Advanced Computer Architecture

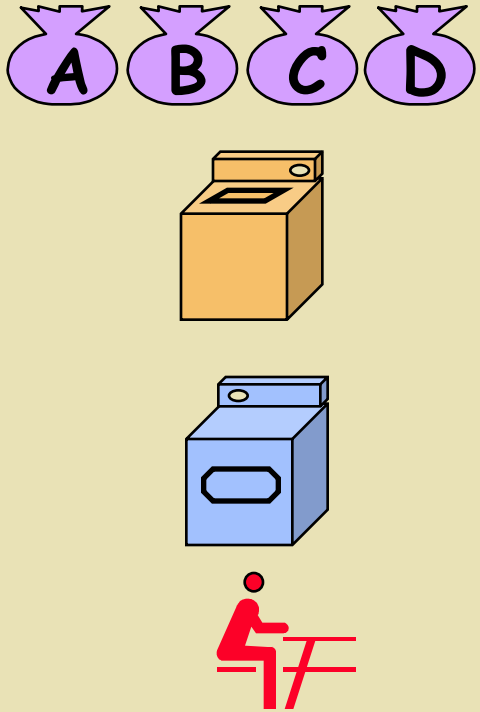
Pipelining

Prof. Roger Luis Uy
De La Salle University
College of Computer Studies

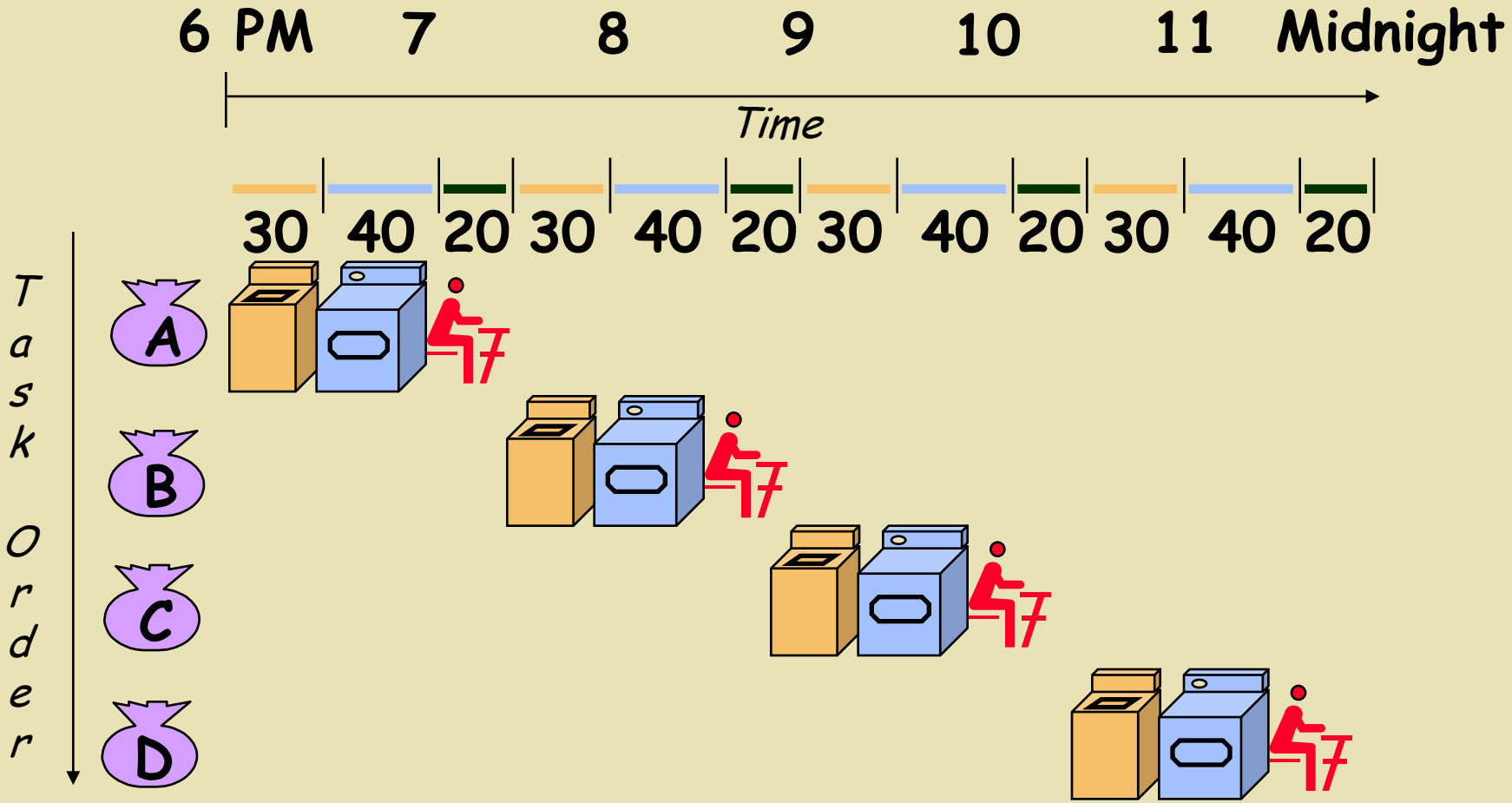
Pipelining: Its Natural!

Laundry Example

- Pedro, Maria, Juan, Gloria each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- "Folder" takes 20 minutes

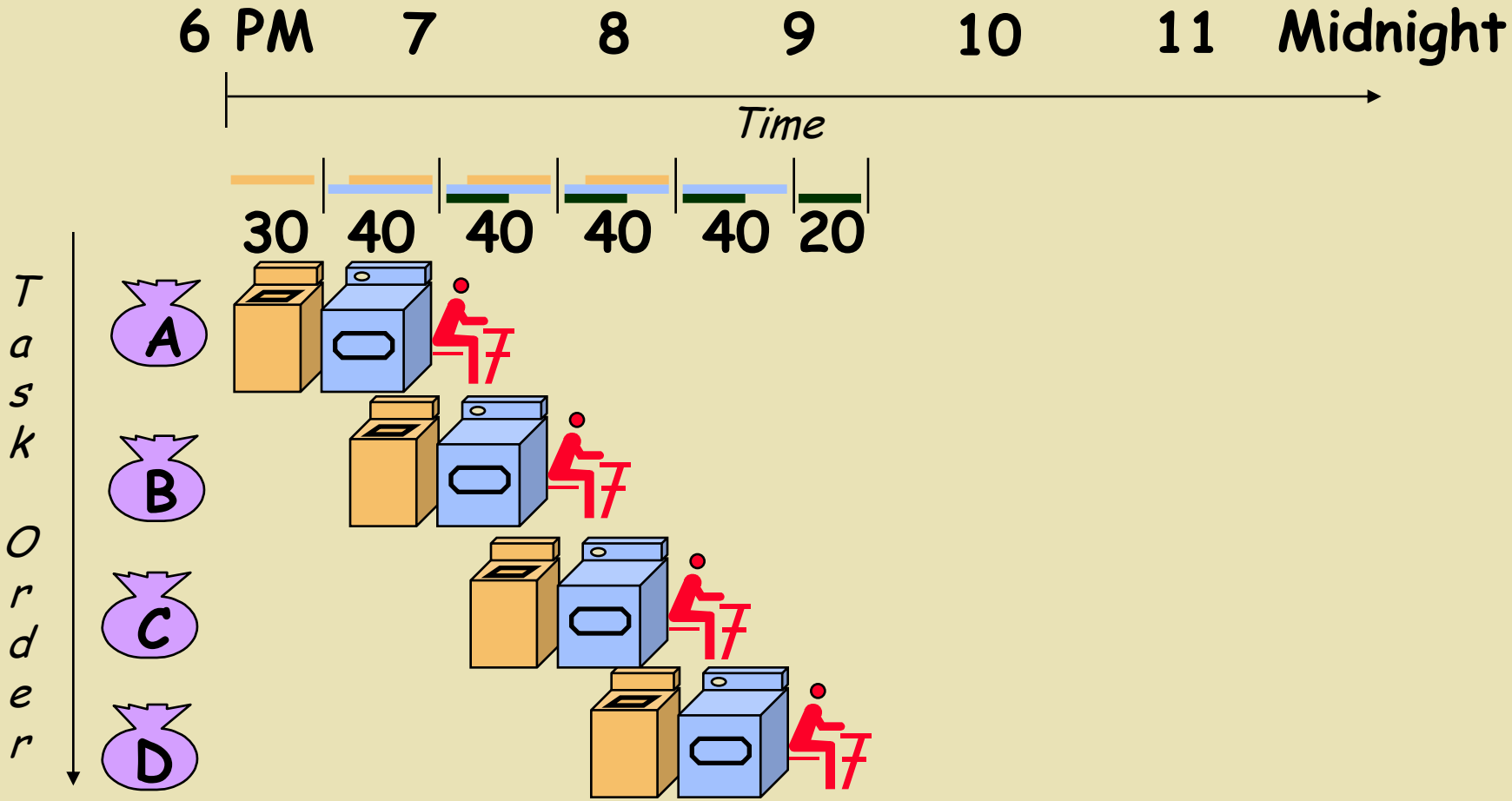


Sequential Laundry



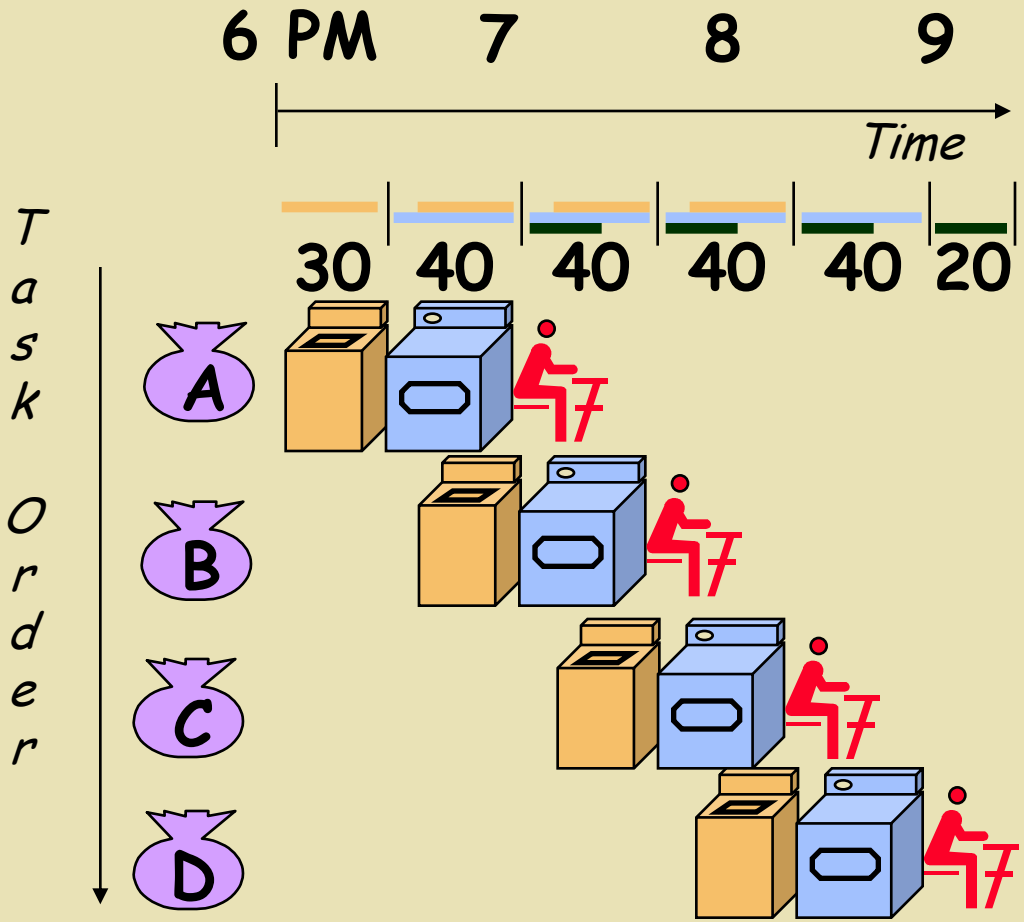
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry
Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

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
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

Pipelining

- It is an implementation technique whereby multiple instructions are overlapped in execution
- Pipe stage/segment – a step in the pipeline that completes part of an instruction
- Instruction pipeline throughput:
  - How often an instruction exits the pipeline

Pipelining

- Time per instruction on the pipelined machine:

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

- Note: This only applies in ideal conditions, such as when all stages are perfectly balanced and pipeline overhead is small

MIPS Cycles

1. Instruction fetch cycle (IF)
2. Instruction decode/register fetch cycle (ID)
3. Execution/effective address cycle (EX)
4. Memory access/branch completion cycle (MEM)
5. Write-back cycle (WB)

MIPS Cycles

1. Instruction fetch cycle (IF)

IR ← Mem [PC]
NPC ← PC + 4

MIPS Cycles

2. Instruction decode/register fetch cycle (ID)

$$A \leftarrow Regs[IR_{6..10}]$$
$$B \leftarrow Regs[IR_{11..15}]$$
$$Imm \leftarrow ((IR_{16})^{48} \,\#\#\, IR_{16..31})$$

Decoding is done in parallel with reading registers bec. these fields are a fixed location in a MIPS instruction format

MIPS Cycles

3. Execution/effective address cycle (EX)

The ALU operates on the operands prepared in the prior cycle, performing of four functions depending on the MIPS instruction type.

- 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 *op* 0)

MIPS Cycles

3. Execution/effective address cycle (EX)

The ALU operates on the operands prepared in the prior cycle, performing of four functions depending on the MIPS instruction type.

- Jump:

  ALUOutput ← Imm << 2

  Cond ← 1

MIPS Cycles

4. Memory access/branch completion cycle (MEM)

The only MIPS instructions active in this cycle are **loads**, **stores**, and **branches**.

* PC ← NPC
*Memory reference:
LMD ← Mem [ALUOutput] or
Mem [ALUOutput] ← B
Branch:
If (cond) PC ← ALUOutput

MIPS Cycles

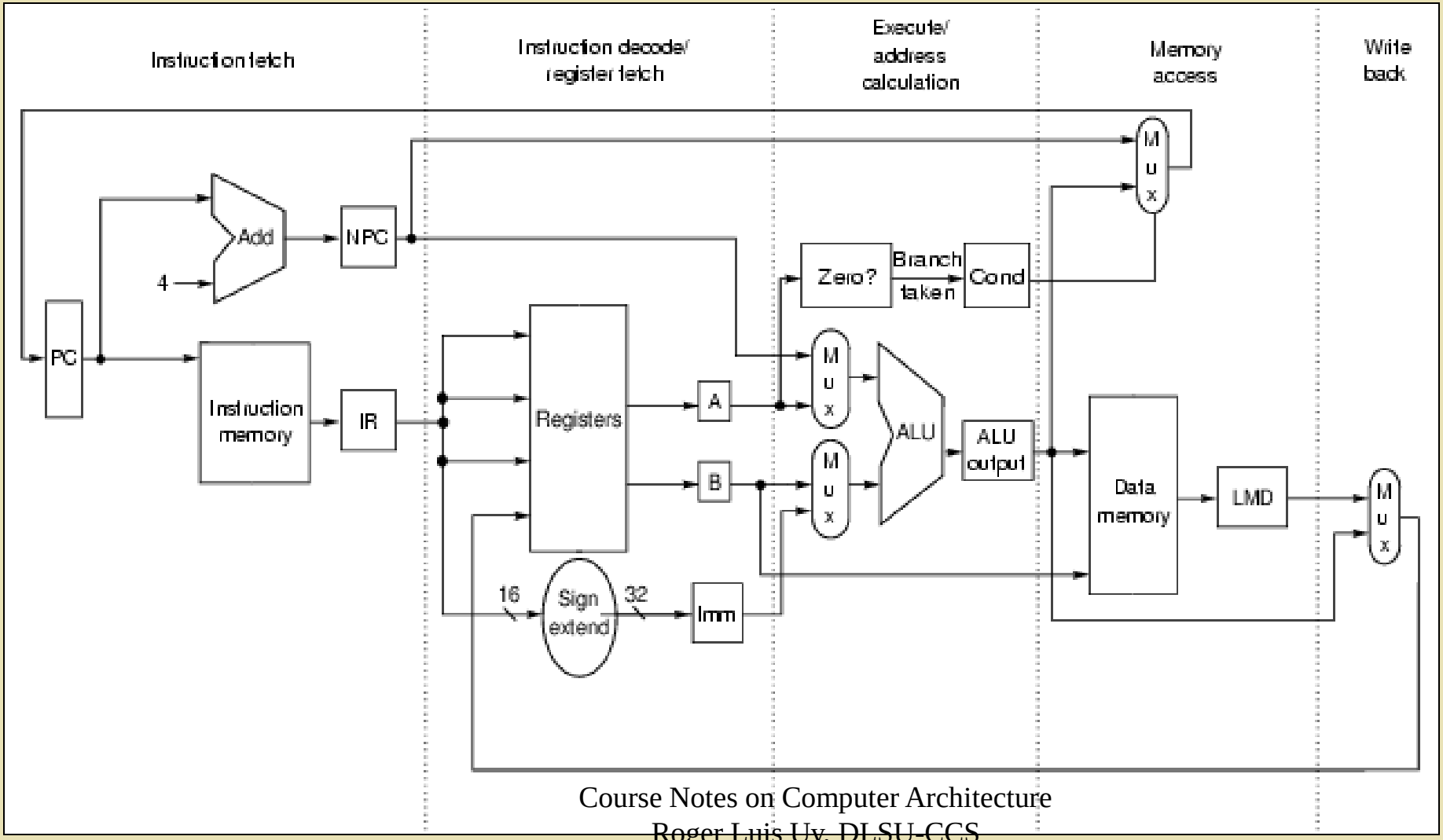
5. Write-back cycle (WB)

•Register-Register ALU instruction:
Regs [IR16..20] ← ALUOutput

•Register-Immediate ALU instruction:
Regs [IR11..15] ← ALUOutput

•Load instruction:
Regs[IR11..15] ← LMD

MIPS Datapath



MIPS Implementation

- Some Observations:
  - Branch & store instructions require 4 cycles and all other instructions require 5 cycles
  - The CPI could be improved by completing the ALU instructions on the 4th cycle
  - There are redundant hardware: two ALUs (not use in the same cycle), different memory for instruction and data

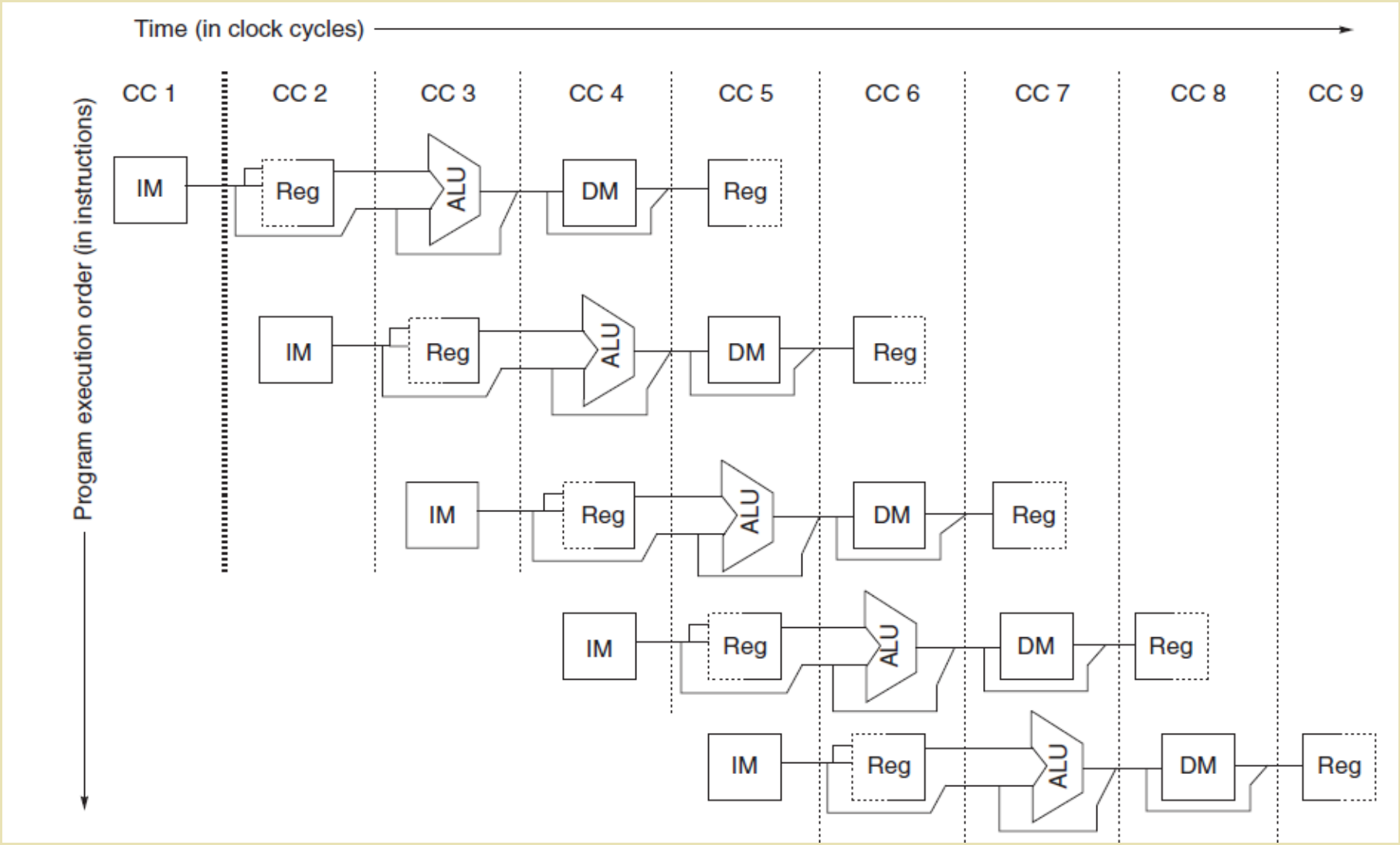
MIPS Pipeline

	Clock Number								
Instruction Number	1	2	3	4	5	6	7	8	9
Inst. I	IF	ID	EX	MEM	WB				
Inst. I+1		IF	ID	EX	MEM	WB			
Inst. I+2			IF	ID	EX	MEM	WB		
Inst. I+3				IF	ID	EX	MEM	WB	
Inst. I+4					IF	ID	EX	MEM	WB

MIPS Pipeline

- Pipeline observation #1:
  - No two different operations with the same datapath resource on the same clock cycle
    - Thus, different instruction and data memory
    - Register file is used in 2 stages: ID & WB. But what happen if read and write on **different** register? How about the same register?
    - PC – Increment PC vs. Branch. It's a problem!

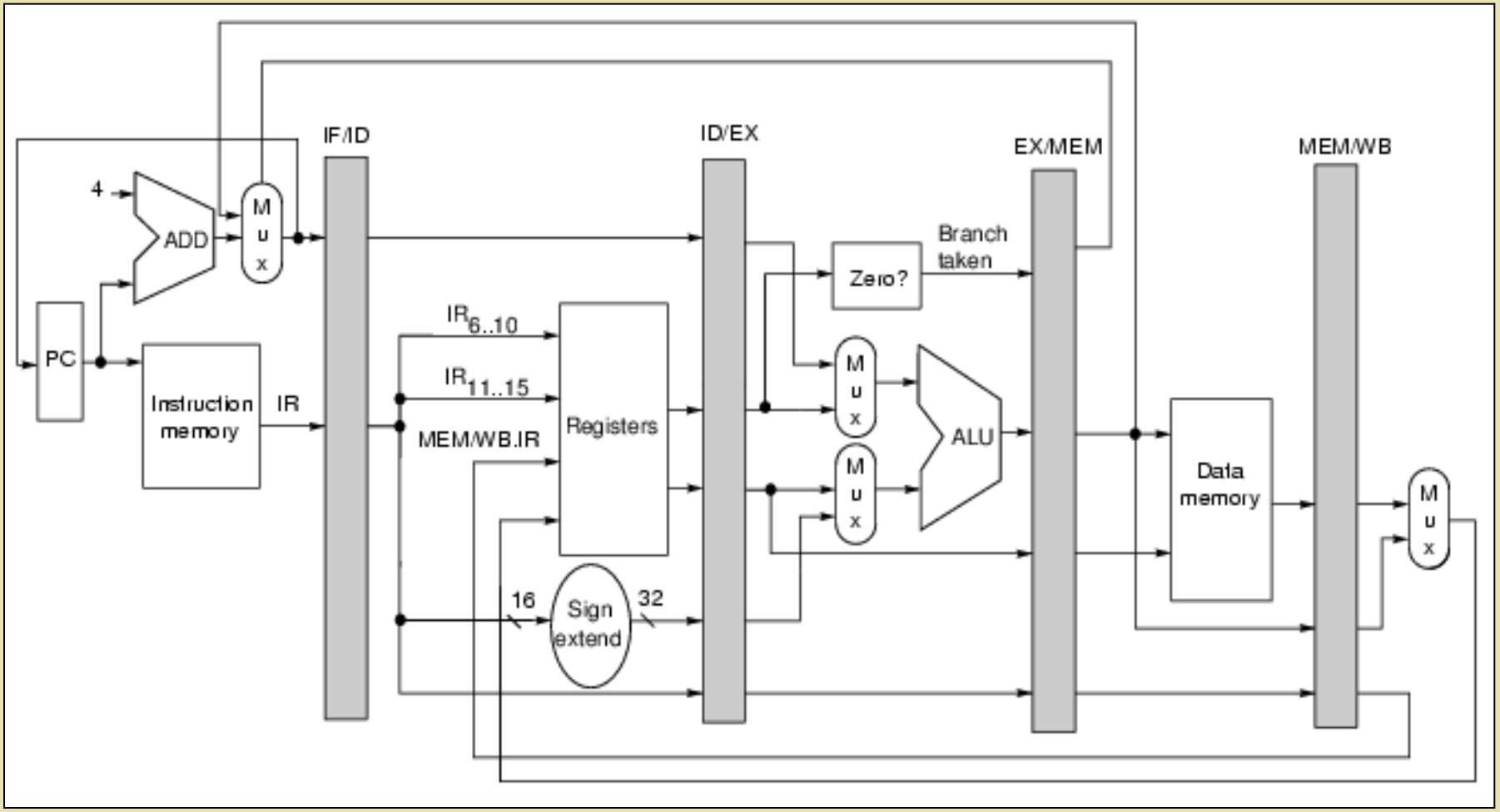
Visualizing Pipelining



MIPS Pipeline

- Pipeline observation #2:
  - Every pipestage is active on every clock cycle thus all operations in a pipe stage must complete in one clock cycle
  - Requires pipeline registers or latches since data needs to be passed from one pipe stage to another
  - The selection multiplexer for the PC has been move to IF stage so that there will be no conflict when a branch occurred, since 2 instructions would try to write different values into the PC

MIPS Pipeline



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[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]);		
	ALU instruction	**Load or store instruction**	**Branch instruction**
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.IR to ID/EX.IR EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.B ← ID/EX.B;	EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm << 2); EX/MEM.cond ← (ID/EX.A == 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[rd]] ← MEM/WB.ALUOutput; or Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD;	

MIPS64 Pipeline Algorithm (simplified)

Course Notes on Computer Architecture

Roger Luis Uy, DLSU-CCS

Stage	Any instruction		
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC,PC ← (if 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 instruction	**Load or store instruction**	**Branch instruction**
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$_{16..20}$] ← MEM/WB.ALUOutput; or Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.ALUOutput;	Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.LMD;	

MIPS64 Pipeline Algorithm (detailed)

Course Notes on Computer Architecture
Roger Luis Uy, DLSU-CCS

MIPS Cycles (Pipeline)

1. Instruction fetch cycle (IF)

IF/ID.IR ← Mem [PC];
IF/ID.NPC, PC ← if ((EX/MEM.opcode == branch) & EX/MEM.cond) {EX/MEM.ALUOutput} else {PC + 4});

MIPS Cycles (Pipeline)

2. Instruction decode/register fetch cycle (ID)

$$ID/EX.A \leftarrow Regs\,[IF/ID.IR_{6..10}];$$
$$ID/EX.B \leftarrow Regs\,[IF/ID.IR_{11..15}];$$
$$ID/EX.Imm \leftarrow ((IF/ID.IR_{16})^{48} \,\#\#\, IF/ID.IR_{16..31});$$
$$ID/EX.NPC \leftarrow IF/ID.NPC;$$
$$ID/EX.IR \leftarrow IF/ID.IR;$$

MIPS (Pipeline)

3. Execution/effective address cycle (EX)

(ALU Instruction)

EX/MEM.ALUOutput ← ID/EX.A *func* ID/EX.B;

or EX/MEM.ALUOutput ← ID/EX.A op ID/EX.IMM;

EX/MEM.IR ← ID/EX.IR;

EX/MEM.COND ← 0;

MIPS (Pipeline)

3. Execution/effective address cycle (EX)

(Load/Store Instruction)

EX/MEM.ALUOutput ← ID/EX.A + ID/EX.IMM;
EX/MEM.B ← ID/EX.B;
EX/MEM.IR ← ID/EX.IR;
EX/MEM.COND ← 0;

MIPS (Pipeline)

3. Execution/effective address cycle (EX)

(Branch Instruction)

EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.IMM <<2)

EX/MEM.COND ← (ID/EX.A op 0)

MIPS (Pipeline)

3. Execution/effective address cycle (EX)

(Jump Instruction)

EX/MEM.ALUOutput ← ID/EX.IMM <<2

EX/MEM.COND ← 1

MIPS Cycles (Pipeline)

4. Memory access (MEM)

(Load Instruction)

MEM/WB.IR ← EX/Mem.IR;
MEM/WB.LMD ← MEM(EX/MEM.ALUOutput);

(Store Instruction)

MEM/WB.IR ← EX/Mem.IR;
MEM(EX/MEM.ALUOutput) ← EX/MEM.B;

(ALU Instruction)

MEM/WB.IR ← EX/Mem.IR;
MEM/WB.ALUOutput) ← EX/MEM.ALUOutput;

MIPS Cycles (Pipeline)

5. Write-back cycle (WB)

•Register-Register ALU instruction:
Regs [MEM/WB.IR16..20] ← MEM/WB.ALUOutput;

•Register-Immediate ALU instruction:
Regs [MEM/WB.IR11..15] ← MEM/WB.ALUOutput;

•Load instruction:
Regs [MEM/WB.IR11..15] ← MEM/WB.LMD;