# ICS 233
# COMPUTER ARCHITECTURE

## Pipelined Processor Design

## Instruction Pipeline Hazards

## Lecture 26



## Instruction Pipeline Hazards

***There are situations in pipelining when the next instruction cannot execute in the following clock cycle.***

***These events are called Pipeline Hazards***

## Instruction Pipeline Hazards

- **Three major causes for pipeline breakdown or hesitation of an instruction pipeline**

- **Structural Hazard due to Resource conflicts**

- **Control Hazard due to Procedural dependencies (caused notably by branch instructions)**

- **Data Hazard due to Data dependencies**



## Performance of Pipelines with stalls

**A stall causes the pipeline performance to degrade from the ideal performance**

**Speedup from pipelining = Average instruction time unpipelined / Average instruction time pipelined**

**= CPI unpipelined x Clock cycle unpipelined / CPI pipelined x Clock cycle pipelined**

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

Pipelining can be thought of as decreasing the CPI or the clock cycle time. The ideal CPI on a pipelined processor is almost always 1.

The pipelined CPI can be computed as :

CPI pipelined = Ideal CPI + Pipeline stall clock cycles per instruction
= 1 + Pipeline stall clock cycles per instruction

## Performance of Pipelines with stalls

Ignoring the cycle time overhead of pipelining and assuming that the stages are perfectly balanced, then the cycle time of the two processors can be equal, leading to

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

One important simple case is where all instructions take the same number of cycles, which must also equal the number of pipeline stages (also called the depth of the pipeline)

In this case, unpipelined CPI s equal to the depth of the pipeline.

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls, this leads to the intuitive result that pipelining can improve performance by the depth of the pipeline.



# ICS 233
# COMPUTER ARCHITECTURE

## Structural Hazards

## Structural Hazards

- **Structural Hazard are caused by Resource conflicts**

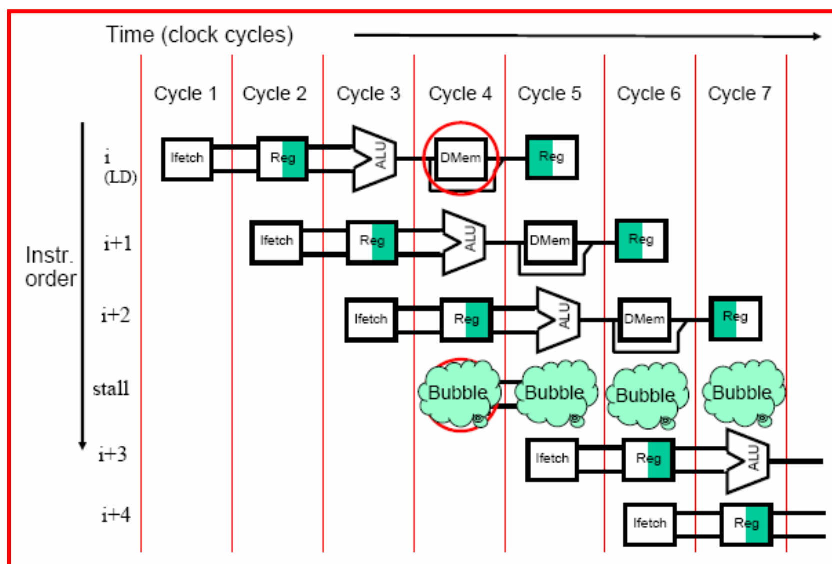
- Resource conflicts occur when a resource such as memory or a functional unit is required by more than one instruction at the same time in the pipeline.

- A notable resource conflict is the main memory of the system.

- For memories with only one read/write port, only one access can be made at any instant.



## One Memory Port Structural Hazard



## Structural Hazards

- ❑ **Resource conflicts can be resolved by**

- ➢ **Duplicating the resources**

**Example :**

- In a 5-stage pipeline, two stages might access the memory at the same instant. This can be achieved by having two memory module ports
- Three arithmetic instructions can be executed simultaneously by providing three functional units in the execute stage of the pipeline

- ➢ **Expensive**



## Structural Hazards

- ❑ **Resource conflicts can be resolved by**

- ➢ **Allowing only one instruction (i.e., only one pipeline stage) in the pipeline to access the data memory at any time.**

**i.e. simultaneous access to the memory by more than one stage in the pipeline is not allowed**

## Structural Hazards

- ❑ **Resource conflicts can be resolved by**

- ➢ **Fetching more than one instruction at a time using multiple memory modules or using memory interleaving**

- ➢ **Provide separate program and data cache memories**



## Pipeline Structural Hazards - Summary

❑ **Structural Hazards**

- Occur when two or more instructions need the same resource

- Common methods for eliminating structural hazards are:
  - ➢ Duplicate resources
  - ➢ Pipeline the resource
  - ➢ Reorder the instructions

## Pipeline Structural Hazards

- Two machines
  - Machine A: Dual ported memory
  - Machine B: Single ported memory, but its pipelined implementation has a clock rate that is 1.05 times faster
  - Ideal CPI = 1 for both
  - Loads are 40% of instructions executed (cause stalls in machine B)

Which is faster?

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

$Speedup_A$ = ((1 x Pipeline Depth)/(1 + 0))x 1 = pipeline depth

$Speedup_B$ = ((1 x Pipeline Depth)/(1 + 0.4))x 1.05 = 0.75 x pipeline depth

Machine A is 1.33 times faster



# ICS 233
# COMPUTER ARCHITECTURE

## Data Hazards

## Data Hazards

**Data dependencies**

- A data dependency describes the normal situation that the data that instructions use depend upon the data created by other instructions .
- Instructions depend upon each other in the program and must be executed in a specific order.
- There are three types of data dependency between instructions :
  - **True data dependency**
  - **Anti-dependency**
  - **Output dependency**



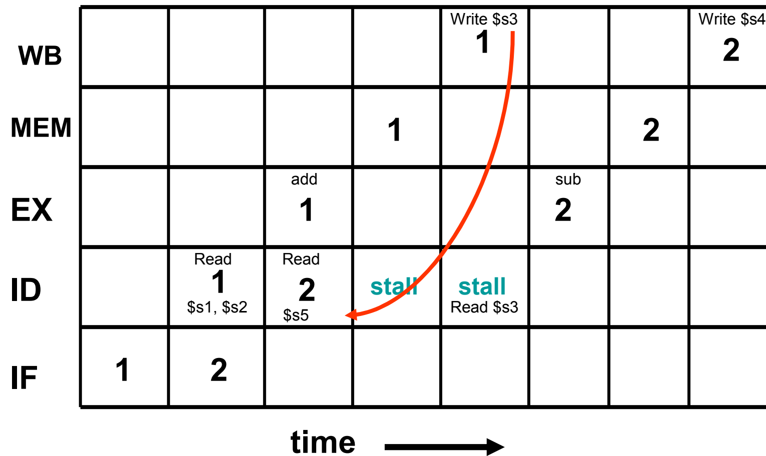
## True Data Dependency

- Consider the code sequence:

```
(1) add  $s3, $s2, $s1        ; $s3 = $s2 + $s1
(2) sub  $s4, $s3, $s5        ; $s4 = $s3 – $s5
```

- It would be incorrect to begin reading $s3 in instruction 2 before instruction 1 has produced its new value for $s3.
- Hence there is a “data” dependency between instructions 1 and 2.
- This dependency is called a **true data dependency**.
- A true data dependency occurs when the value produced by an instruction is required by a subsequent instruction.
- **True data dependencies are also called Read-After-Write (RAW) hazards because we are reading a value after writing to it.**
- True dependencies are also sometimes known as **flow dependencies** because the dependency is due to the flow of data in the program.

## True Data Dependency




**Space-Time Diagram**



## True Data Dependency

**sub $2, $1, $3**

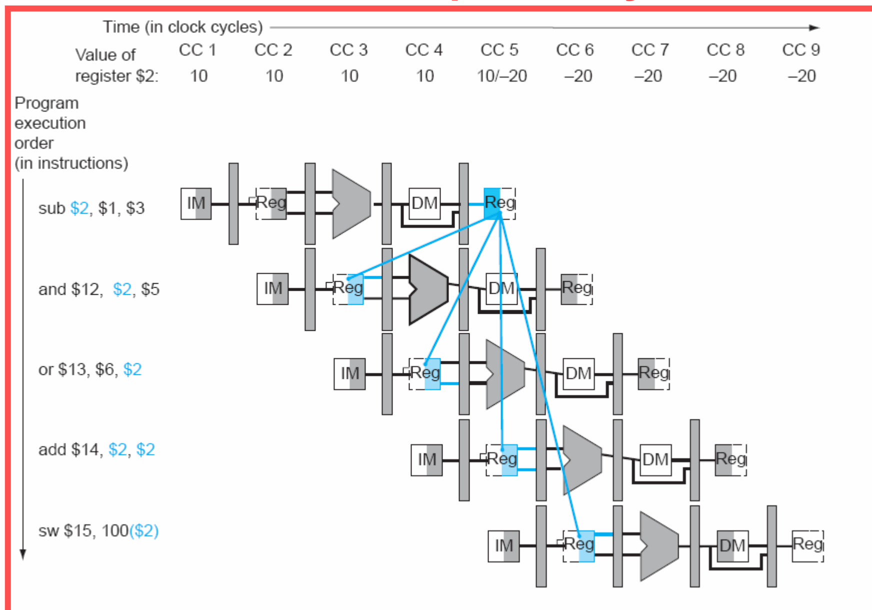
**and $12, $2, $5**

**or $13, $6, $2**

**add $14, $2, $2**

**sw $15, 100($2)**

## True Data Dependency






## Anti-dependency

- Consider the code sequence:

| | | |
|---|---|---|
| (1) | **add $s4, $s2, $s1** | **; $s3 = $s2 + $s1** |
| (2) | **sub $s2, $s3, $s5** | **; $s2 = $s3 – $s5** |

- Instruction 2 must not produce its result in $s2 before instruction 1 reads $s2, otherwise instruction 1 would use the value produced by instruction 2 rather than the previous value of $s2.
- This type of dependency is called an **Anti-dependency (also called a Write-After-Read (WAR) hazard)** and occurs when an instruction writes to a location which has been read by a previous instruction.

## Anti-dependency

- In most pipelines, reading occurs in a stage before writing, so normally if the instructions remain in program order in the pipeline, an anti-dependency would not be a problem.
- It becomes a problem if the pipeline structure is such that writing can occur before reading in the pipeline, or the instructions are not processed in program order within the pipeline.
- The latter situation could occur for example if instructions are allowed to overtake stalled instructions in the pipeline.
- For example, suppose instruction 1 was stalled because of a resource conflict.
- Instruction 2 could be allowed to overtake instruction 1 in the pipeline, but then we could have an anti-dependency (write-after-read hazard).



## Output Dependency

- Consider the code sequence:

```
(1) add  $s3, $s2, $s1     ;$s3 = $s2 + $s1
(2) sub  $s4, $s3, $s5     ;$s4 = $s3 – $s5
(3) add  $s3,  $s2, $s5    ;$s3 = $s2 + $s5
```

- Instruction 1 must produce its result in $s3 before instruction 3 produces its result in $s3 otherwise instruction 2 might use the wrong value of $s3.
- This type of dependency is called an **Output dependency (also called a Write After Write (WAW) hazard)** and occurs when a location is written to by two instructions.
- Again the dependency would not be significant if all instructions write at the same time in the pipeline and instructions are processed in program order.
- Actually output dependencies are a form of resource conflict, because the register in question, $s3, is accessed by two instructions.
- The register is being reused and consequently, the use of another register in the instruction would eliminate the potential problem.

## Detecting Data Hazards

- **When data dependency occurs, there are two possible strategies :**

  - Detect the data dependencies and then hold up the pipeline completely until the dependencies have been resolved

  - Allow all instructions to be fetched into the pipeline but only allow independent instructions to proceed to their completion, and delay instructions which are dependent upon other, not yet executed, instructions until these instructions are executed.



## Detecting Data Hazards

- **Data Dependencies can be detected by considering read and write operations on specific locations accessible by the instructions**
- **Read-After-Write (RAW) Hazard**
  - Exists if a read operation occurs before a previous write operation has been completed, and hence the read operation would obtain an incorrect value. (a value not yet updated
- **Write-After-Read (WAR) Hazard**
  - Exists when a write operation occurs before a previous read operation has had time to complete, and again the read operation would obtain an incorrect value ( a prematurely updated value).
- **Write-After-Write (WAW) Hazard**
  - Exists if there are two write operations upon a location such that the second write operation in the pipeline completes before the first. Hence the written value will be altered by the first write operation when it completes.

## Detecting Data Hazards

- **Berstein's Conditions for detecting data hazards**

A potential hazard can be identified between instruction i and instruction j when at least one of the following condition fails.

**For Read-After-Write $O(i) \cap I(j) = \emptyset$**
**For Write-After-Read $I(i) \cap O(j) = \emptyset$**
**For Write-After-Write $O(i) \cap O(j) = \emptyset$**

O(i) indicates the set of output locations altered by instruction i.
I(i) indicates the set of input locations read by instruction i
Ø indicates an empty set.



## Detecting Data Hazards

- **Berstein's Conditions for detecting data hazards**

**Example :**

Consider the following code sequence :

```
1.   add $s3, $s2, $s1   ; $s3=$s2 + $s1
2.   sub $s5, $s4, $s3   ; $s5=$s4-$s3
```

**O(1) = ($s3)** **O(2) = ($s5)**
**I(1) = ($s1, $s2)** **I(2) = ($s4, $s3)**

Read-After-Write hazard since $O(1) \cap I(2) = \{\$s3\}$
No Write-After-Read hazard since $I(1) \cap O(2) = \emptyset$
No Write-After-Write hazard since $O(1) \cap O(2) = \emptyset$
All conditions are satisfied, there are no hazards

## Detecting Data Hazards

- **Example : Using Compiler to detect & resolve Data Hazards**

```
sub  $2, $1, $3
and  $12, $2, $5
or  $13, $6, $2
add  $14, $2, $2
sw  $15, 100($2)
```

- **In this case, the compiler needs to insert two independent instructions between the sub and the and instructions, thereby making the hazard disappear.**
- **When no such independent instructions can be found, the compiler inserts instructions guaranteed to be independent i.e., nop instructions**

```
sub  $2, $1, $3
nop
nop
and  $12, $2, $5
or  $13, $6, $2
add  $14, $2, $2
sw  $15, 100($2)
```

Although this code works properly for this pipeline, these two nops occupy 2 clock cycles that do no useful work.