RISC-V: A Didactic Platform

Documentation

Jules PERRIN, Professor: Theo Kluter

June 12, 2023

Contents

1 Introduction

This document describes the RV32IM processor and its architecture. I've tried to make it as clear as possible such that everyone with a basic knowledge of computer architecture can understand it. It will describe each significant module of the processor, what it does and what are the different inputs and outputs with an image of the module. To avoid repeating too much the signal, I will only list the signals directly linked to the module, that's why most of the stages don't have any signals listed because it's their inside modules that are using them and will be listed here instead.

It is recommended to follow this document with the source code of the processor, which is available in the Verilog folder of the project. Don't hesitate to use the table of contents to navigate through the different parts of the document. If you see any mistake or are confused by something don't hesitate to contact me at the mail address jules.perrin@epfl.ch.

2 Parameters

All the different constants were used in the project such that the different values linked to the different operators for the LSU, BRU and ALU. It is not an ideal solution but since Verilog doesn't have a better way to share constants by for example doing enums, I decided to do it this way. So all the constants are in the parameters.vh file inside the Verilog directory.

3 Fetch

3.1 IF Stage

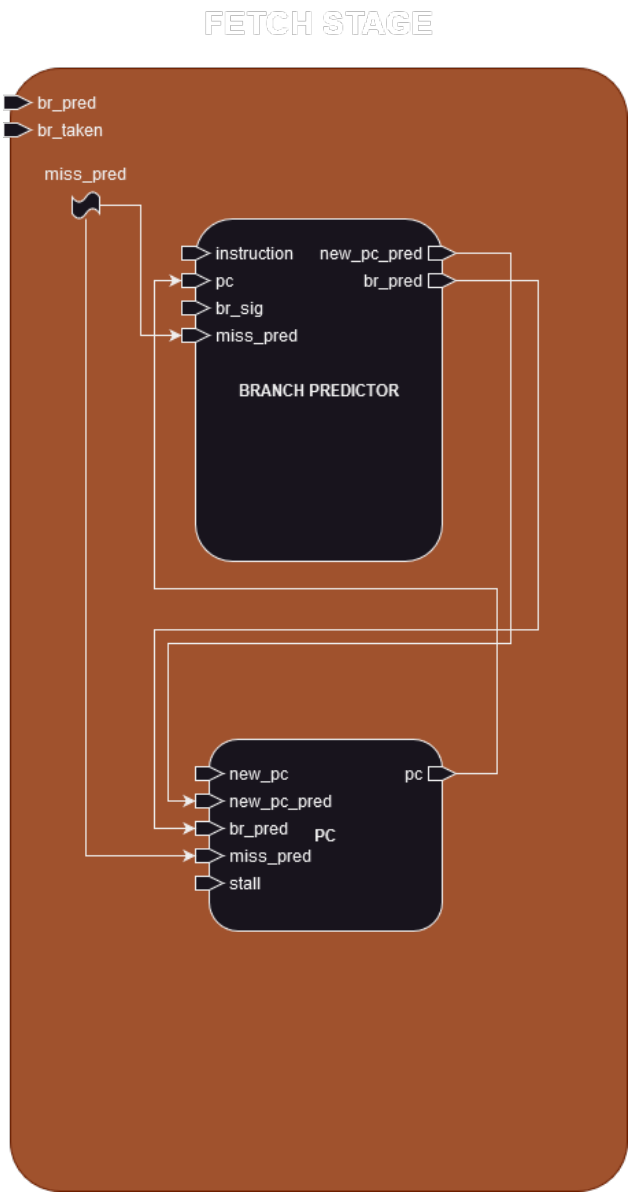

Figure 1: Diagram of the IF Stage

The IF stage is responsible for fetching the next instruction from the ROM and passing it to the ID stage. The IF stage contains two different modules: the PC and the branch predictor. The IF stage is the only stage that computes a value outside of a value, it uses the *br_pred* signal and the *br_taken* signal to compute if there is a miss prediction such that the PC and the branch predictor can be updated accordingly.

Signals:

- Input: *br_pred*, This signal is representing the state of the prediction made by the branch predictor in the EX stage.
- Input: *br_taken*, This signal is representing the state of the branch in the EX stage.
- Output: *miss_pred*, This signal is representing if there is a miss prediction or not.

3.2 Branch Predictor

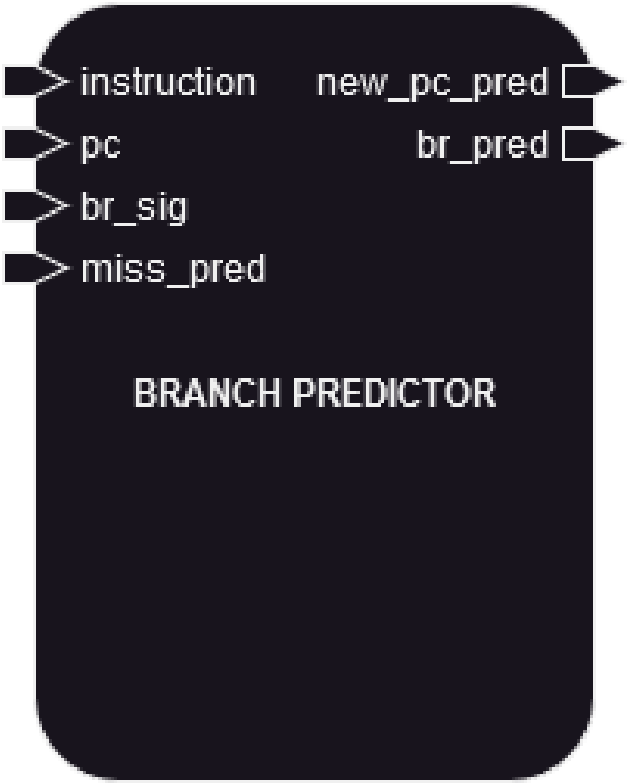

Figure 2: Diagram of the Branch Predictor

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, 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 PC

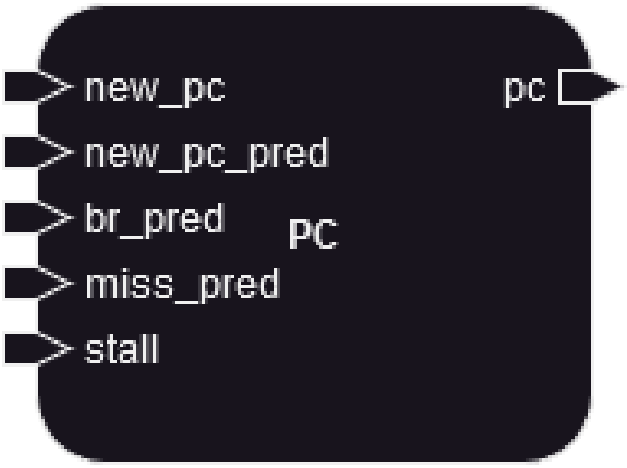

Figure 3: Diagram of the PC

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 represent the next PC given by the branch unit in the EX stage.
- Input: *new_pc_pred*, This signal represent 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.

4 Decode

4.1 ID Stage

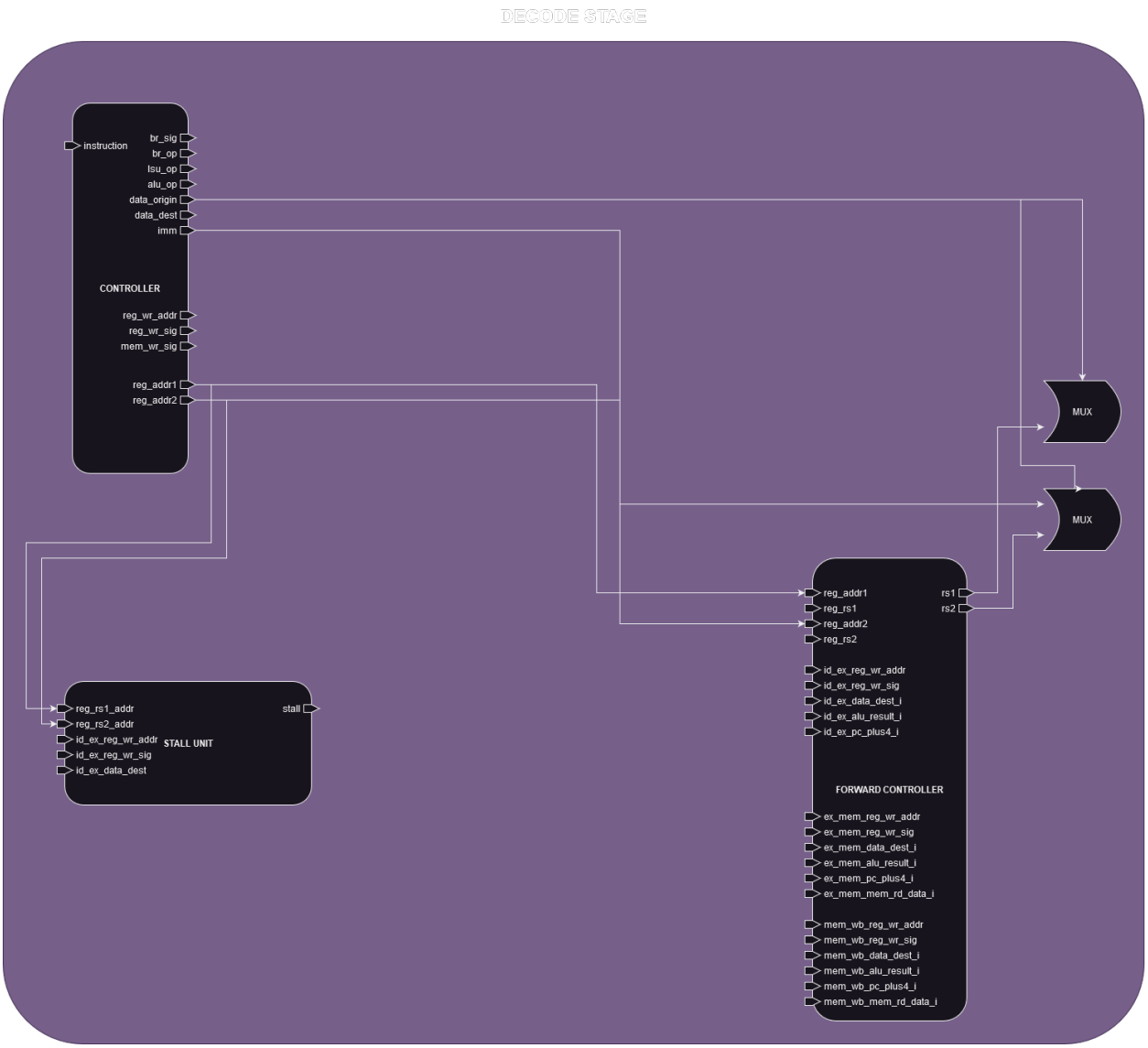

Figure 4: Diagram of the ID stage

The ID stage is the second stage of the pipeline. It is responsible for decoding the instruction and forwarding the data to the next stage. It is composed of the following modules: the controller which is the main module of the stage, the stall unit, the forward controller and 2 smaller modules which are 2 muxes used to select between the different registers and the immediate value or the pc.

4.2 Controller

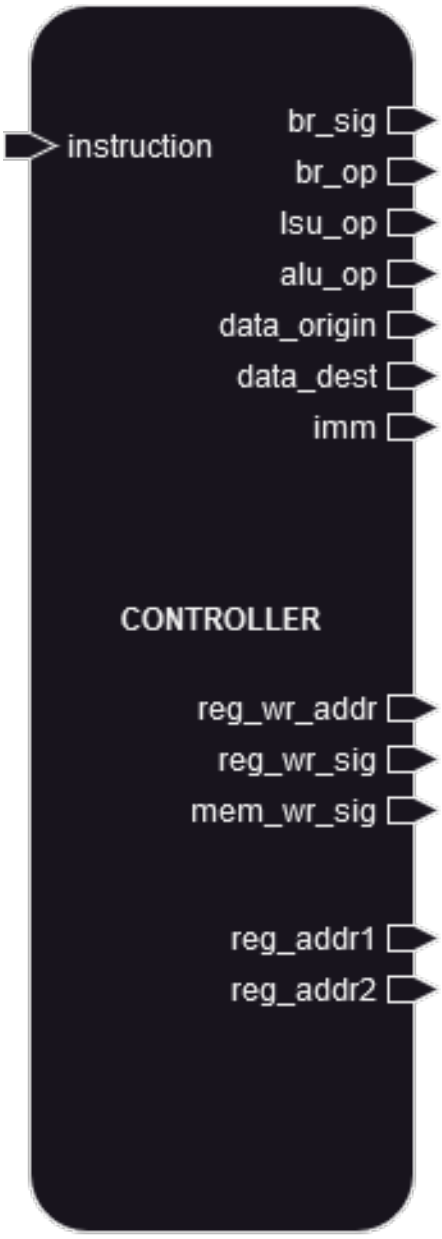

Figure 5: Diagram of the Controller

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. For more information about the different types of instruction and how the data is encoded in the instruction, please refer to the RISC-V manual [1].

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.

4.3 Stall Unit

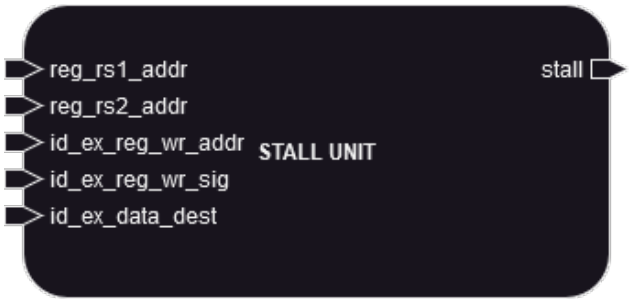

Figure 6: Diagram of the Stall Unit

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_rs1_addr, This signal is representing the first register address that is being used by the current instruction.
- Input: reg_rs2_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.

4.4 Forward Controller

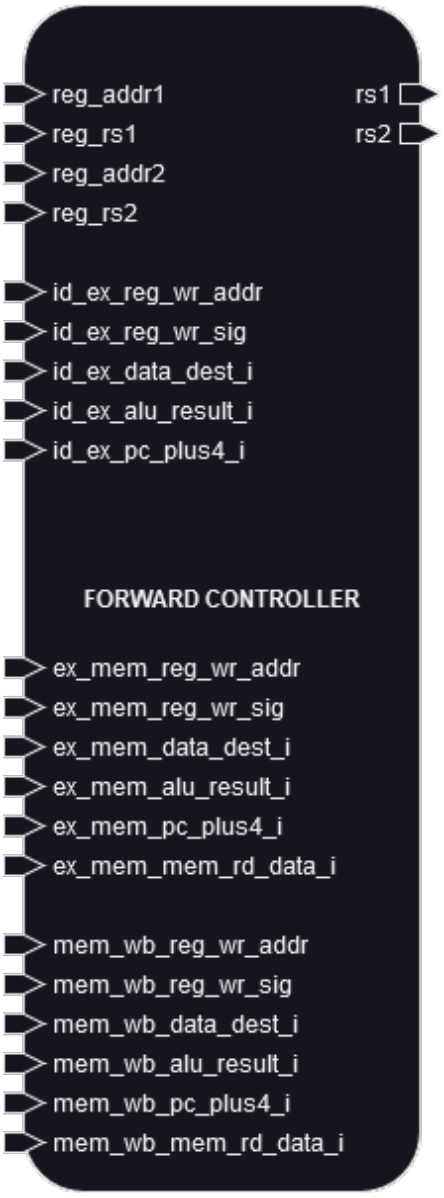

Figure 7: Diagram of the Forward Controller

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: $mem\_wb\_pc\_plus4$, 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: $rs1$, This signal is representing the value that should be used as rs1 in the ALU.
- Output: $rs2$, This signal is representing the value that should be used as rs2 in the ALU.

4.5 Mux

Figure 8: 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.

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.

5 Execute

5.1 EX Stage

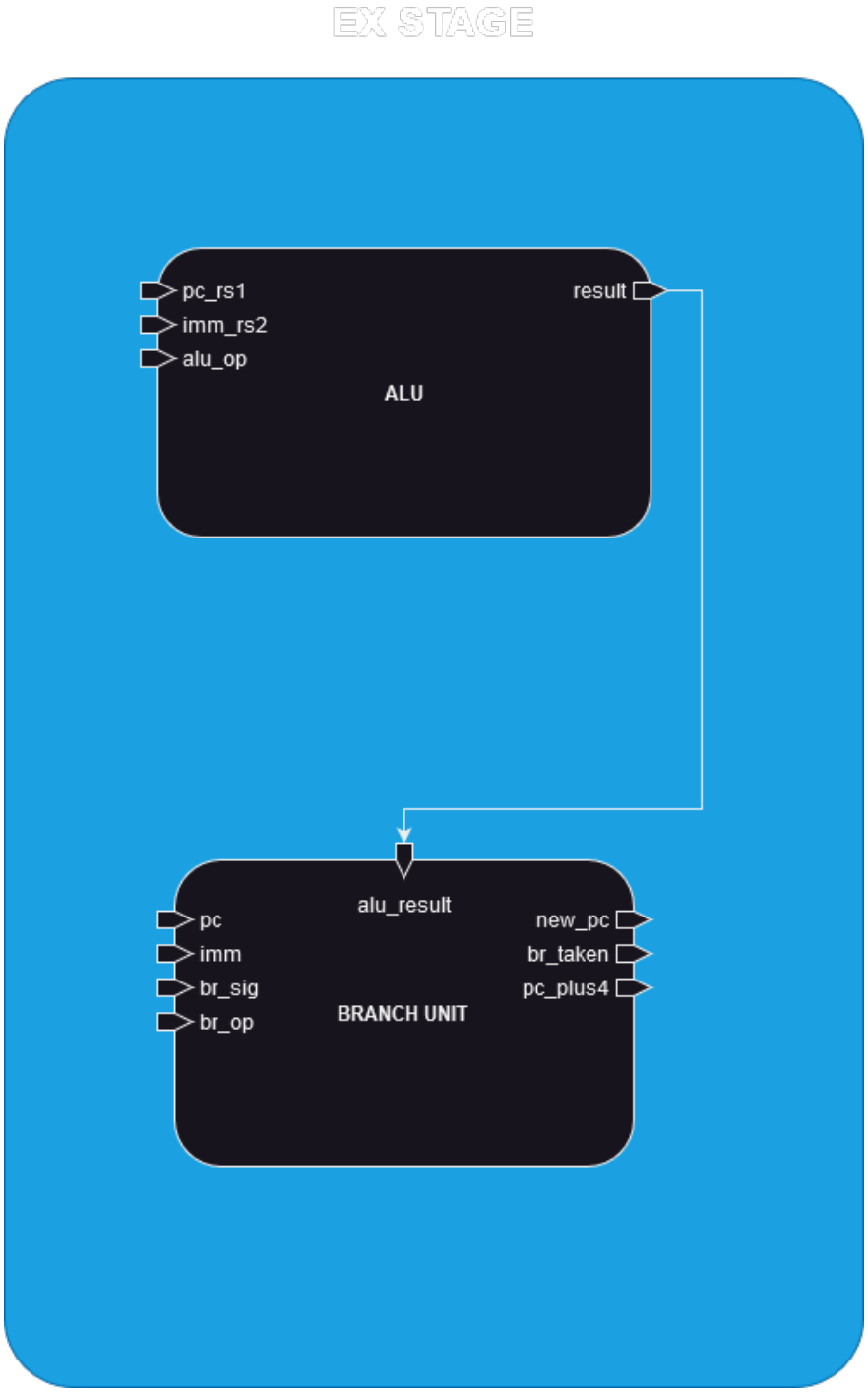

Figure 9: Diagram of the EX stage

The EX stage is the third stage of the pipeline. It is responsible for executing the instruction and forwarding the data to the next stage. It has two main modules inside of it, the ALU and the Branch Unit. The ALU is responsible for executing the arithmetic and logic operations and the Branch Unit is responsible for computing the next PC.

5.2 Arithmetic Logic Unit

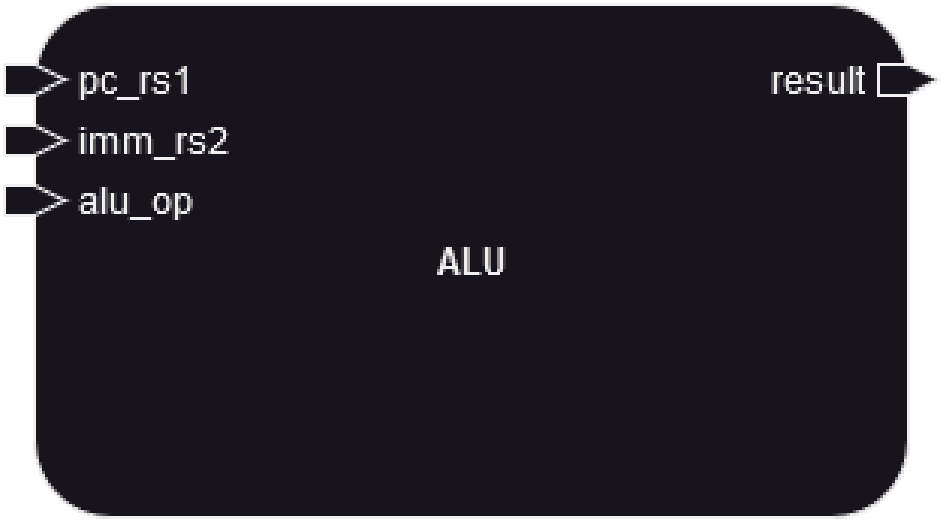

Figure 10: Diagram of the ALU

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.

5.3 Branch Unit

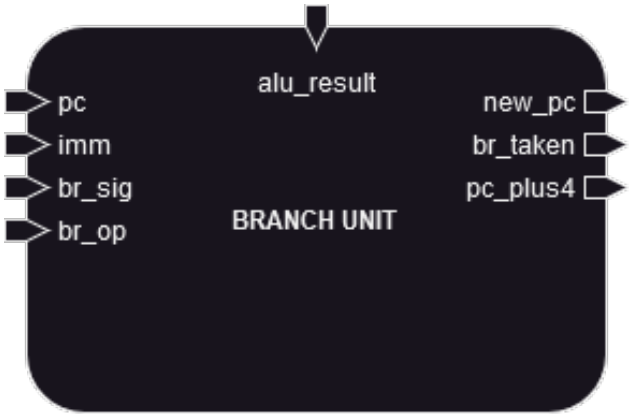

Figure 11: Diagram of the Branch Unit

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.

6 Memory

6.1 Memory Stage

Figure 12: Memory stage

The memory stage is responsible for the memory access (RAM) and managing the different READ and WRITE to it. It contains only one module called the LSU (Load Store Unit) that manage the different interaction with the memory.

6.2 Load Store Unit

Figure 13: 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.

7 Write Back

7.1 Write Back Stage

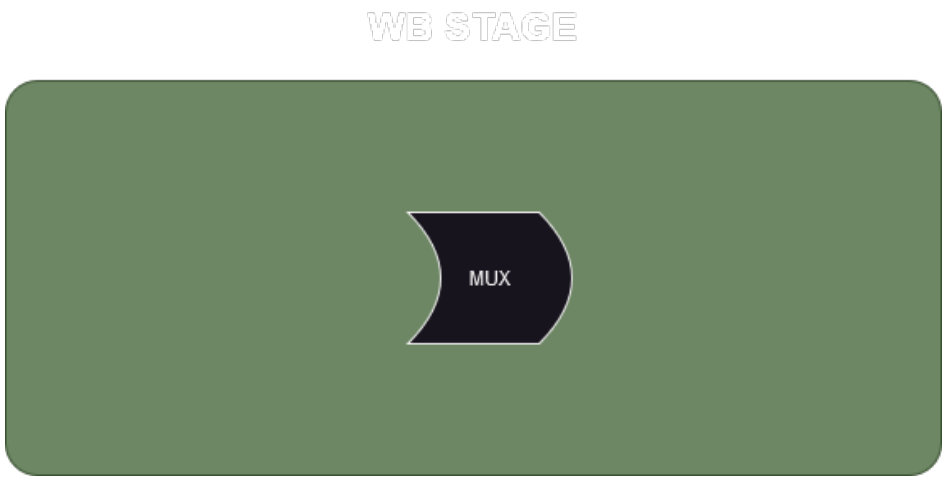

Figure 14: Write Back stage

The writeback stage is the fifth and last stage of the pipeline. It is the simpler stage that contains only one module, a simple 3 entry mux that choose which data between the Branch Unit, the ALU and the LSU will be written back to the register file.

7.2 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.

8 Pipeline Registers

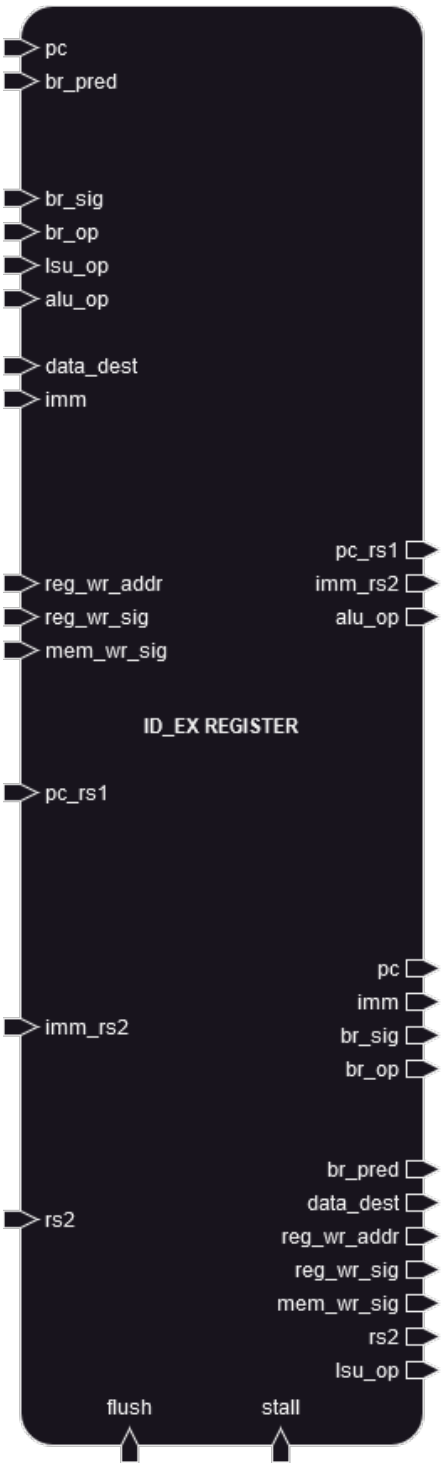

Figure 16: ID-EX Pipeline register example

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.

9 ROM, RAM and Register File

9.1 ROM

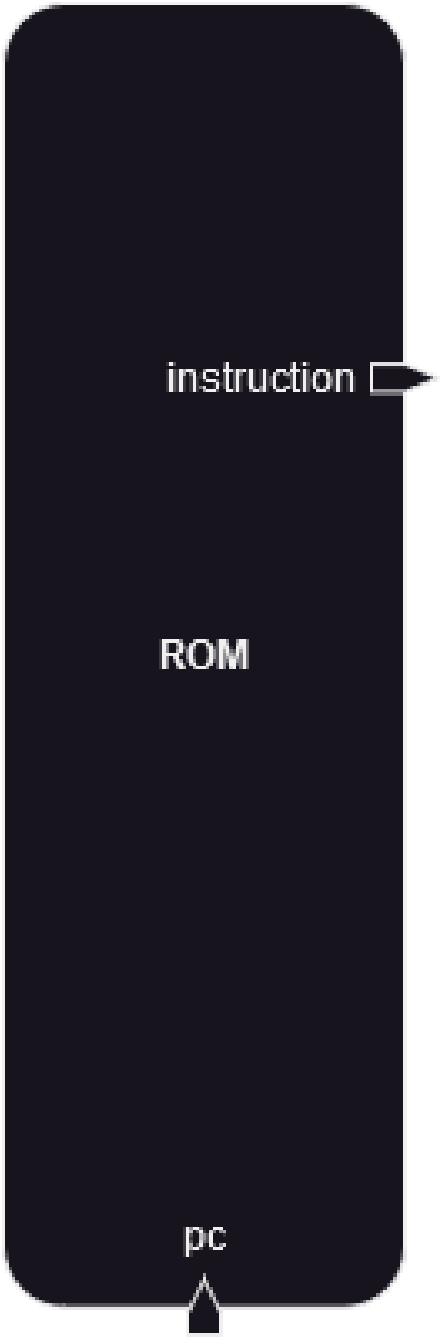

Figure 17: ROM

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.

9.2 RAM

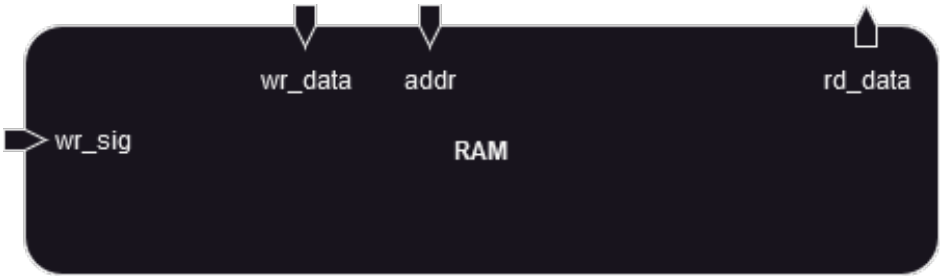

Figure 18: RAM

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.

9.3 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: $rs1\_addr$, 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 $rs2\_addr$.

10 Toolchain

I've also put as disposition an "easy" way to generate programs to use as input to the ROM of the processor. You still have to install the RISC-V toolchain to use it and configure correctly the bash or zsh file to indicate where the toolchain is installed. Installing the toolchain can take quite a while since you'll have to compile it and can take more than 45min and will highly depend on what CPU you have.

Here is a step-by-step guide to installing the toolchain:

- Clone the toolchain from here.

```
git clone https://github.com/riscv/riscv-gnu-toolchain.git
```

- Navigate into the cloned directory.

```
cd riscv-gnu-toolchain
```

- Run the configuration script.

```
./configure --prefix=/opt/riscv --with-arch=rv32im
```

- Compile the toolchain. This step may take a while.

```
make
```

- Once the compilation is done, add the toolchain to your PATH in your bash or zsh configuration file.

```
echo 'export PATH=$PATH:/opt/riscv/bin' >> ~/.bashrc
source ~/.bashrc
```

After that, you should be able to use the toolchain. To test it, you can try making a simple program like the ones in the *c_code* folder of the project. You can simply compile them with the following command:

```
make
```

That will generate a .hex file that you can use as input to the ROM of the processor. It will also give you the assembly code of the program in the .s file so that you can see what the compiler generated.

11 Running Tests

Running and creating new tests is pretty easy to do, normally if you create new verilog files in the Verilog folder or new tests in tb folder, it will automatically add it to the compiling process. But how to run the tests?

You have to simply install iverilog first using the command on Ubuntu or any Debian based distribution:

```
sudo apt-get install iverilog -y
```

Then goes to the tb folder and run the command:

```
make
```

It should output the result in the terminal and provide you with a .vcd file that you can open with gtkwave to see the waveform. Of course in the case of you added new tests, you have to add the fact of creating the .vcd file in the testbench file.

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#chapter.2 (visited on 06/11/2023).