# Advanced Computer Architectures

*Claudio Tessa - 2024/2025*

# 1 Pipelining: Basic Concepts

**Pipelining** is an implementation technique where multiple instructions are overlapped in execution. Like an assembly line, every step of the pipeline completes a part of an instruction.

## 1.1 RISC-V Instruction Set

RISC architectures are characterized by a few key **properties**:

- All operations on data apply to data in registers and change the entire register.
- The only operations that affect memory are load and store operations.
- The instruction formats are few in number.

These simple properties lead to dramatic simplification in the implementation of the pipelining.

There are three classes of instructions:

1. **ALU instructions** - These instructions take two registers (or a register and a sign-extended immediate), operate on them, and store the result into a third register.
   - `add rd, rs1, rs2` $\iff$ `rd = rs1 + rs2`
   - `addi rd, rs1, 4` $\iff$ `rd = rs1 + 4`
2. **Load and store instructions** - These instructions take a register source and an immediate, called *offset*. The sum of the content of the source register and the offset is used as the memory address. Another register is also taken to performing the operation.
   - `ld rd, offset (rs1)` $\iff$ `rd = Memory[rs1 + offset]`
   - `sd rs2, offset (rs1)` $\iff$ `Memory[rs1 + offset] = rs2`
3. **Branches and Jumps** - Branches are conditional transfers of control, while jumps are unconditional.
   - `beq rs1, rs2, L1` $\iff$ `if (rs1 == rs2) then go to L1`
   - `j L1` $\iff$ `go to L1`

## 1.2 Phases Of Execution Of RISC-V Instructions

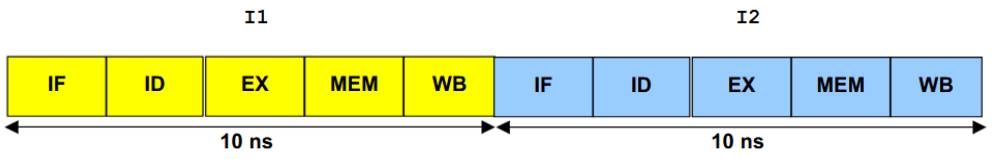
Every instruction takes at most 5 clock cycles. The 5 clock cycles are as follows:

1. **IF (Instruction Fetch)** - Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by, adding 4 as every

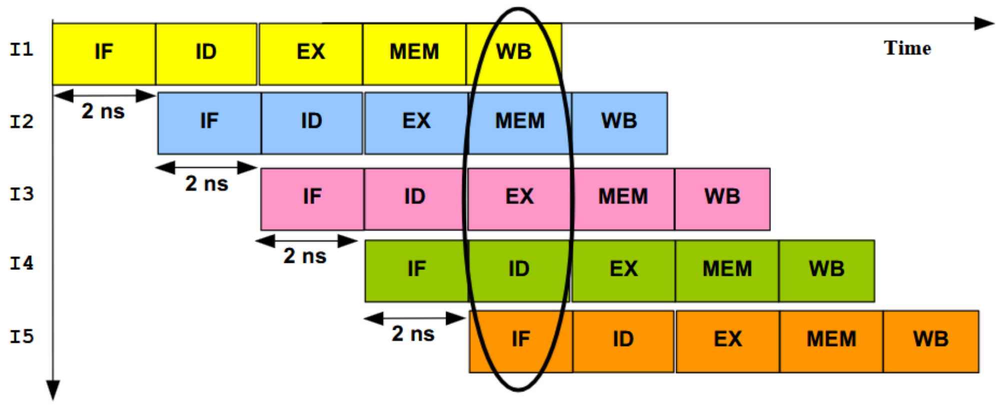
instruction is 4 bytes long in memory.
2. **ID (Instruction Decode)** - Decode the instruction and read the registers specified in the instruction. Decoding is done in parallel with reading registers.
3. **EX (execution)** - The ALU operates on the operands prepared in the previous cycle.
4. **MEM (Memory Access)** - Based on the instruction, the memory writes or reads data using the effective address computed in the previous cycle.
5. **WB (Write Back)** - Register-Register ALU instruction or lead instruction

## 1.3 RISC-V Pipelining

Instead of executing every instruction sequentially, as follows:



we can pipeline the execution by simply starting a new instruction on each clock cycle:



Each of the clock cycles now becomes a **pipe stage**. Although each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction.

However, we must ensure that the instructions in different stages of the pipeline do not interfere with one another. This separation is done by introducing **pipeline registers** between successive stages of the pipeline, so that at the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next cycle.

## 1.4 Pipeline Hazards

A **hazard** (conflict) is a situation that prevents the next instruction from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

1. **Structural hazards** - Attempt to use the same resource from different instructions simultaneously.
2. **Data hazards** - Attempt to use a result before it is ready.

3. **Control hazards** - Attempt to make a decision on the next instruction to execute before the condition is evaluated.

Hazards in the pipeline can make it necessary to **stall** the pipeline.

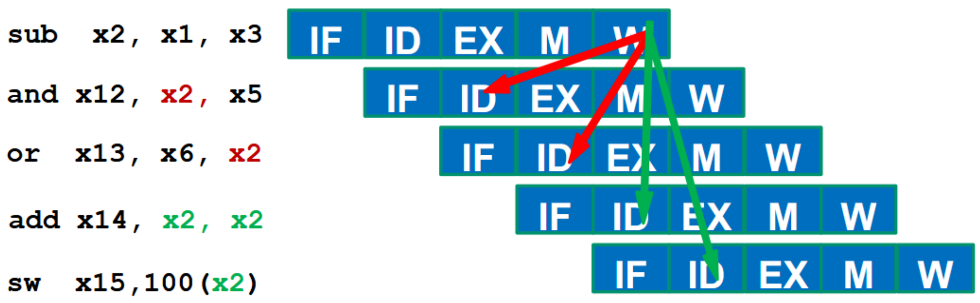
Note that there are **no structural hazards** in the RISC-V architecture, as the instruction memory is separated from the data memory, and the same register can be read and written in the same clock cycle by different instructions.

## 1.5 Data Hazards

Data hazards can arise when instructions that are dependent on each other are **too close** in the pipeline. Data hazards generate **RAW hazards** in the pipeline. RAW (Read After Write) hazard happens when instruction $n+1$ tries to read a source operand before the previous instruction $n$ has written its value in said register.

---

***Example***



```
sub  x2, x1, x3
and x12, x2, x5
or  x13, x6, x2
add x14, x2, x2
sw  x15,100(x2)
```

---

We will assume that the RF (register file) reads from registers in the second half of the clock cycle and writes to registers in the first half of the clock cycle. In this way, it is possible to read and write the same register in the same clock cycle without any stall.

Now, to solve the data hazard problem, we can use different techniques:

- **Compilation techniques** (*static-time techniques*) - They must be implemented before running the program, at compile time.
  - Insertion of `nop` (no operation) instructions. A `nop` effectively makes the processor "wait" (by executing empty instructions) one clock cycle. Enough `nop` can be inserted until the hazard is no longer there.
  - Instruction scheduling to avoid that correlating instructions are too close. The compiler tries to insert independent instructions among correlated instructions, otherwise inserts `nop`.
- **Hardware techniques** (*runtime techniques*) - Can be implemented directly at runtime when an hazard is detected.

- Insertion of *stalls* in the pipeline. Stalls effectively make the pipeline wait a few clock cycles until the current instruction finishes and the hazard is no longer there. They are analogous in performance to `nop` instructions, but they stop the instructions at any stage, rather than just making the processor wait before starting a new instruction.
- Data forwarding or bypassing. It works by using temporary results stored in the pipeline registers instead of waiting for the write back of results in the RF. In other words, the result from the ALU can be fed back in the ALU at the next stage, rather than waiting for it to be stored and then read from a register.

## 1.6 Performance Evaluation Of Pipelines

To evaluate the performance we use the following metrics: **IC** (Instruction Count), CPI (**Clocks** Per Instructions), and IPC (**Instructions** Per Clock):

- $\#\text{Clock Cycles} = \text{IC} + \#\text{Stall Cycles} + 4$
- $\text{CPI} = \frac{\#\text{Clock Cycles}}{\text{IC}} = \frac{\text{IC} + \#\text{Stall Cycles} + 4}{IC}$
- $\text{IPC} = \frac{1}{\text{CPI}}$
- $\text{MIPS} = \frac{f_{\text{clock}}}{CPI \times 10^6}$, where
  - $f_{\text{clock}}$ is the frequency of the clock
  - MIPS stands for Millions of Instructions per Second

Let's now evaluate the performance of a loop. Consider $n$ iterations of a loop composed of $m$ instructions per iteration, requiring $k$ stalls per iteration:

- $\text{IC}_{\text{per\_iter}} = m$
- $\#\text{ Clock Cycles}_{\text{per\_iter}} = \text{IC}_{\text{per\_iter}} + \#\text{ Stall Cycles}_{\text{per\_iter}} + 4$
- $\text{CPI}_{\text{per\_iter}} = \frac{\#\text{ Clock Cycles}_{\text{per\_iter}} = \text{IC}_{\text{per\_iter}} + \#\text{ Stall Cycles}_{\text{per\_iter}} + 4}{\text{IC}_{\text{per\_iter}}} = \frac{m+k+4}{m}$
- $\text{MIPS}_{\text{per\_iter}} = \frac{f_{\text{clock}}}{\text{CPI}_{\text{per\_iter}} \times 10^6}$

Let's now evaluate the asymptotic performance on the same loop:

- $\text{IC}_{\text{AS}} = m \cdot n$
- $\#\text{ Clock Cycles} = \text{IC}_{\text{AS}} + \#\text{ Stall Cycles}_{\text{AS}} + 4$
- $\text{CPI}_{\text{AS}} = \lim_{n\to\infty} \frac{\text{IC}_{\text{AS}} + \#\text{ Stall Cycles}_{\text{AS}} + 4}{\text{IC}_{\text{AS}}} = \lim_{n\to\infty} \frac{m \cdot n + k \cdot n + 4}{m \cdot n} = \frac{m+k}{m}$
- $\text{MIPS}_{\text{AS}} = \frac{f_{\text{clock}}}{\text{CPI}_{\text{AS}} \times 10^6}$

Overall, the **ideal CPI** on a pipelined processor would be 1, but stalls cause the pipeline performance to degrade from the ideal performance, so we have:

$$\text{Avg CPI} = \frac{\text{Ideal CPI} + \text{Pipe Stall Cycles per Instruction}}{1 + \text{Pipe Stall Cycles per Instruction}}$$

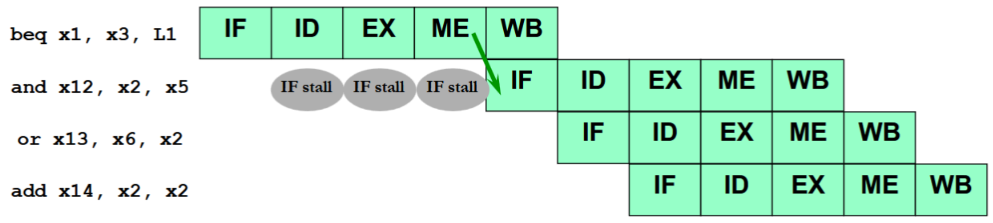
Pipeline stall cycles per instructions are due to structural hazards, data hazards, control hazards, and memory stalls.

## 1.7 Control Hazards

Control hazards can cause a greater loss in performance than data hazards. When a branch is executed, it may or may not change the PC. If a branch changes the PC to its target address, it is a **taken branch**. If it falls through, it is **not taken**.

### 1.7.1 Conservative Solution

To feed the pipeline, we need to fetch a new instruction at each clock cycle, but the branch decision is not yet ready. This problem to choose the correct instruction to be fetched after a branch is called **Control Hazard** or **Branch Hazard**. Control hazards arise from the pipelining of conditional branches and other `jump` instructions changing the PC. They reduce the performance from the ideal speedup gained by the pipelining because **it is needed to stall the pipeline until branch resolution**.



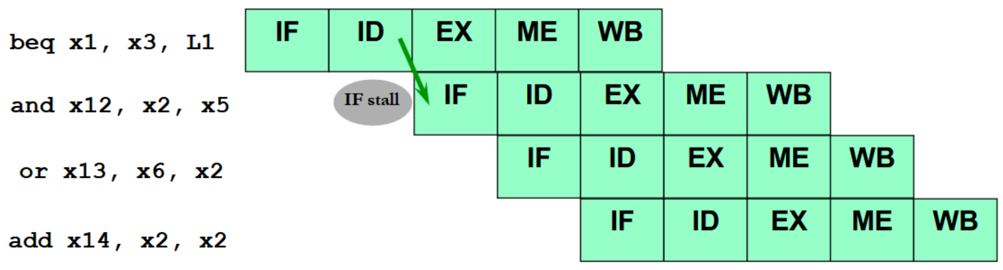
This way, each branch costs a penalty of **3 stalls** to decide and fetch the correct instruction flow in the pipeline.

### 1.7.2 Early Evaluation Of Branch in ID Stage

The idea is to improve the performance by

1. Compare the registers to derive the branch outcome (BO)
2. Compute the branch target address (BTA)
3. Update the PC register

**as soon as possible in the pipeline**. RISC-V pipeline anticipates the steps 1, 2, 3 **during the ID stage.**



Her we can insert only 1 stall before the BTA is calculated.

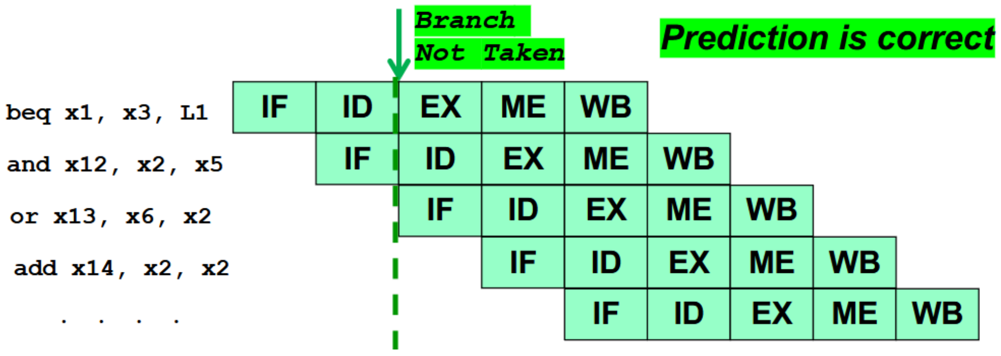
### 1.7.3 Static Branch Prediction Techniques

In static branch prediction techniques, the prediction is fixed at **compile time** for each branch, during the entire execution of the program. There are different types of static branch prediction techniques.

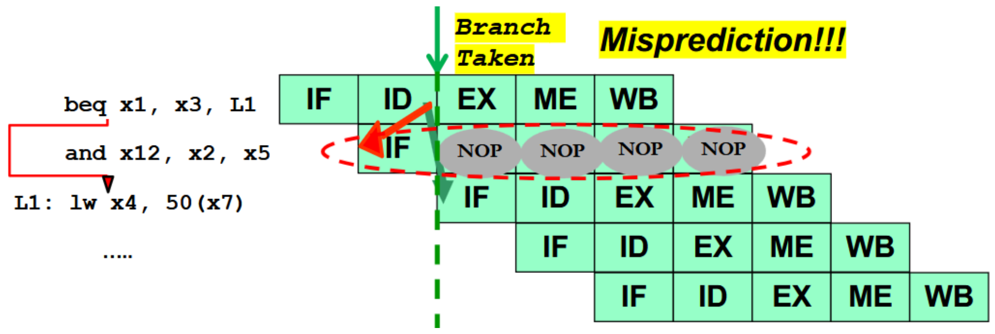
#### 1.7.3.1 Branch Always Not Taken (*Predicted-Not-Taken*)

This is the **easiest prediction**, always assume that the branch will be **not taken**. It is suitable for `if-then-else` statements, when the `then` clause is the most likely and the program will continue sequentially.

The next instruction in the pipeline is simply fetched as normal. If the branch outcome then is actually not taken, the pipeline will be fine.



What happens if the branch will be taken? In this case it will be necessary to **flush** the instruction after the branch (as it should have not been executed) and fetch the next instruction at the target address. If a branch is taken, it will cost 1 stall penalty.



This increases performance depending on how many *taken* branches there are.

#### 1.7.3.2 Branch Always Taken (*Predicted-Taken*)

This is the dual case of the previous technique: we assume every branch as **taken**. It is suitable for **backward branches** such as `for` loops and `do-while` loops, that are most likely taken.

The problem here is that we need to know the **branch target address (BTA)** to start fetching the instructions at the target. In the **IF** stage, we need to add a **branch target**

**buffer** where to store the **predicted target address** based on the previous branch behavior.

This makes it similar to the predicted-not-taken technique: if the prediction is wrong, we get one cycle penalty, otherwise the pipeline continues as normal.

#### 1.7.3.3 Backward Taken Forward Not Taken (*BTFNT*)

In this case the **prediction is based on the branch direction**: backward-going branches are predicted as **taken**, while forward-going branches are predicted as **not-taken**.

#### 1.7.3.4 Profile-Driven Prediction

We **profile** the behavior of the application program by several early execution runs by using different datasets. The prediction is based on **profiling information** about the branch behavior collected during earlier runs.

The profile-driven prediction methos requires a **compiler hint bit** encoded by the compiler in the branch instruction format:

- Set to **1** if **taken** is the most probable branch outcome
- Set to **0** if **not taken** is the most probable branch outcome

#### 1.7.3.5 Delayed Branch

Instead of stalling the pipeline for one cycle, the compiler tries to find a valid and useful instruction to be scheduled while the branch is being resolved (called the **branch delay slot**). There are 4 ways to schedule an instruction in the branch delay slot:

1. **From before** - The branch delay slot is scheduled with an **independent** instruction from before the branch. The instruction in the branch delay slot is **always executed**, regardless of the branch outcome. The execution will then continue, based on the branch outcome, in the right direction.
2. **From target** - The branch delay slot is scheduled with one instruction from the target of the branch (*taken* path) This strategy is preferred when the branch is **taken with high probability**, such as `do-while` loop branches. If the branch is then **mispredicted** (not taken), the instruction in the delay slot must be flushed (unless it's ok to execute it anyway)
3. **From fall-through** - The branch delay slot is scheduled with one instruction from the fall-through of the branch (*not taken* path). This strategy is preferred when the branch is **not taken with high probability**, such as `if-then-else` statements where the `else` path is less probable. If the branch is then **mispredicted** (taken), the instruction in the delay slot must be flushed (unless it's ok to execute it anyway)
4. **From after** - dual to *from before*.

### 1.7.4 Dynamic Branch Prediction Techniques

In dynamic branch prediction techniques, the prediction for each branch can change at **runtime** during the program execution.

The idea is to use the past behavior (runtime behavior) to predict at runtime the future branch behavior. To do this, we use hardware to **dynamically** predict the outcome of a branch. This mean that the prediction can change at runtime if the branch changes its behavior during execution.

Dynamic branch prediction is based on **two interactive hardware blocks**:

1. Branch Outcome Predictor (BOP) - To predict the direction of a branch (taken or not taken).
2. Branch Target Buffer (BTB) - To predict the branch target address.

They are placed in the instruction fetch (IF) stage, to predict the next instruction to read in the instruction cache.

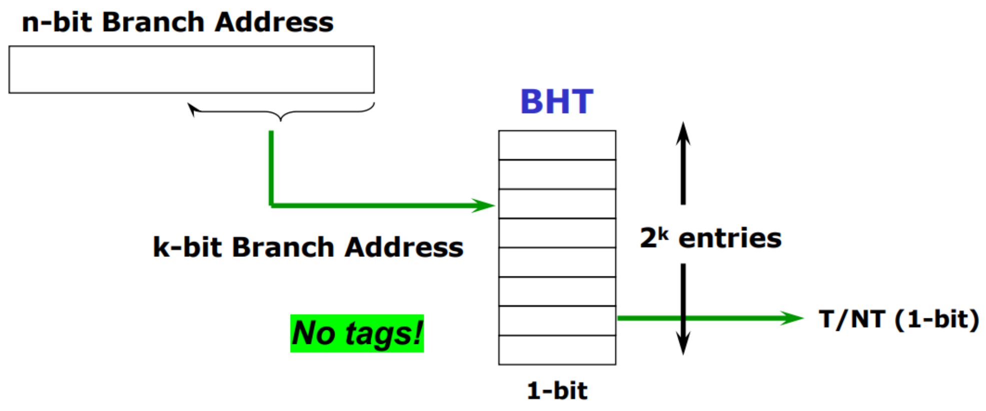
If the branch is predicted by the BOP in the IF stage as **not taken**, then the PC is incremented as usual. If the prediction is correct, there will be no penalty, otherwise we need to flush the instruction fetched after the branch with a one-cycle penalty.

If the branch is predicted by the BOP in IF stage as **taken**, the BTP gives the predicted target address. Similarly to the previous case, if the prediction is correct, there will be no penalty, otherwise we need to flush the instruction fetched after the branch with a one-cycle penalty.

### 1.7.4.1 Branch History Table (1-bit)

The **branch history table** is a table containing 1 bit for each branch that says whether the branch was recently taken or not taken. The behavior is controlled by a **finite state machine** with only 2 states to remember the last direction taken by the branch.

The table is indexed by the lower portion $k$-bit of the address of the branch instruction, to keep the size of the table limited. For locality reasons, we would expect that the most significant bits of the branch address are **not** changed. The table has **no tag check** (every access is a hit). The prediction bit may have been put there by another branch with the same low order address bits, but this doesn't matter, as the **prediction is just a hint**.



A **misprediction** occurs either when:

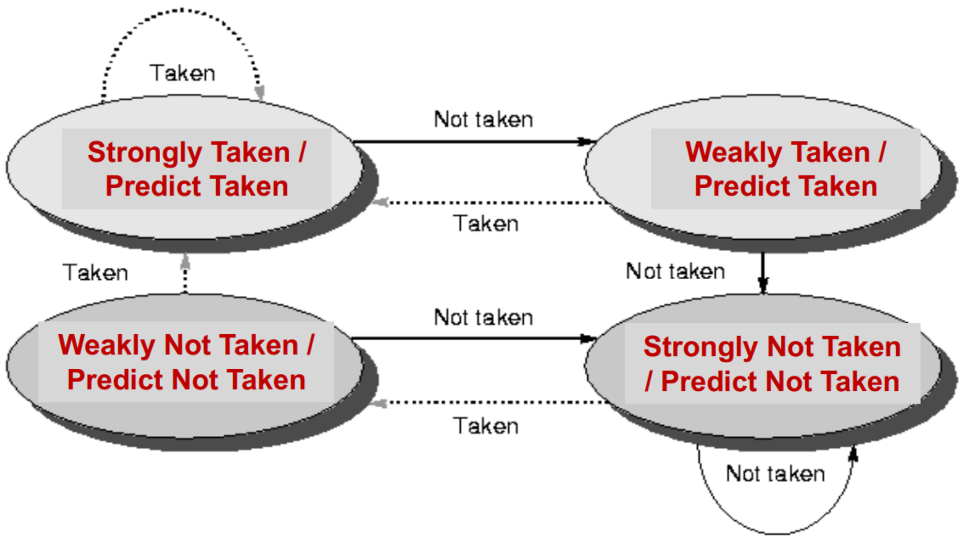
- The prediction is incorrect for that branch
- The same index has been referenced by two different branches, and the previous history refers to the other branch (this can occur because there is no tag check). To reduce this problem, it is enough to reduce the number of rows in the BHT (that is, to increase $k$), or to use a hashing function

In a **loop branch**, for example, a branch is almost always T, and then NT once at the exit of the loop. Therefore, the 1-bit BHT causes 2 mispredictions:

1. At the **last loop iteration**, since the prediction bit is T, while we need to exit from the loop
2. When we re-enter the loop, at the **first iteration** we need the branch to stay in the loop, while the prediction bit was flipped to NT on previous execution of the last iteration of the loop

### 1.7.4.2 Branch History Table (2-bit)

We can use a 2-bit BHT to encode 4 states, in order to change the prediction only after 2 mispredictions. The finite state machine then becomes

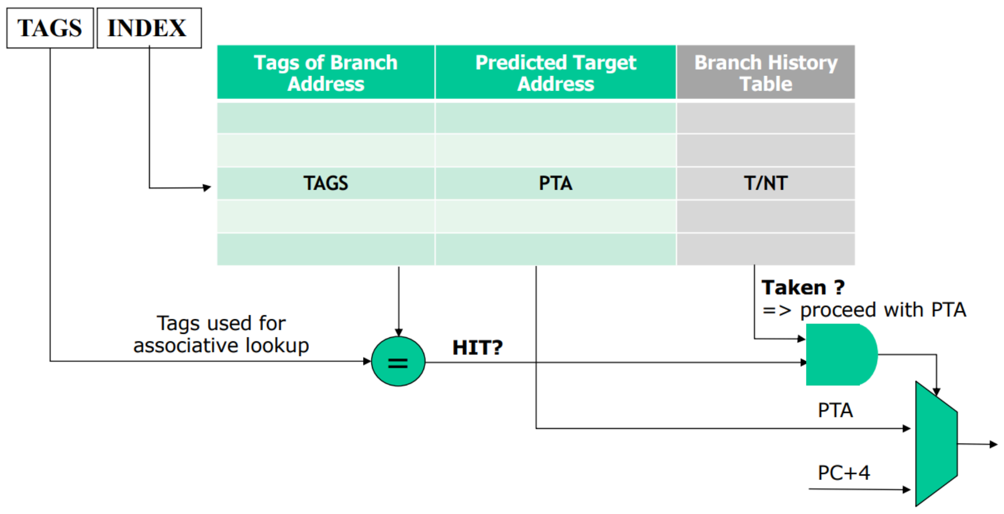

Taken
Strongly Taken / Predict Taken
Not taken
Weakly Taken / Predict Taken
Taken
Taken
Not taken
Weakly Not Taken / Predict Not Taken
Not taken
Strongly Not Taken / Predict Not Taken
Taken
Not taken

Coming back to the previous example, in the last loop iteration, we mispredict the branch but we do not need to change the prediction. When re-entering the loop, the branch is correctly predicted as taken.

### 1.7.4.3 Branch Target Buffer

The **branch target buffer (BTB)** is a cache storing the **predicted target address (PTA)** for the taken-branch instructions. The PTA is expressed as PC-relative.

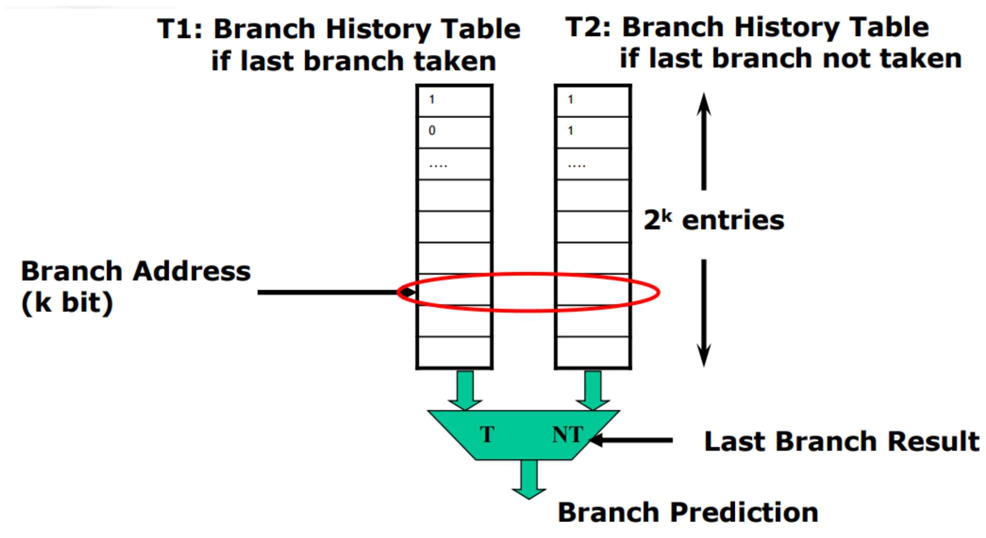
The BTB is designed as a direct-mapped cache placed in the **IF stage** by using the address of the fetched branch instruction to index the cache. Then, **tags** are used for the associative lookup. The branch target buffer is used in combination with the branch history table in the IF stage.



## 1.7.4.4 Correlating Branch Predictors

The 2-bit BHT uses only the recent behavior of a single branch to predict the future behavior of that branch. We can consider use the following **idea**: the behavior of recent branches are **correlated**, meaning that the recent behavior of **other branches** (rather than just the current branch) can also influence the prediction of the current branch.

This type of branch predictors that use the behavior of other branches to make a prediction are called **correlating predictors**.


T1: Branch History Table if last branch taken
T2: Branch History Table if last branch not taken
1
0
....
1
1
....
$2^k$ entries
Branch Address (k bit)
T
NT
Last Branch Result
Branch Prediction

We record if the most recently executed branches have been **taken** or **not taken**. The branch is predicted based on the previous executed branch by selecting the appropriate 1-bit BHT:

- One prediction is used if the last branch executed was **taken**
- Another prediction is used if the last branch executed was **not taken**

Normally, the last branch executed is **not** the same instruction as the branch being predicted (although this can occur in simple loops with no other branches in the loop).

In general, an $(m, n)$ correlating predictor records the last $m$ branches to choose from $2^n$ BHTs, each of which is an $n$-bit predictor.

The branch prediction buffer can be indexed by using a concatenation of low-order bits from the branch address with $m$-bit global history (i.e., global history of the most recent $m$ branches). For example, a 2-bit BHT predictor with no global history is simply a (0, 2) predictor.

### 1.7.4.5 Two-Level Adaptive Branch Predictors

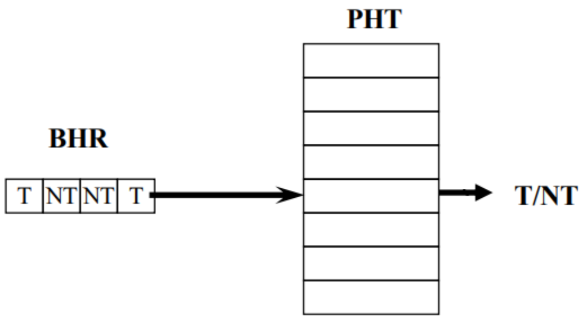
The first level history is recorded in one or more $k$-bit shift register, called **Branch History Register (BHR)**, which records the outcomes of the $k$ most recent branches (**used as global history**).

The second level history is recorded in one or more tables, called **Pattern History Table (PHT)** of 2-bit saturating counters (**used as a local history**).

The BHR is used to index the PHT to select which 2-bit counter to use. Once the 2-bit counter is selected, the prediction is made using the same methos as in the 2-bit counter scheme.

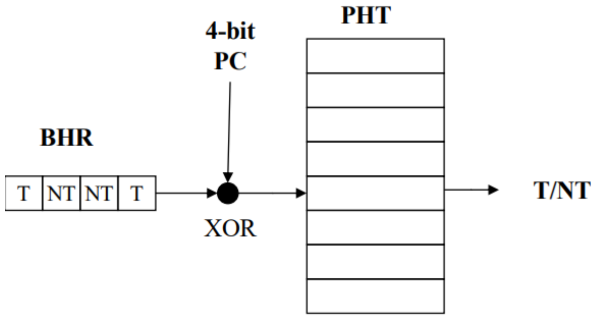
### 1.7.4.6 Global Adaptive Predictor

The Global Adaptive (GA) predictor uses the correlation between the current branch and the other branches in the global history to make the prediction: a PHT (local history) indexed by the content of BHT (global history).



### 1.7.4.7 GShare Predictor

It is a variation of the GA predictor where we want to correlate the BHR recording the outcomes of the most recent branches (global history) with the low-order bits of the branch address. **GShare**: we make the XOR of 4-bit BHR (global history), with the low-order 4-bit of PC (branch address) to index the PHT(local history)



### 1.7.5 Performance Of Branch Prediction Techniques

The **performance** of a branch prediction technique depends on:

- **Accuracy** - Measured in terms of percentage of incorrect predictions given by the predictor.
- **Cost** - Measured in terms of time lost to execute useless instructions (misprediction penalty) given by the processor architecture. The cost increases for deeply pipelined processors.
- **Branch frequency** - Given by the application. the importance of accurate branch prediction is higher in programs with higher number of branch instructions.

## 1.8 Instruction Level Parallelism

Pipelining overlaps the execution of instructions, exploiting the **instruction level parallelism**. The goal is to maximize the throughput, instructions per clock (IPC), and minimize the clocks per instructions (CPI). Pipelining improves instructions throughput, but not the latency of the single instruction.

Determining **dependencies** among instructions is critical to define the amount of parallelism existing in a program. If two instructions are **dependent** on each other, they cannot be executed in parallel, they must be executed in order, or only partially overlapped. There are three different types of dependencies in a code:

1. **True data dependencies** - Instruction $j$ is dependent on some data produced by a previous instruction $i$.
2. **Name Dependencies** - Two instructions use the same register or memory location.
3. **Control dependencies** - Seen in 1.7 Control Hazards. They impose the order of instructions.

### 1.8.1 Name Dependencies

A **name dependency** occurs when two instructions use the same register or memory location (called **data**), but there is **no flow of data** between the instructions associated with that name.

Name dependencies are **not** true data dependencies, since there is no value (no data flow) being transmitted between the two instructions, here it's just a **register reuse**.

Consider instruction $I_i$ that precedes instruction $I_j$ in the program order. There are two types of name dependencies:

1. **Anti-dependencies** - When $I_j$ writes in a register or memory location that instruction $I_i$ reads, it can generate a **Write After Read (WAR) hazard**. Original instructions ordering must be preserved to ensure that $I_i$ reads the previous value.

$$\begin{aligned} I_i:&\quad r_3 \leftarrow \textcolor{red}{r_1} \text{ operation } r_2 \\ I_j:&\quad \textcolor{red}{r_1} \leftarrow r_4 \text{ operation } r_5 \end{aligned}$$

2. **Output dependencies** - When $I_i$ and $I_j$ write in the same register or memory location, it can generate a **Write After Write (WAW) hazard**. Original instructions ordering must be preserved to ensure that the value finally written corresponds to $I_j$.

$$\begin{aligned} I_i:&\quad \textcolor{red}{r_3} \leftarrow r_1 \text{ operation } r_2 \\ I_j:&\quad \textcolor{red}{r_3} \leftarrow r_6 \text{ operation } r_7 \end{aligned}$$

We can solve name dependencies with **register renaming**: if the register used can be changed, then the instructions do not conflict anymore. This can be easily done if there are enough registers available in the ISA (instruction set architecture). It can be either done statically by the compiler, or dynamically by the hardware.

On the other side, dependencies through memory locations are more difficult to detect (**memory disambiguation** problem), since two addresses may refer to the same location but can look different.

A dependency of this kind can potentially generate an hazard, but the number of stalls to eliminate the hazard are part of the pipeline architecture (dependencies are a property of the **program**, while hazards are a property of the **architecture**).

In order for a program to be correct, it is necessary to respect these two critical **properties** (normally preserved by maintaining both data and control dependencies during scheduling):

1. **Data flow**: the actual flow of data values among instructions that produces the correct results and consumes them.
2. **Exception behavior**: preserving this property means that any changes in the ordering of instruction execution must not change how exception are raised in the program.

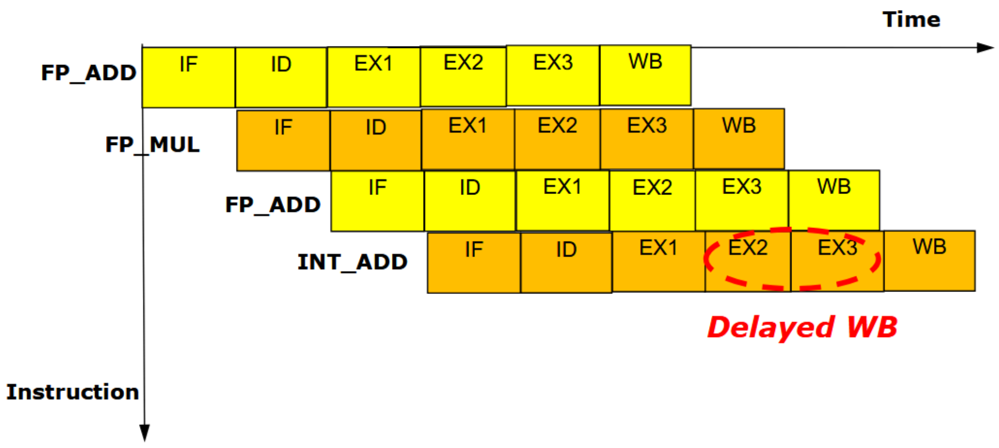
### 1.8.2 Multi-Cycle Pipelining

We will uses these basic assumptions:

- We consider **single-issue** processors (one instruction issued per clock).
- Instructions are **issued in-order**.

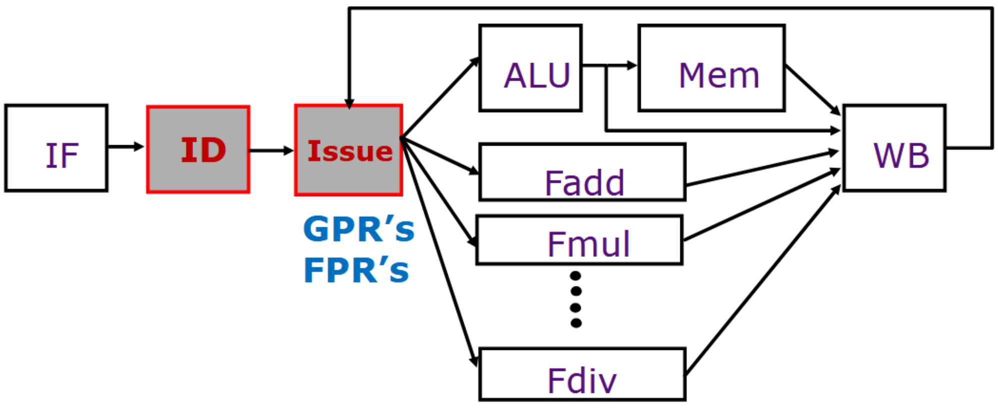
- The **execution stage** might require multiple cycles latency, depending on the operation type.
- **Memory stages** might require multiple cycles access time due to instruction and data cache misses.

### 1.8.2.1 Multi-cycle In-order Pipeline



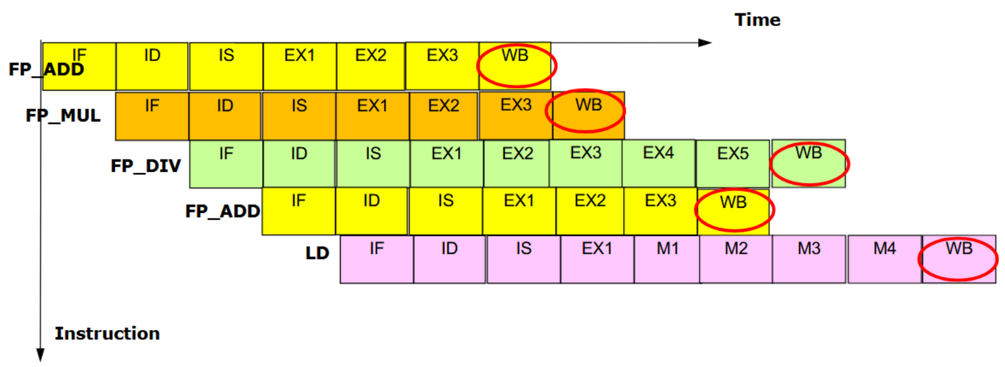
We se that, with this technique, the instructions are **issued in-order**, and **committed in-order**. This **avoids the generation of WAR and WAW hazards**, and preserves the p**recise exception** model.

### 1.8.2.2 Multi-cycle Out-of-order Pipeline



We can see the following characteristics:

- The ID stage is **split in 2** stages: *instruction decode* (**ID**) and *register read* (**Issue**).
- There are **multiple** functional units with **variable latency**.
- There exist **multi-cycle floating-point instructions** with long latency.
- The memory systems have variable access time, **multi-cycle memory accesses** due to data cache misses (unpredictable statically).
- No more commit point: **Out-of-order** commit. There is the need to **check for WAR and WAW** hazards and **imprecise exceptions**.



Even if the instructions are issued in-order, there is an **out-of-order commit** of instructions. This means that there is the need to **check for WAR and WAW** hazards and **imprecise exceptions.**

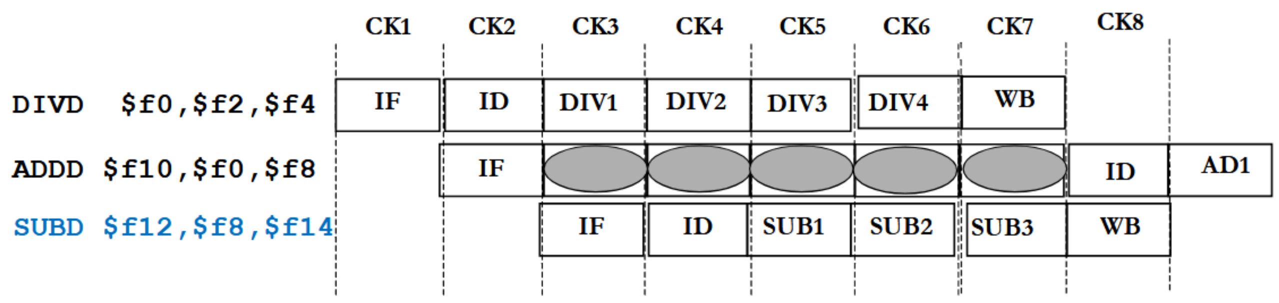
## 1.8.3 Dynamic Scheduling

Hazards due to true data dependencies that cannot be solved by forwarding cause **stalls** of the pipeline. This means that no new instructions are fetched nor issued, even if they are not data dependent.

The **solution** is to allow data independent instructions behind a stall to proceed. The hardware manages dynamically the instruction execution to reduce stalls: an instruction execution begins as soon as their operands are unavailable. This generates out-of-order execution and completion.

---

***Example***



Here `ADDD` **stalls** for RAW hazard on `$f0` (waiting many clock cycles for `DIVD` commit), `SUBD` would stall even if not data dependent on any previous instruction. The idea is to enable `SUBD` to proceed. This generates **out-of-order execution**.

---

## 1.8.4 Imprecise exceptions

An exception is **imprecise** if the processor state when an exception is raised does not look exactly as if the instructions were executed in-order.

Imprecise exceptions can occur **out-of-order** because:

- The pipeline may have **already** completed instructions that are **later** in the program order than the instruction causing the exception.
- The pipeline may have **not yet** completed some instructions that are **earlier** in the program order than the instruction causing the exception.

Imprecise exception make it difficult to restart execution after handling.

## 1.8.5 Multiple-Issue Processors

A scalar pipeline (the one seen up until now) are limited to a $CPI_{\text{ideal}} = 1$. That is, it can never fetch and execute more than **one instruction per clock** (single-issue). It can be even worst due to **stalls** added to solve hazards.

To reach higher performance, **more parallelism** must be extracted from the program (in other words, we need multiple-issue). It means fetching and executing **more than one** instruction per clock.

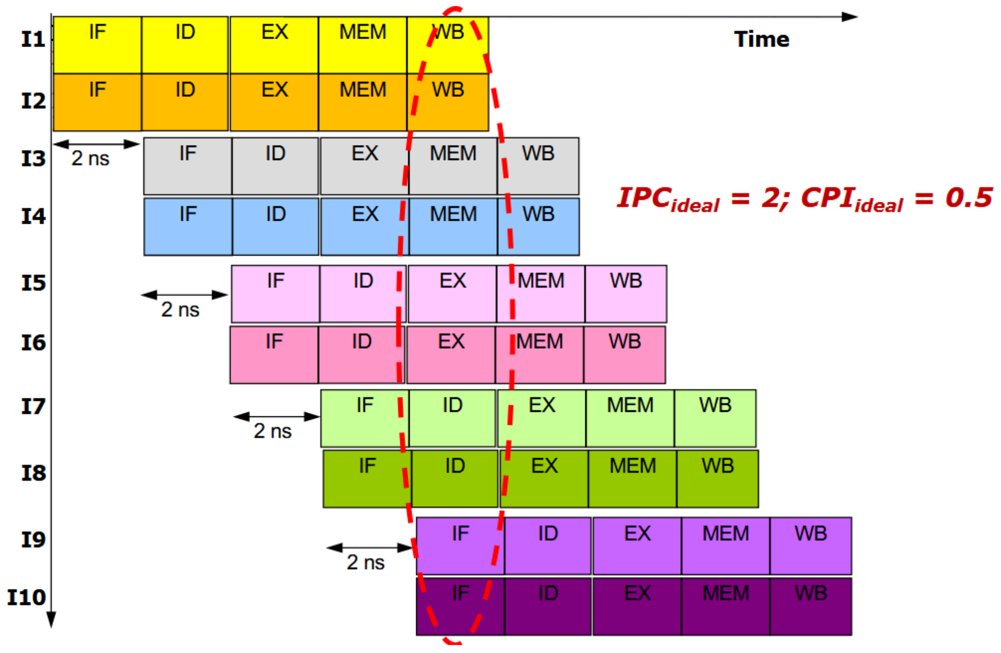
Instruction dependencies must be detected and solved: instructions must be *re-ordered* (i.e., **scheduling**) to achieve the highest instruction level parallelism given the available resources:

$$IPC_{\text{ideal}} > 1 \implies CPI_{\text{ideal}} < 1$$

---

***Example: Dual-Issue Pipeline***

With a 2-issue 5-stage pipeline there are $2 \times 5$ different instructions overlapped:



---

### 1.8.5.1 Dynamic Scheduling

**Dynamic scheduling** can be applied to **single-issue scalar** processor, but also to **multi-issue** processors (**superscalar** processors).

However, multiple-issue pipelines have disadvantages:

- Very complex logic and cost to check and manage dependencies at runtime (i.e., to decide with instructions can be issued at every clock cycle).
- Cycle time limited by scheduling logic (dispatcher and associated dependency-checking logic).
- It does not scale well. It is almost impractical to make the issue-width greater than 4.

### VLIW (Very Long Instruction Word) and Static Scheduling

VLIW processors issue a fixed number of instructions formatted either as one large instruction, or as a fixed instruction packet with parallelism among instructions explicitly indicated by the instruction.

VLIW relies on the compiler to schedule the code for the processor. This makes it similar to static scheduling, although static scheduling issues a varying (rather than a fixed) number of instructions per clock.

The advantages of VLIW increases as the maximum issue rate grows. Static scheduling, on the other hand, is typically used for narrow issue width, as their advantages diminish as the issue width grows.

These are the main disadvantages of static scheduling:

- **Unpredictable branch behavior**: the code parallelization is limited to basic blocks.
- **Unpredictable cache behavior**: variable memory latency for hits/misses.
- **Complexity of compiler technology**: the compiler needs to find a lot of parallelism in order to keep the multiple functional units of the processors busy.
- **Code size explosion** duo to insertion of NOPs.
- **Low code portability and binary code incompatibility**: consequence of the larger exposure of the microarchitecture (implementation details) at the compiler in the generated code.
- **Low performance portability**.

## ILP Limitations

The **issue width** is the number of instructions that can be issued in a single cycle by a **multiple-issue processor**.

When superscalars were invented, 2- and rapidly 4-issue width processors were created. However, due to the intrinsic level of parallelism of a program, it is hard to decide which 8, or

16, instructions can execute every cycle.

The solution is to **introduce more levels of parallelism**

## Scoreboard Dynamic Scheduling Algorithm

In the case of the **simple scalar pipeline**, hazards due to true data dependences that cannot be solved by forwarding cause the stall of the pipeline. No new instructions can be fetched nor issued even if there are not data dependencies in the new instructions.

The **solution** is to allow data independent instructions behind a stall to proceed. The hardware rearranges dynamically the instruction execution to reduce stalls. This causes out-of-order execution and out-of-order commit.

The **scoreboard** is a dynamic scheduling algorithm (implemented at the hardware level). We will make use of the following basic assumptions:

- We consider a **single-issue** processor.
- Instruction Fetch stage fetches and issued instructions in program order (**in-order issue**).
- Instruction execution begins as soon as operands are ready, whenever not dependent on previous instructions (**no RAW** hazards)
- There are **multiple** pipelined Functional Units with **variable latencies**.
- Execution stage might require **multiple cycles**, depending on the operation type and latency.
- Memory stage might require **multiple cycles** access time due to data cache misses.

This means that there is **out-of-order execution** and **out-of-order commit**, which introduces the possibility of **WAR** and **WAW** hazards.

## Scoreboard basics

Scoreboard allows **data independent instructions behind a stall to proceed**, not waiting for previous instructions.

In the time between when an instruction begins execution and when in completes execution, the instruction is **in execution**. The scoreboard pipeline allows **multiple instructions in execution at the same time**, meaning that it requires **multiple pipelined functional units**.

To recap, we have **in-order-issue**, but **out-of-order execution** and **out-of-order completion (commit)**, with no forwarding.

Scoreboard divides the **ID** stage in **two stages**:

1. **Issue** - Decode instructions and check for structural hazards.

2. **Read operands (RR)** - Wait until not dependent on previous instructions and no data hazards, then read operands.

Scoreboard allows instructions