP-2015-Prudek-Martin-Bp_2015_prudek_martin.pdf

[18] "The RISC-V organisation" , https://riscv.org/

[19] https://github.com/BrunoLevy/learn-fpga/blob/master/FemtoRV/TUTORIALS/FROM_BLINKER_TO_RISCV/README.md#step-12-size-optimization-the-incredible-shrinking-core

[20] "The RISC-V organisation", https://wiki.riscv.org/display/HOME/RISC-V+Technical+Specifications

[21] m-labs, "Migen logic design toolkit", https://m-labs.hk/gateware/migen/

[22] https://stackoverflow.com/questions/36852808/modify-ice40-bitstream-to-load-new-block-ram-content

[23] "RVapo code repository", https://gitlab.fel.cvut.cz/otrees/fpga/rvapo-vhdl

[24] "PXMC code repository", https://gitlab.fel.cvut.cz/otrees/motion/pxmc-linux/-/tree/mz_apo-pmsm-rvapo?ref_type=heads

[25] https://gitlab.com/pikron/projects/lx_cpu/lx-rocon

[26] "PXMC library", https://pxmc.org/

[27] https://github.com/BrunoLevy/learn-fpga/tree/master/FemtoRV

[28] T. Kranenburg, "MB-Lite", https://opencores.org/projects/mblite

[29] P. Porazil, "MZ-APO board" ,https://gitlab.com/pikron/projects/mz_apo/microzed_apo

[30] Píša, P., "Computer architectures course at CTU", https://cw.fel.cvut.cz/wiki/courses/b35apo/en/documentation/mz_apo-howto/start

[31] CTU, "CAN FD IP core" https://canbus.pages.fel.cvut.cz/

Appendix A – List of attachments

The following is also a part of this work:

- **rvapo-vhdl** The processor source code with all its utilities, also found in a gitlab repository[23].
- **pxmc-linux** Motor controller using the processor designed in this work, also found in a gitlab repository in the mz_apo-pmsm-rvapo branch[24].
- Appendix B - Core Documentation - more detalied description of the processor core.
- Appendix C - motor control hardware schematic - schematic of the board used to connect the motor to mzapo board.
- Appendix D - MicroZed APO schematic - schematic of the mzapo board.

Appendix B – Core Documentation

This chapter describes the internal workings of the RVapo core. It has one interconnect that puts the core together from modules. The goal is that all modules – except for register file – should be stateless combinational logic, and in singlecycle processor variant this is achieved. In the pipelined variant, IF, EX and MEM can do multicycle operations. EX achieves this using clock, but MEM goes around it by using a feedback through interconnect as input and in IF the stall is controlled externally by the instruction memory.

Following is a diagram of the core design, with blue blocks in the middle being pipelined.

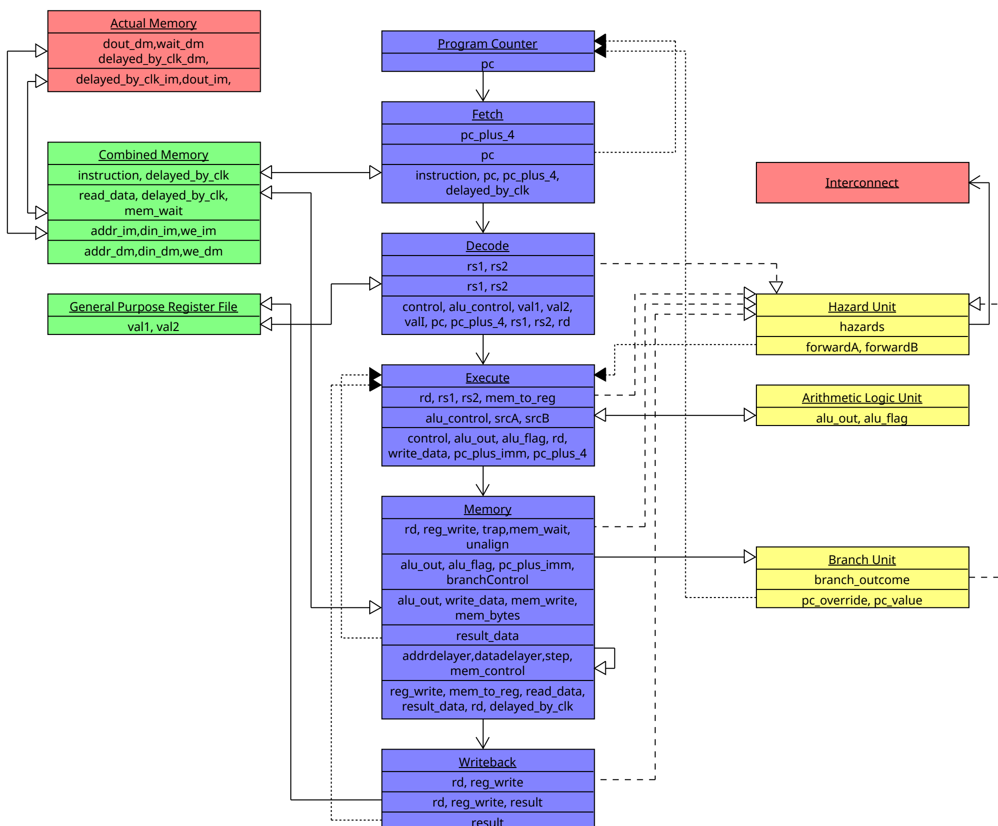


Figure 8.1: core

Next are tables describing entities and the signals between them. Signals with the role PASS are just passed on from another component without change. Records

that repeat in the interface of entity are elaborated only once. Also, the records connecting stages are always last in their source components and first in their target components. Because stage components are listed here in order of pipeline, this would mean describing the same record twice in immediate succession. Instead, it is only detailed in source component.

8.4 General Records

Records that are used several times in various components are described in detail here and only referenced later.

ControlUnit_t	
Name	**Description**
branch	Type of branch instruction, can be none.
reg_write	Result of instruction is written to register.
mem_to_reg	Result of instruction comes from memory.
mem_write	Instruction writes to memory.
mem_control	Memory load/store width and extend behavior.
pc_plus_imm_to_reg	Result of instruction is PC+immediate.
trap	Instruction is a trap – fault or intentional stop.
Clock_t	
pulse	Clock pulse.
enable	Turn the core on/o . Ignored, as reset does the same.
reset	Reset the core to default values.

8.5 Main Interconnect

Comes in two types: pipelined and singlecycle. Both have the same interface. Since they are top-level components, no records can be used in their interface, as Vivado (and possibly other synthesis tools) does not support that.

Aside from interconnecting components, they maintain global state of the CPU by means of a automaton with following states:

- Stack initialisation, where register and memory control signals are redirected to read initial stack value from predefined location (see 6.3). This actually includes two states to support delayed memory. Delay longer than one cycle is not supported, although it could be easily added by stalling the state change by mem_wait signal.
- PC initialisaton, similar to stack.
- Normal execution, the only state where pipeline is moving. Singlecycle version has single-stage pipeline, with the stage register being PC.
- Trapped state. Causes described in 4.4.

Pipelined Differences

The main di erence is that pipelined interconnect inserts registers between stages, and stalls/flushes them as required by hazard unit (see 4.3). It also sends a 'fresh'

signal to stages that are not being stalled so as to only start any internal state automaton when there is new work. Otherwise two multicycle operations of di erent length starting at once could lead to deadlock.

Part of the 'fresh' mechanism are also memory operation requests to prevent repeating AXI transaction or other slow operations when the CPU is stalled. The default is that interconnect automatically issues a request every time IF or MEM is 'fresh'. For unaligned data memory access it becomes more complicated since two memory transaction may be required for one program instruction, so when the splitting of unaligned accesses is enabled, MEM stage entity is responsible for issuing data memory requests.

When memory read is delayed by one clock cycle, it also does register bypassing, and modifies the word read from memory when the instruction is other than lw, since the M stage's logic for this gets bypassed.

It controls debugger, which means following tasks:

- Filtering the debug control signals: internal step and issue are raised for one clock period after rising edge of the external ones. Essentially reencoding information from edges to levels.
- Exposing result of instructions. This is complicated by the NOPs that are issued when pausing, since their result is a zero which would overwrite the debug instruction result after one clock period, making it hard to read. This is solved by clearing the result on issue of debug instruction and then changing it only when instruction result is not zero.
- Restarting execution after trap if debug signals command it.
- Generating control of debug memory (optionally per generics).

core: InterconnectPipeline		
Role	Name	Description
	clk	Clock pulse.
	rst	Reset the core to default values.
	trapped	Indicates that the core is halted and the reason.
	addr_hm	Debug memory address to write.
	din_hm	Data to debug memory.
	en_hm	Enable debug memory.
	we_hm	Debug memory byte enable.
	dsel	Select input to debug memory.
PASS	addr_dm	Data memory address.
PASS	din_dm	Write to data memory.
	dout_dm	Read from data memory.
PASS	en_dm	Enable data memory.
PASS	rst_dm	Reset the data memory.
PASS	we_dm	Data memory byte enable.
	delayed_by_clk_dm	Data memory read is delayed by one clock cycle.
	wait_dm	Data fetch time unknown, just wait.
	req_dm	New data memory transaction request.
PASS	addr_im	Instruction memory address to read.
PASS	din_im	Write to instruction memory. Not used.
	dout_im	Read from instruction memory.
PASS	en_im	Enable instruction memory.
PASS	rst_im	Reset the instruction memory.
PASS	we_im	Instruction memory byte enable.
	delayed_by_clk_im	Instruction memory read is delayed by one clock cycle.
	wait_im	Instruction fetch time unknown, just wait.
	req_im	New instruction memory transaction request.
	pause	Pause execution of program and enable debugger.
	issue	Debug instruction is valid and should be executed.
	step	Execute one program instruction.
PASS	debug_instr	The debug instruction.
	result	Result of last debug instruction.

8.6 Program Counter

Selects next value of program counter – PC+4 or branch unit output.

inst_pc: StageProgramCounter		
Role	Name	Description
	Intcon_F2PC	Fetch -> Program Counter
	pc_plus_4	PC+4.
	Intcon_BU2PC	Branch unit -> Program Counter
	pc_override	Indicates jump.
	pc_value	Jump target.
	Intcon_PC2F	Program Counter -> Fetch
	pc	Address of next instruction, selected by pc_override.

8.7 Fetch

Fetches next instruction from memory and increments PC. It also implements a part of debugger – it can pause the CPU by not incrementing PC and sending NOPs or debug instruction to ID. This behaviour is controller by level of appropriate signal, so to really make one step or issue debug instruction once, one-cycle duration of apropriate signals must be ensured. This is normally done by interconnect.

Note that pause signal of debugger is not stall. The pipeline keeps running, but debugger instructions are injected into it instead of those from instruction memory.

inst_f: StageFetch		
Role	**Name**	**Description**
	Intcon_PC2F	Program Counter -> Fetch
	Intcon_IM2F	Instruction memory -> Fetch
	instruction	Instruction read from memory.
	delayed_by_clk	Indicates that memory read is delayed.
	Intcon_I2F	Interconnect -> Fetch
	pause	Pause execution and enable debugger.
	issue	Debug instruction is valid and should be executed.
	step	Execute a program instruction.
PASS	debug_instr	The debug instruction.
	Intcon_F2IM	Fetch -> Instruction memory
PASS	pc	Address of next instruction.
	Intcon_F2PC	Fetch -> Program counter
	pc_plus_4	PC+4.
	Intcon_F2D	Fetch -> Decode
PASS	pc	Address of current instruction.
	pc_plus_4	PC+4.
PASS	instruction	Instruction read from memory.
PASS	delayed_by_clk	Indicates that memory read is delayed.

8.8 Decode

Decodes instruction, generating control signals for other core components and reads registers. rv32im supported, as well as the clz instruction from the rv32_Zbb set, added because division4.8 needed a leading zero counter and might as well use it in full. This entity is fully combinatorial.

inst_d: StageDecode		
Role	**Name**	**Description**
	Generics	
	enable_M	Enables the rv32m instruction set extension.
	Intcon_F2D	Fetch -> Decode
	Intcon_GPR2D	Register file -> Decode
	val1 val2	Value of first read register. Value of second read register.
	Intcon_D2HU	Decode -> Hazard unit
	rs1 rs2	Number of first read register. Number of second read register.
	Intcon_D2GPR	Decode -> Register file
	rs1 rs2	Number of first read register. Number of second read register.
	Intcon_D2E	Decode->Execute
PASS	pc	Address of current instruction.
PASS	pc_plus_4	PC+4.
	rs1	Number of first read register.
	rs2	Number of second read register.
	rd	Number of write register. Zero if none.
PASS	val1	Value of first read register.
PASS	val2	Value of second read register.
	val1	Immediate value.
	control	Control signals.
	alu_control	ALU operation. ADD if don't care.
	alu_source	Source selectior of ALU B input. Immediate if don't care.
	mtype	Type of M instruction. Valid when alu_control is MUL.

8.9 Execute

Forwards data sources, calculates PC+immediate and communicates with the ALU, which is a separate entity so as to make changing supported instruction set easier. Uses clock for multicycle operations in the M extension.

inst_e: StageExecute		
Role	Name	Description
	Generics	
	enable_M	Enables the rv32m instruction set extension.
	Intcon_D2E	Decode -> Execute
	Intcon_HU2E	Hazard unit -> Execute
	forwardA forwardA	Forwarding control of ALU input A. Forwarding control of ALU input B.
	Intcon_ALU2E	ALU -> Execute
	alu_out alu_flag alu_wait	ALU result. Zero flag. Multicycle operation in ALU, stall.
	Intcon_M2E	Memory -> Execute
	result_data	Forward from memory stage. Either ALU result or PC+4.
	Intcon_W2E	Writeback -> Execute
	result_data	Instrution result forward from writeback stage.
	Intcon_E2HU	Execute -> Hazard unit
PASS PASS PASS	rs1 rs2 rd mem_to_reg	Number of first read register. Number of second read register. Number of write register. Instruction result comes from memory.
	Intcon_E2ALU	Execute -> ALU
PASS	alu_control srcA srca multype fresh	ALU operation. ADD if don't care. ALU input A. ALU input A. Enum that gives type of multiplication instruction. Pipeline has moved, restart internal state automaton.
	Intcon_E2M	Execute -> Memory
PASS PASS PASS PASS PASS	pc_plus_4 pc_plus_imm alu_out alu_flag write_data rd control	PC+4. PC+immediate value. ALU result. Zero flag. Data to write to memory (ALU result is then address). Number of write register. Zero if none. Information from ID about operations to be done.
PASS	fresh	Pipeline has moved, restart internal state automaton.
	req$_d m$	New memory request issued.

8.10 Memory

This stage manages memory access. In the basic version, it is quite straightforward – it just sets byte mask, sign extends and trims the 4B word as required (RV32I instruction set allows reading Word, Halfword and Byte).

The version that correctly implements unaligned access is much more complicated. It is implemented around a double-width byte enable register, illustrated in Figure 8.2.

One in this register means that the corresponding byte should be read or written, so when the upper part of the register is not empty, memory access is repeated on a

Step 1	0	0	0	1	1	0	0	0
Step 2	0	0	0	0	0	0	0	1

Figure 8.2: Byte access register.

higher address.

This means that this entity has a state, which is maintained using a feedback loop through interconnect, so the entity itself is not aware of clock. This means that advancing of state can be easily stalled externally, which is useful for accessing AXI bus and other slow peripherals.

At most two accesses can be performed. The final result is then assembled from combination of current operation result and previous result from the feedback loop.

Another problem that arises is that regular FPGA memory has one clock cycle delay. This could be adressed by stalling, but then unaligned memory read would take four cycles. Instead the read has three steps, where the data read during first one is ignored, since it is actually data from previous instruction.

Aside from all that, the M stage also does first step of selecting instruction result – ALU output, PC+immediate or PC+4 (the last one is used for JAL instructions). Result of this selection is used in short forwards.

inst_e: StageMemory		
Role	**Name**	**Description**
	Intcon_E2M	Execute -> Memory
	Intcon_DM2M	Data memory -> Memory
	read_data	Read from data memory.
	delayed_by_clk	Indicates that memory read is delayed.
	mem_wait	Memory read with arbitrary delay.
	Intcon_M2HU	Memory -> Hazard unit
PASS	rd	Number of write register. Zero if none.
	reg_write	Instruction result shall be written to register.
	trap	Indicates a trap instruction.
PASS	mem_wait	Memory read with arbitrary delay.
	unalign	Unaligned memory access.
	Intcon_M2BU	Memory -> Branch unit
PASS	alu_out	ALU result
PASS	alu_flag	Zero flag
PASS	pc_plus_imm	PC+immediate value
	branch	Type of branch to do.
	Intcon_M2DM	Memory -> Data memory
PASS	alu_out	ALU result, serves as address.
PASS	write_data	Data to write to memory.
	mem_write	Instruction writes to memory.
	mem_bytes	Byte enable for memory writing.
	Intcon_M2E	Memory -> Execute
	result_data	Forward from memory stage. Either ALU result or PC+4.
	Intcon_M2W	Memory -> Writeback
PASS	read_data	Read from data memory.
	result_data	ALU out or PC+4 depending on instruction.
PASS	rd	Number of write register. Zero if none.
	reg_write	Instruction result shall be written to register.
	mem_to_reg	Instruction result comes from memory.
PASS	delayed_by_clk	Indicates that memory read is delayed.
	Intcon_M2M	Memory -> Memory
	addrdelayer	Address on previous clock cycle.
	datadelayer	Read/write data on previous clock cycle.
	step	State of unaligned memory access.
	mem_control	Memory control command on previous clock.
	fresh	Pipeline has moved, restart internal state automaton.

8.11 Writeback

Does second step of selecting instruction result – memory or result of first step done in Memory stage. This is written to registers and also used in long forwards.

When memory read is delayed by clock edge, the correct value is available only for one clock period. Should such read be followed by a stall in any pipeline stage, M2W.read_data will become invalid. Therefore writeback stage must contain a register to preserve the correct value. Gating register write using fresh will not work on forwards and debugger results.

inst_w: StageWriteback		
Role	**Name**	**Description**
	Intcon_M2W	Memory -> Writeback
	Intcon_W2HU	Writeback -> Hazard unit
PASS	rd	Number of write register. Zero if none.
PASS	reg_write	Instruction result shall be written to register.
	Intcon_W2GPR	Writeback -> Register file
	result	Result of instruction.
PASS	rd	Number of write register. Zero if none.
PASS	reg_write	Instruction result shall be written to register.
	Intcon_W2E	Writeback -> Execute
	result_data	Instruction result forwarding.
	fresh	Pipeline has moved, restart internal state automaton.

8.12 Combined Memory

Combines data and instruction memory into one entity and converts signals from RVapo internals to memory interface. The only di erence is that RVapo uses byte enable for writing and reading, while Zynq FPGA RAMs require byte enable to be all zeros for reading and always read whole word.

inst_cm: CombinedMemory		
Role	**Name**	**Description**
	Clock_t	Interconnect -> Combined memory
	Intcon_F2IM	Fetch -> Instruction memory
	pc	Address of next instruction.
	Intcon_M2DM	Memory -> Data memory
PASS	alu_out	ALU result, serves as address.
PASS	write_data	Data to write to memory.
	mem_write	Instruction writes to memory.
	mem_bytes	Byte enable for memory writing.
	Intcon_IM2F	Instruction memory -> Memory
	instruction	Instruction read from memory
	delayed_by_clk	Indicates that memory read is delayed
	Intcon_DM2M	Data memory -> Memory
PASS	read_data	Read from data memory.
PASS	delayed_by_clk	Indicates that memory read is delayed.
PASS	mem_wait	Memory read with arbitrary delay.
	Intcon_DM2V	Combined memory -> Actual data memory
PASS	addr	Address to read/write.
PASS	din	Data to memory.
	dout	Data read from memory.
	en	Enable memory.
	rst	Reset the memory.
	we	Byte enable.
PASS	delayed_by_clk	Memory read is delayed by one clock cycle.
	mem_wait	Memory read with arbitrary delay(stall until 0).
	Intcon_DM2V	Interconnect -> Actual instruction memory

8.13 Register File

General purpose registers, with bypassing to read a value in same cycle it was written. The bypassing is disabled in single cycle core, which is implemented using VHDL generics.

This entity is actually just adapter, like Combined Memory, but the actual register definition entity, named GeneralPurposeRegister_Target, has identical interface, so it was omitted. The purpose of this construct, left over from Lavus's work, is presumably support of FPGA platforms that can not infer the register file correctly, so then the actual implementation of registers could be swapped easily.

The Decode stage does no operations on the registers, so they could, for instance, be implemented using the BRAM hardware macro, with its output registers being integrated in the Decode-Execute stage register in a manner similar to register bypassing on memory4.2. This trick could be practically useful when implementing multiple register sets for faster interrupt handling or switching process contexts, where register sets could be switched by adding prefix to register address at almost zero cost.

inst_gpr: GeneralPurposeRegister		
Role	**Name**	**Description**
	Generics	
	register_bypass	Enables register bypassing.
	Clock_t	Interconnect -> Register file
	Intcon_D2GPR	Decode -> Register file
	rs1 rs2	Number of first read register. Number of second read register.
	Intcon_W2GPR	Writeback -> Register file
	result rd reg_write	Result of instruction. Number of write register. Zero if none. Instruction result shall be written to register.
	Intcon_GPR2D	Register file -> Decode
	val1 val2	Value of first read register. Value of second read register.

8.14 Hazard Unit

Controls forwards and stalling. Stalling or flushing happens, in descending order of priority:

- When stage Fetch is waiting for instruction memory. Branches can not be executed, because PC must remain constant while fetching instruction, so stage MEM where branch unit resides must stall. If stage MEM (and therefore branch unit) was not stalled, the new PC value would be pushed into IF, which would ignore it because it is still fetching instruction from the previous address.
- When waiting for data memory. Raised in uncore (mem_wait). M must obviously stall, and so all before it too. WB is also stalled because result of MEM is not valid. Flushing it would work too, but when WB is stalled, it keeps result of last instruction, which requires no change in forwarding, because the

forwardable value is available until pipeline moves. But if WB were flushed, then the forwarded value would be lost and a new register would have to be inserted somewhere to keep it working.

- When doing unaligned memory access. Raised in core (unalign). The only di erence from above is that MEM feedback is not stalled. It has lower priority so that when unaligned access is done into slow memory, both stall signal are raised, and when memory completes operation, MEM feedback is unstalled for one cycle, advancing state of MEM stage while keeping the rest of the processor stalled.
- When branching. Branch stall and memory stall can not happen simultaneously, since branch unit is in MEM stage of pipeline. Therefore it does not matter how its priority compares to that of memory stalls. The reason it does not have higher priority than instruction memory stalls is that it would require ability to abort instructionmemory read, which might not always be possible, and in fact our implementation of AXI bus does not support this.

  ID,EX,MEM are flushed – since reaction to hazard signals is on next clock cycle, this will drop instructions that were in IF,ID,EX when branch was in MEM.
- When data needed by this instruction was loaded from memory by the previous one. This can not be forwarded because the load will be in progress while the next instruction is in ALU.

  However, it would be possible to forward if the loaded data was to be stored in the next cycle. This is not implemented.
- When there is multicycle operation in EX.

inst_hu: HazardUnit		
Role	**Name**	**Description**
	Intcon_BU2HU	Branch unit -> Hazard unit
	branch_outcome	Indicates whether a branch is taken.
	Intcon_D2HU	Decode -> Hazard unit
	rs1	Number of first read register.
	rs2	Number of second read register.
	Intcon_E2HU	Execute -> Hazard unit
PASS	rs1	Number of first read register.
PASS	rs2	Number of second read register.
PASS	rd	Number of write register.
	mem_to_reg	Instruction result comes from memory.
	Intcon_M2HU	Memory -> Hazard unit.
PASS	rd	Number of write register. Zero if none.
	reg_write	Instruction result shall be written to register.
	trap	Indicates a trap instruction.
PASS	mem_wait	Memory read with arbitrary delay.
	unalign	Unaligned memory access.
	Intcon_W2HU	Writeback -> Hazard unit
PASS	rd	Number of write register. Zero if none.
PASS	reg_write	Instruction result shall be written to register.
	Intcon_HU2E	Hazard unit -> Execute
	forwardA	Forwarding control of ALU input A.
	forwardB	Forwarding control of ALU input B.
	hazards_t	Hazard unit -> Interconnect
	stall_fetch	Stall PC to Fetch latch.
	stall_decode	Stall Decode to Execute latch.
	flush_decode	Stall PC to Fetch latch.
	flush_execute	Zero out the Decode to Execute latch.
	flush_memory	Flust Execute to Memory latch.
PASS	trap	Indicates a trap instruction.
	stall_all	Stall Execute to Memory and Memory to Writeback.
	stall_mem_feedback	Stall feedback loop of Memory.

8.15 Branch Unit

Decides whether and where to branch using result of arithmetic operations done in Execute stage. It does oes not compute branch target of conditions on its own, rather it is just a if-else sequence. This is because all calculations required for jumps can be reused for other instructions.

inst_bu: BranchUnit		
Role	**Name**	**Description**
	Intcon_M2BU	Memory -> Branch unit
PASS	alu_out	ALU result
PASS	alu_flag	Zero flag
PASS	pc_plus_imm	PC+immediate value
	branch	Type of branch to do.
	Intcon_BU2HU	Branch unit -> Hazard unit
	branch_outcome	Indicates whether a branch is taken.
	Intcon_BU2PC	Branch unit -> Program Counter
	pc_override	Indicates jump
	pc_value	Jump target

8.16 Arithmetic-logical Unit

Calculates arithmetic and logic operations required for the RV32I instructions, which means addition, subtraction, bit shift, and, or, xor, compare. All these are combinatorial and meant to be finished in a single cycle. Optionally includes a submodule for the M extension.

A major downside of its design, is that it provides result of only one chosen operation, which makes implementation of instructions that require two ALU operations ine cient. As of now that means only conditional jumps, which require comparison of two registers and addition of PC and immediate value. This can be resolved fairly easily by adding an independent adder, but could be problematic if instruction set was to be expanded more. It would also be wasteful if instruction-level parallelism were to be implemented like in performance cores.

This design choice carries over from QtRvSim where it was made in interest of simplicity; having two outputs and four inputs to the ALU would make its schematic3.1 a lot messier, and experience from teaching shows that students struggle with it as is.

8.16.1 Optimised ALU

The basic ALU is written as simple as it gets, with a switch on `alu_control` signal and each operation implemented in its own case. Due to space constraints on the ice40 FPGA, an optimised ALU was implemented which modifies subtraction so that its result can be used in comparison, and also has only one barrel shifter that can be used for all three shift instructions – right, left arithmetic and left logical. The ideas are taken from[19].

inst_alu: ArithmeticLogicUnit		
Role	Name	Description
	Intcon_E2ALU	Execute -> ALU
	alu_control srcA srca multype fresh	ALU operation. ADD if don't care. ALU input A ALU input A Type of multiplication. Pipeline has moved, restart internal state automaton.
	Intcon_ALU2E	ALU -> Execute
	alu_out alu_flag alu_wait	ALU result Zero flag Multicycle operation, stall.

8.17 M Calculator

Calculates arithmetic operations required for the rv32m instructions, which means multiplication, division and modulo. Algorithms are described here4.8. Time required for modulo and division depends on operands. Multiplication is done in one cycle. However, increasing the instruction set means increasing complexity of multiplexers, and furthermore Vivado recommends registers at input and output of DSP blocks it uses for multiplication. Therefore, the instruction of multiplication actually takes three cycles to execute, with the middle one being used only for the calculation.

Requires a submodule for the count leading zeros operation, which is outsourced so that it may be reused for the rv32_Zbb extension. It gives no indication when exactly is this operation required, instead it is assumed that output of clz will be routed to this entity whenever it is active, in line with the concept that ALU computes exactly one operation at a time.

inst_m: Multiply		
Role	Name	Description
	Clock_t	ALU -> Multiply
	Intcon_ALU2MUL	ALU -> Multiply
	fresh operation start asigned bsigned a b	Pipeline has moved. What calculation should be done. Calculation requested. Ignored unless fresh asserted. A should be treated as signed. B should be treated as signed. operand A (dividend) operand B (divisor)
	Intcon_MUL2ALU	Multiply -> ALU
	c done clz_request	Computation result Result is valid. The value where leading zeros should be counted.

8.18 Bitmanip

Entity for implementation of bit manipulation instruction. Currently only clz from the rv32_Zbb set is implemented, because it is used for division.

inst_m: Multiply		
Role	**Name**	**Description**
	Intcon_ALU2B	ALU -> Multiply
	srcA srcB	operand A operand B
	Intcon_B2ALU	Multiply -> ALU
	result	result

Appendix C – motor control hardware schematic

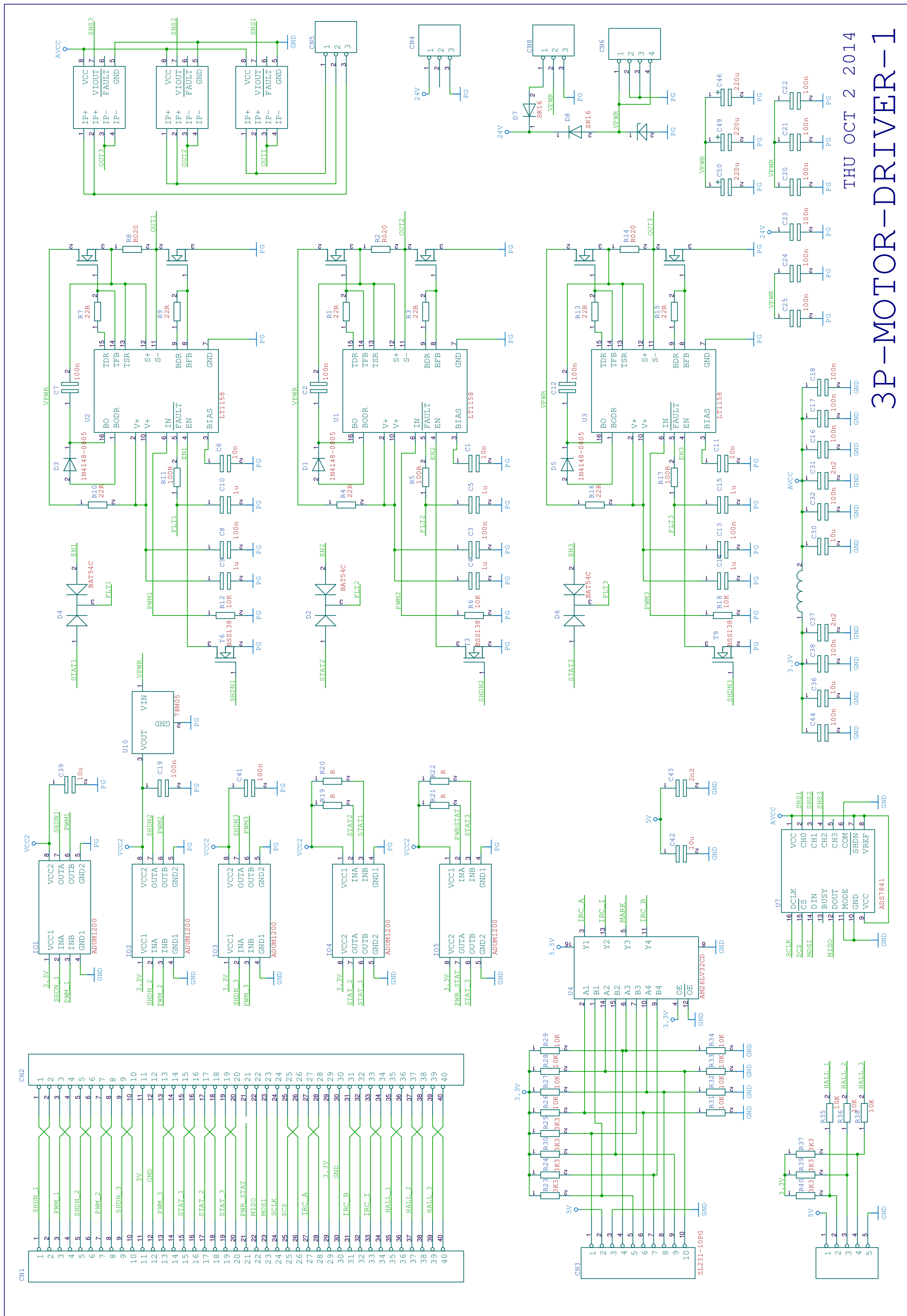


Figure 8.3: Motor HW schematic

Appendix D – mzapo board schematic

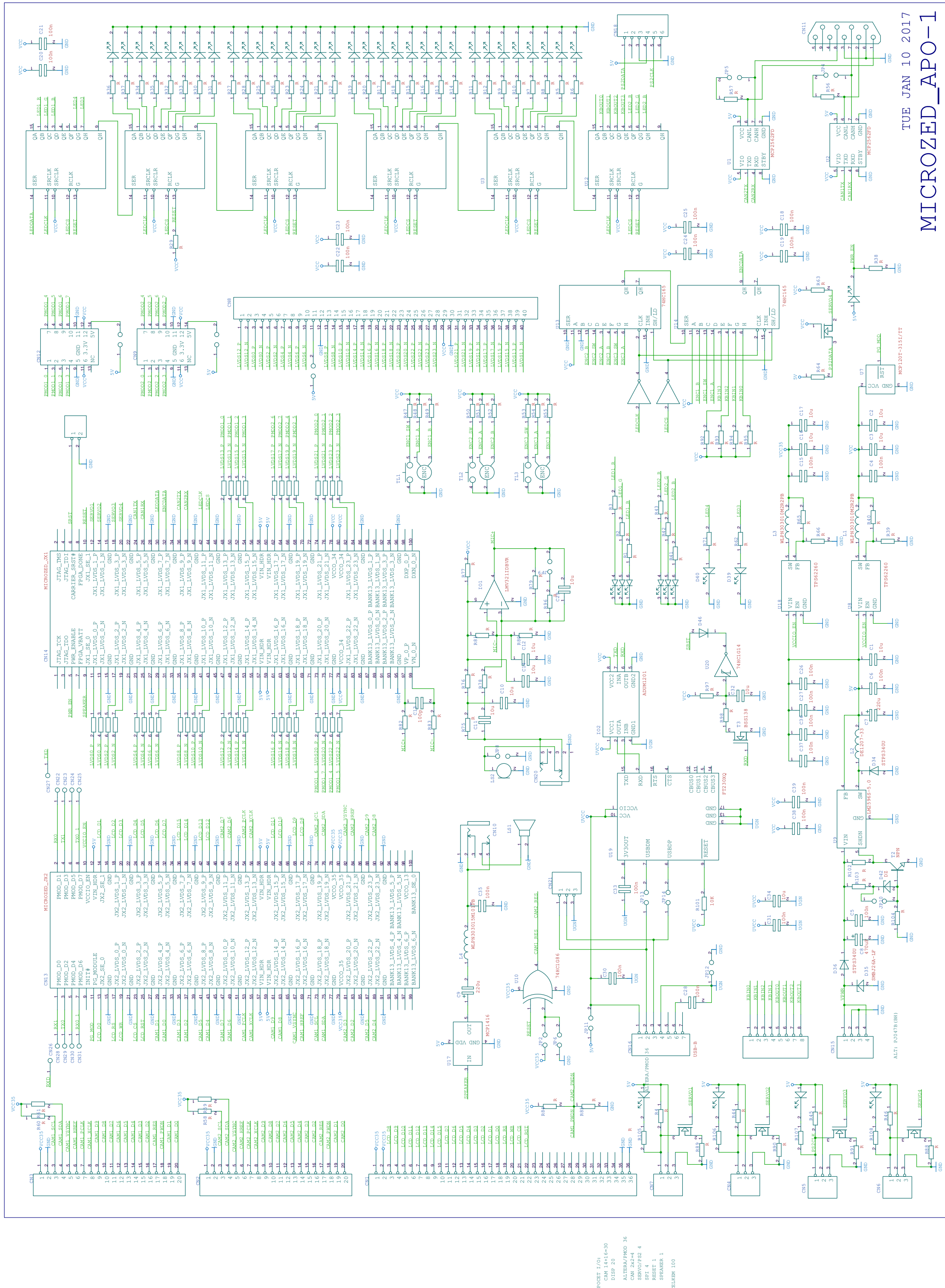


Figure 8.4: mz-apo board schematic