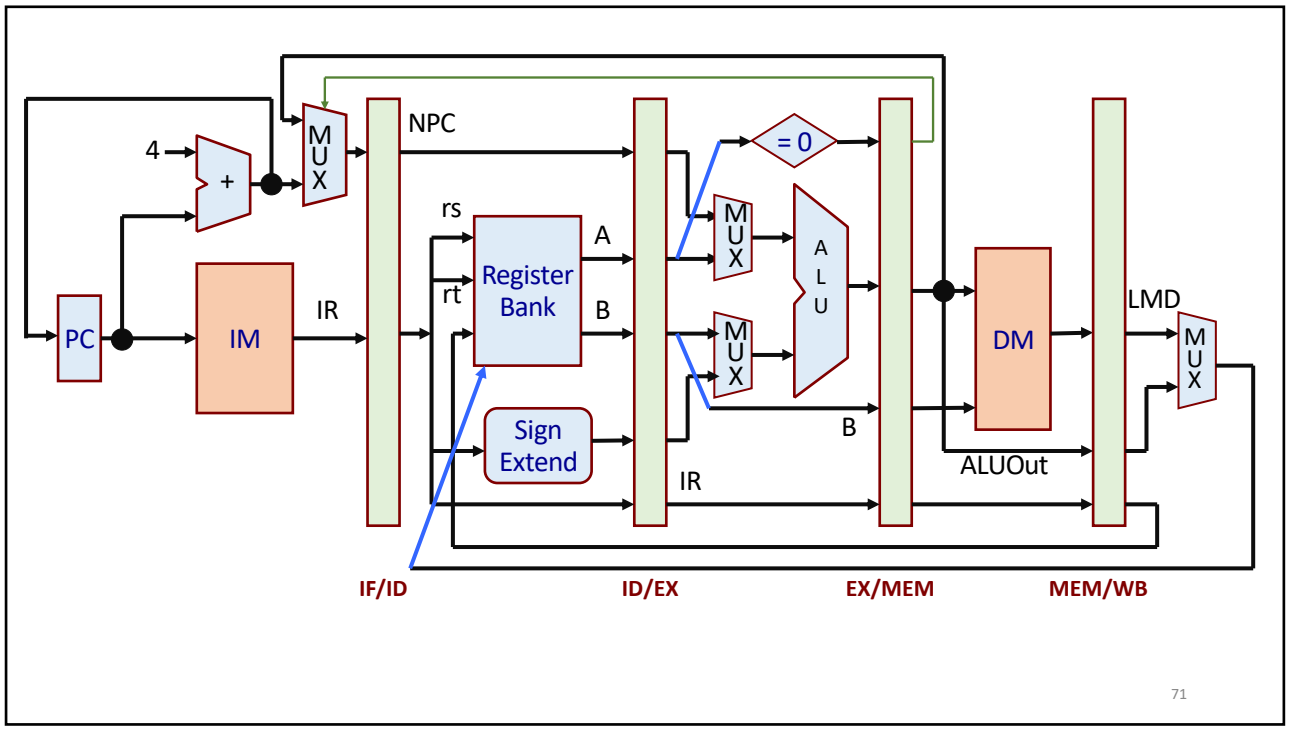



71

# Pipeline Hazards

72

## Introduction

- **Pipeline Hazards:**
  - Ideally, an instruction pipeline should complete the execution of an instruction every clock cycle.
  - Hazards refer to situations that prevent a pipeline from operating at its maximum possible clock speed.
    - Prevents some instructions from executing during its designated clock cycle.
- Three types of hazards:
  *a)* *Structural hazard*: Arise due to resource conflicts.
  *b)* *Data hazard*: Arise due to data dependencies between instructions.
  *c)* *Control hazard*: Arise due to branch and other instructions that change the PC.

73

73

## What happens when hazards appear?

- We can use special hardware and control circuits to avoid the conflict that arise due to hazard.
- As an alternative, we can insert *stall cycles* in the pipeline.
  - When an instruction is stalled, all instructions that *follow* also get stalled.
  - Number of stall cycles depends on the criticality of the hazard.
  - Instructions *before* the stalled instruction can continue, but no new instructions are fetched during the duration of the stall.
- In general, hazards result in *performance degradation*.

74

74

## Calculation of Performance Degradation

$$\text{Pipeline speedup} = \frac{XT_{nopipe}}{XT_{pipe}} = \frac{CPI_{nopipe} \times C_{nopipe}}{CPI_{pipe} \times C_{pipe}} = \frac{C_{nopipe}}{C_{pipe}} \times \frac{CPI_{nopipe}}{CPI_{pipe}}$$

- We can treat pipelining as a way to reduce the CPI, or reduce C.
  - Let us consider that we are decreasing the CPI.
- The ideal CPI of an instruction pipeline can be written as:

$$\text{Ideal CPI} = \frac{CPI_{nopipe}}{\text{Pipeline Depth}} \Rightarrow \text{Speedup} = \frac{C_{nopipe}}{C_{pipe}} \times \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{CPI_{pipe}}$$

75

75

- If we restrict ourselves to stall cycles only, we can write:

  $CPI_{pipe}$ = Ideal CPI + (Pipeline Stall Cycles per Instruction)

- We can thus write:

$$\text{Speedup} = \frac{C_{nopipe}}{C_{pipe}} \times \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles / instruction}}$$

- If we ignore the increase in clock cycle time due to pipelining (i.e. $C_{nopipe} = C_{pipe}$)

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles / instruction}}$$

76

76

## (a) Structural Hazards

- They arise due to resource conflicts when the hardware cannot support overlapped execution under all possible scenarios.
- Examples:
  - Single memory (cache) to store instructions and data → while an instruction is being fetched some other instruction is trying to read or write data.
  - An instruction is trying to read data from the register bank (in ID stage), while some other instruction is trying to write into a register (in WB stage).
  - Some functional unit (like floating-point ADD or MULTIPLY) is not fully pipelined.
    - A sequence of instructions that try to use the same functional unit will result in stalls.

77

77

- Illustration:
  - Structural hazard in a single-port memory system, which stores both instructions and data.

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| **LW R1,10(R2)** | IF | ID | EX | MEM | WB | | | |
| **ADD R3,R4,R5** | | IF | ID | EX | MEM | WB | | |
| **SUB R10,R2,R9** | | | IF | ID | EX | MEM | WB | |
| **AND R5,R7,R7** | | | | STALL | IF | ID | EX | MEM |
| **ADD R2,R1,R5** | | | | | STALL | IF | ID | EX |

**Insert stall cycle to eliminate the structural hazard.**

78

78

## Example 2

- Consider an instruction pipeline with the following features:
  - Data references constitute 35% of the instructions.
  - Ideal CPI ignoring structural hazard is 1.3.

How much faster is the ideal machine without the memory structural hazard, versus the machine with the hazard?

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles / instruction}}$$

$$\text{Speedup}_{ideal} = \frac{1.3 \times \text{Pipeline Depth}}{1.3 + 0}$$

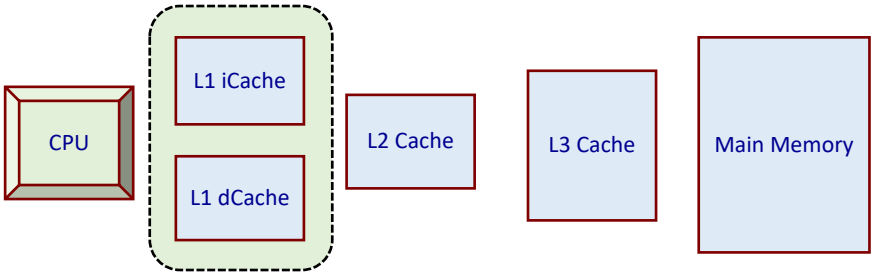
$$\text{Speedup}_{real} = \frac{1.3 \times \text{Pipeline Depth}}{1.3 + 0.35 \times 1}$$

$$\frac{\text{Speedup}_{ideal}}{\text{Speedup}_{real}} = \frac{1.65}{1.3} = 1.27$$

79

79

- Why structural hazards appear in real machines?
  a) To reduce the cost of implementation.
  b) Pipelining all the functional units may be too costly.
  c) If structural hazards are not frequent, it may not be worth the effort and cost to avoid it.
- Memory access structural hazard is quite frequent.
  - Makes use of separate instruction and data caches in the first level.

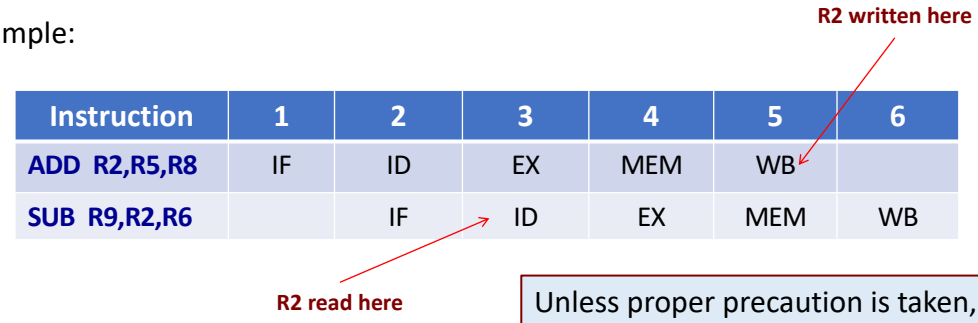



80

80

## (b) Data Hazards

- Data hazards occur due to data dependencies between instructions that are in various stages of execution in the pipeline.
- Example:




| Instruction | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| ADD R2,R5,R8 | IF | ID | EX | MEM | WB | |
| SUB R9,R2,R6 | | IF | ID | EX | MEM | WB |

Unless proper precaution is taken, the SUB instruction can fetch the wrong value of R2.

81

81

- Naïve solution by inserting stall cycles:
  - After the SUB instruction is decoded and the control unit determines that there is a data dependency, it can insert stall cycles and re-execute the ID stage again later.
  - 3 clock cycles are wasted.

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| ADD R2,R5,R8 | IF | ID | EX | MEM | WB | | | |
| SUB R9,R2,R6 | | IF | ID | STALL | STALL | ID | EX | MEM |
| Instr. (i+2) | | | IF | STALL | STALL | IF | ID | EX |
| Instr. (i+3) | | | | STALL | STALL | | IF | ID |
| Instr. (i+4) | | | | STALL | STALL | | | IF |

82

82

- How to reduce the number of stall cycles?
  - We shall explore two methods that can be used together.
    1) **Data forwarding / bypassing**: By using additional hardware consisting of multiplexers, the data required can be forwarded as soon as they are computed, instead of waiting for the result to be written into the register.
    2) **Concurrent register access**: By splitting a clock cycle into two halves, register read and write can be carried out in the two halves of a clock cycle (register write in first half, and register read in second half).

83

83

## (1) Reducing Data Hazards using Bypassing

- Basic idea:
  - The result computed by the previous instruction(s) is stored in some register within the data path (e.g. ALUOut).
  - Take the value directly from the register and forward it to the instruction requiring the result.
- What is required?
  - Additional data transfer paths are to be added in the data path.
  - Requires multiplexers to select these additional paths.
  - The control unit identifies data dependencies and selects the multiplexers in a suitable way.

84

84

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| ADD R4, R5, R6 | IF | ID | EX | MEM | WB | | | | |
| SUB R3, R4, R8 | | IF | ID | EX | MEM | WB | | | |
| ADD R7, R2, R4 | | | IF | ID | EX | MEM | WB | | |
| AND R9, R4, R10 | | | | IF | ID | EX | MEM | WB | |
| OR R11, R4, R5 | | | | | IF | ID | EX | MEM | WB |

- The first instruction computes *R4*, which is required by all the subsequent four instructions.
- The dependencies are depicted by red arrows (result written in *WB*, operands read in *ID*).
- The last instruction (OR) is not affected by the data dependency.

85

85

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| ADD R4, R5, R6 | IF | ID | EX | MEM | WB | | | | |
| SUB R3, R4, R8 | | IF | ID | EX | MEM | WB | | | |
| ADD R7, R2, R4 | | | IF | ID | EX | MEM | WB | | |
| AND R9, R4, R10 | | | | IF | ID | EX | MEM | WB | |
| OR R11, R4, R5 | | | | | IF | ID | EX | MEM | WB |

**Data forwarding requirements:**

- The first instruction (ADD) finishes computing the result at the end of *EX* (shown in RED), but is supposed to write into *R4* only in *WB*.
- We need to forward the result directly from the output of the ALU (in *EX* stage) to the appropriate ALU input registers of the following instructions.
  - To be used in the respective *EX* stages, shown by the arrow.

86

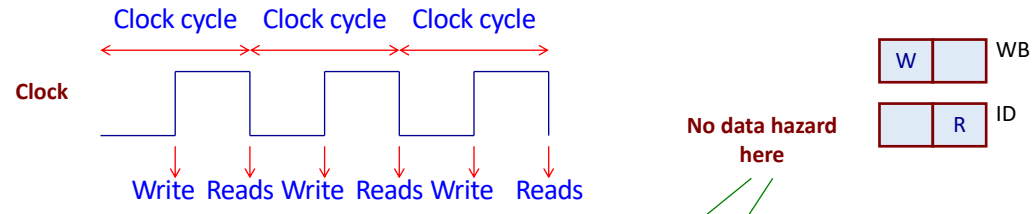
86

## (2) Reducing Data Hazard by Splitting Register Read/Write

- The data forwarding concept as discussed can solve the data hazards between ALU instructions.
- It is desirable to reduce the number of instructions to be forwarded, since each level requires special hardware.
  - In the forwarding scheme as discussed, we need to forward *3 instructions*.
  - We can *reduce this number to 2* by avoiding the conflict between the first and the fourth (AND) instruction, where WB and ID are accessed in the same cycle.
  - We perform register write (in WB) during the first half of the clock cycle, and the register reads (in ID) during the second half of the clock cycle.
    - The data hazard between the first and fourth instructions get eliminated.

87

87



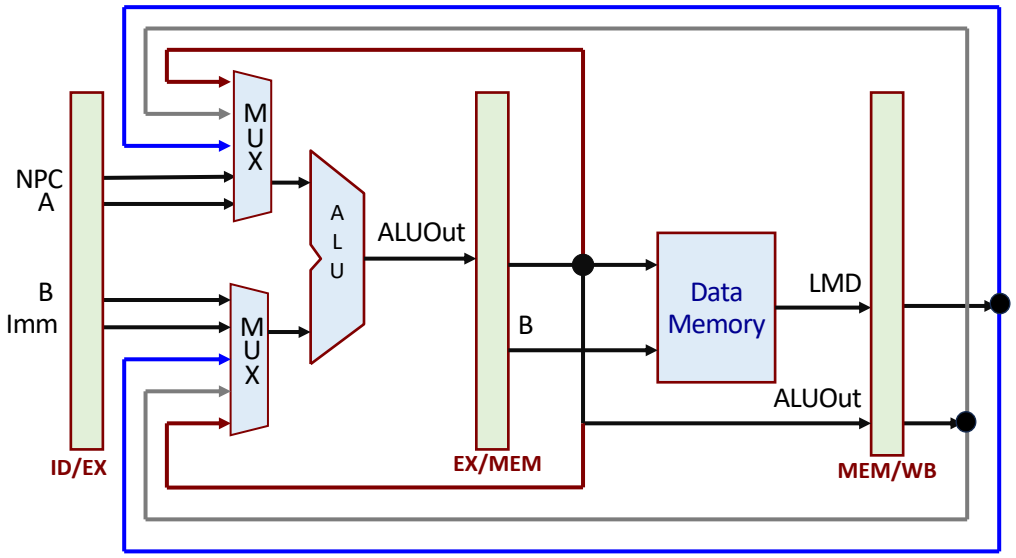

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **ADD R4, R5, R6** | IF | ID | EX | MEM | WB | | | | |
| **SUB R3, R4, R8** | | IF | ID | EX | MEM | WB | | | |
| **ADD R7, R2, R4** | | | IF | ID | EX | MEM | WB | | |
| **AND R9, R4, R10** | | | | IF | ID | EX | MEM | WB | |
| **OR R11, R4, R5** | | | | | IF | ID | EX | MEM | WB |

88

88

## Data Forwarding Hardware

Only the relevant parts of the MIPS32 pipeline data path is shown.




89

89

## Data Hazard while Accessing Memory

- In MIPS32 pipeline, memory references are always kept in order, and so data hazards between memory references never occur.
  - Cache misses can result in pipeline stalls.

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| SW R2, 100(R5) | IF | ID | EX | MEM | WB | |
| LW R10, 100(R5) | | IF | ID | EX | MEM | WB |

Accessed in order; no hazard

90

90

- A LOAD instruction followed by the use of the loaded data.
  - Example of a data hazard that requires *unavoidable pipeline stalls*.

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LW R4, 100(R5) | IF | ID | EX | MEM | WB | | | |
| SUB R3, R4, R8 | | IF | ID | EX | MEM | WB | | |
| ADD R7, R2, R4 | | | IF | ID | EX | MEM | WB | |
| AND R9, R4, R10 | | | | IF | ID | EX | MEM | WB |

**ALU wants to use the loaded data; Forwarding cannot solve**

**Loaded data available here**

**Forwarding can solve**

91

91

- What is the solution?
  - As we have seen the hazard cannot be eliminated by forwarding alone.
  - Common solution is to use a hardware addition called *pipeline interlock*.
    - The hardware detects the hazard and stalls the pipeline until the hazard is cleared.
  - The pipeline with stall is shown.

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| LW R4, 100(R5) | IF | ID | EX | MEM | WB | | | | |
| SUB R3, R4, R8 | | IF | ID | Stall | EX | MEM | WB | | |
| ADD R7, R2, R4 | | | IF | Stall | ID | EX | MEM | WB | |
| AND R9, R4, R10 | | | | Stall | IF | ID | EX | MEM | WB |

92

92

- A terminology:
  - **Instruction Issue**: This is the process of allowing an instruction to move from the ID stage to the EX stage.
  - In the MIPS32 pipeline, all possible data hazards can be checked in the ID stage itself.
    - If a data hazard (LOAD followed by use) exists, the instruction is *stalled* before it is *issued*.

93

93

- A common example:

  A = B + C

- Pipelined execution of the corresponding MIPS32 code is shown.

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **LW R1, B** | IF | ID | EX | MEM | WB | | | | |
| **LW R2, C** | | IF | ID | EX | MEM | WB | | | |
| **ADD R5, R1, R2** | | | IF | ID | Stall | EX | MEM | WB | |
| **SW R5, A** | | | | IF | Stall | ID | EX | MEM | WB |

***Instruction Scheduling or Pipeline Scheduling***:

- Compiler tries to avoid generating code with a LOAD followed by an immediate use.

94

94

## Example 3

A C code segment:

```
x = a – b;
y = c + d;
```

**MIPS32 code:**

```
LW   R1, a
LW   R2, b
SUB  R8, R1, R2
SW   R8, x
LW   R1, c
LW   R2, d
ADD  R9, R1, R2
SW   R9, y
```

Two load interlocks

**Scheduled MIPS32 code:**

```
LW   R1, a
LW   R2, b
LW   R3, c
SUB  R8, R1, R2
LW   R4, d
SW   R8, x
ADD  R9, R3, R4
SW   R9, y
```

Both load interlocks are eliminated

95

95

- Some observations:
  - Pipeline scheduling can increase the number of registers required, but results in performance improvement in general.
  - A LOAD instruction requiring that the following instruction do not use the loaded value is called a *delayed load*.
  - A pipeline slot after a LOAD is called the *load delay slot*.
  - If the compiler cannot move some instruction to fill up the delay slot (scheduling), it can insert a NOP (No Operation) instruction.

96

96

## Types of Data Hazards

- Broadly three classes of data hazards are possible:
  - **a)** **Read After Write (RAW)** – Most common scenario.
  - **b)** **Write After Read (WAR)** – This kind of hazard cannot happen in MIPS32 as all reads are early (in ID) and all writes are late (in WB).
  - **c)** **Write After Write (WAW)** – This kind of hazard is possible in pipelines where write can happen in more than one pipeline stages.
- For MIPS32 integer pipeline, only RAW hazard is possible, and only one type (*LOAD followed by immediate use*) can result in performance loss.
  - Compiler tries to eliminate the performance loss using instruction scheduling.

97

97

## (c) Control Hazard

- Control hazards arise because of branch instructions being executed in a pipeline.
  - Can cause greater performance loss than data hazards.
- What happens when a branch is executed?
  - The value of PC may or may not change to something other than $PC_{old} + 4$.
  - If the branch is taken, the PC is normally not updated until the end of MEM.
  - The next instruction can be fetched only after that (3 stall cycles).
  - Actually, by the time we know than an instruction is a branch, the next instruction has already been fetched.
    - We have to *redo the fetch* if it is a branch.

98

98

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **BEQ Label** | IF | ID | EX | MEM | WB | | | | | | |
| **Instr. (i+1)** | | *IF* | Stall | Stall | IF | ID | EX | MEM | WB | | |
| **Instr. (i+2)** | | | Stall | Stall | Stall | IF | ID | EX | MEM | WB | |
| **Instr. (i+3)** | | | | Stall | Stall | Stall | IF | ID | EX | MEM | WB |
| **Instr. (i+4)** | | | | | Stall | Stall | Stall | IF | ID | EX | MEM |
| **Instr. (i+5)** | | | | | | Stall | Stall | Stall | IF | ID | EX |
| **Instr. (i+6)** | | | | | | | Stall | Stall | Stall | IF | ID |

Example:
We have seen that 3 cycles are wasted for every branch.
Assume – 30% branch frequency, $CPI_{ideal} = 1$.

Actual CPI = 0.7 x 1 + 0.3 x 4
= 1.9
*Speed becomes almost half.*

99

99

## How to Reduce Branch Stall Penalty?

- The penalty can be reduced if both the following are achieved:
  a) Determine whether the branch is taken earlier in the pipeline.
  b) Compute the branch target address earlier in the pipeline.
- How to achieve (a) in MIPS32?
  - In MIPS32, the branches require testing a register for zero, or comparing the values of two registers.
  - Possible to complete this decision by the end of the ID cycle where the registers are pre-fetched, by adding special comparison logic.

100

100

- How to achieve (b) in MIPS32?
  - Both the possible PC values can be pre-computed in the ID stage by using a separate adder.
- By the end of the ID cycle, we know whether the branch is taken and also know the branch target address.
  - Requires a single cycle stall during branches.

101

101

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **BEQ Label** | IF | ID | EX | MEM | WB | | | | | | |
| **Instr. (i+1)** | | *IF* | IF | ID | EX | MEM | WB | | | | |
| **Instr. (i+2)** | | | Stall | IF | ID | EX | MEM | WB | | | |
| **Instr. (i+3)** | | | | Stall | IF | ID | EX | MEM | WB | | |
| **Instr. (i+4)** | | | | | Stall | IF | ID | EX | MEM | WB | |
| **Instr. (i+5)** | | | | | | Stall | IF | ID | EX | MEM | WB |

***How to reduce pipeline branch penalty?***

102

102

## Reducing Pipeline Branch Penalties

- Four techniques based on compile-time guesses are discussed.
  a) The simplest scheme is to freeze the pipeline and insert (one) stall cycle.
     - Main advantage is simplicity.
  b) We predict that the branch is *not taken*, and allow the hardware to continue as if the branch was not executed.
     - Must not change the machine state until the branch outcome is finally known.
     - If the prediction is wrong (i.e. branch is taken), we stop the pipeline and restart the fetch from the target address.
     - Illustration on next slide.

103

103

**Not Taken Branch (no penalty)**

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| BEQ Label | IF | ID | EX | MEM | WB | | | | | |
| Instr. (i+1) | | IF | ID | EX | MEM | WB | | | | |
| Instr. (i+2) | | | IF | ID | EX | MEM | WB | | | |
| Instr. (i+3) | | | | IF | ID | EX | MEM | WB | | |
| Instr. (i+4) | | | | | IF | ID | EX | MEM | WB | |

**Taken Branch (1 cycle penalty)**

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| BEQ Label | IF | ID | EX | MEM | WB | | | | | |
| Instr. (i+1) | | *IF* | IF | ID | EX | MEM | WB | | | |
| Instr. (i+2) | | | Stall | IF | ID | EX | MEM | WB | | |
| Instr. (i+3) | | | | Stall | IF | ID | EX | MEM | WB | |
| Instr. (i+4) | | | | | Stall | IF | ID | EX | MEM | WB |

104

104

c) We predict that the branch is *taken*.

- As soon as the branch outcome is decoded and the target address computed, fetching of instructions can commence from the computed target.
- Since in MIPS32 pipeline, we come to know about the branch outcome and target address together (at the end of ID), there is *no advantage* in this approach.
- May be used for complex instruction machines where the branch outcome is known later than the branch target address.

**Taken or Not Taken Branch (1 cycle penalty)**

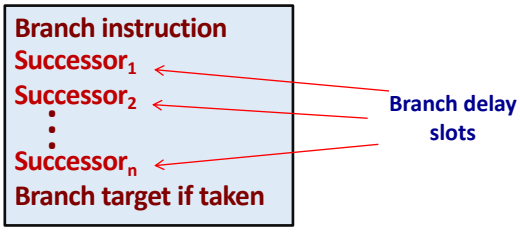
| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **BEQ Label** | IF | ID | EX | MEM | WB | | | | | |
| **Instr. (i+1)** | | *IF* | IF | ID | EX | MEM | WB | | | |
| **Instr. (i+2)** | | | Stall | IF | ID | EX | MEM | WB | | |
| **Instr. (i+3)** | | | | Stall | IF | ID | EX | MEM | WB | |
| **Instr. (i+4)** | | | | | Stall | IF | ID | EX | MEM | WB |

105

105

d) We can use a technique called *delayed branch*, which makes the hardware simple but puts more responsibility on the compiler.

- If a branch instruction incurs a penalty of *n* stall cycles, the execution cycle of a branch instruction is defined as follows:



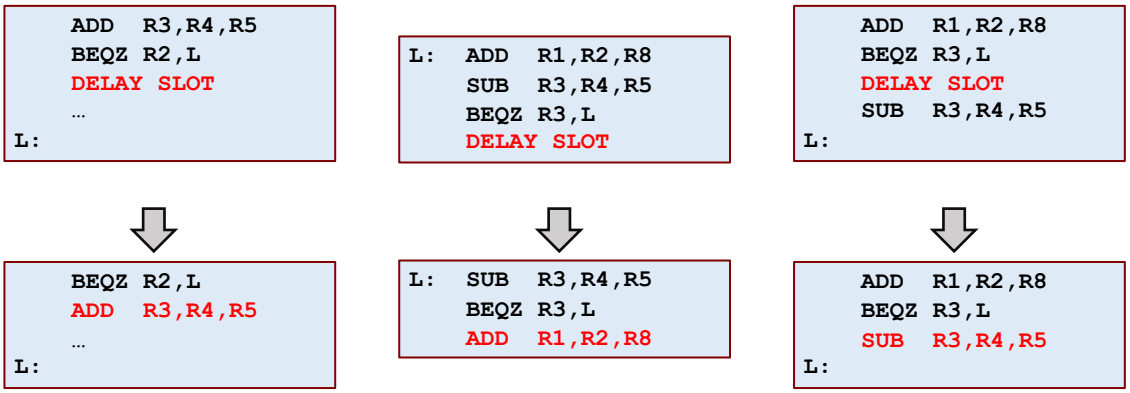

**For MIPS32, n=1**

- Just like in load-delay slots, the task of the compiler is to try and fill up the branch-delay slots with meaningful instructions.
- Instructions in the branch-delay slot are *always executed* irrespective of the outcome of the branch.

106

106

## Some Examples




107

107

- Additional overhead for delayed branches:
  - Multiple PC's (one plus the length of the delay slot, i.e. $n + 1$) are needed to correctly restore the state when an interrupt occurs.
  - Consider a taken branch instruction followed by instruction(s) in the delay slot(s).
    - The PC of branch target and PC's of delay slots are not sequential.

108

108

## Example 4

- Consider the MIPS32 pipeline with ideal CPI of 1. Assume that 20% of the instructions executed are branch instructions, out of which 75% are taken branches. Evaluate the pipeline speedup for the four techniques discussed to reduce branch penalties.

- **Solution**:

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles / instruction}}$$

$$= \frac{\text{Pipeline Depth}}{1 + (\text{Branch frequency} \times \text{Branch penalty})}$$

109

109

a) Stall pipeline

Branch penalty = 1
Speedup = 5 / (1 + 0.20 x 1)
= 4.17

a) Predict not taken

Branch penalty = 1
Speedup = 5 / (1 + (0.20 x 0.75))
= 4.35

c) Predict taken

Branch penalty = 1
Speedup = 5 / (1 + 0.20 x 1)
= 4.17

d) Delayed branch

Branch penalty = 0.5
Speedup = 5 / (1 + (0.20 x 0.5))
= 4.55

110

110