ECE 485 - Computer Organization and Design
MIPS Pipeline CPU Design and Implementation

Alex Maliwat , Hazim Zain-Edin

Abstract

This project is the culmination of all lectures and skills gained over the course of this semester. The final project goal is to design and implement a 32 bit RISC architecture inspired by the MIPS processor. For completion, the ISA should be able to handle the ISA given by Table 1 in the assignment sheet, namely, R-types, load and stores, add immediate, jumps, and a new instruction called ‘Add1’. The processor will be tested by vhdl testbenches in Vivado on the ECE Endeavour server.

Introduction and System Design

Introduction

The objective of the implementation of the processor is to understand the pipeline model described in lecture. Each of the typical 5 stages must be implemented in line with the instruction set architecture. Each component and each stage will be created from scratch. The pipeline datapath will be modified from Figure 1 in the assignment sheet (shown below) to implement jump and add1. Below, each stage will be quickly described and modifications from the given pipeline will be highlighted.

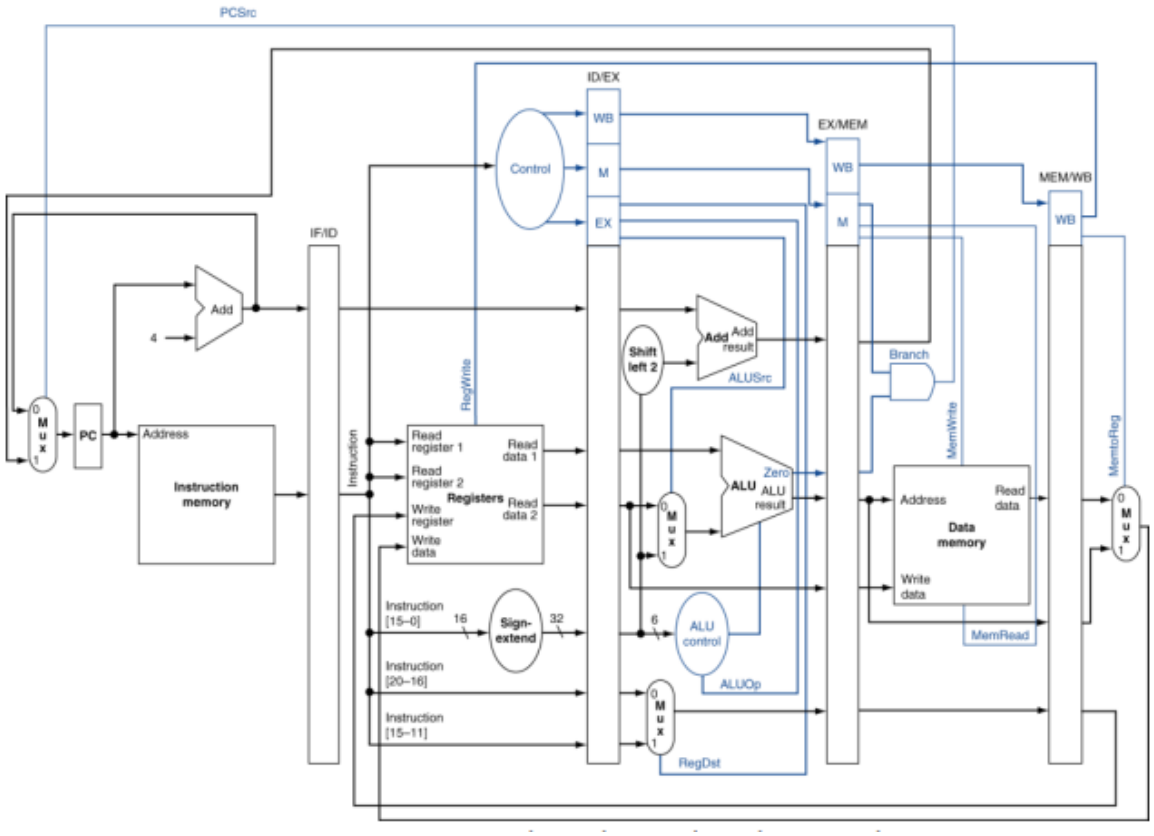

Figure 1: given pipeline to be modified

System Design and Modifications

Instruction fetch:

First will be the instruction fetch (IF) stage. This section encompasses the PC selection and instruction memory. The only modification to be made was that the '1' source on the PCSrc mux is no longer the branch address and is now the jump address. This stage will fetch the instruction at the PC from the instruction memory and will send the 32 bit instruction value to the buffer.

In our pipeline, we selected not to increment this PC in this stage. That will be done in the EX stage.

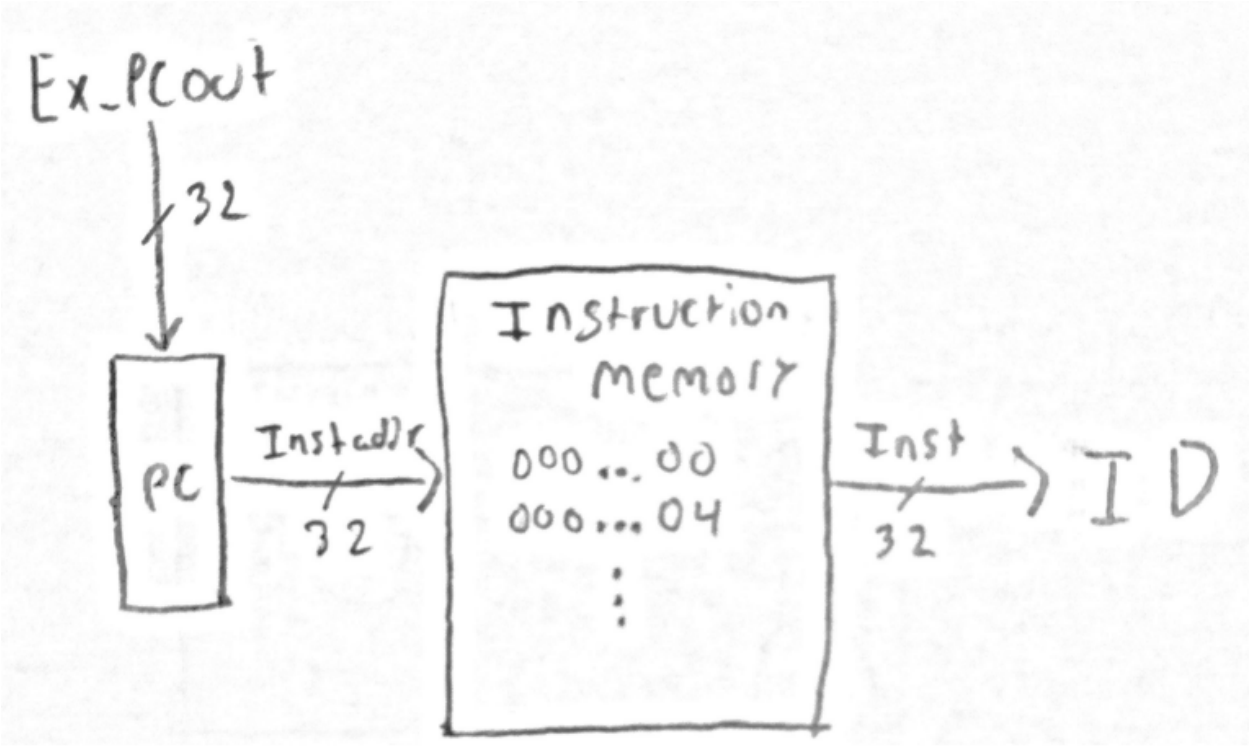

Figure 2: Datapath for the Instruction Fetch stage

Instruction decode:

The instruction decode (ID) stage manages the decoding of the 32 bit instruction from the IF stage into the necessary parts for the rest of the processor. From top to bottom in Figure 1:

- The control unit takes in the opcode given by the first 6 bits [31:26] of each instruction and decodes it into each stage control signals needed in the rest of the pipeline. Here is where the type of instruction is determined.

- The register file first writes the write data and the write register information and updates itself in the first half of the cycle (if indicated by the RegWrite signal). In the second half, the register file uses the 5 bit long fields used for R-type instructions and decodes them into the register numbers. It then outputs the 2 read values in those registers to the buffer.
- The sign extender takes in what is the immediate value for the I-type instruction (and the funct code for R-type) and extends it to 32 bits into the buffer.
- RT and RD are put into the buffer for later use for write selection.

Control Logic:

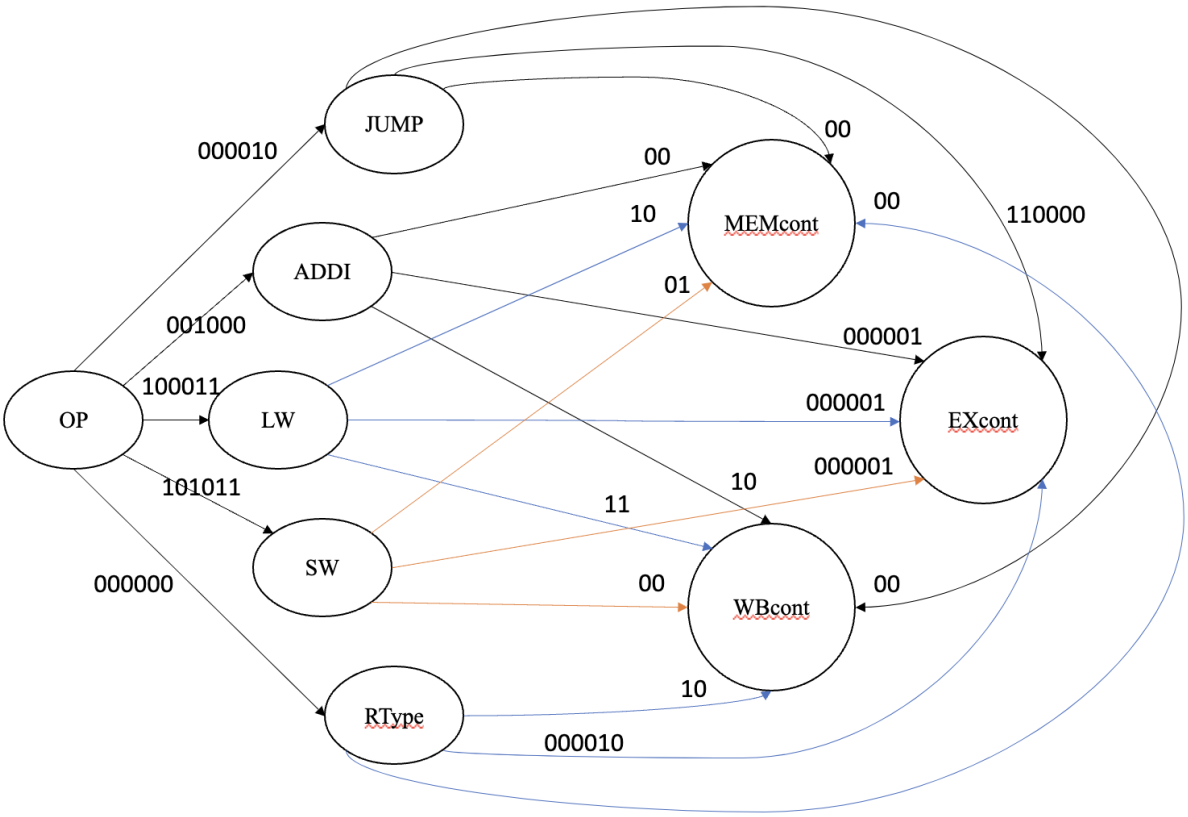

Figure 2: Control Unit FSM and respective control signals

Instruction	OP	RS	RT	RD	SHAMT	FUNCT
sub $s2, $s1, $s3	000000	10001	10011	10010	00000	100010
and $t2, $s2, $t5	000000	01101	01010	01010	00000	100100
or $t2, $s0, $t2	000000	10000	01010	01010	00000	100101

add1 $s3, $t1, $t0	001000	01001	01000	10011	00000	100000
lw $t3, 100($s2)	100011	10010	01011	0000000001100100		
addi $s4, $t3, 200	001000	01011	10100	0000000011001000		
sw $t1, 100($t2)	101011	01010	01001	0000000001100100		
nor $t1, $s1, $t0	000000	10001	01000	01001	00000	100111
slt $t1, $s2, $s2	000000	10010	10010	01001	00000	101010
j $2500	000010	00000000000000100111000100				
	OP	Shift	Immediate Values			

Table 3: Instruction decode and bits

The control unit is very easily broken down into its main components. Its main objective is to send the respective control signals to the specified modules, like select bits in muxs or enables for the ALU. The signal imputed to the control unit originates from the instruction decode phase where an opcode is passed in. This code is then associated a type of command, in the MIPS architecture we have R, I, and J type commands. In table 3 you can see the instruction decoded to the OP code which is then sent over to the control unit. Seen in figure 2 the OP code then is linked to its family of commands. Once linked it then assigns respective values to a second level of what you can call control vectors. Specifically the Execution Controller, Memory Controller, and Write Back Controller. A breakdown of each respective values significance can be seen in table 4 below, and they are explained in detail through the report.

EX(0) = ALUsrc	EX(1) = REGdst	EX(2-3) = ALUOP	EX(4) = JumpEn	EX(5) = PCsrc
MEM(0) = Write	MEM(1) = Read			
WB(0) = toREG	WB(1) = Write			

Table 4: Control Unit Outputs Decoded

R-Type instructions: As seen in figure 2 these will always receive a ‘000000’ for the OP code. The corresponding control signals are as follows: The reg_write signal will be set to 1 which will signal the Registers Unit that the result of the computation done at the ALU will be written back to the register specified by the write_reg input. The alu_src signal will be set to 0 which selects the data from read_register_2 (denoted by the Rt field of the Rt) as the input to the ALU. The reg_dst signal will be set to 1 which selects inst[15:11] as the write_reg input for the Register Unit, this corresponds with a t or s register. The mem_read and mem_write signals will both be

set to 0 as there will be no read or write operations done at the Data Memory Unit, yet the writeback_register will be 1 since its updating that value. Finally, the APUOP signal will be set to 00 in binary which signals the ALU Control Unit to use the funct field of the instruction to determine which operation to perform.

J-Type instructions: We wher tasked with creating lw, sw, addi, and jump, with OpCodes of 100011, 101011, 001000, and 000010 respectively. When passed to the control unit, the corresponding control signals are as follows: The reg_write signal will be set to 1 in the lw and addi cases which will signal the Registers Unit that the result of the computation done at the ALU will be written back to the register specified by the value at the write_reg input. For the sw case, the reg_write signal will be set to 0 as we intend on writing to memory. For all three cases, the ALUsrc signal will be set to 1 due to the processors need to select data from the signextened vector. The regdst signal will be set to 0 which selects inst[20:16] as the write_reg input for the Register Unit. For lw, the mem_read signal will be set to 1 to signal the Data Memory Unit to take the data located at the index specified by the alu_result input and output it to read_data. On the other hand, the mem_read signal will be set to 0 for the sw and addi instructions as we are not loading any value from memory. The mem_write signal for the lw and addi instructions will be set to 0 as we are not storing any data in memory. The mem_to_reg signal will be set to 0 for the lw instruction which signals the multiplexer to take the output from read_data and route it back to the write_data input of the Registers Unit. For the addi instruction, the mem_to_reg signal will be set to 1 which signals the multiplexer to take the output from the alu_result and route it back to the write_data input of the Register Unit. In the sw case, the mem_to_reg signal is “don’t care”. At the same time all the values of the jump control unit will be “dont care” with the exception of the jump_enable.

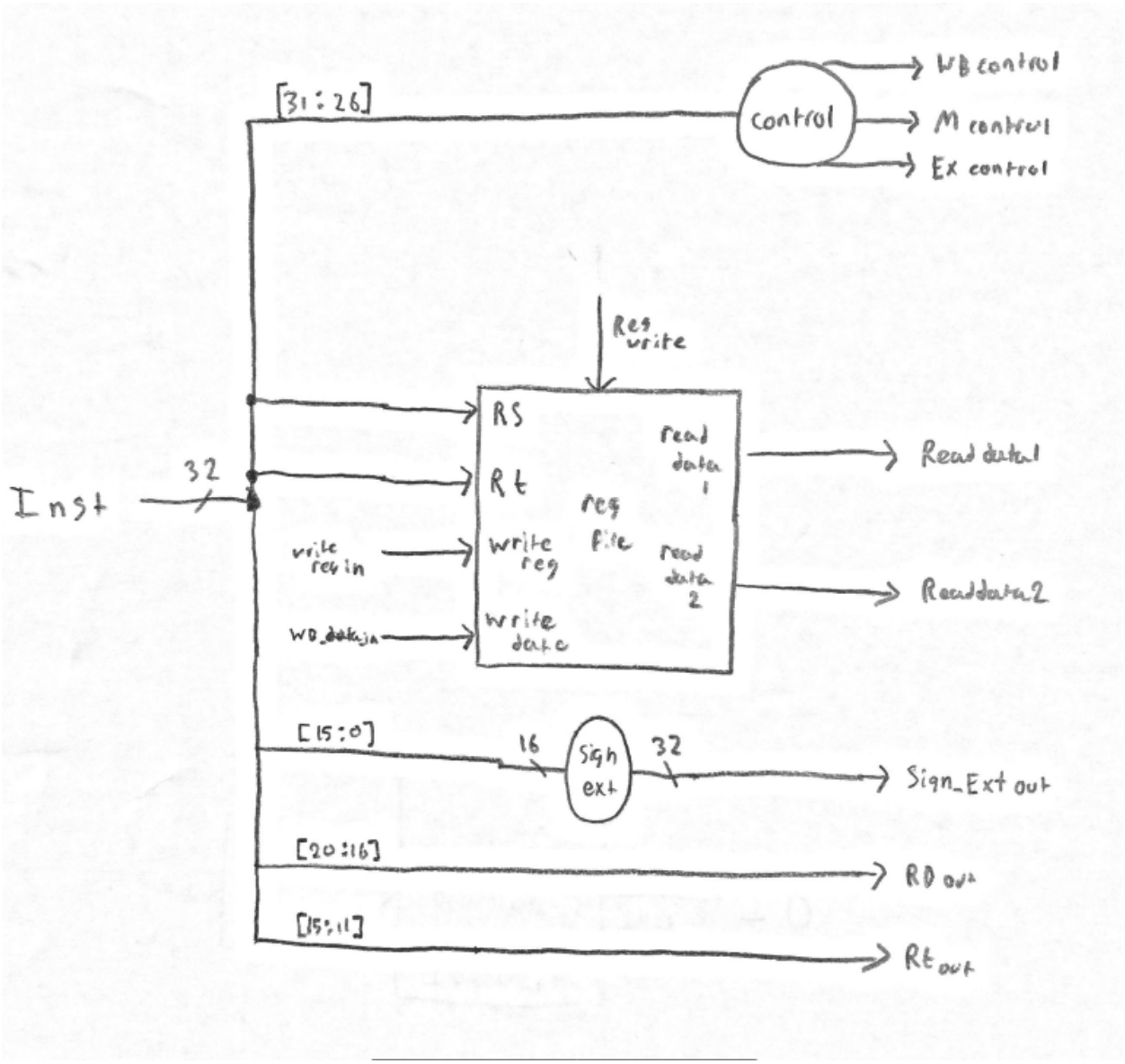

Figure 5: Instruction Decode Datapath

Execute:

The EX stage is where the majority of the computation takes place. This stage holds the ALU, where everything is computed. It also is where selection of the ALU sources and operations are. The jump address and PC increment takes place here. Finally, the writeback destination register is selected and passed into the buffer.

The EX stage is very simple, it takes in the values from the control unit discussed above in the instruction decode stage, then at the same time it inputs the sign_extend where the function code

or the immediate value depending on the instruction. It ustilizes that along with the ALU op to process and output an ALU_Control signal. This is then used alongside the register values into the ALU for the particular instruction set.

Here, modifications are made to the ALU to be able to increment the regs given by Add1 and also increment the PC+4 in order to process the next instruction. The branch calculation is also taken out here, as it is not required for the given ISA.

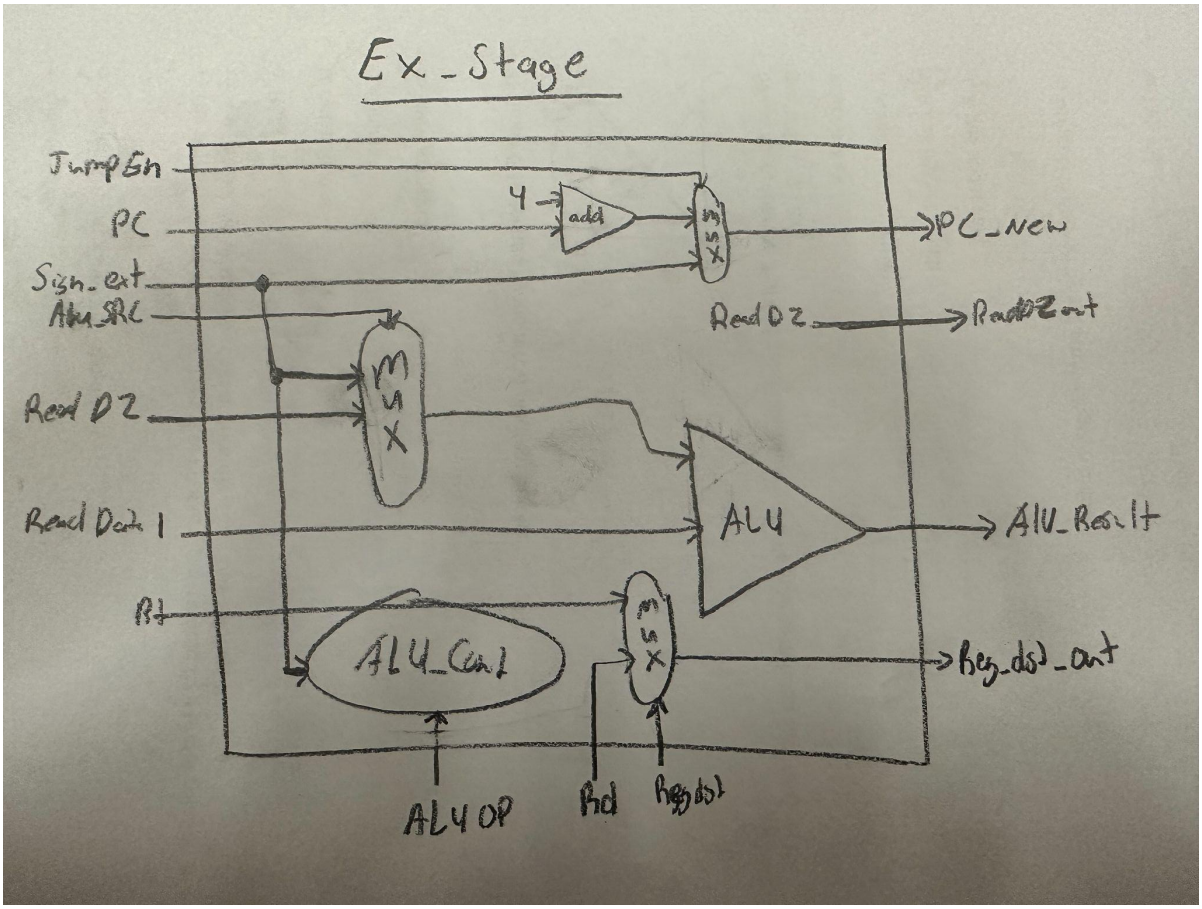

Figure 6: EX Stage Datapath

Memory Access:

The MEM stage deals with the data memory reads and writes. Using the MemWrite and MemRead control signals, it decides whether to deal with the sw or lw instructions with the given ALU output and write data from the read data from the regfile. Very simple design and it was created with an arbitrary amount of storage this is only for hypothetical purposes and would have a real quantity in a realistic situation.

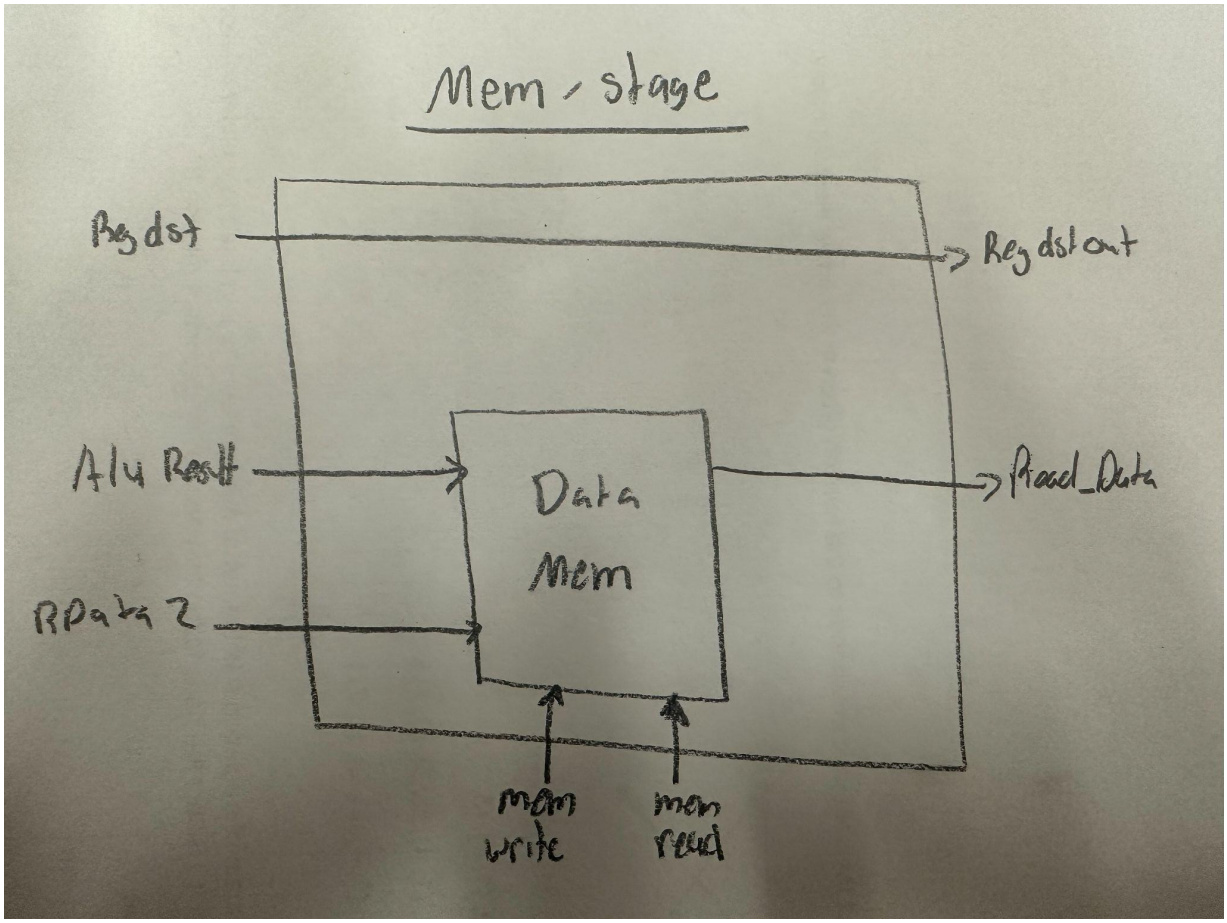

Figure 7: Mem Stage Datapath

Write Back:

The final stage, WB, simply writes back the data to the reg file from the source indicated by the MemtoReg signal.

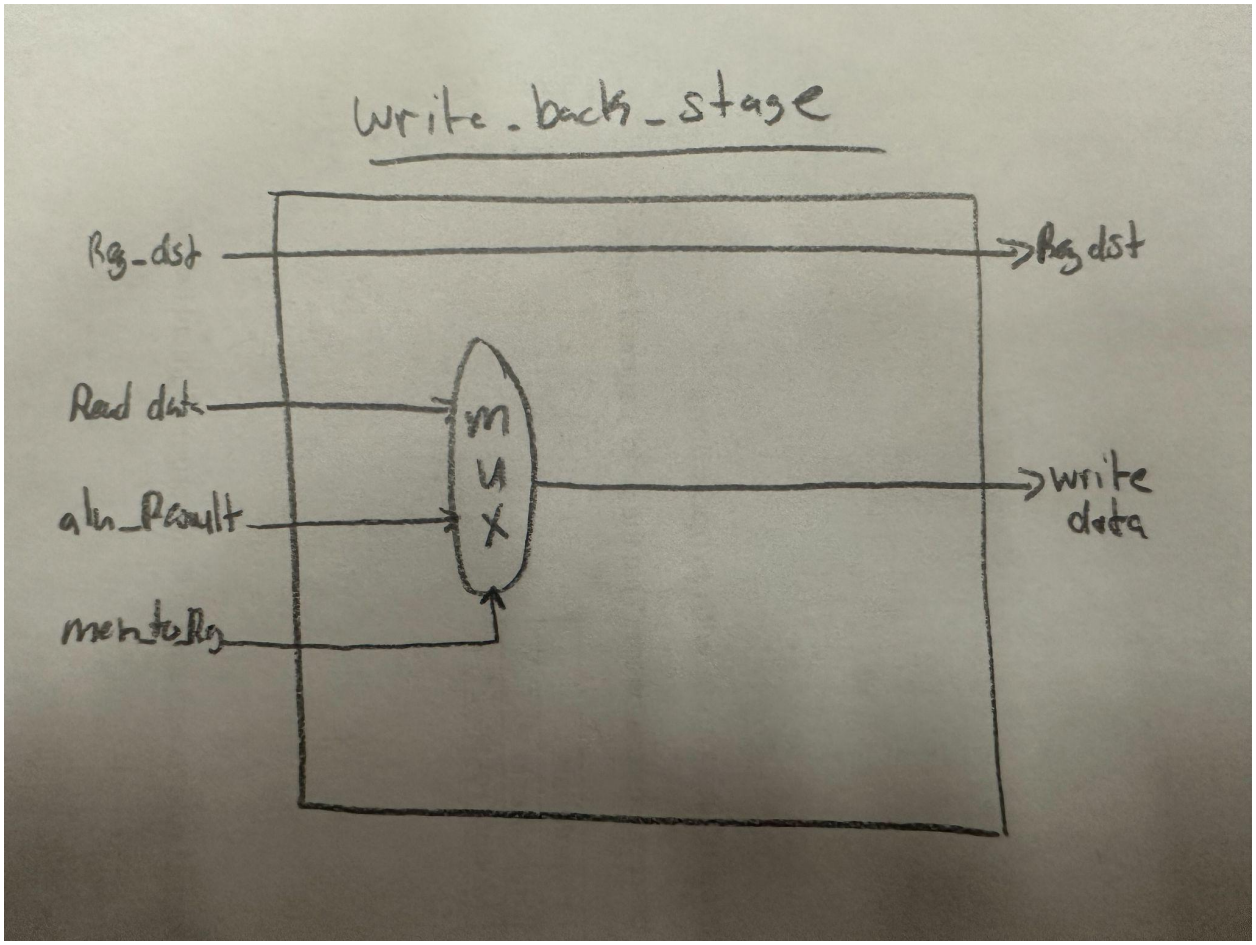

Figure 9: Write Back Stage Datapath

Results and Interpretation

Chaining the stages

Here, we will show each stage being chained together one by one to show their individual functionality.

Instruction fetch:

For the instruction fetch testbench, The PC was manually input into the newPC input on the IF entity every clock cycle. Below shows the output of the testbench. This shows the functionality, and every instruction is being correctly fetched into the buffer.

From top to bottom is the NewPC, the PC carried out of the IF stage, and the instruction out on bottom.

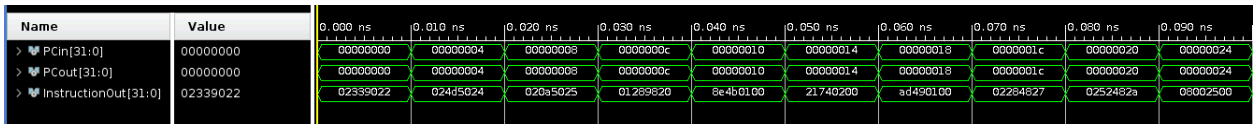

Figure 10: Instruction Fetch Testbench

Instruction Decode:

The instruction decode test bench tested two functionalities: The decoding of instructions passed in from the IF stage and writing to the register file from a manual input.
From top to bottom is:

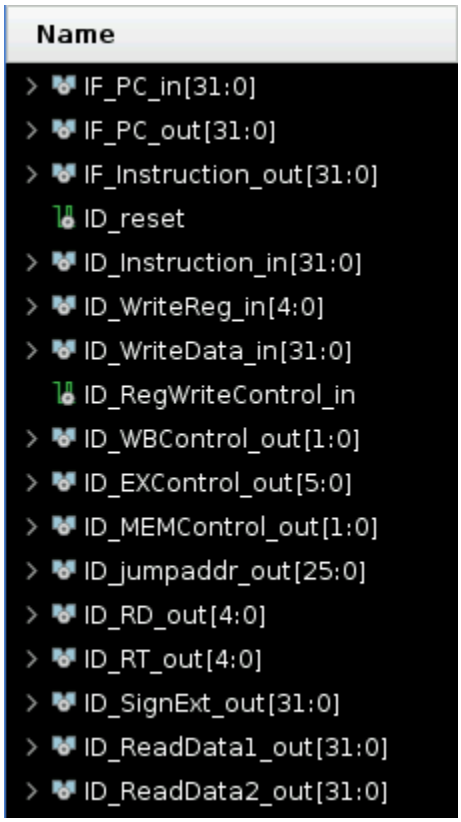

Figure 12: ID testbench signals

Within this test, the goal was not necessarily to test the pipeline. The goal here was to test the decoding of each instruction. Incidentally, the implementation is allowing for proper transition where the fetched instruction is decoded in the next cycle with a new instruction then being fetched.

The first cycle is not relevant because it is simply the first fetch.

Note upon the fetch of the instruction, the ID stage decodes in the next cycle.

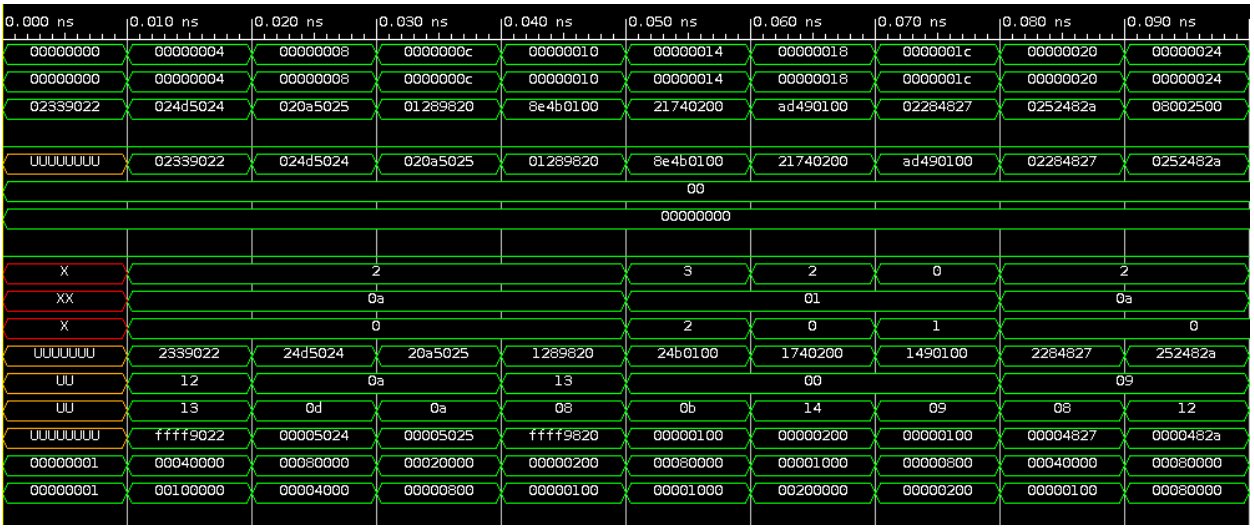

Figure 13: Instruction Decode Testbench

Based on the control logic in Table 1.1. The decoded instructions are correct.

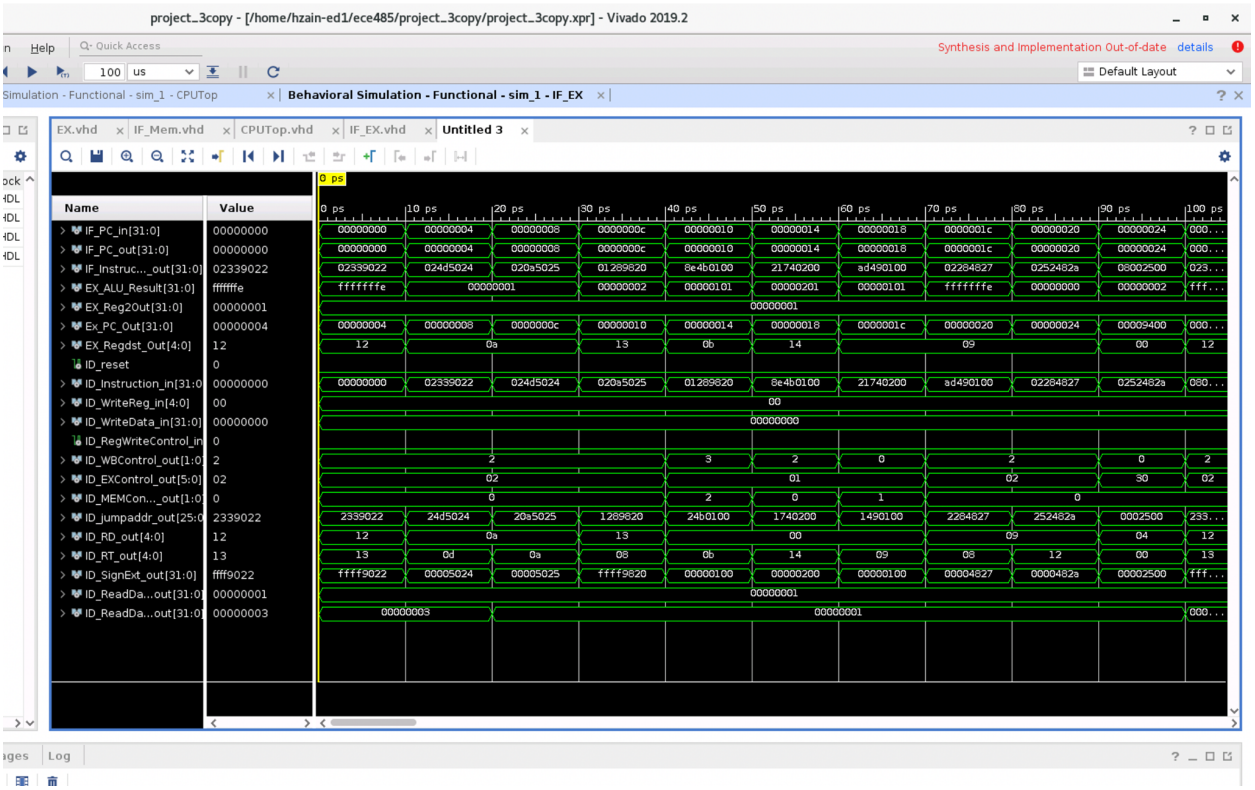

Figure 14: Highlighting the EX stage with data from the Instruction fetch and decode

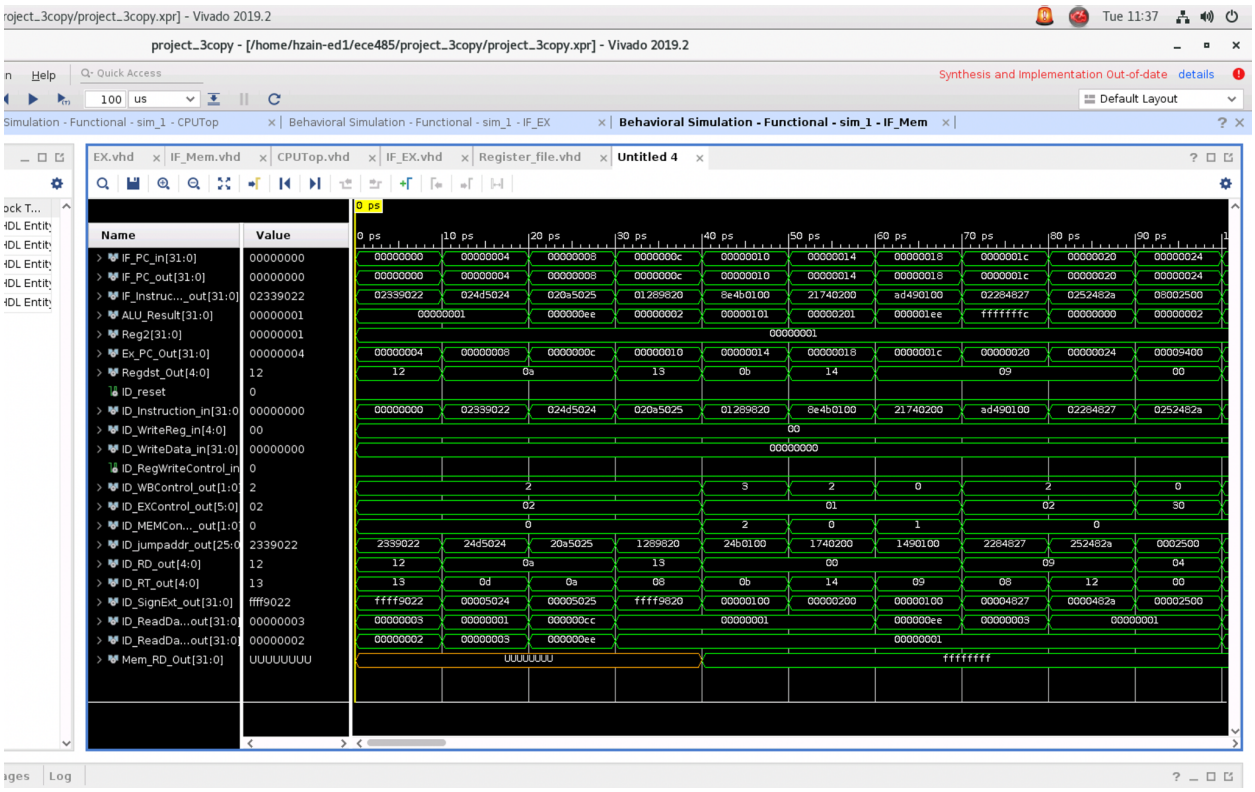

Figure 15: Showing the implementation of IF through MEM and stages between

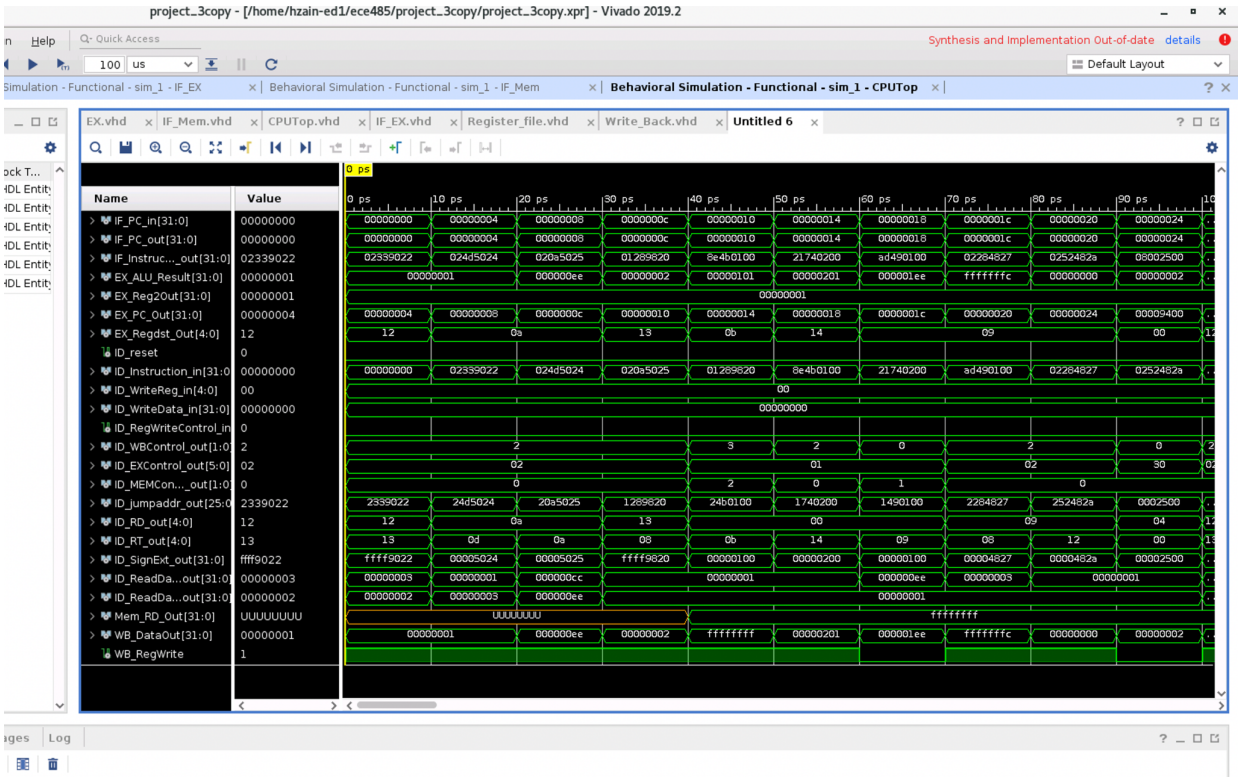

Figure 16: Showing the full CPU implementation

Challenges and Improvements with the Pipeline

The main challenge with the pipeline was combining each stage over sequential clock cycles, and also automating the pipeline such that the instructions were fetched in a way that would avoid data hazards.

The final pipeline implementation hardcoded the buffers in the testbench (Figure 7 and 8). As this happens, the signals are directly mapped to the inputs of each stage. This “implementation” only works as far as the first instruction. As expected, each value propagates through the pipeline after each cycle.

As the first instruction finishes, and the second one starts the ports stay mapped from the outputs of the previous stage. As each stage happens throughout the nth instruction, the values update after the control signal changes, potentially causing a register or memory location to change if the reg write is enabled to run. In this case causing undefined behavior in the cpu, particularly during load and store operations. A similar situation can take place in the data memory with reading and writing as control signals change but values stay the same. .The first thing to implement is buffer registers. This would prevent the control signal propagation through the circuit before the proper stage. From this, each stage will be executed properly and no overwrites or leakage into registers or memory will take place.

The second thing to be implemented would be a proper clock. As it currently stands, each cycle was hand-forced on the CPU and it has no autonomous PC changing. This is in part to do with the previous discussion on buffers, but also is impacted by a lack of clock. With a clock, the PC does not need to be forced defined as it would be automatically written throughout the execution of the PC.

The final thing to be changed is the fetching of the PC itself and stalls. As the clock and the buffers create proper functionality, what comes next is optimization. Being that the forwarding was not required for this project, the most optimal way to have the next IF is after the MEM stage is executed. From this, moving the jump and PC inc into the Mem stage would be optimal. To prevent undefined behavior from this, stalls would need to be implemented such that no instruction is repeated as IF cycles to MEM for a given instruction.

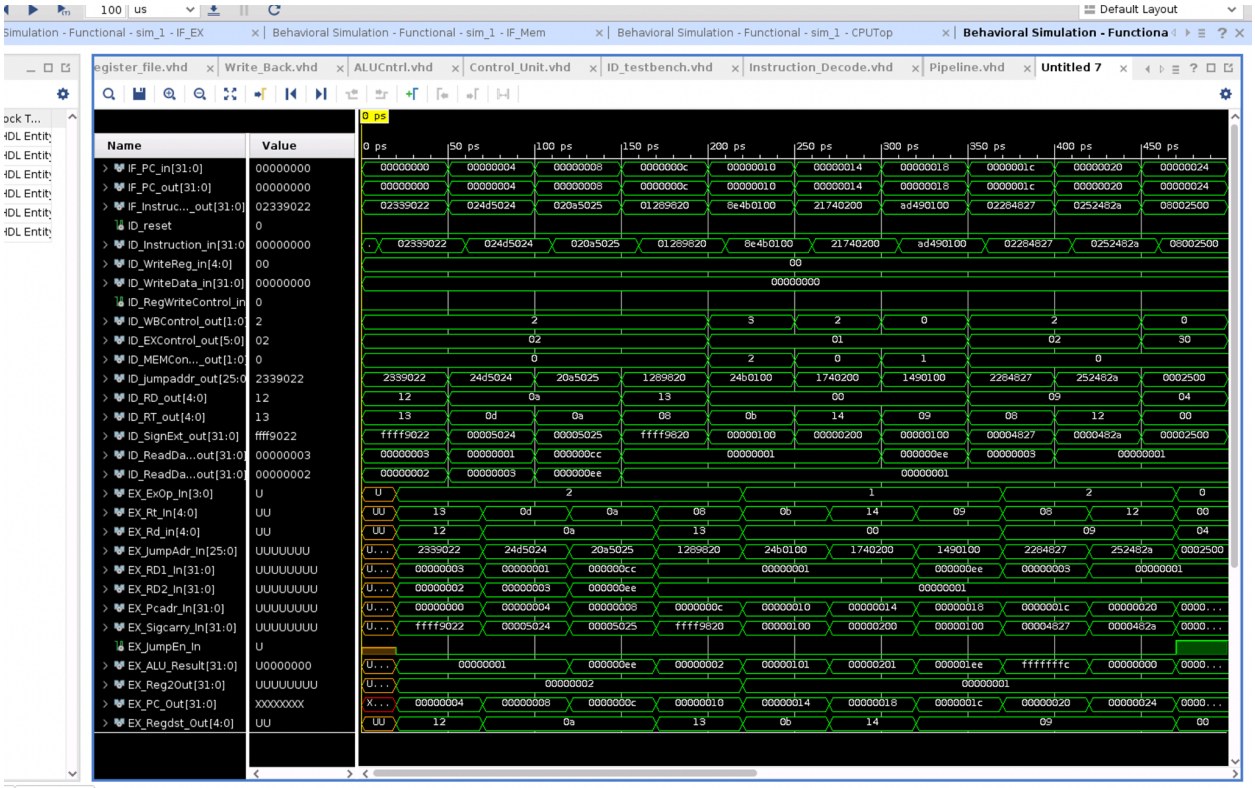

Figure 17: The IF through EX Stage outputs for the pipeline testbench

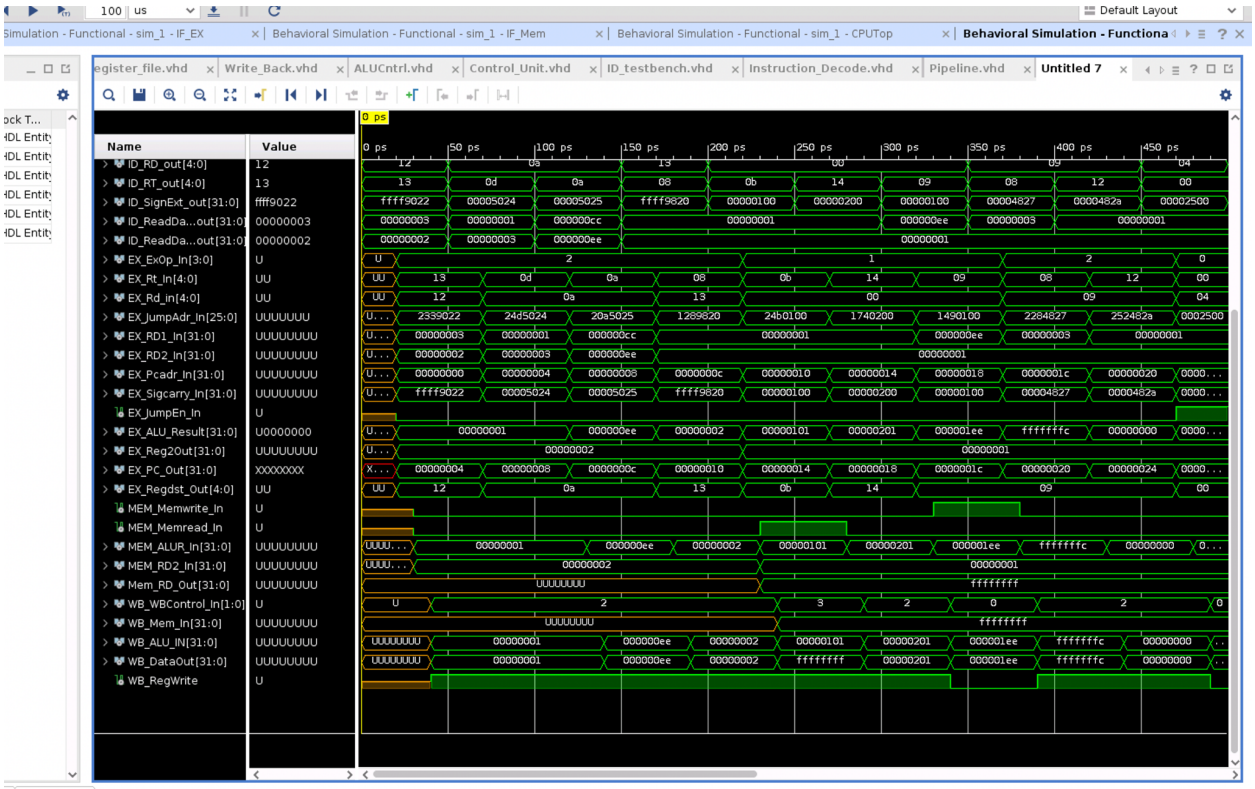

Figure 18: The MEM and WB Stage outputs for the pipeline testbench

Conclusion

For the single issue version of the processor, the critical path was 10 clock cycles, or 100 ps. Overall, each stage was individually completed and successfully was able to reproduce accurate data. Then pieced all together to create a full CPU where excluding a proper clock all data was accurately carried through and results remained correct and persistent. All that was missing was a proper clock, and hazard modules, in order to make it fully functional as a pipeline architecture. Overall it was a great FPGA learning experience and allowed us to really stretch our minds to think conceptually about how these processors where created and how they function.

The final datapath is shown above when each schedmaic is pieced together.

Appendix

Appendix A: Contribution

Stages

- IF, ID, WB: Alex Maliwat
- EX and MEM: Hazim Zain-Edin

Report

- Control Unit, EX, ALU, Memory: Hazim Zain-Edin
- Introduction, Abstract, IF, WB, Conclusion: Alex Maliwat
- System Design: Alex Maliwat , Hazim Zain-Edin

Code is seen in the rest of the ZIP file.