Bilgisayar Mimarisi

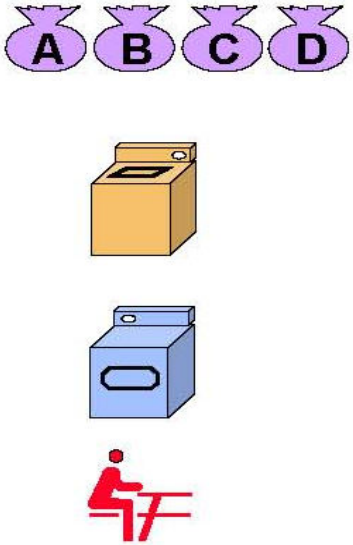
Bölüm 6

Pipelining

PIPELINING EXAMPLE 1

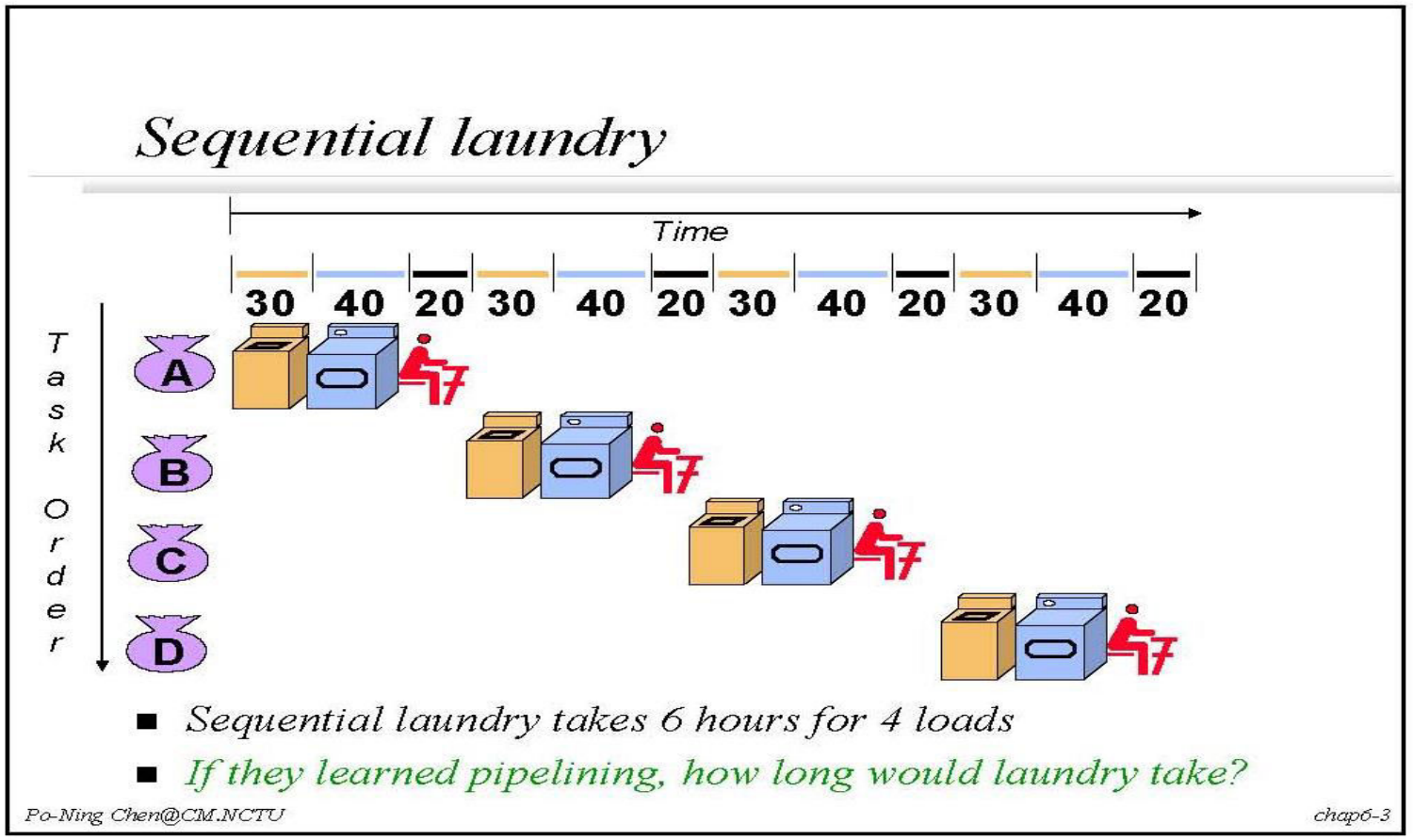
Pipeline is natural!

- *Laundry Example*
  - *Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold*
  - *Washer takes 30 minutes*
  - *Dryer takes 40 minutes*
  - *Folder takes 20 minutes*

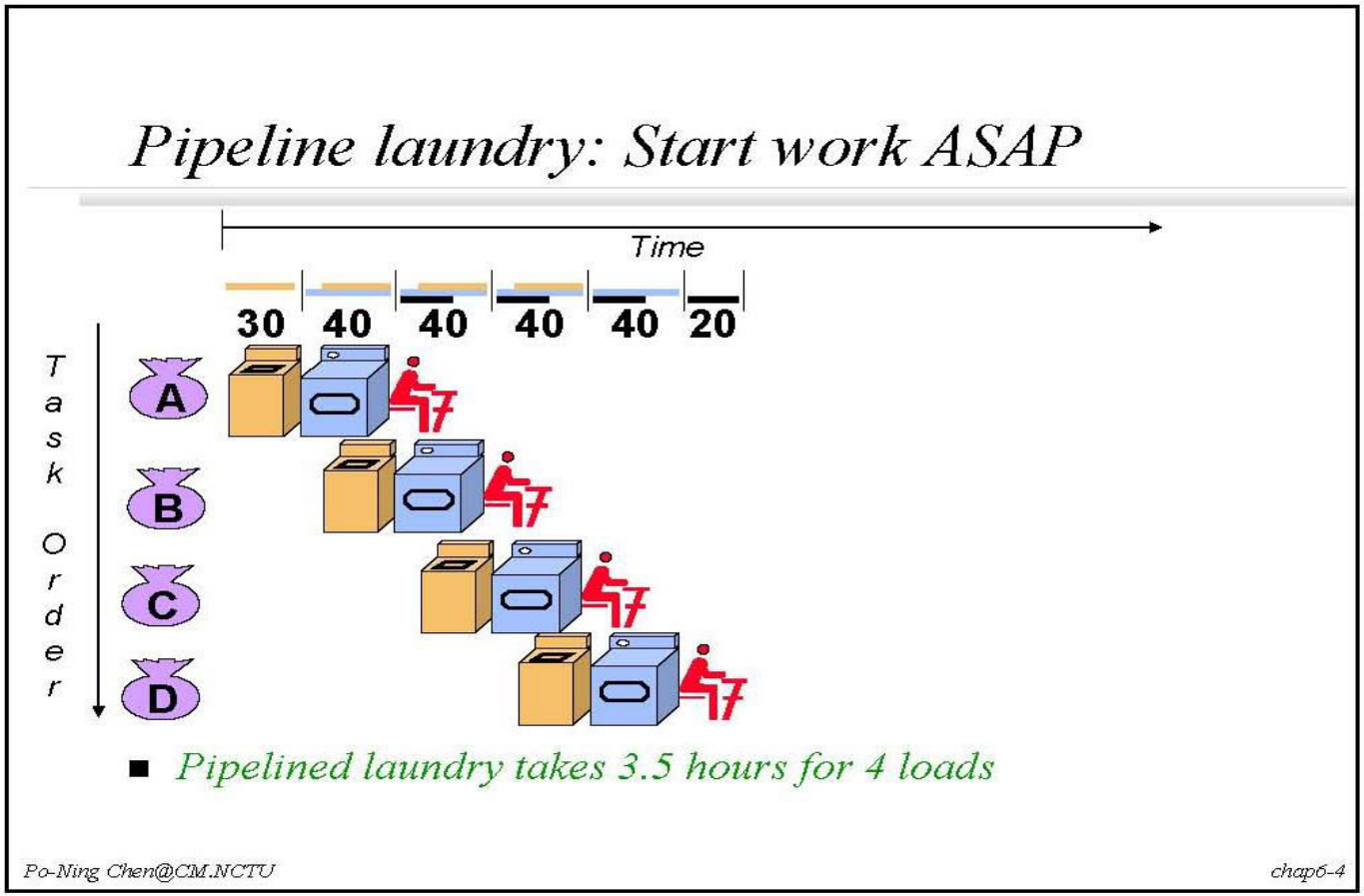


Po-Ning Chen@CM.NCTU *chap6-2*

SEQUENTIAL PROCESS

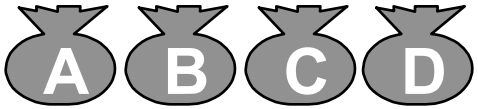


PIPELINING PROCESS

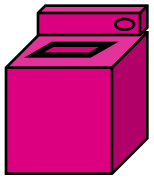


PIPELINING EXAMPLE 2

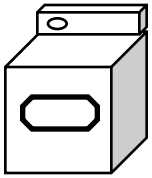
- Laundry Example
- Sammy, Marc, Griffy, Albert each have one load of clothes to wash, dry, and fold



- Washer takes 30 minutes



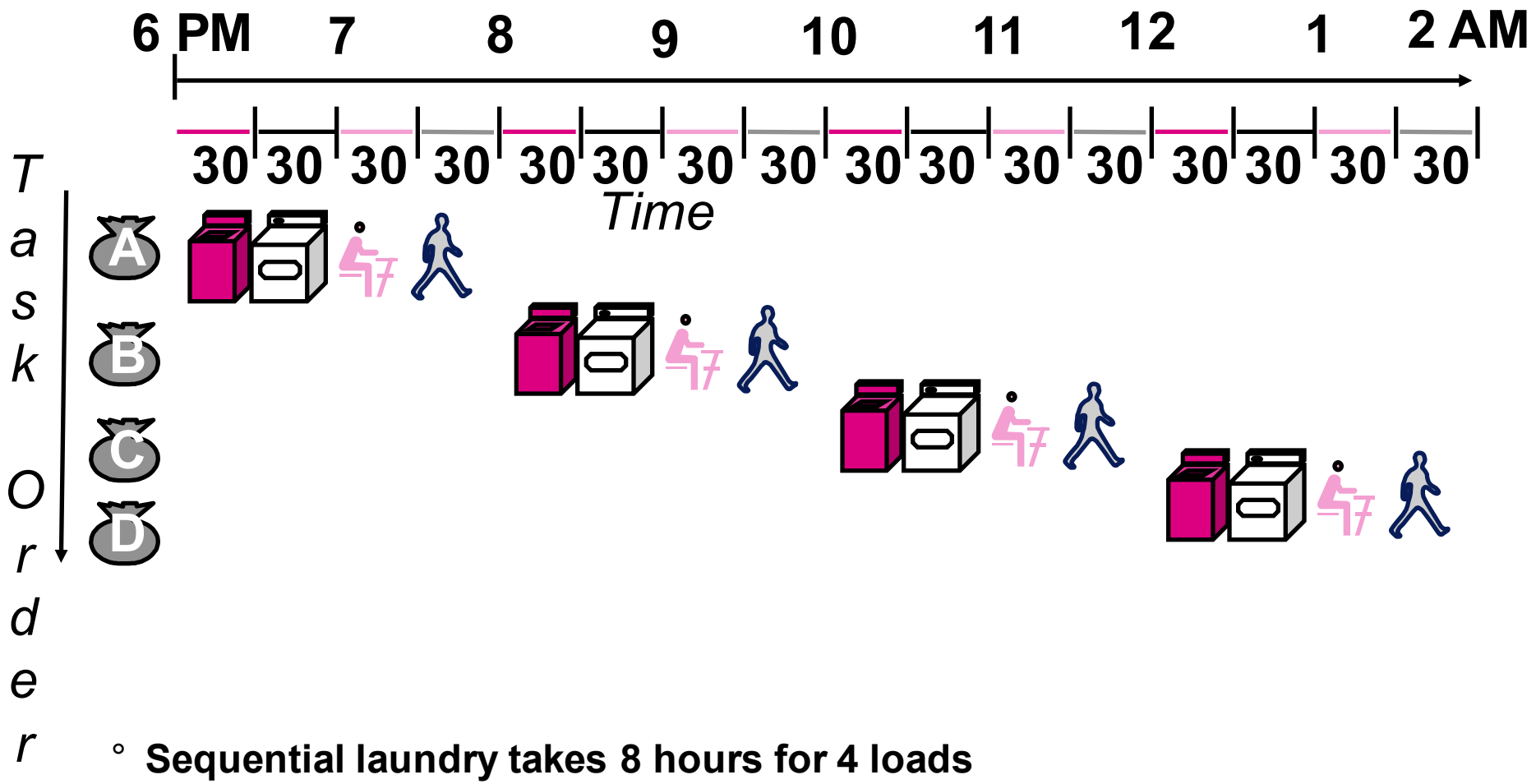
- Dryer takes 30 minutes



- "Folder" takes 30 minutes

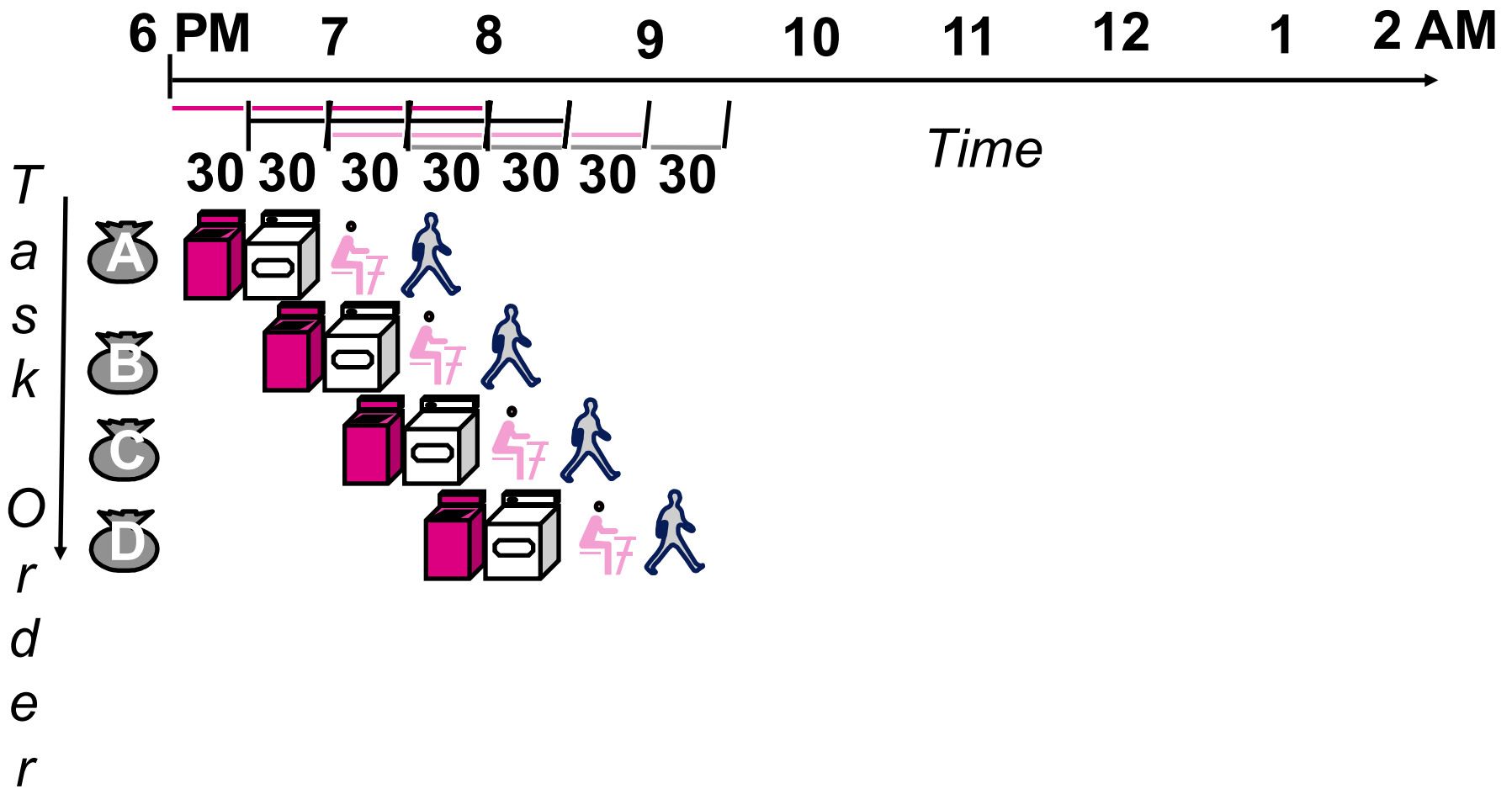
- "Stockman takes 30 minutes to put clothes into drawers

Sequential Laundry



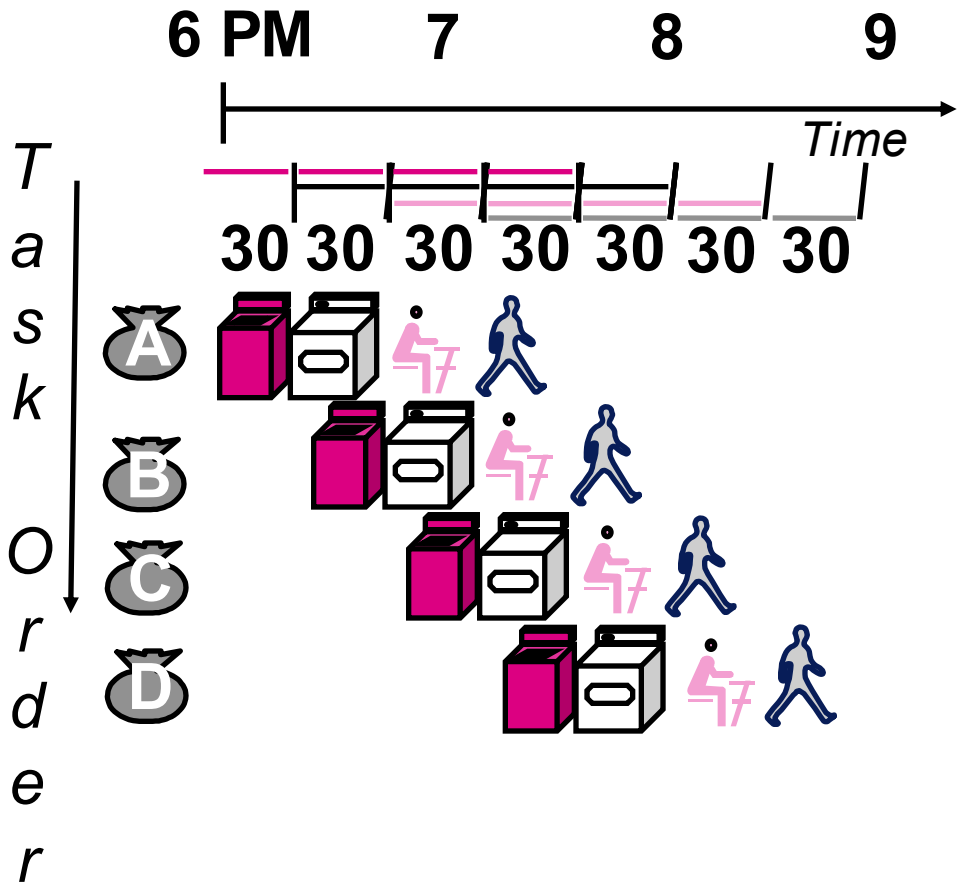
- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry:



° 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
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependences

PIPELINE ORGANIZATIONS

1. Arithmetic Pipeline

° divides an arithmetic operation into suboperations for execution in the pipeline segments.

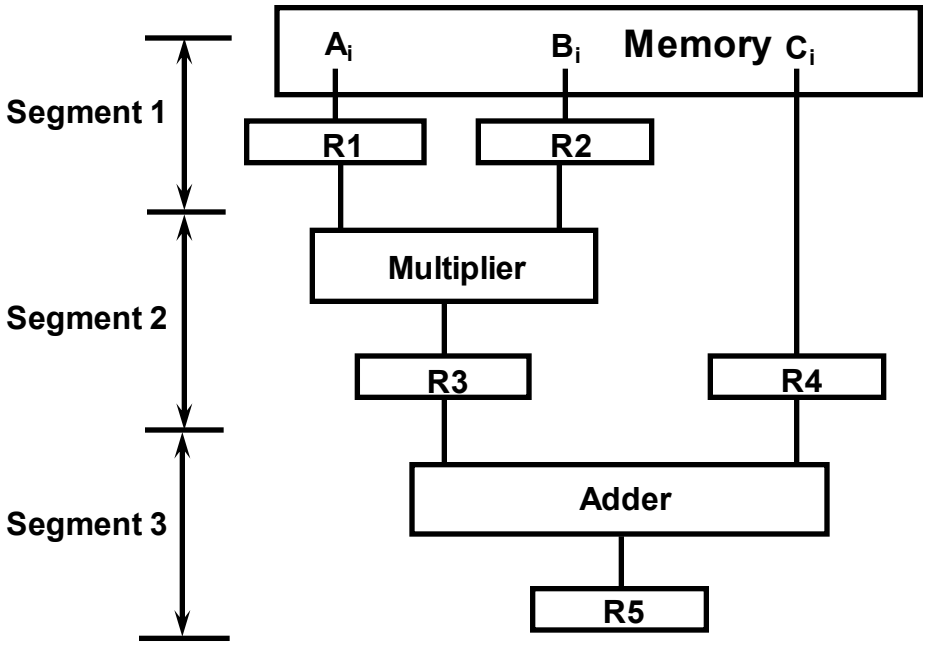
2. Instruction Pipeline

° operates on a stream of instructions by overlapping the *fetch*, *decode* and *execute* phases of the instruction cycle.

PIPELINING

A technique of decomposing a sequential process into suboperations, with each subprocess being executed in a partial dedicated segment that operates concurrently with all other segments.

$A_i * B_i + C_i$ for i = 1, 2, 3, ... , 7



$R1 \leftarrow A_i$, $R2 \leftarrow B_i$ Load A_i and B_i

$R3 \leftarrow R1 * R2$, $R4 \leftarrow C_i$ Multiply and load C_i

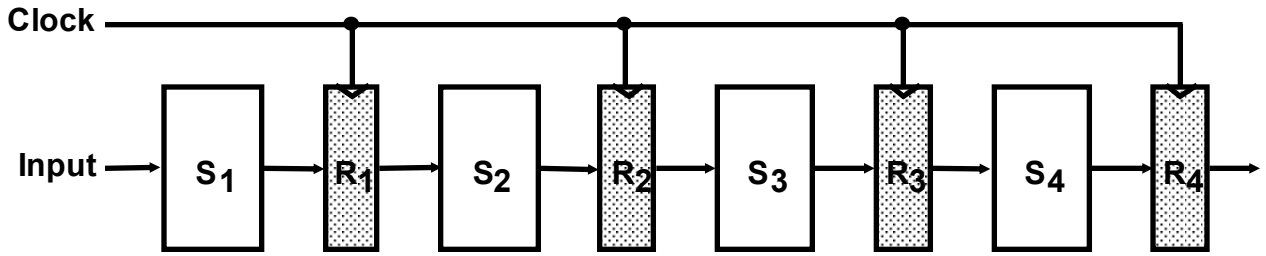
$R5 \leftarrow R3 + R4$ Add

OPERATIONS IN EACH PIPELINE STAGE

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1			
2	A2	B2	A1 * B1	C1	
3	A3	B3	A2 * B2	C2	A1 * B1 + C1
4	A4	B4	A3 * B3	C3	A2 * B2 + C2
5	A5	B5	A4 * B4	C4	A3 * B3 + C3
6	A6	B6	A5 * B5	C5	A4 * B4 + C4
7	A7	B7	A6 * B6	C6	A5 * B5 + C5
8			A7 * B7	C7	A6 * B6 + C6
9					A7 * B7 + C7

GENERAL PIPELINE

General Structure of a 4-Segment Pipeline



Space-Time Diagram

	1	2	3	4	5	6	7	8	9
Segment 1	T1	T2	T3	T4	T5	T6			
2		T1	T2	T3	T4	T5	T6		
3			T1	T2	T3	T4	T5	T6	
4				T1	T2	T3	T4	T5	T6

Clock cycles

PIPELINE SPEEDUP

n: Number of tasks to be performed

Conventional Machine (Non-Pipelined)

t_n: Clock cycle

τ_1: Time required to complete the n tasks

$\tau_1 = n * t_n$

Pipelined Machine (k stages)

t_p: Clock cycle (time to complete each suboperation)

τ_k: Time required to complete the n tasks

$\tau_k = (k + n - 1) * t_p$

Speedup

S_k: Speedup

$$S_k = n*t_n / (k + n - 1)*t_p$$

$$\lim_{n \to \infty} S_k = \frac{t_n}{t_p} \quad (= k, \text{ if } t_n = k * t_p)$$

PIPELINE AND MULTIPLE FUNCTION UNITS

Example

- **4-stage pipeline**
- **subopertion in each stage; t_p = 20nS**
- **100 tasks to be executed**
- **1 task in non-pipelined system; 20*4 = 80nS**

Pipelined System

$$(k + n - 1)*t_p = (4 + 99) * 20 = 2060nS$$

Non-Pipelined System

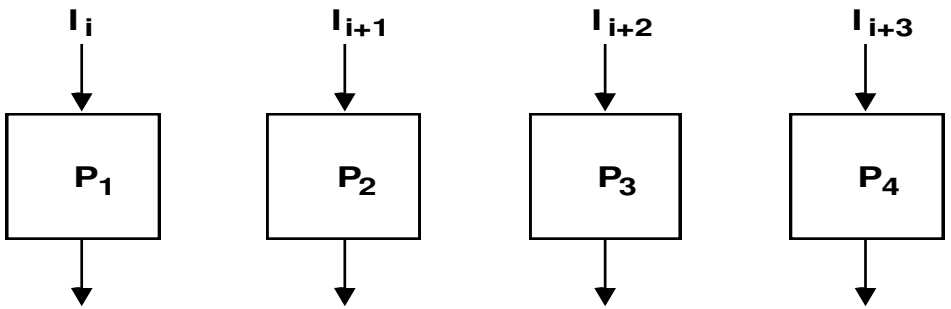
$$n*k*t_p = 100 * 80 = 8000nS$$

Speedup

$$S_k = 8000 / 2060 = 3.88$$

4-Stage Pipeline is basically identical to the system with 4 identical function units

Multiple Functional Units

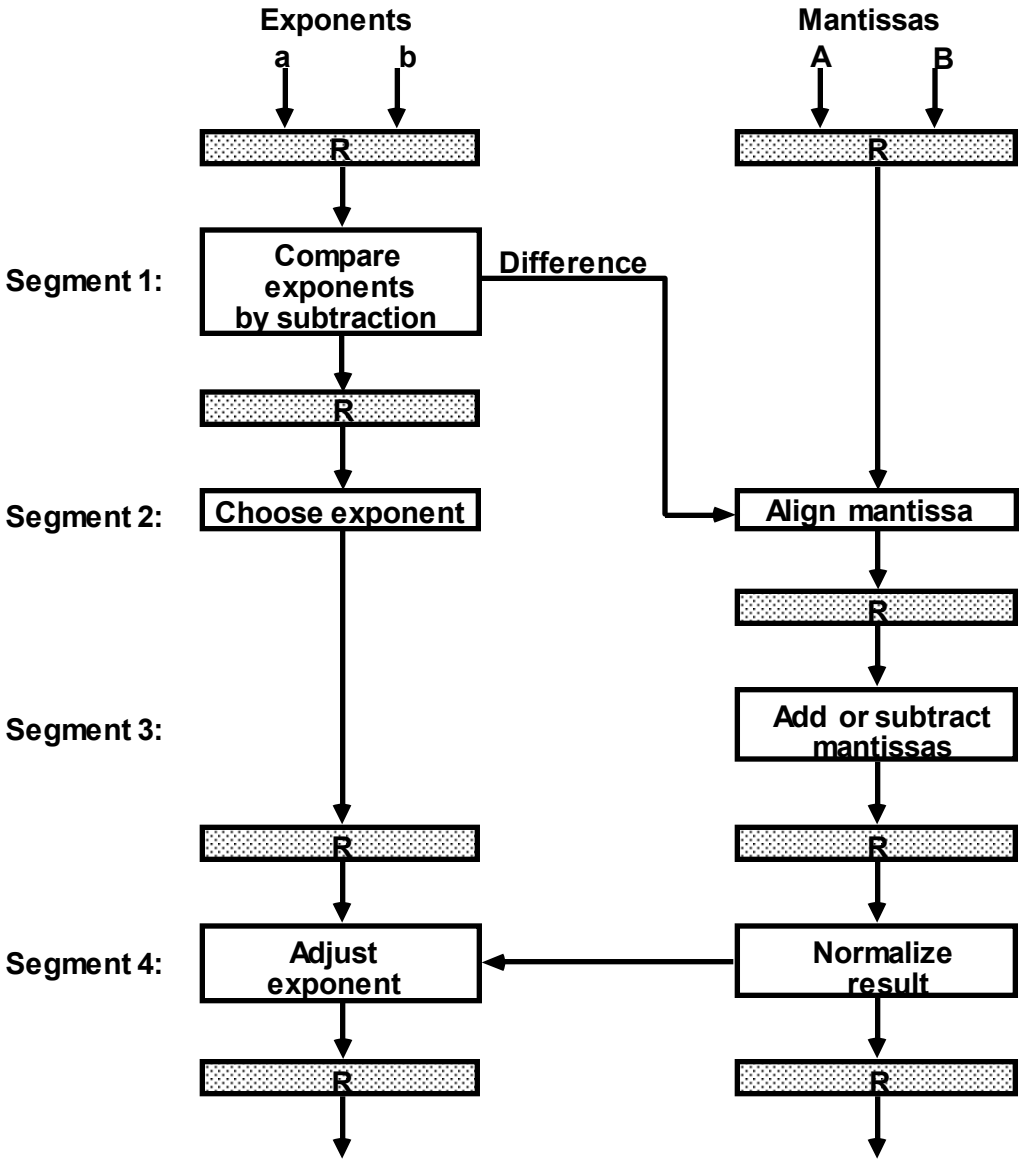


ARITHMETIC PIPELINE

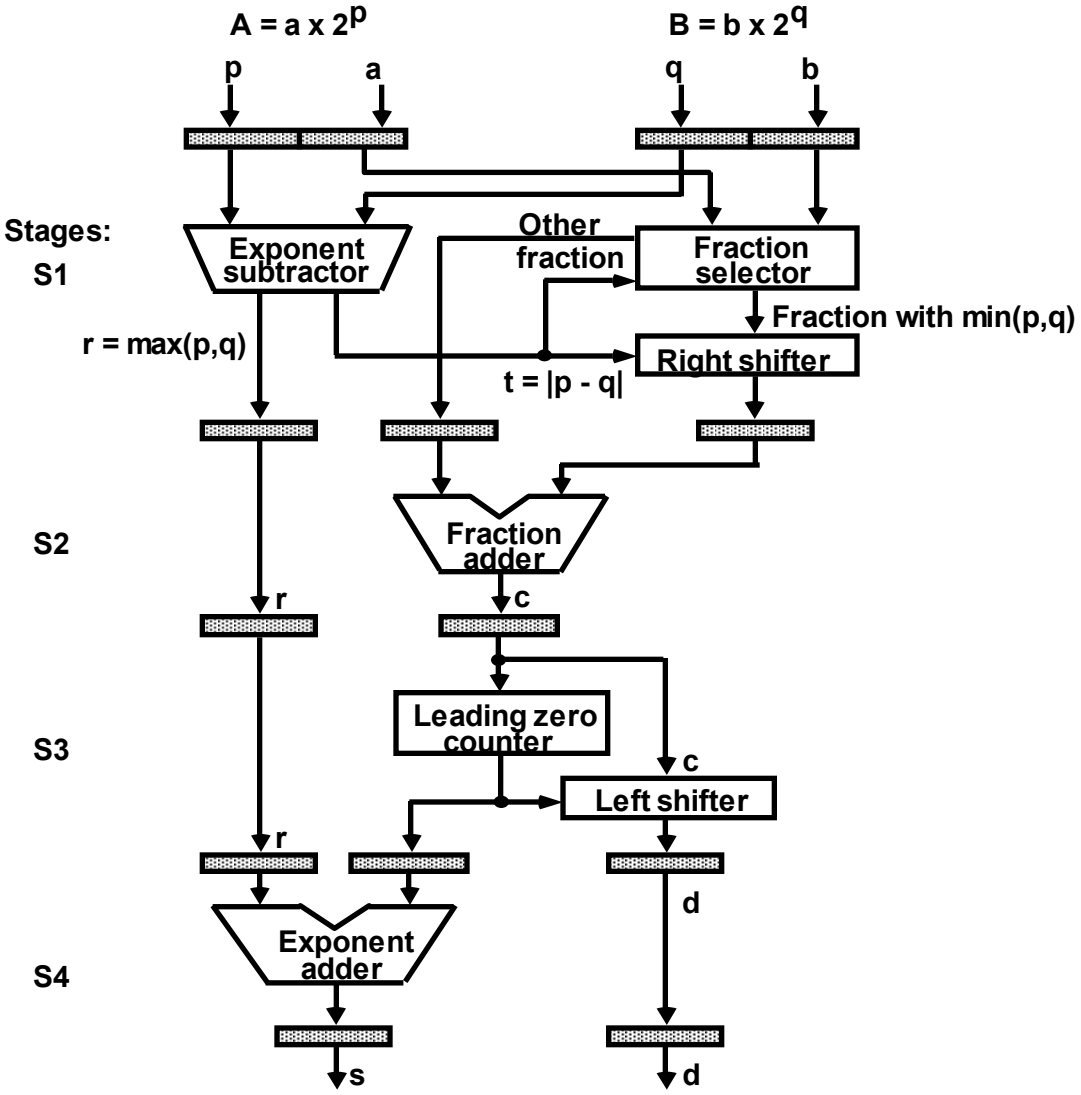
Floating-point adder

$X = A \times 2^a$
$Y = B \times 2^b$

[1] Compare the exponents
[2] Align the mantissa
[3] Add/sub the mantissa
[4] Normalize the result



4-STAGE FLOATING POINT ADDER



FP Pipeline Speed

- 4 segment delays for FP Adder

t_1 = 60 ns, t_2 = 70 ns, t_3 = 100 ns, t_4 = 80 ns,

t_r =10 ns (delay in the intermediate registers)

Pipeline procedure : $t_p = t_3 + t_r$ =110 ns

Nonpipelined proc: $t_n = t_1 + t_2 + t_3 + t_4 + t_r$ =320 ns

Speedup = S = t_n / t_p =2.9

INSTRUCTION PIPELINE

INSTRUCTION CYCLE

Six Phases* in an Instruction Cycle

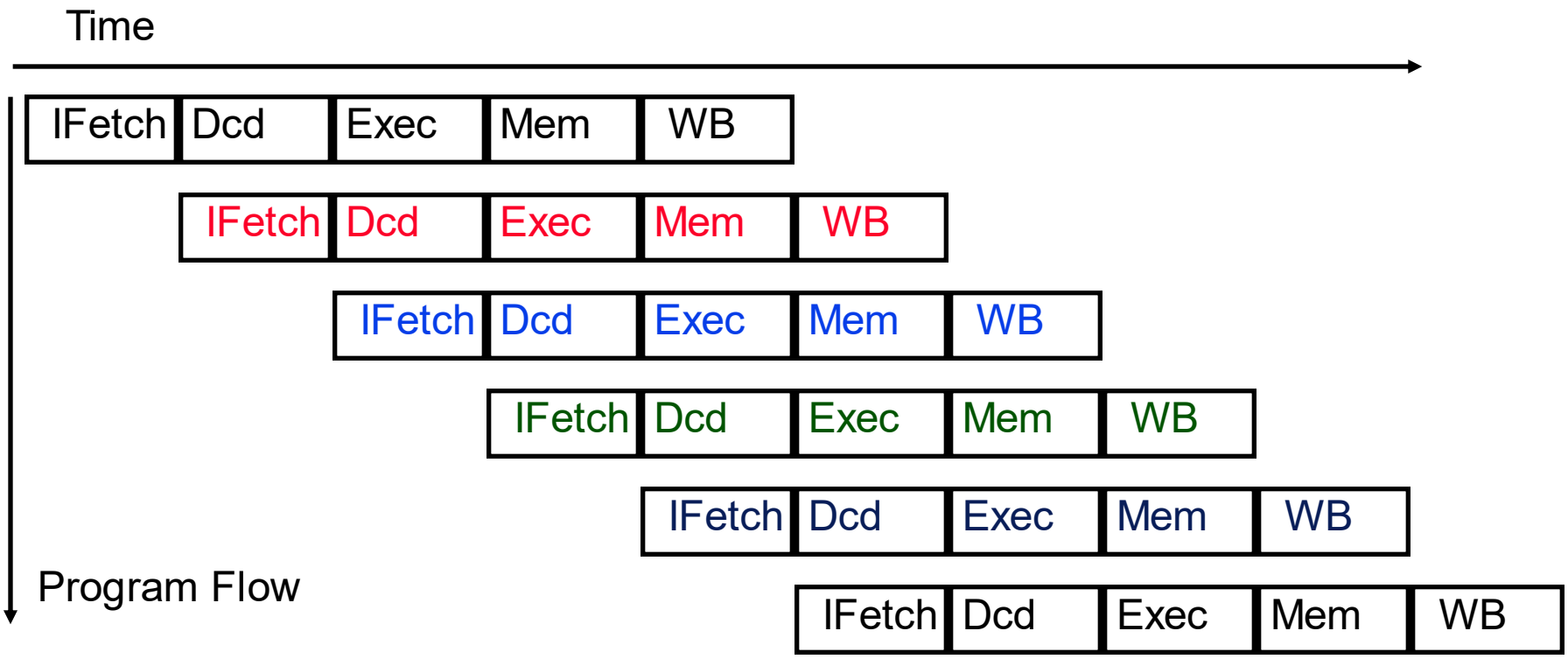
[1] Fetch an instruction from memory
[2] Decode the instruction
[3] Calculate the effective address of the operand
[4] Fetch the operands from memory
[5] Execute the operation
[6] Store the result in the proper place

*** Some instructions skip some phases**
***Two or more segments may require memory access at the same time.**
*** Storage of the operation result into a register is done automatically in the execution phase**

==> 4-Stage Pipeline

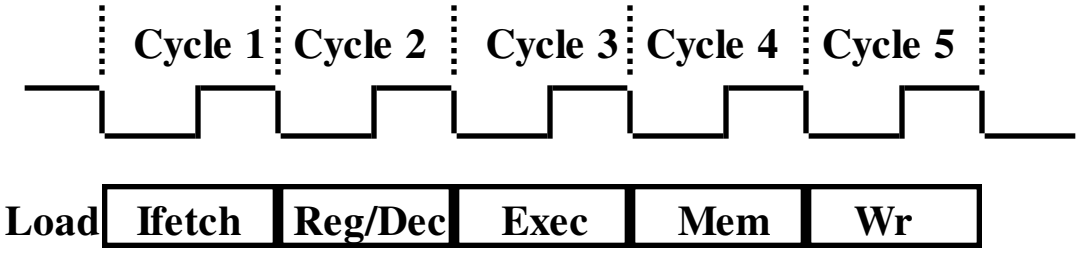
[1] FI: Fetch an instruction from memory
[2] DA: Decode the instruction and calculate the effective address of the operand
[3] FO: Fetch the operand
[4] EX: Execute the operation

MIPS Pipelined Execution Representation



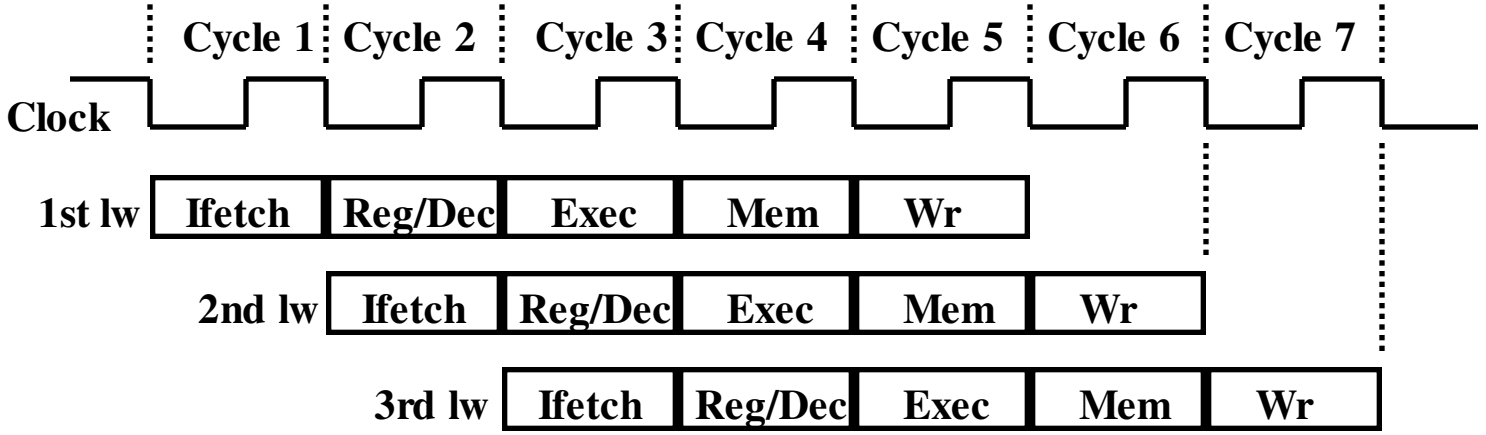
Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

The Five Stages of Load



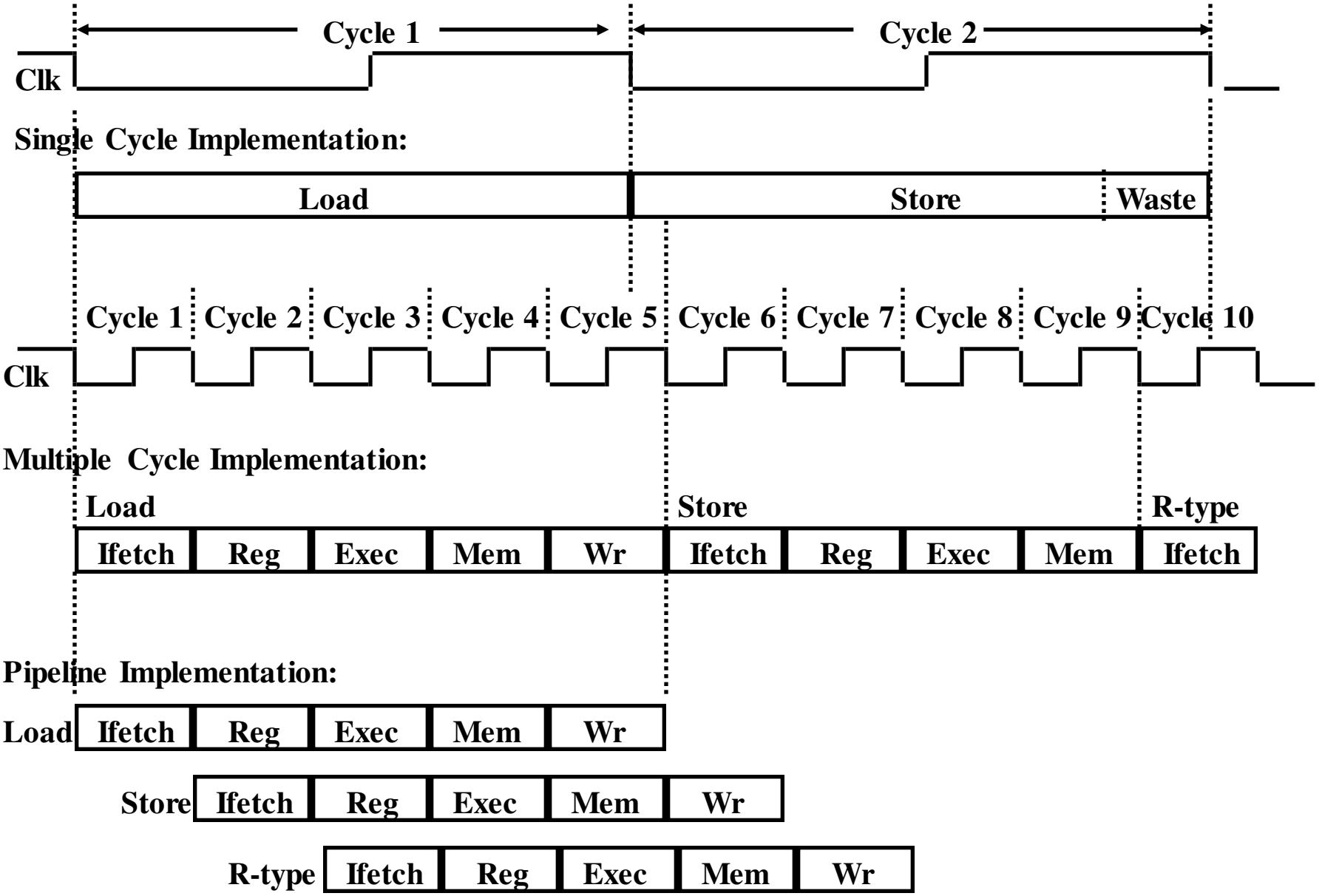
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

Pipelining the Load Instruction

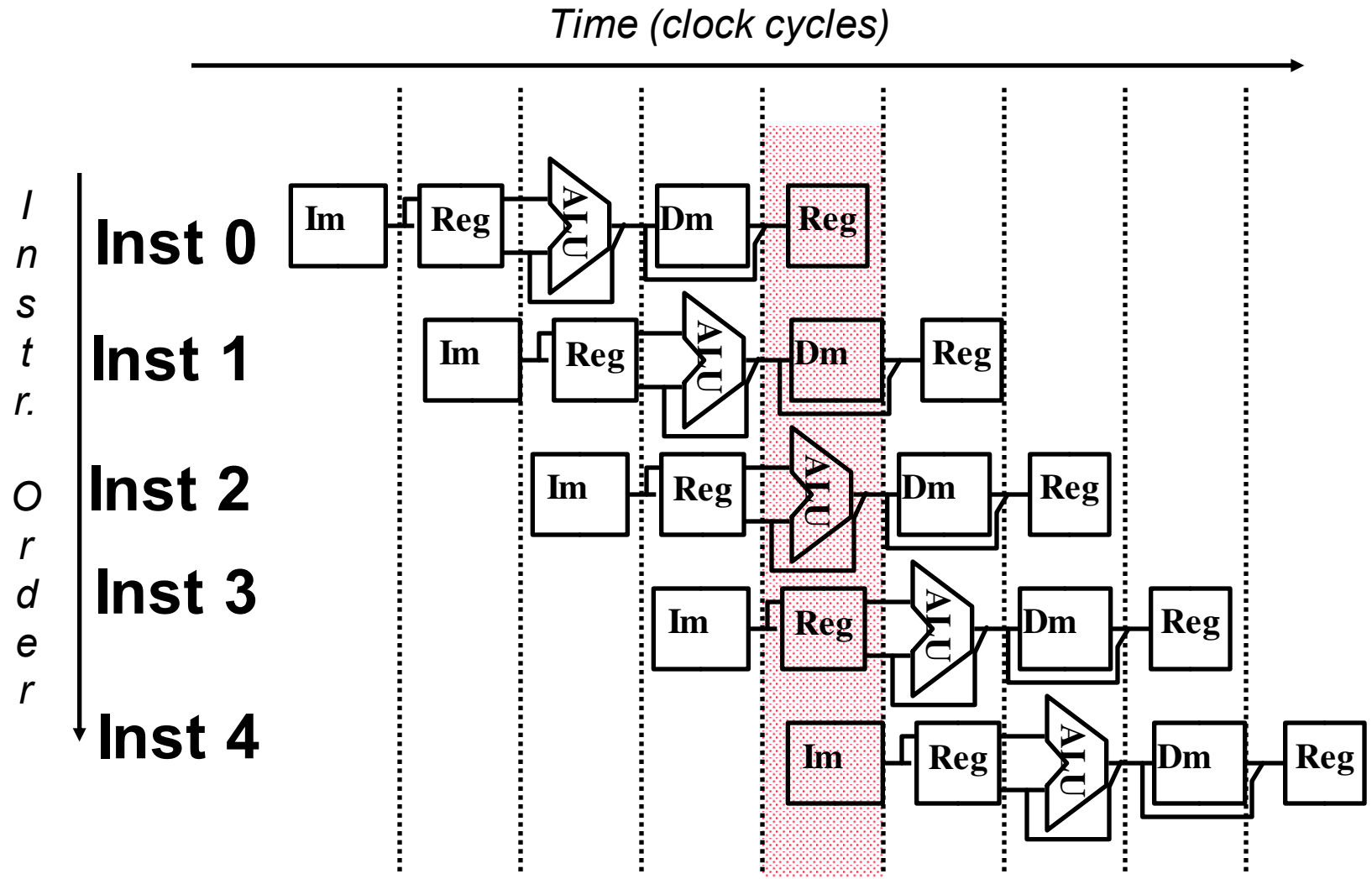


- The five independent functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's Read ports (bus A and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the Mem stage
  - Register File's Write port (bus W) for the Wr stage
- One instruction enters the pipeline every cycle
  - One instruction comes out of the pipeline (complete) every cycle
  - The "Effective" Cycles per Instruction (CPI) is 1

Single Cycle, Multiple Cycle, vs. Pipeline



Why Pipeline? Because the resources are there!

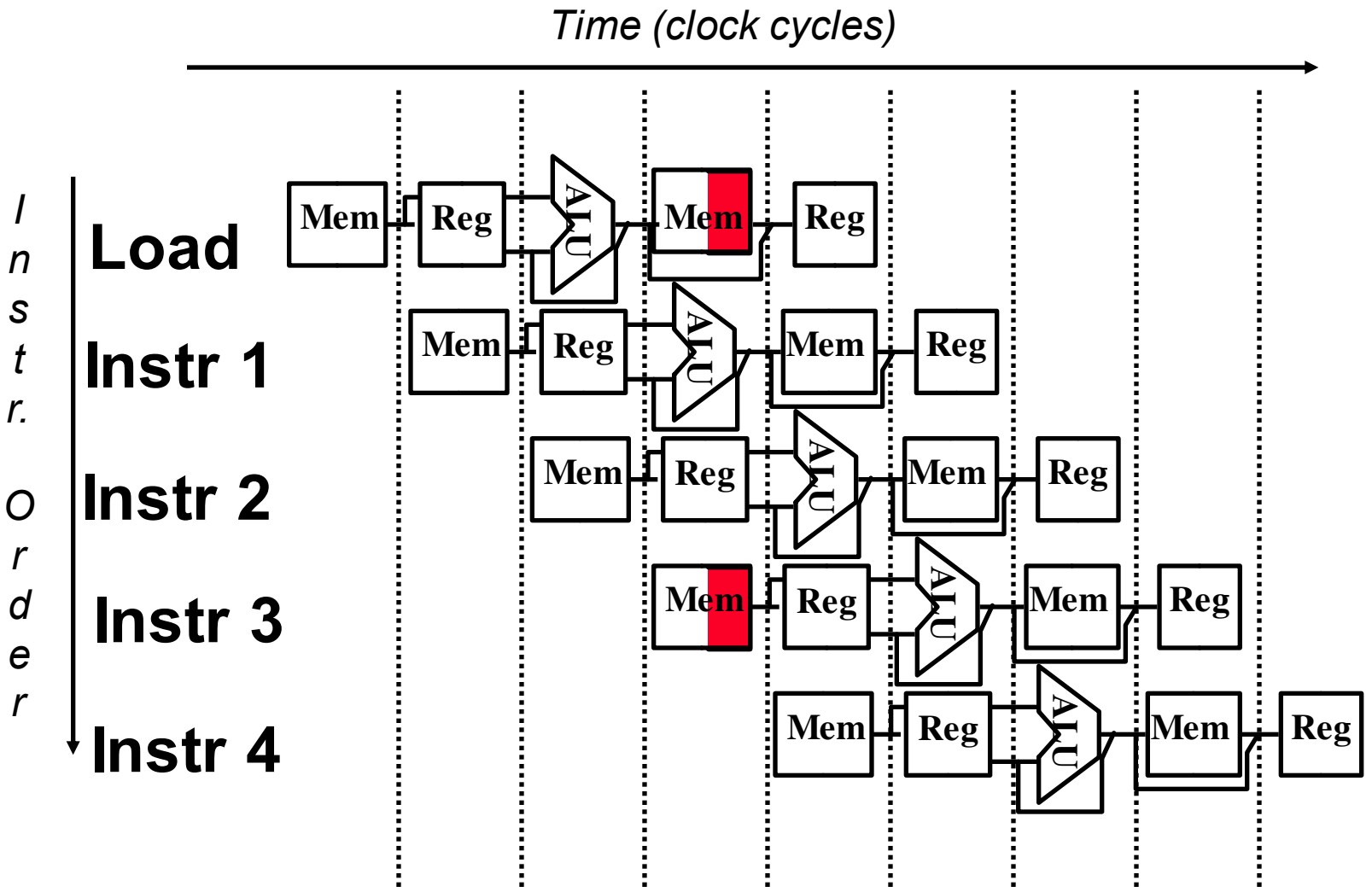


Can pipelining get us into trouble?

- Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource two different ways at the same time
    - E.g., combined washer/dryer would be a structural hazard
  - data hazards: 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 hazards: attempt to make a decision before condition is evaulated
    - E.g., washing football uniforms and need to get proper detergent level;
    - branch instructions

- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards

Single Memory is a Structural Hazard

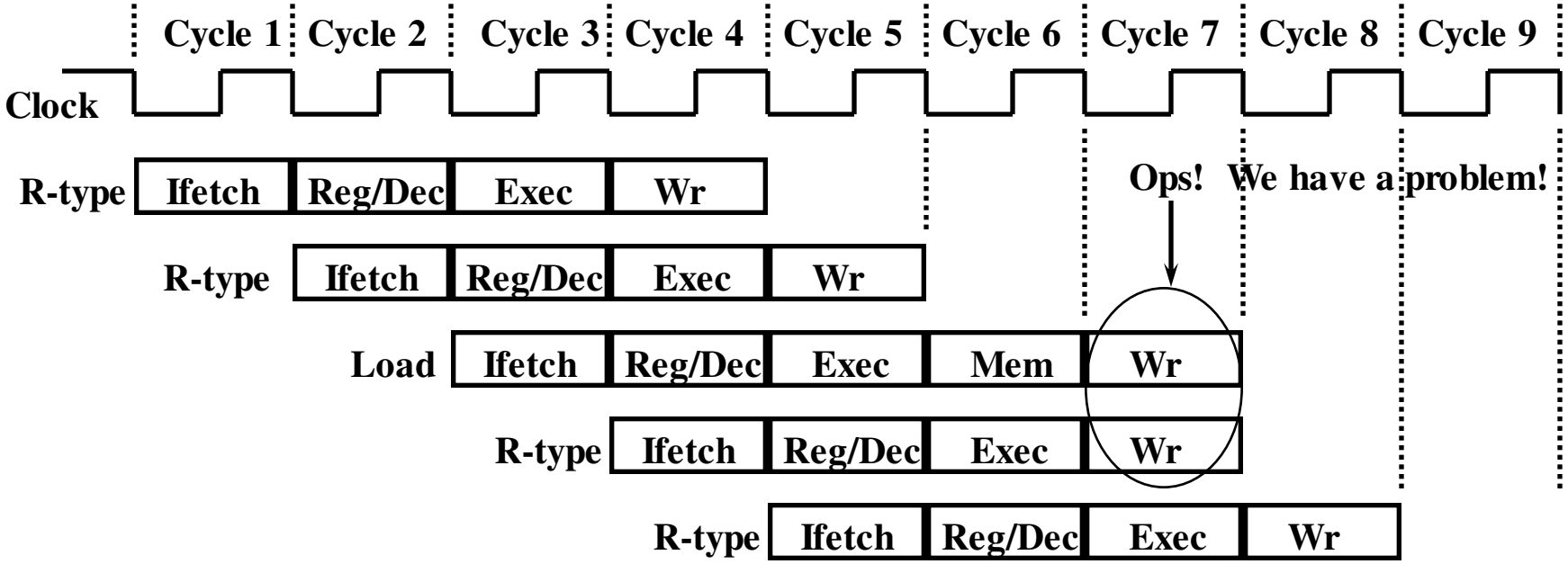


Detection is easy in this case! (right half highlight means read, left half write)

Structural Hazards limit performance

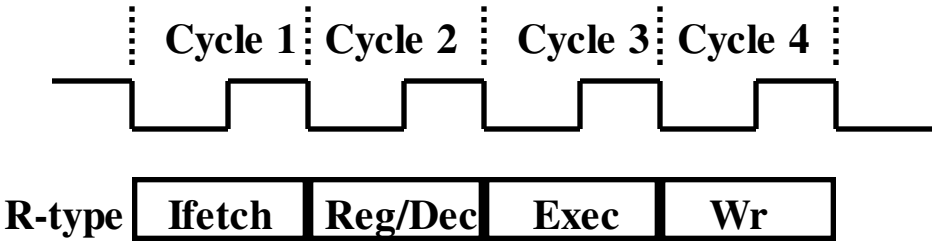
- Example: if 1.3 memory accesses per instruction and only one memory access per cycle then
  - average CPI 1.3
  - otherwise resource is more than 100% utilized
  - More on Hazards later

Pipelining the R-type and Load Instruction



- We have a problem:
  - Two instructions try to write to the register file at the same time!

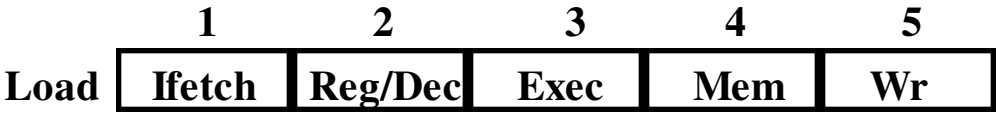
The Four Stages of R-type



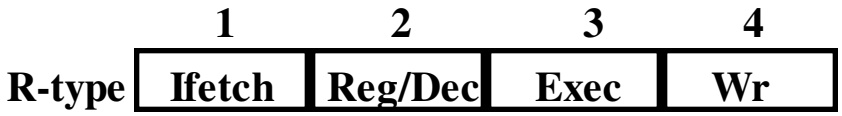
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: ALU operates on the two register operands
- Wr: Write the ALU output back to the register file

Important Observation

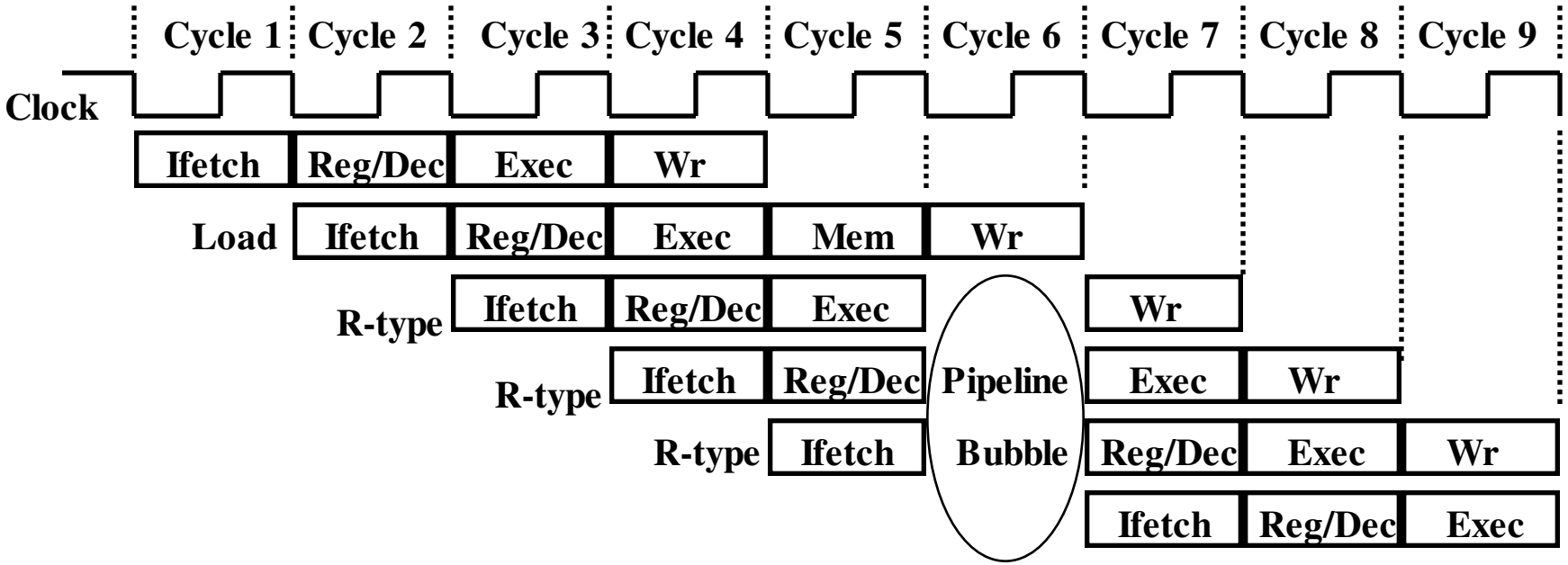
- Each functional unit can only be used once per instruction
- Each functional unit must be used at the same stage for all instructions:
  - Load uses Register File's Write Port during its 5th stage



  - R-type uses Register File's Write Port during its 4th stage



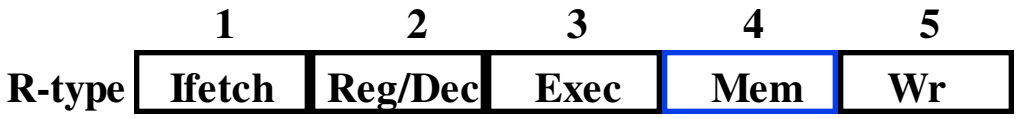
Solution 1: Insert "Bubble" into the Pipeline

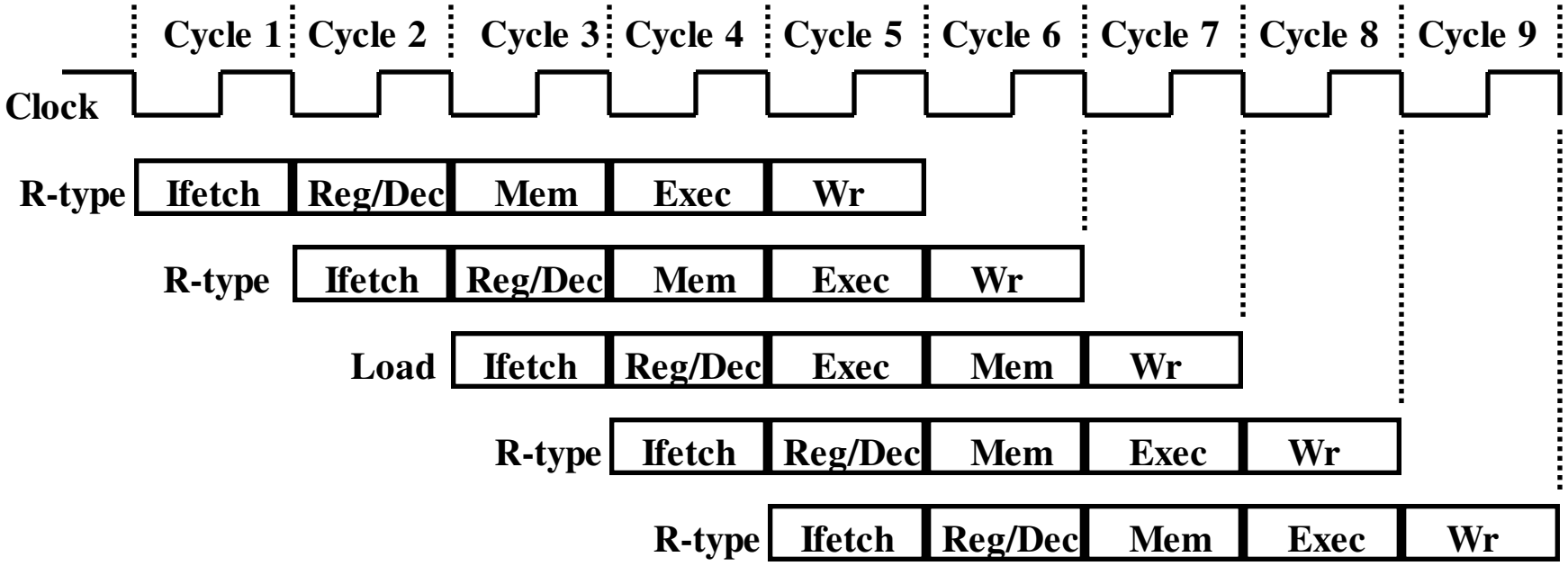


- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex
- No instruction is completed during Cycle 5:
  - The "Effective" CPI for load is >1

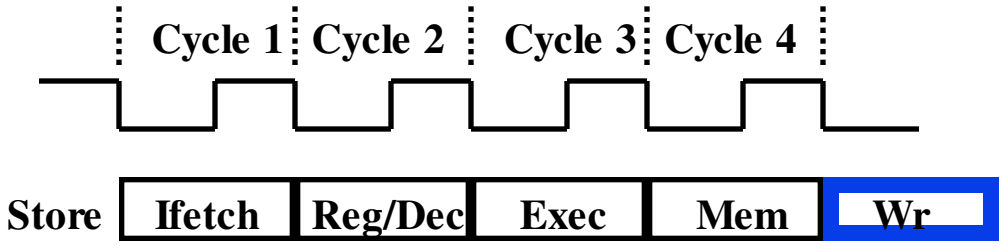
Solution 2: Delay R-type's Write by One Cycle

° Delay R-type's register write by one cycle:
  - Now R-type instructions also use Reg File's write port at Stage 5
  - Mem stage is a NOOP stage: nothing is being done



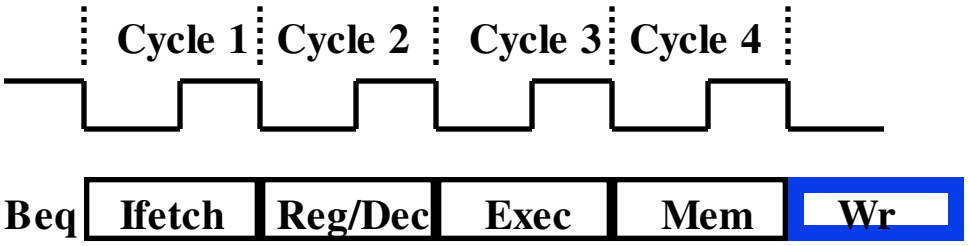


The Four Stages of Store



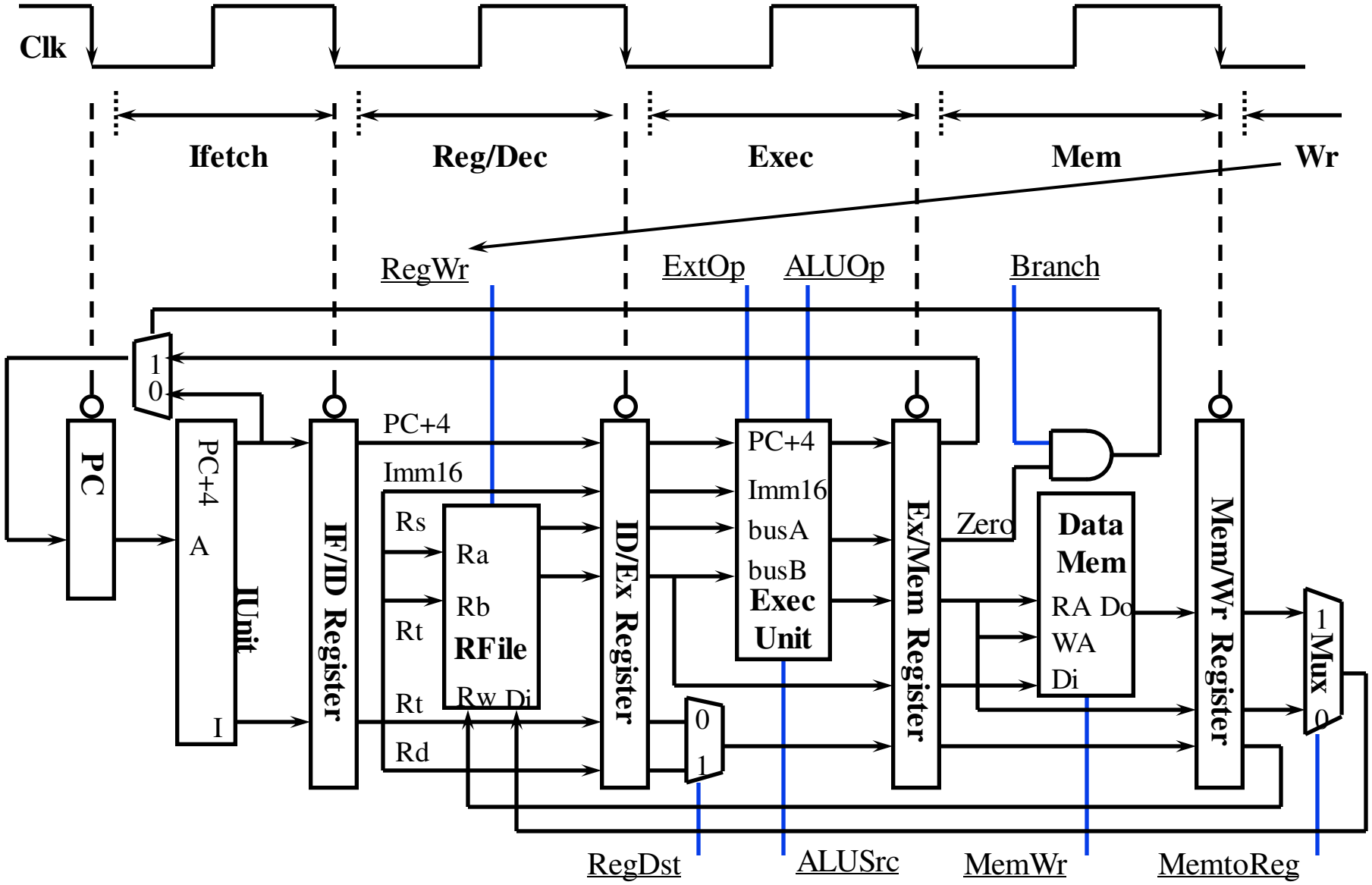
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

The Four Stages of Beq



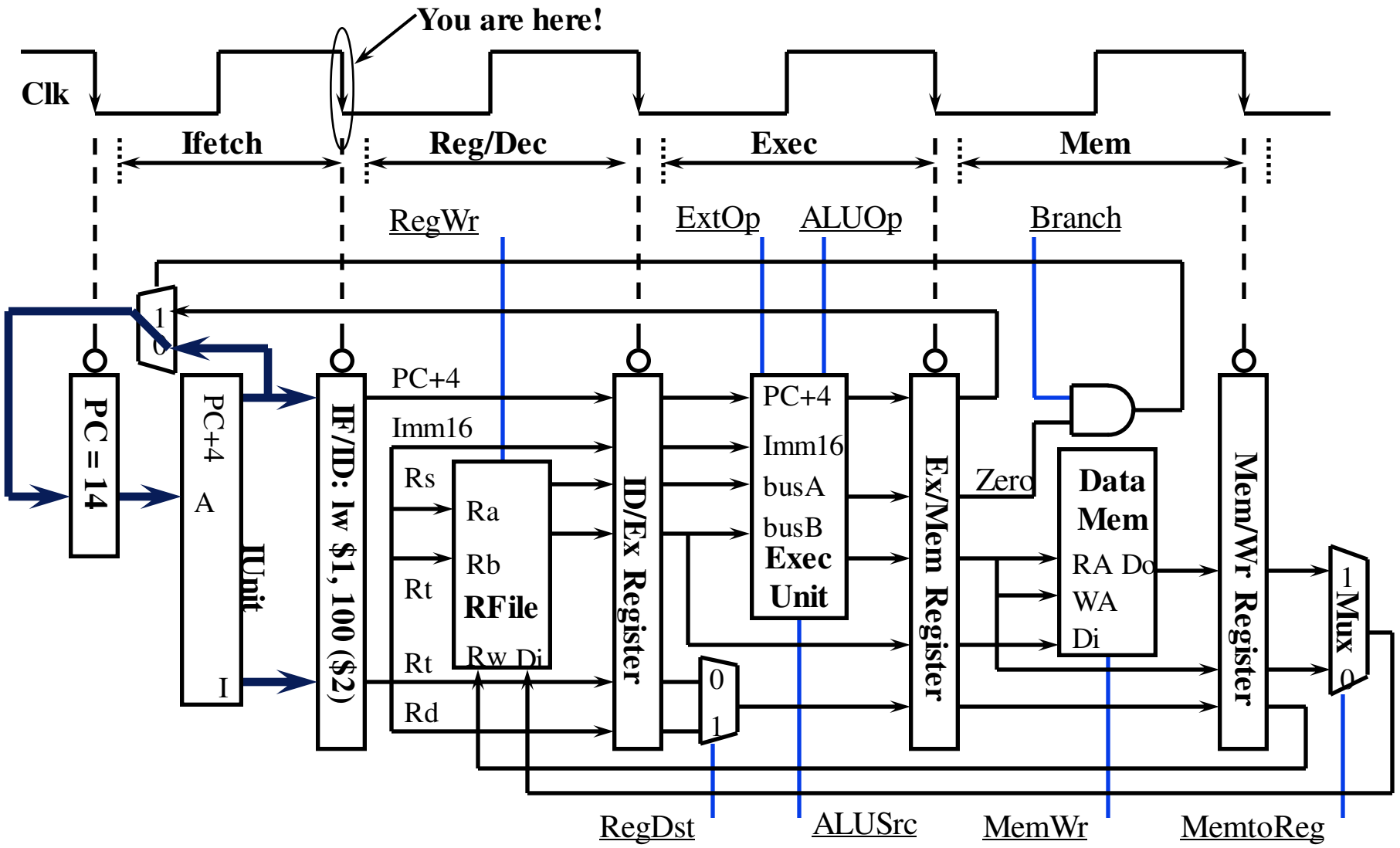
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: ALU compares the two register operands
  - Adder calculates the branch target address
- Mem: If the registers we compared in the Exec stage are the same,
  - Write the branch target address into the PC

A Pipelined Datapath



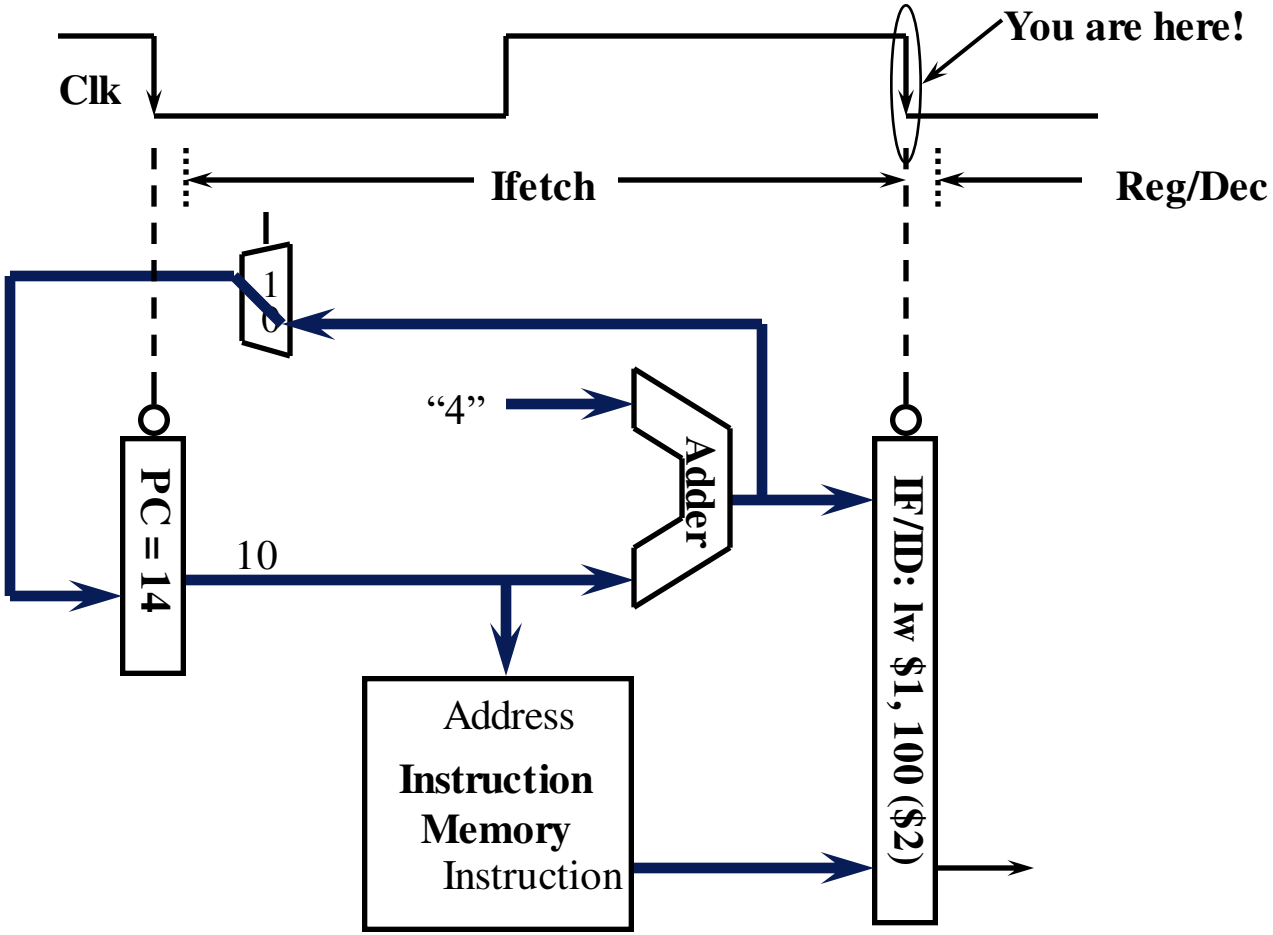
The Instruction Fetch Stage

° Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]



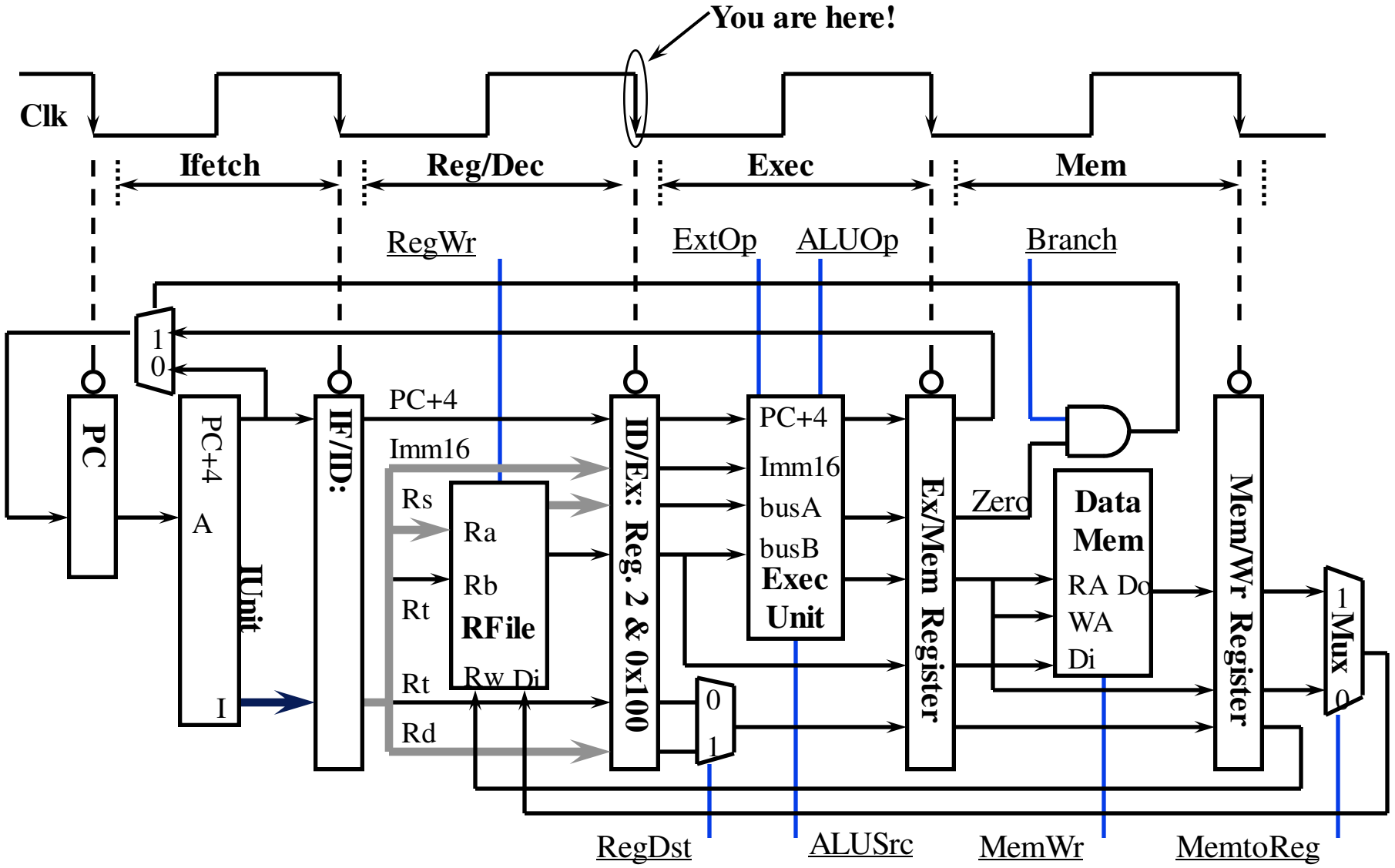
A Detail View of the Instruction Unit

° Location 10: lw $1, 0x100($2)



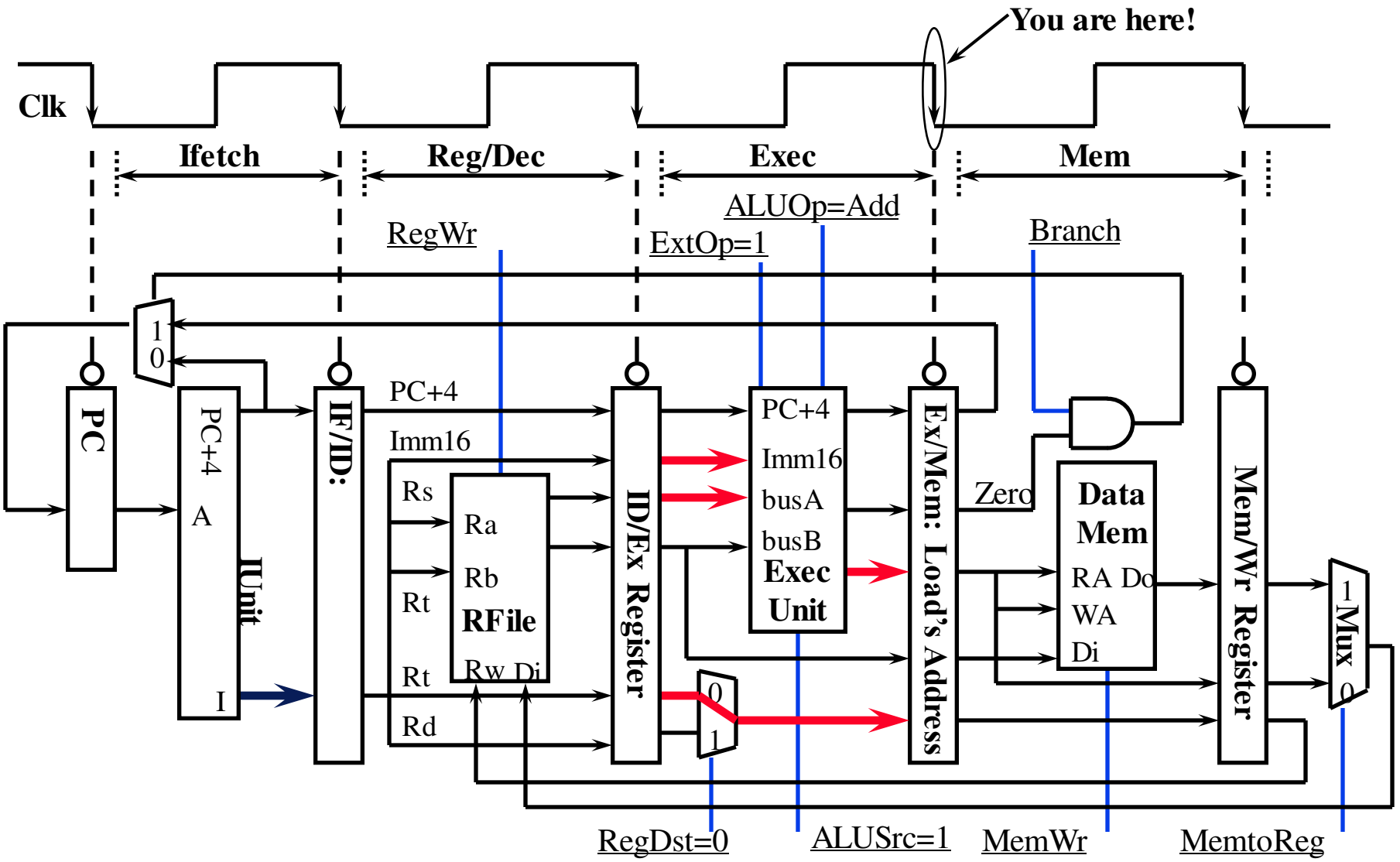
The Decode / Register Fetch Stage

° Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]

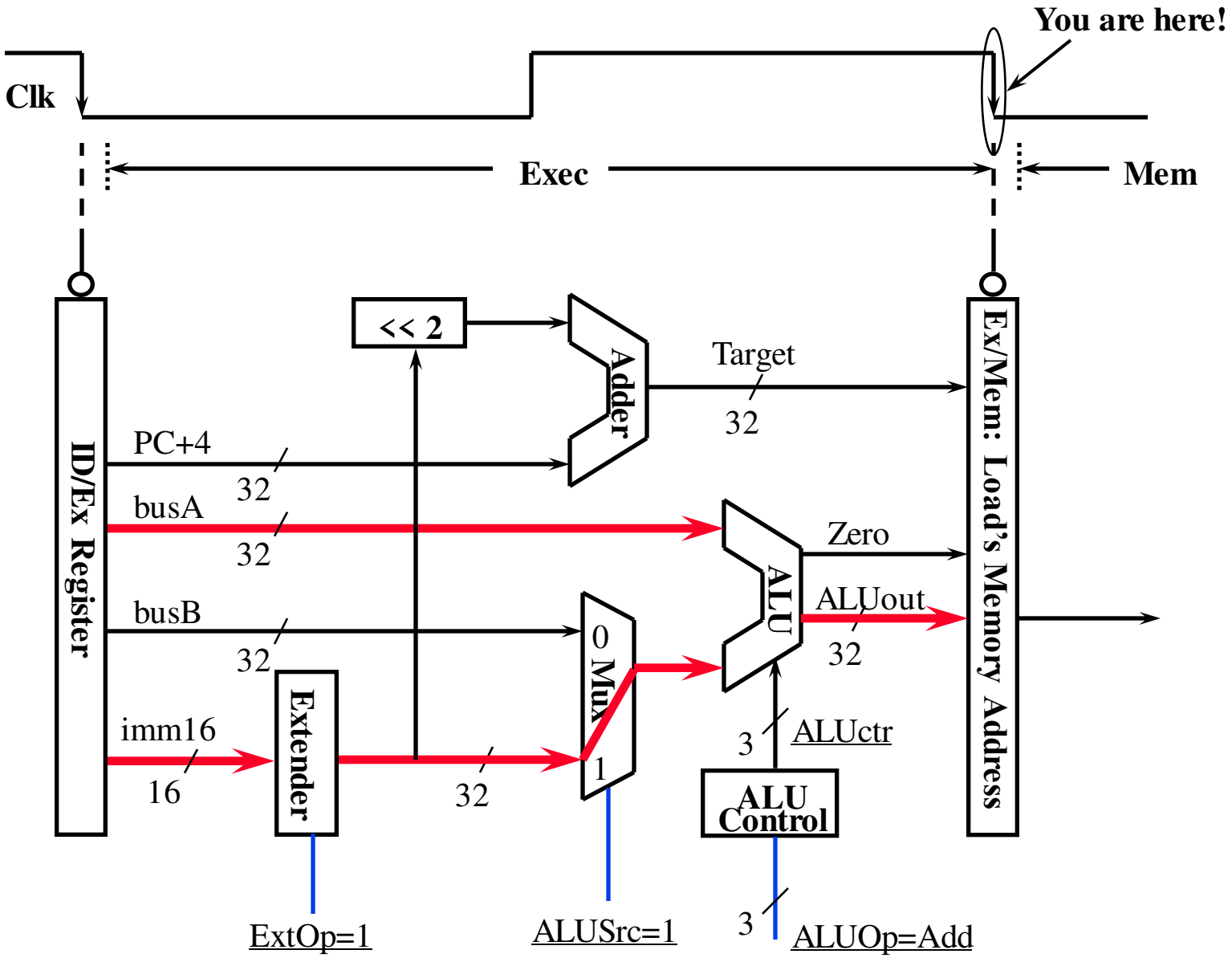


Load's Address Calculation Stage

° Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]

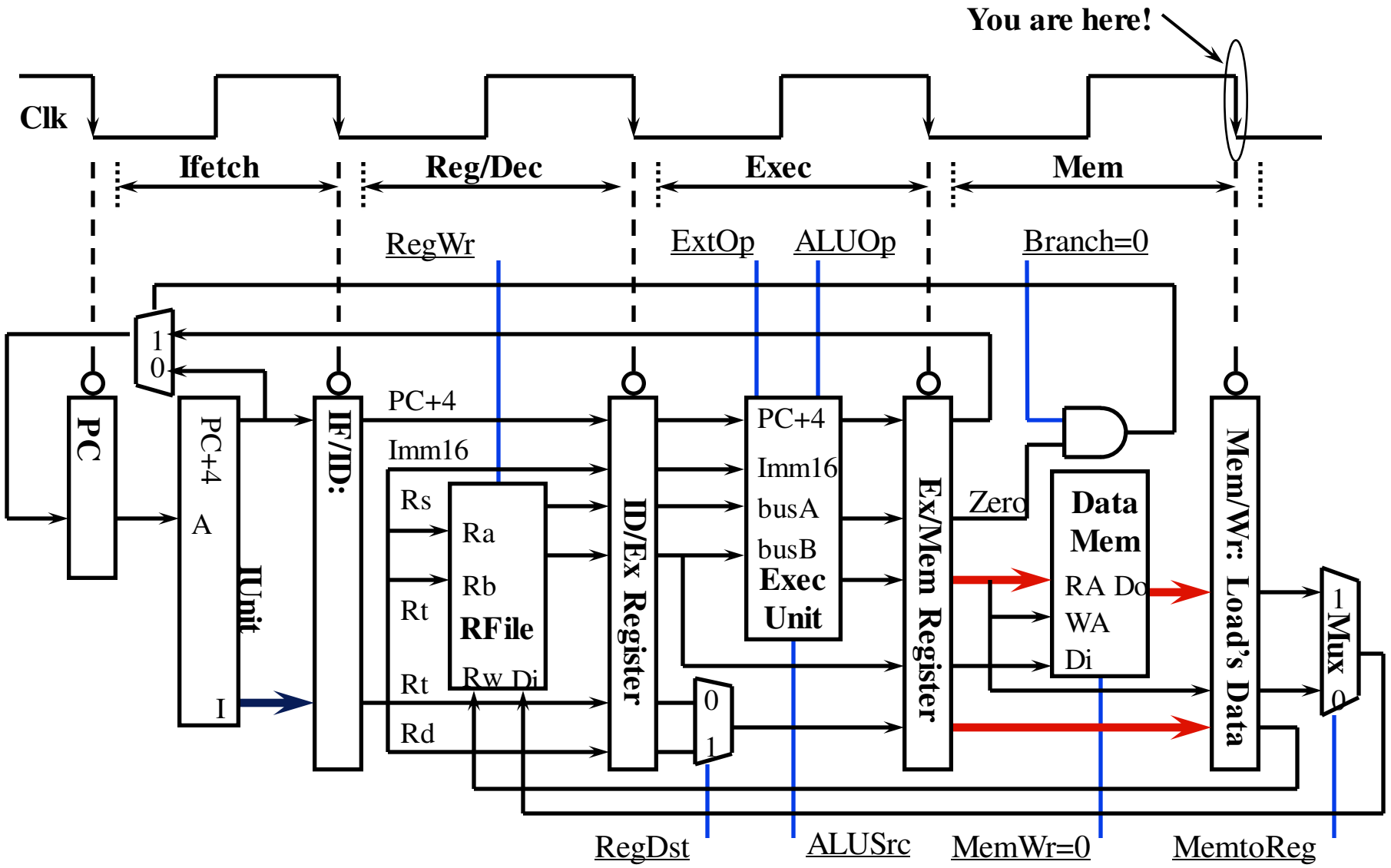


A Detail View of the Execution Unit



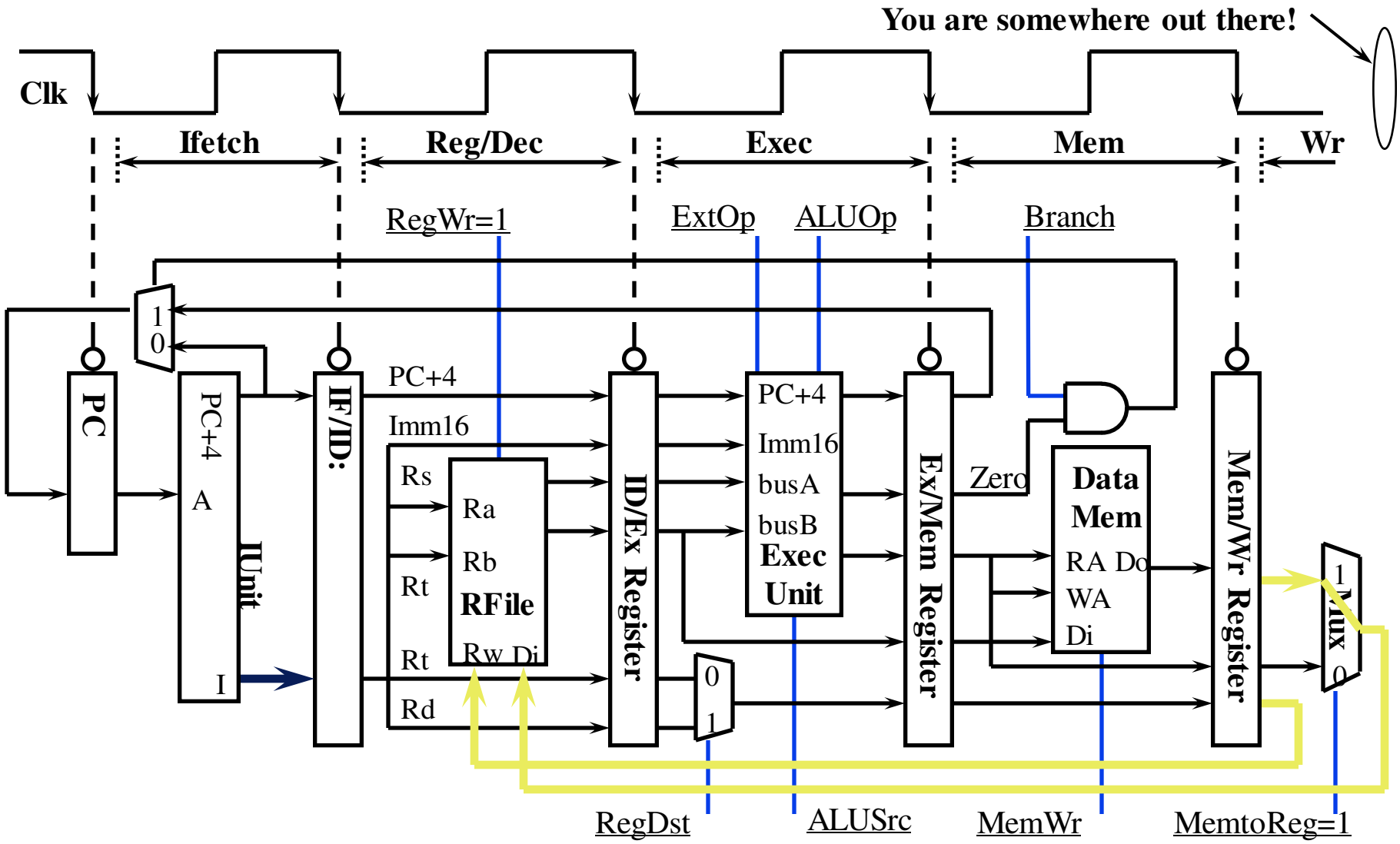
Load's Memory Access Stage

° Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]



Load's Write Back Stage

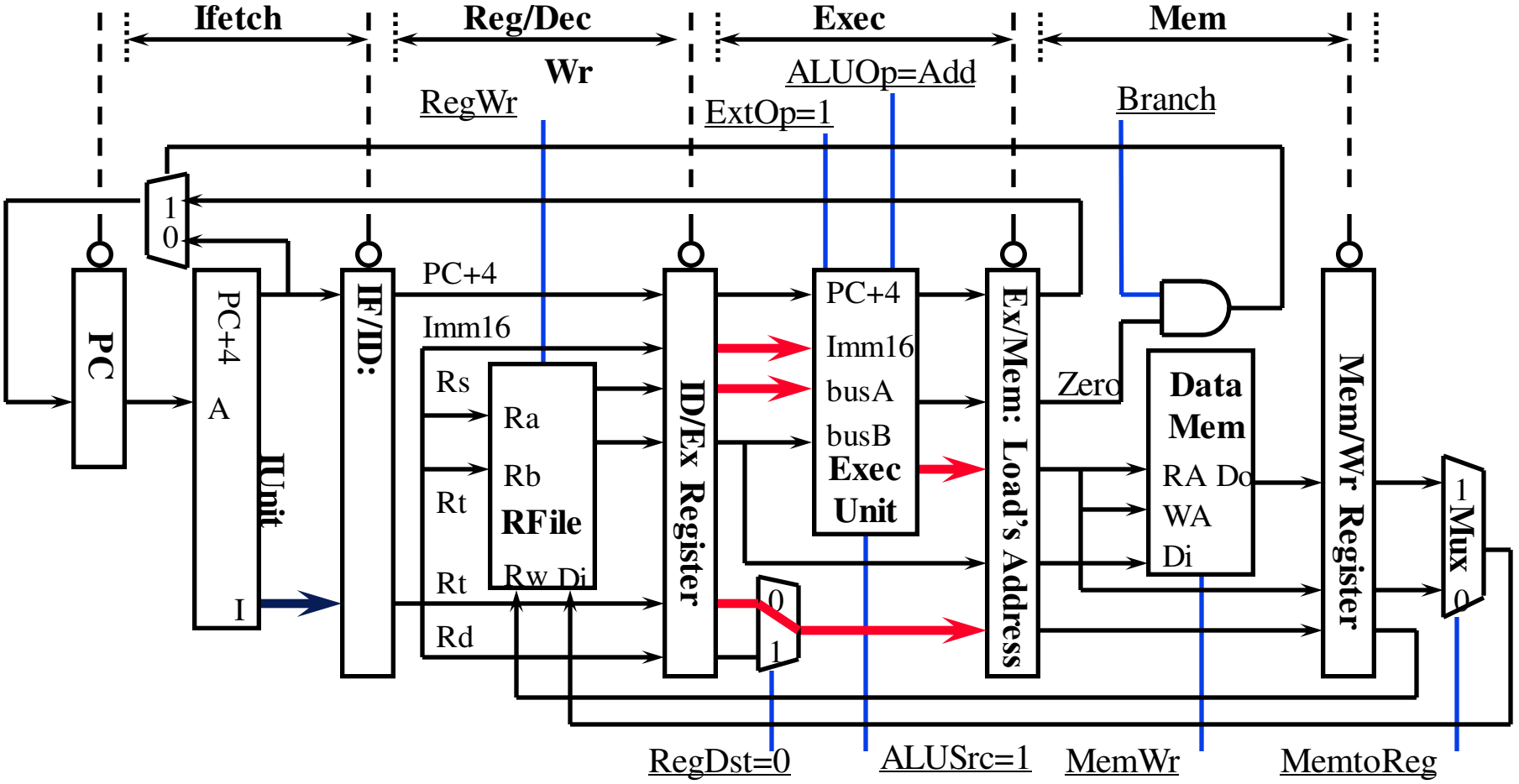
° Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]



How About Control Signals?

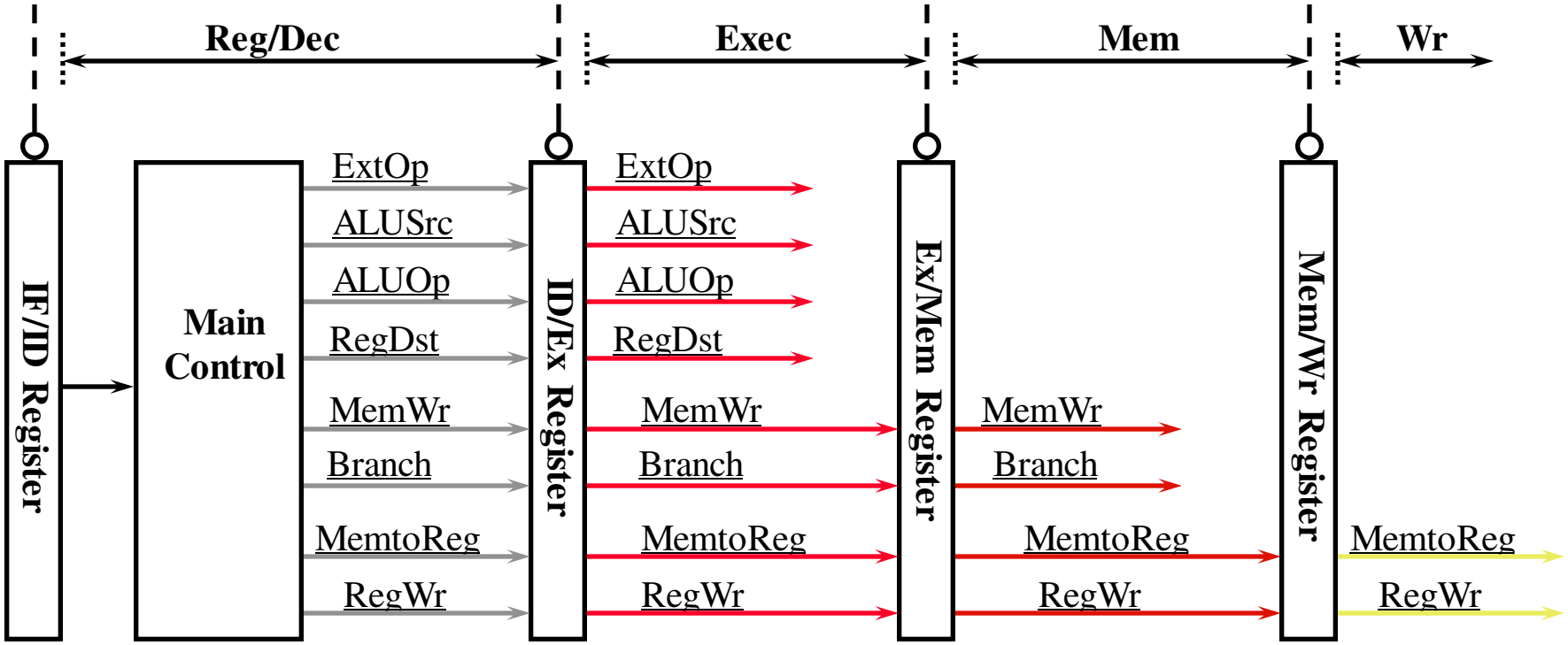
- Key Observation: Control Signals at Stage N = Func (Instr. at Stage N)
  - N = Exec, Mem, or Wr

- Example: Controls Signals at Exec Stage = Func(Load's Exec)

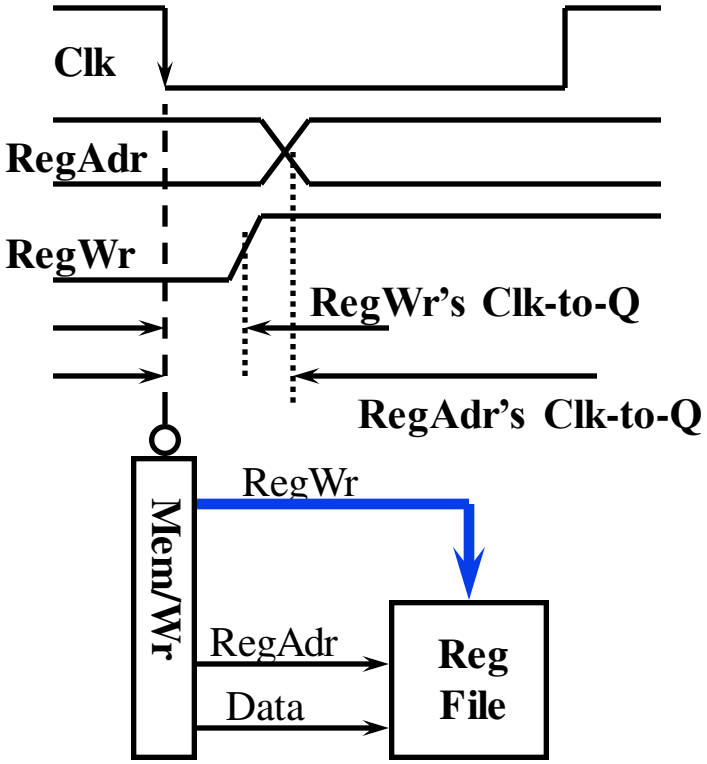


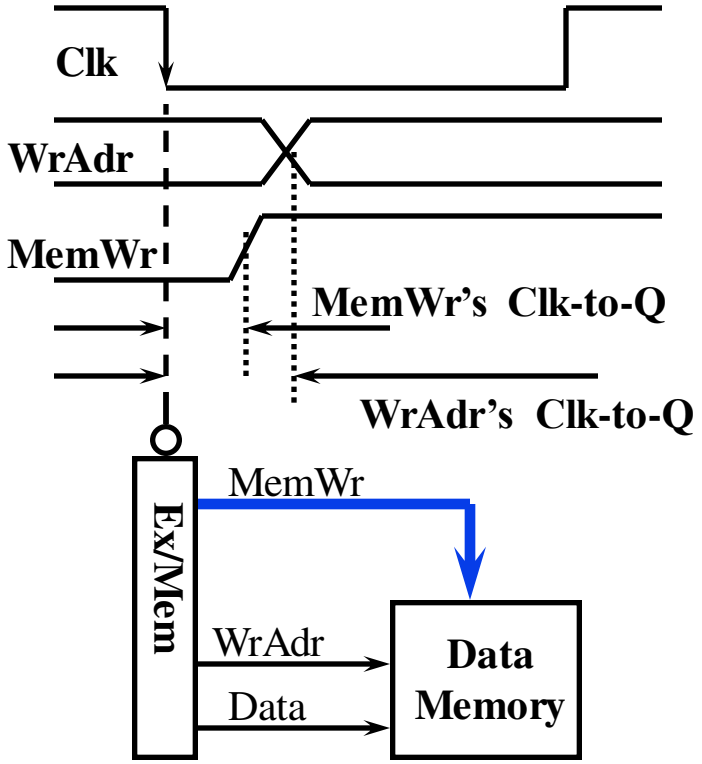
Pipeline Control

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



Beginning of the Wr's Stage: A Real World Problem

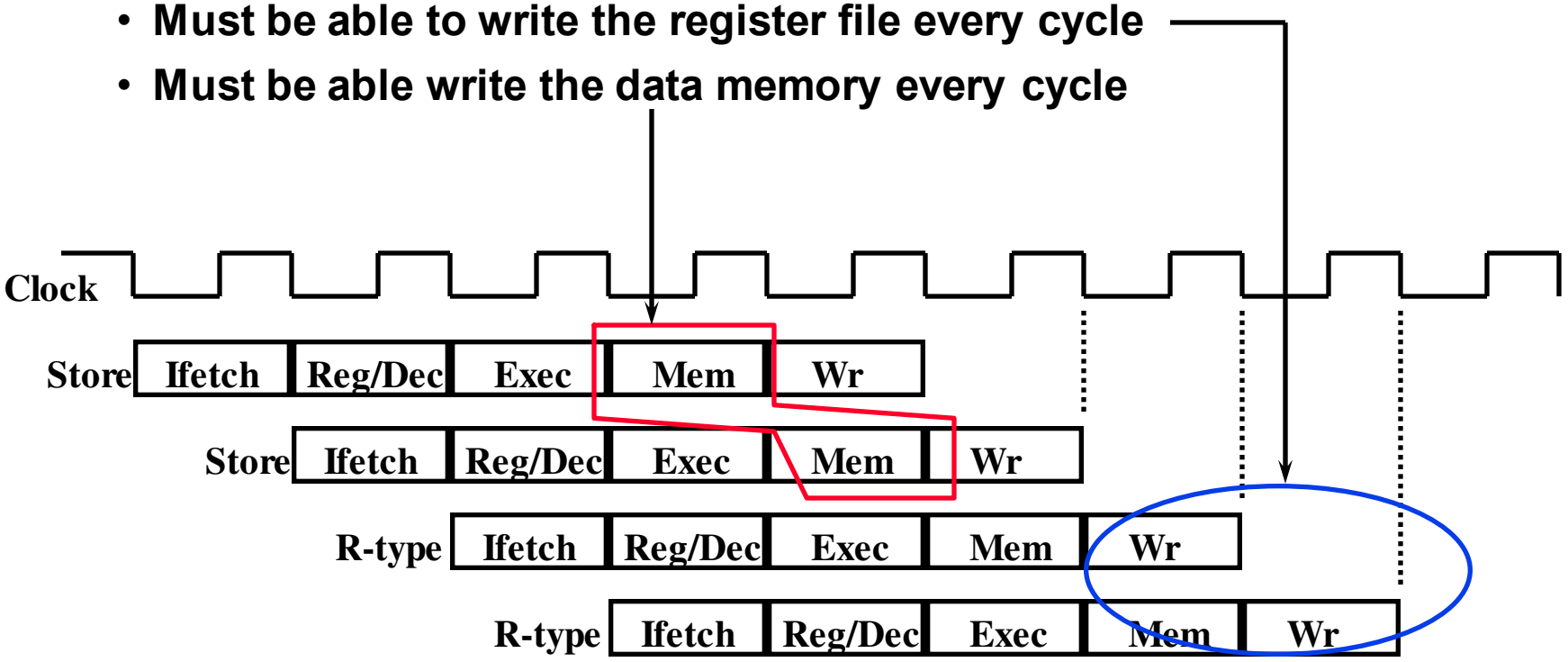




- At the beginning of the Wr stage, we have a problem if:
  - RegAdr's (Rd or Rt) Clk-to-Q > RegWr's Clk-to-Q
- Similarly, at the beginning of the Mem stage, we have a problem if:
  - WrAdr's Clk-to-Q > MemWr's Clk-to-Q
- We have a race condition between Address and Write Enable!

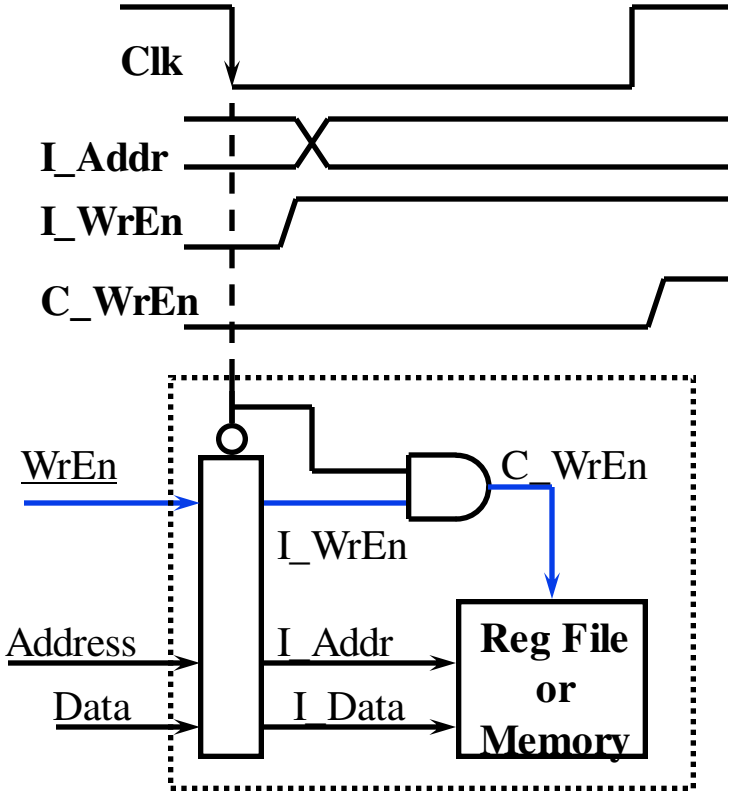
The Pipeline Problem

- Multiple Cycle design prevents race condition between Addr and WrEn:
  - Make sure Address is stable by the end of Cycle N
  - Asserts WrEn during Cycle N + 1
- This approach can NOT be used in the pipeline design because:
  - Must be able to write the register file every cycle
  - Must be able write the data memory every cycle



Synchronize Register File & Synchronize Memory

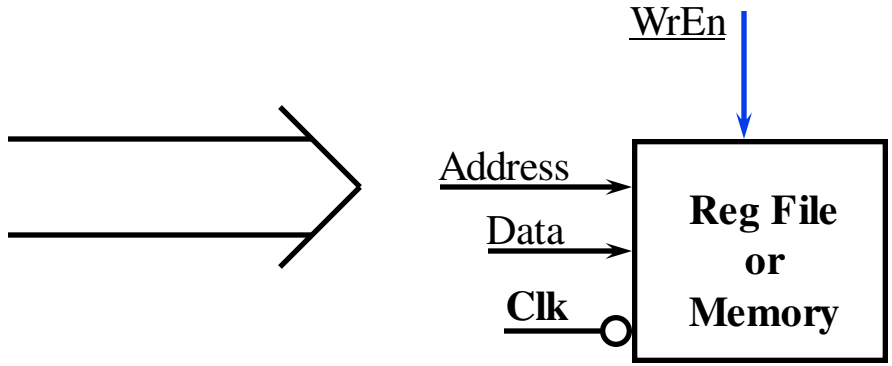
- Solution: And the Write Enable signal with the Clock
  - This is the ONLY place where gating the clock is used
  - MUST consult circuit expert to ensure no timing violation:
    - Example: Clock High Time > Write Access Delay



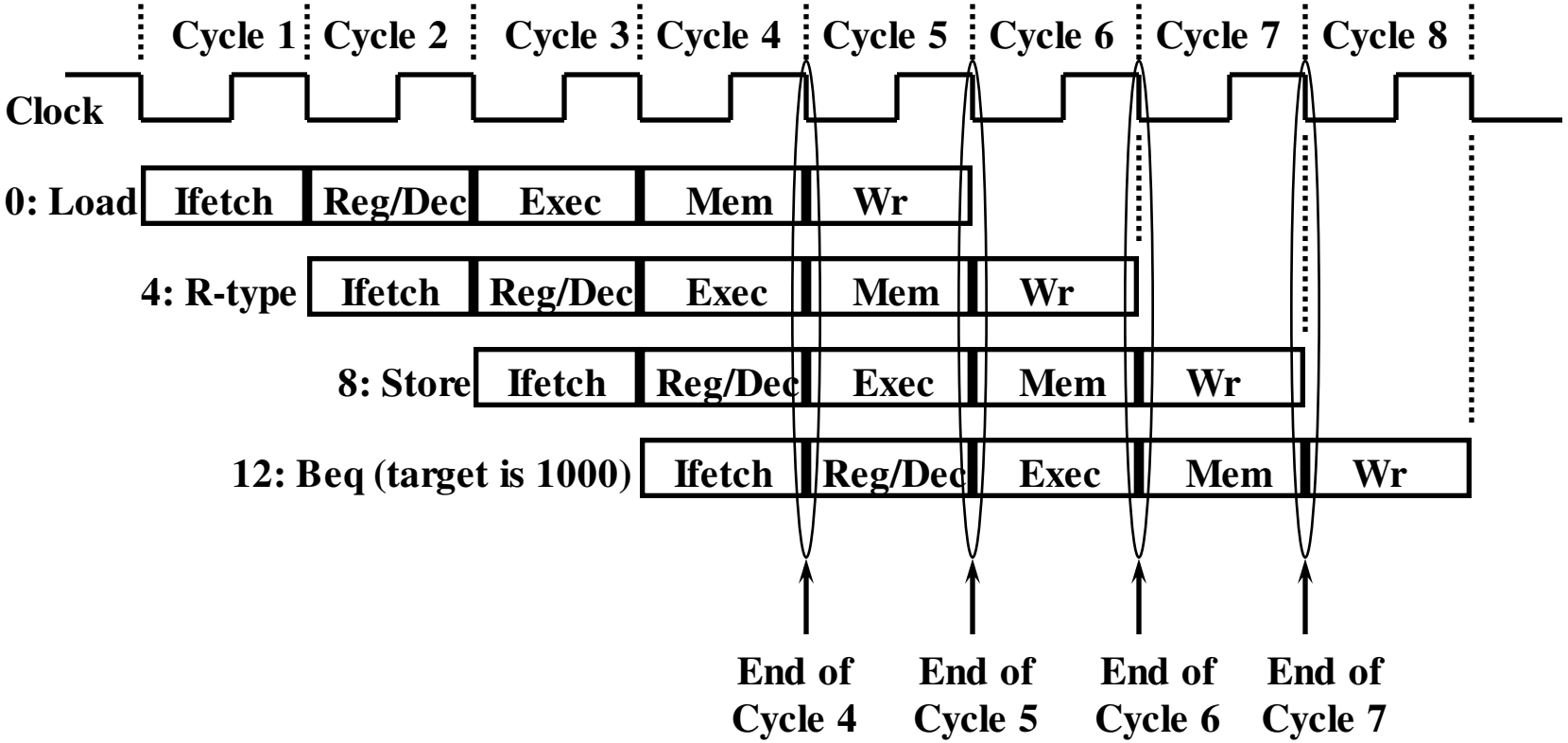
Synchronize Memory and Register File

Address, Data, and WrEn must be stable at least 1 set-up time before the Clk edge

Write occurs at the cycle following the clock edge that captures the signals



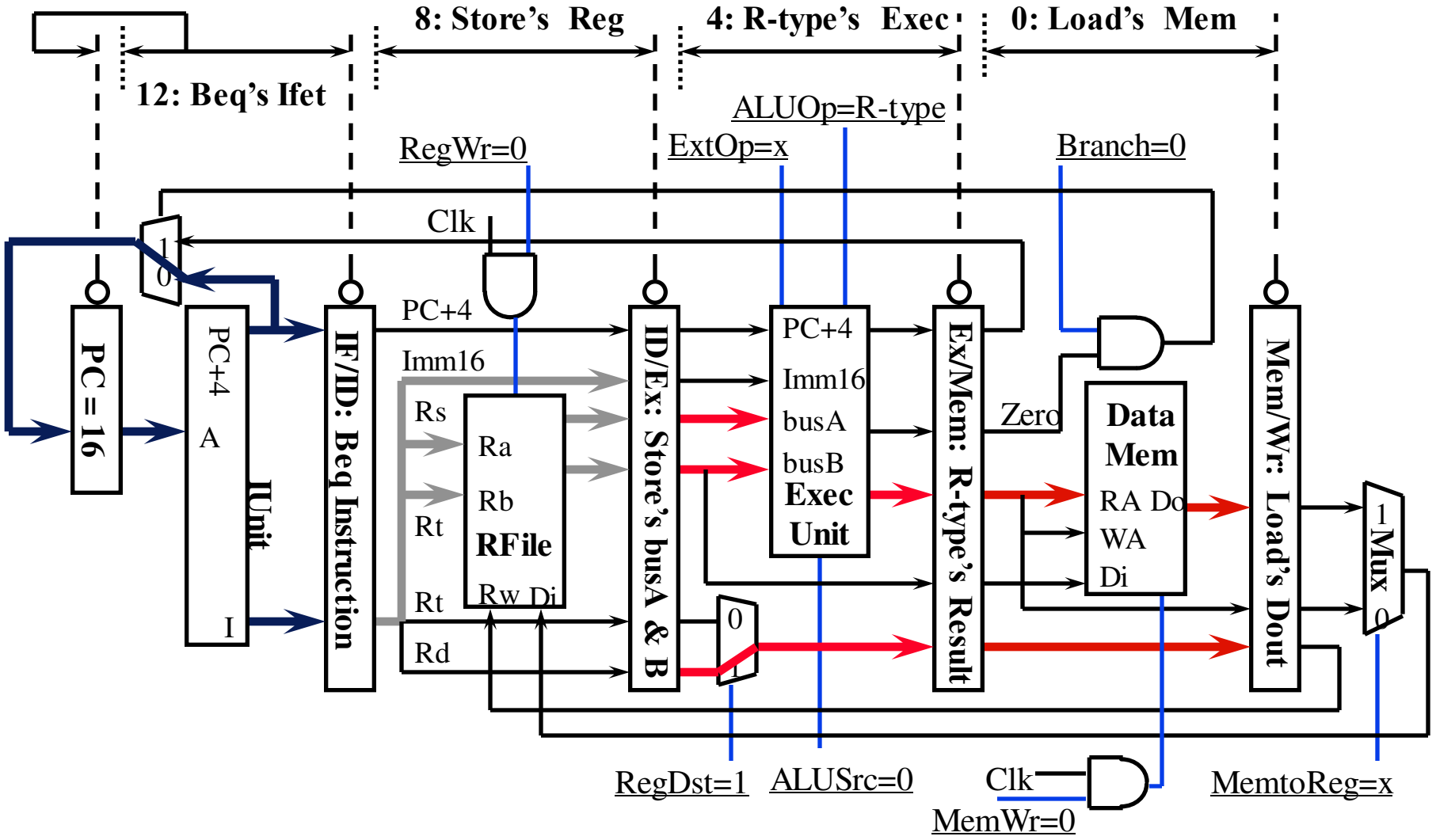
A More Extensive Pipelining Example



- End of Cycle 4: Load's Mem, R-type's Exec, Store's Reg, Beq's Ifetch
- End of Cycle 5: Load's Wr, R-type's Mem, Store's Exec, Beq's Reg
- End of Cycle 6: R-type's Wr, Store's Mem, Beq's Exec
- End of Cycle 7: Store's Wr, Beq's Mem

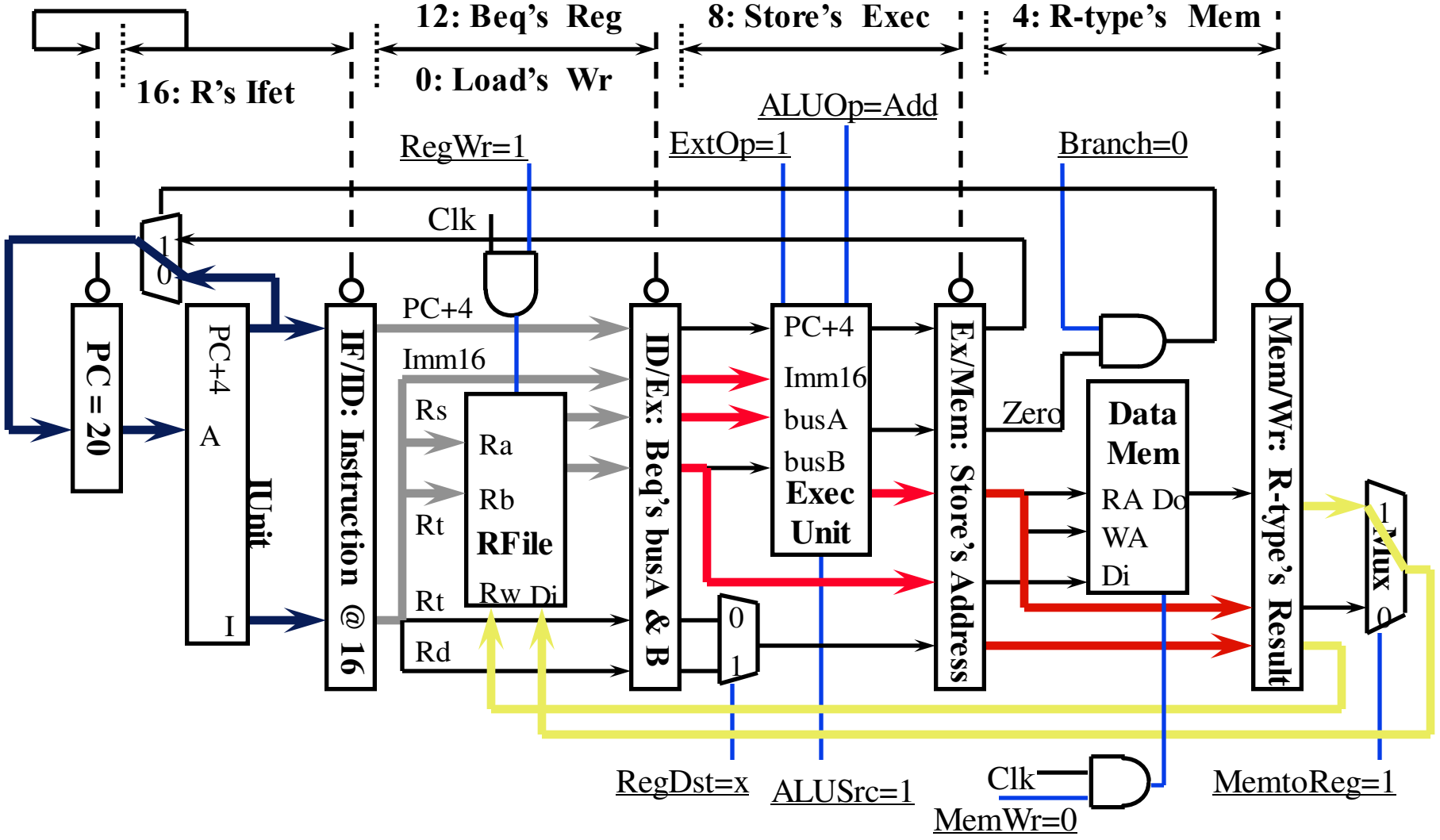
Pipelining Example: End of Cycle 4

° 0: Load's Mem 4: R-type's Exec 8: Store's Reg 12: Beq's Ifetch



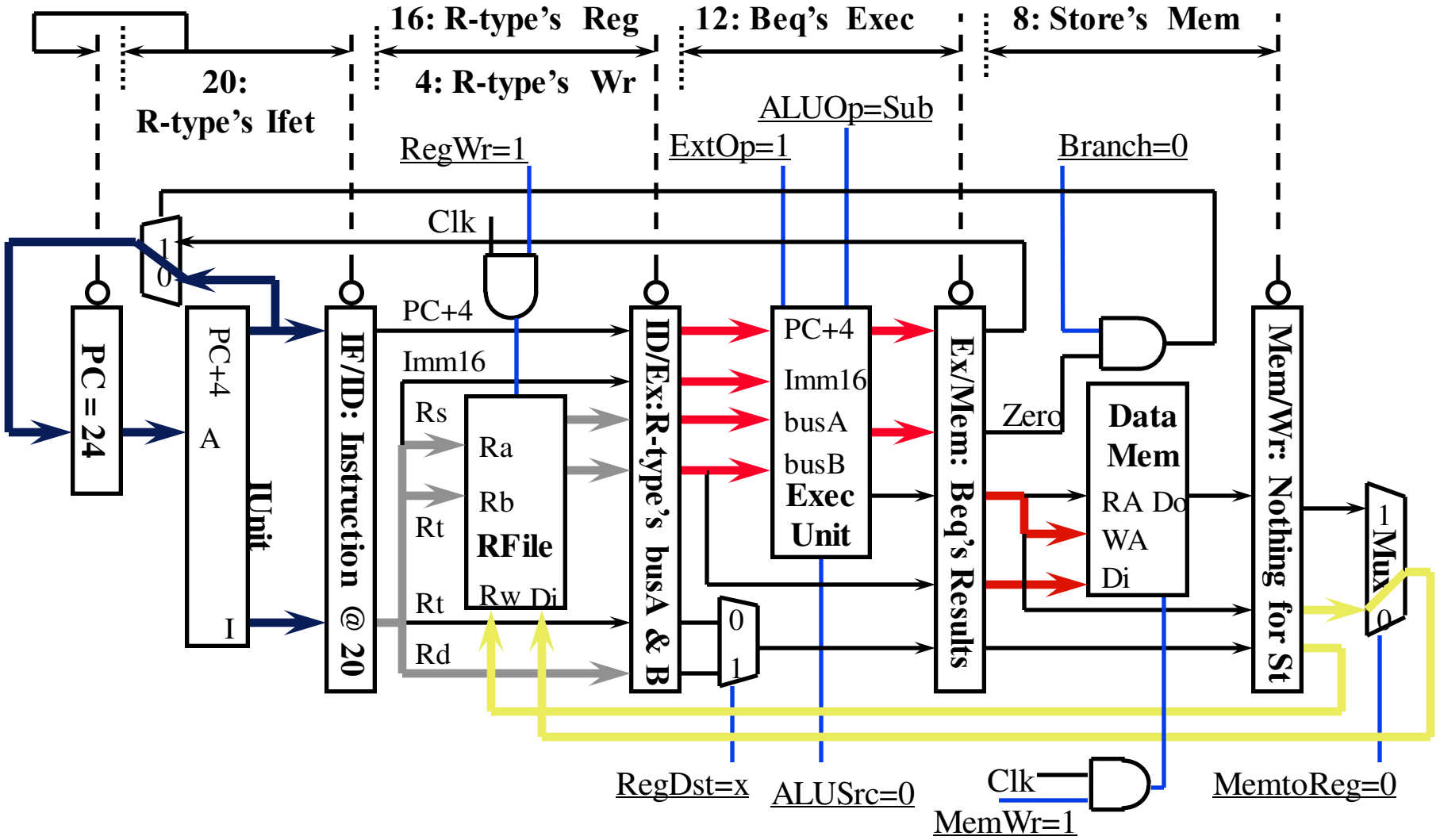
Pipelining Example: End of Cycle 5

° 0: Lw's Wr 4: R's Mem 8: Store's Exec 12: Beq's Reg 16: R's Ifetch



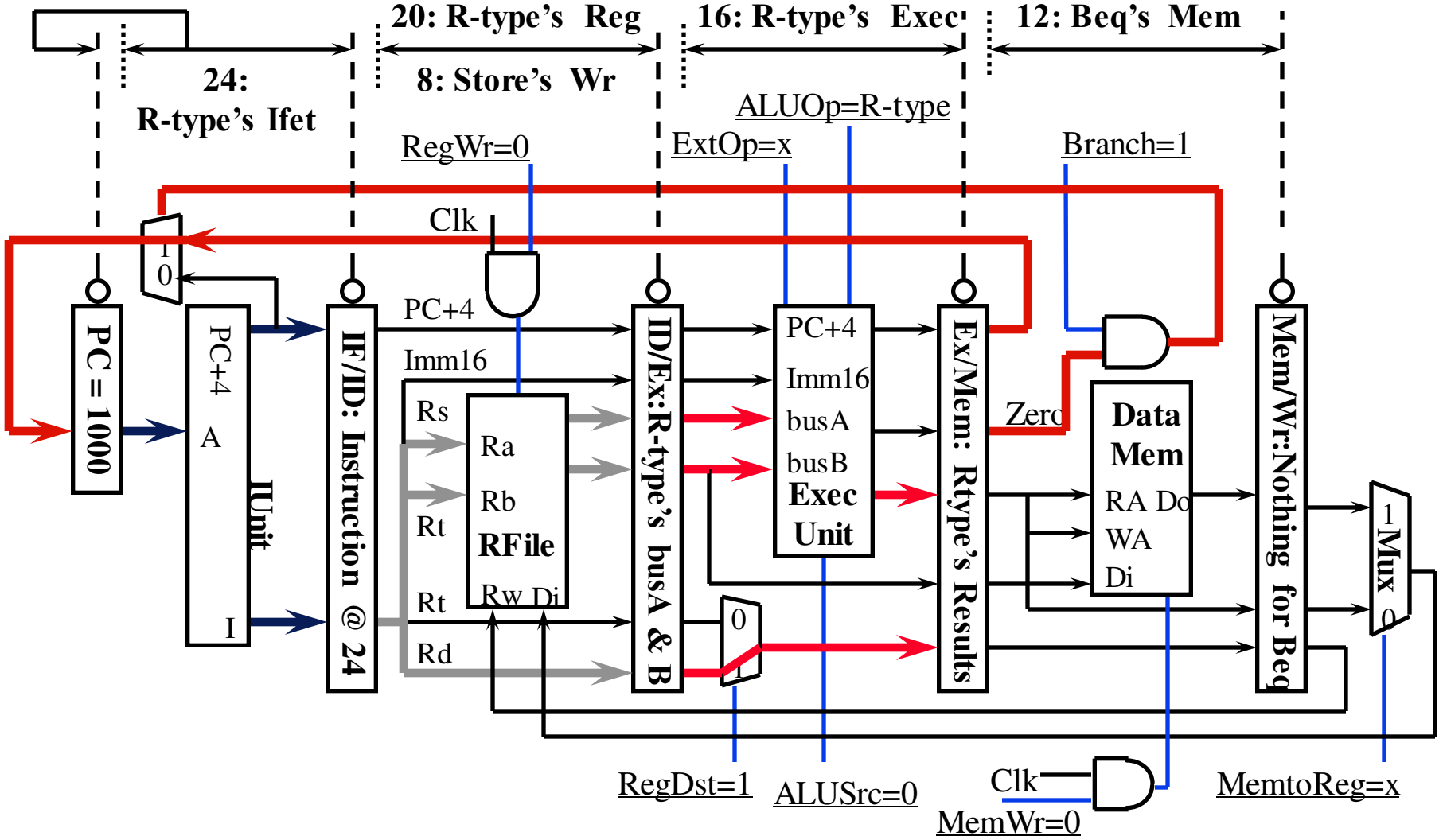
Pipelining Example: End of Cycle 6

° 4: R's Wr 8: Store's Mem 12: Beq's Exec 16: R's Reg 20: R's Ifet

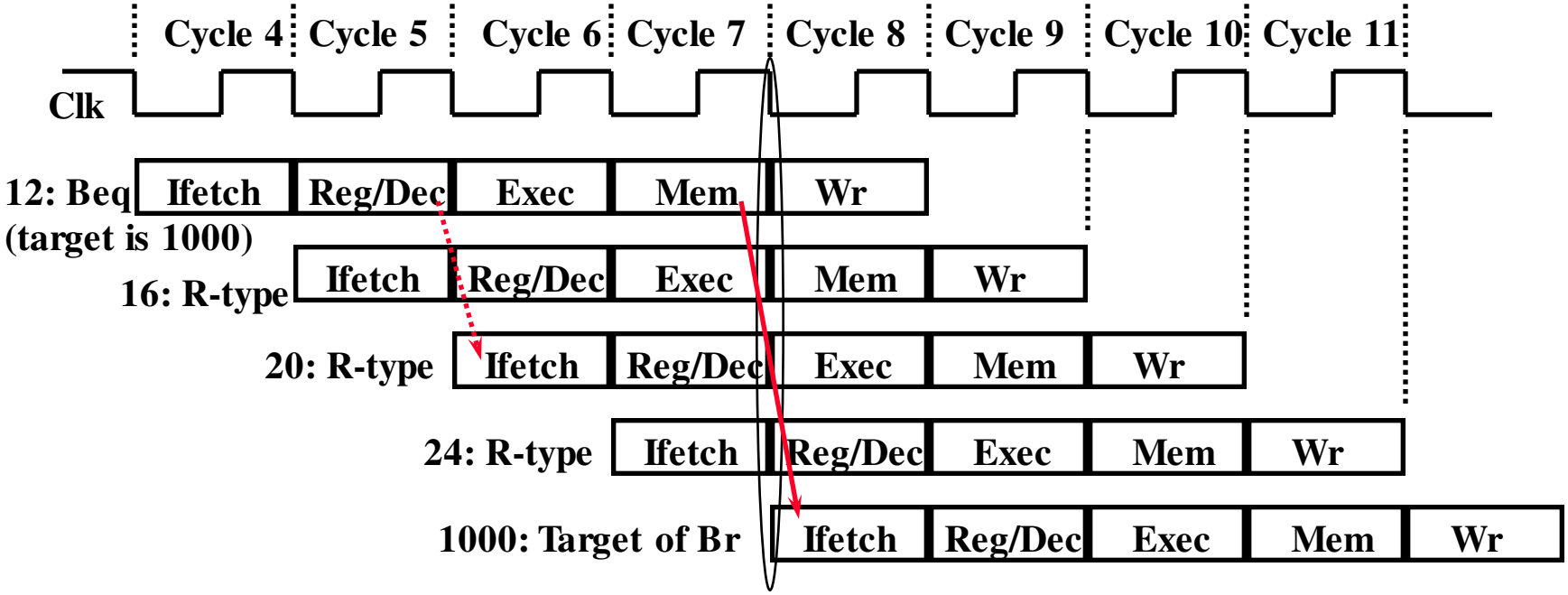


Pipelining Example: End of Cycle 7

° 8: Store's Wr 12: Beq's Mem 16: R's Exec 20: R's Reg 24: R's Ifet

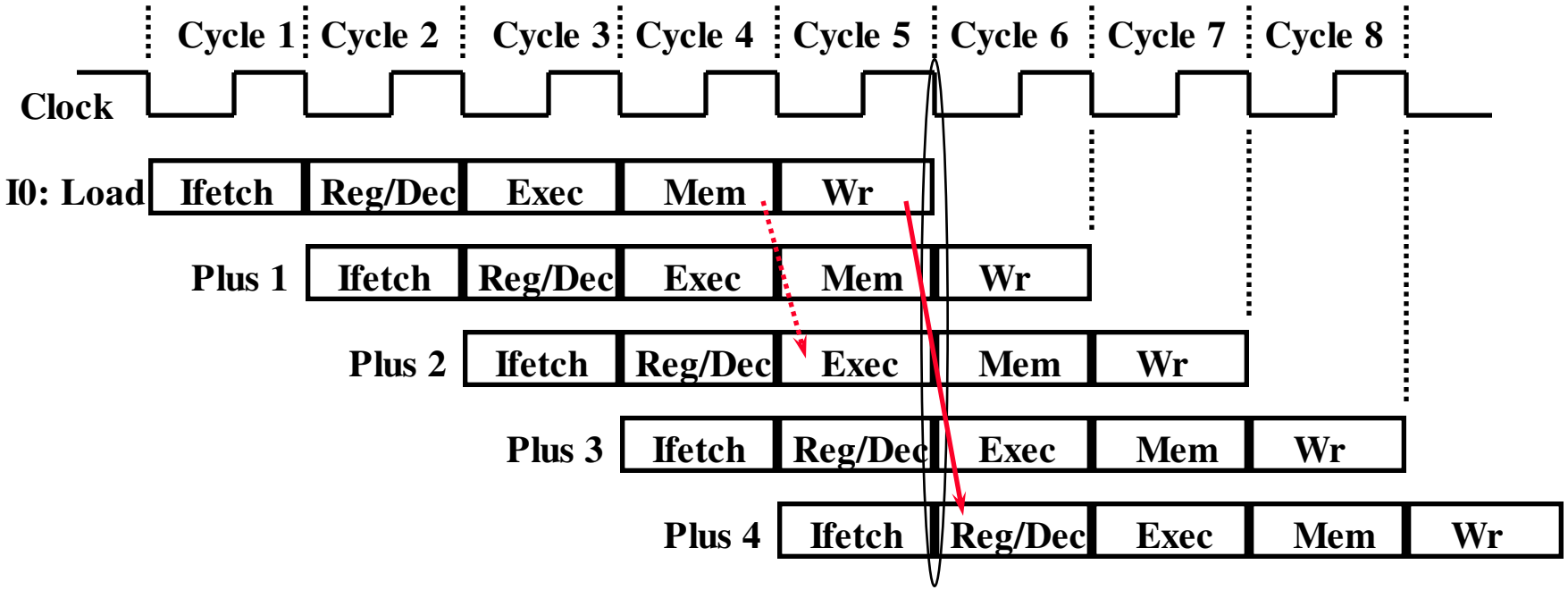


The Delay Branch Phenomenon



- Although Beq is fetched during Cycle 4:
  - Target address is NOT written into the PC until the end of Cycle 7
  - Branch's target is NOT fetched until Cycle 8
  - 3-instruction delay before the branch take effect
- This is referred to as Branch Hazard:
  - Clever design techniques can reduce the delay to ONE instruction

The Delay Load Phenomenon



- Although Load is fetched during Cycle 1:
  - The data is NOT written into the Reg File until the end of Cycle 5
  - We cannot read this value from the Reg File until Cycle 6
  - 3-instruction delay before the load take effect
- This is referred to as Data Hazard:
  - Clever design techniques can reduce the delay to ONE instruction

Soru

PIPELINING

For all following questions we assume that:

a) Pipeline contains 5 stages: IF, ID, EX, M and W;
b) Each stage requires one clock cycle;
c) All memory references hit in cache;
d) Following program segment should be processed:

```
// ADD TWO INTEGER ARRAYS
        LW     R4     # 400
L1:     LW     R1,    0 (R4)      ; Load first operand
        LW     R2, 400 (R4)       ; Load second operand
        ADDI   R3, R1, R2         ; Add operands
        SW     R3,    0 (R4)      ; Store result
        SUB    R4, R4, #4         ; Calculate address of next element
        BNEZ   R4, L1             ; Loop if (R4) != 0
```

SORU 4 :

a) Klasik (nonpiplened) mimari üzerinde üstteki program segmenti kaç saat çevriminde icra edilir?

YANIT: Saat çevrimi sayısı (clock cycle)

= [1. Komut + Döngüdeki komut sayısı x döngü çevrim sayısı]x CPI

= [1 + 6 x 400 /4] x 5

=3005 clock cycle

b) Pipelined mimari üzerinde üstteki program segmenti kaç saat çevriminde icra edilir? Döngü işlemlerini zamanlama diyagramı üzerinde gösteriniz?

Instruction	Clock cycle number															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LW R1, 0 (R4)																	
LW R2, 400 (R4)																	
ADDI R3,R1,R2																	
SW R3, 0 (R4)																	
SUB R4, R4, #4																	
BNEZ R4, L1																	

YANIT:

Instruction	Clock cycle number															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
LW R1, 0 (R4)	IF	ID	Ex	M	W											
LW R2,400(R4)		IF	ID	Ex	M	W										
ADDI R3,R1,R2			IF	ID	*	*	Ex	M	W							
SW R3, 0 (R4)				IF	*	*	ID	Ex	*	M	W					
SUB R4, R4, #4							IF	ID	*	Ex	M	W				
BNEZ R4, L1								IF	*	ID	*	*	Ex	M	W	

3.komut ID stage sonrası 5 ve 6. Çevrimlerde STALL (gecikme) olması gerekir.. Toplama yapabilmek için R1 ve R2 içinde operandların bulunması gerekir..

4.komut için 9.cc de STALL yapılır. (9.cc de ADD komutu ile R3'e sonuç yazdırıldığı için)

6. komut BNEZ 11 ve 12 cc da STALL yapılır, R4 içeriği 12. cc sonunda aktifleşmektedir.

Döngü için 15 cc gerekmektedir.
Saat çevrimi sayısı (clock cycle)

= [1. Komut IF katı + Döngüdeki saat çevrimi sayısı x döngü sayısı]x CPI

= [1 + 15x 400/4]x 1

= 1501 cc

c)
Speedup= non-pipelined cc sayısı / pipelined cc sayısı = 3005 / 1501 = 2 (yaklaşık)

Summary

- Disadvantages of the Single Cycle Processor
  - Long cycle time
  - Cycle time is too long for all instructions except the Load
- Multiple Clock Cycle Processor:
  - Divide the instructions into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
- Pipeline Processor:
  - Natural enhancement of the multiple clock cycle processor
  - Each functional unit can only be used once per instruction
  - If a instruction is going to use a functional unit:
    - it must use it at the same stage as all other instructions
  - Pipeline Control:
    - Each stage's control signal depends ONLY on the instruction that is currently in that stage