# CS / EE 320
# Computer Organization and Assembly Language
## Spring 2025
## Lecture 18

**Shahid Masud**

**Topics: Hazards and other problems in 5 Stage MIPS Pipeline**

# Topics

- **Hazards** in Pipelining
- **Structural Hazards** due to hardware constraints, Von-Neuman, Memory access limitations, etc.
- **Data Hazards**
  - Availability of Correct Data
  - Stall operation
  - Forwarding Operation
- **Control Hazards**
  - Branches and Jumps
  - Branch Prediction
- Examples of Different Types of Hazards
- Pipeline Diagrams of Different Types of Hazards

# Pipeline Hazards

# Problems in Pipelined CPU Architecture

Limits to pipelining: **Hazards prevent next instruction from executing during its designated clock cycle**:

- **Structural hazards:** HW cannot support this combination of instructions (e.g. single person to fold and put clothes away)

- **Data hazards:** Instruction depends on result of prior instruction still in the pipeline (missing clothes)

- **Control hazards:** Pipelining of branches & other instructions that change the PC

- **Common solution** is to "**stall**" the pipeline until the hazard is resolved, inserting one or more "**bubbles**" in the pipeline

# Data Hazards – in a nutshell

- Cause **Stalls** in Pipelines as Data required by succeeding instructions in pipeline is not yet available in memory or registers

- Following measures are possible to ensure correct operation and improve CPU throughput:
  - Introducing NOP instructions that add wait states in pipelines
  - Instruction Re-ordering so that instructions without Dependencies are processed earlier to allow sufficient time for Dependencies
  - Equip ALU with Forwarding where ALU output can be directly connected to ALU input to handle data dependencies

- Look at some code examples to improve throughput using NOP and Forwarding
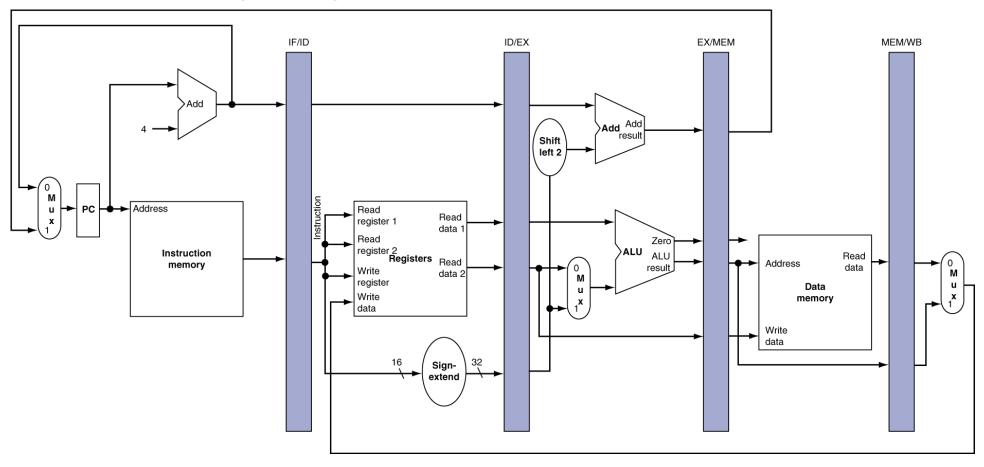
# MIPS CPU with Pipeline

**Comprises Five distinct stages**, **one operation step per stage**

1. **IF:** Instruction fetch from memory
2. **ID:** Instruction decode & register read
3. **EX:** Execute operation or calculate address
4. **MEM:** Access memory operand
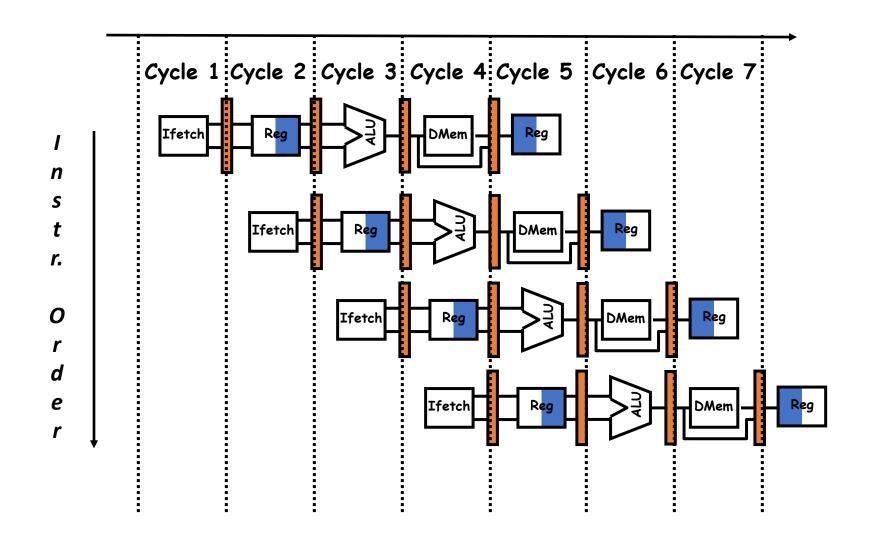5. **WB:** Write result back to register

- **Need registers between stages**
  - **To hold information produced in previous cycle, required in subsequent cycles**

# The Basic Pipeline For MIPS – Simplified Diagram

# Structural Hazard

# Structural Hazards - Reason

- **Conflict situation for use of a hardware resource**

- **Von Neumann limitations**

- In MIPS pipeline with a **single memory**
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"

- Hence, **pipelined data paths require separate instruction/data memories**
  - Or separate instruction/data caches

# Structural Hazards

- **If not all possible combinations of instructions can be executed, structural hazards occur**

- **Pipelines stall result of this hazard, CPI increased from the usual "1"**

- **Avoid structural hazards by duplicating resources**
  - **e.g. an ALU to perform an arithmetic operation and an adder to increment PC**

# Common Structural Hazards

**Some common Structural Hazards:**

- **Memory** – same memory for reads and writes
- **Floating point** - Since many floating-point instructions require many cycles, it's easy for them to interfere with the other
- Same reasoning for other special hardware units

**Structural hazards are reduced with these rules:**

- **Each instruction uses a resource at most once**
- **Always use the resource in the same pipeline stage**
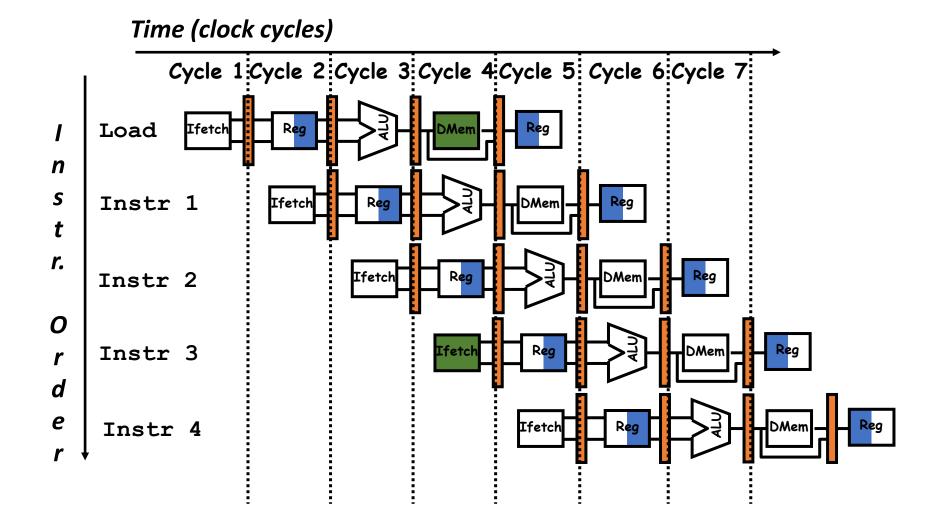- **Use the resource for one cycle only**

**(More in the pipeline diagrams in following slides)**
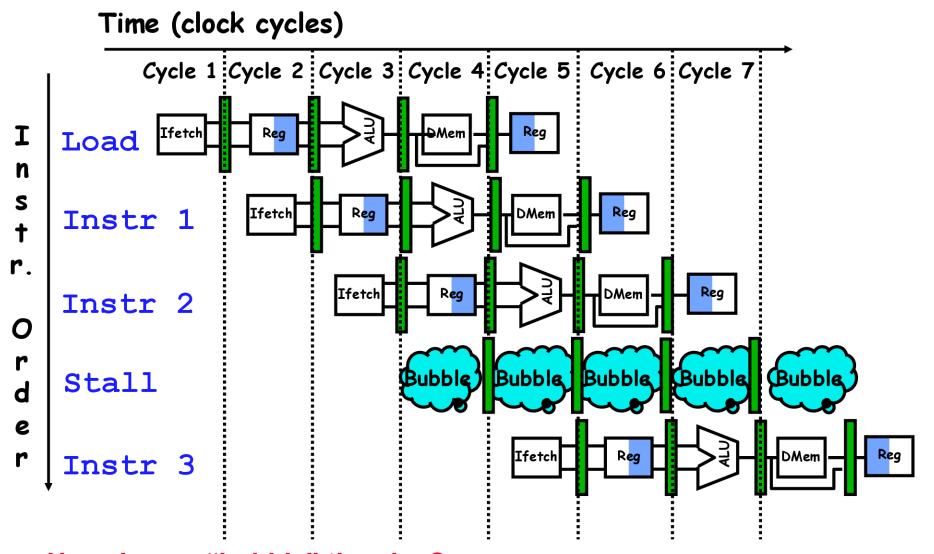
# Structural Hazards

When two or more different instructions want to use same hardware resource in the same cycle

e.g., MEM uses the same memory port as IF as shown in this slide

*Time (clock cycles)*

# One Memory Port/Structural Hazards



Time (clock cycles)

# Or, alternatively – in a table
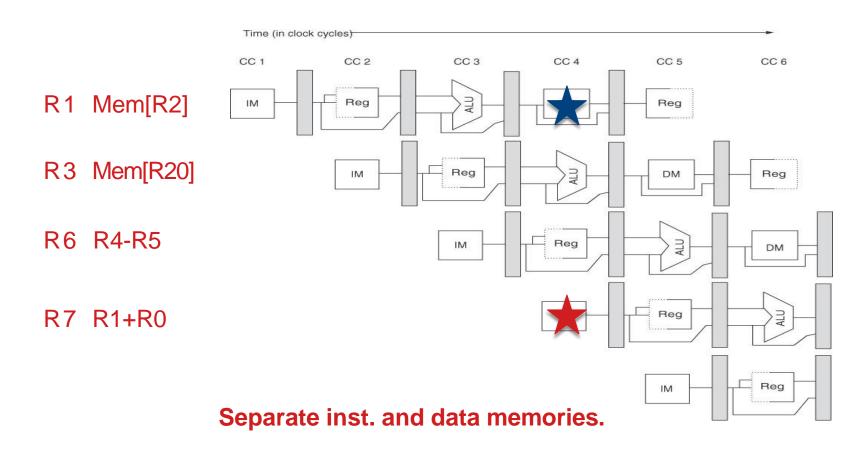
**Clock Number**

| Inst. # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| LOAD | IF | ID | EX | MEM | WB | | | | | |
| Inst. i+1 | | IF | ID | EX | MEM | WB | | | | |
| Inst. i+2 | | | IF | ID | EX | MEM | WB | | | |
| Inst. i+3 | | | | stall | IF | ID | EX | MEM | WB | |
| Inst. i+4 | | | | | | IF | ID | EX | MEM | WB |
| Inst. i+5 | | | | | | | IF | ID | EX | MEM |
| Inst. i+6 | | | | | | | | IF | ID | EX |

- **LOAD instruction "steals" an instruction fetch cycle which will cause the pipeline to stall.**

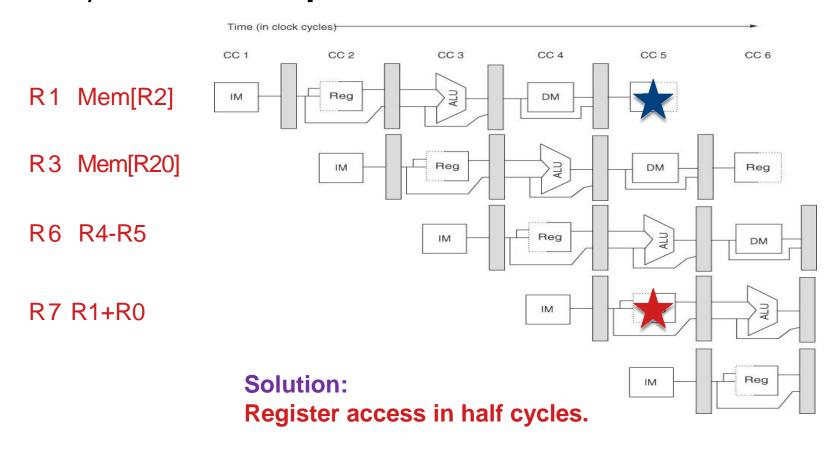- **Thus, no instruction completes on clock cycle 8**

# Structural Hazards

- 1. Unified memory for instruction and data
- 2. Register file with shared read/write access ports



R 1   Mem[R2]

R 3   Mem[R20]

R 6   R4-R5

R 7   R1+R0

**Separate inst. and data memories.**

☐ **Structural Hazard due to Register file with shared read/write access ports**



R1 Mem[R2]

R3 Mem[R20]

R6 R4-R5

R7 R1+R0

**Solution:**
**Register access in half cycles.**

# What is realistic solution of Structural Hazard?

- **Answer:   Add more hardware.**
  - **(especially for the memory access example – i.e. the common case)**
    - **CPI degrades quickly from our ideal '1' for even the simplest of cases…**

# Data Hazard

# Data Hazards

**These occur when, at any time, there are instructions active that need to access the "same data (memory or register) locations ".**

Where there's real trouble is when we have two instructions:

instruction A

instruction B

and **B manipulates (reads or writes) data before A does**.

**This violates the order of the instructions, since the architecture implies that A completes entirely before B is executed.**

# Data Hazards ... contd

- **These exist because of pipelining**

- **Why do they exist???**
  - **Pipelining changes "when" data operands are read and written**
  - **Order differs from order seen by sequentially executing instructions on non-pipelined machine**

- **Consider this example:**
  - ADD R1, R2, R3
  - SUB R4, R1, R5
  - AND R6, R1, R7
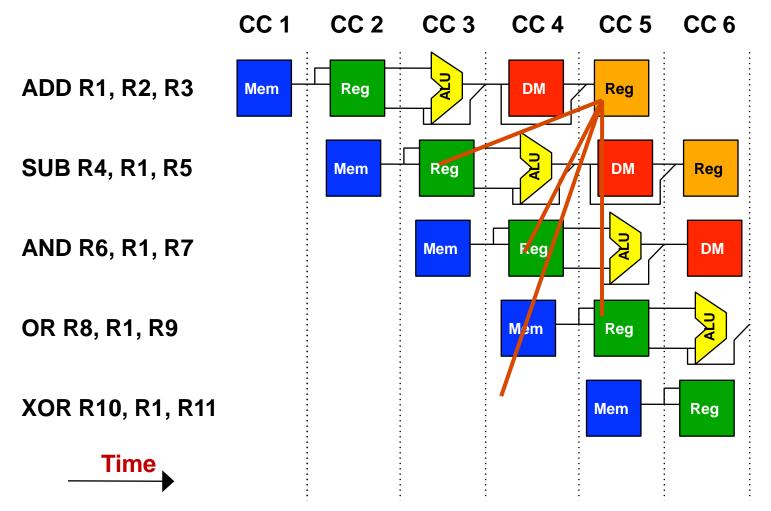  - OR R8, R1, R9
  - XOR R10, R1, R11

All instructions after ADD use result of ADD

ADD writes the register in WB but SUB needs it in ID.

**This is a data hazard**

# Illustrating a Data Hazard



**ADD instruction causes a hazard in next 3 instructions because register not written until after those 3 read it.**

# Data Hazard Types

- **There are actually 3 different kinds of data hazards:**
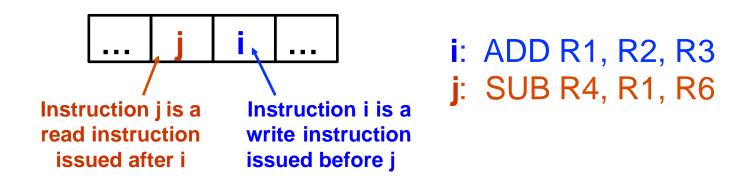
  – **Read After Write (RAW)**

  – **Write After Write (WAW)**

  – **Write After Read (WAR)**

Common in MIPS Pipelined CPU

# Read after write (RAW) hazards

- **With RAW hazard, instruction j tries to read a source operand before instruction i writes it.**

- **Thus, j would incorrectly receive old or incorrect value**

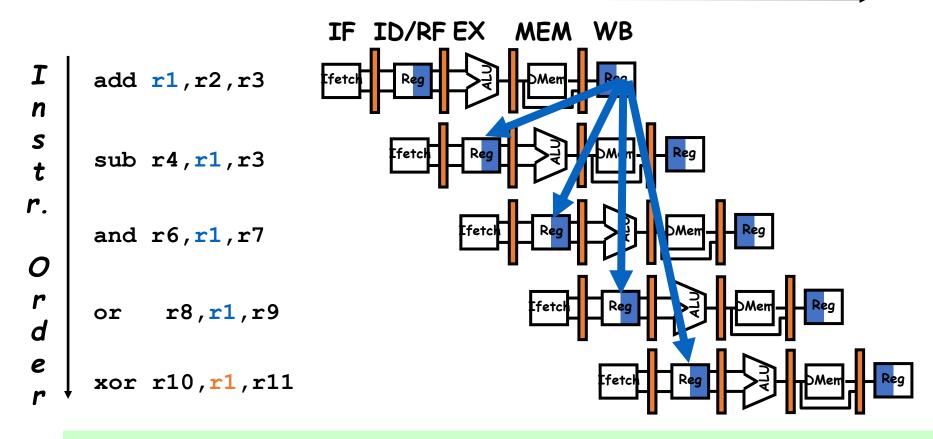- **Example:**

| ... | j | i | ... |
|-----|---|---|-----|

**i**: ADD R1, R2, R3
**j**: SUB R4, R1, R6

Instruction j is a read instruction issued after i

Instruction i is a write instruction issued before j

- **We Can use "stalling" or "forwarding" to resolve this hazard**

Time (clock cycles)

IF  ID/RF  EX  MEM  WB

**Instr. Order**

add **r1**,r2,r3

sub r4,**r1**,r3

and r6,**r1**,r7

or    r8,**r1**,r9

xor r10,**r1**,r11

The use of the result of the ADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.
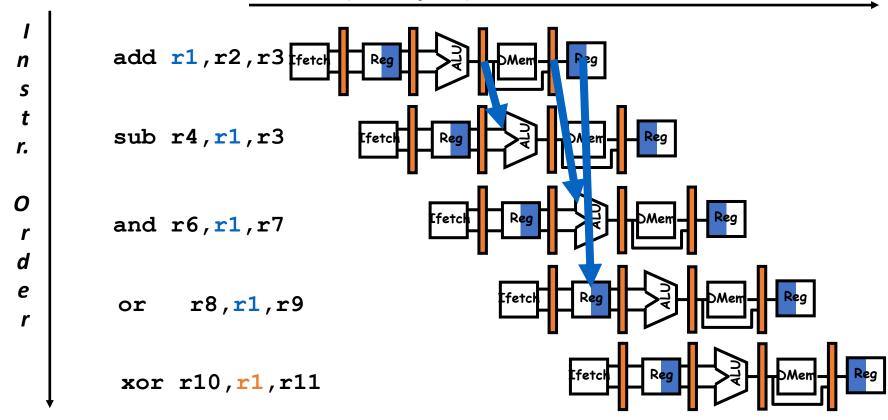
# Data Hazards – possible solution

**Forwarding To Avoid Data Hazard**

Forwarding is the concept of making data available to the input of the ALU for subsequent instructions, even though the generating instruction hasn't gotten to WB in order to write the memory or registers.

*Time (clock cycles)*

*Instr. Order*

add **r1**,r2,r3    Ifetch  Reg  ALU  DMem  Reg

sub r4,**r1**,r3    Ifetch  Reg  ALU  DMem  Reg

and r6,**r1**,r7    Ifetch  Reg  ALU  DMem  Reg

or   r8,**r1**,r9    Ifetch  Reg  ALU  DMem  Reg

xor r10,**r1**,r11    Ifetch  Reg  ALU  DMem  Reg

**Discussed Later**

**Read After Write (RAW)**
Instr$_J$ tries to read operand before Instr$_I$ writes it

Execution Order is:
**Instr$_I$**
**Instr$_J$**

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- Caused by a "**Dependence**" (in compiler nomenclature).  This hazard results from an actual need for communication.

**Write After Read (WAR)**
Instr$_J$ tries to write operand *before* Instr$_I$ reads i

- Gets wrong operand

Execution Order is:
**Instr$_I$**
**Instr$_J$**

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "anti-dependence" by compiler writers.
  This results from reuse of the name "r1".

- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

**Write After Write (WAW)**

Instr$_J$ tries to write operand _before_ Instr$_I$ writes it

- Leaves wrong result ( Instr$_I$ not Instr$_J$ )

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

Execution Order is:
**Instr$_I$**
**Instr$_J$**

- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1".

- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
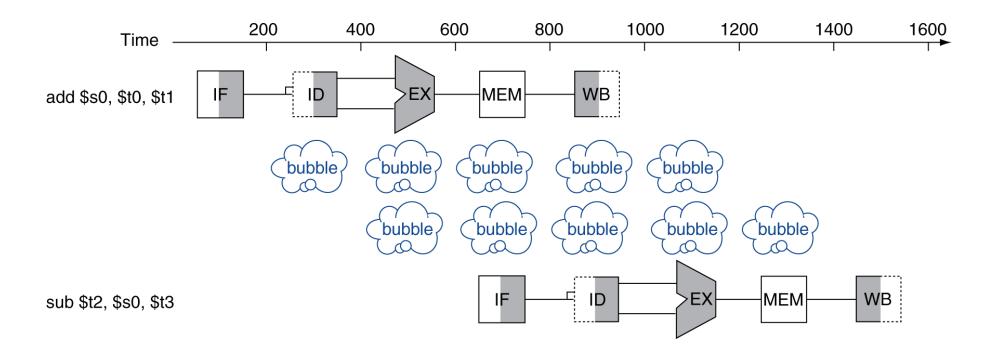
- Will see WAR and WAW in later more complicated pipes

# Data Hazard and Stall

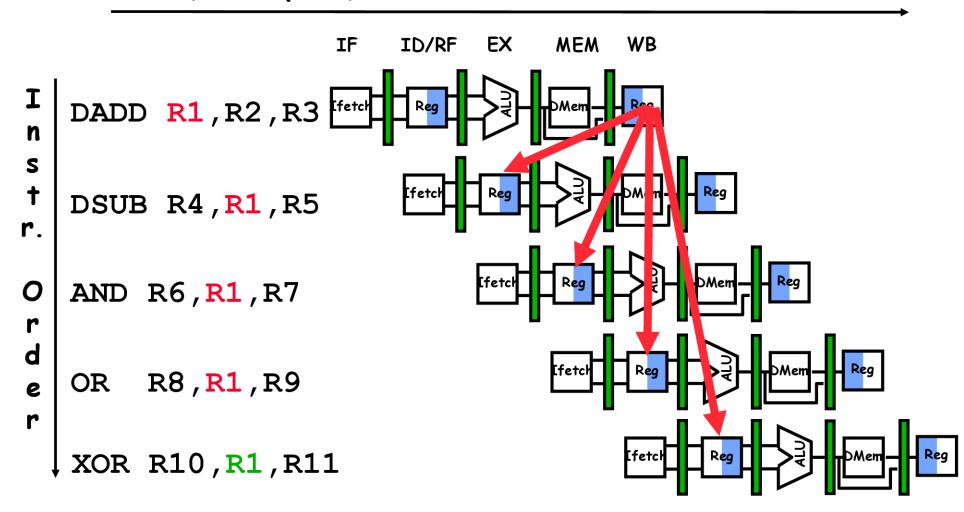- An instruction depends on completion of data access by a previous instruction
  - add        $s0, $t0, $t1
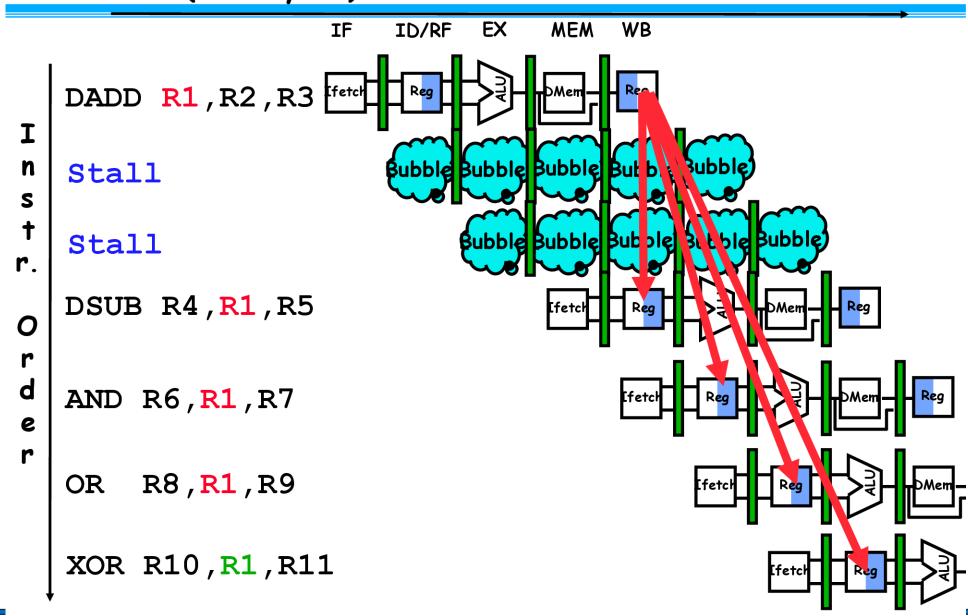    sub        $t2, $s0, $t3

# Data Hazard on R1

**Time (clock cycles)**

IF     ID/RF    EX    MEM   WB

**I n s t r .   O r d e r**

DADD R1,R2,R3

DSUB R4,R1,R5

AND R6,R1,R7

OR  R8,R1,R9

XOR R10,R1,R11

# Solution #1: Insert stalls

# Data Hazards

■ Data Hazards

◆ Occur when the **pipeline changes the order of read/write accesses to operands s**o that the order differs from the order seen by sequentially executing instructions on an unpipelined processor.

» Occur when some functional unit is not fully pipelined, or

» No enough duplicated resources.

◆ A example of pipelined execution

DADD  R1, R2, R3

DSUB  R4, R1, R5

AND   R6, R1, R7

OR    R8, R1, R9

XOR   R10, R1, R11

# Data Hazards – table representation

| LW    R1, 0(R2) | IF | ID | EX | MEM | WB |     |     |     |     |
|-----------------|----|----|----|-----|-----|-----|-----|-----|-----|
| SUB   R4, R1, R5 |    | IF | ID | EX  | MEM | WB  |     |     |     |
| AND   R6, R1, R7 |    |    | IF | ID  | EX  | MEM | WB  |     |     |
| OR    R8, R1, R9 |    |    |    | IF  | ID  | EX  | MEM | WB  |     |

| LW    R1, 0(R2) | IF | ID | EX | MEM | WB |     |     |     |     |
|-----------------|----|----|----|-----|-----|-----|-----|-----|-----|
| SUB   R4, R1, R5 |    | IF | ID | stall | EX  | MEM | WB  |     |     |
| AND   R6, R1, R7 |    |    | IF | stall | ID  | EX  | MEM | WB  |     |
| OR    R8, R1, R9 |    |    |    | stall | IF  | ID  | EX  | MEM | WB  |

# Memory Data Hazards - RAW

- **Like register hazards, we can also have <u>memory</u> <u>hazards</u>**
  - **RAW:**
    - store R1, 0(SP)
    - load R4, 0(SP)

|  | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **Store R1, 0(SP)** | F | D | EX | M | WB | |
| **Load R1, 0(SP)** | | F | D | EX | M | WB |

  - **In simple pipeline, memory hazards are easy to address**
    - **In order, one at a time, read & write in same stage**
  - **In general though, more difficult than register hazards**

# Data Hazards and the Compiler

- **Compiler should be able to help eliminate some stalls caused by data hazards**

- **i.e. compiler could restrict generating a LOAD instruction that is immediately followed by instruction that uses result of LOAD's destination register**

# What about Control Logic in Hazard?

- **For MIPS integer pipeline, all data hazards can be checked during ID phase of pipeline**

- **If data hazard, instruction stalled before its issued**

- **Whether forwarding is needed can also be determined at this stage, controls signals set (discussed later)**

- **If hazard detected, control unit of pipeline must stall pipeline and prevent instructions in IF, ID from advancing**

# Hazards vs. Dependencies

- **dependence: fixed property of instruction stream**
  - **(i.e., program)**

- **hazard: property of <u>program</u> <u>and</u> <u>processor</u> <u>organization</u>**
  - **implies potential for executing things in wrong order**
    - **potential only exists if instructions can be simultaneously "in-flight" in the pipeline**
    - **property of dynamic distance between instructions vs. pipeline depth**

- **For example, can have RAW dependence with or without hazard**
  - **depends on pipeline**

# Some Situations of "Dependence" that cause Hazard

| Situation | Example | Action |
|---|---|---|
| **No Dependence** | LW R1, 45(R2)<br>ADD R5, R6, R7<br>SUB R8, R6, R7<br>OR R9, R6, R7 | No hazard possible because no dependence exists on R1 in the immediately following three instructions. |
| **Dependence requiring stall** | LW R1, 45(R2)<br>ADD R5, R1, R7<br>SUB R8, R6, R7<br>OR R9, R6, R7 | Comparators detect the use of R1 in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX |
| **Dependence overcome by forwarding (introduced later)** | LW R1, 45(R2)<br>ADD R5, R6, R7<br>SUB R8, R1, R7<br>OR R9, R6, R7 | Comparators detect the use of R1 in SUB and forward the result of LOAD to the ALU in time for SUB to begin with EX |
| **Dependence with accesses in order** | LW R1, 45(R2)<br>ADD R5, R6, R7<br>SUB R8, R6, R7<br>OR R9, R1, R7 | No action is required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half. |

# Forwarding or Bypassing

# Forwarding
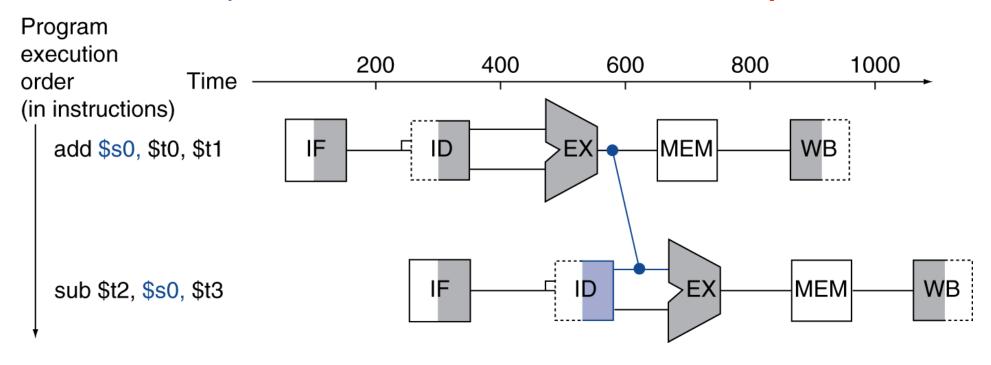
- **Problem illustrated previously can be solved**
  – **with forwarding**

- **E.g. Can we move the result from EX/MEM register to the beginning of ALU (where SUB needs it)?**
  – **Yes!**

- **Generally speaking:**
  – **Forwarding occurs when a result is passed directly to functional unit that requires it.**
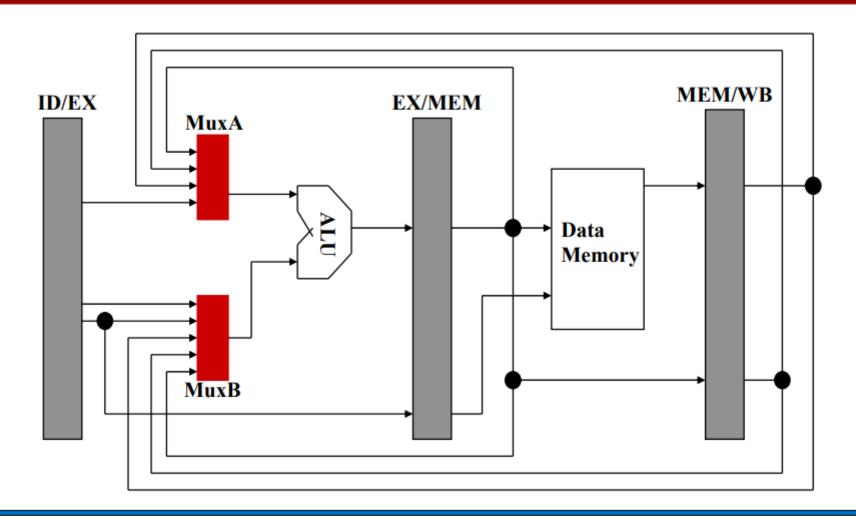  – **Result goes from output of one unit to input of another**

- **Use result when it is computed**
  - Don't wait for it to be stored in a register
  - Requires **extra connections in the datapath**

## Forward Hardware - Datapath
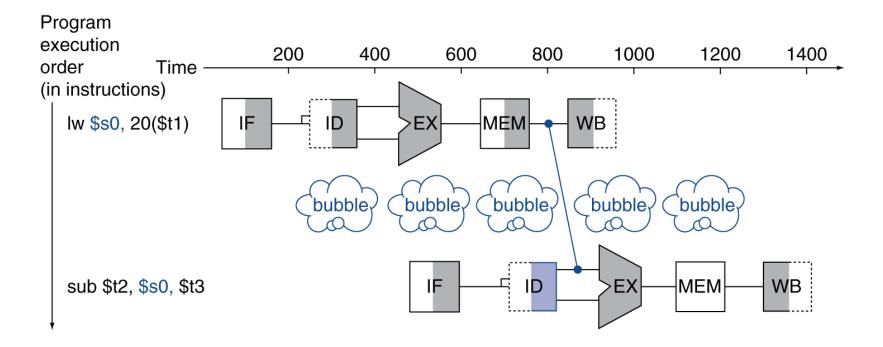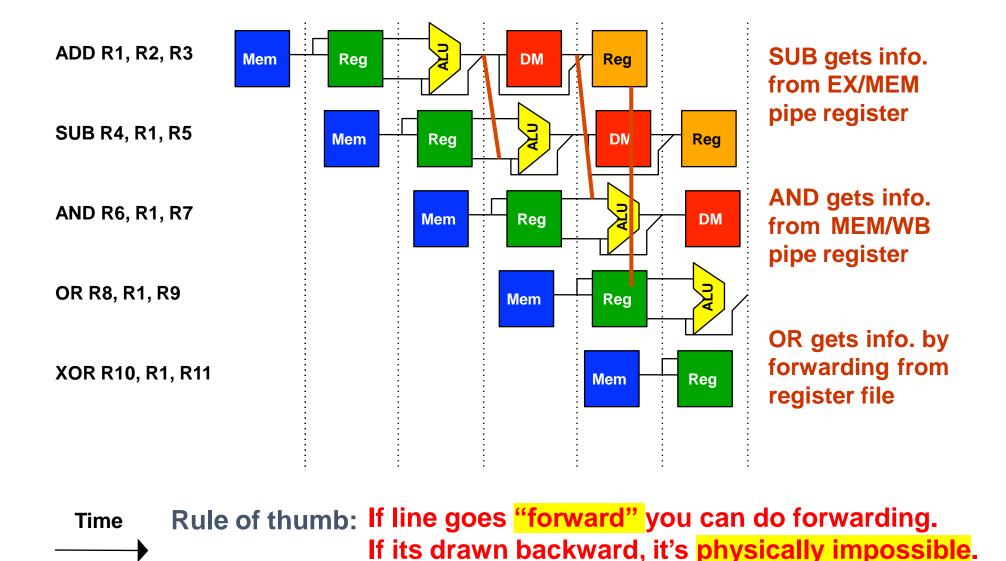
- **Can't always avoid stalls by forwarding**
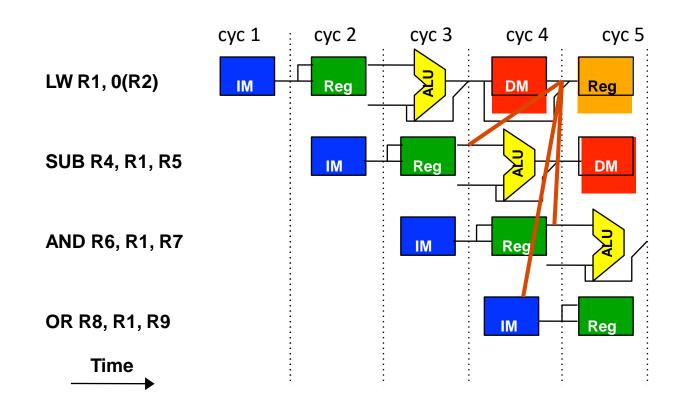  - **If value not computed when needed**
  - **Can't forward backward in time!**
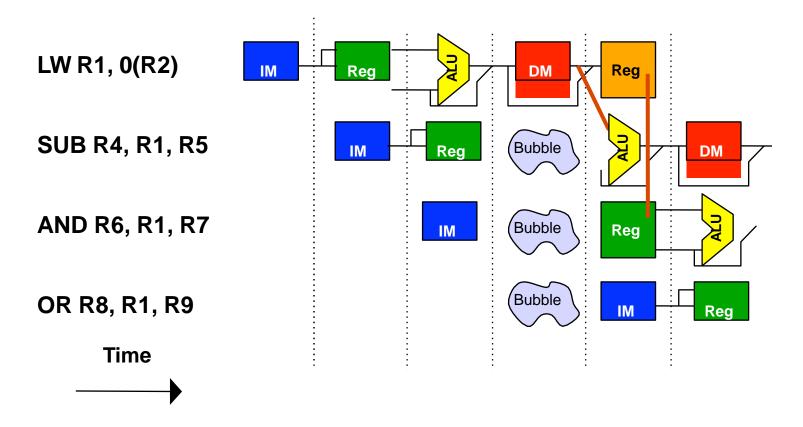
# Where can we Introduce Forwarding? Example



ADD R1, R2, R3

SUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

XOR R10, R1, R11

**SUB gets info. from EX/MEM pipe register**

**AND gets info. from MEM/WB pipe register**

**OR gets info. by forwarding from register file**

Time

**Rule of thumb:** If line goes **"forward"** you can do forwarding.
If its drawn backward, it's **physically impossible.**

# Forwarding doesn't always Work



cyc 1    cyc 2    cyc 3    cyc 4    cyc 5

LW R1, 0(R2)

SUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

Time

**Load has a latency that forwarding can't solve.**

**Pipeline must stall until hazard cleared (starting with instruction that wants to use data until source produces it).**

## Can't get data to subtract as result needed at beginning of CC 4, but not produced until end of CC 4

**LW R1, 0(R2)**

**SUB R4, R1, R5**

**AND R6, R1, R7**

**OR R8, R1, R9**

**Time**

**Insertion of bubble causes Number of cycles for completing this sequence to grow by 1**

## Simple Solution to RAW

- Hardware detects RAW and **stalls**
- Assumes register written then read each cycle
    - low cost to implement, simple
    - reduces IPC

## Minimizing RAW stalls
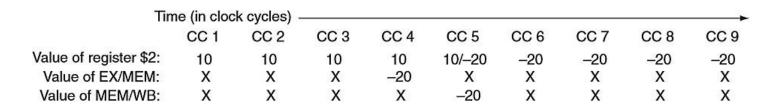
- **Bypass / forward / shortcircuit** (**"forwarding"**)
- Use data before it is in the register
    - reduces/avoids stalls
    - complex
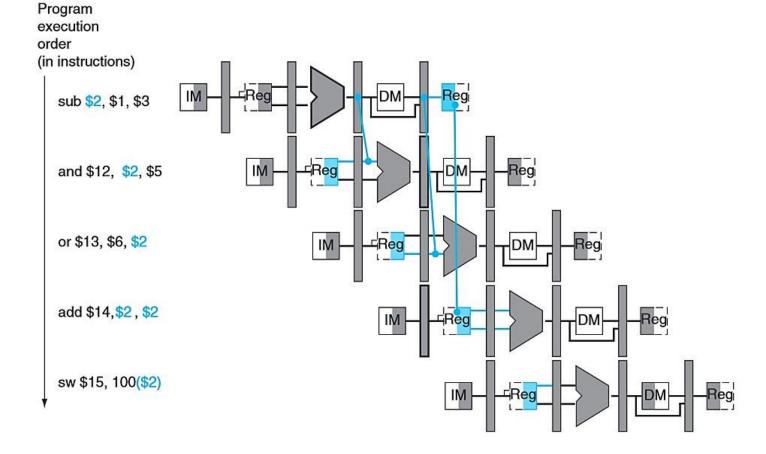
Note the change in our dependency diagram. Now, we can see that the inputs to the ALU are dependent on pipeline registers rather than the results of the WB stage.

Forwarding works by allowing us to grab the inputs of the ALU not only from the ID/EX pipeline register, but from any other pipeline register.
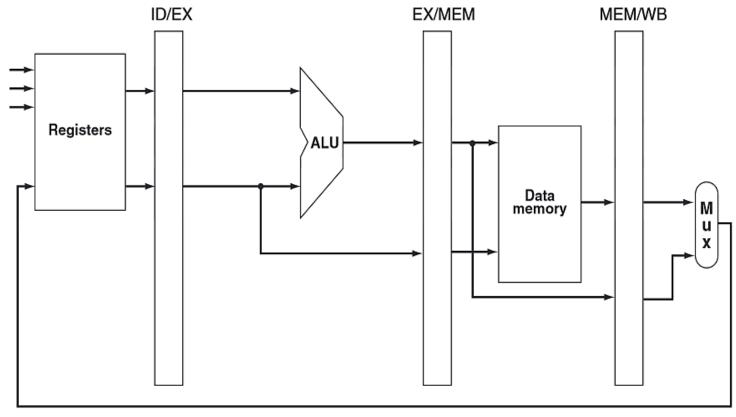
MIPS pipelined datapath which has no support for forwarding
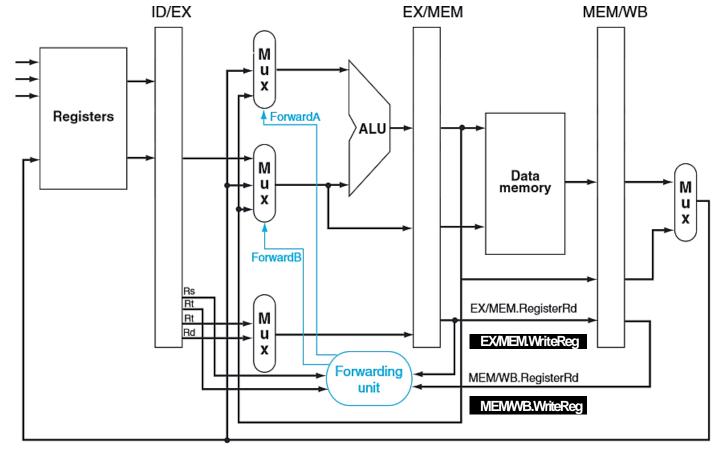


a. No forwarding

Now with forwarding!

Instead of directly piping the ID/EX pipeline register values into the ALU, we now have **multiplexors** that allow us to choose where the input should come from.
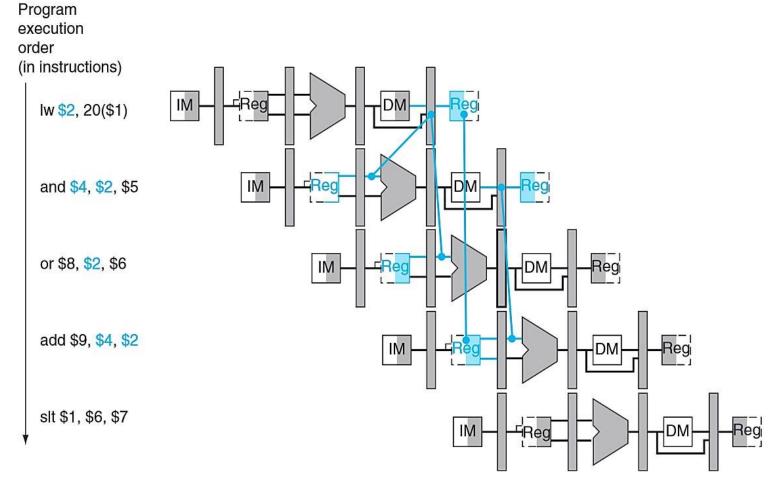


b. With forwarding

# Data Hazard and Stall

Consider the following sequence of instructions.

```
lw   $2, 20($1)
and  $4,  $2,  $5
or   $8,  $2,  $6
add  $9,  $4,  $2
slt  $1,  $6,  $7
```
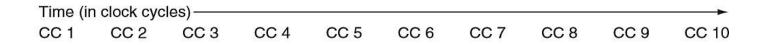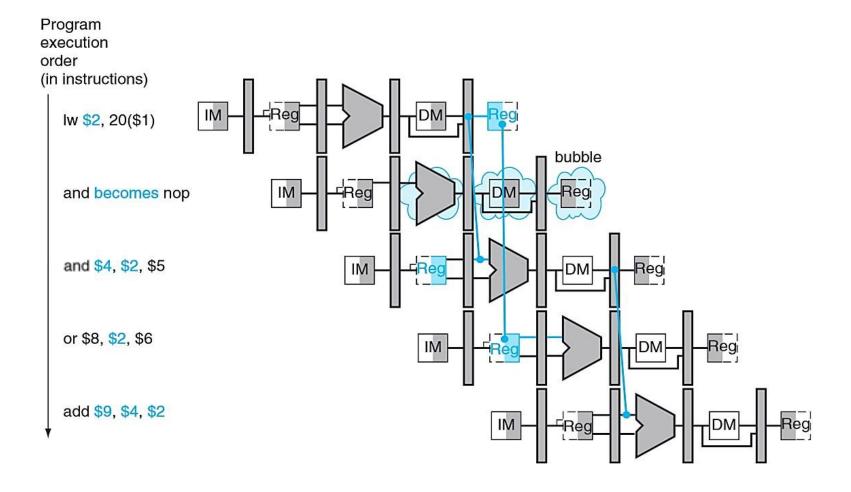
Even with forwarding, our dependency from a load word instruction goes backward in time.

# CONTROL HAZARDS

# Branch / Control Hazards

- **So far, we've limited discussion of hazards to:**
  - **Arithmetic/logic operations (remember Forwarding)**
  - **Data transfers (remember Data Hazards)**

- **Also need to consider hazards involving branches:**
  - **Example:**
    - **40:  beq      $1, $3, 28          # (28 leads to address 72)**
    - **44:  and      $12, $2, $5**
    - **48:  or       $13, $6, $2**
    - **52:  add      $14, $2, $2**
    - **72:  lw       $4, 50($7)**

- **How long will it take before the branch decision takes effect?**
  - **What happens in the meantime?**

# Control Hazards

- Arise in **Conditional Branch and Jump Instructions**
- By the time Branch conditions are evaluated and new PC is updated (in case Branch Taken), there are already two further instructions in the pipeline. This slows down CPU throughput.
- Objective is to handle Branches so that the pipeline stages are not wasted.
- Two Broad Techniques:
  - **Speculation** – Eg. Assume Branch Always Taken or Branch Never Taken
  - **Branch Prediction** – Predict likely outcome of Branch Instructions to reduce wasted time cycles in pipelines
- Will be studied further

# Introduction to Branch Prediction

In *branch prediction*, we attempt to predict the branching decisions and act accordingly.

When we assumed the branch wasn't taken, we were making a simple static prediction. Luckily, the performance cost on a 5-stage pipeline is low but on a *deeper* pipeline with many more stages, that could be a huge performance cost!

In *dynamic branch prediction*, we look up the address of the instruction to see if the branch was taken last time. If so, we will predict that the branch will be taken again and optimistically fetch the instructions from the branch target rather than the subsequent instructions.

# Readings

- Chap 4 of P&H Textbook