

# Pipelining

# A Pipelined Datapath

- Five stages, one 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 to add pipeline registers between stages

# 5 Steps of DLX Datapath

I-type instruction

6	5	5	16
Opcode	rs	rt	Immediate

Encodes: Loads and stores of bytes, half words, words, double words. All immediates ( $rt \leftarrow rs \text{ op immediate}$ )

Conditional branch instructions ( $rs$  is register,  $rd$  unused)  
Jump register, jump and link register  
( $rd = 0$ ,  $rs = \text{destination}$ ,  $\text{immediate} = 0$ )

R-type instruction

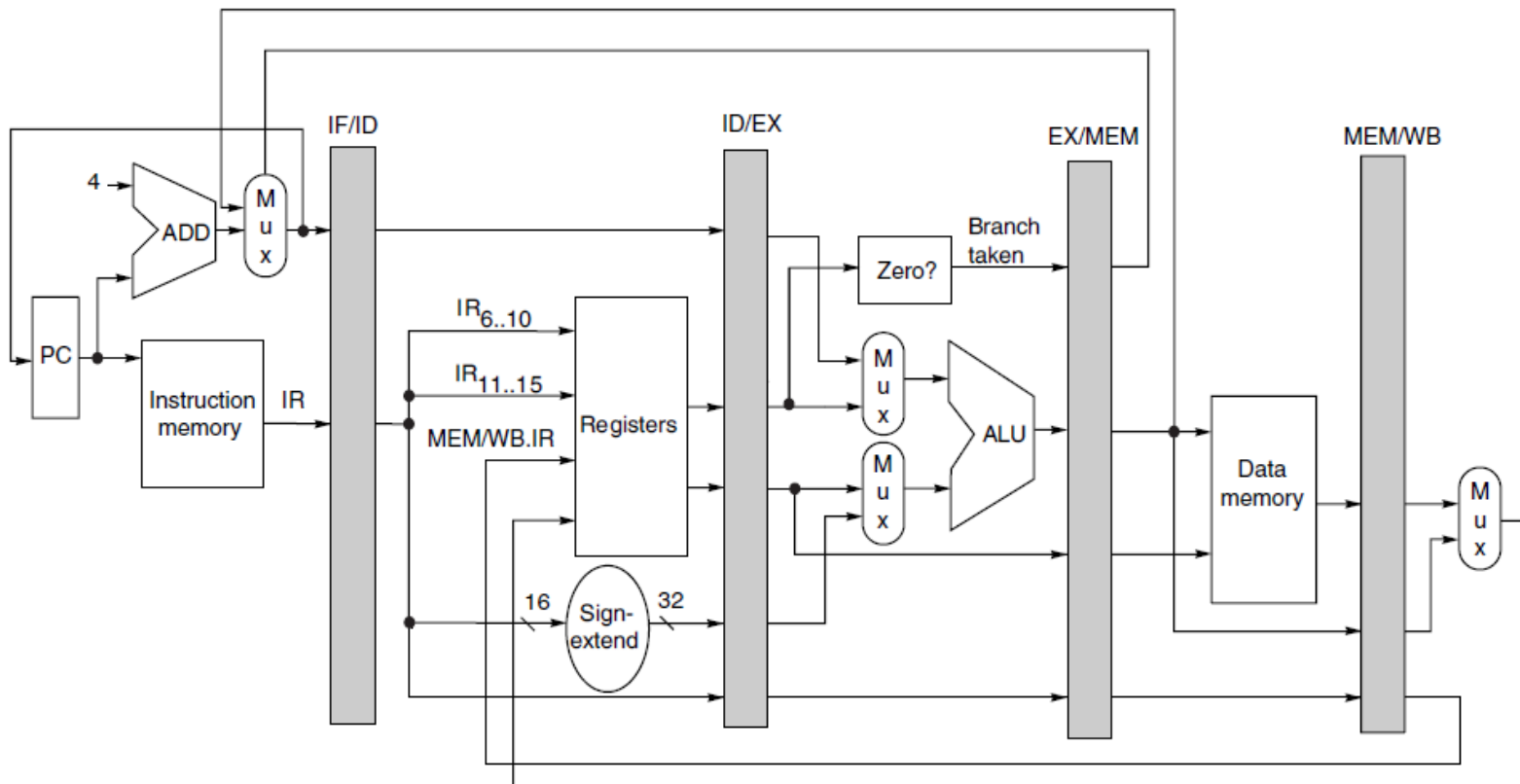
6	5	5	5	5	6
Opcode	rs	rt	rd	shamt	funct

Register-register ALU operations:  $rd \leftarrow rs \text{ funct } rt$   
Function encodes the data path operation: Add, Sub, ...  
Read/write special registers and moves

J-type instruction

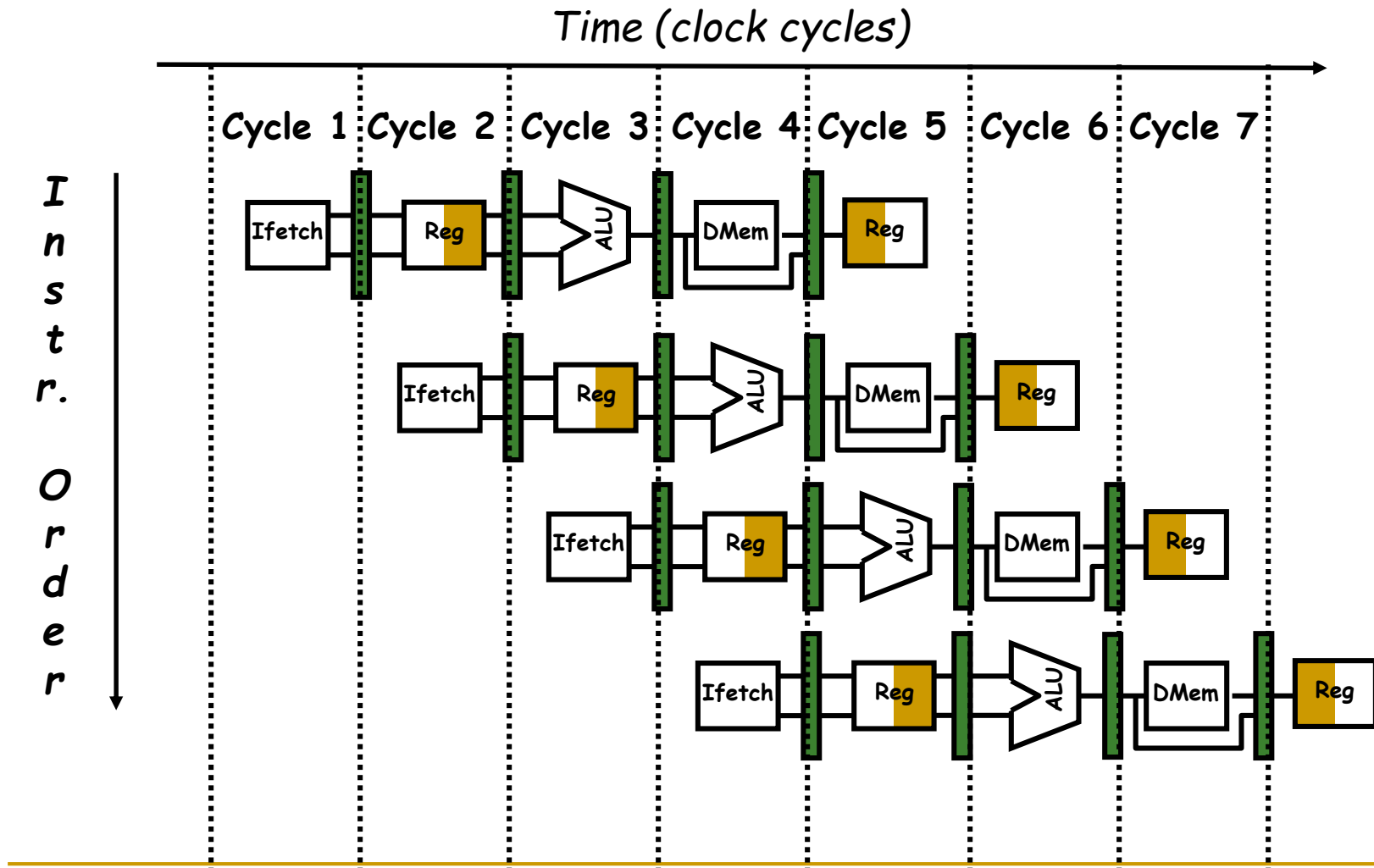
6	26
Opcode	Offset added to PC

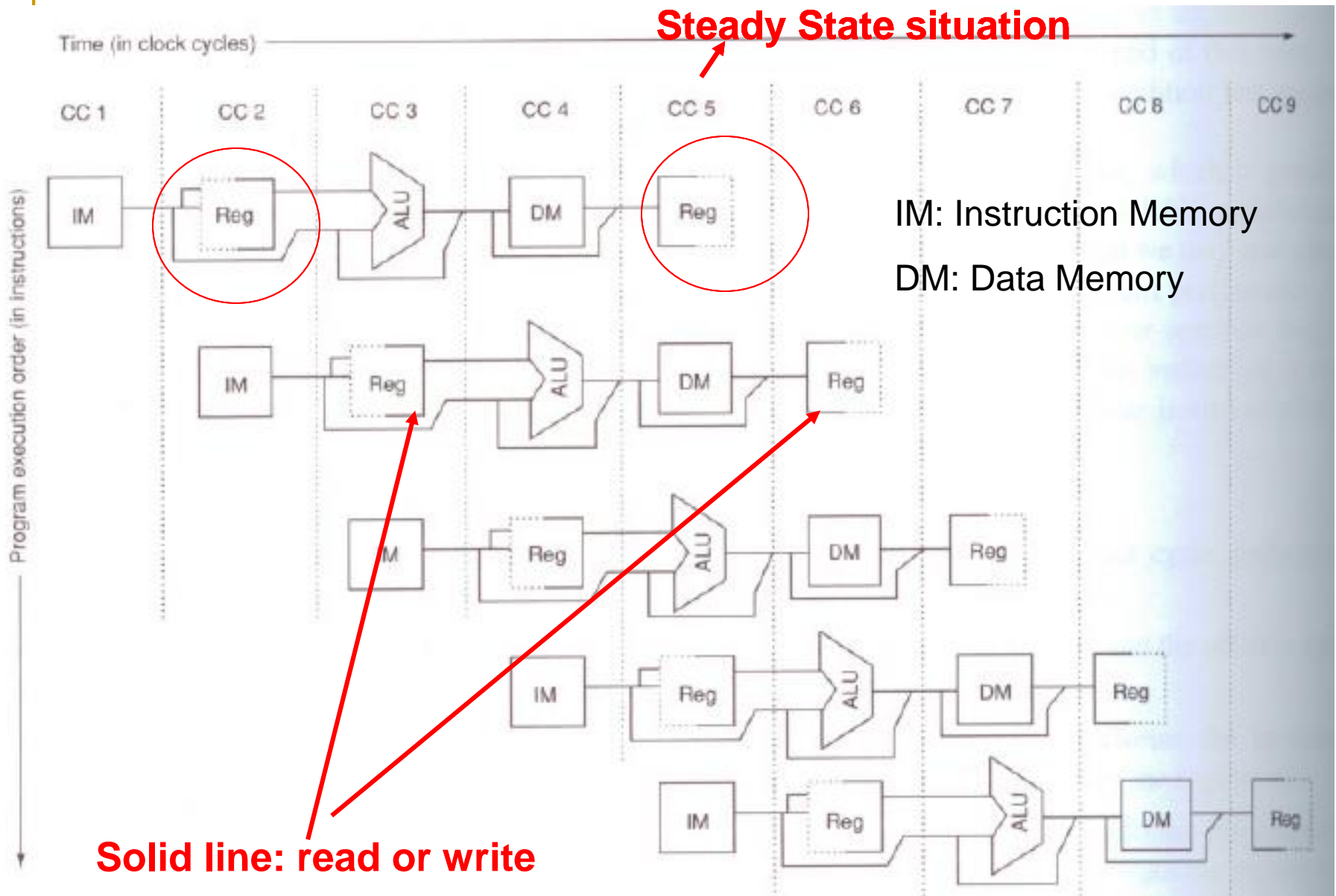
Jump and jump and link  
Trap and return from exception



Stage	Any instruction		
IF	$IF/ID.IR \leftarrow Mem[PC];$ $IF/ID.NPC, PC \leftarrow (if ((EX/MEM.opcode == branch) \& EX/MEM.cond) \{EX/MEM.ALUOutput\} else \{PC+4\});$		
ID	$ID/EX.A \leftarrow Regs[IF/ID.IR[rs]]; ID/EX.B \leftarrow Regs[IF/ID.IR[rt]]; ID/EX.NPC \leftarrow IF/ID.NPC; ID/EX.IR \leftarrow IF/ID.IR;$ $ID/EX.Imm \leftarrow sign-extend(IF/ID.IR[immediate field]);$		
	ALU instruction	Load or store instruction	Branch instruction
EX	$EX/MEM.IR \leftarrow ID/EX.IR;$ $EX/MEM.ALUOutput \leftarrow ID/EX.A \text{ func } ID/EX.B;$ or $EX/MEM.ALUOutput \leftarrow ID/EX.A \text{ op } ID/EX.Imm;$	$EX/MEM.IR \text{ to } ID/EX.IR$ $EX/MEM.ALUOutput \leftarrow ID/EX.A + ID/EX.Imm;$  $EX/MEM.B \leftarrow ID/EX.B;$	$EX/MEM.ALUOutput \leftarrow ID/EX.NPC + (ID/EX.Imm \ll 2);$  $EX/MEM.cond \leftarrow (ID/EX.A == 0);$
MEM	$MEM/WB.IR \leftarrow EX/MEM.IR;$ $MEM/WB.ALUOutput \leftarrow EX/MEM.ALUOutput;$	$MEM/WB.IR \leftarrow EX/MEM.IR;$ $MEM/WB.LMD \leftarrow Mem[EX/MEM.ALUOutput];$ or $Mem[EX/MEM.ALUOutput] \leftarrow EX/MEM.B;$	
WB	$Regs[MEM/WB.IR[rd]] \leftarrow MEM/WB.ALUOutput;$ or $Regs[MEM/WB.IR[rt]] \leftarrow MEM/WB.ALUOutput;$	For load only: $Regs[MEM/WB.IR[rt]] \leftarrow MEM/WB.LMD;$	

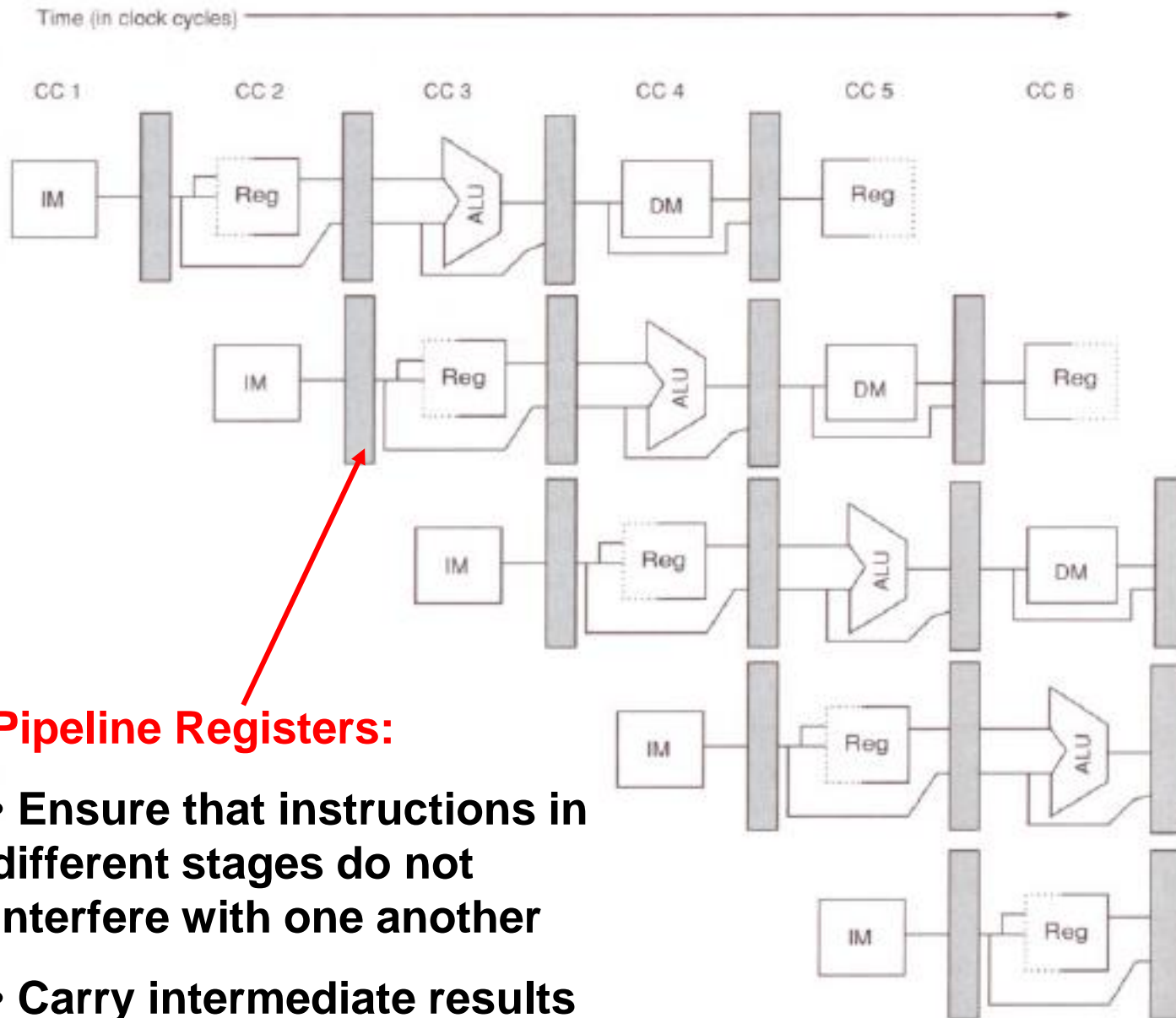
# Visualizing Pipelining





**Read: second half of the cycle**

**Write: first half of the cycle**



### Pipeline Registers:

- Ensure that instructions in different stages do not interfere with one another
- Carry intermediate results from one stage to another

# Basic Performance issues in pipelining:

## An example

### ■ Unpipelined processor:

- ❑ Clock cycle time: 1 ns
- ❑ ALU: 40%, uses 4 cycles
- ❑ Branches: 20%: uses 4 cycles
- ❑ Memory: 40%: uses 5 cycles

### ■ Pipelined processor:

- ❑ Clock cycle time: 1.2 ns
- ❑ Ignoring any latency impact (Ideal CPI on a pipelined machine = 1)

### ■ How much speedup?



## ■ Unpipelined:

- Average instruction exe time  
= clock cycle time\* average CPI  
= 1 ns \*(0.4\*4+0.2\*4+0.4\*5) = 4.4 ns

## ■ Pipelined:

- Average instruction exe time  
= clock cycle time\* ideal CPI=1.2 ns \* 1 = 1.2 ns

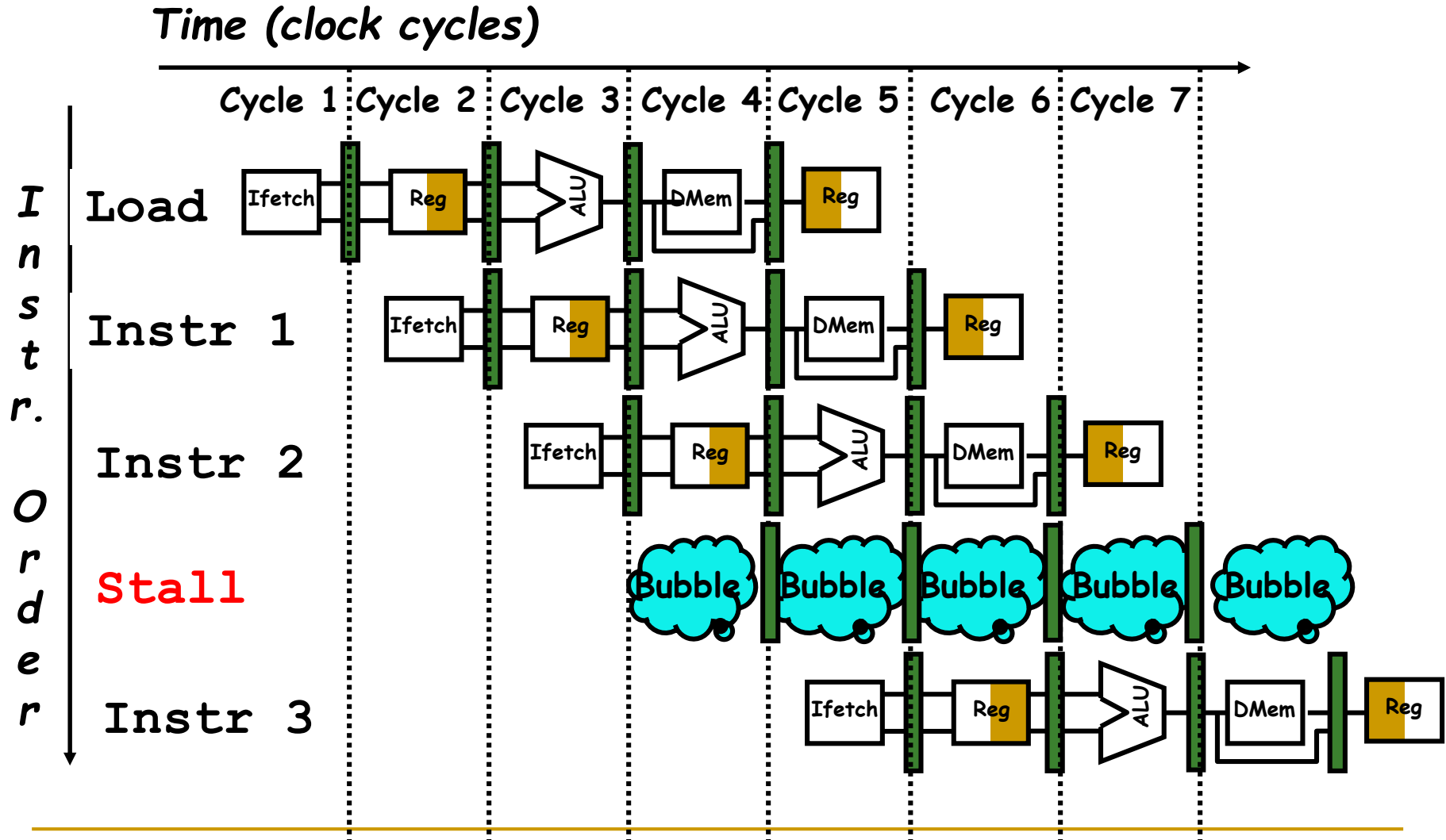
## ■ Speedup = 4.4/1.2 =3.7 times

# Limits to pipelining

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

- ❑ Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
- ❑ Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
- ❑ Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

# Hazards => Stalls



# Speed Up Equation for Pipelining

$$\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{CPI}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

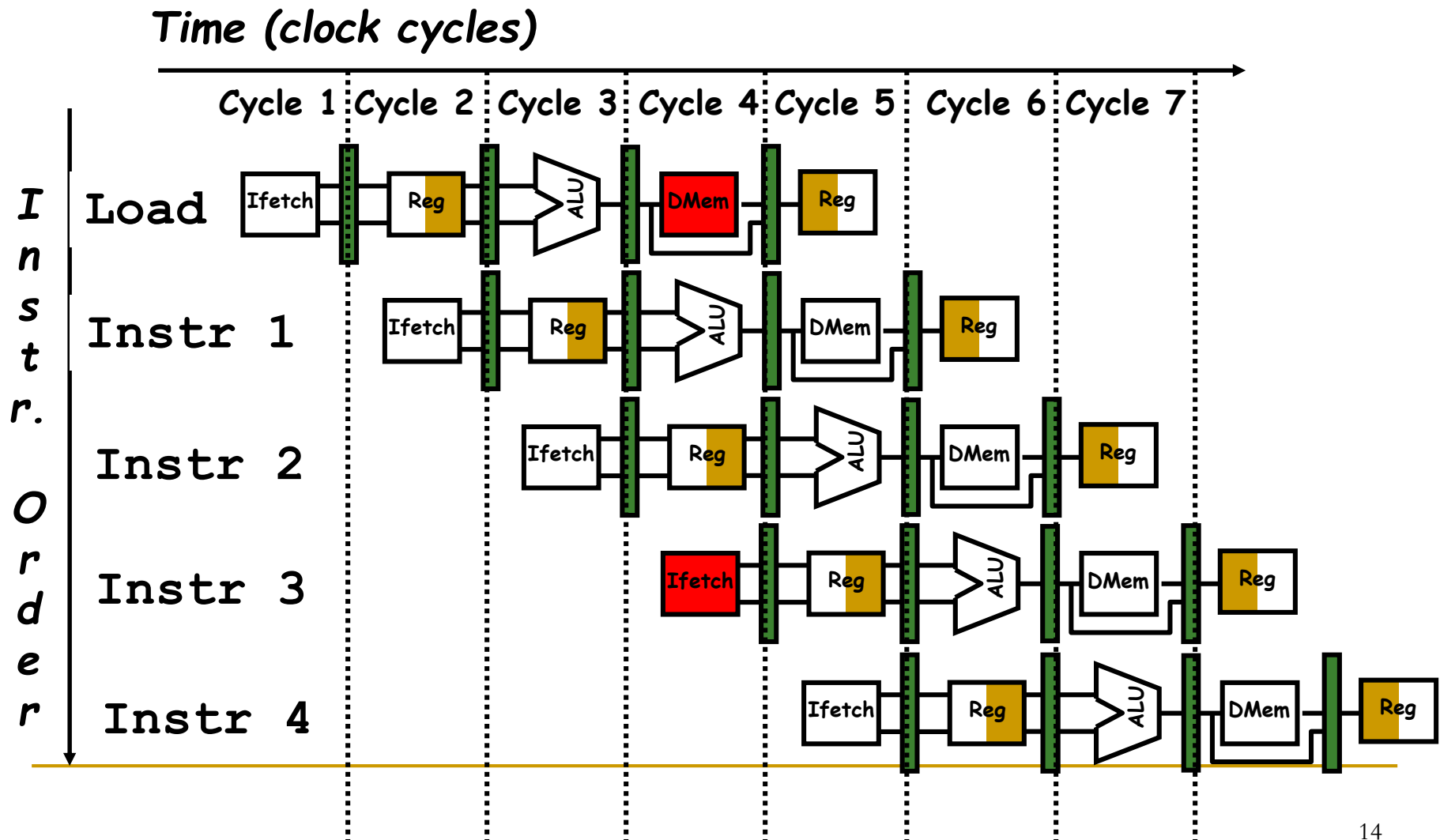
$$\text{Ideal CPI on a pipelined machine} = 1$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

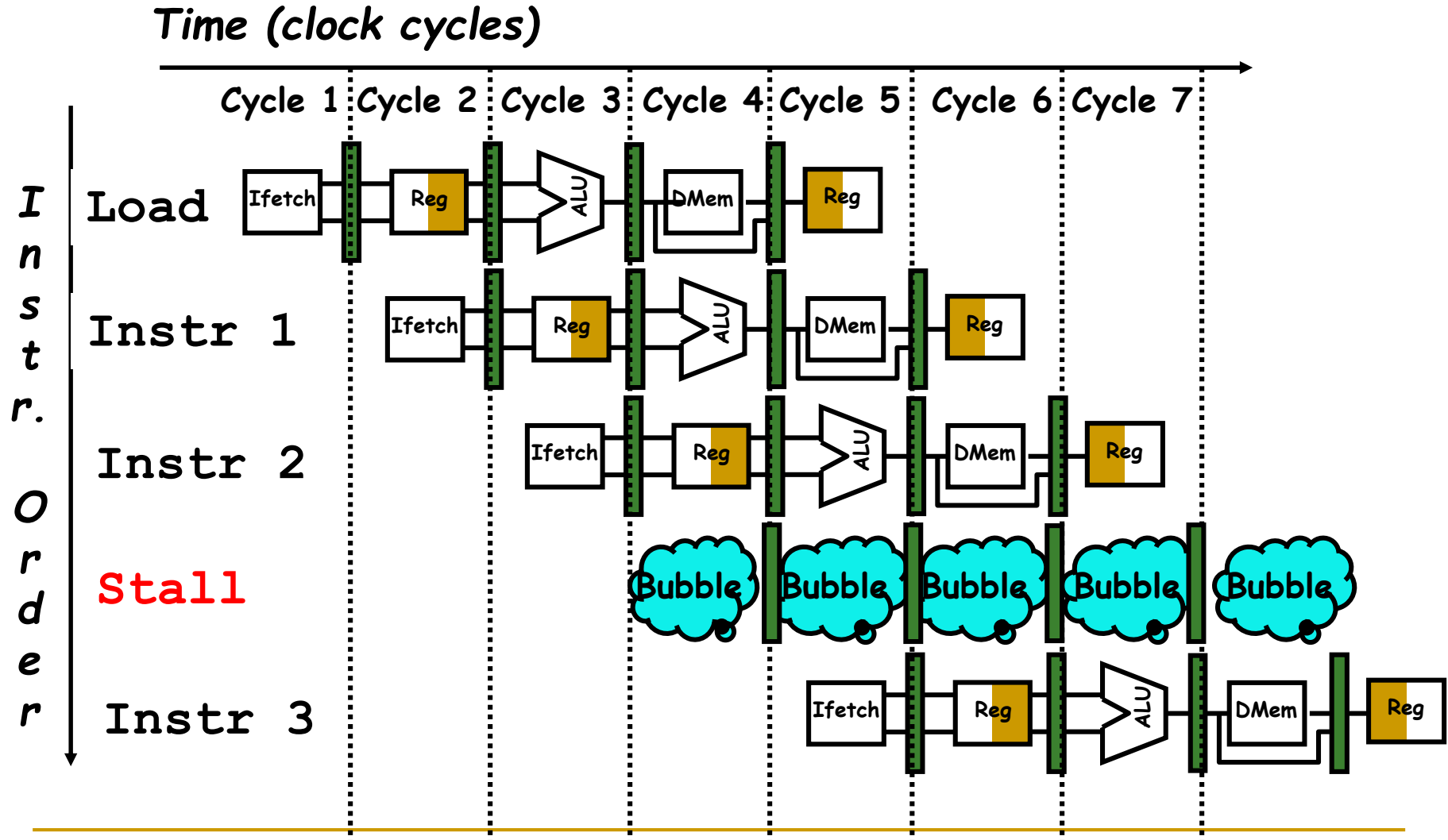
# Structural Hazard

- If some combination of instructions cannot be accommodated because of resource conflicts, the machine is said to have a structural hazard
- Common solution: stall the pipeline until no hazard in the pipeline
- A stall is commonly called a pipeline bubble

# One Memory Port/Structural Hazards



# One Memory Port/Structural Hazards



# Example

- Data reference: 40%
- Processor with structural hazard has a clock rate 1.05 times higher than the clock rate of the processor without the hazard
- Discarding any other performance losses
- Which processor is faster?





# Discussion on Structural Hazard

- If all other factors are equal, a processor without structural hazards will have a lower CPI
- Why would a designer allow structural hazard?
  - To reduce cost of the unit
  - Eg. Supporting both an instruction cache and data cache every cycle requires twice the total memory bandwidth
  - Eg. Functional unit to deal with Floating point instruction

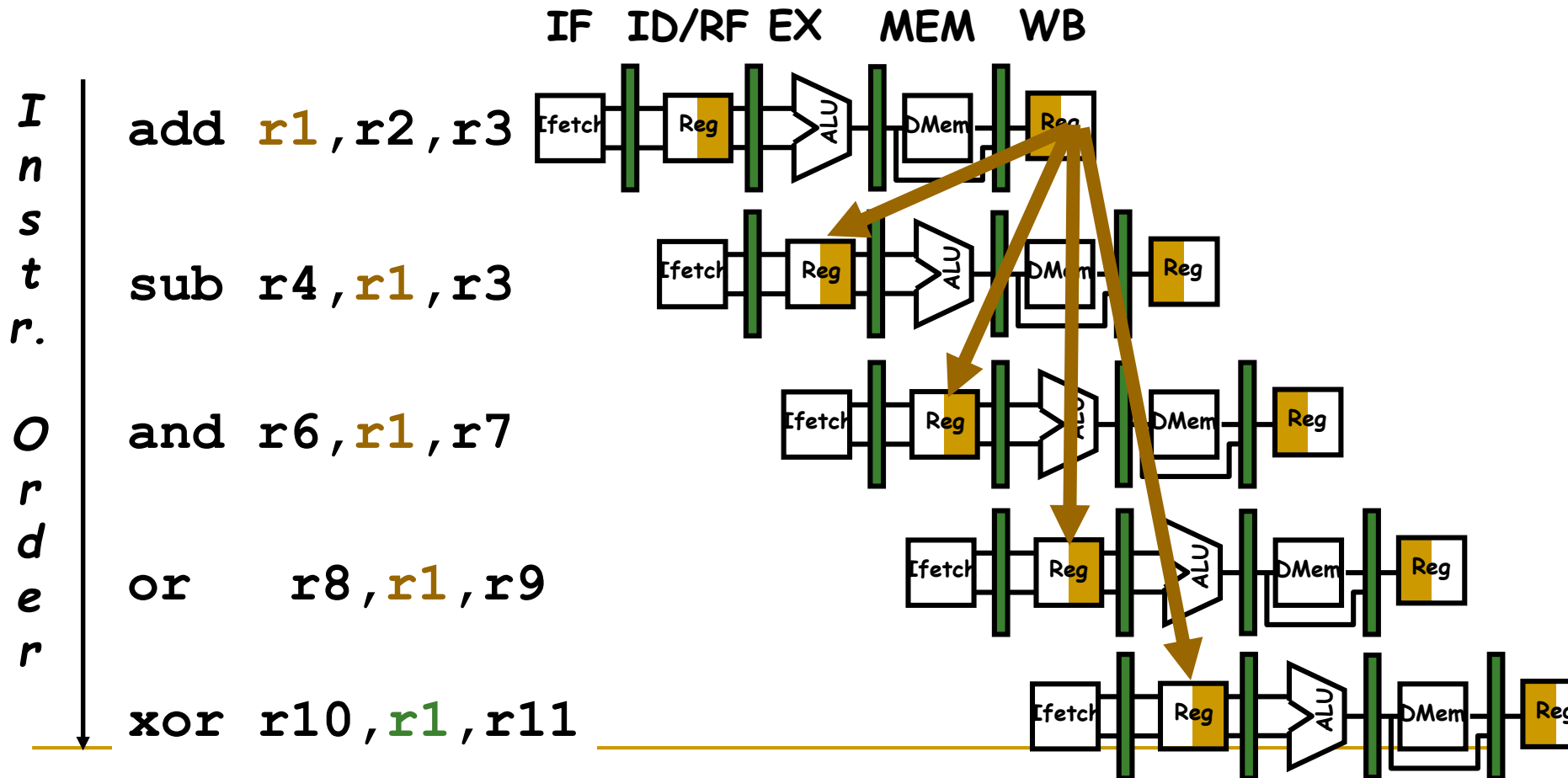
---

# Data Hazard

- Data hazards occur when the pipeline changes the order or read/write accesses to operands so that the order differs from the order seen sequentially executing instructions on an un-pipelined machine

# Data Hazard on R1

Time (clock cycles)



# Three Generic Data Hazards

- Read After Write (RAW)

Instr<sub>j</sub> tries to read operand before Instr<sub>i</sub> writes it

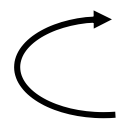
 I: add **r1**, r2, r3  
J: sub r4, **r1**, r3

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

# Three Generic Data Hazards

- Write After Read (WAR)

Instr<sub>j</sub> writes operand before Instr<sub>i</sub> reads it



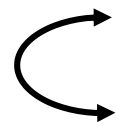
```
I: sub r4, r1, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

- Called an “**anti-dependence**” by compiler writers. This results from reuse of the name “**r1**”.
- Can’t happen in DLX 5 stage pipeline because:
  - ❑ All instructions take 5 stages, and
  - ❑ Reads are always in stage 2, and
  - ❑ Writes are always in stage 5

# Three Generic Data Hazards

- Write After Write (WAW)

Instr<sub>j</sub> writes operand before Instr<sub>i</sub> writes it.



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

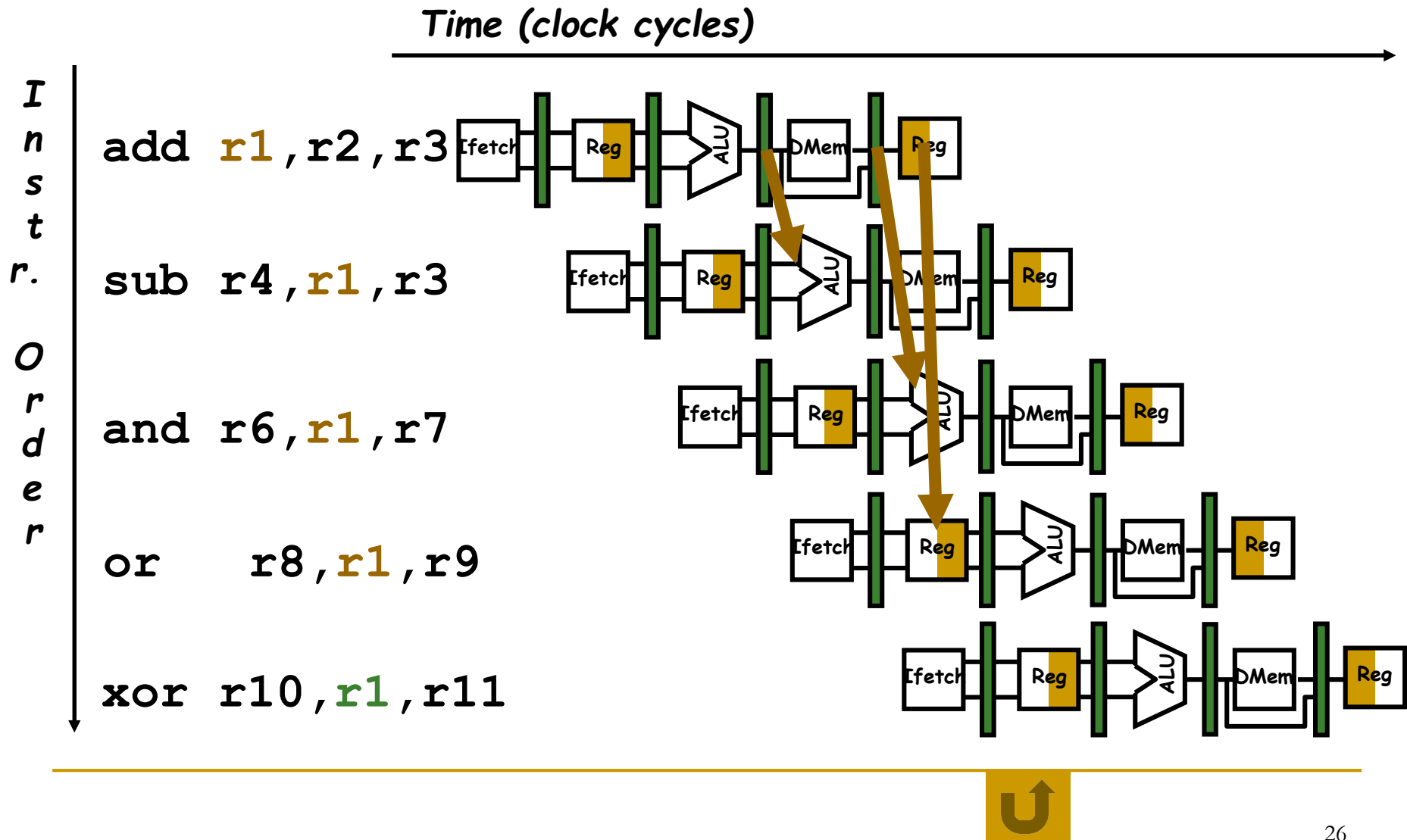
- Called an “**output dependence**” by compiler writers  
This also results from the reuse of name “**r1**”.
- Can’t happen in DLX 5 stage pipeline because:
  - ❑ All instructions take 5 stages, and
  - ❑ Writes are always in stage 5
- Will see WAR and WAW in more complicated pipelines

# Minimizing Data Hazard Stalls by Forwarding

- Data hazard problem can be solved with a hardware technique called **forwarding** (also called bypassing)
- A result is forwarded from the output of one unit to the input of another
- Not all potential data hazards can be handled by forwarding

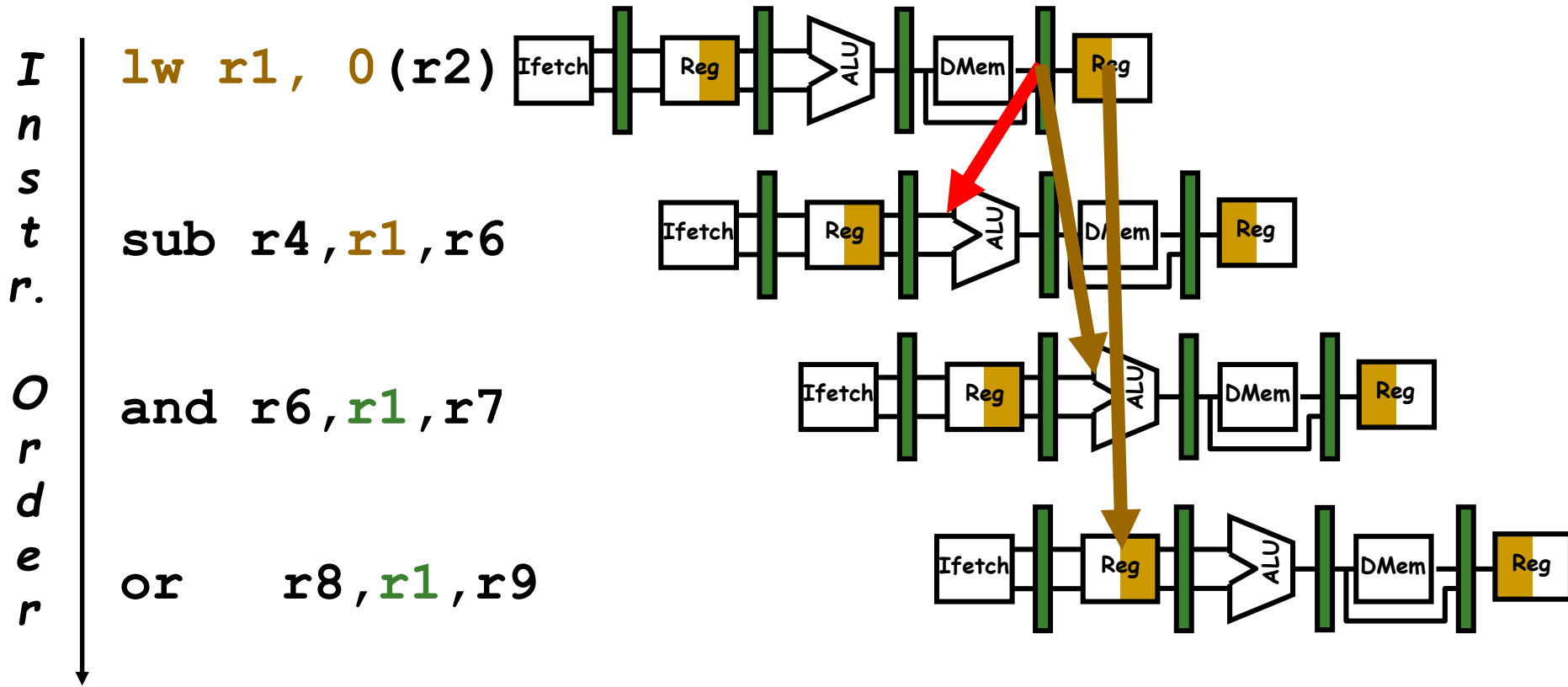


# Forwarding to Avoid Data Hazard

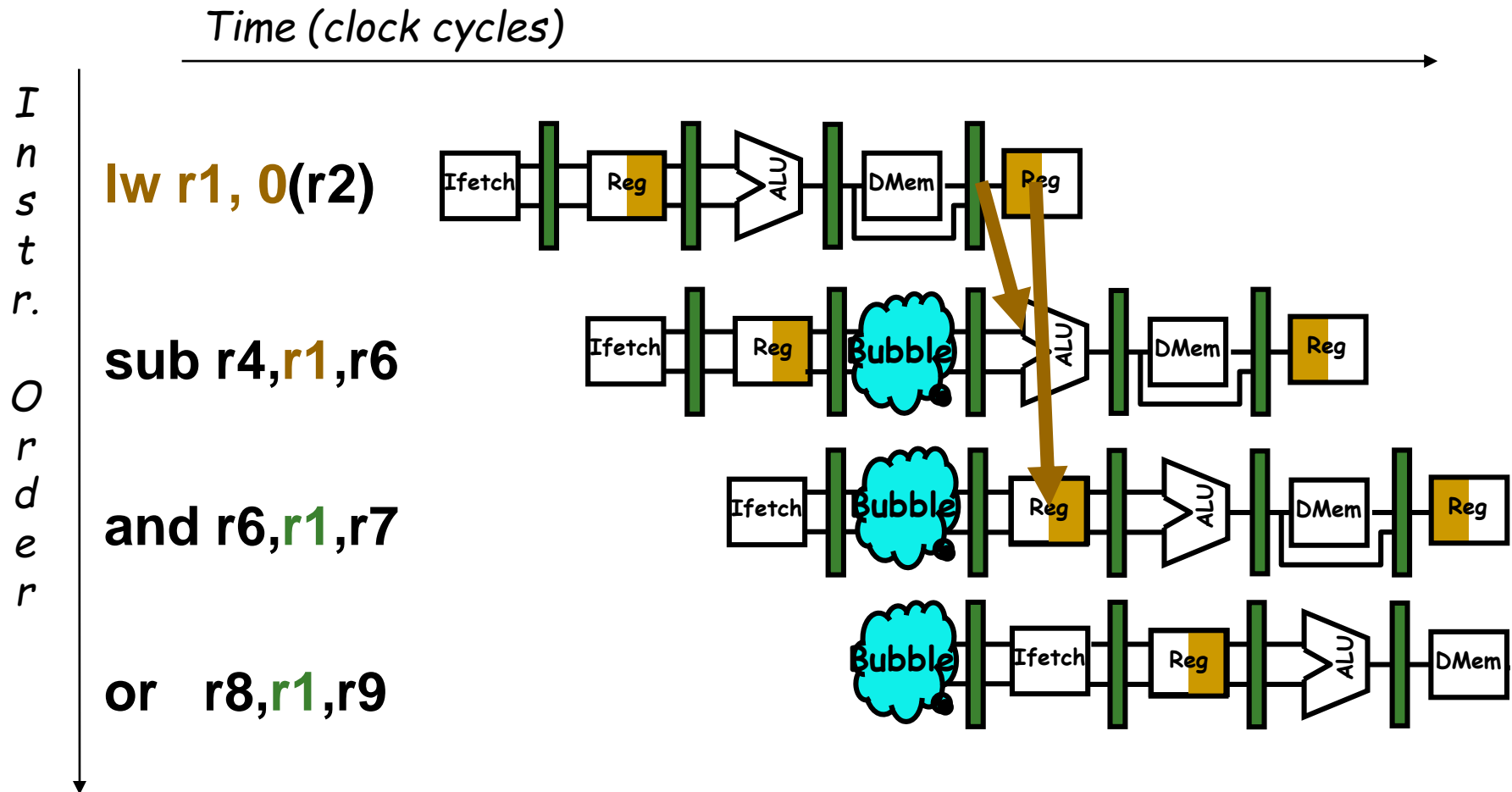


# Data Hazard Even with Forwarding

Time (clock cycles)



# Data Hazard Even with Forwarding



# Software Scheduling to Avoid Load Hazards

Try producing fast code for

$a = b + c;$

$d = e - f;$

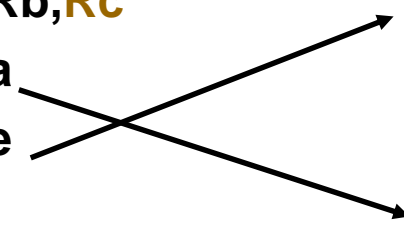
assuming  $a, b, c, d, e,$  and  $f$  in memory.

Slow code:

```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

Fast code:

```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
```



# Implementing the Control for DLX Pipeline

- If a data hazard exists, the instruction is stalled
- Detecting interlocks early in the pipeline reduces the hardware complexity
- Detect the hazard or forwarding at the beginning of a clock cycle that uses an operand

# Load Interlock can be detected by detection hardware

Situation	Example code sequence	Action
No dependence	LD <b>R1</b> ,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR    R9,R6,R7	No hazard possible because no dependence exists on R1 in the immediately following three instructions.
Dependence requiring stall	LD <b>R1</b> ,45(R2) DADD R5, <b>R1</b> ,R7 DSUB R8,R6,R7 OR    R9,R6,R7	Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX.
Dependence overcome by forwarding	LD <b>R1</b> ,45(R2) DADD R5,R6,R7 DSUB R8, <b>R1</b> ,R7 OR    R9,R6,R7	Comparators detect use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX.
Dependence with accesses in order	LD <b>R1</b> ,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR    R9, <b>R1</b> ,R7	No action 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.

# Logic to detect the Load Interlocks

Previous  
instruction

Reg-reg ALU has two source regs:  
Any of the two source regs

Opcode field of ID/EX (ID/EX.IR <sub>0..5</sub> )	Opcode field of IF/ID (IF/ID.IR <sub>0..5</sub> )	Matching operand fields
Load	Register-register ALU	ID/EX.IR[rt] == IF/ID.IR[rs]
Load	Register-register ALU	ID/EX.IR[rt] == IF/ID.IR[rt]
Load	Load, store, ALU immediate, or branch	ID/EX.IR[rt] == IF/ID.IR[rs]

Reg-reg ALU (R type)



Load, store, ALU immediate, branch (I type)



# Forwarding Conditions 1/2

Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$EX/MEM.IR[rd] == ID/EX.IR[rs]$
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	$EX/MEM.IR[rd] == ID/EX.IR[rt]$
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$MEM/WB.IR[rd] == ID/EX.IR[rs]$
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	$MEM/WB.IR[rd] == ID/EX.IR[rt]$



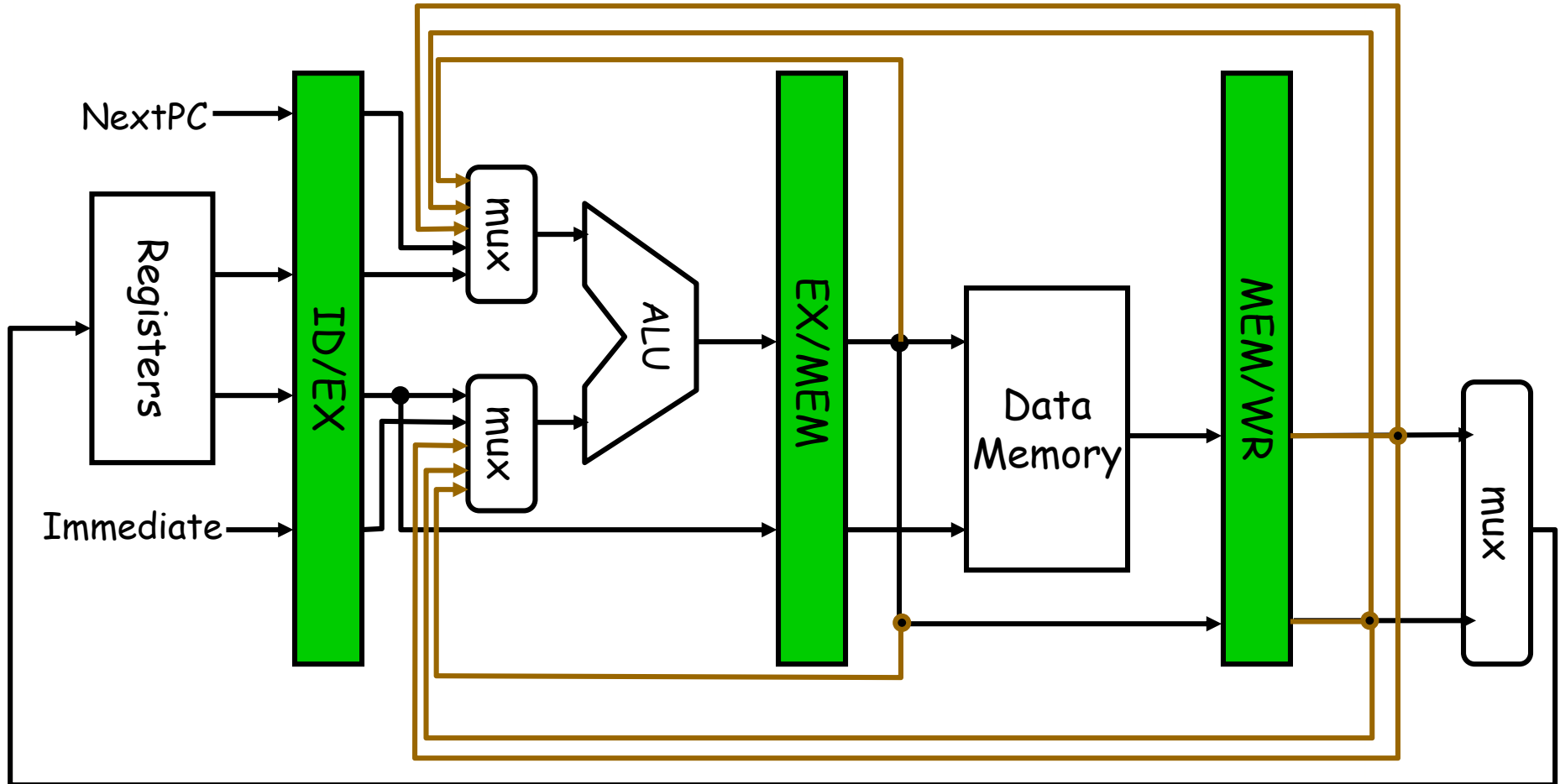


# Forwarding Conditions 2/2



Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$EX/MEM.IR[rt] == ID/EX.IR[rs]$
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	$EX/MEM.IR[rt] == ID/EX.IR[rt]$
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$MEM/WB.IR[rt] == ID/EX.IR[rs]$
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	$MEM/WB.IR[rt] == ID/EX.IR[rt]$
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$MEM/WB.IR[rt] == ID/EX.IR[rs]$
MEM/WB	Load	ID/EX	Register-register ALU	Bottom ALU input	$MEM/WB.IR[rt] == ID/EX.IR[rt]$

# HW Change for Forwarding



# Control Hazards

- Control hazards can cause a greater performance loss for the DLX pipeline than do data hazards
- When a branch is executed, it may or may not change the PC
  - Taken
  - Untaken

# Control Hazard on Branches

## Three Stage Stall

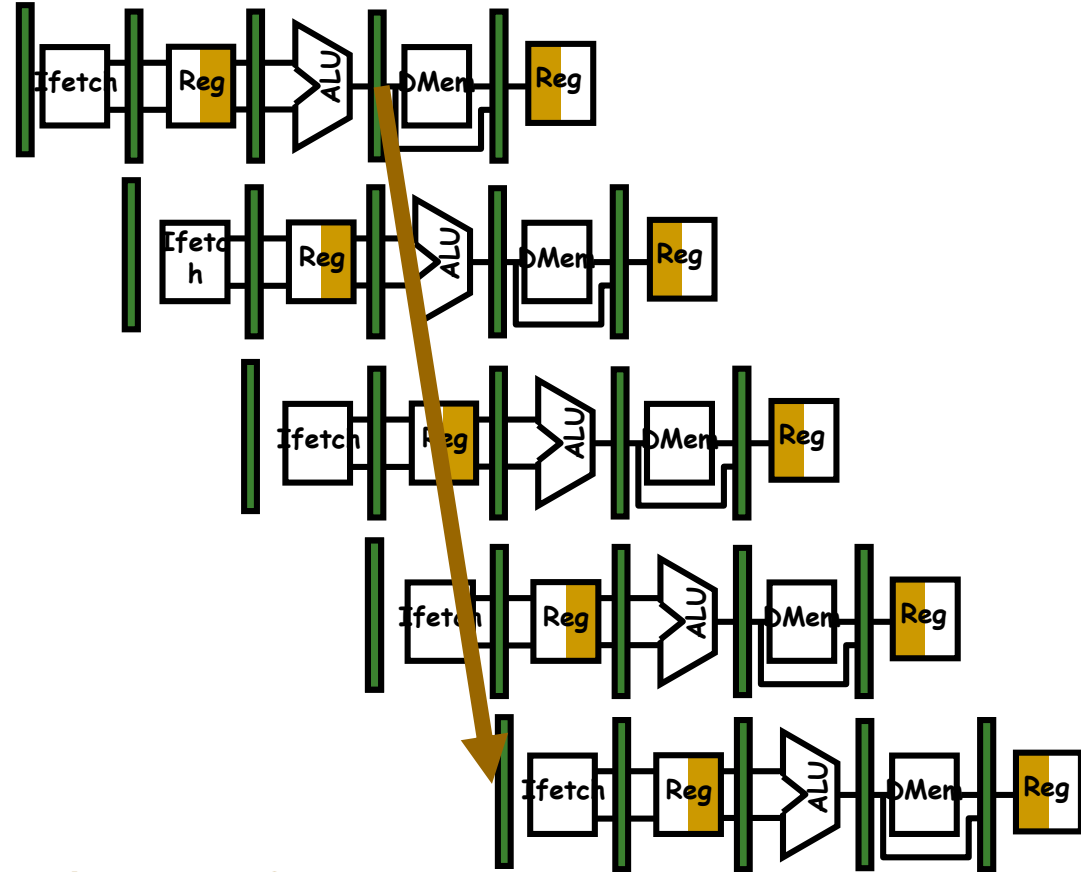
10: beq r1, r3, 36

14: and r2, r3, r5

18: or r6, r1, r7

22: add r8, r1, r9

36: xor r10, r1, r11



What do you do with the 3 instructions in between?

How do you do it?

Where is the "commit"?

# Branch Stall Impact

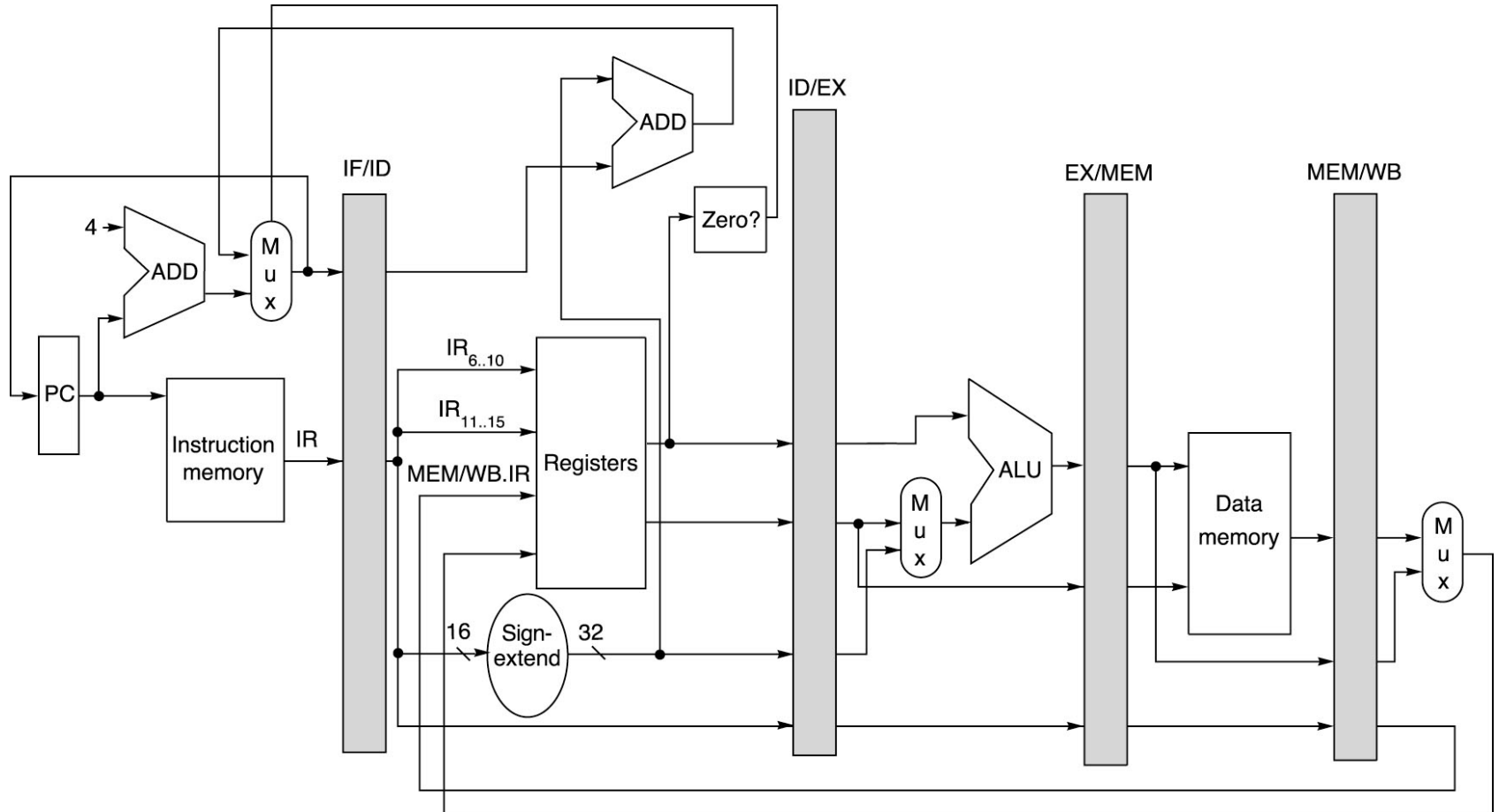
- If CPI = 1, 30% branch,  
Stall 3 cycles => new CPI = 1.9

Op	Freq	Cycles
Other	70%	1
Branch	30%	4

$$0.7 \cdot 1 + 0.3 \cdot 4 = 0.7 + 1.2 = 1.9$$

- Two part solution:
  - ❑ Determine branch taken or not sooner, AND
  - ❑ Compute taken branch address earlier
- DLX branch tests if register = 0 or  $\neq$  0
- DLX Solution:
  - ❑ Move **Zero test** to **ID** stage
  - ❑ Adder to calculate **new PC** in **ID** stage
  - ❑ 1 clock cycle penalty for branch versus 3

# Revised pipeline structure



# Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- ❑ Execute successor instructions in sequence
- ❑ “Squash” instructions in pipeline if branch actually taken
- ❑ Advantage of late pipeline state update
- ❑ 47% DLX branches not taken on average
- ❑ PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- ❑ 53% DLX branches taken on average
- ❑ But haven't calculated branch target address in DLX
  - DLX still incurs 1 cycle branch penalty
  - Other machines: branch target known before outcome

# Four Branch Hazard Alternatives

## #4: Delayed Branch

- Define branch to take place **AFTER** a following instruction

branch instruction

sequential successor<sub>1</sub>

sequential successor<sub>2</sub>

.....

sequential successor<sub>n</sub>

branch target if taken



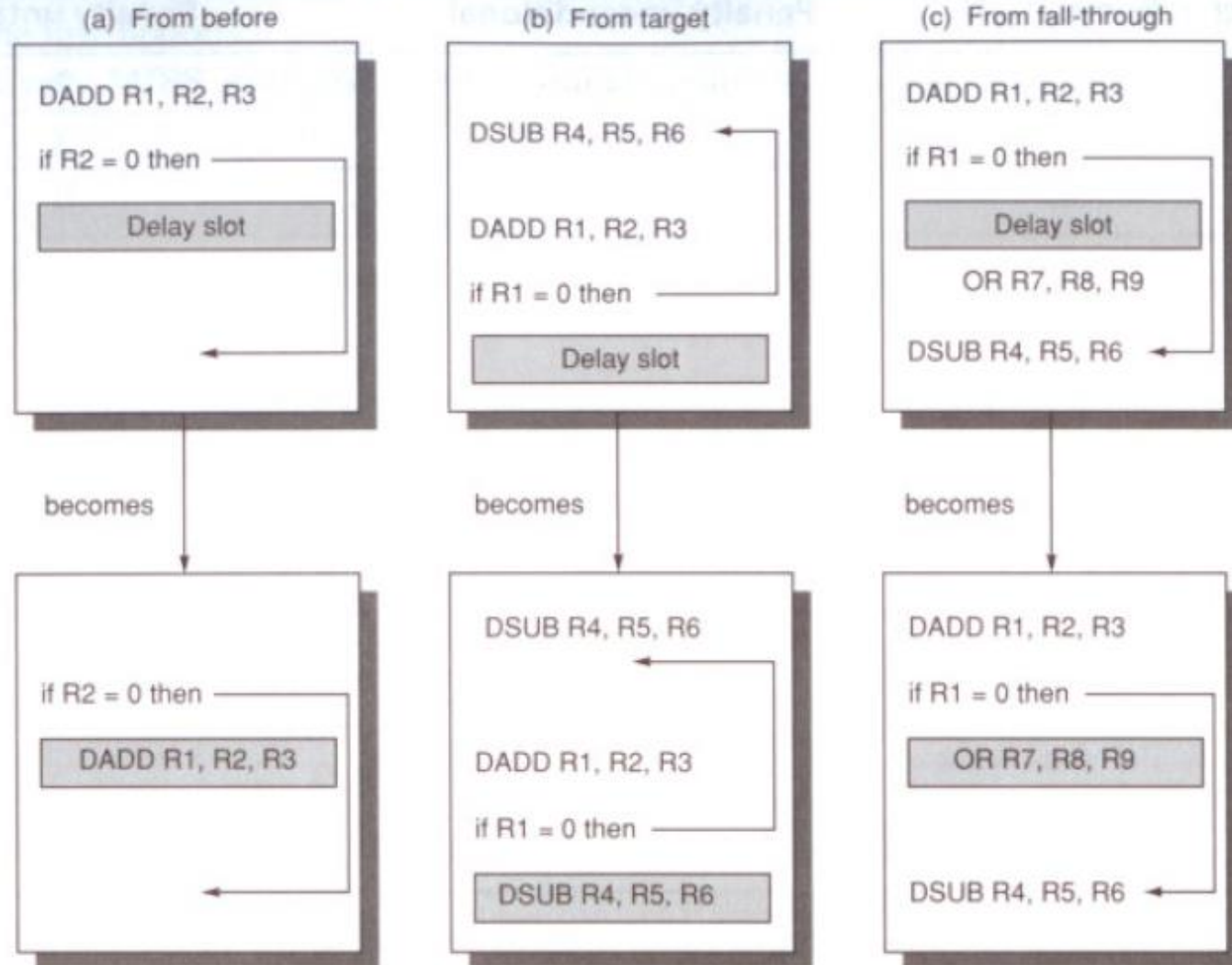
**Branch delay of length  $n$**

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline

- DLX uses this



# Delayed Branch



# Delayed-branch scheduling schemes and their requirements

Scheduling Strategy	Requirements	Improves performance when
From Before	Branch must not depend on the rescheduled instruction	always
From target	Must be OK to execute rescheduled instructions if branch is not taken.	When branch is taken
From fall through	Must be OK to execute instructions if branch is taken	When branch is not taken

# Delayed Branch

- Where to get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: only valuable when branch taken
  - From fall through: only valuable when branch not taken
  - **Canceling branches** allow more slots to be filled
- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar): cannot easily hide the long delays
  - **More powerful prediction scheme is desired for deep pipelines**

# Evaluating Branch Alternatives

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

<i>Scheduling scheme</i>	<i>Branch penalty</i>	<i>CPI</i>	<i>speedup v. unpipelined</i>	<i>speedup v. stall</i>
Stall pipeline	3	1.42	3.5	1.0
Predict taken	1	1.14	4.4	1.26
Predict not taken	1	1.09	4.5	1.29
Delayed branch	0.5	1.07	4.6	1.31

---

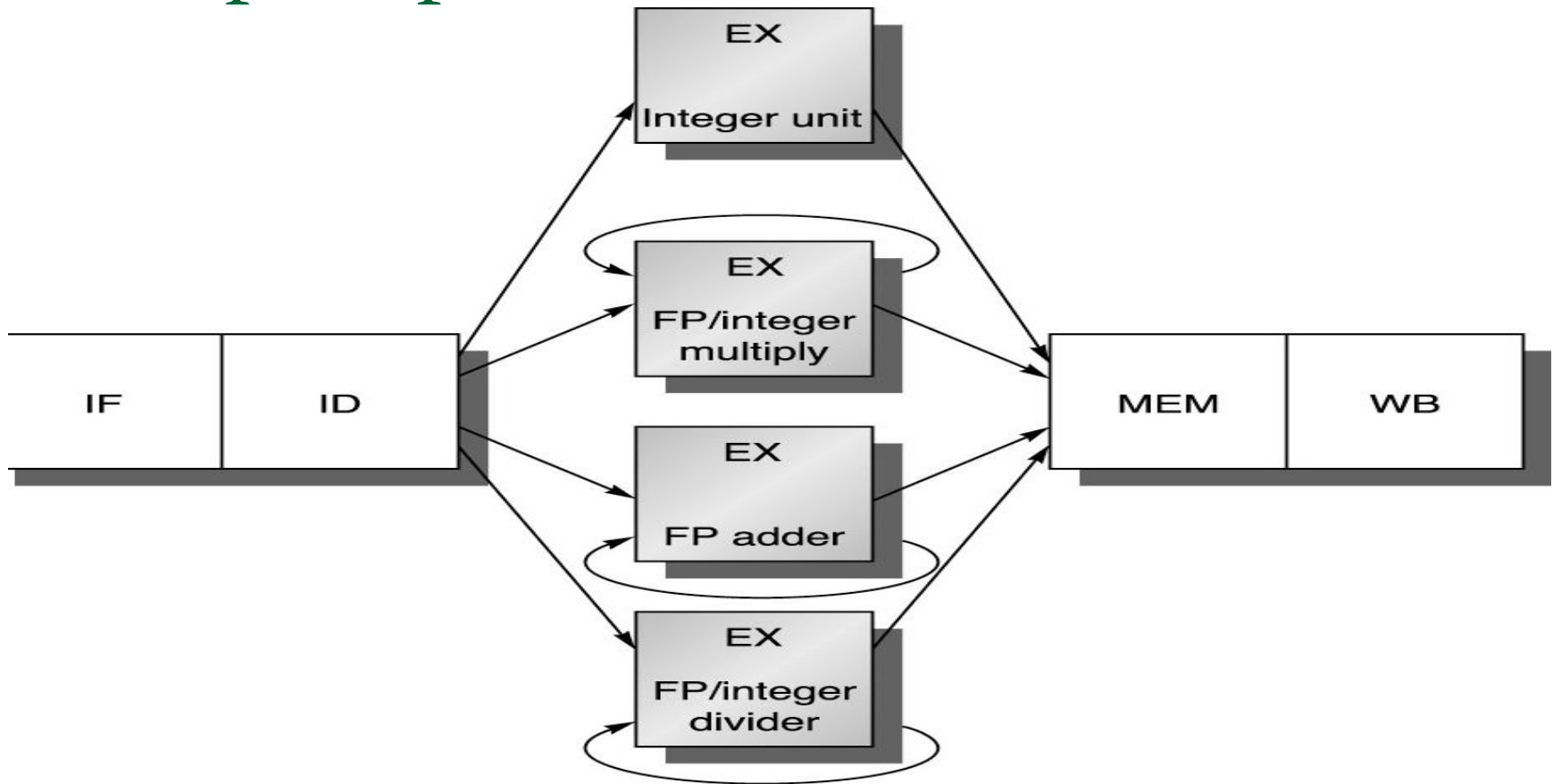
# What makes pipelining hard to implement

- Dealing with exceptions
- Stopping and restarting execution
- Instruction set complication

# Exceptions of DLX pipeline

Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory- protection violation
ID	Undefined or illegal op-code
EX	Arithmetic Exception
MEM	Page Fault on data fetch; misaligned memory access; memory- protection violation
WB	None

# Extending the DLX pipeline to handle multiple operations



# Latencies and initiation intervals for function units

# of intervening cycles between an instruction that produces a result and an instruction that uses the result.

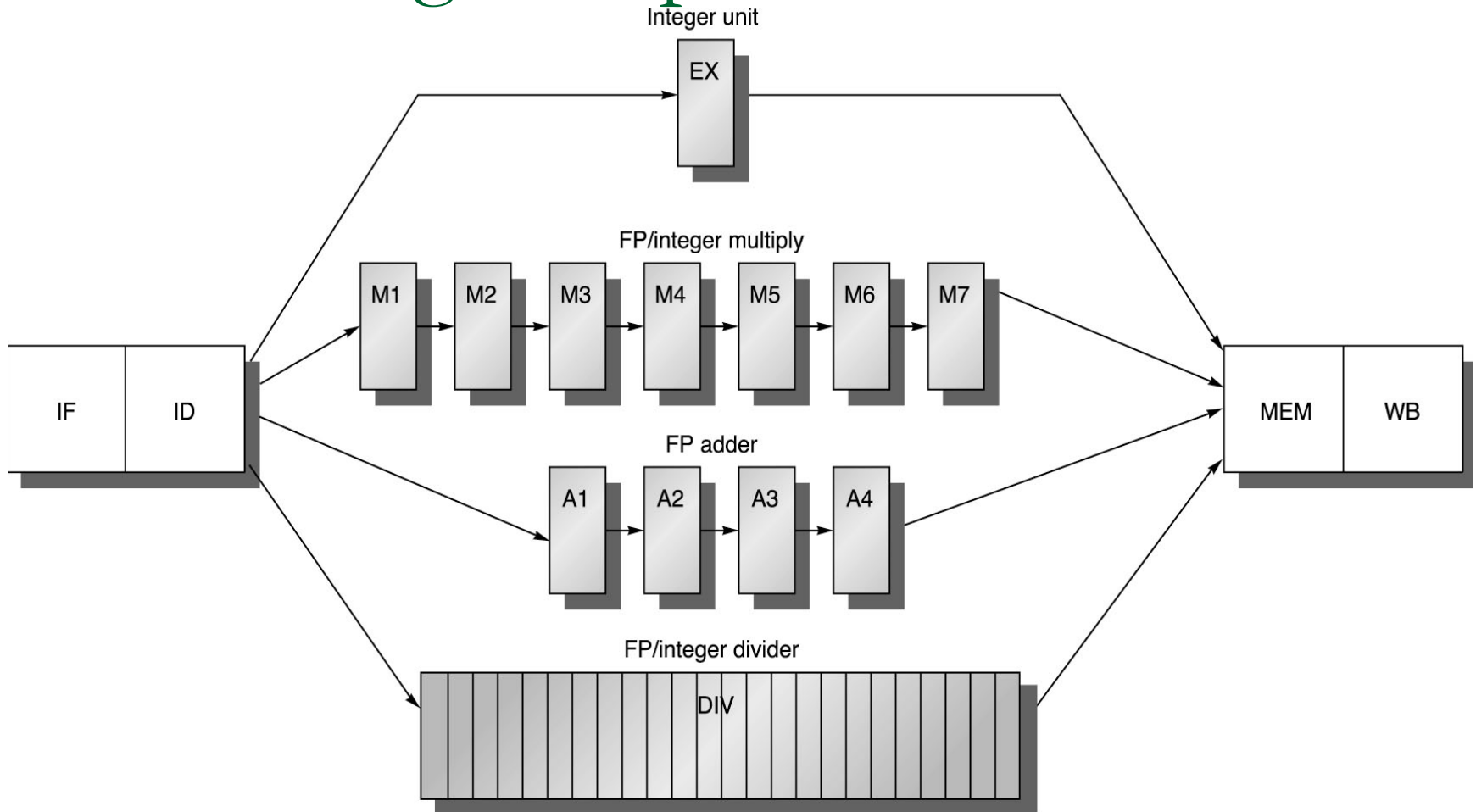
Functional unit	Latency (clock cycle)	Initial interval (Repeat interval)
Integer ALU	0	1
Data memory	1	1
FP add	3	1
FP multiply	6	1
FP divide	24	25

# of clock cycles that must elapse between issuing two operations of a given type

- As we can observe from the table, EX stage needs not only one cycle



# A pipeline that supports multiple outstanding FP operations



# A pipeline that supports multiple outstanding FP operations

- The divide unit is not fully pipelined. Structural hazards can occur.
- The instructions have varying running times. The number of register writes required in a cycle can be larger than 1.
- Instructions can complete in a different order than they were issued.
- Stalls for RAW hazards are more frequent.

# Multiple Write Back Simultaneously

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

# RAW hazards

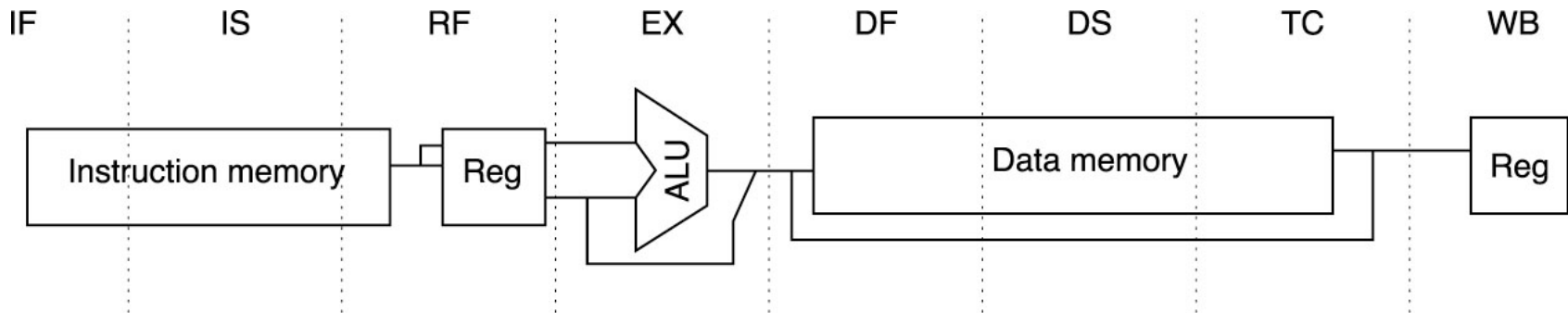
- The longer pipeline raises the frequency of stalls.
- In this example, each instruction is dependent on the previous one.

Instruction	Clock cycle number																
	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		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D F2,F0,F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	WB
S.D F2,0(R2)					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM

# The MIPS R4000 Pipeline

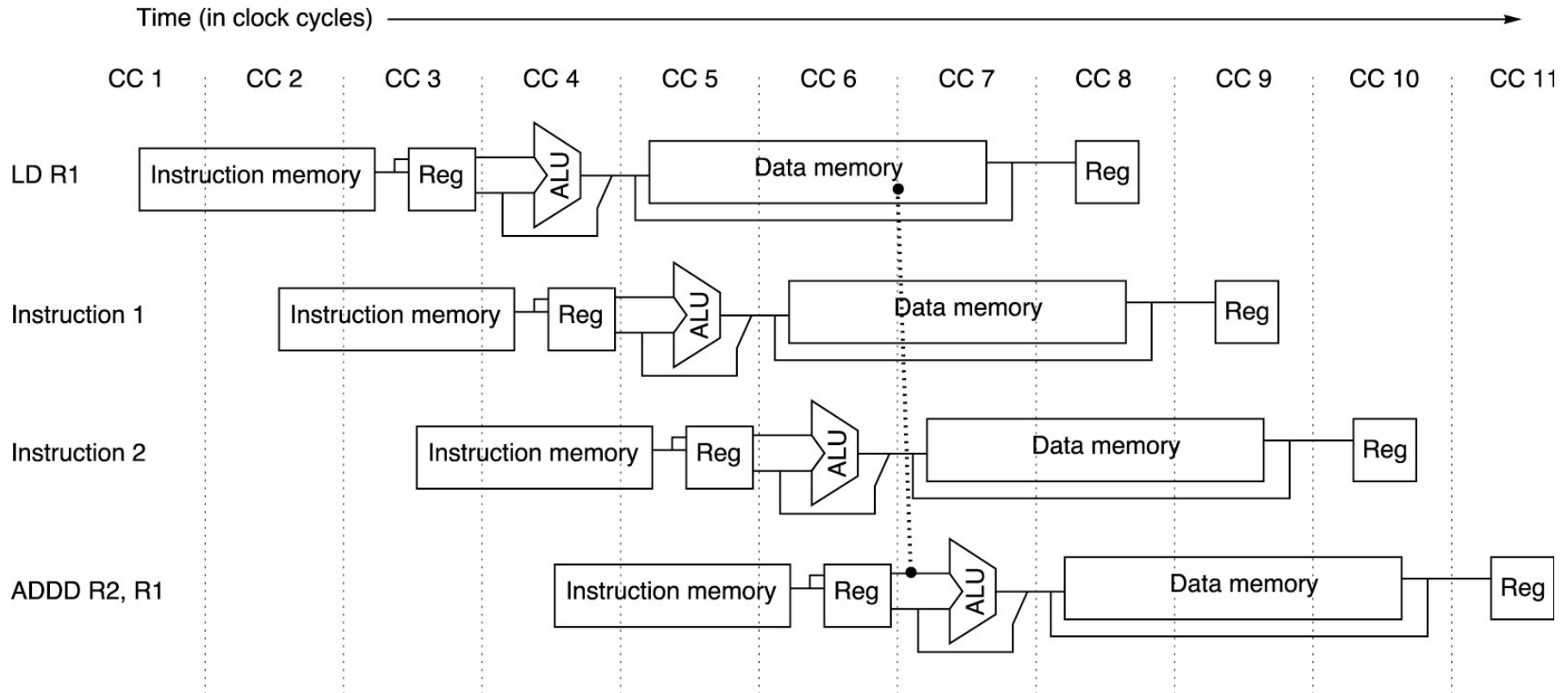
- The eight pipeline stages of MIPS R4000
  - IF – First half of instruction fetch
  - IS – Second half of instruction fetch
  - RF – Instruction decode and register fetch
  - EX – Execution
  - DF – First half of data cache access
  - DS – Second half of data cache access
  - TC – Tag check (check cache hit or miss)
  - WB – Write back

# The eight stage pipeline structure of MIPS R4000

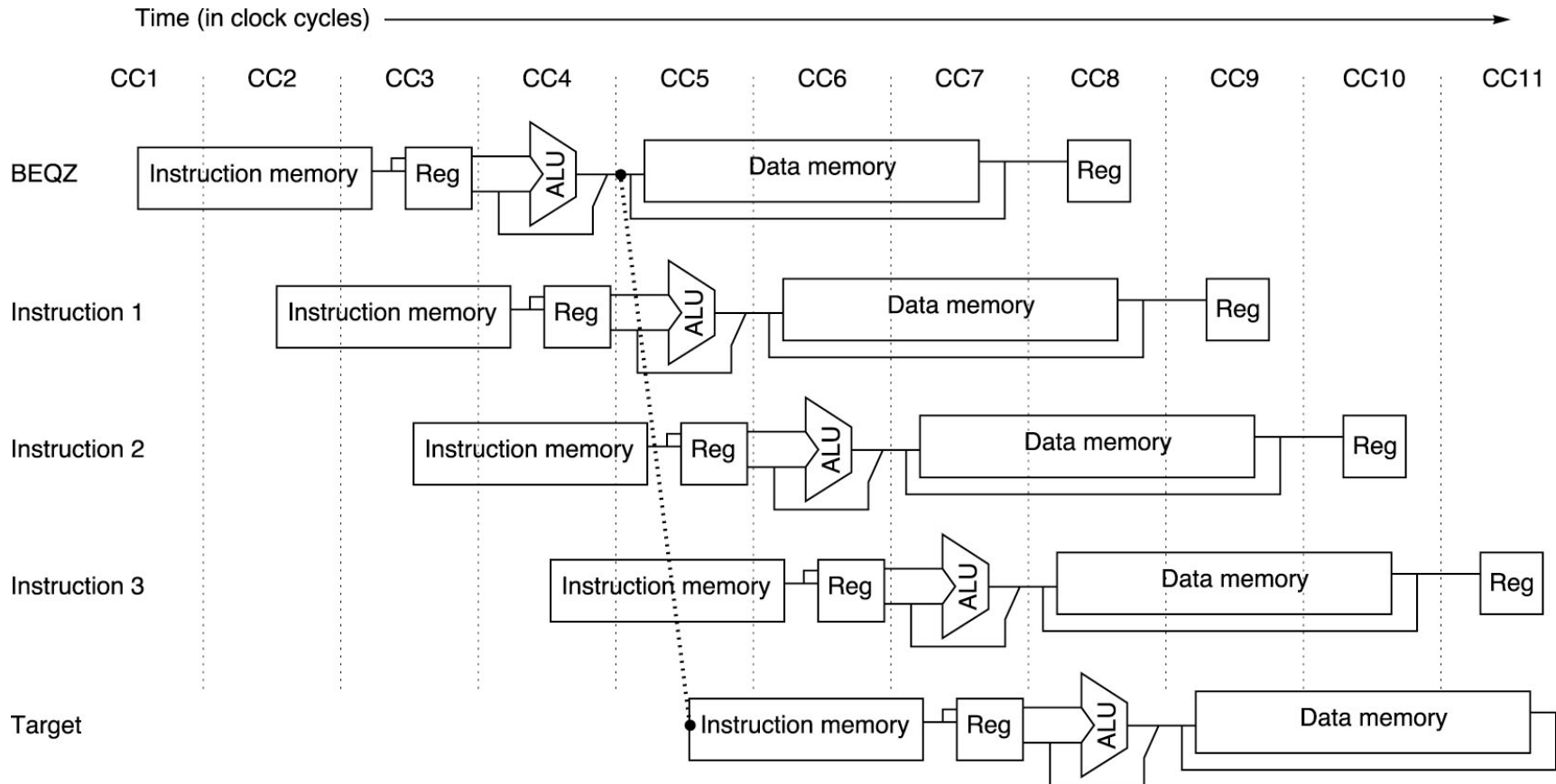


© 2003 Elsevier Science (USA). All rights reserved.

# A two-cycle load delay of R4000 integer pipeline



# The basic branch delay of R4000 pipeline





# Summary: Longer latency pipelines

- The divide unit is not fully pipelined, structural hazards can occur.
- Varying running time
  - Number of register writes required in one cycle can be  $> 1$
  - WAW hazards are possible (not reach WB in order)
  - out of order completion causing problems with exception
- Stalls for RAW will be more frequent
- WAR is not possible in this case since the register reads always occurs in ID