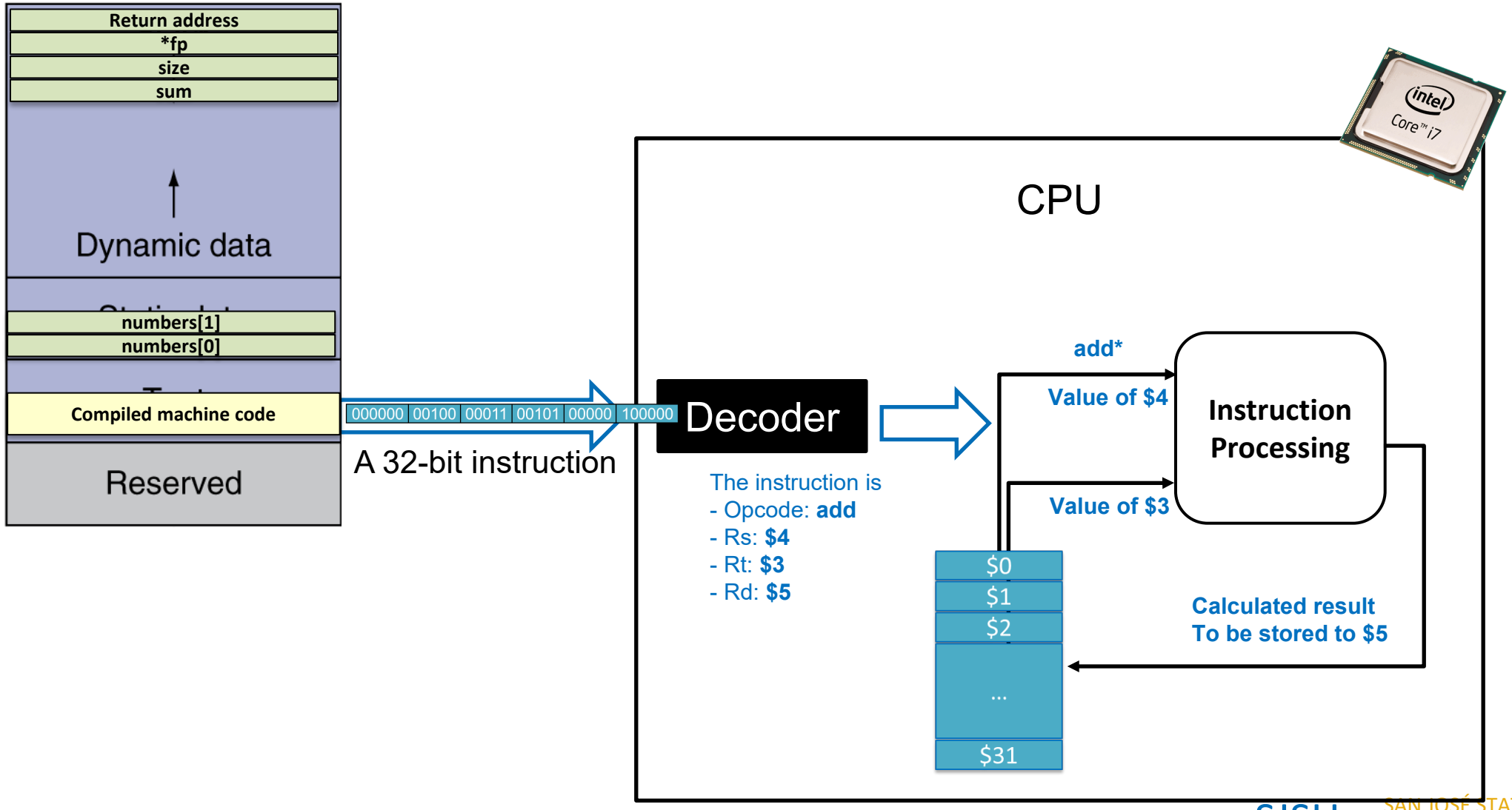
CMPE 200
Computer Architecture & Design

Lecture 3.
Processor Microarchitecture and Design (1)

Haonan Wang

SAN JOSÉ STATE
UNIVERSITY

Instruction Processing



Instruction Execution on CPU

- To run a software, CPU
  - **Fetches** instructions from memory in the right order
  - **Decodes** the instruction and **Reads** the operand values from register file
  - **Executes** the computation by using ALU
  - **Loads/store** data from/to memory
  - **Writes back** the computation results to register file

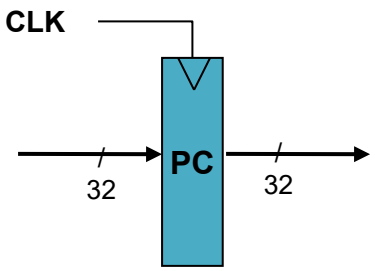
Implementing a Single-Cycle CPU

- **All instructions execute in a single clock cycle**
  - Instructions to Implement: LW, SW; ADD, SUB, AND, OR, SLT; BEQ
  - CPI = 1
- **Different instructions need different execution times**
  - i.e. LW needs to access memory *vs.* ADD does not need to access memory
- **Single-cycle CPU's clock cycle period should be set to the longest instruction's execution time**

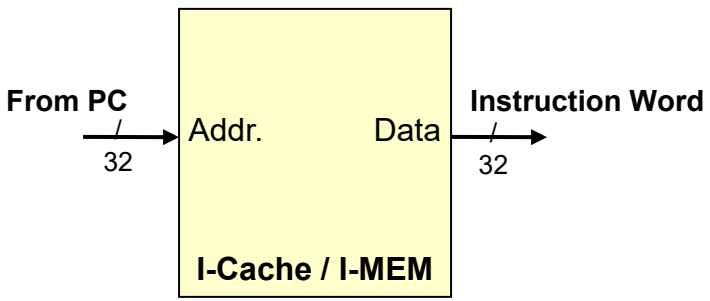
Fetch Components

- **Required operations**
  - Taking address from PC and reading instruction from memory
  - Incrementing PC to point at next instruction

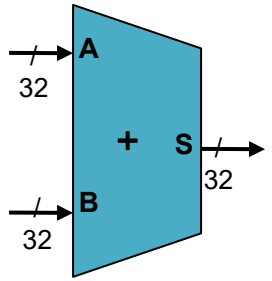
- **Components**
  - PC register
  - Instruction Memory / Cache
  - Adder to increment PC value



Register

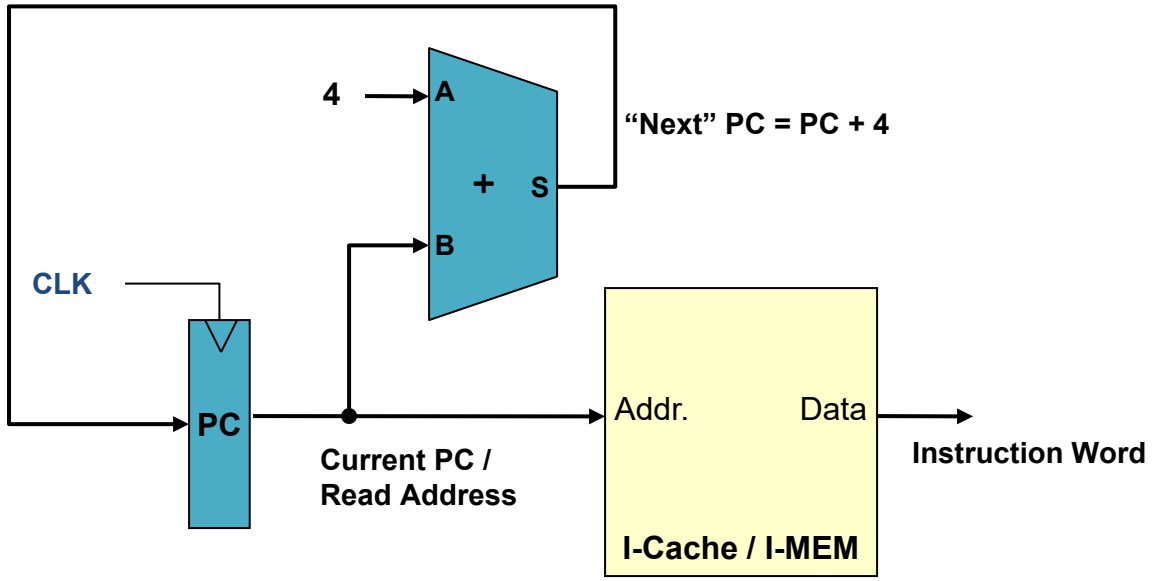


Memory



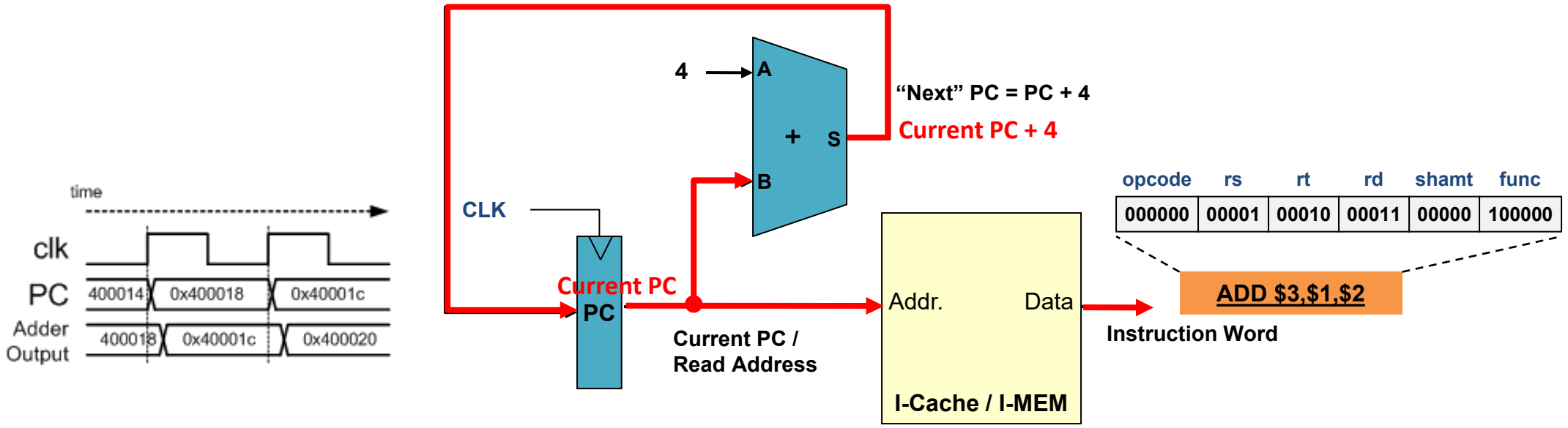
Adder

Fetch Data Path



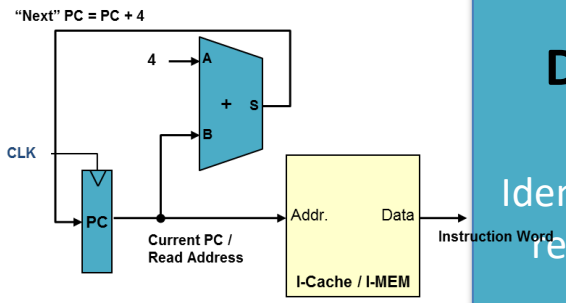
Fetch Data Path

- PC value serves as address to instruction memory while also being incremented by 4 using the adder
- Instruction word is returned by memory after some delay
- New PC value is clocked into PC register at the end of clock cycle



Decode

- **What is the instruction and operands?**



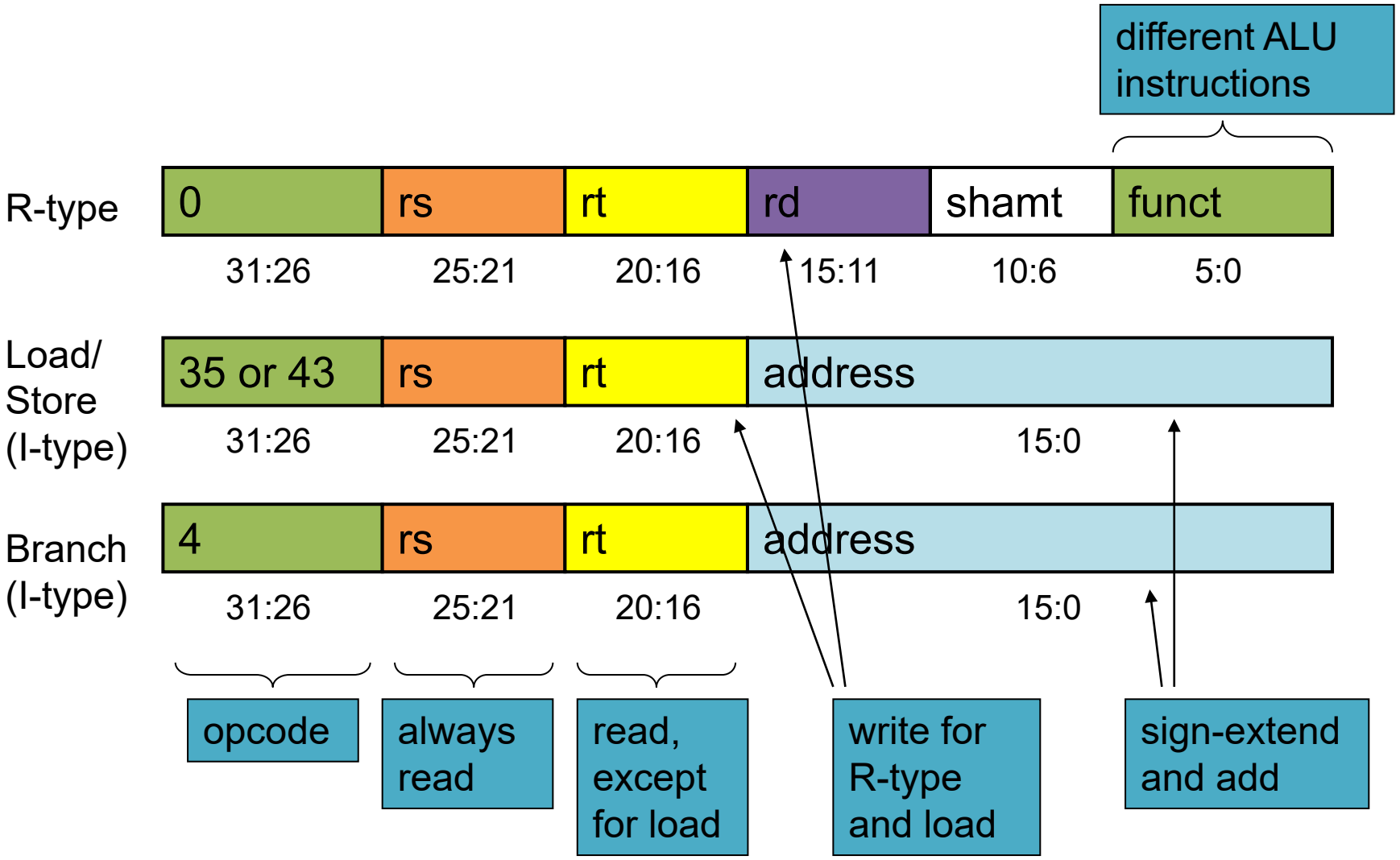
Decode & Register
Identify opcode & read operands

Execute

Mem

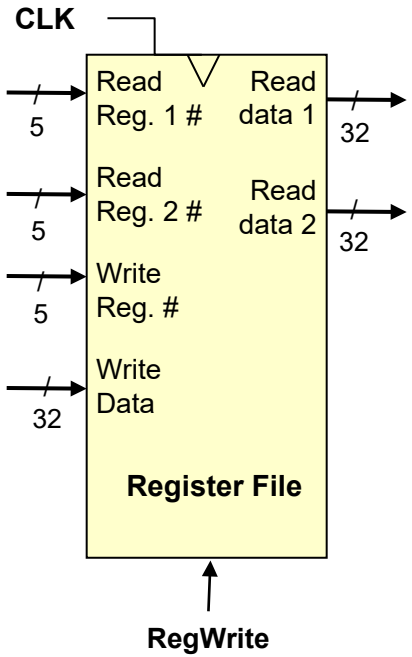
Write Result
Write to dst register

Instruction Formats



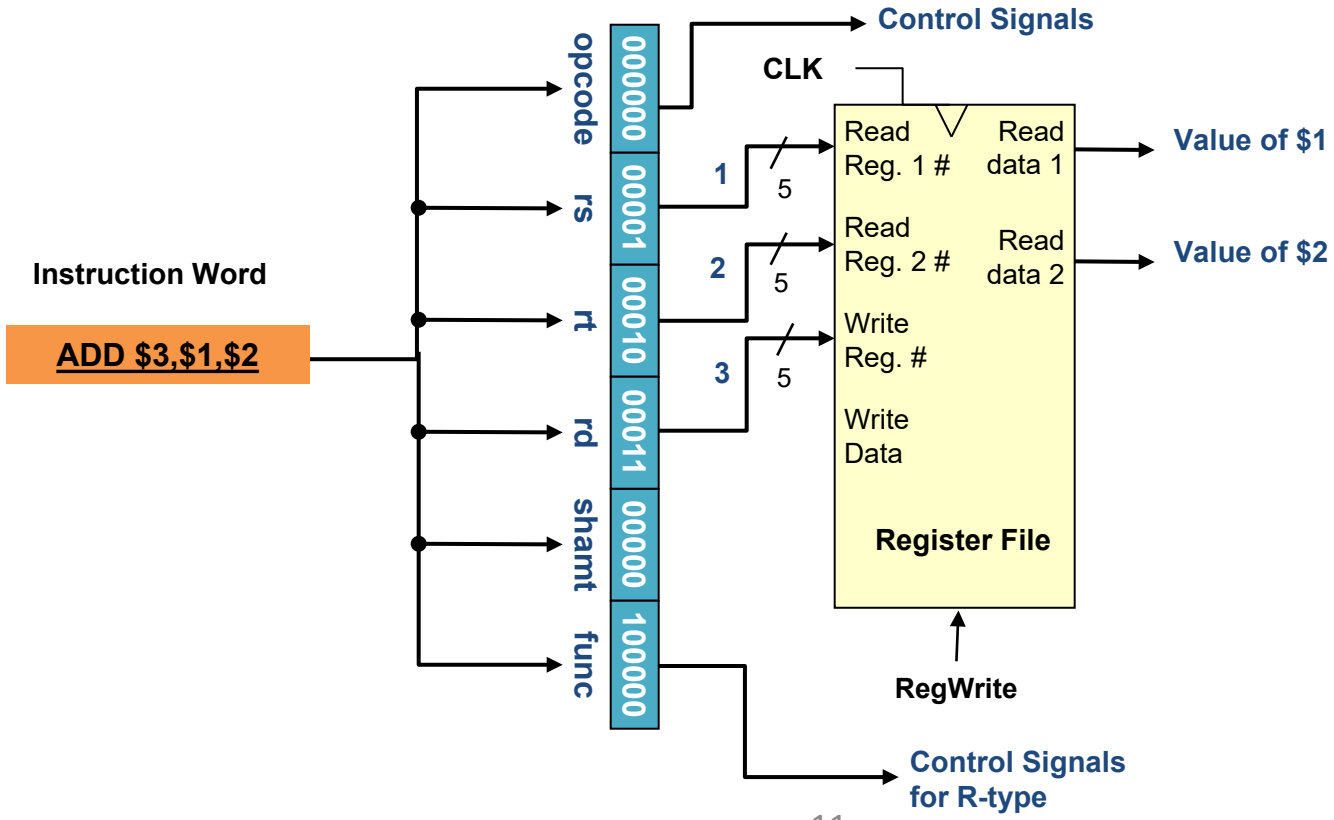
Decode Components

- **Register file is a collection of registers**
- **To support R-type instruction,**
  - 4 Input ports : two src register ids, a dst register id and the result value
  - 2 Output ports : two operand values



Decode Data Path

- **R-type**
  - Opcode and func. field are decoded to produce control signals
  - Three register ids are passed to Register file



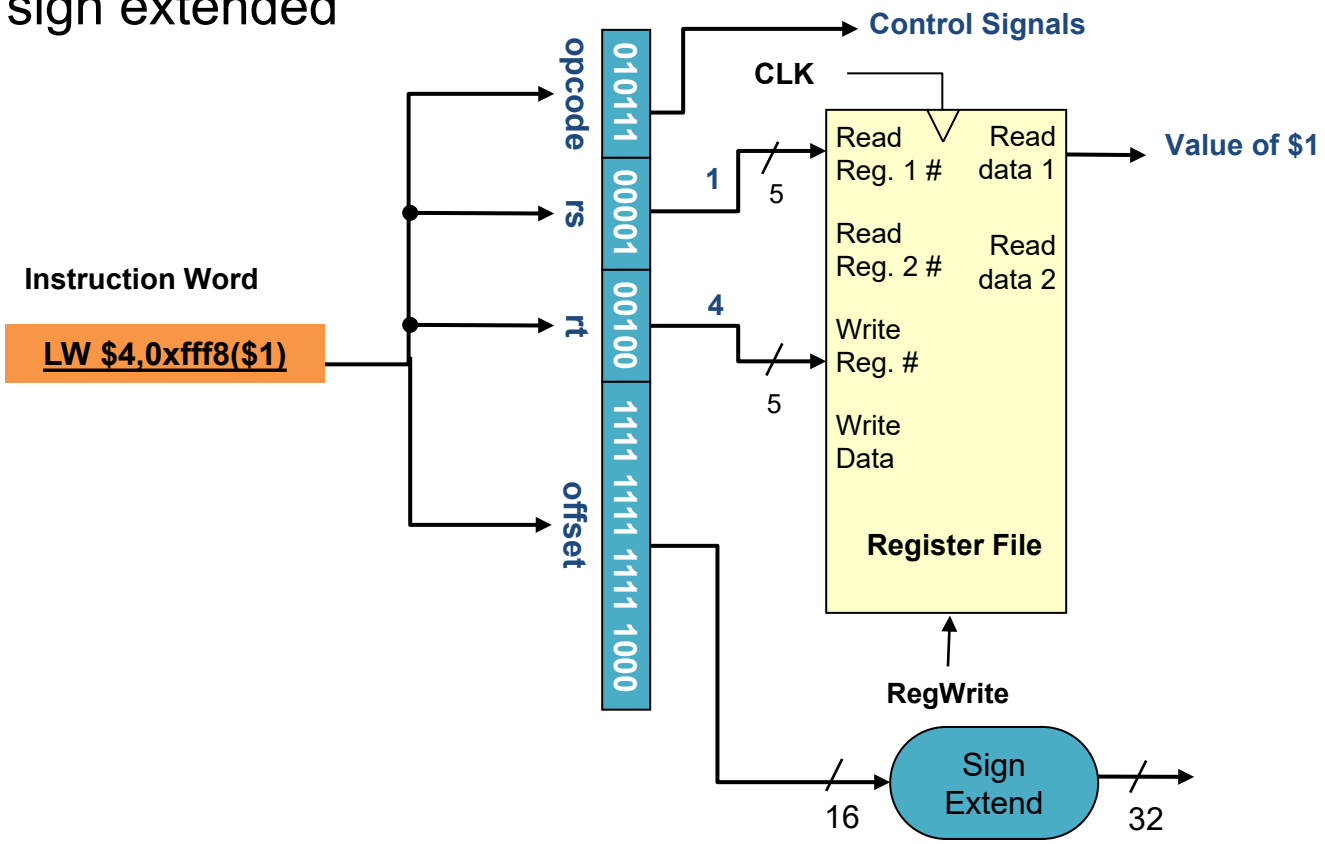
Sign Extension Unit

- **In a 'LW' or 'SW' instructions with their base register + offset format, the instruction only contains the offset as a 16-bit value**
  - Example: LW $4,-8($1)
  - Machine Code: 0x8c24fff8

- **A 16-bit offset must be extended to 32-bits to add to the base register**

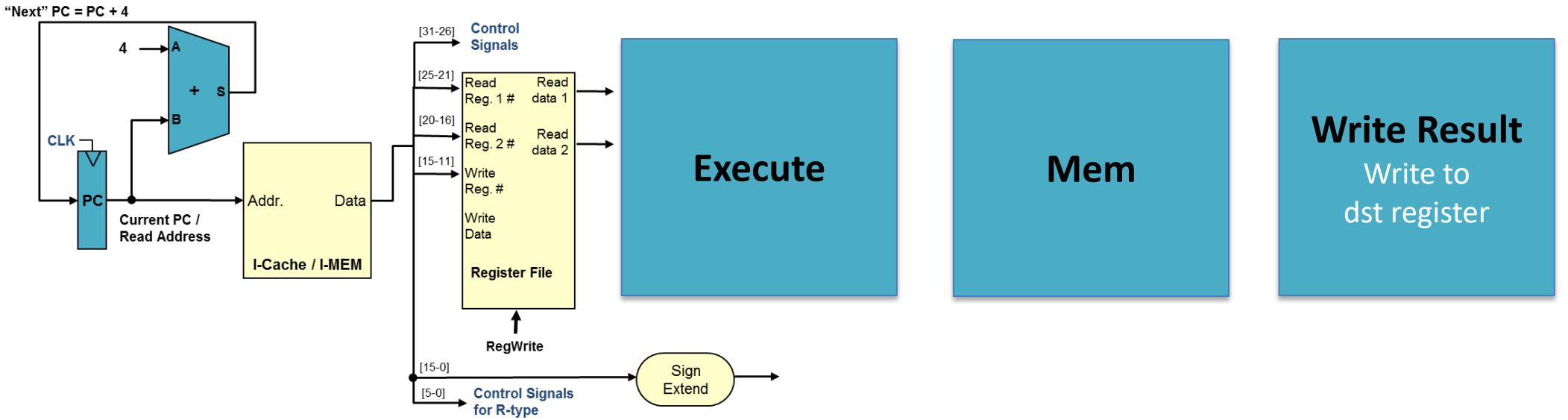
Decode Data Path

- **I-type**
  - Opcode field is decoded to produce control signals
  - Two register ids are passed to Register file
  - Offset is sign extended



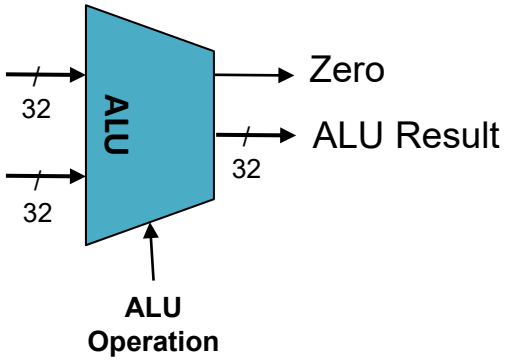
Execution & Write Result

- **Let's execute the instruction!**

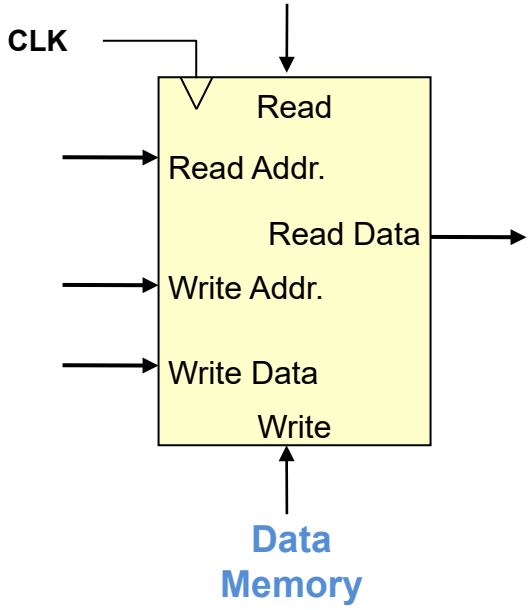


Execution Components

- **ALU is used for**
  - R-type instructions to operate various arithmetic operations
  - LW/SW instructions to operate (base address + offset) to generate the target memory address
  - Branch instructions to compare (subtract) the two operand values
- **Data Memory for LW/SW instructions**
- **Shift-left-2 and another Adder for Branch instructions to calculate new PC value**

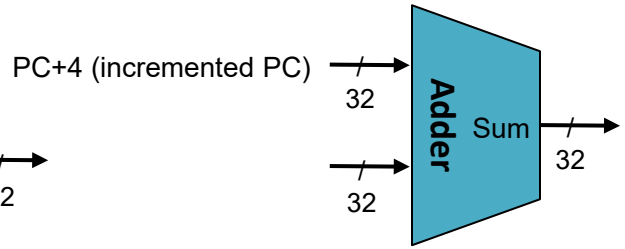


ALU



Data Memory

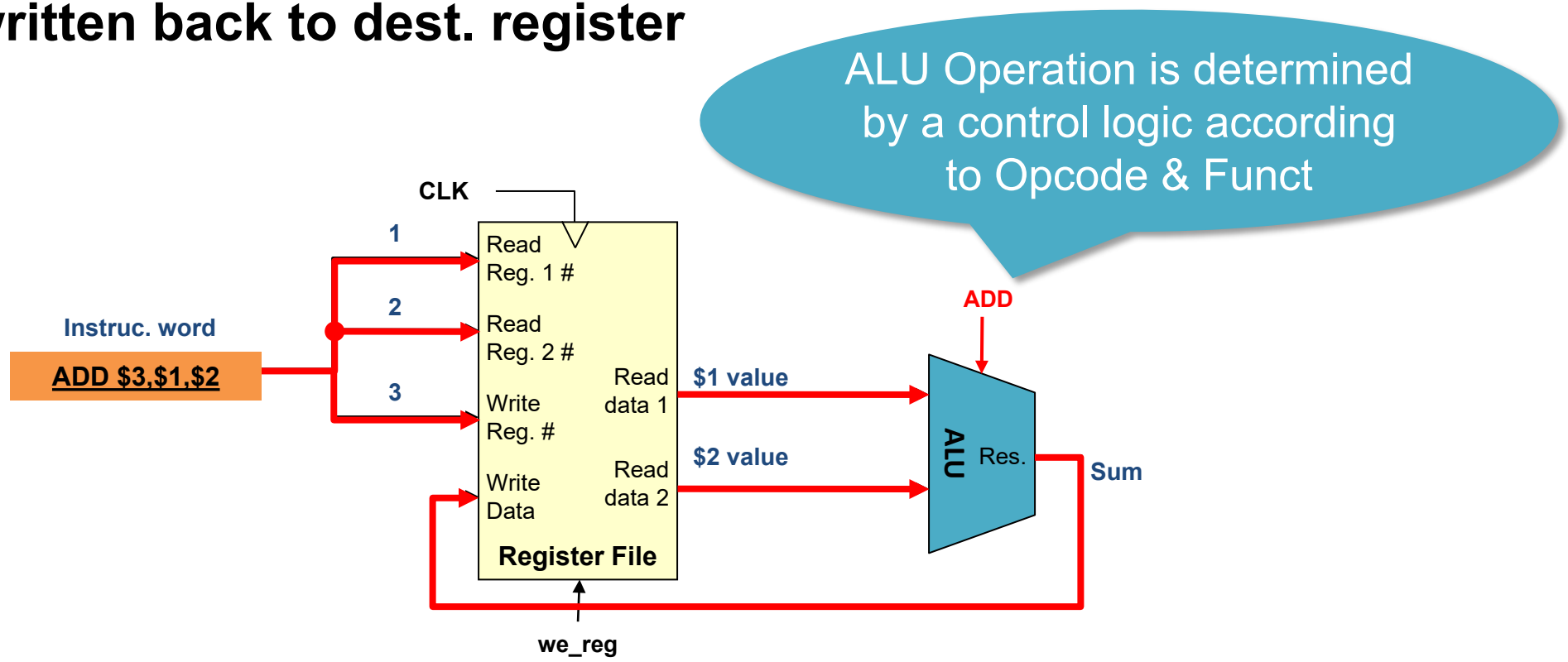
Shift-left-2



Adder

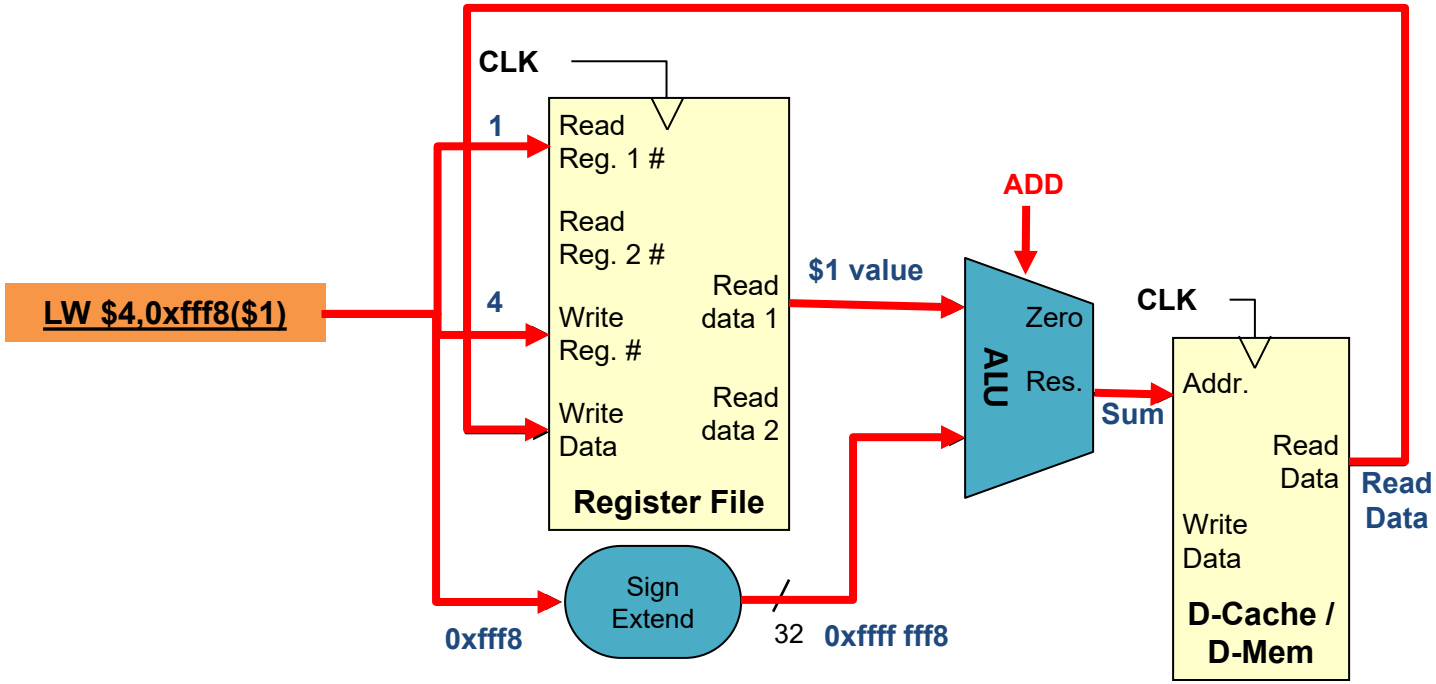
Datapath for R-type Instructions

- **ALU takes inputs from register file and performs the add, sub, and, or, slt, operations**
- **Result is written back to dest. register**



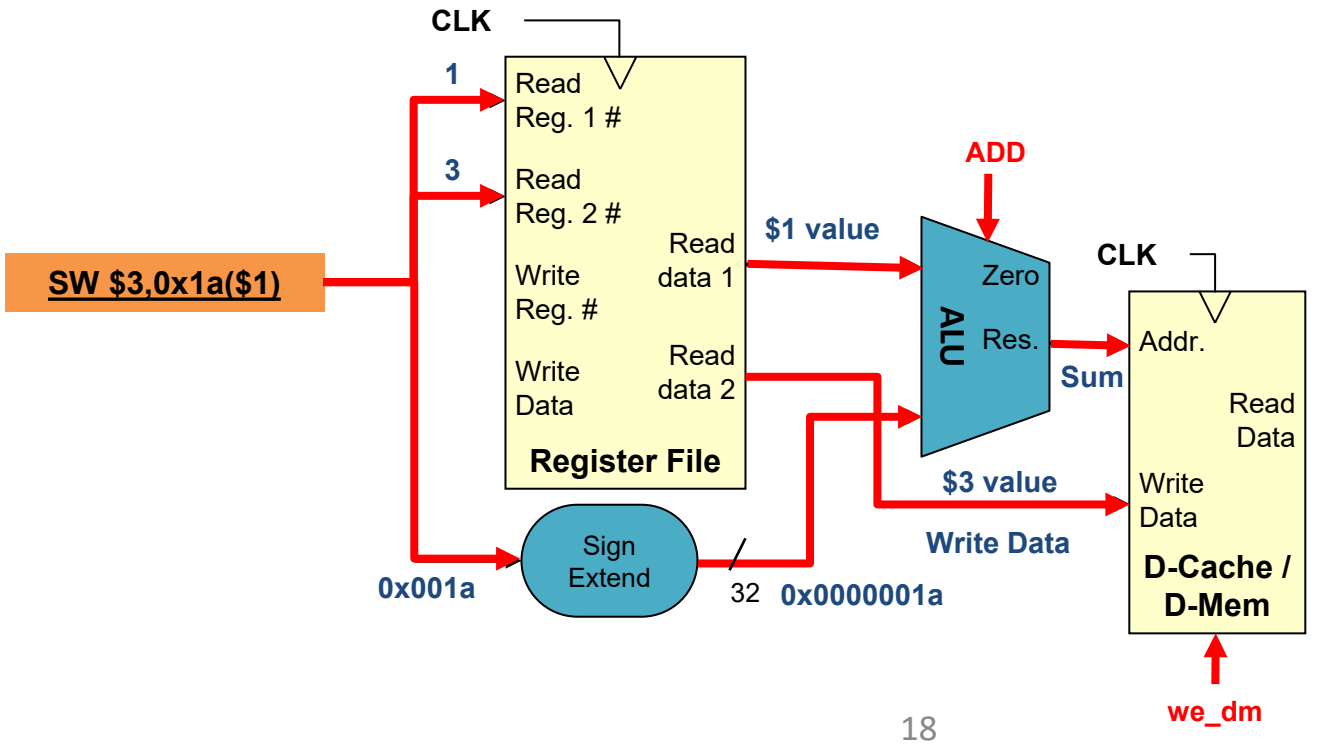
Memory Access Datapath: LW

- Operands are read from register file while offset is sign extended
- ALU calculates target memory address
- Memory access is performed
- If LW, read data is written back to register



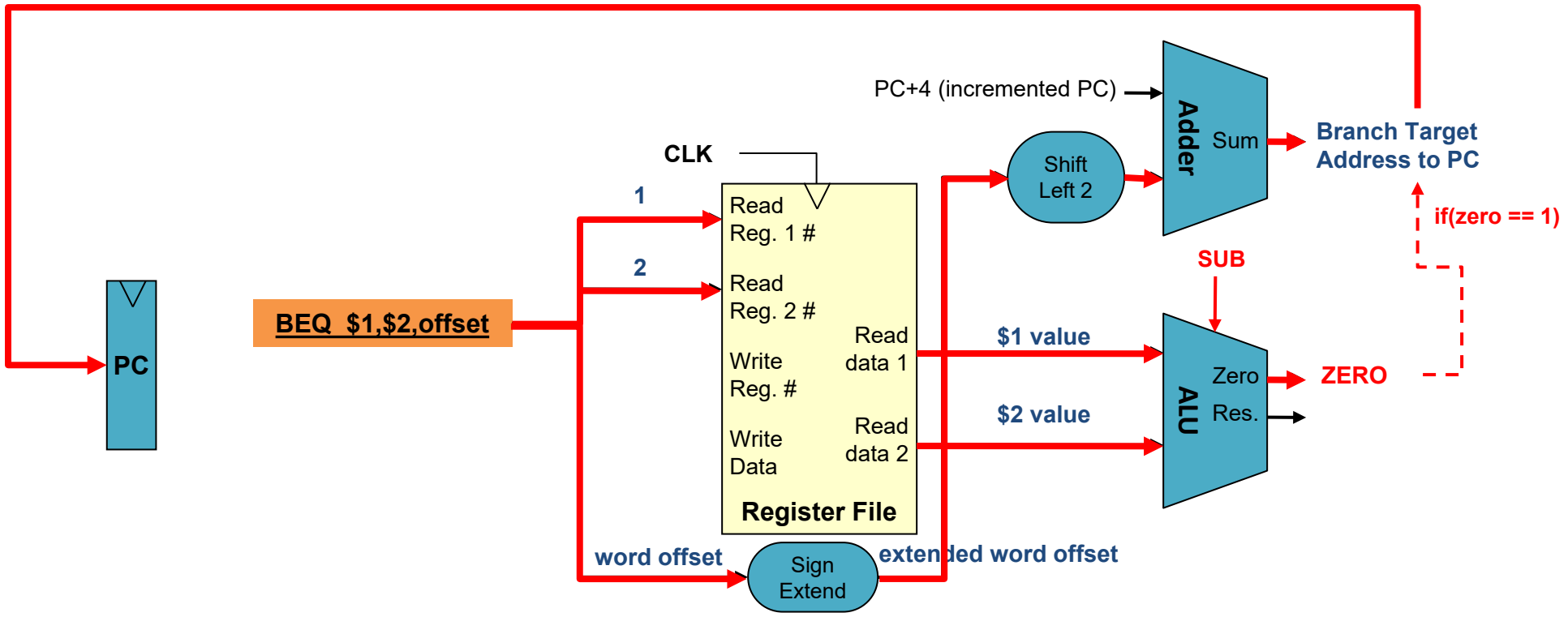
Memory Access Datapath: SW

- Operands are read from register file while offset is sign extended
- ALU calculates target memory address
- If SW, data is written to the memory

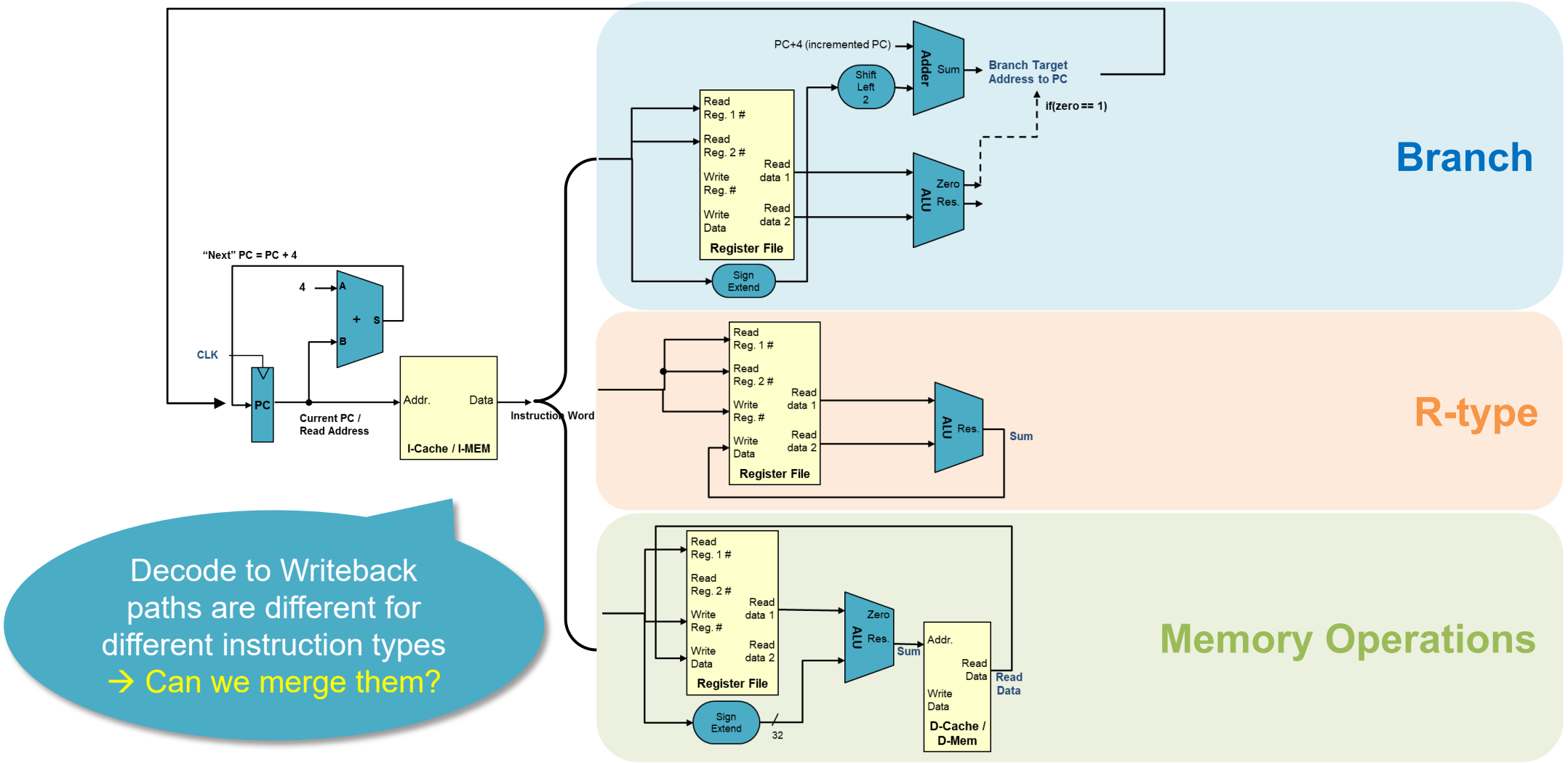


Branch Datapath: BEQ

- ALU for comparison (examine 'zero' output)
- Sign extension unit for branch offset
- Adder to add PC and offset

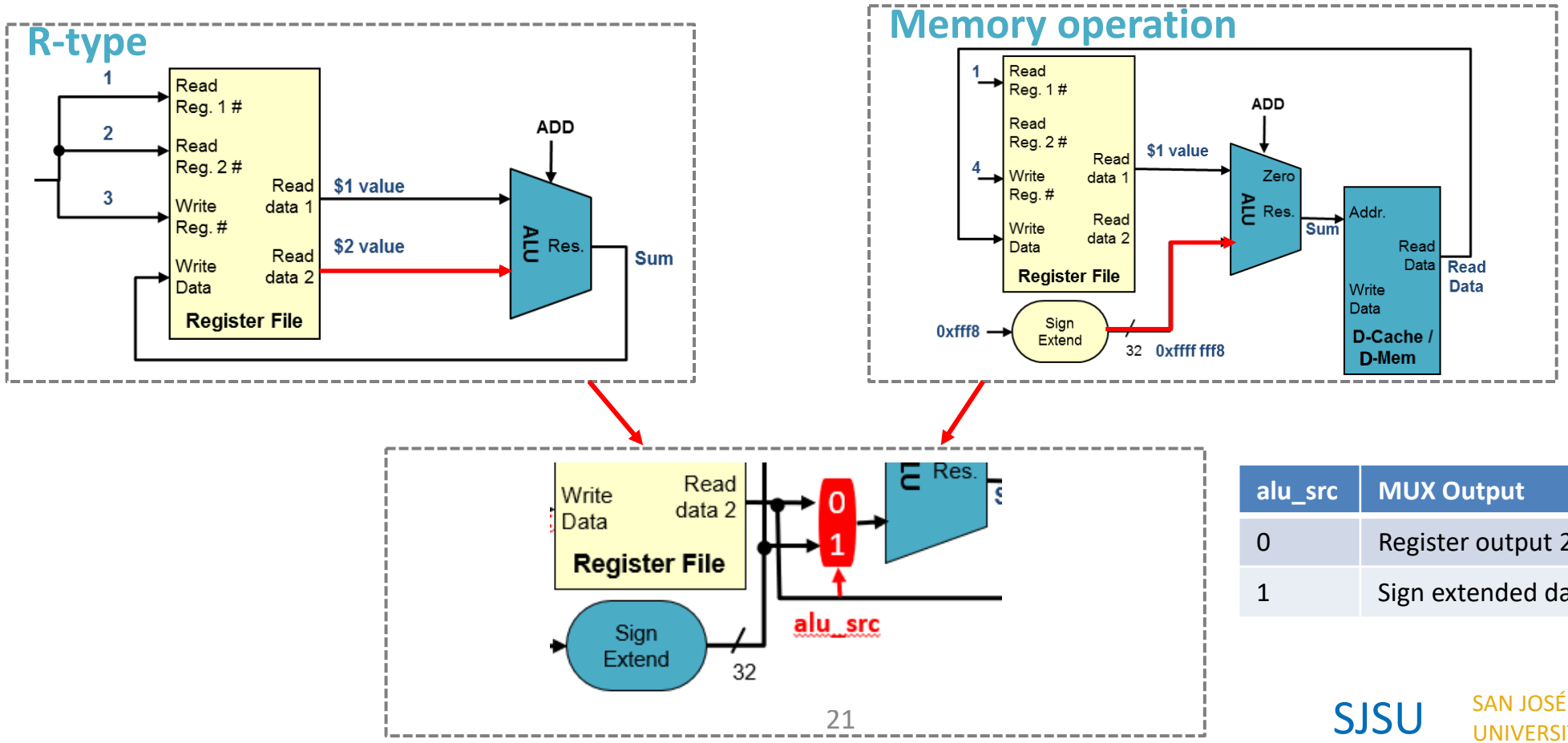


Single-Cycle Datapath



alu_src Mux

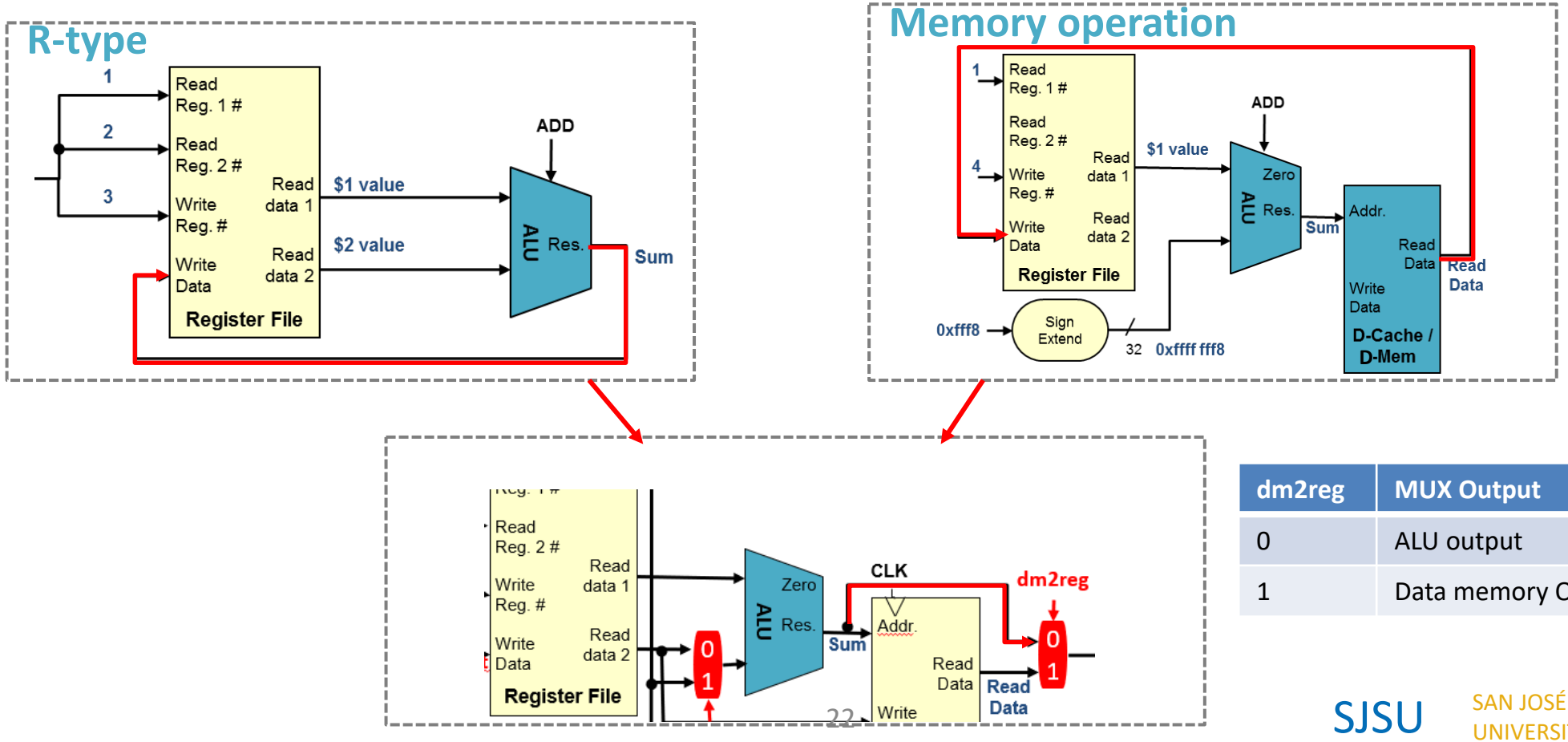
- **Mux controlling second input to ALU**
  - ALU instruction provides Read Register 2 data to the 2nd input of ALU
  - LW/SW uses 2nd input of ALU as an offset to form effective address



alu_src	MUX Output
0	Register output 2
1	Sign extended data

dm2reg Mux

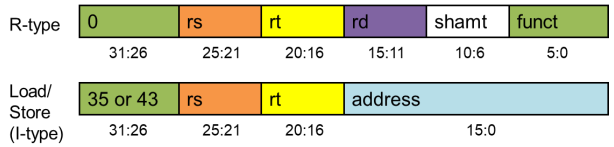
- **Mux controlling writeback value to register file**
  - ALU instructions use the result of the ALU
  - LW uses the read data from data memory

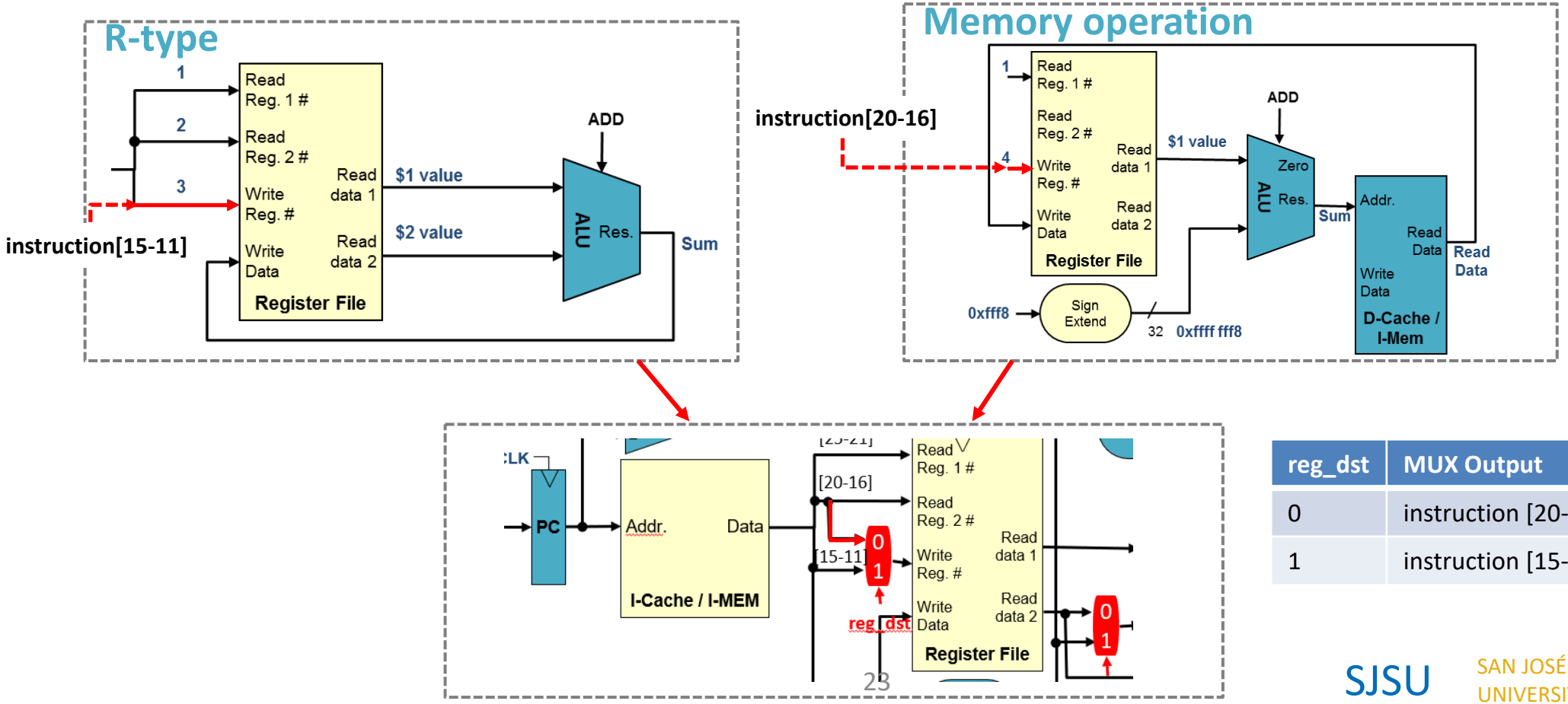


dm2reg	MUX Output
0	ALU output
1	Data memory Output

reg_dst Mux

- Different destination register ID fields for ALU and LW instructions

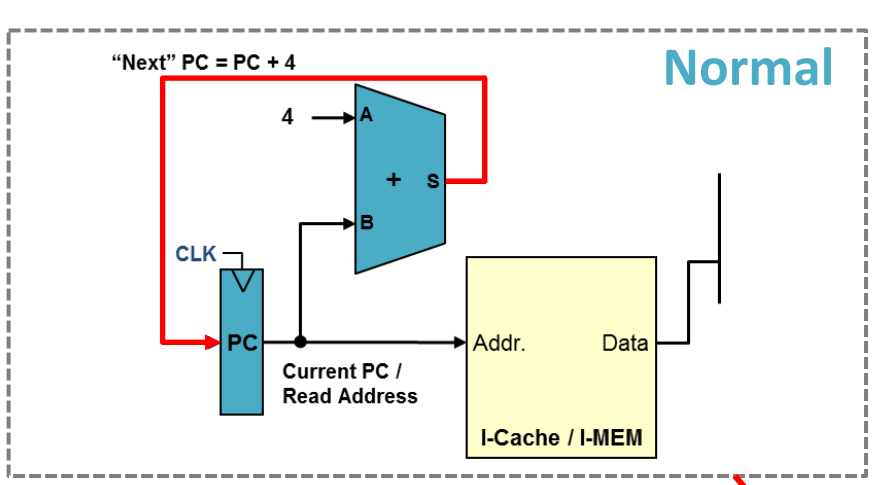


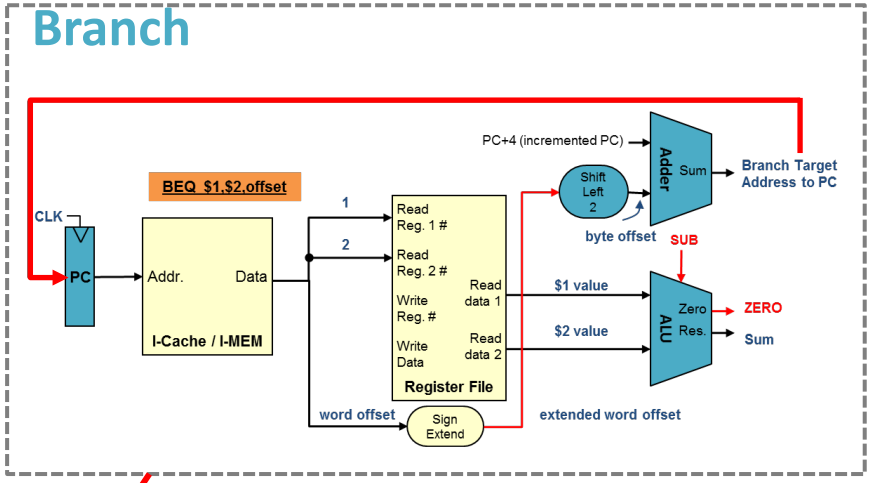


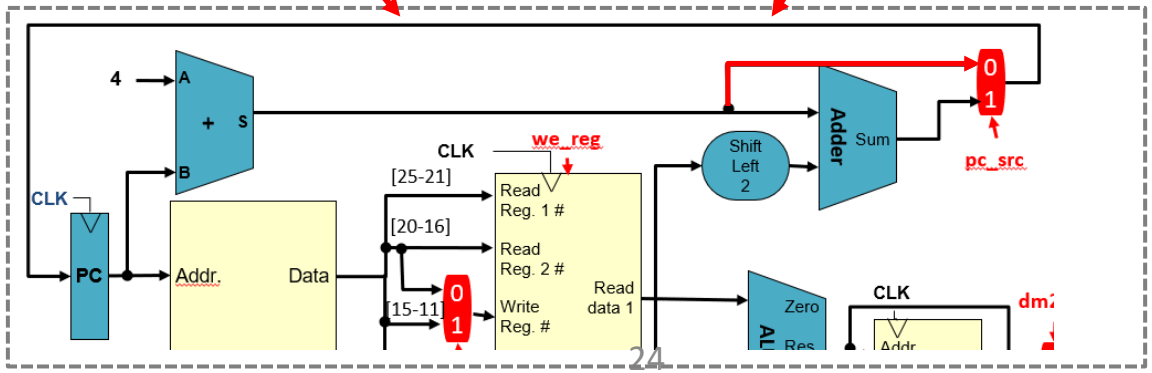
reg_dst	MUX Output
0	instruction [20-16]
1	instruction [15-11]

pc_src Mux

- **Next instruction can either be at the next sequential address (PC+4) or the branch target address (PC+offset)**

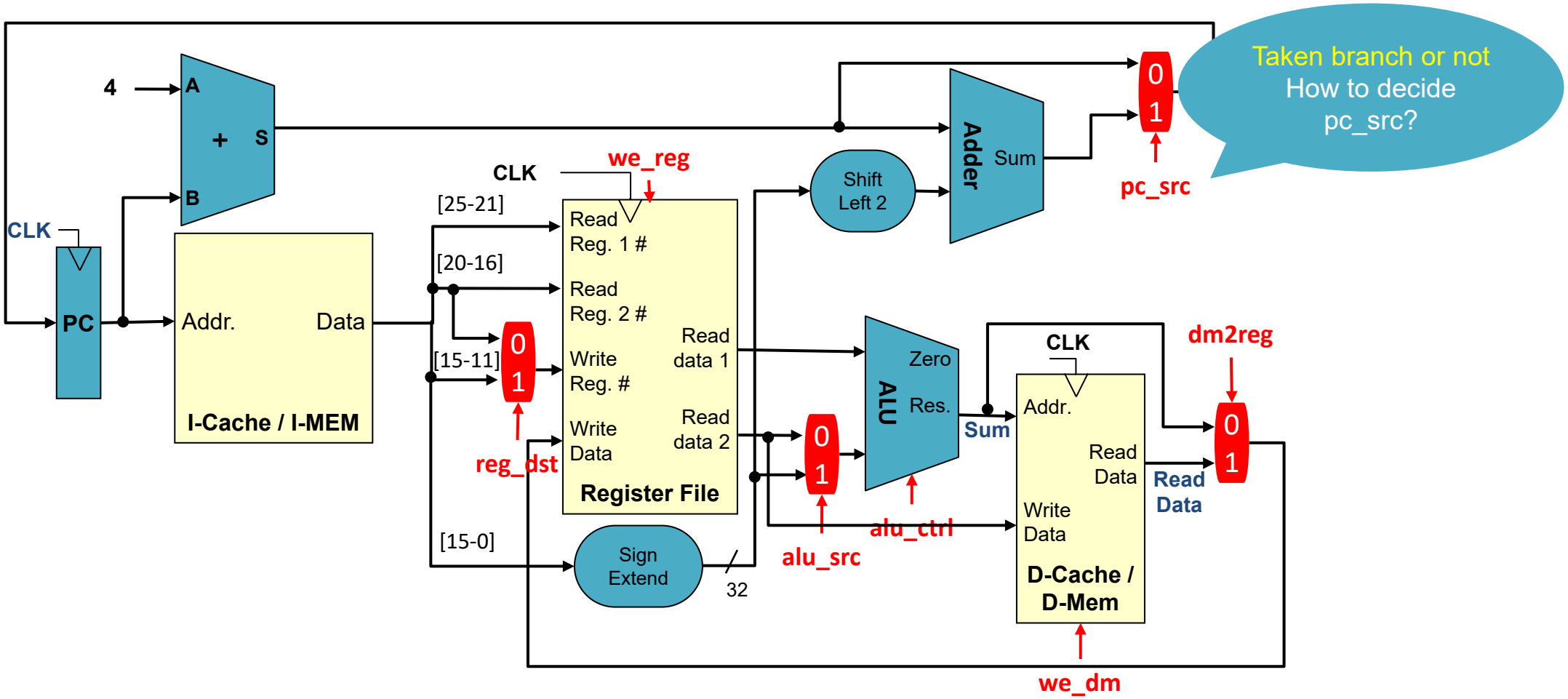




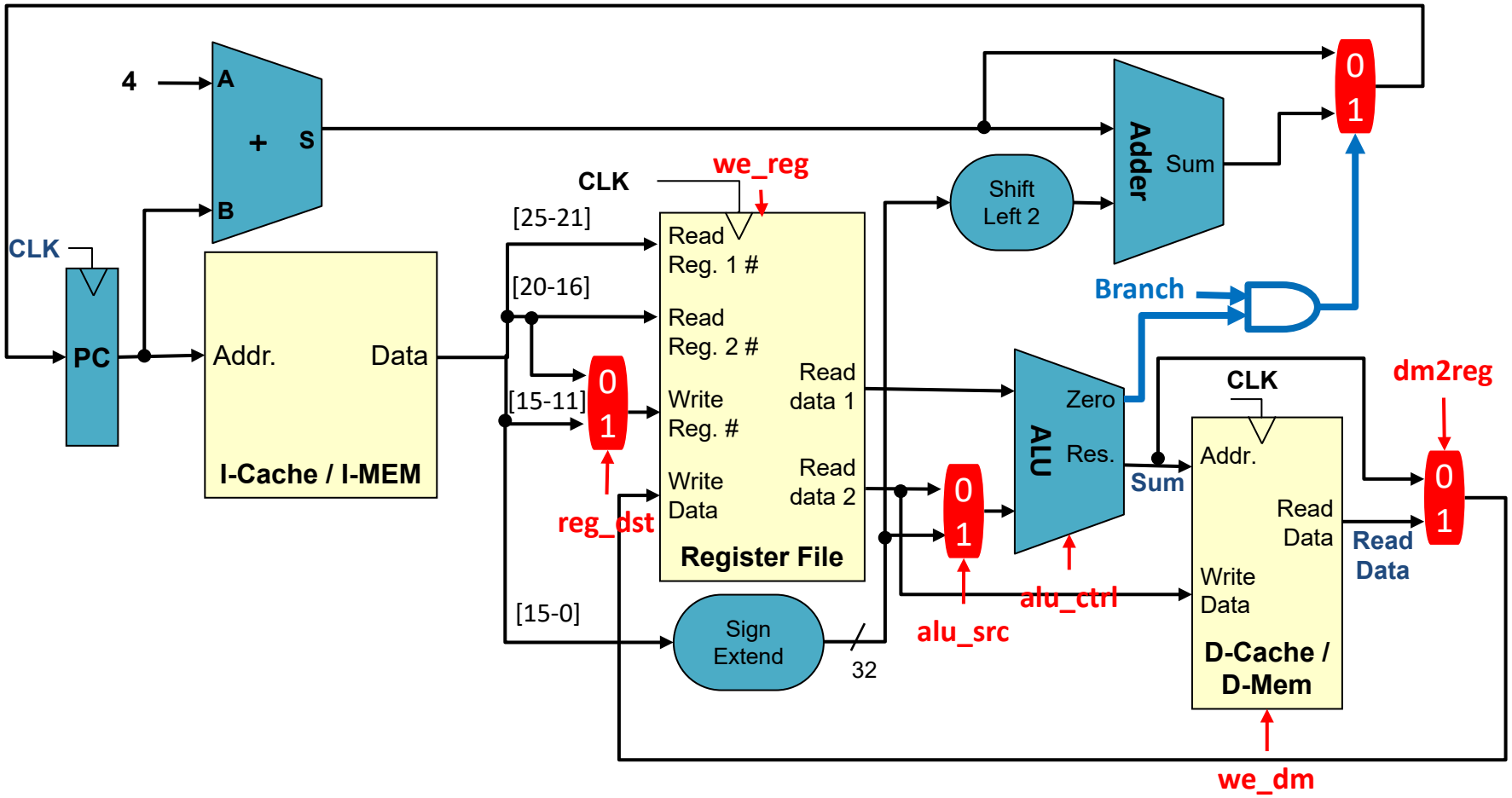


pc_src	MUX Output
0	PC+4
1	Branch target address

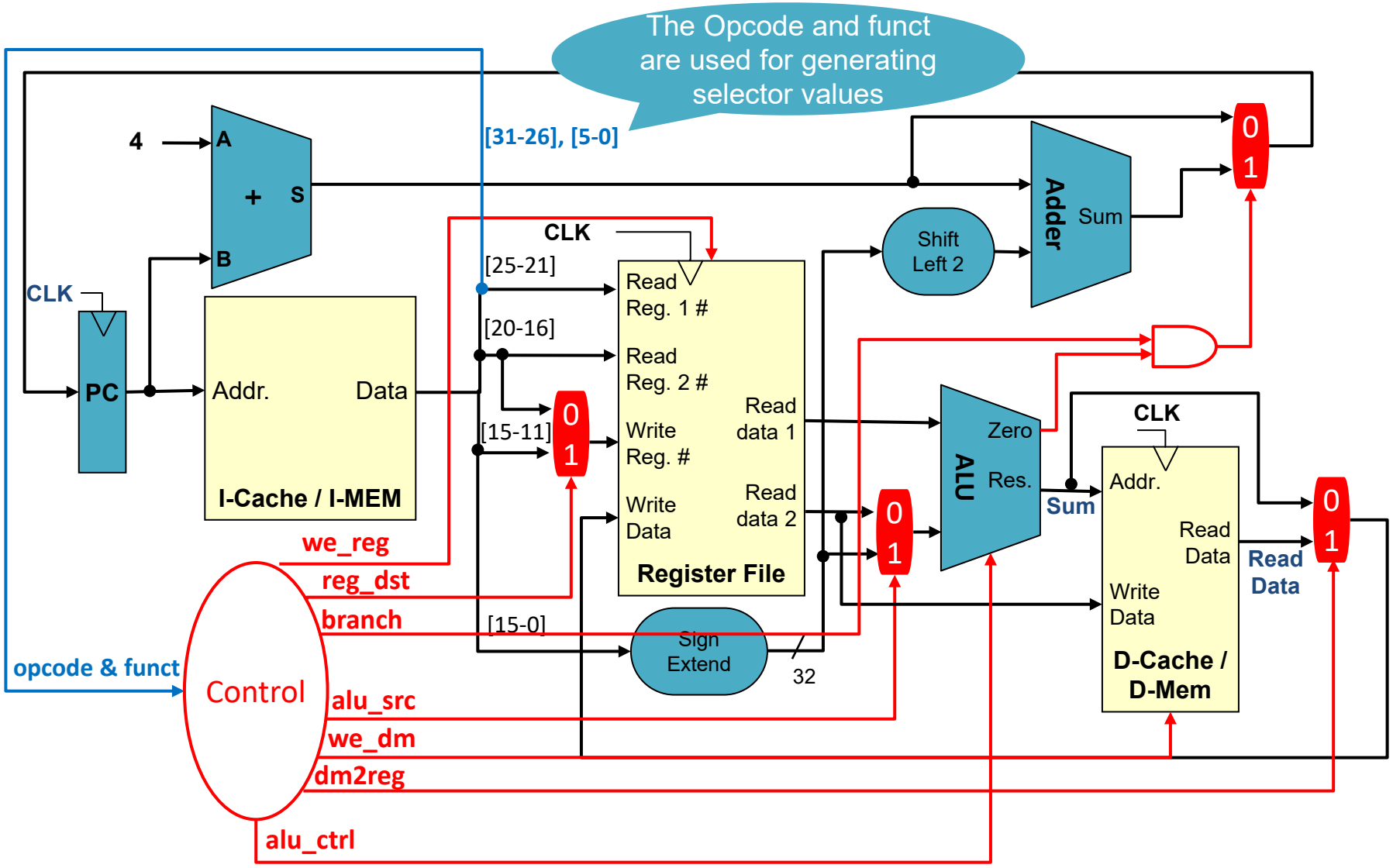
Single-cycle CPU Datapath



Single-cycle CPU Datapath

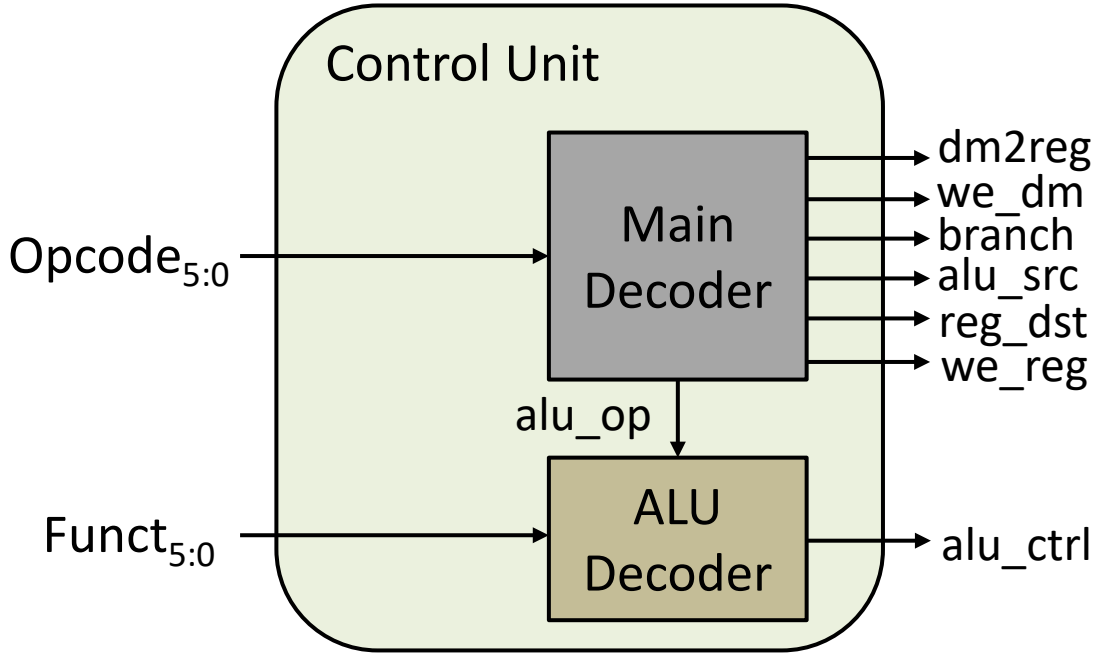


Single-cycle CPU Datapath



Single-cycle CPU Control Unit

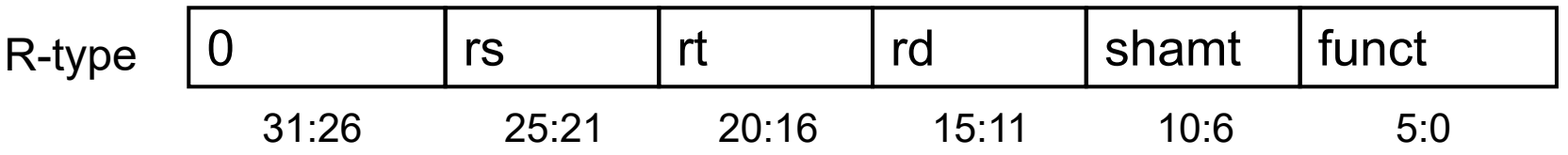
- **The control unit is a combinational decoder for the single-cycle CPU**
- **The CU contains**
  - A main decoder for Opcode
  - An ALU decoder for Funct



ALU Usage

- **Load/Store:**
  - Memory address = Base address + offset → add
- **Branch:**
  - Branch if two values are equal/not equal → subtract
- **R-type:**
  - Opcode & Funct field together indicates an ALU operation

These three instruction types can be distinguished by Opcode



Single-cycle CPU Control Unit

- **Main Decoder is used to generate instruction type indicator & mux control**
  - To cover 3 types of instructions, we need a **2-bit indicator, alu_op**
  - alu_op will be the input of ALU Decoder

- **For R-type instructions, the ALU Decoder will also check the Funct field**

	Input	Output						
Instruction	Opcode	we_reg	reg_dst	alu_src	branch	we_dm	dm2reg	$alu\_op_{1:0}$
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

ALU Decoder

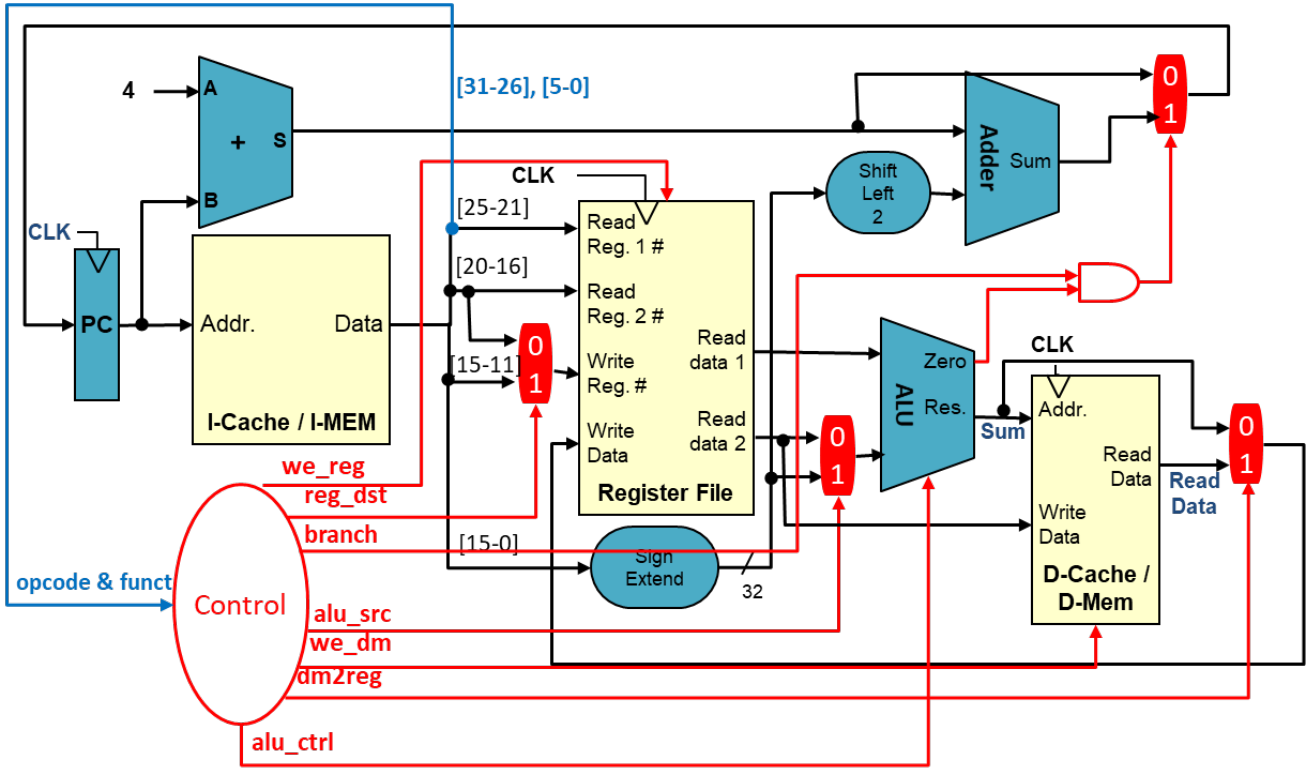
Input		Input			Output
opcode	**$alu\_op_{1:0}$**	**Operation**	**funct**	**ALU function**	**$alu\_ctrl_{2:0}$**
lw	00	load word	XXXXXX	ADD	010
sw	00	store word	XXXXXX	ADD	010
beq	01	branch equal	XXXXXX	SUB	110
R-type	10	ADD	100000	ADD	010
		SUB	100010	SUB	110
		AND	100100	AND	000
		OR	100101	OR	001
		SLT	101010	SLT	111

So Far…

- **Our design supports the following instructions**
  - Memory Reference Instructions:
    - **LW, SW**
  - Arithmetic and Logic Instructions:
    - **ADD, SUB, AND, OR, SLT**
  - Branch and Jump Instructions:
    - **BEQ**

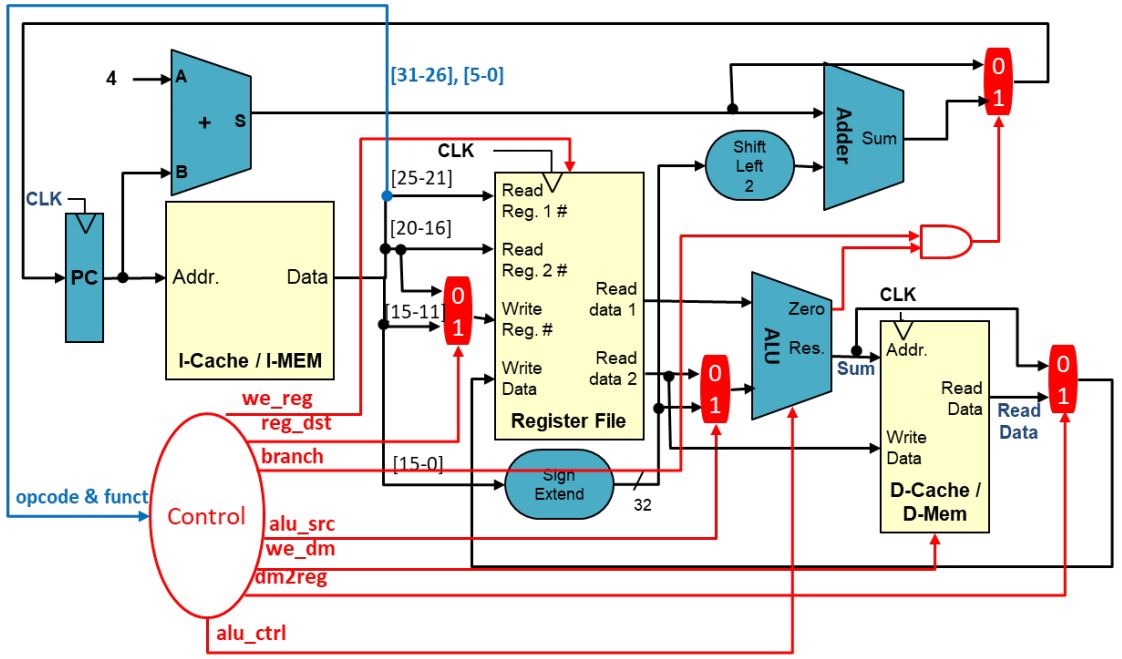
- **How to extend the design to support more MIPS instructions?**

Extension to Support addi



- No need to extend the current datapath
- Need to extend the control unit, specifically only the **Main Decoder**

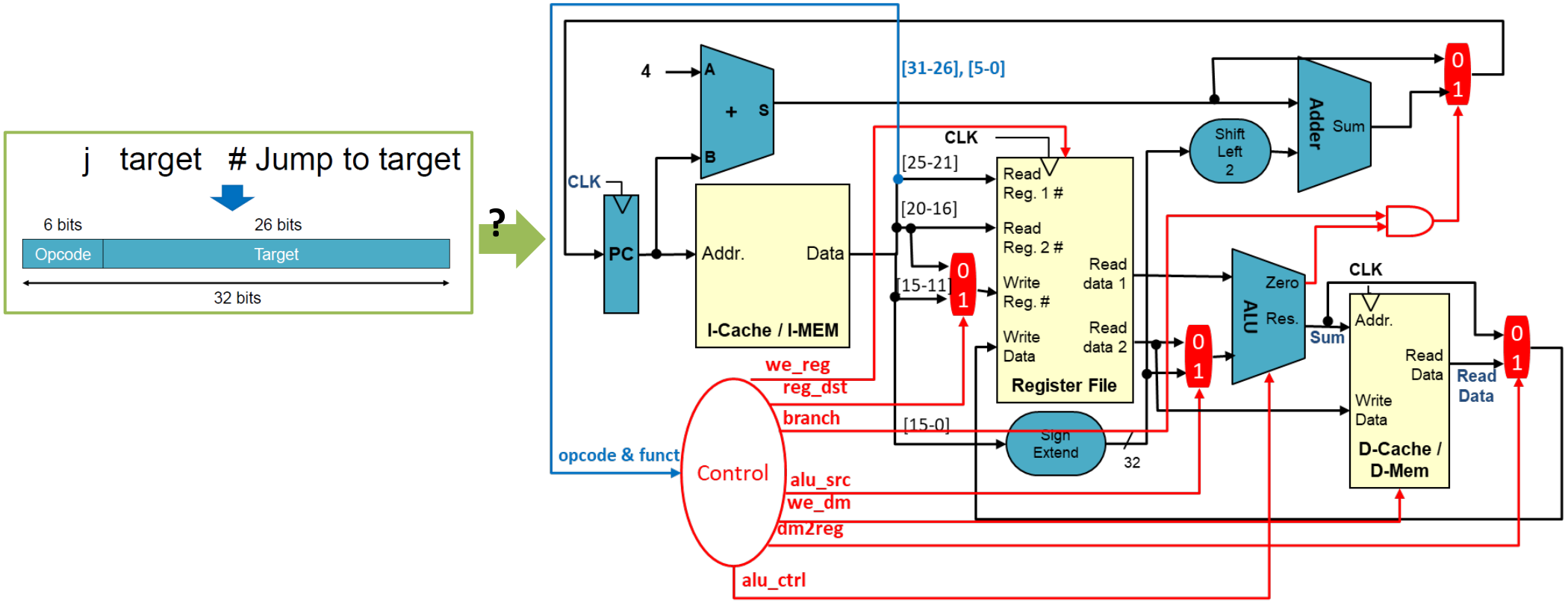
Main Decoder for addi



$alu\_op_{1:0}$	Main Decoder's Message
00	Do "ADD" operation for lw and sw
01	Do "SUB" operation for beq
10	Check Funct field for R-type
11	Not used

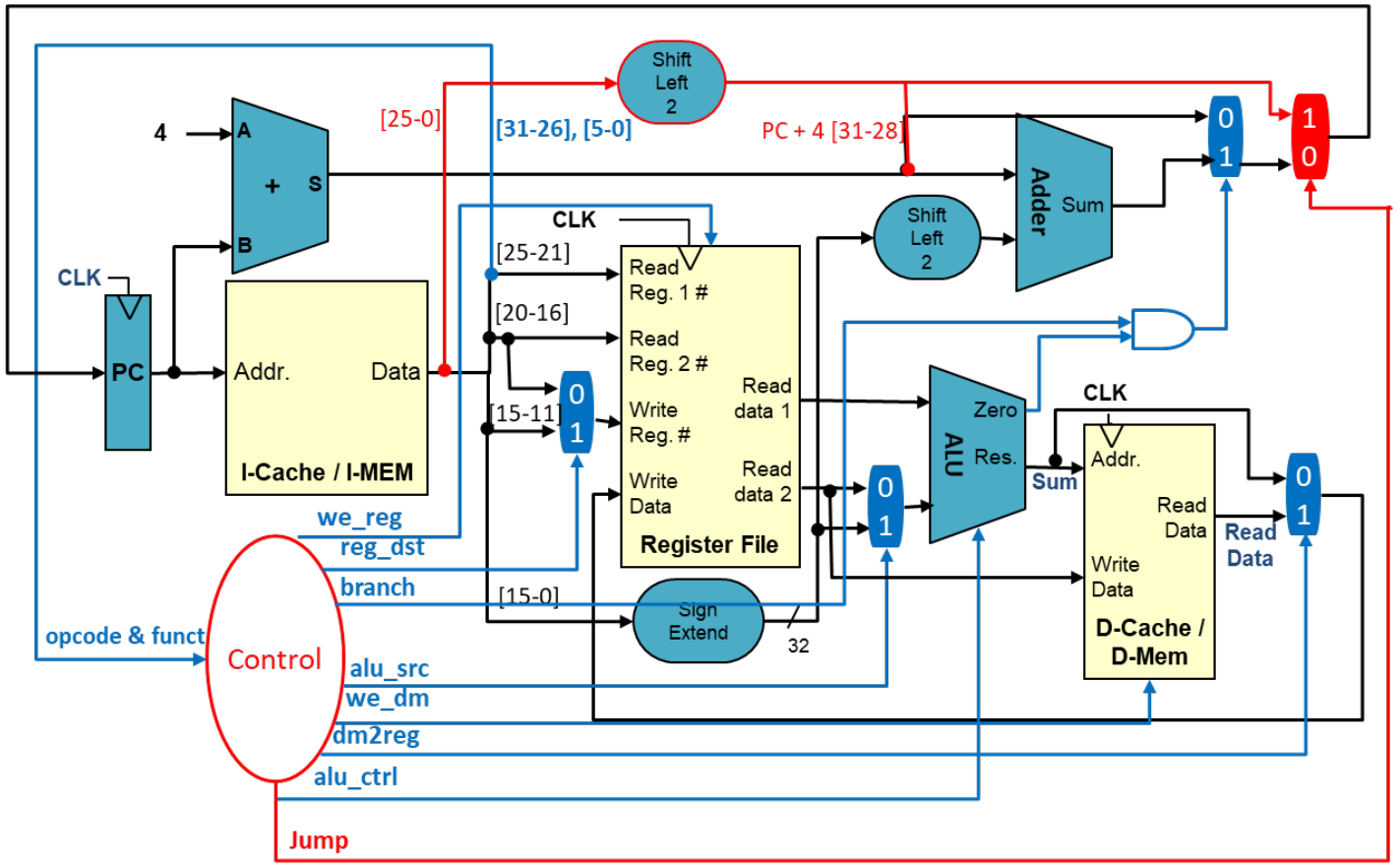
Instruction	Opcode	we_reg	reg_dst	alu_src	branch	we_dm	dm2reg	$alu\_op_{1:0}$
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	**001000**	**1**	**0**	**1**	**0**	**0**	**0**	**00**

Extension to Support `j`



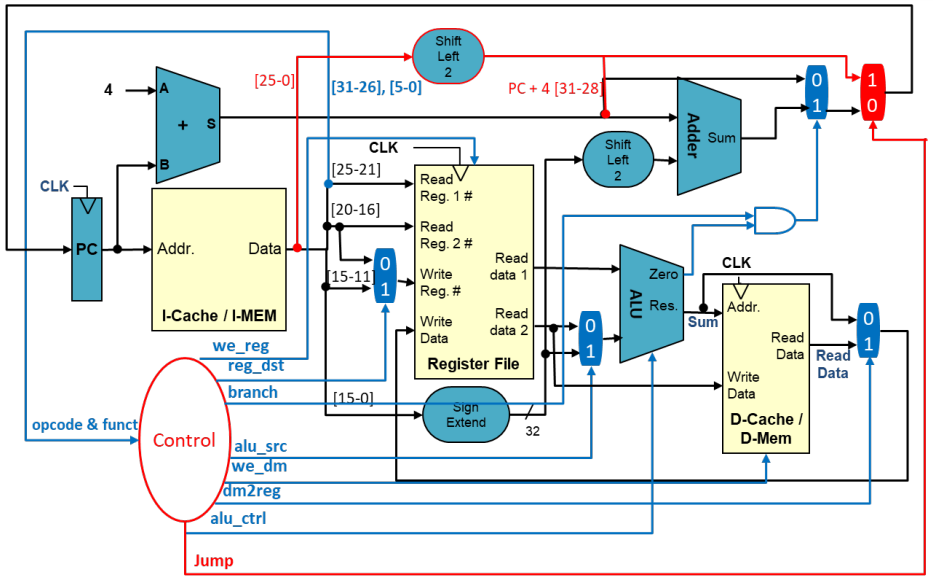
- Can we support `j` with this datapath?

Extension to Support `j`



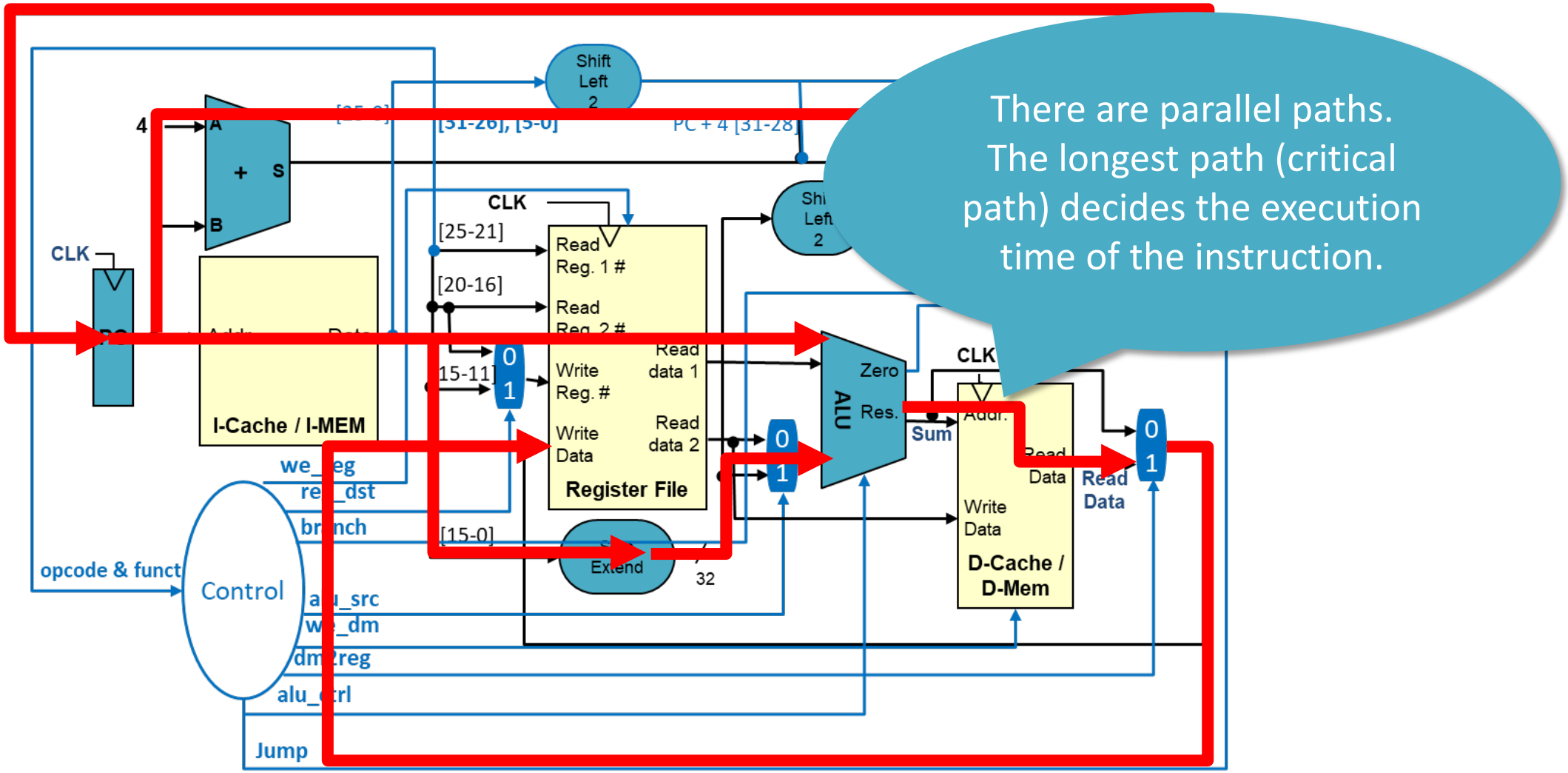
- **Both datapath and control unit (main decoder) need to be extended**

Main Decoder for `j`



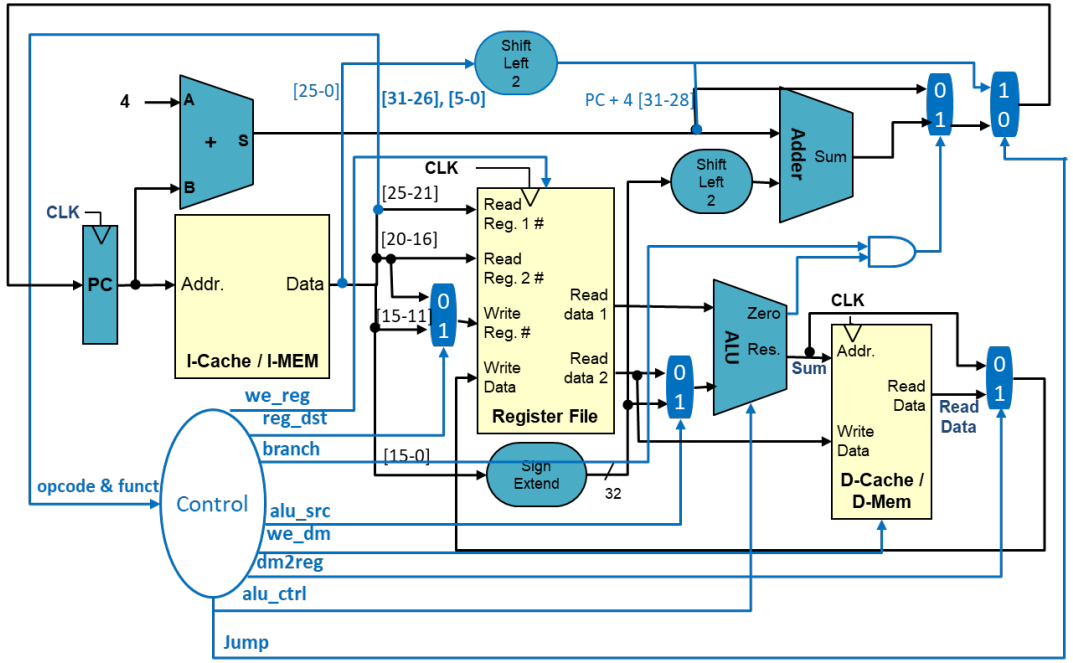
Instruction	Opcode	we_reg	reg_dst	alu_src	branch	we_dm	dm2reg	$alu\_op_{1:0}$	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	**000010**	**0**	**X**	**X**	**X**	**0**	**X**	**XX**	**1**

Datapath of `LW` instruction



Single-Cycle CPU Performance Analysis

Function Unit	Parameter	Delay (ps)
Register clock-to-Q	T_{pcq_PC}	30
MUX	T_{MUX}	25
Sign-extend	T_{s_ext}	25
ALU	T_{ALU}	200
Mem read	T_{mem}	250
Register file read	T_{RFread}	150
Register file write	$T_{RFwrite}$	20



Critical path for `LW` =

= (30 + 250 + 150 + 200 + 250 + 25 + 20) ps

= 925 ps

Single-Cycle CPU Performance Analysis

- **The longest instruction's execution time is 925 ps.**
- **If we run 100 billion instructions, what is the total execution time?**

The clock cycle period should be set to 925 ps

CPU time = # instructions x CPI x cycle period

= $100 \times 10^{9} \times 1 \times 925 \times 10^{-12}$

= 92.5 seconds

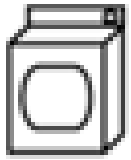
Performance Issues

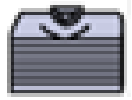
- **Longest delay determines clock period**
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- **Violates design principle**
  - Making the common case fast
  - More arithmetic operations than memory operations
- **Most modern CPUs use multi-cycle processors**
  - Each instruction takes multiple cycles
  - Multiple instructions can execute concurrently (parallelism → pipelining)

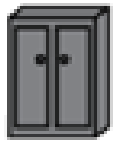
Pipelining Analogy

- **Laundry tasks**


Place one dirty load of clothes in the washer

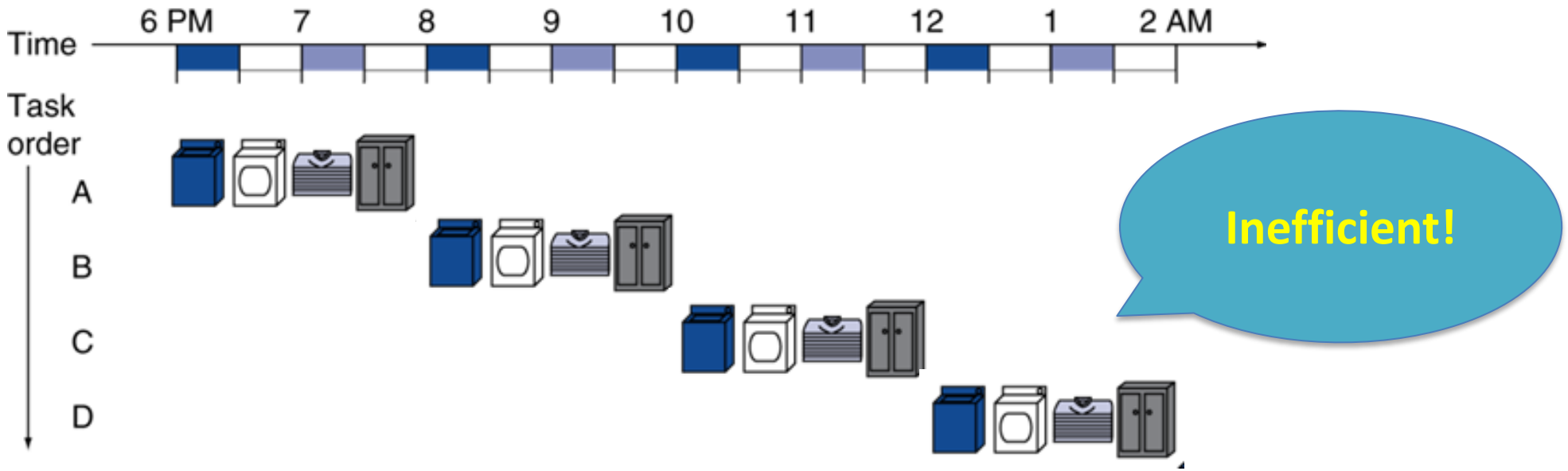

Place the wet load in the dryer


Place the dry load on a table and fold


Ask your roommate to put the clothes away

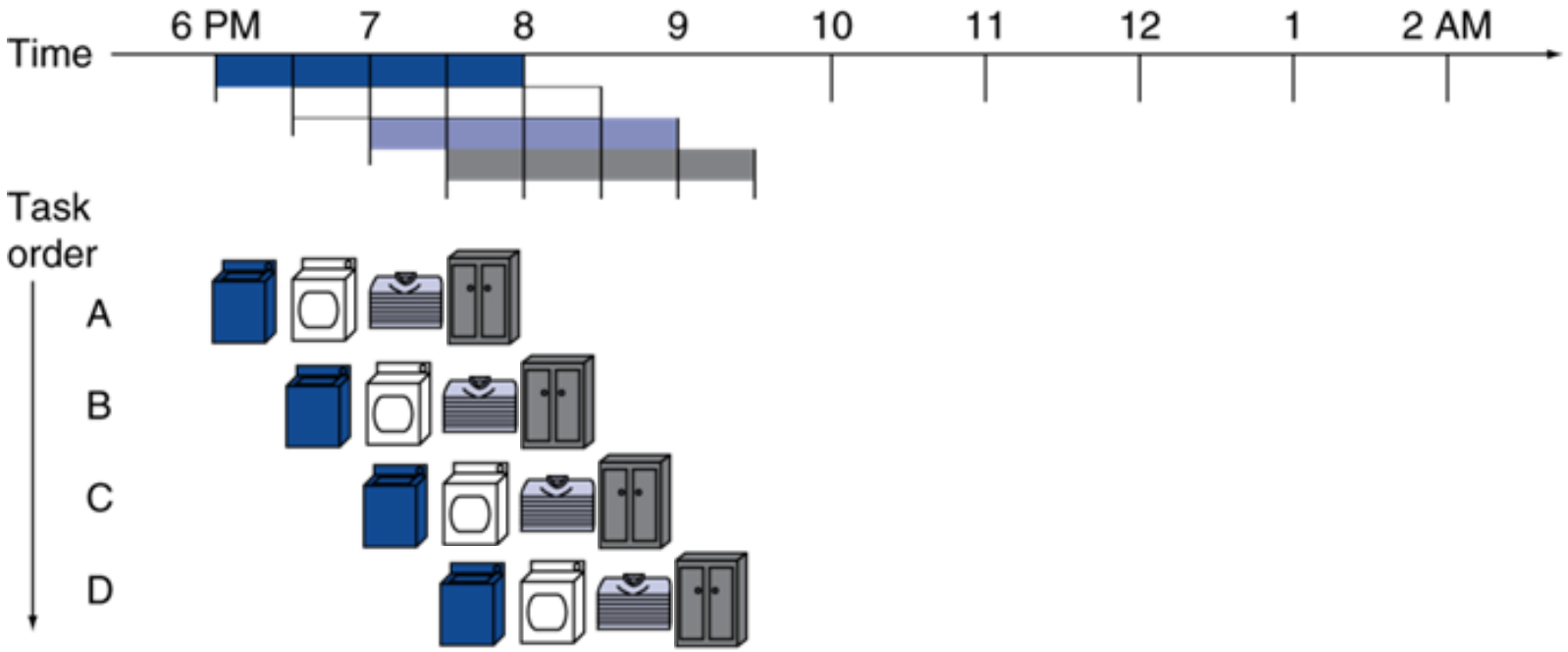
Pipelining Analogy

- **Lots of dirty clothes -- need to wash them over four separate loads**
- **Assume each task takes 30 minutes**



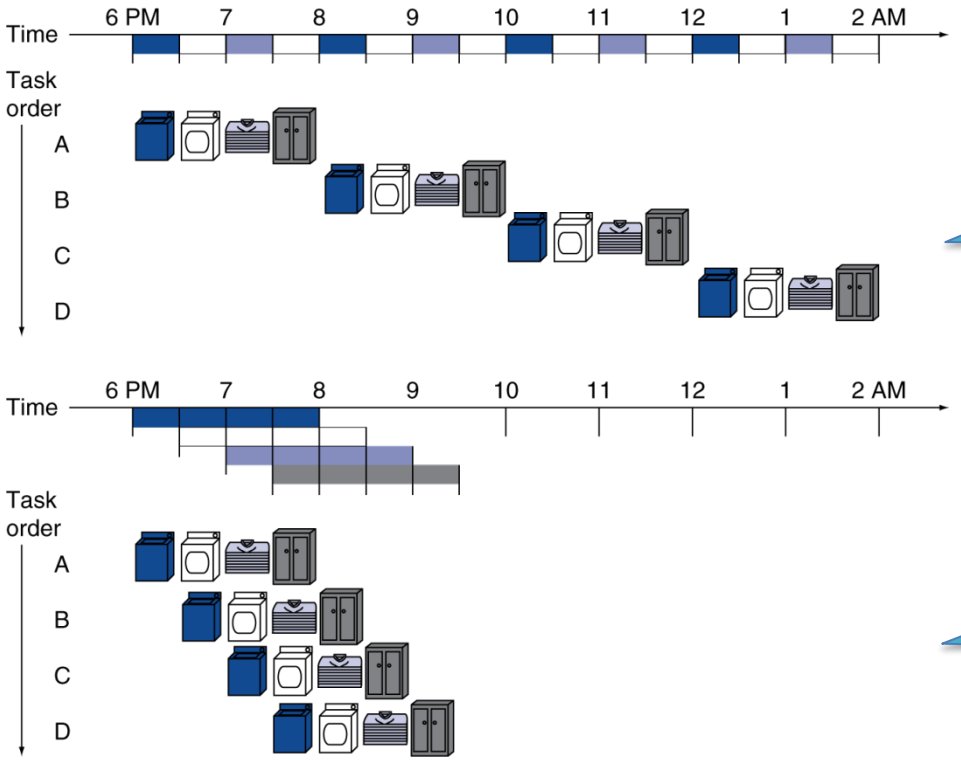
Pipelining Analogy

- **You can start the next load as soon as the washer finishes washing the previous load**



Pipelining Analogy

- **We can parallelize the single-cycle CPU tasks in a similar way**
- **Pipelined laundry → overlapping execution**

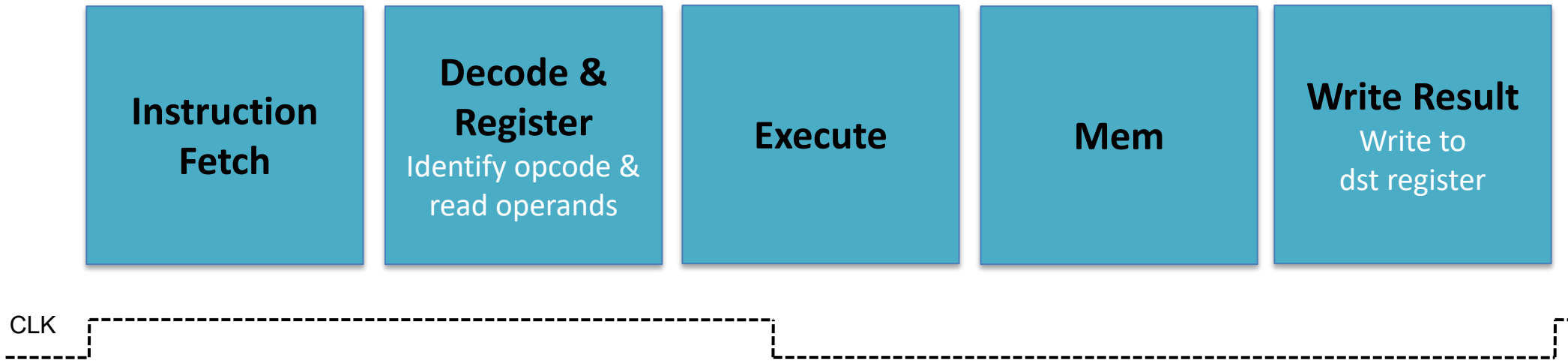


Single-cycle CPU
clock cycle period = 2hrs

Pipelined CPU
clock cycle period = 0.5hrs

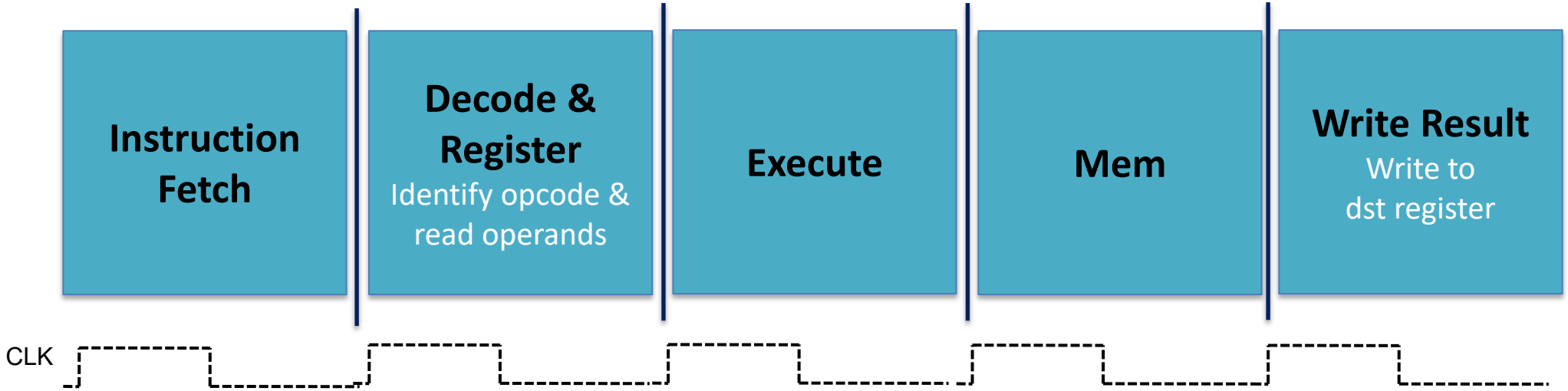
MIPS Pipeline

- **5 steps in one cycle for single-cycle CPU design**
  - With each step using separate resources



MIPS Pipeline

- **Entire datapath can be broken into five stages**
- **Cycle period is changed to the time taken for a stage**
  - Shorter clock cycle
  - Higher CPI (e.g., CPI = 5 in MIPS)



MIPS Pipeline

- **Entire datapath can be broken into five stages**
- **Cycle period is changed to the time taken for a step**
  - Shorter clock cycle
  - Higher CPI (e.g., CPI = 5 in MIPS)
  - System can achieve **CPI = 1** by overlapping Multiple Instructions

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11
Inst1	IF	ID	EXE	MEM	WB						
Inst2		IF	ID	EXE	MEM	WB					
Inst3			IF	ID	EXE	MEM	WB				
Inst4				IF	ID	EXE	MEM	WB			
Inst5					IF	ID	EXE	MEM	WB		

cc: clock cycle

Initial pipeline filling phase

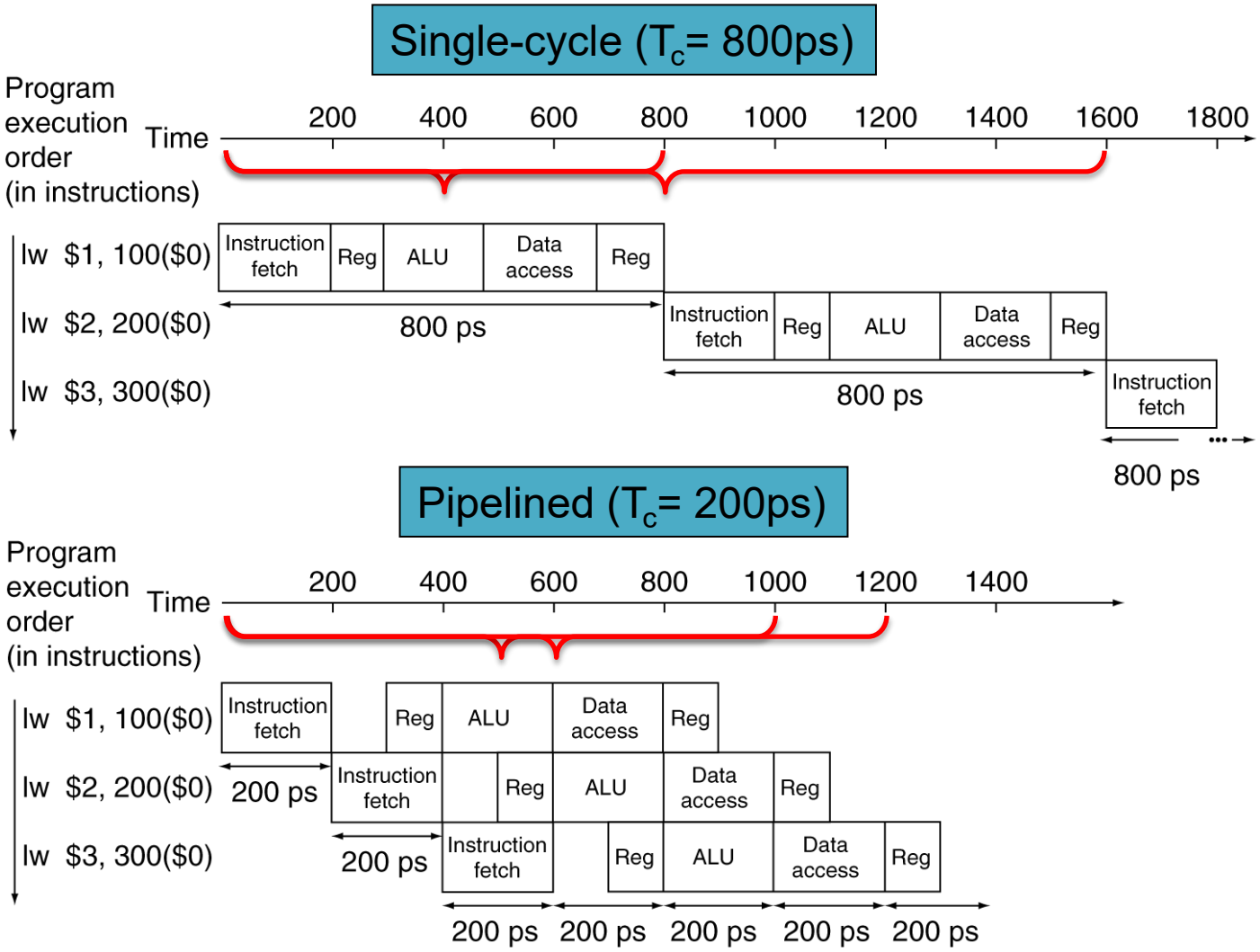
After initial pipeline filling phase, we can finish an instruction every cycle

Pipeline Performance Example

- **100ps for register IOs (i.e., decode, writeback); 200ps for other stages**
- **What would be the cycle period in single-cycle and pipelined datapath?**
- **What would be the execution time of lw instruction?**

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-type	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance Example



T_c = Longest instruction

T_{lw} = 800ps

2 x T_{lw} = 1600ps

...

T_c = Longest Stage

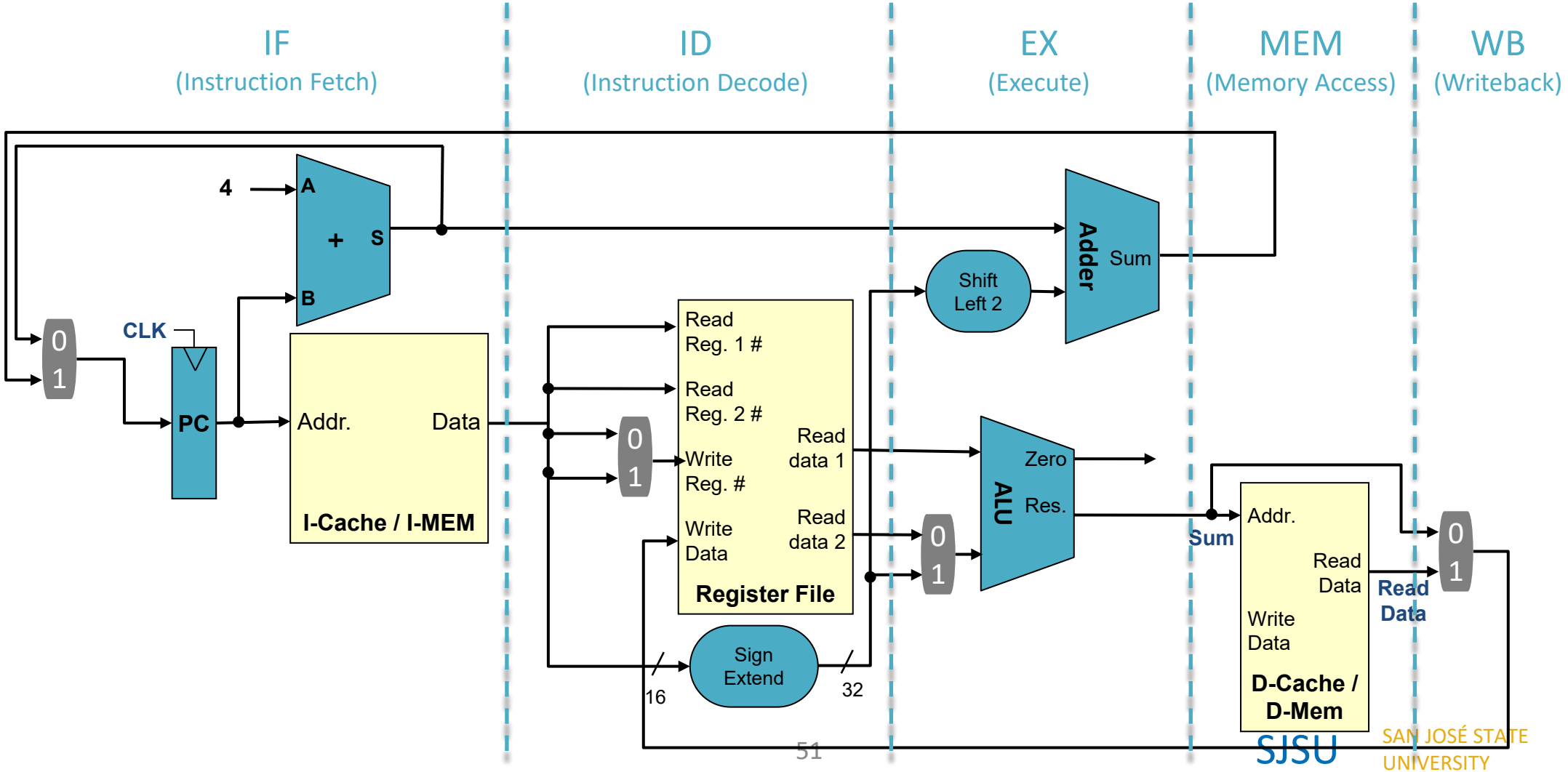
T_{lw} = 1000ps

2 x T_{lw} = 1200ps

...

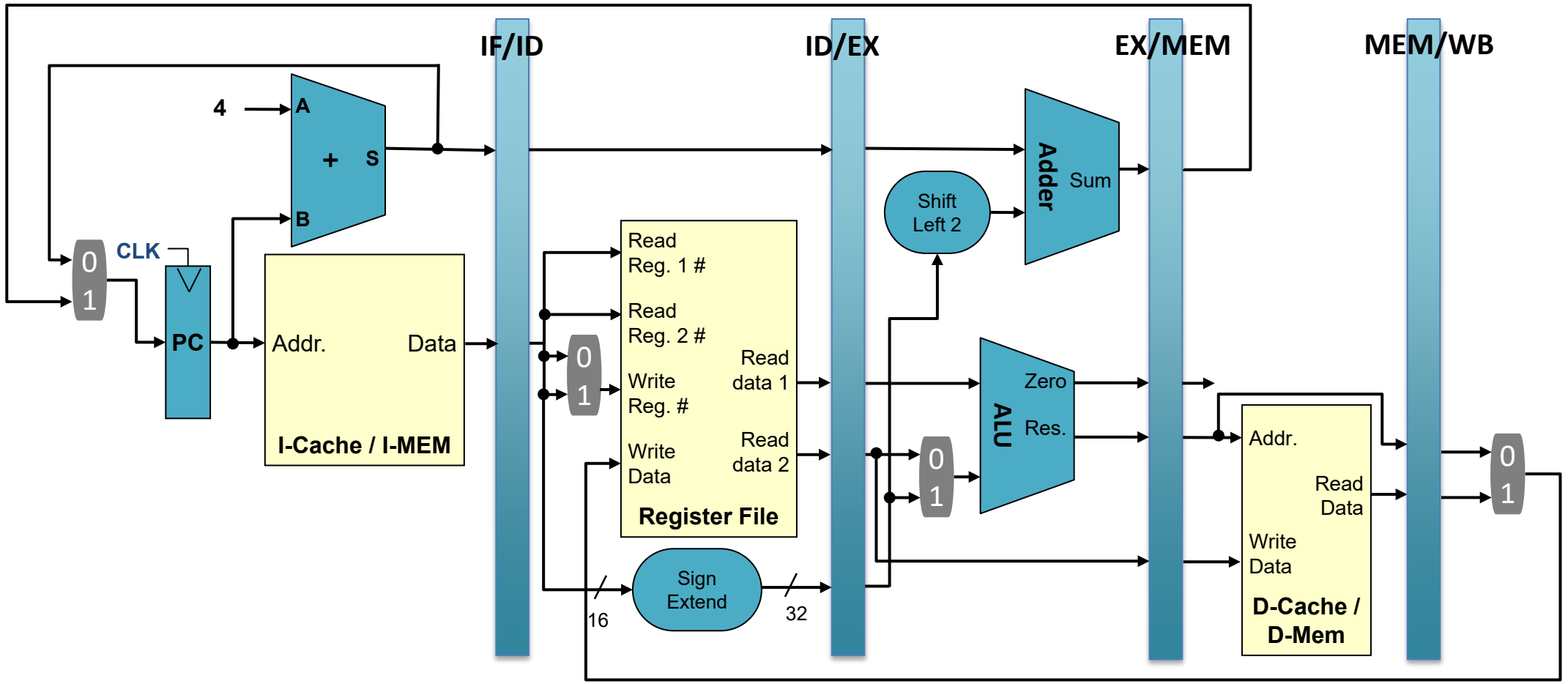
Basic 5 Stage Pipeline

- **Same structure as single cycle but now broken into 5 stages**

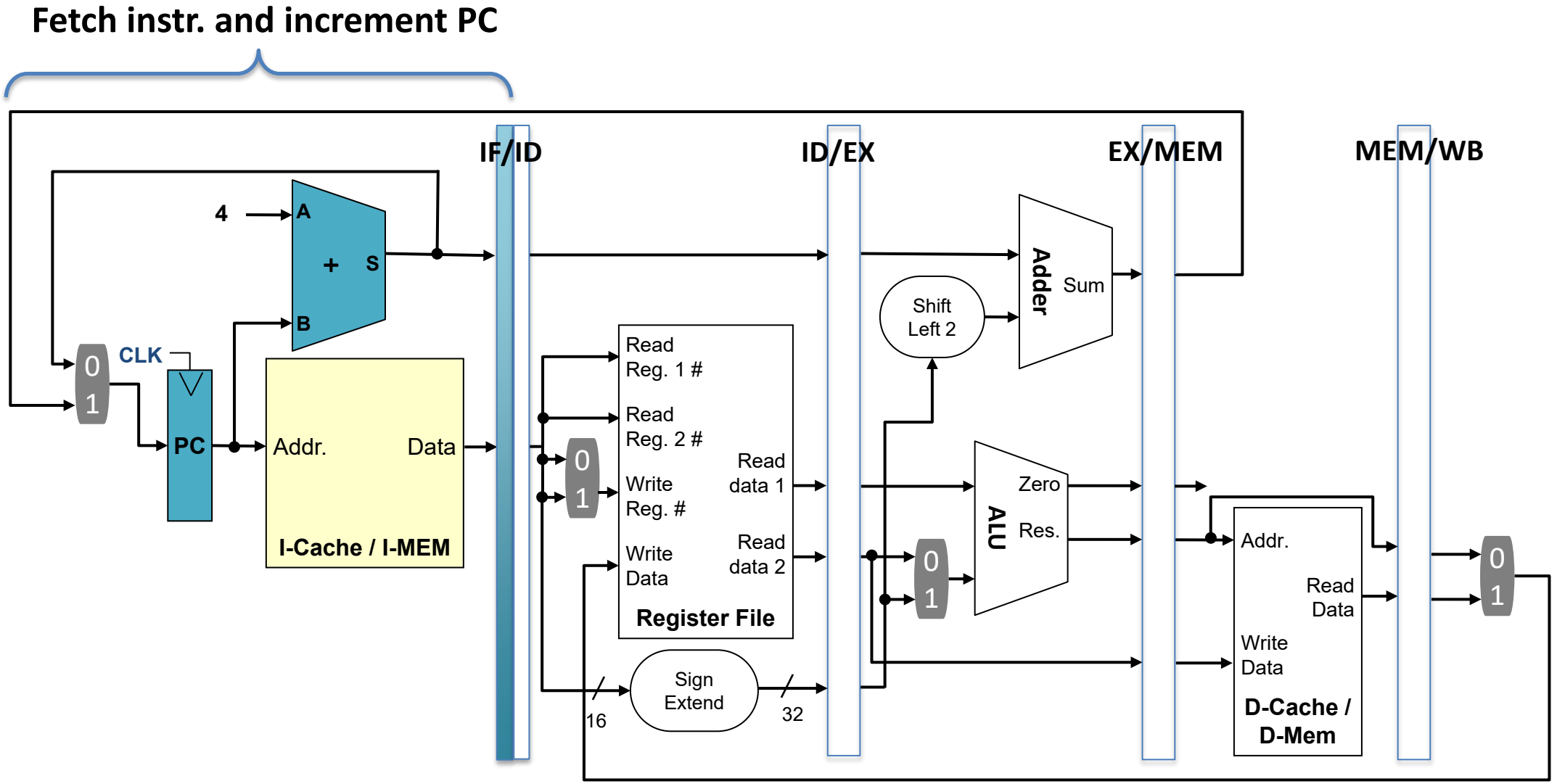


Pipeline Stage Registers

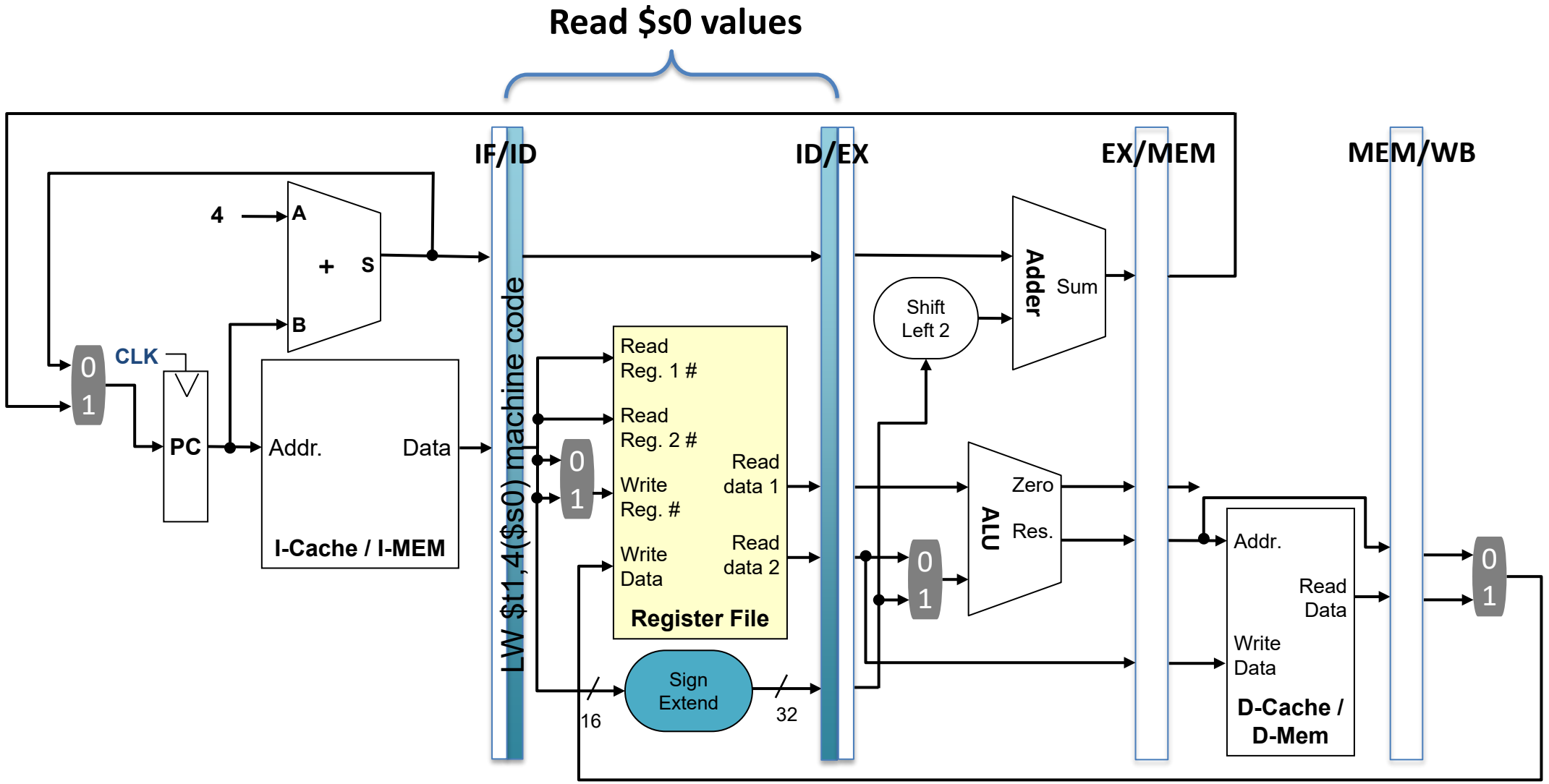
- **Intermediate results need to be stored to allow stage reuse**



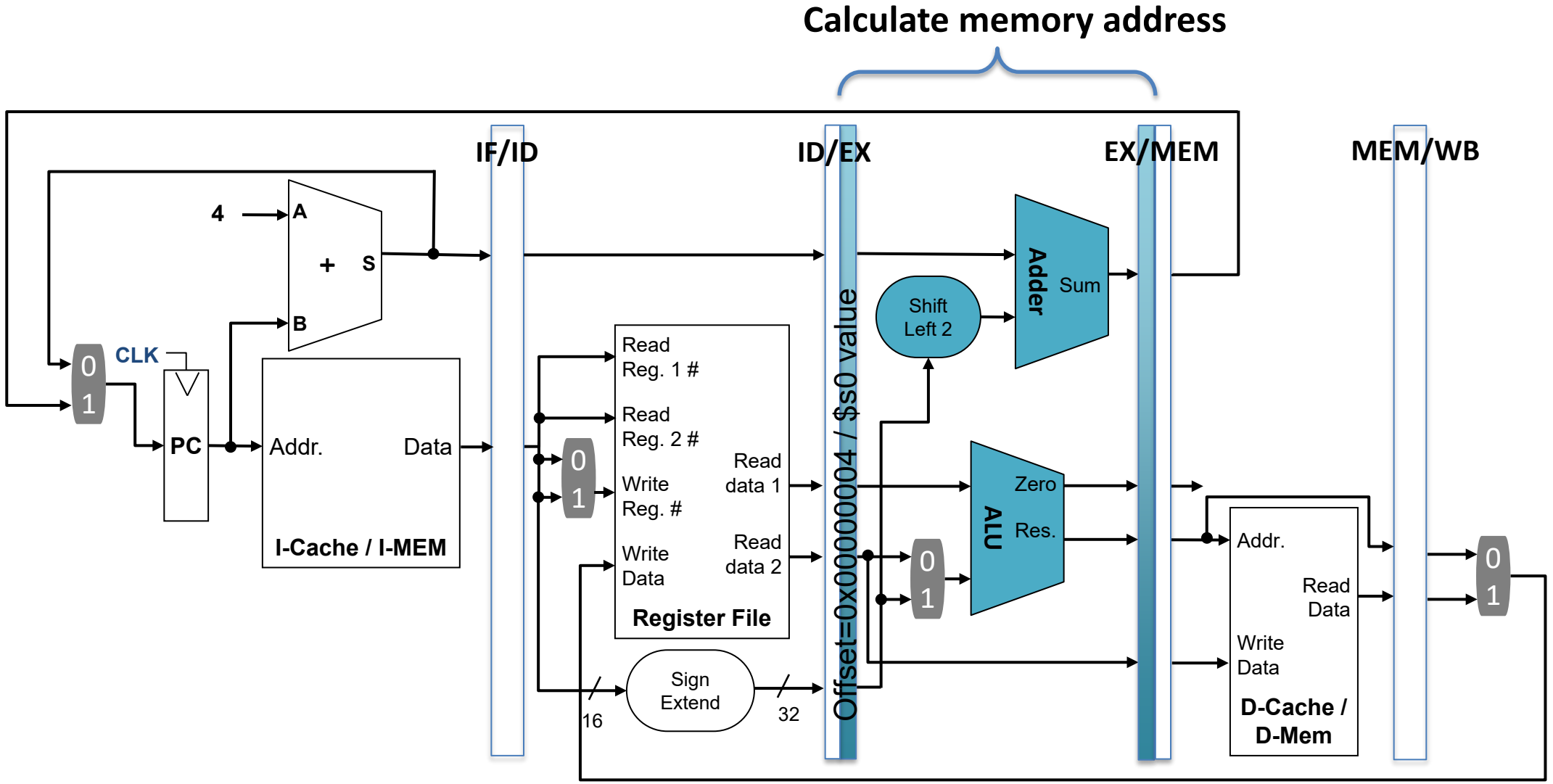
Pipeline Example: LW $t1, 4($s0)



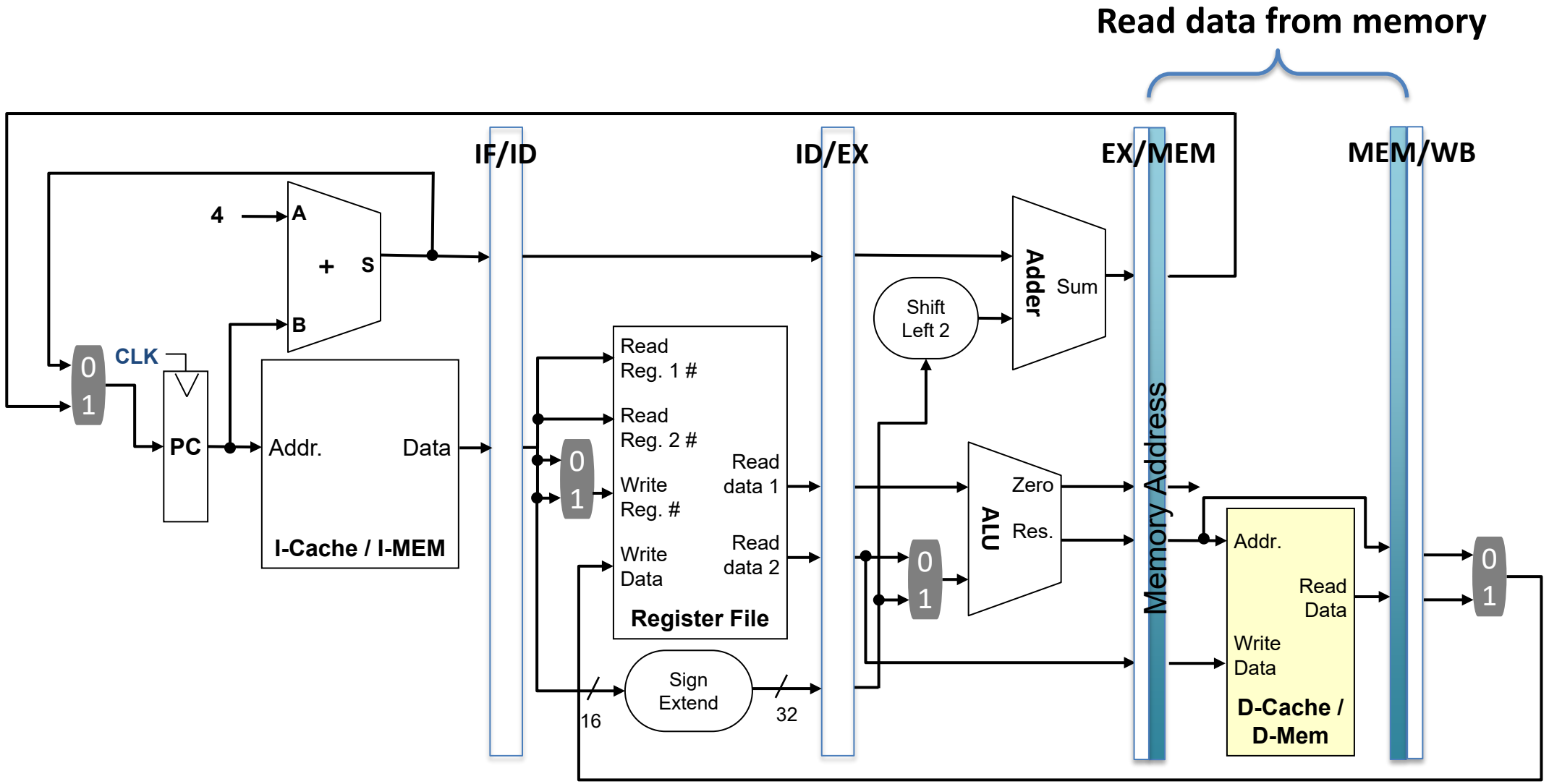
Pipeline Example: LW $t1, 4($s0)



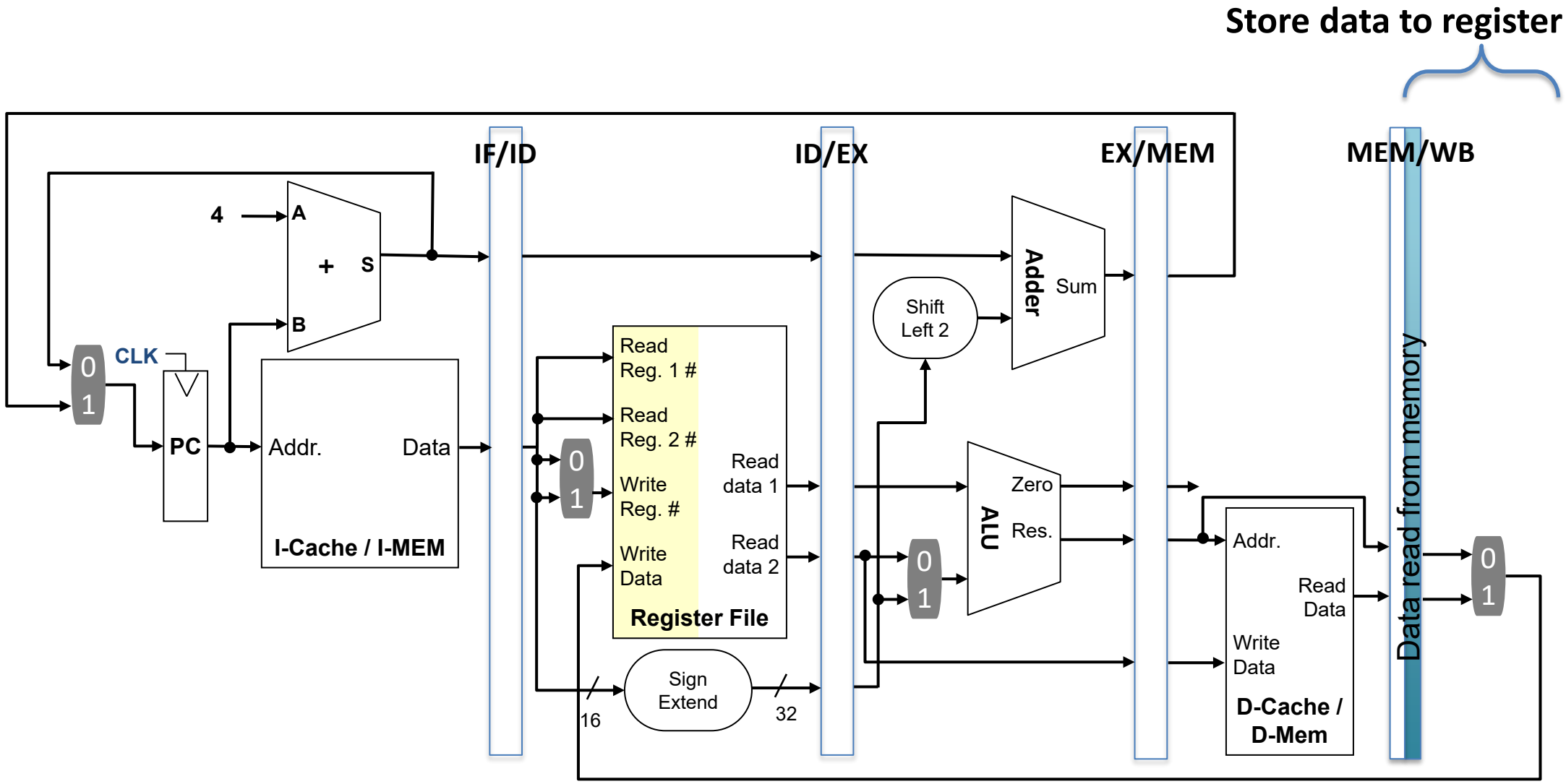
Pipeline Example: LW $t1, 4($s0)



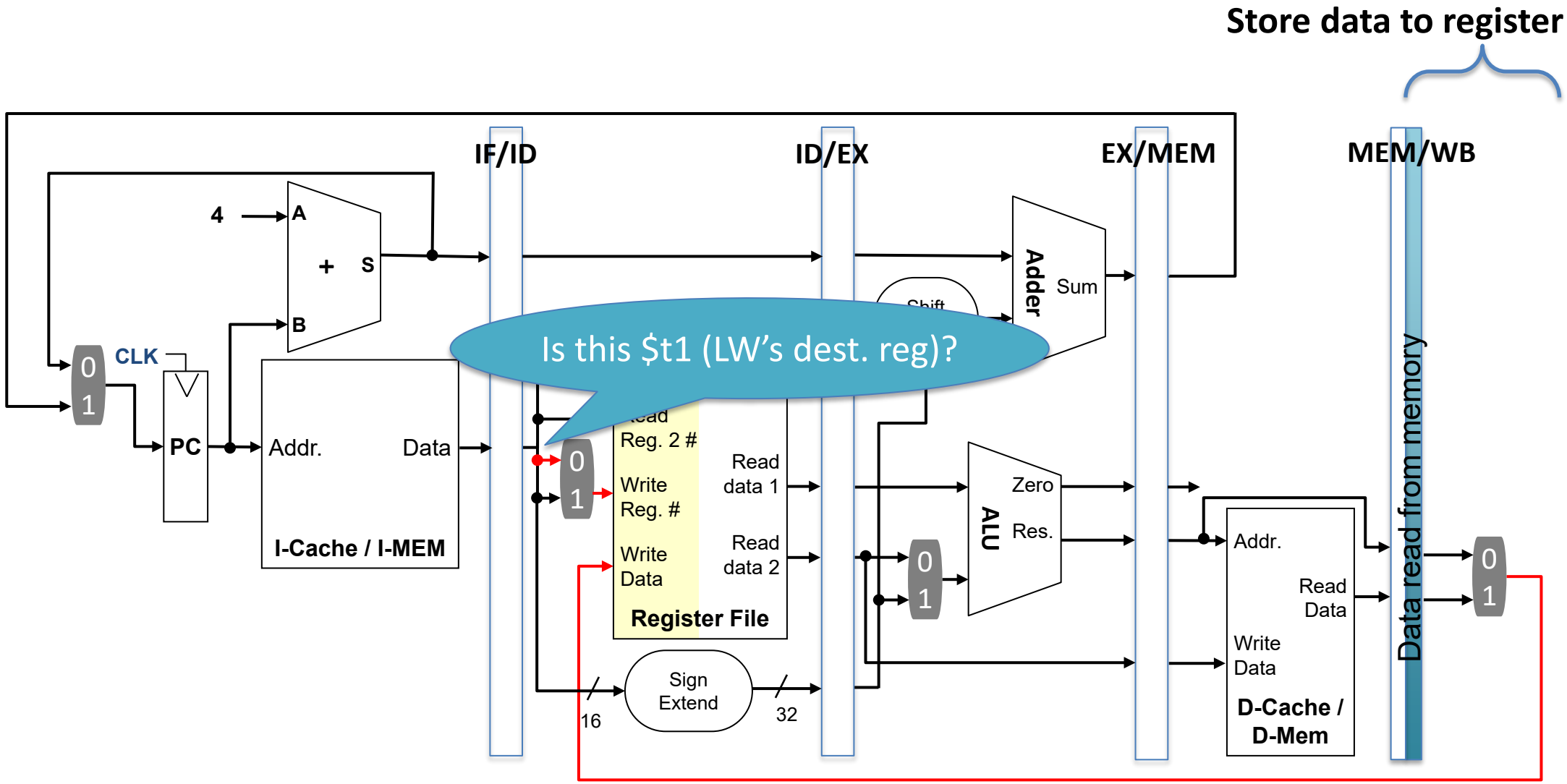
Pipeline Example: LW $t1, 4($s0)



Pipeline Example: LW $t1, 4($s0)

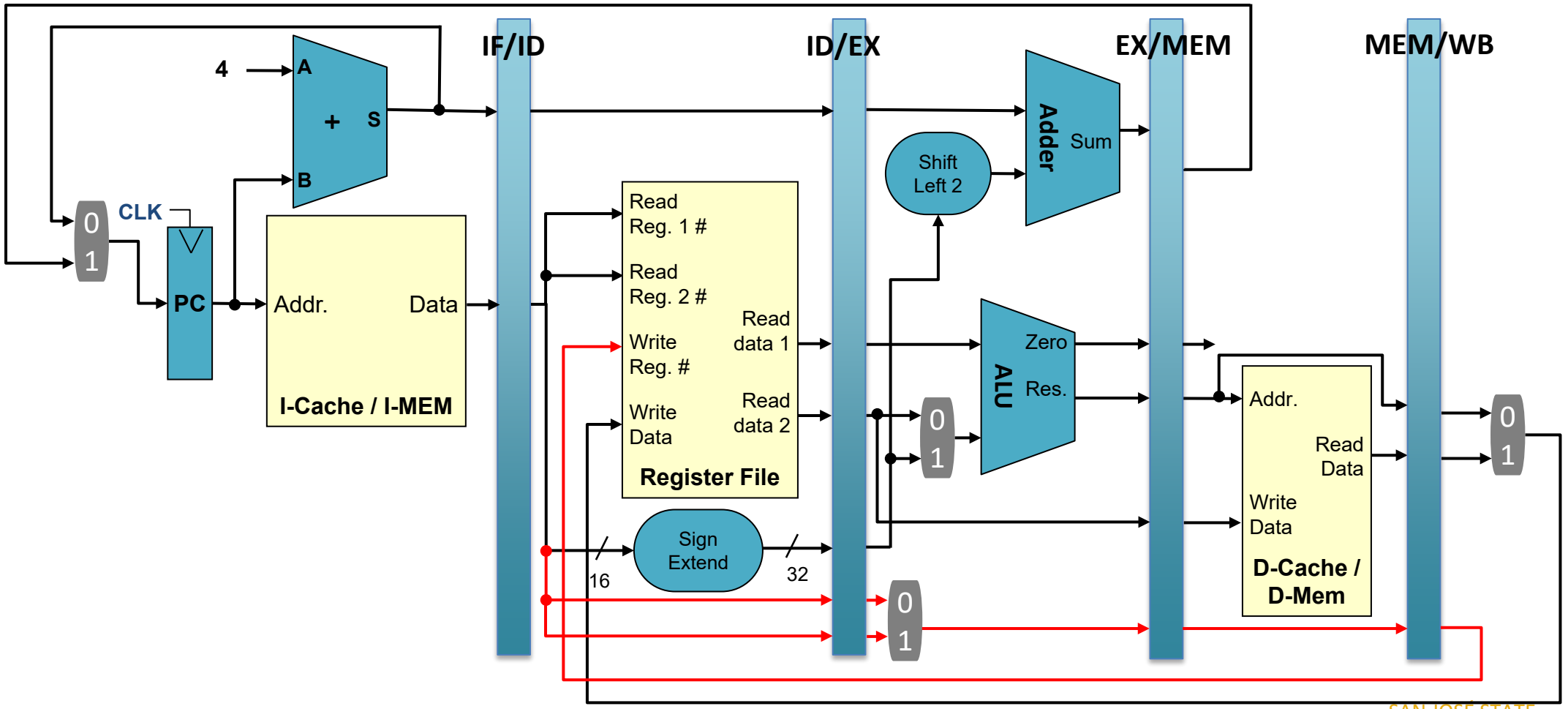


Pipeline Example: LW $t1, 4($s0)

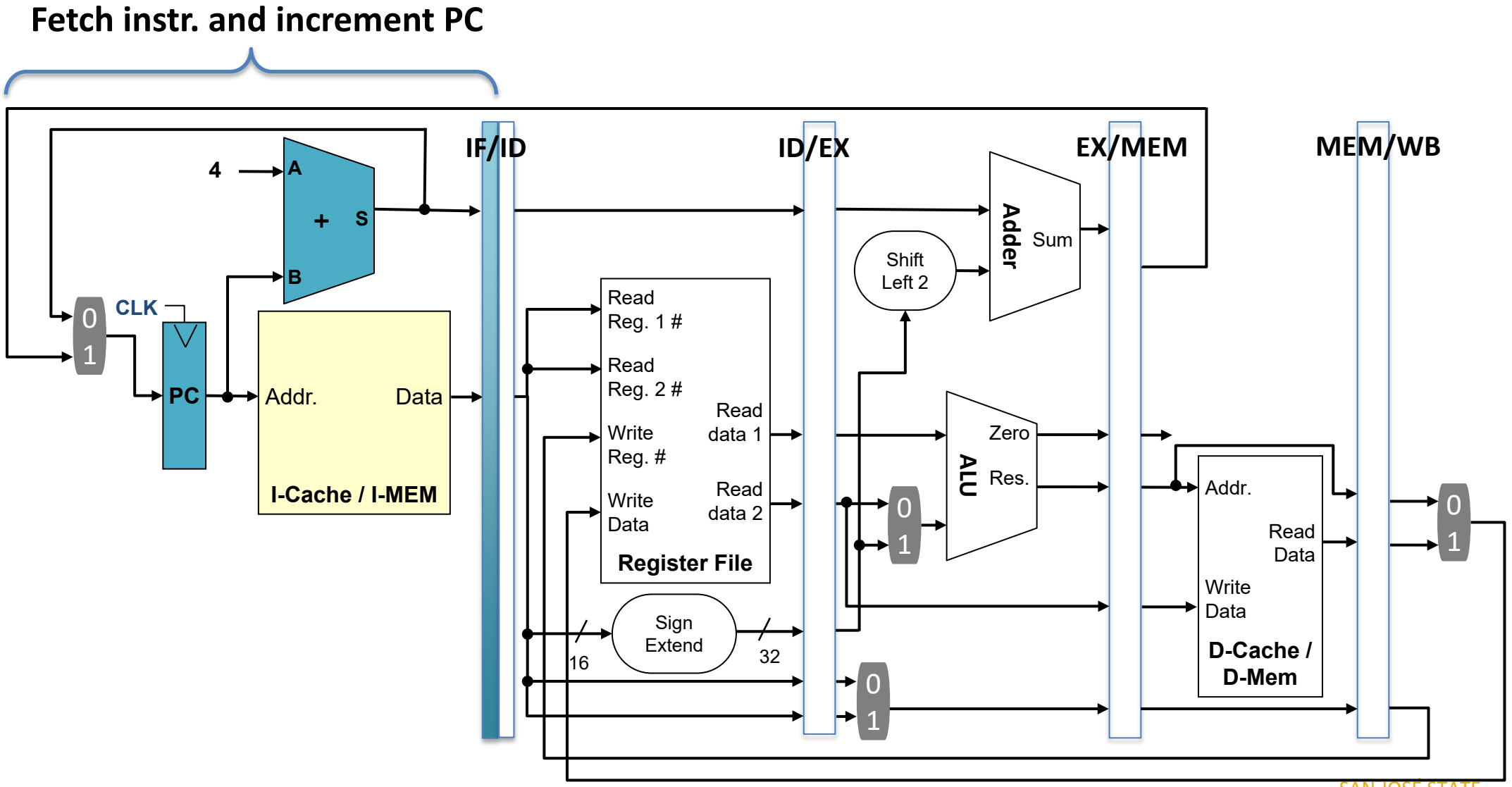


Revised Datapath

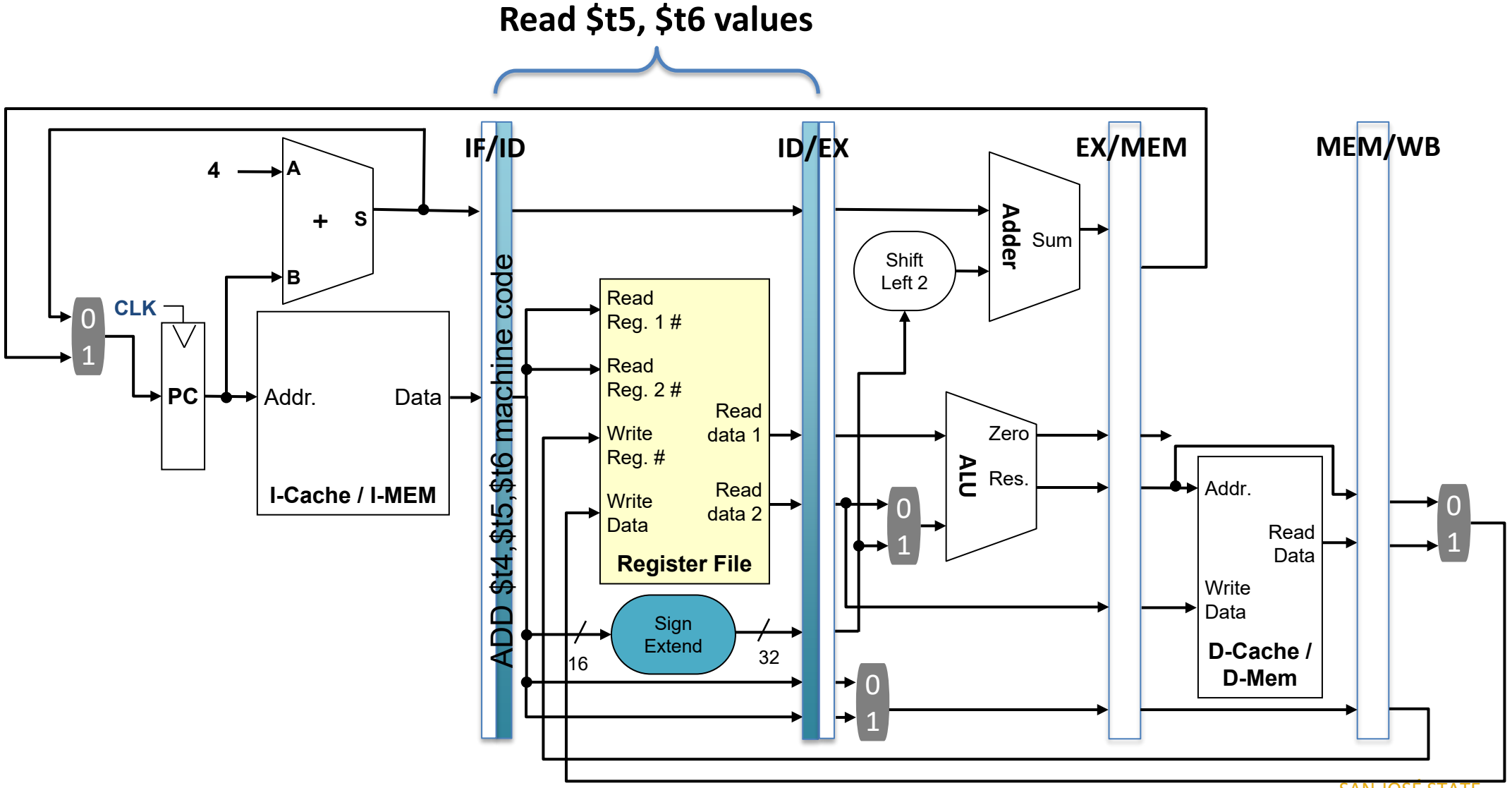
- Dest. register number should be stored in the pipeline registers
- The dest. register number is passed together with data during writeback



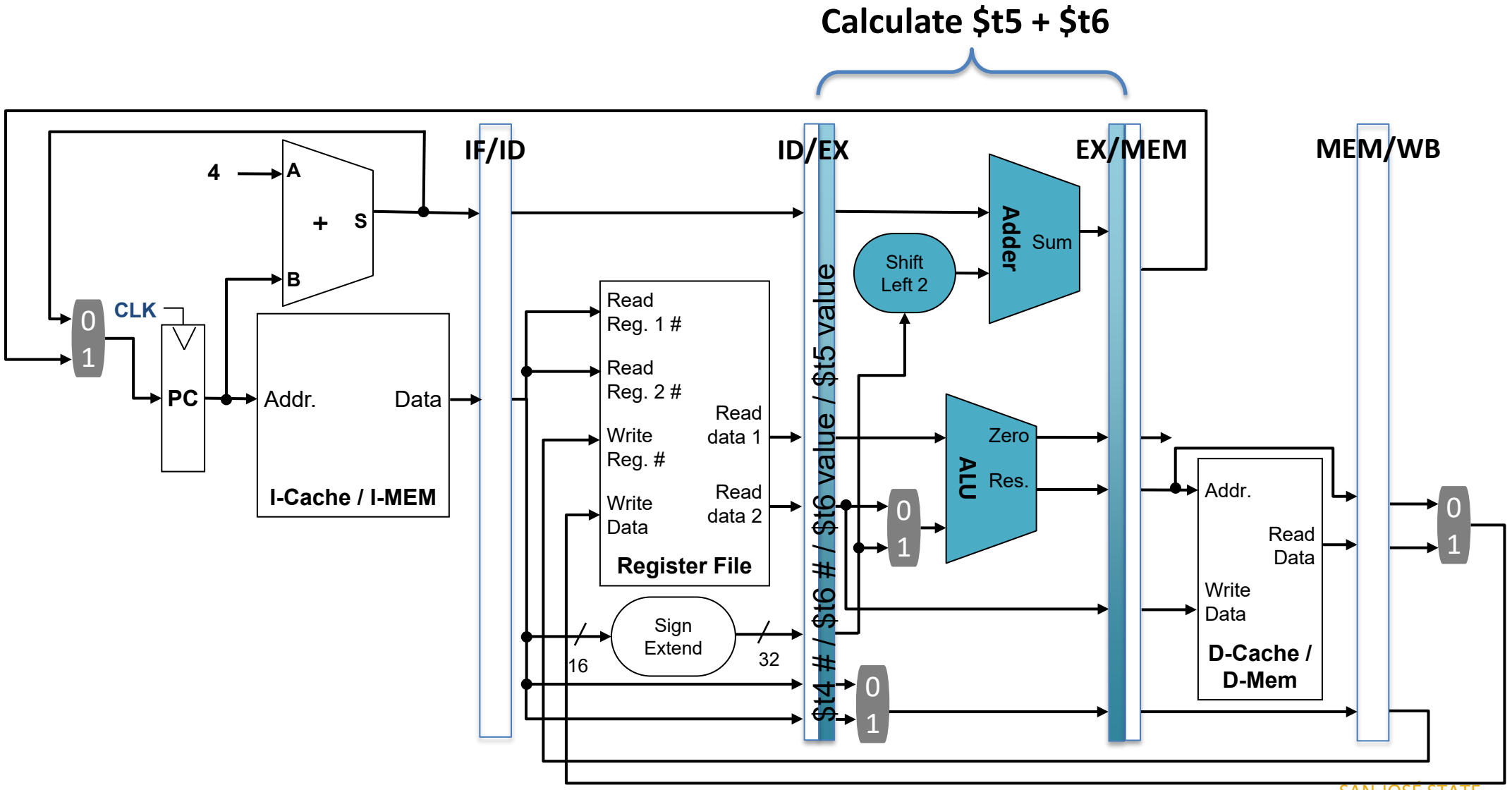
Pipeline Example: ADD $t4,$t5,$t6



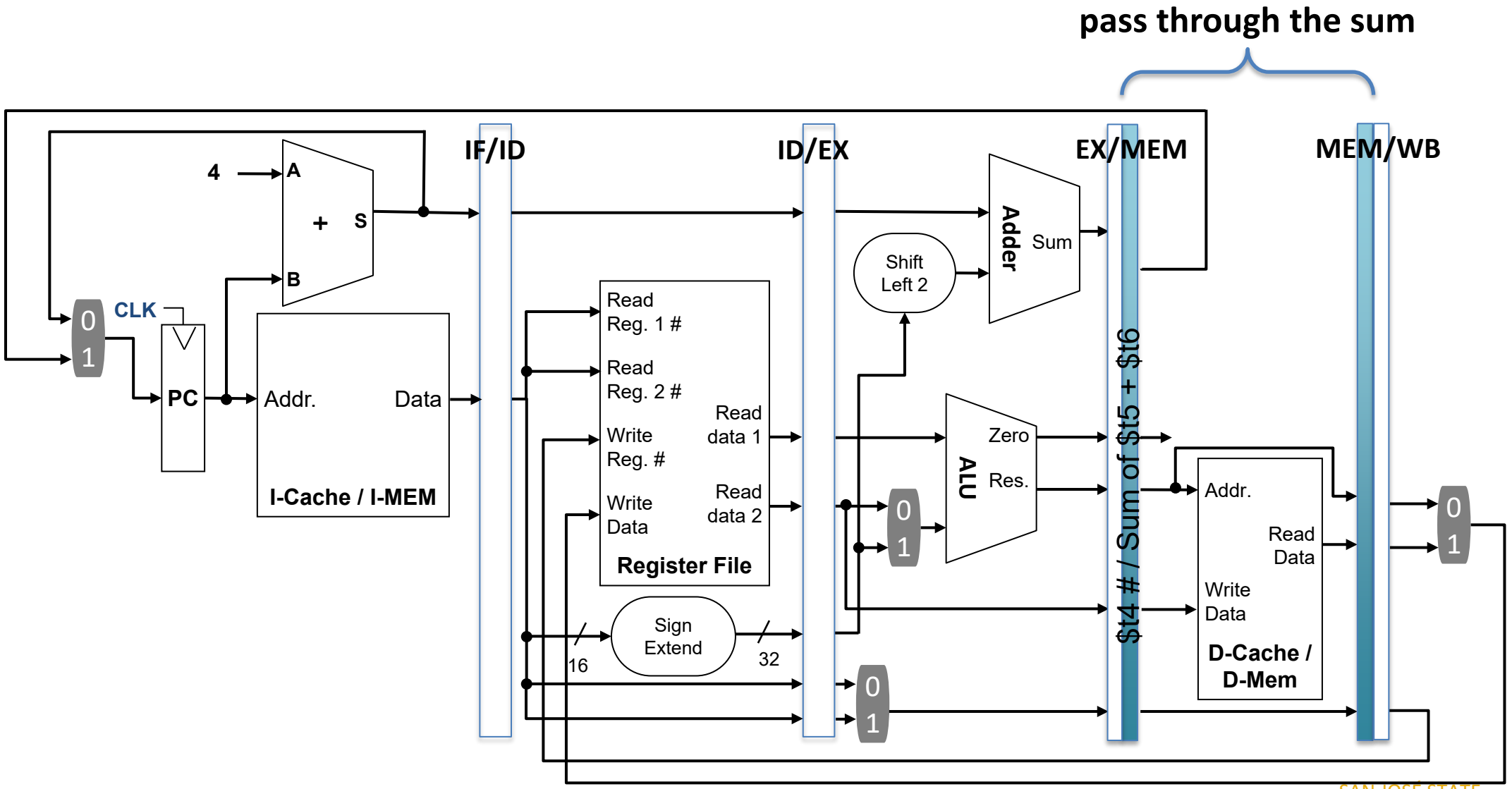
Pipeline Example: ADD $t4,$t5,$t6



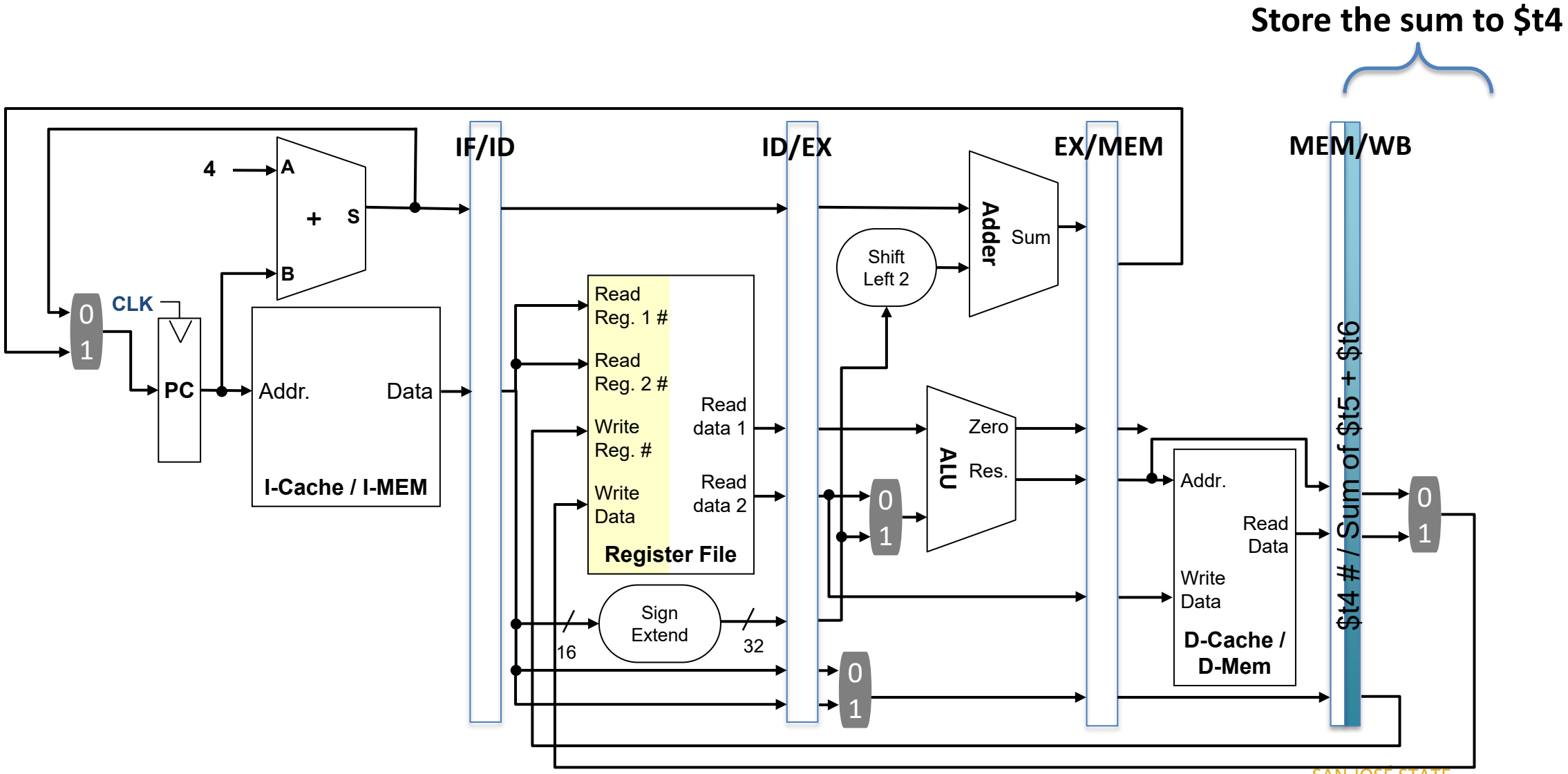
Pipeline Example: ADD $t4,$t5,$t6



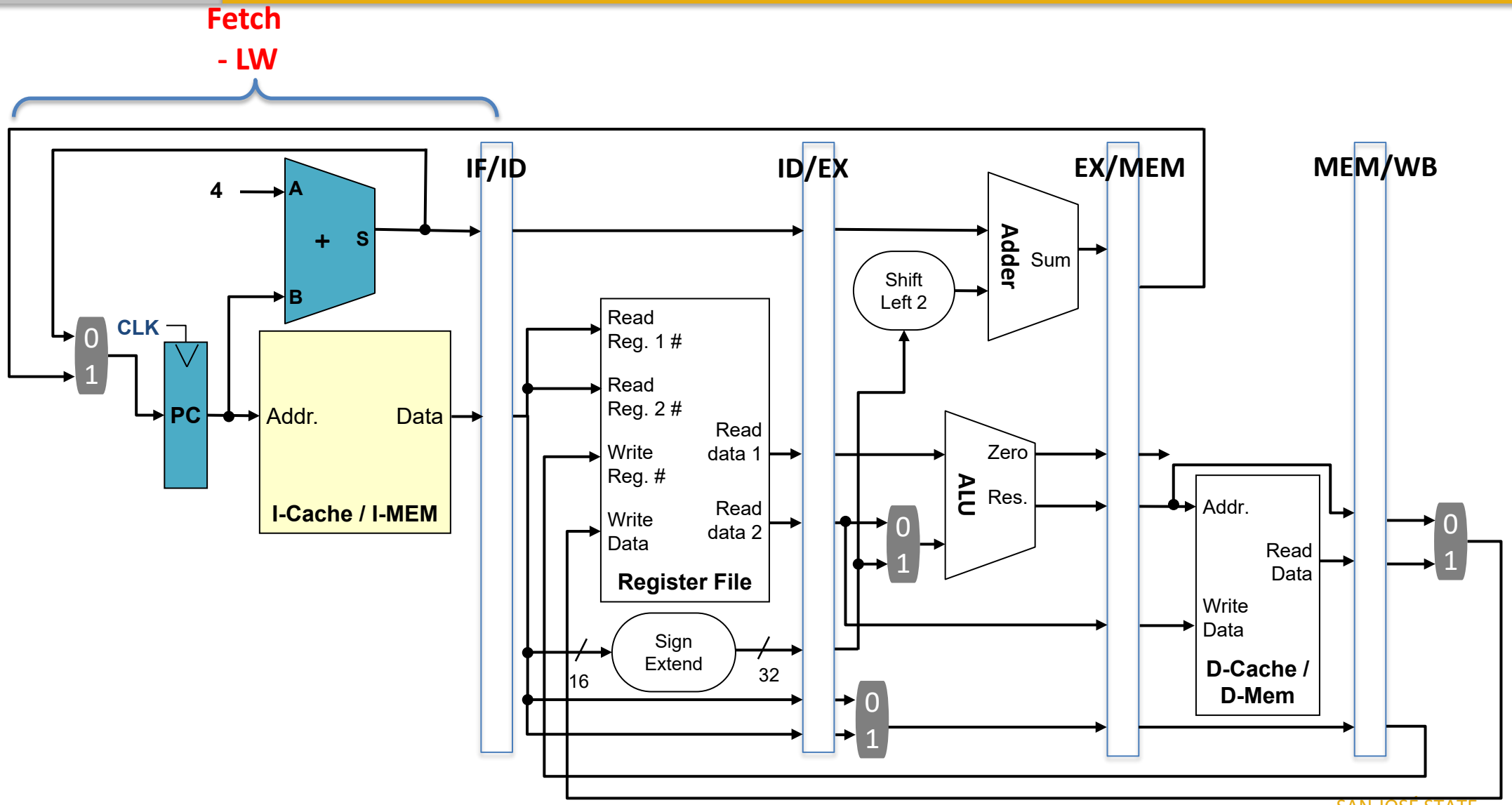
Pipeline Example: ADD $t4,$t5,$t6



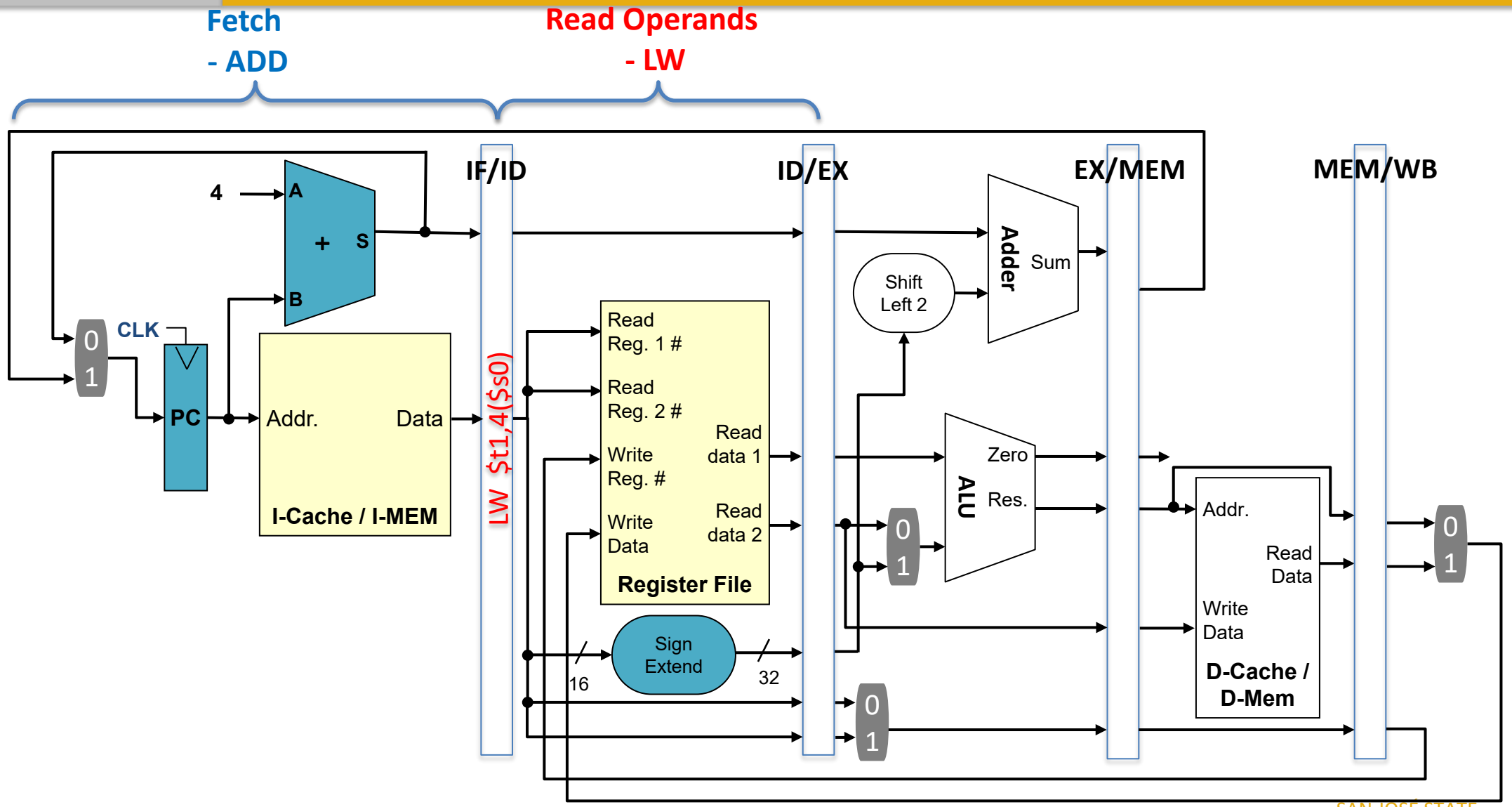
Pipeline Example: ADD $t4,$t5,$t6



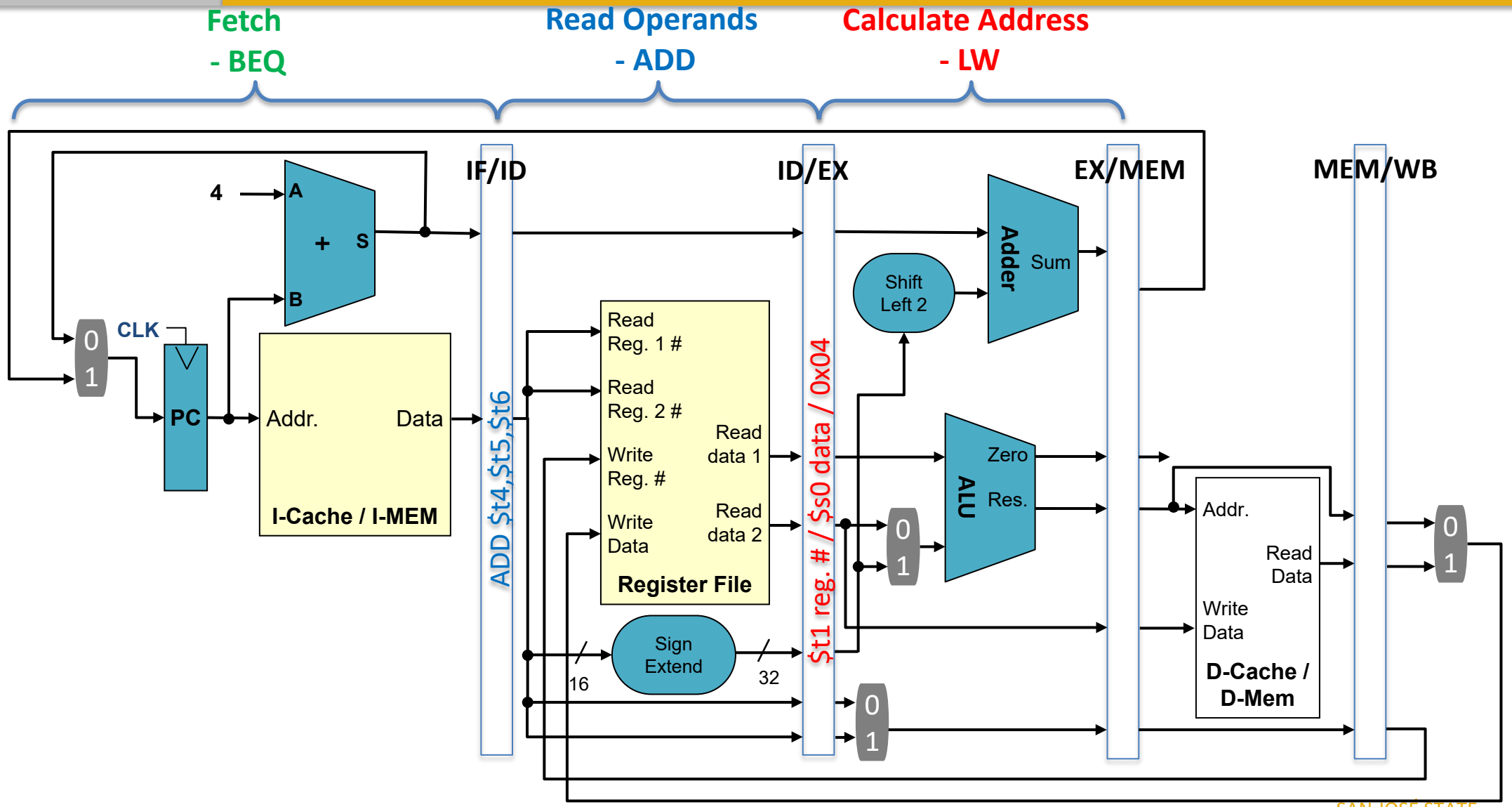
Multiple Instructions in 5-Stage Pipeline



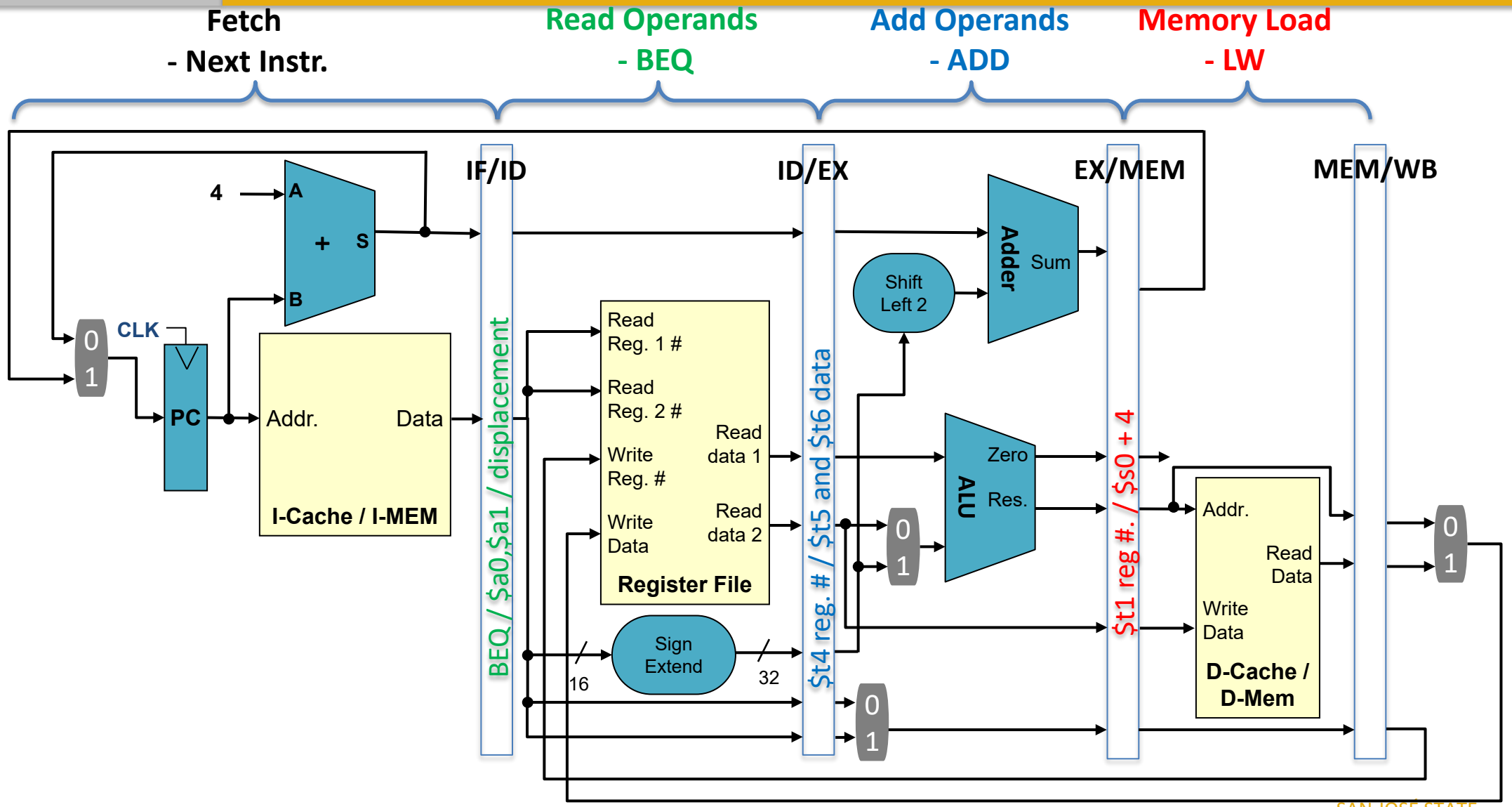
Multiple Instructions in 5-Stage Pipeline



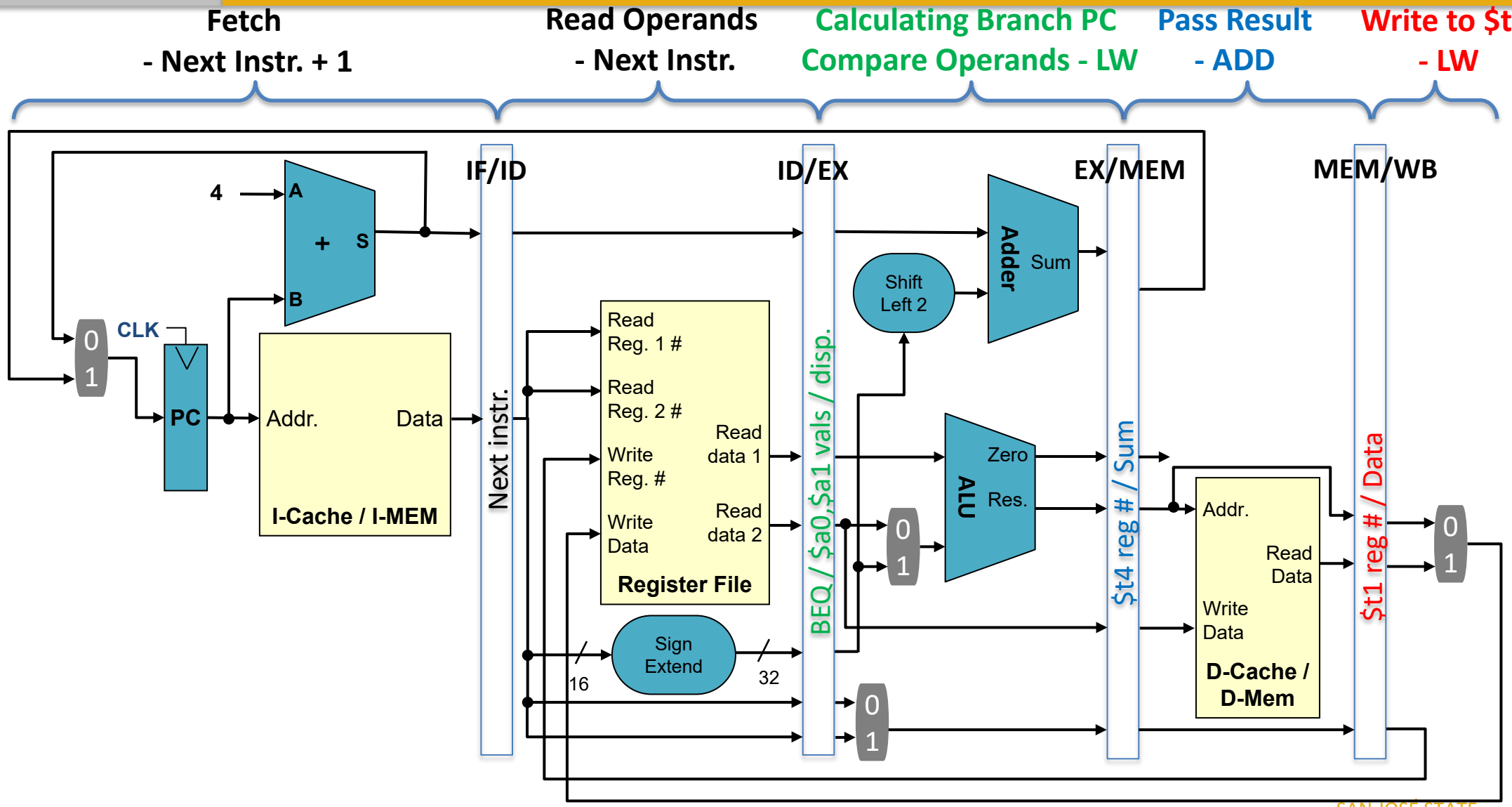
Multiple Instructions in 5-Stage Pipeline



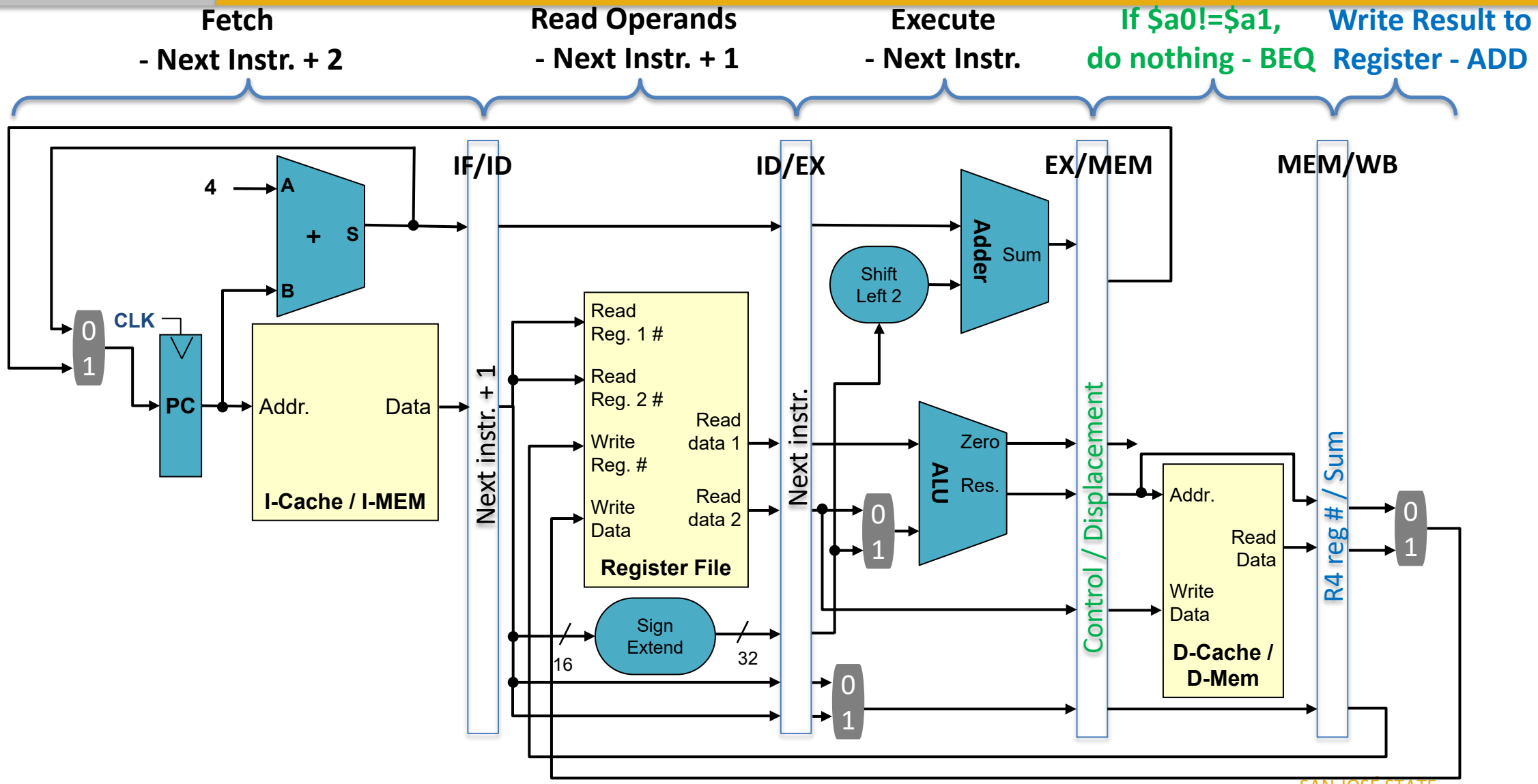
Multiple Instructions in 5-Stage Pipeline



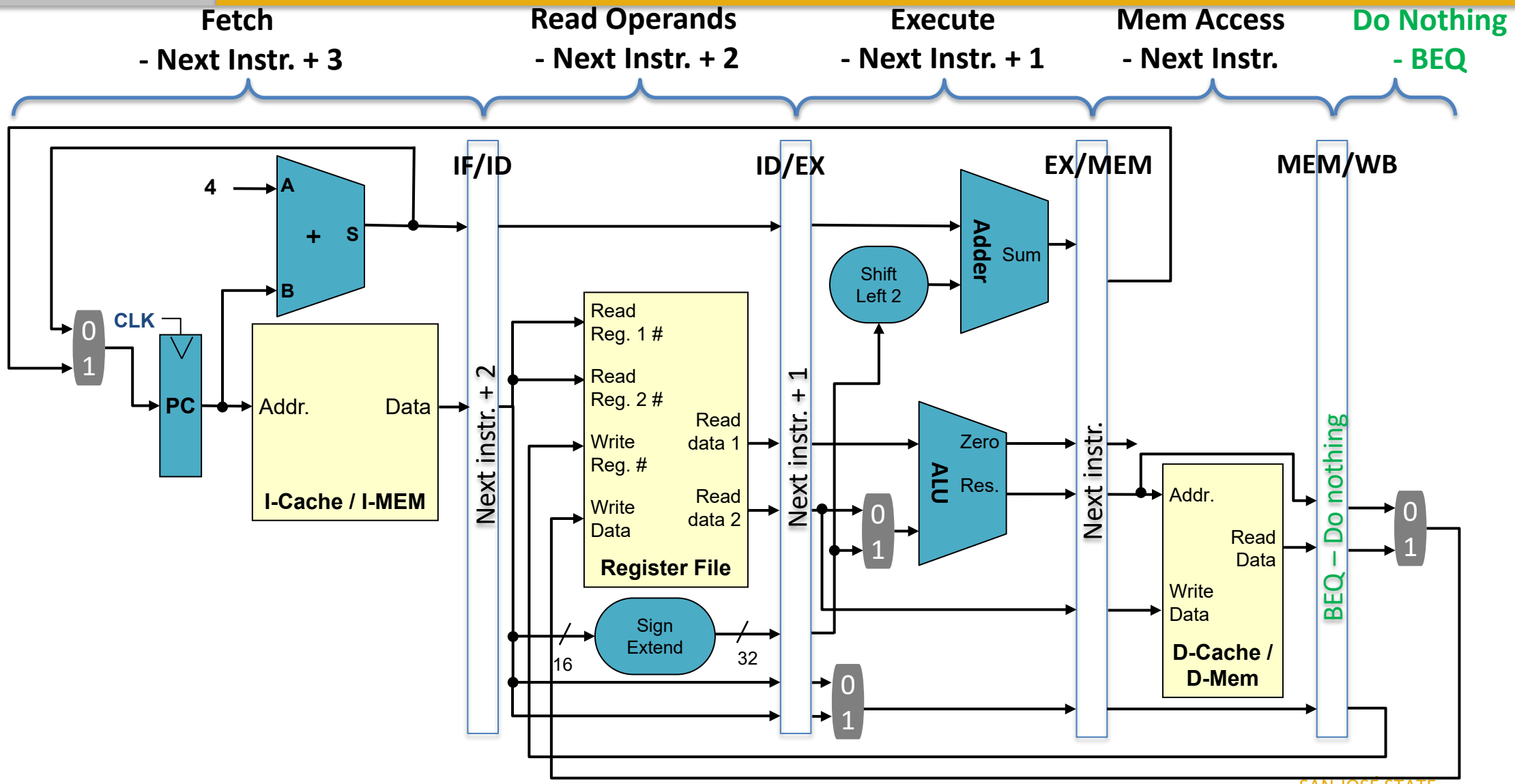
Multiple Instructions in 5-Stage Pipeline



Multiple Instructions in 5-Stage Pipeline



Multiple Instructions in 5-Stage Pipeline



Pipelined Timing

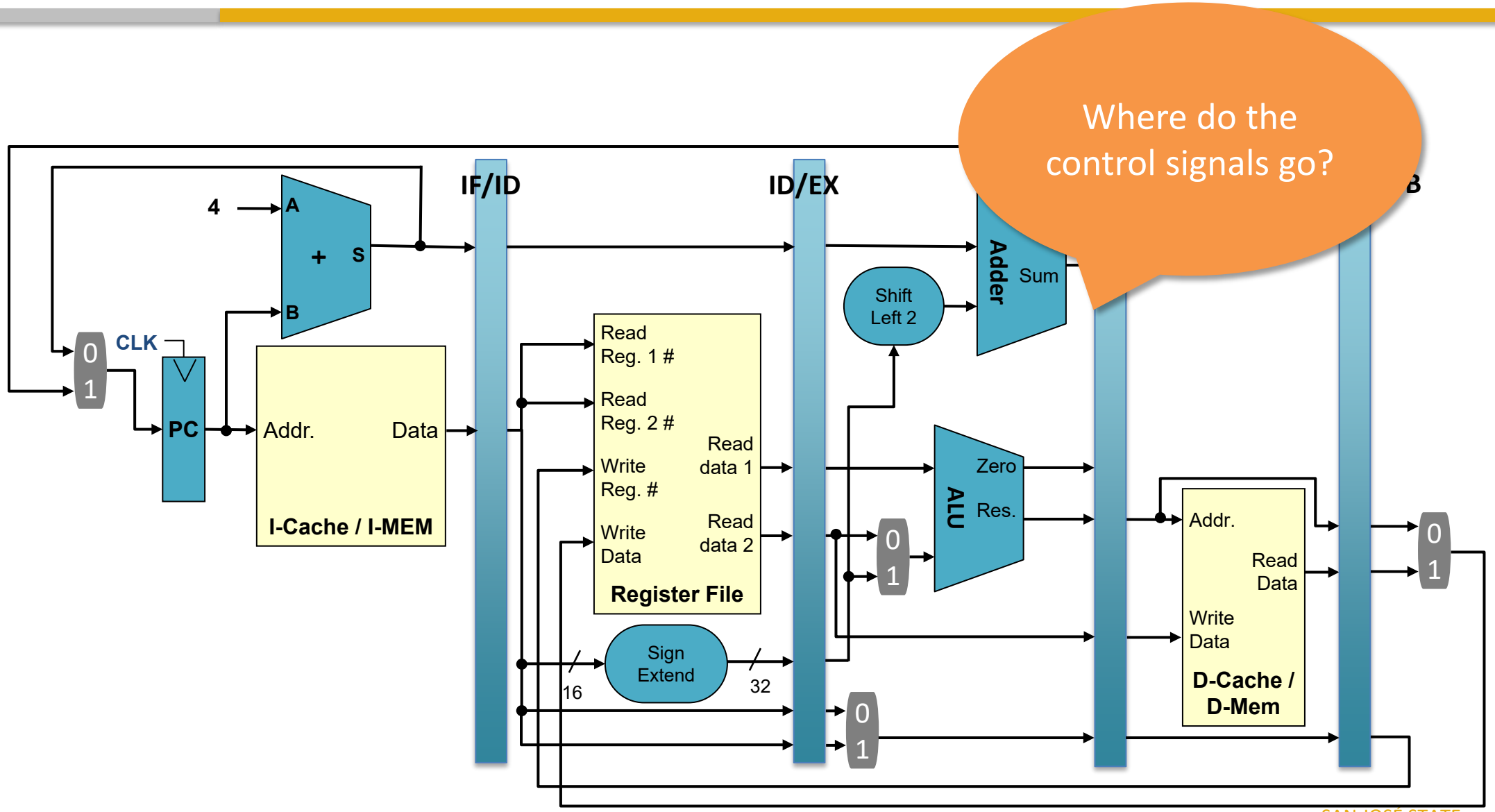
- Suppose we execute **N instructions** using a **K stage** datapath
- Total execution cycle: ***K+N-1 cycles***
  - K cycle for 1st inst + (N-1) cycles for remaining insts
  - Assume we keep the pipeline full

7 Instrs. =

11 clocks (5 + 7 – 1)

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11
LW	IF	ID	EXE	MEM	WB						
ADD		IF	ID	EXE	MEM	WB					
SUB			IF	ID	EXE	MEM	WB				
SW				IF	ID	EXE	MEM	WB			
AND					IF	ID	EXE	MEM	WB		
OR						IF	ID	EXE	MEM	WB	
XOR							IF	ID	EXE	MEM	WB

5-Stage Pipelined CPU



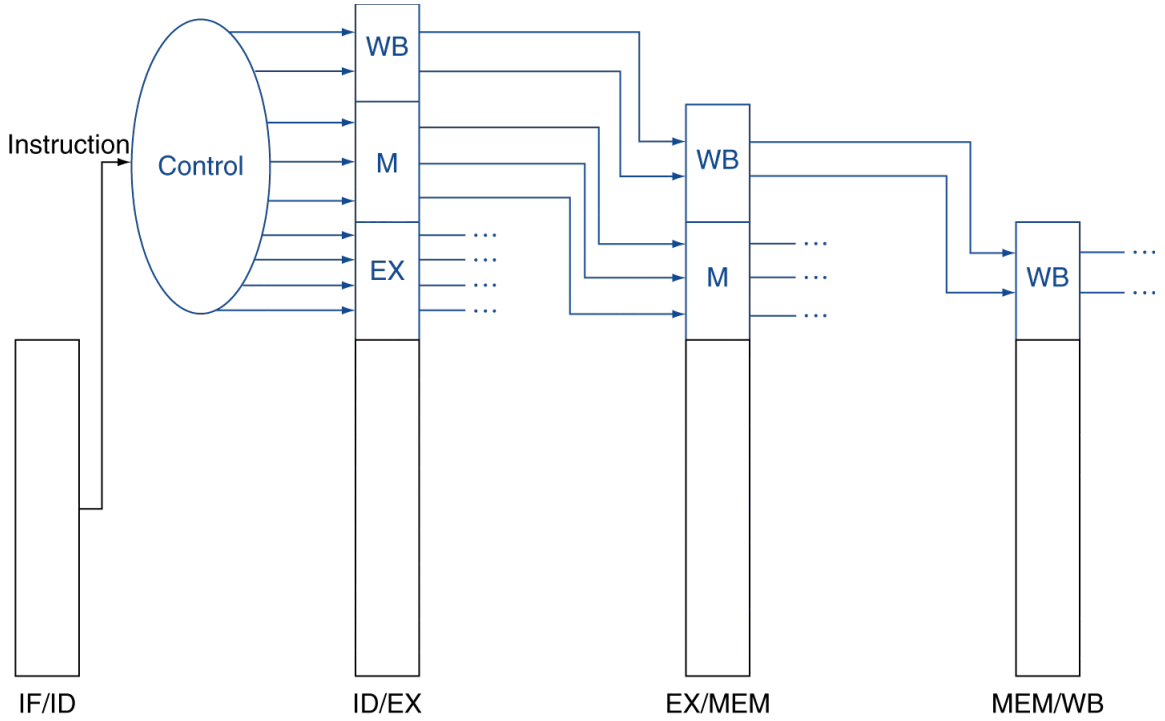
Pipelined Control

- **Using the same control signals as in Single-cycle CPU**
- **Grouped in three pipeline stages**

Instruction	EXE stage			MEM stage		WB stage	
	$alu\_op_{1:0}$	reg_dst	alu_src	branch	we_dm	dm2reg	we_reg
R-type	10	1	0	0	0	0	1
lw	00	0	1	0	0	1	1
sw	00	X	1	0	1	X	0
beq	01	X	0	1	0	X	0

Pipelined Control

- **Control signals are generated in ID stage and used by the following three stages**
- **Pipeline registers are extended to also include control information**



Throughput

- **Throughput (T) = # of instructions executed / time**
- **IPC = Instruction Per Cycle = 1 / CPI**
  - For a large number of instructions, the IPC of a pipelined processor is **1 instruction per clock cycle**
  - **Only when we keep the pipeline full of instructions**

Assume we execute **N instructions** using a **K stage** datapath

Pipeline IPC	N / (N+K-1)
Let n → ∞ (n = infinity)	1

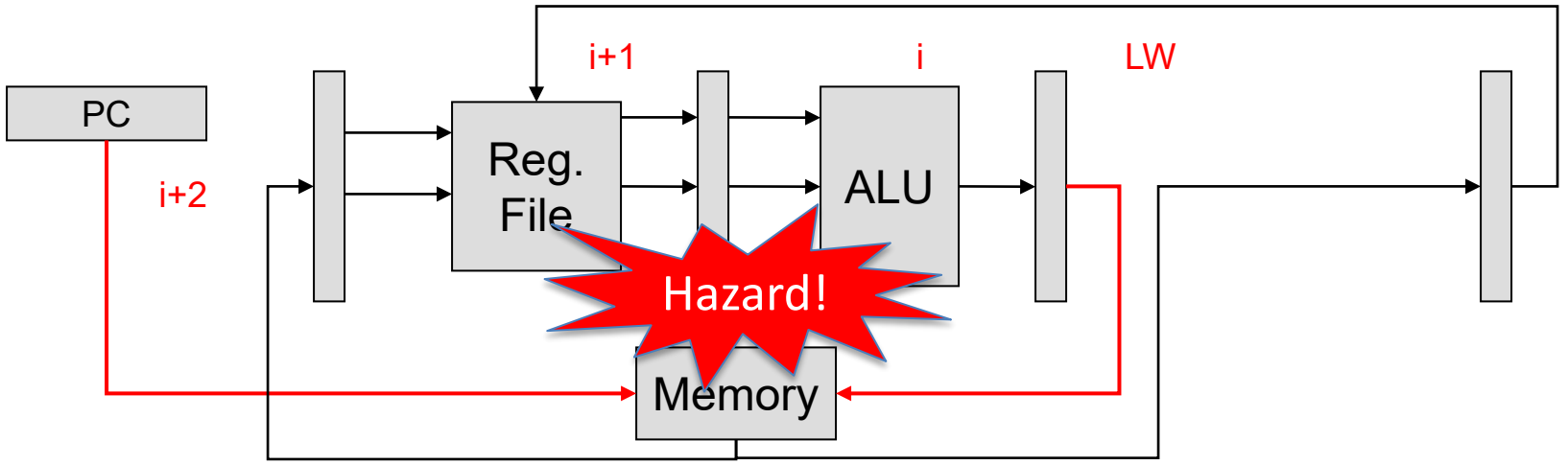
Hazards

- **Any sequence of instructions that prevent full pipeline utilization**
  - Often causes the pipeline to **stall** for one or more instructions
- **Structural Hazards**
  - HW organization cannot support certain combinations of instructions being overlapped
- **Data Hazards**
  - Data dependencies
- **Control Hazards**
  - Branches & changes to PC in the pipeline
  - If branch is determined to be taken later in the pipeline, flush (delete) the instructions in the pipeline that shouldn't be executed

Structural Hazards

- **Combinations of instructions that cannot be overlapped in the given order due to HW constraints**
  - Often due to lack of HW resources

- **Example: A single memory rather than separate I & D Memories**
  - Structural hazard whenever an instruction needs to perform a data access (i.e. 'lw' or 'sw')



Data Hazards

- **Read-After-Write (RAW)**
  - A following instruction reads a result from a previous instruction
- Write-After-Read (WAR)
- Write-After-Write (WAW)

- **RAW Example**
  - LW $t1,4($s0)
  - ADD $t5,$t1,$t4

Initial Conditions:

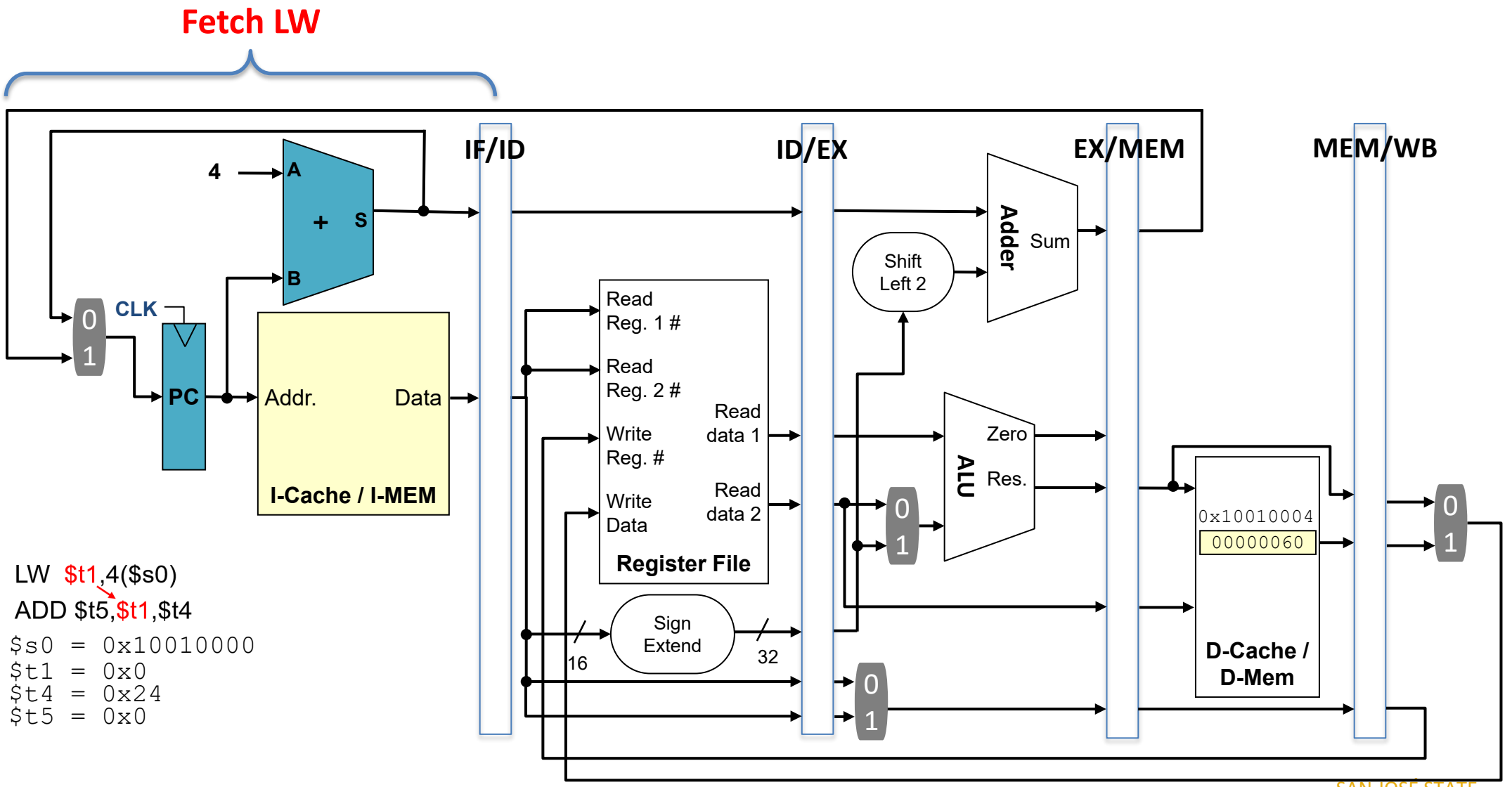
```
$s0 = 0x10010000
$t1 = 0x0
$t4 = 0x24
$t5 = 0x0
```

00000060	0x10010004
12345678	0x10010000

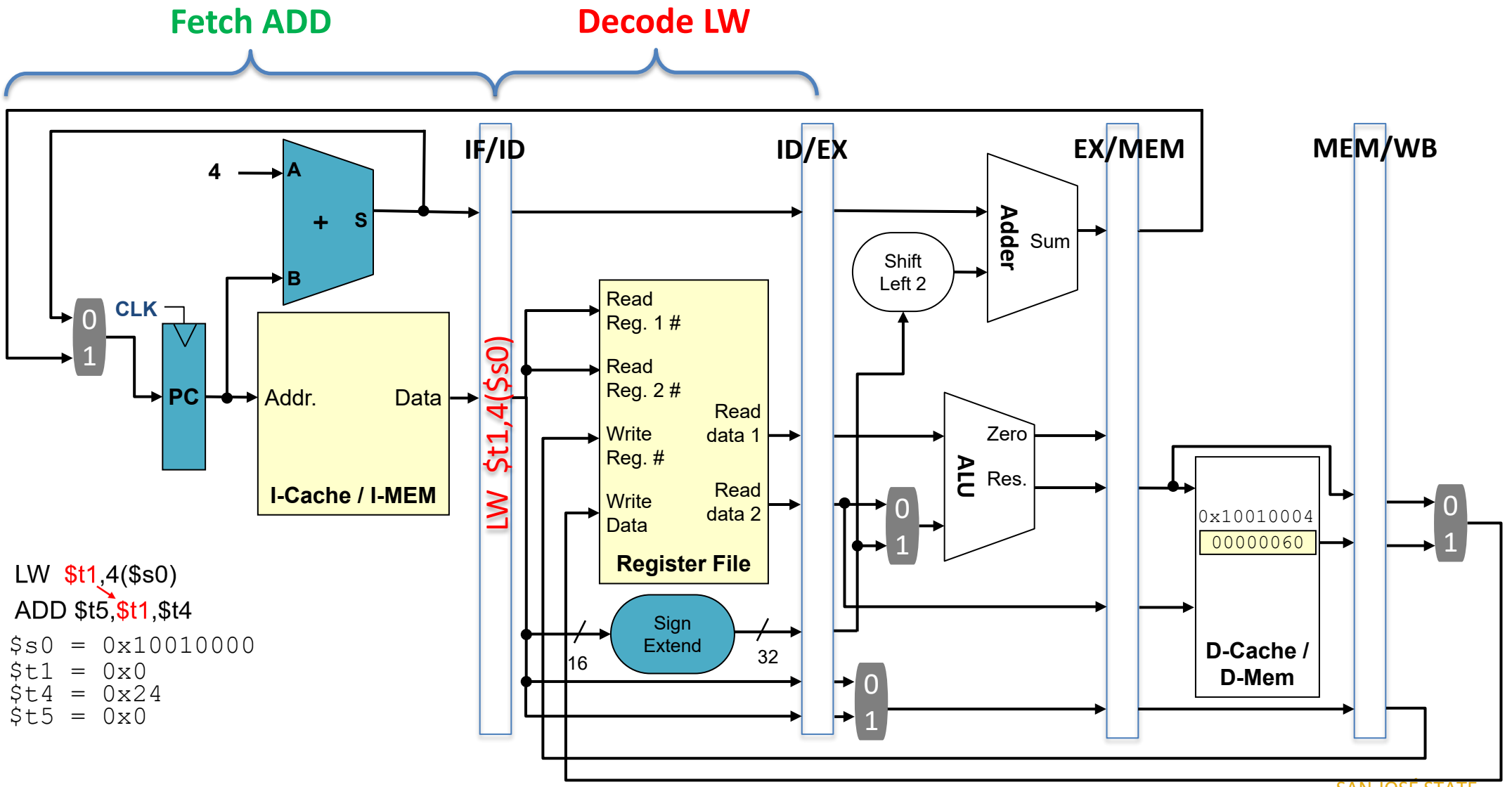
After execution values should be:

```
$s0 = 0x10010000
$t1 = 0x60
$t4 = 0x24
$t5 = 0x84
```

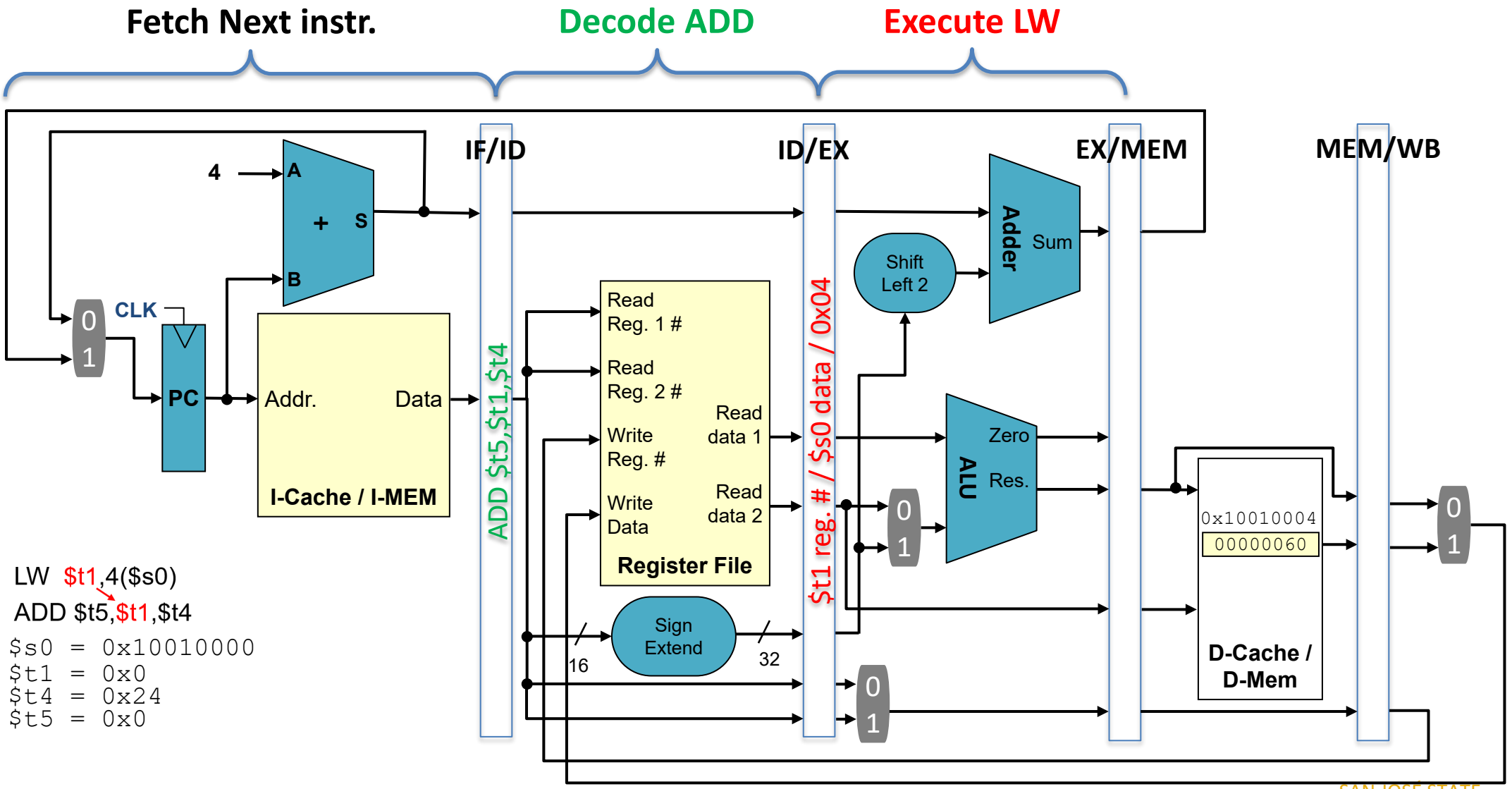
Data Hazards Example



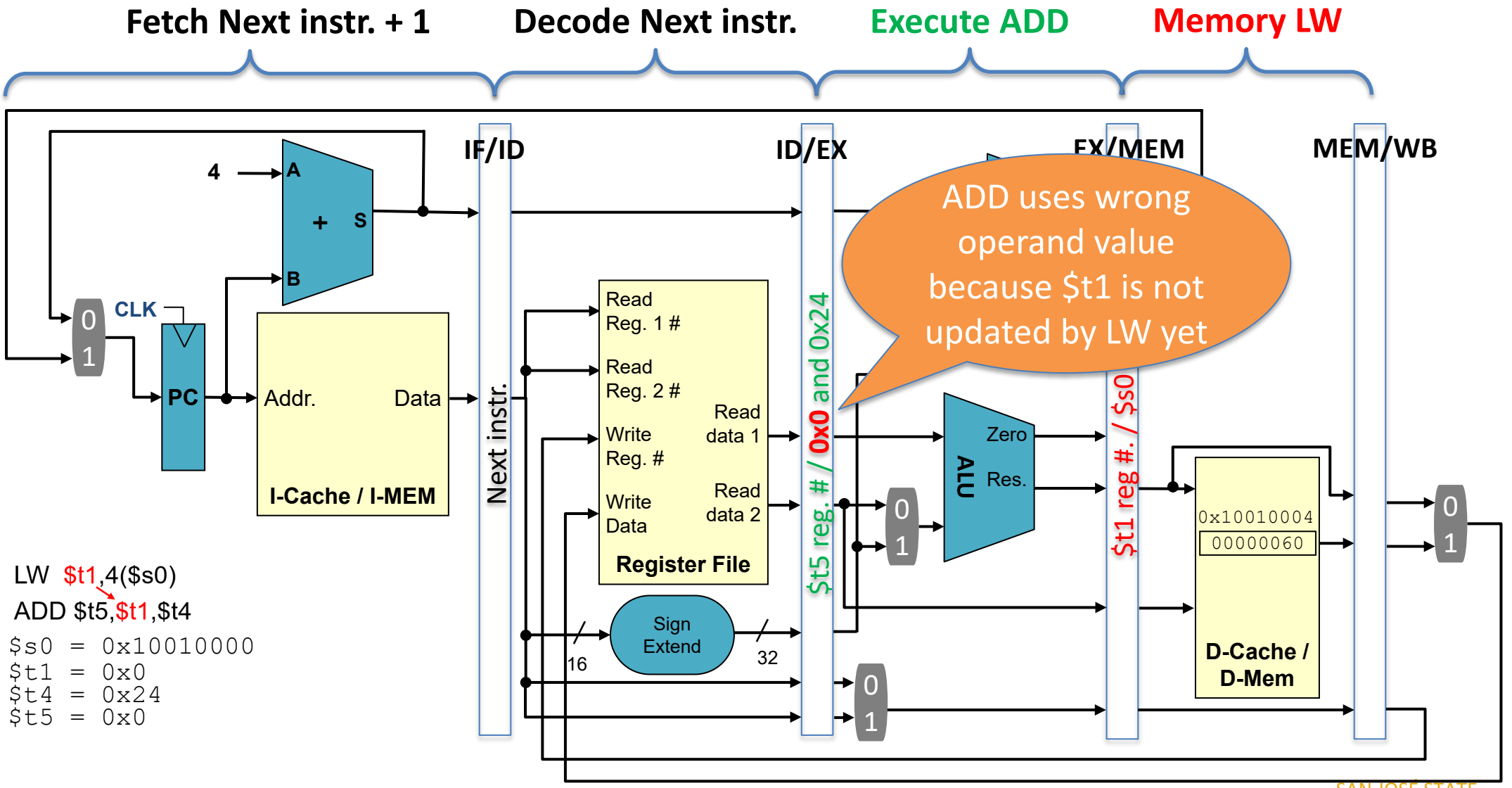
Data Hazards Example



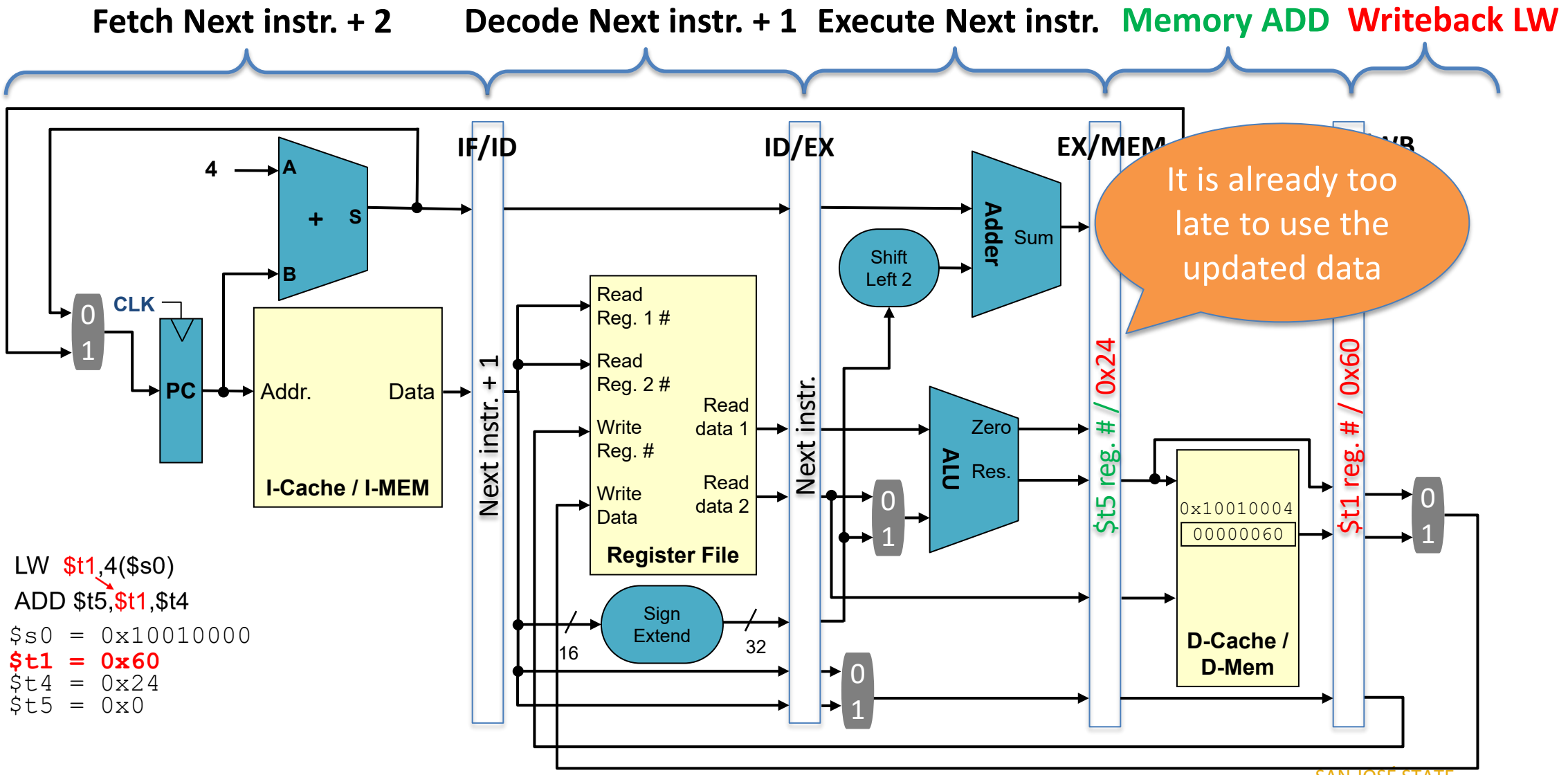
Data Hazards Example



Data Hazards Example



Data Hazards Example



Data Hazards Solutions

- **Hardware**
  - Stall the pipeline until the result is written back to the register file
  - Use forwarding / bypassing

- **Compiler**
  - Reorder Code

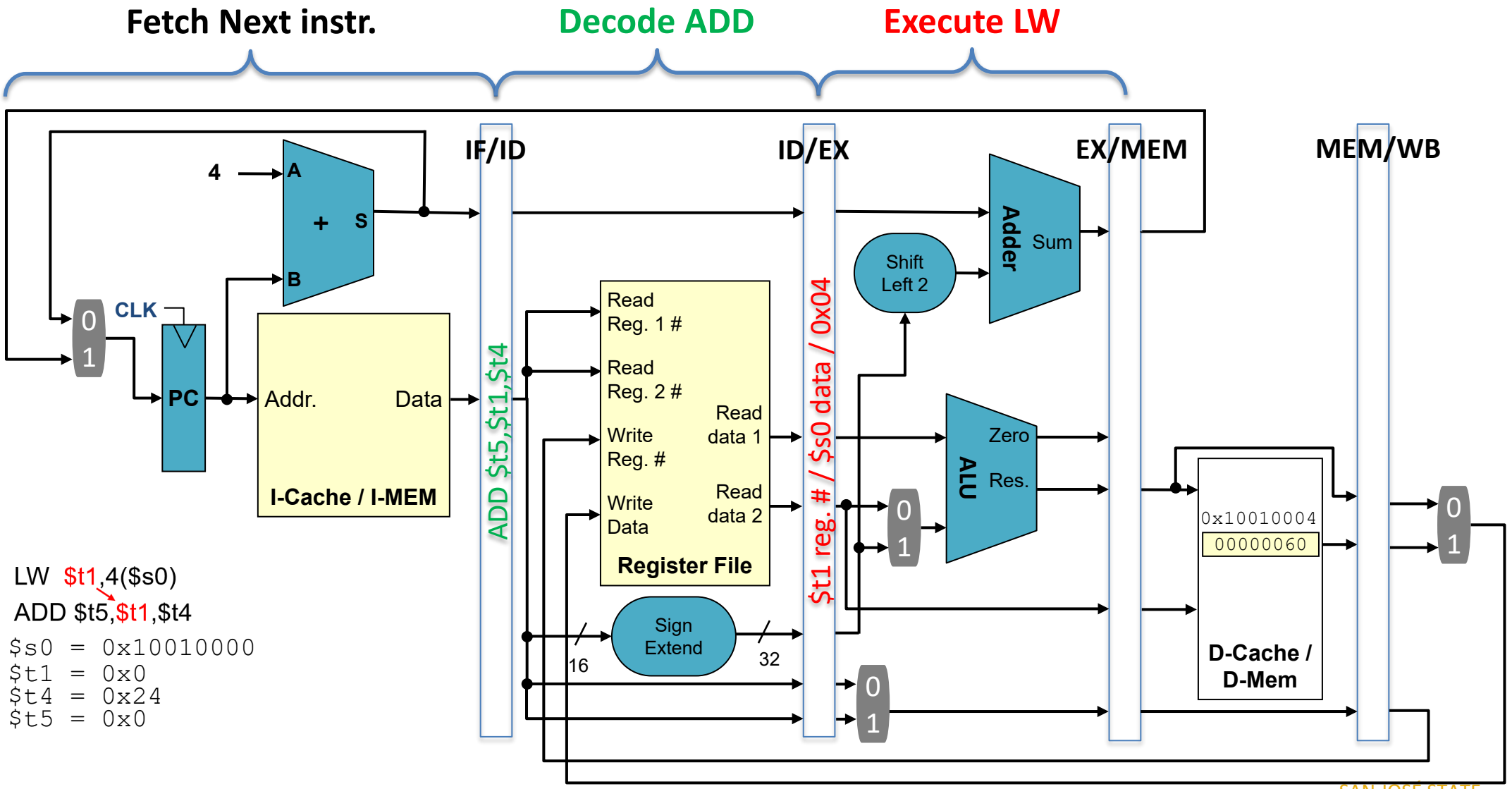
Stalling the pipeline:

- All instructions in front of the stalled instruction can continue

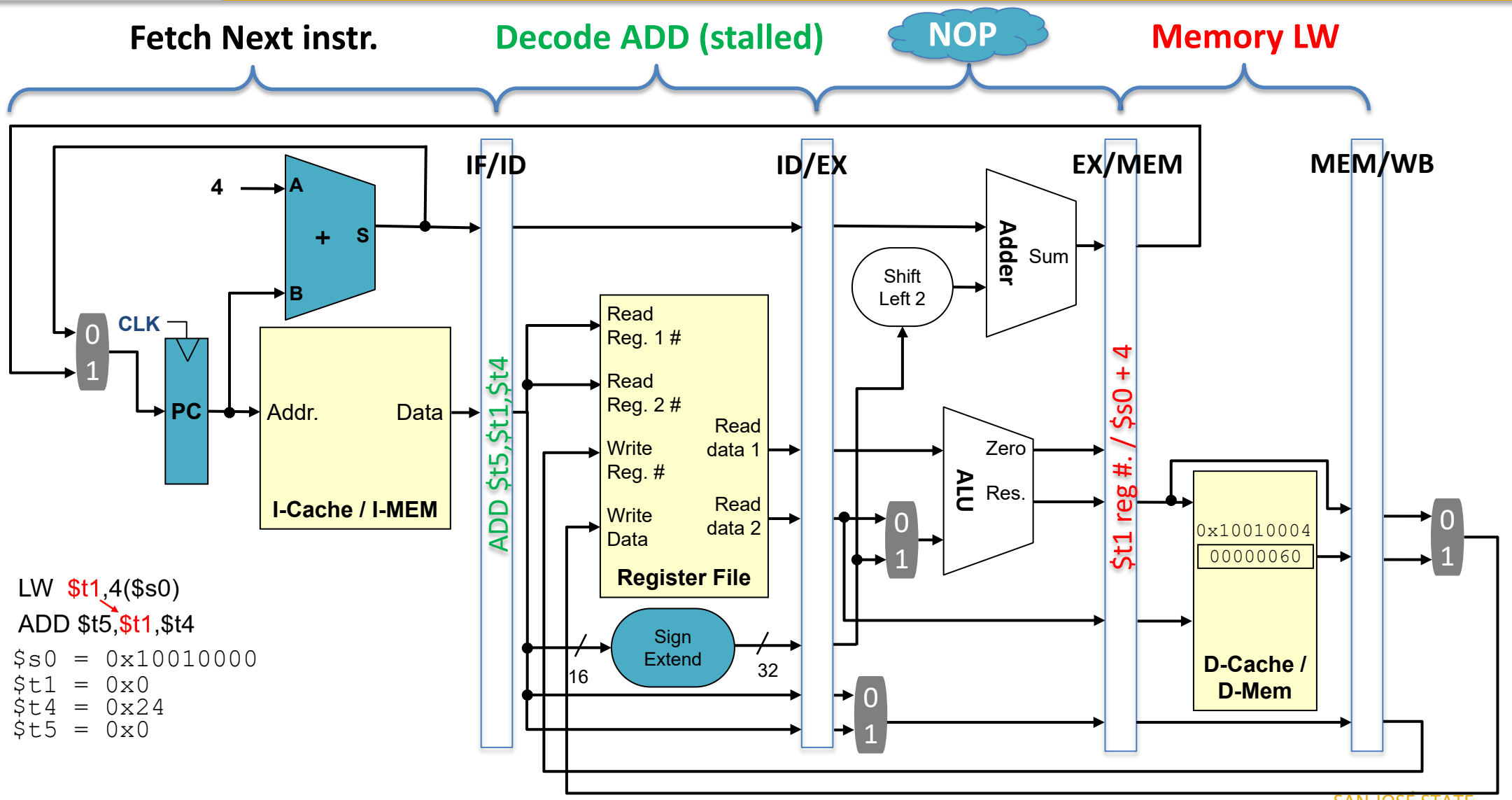
- All instructions behind the stalled instruction must also stall

- Stalling inserts “bubbles” / nops (no-operations) into the pipeline
  - A “nop” is an actual instruction in the MIPS ISA that does NOTHING

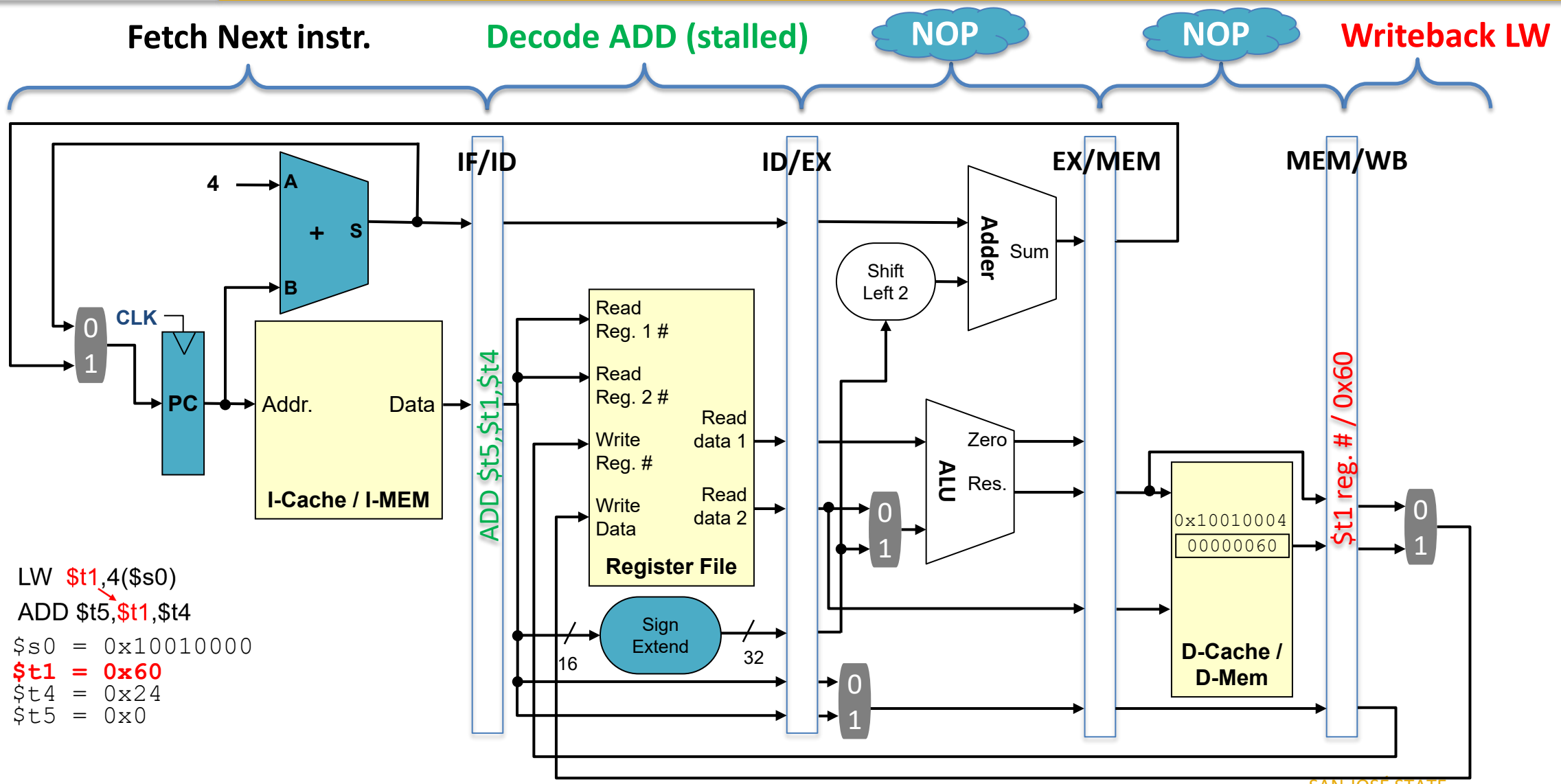
Pipeline Stall for Data Hazards



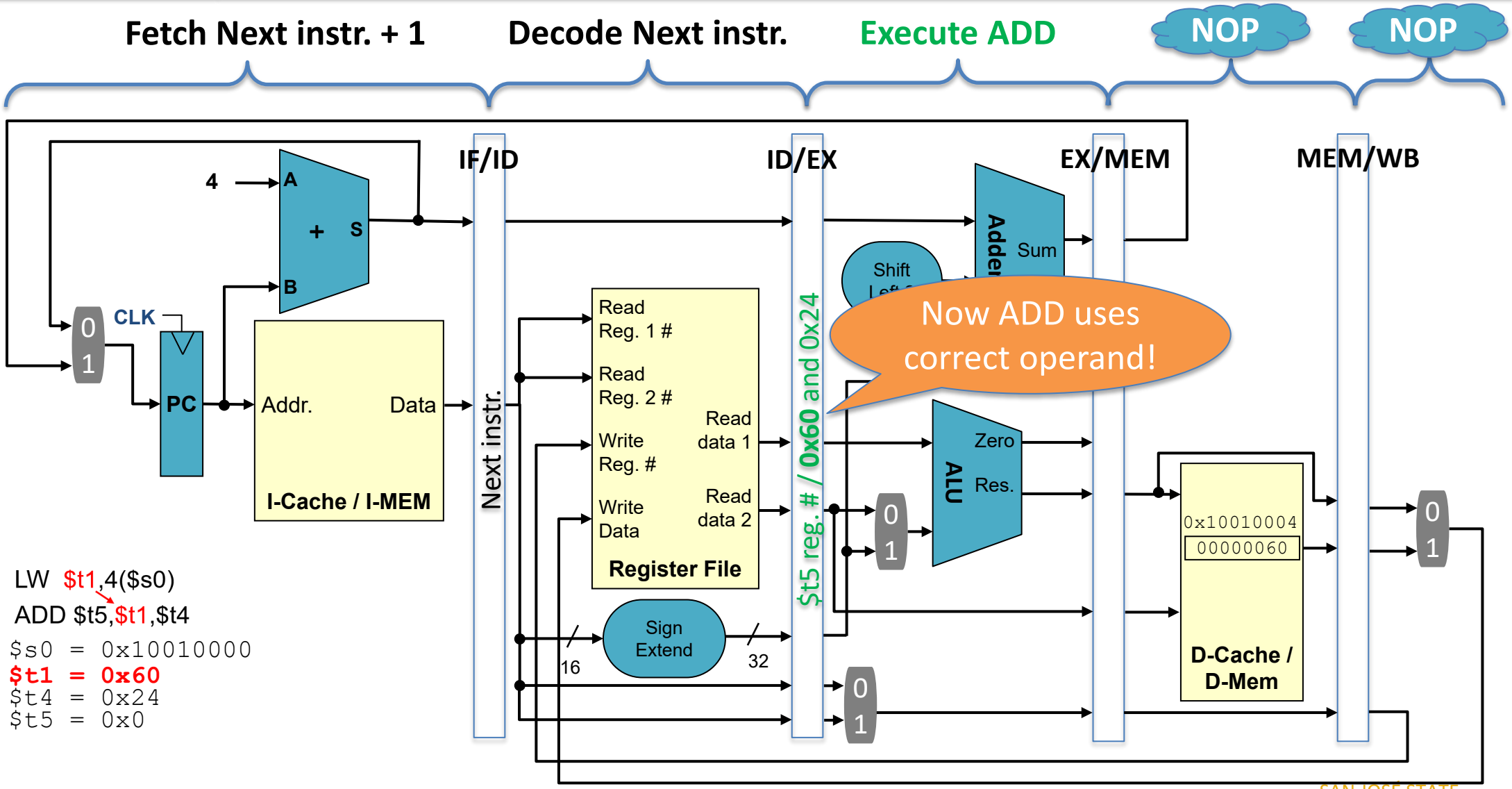
Pipeline Stall for Data Hazards



Pipeline Stall for Data Hazards



Pipeline Stall for Data Hazards



Data Hazards Example

- **Using stalls to handle dependencies**

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11
LW	IF										
ADD											
Next instr.											
Next+1											
Next+2											
Next+3											
Next+4											

Data Hazards Example

- **Using stalls to handle dependencies**

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11
LW	IF	ID									
ADD		IF									
Next instr.											
Next+1											
Next+2											
Next+3											
Next+4											

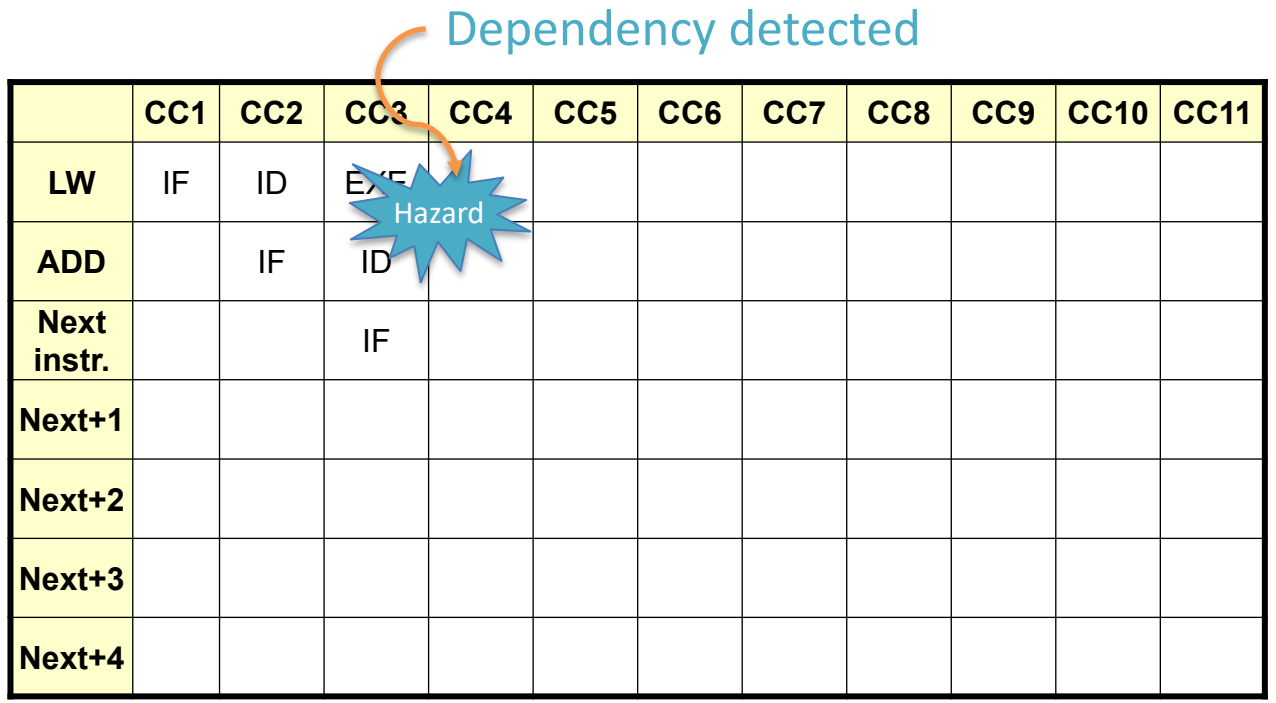
Data Hazards Example

- **Using stalls to handle dependencies**

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11
LW	IF	ID	EXE								
ADD		IF	ID								
Next instr.			IF								
Next+1											
Next+2											
Next+3											
Next+4											

Data Hazards Example

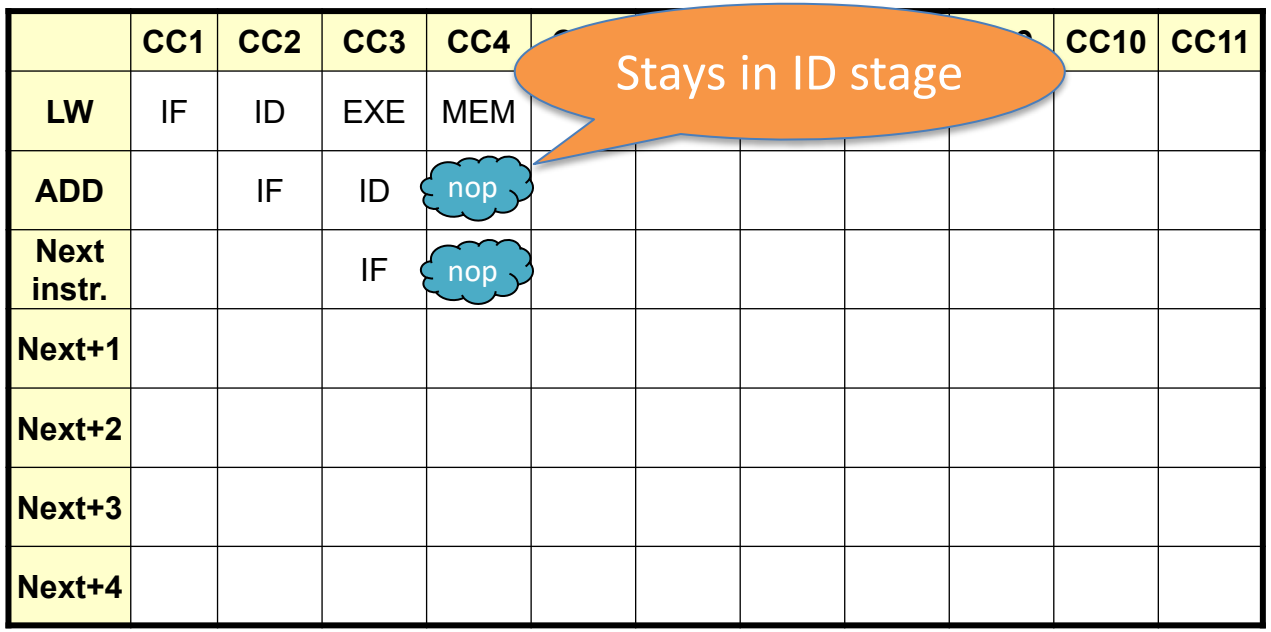
- **Using stalls to handle dependencies**



	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11
LW	IF	ID	EX								
ADD		IF	ID								
Next instr.			IF								
Next+1											
Next+2											
Next+3											
Next+4											

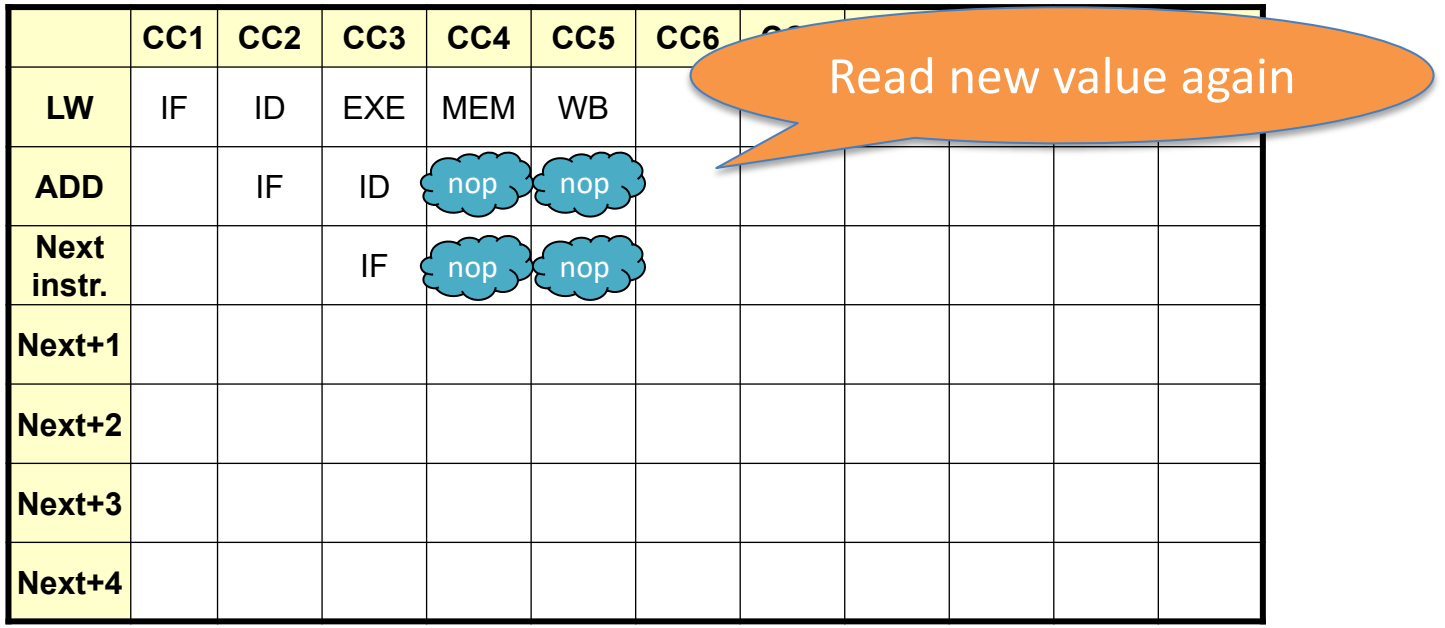
Data Hazards Example

- **Using stalls to handle dependencies**



Data Hazards Example

- **Using stalls to handle dependencies**



Data Hazards Example

- **Using stalls to handle dependencies**

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11
LW	IF	ID	EXE	MEM	WB						
ADD		IF	ID	nop	nop	EXE					
Next instr.			IF	nop	nop	ID					
Next+1						IF					
Next+2											
Next+3											
Next+4											

Data Hazards Example

- **Using stalls to handle dependencies**

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	CC12	CC13
LW	IF	ID	EXE	MEM	WB								
ADD		IF	ID	nop	nop	EXE	MEM	WB					
Next instr.			IF	nop	nop	ID	EXE	MEM	WB				
Next+1						IF	ID	EXE	MEM	WB			
Next+2							IF	ID	EXE	MEM	WB		
Next+3								IF	ID	EXE	MEM	WB	
Next+4									IF	ID	EXE	MEM	WB

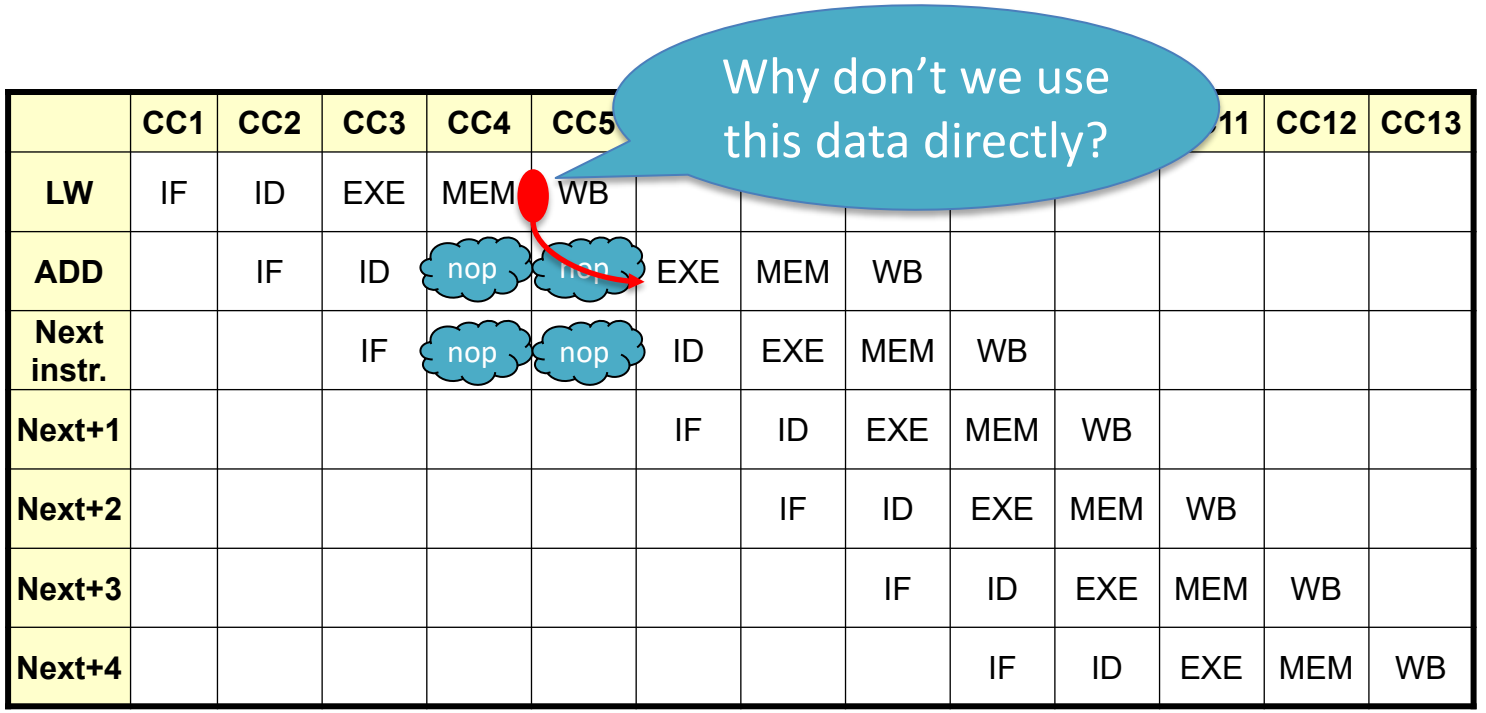
Throughput = 7 insts/13 cycles = 0.53

vs.

Throughput without hazard = 7 insts/11 cycles = 0.63

Pipeline Stall for Data Hazards

- **ADD can read the updated register value from the register file in the same cycle as LW writes the loaded data to register**
- **Do we really need to wait until LW updates the register file?**

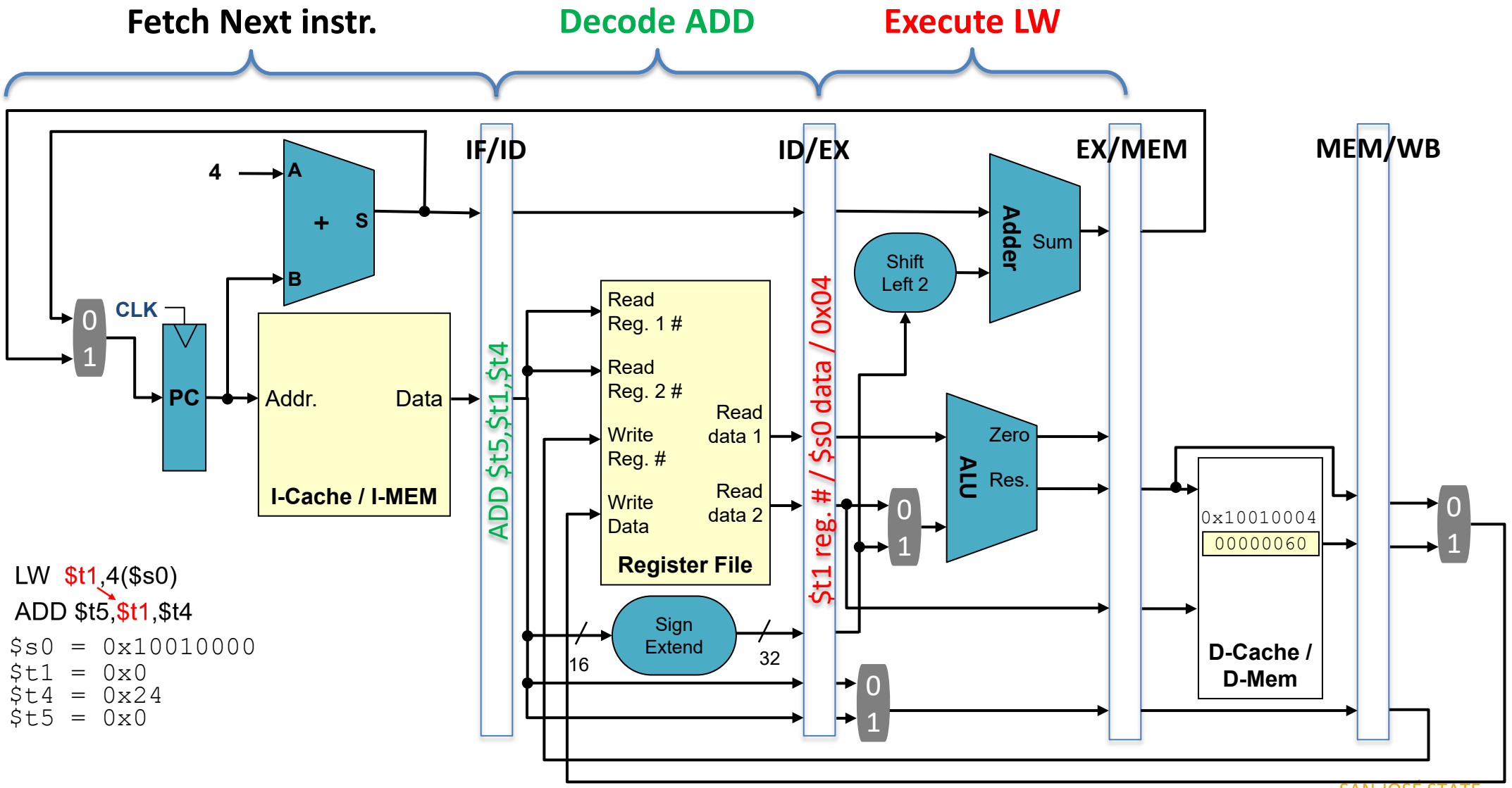


	CC1	CC2	CC3	CC4	CC5							CC12	CC13
LW	IF	ID	EXE	MEM	WB								
ADD		IF	ID	nop	nop	EXE	MEM	WB					
Next instr.			IF	nop	nop	ID	EXE	MEM	WB				
Next+1						IF	ID	EXE	MEM	WB			
Next+2							IF	ID	EXE	MEM	WB		
Next+3								IF	ID	EXE	MEM	WB	
Next+4									IF	ID	EXE	MEM	WB

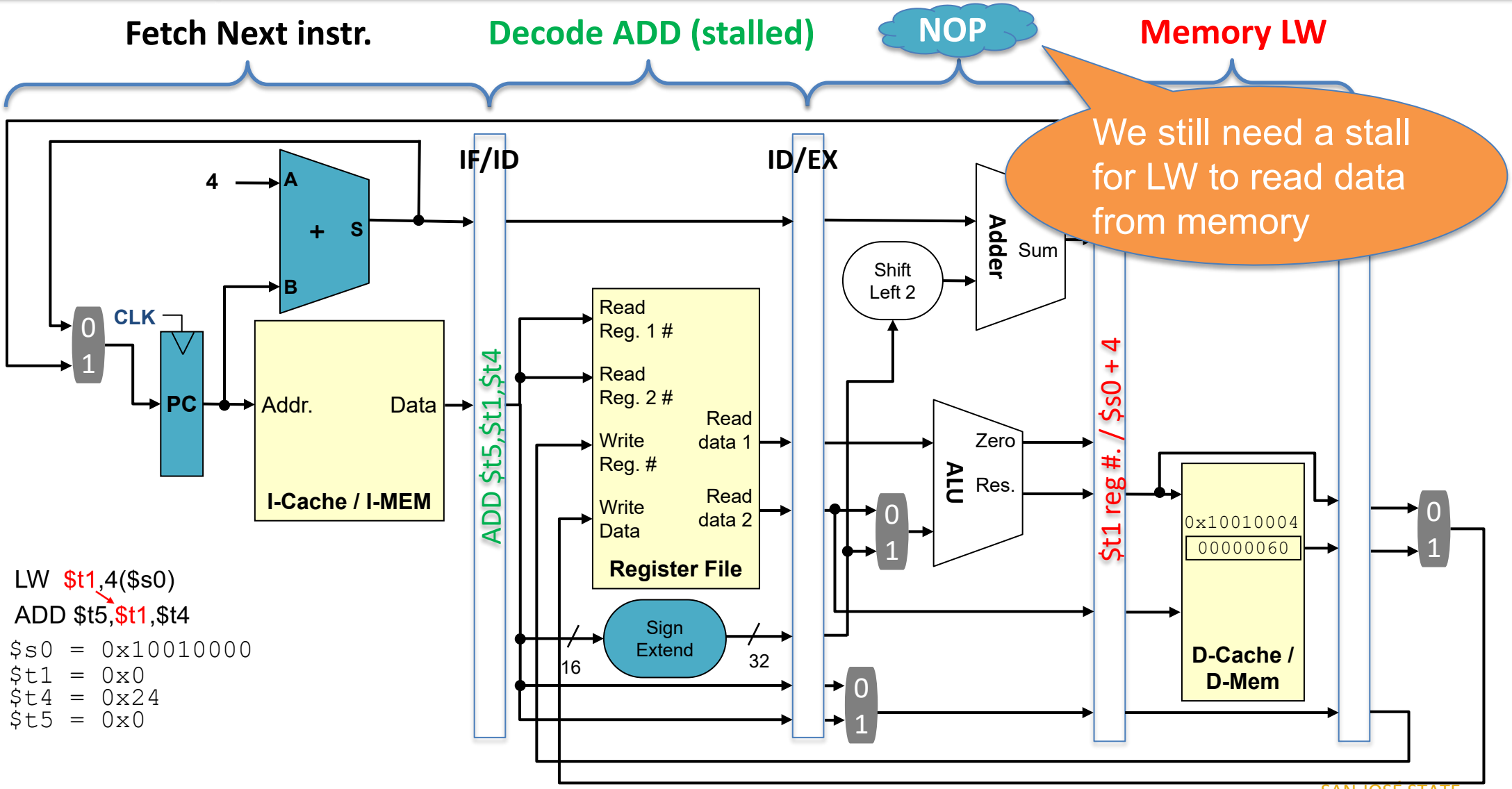
Data Forwarding

- **Also known as “bypassing”**
- **Take results still in the pipeline (not yet written back to a register) and pass them to dependent instructions**
- **Forwarding Path**
  - Load instruction: from WB to EXE
  - R-type instructions: from MEM to EXE

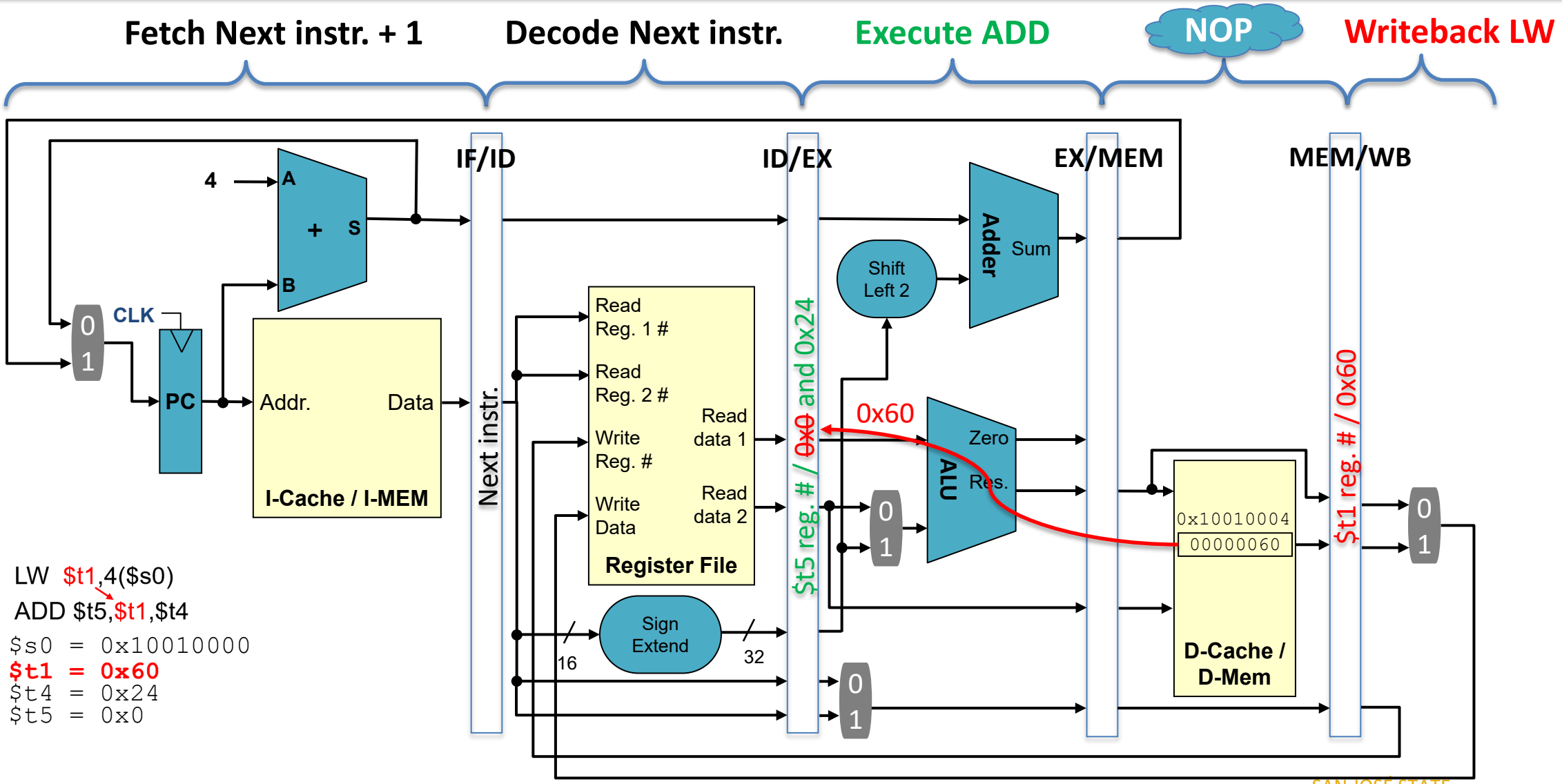
Data Forwarding



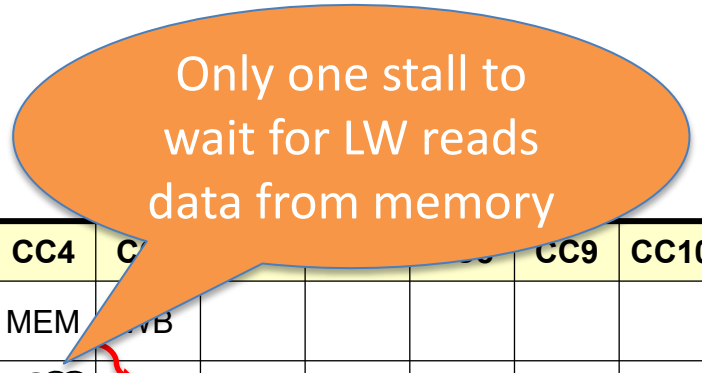
Data Forwarding



Data Forwarding



Data Forwarding



Saved 1 cycles compared to w/o forwarding

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	CC12	CC14
LW	IF	ID	EXE	MEM	WB								
ADD		IF	ID	nop	EXE	MEM	WB						
Next instr.			IF	nop	ID	EXE	MEM	WB					
Next+1					IF	ID	EXE	MEM	WB				
Next+2						IF	ID	EXE	MEM	WB			
Next+3							IF	ID	EXE	MEM	WB		
Next+4								IF	ID	EXE	MEM	WB	

Data Forwarding

- **As far as earlier instruction's result is in pipeline, the result value can be forwarded to the following instructions**

- **In arithmetic operations, no stall is needed**

ADD $t3,$t1,$t2
SUB $t5,$t3,$t4
XOR $t7,$t5,$t3

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
ADD	IF	ID	EXE	MEM	WB					
SUB		IF	ID	EXE	MEM	WB				
XOR			IF	ID	EXE	MEM	WB			
Next+1				IF	ID	EXE	MEM	WB		
Next+2					IF	ID	EXE	MEM	WB	
Next+3						IF	ID	EXE	MEM	WB

Forwarding Logic

- **Forwarding from MEM**
  - if (**we_regM** and (**rsE** ≠ 0) and (**rf_waM** = **rsE**))
    ForwardAE = 10
  - if (**we_regM** and (**rtE** ≠ 0) and (**rf_waM** = **rtE**))
    ForwardBE = 10

rsE & rtE
: source register id in EXE stage
rf_waM
: destination register id in MEM stage
we_regM
: register file write enable in MEM stage

ADD $t3,$t1,$t2
SUB $t5,$t3,$t4
XOR $t7,$t5,$t3

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
ADD	IF	ID	EXE	MEM	WB					
SUB		IF	ID	EXE	MEM	WB				
XOR			IF	ID	EXE	MEM	WB			
Next+1				IF	ID	EXE	MEM	WB		
Next+2					IF	ID	EXE	MEM	WB	
Next+3						IF	ID	EXE	MEM	WB

Forwarding Logic

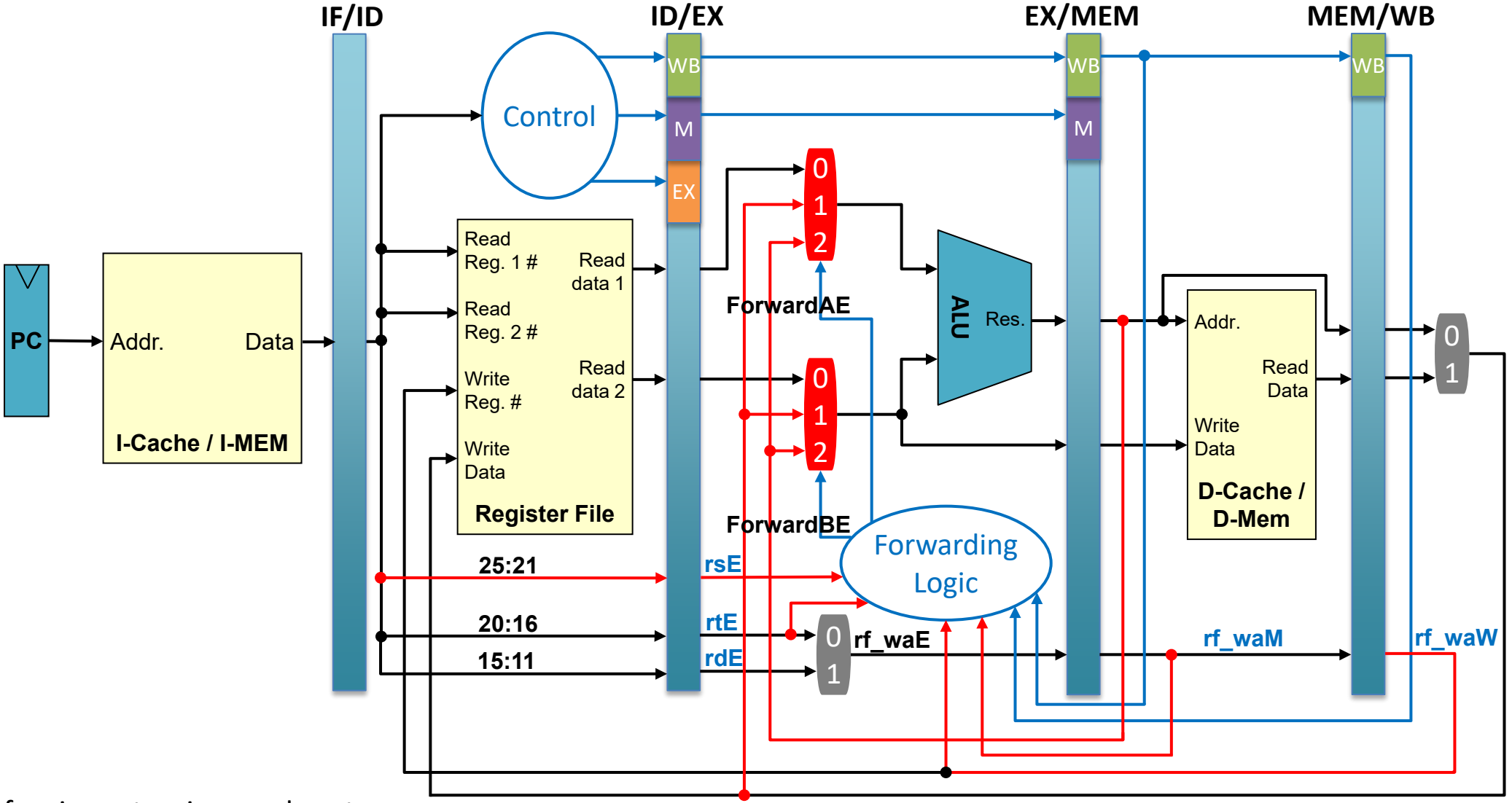
- **Forwarding from WB**
  - if (**we_regW** and (**rsE** ≠ 0) and (**rf_waW** = **rsE**))
    ForwardAE = 01
  - if (**we_regW** and (**rtE** ≠ 0) and (**rf_waW** = **rtE**))
    ForwardBE = 01

rf_waW
: destination register id in WB stage
we_regW
: register file write enable in WB stage

ADD $t3,$t1,$t2
SUB $t5,$t3,$t4
XOR $t7,$t5,$t3

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
ADD	IF	ID	EXE	MEM	WB					
SUB		IF	ID	EXE	MEM	WB				
XOR			IF	ID	EXE	MEM	WB			
Next+1				IF	ID	EXE	MEM	WB		
Next+2					IF	ID	EXE	MEM	WB	
Next+3						IF	ID	EXE	MEM	WB

Forwarding Path



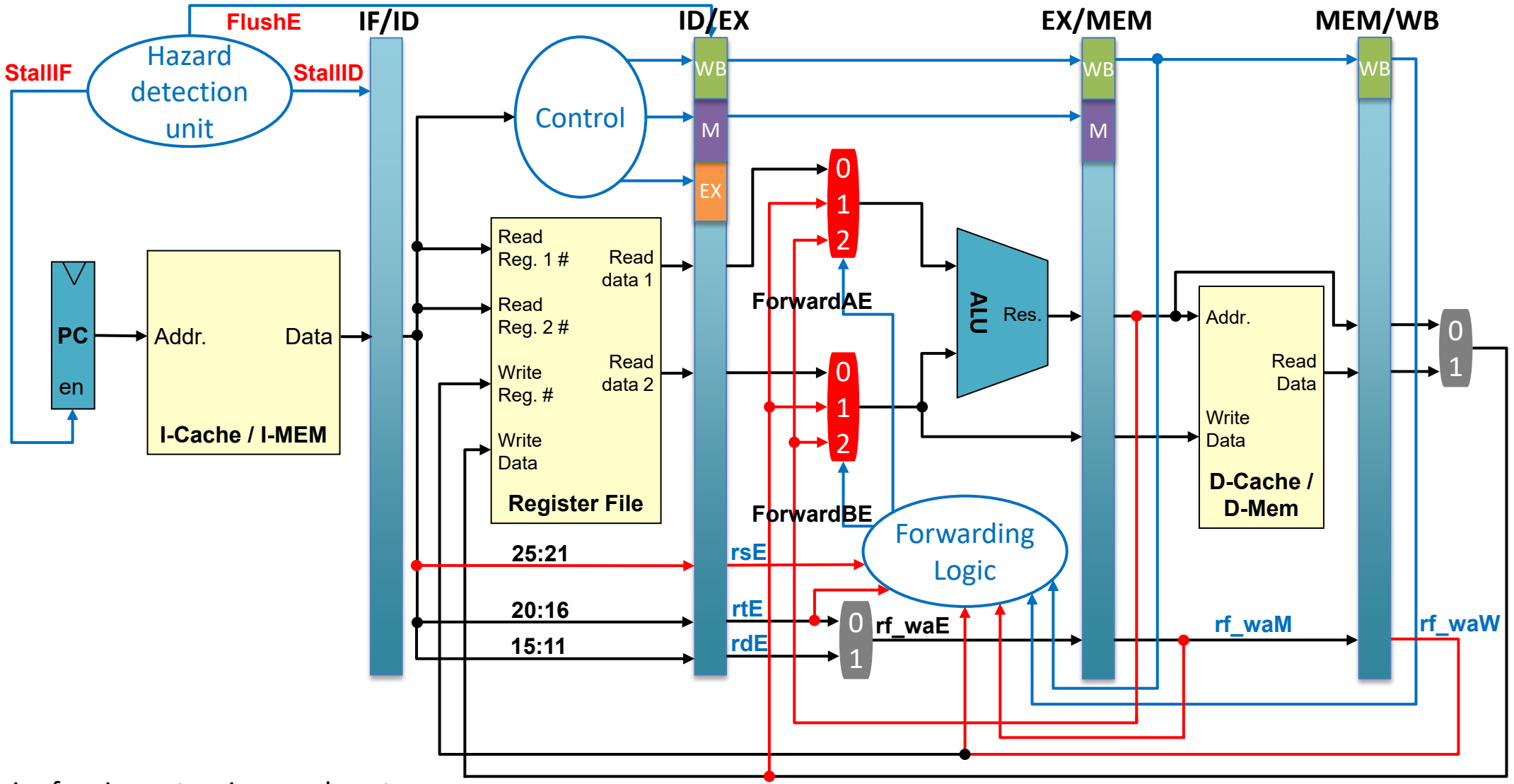
Note: Logics for sign extensions and next pc calculation are omitted for simplicity

Stall With Forwarding

- **For LW sourced dependency, we still need to stall for one cycle**
  - Freeze the PC and IF/ID pipeline register for one clock cycle
  - Flush the ID/EXE pipeline register

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	CC12	CC14
LW	IF	ID	EXE	MEM	WB								
ADD		IF	ID	nop	EXE	MEM	WB						
Next instr.			IF	nop	ID	EXE	MEM	WB					
Next+1					IF	ID	EXE	MEM	WB				
Next+2						IF	ID	EXE	MEM	WB			
Next+3							IF	ID	EXE	MEM	WB		
Next+4								IF	ID	EXE	MEM	WB	

Forwarding Path



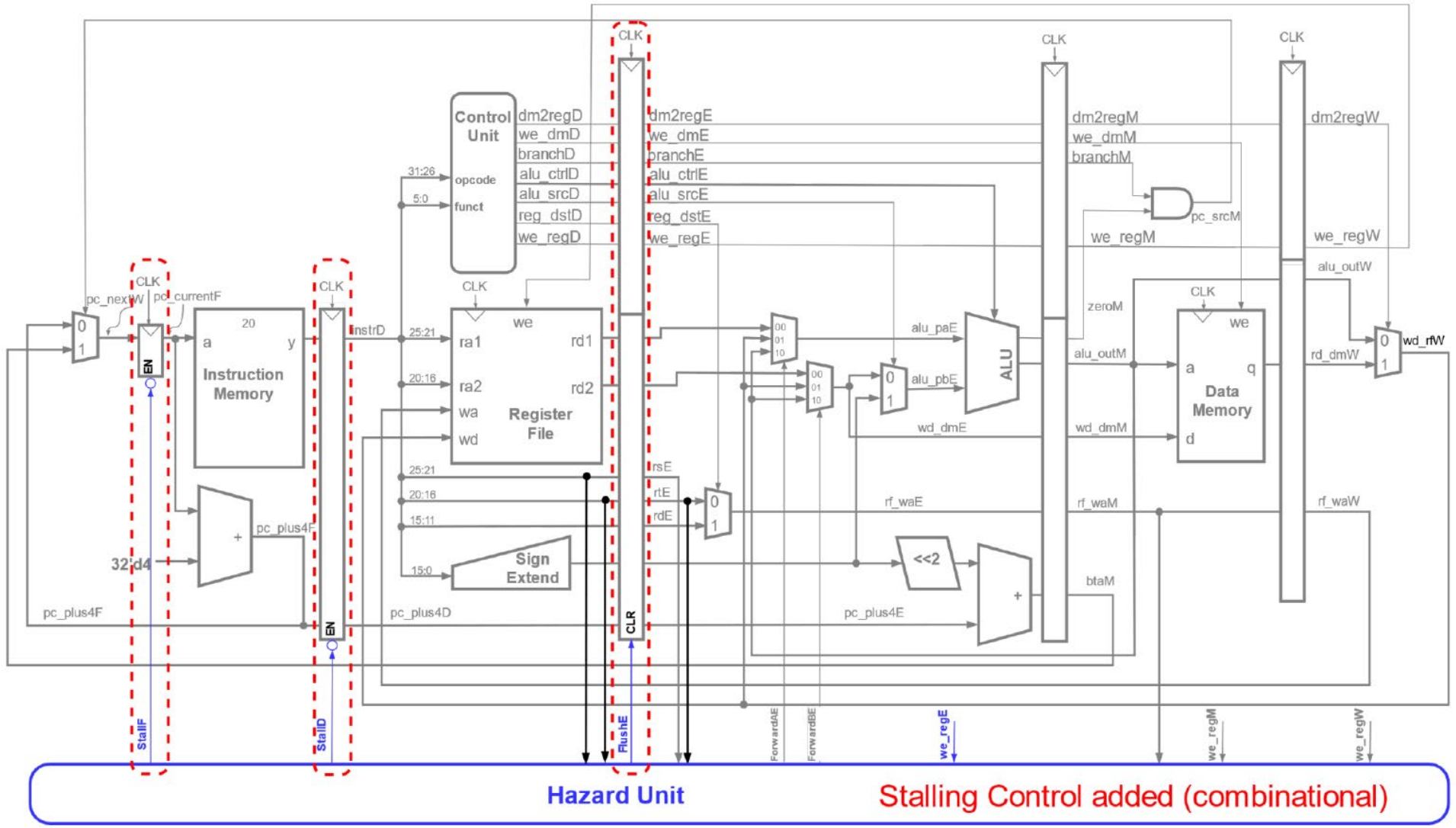
Note: Logics for sign extensions and next pc calculation are omitted for simplicity

Stalling Condition

- **Stalling Logic**
  - `lwstall` = ((`rsD` == `rtE`) or (`rtD` == `rtE`)) and `dm2regE`
  - `stallF` = `stallD` = `FlushE` = `lwstall`

`rsD` & **`rtD`**
: source register id in ID stage
`rtE`
: destination register in EXE stage
`dm2regE`
: data memory to register write signal of the instruction in EXE stage

Complete Datapath



Pipeline Latency with Forwarding/Stall

- **Number of cycles we must stall to execute a dependent instruction:**

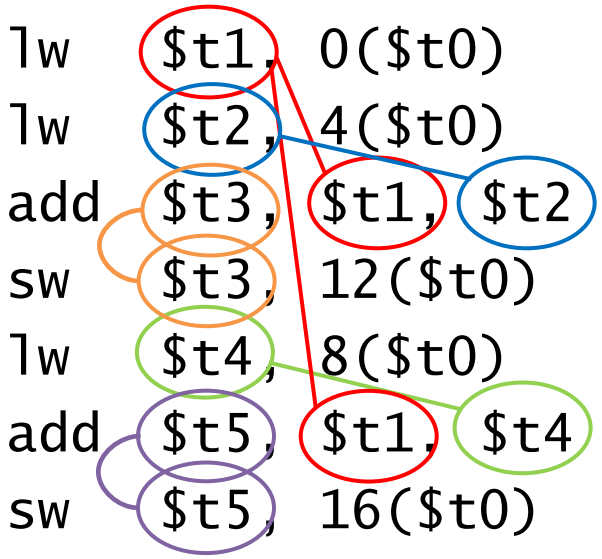
Instruction that is the dependency source	w/o Forwarding (cycles)	w/ Forwarding (cycles)
LW	2	1
Other instructions that update register file	2	0

- **Data Hazards Solutions:**
  - Hardware: stalling & forwarding

  - Software: compiler -- code reordering

Code Scheduling to Avoid Stalls

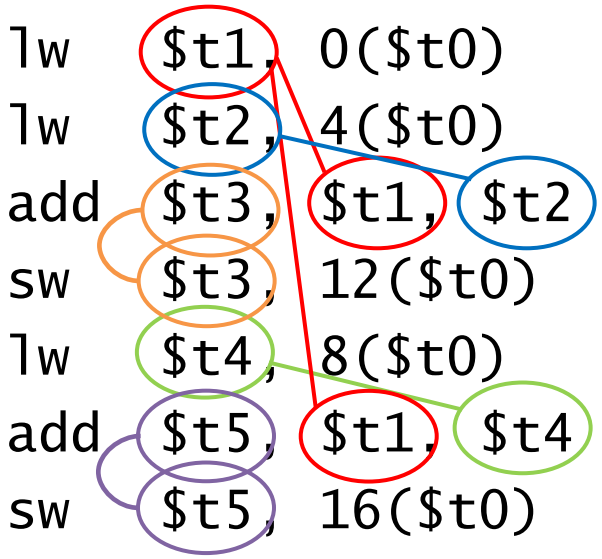
- **Compiler can reorder code to avoid the stalls**
- **C code for `A = B + E; C = B + F;`**



Find RAW dependencies and check if stall is needed

Code Scheduling to Avoid Stalls

- **Compiler can reorder code to avoid the stalls**
- **C code for `A = B + E; C = B + F;`**



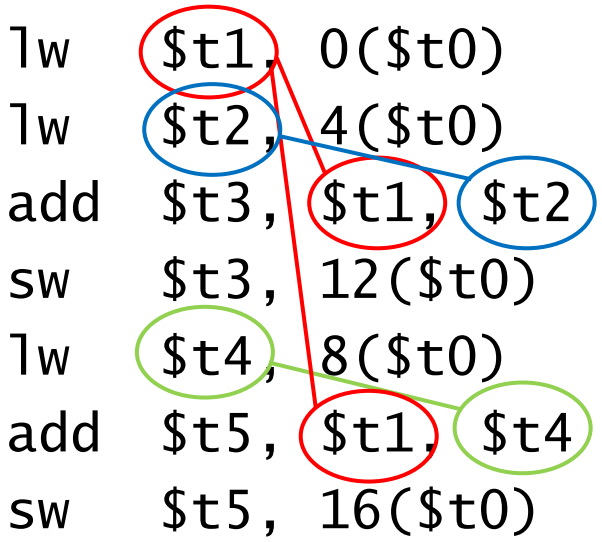
Find RAW dependencies and check if stall is needed

When forwarding is used, only lw instruction causes stall
→ Leave lw instructions only

Code Scheduling to Avoid Stalls

- **Compiler can reorder code to avoid the stalls**
- **C code for `A = B + E; C = B + F;`**

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```



Find RAW dependencies and check if stall is needed

lw instructions cause 1-cycle stalls
→ Leave instructions that are executed back-to-back only

Code Scheduling to Avoid Stalls

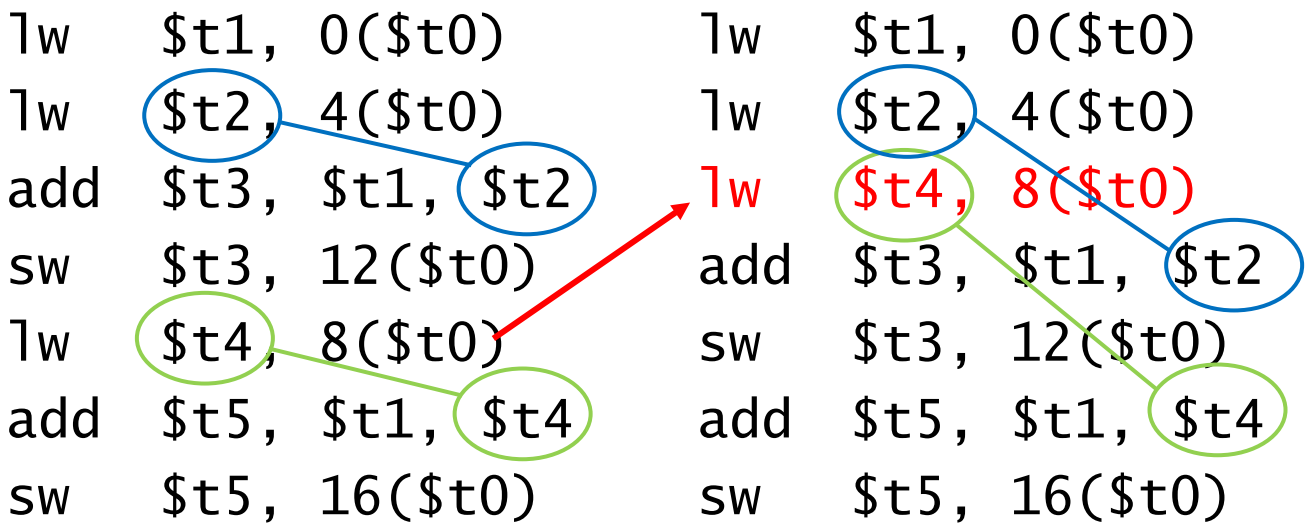
- **Compiler can reorder code to avoid the stalls**
- **C code for `A = B + E; C = B + F;`**

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

Now, check if the lw and dependent instructions **must be** executed back-to-back

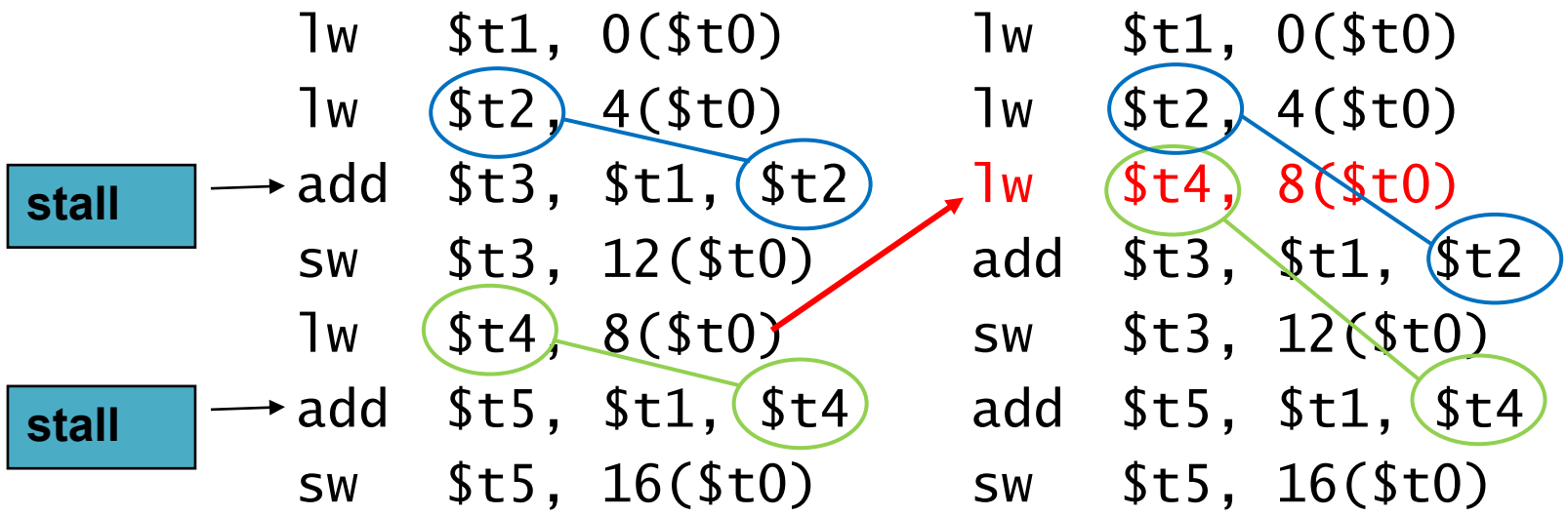
Code Scheduling to Avoid Stalls

- **Compiler can reorder code to avoid the stalls**
- **C code for `A = B + E; C = B + F;`**



Code Scheduling to Avoid Stalls

- **Compiler can reorder code to avoid the stalls**
- **C code for `A = B + E; C = B + F;`**



Total execution time

5 cycles for the first inst +
6 cycles for the remaining insts +
2 x 1 stall cycles = 13 cycles

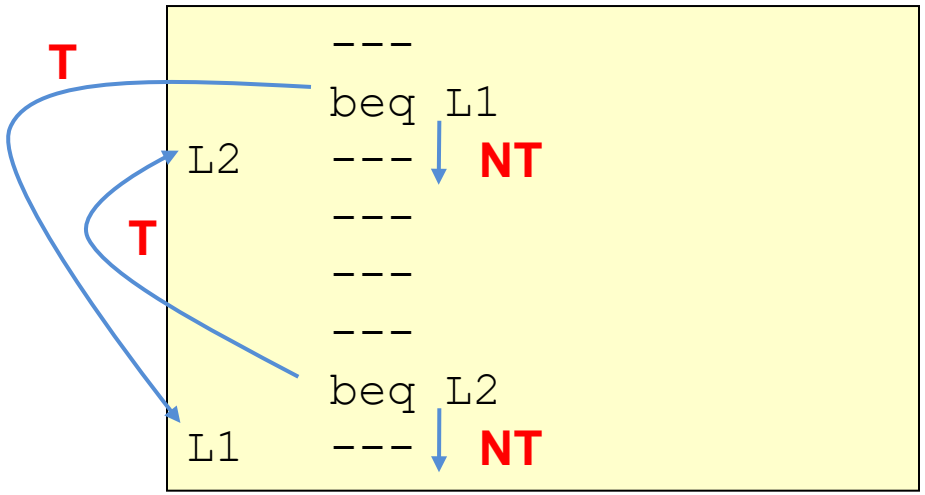
5 cycles for the first inst +
6 cycles for the remaining insts +
0 stall cycles = 11 cycles

Control Hazard

- **Branch outcomes: Taken (T) or Not-Taken (NT)**

- **Not known until late in the pipeline**
  - Prevents us from fetching future instructions
  - Rather than stall, we can predict the outcome and keep fetching
  - We only need to correct the pipeline if we guess wrong

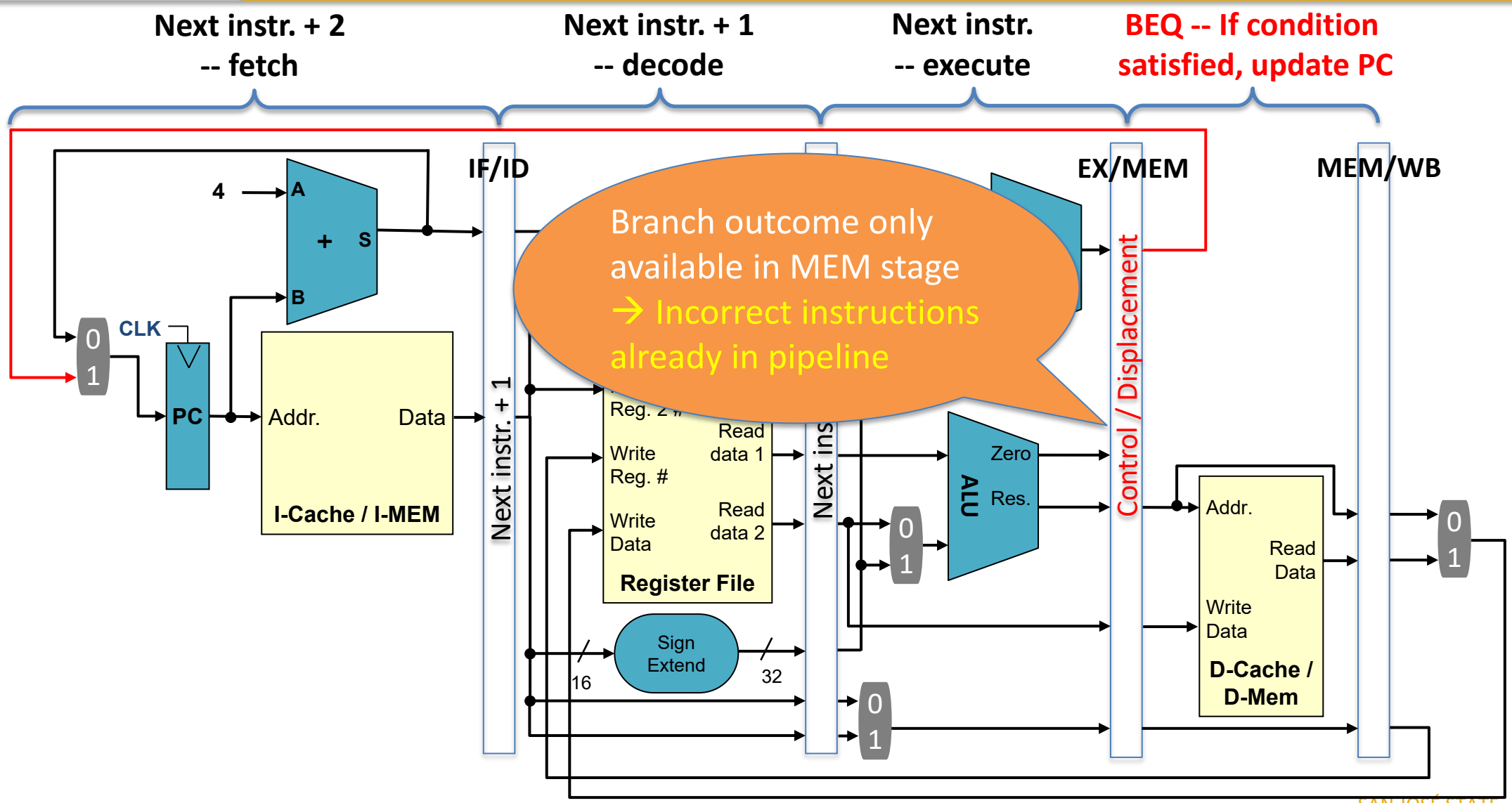
- **Static Predictions**
  - Predict Taken
  - Predict Not Taken



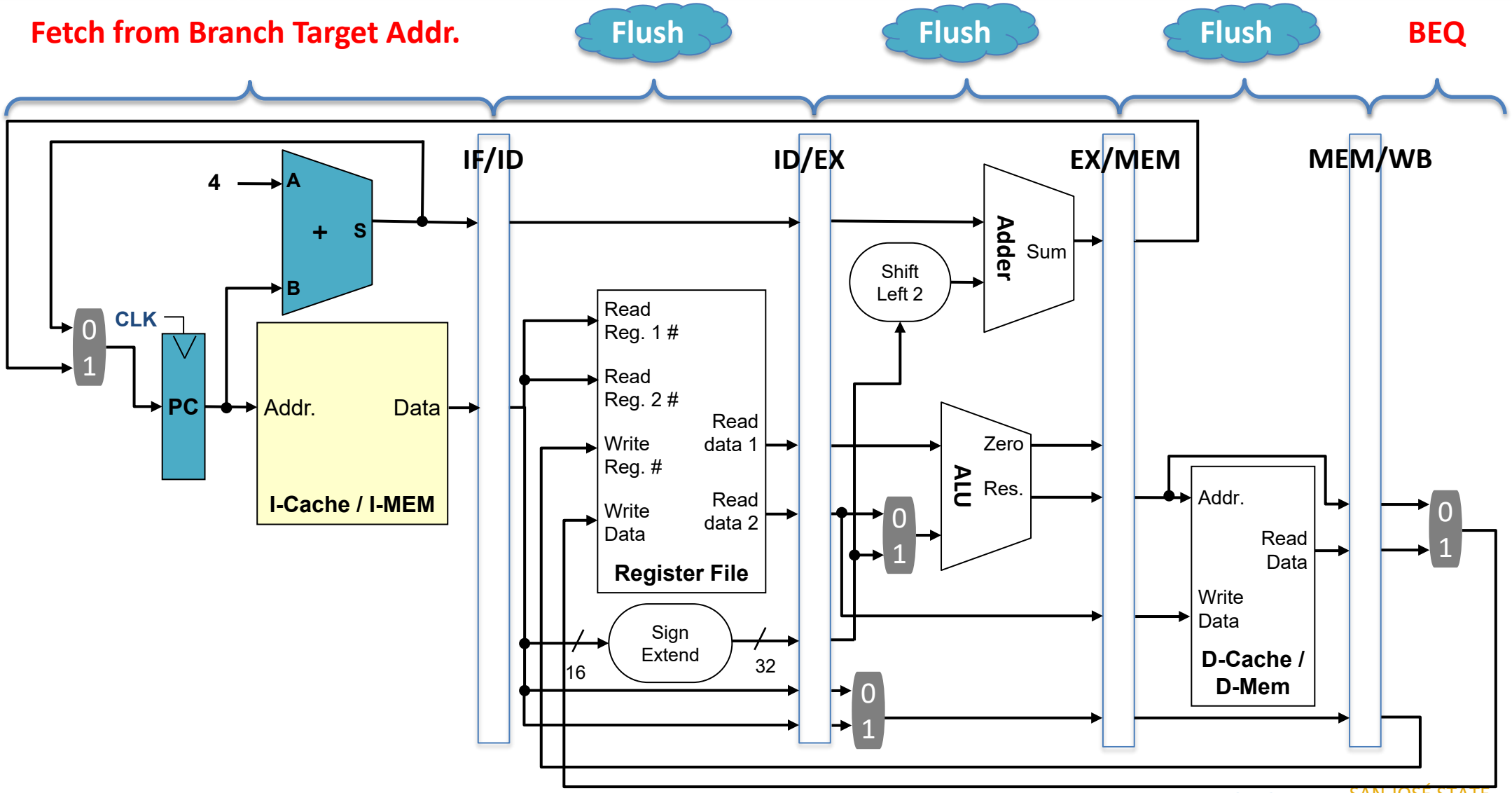
Statically Predict Not Taken

- **Keep fetching instructions from the PC + 4 by assuming that the branch will be not taken**
- **Once branch is turned out to be taken, flush the incorrectly fetched instructions**
- **In the following cycle, fetch the instruction from the branch target address**

Statically Predict Not Taken: Issues



Statically Predict Not Taken: Issues



Static Branch Prediction Penalty

- **Penalty = number of instructions that need to be flushed on misprediction**

- **For static branch prediction:**
  - The branch outcome and target address is available at the MEM stage and passed back to the Fetch stage
  - If mispredicted, instructions of the correct path will be fetched on the next cycle

- **A 3-cycle branch penalty when mispredicted**

Statically Predict Not Taken

Actual Branch Outcome

```
     BEQ  $a0,$a1,L1  (NT)
L2:  ADD  $s1,$t1,$t2
     SUB  $t3,$t0,$s0
     OR   $s0,$t6,$t7
     BNE  $a0,$s1,L2  (T)
L1:  AND  $t3,$t6,$t7
     SW   $t5,0($s1)
     LW   $s2,0($s5)
```

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
BEQ	IF	ID	EXE	MEM						
ADD		IF	ID	EXE						
SUB			IF	ID						
OR				IF						

NT!

Statically Predict Not Taken

Actual Branch Outcome

```
    BEQ  $a0,$a1,L1  (NT)
L2: ADD  $s1,$t1,$t2
    SUB  $t3,$t0,$s0
    OR   $s0,$t6,$t7
    BNE  $a0,$s1,L2  (T)
L1: AND  $t3,$t6,$t7
    SW   $t5,0($s1)
    LW   $s2,0($s5)
```

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
BEQ	IF	ID	EXE	MEM	WB					
ADD		IF	ID	EXE	MEM					
SUB			IF	ID	EXE					
OR				IF	ID					
BNE					IF					

Statically Predict Not Taken

Actual Branch Outcome

```
    BEQ  $a0,$a1,L1  (NT)
L2: ADD  $s1,$t1,$t2
    SUB  $t3,$t0,$s0
    OR   $s0,$t6,$t7
    BNE  $a0,$s1,L2  (T)
L1: AND  $t3,$t6,$t7
    SW   $t5,0($s1)
    LW   $s2,0($s5)
```

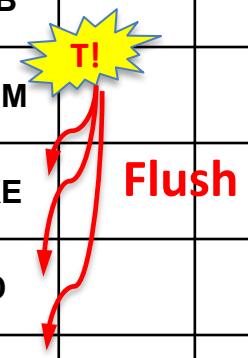
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
BEQ	IF	ID	EXE	MEM	WB					
ADD		IF	ID	EXE	MEM	WB				
SUB			IF	ID	EXE	MEM				
OR				IF	ID	EXE				
BNE					IF	ID				
AND						IF				

Statically Predict Not Taken

Actual Branch Outcome

```
     BEQ  $a0,$a1,L1  (NT)
L2:  ADD  $s1,$t1,$t2
     SUB  $t3,$t0,$s0
     OR   $s0,$t6,$t7
     BNE  $a0,$s1,L2  (T)
L1:  AND  $t3,$t6,$t7
     SW   $t5,0($s1)
     LW   $s2,0($s5)
```

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
BEQ	IF	ID	EXE	MEM	WB					
ADD		IF	ID	EXE	MEM	WB				
SUB			IF	ID	EXE	MEM	WB			
OR				IF	ID	EXE	MEM	WB		
BNE					IF	ID	EXE	MEM		
AND						IF	ID	EXE		
SW							IF	ID		
LW								IF		



Statically Predict Not Taken

Actual Branch Outcome

```
    BEQ  $a0,$a1,L1  (NT)
L2: ADD  $s1,$t1,$t2
    SUB  $t3,$t0,$s0
    OR   $s0,$t6,$t7
    BNE  $a0,$s1,L2  (T)
L1: AND  $t3,$t6,$t7
    SW   $t5,0($s1)
    LW   $s2,0($s5)
```

Instruction in the target address (L2)

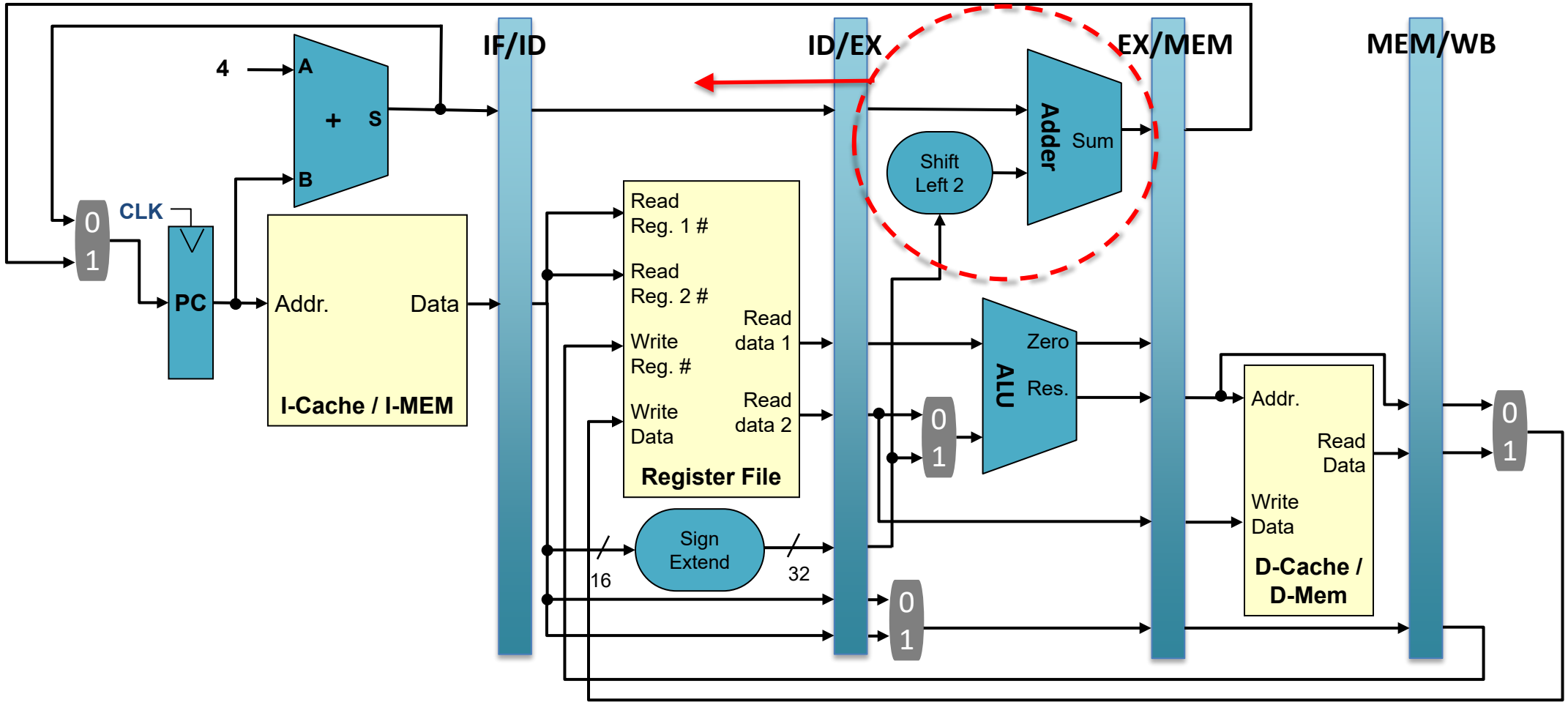
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
BEQ	IF	ID	EXE	MEM	WB					
ADD		IF	ID	EXE	MEM	WB				
SUB			IF	ID	EXE	MEM	WB			
OR				IF	ID	EXE	MEM	WB		
BNE					IF	ID	EXE	MEM	WB	
AND						IF	ID	EXE	nop	nop
SW							IF	ID	nop	nop
LW								IF	nop	nop
ADD									IF	ID
SUB										IF

Predict Taken?

- **It is not possible to statically predict all the branches to be taken**
  - The branch target address is not computed until the EX stage, so we don't have the PC values even needed to predict taken
- **Dynamic branch prediction + branch target buffer enables to predict taken**
  - Predict untaken a few times to collect branch target address of each branch and then predict taken based on the history
- **Can we reduce branch penalty with change of datapath?**

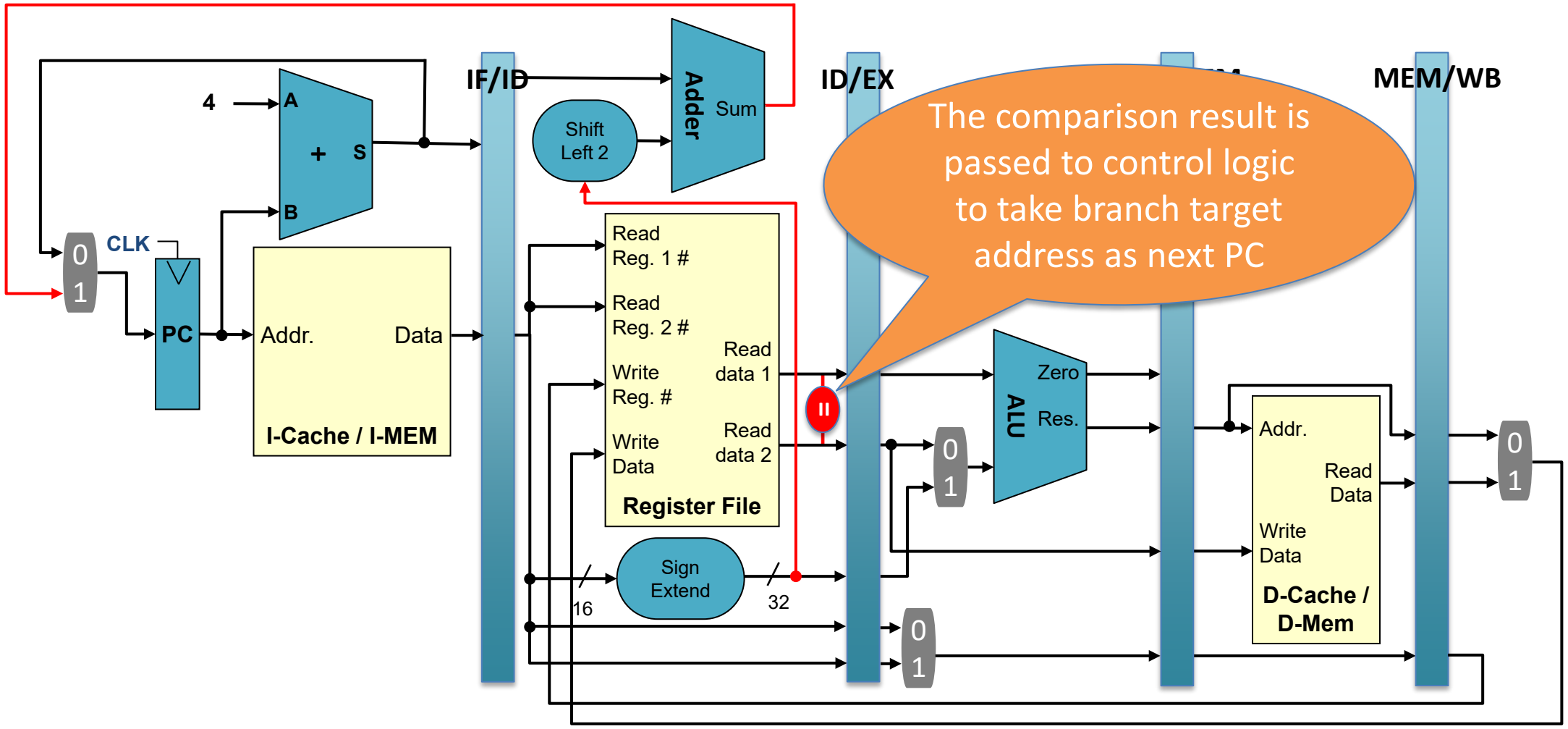
Early Branch Determination

- **Target address can be calculated earlier by moving shift-left-2 and Adder**



Early Branch Determination

- **Target address can be calculated earlier by moving shift-left-2 and Adder**



Early Determination w/ Predict NT

Actual Branch Outcome

```
    BEQ  $a0,$a1,L1  (NT)
L2: ADD  $s1,$t1,$t2
    SUB  $t3,$t0,$s0
    OR   $s0,$t6,$t7
    BNE  $a0,$s1,L2  (T)
L1: AND  $t3,$t6,$t7
    SW   $t5,0($s1)
    LW   $s2,0($s5)
```

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
BEQ	IF	ID	EXE	MEM	WB					
ADD		IF	ID	EXE	MEM	WB				
SUB			IF	ID	EXE	MEM				
OR				IF	ID	EXE				
BNE					IF	ID				
AND						IF				

T!

Flush

Early Determination w/ Predict NT

Actual Branch Outcome

```
    BEQ  $a0,$a1,L1  (NT)
L2: ADD  $s1,$t1,$t2
    SUB  $t3,$t0,$s0
    OR   $s0,$t6,$t7
    BNE  $a0,$s1,L2  (T)
L1: AND  $t3,$t6,$t7
    SW   $t5,0($s1)
    LW   $s2,0($s5)
```

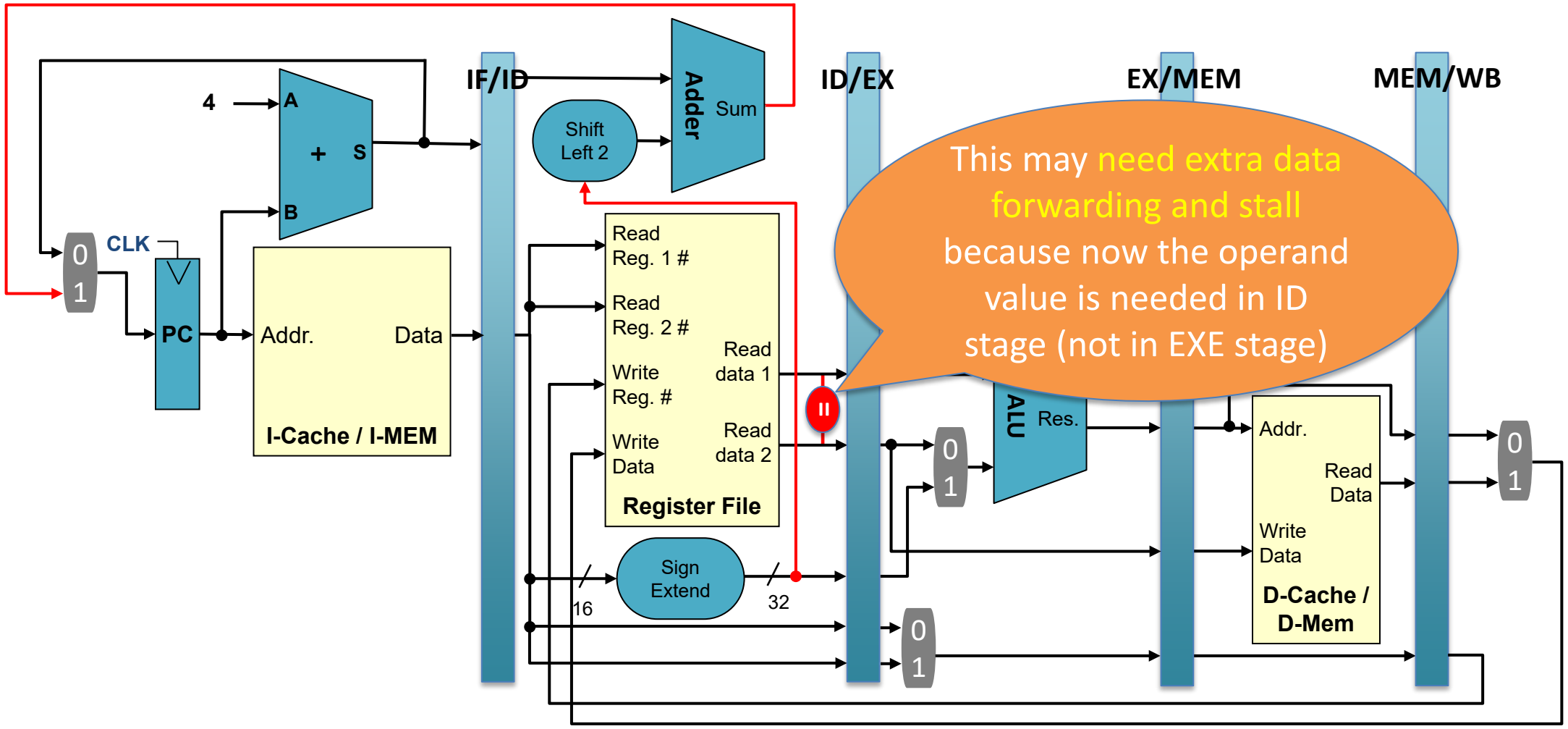
Instruction in the target address (L2)

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
BEQ	IF	ID	EXE	MEM	WB					
ADD		IF	ID	EXE	MEM	WB				
SUB			IF	ID	EXE	MEM	WB			
OR				IF	ID	EXE	MEM	WB		
BNE					IF	ID	T! XE	MEM	WB	
AND						IF	nop	nop	nop	nop
ADD							IF	ID	EXE	MEM
SUB								IF	ID	EXE

Flush

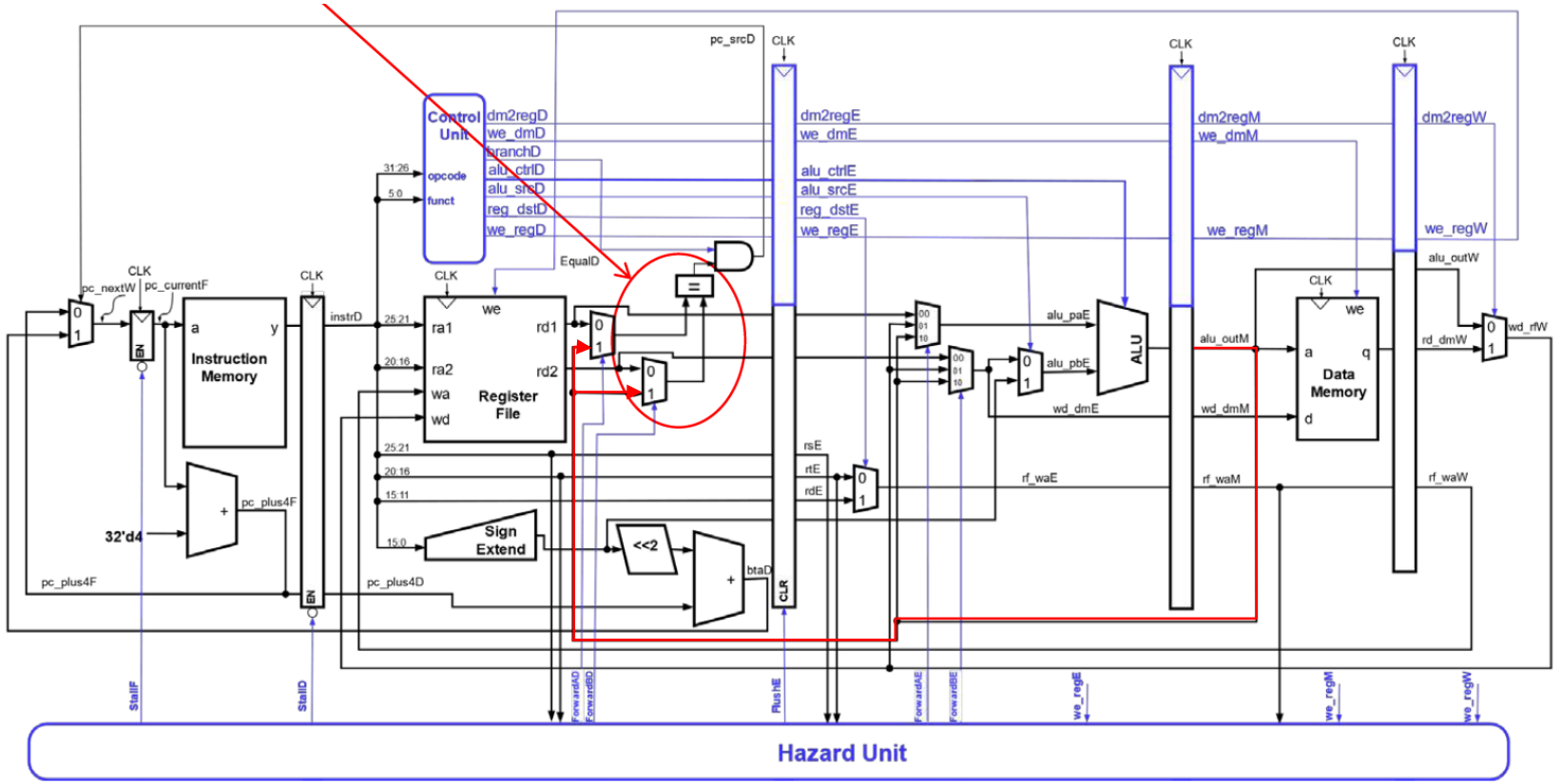
Early Branch Determination: Issues

- **Target address can be calculated earlier by moving shift-left-2 and Adder**



Early Branch Determination

- **Forwarding logic for early branch determination**
  - **`ForwardAD = branchD`** and **`we_regM`** and (**`rsD`** ≠ 0) and (**`rf_waM`** == **`rsD`**)
  - **`ForwardBD = branchD`** and **`we_regM`** and (**`rtD`** ≠ 0) and (**`rf_waM`** == **`rtD`**)



Early Determination w/ Predict NT

Actual Branch Outcome

```
     BEQ  $a0,$a1,L1  (NT)
L2:  ADD  $s1,$t1,$t2
     SUB  $t3,$t0,$s0
     OR   $s0,$t6,$t7
     BNE  $s0,$s1,L2  (T)
L1:  AND  $t3,$t6,$t7
     SW   $t5,0($s1)
     LW   $s2,0($s5)
```

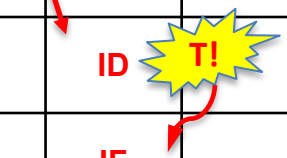
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
BEQ	IF	ID	EXE	MEM	WB					
ADD		IF	ID	EXE	MEM	WB				
SUB			IF	ID	EXE	MEM				
OR				IF	ID	EXE				
BNE					IF	ID				
AND						IF				

Early Determination w/ Predict NT

Actual Branch Outcome

```
    BEQ  $a0,$a1,L1  (NT)
L2: ADD  $s1,$t1,$t2
    SUB  $t3,$t0,$s0
    OR   $s0,$t6,$t7
    BNE  $s0,$s1,L2  (T)
L1: AND  $t3,$t6,$t7
    SW   $t5,0($s1)
    LW   $s2,0($s5)
```

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
BEQ	IF	ID	EXE	MEM	WB					
ADD		IF	ID	EXE	MEM	WB				
SUB			IF	ID	EXE	MEM	WB			
OR				IF	ID	EXE	MEM			
BNE					IF	ID	ID			
AND						IF	IF			



Early Determination w/ Predict NT

Actual Branch Outcome

```
    BEQ  $a0,$a1,L1  (NT)
L2: ADD  $s1,$t1,$t2
    SUB  $t3,$t0,$s0
    OR   $s0,$t6,$t7
    BNE  $s0,$s1,L2  (T)
L1: AND  $t3,$t6,$t7
    SW   $t5,0($s1)
    LW   $s2,0($s5)
```

Instruction in the target address (L2)

	CC1	CC2	CC3						
BEQ	IF	ID	EXE						
ADD		IF	ID						
SUB			IF	ID	EXE				
OR				IF	ID	EXE		WB	
BNE					IF	ID	ID	T!	MEM	WB
AND						IF	IF	nop	nop	nop
ADD								IF	ID	EXE
SUB									IF	ID

If BNE has dependency with its preceding instruction, OR, one cycle stall is required to get the OR's result value
→ Still better than normal 3 cycle flushing

Early Branch Determination

- **Stalling logic for early branch determination**
  - `branchstall` =

    `branchD` and `we_regE` and ((`rf_waE` == `rsD`) or (`rf_waE` == `rtD`))

    or

    `branchD` and `dm2regM` and ((`rf_waM` == `rsD`) or (`rf_waM` == `rtD`))

  - `StallF = StallD = FlushE = lwstall OR branchstall`

Branch Delay Slot

Actual Branch Outcome

```
    BEQ  $a0,$a1,L1  (NT)
L2: ADD  $s1,$t1,$t2
    SUB  $t3,$t0,$s0
    OR   $s0,$t6,$t7
    BNE  $a0,$s1,L2  (T)
L1: AND  $t3,$t6,$t7
    SW   $t5,0($s1)
    LW   $s2,0($s5)
```

Instruction in the target address (L2)

Find an instruction that
1) has to be executed regardless branch outcome
2) independent to branch

The instruction will be placed in the branch delay slot (right after branch instruction) instead of fetching wrong instruction and flushing

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
				MEM	WB					
				EXE	MEM	WB				
				IF	ID	EXE	MEM	WB		
BNE					IF	ID	T! EXE	MEM	WB	
						IF	Flush nop	nop	nop	nop
							IF	ID	EXE	MEM
								IF	ID	EXE

Branch Delay Slot

Actual Branch Outcome

```
    BEQ  $a0,$a1,L1  (NT)
L2: ADD  $s1,$t1,$t2
    SUB  $t3,$t0,$s0
    BNE  $a0,$s1,L2  (T)
    OR   $s0,$t6,$t7
L1: AND  $t3,$t6,$t7
    SW   $t5,0($s1)
    LW   $s2,0($s5)
```

Instruction in the target address (L2)

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
BEQ	IF	ID	EXE	MEM	WB					
ADD		IF	ID	EXE	ME					
SUB			IF	ID	EXE	MEM	WB			
BNE				IF	ID	EXE	MEM	WB		
OR					IF	ID	EXE	MEM	WB	
ADD						IF	ID	EXE	MEM	WB
SUB							IF	ID	EXE	MEM

No Flush & Better Performance

Branch Delay Slot

- **Branch delay slot is more useful for unconditional branches such as `j`, `jal`, `jr` because they are always taken**

- **For example, if `jal` uses branch delay slot,**
  - After `jal`, `$ra` is updated with PC + 8
  - The instruction in PC + 4 is executed after jal before entering the subfunction

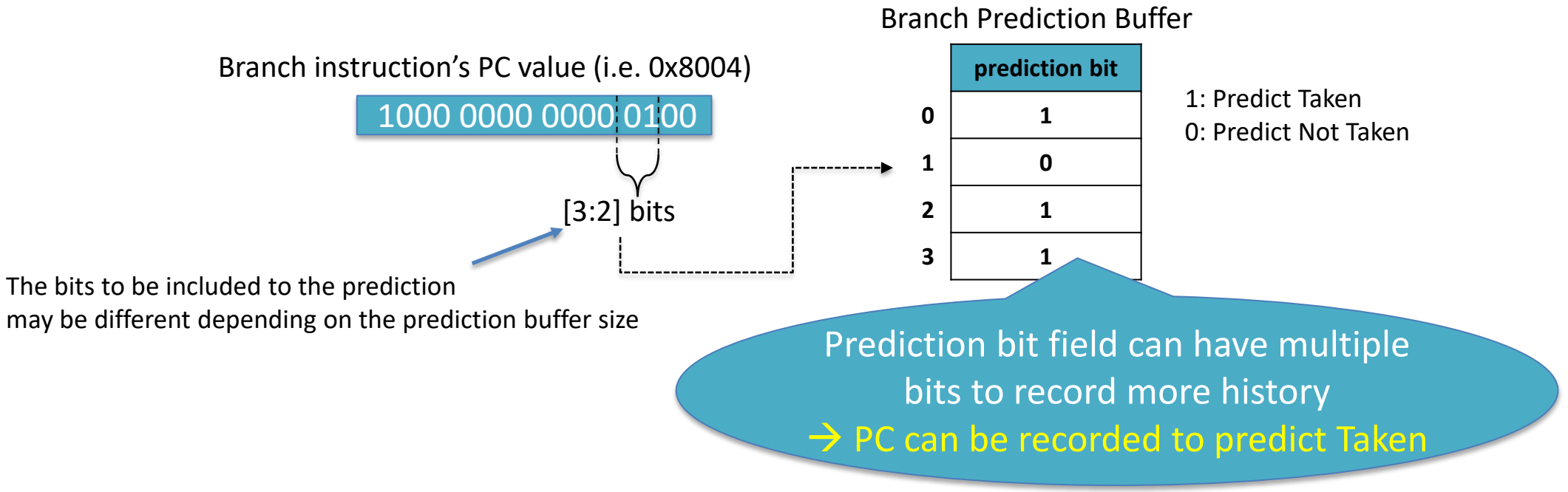
Dynamic Branch Prediction

- **Monitor the branch patterns for better prediction**
  - i.e. loops usually have a few iterations, so the branch in the loop is likely to be not taken at least for a few times

- **Branch Prediction Buffer (branch history buffer) is used to maintain each branch's outcome history**
  - Small storage space indexed with certain bits of branch instruction's PC value
  - Each entry maintains the previous outcome of each branch
  - Once the branch outcome is available, the corresponding entry is updated

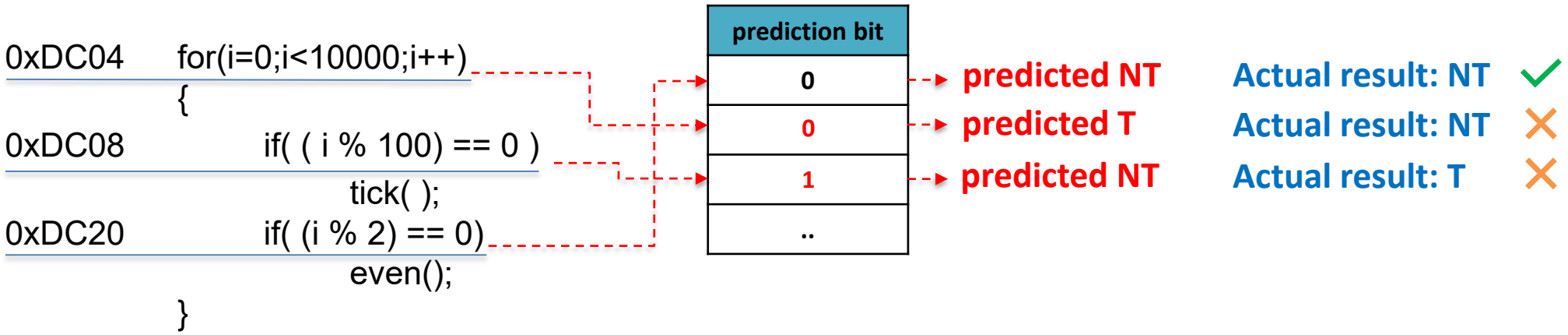
Branch Prediction Buffer

- **Aliasing issue**
  - Different branch instructions may map to the same branch prediction buffer entry if their PC values have the same LSBs, affecting each others' predictions
    - E.g., 0x8004 and 0x8104 will access the same entry if [3:2] bits are used for indexing
    - Inevitable but can be alleviated with larger buffer



1-bit Predictor

- **Each entry of branch prediction buffer is 1-bit (1: Taken, 0: Not Taken)**
  - The entry value is the latest outcome of the branch
  - Next outcome of the branch is predicted based on current value
  - Example: Assume that the actual outcomes of the branches at **0xDC04**, **0xDC08**, and **0xDC20** are **untaken**, **taken**, and **untaken**, respectively



Is 1 bit Enough?

```
0xDC04    for(i=0;i<10000;i++)
          {
0xDC08          if( ( i % 100) == 0 )
                    tick( );
0xDC20          if( (i % 2) == 0)
                    even();
          }
```

Not taken and continue to the loop body

Taken to exit the loop

NT TN

DC04: NNNNNNN ... NNNNNNNNNNTNNNNNNNNN ...

10,000 iterations 10,000 iterations ...

Mis-predictions for every first and last iterations → **99.998% Correct Prediction**

Mis-predictions: 2 / 10,000

DC08: TTTTT ... TNTTTTT ... TNTTTTT ...

98.0% Correct Prediction

Mis-predictions: 2 / 100

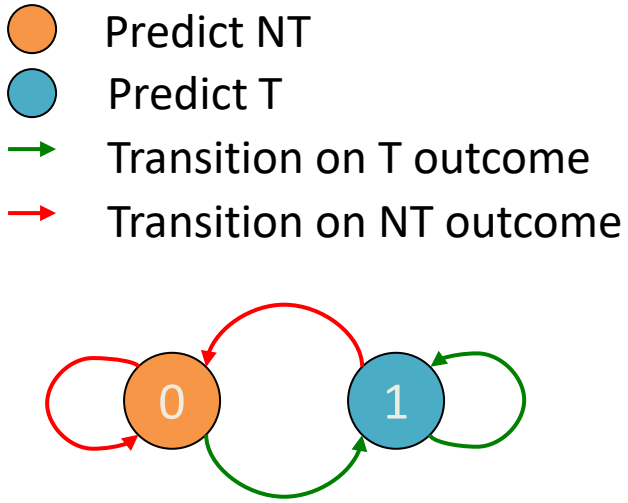
DC20: TNTNTNTNTNTNTNTNTNTNTNTNTNTNT ...

0.0% Correct Prediction

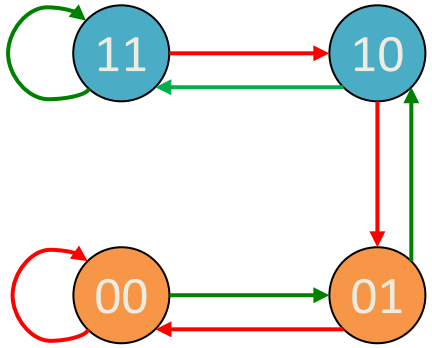
Mis-predictions: 2 / 2

Using 2-bit History

- **2-bit Saturating Counter in each branch prediction buffer entry**
  - Could have more than two bits but two bits cover most patterns (i.e. loops)

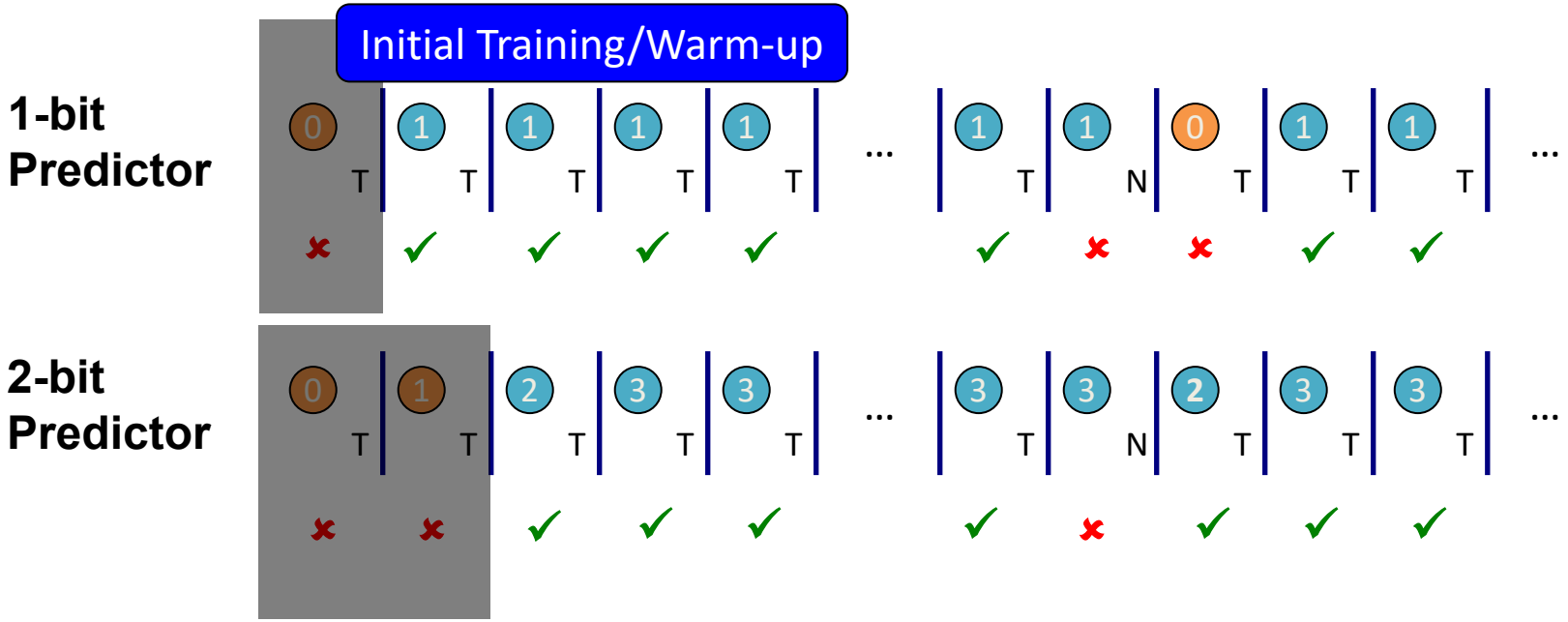


FSM for 1-bit Prediction



FSM for 2-bit Saturating Counter

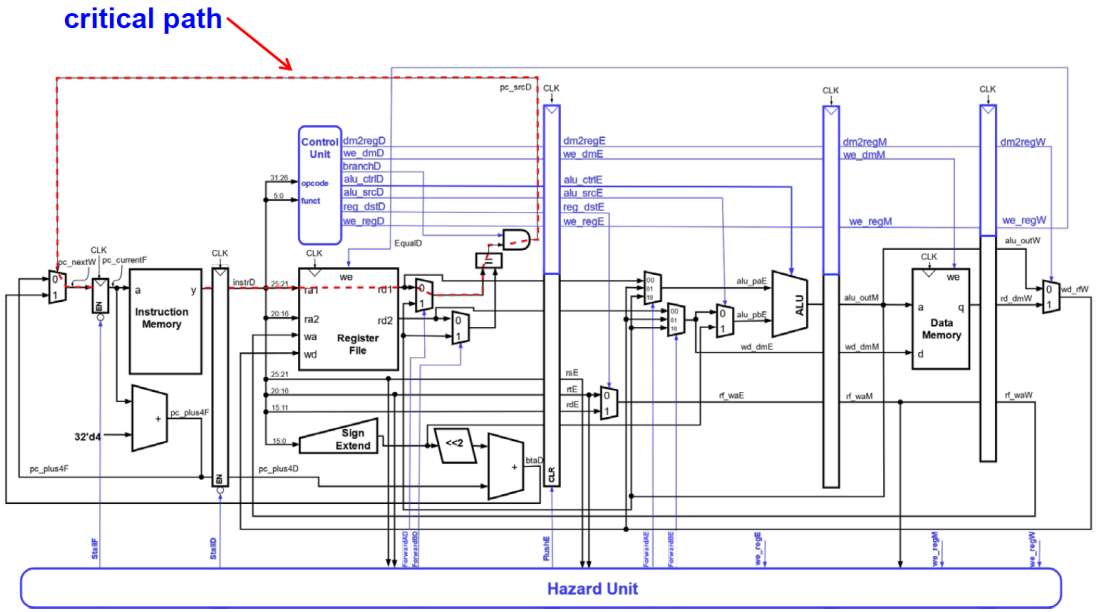
Example



Only 1 Misprediction per N branches now!
Correct prediction rates:
DC04: 99.999%, DC08: 99.0%, DC20: 50%

Pipelined CPU Performance Analysis

Function Unit	Parameter	Delay (ps)
Register clock-to-Q	T_{pcq}	30
Clock setup time	T_{setup}	20
MUX	T_{MUX}	25
Sign-extend	T_{s_ext}	25
ALU	T_{ALU}	200
Mem read	T_{mem}	250
AND	T_{AND}	15
EQ	T_{eq}	40
Register file read	T_{RFread}	150
Register file write	$T_{RFwrite}$	20



Critical path = max {

$T_{pcq} + T_{i\text{-}mem} + T_{setup}$,

$2\,(\,T_{RFread} + T_{mux} + T_{eq} + T_{AND} + T_{mux} + T_{setup}\,)$,

$T_{pcq} + T_{mux} + T_{mux} + T_{ALU} + T_{setup}$,

$T_{pcq} + T_{memread} + T_{setup}$,

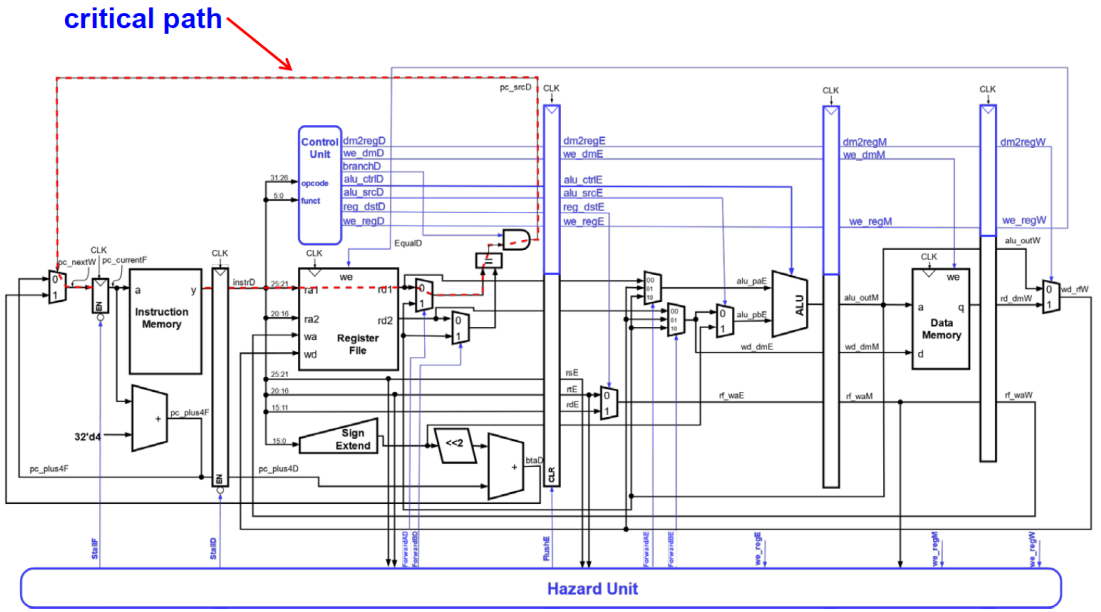
$2\,(\,T_{pcq} + T_{mux} + T_{RFwrite}\,)$

}

RF write and read need to be finished in the first and second half of each cycle, respectively

Pipelined CPU Performance Analysis

Function Unit	Parameter	Delay (ps)
Register clock-to-Q	T_{pcq}	30
Clock setup time	T_{setup}	20
MUX	T_{MUX}	25
Sign-extend	T_{s_ext}	25
ALU	T_{ALU}	200
Mem read	T_{mem}	250
AND	T_{AND}	15
EQ	T_{eq}	40
Register file read	T_{RFread}	150
Register file write	$T_{RFwrite}$	20



Critical path = 2 ($T_{RFread} + T_{mux} + T_{eq} + T_{AND} + T_{mux} + T_{setup}$)

= 2 (150 + 25 + 40 + 15 + 25 + 20) = **550 ps**

SAN JOSÉ STATE UNIVERSITY *powering* SILICON VALLEY