Alihsa Mehta
Professor Quon
ECE 4300

FInal Code Report

Github Repo: https://github.com/alishamehta/ECE-4300-Final-Exam

Section 1. Fetch

The purpose of the fetch stage is to receive the instruction and supply it to the pipeline. It also updates the program counter to point to the next instruction.

There are 5 components of the fetch stage:

1. A 2x1 Multiplexer
2. An adder
3. Program Counter (PC)
4. Instruction memory
5. The IF/ID latch

The MUX has two 32-bit inputs and a 1-bit select. When the select is low, the MUX passes the output of the adder as the next instruction address. When the select is high, the MUX instead forwards an alternate 32-bit input as the next instruction address. The MUX output is connected to the PC register, and that value is also sent to instruction memory, which uses it to fetch the 32-bit instruction. The instruction fetched from memory is stored in the IF/ID latch, along with the incremented PC value from the adder. The outputs of the Fetch stage (e.g. next instruction and instruction data) get passed through the IF/ID latch to be the inputs of the Decode stage.

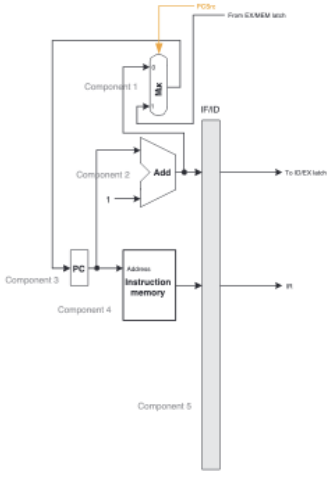
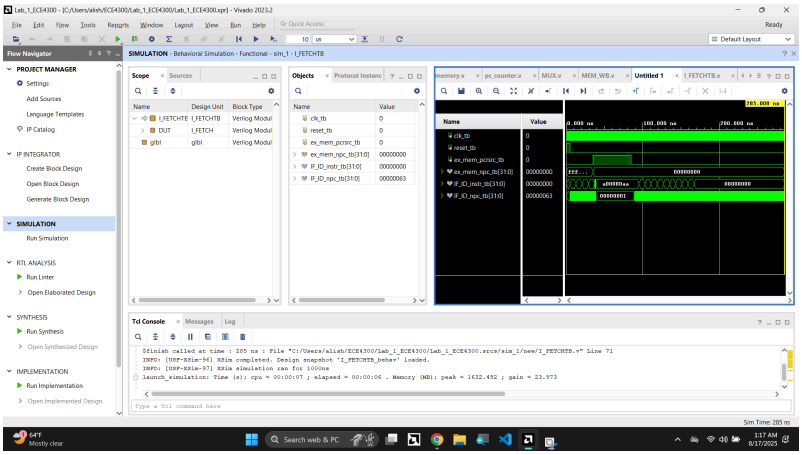


Figure 1.2: The IF stage



Anticipated outputs are correct because:
PC starts at 0, increments by 4 each cycle (00000000 -> 00000004 -> 00000008).
Instruction memory returns the word at each PC address (00000001, 00000063, a00000aa).
IF/ID latch correctly holds both the instruction and the incremented PC.

Section 2. Decode

The purpose of the decode stage is to interpret the instruction, read operands from the register file, generate control signals, and prepare all necessary data for the execute stage.
There are 4 components of the decode stage:

1. Control unit
2. Register file
3. Sign-extend unit
4. ID/EX latch

In the Decode stage the opcode is sent to the control unit, which generates control signals for later stages. The register file receives the rs and rt fields from the instruction and outputs the corresponding register values. At the same time, the immediate field is sign-extended to 32 bits. All of these values are stored into the ID/EX latch for use in the next stage.

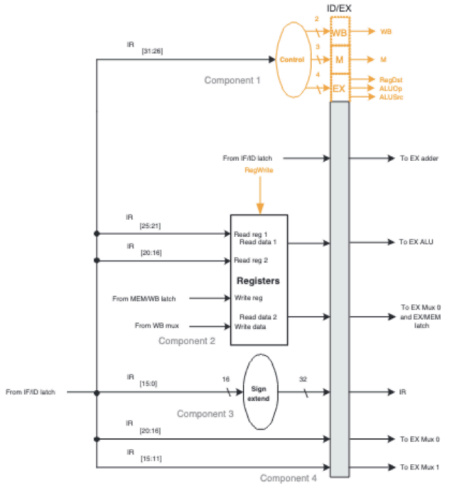
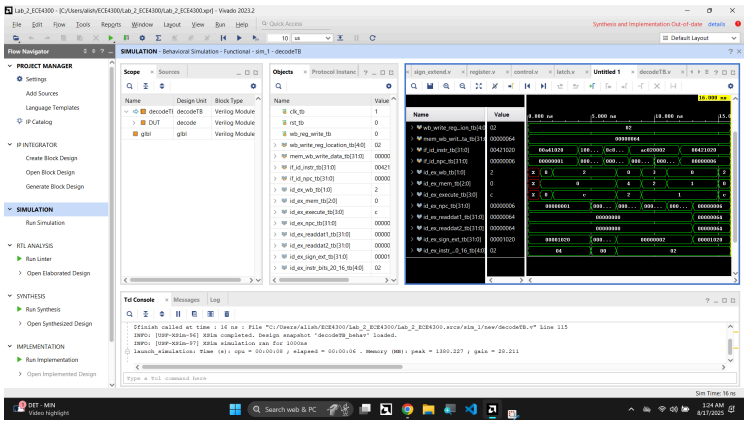


Figure 2.1: The ID stage



Anticipated outputs are correct because:
Inputs

Instruction (IF/ID): 0x00421020 -> decodes to add (opcode=0x00, funct=0x20).
NPC (IF/ID): 0x00000064 (this is PC+4 carried from IF).

Register File Reads
ReadData1 (rs) = 0x00000002

ReadData2 (rt) = 0x00000002
Sign-Extend
Sign-extended immediate = 0x00000000
Control Signals
RegDst=1 (write to rd).
ALUSrc=0 (use register operand, not immediate).
MemtoReg=0, MemRead=0, MemWrite=0, Branch=0.
RegWrite=1 (enable write-back to register file).

Values Latched into ID/EX
NPC -> 0x00000064
ReadData1 -> 0x00000002
ReadData2 -> 0x00000002
SignExt -> 0x00000000
rt (Instr[20:16]) -> 0x02, rd (Instr[15:11]) -> 0x02
Control bundle -> matches the above (RegDst=1, ALUSrc=0, RegWrite=1).

ID/EX cleanly carries operands, control, and NPC=0x64 into Execute.

Section 3. Execute

The purpose of the execute stage is that it carries out the core computations of the instruction. The ALU control unit determines the operation, the MUXes select between operand sources and destination registers, the ALU and adder generate results, and the EX/MEM latch stores everything needed for the Memory stage.

There are 6 components of the execute stage:

1. The adder
2. The ALU
3. 2x1 Mux (32 bits)
4. he ALU control unit
5. 2x1 Mux (5 bits as inputs and outputs)
6. EX/ MEM latch

The Execute stage begins with the ID/EX latch supplying the operands, control signals, and instruction fields. A 32-bit 2x1 multiplexer then chooses between the second register operand (ReadData2) and the 32-bit sign-extended immediate. The selected value, along with ReadData1, is passed into the 32-bit ALU. The ALU Control Unit interprets the ALUOp signal (2–4 bits) and the funct field (6 bits) to determine which specific operation the ALU should perform. Meanwhile, a 5-bit 2x1 multiplexer selects between the rt field (5 bits) and the rd field (5 bits) to identify the correct destination register. Finally, the outputs of this stage ,including the 32-bit

ALU result, the 5-bit destination register number, the 32-bit ReadData2, the one-bit Zero, and the relevant control signals are stored in the EX/MEM latch to be used in the Memory stage.

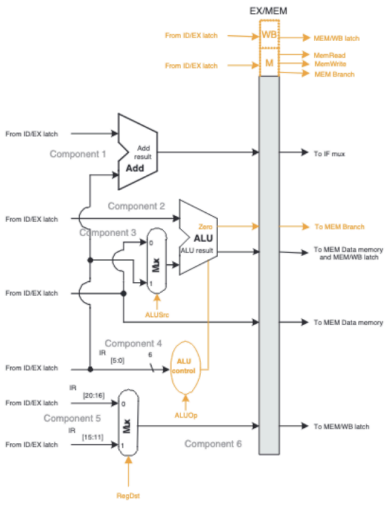
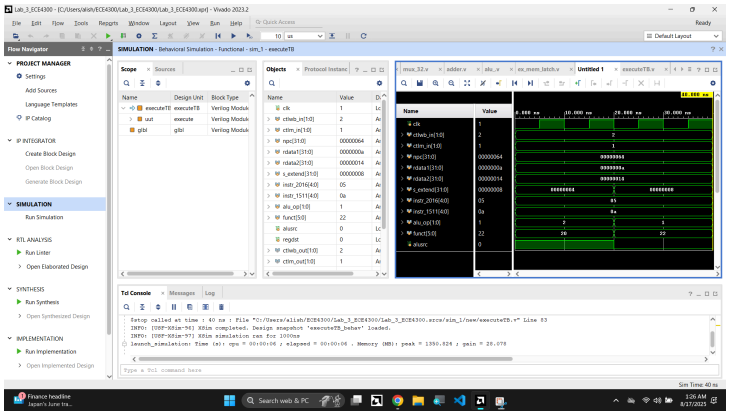


Figure 3.1: The EX stage



The funct field value of 22 specifies subtraction, so the ALU performs 10 – 20 = –10. Since ALUSrc is 0, the second operand comes from the register value of 20. With RegDst set to 1, the destination register is selected as rd, which is register 10. The adder calculates npc + (s_extend << 2) = 132, which is consistent with the shifted value addition. All of these results are then stored in the EX/MEM latch.

Section 4. Memory

The purpose of the memory stage is to access data memory for load and store instructions, while forwarding results and control signals for all instructions into the MEM/WB latch.

There are 6 components of the execute stage:

1. AND gate
2. Data memory
3. MEM/WB latch

The Memory stage takes the 32-bit ALU result, the 32-bit register data (ReadData2), the 5-bit destination register number, the 1-bit Zero, and several control signals (MemRead, MemWrite, MemtoReg, RegWrite, Branch) from the EX/MEM latch. The ALU result serves as the 32-bit memory address, while ReadData2 is used as the 32-bit input data if a store instruction is being executed. The AND gate takes the 1-bit Branch signal and the 1-bit Zero as inputs and produces a 1-bit branch decision output. The data memory receives the 32-bit address (ALU result) along with control signals and outputs a 32-bit memory value if MemRead is asserted, or writes the 32-bit ReadData2 value if MemWrite is asserted. Finally, the MEM/WB latch captures the 32-bit

memory data, the 32-bit ALU result, the 5-bit destination register, and the forwarded control signals (MemtoReg, RegWrite). These outputs are then passed to the Write Back stage for final register updates.

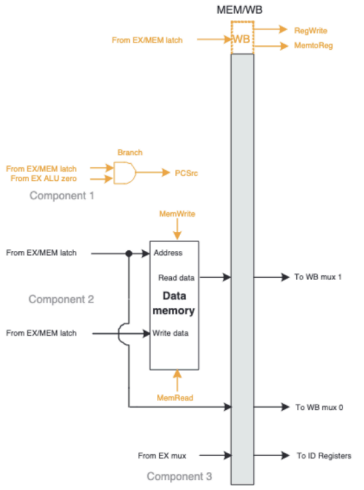
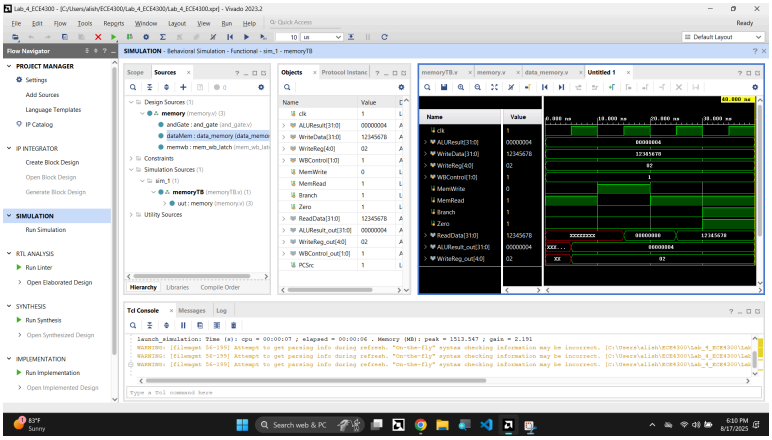


Figure 4.1: The MEM stage



Anticipated outputs are correct:
ALUResult = 0x00000004 is passed through as the memory address and forwarded to the MEM/WB latch. WriteData = 0x12345678 is written to memory because MemWrite = 1.MemRead = 0, so no new memory value is fetched, but the written data is still forwarded. Branch = 1 and Zero = 1, so the AND gate outputs 1, meaning the branch condition is true. WriteReg = 0x02 and WBControl = 01 are forwarded unchanged into the MEM/WB latch.

Section 5. Writeback

There are 1 component of the writeback stage:

1. WB Stage (with inputs form MEM/WB latch and iD registers as output)

The Write Back stage takes the 32-bit ALU result, 32-bit memory data, 5-bit destination register, and control signals from the MEM/WB latch. Using the MemtoReg multiplexer, it chooses the correct 32-bit value to write back and forwards it, along with the destination register number and RegWrite signal, to the register file in the Decode stage. This completes the instruction's execution by updating the register file with the final result.

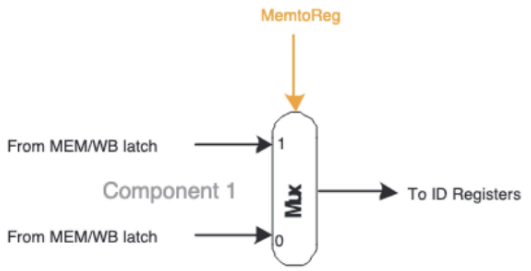
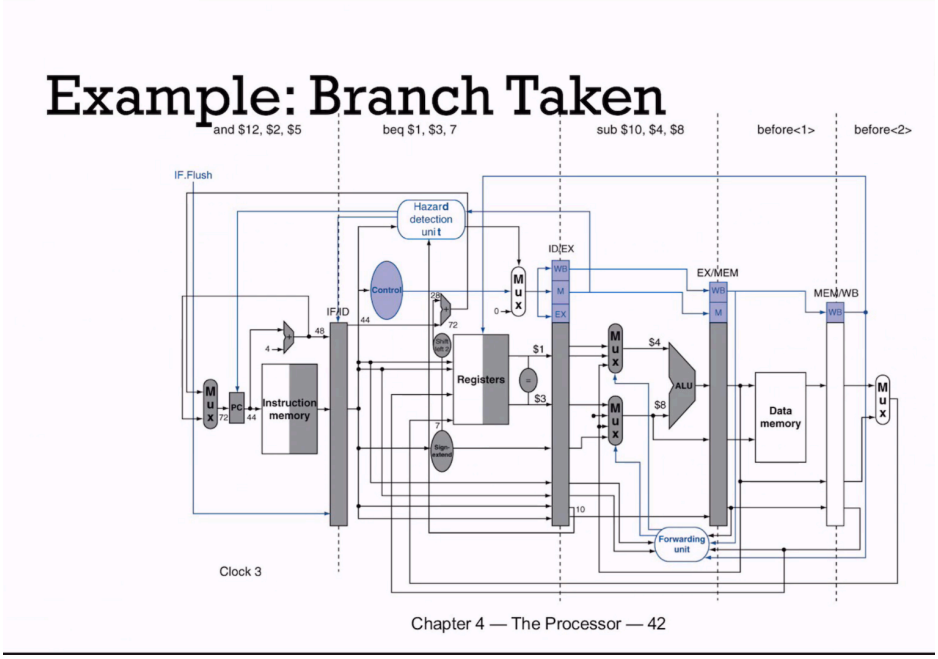


Figure 5.1: The WB stage

Section 6. Conclusion

The implementation of the lab is a simple MIPS pipeline.



In this baseline design, hazards are managed using a hazard detection unit in the ID stage and forwarding paths that feed results from the EX/MEM and MEM/WB latches back to the ALU inputs in the EX stage. This reduces stalls by letting dependent instructions receive results without waiting for the Write Back stage. However, hazards still occur when a load instruction is immediately followed by an instruction that needs its result, requiring the hazard unit to insert a stall. The image also illustrates control hazards, where the outcome of a branch is not known until later in the pipeline, forcing the pipeline to flush or stall until the correct PC is chosen.

One improvement is to move branch evaluation into the ID stage or add a simple branch prediction mechanism in IF to reduce wasted cycles. Another upgrade is to adopt Tomasulo's algorithm, which replaces the in-order execution seen in the image with reservation stations that hold instructions until their operands are ready. Tomasulo's approach uses register renaming to eliminate false dependencies and a common data bus (CDB) to broadcast results directly to all

waiting instructions. A reorder buffer (ROB) then ensures instructions complete and update the register file in program order, preserving precise exceptions. Compared to the in-order pipeline shown in the image, Tomasulo's algorithm allows more parallelism and reduces stalls, keeping the pipeline busier and improving overall throughput.

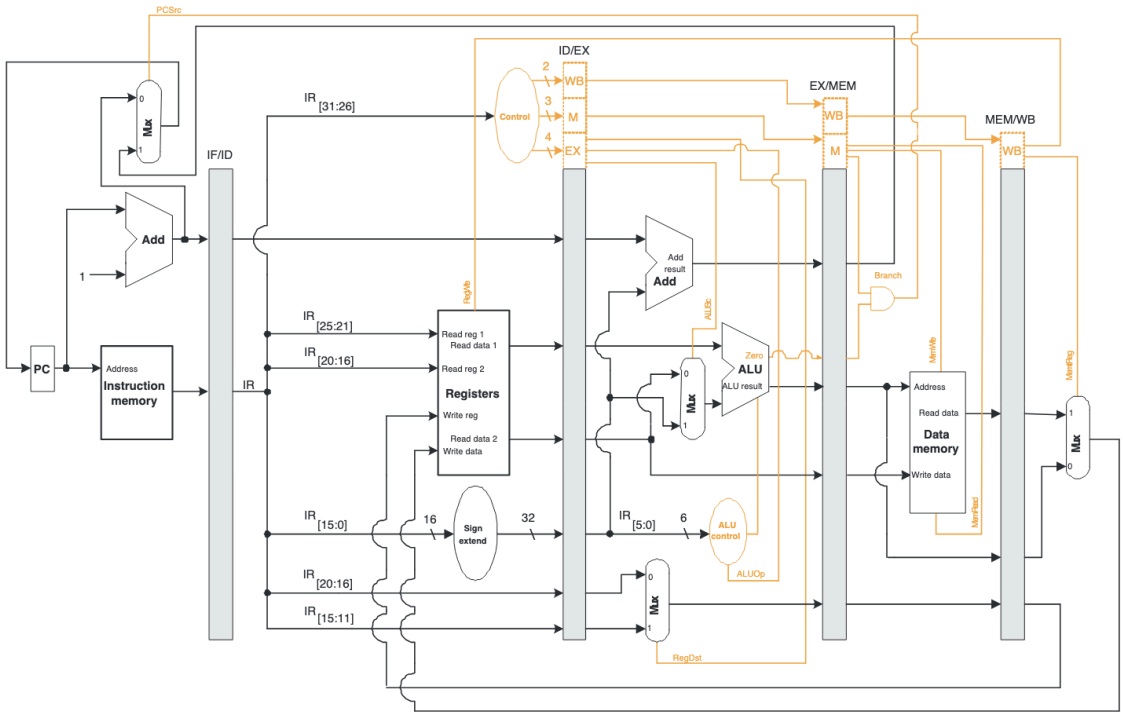


Figure 1.1: The revised MIPS datapath