Homework 4

Due date: Thursday, November 5, 2026, 11:59 PM on Gradescope

Overview:
In HW3 you wrote a **single-cycle** simulator: one instruction started and finished every cycle. In HW4 you turn that into a **5-stage pipeline**, where up to five instructions are in flight at once, one in each of the five classic stages: **fetch (IF), decode (ID), execute (EX), memory access (MEM), and write back (WB)**. Your simulator still reads a program and an initial memory image from the same two text files, but now it clocks the pipeline one cycle at a time until it drains.

The instructions, encodings, memory and register semantics are unchanged from HW3, so your decoder, ALU, memory, and register-file logic carries straight over. **The new work is the pipeline control** that keeps overlapping instructions correct: forwarding, a load-use stall, and control-hazard handling.

What You'll Practice:

- Modeling a 5-stage pipeline with explicit pipeline registers (IF, ID, EX, MEM, WB)
- **Data hazards:** ALU-to-ALU forwarding from EX/MEM and MEM/WB
- **Load-use hazards:** detecting them and inserting a one-cycle stall (a bubble)
- **Control hazards:** resolving branches and jumps without speculation
- Reasoning about what each stage sees "this cycle" versus "next cycle," and why stages are processed back to front

What Is Provided:
A working skeleton that includes command-line parsing, the run-loop, the four output-writers, the five pipeline-register stage containers (State.IF … State.WB, each seeded with only nop and PC), and the memory/register-file constructors. As in HW3, it does not ship opcode tables, the ALU, sign_extend, or the immediate decoders, you write those.

You implement the same six method names as HW3: **InsMem.readInstr, DataMem.readDataMem, DataMem.writeDataMem, RegisterFile.readRF, RegisterFile.writeRF, and the core step.** The first five are identical to HW3, copy your HW3 code in. The core method is now FiveStageCore.step (HW3's was SingleCycleCore.step). This is the focus for implementing HW4. You are also responsible for the instruction decoding, sign_extend / immediate handling, and the ALU, exactly as you were in HW3.

Setup:

1. **Accept the assignment on Classroom50 here:** https://classroom50.org/cmpe120/cmpe-120-fa26/assignments/hw4/accept, then open the repository in your editor.
2. Follow the README to run and test your pipelined simulator. The input files, the --iodir argument, environment setup, and the four output file names are exactly as in HW3, see the HW3 spec if you need a refresher.

The Pipeline:

A single-cycle machine does the whole **fetch–decode–execute–memory–write-back** flow in one clock tick. A pipeline splits that work into five stages and keeps a different instruction in each, so once it is full, one instruction finishes every cycle instead of taking five cycles each. The five stages are:

Stage	Name	What Happens
IF	Instruction Fetch	Read the instruction at the fetch PC, compute PC+4
ID	Instruction Decode	Decode fields, read registers, resolve branches and jumps
EX	Execute	The ALU computes a result or a memory address
MEM	Memory	Load or store data memory
WB	Write Back	Write the result into the register file

Between the stages sit pipeline registers that carry one instruction's state from one stage to the next. The skeleton provides the five stage containers: `State.IF`, `State.ID`, `State.EX`, `State.MEM` and `State.WB`, but each starts with only two keys, `nop` and `PC`, which is all the given run-loop needs to print the trace. You add the remaining fields your datapath carries between stages (decoded fields, register values, ALU inputs/result, control bits, and so on). An empty stage is a bubble, marked `nop=True`.

Each call to `step()` advances the pipeline by exactly one clock. You read from `self.state` (this cycle) and write into `self.nextState` (what the registers will hold next cycle); the skeleton copies `nextState` onto state at the end of the clock.

Process the stages back to front: WB, then MEM, then EX, then ID, then IF, so that a later stage sees the value a younger stage produced this same cycle (this is how forwarding and the write-before-read register file work).

Note: that the per-instruction behavior of each instruction, what the ALU computes, how immediates are sign-extended, what LW/SW/BEQ/BNE/JAL/JALR do, how HALT is detected, is unchanged from HW3. Reuse the HW3 spec for those details. HW4 only changes when and how those steps are spread across cycles and how hazards between overlapping instructions are resolved.

Hazards:
Because instructions overlap, an instruction can need a result that an older instruction has computed but not yet written back. These are called hazards.

Handle **Data hazards** by **forwarding**. When an instruction in EX needs a register that an instruction ahead of it (in MEM or WB) is going to write, do not wait for the write-back; forward the newer value straight into EX. Forward from the EX/MEM pipeline register instead of from the MEM/WB register when both match the same register because EX/MEM holds the more recent value. Never forward from x0, and never forward from an instruction whose write is disabled.

Handle **load-use hazards** by **stalling**. A load does not have its value until the end of MEM, so an instruction that uses a load's result in the very next cycle cannot be fed by forwarding alone. Detect this case in ID (the consumer is in ID while the load is in EX) and stall one cycle: hold IF and ID, and insert one bubble into EX. After that single stall the loaded value is available by forwarding.

Handle **control hazards** by **stalling**. Do not predict branches and do not speculate. Resolve BEQ, BNE, JAL and JALR in ID. When a control instruction is in ID, stall IF for that cycle and insert a bubble, then redirect fetch to the correct next PC, the taken target if the branch is taken, or PC+4 if it is not. As in HW3, branch and jump offsets are relative to the address of the branch itself (pc + immediate), not pc + 4. A branch whose operands are still being produced by a load must itself stall until the operand is ready, exactly like any other load-use. Never let a wrong-path instruction reach EX.

Cycles:
One instruction retires (leaves WB) per cycle once the pipe is full. `instructions_executed` counts instructions that leave WB, not clocks. HALT and bubbles are never counted. HALT stops fetching; the pipeline then drains the instructions already inside it, and the simulation ends when all five stages are empty.

Output Files:

The simulator writes the same four files into `--iodir` when the simulation ends, and the skeleton's writers produce them for you. Three of the four are byte-for-byte the same format as HW3, reuse the HW3 spec for them:

- `RFResult.txt`: the 32 registers, 32 binary characters each (unchanged).
- `DMEMResult.txt`: all 1000 bytes of data memory, 8 binary characters each (unchanged).
- `PerformanceMetrics.txt`: Total Cycles, Instructions Executed, CPI, IPC (same four lines as HW3). The only difference is in the values: because of pipeline fill, drain and stalls, Total Cycles now exceeds Instructions Executed, so CPI is above 1.0 and IPC below 1.0. instructions_executed counts instructions that leave WB (HALT and bubbles excluded).

TraceResult.txt - NEW FORMAT:

One line per clock cycle, in order. The six columns are the cycle number (from 0) and the byte address of the instruction occupying each of the five pipeline stages that cycle: `cycle IF.PC ID.PC EX.PC MEM.PC WB.PC`. A stage that holds a bubble (because the pipe is still filling, draining, stalled, or flushed) prints `-1` for that column. As in HW3, lines beginning with # are comments, so the skeleton's header is ignored.

Example:

Code:

```
.data 7 11 13 17      ; mem[0]=7, mem[4]=11, mem[8]=13, mem[12]=17

LW   x1, x0, #0       ; PC 0    x1 = 7
ADD  x2, x1, x1       ; PC 4    x2 = 14    load-use with the LW above
LW   x3, x0, #4       ; PC 8    x3 = 11
LW   x4, x0, #8       ; PC 12   x4 = 13
ADD  x5, x3, x4       ; PC 16   x5 = 24    load-use with the LW at PC
12
ADD  x6, x5, x1       ; PC 20   x6 = 31
SW   x6, x0, #16      ; PC 24   mem[16] = 31
HALT                  ; PC 28
```

Trace:

```
# cycle IF.PC ID.PC EX.PC MEM.PC WB.PC
0 0 -1 -1 -1 -1
1 4 0 -1 -1 -1
2 8 4 0 -1 -1
3 8 4 -1 0 -1
4 12 8 4 -1 0
```

```
5 16 12 8 4 -1
6 20 16 12 8 4
7 20 16 -1 12 8
8 24 20 16 -1 12
9 28 24 20 16 -1
10 -1 -1 24 20 16
11 -1 -1 -1 24 20
12 -1 -1 -1 -1 24
```

Reading the example: at cycle 0 only IF is busy (fetching the instruction at PC 0). The pipe fills over the next few cycles. The -1 in the EX column at cycle 3 (with ID repeating PC 4 and IF repeating PC 8) is the bubble from a one-cycle load-use stall: the load at PC 0 moved on to MEM while its consumer at PC 4 waited in ID and a bubble went into EX.

After HALT is fetched at cycle 9, IF prints -1 from cycle 10 on and the pipe drains one stage per cycle. The SW at PC 24 is the last instruction; cycle 12 (-1 -1 -1 -1 24) is the final row, the cycle it leaves WB. The run ends there because the pipe is empty on the next cycle, so there is no all -1 row, the final row still shows the last instruction in WB. HALT is fetched but never retires, so it never appears in ID or later.

Note: As in HW3, what matters is the values and the count: RFResult.txt needs exactly 32 lines, DMEMResult.txt exactly 1000, and `TraceResult.txt` exactly one row per cycle. Total Cycles and Instructions Executed must be exact; CPI and IPC are compared with about 0.5% tolerance.

Running & Testing:

Run and test exactly as in HW3:

```
python main.py --iodir tests/01_rtype_alu/input   # run one case
python run_tests.py                               # all public cases
python run_tests.py 01 05                         # just cases 01 and
05
```

Add `--verbose` to print one line per clock showing the pipeline occupancy, the quickest way to watch a stall or a flush happen. You can still write your own programs in `asm/` and build them with assemble.py (see the `README`); `asm/hazards.asm` is a ready-made program that exercises a load-use stall and ALU forwarding.

Gradescope Submission:

There are 25 test cases, each a RISC-V program written to exercise the pipeline: back-to-back ALU chains, load-use stalls, store/load visibility, forward and backward branches, and call/return.

8 are public and live in your repository, so you can see both the program and its expected output. 17 are hidden, used only by the autograder; they cover the same instructions and hazards.

As in HW3, input/code.asm lists the address, encoding and disassembly of every instruction, and is the fastest way to see whether a failing case is a decoder, forwarding, stall, control-flow, or memory problem.

Scoring is out of 100 points: 25 cases × 4 points, each of the four output files worth 1 point. A case earns full marks only if all four files match; per-file partial credit is reported.

Submission mechanics (keep the file named `main.py`, do not change `--iodir` or the output file names, do not modify the provided writers, 10-second-per-case limit, GitHub submission on Gradescope) are identical to HW3.

Troubleshooting:
For decoder, byte-order, sign-extension and HALT issues, see the HW3 troubleshooting list, those failure modes are unchanged.

New pipeline-specific ones:

- NotImplementedError: FiveStageCore.step: you have not filled in the pipeline. The five memory/register methods are copied from HW3; step is the new work, and since the skeleton ships no decoder, ALU, or pipeline-register fields, those are part of it.
- Trace off by the number of bubbles: a stage holding a stall or flush bubble must print -1, not a repeated PC. During a load-use stall, IF and ID repeat their PC while EX prints -1.
- A value is one instruction stale: your forwarding is missing or backwards. When EX/MEM and MEM/WB both target the same register, EX/MEM wins because it is newer. Process the stages back to front so each stage sees what a younger stage produced this cycle.
- A load's result is wrong in the next instruction: forwarding alone cannot cover a load-use. You need the one-cycle stall first, then forward. The consumer is detected in ID while the load is in EX.
- Right cycles but wrong redirect: resolve the control instruction in ID and insert a bubble; never predict or speculate. (The offset itself is relative to the branch PC, as in HW3.)
- A wrong-path instruction executed: you fetched past a taken branch before resolving it. On a control instruction in ID, stall IF and insert a bubble that cycle, then fetch the resolved target.

- Runs past HALT, or a length mismatch: HALT stops fetching and is not retired. Drain the stages already in flight, then stop when all five are empty.
- Too slow: use plain lists and integers. Do not rebuild the memory image on every access, and build output with join rather than repeated append.