

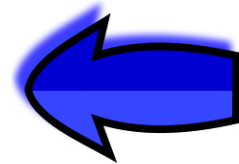
Pipelining Hazards and Mitigations

**Slide courtesy: Smruti
Ranjan Sarangi**

Slides adapted by: Dr Sparsh Mittal

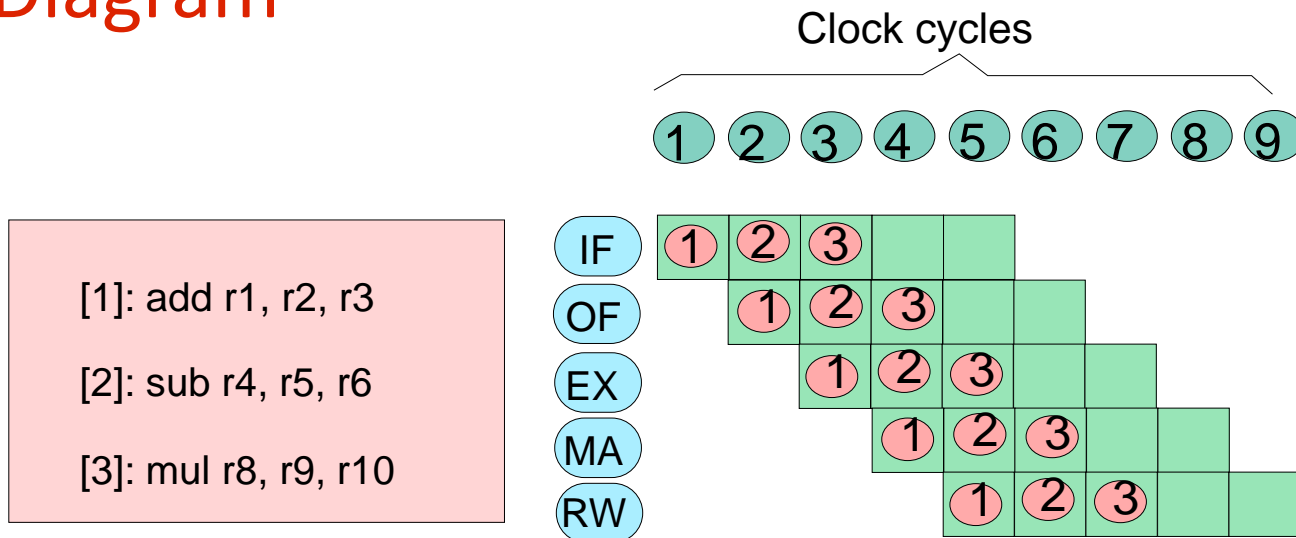
Outline

- * Pipeline Hazards
- * Pipeline with Interlocks



Pipeline Hazards

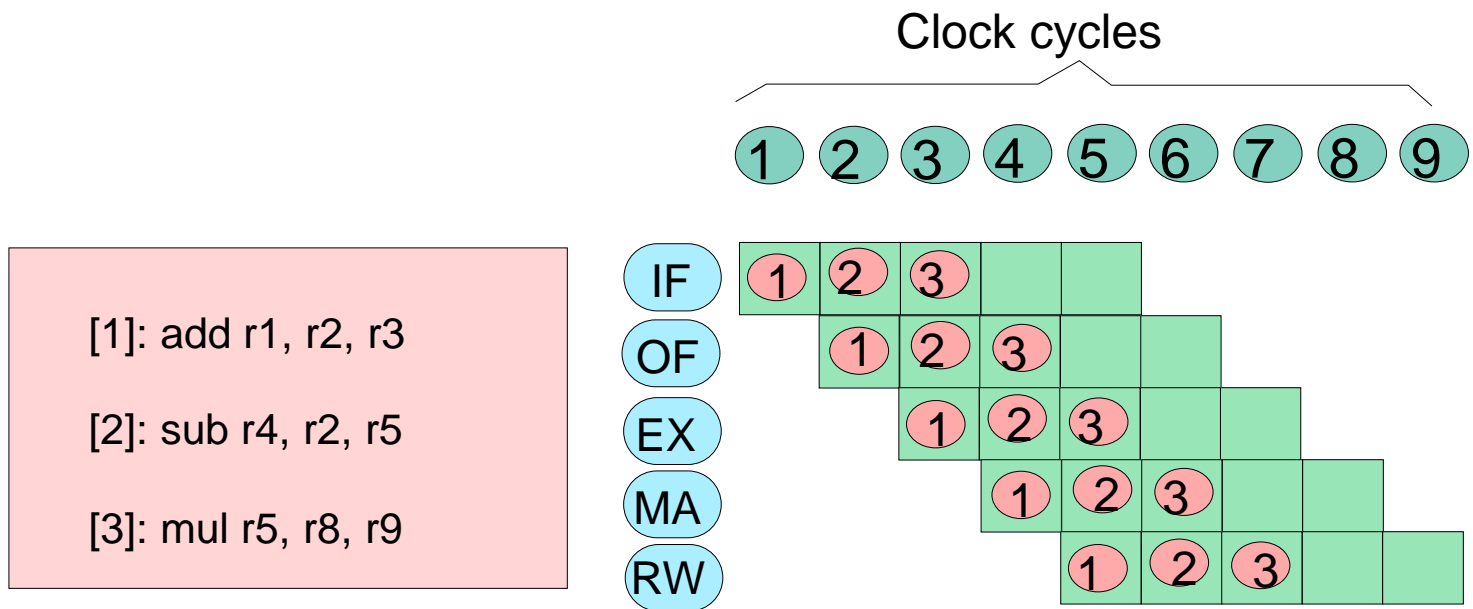
- * Now, let us consider correctness
- * Let us introduce a new tool → Pipeline Diagram



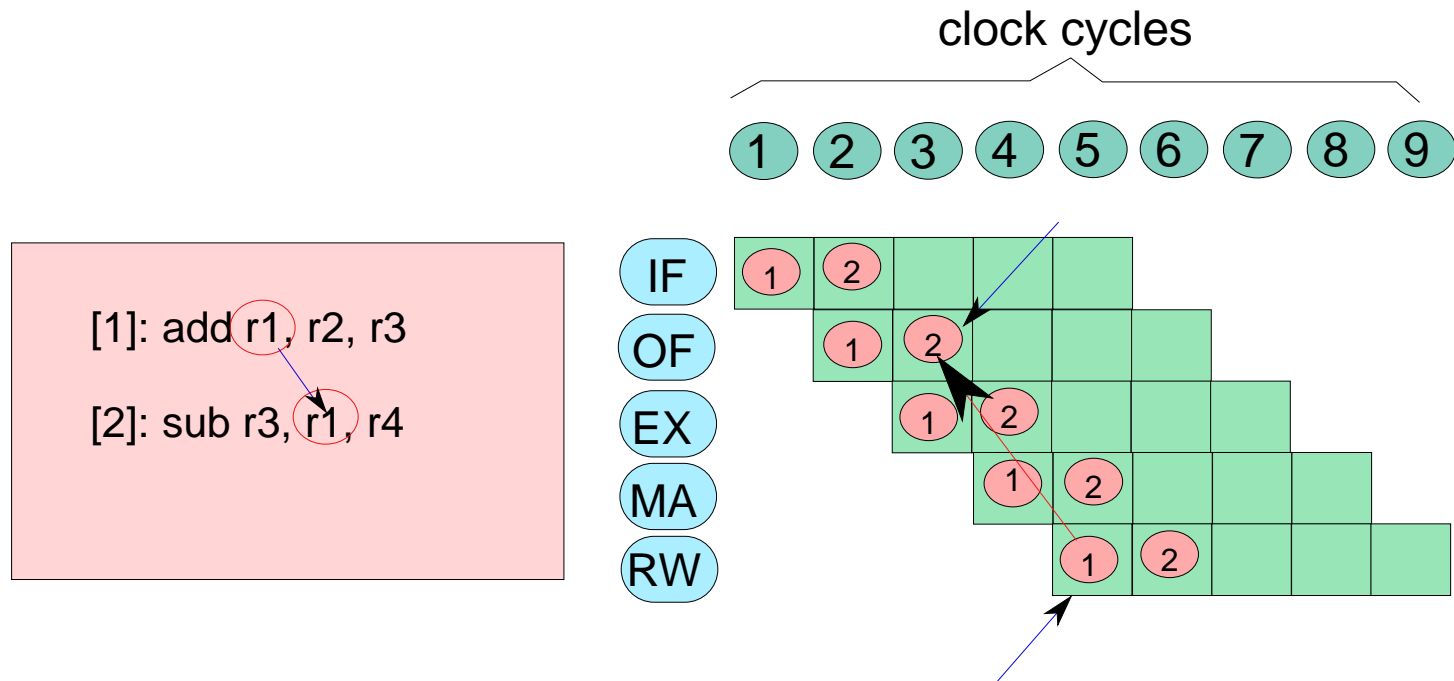
Rules for Constructing a Pipeline Diagram

- * It has 5 **rows**
 - * One for each stage
 - * The rows are **named** : IF, OF, EX, MA, and RW
- * Each **column** represents a clock cycle
- * Each **cell** represents the execution of an instruction in a stage
 - * It is **annotated** with the name(label) of the instruction
- * Instructions proceed from one stage to the next across clock cycles

Example



Data Hazards



* Instruction 2 will read incorrect values !!!

Data Hazard

Definition: A **hazard** is defined as the possibility of erroneous execution of an instruction in a pipeline. A data hazard represents the possibility of erroneous execution because of the unavailability of data, or the availability of **incorrect** data.

- * This situation represents a **data hazard**
- * In specific,
 - * it is a **RAW** (read after write) hazard
- * The earliest we can dispatch instruction 2, is cycle 5

Other Types of Data Hazards

- * Our pipeline is in-order

Definition: In an in-order pipeline (such as ours), a preceding instruction is always ahead of a succeeding instruction in the pipeline. Modern processors however use out-of-order pipelines that break this rule. It is possible for later instructions to execute before earlier instructions.

- * We will only have RAW hazards in our pipeline.
- * Out-of-order pipelines can have WAR and WAW hazards

WAW Hazards

```
[1]: add r1, r2, r3  
[2]: sub r1, r4, r3
```

- * Instruction [2] cannot write the value of r1, before instruction [1] writes to it, will lead to a WAW hazard

WAR Hazards

```
[1]: add r1, r2, r3  
[2]: add r2, r5, r6
```

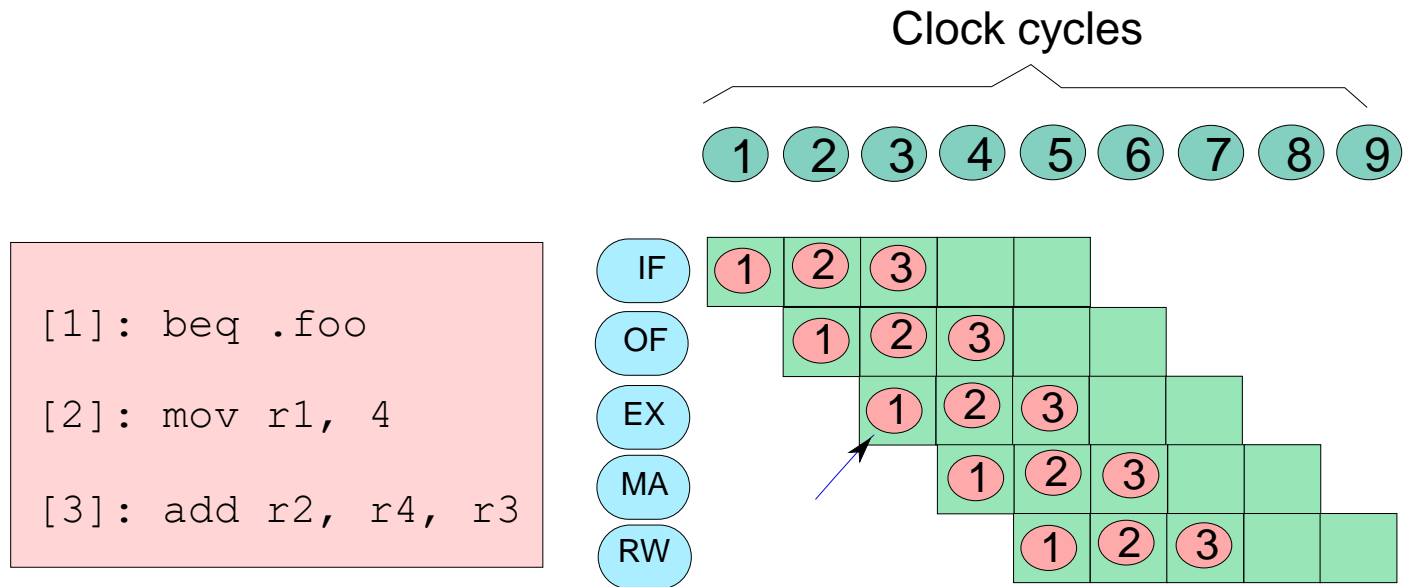
- * **Instruction** [2] cannot **write** the value of r2, before **instruction** [1] reads it → will lead to a **WAR** hazard

Control Hazards

```
[1]: beq .foo  
[2]: mov r1, 4  
[3]: add r2, r4, r3  
...  
...  
.foo:  
[100]: add r4, r1, r2
```

- * If the branch is **taken**, **instructions** [2] and [3], might get fetched, incorrectly

Control Hazard – Pipeline Diagram



- * The two **instructions** fetched immediately after a branch instruction might have been fetched **incorrectly**.

Control Hazards

- * The two **instructions** fetched immediately after a branch instruction might have been fetched **incorrectly**.
- * These instructions are said to be on the **wrong path**
- * A **control hazard** represents the **possibility** of **erroneous** execution in a **pipeline** because instructions in the **wrong path** of a **branch** can possibly get **executed** and save their results in memory, or in the register file

Structural Hazards

- * A **structural hazard** may occur when two instructions have a conflict on the same set of resources in a cycle
- * Example :
 - * Assume that we have an add instruction that can read one operand from memory
 - * `add r1, r2, 10[r3]`

Structural Hazards - II

```
[1]: st r4, 20[r5]  
[2]: sub r8, r9, r10  
[3]: add r1, r2, 10[r3]
```

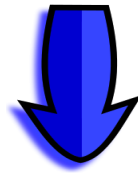
- * This code will have a structural hazard
 - * [3] tries to read 10[r3] (MA unit) in cycle 4
 - * [1] tries to write to 20[r5] (MA unit) in cycle 4
- * Does not happen in in-order pipeline

Software Solution for Data Hazard

- Goal: There have to be at least 3 instructions between a producer and a consumer instruction.
- Solutions:
 1. Insert `nop` instructions
 2. Reorder `code`

Software Solution for Data Hazard

```
[1]: add r1, r2, r3  
[2]: sub r3, r1, r4
```

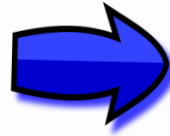


```
[1]: add r1, r2, r3  
[2]: nop  
[3]: nop  
[4]: nop  
[5]: sub r3, r1, r4
```

Software Solution for Data Hazard

Code Reordering

```
add r1, r2, r3
add r4, r1, 3
add r8, r5, r6
add r9, r8, r5
add r10, r11, r12
add r13, r10, 2
```



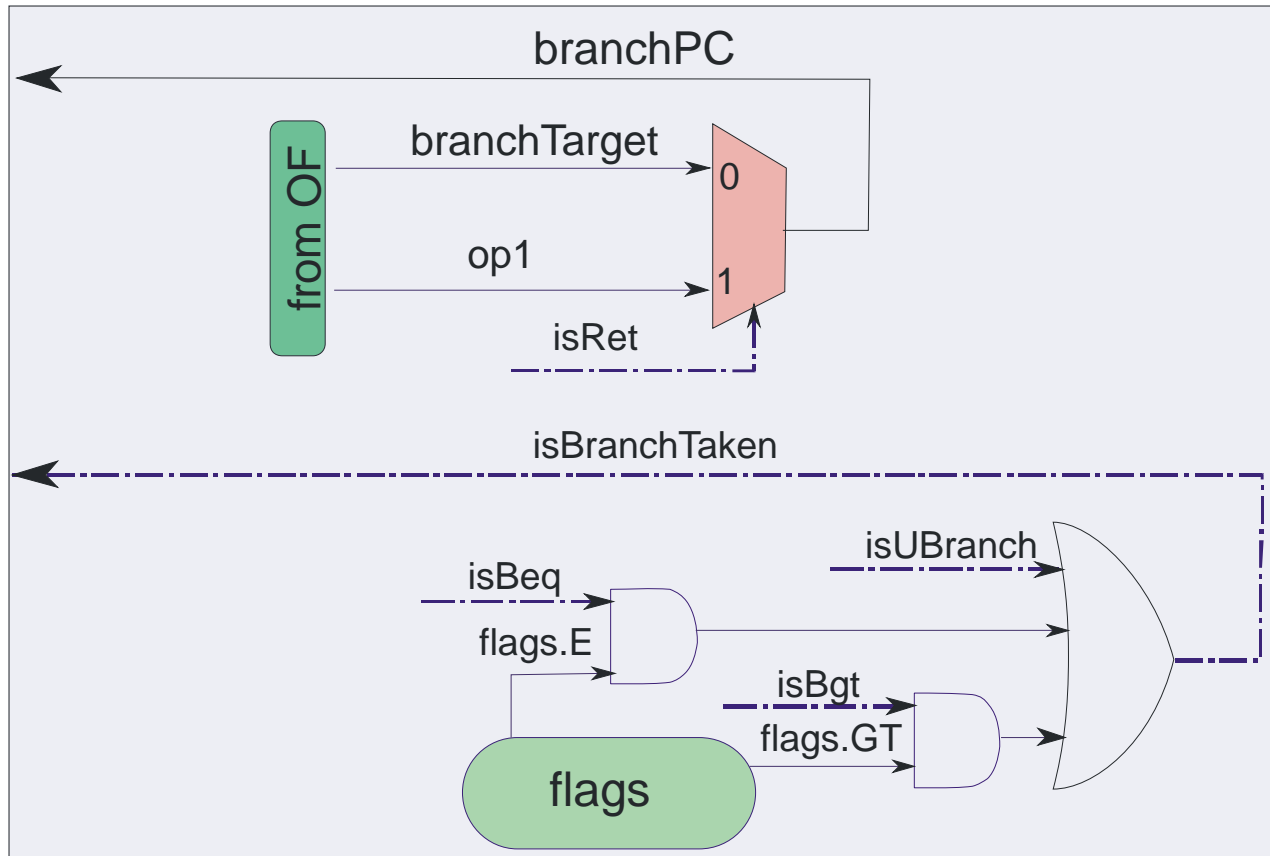
```
add r1, r2, r3
add r8, r5, r6
add r10, r11, r12
nop
add r4, r1, 3
add r9, r8, r5
add r13, r10, 2
```

There have to be at least 3 instructions between a producer and a consumer instruction.

Control Hazards

- * **Trivial Solution** : Add two nop instructions after every branch
- * **Better solution** : based on following observations:
 - * We get the branch outcome and branch target at the end of EX stage.

EX Stage – Branch Unit



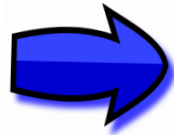
Generates the isBranchTaken Signal

Solution for Control Hazards

- * If branch is at PC of Q, then, we normally fetch Q+4 and Q+8. If branch not taken, these instructions are on right path. Otherwise, they are on wrong path.
- * We find two instructions that get executed regardless of outcome of branch.
- * These instructions are said to be in the **delay slots**

Example with 2 Delay Slots

```
add r1, r2, r3
add r4, r5, r6
b .foo
add r8, r9, r10
```

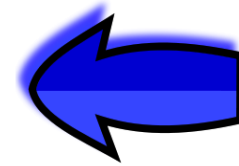


```
b .foo
add r1, r2, r3
add r4, r5, r6
add r8, r9, r10
```

- * The **compiler** transfers **instructions** before the branch to the **delay slots**.
- * If it cannot find 2 valid **instructions**, it inserts **nops**.

Outline

- * Pipeline Hazards
- * Pipeline with Interlocks



Why interlocks ?

- * We cannot always **trust** the **compiler** to do a good job, or even introduce **nop** instructions correctly.
- * **Compilers** now need to be tailored to specific hardware.
- * We should ideally not expose the details of the **pipeline** to the **compiler** (might be confidential also)
- * Hardware mechanism to enforce correctness → **interlock**

Two kinds of Interlocks

- * Data-Lock

- * Do not allow a consumer instruction to move beyond the OF stage till it has read the correct values. Implication : Stall the IF and OF stages.

- * Branch-Lock

- * We never execute instructions in the wrong path.
- * The hardware needs to ensure both these conditions.

Conceptual Look at Pipeline with Interlocks

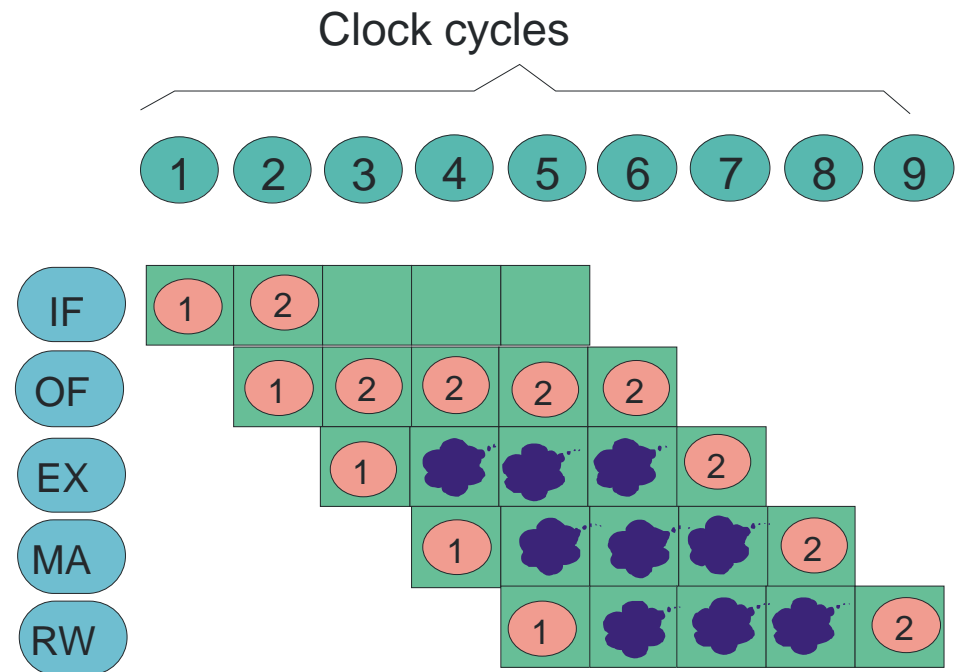
```
[1]: add r1, r2, r3  
[2]: sub r4, r1, r2
```

- * We have a **RAW hazard**
- * We need to **stall**, instruction [2] at the OF stage for 3 cycles.
- * We need to keep sending **nop** instructions to the **EX stage** during these 3 cycles

Example



[1]: add r1, r2, r3
[2]: sub r4, r1, r2



A Pipeline Bubble

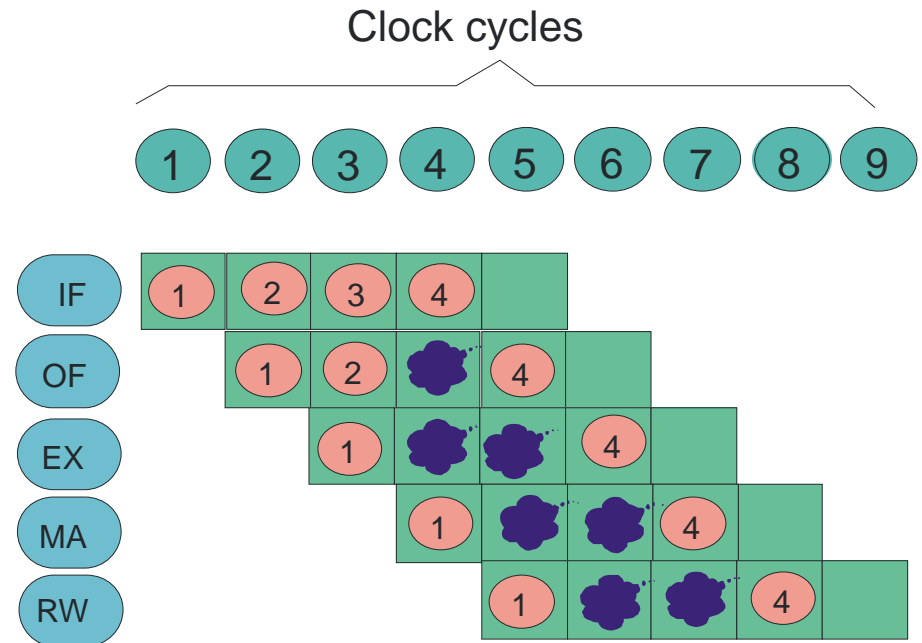
- * A **pipeline bubble** is inserted into a stage, when the **previous stage** needs to be stalled
- * It is a **nop** instruction
- * To insert a **bubble**
 - * Create a **nop** instruction packet
 - * OR, Mark a designated **bubble bit** to 1

Bubbles in the Case of a Branch Instruction



bubble

```
[1]: beq. foo
[2]: add r1, r2, r3
[3]: sub r4, r5, r6
....
....
.foo:
[4]: add r8, r9, r10
```



Control Hazards and Bubbles

- * We know that an instruction is a **branch** in the OF stage
- * When it reaches the EX stage and the branch is taken, let us **convert** the instructions in the IF, and OF stages to **bubbles**
- * Ensures the **branch-lock** condition

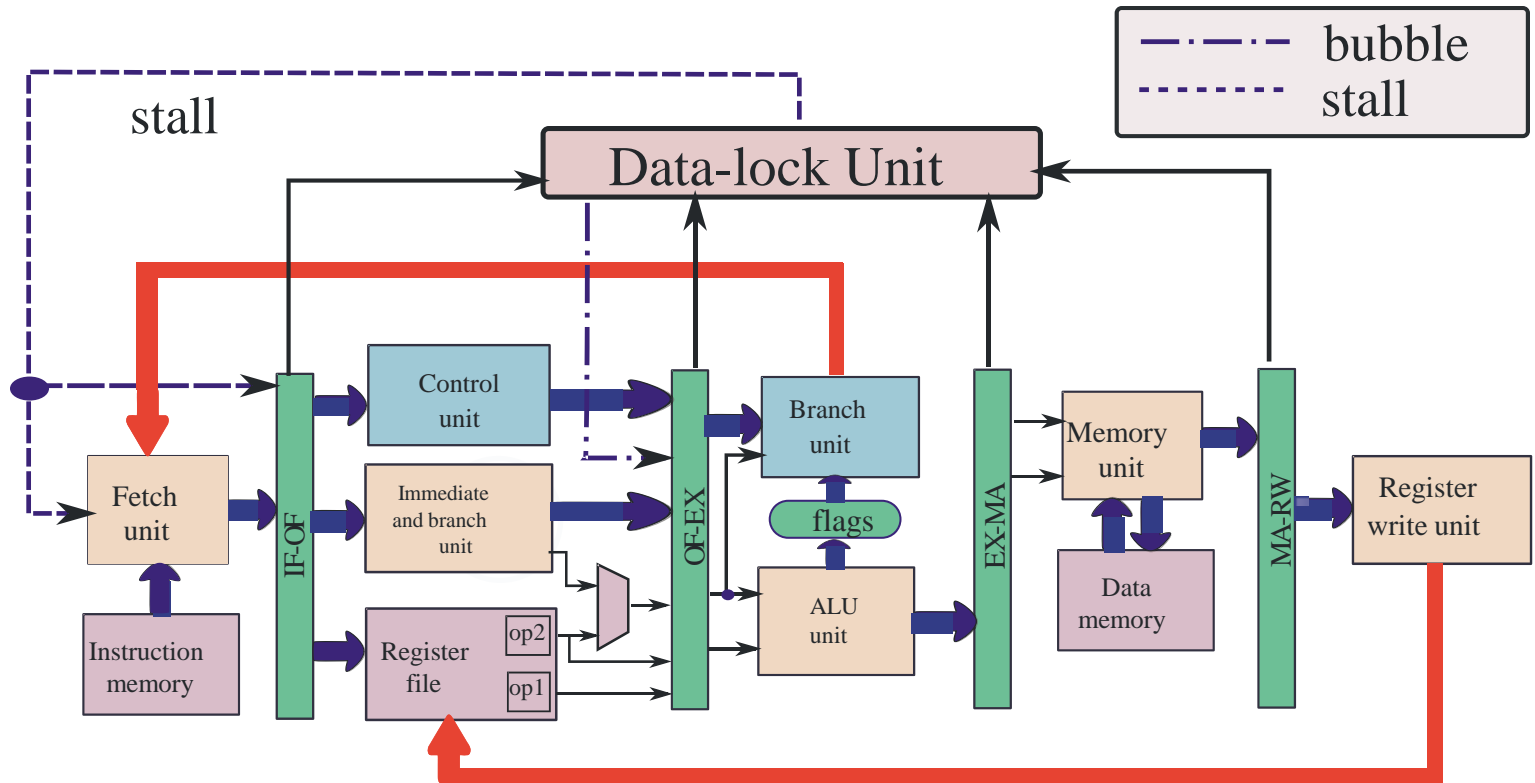
Ensuring the Data-Lock Condition

- * When an **instruction** reaches the **OF** stage, check if it has a **conflict** with any of the instructions in the **EX, MA, and RW** stages
- * If there is **no conflict**, **nothing** needs to be done
- * Otherwise, **stall** the **pipeline** (IF and OF stages only)

How to Stall a Pipeline ?

- * Disable the **write functionality** of :
 - * The **IF-OF** register
 - * and the **Program Counter (PC)**
- * To insert a **bubble**
 - * Write a **bubble** (**nop** instruction) into the **OF-EX** register

Data Path with Interlocks (Data-Lock)



Ensuring the Branch-Lock Condition

- * If the **branch instruction** in the **EX** stage is **taken**, then **invalidate** the instructions in the IF and OF stages. Start **fetching** from the **branch target**.
- * Otherwise, do not take **any special action**

Data Path with Interlocks

