CSEN 702

Microprocessors

Lecture 3

# Review on MIPS64 pipelining and floating point (Part 2)

**Reading**

Textbook Chapter 1.1, 1.2, 1.3, Appendix A, Appendix C

# Performance of pipelines with stalls

- General formula of speedup

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$=$$

$$=$$

# Performance of pipelines with stalls (2)

- Let's take **approach (A)** from previous lecture: goal of pipeline is reducing CPI of instruction, keeping same clock cycle time.

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction} \\ &= 1 + \text{Pipeline stall clock cycles per instruction}\end{aligned}$$

Result

$$\textit{Speedup} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

Remember in this approach, same clock cycle time so the division = 1

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

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

if we assume all instructions take same number of cycles = number of pipeline stages (pipeline depth)

With no stalls, the speed-up is actually equal to the pipeline depth. (for example, speedup = 5 for a 5-stage pipeline)

# Performance of pipelines with stalls (3)

- Let's take **approach (B)**: goal of pipeline is reducing clock cycle time.
- → **we can assume that the CPI of the un-pipelined processor, as well as that of the pipelined processor, is = 1**

- **Ideally**, the clock cycle on the pipelined processor is smaller than the clock

# Performance of pipelines with stalls (3)

- **Continuing with approach (B)**

$$\text{Speedup from pipelining} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

= pipeline depth
(from previous slide)

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

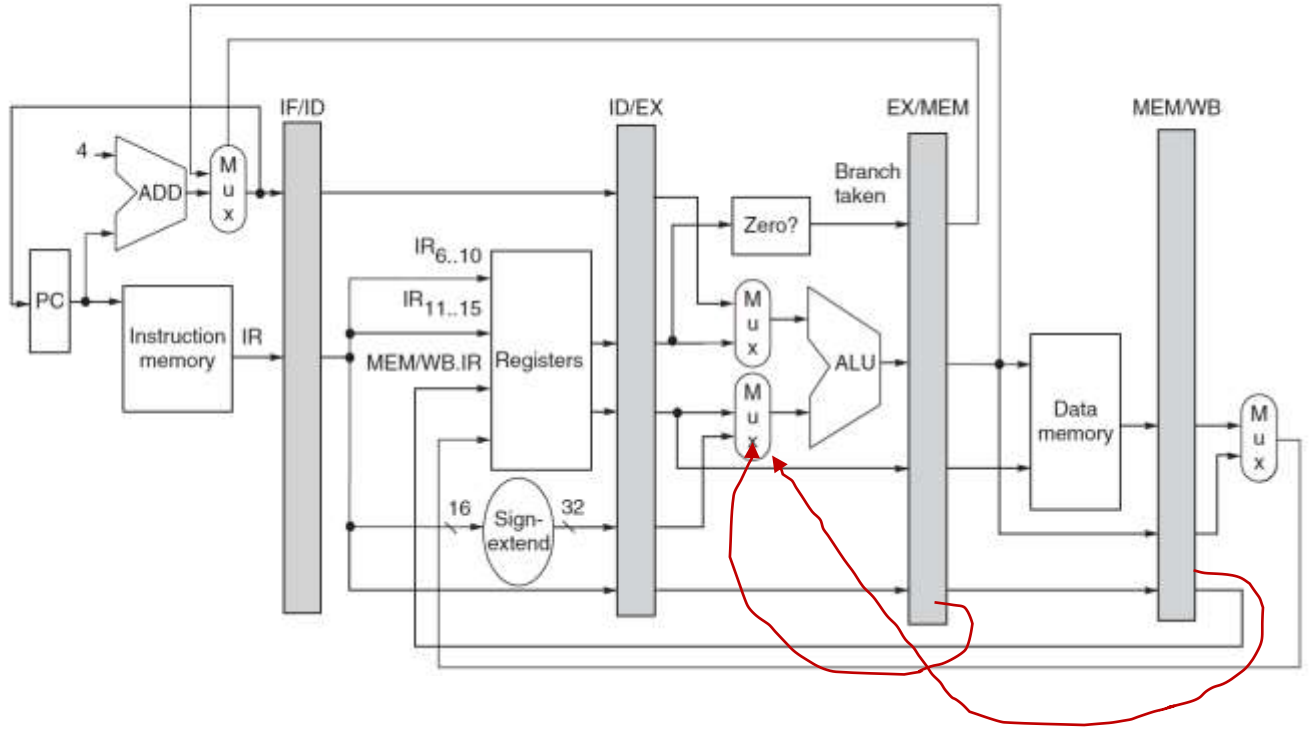
Same result we got with approach (A)

# Minimizing data hazards using ***forwarding***

- A hardware technique, also called bypassing, to **forward** results needed back in the pipeline to the ALU without waiting for the pipeline registers.

1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.

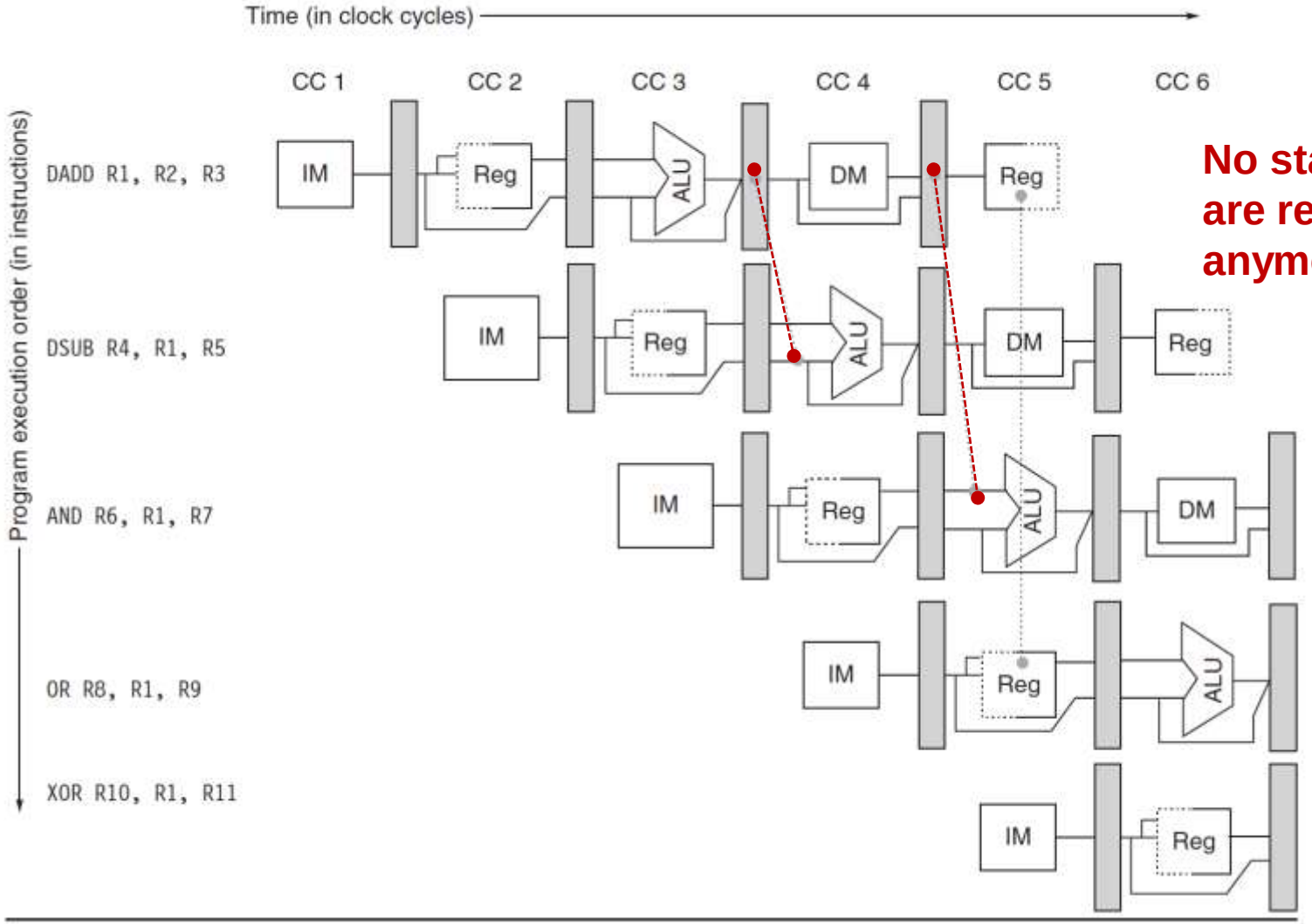
2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.



# Re-visiting example 1

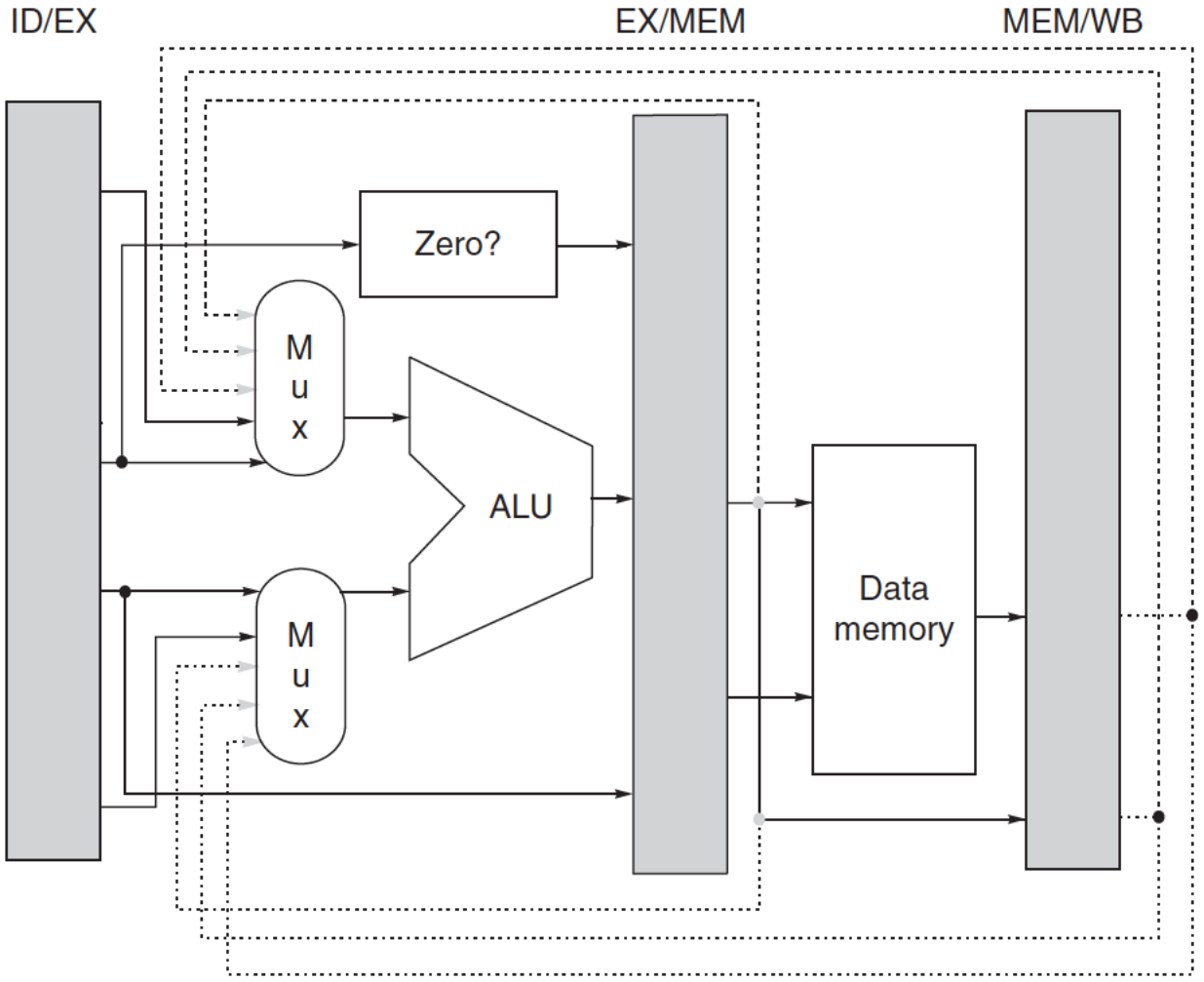
**Example 1**

```
DADD    R1,R2,R3
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
XOR     R10,R1,R11
```




**No stalls are required anymore!**

# Changes in the datapath




Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs.

The paths correspond to a bypass of:

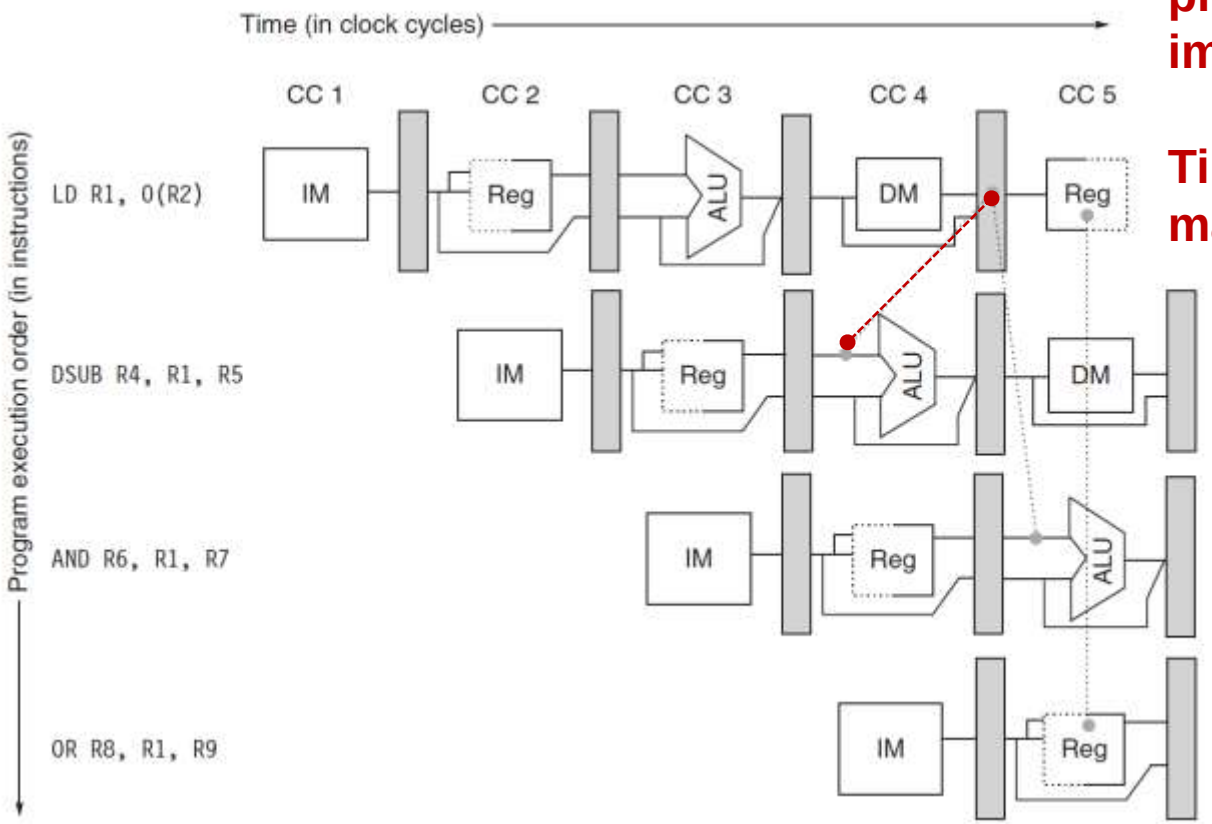
(1) the ALU output at the end of the EX
(2) the ALU output at the end of the MEM stage
(3) the memory output at the end of the MEM stage.

# Stalls are not always avoidable!

**Example 2**

```
LD      R1,0(R2)
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
```

What about the AND and the OR?
Can forwarding work?




**For DSUB, this is physically impossible.**

**Time travel maybe? ☺**

# Example 2- stalls required and effects

**Example 2**

```
LD      R1,0(R2)
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
```

- As we saw, a stall is required for DSUB.
- Once we stall, we have stall the entire pipeline. (represented vertically)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| LD | R1,0(R2) | IF | ID | EX | MEM | WB | | | |
| DSUB | 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 |

# Control hazards

- They originate from branch instructions that may or may not change the PC address.
- Simple techniques involve treating the branch as always taken or always not taken (**static prediction**), and when the decision is known, flush the pipeline if needed → penalty!

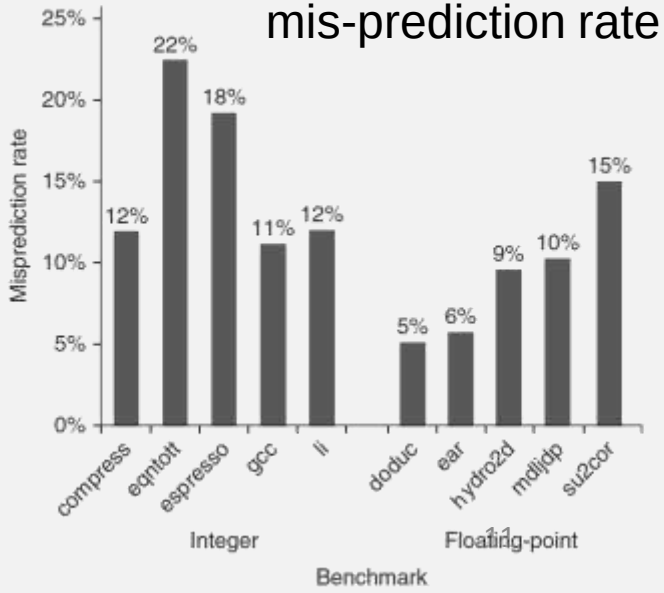
$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

Is this accurate?

: Branch frequency × Branch penalty

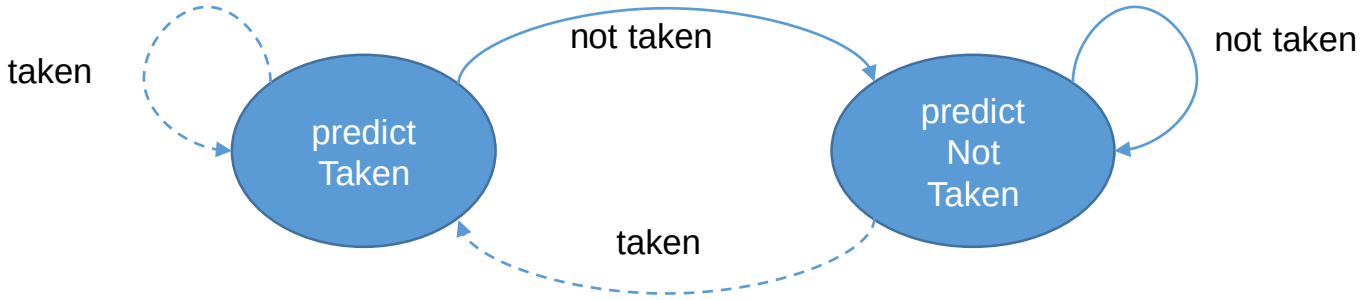
$$= \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

mis-prediction rate



# Dynamic prediction

- Use of prediction buffers or Branch history tables (BHT)
- **1-bit prediction:**
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken) as 1 bit.
  - To execute a branch:
    - Check BHT, expect (predict) the same outcome
    - Start fetching from fall-through (next instruction) or target address
    - If wrong, flush pipeline and **flip** prediction bit in the table



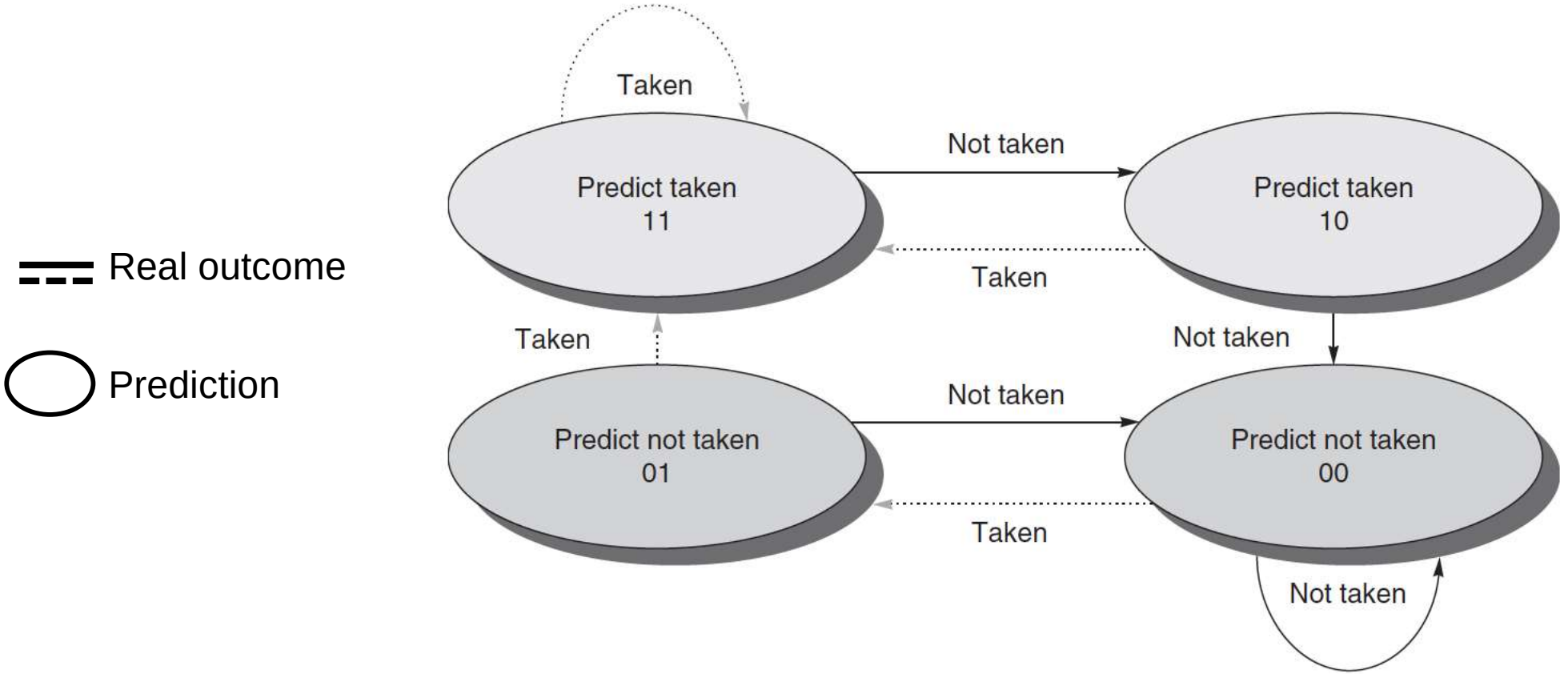
# 1-bit predictor problem

- Assume you have 2 nested loops (for loops for example)
- Inner loop branches mis-predicted twice!
    1. Mis-predict as taken on last iteration of inner loop
    2. Then mis-predict as not taken on first iteration of inner loop next time around

```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```

# 2-bit predictor

- A prediction must miss twice before it's changed in the BHT.



Part 2

# Floating point operations

# Floating-point operations

- FP operations take longer to execute than integer operations. i.e. multiplication, division and so on.

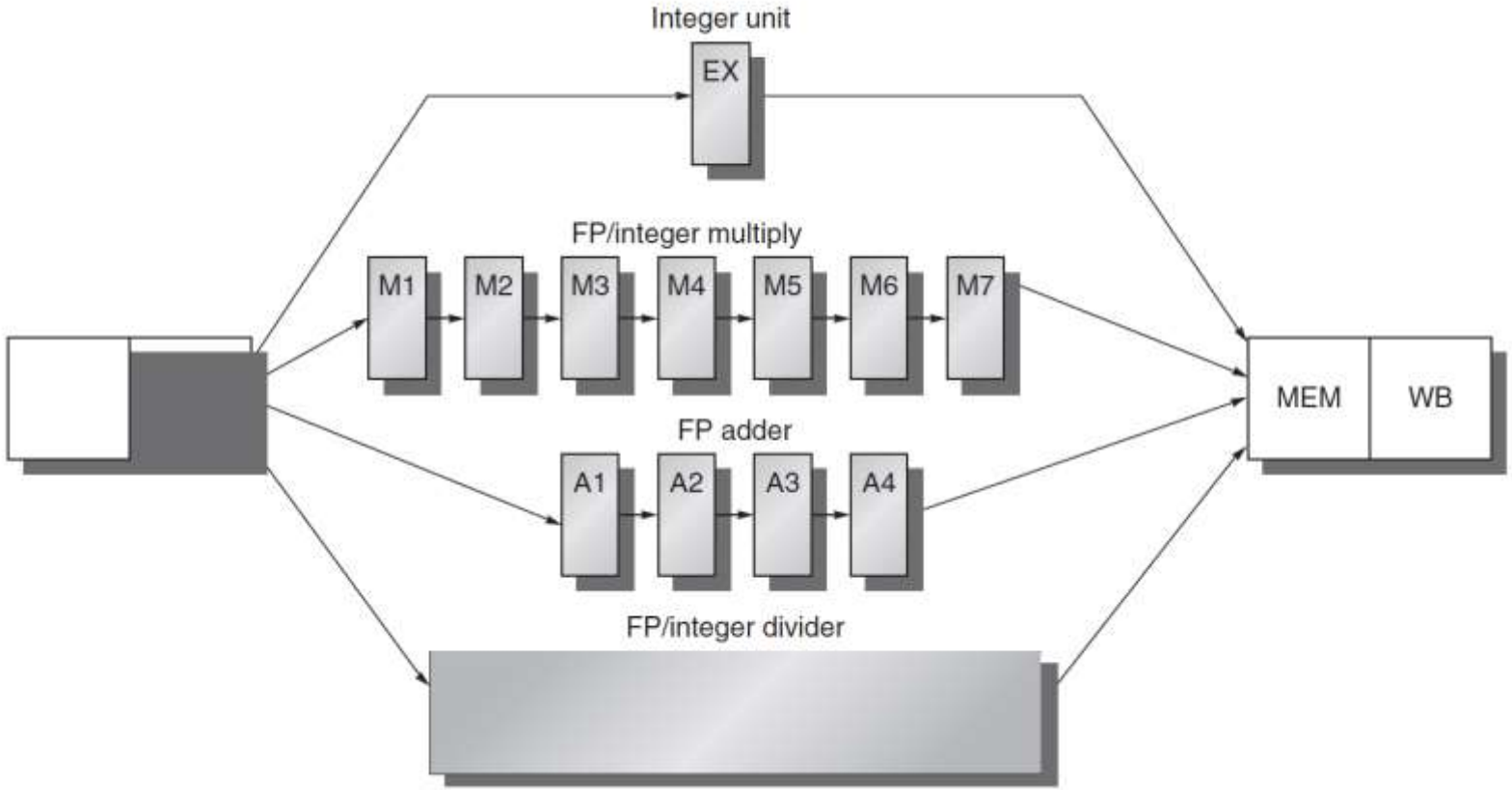
- Same integer pipeline but with 2 differences:
  - The EX stage may be repeated n times, as needed
  - Several FP functional units, each with its own functionality

- Stalls can occur if the instruction issued causes structural hazard to the unit it uses, or data hazard.

# Function units

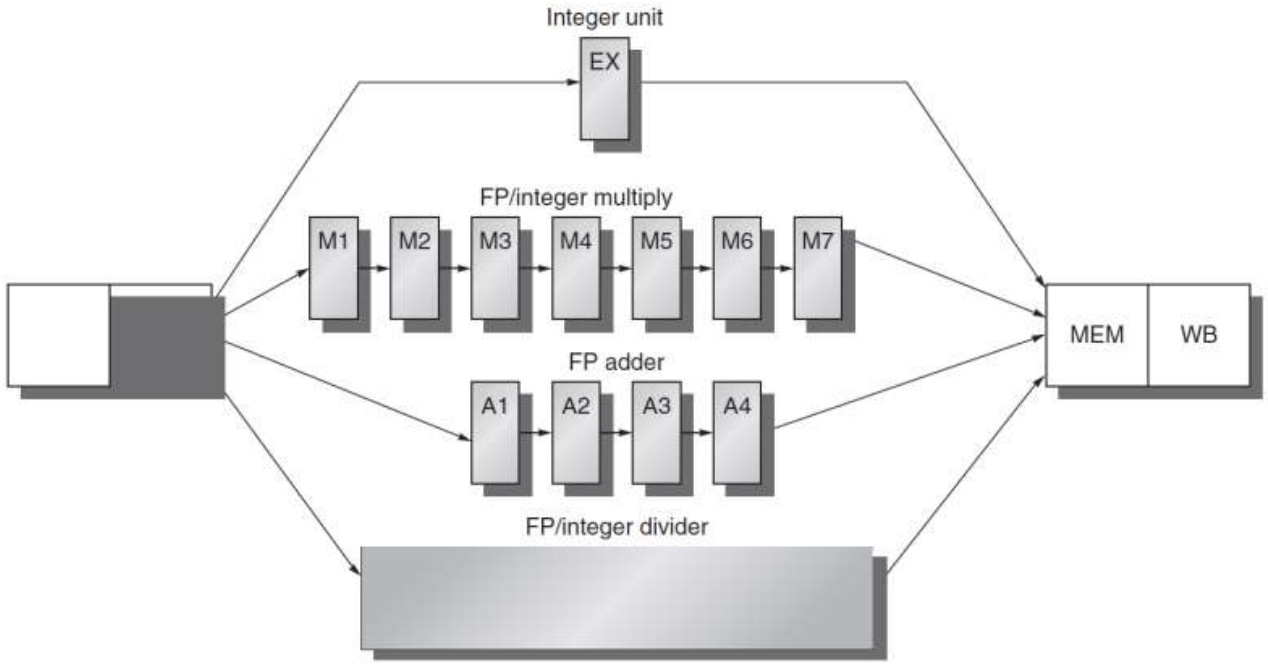
- Assume we have 4 function units:
  1. The **main integer unit** that handles loads and stores, integer ALU operations, and branches
  2. FP and integer **multiplier**
  3. FP **adder** that handles FP add, subtract
  4. FP and integer **divider**

# Adding the floating point hardware




- In this implementation, the **FP multiplier** is pipelined into 7 stages, the **FP adder** into 4 stages, but the **FP divider** is not pipelined and requires 24 cycles to complete.

# Adding the floating point hardware (2)




- The latency in instructions between the issue of an FP operation and the use of the result of that operation is determined by the number of cycles spent in the execution stages.
- For example, the **fourth instruction after an FP add** can use the result of the FP add.
- For integer ALU operations, the depth of the execution pipeline is always one and the next instruction can use the results (seen before)
- Because the **divide unit** is not fully pipelined, structural hazards can occur. These will need to be detected and issuing instructions will need to be stalled.

# Pipeline timing of FP operations

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MUL.D | IF | ID | *M1* | M2 | M3 | M4 | M5 | M6 | **M7** | MEM | WB |
| ADD.D | | IF | ID | *A1* | A2 | A3 | **A4** | **MEM** | WB | | |
| L.D | | | IF | ID | *EX* | **MEM** | WB | | | | |
| S.D | | | | IF | ID | *EX* | *MEM* | WB | | | |

- The stages in *italics* show where data is needed.
- The stages in **bold** show where result is available.
- Remember the .D means double precision float (64 bits)

# Example 1

- Consider this code and fill its timing sequence.

| | Clock cycle number | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Instruction** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** |
| `L.D F4,0(R2)` | IF | ID | EX | MEM | WB | | | | | | | | | | | | |
| `MUL.D F0,F4,F6` | | | | | | | | | | | | | | | | | |
| `ADD.D F2,F0,F8` | | | | | | | | | | | | | | | | | |
| `S.D F2,0(R2)` | | | | | | | | | | | | | | | | | |

# End of lecture 3