Assignment: **RVX10-P** — Five-Stage Pipelined Implementation of the RVX10 Core

Context. In the previous assignment, you extended your single-cycle RV32I processor by adding ten custom ALU instructions under the **RVX10** extension. In this lab, you will transform that same design into a **five-stage pipelined implementation** called **RVX10-P**. Your pipelined CPU must correctly execute all base RV32I and RVX10 instructions and produce identical architectural results as the single-cycle version.

Learning Outcomes

- Understand the five pipeline stages and their separation by pipeline registers.
- Implement forwarding, stalling, and flushing for hazard handling.
- Verify correctness and measure cycle-level performance.
- Appreciate the throughput improvement introduced by pipelining.

Pipeline Overview

Your processor will follow the standard five-stage structure:

Stage	**Description**
IF	Instruction Fetch
ID	Instruction Decode / Register Read
EX	Execute (ALU and branch decision)
MEM	Data Memory Access
WB	Write Back to Register File

Each stage communicates with the next through dedicated pipeline registers: `IF/ID`, `ID/EX`, `EX/MEM`, and `MEM/WB`.

Rules & Implementation Guidelines

- Keep the same ISA coverage as your RVX10 single-cycle core (RV32I + 10 custom instructions).
- Each stage must complete in one clock cycle; all combinational logic from the single-cycle design must now be distributed across stages.
- Introduce **pipeline registers** to hold intermediate values (e.g., PC, instruction, operands, ALU results, control bits).
- Add two new control blocks:
  (a) **Forwarding Unit:** selects ALU inputs from EX/MEM or MEM/WB if data hazards exist.

(b) **Hazard Detection Unit:** detects load-use hazards and generates stall/flush signals.

- On a taken branch or jump, **flush** the next instruction (convert to NOP).
- Writes to `x0` must still be ignored.

Design Tasks

1. Partition the Single-Cycle Datapath

Divide your single-cycle RVX10 datapath into five pipeline stages. Create pipeline register modules or bundled signals:

- `IF/ID`: holds PC and fetched instruction.
- `ID/EX`: holds register values, immediate, control bits.
- `EX/MEM`: holds ALU output, destination register, and memory control signals.
- `MEM/WB`: holds data memory read result and final write-back control.

2. Integrate Hazard Handling

Data Hazards:

- Implement forwarding paths from MEM and WB stages to ALU inputs.
- Detect load-use dependency: if EX stage needs data from a load still in MEM, stall IF and ID, and flush ID/EX.

Control Hazards:

- Evaluate branches in the EX stage.
- If a branch is taken, flush the instruction in IF/ID and update PC.

3. Extend Support to RVX10 Instructions

Your ALU and control unit must handle the same 10 custom operations: `ANDN, ORN, XNOR, MIN, MAX, MINU, MAXU, ROL, ROR, ABS`. Since all are ALU-only, they naturally fit in the EX stage. No change is needed in MEM or WB stages other than correct forwarding.

4. Verification

- Use the same test programs as before (that store 25 at address 100 upon success).
- Ensure all tests pass identically in RVX10-P.
- Check that the pipeline executes multiple instructions concurrently (view in GTKWave).
- Compare number of total cycles with the single-cycle core for identical programs.

5. Performance Counters (Optional Bonus)

Add simple counters:

```
reg [31:0] cycle_count, instr_retired;
always @(posedge clk) cycle_count <= cycle_count + 1;
if (RegWriteW) instr_retired <= instr_retired + 1;
```

Compute average CPI = cycle_count / instr_retired.

Deliverables

- `src/riscvpipeline.sv` – top-level module.
- `src/controller.sv`, `src/datapath.sv`.
- `src/hazard_unit.sv`, `src/forwarding_unit.sv`.
- `tests/rvx10_pipeline.hex`.
- `tb/tb_pipeline.sv` – self-checking testbench.
- `docs/REPORT.md` – design description, hazard logic, waveform screenshots.

Evaluation Rubric

Aspect	**Weight**	**Description**
Functional correctness	40%	Passes all RV32I + RVX10 tests
Hazard handling	25%	Correct forwarding, stall, and flush
Code quality	15%	Clean modular organization, comments
Waveform/report	10%	Shows pipeline overlap clearly
Optional features	10%	Counters or extra diagnostic views

Checklist Before Submission

- Single test program finishes with store of **25** to memory address **100**.
- `x0` register remains constant at zero.
- Forwarding verified for back-to-back ALU ops.
- One-cycle stall correctly inserted for load-use.
- Pipeline flush works for taken branches.

Public Repository and Visibility

To promote open learning and professional visibility:

- Host your complete project in a public **GitHub repository** named `rvx10p_<rollno>` or similar.
- The repository must include:
  - `/src` – Verilog/SystemVerilog source files
  - `/tb` – testbench and memory images
  - `/docs` – README, design report, and waveform screenshots
- Include a short project description in your README:

  > *"RVX10-P: A Five-Stage Pipelined RISC-V Core supporting RV32I + 10 Custom ALU Instructions, developed under the course* ***Digital Logic and Computer Architecture*** *taught by Dr. Satyajit Das, IIT Guwahati."*

- After publishing, make a brief **LinkedIn post** summarizing your project and **tag Dr. Satyajit Das (www.linkedin.com/in/satyajit-das-1105a86a) and mention 'IIT Guwahati – RVX10 Core Project'**. This provides teaching credit and professional visibility for your work.

Expected Outcome

A working five-stage pipelined processor supporting all RV32I and RVX10 instructions with correct results and visibly overlapped instruction execution, demonstrating lower CPI compared to your single-cycle RVX10 core.