Project 1: femtoRV32

Milestone 3. Pipelined Version

CSCE 3301 - Computer Architecture (Spring 2023)

The American University in Cairo

Amer Elsheikh **900196010**

Gehad Ahmed **900205068**

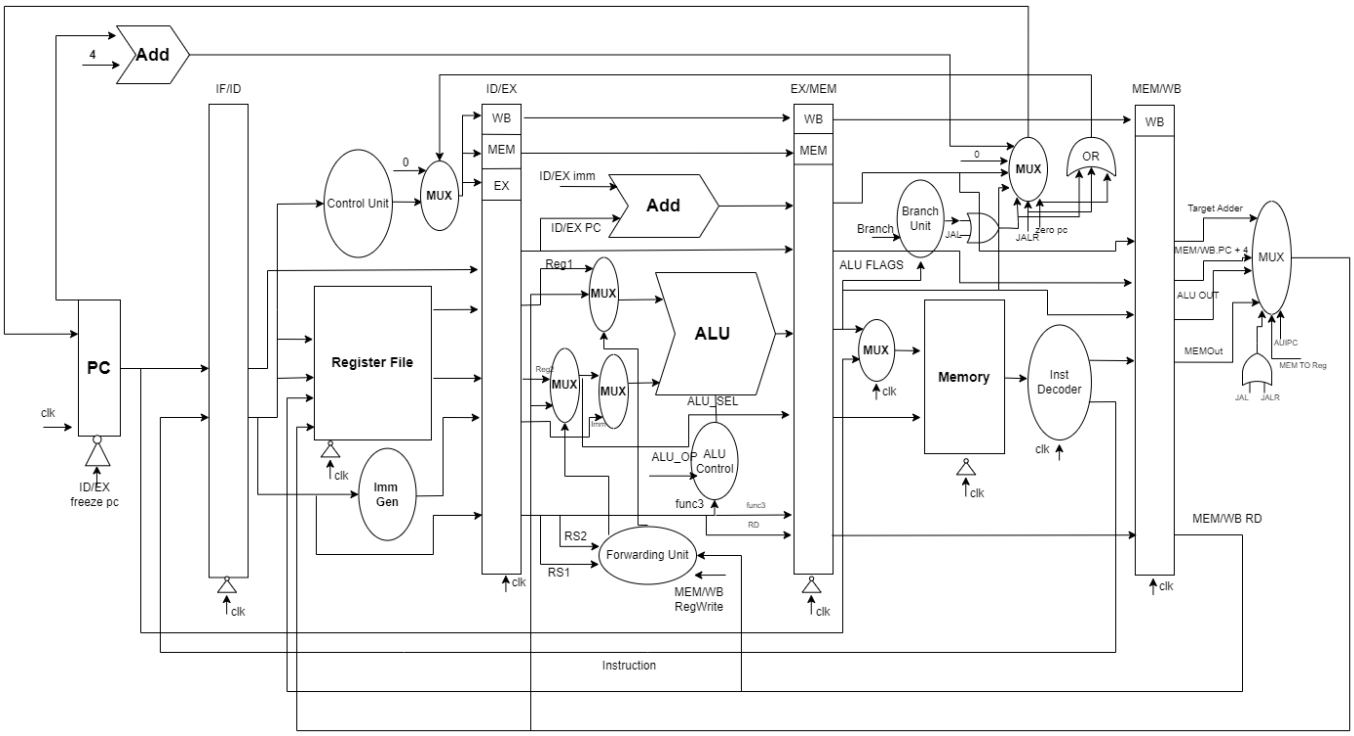
Dr. Cherif Salama

Architecture:

We began by dividing the single-cycle datapath we developed into a pipelined 3 stages:

- First Stage: Instruction Fetch and Instruction Decode
- Second Stage: Execution and Memory
- Third Stage: Write back.

We drew and built the datapath below. You can find the datapath image in the datapath folder.



Implementation:

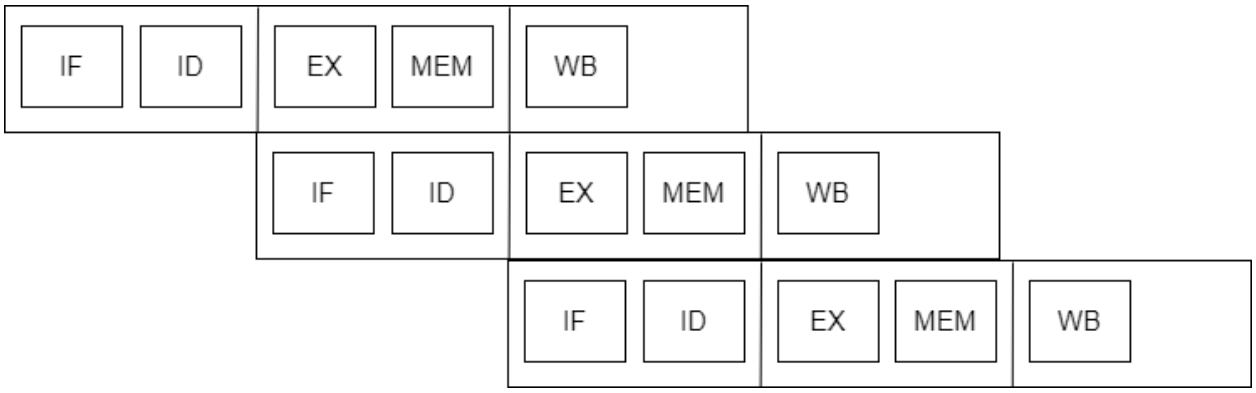
We discuss the modules we changed, and the modifications we did to it in comparison to the single cycle processor we made in the first stage:

Pipelining:

As seen in the architecture, we added 4 big registers as in the normal 5-stage pipeline. But in order to handle having one memory, we merged every two stages together by adjusting the clock as follows:

- PC updates at the **positive** edge of the clock
- IF/ID updates at the **negative** edge of the clock.
- ID/EX updates at the **positive** edge of the clock
- EX/MEM updates at the **negative** edge of the clock.
- MEM/WB updates at the **positive** edge of the clock
- Regfile updates at the **negative** edge of the clock.

This even changing of the clock allows us to do every two consecutive stages of the old 5-stage pipeline into 1 new stage in our new 3-stage pipeline as follows:



The purpose of that is to guarantee that the memory(MEM) stage never happens at the same time of the instruction fetching (IF) stage in order for the processor to have a single ported memory instead of two separate memories for the data and the instructions. We will elaborate on the memory itself later.

As seen in the architecture, at every half-stage, we see what we will need in the upcoming cycles regarding the information we currently have and we pass it through the intermediate registers.

Single Memory:

We have a byte addressable 4KB memory, single ported memory for both data and instructions. To initialize it, we used the “readmemh” command to read the instructions from “program1.mem”, and then add the data starting at index 500, and we consider the memory as an instruction memory when the clk is 1 and data memory otherwise.

For our test generation, we wrote tests in assembly then assembled it to the hex format. After this, we made a simple C++ script to take the hex file and divide each instruction into 4 lines, each containing 2 bytes, to adhere with the little indianity of RISC V.

Now, our memory has to support different load and store instructions (byte, half, word, byteU, HalfU), and for this we have the func3 input to know which one is selected, and In case of loading byte or halfword, we do sign extension.

We make both memory sizes small (200 bytes) when we synthesize on the FPGA. Otherwise, the implementation fails.

Forwarding Unit:

A pipelined processor, although much more efficient, introduces a lot of hazards and complexities that we must handle. The first of which are the data and load use hazards.

To handle these, we first notice that the new 3-stage pipeline only has data hazards between only adjacent instructions as the instruction before the adjacent instruction writes back before the current instruction executes. Also, it allows us to handle data use hazards as normal RAW hazards.

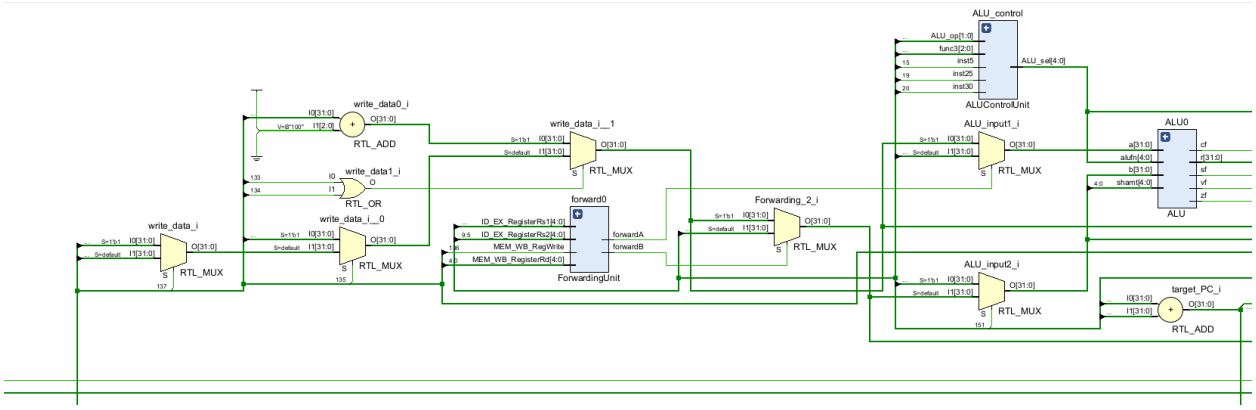
To handle both, we developed the forwarding unit which takes MEM_WB_RegWrite, ID_EX_RegisterRs1, ID_EX_RegisterRs2, MEM_WB_RegisterRd as inputs and produce the following bit outputs:

- forwardA: It is set if RS1 has data dependency with the instruction before it.
- forwardB: It is set if RS2 has data dependency with the instruction before it.

In the top module, we create a MUX that chooses between the data from register RS1 or the data from the WB stage based on forwadA and put it to ALU_INPUT1

We also make another MUX that chooses between the data from register RS2 or the data from the WB stage based on forwadB. However, another MUX is put after that to choose between the output of the first MUX and the immediate to be put to ALU_INPUT2

We do this to pass the output of the first MUX to the EX/MEM register to be used later for memory addressing.



Flushing Unneeded Instructions:

The normal act for the PC is to increase 4 every cycle bu three other cases may happen as shown in the architecture:

- new_PC may be set to 0 if the instructions are Fence and ECALL
- new_PC may be set to the result of the target_PC adder if there is a successful branch or if the instruction is JAL.
- new_PC may be set to the ALU output if the instruction is JALR.

We know this at the MEM stage as seen by the PC MUX in our datapath. However, when we know that, a new instruction is already in the pipeline in the [IF, ID] stages, and we need to stop it from continuing and fetch the right instruction instead.

To do that, we just do an OR to the signals that cause the above 3 situations and the result, call it jump_pc, we pass it to a MUX before the ID/EX register. If jump_pc is set, we put **zeros** instead of the control signals in the ID/EX register. That has the effect of canceling this instruction.

You can find the whole schematic and a zoom_in to the different parts inside the schematic folder.

Testing:

We created several tests to be able to run our processor on all instructions.
In the tests folder, we included every test in a separate folder where each folder contains the following files:

- assembly.txt: has the assembly code for this test program.
- memory.txt: contains all the needed data to be added to the memory to be used.
- program.hex: contains the hexadecimal representation of our program.
- program.mem: this is the memory file for the instructions to be used in our processor, where the instructions are written in a byte addressable format.
- Simulation.png: is a screenshot of the working simulation used for testing the program.

Test 1 (Basic Program): This is the program that was previously used in the lab, to test that the previously supported instructions weren't corrupted.

Test 2 (Fibonacci): This a program to test fibonacci of the 8th term (which is 21), and it was executed correctly.

Test 3 (Rest of Instructions): This is a program with simple testing to all the other untested instructions to ensure they get executed properly.

Test 4 (Multiplication): This is a program with simple testing to all the multiplication and division instructions in the RV-32M standard extension.

In this table, we are highlighting the instructions tested in each test.

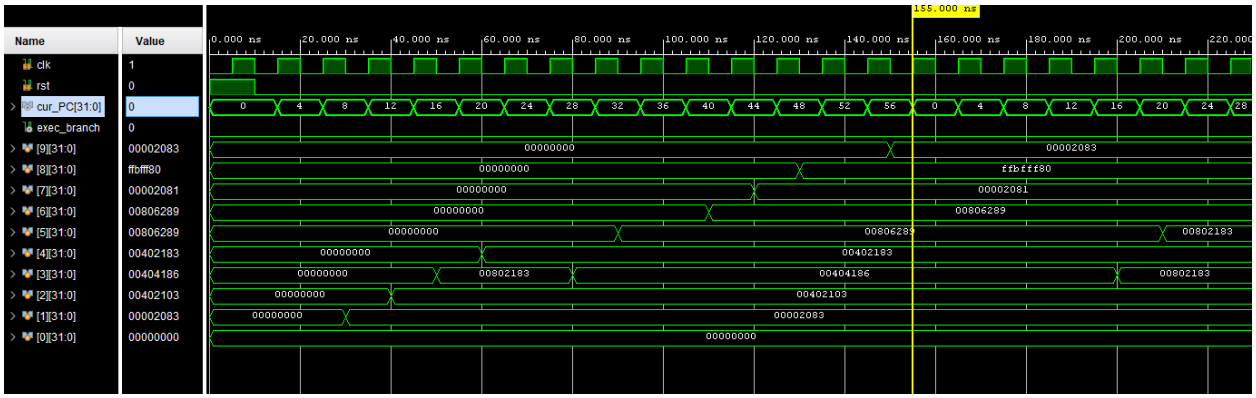
LUI	BNE	LB	SB	SLTIU	SRLI	SLT	OR
AUIPC	BLT	LH	SH	XORI	SRAI	SLTU	AND
JAL	BGE	LW	SW	ORI	ADD	XOR	FENCE
JALR	BLTU	LBU	ADDI	ANDI	SUB	SRL	ECALL
BEQ	BGEU	LHU	SLTI	SLLI	SLL	SRA	EBREAK

And here are the instructions in the RV-32M standard extension.

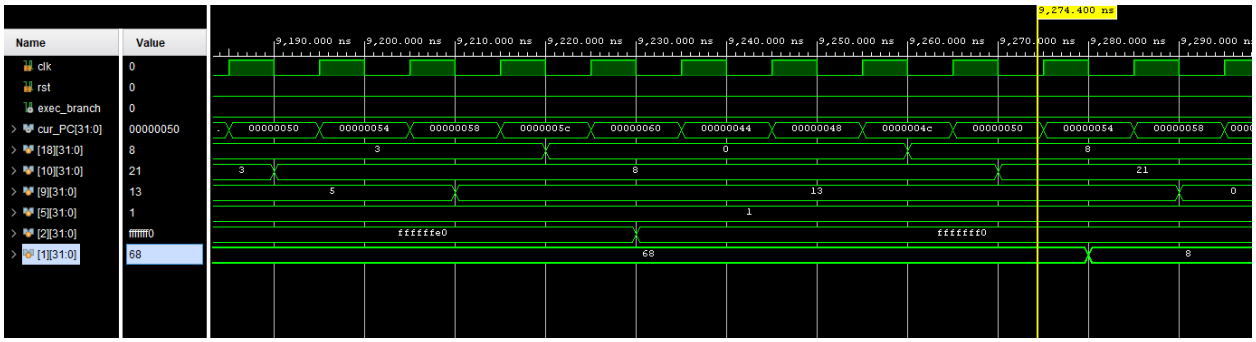
MUL	MULH	MULHSU	MULHU	DIV	DIVU	REM	REMU

And as illustrated above, **all 48 instructions have been tested and they are all working correctly**.

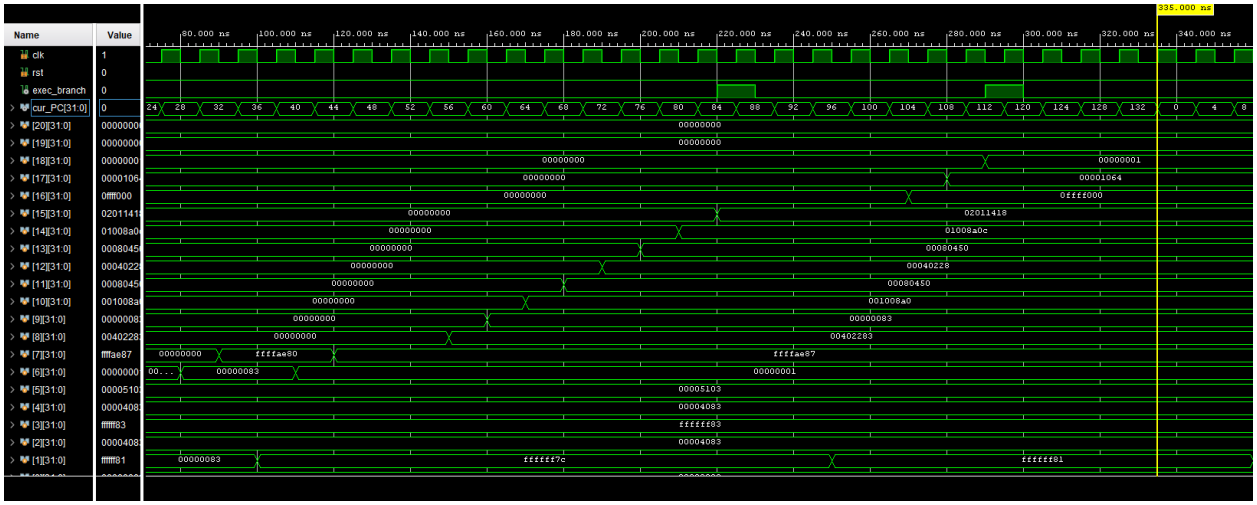
Screenshots of the simulation output of the three tests:



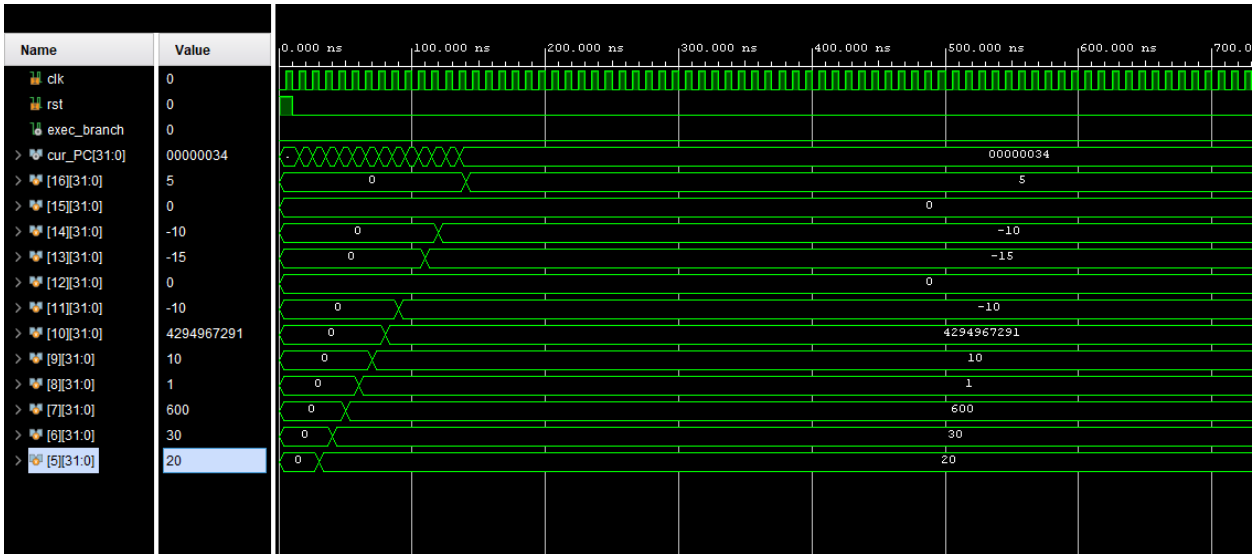
Test 1



Test 2 (In reg[10], we has 21 for the 8th fibonacci number)



Test 3



Test 4

What Works

All instructions with the multiplication extension are working fine, and tested with corner cases. The Project synthesized successfully.

What Does not Work

The program synthesized and ran on the FPGA but the FPGA only shows zeros for the write back.

Bonus

We implemented the following bonus:

- Add support for integer multiplication and division to effectively support the full RV32IM instruction set.
- Partial support for the FPGA, the project synthesize with no error but with wrong output.