

CS 3410 Lab 4

Fall 2026



Agenda

1 Circuit Waveform Practice

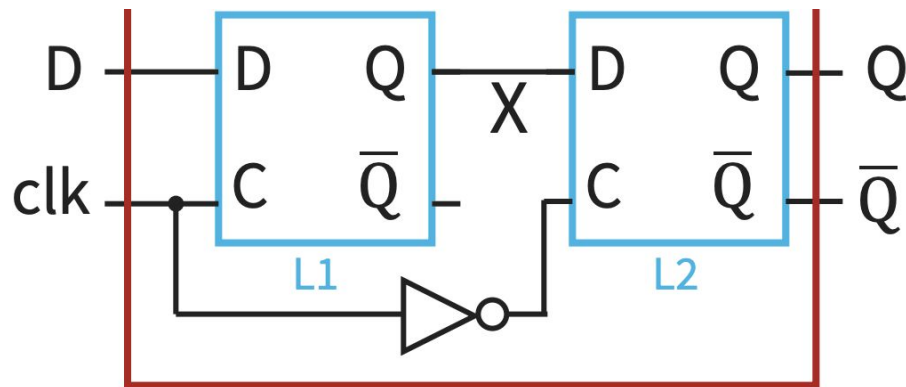
2 CPU Overview

3 Decoding and Encoding

4 Assignment Tips

Circuit Waveform Practice

D Flip-Flop



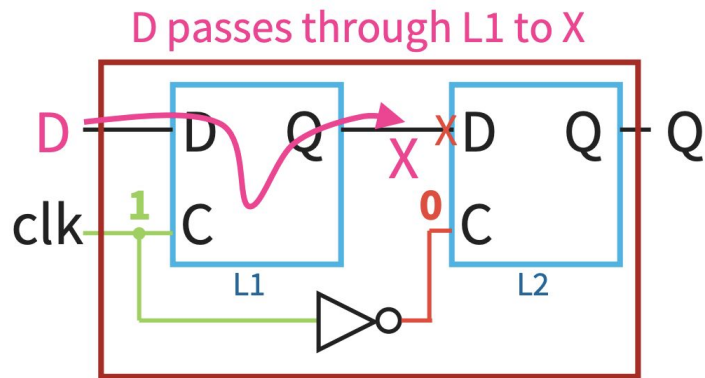
- Edge Triggered (positive or negative). The one above is negative-edge triggered.
- First D-Latch allows input to go through ($D \rightarrow X$) when clock is high.
- Second D-Latch allows input to go through ($X \rightarrow Q$) when clock is low.
- Combining these two means that ($D \rightarrow Q$) occurs when clock falls ($1 \rightarrow 0$).

D Flip-Flop Continued

Clock = 1:

L1 transparent, L2 opaque

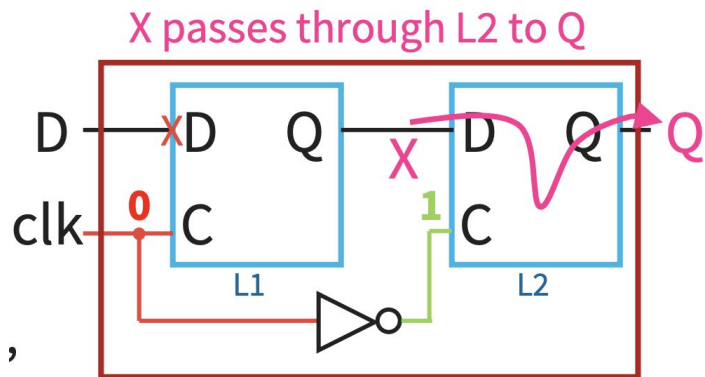
When CLK rises ($0 \rightarrow 1$), now X can change,
Q does not change anymore, X gets value D.



Clock = 0:

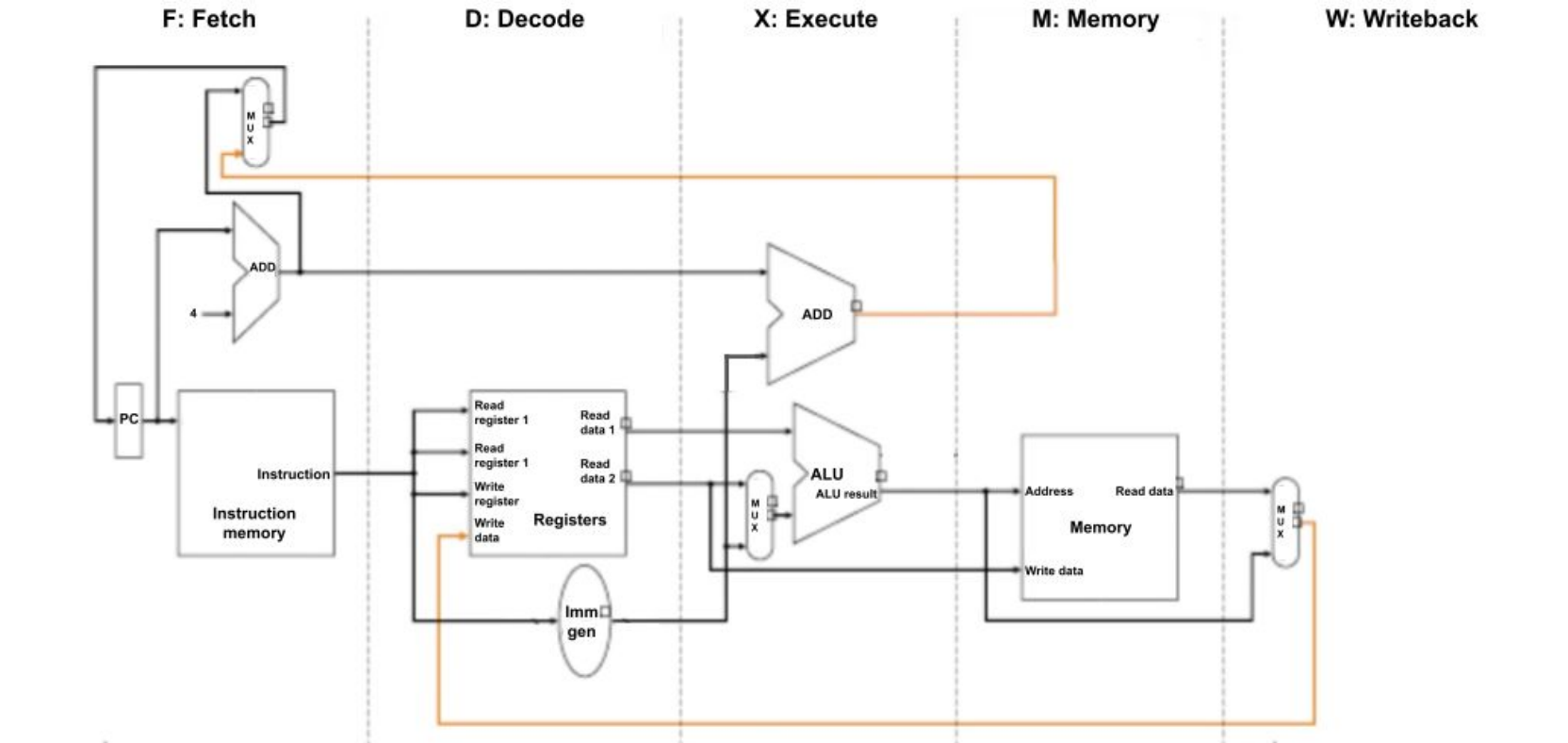
L1 opaque, L2 transparent

When CLK falls ($1 \rightarrow 0$), now Q can change,
X cannot change anymore, Q gets value X.



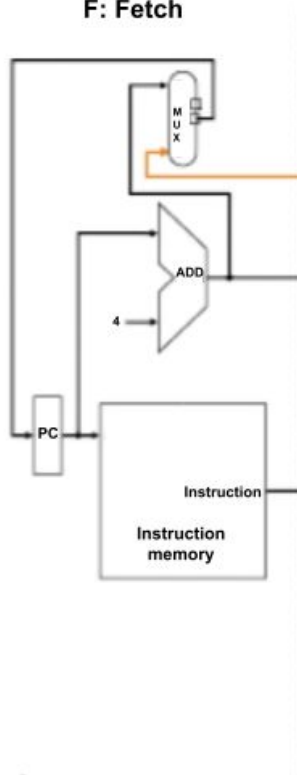
CPU Overview

A 5-Stage RISC Processor



Stage 1: Fetch

F: Fetch



Fetches instruction found at the PC.

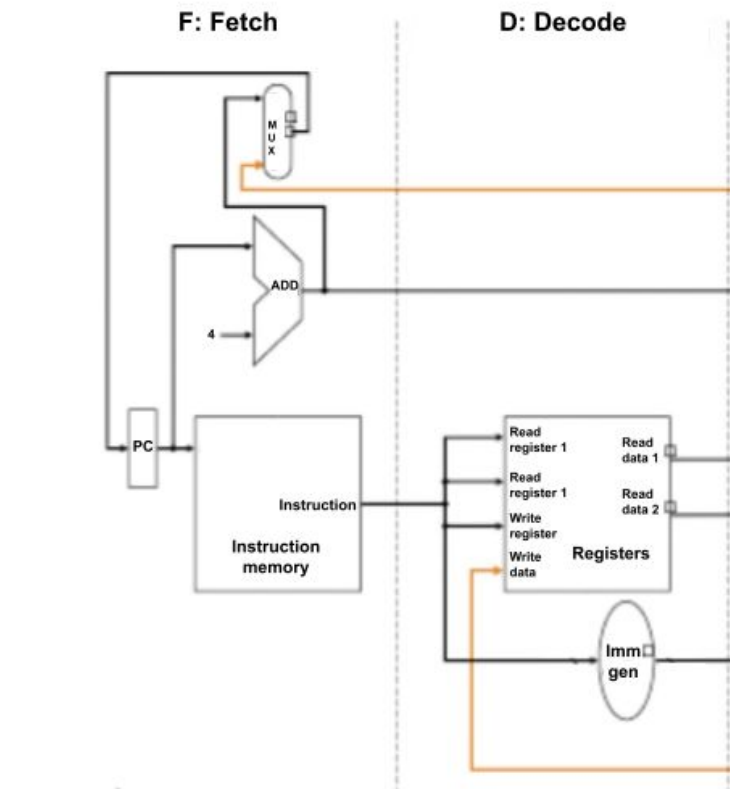
Instruction Memory:

- Stores instructions to execute on the CPU.

Program Counter (PC):

- Points to the next instruction to execute.
- Increments by 4 (1 word) to get next instruction.
- Or by an immediate when we need to jump or branch.

Stage 2: Decode



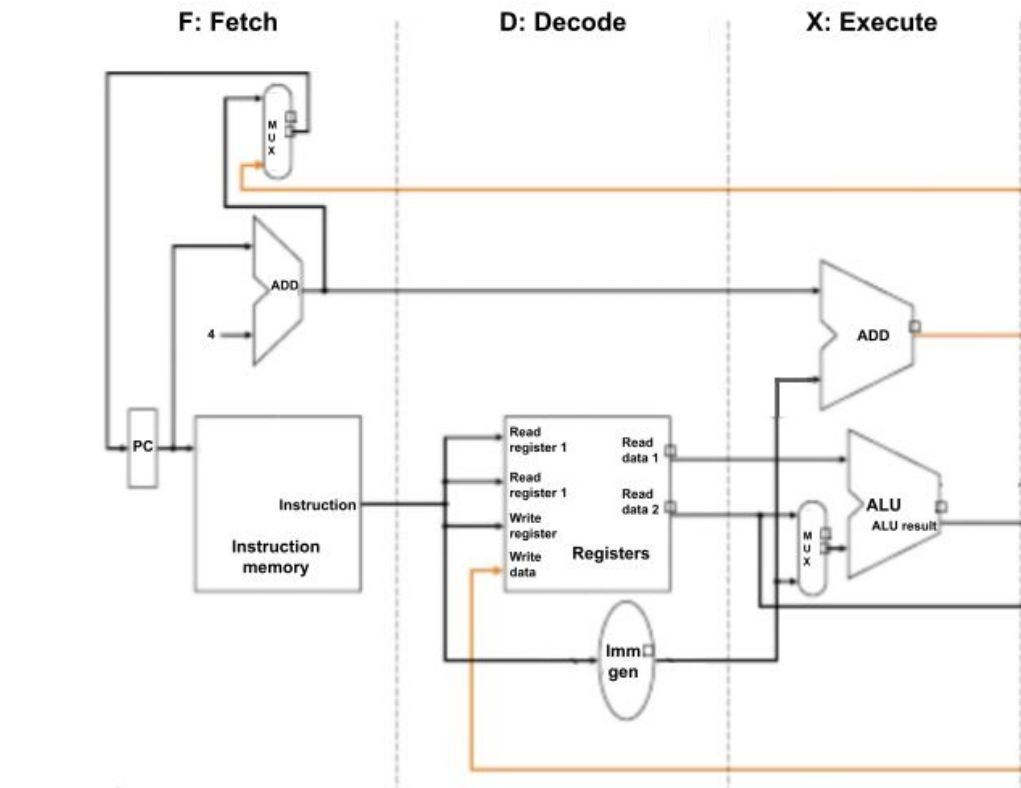
Decode parts of the instruction to decide how to execute it.

Register File:

- Extract values for `rs1`, `rs2` and `rd` registers.
- Extract immediate, sign-extend to convert to 64-bit number.

Note: Not all instructions will use `rs1`, `rs2` or `rd` registers.

Stage 3: Exec



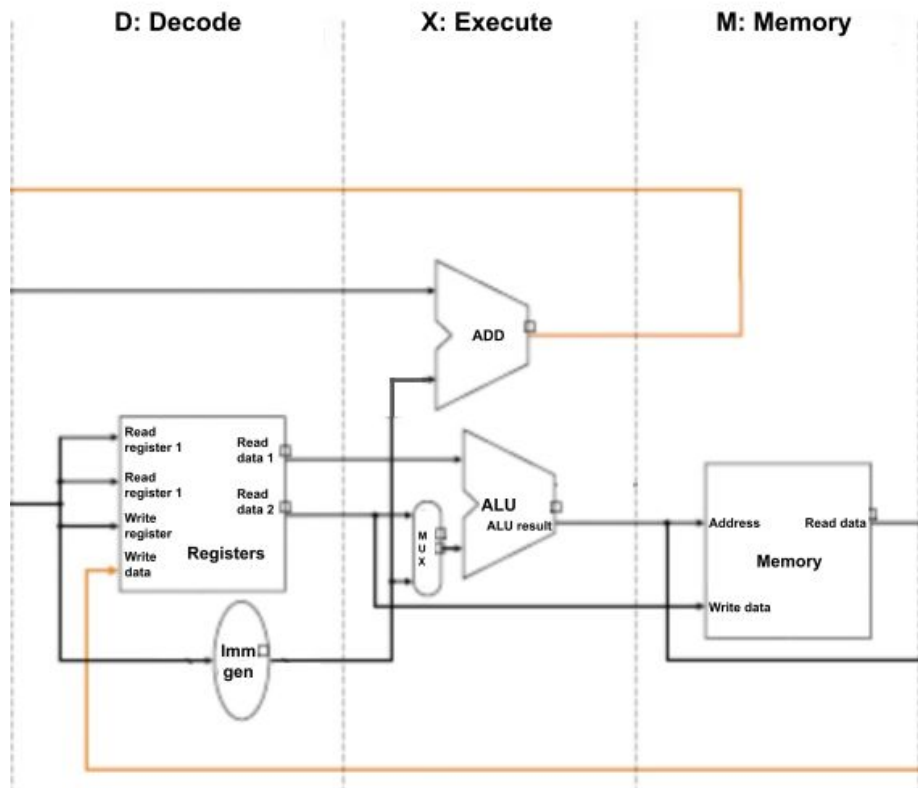
Adder (top):

- Calculate $PC + \text{offset}$. This may or may not be used depending on the instruction.

ALU:

- Performs an operation using `rs1` and either `rs2` or `immediate`.
- Type of op also depends on instruction.

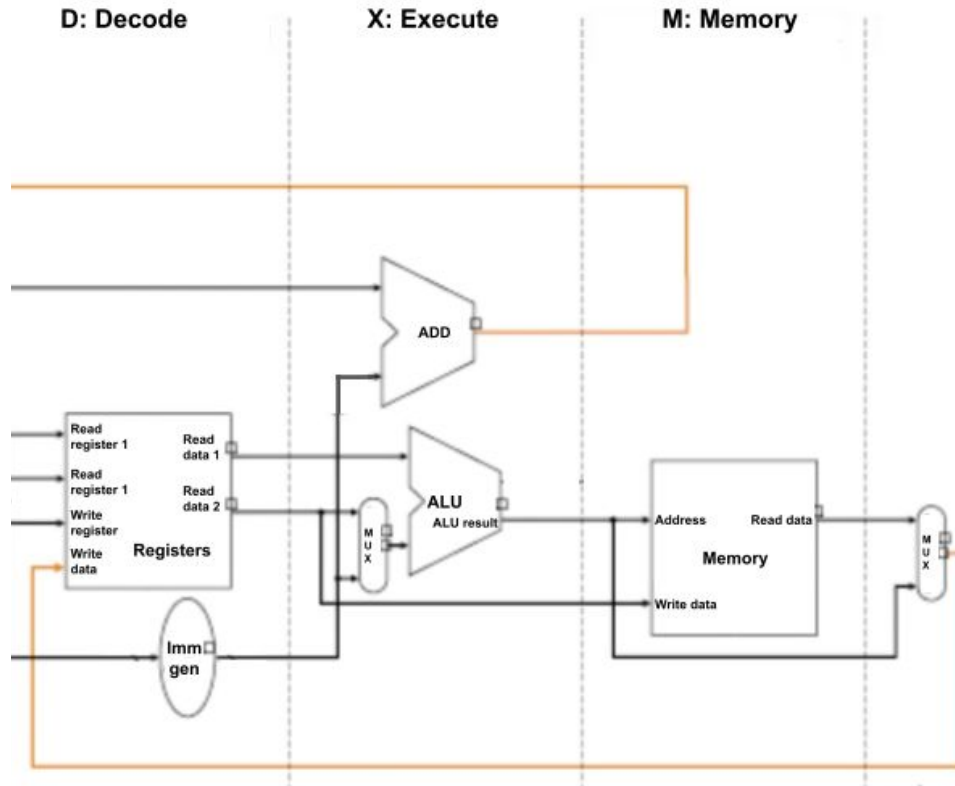
Stage 4: Memory



Data Memory:

- May carry out either a **read** or **write** operation, depending on instruction.
- If **write**, this stage writes data from a register to a particular address.
- If **read**, it reads from a particular location and forwards it to the next stage.

Stage 5: Writeback



The mux decides what value to write back to the register file.

- For **load** instructions, write the output from memory to the 'Write data' register.
- For arithmetic instructions (like **add**), write back the ALU output to the 'Write data' register.

Note: Not all instructions will use this stage (think store instructions).

Decoding and Encoding

I-type Instructions

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
imm[11:0]	rs1	funct3	rd	opcode

- ``addi`` and ``andi`` are examples of I-type instructions



I-type Instructions

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
imm[11:0]	rs1	funct3	rd	opcode

- ``addi`` and ``andi`` are examples of I-type instructions
- We use the **opcode** and **funct3** (and sometimes funct7) bits of the instruction to decide how the instruction will execute

I-type Instructions

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
imm[11:0]	rs1	funct3	rd	opcode

- `addi` and `andi` are examples of I-type instructions
- We use the **opcode** and **funct3** (and sometimes funct7) bits of the instruction to decide how the instruction will execute

Instruction	opcode	funct3
addi	0010011	000
andi	0010011	111

Assignment Tips

Some useful tips for the assignment

- Use the **`info` struct** to store relevant instruction data between processor stages, including what type of instruction it is. We provide a mapping from instructions to integers via the #define macros in logic.h.
- Think how the **`memory` stage** will be used by different instructions (**hint: some won't need to read or write!**). Sometimes this Memory stage will be a no-op that propagates information to the Writeback stage from the ALU.
- Consider how your writeback stage should update the state of the program (the PC counter) to prepare it for the next instruction. (maybe **PC + 4**? maybe **PC + offset**?)



How to test your implementation

- Make sure you clone the assignment repo and your terminal is under that folder
- To test your implementation for the assignment, follow the following steps:
 - Compile your implementation with ***“rv make”***.
 - Write a test script in RISC-V assembly, let's call it ***“prog.s”***.
 - Compile the test script using ***“rv asbin prog.s”***.
 - Run the implementation using ***“rv qemu runner < prog.bin”***.
 - You can also initialize the registers: ***“rv qemu runner 1@0x1 < prog.bin”***.
- When doing multiple tests, highly recommend using the **test_parser.py** script, through ***“rv python3 test_parser.py tests.txt”***
- All this information (and more) can be found in the assignment instruction.

