ECE 251: Computer Architecture

Week 11: Pipelining & Hazard Resolution

Prof Rob Marano

Spring 2026

1. Transition to True Hardware Pipelining

- **Core Concept**: Multiple instructions are actively executed simultaneously across discrete hardware zones (Overlap Execution).
- **The Goal**: Process N instructions in N clock cycles, driving Cycles Per Instruction (CPI) down to 1.0.
- **The Reality**: The global machine clock is constrained strictly by the slowest hardware stage (e.g., Memory retrieval).
- **The Trade-off**: Throughput dramatically increases over Single-Cycle, but the physical duration applied to an individual isolated instruction (Latency) slightly increases due to pipeline logic overhead.

2. The Amdahl's Law Limit

- Pipelining overlaps execution, scaling speedup proportionally by hardware stages k.
- **Amdahl's Formula**: Speedup $= \frac{1}{(1-p)+\frac{p}{k}}$
- **MIPS Assembly Mapping (p)**: If $p = 0.8$, 80% of instructions execute flawlessly in parallel (e.g. bypassed ALU math, safely predicted branches).
- **The Penalty Mapping ($1-p$)**: 20% of execution triggers a hard structural stop via **Hazards** (e.g., `lw` memory stalls, pipeline flushes).
- **The Mathematical Ceiling**: Even as pipelining stages approach infinity ($k \to \infty$), the maximum physical speedup strictly caps at $\frac{1}{1-p}$. Hazards physically dictate CPU limits!

2b. Resolving Break-Even Parallelization

- **Problem**: An 8-stage pipeline ($k = 8$) *must* achieve a 4.0× speedup over a single-cycle implementation. Calculate the fraction of instructions (p) that must be completely free of hazards to hit this target.
- **Algebraic Execution**:

$$\begin{aligned} 4.0 &= \frac{1}{(1-p) + \frac{p}{8}} \\ (1-p) + \frac{p}{8} &= 0.25 \\ 1 - \frac{7p}{8} &= 0.25 \\ \frac{7p}{8} &= 0.75 \implies 7p = 6 \implies p \approx 85.71\% \end{aligned}$$

- **Result**: The compiler and hardware forwarders must orchestrate a flawless bypass environment for 85.71% of executed code.

3. Establishing Pipeline Registers

Because data must be preserved securely while instructions shift into the next functional stage, we partition the datapath using large D-Flip-Flop arrays called **Pipeline Registers**.

1. **IF/ID**: Holds fetched 32-bit instruction payload and sequential PC.
2. **ID/EX**: Holds Registry data (A, B), Sign-Extended Immediate, and target Register Addresses.
3. **EX/MEM**: Holds computed ALU output and target Write Destination address.
4. **MEM/WB**: Holds loaded RAM payload preparing for destination registry push.

3b. Pipelined Control Propagation

- **The Reality**: The Decoder calculates all 9 multiplexer bits (e.g. `RegDst`, `MemWrite`) inside the `ID` stage.
- **The Risk**: We **cannot** broadcast these control signals globally! Overlapping instructions would instantly cross-corrupt each other.
- **The Solution**: Control logic natively "rides the pipe". The 9 bits are wired into the `ID/EX` pipeline register boundary explicitly alongside the data.
- **Asynchronous Unpacking**: As instructions progress, Control pieces correctly split off. 4 bits split locally into the ALU (EX stage), 3 bits buffer down the boundary and split into Memory (MEM stage), and the final 2 bits survive until WriteBack.

4. SystemVerilog: The Pipeline Register Stage

```
// SystemVerilog ID/EX Pipeline Register DFF
module id_ex_reg (input  logic        clk, reset,
                  input  logic        RegWriteD, MemWriteD, // Control
                  input  logic [31:0] RD1D, RD2D, ImmD,     // Data
                  input  logic [4:0]  RsD, RtD, RdD,        // Targeting
                  output logic        RegWriteE, MemWriteE,
                  output logic [31:0] RD1E, RD2E, ImmE,
                  output logic [4:0]  RsE, RtE, RdE);

  always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
      RegWriteE <= 0; // Evaluate No-Op flush...
    end else begin
      RegWriteE <= RegWriteD;
      RD1E      <= RD1D;
```

5. Data Hazards: RAW (Read-After-Write)

- **Data Hazards** manifest when a physical target instruction attempts to consume algebraic variables generated by an older instruction still evaluating inside the pipeline.
- Example (Read-After-Write RAW calculation bypass):
  - `add $s0, $t0, $t1` → `$s0` locked for modification via ALU
  - `sub $t2, $s0, $t3` → Immediately demands `$s0` for subtraction input
- **Solution**: We do not wait for the fifth stage (WB). The ALU calculates `$s0` locally in stage 3 (EX). We algorithmically **Forward** (bypass) the data natively backwards utilizing multiplexers.
- **Physical Stalling (The Bubble)**: If the data isn't algebraically ready (e.g. a memory load `lw` taking 4 stages), forwarding is physically impossible in time. We must stall the pipeline by synthesizing an inert algorithmic bubble! *Did you know? MIPS architecture natively invokes the raw code* `0x00000000` *translated identically as* `sll $zero, $zero, 0` *to safely delay execution 1 cycle without corrupting registries!*

5b. Mathematical Pairs of Hazard Conditions

Before physically bypassing logic, the CPU must mathematically detect exact execution collisions natively sitting inside the `EX` hardware stage:

- **EX Hazards (1-Cycle Delay)**: The physically preceding instruction modifies a register we demand right now.
  - `1a: EX/MEM.Rd == ID/EX.Rs`
  - `1b: EX/MEM.Rd == ID/EX.Rt`
- **MEM Hazards (2-Cycle Delay)**: An older instruction sitting securely two boundary registers deep is modifying a register we need.
  - `2a: MEM/WB.Rd == ID/EX.Rs`
  - `2b: MEM/WB.Rd == ID/EX.Rt`
- *Caveat*: The Hardware physically verifies `RegWrite == 1` and `Rd != $0` before forcefully throwing multiplexer routing pins!

6. SystemVerilog: The Hazard / Forwarding Unit

```
module hazard_unit (input  logic [4:0] RsE, RtE,   // Current Stage Src Regs
                    input  logic [4:0] WriteRegM,  // Prev Stage Dest Reg
                    input  logic       RegWriteM,  // Validation Flag
                    output logic [1:0] ForwardAE, ForwardBE);

  // Forwarding evaluation for ALU Source Port A
  always_comb begin
    if ((RegWriteM) && (WriteRegM != 0) && (WriteRegM == RsE)) begin
        // The execution source register mathematically collides with
        // the pending write destination of the previous cycle.
        ForwardAE = 2'b10; // Trigger MUX to route ALUOutM backwards to ALU In
    end else begin
        ForwardAE = 2'b00; // Use clean baseline Register File extraction
    end
  end
```

7. Control Hazards: Pipeline Penalties

- **The Problem**: The processor blindly fetches sequential code (`PC + 4`). Upon striking a branch instruction (`beq`), the true execution path is unknown until the ALU evaluates conditions inside the third EX phase.
- **Software Fix (See MIPS Run - Branch Delay Slot)**: Early architectures conserved transistor budgets by deliberately exposing pipelines to compilers. The hardware mechanically executed the instruction directly following the branch. Software C-Compilers had to meticulously pack this "Delay Slot" with safe, independent execution logic or risk corrupted state vectors.
- **Hardware Evolution (Predict & Flush)**: By predicting *Branch Not Taken*, the datapath optimistically marches forward. If evaluated incorrectly as *Taken*, the pipeline physically halts, triggers the `reset` line on `IF/ID` bounds, and structurally collapses the rogue payload into harmless `0's` (NOP bubbles).

8. Exceptions & Interrupts

- Pipelining inherently disrupts orderly procedural progression. A division-by-zero execution fault or hardware interrupt must freeze standard datapath fetching operations immediately.
- The logic traps the specific active `PC` sequence inside the `EPC` (Exception Program Counter) tracking register, explicitly squashes pipeline operations locally, and drives processing towards the local OS Exception Handler boot protocol vector mapping mapping.

8b. The Architect's Synthesis: Designing a Pipeline

At the highest computational architecture level, we balance raw clock speed against physical structural constraints using three main techniques:

- **Slicing the Datapath**: Physically divide ALU logic using isolating *D*-Flip-Flop boundaries to overlap instruction phases flawlessly.
- **Propagating Synchronous Control**: Hardware mechanically pushes dynamic Boolean multiplexer routing combinations laterally across boundary boundaries identically alongside corresponding payload execution packets.
- **Resolving Collisions Natively**: Deploying explicitly wired Hazard Unit blocks running concurrently with live executions to physically throw physical algorithmic bypass overrides or to manually force procedural pipeline "Bubbles".

8c. SystemVerilog 5-Stage Pipelined Synthesis

- **Physical Codebase**: Developed within `pipelined_cpu/`, mapping the exact MIPS32 framework against rigorous SystemVerilog execution limitations.
- **Register Pipeline Boundaries**: The datapath utilizes massive array-bound `always_ff` flip-flops to discretely separate `IF/ID`, `EX/MEM`, and `MEM/WB` hardware.
- **Active Structural Resolvers**: The `hazard.sv` module actively scans execution states to natively map `forwardaE` datapath shortcut overlaps, or directly forces architectural NOP bubbles (`StallD`, `FlushE`) against Load-Use collisions.
- **Performance Conclusion**: Single-cycle maxes at 800 MHz ($CPI = 1.0$), Multi-cycle leaps to 3.3 GHz ($CPI = 4.0$). Pipelining achieves the massive 5.0 GHz frequency while restoring an optimal theoretical $CPI = 1.0$ through parallel overlap.
- **Run locally**: `make clean all` tracks complete output cycles automatically.

9. Week 11 Summary

- **Pipeline Strategy**: Divide datapath execution linearly over discrete logical D-flip-flop boundary constraints natively driving isolated logic stages concurrently over absolute clock phases.
- **Hazard Physics**: Resolving algebraic resource overlaps necessitates explicit mathematical handling logic physically designed across tracking registers vs core ALU forwarding constraints.
- **Amdahl's Cap**: Hazards define the sequential ceiling $1 - p$ capping overall pipeline scaling potential.
- **Next Session**: Extending into Advanced Interrupts and Memory Hierarchy evaluations defining system-level architectural caching logic frameworks.