PIPELINING

B649
Parallel Architectures and Programming

Announcements

- Blogs
- Presentation topics and teams
  - ⋆ already taken:
    - ✳ ISA (App. J)
    - ✳ Hardware and software for VLIW and EPIC (App. G)
    - ✳ Large Scale Multiprocessors and Scientific Applications (App. H)
- Heads-up
  - ⋆ Blog question
  - ⋆ Assignment 1

Why Pipelining?

- Instruction-Level Parallelism (ILP)
- Reducing Cycles Per Instruction (CPI)
  - ⋆ if instructions may take multiple cycles

u = Time per instruction on unpipelined machine
n = Number of pipelined stages
time per pipelined instruction = u / n

- Decreasing the clock cycle time
  - ⋆ if each instruction takes one (long) cycle
- Invisible to the programmer

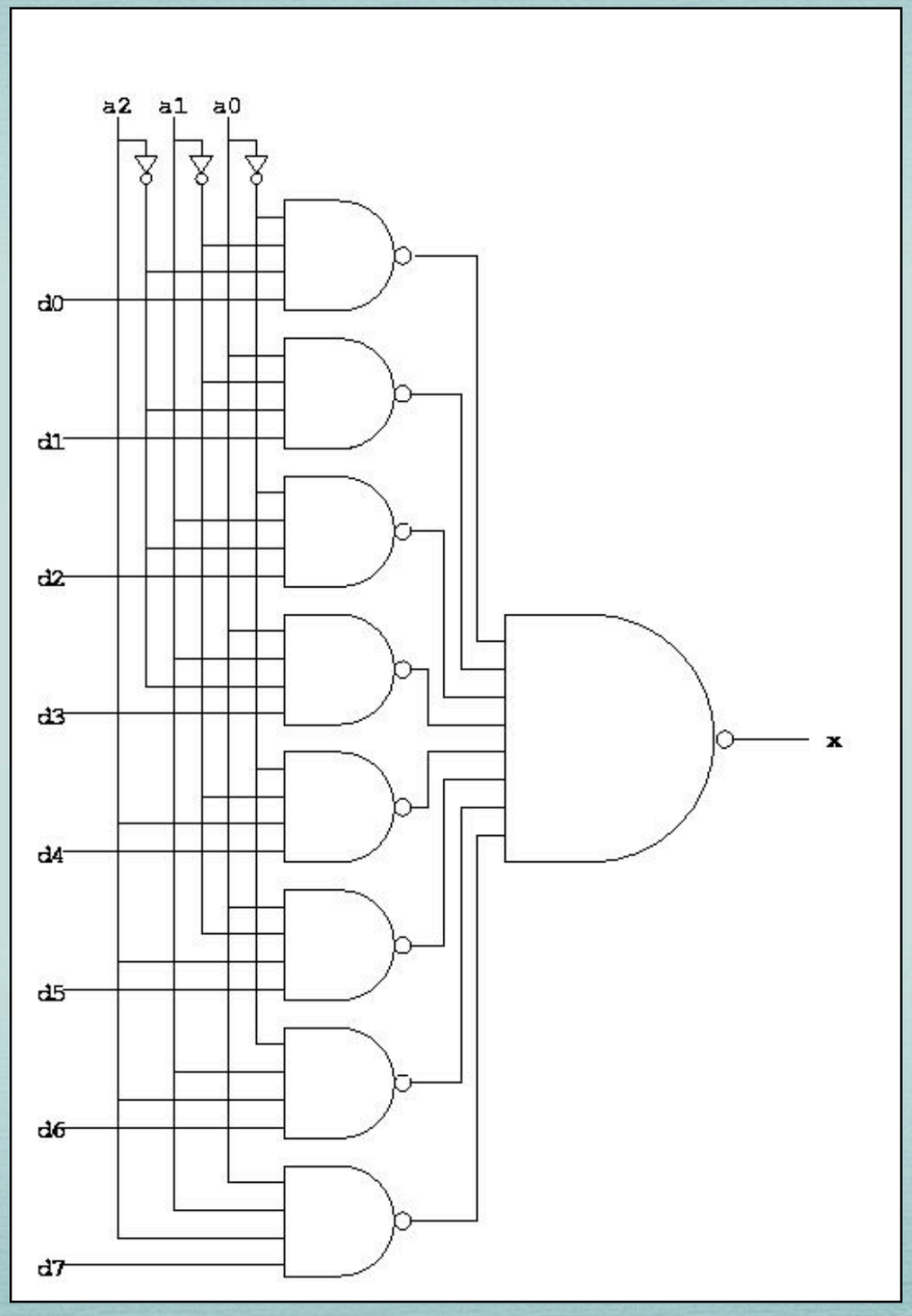
Basics of a RISC Instruction Set

- All operations on data registers
- Only load and store access memory
- All instructions are of one (fixed) size
- MIPS64 (64-bit) instructions for example

Instruction Set Overview

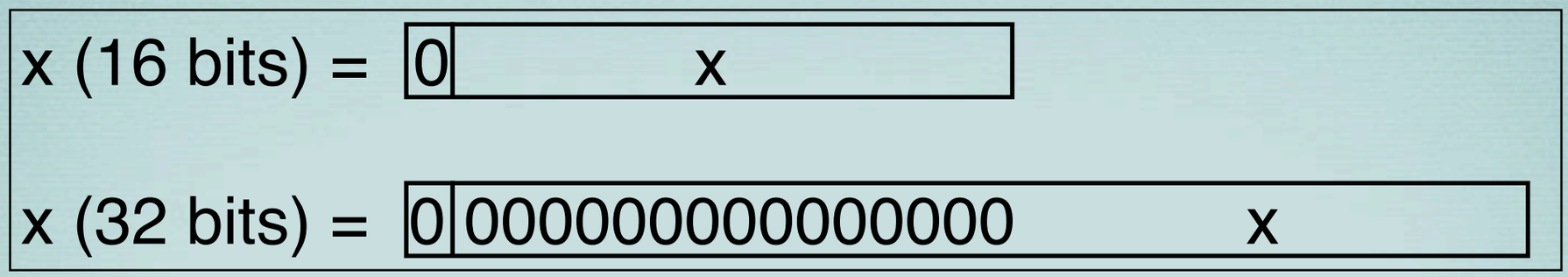
- ALU instructions
  - ⋆ R1 ← R2 op R3
    - ✳ R1, R2, R3: registers
  - ⋆ R1 ← R2 op I
    - ✳ I: signed extended 16-bit immediate
- Load and store instructions
  - ⋆ LD R1:O, R2
    - ✳ R1, R2: registers, O: 16-bit signed extended 16-bit immediate
- Branch and jump instructions
  - ⋆ comparison between two registers, or register and zero
  - ⋆ no unconditional jump

Digression: Recall Multiplexer

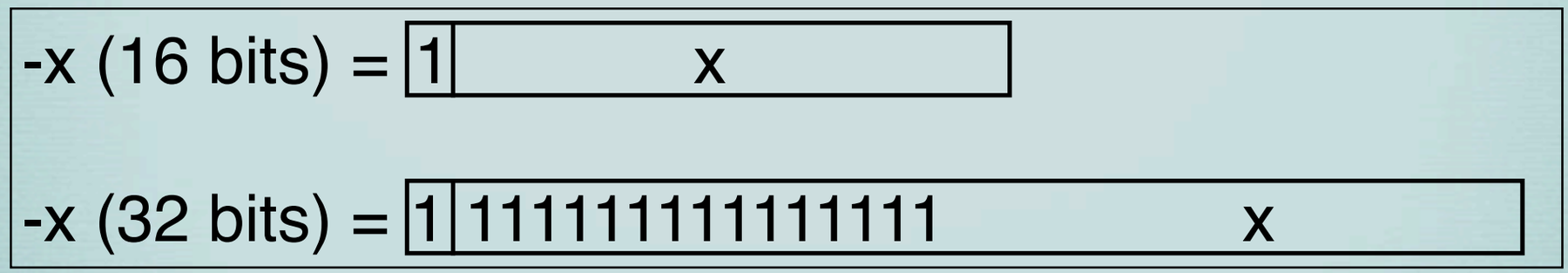


Digression: Sign Extension

- Positive Number: extend with zeroes

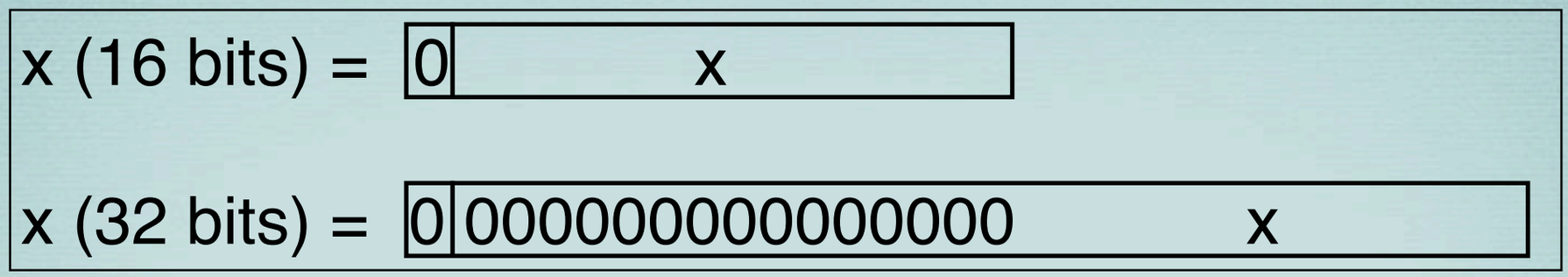


- Negative number: extend with ones

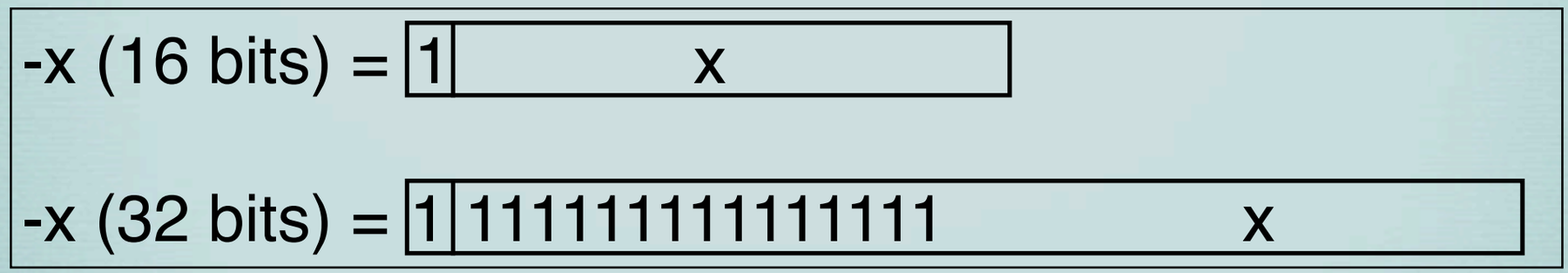


Digression: Sign Extension

- Positive Number: extend with zeroes



- Negative number: extend with ones



$$
\begin{aligned}
\text{-x (16 bits)} &= (2^{16} - x - 1) \\
\text{-x (extended)} &= (2^{16} - x - 1) + 2^{16}(2^{16} - 1) \\
&= (2^{32} - x - 1) \\
&= \text{-x (32 bits)}
\end{aligned}
$$

Simple Implementation

- IF: Instruction fetch cycle
- ID: Instruction decode / register fetch cycle
- EX: Execute / effective address cycle
- MEM: Memory access cycle
- WB: Write-back cycle

Simple Implementation

- IF: Instruction fetch cycle
  - ⋆ fetch current instruction from PC, add 4 to PC
- ID: Instruction decode / register fetch cycle
- EX: Execute / effective address cycle
- MEM: Memory access cycle
- WB: Write-back cycle

Simple Implementation

- IF: Instruction fetch cycle
- ID: Instruction decode / register fetch cycle
  - ⋆ decode instruction, read registers (*fixed field decoding*)
  - ⋆ do equality test on registers for possible branch
  - ⋆ sign extend offset field, in case it is needed
  - ⋆ add offset to possible branch target address
- EX: Execute / effective address cycle
- MEM: Memory access cycle
- WB: Write-back cycle

Simple Implementation

- IF: Instruction fetch cycle
- ID: Instruction decode / register fetch cycle
- EX: Execute / effective address cycle
  - ⋆ memory reference: base address+offset to compute effective address
  - ⋆ register-register ALU instruction: perform the operation
  - ⋆ register-immediate ALU instruction: perform the operation
- MEM: Memory access cycle
- WB: Write-back cycle

Simple Implementation

- IF: Instruction fetch cycle
- ID: Instruction decode / register fetch cycle
- EX: Execute / effective address cycle
- MEM: Memory access cycle
  - ⋆ read or write based on effective address computed in last cycle
- WB: Write-back cycle

Simple Implementation

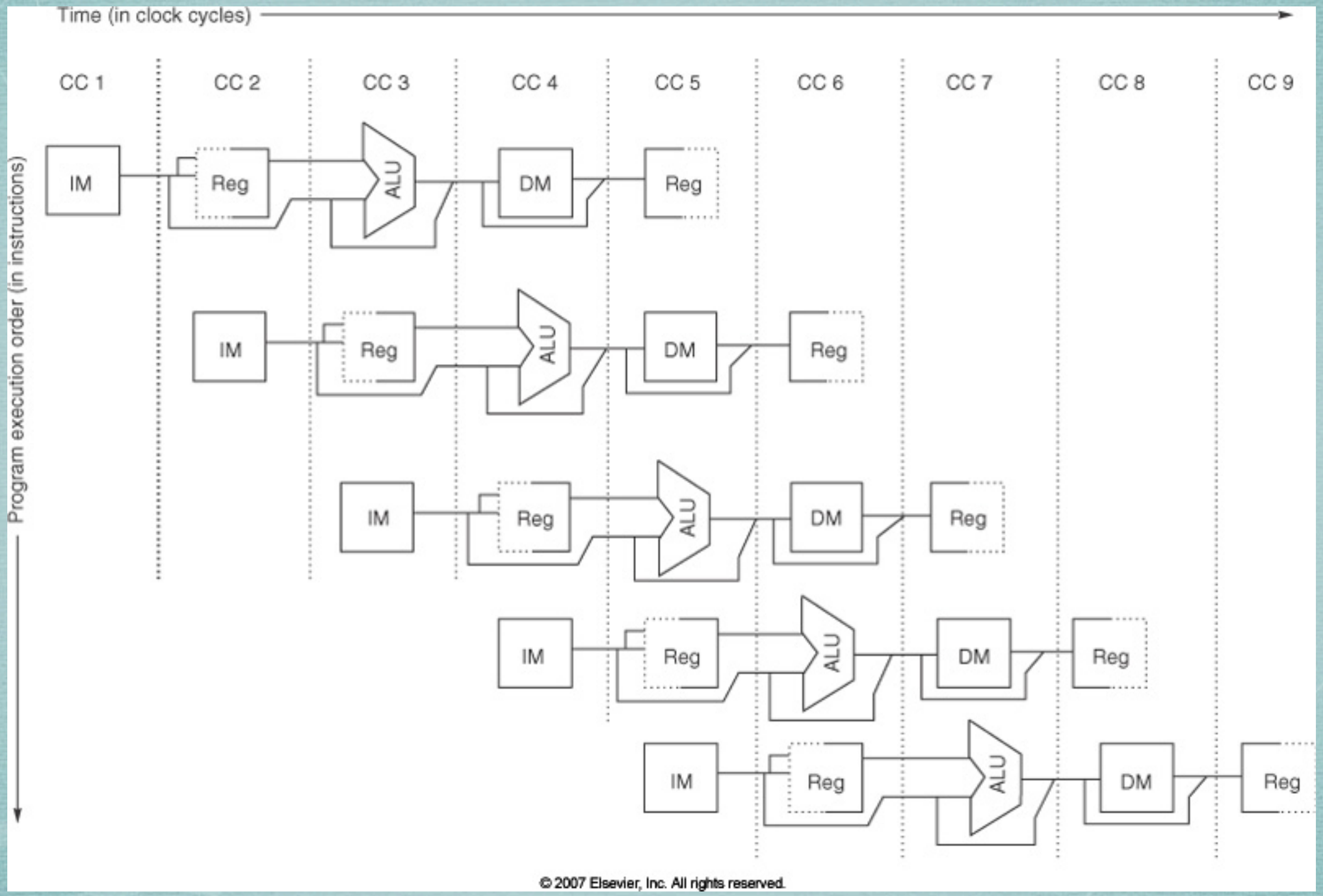
- IF: Instruction fetch cycle
- ID: Instruction decode / register fetch cycle
- EX: Execute / effective address cycle
- MEM: Memory access cycle
- WB: Write-back cycle
  - ⋆ register-register ALU instruction or Load: write result (computed or loaded from memory) into register file

Simple Implementation

- IF: Instruction fetch cycle
- ID: Instruction decode / register fetch cycle
- EX: Execute / effective address cycle
- MEM: Memory access cycle
- WB: Write-back cycle

branch = 2 cycles
store = 4 cycles
all else = 5 cycles
CPI = 4.54, assuming 12% branches, 10% stores

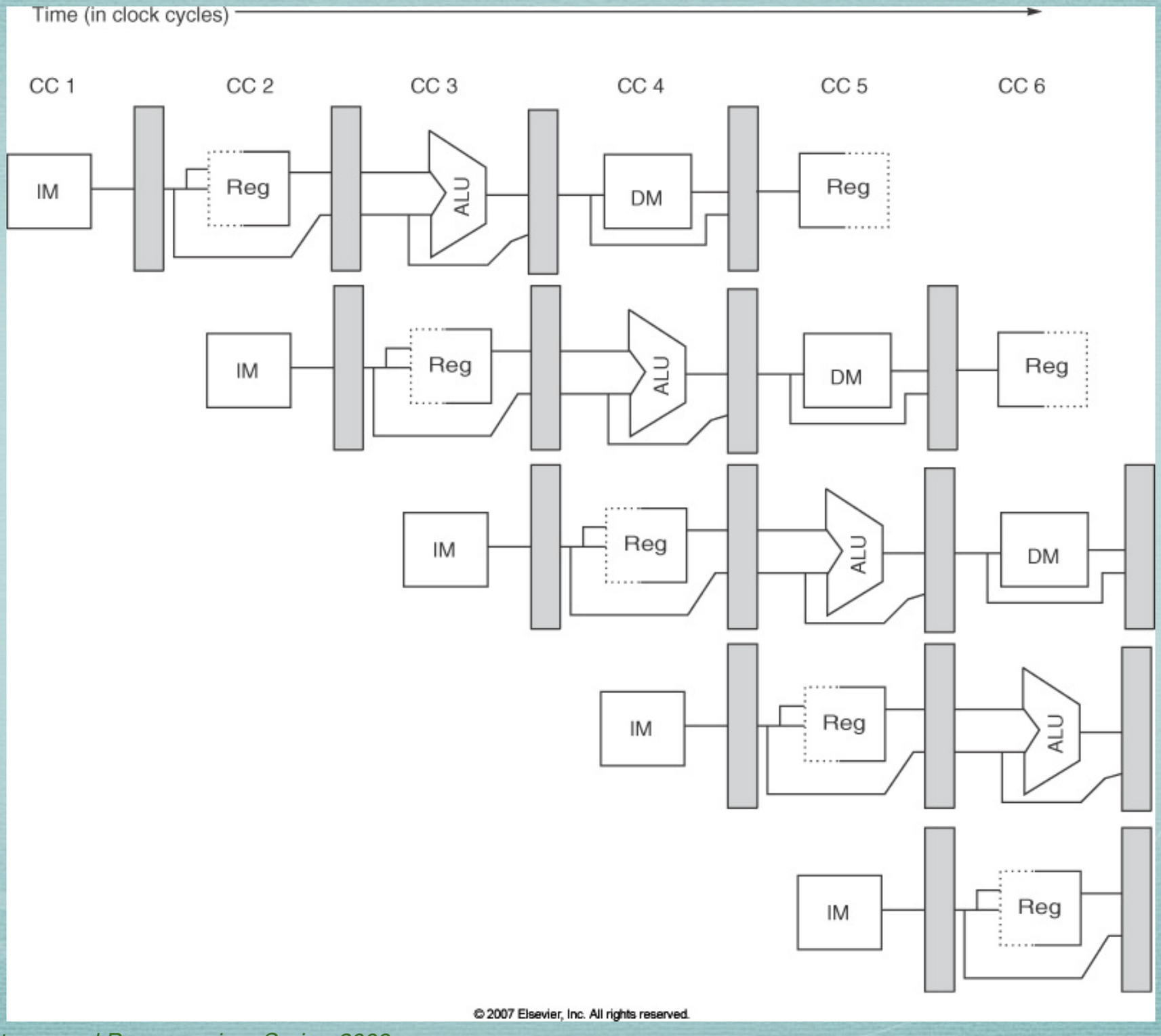
Simple Pipelined Implementation



Some Considerations

- Resource evaluation
  - ⋆ avoid resource conflicts across stages
- Separate instruction and data memories
  - ⋆ typically, with separate I and D caches
- Register access
  - ⋆ write in first half, read in second half
- PC not shown
  - ⋆ also need an adder to compute branch target
  - ⋆ branch does not change PC until ID (second) stage
    - ✳ ignore for now!

Prevent Interference



Observations

- Each instruction takes the same number of cycles
- Instruction throughput increases
  - ⋆ hence programs run faster
- Imbalance among pipeline stages reduces performance
- Overheads
  - ⋆ pipeline delays (register setup time)
  - ⋆ clock skew (clock cycle ≥ clock skew + latch overhead)
- Hazards ahead!

Pipeline Hazards

- Structural hazards
  - ⋆ not all instruction combinations possible in parallel
- Data hazards
  - ⋆ data dependence
- Control hazards
  - ⋆ control dependence

Pipeline Hazards

- Structural hazards
  - ⋆ not all instruction combinations possible in parallel
- Data hazards
  - ⋆ data dependence
- Control hazards
  - ⋆ control dependence

Hazards make it necessary to **stall** the pipeline

Quantifying the Stall Cost

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

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

CPI pipelined = Ideal CPI + Pipeline stall cycles per instruction
= 1 + Pipeline stall cycles per instruction

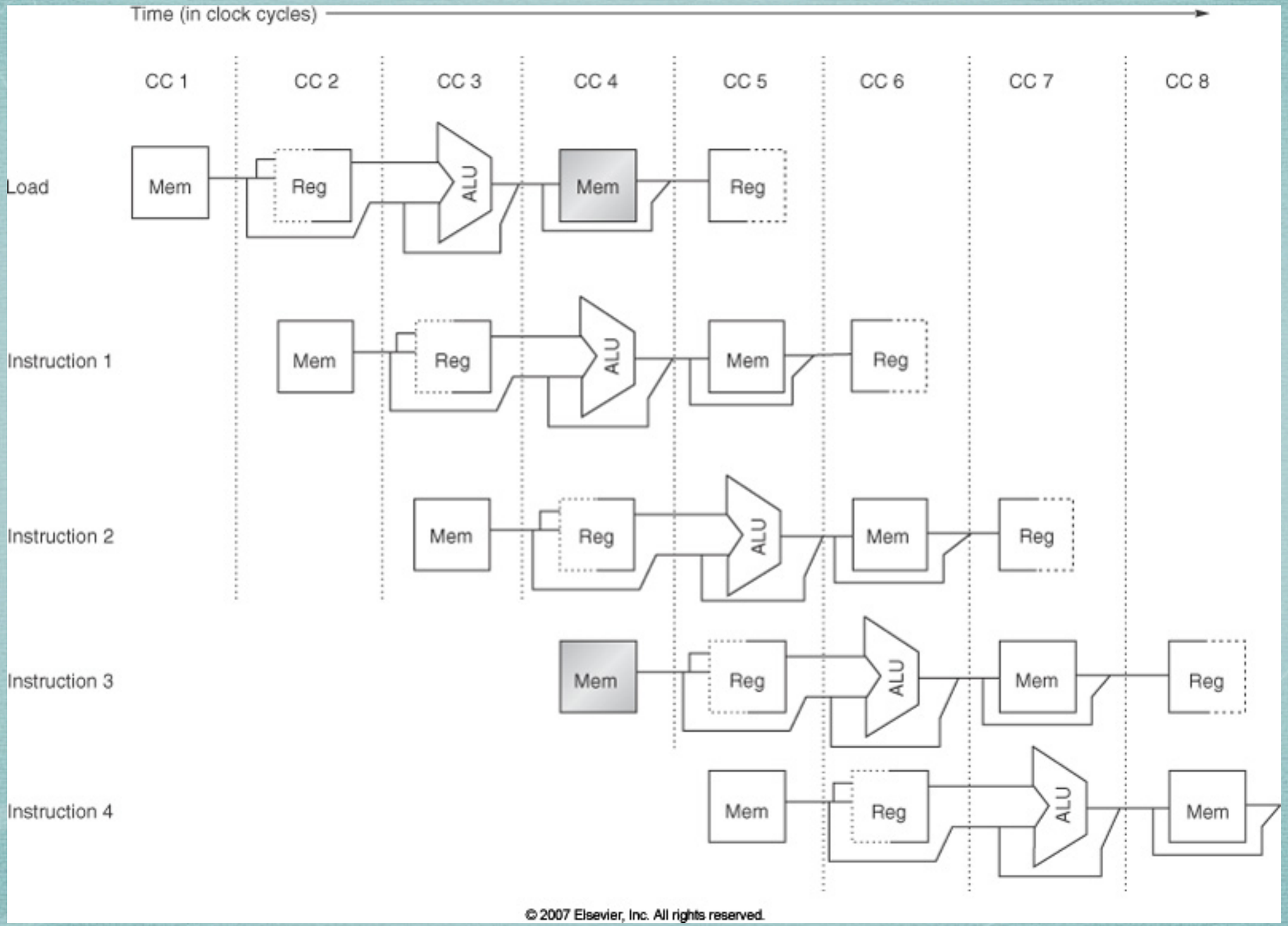
Ignoring pipeline overheads, assuming balanced stages,

Clock cycle unpipelined = Clock cycle pipelined

$$\text{Speedup} = \frac{\text{CPI Unpipelined} (\approx \text{Pipeline depth})}{1 + \text{Pipeline stall cycles per instruction}}$$

STRUCTURAL HAZARDS

Structural Hazard Example: Mem. Port Conflict



DATA HAZARDS

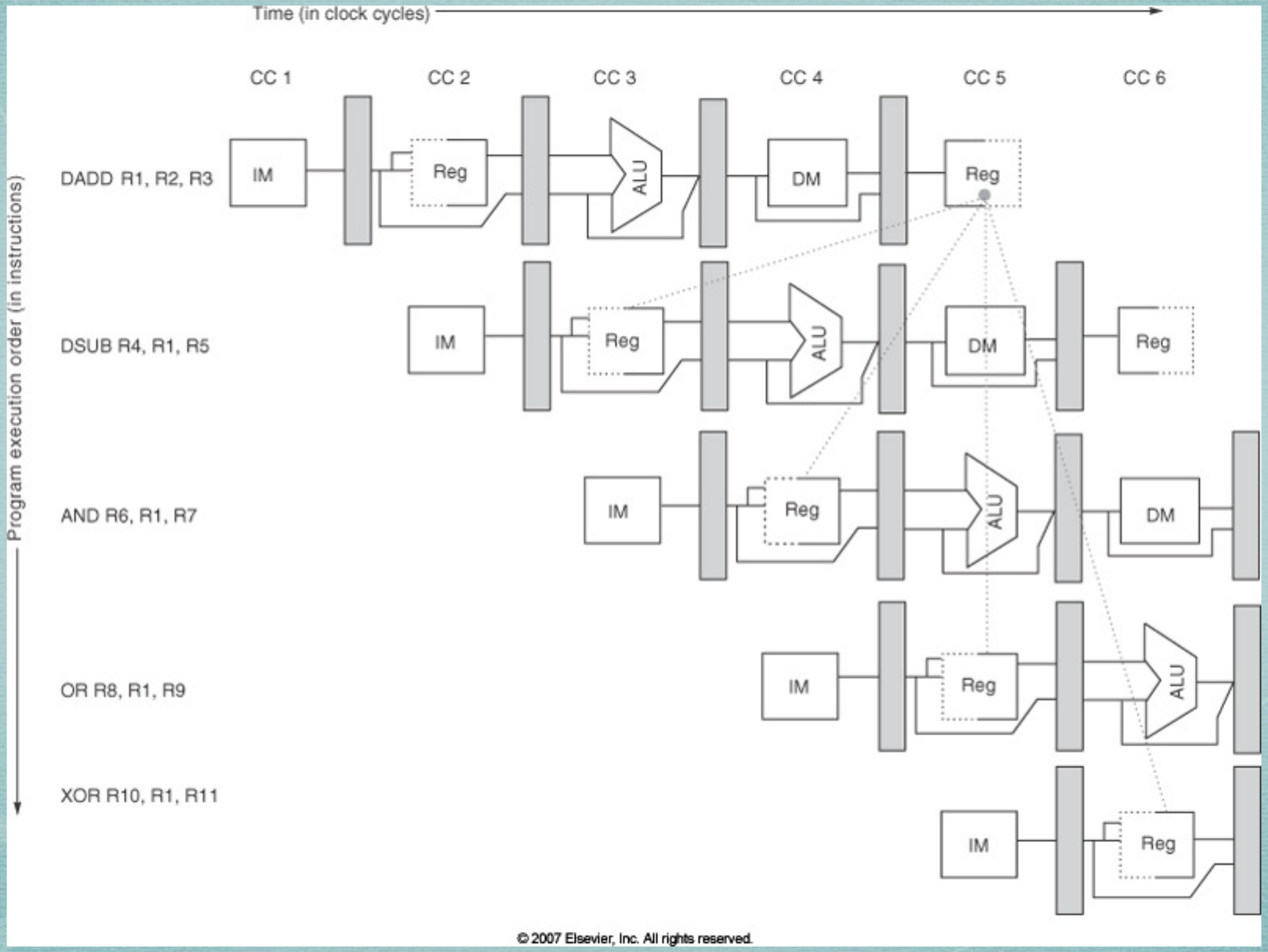
Data Hazard Types

- RAW: Read After Write
  - ⋆ true dependence
- WAR: Write After Read
  - ⋆ anti-dependence
- WAR: Write After Write
  - ⋆ output dependence
- RAR: Read After Read
  - ⋆ input dependence

Data Hazard Types

- RAW: Read After Write
  - ⋆ true dependence
- WAR: Write After Read
  - ⋆ anti-dependence
- WAR: Write After Write
  - ⋆ output dependence
- ~~RAR: Read After Read~~
  - ~~⋆ input dependence~~

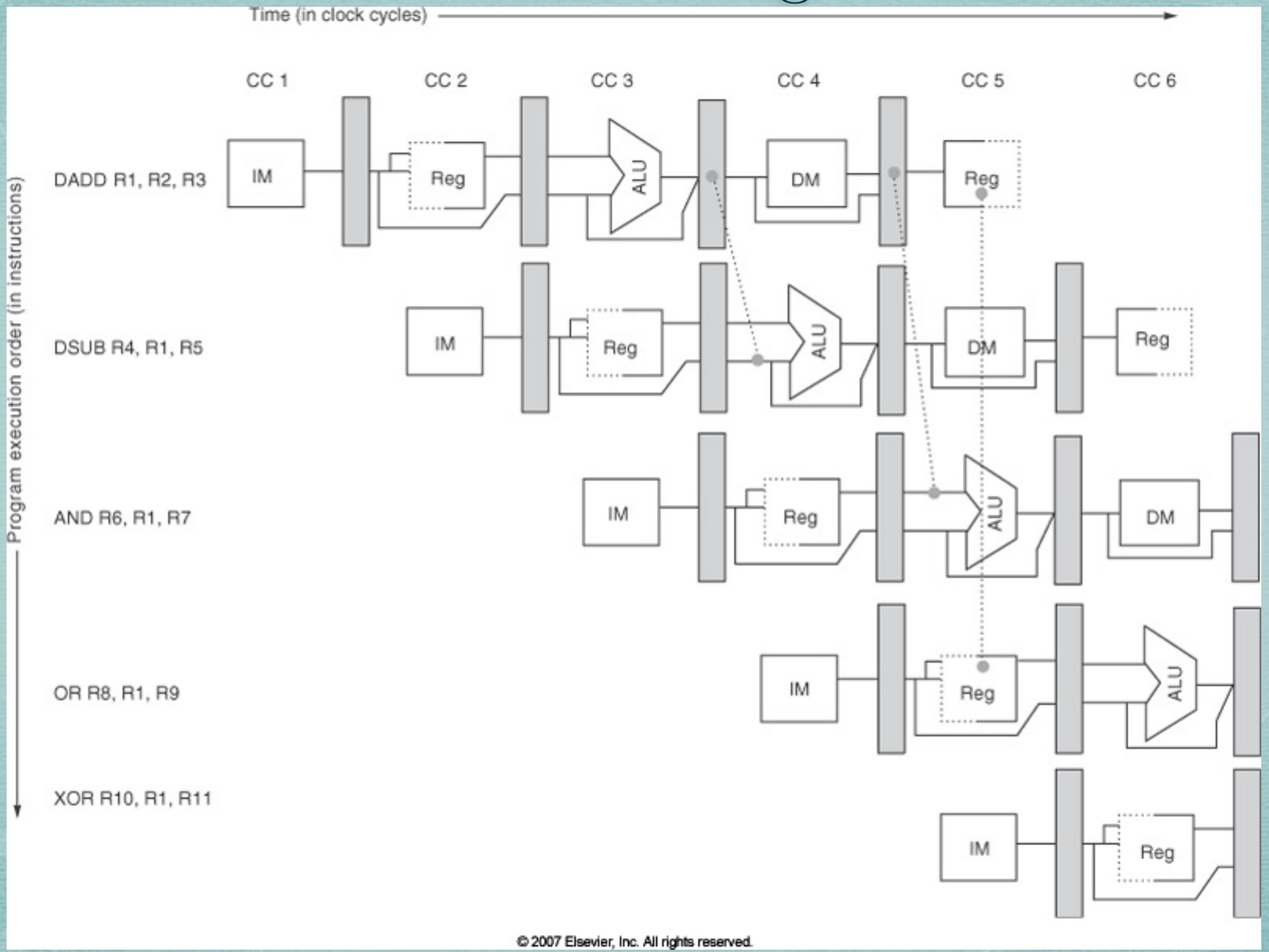
Data Hazard Example



Ameliorating Data Hazards

- Idea:
  - ⋆ ALU results from EX/MEM and MEM/WB registers fed back to ALU inputs
  - ⋆ if previous ALU operation wrote the register needed by the current operation, select the forwarded result

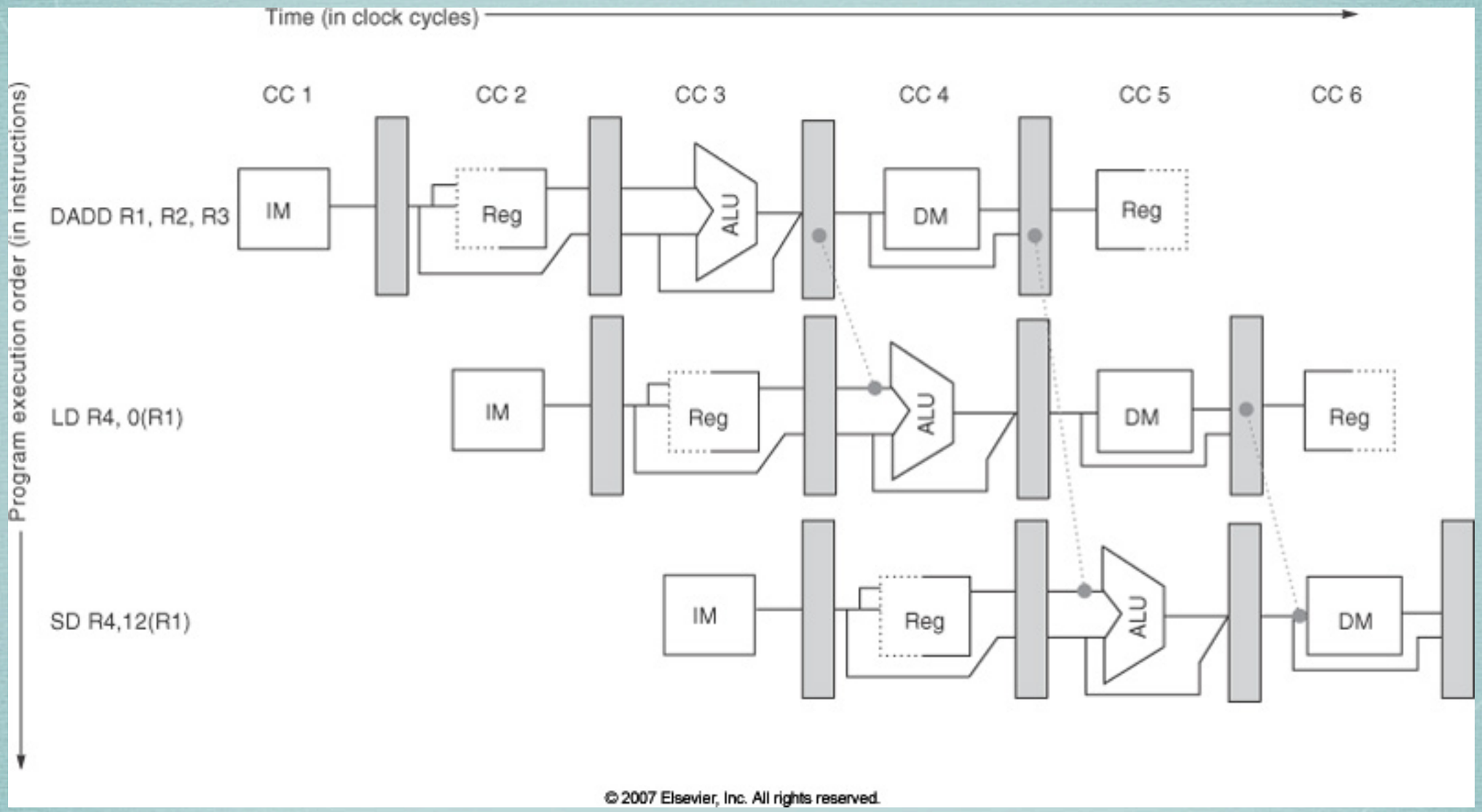
Forwarding



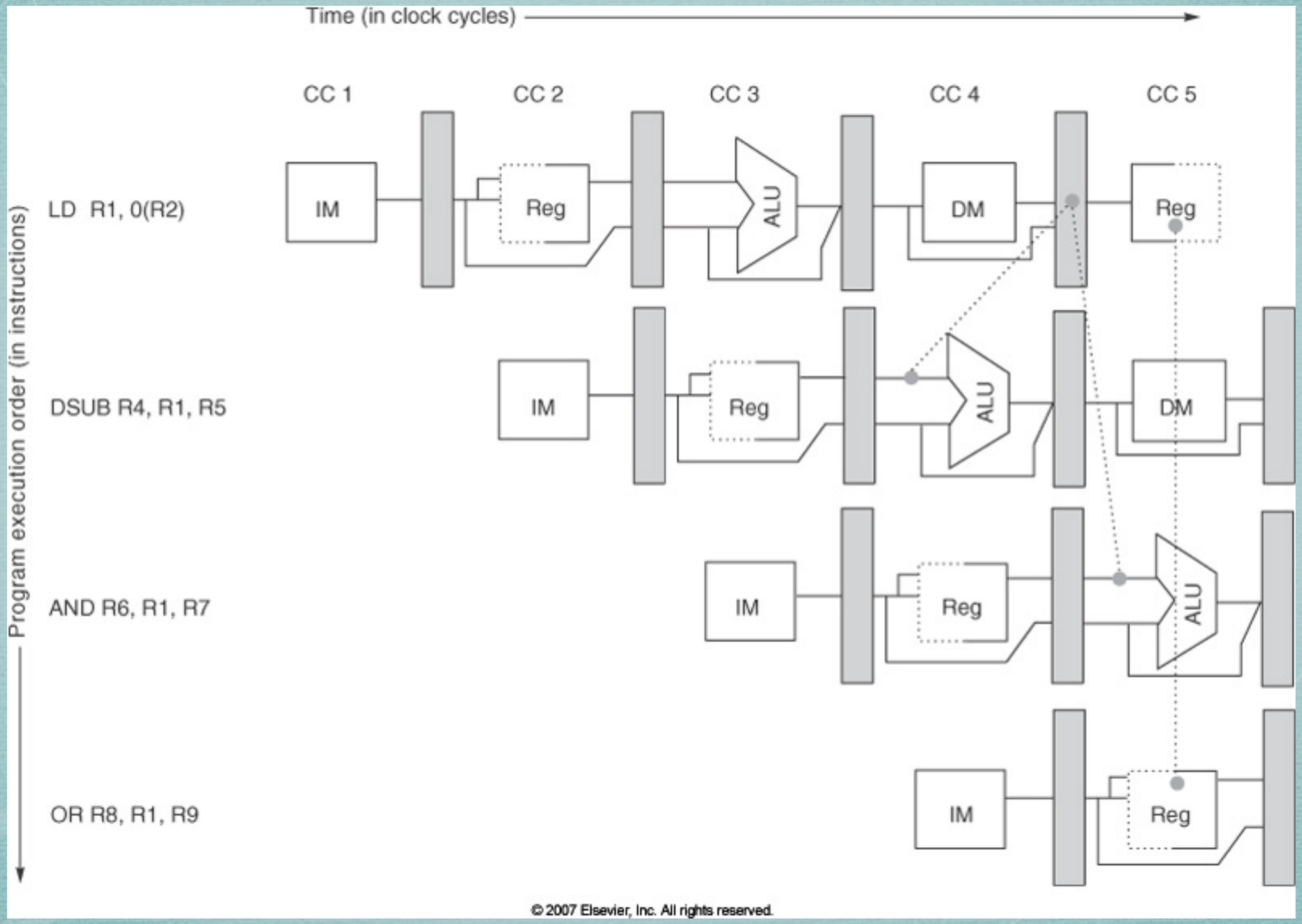
Ameliorating Data Hazards

- Idea:
  - ⋆ ALU results from EX/MEM and MEM/WB registers fed back to ALU inputs
  - ⋆ if previous ALU operation wrote the register needed by the current operation, select the forwarded result
- Observations:
  - ⋆ forwarding needed across multiple cycles (how many?)
  - ⋆ forwarding may be implemented across functional units
    - ✳ e.g., output of one unit may be forwarded to input of another, rather than the input of just the same unit

Forwarding Across Multiple Units



Not All Stalls Avoided



CONTROL HAZARDS

Handling Branch Hazard

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

Reducing Branch Penalty

- "Freeze" or "flush" the pipeline
- Treat every branch as not-taken ("predicted-untaken")
  - ⋆ need to handle taken branches by roll-back
- Treat every branch as taken ("predicted-taken")
- Delayed branch

Freezing Pipeline

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

Delayed Branch

```
branch instruction
sequential successor_1
branch target if taken
```

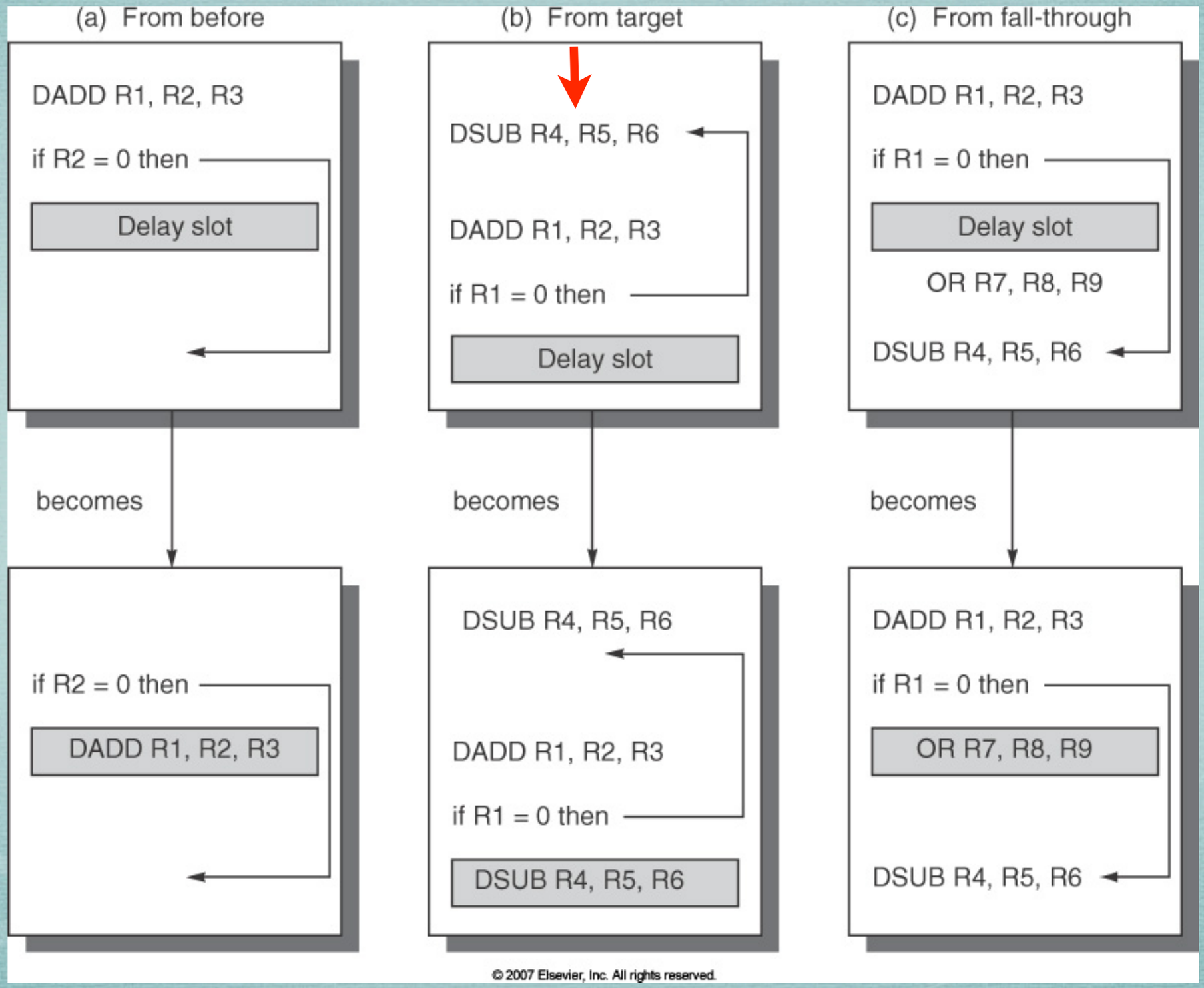
Behavior of Delayed Branch

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

Schedule of Branch Delay Slot

Schedule of Branch Delay Slot



HOW IS PIPELINING IMPLEMENTED?

Simple MIPS Implementation

- Instruction fetch cycle (IF)
- Instruction decode/register fetch cycle (ID)
- Execution / effective address cycle (EX)
- Memory access / branch completion cycle (MEM)
- Write-back cycle (WB)

Simple MIPS Implementation: IF
(IF → ID → EX → MEM → WB)

- Fetch

```
IR ← Mem[PC];
NPC ← PC + 4;
```

Simple MIPS Implementation: ID
(IF → ID → EX → MEM → WB)

- Decode

```
A ← Regs[rs];
B ← Regs[rt];
Imm ← sign-extended immediate field of IR;
```

Simple MIPS Implementation: ID
(IF → ID → EX → MEM → WB)

- Execution
  - ⋆ Memory reference

```
ALUOutput ← A + Imm;
```

  - ⋆ Register-Register ALU instruction

```
ALUOutput ← A func B;
```

  - ⋆ Register-Immediate ALU instruction

```
ALUOutput ← A op Imm;
```

  - ⋆ Branch

```
ALUOutput ← NPC + (Imm << 2);
Cond ← (A == 0);
```

Simple MIPS Implementation: ID
(IF → ID → EX → MEM → WB)

- Memory access / branch completion
  - ⋆ Memory reference

```
LMD ← Mem[ALUOutput] or
Mem[ALUOutput] ← B;
```

  - ⋆ Branch

```
if (cond) PC ← ALUOutput;
```

Simple MIPS Implementation: ID
(IF → ID → EX → MEM → WB)

- Write-back
  - ⋆ Register-Register ALU instruction

```
Regs[rd] ← ALUOutput;
```

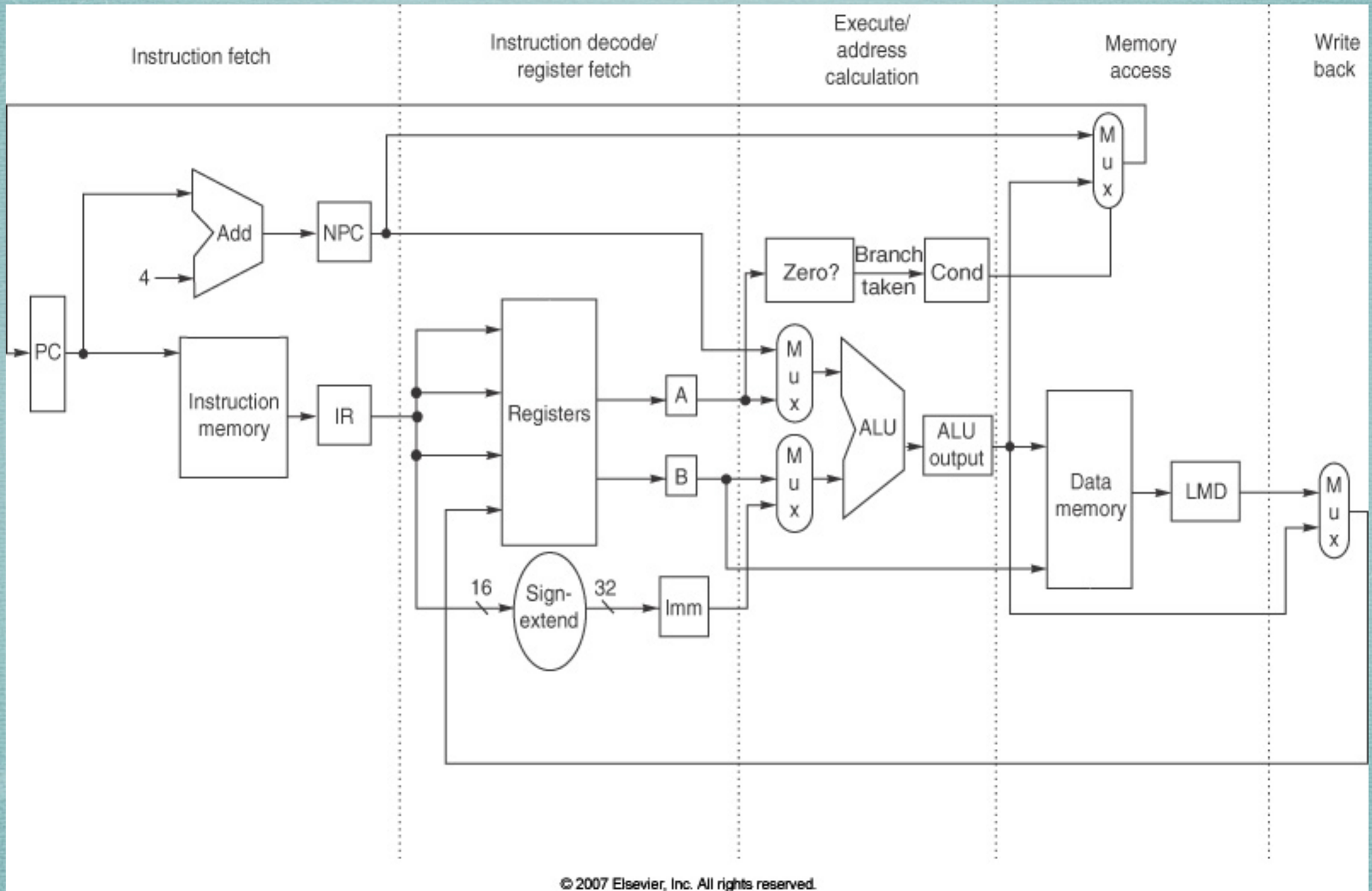
  - ⋆ Register-Immediate ALU instruction

```
Regs[rt] ← ALUOutput;
```

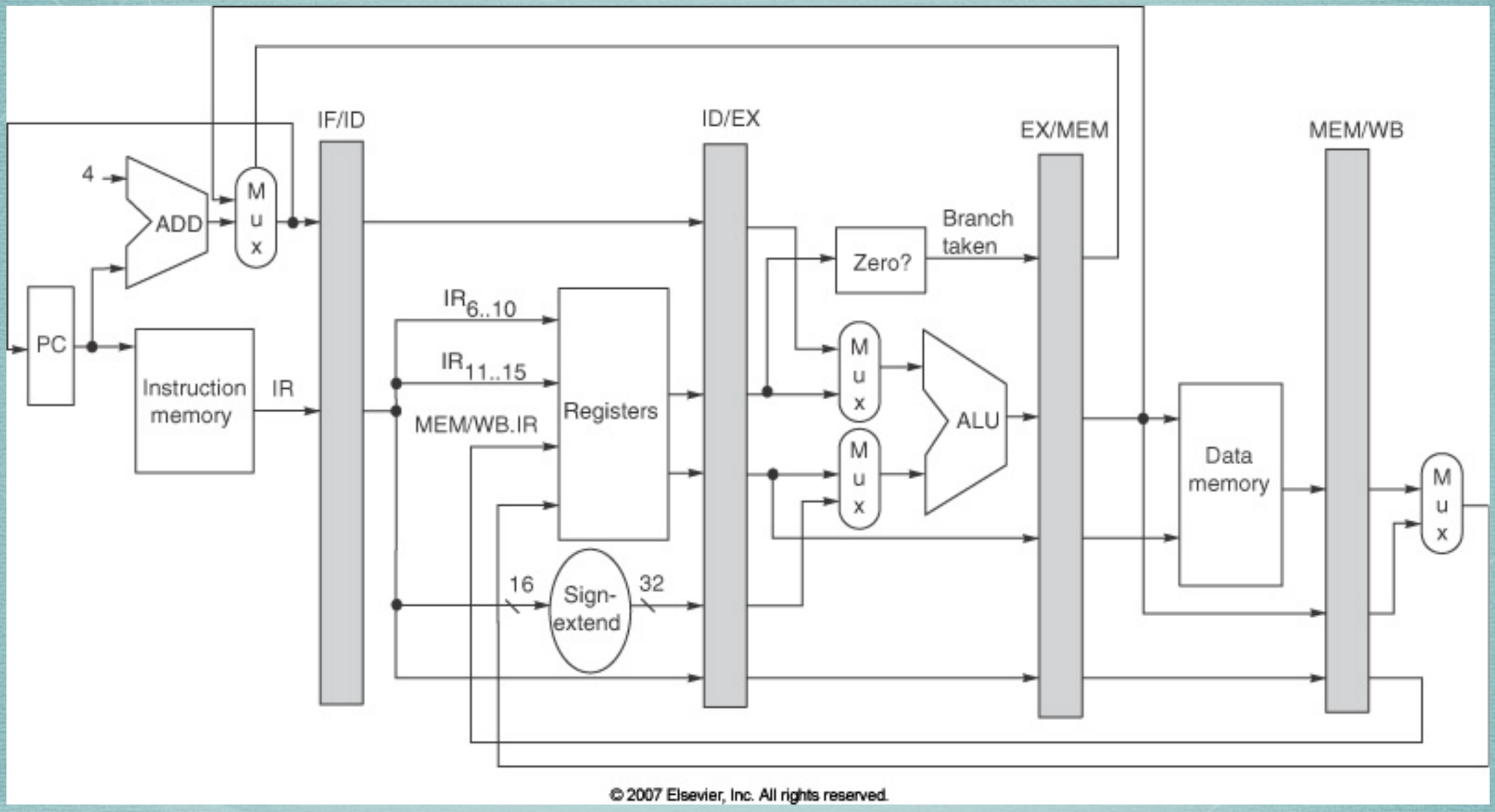
  - ⋆ Load instruction

```
Regs[rt] ← LMD;
```

MIPS Data Path



MIPS Data Path: Pipelined



Situations for Data Hazard

Situation	Example code sequence	Action

No dependence	`LD R1,45(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 R1,45(R2)`
	`DADD R5,R1,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 R1,45(R2)`
	`DADD R5,R6,R7`
	`DSUB R8,R1,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 R1,45(R2)`
	`DADD R5,R6,R7`
	`DSUB R8,R6,R7`
	`OR R9,R1,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 Data Hazards

Opcode field of ID/EX ($ID/EX.IR_{0..5}$)	Opcode field of IF/ID ($IF/ID.IR_{0..5}$)	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]

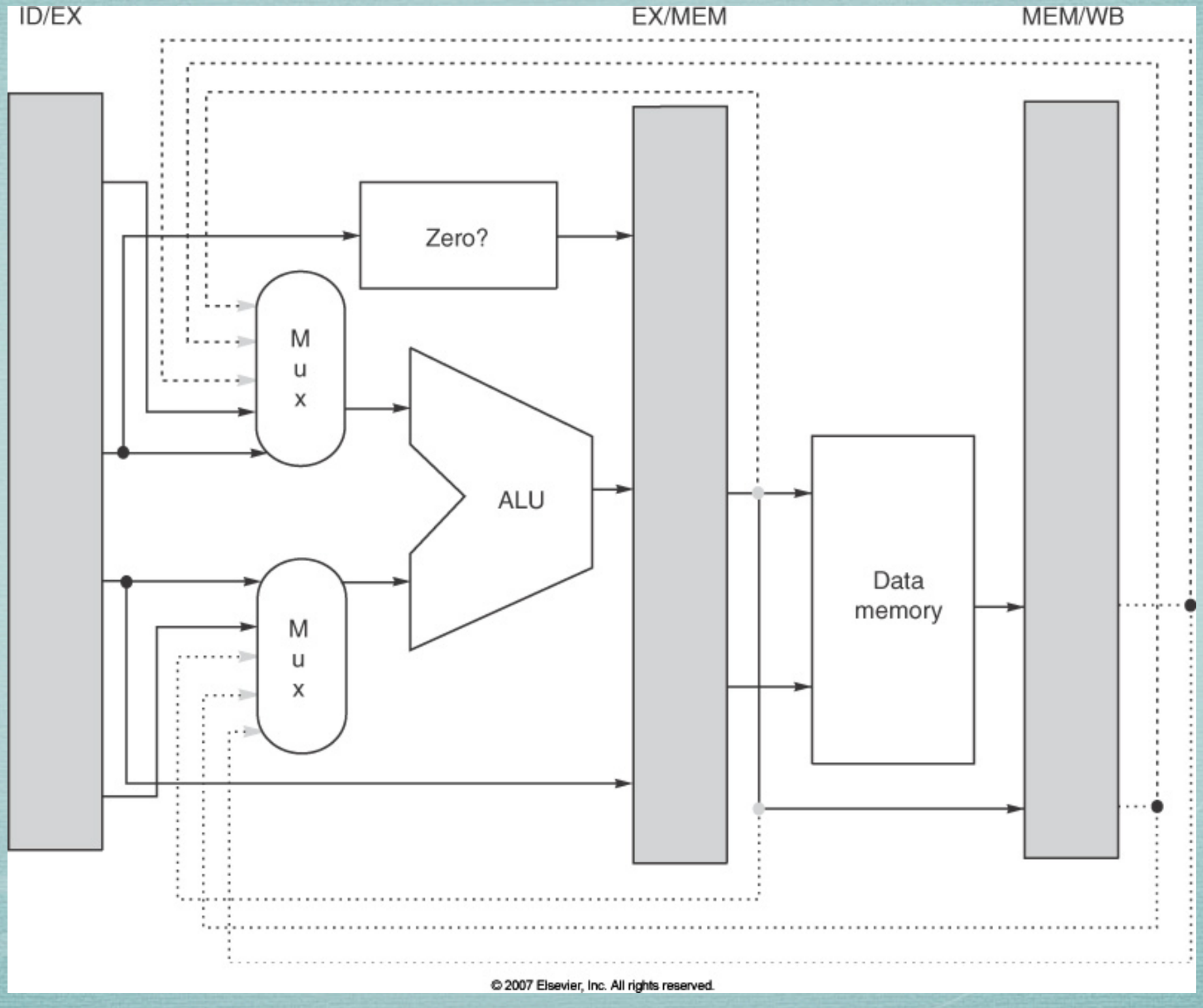
Forwarding

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]
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]

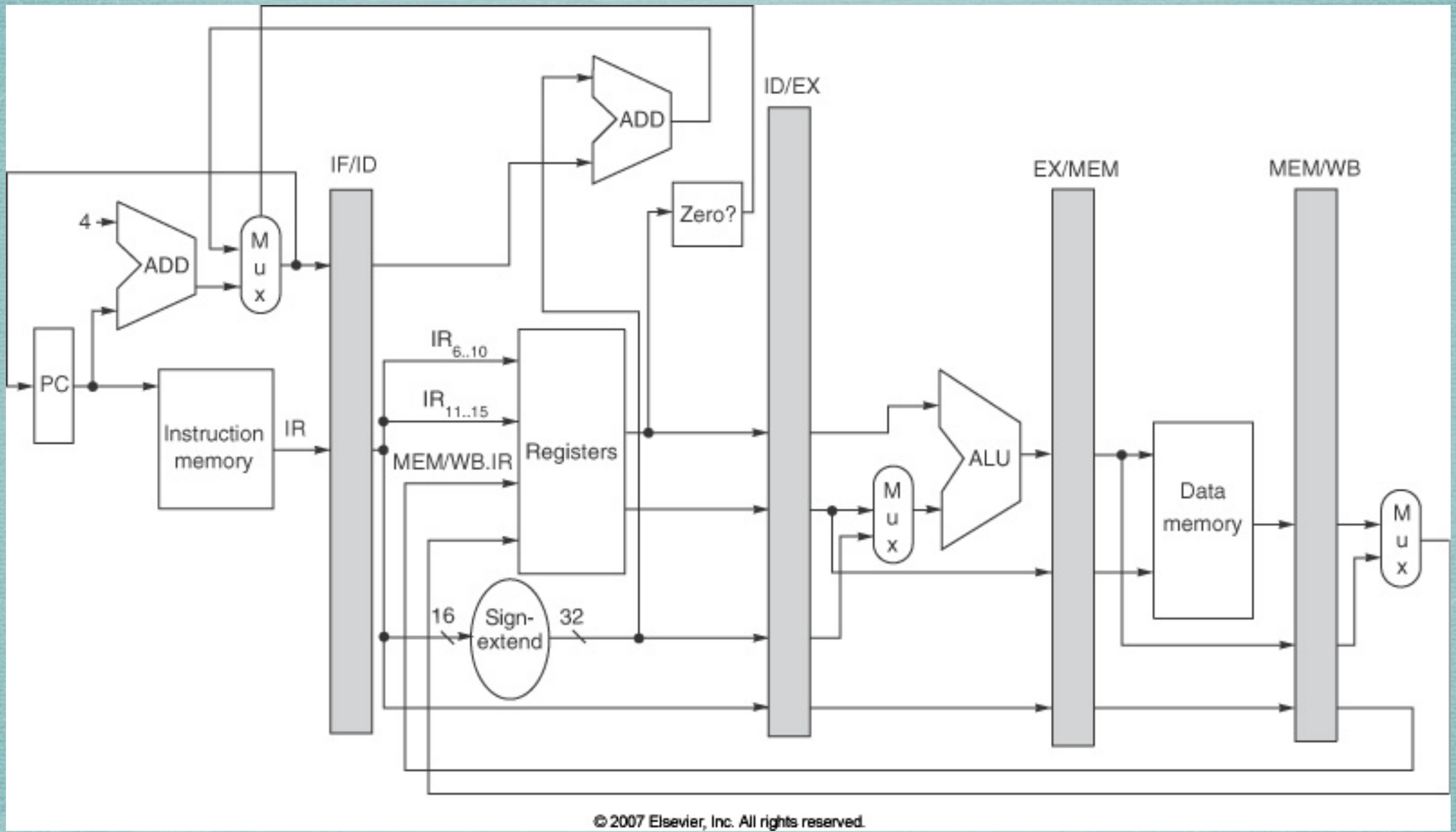
Forwarding

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]
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]

Forwarding Implemented



Reducing the Branch Delay



EXCEPTIONS

Types of Exceptions

- I/O device request
- Invoking an OS service
- Tracing
- Breakpoint
- Integer arithmetic overflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory access
- Memory protection violation
- Undefined or unimplemented instruction
- Hardware malfunction
- Power failure

Exception Categories

- Synchronous vs Asynchronous
- User requested vs coerced
- User maskable vs nonmaskable
- Within vs between instructions
- Resume vs terminate

Exception Categorization

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Handling Exceptions

- Force a "trap" into the pipeline on the next IF
- Turn of all writes until the trap is taken off
  - ⋆ place zeros into pipeline latches of instructions following the one causing exception
- Exception handler saves PC and returns to it
- Delayed branches cause a problem
  - ⋆ need to save delay slots plus one number of PCs

(Precise) Exceptions in MIPS

Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

(Precise) Exceptions in MIPS

Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

LD	IF	ID	EX	MEM	WB	
DADD		IF	ID	EX	MEM	WB

(Precise) Exceptions in MIPS

Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

LD	IF	ID	EX	MEM	WB	
DADD		IF	ID	EX	MEM	WB

Use exception status vector for each instruction

Complications due to Complex Instructions

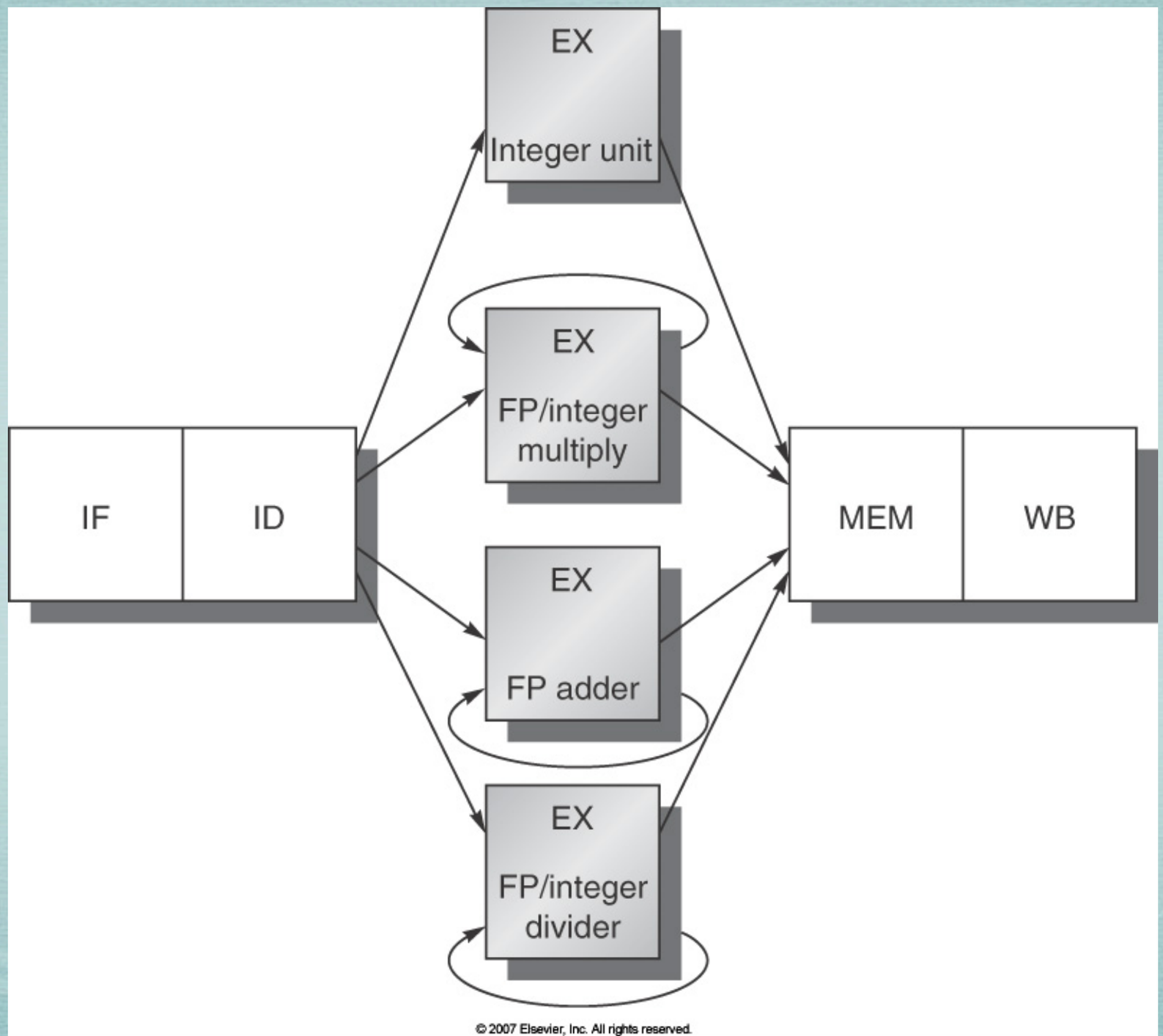
- Instructions that change processor states before they are committed
  - ⋆ autoincrement
  - ⋆ string copy (change memory state)
- State bits
  - ⋆ implicitly condition codes (flags)
    - ✳ instructions setting condition codes not allowed in delay slots
- Multicycle operations
  - ⋆ use of microinstructions

ADDING FLOATING POINT SUPPORT

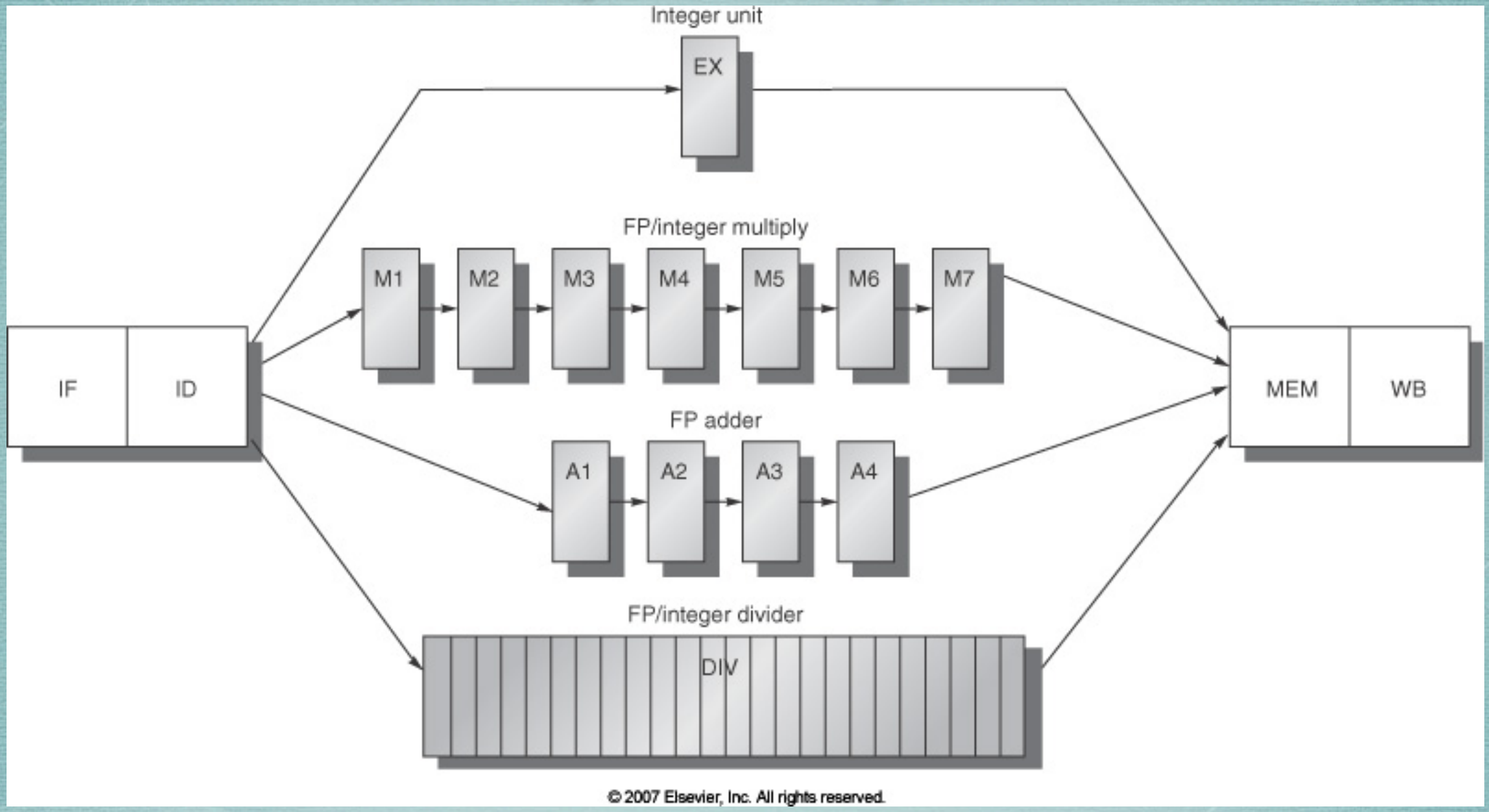
Functional Units

- Main integer unit
  - ⋆ handles loads, stores, integer ALU operations, branches
- FP and integer multiplier
- FP adder
  - ⋆ handles FP add, subtract, and conversion
- FP and integer divider

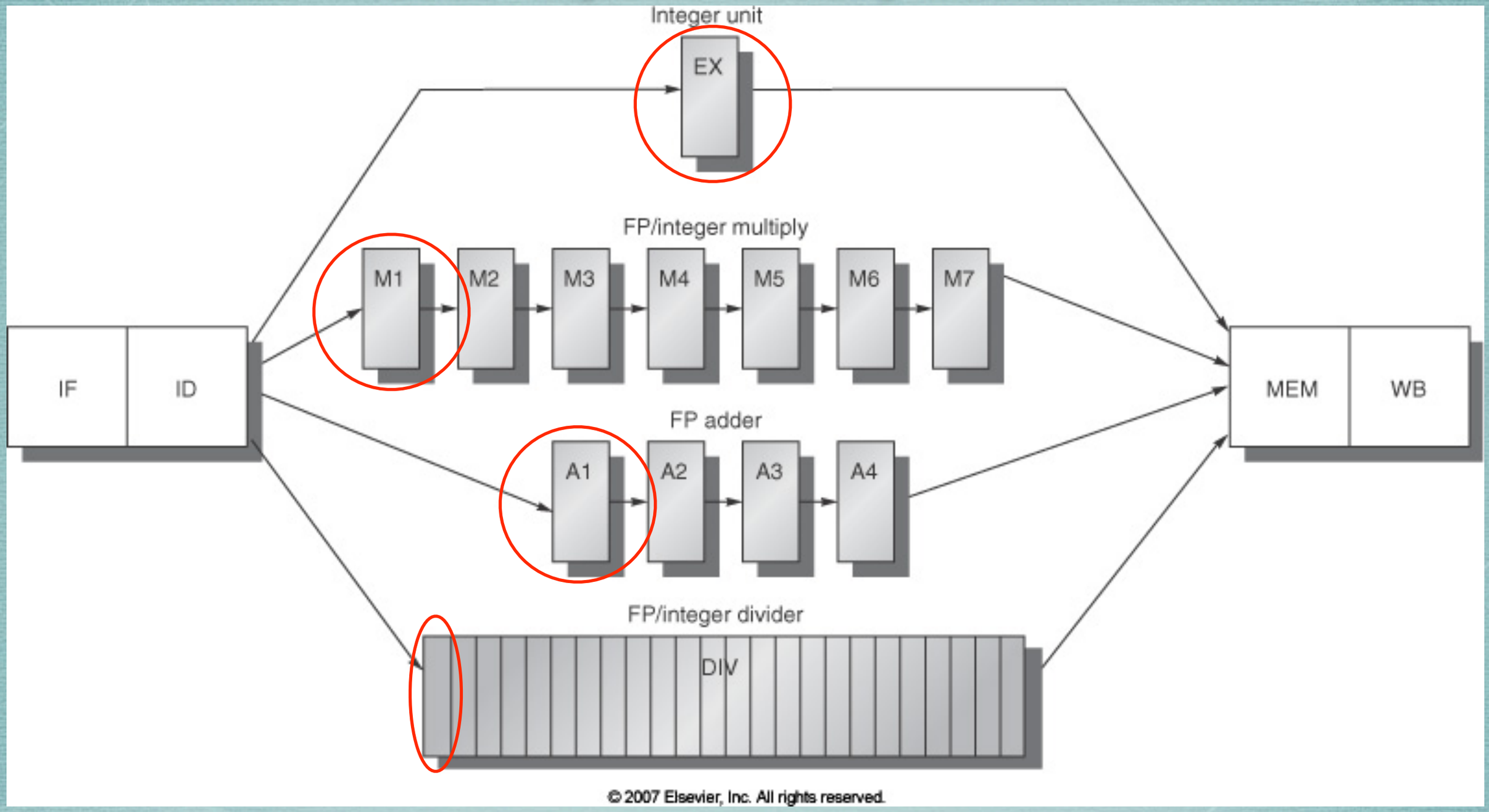
Extended Pipeline



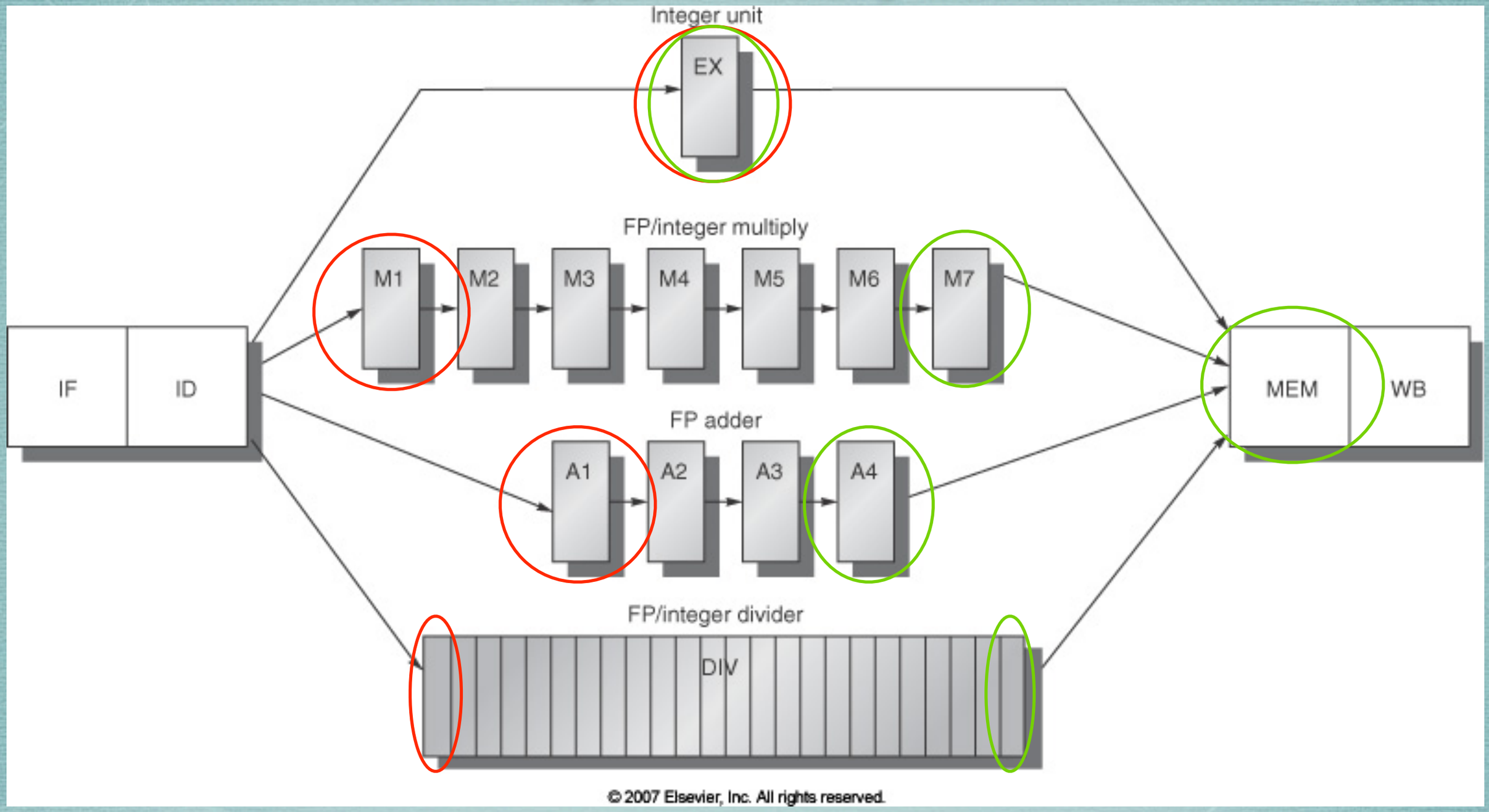
Extended Pipeline: Expanded View



Extended Pipeline: Expanded View



Extended Pipeline: Expanded View



Additional Complications

- Structural hazard because divide unit is not pipelined
- Number of register writes in a cycle may more than one
  - WAW hazards possible
    - ⋆ WAR hazards not possible
  - Maintaining precise exceptions
    - ⋆ out-of-order completion
  - More number of stalls due to RAW hazards, due to longer pipeline

Example: RAW Hazard

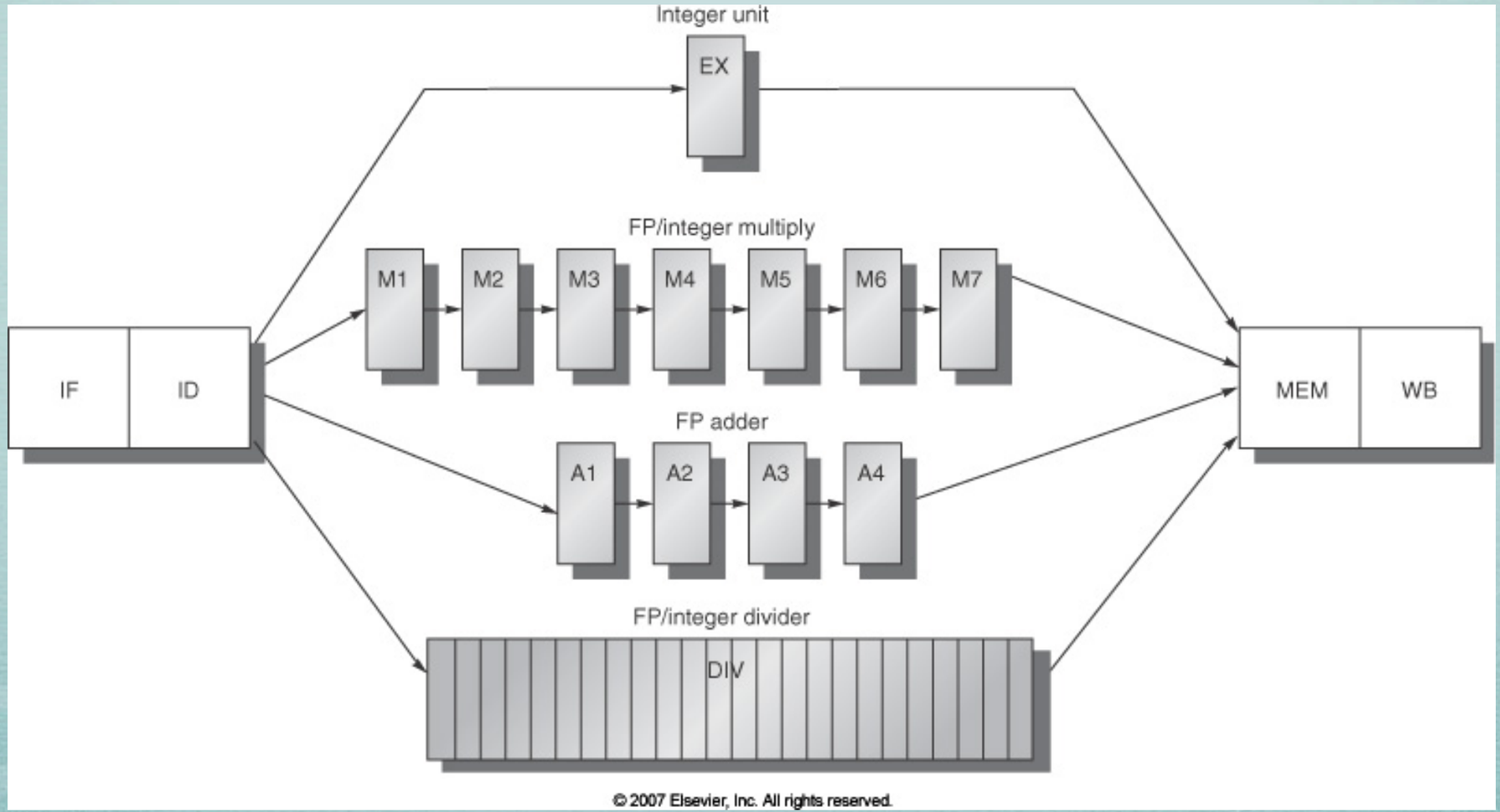
	Clock cycle number																
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L.D F4,0(R2)	IF	ID	EX	MEM	WB												
MUL.D F0,F4,F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D F2,F0,F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	WB
S.D F2,0(R2)					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM

Example 2

	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

Handling the Problems

- Register file conflict detection
  - ⋆ detect at ID stage
  - ⋆ detect at MEM or WB stage

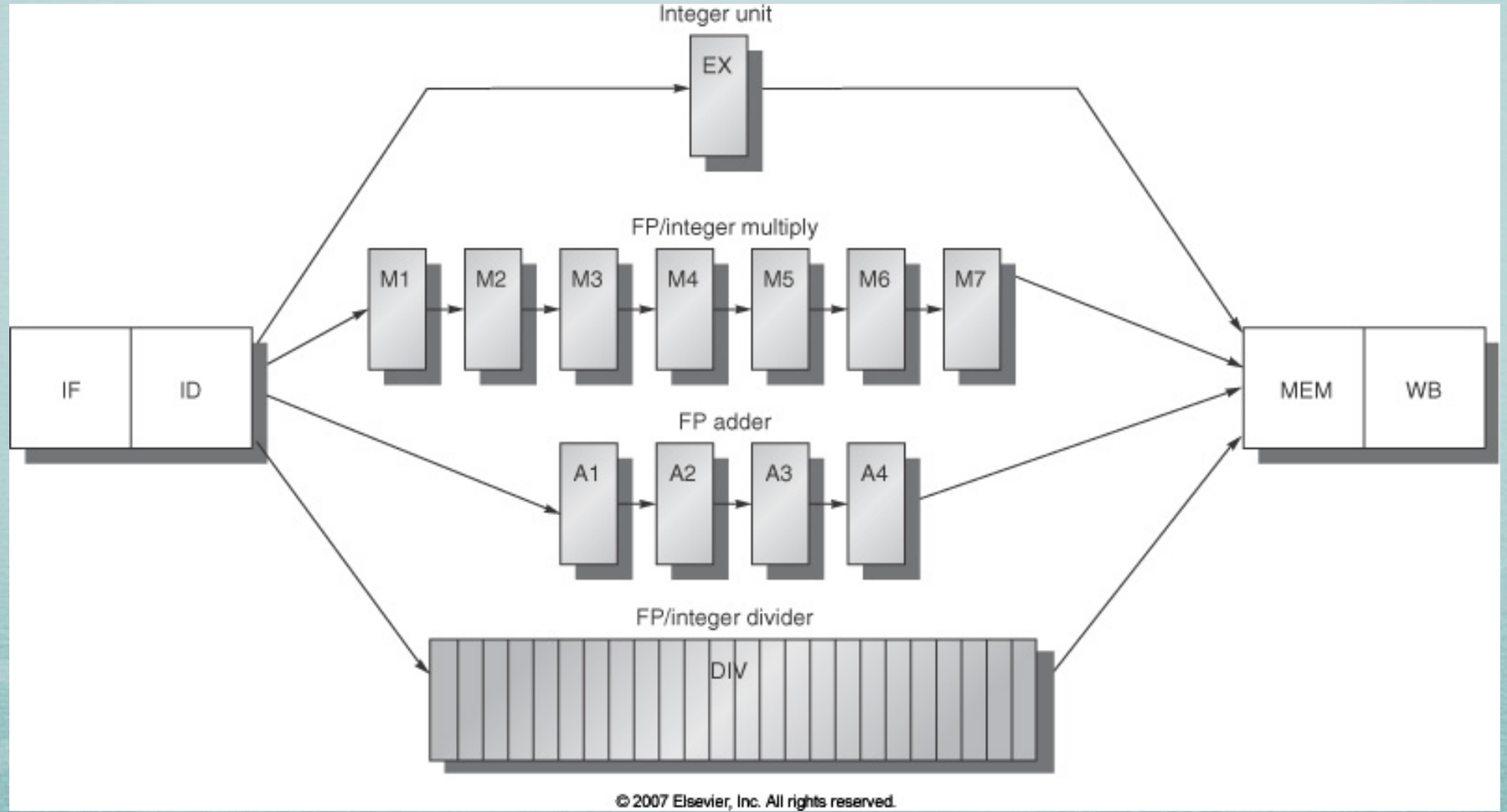


Handling the Problems

- Register file conflict detection
  - ⋆ detect at ID stage, or
  - ⋆ detect at MEM or WB stage
- WAW hazard detection
  - ⋆ delay the issue of second instruction, or
  - ⋆ stamp out the result of the first instruction (convert it into noop)

Summary of Checks

- Check for structural hazards
- Check for RAW data hazard
- Check for WAW data hazard



Maintaing Precise Exceptions

```
DIV.D       F0,F2,F4
ADD.D       F10,F10,F8
SUB.D       F12,F12,F14
```

- Ignore the problem
  - ⋆ settle for imprecise exceptions
- Buffer the results
  - ⋆ simple buffer could be very big
  - ⋆ history file
  - ⋆ future file
- Let trap handling routines clean up
- Hybrid scheme: allow issue when earlier instructions can no longer raise exception

DYNAMIC SCHEDULING

Idea Behind Dynamic Scheduling

- Split ID stage:
  - ⋆ Issue -- decode, check for structural hazards
  - ⋆ Read operands -- wait until no data hazards, then read
- Other stages remain as before

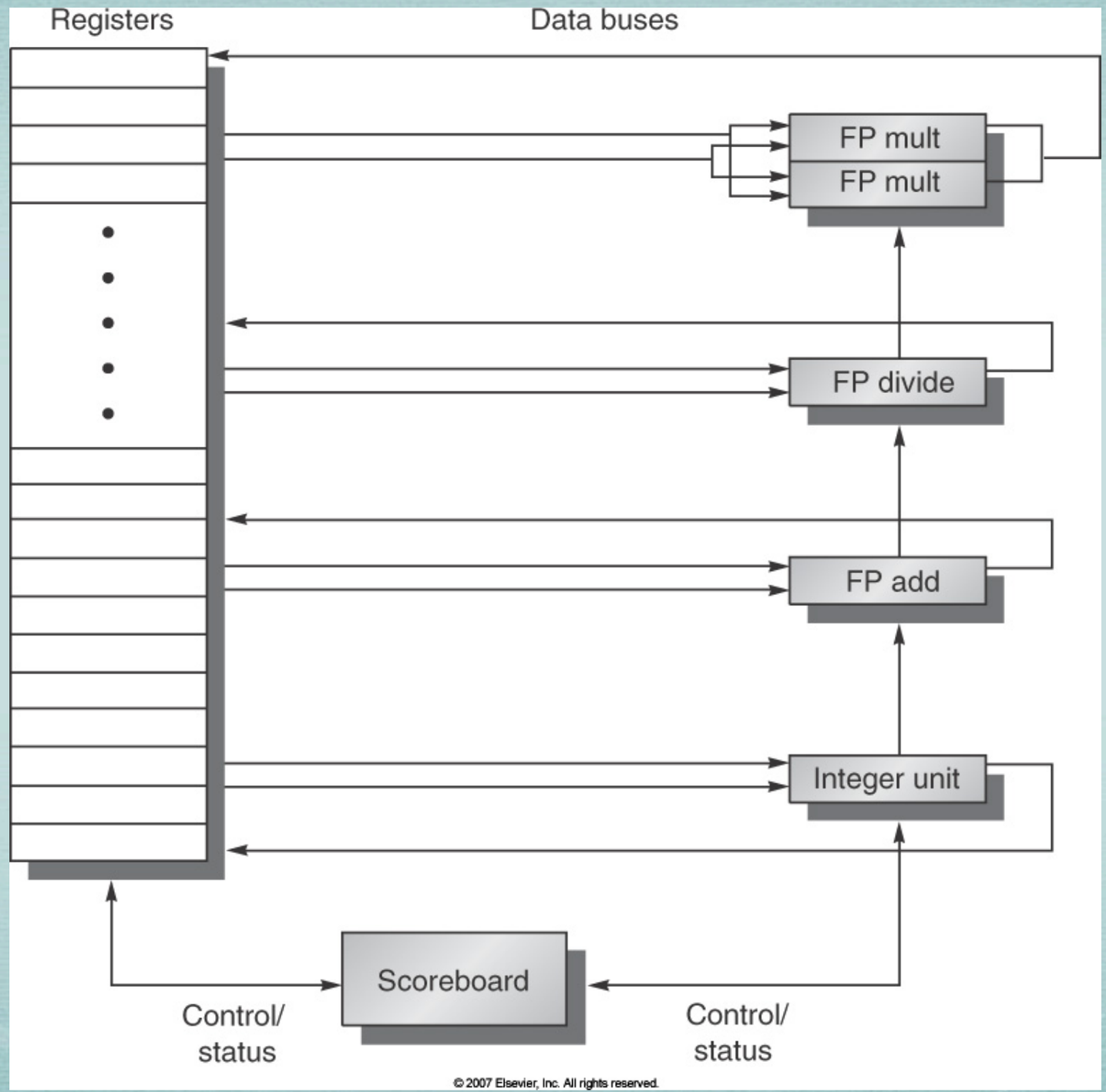
Idea Behind Dynamic Scheduling

- Split ID stage:
  - ⋆ Issue -- decode, check for structural hazards
  - ⋆ Read operands -- wait until no data hazards, then read
- Other stages remain as before

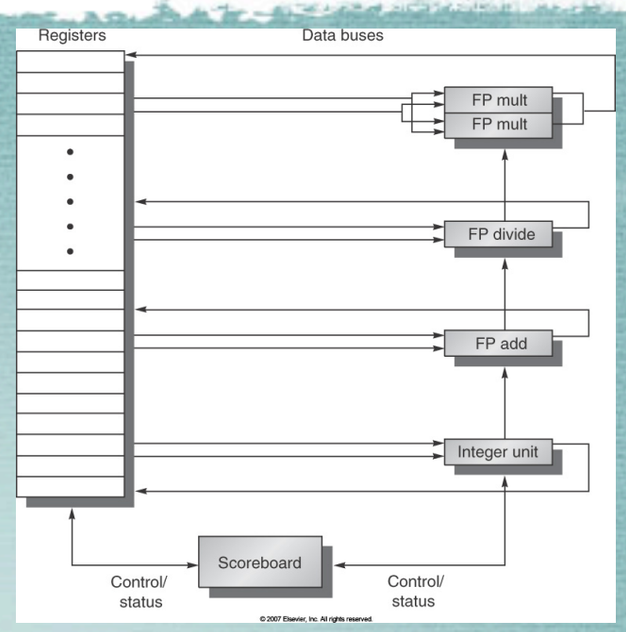
```
DIV.D   F0, F2, F4
ADD.D   F10,F0,F8
SUB.D   F8,F8,F14
```

Must take care of WAR and WAW hazards.

Scoreboarding (CDC 6600)



Scoreboarding (CDC 6600)



- Issue
  - ⋆ check for free functional units
  - ⋆ check if another instruction has the same destination register (WAW hazard detection)
- Read operands
  - ⋆ monitor availability of source operands (RAW hazard detection)
- Execution
  - ⋆ functional unit notifies scoreboard of completion
- Write result
  - ⋆ check for WAR hazards, stall write if necessary

Scoreboarding: Effectiveness

- Relative easy to implement the logic
  - ⋆ only as much as a functional unit
  - ⋆ but four times as many buses as without scoreboard
- Reduces Clocks Per Instruction (CPI)
  - ⋆ tries to make use of the available Instruction Level Parallelism (ILP)
  - ⋆ 1.7x speedup for Fortran, 2.5x for hand-coded assembly

Scoreboarding: Limitations

- Overlapping instructions must be picked from a single basic block
- Window: number of scoreboard entries
- Number and types of functional units
- Presence of anti- and output-dependences

Pitfalls

- Unexpected execution sequences may cause unexpected hazards

```
        BNEZ    R1, foo
        DIV.D   F0,F2,F4
        ...
        ...
foo:    L.D     F0,qrs
```

- Extensive pipelining can impact other aspects of a design
- Evaluating dynamic or static scheduling on the basis of unoptimized code