

# CS510 Computer Architecture

## Lecture 06: CPU Pipelining II

**Soontae Kim**

**Spring 2017**

**School of Computing, KAIST**

# Notice

- **Homework assignment#1**
  - Due on March 31 (Friday)
  - Available on class web site

# Static Compiler Instruction Scheduling (Re-Ordering) for Data Hazard Stall Reduction

- Many types of stalls resulting from data hazards are very frequent. For example:

$$A = B + C$$

|  |
|--|
| Static = At compilation time by the compiler<br>Dynamic = At run time by hardware in the CPU |
|--|

produces a stall when loading the second data value (C).

- Rather than allow the pipeline to stall, the compiler could sometimes schedule the pipeline to avoid stalls.
- Compiler instruction scheduling involves rearranging the code sequence (instruction reordering) to eliminate or reduce the number of stall cycles.

# Static Compiler Instruction Scheduling Example

- For the code sequence:

$a = b + c$

$d = e - f$

a, b, c, d, e, and f  
are in memory

- Assuming loads have a latency of one clock cycle, compiler schedule eliminates stalls:

Original code with stalls:

*Stall* → LD Rb,b  
LD **Rc**,c  
DADD Ra,Rb,**Rc**  
SD Ra,a  
LD Re,e  
*Stall* → LD **Rf**,f  
DSUB Rd,Re,**Rf**  
SD Rd,d

read after write dependency

Scheduled code with no stalls:

LD Rb,b  
LD Rc,c  
**LD Re,e**  
DADD Ra,Rb,Rc  
LD Rf,f  
**SD Ra,a**  
DSUB Rd,Re,Rf  
SD Rd,d

2 stalls for original code

No stalls for scheduled code

# Control Hazards

- When a conditional branch is executed it may change the PC and, without any special measures, leads to stalling the pipeline for a number of cycles until the branch condition is known (branch is resolved).  
 =freezing the pipeline
  - Otherwise the PC may not be correct when needed in IF
- In current MIPS pipeline, the conditional branch is resolved in stage 4 (MEM stage) resulting in three stall cycles as shown below:

| Branch instruction   | IF | ID    | EX    | MEM   | WB |    |    |    |     |     |
|----------------------|----|-------|-------|-------|----|----|----|----|-----|-----|
| Branch successor     |    | stall | stall | stall |    | IF | ID | EX | MEM | WB  |
| Branch successor + 1 |    |       |       |       |    | IF | ID | EX | MEM | WB  |
| Branch successor + 2 |    |       |       |       |    |    | IF | ID | EX  | MEM |
| Branch successor + 3 |    |       |       |       |    |    |    | IF | ID  | EX  |
| Branch successor + 4 |    |       |       |       |    |    |    |    | IF  | ID  |
| Branch successor + 5 |    |       |       |       |    |    |    |    |     | IF  |

Assuming we stall the pipeline on a branch instruction:

Three clock cycles are wasted for every branch for current MIPS pipeline

Branch Penalty = stage number where branch is resolved - 1

here Branch Penalty = 4 - 1 = 3 Cycles

# Reducing Branch Stall Cycles

**Hardware measures to reduce branch stall cycles:**

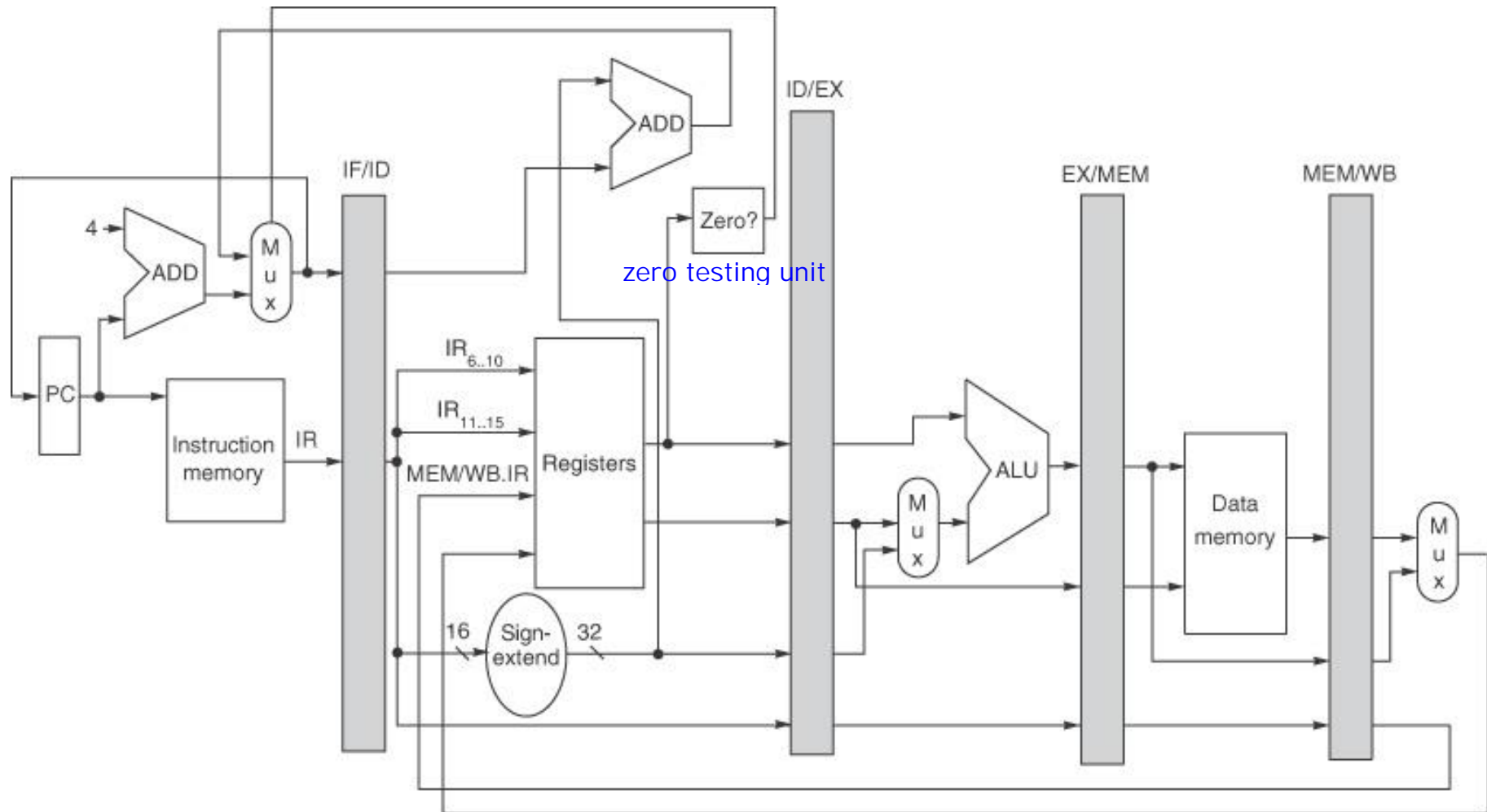
- 1- Find out whether a branch is taken earlier in the pipeline.**
- 2- Compute the taken PC earlier in the pipeline.**

**In MIPS:**

- In MIPS branch instructions BEQZ, BNE, test registers for equality to zero.**
- This can be completed in the ID cycle by moving the zero test into that cycle.**
- Both PCs (taken and not taken) must be computed early.**
- Requires an additional adder because the current ALU is not usable until EX stage.**
- This results in just a single stall cycle on branches.**

**Branch resolved in stage 2 (ID)**  
**Branch Penalty = 2 - 1 = 1**

**Modified MIPS Pipeline:**  
**Conditional Branches**  
**Completed in ID Stage**



# *How to handle branches in a pipelined CPU?*

- 1 One scheme discussed earlier is to freeze the pipeline whenever a conditional branch is decoded by holding or deleting any instructions in the pipeline until the branch destination is known (zero pipeline registers, control lines).
- 2 Another method is to predict that the branch will not be taken where the state of the machine is not changed until the branch outcome is definitely known. Execution here continues with the next sequential instruction; stall occurs here when the branch is taken.
- 3 Another method is to predict that the branch will be taken and begin fetching and executing at the target; stall occurs here if the branch is not taken. (*harder to implement, more on this later*).
- 4 Delayed Branch: An instruction following the branch in a branch delay slot is executed whether the branch is taken or not (part of the ISA).



# Predict Branch Not-Taken Scheme

## Not Taken Branch (no stall)

|                            |    |    |    |     |     |     |           |
|----------------------------|----|----|----|-----|-----|-----|-----------|
| Untaken branch instruction | IF | ID | EX | MEM | WB  |     |           |
| Instruction $i + 1$        |    | IF | ID | EX  | MEM | WB  |           |
| Instruction $i + 2$        |    |    | IF | ID  | EX  | MEM | WB        |
| Instruction $i + 3$        |    |    |    | IF  | ID  | EX  | MEM WB    |
| Instruction $i + 4$        |    |    |    |     | IF  | ID  | EX MEM WB |

## Taken Branch (stall)

|                          |    |    |      |      |      |      |           |
|--------------------------|----|----|------|------|------|------|-----------|
| Taken branch instruction | IF | ID | EX   | MEM  | WB   |      |           |
| Instruction $i + 1$      |    | IF | idle | idle | idle | idle |           |
| Branch target            |    |    | IF   | ID   | EX   | MEM  | WB        |
| Branch target + 1        |    |    |      | IF   | ID   | EX   | MEM WB    |
| Branch target + 2        |    |    |      |      | IF   | ID   | EX MEM WB |

Assuming the MIPS pipeline with reduced branch penalty = 1

The predict-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom).

## Stall when the branch is taken

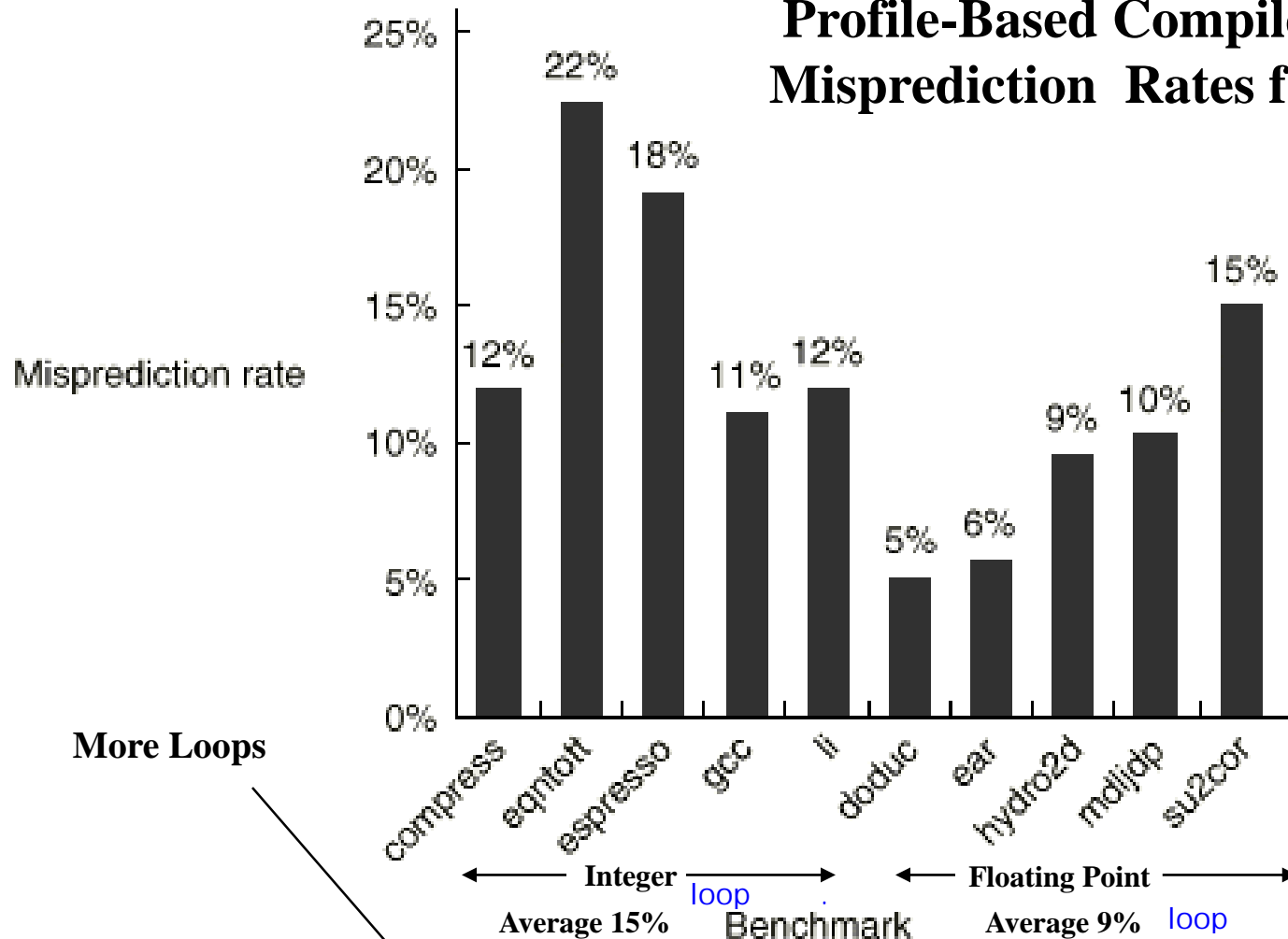
**Pipeline stall cycles from branches = frequency of taken branches X branch penalty**

# **Static Compiler Branch Prediction**

- **Static Branch prediction encoded in branch instructions using one prediction bit, 0 = Not Taken, 1 = Taken**
  - **Must be supported by ISA, Ex: HP PA-RISC, PowerPC, UltraSPARC**
- **Two basic methods exist to statically predict branches at compile time:**
  - 1 **By examination of program behavior and the use of information collected from earlier runs of the program.**
    - **For example, a program profile may show that most forward branches and backward branches (often forming loops) are taken. The simplest scheme in this case is to just predict the branch as taken.**
  - 2 **To predict branches on the basis of branch direction, choosing backward branches as taken and forward branches as not taken.**

(Static)

## Profile-Based Compiler Branch Misprediction Rates for SPEC92



More Loops

Misprediction rate for a profile-based predictor varies widely but is generally better for the FP programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%.

# Reduction of Branch Penalties:

## Delayed Branch

- When delayed branch is used, the branch is delayed by  $n$  cycles, following this execution pattern:

conditional branch instruction  
sequential successor<sub>1</sub>  
sequential successor<sub>2</sub>  
.....  
sequential successor<sub>n</sub>  
branch target if taken

n branch delay slots

- The sequential successor instructions are said to be in the **branch delay slots**. **These instructions are executed whether the branch is taken or not.**
- In Practice, all machines that utilize delayed branches have **a single branch delay slot**. (All RISC ISAs)
- The job of the compiler is to make the successor instruction in the delay slot a valid and useful instruction.

# Delayed Branch Example

**Not Taken Branch (no stall)**

|                                      |    |    |    |     |     |     |     |        |
|--------------------------------------|----|----|----|-----|-----|-----|-----|--------|
| Untaken branch instruction           | IF | ID | EX | MEM | WB  |     |     |        |
| Branch delay instruction ( $i + 1$ ) |    | IF | ID | EX  | MEM | WB  |     |        |
| Instruction $i + 2$                  |    |    | IF | ID  | EX  | MEM | WB  |        |
| Instruction $i + 3$                  |    |    |    | IF  | ID  | EX  | MEM | WB     |
| Instruction $i + 4$                  |    |    |    |     | IF  | ID  | EX  | MEM WB |

**Taken Branch (no stall)**

|                                      |    |    |    |     |     |     |     |        |
|--------------------------------------|----|----|----|-----|-----|-----|-----|--------|
| Taken branch instruction             | IF | ID | EX | MEM | WB  |     |     |        |
| Branch delay instruction ( $i + 1$ ) |    | IF | ID | EX  | MEM | WB  |     |        |
| Branch target                        |    |    | IF | ID  | EX  | MEM | WB  |        |
| Branch target + 1                    |    |    |    | IF  | ID  | EX  | MEM | WB     |
| Branch target + 2                    |    |    |    |     | IF  | ID  | EX  | MEM WB |

**The behavior of a delayed branch is the same whether or not the branch is taken.**

**Single Branch Delay Slot Used**

**Assuming branch penalty = 1 cycle**

# Branch-delay Slot Scheduling Strategies

The branch-delay slot instruction can be chosen from three cases:

**A** An independent instruction from before the branch:  
Always improves performance when used. The branch must not depend on the rescheduled instruction.

Common

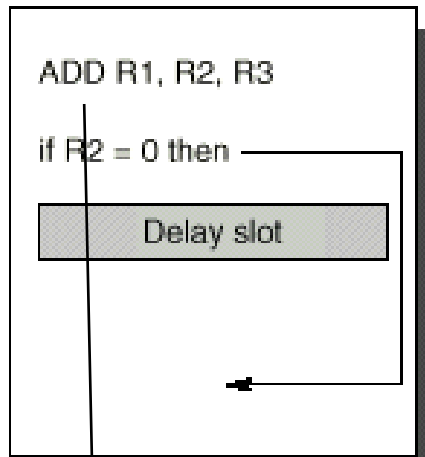
↑  
Hard  
to  
Find  
↓  
**B** An instruction from the target of the branch:  
Improves performance if the branch is taken and may require instruction duplication. This instruction must be safe to execute if the branch is not taken.

**C** An instruction from the fall through instruction stream:  
Improves performance when the branch is not taken. The instruction must be safe to execute when the branch is taken.

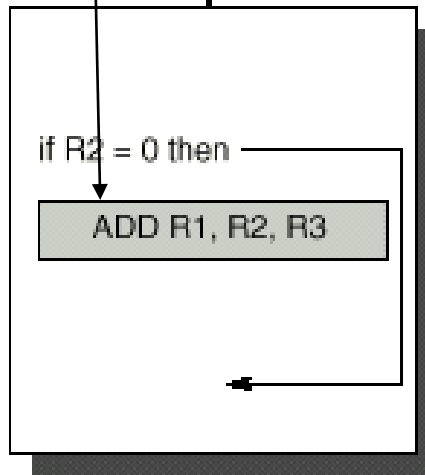
The performance and usability of cases **B**, **C** is improved by using a canceling or nullifying branch.

**Example:  
From the  
body of a loop**

**(A)** From before

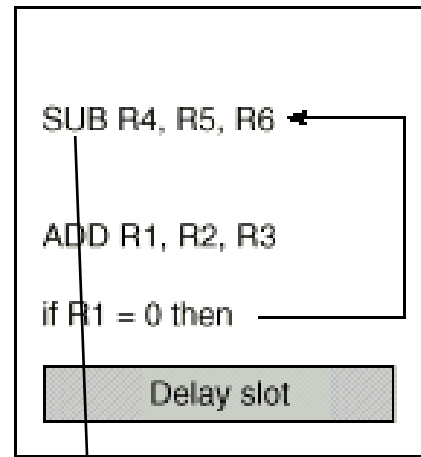


Becomes

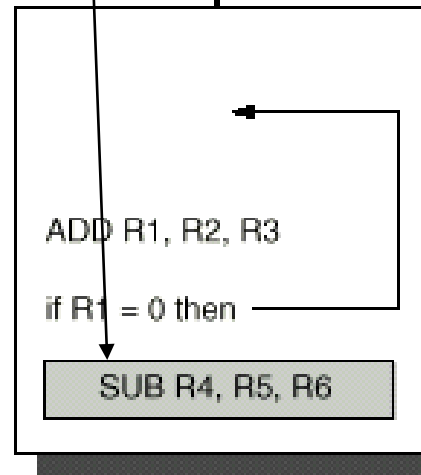


Most Common

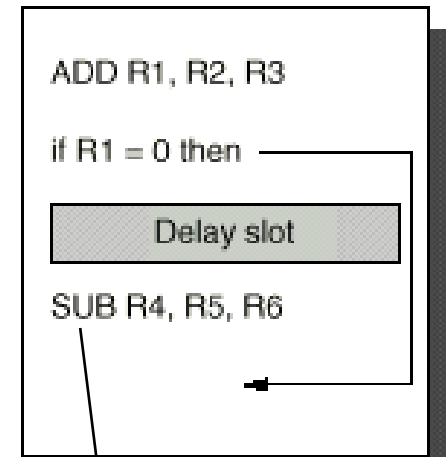
**(B)** From target



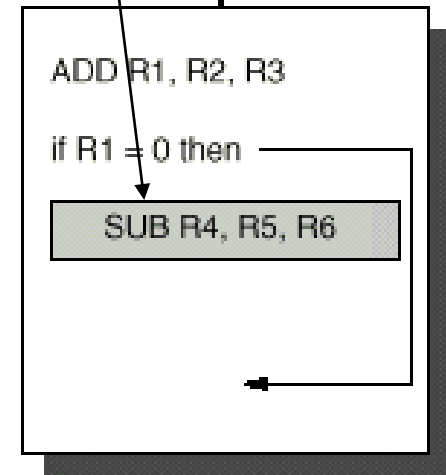
Becomes



**(C)** From fall through



Becomes



**Scheduling the branch-delay slot.**

# Branch-delay Slot: Canceling Branches

- In a canceling branch, a static compiler branch direction prediction is included with the branch-delay slot instruction.
- When the branch goes as predicted, the instruction in the branch delay slot is executed normally.
- When the branch does not go as predicted, the instruction is turned into a no-op (i.e. cancelled).
- Canceling branches eliminate the conditions on instruction selection in strategies **B** and **C**
- The effectiveness of this method depends on whether we predict the branch correctly.



| Scheduling strategy    | Requirements   | Improves performance when?  |
|------------------------|--|---|
| (a) From before branch | Branch must not depend on the rescheduled instructions.  | Always.   |
| (b) From target        | Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions. | When branch is taken. May enlarge program if instructions are duplicated. |
| (c) From fall through  | Must be OK to execute instructions if branch is taken.   | When branch is not taken.   |

### Delayed-branch scheduling schemes and their requirements.

#### Branch Goes Not As Predicted

|                                      |    |    |    |      |      |      |                    |
|--------------------------------------|----|----|----|------|------|------|--------------------|
| Untaken branch instruction           | IF | ID | EX | MEM  | WB   |      |                    |
| Branch delay instruction ( $i + 1$ ) |    | IF | ID | idle | idle | idle | Cancelled or No-OP |
| Instruction $i + 2$                  |    |    | IF | ID   | EX   | MEM  | WB                 |
| Instruction $i + 3$                  |    |    |    | IF   | ID   | EX   | MEM WB             |
| Instruction $i + 4$                  |    |    |    |      | IF   | ID   | EX MEM WB          |

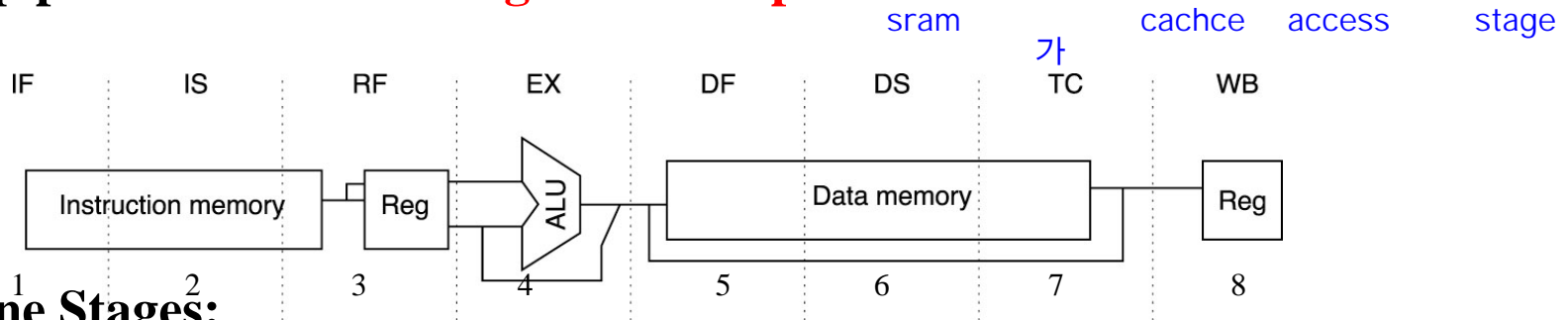
#### Branch Goes As Predicted

|                                      |    |    |    |     |     |     |           |
|--------------------------------------|----|----|----|-----|-----|-----|-----------|
| Taken branch instruction             | IF | ID | EX | MEM | WB  |     |           |
| Branch delay instruction ( $i + 1$ ) |    | IF | ID | EX  | MEM | WB  | Normal    |
| Branch target                        |    |    | IF | ID  | EX  | MEM | WB        |
| Branch target + 1                    |    |    |    | IF  | ID  | EX  | MEM WB    |
| Branch target + 2                    |    |    |    |     | IF  | ID  | EX MEM WB |

FIGURE 3.30 The behavior of a predicted-taken cancelling branch depends on whether the branch is taken or not.

# The MIPS R4000 Integer Pipeline

- Implements MIPS64 but uses an 8-stage pipeline instead of the classic 5-stage pipeline to achieve a **higher clock speed**. instruction memory, data memory stage가



## • Pipeline Stages:

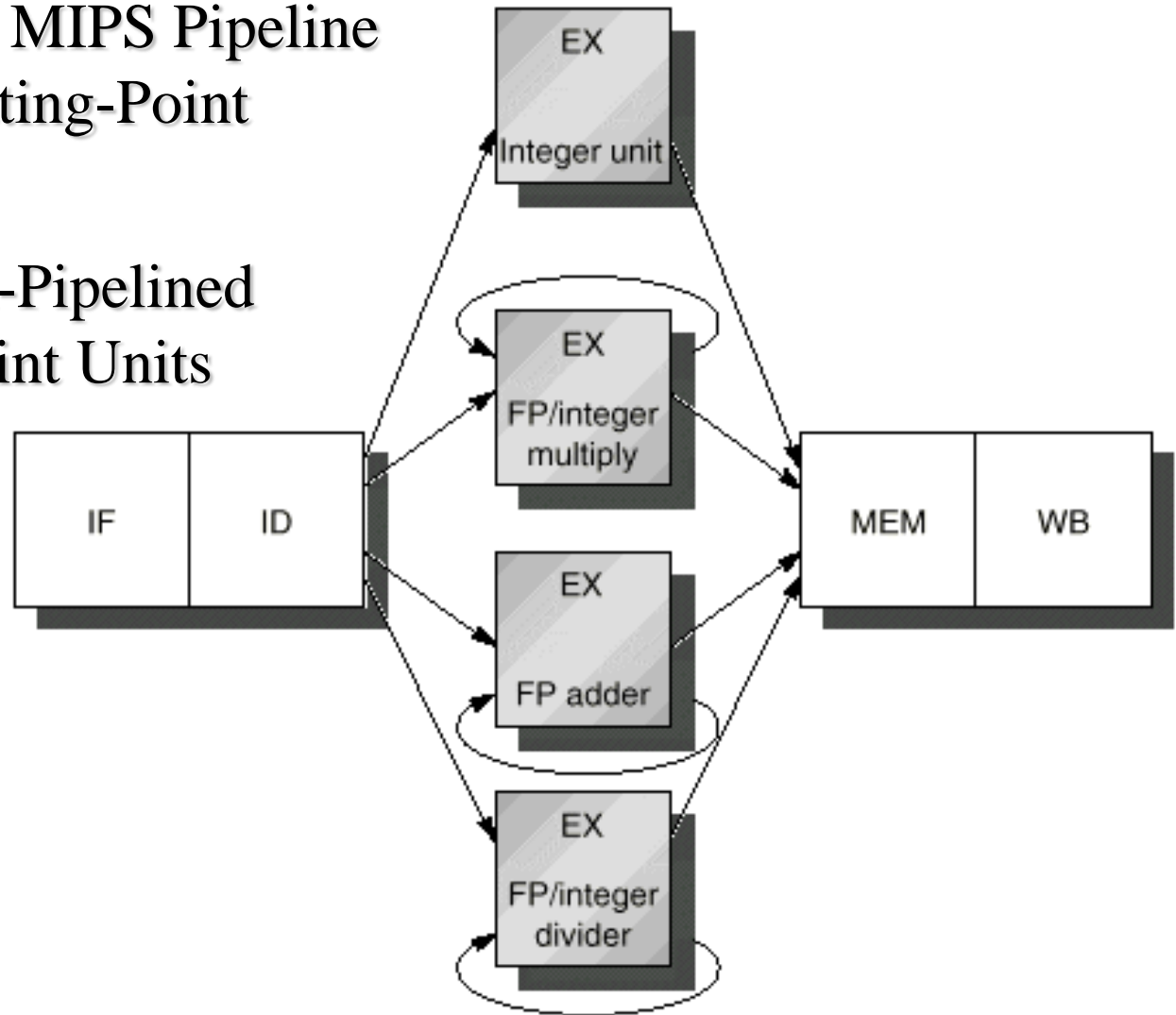
- **IF:** First half of instruction fetch. Start instruction cache access.
- **IS:** Second half of instruction fetch. Complete instruction cache access.
- **RF:** Instruction decode and register fetch, hazard checking.
- **EX:** Execution including branch-target and condition evaluation.
- **DF:** Data fetch, first half of data cache access.
- **DS:** Second half of data fetch access. Complete data cache access. Data available if a cache hit
- **TC:** Tag check, determine data cache access hit.
- **WB:** Write back for loads and register-register operations.
- **Branch resolved in stage 4. Branch Penalty = 3 cycles if taken ( 2 with branch delay slot)**

# Floating Point/Multicycle Pipelining in MIPS

- Completion of MIPS EX stage floating point arithmetic operations in one or two cycles is impractical since it requires:
  - A much longer CPU clock cycle, and/or
  - An enormous amount of logic.
- Instead, the floating-point pipeline will allow for a longer latency (more than 1 EX cycles).
- Floating-point operations have the same pipeline stages as the integer instructions with the following differences:
  - The EX cycle may be repeated as many times as needed (more than 1 cycle).
  - There may be multiple floating-point functional units.
  - A stall will occur if the instruction to be issued either causes a structural hazard for the functional unit or causes a data hazard.
- **The latency of functional units** is defined as the number of intervening cycles between an instruction producing the result and the instruction that uses the result (usually equals stall cycles with forwarding used).
- **The initiation or repeat interval** is the number of cycles that must elapse between issuing instructions of a given type.

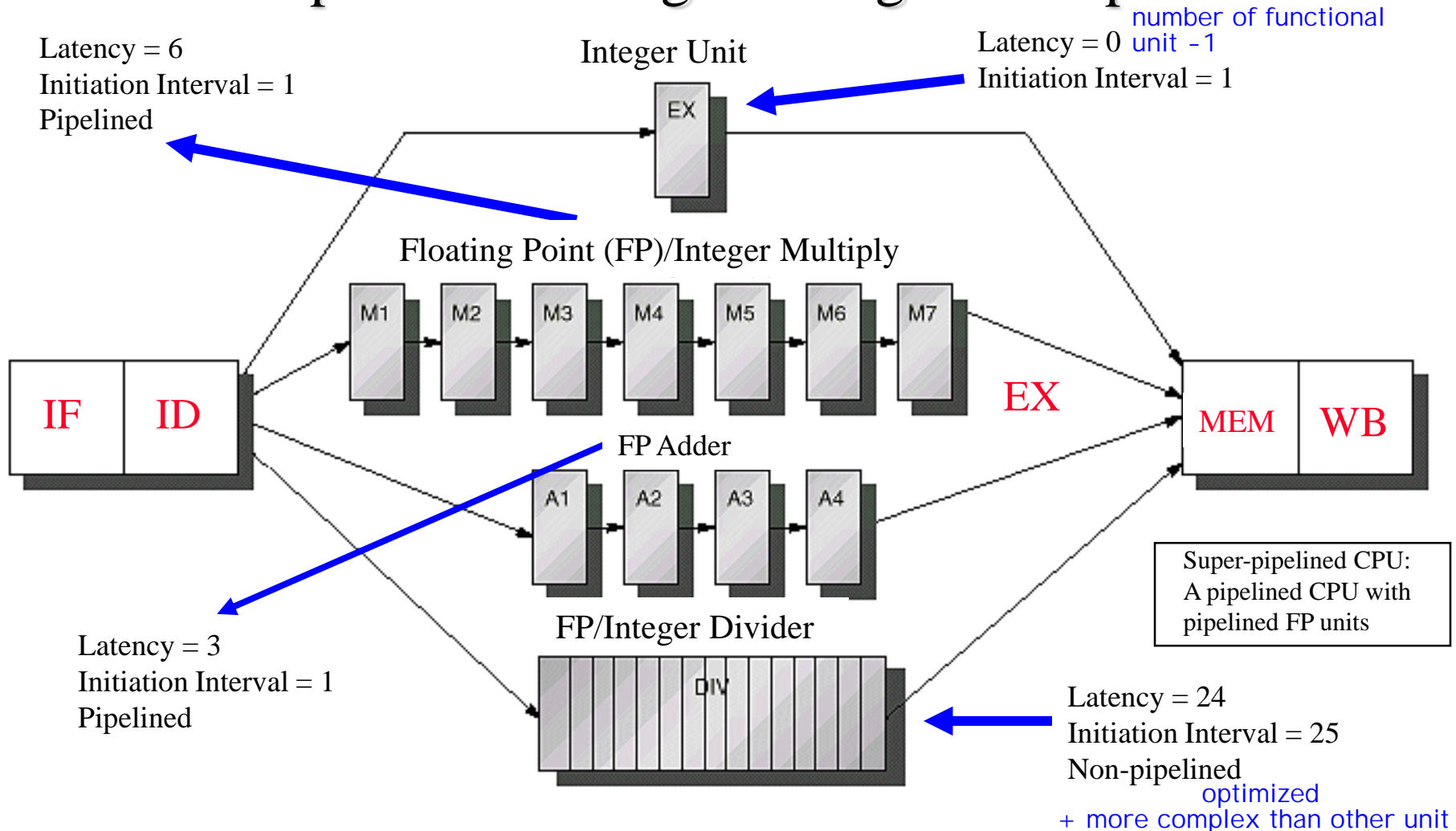
# Extending The MIPS Pipeline to Handle Floating-Point Operations:

## Adding Non-Pipelined Floating Point Units



The MIPS pipeline with three additional unpipelined, floating-point functional units (FP FUs)

# Extending The MIPS Pipeline: Multiple Outstanding Floating Point Operations



**A pipeline that supports multiple outstanding FP operations.**

In-Order Single-Issue MIPS Pipeline with FP Support

# Latencies and Initiation Intervals For Functional Units

| Functional Unit                        | Latency | Initiation Interval |
|--|---------|---------------------|
| Integer ALU                            | 0       | 1                   |
| Data Memory<br>(Integer and FP Loads)  | 1       | 1                   |
| FP add                                 | 3       | 1                   |
| FP multiply<br>(also integer multiply) | 6       | 1                   |
| FP divide<br>(also integer divide)     | 24      | 25                  |

Latency usually equals stall cycles when full forwarding is used