Advanced Computer Architectures - Notes - v0.9.0

260236

May 2025

Preface

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

- Computer Architecture: A Quantitative Approach. [1]

- Pipelining slides. [2]

- Course slides. [3]

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

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 <mark>divide the execution of an instruction into distinct stages and process different instructions simultaneously in these stages</mark>. 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**:

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

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

    $$rd \leftarrow rs1 + rs2$$

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

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

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

    $$rd \leftarrow rs1 + 4$$

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

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

    Load data into register `rd` from an address formed by adding `rs1` to a signed `offset`.
    $$rd \leftarrow M\left[rs1 + offset\right]$$

    - **Stores** data in memory:

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

    Store data from register `rs2` to an address formed by adding *rs1* to a signed *offset*.
    $$M\left[rs1 + offset\right] \leftarrow rs2$$

- **Branching Instructions (Control Flow Management)**

  - **Conditional Branches**

    * Branch on equal:

    ```
    1 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:

    ```
    1 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):

    ```
    1 j L1
    ```

    Jump directly to the L1 label.

    * Jump to the address stored in a register (jump register):

    ```
    1 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 $5\times$ 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 $5\times$ (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 × 2 ns each).

- In a pipelined system, once the pipeline is full, an instruction completes every cycle (every 2 ns), achieving a 5× 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.

15

- **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 15). 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 27) 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 67) 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

```
1 sub x2, x1, x3   # Compute x2 = x1 - x3
2 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

```
1  sub x2, x1, x3
2  # Independent instruction
3  # (can execute while sub is completing)
4  add x4, x10, x11
5  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 19.

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**:

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

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 $= 500$MHz

Metrics:

- Total Clock Cycles:

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

- CPI Calculation:

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

- MIPS Calculation:

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

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**:

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

- `MIPS` **per Iteration**:

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

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

- `CPI` **per Iteration**:

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

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

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

For **large programs**, startup stalls become negligible, and **performance depends mainly on stall cycles per iteration**. <mark>Minimizing</mark> k (<mark>stalls per iteration</mark>) 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 <mark>decisions about the next instruction to execute are often dependent on certain conditions</mark>. 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**.

❷ 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.

❷ 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**.

beq x1, x3, L1	IF	ID	EX	ME	WB			
and x12, x2, x5	IF stall		IF	ID	EX	ME	WB	
or x13, x6, x2				IF	ID	EX	ME	WB
add x14, x2, x2					IF	ID	EX	ME	WB

🔄 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 32). 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:

```
1  z = x + y;
2  if (z > 0)
3      w = x;
4  else
5      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 36): 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 <mark>non-trivial</mark> 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). <mark>To avoid delays</mark>, 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 <u>not</u> 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 <u>not</u> 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 <mark>applications with stable branch behavior</mark>.

- ✔ **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 45 - From **before** the branch.

2. Section 2.5.5.2, page 46 - From the **target** of the branch.

3. Section 2.5.5.3, page 48 - From the **fall-through path** (i.e., the sequential next instruction).

4. Section 2.5.5.4, page 50 - 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:

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

After scheduling:

```
beq x1, x2, x3
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.

---

❷ 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:

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

After scheduling:

```
1 add x1, x2, x3       # if branch taken, go here!
2 if x1 == 0 then      # branch condition
3 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.

---

❷ 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:

```
1  Branch A → L1
2  Branch B → L1
3
4  L1:
5      Instruction X
6      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**.

---

❷ 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:

```
1  Instr A       # Before branch
2  Branch        # Branch condition
3  Instr B       # PC + 4 (fall-through)
4  Instr C       # After fall-through ← from after
5  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**: <mark>if a branch behaved a certain way in the past, it's likely to behave the same way again</mark>. 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 <mark>simplest version uses 1 bit per branch</mark> 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 $\Rightarrow$ predict Taken.

- If last outcome was Not Taken $\Rightarrow$ 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** $\Rightarrow$ keep bit unchanged
        * **Incorrect prediction** $\Rightarrow$ 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:

$$\text{T T T T T T T T T NT}$$

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:

```
1  if (A)          // Branch 1
2      ...
3  if (B)          // Branch 2
4      ...
5  if (C)          // Branch 3
6      ...
```

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` $\rightarrow$ 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 <mark>constraints</mark> 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.

❷ 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 <mark>actual requirement for program correctness</mark>.

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**
>
> ```
> 1 I1:    r3 ← r1 + r2    # writes to r3
> 2 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 <mark>eliminate these artificial constraints</mark> 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):
>
> ```
> 1 I1:    r3 ← r1 + r2
> 2 I2:    r1 ← r4 + r5
> ```
>
> Renamed:
>
> ```
> 1 I1:    r3 ← r1 + r2
> 2 I2:    r9 ← r4 + r5     # write to r9 instead of r1
> ```
>
> Now, there is no conflict, I2 can proceed independently of I1.
> Original code (WAW):
>
> ```
> 1 I1:    r3 ← r1 + r2
> 2 I2:    r3 ← r4 + r5
> ```
>
> Renamed:
>
> ```
> 1 I1:    r3 ← r1 + r2
> 2 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

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

In assembly:

```
1 I1:  bgtz  r1, LABEL    # branch if r1 > 0
2 I2:  ...                # instruction before LABEL
3 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,

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 <mark>strict discipline</mark>:

- 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.

❷ The Need for Dynamic Scheduling

Consider the following instruction sequence:

```
1 I1:  F0  ← F2 / F4   # long-latency divide
2 I2:  F10 ← F0 + F8   # depends on F0
3 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 <mark>issue only one instruction per clock cycle</mark>. 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:

$$\text{IF} \to \text{ID} \to \text{EX} \to \text{MEM} \to \text{WB}$$

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 74).

- 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 77).

- 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 82).

- 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?

❷ 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, **misspeculation 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 73).
  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 writeback (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 70) 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. [1]

- 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 18), 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 18) 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 <u>three types</u> of data hazards <u>dynamically</u>**:

✅ **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 in-
   struction 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 <mark>vulnerable to WAR and WAW hazards</mark>, 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:

```
1  LD    F6, 34(R2)
2  LD    F2, 45(R3)
3  MULTD F0, F2, F4      # RAW on F2
4  SUBD  F8, F6, F2      # RAW on F6, F2
5  DIVD  F10, F0, F6     # RAW on F0, F6
6  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 struc-
tures introduced in the section 3.6.6, page 106:

- **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.

121

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

- Intel Pentium II

- HP PA-8000

- PowerPC 604

- MIPS R10000

⚖ 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

❷ **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 69) 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:

```
1 DIV.D  F0, F2, F4
2 ADD.D  F6, F0, F8        ; RAW on F0
3 S.D    F6, 0(R1)         ; RAW on F6
4 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:

```
1 DIV.D  F0, F2, F4
2 ADD.D  S, F0, F8         ; RAW still valid
3 S.D    S, 0(R1)          ; Now reads S
4 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 ≠ 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. [1] 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 $\rightarrow$ use `Vi`.
  - If `Qi` $\neq$ `0`, value is pending $\rightarrow$ 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.

�över 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 142), where they wait until operands and functional units are ready.

❷ 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**.

❷ 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:

```
1  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:

```
1  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`)**:

$$\text{CDB} \leftarrow \langle \text{tag} = \text{RS3}, \text{value} = \text{result} \rangle$$

   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:
      $$\text{Qj} \leftarrow 0 \quad \lor \quad \text{Qk} \leftarrow 0$$
      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:
>
> $$\langle \texttt{Tag} = \texttt{RS1}, \texttt{Value} = \dots \rangle$$
>
> 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` $\Rightarrow$ waits on `LB1`'s result

3. Instruction 6: `F6 ← ADD.D` $\Rightarrow$ 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 $\langle$`Tag = MUL1`, `Value = (result)`$\rangle$

2. The **Register File** updates:
   - If `Qi = MUL1` $\rightarrow$ write result into the register.
   - If `Qi ≠ MUL1` $\rightarrow$ 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:

```
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
```

❓ 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` → F6 = F2 + F4

- `SUBD F6, F2, F4` → 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:

$$A \div B = A \times \left(\frac{1}{B} \right)$$

Instead of implementing a full-blown divider, many systems compute the **reciprocal of** B $\left(\frac{1}{B}\right)$ and then multiply:

```
1 DIV.D    F10, F0, F6     ; F10 ← F0 ÷ F6
2 ≈
3 RECIP    F12, F6         ; F12 ←  1 ÷ F6
4 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)	Load1⁰		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		Name	Vj	Qj	Vk	Qk
`ADD1`	`v(F6)`			`Load2`		`MUL1`		`Load2`	`v(F4)`	
`ADD2`						`MUL2`		`MUL1`	`v(F6)`	
`ExADD`						`ExMUL`				

ExADD.   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~~ $\nearrow$ 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		<mark>v(F8)</mark>		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.

171

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	<mark>59</mark>
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)		<mark>v(F10)</mark>		

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 70) we gave a very short presentation dedicated to understand how to solve name dependencies, and in section 3.7.2 (page 135) 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 **bottle-necks** 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.

❷ 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 136).

❷ Why Renaming?

Consider the following floating-point loop:

```
Loop:   LD    F0, 0(R1)        ; Load word into F0 from address 0+R1
        MULTD F4, F0, F2       ; Multiply F0 and F2, result into F4
        SD    F4, 0(R1)        ; Store word from F4 to address 0+R1
        SUBI  R1, R1, #8       ; Decrement R1 by 8
        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 minimize and 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:

```
1  LD     F0, 0(R1)
2  MULTD  F4, F0, F2
3  SD     F4, 0(R1)
4  LD     F6, -8(R1)
5  MULTD  F8, F6, F2
6  SD     F8, -8(R1)
7  LD     F10, -16(R1)
8  MULTD  F12, F10, F2
9  SD     F12, -16(R1)
10 LD     F14, -24(R1)
11 MULTD  F16, F14, F2
12 SD     F16, -24(R1)
13 SUBI   R1, R1, #32
14 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:

```
1  LD     F0, 0(R1)
2  LD     F6, -8(R1)
3  LD     F10, -16(R1)
4  LD     F14, -24(R1)
5  MULTD  F4, F0, F2
6  MULTD  F8, F6, F2
7  MULTD  F12, F10, F2
8  MULTD  F16, F14, F2
9  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 $\rightarrow$ `Vj = R(F2)`
     - `F0` is *not* ready, **waiting on** `Load1` to produce it $\rightarrow$ `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		<mark>Mult1</mark>						

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.

195

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]`).
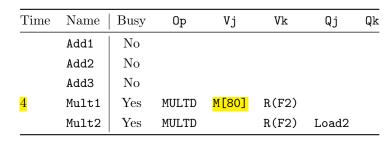
    - The second load (`Load2`) completes its execution in 1 cycle thanks to the cache hit. In the next cycle, the result is written back. This instruction was blocked by the previous load to avoid too aggressive memory disambiguation.

    - 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.

❷ 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 <mark>break</mark> 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**: <mark>allocate an entry</mark> in the ROB, marking the instruction's place in program order.

- During **execution**: <mark>store the result</mark> in the ROB as soon as the instruction finishes.

- During **commit**: <mark>only commit</mark> the instruction (i.e., update architectural registers or memory) <mark>if</mark>:

    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:

```
1 Instr 1 → OK
2 Instr 2 → OK
3 Instr 3 → Exception (e.g., divide by zero)
4 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
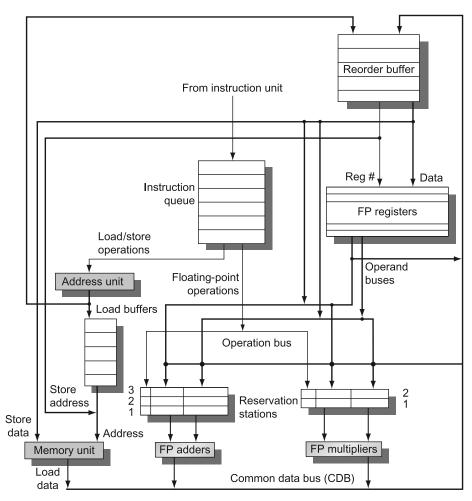
Figure 27: The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation (ROB). [1]

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 187.

  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**.

❷ 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**, <u>before</u> 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.**

<mark>Instructions don't wait statically, they dynamically listen and proceed as soon as they can, maximizing execution overlap.</mark>

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.**

<mark>Results are not immediately written to registers; they are buffered safely into the ROB and forwarded to dependent instructions dynamically.</mark>

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* [1], 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`).

237

- – 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 260, 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:

$$\text{Execution Time}(B) = \left(1 + \frac{n}{100}\right) \times \text{Execution Time}(A)$$

$$15 = \left(1 + \frac{50\%}{100}\right) \times 10$$

$$15 = \frac{3}{2} \times 10$$

$$15 = \frac{30}{2} \checkmark$$

🕐 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**:

$$\text{CPU Time} = \text{Clock Cycles} \times T_{\text{CLK}} = \frac{\text{Clock Cycles}}{f_{\text{CLK}}} \qquad (14)$$

- **Expanded form based on instructions**:

$$
\begin{aligned}
\text{CPU Time} \;&=\; \text{Instruction Count} \times \text{Cycles Per Instruction} \times T_{CLK} \\[2mm]
&=\; \frac{\text{Instruction Count} \times \text{Cycles Per Instruction}}{f_{\text{CLK}}}
\end{aligned}
$$

$$(15)$$

- **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:

$$\text{CPU Time} = \left(\sum_{i=1}^{n} (\text{CPI}_i \times \text{I}_i) \right) \times T_{CLK} \qquad (16)$$

  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 242):

$$\text{CPU Time} = \frac{\text{IC} \times \text{CPI Weighted}}{f_{\text{CLK}}}$$

🏎 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.

$$\text{CPI} = \frac{\text{Total Clock Cycles}}{\text{Instruction Count (IC)}} \qquad (17)$$

  Where the Total CLock Cycles is:

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

  And the Instruction Count is the total number of instructions executed.

- **Instructions Per Cycle (IPC)** is the reciprocal of CPI:

$$\text{IPC} = \frac{1}{\text{CPI}} \qquad (18)$$

- **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:

$$\text{CPI} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \text{F}_i \right) \tag{19}$$

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:

- $43\% \to 0.43$

- $21\% \to 0.21$

- $12\% \to 0.12$

- $12\% \to 0.12$

- $12\% \to 0.12$

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**:

$$\text{ExTime(w/ E)} = \left((1 - F) + \frac{F}{S} \right) \times \text{ExTime(w/o E)} \qquad (23)$$

  Where:

  – "w/ E" = "*with Enhancement*"
  – "w/o E" = "*without Enhancement*"

- **Overall Speedup**:

$$\text{Speedup(E)} = \frac{1}{(1 - F) + \frac{F}{S}} \qquad (24)$$

  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:

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{FractionE}) + \dfrac{\text{FractionE}}{\text{SpeedupE}}} \qquad (25)$$

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:

$$
\begin{aligned}
\text{Speedup}_{\text{overall}} &= \frac{1}{(1 - 0.4) + \dfrac{0.4}{10}} \\
&= \frac{1}{0.6 + 0.04} \\
&= \frac{1}{0.64} \\
&= 1.56
\end{aligned}
$$

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:

$$T_{\mathrm{CLK}} > \text{time of the slowest stage}$$

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 $\frac{1}{\mathrm{CPI}}$

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:

$$\mathrm{CPI} = \frac{\text{Clock Cycles}}{\text{IC}} = \frac{(\text{IC} + \text{Stall Cycles} + 4)}{\text{IC}} \quad (28)$$

- **MIPS calculation**:

$$\mathrm{MIPS} = \frac{f_{\mathrm{clock}}}{\mathrm{CPI} \times 10^6} \quad (29)$$

> **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:

$$\text{Pipeline Speedup} = \frac{\text{Avg Exec Time Unpipelined}}{\text{Avg Exec Time Pipelined}} = \frac{\text{CPI}_{\text{unp}} \times T_{\text{clk, unp}}}{\text{CPI}_{\text{pipe}} \times T_{\text{clk, pipe}}} \quad (40)$$

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:

$$\text{Pipeline Speedup} = \frac{\text{CPI}_{\text{unp}}}{1 + \text{CPI}_{\text{pipe}}} = \frac{\text{CPI}_{\text{unp}}}{1 + \text{Stall Cycles per Instruction}} \quad (41)$$

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):

$$\text{Pipeline Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Stall Cycles per Instruction}} \quad (42)$$

If there were **no pipeline stalls (ideal case)**, this leads to the intuitive result that pipelining improves performance by the depth of the pipeline:

$$\text{Speedup} = \text{Pipeline Depth} \quad (43)$$

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:

$$\text{Pipeline Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Branch Frequency} \times \text{Branch Penalty}} \quad (44)$$

- **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:

$$\text{PSCIB} = \text{Branch Frequency} \times \text{Branch Penalty} \quad (45)$$

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.

$$\boxed{\text{Processor}}$$

$$\boxed{\text{L1 cache}}$$

$$\boxed{\text{L2 cache}}$$

$$\boxed{\text{Main Memory}}$$

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.

253

⚖ 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:

$$\begin{aligned} \text{Misses per Instruction} = \ & \frac{\#\text{ misses}}{\text{IC}} \times \frac{\#\text{ mem. accesses}}{\text{IC}} \times \text{Miss Rate} \\[2mm] = \ & \text{MAPI} \times \text{Miss Rate} \end{aligned} \tag{57}$$

  (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:

$$\begin{aligned} \text{CPU Time} = \quad & \text{IC} \times (\text{CPI}_{\text{exec}} + \text{Stalls per Instruction} + \\ & \text{MAPI} \times \text{Miss Rate} \times \text{Miss Penalty}) \times \\ & T_{CLK} \end{aligned} \qquad (60)$$

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 $\rightarrow$ time to access L2.

- Stall cycles after an L2 miss $\rightarrow$ time to access main memory.

Average **memory stall cycles per instruction** (considering both L1 and L2) is:

$$\text{Memory stall cycles per instruction} = \begin{aligned} &(\text{Misses}_{L1} \cdot \text{Hit Time}_{L2}) + \\ &(\text{Misses}_{L2} \cdot \text{Miss Penalty}_{L2}) \end{aligned} \tag{61}$$

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.

References

[1] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. ISSN. Elsevier Science, 2017.

[2] Cristina Silvano. Lesson 1, pipelining. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

[3] Cristina Silvano. Advanced computer architecture. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024-2025.

Index

E

F

V

W