微算機系統 Fall 2020
Microprocessor Systems

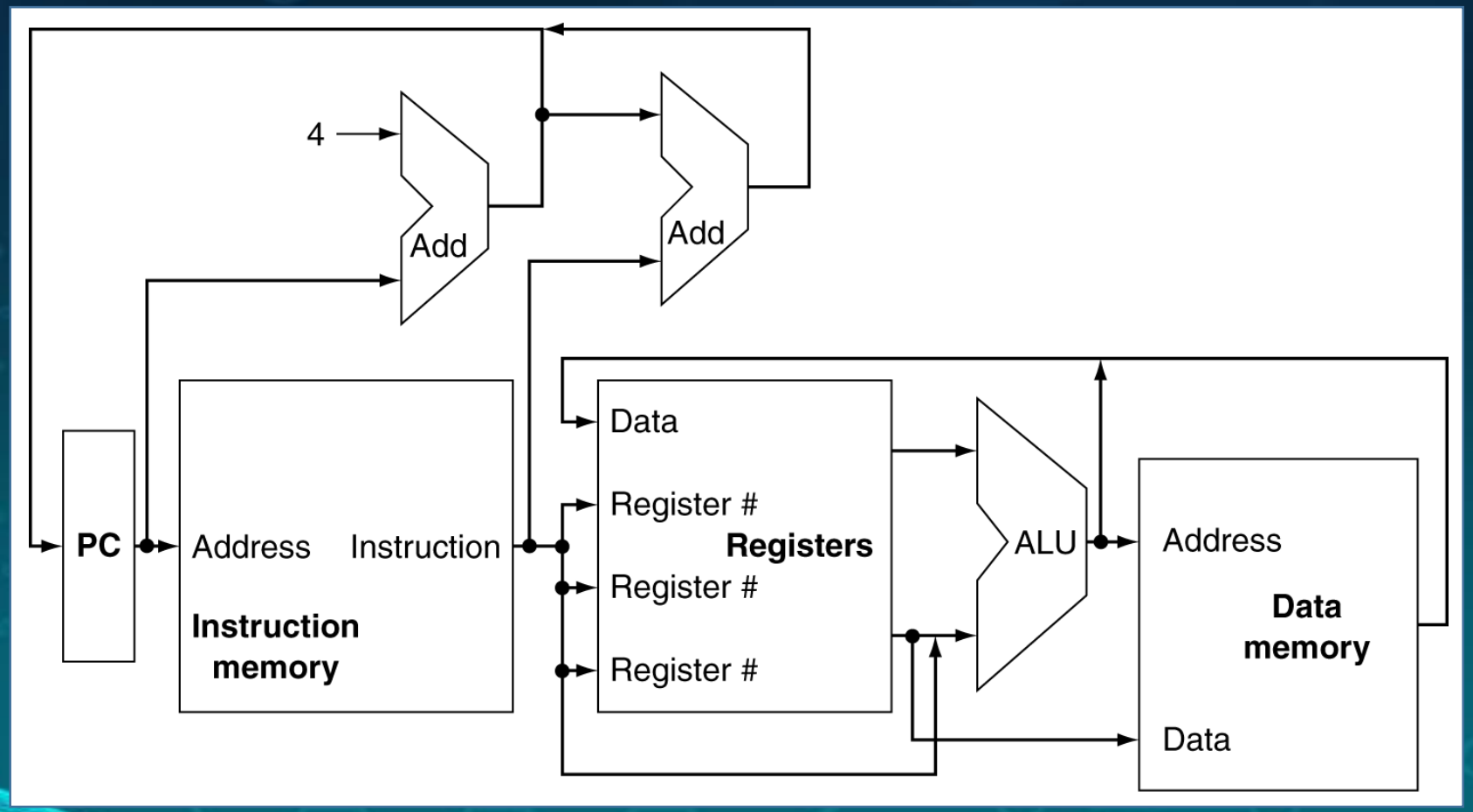
Instructor：Yen-Lin Chen(陳彥霖), Ph.D. Professor
Dept. Computer Science and Information Engineering
National Taipei University of Technology

計算機組織相關內容參考

Introduction

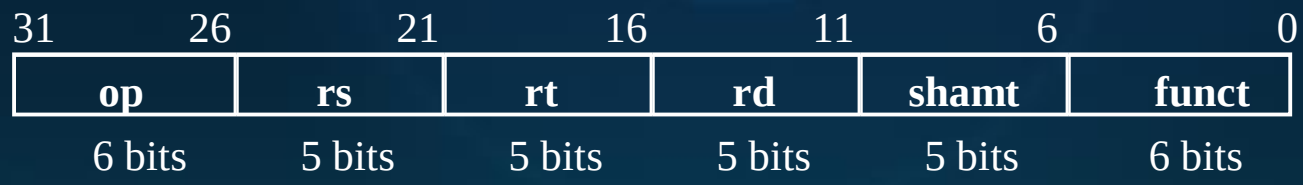
- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version
  - A more realistic **pipelined** version
- Simple subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`

CPU Overview



Step 1: Analyze Instruction Set

- R-type MIPS instructions are 32 bits long:
  - R-type:

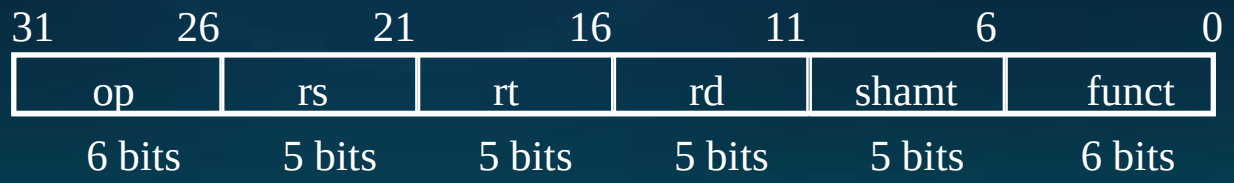


- These fields are:
  - op: operation of the instruction
  - rs, rt, rd: source and destination register
  - shamt: shift amount
  - funct: selects variant of the “op” field

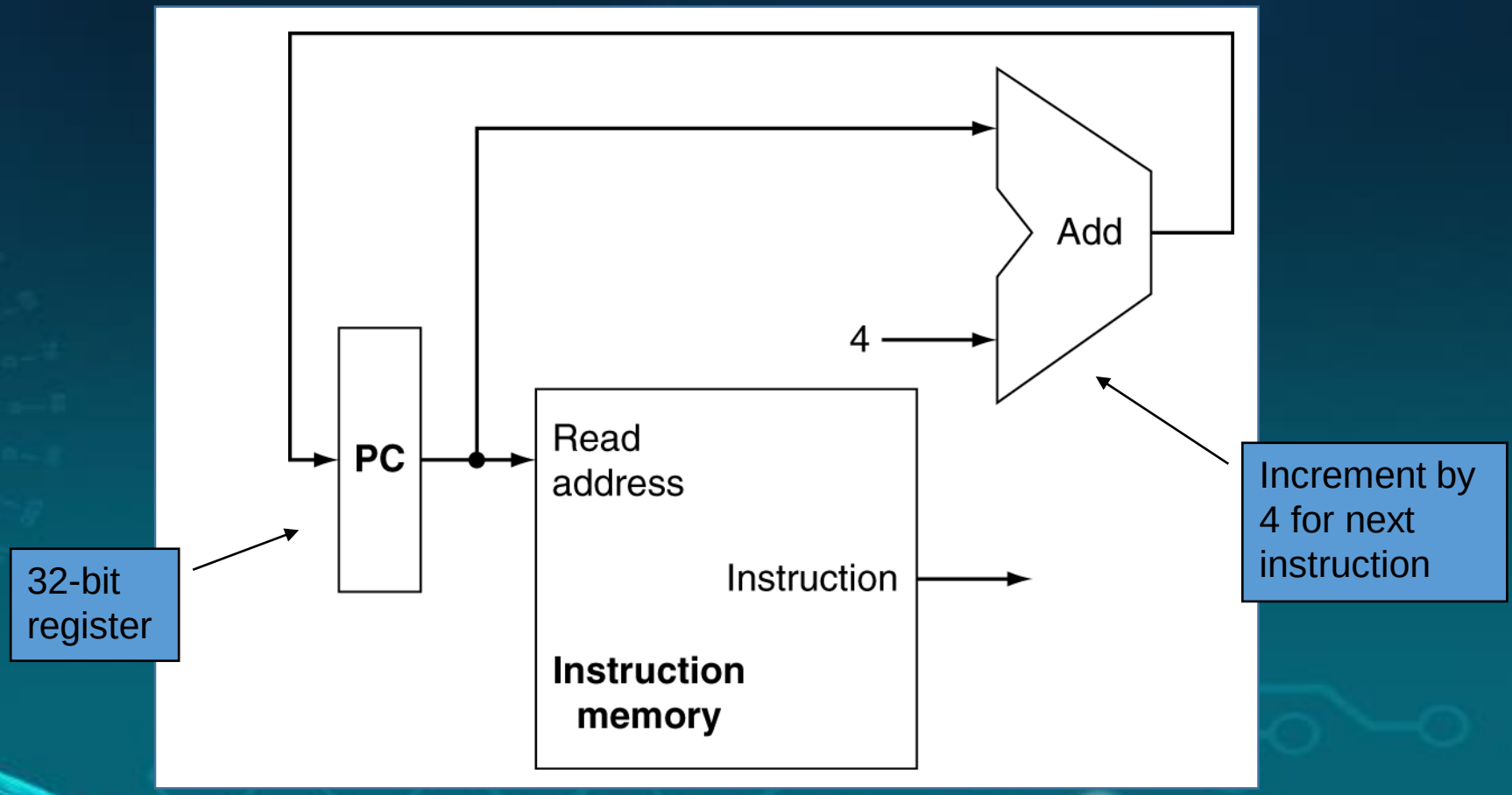
Example: R-type MIPS Subset

R-Type:

- add rd, rs, rt
- sub rd, rs, rt
- and rd, rs, rt
- or rd, rs, rt
- slt rd, rs, rt



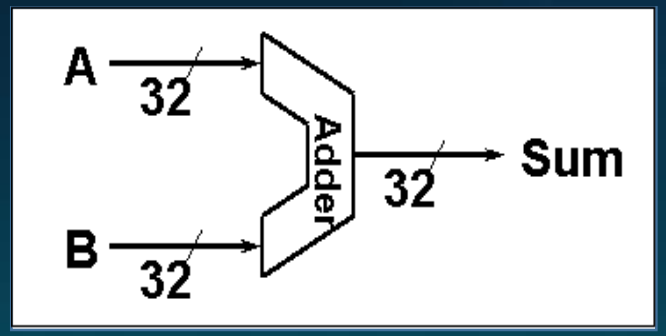
Instruction Fetch



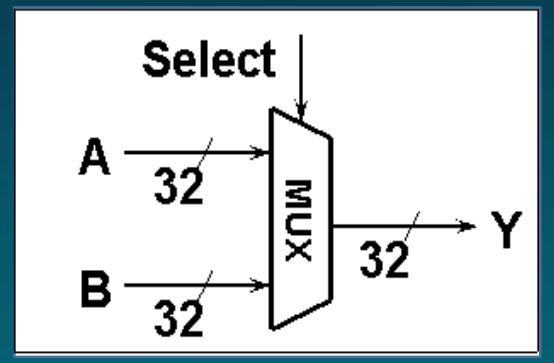
Step 2a: Datapath Components

- Basic building blocks of combinational logic elements:

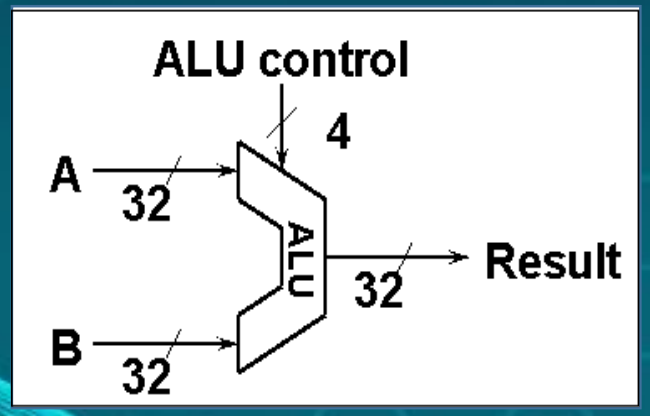
- Arithmetic
  - Add and sub register or extended immediate
  - Add 4 or extended immediate to PC
- Registers (32 x 32)
  - read RS
  - read RT
  - Write RT or RD
- Memory
  - store instructions and data
- PC
- Branch
  - Extender for zero- or sign-extension



Adder for PC



MUX to select data



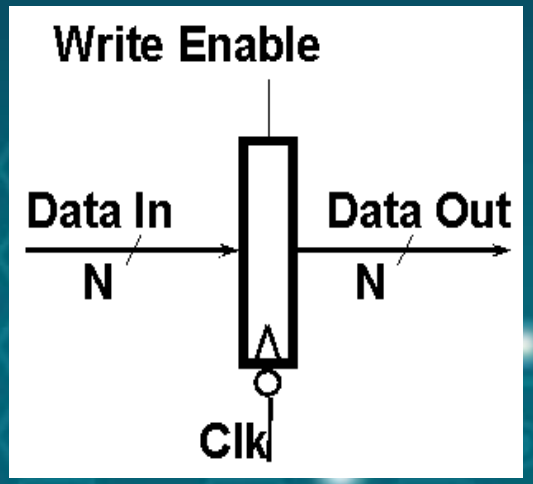
ALU for operations

Step 2b: Datapath Components

Storage elements:

- Register:
  - Similar to the D Flip Flop except
    - N-bit input and output
    - Write Enable input
  - Write Enable:
    - negated (0): Data Out will not change
    - asserted (1): Data Out will become Data In

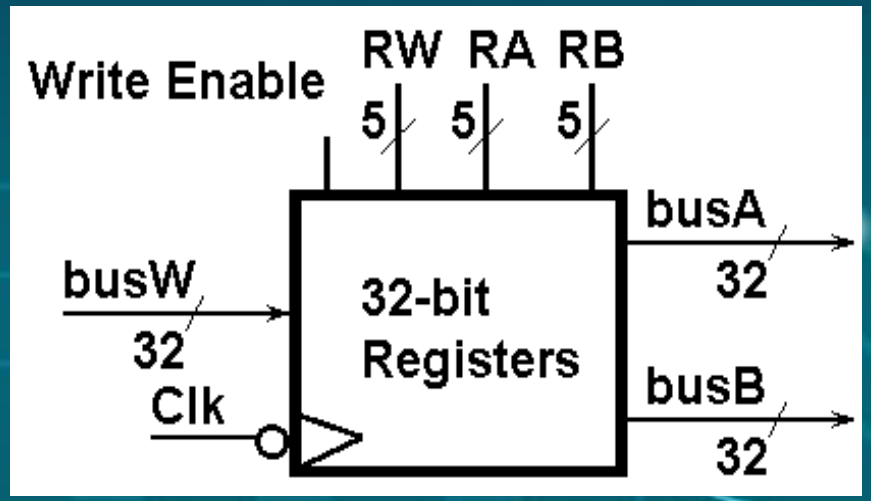
- Arithmetic
  - Add and sub register or extended immediate
  - Add 4 or extended immediate to PC
- Registers (32 x 32)
  - read RS
  - read RT
  - Write RT or RD
- Memory
  - store instructions and data
- PC
- Branch
  - Extender for zero- or sign-extension



Storage Element: Register File

- Consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
- Register is selected by:
  - RA selects the register to put on busA
  - RB selects the register to put on busB
  - RW selects the register to be written via busW when Write Enable is 1
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read, behaves as a combinational circuit

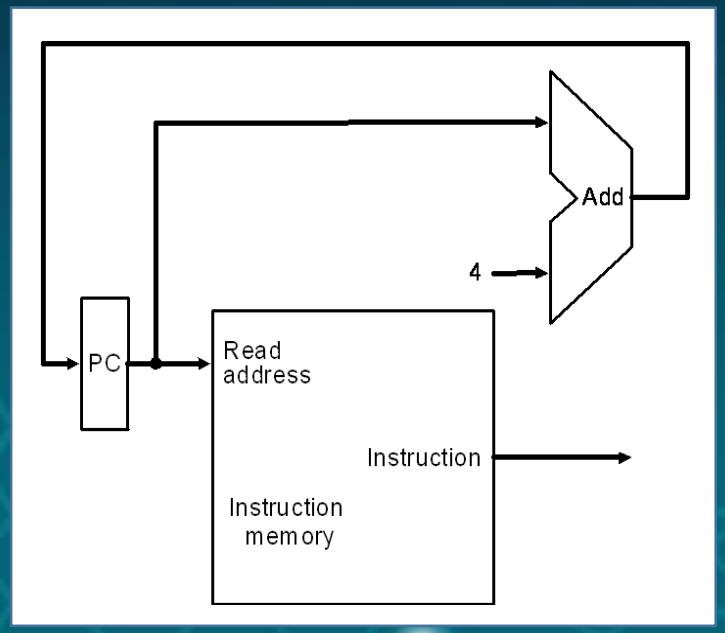
- Arithmetic
  - Add and sub register or extended immediate
  - Add 4 or extended immediate to PC
- Registers (32 x 32)
  - read RS
  - read RT
  - Write RT or RD
- Memory
  - store instructions and data
- PC
- Branch
  - Extender for zero- or sign-extension



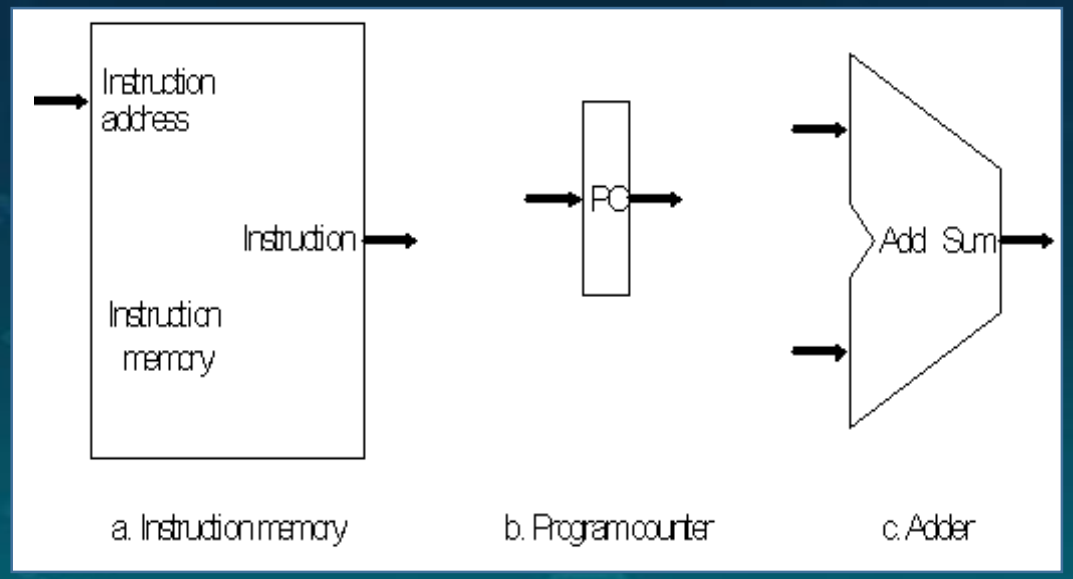
Step 3a: Datapath Assembly

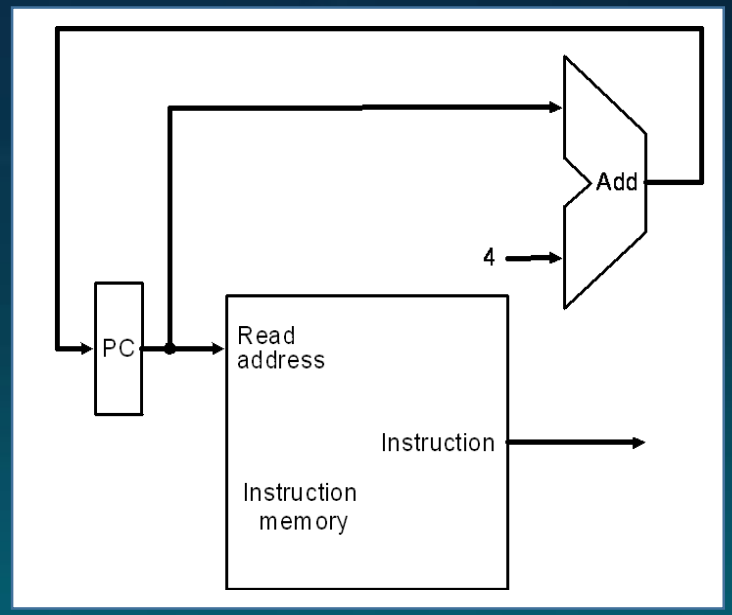
- Instruction fetch unit: a common operations
  - Fetch the instruction: mem[PC]
  - Update the program counter:
    - Sequential code: PC ← PC + 4
    - Branch and Jump: PC ← "Something else"

- Arithmetic
  - Add and sub register or extended immediate
  - Add 4 or extended immediate to PC
- Registers (32 x 32)
  - read RS
  - read RT
  - Write RT or RD
- Memory
  - store instructions and data
- PC
- Branch
  - Extender for zero- or sign-extension



Datapath: Instruction Store/Fetch & PC Increment

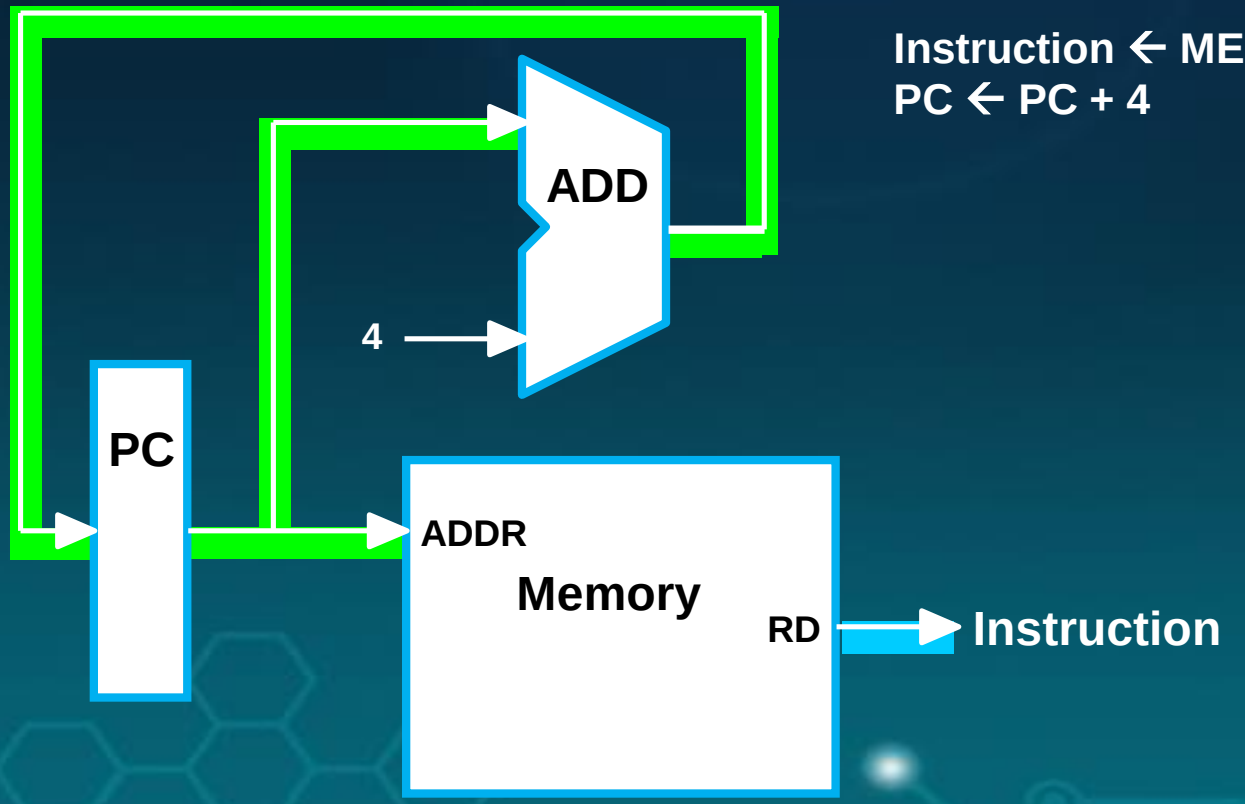




Three elements used to store and fetch instructions and increment the PC

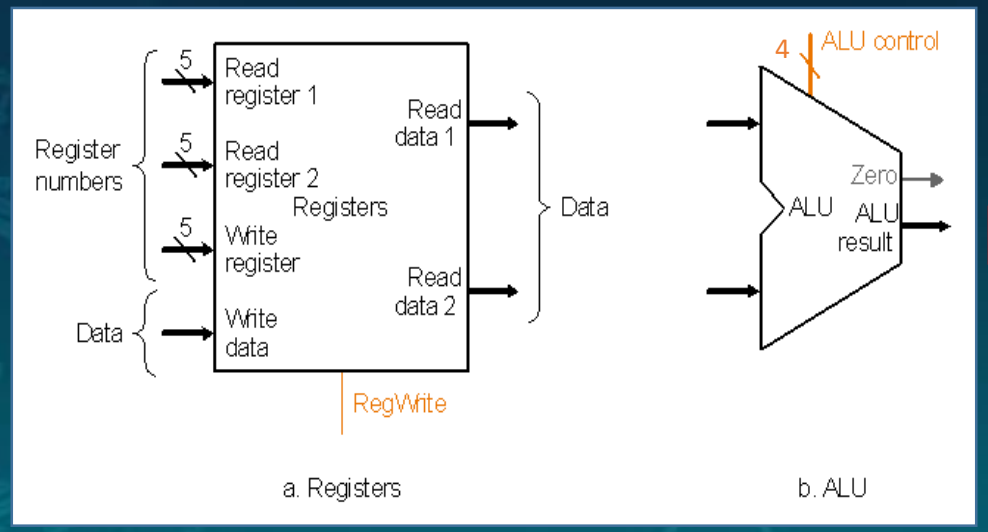
Datapath

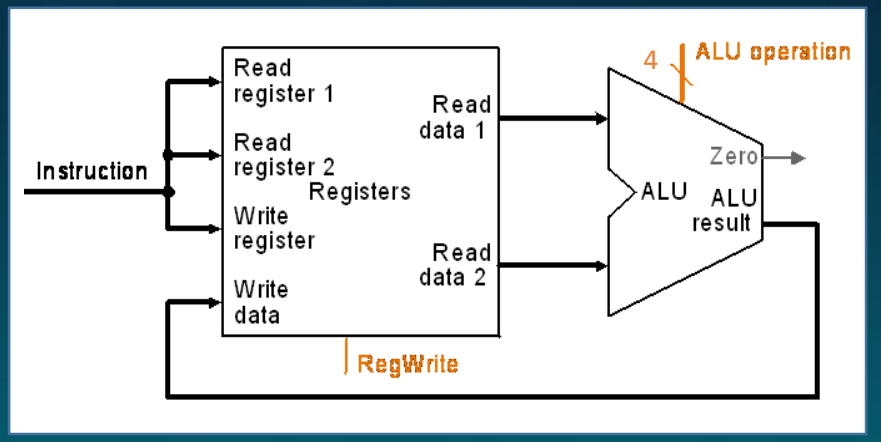
Animating the Datapath: PC



Instruction ← MEM[PC]
PC ← PC + 4

Datapath: R-Format Instruction



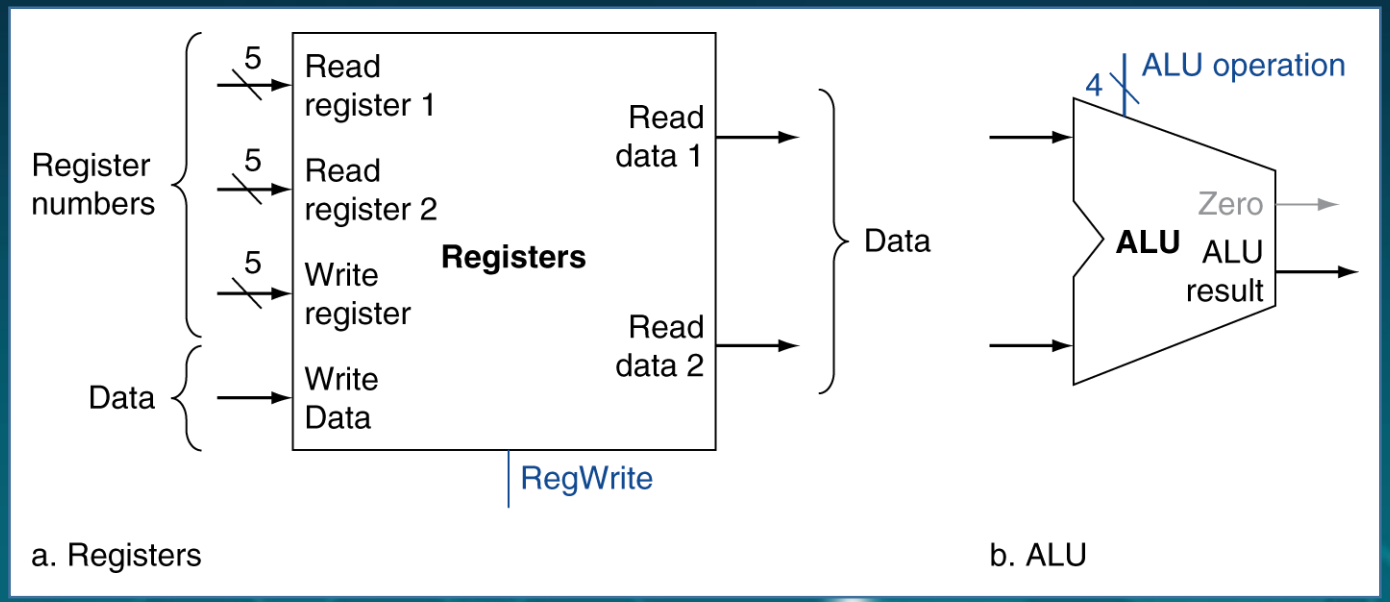


Two elements used to implement R-type instructions

Datapath

Step 3b: R-Format Instructions

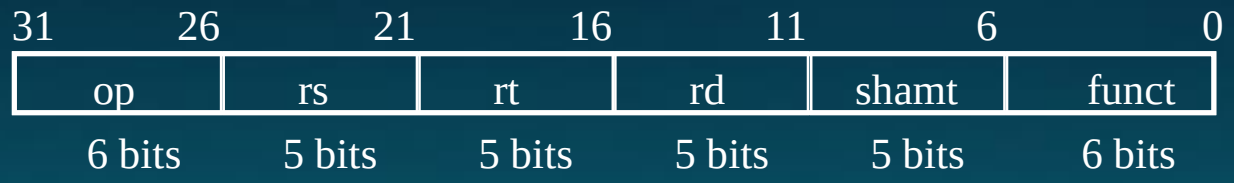
- Read two register operands
- Perform arithmetic/logical operation
- Write register result

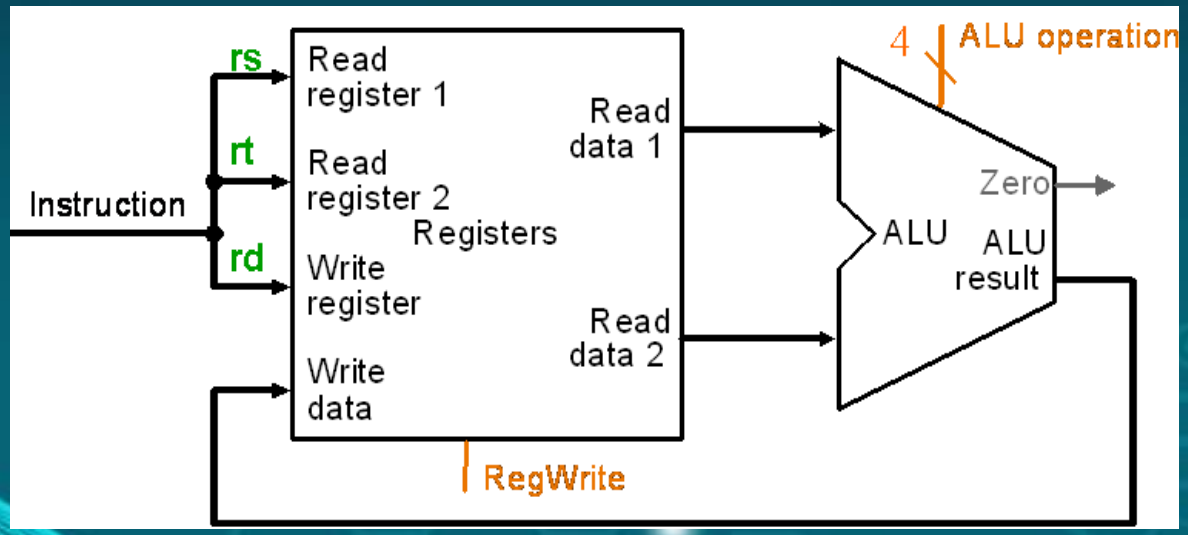


Step 3b: Add and Subtract

- R[rd] ← R[rs] op R[rt]
  - Ex: *add rd, rs, rt*
  - RA, RB, RW of the register file come from instruction's rs, rt, and rd fields
  - ALU operation and RegWrite: control logic after decode

- Arithmetic
  - Add and sub register or extended immediate
  - Add 4 or extended immediate to PC
- Registers (32 x 32)
  - read RS
  - read RT
  - Write RT or RD
- Memory
  - store instructions and data
- PC
- Branch
  - Extender for zero- or sign-extension

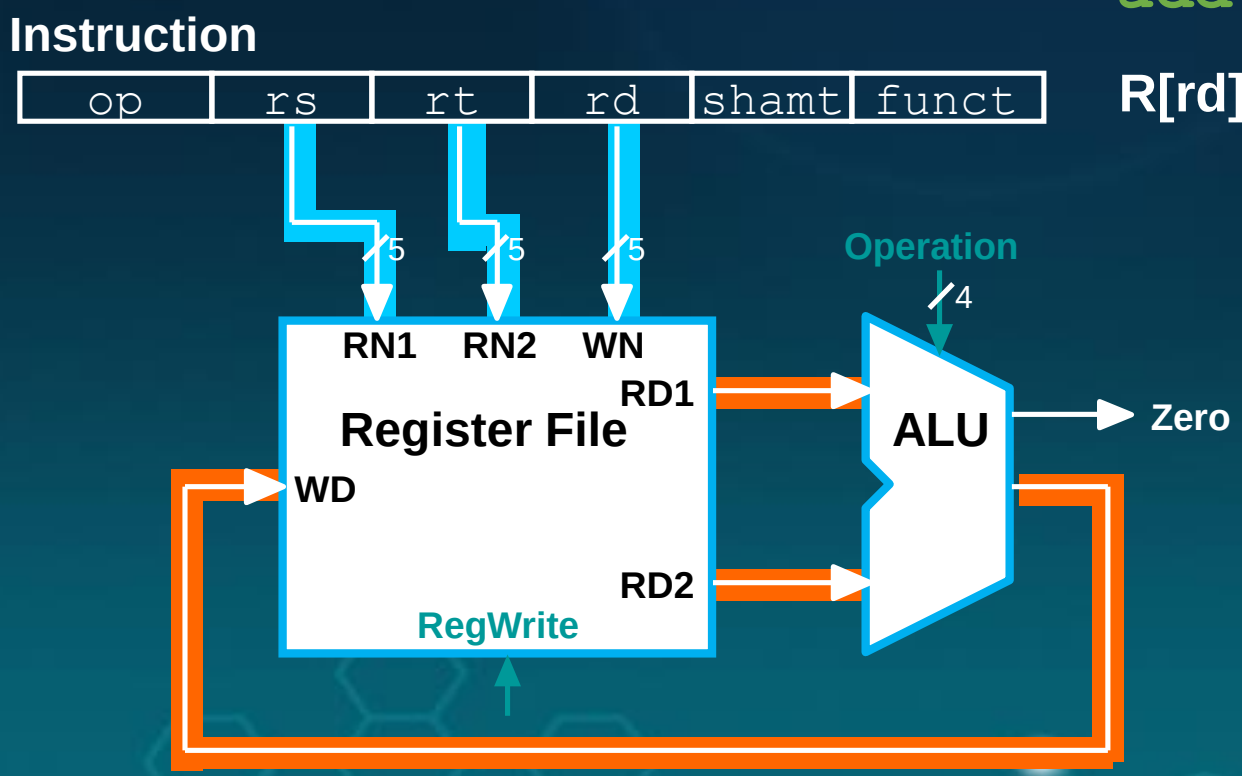




Animating the Datapath: R-type

```
add rd, rs, rt
```

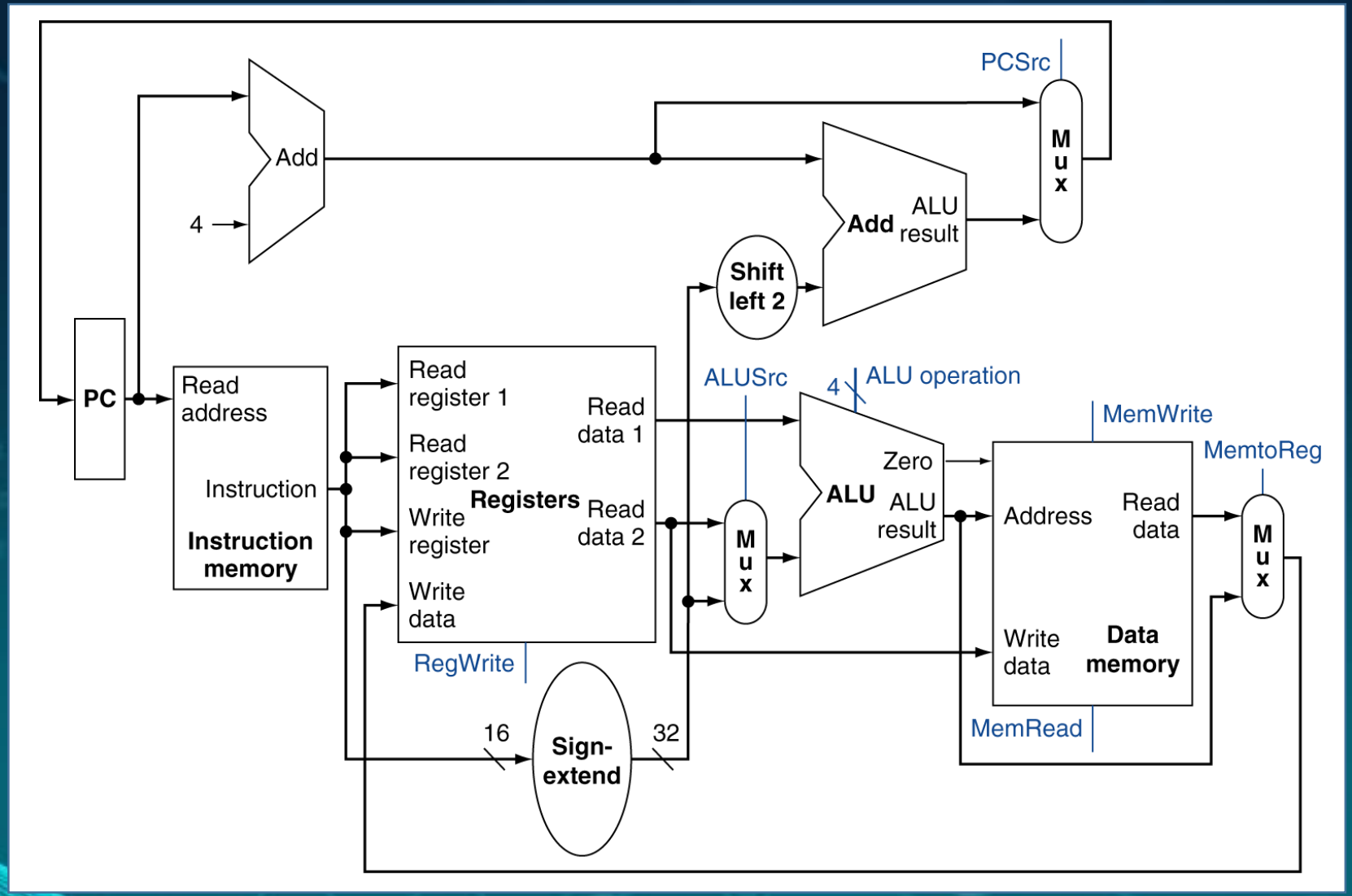
R[rd] ← R[rs] + R[rt];



Composing the Elements

- First-cut data path does an instruction in single clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

Full Datapath

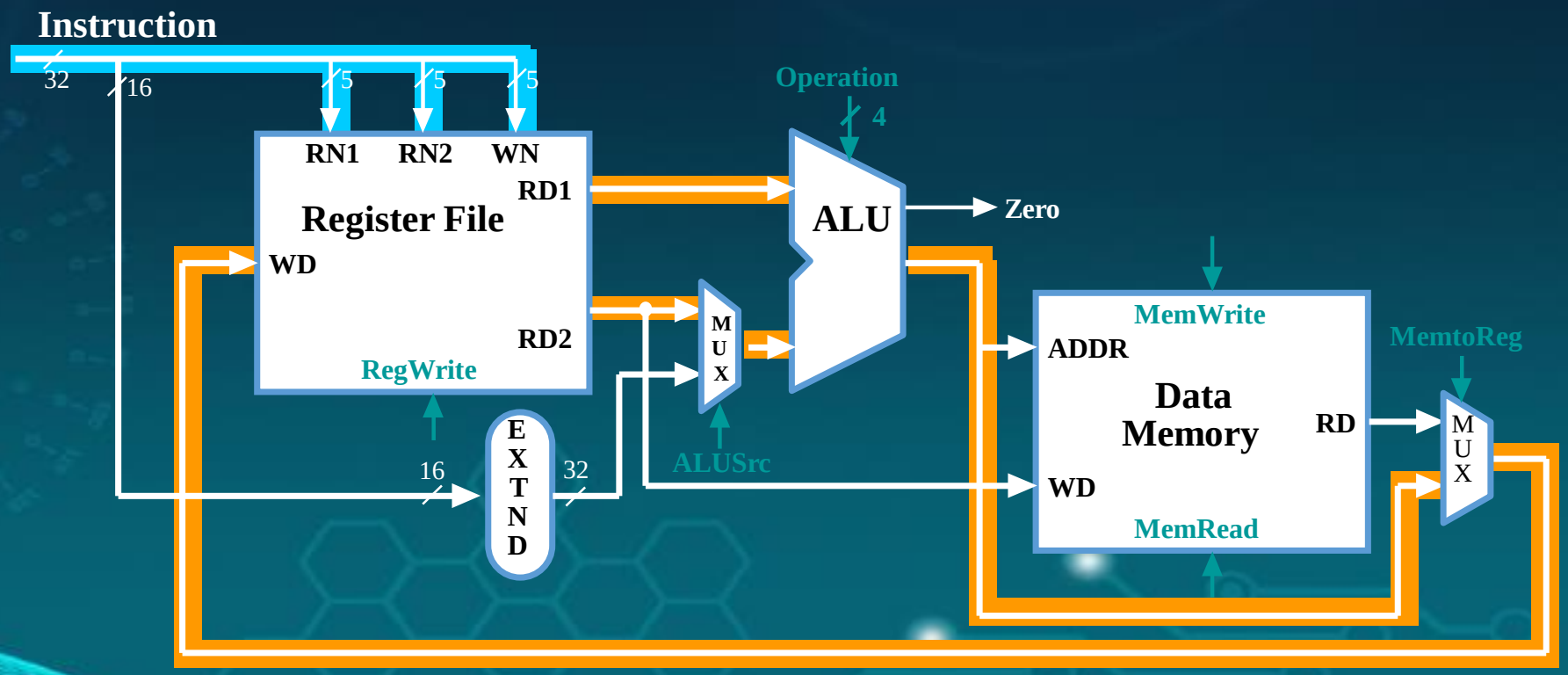


Single-cycle Implementation of MIPS: A Quick Note

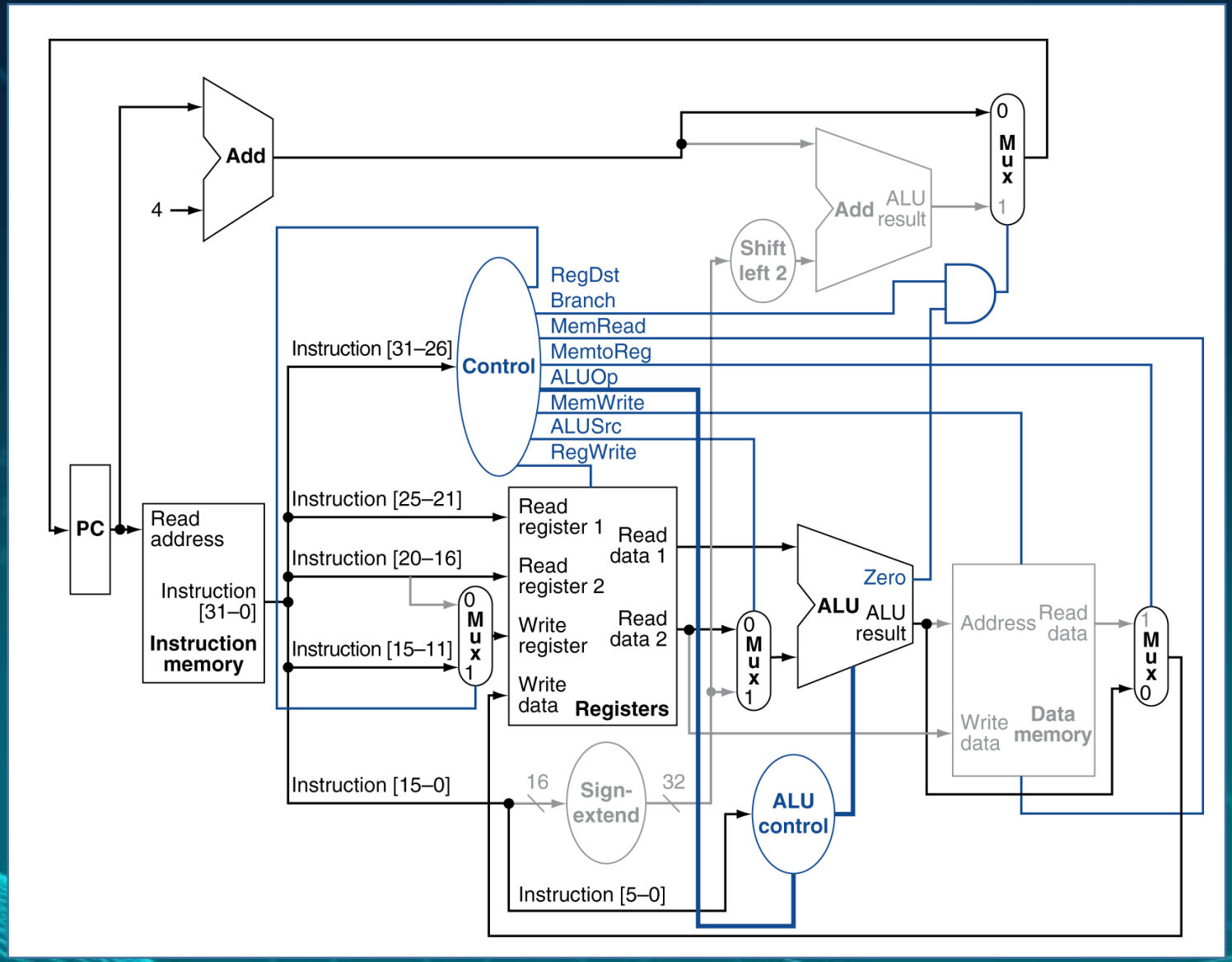
- Our first implementation of MIPS will use a *single* long clock cycle for every instruction
- Every instruction begins on one *rising(or*, falling) clock edge and ends on the next *rising* (or, falling) clock edge
- This approach is *not practical* as it is much slower than a ***pipeline*** **implementation** where different instruction classes can take different numbers of cycles (discussed later)
- Even though the single-cycle approach is not practical it is simple and useful to understand first

Animating the Datapath: R-type Instruction

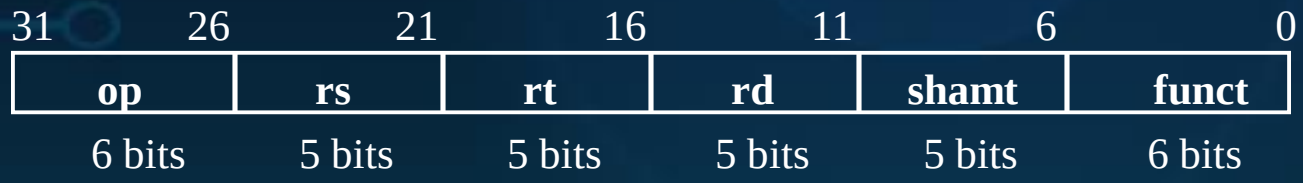
add rd,rs,rt



R-Type Instruction



Running Datapath with Control



- add rd, rs, rt

mem[PC]
PC+4

R[rs], R[rt]

R[rs] + R[rt]

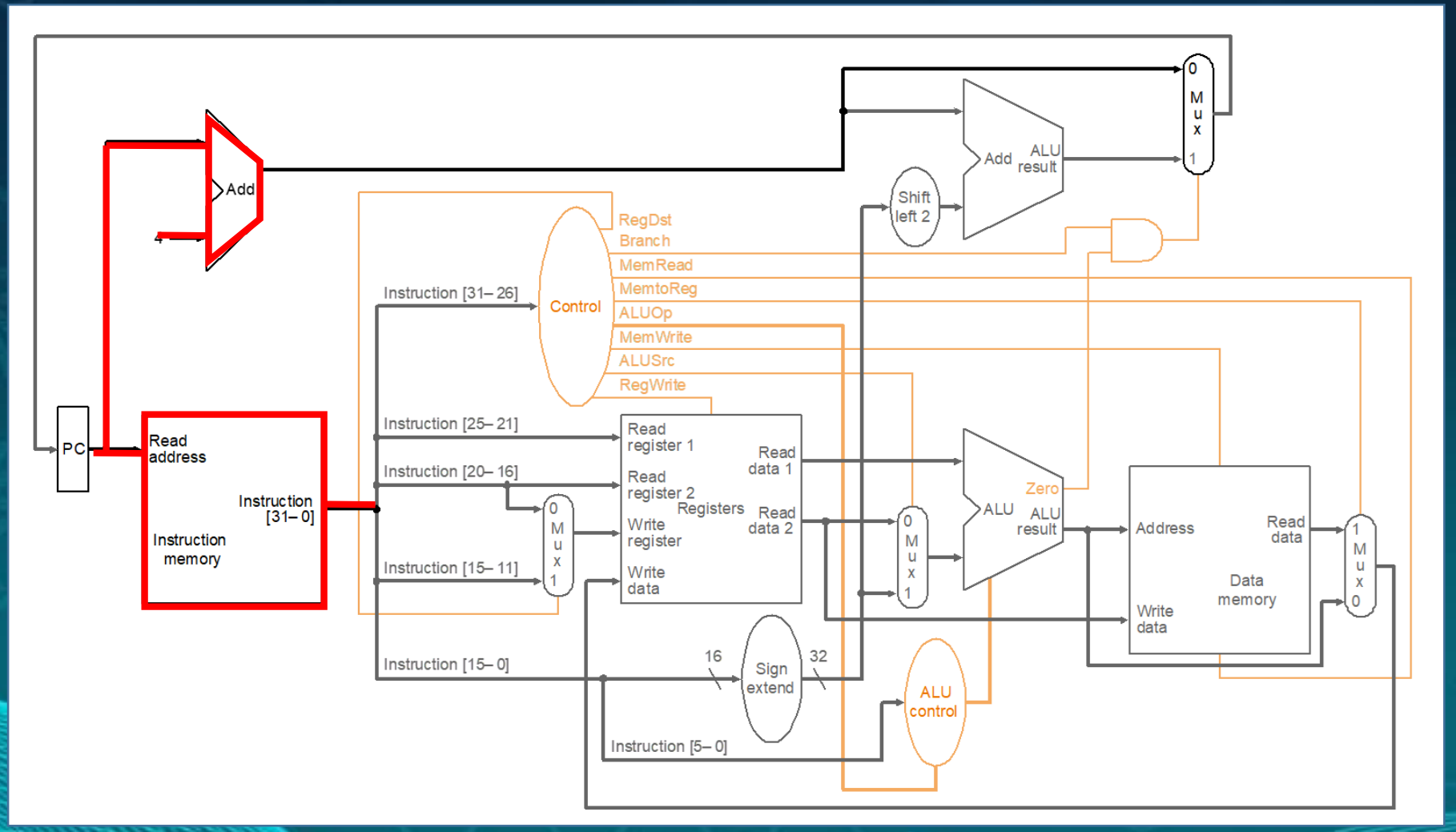
R[rd] ← ALU
PC ← PC+4

Hint for multi-cycle and pipeline design (discussed later)

1. **Fetch the instruction from memory**
2. **Instruction decode and read operands**
3. **Execute the actual operation**
4. **Write back to target register**

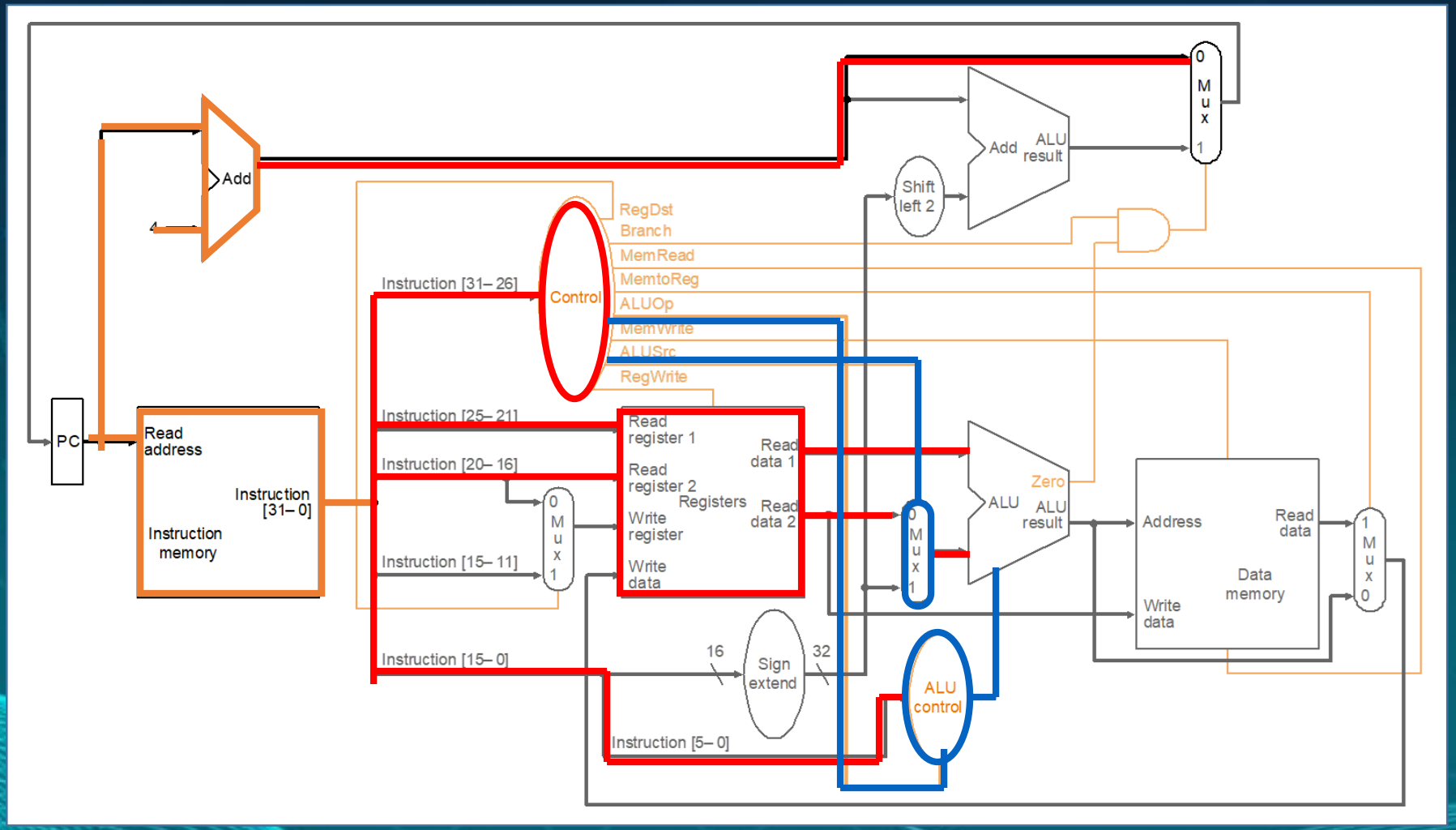
Instruction Fetch at Start of ADD

- instruction ← mem[PC]; PC + 4



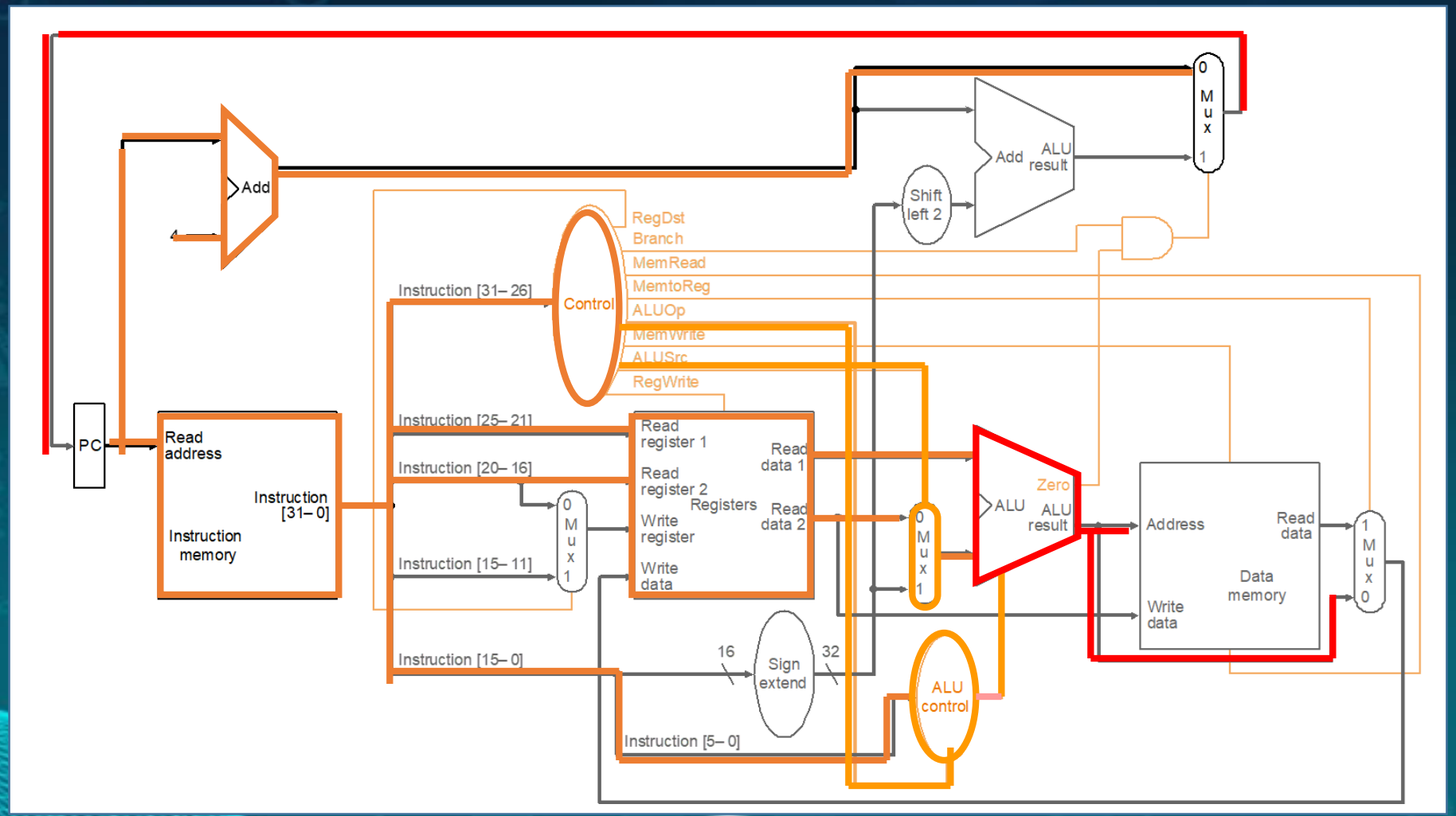
Instruction Decode of ADD

- Fetch the two operands and decode instruction:



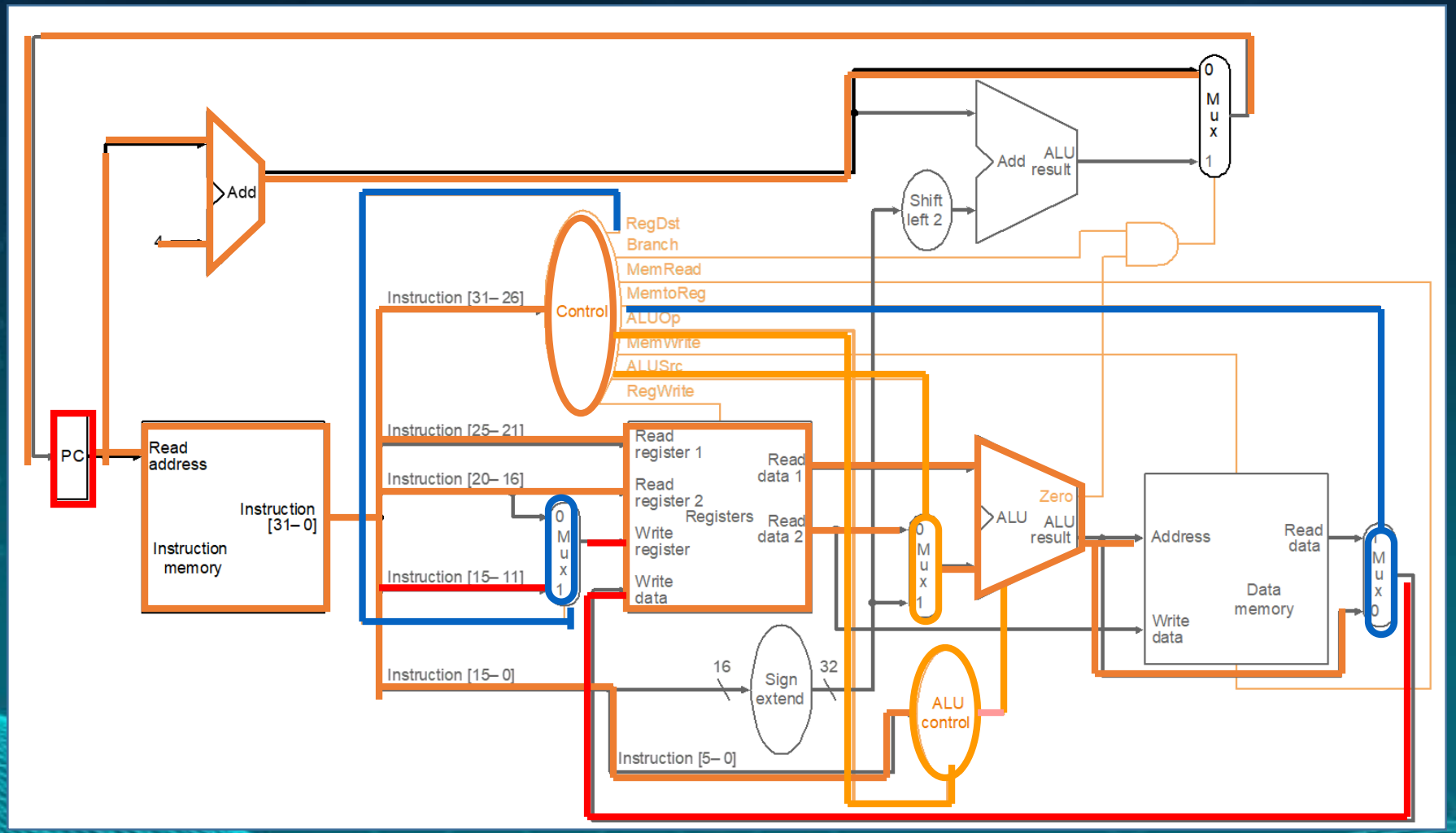
ALU Operation during ADD

- R[rs] + R[rt]



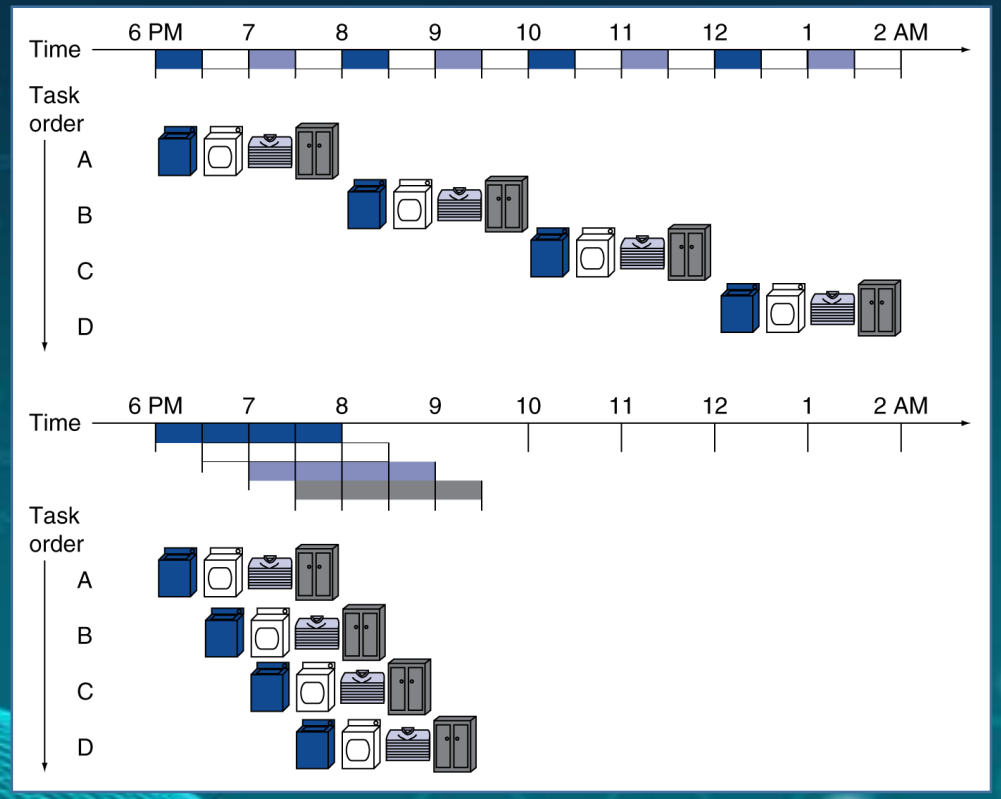
Write Back at the End of ADD

- R[rd] ← ALU; PC ← PC + 4



Pipelining Analogy

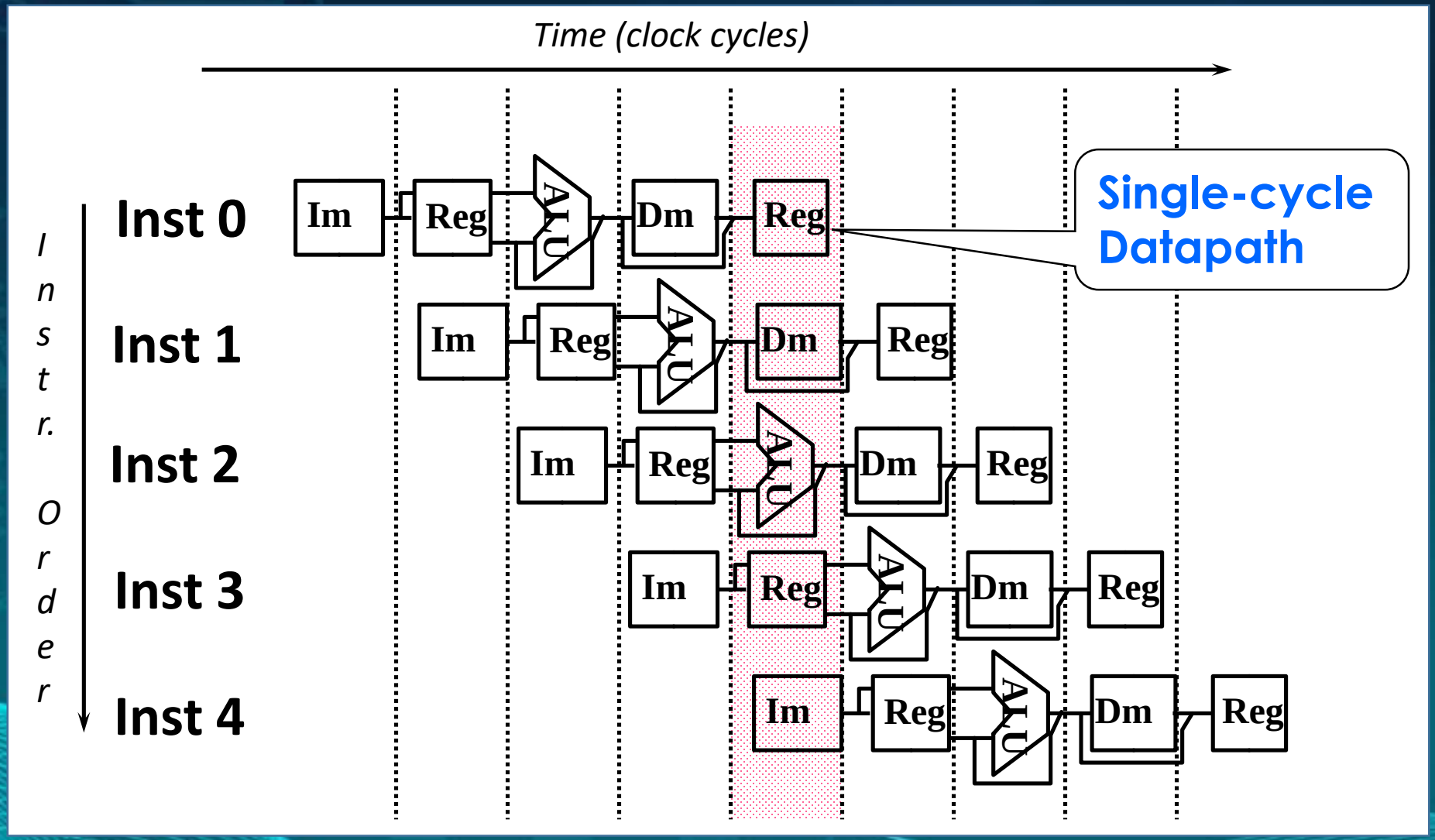
- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup
    = 8/3.5 = 2.3
- Non-stop:
  - Speedup
    = 2n/0.5n + 1.5 ≈ 4
    = number of stages

Why Pipeline?

Because the Resources are There!



MIPS Pipeline

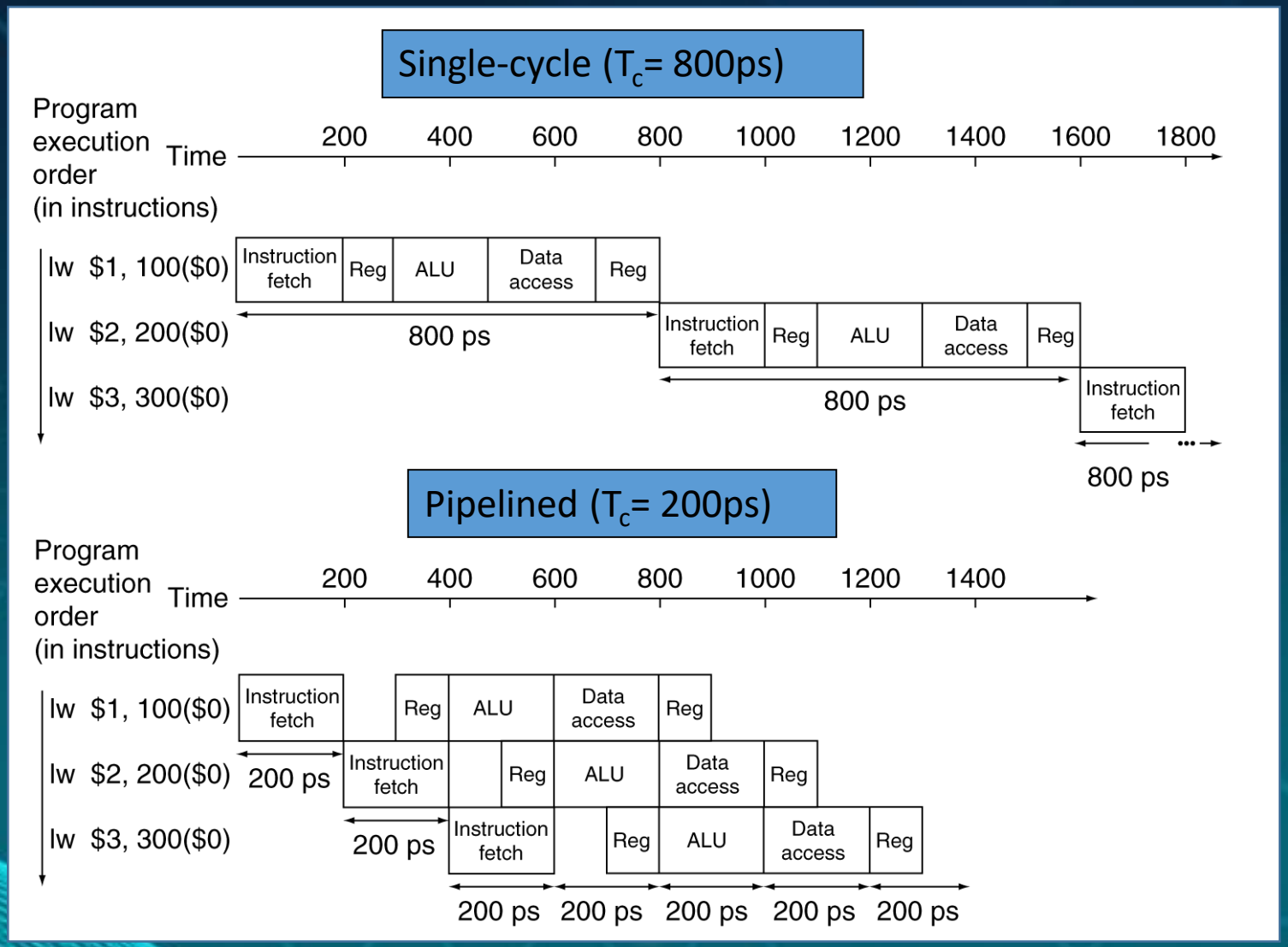
- Five stages, one step per stage
  1. **IF**: Instruction fetch from memory
  2. **ID**: Instruction decode & register read
  3. **EX**: Execute operation or calculate address
  4. **MEM**: Access memory operand
  5. **WB**: Write result back to register

Pipeline Performance

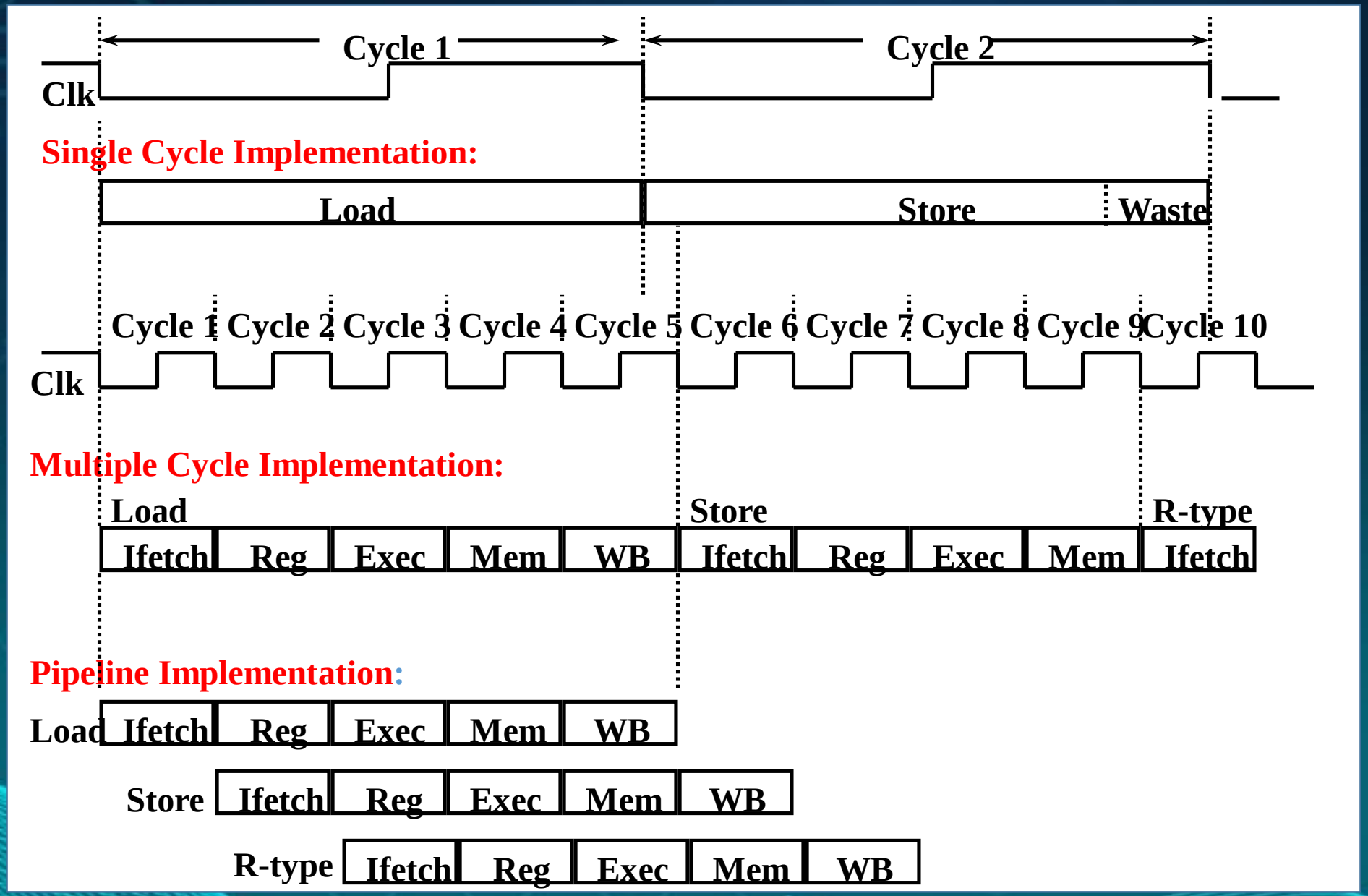
- Assume time for stages is
  - 100ps(0.1 ns) for **register read** or **write**
  - 200ps(0.2 ns) for other stages
- Compare **pipelined datapath** with **single-cycle datapath**

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

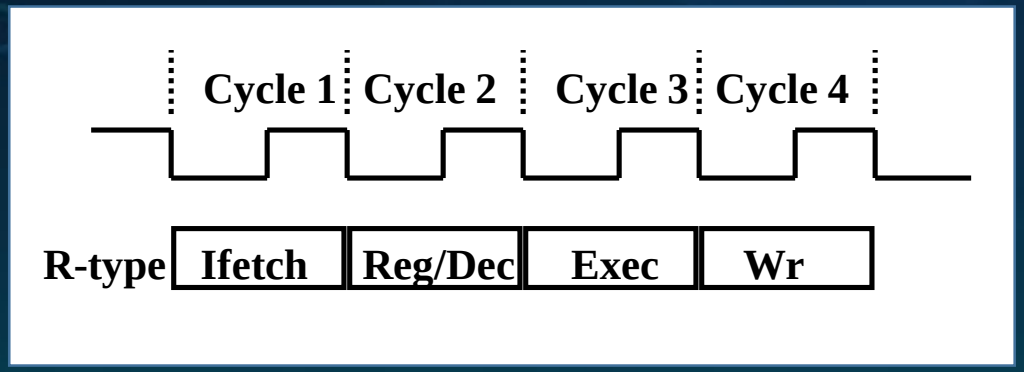
Pipeline Performance



Single-, Multi-Cycle, vs. Pipeline



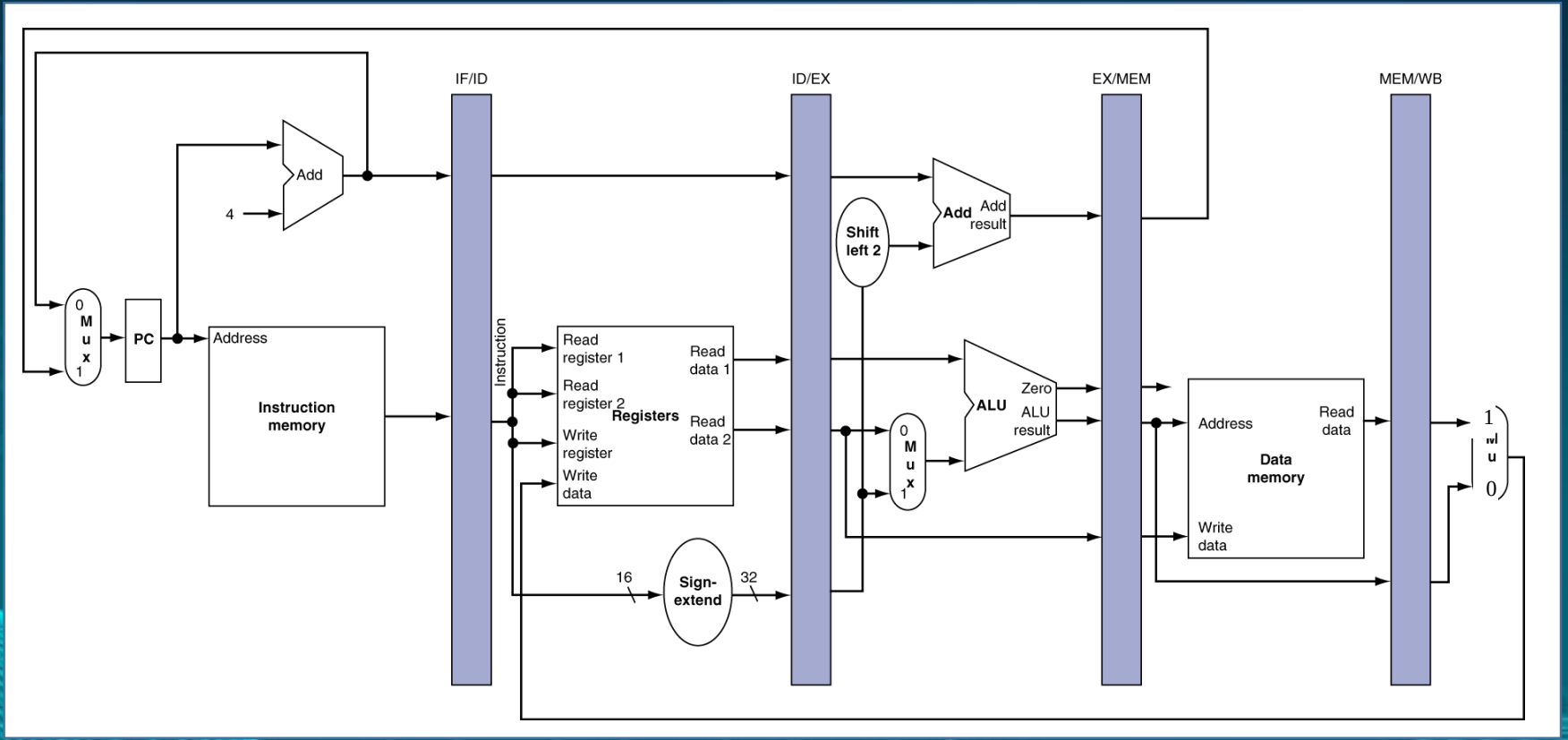
Pipelinging R-type Instructions



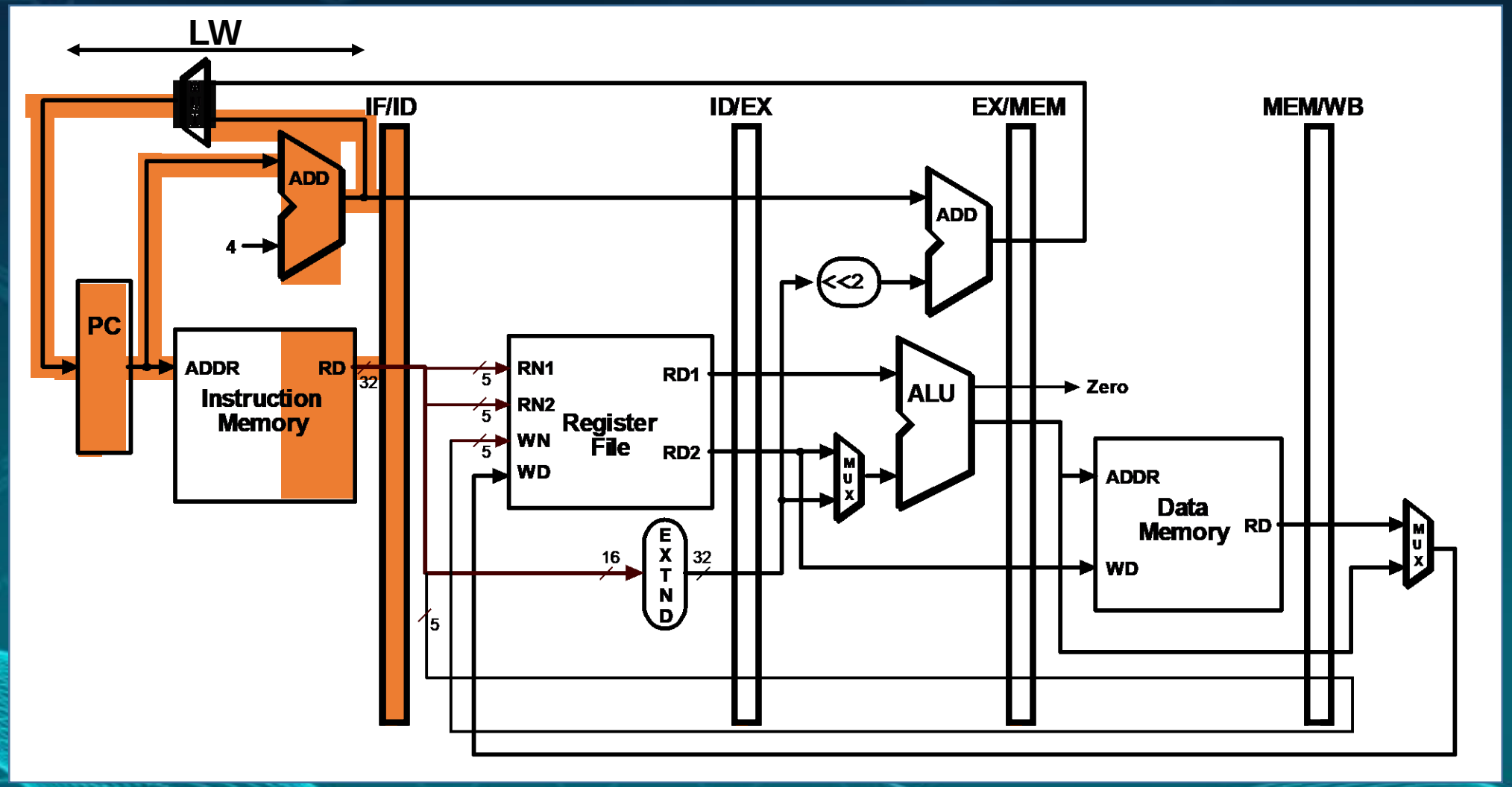
- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX: ALU operates on the two register operands
- WB: write ALU output back to the register file

Pipeline registers

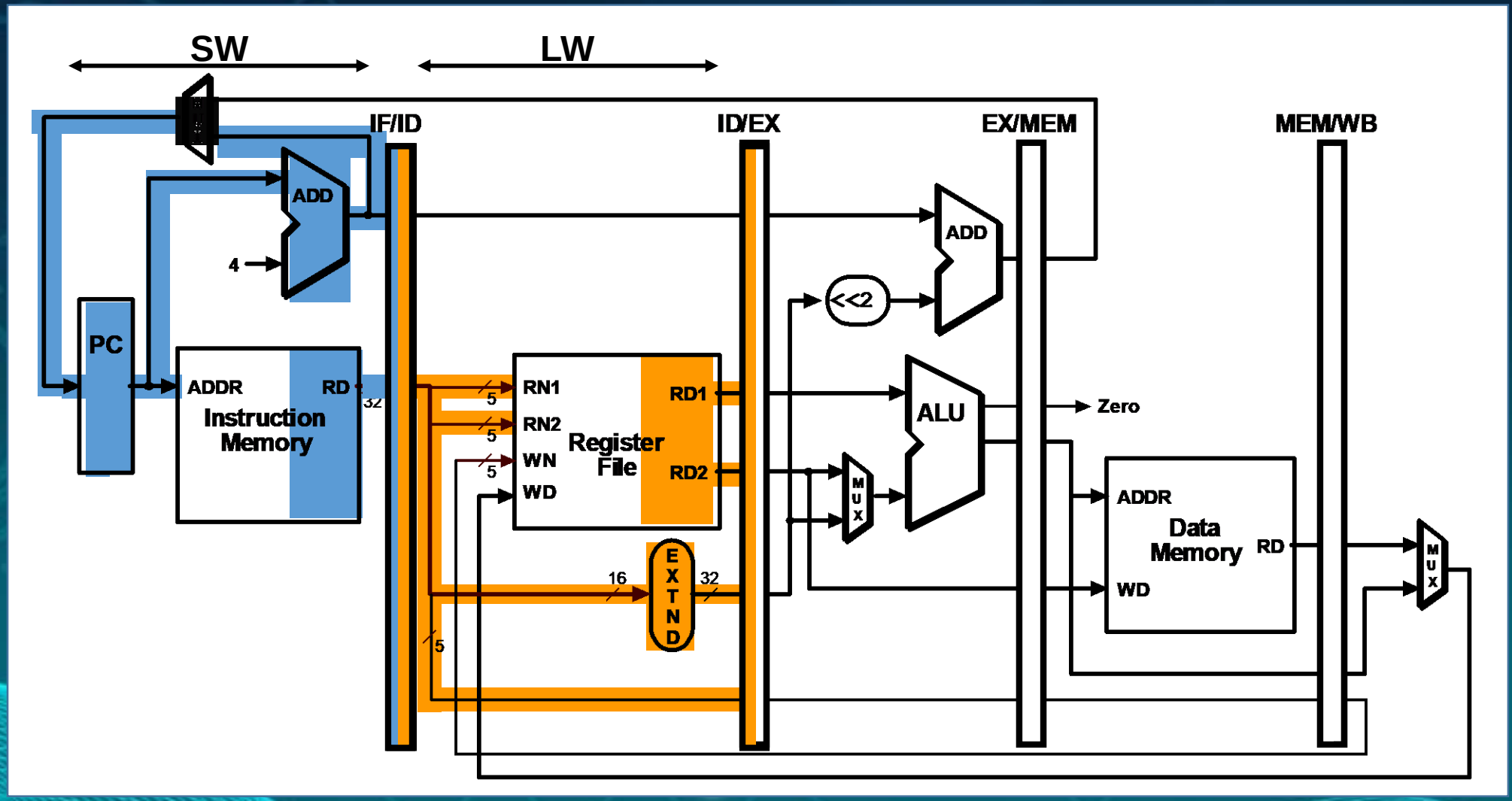
- Need registers between stages
  - To hold information produced in previous cycle



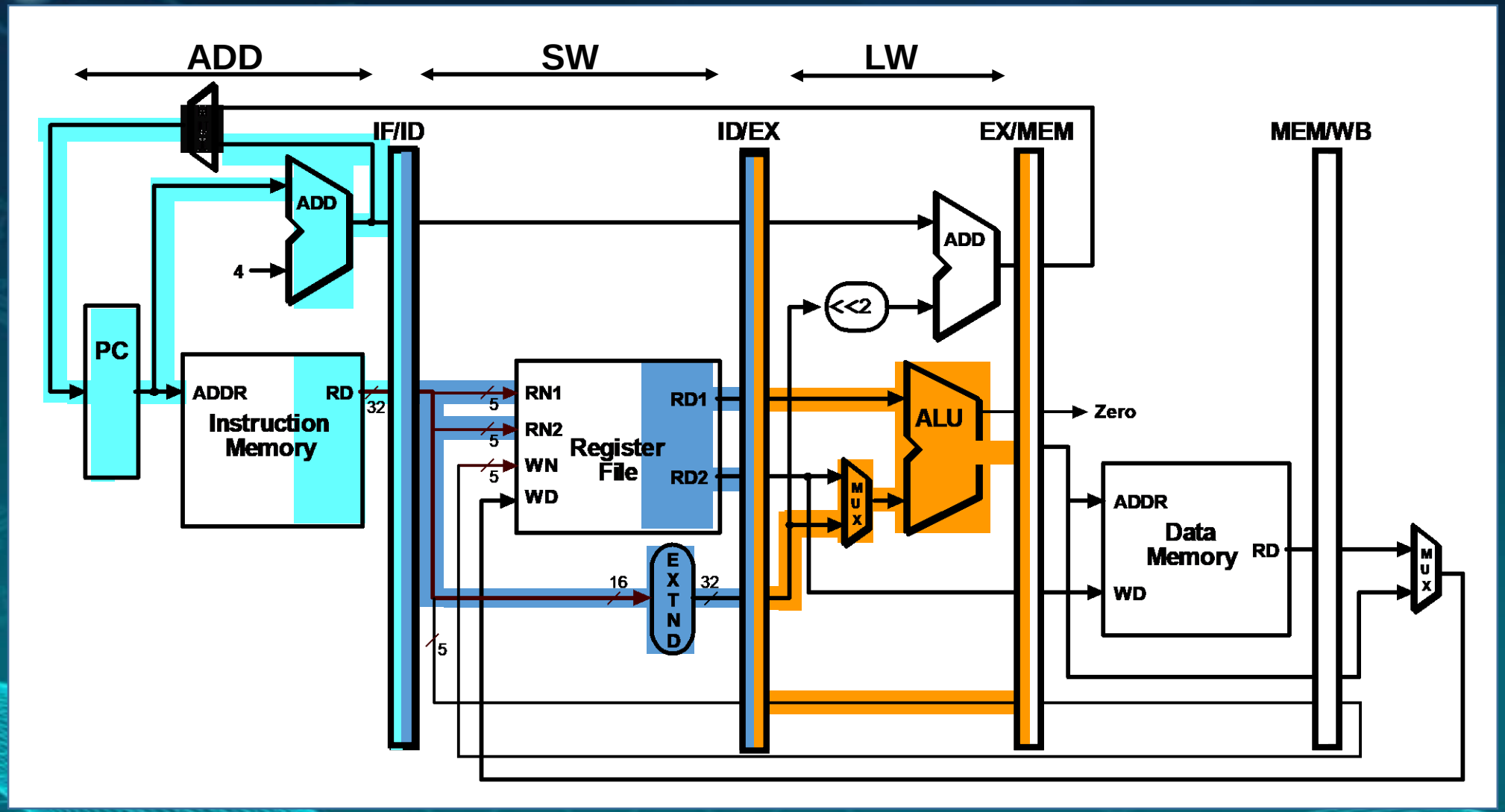
Single-Clock-Cycle Diagram: Clock Cycle 1



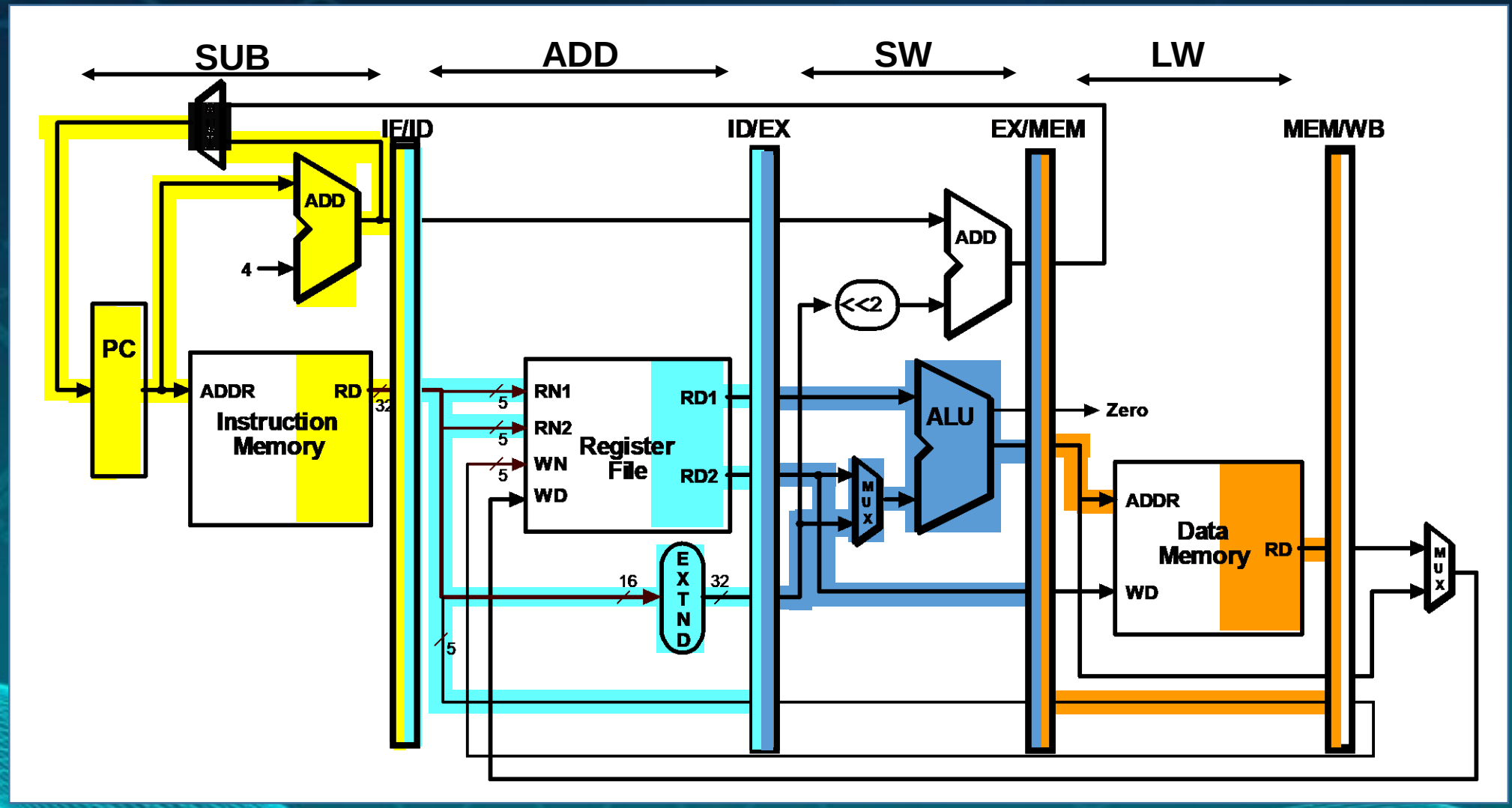
Single-Clock-Cycle Diagram: Clock Cycle 2



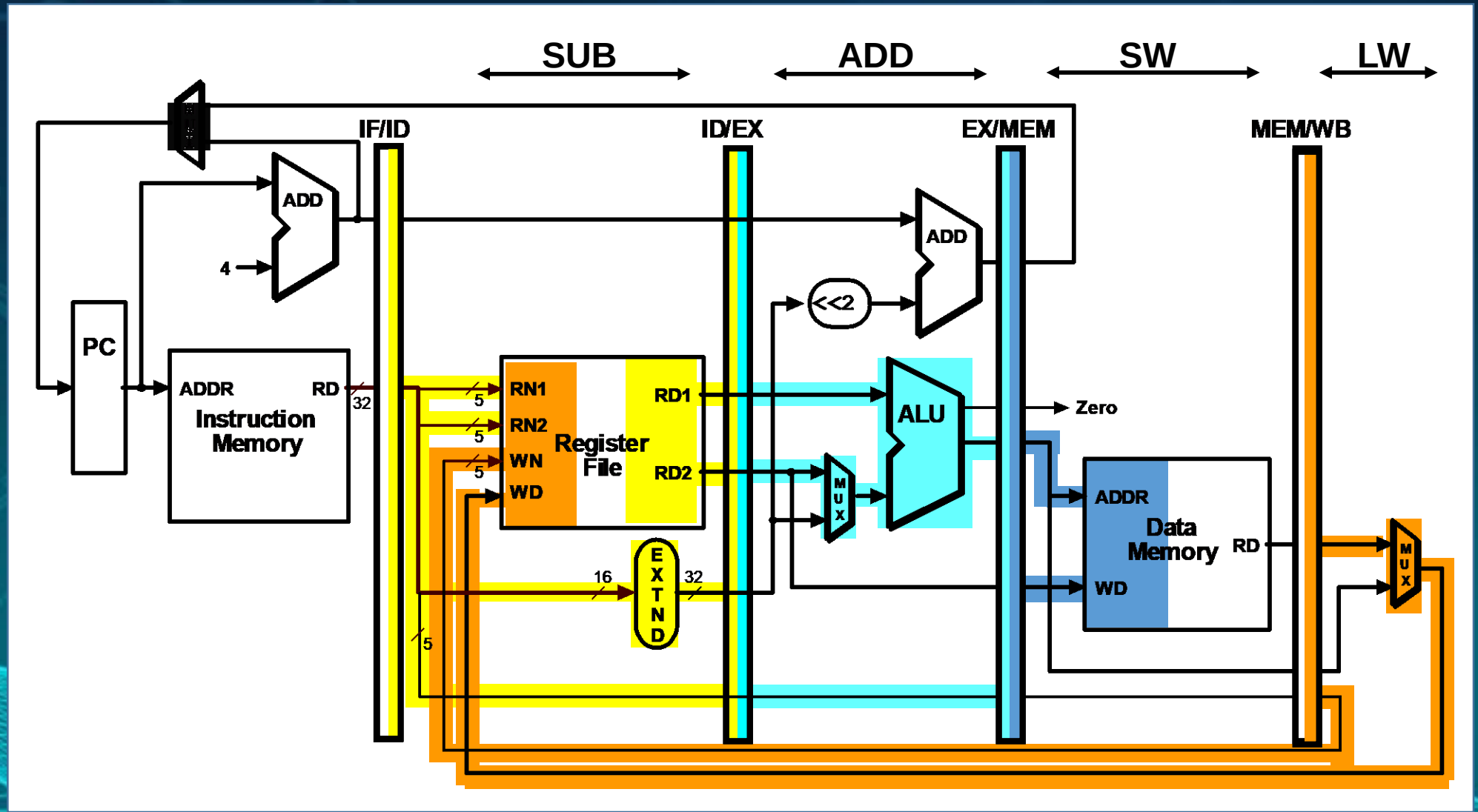
Single-Clock-Cycle Diagram: Clock Cycle 3



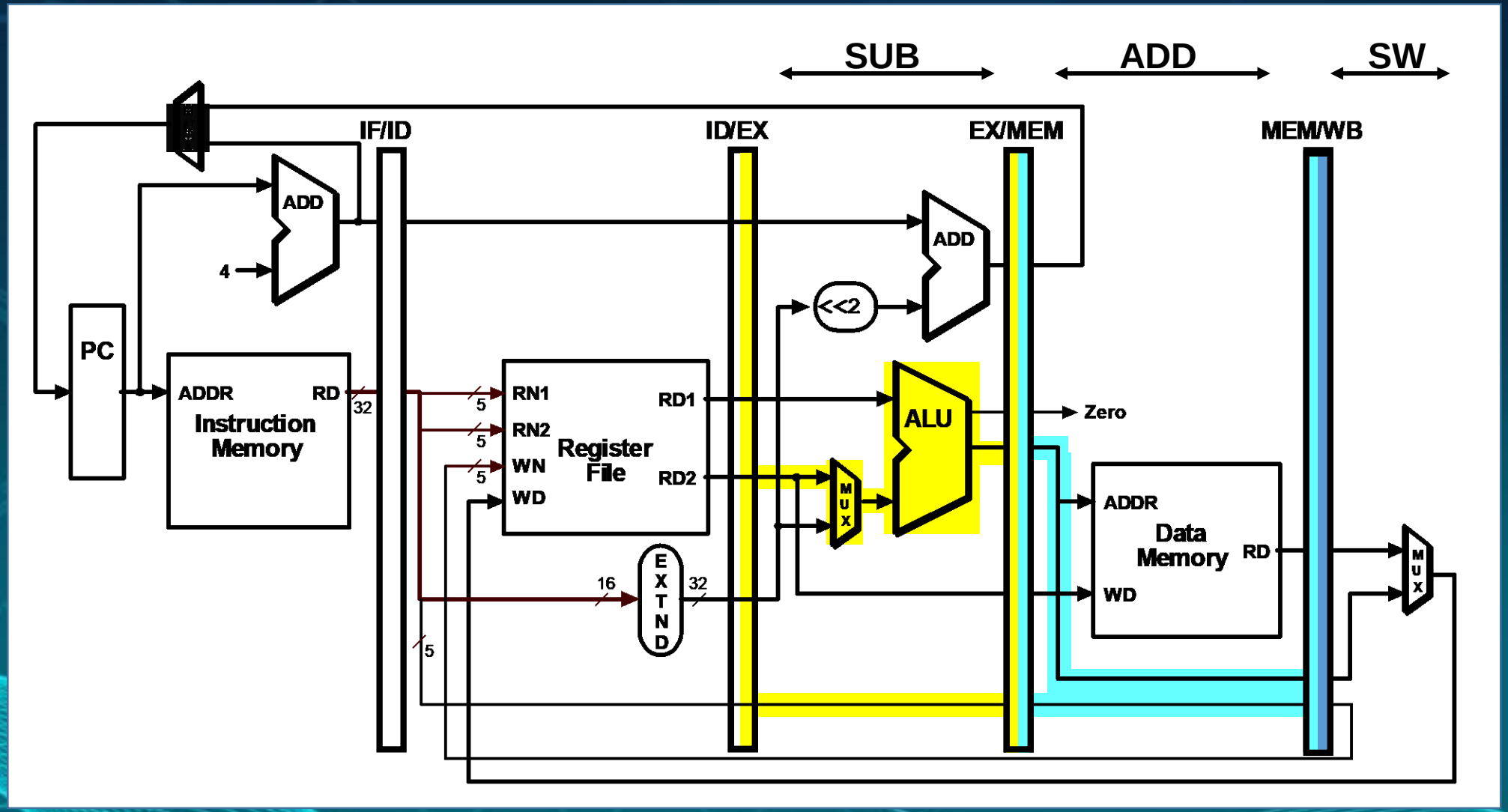
Single-Clock-Cycle Diagram: Clock Cycle 4



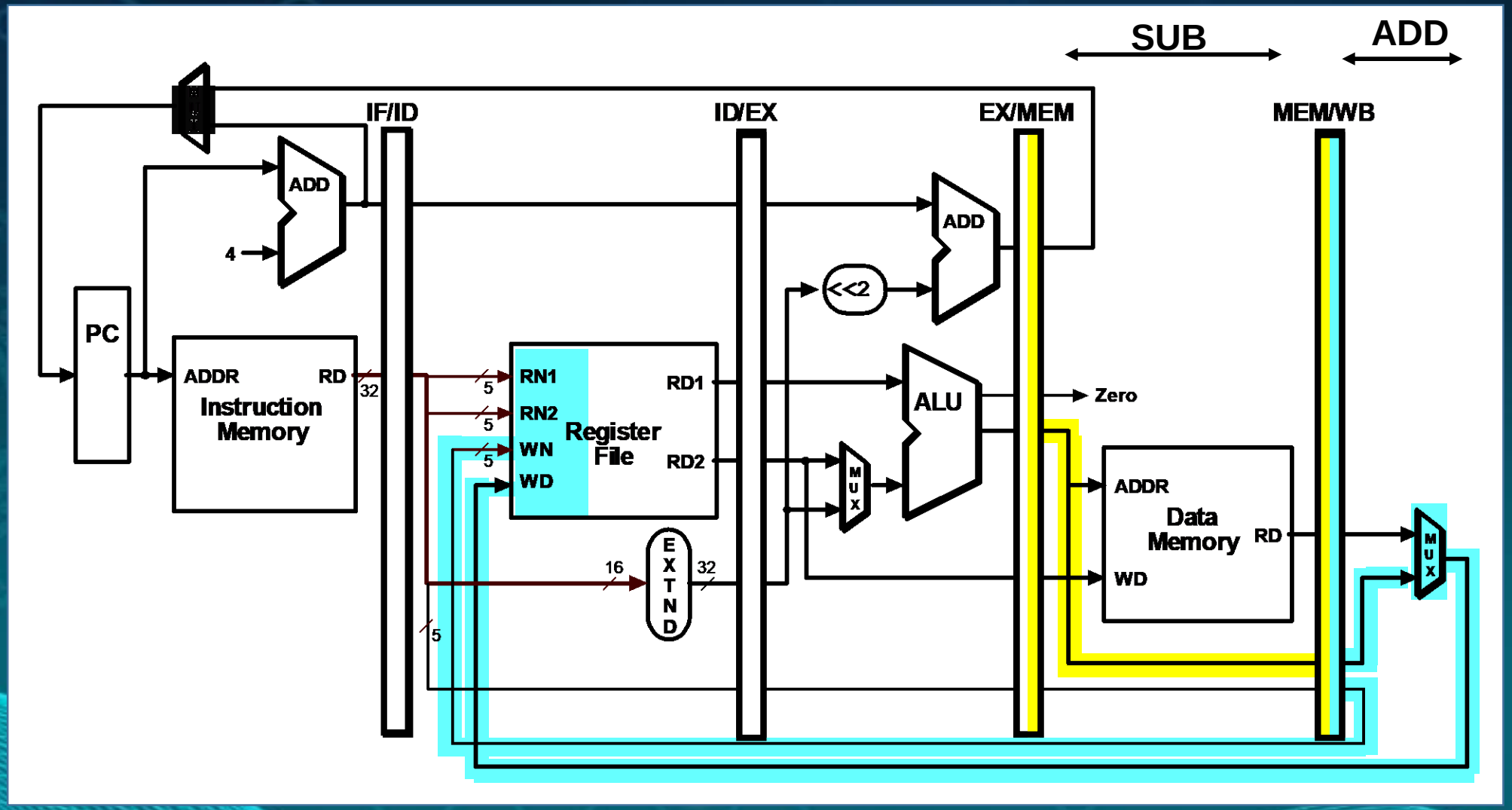
Single-Clock-Cycle Diagram: Clock Cycle 5



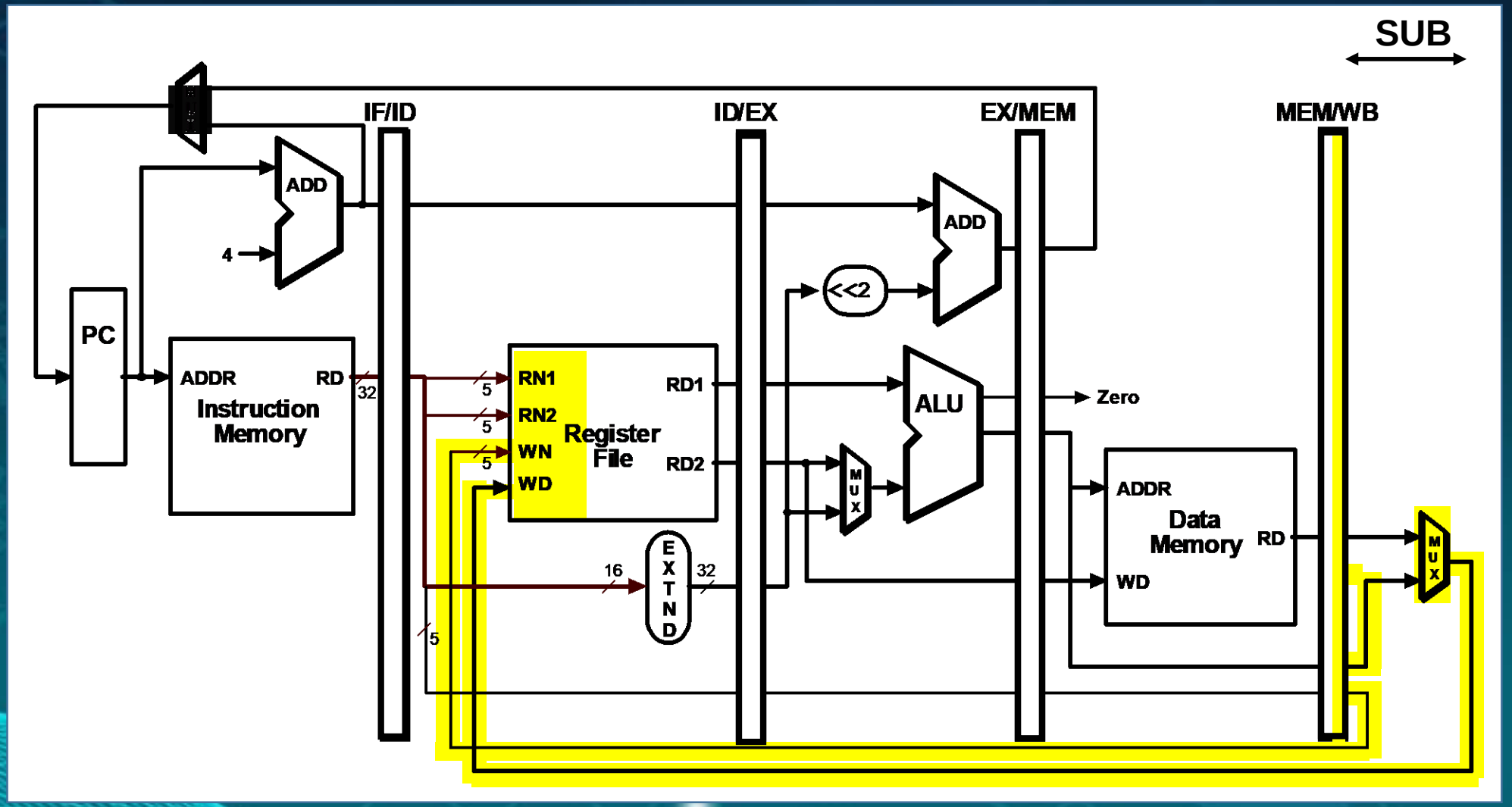
Single-Clock-Cycle Diagram: Clock Cycle 6



Single-Clock-Cycle Diagram: Clock Cycle 7

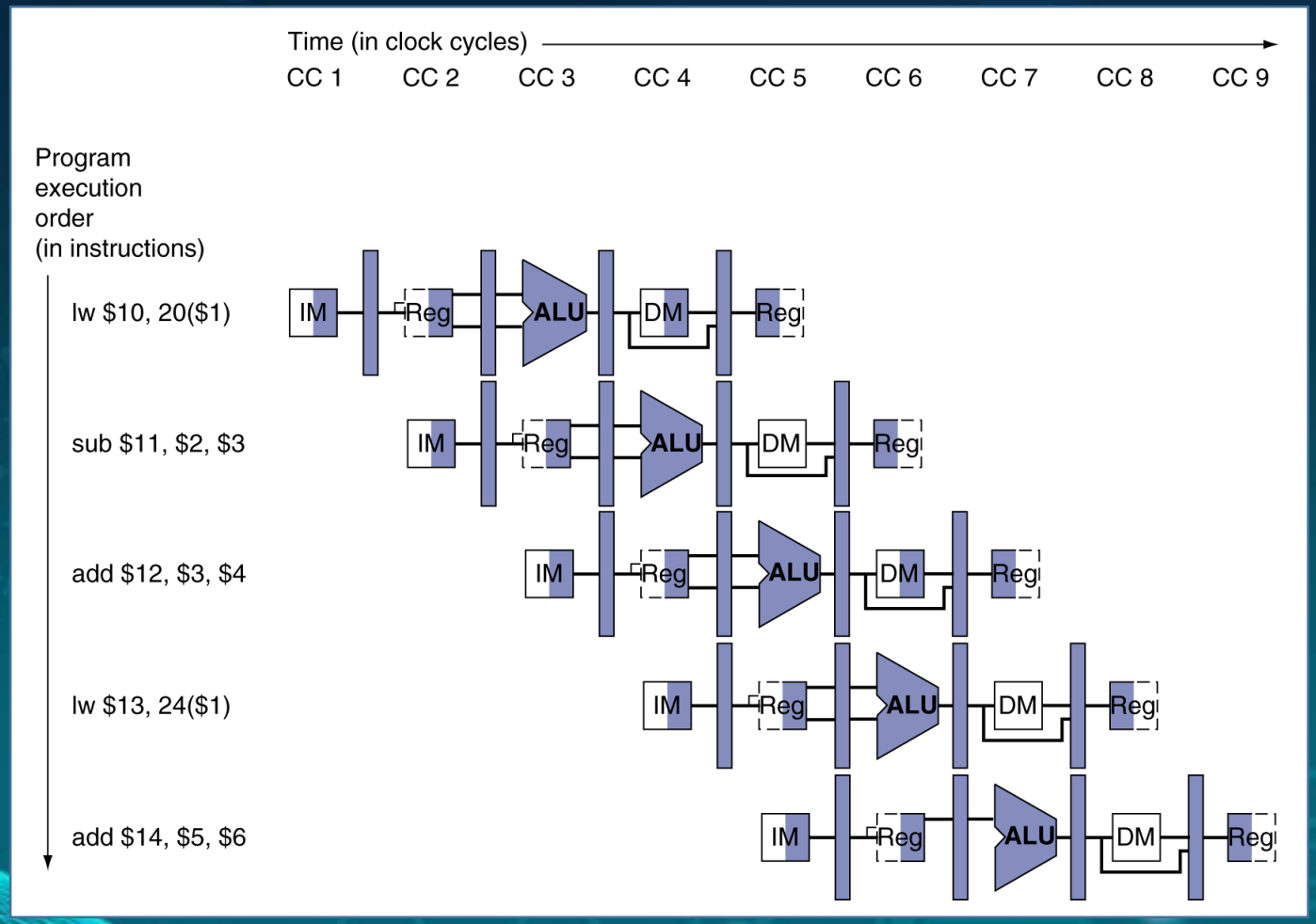


Single-Clock-Cycle Diagram: Clock Cycle 8



Alternative View – Multi-Cycle Pipeline Diagram

- Form showing resource usage



Summary of Pipeline Basics

- Pipelining is a fundamental concept
  - Multiple steps using distinct resources
  - Utilize capabilities of datapath by pipelined instruction processing
    - Start next instruction while working on the current one
    - Limited by length of longest stage
    - Need to detect and resolve hazards

- What makes it easy in MIPS?
  - All instructions are of the same length
  - Just a few instruction formats
  - Memory operands only in loads and stores

- What makes pipelining hard? hazards

Pipeline Hazards

- Pipeline Hazards:
  - Structural hazards: attempt to use the same resource in two different ways at the same time
  - Data hazards: attempt to use item before ready
    - Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: attempt to make decision before condition is evaluated
    - Branch instructions
- Can always resolve hazards by waiting (stalls)
  - pipeline control must detect the hazard
  - take action (or delay/stall action) to resolve hazards

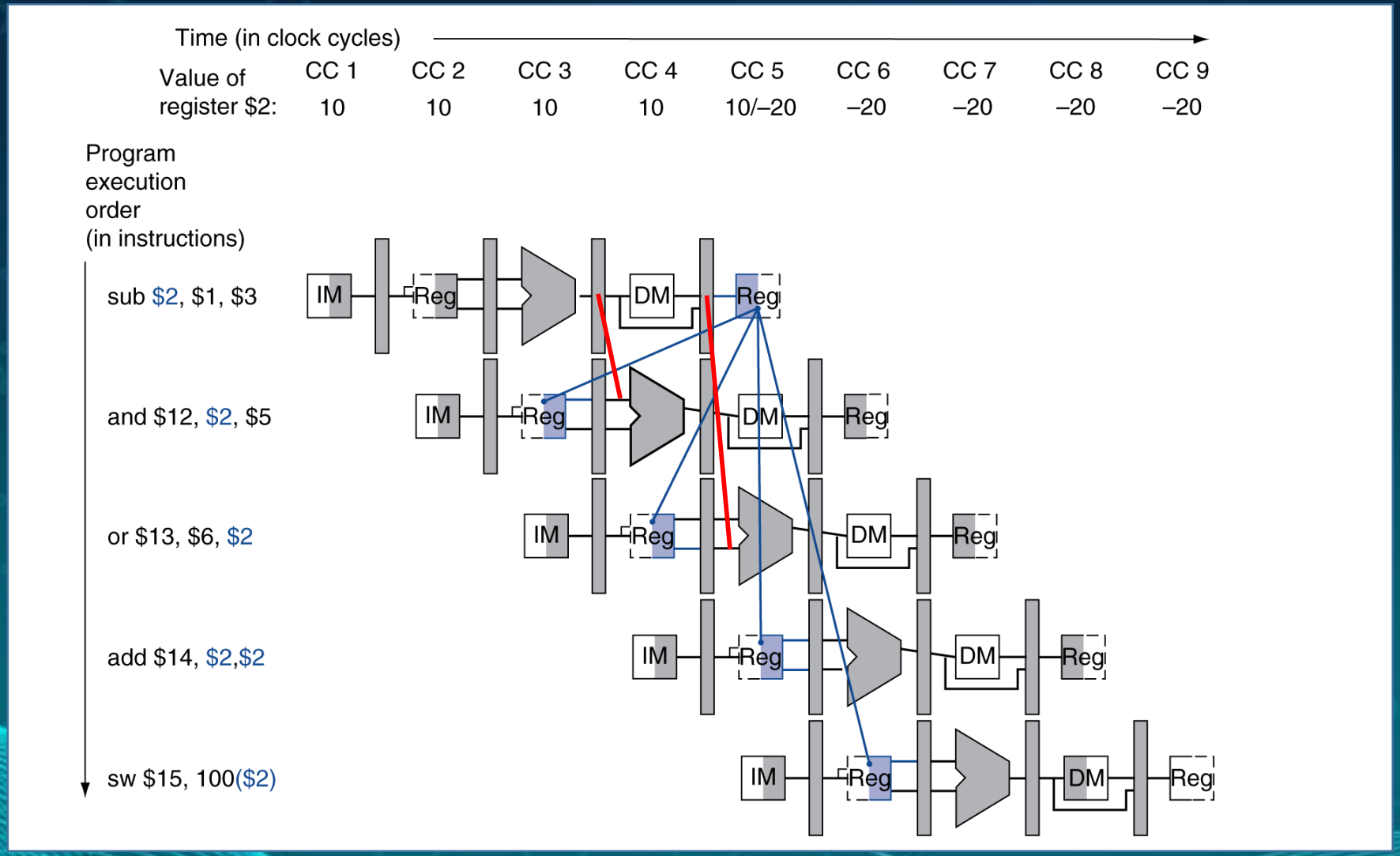
Data Hazards in ALU Instructions

- Consider this sequence:

```
sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```

- We can resolve hazards with **forwarding**
  - How do we detect when to forward?

Dependencies & Forwarding



Hardware Solution: Forwarding

- Idea: *use **intermediate data***, **do not wait for result** to be finally written to the destination register.
- Two steps:
  1. ***Detect*** data hazard
  2. ***Forward*** intermediate data to resolve hazard

Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., **ID/EX.RegisterRs** = register number for Rs sitting in **ID/EX** pipeline register
- ALU operand register numbers in EX stage are given by
  - **ID/EX.RegisterRs**, **ID/EX.RegisterRt**
- Data hazards when
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

Hazard Detection

```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

- Hazard conditions:

  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

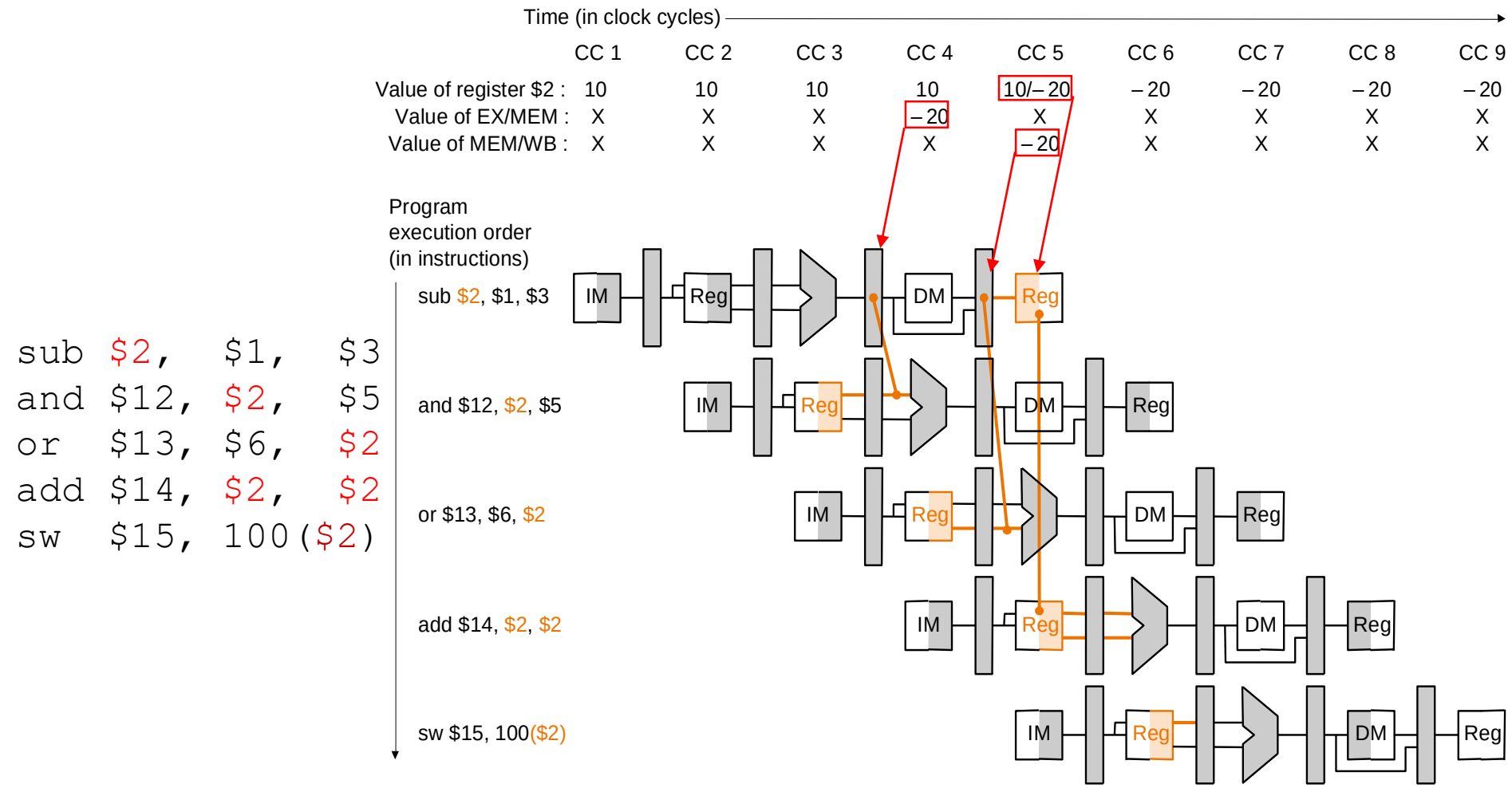
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

  2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

  - E.g., in the earlier example, first hazard between `sub $2, $1, $3` and `and $12, $2, $5` is detected when the **`and`** is in EX stage and the `sub` is in MEM stage because
    - `EX/MEM.RegisterRd = ID/EX.RegisterRs = $2 (1a)`
- Whether to forward also depends on:
  - if the instruction in **EX/MEM** (or **MEM/WB**) is going to **write a register** – if not, no need to forward, even if there is register number match as in conditions above
    - Only if **EX/MEM.RegWrite**, **MEM/WB.RegWrite**
  - if the destination register of the later instruction is $0 – in which case there is no need to forward value ($0 is always 0 and never overwritten)
    - Only if EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0

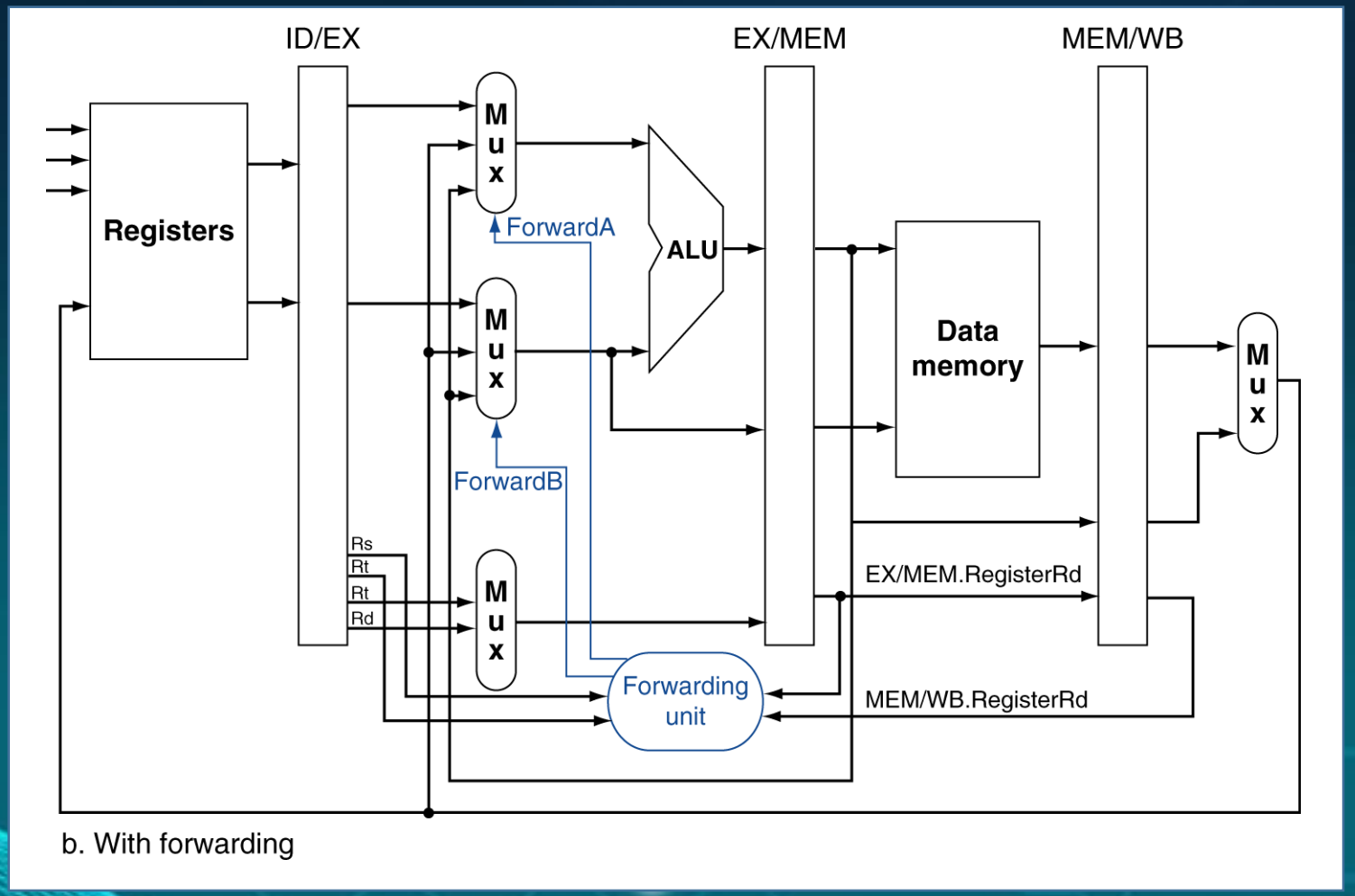
Data Forwarding

```
sub $2,   $1,   $3
and $12,  $2,   $5
or  $13,  $6,   $2
add $14,  $2,   $2
sw  $15,  100($2)
```



Dependencies between pipelines move forward in time

Forwarding Paths



Data Hazard: Detection and Forwarding

- Forwarding unit determines multiplexor control according to the following rules:

1. EX hazard

```
if (      EX/MEM.RegWrite                                  // if there is a write...
    and ( EX/MEM.RegisterRd ≠ 0 )                          // to a non-$0 register...
    and ( EX/MEM.RegisterRd = ID/EX.RegisterRs ) )  // which matches, then...
          ForwardA = 10

if (      EX/MEM.RegWrite                                  // if there is a write...
    and ( EX/MEM.RegisterRd ≠ 0 )                          // to a non-$0 register...
    and ( EX/MEM.RegisterRd = ID/EX.RegisterRt ) )  // which matches, then...
          ForwardB = 10
```

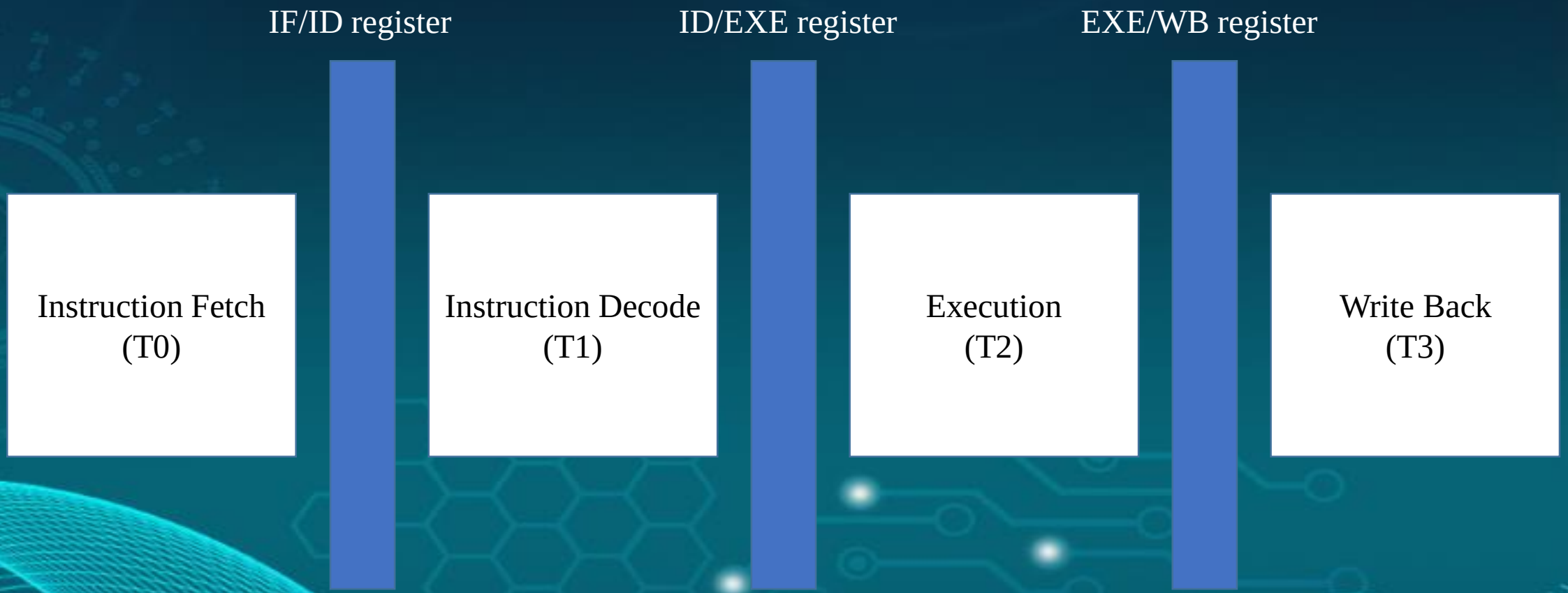
期末專案：simple CPU pipeline實現

繳交規定

- 檢查期限: 1/13 (三) 中午12:30截止
- 報告繳交期限: 1/18 (一) 16:10 上傳至北科i學園PLUS->作業
- 繳交格式:北科i學園PLUS->文件(Document)->微算機系統_報告格式
- 詳細繳交規定請參照**2020 Fall 微算機系統**社團發文

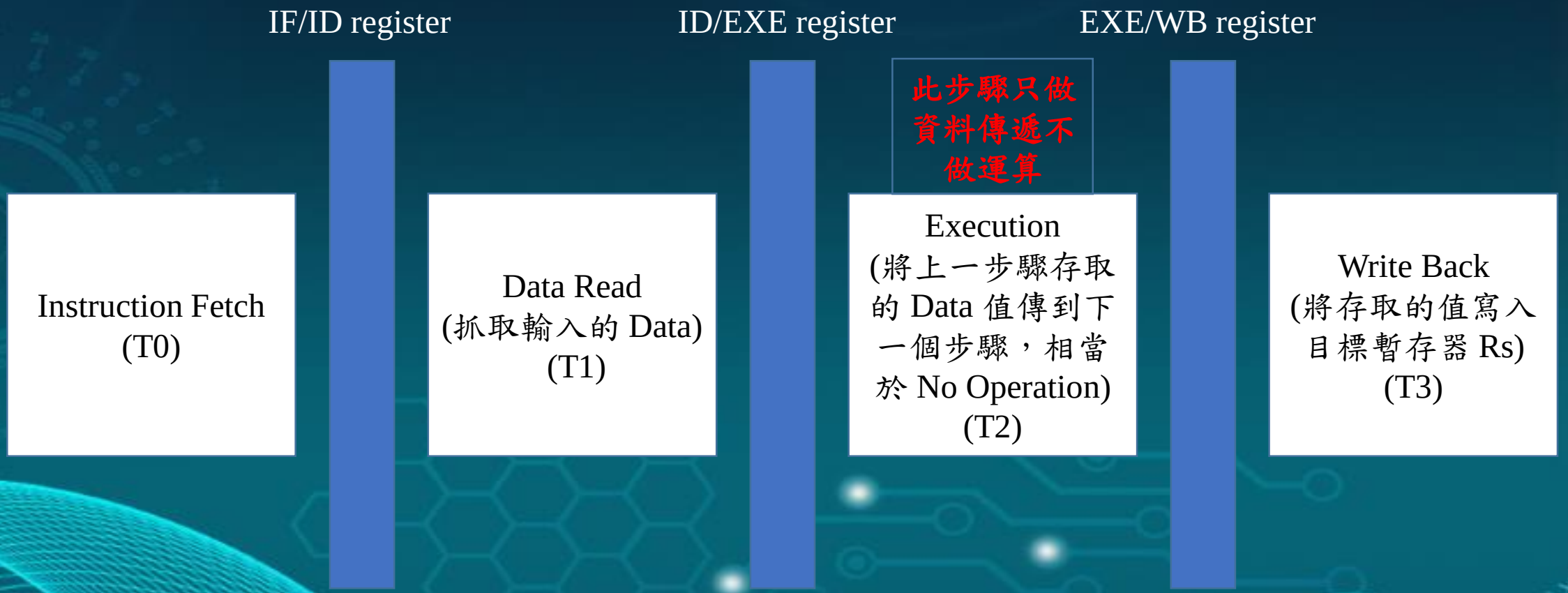
類MIPS R-format指令pipeline流程

- 參照mips架構，設計出R-type指令運作流程。



類MIPS Load指令pipeline流程

- 參照mips架構，設計出 Load 指令運作流程。

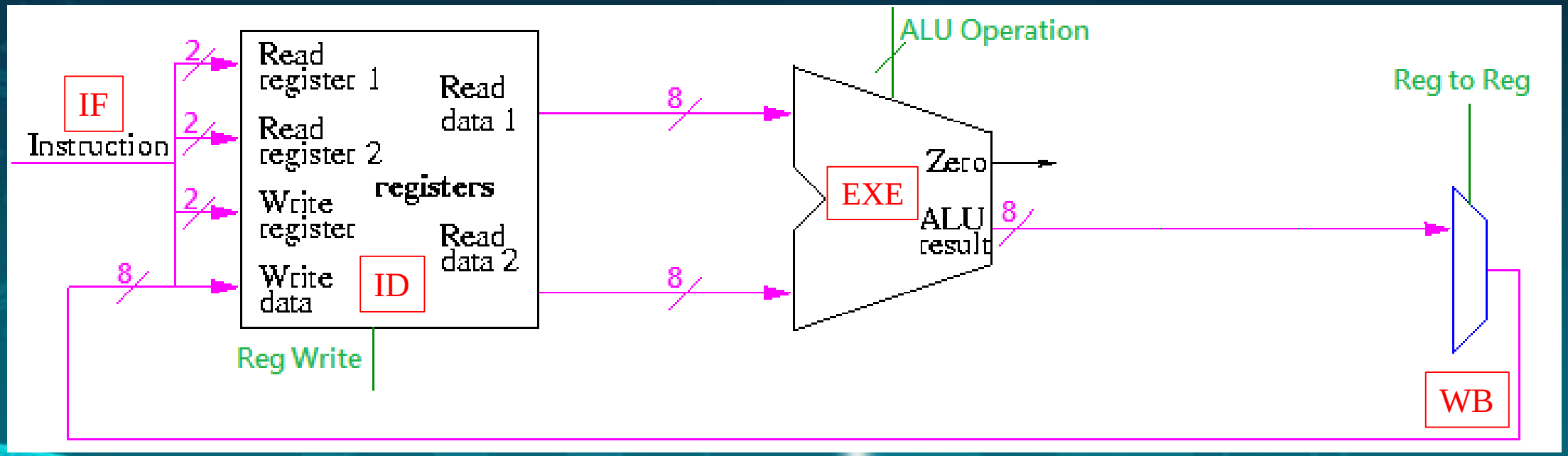


類MIPS Move指令pipeline流程

- 參照mips架構，設計出 Move 指令運作流程。

類MIPS R-format指令pipeline電路圖

- 參照mips架構，設計出R-type指令運作流程。



Hint

- 本專案不使用 Memory Access cycle。
- Pipeline 須包含以下 cycle
  - Instruction Fetch : 解析指令
  - Instruction Decode (register read) : 讀入指令所需暫存器值
  - Execution : ALU運算
  - Write Back : 把運算結果寫回暫存器
- 所需暫存器
  - IF/ID Register
  - ID/EXE Register
  - EXE/WB Register

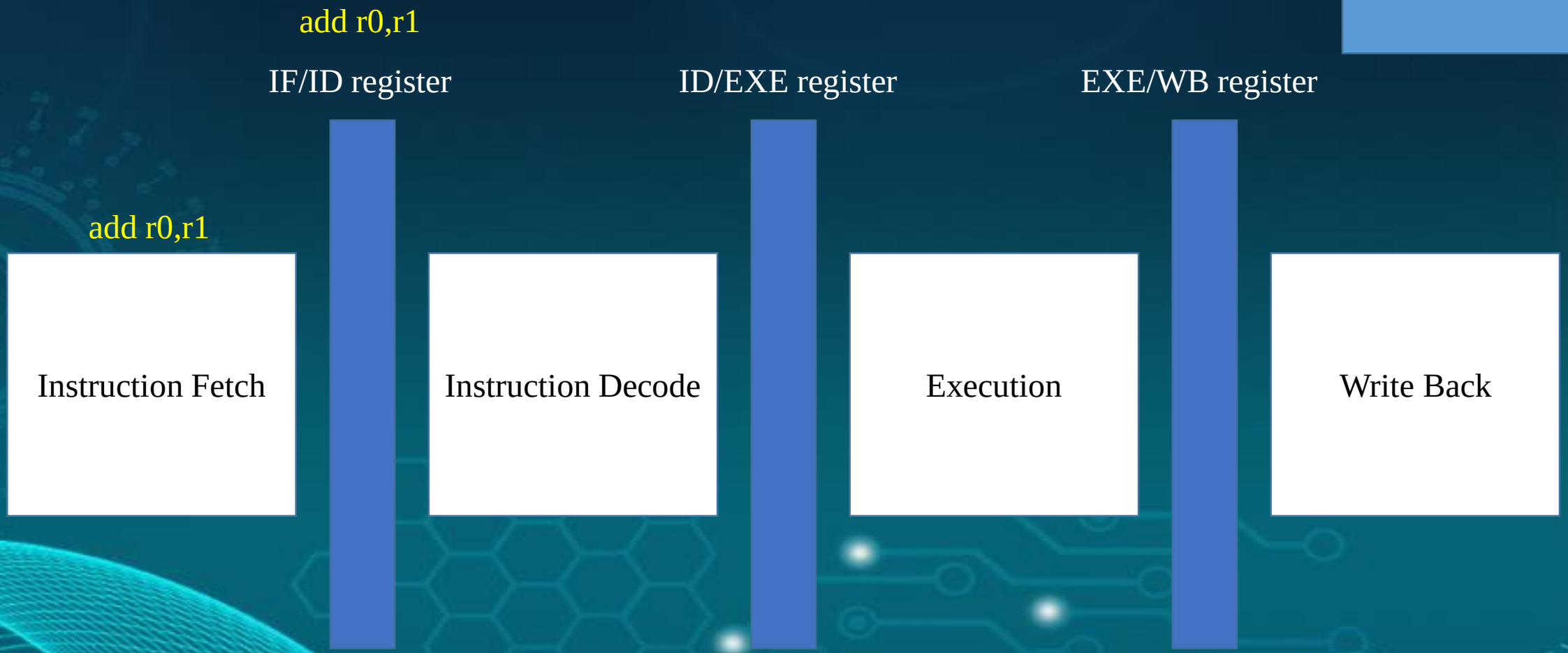
Hazard發生狀況(clock 1)

指令(假設r0= 0 , r1= 1 , r2= 2)

- add r0,r1
- add r0,r2

Register File

r0 = 0
r1 = 1
r2 = 2



Hazard發生狀況(clock 2)

- add r0,r1
- add r0,r2

Register File

r0 = 0
r1 = 1
r2 = 2

從register file讀取值
r0=0
r1=1

add r0,r2

IF/ID register

ID/EXE register

EXE/WB register

add r0,r2

add r0,r1

Hazard發生狀況(clock 3)

- add r0,r1
- add r0,r2

Register File

r0 = 0
r1 = 1
r2 = 2

從register file讀取值

r0=0
r1=1

Hazard 發生

r0=1

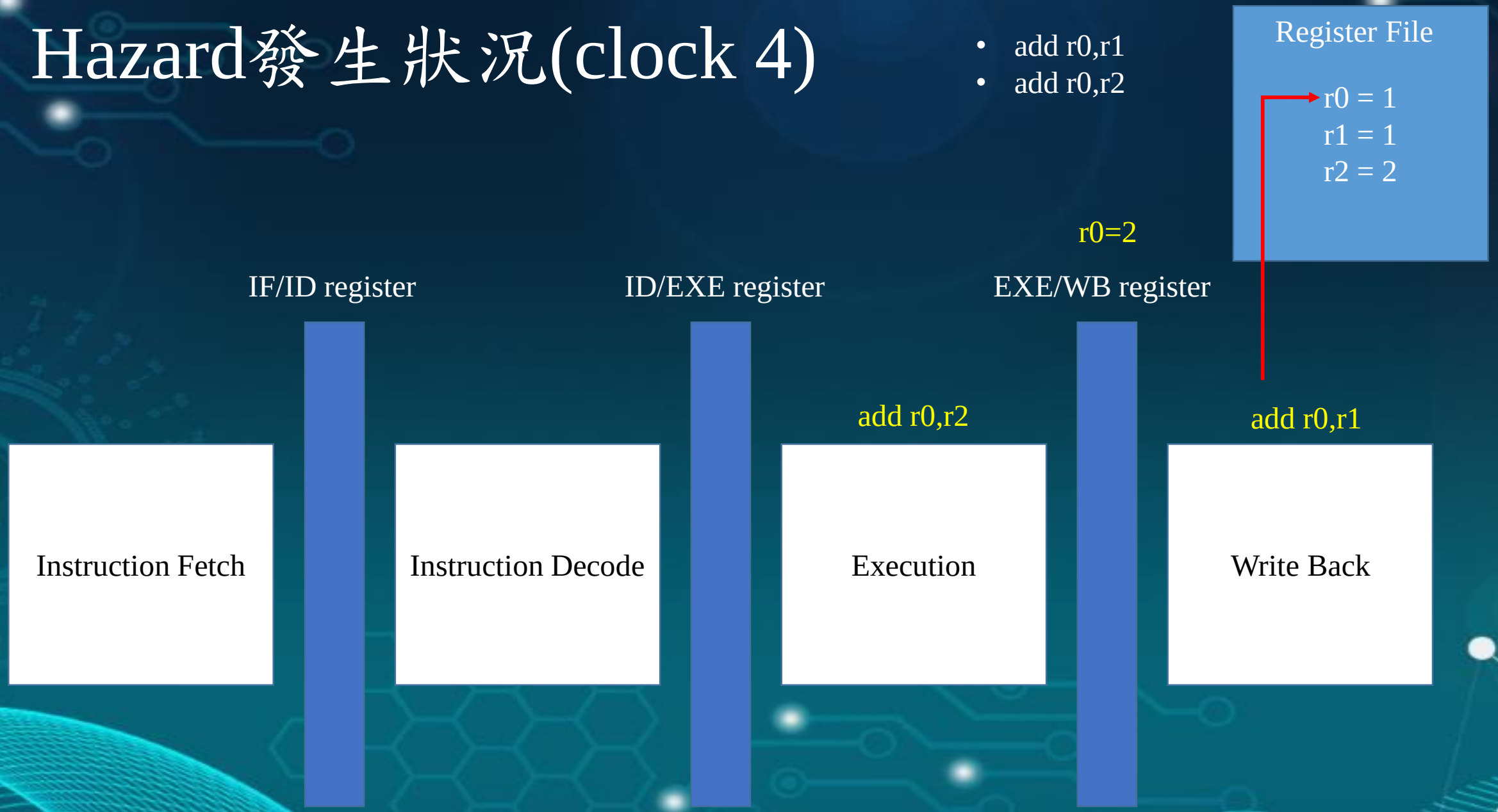
IF/ID register | ID/EXE register | EXE/WB register

add r0,r2 | add r0,r1

Instruction Fetch | Instruction Decode | Execution | Write Back

Hazard發生狀況(clock 4)

- add r0,r1
- add r0,r2



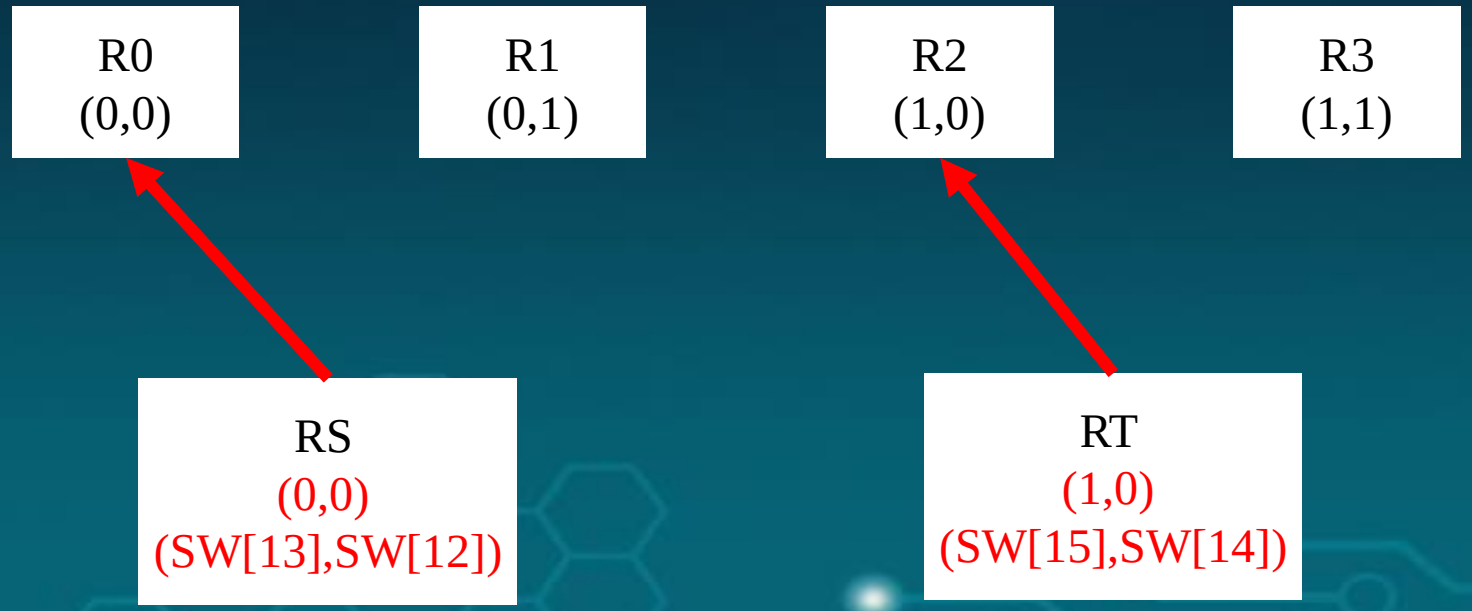
基本功能

- 基本功能有：**LOAD**、**MOVE**、**AND**、**OR**、**ADD**、**SUB**等六個功能，詳細介紹如下表：

運算	中文說明	執行的功能
Load Rs	將Data載入暫存器Rs	Rs←Data
Move Rs, Rt	將暫存器Rt的值覆蓋暫存器Rs	Rs←Rt
And Rs, Rt	以暫存器Rs與Rt之資料bit2bit進行and運算，並將結果儲存於Rs內	Rs←(Rs and Rt)
Or Rs, Rt	以暫存器Rs與Rt之資料bit2bit進行or運算，並將結果儲存於Rs內	Rs←(Rs or Rt)
Add Rs, Rt	將暫存器Rs與暫存器Rt之資料進行數值相加，並儲存於Rs中	Rs←(Rs + Rt)
Sub Rs, Rt	將暫存器Rs與暫存器Rt之資料進行數值相減，並儲存於Rs中	Rs←(Rs + Rt)

RS、RT概念

- RS和RT分別指到要做動作的暫存器(七段顯示器須同步更新)，例如Add R0, R2 就要將RS指到R0，RT則指到R2後，opcode為0010來進行Add的動作。



專案要求

- 所有功能皆須以pipeline實現才有分數。
- 總共有四個暫存器(R0,R1,R2,R3)，使用RS和RT分別指定要使用的暫存器(七段顯示器須同步更新)。
- 使用七段顯示器顯示 Rs、Rt 以及 Data 的值。
- Load 是必要完成的功能，需實所有功能本次實驗才有分數。

專案要求－七段顯示器的使用

- 七段顯示器顯示為十六進制，兩顆七段顯示器為一組，範圍為1~FF (超過FF顯示末兩位)。
- 左邊兩組顯示 Rt 及 Rs 兩個暫存器。
- 最右邊一組則持續顯示 Data 的數值
- 執行指令時，若暫存器的內容有被改變，則七段顯示器須同步更新。
- 七段顯示器的顯示範圍為 0-FF。

專案要求－LED 燈的使用

- 在執行各 Cycle 時，須以綠色 LED 燈顯示該 Cycle 是否正在執行。
- 偵測 Data Hazard時，須以紅色 LED 燈顯示是否有 Hazard 產生。

按鈕開關的使用(僅供參考)

- 當指撥開關接設定好後，須使用按鈕開關將其輸入，在這裡，指令和暫存器(或Data)的輸入是使用指令時序的方式，也就是說指撥開關設定好後，其運作程序範例如下(使用clock觸發)：

1. 依照instruction code來判斷，輸入指定暫存器與執行指令，並顯示更新暫存器的數值。
2. 完整功能表中的 No instruction 功能，指的是此 clock 不再輸入 Instruction 的狀態。

指撥開關的使用(完整功能)

- 指撥開關以 8bits 輸入 Instruction code (4bits為 Opcode，2bits 為 Rs, 2bit 為 Rt)，8Bits 指定 Data，說明如下：

Instruction	Instruction code			usage	Result
	opcode	Rs	Rt		
Load	0000	2bit	2bit	Load Rs	Rs←Data
Move	0001			Move Rs, Rt	Rs←Rt
Add	0010			Add Rs, Rt	Rs←Rs + Rt
Sub	0011			Sub Rs, Rt	Rs←Rs – Rt
And	0100			And Rs, Rt	Rs←Rs & Rt
Or	0101			Or Rs, Rt	Rs←Rs | Rt
Nor	0110			Nor Rs, Rt	Rs←Rs Nor Rt
Slt	0111			Slt Rs, Rt	if (Rs<Rt) Rs=1;else Rs=0
No instruction fetch	1111				No instruction fetch

實驗配分

- 紫色與黃色區塊皆須以pipeline實現，共 50%

- Hazard 偵測與解決，共 30% ：
  - 偵測 data hazard + 使用NOP解決 15%
  - 使用 forward 解決 hazard 15%

- 報告 30%

功能	得分
基本 Pipeline 實現與 Load/Move	30%
Add	20% (左側功能皆須以 Pipeline 實現才有得分)
Sub	
And	
Or	
Nor	
Slt	

指定腳位 – 輸入

Name	Pin Location
Data (0)	PIN_AB28 **(SW[0])**
Data (1)	PIN_AC28 **(SW[1])**
Data (2)	PIN_AC27 **(SW[2])**
Data (3)	PIN_AD27 **(SW[3])**
Data (4)	PIN_AB27 **(SW[4])**
Data (5)	PIN_AC26 **(SW[5])**
Data (6)	PIN_AD26 **(SW[6])**
Data (7)	PIN_AB26 **(SW[7])**

Name	Pin Location
Instruction code opcode (0)	PIN_AC25 (**SW[8]**)
Instruction code opcode (1)	PIN_AB25 (**SW[9]**)
Instruction code opcode (2)	PIN_AC24 (**SW[10]**)
Instruction code opcode (3)	PIN_AB24 (**SW[11]**)
Instruction code RS (4)	PIN_AB23 (**SW[12]**)
Instruction code RS (5)	PIN_AA24 (**SW[13]**)
Instruction code RT (6)	PIN_AA23 (**SW[14]**)
Instruction code RT (7)	PIN_AA22 (**SW[15]**)

Name	Pin Location
Clock	PIN_M23 **(KEY[0])**

指定腳位 – 七段顯示器 Data

Name	Pin Location
HEX1[0]	PIN_M24
HEX1[1]	PIN_Y22
HEX1[2]	PIN_W21
HEX1[3]	PIN_W22
HEX1[4]	PIN_W25
HEX1[5]	PIN_U23
HEX1[6]	PIN_U24

Data 十位輸出

Name	Pin Location
HEX0[0]	PIN_G18
HEX0[1]	PIN_F22
HEX0[2]	PIN_E17
HEX0[3]	PIN_L26
HEX0[4]	PIN_L25
HEX0[5]	PIN_J22
HEX0[6]	PIN_H22

Data 個位輸出

指定腳位 – 七段顯示器 Rs

Name	Pin Location
HEX3[0]	PIN_V21
HEX3[1]	PIN_U21
HEX3[2]	PIN_AB20
HEX3[3]	PIN_AA21
HEX3[4]	PIN_AD24
HEX3[5]	PIN_AF23
HEX3[6]	PIN_Y19

Rs 十位輸出

Name	Pin Location
HEX2[0]	PIN_AA25
HEX2[1]	PIN_AA26
HEX2[2]	PIN_Y25
HEX2[3]	PIN_W26
HEX2[4]	PIN_Y26
HEX2[5]	PIN_W27
HEX2[6]	PIN_W28

Rs 個位輸出

指定腳位 – 七段顯示器 Rt

Name	Pin Location
HEX5[0]	PIN_AD18
HEX5[1]	PIN_AC18
HEX5[2]	PIN_AB18
HEX5[3]	PIN_AH19
HEX5[4]	PIN_AG19
HEX5[5]	PIN_AF18
HEX5[6]	PIN_AH18

Rt 十位輸出

Name	Pin Location
HEX4[0]	PIN_AB19
HEX4[1]	PIN_AA19
HEX4[2]	PIN_AG21
HEX4[3]	PIN_AH21
HEX4[4]	PIN_AE19
HEX4[5]	PIN_AF19
HEX4[6]	PIN_AE18

Rt 個位輸出

指定腳位 – LED 燈

Name	Pin Location
LEDR[0] (Hazard Detection)	PIN_G19

Hazard 偵測

Name	Pin Location
LEDG[0] (Instruction Fetch)	PIN_E21
LEDG[1] (Instruction Decode)	PIN_E22
LEDG[2] (Execution)	PIN_E25
LEDG[3] (Write Back)	PIN_E24

Cycle 執行狀況