# ECSE 425 Lecture 6: Pipelining

H&P, Appendix A

## Last Time

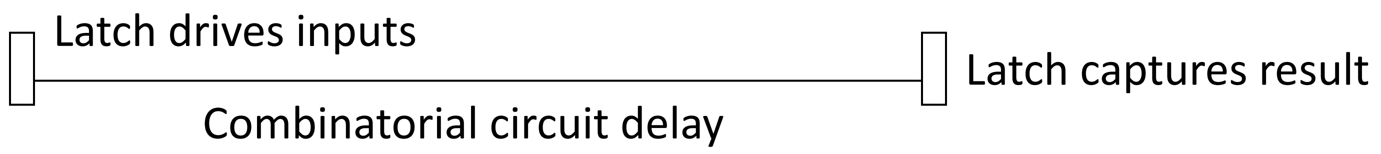
- Processor Performance Equation
- System performance
- Benchmarks

## Today

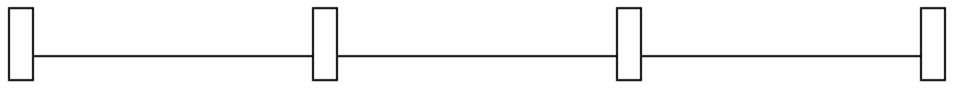
- Pipelining Basics
- RISC Instruction Set Architecture
- Unpipelined RISC Implementation
- First glance: Pipelining RISC

## What is Pipelining?

- Consider the time needed (gate delays) to execute an instruction
  - The time between two clock edges



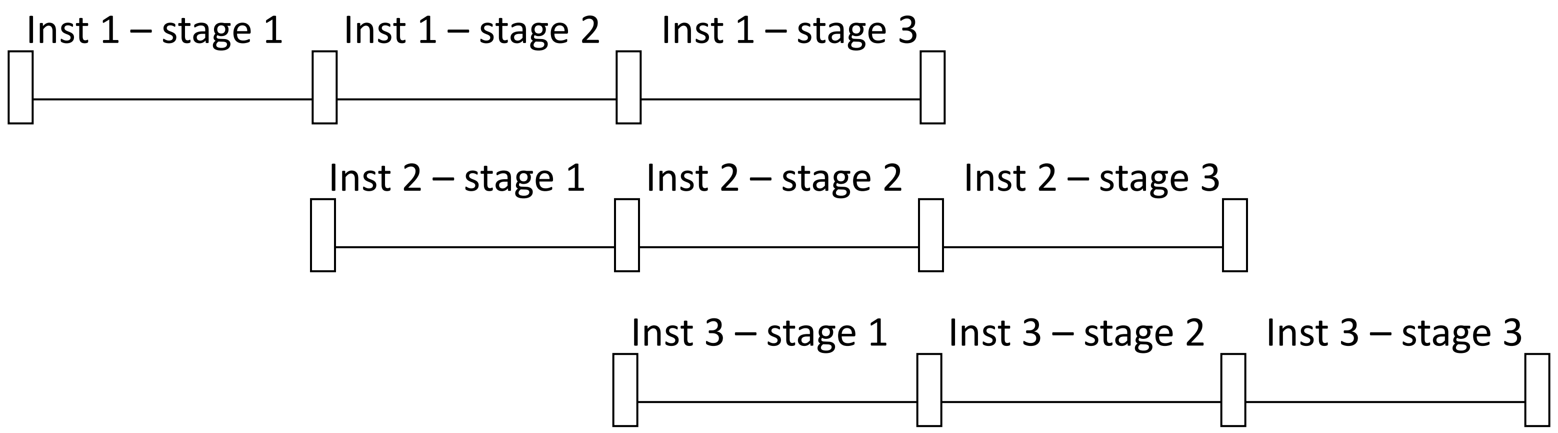
  - While early gates switch, later gates idle: inefficient.
- Divide the work into stages and add a register after each stage:



- Efficiency improves if each stage is always working

## Pipelining Basics

- Instructions advance through the stages in sequence
  - Instruction are "committed" as they leaves the last stage
- Each stage simultaneously works on different instructions
  - *n* pipline stages ⇒ *n* concurrent instructions!



## Ideal Pipelining

$$\text{Time per instruction} = \frac{\text{Time per instruction unpipelined}}{\#\ \text{pipeline stages}}$$

- Pipelining reduces either:
  - the average execution time per instruction, or
  - the number of cycles required for execution (CPI)
- Ideally, all stages have the same delay (balanced)
  - Cycle time is determined by the longest stage
- Ideally, throughput increases by *n* when employing *n* pipeline stages

# Pipelining is Not Ideal

*Reality: overheads and hazards result in trade-offs*

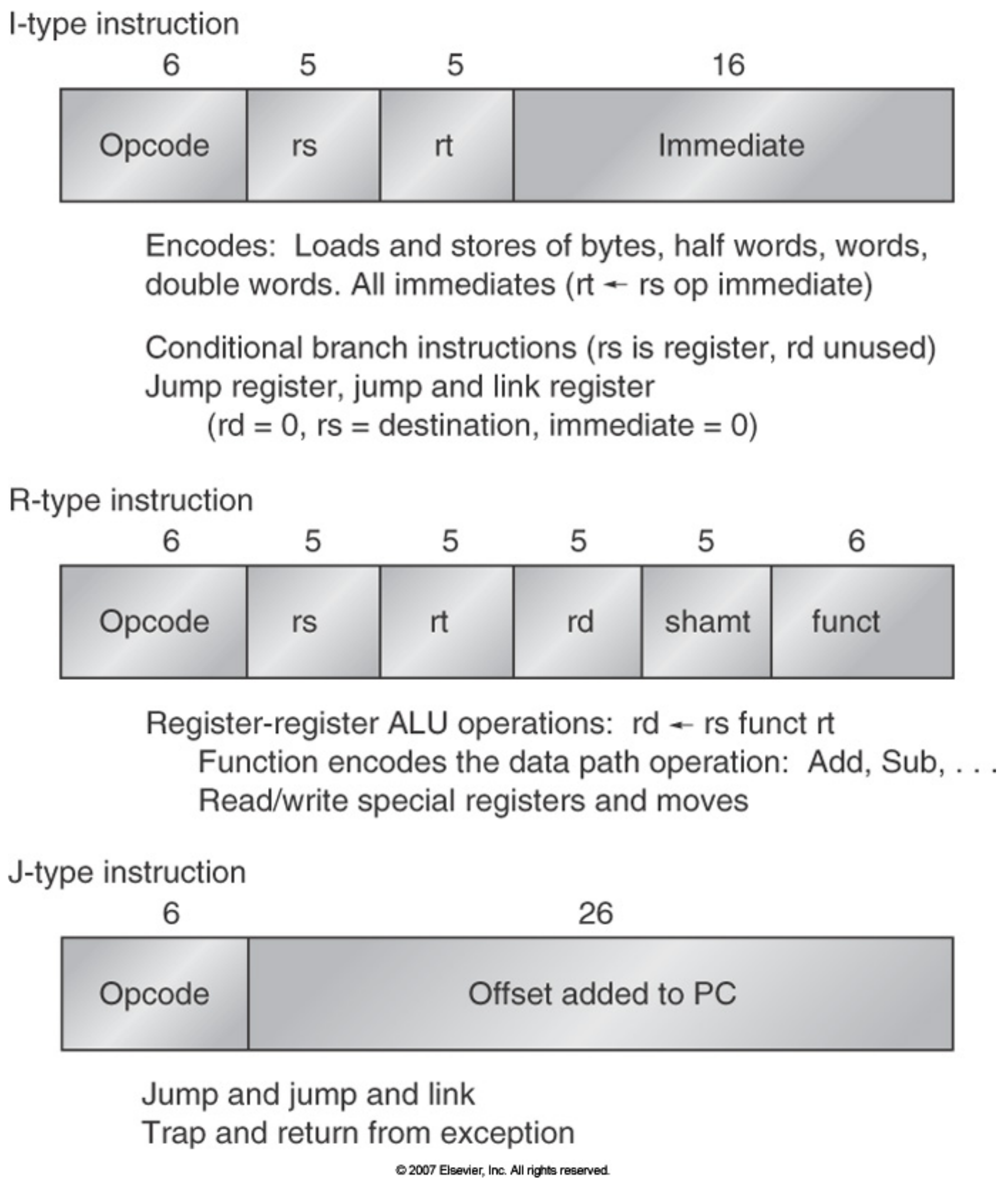
- New sources of overhead
  - Pipeline registers add delay
  - Pipeline stages can't be balanced perfectly
- Hazards
  - Structural: instructions may contend for resources
  - Data: instructions may depend on each other for inputs
  - Control: current instruction may determine the next
- Increased memory traffic
  - Fetch instructions
  - Load or store data

## Review: RISC Instruction Set

- Reduced Instruction Set Computing
  - Simple ISA designed for efficient pipelining
  - All operations on data modify registers
  - Only memory operations are loads and stores
  - Instructions typically have one size
- Three basic instruction classes
  - Load and store
  - ALU operations
  - Branch and jump

## Review: RISC Instruction Classes

- Load and store
  LD R1, 30(R0)
  LD R2, 100(R0)
  …
  ST R3, 200(R0)
- ALU operations
  ADD R3,R1,R2
- Branch and jump
  BNEZ R3, target

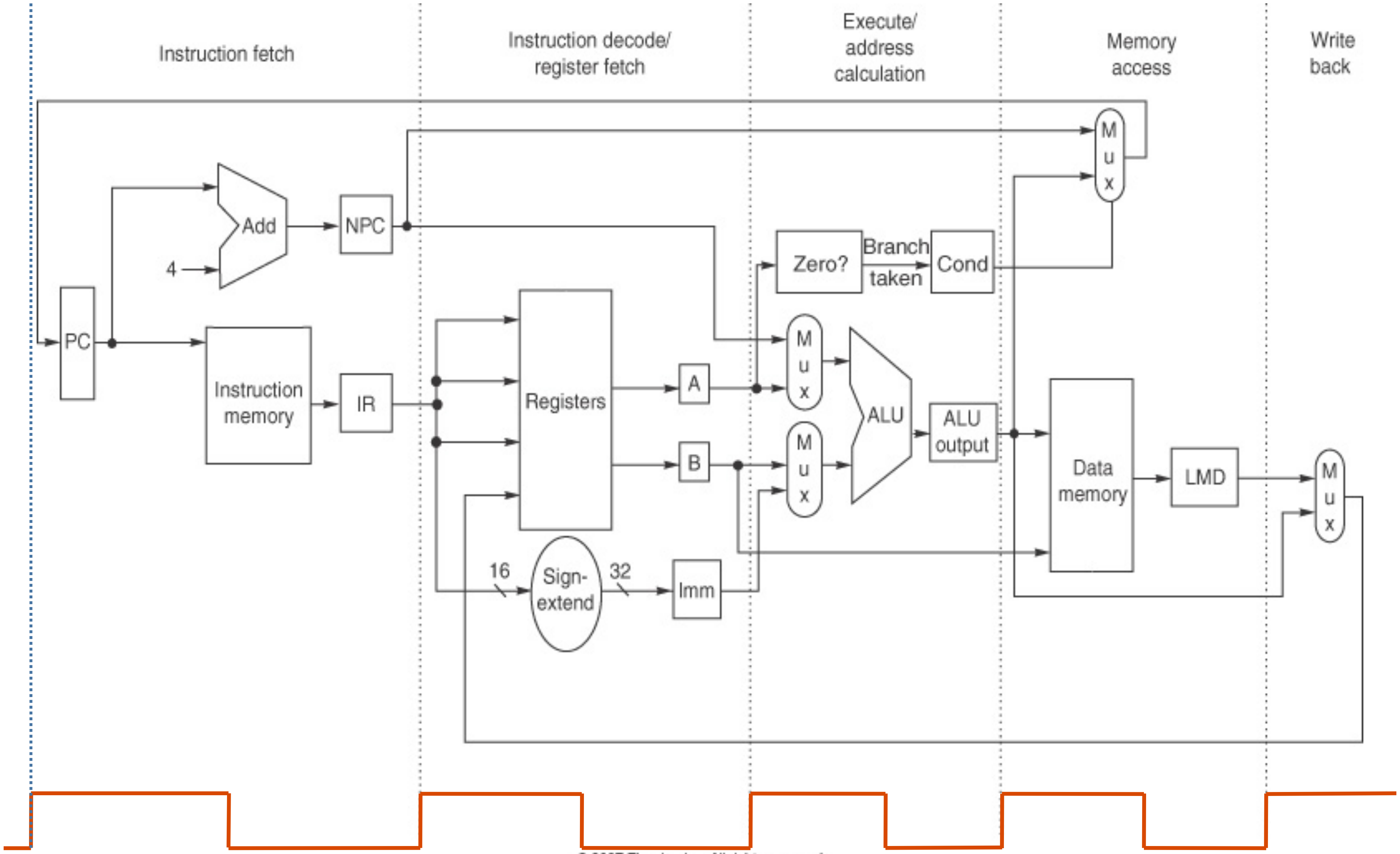


I-type instruction
6
5
5
16
Opcode
rs
rt
Immediate
Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)
Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)
R-type 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
© 2007 Elsevier, Inc. All rights reserved.

## For More Information

- We'll use MIPS RISC throughout the course
- See Appendix B for more information
  - Refer to Figures B.22-B.25 in particular

# Unpipelined RISC



# Unpipelined RISC: Instruction Fetch

| | | | | |
|---|---|---|---|---|
| ***IF*** | **ID** | **EX** | **MEM** | **WB** |

1. Instruction Fetch (IF)

$$IR \leftarrow Mem[PC];$$
$$PC \leftarrow PC + 4;$$

– Send PC to memory to fetch the current instruction

– Update PC

# Unpipelined RISC: Instruction Decode

| | | | | |
|---|---|---|---|---|
| IF | *ID* | EX | MEM | WB |

2. Instruction Decode / Register Fetch (ID)

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

- – Decode instruction and read registers
- – Sign-extend immediate value

## Unpipelined RISC: Execution

| | | | | |
|---|---|---|---|---|
| IF | ID | *EX* | MEM | WB |

3. Execution (EX)
   - ALU operates on the operands prepared in ID stage

- Memory op: form effective address
  - ALUOutput ← A + Imm;
- Reg-Reg ALU op:
  - ALUOutput ← A op B;
- Reg-Imm ALU op:
  - ALUOutput ←A op Imm;
- Branch:
  - ALUOutput ← NPC + (Imm << 2);
  - Cond ← (A == 0)

# Unpipelined RISC: Memory Access

| | | | | |
|---|---|---|---|---|
| **IF** | **ID** | **EX** | ***MEM*** | **WB** |

4. Memory Access (MEM)

```
PC ← NPC;
Load:
    LMD ← Mem[ALUOutput];
Store:
    Mem[ALUOutput] ← B;
Branch:
    If (cond) PC ← ALUOutput;
```

- Load: read from the effective address in memory
- Store: write register value to the effective address
- Branch: update PC if the condition bit is set

## Unpipelined RISC: Write-back

| IF | ID | EX | MEM | *WB* |
|---|---|---|---|---|

5. Write-back (WB)

   Reg-Reg ALU:

   Regs[rd] ← ALUOutput;

   Reg-Imm ALU:

   Regs[rt] ← ALUOutput;

   Load:

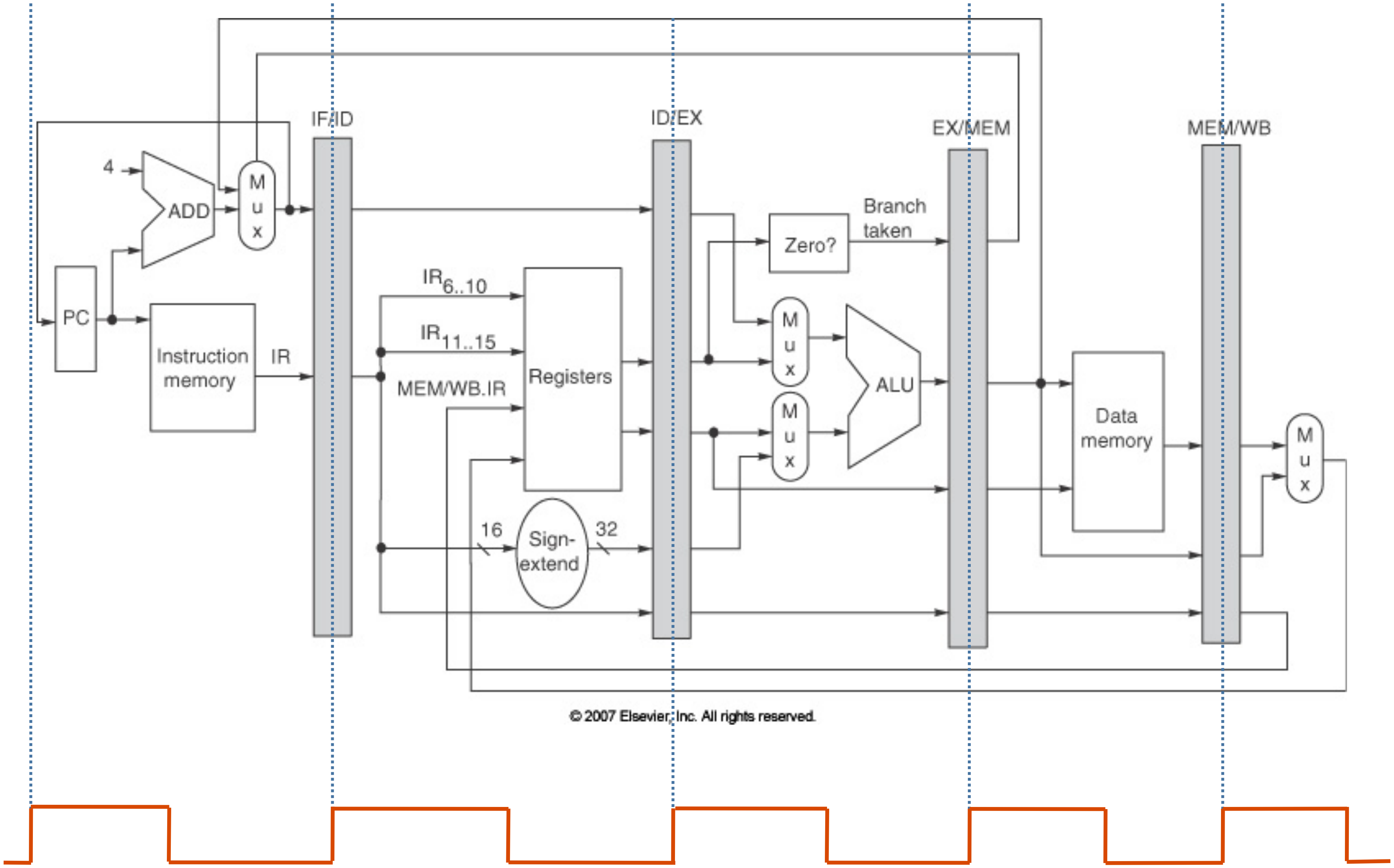
   Regs[rt] ← LMD;

   – Reg-X ALU or Load: write the result into the register file

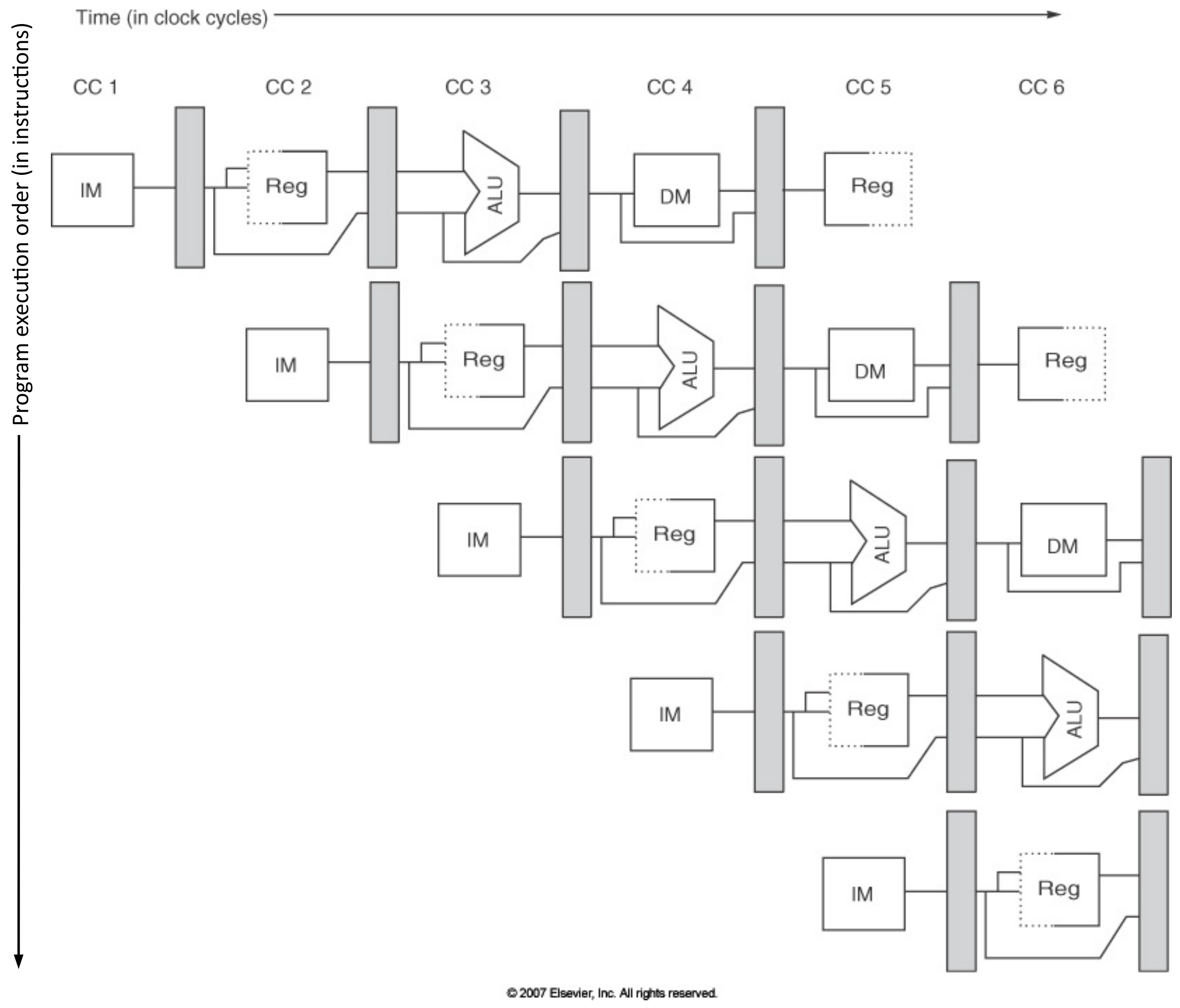
## Unpipelined RISC: Summary

- Execution times without pipelining
  - Branches and stores: 4 cycles
  - Others: 5 cycles
- Typical instruction mix
  - Branches and stores 22%
  - Others: 78%
- What is the CPI of this unpipelined RISC processor?

# Basic MIPS RISC Pipeline



# Pipelining: Many Data Paths in One



Time (in clock cycles)
Program execution order (in instructions)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
IM
Reg
ALU
DM
Reg
© 2007 Elsevier, Inc. All rights reserved.

## Supporting Pipelining

- Pipelining requires more memory bandwidth
  - Simultaneously fetch instructions (IF), access data (MEM)
  - Cache instructions and data in separate memories
- Pipelining requires more register file bandwidth
  - Simultaneously read (ID) and write (WB) registers
  - Write in the first half CC, read in the second half
- Pipelining requires extra registers to store intermediate results
  - Additional state requires additional power, area, etc.

## Summary

- Ideal pipelining: divide work into *n* stages to increase throughput by *n* times!
- RISC Instruction Set
  - Small set of simple operations
  - Ideal for applying pipelining
- Unpipelined RISC implementation
  - IF, ID, EX, MEM, WB
- Ideal pipelining requires more
  - Memory bandwidth
  - Register file bandwidth
  - Architectural state

## Next Time

- Basic pipeline performance issues
- Pipeline hazards
  - Structural
  - Data
  - Control
- Hazard mitigation