EITF20: Computer Architecture
Part2.2.1: Pipeline-1

Liang Liu
liang.liu@eit.lth.se

Outline

- ☐ **Reiteration**
- ☐ **Pipelining**
- ☐ **Harzards**
  - • Structural hazards
  - • Data hazards
  - • Control hazards
- ☐ **Implementation issues**
- ☐ **Multi-cycle operations**
- ☐ **Summary**

Previous lecture

☐ **Instruction set architecture**

Instruction set principles

☐ **Classification of instruction sets**

☐ **What's needed in an instruction set?**

- Addressing
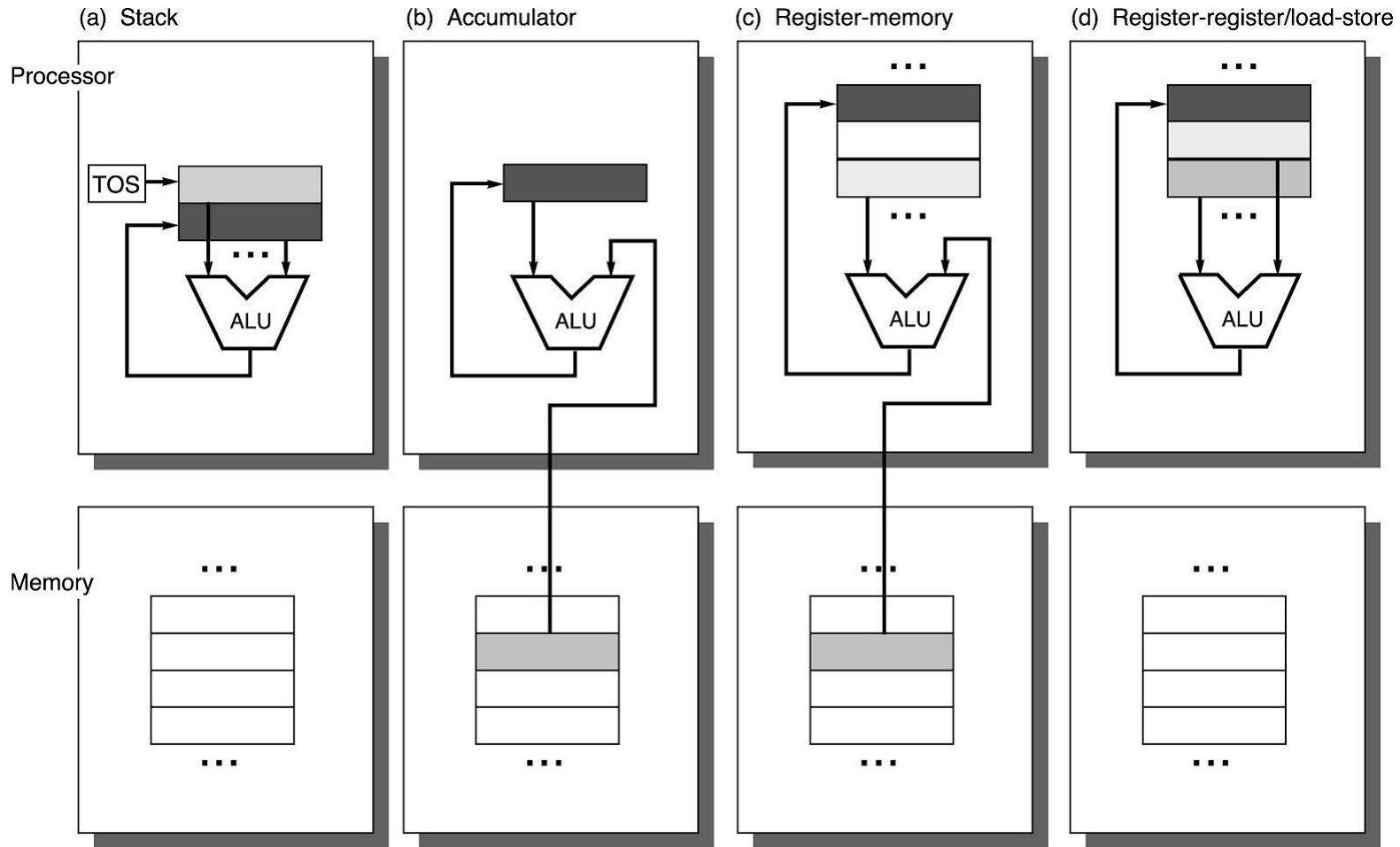- Operands
- Operations
- Control Flow

☐ **Encoding**

☐ **The impact of the compiler**

☐ **The MIPS instruction set architecture**

ISA Classes

(a) Stack   (b) Accumulator   (c) Register-memory   (d) Register-register/load-store

Processor

TOS

ALU

Memory

ISA and compiler

- **Instruction set architecture is a compiler target**
- **By far most instructions executed are generated by a compiler (exception certain special purpose processors)**
- **interaction compiler - ISA critical for overall performance**

Application

Operating System

Compiler

Instruction Set Architecture

Instr. Set Proc.   I/O system

Digital Design

Circuit Design
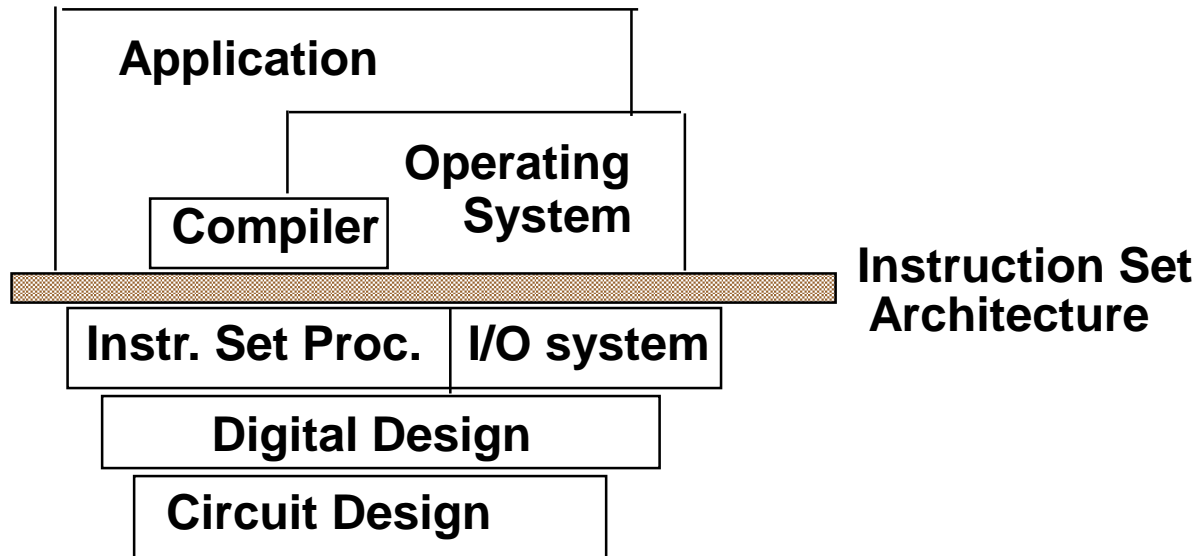
The MIPS64 architecture

☐ **An architecture representative of modern ISA:**

- 64 bit load/store GPR architecture
- 32 general integer registers (R0 = 0) and 32 floating point registers
- Supported data types: bytes, half word (16 bits), word (32 bits), double word (64 bits), single/double precision IEEE floating points
- Memory byte addressable with 64 bit addresses
- Addressing modes: immediate and displacement
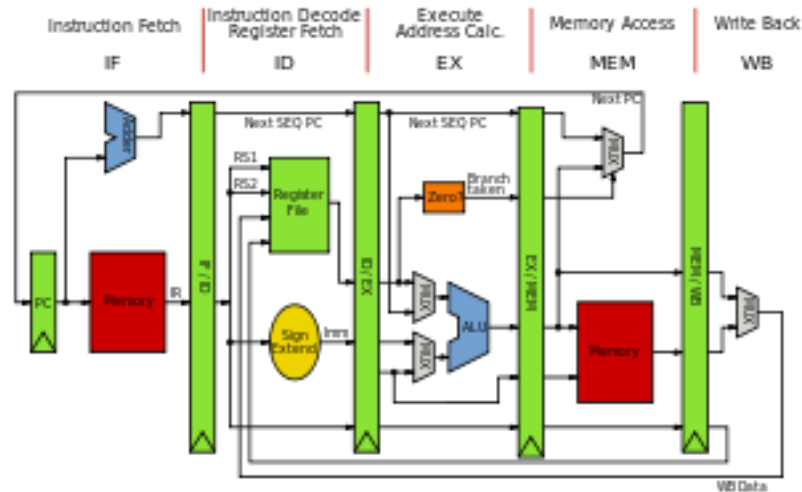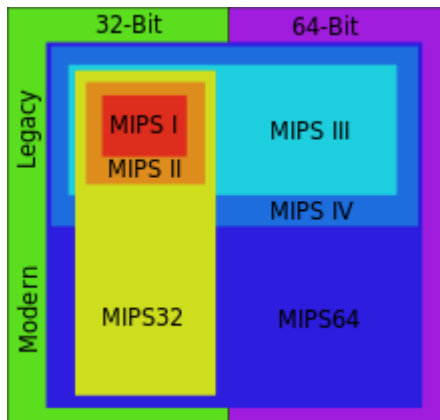
MIPS instructions

☐ **MIPS instructions classes:**

- Arithmetic/logical/shift/comparison
- Control instructions (branch and jump)
- Load/store
- Other (exception, register movement to/from GP registers, etc.)

MIPS instruction example

LW	R1,60(R7)	Load word
SB	R2,41(R5)	Store byte
MUL	R2,R1,R3	Integer multiply
AND	R3,R2,R1	Logical AND
DADDI	R5,R6,#17	Add immediate
J	lable	Jump
BEQZ	R4,lable	Branch if R4 zero
JALR	R7	Procedure call

MIPS instruction format

I-type instruction

6	5	5	16
Opcode	rs	rt	Immediate

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction

6	5	5	5	5	6
Opcode	rs	rt	rd	shamt	funct

Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

J-type instruction

6	26
Opcode	Offset added to PC

Jump and jump and link
Trap and return from exception

Outline

☐ Reiteration

☐ **Pipelining**

☐ Harzards

- Structural hazards
- Data hazards
- Control hazards

☐ Implementation issues

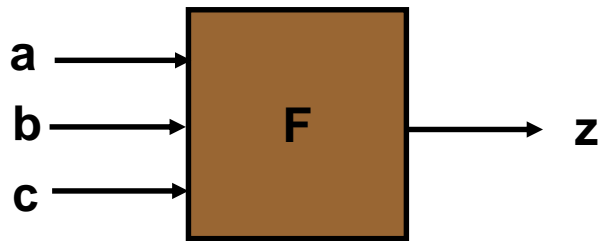☐ Multi-cycle operations

☐ Summary

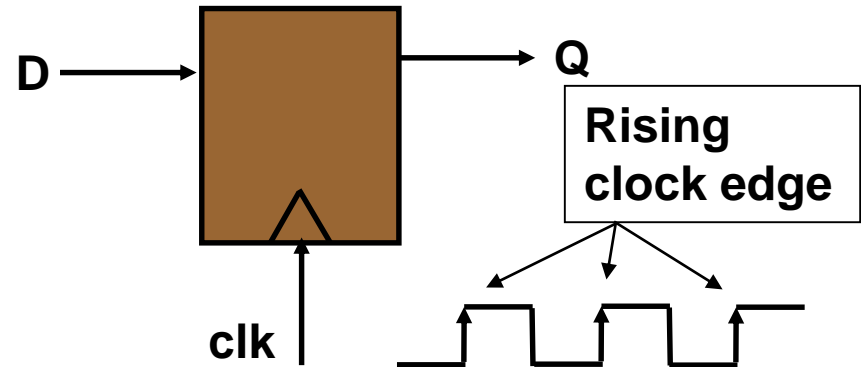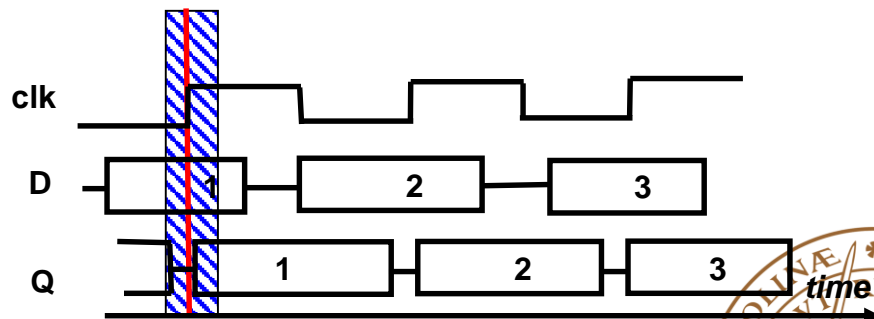# Two basic digital components

Combinational Logic

a →
b → [**F**] → z
c →

Always:

    z <= F(a, b, c);

Propagation delay:
After presenting new inputs
Worst case delay before
producing correct output

Register

D → [ ] → Q

clk

Rising clock edge

if clk' event and clk= '1' then
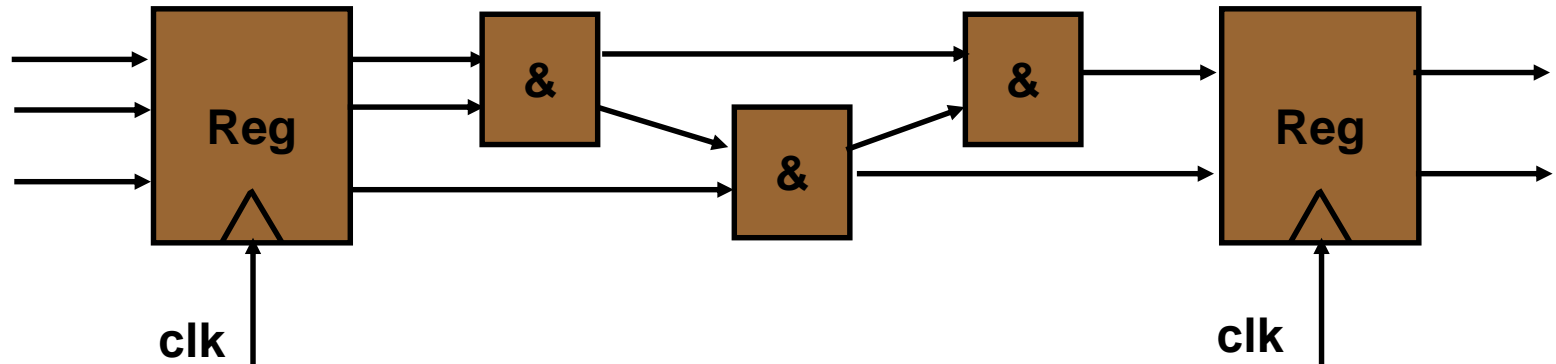   Q <= D;

clk

D   1   2   3

Q   1   2   3

time

Clock Frequency (RTL)

☐ What is the maximum clock frequency?

Register
   Setup time: Tsu   200ps
   Hold time: Th   100ps
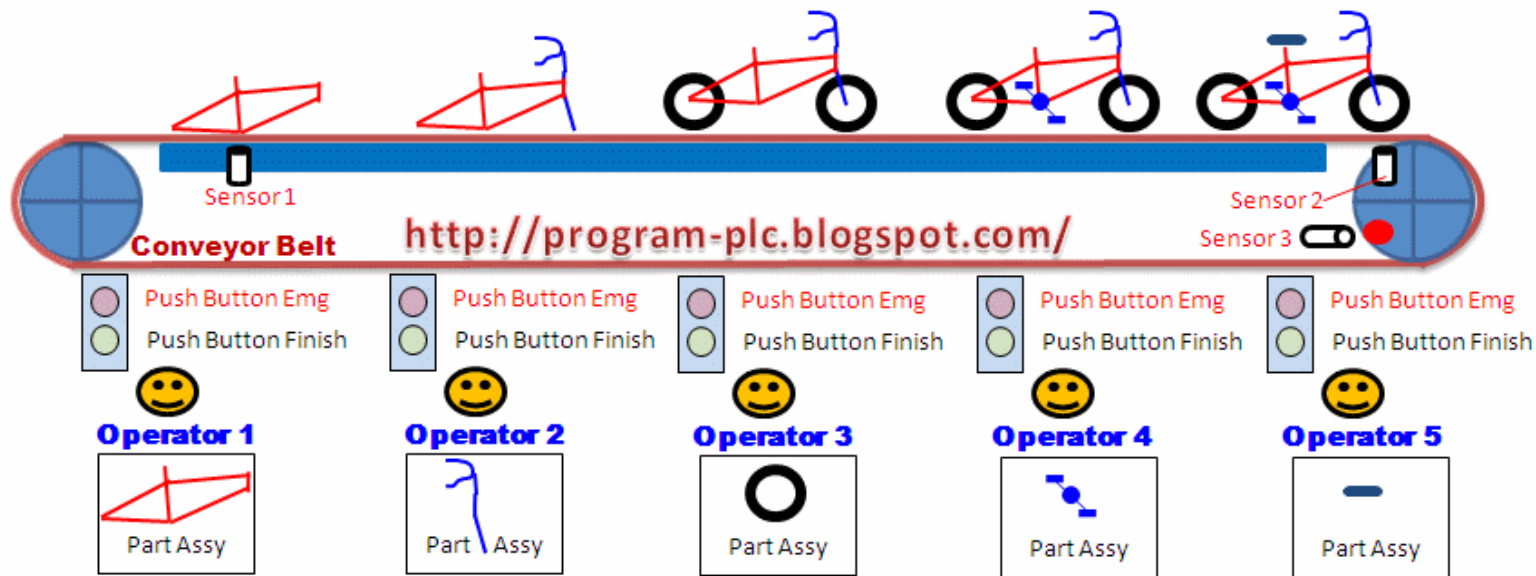**AND-gate**
   Propagation delay: Tprop   250ps
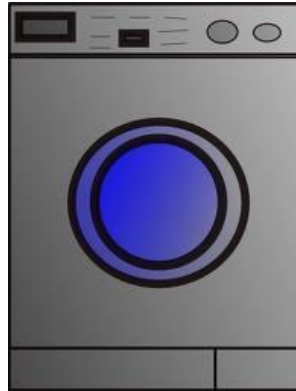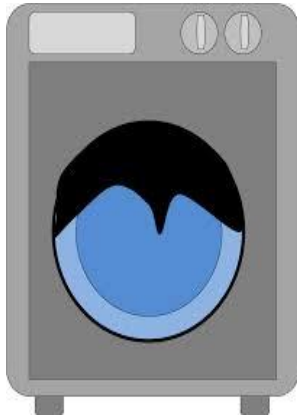
$$250 \times 3 + 200 = 0.95\text{ns}$$
$$f = 1.05\text{GHz}$$

The Assembly Line ...

The general pipeline principle

☐ **Start again from laundry room**

☐ **Small laundry has one washer, one dryer and one folder, it takes 110 minutes to finish one load:**
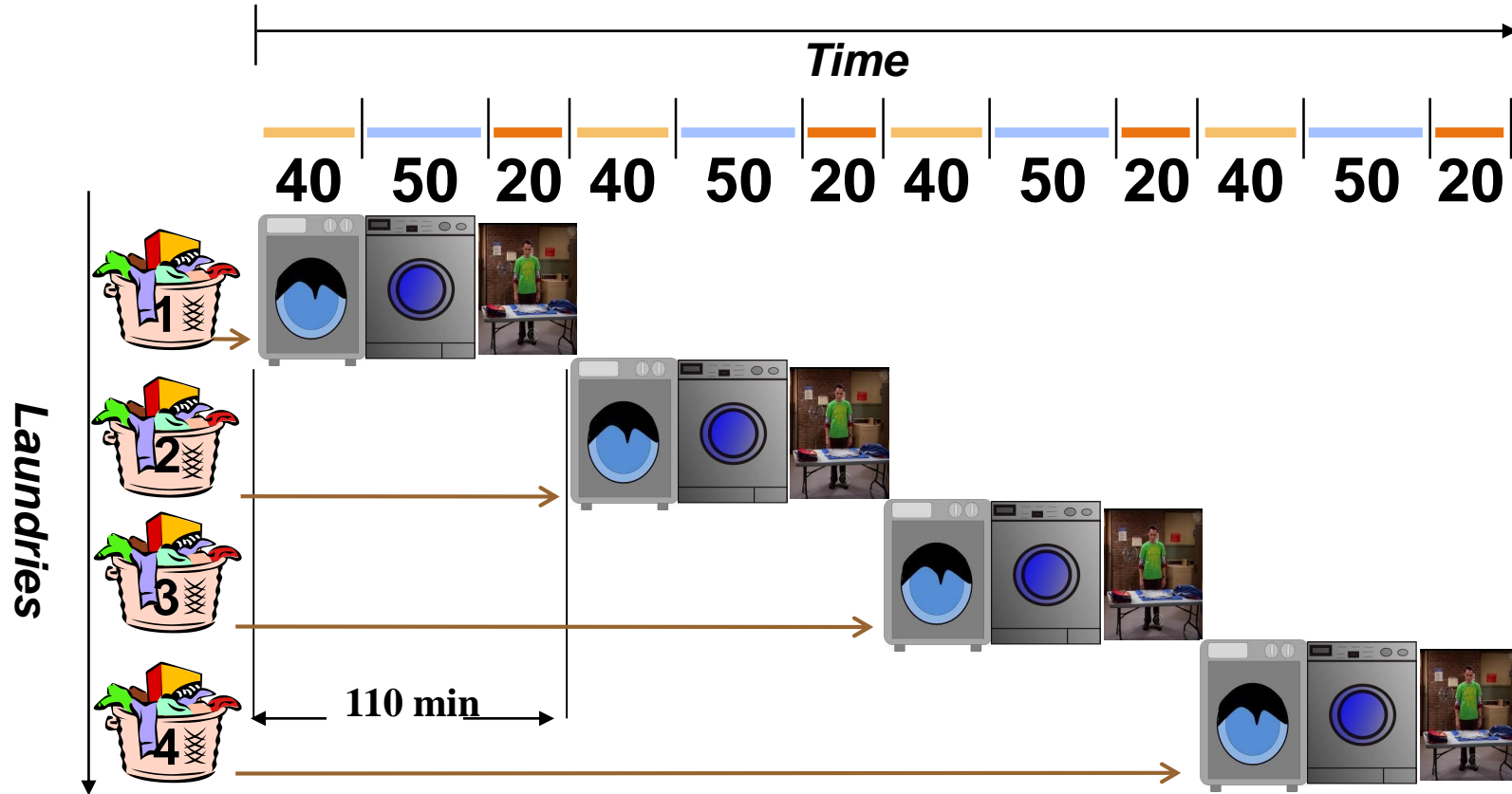
- •Washer takes 40 minutes
- •Dryer takes 50 minutes
- •"Folding" takes 20 minutes

☐ **Need to do 4 laundries**

Not very smart way...

Time

| 40 | 50 | 20 | 40 | 50 | 20 | 40 | 50 | 20 | 40 | 50 | 20 |

Laundries

110 min

Total = N*(Washer+ Dryer+Folder)

= _____**440**_____ mins
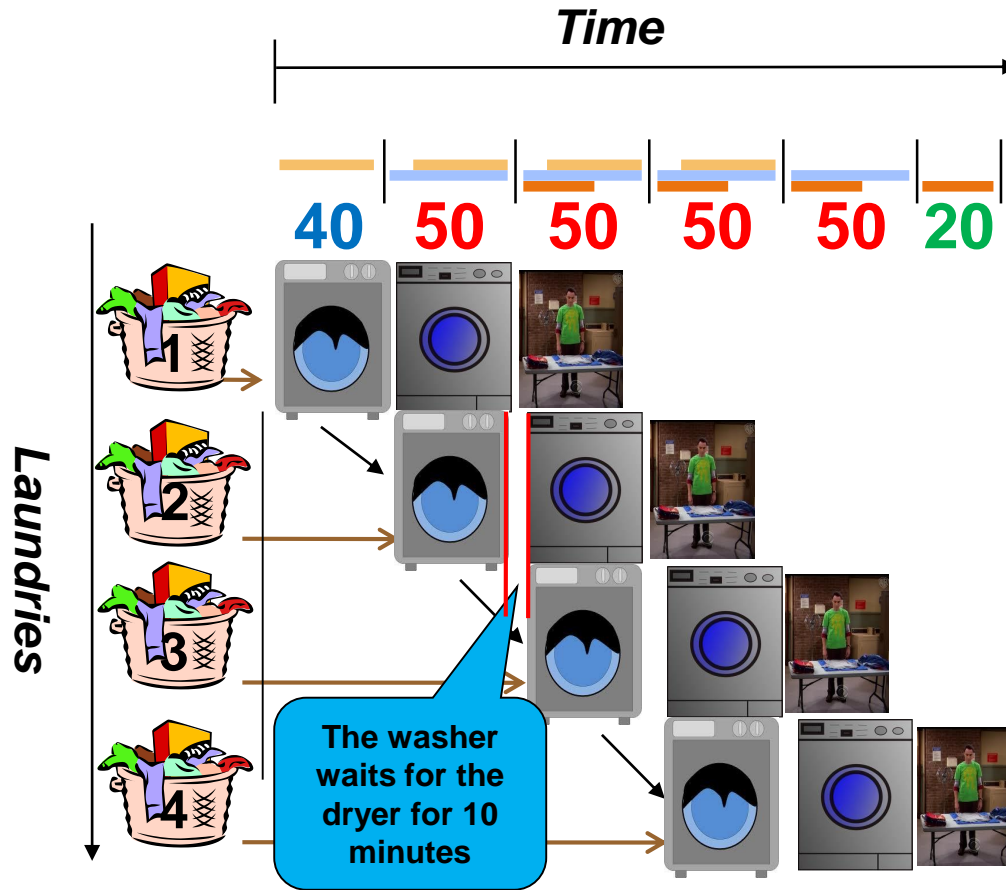
If we pipelining

Total = Washer+N*Max(Washer,Dryer,Folder)+Folder

= _____ 260 _____ mins

Pipeline Facts
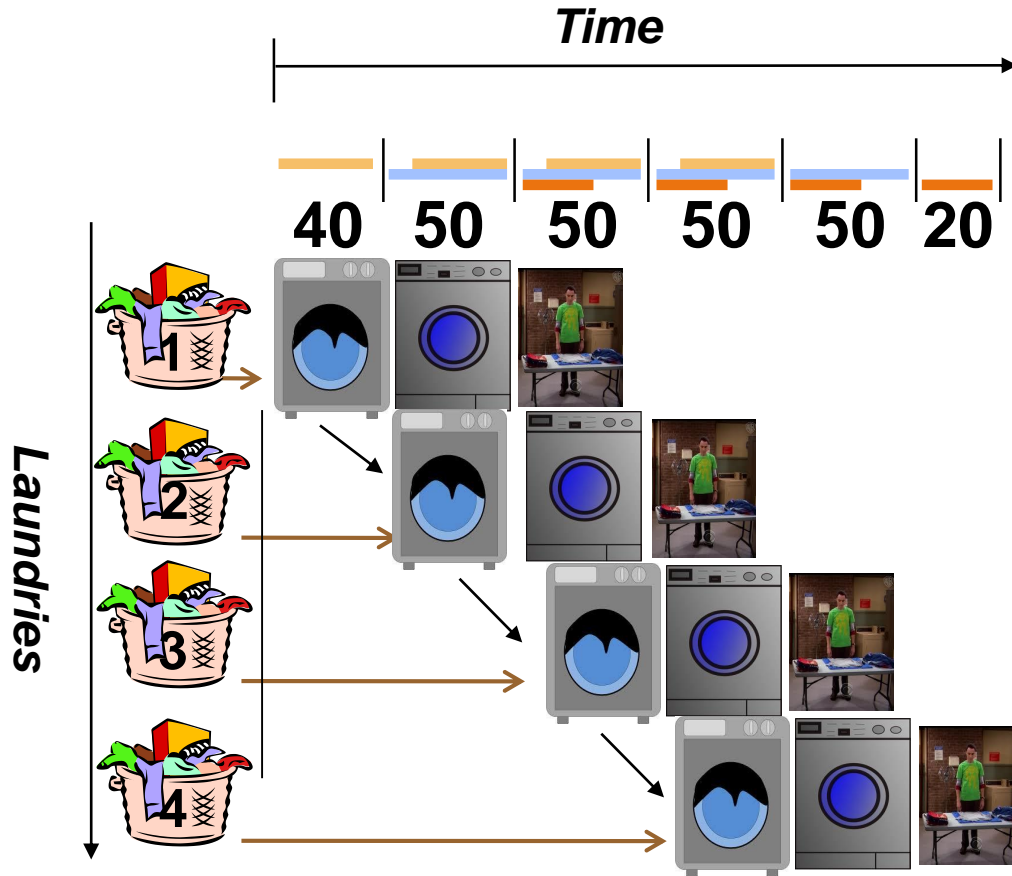
- □ **Multiple** tasks operating simultaneously
- □ **Pipelining doesn't help latency of single task, it helps throughput of entire workload**
- □ **Pipeline rate limited by slowest pipeline stage**
- □ **Unbalanced lengths of pipe stages reduces speedup**
- □ **Potential speedup ∝ Number of pipe stages**

Pipeline timing diagrams
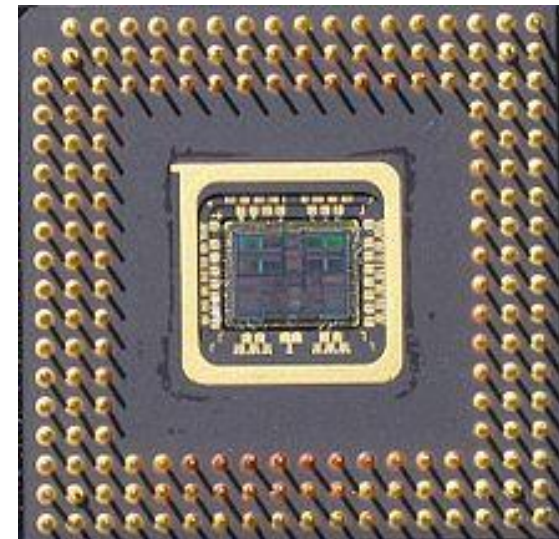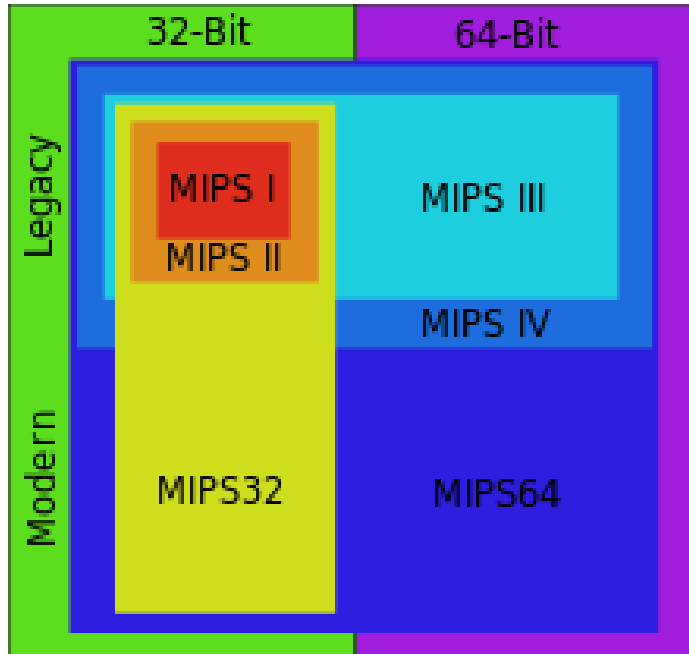
Clock cycle →

Pipeline stages		i	i+1	i+2	i+3	
Input		X_i	X_{i+1}	X_{i+2}	X_{i+3}	…
F Reg			$F(X_i)$	$F(X_{i+1})$	$F(X_{i+2})$	…
G Reg			$G(X_i)$	$G(X_{i+1})$	$G(X_{i+2})$	
H Reg				$H(X_i)$	$H(X_{i+1})$	$H(X_{i+2})$

MIPS pipeline

One core – the MIPS data-path
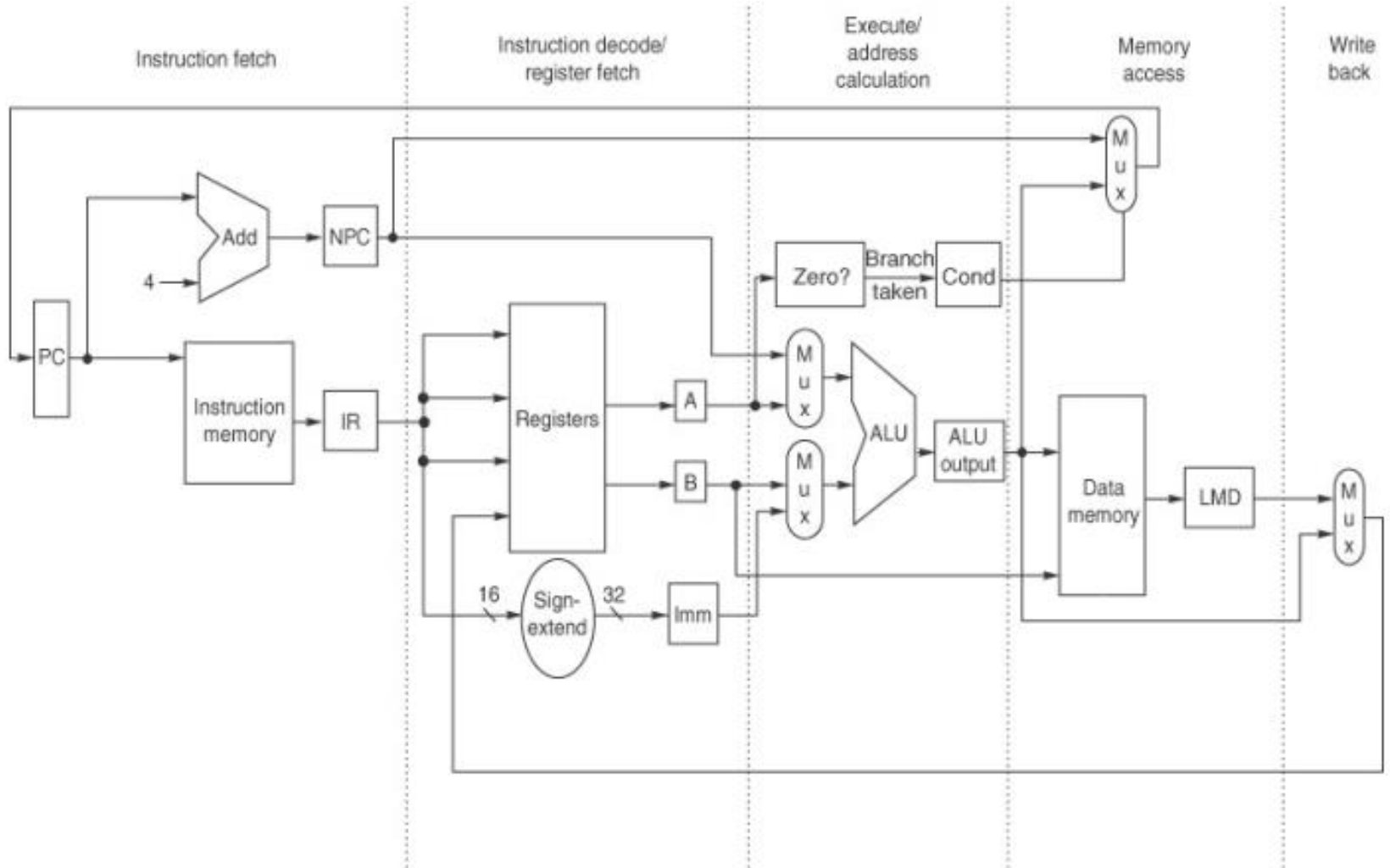
One core – the MIPS data-path
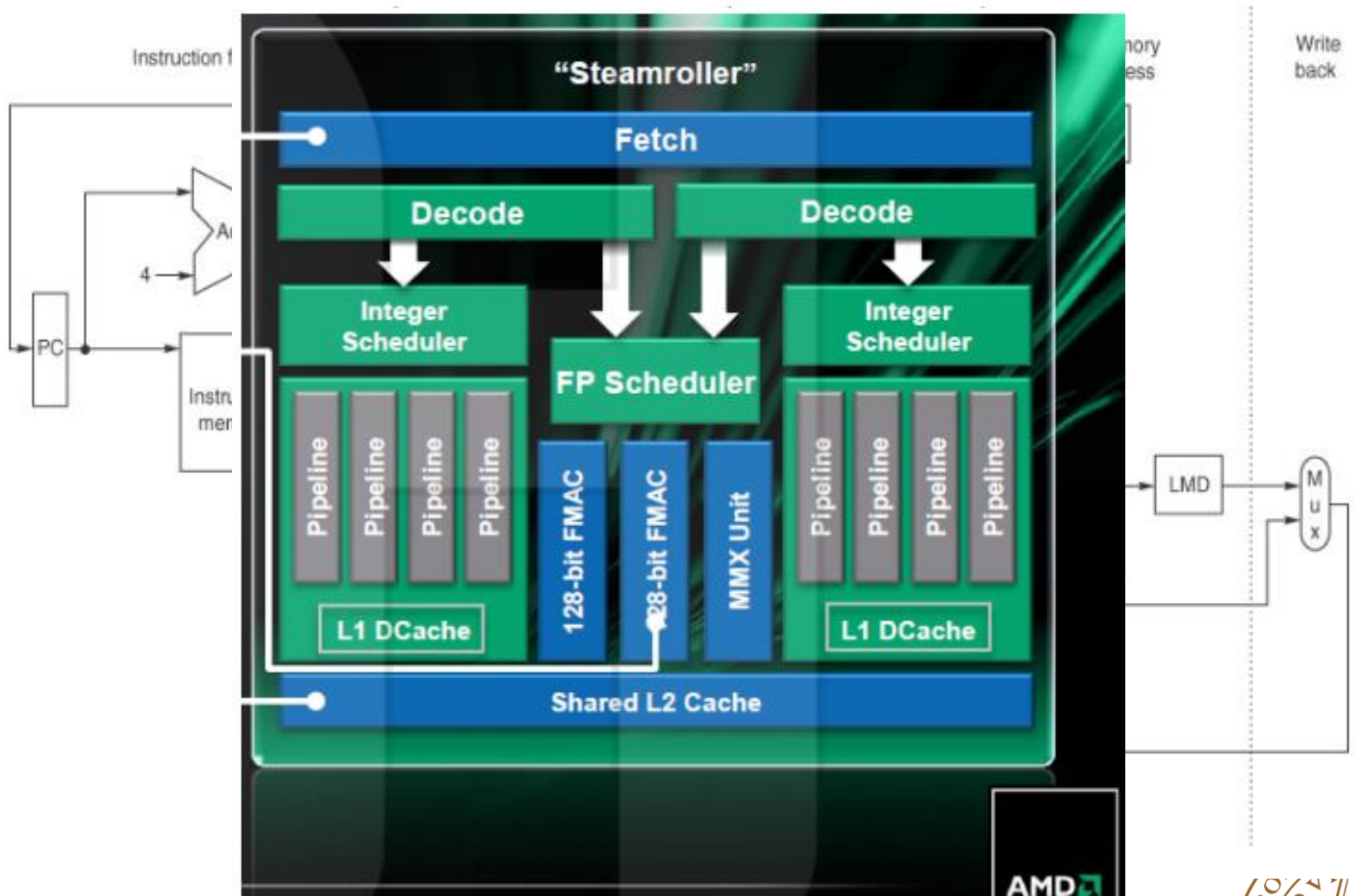
Classic RISC 5-stage pipeline
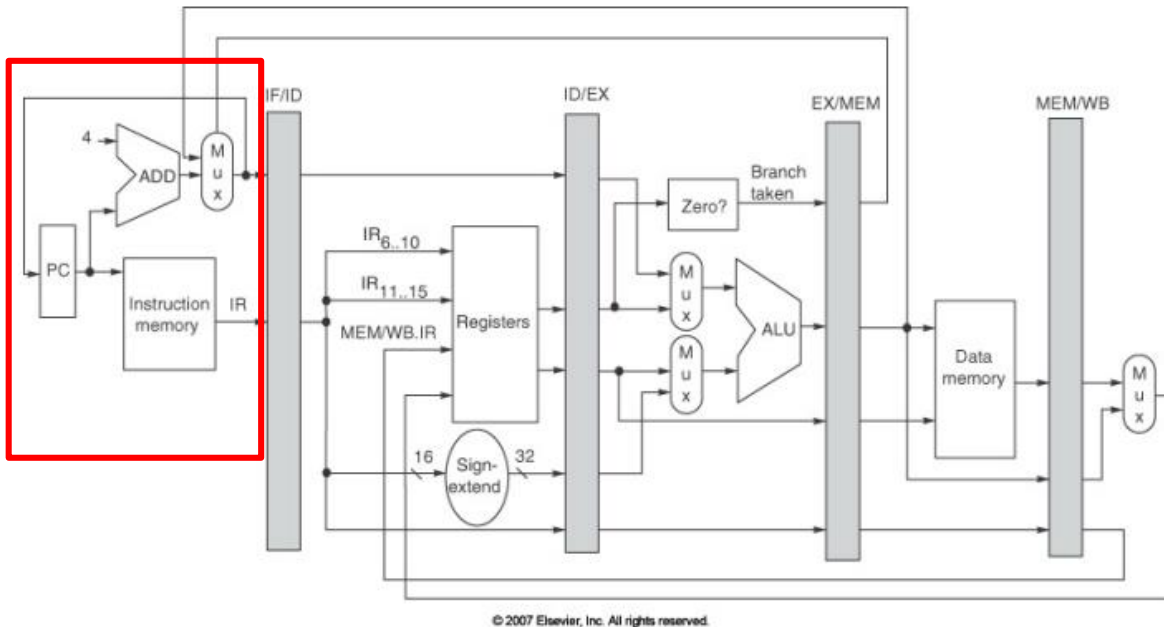
Passed To Next Stage
IR <- Mem[PC]
NPC <- PC + 4

Instruction Fetch (IF):

- Send out the PC and fetch the instruction from memory into the instruction register (IR); increment the PC by 4 to address the next sequential instruction.
- IR holds the instruction that will be used in the next stage.
- NPC holds the value of the next PC (either sequential or jump).

Classic RISC 5-stage pipeline
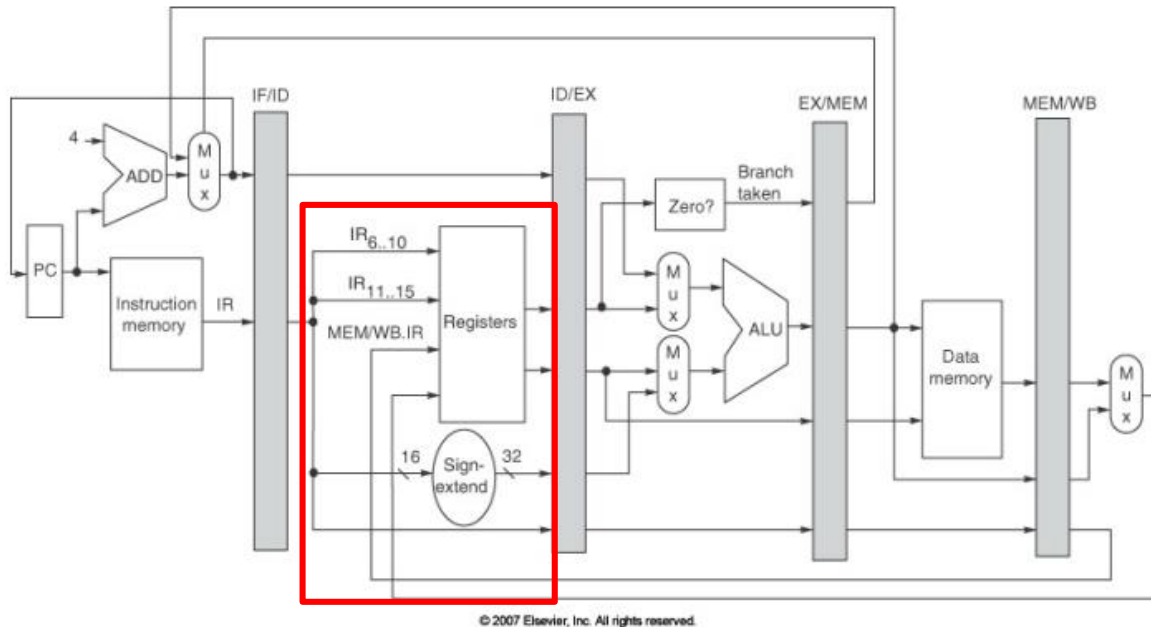
Passed To Next Stage
A <- Regs[IR6..IR10];
B <- Regs[IR10..IR15];
Imm <- ((IR16) ##IR16-31

© 2007 Elsevier, Inc. All rights reserved.

Instruction Decode/Register Fetch Cycle (ID):

- Decode the instruction and access the register file to read the registers.
- The outputs of the general purpose registers are read into two temporary registers (A & B) for use in later clock cycles.
- Extend the sign of the lower 16 bits of the Instruction Register (immediate).

Classic RISC 5-stage pipeline

© 2007 Elsevier, Inc. All rights reserved.
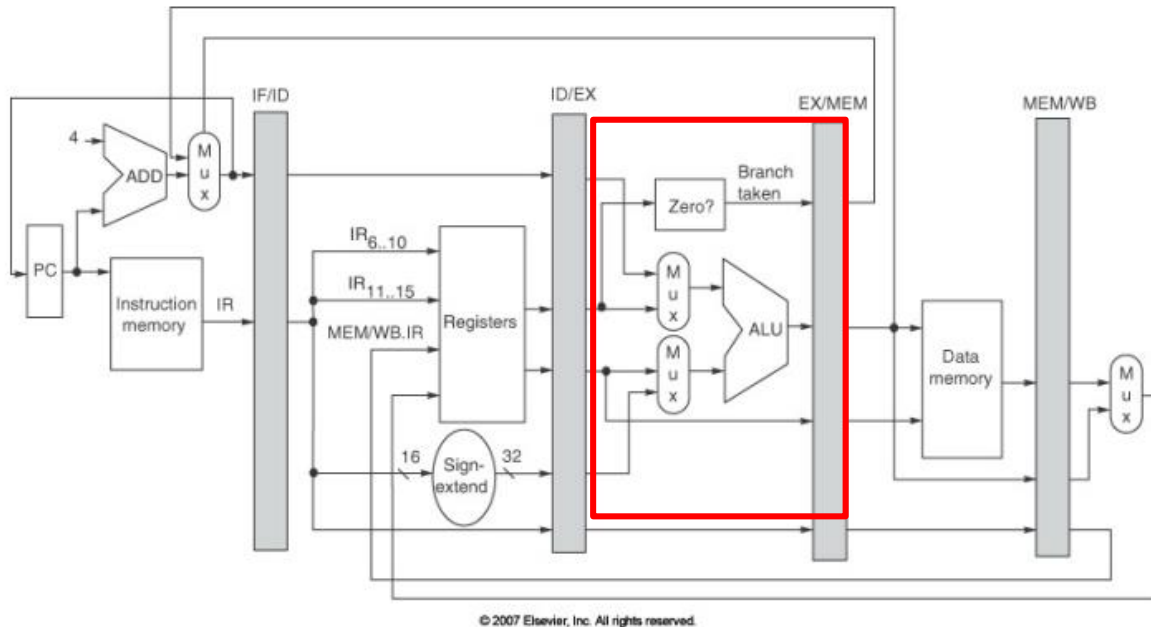
Passed To Next Stage
A <- A func. B
cond = 0;

Execute Address Calculation (EX):

- Perform an operation (for an ALU) or an address calculation (if it's a load/store or a Branch).
- If an ALU, actually do the operation.
- If an address calculation, figure out how to obtain the address

Classic RISC 5-stage pipeline

© 2007 Elsevier, Inc. All rights reserved.
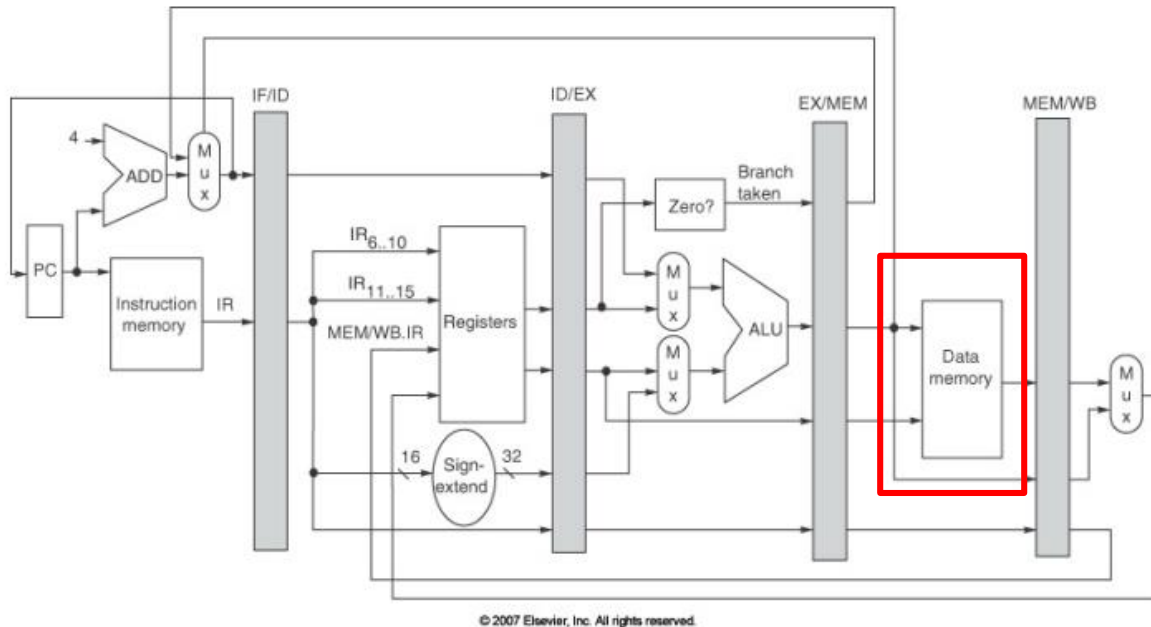
Passed To Next Stage
A = Mem[prev. B]
or
Mem[prev. B] = A

MEMORY ACCESS (MEM):
- If this is an ALU, do nothing.
- If a load or store, then access memory.

Classic RISC 5-stage pipeline
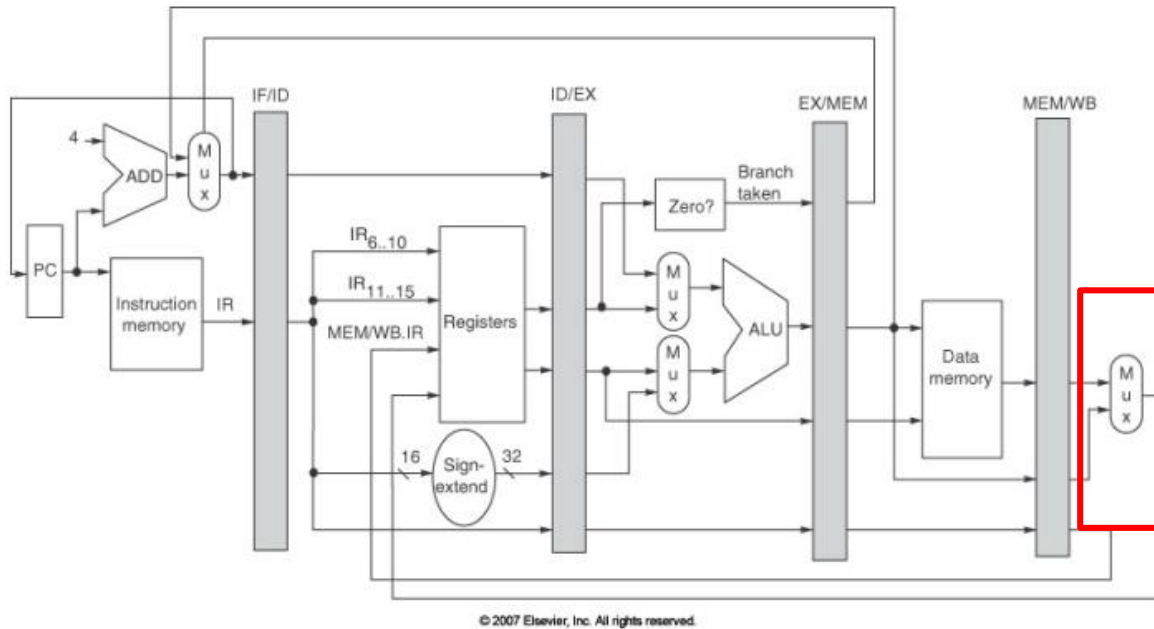
Passed To Next Stage
Regs <- A, B;

© 2007 Elsevier, Inc. All rights reserved.

WRITE BACK (WB):

- Update the registers (GPR) from either the ALU or from the data loaded.

Power breakdown
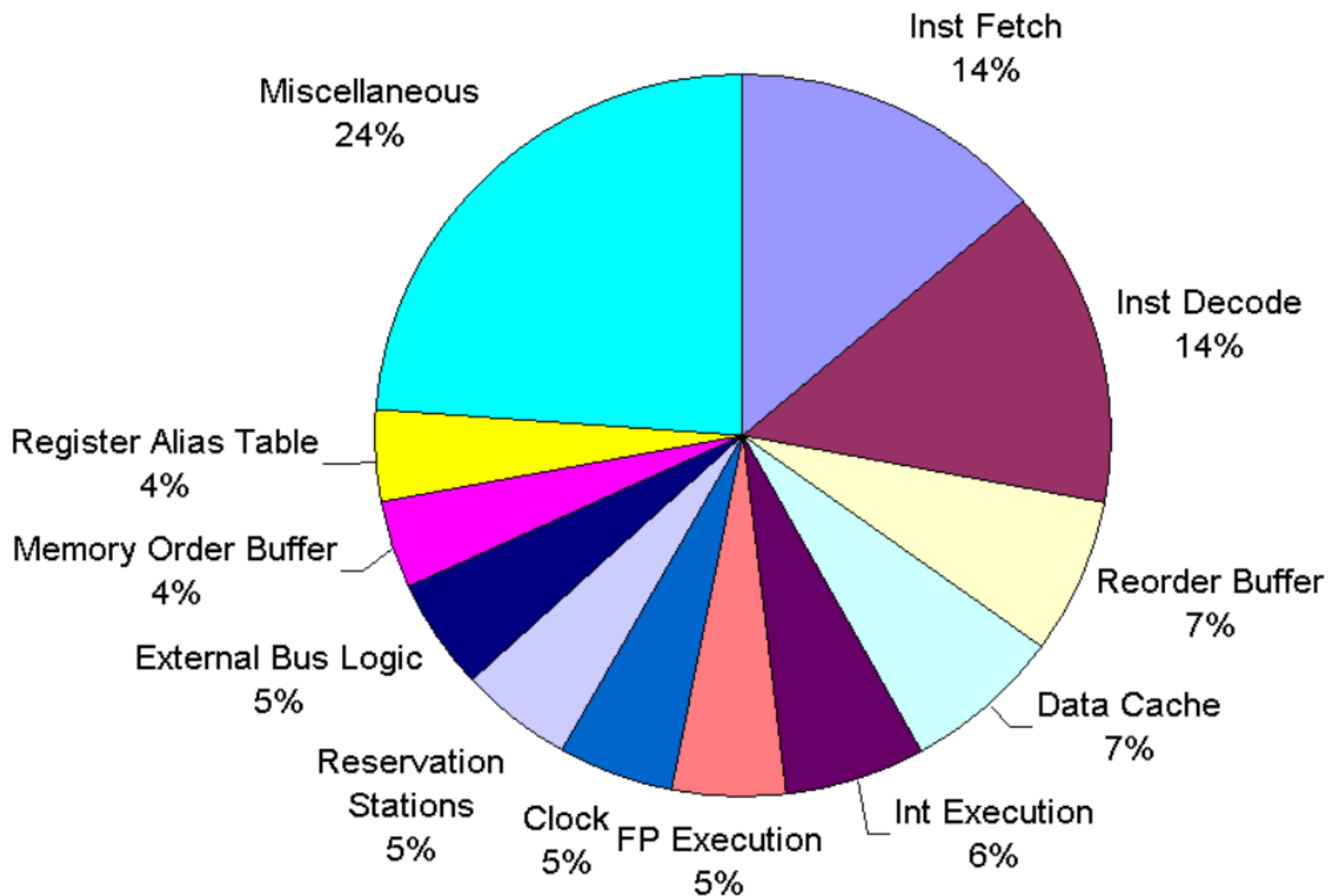
Speed up

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

Speed up

Non-Pipelined
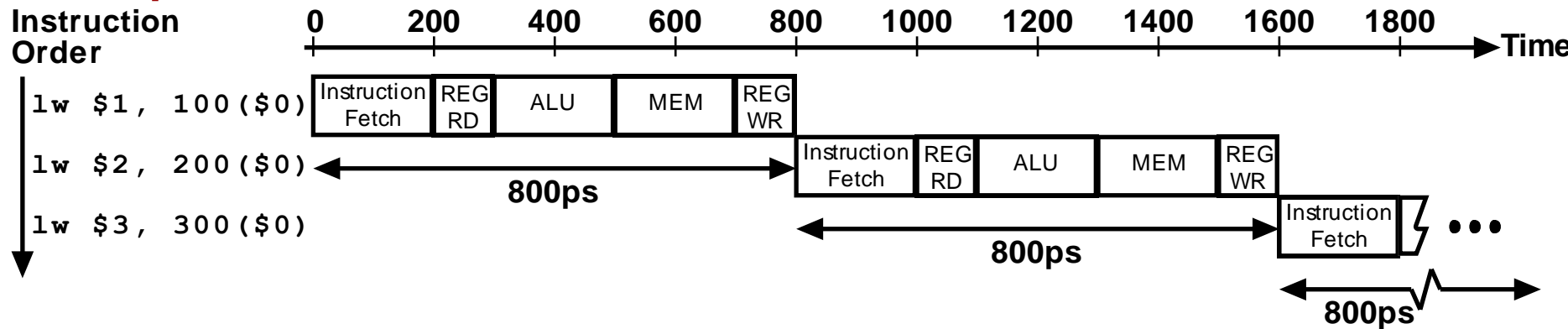
Pipelined
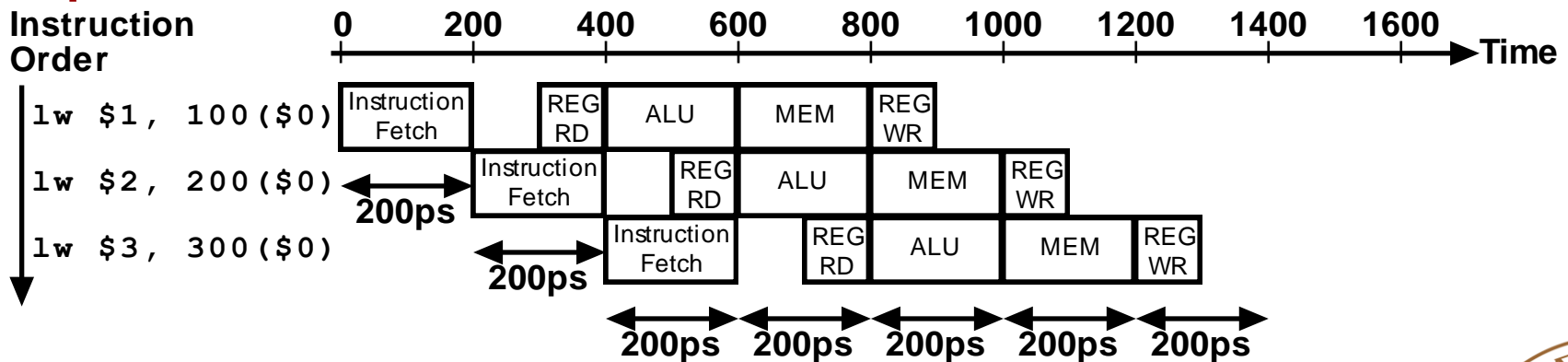
Speed up (idea case)

Consider an un-pipelined processor. Assume:

- It has a 1 ns clock cycle and it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations
- Relative frequencies of these operations are 40%, 20%, and 40%

Pipelined:

- Due to clock skew and setup, pipelining adds 0.2ns of overhead to the clock

The speedup in the instruction execution rate we can gain:

Average instruction execution time
    = 1 ns * ((40% + 20%)*4 + 40%*5)
    = 4.4ns

Speedup from pipeline
    = Average instruction time unpiplined/Average instruction time pipelined
    = 4.4ns/1.2ns = 3.7

Outline

☐ Reiteration

☐ Pipelining

☐ **Harzards**

- Structural hazards
- Data hazards
- Control hazards

☐ Implementation issues

☐ Multi-cycle operations

☐ Summary

Fundamental limitations

☐ **Hazards can prevent next instruction from executing during its designated clock cycle:**

- Structural hazards: Simultaneous use of a HW resource
- Data hazards: Data dependencies between instructions
- Control hazards: Change in program flow

A way of solving hazards is to serialize the execution by inserting "bubbles" that effectively stall the pipeline (interlock).

Structure hazard

Time (clock cycles)
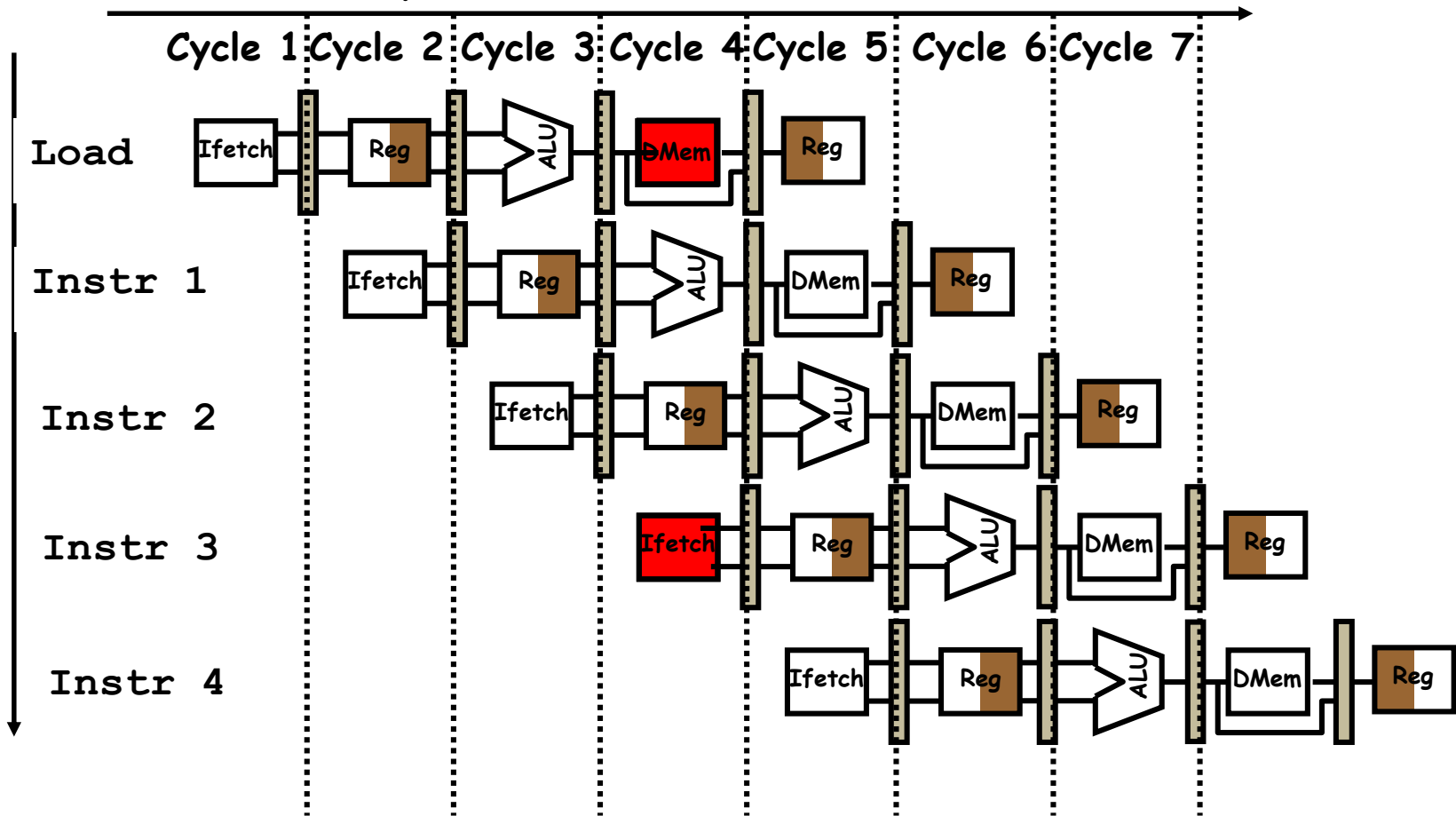
When two or more different instructions want to use same hardware resource in same cycle, e.g., MEM uses the same memory port (only one memory port)

A simple solution

Time (clock cycles)

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7
Load	Ifetch	Reg	ALU	DMem	Reg		
Instr 1		Ifetch	Reg	ALU	DMem	Reg	
Instr 2			Ifetch	Reg	ALU	DMem	Reg
Stall				Bubble	Bubble	Bubble	Bubble	Bubble
Instr 3					Ifetch	Reg	ALU	DMem	Reg
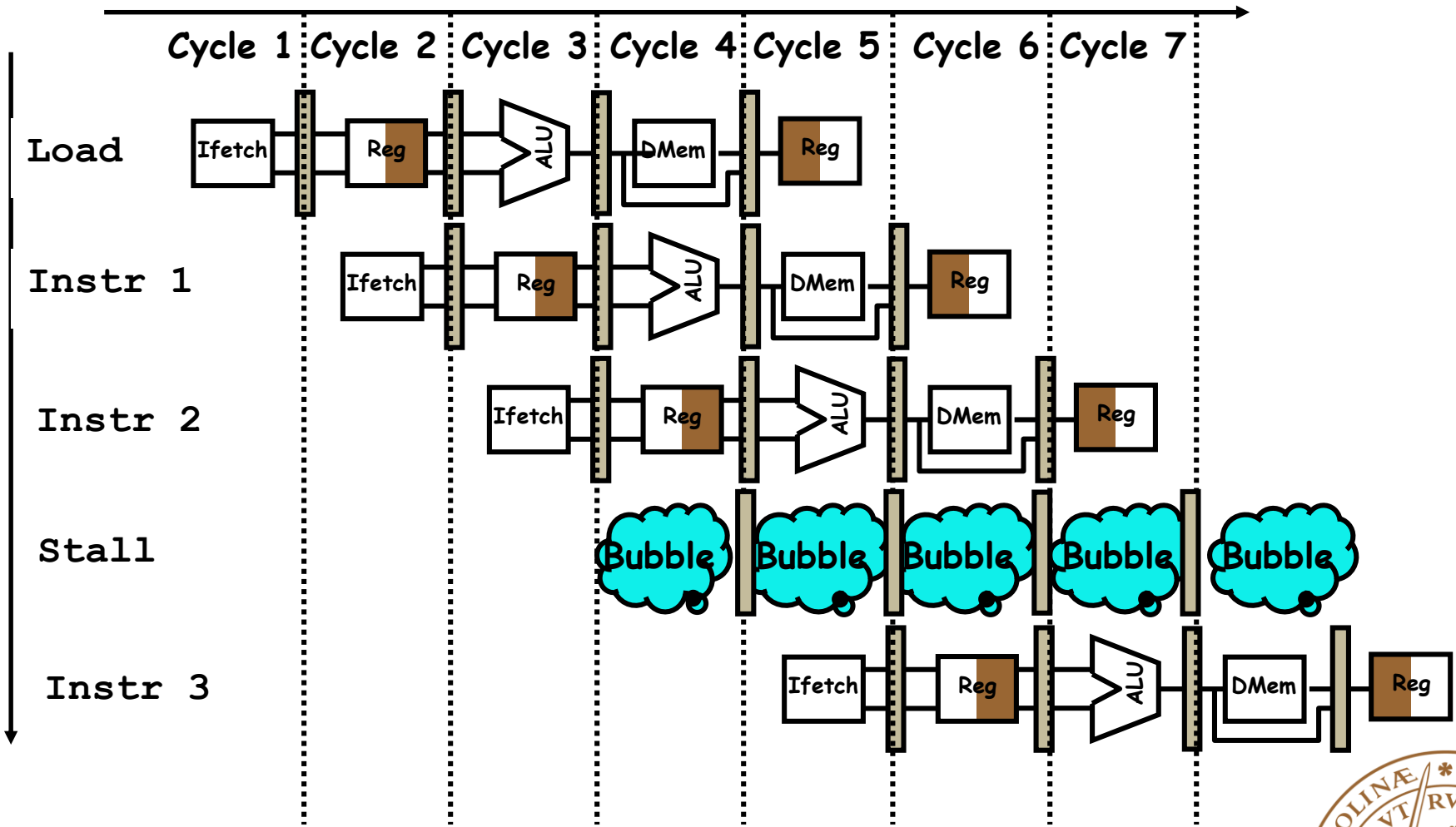
Other solutions

□ **Stall**

- low cost, simple
- Increases CPI
- use for rare case since stalling has performance effect

□ **Pipeline hardware resource**

- useful for multi-cycle resources
- good performance
- sometimes complex

□ **Replicate resource**

- good performance
- increases cost (+ maybe interconnect delay)
- useful for cheap resources

Dual/single port memory (65nm)

size	Single-port	Dual-port
64*16	$12um^2$/bit	$23um^2$/bit
256*16	$4.6um^2$/bit	$8um^2$/bit
512*16	$4um^2$/bit	$6um^2$/bit

Data hazard

Time (clock cycles)

add **r1**,r2,r3

sub r4,**r1**,r3

and r6,**r1**,r7

or   r8,**r1**,r9

xor r10,**r1**,r11
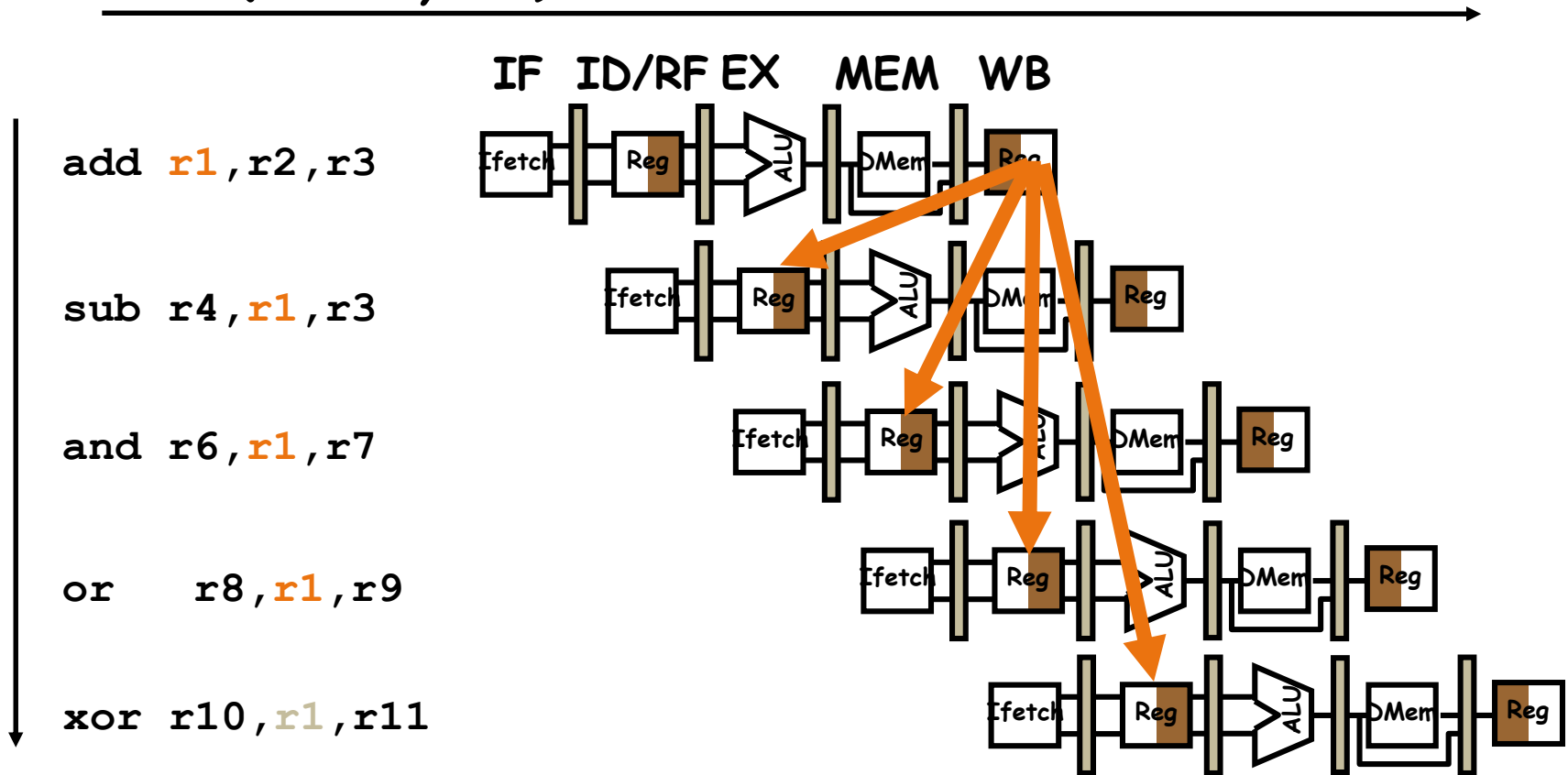
The use of the result of the ADD instruction in the next 3 instructions causes a hazard, since the register is not written until after those instructions read it.

Fundamental types of data hazard

Code sequence:
$$Op_i \quad A$$
$$Op_{i+1} \quad A$$

☐ **RAW (Read-After-Write)**
- Instruction $i + 1$ reads A and i modifies A
- Instruction $i+1$ reads old A!

☐ **WAR (Write-After-Read)**
- Instruction $i + 1$ modifies A and instruction i reads new A

☐ **WAW (Write-After-Write)**
- Instructions i and $i + 1$ both modifies A
- The value in A is the one written by instruction i

☐ **(RAR?)**

Strategies for data hazard

☐ **Interlock**

- Wait for hazard to clear by holding dependent instruction in issue stage

☐ **Forwarding**

- Resolve hazard earlier by bypassing value as soon as available

☐ **Speculate**

- Guess on value, correct if wrong

Interlock and forwarding

```
add x1, x3, x5
sub x2, x1, x4
```


add x1, x3, x5

bubble

bubble

bubble

sub x2, x1, x4

Instruction interlocked in decode stage

add x1, x3, x5

sub x2, x1, x4

Bypass around ALU with no bubbles
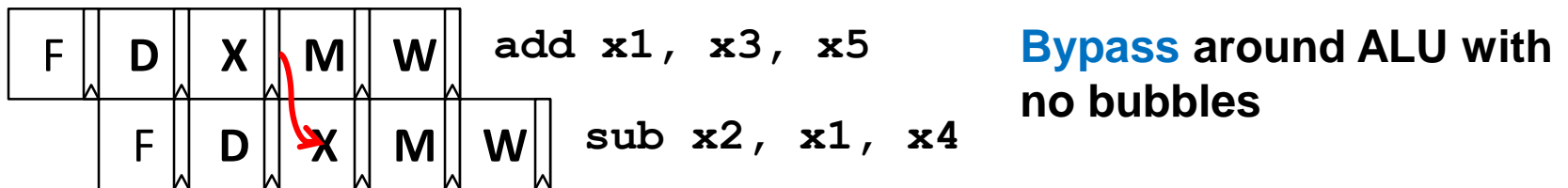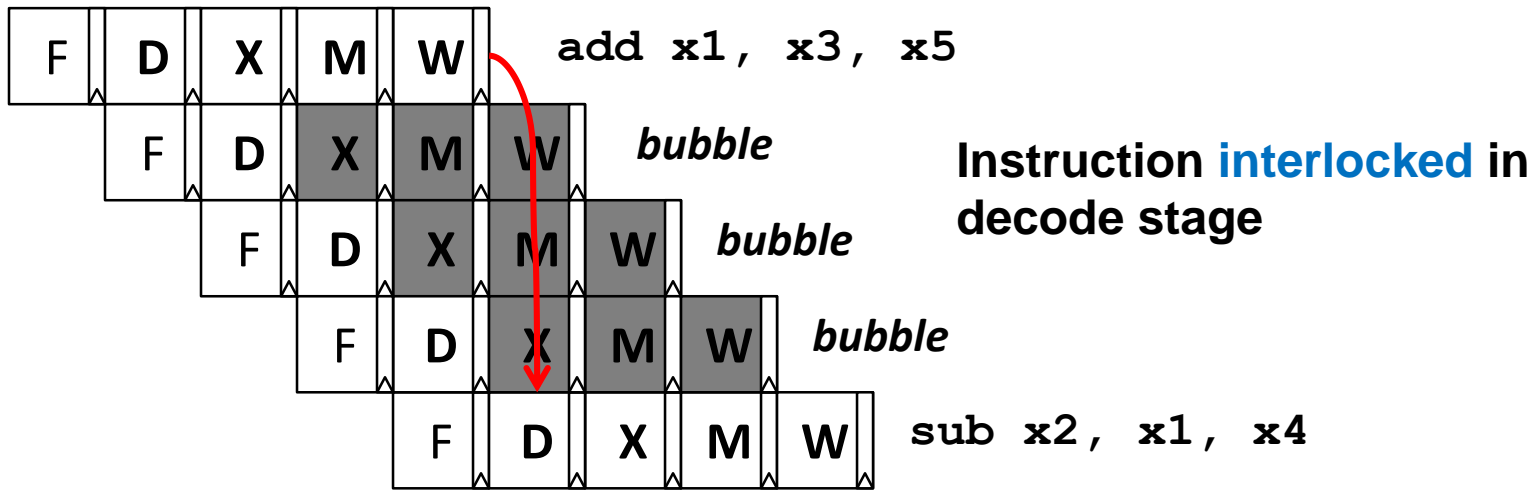
Hardware support of forwarding
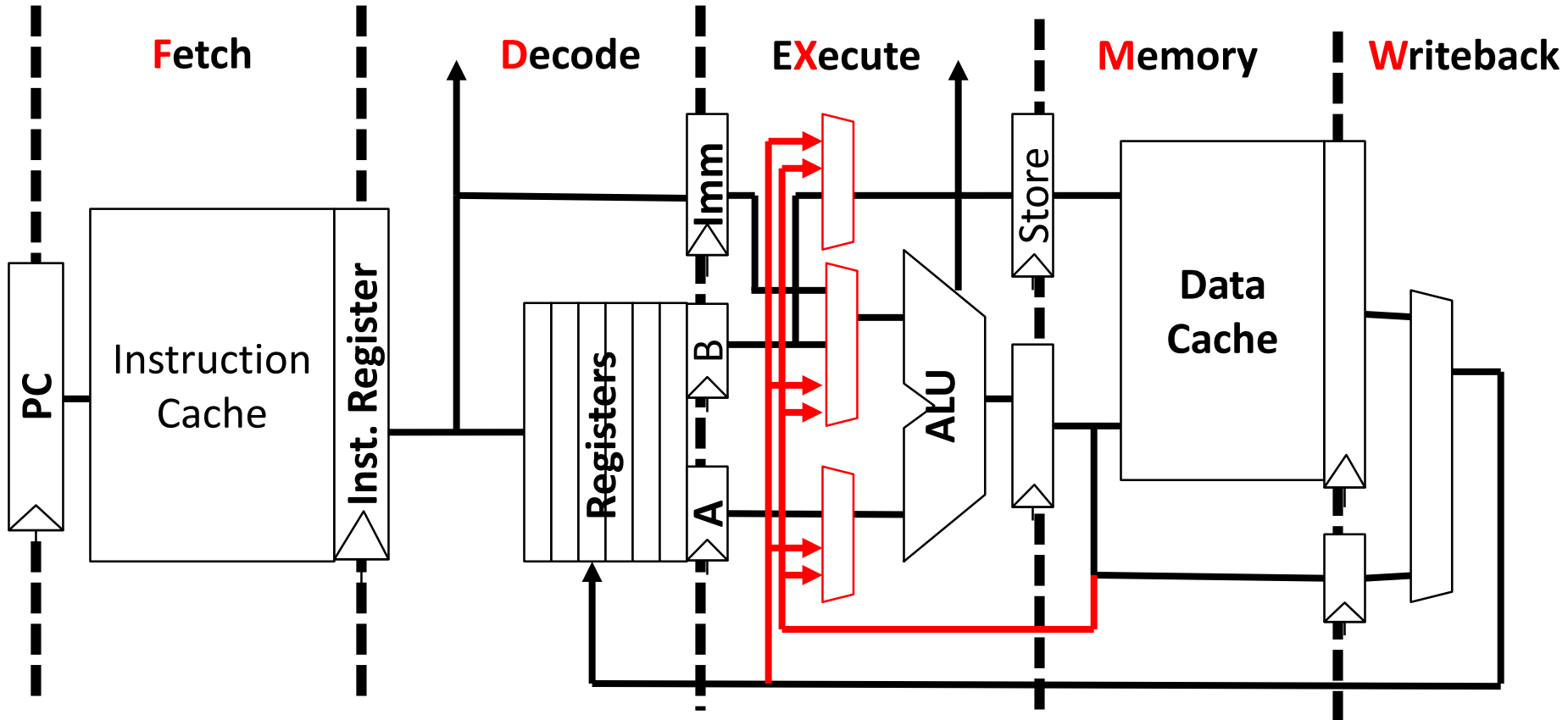
Data hazard with forwarding
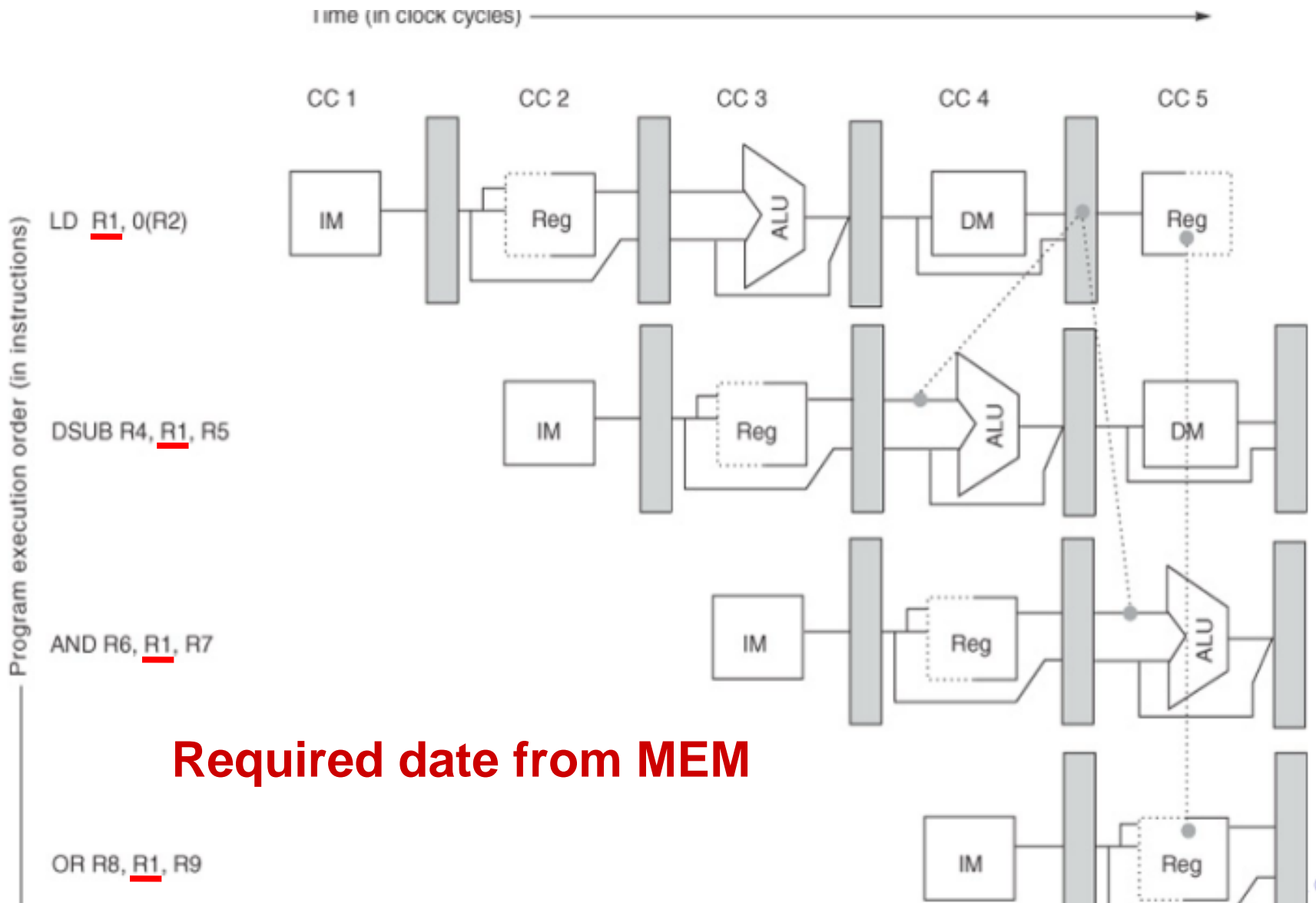
Required date from MEM
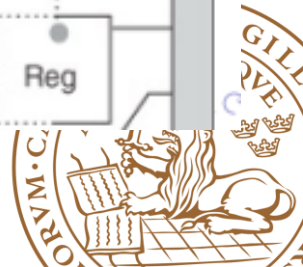
Data hazard with forwarding

Software scheduling of load

Try producing fast code for
a = b + c;
d = e - f;
assuming a, b, c, d, and f are
in memory

```
LD        R1, B
LD        R2, C
DADD      R3,R1,R2
SD        R3, A
LD        R5, F
LD        R4, E
DSUB      R6,R4,R5
SD        R6, D
```
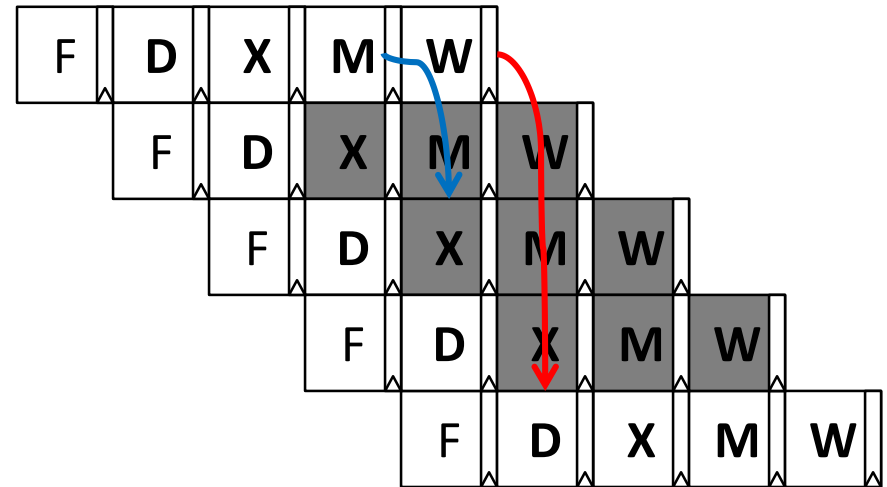

How many stalls?
How many stalls with hardware forwarding?

Software scheduling of load

Try producing fast code for
a = b + c;
d = e - f;
assuming a, b, c, d, e, and f are in memory.

			Code re-order	
LD	R1, B		LD	R1, B
LD	R2, C		LD	R2, C
DADD	R3,R1,R2		LD	R4, E
SD	R3, A		LD	R5, F
LD	R5, F		DADD	R3,R1,R2
LD	R4, E		DSUB	R6,R4,R5
DSUB	R6,R4,R5		SD	R3, A
SD	R6, D		SD	R6, D

Architecture dependent optimization

Scheduling vs un-scheduling
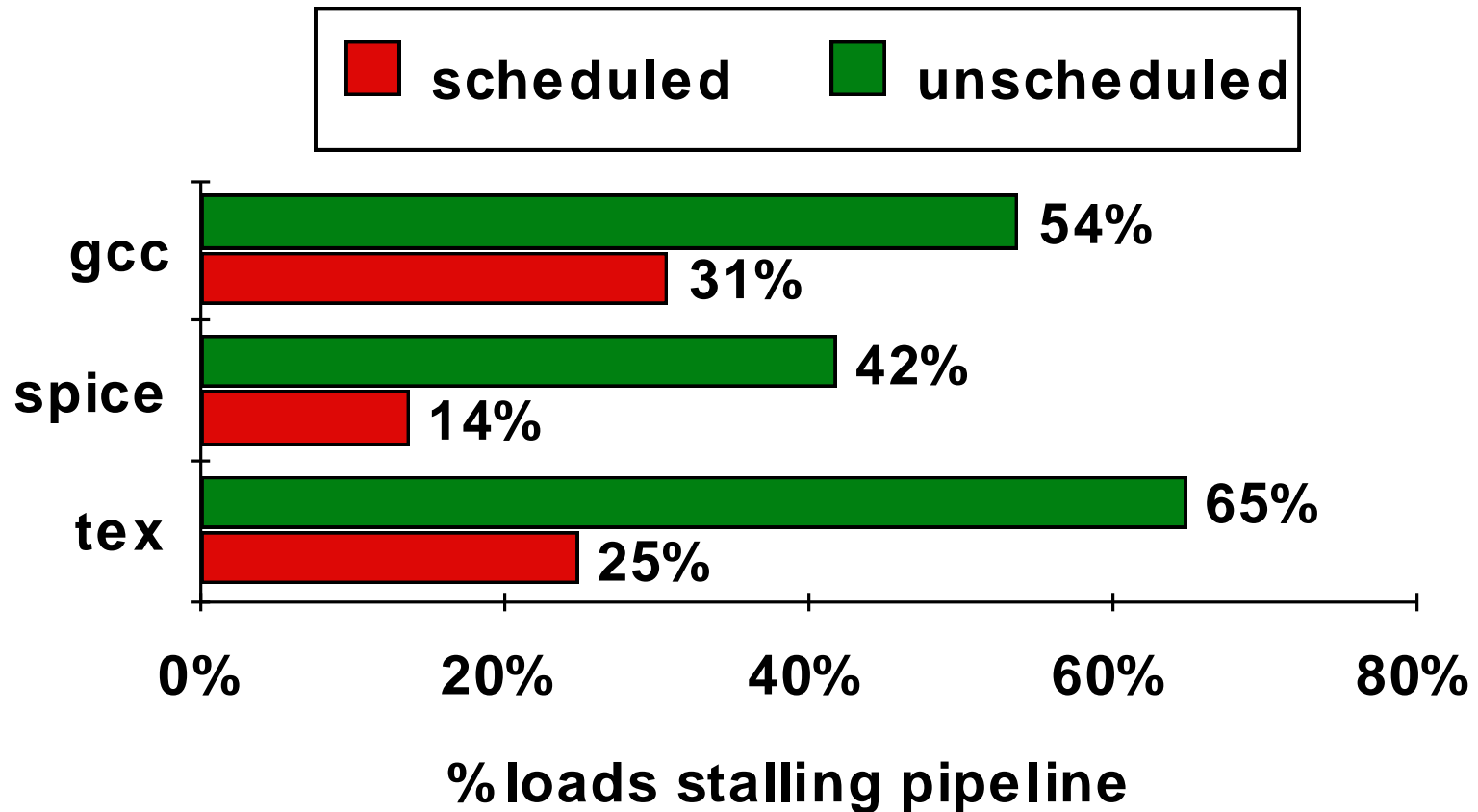
Control hazard

☐ **Control hazard**

- Need to find the destination of a branch, and can't fetch any new instructions until we know that destination

Branch instruction	IF	ID	EX	MEM	WB						
Branch successor		IF	*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	

☐ **Assume: branches are not resolved until the MEM stage**

☐ **Three wasted clock cycles:**

- two stalls
- one extra instruction fetch (IF)

☐ **If branch is not taken, the extra IF not needed**

Hardware support to reduce control hazard

- ☐ **Calculate target address and test condition in ID**
- ☐ **1 clock cycle branch penalty instead of 3!**
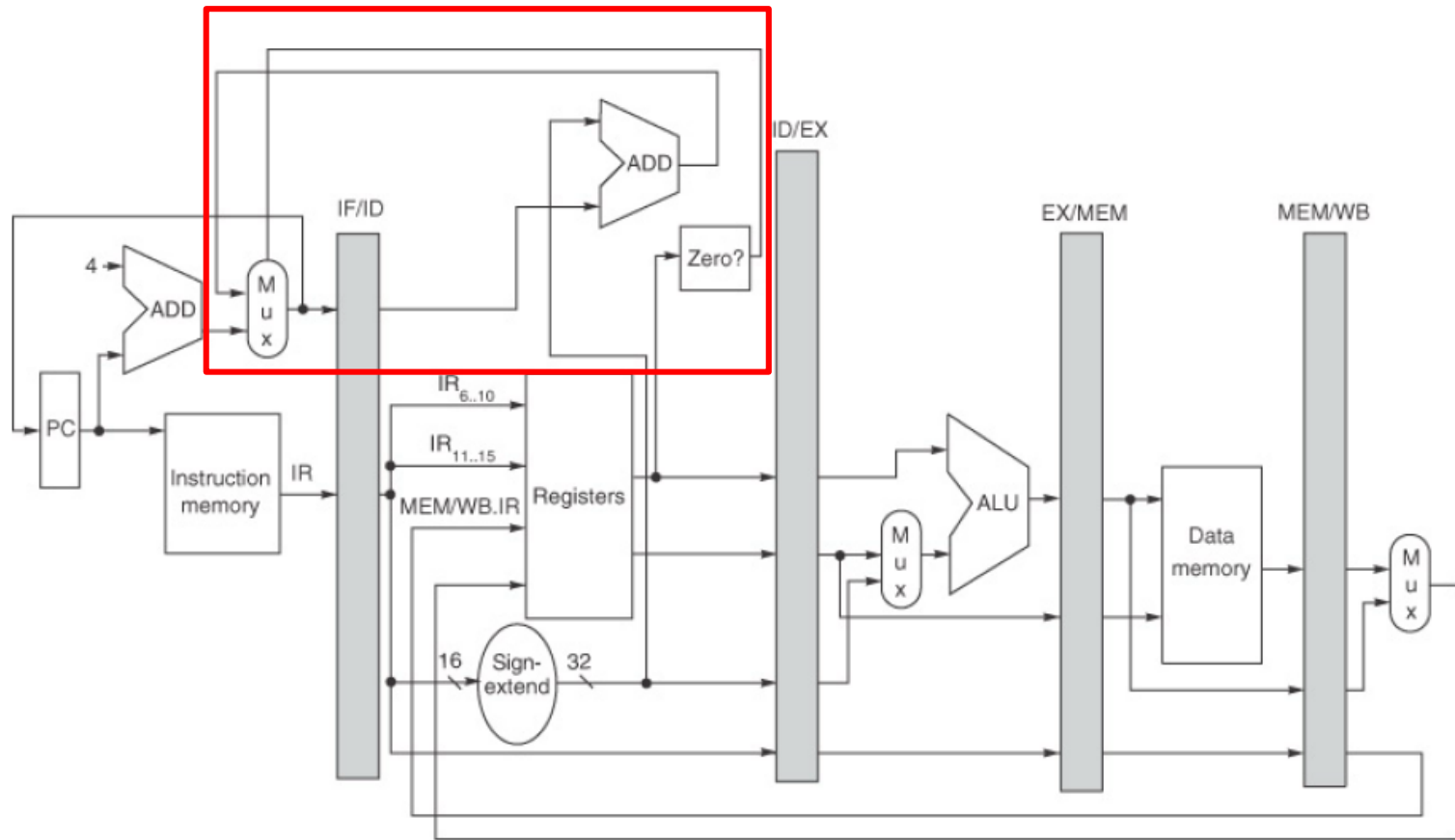
Four control hazard alternatives

☐ **Stall until branch condition and target is known**

☐ **Predict branch not taken**

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if the branch is actually taken
- **Works well if state is updated late in the pipeline (as in MIPS)**
- 33 % MIPS conditional branches not taken on average

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 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

one cycle penalty if taken

Four control hazard alternatives

□ **Predict Branch taken (benefit in our case?)**

- 67 % MIPS conditional branches taken on average
- MIPS calculates target address in ID stage! Still one cycle penalty

□ **Delayed branch**

- Define branch to take place after a following instruction
- Slot delay allows proper decision and branch target address calculation in a 5 stage pipeline such as MIPS

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 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

Compiler support for delay branch

- ☐ **Scheduling "from before" is always safe (if independent)**
- ☐ **Scheduling "from target" or "fall through" is not always safe**

(a) From before

DADD R1, R2, R3

if R2 = 0 then ──

    Delay slot

becomes

if R2 = 0 then ──

    DADD R1, R2, R3

(b) From target

DSUB R4, R5, R6 ◄──

DADD R1, R2, R3

if R1 = 0 then ──

    Delay slot

becomes

DSUB R4, R5, R6

DADD R1, R2, R3

if R1 = 0 then ──

    DSUB R4, R5, R6

(c) From fall-through

DADD R1, R2, R3

if R1 = 0 then ──

    Delay slot

OR R7, R8, R9

DSUB R4, R5, R6 ◄──

becomes

DADD R1, R2, R3

if R1 = 0 then ──

    OR R7, R8, R9

DSUB R4, R5, R6 ◄──
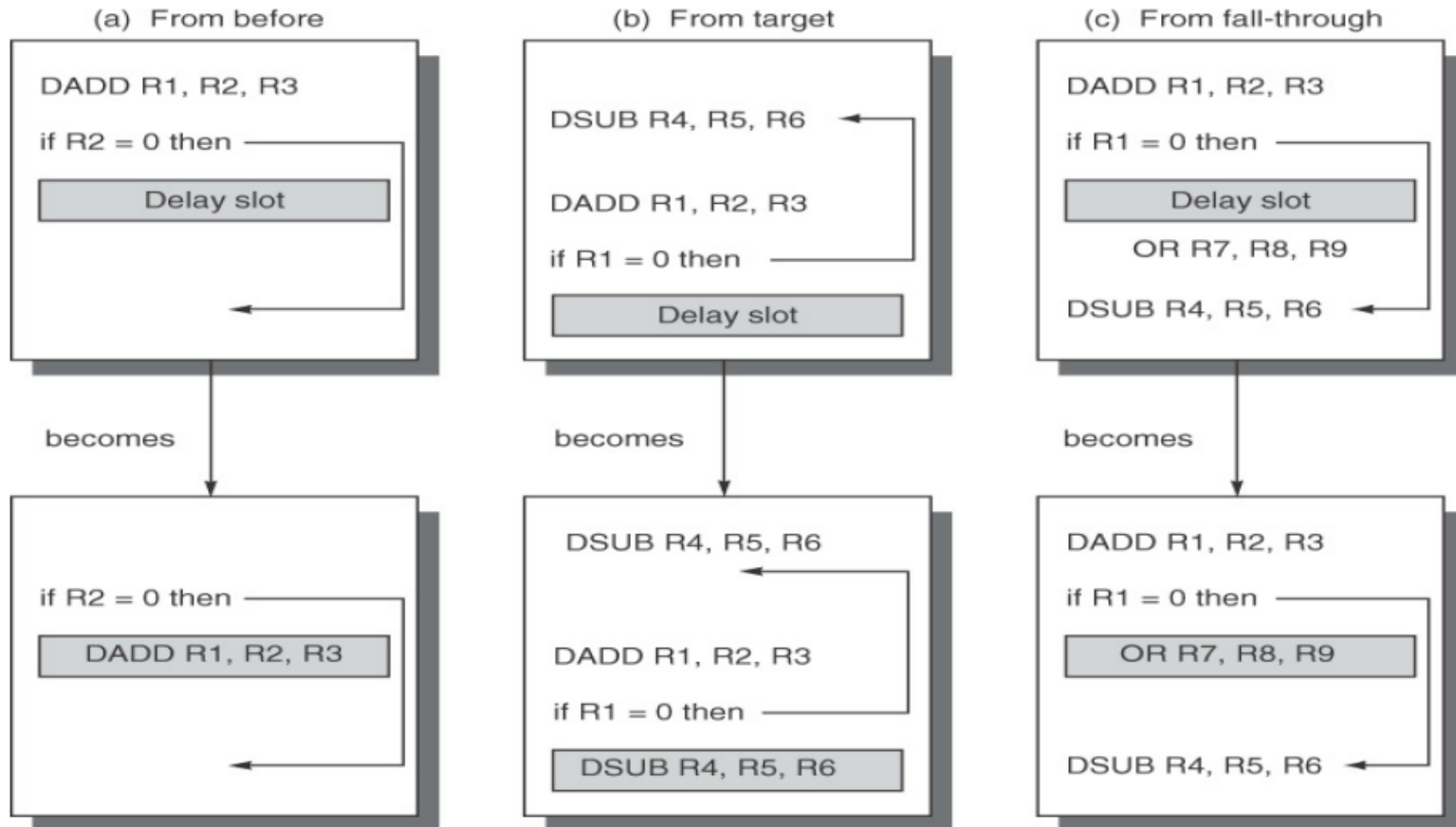
Pipeline speed up

$$CPI_{pipelined} = IdealCPI + \text{\# stall-cycles/instruction}$$

$$Speedup = \frac{AverageInstructionTime_{unpipelined}}{AverageInstructionTime_{pipelined}}$$

$$Speedup = \frac{CPI_{unpipelined}}{\text{Ideal CPI} + \text{\# stall-cycles/instr}} * \frac{TC_{unpipelined}}{TC_{pipelined}}$$

$$\approx \frac{\#stages}{1 + \text{\# stall-cycles/instruction}}$$

Outline

What's hard to implement
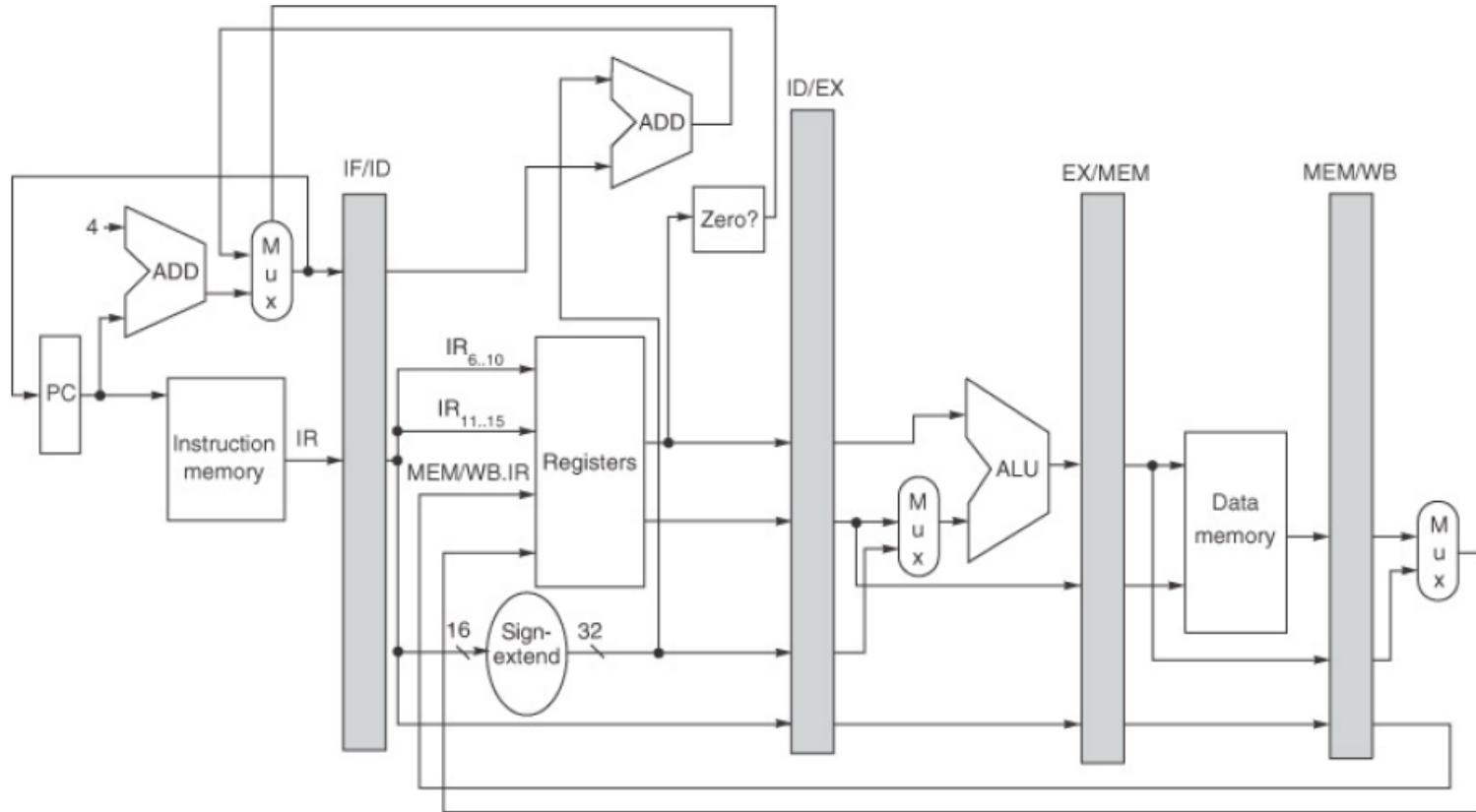
☐ **Exceptions (fault, interrupt)**

Exceptions
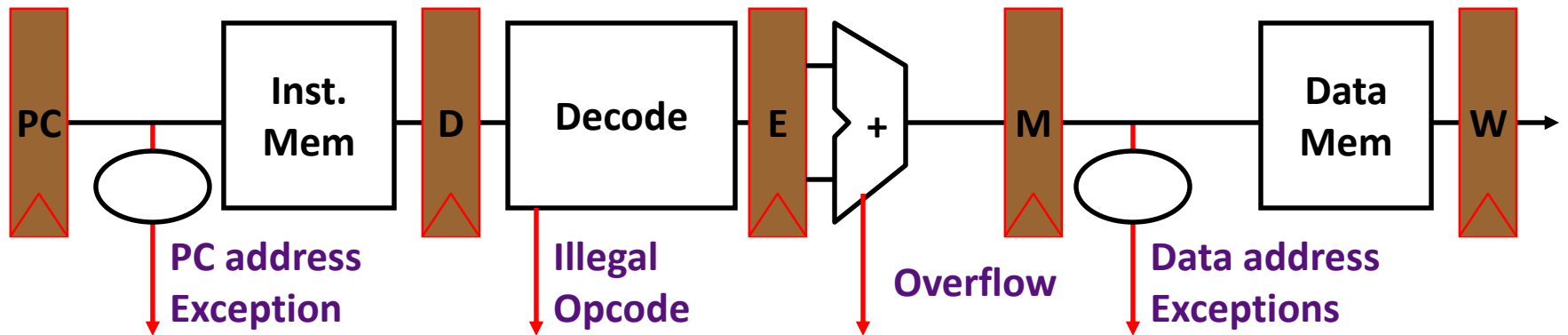
☐ When an interrupt occurs:

- How to stop the pipeline?
- How to restart the pipeline?
- Who caused the interrupt?

☐ A pipeline implements precise exceptions if:

- All instructions before the faulting instruction can complete
- All instructions after (and including) the faulting instruction can safely be restarted

Exceptions are difficult in pipeline

❑ **We need to be able to restart an instruction that causes an exception:**

- Force a trap instruction (e.g., some special routine call) into the pipeline
- Turn off all writes for the faulting instruction
- Save the PC for the faulting instruction to be used in return from exception handling
- If delayed branch is used we may need to save several PC's

Solution for simple MIPS

☐ **Need to add control and datapaths to support exceptions and interrupts.**

☐ **When an exception or interrupt occurs, the following must be done:**

- EPC <= PC
- Cause <= (cause code for event)
- Status <= (fault)
- PC <= (handler address)

☐ **To return from an exception or datapath, the following must be done:**

- PC <= EPC
- Status <= (fault clear)

Exceptions are difficult in pipeline

☐ **Exceptions may be generated out-of-(program) order**

IF	ID	EX	MEM	WB
page fault on instruction fetch, misaligned memory access, protection violation	undefined or illegal opcode	arithmetic exception	page fault on data fetch, misaligned memory access, memory protection	none

LD (faults in MEM)
DADD (faults in IF)
⟹ The DADD faults before the LD

Solution for simple MIPS

☐ **Add a hardware status vector containing exceptions**

☐ **Pass along with instruction in the pipeline**

- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions *for a given instruction*

☐ **Turn of writes when an exception entered in the status vector**

☐ **Handle exceptions from status vector in WB (in program order)**

- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage
- Inject external interrupts at commit point (override others)

Handle at commit point not when exception happens

Solution for simple MIPS
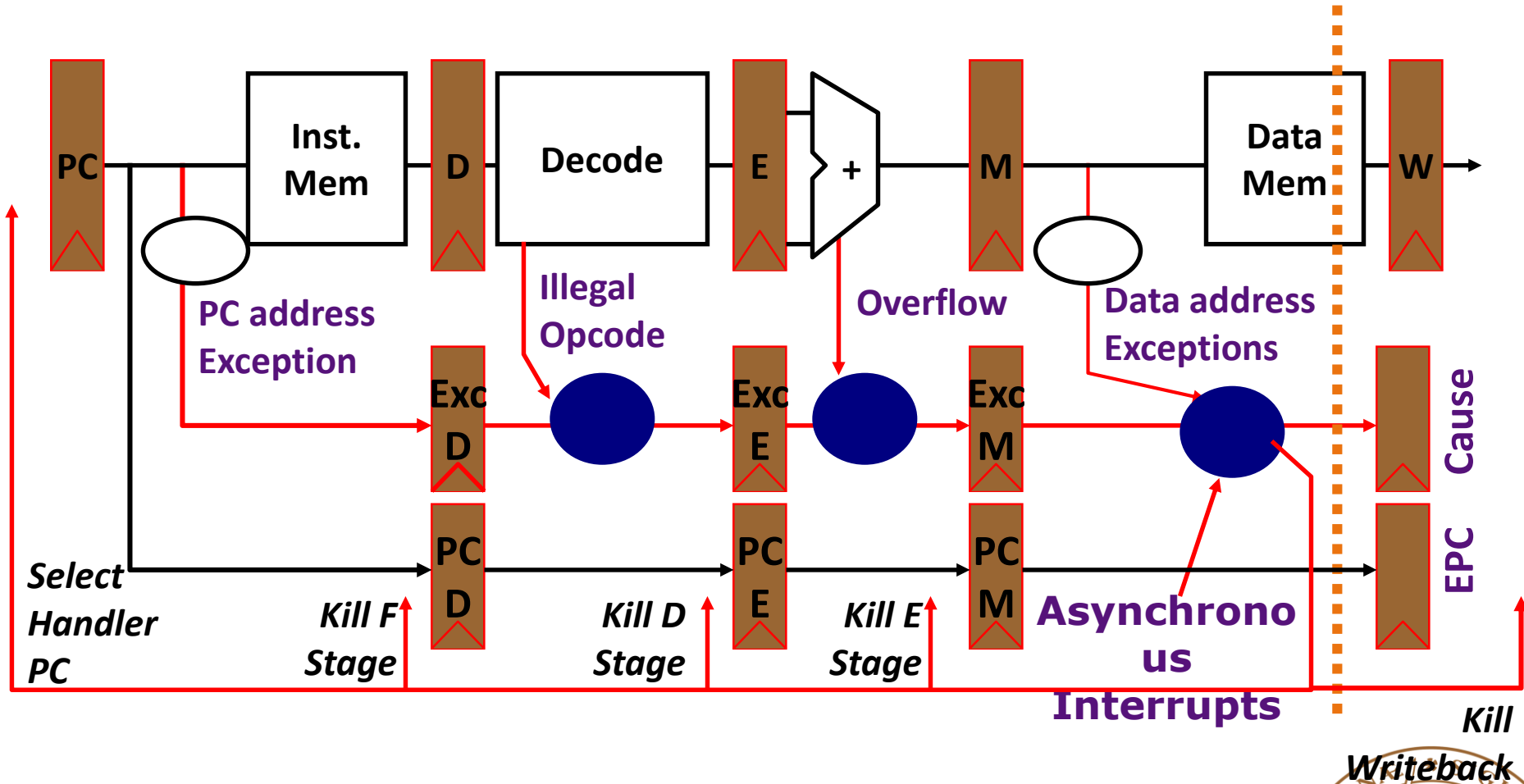
Outline

Multi-cycle instruction in pipeline (FP)
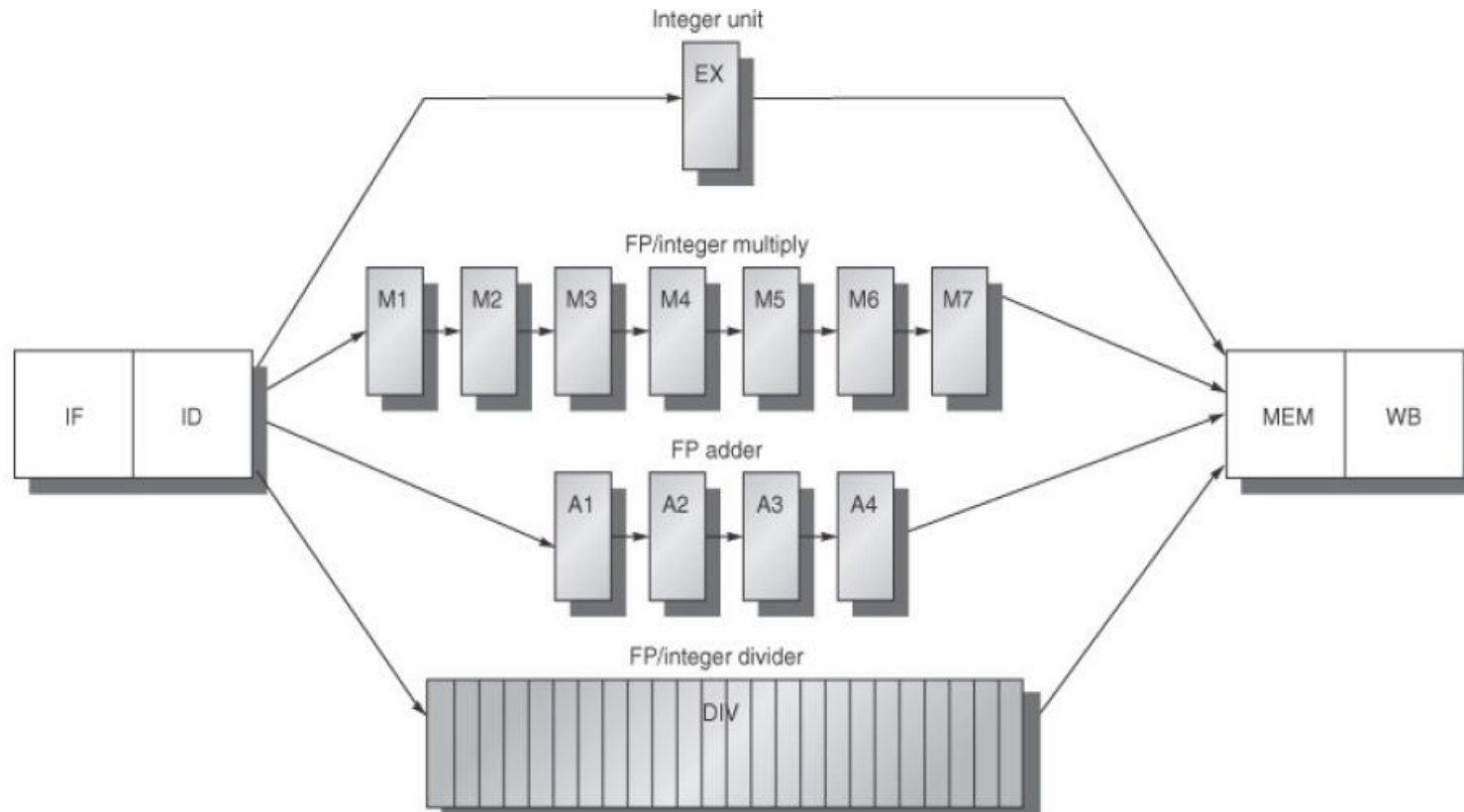
FP Instruction	Latency	Initiation Rate
Add, Subtract	3	1
Multiply	6	1
Divide	24	25

Parallelism between integer and FP

MUL.D	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM
ADD.D		IF	ID	A1	A2	A3	A4	MEM	WB
DSUB			IF	ID	EX	MEM	WB
LD				IF	ID	EX	MEM	WB

☐ **Instructions are issued in order**

☐ **Instructions may be completed out of order**

Pipeline hazard

☐ **Structural hazards**

☐ **RAW hazards**

☐ **WAW harzards**

☐ **WAR hazards**

Pipeline hazard

☐ **Structural hazards. Stall in ID stage if:**

- The functional unit is occupied (applicable to DIV only)
- Any instruction already executing will reach the MEM/WB stage at the same time as this one

☐ **RAW hazards:**

- Normal bypassing from MEM and WB stages
- Stall in ID stage if any of the source operands is destination operand in any of the FP functional units

					Clock cycle number						
Instruction	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
L.D F2,0(R2)							IF	ID	EX	MEM	WB

Pipeline hazard

☐ **WAR hazards?**

- There are no WAR-hazards since the operands are read (in ID) before the EX-stages in the pipeline

☐ **WAW hazard**

```
DIV.D     F0,F2,F3      FP divide 24 cycles
...
SUB.D     F0,F8,F10     FP subtract 3 cycles
```

- SUB finishes before DIV which will overwrite the result from SUB!
- are eliminated by stalling SUB until DIV reaches MEM stage
- When WAW hazard is a problem?

Summary

☐ **Pipelining (ILP):**
  - Speeds up throughput, not latency
  - Speedup ≤ #stages

☐ **Hazards limit performance, generate stalls:**
  - Structural: need more HW
  - Data (RAW,WAR,WAW): need forwarding and compiler scheduling
  - Control: delayed branch, branch prediction

$$Speedup = \frac{CPI_{unpipelined}}{Ideal\ CPI + \#\ stall\text{-}cycles/instr} * \frac{TC_{unpipelined}}{TC_{pipelined}}$$

$$\approx \frac{\#stages}{1 + \#\ stall\text{-}cycles/instruction}$$

☐ **Complications:**
  - Precise exceptions may be difficult to implement
  - The instruction set can be more or less suited for pipelining