Lecture 06

Pipelining, Branch Prediction & Parallelism

5-Stage Pipeline · Hazards · Branch Prediction · Flynn's Taxonomy · ILP

SENG 21213 · Computer Architecture & Operating Systems

Learning Objectives — By the end of this lecture you should be able to:

- 1. Describe the operation of each stage in the 5-stage RISC pipeline
- 2. Calculate pipeline speedup and explain the impact of hazards on ideal CPI
- 3. Identify structural, data (RAW), and control hazards and describe specific resolution techniques
- 4. Trace 2-bit saturating counter branch predictor state transitions and count mispredictions
- 5. Classify processor organisations using Flynn's taxonomy (SISD, SIMD, MISD, MIMD)
- 6. Describe superscalar, OOO execution, VLIW, and SMT techniques for exploiting ILP

Section 1 · Instruction Pipelining

📖 **Stallings Reference**

- Chapter 14: Instruction Level Parallelism and Superscalar Processors
- Chapter 12: Processor Structure and Function — Pipeline organisation

Pipelining is the technique of decomposing instruction execution into discrete stages and overlapping the execution of multiple instructions. While instruction N executes, instruction N+1 decodes, and instruction N+2 fetches — achieving up to 1 instruction per clock cycle in steady state.

1.1 The 5-Stage RISC Pipeline

Stage	Abbreviation	Hardware Used	Operation
Instruction Fetch	IF	Instruction cache, PC register	MAR ← PC; IR ← I-Cache[MAR]; PC ← PC+4
Instruction Decod	ID	Control unit, register file	Decode opcode; read rs, rt from register file; sign-extend immediate

Stage	Abbreviation	Hardware Used	Operation
e			
Execute	EX	ALU, forwarding unit	ALU computes: result, memory address, or branch target; flags set
Memory Access	MEM	Data cache	Load: D-Cache[ALUresult]; Store: D-Cache[ALUresult] ← register; others: pass-through
Write Back	WB	Register file	Write ALU result or loaded data to destination register rd

1.2 Pipeline Performance

★ Pipeline Throughput Analysis

- Ideal CPI = 1 (one instruction completes per cycle in steady state)
- Start-up latency: first result appears after k cycles (k = number of stages = 5)
- Total cycles for N instructions: k + (N−1) ≈ N for large N
- Theoretical speedup: k × (for large N)
- Actual speedup < k due to: hazards, branch mispredictions, memory stalls
- Example: 100 instructions on 5-stage pipeline: 5 + 99 = 104 cycles. Without pipeline: 500 cycles. Speedup = 500/104 = 4.8× (not 5× due to fill/drain)

1.3 Pipeline Hazards — Three Types

Structural Hazards

Two instructions need the same hardware resource at the same time. Classic example: a unified memory — if instruction N is in MEM stage (data access) and instruction N+3 is in IF stage (instruction fetch), they both need memory simultaneously.

✓ Solution — Structural Hazard

- Separate instruction cache (L1-I) from data cache (L1-D) — Modified Harvard architecture. Each has its own port, eliminating the IF/MEM conflict. This is why all modern CPUs have split L1 caches.

Data Hazards (RAW — Read After Write)

An instruction needs a result that a previous instruction has not yet written to the register file. The most common hazard type.

```
ADD R1, R2, R3    # EX stage: R1 ← R2 + R3 (result written in WB at cycle 5)
SUB R4, R1, R5    # ID stage: needs R1 — but R1 not written yet! → stall 2 cycles
AND R6, R1, R7    # also needs R1 — stall 1 cycle (if no forwarding)
```

Solution	Mechanism	Latency	Hardware Cost
Pipeline stalls (bubbles)	Detect hazard in ID; insert NOP cycles; wait for WB	2 cycles stall per RAW for EX→EX	Hazard detection logic only
Forwarding (bypassing)	Route EX/MEM result directly to ALU input (skips WB/ID)	0 cycles for EX→EX; 1 cycle for MEM→EX	Forwarding muxes and control logic
Out-of-order scheduling	Re-order independent instructions to fill slots	0 cycles (compiler or hardware)	Complex hardware or smart compiler

Control Hazards (Branch Hazards)

A conditional branch changes the PC — but the CPU has already fetched 1–2 subsequent instructions that may be wrong.

Strategy	Mechanism	Penalty on Mispredict	Accuracy
Flush on branch	Always flush pipeline on branch; wait for resolution	2–3 cycles always	No prediction needed — very slow
Predict not-taken	Continue fetching sequential instructions; flush if branch taken	2–3 cycles (only if taken)	~40–60% for typical programs
Predict always-taken	Always predict branch taken; squash if not taken	2–3 cycles (only if not taken)	~60–70% for loops
Delayed branch (MIPS)	Fill branch delay slot with useful instruction (compiler responsibility)	0 cycles (useful work done)	Always correct — no prediction
2-bit saturating counter	4-state predictor remembers recent history; very stable for loops	2–3 cycles on mispredict	~90% accuracy
Tournament predictor (modern)	Meta-predictor selects between global and local predictors	2–15 cycles on mispredict	~98% accuracy

1.4 The 2-Bit Saturating Counter Predictor

State machine with 4 states. Biased toward the last prediction — one misprediction does NOT flip the prediction:

State	Prediction	After Taken	After Not Taken
Strongly Taken (11)	Taken	Strongly Taken (11)	Weakly Taken (10)
Weakly Taken (10)	Taken	Strongly Taken (11)	Weakly Not Taken (01)
Weakly Not Taken (01)	Not Taken	Weakly Taken (10)	Strongly Not Taken (00)
Strongly Not Taken (00)	Not Taken	Weakly Not Taken (01)	Strongly Not Taken (00)

Section 2 · Parallelism Beyond the Single Core

2.1 Flynn's Taxonomy

Class	Instruction Streams	Data Streams	Description	Example
SISD	1	1	Classic sequential processor. Executes one instruction on one datum per cycle.	Single-core CPU (pre-1990)
SIMD	1	Many	ONE instruction applied to MULTIPLE data elements simultaneously. Vector / array processing.	GPU (CUDA cores), x86 SSE/AVX, ARM NEON
MISD	Many	1	Multiple instructions on the same data. Theoretical; few practical examples.	Fault-tolerant aerospace computers
MIMD	Many	Many	Most general: multiple processors each running independent instruction and data streams.	SMP, multicore, cloud clusters

2.2 Instruction-Level Parallelism (ILP) Techniques

Technique	Description	Hardware Complexity
Superscalar	Fetch and issue multiple independent instructions per cycle to replicated functional units (e.g. 2 ALUs).	Medium — must detect independence between instructions in-flight
Out-of-order (OOO) Execution	CPU executes instructions in an order different from program order when data dependencies allow. Results committed in order.	High — reservation stations, reorder buffer, register renaming
Very Long Instruction Word (VLIW)	Compiler packs multiple independent operations into one wide instruction word. No runtime hazard detection.	Low hardware; high compiler complexity — brittle to pipeline changes
Speculative Execution	Execute instructions before knowing if they are needed (after branch). Discard results if wrong path.	High — branch predictor, commit/rollback logic
Simultaneous Multithreading (SMT)	One physical core appears as 2+ logical CPUs; runs instructions from multiple threads each cycle to hide latency.	Medium — shared functional units; separate register sets per thread

Section 3 · Practice Exercises

✎ Exercise 6.1 [12 marks] — Pipeline Performance

- (a) [4] A 5-stage pipeline (IF/ID/EX/MEM/WB) runs at 1 GHz. A 10-instruction sequence has: 2 RAW hazards requiring 2-cycle stalls each (no forwarding); 1 branch hazard requiring a 3-cycle flush. Calculate total execution cycles and effective CPI.

- (b) [4] With forwarding enabled, the 2-cycle stalls reduce to 0-cycle stalls (data arrives via forwarding). The branch flush remains 3 cycles. Recalculate total cycles and CPI. What is the speedup from forwarding?
- (c) [4] A loop body consists of 8 instructions. The loop runs 1000 iterations. The loop branch is always taken (loop condition) except on the final iteration. With the 2-bit predictor starting in Strongly Not Taken state: trace the first 3 iterations' branch predictions. How many mispredictions occur in 1000 iterations total? Calculate the total branch penalty cycles.

✎ Exercise 6.2 [10 marks] — Data Hazard Analysis

- (a) [6] Identify all RAW hazards in the following MIPS code. For each, state: (i) which instructions conflict, (ii) which register, (iii) stall cycles without forwarding, (iv) stall cycles with full forwarding: ADD $t1, $t2, $t3 LW $t4, 0($t1) SUB $t5, $t1, $t4 OR $t6, $t5, $t4 AND $t7, $t1, $t6
- (b) [4] Reorder the instructions in (a) to eliminate as many stalls as possible (assume forwarding is available). What is the minimum number of stall cycles achievable? Which reordering did you apply?

✎ Exercise 6.3 [8 marks] — Flynn's Taxonomy and SIMD

- (a) [4] A GPU executes a vector addition C[i] = A[i] + B[i] for 8192 elements. Each SIMD unit processes 32 elements per instruction. The GPU has 64 SIMD units running in parallel. A scalar CPU takes 1 ns per addition. (i) How many SIMD instructions execute on the GPU? (ii) If each GPU instruction takes 5 ns and the GPU runs 64 units in parallel, what is the GPU execution time? (iii) What is the speedup over the scalar CPU?
- (b) [4] Explain why MISD (Multiple Instruction, Single Data) has almost no practical implementations. Describe one specific use case (hint: fault tolerance). Why is MIMD the most commercially important class in Flynn's taxonomy?