



LUND
UNIVERSITY

EITF20: Computer Architecture

Part2.2.1: Pipeline-1

Liang Liu
liang.liu@eit.lth.se



Previous lecture

□ Instruction set architecture

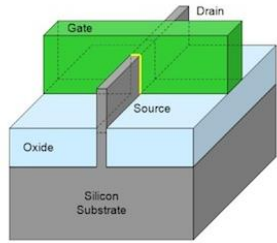


ISA Classification

- ❑ **Where are operands stored?**
 - registers, memory, stack, accumulator
- ❑ **How many explicit operands are there?**
 - 0, 1, 2, or 3
- ❑ **How is the operand location specified?**
 - register, immediate, indirect, . . .
- ❑ **What type & size of operands are supported?**
 - byte, int, float, double, string, vector. . .
- ❑ **What operations are supported?**
 - add, sub, mul, move, compare . . .
- ❑ **How is the operation flow controlled?**
 - branches, jumps, procedure calls . . .
- ❑ **What is the encoding format**
 - fixed, variable, hybrid...



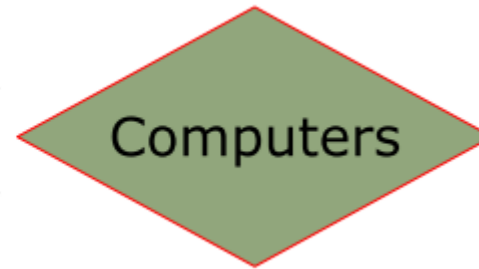
What affects the computer architecture?



Technology

Software

Applications



ISA Classification

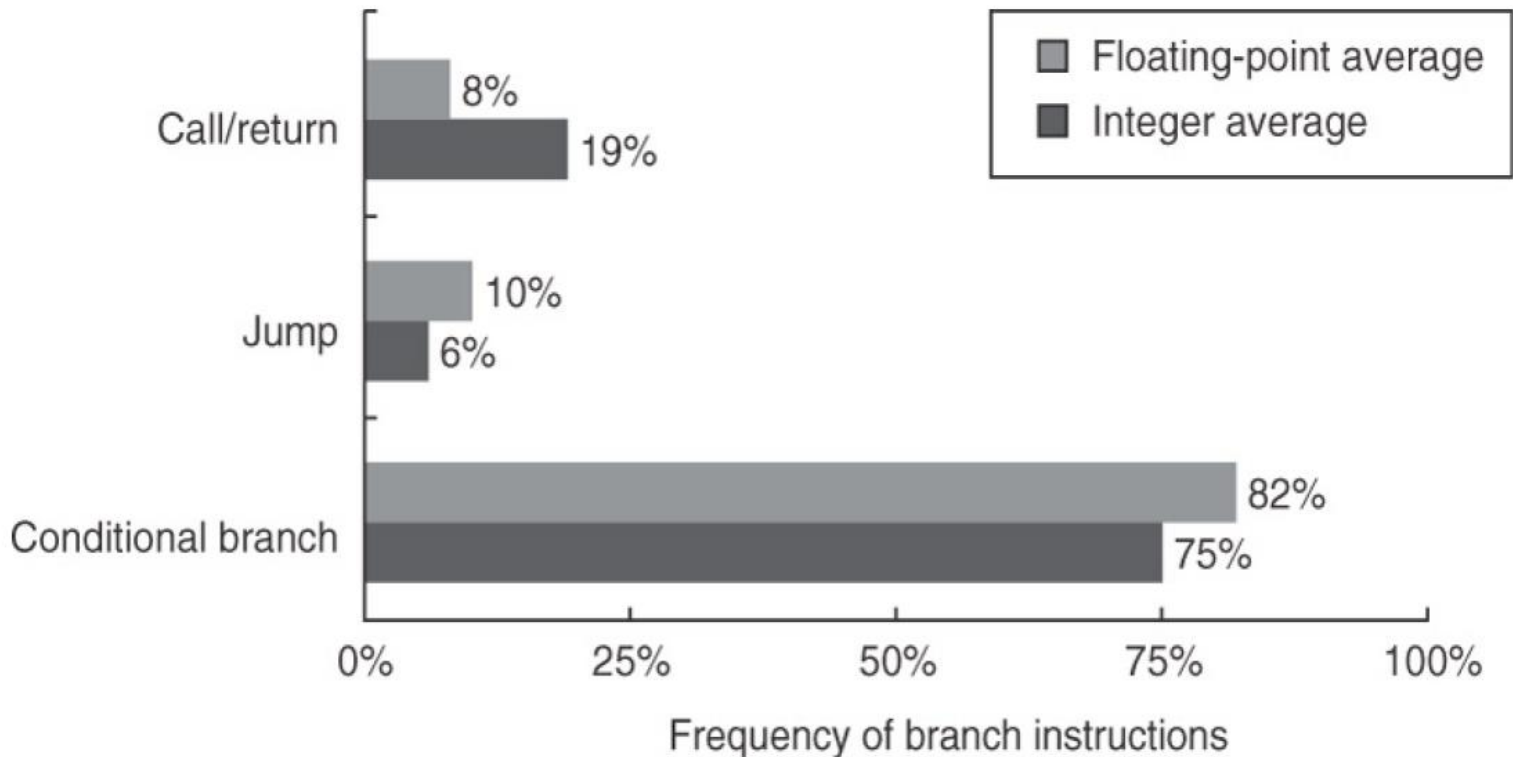
□ What's needed in an instruction set?

- Addressing
- Operands
- Operations
- **Control Flow**



Types of control instructions

- Conditional branches
- Unconditional branches (jumps)
- Procedure call/returns



© 2007 Elsevier Inc. All rights reserved.



Outline

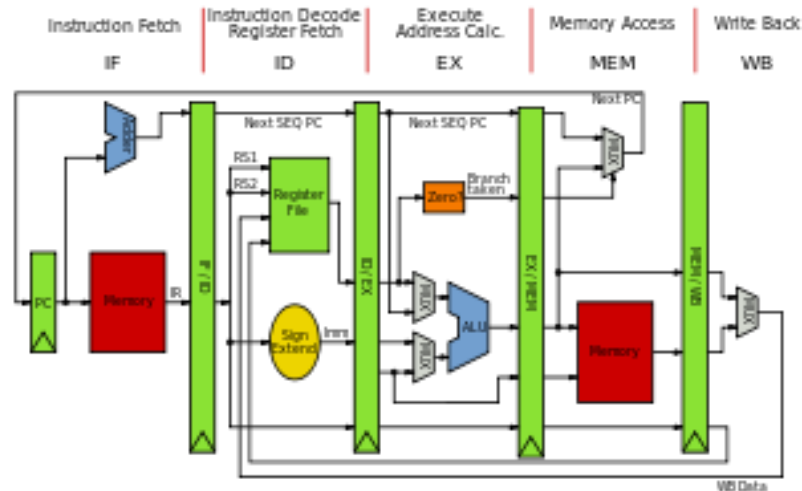
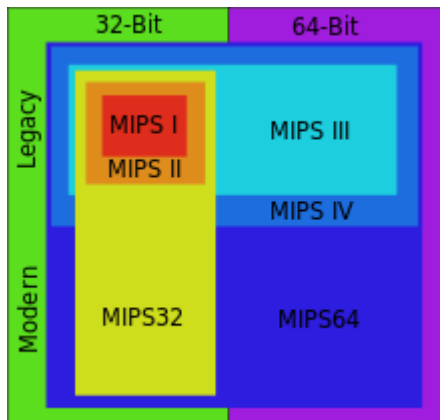
- Reiteration
- Instruction Set Principles
- The Role of Compilers
- **MIPS**



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 and double precision IEEE floating points
- Memory byte addressable with 64-bit addresses
- Addressing modes: immediate and displacement



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



Outline

□ Pipelining

□ Harzards

- Structural hazards
- Data hazards
- Control hazards

□ Implementation issues

□ Multi-cycle operations

□ Summary



Outline

□ Reiteration

□ **Pipelining**

□ Harzards

- Structural hazards
- Data hazards
- Control hazards

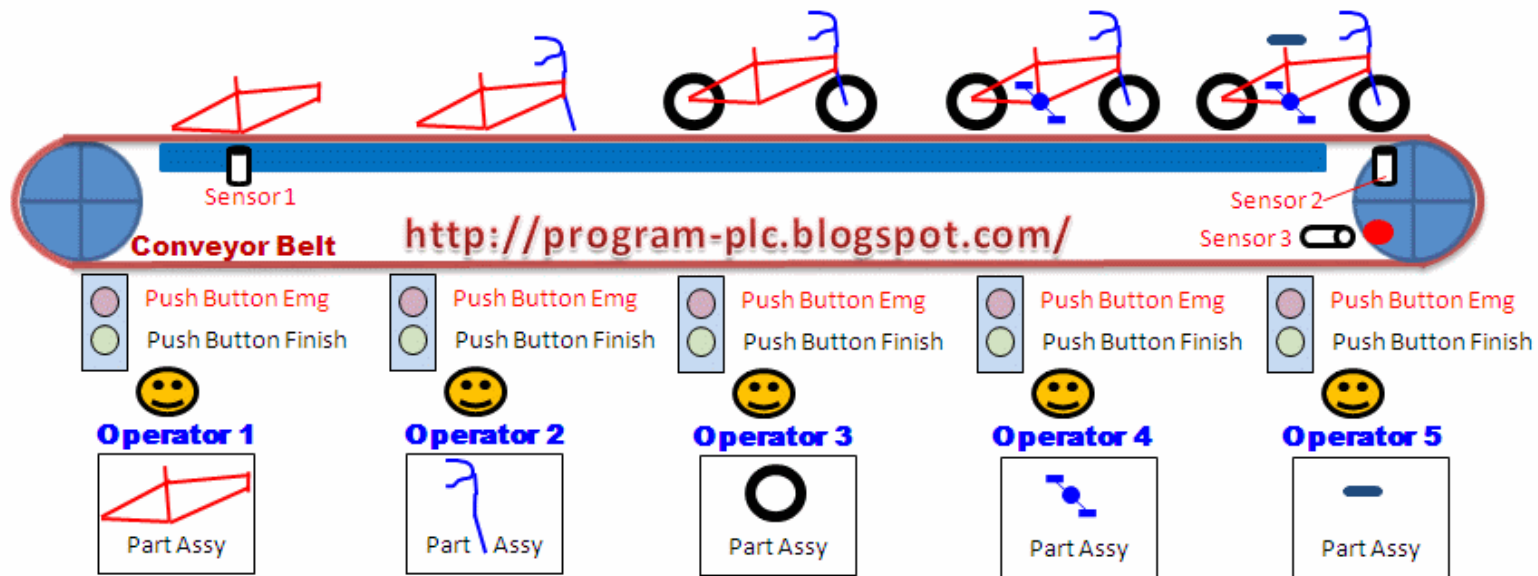
□ Implementation issues

□ Multi-cycle operations

□ Summary

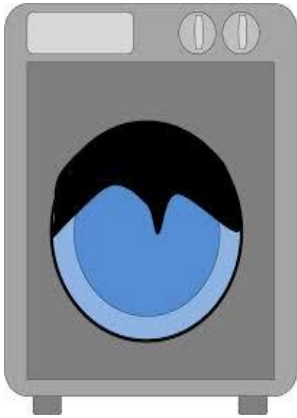


The Assembly Line ...



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:

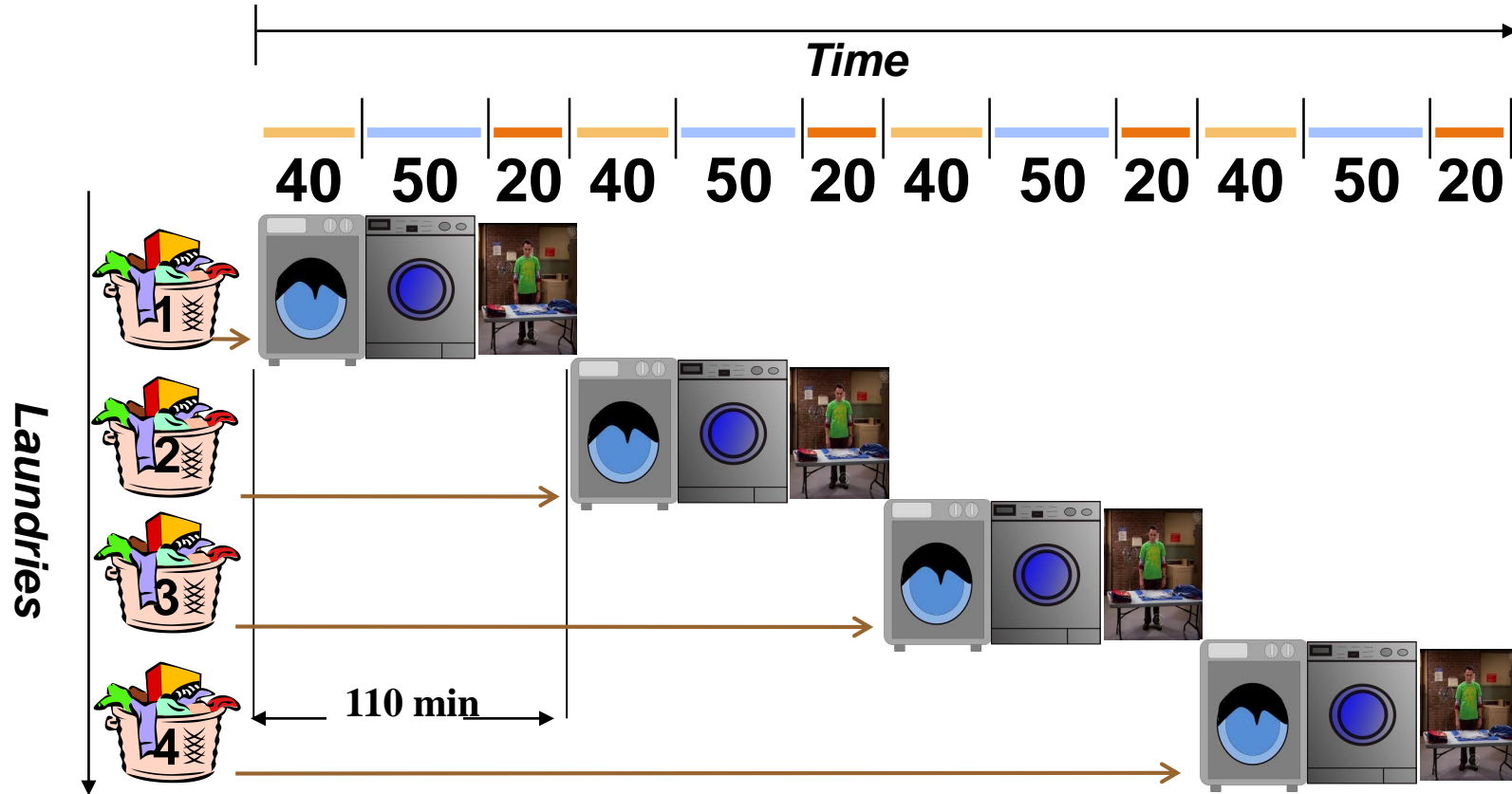
- Washer takes 40 minutes
- Dryer takes 50 minutes
- “Folding” takes 20 minutes



□ Need to do 4 laundries



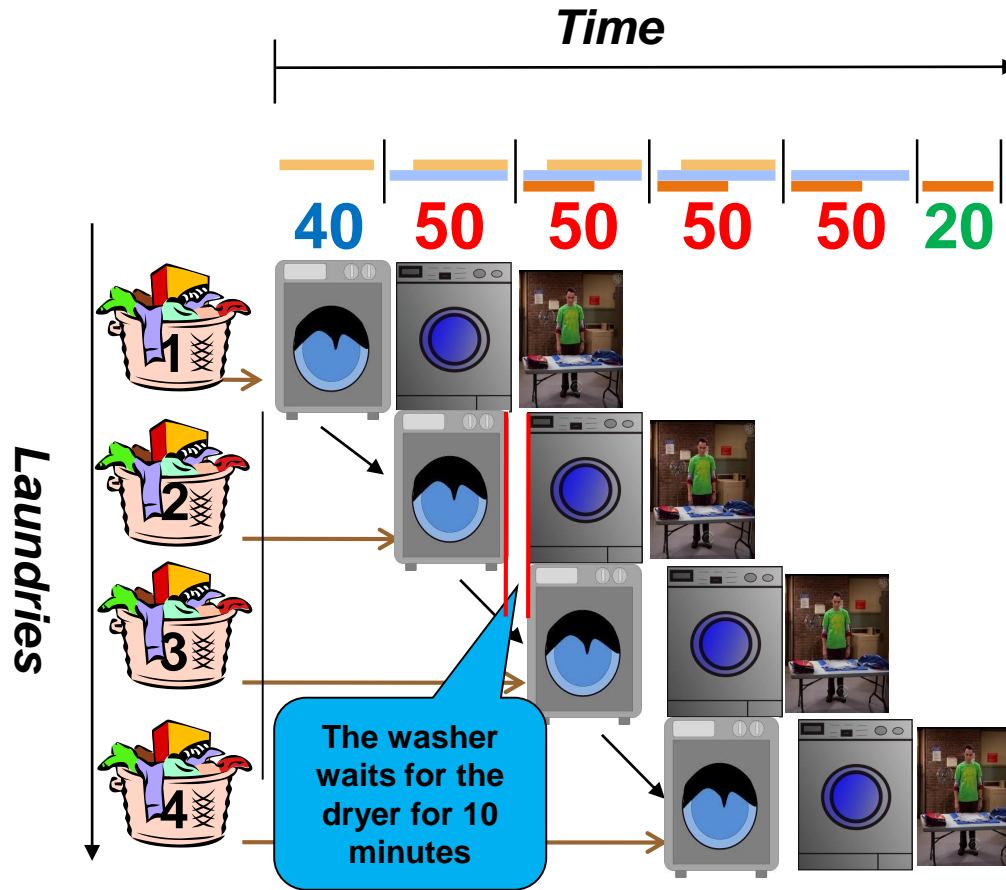
Not very smart way...



$$\begin{aligned} \text{Total} &= N * (\text{Washer} + \text{Dryer} + \text{Folder}) \\ &= \underline{\quad 440 \quad} \text{ mins} \end{aligned}$$



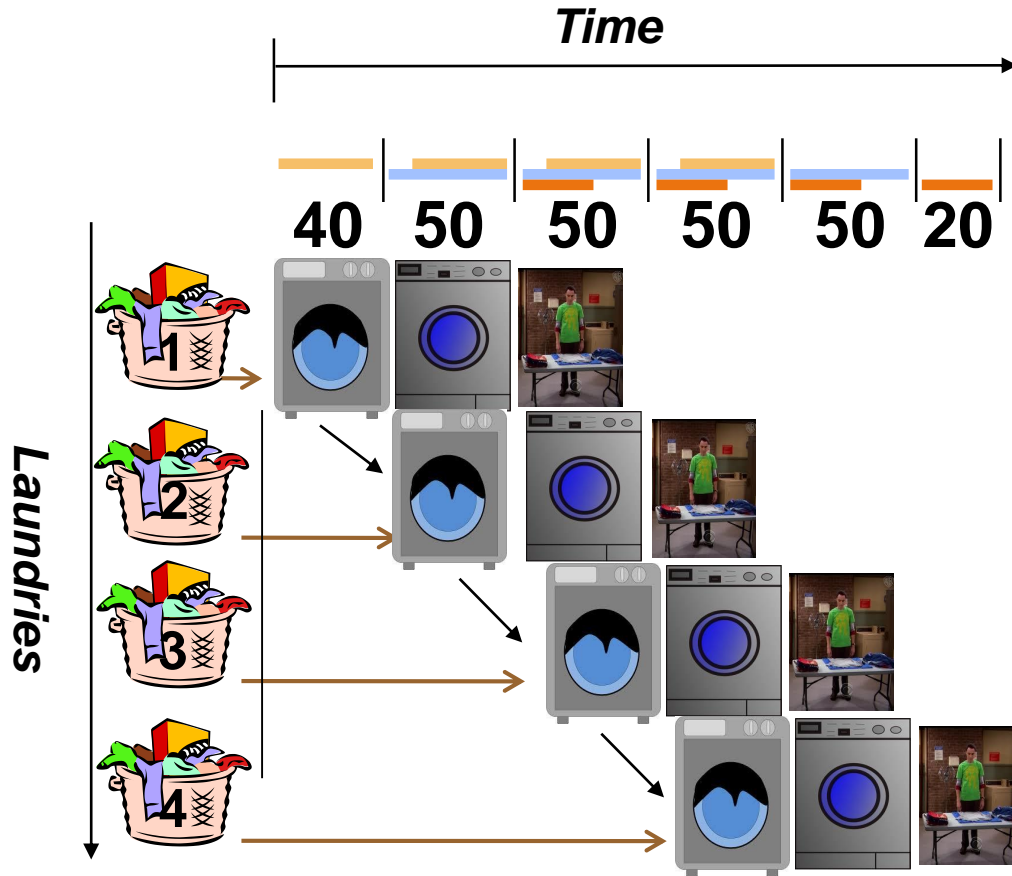
If we pipelining



$$\begin{aligned} \text{Total} &= \text{Washer} + N * \text{Max}(\text{Washer}, \text{Dryer}, \text{Folder}) + \text{Folder} \\ &= \underline{\quad 260 \quad} \text{ mins} \end{aligned}$$



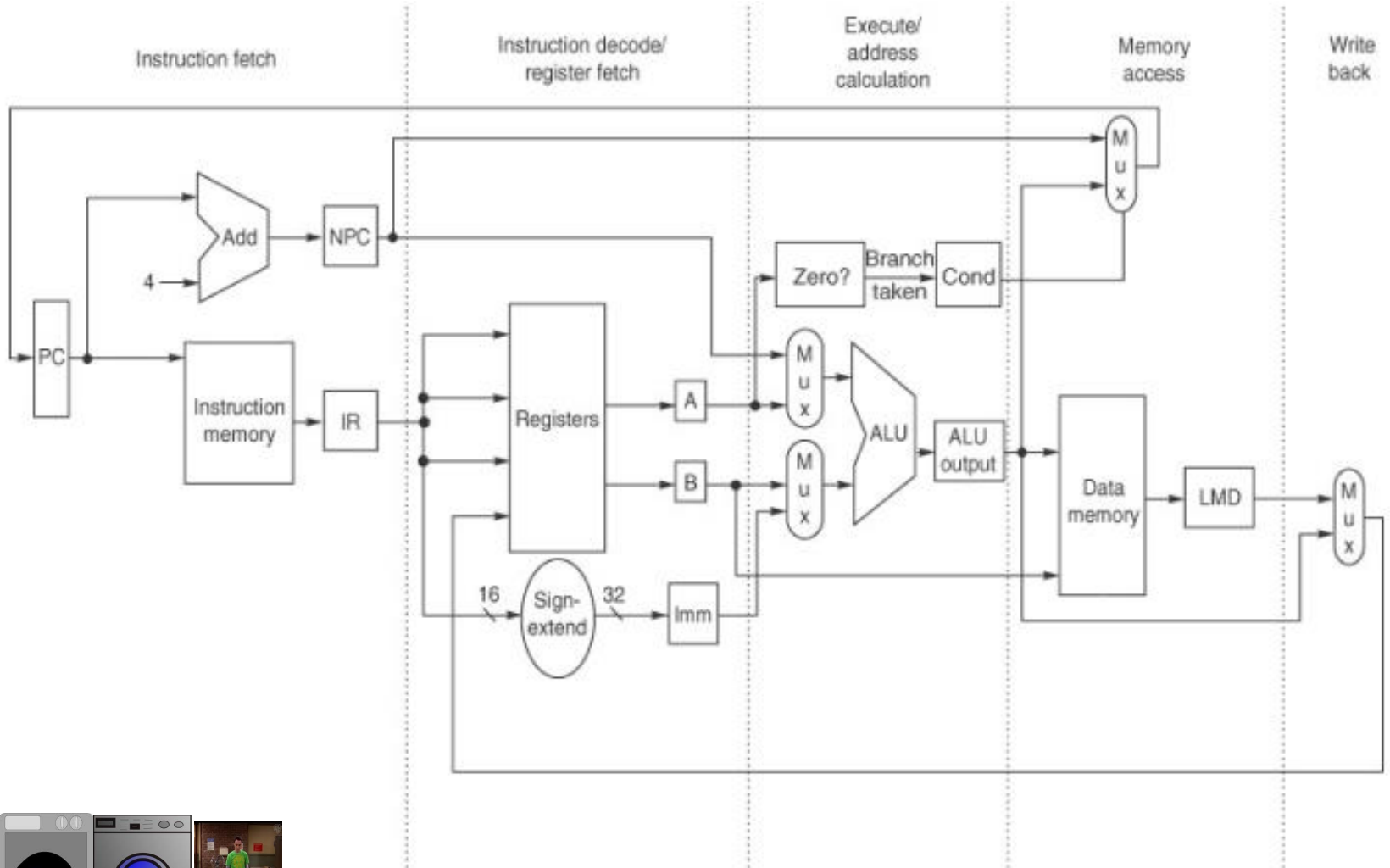
Pipeline Facts



- Multiple (independent) 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 \propto Number of pipe stages



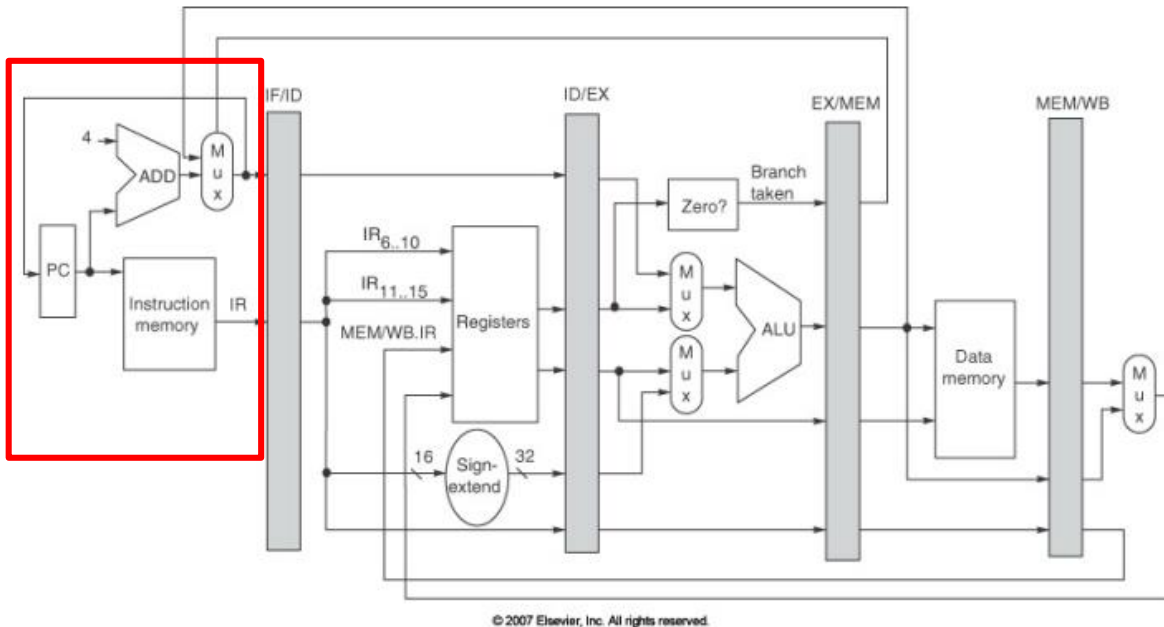
One core – the MIPS data-path



© 2007 Elsevier, Inc. All rights reserved.



Classic RISC 5-stage pipeline



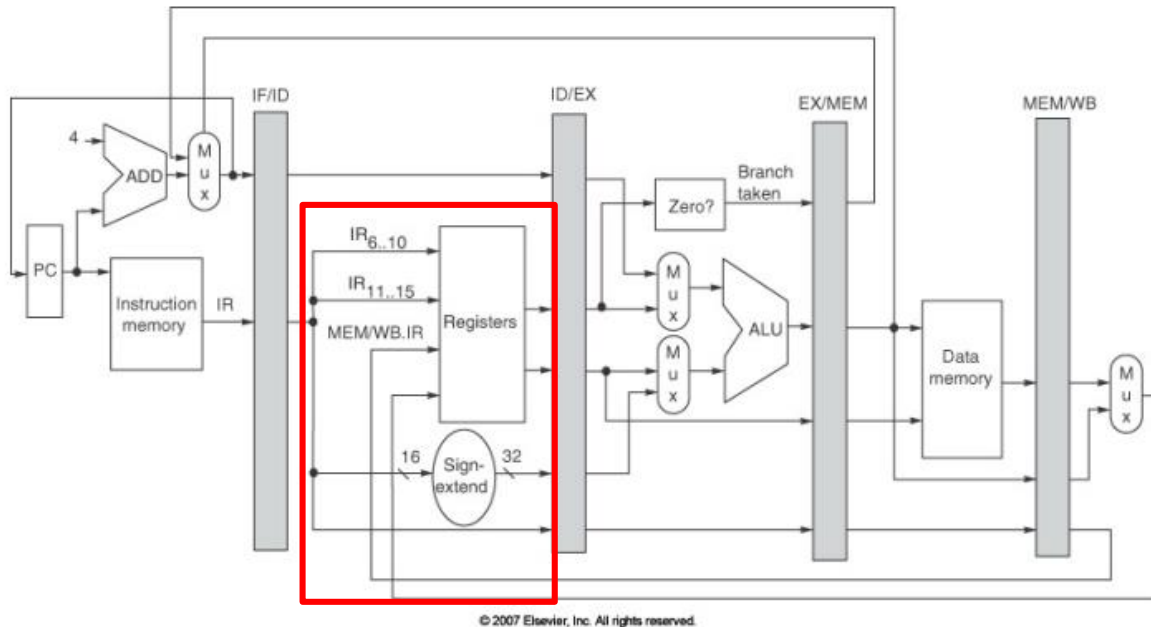
Passed To Next Stage
IR ← Mem[PC]
NPC ← PC + 4

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 (reg) **holds** the instruction that will be used in the next stage.
- NPC (reg) **holds** the value of the next PC (either sequential or jump).



Classic RISC 5-stage pipeline



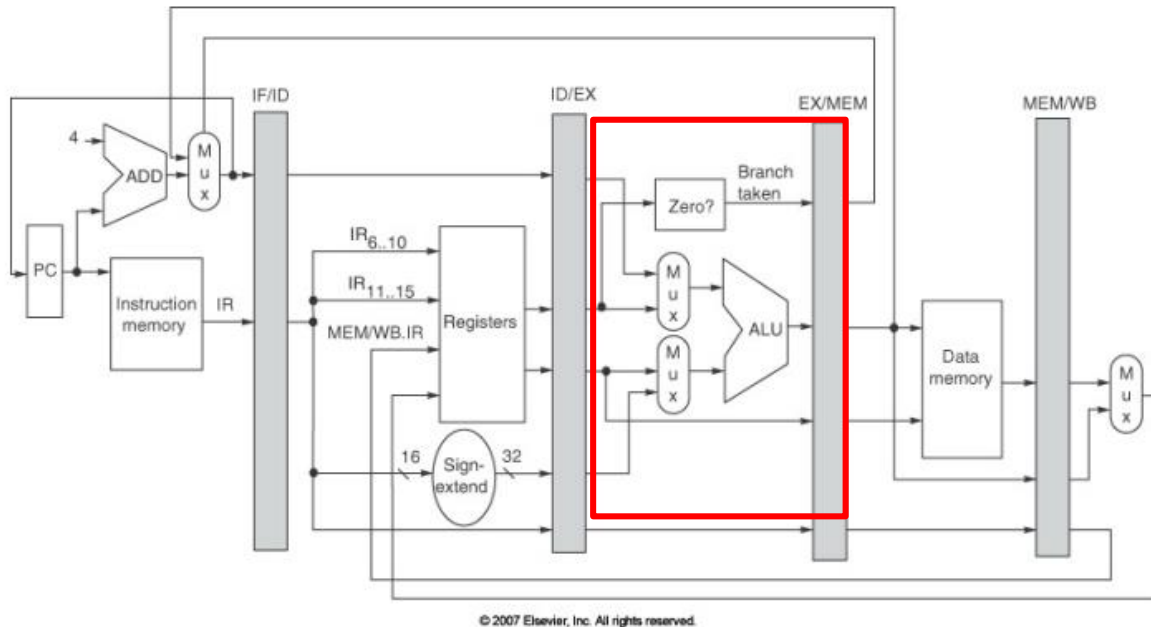
Passed To Next Stage
A <- Regs[IR6..IR10];
B <- Regs[IR10..IR15];
Imm <- ((IR16) ##IR16-31

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



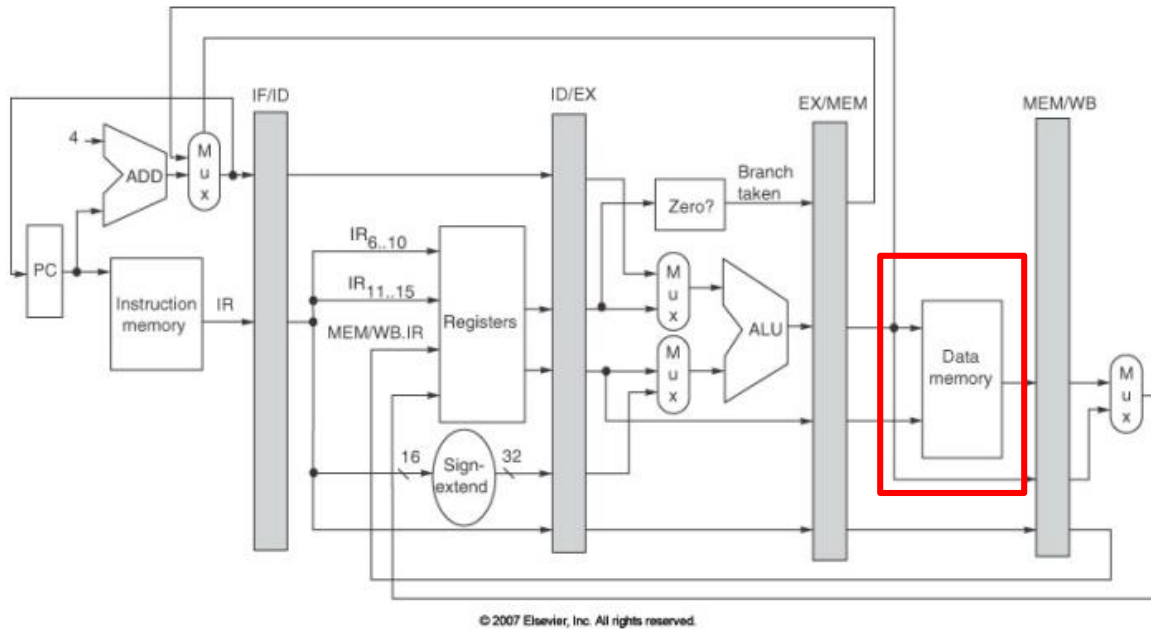
Passed To Next Stage
A <- A func. B
cond = 0;

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, do the operation.
- If a (memory) address calculation, figure out how to obtain the address
- Branch decision



Classic RISC 5-stage pipeline



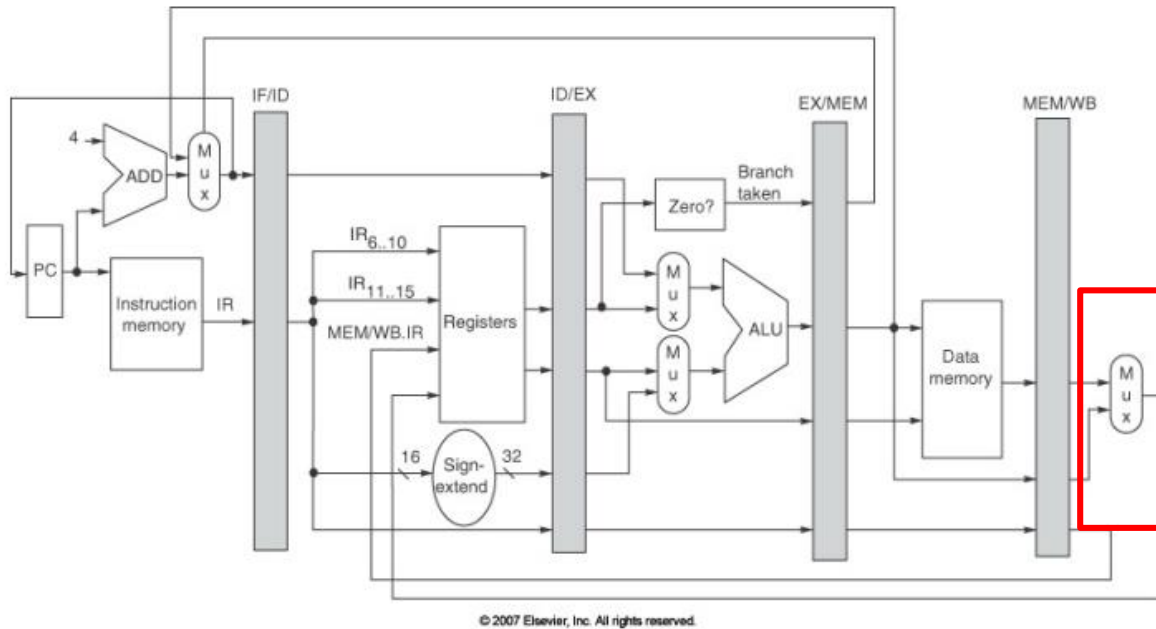
Passed To Next Stage
A = Mem[prev. B]
or
Mem[prev. B] = A

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;

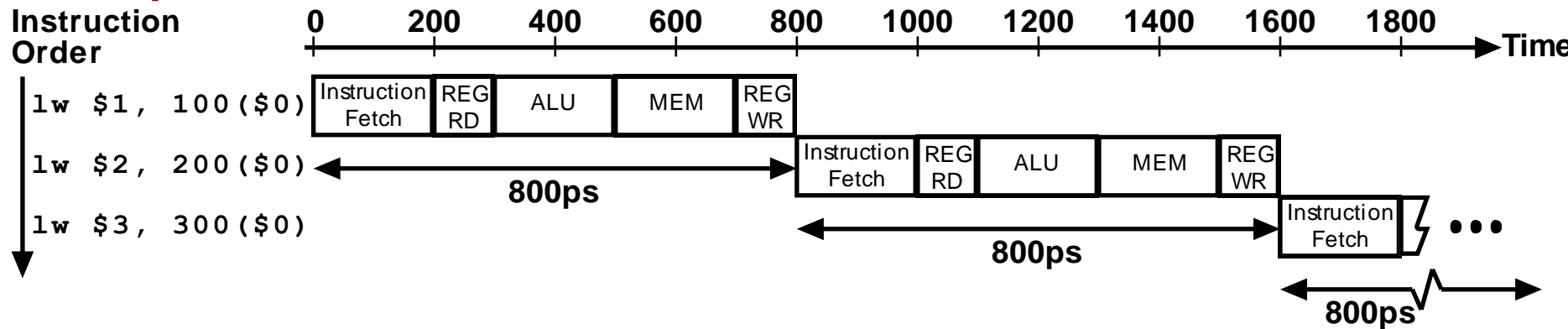
WRITE BACK (WB):

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

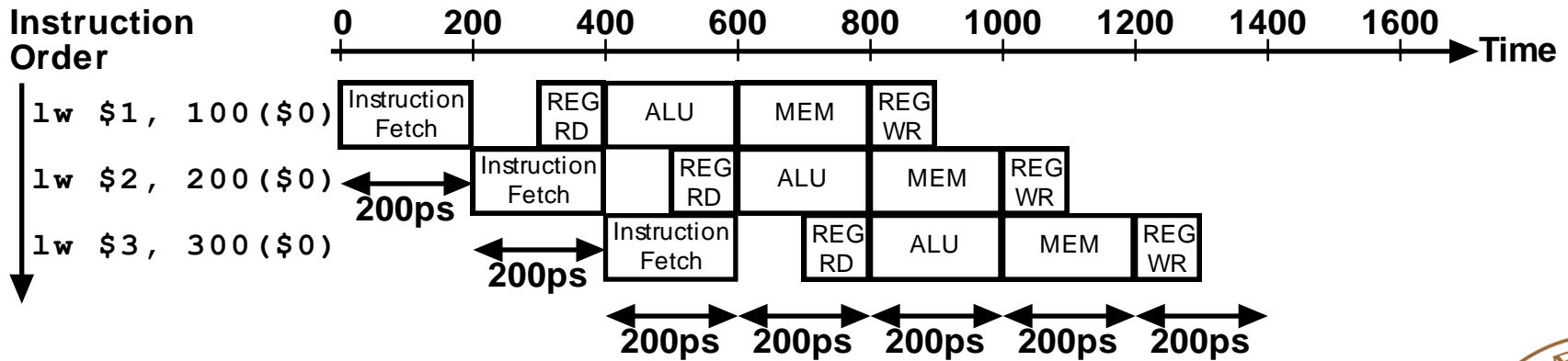


Speed up

Non-Pipelined



Pipelined



Outline

□ Reiteration

□ Pipelining

□ **Hazards**

- Structural hazards
- Data hazards
- Control hazards

□ Implementation issues

□ Multi-cycle operations

□ Summary



Fundamental limitations

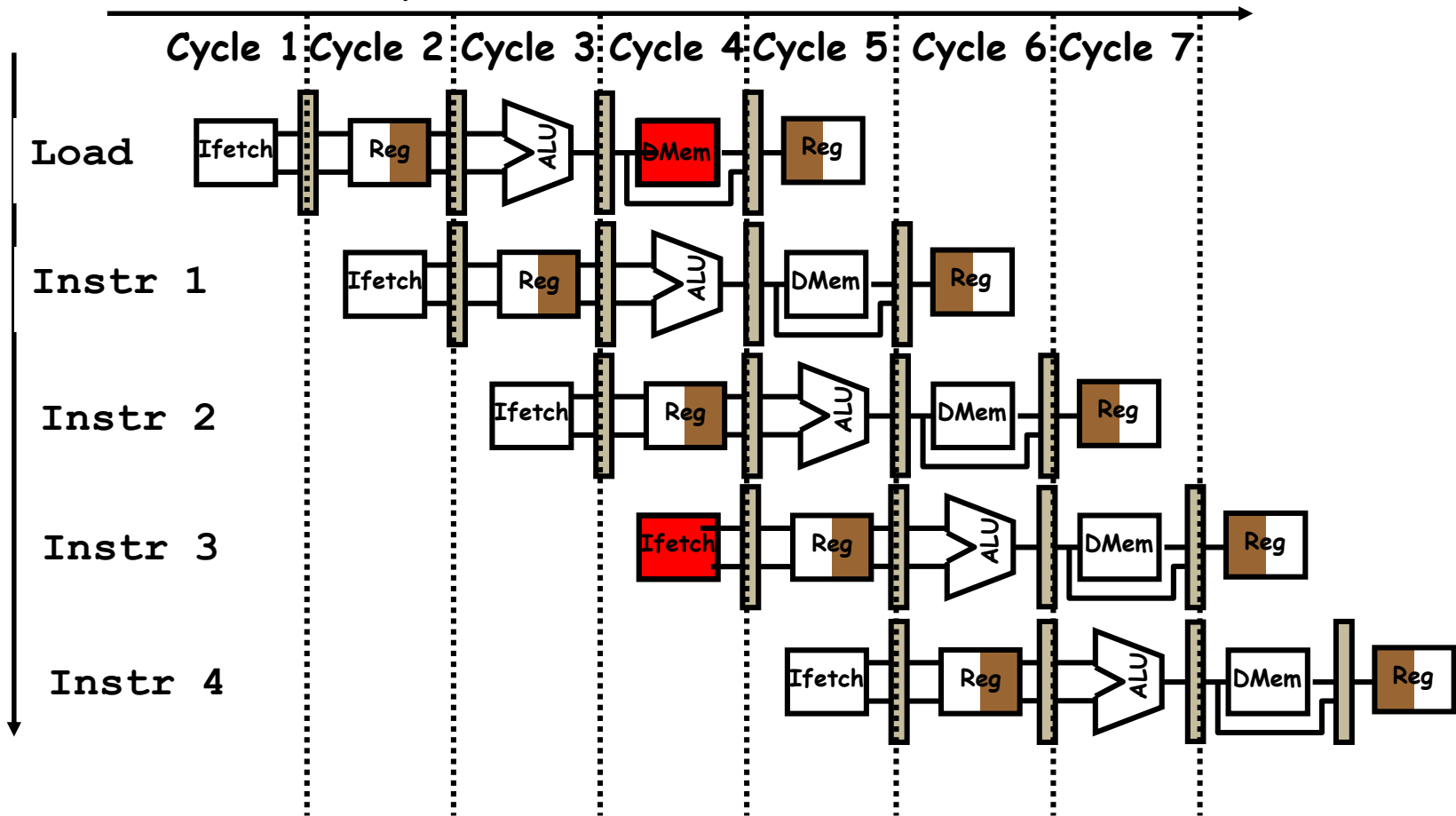
□ Hazards can prevent next instruction from executing during its designated clock cycle:

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



Structure hazard

Time (clock cycles)

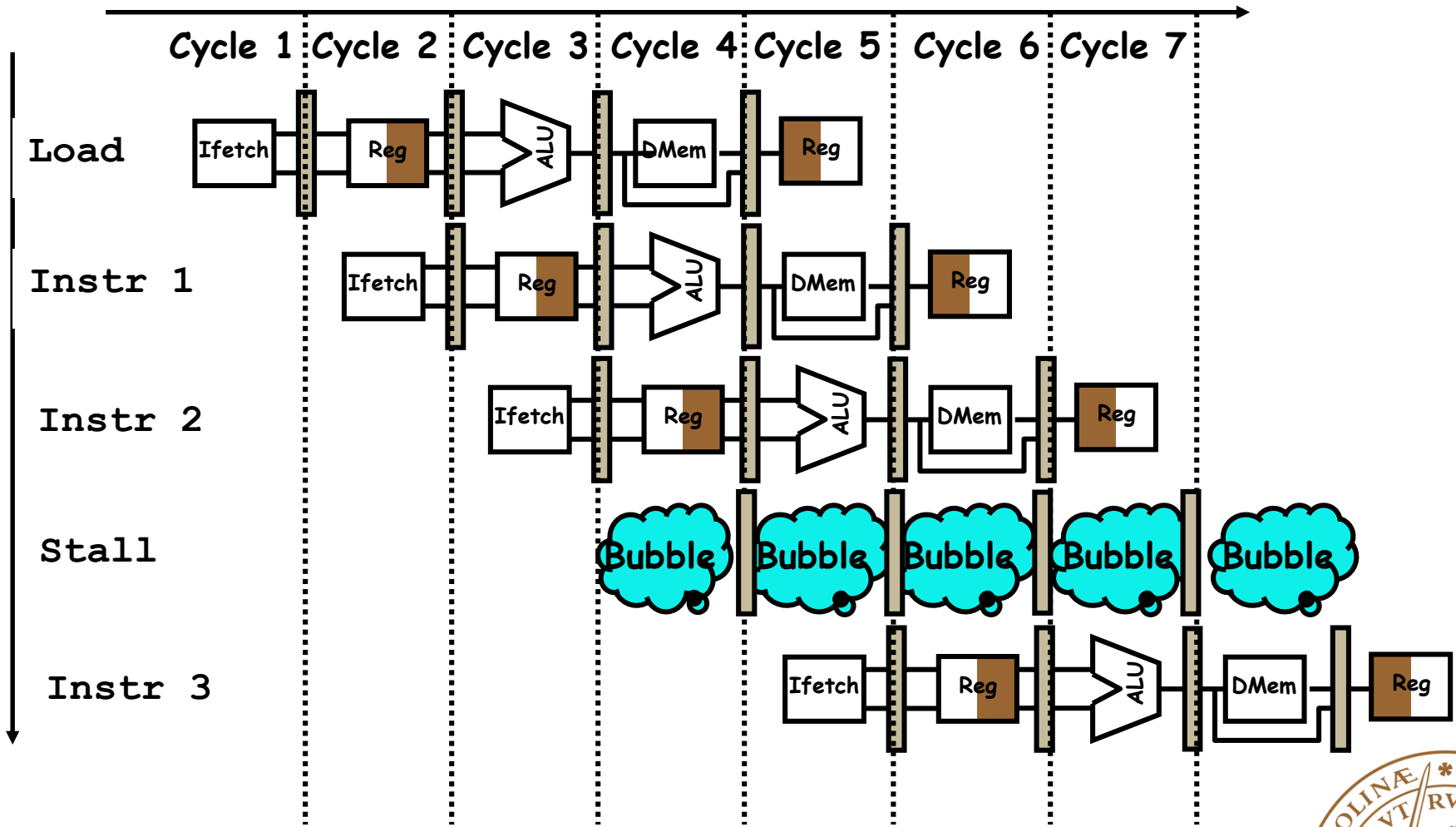


When two or more different instructions want to use same hardware resource in same cycle, e.g., MEM uses the same memory port (only one memory port)



A simple solution

Time (clock cycles)



Other solutions

□ Stall

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

□ Pipeline hardware resource

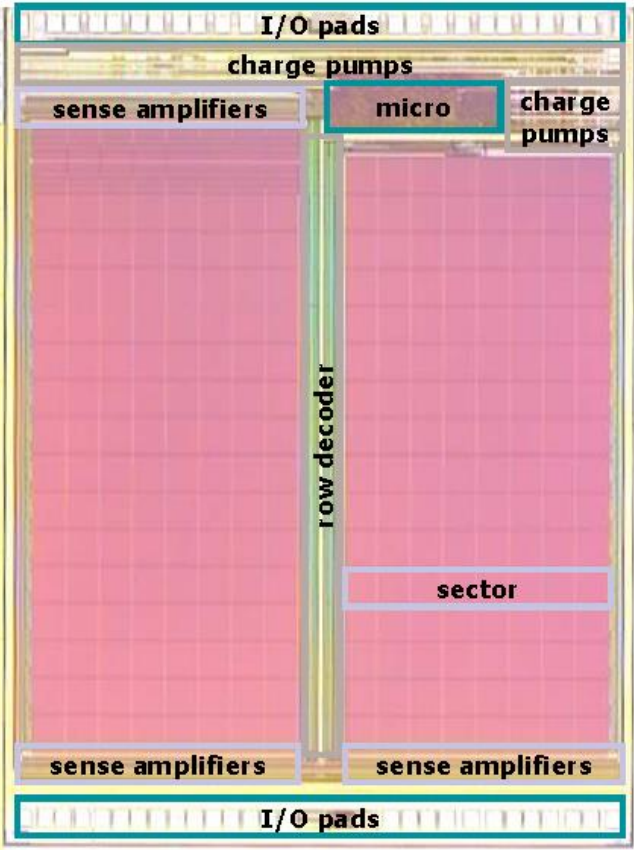
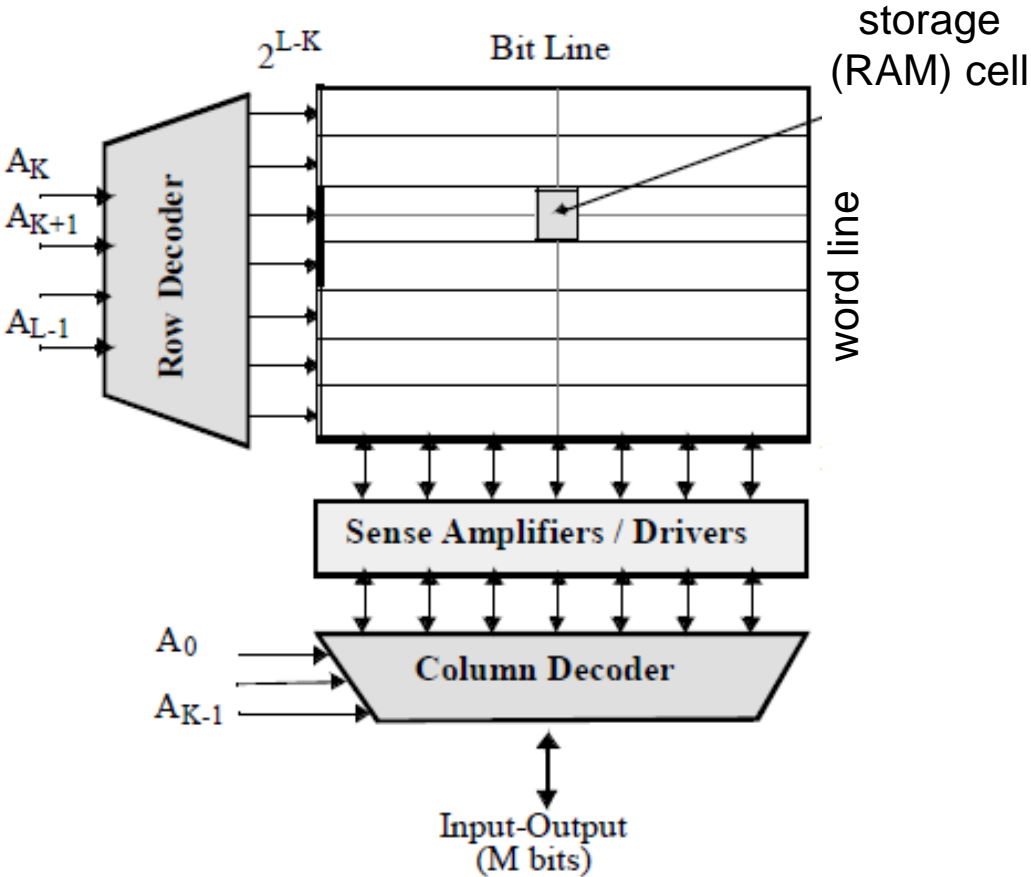
- useful for multi-cycle resources
- good performance
- sometimes complex

□ Replicate resource

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

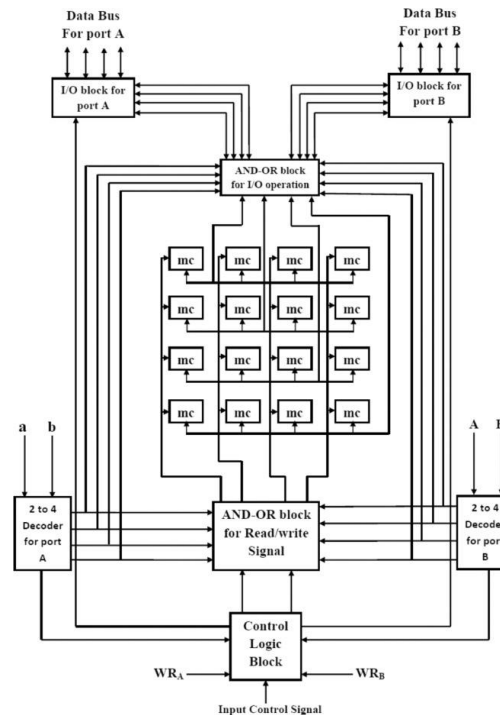


Memory Architecture



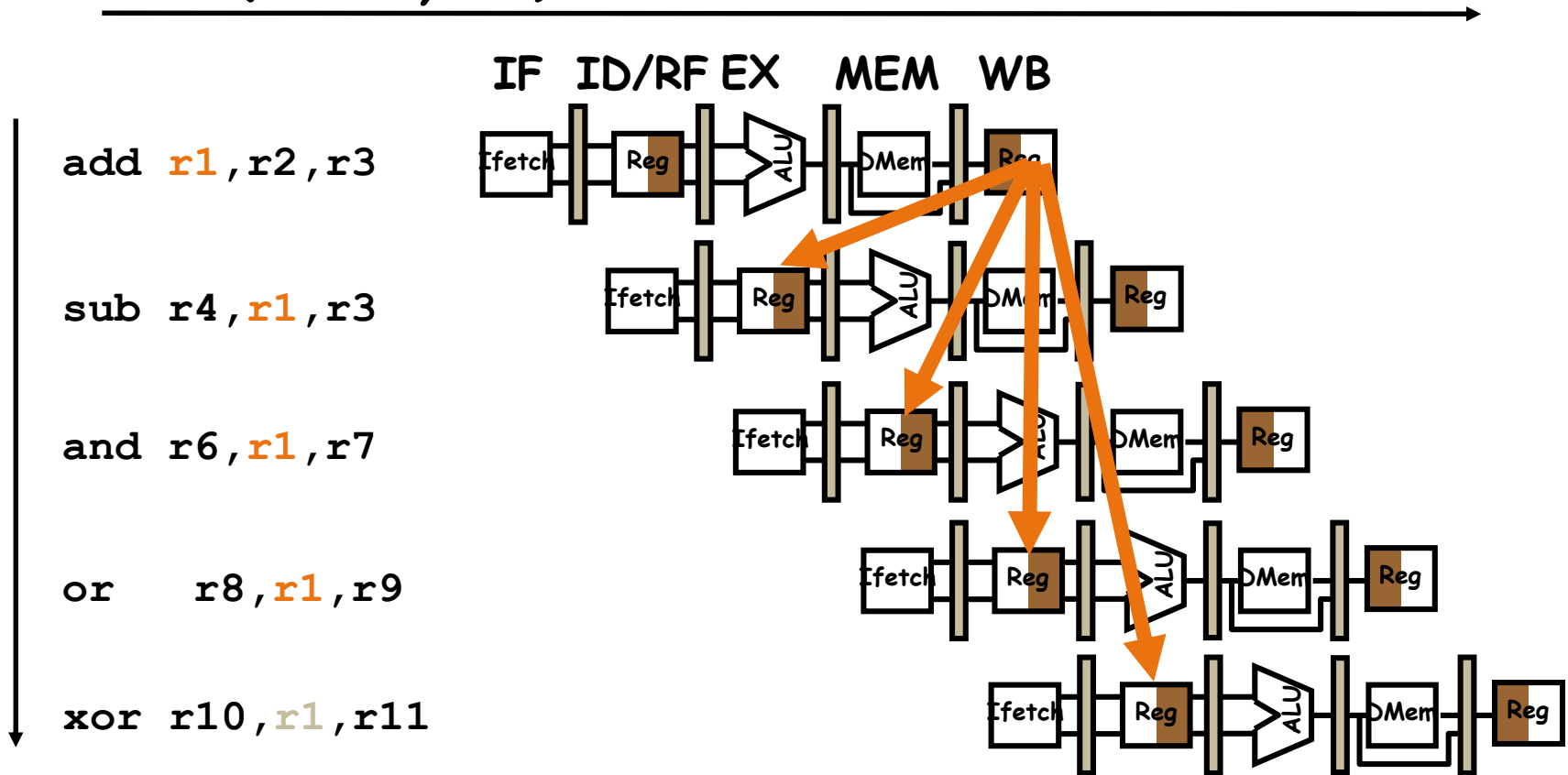
Dual/single port memory (65nm)

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)



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	A
	Op_{i+1}	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

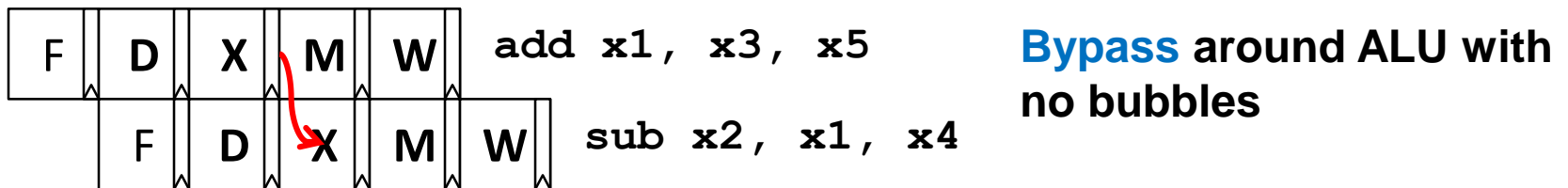
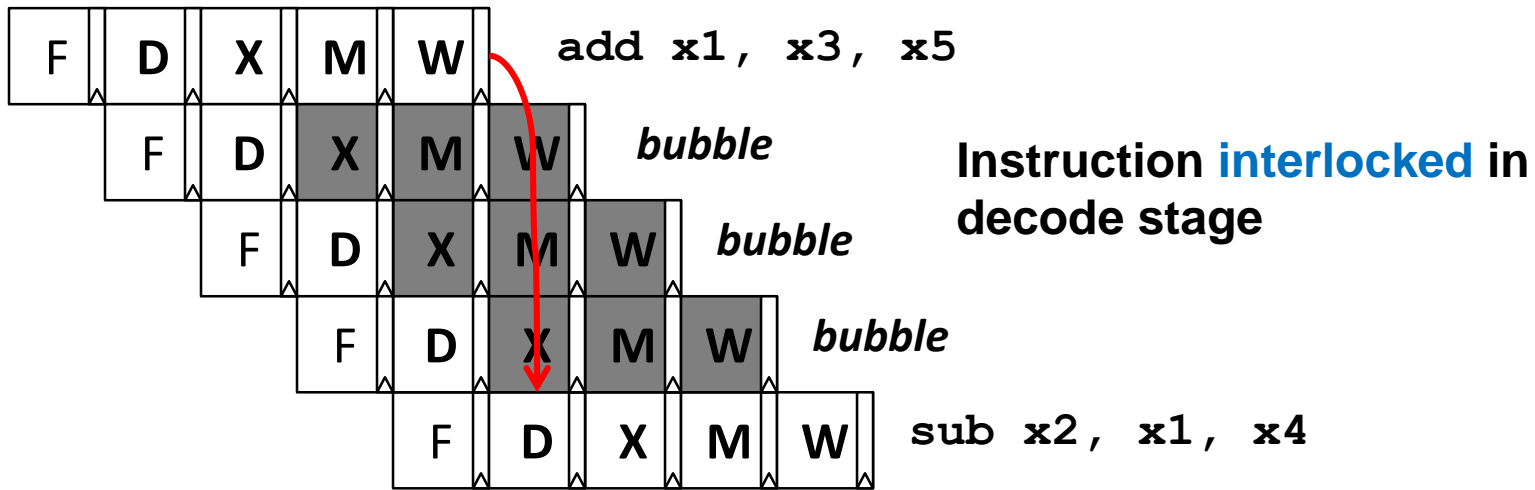
□ Forwarding

- Resolve hazard earlier by bypassing value as soon as available

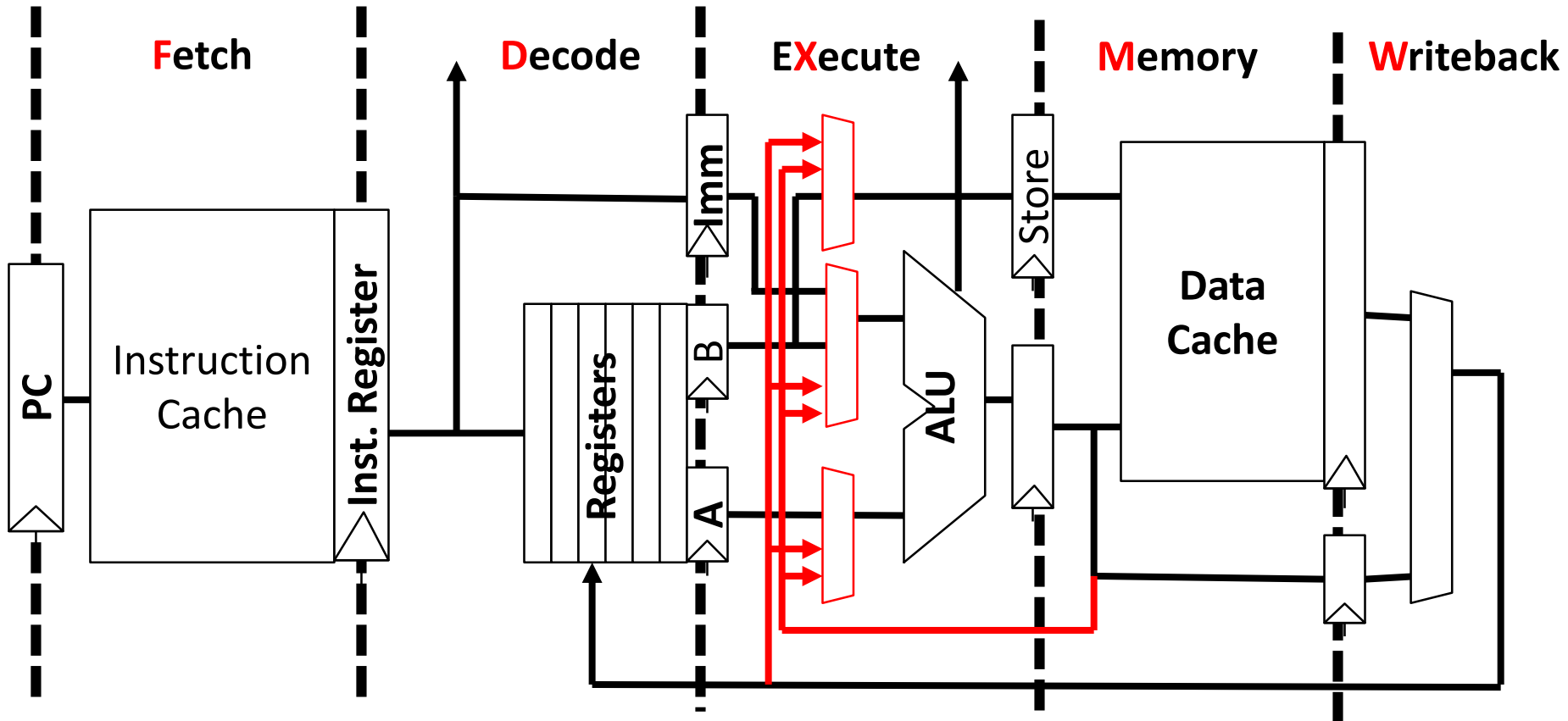


Interlock and forwarding

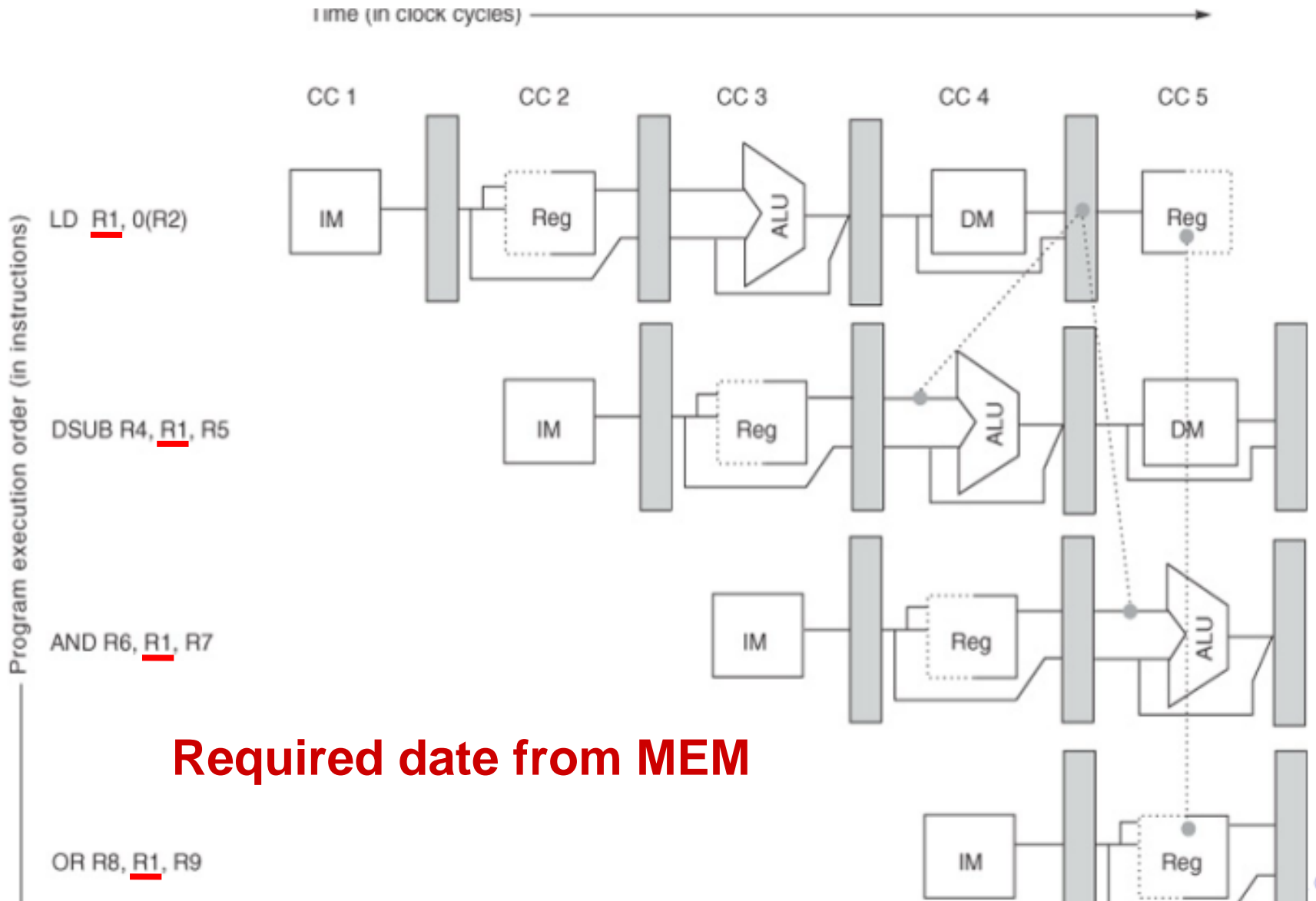
```
add x1, x3, x5
sub x2, x1, x4
```



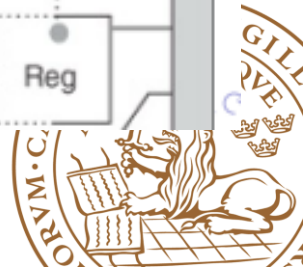
Hardware support of forwarding



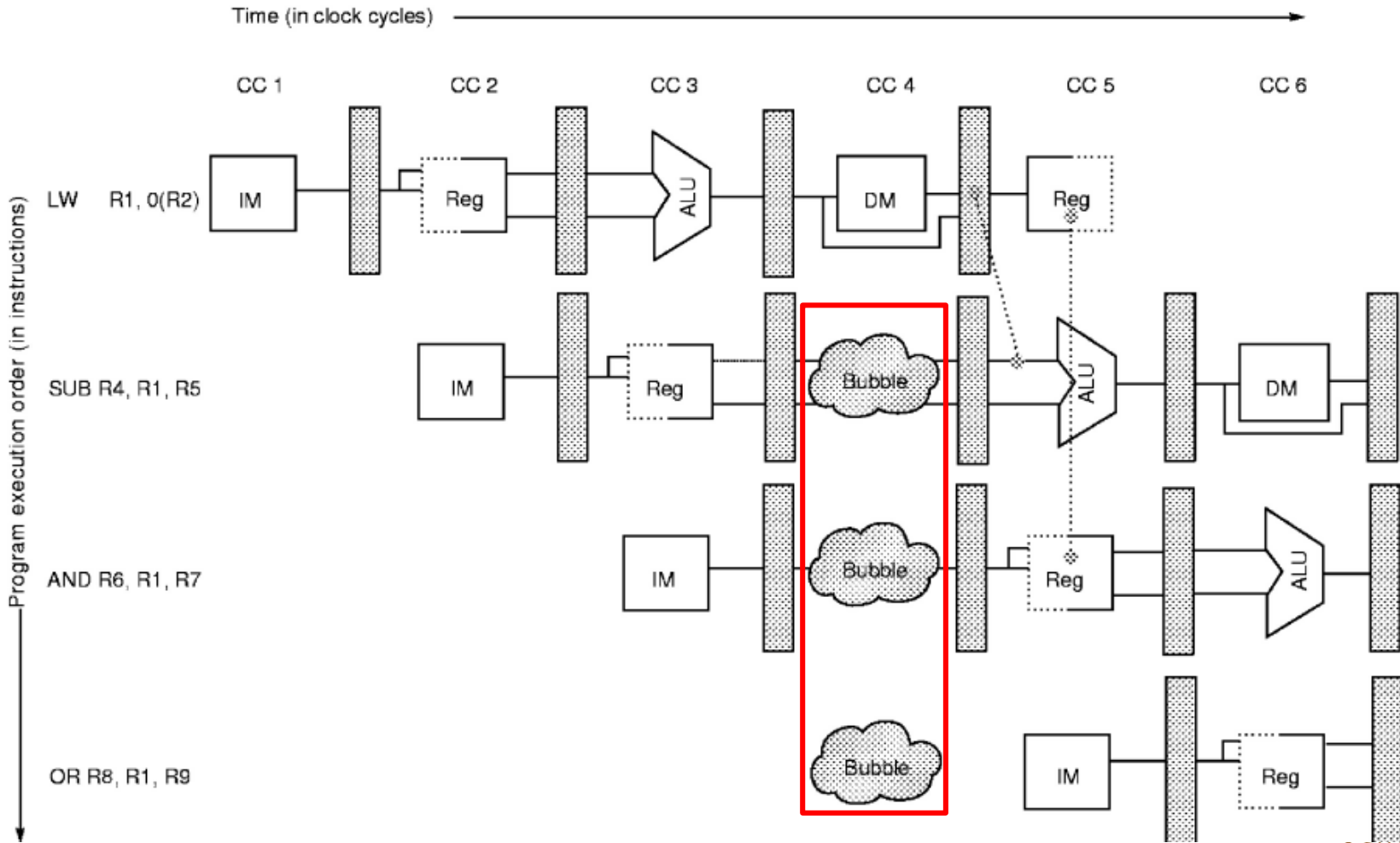
Data hazard with forwarding



Required data from MEM



Data hazard with forwarding



Software scheduling of load

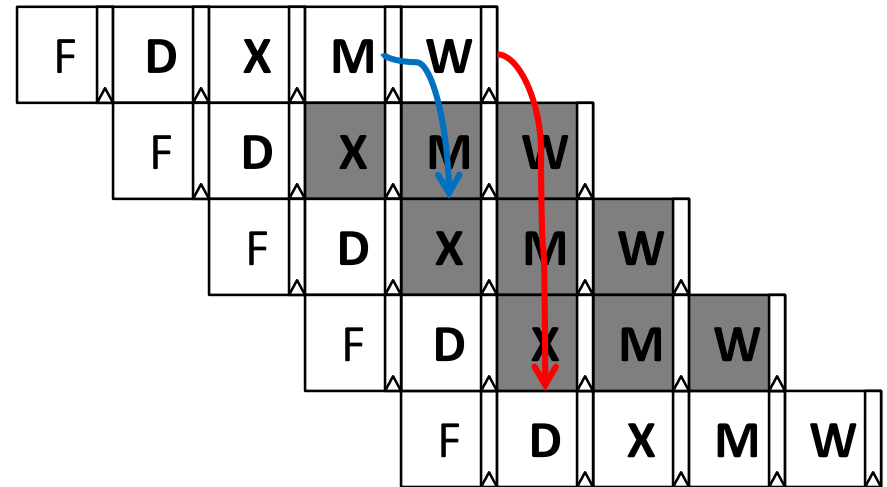
Try producing fast code for

$a = b + c;$

$d = e - f;$

assuming $a, b, c, d, e,$ 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



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	<i>stall</i>	<i>stall</i>	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:

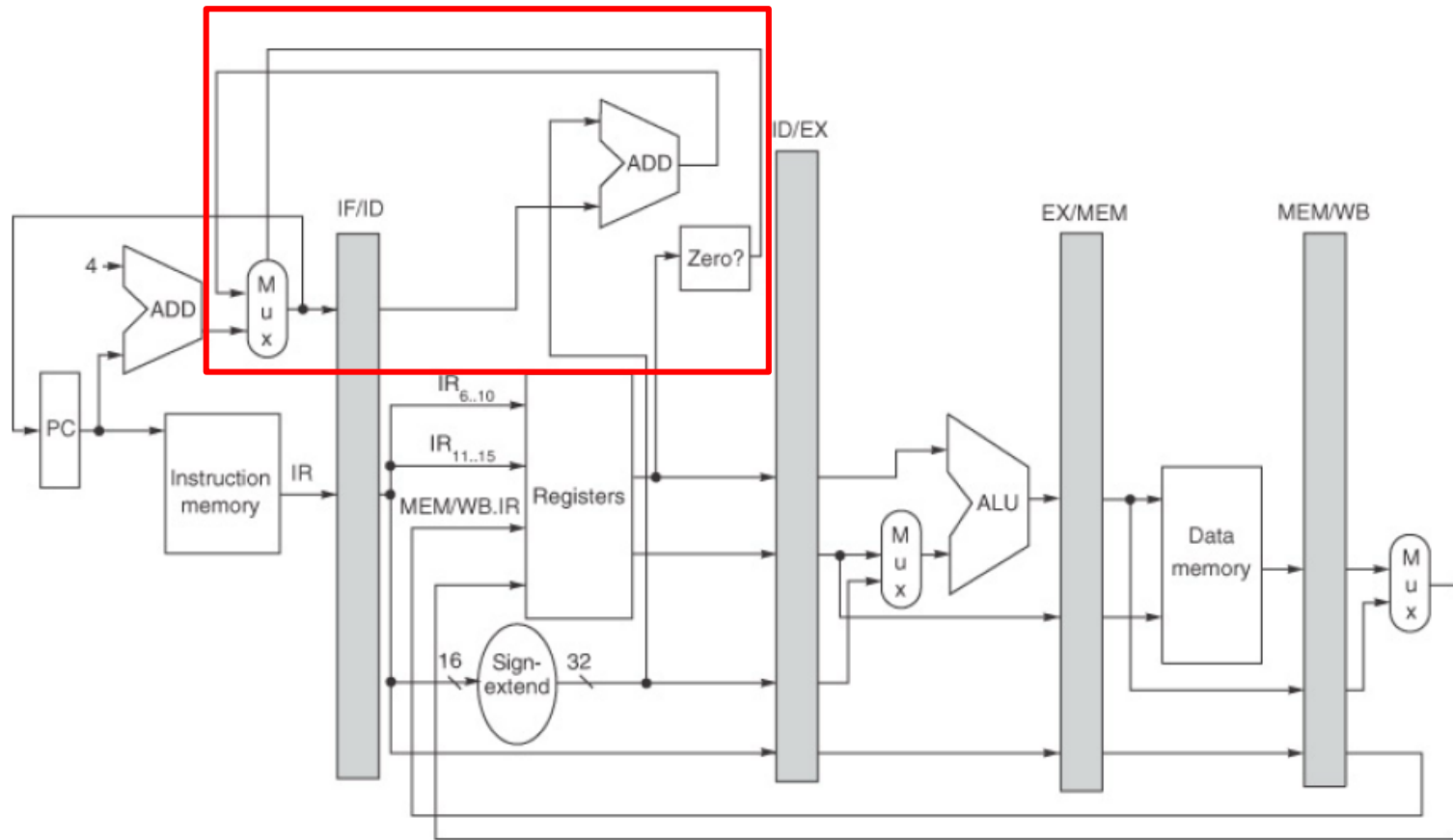
- two stalls
- one extra instruction fetch (IF)

□ If branch is not taken, the extra IF is not needed



Hardware support to reduce control hazard

- Calculate target address and test condition in ID
- 1 clock cycle branch penalty instead of 3!



© 2007 Elsevier, Inc. All rights reserved.



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 (WRITE) late in the pipeline**
- 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**

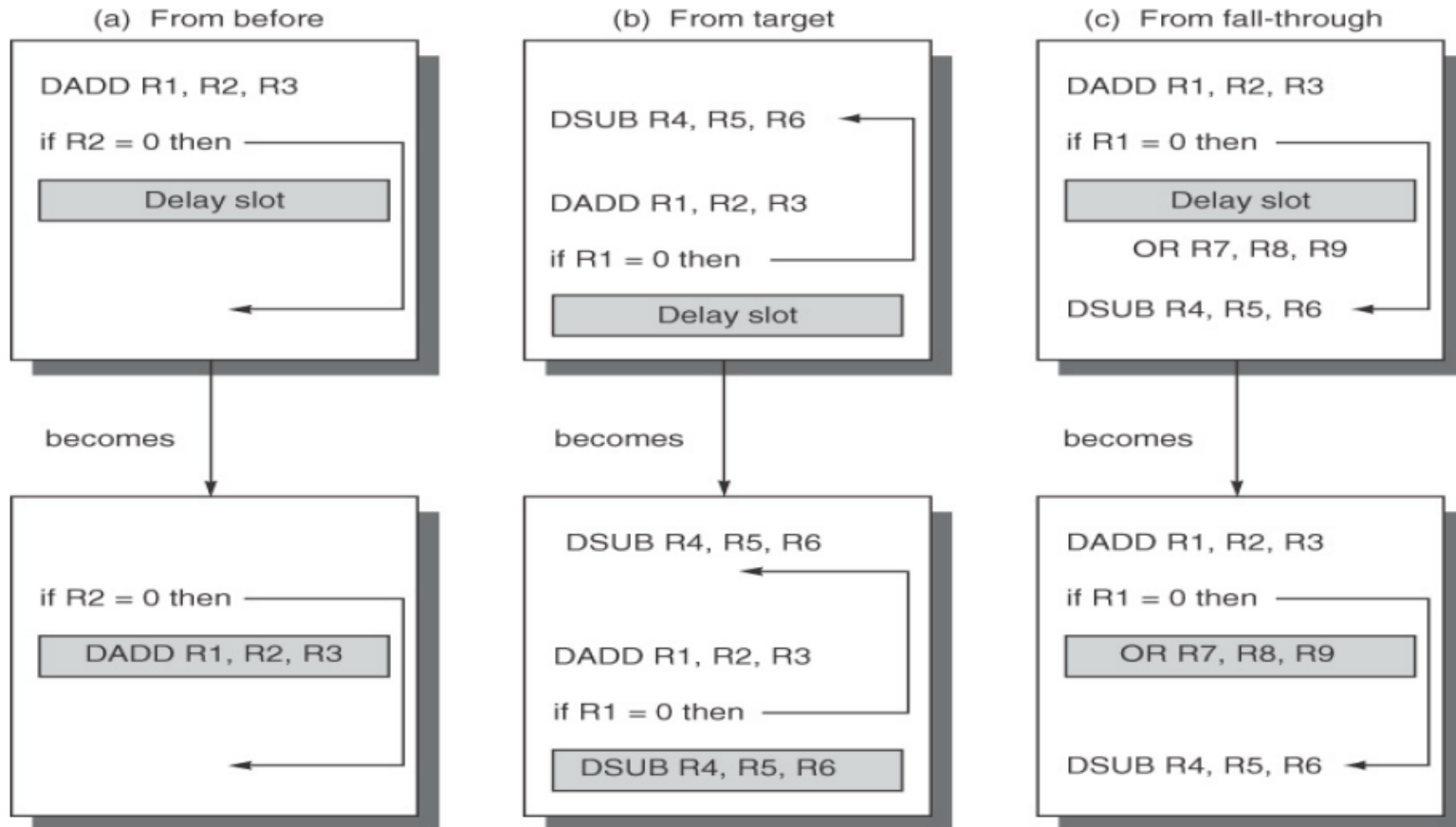
□ Delayed branch

- Schedule **independent** instruction to be exe after the Branch to fill the one-cycle gap



Compiler support for delay branch

- Scheduling “from before” is safe (if independent)
- Scheduling “from target” or “fall through” is not always safe



© 2007 Elsevier, Inc. All rights reserved.



Pipeline speed up

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

$$\text{Speedup} = \frac{\text{AverageInstructionTime}_{\text{unpipelined}}}{\text{AverageInstructionTime}_{\text{pipelined}}}$$

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



Outline

□ Reiteration

□ Pipelining

□ Harzards

- Structural hazards
- Data hazards
- Control hazards

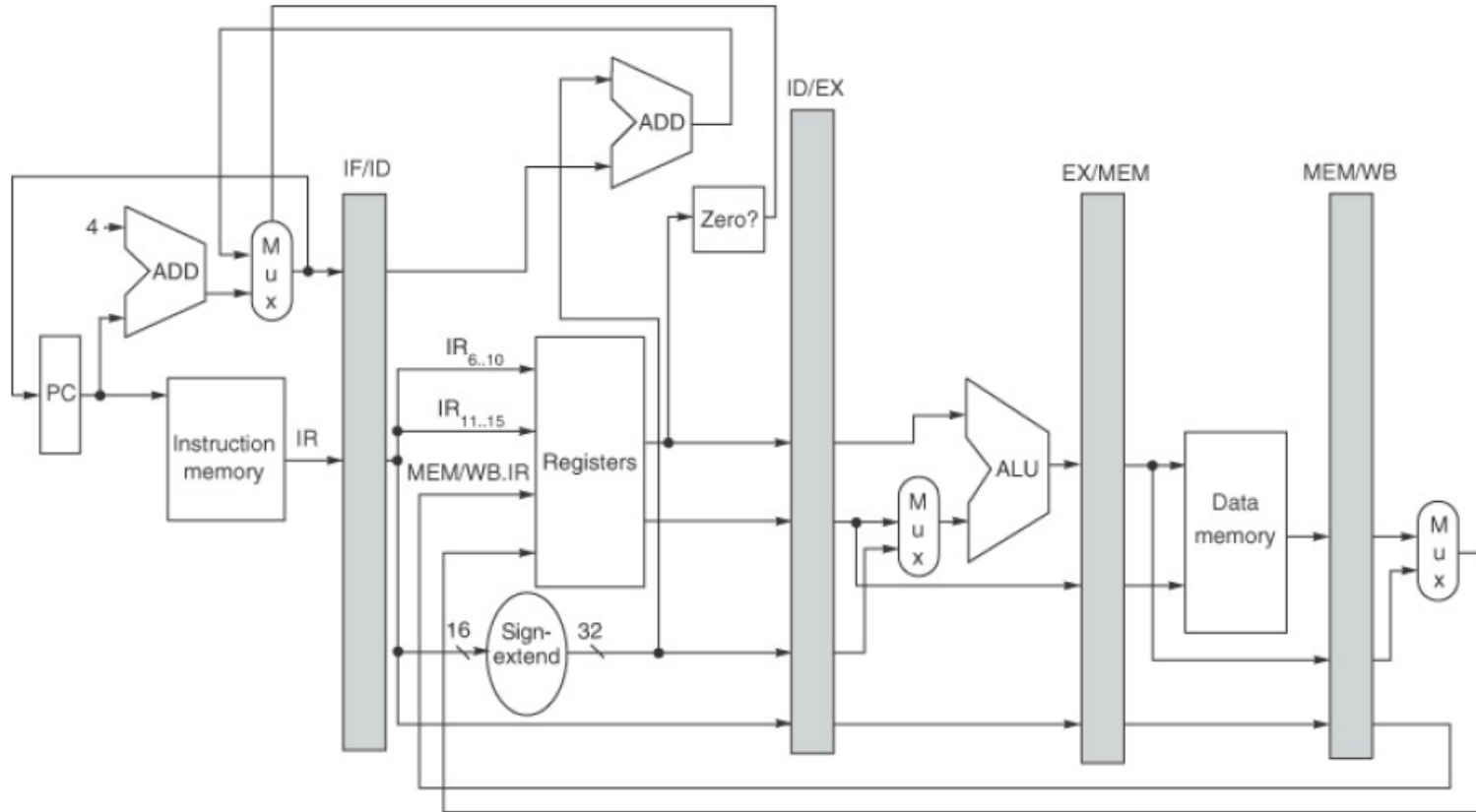
□ **Implementation issues**

□ Multi-cycle operations

□ Summary



What's hard to implement

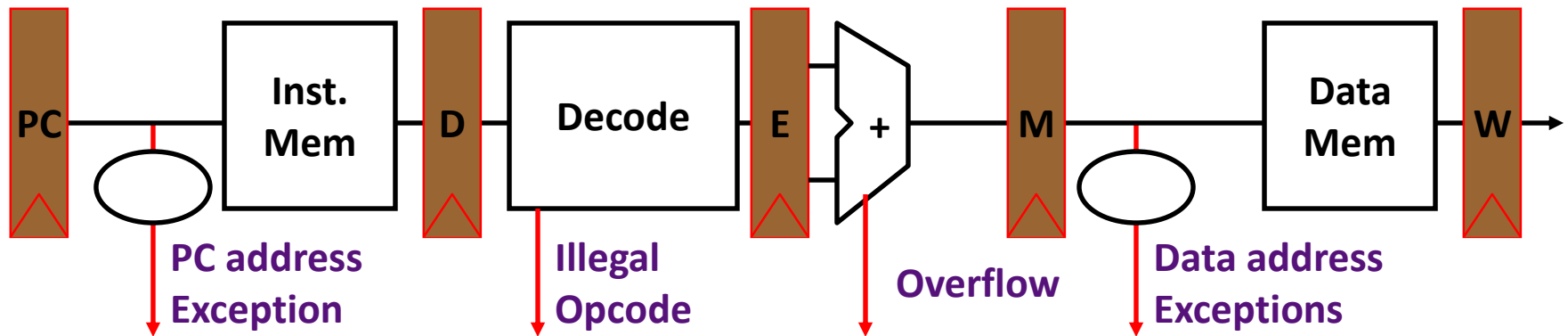


© 2007 Elsevier, Inc. All rights reserved.

□ 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 to handle the exception) 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



Solution for simple MIPS

- ❑ Need to add control and data-paths to support exceptions and interrupts.
- ❑ When an exception or interrupt occurs, the following must be done:
 - $EPC \leq PC$
 - Cause \leq (cause code for event)
 - Status \leq (fault)
 - PC \leq (handler address)
- ❑ To return from an exception or datapath, the following must be done:
 - PC \leq EPC
 - Status \leq (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



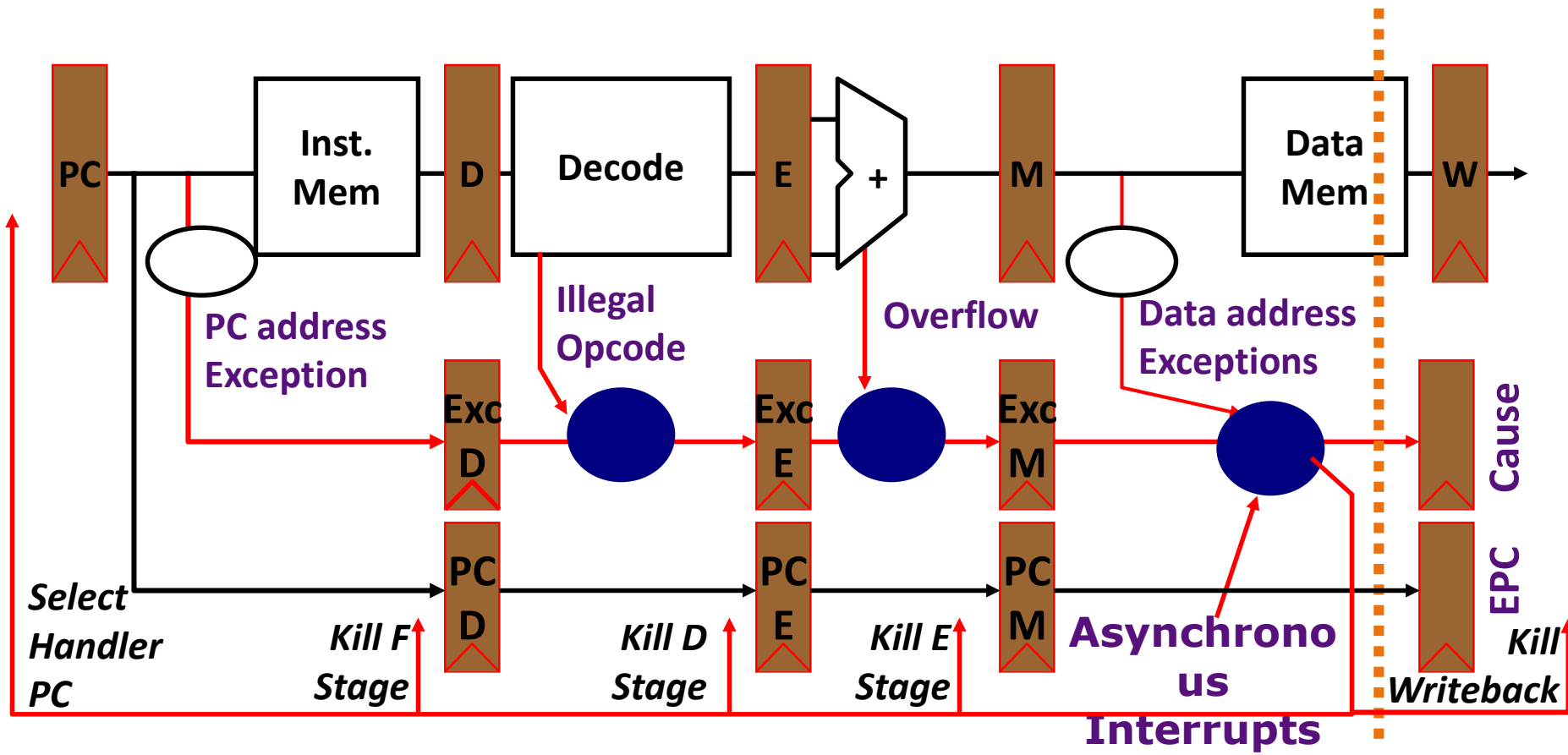
Solution for simple MIPS

- Add a hardware status vector containing exceptions
- Pass along with instruction in the pipeline
 - Hold exception flags in pipeline **until commit point**
- **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

**Handle at commit point NOT
at exception point**



Solution for simple MIPS



LD (faults in MEM)	F	D	X	M	W	
DADD (faults in IF)		F	D	X	M	W



Outline

□ Reiteration

□ Pipelining

□ Harzards

- Structural hazards
- Data hazards
- Control hazards

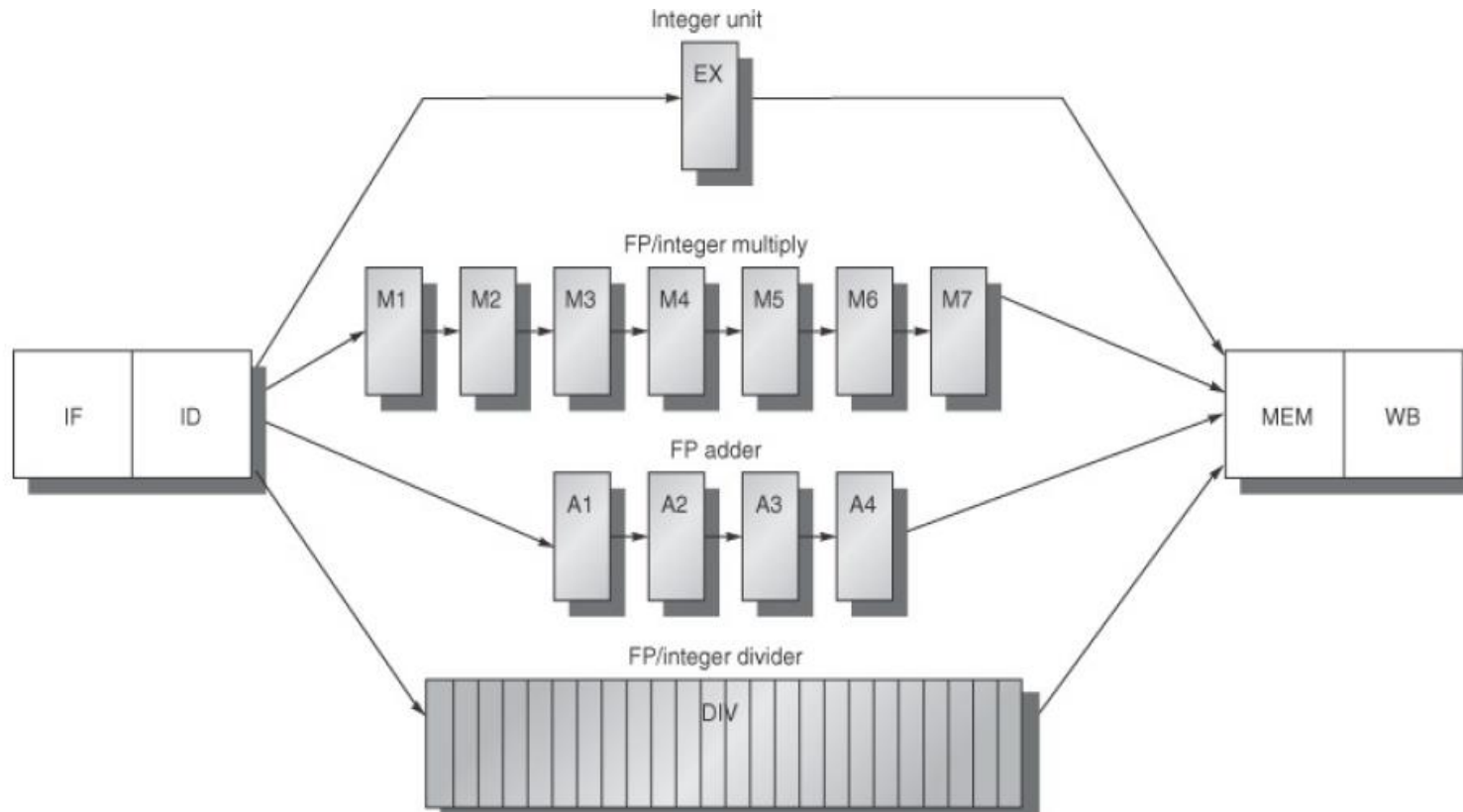
□ Implementation issues

□ **Multi-cycle operations**

□ Summary



Multi-cycle instruction in pipeline (FP)



<i>FP Instruction</i>	<i>Latency</i>	<i>Initiation Rate</i>
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 hazards
- 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

Instruction	Clock cycle number										
	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, they are in-order

□ 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?

How about exception?



Exception

Example: DIV.D F0,F2,F4 (24 cycles)
 ADD.D F10,F10,F8 (3 cycles)
 SUB.D F12,F12,F14 (3 cycles)

□ Suppose

- the SUB instruction generates an arithmetic trap
- DIV instruction hasn't completed
- ADD instruction have completed

Imprecise interrupt signaling: Another problem with out-of-order completion



Summary

□ Pipelining (ILP):

- Speeds up throughput, not latency
- Speedup \leq #stages

□ Hazards limit performance, generate stalls:

- Structural: need more HW
- Data (RAW,WAR,WAW): need forwarding and compiler scheduling
- Control: delayed branch, branch prediction

$$\text{Speedup} = \frac{\text{CPI}_{\text{unpipelined}}}{\text{Ideal CPI} + \# \text{ stall-cycles/instr}} * \frac{T_{C_{\text{unpipelined}}}}{T_{C_{\text{pipelined}}}}$$
$$\approx \frac{\# \text{ stages}}{1 + \# \text{ stall-cycles/instruction}}$$

□ Complications:

- Precise exceptions may be difficult to implement

