RISC-V: A Didactic Platform
Report

Jules PERRIN, Professor: Theo KLUTER

June 12, 2023

Contents

1 Introduction

1.1 Motivation

This project, RISC-V: A Didactic Platform, embarks on the pursuit of simplifying complex computer architecture concepts, targeting an academic audience who is striving to comprehend the intricate world of processor design. It specifically focuses on the creation of a RV32I processor, based on the RISC-V instruction set architecture.

RISC-V, an open standard instruction set architecture (ISA), has increasingly gained popularity due to its simplified design and flexibility for customization. While its usage has been witnessed across a multitude of applications, its potential as an educational tool remains largely unexplored. The objective of this project is to bridge this gap by providing a detailed, comprehensible, and implementable design of an RV32IM processor, leveraging the RISC-V architecture.

This project not only contributes to the existing body of knowledge around RISC-V and processor design but also aims to democratize access to information on complex computing architectures. Through this platform, users will not only learn the theory behind processor design but will also gain valuable hands-on experience with the actual implementation, enabling them to better understand the interplay between hardware and software in a computing system.

In an era where the understanding of computer architecture is pivotal for both hardware and software development, a project like RISC-V: A Didactic Platform can serve as a crucial resource. By harnessing the power of the RISC-V architecture and presenting it through a user-friendly and accessible platform, we hope to elevate the understanding of processor design to new heights.

1.2 Project Scope

Here are the different main focus points of the project:

- Openess: It is an important point for the project to use as much as possible open source software and also for the project to be as open as possible such that people can work on it with the less possible restrictions.
- Comprehensive: Since the project needs to be open source and be used by academic people to understand but also add things on top of it, the project needs to focus on being as comprehensive as possible.
- Simplicity: This point is linked a bit with the last one, but the project doesn't aim as performance first, but more as ease of understanding, so if a solution can be easier but degrade a bit the performance, it should be taken except if it adds an educational value to do it

1.3 Project Objectives

Overall, the project aims to give a very basic implementation of an RV32IM processor such that it can be used as an academic tool to teach and explain how a RISC processor works and how to build one using HDL languages such as Verilog. Here are the main objectives that has been set for the project:

- First create an unpipelined version of the RV32IM processor as a proof of concept
- Pipeline the existing unpipelined version of the processor
- Fix the different issues caused by the pipelining of the processor, mainly the data dependency but also the branch prediction problem by either stall or implementing some forwarding paths and branch predictor and mechanisms to make everything work correctly
- The simulation should works on Icarus Verilog to make the whole process more open instead of relying on proprietary software such as Questa Modelsim.
- Add extensive testing of the different main modules to make the processor as reliable as possible.
- Create a CI pipeline such that it runs automatically the different tests and checks at each commit to make sure that the code is always working.
- Add an easy way for users to create programs and load them into the ROM memory.
- Document everything such that users can easily understand how things work and can work on it easily.

2 Background Research and Knowledge

2.1 RISC-V

RISC-V is an open-source instruction set architecture (ISA) based on the reduced instruction set computer (RISC) principles. It is a free and open ISA enabling a new era of processor innovation through open standard collaboration.

Most of the research I've done on it to understand how it works and how to implement it has been done on the official website of the RISC-V foundation [1, p. 9-25] and of course more precisely the chapter about the RV32I base integer instruction set. Inside it describes how the different instructions are encoded with the 6 different formats [1, p. 104-105] that are used to encode the different instructions but also how the different instructions should behave using a mix of the manual but also a reference card [2, p. 81-83].

2.2 Architecture

2.2.1 Unpipeline

The unpipelined version of the CPU is mostly inspired by the one we did during the Comparch 1-2 courses at EPFL since NiosII and RISC-V are very similar. The main difference is that the RISC-V is a 32 bits architecture while NiosII is a 16 bits architecture and also that RISC-V has a lot more instructions than NiosII with small differences in how they work.

2.2.2 Pipeline

The pipelined version is inspired also mostly by what we did during the Comparch 2 course at EPFL but since we didn't implement the branch predictor and the forwarding paths, I had to imagine and think about most of the implementation myself. I've only done a couple of research to see if I could find some examples of such implementation mostly about forwarding paths since I wasn't sure how to implement it correctly at first glance. I've used mainly this pdf [3] to reassure myself that I was doing it correctly. The branch predictor part has an algorithm I haven't invented called a two-level adaptive branch predictor [4]. It uses a 2 bits saturating counter but also knows patterns such that the predictions apply locally to the branch. It is a very simple algorithm but it works well enough for this project.

2.3 Verilog

Not knowing anything about Verilog since at EPFL we used VHDL instead, I had to learn it from scratch. For that, I've used a website given by the professor [5] that explains the basics of Verilog and how to use it. From here I've started creating small modules and searching for examples on the internet to see how other people were doing it and also when needed I've used different forums to find answers to my questions and when needed consulted the official Verilog documentation [6].

2.4 Simulation Tools

During the project, I mainly used a close source simulator that we were using at EPFL and developed by Intel called Questa Modelsim. It is a great tool combining compiling and seeing the waveforms of the signals in the same window. It is also very easy to use and has a lot of features that I haven't used during the project. The only downside is that it is not free and it is quite buggy sometimes and you need to activate an option called `full visibilty` to have access to all the signals in the waveform window. At the end of the project, since I had to implement the CI, I needed a free simulator with a command line interface and I've used Icarus Verilog. It is a free simulator that is not too hard to use and has a command-line interface. The only downsides I've found while using it are that there is not a lot of documentation [7] and it is a bit less user-friendly than Questa Modelsim. You have to use a different tool to see the waveforms like GTKWave and you don't have access to the array of signals easily like in Questa Modelsim. Also, one weird issue I've encountered is that the simulation gives different results than Questa Modelsim because the full visibility option does things differently than Icarus Verilog, but I've luckily found the issue even though to this day I don't know why it was doing that.

2.5 RISC-V Toolchain

2.5.1 Developping using Assembly Code

It is not that easy to find a way to compile a RISC-V RV32I assembly code into a hexadecimal file that can be read by the ROM module. So for that, I've found two projects that helped me develop simple programs for testing purposes. The first one is a Python script that converts pseudo instructions into hexadecimal instructions [8]. The second one is a website that converts assembly code into hexadecimal instructions [9]. One last tool I've used to reverse the hexadecimal instructions into assembly code is a website [10] but also used it to modify one instruction at a time to see what it does.

2.5.2 RISC-V GNU Compiler Toolchain

The RISC-V GNU Compiler Toolchain is a set of tools that allows you to compile C/C++ code into RISC-V assembly code. It is composed of a compiler, an assembler, a linker and a debugger. To use it you need to compile it from the source code which was not that easy to do because the documentation for such a small architecture set is not that great and since we're developing bare metal programs, it is even harder to find information about it. So I've used the official documentation [11] and also a tutorial I've found on the internet [12] to compile it and finally use it.

3 Design

3.1 Parameters

All the different parameters/constants used in the design of the processor are accessible in the file `parameters.vh` contained in the `verilog` folder inside the project. I would have loved to not use such technic and instead do enums but Verilog doesn't support them and from what I've seen from other projects, it's one of the most common ways to do it.

3.2 Unpipelined Design

The first step in the project other than learning how to develop in Verilog was to create an unpipelined version of the intended processor. This version is quite more simple than the pipelined version as it doesn't have to deal with the hazards that can occur in a pipelined and issues with branches and jumps but also because it doesn't need any pipeline registers in between the different stages. I will give you an overall view of the design without the details of each connection between the different modules since it is not the main focus of this project and was more of a way to learn how to use Verilog and the tools associated with it but also a temporary step before the pipelined version.

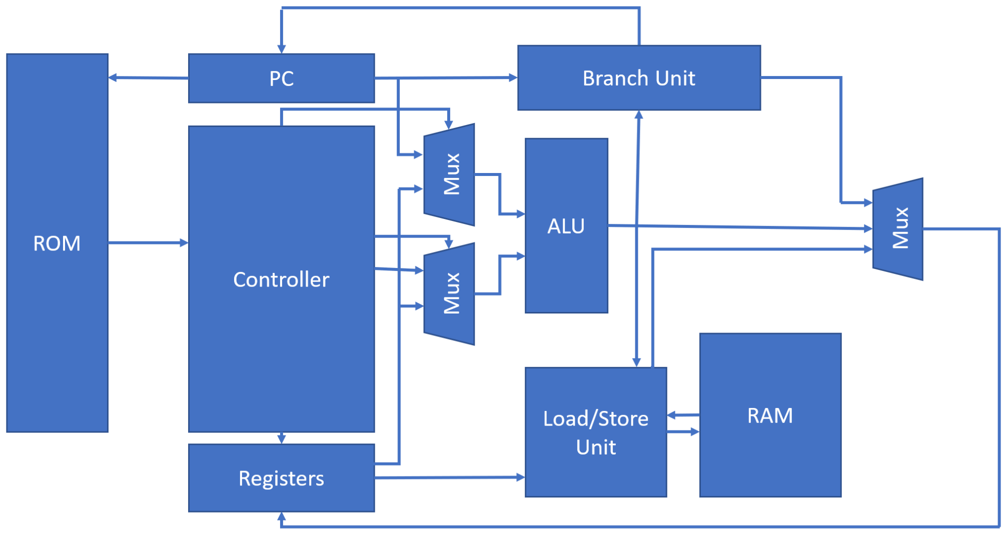

Figure 1: Unpipelined CPU design

I will not go into the details of each module since it will be done in the next section about the pipelined version but I will give you a brief overview of how the unpipelined version works. First, the instruction is loaded from the ROM using the address given by the PC (Program Counter). Then the instruction needs to be decoded to know what it does and what are the operands and that's the job of the Controller that will try to match the different types of instruction to finally find the right one. Once it is done, it will output the different control signals to the register file, the ALU, the branch unit and the load/store unit such that it executes the instruction correctly with the correct data. The role of the ALU (Arithmetic Logic Unit) is to execute the different arithmetic and logic operations such as addition, subtraction, multiplication, division, bitwise operations, etc. In the case of a branch instruction, the branch unit will check if the instruction is a conditional one or not. In case it is one then it uses the result of the ALU to check if the condition is true or not and update the PC accordingly. Finally, the load/store unit will load or store data from or to the RAM depending on the instruction. It is also using the result of the ALU since we can offset the address by an immediate value and that needs to be done by the ALU. At the end of the cycle, a MUX will select among the three different results (ALU, branch unit and load/store unit) the one that will be written to the register file according of course to the instruction executed. Everything is done in one cycle and the next instruction is loaded from the ROM and the process starts again.

As we can see, the unpipelined version is quite simple then why do we need to pipeline it? The answer is performance. Indeed, the unpipelined version is quite slow since it needs to wait for the instruction to be executed before loading the next one. But that can be improved by pipelining the processor that will divide the execution of an instruction into multiple stages and execute multiple instructions at the same time and that's what we will see in the next section.

3.3 Pipelined Design

So what is the purpose of a Pipelined version of a CPU? As stated in the previous section, the unpipelined version is quite slow since it needs to wait for the instruction to be executed before loading the next one. Pipelining follows the idea of dividing the work among smaller parts and executing them in parallel. In the case of a CPU, it means that we will divide the execution of an instruction into multiple stages and execute multiple instructions at the same time. Since each part has a smaller quantity of work to do, we can hopefully increase the clock frequency and thus obtaining in average a better throughput resulting in better performance.

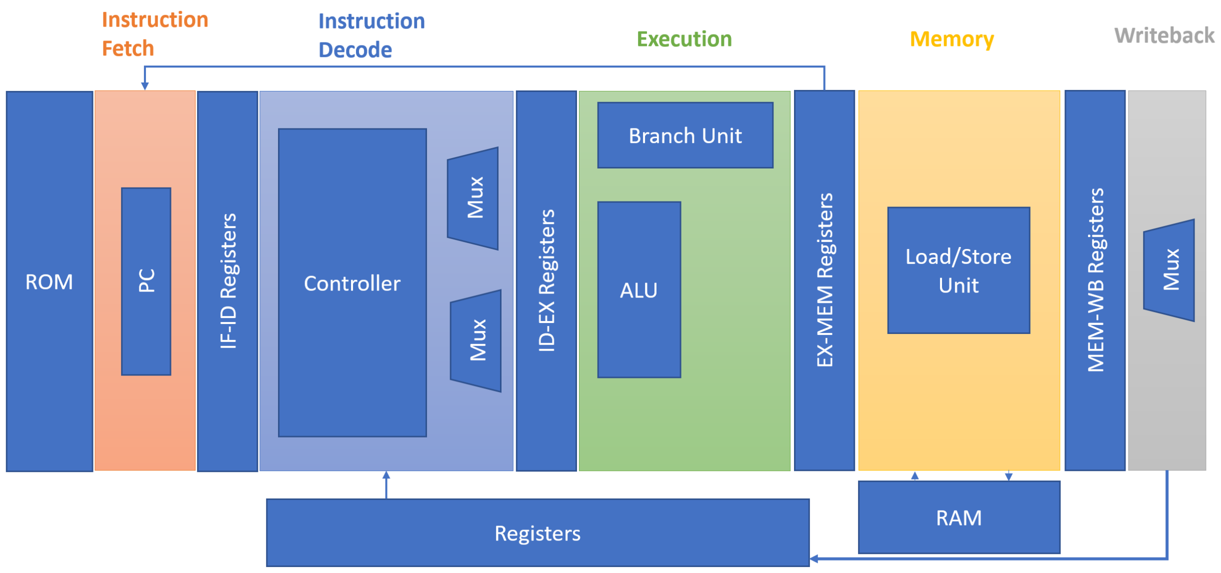

Figure 2: Pipelined CPU design

So here is the idea, we will divide the execution of instruction into five stages: Instruction

Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) and Write Back (WB). Each stage will be executed in parallel and will be connected to the next one using pipeline registers. The pipeline registers are used to store the data between each stage and are synchronized with the clock. The IF stage will fetch the instruction from the ROM using the PC (Program Counter) and will send it to the ID stage. The ID stage will decode the instruction and send the control signals to the different units (ALU, Branch Unit, Load/Store Unit) and will also send the operands to the EX stage. The EX stage will execute the instruction and send the result to the MEM stage. The MEM stage will load or store data from or to the RAM and send the result to the WB stage. Finally, the WB stage will write the result to the register file.

But for the most attentive reader or anybody with a bit of experience in computer architecture, you may have noticed that there is a problem with this design. This design will lead so some issues in two different ways.

- The first one is that since instruction is not executed anymore in one cycle, its result will not be available in the next cycle. This is an issue because if the next instruction depends on the result of the previous one, it will not be able to execute correctly and will use outdated data from the registers. This is called a **Data Hazard**.
- The second one is related to the branching mechanism, since the result of the branch will be available only after the EX stage, the PC will not be updated before that leading to possible wrong instructions being fetched and executed in the pipeline. This is called a **Control Hazard**.

For resolving these two issues we have two solutions. Either stalling the pipeline when a hazard is occurring or when a branch is detected at the IF stage such that we wait for the result before fetching any new instruction. One issue with this approach is of course performance which is unfortunate since the main purpose of pipelining is to improve performance.

Another solution for the data hazard is to use **Forwarding** which consists of forwarding the result of the ALU to the EX stage and the result of the MEM stage to the EX and MEM stages. The only moment we will have to stall the pipeline is when we have a load instruction followed by an instruction using the result of the load. In this case, we will have to stall the pipeline for one cycle to wait for the result of the load.

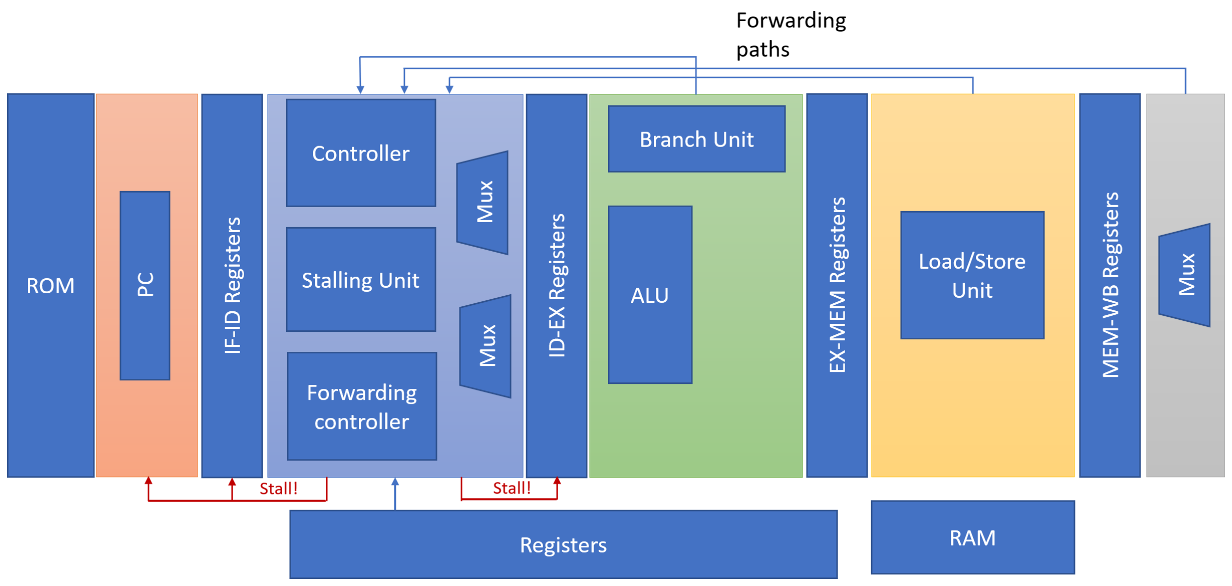

Figure 3: Pipelined CPU design with forwarding paths

Another solution for the control hazard is to use **Branch Prediction** which consists of predicting the result of the branch and fetching the instruction at the predicted address. It is not always possible to predict correctly the result of a branch but it is possible to use some heuristics to improve the prediction accuracy. For example, we can predict that a branch will not be taken if the previous branch was not taken. This is called a **Branch Not Taken** heuristic. Another heuristic is to predict that a branch will be taken if the previous branch was taken. This is called a **Branch Taken** heuristic. Of course, you can develop more complex heuristics but this is out of the scope of this project and I invite you to use the link in the reference to the given algorithm I've chosen to implement in the CPU. But when the prediction is wrong, we will have to flush the pipeline and restart the execution from the correct address.

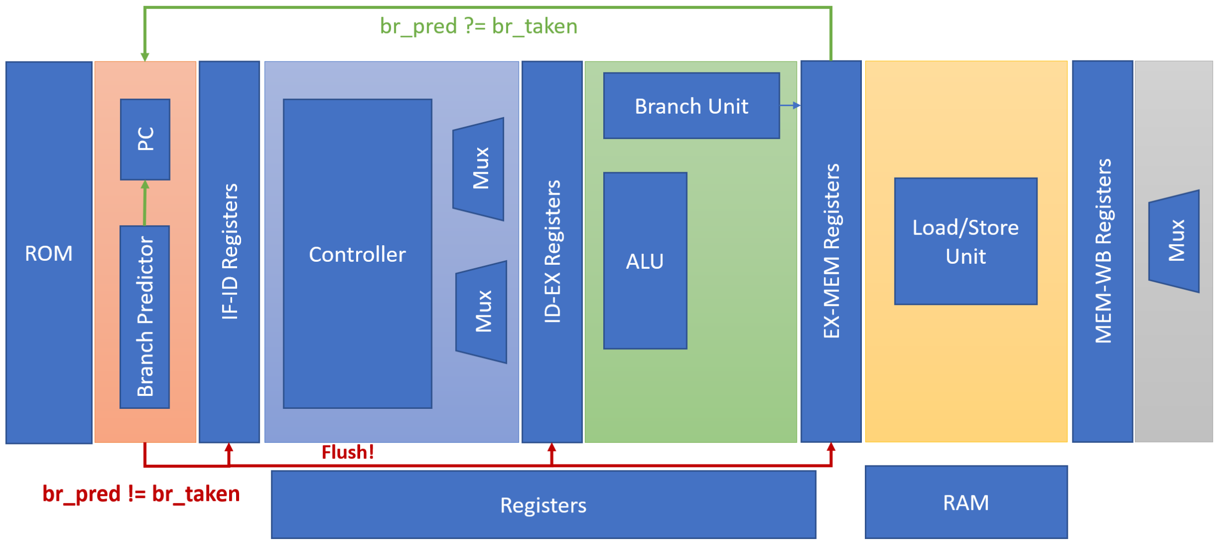

Figure 4: Pipelined CPU design with branch predictor

Now we can see the full simplified representation of the pipelined CPU design.

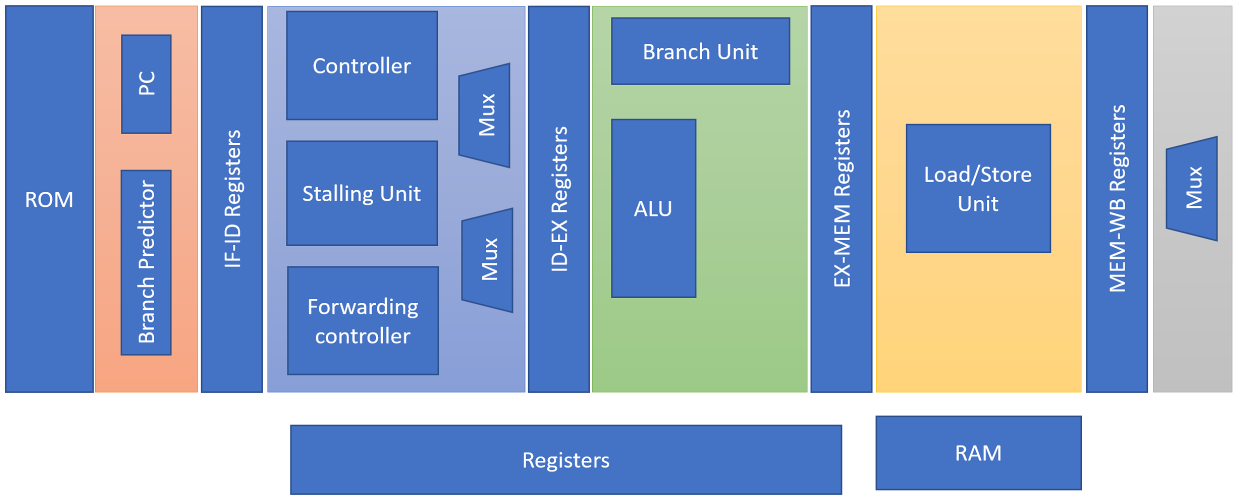

Figure 5: Pipelined CPU simplified design with forwarding paths and branch predictor

A more detailed version with all the different signals and how they are connected is available in the appendix A and also as a diagram inside the project `diagrams` folder that can be opened here.

Now that we have a better understanding of the overall design, I can go into more detail about each different stage and the modules inside of them.

3.3.1 IF Stage

3.3.1.1 PC

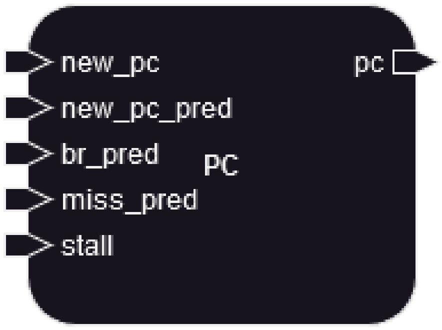

Figure 6: Diagram of the PC

The PC is responsible for computing the next PC. It uses the different input signals to know which PC to compute. for example, if it is a branch, the branch predictor will tell him what to do, and if the branch predictor is wrong it will be updated accordingly. If it is a more classic instruction such as an ADD, it will simply increment the PC by 4 etc.

Signals:

- Input: *new_pc*, This signal represents the next PC given by the branch unit in the EX stage.
- Input: *new_pc_pred*, This signal represents the next PC given by the branch predictor.
- Input: *br_pred*, This signal is representing the state of the prediction made by the branch predictor in the EX stage.
- Input: *miss_pred*, This signal is representing if there is a miss prediction or not.
- Input: *stall*, This signal is representing if the pipeline is stalled or not due to a data dependency in the ID stage.
- Output: *pc*, This signal is representing the current PC.

3.3.1.2 Branch Predictor

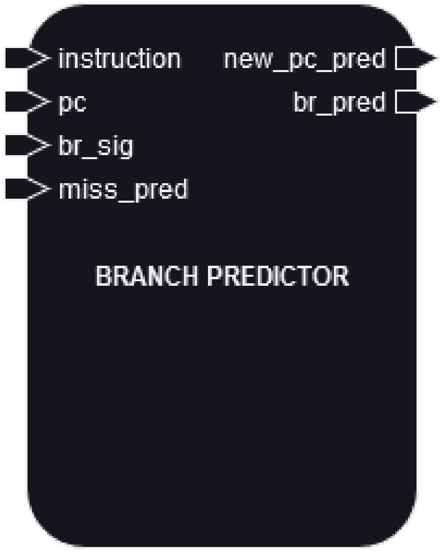

Figure 7: Diagram of the Branch Predictor

The branch predictor as its name suggests is responsible for predicting if the branch will be taken or not. The algorithm that is being used is the two-level adaptive branch predictor [4], which I will not describe in detail but reuse the idea of a 2-bit saturating counter but apply a bit of the notion of locality and pattern recognition. Of course, the algorithm could be improved or replaced by any other algorithm and it is up to the user to do it if he wants to. It works at the beginning by doing a simple matching on the current instruction to see if it is a branch instruction. If that's the case we look if it is a conditional branch or not, if it is not we simply predict that the branch will be taken for the JAL instruction, but the JALR one will be always predicted as not taken for data dependency reasons. If it is a conditional branch we simply use the algorithm described above to predict if the branch will be taken or not. The algorithm updates the prediction depending on the actual result of the branch that is represented by the *miss_pred* signal. If the prediction is taken, we compute the next PC

Signals:

- Input: *instruction*, This signal is representing the current instruction that is being fetched.
- Input: *pc*, This signal is representing the current PC.
- Input: *br_sig* This signal is representing the state of the current instruction in the EX stage. It is used to know if the instruction is a branch or not. such that it updates only the algorithm when it is a branch instruction.
- Input: *miss_pred*, This signal is representing the state of the prediction made by the branch predictor in the EX stage.

- Output: *new_pc_pred*, This signal is representing the next PC that will be used if the prediction is taken.
- Output: *br_pred*, This signal is indicating if the branch is predicted as taken or not.

3.3.2 ID Stage

3.3.2.1 Controller

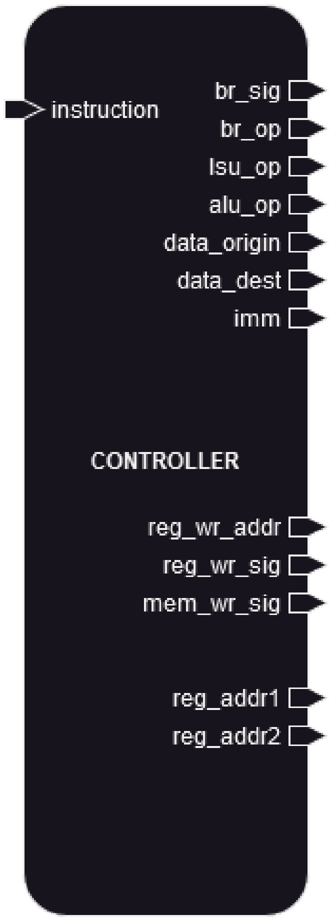

Figure 8: Diagram of the Controller

The controller is the main module of the ID stage. It is responsible for decoding the current instruction which is the action of extracting the different fields of the instruction and forwarding them to the next stage. It follows the 6 different types of encoding of instruction as described if the RISC-V manual [1] to decode correctly the instruction. The controller is the most complex module of the CPU and it took me quite a lot of time to get it right but I also keep it as simple as possible such that it is pretty easy to follow the logic of the module.

Signals:

- Input: *instruction*, This signal is representing the current instruction that is being decoded.
- Output: *br_sig*, This signal is representing the state of the current instruction. It is used to know if the instruction is a branch or not.
- Output: *br_op*, This signal is representing the type of branch that is being executed.
- Output: lsu_op, This signal is representing the type of load or store that is being executed.
- Output: *alu_op*, This signal is representing the type of ALU operation that is being executed.
- Output: *data_origin*, This signal is representing the origin of the data that is being used by the ALU. That could be the registers, or one register and an immediate or one register and the PC.
- Output: *data_dest*, This signal will be useful in the write-back stage to know which data to write back, so either the ALU result or the data from the memory or the next PC (so the PC+4).
- Output: *imm*, This signal is representing the immediate value that is being used by the ALU.
- Output: *reg_wr_addr*, This signal is representing the register address that will be written back.
- Output: *reg_wr_sig*, This signal is representing if we want to write to the register file or not.
- Output: *mem_wr_sig*, This signal is representing if we want to write to the memory or not.
- Output: *reg_addr*1, This signal is representing the first register address that has been extracted from the instruction.
- Output: *reg_addr*2, This signal is representing the second register address that has been extracted from the instruction.

3.3.2.2 Forward Controller

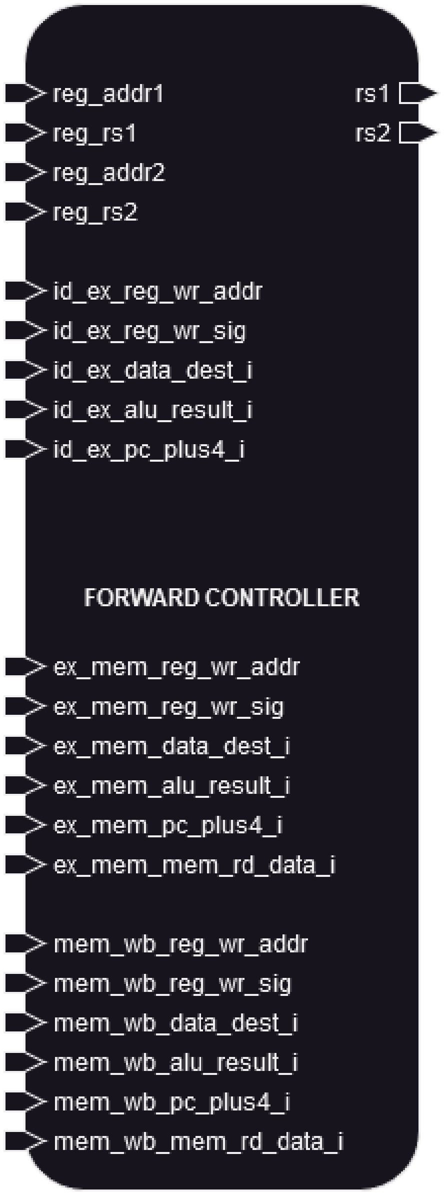

Figure 9: Diagram of the Forward Controller

The forward controller is a module that is used to control what are the values used as rs1 and rs2 in the ALU. For example, if you have a data dependency between two instructions, the first one is a load and the second one is an add, you need to forward the result of the load to the ALU. This module is responsible for that and instead of using the value of the register file, it will use the value that is being forwarded.

Signals:

- Input: reg_addr1, This signal is representing the first register address that is being used by the current instruction.

- Input: reg_rs1, This signal is representing the value of the first register that is being used by the current instruction.
- Input: reg_addr2, This signal is representing the second register address that is being used by the current instruction.
- Input: reg_rs2, This signal is representing the value of the second register that is being used by the current instruction.
- Input: *id_ex_reg_wr_addr*, This signal is representing the register address that is being written by the previous instruction.
- Input: *id_ex_reg_wr_sig*, This signal is representing if the previous instruction is written to the register file or not.
- Input: *id_ex_data_dest*, This signal is representing the origin of the data that is being used by the ALU in the previous instruction.
- Input: *id_ex_alu_result*, This signal is representing the result of the ALU in the previous instruction.
- Input: *id_ex_pc_plus*4, This signal is representing the pc plus 4 of the previous instruction.
- Input: *ex_mem_reg_wr_addr*, This signal is representing the register address that is being written by the previous instruction.
- Input: *ex_mem_reg_wr_sig*, This signal is representing if the previous instruction is written to the register file or not.
- Input: *ex_mem_data_dest*, This signal is representing the origin of the data that is being used by the ALU in the previous instruction.
- Input: *ex_mem_alu_result*, This signal is representing the result of the ALU in the previous instruction.
- Input: *ex_mem_pc_plus*4, This signal is representing the pc plus 4 of the previous instruction.
- Input: *ex_mem_mem_rd_data*, This signal is representing the data that is being read from the memory in the previous instruction.
- Input: *mem_wb_reg_wr_addr*, This signal is representing the register address that is being written by the previous instruction.
- Input: *mem_wb_reg_wr_sig*, This signal is representing if the previous instruction is written to the register file or not.
- Input: *mem_wb_data_dest*, This signal is representing the origin of the data that is being used by the ALU in the previous instruction.

- Input: *mem_wb_alu_result*, This signal is representing the result of the ALU in the previous instruction.
- Input: *mem_wb_pc_plus*4, This signal is representing the pc plus 4 of the previous instruction.
- Input: *mem_wb_mem_rd_data*, This signal is representing the data that is being read from the memory in the previous instruction.
- Output: *rs*1, This signal is representing the value that should be used as rs1 in the ALU.
- Output: *rs*2, This signal is representing the value that should be used as rs2 in the ALU.

3.3.2.3 Stall Unit

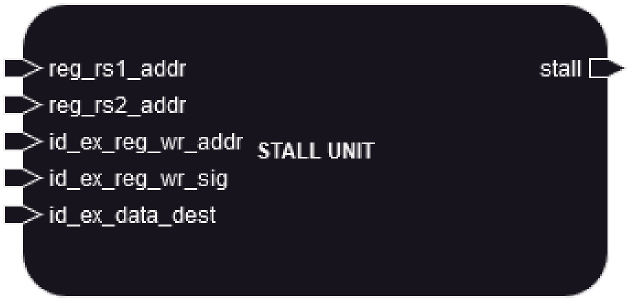

Figure 10: Diagram of the Stall Unit

The stall unit is a small module that is used to stall the pipeline when a data dependency that cannot be resolved by forwarding is detected which should only happen if we use the result of a load instruction in the next instruction. It is simply comparing the register addresses of the current instruction with the register that is being written by the previous instruction. If there is a match, it looks what is the operation that is being executed by the current instruction and if it is a load, it will stall the pipeline.

Signals:

- Input: *reg_rs*1*_addr*, This signal is representing the first register address that is being used by the current instruction.
- Input: *reg_rs*2*_addr*, This signal is representing the second register address that is being used by the current instruction.
- Input: *id_ex_reg_wr_addr*, This signal is representing the register address that is being written by the previous instruction.
- Input: *id_ex_reg_wr_sig*, This signal is representing if the previous instruction is written to the register file or not. It is used to differentiate between a load and a store.

- Input: *id_ex_data_dest*, This signal is representing the origin of the data that is being used by the ALU in the previous instruction. In this case only if the previous instruction has a data dest of MEM then we need to stall the pipeline.
- Output: *stall*, This signal is representing if we need to stall the pipeline or not.

3.3.2.4 2 Entries Muxes

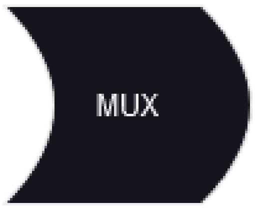

Figure 11: Diagram of the 2 Entries Mux

The mux is a small module that is used to select between the two inputs that it has. It is used 2 times in this module, for selecting between $rs1$ and PC and for selecting between $rs2$ and imm. Sorry, I haven't used the standard shape of a MUX but I haven't found one on the website I'm using to draw the different diagrams.

Signals:

- Input: a, This signal is representing the first input of the mux.
- Input: b, This signal is representing the second input of the mux.
- Input: *sel*, This signal is representing the select signal of the mux.
- Output: *out*, This signal is representing the output of the mux.

3.3.3 EX Stage

3.3.3.1 Arithmetic Logic Unit

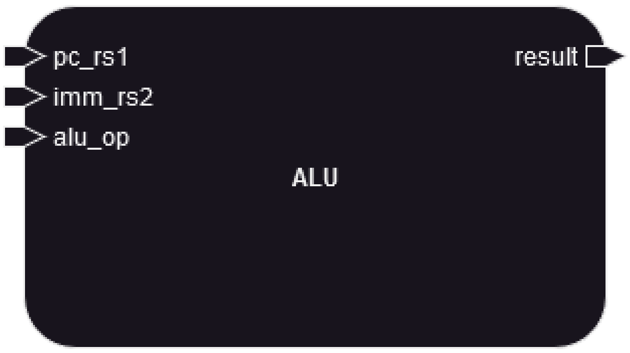

Figure 12: Diagram of the ALU

The ALU is a module that is responsible for executing the arithmetic and logic operations. So every mathematical operation is done in this module.

Signals:

- Input: *a*, This signal is representing the first input of the ALU.
- Input: *b*, This signal is representing the second input of the ALU.
- Input: *op*, This signal is representing the operation that the ALU will execute.
- Output: *out*, This signal is representing the output of the ALU. So for example the ADD, SUB etc.

3.3.3.2 Branch Unit

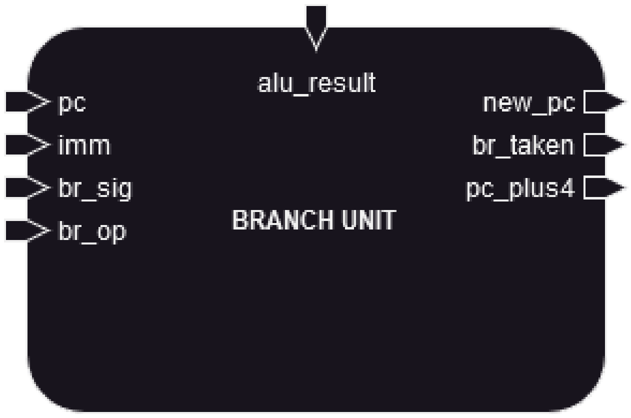

Figure 13: Diagram of the Branch Unit

The Branch Unit is a module that is responsible for computing the next PC. It is used in the EX stage. It uses the result of the ALU for conditional branching to know if the branch should be taken or not.

Signals:

- Input: *alu_result*, This signal is representing the result of the ALU. Used for conditional branching.
- Input: *pc*, This signal is representing the current PC.
- Input: *imm*, This signal is representing the immediate value of the instruction. It is used as an offset for the branch.
- Input: *branch_sig*, This signal mark if the instruction is a branch or not.
- Input: *br_op*, This signal is representing the branch operation like for example BEQ, BNE etc.
- Output: *new_pc*, This signal is representing the new PC computed by the branch unit.

- Output: *pc_plus_four*, This signal is representing the PC + 4. Used to save the value of the next instruction to come back at it after a branch occurred.
- Output: *br_taken*, This signal is representing if the branch is taken or not. Used in the IF stage to compare to the branch predictor and know if the pipeline needs to be flushed or not.

3.3.4 MEM Stage

3.3.4.1 Load Store Unit

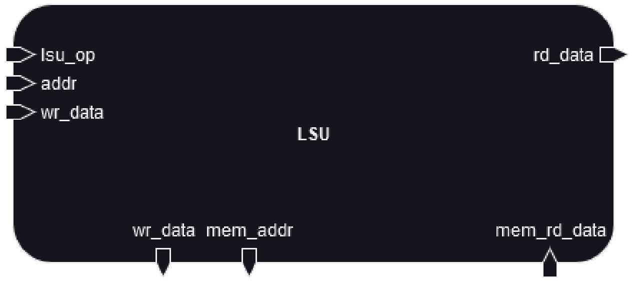

Figure 14: Load Store Unit

The LSU is a module that is responsible for memory access. It will manage the different READ and WRITE to the memory and for example mask the data you want to read or write according to the given instruction.

Signals:

- Input: *lsu_op*, This signal is representing the type of memory access you want to perform.
- Input: *addr*, This signal is representing the address of the memory you want to access.
- Input: *wr_data*, This signal is representing the data you want to write to the memory.
- Input: *mem_rd_data*, This signal is representing the value that has been read from the memory.
- Output: *wr_data*, This signal is representing the value that should be written to the memory and correctly masked according to the given instruction.
- Output: *mem_addr*, This signal is representing the address you want to write or read to the memory.
- Output: *rd_data*, This signal represents the value that has been read from the memory and correctly masked according to the given instruction.

3.3.5 WB Stage

3.3.5.1 3 Entries Mux

Figure 15: Diagram of the 3 Entries Mux

The mux is a small module that is used to select between the three inputs that it has. In this case, it is used to select between the Branch Unit, the ALU, and the LSU result and then write it back to the register file.

Signals:

- Input: *alu_out*, This signal is representing the output of the ALU.
- Input: *lsu_out*, This signal is representing the output of the LSU.
- Input: *pc_plus*4, This signal is representing the output of the Branch Unit.
- Input: *data_dest*, This signal is representing the select signal of the mux.
- Output: *wb_data*, This signal is representing the data that will be written back to the register file.

3.3.6 Pipeline Registers

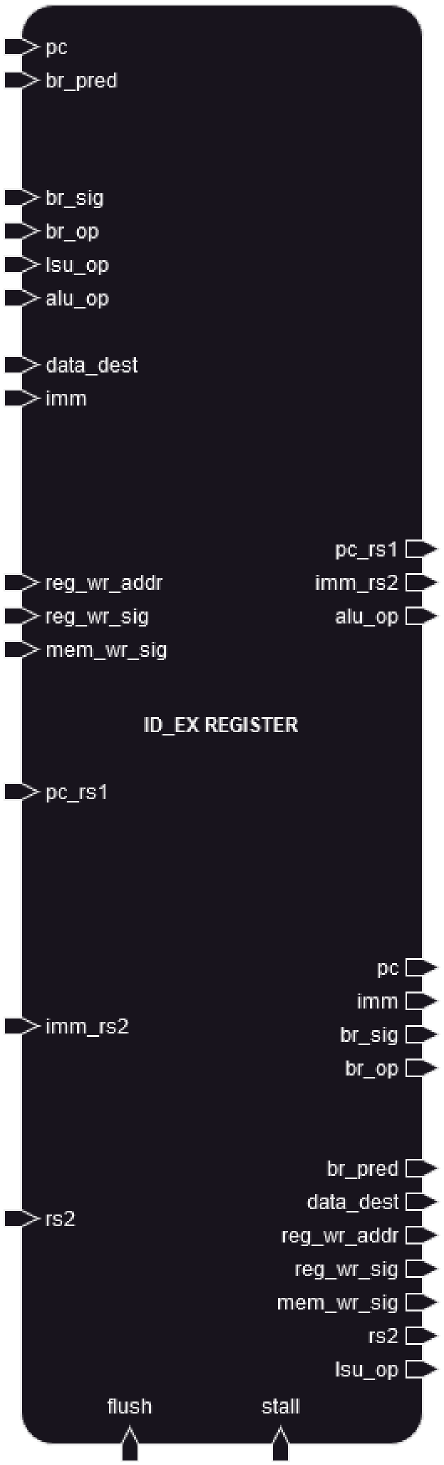

Figure 16: ID-EX Pipeline register example

I will not explain the 4 different pipeline register stages in detail, and I will mainly describe as an example the pipeline between the ID Stage and the Ex Stage. The role of a pipeline register is to store the data that will be passed to the next stage at each clock cycle such that you can increase throughput by increasing the clock frequency since you've divided the overall work in smaller chunks so you can do it in less time (theoretically). What is also interesting here is the *stall* and *flush* signals. The first one is used to stop updating the pipeline when a data dependency is encountered. The second one is used when the branch predictor did a bad guess. In this case we need to flush the register that has incorrect instructions stored in them and instead just fill them with something similar to a NOP instruction.

3.3.7 ROM, RAM and Register File

3.3.7.1 ROM

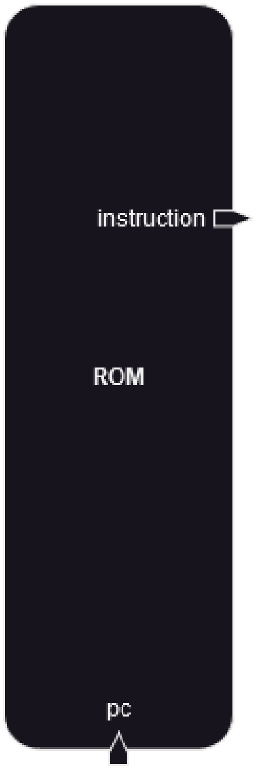

Figure 17: ROM

The ROM is responsible for storing the program that is being executed by the processor. As its name indicate, it is a read-only memory and for the moment it is capable of storing up-to 1024 instructions.

Signals:

- Input: *pc*, This signal gives the address that needs to be read in the memory.
- Output: *instruction*, This signal is representing the instruction that has been read from the ROM.

3.3.7.2 RAM

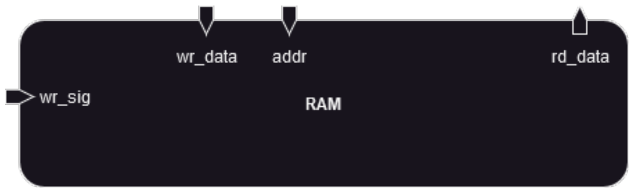

Figure 18: RAM

The RAM is the main memory of the CPU, it can be used to store and read values. In this project, the RAM is also 1024 words long and each word is 32 bits long.
Signals:

- Input: *addr*, This signal gives the address that needs to be read or written in the memory.
- Input: *wr_sig*, This signal indicates if the given address need to be written or read
- Input: *wr_data*, This signal gives the data that needs to be written in the memory.
- Output: *rd_data*, This signal is representing the data that has been read from the RAM.

3.3.7.3 Register File

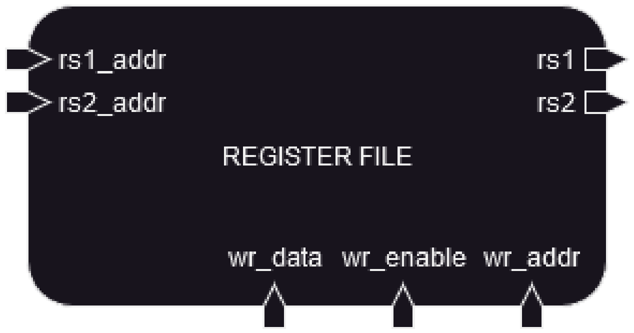

Figure 19: Register File

The register file is a one-cycle really small memory. The RV32I standard requires 32 registers, each 32 bits long. Only the first (zero) register has a given value of 0, the other can be used at will but as with any other architecture there are some standards for what a register is being used for.
Signals:

- Input: *rs*1*_addr*, This signal gives the address of the first register that needs to be read.

- Input: *rs2_addr*, This signal gives the address of the second register that needs to be read.
- Input: *wr_addr*, This signal gives the address of the register that needs to be written.
- Input: *wr_data*, This signal gives the data that needs to be written in the memory.
- Input: *wr_enable*, This signal indicates whether the register should be written with the *wr_data* value.
- Output: *rs1*, This signal gives the value read at address *rs1_addr*.
- Output: *rs2*, This signal gives the value read at address *rs2_addr*.

4 Testing

There is some testbench that has been written for the biggest module of the project. The only notable exception is the controller for time but also since it is mostly tested by the general CPU testbench. So here is the testbench that has been written:

- ALU
- Branch Unit
- CPU
- Forward controller
- LSU
- PC
- Register File
- Stall Unit

Most of this testbench is testing the module with a lot of different (mostly random) inputs and checking that the output is correct. Some are also testing more specific cases like corner cases that could lead to some issues in the design. The testbench for the CPU is a bit special since it is testing the whole CPU with a simple program computing a recursive sum of the first 10 integers testing most of the module of the CPU. When writing the test I discovered some issues in the design, mostly in the ALU and the Branch Unit. For in the ALU I was shifting not correctly in one of my instructions, and in the Branch Unit the output of the alu wasn't treated as a signed number. So I've fixed. It's also with the CPU testbench that I've discovered the issue with the different values between Questa Modelsim with full visibility and Icarus Verilog. I still think that more tests could be added to the different testbenches to have even better coverage but I think that the current state is pretty okay.

References

[1] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. 2017. URL: `https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf` (visited on 06/11/2023).

[2] John Winans. *RISC-V Assembly Language Programming*. 2022. URL: `https://github.com/johnwinans/rvalp/releases/download/v0.18.1/rvalp.pdf` (visited on 06/11/2023).

[3] University of Washington. *Forwarding*. URL: `https://courses.cs.washington.edu/courses/cse378/07au/lectures/L12-Forwarding.pdf` (visited on 06/11/2023).

[4] Tse-Yu Yeh and Yale N. Patt. *Two-Level Adaptive Training Branch Prediction*. 1991. URL: `https://www.inf.pucrs.br/~calazans/graduate/SDAC/saltos.pdf` (visited on 06/11/2023).

[5] Chipverify. *Verilog Tutorial*. URL: `https://www.chipverify.com/verilog/verilog-tutorial` (visited on 06/11/2023).

[6] IEEE Computer Society. *IEEE Standard for Verilog Hardware Description Language*. 2005. URL: `http://staff.ustc.edu.cn/~songch/download/IEEE.1364-2005.pdf` (visited on 06/11/2023).

[7] Stephen Williams. *Getting Started With Icarus Verilog*. 2023. URL: `https://steveicarus.github.io/iverilog/usage/getting_started.html` (visited on 06/11/2023).

[8] Don Dennis. *RISCV-RV32I-Assembler*. Apr. 15, 2019. URL: `https://github.com/metastableB/RISCV-RV32I-Assembler` (visited on 06/11/2023).

[9] Lucas Teske. *RISC-V Online Assembler*. URL: `https://riscvasm.lucasteske.dev/#` (visited on 06/11/2023).

[10] Davis LupLab @ University of California. *RISC-V Instruction Encoder/Decoder*. 2023. URL: `https://luplab.gitlab.io/rvcodecjs/` (visited on 06/11/2023).

[11] RISC-V Foundation. *RISC-V GNU Compiler Toolchain*. URL: `https://github.com/riscv-collab/riscv-gnu-toolchain`.

[12] Vivonomicon. *Bare-metal RISC-V Development with the GD32VF103CB*. Feb. 11, 2019. URL: `https://vivonomicon.com/2020/02/11/bare-metal-risc-v-development-with-the-gd32vf103cb/`.

A Detailed Pipelined CPU Design

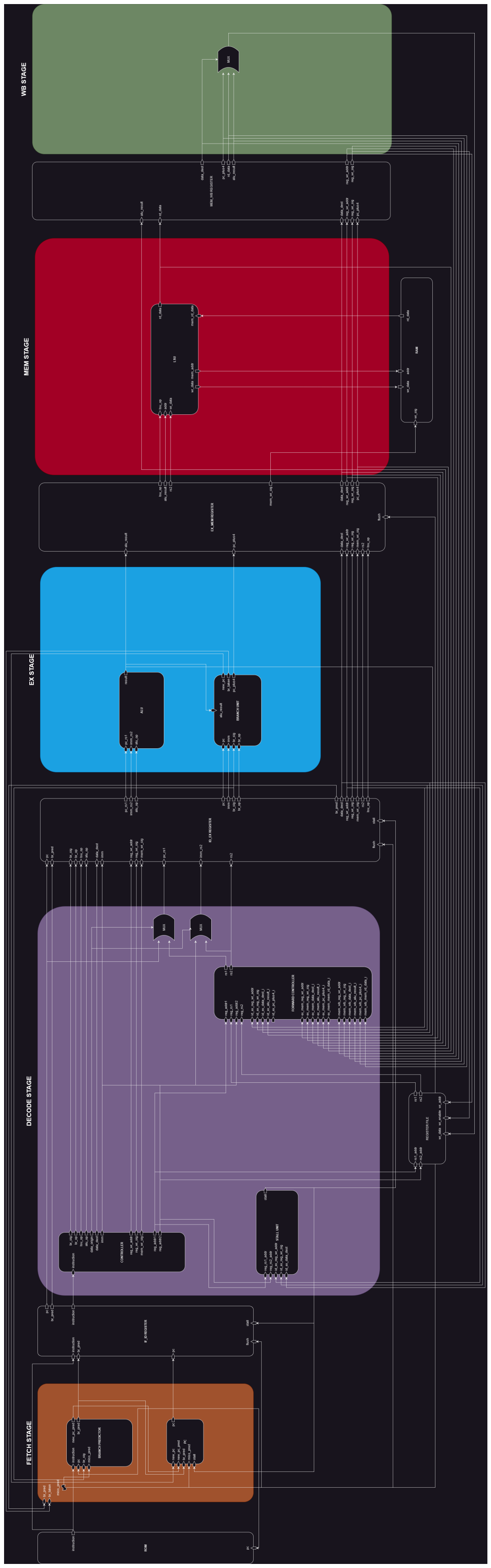

Figure 20: Detailed Pipelined CPU Design