Advanced Computer Architectures - Notes - v0.12.0

260236

August 2025

Preface

Every theory section in these notes has been taken from two sources:

- Computer Architecture: A Quantitative Approach. [2]
- Pipelining slides. [3]
- Course slides. [4]

About:

GitHub repository

These notes are an unofficial resource and shouldn't replace the course material or any other book on advanced computer architectures. It is not made for commercial purposes. I've made the following notes to help me improve my knowledge and maybe it can be helpful for everyone.

As I have highlighted, a student should choose the teacher's material or a book on the topic. These notes can only be a helpful material.

For the midterm, I created a simple formulary:

Formulary

Contents

1 Pipelining **6**
- 1.1 Basic Concepts . . . 6
- 1.2 RISC-V Pipelining . . . 11
  - 1.2.1 Pipelined execution of instructions . . . 14
  - 1.2.2 Pipeline Implementation . . . 16
- 1.3 Problem of Pipeline Hazards . . . 18
  - 1.3.1 RISC-V Optimized Pipeline . . . 20
  - 1.3.2 Solutions to RAW Hazards . . . 23
- 1.4 Performance evaluation . . . 25

2 Control Hazards and Branch Prediction **28**
- 2.1 Conditional Branch Instructions . . . 28
- 2.2 Control Hazards . . . 30
- 2.3 Naïve Solutions to Control Hazards . . . 32
- 2.4 Intro to Branch Prediction . . . 35
- 2.5 Static Branch Prediction . . . 36
  - 2.5.1 Branch Always Not Taken . . . 37
  - 2.5.2 Branch Always Taken . . . 39
  - 2.5.3 Backward Taken Forward Not Taken (BTFNT) . . . 42
  - 2.5.4 Profile-Driven Prediction . . . 43
  - 2.5.5 Delayed Branch . . . 44
    - 2.5.5.1 From Before . . . 46
    - 2.5.5.2 From Target . . . 47
    - 2.5.5.3 From Fall-Through . . . 49
    - 2.5.5.4 From After . . . 51
- 2.6 Dynamic Branch Prediction . . . 52
  - 2.6.1 1-bit Branch History Table . . . 54
  - 2.6.2 2-bit Branch History Table . . . 56
  - 2.6.3 Branch Target Buffer . . . 58
  - 2.6.4 Correlating Branch Predictors . . . 60
    - 2.6.4.1 (1,1) Correlating Predictors . . . 62
    - 2.6.4.2 (2,2) Correlating Predictors . . . 64
  - 2.6.5 Two-Level Adaptive Branch Predictors . . . 66

3 Instruction Level Parallelism **68**
- 3.1 The problem of dependencies . . . 68
  - 3.1.1 Data Dependencies . . . 69
  - 3.1.2 Name Dependencies . . . 70
  - 3.1.3 Control Dependencies . . . 73
- 3.2 Multi-Cycle Pipelining . . . 74
  - 3.2.1 Multi-Cycle In-Order Pipeline . . . 75
  - 3.2.2 Multi-Cycle Out-of-Order Pipeline . . . 78
- 3.3 Dynamic Scheduling . . . 81
- 3.4 Multiple-Issue Processors . . . 83
  - 3.4.1 Introduction to Multiple-Issue Pipelines . . . 83
  - 3.4.2 Evolution Towards Superscalar Execution . . . 86
  - 3.4.3 Superscalar Processors . . . 89
  - 3.4.4 Static vs Dynamic Scheduling . . . 92

3.5 ILP Limitations & Alternatives . . . 95
3.6 Scoreboard: Dynamic Scheduling Algorithm . . . 97
3.6.1 Assumptions and Architecture . . . 97
3.6.2 Pipeline Stage Refinement . . . 99
3.6.3 Hazard Management (RAW, WAR, WAW) . . . 101
3.6.4 Control Logic and Stages . . . 103
3.6.5 Summary . . . 105
3.6.6 Scoreboard Data Structures . . . 107
3.6.7 In-Depth Execution Example . . . 109
3.7 Tomasulo's Algorithm . . . 134
3.7.1 Introduction . . . 134
3.7.2 Register Renaming: Static vs. Implicit . . . 136
3.7.3 Basic Concepts of Tomasulo's Algorithm . . . 138
3.7.4 Architecture . . . 140
3.7.5 Stages . . . 145
3.7.5.1 Stage 1: Issue . . . 145
3.7.5.2 Stage 2: Start Execution . . . 148
3.7.5.3 Stage 3: Write Result . . . 151
3.7.6 In-Depth Execution Example . . . 153
3.7.7 Tomasulo vs. Scoreboarding . . . 177
3.7.8 Register Renaming . . . 180
3.7.8.1 Introduction . . . 180
3.7.8.2 Loop Unrolling and Code Scheduling . . . 182
3.7.8.3 How Tomasulo Overlaps Iterations of Loops . . . 187
3.7.8.4 Tomasulo Loop Execution . . . 188
3.7.9 Reorder Buffer (ROB) . . . 216
3.7.9.1 Hardware-based Speculation . . . 216
3.7.9.2 Why ROB is really needed . . . 217
3.7.9.3 ROB as a Data Communication Mechanism . . . 220
3.7.9.4 Architecture . . . 222
3.7.9.5 Speculative Tomasulo Algorithm with ROB . . . 232

4 Performance Evaluation 240
4.1 Basic Concepts and Performance Metrics . . . 240
4.2 Amdahl's Law . . . 245
4.3 Pipelined Processors . . . 247
4.4 Memory Hierarchy . . . 251

5 VLIW (Very Long Instruction Word) 260
5.1 Introduction . . . 260
5.2 Data dependencies . . . 264
5.3 Statically Scheduled Processors . . . 267
5.4 Code Scheduling . . . 269
5.4.1 Scheduling Basics . . . 270
5.4.2 Dependence Graph and Critical Path . . . 271
5.4.3 ASAP Scheduling Algorithm (As Soon As Possible) . . . 273
5.4.4 List-Based Scheduling Algorithm . . . 275
5.4.5 Local vs Global Scheduling . . . 277
5.4.6 Local Scheduling Techniques . . . 279
5.4.6.1 Loop Unrolling . . . 279

5.4.6.2 Software Pipelining 283
5.4.7 Global Scheduling 286
5.4.7.1 Trace Scheduling 286
5.4.7.2 Superblock Scheduling 288

6 Advanced Memory **291**
6.1 Introduction 291
6.2 Principle of Locality 294
6.3 So, what is a Cache? 295
6.4 Cache Performance Metrics 297
6.5 Cache Architecture 298
6.5.1 Block Placement: *Where can a block be placed?* 299
6.5.2 Block Identification: *How is a block found?* 302
6.5.3 Replacement Strategy: *Which block should be replaced?* . 303
6.5.4 Write Strategy: *What happens on a write?* 305
6.6 Miss Penalty Reduction 309
6.7 Design Space of Cache 311
6.8 Cache Miss Classification 313
6.9 Improving Cache Performance 315
6.9.1 Reducing Miss Rate Techniques 317
6.9.1.1 Increasing Cache Size 317
6.9.1.2 Increasing Block Size 318
6.9.1.3 Increasing Associativity 320
6.9.1.4 Victim Cache 322
6.9.1.5 Pseudo-Associativity & Way Prediction 324
6.9.1.6 Hardware Prefetching (Instructions & Data) . . 329
6.9.1.7 Software Prefetching (Compiler-Controlled) . . . 332
6.9.1.8 Compiler Optimizations (for Cache Performance) 334
6.9.2 Reducing Miss Penalty Techniques 336
6.9.2.1 Read Priority over Write on Miss 336
6.9.2.2 Sub-block Placement 339
6.9.2.3 Early Restart & Critical Word First 340
6.9.2.4 Non-Blocking Caches 342

7 Exams **344**
7.1 2025 344
7.1.1 Midterm - May 5 344
7.1.2 February 11 378

Index **420**

1 Pipelining

1.1 Basic Concepts

Pipelining is a fundamental **technique** in computer architecture aimed at **improving instruction throughput by overlapping the execution of multiple instructions**. The main idea behind pipelining is to divide the execution of an instruction into distinct stages and process different instructions simultaneously in these stages. This approach significantly increases the efficiency of instruction execution in modern processors.

Understanding the RISC-V instruction set

Before delving into pipelining, it is essential to understand the **basic instruction set** of the RISC-V architecture. The instruction set consists of three major categories:

- **ALU Instructions (Arithmetic and Logic Operations)**
  - Performs **addition between registers**:

```
add rd, rs1, rs2
```

    Performs the addition between the values in registers `rs1` and `rs2` and stores the result in register `rd`.

$$\texttt{rd} \leftarrow \texttt{rs1} + \texttt{rs2}$$

  - Performs an **addition between a constant and a register**:

```
addi rd, rs1, 4
```

    Performs the addition between the value in register `rs1` and the value `4` and stores the result in register `rd`.

$$\texttt{rd} \leftarrow \texttt{rs1} + 4$$

- **Load/Store Instructions (Memory Operations)**
  - **Loads** data from memory:

```
ld rd, offset(rs1)
```

    Load data into register `rd` from an address formed by adding `rs1` to a signed `offset`.

$$\texttt{rd} \leftarrow M\left[\texttt{rs1} + \texttt{offset}\right]$$

  - **Stores** data in memory:

```
sd rs2, offset(rs1)
```

    Store data from register `rs2` to an address formed by adding *rs1* to a signed *offset*.

$$M\left[\texttt{rs1} + \texttt{offset}\right] \leftarrow \texttt{rs2}$$

- **Branching Instructions (Control Flow Management)**
  - **Conditional Branches**
    - Branch on equal:

```
beq rs1, rs2, L1
```

      Branch to the label `L1` if the value in register `rs1` is equal to the value in register `rs2`.

$$\texttt{rs1} = \texttt{rs1} \xrightarrow{\text{go to}} \texttt{L1}$$

    - Branch on not equal:

```
bne rs1, rs2, L1
```

      Branch to the label `L1` if the value in register `rs1` is not equal to the value in register `rs2`.

$$\texttt{rs1} \neq \texttt{rs2} \xrightarrow{\text{go to}} \texttt{L1}$$

  - **Unconditional Jumps**
    - Jump to the label (jump):

```
j L1
```

      Jump directly to the `L1` label.
    - Jump to the address stored in a register (jump register):

```
jr ra
```

      Take the value in register `ra` and use it as the address to jump to. So it is assumed that `ra` contains an address.

These basic instructions will be used throughout the course.

Execution phases in RISC-V

1. **IF (Instruction Fetch)**: The instruction is **fetched** from memory.
2. **ID (Instruction Decode)**: The instruction is **decoded**, and the **required registers are read**.
3. **EX (Execution)**: The instruction is **executed**, typically involving ALU operations.
4. **ME (Memory Access)**: For *load*/*store* instructions, this stage **reads from** or **writes to** memory.
5. **WB (Write Back)**: The **result is written back** to the destination register.

These five stages form the basis of the RISC-V pipeline.

Implementation of the RISC-V Data Path

The **RISC-V Data Path** is a fundamental component of the processor's architecture, responsible for **executing instructions efficiently by coordinating various hardware units**. It defines how instructions flow through different stages of execution, interacting with memory, registers, and the Arithmetic Logic Unit (ALU).

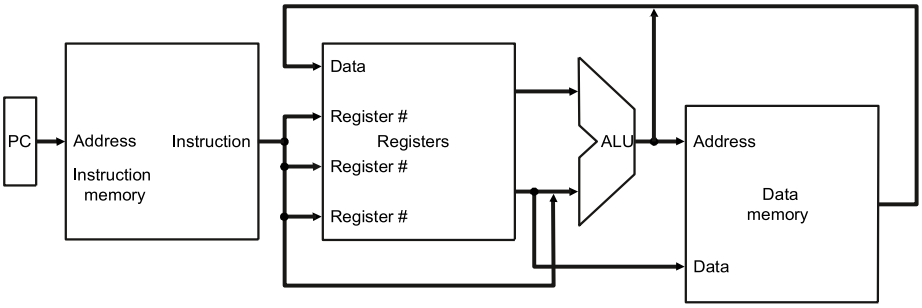


Figure 1: Generic implementation of the RISC-V Data Path.

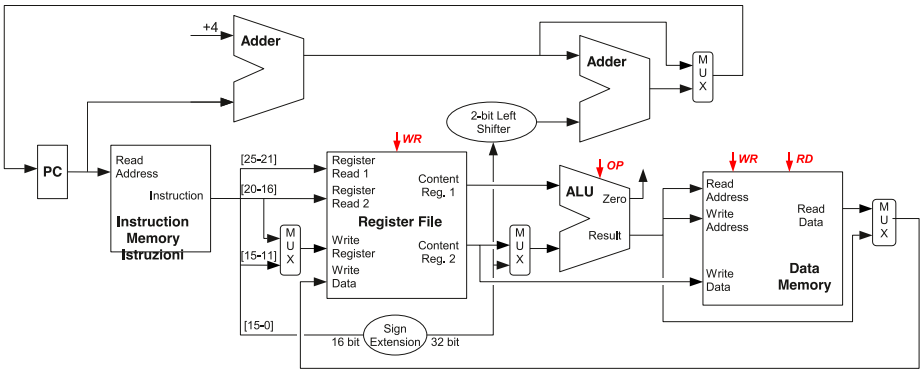


Figure 2: Specific implementation of the RISC-V Data Path.

Its fundamental components include:

- **Instruction Memory and Data Memory Separation**. RISC-V adopts a Harvard Architecture style, where the **Instruction Memory (IM)** and **Data Memory (DM) are separate**. This **prevents structural hazards** where instruction fetch and memory access could conflict in a single-memory design (this topic will be addressed later).
- **General-Purpose Register File (RF)**. It consists of **32 registers**, each **32-bit** wide. The register file has **two read ports** and **one write port** to support simultaneous read and write operations. This setup allows faster register access, which is crucial for pipelined execution.
- **Program Counter (PC)**. It **holds the address of the next instruction to be fetched**. Automatically increments during execution, typically by 4 bytes (for 32-bit instructions).

- **Arithmetic Logic Unit (ALU)**. **Performs arithmetic and logical operations** required by instructions. Inputs to the ALU come from registers or immediate values decoded from the instruction.

Other components that we can see in the general implementation of the RISC-V data path are:

- **Register File**. Stores temporary values used by instructions. Contains read ports (two registers can be read simultaneously for ALU operations) and write port (one register can be updated per clock cycle). The register file ensures high-speed execution of operations by reducing memory accesses.
- **Instruction Fetch (IF)**. The PC (Program Counter) retrieves the next instruction from Instruction Memory. The PC is incremented using an adder (`PC + 4`), ensuring sequential instruction flow.
- **Instruction Decode (ID)**. Extracts opcode (determines the instruction type), source and destination registers, immediate values (if present). It reads values from the Register File based on instruction requirements.
- **Execution (EX)**. The ALU performs arithmetic and logical operations. A multiplexer (MUX) selects the second operand: a register value (for R-type instructions) or an immediate value (for I-type instructions like `addi`). The ALU result is forwarded to the next stage.
- **Memory Access (ME)**. Load (`ld`) and Store (`sd`) instructions interact with data memory. Data is either **loaded from memory into a register** or **stored from a register into memory**.
- **Write Back (WB)**. The result from ALU or memory is **written back to the Register File**.

Example 1: Data Path Execution Example

Let's consider a simple RISC-V **load instruction** (`ld x10, 40(x1)`) passing through the data path:

1. **IF Stage**: Instruction Fetch
   - `PC` → Instruction Memory → `ld x10, 40(x1)` fetched
   - `PC` updated to `PC + 4`
2. **ID Stage**: Instruction Decode
   - Registers read: `x1` (base register for memory access)
   - Immediate value extracted: `40`
3. **EX Stage**: Execution
   - ALU calculates memory address: `x1 + 40`
4. **ME Stage**: Memory Access
   - Data is loaded from `M[x1 + 40]`
5. **WB Stage**: Write Back
   - Data stored in `x10`

1.2 RISC-V Pipelining

Pipelining is analogous to an assembly line in a factory. Instead of waiting for one instruction to complete before starting the next, **different instructions are executed simultaneously in different stages**.

If we consider a **non-pipelined execution**:

- Each instruction completes all five stages sequentially before the next instruction starts.
- If each instruction stage (IF stage, ID stage, etc.) takes, say, 2 nanoseconds, executing all stages of an instruction (IF, ID, EX, MEM, WB) takes 5 times 2 nanoseconds, then 10 nanoseconds. If we also want to execute 5 instructions, we need 10 nanoseconds times 5, then 50 nanoseconds!

Now, we consider a **pipelined execution**:

- Once the first instruction moves to the second stage, the next instruction starts in the first stage.
- The **pipeline becomes fully utilized** after the first few cycles, significantly **improving throughput**.

In an **ideal scenario**, a 5-stage pipeline should provide a speedup of 5× reducing execution time to:

$$(5 + 4) \times 2 \text{ ns } = 18 \text{ ns}$$

Where 5 are the steps of the first instruction, 5 are the steps of the last instruction, minus 1 because one step is already counted in the first instruction, so 4. Therefore, 9 is multiplied by 2 nanoseconds, the time taken by each stage. The result, 18 nanoseconds, is the time it takes the pipeline to execute 5 instructions in an ideal scenario.

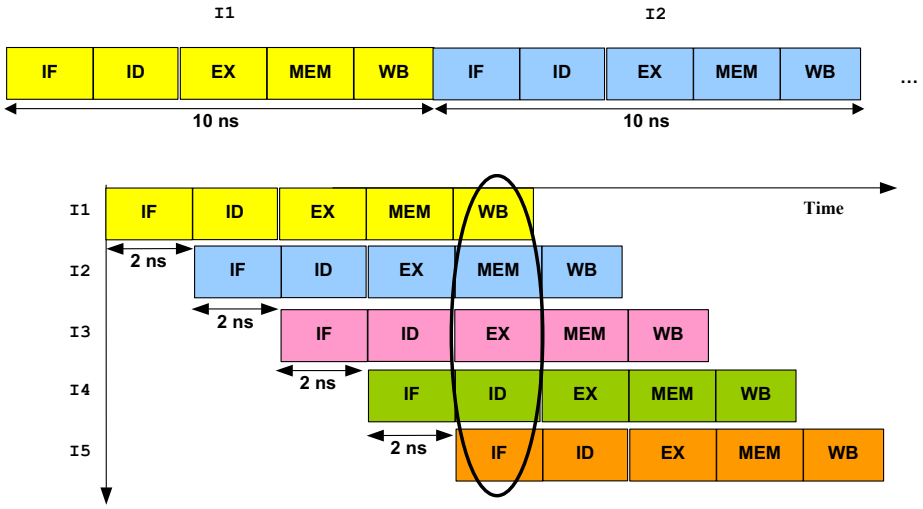


Figure 3: Sequential vs. Pipelining execution.

Pipeline Performance and Speedup

The ideal performance improvement from pipelining is derived from the fact that **once the pipeline is filled**, **a new instruction completes every cycle**. The key performance metrics include:

- **Latency (Execution Time)**: The total time to complete a single instruction does not change (sequential or pipeline).
- **Throughput (Instructions per Unit Time)**: The number of completed instruction per unit time significantly increases.
- **Speedup Calculation**
  - A non-pipelined CPU with 5 execution cycles of 2 ns would take 10 ns per instruction.
  - A pipelined CPU with 5 stages of 2 ns results in 1 instruction completing every 2 ns.
  - This gives a theoretical speedup of 5× (ideal case).

Unfortunately, real-world implementations are subject to **pipeline hazards** that reduce efficiency.

Understanding Pipelining Performance

Pipelining **improves instruction throughput** by allowing multiple instructions to be processed simultaneously in different stages. The **execution of an instruction is divided into 5 pipeline stages**:

1. IF (Instruction Fetch)
2. ID (Instruction Decode)
3. EX (Execution)
4. MEM (Memory Access)
5. WB (Write Back)

Each stage takes 2 ns (a *pipeline cycle*), meaning that an **instruction moves from one stage to the next every 2 ns**. Now, let's analyze the timeline of instruction execution:

Clock Cycle	IF	ID	EX	MEM	WB
1st (0-2 ns)	`I1`				
2nd (2-4 ns)	`I2`	`I1`			
3rd (4-6 ns)	`I3`	`I2`	`I1`		
4th (6-8 ns)	`I4`	`I3`	`I2`	`I1`	
5th (8-10 ns)	`I5`	`I4`	`I3`	`I2`	`I1`
6th (10-12 ns)	`I6`	`I5`	`I4`	`I3`	`I2`
7th (12-14 ns)	`I7`	`I6`	`I5`	`I4`	`I3`
8th (14-16 ns)	`I8`	`I7`	`I6`	`I5`	`I4`
9th (16-18 ns)	`I9`	`I8`	`I7`	`I6`	`I5`

Table 1: Pipelining timeline execution in an ideal case.

- The first instruction `I1` takes 5 (clock) cycles to complete, i.e., 10 ns.
- However, starting from cycle 5, a new instruction finishes every cycle (every 2 ns).
- In a non-pipelined system, each instruction would take 10 ns (5 stages $\times$ 2 ns each).
- In a pipelined system, once the pipeline is full, an instruction completes every cycle (every 2 ns), achieving a 5$\times$ speedup compared to the non-pipelined execution.

Thus, after an initial "fill" time (1st, 2nd, 3rd, 4th), **a new instruction completes every 2 ns** (from 5th to 6th, `I1` is finished; from 6th to 7th, `I2` is finished; from 7th to 8th, `I3` is finished), which is the duration of a single pipeline stage.

1.2.1 Pipelined execution of instructions

Each RISC-V instruction follows the five pipeline stages, but their interactions with the pipeline vary depending on the instruction type.

- **ALU Instructions** (e.g., `op $x, $y, $z`)

  These are register-based operations that do not require memory access. Since there is no memory operation, the instruction **bypasses the ME stage**.

Stage	Description
IF	Fetch instruction from memory
ID	Decode instruction, read registers `$y` and `$z`
EX	Perform ALU operation (`$x = $y + $z`)
ME	No memory access (skipped)
WB	Write the ALU result to `$x`

- **Load Instructions** (e.g., `lw $x, offset($y)`)

  These instructions retrieve data from memory and store it in a register. The **memory access stage** (ME) **is crucial** here since the instruction must fetch data from memory.

Stage	Description
IF	Fetch instruction from memory
ID	Decode instruction, read base register `$y`
EX	Compute memory address (`$y + offset`)
ME	Read data from memory
WB	Write data into destination register `$x`

- **Store Instructions** (e.g., `sw $x, offset($y)`)

  These instructions write data from a register into memory. Unlike `lw`, **store instructions do not require the WB stage**, as data is written directly into memory.

Stage	Description
IF	Fetch instruction from memory
ID	Decode instruction, read base register `$y` and source register $x
EX	Compute memory address (`$y + offset`)
ME	Write `$x` into memory at the computed address
WB	No write-back stage (skipped)

- **Conditional Branches** (e.g., `beq $x, $y, offset`)

  Branching introduces control hazards, as the pipeline needs to determine whether the branch is taken or not. Branches can introduce **stalls** due to dependencies on comparison results. This issue is typically mitigated using branch prediction.

Stage	Description
IF	Fetch instruction from memory
ID	Decode instruction, read registers `$x` and `$y`
EX	Compare `$x` and `$y`, compute target address
ME	No memory access (skipped)
WB	Update PC if branch is taken

This section breaks down how **different types of instructions behave in the pipeline**:

- ALU Instructions complete in the EX stage and do not use memory.
- Load Instructions require a memory access in the ME stage.
- Store Instructions write to memory instead of registers.
- Branch Instructions introduce control hazards because they may change the PC.

This means that **not all instructions behave the same** in the pipeline. Some instructions **skip certain stages** (e.g., stores do not have WB, ALU instructions skip ME), and some instructions **introduce potential problems** (e.g., branches can cause delays).

In conclusion, this section sets the stage for understanding pipeline stalls, forwarding, and hazard resolution techniques that are essential for designing high-performance processors.

1.2.2 Pipeline Implementation

The **RISC-V pipeline implementation** is designed to efficiently execute multiple instructions simultaneously, following the classical five-stage pipeline model:

1. IF (Instruction Fetch)
2. ID (Instruction Decode)
3. EX (Execution)
4. MEM (Memory Access)
5. WB (Write Back)

Each clock cycle, a new instruction enters the pipeline while previous instructions move to the next stage, allowing **five different instructions to be in execution at the same time**.

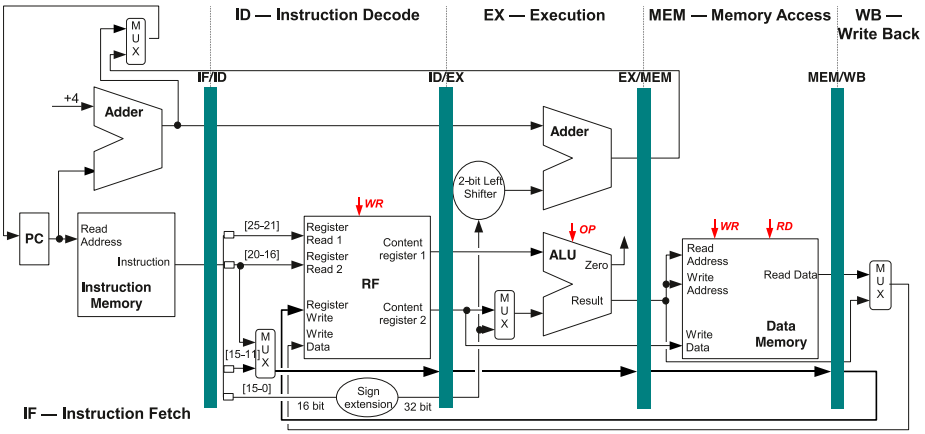


Figure 4: Structure of RISC-V pipeline.

Execution Stages and Pipeline Modules

Each stage of the pipeline corresponds to a specific hardware module in the CPU. The RISC-V pipeline is composed of five primary hardware modules:

- **Instruction Fetch (IF) Module**: Fetches instructions from instruction memory and updates the PC.
- **Instruction Decode (ID) Module**: Decodes the fetched instruction and reads register values.
- **Execution (EX) Module**: Performs arithmetic/logical operations in the ALU or computes memory addresses.
- **Memory Access (MEM) Module**: Reads from or writes data to memory.

- **Write Back (WB) Module**: Writes the computed result back into the register file.

Each module is responsible for a specific **stage of execution**, and together they allow overlapping execution of multiple instructions.

Pipeline Registers

To **maintain separation between stages**, **pipeline registers** are used (see Figure 4, page 16). These registers store intermediate results and ensure proper communication between stages:

- **IF/ID Register**: Holds fetched instruction and updated PC.
- **ID/EX Register**: Stores decoded instruction, read register values, and control signals.
- **EX/MEM Register**: Holds ALU results, destination register, and memory access information.
- **MEM/WB Register**: Stores memory data or ALU result to be written back to registers.

These pipeline registers **eliminate the need for re-fetching or re-decoding instructions** at each cycle, thus maintaining pipeline efficiency.

1.3 Problem of Pipeline Hazards

Assumptions Made

Until now, our discussion on the RISC-V pipeline implementation has relied on several key assumptions to simplify the analysis and focus on fundamental concepts. These **assumptions help in understanding the ideal case of pipelining** before introducing complexities like hazards and optimizations.

1. All instructions are independent, so there are no dependencies between them.
2. No branches or jumps that change execution flow.

This is a theoretical idealization, because in real-world scenarios, **hazards** (structural, data, and control) **interfere with smooth execution**. Also, our second assumption ignores **branch instructions** (`beq`, `bne`, `j`, `jr`), which **cause control hazards** that require branch prediction or pipeline flushing.

What is a Pipeline Hazard?

Now that we have understood the ideal execution of a RISC-V pipeline, we must discuss pipeline hazards, which are obstacles that prevent the pipeline from operating at maximum efficiency.

A **Hazard** (or conflict) is a phenomenon that occurs when the **overlapping execution of instructions in the pipeline changes the expected order of instruction execution**. This can lead to incorrect results or the **need to insert stalls** (*pipeline bubbles*), reducing performance.

In other words, hazards cause the next instruction in the pipeline to be delayed, which reduces the ideal throughput of 1 instruction per cycle. Thus, hazards disrupt the smooth flow of instructions and require techniques to resolve them.

Classes of Pipeline Hazards

- **Structural Hazards**: Attempt to use the same resource from different instructions simultaneously.

  Example: Single memory for both instruction and data access.

- **Data Hazards**: Attempt to use a result before it is ready.

  Example: Instruction depending on a result of a previous instruction still in the pipeline.

- **Control Hazards**: Try to make a decision about the next statement to execute before the condition is evaluated.

  Example: Conditional branch execution.

Structural Hazards

A **structural hazard** occurs when **multiple pipeline stages need to use the same hardware resource at the same time**.

✔ **Structural Hazard cannot be applied to RISC-V**. This is a great thing, because thanks to the Harvard Architecture, **RISC-V uses separate instruction and data memory**, and this adoption avoids structural hazards.

Control Hazards

A **control hazard** (section 2, page 28) occurs when the **pipeline does not know which instruction to fetch next, usually due to a branch or jump instruction**. It is discussed in the following sections.

Data Hazards

A **data hazard** (section 3.1, page 68) occurs when an **instruction depends on the result of a previous instruction that is still in the pipeline**.

There are several types of data hazards:

- **RAW (Read After Write)**. An instruction tries to read a register before a previous instruction writes to it.

  Example:

```
1 lw x2, 0(x1)
2 add x3, x2, x4
```

  The `add` instruction needs `x2`, but `x2` is still being fetched from memory in the MEM stage. Without hazard resolution, the processor would get the wrong value for `x2`.

- **WAR (Write After Read)**. A later instruction writes to a register before an earlier instruction reads it (rare in RISC).

- **WAW (Write After Write)**. Two instructions try to write to the same register in the wrong order.

1.3.1 RISC-V Optimized Pipeline

The **RISC-V optimized pipeline** introduces refinements that reduce stalls, improve data access, and enhance instruction throughput. The key optimizations include:

✔ **Efficient Register File Access**. In the standard RISC-V pipeline, register accesses **happen in two stages**:

- ID (Instruction Decode) → Reads register values.
- WB (Write Back) → Writes computed values back to registers.

Optimization: Read and Write in the Same Cycle

- In the optimized pipeline:
  - Register **writing** happens in the *first half* of the **clock cycle**;
  - While register **reading** happens in the *second half* of the **clock cycle**.
- This means an instruction can write its result to a register in WB, and the **next instruction can immediately read** that value in ID during the **same cycle**.

This optimization removes unnecessary stalls when an instruction immediately depends on a result written in the previous cycle.

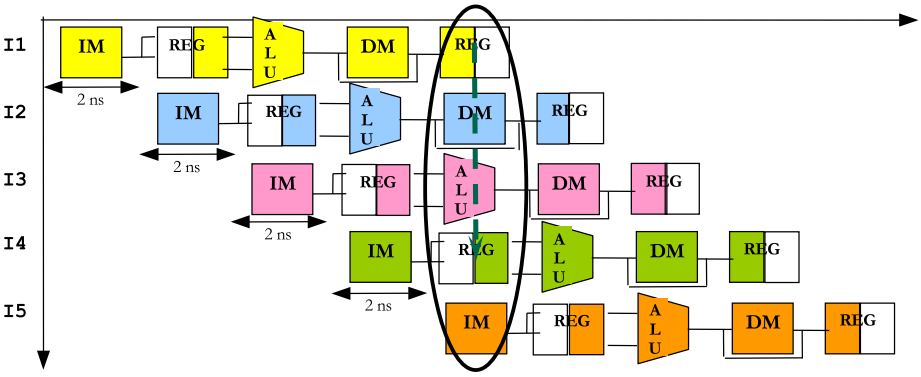


Figure 5: Visual **example** of an optimized pipeline; here the result (WB stage) of `I1` is written in the first half of the clock cycle and the read (ID stage) of `I4` is done in the second half of the clock cycle. So there is no hazards!

✔ **Forwarding (Bypassing) to Reduce Stalls**. **Forwarding** (also called **bypassing**) is a hardware technique that **eliminates stalls by providing ALU results directly to dependent instructions without waiting for the WB stage**. It is a possible solution for Data Hazards.

Forwarding Paths: To support forwarding, the **pipeline includes extra paths** that allow instructions to fetch values from intermediate pipeline registers instead of waiting for WB.

- **EX/EX Path**. Allows ALU results to be forwarded from **EX stage output to the next EX stage input**. Used when an **instruction depends on an arithmetic result of the previous instruction**.

Example 2: EX/EX Forwarding

```
sub x2, x1, x3   # Compute x2 = x1 - x3
and x12, x2, x5  # Use x2 immediately
```

Cycle	`sub x2, x1, x3`	`and x12, x2, x5`
1	IF	
2	ID	IF
3	EX	ID
4	MEM	*Stall*
5	WB	*Stall*
6		EX
7		MEM
8		WB

The `and` instruction **must wait until WB writes `x2` to the register file**. Two stall cycles are introduced and this wastes execution time.
Instead of waiting for WB, we forward the ALU result from the EX stage of `sub` directly to the EX stage of `and`.

Cycle	`sub x2, x1, x3`	`and x12, x2, x5`
1	IF	
2	ID	IF
3	EX	ID
4	MEM	EX (forwarded `x2` from EX)
5	WB	MEM
6		WB

In cycle 4, `and x12, x2, x5` gets the forwarded `x2` from the EX stage of `sub`, **removing stalls**.
This is EX/EX forwarding, taking ALU results from one EX stage directly into the next EX stage.

- **MEM/EX Path**. Forwards the ALU result from **MEM stage to EX stage**. Used when an **instruction depends on an ALU operation two cycles before**.

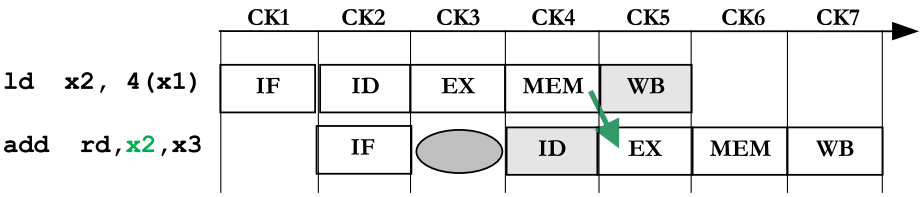


Figure 6: Example of MEM/EX path.

- **MEM/MEM Path**. Forwarding directly **between two memory operations in the MEM stage**. It **removes stalls in Load/Store dependencies**.

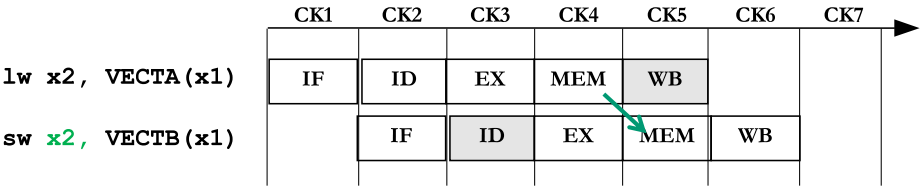


Figure 7: Example of MEM/MEM path.

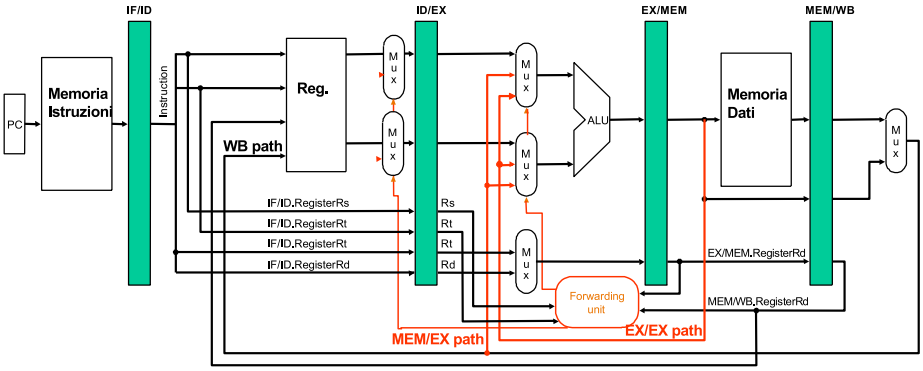


Figure 8: Implementation of RISC-V with Forwarding Unit.

1.3.2 Solutions to RAW Hazards

To handle RAW hazards, we can **use both** static (compile-time) and dynamic (hardware-based) **techniques**. These include:

- **Static (compile-time)**:
  - ✔ `nop` **insertion**: compiler adds empty instructions to delay execution.
  - ✔ **Instruction Scheduling**: compiler reorders instructions to avoid conflicts.
- **Dynamic (hardware-based)**:
  - ✔ **Pipeline Stalling (bubbles)**: inserts delay cycles when necessary.
  - ✔ **Forwarding (bypassing)**: uses intermediate values from the pipeline instead of waiting.

✔ *Static (compile-time) solution*: Inserting `nops` (naïve)

One simple way to handle RAW hazards is to **insert `nop` instructions manually between dependent instructions**. This gives the pipeline time to complete the write-back of the needed value.

Key takeaway of inserting `nops`:

- ✘ **Simple**, but inefficient because it wastes clock cycles. It should be the very last solution considered.
- ✘ Instead of using useful instructions, the **processor waits**, reducing performance.

Example 3: `nop` insertion

```
1 sub x2, x1, x3
2 nop               # Delay slot (bubble)
3 and x12, x2, x5   # Now x2 is ready
```

✔ *Static (compile-time) solution*: Instruction Scheduling

A more efficient technique is **instruction reordering**, also known as **compiler scheduling**. The **compiler reorders instructions to avoid data hazards** without inserting `nops`.

Key takeaway of instruction scheduling:

- Instruction reordering is a **compiler optimization**.
- ✔ It works well **if independent instructions are available**.
- ✘ In some cases, no independent instructions exist, so **stalling or forwarding is needed**.

Example 4: Instruction Scheduling

```
sub x2, x1, x3
# Independent instruction
# (can execute while sub is completing)
add x4, x10, x11
and x12, x2, x5   # Now x2 is ready
```

Instead of a `nop`, we insert `add x4, x10, x11`, which does not depend on `x2`. This keeps the pipeline utilized while avoiding RAW hazards.

Dynamic (hardware-based): Pipeline Stalling (Bubble Insertion)

When no independent instructions can be scheduled, the **hardware must stall the pipeline** by inserting a **bubble (stall cycle)**.

Key takeaway of pipeline stalling:

- ✗ Stalling is simple but **reduces performance** (pipeline sits idle).
- ✓ We **prefer forwarding** (next solution) instead of stalling.

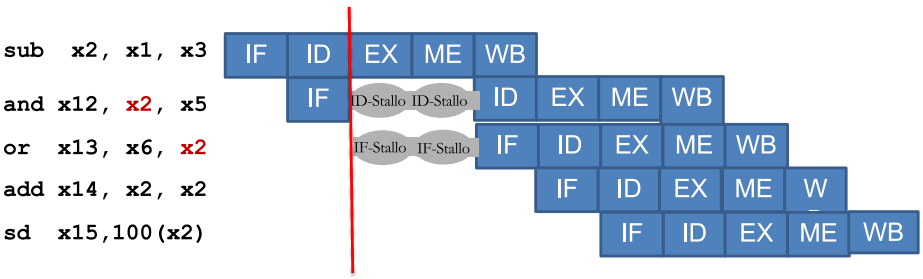


Figure 9: Example of inserting stalls.

Dynamic (hardware-based): Forwarding (Bypassing)

Forwarding is an optimized hardware technique that avoids pipeline stalls by **directly passing results between pipeline registers**. The entire implementation has already been explained on page 20.

Key takeaway of forwarding:

- ✓ **Forwarding is the best solution** because it eliminates stalls and maximizes performance.
- It **requires extra hardware** (MUX and control logic), but it significantly improves throughput.

1.4 Performance evaluation

Evaluating the performance of a pipelined processor is essential to understanding the impact of stalls, hazards, and instruction throughput. In an **ideal scenario**, a **pipeline achieves one instruction per cycle** (CPI = 1), but real-world execution includes pipeline stalls, which degrade performance.

Several **key metrics** are used to evaluate the efficiency of pipelining:

- **Instruction Count (IC)**. Represents the **total number of instructions executed**. Used as a basis for performance calculations.
- **Clocks Per Instruction (CPI)**. CPI measures the **average number of clock cycles required to execute one instruction**. Ideal CPI for a pipelined processor is 1, but hazards and stalls increase CPI.

$$\text{CPI} = \frac{\text{Total Clock Cycles}}{\text{Instruction Count (IC)}} \tag{1}$$

Where the total clock cycles is:

$$\text{Total Clock Cycles} = \text{IC} + 4 + \text{Stall Cycles} \tag{2}$$

Where +4 is the **fill time of the first instruction**. The +4 represents the initial pipeline fill time required before the pipeline reaches full execution throughput.

Example 5: Why is the Pipeline Startup Overhead +4?

A 5-stage pipeline (IF, ID, EX, MEM, WB) requires 4 extra cycles before the first instruction completes. Consider the following scenario:

Clock Cycle	IF	ID	EX	MEM	WB
1	I1				
2	I2	I1			
3	I3	I2	I1		
4	I4	I3	I2	I1	
5	I5	I4	I3	I2	I1
6	I6	I5	I4	I3	I2
7	I7	I6	I5	I4	I3

The first instruction (I1) requires 5 cycles to complete. The next instruction (I2) completes in cycle 6, and so on. After the first 5 cycles, the **pipeline reaches steady state**, completing 1 instruction per cycle (ideal scenario, no hazards).

- **Instruction Per Clock (IPC)**. IPC is the inverse of CPI:

$$\text{IPC} = \frac{1}{\text{CPI}} \tag{3}$$

Measures **how many instructions complete per clock cycle**.

- **Millions of Instructions Per Second (MIPS)**. Evaluates processor speed in terms of **millions of instructions executed per second**:

$$\texttt{MIPS} = \frac{f_{\text{clock}}}{\texttt{CPI} \times 10^6} \tag{4}$$

Higher **clock frequency** (f_{clock}) and lower CPI result in better MIPS.

Example 6: Performance Calculation

Given:

- Instruction Count (IC) = 5
- Stall Cycles = 2
- Clock Frequency = 500MHz

Metrics:

- Total Clock Cycles:

$$\text{Clock Cycles} = \texttt{IC} + \text{Stall Cycles} + 4 = 5 + 2 + 4 = 11$$

- CPI Calculation:

$$\texttt{CPI} = \frac{11}{5} = 2.2$$

- MIPS Calculation:

$$\texttt{MIPS} = \frac{500 \text{ MHz}}{2.2 \times 10^6} = 227$$

Without stalls, CPI would be 1 (ideal pipeline). But stalls increase CPI, reducing MIPS and overall efficiency.

Performance in Loops and Asymptotic Analysis

When evaluating **loops** or **long-running programs**, we use asymptotic performance metrics.

For a loop with:

- m **instructions** per iteration.
- k **stall** cycles per iteration.
- n **iterations**.

We have:

- **Clock Cycles per Iteration**:

$$\text{Clock Cycles per Iteration} = m + k + 4 \tag{5}$$

- `CPI` **per Iteration**:

$$\texttt{CPI}_{\text{iter}} = \frac{(m + k + 4)}{m} \tag{6}$$

- `MIPS` **per Iteration**:

$$\texttt{MIPS}_{\text{iter}} = \frac{f_{\text{clock}}}{\texttt{CPI}_{\text{iter}} \times 10^6} \tag{7}$$

For **large** n, the impact of pipeline startup delay (+4 cycles) is reduced:

- `CPI` **per Iteration**:

$$\begin{aligned} \texttt{CPI}_{\text{AS}} &= \lim_{n\to\infty} \frac{(\texttt{IC}_{\text{AS}} + \text{Stall Cycles}_{\text{AS}} + 4)}{\texttt{IC}_{\text{AS}}} \\ &= \lim_{n\to\infty} \frac{(m \times n + k \times n + 4)}{(m \times n)} \\ &= \frac{(m + k)}{m} \end{aligned} \tag{8}$$

- **Millions of Instructions Per Second (`MIPS`)**:

$$\texttt{MIPS}_{\text{AS}} = \frac{f_{\text{clock}}}{\texttt{CPI}_{\text{AS}} \times 10^6} \tag{9}$$

For **large programs**, startup stalls become negligible, and **performance depends mainly on stall cycles per iteration**. Minimizing k (stalls per iteration) is crucial to achieving high efficiency.

Why `CPI` is Greater than 1 in Real Pipelines

Even with an **optimized pipeline**, **real-world execution is affected by hazards**. Thus, actual CPI is always greater than 1, even in well-optimized designs.

2 Control Hazards and Branch Prediction

2.1 Conditional Branch Instructions

In pipelined processor architectures, control flow is not always linear, and decisions about the next instruction to execute are often dependent on certain conditions. This introduces the necessity for **conditional branch instructions**, particularly relevant in RISC-V architectures, where typical instructions include:

- `beq` (branch if equal): `beq rs1, rs2, L1`

  Transfers execution to the label `L1` if the contents of registers `rs1` and `rs2` are equal.

- `bne` (branch if not equal): `bne rs1, rs2, L1`

  Transfers control to `L1` if `rs1` and `rs2` hold different values.

These branch instructions are essential in implementing control structures such as loops, conditionals (`if`/`else`), and function returns.

At the hardware level, the **Branch Target Address (BTA)** plays a central role. This **address** represents **where the processor should continue execution if the branch is taken** (i.e., if the condition specified by the branch instruction evaluates as true). When the condition is satisfied, the processor **updates the Program Counter (PC)** with the BTA, thus redirecting the flow of instruction fetch.

Conversely, if the **condition is not satisfied**, the **branch is not taken**, and the processor continues **sequential execution**. In RISC-V, since instructions are generally 32 bits (4 bytes) long, the next instruction is fetched from `PC + 4`.

Understanding whether a branch is taken or not is crucial for instruction fetch in pipelined architectures. **Mispredicting** this can introduce **performance penalties**, which are addressed in detail through the study of control hazards and branch prediction techniques in later sections.

Execution of Branches in Pipelined Architectures

When executing a **branch instruction**, such as `beq rx, ry, L1`, the processor must **compare two registers** (`rx`, `ry`) to determine whether the branch should be **taken** (i.e., jump to label `L1`) or **not taken** (continue sequentially). In **RISC-V**, the **Branch Outcome** (Taken/Not Taken) and **Branch Target Address (BTA)** are **calculated during the EX stage** (when the ALU performs arithmetic and logical operations):

- EX Stage:
  - Compare `rx` and `ry` using the ALU.
  - Compute `PC + offset` to obtain the BTA.

- ME Stage:
  - Based on the comparison, update the PC to either `PC + 4` (if not taken) or `PC + offset` (if taken).

MIPS follows a similar structure but emphasizes that **branch decisions are finalized at the end of the EX stage**, with the **PC update happening at the ME stage**. This introduces a **delay in resolving the branch**, which becomes critical for understanding control hazards.

⚠ Implication for IF

Since new instructions are fetched every clock cycle, the processor needs to decide early which instruction to fetch next. However, with branches, this decision is **not immediately clear because the branch condition hasn't yet been evaluated**. The Program Counter (PC) **cannot be updated correctly until the branch outcome is known**, leading to potential pipeline stalls or incorrect instruction fetches.

2.2 Control Hazards

In a pipelined architecture, one of the primary challenges in achieving high performance is dealing with **Control Hazards**, also known as **branch hazards**. These **arise due to the presence of conditional branch instructions**, where the **processor must decide the next instruction to fetch before knowing whether the branch will be taken**.

What really causes Control Hazards?

To sustain the pipeline and avoid idle stages, a processor needs to fetch one instruction per clock cycle. However, with branch instructions, this becomes problematic because the **branch decision** (branch outcome) and the **branch target address** (BTA) are **not immediately available**. Specifically, **during the IF stage**, when the next instruction is fetched, the **processor still does not know whether the branch will be taken or not**, because this information typically becomes available later in the pipeline.

This leads to uncertainty:

- *Which instruction should be fetched after a branch?*
- *Should it be sequential instruction (`PC + 4`) or the instruction at the BTA?*

If the processor fetches the wrong instruction, it might **need to discard or "flush"** it later, **wasting valuable cycles**. Alternatively, the **processor may stall**, delaying the fetching of any instruction until the branch decision is known, which also hurts performance.

The key issue is this: in a pipeline, the **instruction stream needs to continue**, but the **correct path is unclear** until the branch condition is evaluated. Thus:

1. Either **wrong-path instructions are fetched** (requiring flushing later)
2. Or **pipeline stalls** are introduced (causing delay and loss of ideal speedup)

Key Takeaways: Control Hazards

- **Definition**: Pipeline hazard due to **uncertainty in branch outcome** during instruction fetch.
- **Cause**: The **branch condition is unresolved** when the next instruction must be fetched (IF stage).
- **Instructions Involved**: Conditional branches (`beq`, `bne`) and jumps, all **instructions modifying the PC**.
- **Pipeline Timing Conflict**: **BO and BTA known only in EX or later**, but instruction fetch **must occur every cycle**.

- **Main Problem**: **Cannot decide** whether to fetch **next sequential instruction** (`PC + 4`) or **BTA**.
- **Possible Outcomes**:
  - **Stall**: Delay fetch until branch resolved.
  - **Fetch wrong instruction** → flush.
- **Performance Impact**: Loss of **ideal pipelining speedup**; **reduced throughput** due to stalls or wasted fetches.
- **Goal of Solutions**: **Mitigate stalls** and **improve fetch accuracy** through early evaluation or prediction.

2.3 Naïve Solutions to Control Hazards

To manage control hazards in a simple and reliable way, one of the **earliest approaches** developed was the **conservative solution** of introducing **branch stalls**. The idea is straightforward: when a branch instruction enters the pipeline, the **processor stalls the pipeline until the branch decision is known and the correct next instruction can be safely fetched**.

How does the conservative solution work?

In the typical 5-stage pipeline, the Branch Outcome (i.e., whether the branch is taken or not) and the Branch Target Address (BTA) are usually resolved **at the end of the EX stage**. However, the **Program Counter** (PC) is actually **updated at the end of the ME stage**. This introduces a **delay of multiple cycles** between the branch instruction entering the pipeline and the point when the next instruction can be fetched with certainty.

To avoid fetching an incorrect instruction, the **processor simply pauses instruction fetch** for **3 clock cycles** after the branch instruction enters the pipeline. These are called **stalls**, essentially empty cycles where no new instructions enter the pipeline. Once the PC is correctly updated based on the branch outcome, instruction fetch resumes.

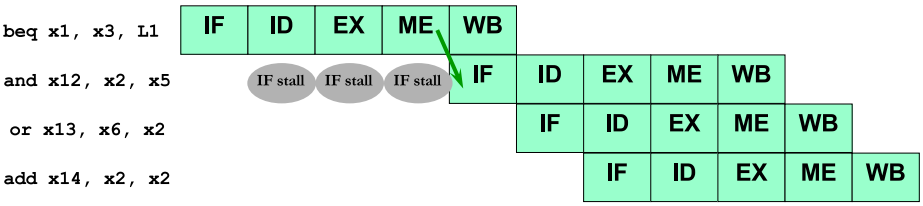


Figure 10: Example of stalls inserted in the pipeline to read the correct value after a branch condition.

Performance Impact

It's pretty obvious that if **each branch introduces a penalty of 3 cycles**, it will **significantly degrade performance**, especially in programs with frequent branching. Since pipelining aims to maximize instruction throughput, **this solution sacrifices speed for correctness**. In fact, it is called conservative because it does not attempt to guess or speculate about the branch outcome. Instead, it waits for certainty, favoring reliability over efficiency.

Can optimized evaluation at the ID stage improve performance?

Although the conservative solution degrades performance, it can be relatively optimized thanks to hardware optimization. Processor designers have introduced **hardware optimizations** that allow the **branch outcome (BO)** and the **branch target address (BTA)** to be **computed earlier** in the pipeline. Specifically, these computations can be **moved from the EX stage to the**

ID stage (during the decoding phase). This optimization is often referred to as **Early Branch Evaluation**.

How Early Branch Evaluation Works

To achieve this, the **Instruction Decode (ID)** stage must be **enhanced with additional hardware logic** that allows it to:

1. **Compare register values** (`rx` and `ry`) to determine the branch condition.
2. **Compute the BTA** using the **sign-extended offset** from the instruction and the current PC value.
3. **Update the PC** as soon as BO and BTA are known.

By doing this, **both BO and BTA are available at the end of the ID stage**, allowing the processor to update the PC immediately and fetch the correct instruction in the following cycle.

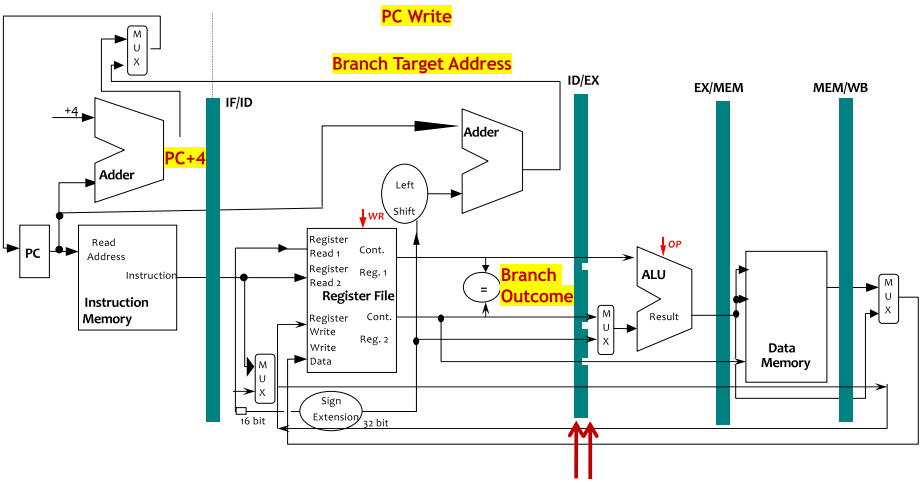


Figure 11: Hardware features modifications to allow early branch evaluation.

Hardware Overhead: Complexity vs. Performance

This optimization requires **more complex hardware**, as the ID stage now includes:

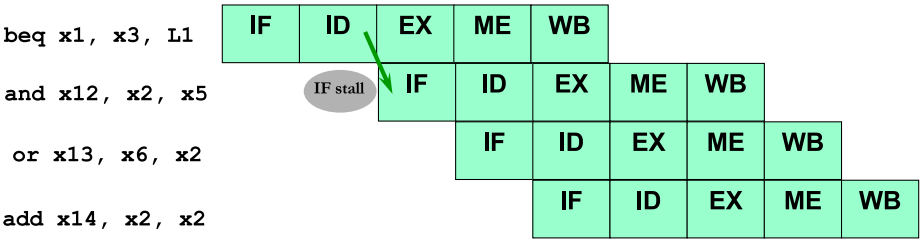
- **ALU logic** for comparison and addition.
- Additional **multiplexer and control signals** to direct the PC update.
- Expanded **data paths** for handling the offset and register values.

Effect on Pipeline Execution

Let's consider an example. In a pipeline using early evaluation:

- The processor **only stalls for 1 cycle** after a branch instruction, as opposed to the 3 cycles required by the conservative approach.
- This **one-cycle stall** allows the processor to fetch the correct instruction **immediately after the branch** is resolved.

The following diagram illustrates that the instruction fetch after the branch is delayed by only one stall cycle, resulting in a **smaller performance hit**.



Conclusion

In summary, by anticipating branch evaluation at the ID stage, the processor reduces the branch penalty to 1 cycle per branch. This is a significant improvement over the 3-cycle stall of conservative stalling. While it introduces **hardware overhead**, it offers a **better balance between performance and correctness** and serves as a stepping stone toward even more advanced techniques, such as branch prediction.

2.4 Intro to Branch Prediction

In modern computer architectures, achieving high performance requires efficient instruction-level parallelism (ILP)[1]. However, one of the **major obstacles to ILP is the occurrence of branch hazards**, which happen when the processor encounters a branch instruction (e.g., `if`, `for`, `while`) and cannot immediately determine which instruction to execute next. To mitigate the performance loss caused by these hazards, branch prediction is employed.

Branch Prediction is essentially a **speculative execution technique** where the **processor *guesses* the outcome of a branch instruction**, whether the branch will be taken (control jumps to the branch target) or not taken (execution continues sequentially), **before the actual result is known**. Instead of stalling the pipeline and waiting for the branch condition to be resolved, the **processor proceeds based on the predicted outcome**. If the prediction:

- ✔ Is **correct**, performance is preserved.
- ✖ Is **wrong** (a **misprediction**), the incorrectly fetched **instructions are flushed**, and **execution restarts at the correct address**, causing a **performance penalty**.

⁝☰ Branch Prediction categories

Branch prediction techniques are generally classified into two main categories:

- **Static Branch Prediction Techniques**. In this method, the **branch direction** (taken/untaken) is **decided at compile time** and **remains fixed during the program's execution**. Static prediction often relies on compiler heuristics or profiling data to guess likely outcomes. Since the behavior doesn't adapt to runtime changes, this **technique works best** when **branch outcomes** are **highly predictable** and **consistent** across executions.
- **Dynamic Branch Prediction Techniques**. Unlike static methods, dynamic prediction uses **hardware mechanisms to observe past branch behavior at runtime** and make predictions accordingly. The **prediction adapts to actual program execution**, making it more effective for applications with **complex or data-dependent control flow**. This method can dynamically switch its guess depending on the *branch history*.

It's important to note that in both static and dynamic techniques, the **processor must avoid updating its internal state** (registers, memory, etc.) **until the branch outcome is known with certainty**. This ensures speculative execution doesn't cause side effects in case of misprediction.

Additionally, **hybrid approaches** are possible, where static and dynamic predictions are combined to optimize performance further.

[1] **Instruction-Level Parallelism (ILP)**: A measure of **how many of the operations in a computer program can be performed simultaneously**. High ILP enables multiple instructions to be executed in parallel within a single processor cycle, exploiting the parallelism inherent in sequential instruction streams through techniques like pipelining, superscalar execution, and out-of-order execution.

2.5 Static Branch Prediction

Static branch prediction represents one of the **simplest approaches** to handling branch hazards. In this technique, the **prediction** regarding whether a branch will be taken or not is **made at compile time** and **remains unchanged throughout program execution**. This method **relies heavily on heuristics or compiler-generated hints**, which estimate the likely behavior of each branch without any consideration of the program's actual runtime behavior.

When does static branch prediction work well?

This approach is particularly effective in scenarios where the **branch behavior is stable and highly predictable**, such as in embedded or domain-specific applications. In such cases, the overhead and complexity of dynamic prediction mechanisms may not justify the potential benefits, making static prediction a practical alternative.

RISC-V assumption

A key architectural note here is the assumption that we are working with a **RISC-V processor**, which is **optimized for early branch evaluation during the Instruction Decode (ID) stage** (see more in section 2.3, page 33). This means that in RISC-V, the decision to predict a branch direction occurs early in the pipeline, minimizing the potential for instruction fetch delays if the prediction is accurate.

2.5.1 Branch Always Not Taken

The **Branch Always Not Taken** strategy is the simplest form of static branch prediction. It operates under the **assumption that the branch condition will never be satisfied**, i.e., the control **flow of the program will continue sequentially as if the branch is not taken**. As a result, instructions immediately following the branch in program order are fetched and executed **without any need to determine or access the Branch Target Address (BTA)**.

When is Branch Always Not Taken effective?

This approach is especially effective for **certain control flow patterns**, such as `if-then-else` structures where the `then` clause is more likely to be executed than the `else` clause. For example:

```
z = x + y;
if (z > 0)
    w = x;
else
    w = y;
```

Assuming `z` is typically positive, the branch is not taken because execution proceeds sequentially to `w = x`. This makes predict-not-taken a suitable and effective default strategy for such cases.

Implementation Details

The prediction is made **at the end of the Instruction Fetch (IF) stage, without calculating or knowing the BTA** (since the branch is always not taken and the next instruction to execute is the `PC + 4`, as always). This makes the approach **lightweight and efficient**.

Misprediction Case

If the actual **branch outcome** (BO) evaluated during the Instruction Decode (ID) stage is **not taken**, then the **prediction is correct**, and **no penalty cycles** are incurred. The pipeline proceeds as planned.

Otherwise, if the actual **branch outcome** (BO) turns out to be **taken**, then the **prediction was incorrect**, leading to a **misprediction penalty**. The processor must:

1. **Flush** the fetched instruction(s) after the branch (turned into `NOP`s).
2. **Fetch the instruction** at the Branch Target Address (**BTA**) and **restart execution from there**.

This results in a **one-cycle branch penalty**, which is minimal but still affects performance.

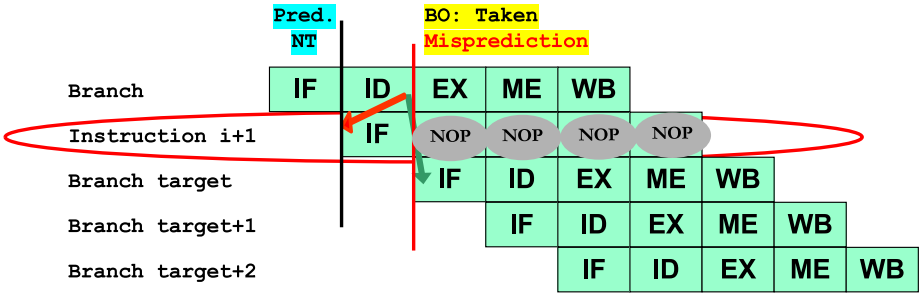


Figure 12: Branch Always Not Taken techniques failed and the processor must flush instruction `i+1` and restart execution from the BTA.

2.5.2 Branch Always Taken

This approach represents the dual case of the previous technique (page 37): it assumes that **every branch will be taken**, meaning the **control flow will jump to the branch target address** rather than continue sequentially. This method is especially **useful for backward branches**, which occur in **loops** such as `for`, `while`, and `do-while`, since these branches are typically **taken repeatedly during loop iterations**.

Implementation Challenge

Unlike the not-taken strategy, where the processor simply continues to `PC + 4`, the **taken strategy requires knowledge of the Branch Target Address (BTA)** at the Instruction Fetch (IF) stage. This is non-trivial because:

- The **BTA depends on the branch instruction's target**, which typically requires decoding.
- ✔ To solve this, we introduce a **Branch Target Buffer (BTB)**, a special hardware structure.

What is BTB and why do we need it?

The **Branch Target Buffer (BTB)** is a **specialized cache** in the processor **designed to predict the target address** of a taken branch instruction **before the branch condition is actually resolved**.

In Branch Always Taken, we assume that the program will jump to a new address (the Branch Target Address, BTA). However, this **BTA is not immediately known during Instruction Fetch (IF)** because it typically requires decoding the branch instruction (Instruction Decode, ID). To avoid delays, the **BTB remembers past branch target addresses**, allowing the processor to quickly predict where to jump when encountering a branch.

How does BTB work? Quickest explanation

- **BTB Structure**, it is a kind of lookup table or cache where:
  - `Key`: **address** of the **branch instruction** (the PC value where the branch resides)
  - `Value`: Predicted Target Address (PTA), i.e., where to jump if the branch is taken
- **BTB Lookup**: when fetching a branch instruction, the processor simultaneously queries the BTB via the branch PC.
  - ✔ If a **match is found** (*cache hit*), the **BTB immediately provides the Predicted Target Address** (PTA), and the **processor starts fetching from that address**, before knowing if the branch is actually taken.

✖ If a **no match** (*cache miss*), the processor might **default to sequential execution** (`PC + 4`) or **wait for the BTA** to be calculated, which causes delay.

Example 1: BTB and Branch Always Taken technique

Let's say:

- A loop branch at address `0x100` typically jumps to `0x80`.
- The BTB stores: `0x100` → `0x80` (key → value).

When the branch at `0x100` is fetched again:

- The BTB predicts the next instruction will be at `0x80` (taken).
- The processor **starts fetching from** `0x80`, **without waiting** to evaluate the branch condition.

If it turns out the **branch was not taken**, the processor **flushes the incorrect fetch** from `0x80` and resumes at `0x104`.

Correct Prediction Path

If the **branch** is indeed **taken**, and the **BTB correctly supplies the BTA**, the processor proceeds **without penalty**. **Execution continues from the target address** just as expected.

Misprediction Case

If the **actual outcome** is **not taken**, the **prediction is incorrect**:

1. The Instruction Fetched (IF) from the **target address** is **flushed** (`NOP`).
2. The **processor must fetch the sequential instruction** at `PC + 4`.
3. **One-cycle penalty** incurred, similar to the not-taken misprediction case.

When is this technique effective?

This method is **well-suited for loop constructs**, where **branches typically go backward** and are **taken with high probability**. For example:

- In a `do-while` loop, the branch is taken almost every time except the last iteration.
- Conversely, in forward branches like `if-then-else`, the branch is less likely to be taken, making this technique less effective.

This underscores that **branch direction** (forward or backward) **can influence the effectiveness of prediction strategies**.

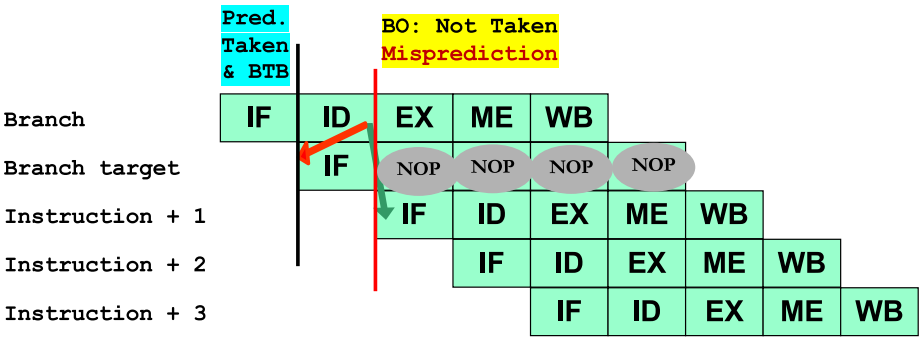


Figure 13: Branch Always Taken techniques failed and the processor must flush instruction `i+1` and restart execution from the BTA.

2.5.3 Backward Taken Forward Not Taken (BTFNT)

The **Backward Taken Forward Not Taken (BTFNT)** strategy represents a **refinement of static prediction** that uses a simple yet effective heuristic: the **direction of the branch**, whether it **jumps backward or forward** in memory, can be **used to predict its outcome**.

Prediction Rule

- **Backward-going branches** (i.e., branches where the target address is lower than the current PC) are predicted as **taken**.
  - These branches **often occur in loops**, where execution loops back to an earlier instruction (e.g., in `for`, `while`, or `do-while` constructs).
- **Forward-going branches** (i.e., target address is greater than the current PC) are predicted as **not taken**.
  - These branches typically correspond to `if-then-else` **constructs**, where the `else` path is **less probable** and **control usually proceeds sequentially**.

Why does this work?

The rationale behind BTFNT lies in **empirical observations**:

- **Loops** tend to execute **multiple times**, hence **backward branches are mostly taken**.
- **Conditional statements** often have **rarely taken `else` paths**, hence **forward branches are mostly not taken**.

Pros and Cons

✔ Simple to implement because BTFNT requires just a comparison of the **target address vs the current PC**:

- `target address < PC` ⇒ **predict taken**
- `target address > PC` ⇒ **predict not taken**

Also, better accuracy than uniform always-taken or always-not-taken, especially for mixed codebases.

✘ Not adaptive; fails for atypical control flows where direction doesn't align with expected behavior.

2.5.4 Profile-Driven Prediction

Profile-Driven Prediction is a static prediction technique that **uses empirical data from previous program executions** to guide the prediction of branch outcomes. Rather than relying solely on heuristics or branch direction, this method **leverages profiling to derive probabilistic insights** about how branches behave under typical conditions.

How does it work?

1. The **target application** is **executed multiple times beforehand**, **using diverse data sets** to simulate realistic execution scenarios.
2. During these early runs, the **behavior of each branch instruction is recorded**. Specifically, how often it was taken or not taken.
3. This **profiling produces statistics** for each branch, e.g., a pattern like:

   `T T T T T T T T NT NT NT`

   "Taken" is most probable.
4. Once the profiling is complete, the **compiler encodes a hint** directly into each branch instruction (e.g., in a dedicated **hint bit** in the instruction format):
   - `1`: if the branch is **usually taken**.
   - `0`: if the branch is **usually not taken**.

   This enables the **processor** to **consult the hint during execution** and predict accordingly, without requiring runtime monitoring.

Advantages

- ✓ Offers **higher accuracy than heuristics alone**, especially for applications with stable branch behavior.
- ✓ **No runtime hardware cost**, since prediction decisions are guided by **static hints**.

Limitations

- ✗ **Static nature**: Predictions don't adapt to runtime variability; changes in data patterns may invalidate the profile.
- ✗ Requires **extra compilation effort**: profiling and hint encoding add complexity to the build process.
- ✗ **Less effective** for programs with **input-dependent control flow**.

2.5.5 Delayed Branch

The **Delayed Branch** technique is a static scheduling approach where the **compiler** plays a central role in mitigating branch penalties. Unlike traditional branch prediction, which involves guessing the outcome of a branch, delayed branching **reorders instructions so that useful work is done regardless of the branch direction**.

Core Concept

When a **branch instruction is executed**, it typically introduces a **delay before the processor can determine where to fetch the next instruction**. During this delay (known as the **branch delay slot**), rather than letting the pipeline sit idle or fetch incorrect instructions, the **compiler schedules an independent instruction to execute in that slot**.

- The instruction in the **branch delay slot is always executed**, **regardless of whether the branch is taken or not**.
- This allows useful work to be completed during what would otherwise be a stall or pipeline bubble.

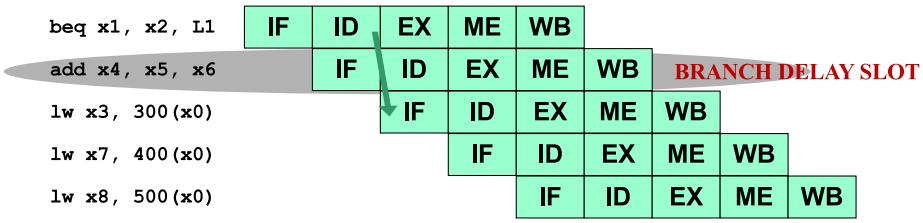


Figure 14: Example Scenario. In this case, the `add` instruction is scheduled after the branch, always executed, and does not affect the branch condition or outcomes.

Compiler Responsibility

A **critical task for the compiler** is **to find a valid and useful instruction** to place in the branch delay slot. The instruction must be:

- **Independent** from the branch decision.
- **Safe to execute** whether the branch is taken or not.

To guide this, the compiler can choose an instruction:

1. Section 2.5.5.1, page 46 - From **before** the branch.
2. Section 2.5.5.2, page 47 - From the **target** of the branch.
3. Section 2.5.5.3, page 49 - From the **fall-through path** (i.e., the sequential next instruction).
4. Section 2.5.5.4, page 51 - From **after** the branch.

We'll explore each of these four scheduling strategies step-by-step next.

Technique	Prediction Source	Complexity
Always Not Taken	Assume `PC + 4`	Very Low
Always Taken	Assume jump to BTA	Moderate (BTB)
BTFNT	Direction-based	Low
Profile-Driven	Prior run data	High (compile time)
Delayed Branch	Compiler scheduled	High (compiler)

Technique	Risk	Best for
Always Not Taken	Mispredict backward branches	`if-then-else`
Always Taken	Mispredict forward branches	Loops (backward branches)
BTFNT	Errors in irregular control flow	Mixed code (simple logic)
Profile-Driven	Profile mismatch	Stable behavior, performance-tuned
Delayed Branch	Wasted work if not efficient	RISC pipelines, e.g., MIPS processors

Table 2: Quick Comparison Table.

2.5.5.1 From Before

In the **"From Before"** strategy of delayed branch scheduling, the **compiler selects an instruction that appears *before* the branch** in program order and **moves it into the branch delay slot**. The selected **instruction must be independent of the branch decision** and safe to execute regardless of whether the branch is taken.

Key Characteristics

- The **instruction in the branch delay slot is always executed**.
- This instruction **will never be flushed**, since it is **guaranteed to execute** irrespective of the branch's outcome.
- **After the delay slot, execution continues normally**, either to the branch target or the fall-through instruction, depending on whether the branch is taken.

Example 2: From Before

Original code:

```
1 add x1, x2, x3
2 beq x2, x4, L1
3 [delay slot, stall]
```

After scheduling:

```
1 beq x1, x2, x3
2 add x1, x2, x3      # delay slot filled
```

Here, the `add` is originally before the `beq` and has **no dependency** on the branch condition. It is safely moved into the delay slot.

Pipeline Behavior

- **Branch Not Taken**
  - The instruction in the delay slot is executed.
  - Execution continues sequentially with the next instruction after the branch.
- **Branch Taken**
  - The delay slot instruction still executes.
  - Execution jumps to the branch target after the delay slot.

The **instruction moved to the delay slot is always executed**.

Advantages

- ✔ **No need for instruction flushing**: the delay slot instruction is always valid.
- ✔ **Efficiency**: reuses existing instructions from earlier in the program to **hide the branch delay**.

2.5.5.2 From Target

In the **"From Target"** strategy, the **compiler schedules an instruction from the *branch target* into the delay slot**. This technique is **useful when the branch is likely to be taken**, as the delay slot instruction corresponds to what would naturally execute next in that control path.

Key Characteristics

- The **delay slot contains an instruction from the branch target path**.
- This strategy is typically **used when the branch is taken with high probability**, such as in loops.
- **Challenge**: If the branch is **not taken**, this **instruction may be invalid** and might have **to be flushed** (if mispredicted), **or it must be safe to execute even if not needed**.

Example 3: From Target

Original code:

```
sub x4, x5, x6       # target instruction
add x1, x2, x3       # branch instruction
if x1 == 0 then      # branch condition
[delay slot, stall]
```

After scheduling:

```
add x1, x2, x3       # if branch taken, go here!
if x1 == 0 then      # branch condition
sub x4, x5, x6       # delay slot filled
```

Here, the `sub` instruction from the branch target is moved into the delay slot. If the branch is taken, execution proceeds smoothly. If not, we either flush `sub` or ensure it causes no side effects.

Pipeline Behavior

- **Branch Taken**
  - The **delay slot** instruction is **part of the intended control flow**.
  - Execution continues with the **next instruction in the target path**.
- **Branch Not Taken**
  - Delay slot **may need to be flushed** (as it's not part of the sequential path), or must be **safe to execute anyway** (no side effects or wasted computation).

Instruction Duplication

When we **move an instruction from the branch target into the delay slot**, we **still need to keep it at the target location** because **other parts of the code might also jump there**.

Let's take an example to illustrate the problem. Let's say:

- We have **two branches that can jump** to label `L1`.
- `Instruction X` is the first instruction at `L1`.
- We move `Instruction X` into the delay slot of one branch, **but the other branch still needs to find instruction `X` at `L1`**.

Here's what happens:

```
Branch A → L1
Branch B → L1

L1:
    Instruction X
    Instruction Y
```

If `Branch A` decides to move `Instruction X` inside its delay slot, `Branch B` cannot see `Instruction X` anymore! Therefore, we have to keep `Instruction X` in two places:

1. In `Branch A`'s delay slot
2. At `Label L1` for `Branch B`

Okay, and that should be a problem? For three reasons:

- We've **duplicated** `Instruction X`.
- **More code = more memory = larger executable**.
- **Harder to maintain**: if we change `Instruction X` in one place, we might **forget to update the duplicate**.

Best Use Case

Loops, particularly `do-while` constructs, where **backward branches are taken most of the time**.

2.5.5.3 From Fall-Through

In the **"From Fall-Through"** strategy, the **compiler selects an instruction that comes after the branch** in program order (i.e., from the fall-through path) and **moves it into the branch delay slot**. This method is **suitable when the branch is unlikely to be taken**, as execution will naturally continue sequentially.

Key Characteristics

- The **fall-through path** is **taken when the *branch is not taken***.
- The **delay slot instruction** comes from this path, meaning it is **executed anyway if the branch is not taken**.
- If the **branch is taken**, the delay slot instruction is either:
  - **Flushed** (discarded), or
  - Must be **safe to execute** (no side effects), even though it becomes useless work.

Example 4: From Fall-Through

Original code:

```
1 add x1, x2, x3
2 if x1 == 0 then      # branch condition
3 [delay slot, stall]
4 or x7, x8, x9        # execute if branch is not taken
5 sub x4, x5, x6       # execute if branch is taken
```

After scheduling:

```
1 add x1, x2, x3
2 if x1 == 0 then      # branch condition
3 or x7, x8, x9        # delay slot filled
4 sub x4, x5, x6       # execute if branch is taken
```

Here, `or x7, x8, x9` is **moved into the delay slot** from the instruction that would **normally execute next** if the branch is **not taken**.

Pipeline Behavior

- **Branch Not Taken** (Mist Likely)
  - **Delay slot instruction** is correctly **executed**.
  - Execution proceeds sequentially.
- **Branch Taken**
  - **Delay slot instruction** is **not needed**.
  - It must be **flushed**, or **safe to execute** even though its result is discarded.

When is this strategy used?

- ✔ When the **branch is not likely to be taken**.
- ✔ **Common in forward branches**, such as `if-then-else`, where `else` is rare.

Strategy	Delay Slot Instruction	Executed when branch
From Fall-Through	Instruction at `PC + 4` (next in sequence)	**Not Taken** (common case)
From Target	Instruction at BTA (label target)	**Taken** (common case)

Table 3: Comparison between “From Target” and “From Fall-Through”.

2.5.5.4 From After

The "From After" scheduling technique is **rarely used because it is too complex to be practical**. However, in the **"From After"** strategy, the **instruction** scheduled in the **branch delay slot** is **taken** from a later point the the code, specifically, from **after the fall-through instruction**.

Let's number the instructions to make it easy:

```
Instr A      # Before branch
Branch       # Branch condition
Instr B      # PC + 4 (fall-through)
Instr C      # After fall-through ← from after
Instr D
```

In "from after", the compiler **moves `Instr C` into the delay slot**, even though `Instr B (PC + 4)` should come right after the branch in normal execution (if not taken).

⚠ Why is this hard?

To safely move `Instr C` up into the delay slot, the **compiler must guarantee**:

1. **No Data Dependency Conflicts**
   - `Instr C` must not use or modify data that depends on `Instr A`, `B`, or the branch outcome. For example, if `Instr C` uses a value computed in `Instr B`, moving it before `Instr B` causes incorrect results.
2. **Safe if Executed Early**
   - Even if the branch is taken and `Instr C` should never execute, now it always executes in the delay slot.
   - So `Instr C` must be safe to execute even when it's not needed.
   - We call this a *speculatively safe instruction*.
3. **No Control Flow Violation**
   - If `Instr C` should only run after a condition is met, moving it earlier might break program logic.

2.6 Dynamic Branch Prediction

While static branch prediction relies on fixed rules or compile-time knowledge, **dynamic branch prediction** aims to **learn and adapt during program execution**. It **uses hardware mechanisms to observe past branch behavior** and **predict future outcomes at runtime**.

Core Idea

Dynamic prediction is **based on a key assumption**: if a branch behaved a certain way in the past, it's likely to behave the same way again. Therefore, instead of guessing statically, the **processor monitors each branch at runtime** and **uses past outcomes to inform future predictions**.

Hardware Components

Dynamic prediction relies on **two tightly-coupled hardware blocks**, both **situated in the Instruction Fetch (IF)** stage:

1. **Branch Outcome Predictor (BOP)**:
   - **Predicts branch direction**: Taken (T) or Not Taken (NT).
   - **Based on runtime history** (past outcomes of this or other branches).
2. **Branch Target Buffer (BTB)**:
   - **Predicts the target address** to jump to if the **branch is predicted taken**.
   - Returns the **Predicted Target Address (PTA)**[2].
   - **Useful only when BOP predicts Taken**; irrelevant if Not Taken.

How it works

During instruction fetch:

1. BOP predicts T/NT.
2. If Taken (T), BTB provides the PTA.
3. The processor fetches the next instruction accordingly.

[2] **Predicted Target Address (PTA)**: The memory address that the processor predicts as the destination for a taken branch. If the branch is predicted taken, the Branch Target Buffer (BTB) provides the PTA so that instruction fetch can continue from this address without waiting for the branch condition to be resolved.

Execution Scenarios

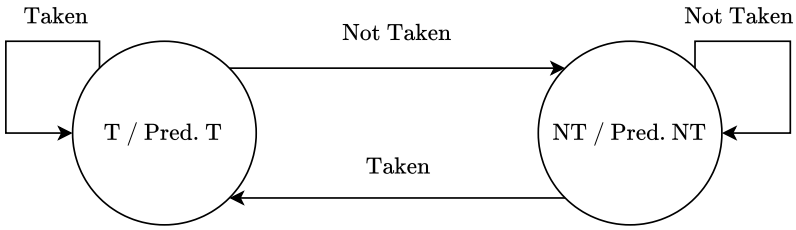
- **Prediction: Not Taken (`PC + 4`)**
  - If Branch Outcome = Not Taken ⇒ ✔ Correct prediction ⇒ No penalty.
  - If Branch Outcome = Taken ⇒ ✖ Misprediction:
    1. Flush next instruction (`NOP`)
    2. Fetch from BTA (to understand where to jump)
    3. One-cycle penalty
- **Prediction: Taken (BTB used)**
  - If Branch Outcome = Taken ⇒ ✔ Correct prediction ⇒ No penalty.
  - If Branch Outcome = Not Taken ⇒ ✖ Misprediction:
    1. Flush fetched target instruction (`NOP`) provided by BTB
    2. Fetch `PC + 4` (next instruction sequentially)
    3. One-cycle penalty

Unlike static prediction, **dynamic prediction is adaptive**. If a branch changes its behavior at runtime, future predictions adjust accordingly.

2.6.1 1-bit Branch History Table

In general, the **Branch History Table (BHT)**, or **Branch Prediction Buffer**, is a **dynamic hardware structure** that **predicts branch outcomes based on recent behavior**. The simplest version uses 1 bit per branch to remember whether the branch was **recently taken or not taken**. For this reason, it is called a **1-bit Branch History Table (1-bit BHT)**. It operates at runtime and uses a **Final State Machine (FSM) with 1-bit history**:

- If last outcome was Taken ⇒ predict Taken.
- If last outcome was Not Taken ⇒ predict Not Taken.



How it works

- **Each branch** instruction's address (or part of it) **indexes a table entry**.
- That **entry holds** a **single bit** (T/NT) representing the **last observed outcome**.
- On next encounter:
  - Use the **bit to predict the outcome**.
  - After actual branch resolution:
    - **Correct prediction** ⇒ keep bit unchanged
    - **Incorrect prediction** ⇒ flip bit

Indexing the Table

- Use k lower bits (right side) of the branch's address as the index.

No tags: **any branch with the same low-order bits shares the entry (can cause interference).**

- 2^k **entries** total.

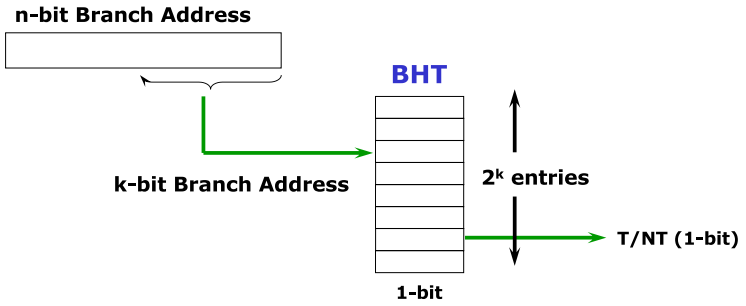


Figure 15: Visual representation of the 1-bit Branch History Table.

Example 5: Accuracy Issue

Consider a loop that executes 10 iterations. The expected behavior of the branch is:

T T T T T T T T T NT

Where the last is Not Taken because the code must exist from the loop and must continue (and not jump).
There are **two mispredictions**:

- **At the end**: Since iteration 9 is marked as Taken, the 10th iteration is predicted to be Taken, since the BHT contains Taken. This throws a misprediction because the branch result is Not Taken.
- **On the next time loop starts**: since the last iteration is stored in the BHT as Not Taken, the BHT has to flip the bit on the next time loop starts, and a misprediction occurs.

As a result, with 100% of 10 iterations, the BHT only catches 8 out of 10 iterations, the accuracy is 80%.

Shortcomings

- ✗ **Flipping prediction after 1 misprediction causes instability**, especially in loops.
- ✗ **Conflict problem**: two different branches with same index overwrite each other's bit.

Partial Solutions

To **reduce interference**:

- Increase table size (more k bits).
- Use hashing to mix address bits better.

2.6.2 2-bit Branch History Table

2-bit Branch History Table (BHT) is an **improvement over 1-bit BHT** designed to increase prediction stability, especially for loops, and reduce mispredictions.

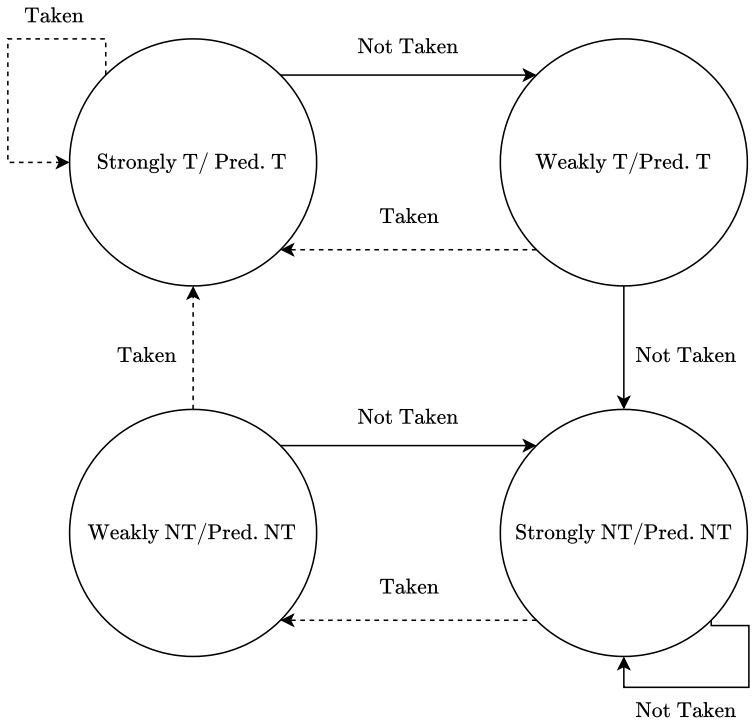
Why 2-bit? The Problem with 1-bit BHT

In loops, 1-bit BHT suffers from **flip-flopping**: **1 misprediction is enough to change the prediction**. This **causes two misprediction per loop**:

- Exiting the loop (NT mispredicted as T).
- Re-entering the loop (T mispredicted as NT).

The 2-bit BHT introduces a **4-state FSM** using **2 bits per entry**. It **requires 2 consecutive mispredictions to change the predicted outcome**, thus adding stability. The FSM states are:

1. **Strongly Taken (ST)** → Predict Taken
2. **Weakly Taken (WT)** → Predict Taken
3. **Weakly Not Taken (WNT)** → Predict Not Taken
4. **Strongly Not Taken (SNT)** → Predict Not Taken



Effect on Loops

Assume a loop with:

`T T T T T T T T T NT`

- Exit NT causes 1 misprediction, but FSM moves from Strongly Taken to Weakly Taken. So the prediction remains Taken.
- Re-enter on the loop causes a Branch Outcome Taken, and the 2-bit BHT predicts correctly because it is on the WT state.

Only 1 misprediction per loop, improving accuracy to 90% (from 80%).

Benefits

- **Improved accuracy** in loops and repetitive patterns.
- **Reduces misprediction penalty** in typical branch-heavy code.
- **Balances** prediction stability and adaptability.

2.6.3 Branch Target Buffer

The **Branch Target Buffer (BTB)** is a **specialized cache used to store target address of taken branches**. The stored Predicted Target Address (PTA) allows the processor to fetch instructions from the target without delay when a branch is predicted taken. The PTA is typically stored in PC-relative format (offset from current PC).

Core Idea

While the Branch History Table (BHT) predicts *whether* a branch will be taken, the Branch Target Buffer (BTB) **predicts *where* the program should go if the branch is taken**. The **BTB stores Predicted Target Addresses** (PTAs) for previously encountered branches and **enables fast redirection of control flow**.

How Is the BTB Structured?

- The BTB is designed as a **direct-mapped cache**:
  - The **address of the branch instruction is used to index** the BTB.
  - **Tags** are **used** for associative lookup to **confirm correctness** (i.e., ensure the indexed entry really belongs to the current branch)
- Components per **entry**:
  - **Tags**: Identifies the branch instruction.
  - **PTA**: The Predicted Target Address.
  - Often combined with **T/NT bits** from a **Branch History Table** (1-bit or 2-bit) for branch outcome prediction.

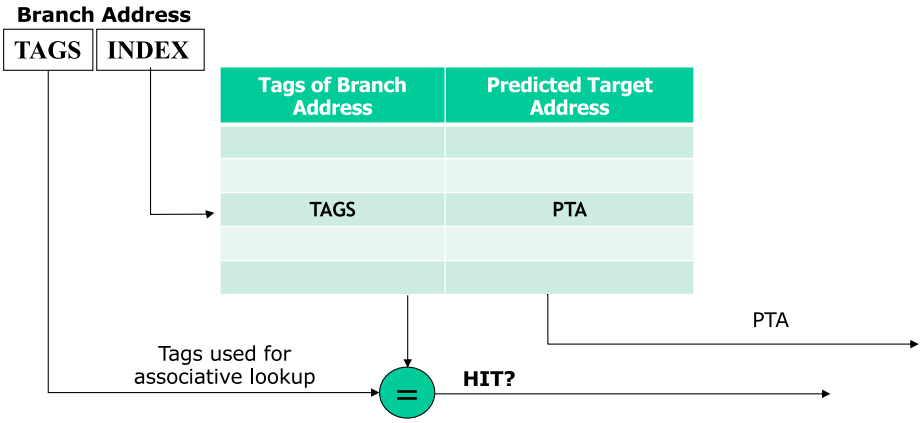


Figure 16: Branch Target Buffer without Branch Outcome Predictor.

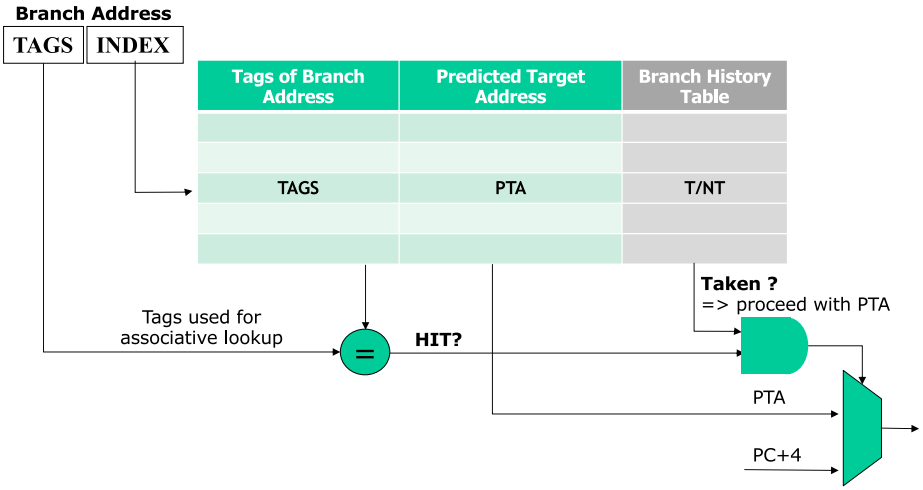


Figure 17: Branch Target Buffer with Branch Outcome Predictor.

BTB in the Pipeline

It is placed in the Instruction Fetch (IF) phase. During fetch:

- The BTB is queried using the current PC (branch address).
- If **hit and BHT predicts Taken**, fetch from PTA.
- If **miss or predict Not Taken**, continue at `PC + 4`.

Prediction	BTB use	Action
Predict Not Taken	BTB **not used**	Fetch from PC + 4
Predict Taken (BTB hit)	BTB **used**	Fetch from PTA stored in BTB
Predict Taken (BTB miss)	BTB **miss**	Stall or default to `PC + 4`, then calculate BTA

Table 4: Summary of Behavior.

Advantages

- ✔ **Eliminates delay** from calculating the **branch target address** manually.
- ✔ Enables speculative instruction fetch from correct target, **improving pipeline efficiency**.

2.6.4 Correlating Branch Predictors

What is the problem?

With standard BHT, we **predict each branch individually**, based only on **its own past behavior**. But real programs often have **branches that influence each other**. For example, let's look at the following code:

```
1 if (x > 0)  // Branch A
2     ...
3 if (x > 0)  // Branch B
4     ...
```

- `Branch B` often behaves like `Branch A`, because they depend on the same condition (`x > 0`).
- A normal predictor doesn't know this. It treats `A` and `B` independently.

Key Idea

Use **global branch history** (outcomes of previous branches) to **improve prediction** for the current branch. This approach exploits correlation between different branches. This technique is called **Correlating Predictors** or **2-level Predictors**.

General Case: (m, n)

In a **(m, n) correlating predictor**, the **past outcomes of the last m branches** are used to **select among 2^m prediction tables**, **each of which uses n-bit prediction entries**.

- m: The number of **global history bits**.
- n: The number of **bits per prediction entry in the BHTs** (e.g., 1 or 2).

It works like this:

1. **Track the Last m Branches**: **Store the outcomes** (T/NT) of the last m **branches** in a **Global History Register (GHR)**. This forms an m-bit global history pattern.
2. **Use GHR to Select Table**: The m-bit GHR selects 1 out of 2^m Branch History Tables (BHTs). Each BHT contains n-bit entries.
3. **Index the Selected Table**: **Use low-order bits** of the branch instruction address (e.g., PC bits) **to index an entry in the selected table**.
4. **Predict Using n-bit Entry**: Use the n-bit entry to predict:
   - 1-bit BHT: predict Taken or Not Taken.
   - 2-bit BHT: use 4-state FSM (Strong and Weak Taken and Not Taken)

So what we have in the **memory** is:

- **Total tables**: 2^m BHTs.
- **Each BHT has**: 2^k entries (k is the number of PC bits used).
- **Total entries**: $2^m \times 2^k$.

Advantages

- **Captures patterns across multiple branches**.
- **Helps in complex control** flow where a branch's outcome depends on prior branches.
- **More accurate** than per-branch-only prediction.

2.6.4.1 (1,1) Correlating Predictors

Use the **result of the last executed branch** (global history, $m = 1$ bit) to **choose between two prediction tables**, each of which has 1-bit entries.

- **1-bit Global History**: Stores last branch outcome (Taken = 1, Not Taken = 0).
- **2 BHTs (T1 & T2)**: Each is a 1-bit predictor table, selected based on global history. We use a 1-bit Branch History Table technique.
- **Indexing**: Use PC low-order bits to index into the selected table.

Consider a pseudo code:

```
if (x > 0)  // Branch A
    ...
if (x > 0)  // Branch B
    ...
```

Let's say if `A` is true, `B` is usually true. The execution walkthrough:

- Cycle 1: **First Execution of `Branch A`**
  1. Global History: unknown or NT (0); because it doesn't track anything yet. We assume Not Taken (0).
  2. Use Table T2 (since `GH = 0`).
  3. Index into T2 with `Branch A`'s PC bits.

  Predict: Not Taken!

  Unfortunately, the Branch Outcome (BO) says Taken ⇒ ✗ **Mispredict** ⇒ update T2 entry to T.

  4. Update Global History = 1 (Taken).
- Cycle 2: **Now Executing Branch `Branch B`**
  1. Global History = 1 (Taken) from `Branch A`.
  2. Use Table T1 (since `GH = 1`).
  3. Index with `Branch B`'s PC bits.

  T1 says: "Try to predict as Taken".

  The Branch Outcome (BO) says Taken ⇒ ✓ **Correct prediction**.

  4. No update needed. The Global History doesn't need an update either, because it is already 1 (Taken).

Since `Branch A` was Taken, `Branch B` is likely to be Taken. This is a smart technique because normal predictors treat `Branch B` alone. Instead, the Correlating Branch Predictor uses context, and the last branch helps predict this one.

Branch	Global History	Table Used	Prediction	Outcome	Update
Branch A	0	T2	NT	T	T2 entry to T
Branch B	1	T1	T	T	No change

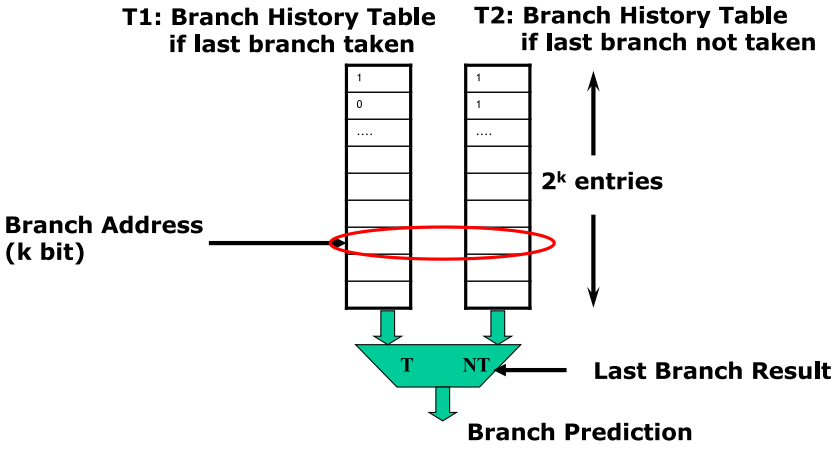


Figure 18: Visual representation of the (1,1) Correlating Predictor.

Aspect	Description
$m = 1$	Use 1-bit GHR (last branch result).
$n = 1$	1-bit prediction, (T or NT).
Tables	2 BHTs (for `GHR = 0` and `GHR = 1`).
Selection Logic	If last branch T, use Table T1; else T2.

Table 5: (1,1) Correlating Predictor.

2.6.4.2 (2,2) Correlating Predictors

The correlating predictors with $m = 2$ and $n = 2$ have the following components:

- **2-bit Global History**: Stores outcomes of the last 2 branches. Forms 4 patterns:
  - `00`: both Not Taken.
  - `01`: last branch Taken, penultimate branch Not Taken.
  - `10`: last branch Not Taken, penultimate branch Taken.
  - `11`: both Taken.
- **4 Prediction Tables**: One for each global history pattern ($2^2 = 4$ BHTs).
- **2-bit entries per BHT**: Each BHT uses 2-bit saturating counters for stable predictions.
- **Indexing**: Use PC low bits + global history to access an entry in a BHT.

Consider a pseudo code:

```
if (A)          // Branch 1
   ...
if (B)          // Branch 2
   ...
if (C)          // Branch 3
   ...
```

Let's simulate `Branch 3`'s prediction, influenced by Branches 1 & 2.

0. **Initial State**
   - Global History Register `(GHR) = 00` (no branches taken yet).
   - BHT for history 00 selected.
   - Predicts `Branch 3` using its 2-bit counter in `BHT[00]`.
1. **Cycle 1: `Branch 1` = Taken**
   - GHR: `00` → `01` (shift in `T = 1`, Taken).
   - Update `BHT[00]` (for `Branch A`), since we used `GHR = 00` before `A`.
2. **Cycle 2: `Branch 2` = Not Taken**
   - GHR: `01` → `10` (T, NT).
   - Update `BHT[01]` (for `Branch B`), since `GHR = 01` before `B`.
3. **Cycle 3: Predict `Branch 3`**
   - `GHR = 10`, so select the BHT that contains `10` for prediction.
   - Use `Branch 3`'s PC low bits `+ GHR = 10` to index `BHT[10]`.

   Check the 2-bit FSM in this entry. Assume the FSM state is Weakly Taken, so the **prediction is Taken**.

   The outcome of `Branch 3` is Taken, so the **prediction is correct** and we update the FSM of entry `BHT[10]`.

Cycle	Branch	Outcome	GHR Before	GHR After	BHT Used for Update
1	A	T	00	01	`BHT[00]`
2	B	NT	01	10	`BHT[01]`
3	C	T	10	01	`BHT[10]`

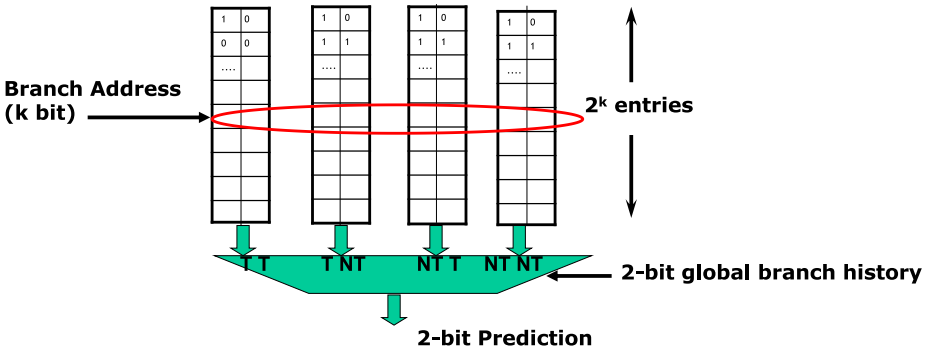


Figure 19: Visual representation of the (2,2) Correlating Predictor.

Aspect	**Description**
$m = 2$	Track outcomes of last 2 branches (`GHR = 2` bits)
$n = 2$	2-bit prediction entries per BHT
Tables	4 BHTs (for `GHR = 00, 01, 10, 11`).
Indexing	`4-bit PC + 2-bit GHR` → 6-bit index for accessing a table
Prediction Stability	More robust due to 2-bit FSM per entry.

Table 6: (2,2) Correlating Predictor.

2.6.5 Two-Level Adaptive Branch Predictors

Two-Level Adaptive Branch Predictors are advanced techniques that aim to provide highly accurate and adaptive predictions by **combining history tracking with pattern-based decision-making**. Unlike simpler predictors that use only the outcome of the last branch or last few outcomes of a single branch, these predictors **consider patterns over time** and **across different branches** to improve prediction accuracy.

Core Concept

The two-level approach consists of:

1. A **history-tracking component**: to record the outcomes of recent branches.
2. A **pattern-based prediction component**: to use that history to make accurate predictions.

This **design allows the processor to learn and adapt to recurring patterns in branch behavior**, which is particularly useful for complex control flows and loops.

Structure

1. **Branch History Register (BHR)**
   - A k**-bit shift register** that records the outcomes of the k most recent branches (e.g., T, NT, NT, T).
   - The BHR can be either:
     - **Global**: one register for all branches.
     - **Local**: separate register for each branch.
2. **Pattern History Table (PHT)**
   - A table of **2-bit saturating counters** (like in 2-bit BHT).
   - **Indexed** using the **content of the BHR**.
   - Each **entry** provides a **prediction** (Taken/Not Taken) and adapts over time.
3. **Prediction Process**
   (a) Use the BHR value to index the PHT.
   (b) Read the 2-bit counter at that entry.
   (c) Predict Taken if in a Taken state, otherwise Not Taken.
   (d) After the actual branch outcome:
      i. Update the 2-bit counter accordingly.
      ii. Shift the actual outcome into the BHR.

Global Adaptive Predictor (GA)

The **Global Adaptive Predictor (GA)** is a **specific form** of the two-level predictor where **global history (BHR) is used to index the PHT**.

- The **BHR is shared across all branches**, and thus captures the global correlation among different branches.
- The **PHT is local** in the sense that it provides per-entry adaptation via 2-bit counters.

The main **advantage** is that by correlating the current branch with the behavior of previous branches (stored in the BHR), the **predictor can detect global patterns and make more informed predictions**.

GShare Predictor

GShare is a **variation of the Global Adaptive Predictor**, designed to **improve the indexing of the PHT and reduce aliasing** (i.e., different branches mapping to the same PHT entry).

Instead of directly using BHR to index the PHT, **GShare performs an XOR** between:

- The **BHR** (global history of recent outcomes).
- The **low-order bits of the program counter (PC)** of the current branch.

The XOR operation **mixes the global history with branch-specific information**, making it more likely that **different branches will access different entries** in the PHT, thus **reducing prediction interference** (aliasing). This allows GShare to **reduce aliasing and have a global and local view**.

Predictor	History used	Indexing to PHT	Benefit
Global Adaptive (GA)	Global BHR	BHR value directly indexes PHT	Simple, effective for globally correlated branches
GShare	Global BHR + PC	BHR XOR PC bits index PHT	Reduces aliasing, captures global + local context

Table 7: Summary of Global Adaptive and GShare.

3 Instruction Level Parallelism

3.1 The problem of dependencies

Instruction-Level Parallelism (ILP) is a foundational concept in modern processor design that aims to **improve performance by executing multiple instructions simultaneously within a single processor core**. The fundamental **premise of ILP is that many instructions within a program can be executed independently**, and thus, **can be overlapped in time**. This section explores the principles of pipelining as a means to exploit ILP, emphasizing its benefits, ideal performance metrics, and the inherent limitations.

However, **instruction dependencies** represent critical constraints on this parallelism. Understanding these dependencies is essential for analyzing the potential parallelism in a program and for designing hardware or compilers that can exploit ILP safely and efficiently.

Instruction dependencies determine which instructions can be executed simultaneously and which ones must respect a specific order of execution. These dependencies are classified into three broad categories: **data dependencies**, **name dependencies**, and **control dependencies**.

Correct Program Behavior

For correct program behavior, **two program properties must always be preserved** during instruction scheduling:

1. **Data Flow**. The correct **values** must be **produced and consumed in the proper order**.
2. **Exception Behavior**. Reordering must **not alter the way exceptions are raised and handled** in the program.

While **dependencies are intrinsic to the program semantics**, **hazards are an architectural artifact of the pipeline implementation**.

3.1.1 Data Dependencies

Data Dependencies, also called **True Data Dependencies**, are the most fundamental type of instruction dependencies in a program. They **express the real flow of data from one instruction to another** and are dictated by the *semantics* of the program. These **dependencies must be strictly preserved** during any reordering or parallel execution of instructions, **otherwise the correctness of the program is compromised**.

Formally, we say there is a data dependencies from instruction I_i to instruction I_j (where I_j follows I_i in program order), **if I_j reads a value that is produced by I_i**. In other words, I_j needs the output of I_i as its input. This is known as a **Read After Write (RAW)** hazard in pipeline terminology.

Why is it called "*true*"?

This type of dependence is "*true*" because the **second instruction cannot proceed correctly until the first one completes its write operation**. It reflects an actual requirement for program correctness.

Example 1: RAW Hazard

```
1 I1:   r3 ← r1 + r2   # produces a value in r3
2 I2:   r4 ← r3 + r5   # consumes the value from r3
```

Here, `I2` is data-dependent on `I1` because it reads from register `r3`, which is written by `I1`. The instructions must execute in order:

- `I1` must execute and complete its write to `r3` before.
- *I2* reads `r3` to perform its own computation.

If this order is violated, e.g., `I2` executes before `I1` finishes, then `I2` will read an incorrect or undefined value.

Why Data dependencies Matter for ILP

Data dependencies define which **instructions must not be executed in parallel**, because doing so would result in violating program semantics.

- In a **pipelined processor**, data dependencies may cause **pipeline stalls**.
- In **out-of-order processors**, special mechanisms (like reservation stations and the reorder buffer) track and resolve data dependencies to allow other independent instructions to proceed while dependent ones wait for operands.

3.1.2 Name Dependencies

Unlike true data dependencies, **Name Dependencies** arise when **two instructions use the same register or memory location**, but **there is no actual flow of data between them**. These dependencies are called **False Dependencies** or **Pseudo-Dependencies** because they are **not required** for program correctness from a data perspective, **but still impose constraints** on instruction scheduling.

These constraints are due to the reuse of names (i.e., identifiers like register names), not due to real dependencies in the data values. They may still cause hazards in a pipeline and need to be addressed, especially when trying to execute instructions in parallel or out of order.

Types of Name Dependencies

There are two main types:

- **Anti-Dependence (Write After Read - WAR)**. An **anti-dependence** occurs when:
  - A first instruction **reads** from a location (register/memory);
  - A second instruction **writes** to that same location, after it.

  This introduces a WAR hazard: if the **second instruction is executed too early** (before the first instruction finishes reading), **it might overwrite the value before the first instruction uses it**.

  Example 2: WAR Hazard

  ```
  1 I1:   r3 ← r1 + r2   # reads r1 and r2
  2 I2:   r1 ← r4 + r5   # writes to r1
  ```

  In this case:
  - `I1` reads from `r1`
  - `I2` writes to `r1`

  There is no data flow between the two (i.e., `I1` doesn't use the result of `I2`, and vice versa), but if `I2` executes before `I1` finishes, the read in **I1** may get a corrupted value.

- **Output Dependence (Write After Write - WAW)**. An **output dependence** occurs when **two instructions write to the same location** (register or memory). This results in a WAW hazard: executing the **second instruction first may overwrite the location**, changing the final value from what the program originally intended.

Example 3: WAW Hazard

```
I1:   r3 ← r1 + r2   # writes to r3
I2:   r3 ← r4 + r5   # writes to r3
```

There's no direct data flow between the two, but the **ordering matters**. If `I2` is supposed to overwrite `r3` after `I1`, reversing the order would result in `I1`'s result being incorrectly seen as the final value in `r3`.

Resolving Name Dependencies: Register Renaming

The key idea in dealing with name dependencies is to **recognize that they are not real** and can be **eliminated if we avoid the reuse of names**. The technique used to eliminate these artificial constraints is called **Register Renaming**. The idea is simple:

- If two instructions refer to the same register but don't actually share data, assign them **different physical registers**.

This is only possible when the underlying hardware (or compiler) provides more physical registers than the number of logical registers visible in the ISA.

Example 4: Resolving WAR and WAW

Original code (WAR):

```
I1:   r3 ← r1 + r2
I2:   r1 ← r4 + r5
```

Renamed:

```
I1:   r3 ← r1 + r2
I2:   r9 ← r4 + r5    # write to r9 instead of r1
```

Now, there is no conflict, `I2` can proceed independently of `I1`.
Original code (WAW):

```
I1:   r3 ← r1 + r2
I2:   r3 ← r4 + r5
```

Renamed:

```
I1:   r3 ← r1 + r2
I2:   r9 ← r4 + r5    # write to a new register
```

⚖ Hardware vs. Software Register Renaming

- **Hardware (Dynamic Renaming)**. **Performed at runtime** by structures such as the Register Alias Table (RAT), typically in out-of-order superscalar processors.
  - ✅ Flexible
  - ❌ Adds hardware complexity
- **Software (Static Renaming)**. **Performed at compile time** by the compiler, particularly for VLIW or statically scheduled processors.
  - ✅ Simpler in hardware
  - ❌ Puts more pressure on compiler technology

3.1.3 Control Dependencies

While data and name dependencies arise from how instructions read and write operands, **Control Dependencies** stem from **the flow of control in the program**, that is, the **presence of branches and conditional execution**.

Control dependencies are fundamentally about **deciding whether an instruction should execute at all**, based on the **result of** a preceding **branch** or **conditional instruction**.

Formally, an instruction I_j is **control-dependent** on a branch instruction I_b if:

- I_j must only execute **if a particular outcome** of I_b is taken.
- But the **decision** made by I_b (e.g., whether to branch or not) is **not yet known** when I_j enters the pipeline.

This introduces uncertainty: *should I_j be fetched and executed, or not?*

Example 5: Control Dependencies

```
if (x > 0)
    A; // Instruction A is control dependent on the condition
     (x > 0)
```

In assembly:

```
I1:  bgtz  r1, LABEL    # branch if r1 > 0
I2:  ...                # instruction before LABEL
I3:  LABEL: A           # instruction A
```

- `A` should only execute **if the branch is taken**.
- But we don't know whether the branch is taken until the condition is resolved, which happens later in the pipeline.

Why Control Dependencies matter for ILP

Control dependencies **limit instruction parallelism**:

- We cannot freely reorder or speculate on the instructions following a branch.
- Waiting for the branch outcome introduces **stalls** in the pipeline.

Thus, exploiting ILP requires **breaking or relaxing control dependencies**, without violating program semantics.

Control Dependencies Solution

We have dedicated an entire section to this topic, see 2, page 28.

3.2 Multi-Cycle Pipelining

As processor microarchitectures evolved to support more complex instructions and higher performance demands, the basic model of a uniform single-cycle pipeline became insufficient. In practice, **many instructions**, especially those involving floating-point operations, memory access, or division, **require more than one clock cycle** to complete their execution or memory stages.

This leads to the development of **multi-cycle pipelines**, where **individual stages** (particularly EX and MEM) may **last for multiple cycles**, depending on the instruction type and runtime events. In such architectures, the ability to manage instruction progress intelligently becomes central to maintaining high throughput and correctness.

Motivation and Assumptions

In a classical 5-stage pipeline (IF, ID, EX, MEM, WB), all **stages are assumed to complete in one clock cycle**. However, this assumption doesn't hold in realistic systems:

- **Integer instructions** may complete in 1-2 cycles.
- **Floating-point operations**, like multiplication or division, can take 3 to 10+ cycles.
- **Memory access** times are unpredictable due to cache hits and misses, which can add variable delays.
- **Instruction fetch** may also stall due to instruction cache misses or branch resolution delays.

To accommodate these characteristics, **processors adopt multi-cycle pipelines**, where:

- Execution latency varies by operation type.
- Memory access can take multiple cycles.
- Functional units are not necessarily pipelined, particularly for floating-point operations.

Basic assumptions in this model:

1. The processor is **single-issue** (one instruction issued per cycle).
2. **Instructions** are typically **issued in-order** (fetched and passed into the pipeline in the order that they appear in the program).
3. Execution and memory stages may involve **multiple functional units with variable latencies**.
4. **Write-back** is often **delayed or synchronized** to ensure consistent state updates and avoid hazards.

3.2.1 Multi-Cycle In-Order Pipeline

In a **Multi-Cycle In-Order Pipeline**, instructions are:

- **Issued in program order** (**in-order issue**).
- **Executed on dedicated functional units**, each potentially requiring multiple cycles.
- **Committed in order**, i.e., write-back to the architectural state occurs in the order instructions were issued (**in-order commit**).

This model retains a strict discipline:

- Even if a later instruction finishes earlier (because it uses a faster unit), it **cannot write back until its turn arrives**.
- This **avoids WAR** (Write After Read) **and WAW** (Write After Write) hazards by ensuring in-order commit.

Advantages

- ✔ Simpler control logic.
- ✔ Preserves the precise exception model.
- ✔ **No need for register renaming or reorder buffers**.

Disadvantages

- ✖ **Poor ILP exploitation**: independent instructions may stall behind slow ones.
- ✖ All instructions are serialized through the same issue logic, even if no true dependence exists.

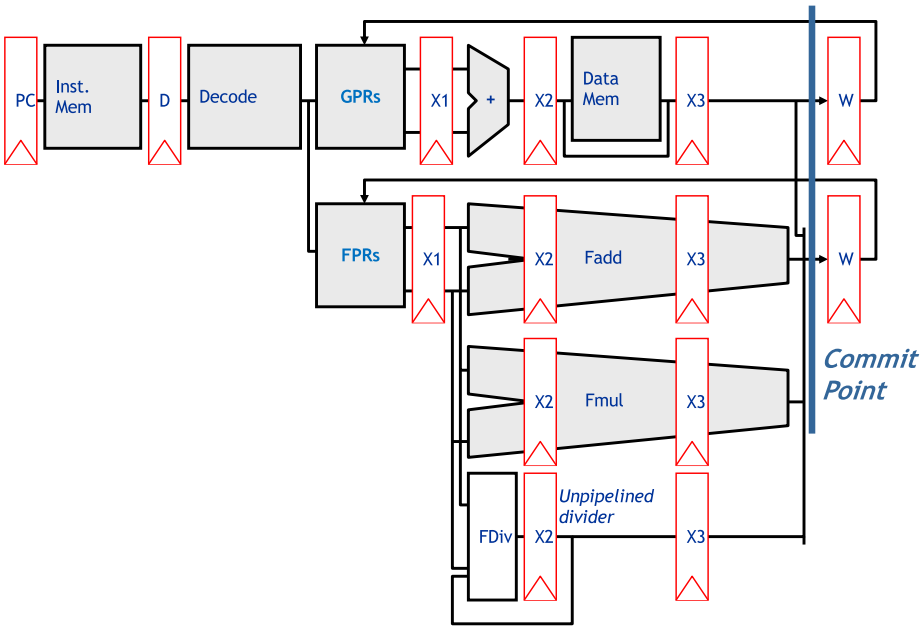


Figure 20: Multi-Cycle In-Order Pipeline architecture. This processor includes **different execution units**, each optimized for a specific operation type:

- `X1-X2`: 2-stage execution, used for basic integer ALU.
- `Fadd`: 3-stage execution, used for floating-point addition.
- `Fmul`: 3-stage execution, used for floating-point multiplication.
- `FDiv`: not pipelined, used for floating-point division (long)

So integer operations may take 2 cycles, add/mul take 3, and divide takes many cycles and cannot overlap because it's not pipelined. The instruction flow moves through:

- IF (Instruction Fetch): from PC and instruction memory.
- D (Decode): identifies operand registers, selects execution unit.
- X1, X2, ...: execution pipeline stages, depending on instruction type.
- Data Mem: if needed (e.g., for load/store).
- W (commit point) + GPRs/FPRs: Write-Back (WB) stage to either General-Purpose Registers or Floating-Point Registers.

Feature	**Description**
Issue	In order
Execution	In order
Completion (write-back)	In order (even if execution latency differs)
Architectural State Updates	In order; results are committed exactly in program order
ILP	Limited, stalls propagate even to independent instructions
Complexity	Moderate, simpler control logic, no renaming or reorder buffer needed
Exceptions	Always precise, easy to track and recover since instructions complete in order

Table 8: Summary of Multi-Cycle In-Order Pipeline.

3.2.2 Multi-Cycle Out-of-Order Pipeline

To overcome the limitations of in-order execution, processors adopt **Multi-Cycle Out-of-Order (OoO) Pipeline**s. It is a more sophisticated architecture that aims to maximize ILP by **executing independent instructions as early as possible**, regardless of program order, and **allowing instructions to complete out of order**.

In this model:

- **Instructions** are still fetched and decoded **in-order** (**in-order issue**).
- After decoding, **instructions are placed into issue queues** or **reservation stations** (**out-of-order execution**).
- As soon as operands are available and a suitable functional unit is free, instructions execute, regardless of original order.
- **Write-back and commit may also occur out-of-order**, although the final architectural state is updated in program order to preserve correctness (**out-of-order commit**).

Advantages

- ✔ **Maximizes ILP** by letting independent instructions execute as soon as possible.
- ✔ **Improves throughput** by keeping all functional units busy.
- ✔ **Hides long latencies**, like FP division or memory misses.

Challenges

Out-of-Order execution introduces serious architecture challenges:

- **WAR and WAW Hazards**. If later instructions write back before earlier ones:
  - ✘ They might overwrite data that's still needed.
  - ✔ Hardware **must detect and prevent** these scenarios.

  This is typically handled using: register renaming and scoreboarding / reservation stations.
- **Imprecise Exceptions**. If an exception occurs (e.g., divide-by-zero), but the processor has already executed and committed later instructions, the architectural state is **no longer consistent** with the point of the fault. To fix this, high-performance CPUs use:
  - ✔ **Reorder Buffers (ROB)** to store results until it's safe to commit them in program order.

✔ **Checkpointing and rollback mechanisms** to recover precise state.

Formally, an **exception is imprecise** occurs if the processor state when an exception is thrown does not look exactly as if the instructions were executed in order.

Feature	**Description**
Issue	In order
Execution	Out of order
Completion (write-back)	Out of order
Architectural State Updates	May occur out of order (but real designs use re-order buffers to enforce in-order commit)
ILP	High
Complexity	High - needs hazard detection, renaming, ROBs
Exceptions	Risk of imprecision without commit logic

Table 9: Summary of Multi-Cycle Out-of-Order Pipeline.

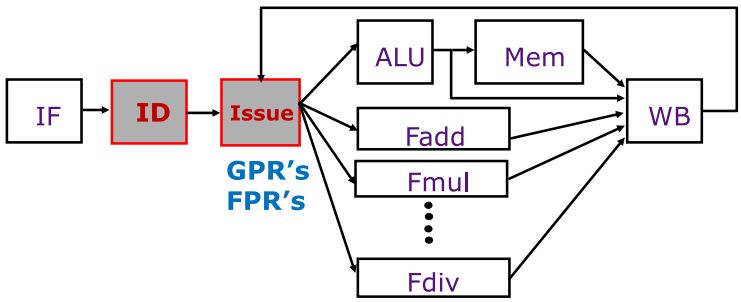


Figure 21: High-Level Multi-Cycle Out-of-Order Pipeline architecture.

- IF (Instruction Fetch): fetches the next instruction in program order from instruction memory using the program counter (PC).
- ID (Instruction Decode). This stage is now split into two sub-stages:
  1. ID: decoding the instruction format and operation type.
  2. Issue: reading registers and checking availability of operands.

  This split is key to **preparing instructions early**, even if they're not ready to execute immediately.
- Functional Units. The processor has **multiple independent execution units**, each possibly multi-cycle and with different latencies:
  - ALU: used for integer arithmetic, logic, and takes 1-2 cycles.
  - Mem: used for load/store, with cache hits/misses, and takes a variable number of cycles.
  - Fadd: used for floating-point addition, and takes ≈ 3 cycles.
  - Fmul: used for floating-point multiplication, and takes ≈ 3 cycles.
  - Fdiv: used for floating-point division, and takes multiple cycles and is not always pipelined.

  Each unit operates independently, and several can be active at once.
- GPRs and FPRs (General/Floating-Point Registers): architectural registers where results are ultimately written. But in this pipeline, results may first go to **temporary storage** until committed (through not shown here explicitly, concepts like **reorder buffer (ROB)** are implied).

Unlike the in-order pipeline, there is **no single commit point** shown. This means out-of-order commit, and introduces the risk of: WAR and WAW hazards, and imprecise exceptions.

3.3 Dynamic Scheduling

As we've seen, static in-order pipelines are limited in their ability to exploit ILP because they stall the entire pipeline when a single instruction is blocked. **Dynamic Scheduling** solves this by allowing **instructions to be issued, executed, and even completed out of order**, as long as doing so does not violate program correctness.

The Need for Dynamic Scheduling

Consider the following instruction sequence:

```
I1:  F0  ← F2 / F4   # long-latency divide
I2:  F10 ← F0 + F8   # depends on F0
I3:  F12 ← F8 - F14  # independent of I1 and I2
```

In a naive in-order pipeline:

- `I2` stalls waiting for `F0`, and
- `I3` stalls behind `I2`, even though it's independent.

This results in lost parallelism. In contrast, **dynamic scheduling** would:

- Allow `I1` to begin and proceed through the divider.
- Stall `I2` because it depends on `F0`.
- Allow `I3` to **proceed and complete** immediately, despite the stall.

This is possible because the processor can **track operand availability** and issue instructions **as soon as dependencies are satisfied**, not based solely on their program order.

How Dynamic Scheduling Works

Instructions are issued in order but may **execute and complete out of order**, depending on operand readiness and unit availability. The processor uses dedicated **hardware structures** to manage this:

- Reservation Stations (or Issue Queues)
- Reorder Buffer (ROB)
- Register Renaming Tables
- Common Data Bus (CDB) for broadcasting results

In a dynamically scheduled pipeline, stages are typically:

- **Fetch (IF)**: Get instruction from memory.
- **Decode (ID)**: Determine opcode, operands, destination.
- **Issue**: Place instruction into reservation station if operands aren't ready.
- **Execute (EX)**: Start when all operands are available.

- **Write Result**: Write result to a temporary buffer or broadcast to waiting instructions.
- **Commit**: Update architectural registers in order.

Benefits

- ✔ **Higher ILP**: Instructions don't wait unnecessarily.
- ✔ **Resource Utilization**: Keeps functional units busy.
- ✔ **Latency Hiding**: Tolerates cache misses, long FP ops.
- ✔ **Exploits Independence**: Independent ops no longer block one another.

Challenges Introduced

While powerful, dynamic scheduling is **complex**:

- ✘ **WAR and WAW Hazards**. With out-of-order execution, later instructions might write before earlier ones. Solution: use register renaming to remove name dependencies.
- ✘ **Imprecise Exceptions**. If a later instruction causes an exception, but earlier ones have already modified state, it becomes hard to roll back. Solution: use a Reorder Buffer (ROB) that holds results temporarily and commits them in program order.
- ✘ **Hardware Cost and Complexity**. Additional logic is needed for:
  - Dependency tracking
  - Wakeup and select logic
  - Common data bus broadcasting
  - Instruction window buffering

3.4 Multiple-Issue Processors

3.4.1 Introduction to Multiple-Issue Pipelines

In previous sections, we explored how pipelining (section 3.2) and dynamic scheduling (section 3.3) help improve instruction throughput by exploiting instruction-level parallelism (ILP). However, traditional scalar **pipelines are fundamentally limited**: they can issue only one instruction per clock cycle. To overcome this limitation and achieve even higher performance, computer architects developed Multiple-Issue Processors.

Definition 1: Multiple-Issue Processors

Multiple-Issue Processors are processors designed to fetch, decode, issue, and execute **more than one instruction per clock cycle**, with the goal of increasing instruction throughput and exploiting instruction-level parallelism (ILP).

They achieve higher performance than scalar processors by issuing multiple independent instructions in parallel, using either hardware-based dynamic scheduling (as in superscalar architectures) or compiler-driven static scheduling (as in VLIW architectures).

This section introduces the key principles of multiple-issue pipelines and lays the foundation for understanding both superscalar and VLIW architectures.

⚠ The Limits of Scalar Pipelines

In a **scalar pipeline**, **only one instruction is issued and completed per clock cycle**, even if other instructions are independent and could be executed in parallel.

Let's consider a classic 5-stage pipeline:

$$\texttt{IF} \rightarrow \texttt{ID} \rightarrow \texttt{EX} \rightarrow \texttt{MEM} \rightarrow \texttt{WB}$$

In an ideal case, the pipeline achieves an IPC (Instructions Per Cycle) of 1. That is:

- 1 instruction finishes per cycle.
- Corresponding CPI (Cycles Per Instruction) is also 1:

$$\text{CPI}_{\text{ideal}} = 1, \quad \text{IPC}_{\text{ideal}} = 1$$

But in reality, hazards (data, control, structural) can cause stalls and the IPC can fall below 1. As we have already discussed in the previous sections.

⚠ **Key Limitation**: Even if the program contains many **independent instructions**, the scalar **pipeline processes them sequentially**, one at a time.

Raising Performance: Introducing Multiple Issue

To extract more parallelism and achieve better throughput, multiple-issue processors aim to:

- Fetch **multiple instructions** per cycle.
- Issue and execute **multiple instructions** in parallel.
- Increase **IPC above 1** ↑ and reduce **CPI below 1** ↓

This means:

- The processor is no longer limited by the sequential issue constraint.
- ILP is exploited across multiple instructions simultaneously.

A simple example is the **dual-issue pipeline**, where up to two instructions can be issued and completed per clock cycle. It allows two independent instructions to proceed through the pipeline in parallel, potentially doubling the instruction throughput compared to a scalar pipeline:

$$\text{IPC}_{\text{ideal}} = 2 \qquad \text{CPI}_{\text{ideal}} = \frac{1}{\text{IPC}_{\text{ideal}}} = \frac{1}{2} = 0.5$$

Definition 2: Dual-Issue Pipeline

A **Dual-Issue Pipeline** is a **type** of multiple-issue processor pipeline that can fetch, decode, and issue **up to two instructions per clock cycle**.

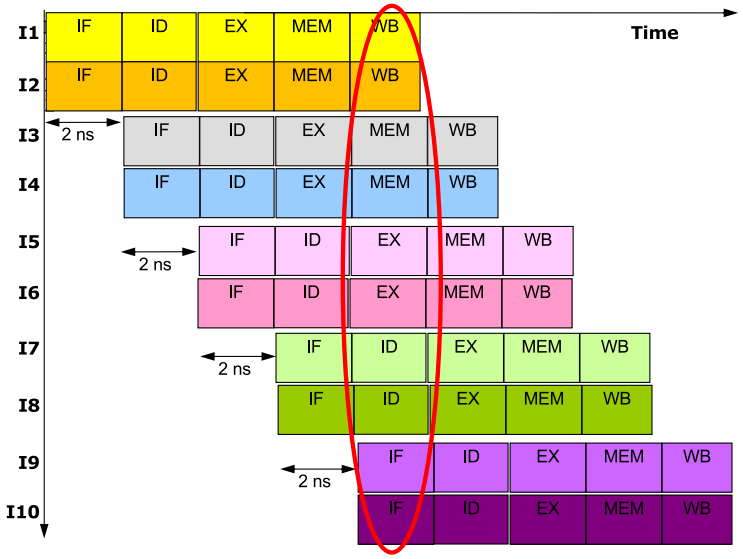


Figure 22: Dual-Issue Pipeline timeline.

Architectural Requirements

It's pretty obvious that multi-issues processors require more hardware resources to support parallelism:

- **Wider Instruction Fetch (IF) units**: able to fetch 2 instructions per cycle from instruction memory.
- **Parallel Instruction Decoders (IDs)**: 2 independent decode units to process both instructions in parallel.
- **Multi-Ported Register File (RF)**:
  - 4 Read Ports: to read up 2 source operands per instruction.
  - 2 Write Ports: to write results from both instructions simultaneously.
- **Duplicated Functional Units**: at least 2 independent units (e.g., 1 ALU or branch, 1 load/store) to allow parallel execution.

These **additions increase complexity**, **area**, and **power consumption**, but allow significant performance gains.

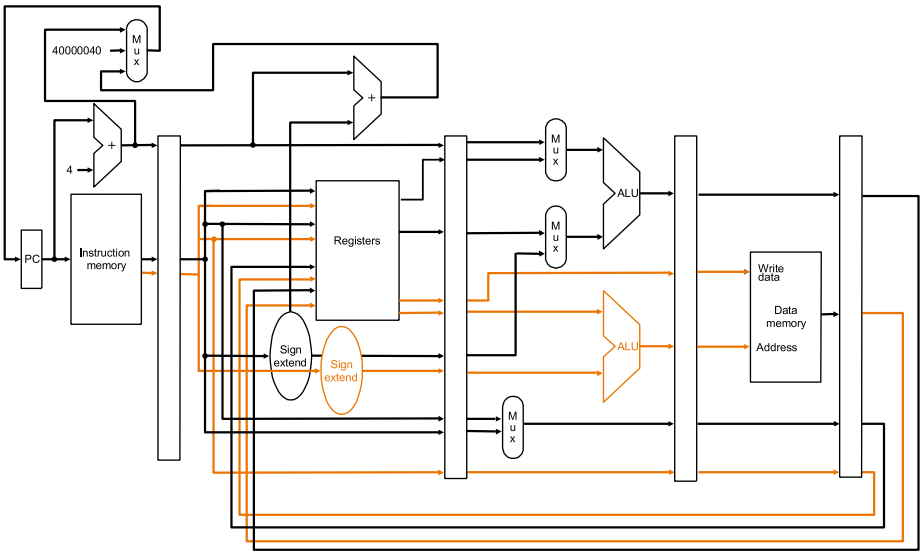


Figure 23: Dual-Issue Pipeline architecture.

3.4.2 Evolution Towards Superscalar Execution

The transition from simple scalar pipelines to high-performance superscalar processors is not abrupt. It is the result of a **progressive refinement** of microarchitectural techniques aimed at exposing and exploiting more Instruction-Level Parallelism (ILP). This section traces the key evolutionary steps that bridge the gap between single-issue scalar designs and fully dynamic multiple-issue architectures.

⧗ Step 1: Single-Issue, In-Order Execution

This is the traditional scalar baseline and was explained in the earlier sections (section 3.2.1, page 75).

- Only **one instruction** is **issued** and **committed** per clock cycle.
- All instructions are fetched, decoded, executed, and written back in **strict program order**.
- Hazards (data, structural, control) cause pipeline stalls that affect all subsequent instructions.
- CPI ≥ 1, IPC ≤ 1.

This model is **simple and ensures precise state at all times**, but it is **severely limited in ILP exploitation**.

⧗ Step 2: Single-Issue, Out-of-Order Execution

To overcome unnecessary stalls caused by instruction dependencies, processors began executing instructions **out of order**, while still **issuing only one instruction per cycle** (section 3.2.2, page 78).

- Instructions are fetched and issued **in program order**.
- But **independent instructions** are allowed to **execute and complete out of order**, as soon as their operands are ready and a functional unit is available.
- Techniques like **dynamic scheduling** (e.g., Tomasulo's algorithm) are used to manage dependencies and operand availability.
- A **commit stage ensures in-order architectural updates**, preserving program correctness and exception handling.

This **significantly improves ILP**, but **throughput is still constrained by the single-issue limit**.

Step 3: Multiple-Issue, In-Order Execution (Dual-Issue Pipeline)

This step involves fetching, decoding, and executing **more than one instruction per cycle**, but **in program order**.

- Typical example: **dual-issue pipelines** (e.g., MIPS dual-issue architecture, section 3.4.1, page 83).
- The **hardware allows the issue of up to two instructions per clock**, provided they are independent and compatible (e.g., one ALU + one Load-/Store).
- Requires hardware additions such as:
  - Multiple functional units,
  - Multi-ported register file,
  - Hazard detection logic across simultaneously issued instructions.

This model **increases IPC (ideal IPC = 2)**, but still suffers from limitations:

- ✗ **Dependent instructions must wait**, even if others could proceed.
- ✗ **Static scheduling** (compiler) or simple hardware interlocks determine issue feasibility.

Step 4: Multiple-Issue, Out-of-Order Execution

This is the **most flexible and powerful configuration**, forming the **basis of superscalar processors**.

- Fetches and decodes **multiple instructions per cycle**.
- Uses **dynamic scheduling logic** to decide which subset can be issued and executed out of order.
- Independent **instructions proceed as soon as their operands are ready**, regardless of program order.
- Results are **committed in order** to maintain a precise architectural state.
- This model requires **complex hardware**:
  - Reservation stations
  - Register renaming
  - Reorder buffer (ROB)
  - Instruction window/wakeup-select logic

This architecture provides the **highest ILP**, as it combines the breadth of multiple issue with the flexibility of out-of-order execution.

The transition toward superscalar execution is a gradual process, where each step builds upon the previous to **mitigate limitations**, **maximize resource utilization**, and **exploit greater ILP**. Superscalar architectures represent the culmination of this evolution, dynamically scheduling and executing multiple instructions in parallel, out of order, while maintaining program correctness and exception safety.

Step	**Issue**	**Execution**	**Commit**	**Example**
1	In-order	In-order	In-order	Scalar pipeline
2	In-order	Out-of-order	In-order	Dynamic single-issue
3	In-order	In-order	In-order	Dual-issue pipeline
4	In-order	Out-of-order	In-order	Superscalar processor

Table 10: Evolution towards superscalar execution.

3.4.3 Superscalar Processors

The culmination of the architectural evolution toward higher ILP is the **superscalar processor**, a class of processors capable of **issuing, executing, and committing multiple instructions per clock cycle**, **dynamically and out of order**. Superscalar architectures aim to exploit maximum parallelism hidden within sequential instruction streams, while preserving the illusion of sequential execution.

Definition 3: Superscalar Processor

A **Superscalar Processor** is a dynamically scheduled, multiple-issue architecture capable of issuing and executing **several instructions per clock cycle**, using complex hardware mechanisms to **detect and exploit instruction-level parallelism at runtime**.

Unlike static VLIW processors[3], where instruction parallelism is exposed at compile time, **superscalar designs rely on hardware to discover and schedule parallel instructions on the fly**.

Key Characteristics

1. **Multiple-Issue Width**, also called issue width, is the **maximum number of instructions** a processor can fetch, decode, issue, and begin executing in a **single clock cycle**. It defines the **instruction throughput potential** of a multiple-issue processor.
   - Therefore, this value is given by the superscalar processor, but obviously it is the maximum and it cannot be always reached (hazards, etc.).
   - The maximum number of instructions per single clock cycle **affects the theoretical IPC limit**:

$$\text{CPI}_{\text{ideal}} = \frac{1}{\text{issue width}}$$

2. **Dynamic Scheduling**
   - **Hardware analyzes instruction dependencies in real time**.
   - **Independent instructions** are allowed to proceed **out of order**.
3. **Out-of-Order Execution**
   - Instructions **execute as soon as operands and resources are ready**.
   - Improves pipeline utilization and hides latencies.

[3] **VLIW (Very Long Instruction Word) Processor**: A type of multiple-issue processor where the compiler statically schedules multiple operations into a single wide instruction word. Each operation in the bundle is executed in parallel, assuming they are independent. VLIW architectures rely on simple hardware and place the burden of dependency checking and scheduling on the compiler rather than the processor.

4. **In-Order Commit**
    - Despite out-of-order execution, the processor updates architectural state in program order.
    - Ensures **precise exceptions** and consistent state.

Core Hardware Components

To support superscalar execution, the **architecture must include**:

- **Multiple Functional Units**: ALUs, FPUs, load/store units, enough to support parallel execution.
- **Register Renaming**: Eliminates WAR and WAW hazards by mapping architectural registers to a larger set of physical registers.
- **Reservation Stations / Issue Queues**: Buffer instructions waiting for operands or functional units.
- **Reorder Buffer (ROB)**: Holds results of completed instructions until they are safe to commit. Ensures correct program order and precise exceptions.
- **Common Data Bus (CDB)**: Broadcasts results to dependent instructions.
- **Instruction Window**: Sliding window of in-flight instructions from which the scheduler selects those ready to issue.

Benefits

- **High ILP**: Multiple independent instructions can be executed in parallel.
- **Dynamic Parallelism**: No need for compiler to expose ILP, hardware finds it at runtime.
- **Latency Hiding**: Long operations (e.g., cache misses, FP div) can be overlapped with other instructions.
- **General Purpose**: Works well with a wide range of programs, even without manual tuning.

Challenges

- **Complex Hardware**: Scheduling, renaming, hazard detection logic increases **area and power**.
- **Issue Logic Scalability**: The complexity of deciding which N instructions to issue out of M in-flight grows quickly.

- **Branch and Memory Dependencies**: Control and memory dependencies still limit achievable ILP.
- **Diminishing Returns**: After 3-4 issue width, real programs rarely expose enough parallelism to fully utilize the issue bandwidth.

Superscalar processors represent a powerful solution to the ILP problem, combining **multiple-issue capability** with **hardware-driven dynamic scheduling**. They dominate the high-performance general-purpose processor space (e.g., x86 and ARM cores), though their complexity and scalability limitations have motivated complementary techniques such as multithreading, vectorization, and heterogeneous architectures.

3.4.4 Static vs Dynamic Scheduling

Instruction-level parallelism (ILP) can be **exploited** either **at compile time** by the compiler or **at runtime** by the hardware. These two approaches give rise to two distinct classes of architectures:

- **Statically scheduled processors**, such as VLIW (Very Long Instruction Word) architectures.
- **Dynamically scheduled processors**, such as superscalar architectures.

This section compares these two philosophies, focusing on their mechanisms, advantages, drawbacks, and the contexts in which each is most appropriate.

Scheduling Type	Performed By	When?
Static	Compiler	At compile time
Dynamic	Hardware	At runtime

Table 11: Key difference: who does the scheduling?

What is VLIW (Static Scheduling)?

VLIW (Very Long Instruction Word) is a **statically scheduled**, **multiple issue processor architecture** in which the compiler selects and packs multiple independent operations into a single, wide instruction word that is **executed in parallel** by the processor.

The **instruction word** in a VLIW architecture consists of **multiple operations** (e.g., an ALU op, a memory op, and a floating-point op) that are **intended to be executed simultaneously**. The **compiler is responsible** for:

- Detecting ILP in the program.
- Scheduling independent instructions to avoid hazards.
- Filling empty slots with `NOP`s when no instruction fits.

In contrast to superscalar processors (which discover ILP dynamically at runtime), **VLIW processors rely entirely on compile-time scheduling**.

✔ **Simple hardware**: no need for out-of-order logic, dependency checks at runtime, or renaming hardware.

✔ **Predictable performance**: useful in embedded or real-time systems.

✔ **Energy efficient**: avoids complex runtime scheduling logic.

✖ **Compiler must do all the work**: requires sophisticated analysis and scheduling. So the performance depends on the quality of the compiler and the amount of visible ILP.

✗ **Binary compatibility issues (limited portability)**: compiled code often tied to a specific machine configuration (e.g., number of functional units). Cannot adapt to unpredictable latencies at runtime (e.g., cache misses, branch misprediction).

✗ **Wasted instruction slots**: when insufficient ILP is found, unused slots become `NOP`s, reducing efficiency.

Superscalar: Dynamic Scheduling

In dynamically scheduled architectures:

- The compiler generates sequential code (as usual, no additional effort).
- The processor's **hardware detects ILP at runtime**, using structures like reservation stations, reorder buffers, and register renaming.
- The processor decides **which instructions to issue and execute** based on operand availability and resource status.

✓ Automatically adapts to **unpredictable latencies** and instruction dependencies.

✓ **Improved performance portability**: no need to recompile code for each variant.

✓ Better at **exploiting ILP in general-purpose programs**.

✗ **Higher hardware complexity**, area, and power consumption.

✗ **Scheduling logic becomes a bottleneck** at wider issue widths.

✗ Greater difficulty in verifying and validating timing behavior.

Feature	Static (VLIW)	Dynamic (Superscalar)
Scheduling responsibility	Compiler	Hardware
Instruction issue	Fixed and pre-planned	Determined at runtime
Flexibility at runtime	Low	High
Hardware complexity	Low	High
Compiler complexity	High	Moderate
Portability of compiled code	Low (machine-dependent)	High
Latency tolerance	Poor (fixed schedule)	Good (adaptive execution)
ILP exploitation	Only what compiler exposes	Also includes dynamic/hidden parallelism

Table 12: Static vs Dynamic Scheduling.

Static and dynamic scheduling represent two fundamentally different approaches to exploiting ILP.

- **VLIW** is ideal for **predictable workloads**, embedded systems, or domain specific processors, where simplicity and determinism matter.
- **Superscalar processors** excel in **general-purpose computing**, where **dynamic behavior and runtime variability** make hardware-managed scheduling more effective.

Ultimately, both models aim to improve throughput, but the trade-off between hardware complexity and compiler sophistication defines their respective domains of success.

3.5 ILP Limitations & Alternatives

Instruction-Level Parallelism (ILP) has been the cornerstone of high performance processor design for decades. Techniques such as pipelining, multiple-issue architectures, dynamic scheduling, and register renaming have pushed ILP to impressive levels. However, **ILP alone has fundamental limits**, both theoretical and practical, which restrict its scalability and efficiency in modern workloads. This section explores **why ILP hits a wall**, and how architects are moving toward **complementary and alternative forms of parallelism**, including multithreading, SIMD, and heterogeneous computing, to sustain performance growth.

⚠ Limitations of ILP

1. **Limited Parallelism in Programs**

   - Many **programs are inherently sequential in logic** (e.g., control-intensive code, algorithms with tight dependencies).
   - Available **ILP is often limited to short instruction windows**. An **instruction window** is the set of instructions that the processor can see and analyze at a give time to find parallelism. Due to hardware constraints (area, power, timing), the instruction window usually holds a few dozen to a few hundred instructions.

     It means that even if the entire program contains parallelism, the processor can only exploit what it sees in its current instruction window. This is a practical limit.
   - **Amdahl's Law bounds the speedup achievable by parallel execution of a sequential program.**[4] Even if we have unlimited issue width, perfect branch prediction, and ideal memory, this law says that we can't speed up the parts of the program that are inherently sequential (e.g., control logic, data dependencies). For example, if 10% of our program is serial ($f = 0.1$), then the best speedup we can ever get is: $\frac{1}{0.1} = 10$.

[4] **Amdahl's Law** states that the **maximum theoretical speedup** of a program is **limited by the fraction of the program that must be executed sequentially**, even if the rest can be infinitely accelerated or parallelized. Let:

- S: speedup
- f: fraction of the program that is serial (cannot be parallelized)
- $(1 - f)$: fraction that is parallelizable
- p: speedup of the parallel portion (e.g., number of processors)

Then:

$$\text{Speedup}\,(p) = \frac{1}{f + \frac{(1-p)}{p}}$$

And the maximum speed (when $p \to \infty$) is:

$$\text{Speedup}_{\text{max}} = \frac{1}{f}$$

Even with infinite hardware resource, the speedup is limited by the serial part of the code.

2. **Dependency Constraints**
    - **True data dependences (RAW) cannot be bypassed or parallelized** because the value doesn't exist yet. No matter how many cores,execution units, or parallel tricks we have, if one instruction computes a value that another must use, the second must wait for the first to finish.
    - Some instructions must wait for preceding results, creating **bubbles** in the issue pipeline.
3. **Control Dependencies and Branches**
    - Branches disrupt instruction flow.
    - Even with speculative execution and prediction, **mispeculation causes flushes and wasted cycles**.
4. **Memory Latency and Aliasing**
    - Cache misses introduce long, unpredictable delays.
    - **Memory dependencies** (e.g., between loads and stores) are difficult to resolve safely at runtime, limiting aggressive scheduling.
5. **Hardware Complexity and Power**
    - The **logic** needed for dependency checking, wakeup-select, register renaming, and instruction window scaling **grows rapidly**.
    - Superscalar processors beyond 4-6 issue width become infeasible to scale due to **power, area, and control path complexity**.

Alternative Forms of Parallelism

1. **Thread-Level Parallelism (TLP) - Multithreading**. Execute **multiple threads** in parallel on the same core or across multiple cores. TLP hides long-latency events (e.g., cache misses) by switching to ready threads.
2. **Data-Level Parallelism (DLP) - SIMD and Vectorization**. Exploits uniform operations over data arrays (e.g., matrix ops, DSP, graphics). Single Instruction, Multiple Data (SIMD), one instruction operates on multiple data elements.
3. **Heterogeneous Computing - Specialized Accelerators**. Use of domain-specific architectures (DSAs) optimized for specific tasks: GPU, TPUs, NPUs, FPGAs. Offload compute-intensive or parallel workloads from the CPU to accelerators.

3.6 Scoreboard: Dynamic Scheduling Algorithm

3.6.1 Assumptions and Architecture

The **Scoreboard** is a **dynamic scheduling** mechanism introduced in the CDC 6600 that enables **out-of-order execution** while maintaining program correctness. It coordinates the flow of instructions in a way that allows independent instructions to execute in parallel, despite pipeline stalls caused by data or structural hazards. This approach was fundamental to enhancing instruction-level parallelism (ILP) without relying on complex compiler-level optimizations.

Assumptions of the Scoreboard Model

To analyze the behavior of the scoreboard, it's crucial to understand the initial architectural assumptions:

- **Single-Issue Processor**: only one instruction can be fetched and issued per cycle, enforcing a serialized dispatching model despite the internal parallelism.
- **In-Order Issue**: instructions are issued in the program order (page 74). However, once issued, they are allowed to execute and complete out-of-order depending on operand availability.
- **No Forwarding Mechanism**: unlike Tomasulo's algorithm, which allows results to be forwarded from functional units directly to waiting instructions, the Scoreboard lacks this feature. Operands are only considered available once written back to the Register File (RF).
- **Multiple Pipelined Functional Units (FUs)**: the architecture assumes the presence of multiple pipelined FUs, e.g. floating-point add, multiply, divide, and integer units; each with potentially variable latency.
- **Latency-Aware Execution**: both the **Execution Stage (EX)** and **Memory Access Stage (ME)** are allowed to span multiple cycles depending on the operation type and cache behavior.
- **Out-of-Order Execution and Commit**: execution and result write-back (or commit) can happen out-of-order, introducing hazards such as:
  - Write After Write (WAW)
  - Write After Read (WAR)

  These are especially critical since there's no register renaming mechanism (page 71) to avoid false dependencies.

This configuration **allows** the scoreboard to **bypass pipeline stalls** by executing independent instructions out-of-order, while relying on a **centralized control logic to track hazards and resource usage**.

Architectural Scheme

The Scoreboard orchestrates **execution by separating three phases**:

1. **Instruction Issue (in-order)**
2. **Instruction Execution (out-of-order)**
3. **Instruction Completion (out-of-order)**

This setup breaks the rigid in-order pipeline flow and increases functional unit utilization. Furthermore:

- Multiple instructions can be in execution simultaneously.
- Precise exceptions are **not** guaranteed due to the possibility of earlier instructions committing after later ones, a model referred to as **imprecise interrupts**.

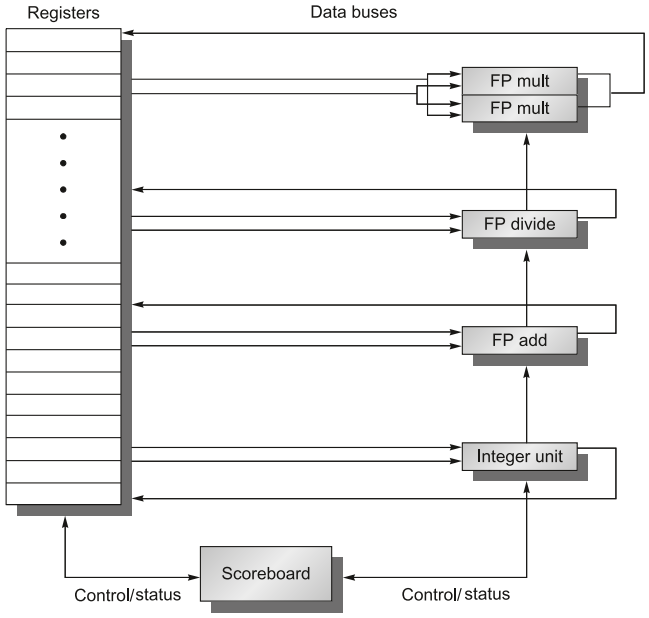


Figure 24: The basic structure of a RISC V processor with a scoreboard. [2]

- A shared Register File feeds data into multiple data buses.
- Each Functional Unit (FU) is independently pipelined and connected to the scoreboard. Units include two FP multipliers, FP adder, FP divider, and integer unit.
- A centralized Scoreboard logic block maintains: Control/Status signals, Dependency tracking, Issue constraints.
- There's a separate Memory Unit, handled similarly to functional units, responsible for data memory operations.

3.6.2 Pipeline Stage Refinement

In a **traditional pipeline**:

- The Instruction Decode (ID) stage performs both decoding and operand reading at once.
- But this **assumes operands are always ready**, which is **not true** in a dynamically scheduled out-of-order pipeline.

So the **scoreboard splits Instruction Decode (ID) stage into**:

1. **Issue Stage**
   - Responsible for **decoding the instruction**.
   - Checks for **structural hazards** (page 19), particularly whether the appropriate functional unit (FU) is available and whether the scoreboard's bookkeeping allows the instruction to proceed (this will become clearer later).
   - Enforces **in-order issue**, instructions are considered strictly in the sequence fetched from memory.
2. **Read Operands Stage (RR)**
   - Waits until **operands are available and not blocked** by earlier instructions.
   - Specifically, **avoids RAW** (Read After Write) hazards (page 19) by deferring operand reads until the register is no longer "reserved" by an active instruction writing to it. In other words, delays reading operands **until they're truly ready**, that is, the producer instruction has completed writing them.
   - **Operands** are then **read from the register file** (since forwarding is not available).

This separation increases the scoreboard's ability to exploit Instruction-Level Parallelism (ILP) while maintaining control over dependency tracking and avoiding illegal hazards.

Flexible Execution Behavior

After the two front-end stages:

- **Out-of-Order Execution**: Once operands are read, instructions may **enter the execution stage as soon as the corresponding FU is available**, regardless of program order.
- **Variable Latency Handling**: Functional units (FUs) may have different latencies (e.g., FP divide vs. add), so instructions finish execution at different times and **write back their results out-of-order**.
- **Out-of-Order Commit**: Because instructions complete independently and there is **no reorder buffer**, the commit (or write-back) stage is also **out-of-order**, unlike more modern precise pipelines.

Stage	Behavior	Order Enforcement
Issue	Decode, FU check	**In-Order**
Read Operands	Wait for availability	**Out-of-Order**
Execute	Run in FU	**Out-of-Order**
Write Result	Commit to Reg. File	**Out-of-Order**

Table 13: Key features of the scoreboard.

What is the scoreboard constantly tracking?

We can think of the scoreboard as a "Control Office" inside the processor. The main job of the scoreboard is to **keep tack of every instruction** that's in the pipeline at the same time, and **make sure they don't mess each other up**.

To do this, it monitors four key things:

1. **Availability of Source Operands**. Every instruction needs to read its inputs (like F2, F4, etc.). The scoreboard checks: **are those registers ready, or is another instruction still going write them (busy)?** If they're **not ready** yet, the **instruction waits** in the Read Operands (RR) stage.

   Why? To **avoid RAW** (Read After Write) hazards, reading too early before the data is correct.

2. **Status of each Functional Unit (FU)**. It knows which **units** (like the adder, multiplier) are **busy** or **free**. It won't assign two instructions to the same FU at the same time, that would **cause structural hazards**.

   Why? So it knows **which instruction can be issued** and which needs to **wait for hardware**.

3. **Pending writes and register conflicts**. If two instructions plan to write to the same register, it keeps track of this. This **helps prevent**:

   - ✔ **WAW** (Write After Write): two instructions writing to the same register in the wrong order.
   - ✔ **WAR** (Write After Read): an instruction overwriting a value that another one still needs to read.

   Why? This avoids **wrong results**, even if instructions execute out-of-order.

4. **Which instructions have completed**. It **tracks** when each instruction **finishes execution** and when it's **allowed to write back** its result.

   Remember: there's **no reorder buffer**, so the scoreboard must **carefully manage write-backs to prevent conflicts**.

   Why? So it knows when to **release resources** and **update registers safely**.

3.6.3 Hazard Management (RAW, WAR, WAW)

A hazard occurs when the pipeline execution of instructions might lead to incorrect results. Hazards arise from:

- Resource conflicts
- Data dependencies
- Instruction ordering mismatches

The **scoreboard handles three types of data hazards dynamically**:

RAW (Read After Write) - True Dependency

Occurs when:

- Instruction `B` tries to **read** a register before instruction `A` has **written** its result.

Scoreboard handling:

- The scoreboard stalls `Instruction B` in the Read Operands (RR)[5] stage until `Instruction A` completes its write.

✔ **Handled in the RR (Read Operands) stage**

WAR (Write After Read) - Anti-Dependency

Occurs when:

- *Instruction A* needs to **read** a register before `Instruction B` **overwrites** it.

Scoreboard handling:

- The scoreboard **stalls** `Instruction B` in the **Write Result** stage until `Instruction A` has read the register.

✔ **Handled in the WR (Write Result) stage**

WAW (Write After Write) - Output Dependency

Occurs when:

- Two instructions write to the same register in the wrong order.

Scoreboard handling:

- The scoreboard **stalls the second instruction** in the **Issue** stage until the first instruction has written its result.

✔ **Handled in the Issue stage (or sometimes delayed to Write stage)**

[5]Note that the Read Operands (RR) stage is the second stage in the Instruction Decode stage. Its purpose is to wait until all source operands are available and not blocked by an active instruction. Only then can the instruction safely read its operands from the register file.

So Scoreboard solves WAR/WAW **explicitly via stalls**, instead of using *register renaming*. This makes scoreboard simpler, but **limits how much parallelism it can safely exploit** compared to more modern approaches.

Hazard	Cause	Scoreboard Action	Handled In
RAW	Read before prior write	Stall reader until value ready	**Read Operands**
WAR	Write before earlier read	Stall writer until read completes	**Write Result**
WAW	Write before earlier write	Stall issuer	**Issue**

Table 14: Hazards managed by the scoreboard.

3.6.4 Control Logic and Stages

The Scoreboard architecture divides instruction **execution into four dynamic control stages**, each governed by centralized logic. These stages (**Issue**, **Read Operands**, **Execution**, and **Write Result**), replace the traditional ID, EX, and WB stages of a standard RISC pipeline.

The key idea is that the scoreboard **monitors dependencies and structural hazards in hardware** and makes real-time decisions on when each instruction can safely advance to the next stage.

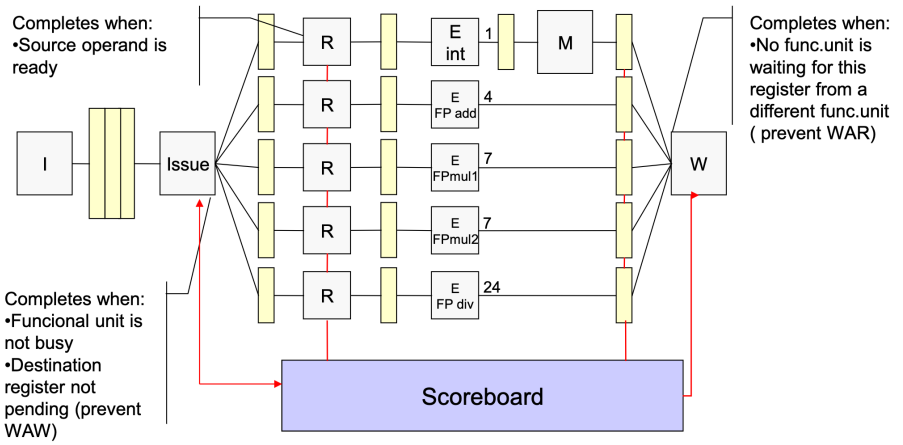


Figure 25: The Scoreboard architecture. Pipeline Flow is: Instruction Fetch (I), Read Operands Blocks (R), Execution Units (E), Memory Stage (M) and Write Result Stage (W).

1. **Issue Stage (In-Order)**. This is the first stage after Instruction Fetch (IF).
   (a) The instruction is **decoded**.
   (b) The scoreboard **checks**:
      i. **Structural Hazards**: *is the required Function Unit (FU) available?*
      ii. **WAW (Write After Write) Hazards**: *is another instruction already writing to the destination register?*

   If **neither hazard exists**, the **instruction is issued** and marked in the scoreboard's internal tables. **Otherwise**, the **instruction stalls**.

   Performance Optimizations: **WAW hazards** are typically **detected** here, but optimizations may postpone this to the **write-back stage**.

2. **Read Operands Stage (Out-of-Order)**. The scoreboard waits for both source operands to become available.
   (a) **RAW** (Read After Write) hazards are **checked dynamically**.
   (b) If any **operand** is still **pending** (i.e. will be written by another instruction), the **scoreboard stalls this stage**.
   (c) Once **ready**, operands are **read from the Register File** (RF) (no forwarding!).
   (d) The instruction is then sent to the Functional Unit (FU) to begin execution.

   This is the **stage that enables out-of-order execution**, as independent instructions may pass each other based on operand readiness.

3. **Execution Stage**
   (a) The instruction **executes** in the assigned Functional Unit.
   (b) Functional Units (FUs) may have **variable latency**, depending on the operation (e.g., divides vs. add).
   (c) Upon completion, the **unit signals the scoreboard**.

   This phase also includes **additional memory access latency for load-/store instructions** affected by cache hit/miss.

4. **Write Result Stage (Out-of-Order)**
   (a) Before writing the result to the destination register, the **scoreboard checks** for:
   - **WAR** (Write After Read) hazards: *is any previous instruction still waiting to read this register?*
   - **Structural hazards**: *are the Register File (RF) write ports available?*

   If **clear**, the **instruction writes back** the result. If a **WAR hazard exists**, the **scoreboard stalls this instruction** until the reading instruction completes.

Stage	Hazard Checked	Order	Description
Issue	Structural, WAW	In-Order	FU availability + dest reg conflict
Read Operands	RAW	Out-of-Order	Waits for source operands
Execute	-	Out-of-Order	Runs in FU (latency varies)
Write Result	WAR, Structural (RF)	Out-of-Order	Writes result if safe

Table 15: Hazards managed by the scoreboard.

3.6.5 Summary

In this section, we present a summary of the Scoreboard's dynamic scheduling algorithm.

The Scoreboard implements a **classic dynamic scheduling mechanism** where instruction progress through the pipeline is dictated not just by structural availability, but also by **true data readiness**.

Pipeline Stage	Order	Hazard Checked	Notes
Issue	**In-order**	Structural, WAW	Instruction decoded, FU reserved
Read Operands	**Out-of-order**	RAW, structural (RF ports)	Wait for all inputs to be ready
Execution	**Out-of-order**	-	Variable latency depending on FU
Write Result	**Out-of-order**	WAR, structural (write port)	Write to reg file if safe

The scoreboard enforces **precise tracking** at each stage to dynamically resolve hazards without register renaming or forwarding.

Execution Properties

- **In-Order Issue**
  - Simplifies the hardware: instructions are always issued in program order.
  - Helps **detect WAW hazards** early.
- **Out-of-Order Read Operands**
  - Once issued, instructions **wait until all operands are available**, then read them from the Register File (RF).
  - No data forwarding! **Operands are read only from the Register File** (RF).
  - Allows **independent instructions** to leapfrog stalled ones.
- **Out-of-Order Execution**
  - Instructions execute as soon as their operands are ready and the FU is free.
  - Multiple instructions can execute **simultaneously** in parallel FUs or pipelined units.
  - Leads to **higher FU utilization and throughput**.
- **Out-of-Order Completion**
  - Results are written back when ready, **unless a WAR hazard** is detected.
  - This breaks precise exception semantics, i.e., exceptions can be **imprecise**.

No Forwarding, No Renaming

- **No data forwarding**: causes **extra stalls** at operand read stage.
- **No register renaming**: makes the scoreboard vulnerable to WAR and WAW hazards, which it **handles by stalls and centralized checks**.

Control Logic Centralization

All control decisions (hazard detection, operand availability, resource usage) are made by a **central scoreboard table**. This avoids complex distributed hardware (as in Tomasulo), but limits the potential for speculation or aggressive scheduling.

3.6.6 Scoreboard Data Structures

At the heart of the scoreboard's centralized control logic are **three hardware data structures** that track the status of instructions, functional units, and register dependencies. These structures allow the scoreboard to make safe, real-time decisions about instruction scheduling, execution, and result writing, all while avoiding hazards.

1. **Instruction Status Table**. Tracks the **lifecycle of each instruction** through the pipeline. For each instruction, the scoreboard record whether it has:

   - Been **issued**
   - **Read operands**
   - **Completed execution**
   - **Written back** the result

   We can think of this as a per-instruction timeline: it tracks which stage the instruction is currently in.

2. **Functional Unit Status Table**. Tracks the **current state** of each **Functional Unit** (FU). Each FU entry includes:

   - `Busy`: whether the FU is currently in use.
   - `Op`: operation being performed (e.g., ADD, MULT).
   - `Fi`: destination register of the operation.
   - `Fj`, `Fk`: source register operands.
   - `Qj`, `Qk`: functional units producing `Fj` and `Fk`.
   - `Rj`, `Rk`: boolean flags indicating if `Fj`, `Fk` are ready.

   These fields help the scoreboard:

   (a) Decide when operands are ready (for RAW)
   (b) Prevent WAW and WAR
   (c) Handle operand read scheduling

3. **Register Result Status Table**. Tracks **which FU will produce each register value**. For each register (e.g., `F0`, `F2`, . . . , `F30`), it stores:

   - The **name of the FU** that will write to it.
   - Or blank (−, don't care) if no instruction is scheduled to write it.

   This structure is essential to:

   - Detect WAW hazards at **issue** stage.
   - Detect WAR hazards at **write-back** stage.
   - Ensure only the latest producing instruction claims the register.

Example 6

Let's say `MULT f0, f2, f4` is issued to `Mult1` (functional unit). The scoreboard will:

1. Mark `Mult1` as `Busy`
2. Set
   - `Op = MULT`
   - `Fi = F0`
   - `Fj = F2`
   - `Fk = F4`
3. Fill `Qj` and `Qk` if other FUs are writing `F2` or `F4`
4. In the **Register Result Status**, assign `F0 = Mult1`

This coordination ensures:

- Other instructions know `F0` will be produced by `Mult1`
- `F2` and `F4` are only read when available
- Subsequent instructions that depend on `F0` will wait

3.6.7 In-Depth Execution Example

The goal of this section is to observe how the scoreboard manages dependencies, tracks resource usage, and handles all threats over time using an example.

★ Initial Setup: Instruction List and Dependencies

The instructions are:

```
LD     F6, 34(R2)
LD     F2, 45(R3)
MULTD  F0, F2, F4       # RAW on F2
SUBD   F8, F6, F2       # RAW on F6, F2
DIVD   F10, F0, F6      # RAW on F0, F6
ADDD   F6, F8, F2       # WAW & WAR on F6, RAW on F8 & F2
```

We have a mix of:

- RAW hazards: `F2`, `F6`, `F0`, `F8`
- WAW/WAR: around register `F6`

During the example, we will show the status of three main hardware data structures introduced in the section 3.6.6, page 107:

- **Instruction Status Table**: tracks the lifecycle state of each instruction in the pipeline.
- **Functional Unit Status Table**: tracks the usage and readiness of each functional unit (FU), and the dependency state of the operands.
- **Register Result Status Table**: tracks which FU is **scheduled to write** to each floating-point register.

1. **Cycle 1**
   - `LD F6, 34(R2)` is issued and begins execution.
   - Integer unit is now **busy**.
   - All other instructions wait.

   No hazard yet. This sets up the first data dependency (`F6` will be written soon)

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1			
`LD F2, 45(R3)`				
`MULTD F0, F2, F4`				
`SUBD F8, F6, F2`				
`DIVD F10, F0, F6`				
`ADDD F6, F8, F2`				

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
1				Integer					

Register result status.

2. **Cycle 2**

   ⚠ Cannot issue `LD F2, 45(R3)` yet, due to **structural hazard** on the `Integer unit`.

   - Execution of first `LD` continues.

   Stall due to structural hazard, despite in-order issue.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2		
`LD F2, 45(R3)`				
`MULTD F0, F2, F4`				
`SUBD F8, F6, F2`				
`DIVD F10, F0, F6`				
`ADDD F6, F8, F2`				

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	`Integer`	Yes	`Load`	`F6`		`R2`				Yes
	`Mult1`	No								
	`Mult2`	No								
	`Add`	No								
	`Divide`	No								

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
2				`Integer`					

Register result status.

3. **Cycle 3**

   - First load finishes execution.
   - Still can't issue second load.

   Memory latency is ideal (1 cycle), but scoreboard doesn't allow overcommit of the integer unit.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	
`LD F2, 45(R3)`				
`MULTD F0, F2, F4`				
`SUBD F8, F6, F2`				
`DIVD F10, F0, F6`				
`ADDD F6, F8, F2`				

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
3				Integer					

Register result status.

4. **Cycle 4**

   - `F6` is written to the register file (RF)
   - `Integer` unit is now **free**.

   Register `F6` becomes available, other instructions depending on it can now move (when their turn comes).

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`				
`MULTD F0, F2, F4`				
`SUBD F8, F6, F2`				
`DIVD F10, F0, F6`				
`ADDD F6, F8, F2`				

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	`Integer`	No								
	`Mult1`	No								
	`Mult2`	No								
	`Add`	No								
	`Divide`	No								

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
4				`Integer`					

Register result status.

5. **Cycle 5**

   - `LD F2, 45(R3)` is issued and starts execution.
   - `Instruction 2` enters pipeline, finally.

   No data hazards here, but we're about to enter the **RAW jungle** starting next cycle.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5			
`MULTD F0, F2, F4`				
`SUBD F8, F6, F2`				
`DIVD F10, F0, F6`				
`ADDD F6, F8, F2`				

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	`Integer`	Yes	`Load`	`F2`		`F3`				Yes
	`Mult1`	No								
	`Mult2`	No								
	`Add`	No								
	`Divide`	No								

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
5		`Integer`							

Register result status.

6. **Cycle 6**

   ✔ `MULTD F0, F2, F4` is issued to `Mult1`.

   ✖ **RAW hazard** on `F2`: operand not yet available (second load not completed).

   - `MULTD` **waits** in the Read Operands stage.
   - `LD F2` is executing (started in Cycle 5).

   Highlights:

   ✔ `MULTD` **is issued** because: the `Mult1` functional unit is free (no structural hazard), and no other instruction is writing to `F0` (no WAW hazard).

   ✖ But execution (of `MULTD`) is blocked because:

   – `F2` (a source operand) is still being loaded by `LD F2`, and this is a RAW (Read After Write) hazard.

   – `MULTD` must **wait until** `LD F2` **writes its result into register** `F2`.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6		
`MULTD F0, F2, F4`	6			
`SUBD F8, F6, F2`				
`DIVD F10, F0, F6`				
`ADDD F6, F8, F2`				

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	Yes	Load	F2		F3				Yes
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	No								
	Divide	No								

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
6	Mult	Integer							

Register result status.

7. **Cycle 7**

   - ✔ `LD F2` completes (data cache hit), result will be written to `F2` next cycle.
   - ✔ `SUBD F8, F6, F2` is **issued** to `Add` unit (free).
   - ✘ But `SUBD` cannot start execution yet:
     - It needs `F6` and `F2` as operands.
     - `F2` just finished loading and is **not yet written back**, so still blocked (**RAW hazard**).
     - `F6` was written earlier by `LD F6` and is now available.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	
`MULTD F0, F2, F4`	6			
`SUBD F8, F6, F2`	7			
`DIVD F10, F0, F6`				
`ADDD F6, F8, F2`				

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	Yes	Load	F2		F3				Yes
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
	Divide	No								

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
7	Mult	Integer			Add				

Register result status.

8. **Cycle 8**

   ✔ `DIVD F10, F0, F62` is **issued** to `Divide` unit (free).

   ✔ `LD F2` **writes back to** `F2`, now `MULTD` and `SUBD` can read `F2` (cycle 9).

   ✘ But execution is blocked because:

   – It needs `F0` (being written by `MULTD`), so **RAW hazard** on `F0`.

   – It also uses `F6`, which is already ready.

   – `F2` just finished loading and is **not yet written back** (this cycle), so still blocked (**RAW hazard**).

This stage shows **cascading dependency chains** forming: `DIVD` waits for `MULTD`, which waits for `LD F2`.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6			
`SUBD F8, F6, F2`	7			
`DIVD F10, F0, F6`	8			
`ADDD F6, F8, F2`				

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	`Integer`	No								
	`Mult1`	Yes	`Mult`	`F0`	`F2`	`F4`			Yes	Yes
	`Mult2`	No								
	`Add`	Yes	`Sub`	`F8`	`F6`	`F2`		`Integer`	Yes	Yes
	`Divide`	Yes	`Div`	`F10`	`F0`	`F6`	`Mult`		No	Yes

Functional unit status.

Clock	`F0`	`F2`	`F4`	`F6`	`F8`	`F10`	`F12`	...	`F30`
8	`Mult`				`Add`	`Divide`			

Register result status.

9. **Cycle 9**

   ✔ `MULTD` and `SUBD` **read operands** (in parallel): scoreboard uses a **multi-port register file** (e.g., 4 read ports) to allow simultaneous operand reads.

   - `MULTD` begins execution on `Mult` unit (10-cycle latency, table "Functional unit status").
   - `SUBD` begins execution on `Add` unit (2-cycle latency).

   ✘ `ADDD` cannot be issued because the `Add` unit is **already in use by** `SUBD` (structural hazard).

Scoreboard enables **parallel out-of-order read** and execution.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9		
`SUBD F8, F6, F2`	7	9		
`DIVD F10, F0, F6`	8			
`ADDD F6, F8, F2`				

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	`Integer`	No								
10	`Mult1`	Yes	`Mult`	`F0`	`F2`	`F4`			Yes	Yes
	`Mult2`	No								
2	`Add`	Yes	`Sub`	`F8`	`F6`	`F2`		`Integer`	Yes	Yes
	`Divide`	Yes	`Div`	`F10`	`F0`	`F6`	`Mult`		No	Yes

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
9	`Mult`				`Add`	`Divide`			

Register result status.

10. **Cycle 10**

    - `MULTD` and `SUBD` are executing.
    - `DIVD` is still waiting for `F0` (from `MULTD`).
    - `ADDD` remains stalled.

    This block shows classic **out-of-order read and execution** behavior.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9		
`SUBD F8, F6, F2`	7	9		
`DIVD F10, F0, F6`	8			
`ADDD F6, F8, F2`				

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
9	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
1	Add	Yes	Sub	F8	F6	F2		Integer	Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult		No	Yes

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
10	Mult				Add	Divide			

Register result status.

11. **Cycle 11**

- ✔ `SUBD` finishes execution (2-cycle latency complete).
- ✔ `SUBD` is now ready to write back to `F8`.
- ✘ But `SUBD` **cannot write yet**, it will occur in the next cycle.
- `MULTD` is still executing (10-cycle latency).
- `DIVD` continues to **wait for** `F0`, which is still in production by `MULTD`.
- `ADDD` remains **stalled**, since the `Add` unit is busy with `SUBD`.

The key idea here is that the scoreboard **only allows one instruction to be written per cycle**; even completed executions must wait their turn.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9		
`SUBD F8, F6, F2`	7	9	11	
`DIVD F10, F0, F6`	8			
`ADDD F6, F8, F2`				

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
8	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
0	Add	Yes	Sub	F8	F6	F2		Integer	Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult		No	Yes

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
11	Mult				Add	Divide			

Register result status.

12. **Cycle 12**

- ✔ `SUBD` writes result to `F8`
- ✔ The `Add` **unit becomes available** again.
- `MULTD` is still executing.
- `DIVD` still waiting for `F0`.
- `ADDD` can now potentially be issued, since the `Add` **unit is no longer busy**.

Hazard check: the **WAW hazard on** `F6` (from `ADDD`) is now **clear** because no one is writing to `F6` at this moment.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9		
`SUBD F8, F6, F2`	7	9	11	12
`DIVD F10, F0, F6`	8			
`ADDD F6, F8, F2`				

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	`Integer`	No								
7	`Mult1`	Yes	`Mult`	`F0`	`F2`	`F4`			Yes	Yes
	`Mult2`	No								
	`Add`	No								
	`Divide`	Yes	`Div`	`F10`	`F0`	`F6`	`Mult`		No	Yes

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
12	`Mult`					`Divide`			

Register result status.

13. **Cycle 13**

- ✔ `ADDD` is **issued** to the `Add` unit: no structural hazard and WAW on `F6` is clear.
- ✔ It **waits** in the Read Operands (RR) stage.
- ✘ Still blocked by a **WAR hazard on** `F6`:
  - `DIVD` is supposed to read `F6`, but `ADDD` wants to write it.
  - `ADDD` must wait until `DIVD` reads `F6` to avoid overwriting it too early.
- `MULTD` still executing.

Classic WAR hazard: writer (`ADDD`) must not overwrite a register untile readers (`DIVD`) finish reading it.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9		
`SUBD F8, F6, F2`	7	9	11	12
`DIVD F10, F0, F6`	8			
`ADDD F6, F8, F2`	13			

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	`Integer`	No								
6	`Mult1`	Yes	`Mult`	`F0`	`F2`	`F4`			Yes	Yes
	`Mult2`	No								
	`Add`	Yes	`Add`	`F6`	`F8`	`F2`			Yes	Yes
	`Divide`	Yes	`Div`	`F10`	`F0`	`F6`	`Mult`		No	Yes

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
13	`Mult`			`Add`		`Divide`			

Register result status.

14. **Cycle 14**

- ✔ `ADDD` reads its operands `F8` and `F2`:
  - `F8` just written by `SUBD` (cycle 12).
  - `F2` is available since cycle 8.
- ✖ `DIVD` is **still waiting** on `F0` (from `MULTD`), so hasn't read `F6` yet.
- Event though `DIVD` was issued before `ADDD`, here `ADDD` **performs operand read first**. This is **out-of-order read**!

Scoreboard allows **out-of-order read operands** when no hazards exist.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9		
`SUBD F8, F6, F2`	7	9	11	12
`DIVD F10, F0, F6`	8			
`ADDD F6, F8, F2`	13	14		

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
5	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
2	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult		No	Yes

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
14	Mult			Add		Divide			

Register result status.

15. **Cycle 15**

✔ `ADDD` **starts execution** on `Add` unit.

- `MULTD` and `DIVD` still wait:
  - `MULTD` still processing.
  - `DIVD` still cannot read `F0`, so it also hasn't read `F6`; keeping the **WAR hazard active** on `F6`.

Execution parallelism: `ADDD` and `MULTD` are both executing, in separate units.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9		
`SUBD F8, F6, F2`	7	9	11	12
`DIVD F10, F0, F6`	8			
`ADDD F6, F8, F2`	13	14		

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
4	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
1	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult		No	Yes

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
15	Mult			Add		Divide			

Register result status.

16. **Cycle 16**

✔ `ADDD` **finishes execution**.

✖ **Cannot write back yet** due to **WAR hazard**:

- `ADDD` wants to write to `F6`, but `DIVD` hasn't yet read it.

- `DIVD` is **still stalled**, waiting for `F0` (result of `MULTD`).
- `MULTD` continues execution (almost done!).

Scoreboard protection: even though `ADDD` finished early, it must **wait** to avoid corrupting data `DIVD` still needs.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9		
`SUBD F8, F6, F2`	7	9	11	12
`DIVD F10, F0, F6`	8			
`ADDD F6, F8, F2`	13	14	16	

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
3	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
0	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult		No	Yes

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
16	Mult			Add		Divide			

Register result status.

17. **Cycle 17**

- Same situation:
  - `ADDD` is waiting to write to `F6`.
  - `DIVD` is waiting to read both `F0` and `F6`.

⚠ **WAR hazard** on `F6` **still active**.

- `MULTD` is **almost done** (last cycle of execution).

Note: WAR hazards delay write-back, not execution.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9		
`SUBD F8, F6, F2`	7	9	11	12
`DIVD F10, F0, F6`	8			
`ADDD F6, F8, F2`	13	14	16	

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
2	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult		No	Yes

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
17	Mult			Add		Divide			

Register result status.

18. **Cycle 18**

- Again, same situation:
  - `ADDD` is **still waiting** to write `F6` (WAR hazard).
  - `DIVD` is still blocked from reading `F0` (pending from `MULTD`).
- `MULTD` completes execution (latency ends here)

Now the scoreboard is ready for `MULTD` to write its result.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9		
`SUBD F8, F6, F2`	7	9	11	12
`DIVD F10, F0, F6`	8			
`ADDD F6, F8, F2`	13	14	16	

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
1	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult		No	Yes

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
18	Mult			Add		Divide			

Register result status.

19. **Cycle 19**

- ✔ `MULTD` **writes** to `F0`.
- ✔ This finally allows `DIVD` to read operands in the **next cycle** (20).
- ⚠ `ADDD` still waits on WAR hazard, `DIVD` still hasn't read `F6`.

This is a key cycle: **RAW** on `F0` is **resolved**, allowing `DIVD` to make progress at last.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9	19	
`SUBD F8, F6, F2`	7	9	11	12
`DIVD F10, F0, F6`	8			
`ADDD F6, F8, F2`	13	14	16	

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
0	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult		No	Yes

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
19	Mult			Add		Divide			

Register result status.

20. **Cycle 20**

- ✔ `DIVD` reads both `F0` and `F6`:
  - `F0` from `MULTD`, now available
  - `F6` from earlier `LD`.
- ✔ With both operands read, **WAR hazard on `F6` is gone**.
- ✔ This **unlocks** `ADDD`, which can now write `F6`.

Scoreboard logic synchronizes dependent events: `DIVD` completes operand read, then `ADDD` can write, then hazard avoided.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9	19	20
`SUBD F8, F6, F2`	7	9	11	12
`DIVD F10, F0, F6`	8			
`ADDD F6, F8, F2`	13	14	16	

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6			Yes	Yes

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
20				Add		Divide			

Register result status.

21. **Cycle 21**

- ✔ `DIVD` finally **reads its operands**:
  - `F0` written by `MULTD` in cycle 19.
  - `F6` read now, which **clears the WAR hazard** blocking `ADDD`.
- ✔ `ADDD` can now **safely write back to** `F6` in the **next cycle**.

Key transitions: `DIVD` finishes operand read (last one of the program); `ADDD` gets the green ligth to write since no instructions are waiting to read `F6`.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9	19	20
`SUBD F8, F6, F2`	7	9	11	12
`DIVD F10, F0, F6`	8	21		
`ADDD F6, F8, F2`	13	14	16	

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	`Integer`	No								
	`Mult1`	No								
	`Mult2`	No								
	`Add`	Yes	`Add`	`F6`	`F8`	`F2`			Yes	Yes
40	`Divide`	Yes	`Div`	`F10`	`F0`	`F6`			Yes	Yes

Functional unit status.

Clock	`F0`	`F2`	`F4`	`F6`	`F8`	`F10`	`F12`	...	`F30`
21				`Add`		`Divide`			

Register result status.

22. **Cycle 22**

✔ `ADDD` **writes result to** `F6`: this completes the final **WAW dependency** involving `F6`.

- `DIVD` is now **executing** (started after operand read).
- No structural or data hazards remain, all previous dependencies are solved.

Everything is now in-flight or completed:

- All operands have been read.
- All issued instructions are either executing or have completed.
- The scoreboard is now **idling** except for the ongoing `DIVD`.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9	19	20
`SUBD F8, F6, F2`	7	9	11	12
`DIVD F10, F0, F6`	8	21		
`ADDD F6, F8, F2`	13	14	16	22

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	`Integer`	No								
	`Mult1`	No								
	`Mult2`	No								
	`Add`	No								
39	`Divide`	Yes	`Div`	`F10`	`F0`	`F6`			Yes	Yes

Functional unit status.

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
22						Divide			

Register result status.

61. **Cycle 61**

✔ `DIVD` **completes its execution**.

– Recall: `DIVD` was issued in cycle 8, and with a long latency (40 cycles), it finally ends execution here.

- The scoreboard marks the `Divide` functional unit as **ready to write**.

This shows how the scoreboard tracks long-latency FUs **without blocking the pipeline**. Other instructions have long since finished.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9	19	20
`SUBD F8, F6, F2`	7	9	11	12
`DIVD F10, F0, F6`	8	21	61	
`ADDD F6, F8, F2`	13	14	16	22

Instruction status.

Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	`Integer`	No								
	`Mult1`	No								
	`Mult2`	No								
	`Add`	No								
0	`Divide`	Yes	`Div`	`F10`	`F0`	`F6`			Yes	Yes

Functional unit status.

Clock	`F0`	`F2`	`F4`	`F6`	`F8`	`F10`	`F12`	...	`F30`
61						`Divide`			

Register result status.

62. **Cycle 62**

✔ `DIVD` **writes result to** `F10`.

- All instructions have now:
  - Been **issued**
  - **Executed**
  - **Written back**

All functional units are idle, the pipeline is now **completely drained**.

Instruction	Issue	Read Op.	Exec Comp	Write Res
`LD F6, 34(R2)`	1	2	3	4
`LD F2, 45(R3)`	5	6	7	8
`MULTD F0, F2, F4`	6	9	19	20
`SUBD F8, F6, F2`	7	9	11	12
`DIVD F10, F0, F6`	8	21	61	62
`ADDD F6, F8, F2`	13	14	16	22

Instruction status.

Time	Name	`Busy`	`Op`	`Fi`	`Fj`	`Fk`	`Qj`	`Qk`	`Rj`	`Rk`
	`Integer`	No								
	`Mult1`	No								
	`Mult2`	No								
	`Add`	No								
	`Divide`	No								

Functional unit status.

Clock	`F0`	`F2`	`F4`	`F6`	`F8`	`F10`	`F12`	...	`F30`
62									

Register result status.

The 62-cycle examle is a demonstration of: efficient hazard management, robust scheduling logic, clean and scalable hardware coordination. The scoreboard shines in showing how **simple rules, carefully enforced, can deliver powerful out-of-order behavior** without complexity of modern speculative or superscalar techniques.

3.7 Tomasulo's Algorithm

3.7.1 Introduction

Tomasulo's Algorithm represents a pivotal innovation in the domain of dynamic scheduling and out-of-order execution within high-performance computing. Developed in 1967 at IBM for the IBM System/360 Model 91, it was introduced as a means to exploit **Instruction-Level Parallelism (ILP)** in the absence of compiler support or source-level reordering. The essential goal was to **overcome pipeline stalls** due to data hazards, particularly **Write After Write (WAW)** and **Write After Read (WAR)** hazards, both of which are challenging to resolve through simple pipeline control mechanisms.

Core Idea

Tomasulo's algorithm enables **instructions to execute out of program order**, yet maintains **data correctness** via hardware-level mechanisms. Central to this approach is the concept of **implicit register renaming**, which dynamically assigns storage locations (reservation stations) to values rather than using architectural register names directly. This mechanisms ensures that no two instructions mistakenly read or overwrite the same register unless there is a true data dependency (RAW, Read After Write).

New features introduced: a kind of Dynamic Scheduling 2.0

- **Implicit Register Renaming**: avoids WAR and WAW hazards by assigning intermediate results to reservation stations rather than architecture registers.
- **Dynamic Scheduling**: unlike static instruction scheduling (done at compile time), Tomasulo's algorithm uses runtime analysis to decide the order of instruction execution (yes, like the Scoreboard algorithm, but smarter).
- **Out-of-Order Execution**: instructions can be issued and begin execution as soon as operands are ready, independently of program order, provided that dependencies are resolved.
- **Common Data Bus (CDB)**: results are broadcast to all units waiting for them, further enabling parallelism.

Historical Significance

Tomasulo's work appeared just three years after the CDC 6600 Scoreboarding mechanism (Seymour Cray's design), which was the first dynamic scheduling mechanisms. Unlike the scoreboard, Tomasulo's algorithm **distributes control** among the **functional units** via **reservation stations**, offering more scalable and parallel data communication through the CDB.

It served as the architectural blueprint for later processors such as:

- Alpha 21264
- HP PA-8000
- MIPS R10000
- Intel Pentium II
- PowerPC 604

⚖ Tomasulo vs. Scoreboarding: What's the difference?

Both **Tomasulo's Algorithm** and the **Scoreboard Algorithm** are techniques for **dynamic instruction scheduling**, meaning they allow instructions to be executed **out of program order** while still preserving correctness. But Tomasulo goes a step further in terms of efficiency and cleverness:

Feature	Scoreboard (CDC 6600)	Tomasulo (IBM 360/91)
Register Renaming	✖ **No**	✔ **Yes** (implicit via reservation stations)
WAR/WAW Hazards Handling	✖ **Needs to stall**	✔ **Avoided by renaming**
Data Communication	✔ **Writes to Register File**	✔ **Used CDB to forward directly**
Control	Centralized scoreboard	Distributed (each FU has its own RS)
Execution Start Condition	Wait for operand *registers*	Wait for operand *values* or *tags*

Table 16: Tomasulo vs. Scoreboarding.

3.7.2 Register Renaming: Static vs. Implicit

First of all, why Register Renaming?

One of the main challenges in pipelined and out-of-order execution is handling **false dependencies**. **False Dependencies** (page 70) occur when instructions appear to depend on each other **because they use the same register name**, but there is **no true data dependency** between them. These are name-related hazards, not value-related. There are two main types:

- **WAR (Write After Dependencies)**: a later instruction writes to a register that a previous instruction needs to read.
- **WAW (Write After Write)**: two instructions write to the same register, but the second one is issued before the first finishes.

These are not *true* data dependencies (like Read After Write, RAW), but **name conflicts**, where different values want to use the same register name.

Register renaming solves this by dynamically or statically mapping registers to different physical storage locations.

Static Register Renaming (Compiler-Based)

In **Static Renaming**, the **compiler performs the renaming** at compile time, allocating **temporary** (non-architectural) **registers** to break WAR and WAW dependencies.

For example:

```
DIV.D   F0, F2, F4
ADD.D   F6, F0, F8       ; RAW on F0
S.D     F6, 0(R1)        ; RAW on F6
MUL.D   F6, F10, F8      ; WAW & WAR on F6
```

There is a **WAW hazard** (both `ADD.D` and `MUL.D` write to `F6`), and a **WAR hazard** (`S.D` reads `F6` while `MUL.D` wants to write it). With static register renaming, the compiler assigns a new register (e.g., `S`) to avoid the conflict:

```
DIV.D   F0, F2, F4
ADD.D   S, F0, F8        ; RAW still valid
S.D     S, 0(R1)         ; Now reads S
MUL.D   F6, F10, F8      ; Safe to use F6
```

Now the hazards are gone: `ADD.D` writes to `S`, which is consumed by the store; `MUL.D` writes to `F6` independently.

Static Register Renaming - Limitation. Static Renaming requires **predicting all hazards** in advance and knowing the **full execution path** (hard with branches, loops, dynamic inputs, etc.). Also, it requires **many more architectural registers** to be encoded in the ISA, not always feasible.

Implicit Register Renaming (Hardware-Based - Tomasulo's way)

Tomasulo's algorithm takes a smarter **dynamic** approach. Instead of using physical register names, it uses **Reservation Stations (RS)** as **Temporary Names (tags)** for operands. This renaming is done **implicitly** by the hardware at runtime.

Using the previous example:

```
DIV.D   F0, F2, F4
ADD.D   F6, F0, F8      ; RAW on F0
S.D     F6, 0(R1)       ; RAW on F6
MUL.D   F6, F10, F8     ; WAW & WAR on F6
```

Tomasulo rewrites it with **RS identifiers**:

```
DIV.D   F0, F2, F4
ADD.D   RS1, F0, F8       ; ADD result goes to RS1
S.D     RS1, 0(R1)        ; Store reads from RS1
MUL.D   F6, F10, F8       ; Now safe, F6 is available
```

- `ADD.D` **doesn't write to** `F6`, but to a *reservation station* named `RS1`.
- `S.D` **reads from** `RS1`, not from `F6`.
- This avoids **WAW (two writes to** `F6`**)** and **WAR (store reads** `F6` **before** `MUL.D` **writes)**.

Tomasulo's algorithm **automatically tracks which RS (Reservation Stations) produces what** and **only writes to the actual register file once the instruction retires**. Until then, everything is handled with RS tags.

This is the **core strength** of Tomasulo over simpler approaches like Scoreboarding or static renaming. It allows **aggressive out-of-order execution** without risking data hazards, making it fundamental in modern CPU design.

Here we have only presented the *secret sauce* of Tomasulo's power, in future sections we will gradually reveal how the tag-based mechanism works in practice.

3.7.3 Basic Concepts of Tomasulo's Algorithm

Goals of Tomasulo's Design

Tomasulo's algorithm was designed to solve a major performance bottleneck in pipelined processors: **pipeline stalls caused by operand unavailability** due to data hazards. The solution? Introduce a distributed, smart scheduling mechanism that:

- Avoids **WAR and WAW** hazards (false dependencies)
- Allows **out-of-order execution**
- Enables **register renaming implicitly**
- Uses **Reservation Stations (RSs)** and a **Common Data Bus (CDB)**

Reservation Stations (RSs): Tomasulo's Brain

Rather than having a central scoreboard (as in CDC 6600), Tomasulo distributes the **control logic and buffering close to the Functional Units (FUs)** using **Reservation Stations**.

Each functional unit (like a floating-point adder or multiplier) **has its own RSs** in front of it. These are **small buffers** that:

- **Hold instruction operands** (or *tags* pointing to where the operand will come from).
- **Wait until operands are ready**.
- **Dispatch instructions into the FU** as soon as everything is available.

This local storage of operands **removes the need to stall** the entire pipeline, **each unit become self-scheduling**.

Implicit Register Renaming with RS Tags

Instead of keeping track of operand names (e.g., `F2`, `F4`, `F6`, etc.), **Tomasulo tracks**:

- Either the **value** of the operand (if available)
- Or the **tag** of the RS that will produce that value (if not yet ready)

This is very powerful because:

- ✔ Registers are replaced by **RS names or actual values**
- ✔ WAR and WAW hazards are **completely avoided**
- ✔ Instruction scheduling becomes **data-driven**

We no longer wait for registers, we **wait for values**, and when they're ready, we go.

Tags, RSs, and the CDB

A critical component that ties everything together is the **Common Data Bus (CDB)**:

1. When a **functional unit finishes** execution, the result is **broadcast on the CDB**.
2. Any **RS waiting for that result's tag** will grab the value and store it in its local buffer.
3. Also, the result is written to the **Register File**, **but only if no newer instruction is overwriting that register**.

This broadcasting mechanism allows Tomasulo to perform a kind of **hardware-level forwarding**, operands are handed off *before* they hit the register file.

Feature	Scoreboarding	Tomasulo
Operand Waiting	Wait for **register**	Wait for **value** or **tag**
Operand Tracking	Centralized	**Distributed** in RSs
WAR/WAW hazards	Cause stalls	**Avoided via renaming**
Communication	Implicit write-back	**Broadcast over CDB**
Renaming	✗ **None**	✓ **Implicit via RS**

Table 17: Compared to Scoreboarding.

Tomasulo replaces rigid, centralized scheduling with a **fluid, decentralized approach**. Reservation Stations **track availability**, **rename registers**, and **drive execution**. The Common Data Bus **broadcasts results** to all who need them.

The processor becomes **dataflow-like**: instructions execute *when their operands are ready*, not when some global scheduler says so.

3.7.4 Architecture

The classic architecture includes:

- An **Instruction Queue** that feeds instructions in program order.
- Several Reservation Stations (RSs) sitting in front of Functional Units (FUs) (like `ADD`, `MUL`, `DIV` units).
- A **Register File** that doesn't just store values, but also tracks **where the next value will come from**.
- A **Common Data Bus (CDB)** that broadcasts results to all RSs and the Register File.

Components of a Reservation Station (RS)

Each **Reservation Station (RS)** is a mini-instruction buffer attached to a specific type of Functional Unit. It holds **1 instruction** and includes the following fields:

Field	Description
`Tag` (name)	Unique ID of the RS (e.g., `RS1`, `RS2`, ...)
`Busy`	A boolean flag: *is the RS holding an instruction?*
`OP`	Operation type (e.g., `ADD.D`, `MUL.D`)
`Vj, Vk`	Value of operands (if available)
`Qj, Qk`	Tag of RS producing `Vj`, `Vk` (if not available)

- If `Vj`/`Vk` are valid → operands are ready.
- If `Qj`/`Qk` are nonzero → wait for the result tagged with that RS name to arrive on the CDB.

Only **one of (`V`, `Q`)** is valid for each operand at any given time. Finally, note that for memory instructions (like Load/Store), `Vj` often holds the **address**, not a register value.

Register File and Store Buffers

The Register File in Tomasulo's architecture does **more than just store values**. Each register contains:

- `Vi`, current value (if available)
- `Qi`, tag of the RS that will produce this value (if pending)

All depends on the `Qi` value:

- If `Qi = 0` → the value is already available → use `Vi`.
- If `Qi` $\neq$ `0` → the register is waiting for an instruction to produce that value.

This tagging makes the register file an elegant interface for:

- **Implicit register renaming**
- **Hazard tracking**
- **Register freeing**

Likewise, **Store Buffers** behave like a special type of RS:

- They **wait** for both **address** and **data** to be ready **before sending it to memory**
- Both elements may depend on other RS outputs, so Store Buffers also monitor the CDB.

Load/Store Buffers

Memory operations need special handling due to:

- Address calculation
- Unknown data dependencies
- Potential memory hazards

So we use:

- **Load Buffers**, consisting of two fields: `Busy`, `Address`. Calculated in two steps:
  1. Address calculation using `base register + offset`.
  2. Wait for memory unit availability to load the data.
- **Store Buffers**, wait for the **data to be stored** and the **address**. Once both are ready, send to memory.

Store Buffers act like RSs, they hold partial information and wait on operand values (tags), same as `ADD` or `MUL` units.

Why this architectural design works?

- **Decouple instruction issue from operand readiness**, then out-of-order execution becomes natural.
- **Multiple RSs per FU**, then enables multiple instructions to be in-flight for the same unit type.
- **Register file doesn't need to "remember everything"**, then RSs temporarily hold data and manage dependencies.
- **Broadcast via CDB**, then all waiting units instantly get what they need, without central control.

In other words, the Tomasulo's architecture is a **beautifully decentralized, tag-driven pipeline**:

- Each **Reservation Station** acts as a micro-scheduler for its FU.
- The **Register File** manages *what is available* vs. *what is still pending*.
- **Load/Store Buffers** elegantly handle memory-side hazards and operand waits.
- The **CDB** ties it all together, broadcasting results as soon as they're ready.

This structure sets the stage for the dynamic instruction lifecycle, which we'll explore in detail in the next section.

Example of Tomasulo structure

The following figure shows an example of RISC-V structure using Tomasulo. It is a very good example to understand how the architecture works.

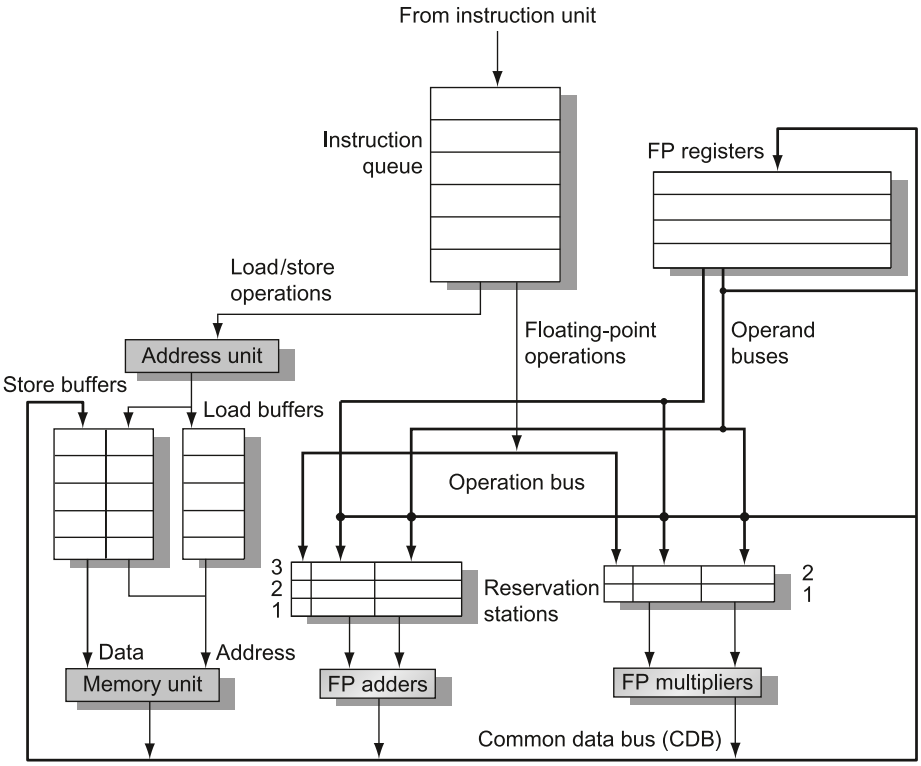


Figure 26: The Basic Structure of a Tomasulo-Style FPU. [2] It illustrates the **hardware datapaht and control structure** used in a **RISC-V Floating-Point Unit (FPU)** that supports **dynamic scheduling via Tomasulo's algorithm**. This architecture shows **how a processor executes FP instructions out of order**, tracks dependencies using **tags**, and delivers results using the **Common Data Bus (CDB)**.

- **Instruction Queue**. Holds instructions waiting to be issued. The instructions are issued in-order (First In, First Out) to **Reservation Stations**.
- **Reservation Stations (RSs)**. Act as **temporary instruction buffers** for each type of FU (e.g., `FP ADD`, `MULT`, `DIV`). Each RS holds:
  - Operation Type (e.g., `ADD.D`, `MUL.D`)
  - Operand values (`Vj`/`Vk`) or tags (`Qj`/`Qk`)
  - Status bits (busy, ready, etc.)

  Waits until **both operands are ready** and the FU is free, then issues to the FU.

- **Function Units (FUs)**. Includes separate **pipelined units** for `ADD`, `MULT`, and `DIV`. Each can execute one instruction at a time. Upon completion, sends result to the **CDB**.
- **Register File (RF)**. Stores FP register values (`F0-F31`). Each register has: **Value field (Vi)**, and **Tag field (Qi)** that is the ID of the RS that will produce its value. When reading a register:
  - If `Qi = 0`, value is ready → use `Vi`.
  - If `Qi` $\neq$ `0`, value is pending → use tag to track it.
- **Common Data Bus (CDB)**. Broadcasts **completed results** (value and tag) to: **all waiting RSs** (those with `Qj`/`Qk` matching the tag), **Register File** (if that tag matches a register **Qi**). Enables **hardware forwarding**: results don't need to wait for write-back to the register file.
- **Load/Store Queue** (optional). In a full Tomasulo implementation, memory operations are handled by **Load and Store buffers**. This diagram is focused on **FP register-register instructions**, so memory is abstracted.

Execution Flow

1. **Issue (Instruction Queue → RS)**
   (a) Fetch next **Floating Point (FP)** instruction.
   (b) If a Reservation Station (RS) for the operation type is available:
      i. Copy instruction into RS
      ii. Fetch operands from Register File (RF)
         - If ready → store value in `Vj/Vk`
         - If not → store producer RS tag in `Qj/Qk`
      iii. Update the destination register's `Qi` with this RS tag (register renaming).
2. **Execute (RS → FU)**
   (a) Once operands are ready (`Qj = Qk = 0`) and FU is free:
      i. RS issues to its Functional Unit (FU)
      ii. FU computes the result (may take multiple cycles)
      iii. Execution is **out of order**, instructions are scheduled based on readiness, not program order.
3. **Write Result (FU → CDB)**
   (a) When the FU finishes:
      i. Sends the result with **its tag** on the **CDB**.
      ii. All RSs waiting on that tag update their operand fields.
      iii. Register file updates any registers whose `Qi` matches the tag.
      iv. The RS becomes free again.

3.7.5 Stages

Tomasulo's execution model is based on three main pipeline stages:

1. **Issue** (**in-order**): decode instruction, send to RS.
2. **Execute** (**out-of-order**): wait for operands, perform operation.
3. **Write Result** (**out-of-order**): broadcast result via CDB to RF and RSs.

Each instruction flows through these three stages, but thanks to the algorithm's design, they don't have to do so in program order after issuing.

3.7.5.1 Stage 1: Issue

The **Issue Stage** is the first step in Tomasulo's 3-phase pipeline. In this stage, instructions are **decoded and dispatched** from the **Instruction Queue (IQ)** into **Reservation Stations (RSs)** (see Figure 26, page 143), where they wait until operands and functional units are ready.

Main Responsibilities of the Issue Stage

1. **Instruction decoding**. Identify the opcode and register operands.
2. **Reservation Station assignment**. Check for an available RS of the appropriate type (e.g., an `ADD.D` goes to an RS attached to the `FP ADD` unit).
3. **Operand availability check**. For each source register:
   - ✔ If **ready**, read its **value** → store in `Vj`/`Vk`.
   - ✘ If **not ready**, read its **tag** → store in `Qj`/`Qk` (wait for it!).
4. **Destination register renaming**. Update the **register file's tag field** (`Qi`). It now points to the RS handling this instruction. This is **implicit register renaming**.

Why is this stage in-order?

Tomasulo always issues instructions **in program order**, one at a time. Why? Three reasons:

- Ensures **precise exceptions**, it helps the machine know which instruction caused a fault.
- Keeps memory operations (like stores) in **correct order**.
- **Prevents complexity** from instruction reordering too early.

Out-of-Order behavior starts **after issue**, in the **Execute** stage.

What can cause a stall during Issue?

1. **Structural Hazard**: no free Reservation Station.
2. **Load/Store queue full**: for memory operations.
3. (in more advanced versions) **Pending branch**: speculative instructions must wait.

Why this Stage is powerful

- Begins **implicit renaming**, eliminating **WAR and WAW** hazards.
- Allows instructions to **wait for operands without blocking the pipeline**.
- Decouples **instruction arrival** from **execution readiness**.

Example 7: Issue stage

Say the instruction is:

```
ADD.D F6, F2, F4
```

1. Tomasulo checks: "*Is there a free `ADD` reservation station?*"
   - ✗ No, insert a **stall**: avoid **structural hazard**.
   - ✓ If there are available reservation stations, then checks:
     (a) Is `F2` ready?
     - ✓ Yes → `Vj` = value of `F2`
     - ✗ No → `Qj` = tag of RS producing `F2`

     (b) Is `F4` ready?
     - ✓ Yes → `Vk` = value of `F4`
     - ✗ No → `Qk` = tag of RS producing `F4`
2. Finally, it updates: `F6.Qi = RS1` (renaming `F6` to `RS1`). This means: "*`F6` is not ready yet, it will be produced by `RS1`*".

From this point on, **any instruction that needs `F6` will wait for `RS1`**, not for `F6` directly.
For example, a second instruction appears:

```
MUL.D F8, F6, F10
```

Tomasulo checks: "*what's the status of `F6`?*". It finds:

`F6.Qi = RS1`

`F6` does not contain a value yet, but its value will come from `RS1`. So now, in the RS for the `MUL` instruction (say `RS2`), we set:

- `Qj = RS1` (because `F6` is not ready, and its result will come from `RS1`).
- `Vk` = value of `F10` (assuming `F10` is ready).

This is the magic: `RS2` knows that it must **listen for `RS1`'s result** on the Common Data Bus (CDB).

Quick Recap

1. Decode instruction
2. Allocate Reservation Station (RS)
3. Read operand values or tags
4. Rename destination register via tag (`Qi`)
5. If no RS → stall

This stage is like **preparing an instruction for its flight**, loading its bags, assigning it a gate, and scheduling it to depart; but waiting for the runway (operands and Functional Unit) to be clear.

3.7.5.2 Stage 2: Start Execution

This is the phase where the instruction **actually performs the computation** in the appropriate **Functional Unit (FU)**, like `ADD`, `MULT`, or `DIV`. But, and this is key, it can only happen **when all operand values are ready** and **the Functional Unit (FU) is available**.

So, execution isn't just about doing the math, it's also about **waiting for the right moment**.

Preconditions: When can an Instruction start executing?

To start execution, three things must happen:

1. **Both operands are available**. In the Reservation Station (RS), this means:

   `Qj = Qk = 0`

   No tags, just valid values in `Vj` and `Vk`.

2. **The Functional Unit (FU) is free**, e.g., the `FP ADD` unit is idle.

3. (optional) **Instruction dependencies are resolved**, e.g., no pending branches or memory issues ahead.

Only **when all conditions are met**, the instruction *leaves* the Reservation Stations (RS) and enters the **Functional Unit** (FU).

What Happens in the Execute Stage?

Once **instructions are in Reservation Stations (RSs)** and waiting for operands, the **Execute Stage begins**. This is where:

- The instruction is **dispatched to the Functional Unit (FU)**.
- It **executes for its latency**.
- The result is **held**, ready to be broadcast in Stage 3.

But unlike a classic pipeline, **Tomasulo waits for readiness**, not for a fixed cycle count.

Let's break it into key sub-steps:

1. **Operand Readiness Check**. Each Reservation Station (RS) monitors:

   - `Qj`
   - `Qk`

   If both are zero, it means that the **values `Vj` and `Vk` are ready**, and no other Reservation Station needs to broadcast them.

   In other words, when both operands are ready, the instruction can **start executing**.

2. **Check FU Availability**. Tomasulo now checks if the **relevant Functional Unit** (e.g., `FP ADD`, `FP MUL`, `FP DIV`) is **free**.

   ✅ If **yes** → dispatch instruction from RS to FU.

   ❌ If **no** → **stall inside the RS** until FU becomes free.

   Multiple Reservation Stations might be ready, but **only one can use a given Functional Unit at a time**. A scheduler chooses which RS gets the FU (usually oldest-first).

3. **Instruction Execution (Latency Counts)**. Once inside the Functional Unit (FU):

   - The instruction performs its **actual arithmetic operation**.
   - This can take multiple cycles depending on the instruction type.

Instruction	Latency (example)
`ADD.D`	2-3 cycles
`MUL.D`	4-7 cycles
`DIV.D`	10+ cycles

   This time is **internal to the FU**, the RS has already "let go" of the instruction. The RS is now **free** and can be reused after the instruction completes Stage 3 (write-back).

4. **Memory Instructions (LOAD/STORE) - Special Case**. Unlike register-based instructions (e.g. `ADD.D F6, F2, F4`), memory instructions **read or write actual memory**, and memory access has some *dangerous side effects*:

   - It's **global**: memory affects all parts of the program.
   - It must be **precise**: no reordering can change program behavior.
   - It can **raise exceptions** (e.g., accessing invalid addresses).

   So Tomasulo **handles LOAD and STORE more carefully** than other instructions.

   - **LOAD Instruction** `L.D F4, 0(R1)`. Load the value from memory address `R1 + 0` into register `F4`. What happens:
     (a) **Wait for base register** (`R1`) to be ready.
       – If `R1` is not ready, the Load Buffer stores a **tag**, just like in RSs.
       – Once ready, calculate **effective address** (e.g., 1000).
     (b) **Wait for access to memory**. If there's no memory conflict, proceed to load.
     (c) **Stage 3 (write)**. The value is broadcast on the **CDB** to: waiting RSs (if any), register `F4` (if still tagged).

LOADs are usually allowed to execute out-of-order (relative to other LOADs), *as long as we're sure no STORE before them is writing to the same address.*

- **STORE Instruction** `S.D F4, 0(R1)`. Store the value in `F4` to memory address `R1 + 0`. This is trickier, because:
  - STORE instructions do **not write results to a register**.
  - They write **into memory**, which is a **shared global state**.
  - If we mess up the ordering, we can cause **wrong program results**.

  What happens:

  (a) **Wait for both**:
    i. The **data to be stored** (`F4`) → may need to track a tag.
    ii. The **address to store to** (`R1 + 0`) → also may depend on a tag.

  (b) **Wait for memory access permission**:
    - Tomasulo ensures **store ordering** is preserved.
    - This means **STOREs happen in program order**.
    - We cannot let a later STORE "jump ahead" of an earlier one.

  (c) **When both data and address are ready** → write to memory (not via CDB).

  STORE instructions do not write anything on the CDB, because they don't produce a result for future instructions, they only update memory.

⚖ Classical Pipeline vs. Tomasulo

Classical Pipeline	Tomasulo
Fixed issue → execute	Waits dynamically for operands
Register-based data flow	Tag-based tracking via RSs
RAW hazards stall issue	RAW stalls **only** inside RSs
WAR/WAW hazards possible	**Eliminated** via register renaming

Table 18: Classical Pipeline vs. Tomasulo.

This is **dataflow execution**: instructions run **as soon as data is ready**, not when the program says so.

↺ Quick Recap

1. Check if `Qj` and `Qk` are 0 (ready).
2. If yes and FU is free → dispatch.
3. Execute in FU (multi-cycle).
4. Prepare to broadcast result on CDB.

3.7.5.3 Stage 3: Write Result

This is the **moment the instruction finishes its computation** in the Functional Unit (FU), and the result is made **globally visible** by broadcasting it over the **Common Data Bus (CDB)**. It's the stage where:

- The **value computed by the FU** is sent to:
  - All **Reservation Stations (RSs)** waiting for it;
  - The **Register File (RF)** (if still tagged with RS).
- The instruction is now officially "**done**".

What happens step-by-step

Let's say we have this instruction in Reservation Station 3 (`RS3`):

```
1 MUL.D F6, F2, F4
```

`RS3` starts execution in the `MUL` unit, takes several cycles, and finally finishes. Now we begin:

1. **Broadcast on CDB**. The Functional Unit (FU) places the result on the **Common Data Bus (CDB) along with its tag** (`RS3`):

   $$\texttt{CDB} \leftarrow \langle \texttt{tag} = \texttt{RS3}, \texttt{value} = \texttt{result} \rangle$$

   This is the Tomasulo version of a "public announcement": "*`RS3` has finished computing a result. Anyone waiting for this, come get it!*".

2. **Reservation Stations listen for Tags**. Every Reservation Station (RS) in the system checks: "*does my `Qj` or `Qk` match the tag on the CDB?*". If yes:

   - The Reservation Station (RS) grabs the **value**.
   - Stores it in `Vj` or `Vk`.
   - Clears the tag:

     $$\texttt{Qj} \leftarrow 0 \quad \vee \quad \texttt{Qk} \leftarrow 0$$

     Meaning: operand is now ready.

   This process allows many instructions to **simultaneously wake up** when the result they needed finally becomes available.

3. **Register File Update**. The **Register File** checks: "*is there any register whose `Qi = RS3`?*". If yes:

   - It writes the result into that register.
   - Clears the tag field (`Qi = 0`), this means "`F6` is now valid and available".

   If the register has already been renamed again (e.g., `Qi = RS4`), we **don't write** to it, this avoid **WAW hazards**.

4. **Free the RS and FU**. The instruction in `RS3` is now complete, `RS3` is marked free, and the Functional Unit (FU) becomes available again for other instructions.

Example 8: why don't we write to a register when it has been renamed?

Suppose we have two instructions writing to the **same register**:

```
1 ADD.D  F6, F2, F4    ; instruction A
2 MUL.D  F6, F6, F8    ; instruction B
```

Instruction 1 is issued first, but instruction 2 also **writes to** `F6` (again). This is a **Write After Write (WAW)** situation.

1. Instruction 1 (`ADD.D`) is issued → assigned to `RS1`
   - Tomasulo sets: `F6.Qi` ← `RS1` ("*F6 will come from RS1*").
2. Instruction 2 (`MUL.D`) is issued **immediately after** → assigned to `RS2`
   - Tomasulo updates again: `F6.Qi` ← `RS2` ("*F6 will now come from RS2*").

Now we have:

- Two instructions writing to `F6`.
- But **only the result of `RS2` should actually end up in `F6`**.
- **`RS1`'s result is now obsolete**, because it was overwritten by a newer instruction.

When `RS1` finishes (WRITE STAGE), `RS1` broadcasts its result on the CDB:

$$\langle \texttt{Tag} = \texttt{RS1}, \texttt{Value} = \ldots \rangle$$

Tomasulo checks the Register File: "*does any register have `Qi = RS1`?*":

- ✅ If **yes**, update that register.
- ❌ If **no** (this case), `F6.Qi = RS2`, not `RS1` anymore.

So we skip the write. Because if we allowed `RS1` to write to `F6` now, it would **overwrite the result of `RS2`**, which hasn't been computed yet, and then we'd break correctness.
The is how **Tomasulo avoids WAW hazards**: only write to a register if it's still waiting for our result. If someone newer came along and renamed the register again, we're out. Our value is no longer needed in the register file.

- CDB allows **many instructions** to grab the result **at the same time**.
- Values are **forwarded before being stored** (no need to wait for register write-back).
- **False dependencies (WAR/WAW)** are avoided automatically.
- **RAW dependencies** are resolved **dynamically**, as instructions "listen" for the values they need.

3.7.6 In-Depth Execution Example

Example 9: Dependence and Hazard Analysis

We're given a sequence of floating-point instructions to be executed by Tomasulo's algorithm:

```
1 L.D F6, 34(R2)       ; F6  ← Mem[R2 + 34]
2 L.D F2, 45(R3)       ; F2  ← Mem[R3 + 45]
3 MUL.D F0, F2, F4     ; F0  ← F2 × F4
4 SUB.D F8, F6, F2     ; F8  ← F6 − F2
5 DIV.D F10, F0, F6    ; F10 ← F0 ÷ F6
6 ADD.D F6, F8, F2     ; F6  ← F8 + F2
```

We begin by identifying **data dependencies** (RAW, WAR, WAW) between these instructions. Let's go instruction by instruction and analyze dependencies:

- Instruction 1: `L.D F6, 34(R2)` → No dependencies, loads into `F6`.
- Instruction 2: `L.D F2, 45(R3)` → No dependencies, loads into `F2`.
- Instruction 3: `MUL.D F0, F2, F4`
  - **RAW** dependency on `F2`, produced by Instr. 2
  - `F4` is assumed ready (not in this instruction stream)
- Instruction 4: `SUB.D F8, F6, F2`
  - **RAW** dependency on `F6`, from Instr. 1
  - **RAW** dependency on `F2`, from Instr. 2
- Instruction 5: `DIV.D F10, F0, F6`
  - **RAW** dependency on `F0`, from Instr. 3
  - **RAW** dependency on `F6`, from Instr. 1 (but will be overwritten later)
- Instruction 6: `ADD.D F6, F8, F2`
  - **RAW** dependency on `F8`, from Instr. 4
  - **RAW** dependency on `F2`, from Instr. 2
  - **WAW** hazard: Instr. 1 wrote `F6`, this one **overwrites** it again.
  - **WAR** hazard: Instr. 5 reads `F6`, but this instruction **writes** it again.

Some key observations:

- Instr. 1 and 2 are **independent loads**, they will issue immediately.
- Instr. 3 (`MUL.D`) has to **wait for** `F2`, but `F4` is assumed ready.

- Instr. 4 and 5 have **multiple dependencies**, they will wait in Reservation Station (RS) until results are available.
- Instr. 6 creates **WAW and WAR** hazards on `F6`, this is where **Tomasulo's register renaming** shines:
  - The second write to `F6` gets a **new tag**.
  - The WAR/WAW conflicts are eliminated by tracking **RS tags instead of `F6` name**.

Example 10: Register Renaming

The goal of this section is to demonstrate how **register renaming eliminates false dependencies** (WAR and WAW) and allows **out-of-order execution** by tagging instructions with Reservation Station IDs instead of register names.

Let's revisit the original instruction sequence:

```
1 L.D F6, 34(R2)
2 L.D F2, 45(R3)
3 MUL.D F0, F2, F4
4 SUB.D F8, F6, F2
5 DIV.D F10, F0, F6
6 ADD.D F6, F8, F2
```

Instead of using `F6`, `F2`, etc. directly, Tomasulo assigns a Reservation Station (RS) to track **who will produce** each value. This Reservation Station (RS) **renames** the register in the hardware temporarily. This helps **avoid**:

- **WAW** hazard → only the most recent instruction writes to the final destination.
- **WAR** hazard → consumers of earlier values don't get overwritten by later writers.

Here's a sample of how registers and reservation stations look during execution:

Instruction	RS	Dest Reg.	`Qi` Field in RF	Renaming Effect
`L.D` → `F6`	`LB1`	`F6`	`F6.Qi` ← `LB1`	`F6` will be produced by `LB1`
`L.D` → `F2`	`LB2`	`F2`	`F2.Qi` ← `LB2`	`F2` will be produced by `LB2`
`MUL.D` → `F0`	`MUL1`	`F0`	`F0.Qi` ← `MUL1`	`F0` will be produced by `MUL1`
`SUB.D` → `F8`	`SUB1`	`F8`	`F8.Qi` ← `SUB1`	`F8` will be produced by `SUB1`
`DIV.D` → `F10`	`DIV1`	`F10`	`F10.Qi` ← `DIV1`	`F10` will be produced by `DIV1`
`ADD.D` → `F6`	`ADD1`	`F6`	`F6.Qi` ← `ADD1`	`F6` now renamed again by `ADD1`

Let's follow `F6` across the timeline:

1. Instruction 1: `F6` ← `L.D` ⇒ RF: `F6.Qi = LB1`

2. Instruction 5: `DIV.D` uses `F6` ⇒ waits on `LB1`'s result
3. Instruction 6: `F6` ← `ADD.D` ⇒ RF: `F6.Qi = ADD1`

So by the time `ADD.D` issues:

- **Register File (RF) no longer tracks** `LB1`, even though that result hasn't been written yet.
- `DIV.D` **is still listening for** `LB1`**'s result**.
- **WAR/WAW hazards are avoided** because consumers and writers track **RS tags**, not the physical register name `F6`.

Now consider propagating the result with tags. Let's say `MUL1` finishes first:

1. It broadcasts ⟨`Tag = MUL1`, `Value = (result)`⟩
2. The **Register File** updates:
   - If `Qi` = `MUL1` → write result into the register.
   - If `Qi` ≠ `MUL1` → skip write.
3. RSs waiting on `Qj = MUL1` or `Qk = MUL1`, grab the value.

Same logic applies for `LB1`, `ADD1`, etc. *Why Register Renaming Solves WAR and WAW?*

Hazard	Traditional Pipeline	Tomasulo's Solution
WAR	Later write may overwrite a value before it's read.	Each reader remembers **which tag** to wait for, not the register name.
WAW	Multiple instructions write to same register → wrong order.	Only the **last tag** (`Qi`) is honored; old ones are discarded.

Register renaming in Tomasulo means:

- Every instruction output gets a **temporary, unique name**, called the *Reservation Station tag*.
- The **Register File tracks tags**, not values, while results are still pending.
- When a **result is ready**:
  1. It's broadcast with its tag
  2. Listeners receive it
  3. Register File updated only **if still needed**.

This make execution:

- **Safe** (no false hazards)
- **Flexible** (out-of-order)
- **Clean** (decouples values from register names)

Now we bring Tomasulo to life cycle-by-cycle. We'll track each instruction through the **three stages**:

- Issue
- Start Execution (Execute Start)
- Write Result (Write)

The instructions are:

```
L.D F6, 34(R2)
L.D F2, 45(R3)
MUL.D F0, F2, F4
SUB.D F8, F6, F2
DIV.D F10, F0, F6
ADD.D F6, F8, F2
```

Why are SUBD and DIVD merged into ADDD and MULD units?

ADDD and SUBD are both **floating-point addition operations**, differing only in the operation:

- ADDD F6, F2, F4 $\rightarrow$ F6 = F2 + F4
- SUBD F6, F2, F4 $\rightarrow$ F6 = F2 - F4 = F2 + (-F4)

They both use the **same operand types**, have the **same input/output behavior** and can be performed by the **same kind of floating-point adder/-subtractor unit**. Therefore, a single ADD unit and its reservation stations can be reused for both operations. That's why in the RS table, SUBD uses an ADD-type RS (e.g., ADD1, ADD2).

Same for MULD and DIVD. These are both **multiplicative FP operations**. Architecturally, many processors use a **shared pipeline or RS pool** for:

- MULD (fastest latency)
- DIVD (slower latency but same operand structure)

So MULD and DIVD share MUL **reservation stations** and a **shared execution unit or Functional Unit slot**. That's why we see:

- MULD F0, F2, F4 issued to MUL1
- DIVD F10, F0, F6 is waiting in MUL2 for F0

They both use the **same RS pool**, because they're functionally similar in structure, and sharing resources saves hardware area. Also note that in floating-point arithmetic we have:

$$\mathtt{A} \div \mathtt{B} = \mathtt{A} \times \left(\frac{1}{\mathtt{B}}\right)$$

Instead of implementing a full-blown divider, many systems compute the **reciprocal of** B $\left(\frac{1}{B}\right)$ and then multiply:

```
DIV.D   F10, F0, F6      ; F10 ← F0 ÷ F6
≈
RECIP   F12, F6          ; F12 ←  1 ÷ F6
MULD    F10, F0, F12     ; F10 ← F0 × F12
```

This transformation is:

- **Valid numerically**, though it may lose precision.
- **Common in vector processors**, **GPUs**, and some scalar floating-point pipelines (e.g., ARM, GPUs).
- Used when latency and area are more important than perfect accuracy.

This allows the processor to **reuse the multiplier** (which is fast and compact) instead of having a separate, slow divider.

1. **Cycle 1**
   - `LD F6, 34(R2)` is issued.
     - `LoadBuffer1` allocated.
     - `F6.Qi` = `Load1`
   - Base address `R2` assumed ready → can start address calculation in next cycle.

Instruction	Issue	Start Execute	Write Result
`LD F6, 34(R2)`	1		
`LD F2, 45(R3)`			
`MULTD F0, F2, F4`			
`SUBD F8, F6, F2`			
`DIVD F10, F0, F6`			
`ADDD F6, F8, F2`			

Instruction status.

Name	Vj	Qj	Vk	Qk
Load1	34		v(R2)	
Load2				
EXLoad				

`EXLoad` (or `EX_LD`). Tracks the **status of the Load Buffer(LB)** and **Load execution units**.

Name	Vj	Qj	Vk	Qk
ADD1				
ADD2				
ExADD				

ExADD. Tracks the use of the **addition** arithmetic **Functional Unit** (FU).

Name	Vj	Qj	Vk	Qk
MUL1				
MUL2				
ExMUL				

ExMUL. Tracks the use of the **multiplication** arithmetic **Functional Unit** (FU).

RF	0	1	2	3	4	5	6	7	8	9	10	...	31
Qi							Load1						

Register Result Status. Shows the state of **register renaming** during execution. In other words, it shows the state of each floating-point register (F0-F31). It is a **snapshot of the Qi field** for each register in the floating-point register file.

2. **Cycle 2**
   - `LD F2, 45(R3)` is issued
     - `LoadBuffer2` allocated
     - `F2.Qi` = `Load2`
   - `LD F6` (Instr. 1) starts execution:
     - Address calculated (`R2 + 34`)
     - Memory load begins

Instruction	Issue	Start Execute	Write Result
`LD F6, 34(R2)`	1	2	
`LD F2, 45(R3)`	2		
`MULTD F0, F2, F4`			
`SUBD F8, F6, F2`			
`DIVD F10, F0, F6`			
`ADDD F6, F8, F2`			

Instruction status.

Name	Vj	Qj	Vk	Qk
`Load1`	34		`v(R2)`	
`Load2`	45		`v(R3)`	
`EXLoad`	34		`v(R2)`	

`EXLoad` (or `EX_LD`).

Name	Vj	Qj	Vk	Qk
`ADD1`				
`ADD2`				
`ExADD`				

`ExADD`.

Name	Vj	Qj	Vk	Qk
`MUL1`				
`MUL2`				
`ExMUL`				

`ExMUL`.

RF	0	1	2	3	4	5	6	7	8	9	10	...	31
`Qi`			`Load2`				`Load1`						

Register Result Status.

3. **Cycle 3**

- `MULTD F0, F2, F4` is issued
  - Reservation Station = `MUL1`
  - Needs:
    - `F2` → still pending from `Load2` → `Qj = Load2`
    - `F4` → assumed ready → `Vk = value = v(F4)`
  - `F0.Qi` = `MUL1`
- `LD F2` (Instr. 2) starts execution: Address calculated (`R3 + 45`); Memory load begins.
- Load (`LD F6`) has a delay and is still not finished. This is important because it causes **dependent instructions to wait**. However, this is the brilliance of Tomasulo's dataflow: these **instructions don't block the pipeline**, they just **wait for the tag (`Load1`) to appear in the CDB**.

Instruction	Issue	Start Execute	Write Result
`LD F6, 34(R2)`	1	2	
`LD F2, 45(R3)`	2		
`MULTD F0, F2, F4`	3		
`SUBD F8, F6, F2`			
`DIVD F10, F0, F6`			
`ADDD F6, F8, F2`			

Instruction status.

Name	Vj	Qj	Vk	Qk
`Load1`	34		`v(R2)`	
`Load2`	45		`v(R3)`	
`EXLoad`	34		`v(R2)`	

`EXLoad` (or `EX_LD`).

Name	Vj	Qj	Vk	Qk
`ADD1`				
`ADD2`				
`ExADD`				

`ExADD`.

Name	Vj	Qj	Vk	Qk
`MUL1`		`Load2`	`v(F4)`	
`MUL2`				
`ExMUL`				

`ExMUL`.

RF	0	1	2	3	4	5	6	7	8	9	10	...	31
Qi	MUL1		Load2				Load1						

Register Result Status.

4. **Cycle 4**

   - `SUBD F8, F6, F2` is issued
     - Reservation Station = `SUB1`
     - Needs:
       * `F6` → pending from `Load1` → `Qj = Load1`
         But the RS Load listens to the CDB for the `Load1` tag. It sees the result of the CDB broadcast and overwrites its state:

         $$\Rightarrow \texttt{Qj = 0} \qquad \texttt{Vj = v(F6)}$$

       * `F2` → pending from `Load2` → `Qk = Load2`
     - `F8.Qi` = `SUB1`
   - No execution start yet for `MUL` or `SUB`, still waiting on operands. BUT, the result of loading the first instruction (value from memory) is sent on the Common Data Bus (CDB). And the `SUB` instruction has grabbed it and is now ready to start executing.

   Also, in the register result status table, we insert the value of `F6` (`v(F6)`) instead of leaving the `Load1` tag. This is because the CDB sends the result to all listeners.

Instruction	Issue	Start Execute	Write Result
`LD F6, 34(R2)`	1	2	4
`LD F2, 45(R3)`	2		
`MULTD F0, F2, F4`	3		
`SUBD F8, F6, F2`	4		
`DIVD F10, F0, F6`			
`ADDD F6, F8, F2`			

Instruction status.

Name	Vj	Qj	Vk	Qk
`Load1`	34		`v(R2)`	
`Load2`	45		`v(R3)`	
`EXLoad`	34		`v(R2)`	

`EXLoad` (or `EX_LD`).

Name	Vj	Qj	Vk	Qk
ADD1	v(F6)	~~Load1~~ 0		Load2
ADD2				
ExADD				

ExADD.

Name	Vj	Qj	Vk	Qk
MUL1		Load2	v(F4)	
MUL2				
ExMUL				

ExMUL.

RF	0	1	2	3	4	5	6	7	8	9	10	...	31
Qi	MUL1		Load2				v(F6)		ADD1				

Register Result Status.

5. **Cycle 5**

- `DIVD F10, F0, F6` is issued
  - Reservation Station = `DIV1`
  - Needs:
    - `F0` → pending from `MUL1` → `Qj = MUL1`
    - `F6` is available thanks to the CDB
  - `F10.Qi` = `DIV1`
- `LD F2` (Instr. 2) starts execution:
  - Addressed calculated (`R3 + 45`)
  - Memory load begins

Instruction	Issue	Start Execute	Write Result
`LD F6, 34(R2)`	1	2	4
`LD F2, 45(R3)`	2	5	
`MULTD F0, F2, F4`	3		
`SUBD F8, F6, F2`	4		
`DIVD F10, F0, F6`	5		
`ADDD F6, F8, F2`			

Instruction status.

Name	Vj	Qj	Vk	Qk
`Load1`				
`Load2`	45		`v(R3)`	
`EXLoad`	45		`v(R3)`	

`EXLoad` (or `EX_LD`).

Name	Vj	Qj	Vk	Qk
`ADD1`	`v(F6)`			`Load2`
`ADD2`				
`ExADD`				

`ExADD`.

Name	Vj	Qj	Vk	Qk
`MUL1`		`Load2`	`v(F4)`	
`MUL2`		`MUL1`	`v(F6)`	
`ExMUL`				

`ExMUL`.

RF	0	1	2	3	4	5	6	7	8	9	10	...	31
`Qi`	`MUL1`		`Load2`				`v(F6)`		`ADD1`		`MUL2`		

Register Result Status.

6. **Cycle 6**

- `ADDD F6, F8, F2` is issued
  - Reservation Station = `ADD2`
  - Needs:
    - `F8` → pending from `ADD1` (`SUB1`) → `Qj = ADD1`
    - `F2` → pending from `Load2` → `Qk = Load2`
  - `F6.Qi` = ~~`Load1`~~ `ADD2`
- **WAR on `F6` has been eliminated**:
  - Tomasulo **renames the destination register** of the `ADDD` instruction: `F6` will now be written by `ADD2`, not by `Load1` (cycle 1).
  - But crucially, `DIVD` was issued **earlier** (previous cycle); it already read `F6`'s value from the Reservation Station of the `Load1`. It stored that value in `Vk` inside its RS (tag `MUL2`).
  - So even though `ADDD` will **overwrite `F6` later**, it doesn't matter: `DIVD` already took what it needed and moved on.

  The same rule is applied to `SUBD`.

Instruction	Issue	Start	Execute	Write Result
`LD F6, 34(R2)`	1	2		4
`LD F2, 45(R3)`	2	5		
`MULTD F0, F2, F4`	3			
`SUBD F8, F6, F2`	4			
`DIVD F10, F0, F6`	5			
`ADDD F6, F8, F2`	6			

Instruction status.

Name	Vj	Qj	Vk	Qk
`Load1`				
`Load2`	45		`v(R3)`	
`EXLoad`	45		`v(R3)`	

`EXLoad` (or `EX_LD`).

Name	Vj	Qj	Vk	Qk
ADD1	v(F6)			Load2
ADD2		ADD1		Load2
ExADD				

ExADD.

Name	Vj	Qj	Vk	Qk
MUL1		Load2	v(F4)	
MUL2		MUL1	v(F6)	
ExMUL				

ExMUL.

RF	0	1	2	3	4	5	6	7	8	9	10	...	31
Qi	MUL1		Load2				ADD2		ADD1		MUL2		

Register Result Status.

7. **Cycle 7**

- `LD F2` **completes execution**. It **broadcasts its result** (value of `F2`) via the Common Data Bus (CDB). **Forwarding** takes place to both:
  - **Register File**: if `Qi = Load2`, then `F2 = value` ⇒ `Qi = 0`
  - **Reservation Stations**:
    - `MULD` (`MUL1`): `Qj = Load2` → now `Vj = v(F2)` and `Qj = 0`
    - `SUBD` (`ADD1`): `Qk = Load2` → now `Vk = v(F2)` and `Qk = 0`
    - `ADDD` (`ADD2`): `Qk = Load2` → now `Vk = v(F2)` and `Qk = 0`

✗ `DIVD F10, F0, F6` still waiting. Needs `F0`, which is being computed by `MULD` (`Tag = MUL1`). Can't execute yet.

✗ `ADDD F6, F8, F2` still waiting. Needs `F0`, but still waiting on `F8`, `tag = ADD1` (result of `SUBD`).

Instruction	Issue	Start Execute	Write Result
`LD F6, 34(R2)`	1	2	4
`LD F2, 45(R3)`	2	5	7
`MULTD F0, F2, F4`	3		
`SUBD F8, F6, F2`	4		
`DIVD F10, F0, F6`	5		
`ADDD F6, F8, F2`	6		

Instruction status.

Name	Vj	Qj	Vk	Qk
Load1				
Load2	45		v(R3)	
EXLoad	45		v(R3)	

`EXLoad` (or `EX_LD`).

Name	Vj	Qj	Vk	Qk
ADD1	v(F6)		v(F2)	
ADD2		ADD1	v(F2)	
ExADD				

`ExADD`.

Name	Vj	Qj	Vk	Qk
MUL1	v(F2)		v(F4)	
MUL2		MUL1	v(F6)	
ExMUL				

ExMUL.

RF	0	1	2	3	4	5	6	7	8	9	10	...	31
Qi	MUL1		v(F2)				ADD2		ADD1		MUL2		

Register Result Status.

8. **Cycle 8**

- `MULTD F0, F2, F4` **starts execution** in `MUL1`:
  - It was waiting on `F2` (now received via CDB) and `F4` was already ready
  - Now: `Qj = 0`, `Qk = 0` → ✔ ready to execute
  - Executing in `MUL1` Functional Unit
- `SUBD F8, F6, F2` **starts execution** in `ADD1`:
  - `F6` was already ready from Cycle 4 (CDB)
  - `F2` now arrives, both operands are ready
  - Executing in `ADD1` Functional Unit

Instruction	Issue	Start Execute	Write Result
`LD F6, 34(R2)`	1	2	4
`LD F2, 45(R3)`	2	5	7
`MULTD F0, F2, F4`	3	8	
`SUBD F8, F6, F2`	4	8	
`DIVD F10, F0, F6`	5		
`ADDD F6, F8, F2`	6		

Instruction status.

Name	Vj	Qj	Vk	Qk
Load1				
Load2				
EXLoad				

`EXLoad` (or `EX_LD`).

Name	Vj	Qj	Vk	Qk
ADD1	v(F6)		v(F2)	
ADD2		ADD1	v(F2)	
ExADD	v(F6)		v(F2)	

`ExADD`.

Name	Vj	Qj	Vk	Qk
MUL1	v(F2)		v(F4)	
MUL2		MUL1	v(F6)	
ExMUL	v(F2)		v(F4)	

`ExMUL`.

RF	0	1	2	3	4	5	6	7	8	9	10	...	31
Qi	MUL1		v(F2)				ADD2		ADD1		MUL2		

Register Result Status.

10. **Cycle 10**

- `SUBD F8, F6, F2` **finishes execution and writes result**:
  - The result of `F8 = F6 - F2` is now available
  - **Broadcasts on the CDB**:
    - `Tag = ADD1` (RS that held `SUBD`)
    - `Value =` result of `F6 - F2`
- Forwarding from CDB occurs:
  - **Register File**:
    - `F8.Qi == ADD1`, then write result in `F8` and set `F8.Qi = 0`
  - **Reservation Stations**:
    - `ADDD F6, F8, F2` is waiting for `F8`;
    - If `Qj == ADD1`, then `Vj = v(F8)` and `Qj = 0`
- The current execution `MULD` is still running (latency is 10 cycles) and will finish around cycle 17.

Instruction	Issue	Start Execute	Write Result
`LD F6, 34(R2)`	1	2	4
`LD F2, 45(R3)`	2	5	7
`MULTD F0, F2, F4`	3	8	
`SUBD F8, F6, F2`	4	8	10
`DIVD F10, F0, F6`	5		
`ADDD F6, F8, F2`	6		

Instruction status.

Name	Vj	Qj	Vk	Qk
Load1				
Load2				
EXLoad				

`EXLoad` (or `EX_LD`).

Name	Vj	Qj	Vk	Qk
ADD1	v(F6)		v(F2)	
ADD2	v(F8)		v(F2)	
ExADD	v(F6)		v(F2)	

`ExADD`.

Name	Vj	Qj	Vk	Qk
MUL1	v(F2)		v(F4)	
MUL2		MUL1	v(F6)	
ExMUL	v(F2)		v(F4)	

`ExMUL`.

RF	0	1	2	3	4	5	6	7	8	9	10	...	31
Qi	MUL1		v(F2)				ADD2		v(F8)		MUL2		

Register Result Status.

11. **Cycle 11**

- `ADDD F6, F8, F2` **starts execution**.
- `MULTD F0, F2, F4` continues execution.
- `DIVD` still waiting.

Instruction	Issue	Start Execute	Write Result
`LD F6, 34(R2)`	1	2	4
`LD F2, 45(R3)`	2	5	7
`MULTD F0, F2, F4`	3	8	
`SUBD F8, F6, F2`	4	8	10
`DIVD F10, F0, F6`	5		
`ADDD F6, F8, F2`	6	11	

Instruction status.

Name	Vj	Qj	Vk	Qk
Load1				
Load2				
EXLoad				

`EXLoad` (or `EX_LD`).

Name	Vj	Qj	Vk	Qk
ADD1				
ADD2	v(F8)		v(F2)	
ExADD	v(F8)		v(F2)	

`ExADD`.

Name	Vj	Qj	Vk	Qk
MUL1	v(F2)		v(F4)	
MUL2		MUL1	v(F6)	
ExMUL	v(F2)		v(F4)	

`ExMUL`.

RF	0	1	2	3	4	5	6	7	8	9	10	...	31
Qi	MUL1		v(F2)				ADD2		v(F8)		MUL2		

Register Result Status.

13. **Cycle 13**

- `ADDD F6, F8, F2` **completes execution and write result**.
- **WAW hazard on** `F6` **is avoided**:
  - `LD F6, 34(R2)` originally wrote to `F6` → `F6.Qi = Load1`
  - `ADDD F6, F8, F2` overwrites `F6` → `F6.Qi = ADD2`
  - But `DIVD`, which uses `F6`, was issued in cycle 5:
    - It read the **value of** `F6` at that time from `Load 1`
    - It stored it into its reservation station
    - So even if `F6` gets overwritten later by `ADDD`, it doesn't matter
    - `DIVD` already has the value it needed

Instruction	Issue	Start Execute	Write Result
`LD F6, 34(R2)`	1	2	4
`LD F2, 45(R3)`	2	5	7
`MULTD F0, F2, F4`	3	8	
`SUBD F8, F6, F2`	4	8	10
`DIVD F10, F0, F6`	5		
`ADDD F6, F8, F2`	6	11	13

Instruction status.

Name	Vj	Qj	Vk	Qk
Load1				
Load2				
EXLoad				

`EXLoad` (or `EX_LD`).

Name	Vj	Qj	Vk	Qk
ADD1				
ADD2	v(F8)		v(F2)	
ExADD	v(F8)		v(F2)	

`ExADD`.

Name	Vj	Qj	Vk	Qk
MUL1	v(F2)		v(F4)	
MUL2		MUL1	v(F6)	
ExMUL	v(F2)		v(F4)	

`ExMUL`.

RF	0	1	2	3	4	5	6	7	8	9	10	...	31
Qi	MUL1		v(F2)				v(F6)		v(F8)		MUL2		

Register Result Status.

18. **Cycle 18**

- `MULTD F0, F2, F4` **completes execution**. Executed in `MUL1` from cycle 8 to 18 (10 cycles total latency) and now broadcasts result on the Common Data Bus (CDB).
- `DIVD` is waiting for `F0`, but it is **now fully ready to start execution in the next cycle**.

Instruction	Issue	Start Execute	Write Result
`LD F6, 34(R2)`	1	2	4
`LD F2, 45(R3)`	2	5	7
`MULTD F0, F2, F4`	3	8	18
`SUBD F8, F6, F2`	4	8	10
`DIVD F10, F0, F6`	5		
`ADDD F6, F8, F2`	6	11	13

Instruction status.

Name	Vj	Qj	Vk	Qk
Load1				
Load2				
EXLoad				

`EXLoad` (or `EX_LD`).

Name	Vj	Qj	Vk	Qk
ADD1				
ADD2				
ExADD				

`ExADD`.

Name	Vj	Qj	Vk	Qk
MUL1	v(F2)		v(F4)	
MUL2	v(F0)		v(F6)	
ExMUL	v(F2)		v(F4)	

`ExMUL`.

RF	0	1	2	3	4	5	6	7	8	9	10	...	31
Qi	v(F0)		v(F2)				v(F6)		v(F8)		MUL2		

Register Result Status.

19. **Cycle 19**

- `DIVD F10, F0, F6` **begins execution in** `MUL2`.
- Division latency is not shown, but based on typical values in FP pipelines (and possibly earlier context), it may be **up to 20 cycles**. Therefore, `DIVD` will likely finish in a later cycle (e.g., 38+ depending on design).

Instruction	Issue	Start Execute	Write Result
`LD F6, 34(R2)`	1	2	4
`LD F2, 45(R3)`	2	5	7
`MULTD F0, F2, F4`	3	8	18
`SUBD F8, F6, F2`	4	8	10
`DIVD F10, F0, F6`	5	19	
`ADDD F6, F8, F2`	6	11	13

Instruction status.

Name	Vj	Qj	Vk	Qk
Load1				
Load2				
EXLoad				

`EXLoad` (or `EX_LD`).

Name	Vj	Qj	Vk	Qk
ADD1				
ADD2				
ExADD				

`ExADD`.

Name	Vj	Qj	Vk	Qk
MUL1				
MUL2	v(F0)		v(F6)	
ExMUL	v(F0)		v(F6)	

`ExMUL`.

RF	0	1	2	3	4	5	6	7	8	9	10	...	31
Qi	v(F0)		v(F2)				v(F6)		v(F8)		MUL2		

Register Result Status.

59. **Cycle 59**

- `DIVD F10, F0, F6` **completes execution and writes result** after a long 40-cycle latency.
- All instructions have issued, executed and written their results. No hazards occured, and Tomasulo's algorithm correctly handled:
  - ✅ RAW (by waiting with tags).
  - ✅ WAR and WAW (by renaming registers with `Qi`, `Qj`, `Qk`).
  - ✅ Long execution delays (like the 40-cycle `DIVD`) without blocking unrelated instructions.

Instruction	Issue	Start Execute	Write Result
`LD F6, 34(R2)`	1	2	4
`LD F2, 45(R3)`	2	5	7
`MULTD F0, F2, F4`	3	8	18
`SUBD F8, F6, F2`	4	8	10
`DIVD F10, F0, F6`	5	19	59
`ADDD F6, F8, F2`	6	11	13

Instruction status.

Name	Vj	Qj	Vk	Qk
Load1				
Load2				
EXLoad				

`EXLoad` (or `EX_LD`).

Name	Vj	Qj	Vk	Qk
ADD1				
ADD2				
ExADD				

`ExADD`.

Name	Vj	Qj	Vk	Qk
MUL1				
MUL2	v(F0)		v(F6)	
ExMUL	v(F0)		v(F6)	

`ExMUL`.

RF	0	1	2	3	4	5	6	7	8	9	10	...	31
Qi	v(F0)		v(F2)				v(F6)		v(F8)		v(F10)		

Register Result Status.

3.7.7 Tomasulo vs. Scoreboarding

Both Tomasulo's algorithm and Scoreboarding (introduced in the CDC 6600) are **dynamic instruction scheduling** techniques designed to:

- Improve **Instruction-Level Parallelism (ILP)**
- Allow **out-of-order execution**
- Handle **data hazards** without compiler involvement

But they differ in:

- *How they track operands*
- *Where renaming happens*
- *How results are forwarded*
- *How decentralized the logic is*

How Tomasulo improves over Scoreboarding

1. **Eliminates WAR and WAW hazards**. Scoreboarding can only resolve RAW hazards, WAR and WAW cause **stalls** in Scoreboarding. Tomasulo's **register renaming via RS tags** removes these conflicts entirely.
2. **Allows hardware-level dataflow execution**. Scoreboarding relies on centralized logic to check operand availability. Tomasulo allows **instructions to execute as soon as operands are available**, independently. This creates a **data-driven execution model**, more similar to modern superscalar and out-of-order CPUs.
3. **Forwarding via the Common Data Bus (CDB)**
   - In Scoreboarding: instructions must wait for register file updates
   - In Tomasulo: values are **broadcast to all RSs and RF** as soon as they're computed
4. **Reservation Stations = Smarter Instruction Buffers**.

   Hold operands, operation type, and source RS tags. Track readiness locally and act like **mini control units**, freeing up the need for centralized logic (Scoreboard).

However, despite its limitations, Scoreboarding:

- **Simpler to implement**
- Historically foundational (used in the CDC 6600)
- More **hardware-economical** for small-scale or embedded designs

But for high-performance out-of-order execution, Tomasulo's model is the clear winner.

Scoreboarding	Tomasulo's Algorithm
Operand Tracking	
Uses **register availability**	Uses **tag-based tracking** via RS
Register Renaming	
✗ None	✓ Implicit via RSs (`Qi`/`Qj`/`Qk`)
WAR/WAW Hazards	
✗ Must be stalled explicitly	✓ Eliminated by renaming
Operand Read Timing	
At execution **start**	At **issue** (or when available) via CDB
Operand Forwarding	
✗ No, must wait for register write-back	✓ Yes, uses **Common Data Bus** (CDB)
Control Model	
Centralized Scoreboard	Decentralized logic at each RS + FU
Functional Units	
Check Scoreboard for availability	RS check tags and readiness locally
Instruction Readiness	
RAW only (no renaming)	RAW + renamed operands (data-driven)
Common Bus	
✗ No shared result forwarding	✓ Shared broadcast on CDB
Complexity	
Simpler renaming logic	More complex, but higher performance

Table 19: Tomasulo vs. Scoreboarding.

⚠ Drawbacks of Tomasulo's Algorithm

While Tomasulo's algorithm was a **revolutionary advancement** in dynamic scheduling, especially for floating-point pipelines, it is not without **limitations**. As computer architecture evolved into superscalar and multithreaded designs, certain drawbacks of Tomasulo's original formulation became more evident.

1. **Complex Hardware Implementation**. Tomasulo's distributed, tag-driven control system requires:
   - **Reservation Stations (RS)** with operand tracking logic
   - **Register File with tag fields** (`Qi`)
   - A **Common Data Bus (CDB)** that broadcasts to **many destinations**
   - Logic to **match tags and grab values** dynamically

While this provides high performance, it increases **hardware area**, **control complexity** and **power consumption**.

In particular, **scaling the CDB** becomes harder with many Functional Units and many RSs, since broadcasting to all of them becomes a bottleneck.

2. **Centralized Common Data Bus (CDB) Bottleneck**. Tomasulo relies on a **single CDB** to broadcast results to all waiting RSs and update the register file. But:

   - Only **one instruction can write** on the CDB per cycle
   - This **limits write throughput**
   - It becomes a **bottleneck** in wide-issue (superscalar) processors

   Modern designs mitigate this with multiple CDBs, local bypassing networks, and reorder buffers (ROB) with centralized commit stages (advanced topics).

3. **Lack of Support for Precise Exceptions**. Tomasulo as originally designed does **not support precise exceptions**.[6] This problem is due to the structure of the algorithm:

   - Tomasulo allows **out-of-order completion** and **in-place register writes**
   - Once a value is written to the register file via CDB, it's hard to "undo" it.

   Scoreboarding can handle precise exceptions more gracefully. However, modern CPUs integrate Tomasulo-style scheduling with ROB (Reorder Buffer) to restore precise exceptions.

4. **Limited Scalability for Superscalar Designs**. Original Tomasulo was built for **one instruction per cycle** pipelines (like IBM 360/91). But in **superscalar** processors:

   - Multiple instructions issue and retire each cycle;
   - The CDB and tag-matching logic must scale accordingly;
   - Reservation Stations structures become more complex with many FUs and wider pipelines.

   Modern out-of-order cores use advanced techniques such as Register Alias Tables (RAT), physical register files, and ROB-based commit logic. These ideas **evolved from Tomasulo** , but allow better scaling and modularity.

[6] In precise exception handling, all instructions **before the faulting one** must complete, and none **after it** should affect the architectural state.

3.7.8 Register Renaming

The Register Renaming techniques have already been discussed in the previous sections. In section 3.1.2 (page 71) we gave a very short presentation dedicated to understand how to solve name dependencies, and in section 3.7.2 (page 136) we have seen what is the difference between static register renaming and implicit register renaming. In this section we will go deeper and understand how to apply this technique to the tomasulo algorithm.

3.7.8.1 Introduction

In modern out-of-order execution processors, one of the most significant **bottlenecks** to exploiting Instruction-Level Parallelism (ILP) lies in **false data dependencies**. Among these, **WAR (Write After Read)** and **WAW (Write After Write)** hazards can severely restrict the freedom to reorder instructions during execution. These are not **true dependencies** (like RAW, Read After Write), but are constraints imposed by the **limited number of registers** visible in the Instruction Set Architecture (ISA).

To tackle these issues, **Register Renaming** is employed, a mechanism that dynamically or statically **replaces architectural registers with a larger set of physical registers**, breaking these false dependencies and enabling more parallel execution.

Types of Register Renaming

There are **two primary approaches** to register renaming:

- **Static Register Renaming**, handled at **compile-time**, often via techniques like **loop unrolling** and aggressive register allocation. This relies on **ISA-visible registers** and requires the compiler to explicitly assign new register names to avoid reuse conflicts.
- **Implicit (Dynamic) Register Renaming**, which is employed in hardware at **runtime**, often as part of advanced scheduling mechanisms such as **Tomasulo's algorithm**. This method uses structures like **reservation stations** and **reorder buffers** to rename registers implicitly and track dependencies through tags instead of register names.

Why Register Renaming is important?

Without register renaming, WAR and WAW hazards can stall instruction issue unnecessarily:

- A WAR hazard arises when an instruction wants to write to a register that a previous instruction still needs to read.
- A WAW hazard occurs when two instructions write to the same register, and their order must be preserved to maintain correctness.

By decoupling the programmer-visible registers from the physical storage used in the microarchitecture, renaming allows:

- ✔ **Multiple instructions** writing to the "same" architectural register to execute out of order.
- ✔ **Concurrent execution** of independent iterations of loops.
- ✔ **Elimination of false dependencies**, improving pipeline utilization and throughput.

Register renaming is a fundamental enabler of modern ILP exploitation. Whether implemented statically via compiler transformations or dynamically through hardware mechanisms like Tomasulo's algorithm, it allows us to **overlap instruction execution** aggressively and avoid performance penalties due to naming limitations of the architectural register file.

3.7.8.2 Loop Unrolling and Code Scheduling

Tomasulo's algorithm is a classic dynamic scheduling method that **naturally incorporates implicit register renaming**. It is designed to **overlap instruction execution** even when instructions involve true or false dependencies, by dynamically resolving hazards at runtime.

In this section, we focus specifically on **how Tomasulo's algorithm deals with a single loop**, showing the issues of data hazards and motivating why **renaming**, even if implicit, is critical. Furthermore, we show three techniques, included register renaming, to reach the best optimization. We also show three techniques, including register renaming, to achieve the best optimization.

Note that these topics are already covered in the Tomasulo section (3.7.2, page 137).

Why Renaming?

Consider the following floating-point loop:

```
1 Loop:  LD     F0, 0(R1)       ; Load word into F0 from address 0+R1
2        MULTD  F4, F0, F2      ; Multiply F0 and F2, result into F4
3        SD     F4, 0(R1)       ; Store word from F4 to address 0+R1
4        SUBI   R1, R1, #8      ; Decrement R1 by 8
5        BNEZ   R1, Loop        ; Branch to Loop if R1 ≠ 0
```

Each loop iteration consists of five instructions. We assume branch prediction correctly predicts the loop branch as taken. *But are there dependencies?* Yes, **several hazards arise**:

1. RAW (Read After Write) on `F0` and `F4`
2. WAW (Write After Write) on `F0` and `F4` across iterations
3. WAR (Write After Read) on `R1` due to address updates and branching

These **hazards limit how aggressively we can overlap loop iterations**. Let's break down the hazard types across consecutive iterations:

- **Within a single iteration**:
  - `MULTD` depends on the result of `LD` (needs `F0`).
  - `SD` depends on the result of `MULTD` (needs `F4`).
  - `BNEZ` depends on `SUBI` to determine the loop condition.
- **Across iterations**:
  - `LD` of next iteration **writes** `F0`, creating a **WAW hazard** with the previous `LD`.
  - `MULTD` of next iteration **writes** `F4`, also leading to **WAW hazard**.
  - `SUBI` and `BNEZ` reuse `R1`, creating **WAR hazards**.

Without renaming, these hazards **would serialize iterations**, nullifying any gain from speculative execution.

Tomasulo's Contribution: Implicit Register Renaming

Tomasulo's algorithm automatically resolves these hazards by **renaming registers dynamically**:

- It uses **reservation stations** that hold **tags** instead of register names.
- When an instruction needs a value, it can either:
  - Use the value directly if it's ready;
  - Or wait for a tag associated with the producing reservation station.
- WAW and WAR hazards are naturally eliminated because **register names are no longer the synchronization point**, tags are!

Thus, **each iteration of the loop can proceed independently** as soon as its operands are available, even if they involve the same architectural register names.

Loop Unrolling combined with Register Renaming

While Tomasulo's algorithm handles implicit renaming at runtime, an alternative strategy to reduce the impact of dependencies, particularly across loop iterations, is to act **at compile time**, through a technique called **Loop Unrolling** combined with **Register Renaming**.

Loop Unrolling is a **compiler optimization technique** (or sometimes a manual optimization) that **expands the loop body** by replicating its operations multiple times in sequence, reducing the number of iterations and minimizing the overhead of branch instructions. Instead of executing a small amount of work many times, we "***stretch out***" multiple loop iterations into a single, larger block of straight-line code.

In our context, Loop Unrolling aims to **expose more parallelism statically** and to **reduce the loop overhead** by executing multiple iterations at once. Register Renaming in this context is crucial: **it eliminates artificial WAW and WAR dependencies** that would otherwise serialize execution.

Let's revisit the same loop we discussed before:

```
1 Loop:  LD    F0, 0(R1)
2        MULTD F4, F0, F2
3        SD    F4, 0(R1)
4        SUBI  R1, R1, #8
5        BNEZ  R1, Loop
```

If we simply replicate this loop multiple times without renaming registers, WAW and WAR hazards would prevent full exploitation of the increased instruction stream. To eliminate such hazards, **we must rename the registers** properly during unrolling.

Here's the unrolled version of the loop, unrolled four times, with **explicit register renaming** to avoid hazards:

```
Loop:  LD     F0, 0(R1)
       MULTD  F4, F0, F2
       SD     F4, 0(R1)

       LD     F6, -8(R1)
       MULTD  F8, F6, F2
       SD     F8, -8(R1)

       LD     F10, -16(R1)
       MULTD  F12, F10, F2
       SD     F12, -16(R1)

       LD     F14, -24(R1)
       MULTD  F16, F14, F2
       SD     F16, -24(R1)

       SUBI   R1, R1, #32
       BNEZ   R1, Loop
```

- Observations
  - New registers are introduced: `F6`, `F8`, `F10`, `F12`, `F14`, `F16`.
  - **No two iterations reuse the same destination register**.
  - **WAW hazards** between iterations are eliminated.
  - **More operations are ready to issue independently**.
  - Original loop: 5 instructions per iteration. Unrolled loop: 14 instructions for 4 iterations, so 3.5 instructions per iteration on average. Thus, unrolling not only **exposes parallelism** but also **increases code density efficiency**.

Benefits

- ✔ **Higher ILP**: Each iteration's operations are now largely independent.
- ✔ **Reduced Loop Overhead**: The branch and counter updates are performed once every 4 iterations, decreasing control dependencies.
- ✔ **Efficient Use of Hardware**: Multiple functional units can work simultaneously, better exploiting available execution bandwidth.
- ✔ **Preparation for Further Optimizations**: The code can now be *rescheduled* to minimize stalls (we'll see this in future sections).

Finally, Tomasulo's algorithm **does not perform loop unrolling** by itself. Loop Unrolling is a **compiler optimization**. However, thanks to **dynamic register renaming and scheduling**, Tomasulo can **overlap multiple loop iterations even without unrolling**. This behavior mimics some of benefits of loop unrolling at runtime, but it doesn't physically unroll the code or replicate instructions.

Even more performance: Code Scheduling with Register Renaming

Once we have **unrolled a loop** and **renamed registers** to avoid false dependencies, the next logical optimization is **code scheduling**. The goal is now to **rearrange the unrolled instructions to minimize stalls**, especially those caused by **true RAW (Read After Write)** dependencies or **long execution latencies** (e.g., memory loads or floating-point operations). Thus, **even with register renaming**, **the order in which instructions are issued** still matters to achieve **maximum parallelism**.

Code Scheduling is the process of **reordering the instructions** in a program (especially after loop unrolling and register renaming) **to maximize parallelism** during execution. It is a **compiler-level** or sometimes **hardware-level** optimization. The goal is to feed the CPU continuously with ready-to-execution instructions.

Let's look at the previous code again, and by unrolling the loop by a factor of 4 and renaming the registers correctly, we had this sequence:

```
LD     F0, 0(R1)
MULTD  F4, F0, F2
SD     F4, 0(R1)
LD     F6, -8(R1)
MULTD  F8, F6, F2
SD     F8, -8(R1)
LD     F10, -16(R1)
MULTD  F12, F10, F2
SD     F12, -16(R1)
LD     F14, -24(R1)
MULTD  F16, F14, F2
SD     F16, -24(R1)
SUBI   R1, R1, #32
BNEZ   R1, Loop
```

Problem: if we execute this exactly in order, some instructions will have to wait unnecessarily for their operands because of true dependencies (particularly between `LD` and `MULTD`). So the **key idea** of code scheduling is:

- **Group independent instructions together** as much as possible.
- **Delay dependent instructions** just enough to satisfy their true data dependencies.
- **Hide long latencies** (e.g., waiting for a memory load) by **doing useful work in parallel**.

After careful rescheduling, the reordered code looks like:

```
LD     F0, 0(R1)
LD     F6, -8(R1)
LD     F10, -16(R1)
LD     F14, -24(R1)
MULTD  F4, F0, F2
MULTD  F8, F6, F2
MULTD  F12, F10, F2
MULTD  F16, F14, F2
SD     F4, 0(R1)
```

```
10 SD     F8, -8(R1)
11 SD     F12, -16(R1)
12 SUBI   R1, R1, #32
13 BNEZ   R1, Loop
14 SD     F16, -24(R1)    ; In branch delay slot
```

The changes are:

- All **loads** are moved upfront, memory latency can be hidden while computing.
- **Multiplications** (dependent on loads) are grouped next.
- **Stores** happen **only after the corresponding multiply is finished**, respecting the RAW dependency.
- `SUBI` and `BNEZ` are moved right before the branch to reduce branching penalties.
- The **last store** (`SD F16`) is placed in the **branch delay slot**, making clever use of a wasted cycle.

And the benefits are:

✔ **Load latency hidden**. Loads start early while compute units are idle.

✔ **Maximal Floating Point unit usage**. Multiple `MULTD`s can be issued in parallel.

✔ **Minimal pipeline stalls**. Operands ready when needed.

✔ **Reduced branch penalties**. Good use of branch delay slot.

Technique	How Tomasulo handles it
Register Renaming	✔ **Yes**, dynamic, using RSs and tags.
Code Scheduling	✔ **Yes**, dynamic, based on operand availability.
Loop Unrolling	✖ **No**, but it can dynamically overlap iterations thanks to renaming + scheduling.

Table 20: Does Tomasulo's algorithm adopt Loop Unrolling, Register Renaming, and Code Scheduling?

3.7.8.3 How Tomasulo Overlaps Iterations of Loops

Tomasulo's algorithm can **dynamically overlap multiple iterations** of a loop **even without unrolling the code**, thanks to **three main tricks**:

- **Implicit Register Renaming via Reservation Stations**. Tomasulo does **not use the architectural register names** (like `F0`, `F4`) to track data anymore. Instead, it replaces every operand with a **dynamic tag** that points to a **Reservation Station (RS)**. This means: two iterations that both write to `F0` are treated separately, because each `LD` or `MULTD` producing `F0` will **have its own RS tag**.

  ✅ **WAR and WAW hazards disappear.**

- **Each instruction gets its own Reservation Station**. In Tomasulo, each new instruction (even from a new loop iteration) **gets its own Reservation Station**. So multiple instances of `LD`, `MULTD`, etc., from different loop iterations **can be active at the same time**. Even if the instructions all logically use `F0`, `F4`, etc., **hardware sees them as completely different** because they are tied to **different RS entries**.

  ✅ **Dynamic loop unrolling effect**: many loop iterations proceed in parallel without modifying the loop code.

- **Multiple Functional Units and Branch Prediction**. To exploit this properly, we must have:

  1. **Multiple Functional Units** (e.g., several FP multipliers and memory load units).
  2. **Branch Prediction** that allows **speculative execution** beyond the loop control instruction (like `BNEZ`).

  Thus, Tomasulo **speculatively fetches, issues, and schedules** instructions from **many iterations ahead**, even **before knowing** if the loop will finish.

  ✅ **Instruction issuing moves faster than control flow.**

Tomasulo overlaps loop iterations **by making the Reservation Stations act like an extended virtual register file**, where each operation, even from different iterations, can proceed independently and dynamically, breaking naming limitations. This is sometimes called: **Dynamic Loop Unrolling**, **Dynamic Specualtive Execution**, **Dynamic Out-of-Order Parallelism**. All enabled automatically by Tomasulo's mechanism.

3.7.8.4 Tomasulo Loop Execution

We now analyze a **concrete example** of how Tomasulo's algorithm **dynamically overlaps multiple loops iterations**, using **register renaming** and **out-of-order execution**.

The loop we consider is:

```
Loop:
        LD    F0, 0(R1)
        MULTD F4, F0, F2
        SD    F4, 0(R1)
        SUBI  R1, R1, #8
        BNEZ  R1, Loop
```

Setup Assumptions

- **Cache miss** on the **first** `LD` → 8 clock cycles.
- **Cache hit** on the **second** `LD` → 1 clock cycles.
- `FP MULTD` operation **latency** → 4 clock cycles.
- **Branch Prediction** → branch assumed **taken** (so iterations continue).
- Only `SUBI` and `BNEZ` clock times are shown explicitly for clarity.
- 5 instructions per loop iteration.

Thus, **memory accesses are slow**, and **floating-point operations are moderately slow** compared to integer operations.

Hazards

✗ **RAW hazards** on `F0`, `F4`, `R1`.

✓ (**eliminated dynamically**, thanks to Tomasulo's implicit register renaming) **WAW hazards** mainly on `R1`.

✓ (**eliminated dynamically**, thanks to Tomasulo's implicit register renaming) **WAW hazards** on `F0` and `F4` across iterations.

Structures in Tomasulo's algorithm

- **Instruction Status Table** (or Instruction Queue, or Tracking Table), tracks the status of each issued instruction through its three major stages: issue, execute and write result. Allow us to monitor the instruction flow cycle-by-cycle.
- **Reservation Stations Table**, temporarily hold instructions waiting to execute, **along with operand values or operand tags** if values are not yet available. Performs dynamic register renaming and resolve dependencies internally before issuing instructions to functional units.

- **Register Result Status Table**, tells **which Reservation Station (RS) is responsible for generating the current value of each architectural register**. Allows forwarding and prevents WAR/WAW hazards, by making reads wait for the right instruction to finish.
- **Load and Store Buffers (Memory Buffers)**, manage memory operations separately, since memory addresses take time to compute and loads/-stores have different latencies. Supports out-of-order memory access while ensuring correct memory ordering when necessary.

1. **Cycle 1**

   - The **first instruction** from iteration 1, `LD F0, 0(R1)`, is issued. It is assigned to Load Buffer 1 (`Load1`), marked as busy. The memory address to load from is calculated: `R1 = 80`, so the load address is `80`.
   - `F0` is now associated with `Load1` in the register status table. This means:
     - Any instruction that wants to read `F0` must wait for `Load1` to complete and broadcast its result.
     - Tomasulo does not write to `F0` immediately, tag `Load1` is what matters now.
   - The other instructions (`MULTD`, `SD`, etc.) are **not issued yet** in this cycle. They are waiting because their source operands depend on `F0` (the load result), which is **not ready** yet. Only the first load is active at this point.

   At cycle 1, we **prepare the first data** to flow through Tomasulo's architecture: we **rename** `F0` dynamically, **assign** it to a load buffer, and **start the memory access** (which will take 8 cycles because of cache miss). All following instructions **must track this tag** (`Load1`) to know when `F0` will be ready.

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1		
1	`MULTD F4, F0, F2`			
1	`SD F4, 0(R1)`			
2	`LD F0, 0, R1`			
2	`MULTD F4, F0, F2`			
2	`SD F4, 0(R1)`			

Instruction status.

	Busy	Addr	FU
Load1	Yes	80	\
Load2	No		\
Load3	No		\
Store1	No		
Store2	No		
Store3	No		

Load and Store Buffers.

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
1	80	Load1								

Register result status.

2. **Cycle 2**

- Instruction `MULTD F4, F0, F2` (from iteration 1) is **issued**. It is assigned to `Mult1` (a multiplication Reservation Station). Operands:
  - `F2` is ready, its value is available → `Vj = R(F2)`
  - `F0` is *not* ready, **waiting on** `Load1` to produce it → `Qj = Load1`.
- `F0` is still pending, being loaded by `Load1`. `F4` will eventually be produced by the `MULTD` operation (currently waiting for `F0` to complete).

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1		
1	`MULTD F4, F0, F2`	2		
1	`SD F4, 0(R1)`			
2	`LD F0, 0, R1`			
2	`MULTD F4, F0, F2`			
2	`SD F4, 0(R1)`			

Instruction status.

	Busy	Addr	FU
`Load1`	Yes	80	\
`Load2`	No		\
`Load3`	No		\
`Store1`	No		
`Store2`	No		
`Store3`	No		

Load and Store Buffers.

Time	Name	Busy	`Op`	`Vj`	`Vk`	`Qj`	`Qk`
	`Add1`	No					
	`Add2`	No					
	`Add3`	No					
	`Mult1`	Yes	`MULTD`		`R(F2)`	`Load1`	
	`Mult2`	No					

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
2	80	Load1		Mult1						

Register result status.

3. **Cycle 3**

- Instruction `SD F4, 0(R1)` (store) from iteration 1 is **issued**. It is assigned to `Store1` (store buffer). The store buffer must wait for `F4` to be computed by `Mult1`. We record the tag of the functional unit that will produce the needed value `F4`.

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1		
1	`MULTD F4, F0, F2`	2		
1	`SD F4, 0(R1)`	3		
2	`LD F0, 0, R1`			
2	`MULTD F4, F0, F2`			
2	`SD F4, 0(R1)`			

Instruction status.

	Busy	Addr	FU
`Load1`	Yes	80	\
`Load2`	No		\
`Load3`	No		\
`Store1`	Yes	80	`Mult1`
`Store2`	No		
`Store3`	No		

Load and Store Buffers.

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	`Add1`	No					
	`Add2`	No					
	`Add3`	No					
	`Mult1`	Yes	`MULTD`		`R(F2)`	`Load1`	
	`Mult2`	No					

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
3	80	`Load1`		`Mult1`						

Register result status.

6. **Cycle 6**

- We have skipped cycle 4 and 5 because we are only waiting for `LD`, `MULTD` or `SD` updates. In cycle 5, register `R1` becomes `72` due to the operation `SUBI R1, R1, #8`.
- Instruction `LD F0, 0(R1)` from iteration 2 is now issued into `Load2` buffer. This new load has a cache hit (remember: only the first load had a cache miss).
- `F0` is again associated with `Load2` (the second load). The previous `F0` result (from `Load1`) is **overwritten** by the new `F0` pending load. Even though the logical register `F0` is overwritten, the data dependency graph remains correct, because **Tomasulo tracks data flow by tags**, not register names. Thus:
  - `MULTD F4, F0, F2` (iteration 1): waits for `Load1` to complete (old `F0`)
  - `MULTD F4, F0, F2` (iteration 2): will wait for `Load2` to complete (new `F0`)

  ✅ **WAW hazard avoided** by dynamic renaming. New instructions get the new `F0` and old instructions already locked the old value.

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1		
1	`MULTD F4, F0, F2`	2		
1	`SD F4, 0(R1)`	3		
2	`LD F0, 0, R1`	6		
2	`MULTD F4, F0, F2`			
2	`SD F4, 0(R1)`			

Instruction status.

	Busy	Addr	FU
`Load1`	Yes	80	\
`Load2`	Yes	72	\
`Load3`	No		\
`Store1`	Yes	80	`Mult1`
`Store2`	No		
`Store3`	No		

Load and Store Buffers.

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F2)	Load1	
	Mult2	No					

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
6	72	Load2		Mult1						

Register result status.

7. **Cycle 7**

   - **Even though the second load is ready** (cache hit, 1 cycle latency, and issued in cycle 6), it must **wait for the first load to complete** because they share the same architectural address and could have aliasing or store/load order dependencies. So unless advanced speculation is used, **loads and stores to the same address must be done in program order**.
     This is called **Memory Disambiguation**. In out-of-order processors, memory disambiguation is the logic that **decides whether two loads/stores/loads can safely reorder**. Tomasulo in its original form **does not perform aggressive disambiguation**.
   - `MULTD F4, F0, F2` (iteration 2) is issued.

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1		
1	`MULTD F4, F0, F2`	2		
1	`SD F4, 0(R1)`	3		
2	`LD F0, 0, R1`	6		
2	`MULTD F4, F0, F2`	7		
2	`SD F4, 0(R1)`			

Instruction status.

	Busy	Addr	FU
`Load1`	Yes	80	\
`Load2`	Yes	72	\
`Load3`	No		\
`Store1`	Yes	80	`Mult1`
`Store2`	No		
`Store3`	No		

Load and Store Buffers.

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	`Add1`	No					
	`Add2`	No					
	`Add3`	No					
	`Mult1`	Yes	`MULTD`		`R(F2)`	`Load1`	
	`Mult2`	Yes	`MULTD`		`R(F2)`	`Load2`	

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
7	72	Load2		Mult2						

Register result status.

8. **Cycle 8**

- `SD F4, 0(R1)` (iteration 2) is issued. It is assigned to `Store2` (store buffer). The store buffer must wait for `F4` to be computed by `Mult2`. We record the tag of the functional unit that will produce the needed value `F4`.
- **WAW (Write After Write) hazards on registers `F0` and `F4` have been resolved**, meaning the first and second iterations are now **fully overlapped**.

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1		
1	`MULTD F4, F0, F2`	2		
1	`SD F4, 0(R1)`	3		
2	`LD F0, 0, R1`	6		
2	`MULTD F4, F0, F2`	7		
2	`SD F4, 0(R1)`	8		

Instruction status.

	Busy	Addr	FU
`Load1`	Yes	80	\
`Load2`	Yes	72	\
`Load3`	No		\
`Store1`	Yes	80	`Mult1`
`Store2`	Yes	72	`Mult2`
`Store3`	No		

Load and Store Buffers.

Time	Name	Busy	`Op`	`Vj`	`Vk`	`Qj`	`Qk`
	`Add1`	No					
	`Add2`	No					
	`Add3`	No					
	`Mult1`	Yes	`MULTD`		`R(F2)`	`Load1`	
	`Mult2`	Yes	`MULTD`		`R(F2)`	`Load2`	

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
8	72	Load2		Mult2						

Register result status.

9. **Cycle 9**

- `LD F0, 0(R1)` (iteration 1, address 80, cache miss) finally completes and writes back in the next cycle (1 cycle latency). Now the `LD F0, 0(R1)` from iteration 2 can start its execution.
- Even though we don't track the `SUBI` instruction, it is issued in this cycle (iteration 2).

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1	9	
1	`MULTD F4, F0, F2`	2		
1	`SD F4, 0(R1)`	3		
2	`LD F0, 0, R1`	6		
2	`MULTD F4, F0, F2`	7		
2	`SD F4, 0(R1)`	8		

Instruction status.

	Busy	Addr	FU
`Load1`	Yes	80	\
`Load2`	Yes	72	\
`Load3`	No		\
`Store1`	Yes	80	`Mult1`
`Store2`	Yes	72	`Mult2`
`Store3`	No		

Load and Store Buffers.

Time	Name	Busy	`Op`	`Vj`	`Vk`	`Qj`	`Qk`
	`Add1`	No					
	`Add2`	No					
	`Add3`	No					
	`Mult1`	Yes	`MULTD`		`R(F2)`	`Load1`	
	`Mult2`	Yes	`MULTD`		`R(F2)`	`Load2`	

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
9	72	Load2		Mult2						

Register result status.

10. **Cycle 10**

- `LD F0, 0(R1)` (iteration 1) now writes back (1 cycle latency). It also sends the result through the CDB. The value written can be represented as `M[80]`, indicating the location of memory at address `80`.

  `Mult1` was listening in the CDB for the `Load1` value; now it receives the update, so it sets the `Qj` entry to zero and updates its `Vj` value with the value in memory at address `80` (`M[80]`).

- The second load (`Load2`) completes its execution in 1 cycle thanks to the cache hit. In the next cycle, the result is written back. This instruction was blocked by the previous load to avoid too aggressive memory disambiguation.

- Even though we don't track the `BNEZ` instruction, it is issued in this cycle (iteration 2). This updates the value in register `R1` from `72` to `64` (minus `8`).

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1	9	10
1	`MULTD F4, F0, F2`	2		
1	`SD F4, 0(R1)`	3		
2	`LD F0, 0, R1`	6	10	
2	`MULTD F4, F0, F2`	7		
2	`SD F4, 0(R1)`	8		

Instruction status.

	Busy	Addr	FU
`Load1`	No		\
`Load2`	Yes	72	\
`Load3`	No		\
`Store1`	Yes	80	`Mult1`
`Store2`	Yes	72	`Mult2`
`Store3`	No		

Load and Store Buffers.

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M[80]	R(F2)		
	Mult2	Yes	MULTD		R(F2)	Load2	

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
10	64	Load2		Mult2						

Register result status.

11. **Cycle 11**

- `LD F0, 0(R1)` (iteration 2) now writes back (1 cycle latency). It also sends the result through the CDB. The value written can be represented as `M[72]`, indicating the location of memory at address `72`.

  `Mult2` was listening in the CDB for the `Load2` value; now it receives the update, so it sets the `Qj` entry to zero and updates its `Vj` value with the value in memory at address `72` (`M[72]`).
- The third load (`Load3`) starts its execution because the cycle continues to execute. So we are now at iteration number 3, at the first load command: `LD F0, 0(R1)`. So we add the entry `Load3` to the table *Load and Store Buffers*.
- `Mult1` continues its execution, it has 3 cycles left to complete.

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1	9	10
1	`MULTD F4, F0, F2`	2		
1	`SD F4, 0(R1)`	3		
2	`LD F0, 0, R1`	6	10	11
2	`MULTD F4, F0, F2`	7		
2	`SD F4, 0(R1)`	8		

Instruction status.

	Busy	Addr	FU
`Load1`	No		\
`Load2`	No		\
`Load3`	Yes	64	\
`Store1`	Yes	80	`Mult1`
`Store2`	Yes	72	`Mult2`
`Store3`	No		

Load and Store Buffers.

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD	M[80]	R(F2)		
4	Mult2	Yes	MULTD	M[72]	R(F2)		

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
11	64	Load3		Mult2						

Register result status.

12. **Cycle 12**

- `MULTD F4, F0, F2` (iteration 3) cannot be issued, because in Tomasulo's algorithm, to issue a new instruction there must be a free Reservation Station available for that operation type (here, floating-point multiply). If no RS is free, the instruction cannot be issued and must wait.

  Since both `Mult1` and `Mult2` are occupied executing previous multiplies, the third multiply (from iteration 3) cannot issue yet.
- `Mult1` continues its execution, it has 2 cycles left to complete.
- `Mult2` continues its execution, it has 3 cycles left to complete.

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1	9	10
1	`MULTD F4, F0, F2`	2		
1	`SD F4, 0(R1)`	3		
2	`LD F0, 0, R1`	6	10	11
2	`MULTD F4, F0, F2`	7		
2	`SD F4, 0(R1)`	8		

Instruction status.

	Busy	Addr	FU
`Load1`	No		\
`Load2`	No		\
`Load3`	Yes	64	\
`Store1`	Yes	80	`Mult1`
`Store2`	Yes	72	`Mult2`
`Store3`	No		

Load and Store Buffers.

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	`Add1`	No					
	`Add2`	No					
	`Add3`	No					
2	`Mult1`	Yes	`MULTD`	`M[80]`	`R(F2)`		
3	`Mult2`	Yes	`MULTD`	`M[72]`	`R(F2)`		

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
12	64	Load3		Mult2						

Register result status.

13. **Cycle 13**

- `LD F0, 0(R1)` completes its execution and writes the result back to the CDB (`M[64]`).
- `Mult1` continues its execution, it has 1 cycle left to complete.
- `Mult2` continues its execution, it has 2 cycles left to complete.

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1	9	10
1	`MULTD F4, F0, F2`	2		
1	`SD F4, 0(R1)`	3		
2	`LD F0, 0, R1`	6	10	11
2	`MULTD F4, F0, F2`	7		
2	`SD F4, 0(R1)`	8		

Instruction status.

	Busy	Addr	FU
`Load1`	No		\
`Load2`	No		\
`Load3`	No		\
`Store1`	Yes	80	`Mult1`
`Store2`	Yes	72	`Mult2`
`Store3`	No		

Load and Store Buffers.

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	`Add1`	No					
	`Add2`	No					
	`Add3`	No					
1	`Mult1`	Yes	`MULTD`	`M[80]`	`R(F2)`		
2	`Mult2`	Yes	`MULTD`	`M[72]`	`R(F2)`		

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
13	64	`M[64]`		`Mult2`						

Register result status.

14. **Cycle 14**

- `Mult1` finishes its execution at the end of this cycle! This will allow the `Store1` to write the result in memory at the address `80`, and also to the `MULTD` instruction in the iteration 3 to be issued.
- `Mult2` continues its execution, it has 1 cycle left to complete.

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1	9	10
1	`MULTD F4, F0, F2`	2	14	
1	`SD F4, 0(R1)`	3		
2	`LD F0, 0, R1`	6	10	11
2	`MULTD F4, F0, F2`	7		
2	`SD F4, 0(R1)`	8		

Instruction status.

	Busy	Addr	FU
`Load1`	No		\
`Load2`	No		\
`Load3`	No		\
`Store1`	Yes	80	`Mult1`
`Store2`	Yes	72	`Mult2`
`Store3`	No		

Load and Store Buffers.

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	`Add1`	No					
	`Add2`	No					
	`Add3`	No					
0	`Mult1`	Yes	`MULTD`	`M[80]`	`R(F2)`		
1	`Mult2`	Yes	`MULTD`	`M[72]`	`R(F2)`		

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
14	64	`M[64]`		`Mult2`						

Register result status.

15. **Cycle 15**

- `MULTD` (iteration 1) writes the result to the CDB; this triggers the `Store1` listener, which waits for the result to be stored in memory (the result of the load, `M[80]`, multiplied by the value of register `F2`), and the `MULTD` of iteration 3, which waits for a free reservation station to be issued.
- `Mult2` finishes its execution at the end of this cycle! This will allow the `Store2` to write the result in memory at the address `72`.

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1	9	10
1	`MULTD F4, F0, F2`	2	14	15
1	`SD F4, 0(R1)`	3		
2	`LD F0, 0, R1`	6	10	11
2	`MULTD F4, F0, F2`	7	15	
2	`SD F4, 0(R1)`	8		

Instruction status.

	Busy	Addr	FU
`Load1`	No		\
`Load2`	No		\
`Load3`	No		\
`Store1`	Yes	80	`M[80]` × `F2`
`Store2`	Yes	72	`Mult2`
`Store3`	No		

Load and Store Buffers.

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	`Add1`	No					
	`Add2`	No					
	`Add3`	No					
4	`Mult1`	Yes	`MULTD`	`M[64]`	`R(F2)`		
0	`Mult2`	Yes	`MULTD`	`M[72]`	`R(F2)`		

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
15	64	M[64]		Mult1						

Register result status.

16. **Cycle 16**

- `MULTD` (iteration 2) writes the result to the CDB; this triggers the `Store2` listener, which waits for the result to be stored in memory (the result of the load, `M[72]`, multiplied by the value of register `F2`).

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1	9	10
1	`MULTD F4, F0, F2`	2	14	15
1	`SD F4, 0(R1)`	3	16	
2	`LD F0, 0, R1`	6	10	11
2	`MULTD F4, F0, F2`	7	15	16
2	`SD F4, 0(R1)`	8		

Instruction status.

	Busy	Addr	FU
`Load1`	No		\
`Load2`	No		\
`Load3`	No		\
`Store1`	Yes	80	`M[80] × F2`
`Store2`	Yes	72	`M[72] × F2`
`Store3`	Yes	64	`Mult1`

Load and Store Buffers.

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	`Add1`	No					
	`Add2`	No					
	`Add3`	No					
3	`Mult1`	Yes	`MULTD`	`M[64]`	`R(F2)`		
	`Mult2`	No					

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
16	64	`M[64]`		`Mult1`						

Register result status.

17. **Cycle 17**

- `SD` (iteration 1) writes the result of the expression `M[80]` $\times$ `F2` to memory at address `80`.
- Since the `SD` instruction (iteration 1) has finished its execution, the second store (iteration 2) can start its execution and send the result to memory between `M[72]` and `F2`.

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1	9	10
1	`MULTD F4, F0, F2`	2	14	15
1	`SD F4, 0(R1)`	3	16	17
2	`LD F0, 0, R1`	6	10	11
2	`MULTD F4, F0, F2`	7	15	16
2	`SD F4, 0(R1)`	8	17	

Instruction status.

	Busy	Addr	FU
`Load1`	No		\
`Load2`	No		\
`Load3`	No		\
`Store1`	No		
`Store2`	Yes	72	`M[72]` $\times$ `F2`
`Store3`	Yes	64	`Mult1`

Load and Store Buffers.

Time	Name	Busy	`Op`	`Vj`	`Vk`	`Qj`	`Qk`
	`Add1`	No					
	`Add2`	No					
	`Add3`	No					
2	`Mult1`	Yes	`MULTD`	`M[64]`	`R(F2)`		
	`Mult2`	No					

Reservation Stations.

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
17	64	M[64]		Mult1						

Register result status.

18. **Cycle 18**

- `SD` (iteration 2) writes the result of the expression `M[72]` $\times$ `F2` to memory at address `72`.

Iter.	Instruction	Issue	Exec Comp	Write Res
1	`LD F0, 0, R1`	1	9	10
1	`MULTD F4, F0, F2`	2	14	15
1	`SD F4, 0(R1)`	3	16	17
2	`LD F0, 0, R1`	6	10	11
2	`MULTD F4, F0, F2`	7	15	16
2	`SD F4, 0(R1)`	8	17	18

Instruction status.

	Busy	Addr	FU
`Load1`	No		\
`Load2`	No		\
`Load3`	No		\
`Store1`	No		
`Store2`	No		
`Store3`	Yes	64	`Mult1`

Load and Store Buffers.

Time	Name	Busy	`Op`	`Vj`	`Vk`	`Qj`	`Qk`
	`Add1`	No					
	`Add2`	No					
	`Add3`	No					
1	`Mult1`	Yes	`MULTD`	`M[64]`	`R(F2)`		
	`Mult2`	No					

Reservation Stations.

Clock	`R1`	`F0`	`F2`	`F4`	`F6`	`F8`	`F10`	`F12`	...	`F30`
18	64	`M[64]`		`Mult1`						

Register result status.

3.7.9 Reorder Buffer (ROB)

3.7.9.1 Hardware-based Speculation

Speculation in hardware is a foundational technique in high-performance processors, used to execute instructions **before it is know whether they are needed**. This is especially relevant when instructions are **control-dependent** on unresolved branches.

In particular, hardware-based speculation enables:

- **Speculative instruction execution** before knowing branch outcomes.
- **Rollback** in case a mispredicted path was taken.

This approach increases Instruction-Level Parallelism (ILP), allowing more instructions to be in-flight, even if some turn out to be unnecessary.

Why speculation needs the ROB

Executing instructions speculatively introduces a challenge: "*how do we prevent side effects (e.g., register or memory updates) from mispredicted instructions?*". The **solution**: **defer the architectural state update** until it's safe to commit.

This is exactly what the **Reorder Buffer (ROB)** is for:

- ✔ It holds results of instructions that have **finished execution** but are **not yet committed**.
- ✔ It allows **instruction results to be written in-order**, preserving program semantics.
- ✔ In case of a misprediction, the ROB enables the processor to **flush invalid speculative results** quickly and precisely.

But the ROB is **critical** for:

- ⚠ Decoupling **execution completion** from **state update (commit)**.
- ⚠ Supporting **precise exceptions** (so the CPU can cleanly stop at the last correct instruction).
- ⚠ Managing **speculative and non-speculative instruction tracking**.

3.7.9.2 Why ROB is really needed

Modern high-performance processors use **out-of-order execution (OoO)** to improve instruction-level parallelism. However, they must still ensure **in-order commit** to preserve correct program behavior. This is where the **Reorder Buffer (ROB)** comes into play.

Why Out-of-Order Execution and In-Order Commit?

Out-of-Order execution because:

- ✖ **Dependencies** (e.g., RAW hazards) and **delays** (e.g., memory access) prevent some instructions from being executed immediately.
- ✔ OoO execution **allows the processor to bypass stalled instructions** and execute independent ones earlier.
- This helps keep functional units busy and **improves throughput**.

For example, if an instruction is waiting for a memory load, another instruction, for example an ALU op, can execute immediately.

Unfortunately, Out-of-Order execution **introduces risk**:

- ⚠ Executing instructions out of order can break the **precise exception model**.
- ⚠ Also, **speculative** instructions (e.g., those after a branch) could be **incorrect**.

To preserve **program correctness**, **instructions must commit (retire) in order**, exactly as they appear in the original program.

ROB's role in bridging OoO and In-Order Commit

The **Reorder Buffer (ROB)** is a hardware structure used in modern out-of-order processors to: **support out-of-order execution while enforcing in-order commitment of instructions**, ensuring correct architectural state and precise exception handling.

The ROB is the key mechanism enabling this balance:

- During **issue**: allocate an entry in the ROB, marking the instruction's place in program order.
- During **execution**: store the result in the ROB as soon as the instruction finishes.
- During **commit**: only commit the instruction (i.e., update architectural registers or memory) if:
  1. It is **at the head** of the ROB
  2. Its **execution has completed**

3. It is **not speculative**

This design ensures:

- ✔ Architectural **state** (register file, memory) is **updated in program order only**.
- ✔ **Speculative results** are kept in the ROB until validated.
- ✔ It is possible to **undo speculated instructions** if needed (e.g., branch misprediction).

The ROB allows a **processor to execute instructions as soon as their operands are ready**, regardless of program order, while **ensuring they commit their results in-order**.

What about Precise Exception support?

An exception is *precise* if:

1. The processor can **stop at a well-defined point**, corresponding to a specific instruction.
2. All **previous instructions are fully committed**, and
3. **No subsequent instruction** has **modified** the architectural **state**.

This allows the OS or exception handler to reliably identify and handle the error. In **out-of-order execution**, instructions complete in a different order than they appear in the program. **Without careful control** instructions *after* the faulting one could have modified registers or memory and this would **leave the system** in an **inconsistent state**.

The **Reorder Buffer (ROB) solves** this by:

- ✔ **Storing all results temporarily**: Instructions write their output to the ROB instead of the register file or memory.
- ✔ **Committing results in-order**: Only when an instruction is at the **head** of the ROB and marked **completed**, its result is written to the architectural state.
- ✔ **Rejecting speculative results**: If an exception or misprediction occurs:
  (a) The ROB **flushes all speculative entries after the faulting instructions**.
  (b) Execution restarts from the faulting instruction or the exception handler.

  This rollback is clean because the **architectural state remains untouched** beyond the last committed instruction.

Example 11: Precise Exception support

Imagine a sequence:

```
Instr 1 → OK
Instr 2 → OK
Instr 3 → Exception (e.g., divide by zero)
Instr 4 → Executed early (OoO), wrote to reg R5
```

✗ Without the ROB:

- Instruction 4 might modify R5 *before* the exception at *Instr 3* is recognized.
- This makes the exception imprecise, corrupting the state.

✓ With the ROB:

- Instruction 4's result is held in the ROB.
- Since `Instr 3` caused an exception, no later instruction commits.
- `R5` is unaffected, precise state is preserved.

Functional Roles of the ROB

1. **Result Buffering**. Holds results of instructions that:
   - Have **completed execution**.
   - But have **not yet committed** to the architectural state (e.g., register file or memory).
2. **Speculative Result Propagation**. ROB acts as a **buffer to pass results among instructions that have started speculatively after a branch**. This allows speculative instructions (e.g., those after a predicted branch) to forward their results internally via the ROB without prematurely updating architectural registers.
3. **Precise Interrupt Support**. Originally introduced in 1988, the ROB was created to:
   - Preserve the **precise interrupt model**.
   - Guarantee that **only committed instructions affect** the architectural **state**.
   - Enable **rollback on branch misprediction or exceptions** by flushing speculative ROB entries.

The ROB is not merely a commit buffer, it is also a **speculation-aware result forwarding structure**, enabling safe communication among instructions that may never actually commit.

3.7.9.3 ROB as a Data Communication Mechanism

In Tomasulo's original algorithm, **Reservation Stations (RSs)** handled register renaming and operand forwarding. However, with the introduction of the ROB, it **replaces the renaming and forwarding function of RSs**. Before we explain how, we need to understand some *basic* concepts of ROB.

What are ROB numbers?

A **ROB number** is simply an **index** (or tag, in Tomasulo's algorithm) **identifying** a specific **entry in the Reorder Buffer**. Each instruction that is issued receives a unique ROB number that corresponds to its place in the ROB, its **slot identifier** (slot ID).

The ROB number is then used in two key ways:

1. **As a destination tag (Register Renaming)**. When an **instruction** is issued and it **will write to a register**:
   - The destination register is **not renamed to another physical register** (like in a pure register renaming scheme).
   - Instead, the **architectural register is mapped to the instruction's ROB number**.

   It is the *identical* logic of tag in Tomasulo algorithm.

2. **For Operand Dependency Resolution**. Later **instructions that depend on the result** of the ROB number do not need to stall:
   - They **record the ROB number** as the operand source.
   - Once the ROB number becomes "ready" (i.e., the value is written to the ROB), dependent instructions can **read the value from the ROB number** and proceed to execution.

   In tomasulo algorithm this feature is provided by the CDB, where each instruction listens to that because it waits for the result of the tag.

   As happens in tomasulo algorithm, this mechanism:

   ✔ Avoids **WAR and WAW hazards**.
   ✔ Enables **data forwarding** even before instructions commit.

Tomasulo	ROB-based Tomasulo
Tags = RS entry IDs	Tags = ROB entry numbers
Values broadcast via CDB	Same, but identified by ROB number
RS buffers result locally	ROB stores result globally
Commit on write-back	Commit delayed and via ROB head

Table 21: Quick comparison between Tomasulo with and without ROB.

Updated Role of Reservation Stations

The ROB not only buffers instruction results but also **propagates those results to dependent instructions** as soon as they are ready, even if not yet committed. So, with the ROB managing renaming and data forwarding:

- ✗ Reservation Stations are **no longer responsible for naming/tagging**.
- ✓ Their role is now focused on:
  - **Buffering decoded instructions** before they're issued to the Functional Units (FUs);
  - **Holding operand values** (or ROB tags) temporarily;
  - Helping reduce **structural hazards**.

Reservation Stations now act like **staging areas**, not tracking results or resolving dependencies directly.

Function	Tomasulo	ROB-based Tomasulo
Register Renaming	RS	ROB
Data forwarding	RS	ROB
Instruction buffering	RS	RS (same)
Operand availability tracking	RS	RS (via ROB tags)

Table 22: Summary of architectural changes.

In modern speculative Tomasulo architectures, the ROB becomes the central structure for both result tracking and inter-instruction communication. RSs are demoted to lightweight instruction and operand buffers.

3.7.9.4 Architecture

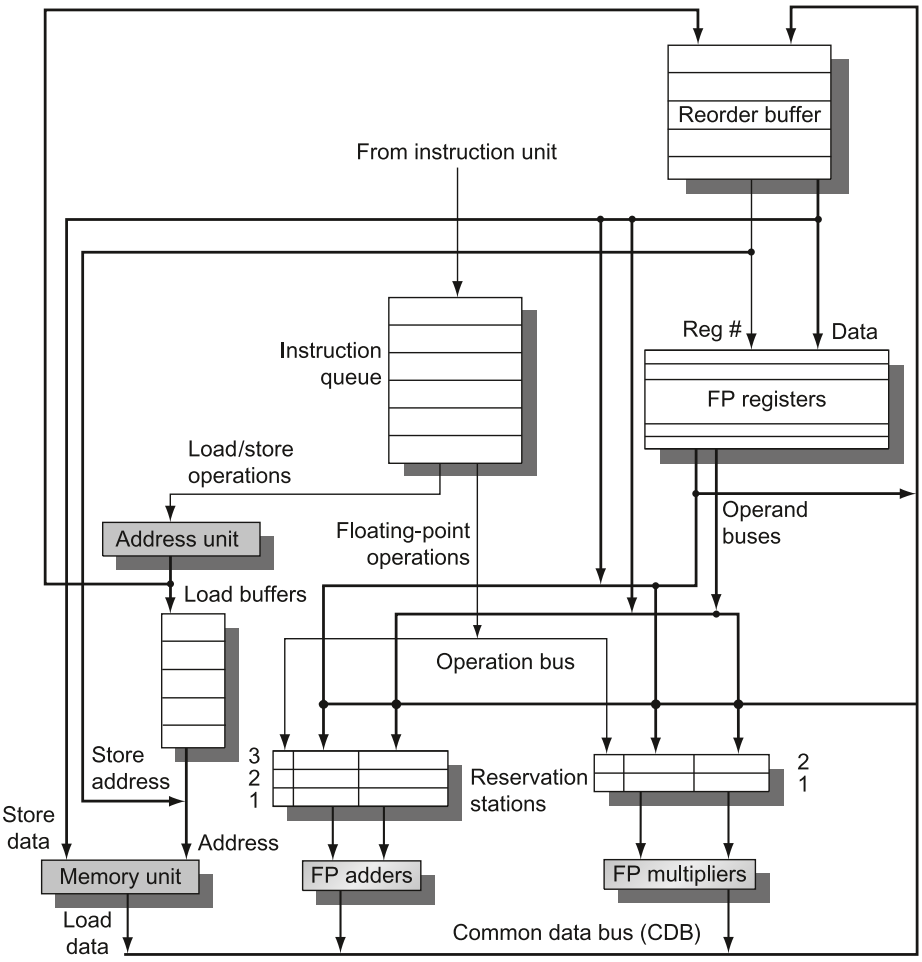


Figure 27: The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation (ROB). [2]

This architecture shows a **speculative version of Tomasulo's algorithm**, integrating a Reorder Buffer (ROB) and eliminating the classic **store buffer**. It's designed for **floating-point**, but the same design applies to general OoO pipelines.

- **Instruction Queue**. Holds fetched instructions awaiting issue. Supplies instructions to **reservation stations** and the **ROB** in parallel.
- **Reservation Stations**. Acts as **temporary buffers** between issue and execution. Each FP unit (adder, multiplier) has dedicated RSs. Stores instructions with operand tags or values. Unlike the classic Tomasulo, **renaming is handled via the ROB**.
- **Floating-Point Units**. Includes **FP Adders** and **FP Multipliers**. Execution units receive instructions from RSs when operands are ready.

- **Common Data Bus (CDB)**. Used to broadcast result values and their corresponding **ROB tag**. All waiting instructions and ROB entries **listen** to the CDB.
- **Reorder Buffer (ROB)**. Central to this architecture, replaces:
  1. Register renaming logic
  2. Store buffers

  Each ROB entry holds critical metadata about one in-flight instruction:
  - **Busy field**. Indicates whether the ROB entry is currently active (busy) or free (available).
  - **Instruction type field**. Specifies the instruction category:
    * Branch (no destination result),
    * Store (destination is memory address),
    * Load/ALU (destination is a register).
  - **Destination field**
    * For load and ALU instructions: **target register** number.
    * For store instructions: **memory address** where the value must be written.
  - **Value field**. Holds the **result** of the instruction after execution, kept until commit.
  - **Ready Bit**. Set when execution has completed and the result is valid.
  - **Speculative Flag**. Shows whether the instruction is executed speculatively (e.g., after a predicted branch) or not.
- **Load Buffers & Address Unit**.

  Load Buffers is a **queue or small table** that holds load instructions **waiting to access memory**. It **temporarily buffers** loads *after issue* and *before* they actually perform a memory access. Its purpose is shown in the example on page 188.

  Address Unit is a specialized hardware block **dedicated to calculating effective addresses** for loads and stores. Given a base register value and an offset, it computes an effective address.

  How Load Buffers and Address Unit Work Together?

  1. **Issue Stage**: the instruction is issued. Allocates an entry in the ROB and Load Buffer.
  2. **Address Calculation**: Address Unit computes the effective address.
  3. **Memory Access Decision**: if there is no preceding store to the same address (or speculation allows), the load can access memory early. If not, the load must wait until memory disambiguation clears it.
  4. **Load Execution**: read data from memory. Write the result into the ROB entry. When ready and safe, commit to the architectural register file.

- **Register File (FP Registers)**: not updated directly after execution. Instead, it is **updated only when instructions commit via the ROB**, preserving the **precise state model**.

Key Innovations in this design

- **ROB replaces store buffers**: the store buffer doesn't exist anymore, now memory writes are delayed until commit.
- **Register renaming is moved from RSs to ROB**: simplifies dependency tracking.
- **Precise exceptions and speculative execution are both supported** via ROB tracking and flushing.
- **CDB remains critical** for result forwarding and readiness detection.

Ready Bit

The **Ready Bit** is a control bit that indicates whether the instruction associated with this ROB entry has **completed execution** and its **result is available**. It is **set to true** when:

1. The instruction finishes execution.
2. The result is written into the ROB.

Its purpose is to **inform dependent instructions that they can now read this value** (via the CDB or directly from the ROB). Also, it enables commit: an instruction can **only commit if its ready bit is set**.

In other words, the ready bit **tracks the availability of the computed result**.

Speculative Flag

The **Speculative Flag** is a control bit that indicates whether the **instruction was issued after an unresolved branch prediction** (or other speculative control decision). It is:

- **Set to true** only when the instruction is **issued speculatively**, before the control flow is confirmed.
- **Cleared** when the speculation is resolved (e.g., branch prediction validated).

Its purpose is to ensure that **speculative instructions do not prematurely update** the architectural state. Allows all speculative instructions to be **flushed** upon misprediction or exception.

In other words, the speculative flag tracks whether the instruction is **tentative** or **safe**.

ROB Structure and Operation: Circular Buffer

The Reorder Buffer (ROB) is implemented as a **circular buffer** (circular FIFO, First-In-First-Out, queue), managed with **two pointers**:

- **Head pointer**. Points to the **oldest instruction** that is **next to commit** (i.e., retire).

  The head **advances** when instructions commit.

- **Tail pointer**. Points to the **next free entry**, where a new issued instruction will be inserted.

  The tails **advances** when new instructions are issued.

Example 12: ROB Circular Buffer

ROB#	BUSY	INSTR. TYPE	READY	DEST	VALUE	SPEC
ROB0	Yes	In	No	F0	–	No
ROB1	No	–	–	–	–	–
ROB2	No	–	–	–	–	–
ROB3	No	–	–	–	–	–

- **Head** points at ROB0 (first to commit once ready).
- **Tail** points at ROB1 (next free entry for a new instruction).

Why Two Pointers?

The head and tail pointers allow the ROB to **efficiently manage** dynamic instruction issue and commit while **preserving program order**, and they **minimize hardware complexity** by avoiding costly entry movement.

1. **Sequential Insertion at the Tail (Issue Stage)**. When a new instruction issues, it needs a free ROB entry. The tail pointer indicates where to insert the new instruction. After insertion, the tail advances to the next free slot. This preserves the program order at issue time.

   New instructions always go at the tail.

2. **Sequential Retirement from the Head (Commit Stage)**. When an instruction completes and satisfies all commit conditions (ready, not speculative, prior instructions retired), it retires. The head pointer shows which instruction should commit next. After commit, the head advances to the next oldest instruction.

   Commit always happens starting from the head, ensuring in-order commit.

3. **Circular Buffer Efficiency**. Memory and hardware are limited: we don't want an infinitely growing ROB. Using a circular buffer:

   - When the tail reaches the end of the buffer, it wraps around to position 0.
   - Same for the head.

   This reuses space efficiently without needing to shift entries manually.

   Circular structure avoids expensive data movement and saves silicon area.

4. **Detecting Full and Empty Conditions**. With just head and tail:

   - If head = tail and entry busy → ROB is full (cannot issue more instructions).
   - If head = tail and entry not busy → ROB is empty (no instructions to retire).

   Simple hardware checks based on two pointers.

Example 13: ROB and Rename Table

The **Rename Table** is a small **hardware table** that records **where the *most recent* value** of each register **will come from**, **while the instruction is still speculative or not yet committed**. It connects logical (architecture) registers (like `F0`, `F2`, etc.) to ROB entries that are producing their updated value.

It works like this:

- When an instruction is issued that **writes** a register:
  - The Rename Table **maps** that register to its **new ROB number**.
- When an instruction needs to **read** a register:
  - It **checks the Rename Table**:
    - If an ROB entry is mapped: the instruction reads the pending value from the ROB.
    - If no mapping: the instruction reads from the architectural register file.

Now we present an example using the ROB and the Rename Table. At the beginning of this example, we have a ROB almost empty and a Rename Table partially filled.

1. **Cycle 1**, first step.

ROB#	BUSY	INSTR. TYPE	READY	DEST	VALUE	SPEC
ROB0	Yes	In	No	F0	–	No
ROB1	Yes	In+1	No	F2	–	No
ROB2	No	–	–	–	–	–
ROB3	No	–	–	–	–	–

Head is at ROB0 and Tail is at ROB2.

Two instructions were issued:

- In writes to F0, mapped to ROB0.
- In+1 writes to F2, mapped to ROB1.

Both are non-speculative (SPEC = No)

Register	F0	F2	F4	F6	F8
Pointer	ROB0	ROB1			

F0 and F2 are now renamed: if anyone uses F0/F2, they will look at the respective ROB entry, not the old register.

2. **Cycle 2**, second step.

ROB#	BUSY	INSTR. TYPE	READY	DEST	VALUE	SPEC
ROB0	Yes	In	No	F0	–	No
ROB1	Yes	In+1	No	F2	–	No
ROB2	Yes	In+2	No	F4	–	No
ROB3	No	–	–	–	–	–

Head is still at ROB0 and Tail is now at ROB3.

Another instruction issued:

- In+2 writes to F4, mapped to ROB2.

Still non-speculative.

Register	F0	F2	F4	F6	F8
Pointer	ROB0	ROB1	ROB2		

3. **Cycle 3**, speculative instruction issued.

ROB#	BUSY	INSTR. TYPE	READY	DEST	VALUE	SPEC
ROB0	Yes	In	No	F0	–	No
ROB1	Yes	In+1	No	F2	–	No
ROB2	Yes	In+2	No	F4	–	No
ROB3	Yes	In+3	No	F6	–	Yes

Head is still at ROB0. As for Tail, since all four entries (ROB0 through ROB3) are occupied and there are no free entries, Tail cannot move forward until there is a free ROB entry. Tail must logically "point" to the next free entry, but since there is none available, it "hovers" behind or at ROB0, waiting for a free entry after a commit. So Tail is *stalled* at ROB0 because the ROB is full.

A new instruction (In+3) was issued and allocated into ROB3. In+3 writes to register F6. The speculative bit is set to "Yes", so this instruction depends on an unresolved branch (it's speculative).

Register	F0	F2	F4	F6	F8
Pointer	ROB0	ROB1	ROB2	ROB3	

4. **Cycle 4**, first execution result ready.

ROB#	BUSY	INSTR. TYPE	READY	DEST	VALUE	SPEC
ROB0	Yes	In	Yes	F0	10	No
ROB1	Yes	In+1	No	F2	–	No
ROB2	Yes	In+2	No	F4	–	No
ROB3	Yes	In+3	No	F6	–	Yes

Head is still at ROB0. Tail is still waiting.

The instruction at ROB0 (In) **completed execution**: it is ready (Yes) and the result is 10. **But it has not yet committed** because commits happen separately.

The value 10 is still in the ROB, not yet in the register file. F0 is still mapped to ROB0 (waiting commit).

Register	F0	F2	F4	F6	F8
Pointer	ROB0	ROB1	ROB2	ROB3	

5. **Cycle 5**, first commit happens.

ROB#	BUSY	INSTR. TYPE	READY	DEST	VALUE	SPEC
ROB0	No	–	–	–	–	–
ROB1	Yes	In+1	No	F2	–	No
ROB2	Yes	In+2	No	F4	–	No
ROB3	Yes	In+3	No	F6	–	Yes

Head is now at ROB1. Tail finally finds an empty entry and points to it (ROB0).

Instruction In (ROB0) **committed**:

- ✔ It was ready (Ready = Yes);
- ✔ Not speculative (Spec = No);
- ✔ Head pointer advanced from ROB0 → ROB1.

Register File updated:

- F0 now officially = 10 (not pending anymore).

Rename Table updated:

- F0 no longer points to a ROB entry, but to the **final committed value** (10).

Register	F0	F2	F4	F6	F8
Pointer	10	ROB1	ROB2	ROB3	

6. **Cycle 6**, new instructions, partial execution.

ROB#	BUSY	INSTR. TYPE	READY	DEST	VALUE	SPEC
ROB0	Yes	In+4	No	F8	–	Yes
ROB1	Yes	In+1	Yes	F2	20	No
ROB2	Yes	In+2	No	F4	–	No
ROB3	Yes	In+3	No	F6	–	Yes

Head is still at ROB1. Tail stalled because ROB is full.

A new instruction (In+4) was issued and allocated to ROB0:

- Destination: F8
- Marked speculative (SPEC = Yes)

Meanwhile the instruction In+1 (in ROB1) has **completed execution** (READY = Yes) with the value 20.

Register	F0	F2	F4	F6	F8
Pointer	10	ROB1	ROB2	ROB3	ROB0

7. **Cycle 7**, commit of a non-speculative instruction.

ROB#	BUSY	INSTR. TYPE	READY	DEST	VALUE	SPEC
ROB0	Yes	In+4	No	F8	–	Yes
ROB1	No	–	–	–	–	–
ROB2	Yes	In+2	No	F4	–	No
ROB3	Yes	In+3	No	F6	–	Yes

Head is now at ROB2. Tail is now at ROB1.

Instruction In+1 in ROB1 has committed, and its destination register F2 now holds the value 20. ROB1 is now free (BUSY = No) and the head moves forward to ROB2.

Register	F0	F2	F4	F6	F8
Pointer	10	20	ROB2	ROB3	ROB0

8. **Cycle 8**, misprediction.

ROB#	BUSY	INSTR. TYPE	READY	DEST	VALUE	SPEC
ROB0	Yes	In+4	No	F8	–	Yes
ROB1	No	–	–	–	–	–
ROB2	Yes	In+2	No	F4	–	No
ROB3	Yes	In+3	No	F6	–	Yes

Head is still at ROB2. Tail is still at ROB1.

ROB0 and ROB3 still contain **speculative instructions** waiting for execution. ROB2 contains non-speculative instruction but not ready yet.

At some point, **misprediction detected**! The branch speculation fails and all speculative instructions must be **flushed** (discarded).

Register	F0	F2	F4	F6	F8
Pointer	10	20	ROB2	ROB3	ROB0

8. **Cycle 8**, after misprediction flush.

ROB#	BUSY	INSTR. TYPE	READY	DEST	VALUE	SPEC
ROB0	No	–	–	–	–	–
ROB1	No	–	–	–	–	–
ROB2	Yes	In+2	No	F4	–	No
ROB3	No	–	–	–	–	–

Head is still at ROB2. Tail is moved to ROB3 because after a misprediction, the **tail must move to the next free ROB entry *after* the last non-speculative instruction**, maintaining strict program order.

ROB0 (In+4) and ROB3 (In+3) were speculative → flushed immediately. Only ROB2 remains:

- It is non-speculative (safe);
- Still waiting to execute or ready.

Register	F0	F2	F4	F6	F8
Pointer	10	20	ROB2	ROB3 (*obsolete*)	ROB0 (*obsolete*)

3.7.9.5 Speculative Tomasulo Algorithm with ROB

The speculative Tomasulo architecture extends the original Tomasulo's algorithm by integrating a Reorder Buffer (ROB). This allows:

- ✔ Dynamic register renaming using ROB entries;
- ✔ In-order commit;
- ✔ Precise exception handling;
- ✔ Efficient speculative execution with safe rollback.

Main Innovations Introduced by the ROB

1. **Pointers towards ROB Entries**. Operands are no longer tracked via Reservation Stations. Instead, **operands are identified by ROB numbers**. This ensures centralized tracking of instruction dependencies.
2. **Implicit Register Renaming**. When a destination register is renamed:
   - It is mapped to a **ROB entry**;
   - This removes **WAR** and **WAW hazards** automatically;
   - Enables **dynamic loop unrolling** without conflicts.
3. **Delayed Update of Architectural State**. Registers and memory are **not updated at execution time**. They are updated **only when the instruction is at the head of the ROB and is ready to commit**. This guarantees correct program state even in speculative execution.
4. **Easy Speculation Management**. By holding all results inside the ROB until commit:
   - **Mispredictions** can be handled by **flushing speculative entries**;
   - **Precise exceptions** are maintained, ensuring a clean rollback mechanism.

Operating phases of the Speculative Tomasulo Algorithm

The execution flow is divided into four main stages:

1. **Issue: Fetch and Prepare the Instruction**

   The **Issue phase** is the step where an instruction is fetched from the **Instruction Queue** and prepared for execution. In this phase:
   - The processor checks if there are **resources available** to allow the instruction to proceed.
   - It allocates both a **Reservation Station (RS) slot** and a **ROB entry**.

   Only if both are available, the instruction can continue.

(a) **Check for Available Slots**. The issue logic checks:

- *Is there a free Reservation Stations?*
- *Is there a free ROB entry?*

If yes, the instruction is issued into the pipeline. **If no**, the **instruction stalls** in the Instruction Queue and waits.

(b) **Fetch Operands**. Once issued, the instruction's **source operands are fetched**. If an operand is:

- **Ready in the Register File** → send it immediately to the Reservation Station.
- **Pending in the ROB** → send the ROB number (tag) to the Reservation Station instead (to wait for the value).

Thus, the RS either: receives the actual operand value (if available), or a **tag** (ROB number) pointing to where the value will appear later.

(c) **Allocate ROB Entry for the Result**. A **ROB entry** is assigned to the instruction. The **ROB number** is sent to the Reservation Station. This number will later be used to **tag the result on the Common Data Bus (CDB)** after execution. This ROB number replaces the register name during speculative execution.

⚠ **What Happens If No Space?** If either no RS slot is free, or no ROB entry is free, then the instruction **cannot issue** and **must stall**. This ensures the system **never overflows** the ROB or Reservation Stations, maintaining control over execution resources.

2. **Execution Started**

After the instruction is successfully issued and resides in a **Reservation Station (RS)**, the next step is **starting execution**. Execution can start only when **all required operands are available**.

(a) **Check Operand Availability**. The Reservation Station **monitors the availability** of operands. If **both operands are available** (i.e., no more pending data hazards):

- Instruction **immediately starts execution** on its operands.
- This stage is called **EX** (Execute).

RAW hazards (Read After Write) must already be solved to start execution.

(b) **Monitoring for Pending Operands**. If one or more operands are **not yet ready** (e.g., still waiting for a previous instruction to produce them):

- The RS **listens** to the **Common Data Bus (CDB)**.
- As soon as the missing operand appears on the CDB, it captures it and becomes ready.

Dynamic readiness: the RS waits without blocking the pipeline.

(c) **Special Case: Store Instructions**. For **store** instructions:

- Only the **base register** (address computation) must be available to start execution.
- At this point, the **effective memory address** is computed.

The store value itself can still arrive later, closer to the commit phase.

Instructions don't wait statically, they dynamically listen and proceed as soon as they can, maximizing execution overlap.

3. **Execution Completed & Write Result**

Once the execution of the instruction finishes, the result must be **broadcast** and **stored** so that dependent instructions can proceed and the instruction can eventually commit.

(a) **Broadcast the Result on the Common Data Bus (CDB)**. The completed instruction **places its result** on the **Common Data Bus**. This **broadcasts** the result to:

- All **Reservation Stations** that are waiting for it;
- The corresponding **ROB entry** (updating its `Value` field).

Any instruction that was stalled waiting for this operand can now proceed.

(b) **Update the ROB Entry**. The `Value` field of the instruction's **ROB entry** is updated with the computed result. The instruction's **Ready bit** inside the ROB is set to **true**, meaning the instruction is now completed and awaiting commit.

The ROB now safely holds the completed result, ready for in-order retirement.

(c) **Release the Reservation Station**. The Reservation Station that held this instruction is now **marked as free**. It becomes available for **issuing new instructions**.

This keeps the hardware resource usage efficient and avoids blocking incoming instructions.

(d) **Special Case: Store Instructions**. For **store** operations:

- The **data to be stored** (not just the address) is written into the ROB's `Value` field.
- If the value wasn't ready at the start of the store's execution, the RS monitors the CDB until the data is captured.

This ensures the store operation has both address and data ready by the time it needs to commit.

Results are not immediately written to registers; they are buffered safely into the ROB and forwarded to dependent instructions dynamically.

4. **Commit: Updating the Architectural State**

   The **Commit phase** is when the result of an instruction is **safely written into the Register File or memory**, and the instruction is **retired from the ROB**. Commit happens strictly **in program order**:

   ✔ Only the instruction at the **head of the ROB** can be considered for commit.

   ✔ The instruction must have **finished execution** and have its **result ready**.

   (a) **Normal Commit**. If the instruction at the head is **completed** (`Ready bit = true`):

   - Its **result is written into the Register File** (if it's a computational instruction);
   - The **ROB entry is freed**.

   The architectural state is updated in strict program order.

   (b) **Store Commit**. If the instruction at the head is a **store**:

   - Instead of updating the Register File, it **writes the data into memory**;
   - Afterward, the instruction is removed from the ROB.

   Memory operations happen precisely at the correct point, preserving correct memory ordering.

   (c) **Branch Commit and Misprediction Handling**. If the instruction at the head is a **branch**:

   ✔ If the branch was **predicted correctly**, it is committed normally, no special action.

   ✘ If the branch was **mispredicted**:

   – The **ROB is flushed**, all speculative instructions after the branch are discarded.

   – Execution **restarts** at the correct successor (true path).

   Mispredictions are corrected cleanly, and speculative damage is avoided.

   This flushing process is sometimes called **graduation** in technical literature[7].

   Only when the instruction is at the ROB head, completed, and non-speculative, it can safely update the real architectural state (register file or memory). This phase ensures that the processor maintains a **precise state** even under heavy speculation and out-of-order execution.

[7] **Graduation** is the final commit of an instruction from the ROB into the architectural state (registers or memory), happening in program order after execution is completed and speculation has been validated. In literature like *Computer Architecture: A Quantitative Approach* [2], graduation and commit are used interchangeably.

In other words, it is another technical term for instruction commit in speculative out-of-order processors.

Example 14: Speculative Tomasulo Algorithm with ROB - Execution Overlap

We are executing this loop:

```
Loop:
        LD     F0, 0(R1)
        MULTD  F4, F0, F2   # RAW hazard on F0
        SD     F4, 0(R1)    # RAW hazard on F4
        SUBI   R1, R1, #8
        BNEZ   R1, Loop     # RAW and WAR hazards on R1
```

- `LD` loads from memory address pointed by `R1` into `F0`.
- `MULTD` depends on `F0` being ready → must wait for the `LD`.
- `SD` depends on `F4` → must wait for `MULTD`.
- `SUBI` modifies `R1` (address computation for next iteration).
- `BNEZ` depends on the result of `SUBI` to decide whether to branch.
- **First iteration issued**

ROB#	BUSY	INSTR. TYPE	READY	DEST.	VALUE	SPEC.
ROB0	Yes	LD F0, 0(R1)	No (exec. start)	F0	–	No
ROB1	Yes	MULTD F4, F0, F2	No (issued)	F4	–	No
ROB2	Yes	SD F4, 0(R1)	No (issued)	M[0+[R1]]	–	No
ROB3	Yes	SUBI R1, R1, #8	No (issued)	R1	–	No
ROB4	Yes	BNEZ R1, Loop	No (issued)	–	–	No
ROB5	No					
ROB6	No					
ROB7	No					

The head is at `ROB0` and the tail is at `ROB5`.

Register #	F0	F2	F4	F6	F8	F10
Pointer	ROB0		ROB1			

The actions are:

- All these 5 instructions are issued one after another.
- Each gets a **ROB entry**.
- The **Rename Table** is updated:
  - `F0` is renamed to `ROB0`;
  - `F4` to `ROB1`;
  - Memory address for `SD` points to `ROB2`;
  - `R1` to `ROB3` (because `SUBI` modifies `R1`).

ROB entries are allocated from `ROB0` to `ROB4`.

- **After Some Cycles (ROB Full)**

ROB#	BUSY	INSTR. TYPE	READY	DEST.	VALUE	SPEC.
ROB0	Yes	LD F0, 0(R1)	Ready to Commit	F0	M[0+[R1]]	No
ROB1	Yes	MULTD F4, F0, F2	No (issued)	F4	–	No
ROB2	Yes	SD F4, 0(R1)	No (issued)	M[0+[R1]]	–	No
ROB3	Yes	SUBI R1, R1, #8	Exec. Started	R1	–	No
ROB4	Yes	BNEZ R1, Loop	No (issued)	–	–	No
ROB5	Yes	LD F0, 0(R1)	No (issued)	F0	–	Yes
ROB6	Yes	MULTD F4, F0, F2	No (issued)	F4	–	Yes
ROB7	Yes	SD F4, 0(R1)	No (issued)	M[0+[R1]]	–	Yes

The head is still at ROB0 and the tail is stall.

Register #	F0	F2	F4	F6	F8	F10
Pointer	ROB5		ROB6			

Now, after some execution, the situation is:

- **ROB is full**: we used all 8 entries (ROB0 to ROB7).
- Meanwhile:
  - LD (ROB0) has completed (Ready = Yes).
  - MULTD (ROB1) is still waiting to execute (waiting on F0).
  - SUBI (ROB3) has started executing (updating R1).
- Second iteration of the loop started speculative issuing:
  - A new LD into F0 (ROB5);
  - A new MULTD into F4 (ROB6);
  - A new SD (ROB7).

Speculative entries being appearing (ROB5-ROB7 marked speculative). New renamings happen:

- F0 is now renamed to ROB5 (overriding previous ROB0).
- F4 is renamed to ROB6.

The processor allows speculative execution past the BNEZ, but only under the assumption that the branch prediction was correct.

- **First Commit and Progress**

ROB#	BUSY	INSTR. TYPE	READY	DEST.	VALUE	SPEC.
ROB0	No					
ROB1	Yes	MULTD F4, F0, F2	Exec. Started	F4	*	No
ROB2	Yes	SD F4, 0(R1)	No (issued)	M[0+[R1]]	–	No
ROB3	Yes	SUBI R1, R1, #8	Exec. Started	R1	–	No
ROB4	Yes	BNEZ R1, Loop	No (issued)	–	–	No
ROB5	Yes	LD F0, 0(R1)	No (issued)	F0	–	Yes
ROB6	Yes	MULTD F4, F0, F2	No (issued)	F4	–	Yes
ROB7	Yes	SD F4, 0(R1)	No (issued)	M[0+[R1]]	–	Yes

*: the value M[0+[R1]]*F2 is under computation. The head is at ROB1 and the tail is at ROB0.

Next cycle:

- ROB0 commits:
  - LD completed → value is written to the Register File (F0 updated).
  - ROB0 is freed (Busy = No).
- MULTD at ROB1 finally starts execution (*): now that F0 is available, it can compute F4 = F0 × F2.
- SUBI at ROB3 also continues execution.

Instructions are allowed to move forward even if speculative ones are pending behind.

- **ROB Remains Full (and Critical Situation)**

ROB#	BUSY	INSTR. TYPE	READY	DEST.	VALUE	SPEC.
ROB0	Yes	SUBI R1, R1, #8	No (issued)	R1	–	Yes
ROB1	Yes	MULTD F4, F0, F2	Exec. Started	F4	*	No
ROB2	Yes	SD F4, 0(R1)	No (issued)	M[0+[R1]]	–	No
ROB3	Yes	SUBI R1, R1, #8	Exec. Started	R1	–	No
ROB4	Yes	BNEZ R1, Loop	No (issued)	–	–	No
ROB5	Yes	LD F0, 0(R1)	No (issued)	F0	–	Yes
ROB6	Yes	MULTD F4, F0, F2	No (issued)	F4	–	Yes
ROB7	Yes	SD F4, 0(R1)	No (issued)	M[0+[R1]]	–	Yes

*: the value M[0+[R1]]*F2 is still under computation. The head is still at ROB1 and the tail is stall.

The situation is:

- ROB remains fully occupied because new speculative instructions filled it (ROB5 to ROB7).

- The ongoing speculative execution **relies heavily** on correct branch prediction (`BNEZ`).
- **Problem**: If the branch was **mispredicted**, all speculative instructions (`ROB5`, `ROB6`, `ROB7`) must be **flushed**.

This example shows how the ROB manages **execution overlap**, **hazard resolution**, **speculation**, and **in-order commit** even under complex dependencies.

4 Performance Evaluation

All formulas and metrics can be found in the Index section, page 420, under the word F (Formula).

4.1 Basic Concepts and Performance Metrics

Performance can be considered from two key viewpoints:

- **Purchasing Perspective**. Given multiple machines, the goal is to determine:
  - *Which machine has the **best performance**.*
  - *Which machine has the **lowest cost**.*
  - *Which machine offers the **best performance-to-cost ratio**.*
- **Design Perspective**. When facing different architectural or technological options, the aim is to identify:
  - *Which design achieves the **greatest performance improvement**.*
  - *Which design **minimizes cost**.*
  - *Which design **optimize the performance-to-cost balance**.*

Both perspective require a **basic for comparison** and **clear performance metrics**. The main goal is to understand how architectural choices impact both **performance** and **cost**.

Two Fundamental Notions of Performance

1. **Execution Time (Latency, Response Time)**: refers to the **time needed to complete a task**.

   Lower execution time = better.

2. **Throughput (Bandwidth)**: measures the **number of tasks completed per unit of time** (e.g., jobs per hour).

   Higher throughput = better.

Response time and throughput often **oppose** each other. Optimizing for one can degrade the other.

Example 1: Planes Comparison, Concorde vs Boeing 747

Concorde is a supersonic airliner.

- **Speed**: Concorde is faster (1350 mph vs 610 mph).
- **Execution Time**: Concorde takes less time to fly Washington DC to Paris (3h vs 6.5h).
- **Throughput**: Boeing transports more passengers per mile (higher "passenger-miles per hour").

Basic Definitions and Calculations

- **Performance Formula**

$$\text{Performance} = \frac{1}{\text{Execution Time}} \tag{10}$$

- **Relative Performance**. If X is $n\%$ faster than Y, then:

$$\text{Performance}\,(X) = \left(1 + \frac{n}{100}\right) \times \text{Performance}\,(Y) \tag{11}$$

Or:

$$\text{Execution Time}\,(Y) = \left(1 + \frac{n}{100}\right) \times \text{Execution Time}\,(X) \tag{12}$$

Example 2

If machine A executes a program in 10 sec and machine B executes same program in 15 sec: A is 50% faster than B or A is 33% faster than B? Using equation 12, if A is 50% faster than B, then it should be true that:

$$\begin{aligned}
\text{Execution Time}\,(B) &= \left(1 + \frac{n}{100}\right) \times \text{Execution Time}\,(A) \\
15 &= \left(1 + \frac{50\%}{100}\right) \times 10 \\
15 &= \frac{3}{2} \times 10 \\
15 &= \frac{30}{2} \checkmark
\end{aligned}$$

Clock Cycles and Clock Frequency

Modern processors operate by executing instructions synchronously according to a **clock signal**:

- **Clock Cycle Time T_{CLK}**: the **duration** of a single clock cycle, measured in seconds (s).
- **Clock Frequency f_{CLK}**: the **number of clock cycles per second**, measured in Hertz (Hz). Defined as:

$$f_{\text{CLK}} = \frac{1}{T_{\text{CLK}}} \tag{13}$$

Some examples:

- $f_{\text{CLK}} = 500\,\text{MHz} \Leftrightarrow T_{\text{CLK}} = \dfrac{1}{500 \times 10^6} = \dfrac{1}{500'000'000} = 2\,\text{ns}$
- $f_{\text{CLK}} = 1\,\text{GHz} \Leftrightarrow T_{\text{CLK}} = \dfrac{1}{1 \times 10^9} = \dfrac{1}{1'000'000'000} = 1\,\text{ns}$

Execution Time (CPU Time)

The **execution time** (also called **CPU time**) is the **total time** needed by a processor to complete a given program. Two main ways to compute CPU time:

- **Basic form**:

$$\text{CPU Time} = \text{Clock Cycles} \times T_{\text{CLK}} = \frac{\text{Clock Cycles}}{f_{\text{CLK}}} \tag{14}$$

- **Expanded form based on instructions**:

$$\begin{aligned} \text{CPU Time} &= \text{Instruction Count} \times \text{Cycles Per Instruction} \times T_{CLK} \\ &= \frac{\text{Instruction Count} \times \text{Cycles Per Instruction}}{f_{\text{CLK}}} \end{aligned} \tag{15}$$

- **Weighted CPU Time**. If the program is composed of different instruction types, with different CPI values, the **total CPU time must be weighted** by how much each instruction type contributes:

$$\text{CPU Time} = \left(\sum_{i=1}^{n} (\text{CPI}_i \times \text{I}_i) \right) \times T_{CLK} \tag{16}$$

  Where:

  - I_i is the number of instructions of type i.
  - T_{CLK} is the clock cycle time.

  The total CPU time is the sum of the CPU times **for each class of instruction**, weighted by how many times each instruction appears.

  Note that weighted CPU time can be calculated using the classic formula, but **using a weighted CPI** (shown on page 243):

$$\text{CPU Time} = \frac{\text{IC} \times \text{CPI Weighted}}{f_{\text{CLK}}}$$

Cycles Per Instruction (CPI) and Instructions Per Cycle (IPC)

- **Clocks Per Instruction (CPI)** (clock cycles per instruction or clocks per instruction) is the **average number of clock cycles** each instruction requires.

$$\text{CPI} = \frac{\text{Total Clock Cycles}}{\text{Instruction Count } (\texttt{IC})} \tag{17}$$

  Where the Total CLock Cycles is:

$$\text{Total Clock Cycles} = \texttt{IC} + 4 + \text{Stall Cycles}$$

  And the Instruction Count is the total number of instructions executed.

- **Instructions Per Cycle (IPC)** is the reciprocal of CPI:

$$\text{IPC} = \frac{1}{\text{CPI}} \tag{18}$$

- **Mixed Programs**. In a real program, we have a mix of operations. Each instruction type can have a different CPI, thus the **total average performance depends on**:

  1. **How often** each instruction type appears (its **frequency**).
  2. **How many cycles** each type needs (its **CPI**).

  Because **not all instructions appear equally**, we must give **more weight** to instructions that appear more frequently. The **Weighted Average CPI** formula is:

$$\text{CPI} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{F}_i) \tag{19}$$

  Where:

  – CPI_i is the CPI of instruction type i.
  – F_i is the **frequency of instruction** type i (percentage). It is calculated as:

$$\text{F}_i = \frac{\text{I}_i}{\text{IC}} \tag{20}$$

    Where:

    * I_i is the number of instructions of type i.
    * IC is the total **Instruction Count** (total number of instructions in the program).

Example 3: CPI and CPU Time Calculation

Given a program with **100 instructions** and the following instruction mix (500 MHz clock frequency):

Instruction Type	Frequency	CPI
ALU	43%	1
Load	21%	4
Store	12%	4
Branch	12%	2
Jump	12%	2

First, convert frequencies to decimals:

- 43% → 0.43
- 21% → 0.21
- 12% → 0.12
- 12% → 0.12
- 12% → 0.12

Now apply the CPI weighted formula 19:

$$\begin{aligned} \text{CPI} &= \sum_{i=1}^{n} (\text{CPI}_i \times \text{F}_i) \\ &= (0.43 \cdot 1) + (0.21 \cdot 4) + (0.12 \cdot 4) + (0.12 \cdot 2) + (0.12 \cdot 2) \\ &= 0.43 + 0.84 + 0.48 + 0.24 + 0.24 \\ &= 2.23 \end{aligned}$$

Thus, the **average CPI** is 2.23. For CPU time, we can use the weighted CPI, but we must calculate the Clock Cycle Time T_{CLK}:

$$\begin{aligned} \text{CPU Time} &= \text{IC} \times \text{CPI Weighted} \times T_{\text{CLK}} \\ &= 100 \times 2.23 \times \frac{1}{(500 \times 10^6)} \\ &= 223 \times \frac{1}{(500 \times 10^6)} \\ &= 223 \times 0.000'000'002\,\text{sec} \\ &= 223 \times 2\,\text{ns} \\ &= 446\,\text{ns} \end{aligned}$$

MIPS (Millions of Instructions Per Second)

MIPS measures how many millions of instructions a processor can execute per second. Two equivalent formulas:

- **Using frequency and CPI**:

$$\text{MIPS} = \frac{f_{\text{CLK}}}{\text{CPI} \times 10^6} \tag{21}$$

- **Using instruction count and execution time**:

$$\text{MIPS} = \frac{\text{Instruction Count}}{\text{Execution Time} \times 10^6} \tag{22}$$

  Where the Execution Time is the CPU Time.

Higher MIPS does not necessarily mean better performance, because MIPS does not account for instruction complexity or actual work done.

4.2 Amdahl's Law

Amdahl's Law describes the **limits** of performance improvement when only **part of a system** is enhanced. Key principles are:

- **Speedup** of a program is **limited** by the part of the program that **cannot be improved**.
- **Making the common case fast** gives the best returns.

Thus, **no matter how much we optimize a part of a system**, the total speedup is **bounded**.

Formal Definition

Let:

- F: fraction of execution time **affected** by the enhancement.
- S: speedup **factor** for the enhanced part.

Then:

- **New Execution Time**:

$$\text{ExTime(w/ E)} = \left((1-F) + \frac{F}{S}\right) \times \text{ExTime(w/o E)} \tag{23}$$

  Where:

  - "w/ E" = "*with Enhancement*"
  - "w/o E" = "*without Enhancement*"

- **Overall Speedup**:

$$\text{Speedup(E)} = \frac{1}{(1-F) + \frac{F}{S}} \tag{24}$$

  Where:

  - $(1-F)$ is a portion of execution time **not improved**.
  - $\frac{F}{S}$ is the improved portion, now **accelerated**.

In many books, we also see:

- **FractionE**: fraction of computation time accelerated.
- **SpeedupE**: speedup factor for the enhanced portion.

Thus:

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1-\text{FractionE}) + \frac{\text{FractionE}}{\text{SpeedupE}}} \tag{25}$$

Example 4: Amdahl's Law

Let us consider an enhancement for a CPU resulting 10 time faster on computation than the original one but the original CPU is busy with computation only 40% of the time. What is the overall speedup gained by introducing the enhancement?

Given:

- FractionE: 0.4 (40% of the time can be accelerated).
- SpeedupE: 10 (enhancement is 10x faster)

Apply Amdahl's Law:

$$\begin{aligned} \text{Speedup}_{\text{overall}} &= \frac{1}{(1-0.4)+\frac{0.4}{10}} \\ &= \frac{1}{0.6+0.04} \\ &= \frac{1}{0.64} \\ &= 1.56 \end{aligned}$$

Thus, even if the improvement is 10x, the overall speedup is only about 1.56x.

If we **only speed up a small part** (small F), the **impact is small**. To achieve **large speedup**, we must:

- Speed up a **large fraction** of the execution time.
- Or make the enhancement extremely fast ($S \to \infty$).

To reach the maximum theoretical speedup, if $S \to \infty$ (perfect acceleration), then:

$$\text{Speedup}_{\text{max}} = \frac{1}{1-F} \tag{26}$$

For example if only 40% can be improved, even with infinite speedup, the maximum is:

$$\text{Speedup}_{\text{max}} = \frac{1}{1-0.4} = \frac{1}{0.6} \approx 1.67$$

This shows **the real bottleneck**: the non-improved part.

4.3 Pipelined Processors

Pipelining affects CPU performance **differently** than simple frequency or instruction count.

- ✔ **Pipelining increases instruction Throughput** (number of instructions completed per unit time).
- ✖ **Pipelining does not reduce Execution Time** (latency) of a single instruction.

In fact slight latency **increase** can happen due to:

- **Imbalance** among **pipeline stages** (some stages are slower, limiting the clock frequency).
- **Overhead** from **pipeline control** (registers, clock skew, pipeline latch delays).

Also, we must remember that the **clock cycle time must be long enough for the slowest stage to finish**. Otherwise, the slowest stage will not finish its work before the clock ticks again, and the pipeline will become inconsistent or crash. So:

$$T_{\text{CLK}} > \text{time of the slowest stage}$$

So, if we **want to make the CPU faster**, we must **balance the stages** (make all stages take about the same time). Otherwise, one slow stage will limit the entire clock speed, even if other stages are very fast. This is why sometimes CPU designers split a slow stage into two smaller stages to balance the pipeline better!

☰ Performance Metrics for Pipelining

Define:

- IC as Instruction Count
- CPI as Cycles Per Instruction
- IPC as Instructions Per Clock $\frac{1}{\text{CPI}}$

We have:

- **Number of clock cycles in pipelined execution**:

$$\text{Clock Cycles} = \text{IC} + \text{Stall Cycles} + \text{Pipeline fill penalty (4 cycles)} \quad (27)$$

For a 5-stage pipeline like MIPS or RISC-V. And the CPI is:

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{IC}} = \frac{(\text{IC} + \text{Stall Cycles} + 4)}{\text{IC}} \quad (28)$$

- **MIPS calculation**:

$$\text{MIPS} = \frac{f_{\text{clock}}}{\text{CPI} \times 10^6} \quad (29)$$

Example 5

Given:

- IC = 5 instructions
- Stall cycles = 2
- Clock = 500 MHz

Step-by-step:

- Clock cycles:
$$5 + 2 + 4 = 11$$
- CPI:
$$\frac{11}{5} = 2.2$$
- MIPS:
$$\frac{500 \times 10^6}{2.2 \times 10^6} \approx 227$$

Thus, the effective throughput is about **227 MIPS**.

Performance in Loops

Suppose we have:

- n **iterations**
- Loop of m **instructions** per iteration
- k **stalls** per iteration

Formulas per iteration:

- **IC per iteration**:
$$\text{IC}_{\text{per iter}} = m \tag{30}$$
- **Clock Cycles per iteration**:
$$\#\ \text{Clock Cycles}_{\text{per iter}} = m + k + 4 \tag{31}$$
- **Cycles Per Instruction per iteration**:
$$\text{CPI}_{\text{per iter}} = \frac{(m + k + 4)}{m} \tag{32}$$
- **MIPS per iteration**:
$$\text{MIPS} = \frac{f_{\text{clock}}}{\text{CPI}_{\text{per iter}} \times 10^6} \tag{33}$$

∞ Asymptotic Performance (Many Iterations)

If the loop runs for $n \to \infty$ iterations:

- **Total Instruction Count per iteration**:

$$\text{IC}_{\text{AS}} = \text{Instruction Count}_{\text{AS}} = m \times n \tag{34}$$

- **Total Clock Cycles per iteration**:

$$\begin{aligned} \#\text{ Clock Cycles}_{\text{AS}} &= \text{IC}_{\text{AS}} + \#\text{ Stall Cycles}_{\text{AS}} + 4 \\ &= (m \times n) + (k \times n) + 4 \end{aligned} \tag{35}$$

- **Cycles Per Instruction per iteration**:

$$\text{CPI}_{\text{AS}} = \lim_{n\to\infty} \frac{\#\text{ Clock Cycles}_{\text{AS}}}{\text{IC}_{\text{AS}}} = \lim_{n\to\infty} \frac{m \times n + k \times n + 4}{m \times n} = \frac{m+k}{m} \tag{36}$$

  The fixed 4 cycles become irrelevant when n becomes very large.

- **MIPS per iteration**:

$$\text{MIPS}_{\text{AS}} = \frac{f_{\text{clock}}}{\text{CPI}_{\text{AS}} \times 10^6} \tag{37}$$

⚖ Ideal vs Realistic Pipelining

The ideal CPI on a pipelined processor would be 1, but stalls cause the pipeline performance to degrade form the ideal performance, so we have:

- **Ideal pipelining**:

$$\text{Ideal CPI} = 1 \tag{38}$$

  1 instruction per cycle after filling the pipeline.

- **Realistic pipelining**:

$$\text{CPI} = 1 + (\text{stall cycles per instruction}) \tag{39}$$

Stalls are caused by:

- Structural Hazards (e.g., hardware resources conflict)
- Data Hazards (e.g., instruction dependencies)
- Control Hazards (e.g., branches)
- Memory Stalls (e.g., cache misses)

Pipeline Speedup

Speedup measures how much **pipelining** performance compared to **unpipelined execution**. General formula:

$$\text{Pipeline Speedup} = \frac{\text{Avg Exec Time Unpipelined}}{\text{Avg Exec Time Pipelined}} = \frac{\text{CPI}_{\text{unp}} \times T_{\text{clk, unp}}}{\text{CPI}_{\text{pipe}} \times T_{\text{clk, pipe}}} \tag{40}$$

If we **ignore the cycle time overhead** of pipelining and we assume the **stages are perfectly balanced**, the clock cycle time of unpipelined/pipelined processors can be equal, so:

$$\text{Pipeline Speedup} = \frac{\text{CPI}_{\text{unp}}}{1 + \text{CPI}_{\text{pipe}}} = \frac{\text{CPI}_{\text{unp}}}{1 + \text{Stall Cycles per Instruction}} \tag{41}$$

If we assume that **each instruction takes the same number of cycles**, which must be **equal to the number of pipeline stages** (called pipeline depth):

$$\text{Pipeline Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Stall Cycles per Instruction}} \tag{42}$$

If there were **no pipeline stalls (ideal case)**, this leads to the intuitive result that pipelining improves performance by the depth of the pipeline:

$$\text{Speedup} = \text{Pipeline Depth} \tag{43}$$

Thus, a 5-stage pipeline **could at best achieve 5x speedup only if there are no stalls**.

Impact of Branches

Branches cause **pipeline stalls**, and their effect is quantified as:

$$\text{Pipeline Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Branch Frequency} \times \text{Branch Penalty}} \tag{44}$$

- **Branch Frequency**: fraction of instructions that are branches.
- **Branch Penalty**: number of stall cycles per branch.

Where the Pipeline Stall Cycles per Instruction due to Branches (PSCIB) is:

$$\text{PSCIB} = \text{Branch Frequency} \times \text{Branch Penalty} \tag{45}$$

4.4 Memory Hierarchy

In modern processors we organize memories as a **hierarchy**.

- At the **top**, **smallest** but **fastest** memories (e.g., L1 cache, CPU registers).
- At the **bottom**, **largest** but **slowest** memories (e.g., DRAM, disk storage).

The idea is to keep most data in fast memories for efficiency.

- **Hit**. The data we want is **found** in the upper (faster) memory level.
- **Hit Rate**. Probability (fraction) of memory accesses that result in a hit.

$$\text{Hit Rate} = \frac{\#\,\text{hits}}{\#\,\text{memory accesses}} \tag{46}$$

- **Hit Time**. Time to **access** the upper memory level and check whether it's a hit.
- **Miss**. The data is **not found** in the upper memory, we must access the slower lower level.
- **Miss Rate**. Probability (fraction) of accesses that result in a miss.

$$\text{Miss Rate} = \frac{\#\,\text{misses}}{\#\,\text{memory accesses}} \tag{47}$$

  Important property:

$$\text{Hit Rate} + \text{Miss Rate} = 1 \tag{48}$$

- **Miss Time**. Total time when there is a miss:

$$\text{Miss Time} = \text{Hit Time} + \text{Miss Penalty} \tag{49}$$

  Where **Miss Penalty** is extra **time needed to fetch data from the lower level and update upper cache**.

AMAT (Average Memory Access Time)

AMAT (Average Memory Access Time) measures the **average time the processor needs to access memory** (whether there's a hit or a miss).

$$\text{AMAT} = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time}$$

It combines both: the fast accesses (hits) and the slow accesses (misses). Since equation 49, we can substitute into the AMAT formula:

$$\text{AMAT} = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times (\text{Hit Time} + \text{Miss Penalty})$$

But by definition:

$$\text{Hit Rate} + \text{Miss Rate} = 1$$

Thus:

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty} \tag{50}$$

This is the most important and most used AMAT formula!

How to improve cache performance?

To reduce AMAT (and improve system speed), we can:

1. **Reduce Hit Time**: Make the **cache smaller**, **simpler**, or **closer to the CPU**.
2. **Reduce Miss Rate**: Improve **cache organization** (better replacement, associativity, prefetching).
3. **Reduce Miss Penalty**: Use **faster lower levels** (e.g., add an L2 or L3 cache).

Unified Cache vs Separate Instruction

There are two major architectures for L1 cache:

- **Unified Cache**: Same L1 cache for both instructions and data.
- **Separate I$ and D$ (Harvard Architecture)**: Different L1 caches for instructions (I$) and data (D$).

For **separate caches** (Harvard Architecture):

- Instruction cache and Data cache have **different miss rates**.
- **Accesses are divided** between instruction and data operations.

This is the AMAT (Average Memory Access Time) for Harvard architectures:

$$\begin{aligned} \text{AMAT}_{\text{Harvard}} = \quad & (\%\text{Instr}) \times \\ & (\text{Hit Time} + \text{Miss Rate}_{\text{I\$}} \times \text{Miss Penalty}) + \\ & (\%\text{Data}) \times \\ & (\text{Hit Time} + \text{Miss Rate}_{\text{D\$}} \times \text{Miss Penalty}) \end{aligned} \tag{51}$$

Example 6: Harvard Architecture

Assumptions:

- 16 KB I$ with Miss Rate = 0.64%
- 16 KB D$ with Miss Rate = 6.47%
- 32 KB Unified Cache with Aggregate Miss Rate = 1.99%
- 33% of accesses are loads/stores, so:
  - 75% accesses are instructions
  - 25% accesses are data
- Hit Time = 1 cycle, Miss Penalty = 50 cycles
- Data hit has 1 more stall for Unified cache (only one port)

Which cache is better?
Calculate Harvard AMAT (equation 51):

$$\begin{aligned}
\text{AMAT}_{\text{Harvard}} &= 0.75 \times (1 + 0.0064 \times 50) + 0.25 \times (1 + 0.0647 \times 50) \\
&= 0.75 \times (1 + 0.32) + 0.25 \times (1 + 3.235) \\
&= 0.75 \times 1.32 + 0.25 \times 4.235 \\
&= 0.99 + 1.05875 \\
&= 2.04875 \approx 2.05\,\text{cycles}
\end{aligned}$$

Calculate AMAT (Unified cache):

$$\begin{aligned}
\text{AMAT} &= 0.75 \times (1 + 0.0199 \times 50) + 0.25 \times (1 + 1 + 0.0199 \times 50) \\
&= 0.75 \times (1 + 0.995) + 0.25 \times (2 + 0.995) \\
&= 0.75 \times 1.995 + 0.25 \times 2.995 \\
&= 1.49625 + 0.74875 \\
&= 2.245 \approx 2.24\,\text{cycles}
\end{aligned}$$

So **Harvard (separate I$ and D$)** gives **better performance** than Unified Cache, because AMAT (Average Memory Access Time) is less.

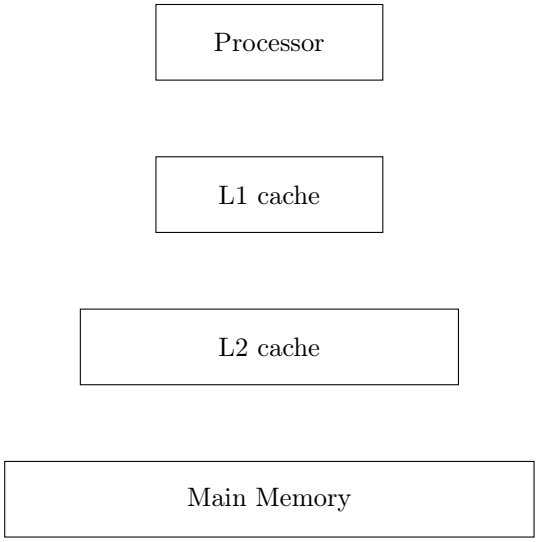
Miss Penalty Reduction: Second Level Cache (L2)

L1 cache must be **very small and fast** (to match the fast CPU cycle time). But small cache, relatively **high miss rate**. Every miss in L1 would normally mean **access to slow main memory** (very expensive!).

The solution is to **insert a larger, slower L2 cache** between L1 and main memory. Thus:

- If L1 misses, we first **try to find the data in L2**.
- Only if L2 also misses, we go to **main memory**.

This **reduces the effective miss penalty** seen by the CPU.



When using both L1 and L2 caches, first:

$$\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \cdot \text{Miss Penalty}_{L1}$$

where:

$$\text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \cdot \text{Miss Penalty}_{L2}$$

Thus, expanding:

$$\begin{aligned} \text{AMAT} = \quad & \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \cdot \\ & (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \cdot \text{Miss Penalty}_{L2}) \end{aligned}$$

or simplified the **AMAT** is:

$$\begin{aligned} \text{AMAT} = \quad & \text{Hit Time}_{L1} + \\ & \text{Miss Rate}_{L1} \cdot \text{Hit Time}_{L2} + \\ & \text{Miss Rate}_{L1} \cdot \text{Miss Rate}_{L2} \cdot \text{Miss Penalty}_{L2} \end{aligned} \tag{52}$$

Each term captures: immediate L1 access, possible L2 access, and final access to main memory if L2 also misses.

⚖ Local vs Global Miss Rates

- **Local Miss Rate**: misses divided accesses at that specific cache level. For example:
  - L1 Local Miss Rate = Misses at L1 ÷ Accesses to L1
  - L2 Local Miss Rate = Misses at L2 ÷ Accesses to L2
- **Global Miss Rate**: misses at a level relative to **total CPU accesses**.
  - For L1:
  $$\text{Global Miss Rate}_{L1} = \text{Miss Rate}_{L1} \tag{53}$$
  - For L2:
  $$\text{Global Miss Rate}_{L2} = \text{Miss Rate}_{L1} \cdot \text{Miss Rate}_{L2} \tag{54}$$

  Global miss rate is what really matters, because it tells us **what percentage of memory accesses are so unlucky** that they: miss in L1 (fast small cache), miss in L2 (bigger slower cache), go down to main memory (very slow).

Example 7: Local vs Global Miss Rates

Let us consider a computer with a L1 cache and L2 cache memory hierarchy. Suppose that in 1000 memory references there are 40 misses in L1 and 20 misses in L2. *What are the various miss rates?*

- L1 Miss Rate:
$$\frac{40}{1000} = 4\%$$
- L2 Local Miss Rate:
$$\frac{20}{40} = 50\%$$
- L2 Global Miss Rate:
$$\text{Miss Rate}_{L1} \cdot \text{Miss Rate}_{L2} = 4\% \cdot 50\% = 2\%$$

Thus, only 2% of all CPU memory accesses end up reaching main memory.

Adding an L2 cache:

- ✔ **Increases hit rate** (fewer accesses to slow main memory),
- ✔ **Reduces effective miss penalty**,
- ✔ **Improves CPU performance** by reducing memory stall cycles.

Impact of Memory Hierarchy on CPU Execution Time

Without cache every memory access would go directly to main memory (very slow). With cache:

- Most accesses are **fast** (cache hits),
- Only a few accesses are **slow** (cache misses).

Thus, **memory stalls** (waiting for memory) become part of the CPU execution time. Therefore, the CPU total execution time depends on:

- **CPU execution cycles** (doing normal instruction work),
- **Memory stall cycles** (waiting for slow memory).

The formal formula for CPU time is:

$$\text{CPU Time} = (\text{CPUexec Cycles} + \text{Memory Stall Cycles}) \times T_{\text{CLK}}$$

Where:

- T_{CLK}: clock cycle time.
- **CPUexec Cycles**: cycles needed for normal instruction execution (ALU operations, Load/Store assuming all hits).

$$\text{CPUexec Cycles} = \text{IC} \times \text{CPI}_{\text{exec}} \tag{55}$$

  Where:

  - IC: Instruction Count.
  - CPI_{exec}: ideal CPI **without considering cache misses**.

- **Memory Stall Cycles**: extra cycles lost because of memory misses. Memory stall cycles are caused by misses times penalty per miss. Specifically:

$$\text{Memory Stall Cycles} = \text{IC} \times \text{Misses per Instruction} \times \text{Miss Penalty} \tag{56}$$

  And:

$$\begin{aligned} \text{Misses per Instruction} &= \frac{\#\text{ misses}}{\text{IC}} \times \frac{\#\text{ mem. accesses}}{\text{IC}} \times \text{Miss Rate} \\ &= \text{MAPI} \times \text{Miss Rate} \end{aligned} \tag{57}$$

  (where MAPI is Memory Accesses per Instruction); Therefore:

$$\text{Memory Stall Cycles} = \text{IC} \times \text{MAPI} \times \text{Miss Rate} \times \text{Miss Penalty} \tag{58}$$

Putting it all together, we get the **Full CPU Time formula**:

$$\text{CPU Time} = \text{IC} \cdot (\text{CPI}_{\text{exec}} + \text{MAPI} \cdot \text{Miss Rate} \cdot \text{Miss Penalty}) \cdot T_{CLK} \tag{59}$$

This formula shows **both**: the "normal" CPU work, and the "extra time" lost due to cache misses. These are also **(ideal) special cases**:

✔ **Ideal cache (100% hits)**:

$$\text{CPU Time} = \text{IC} \times \text{CPI}_{\text{exec}} \times T_{CLK}$$

✖ **No cache (100% misses)**:

$$\text{CPU Time} = \text{IC} \times (\text{CPI}_{\text{exec}} + \text{MAPI} \times \text{Miss Penalty}) \times T_{CLK}$$

⚠ **Including Pipeline Stalls**. If we also want to consider **pipeline stalls** (caused by hazards like data dependencies), the full CPU Time becomes:

$$\begin{aligned} \text{CPU Time} = \quad & \text{IC} \times (\text{CPI}_{\text{exec}} + \text{Stalls per Instruction} + \\ & \text{MAPI} \times \text{Miss Rate} \times \text{Miss Penalty}) \times \\ & T_{CLK} \end{aligned} \tag{60}$$

Where **Stalls per Instruction** is the average extra cycles lost per instruction due to pipeline hazards. So we can see that **pipeline hazards** and **memory "misses"** both **contribute to processor slowdowns**.

This is important because even if we build the fastest CPU core in the world, if our memory system is slow, our real performance will be bad! Memory system design (good caches, good pipelines) is **essential** for real CPU performance.

⚠ Memory Stalls in L1 and L2 Caches

In a **hierarchical cache system**:

- A memory access **first** tries L1,
- If **L1 misses**, tries **L2**,
- If **L2 also misses**, goes to **main memory** (very slow).

Each **level of cache** can **introduce extra stall cycles**:

- Stall cycles after an L1 miss → time to access L2.
- Stall cycles after an L2 miss → time to access main memory.

Average **memory stall cycles per instruction** (considering both L1 and L2) is:

$$\begin{aligned}\text{Memory stall cycles per instruction} = \quad & (\text{Misses}_{L1} \cdot \text{Hit Time}_{L2}) + \\ & (\text{Misses}_{L2} \cdot \text{Miss Penalty}_{L2})\end{aligned} \tag{61}$$

Where:

- Misses_{L1} = Misses per instruction at L1.
- Misses_{L2} = Misses per instruction at L2.
- Hit Time_{L2} = Time to access L2 cache (after L1 miss).
- Miss Penalty_{L2} = Time to access main memory (after L2 miss).
- $(\text{Misses}_{L1} \cdot \text{Hit Time}_{L2})$ = The **first term** is for accesses that hit in L2 (still slower than L1 but faster than main memory).
- $(\text{Misses}_{L2} \cdot \text{Miss Penalty}_{L2})$ = The **second term** is for accesses that miss in both L1 and L2 (very slow).

If we want to **find misses per instruction**, they depend on MAPI (Memory Accesses Per Instruction) and miss rates:

- **Misses at L1 per instruction**

$$\text{Misses}_{L1} = \text{MAPI} \times \text{Miss Rate}_{L1}$$

- **Misses at L2 per instruction**

$$\text{Misses}_{L2} = \text{MAPI} \times \text{Global Miss Rate}_{L2}$$

  Where:

$$\text{Global Miss Rate}_{L2} = \text{Miss Rate}_{L1} \times \text{Miss Rate}_{L2}$$

Considering everything, the **Final Full CPU Time Formula** is:

$$\text{CPU Time} = \text{IC} \times (\text{CPI}_{\text{exec}} + \text{Memory stall cycles per instruction}) \times T_{\text{CLK}} \quad (62)$$

Where:

$$\begin{aligned} \text{Memory stall cycles per instruction} = \; & (\text{MAPI} \times \\ & \text{Miss Rate}_{L1} \times \\ & \text{Hit Time}_{L2}) + \\ & (\text{MAPI} \times \\ & \text{Miss Rate}_{L1} \times \\ & \text{Miss Rate}_{L2} \times \\ & \text{Miss Penalty}_{L2}) \end{aligned} \quad (63)$$

Putting it all together, the CPU Time is:

$$\text{IC} \cdot (\text{CPI}_{\text{exec}} + \text{MAPI} \cdot (\text{MR}_{L1} \cdot \text{HT}_{L2} + \text{MR}_{L1} \cdot \text{MR}_{L2} \cdot \text{MP}_{L2})) \cdot T_{\text{CLK}} \quad (64)$$

This is the most complete formula that accounts for:

- L1 and L2 cache effects,
- Main memory access,
- How memory stalls degrade CPU performance.

5 VLIW (Very Long Instruction Word)

5.1 Introduction

Traditional **compilers** use **static code scheduling** to exploit Instruction-Level Parallelism (ILP). Key tasks of the compiler:

- Detect **parallelizable** instructions considering: Hardware resource constraints and Data dependencies.
- **Schedule** instructions to execute **in parallel** when possible.
- Otherwise, **insert** `NOPs` (No Operations) if no safe parallel execution is possible.

Statically scheduled processors trust the compiler to "fill the pipeline" and avoid hazards at compile time.

VLIW Processors: Alternative Way to Extract ILP

VLIW (Very Long Instruction Word) processors are a class of architectures designed to **execute multiple operations in parallel during a single clock cycle**, but unlike superscalar processors, they **rely on the compiler** rather than on dynamic hardware mechanisms **to detect and schedule parallelism**.

The fundamental idea behind VLIW is to **group several independent operations together into a single long instruction word**, called a bundle. This bundle is **composed of several fixed slots**, each one **corresponding to a different functional unit in the processor**, such as an integer ALU, a floating-point unit, a load/store unit, or a branch unit.

For example, in a 5-issue VLIW processor, a single bundle would typically carry up to five operations: integer, floating-point, load/store, branch, etc.

The **key difference** compared **to traditional superscalar** processors lies in who decides what can run in parallel. In superscalar designs, the hardware dynamically analyzes dependencies between instructions at runtime, which requires complex circuitry for hazard detection and scheduling. In VLIW architectures, this analysis is performed entirely at compile time. The **compiler is responsible for**:

- **Statically identify** independent operations.
- **Solve structural hazards** (e.g., two operations trying to use the same hardware unit).
- **Solve data hazards** (dependencies between instructions).
- Insert **NOPs** when necessary.

When no useful instruction is available for a particular slot, the compiler inserts a `NOP` (no-operation). In cases where conflicts cannot be avoided, NOPs are inserted into the bundle to fill empty slots.

Why move to Compiler?

The problem with Superscalar is that the hardware requires complex dynamic scheduling and dependency checking, which **costs area and power**.

VLIW Idea is:

- Push **complexity** to the **compiler**.
- **Compiler groups parallel operations** into a **single bundle**.
- No need for runtime dependency checking anymore.

The VLIW paradigm is characterized by the use of **very wide instruction words**, **each containing multiple independent operations** ("syllables"). A multiple-issue VLIW processor typically features specialized units for integer, floating-point, memory access, and branch instructions. For example, a 4-issue VLIW processor will have four operation slots per bundle, each connected to a different unit.

Multiple-issue VLIW: Operation Latencies

An important aspect in VLIW scheduling is the **management of operation latencies**. Each operation type may require a **different number of clock cycles to complete**. Integer operations, memory accesses, and floating-point computations may all have varying latencies, and these differences must be carefully considered during scheduling. The **compiler must plan the execution** so that operations complete in the correct order and without causing unnecessary stalls in the pipeline.

In summary, VLIW architectures represent a shift of complexity from the hardware to the compiler, aiming for more efficient and simpler processor designs while demanding sophisticated compiler techniques to fully exploit available parallelism.

Single Program Counter and Branch Management

In a VLIW processor, even though multiple operations are issued simultaneously, the architecture still uses a **single program counter** to fetch instructions. Each instruction fetched corresponds to a **bundle**, which can contain multiple parallel operations.

Importantly, within each bundle, there can be **at most one branch instruction** that affects control flow. This constraint simplifies the handling of branches, making it easier for the processor to predict and manage the program's execution path.

Shared Multi-Ported Register File

Since a VLIW processor issues multiple operations in parallel, the **register file must support multiple simultaneous reads and writes**. In a typical 4-issue VLIW machine, the register file needs to have enough ports to read **eight source operands** and write **four destination results** every clock cycle.

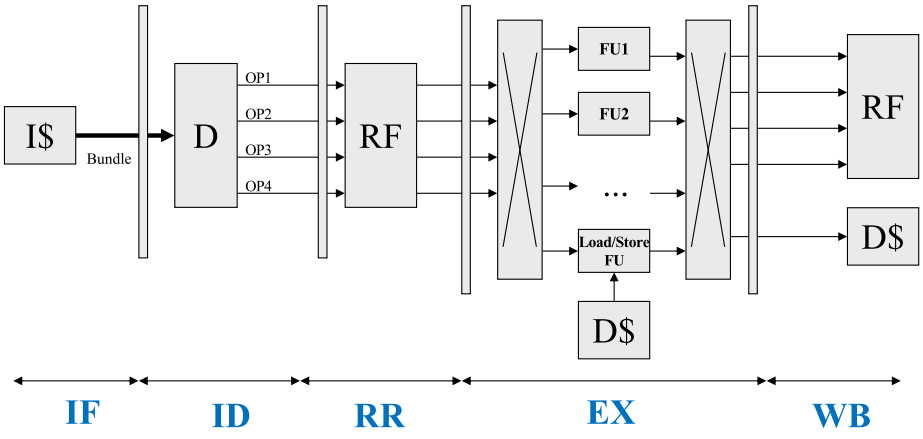
This requirement implies a **shared, multi-ported register file**, which is one of the non-trivial aspects of the hardware design. It ensures that all functional units can access their operands without creating bottlenecks.

Importance of Parallelism and the Use of NOPs

To achieve high performance, the **compiler must ensure that the source code has enough parallelism** to fill all the available operation slots in each bundle. When **sufficient independent instructions are not available**, **NOPs are inserted** into the empty slots. This insertion preserves the fixed structure of the bundle but can lead to inefficiencies if the application does not expose enough ILP.

Pipeline Organization

VLIW processors often use a **pipelined execution model** similar to traditional RISC architectures. A standard pipeline might include stages like **Instruction Fetch (IF)**, **Instruction Decode (ID)**, **Register Read (RR)**, **Execution (EX)**, and **Write Back (WB)**.



In the previous figure, a **5-stage pipeline** is used, with each bundle passing through these stages. Each operation within a bundle proceeds independently through the functional units connected to the pipeline.

Operation Dispatch and Decode

If the architecture dedicates **one functional unit per slot**, the decode stage becomes relatively simple. Each operation is directed straight to its corresponding functional unit for execution without the need for complex arbitration.

However, if there are **more functional units than slots**, meaning there is a surplus of parallel hardware resources, the architecture must include a **dispatch network**. This network is responsible for forwarding each operation, along with its operands, to the appropriate available functional unit. In this way, the design of the dispatch system depends heavily on the organization of functional units relative to the number of issue slots defined by the instruction bundle format.

5.2 Data dependencies

In VLIW architectures, **data dependencies**:

- True dependencies (RAW: Read After Write)
- Anti-dependencies (WAR: Write After Read)
- Output dependencies (WAW: Write After Write)

Are **handled entirely by the compiler**. During the compilation process, the **compiler analyzes the functional unit latencies and rearranges the instruction sequence to eliminate conflicts**. When dependencies cannot be resolved by reordering, the compiler inserts `NOP`s to delay operations and maintain correct execution order.

This static management of hazards is **crucial** because the hardware in VLIW processors does not dynamically detect or resolve data hazards during execution.

Operation Latency Management

In order to correctly schedule instructions, the compiler must also consider **operation latencies**.

If an **operation**, such as a multiplication, **requires multiple cycles** to complete, the **compiler must account for this delay explicitly**. It can either rearrange independent instructions to fill the delay or insert `NOP`s to stall dependent instructions until the result is ready.

For instance, if an instruction `C = A * B` has a latency of two cycles, the compiler may schedule a `NOP` after it, before allowing an instruction that depends on `C` to execute. This **careful latency management ensures that data hazards are avoided** and that results are available when needed, even when using pipelined functional units.

WAR, WAW, and Structural Hazards

Besides true dependencies, **anti-dependencies (WAR)** and **output dependencies (WAW)** are also statically resolved at compile time. The compiler can manage these by:

- ✔ Appropriately **choosing the timing of operations**.
- ✔ Using **register renaming** techniques to avoid conflicts.

Structural hazards, such as two operations trying to use the same hardware resource simultaneously, are similarly **resolved** by the compiler, either **through scheduling decisions** or **resource management strategies**.

In addition to data and structural hazards, the compiler may provide **static branch prediction hints**. However, if the **prediction is wrong**, it is the **hardware's responsibility** to detect the misprediction and flush the pipeline, just as in *traditional architectures*.

Maintaining In-Order Write-Back

An **important constraint** in VLIW processors is that **all operations in a bundle should complete their write-back stage at the same clock cycle**. This synchronization prevents structural hazards in accessing the register file and avoids WAR and WAW hazards during register writes.

If operations within a bundle have **different latencies**, they are **all forced to align to the longest latency operation** among them. **Otherwise**, if operations were allowed to complete independently, **out-of-order write-backs** would occur, and the **processor would require additional hardware to track and resolve register file conflicts**, precisely the kind of complexity VLIW designs try to avoid.

Register Pressure

One important issue in VLIW architectures is the problem of **register pressure**. Because VLIW bundles can issue multiple operations simultaneously, and because many **operations have multicycle latencies**, a **large number of registers may be occupied at the same time by intermediate results waiting to complete**.

For instance, if a bundle contains two floating-point operations, each requiring five clock cycles to complete, their destination registers remain occupied for the entire duration of those five cycles. Since a VLIW processor continues issuing new bundles each cycle, this quickly accumulates a large number of active registers. In the given example, five bundles are issued while the first two results are still pending, resulting in up to twenty operations competing for register resources.

The situation is further exacerbated by **register renaming**, a technique used by the compiler to solve WAR (Write After Read) and WAW (Write After Write) hazards statically. Register renaming:

- ✔ **Avoids conflicts**
- ✖ **Increases the number of physical register needed**, thereby amplifying register pressure.

Managing register pressure is therefore a **crucial task** for the compiler, often requiring careful allocation strategies or even **spilling values temporarily to memory** when the **register file becomes saturated**.

⚠ Dynamic Events in VLIW

Although VLIW processors rely heavily on static compilation, **some dynamic events cannot be fully predicted or controlled at compile time**.

- A common example is the **data cache miss**. While the latency of a cache miss is known in general terms (e.g., memory access delay), whether a particular memory access will cause **a miss depends on runtime conditions**. If a cache miss occurs, the **processor experiences a stall**, **which was not anticipated during static scheduling**.
- Another dynamic event is the **branch misprediction**. Although the compiler may provide static branch hints, actual program execution might differ. When a **branch is mispredicted**, the **processor must dynamically flush the pipeline** to discard speculative instructions that were fetched and partially executed under the wrong control flow assumption.

In both cases, while VLIW simplifies hardware by eliminating dynamic scheduling for regular instruction flow, it still requires mechanisms to handle these unpredictable runtime events.

5.3 Statically Scheduled Processors

In **statically scheduled architectures**, the **compiler is responsible for arranging instructions to exploit Instruction-Level Parallelism** (ILP). By carefully analyzing data dependencies and resource constraints, the compiler reorders operations to maximize parallel execution and minimize stalls.

A **Basic Block** is the **main unit considered during static scheduling**. It consists of a **straight-line code sequence without branches except at the entry and exit points**. However, in practice, basic blocks tend to be **small**: for typical programs, only around 4 to 7 instructions are found between branches. Moreover, even within a basic block, **true data dependencies** (RAW, Read After Write) further **limit the amount of exploitable parallelism**, forcing some instructions to be issued sequentially.

As a result, achieving substantial performance improvements requires techniques that go **beyond individual basic blocks**, aiming to exploit ILP across larger regions of code.

VLIW Processors: Main Advantages

- VLIW architectures represent a **powerful solution for statically exploiting ILP**. The extensive use of compiler optimizations enables VLIW processors to achieve **high performance**, even **without the complex runtime scheduling hardware** used in superscalar designs.
- Since the compiler performs *dependency checking* and *scheduling at compile time*, it has the **advantage of analyzing the program with a much larger instruction window** than would be feasible in hardware. This allows **more opportunities to detect parallelism**, especially when aggressive code transformations, such as loop unrolling or software pipelining, are applied.
- By transferring complexity from hardware to software, VLIW processors achieve a **significant reduction in hardware complexity**. Smaller die areas lead to **cheaper production costs** and **lower power consumption**, making VLIWs attractive for embedded applications.

  Moreover, the **fixed format of VLIW instructions simplifies the decode logic**, and **scaling** to a larger number of functional units **becomes easier**.

Open Challenges of VLIW Architectures

Despite these advantages, VLIW designs face several open challenges:

- First, they rely heavily on **strong compiler technology**. Sophisticated compilation techniques are necessary not only to detect parallelism within basic blocks but also to **manage parallelism across larger code regions**, which increases the complexity of compiler design.

- Another issue is the **increase in code size**. Because the instruction bundles have a fixed structure, and NOPs must be inserted to handle scheduling gaps, **increase in code size**. While code compression techniques can mitigate this effect, they **add extra decoding complexity to the processor**.
- **Register management** is also a concern. The extensive use of **register renaming** to avoid WAR and WAW hazards **increases the number of required registers** and complicates the organization of the register file. Clustered VLIW designs have been proposed to partially address this problem.
- Finally, **binary incompatibility** is a **major limitation**.

  Even small changes in the number of slots, the types of functional units, or their latencies can render code compiled for one **VLIW processor incompatible with another**, even if they share the same instruction set architecture.

  This **lack of portability contrasts with more dynamic architectures**, where binary compatibility is preserved across generations.

Although solutions such as **Just-In-Time compilation** have been explored, they **introduce additional complexity and are rarely adopted in practice**. For this reason, VLIW processors are primarily **employed in embedded systems**, where binaries can be compiled specifically for the target hardware and do not require high portability.

5.4 Code Scheduling

As we saw on page 185, Code Scheduling is the process of rearranging program instructions to maximize parallelism during execution. The primary goal of **Code Scheduling** is to **statically reorder instructions** within object code in such a way that:

- ✔ Execution time is minimized.
- ✔ Semantic correctness is preserved.

In simple terms, we want the program to run as **fast as possible without changing what it computes**. This is essential because in architectures like VLIW (Very Long Instruction Word), the hardware doesn't handle instruction reordering dynamically. Instead, the **compiler is fully responsible** for finding parallelism and deciding the scheduling.

❓ Motivation in VLIW Architectures

In VLIW architectures, the **responsibility of exploiting** instruction-level parallelism (**ILP**) is **shifted from the hardware to the compiler**. This shift implies several things:

- The **hardware is simpler**: no complex dynamic scheduling (like Tomasulo or Scoreboarding).
- **Parallel instructions are issued together** as part of a wide instruction word.
- **Each instruction** slot in a VLIW word maps to a specific **functional unit**.

❓ **Why do we need scheduling in this context?** Because a VLIW machine expects **parallel instructions to be packed statically** in the same bundle. If the compiler fails to schedule efficiently, the result is many **NOPs** and a waste of hardware resources.

⚠ Semantic Correctness and Performance

It's not enough to just reorder instructions for performance. We must also preserve:

- **True data dependencies (RAW)**, instructions must respect the order of reads after writes.
- Avoid introducing errors by violating:
  - **Anti-dependencies (WAR)**
  - **Output dependencies (WAW)**

The scheduler (compiler) must:

- ✔ Carefully **analyze the dependency graph** of the instructions.
- ✔ **Reorder only when it's safe.**
- ✔ **Possibly apply renaming** or other transformations to eliminate false dependencies.

5.4.1 Scheduling Basics

A **Basic Block** is the smallest unit of code for scheduling purposes. It's a straight-line code sequence that has:

- **One single entry point** (i.e., no jumps into the middle).
- **One single exit point** (i.e., only the last instruction may jump elsewhere).

In practice:

- No branches except at the **start (entry)** and **end (exit)**.
- All instructions inside a basic block **execute sequentially and always together**.
- Used by compilers to isolate code regions where **safe reordering** can be performed.

This isolation is essential for local scheduling techniques like **loop unrolling** and **software pipelining**.

Dependence Graph (DAG, Directed Acyclic Graph)

Once a basic block is identified, the **compiler builds a dependence graph** to model how instructions depend on each other.

- Each **node** in the graph represents an **instruction**.
- Each **edge** denotes a **data dependence** between two instructions.
- The **type of edge** can correspond to:
  - **RAW** (true dependence).
  - **WAR**, **WAW** (name dependencies, can often be eliminated via register renaming).

This **graph helps determine**:

- ✔ Which instructions are **ready to execute**.
- ✔ Which instructions must **wait for others to complete**.
- ✔ The **longest dependency chain**, which limits parallelism.

Why is this graph useful?

There are two main reasons:

1. It **exposes parallelism**: independent instructions can be scheduled in the same cycle or in parallel execution units.
2. It defines the **Critical Path** of execution: the **longest path from a source to a sink node** (i.e., sequential dependencies that can't be parallelized).

5.4.2 Dependence Graph and Critical Path

A **Dependence Graph** (also called a **Data Dependence Graph** or **Instruction Dependence Graph**) is a **directed acyclic graph (DAG)** used by the compiler to represent **data dependencies among instructions** in a **basic block**. It is fundamental in instruction scheduling, especially in VLIW and other statically scheduled architectures.

How to Build the Dependence Graph

To build a Dependence Graph from a basic block:

1. **Create a node** for each individual instruction in the basic block.
2. **Add edges** between nodes based on dependencies: an edge from instruction $I_1 \rightarrow I_2$, means that I_2 depends on the results of I_1, and I_2 cannot execute before I_1.
3. **Label each edge** with the type of dependency (usually RAW is critical, WAR/WAW can often be resolved by renaming).
4. **Annotate each node** with the instruction **latency** (how long it takes to execute) and other scheduling metadata (like priority).

Each path in the graph shows **which instructions must wait for which**. The deeper the graph, the more serialization is required.

Longest Path Computation

Each node is annotated with:

$$\text{Longest Path}\,(i) = \max\left[\text{Longest Path}\,(p)\right] + \text{Latency}\,(i) \tag{65}$$

Where p are all the **predecessors** of instruction i.

- This gives the **maximum cumulative latency** needed to reach instruction i.
- It reflects **how deep a node is in terms of data dependencies**.

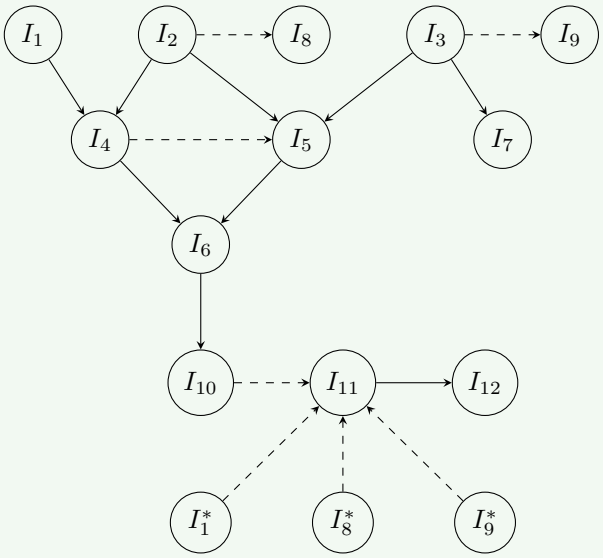
Once all node values are computed, the **maximum among them gives the Critical Path**.

Example 1: Exam - 11 february 2025

Let's consider the following `LOOP` code:

```
1  LOOP:   LD F1, A(R1)
2          LD F2, A(R2)
3          LD F3, A(R3)
4          FADD F1, F1, F2
5          FADD F2, F2, F3
6          FMUL F1, F1, F2
7          FADD F3, F3, F3
8          ADDUI R2, R1, 8
9          ADDUI R3, R1, 8
10         SD F1, B(R1)
11         ADDUI R1, R1, 4
12         BNE R1, R6, LOOP
```

The DAG is:



An oriented line indicates a RAW dependency. A dashed line indicates a WAR or WAW dependency. For this reason, labels are not inserted. Additionally, since the latency is calculated by hand, no text is inserted to avoid confusion in the graph. The asterisk indicates that these nodes were redrawn for readability.

What is the Critical Path?

The **Critical Path** is the **longest path** through the dependence graph, starting from an entry instruction and ending at an instruction with no successors.

- It represents the **minimum number of clock cycles** needed to execute the basic block, even with unlimited resources.
- Any attempt to **speed up execution** (through scheduling or parallelism) **cannot beat this lower bound**.

If our goal is to schedule instructions as efficiently as possible, the critical path is our **primary constraint**.

5.4.3 ASAP Scheduling Algorithm (As Soon As Possible)

ASAP (As Soon As Possible) Scheduling is a **greedy scheduling algorithm** used in instruction scheduling. Its main idea is to **schedule each operation as early as possible** once its **dependencies are satisfied**.

- It does not consider resource constraints.
- It focuses only on **data readiness**.
- It is useful for computing **lower bounds** on execution time and for **priority estimation**.

How ASAP Scheduling Works

1. Build the **dependence graph** (page 271).
2. Initialize a **ready list** with all **source nodes** (instructions with no predecessors). A **Ready List** is a list of all instructions **ready to be scheduled** at a given cycle. An instruction is ready if:
   - All its predecessors have been scheduled.
   - Its operands are available (i.e., computed and propagated).
3. For each cycle:
   - Schedule every instruction in the ready list.
   - Update successors (check if all their predecessors are now scheduled).
   - Move newly ready instructions into the next cycle's ready list.
4. Repeat until all instructions are scheduled.

ASAP produces **one of the shortest possible schedules**, assuming no resource constraints. The result gives the **earliest cycle** at which each instruction can execute.

Why use ASAP Scheduling if it ignores resource constraints?

1. **It Computes a Theoretical Lower Bound**. ASAP gives us the earliest possible execution cycle for each instruction, assuming infinite resources. The critical path of the dependence graph emerges directly from ASAP: it is the shortest possible execution time, even for a perfectly parallel machine. We use **ASAP as a baseline** because **we know we can't do better than this in terms of schedule length**.
2. **It Helps Assign Priorities for Real Scheduling Algorithms**. In list-based scheduling, instructions are chosen from the ready list based on priority. One way to assign this priority is using the Longest Path to Sink (from ASAP): instructions deeper in the graph (closer to the end) are more urgent. So **even if ASAP isn't the final schedule, it informs how to choose wisely in realistic algorithms**.

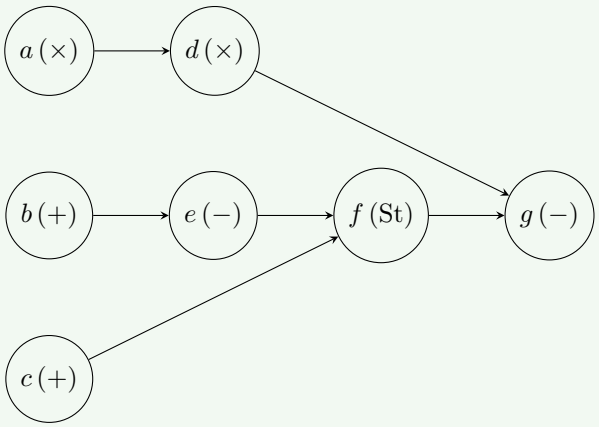
3. **It's a Building Block for More Advanced Scheduling**. Many algorithms, like List Scheduling, start by computing the ASAP schedule. Then they incorporate resource constraints and adjust from there. Think of it as the "**first approximation**", **later refined under constraints**.

4. **Compiler Simplicity in Early Stages**. In early compiler passes (before hardware-specific optimization), ASAP can: help restructure code, estimate ILP, and guide unrolling or pipelining transformations.

A great analogy: ASAP is like planning a trip **assuming we hit green lights at every intersection**. It's not realistic, but it **gives us a best-case travel time**, which is useful for planning and comparison.

Example 2: ASAP Scheduling Algorithm

Consider the following dependence graph:



Each node has the corresponding operation in brackets. For example, "St" is the store operation. Multiply operations take three latency cycles; ALU operations (addition and subtraction) take one; and the store operation takes two.

Ready List

Cycle	Ready List
1	a, b, c
2	e
3	f
4	d
5	
6	
7	g

Resource Reservation Table

Cycle	ALU1	ALU2	L/S	MUL
1	b	c		a
2	e			a
3	f		f	a
4			f	d
5				d
6				
7	g			

5.4.4 List-Based Scheduling Algorithm

The **List-Based Scheduling Algorithm** is a realistic and practical scheduling technique that improves over ASAP by taking into account **resource constraints**.

Unlike ASAP (which assumes infinite resources), **list-based scheduling**:

- Works with a limited number of functional units (ALU, MUL, etc.).
- Ensures that at each cycle:
  - ✔ We **do not oversubscribe** any hardware unit.
  - ✔ We only schedule instructions **whose operands are ready**.

This reflects **real hardware limitations** in VLIW superscalar, or statically scheduled pipelined processors.

Algorithm Steps

At each cycle, the algorithm maintains a **Ready Set**, similar to ASAP. But now, when multiple instructions are ready, and **not all can be scheduled** in parallel, it must **choose** based on a **priority**.

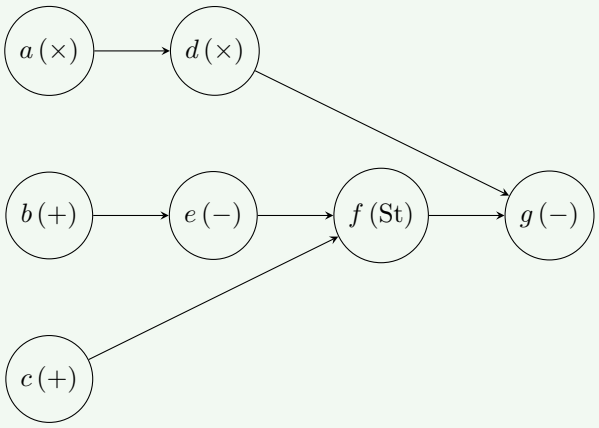
1. Build the dependence graph.
2. Compute **priorities** for all nodes (typically with longest-path-to-sink). Nodes closer to the end of the graph are **more critical** (less slack). The compiler calculates for each instruction:

   $$\text{priority}\,(i) = \text{Max path length from } i \text{ to a sink node}$$

   Higher priority, more urgent to schedule.
3. Initialize the **Ready Set** with nodes that have no predecessors.
4. For each cycle:
   - While resources are available and Ready Set is not empty:
     - Pick the **highest-priority instruction** that fits into available resources.
     - Assign it to the current cycle.
     - Mark it as scheduled
   - Update the Ready Set: add successors whose dependencies are now satisfied.
5. Repeat until all instructions are scheduled.

Example 3: List-Based Scheduling Algorithm

Consider the following dependence graph:



Each node has the corresponding operation in brackets. For example, "St" is the store operation. Multiply operations take three latency cycles; ALU operations (addition and subtraction) take one; and the store operation takes two.

Ready List

Cycle	**Ready List**
1	a, b, c
2	c, e
3	e
4	d, f
5	
6	
7	g

Resource Reservation Table

Cycle	**ALU1**	**L/S**	**MUL**
1	b		a
2	c		a
3	e	f	a
4		f	d
5			d
6			d
7	g		

VLIW Code

Cycle	**ALU1**	**L/S**	**MUL**
1	b		a
2	c		
3	e		
4		f	d
5			
6			
7	g		

5.4.5 Local vs Global Scheduling

In modern compilers, especially for statically scheduled architectures like VLIW, **instruction scheduling can be applied at two levels**:

- **Local Scheduling: Within Basic Blocks**. **Local scheduling** operates **inside a single basic block**, where:
  - The control flow is **linear** (no internal branches).
  - The compiler has **full knowledge** of all instructions and dependencies.

  🎯 The **objective** is to:
  - Maximize instruction-level parallelism (ILP).
  - Minimize execution time of each block.
  - Efficiently fill VLIW slots with real instructions (reduce NOPs).

  🛠 **Techniques**
  - ASAP scheduling (page 273).
  - List-based scheduling (page 275).
  - Loop unrolling (page 279).
  - Software pipelining (page 283).

  Local scheduling is **simpler** and **effective**, but limited:

  ✖ It **cannot move instructions** across block boundaries.

  ✖ **ILP is bounded** by the size and dependencies of the block.

- **Global Scheduling: Across Basic Blocks**. **Global Scheduling** considers a **wider scope**, across **multiple basic blocks**, allowing reordering of instructions:
  - Across **branches**.
  - Across **loop boundaries**.
  - Even across **entire control paths**.

  🎯 The **goal** is to:
  - Exploit **more ILP than local scheduling** allows.
  - Hide or overlap long-latency operations.
  - Restructure code for better pipeline utilization.

  🚫 **Challenges**
  - Must preserve **control and data dependences**.
  - Must generate **compensation code** to fix incorrect speculative moves.
  - More **complex** and **expensive** to implement in the compiler.

Technique	Type	Description
Loop Unrolling	Local	Replicates loop body to increase basic block size and expose ILP.
Software Pipelining	Local	Reorganizes instructions from different iterations to overlap them.
Trace Scheduling	Global	Predicts a frequent path (trace) and aggressively schedules it.
Superblock Scheduling	Global	Extension of trace scheduling; allows multiple exits, single entry.

Table 23: Techniques Overview.

5.4.6 Local Scheduling Techniques

5.4.6.1 Loop Unrolling

Loop Unrolling is a **compiler optimization** that duplicates the **body of a loop multiple times**, reducing the number of loop control instructions and exposing **more parallelism** within the loop body.

What is the goal?

- To **increase the size of the basic block** inside a loop.
- To **expose more independent instructions** to the scheduler.
- To **reduce the number of branches and loop counters**, lowering control.

Benefits

1. **More instruction-level parallelism (ILP)**, more instructions can be scheduled in parallel.
2. **Reduced loop overhead**, fewer `BNE`, `SUBI`, etc., executed per iteration.
3. **Better resource utilization**, more functional units are kept busy (fewer NOPs).

Downsides

1. **Increased register pressure**, more temporary values, so more registers needed.
2. **Increased code size**, can lead to **instruction cache misses**.

Example 4: No Unrolling vs. Loop Unrolling

Consider the following code:

```
for (i=1000; i>0; i--)
    x[i] = x[i] + s;

```

- No Unrolling, one iteration in Assembly:

```
Loop: LD     F0, 0(R1)
      ADD    F4, F0, F2
      SD     F4, 0(R1)
      SUBI   R1, R1, 8
      BNE    R1, R2, Loop
```

  - Execution time: **10 cycles per iteration**
  - Efficiency: 5 instructions, 10 cycles, then **50%** (IPC = 0.5)

- 4× Loop Unrolling (not yet optimized):

```
Loop: LD   F0,    0(R1)
      ADD  F4,    F0, F2
      SD   F4,    0(R1)

      LD   F0,   -8(R1)
      ADD  F4,   F0, F2
      SD   F4,   -8(R1)

      LD   F0, -16(R1)
      ADD  F4,   F0, F2
      SD   F4, -16(R1)

      LD   F0, -24(R1)
      ADD  F4,   F0, F2
      SD   F4, -24(R1)

      SUBI R1, R1, 32
      BNE  R1, R2, Loop
```

Only **true dependences** remain between `SUBI` and `BNE`. But there's a problem: we are **reusing** `F0`, `F4`, and leads to **WAW and WAR hazards**.

Register Renaming to Resolve Name Dependencies

As we saw in the previous example, there are WAW and WAR hazards due to the use of the same register during loop unrolling. To remove false dependencies (WAW and WAR), the compiler performs **register renaming**:

```
Loop: LD   F0,    0(R1)
      ADD  F4,    F0, F2
      SD   F4,    0(R1)

      LD   F6,   -8(R1)
      ADD  F8,   F6, F2
      SD   F8,   -8(R1)

      LD   F10, -16(R1)
      ADD  F12, F10, F2
      SD   F12, -16(R1)

      LD   F14, -24(R1)
      ADD  F16, F14, F2
      SD   F16, -24(R1)

      SUBI R1, R1, 32
      BNE  R1, R2, Loop
```

Now all instructions use **unique registers**, and only **true dependencies remain**. It allows **maximum scheduling freedom**.

- **Execution time**: 16 cycles per 4 iterations, then 4 cycles per iteration.
- **Loop overhead**: 4 cycles per 4 iterations, then 1 cycle per iteration.
- **Efficiency**: 14 instructions, 16 cycles, then $14 \div 16 = 87.5\%$

Effect on ILP After renaming, instructions are independent and can be **reordered or scheduled in parallel**:

- Original loop: 1 useful instruction per cycle.
- After unrolling and renaming: up to 4 useful instructions per cycle (in theory)

This enables **better pipelining**, **more efficient use of wide VLIW issue slots**, and **less stalling**.

Concept	Description
Loop unrolling	Duplicates loop body to expose more ILP.
Benefit	Reduces loop overhead, increases block size.
Drawback	Increases register pressure and code size.
Register renaming	Eliminates name dependencies (WAW/WAR) to enable parallel scheduling.
Result	More efficient and parallel scheduling possible.

Performance Metrics of Unrolling

When applying **loop unrolling**, the goal is not just to make the code longer, it's to **execute faster**. We can measure this improvement through several performance metrics:

- **Execution Time**. The total **execution time per loop iteration** reflects how efficiently the instructions are executed.
  - Execution time **includes** useful operations **plus stalls/NOPs**.
  - Lower execution time = better use of ILP and scheduling.

  We typically express it as:

$$\text{Execution time} = \frac{\text{Cycles to execute unrolled loop}}{\text{Number of iterations covered}} \tag{66}$$

- **Loop Overhead**. Loop overhead includes **instructions that are not part of the loop's core computation**:
  - `SUBI` (loop counter update).
  - `BNE` (branch).
  - Any `NOP` needed to avoid hazards.

  Reducing overhead:
  - Makes more room for **useful instructions**.
  - Boosts **efficiency**.

  In unrolling we do 4× the work **but only 1 loop update**, loop overhead is **amortized** over more iterations.

- **Code Efficiency**. Code efficiency **quantifies how well instruction slots are used**. Defined as:

$$\text{Efficiency} = \frac{\text{Number of useful operations}}{\text{Total number of instruction slots}} \tag{67}$$

This tells us **how many issue slots are actually contributing to the computation**.

5.4.6.2 Software Pipelining

Software Pipelining is a scheduling technique where instructions from **different iterations** of a loop are **reordered and overlapped** to keep functional units busy every cycle.

Instead of executing one full iteration at a time, the processor **starts a new iteration every cycle**, with different stages of previous iterations still in progress, similar to hardware pipelining.

⚖ Comparison with Loop Unrolling

Aspect	Loop Unrolling	Software Pipelining
Structure	Duplicates code	Keeps loop body same size
ILP exposure	Gained by replicating loop body	Gained by reordering across iterations
Code size	Increases (proportional to unrolling factor)	Remains compact
Scheduling scope	Within a block	Across multiple iterations
Register pressure	High (due to duplication)	Very high (many live temporaries)
Throughput	Good (bounded by unrolling factor)	Potentially optimal (1 iteration per cycle)

Table 24: Comparison with Loop Unrolling.

☰ Phases of Software Pipelined Loop

A software-pipelined loop executes in **three distinct phases**:

1. **Startup (Prologue)**. Initial cycles where instructions from the **first few iterations** begin execution. Not all stages are filled yet.
2. **Steady State**. The **core phase**: every cycle completes one iteration, while other iterations are in progress. This is where **ILP is fully exploited**.
3. **Drain (Epilogue)**. Final cycles where instructions from the **last few iterations** finish. No new iterations start; pipeline "drains" out.

Concept	Description
Software pipelining	Reorders loop instructions from different iterations.
Objective	Maximize ILP by filling the execution pipeline every cycle.
Benefits	Compact code, steady throughput, ideal for VLIW.
Trade-offs	High register pressure, scheduling complexity.
Execution phases	Startup (filling), Steady (full overlap), Drain (finishing up).

Example 5: Software Pipelining

We start with the original scalar loop:

```
Loop: LD    F0, 0(R1)
      ADD   F4, F0, F2
      SD    F4, 0(R1)
      SUBI  R1, R1, 8
      BNE   R1, R2, Loop
```

1. Loads a value from memory.
2. Adds a scalar value `F2`.
3. Stores it back.
4. Updates the pointer.
5. Branches if not done.

Execution time per iteration: 5-6 cycles. So only **one useful result per loop iteration**, poor ILP.

To apply software pipelining, we **reorder instructions** from **different iterations** to **fill functional units**. Assume all dependencies are respected (RAW), and rename registers to avoid WAR/WAW hazards.

```
LD    F0, 0(R1)      # I0
ADD   F4, F0, F2     # I1 from previous iteration
SD    F4, -8(R1)     # I2 from 2 iterations ago
```

This version performs `LD` **for current iteration**, `ADD` **for previous**, `SD` **for 2 iterations ago**. This matches the **steady state** of a pipelined execution: **each cycle does useful work from 3 iterations simultaneously**. We now issue **one full result per cycle** after startup.

Now let's handle address computation correctly. Assume:

- `R1` points to current element.
- We track offset correctly for each instruction.

```
# Prologue (Startup)
LD    F0, 0(R1)

# Loop body (Steady state)
Loop: LD    F6, -8(R1)       # LD for next iteration
      ADD   F4, F0, F2       # ADD from previous iteration
      SD    F4, 0(R1)        # SD from 2 iterations ago
      SUBI  R1, R1, 8
      BNE   R1, R2, Loop
      MOV   F0, F6           # prepare F0 for next ADD
```

Operation	From Iteration
LD F6	i + 1
ADD F4	i
SD F4	i - 1

- F0 is shifted forward via MOV.
- Pipelined stages are handled via register flow.
- Control remains simple, and code is compact.

Version	Throughput	Notes
Classic Loop	1 result every 5 cycles	Poor resource utilization.
Pipelined (steady)	**1 result per cycle**	Maximized ILP.
Code size	Small (no unrolling)	More compact than unrolling.

5.4.7 Global Scheduling

5.4.7.1 Trace Scheduling

Trace Scheduling is a **global instruction scheduling technique** that aims to optimize the **most frequently executed paths** (called **traces**) through a program's control flow graph (CFG).

It allows the compiler to move instructions **across basic block boundaries**, including across branches, to expose more **instruction-level parallelism (ILP)** and fill wide VLIW instruction words more efficiently.

- **Step 1: Trace Selection**. A **trace** is a likely path through the code, a **linear sequence of basic blocks** that are expected to execute one after the other **most of the time**.

  How traces are selected

  - Based on **profiling information**, static heuristics, or feedback from runs.
  - Hot paths (e.g., common branches of `if-else` or main loop bodies) are prioritized.
  - A trace may include: **forward branches**, **backward branches**, multiple blocks treated as a **single scheduling unit**.

  We can think of a trace as a **"super-block" of code** that is optimized as a single unit, assuming that it is frequently executed in that order.

- **Step 2: Trace Compaction (Scheduling)**. Once the trace is selected, the compiler:

  - Builds a **dependence graph** for the entire trace.
  - Applies **list-based scheduling** across all blocks in the trace.
  - Moves instructions **upward** across branches if possible, to fill unused slots (VLIW-friendly).

  This reordering allows:

  ✔ Filling of issue slots across multiple basic blocks.
  ✔ Earlier execution of long-latency operations (e.g., memory loads).

Speculation and Compensation Code

Problem. When instructions are **moved across branches**, they might be executed **in paths where they shouldn't**. This is called **Specualtive Execution**:

- The compiler **speculates** that a certain path is taken.
- It moves instructions **before a branch**, even if they are only **valid in one branch**.

Solution: Compensation Code. To **preserve correctness**, the compiler adds **compensation code** in **off-trace blocks** to:

- Recompute or restore the correct values.
- Prevent side effects (e.g., stores) from speculates

Definition 1: Compensation Code

Compensation Code is **extra code inserted by the compiler** to **preserve program correctness** after performing speculative instruction movement during global scheduling (e.g., in trace scheduling).

Example 6: Compensation Code

Suppose this trace is selected:

```
if (x > 0)
    A = B + C;  // part of the trace
else
    A = D + E
```

Compiler speculatively moves `A = B + C` above the `if`. Then in the `else` block, it inserts:

```
A = D + E;       // compensation code
```

This ensures correctness **if the speculation fails**.

Step	Action
Trace Selection	Choose most likely sequence of basic blocks (the "hot path").
Trace Compaction	Reorder instructions across blocks to improve ILP and scheduling.
Speculation	Move instructions before branches, assuming likely path.
Compensation Code	Inserted to preserve semantics if speculation was wrong.

Table 25: Trace Scheduling summary.

5.4.7.2 Superblock Scheduling

Superblock Scheduling is an evolution of **Trace Scheduling**, designed to simplify speculation management while still providing **high ILP** for **VLIW** or statically scheduled superscalar processors.

It is a **global instruction scheduling technique** that extends Trace Scheduling, with two key design changes:

1. It restricts code regions to have **only one entry point**.
2. It allows **multiple exits**.

This enables the compiler to **aggressively schedule instructions across basic blocks**, including across branches, but with **simpler and safer control-flow management** than full trace scheduling.

⚖ Superblocks vs Traces

Feature	**Trace** Scheduling	**Superblock** Scheduling
Entry points	Multiple (any predecessor block).	**Single entry point** (one predecessor).
Exit points	Multiple.	Multiple.
Control flow	General, complex.	**Simplified, controlled**.
Speculation	Really aggressive, harder to manage.	More manageable.
Compensation code	May be needed on both sides of branches.	Needed only on **one side** (outside the superblock).

❓ Why is this useful?

In trace scheduling, we may have **multiple entry points** into the trace, which means any speculative movement of instructions must be **repaired in many paths** (with compensation code). In Superblock Scheduling, by **enforcing a single-entry rule**, and using **tail duplication** to eliminate side entries, the compiler can:

- ✔ Speculate more **safely**.
- ✔ Insert **fewer compensation instructions**.
- ✔ Retain most of the **performance benefit** of traces.
- ✔ Keep control flow **more manageable**.

Typical Structure of a Superblock

A **superblock** is a **linear sequence of basic blocks** that:

- Starts from a **unique entry point**.
- Ends at **multiple potential exists**.
- Is built around a **frequently executed path** (like a hot loop body or branch).

How is it built?

1. **Profiling** (or prediction) selects a hot path.
2. **Tail Duplication** is applied to remove alternative entries. Compiler **duplicates the tails** (exit blocks) of a region, because this ensures all control flow into the superblock happens from **a single point**.
3. The resulting region becomes a **superblock** (safe to schedule aggressively).

We can think of it as a **safer, more controlled trace**, scheduled to maximize instruction-level parallelism (ILP) on VLIW or superscalar machines.

Example 7: Superblock

Original code:

```
if (x > 0)
    A = B + C;
else
    A = D + E;
```

We want to **schedule more instructions in parallel**. But the compiler sees a branch:

- In one path: `A = B + C`
- In the other: `A = D + E`

To do **Superblock Scheduling**, we assume that **the first path** (`x > 0`) **is more frequent** (we get this from profiling).

1. We take the **most frequent path** (`x > 0`).
2. We include its block: `A = B + C`.
3. We move it **before the branch**, so it's scheduled early (speculative move).
4. We make the region a **superblock**: a linear code region with one entry.

```
A = B + C          // speculated! Scheduled early
if (x <= 0)
    A = D + E      // compensation code
```

Now:

- If the branch is *not taken* (`x > 0`), then `A = B + C` is correct.
- If the branch *is taken* (`x` $\leq$ `0`), then we **fix** `A` using compensation code.

It is called a "superblock" because we treat that piece of code as one large block when scheduling. The compiler: optimizes this region together, tries to issue as many instructions per cycle as possible, and assumes most of the time the "hot path" will be followed.

6 Advanced Memory

6.1 Introduction

Modern processors operate at very high speeds, and any delay in accessing data can drastically reduce performance. Yet, fast memory is expensive and limited in size, while large memory is slow. This chapter provides strategies to **bridge the gap** between processor speed and memory latency, introducing techniques to **optimize access time, reduce misses, and manage complexity**.

Understanding **advanced memory hierarchy** allows us to:

- Reduce the **Average Memory Access Time (AMAT)**.
- Optimize system performance while managing cost and power.
- Design efficient **cache architectures**, essential in high-performance systems (from embedded CPUs to data centers)

Motivation for Hierarchical Memory

Programmers naturally wish for: "*an unlimited amount of memory, as fast as the CPU*". But this ideal is physically and economically unfeasible:

- **Fast memories** (SDRAM) are **very expensive** and **power-hungry**.
- **Slow memories** (DRAM) are **cheaper** and **larger**, but much **slower**.

Solution: A memory hierarchy. Build a layered system of memories, where each layer:

- Gets **faster** and **smaller** as we go **upward**.
- Gets **slower** and **larger** as we go **downward**.

Each upper level stores a **subset** of data from the lower level. This way, **frequently accessed data** stays in **fast memory (cache)** and **less frequent data** is stored deeper (L2, L3, DRAM, Disk).

Illusion of a Large, Fast Memory

Even though the real system consists of many memory levels, **the processor "sees" a single unified memory**. Thanks to the hierarchy and locality principles:

- Most accesses hit the **fast upper-level caches**.
- Rarely used data is retrieved from lower levels (with longer latency).

This *illusion* is what enables high-speed execution even with cost-effective memory.

There is a sort of hierarchy with different levels of access to memory benefits. Do developers manually choose fast or slow memory?

No, typically **developer do** *not* **manually choose** which memory level (L1, L2, DRAM, etc.) stores the data. Instead, the **hardware and compiler manage this automatically**, using a memory hierarchy. *So what's going on?*

- The developer writes code as if there's **one big memory**.
- The system **automatically** stores:
  - Frequently used data in **fast memory (L1 cache)**.
  - Less-used data in **slower memory (L2/L3 cache or RAM)**.
- The **processor checks the fast levels first** and, only if needed, moves to slower levels.

This is what we call the **illusion of a large and fast memory**.

How is locality linked to the hierarchy?

Locality is a **behavioral pattern of how programs access memory**. The **memory hierarchy is *designed to exploit locality***. There are two types of locality in programs:

- **Temporal Locality**: "***if we used this data recently, we will likely use it again soon***". For example:

```
for (int i = 0; i < 100; i++) {
    sum += a[i];   // temporal locality for variable ‘sum‘
}
```

  Where `sum` is read and written **many times in a short span**, so **keep it in the L1 cache** to avoid loading from RAM every time.

- **Spatial Locality**: "***if we access this memory address, we will probably access nearby ones soon***". For example:

```
for (int i = 0; i < 1000; i++) {
    // spatial locality: accessing sequential array elements
    x = arr[i];
}
```

  If we access `arr[0]`, we will soon access `arr[1]`, `arr[2]`, etc. The system loads **entire blocks** (not just one word) into cache, anticipating the next accesses.

The hierarchy is effective only if programs exhibit locality. Fast memory (caches) are **small**, so they **rely on locality** to keep relevant data. Without locality, caches would be useless: every access would go to slow memory.

Example 1: Memory Hierarchy Analogy

Think of a **bookshelf system**:

- We keep our most-used books on our desk (L1).
- Others in our room shelf (L2).
- The rest in the university library (main memory).
- The rarest in the city archive (disk).

If we're smart about what to keep close (using locality), we'll avoid most slow trips.

6.2 Principle of Locality

Cache memories are effective because programs tend to access data and instructions in **predictable patterns**. These patterns are captured by the **Principle of Locality**:

- **Temporal Locality** ("*Time-based reuse*"). **If a memory location is accessed, it's likely to be accessed again *soon***. So recently accessed data stays in cache, because it's likely to be accessed again.

  For example, repeated access to loop variables:

```
for (int i = 0; i < n; ++i) {
    sum += a[i];
}
```

  Or reuse of stack frames in recursive functions or instruction fetches inside tight loops.

  > **Cache strategy.** Keep recently accessed **blocks** in cache. Don't replace them unless absolutely necessary (see LRU in later sections).

- **Spatial Locality** ("*Nearby reuse*"). **If a memory location is accessed, it's likely that *nearby locations* will be accessed soon**. In other words, if we need `a[i]`, we'll probably also need `a[i+1]`, `a[i+2]`, etc.

  For example, sequential instruction execution, or accessing elements of an array, or scanning a matrix row by row.

  > **Cache strategy.** Fetch **entire blocks**, not single words (e.g., 64-byte lines). Use **block size > 1 word** to bring adjacent memory into cache.

How Caches Exploit Locality

With **Cache Block** we refer to the smallest unit of data moved from main memory to cache. It is typically 32, 64 or 128 bytes. It helps bring **spatial neighbors** into the cache.

Locality Type	Cache Mechanism
Temporal	Keep recently accessed data in cache
Spatial	Fetch blocks (containing multiple nearby elements)

Most modern cache architectures assume that the programmer writes code that follows locality principles, which i why **compiler optimizations and programming style** matter.

6.3 So, what is a Cache?

Definition 1: Cache

A **Cache** is a small, fast **memory** located **close to the processor** that temporarily **stores copies of data from the main memory**, aiming to reduce the average time needed to access data and instructions.

Modern memory systems are structured hierarchically:

- **Cache (upper level)**: small, fast, but expensive (SRAM). It is very close to the CPU and stores a subset of main memory to reduce access latency.
- **Main memory (lower level)**: larger, slower, but cheaper (DRAM). It stores the complete working set and is accessed only if data is not found in cache.

The cache acts as a **buffer** between fast CPU and slow main memory.

Terminology

- **Cache Block** or **Cache Line**. The smallest unit of data transferred between memory and cache. It is typically 32 to 128 bytes. Since a cache is divided into multiple blocks, we use:

$$\text{Number of blocks} = \frac{\text{Cache size}}{\text{Block size}} \tag{68}$$

To calculate the number of blocks in the cache.

Example 2

With a 64 KB cache, and 16-byte per block:

$$\frac{64 \times 1024}{16} = 4096 \,\text{blocks}$$

✔ **Cache Hit**. Occurs when the requested memory address is **already present in the cache**. Fast access, minimal latency.

✖ **Cache Miss**. Occurs when the requested address is **no in the cache**. The block must be fetched from lower-level memory.

Consequences of a miss

1. **Stall** CPU.
2. Fetch block from memory.
3. **Write** block to cache.
4. Retry access (which now becomes a hit).

- **Hit Time**. It is the time to:

  1. Access the cache.
  2. Determine if it's a hit.
  3. Return data (if present).

  It is fast, typically 1-3 CPU cycles.

- **Miss Penalty**. It is the time to:

  1. Fetch the block from main memory (or next-level cache).
  2. Update cache.
  3. Resume the CPU.

  Much **slower** than hit time, often 10s to 100s of cycles.

- **Average Memory Access Time (AMAT)**

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty} \tag{69}$$

  This is the **expected cost of accessing memory**, accounting for both hits and misses.

Example 3

For example:

- Hit Time: 1 cycle.
- Miss Rate: 5%.
- Miss Penalty: 100 cycles.

$$\text{AMAT} = 1 + 0.05 \times 100 = 6 \text{ cycles}$$

6.4 Cache Performance Metrics

This section is about quantifying how effective a cache is, and how to improve it. Similar topics were covered on page 251.

The **Average Memory Access Time (AMAT)** is the average time the CPU needs to access memory, considering both hits and misses. So, when accessing memory, we either get a **hit** or a **miss**:

$$\begin{aligned} \text{AMAT} &= (\text{Probability of Hit}) \cdot (\text{Time if Hit}) + \\ &\quad (\text{Probability of Miss}) \cdot (\text{Time if Miss}) \end{aligned}$$

Which is written as:

$$\text{AMAT} = \text{Hit Rate} \cdot \text{Hit Time} + \text{Miss Rate} \cdot \text{Miss Time} \tag{70}$$

This is the **most general formula**, where:

- **Hit Rate**: fraction of memory accesses that result in a hit.
- **Hit Time**: time to check cache and return data on a hit.
- **Miss Rate**: fraction of memory accesses that result in a miss.
- **Miss Time**: time to
  1. **Check** the **cache** (same time as a *Hit Time*).
  2. **Fetch** the **data from main memory** (same time as a *Miss Penalty*)

  So it is reasonable to write the Miss Time as follows:

$$\text{Miss Time} = \text{Hit Time} + \text{Miss Penalty} \tag{71}$$

However, if the Miss Time is equal to the sum of the Hit Time and the Miss Penalty, then the most general formula can be **simplified**:

$$\begin{aligned} \text{AMAT} &= \text{Hit Rate} \cdot \text{Hit Time} + \text{Miss Rate} \cdot \text{Miss Time} \\ &= \text{Hit Rate} \cdot \text{Hit Time} + \text{Miss Rate} \cdot \\ &\quad (\text{Hit Time} + \text{Miss Penalty}) \\ &= \text{Hit Rate} \cdot \text{Hit Time} + \text{Miss Rate} \cdot \text{Hit Time} + \\ &\quad \text{Miss Rate} \cdot \text{Miss Penalty} \\ &= \text{Hit Time} \cdot \underbrace{(\text{Hit Rate} + \text{Miss Rate})}_{\%\,\text{Miss} + \%\,\text{Hit} = 100\% = 1} + \text{Miss Rate} \cdot \text{Miss Penalty} \end{aligned}$$

So the **simplified and most common** version is:

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty} \tag{72}$$

Even if a cache **misses**, we still pay the **Hit Time**, because we have to *check* the cache to know that it's a miss. So every access:

- 👎 Pays **Hit Time**.
- 👎 Plus, if it's a miss, it pays **Miss Penalty**.

6.5 Cache Architecture

Each **Cache Block** (also called **Cache Line**) is a row in the cache that stores part of memory.

A typical cache line has **three fields**:

- **Valid Bit**: Indicates if the **block contains valid** (i.e., initialized) **data**.
- **Tag**: The high-order bits (most significant bits, leftmost side) of the memory address, used to **identify the block**.
- **Data**: The actual words/bytes of memory copied into the cache line.

At startup, **all valid bits = 0** (cache is empty).

Example 4: Visual Representation

V	TAG	DATA
1	0x001AE	[Word0, Word1, Word2...]

- `V`: valid bit.
- `TAG`: identifies *which memory block* this is.
- `DATA`: contains the content of that memory block (usually several words).

Why the TAG?

Let's say the CPU requests a memory address like `0x1AE023`. The cache checks:

1. **Index**: determines (extract) which row to look in.
2. **Tag**: is this row the one holding `0x1AE`?
3. **Valid**: is the content even initialized?

If both Tag and Valid match, it's a **hit**.

Four key cache design questions

- **Block placement** - ***Where can a block be placed?*** Page 299.
- **Block identification** - ***How is a block found?*** Page 302.
- **Replacement strategy** - ***Which block should be replaced?*** Page 303.
- **Write strategy** - ***What happens on a write?*** Page 305.

6.5.1 Block Placement: *Where can a block be placed?*

This question is about how memory addresses map to positions in the cache. It defines the **mapping policy** between **main memory blocks** and **cache lines**.

There are **three classic cache organizations**:

1. **Direct-Mapped Cache**. It is the simplest and most intuitive. Each block from memory **can go in only one location** in the cache.

$$\text{Cache Index} = (\text{Block Address}) \mod (\text{Number of Cache Blocks}) \quad (73)$$

   This means:

   - ✅ Easy and **fast lookup** (just compute index).
   - ❌ But high risk of **conflict misses** (collision): two blocks that map to the same index will evict each other repeatedly.

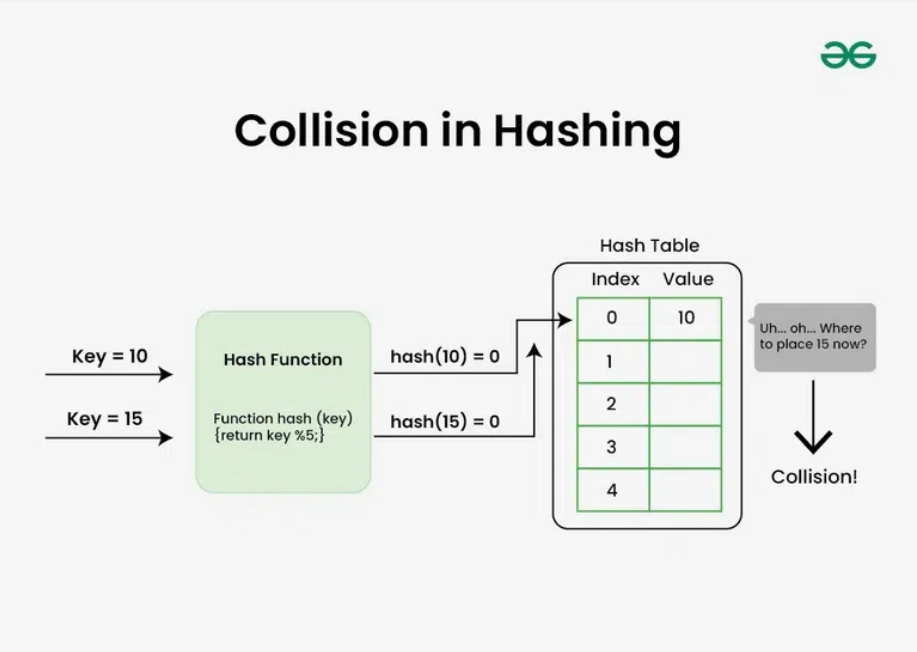


Figure 28: A great analogy is in a hashing environment, where a collision occurs if a key is the same. In our case, it's the same: same index, collision! [1]

Example 5: Direct-Mapped Cache

Cache has 8 blocks: indexes from 0 to 7. The memory block of 12 is calculated as follows:

$$12 \mod 8 = 4$$

And it can only go in **cache block 4**.

Memory Block	Cache Block
4	4
12	4
20	4

So if we alternate access to blocks 12 and 20, we get repeated evictions, then poor performance.

2. **Fully Associative Cache**. A block from memory **can be placed in any cache line**. There is no index in the address (no needed), and the mapping rule is a **full `TAG` comparison across all cache lines**. This means:

   - ✅ **No conflict misses** (since any block can go anywhere).
   - ✅ Ideal flexibility.
   - ❌ **Expensive** hardware: must compare TAGs for *all* cache lines. Because to do this, the hardware includes:
     - N **comparators**, each comparing the input tag with the tag stored in one of the N cache lines.
     - A **priority encoder** or logic to select the matching line (if any).
   - ❌ Slower access.

3. **N-Way Set-Associative Cache**. A compromise between direct-mapped and fully associative.

   - Cache is divided into **sets**.
   - Each set contains n **lines** (ways).
   - A memory block maps to **exactly one set**, but can be placed in **any of the n lines** within that set.

   The mapping rule is:

   $$\text{Set Index} = (\text{Block Address}) \mod (\text{Number of Sets}) \tag{74}$$

   This reduces conflict misses **while keeping lookup cost low** (only compare tags inside the set).

Example 6: N-Way Set-Associative Cache

Imagine a 2-way set-associative cache with 4 sets (each set has 2 blocks). For example, the **block 12** goes into **Set 0**:

$$12 \mod 4 = 0$$

In Set 0 it can go into **either Way 0 or Way 1**.

Memory Block	Set Index
0	0
4	0
8	0
12	0

Each of these blocks competes for 2 slots in Set 0, so **fewer conflicts** than direct-mapped.

Summary of Block Placement Policies:

- **Direct-Mapped**

Mapping Rule	Flexibility	Hardware Cost	Risk of Conflict Miss
Block Address mod #Blocks	Rigid (1 line)	Simple	High

- **Fully Associative**

Mapping Rule	Flexibility	Hardware Cost	Risk of Conflict Miss
Any block to any line (compare all)	Full Freedom	Expensive	None

- **N-Way Set-Associative**

Mapping Rule	Flexibility	Hardware Cost	Risk of Conflict Miss
Block Address mod #Sets	Medium	Moderate	Controlled

6.5.2 Block Identification: *How is a block found?*

When the CPU requests a memory address, the cache must **quickly determine** whether that address is stored inside it, and if so, in which line. This process depends on the **cache mapping type** that we studied in the previous section.

General Method

Regardless of mapping type, the process involves:

1. **Index**. Determines *where to look* (which set or which specific line in direct-mapped caches).
2. **Tag**. Stored in the cache line; must match the tag bits extracted from the CPU's requested address.
3. **Valid Bit**. Ensures the entry contains real, initialized data.

If **index matches**, **tag matches**, and **valid bit = 1**, we have a *hit*. Otherwise a *miss*.

How it works in each mapping type

- **Direct-Mapped Cache**
  1. Use **index bits** from the address to locate exactly **one cache line**.
  2. Compare **tag bits** in that line to the requested address's tag.
  3. Check **valid bit**.
  4. **Hit** if both match; otherwise, **miss**.
- **Set-Associative Cache**
  1. Use **index bits** to select the **set**. Inside that set, there are n **lines** (ways).
  2. Compare n **lines** of each way in that set in parallel.
  3. If one matches and **valid** $= 1 \rightarrow$ **hit**; otherwise, **miss**.
- **Fully Associative Cache**
  1. **No index** bits; the whole cache is **one big set**.
  2. Compare the requested tag to **every tag** in the cache in parallel.
  3. **Hit** if any match with **valid** $= 1$; otherwise **miss**.

Mapping Type	How Block is Found
Direct-Mapped	Use index to find 1 line, compare its tag.
Set-Associative	Use index to find 1 set, compare tags of n ways in that set.
Fully Associative	Compare tag with every line in the cache.

6.5.3 Replacement Strategy: *Which block should be replaced?*

When a cache miss occurs, the cache must bring a new block from the lower memory level. If the cache (or the relevant set) is **full**, one of the existing blocks must be **evicted** to make space. But, *which block should be replaced?*

Depends on Mapping Type

This decision depends on the type of mapping we have in the cache. With a direct-mapped cache, the answer is clear: always replace. However, with a set-associative or fully associative cache, it depends on the replacement policy.

1. **Direct-Mapped Cache**. No choice: the mapping rule already determines **exactly one line** for the block. The existing block in that line is **always replaced**. So more conflict misses.
2. **Set-Associative** or **Fully Associative Cache**. There are multiple possible slots (n lines in the set, or all lines for fully associative). A **replacement policy** decides which one to evict. A good replacement policy (like LRU, see below) can significantly reduce misses, especially in workloads with strong temporal locality.

Common Replacement Policies

A **Set-Associative** or **Fully Associative Cache** can adopt one of the following common replacement policies:

1. **Random Replacement**. Pick any block in the set at random.
   - ✔ **Simple hardware** and **fast**.
   - ✔ **Avoids always evicting the same block** in some pathological patterns.
   - ✘ It **doesn't exploit locality** because it is random and doesn't follow a heuristic. It **may evict a frequently used block**.
2. **FIFO (First-In First-Out)**. Evict the block that has been in the cache the longest (oldest arrival).
   - ✔ **Easy to implement** with a queue per set.
   - ✘ **Ignores recent usage**, so might **evict a frequently used block** if it's old.
3. **LRU (Least Recently Used)**. Evict the block that has **not been used for the longest time**.
   - ✔ Matches the idea of **temporal locality**: recently used data is likely to be used again.
   - ✘ More **complex hardware**: must track usage order for each block in the set.

Example 7: 2-Way Set-Associative Cache (LRU Policy)

Set 0 has:

- Way 0 → Block 4 (last used 5 cycles ago).
- Way 1 → Block 8 (last used 20 cycles ago).

CPU requests Block 12 (maps to Set 0), miss occurs. LRU chooses Way 1 (Block 8) to replace, since it was used least recently.

6.5.4 Write Strategy: *What happens on a write?*

When the CPU writes to a memory address and the block is cached, we must decide:

1. **Where does the write go?** (*Write Policies*)
   - Only in the cache?
   - In both cache and main memory?
2. **What if the address is not in the cache?** (*Write Miss Policies*)
   - Do we bring it into the cache before writing?
   - Or write directly to memory without caching it?

Step 1: Write Policies (*Where does the write go?*)

- **Write-Through**. The value is written **to both**:
  - The cache block (if present).
  - The corresponding location in main memory.

  This ensures that the **memory** is always **up to date**. But it needs a **write buffer** to avoid stalling the CPU while memory is updated.

  ✔ **Simpler** to implement.
  ✔ Memory always **coherent** with cache.
  ✘ More **memory traffic** (every write goes to memory).
  ✘ **Slower** overall **if no write buffer**.
- **Write-Back**. The value is written **only to cache**. The modified cache block is marked **dirty** (with a **Dirty Bit**). And the main memory is updated **only when the dirty block is evicted**.

  ✔ Less **memory traffic** (multiple writes to same block cost only one memory update).
  ✔ **Faster in write-intensive workloads**.
  ✘ More **complex** (need dirty bits and eviction logic).
  ✘ Main memory may be **out of date** until eviction.

Step 2: Write Miss Policies (*What if the block is NOT in the cache?*)

- **Write Allocate** (aka **Fetch on Write**). On a write miss:
  1. Load the block into the cache (same as a read miss).
  2. Then perform the write in the cache.

  Good for temporal locality: if we write once, we might write again soon.
- **No Write Allocate** (aka **Write Around**). On a write miss, **do not** load block into cache, but write directly to main memory. So, skip the cache and go directly to the main memory. **Good when writes are rare or sequential** (no need to keep data around).

Cache Policy	Write Miss Policy	Why
Write-Back	Write Allocate	Writes are kept in cache, so allocate makes sense.
Write-Through	No Write Allocate	Avoids unnecessary block fetch before write.

Table 26: Typical combinations in real systems.

Aspect	Write-Through	Write-Back
Write Location	Cache + Memory	Cache only
Memory Coherence	Always up to date	Updated only on eviction
Memory Traffic	High	Low
Complexity	Low	High (dirty bit)

Miss Policy	Write Allocate	No Write Allocate
Action on Miss	Bring block into cache	Write directly to memory
Best With	Write-Back	Write-Through

Write Operation Timeline

We'll use:

- **CPU**: issuing a write to memory address `X`.
- **Cache**: may or may not contain block `X`.
- **Memory**: slower DRAM.

1. **Write Hit** (data is already in cache).
   - **Write-Through**
     (a) CPU writes new data into cache line for block `X`.
     (b) Cache immediately sends the same write to main memory.
     (c) Memory is always **coherent** (same as cache).
     Latency: cache hit time (plus background memory update if write buffer exists).
     Memory traffic: **1 write** to memory.
   - **Write-Back**
     (a) CPU writes new data into cache line for block `X`.
     (b) Set **dirty bit** = **1** for that cache line.
     (c) Main memory is **not updated now**, will be updated when block `X` is evicted.
     Latency: just the cache hit time.
     Memory traffic: **0 writes** now (delayed until eviction).
2. **Write Miss** (data is not in cache).
   - **Case A: Write Allocate**
     – **Write-Back + Write Allocate** (*most common*):
       (a) Cache fetches block `X` from memory into a cache line.
       (b) CPU writes new data into cache line.
       (c) Dirty bit is set.
       (d) Memory is updated **later** when block is evicted.
       Memory traffic: **1 block read** (fetch) + **1 block write** later at eviction.
     – **Write-Through + Write Allocate**:
       (a) Cache fetches block `X` from memory into cache.
       (b) CPU writes into cache line.
       (c) Cache immediately writes to main memory as well.
       Memory traffic: **1 block read** + **1 word write** immediately.

- **Case B: No Write Allocate**
  - **Write-Through + No Write Allocate** (*common combination*):
    (a) Cache does not load block `X`.
    (b) CPU writes directly to memory.
    (c) Cache content is unchanged.
    Memory traffic: **1 word write** only.
  - **Write-Back + No Write Allocate**:
    (a) Cache does not load block `X`.
    (b) CPU writes directly to memory.
    (c) Cache content is unchanged.
    Memory traffic: **1 word write** only, no dirty bit set.

6.6 Miss Penalty Reduction

When a cache miss happens, the CPU must fetch the required block from a lower memory level, which takes much longer than a hit. This extra time is called the **Miss Penalty**. Because miss penalties can be **tens or hundreds of cycles**, **reducing them has a huge impact** on overall performance.

Role of the L2 Cache

The most common way to reduce miss penalty is to introduce a **Second-Level Cache (L2)** between the L1 cache and main memory.

Its main job is to **catch L1 cache misses** before they reach main memory. Since main memory is much slower than L1 or L2, resolving miss in L2 instead of DRAM **dramatically reduces the miss penalty**.

Larger than L1: typically hundreds of KB to several MB.

Slower than L1, but still **much faster than DRAM**.

- Usually **unified**: stores both instructions and data.
- Implemented in **SRAM** like L1, but with slightly longer access time.
- Can be **on-chip** (modern CPUs) or **off-chip** (older designs).

When an L1 miss occurs, instead of going directly to DRAM:

1. **CPU request** → goes to L1 cache.
2. If **L1 hit** → data returned immediately.
3. If **L1 miss** → request sent to L2 cache.
4. If **L2 hit** → data returned quickly to L1 (and CPU).
5. If **L2 miss** → data fetched from main memory (very slow).

It works because L1 caches are small (to keep them fast), so they miss more often. L2 caches are bigger and can hold more blocks, so they can keep data that L1 had to evict. This way, a **large fraction of L1 misses are "saved" by L2 before going to DRAM**.

AMAT in Presence of L2

Let's define:

- HT_1 = L1 hit time.
- MR_1 = L1 miss rate.
- HT_2 = L2 hit time.

- MR_2 = **L2 Local Miss Rate** (fraction of L1 misses that also miss in L2):

$$\text{MR}_2 = \frac{\text{L2 misses}}{\text{L1 misses}} \tag{75}$$

  It measures how good L2 is at catching L1 misses.

- MP_2 = Miss Penalty from L2 to main memory.
- $\text{MR}_{1,2}$ = **Global Miss Rate** (to main memory):

$$\text{MR}_{1,2} = \text{MR}_1 \times \text{MR}_2 = \frac{(\text{L2 misses})}{(\text{Total CPU Accesses})} \tag{76}$$

  This tells us how often we actually have to access main memory.

The formula becomes:

$$AMAT = \text{HT}_1 + \text{MR}_1 \times (\text{HT}_2 + \text{MR}_2 \times \text{MP}_2) \tag{77}$$

Every L1 misses costs at least an **L2 access time** (HT_2). If L2 also misses (MR_2), we pay the extra penalty of going to main memory (MP_2).

Example 8: AMAT in Presence of L2

Imagine:

- $\text{HT}_1 = 1$ cycle
- $\text{MR}_1 = 5\%$
- $\text{HT}_2 = 10$ cycles
- $\text{MR}_2 = 20\%$
- $\text{MP}_2 = 100$ cycles

Step-by-step:

1. L1 hit $\rightarrow$ 1 cycle.
2. L1 miss (5% of cases) $\rightarrow$ go to L2 (10 cycles).
3. In 20% of those L2 accesses $\rightarrow$ miss again and pay 100 cycles to access DRAM.

$$\begin{aligned} AMAT &= 1 + 0.05 \times (10 + 0.20 \times 100) \\ &= 1 + 0.05 \times (10 + 20) \\ &= 1 + 0.05 \times 30 \\ &= 1 + 1.5 = 2.5 \text{ cycles} \end{aligned}$$

Without L2:

$$AMAT = 1 + 0.05 \times 100 = 6 \text{ cycles}$$

L2 reduced the AMAT from **6 cycles** to **2.5 cycles**.

6.7 Design Space of Cache

Designing a cache involves tuning several parameters that all interact with each other. Changing one parameter usually impacts hit time, miss rate, and miss penalty, so it's always about trade-offs.

1. **Cache Size**
   - **Bigger cache**
     - ✔ Lower miss rate (can store more of the working set).
     - ✘ Higher hit time (more bits to check, longer wires, more complex lookup).
     - ✘ More area and power consumption.
   - **Smaller cache**
     - ✔ Faster hit time, lower power.
     - ✘ Higher miss rate.

   We can't make L1 huge without slowing it down, that's why big caches are placed at L2 or L3.

2. **Block Size (Cache Line Size)**
   - **Larger block**
     - ✔ Exploits spatial locality better (fetches more neighboring data on each miss).
     - ✘ Higher miss penalty (more bytes to transfer on a miss).
     - ✘ Possible increase in **conflict misses** if large blocks reduce the number of total blocks.
   - **Smaller block**
     - ✔ Lower miss penalty (less data to fetch per miss).
     - ✔ More blocks → potentially lower conflict misses.
     - ✘ Less spatial locality exploitation.

3. **Associativity**
   - **Direct-Mapped**
     - ✔ Simple, fast hit time.
     - ✘ High conflict miss rate.
   - **Fully Associative**
     - ✔ No conflict misses.
     - ✘ Slow and expensive to implement for large caches.
   - **N-Way Set Associative**
     - ✔ Balance between conflict misses and complexity.
     - ✘ Higher hit time than direct-mapped.

   General trend: Increasing associativity reduces **conflict misses** but increases **hit time** and hardware complexity.

4. **Replacement Policy**. Determines which block is evicted on a miss (in associative caches):

   - **LRU (Least Recently Used)**: good with temporal locality but expensive for high associativity.
   - **Random**: simple, avoids pathological patterns, but ignores usage.
   - **FIFO**: simple, but can evict still-hot blocks.

   Trend: For low associativity (2-4 ways), LRU is common; for high associativity, pseudo-LRU or random is used.

5. **Write Policy**. Two decisions here:

   - **Write-Through vs Write-Back**
     - Write-through: simpler, always update memory; higher memory traffic.
     - Write-back: update cache only, mark dirty; write to memory only on eviction.
   - **Write Allocate vs No Write Allocate**
     - Write Allocate: bring the block into cache on a write miss; better with write-back.
     - No Write Allocate: write directly to memory on a miss; better with write-through.

Parameter	Bigger/More	Smaller/Less
Cache size	↓ miss rate, ↑ hit time	↑ miss rate, ↓ hit time
Block size	↑ spatial locality, ↑ penalty	↓ penalty, ↓ spatial gain
Associativity	↓ conflict misses, ↑ hit time	↑ conflict misses, ↓ hit time
Replacement	↑ LRU accuracy, ↑ hardware cost	↓ complexity, ↑ random evictions
Write policy	↓ traffic (write-back), ↑ complexity	↑ traffic (write-through), ↓ complexity

Table 27: Design Trade-offs.

6.8 Cache Miss Classification

Whenever the CPU requests data, the **cache may or may not contain it**:

- ✔ If it's there ⇒ **Cache Hit**
- ✘ If it's not ⇒ **Cache Miss**

There are **different causes** for cache misses. Understanding their classification helps in designing **better memory hierarchies**.

The 3 Caches Model of Cache Misses

This classic classification includes:

- **Compulsory Misses** (aka **Cold Start Misses**) occurs the **first time** a block is accessed, so it's **not in the cache yet**. It occurs because the cache starts empty and we must load the block from main memory. For example:

```
// first time a[0] is accessed → compulsory miss
int x = a[0];
```

  These **always occur**, even with infinite cache. Can be **reduced with prefetching** or **larger blocks** (use spatial locality).

- **Capacity Misses**. The cache is **too small** to hold all the data the program is working on. So, **useful blocks are evicted** and later needed again, causing misses. For example:

```
for (i = 0; i < 100000; i++) {
    sum += arr[i];
}
```

  If the array `arr` is larger than the cache, older blocks will be replaced, causing misses later. Reducing these misses requires a **larger cache**. Still subject to **cost, power, and hit time trade-offs**.

- **Conflict Misses** (aka **Collision** or **Interference Misses**). Happens when **two or more blocks map to the same cache set**, and they overwrite each other, **even if the cache is not full**. Occurs in **direct-mapped caches** and **set-associative caches** with limited associativity. For example:

```
// Suppose these addresses map to the same set:
int a = mat[0][0];       // → cache set X
int b = mat[512][0];     // → same cache set X
```

  They keep evicting each other: **ping-pong effect**. Can be reduced by: **higher associativity**, **better mapping**, using **victim cashes**.

- (in multiprocessors) **Coherence Misses**. Introduced later with **shared-memory multiprocessors**. It occurs when **another processor modifies a cache block** that a core was using. The block must be **invalidated**, leading to a **miss on next access**. For example:
  - Processor A and B both cache variable `x`.
  - A writes to `x`, B's copy is invalidated.
  - B tries to read `x`, then coherence miss.

  These are handled by **cache coherence protocols** (like MESI), and it is only relevant in **multi-core systems**.

Type	Cause	How to reduce
Compulsory	First-time access	Larger blocks, prefetching
Capacity	Working set > cache size	Larger cache
Conflict	Multiple blocks map to the same location	Higher associativity, victim cache
Coherence	Invalidation from another processor	Coherence protocol (MESI, MOESI, etc.)

Table 28: Cache Miss Classification.

6.9 Improving Cache Performance

One of the main challenges in memory system design is achieving **low-latency memory access** while maintaining **high throughput and efficiency**. Since memory is a major performance bottleneck, architects aim to **minimize delays caused by cache misses**.

Key Metric: Average Memory Access Time (AMAT)

The **Average Memory Access Time (AMAT)** equation **defines how effective a memory hierarchy is**:

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty} \tag{78}$$

Each term in this formula captures a different aspect of performance:

- **Hit Time**: Time to access data in the cache (when it's a hit).
  The unit is cycles.
- **Miss Rate**: Fraction of memory accesses that result in a miss.
  The unit is %.
- **Miss Penalty**: Time to fetch data from the next memory level.
  The unit is cycles.

Goal: Minimize AMAT

We can improve cache performance through **three orthogonal strategies**, each targeting one component of AMAT:

1. **Reduce Miss Rate**. This means making the cache *more effective* in storing the right data. **Techniques include**:
   - Larger caches.
   - Larger blocks (but with care).
   - Higher associativity.
   - Software and hardware prefetching.
   - Compiler optimizations.

   Result: fewer accesses go to slower memory levels.
2. **Reduce Miss Penalty**. Once a miss happens, the delay can be high. We aim to **reduce the time spent waiting** by:
   - Prioritizing reads.
   - Using victim caches.
   - Sub-blocking.
   - Early restart or critical word first.
   - Second-level caches.

- Non-blocking caches.

Result: the *impact* of a miss is reduced.

3. **Reduce Hit Time**. Even on a hit, we want the **fastest possible access**. This can be achieved by:
   - Making the L1 cache small and simple.
   - Avoiding address translation delays.
   - Using pipelined writes.
   - Virtually-indexed, physically-tagged caches.

   Result: faster access when hit, then lower AMAT baseline.

Strategy	Targeted Term	Goal
Reduce Miss Rate	Miss Rate	Prevent misses from occurring
Reduce Miss Penalty	Miss Penalty	Make misses less costly
Reduce Hit Time	Hit Time	Make hits as fast as possible

6.9.1 Reducing Miss Rate Techniques

6.9.1.1 Increasing Cache Size

One intuitive way to reduce the number of cache misses is to simply make the cache **larger**. A bigger cache can store **more blocks**, which increases the chance that **future accesses will hit**, especially in programs with large working sets.

Effect on Miss Rate

Increasing cache size reduces **capacity misses**. If a miss happens because a previously used block was evicted to make room, then increasing the cache may **prevent such eviction**. Miss rate **typically decreases** with increasing cache size, but with **diminishing returns**. Empirical studies show that doubling the cache size doesn't halve the miss rate.

Trade-offs and Drawbacks

While it's effective, this approach isn't free. Larger caches come with:

Factor	Effect
Hit Time	✗ Usually increases (larger cache, more complex logic and longer access path).
Area	✗ Grows significantly (more memory cells, more tag storage).
Power Consumption	✗ Higher due to increased access energy.
Cost	✗ More silicon area, higher production cost.

So, a larger cache **may reduce miss rate**, but could also **increase hit time** (page 251), hurting the *overall* AMAT.

In practice:

- **L1 caches** are kept **small** and fast (to preserve CPU lock frequency).
- **L2/L3 caches** are larger and slower, used to **absorb the remaining misses**.

Increasing cache size is an **effective way to reduce capacity misses**, but it comes with trade-offs in area, latency, power, and cost. It **must be balanced** with the needs of the processor pipeline and application behavior.

6.9.1.2 Increasing Block Size

When a program accesses a memory location, it is **very likely** to soon access **neighboring addresses**: this is **Spatial Locality** (page 292).

By **increasing the cache block size** (i.e., the amount of data brought in with each cache miss), we **preload** nearby data into the cache before it's even requested.

Effect on Miss Rate

Compulsory misses[8] are reduced: since a single access brings in **more than one word**, **fewer unique block fetches** are needed. The effectiveness depends on **how well the program exhibits spatial locality**[9].

Example 9

```
for (int i = 0; i < 100; i++)
    sum += a[i];
```

If `a[i]` and `a[i + 1]` are in the same larger block, we benefit from fetching both in one miss.

Drawbacks and Trade-offs

While increasing block size helps with **spatial locality**, there are serious **downsides**:

Trade-Off	Impact
Increased Miss Penalty	Larger blocks take more time to transfer from lower levels → slower cache refill.
Fewer Blocks in Cache	For a fixed cache size, larger blocks = fewer blocks stored → **more conflict/capacity misses**.
Wasted Bandwidth	If spatial locality is weak, much of the block may be unused.
Power and Energy	More data fetched and written increases energy usage.

At some point, **increasing block size hurts more than it helps**.

[8] Compulsory Misses, or Cold Start Misses, occurs the first time a block is accessed, so it's not in the cache yet (see page 313 for more).

[9] Spatial Locality says: "*if we access this memory address, we will probably access nearby ones soon*".

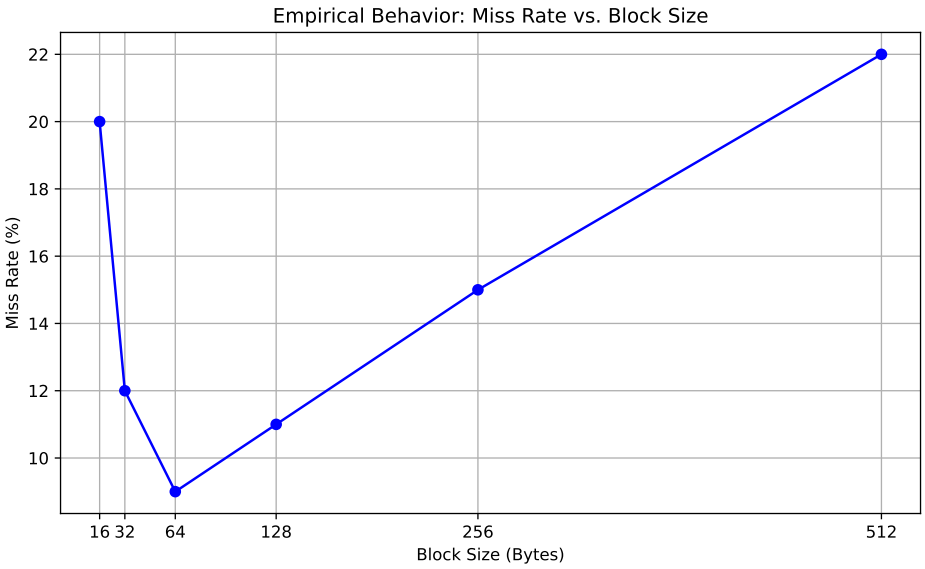
Empirical Behavior

Empirical behavior refers to how something actually behaves in practice, based on measured data, experiments, or real-world observations, not just theoretical analysis.

In our context (Block Size vs. Miss Rate), theory says: "*increasing block size should reduce compulsory misses thanks to spatial locality*". But when researchers actually test it on real systems and real programs, they observe that:

- Miss rate **initially decreases** as block size increases.
- But beyond an optimal size, it begins to **increase again** due to:
  - **Fewer total blocks** (higher conflict/capacity pressure).
  - **Higher miss penalty**.

This forms a **U-shaped curve** when plotting miss rate vs. block size.



- Common block sizes: **16, 32, 64 bytes** (chosen based on workload).
- Larger blocks may be beneficial for **instruction caches** (sequential fetch).
- Data caches are more sensitive, **balance carefully**.

Increasing block size helps ✔ **reduce compulsory misses** by exploiting **spatial locality**, but only up to a point. It can ✖ **increase conflict and capacity misses** and make ✖ **misses more expensive**.

6.9.1.3 Increasing Associativity

What is Associativity?

In the cache memory context, **Cache Associativity** describes **how many places in the cache a given block of main memory can be stored**. It answer the question: "*if we need to store this memory block in the cache, how many cache locations are possible candidates?*". There are three main types:

- **Direct-Mapped** (page 299): Associativity = 1.
- **Fully Associative** (page 300): Associativity = Number of lines in the cache.
- **N-Way Set-Associative** (page 300): Associativity = n-way, with $n > 1$.

So, associativity is essentially the "number of candidate cache lines per set" that a block can occupy. Higher associativity reduces conflict misses but increases hardware cost and lookup time.

In **direct-mapped caches** (page 299), each memory block maps to exactly **one cache location**. This simplicity results in **fast access**, but it can also cause many **conflict misses**, especially when multiple frequently used addresses map to the same index.

Solution: Use **set-associative caches** (page 300), where each set contains **multiple slots (ways)** for blocks to be stored. This **reduces conflicts** by offering **more choices** for where a block can go.

Terminology Refresher

Cache Type	Behavior
Direct-Mapped	1 block per set (1-way associative).
2-Way Set-Associative	2 blocks per set.
Fully Associative	Any block can go anywhere in the cache (all in one set).

Effect on Miss Rate

- As associativity increases, **conflict misses decreases**.
- **Capacity and compulsory misses stay the same**.
- **Fully associative caches** (page 300) **eliminate all conflict misses**, but they are expensive and slower.

Example 10

If the cache is direct-mapped, these blocks **overwrite each other**, causing many **conflict misses**:

```
a[i] → cache set 5
b[i] → cache set 5
c[i] → cache set 5
```

But with a **4-way set-associative cache**, all 3 blocks could coexist in **set 5**, avoiding those misses.

Trade-offs and Drawbacks

Trade-Off	Effect
Hit Time Increases	More ways = more tag comparisons = slower access.
Area and Complexity Increase	More comparators, multiplexers, and logic.
Power Usage Increases	More hardware is active on each access.

Design trade-off: There's a **sweet spot**; usually **2-way or 4-way** associativity gives good results without hurting hit time too much.

Rule of Thumb: 2:1 Cache Rule

A **direct-mapped cache of size** N has **about the same miss rate** as a **2-way set-associative cache of size** $\frac{N}{2}$. This tells us that we can **trade associativity for size**: higher associativity sometimes compensates for a smaller cache.

In general, increasing associativity **reduces conflict misses** by allowing more blocks to coexist in the same set, but it comes at the cost of **higher hit time, area, and power**. Designers must **balance associativity with hit time** and hardware complexity.

6.9.1.4 Victim Cache

Even with set-associative caches, **conflict misses**[10] can still occur when frequently accessed blocks map to the same set and keep evicting each other.

Instead of increasing associativity (which**slows down hit time** and **increases complexity**), we can **add a small buffer** to catch those recently evicted blocks.

Basic Idea

A **Victim Cache** is:

- **Small** (e.g., 4-16 entries).
- **Fully Associative** cache.
- Placed **between the main cache and the next lower level** in the hierarchy.

Process:

1. If a block is evicted from the main cache, it is moved into the Victim Cache.
2. On a cache miss in the main cache, **check the victim cache**:
   - If found there → **swap** it with the block in the main cache (**Victim Block Swap**).
   - If not found → fetch from lower memory level.

Effect on Miss Rate

Victim caches **reduce conflict misses** significantly, especially for **direct-mapped caches**. For example, if two memory blocks keep evicting each other from the same line in a direct-mapped cache, the victim cache allows them to **coexist** without forcing an immediate main-memory access.

Trade-offs and Costs

Factor	Impact
✔ **Miss Rate Reduction**	Very effective for small conflict sets.
✔ **No Change to Main Cache Hit Time**	The main cache remains fast.
✘ **Extra Hardware**	Additional storage, tag comparators, swap logic.
✘ **Area and Power**	Still small, but non-zero overhead.

[10] Conflict Miss happens when two or more blocks map to the same cache set, and they overwrite each other, even if the cache is not full.

Simplified Flow

1. CPU Request.
2. Main Cache Hit?
3. Yes → Done.
4. No, check Victim Cache.
5. Found?
6. Yes → Swap into Main Cache (Fast Recovery).
7. No → Fetch from Lower Level.

A **victim cache** is a low-cost, high-impact technique for **reducing conflict misses** without hurting hit time. It acts as a **safety net** for blocks that would otherwise be lost to the next memory level.

6.9.1.5 Pseudo-Associativity & Way Prediction

Increasing associativity reduces **conflict misses**, but it also **slows down the hit time** because the cache must compare **more tags in parallel**.

These techniques aim to **get some of the benefits of associativity** without paying the full hit-time penalty.

- **Way Prediction** (for Set-Associative Caches).

  The Problem with Set-Associative Caches. In an **n-way set-associative cache**:

  - Each **set** has n **ways** (cache lines).
  - When we access memory, we:
    1. Find the correct **set** using the index bits.
    2. **Compare the tag** against all n ways in parallel to see if one matches (a *hit*).

  The main issue, however, is that comparing all n ways in parallel requires n **comparators** and n **tag lookups** per cycle. This **increases both power consumption and access time**.

  The Idea of Way Prediction. Instead of checking *all* ways at once, **predict** which may is likely to have the block:

  - Hardware keeps a "**way predictor**" (like a small table) that records the *last used way* for each set. The **Way Predictor Table (WPT)**:
    - **Indexed by**: the **set index bits** from the address.
    - **Entry size**: enough bits to encode one of the n ways. For example, 4-way cache, need 2 bits per entry.
    - **Storage size**:

      $$\text{Number of sets} \times \text{Bits per entry}$$

    - **Contents**: "*Last hit way*" for that set.

    Visually:

Set Index	Predict Way (2 bits)	Meaning
0	00	Way 0
1	10	Way 2
2	01	Way 1
3	11	Way 3
4	10	Way 2
5	00	Way 0

    Each row corresponds to a set in the cache. The Predicted Way column stores a small code (2 bits here) telling which way is most likely to have the next hit for that set. The Meaning column shows the human-readable interpretation of those bits (Way 0, Way 1, etc.).

This tiny table sits alongside the cache, indexed by the same set index bits as the cache itself.

For example, 4-way cache with 1024 sets:

* Index bits: 10 bits (for 1024 sets).
* Way encoding: 2 bits.
* Way Predictor Table size: $1024 \times 2 = 2048$ bits ≈ 256 bytes.

– When accessing memory:

1. **Address split**:
   * **Tag bits**.
   * **Index bits**, select set in both the cache and WPT.
2. **WPT lookup**: read the predicted way (e.g., "*Way 2*").
3. **Access predicted way**: fetch **tag** and **data** from only that way.
4. **Tag check**:

   ✔ If tag matches → ✔ **Correct Prediction Hit**.

   ✘ If tag mismatches → ✘ **Prediction Failure**:

   (a) The **remaining** $n-1$ **ways** in the same set are read sequentially or in parallel (depends on design) to locate the block.

   (b) **Two possibilities**:
   · **Block is found in one of these ways** → **Misprediction-Hit**. For example, predicted *Way 2*, but the block is in *Way 0* (same set!).
   · **Block not found in any way** → *Miss* → fetch from lower level (L2, DRAM).

   The WPT is **updated only on hits**.

Example 11

Let's assume a **4-way set-associative cache**, with:

$$\text{WPT (Set 12)} = \text{Way 3}$$

* **Access 1 - Correct Prediction Hit**
  · Address → Set 12, Tag matches *Way 3* → Data returned in 1 cycle.
  · WPT(12) stays *Way 3*.
* **Access 2 - Misprediction-Hit**
  · Address → Set 12, WPT says *Way 3*.
  · Tag mismatch → Search remaining ways: Check *Way 0* → match found.
  · Data returned with +1 cycle penalty.
  · Update WPT(12) = *Way 0*.

* **Access 3 - Miss**
  - Address → Set 12, WPT says *Way 0*.
  - Tag mismatch → Search remaining ways: No match → Miss.
  - Load block into *Way 2* (chosen by LRU).
  - Update WPT(12) = *Way 2*.

This **predict → check predicted way → search specific alternative way(s) → update table** loop is the real hardware logic.

Why this Works. Many programs have **temporal locality**, so if a block is in the cache now, it will likely still be in the cache when accessed again. This means the **last used way** for a set is often the **next** way that will be hit. Accuracy of way prediction can be 80-95%, so most accesses are fast.

- ✔ Reduces **power** (only one way read at a time for most hits).
- ✔ Potentially reduces **access time** (one comparator instead of n).
- ✔ Keeps the **flexibility** of set-associative caches.
- ✘ **Penalty** for misprediction: 1-2 extra cycles.
- ✘ Needs **way predictor table** (small overhead).
- ✘ Works best when there's strong locality, worse with random access.

Example 12: The Hotel Room Analogy

Imagine a big hotel with **many floors** (***sets***) and **several rooms per floor** (***ways***).

- Each **guest** is like a **data block**.
- We are the **CPU**, looking for a specific guest.

Without way prediction

Every time we visit a floor, we knock on the door of **every single room** on that floor in parallel to see if our guest is inside.

- Fast in a weird "everyone answers at once" way,
- But it's noisy, energy-draining, and requires a lot of "staff" to knock on all doors simultaneously.

With way prediction

The receptionist keeps a **small notebook** (the ***way predictor***) that remembers **the last room each guest stayed in** on each floor.

  - When we arrive at a floor, the receptionist says: "Last time this guest was on this floor, they were in **Room 3**".
  - We knock **only on Room 3** first.
    * If they're still there → we're done in 1 knock (fast, quiet).
    * If they've moved → we knock on the other doors (a little slower).

Why it works

Guests usually stay in the **same room** until checkout (***temporal locality***). Even if they move, most of the time we guess right, so we save knocking on every door.

Way prediction is **not** about magically knowing where *new guests* will go, it's about remembering where **current guests** are likely to be, and checking there first.

- **Pseudo-Associativity** (or **Column-Associativity**, for Direct-Mapped caches) is a clever trick to make a direct-mapped cache behave a bit like a 2-way set-associative cache without doubling the parallel hardware.

  The Problem it solves. A **direct-mapped cache** has **exactly one possible location** for each memory block. If two blocks map to the same line, they constantly evict each other (**conflict misses**). A **2-way set-associative cache** would fix that, but it requires:

  - Two tag comparisons in parallel.
  - Two data arrays to read in parallel.
  - More area, power, and complexity.

  The Idea of Pseudo-Associativity. Instead of having two ways checked in parallel, a pseudo-associative cache checks them **sequentially**:

  - For each index, store **two possible locations** (primary and secondary).
  - On an access:
    1. **Check the primary location** (like a normal direct-mapped cache). If tag matches → **hit** in 1 cycle.
    2. If not a match → check the **secondary location** (the "pseudo-way").
       * If tag matches → **hit** but with an extra cycle of latency.
       * If still no match → miss → fetch from lower memory.

  How the secondary location is found. Often, the secondary location is **another index computed from the address** (e.g., by flipping the MSB of the index or XORing certain bits). This ensures that if two

blocks map to the same primary index, they *probably* don't also share the same secondary index.

Why it works. Most accesses still hit in the primary location, so it is fast. If there's a conflict, there's a good cache the block is in the secondary location, then conflict miss avoided, but with slightly higher hit time. Greatly reduces **conflict misses** with little extra hardware compared to a full set-associative design.

Key Difference

- **Way Prediction**: Works for set-associative caches → reduces hit time.
- **Pseudo-Associativity**: Works for direct-mapped → reduces conflict misses.

Both techniques are **hybrid approaches**. They try to **combine the speed of direct-mapped** with the **miss rate benefits of associativity**. Best suited for designs where **hit time is critical** and full associativity is too costly.

6.9.1.6 Hardware Prefetching (Instructions & Data)

Even with optimized cache size, associativity, and victim caches, **compulsory misses** will still occur: the first time a block is accessed, it has to be fetched.

Idea: If we can *predict* what data or instructions will be needed soon, we can bring them into the cache early. When the CPU asks for them, they're already there, so **no miss penalty**.

- **Instruction Prefetching**. Instruction streams often have **strong spatial locality** (instructions are usually executed sequentially). **Hardware prefetch logic** detects sequential access patterns and preloads **the next block of instructions** while the current one is being executed.

  ✔ Reduces **compulsory misses** for sequential code.
  ✔ Keeps the instruction pipeline fed.

Example 13: Alpha AXP 21064

The **Alpha AXP 21064** was a **high-performance microprocessor** from Digital Equipment Corporation (DEC) released in the early 1990s, one of the first in the Alpha architecture family.

It was one of the earliest commercially available CPUs to implement **stream buffer-based hardware prefetching** for caches.

The 21064's data cache was direct-mapped, which is fast but prone to **conflict misses**. To help hide memory latency, DEC implemented **hardware prefetching using stream buffers**:

- When the CPU missed in the cache for an address, the hardware **predicted the next sequential address** that would be needed.
- It loaded them into **stream buffers** in advance, so when the CPU accessed them, they were ready.

This prefetch mechanism was **triggered automatically** by the detection of sequential access patterns (common in loops and array processing).

Before this, hardware prefetching wasn't common in general-purpose CPUs, prefetching was mostly a software/compiler job. The Alpha 21064's implementation showed that **simple hardware mechanisms could significantly reduce memory stall cycles**. Later CPUs (from MIPS, PowerPC, Intel, AMD, etc.) adopted similar or more advanced prefetching techniques.

In summary, on a miss:

1. Fetch the **requested block** into the instruction cache.
2. Also fetch the **next sequential block** into a separate **instruction stream buffer**.

If the next fetch hits in the stream buffer, the block is moved to the cache, and the next block after that is prefetched.

- **Data Prefetching**. Data Prefetching means **bringing data into the cache *before* the CPU actually requests it**, so that when it needs it, it's already there. This **hides memory latency**. Prefetching can be done by:
  - **software** (compiler or programmer inserts prefetch instructions).
  - **Hardware** (CPU detects patterns and fetches ahead automatically).

  Data prefetching is more challenging because data access patterns can be irregular. However, in many workloads (e.g., scientific computing, multimedia), data is accessed in predictable sequences.

  1. CPU detects a **patter of accesses**, usually sequential or strided addresses (e.g., `A[0]`, `A[1]`, `A[2]`, ...).
  2. It predicts the **next address** we'll want.
  3. It sends a request to memory/cache hierarchy **in advance**.
  4. When we actually access that address later, it's already in the cache, then **hit** instead of miss.

  More specifically:

  1. **Cache Miss**. CPU requests address `X`. It's not in the cache, then **miss**. Cache controller fetches block `X` from memory **into the cache**.
  2. **Prefetch the next block(s)**. The hardware prefetcher guesses the **next block(s)** we'll need (usually sequential, if we missed on block `X`, next will be `X+1`, `X+2`, etc). These predicted blocks are **not** put into the main cache immediately. Instead, they are stored in the **D-stream buffer**.

     What is a "*hardware data prefetcher*" with D-stream buffers?

     It's a **small dedicated hardware unit** sitting next to the cache. A **D-Stream Buffer** is a temporary holding area for **prefetched data** that isn't in the cache yet, but might be needed soon.

     It's a sort of "waiting room" for data that the CPU hasn't requested yet, but that the prefetcher predicts it will.
  3. **CPU requests prefetched data**. If later the CPU requests block `X+1`:
     (a) Prefetcher sees it's already in the D-stream buffer.
     (b) Moves it **quickly** from the D-stream buffer into the cache → **fast hit**.

Why not put prefetched data directly into the cache?

Prefetched data might never be used. If we store it directly in cache, we risk evicting useful data (**cache pollution**). Stream buffers keep it separate until it's actually needed, so only *useful* prefetched data enters the cache.

Example 14: Data Prefetching Analogy

Think of the **cache** as the table we're eating at, and the **D-stream buffer** as a **side tray** where the waiter puts the next dish they think we'll want.

- If they guessed right, they slide it onto our plate instantly.
- If they guessed wrong, they take it away without messing with our current plate.

Hardware prefetching **guesses future accesses** and proactively brings data or instructions closer to the CPU, reducing **compulsory** and some **capacity misses**, but must be carefully tuned to avoid **wasting bandwidth**.

6.9.1.7 Software Prefetching (Compiler-Controlled)

Hardware prefetchers work automatically but **cannot always detect complex or irregular access patterns**. A **compiler** (or programmer) often has more information about upcoming memory accesses and can explicitly request that certain data be loaded in advance.

This **targets compulsory and some capacity misses** by overlapping memory fetches with useful computation.

Idea

The **compiler analyzes the program** (often using profiling data) to identify memory accesses that could cause misses. It inserts **special prefetch instructions** into the code, placed *before* the data is actually needed. These instructions **fetch the block into cache** or **into a register**, giving enough time for the load to complete before use.

Types of Software Prefetching

1. **Register Prefetching**. Prefetch into **CPU registers** for immediate use. For example, HP PA-RISC "load" instruction. Useful for small, predictable data needs.
2. **Cache Prefetching**. Prefetch into **cache** not into registers. For example, MIPS IV, PowerPC, SPARC v9 have dedicated cache prefetch instructions. Data stays in cache until needed by normal load.

Example 15

Here, the compiler issues a prefetch for `A[i+8]` while the CPU is still working on `A[i]`.

- **Without Prefetching**

```
for (i = 0; i < N; i++) {
    sum += A[i];  // May cause many misses
}
```

- **With Compiler-Inserted Prefetch**

```
for (i = 0; i < N; i++) {
    __prefetch(&A[i+8]);  // Load ahead of time
    sum += A[i];
}
```

Performance Considerations

Prefetch instructions **occupy instruction slots**, so the CPU must be able to issue them without stalling important work. In **superscalar processors**, the difficulty is reduced because multiple instructions can be issued in parallel. The **distance between prefetch and actual use** is crucial:

- Too close → miss penalty still occurs.
- Too far → data might be evicted before use.

Trade-offs

Pros	Cons
✔ Reduces compulsory/capacity misses for predictable accesses.	✖ Wastes bandwidth if data not used.
✔ Works for irregular patterns hardware may miss.	✖ Adds extra instructions (execution overhead).
✔ Can be tuned by the compiler based on profiling.	✖ Needs accurate prediction of future accesses.

Software prefetching leverages **compiler insight** to prepare data before it's needed. It is especially useful for **predictable loops** and **regular patterns**, but must be carefully tuned to avoid wasted bandwidth and unnecessary instruction overhead.

6.9.1.8 Compiler Optimizations (for Cache Performance)

Even with prefetching, a program's **structure** might still cause many cache misses. The compiler can transform the code and data layout to **increase spatial and temporal locality**, thus reducing misses **before the hardware even runs the code**.

Basic Idea

1. **Profile the program** to understand typical memory access patterns.
2. **Reorganize code** and **restructure data** so that:
   - Related data is stored **together** (improves spatial locality).
   - Frequently reused data stays **in cache longer** (improves temporal locality).
3. Generate machine code that exploits these new layouts.

Techniques

- **Merging Arrays**. Combine multiple arrays into one array of structures. Reduces conflict misses and improves spatial locality.
  - **Before**

```
int val[SIZE];
int key[SIZE];
```

  - **After**

```
struct {
    int val;
    int key;
} merged_array[SIZE];
```

  Both `val` and `key` for the same index end up in the **same cache block**.

- **Loop Interchange**. Swap the order of nested loops to match memory layout (row-major or column-major). Improves spatial locality by accessing consecutive memory locations.
  - **Before**

```
for (j = 0; j < 100; j++)
    for (i = 0; i < 5000; i++)
        x[i][j] = 2 * x[i][j];
```

  - **After**

```
for (i = 0; i < 5000; i++)
    for (j = 0; j < 100; j++)
        x[i][j] = 2 * x[i][j];
```

  Now accesses move **sequentially through memory**.

- **Loop Fusion**. Combine two loops that iterate over the same data into one loop. Reduces repeated loading of the same data.
  - **Before**

```
for (i = 0; i < N; i++)
    A[i] = B[i] + 1;
for (i = 0; i < N; i++)
    C[i] = A[i] * 2;
```

  - **After**

```
for (i = 0; i < N; i++) {
    A[i] = B[i] + 1;
    C[i] = A[i] * 2;
}
```

  Improves **temporal locality**, `A[i]` stays in cache between operations.

- **Loop Blocking (Tiling)**. Process data in **small chunks (blocks)** that fit in the cache. Common in matrix operations.
  - **Before**

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        process(A[i][j]);
```

  - **After**

```
for (ii = 0; ii < N; ii += B)
    for (jj = 0; jj < N; jj += B)
        for (i = ii; i < ii + B; i++)
            for (j = jj; j < jj + B; j++)
                process(A[i][j]);
```

  Improves **temporal locality**, elements in the block are reused before being evicted.

⚠ Trade-offs

Pros	Cons
✔ Works for predictable patterns.	✘ Requires compiler complexity.
✔ Reduces both compulsory & capacity misses.	✘ May increase code complexity & size.
✔ No hardware cost.	✘ Limited effect on unpredictable access patterns.

Compiler optimizations restructure both **data** and **loops** to **align program access patterns with cache organization**. When combined with hardware techniques, they significantly reduce miss rate **without changing the hardware**.

6.9.2 Reducing Miss Penalty Techniques

6.9.2.1 Read Priority over Write on Miss

A cache miss can happen in two main ways:

- **Read miss**: The CPU needs data from memory (load instruction).
- **Write miss**: The CPU wants to store data to memory (store instruction) and the block isn't in the cache.

When a miss happens, the cache must send a **request to the next memory level** (L2, DRAM). But memory is slow and usually **only one outstanding miss can be serviced at a time per port**.

Idea: when both reads and writes compete for memory access, **give priority to reads** because:

- Reads **stall the CPU** immediately (we can't execute without the data).
- Writes often go to a **write buffer** and can be delayed without stopping execution.

So are we assuming that a read and a write miss happen at the same time? Yes, we're talking about a situation where two independent memory operations are outstanding in the cache/memory system at the same time. That can happen in a modern CPU because pipelines are out-of-order and non-blocking. For example, imagine:

1. Cycle 10 → CPU issues load from address A → **read miss**.
2. Cycle 12 → CPU issues store to address B → **write miss**.
3. Both go into the miss queue.
4. Memory controller sees both pending, **picks read first** to reduce CPU stall.
5. Write sits in the write buffer until read completes, then gets sent to memory.

The "*simultaneous*" read miss and write miss doesn't mean they happen in the *exact same cycle*, it means they are **both pending before memory has finished handling the first one**, so the controller has to choose an order.

Basic Concept

- Case 1: **Write-Through with Write Buffer**. Writes go immediately to memory but are buffered to avoid stalling.
    1. CPU issues a **write** → cache updated (write-through, value is written to cache block and main memory) → **write buffer** stores the memory update so the CPU doesn't wait.
    2. Later, CPU issues a **read miss**.

3. Before fetching from memory, the cache controller **checks the write buffer**:
   - If the read address matches an address in the write buffer → **read directly from the buffer** so we don't get stale data.
   - If no match → bypass the write requests in the buffer and **immediately send the read miss to memory** (read priority).
4. Writes in the buffer are sent to memory later, in background, when the bus is free.

Why it works: ensures read misses aren't delayed by pending writes and still guarantees correctness if the needed data is in the write buffer.

- Case 2: **Write-Back with Dirty Block Replacement**. Writes are deferred until a dirty block is evicted.

1. Cache uses **write-back** policy → writes only update cache and mark block as dirty.
2. On a **read miss**, we need to bring a new block into the cache.
3. If the victim block is **clean** (Dirty Bit is 0) → just replace it.
4. If the victim block is **dirty** (Dirty Bit is 1):
   - ✗ **Without read priority**: we'd write the dirty block back to memory *before* fetching the new block → long delay before CPU gets the new data.
   - ✓ **With read priority**:
     (a) Copy the dirty block into the **write buffer** (so it can be written to memory later).
     (b) **Immediately start fetching the new block** for the read miss.
     (c) Once memory is free, the dirty block in the write buffer is written back in the background.

Why it works: the CPU gets the missing data sooner and the dirty block write-back doesn't block the read fetch.

These "*read priority*" methods aren't about *reordering a generic queue arbitrarily*, they're about **changing the handling sequence inside the cache controller's normal process** so that when a **read miss** and a **write-related action** would normally compete for memory access, the **read gets serviced first** within that specific policy's procedure.

In other words:

- In **write-through with write buffer**: instead of draining the write buffer first, we pause it (unless the read needs its data) and go straight to the read miss fetch.
- In **write-back with dirty block replacement**: instead of writing the dirty victim to memory *before* fetching the new block, we park it in the write buffer and fetch the read's block immediately.

Benefits

- ✔ Reduces **CPU stall time** by overlapping write operations with read miss handling.
- ✔ Keeps the processor busy instead of waiting for slow write completions.

Challenges

- ✖ **RAW hazard through memory**. If the read request matches an address in the write buffer, we must read from there instead of memory.
- ✖ **Write buffer overflow**. If too many writes accumulate, read requests can still be delayed.
- ✖ **Control complexity**. Cache controller must track dependencies between writes and reads.

6.9.2.2 Sub-block Placement

Normally, when a cache miss occurs, the cache **fetches the entire block** (e.g., 64 bytes) from the next memory level. This **increases miss penalty** because:

- The transfer time is proportional to block size.
- We might not even need all the words in that block.

Idea: Break each cache block into **smaller pieces (sub-blocks)** with **individual valid bits**. This way, on a miss, we can **load only the needed sub-block** instead of the entire block.

How it works

- Each block is divided into **sub-blocks** (e.g., 4 words per block, becomes 4 sub-blocks).
- Each sub-block has its own **Valid Bit**.
- On a miss:
  - Only the requested sub-block is fetched.
  - Other sub-blocks in the same block are marked invalid until fetched.

Benefits

- ✔ **Lower Miss Penalty**: Only part of the block is transferred, then less latency.
- ✔ **Better Bandwidth Usage**: Avoids fetching unused data.
- ✔ Can be useful for **irregular access patterns** where spatial locality is weak.

It is useful when **memory bandwidth is limited** or when **access patterns are sparse** (e.g., random access over large arrays). It is also useful in systems where the **miss penalty is high** and spatial locality is low.

Drawbacks

- ✘ **Lower Spatial Locality Exploitation**: Since we don't fetch the whole block, nearby addresses may miss later.
- ✘ **Extra Tag/Valid Bit Storage**: Need valid bits for each sub-block.
- ✘ **More Complex Control Logic**: Cache controller must track which sub-blocks are valid.

Sub-block placement trades off **spatial locality** for **lower miss penalty** and better bandwidth usage. It's especially useful when **large block sizes** would otherwise cause excessive transfer delays.

6.9.2.3 Early Restart & Critical Word First

Early Restart & Critical Word First are **two related techniques** that aim to let the CPU resume execution faster **when a cache miss occurs**.

In a normal cache miss:

1. The cache requests the **entire block** from the next memory level.
2. The CPU waits until **all words** in the block have arrived.
3. Only then does the CPU resume execution.

Problem: This wastes cycles if the needed word arrivers before the rest of the block.

Solution: Send the **critical word** (the one the CPU actually requested) **first**, so execution can resume earlier.

Techniques

- **Early Restart**
  - Fetch the block **in normal order** from memory.
  - As soon as the **requested word** arrives, **send it to the CPU** and let it resume.
  - Continue transferring the rest of the block into the cache in the background.

  For example:

  - Block size: 4 words (W0, W1, W2, W3).
  - CPU requests W2.
  - Memory returns W0 → W1 → **W2** → W3.
  - On W2's arrival → **CPU resumes**, W3 still coming.

  Benefit

  - ✔ Works well when **memory returns words sequentially** in block order.
  - ✔ Reduces **CPU stall time** for some misses.
- **Critical Word First** (aka **Wrapped Fetch**).
  - When a miss occurs, **start fetching at the requested word**, then wrap around to the start of the block.
  - The **requested word is fetched first**, not just delivered first.

  For example:

  - Block size: 4 words (W0, W1, W2, W3).

- CPU requests W2.
- Memory returns **W2** → W3 → W0 → W1.
- W2 is delivered to CPU immediately → stall time minimized.

Benefit

- ✔ More effective than Early Restart when **requested word is late in block order**.
- ✔ Particularly useful when **miss penalty is dominated by first-word latency**.

Trade-offs

- ✖ **Controller Complexity**: Critical Word First requires non-sequential wrap-around fetch logic.
- ✖ **Lower Spatial Locality Use**: If other words are needed soon, reordering fetch might not help.
- ✖ **Memory Bus Efficiency**: Some memory systems prefer sequential transfers for efficiency.

Both Early Restart and Critical Word First **reduce miss penalty** by **delivering the requested word sooner**, but Critical Word First goes further by **fetching it first** from memory.

6.9.2.4 Non-Blocking Caches

The problem with classic (blocking) caches

In a **blocking cache**, when we get a **miss**, the cache controller stalls the CPU until the miss is resolved and the data is in the cache. Even if the CPU could access **other data already in the cache**, it can't. It must wait for the miss to finish.

```
sum += A[i];    // miss → wait...
sum += B[0];    // could be a hit, but must wait anyway
```

The idea of Non-Blocking Caches

A **Non-Blocking Cache** lets the CPU **continue to service hits even when a miss is being processed**.

- This means the cache can have **multiple memory transactions in-flight**.
- The hardware must track each outstanding miss (via **Miss Status Holding Registers (MSHRs)**).
- The CPU can keep executing instructions that do not depend on the missing data.

Hit Under Miss

Hit Under Miss is the **simplest level of non-blocking cache capability**. The cache can handle hits to other addresses while one miss is outstanding.

1. Miss on address `X` → request sent to memory.
2. While waiting, CPU requests address `Y`.
3. If `Y` is in the cache → serve it immediately (**hit under miss**).
4. If `Y` is also miss → depends on capability (next topic, miss under miss).

Benefit: hides some of the miss by overlapping **independent hits** with miss handling.

Going beyond: "Miss Under Miss"

A more advanced feature is **Miss Under Miss** (aka **Multiple Outstanding Misses**):

- Allows **starting a second miss while the first one is still being serviced**.
- Requires **multiple Miss Status Holding Registers (MSHRs)** to track all pending misses, and **non-blocking memory controllers** and **out-of-order handling** of cache fills.

✔ **Benefit:** maximizes **memory-level parallelism (MLP)** and is particularly useful for high-latency main memory or multi-core systems.

⚠ Trade-offs

- ✖ **Hardware Complexity**: Tracking multiple misses is non-trivial..
- ✖ **Extra Area & Power**: Miss Status Holding Registers (MSHRs) consume silicon and energy.
- ✖ **Consistency Handling**: Must ensure memory ordering rules are respected.

Non-blocking caches **reduce effective miss penalty** by overlapping miss servicing with useful work:

- **Blocking cache**: One miss stops *all* accesses.
- **Hit Under Miss**: One miss stops *new misses*, but hits to other addresses still work.
- **Miss Under Miss**: Multiple misses can be serviced in parallel.

7 Exams

7.1 2025

7.1.1 Midterm - May 5

Exercise 1.A - Software Pipelining

Consider the following software pipelined loop `SP_LOOP` *and the corresponding start-up and finish-up code:*

```
START_UP:   LD F0, 0 (R1)
            LD F2, 0 (R2)
            FADD F4, F0, F0
            FADD F6, F2, F2
            LD F0, 8 (R1)
            LD F2, 8 (R2)

SP_LOOP:    SD F4, 0 (R1)
            SD F6, 0 (R2)
            FADD F4, F0, F0
            FADD F6, F2, F2
            LD F0, 16 (R1)
            LD F2, 16 (R2)
            ADDUI R1, R1, 8
            ADDUI R2, R2, 8
            ADD R3, R2, R1
            BNE R3, R4, SP_LOOP

FINISH-UP:  SD F4, 8 (R1)
            SD F6, 8 (R2)
            FADD F4, F0, F0
            FADD F6, F2, F2
            SD F4, 16 (R1)
            SD F6, 16 (R2)
```

Reconstruct the original (non-pipelined) version of the code.

Answer: **Software Pipelining** is a loop optimization technique. Instead of processing iterations strictly sequentially, it overlaps instructions from multiple iterations. So in the pipelined version:

- The loop body does *parts* of multiple iterations at the same time.
- Because of this, we need extra **start-up** code (to prime the pipeline) and **finish-up** code (to flush the remaining operations).

The original loop must do **one iteration at a time**. In the original **non-pipelined loop**, we:

- Load the data for the current iteration.
- Do the computation.
- Store the results.
- Update pointers.
- Check whether you are done.
- Repeat.

In software pipelining we always have:

1. Start-up code
2. Steady-state loop
3. Finish-up code

Because the pipeline overlaps *parts of different iterations*. So we need:

- Extra code at the **start** to *prime* the pipeline with partial work.
- The **main loop** which does the steady overlapping work for all *middle* iterations.
- Extra code at the **end** to *drain* the pipeline, to finish the partial work left from the last overlapping iterations.

In other words, we need to reverse engineer the original code for this exercise.

The software pipelined loop usually overlaps:

- *Load* for **next** iteration.
- *Compute* for **current** iteration.
- *Store* for **previous** iteration.

The goal is trace the dependency chain:

- Which `LD` feeds which `FADD`.
- Which `FADD` feeds which `SD`.

So the key is to identify the loop-carried data flow and align addresses to match.

General 5-step method:

1. **Mark what the pipelined loop does**. In our case:

```
SP_LOOP:
    SD F4, 0(R1)          # Store result of previous iteration?
    SD F6, 0(R2)

    FADD F4, F0, F0       # Compute for current iteration?
    FADD F6, F2, F2

    LD F0, 16(R1)         # Load for next iteration
    LD F2, 16(R2)

    ADDUI R1, R1, 8       # Increment base pointers
    ADDUI R2, R2, 8
```

2. **Figure out the purpose**
    - `LD`: prefetch next input.
    - `FADD`: do the math for the loaded input.
    - `SD`: store the result **from previous iteration**.

3. **Sketch the timeline**. Think conceptually:

Pipeline Stage	Operation
1. Load	Loads new data for `i + 1`
2. Compute	Uses old load for `i`
3. Store	Saves result for `i - 1`

4. **Match start-up and finish-up**
   - **Start-up**: primes the pipeline, so gets first `F0`, `F2` ready.
   - **Loop**: repeats the middle chunk.
   - **Finish-up**: drains the leftover results.
5. **Write the original loop**. For each iteration:
   - Load `F0, F2` ← data for i
   - Compute `F4 = F0 + F0`, `F6 = F2 + F2` ← process for i
   - Store `F4, F6` ← save result for i
   - Increment pointers
   - Check loop condition

So, here's the final solution:

```
LOOP:   LD F0, 0 (R1)
        LD F2, 0 (R2)
        FADD F4, F0, F0
        FADD F6, F2, F2
        SD F4, 0 (R1)
        SD F6, 0 (R2)
        ADDUI R1, R1, 8
        ADDUI R2, R2, 8
        ADD R3, R2, R1
        BNE R3, R4, LOOP
```

Exercise 1.B - Software Pipelining

*Give the same software pipelined loop of **Exercise 1.A**:*

```
SP_LOOP:    SD F4, 0 (R1)
            SD F6, 0 (R2)
            FADD F4, F0, F0
            FADD F6, F2, F2
            LD F0, 16 (R1)
            LD F2, 16 (R2)
            ADDUI R1, R1, 8
            ADDUI R2, R2, 8
            ADD R3, R2, R1
            BNE R3, R4, SP_LOOP
```

Consider a **3-issue VLIW** machine with **fully pipelined functional units**:

- **1 Memory Units with 3 cycles latency**
- **1 FP ALUs with 3 cycles latency**
- **1 Integer ALU with 1 cycle latency to next Int/FP & 2 cycle latency to next Branch**

The branch is completed with 1 cycle delay slot (branch solved in ID stage). **No branch prediction**. In the Register File, it is possible to read and write at the same address at the same clock cycle.

1. *Considering one iteration of the `SP_LOOP`, complete the following table by using the **list-based scheduling** (do NOT introduce any loop unrolling and modifications to loop indexes) on the 3-issue VLIW machine including the BRANCH DELAY SLOT. Please do not write in NOPs.*

   Answer: In a **VLIW architecture** (Very Long Instruction Word), a single instruction word encodes **multiple operations** that execute *in parallel*. The compiler does the heavy lifting: it schedules independent instructions side by side in the same very long instruction word. The hardware just *fires* all operations in parallel, it does **not** dynamically check for dependencies at runtime (unlike out-of-order superscalar).

   What does *3-issue* mean? It is the issue width and is the number of operations the processor *fetches, decodes, and executes* in parallel **per cycle**. So, with 3-issue, the processor can execute up to **3 instructions per clock cycles**. They must be **independent** enough for the compiler to pack them together.

   What does *fully pipelined functional units* mean? Each type of operation is done in its own **functional unit** (e.g., integer ALU, floating-point adder, load/store unit, etc.). **Fully pipelined** means multiple operations can be in *different pipeline stages* of the same unit simultaneously. So, every clock cycle a new operation can start, even if a previous one isn't finished yet. For example, if an `FADD` unit is fully pipelined and the `FADD` takes 4 cycles to complete, it can accept **one new `FADD` per cycle**, the result pops our 4 cycles later.

What is *list-based scheduling*? List-based scheduling is a classic **compiler scheduling algorithm** used to assign operations to *time slots* (cycles) and *resources* (e.g., functional units) **while respecting dependencies**.

- It tries to pack **as many independent instructions as possible** into each cycle.
- It tries to **fill all slots** of a wide-issue machine (VLIW, superscalar).
- It tries to respect resource limits (e.g., 1 integer ALU, 1 FP adder).

It is called list-based because the compiler first builds a **dependency graph** (where the nodes are the operations and the edges are the dependencies) and then creates a **list** of **ready operations**. These are instructions **whose inputs are ready** and **whose predecessors have been completed**. Each cycle:

- It picks operations from the list.
- It assigns them to available slots/resources.
- It marks successors ready when dependencies clear.

What does *In the Register File, it is possible to read and write at the same address in the same clock cycle* mean? The **register file** supports *simultaneous read and write* to the **same register**. So, in one cycle:

- We might **read register** `F0` to feed an `FADD`.
- AND **write back a result** to `F0` in the same cycle.

The hardware guarantees the read gets the **old value**, the write updates it for **later cycles**.

If the register file did not allow this, we'd have to schedule dependent instructions in separate cycles to avoid conflict (e.g., we couldn't read `F0` to feed `FADD` while overwriting `F0` with a new `LD`). Otherwise, if it is allowed, in same cycle we can:

- `LD F0, ...` (writes `F0`)
- `FADD F4, F0, F0` (reads old `F0`)

Here are some notes about the solution:

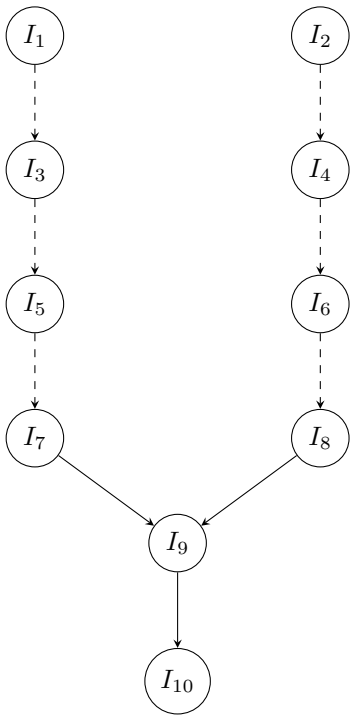
- We are using software pipelining, so the `SD` and `FADD` operators can start together because the `SD` operator stores the results of the previous iteration (or the initial loop, if it is the first iteration).
- The `ADDUI` operator starts when the first `SD` store finishes, thanks to the register file. In the same cycle, the load operation begins, as it loads the operands for the next iteration.
- Cycle 6 stalls because the `BNE` operator must wait for the result of the previous ADD and cannot start in cycle 6 because it reads old values.

What should we do about our loop?

(a) Build the **DAG** for one iteration. A Data Dependency Graph (DAG) for scheduling has nodes, which are operations or instructions, and edges, which are directed and show true dependencies (RAW). For example, an edge from `A` to `B` means that `B` must wait for `A` to finish.

(b) Draw a dashed line if there are WAR or WAW hazards. This means we can perform these two instructions in the same cycle if a FU free is available.

(c) In the table, write the instructions from start to finish.



	Memory Unit	Floating Point Unit	Integer Unit
C1	`SD F4, 0 (R1)`	`FADD F4, F0, F0`	
C2	`SD F6, 0 (R2)`	`FADD F6, F2, F2`	
C3	`LD F0, 16 (R1)`		`ADDUI R1, R1, 8`
C4	`LD F2, 16 (R2)`		`ADDUI R2, R2, 8`
C5			`ADD R3, R2, R1`
C6			
C7			`BNE R3, R4, SP_LOOP`
C8			(branch delay)

2. *How long is the critical path for a single iteration?*

   Answer: in **instruction scheduling**, the **critical path** is the longest chain of dependent operations that must be executed *in sequence* because of true data dependencies (RAW). It determines the **minimum possible latency** to complete *one full logical iteration*. The critical path length depends on the data dependencies and the assumed functional unit latencies, **not** the scheduling algorithm.

   In our case, there are 8 cycles for each iteration, so the **critical path is 8**.

3. *What performance did you achieve in CPI?*

   Answer: The Clock Per Instruction (CPI) can be calculated using the following formula:

$$\text{CPI} = \frac{\#\text{ Clock Cycles}}{\text{IC}} = \frac{\#\text{ Clock Cycles}}{\#\text{ Instructions}} = \frac{8}{10} = 0.8$$

4. *What performance did you achieve in FP ops per cycles?*

   Answer: FP ops per cycle measures **how well we are utilizing the FP functional units**. Here, there are 2 floating-point additions (`FADD F4`, `FADD F6`):

$$\text{FP ops / cycle} = \frac{\#\text{ Floating-Point ops}}{\#\text{ Instructions}} = \frac{2}{8} = 0.25$$

5. *How much is the code efficiency?*

   Answer: When asked for **code efficiency** in this VLIW/ILP context, they usually mean:

$$\text{Efficiency} = \frac{\text{Achieved IPC}}{\text{Maximum Theoretical IPC}}$$

   So we measure **how much of the machine's peak parallelism we actually use**. The achieved Instruction Per Cycle (IPC) is:

$$\text{Achieved IPC} = \frac{\#\text{ Instructions}}{\#\text{ Clock Cycles}} = \frac{10}{8} = 1.25$$

   Here, VLIW is 3-issue, so it can execute up to **3 ops per cycle** if there's no data or resource limit (Peak IPC = 3). So, code efficiency is:

$$\text{Efficiency} = \frac{1.25}{3} = 0.4167 \approx 41.67\% \approx 42\%$$

   We're achieving about 41.67% ($\approx 42\%$) of the maximum possible execution parallelism.

Exercise 2.A - Dependency Analysis

*Consider the same software pipelined loop of **Exercise 1.A** containing multiple types of intra-loop dependences. Complete the following table by inserting all types of true-data-dependences, anti- dependences and output dependences for each instruction:*

Answer:

I#	**Type of instruction**	**Analysis of dependences:** 1. True data dependence with `I#` for `$Fx` 2. Anti-dependence with `I#` for `$Fy` 3. Output-dependence with `I#` for `$Fz`
I1	`SD F4, 0 (R1)`	None
I2	`SD F6, 0 (R2)`	None
I3	`FADD F4, F0, F0`	Anti-dependence with `I1` for `F4`
I4	`FADD F6, F2, F2`	Anti-dependence with `I2` for `F6`
I5	`LD F0, 16 (R1)`	Anti-dependence with `I3` for `F0`
I6	`LD F2, 16 (R2)`	Anti-dependence with `I4` for `F2`
I7	`ADDUI R1, R1, 8`	Anti-dependence with `I1` for `R1` Anti-dependence with `I5` for `R1`
I8	`ADDUI R2, R2, 8`	Anti-dependence with `I2` for `R2` Anti-dependence with `I6` for `R2`
I9	`ADD R3, R2, R1`	True data dependence with `I7` for `R1` True data dependence with `I8` for `R2`
I10	`BNE R3, R4, SP_LOOP`	True data dependence with `I9` for `R3`

Exercise 2.B - Tomasulo

Consider the same assembly code to be executed on a CPU with dynamic scheduling based on ***Tomasulo algorithm*** *with all cache HITS, a single Common Data Bus and:*

- *2 RESERVATION STATIONS (**RS1**, **RS2**) with 2 LOAD/STORE units (**LDU1**, **LDU2**) with latency **4***
- *2 RESERVATION STATIONS (**RS3**, **RS4**) with 1 FP unit (**FPU1**) with latency **4***
- *2 RESERVATION STATIONS (**RS5**, **RS6**) with 2 INT_ALU/BR units (**ALU1**, **ALU2**) with latency **2***

Please complete the following table:

Answer: Before solving the Tomasulo, we need to remember how it is composed (see page 143).

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0						
I4: FADD F6, F2, F2						
I5: LD F0, 16 (R1)						
I6: LD F2, 16 (R2)						
I7: ADDUI R1, R1, 8						
I8: ADDUI R2, R2, 8						
I9: ADD R3, R2, R1						
I10: BNE R3, R4, LOOP						

1. **Cycle 1**:

	Vj	Qj	Vk	Qk
RS1	F4		-	
RS2				
LDU1				
LDU2				

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				

	Vj	Qj	Vk	Qk
RS5				
RS6				
ALU1				
ALU2				

2. **Cycle 2**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0						
I4: FADD F6, F2, F2						
I5: LD F0, 16 (R1)						
I6: LD F2, 16 (R2)						
I7: ADDUI R1, R1, 8						
I8: ADDUI R2, R2, 8						
I9: ADD R3, R2, R1						
I10: BNE R3, R4, LOOP						

	Vj	Qj	Vk	Qk
RS1	F4		-	
RS2	F6		-	
LDU1	F4		-	
LDU2				

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				

	Vj	Qj	Vk	Qk
RS5				
RS6				
ALU1				
ALU2				

Unit	Remaining cycles
LDU1	4
LDU2	
FPU1	
ALU1	
ALU2	

3. **Cycle 3**: Here, we have a anti-dependence hazard (WAR) because the third instruction writes the sum to register `F4` and the first instruction reads `F4`. In Tomasulo, though, we don't care about this because the issue stage stalls if and only if there are no reservation stations, the Load/Store queue is full (for memory ops), or for speculative instructions (for prediction, which is not our case). Therefore, we issue and start execution. At the end of execution, we will determine whether to write the result or wait.

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3			WAR I1, F4	RS3	
I4: FADD F6, F2, F2						
I5: LD F0, 16 (R1)						
I6: LD F2, 16 (R2)						
I7: ADDUI R1, R1, 8						
I8: ADDUI R2, R2, 8						
I9: ADD R3, R2, R1						
I10: BNE R3, R4, LOOP						

	Vj	Qj	Vk	Qk
RS1	F4		-	
RS2	F6		-	
LDU1	F4		-	
LDU2	F6		-	

	Vj	Qj	Vk	Qk
RS3	F0		F0	
RS4				
FPU1				

	Vj	Qj	Vk	Qk
RS5				
RS6				
ALU1				
ALU2				

Unit	Remaining cycles
LDU1	3
LDU2	4
FPU1	
ALU1	
ALU2	

4. **Cycle 4**: It's the same situation as before with the hazard (WAR), but now it's `I4` with `I2`.

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
`I1: SD F4, 0 (R1)`	1	2	6	None	`RS1`	`LDU1`
`I2: SD F6, 0 (R2)`	2	3	7	None	`RS2`	`LDU2`
`I3: FADD F4, F0, F0`	3	4		`WAR I1, F4`	`RS3`	`FPU1`
`I4: FADD F6, F2, F2`	4			`WAR I2, F6`	`RS4`	
`I5: LD F0, 16 (R1)`						
`I6: LD F2, 16 (R2)`						
`I7: ADDUI R1, R1, 8`						
`I8: ADDUI R2, R2, 8`						
`I9: ADD R3, R2, R1`						
`I10: BNE R3, R4, LOOP`						

	Vj	Qj	Vk	Qk
RS1	F4		-	
RS2	F6		-	
LDU1	F4		-	
LDU2	F6		-	

	Vj	Qj	Vk	Qk
RS5				
RS6				
ALU1				
ALU2				

	Vj	Qj	Vk	Qk
RS3	F0		F0	
RS4	F2		F2	
FPU1	F0		F0	

Unit	Remaining cycles
LDU1	2
LDU2	3
FPU1	4
ALU1	
ALU2	

5. **Cycle 5**: Instruction 4 cannot begin because there are no available functional units. Therefore, it remains idle at reservation station 4. Instructions 5 and 6 cannot start either because there are no load/store reservation units available. `ADDUI` operations cannot start because instructions 5 and 6 need to read the `R1` and `R2` registers (WAR hazard).

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4		WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4			WAR I2, F6	RS4	
I5: LD F0, 16 (R1)				WAR I3, F0		
I6: LD F2, 16 (R2)				WAR I4, F2		
I7: ADDUI R1, R1, 8				WAR I5, R1		
I8: ADDUI R2, R2, 8				WAR I6, R2		
I9: ADD R3, R2, R1				RAW R1, R2		
I10: BNE R3, R4, LOOP				RAW I9, R3		

	Vj	Qj	Vk	Qk
RS1	F4		-	
RS2	F6		-	
LDU1	F4		-	
LDU2	F6		-	

	Vj	Qj	Vk	Qk
RS3	F0		F0	
RS4	F2		F2	
FPU1	F0		F0	

	Vj	Qj	Vk	Qk
RS5				
RS6				
ALU1				
ALU2				

Unit	Remaining cycles
LDU1	1
LDU2	2
FPU1	3
ALU1	
ALU2	

6. **Cycle 6**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4		WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4			WAR I2, F6	RS4	
I5: LD F0, 16 (R1)				WAR I3, F0		
I6: LD F2, 16 (R2)				WAR I4, F2		
I7: ADDUI R1, R1, 8				WAR I5, R1		
I8: ADDUI R2, R2, 8				WAR I6, R2		
I9: ADD R3, R2, R1				RAW R1, R2		
I10: BNE R3, R4, LOOP				RAW I9, R3		

	Vj	Qj	Vk	Qk
RS1	F4		-	
RS2	F6		-	
LDU1	F4		-	
LDU2	F6		-	

	Vj	Qj	Vk	Qk
RS5				
RS6				
ALU1				
ALU2				

	Vj	Qj	Vk	Qk
RS3	F0		F0	
RS4	F2		F2	
FPU1	F0		F0	

Unit	Remaining cycles
LDU1	0
LDU2	1
FPU1	2
ALU1	
ALU2	

7. **Cycle 7**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4		WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4			WAR I2, F6	RS4	
I5: LD F0, 16 (R1)	7			WAR I3, F0	RS1	
I6: LD F2, 16 (R2)				WAR I4, F2		
I7: ADDUI R1, R1, 8				WAR I5, R1		
I8: ADDUI R2, R2, 8				WAR I6, R2		
I9: ADD R3, R2, R1				RAW R1, R2		
I10: BNE R3, R4, LOOP				RAW I9, R3		

	Vj	Qj	Vk	Qk
RS1	16		R1	
RS2	F6		-	
LDU1				
LDU2	F6		-	

	Vj	Qj	Vk	Qk
RS3	F0		F0	
RS4	F2		F2	
FPU1	F0		F0	

	Vj	Qj	Vk	Qk
RS5				
RS6				
ALU1				
ALU2				

Unit	Remaining cycles
LDU1	
LDU2	0
FPU1	1
ALU1	
ALU2	

8. **Cycle 8**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4	8	WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4			WAR I2, F6	RS4	
I5: LD F0, 16 (R1)	7	8		WAR I3, F0	RS1	LDU1
I6: LD F2, 16 (R2)	8			WAR I4, F2	RS2	
I7: ADDUI R1, R1, 8				WAR I5, R1		
I8: ADDUI R2, R2, 8				WAR I6, R2		
I9: ADD R3, R2, R1				RAW R1, R2		
I10: BNE R3, R4, LOOP				RAW I9, R3		

	Vj	Qj	Vk	Qk
RS1	16		R1	
RS2	16		R2	
LDU1	16		R1	
LDU2				

	Vj	Qj	Vk	Qk
RS3	F0		F0	
RS4	F2		F2	
FPU1	F0		F0	

	Vj	Qj	Vk	Qk
RS5				
RS6				
ALU1				
ALU2				

Unit	Remaining cycles
LDU1	4
LDU2	
FPU1	0
ALU1	
ALU2	

9. **Cycle 9**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4	8	WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4	9		WAR I2, F6	RS4	FPU1
I5: LD F0, 16 (R1)	7	8		WAR I3, F0	RS1	LDU1
I6: LD F2, 16 (R2)	8	9		WAR I4, F2	RS2	LDU2
I7: ADDUI R1, R1, 8	9			WAR I5, R1	RS5	
I8: ADDUI R2, R2, 8				WAR I6, R2		
I9: ADD R3, R2, R1				RAW R1, R2		
I10: BNE R3, R4, LOOP				RAW I9, R3		

	Vj	Qj	Vk	Qk
RS1	16		R1	
RS2	16		R2	
LDU1	16		R1	
LDU2	16		R2	

	Vj	Qj	Vk	Qk
RS5	R1		8	
RS6				
ALU1				
ALU2				

	Vj	Qj	Vk	Qk
RS3				
RS4	F2		F2	
FPU1	F2		F2	

Unit	Remaining cycles
LDU1	3
LDU2	4
FPU1	4
ALU1	
ALU2	

10. **Cycle 10**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4	8	WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4	9		WAR I2, F6	RS4	FPU1
I5: LD F0, 16 (R1)	7	8		WAR I3, F0	RS1	LDU1
I6: LD F2, 16 (R2)	8	9		WAR I4, F2	RS2	LDU2
I7: ADDUI R1, R1, 8	9	10		WAR I5, R1	RS5	ALU1
I8: ADDUI R2, R2, 8	10			WAR I6, R2	RS6	
I9: ADD R3, R2, R1				RAW R1, R2		
I10: BNE R3, R4, LOOP				RAW I9, R3		

	Vj	Qj	Vk	Qk
RS1	16		R1	
RS2	16		R2	
LDU1	16		R1	
LDU2	16		R2	

	Vj	Qj	Vk	Qk
RS5	R1		8	
RS6	R2		8	
ALU1	R1		8	
ALU2				

	Vj	Qj	Vk	Qk
RS3				
RS4	F2		F2	
FPU1	F2		F2	

Unit	Remaining cycles
LDU1	2
LDU2	3
FPU1	3
ALU1	2
ALU2	

11. **Cycle 11**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4	8	WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4	9		WAR I2, F6	RS4	FPU1
I5: LD F0, 16 (R1)	7	8		WAR I3, F0	RS1	LDU1
I6: LD F2, 16 (R2)	8	9		WAR I4, F2	RS2	LDU2
I7: ADDUI R1, R1, 8	9	10		WAR I5, R1	RS5	ALU1
I8: ADDUI R2, R2, 8	10	11		WAR I6, R2	RS6	ALU2
I9: ADD R3, R2, R1				RAW R1, R2		
I10: BNE R3, R4, LOOP				RAW I9, R3		

	Vj	Qj	Vk	Qk
RS1	16		R1	
RS2	16		R2	
LDU1	16		R1	
LDU2	16		R2	

	Vj	Qj	Vk	Qk
RS5	R1		8	
RS6	R2		8	
ALU1	R1		8	
ALU2	R2		8	

	Vj	Qj	Vk	Qk
RS3				
RS4	F2		F2	
FPU1	F2		F2	

Unit	Remaining cycles
LDU1	1
LDU2	2
FPU1	2
ALU1	1
ALU2	2

12. **Cycle 12**: "*The instruction 7 has finished executing, and it wants to write the result back. Why can't it do that?*" Because instruction 5 has higher priority, being the first in chronological order, and the common data bus can host only one value at a time.

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4	8	WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4	9		WAR I2, F6	RS4	FPU1
I5: LD F0, 16 (R1)	7	8	12	WAR I3, F0	RS1	LDU1
I6: LD F2, 16 (R2)	8	9		WAR I4, F2	RS2	LDU2
I7: ADDUI R1, R1, 8	9	10		WAR I5, R1	RS5	ALU1
I8: ADDUI R2, R2, 8	10	11		WAR I6, R2	RS6	ALU2
I9: ADD R3, R2, R1				RAW R1, R2		
I10: BNE R3, R4, LOOP				RAW I9, R3		

	Vj	Qj	Vk	Qk
RS1	16		R1	
RS2	16		R2	
LDU1	16		R1	
LDU2	16		R2	

	Vj	Qj	Vk	Qk
RS5	R1		8	
RS6	R2		8	
ALU1	R1		8	
ALU2	R2		8	

	Vj	Qj	Vk	Qk
RS3				
RS4	F2		F2	
FPU1	F2		F2	

Unit	Remaining cycles
LDU1	0
LDU2	1
FPU1	1
ALU1	0
ALU2	1

13. **Cycle 13**: Each functional unit has finished executing and is ready to write back the result. Again, since there is only one common data bus, we issue each instruction in chronological order. In this case, it is instruction 4.

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4	8	WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4	9	13	WAR I2, F6	RS4	FPU1
I5: LD F0, 16 (R1)	7	8	12	WAR I3, F0	RS1	LDU1
I6: LD F2, 16 (R2)	8	9		WAR I4, F2	RS2	LDU2
I7: ADDUI R1, R1, 8	9	10		WAR I5, R1	RS5	ALU1
I8: ADDUI R2, R2, 8	10	11		WAR I6, R2	RS6	ALU2
I9: ADD R3, R2, R1				RAW R1, R2		
I10: BNE R3, R4, LOOP				RAW I9, R3		

	Vj	Qj	Vk	Qk
RS1				
RS2	16		R2	
LDU1				
LDU2	16		R2	

	Vj	Qj	Vk	Qk
RS5	R1		8	
RS6	R2		8	
ALU1	R1		8	
ALU2	R2		8	

	Vj	Qj	Vk	Qk
RS3				
RS4	F2		F2	
FPU1	F2		F2	

Unit	Remaining cycles
LDU1	
LDU2	0
FPU1	0
ALU1	0 (−1)
ALU2	0

14. **Cycle 14**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4	8	WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4	9	13	WAR I2, F6	RS4	FPU1
I5: LD F0, 16 (R1)	7	8	12	WAR I3, F0	RS1	LDU1
I6: LD F2, 16 (R2)	8	9	14	WAR I4, F2	RS2	LDU2
I7: ADDUI R1, R1, 8	9	10		WAR I5, R1	RS5	ALU1
I8: ADDUI R2, R2, 8	10	11		WAR I6, R2	RS6	ALU2
I9: ADD R3, R2, R1				RAW R1, R2		
I10: BNE R3, R4, LOOP				RAW I9, R3		

	Vj	Qj	Vk	Qk
RS1				
RS2	16		R2	
LDU1				
LDU2	16		R2	

	Vj	Qj	Vk	Qk
RS5	R1		8	
RS6	R2		8	
ALU1	R1		8	
ALU2	R2		8	

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				

Unit	Remaining cycles
LDU1	
LDU2	0 (−1)
FPU1	
ALU1	0 (−2)
ALU2	0 (−1)

15. **Cycle 15**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4	8	WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4	9	13	WAR I2, F6	RS4	FPU1
I5: LD F0, 16 (R1)	7	8	12	WAR I3, F0	RS1	LDU1
I6: LD F2, 16 (R2)	8	9	14	WAR I4, F2	RS2	LDU2
I7: ADDUI R1, R1, 8	9	10	15	WAR I5, R1	RS5	ALU1
I8: ADDUI R2, R2, 8	10	11		WAR I6, R2	RS6	ALU2
I9: ADD R3, R2, R1				RAW R1, R2		
I10: BNE R3, R4, LOOP				RAW I9, R3		

	Vj	Qj	Vk	Qk
RS1				
RS2				
LDU1				
LDU2				

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				

	Vj	Qj	Vk	Qk
RS5	R1		8	
RS6	R2		8	
ALU1	R1		8	
ALU2	R2		8	

Unit	Remaining cycles
LDU1	
LDU2	
FPU1	
ALU1	0 (−3)
ALU2	0 (−2)

16. **Cycle 16**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4	8	WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4	9	13	WAR I2, F6	RS4	FPU1
I5: LD F0, 16 (R1)	7	8	12	WAR I3, F0	RS1	LDU1
I6: LD F2, 16 (R2)	8	9	14	WAR I4, F2	RS2	LDU2
I7: ADDUI R1, R1, 8	9	10	15	WAR I5, R1	RS5	ALU1
I8: ADDUI R2, R2, 8	10	11	16	WAR I6, R2	RS6	ALU2
I9: ADD R3, R2, R1	16			RAW R1, R2	RS5	
I10: BNE R3, R4, LOOP				RAW I9, R3		

	Vj	Qj	Vk	Qk
RS1				
RS2				
LDU1				
LDU2				

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				

	Vj	Qj	Vk	Qk
RS5	R2		R1	
RS6	R2		8	
ALU1				
ALU2	R2		8	

Unit	Remaining cycles
LDU1	
LDU2	
FPU1	
ALU1	
ALU2	0 (−3)

17. **Cycle 17**: This is the power of Tomasulo and its tag. This algorithm doesn't block the tenth instruction because it lacks the operand `R3`. Instead, it says, "*You cannot retrieve the value of register `R3`. No problem. Meanwhile, you can issue the instruction. When the `R3` value is ready, it will be sent to the Common Data Bus.*". Thus, instruction 10 waits on the Common Data Bus for the value of register `R3`. "*But how does it know when the value is ready?*" Simple. Use the tag logic. The value will be sent on the Common Data Bus with the key `ALU1`. Instruction 10 writes the tag "`ALU1`" to the reservation station, and when the Common Data Bus sends that tag, it takes the value.

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
`I1: SD F4, 0 (R1)`	1	2	6	None	RS1	LDU1
`I2: SD F6, 0 (R2)`	2	3	7	None	RS2	LDU2
`I3: FADD F4, F0, F0`	3	4	8	WAR I1, F4	RS3	FPU1
`I4: FADD F6, F2, F2`	4	9	13	WAR I2, F6	RS4	FPU1
`I5: LD F0, 16 (R1)`	7	8	12	WAR I3, F0	RS1	LDU1
`I6: LD F2, 16 (R2)`	8	9	14	WAR I4, F2	RS2	LDU2
`I7: ADDUI R1, R1, 8`	9	10	15	WAR I5, R1	RS5	ALU1
`I8: ADDUI R2, R2, 8`	10	11	16	WAR I6, R2	RS6	ALU2
`I9: ADD R3, R2, R1`	16	17		RAW R1, R2	RS5	ALU1
`I10: BNE R3, R4, LOOP`	17			RAW I9, R3	RS6	

	Vj	Qj	Vk	Qk
RS1				
RS2				
LDU1				
LDU2				

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				

	Vj	Qj	Vk	Qk
RS5	R2		R1	
RS6		ALU1	R4	
ALU1	R2		R1	
ALU2				

Unit	Remaining cycles
LDU1	
LDU2	
FPU1	
ALU1	2
ALU2	

19. **Cycle 19**: We skip cycle 18 because no operations have been performed. In instruction 10, the reservation station reads the value from the common data bus and stores it in memory. It is now ready to be executed!

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4	8	WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4	9	13	WAR I2, F6	RS4	FPU1
I5: LD F0, 16 (R1)	7	8	12	WAR I3, F0	RS1	LDU1
I6: LD F2, 16 (R2)	8	9	14	WAR I4, F2	RS2	LDU2
I7: ADDUI R1, R1, 8	9	10	15	WAR I5, R1	RS5	ALU1
I8: ADDUI R2, R2, 8	10	11	16	WAR I6, R2	RS6	ALU2
I9: ADD R3, R2, R1	16	17	19	RAW R1, R2	RS5	ALU1
I10: BNE R3, R4, LOOP	17			RAW I9, R3	RS6	

	Vj	Qj	Vk	Qk
RS1				
RS2				
LDU1				
LDU2				

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				

	Vj	Qj	Vk	Qk
RS5	R2		R1	
RS6	R3		R4	
ALU1	R2		R1	
ALU2				

Unit	Remaining cycles
LDU1	
LDU2	
FPU1	
ALU1	0
ALU2	

20. **Cycle 20**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4	8	WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4	9	13	WAR I2, F6	RS4	FPU1
I5: LD F0, 16 (R1)	7	8	12	WAR I3, F0	RS1	LDU1
I6: LD F2, 16 (R2)	8	9	14	WAR I4, F2	RS2	LDU2
I7: ADDUI R1, R1, 8	9	10	15	WAR I5, R1	RS5	ALU1
I8: ADDUI R2, R2, 8	10	11	16	WAR I6, R2	RS6	ALU2
I9: ADD R3, R2, R1	16	17	19	RAW R1, R2	RS5	ALU1
I10: BNE R3, R4, LOOP	17	20		RAW I9, R3	RS6	ALU2

	Vj	Qj	Vk	Qk
RS1				
RS2				
LDU1				
LDU2				

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				

	Vj	Qj	Vk	Qk
RS5				
RS6	R3		R4	
ALU1				
ALU2	R3		R4	

Unit	Remaining cycles
LDU1	
LDU2	
FPU1	
ALU1	
ALU2	2

22. **Cycle 22**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
I1: SD F4, 0 (R1)	1	2	6	None	RS1	LDU1
I2: SD F6, 0 (R2)	2	3	7	None	RS2	LDU2
I3: FADD F4, F0, F0	3	4	8	WAR I1, F4	RS3	FPU1
I4: FADD F6, F2, F2	4	9	13	WAR I2, F6	RS4	FPU1
I5: LD F0, 16 (R1)	7	8	12	WAR I3, F0	RS1	LDU1
I6: LD F2, 16 (R2)	8	9	14	WAR I4, F2	RS2	LDU2
I7: ADDUI R1, R1, 8	9	10	15	WAR I5, R1	RS5	ALU1
I8: ADDUI R2, R2, 8	10	11	16	WAR I6, R2	RS6	ALU2
I9: ADD R3, R2, R1	16	17	19	RAW R1, R2	RS5	ALU1
I10: BNE R3, R4, LOOP	17	20	22	RAW I9, R3	RS6	ALU2

	Vj	Qj	Vk	Qk
RS1				
RS2				
LDU1				
LDU2				

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				

	Vj	Qj	Vk	Qk
RS5				
RS6	R3		R4	
ALU1				
ALU2	R3		R4	

Unit	Remaining cycles
LDU1	
LDU2	
FPU1	
ALU1	
ALU2	0

Calculate the CPI.

Answer:

$$\text{CPI} = \frac{\#\text{ Clock Cycles}}{\#\text{ Instructions}} = \frac{22}{10} = 2.2$$

Question 3: Loop Unrolling

*Let's consider the following loop code where registers **R1** and **R2** are initialized to **0** and **R3** is initialized to **400**:*

```
LOOP:   LD F2, 0 (R1)
        LD F4, 0 (R2)
        FADD F6, F2, F4
        SD F6, 0 (R1)
        ADDUI R1, R1, 4
        ADDUI R2, R2, 4
        BNE R1, R3, LOOP
```

Answer to the following questions:

- *How many iterations of the loop?*

  Answer: 100 iterations. This is because register `R1` is initialized to `0` and is incremented by `4` each iteration until it reaches `400`:

  $$\text{Iterations} = \frac{400}{4} = 100$$

  So the loop executes `100` iterations before `R1` equals `R3` and the loop exits. Note that the `SD F6, 0(R1)` instruction stores `F6` back into memory at the address in `R1`; it does not write to `R1`. Therefore, only the `ADDUI` operator modifies the `R1` register.

- *How many instructions per iteration?*

  Answer: The number of instructions per iteration equals the number of instructions in the loop block, which is 7.

- *How many instructions due to the loop overhead per iteration?*

  Answer: When we write a loop in assembly, **some instructions do useful work**, and **some instructions are only there to control the loop**.

  - **Useful work**: instructions that do the actual computation we want (load data, do the calculation, store result).
  - **Overhead**: instructions that *only* exist to keep the loop going; updating counters, chang ing pointers, checking conditions, branching.

  In our case, we have 4 useful operations (2 loads, 1 add, and 1 store), and **3** that *only* exist to make the loop repeat: incrementing pointers/counters (`ADDUI R1`, `ADDUI R2`) and testing the loop condition (`BNE`). Loop overhead is extra work we must do every iteration to **make the loop repeat**, but it does not contribute to the real useful computation.

- *How many instructions are executed globally?*

  Answer: There are 7 instructions for each iteration. Since there are 100 iterations, **700 instructions** (7×100) are executed globally.

- *How many branch instructions are executed globally?*

  Answer: There is 1 branch instruction for each iteration, so 100 branch instructions (1×100) are executed globally.

Let's consider a loop unrolling version of the code with unrolling factor 2.

- *How many iterations of the unrolled loop?*

  Answer: Instead of repeating a small body many times, we manually replicate the body multiple times inside the loop; so each iteration does more useful work, but we run the loop fewer times (loop unrolling).

  If we unroll by a factor of 2, the loop body is rewritten to do 2 operations per iteration:

```
1  LOOP_UNROLLING_2:   LD F2, 0(R1)
2                      LD F4, 0(R2)
3                      FADD F6, F2, F4
4                      SD F6, 0(R1)
5
6                      LD F8, 4(R1)
7                      LD F10, 4(R2)
8                      FADD F12, F8, F10
9                      SD F12, 4(R1)
10
11                     ADDUI R1, R1, 8
12                     ADDUI R2, R2, 8
13                     BNE R1, R3, LOOP_UNROLLING_2
```

  So each iteration now does twice as much useful work and we only need half as many loop iterations:

$$\text{New Iteration Count} = \frac{\text{Original Iteration Count}}{\text{Unrolling Factor}} = \frac{100}{2} = 50$$

- *How many instructions per iteration?*

  Answer: The instructions are now 11.

- *How many instructions due to the loop overhead per iteration?*

  Answer: There are always 3 because we have unrolled useful work, not overhead work.

- *How many instructions are executed globally?*

  Answer: There are 11 instructions for each iteration. We repeat the cycle 50 times, so a total of 550 instructions are executed.

- *How many branch instructions are executed globally?*

  Answer: There is only one branch instruction per cycle and 50 iterations, so a total of 50 branch instructions are executed globally.

- *How many more registers are needed due to register renaming?*

  Answer: Original loop has 3 FP registers (`F2` holds first operand, `F4` holds second operand, and `F6` holds result). When we unroll the loop by factor 2, we do two sums at the same time. If we use the same registers, we overwrite the result of the first sum before we store it. So we need new registers for the second pair.

Therefore, we can conclude that:

- For the first sum: `F2`, `F4`, `F6`.
- For the second sum: `F8`, `F10`, `F12`.

So we need 3 more FP registers. In general, with an unrolling factor X, we need:

$$(X-1) \times (\text{number of live registers per body}) \quad \text{more registers}$$

- *What are the registers that need to be renamed?*

  Answer: As mentioned earlier, the registers that need to be renamed are `F2`, `F4`, and `F6`.

- *How much is the global instruction count decrease?*

  Answer: Before unrolling, the global instructions to execute was 700, and now 550. So we have 150 instructions less than before, that correspond to:

$$700 : 100 = 150 : x \quad \Rightarrow \quad \frac{150 \cdot 100}{700} = 21.428571429 = 21.43\%$$

Let's consider a loop unrolling version of the code with unrolling factor 4.

- *How many iterations of the unrolled loop?*

  Answer: If we unroll by a factor of 4, the loop body is rewritten to do 4 operations per iteration:

```
1  LOOP_UNROLLING_4:   LD F2, 0(R1)
2                      LD F4, 0(R2)
3                      FADD F6, F2, F4
4                      SD F6, 0(R1)
5
6                      LD F8, 4(R1)
7                      LD F10, 4(R2)
8                      FADD F12, F8, F10
9                      SD F12, 4(R1)
10
11                     LD F14, 8(R1)
12                     LD F16, 8(R2)
13                     FADD F18, F14, F16
14                     SD F18, 8(R1)
15
16                     LD F20, 12(R1)
17                     LD F22, 12(R2)
18                     FADD F24, F20, F22
19                     SD F24, 12(R1)
20
21                     ADDUI R1, R1, 16
22                     ADDUI R2, R2, 16
23                     BNE R1, R3, LOOP_UNROLLING_4
```

$$\text{New Iteration Count} = \frac{\text{Original Iteration Count}}{\text{Unrolling Factor}} = \frac{100}{4} = 25$$

- *How many instructions per iteration?*

  Answer: The instructions are now 19.

- *How many instructions due to the loop overhead per iteration?*

  Answer: There are always 3 instructions due to the loop overhead per iteration.

- *How many instructions are executed globally?*

  Answer:

  $$19 \text{ instructions} \cdot 25 \text{ iterations} = 475$$

- *How many branches are executed?*

  Answer: There is always 1 branch for each iteration, so a total of 25 branches are executed.

- *How many more registers are needed due to register renaming?*

  Answer:

  $$(4-1) \cdot 3 = 9 \text{ more registers}$$

- *What are the registers that need to be renamed?*

  Answer: It's always the three registers `F2`, `F4`, and `F6`, but three times.

- *How much is the global instruction count decrease?*

  Answer:

  $$700 : 100 = (700 - 475) : x \quad \Rightarrow \quad \frac{225 \cdot 100}{700} = 32.142857143 = 32.14\%$$

- *What is the best unrolling factor between 2 and 4? Explain why.*

  Answer: Factor 4 is preferable to Factor 2 because it reduces loop control overhead per useful operation even further. The number of iterations is halved again, from 50 to 25, and the overhead instructions per sum drop from 1.5 to 0.75.

  However, a higher unrolling factor requires more register names due to overlapping live values. In this case, factor 4 increases the demand for floating-point registers from 3 to 12. This increase must be handled by the hardware's register renaming; otherwise, spilling will occur. Therefore, factor 4 is preferable if the register file and rename logic are large enough to store all live values in registers without spilling to memory. Otherwise, factor 2 may be safer.

Quiz 4 - Branch Prediction

During the execution of a program, the number of mispredictions depends on the initialization and the size of the Branch History Table (BHT). True or False? Motivate.

Answer: A BHT stores the *history* (past behavior) of branches, usually with n-bit saturating counters. The size of the BHT determines **how many unique branches can be tracked independently**

- Small BHT → more aliasing → different branches share the same entry → more chance of misprediction.
- Large BHT → fewer collisions → better prediction accuracy .

At program start, the BHT entries are **initialized** (often all taken or all not take, or all weakly taken). If the actual branch pattern is opposite to the initialization, we'll have **mispredictions** until the counters adapt. So initialization affects startup behavior (how many initial mispredictions we get) and size of BHT affects aliasing (whether unrelated branches interfere with each other, more or fewer mispredictions overall).

The correct answer is: **True**, because the number of mispredictions depends on how the BHT entries are initialized and on the size of the BHT, which determines how many branches can be tracked without aliasing.

Quiz 5 - Branch Prediction

Consider the following assembly code executed by a processor with 1-entry 1-bit Branch History Table initialized as Taken:

```
1         ADDI R1, R0, 0
2         ADDI R2, R0, 100
3 LOOP:   BEQ R1, R2, DONE
4         ADD R5, R5, R4
5         ADDI R1, R1, 1
6         J LOOP
7 DONE:
```

*Assuming **only** the branch instruction accesses the BHT:*

- *How many accesses to the Branch History Table?*
- *How many mispredictions?*

True or False? Motivate.

Answer: `R1` starts at `0`. Each loop iteration: `R1` is incremented by `1`, and the loop stops when `R1` equals `100`. Therefore, `BEQ` is executed `101` times: One hundred times when `R1` is not equal to `R2` (the branch is not taken), and one time when `R1` is equal to `R2` (the branch is taken). For the first `100` iterations: The `BEQ` condition is false, so the branch is not taken. Therefore, the predictor's Taken prediction is incorrect, resulting in a mispredict. For the 101st `BEQ`, BHT predicts "Not Taken", but the branch is taken because `R1` is equal to `R2`, so there is a mispredict.

Quiz 6 - Superscalar Processors

*Considering superscalar processors, answer **TRUE** or **FALSE** to the following questions, **motivating your answers**.*

- *Doubling the number of issues reduces the CPI metric.*

  Answer: **True**. In superscalar processors, the CPI (Cycles Per Instruction) is ideally:

  $$\text{CPI}_{\text{ideal}} = \frac{1}{\text{issue width}}$$

  So if we double the issue width, theoretical CPI reduces. This is true only under the assumption of perfect conditions: infinite ILP, no hazards, no cache misses, no branch mispredictions, perfect decode, dispatch, execution bandwidth.

- *Out-of-order execution can be generated by data cache misses.*

  Answer: **True**. Cache misses introduce stalls for dependent instructions; out-of-order execution allows independent instructions to bypass stalled loads, maximizing utilization of execution units.

- *The introduction of dynamic branch predictors improves the CPI metric.*

  Answer: **True**. Dynamic branch predictors reduce the number of branch mispredictions by learning from past branch behavior. Fewer mispredictions mean fewer pipeline flushes and stalls, which directly lowers the average Cycles Per Instruction (CPI) and improves overall processor performance.

7.1.2 February 11

Exercise 1 - Dependency Analysis + Tomasulo

Let's consider the following assembly code containing multiple types of intra-loop dependences. Complete the following table by inserting all types of true-data-dependences, anti-dependences and output dependences for each instruction:

Answer:

I#	Type of instruction	Analysis of dependences: 1. True data dependence with I# for $Fx 2. Anti-dependence with I# for $Fy 3. Output-dependence with I# for $Fz
I0	`LD $F0, A($R0)`	None
I1	`LD $F2, B($R0)`	None
I2	`FADD $F4, $F0, $F2`	True data dependence with I0 for $F0 True data dependence with I1 for $F2
I3	`FADD $F6, $F4, $F4`	True data dependence with I2 for $F4
I4	`SD $F4, C($R0)`	True data dependence with I2 for $F4
I5	`SD $F6, D($R0)`	True data dependence with I3 for $F6
I6	`ADDI $R0, $R0, 4`	Anti-dependence with I0 for $R0 Anti-dependence with I1 for $R0 Anti-dependence with I4 for $R0 Anti-dependence with I5 for $R0
I7	`BNE $R0, $R1, LOOP`	True data dependence with I6 for $R0

Let's consider the previous assembly code to be executed on a CPU with dynamic scheduling based on TOMASULO algorithm with all cache HITS, a single Common Data Bus and:

- *2 RESERVATION STATIONS (**RS1**, **RS2**) with 2 LOAD/STORE units (**LDU1**, **LDU2**) with latency **6***
- *2 RESERVATION STATION (**RS3**, **RS4**) with 2 FP unit12 (**FPU1**, **FPU2**) with latency **2***
- *1 RESERVATION STATION (**RS5**) with 1 INT_ ALU/BR unit (**ALU1**) with latency **1***

Answer:

1. **Cycle 1**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
`LD $F0, A($R0)`	1	2	8	None	RS1	LDU1
`LD $F2, B($R0)`	2	3	9	None	RS2	LDU2
`FADD $F4, $F0, $F2`						
`FADD $F6, $F4, $F4`						
`SD $F4, C($R0)`						
`SD $F6, D($R0)`						
`ADDI $R0, $R0, 4`						
`BNE $R0, $R1, LOOP`						

	Vj	Qj	Vk	Qk
RS1	A		$R0	
RS2				
LDU1				
LDU2				

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				
FPU2				

	Vj	Qj	Vk	Qk
RS5				
ALU1				

Unit	Remaining cycles
LDU1	
LDU2	
FPU1	
FPU2	
ALU1	

2. **Cycle 2**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
`LD $F0, A($R0)`	1	2	8	None	`RS1`	`LDU1`
`LD $F2, B($R0)`	2	3	9	None	`RS2`	`LDU2`
`FADD $F4, $F0, $F2`						
`FADD $F6, $F4, $F4`						
`SD $F4, C($R0)`						
`SD $F6, D($R0)`						
`ADDI $R0, $R0, 4`						
`BNE $R0, $R1, LOOP`						

	Vj	Qj	Vk	Qk
RS1	A		$R0	
RS2	B		$R0	
LDU1	A		$R0	
LDU2				

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				
FPU2				

	Vj	Qj	Vk	Qk
RS5				
ALU1				

Unit	Remaining cycles
LDU1	6
LDU2	
FPU1	
FPU2	
ALU1	

3. **Cycle 3**: Here, instruction 2 has a RAW hazard with instructions 0 and 1 for registers `F0` and `F2`. Therefore, it uses the tag mechanism to wait for their values. In other words, the instruction is issued, but will not start until the values of `F0` and `F2` are forwarded to the Common Data Bus.

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
`LD $F0, A($R0)`	1	2	8	None	`RS1`	`LDU1`
`LD $F2, B($R0)`	2	3	9	None	`RS2`	`LDU2`
`FADD $F4, $F0, $F2`	3			`RAW I0, I1`	`RS3`	
`FADD $F6, $F4, $F4`						
`SD $F4, C($R0)`						
`SD $F6, D($R0)`						
`ADDI $R0, $R0, 4`						
`BNE $R0, $R1, LOOP`						

	Vj	Qj	Vk	Qk
RS1	A		$R0	
RS2	B		$R0	
LDU1	A		$R0	
LDU2	B		$R0	

	Vj	Qj	Vk	Qk
RS3		LDU1		LDU2
RS4				
FPU1				
FPU2				

	Vj	Qj	Vk	Qk
RS5				
ALU1				

Unit	Remaining cycles
LDU1	5
LDU2	6
FPU1	
FPU2	
ALU1	

4. **Cycle 4**: It's a similar situation to before. The second addition cannot be completed because it requires the result of the previous instruction. Therefore, it waits for the result on the common data bus.

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
`LD $F0, A($R0)`	1	2	8	None	`RS1`	`LDU1`
`LD $F2, B($R0)`	2	3	9	None	`RS2`	`LDU2`
`FADD $F4, $F0, $F2`	3			`RAW I0, I1`	`RS3`	
`FADD $F6, $F4, $F4`	4			`RAW I2, F6`	`RS4`	
`SD $F4, C($R0)`						
`SD $F6, D($R0)`						
`ADDI $R0, $R0, 4`						
`BNE $R0, $R1, LOOP`						

	Vj	Qj	Vk	Qk
RS1	A		$R0	
RS2	B		$R0	
LDU1	A		$R0	
LDU2	B		$R0	

	Vj	Qj	Vk	Qk
RS3		LDU1		LDU2
RS4		FPU1		FPU1
FPU1				
FPU2				

	Vj	Qj	Vk	Qk
RS5				
ALU1				

Unit	Remaining cycles
LDU1	4
LDU2	5
FPU1	
FPU2	
ALU1	

5. **Cycle 5**: The fourth instruction cannot be issued here because there are no reservation stations available, which constitutes a structural hazard. This means Tomasulo stops execution because it guarantees in-order issuance, meaning the instructions are issued in order. Some updates will be seen at the end of the first instruction (cycle 8).

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
`LD $F0, A($R0)`	1	2	8	None	`RS1`	`LDU1`
`LD $F2, B($R0)`	2	3	9	None	`RS2`	`LDU2`
`FADD $F4, $F0, $F2`	3			`RAW I0, I1`	`RS3`	
`FADD $F6, $F4, $F4`	4			`RAW I2, F6`	`RS4`	
`SD $F4, C($R0)`				`STRUCTURAL`		
`SD $F6, D($R0)`						
`ADDI $R0, $R0, 4`						
`BNE $R0, $R1, LOOP`						

	Vj	Qj	Vk	Qk
RS1	A		$R0	
RS2	B		$R0	
LDU1	A		$R0	
LDU2	B		$R0	

	Vj	Qj	Vk	Qk
RS3		LDU1		LDU2
RS4		FPU1		FPU1
FPU1				
FPU2				

	Vj	Qj	Vk	Qk
RS5				
ALU1				

Unit	Remaining cycles
LDU1	3
LDU2	4
FPU1	
FPU2	
ALU1	

8. **Cycle 8**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
LD $F0, A($R0)	1	2	8	None	RS1	LDU1
LD $F2, B($R0)	2	3	9	None	RS2	LDU2
FADD $F4, $F0, $F2	3			RAW I0, I1	RS3	
FADD $F6, $F4, $F4	4			RAW I2, F6	RS4	
SD $F4, C($R0)				STRUCTURAL		
SD $F6, D($R0)						
ADDI $R0, $R0, 4						
BNE $R0, $R1, LOOP						

	Vj	Qj	Vk	Qk
RS1	A		$R0	
RS2	B		$R0	
LDU1	A		$R0	
LDU2	B		$R0	

	Vj	Qj	Vk	Qk
RS3	$F0			LDU2
RS4		FPU1		FPU1
FPU1				
FPU2				

	Vj	Qj	Vk	Qk
RS5				
ALU1				

Unit	Remaining cycles
LDU1	0
LDU2	1
FPU1	
FPU2	
ALU1	

9. **Cycle 9**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
LD $F0, A($R0)	1	2	8	None	RS1	LDU1
LD $F2, B($R0)	2	3	9	None	RS2	LDU2
FADD $F4, $F0, $F2	3			RAW I0, I1	RS3	
FADD $F6, $F4, $F4	4			RAW I2, F6	RS4	
SD $F4, C($R0)	9			STR., RAW I2, $F4	RS1	
SD $F6, D($R0)						
ADDI $R0, $R0, 4						
BNE $R0, $R1, LOOP						

	Vj	Qj	Vk	Qk
RS1		RS3	$R0	
RS2	B		$R0	
LDU1				
LDU2	B		$R0	

	Vj	Qj	Vk	Qk
RS3	$F0		$F2	
RS4		FPU1		FPU1
FPU1				
FPU2				

	Vj	Qj	Vk	Qk
RS5				
ALU1				

Unit	Remaining cycles
LDU1	
LDU2	0
FPU1	
FPU2	
ALU1	

10. **Cycle 10**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
LD $F0, A($R0)	1	2	8	None	RS1	LDU1
LD $F2, B($R0)	2	3	9	None	RS2	LDU2
FADD $F4, $F0, $F2	3	10		RAW I0, I1	RS3	FPU1
FADD $F6, $F4, $F4	4			RAW I2, F6	RS4	
SD $F4, C($R0)	9			STR., RAW I2, $F4	RS1	
SD $F6, D($R0)	10			RAW I3, $F6	RS2	
ADDI $R0, $R0, 4						
BNE $R0, $R1, LOOP						

	Vj	Qj	Vk	Qk
RS1		RS3	$R0	
RS2		RS4	$R0	
LDU1				
LDU2				

	Vj	Qj	Vk	Qk
RS3	$F0		$F2	
RS4		FPU1		FPU1
FPU1	$F0		$F2	
FPU2				

	Vj	Qj	Vk	Qk
RS5				
ALU1				

Unit	Remaining cycles
LDU1	
LDU2	
FPU1	2
FPU2	
ALU1	

11. **Cycle 11**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
LD $F0, A($R0)	1	2	8	None	RS1	LDU1
LD $F2, B($R0)	2	3	9	None	RS2	LDU2
FADD $F4, $F0, $F2	3	10		RAW I0, I1	RS3	FPU1
FADD $F6, $F4, $F4	4			RAW I2, F6	RS4	
SD $F4, C($R0)	9			STR., RAW I2, $F4	RS1	
SD $F6, D($R0)	10			RAW I3, $F6	RS2	
ADDI $R0, $R0, 4	11				RS5	
BNE $R0, $R1, LOOP						

	Vj	Qj	Vk	Qk
RS1		RS3	$R0	
RS2		RS4	$R0	
LDU1				
LDU2				

	Vj	Qj	Vk	Qk
RS3	$F0		$F2	
RS4		FPU1		FPU1
FPU1	$F0		$F2	
FPU2				

	Vj	Qj	Vk	Qk
RS5	$R0		4	
ALU1				

Unit	Remaining cycles
LDU1	
LDU2	
FPU1	1
FPU2	
ALU1	

12. **Cycle 12**: The last operation cannot be issued due to a structural hazard in the `ALU1` functional unit. Note: The professor's solution has an error. The structural hazard must be on the `RS5` because it is the only one capable of performing the operation.

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
`LD $F0, A($R0)`	1	2	8	None	`RS1`	`LDU1`
`LD $F2, B($R0)`	2	3	9	None	`RS2`	`LDU2`
`FADD $F4, $F0, $F2`	3	10	12	`RAW I0, I1`	`RS3`	`FPU1`
`FADD $F6, $F4, $F4`	4			`RAW I2, F6`	`RS4`	
`SD $F4, C($R0)`	9			`STR., RAW I2, $F4`	`RS1`	
`SD $F6, D($R0)`	10			`RAW I3, $F6`	`RS2`	
`ADDI $R0, $R0, 4`	11	12			`RS5`	`ALU1`
`BNE $R0, $R1, LOOP`				`STRUCTURAL`		

	Vj	Qj	Vk	Qk
RS1	$F4		$R0	
RS2		RS4	$R0	
LDU1				
LDU2				

	Vj	Qj	Vk	Qk
RS3	$F0		$F2	
RS4	$F4		$F4	
FPU1	$F0		$F2	
FPU2				

	Vj	Qj	Vk	Qk
RS5	$R0		4	
ALU1	$R0		4	

Unit	Remaining cycles
LDU1	
LDU2	
FPU1	0
FPU2	
ALU1	1

13. **Cycle 13**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
LD $F0, A($R0)	1	2	8	None	RS1	LDU1
LD $F2, B($R0)	2	3	9	None	RS2	LDU2
FADD $F4, $F0, $F2	3	10	12	RAW I0, I1	RS3	FPU1
FADD $F6, $F4, $F4	4	13		RAW I2, F6	RS4	FPU2
SD $F4, C($R0)	9	13		STR., RAW I2, $F4	RS1	LDU1
SD $F6, D($R0)	10			RAW I3, $F6	RS2	
ADDI $R0, $R0, 4	11	12	13		RS5	ALU1
BNE $R0, $R1, LOOP				STRUCTURAL		

	Vj	Qj	Vk	Qk
RS1	$F4		$R0	
RS2		RS4	$R0	
LDU1	$F4		$R0	
LDU2				

	Vj	Qj	Vk	Qk
RS3				
RS4	$F4		$F4	
FPU1				
FPU2	$F4		$F4	

	Vj	Qj	Vk	Qk
RS5	$R0		4	
ALU1	$R0		4	

Unit	Remaining cycles
LDU1	6
LDU2	
FPU1	
FPU2	2
ALU1	0

14. **Cycle 14**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
LD $F0, A($R0)	1	2	8	None	RS1	LDU1
LD $F2, B($R0)	2	3	9	None	RS2	LDU2
FADD $F4, $F0, $F2	3	10	12	RAW I0, I1	RS3	FPU1
FADD $F6, $F4, $F4	4	13		RAW I2, F6	RS4	FPU2
SD $F4, C($R0)	9	13		STR., RAW I2, $F4	RS1	LDU1
SD $F6, D($R0)	10			RAW I3, $F6	RS2	
ADDI $R0, $R0, 4	11	12	13		RS5	ALU1
BNE $R0, $R1, LOOP	14			STRUCTURAL	RS5	

	Vj	Qj	Vk	Qk
RS1	$F4		$R0	
RS2		RS4	$R0	
LDU1	$F4		$R0	
LDU2				

	Vj	Qj	Vk	Qk
RS3				
RS4	$F4		$F4	
FPU1				
FPU2	$F4		$F4	

	Vj	Qj	Vk	Qk
RS5	$R0		$R1	
ALU1				

Unit	Remaining cycles
LDU1	5
LDU2	
FPU1	
FPU2	1
ALU1	

15. **Cycle 15**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
LD $F0, A($R0)	1	2	8	None	RS1	LDU1
LD $F2, B($R0)	2	3	9	None	RS2	LDU2
FADD $F4, $F0, $F2	3	10	12	RAW I0, I1	RS3	FPU1
FADD $F6, $F4, $F4	4	13	15	RAW I2, F6	RS4	FPU2
SD $F4, C($R0)	9	13		STR., RAW I2, $F4	RS1	LDU1
SD $F6, D($R0)	10			RAW I3, $F6	RS2	
ADDI $R0, $R0, 4	11	12	13		RS5	ALU1
BNE $R0, $R1, LOOP	14	15		STRUCTURAL	RS5	ALU1

	Vj	Qj	Vk	Qk
RS1	$F4		$R0	
RS2	$F4		$R0	
LDU1	$F4		$R0	
LDU2				

	Vj	Qj	Vk	Qk
RS3				
RS4	$F4		$F4	
FPU1				
FPU2	$F4		$F4	

	Vj	Qj	Vk	Qk
RS5	$R0		$R1	
ALU1	$R0		$R1	

Unit	Remaining cycles
LDU1	4
LDU2	
FPU1	
FPU2	0
ALU1	1

16. **Cycle 16**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
LD $F0, A($R0)	1	2	8	None	RS1	LDU1
LD $F2, B($R0)	2	3	9	None	RS2	LDU2
FADD $F4, $F0, $F2	3	10	12	RAW I0, I1	RS3	FPU1
FADD $F6, $F4, $F4	4	13	15	RAW I2, F6	RS4	FPU2
SD $F4, C($R0)	9	13		STR., RAW I2, $F4	RS1	LDU1
SD $F6, D($R0)	10	16		RAW I3, $F6	RS2	LDU2
ADDI $R0, $R0, 4	11	12	13		RS5	ALU1
BNE $R0, $R1, LOOP	14	15	16	STRUCTURAL	RS5	ALU1

	Vj	Qj	Vk	Qk
RS1	$F4		$R0	
RS2	$F4		$R0	
LDU1	$F4		$R0	
LDU2	$F4		$R0	

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				
FPU2				

	Vj	Qj	Vk	Qk
RS5	$R0		$R1	
ALU1	$R0		$R1	

Unit	Remaining cycles
LDU1	3
LDU2	6
FPU1	
FPU2	
ALU1	0

17. **Cycle 17**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
LD $F0, A($R0)	1	2	8	None	RS1	LDU1
LD $F2, B($R0)	2	3	9	None	RS2	LDU2
FADD $F4, $F0, $F2	3	10	12	RAW I0, I1	RS3	FPU1
FADD $F6, $F4, $F4	4	13	15	RAW I2, F6	RS4	FPU2
SD $F4, C($R0)	9	13		STR., RAW I2, $F4	RS1	LDU1
SD $F6, D($R0)	10	16		RAW I3, $F6	RS2	LDU2
ADDI $R0, $R0, 4	11	12	13		RS5	ALU1
BNE $R0, $R1, LOOP	14	15	16	STRUCTURAL	RS5	ALU1

	Vj	Qj	Vk	Qk
RS1	$F4		$R0	
RS2	$F4		$R0	
LDU1	$F4		$R0	
LDU2	$F4		$R0	

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				
FPU2				

	Vj	Qj	Vk	Qk
RS5				
ALU1				

Unit	Remaining cycles
LDU1	2
LDU2	5
FPU1	
FPU2	
ALU1	

19. **Cycle 19**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
LD $F0, A($R0)	1	2	8	None	RS1	LDU1
LD $F2, B($R0)	2	3	9	None	RS2	LDU2
FADD $F4, $F0, $F2	3	10	12	RAW I0, I1	RS3	FPU1
FADD $F6, $F4, $F4	4	13	15	RAW I2, F6	RS4	FPU2
SD $F4, C($R0)	9	13	19	STR., RAW I2, $F4	RS1	LDU1
SD $F6, D($R0)	10	16		RAW I3, $F6	RS2	LDU2
ADDI $R0, $R0, 4	11	12	13		RS5	ALU1
BNE $R0, $R1, LOOP	14	15	16	STRUCTURAL	RS5	ALU1

	Vj	Qj	Vk	Qk
RS1	$F4		$R0	
RS2	$F4		$R0	
LDU1	$F4		$R0	
LDU2	$F4		$R0	

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				
FPU2				

	Vj	Qj	Vk	Qk
RS5				
ALU1				

Unit	Remaining cycles
LDU1	0
LDU2	3
FPU1	
FPU2	
ALU1	

22. **Cycle 22**:

Instruction	Issue	Start Exec	Write Res	Hazard	RSi	Unit
`LD $F0, A($R0)`	1	2	8	None	`RS1`	`LDU1`
`LD $F2, B($R0)`	2	3	9	None	`RS2`	`LDU2`
`FADD $F4, $F0, $F2`	3	10	12	`RAW I0, I1`	`RS3`	`FPU1`
`FADD $F6, $F4, $F4`	4	13	15	`RAW I2, F6`	`RS4`	`FPU2`
`SD $F4, C($R0)`	9	13	19	`STR., RAW I2, $F4`	`RS1`	`LDU1`
`SD $F6, D($R0)`	10	16	22	`RAW I3, $F6`	`RS2`	`LDU2`
`ADDI $R0, $R0, 4`	11	12	13		`RS5`	`ALU1`
`BNE $R0, $R1, LOOP`	14	15	16	`STRUCTURAL`	`RS5`	`ALU1`

	Vj	Qj	Vk	Qk
RS1				
RS2	$F4		$R0	
LDU1				
LDU2	$F4		$R0	

	Vj	Qj	Vk	Qk
RS3				
RS4				
FPU1				
FPU2				

	Vj	Qj	Vk	Qk
RS5				
ALU1				

Unit	Remaining cycles
LDU1	
LDU2	0
FPU1	
FPU2	
ALU1	

*Calculate the **CPI**.*

Answer:

$$\text{CPI} = \frac{\#\text{ Clock Cycles}}{\#\text{ Instructions}} = \frac{22}{8} = 2.75$$

Exercise 2 - VLIW Scheduling

Let's consider the following LOOP code:

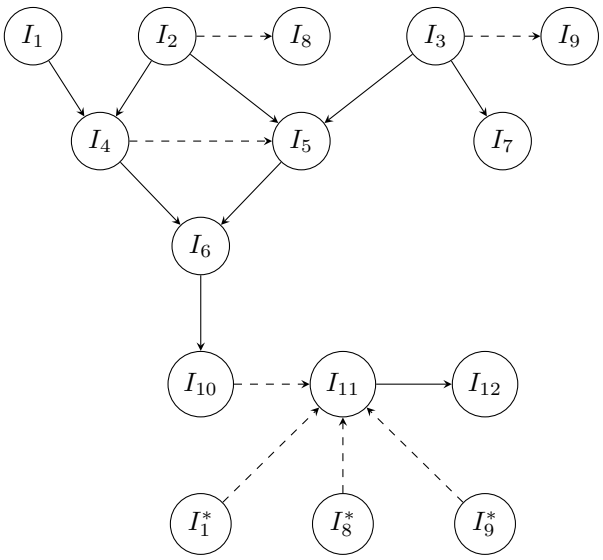
```
LOOP:   LD F1, A(R1)
        LD F2, A(R2)
        LD F3, A(R3)
        FADD F1, F1, F2
        FADD F2, F2, F3
        FMUL F1, F1, F2
        FADD F3, F3, F3
        ADDUI R2, R1, 8
        ADDUI R3, R1, 8
        SD F1, B(R1)
        ADDUI R1, R1, 4
        BNE R1, R6, LOOP
```

Given a **3-issue VLIW** machine with **fully pipelined functional units**:

- **1 Memory Unit with 3 cycles latency**
- **1 FP ALU with 2 cycles latency**
- **1 Integer ALU with 1 cycle latency to next Int/FP & 2 cycle latency to next Branch**

The branch is completed with 1 cycle delay slot (branch solved in ID stage). **No branch prediction**. In the Register File, it is possible to read and write at the same address at the same clock cycle. *Considering one iteration of the loop, complete the following table by using the* ***list-based scheduling*** (do NOT introduce any software pipelining, loop unrolling and modifications to loop indexes) on the *4-issue VLIW machine including the BRANCH DELAY SLOT*. Please do not write in NOPs.

Answer:



	Memory Unit	Floating Point Unit	Integer Unit
C1	`LD F1, A(R1)`		
C2	`LD F2, A(R2)`		`ADDUI R2, R1, 8`
C3	`LD F3, A(R3)`		`ADDUI R3, R1, 8`
C4			
C5		`FADD F1, F1, F2`	
C6		`FADD F2, F2, F3`	
C7		`FADD F3, F3, F3`	
C8		`FMUL F1, F1, F2`	
C9			
C10	`SD F1, B(R1)`		`ADDUI R1, R1, 4`
C11			
C12			`BNE R1, R2, LOOP`
C13			(delay slot)
C14			
C15			

1. *How long is the critical path?*

   Answer: The critical path is determined by the latency of the last instruction, `BNE`, so it is **13**.

2. *How much is the code efficiency?*

   Answer: Efficiency is measured as follows:

$$\text{Efficiency} = \frac{\text{Instruction Count}}{\#\text{ cycles} \times \#\text{ issues}} = \frac{12}{13 \cdot 3} = 0.307692308 = 30.77\%$$

Exercise 3 - MESI Protocol

Let's consider the following access patterns on a **4-processor** system with a direct-mapped, write-back cache with one cache block per processor and a two-cache block memory.

Assume the **MESI protocol** is used with **write-back** caches, **write-allocate**, and **write-invalidate** of other caches. *Please COMPLETE the following table:*

Cycle	After Op.	P0 cache block state	P1 cache block state	P2 cache block state	P3 cache block state	Mem. at bl. 0 up to date?	Mem. at bl. 1 up to date?
1	P0: Read Bl. 1	Excl (1)	Invalid	Invalid	Invalid	Yes	Yes
2	P2: Read Bl. 0	Excl (1)	Invalid	Excl (0)	Invalid	Yes	Yes
3	P3: Read Bl. 0						
4	P1: Write Bl. 0						
5	P0: Write Bl. 1						
6	P3: Read Bl. 1						

Answer: **MESI** is a **cache coherence protocol**. It guarantees that multiple caches in a **shared-memory multiprocessor** see a **coherent view** of main memory.

Why is it needed? When multiple processors have private caches and share a memory space, they may each hold *local copies* of the same memory block. If one processor modifies its copy, the other copies may become **stale** (incoherent). The MESI protocol ensures that all copies are kept consistent, meaning we never use an old value when we shouldn't.

MESI States Each block in a cache can be in one of **4 states**:

- **M** (**Modified**): Block is valid, *dirty* (modified). It *differs* from main memory. This cache *owns* the latest value.
- **E** (**Exclusive**): Block is valid, *clean*. It matches memory. This cache is the *only* owner.
- **S** (**Shared**): Block is valid, clean, and may be in *multiple* caches. All have the same value as memory.
- **I** (**Invalid**): Block is *not* valid in this cache. Must be fetched on access.

Coherence is enforced by invalidating or updating other caches:

- If a cache writes to a block, it must invalidate all other caches that have that block (**Write-Invalidate**).
- If a cache needs a block held dirty by another, that cache must supply the up-to-date data (and possibly write back to memory).

What is Direct-Mapped Cache? **Direct-Mapped** is the simplest cache mapping:

- Each memory block maps to **exactly one** cache block.
- There's only one place in the cache where a given memory block can go.

In this exercise, each processor has a direct-mapped cache with **only one block**, so each processor can store **only one block at a time**.

What does Write-Back mean? Caches can be:

- **Write-Through**: Every write to the cache is also immediately written to main memory. Simpler, but more traffic.
- **Write-Back**: Writes go only to the cache at first. The block is marked *dirty*. The updated block is written back to main memory only when it is *evicted* or needed by another processor.

In MESI: The **Modified (M)** state marks blocks that are *dirty*. These blocks must be written back before eviction or when another processor requests them.

What does Write-Allocate mean? **Write-Allocate** means that when a processor writes to a block that is not present in its cache (*write miss*), it:

- **Allocates**: It *loads* the block into its cache.
- Then performs the write in the cache.

This is typical with write-back caches: the block must be in the cache so that it can later be written back to memory if needed.

What does "One cache block per processor" mean? In this exercise:

- Each processor has **one slot in its cache**.
- That slot can hold *one memory block* at a time.
- So if a processor needs to read/write a new block, it may need to *evict* the old one (and write it back if it's dirty).

What does "Two-cache-block memory" mean? This means that **main memory** has **only 2 data blocks** available: **block 0** and **block 1**. So:

- The whole example focuses only on those two blocks.
- Any cache action involves either block 0 or block 1.
- The goal is to track how these two blocks move between the processors' caches and the main memory.

In summary, the entire scenario is as follows:

- A *tiny system* with 4 processors.
- Each has a *tiny cache* (1 block).
- The system has only 2 memory blocks.
- Using MESI, caches cooperate to keep shared blocks consistent.
- *Write-back*: modified blocks may not immediately update memory.
- *Write-allocate*: writes on misses bring the block in.

0. In the **initial state**, everything is up to date, but each block is invalid.
1. **Operation 1**: "`P0: Read block 1`". Since all the blocks are invalid, this is a *cache miss* operation. Therefore, processor 0 (`P0`) loads block 1 from memory. Since no one else has it, the state is **Exclusive**. The memory is still up to date because it was "*read-by*", not "*written-to*".

Cycle	After Op.	P0 cache block state	P1 cache block state	P2 cache block state	P3 cache block state	Mem. at bl. 0 up to date?	Mem. at bl. 1 up to date?
0		Invalid	Invalid	Invalid	Invalid	Yes	Yes
1	P0: Read Bl. 1	Excl (1)	Invalid	Invalid	Invalid	Yes	Yes

2. **Operation 2**: "`P2: Read Bl. 0`". Processor 2 has an invalid cache block, resulting in a *cache miss*. Since nobody else has block 0, processor 2 loads it. It gets it in **Exclusive** mode. Again, the memory is up to date because it was read.

Cycle	After Op.	P0 cache block state	P1 cache block state	P2 cache block state	P3 cache block state	Mem. at bl. 0 up to date?	Mem. at bl. 1 up to date?
0		Invalid	Invalid	Invalid	Invalid	Yes	Yes
1	P0: Read Bl. 1	Excl (1)	Invalid	Invalid	Invalid	Yes	Yes
2	P2: Read Bl. 0	Excl (1)	Invalid	Excl (0)	Invalid	Yes	Yes

3. **Operation 3**: "`P3: Read Bl. 0`". Processor 3 has an invalid cache block, resulting in a *cache miss*. This time, however, a copy of block 0 is in processor 2's cache, so processor 3 can request a copy of it. Now, since the value of block 0 is shared by different caches, we transition from **Exclusive** to **Shared**. Finally, the memory remains updated (no writes).

Cycle	**After Op.**	**P0 cache block state**	**P1 cache block state**	**P2 cache block state**	**P3 cache block state**	**Mem. at bl. 0 up to date?**	**Mem. at bl. 1 up to date?**
0		Invalid	Invalid	Invalid	Invalid	Yes	Yes
1	P0: Read Bl. 1	Excl (1)	Invalid	Invalid	Invalid	Yes	Yes
2	P2: Read Bl. 0	Excl (1)	Invalid	Excl (0)	Invalid	Yes	Yes
3	P3: Read Bl. 0	Excl (1)	Invalid	Shr (0)	Shr (0)	Yes	Yes

4. **Operation 4**: "`P1: Write Bl. 0`". Since Processor 1 does not have Block 0, there is a cache miss. Block 0 is in processors 2 and 3's caches in Shared mode. So, MESI:

   (a) P1 needs exclusive access, so it invalidates all other copies.

   (b) P2 and P3 are invalidated by P1 and become invalid.

   (c) P1 loads block 0, writes it, and transitions from **Exclusive** to **Modified**.

   Since it's write-back, the memory for block 0 becomes stale (marked as out-of-date).

Cycle	**After Op.**	**P0 cache block state**	**P1 cache block state**	**P2 cache block state**	**P3 cache block state**	**Mem. at bl. 0 up to date?**	**Mem. at bl. 1 up to date?**
0		Invalid	Invalid	Invalid	Invalid	Yes	Yes
1	P0: Read Bl. 1	Excl (1)	Invalid	Invalid	Invalid	Yes	Yes
2	P2: Read Bl. 0	Excl (1)	Invalid	Excl (0)	Invalid	Yes	Yes
3	P3: Read Bl. 0	Excl (1)	Invalid	Shr (0)	Shr (0)	Yes	Yes
4	P1: Write Bl. 0	Excl (1)	Mod (0)	Invalid	Invalid	No	Yes

5. **Operation 5**: "`P0: Write Bl. 1`". Processor 0 already has Block 1 in **Exclusive** mode. When it writes, it switches from Exclusive to **Modified** mode. Because a write was made, the memory is no longer up to date. Since no one has block 1, it doesn't invalidate the other caches.

Cycle	After Op.	P0 cache block state	P1 cache block state	P2 cache block state	P3 cache block state	Mem. at bl. 0 up to date?	Mem. at bl. 1 up to date?
0		Invalid	Invalid	Invalid	Invalid	Yes	Yes
1	P0: Read Bl. 1	Excl (1)	Invalid	Invalid	Invalid	Yes	Yes
2	P2: Read Bl. 0	Excl (1)	Invalid	Excl (0)	Invalid	Yes	Yes
3	P3: Read Bl. 0	Excl (1)	Invalid	Shr (0)	Shr (0)	Yes	Yes
4	P1: Write Bl. 0	Excl (1)	Mod (0)	Invalid	Invalid	No	Yes
5	P0: Write Bl. 1	Mod (1)	Mod (0)	Invalid	Invalid	No	No

6. **Operation 6**: "`P3: Read Bl. 1`". Since processor 3 does not have block 1, there is a *cache miss*. Processor 0 has block 1 in modified mode.
   (a) Another processor needs the data.
   (b) P0 must supply the *latest value*, so P0 does *write-back* or sends directly.
   (c) Result: P3 loads up-to-date block, and both now have it in **Shared**.

   Now the memory for block 1 is clean again (assuming P0 did a write-back).

Cycle	After Op.	P0 cache block state	P1 cache block state	P2 cache block state	P3 cache block state	Mem. at bl. 0 up to date?	Mem. at bl. 1 up to date?
0		Invalid	Invalid	Invalid	Invalid	Yes	Yes
1	P0: Read Bl. 1	Excl (1)	Invalid	Invalid	Invalid	Yes	Yes
2	P2: Read Bl. 0	Excl (1)	Invalid	Excl (0)	Invalid	Yes	Yes
3	P3: Read Bl. 0	Excl (1)	Invalid	Shr (0)	Shr (0)	Yes	Yes
4	P1: Write Bl. 0	Excl (1)	Mod (0)	Invalid	Invalid	No	Yes
5	P0: Write Bl. 1	Mod (1)	Mod (0)	Invalid	Invalid	No	No
6	P3: Read Bl. 1	Shr (1)	Mod (0)	Invalid	Shr (1)	No	Yes

Note: Read from **Modified** forces a write-back (or direct transfer) to maintain consistency.

Exercise 4 - Reordered Buffer

Let's consider the following assembly loop where registers ***$R1*** *and* ***$R2*** *are initialized at 0 and 40 respectively:*

```
L0:     LD $F2, 0 ($R1)
L1:     ADDD $F4, $F2, $F2
L2:     SD $F4, 0 ($R1)
L3:     ADDI $R1, $R1, 4
L4:     BNE $R1, $R2, L0 # branch predicted as taken
```

- *How many loop iterations?*

  Answer: The register `R1` is incremented by 4 at each iteration:

  $$\text{Iterations} = \frac{40}{4} = 10$$

- *How many instructions per iteration?*

  Answer: 5 instructions.

- *How many control vs datapath instructions?*

  Answer: To answer the question, we count **how many instructions in the loop control the flow** vs **how many do actual data processing**. In our loop:

  ```
  L0:     LD $F2, 0($R1)          # datapath: load data
  L1:     ADDD $F4, $F2, $F2      # datapath: compute
  L2:     SD $F4, 0($R1)          # datapath: store data
  L3:     ADDI $R1, $R1, 4        # control: index update
  L4:     BNE $R1, $R2, L0        # control: conditional branch
  ```

  - **Data path instructions**: `LD`, `ADDD`, `SD`, 3 instructions.
  - **Control instructions**: `ADDI` (updates loop index), `BNE` (branch), 2 instructions.

  So, the answer is 2 control vs 3 datapath instructions. In general:

  - **Control instructions**: branches, jumps, index updates.
  - **Datapath instructions**: load, store, ALU/FPU operations that compute or move data.

Write the unrolled version of the loop with unrolling factor 2 by using Register Renaming.

Answer:

```
L0:     LD $F2, 0($R1)          # datapath: load data
L1:     ADDD $F4, $F2, $F2      # datapath: compute
L2:     SD $F4, 0($R1)          # datapath: store data
L3:     LD $F6, 4($R1)          # datapath: load data
L4:     ADDD $F8, $F6, $F6      # datapath: compute
L5:     SD $F8, 0($R1)          # datapath: store data
L6:     ADDI $R1, $R1, 8        # control: index update
L7:     BNE $R1, $R2, L0        # control: conditional branch
```

- *How many loop iterations?*

  Answer: The register `R1` is incremented by 8 at each iteration:

$$\text{Iterations} = \frac{40}{8} = 5$$

- *How many instructions per iteration?*

  Answer: 8 instructions.

- *How many control vs datapath instructions?*

  Answer: 2 control vs 6 datapath instructions.

*Execute the unrolled version of the loop by the **Speculative Tomasulo** architecture with a 10-entry ROB and:*

- *4 Load Buffers (Load1, Load2, Load3, Load4);*
- *4 FP Reservation Stations (FP1, FP2, FP3, FP4)*
- *2 Integer Reservation Stations (Int1, Int2)*

*Complete the ROB and the Rename Table until the ROB becomes **full** while the first instruction is still in execution due to a cache miss (*):*

ROB#	Instruction	Dest.	Res. Alloc.	Ready/Status	Spec.
ROB0	L0: LD $F2, 0 ($R1)	$F2	Load1	No, exec.(*)	No
ROB1	L1: ADDD $F4, $F2, $F2	$F4	FP1	No, issued	No
ROB2					
ROB3					
ROB4					
ROB5					
ROB6					
ROB7					
ROB8					
ROB9					

ROB Table

Reg #	ROB #
$F0	-
$F2	
$F4	
$F6	
$F8	

Rename Table

Answer:

1. First, we allocate the entries given by the exercise in the Rename Table:

Reg #	ROB #
$F0	-
$F2	ROB0
$F4	ROB1
$F6	
$F8	

Rename Table

2. We issue the L2 instruction. The operation's destination is memory. The status has not been issued yet because we are filling the ROB due to a cache miss, and the processor is waiting for the result of the operation.

ROB#	Instruction	Dest.	Res. Alloc.	Ready/Status	Spec.
ROB0	L0: LD $F2, 0 ($R1)	$F2	Load1	No, exec.(*)	No
ROB1	L1: ADDD $F4, $F2, $F2	$F4	FP1	No, issued	No
ROB2	L2: SD $F4, 0($R1)	Mem	-	No, issued	No
ROB3					
ROB4					
ROB5					
ROB6					
ROB7					
ROB8					
ROB9					

ROB Table

3. We issue the `L3` instruction. The destination is the left operand, register `F6`. We use `Load 2` as the functional unit.

ROB#	Instruction	Dest.	Res. Alloc.	Ready/Status	Spec.
ROB0	L0: LD $F2, 0 ($R1)	$F2	Load1	No, exec.(*)	No
ROB1	L1: ADDD $F4, $F2, $F2	$F4	FP1	No, issued	No
ROB2	L2: SD $F4, 0($R1)	Mem	-	No, issued	No
ROB3	L3: LD $F6, 4($R1)	$F6	Load2	No, issued	No
ROB4					
ROB5					
ROB6					
ROB7					
ROB8					
ROB9					

ROB Table

Reg #	ROB #
$F0	-
$F2	ROB0
$F4	ROB1
$F6	ROB3
$F8	

Rename Table

4. We issue the L4 instruction.

ROB#	Instruction	Dest.	Res. Alloc.	Ready/Status	Spec.
ROB0	L0: LD $F2, 0 ($R1)	$F2	Load1	No, exec.(*)	No
ROB1	L1: ADDD $F4, $F2, $F2	$F4	FP1	No, issued	No
ROB2	L2: SD $F4, 0($R1)	Mem	-	No, issued	No
ROB3	L3: LD $F6, 4($R1)	$F6	Load2	No, issued	No
ROB4	L4: ADDD $F8, $F6, $F6	$F8	FP2	No, issued	No
ROB5					
ROB6					
ROB7					
ROB8					
ROB9					

ROB Table

Reg #	ROB #
$F0	-
$F2	ROB0
$F4	ROB1
$F6	ROB3
$F8	ROB4

Rename Table

5. We issue the L5 instruction.

ROB#	Instruction	Dest.	Res. Alloc.	Ready/Status	Spec.
ROB0	L0: LD $F2, 0 ($R1)	$F2	Load1	No, exec.(*)	No
ROB1	L1: ADDD $F4, $F2, $F2	$F4	FP1	No, issued	No
ROB2	L2: SD $F4, 0($R1)	Mem	-	No, issued	No
ROB3	L3: LD $F6, 4($R1)	$F6	Load2	No, issued	No
ROB4	L4: ADDD $F8, $F6, $F6	$F8	FP2	No, issued	No
ROB5	L5: SD $F8, 0($R1)	Mem	-	No, issued	No
ROB6					
ROB7					
ROB8					
ROB9					

ROB Table

6. We issue the L6 instruction.

ROB#	Instruction	Dest.	Res. Alloc.	Ready/Status	Spec.
ROB0	L0: LD $F2, 0 ($R1)	$F2	Load1	No, exec.(*)	No
ROB1	L1: ADDD $F4, $F2, $F2	$F4	FP1	No, issued	No
ROB2	L2: SD $F4, 0($R1)	Mem	-	No, issued	No
ROB3	L3: LD $F6, 4($R1)	$F6	Load2	No, issued	No
ROB4	L4: ADDD $F8, $F6, $F6	$F8	FP2	No, issued	No
ROB5	L5: SD $F8, 0($R1)	Mem	-	No, issued	No
ROB6	L6: ADDI $R1, $R1, 8	$R1	Int1	No, issued	No
ROB7					
ROB8					
ROB9					

ROB Table

7. We issue the L7 instruction.

ROB#	Instruction	Dest.	Res. Alloc.	Ready/Status	Spec.
ROB0	L0: LD $F2, 0 ($R1)	$F2	Load1	No, exec.(*)	No
ROB1	L1: ADDD $F4, $F2, $F2	$F4	FP1	No, issued	No
ROB2	L2: SD $F4, 0($R1)	Mem	-	No, issued	No
ROB3	L3: LD $F6, 4($R1)	$F6	Load2	No, issued	No
ROB4	L4: ADDD $F8, $F6, $F6	$F8	FP2	No, issued	No
ROB5	L5: SD $F8, 0($R1)	Mem	-	No, issued	No
ROB6	L6: ADDI $R1, $R1, 8	$R1	Int1	No, issued	No
ROB7	L7: BNE $R1, $R2, L0	-	Int2	No, issued	No
ROB8					
ROB9					

ROB Table

8. We issue `L0` and `L1`. Since we need to fill out the entire ROB table and we don't know whether the branch will be true or false, we made a prediction. Therefore, each speculative entry will be set to true.

ROB#	Instruction	Dest.	Res. Alloc.	Ready/Status	Spec.
ROB0	L0: LD $F2, 0 ($R1)	$F2	Load1	No, exec.(*)	No
ROB1	L1: ADDD $F4, $F2, $F2	$F4	FP1	No, issued	No
ROB2	L2: SD $F4, 0($R1)	Mem	-	No, issued	No
ROB3	L3: LD $F6, 4($R1)	$F6	Load2	No, issued	No
ROB4	L4: ADDD $F8, $F6, $F6	$F8	FP2	No, issued	No
ROB5	L5: SD $F8, 0($R1)	Mem	-	No, issued	No
ROB6	L6: ADDI $R1, $R1, 8	$R1	Int1	No, issued	No
ROB7	L7: BNE $R1, $R2, L0	-	Int2	No, issued	No
ROB8	L0: LD $F2, 0($R1)	$F2	Load3	No, issued	Yes
ROB9	L1: ADDD $F4, $F2, $F2	$F4	FP3	No, issued	Yes

ROB Table

Reg #	ROB #
$F0	-
$F2	~~ROB0~~ ROB8
$F4	~~ROB1~~ ROB9
$F6	ROB3
$F8	ROB4

Rename Table

Question 1: Instruction-Level and Thread-Level Parallelism

Modern processors exploit **Instruction Level Parallelism** and **Thread Level Parallelism**. *Answer to the following questions:*

- *Explain the main concepts for each approach (Instruction Level Parallelism ILP, Thread Level Parallelism TLP).*

  Answer:

  - **Instruction-Level Parallelism (ILP)**. ILP means **executing multiple independent instructions from a single instruction stream in parallel**. The goal is to exploit *fine-grain* parallelism *within* a single thread.

    Many instructions in a program do not depend on each other, so they can be executed simultaneously if the hardware can exploit this.

    * **Pipelining**: Overlap the execution of multiple instructions by dividing execution into stages (fetch, decode, execute, memory, write-back). While one instruction is being decoded, the next can be fetched, and so on.
    * **Superscalar execution**: Issue multiple instructions per clock cycle. A superscalar processor has multiple pipelines.
    * **Out-of-order execution**: Hardware reorders instructions dynamically to maximize parallelism while preserving correctness.
    * **Speculation**: Predict branches and execute along the predicted path before knowing if it is correct.
    * **Register renaming**: Remove false dependencies (WAW, WAR hazards) by dynamically mapping logical registers to physical registers.

    ILP aims to increase *single-thread performance*. Performance improvement depends on the amount of independent instructions and on how well the hardware can extract and schedule them.

  - **Thread-Level Parallelism (TLP)**. TLP means **executing multiple independent instruction streams (threads) in parallel**. This exploits coarse-grain parallelism between threads.

    Many programs or tasks can be decomposed into multiple independent threads. If there are multiple cores, these threads can run simultaneously.

    * **Multithreading**: Hardware executes instructions from multiple threads on a single core to better utilize pipeline resources (e.g., SMT, Simultaneous Multithreading).
    * **Multiprocessing (Multicore)**: Multiple cores, each with its own pipeline(s), execute different threads.
    * **Clusters or Shared Memory Multiprocessors**: Multiple processors/cores share memory; they communicate and synchronize to solve larger problems.

    To coordinate threads, programmers use synchronization primitives (locks, barriers). Memory consistency models define how memory updates become visible.

TLP increases *throughput* and/or *responsiveness*. It is the main driver behind modern multicore processors. It shifts the burden to software: the program must be parallelized.

Aspect	ILP	TLP
Parallelism granularity	Fine-grain: within a single thread	Coarse-grain: multiple threads
Handled by	Mostly hardware	Both hardware & software
Example	Pipelining, superscalar, out-of-order	Multicore CPUs, multithreading
Goal	Speed up single program thread	Speed up multi-threaded workload

Table 29: Comparison ILP vs TLP.

- *Which type of technique can be applied in superscalar processors to combine both ILP and TLP?*

  Answer: In a superscalar processor, the main technique to combine both ILP and TLP is **Simultaneous Multithreading (SMT)**.

  - A superscalar processor can issue multiple instructions per cycle, this exploits Instruction-Level Parallelism (ILP).
  - However, sometimes a single thread cannot supply enough independent instructions to keep all issue slots busy (due to data dependencies, cache misses, or control hazards).
  - To keep the pipelines full, Simultaneous Multithreading allows **instructions from multiple hardware threads** to be **fetched and issued in the same cycle**.
  - In other words, SMT **interleaves instructions from different threads into the same superscalar pipeline**, and this combines ILP (*issuing multiple instructions per cycle*) with TLP (*multiple threads*).

  The most common SMT implementation is Intel's Hyper-Threading technology.

  In summary, **SMT** leverages the *wide superscalar pipeline* to execute instructions from multiple threads *at the same time*. This effectively *combines ILP and TLP* on a singole core.

- *Explain what type of **instruction scheduling** is used in superscalar processors to combine both ILP and TLP?*

  Answer: In superscalar processors, instruction scheduling is about **deciding when and in what order instructions issue to the multiple pipelines**. Scheduling ensures that dependencies are respected and hardware resources are efficiently used.

  - **Static scheduling**: The **compiler** reorders instructions **before runtime**. Used in simple in-order superscalar pipelines (like early RISC designs).
  - **Dynamic scheduling**: The **processor reorders instructions at runtime** to exploit more ILP. This is the standard for modern high-performance superscalar cores.

Almost all modern PC superscalar cores (Intel, AMD, ARM) do dynamic scheduling, with out-of-order execution. The most common algorithm for dynamic scheduling is the Tomasulo algorithm, which uses register renaming to eliminate false dependencies.

When the processor implements **Simultaneous Multithreading** (SMT):

- The **instruction window** holds instructions from *multiple hardware threads* simultaneously.
- The **dynamic scheduling logic** treats all the instructions *from all active threads* together.
- At each cycle, it issues the *ready instructions*, from whichever threads have instructions ready, to the available execution units.

This means the same dynamic out-of-order scheduler works across multiple threads.

- *What are the **main hardware modifications** required for a generic superscalar processor to support both ILP and TLP?*

  Answer: To combine ILP and TLP **within each core**, we typically add Simultaneous Multithreading (SMT). So, what we need is:

  - **Multiple architectural register sets**. Each hardware thread needs its own copy of program-visible registers:
    * Every thread needs its own *program counter (PC)*.
    * Every thread needs its own *architectural registers* (e.g. x86 has 16 GPRs, ARM has 31 GPRs, plus FP regs).

    If we didn't keep these separate, two threads would overwrite each other's registers.

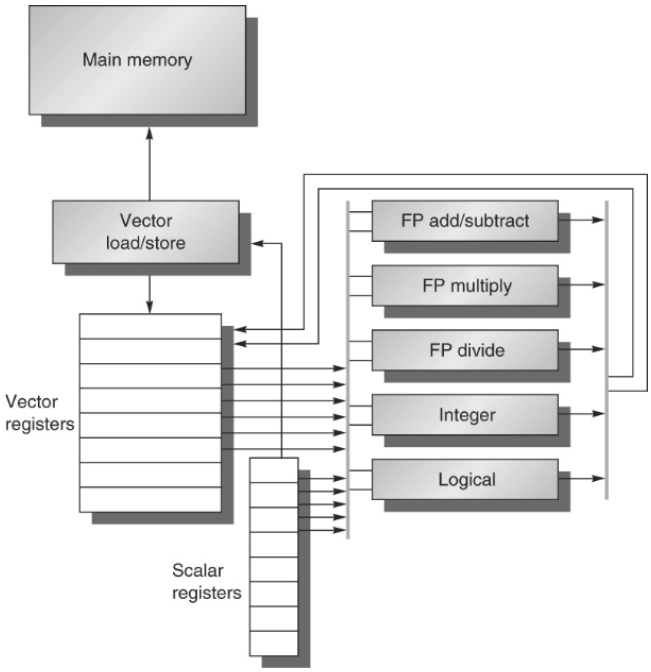
    How is this solved in hardware? The physical register file is big enough to hold *many* physical registers. Each thread has a logical map that says: "*My ISA register* `RAX` *is currently mapped to physical register* `#45`". This is the *register renaming table*. **SMT adds one register map per hardware thread**. This lets the hardware:
    * Track which physical registers belong to which thread's logical registers.
    * Keep architectural state for each thread separate, but reuse the same physical register file.

    This is needed so the processor can fetch, decode, and track multiple *independent* contexts. For example, Intel's SMT (Hyper-Threading) duplicates only the architectural state per thread, but the physical register file is shared and dynamically renamed.
  - **Thread ID tagging in pipeline**. Each instruction must carry a **Thread ID (TID)** through the pipeline. Reservation stations, reorder buffer (ROB), load-store queues, all must track which thread each instruction belongs to. The commit logic uses the TID to retire instructions into the correct thread's architectural state.

- **Thread-aware instruction fetch**. The front-end fetch unit must support fetching from multiple program counters (PCs) in the same cycle or round-robin across active threads. For example, Intel Hyper-Threading allows two threads, front-end selects which one to fetch from each cycle (based on fairness, resource usage, branch miss, stalls).
- **Larger shared resources**. *Instruction window*, *reservation stations*, *ROB*, *rename table*, all must be **larger** to hold more total in-flight instructions. E.g., if a single thread can fill 200 μops in the ROB, with SMT we might need to handle 2× or 4× as many μops in-flight to maintain high throughput.
- **Thread scheduling logic**. The issue logic must pick instructions not just by readiness and dependencies, but also across *multiple threads*. This keeps execution units busy when one thread stalls (e.g., on a cache miss).
- **Shared cache + coherent pipeline queues**. L1/L2 caches must be shared among the SMT threads to allow fine-grained switching. Coherence is easier than in multicore because threads share the same core and same cache hierarchy.
- **Interrupt/exception support per thread**. Exceptions and interrupts must be tracked *per thread*. The pipeline must be able to squash/rollback only the instructions belonging to the faulting thread.

Question 2: Vector Processors

Consider a vector processor architecture, as VMIPS shown in the figure.



- Present the main concepts;
- Present the main advantages of vector execution with respect to scalar execution, detailing the features of the vector architecture that provide the advantage.

Answer:

Deepening: *What is VMIPS?*

Vector MIPS is an educational extension of the classic MIPS RISC architecture, designed to illustrate how a vector processor works. The "V" means it extends the scalar MIPS with vector instructions, vector registers, and vector functional units. The core idea is one vector instruction applies an operation to all elements of a vector register (element-wise) in hardware. For example, let's say we have a vector register `V1` holding 64 floating point numbers, a scalar register `F2` holding the scalar value `10`, then we could write: `VADDVS V2, V1, F2`; that it corresponds to:

```
for i = 0 to V1_length-1:
    V2[i] = V1[i] + F2
```

The hardware streams the elements of `V1` through a pipelined adder. Each cycle, one pair: (`V1[i]`, `scalar`) is read, added, and the result is written to `V2[i]`. After an initial pipeline latency, we get one result per cycle, so throughput is very high.

The figure shows a *vector processor datapath*, this type of architecture is typical of Cray-style supercomputers and is the basis for modern SIMD units.

- **Vector Registers**: Large registers that hold whole *vectors*, each vector register stores multiple elements (e.g. 64 or 128 floats). This enables operations on entire arrays in a single instruction.
- **Scalar Registers**: Hold normal single-word scalar values (e.g., loop bounds, offsets).
- **Vector Load/Store Unit**: Moves entire vectors between *main memory* and the vector registers. Memory is accessed in *strided* or *sequential* patterns.
- **Multiple Vector Functional Units**: Specialized vector ALUs: FP add/subtract unit, FP multiply unit, FP divide unit, Integer unit, Logical unit. These execute vector operations in parallel, multiple elements flow through the pipeline concurrently.
- **Pipelined Execution**: Each functional unit is deeply pipelined, one element per cycle throughput after an initial latency.

A single **vector instruction** specifies an operation on an entire vector register, the hardware pipelines iterate over the elements automatically.
Advantages of vector execution vs. scalar execution:

- ✔ **High throughput via data-level parallelism**
  - Scalar: One instruction operates on one element at a time → many instructions → high instruction bandwidth needed.
  - Vector: One vector instruction does N operations → **reduces instruction fetch, decode, and issue overhead**.
- ✔ **Better memory bandwidth efficiency**. Vector loads/stores transfer *entire contiguous blocks* → hardware can stream efficiently. Strided accesses help with matrices. Memory latency is amortized over large transfers.
- ✔ **Deep pipelining with high utilization**. Pipelines stay full because the same operation repeats over many elements. High utilization of functional units → no pipeline bubbles for loop overhead.
- ✔ **Compact code → fewer branch hazards**. Fewer loop control instructions → fewer branches. Branch misprediction cost is lower. Control flow overhead (loop index increments, tests) is eliminated for the vector part.
- ✔ **Easier to overlap computation and memory**. Because operations are predictable and regular, hardware can prefetch next vectors while current computation runs. Software and hardware together can schedule operations to hide memory latency.

A vector processor like VMIPS is specialized for **data-level parallelism**: a single instruction describes operations on many data elements. This reduces instruction overhead, improves pipeline utilization, maximizes memory throughput, and delivers higher performance than scalar execution for the same silicon area when data-parallel problems are available.

Quizzes

1. **Question 1 (format Multiple Choice - Single answer)**. Let's consider the following code executed by a Vector Processor with:

   - Vector Register File composed of 32 vectors of 8 elements per 64 bits/element;
   - Scalar FP Register File composed of 32 registers of 64 bits;
   - One Load/Store Vector Unit with operation chaining and memory bandwidth 64 bits;
   - One ADD/SUB Vector Unit with operation chaining;
   - One MUL/DIV Vector Unit with operation chaining.

```
1 L.V V1, RA      # Load vector from memory address RA into V1
2 L.V V3, RB      # Load vector from memory address RB into V3
3 MULVS.D V1, V1, F0 # FP multiply vector V1 to scalar F0
4 ADDVV.D V2, V1, V1 # FP add vectors V1 and V1
5 MULVS.D V2, V2, F0 # FP multiply vector V2 to scalar F0
6 ADDVV.D V3, V2, V3 # FP add vectors V2 and V3
7 S.V V1, RX      # Store vector V3 into memory address RX
8 S.V V2, RY      # Store vector V3 into memory address RY
9 S.V V3, RZ      # Store vector V3 into memory address RZ
```

   How many convoys? How many clock cycles to execute the code? ***(SINGLE ANSWER)***

   ✗ **Answer 1**: 2 convoys; 16 clock cycles
   ✗ **Answer 2**: 3 convoys; 24 clock cycles
   ✗ **Answer 3**: 4 convoys; 32 clock cycles
   ✓ **Answer 4**: 5 convoys; 40 clock cycles
   ✗ **Answer 5**: 6 convoys; 48 clock cycles

   A **Convoy** is a **group of vector instructions that can be issued together in the same cycle**, respecting **structural** and **data dependency constraints**, so that the vector processor's functional units are fully utilized without conflicts.

   - A convoy is **like a bundle of operations** that run **in lock-step** on different functional units.
   - The vector processor may have separate pipelines for Load/Store, Vector Add/Subtract, Vector Multiply/Divide, etc.
   - If there is **no resource conflict** (e.g., we don't need two loads at the same time if we only have one load unit) **and no data hazard** (e.g., we don't need the result of an instruction that hasn't finished yet), we can **pack them into the same convoy**.
   - Convoys are **executed sequentially**, but the instructions **within a convoy execute in parallel**, overlapping their pipelines.

Chaining is a hardware feature in **vector processors** that allows **different vector functional units to overlap execution on dependent vector operations**, by **directly forwarding partial results** element by element, **without waiting for the entire vector result to be written back to the vector register file first**.

Vector pipelines are deep, and each vector operation processes one element per cycle once the pipeline is full. Normally, if `Instruction B` depends on `Instruction A`:

- **Without chaining**: `B` must wait until `A` computes **the entire vector result** and writes it back to a vector register.
- **With chaining**: `B` can start **as soon as the first element from `A` is produced**, passing the result directly between functional units for each element.

So we get **element-wise overlap → more parallelism**, **shorter total execution time**, **fewer convoys**.

Motivate your answer by completing the following table:

	Load/Store Vector Unit	Add/Sub Vector Unit	Mul/Div Vector Unit
1^ convoy	`L.V V1, RA`		
2^ convoy	`L.V V3, RB`		`MULVS.D V1, V1, F0`
3^ convoy	`S.V V1, RX`	`ADDVV.D V2, V1, V1`	
4^ convoy			`MULVS.D V2, V2, F0`
5^ convoy	`S.V V2, RY`	`ADDVV.D V3, V2, V3`	
6^ convoy	`S.V V2, RY`		

Without chaining.

	Load/Store Vector Unit	Add/Sub Vector Unit	Mul/Div Vector Unit
1^ convoy	`L.V V1, RA`	`ADDVV.D V2, V1, V1`	`MULVS.D V1, V1, F0`
2^ convoy	`S.V V1, RX`		
3^ convoy	`L.V V3, RB`	`ADDVV.D V3, V2, V3`	`MULVS.D V2, V2, F0`
4^ convoy	`S.V V2, RY`		
[.3em] **6^ convoy** **5^ convoy**	`S.V V3, RZ`		

With chaining (solution).

2. **Question 2 (format True/False)**. In VLIW architectures, the compiler can detect parallelism only in basic blocks of the code.

   Answer: **False**. A basic block is a straight-line piece of code with no branches in except the entry, and no branches out except the exit. In a basic block, the compiler easily sees all data dependencies. But real programs have branches, loops, and procedure calls, so a compiler that only detects ILP inside each basic block finds very limited parallelism.

VLIW compilers do:

- **Basic block scheduling**: VLIW machines rely on **static compile-time scheduling**. They issue wide instructions with many slots (e.g., 4-8 operations in one VLIW bundle). Most basic blocks in normal code are small, often only 5-15 instructions.
- **Global code scheduling**:
  - **Trace scheduling (Fisher)**: Combines multiple basic blocks along the most likely execution paths to form a longer region → reorders instructions globally.
  - **Software pipelining (modulo scheduling)**: Especially in loops, overlaps iterations → exposes more ILP than is visible in a single basic block.
  - **Loop unrolling and reordering**: same idea, bigger region, more ILP to fill the VLIW slots.

3. **Question 3 (format True/False)**. A 4-issue VLIW processor requires 4 Program Counters to load the necessary instructions in the 4 parallel lanes

   Answer: **False**. Because in VLIW, we don't have multiple independent program counters. We have one program counter that points to a very wide instruction word which contains all the parallel operations to be issued together.

4. **Question 4 (format Multiple Choice - Single answer)**. *In the **Speculative Tomasulo Architecture**, what type of hardware block is used to undo speculative instructions in case of a mispredicted branch?*

   ✔ **Answer 1**: Reorder Buffer;

   ✘ **Answer 2**: Instruction Dispatcher;

   ✘ **Answer 3**: Store Buffer;

   ✘ **Answer 4**: Reservation Stations;

   ✘ **Answer 5**: Load Buffers;

   Motivate your answer by explaining what happens in the case of a mispredicted branch.

   Answer: The hardware block is the ROB (Reorder Buffer). During a misprediction, the hardware can simply invalidate all ROB entries younger than the branch because all results are only in the ROB. The renaming table shows instructions made thanks to a prediction because they have a speculative flag set to true. Finally, the PC is reset to the correct target.

References

[1] GeeksforGeeks. Collision resolution techniques [image]. https://www.geeksforgeeks.org/dsa/collision-resolution-techniques/, July 23 2025. Image illustrating collision in hashing.

[2] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. ISSN. Elsevier Science, 2017.

[3] Cristina Silvano. Lesson 1, pipelining. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

[4] Cristina Silvano. Advanced computer architecture. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024-2025.

Index

Symbols

1-bit Branch History Table (1-bit BHT) 54
2-bit Branch History Table (BHT) 56
2-level Predictors 60

A

Address Unit 223
AMAT (Average Memory Access Time) 251
AMAT in Presence of L2 309
Amdahl's Law 95, 245
Anti-Dependence (Write After Read - WAR) 70
ASAP (As Soon As Possible) Scheduling 273
Average Memory Access Time (AMAT) 296, 297, 315

B

Backward Taken Forward Not Taken (BTFNT) 42
Basic Block 267, 270
Branch Always Not Taken 37
Branch Delay Slot 44
Branch History Register (BHR) 66
Branch History Table (BHT) 54
Branch Outcome 28
Branch Outcome Predictor (BOP) 52
Branch Prediction 35
Branch Prediction Buffer 54
Branch Target Address (BTA) 28
Branch Target Buffer (BTB) 39, 52, 58

C

Cache 295
Cache - Block identification 298
Cache - Block placement 298
Cache - Replacement strategy 298
Cache - Write strategy 298
Cache Associativity 320
Cache Block 294, 295, 298
Cache Hit 295, 313
Cache Line 295, 298
Cache Miss 295, 313
Cache Prefetching 332
Cache Replacement Policy: FIFO (First-In First-Out) 303
Cache Replacement Policy: LRU (Least Recently Used) 303
Cache Replacement Policy: Random Replacement 303
Cache Write Miss Policies: Fetch on Write 306
Cache Write Miss Policies: No Write Allocate 306
Cache Write Miss Policies: Write Allocate 306
Cache Write Miss Policies: Write Around 306
Cache Write Policies: Write-Back 305

Cache Write Policies: Write-Through 305
Cache Write-Back 399
Cache Write-Invalidate 398
Cache Write-Through 399
Capacity Misses 313
Chaining 417
Code Efficiency 282
Code Scheduling 185, 269
Coherence Misses 314
Cold Start Misses 313
Collision 313
Column-Associativity 327
Common Data Bus (CDB) 139, 144
Compensation Code 287
Compiler Scheduling 23
Compulsory Misses 313
Conflict Misses 313
Control Dependencies 73
Control Hazards 18, 19, 30
Convoy 416
Correlating Predictors 60
Critical Path 270, 272
Critical Word First 340

D

D-Stream Buffer 330
Data 298
Data Dependence Graph 271
Data Dependencies 69
Data Hazards 18, 19
Data Prefetching 330
Delayed Branch 44
Delayed Branch Scheduling: From After 51
Delayed Branch Scheduling: From Before 46
Delayed Branch Scheduling: From Fall-Through 49
Delayed Branch Scheduling: From Target 47
Dependence Graph 271
Direct-Mapped 399
Direct-Mapped Cache 299
Dirty Bit 305
Dual-Issue Pipeline 84
Dynamic Branch Prediction Techniques 35
Dynamic Loop Unrolling 187
Dynamic Out-of-Order Parallelism 187
Dynamic Renaming - Hardware-side 72
Dynamic Scheduling 81
Dynamic Specualtive Execution 187

E

Early Branch Evaluation 33

Early Restart 340
EX (Execution) 7
EX/EX Path 21
Execution Stage 104
Execution Time 281
Execution Time (Latency, Response Time) 240

F

False Dependencies 70, 136
Floating Point (FP) 144
Floating-Point Unit (FPU) 143
Formula Memory Hierarchy: AMAT 251
Formula Memory Hierarchy: AMAT (L1, L2) 254
Formula Memory Hierarchy: AMAT Harvard 252
Formula Memory Hierarchy: Final Full CPU Time Formula 259
Formula Memory Hierarchy: Full CPU Time 256
Formula Memory Hierarchy: Hit Rate 251
Formula Memory Hierarchy: Memory stall cycles per instruction 258
Formula Memory Hierarchy: Miss Rate 251
Formula Memory Hierarchy: Miss Time 251
Formula Pipeline Loops (AS): IC_{AS} per iter 249
Formula Pipeline Loops (AS): MIPS per iteration 249
Formula Pipeline Loops (AS): Total Clock Cycles per iter 249
Formula Pipeline Loops AS: Cycles Per Instruction per iter 249
Formula Pipeline Loops: Clock Cycles per iter 248
Formula Pipeline Loops: Cycles Per Instruction per iter 248
Formula Pipeline Loops: IC per iter 248
Formula Pipeline Loops: MIPS per iter 248
Formula Pipeline: MIPS 247
Formula Pipeline: Number of clock cycles 247
Formula Speedup: Pipeline Speedup 250
Formula: Clock Cycle Time f_{CLK} 241
Formula: Clock Cycle Time T_{CLK} 241
Formula: Clocks Per Instruction (CPI) 25, 242
Formula: CPU Time 242
Formula: Execution Time 242
Formula: Instruction Count (`IC`) 25
Formula: Instruction Frequency 243
Formula: Instructions Per Cycle (IPC) 242
Formula: MIPS 244
Formula: Performance 241
Formula: Relative Performance 241
Formula: Total Clock Cycles 25
Formula: Weighted Average CPI 243
Formula: Weighted CPU Time 242
Forwarding 20
Fully Associative Cache 300
Function Units (FUs) 144
Functional Unit Status Table 107

G

Global Adaptive (GA) 67
Global History Register (GHR) 60
Global Miss Rate 255, 310
Graduation 235
GShare 67

H

Hazard 18
Hit 251
Hit Time 251, 296, 315
Hit Under Miss 342

I

ID (Instruction Decode) 7
IF (Instruction Fetch) 7
Imprecise Exceptions 79
In-Order Issue 74
Instruction Dependence Graph 271
Instruction Per Clock (`IPC`) 25
Instruction Prefetching 329
Instruction Queue 143
Instruction Status Table 107
instruction window 95
Instruction-Level Parallelism (ILP) 35, 68, 410
Interference Misses 313
Issue Stage 99
Issue Stage (In-Order) 103

L

L2 cache 254
L2 Local Miss Rate 310
List-based scheduling 348
List-Based Scheduling Algorithm 275
Load Buffers 141, 223
Load/Store Queue 144
Local Miss Rate 255
Locality 292
Loop Blocking (Tiling) 335
Loop Fusion 335
Loop Interchange 334
Loop Overhead 281
Loop Unrolling 183, 279

M

ME (Memory Access) 7
MEM/EX Path 22
MEM/MEM Path 22
Memory Accesses per Instruction (MAPI) 256
Memory Disambiguation 196

Merging Arrays 334
MESI 398
Millions of Instructions Per Second (MIPS) 26
Misprediction-Hit 325
Misprediction 35
Miss 251
Miss Penalty 251, 296, 309, 315
Miss Rate 315
Miss Status Holding Registers (MSHRs) 342
Miss Time 297
Miss Under Miss 342
Multi-Cycle In-Order Pipeline 75
Multi-Cycle Out-of-Order (OoO) Pipeline 78
Multiple Outstanding Misses 342
Multiple-Issue Processors 83
Multiple-Issue Width 89

N

N-Way Set-Associative Cache 300
Name Dependencies 70
Non-Blocking Cache 342

O

Output Dependence (Write After Write - WAW) 70

P

Pattern History Table (PHT) 66
Pipeline Registers 17
Pipelining 6
Predicted Target Address (PTA) 52
Principle of Locality 294
Profile-Driven Prediction 43
Pseudo-Associativity 327
Pseudo-Dependencies 70

R

RAW (Read After Write) 19
Read After Write (RAW) 69
Read Operands Stage (Out-of-Order) 104
Read Operands Stage (RR) 99
Ready Bit 224
Ready List 273
Ready Set 275
Register File (RF) 144
Register Prefetching 332
Register Renaming 71, 180
Register Result Status Table 107
Rename Table 226
Reorder Buffer (ROB) 216, 217
Reservation Station (RS) 140

Reservation Stations 138
Reservation Stations (RS) 137
Reservation Stations (RSs) 143
RISC-V Data Path 8
ROB Head pointer 225
ROB number 220
ROB Tail pointer 225

S

Scoreboard 97
Second-Level Cache (L2) 309
Simultaneous Multithreading (SMT) 411
Software Pipelining 283, 344
Spatial Locality 292, 294
Specualtive Execution 286
Speculative Flag 224
Static Branch Prediction Techniques 35
Static Renaming 136
Static Renaming - Software-side 72
Store Buffers 141
Strongly Not Taken (SNT) 56
Strongly Taken (ST) 56
Structural Hazards 18, 19
Superblock Scheduling 288
Superscalar Processor 89

T

Tag 298
Tail Duplication 289
Temporal Locality 292, 294
Temporary Names (tags) 137
Thread-Level Parallelism (TLP) 410
Throughput (Bandwidth) 240
Tomasulo's Algorithm 134
Trace Compaction (Scheduling) 286
Trace Scheduling 286
Trace Selection 286
True Data Dependencies 69
Two-Level Adaptive Branch Predictors 66

V

Valid Bit 298
Victim Block Swap 322
Victim Cache 322
VLIW (Very Long Instruction Word) 92, 260
VLIW (Very Long Instruction Word) Processor 89

W

WAR (Write After Read) 19
WAW (Write After Write) 19

Way Prediction 324
Way Predictor Table (WPT) 324
WB (Write Back) 7
Weakly Not Taken (WNT) 56
Weakly Taken (WT) 56
Wrapped Fetch 340
Write Result Stage (Out-of-Order) 104
Write-Allocate 399