Advanced Computer Architectures - Notes - v0.4.0

260236

April 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.

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 divide the execution of an instruction into distinct stages and process different instructions simultaneously in these stages. This approach significantly increases the efficiency of instruction execution in modern processors.

Understanding the RISC-V instruction set

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

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

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

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

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

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

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

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

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

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

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

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

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

  - **Stores** data in memory:

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

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

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

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

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

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

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

    - Branch on not equal:

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

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

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

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

      ```
      j L1
      ```

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

      ```
      jr ra
      ```

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

These basic instructions will be used throughout the course.

Execution phases in RISC-V

1. **IF (Instruction Fetch)**: The instruction is **fetched** from memory.

2. **ID (Instruction Decode)**: The instruction is **decoded**, and the **required registers are read**.

3. **EX (Execution)**: The instruction is **executed**, typically involving ALU operations.

4. **ME (Memory Access)**: For *load*/*store* instructions, this stage **reads from** or **writes to** memory.

5. **WB (Write Back)**: The **result is written back** to the destination register.

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

Implementation of the RISC-V Data Path

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

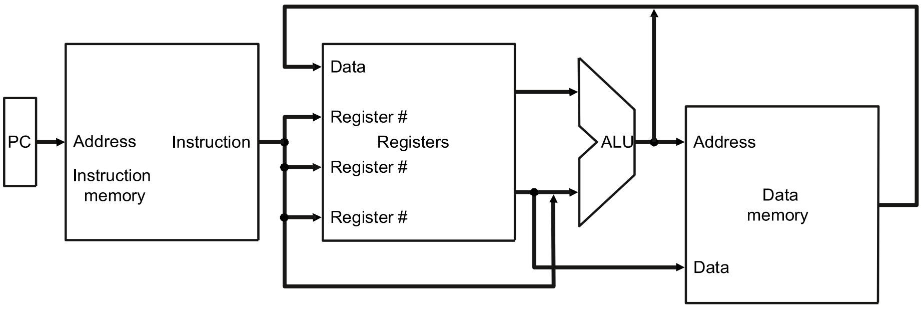


Figure 1: Generic implementation of the RISC-V Data Path.

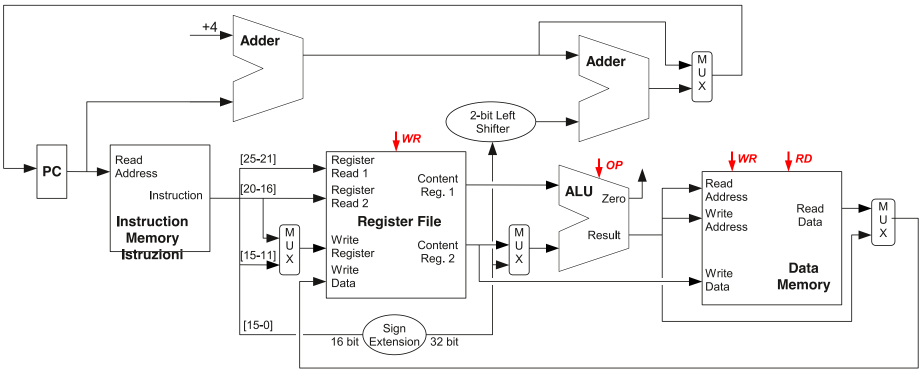


Figure 2: Specific implementation of the RISC-V Data Path.

Its fundamental components include:

- **Instruction Memory and Data Memory Separation**. RISC-V adopts a Harvard Architecture style, where the **Instruction Memory (IM)** and **Data Memory (DM) are separate**. This **prevents structural hazards** where instruction fetch and memory access could conflict in a single-memory design (this topic will be addressed later).

- **General-Purpose Register File (RF)**. It consists of **32 registers**, each **32-bit** wide. The register file has **two read ports** and **one write port** to support simultaneous read and write operations. This setup allows faster register access, which is crucial for pipelined execution.

- **Program Counter (PC)**. It **holds the address of the next instruction to be fetched**. Automatically increments during execution, typically by 4 bytes (for 32-bit instructions).

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

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

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

Example 1: Data Path Execution Example

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

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

1.2 RISC-V Pipelining

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

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

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

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

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

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

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

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

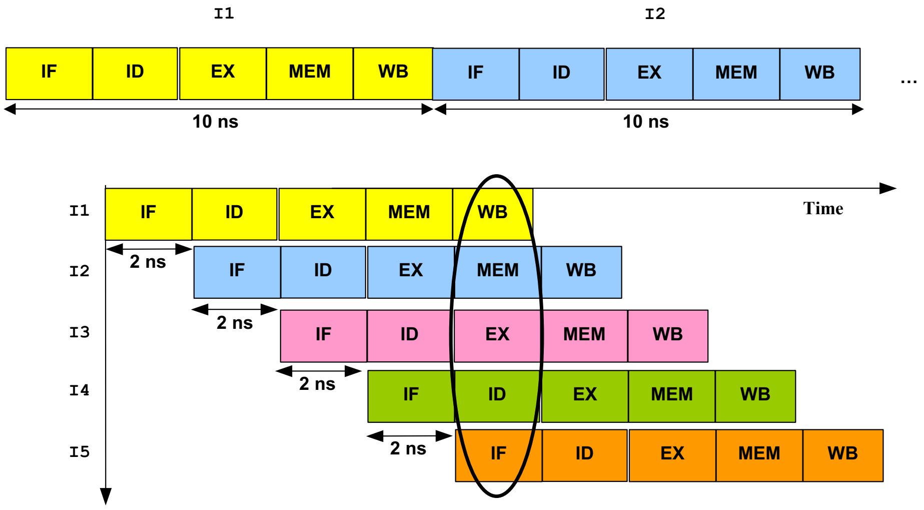


Figure 3: Sequential vs. Pipelining execution.

Pipeline Performance and Speedup

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

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

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

Understanding Pipelining Performance

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

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

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

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

Table 1: Pipelining timeline execution in an ideal case.

- The first instruction `I1` takes 5 (clock) cycles to complete, i.e., 10 ns.
- However, starting from cycle 5, a new instruction finishes every cycle (every 2 ns).
- In a non-pipelined system, each instruction would take 10 ns (5 stages × 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**.

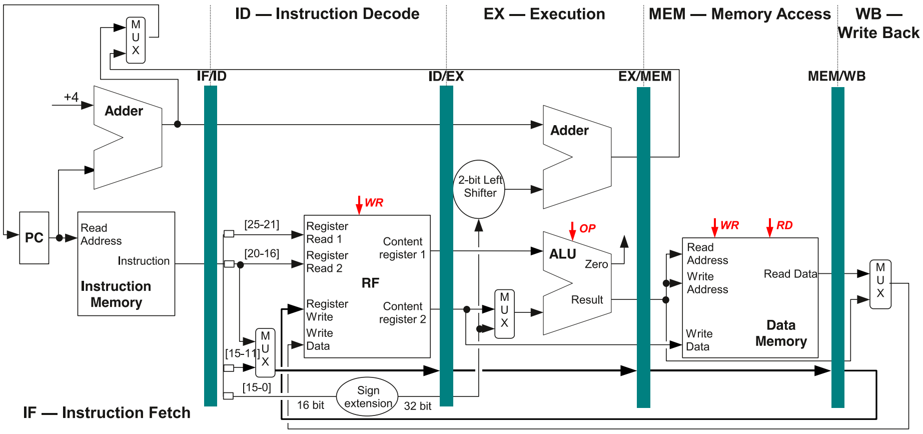


Figure 4: Structure of RISC-V pipeline.

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.

- **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:

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

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

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

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

1.3.1 RISC-V Optimized Pipeline

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

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

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

Optimization: Read and Write in the Same Cycle

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

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

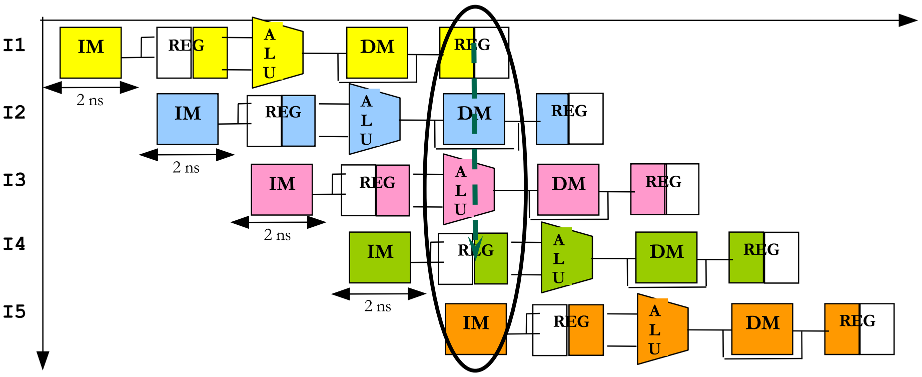


Figure 5: Visual **example** of an optimized pipeline; here the result (WB stage) of `I1` is written in the first half of the clock cycle and the read (ID stage) of `I4` is done in the second half of the clock cycle. So there is no hazards!

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

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

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

Example 2: EX/EX Forwarding

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

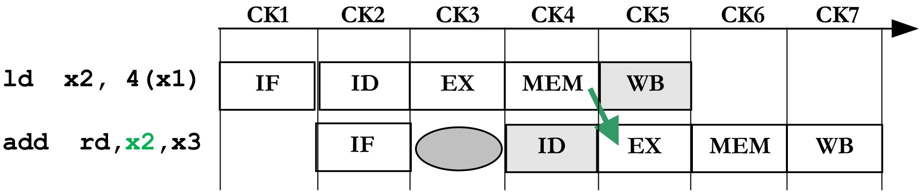
Cycle	`sub x2, x1, x3`	`and x12, x2, x5`
1	IF	
2	ID	IF
3	EX	ID
4	MEM	*Stall*
5	WB	*Stall*
6		EX
7		MEM
8		WB

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

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

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

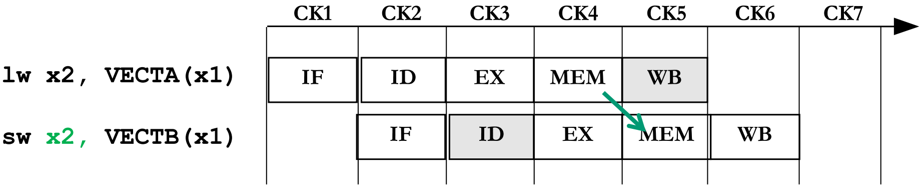
- **MEM/EX Path**. Forwards the ALU result from **MEM stage to EX stage**. Used when an **instruction depends on an ALU operation two cycles before**.



	CK1	CK2	CK3	CK4	CK5	CK6	CK7
ld x2, 4(x1)	IF	ID	EX	MEM	WB		
add rd,x2,x3		IF		ID	EX	MEM	WB

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**.



	CK1	CK2	CK3	CK4	CK5	CK6	CK7
lw x2, VECTA(x1)	IF	ID	EX	MEM	WB		
sw x2, VECTB(x1)		IF	ID	EX	MEM	WB	

Figure 7: Example of MEM/MEM path.

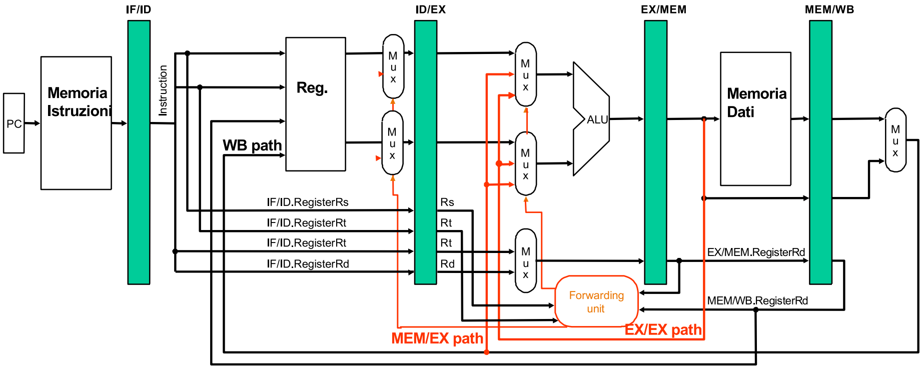


Figure 8: Implementation of RISC-V with Forwarding Unit.

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 `nop`s (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 `nop`s:

✗ **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

```
sub x2, x1, x3
nop                # Delay slot (bubble)
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 `nop`s.

Key takeaway of instruction scheduling:

- Instruction reordering is a **compiler optimization**.

✓ It works well **if independent instructions are available**.

✗ In some cases, no independent instructions exist, so **stalling or forwarding is needed**.

Example 4: Instruction Scheduling

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

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

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

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

Key takeaway of pipeline stalling:

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

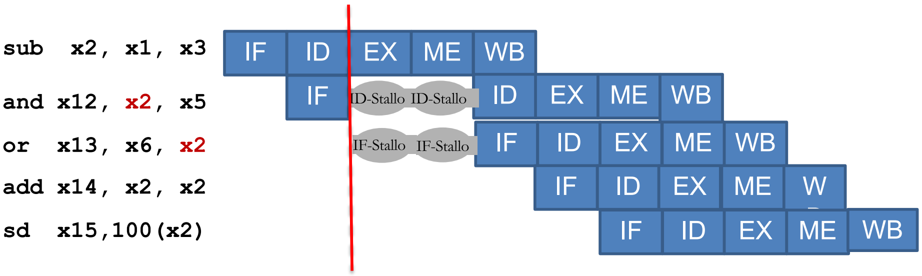


Figure 9: Example of inserting stalls.

✓ *Dynamic (hardware-based)*: Forwarding (Bypassing)

Forwarding is an optimized hardware technique that avoids pipeline stalls by **directly passing results between pipeline registers**. The entire implementation has already been explained on page 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** ($\texttt{CPI} = 1$), 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`.

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

  Where the total clock cycles is:

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

  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`:

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

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

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

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

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

Example 6: Performance Calculation

Given:

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

Metrics:

- Total Clock Cycles:

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

- `CPI` Calculation:

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

- `MIPS` Calculation:

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

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

Performance in Loops and Asymptotic Analysis

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

For a loop with:

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

We have:

- **Clock Cycles per Iteration**:

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

- **`CPI` per Iteration**:

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

- **`MIPS` per Iteration**:

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

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

- **`CPI` per Iteration**:

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

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

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

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

Why `CPI` is Greater than 1 in Real Pipelines

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

2 Control Hazards and Branch Prediction

2.1 Conditional Branch Instructions

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

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

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

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

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

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

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

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

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

Execution of Branches in Pipelined Architectures

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

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

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

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

⚠ Implication for IF

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

2.2 Control Hazards

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

What really causes Control Hazards?

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

This leads to uncertainty:

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

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

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

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

Key Takeaways: Control Hazards

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

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

2.3 Naïve Solutions to Control Hazards

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

How does the conservative solution work?

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

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

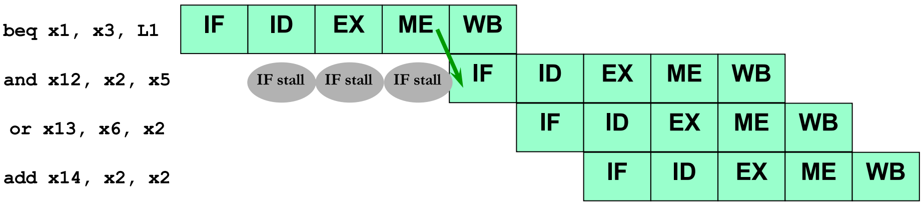


Figure 10: Example of stalls inserted in the pipeline to read the correct value after a branch condition.

Performance Impact

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

Can optimized evaluation at the ID stage improve performance?

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

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

How Early Branch Evaluation Works

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

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

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

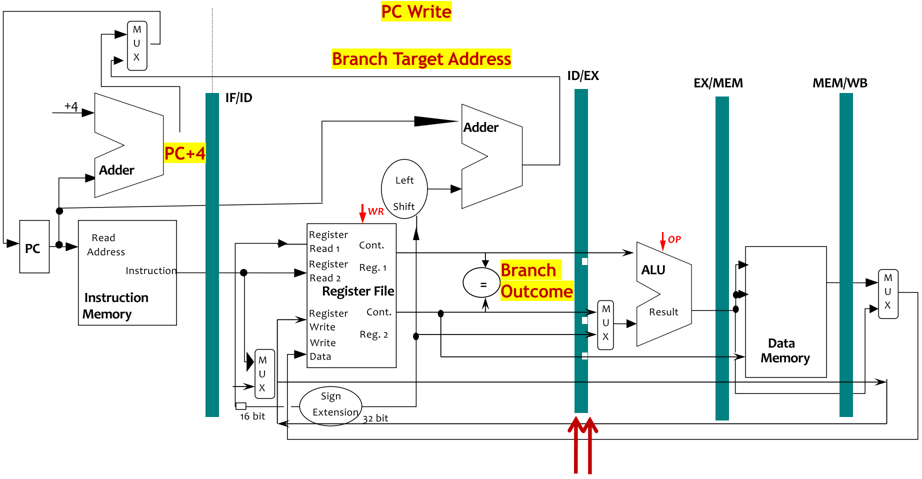


Figure 11: Hardware features modifications to allow early branch evaluation.

Hardware Overhead: Complexity vs. Performance

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

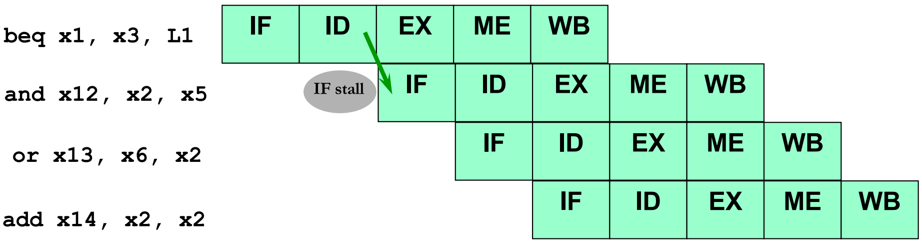
- **ALU logic** for comparison and addition.
- Additional **multiplexer and control signals** to direct the PC update.
- Expanded **data paths** for handling the offset and register values.

Effect on Pipeline Execution

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

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

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



Conclusion

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

2.4 Intro to Branch Prediction

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

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

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

Branch Prediction categories

Branch prediction techniques are generally classified into two main categories:

- **Static Branch Prediction Techniques**. In this method, the **branch direction** (taken/untaken) is **decided at compile time** and **remains fixed during the program's execution**. Static prediction often relies on compiler heuristics or profiling data to guess likely outcomes. Since the behavior doesn't adapt to runtime changes, this **technique works best** when **branch outcomes** are **highly predictable** and **consistent** across executions.

- **Dynamic Branch Prediction Techniques**. Unlike static methods, dynamic prediction uses **hardware mechanisms to observe past branch behavior at runtime** and make predictions accordingly. The **prediction adapts to actual program execution**, making it more effective for applications with **complex or data-dependent control flow**. This method can dynamically switch its guess depending on the *branch history*.

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

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

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

2.5 Static Branch Prediction

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

✓ When does static branch prediction work well?

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

⚠ RISC-V assumption

A key architectural note here is the assumption that we are working with a **RISC-V processor**, which is **optimized for early branch evaluation during the Instruction Decode (ID) stage** (see more in section 2.3, page 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:

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

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

Implementation Details

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

Misprediction Case

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

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

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

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

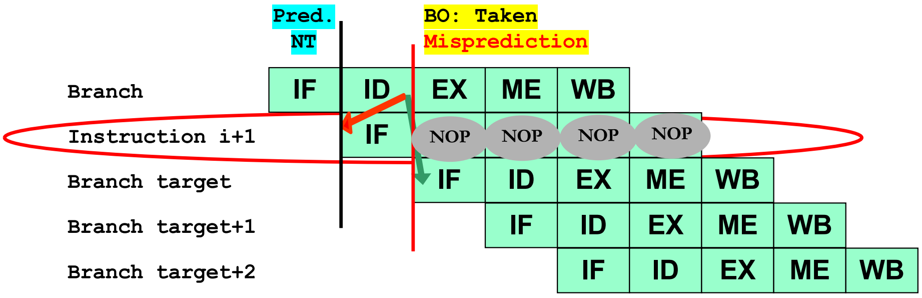


Figure 12: Branch Always Not Taken techniques failed and the processor must flush instruction `i+1` and restart execution from the BTA.

2.5.2 Branch Always Taken

This approach represents the dual case of the previous technique (page 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 non-trivial because:

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

? What is BTB and why do we need it?

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

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

How does BTB work? Quickest explanation

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

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

Example 1: BTB and Branch Always Taken technique

Let's say:

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

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

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

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

✓ Correct Prediction Path

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

✗ Misprediction Case

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

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

When is this technique effective?

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

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

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

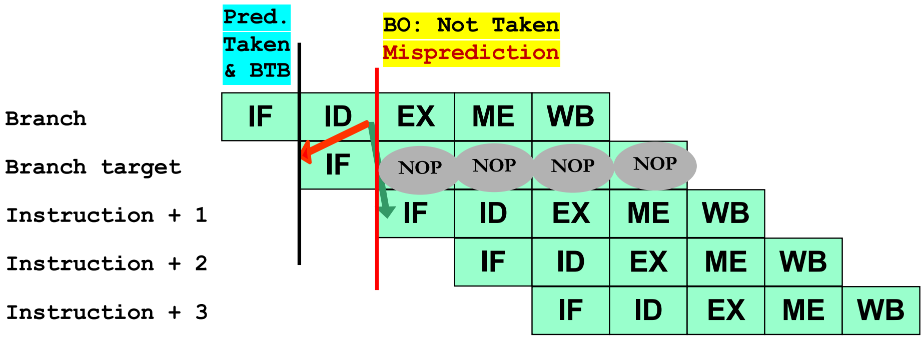


Figure 13: Branch Always Taken techniques failed and the processor must flush instruction `i+1` and restart execution from the BTA.

2.5.3 Backward Taken Forward Not Taken (BTFNT)

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

Prediction Rule

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

Why does this work?

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

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

Pros and Cons

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

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

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

Advantages

✓ Offers **higher accuracy than heuristics alone**, especially for applications with stable branch behavior.

✓ **No runtime hardware cost**, since prediction decisions are guided by **static hints**.

Limitations

✗ **Static nature**: Predictions don't adapt to runtime variability; changes in data patterns may invalidate the profile.

✗ Requires **extra compilation effort**: profiling and hint encoding add complexity to the build process.

✗ **Less effective** for programs with **input-dependent control flow**.

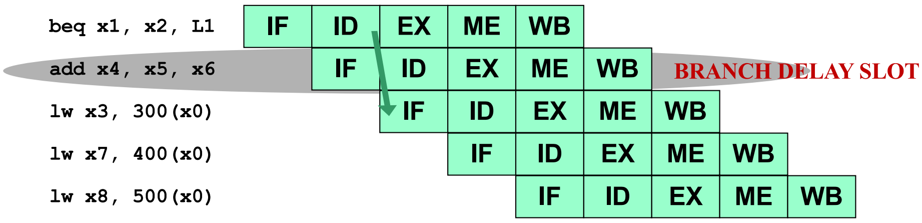
2.5.5 Delayed Branch

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

Core Concept

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

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



`beq x1, x2, L1`	IF	ID	EX	ME	WB				
`add x4, x5, x6`		IF	ID	EX	ME	WB			
`lw x3, 300(x0)`			IF	ID	EX	ME	WB		
`lw x7, 400(x0)`				IF	ID	EX	ME	WB	
`lw x8, 500(x0)`					IF	ID	EX	ME	WB

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.

Pipeline Behavior

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

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

Advantages

✓ **No need for instruction flushing**: the delay slot instruction is always valid.

✓ **Efficiency**: reuses existing instructions from earlier in the program to **hide the branch delay**.

2.5.5.2 From Target

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

Key Characteristics

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

Example 3: From Target

Original code:

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

After scheduling:

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

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

Pipeline Behavior

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

Instruction Duplication

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

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

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

Here's what happens:

```
Branch A → L1
Branch B → L1

L1:
    Instruction X
    Instruction Y
```

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

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

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

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

Best Use Case

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

2.5.5.3 From Fall-Through

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

Key Characteristics

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

Example 4: From Fall-Through

Original code:

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

After scheduling:

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

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

Pipeline Behavior

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

✓ When is this strategy used?

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

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

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

2.5.5.4 From After

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

Let's number the instructions to make it easy:

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

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

⚠ Why is this hard?

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

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

2.6 Dynamic Branch Prediction

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

Core Idea

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

Hardware Components

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

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

How it works

During instruction fetch:

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

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

? Execution Scenarios

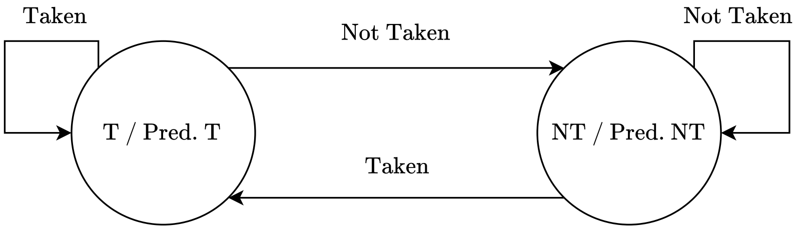
- **Prediction: Not Taken (`PC + 4`)**
  - If Branch Outcome = Not Taken ⇒ ✓ Correct prediction ⇒ No penalty.
  - If Branch Outcome = Taken ⇒ ✗ Misprediction:
    1. Flush next instruction (`NOP`)
    2. Fetch from BTA (to understand where to jump)
    3. One-cycle penalty
- **Prediction: Taken (BTB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

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

2.6.1 1-bit Branch History Table

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

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



How it works

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

Indexing the Table

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

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

- 2^k **entries** total.

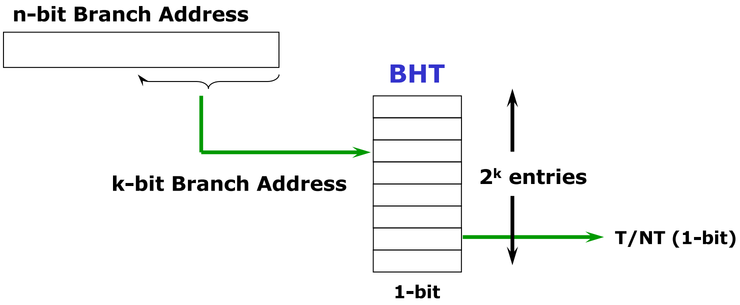


Figure 15: Visual representation of the 1-bit Branch History Table.

Example 5: Accuracy Issue

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

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

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

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

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

⊗ Shortcomings

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

✓ Partial Solutions

To **reduce interference**:

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

2.6.2 2-bit Branch History Table

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

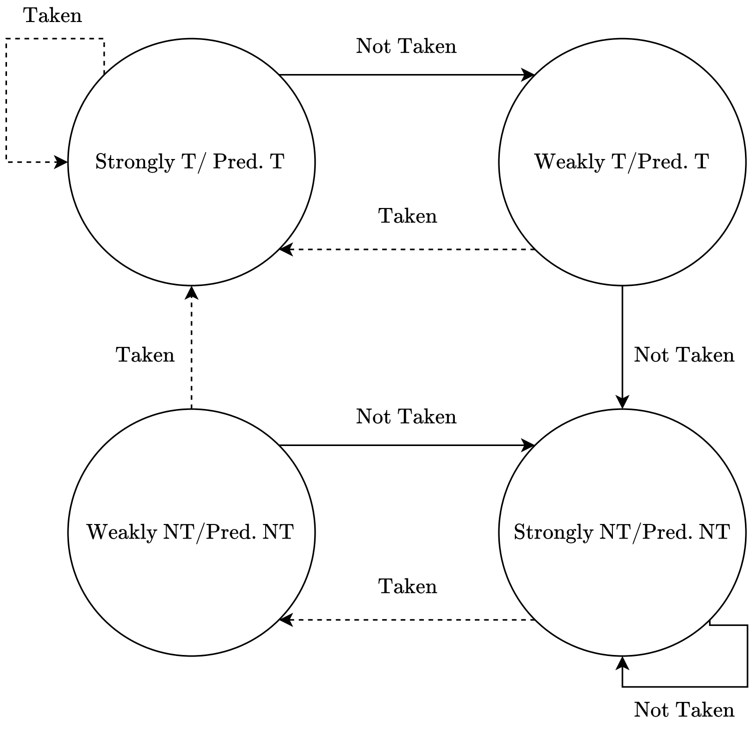
✓ Why 2-bit? The Problem with 1-bit BHT

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

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

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

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



✓ Effect on Loops

Assume a loop with:

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

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

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

✓ Benefits

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

2.6.3 Branch Target Buffer

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

Core Idea

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

How Is the BTB Structured?

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

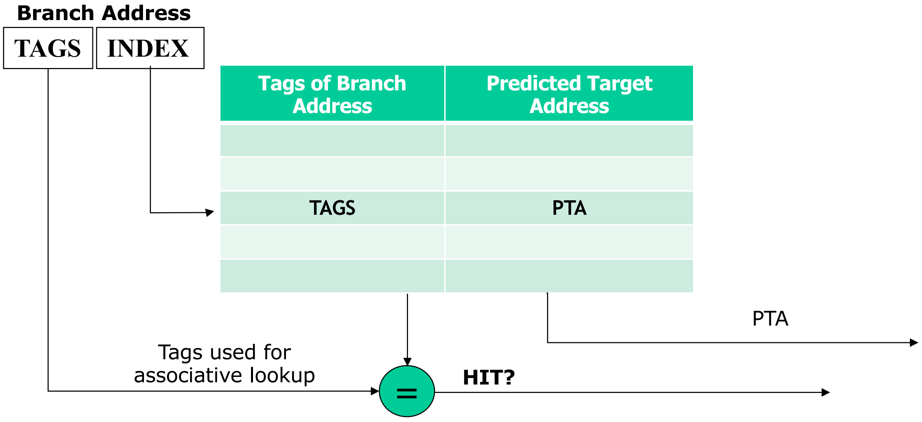


Figure 16: Branch Target Buffer without Branch Outcome Predictor.

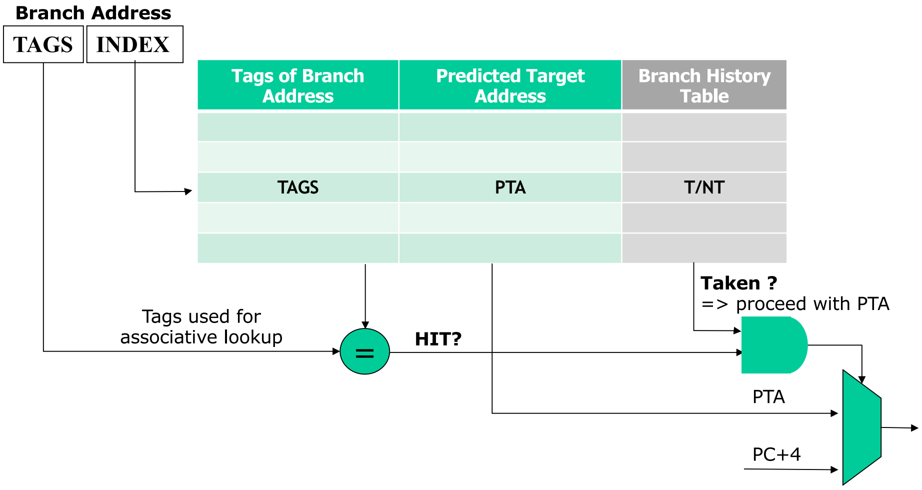


Figure 17: Branch Target Buffer with Branch Outcome Predictor.

BTB in the Pipeline

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

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

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

Table 4: Summary of Behavior.

✓ Advantages

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

2.6.4 Correlating Branch Predictors

What is the problem?

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

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

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

Key Idea

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

General Case: (m, n)

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

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

It works like this:

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

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

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

✓ Advantages

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

2.6.4.1 (1,1) Correlating Predictors

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

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

Consider a pseudo code:

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

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

- Cycle 1: **First Execution of `Branch A`**
  1. Global History: unknown or NT (0); because it doesn't track anything yet. We assume Not Taken (0).
  2. Use Table T2 (since `GH = 0`).
  3. Index into T2 with `Branch A`'s PC bits.
  - 🪄 Predict: Not Taken!
  - 👎 Unfortunately, the Branch Outcome (BO) says Taken ⇒ ✗ **Mispredict** ⇒ update T2 entry to T.
  4. Update Global History = 1 (Taken).
- Cycle 2: **Now Executing Branch `Branch B`**
  1. Global History = 1 (Taken) from `Branch A`.
  2. Use Table T1 (since `GH = 1`).
  3. Index with `Branch B`'s PC bits.
  - 🪄 T1 says: "Try to predict as Taken".
  - ⏲ The Branch Outcome (BO) says Taken ⇒ ✓ **Correct prediction**.
  4. No update needed. The Global History doesn't need an update either, because it is already 1 (Taken).

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

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

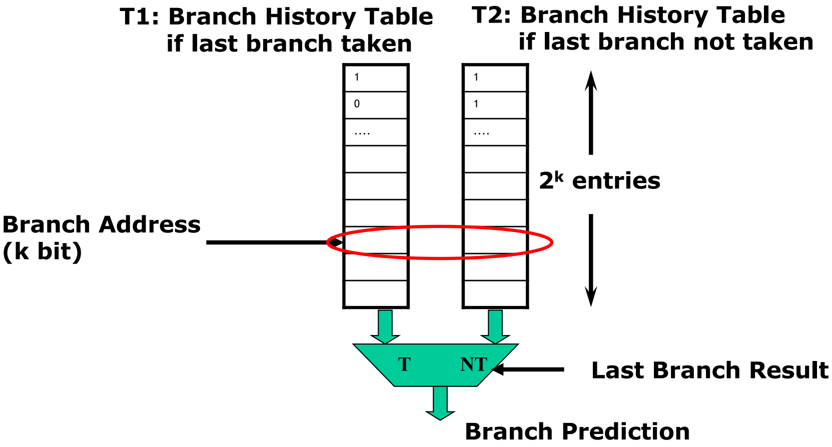


Figure 18: Visual representation of the (1,1) Correlating Predictor.

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

Table 5: (1,1) Correlating Predictor.

2.6.4.2 (2,2) Correlating Predictors

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

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

Consider a pseudo code:

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

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

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

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

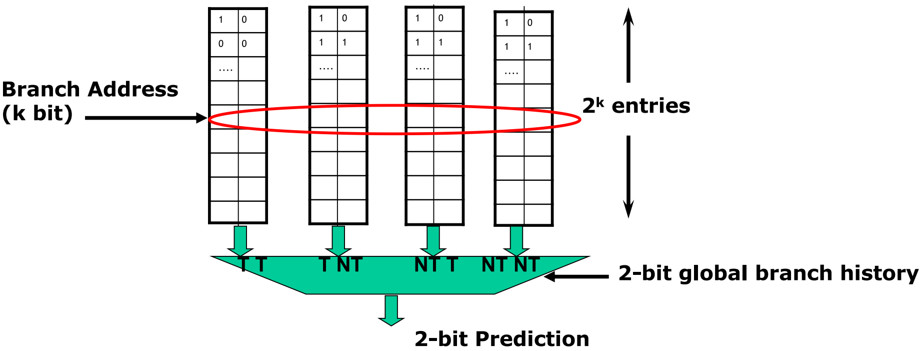


Figure 19: Visual representation of the (2,2) Correlating Predictor.

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

Table 6: (2,2) Correlating Predictor.

2.6.5 Two-Level Adaptive Branch Predictors

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

Core Concept

The two-level approach consists of:

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

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

Structure

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

Global Adaptive Predictor (GA)

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

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

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

GShare Predictor

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

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

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

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

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

Table 7: Summary of Global Adaptive and GShare.

3 Instruction Level Parallelism

3.1 The problem of dependencies

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

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

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

✓ Correct Program Behavior

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

1. **Data Flow**. The correct **values** must be **produced and consumed in the proper order**.

2. **Exception Behavior**. Reordering must **not alter the way exceptions are raised and handled** in the program.

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

3.1.1 Data Dependencies

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

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

Why is it called "*true*"?

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

Example 1: RAW Hazard

```
I1:    r3 ← r1 + r2    # produces a value in r3
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

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

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

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

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

Example 3: WAW Hazard

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

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

Resolving Name Dependencies: Register Renaming

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

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

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

Example 4: Resolving WAR and WAW

Original code (WAR):

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

Renamed:

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

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

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

Renamed:

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

⚖ Hardware vs. Software Register Renaming

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

3.1.3 Control Dependencies

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

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

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

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

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

Example 5: Control Dependencies

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

In assembly:

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

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

Why Control Dependencies matter for ILP

Control dependencies **limit instruction parallelism**:

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

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

Control Dependencies Solution

We have dedicated an entire section to this topic, see 2, page 27.

3.2 Multi-Cycle Pipelining

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

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

Motivation and Assumptions

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

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

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

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

Basic assumptions in this model:

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

3.2.1 Multi-Cycle In-Order Pipeline

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

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

This model retains a strict discipline:

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

Advantages

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

Disadvantages

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

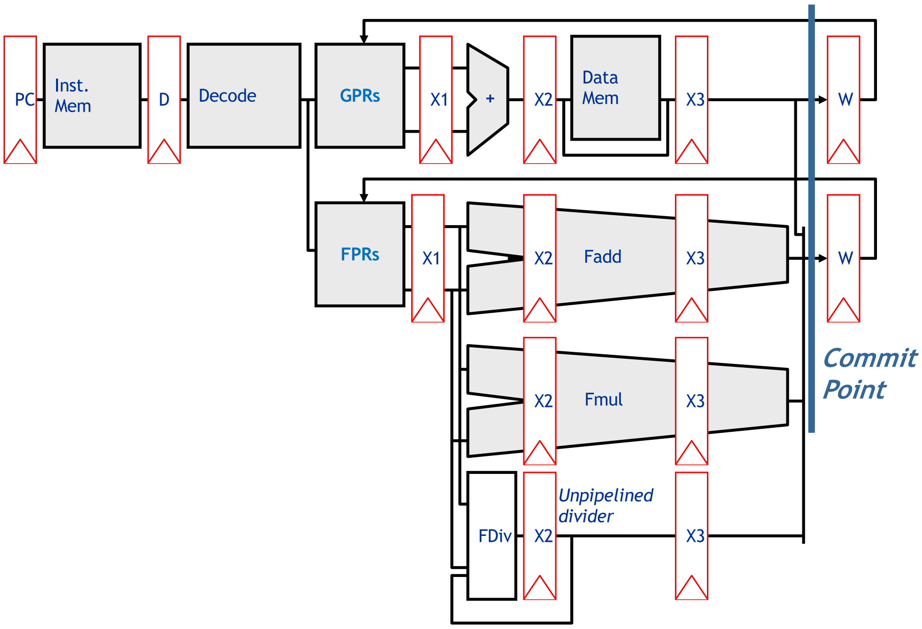


Figure 20: Multi-Cycle In-Order Pipeline architecture. This processor includes **different execution units**, each optimized for a specific operation type:

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

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

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

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

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

3.2.2 Multi-Cycle Out-of-Order Pipeline

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

In this model:

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

✓ Advantages

✓ **Maximizes ILP** by letting independent instructions execute as soon as possible.

✓ **Improves throughput** by keeping all functional units busy.

✓ **Hides long latencies**, like FP division or memory misses.

⚠ Challenges

Out-of-Order execution introduces serious architecture challenges:

- **WAR and WAW Hazards**. If later instructions write back before earlier ones:

  ✗ They might overwrite data that's still needed.

  ✓ Hardware **must detect and prevent** these scenarios.

  This is typically handled using: register renaming and scoreboarding / reservation stations.

- **Imprecise Exceptions**. If an exception occurs (e.g., divide-by-zero), but the processor has already executed and committed later instructions, the architectural state is **no longer consistent** with the point of the fault. To fix this, high-performance CPUs use:

  ✓ **Reorder Buffers (ROB)** to store results until it's safe to commit them in program order.

- ✓ **Checkpointing and rollback mechanisms** to recover precise state.

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

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

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

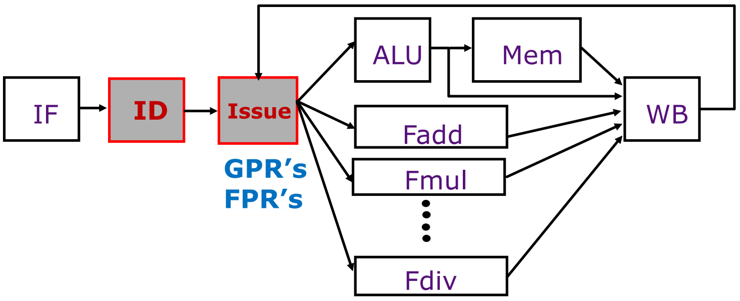


Figure 21: High-Level Multi-Cycle Out-of-Order Pipeline architecture.

- IF (Instruction Fetch): fetches the next instruction in program order from instruction memory using the program counter (PC).
- ID (Instruction Decode). This stage is now split into two sub-stages:

  1. ID: decoding the instruction format and operation type.
  2. Issue: reading registers and checking availability of operands.

  This split is key to **preparing instructions early**, even if they're not ready to execute immediately.

- Functional Units. The processor has **multiple independent execution units**, each possibly multi-cycle and with different latencies:

  – ALU: used for integer arithmetic, logic, and takes 1-2 cycles.

  – Mem: used for load/store, with cache hits/misses, and takes a variable number of cycles.

  – Fadd: used for floating-point addition, and takes ≈ 3 cycles.

  – Fmul: used for floating-point multiplication, and takes ≈ 3 cycles.

  – Fdiv: used for floating-point division, and takes multiple cycles and is not always pipelined.

  Each unit operates independently, and several can be active at once.

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

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

3.3 Dynamic Scheduling

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

The Need for Dynamic Scheduling

Consider the following instruction sequence:

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

In a naive in-order pipeline:

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

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

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

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

How Dynamic Scheduling Works

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

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

In a dynamically scheduled pipeline, stages are typically:

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

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

✓ Benefits

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

⚠ Challenges Introduced

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

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

3.4 Multiple-Issue Processors

3.4.1 Introduction to Multiple-Issue Pipelines

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

Definition 1: Multiple-Issue Processors

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

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

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

⚠ **The Limits of Scalar Pipelines**

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

Let's consider a classic 5-stage pipeline:

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

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

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

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

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

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

Raising Performance: Introducing Multiple Issue

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

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

This means:

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

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

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

Definition 2: Dual-Issue Pipeline

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

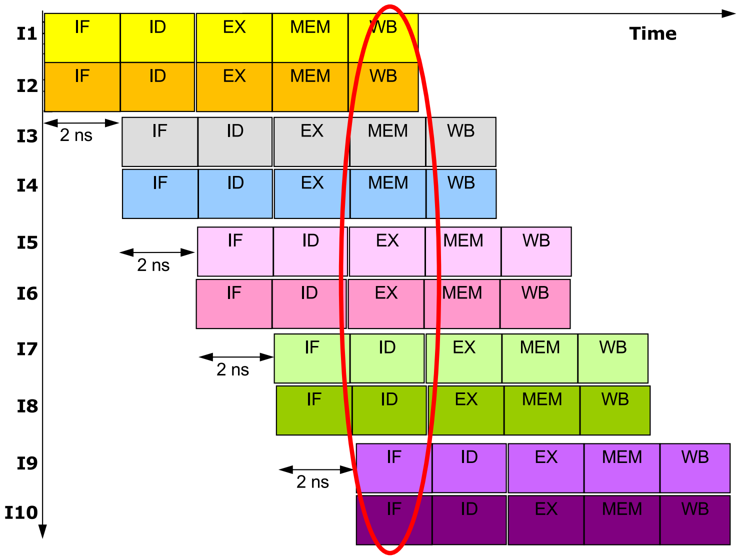


Figure 22: Dual-Issue Pipeline timeline.

Architectural Requirements

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

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

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

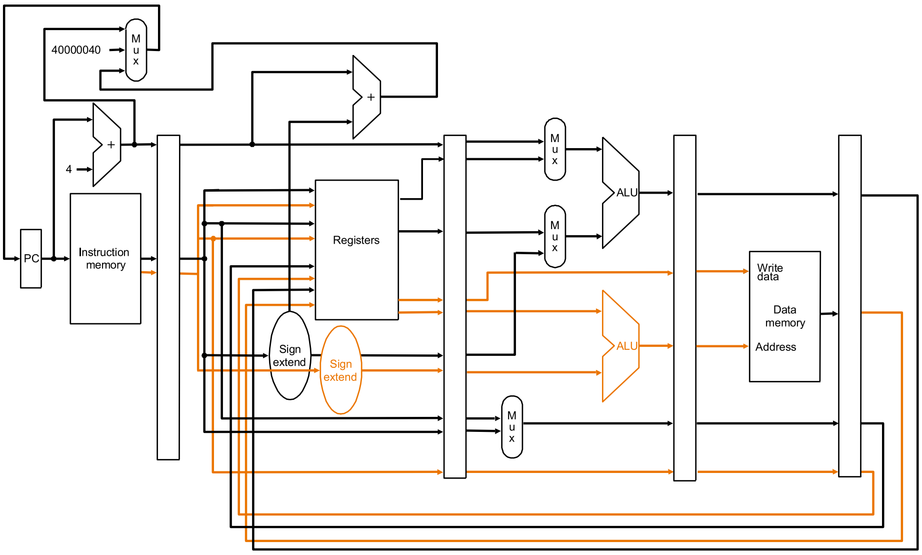


Figure 23: Dual-Issue Pipeline architecture.

3.4.2 Evolution Towards Superscalar Execution

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

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

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

? 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:

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

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 write-back (or commit) can happen out-of-order, introducing hazards such as:
  - Write After Write (WAW)
  - Write After Read (WAR)

  These are especially critical since there's no register renaming mechanism (page 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**.

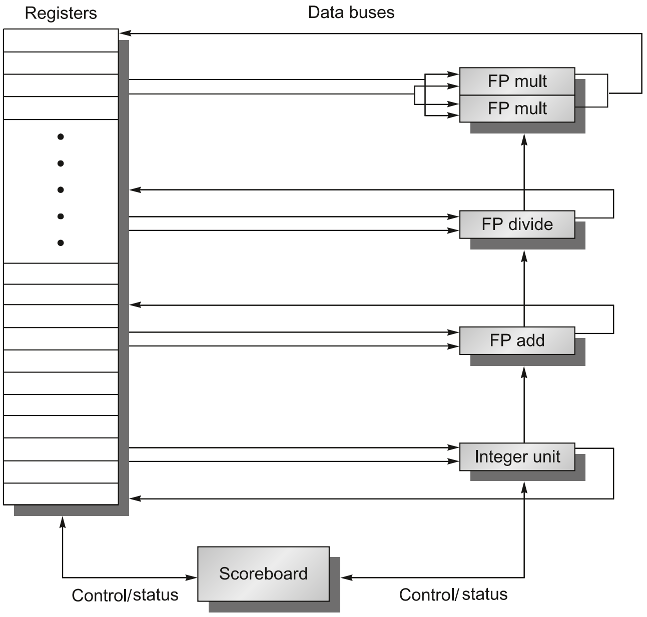


Figure 24: The basic structure of a RISC V processor with a scoreboard. [1]

- 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.

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