CMCS 611-101
Advanced Computer Architecture

Lecture 6

Introduction to Pipelining

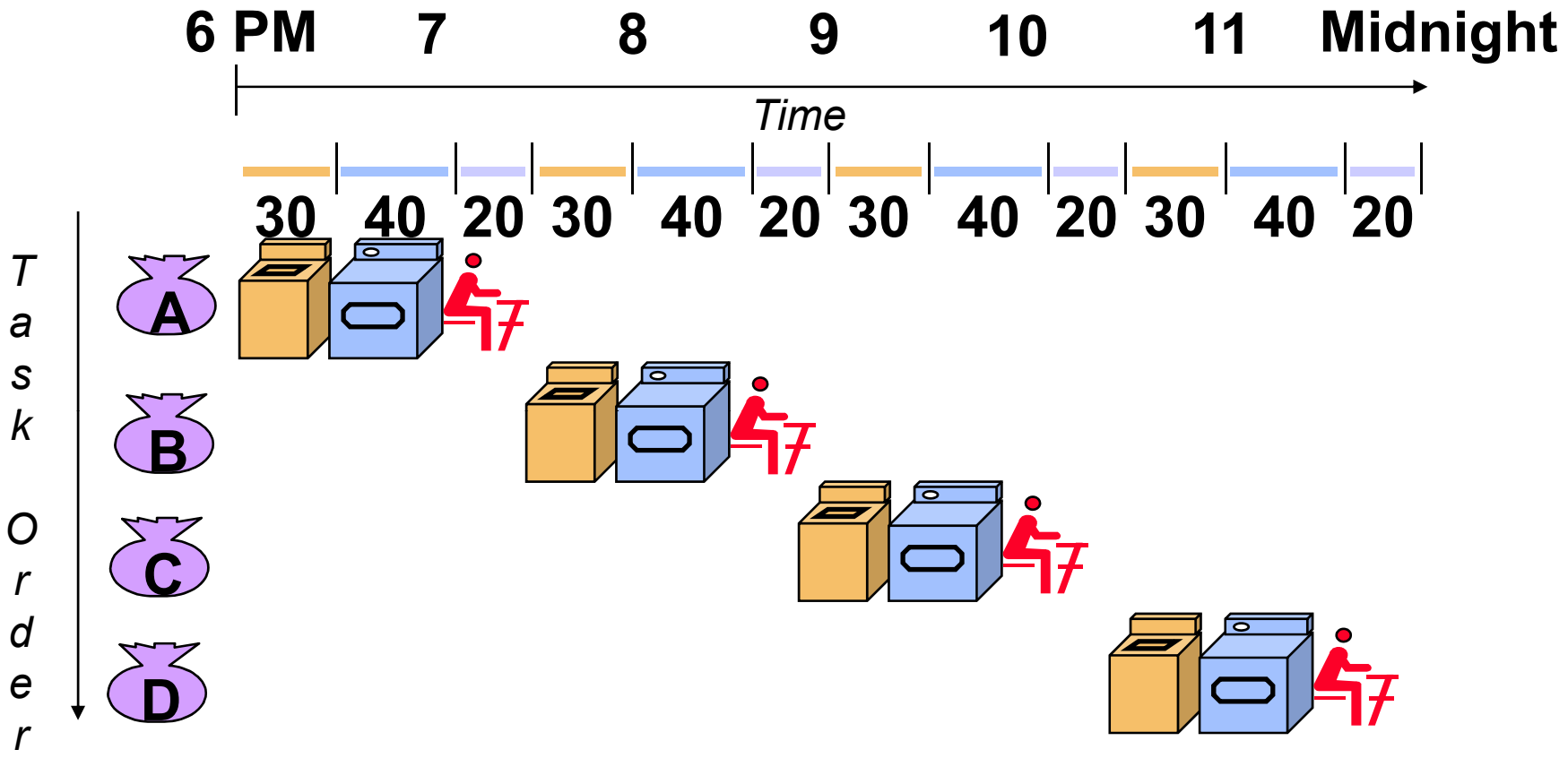
September 23, 2009

www.csee.umbc.edu/~younis/CMSC611/CMSC611.htm

Lecture's Overview

- ***Previous Lecture:***
  - ➔ Type and size of operands
    (Famous data types, effect of operand size on design complexity)
  - ➔ Encoding the instruction set
    (Fixed, variable and hybrid encoding, the store program concept)
  - ➔ The role of the compiler
    (Compilation process, compiler optimization, linking and loading)
  - ➔ Effect of ISA on Compiler Complexity
    (Regularity, Primitives, not solutions, Simplify trade-offs, Static binding)
- ***This Lecture:***
  - An overview of pipelining
  - Pipeline performance
  - Pipelined hazards

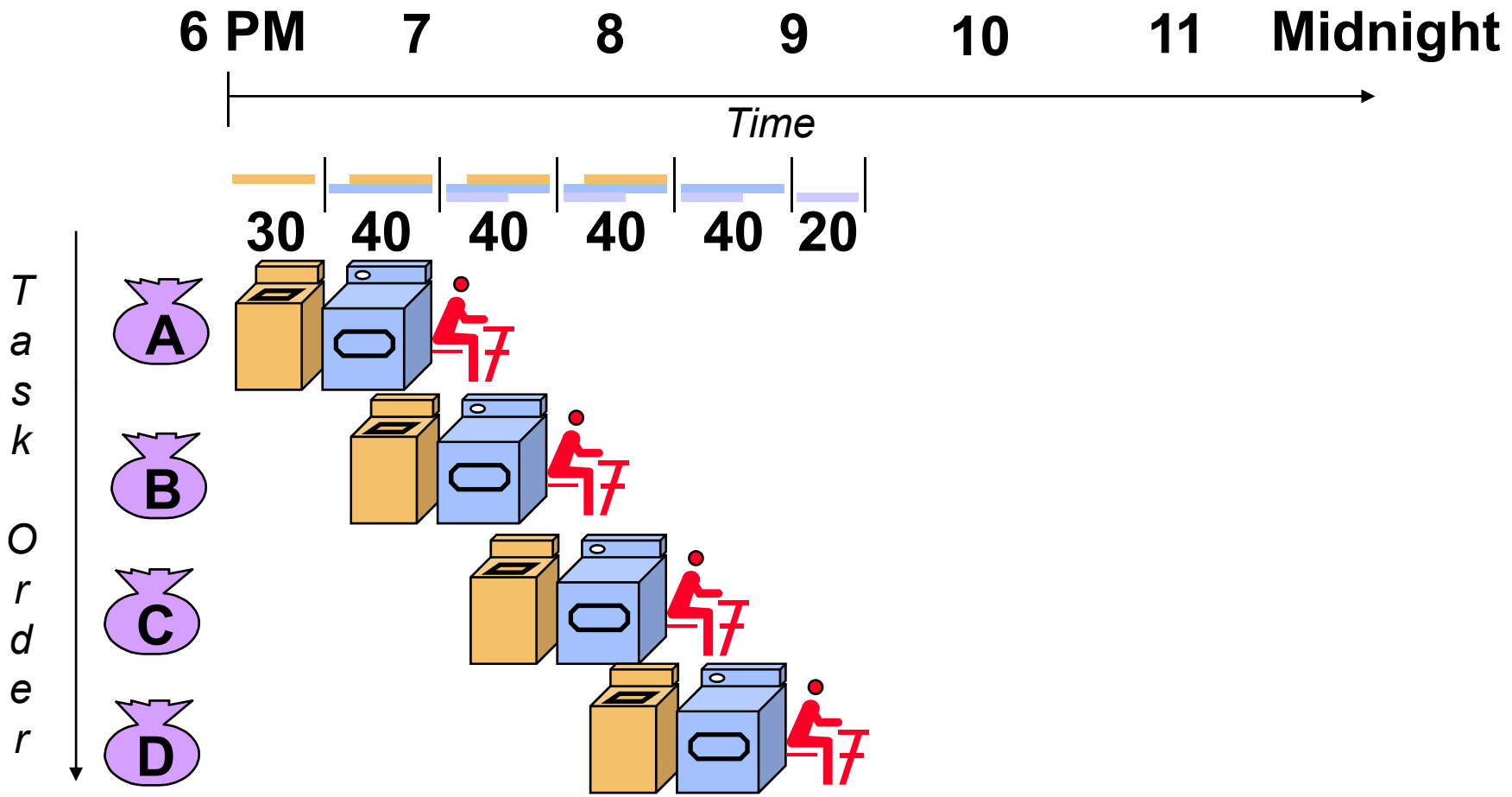
Sequential Laundry



- Washer takes 30 min, Dryer takes 40 min, folding takes 20 min
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

* Slide is courtesy of Dave Patterson

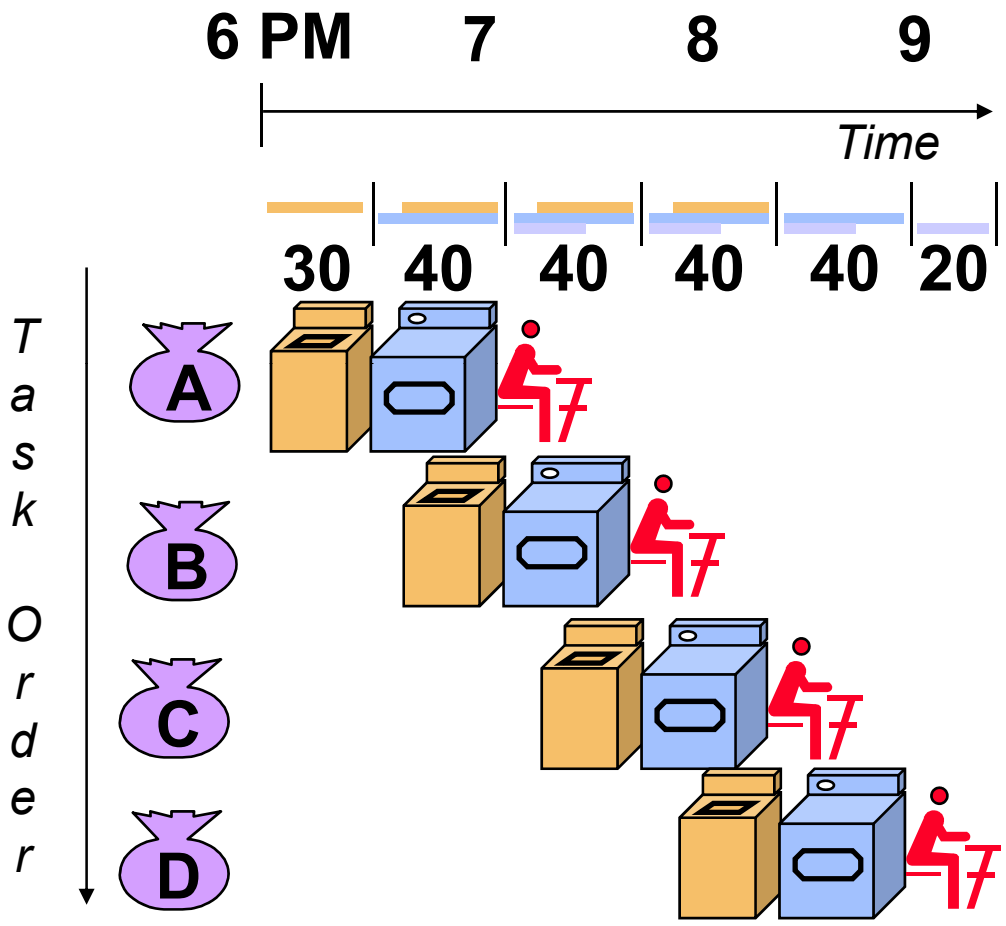
Pipelined Laundry



- Pipelining means start work as soon as possible
- Pipelined laundry takes 3.5 hours for 4 loads

* Slide is courtesy of Dave Patterson

Pipelining Lessons



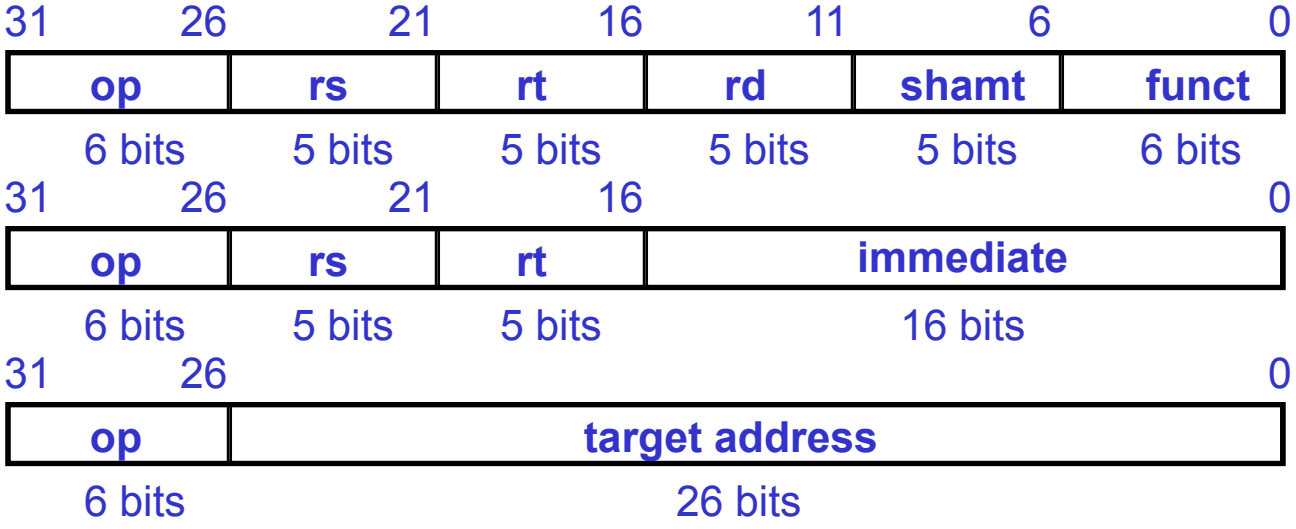
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduce speedup
- Stall for Dependencies

* Slide is courtesy of Dave Patterson

Basics of a RISC Instruction Set

- RISC architectures are characterized by the following features that dramatically simplifies the implementation:
  1. All ALU operations apply only on data in registers
  2. Memory is affected only by load and store operations
  3. Instructions follow very few formats and typically are of the same size

- All MIPS instructions are 32 bits, following one of three formats:



* Slide is courtesy of Dave Patterson

MIPS Instruction format

❶ Register-format instructions:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

op: Basic operation of the instruction, traditionally called opcode
rs: The first register source operand
rt: The second register source operand
rd: The register destination operand, it gets the result of the operation
shmat: Shift amount (explained in future lectures)
funct: This field selects the specific variant of the operation of the op field

- MIPS assembly language includes two conditional branching instructions using PC -relative addressing:

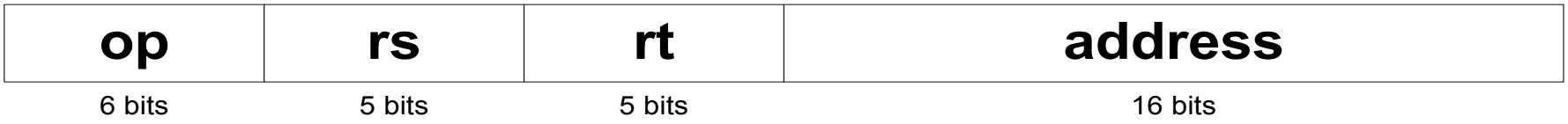
```
beq register1, register2, L1   # go to L1 if (register1) = (register2)
bne register1, register2, L1   # go to L1 if (register1) ≠ (register2)
```

- Examples:

```
add    $t2, $ t1, $ t1    # Temp reg $t2 = 2 $t1
sub    $t1, $s3, $s4      # Temp reg $t1 = $s3 - $s4
and    $t1, $ t2, $ t3    # Temp reg $t1 = $t2 . $t
bne    $s3, $s4, Else     # if $s3 ≠ $s4 jump to Else
```

MIPS Instruction format

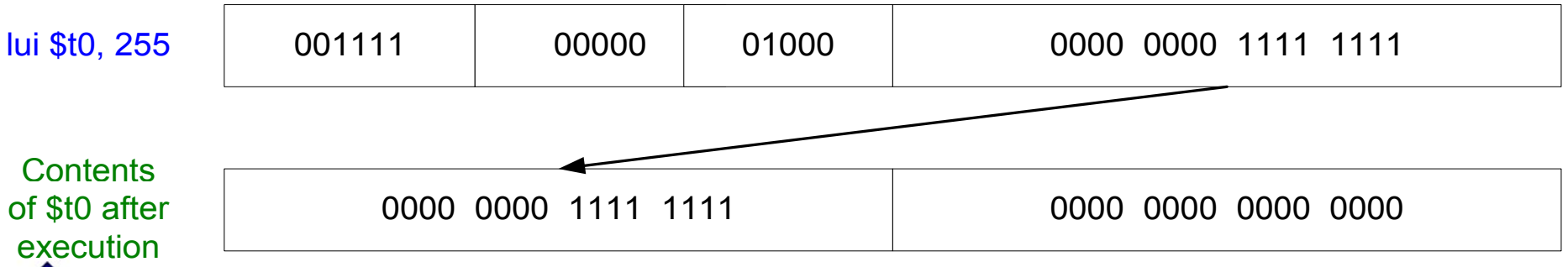
❷ Immediate-type instructions:



- The 16-bit address means a load word instruction can load a word within a region of $\pm 2^{15}$ bytes of the address in the base register
- Examples: lw $t0, 32($s3) , sw $t1, 128($s3)
- MIPS handle 16-bit constant efficiently by including the constant value in the address field of an I-type instruction (Immediate-type)

```
addi  $sp, $sp, 4   #$sp = $sp + 4
```

- For large constants that need more than 16 bits, a load upper-immediate (*lui*) instruction is used to concatenate the second part



Addressing in Branches & Jumps

- I-type instructions leaves only 16 bits for address reference limiting the size of the jump
- MIPS branch instructions use the address as an increment to the PC allowing the program to be as large as 2^{32} (called *PC-relative addressing*)
- Since the program counter gets incremented prior to instruction execution, the branch address is actually relative to (PC + 4)
- MIPS also supports an J-type instruction format for large jump instructions

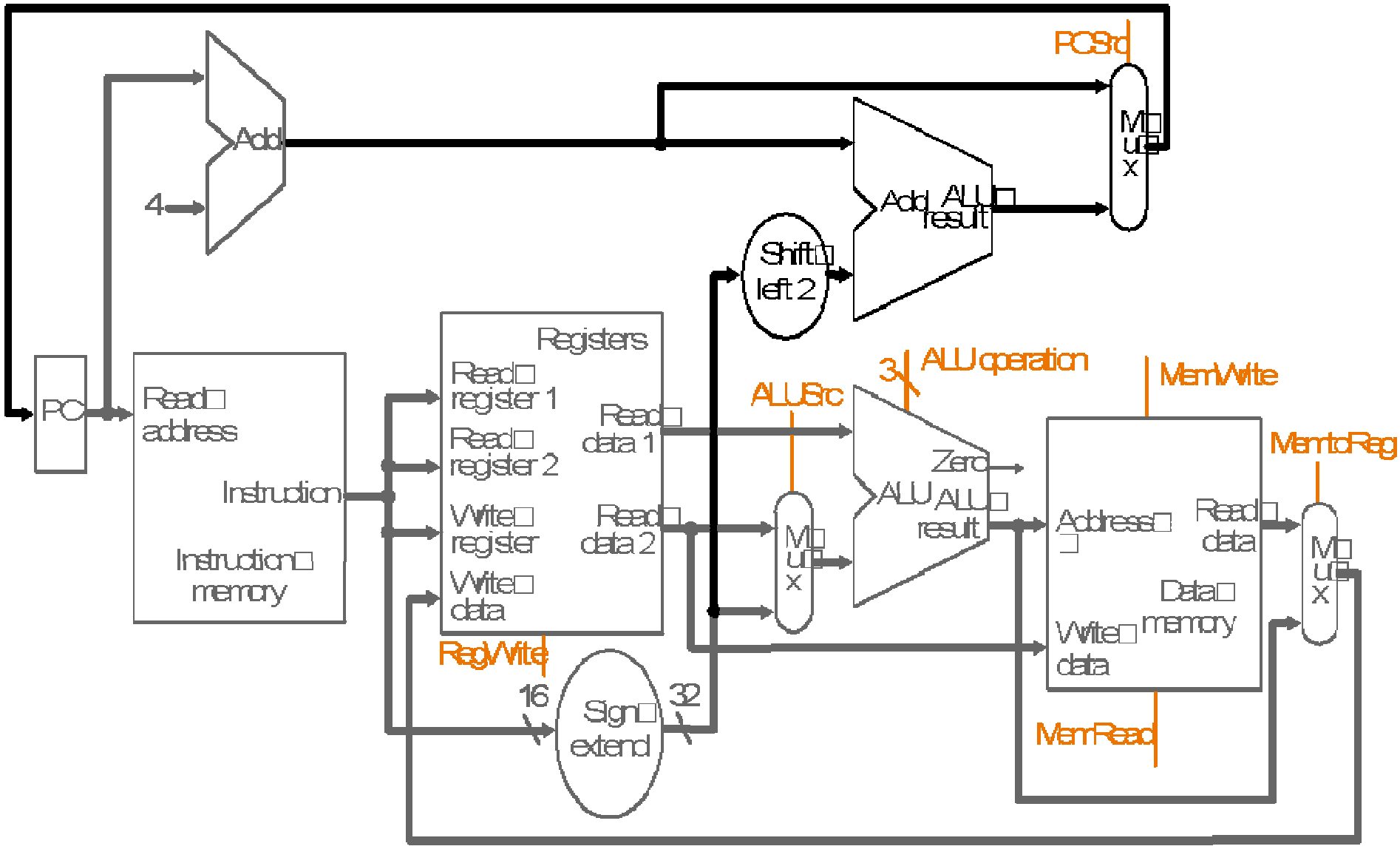
op	address
6 bits	26 bits

- The 26-bit address in a J-type instruct. is concatenated to upper 8 bits of PC

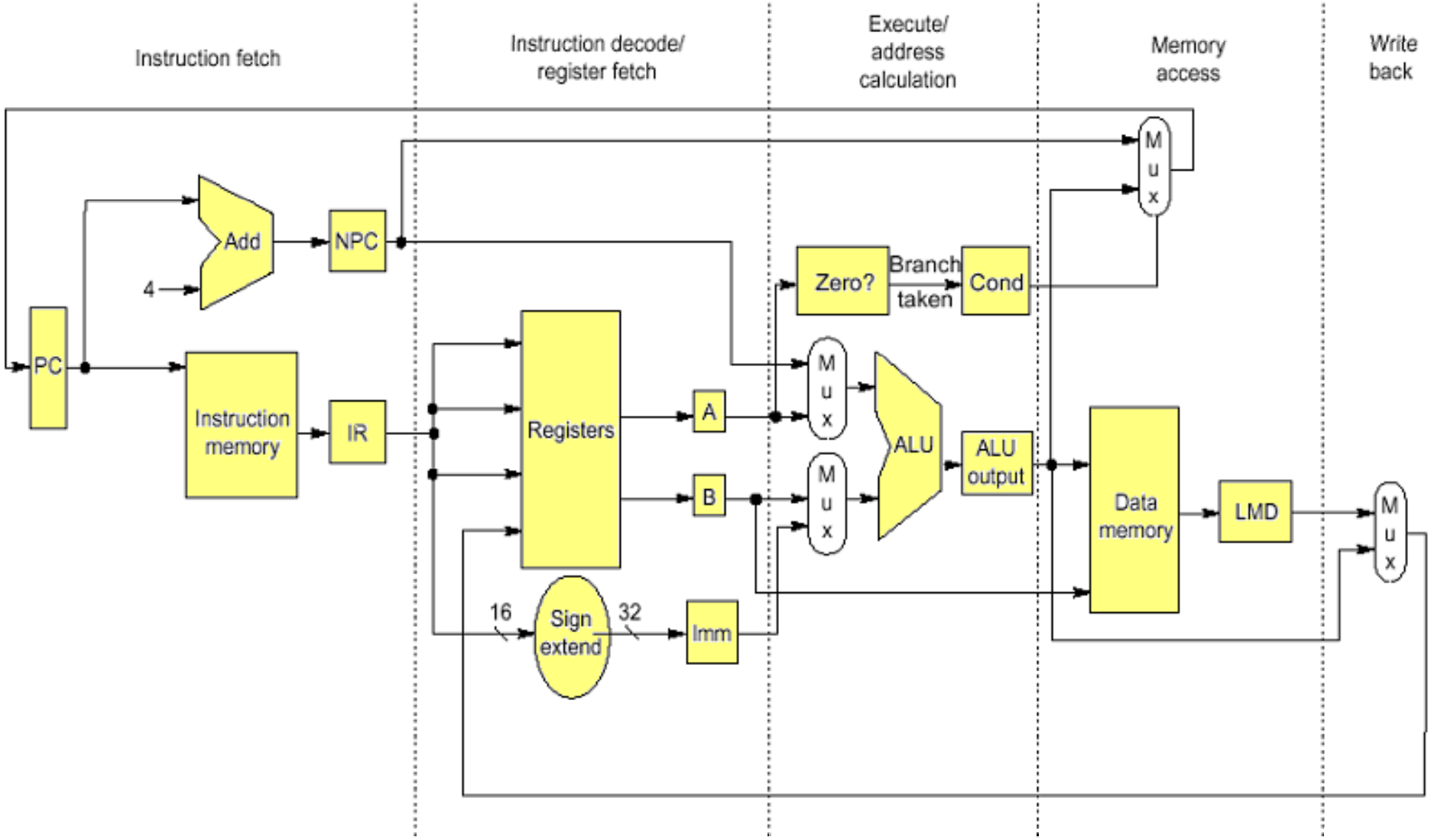
```
Loop:   add   $t1, $s3, $s3
        add   $t1, $t1, $t1
        add   $t1, $t1, $s6
        lw    $t0, 0($t1)
        bne   $t0, $s5, Exit
        add   $s3, $s3, $s4
        j     Loop
Exit:
```

80000	0	19	19	9	0	32
80004	0	9	9	9	0	32
80008	0	9	22	9	0	32
80012	35	9	8	0		
80016	5	8	21	8		
80020	0	19	20	19	0	32
80024	2	80000				
80028	. . .					
80012	35	9	8	0		

A Simple Implementation of MIPS



Single-cycle Instruction Execution



❶

Multi-Cycle Implementation of MIPS

❶ **Instruction fetch cycle (IF)**

IR ← Mem[PC]; NPC ← PC + 4

❷ **Instruction decode/register fetch cycle (ID)**

A ← Regs[$IR_{6..10}$]; B ← Regs[$IR_{11..15}$]; Imm ← ((IR_{16})^{16} ##$IR_{16..31}$)

❸ **Execution/effective address cycle (EX)**

Memory ref: ALUOutput ← A + Imm;

Reg-Reg ALU: ALUOutput ← A *func* B;

Reg-Imm ALU: ALUOutput ← A *op* Imm;

Branch: ALUOutput ← NPC + Imm; Cond ← (A op 0)

❹ **Memory access/branch completion cycle (MEM)**

Memory ref: LMD ← Mem[ALUOutput] or Mem(ALUOutput] ← B;

Branch: if (cond) PC ←ALUOutput;

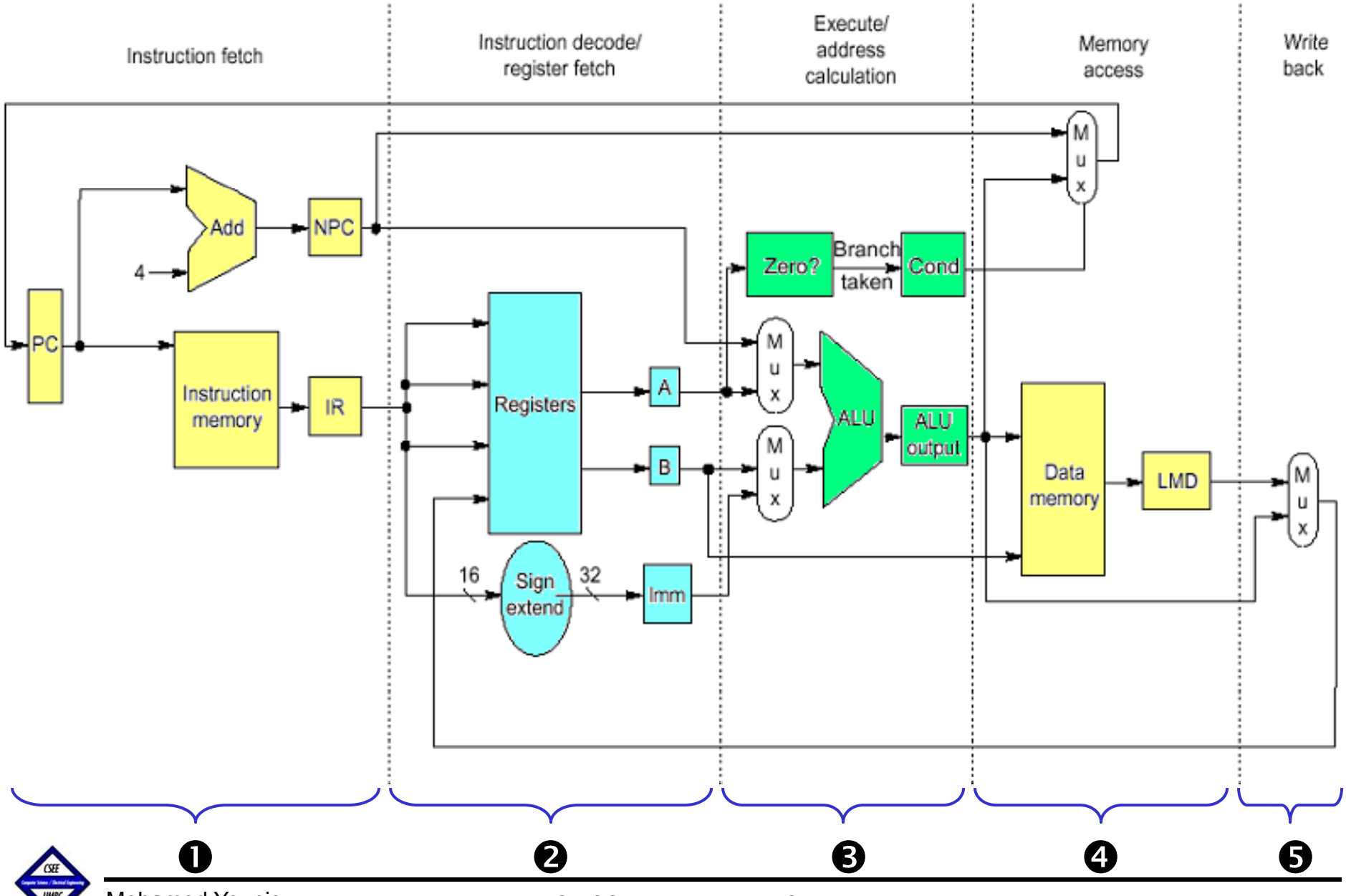
❺ **Write-back cycle (WB)**

Reg-Reg ALU: Regs[$IR_{16..20}$] ← ALUOutput;

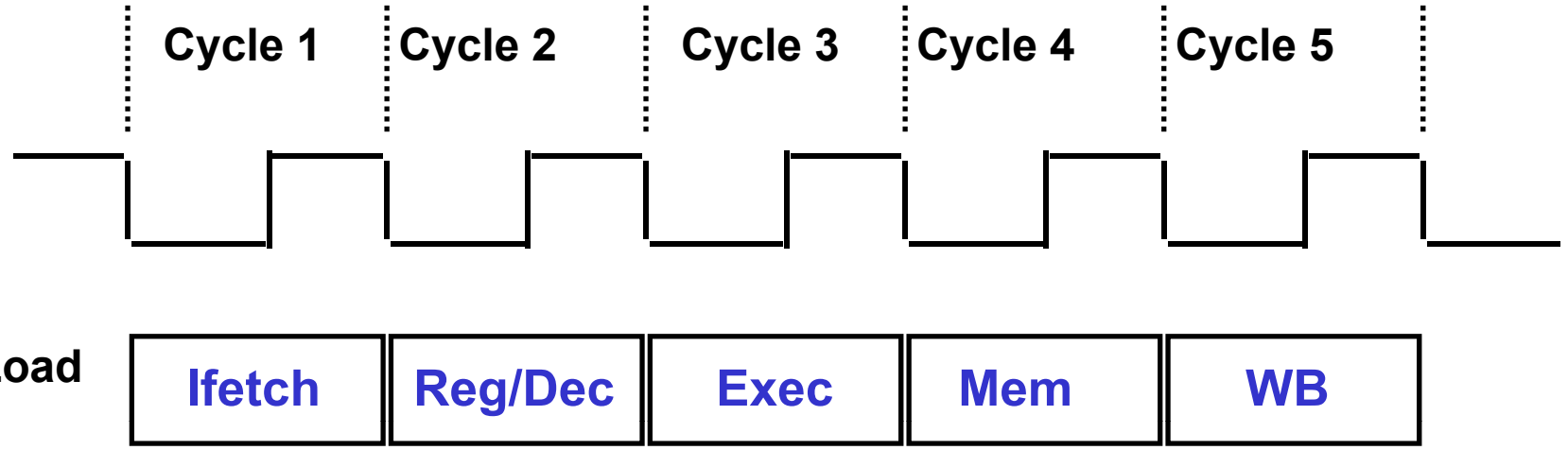
Reg-Imm ALU: Regs[$IR_{11..15}$] ← ALUOutput;

Load: Regs[$IR_{11..15}$] ← LMD;

Multi-cycle Instruction Execution



Stages of Instruction Execution

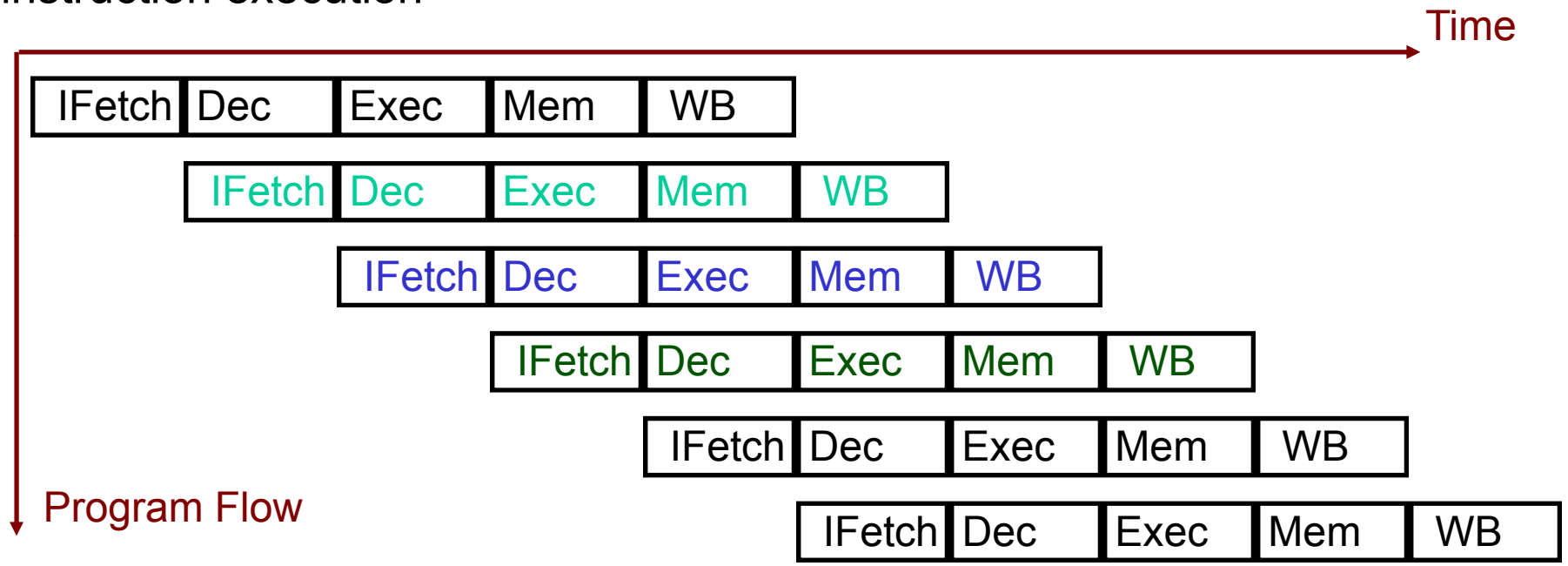


- The load instruction is the longest
- All instructions follows at most the following five steps:
  - ➔ Ifetch: Instruction Fetch
    - Fetch the instruction from the Instruction Memory and update PC
  - ➔ Reg/Dec: Registers Fetch and Instruction Decode
  - ➔ Exec: Calculate the memory address
  - ➔ Mem: Read the data from the Data Memory
  - ➔ WB: Write the data back to the register file

* Slide is courtesy of Dave Patterson

Instruction Pipelining

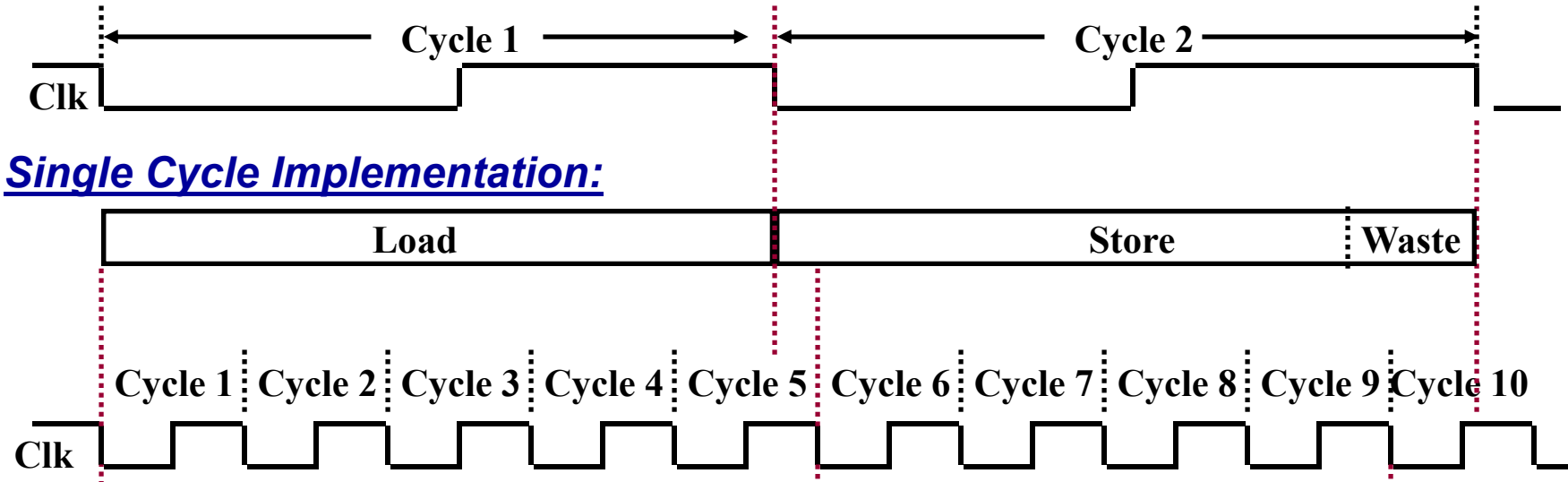
- Start handling of next instruction while the current instruction is in progress
- Pipelining is feasible when different devices are used at different stages of instruction execution



$$\text{Time between instructions}_{pipelined} = \frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of pipe stages}}$$

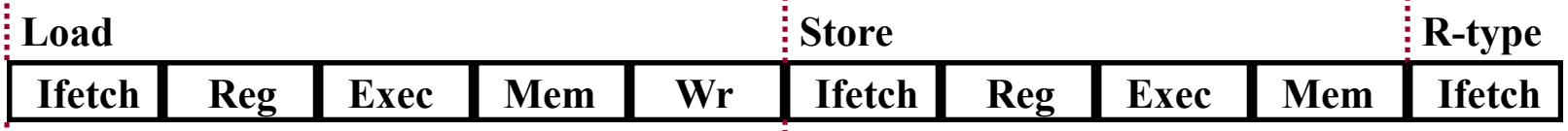
Pipelining improves performance by increasing instruction throughput

Single Cycle, Multiple Cycle, vs. Pipeline

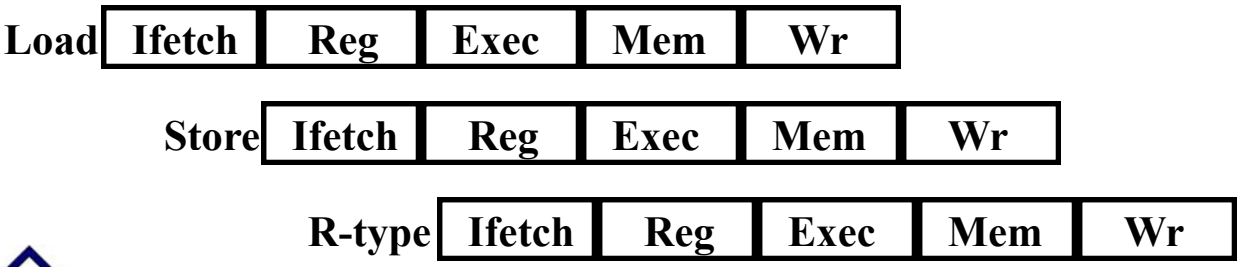


Single Cycle Implementation:

Multiple Cycle Implementation:

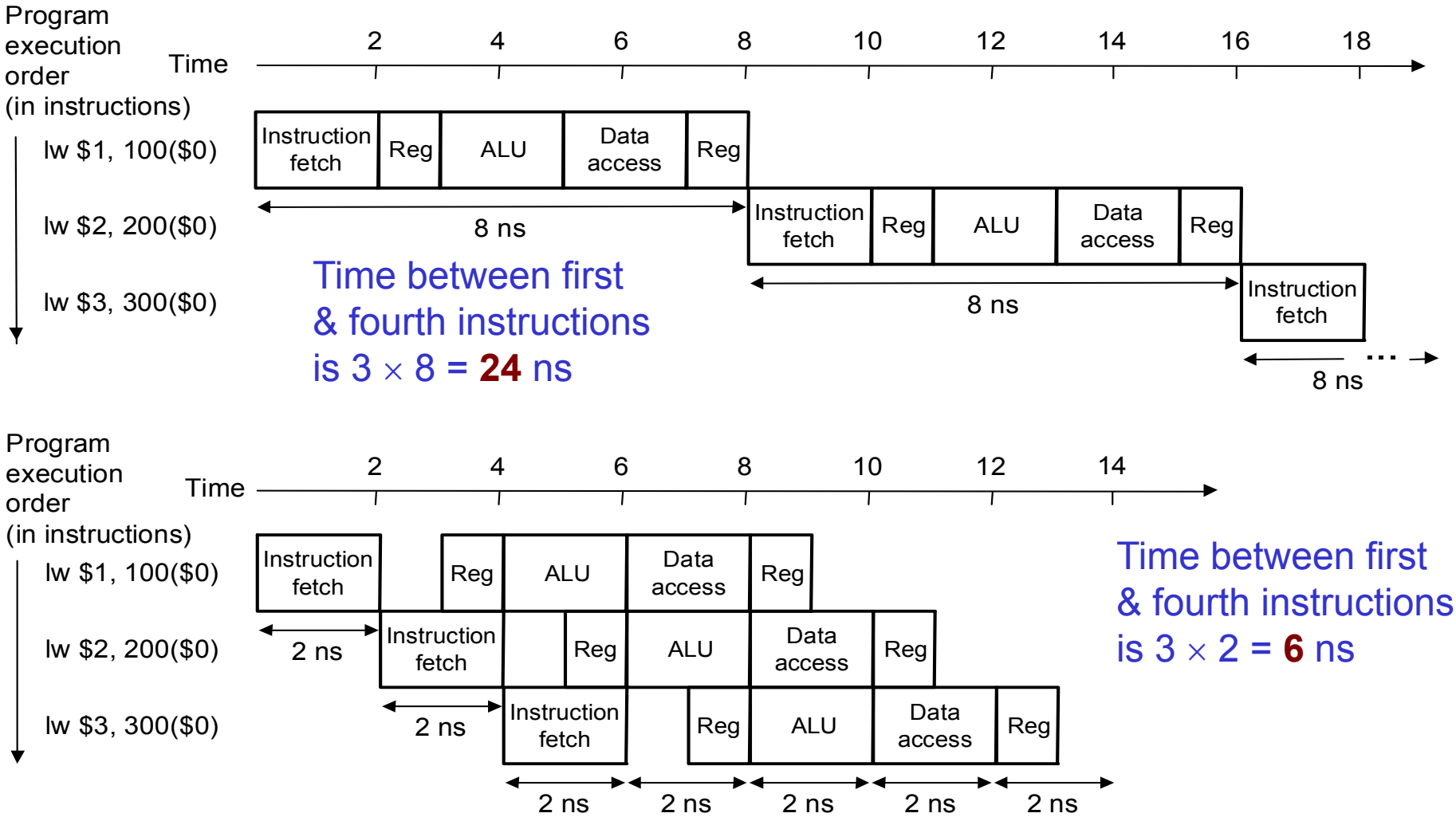


Pipeline Implementation:



* Slide is courtesy of Dave Patterson

Example of Instruction Pipelining



Ideal and upper bound for speedup is number of stages in the pipeline

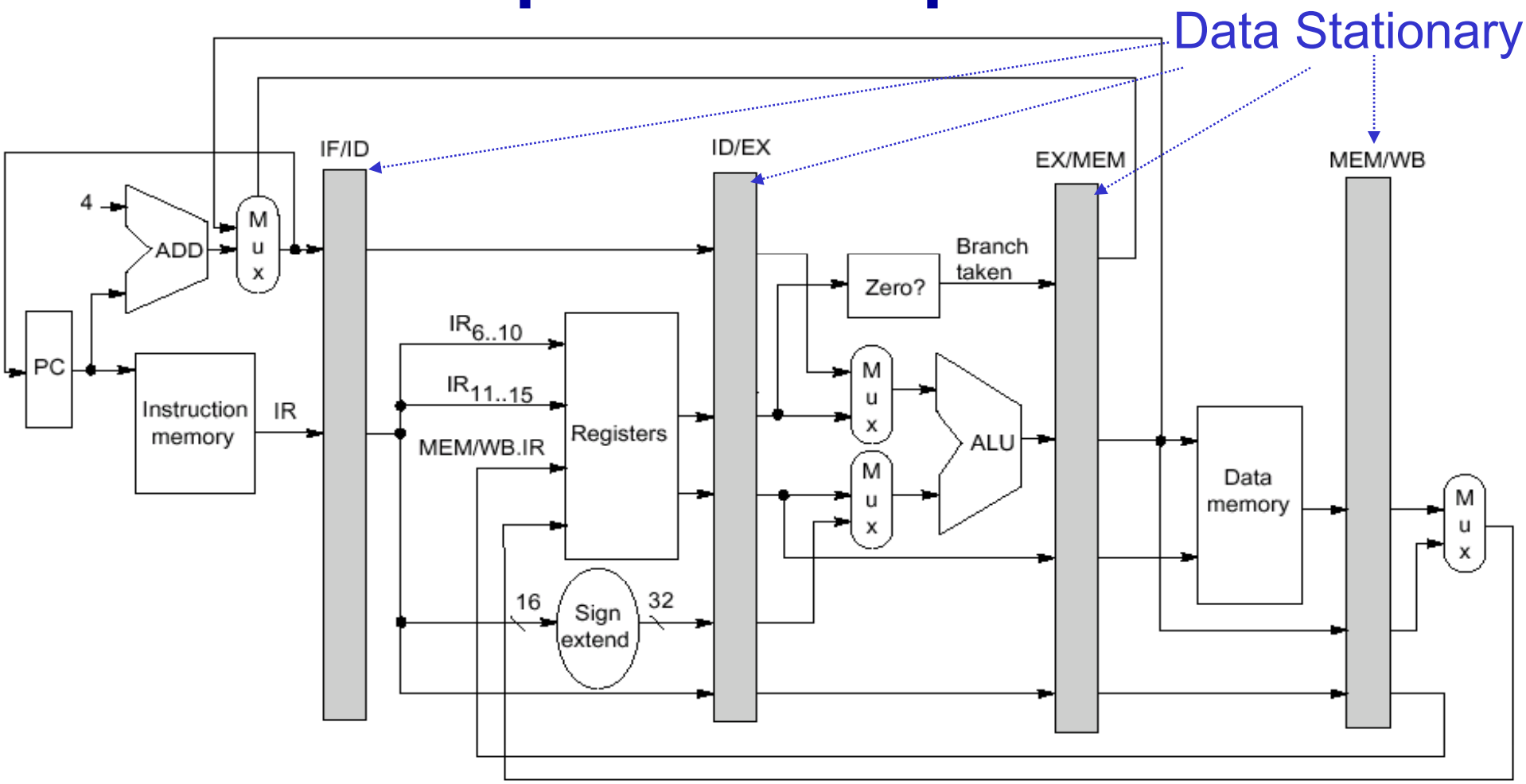
Pipeline Performance

- Pipeline increases the instruction throughput but does not reduce the execution time of the individual instruction
- Execution time of the individual instruction in pipeline can be slower due:
  - ➔ Additional pipeline control compared to none pipeline execution
  - ➔ Imbalance among the different pipeline stages
- Suppose we execute 100 instructions:
  - ➔ Single Cycle Machine
    - 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
  - ➔ Multi-cycle Machine
    - 10 ns/cycle x 4.2 CPI (due to inst mix) x 100 inst = 4200 ns
  - ➔ Ideal 5 stages pipelined machine
    - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns
- Due to fill and drain effects of a pipeline ideal performance can be achieved only for very large instructions

Example: a sequence of 1000 load instructions would take 5000 cycles on a multi-cycle machine while taking 1004 on a pipeline machine

$\Rightarrow$ speedup = 5000/1004 $\cong$ 5

Pipeline Datapath



- Every stage must be completed in one clock cycle to avoid stalls
- Values must be latched to ensure correct execution of instructions
- The PC multiplexer has moved to the IF stage to prevent two instructions from updating the PC simultaneously (in case of branch instruction)

Pipeline Stage Interface

Stage	Any Instruction		
IF	IF/ID.IR ← MEM[PC] ; IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) & EX/MEM.cond) {EX/MEM.ALUOutput } else { PC + 4 }) ;		
ID	ID/EX.A = Regs[IF/ID. IR $_{6..10}$]; ID/EX.B ← Regs[IF/ID. IR $_{11..15}$]; ID/EX.NPC ← IF/ID.NPC ; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← (IF/ID. IR $_{16}$) ^{16} ## IF/ID. IR $_{16..31}$;		
	ALU	***Load or Store***	***Branch***
EX	EX/MEM.IR = ID/EX.IR; EX/MEM. ALUOutput ← ID/EX.A func ID/EX.B; Or EX/MEM.ALUOutput ← ID/EX.A op ID/EX.Imm; EX/MEM.cond ← 0;	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.cond ← 0; EX/MEM.B ← ID/EX.B;	EX/MEM.ALUOutput ← ID/EX.NPC + ID/EX.Imm; EX/MEM.cond ← (ID/EX.A op 0);
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput] ; Or Mem[EX/MEM.ALUOutput] ← EX/MEM.B ;	
WB	Regs[MEM/WB. IR $_{16..20}$] ← EM/WB.ALUOutput; Or Regs[MEM/WB. IR $_{11..15}$] ← MEM/WB.ALUOutput ;	For load only: Regs[MEM/WB. IR $_{11..15}$] ← MEM/WB.LMD;	

Pipeline Hazards

- Pipeline hazards are cases that affect instruction execution semantics and thus need to be detected and corrected
- Hazards types

Structural hazard: attempt to use a resource two different ways at same time

➔ E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)

➔ Single memory for instruction and data

Data hazard: attempt to use item before it is ready

➔ E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer

➔ instruction depends on result of prior instruction still in the pipeline

Control hazard: attempt to make a decision before condition is evaluated

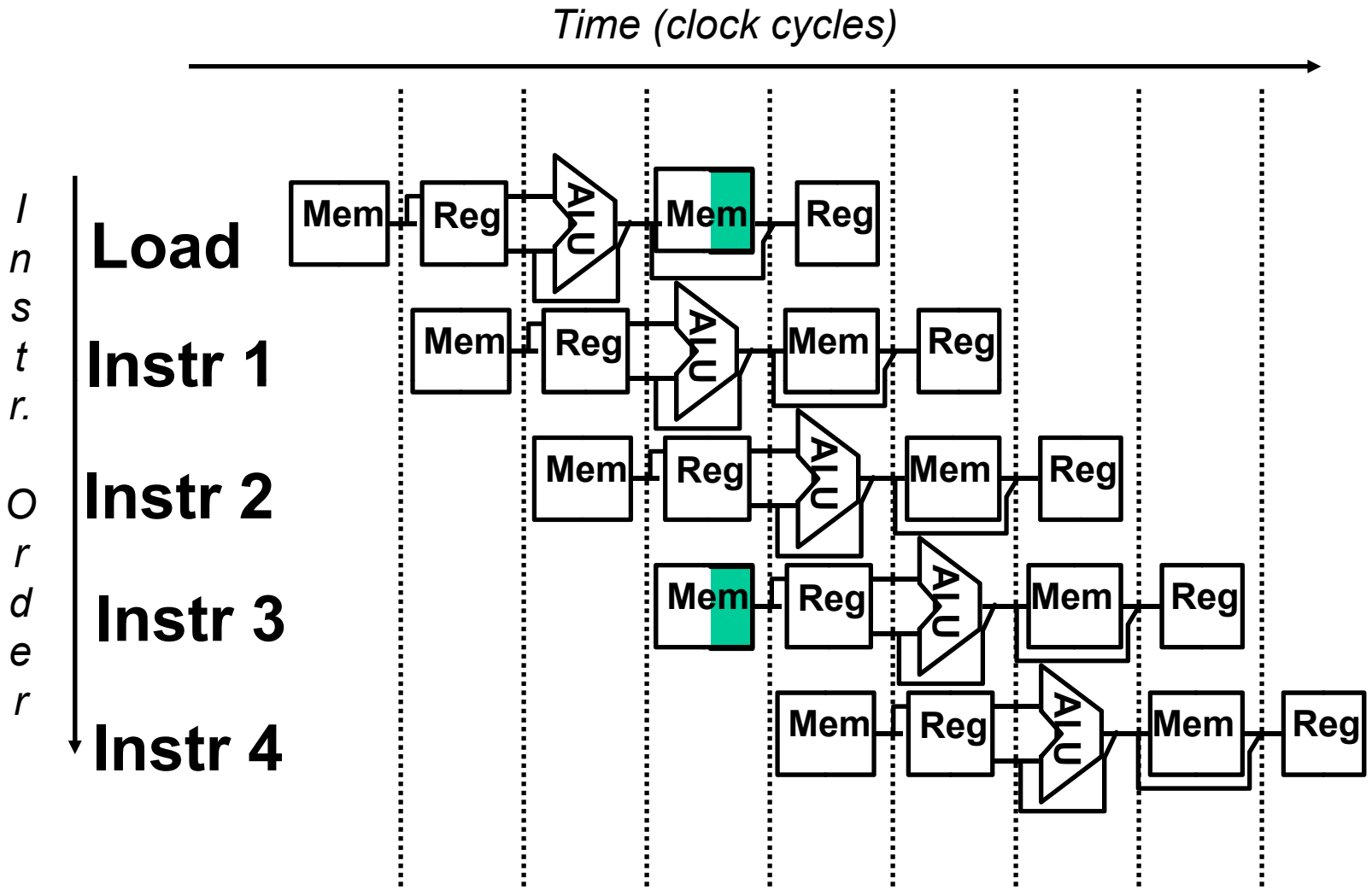
➔ E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in

➔ branch instructions

- Hazards can always be resolved by waiting

* Slide is courtesy of Dave Patterson

Single Memory is a Structural Hazard



- Can be easily detected
- Resolved by inserting idle cycles

* Slide is courtesy of Dave Patterson

Stalls & Pipeline Performance

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

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

- Ideally the CPI of the pipeline execution is 1 (after fill-up), thus

➔ CPI pipelined = Ideal CPI + Pipeline stall clock per instruction

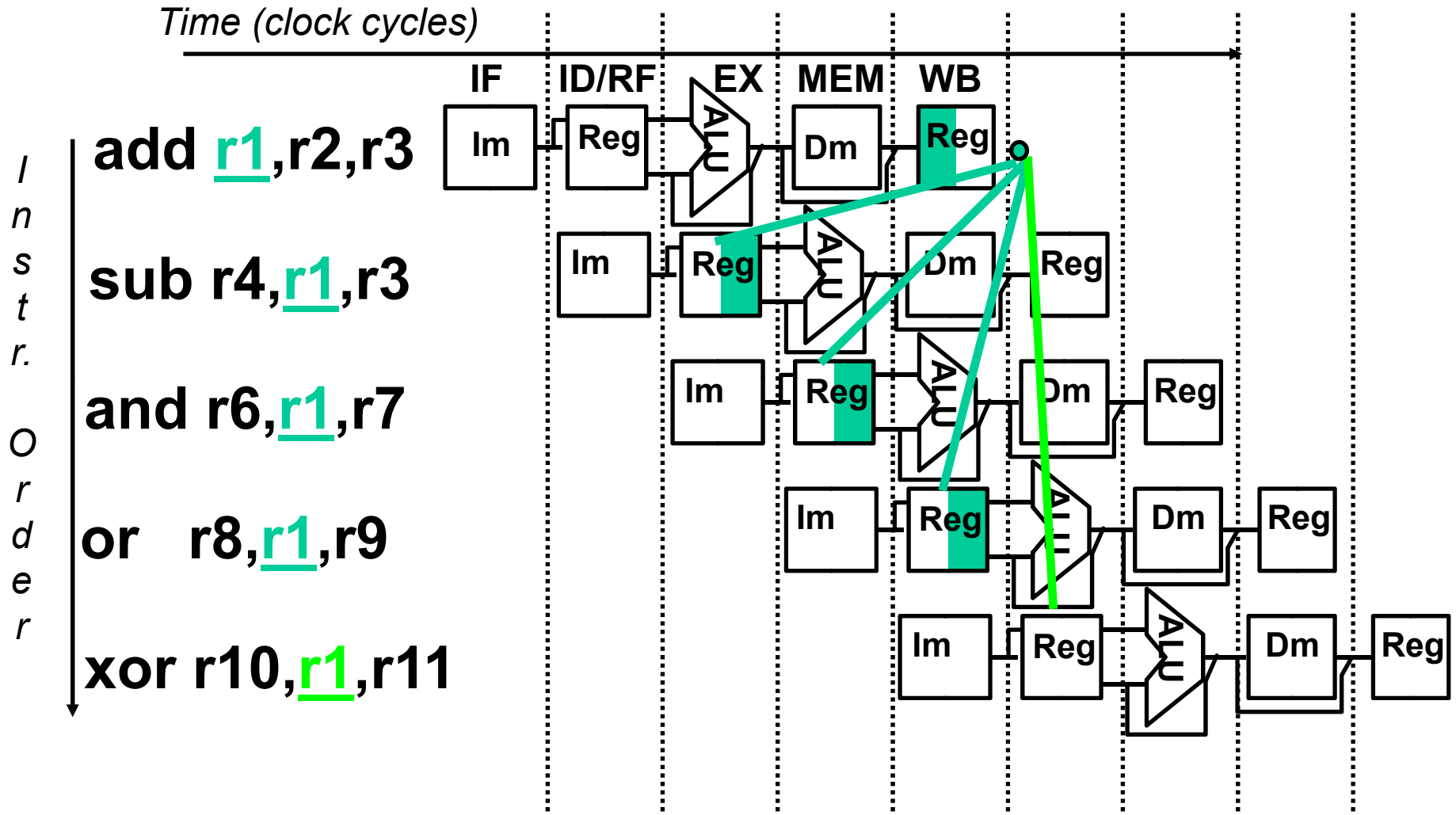
= 1 + Pipeline stall clock per instruction

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

- Assuming all pipeline stages are balanced, then

$$\text{Speedup} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}$$

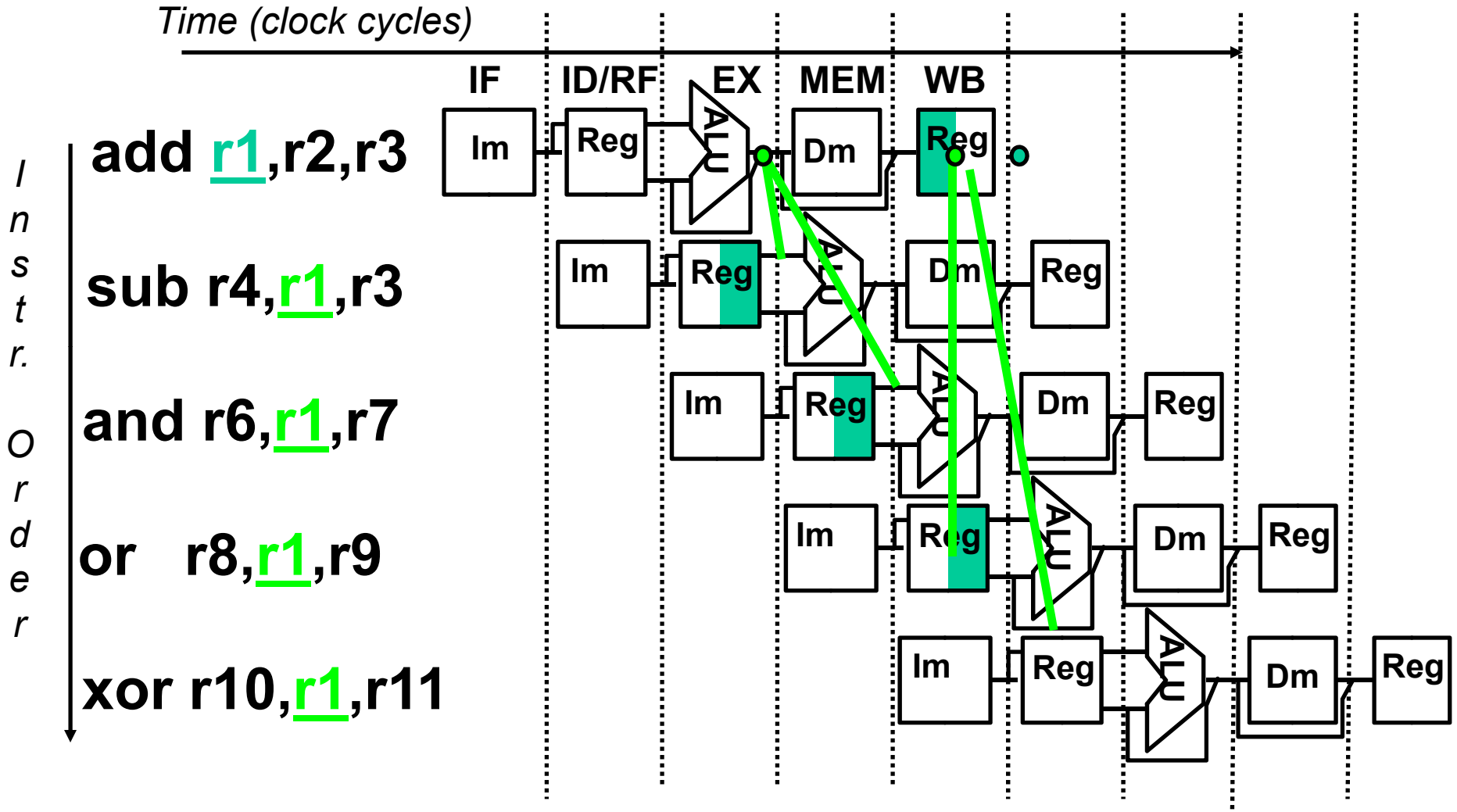
Data Hazard



Dependencies backwards in time are hazards

* Slide is courtesy of Dave Patterson

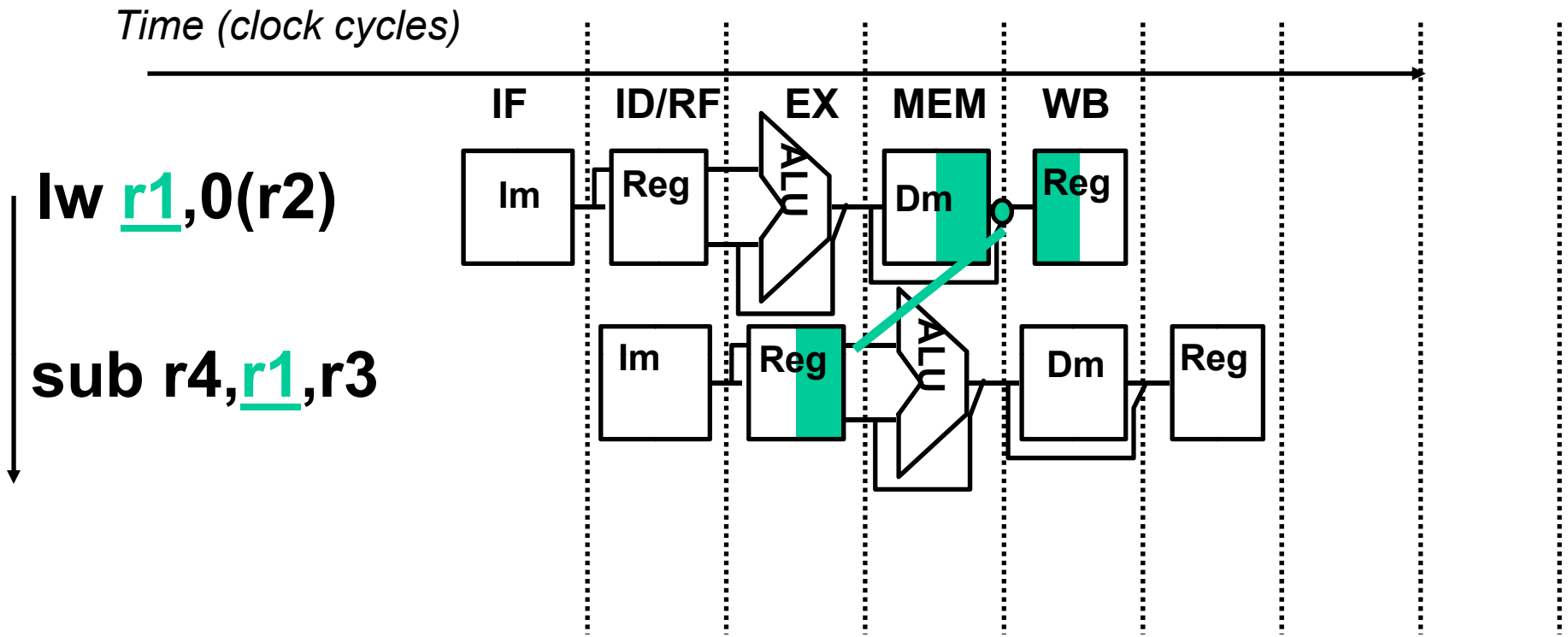
Data Hazard Solution



"Forward" result from one stage to another

* Slide is courtesy of Dave Patterson

Resolving Data Hazards for Loads

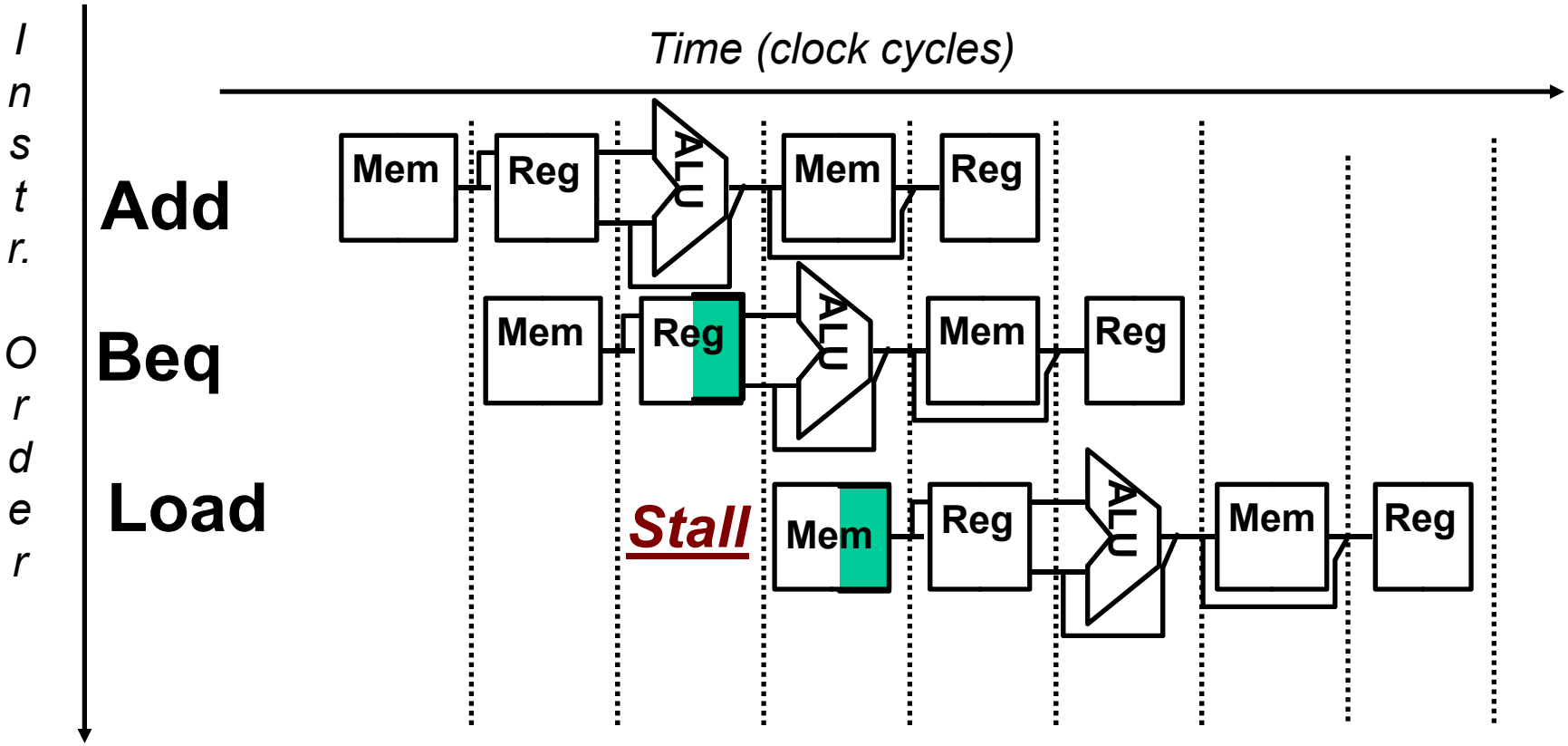


- Dependencies backwards in time are hazards
- Cannot solve with forwarding
- Must delay/stall instruction dependent on loads

* Slide is courtesy of Dave Patterson

Control Hazard

- Stall: wait until decision is clear
  - ➔ Its possible to move up decision to 2nd stage by adding hardware to check registers as being read

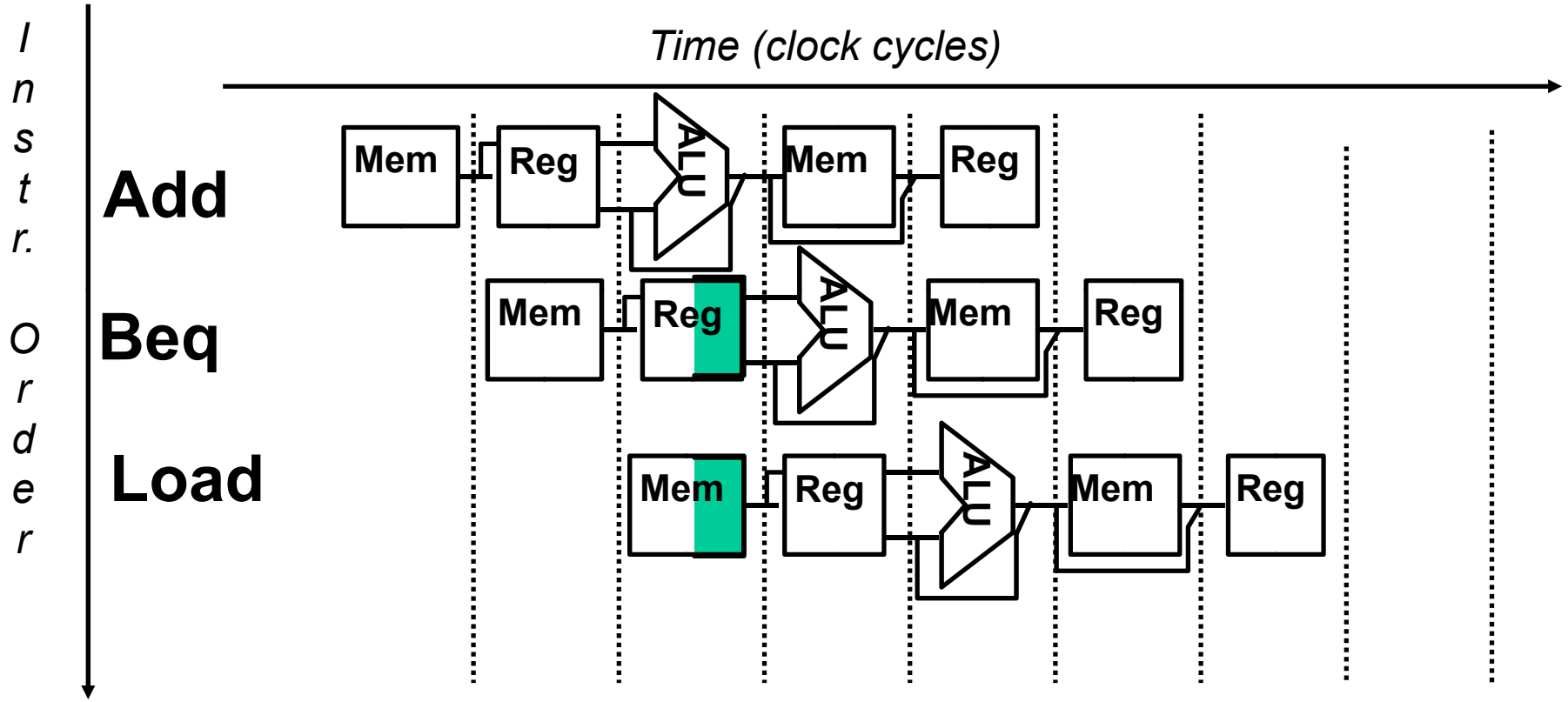


- Impact: 2 clock cycles per branch instruction ⇒ slow

* Slide is courtesy of Dave Patterson

Control Hazard Solution

- Predict: guess one direction then back up if wrong
  - ➔ Predict not taken

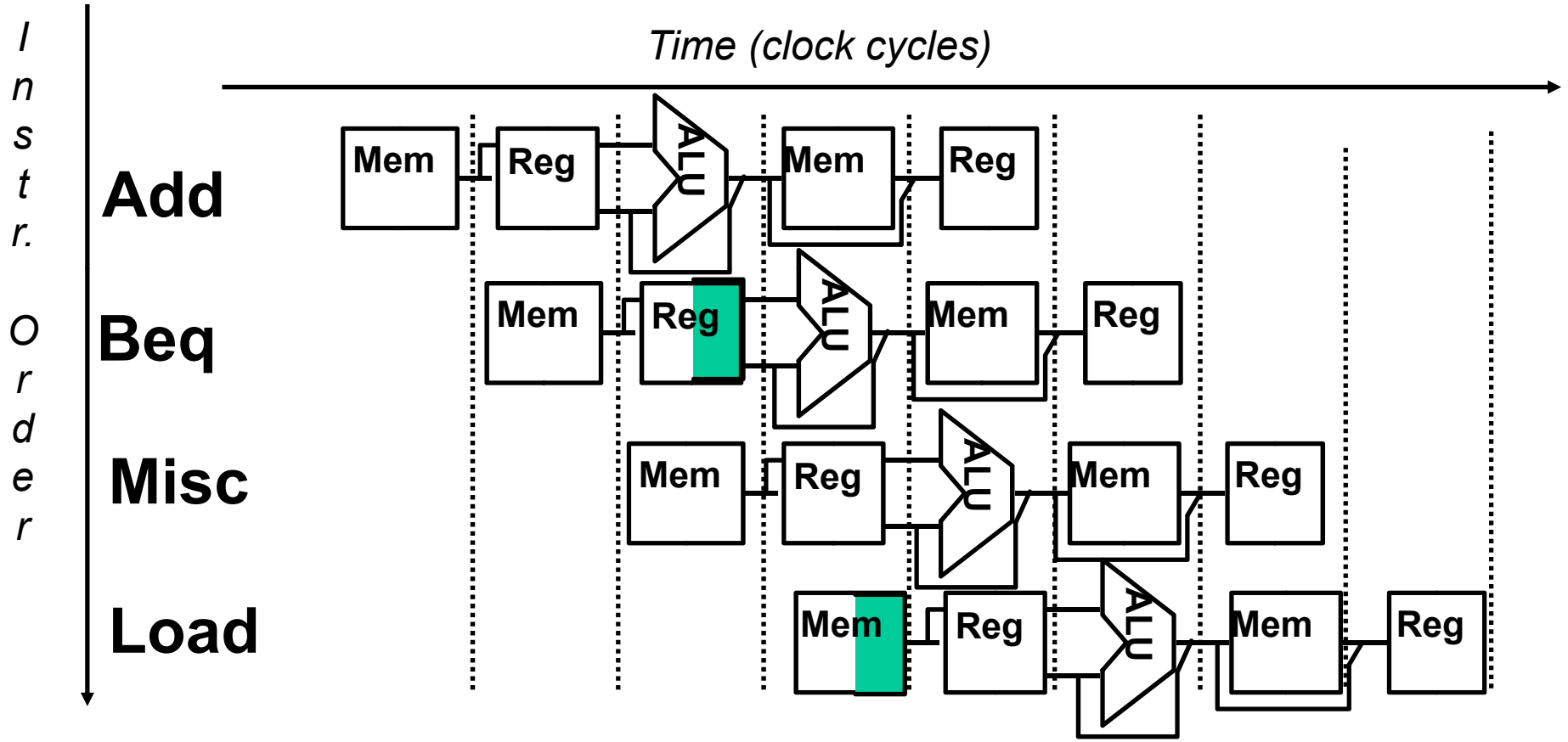


- Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right - 50% of time)
- More dynamic scheme: history of 1 branch (- 90%)

* Slide is courtesy of Dave Patterson

Control Hazard Solution

- Redefine branch behavior (takes place after next instruction) "delayed branch"



- Impact: 0 clock cycles per branch instruction if can find instruction to put in "slot" (- 50% of time)

* Slide is courtesy of Dave Patterson

Conclusion

❑ *Summary*

➔ An overview of Pipelining

- Pipelining concept is natural
- Start handling of next instruction while current one is in progress

➔ Pipeline performance

- Performance improvement by increasing instruction throughput
- Ideal and upper bound for speedup is number of stages in pipeline

➔ Pipelined hazards

- Structural, data and control hazards
- Hazard resolution techniques

❑ *Next Lecture*

➔ Data and control Hazards

➔ Pipelined control

Reading assignment includes Appendix A.1 & A.2 in the textbook