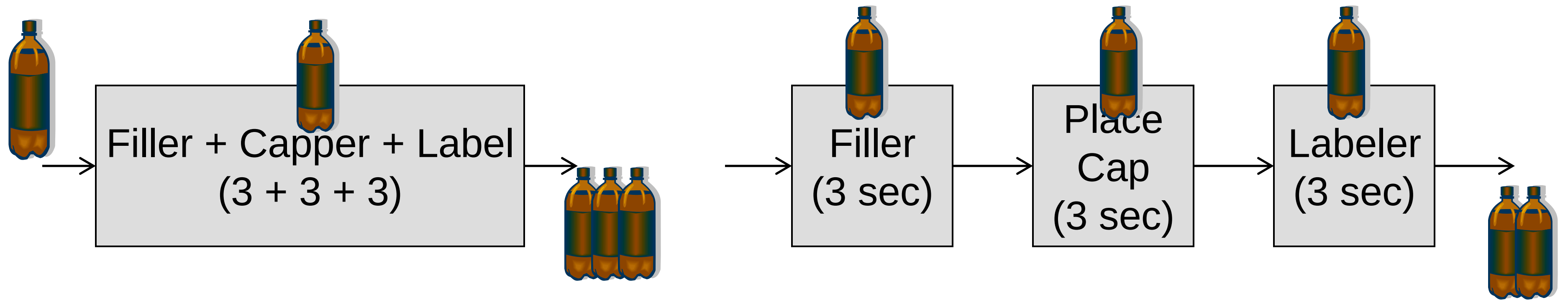
EE 457 Unit 6a

Basic Pipelining Techniques

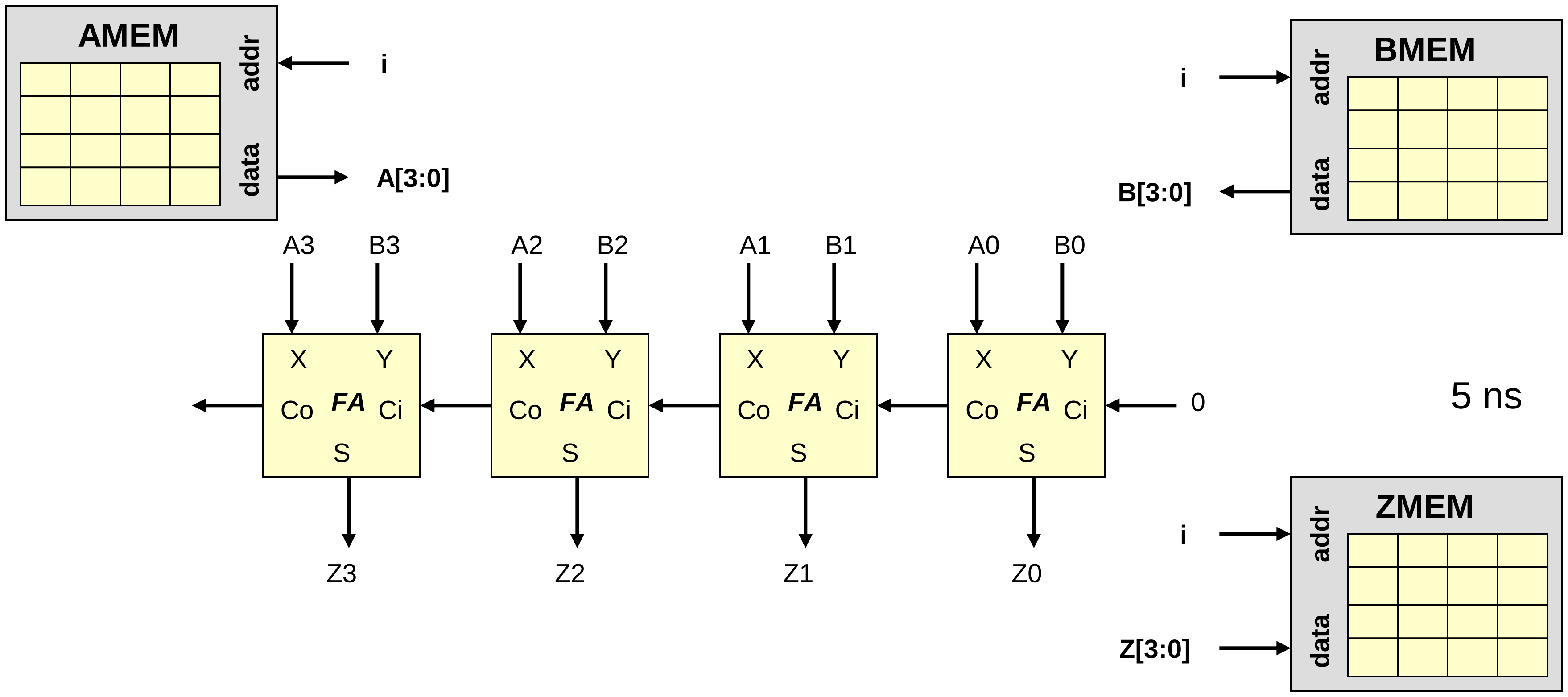
Pipelining Introduction

- Consider a drink bottling plant
  - Filling the bottle = 3 sec.
  - Placing the cap = 3 sec.
  - Labeling = 3 sec.
- Would you want…
  - Machine 1 = Does all three (9 secs.), outputs the bottle, repeats…
  - Machine 2 = Divided into three parts (one for each step) passing bottles between them
- Machine 2 offers ability to overlap steps

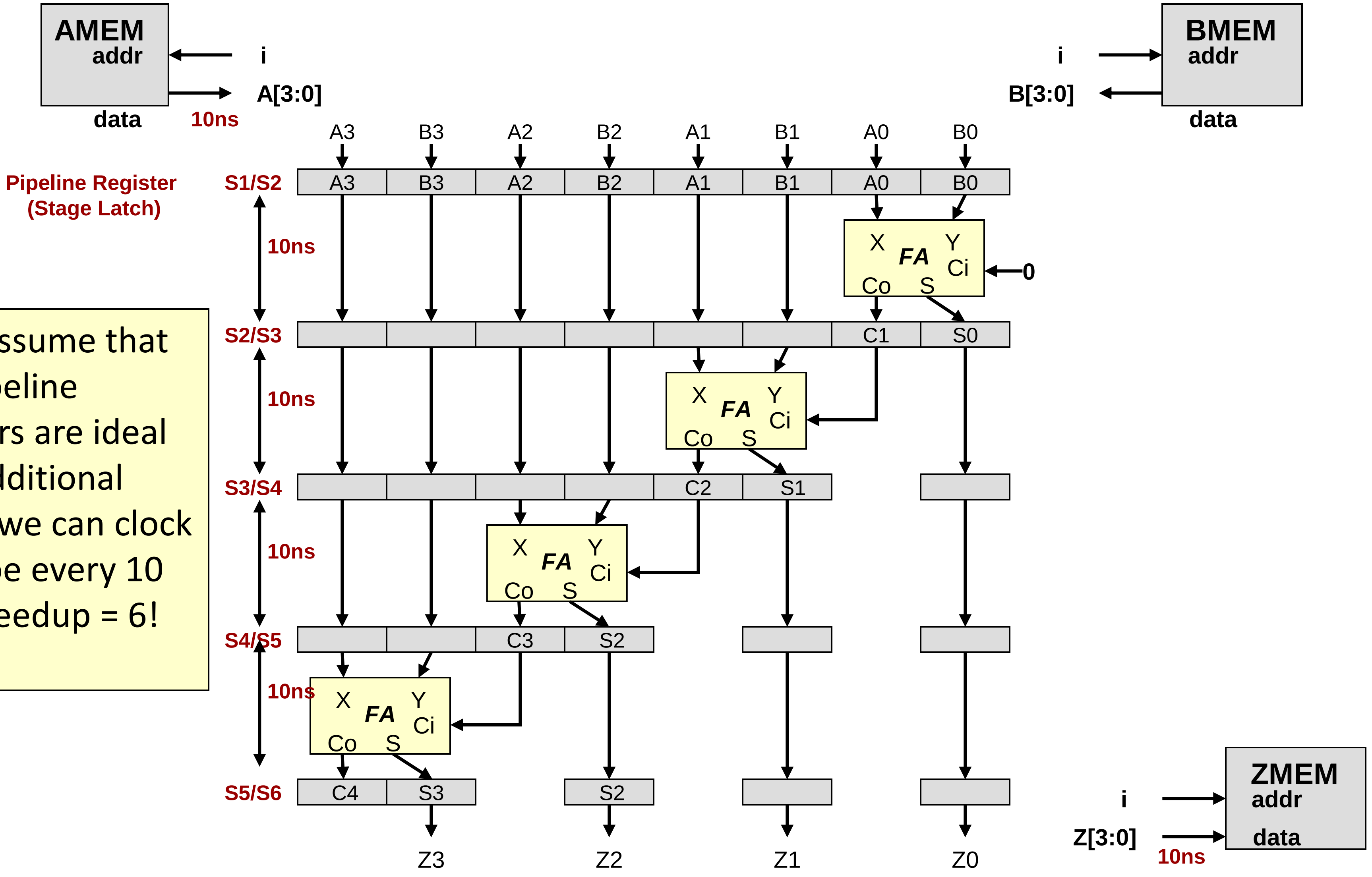


Summing Elements

- Consider adding an array of 4-bit numbers:
  - Z[i] = A[i] + B[i]
  - Delay: 10ns Mem. Access (read or write), 10 ns each FA
  - Clock cycle time = 10 (read) + (10 + 10 + 10 + 10) + 10 (write)



Pipelined Adder



If we assume that the pipeline registers are ideal (0ns additional delay) we can clock the pipe every 10 ns. Speedup = 6!

More Pipelining Examples

- Car Assembly Line
- Wash/Dry/Fold
  - Would you buy a combo washer + dryer unit that does both operations in the same tank??
- Freshman/Sophomore/Junior/Senior

Balancing Pipeline Stages

Clock period must equal the *LONGEST* delay from register to register

- Fig. 1: If total logic delay is 20ns => 50MHz
  - Throughput: 1 instruc. / 20 ns
- Fig. 2: Unbalanced stage delays limit the clock speed to the slowest stage (worst case)
  - Throughput: 1 instruc. / 10 ns => 100MHz
- Fig. 3: Better to split into more, balanced stages
  - Throughput: 1 instruc. / 5 ns => 200MHz
- Fig. 4: Are more stages better
  - Ideally: 2x stages => 2x throughput
  - Throughput: 1 instruc. / 2.5 ns => 400MHz
  - **Each register adds extra delay so at some point deeper pipelines don't pay off**

Fig. 1

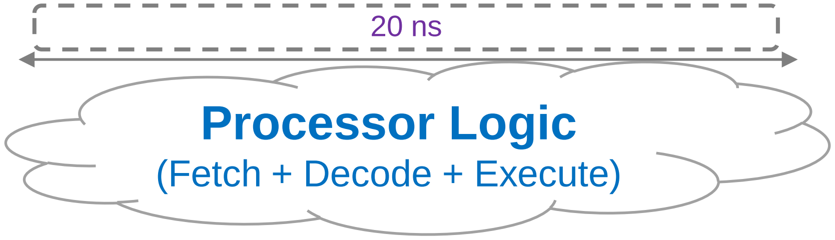


Fig. 2

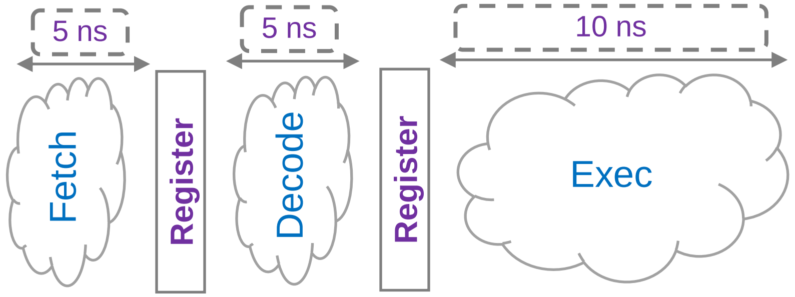


Fig. 3

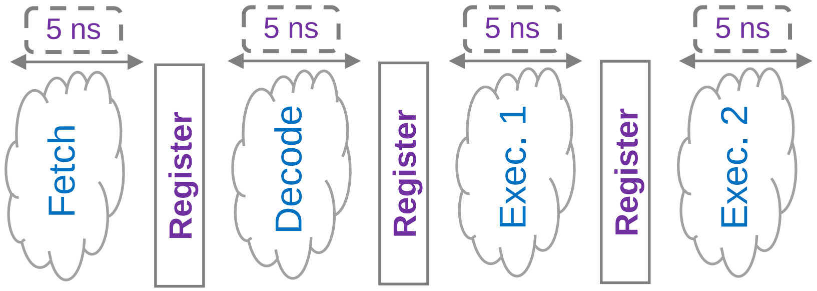
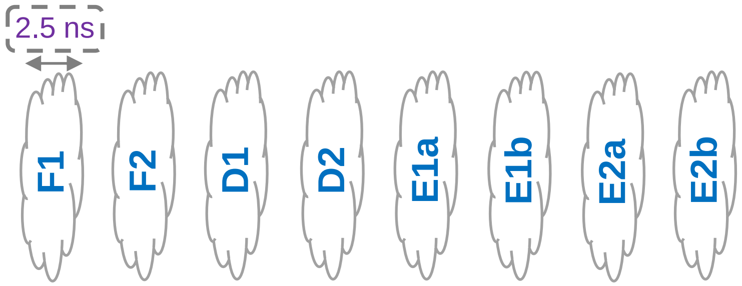


Fig. 4



Processors & Pipelines

- Overlaps execution of multiple instructions
- Natural breakdown into stages
  - Fetch, Decode, Execute, Memory, Write-Back
- Fetch an instruction, while decoding another, while executing another

	CLK 1	CLK 2	CLK 3	CLK 4
Inst 1	Fetch	Decode	Execute	
Inst 2		Fetch	Decode	Execute
Inst 3			Fetch	Decode
Inst 4				Fetch

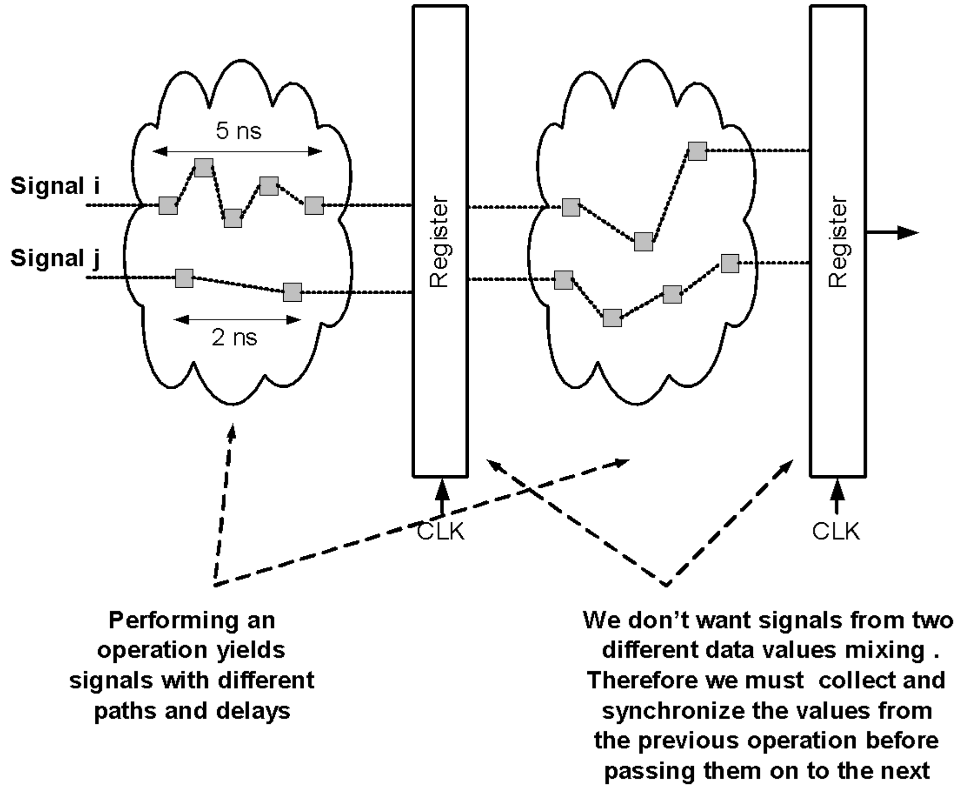
Pipelining (Instruction View)

	Fetch	Decode	Exec.
Clk 1	Inst. 1		
Clk 2	Inst. 2	Inst. 1	
Clk 3	Inst. 3	Inst. 2	Inst. 1
Clk 4	Inst. 4	Inst. 3	Inst. 2
Clk 5	Inst. 5	Inst. 4	Inst. 3

Pipelining (Stage View)

Need for Registers

- Provides separation between combinational functions
  - Without registers, fast signals could "catch-up" to data values in the next operation stage
  - With registers, inputs are "stable"

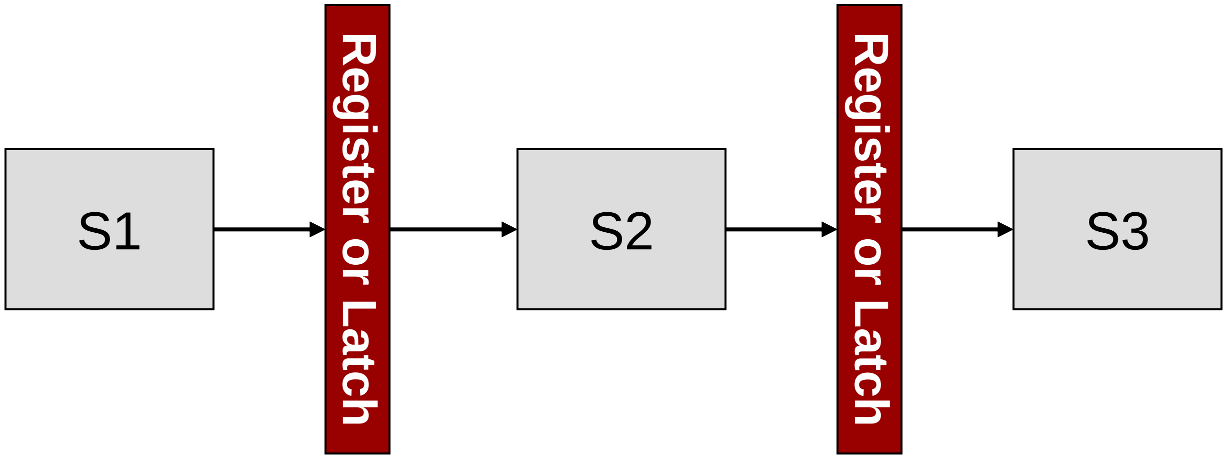


Performing an operation yields signals with different paths and delays

We don't want signals from two different data values mixing . Therefore we must collect and synchronize the values from the previous operation before passing them on to the next

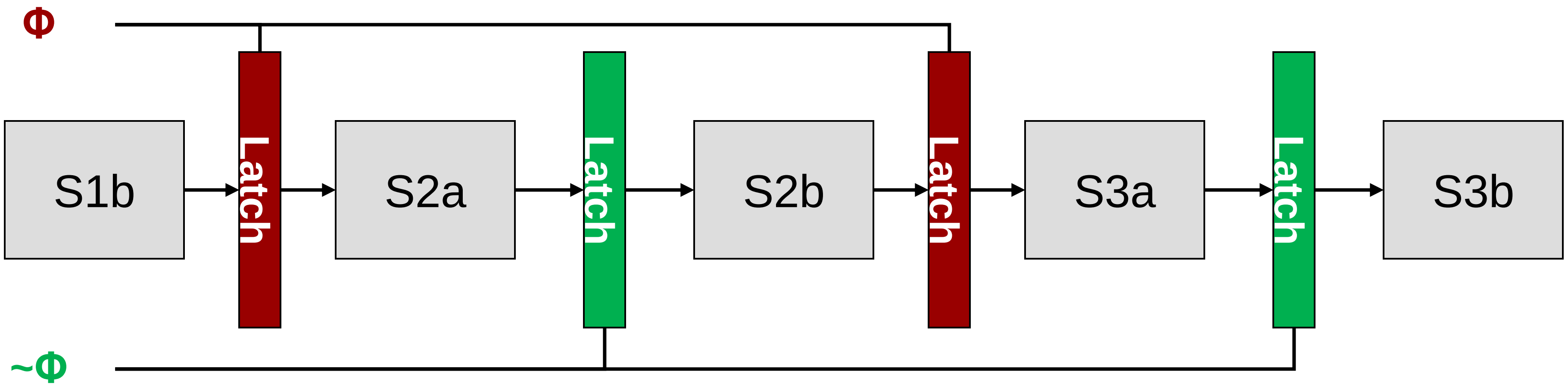
To Register or Latch?

- Should we use pipeline (stage)
  – Registers [edge-sensitive] …or…
  – Latches [level-sensitive]
- Latches may allow data to pass through multiple stages in a single clock cycle
- Answer: Registers in this class!!



But Can We Latch?

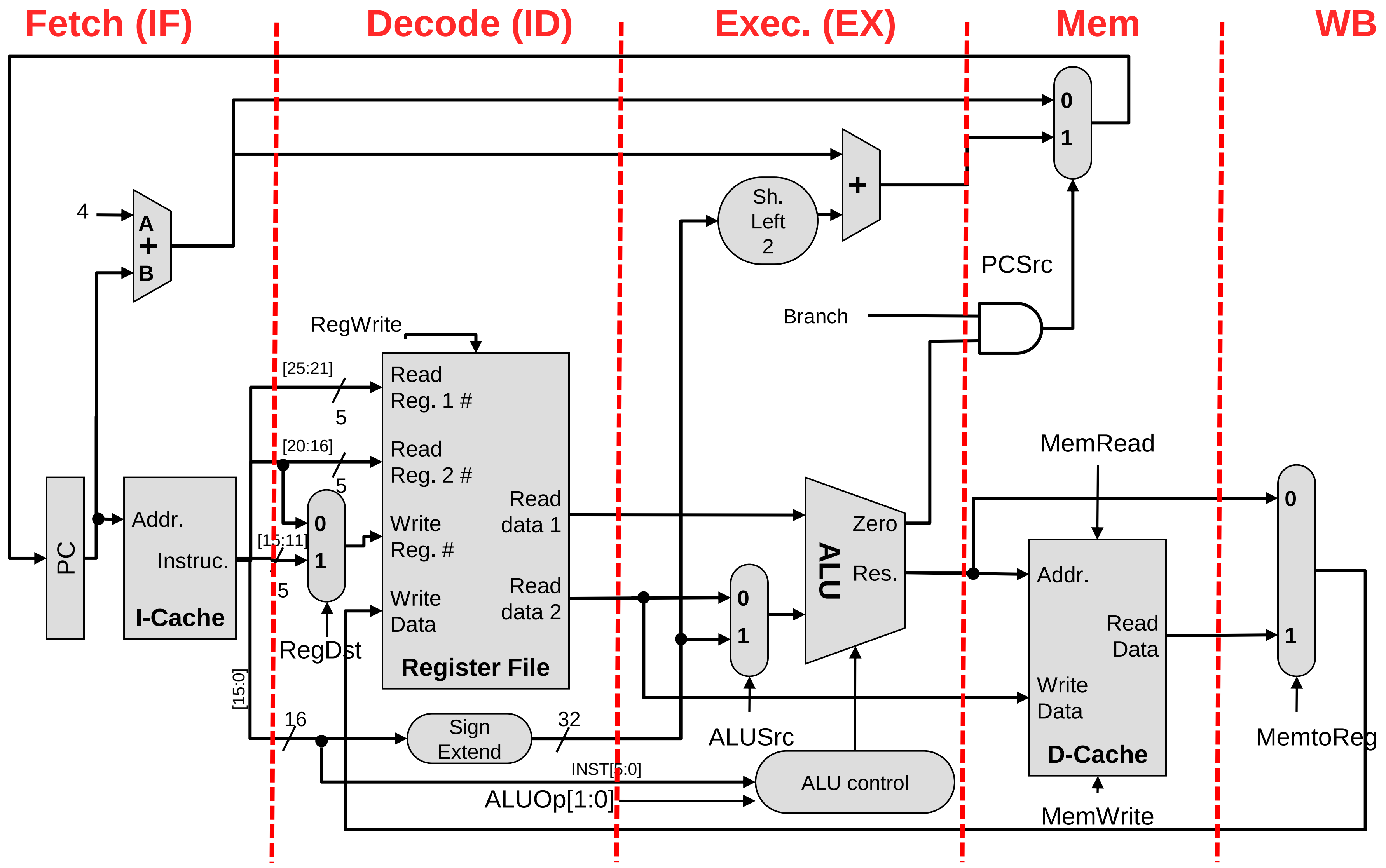
- We can latch if we run the latches on opposite phases of the clock or have a so-called 2-phase clock
  - Because each latch runs on the opposite phase data can only move one step before being stopped by a latch that is in hold (off) mode
- You may learn more about this in EE577a or EE560 (a technique known as Slack Borrowing & Time Stealing)



Pipelining Introduction

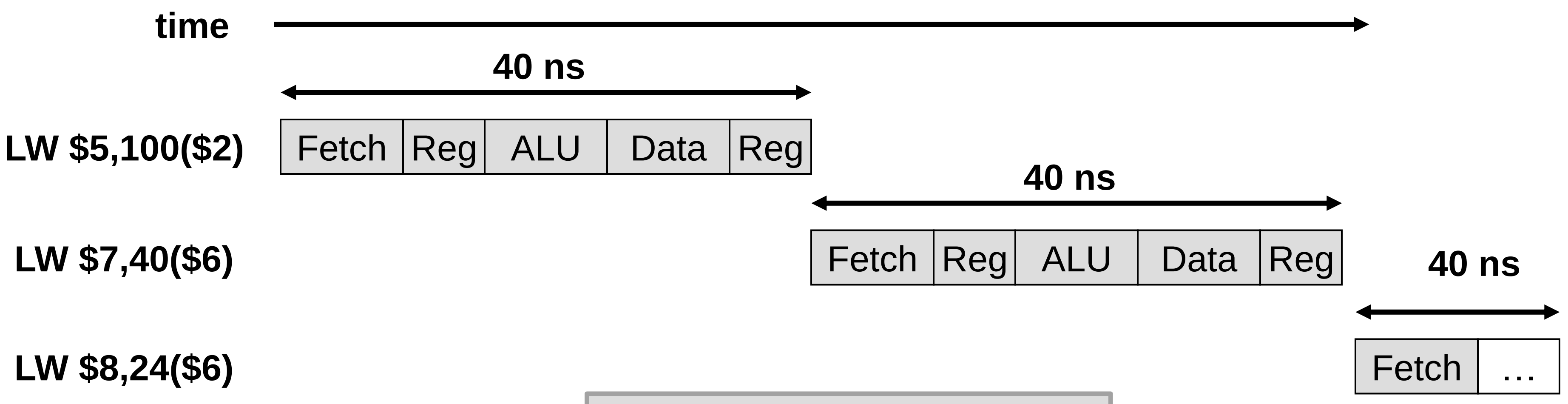
- Implementation technique that overlaps execution of multiple instructions at once
- Improves throughput rather a single-instruction execution latency
- Slowest pipeline stage determines clock cycle time [e.g. a 30 min. wash cycle but 1 hour dry time means 1 hours per load]
- Assuming k stages and perfectly balanced stages:
  - Time before starting next instruc.$_{\text{Pipelined}}$ = Time before starting next instruc.$_{\text{Non-Pipelined}}$ / **# of Stages**
- A 5-stage pipelined CPU may not realize this speedup 5x b/c…
  - The stages may not be perfectly balanced
  - The overhead of filling up the pipe initially
  - The overhead (setup time and clock-to-Q) delay of the stage registers
  - Inability to keep the pipe full due to branches & data hazards

Single-Cycle CPU Datapath



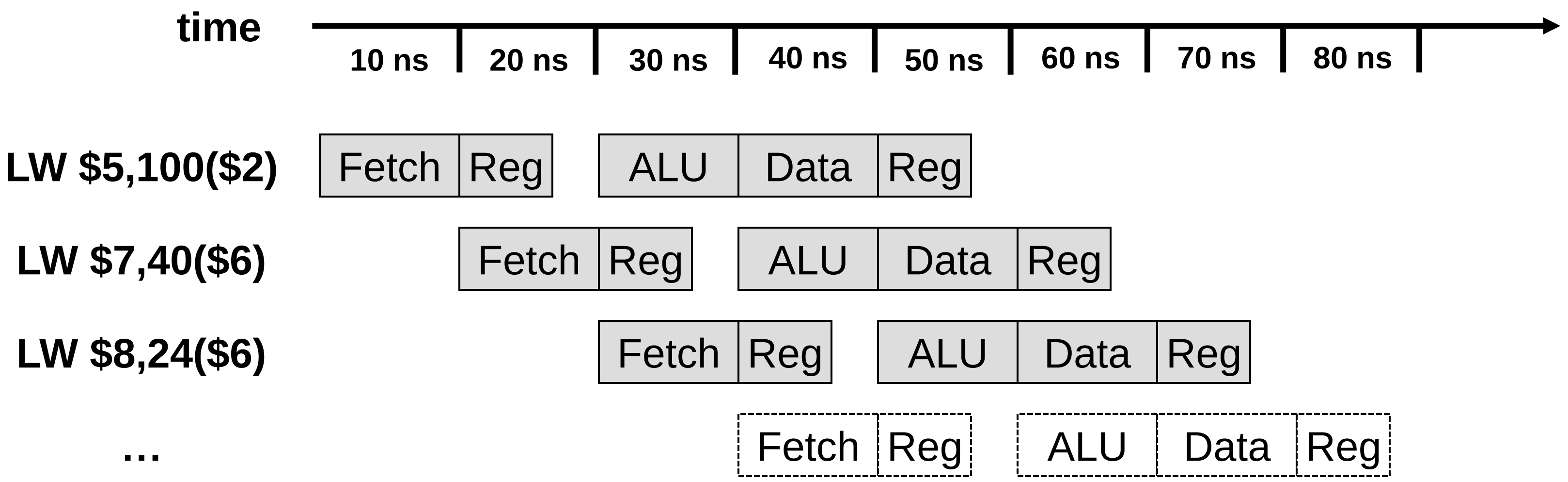
Non-Pipelined Execution

Instruction	Fetch (I-MEM)	Reg. Read	ALU Op.	Data Mem	Reg. Write	Total Time
Load	10 ns	5 ns	10 ns	10 ns	5 ns	40 ns
Store	10 ns	5 ns	10 ns	10 ns		35 ns
R-Type	10 ns	5 ns	10 ns		5 ns	30 ns
Branch	10 ns	5 ns	10 ns			25 ns
Jump	10 ns	5 ns				10 ns



3 Instructions = 3*40 ns

Pipelined Execution



- Notice that even though the register access only takes 5 ns it is allocated a 10 ns slot in the pipeline
- Total time for these 3 pipelined instructions =
  - 70 ns = 50 ns for 1st instruc + 2*10ns for the remaining instructions to complete
- The speedup looks like it is only 120 ns / 70 ns = 1.7x
- But consider 1003 instructions: 1000*40 / 10070 = 3.98 => 4x
  - The overhead of filling the pipeline is amortized over steady-state execution when the pipeline is full

Pipelined Timing

- Execute n instructions using a k stage datapath
  - i.e. Multicycle CPU w/ k steps or single cycle CPU w/ clock cycle k times slower
- w/o pipelining: ***n*k cycles***
  - n instrucs. * k CPI
- w/ pipelining: ***k+n-1 cycles***
  - k cycle for 1st instruc. + (n-1) cycles for n-1 instrucs.
  - Assumes we keep the pipeline full

	Fetch 10ns	Decode 10ns	Exec. 10ns	Mem. 10ns	WB 10ns
C1	ADD				
C2	SUB	ADD			
C3	LW	SUB	ADD		
C4	SW	LW	SUB	ADD	
C5	AND	SW	LW	SUB	ADD
C6	OR	AND	SW	LW	SUB
C7	XOR	OR	AND	SW	LW
C8		XOR	OR	AND	SW
C9			XOR	OR	AND
C10				XOR	OR
C11					XOR

Pipeline Filling

Pipeline Full

Pipeline Emptying

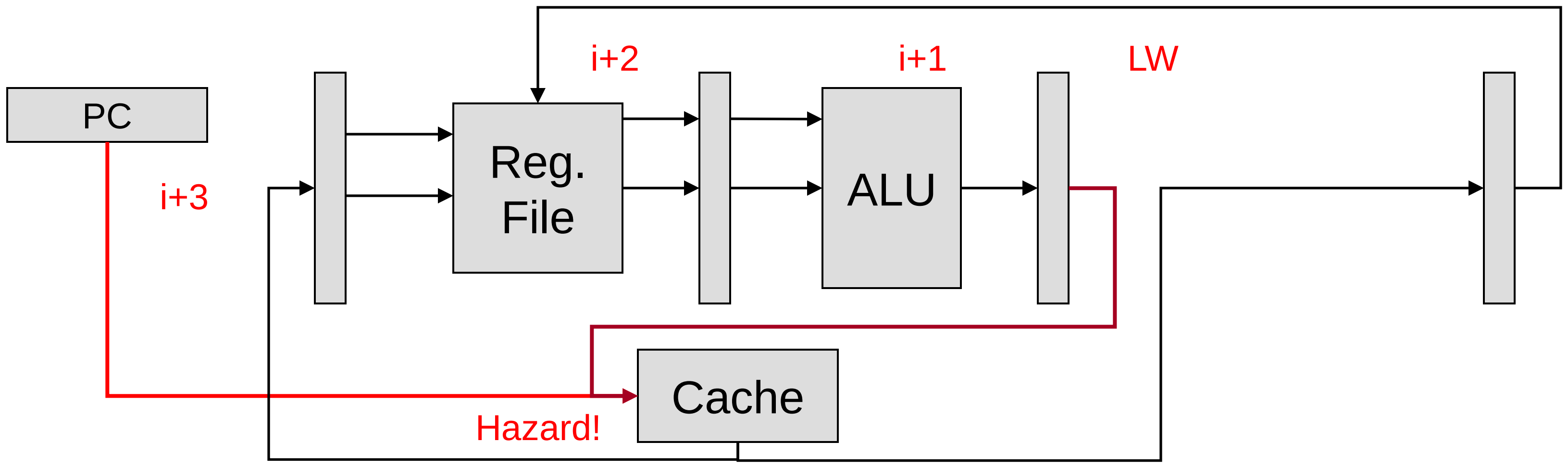
7 Instrucs. = 11 clocks (5 + 7 – 1)

Designing the Pipelined Datapath

- To pipeline a datapath in five stages means five instructions will be executing ("in-flight") during any single clock cycle
- Resources cannot be shared between stages because there may always be an instruction wanting to use the resource.
  - This is known as a **STRUCTURAL HAZARD**
  - Each stage needs its own resources
  - The single-cycle CPU datapath also matches this concept of no shared resources
  - We can simply divide the single-cycle CPU into stages

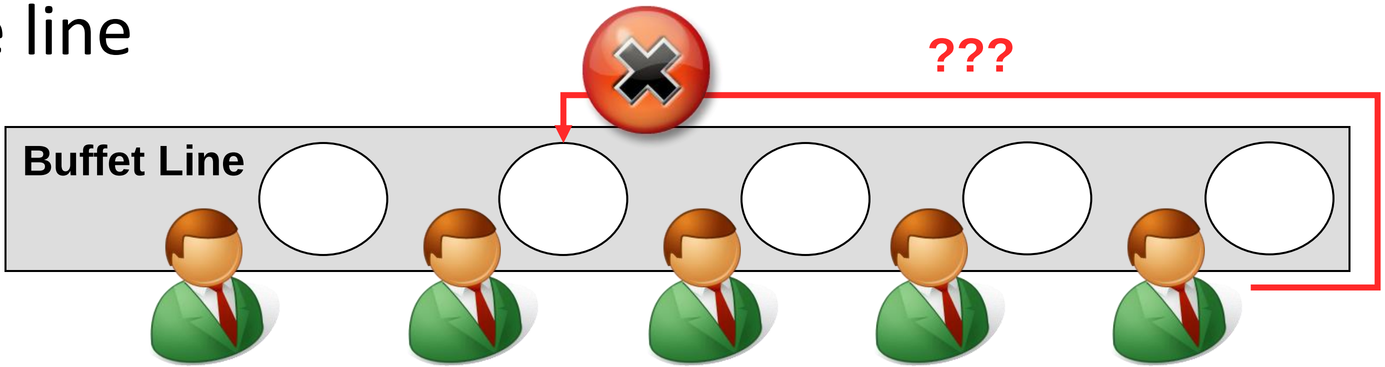
Structural Hazard Example

- **Example structural hazard**: A single SHARED cache (instruction + data) rather than separate instruction & data caches
  - Structural hazard any time an instruction needs to perform a data access (i.e. `lw` or `sw`) since we always want to fetch a new instruction each clock cycle



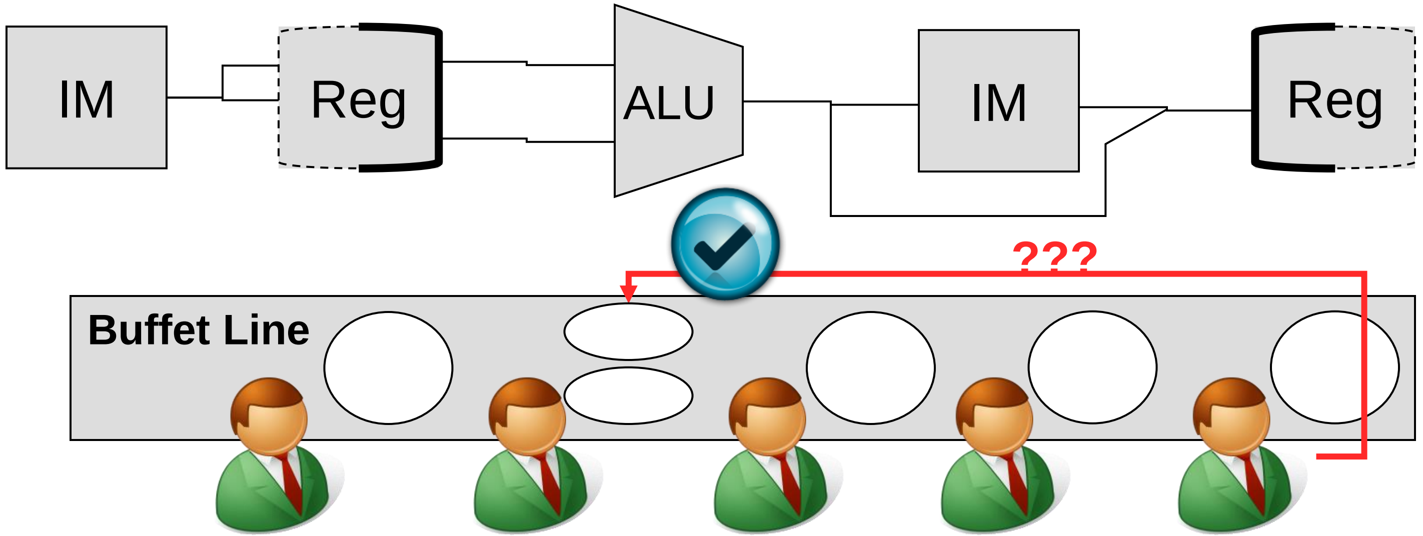
Information Flow in a Pipeline

- Data or control information should flow only in the forward direction in a linear pipeline
  - Non-linear pipelines where information is fed back into a previous stage occurs in more complex pipelines such as floating point dividers
- The CPU pipeline is like a buffet line or cafeteria where people can not try to revisit a a previous serving station without disrupting the smooth flow of the line



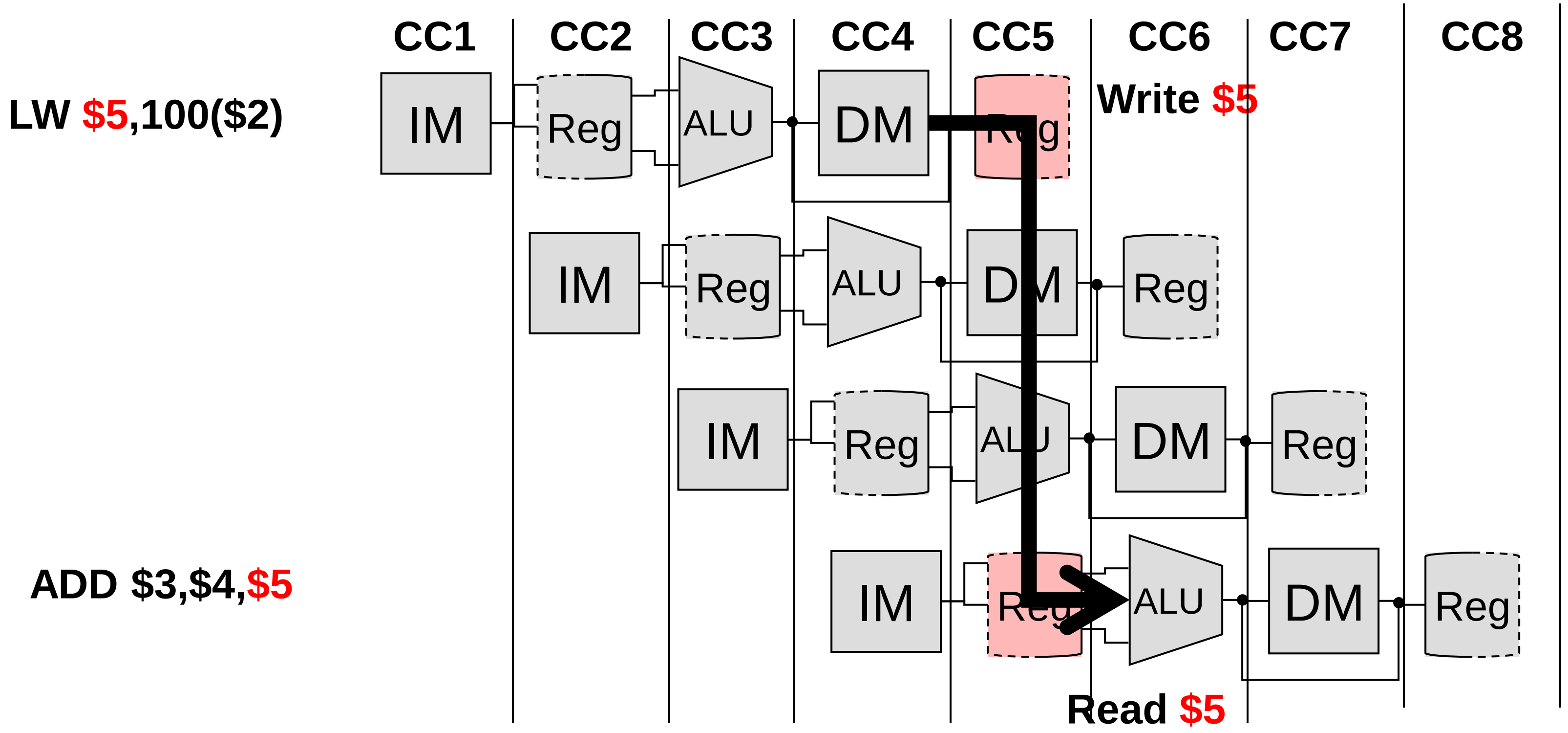
Register File

- Don’t we have a non-linear flow when we write a value back to the register file?
  - An instruction in WB is re-using the register file in the ID stage
  - Actually we are utilizing different “halves” of the register file
    - ID stage reads register values
    - WB stage writes register value
  - Like a buffet line with 2 dishes at one station



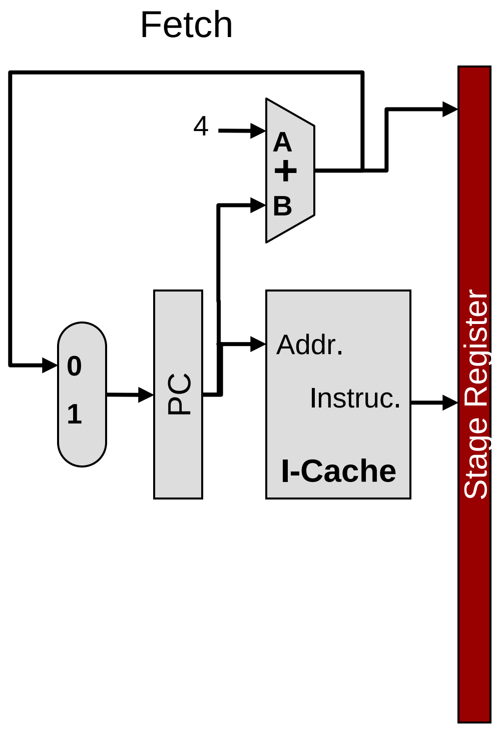
Register File

- Only an issue if WB to same register as being read
- Register file can be designed to do "internal forwarding" where the data being written is immediately passed out as the read data



Pipelining the Fetch Phase

- Note that to keep the pipeline full we have to fetch a new instruction every clock cycle
- Thus we have to perform PC = PC + 4 every clock cycle
- Thus there shall be no pipelining registers in the datapath responsible for PC = PC +4
- Support for branch/jump warrants a lengthy discussion which we will perform later

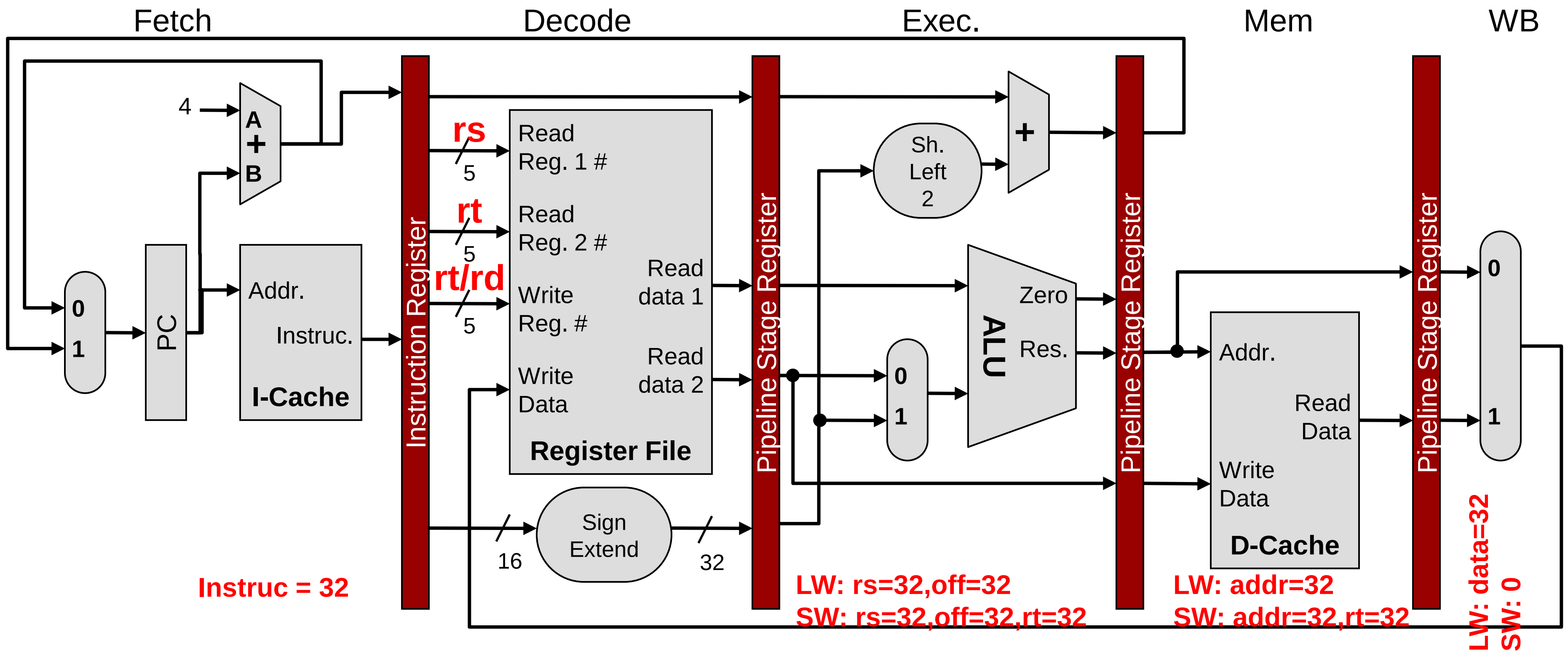


PIPELINE CONTROL

Basic 5 Stage Pipeline

- Compute the size of each pipeline register (find the max. info needed for any instruction in each stage)
- To simplify, just consider LW/SW (Ignore control signals)

LW $10,40($1)	Op = 35	rs=1	rt=10	immed.=40
SW $15,100($2)	Op = 43	rs=2	rt=15	immed.=100

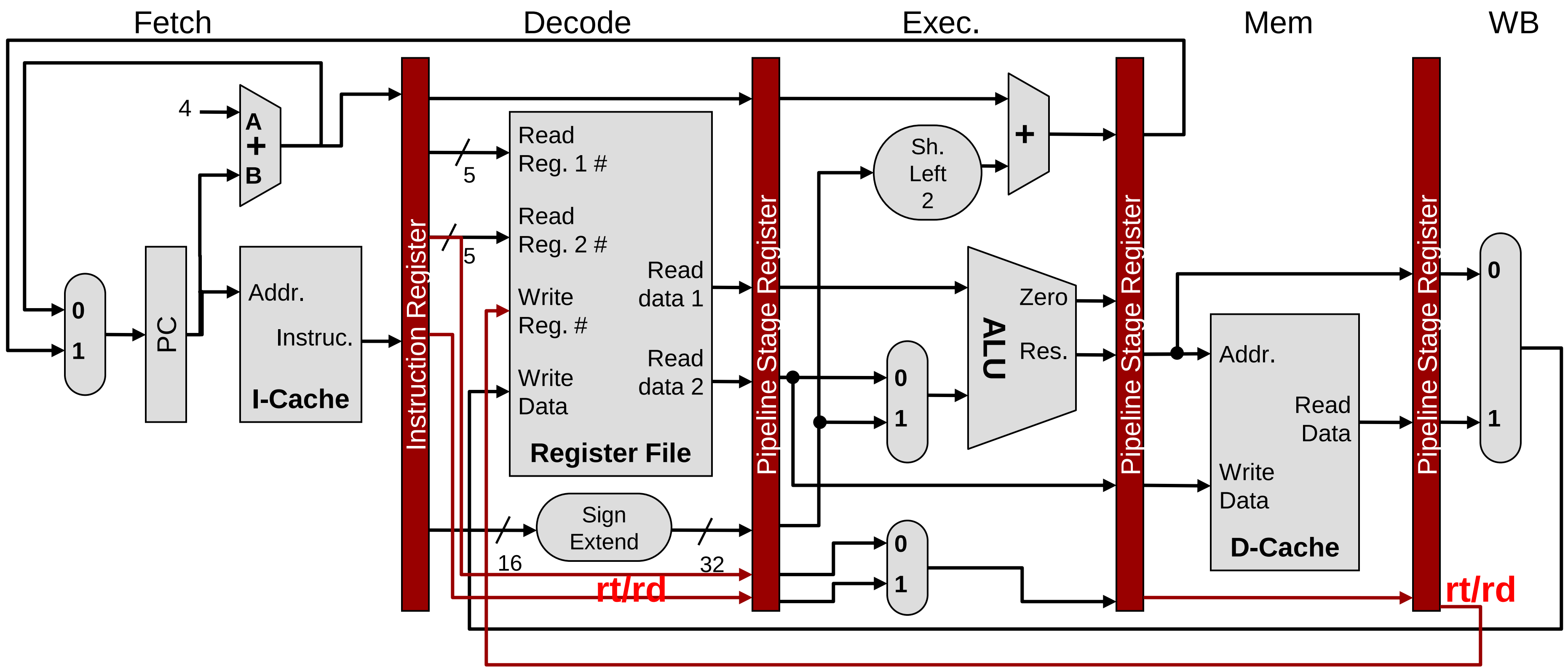


Basic 5 Stage Pipeline

- There is a bug in the load instruction implementation
- Which register is written with the data read from memory?
- We need to preserve the dest. register number by carrying it through the pipeline with us
- In general this is true for all signals needed later in the pipe

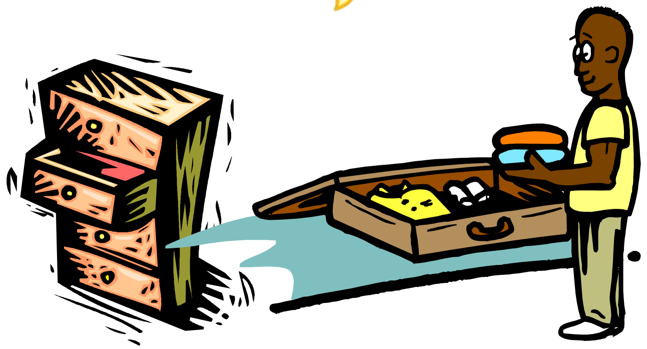
LW $10,40($1)

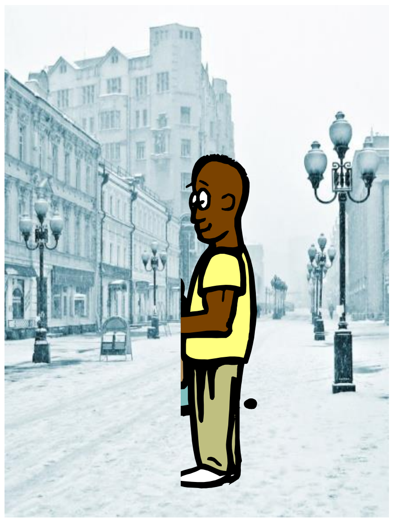
SW $15,100($2)



Pipeline Packing List

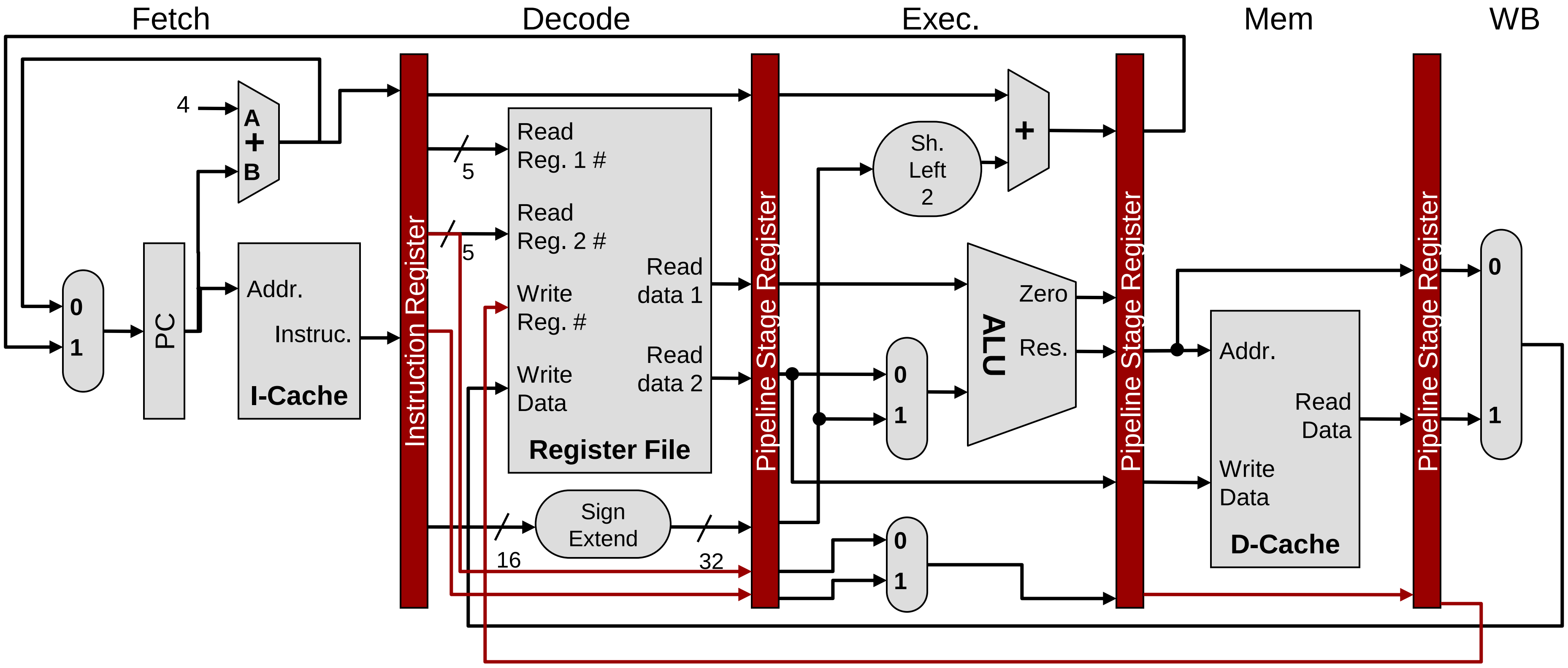
- Just as when you go on a trip you have to pack everything you need in advance since you cannot come back to your closet, so in pipelines you have to take all the control and data you will need with you down the pipeline until you use it





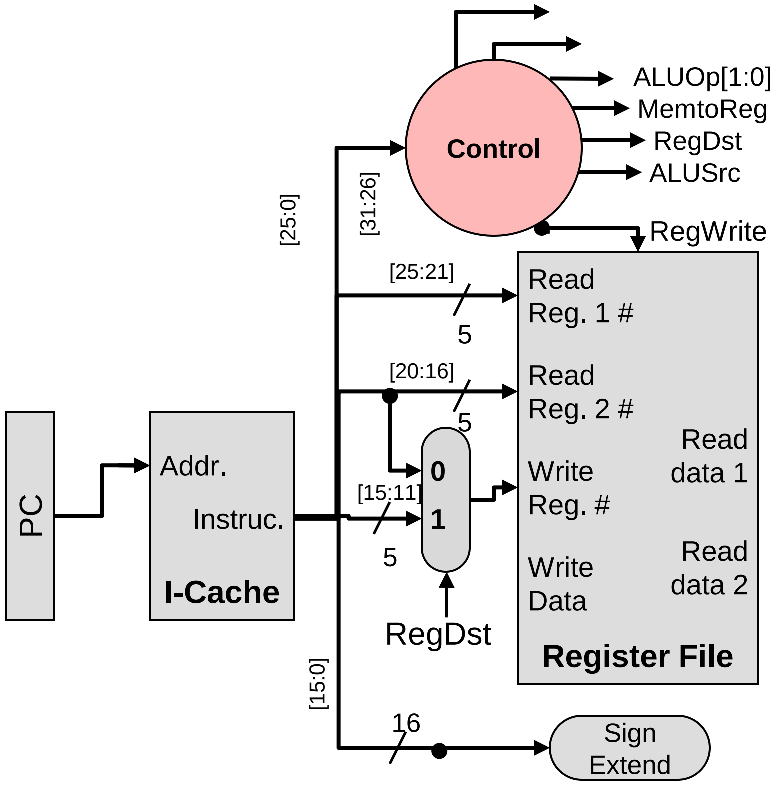
Pipeline Control Overview

- We will just consider basic (simple) pipeline control and deal with problems related to branch and data hazards later
- It is assumed that the PC and pipeline register update on each clock cycle so no separate write enable signals are needed for these registers



Control Signal Generation

- Recall from the Single-Cycle CPU discussion that there is no state machine control, but a simple translator (combinational logic) to translate the 6-bit opcode into these 9 control signals
- Since the datapaths of the single-cycle and pipelined CPU are essentially the same, so is the control
- The main difference is that the control signals are generated in one clock cycle and used in a subsequent cycle (later pipeline stage)
- We can produce all our signals in the ID stage and use the pipeline registers to store and pass them to the consuming stage



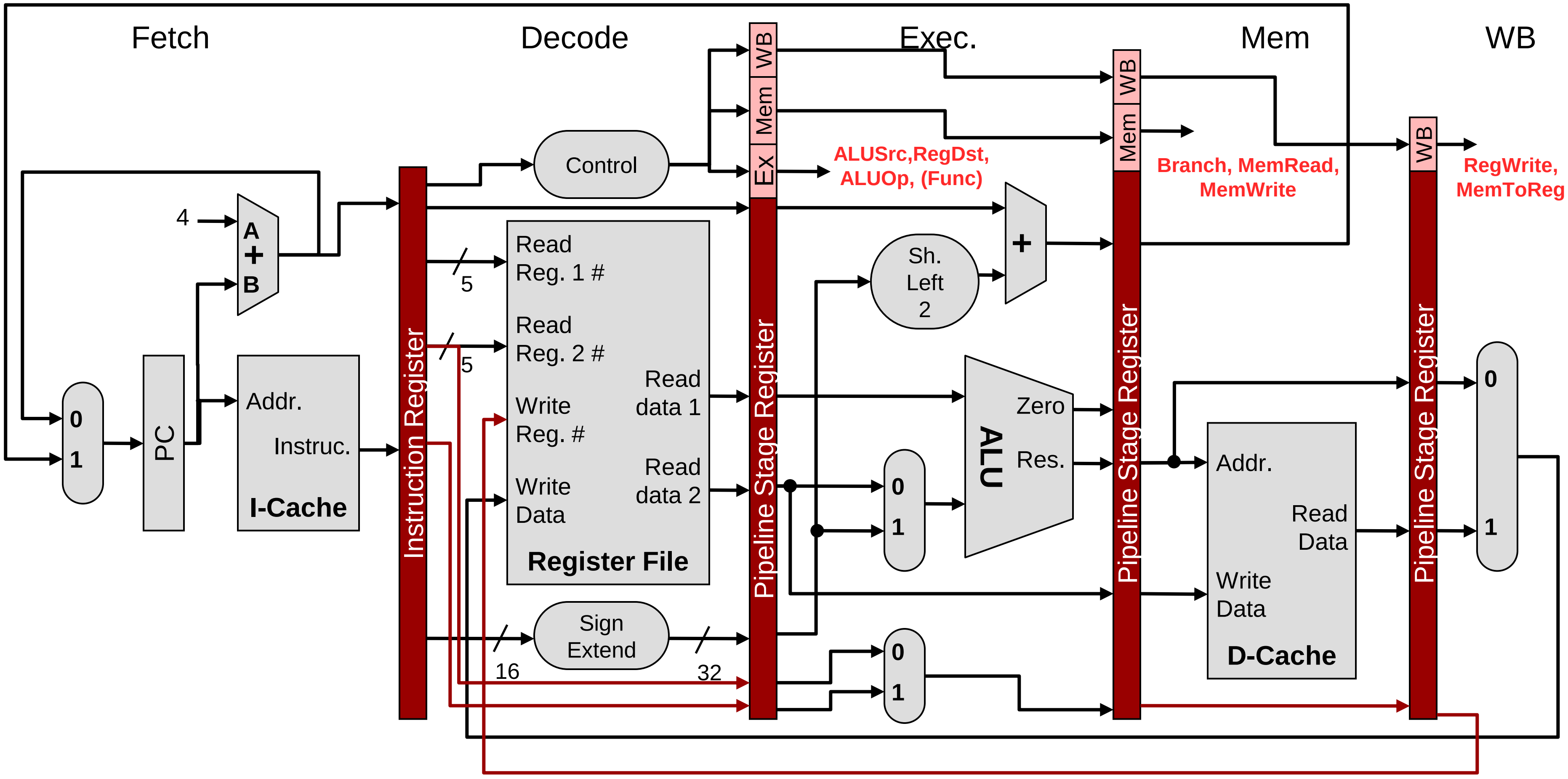
Control Signals per Stage

- How many control signals are needed in each stage

	Execution Stage = 4 signals (10 if you count function codes)				**Mem stage = 3 signals**			**WB Stage = 2 signals**	
Instruction	Reg Dst	ALU Src	ALU Op[1:0]	Func[5:0]	Branch	Mem Read	Mem Write	Reg Write	Memto-Reg
R-format	1	0	10	…	0	0	0	1	0
LW	0	1	00	X	0	1	0	1	1
SW	X	1	00	X	0	0	1	0	X
Beq	X	0	01	X	0	0	0	0	X

Basic 5 Stage Pipeline

- Control is generated in the decode stage and passed along to consuming stages through stage registers



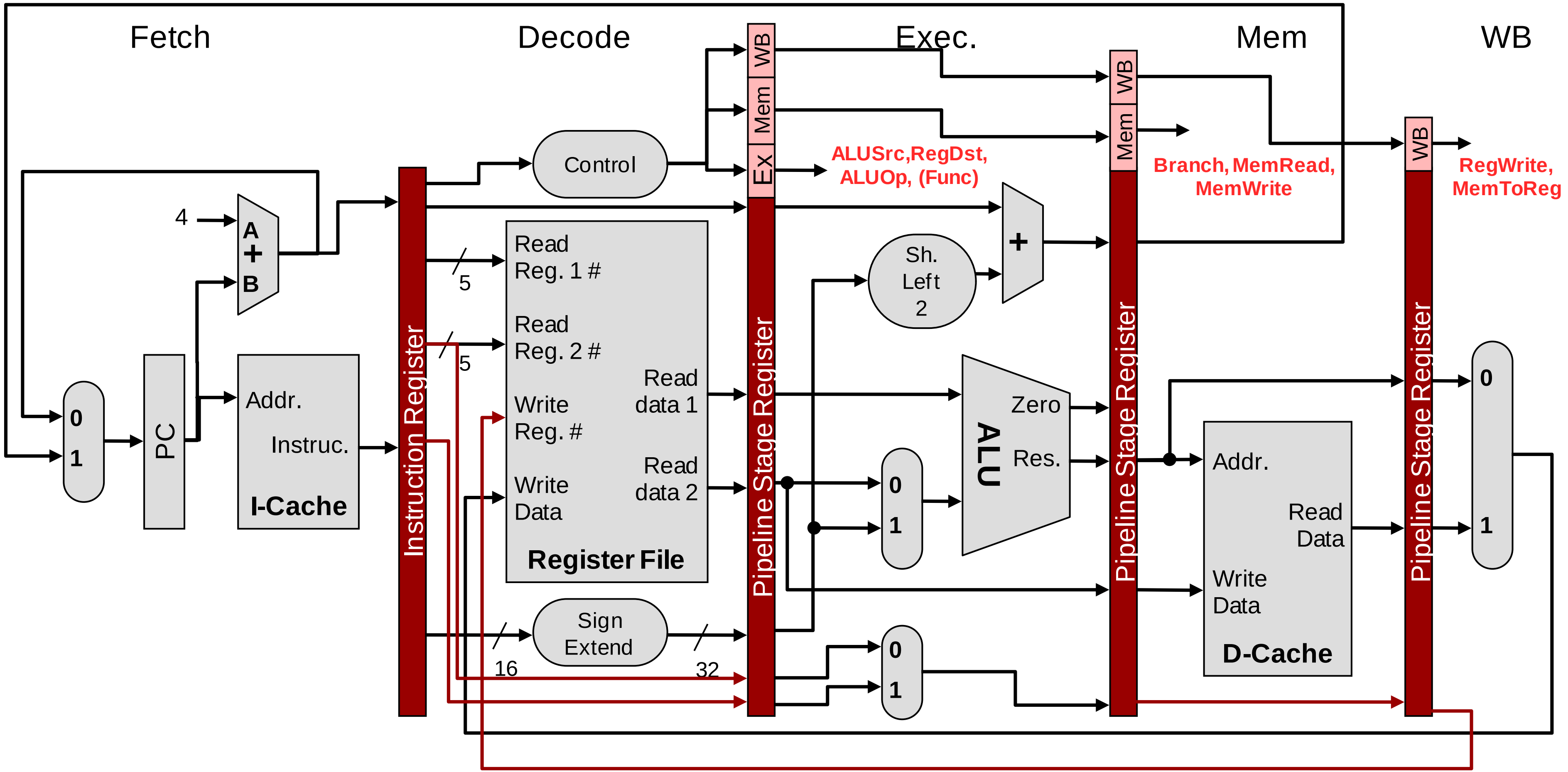
Stage Control

- **Instruction Fetch:** The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage
- **ID/RF**: As in the previous stage the same thing happens at every clock cycle so there are no optional control lines to set
- **Execution**: The signals to be set are RegDst, ALUop/Func, and ALUSrc. The signals determine whether the result of the instruction written into the register specified by bits 20-16 (for a load) or 15-11 for an R-format), specify the ALU operation, and determine whether the second operand of the ALU will be a register or a sign-extended immediate
- **Memory Stage**: The control lines set in this stage are Branch, MemRead, and MemWrite. These signals are set for the BEQ, LW, and SW instructions respectively
- **WriteBack**: the two control lines are RegWrite , which writes the chosen register, and MemToReg, which decides between the ALU result or memory value as the write data

Exercise:

- On copies of this sheet, show this sequence executing on the pipeline:

1. LW $10,40($1) 2. **SUB $11,$2,$3** 3. **AND $12,$4,$5**
4. **OR $13,$6,$7** 5. **ADD $14,$8,$9**

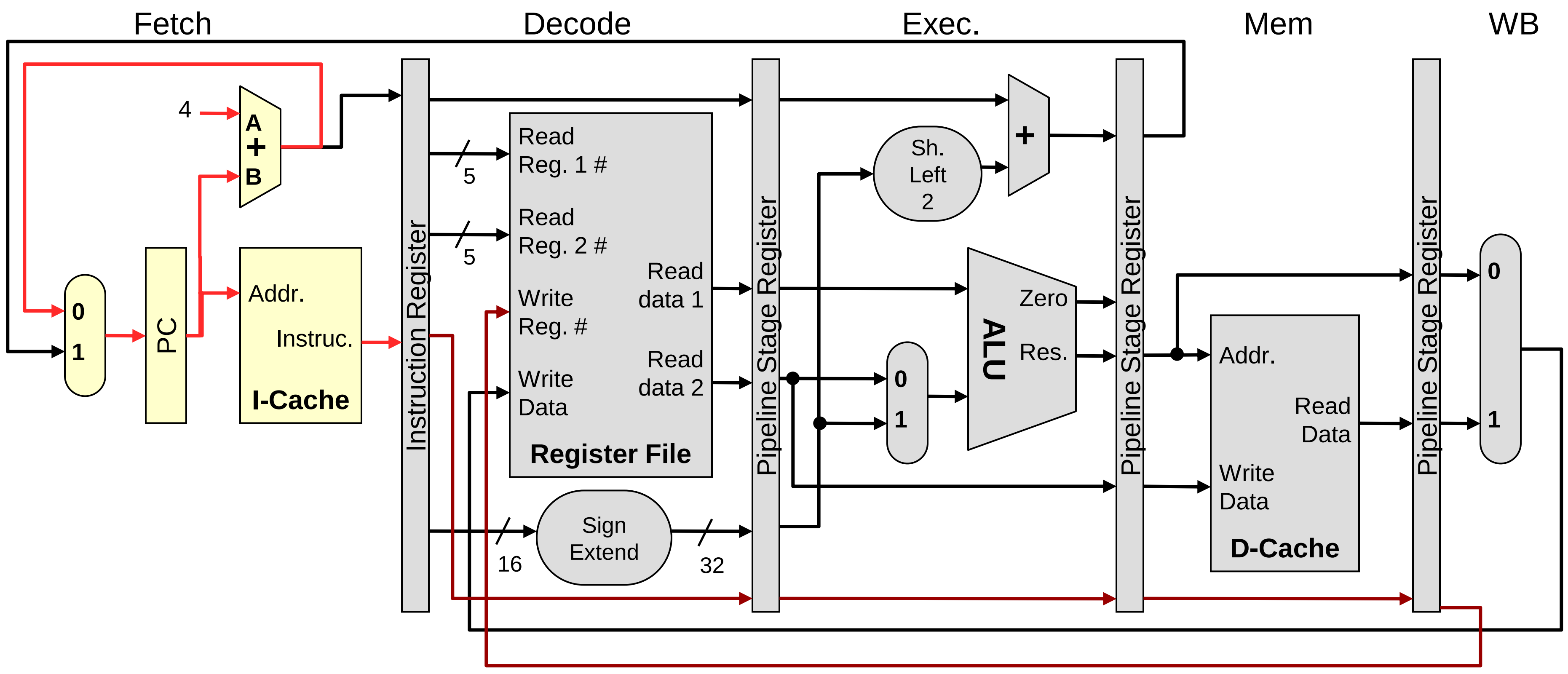


Review

- Although an instruction can begin at each clock cycle, an individual instruction still takes five clock cycles
- Note that it takes four clock cycle before the five-stage pipeline is operating at full efficiency
- Register write-back is controlled by the WB stage even though the register file is located in the ID stage; the correct write register ID is carried down the pipeline with the instruction data
- When a stage is inactive, the values of the control lines are deasserted (shown as 0's) to prevent anything harmful from occurring
- No state machine is needed; sequencing of the control signals follows simply from the pipeline itself (i.e. control signals are produced initially but delayed by the stage registers until the correct stage / clock cycle for application of that signal)

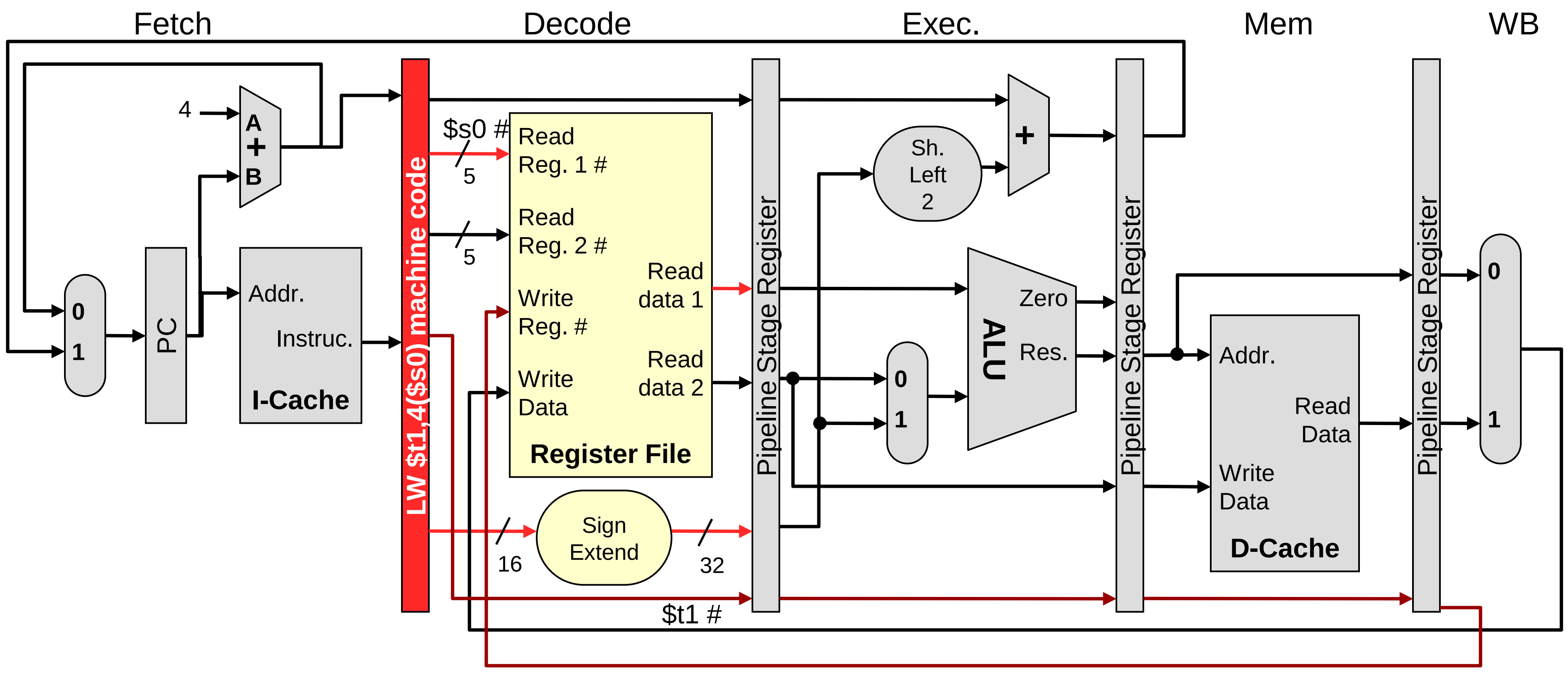
ADDITIONAL REFERENCE

LW $t1,4($s0): Fetch



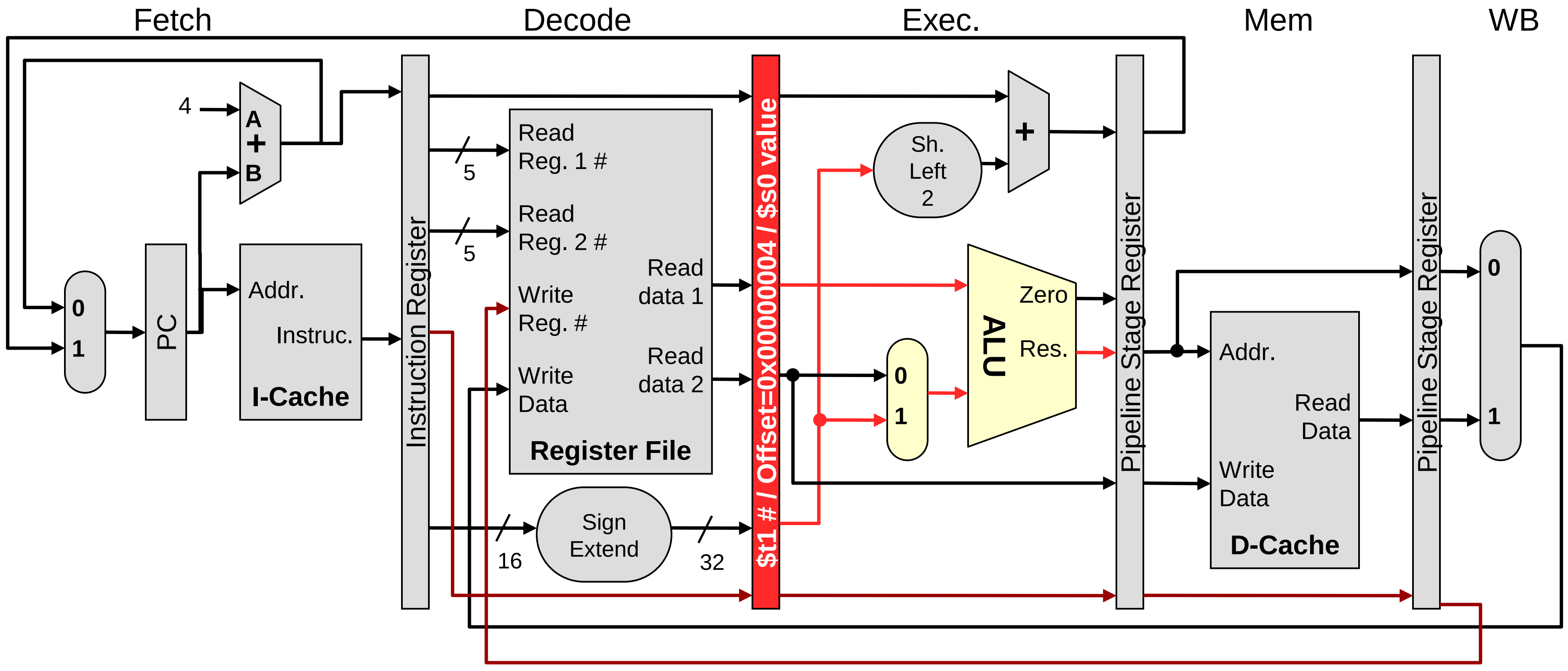
Fetch LW and increment PC

LW $t1,4($s0): Decode



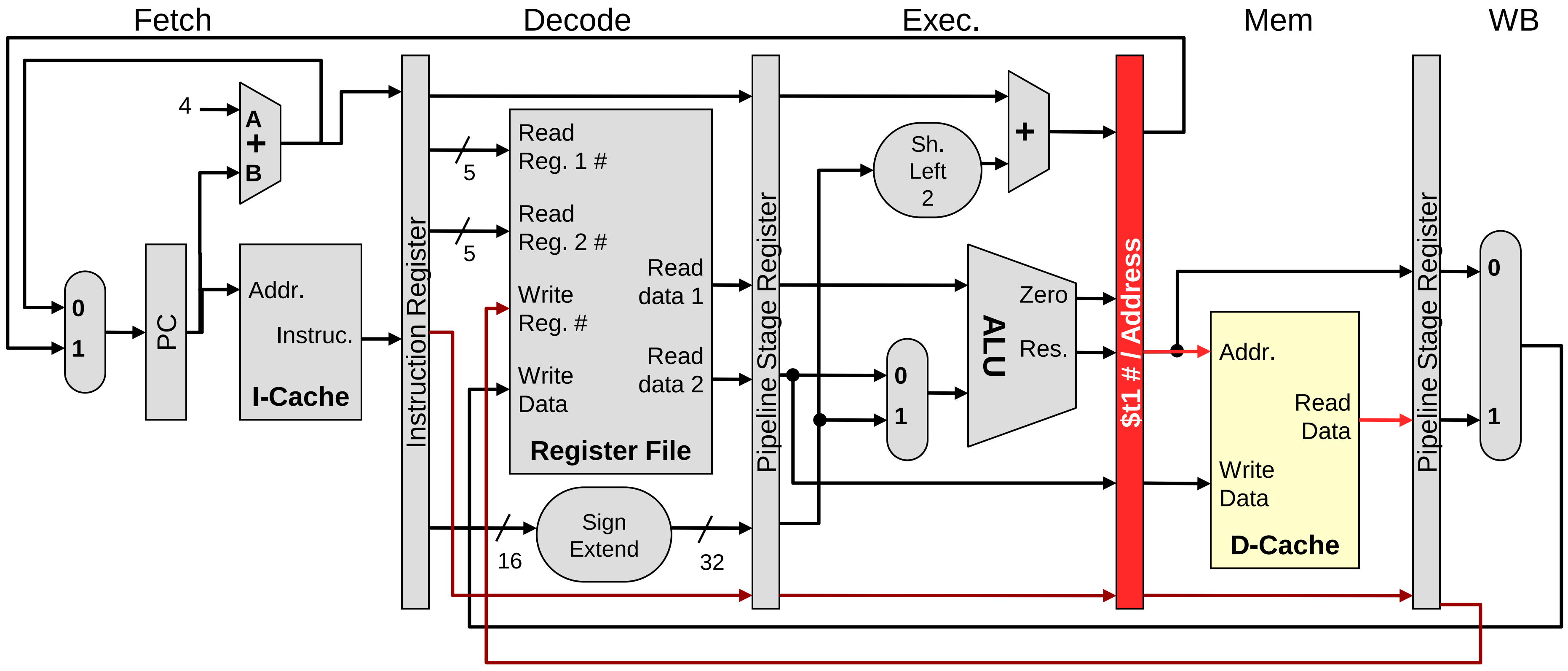
Decode instruction
and fetch operands

LW $t1,4($s0): Execute



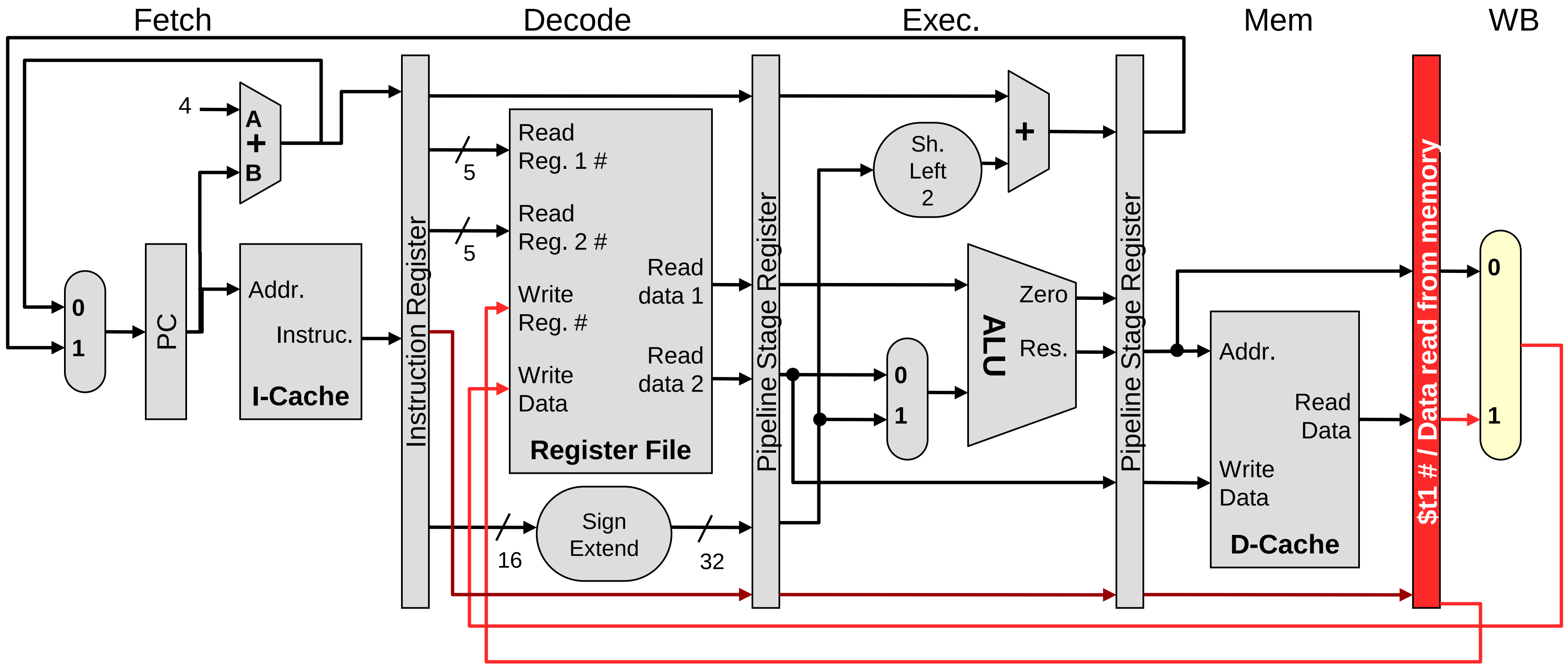
Add offset 4 to
$s0 value

LW $t1,4($s0): Memory



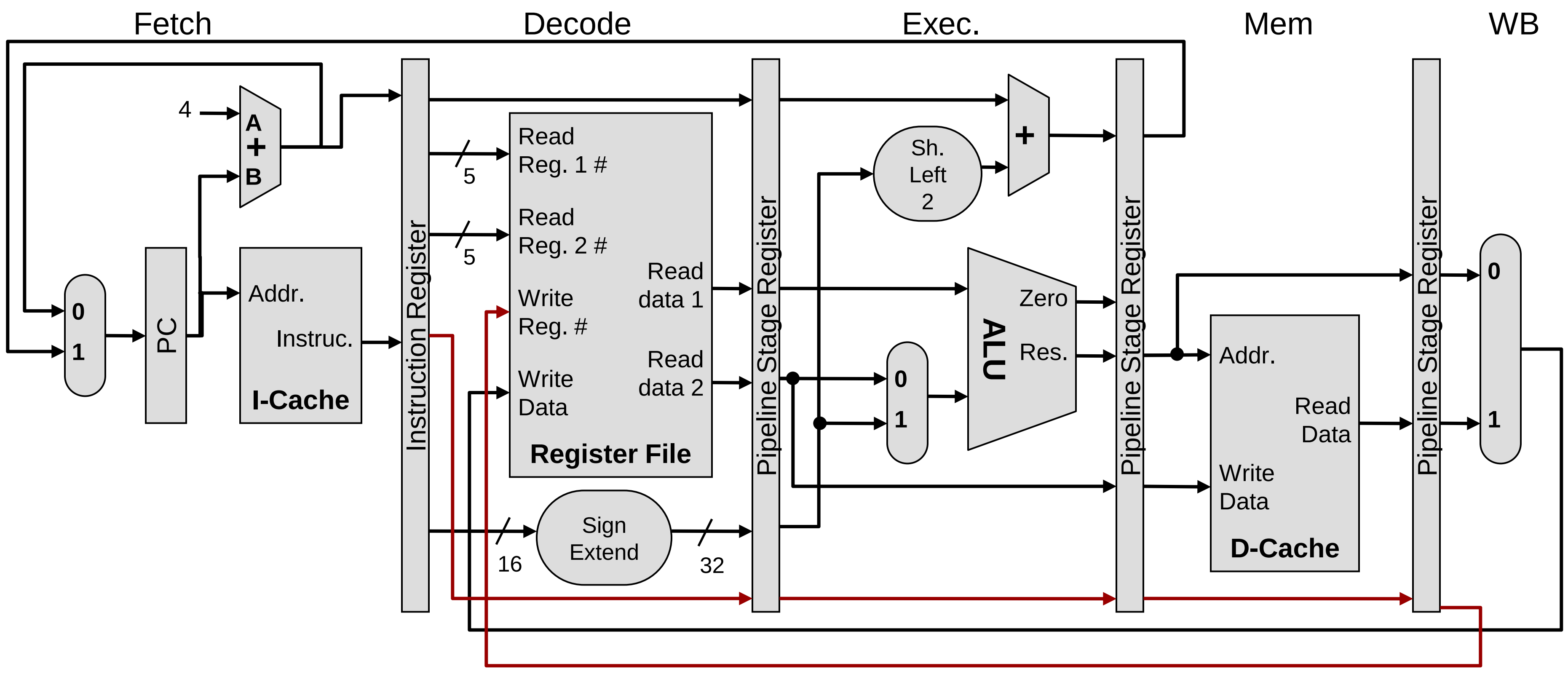
Read word from memory

LW $t1,4($s0): Writeback



Write word to $t1

LW $t1,4($s0)



Fetch LW

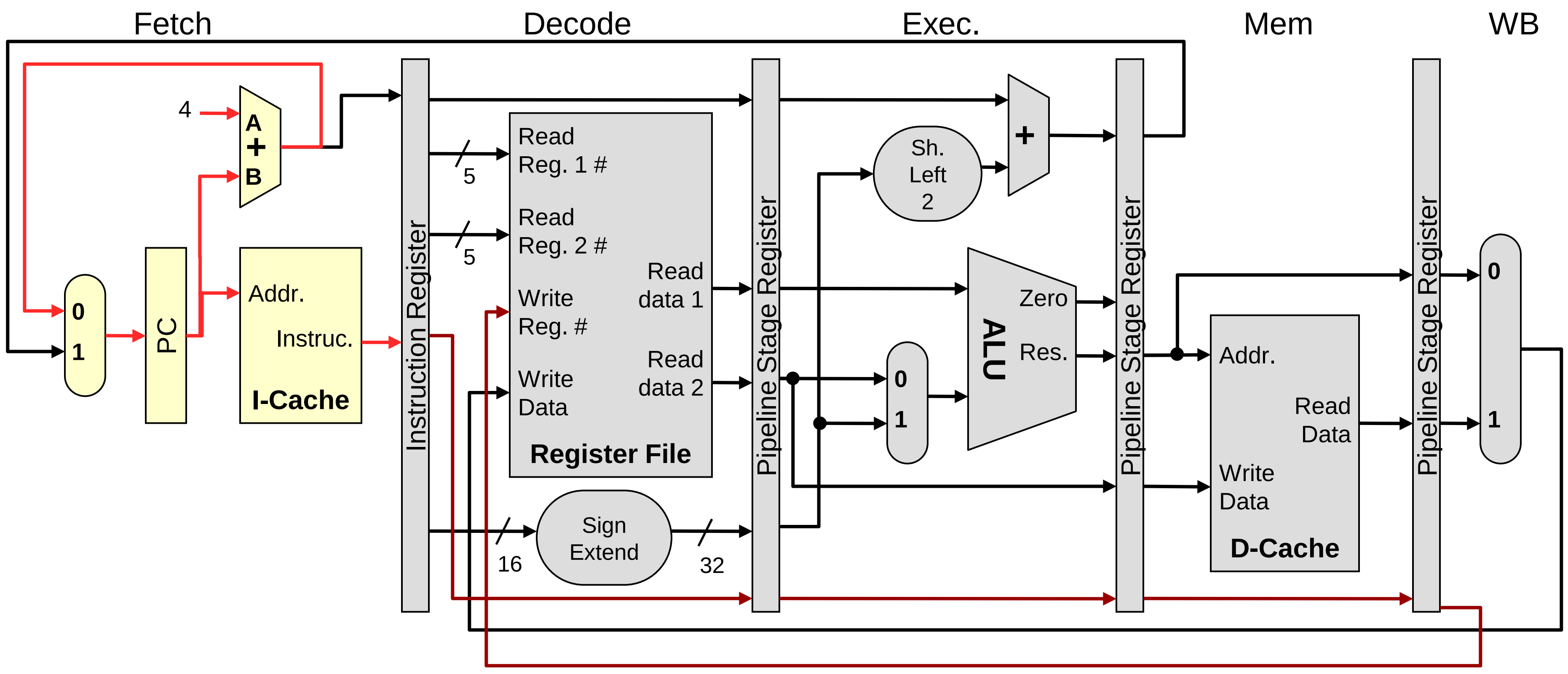
Decode instruction and fetch operands

Add offset 4 to $s0

Read word from memory

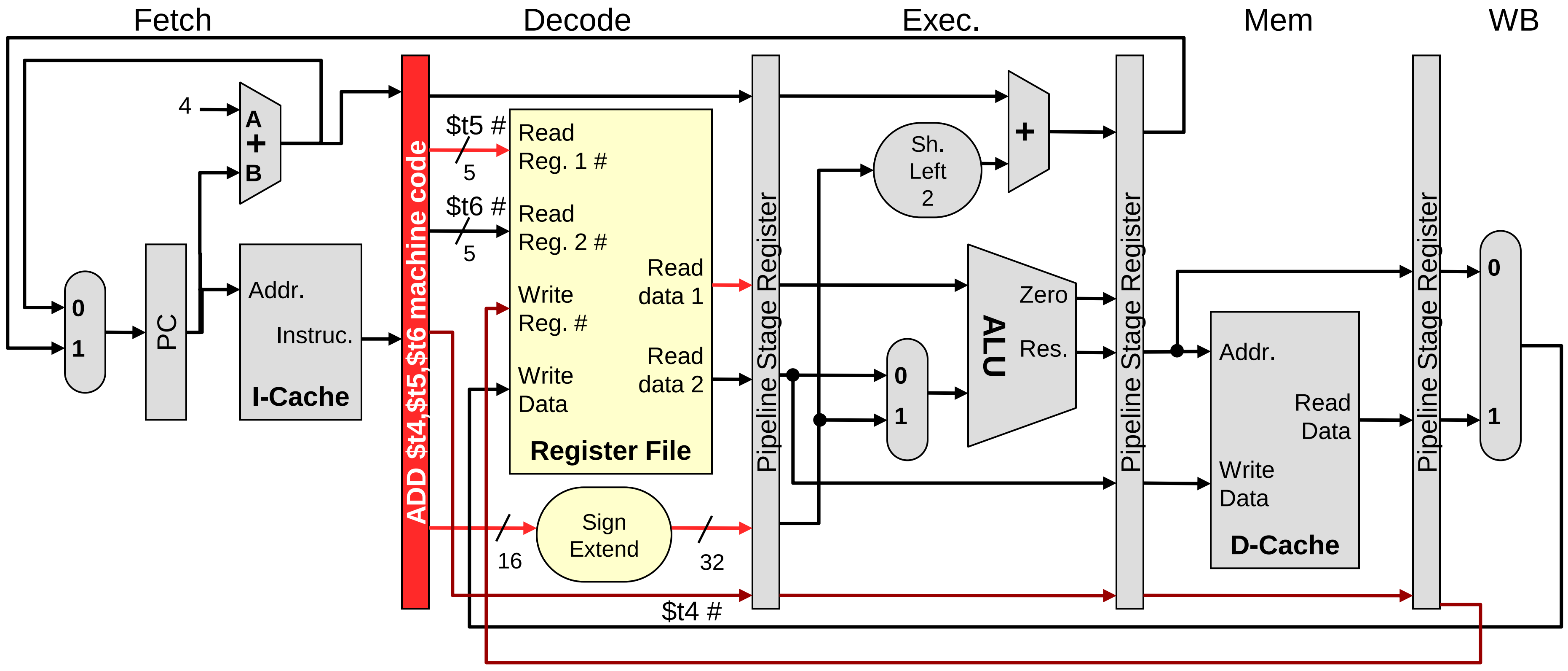
Write word to $t1

ADD $t4,$t5,$t6: Fetch



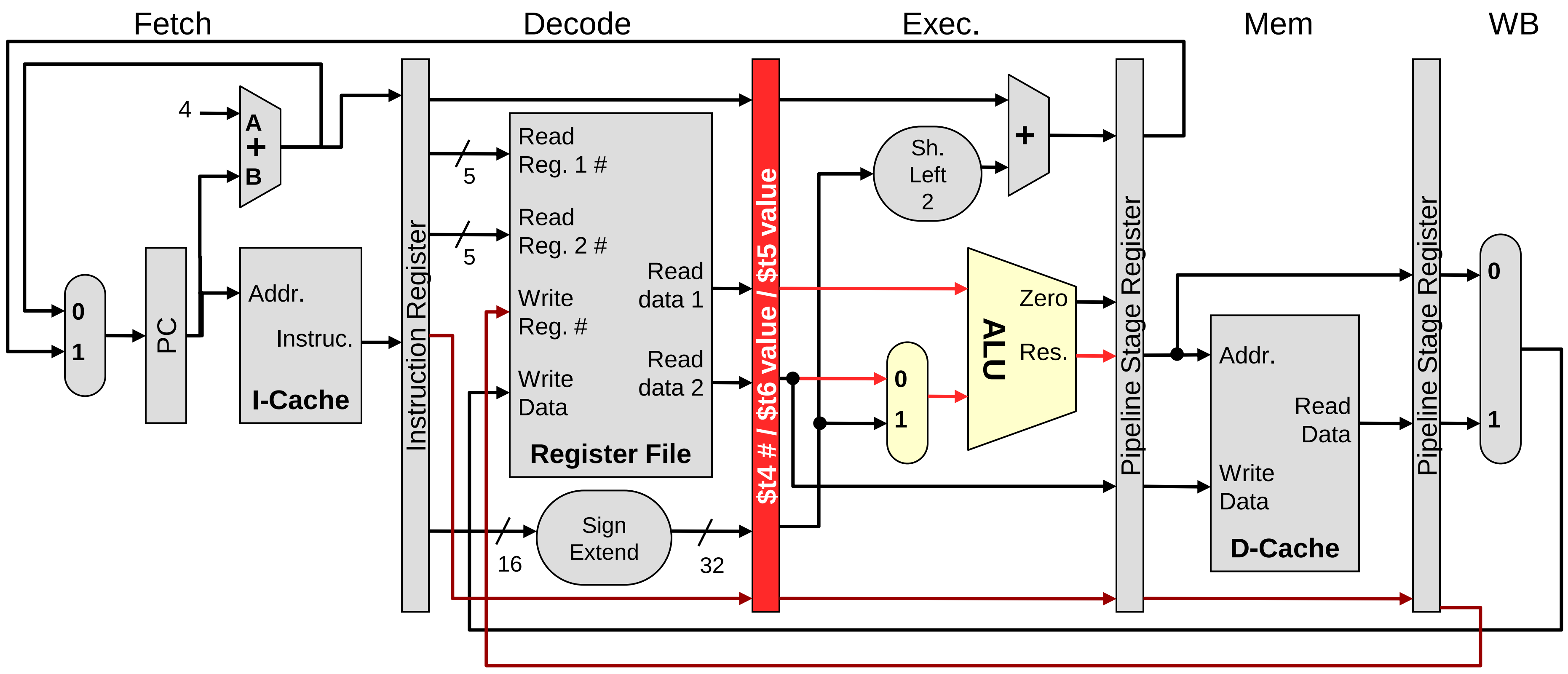
Fetch ADD and increment PC

ADD $t4,$t5,$t6: Decode



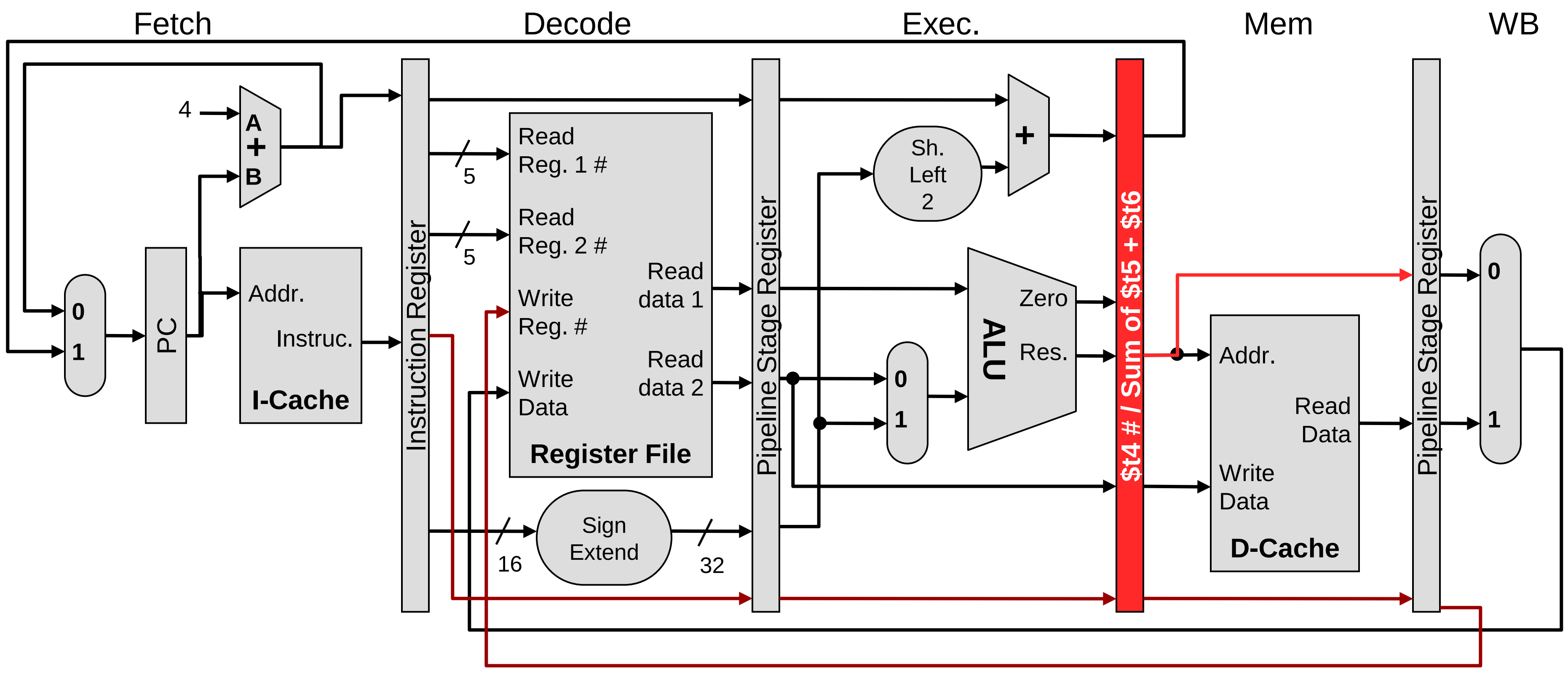
Decode instruction and fetch operands

ADD $t4,$t5,$t6: Execute



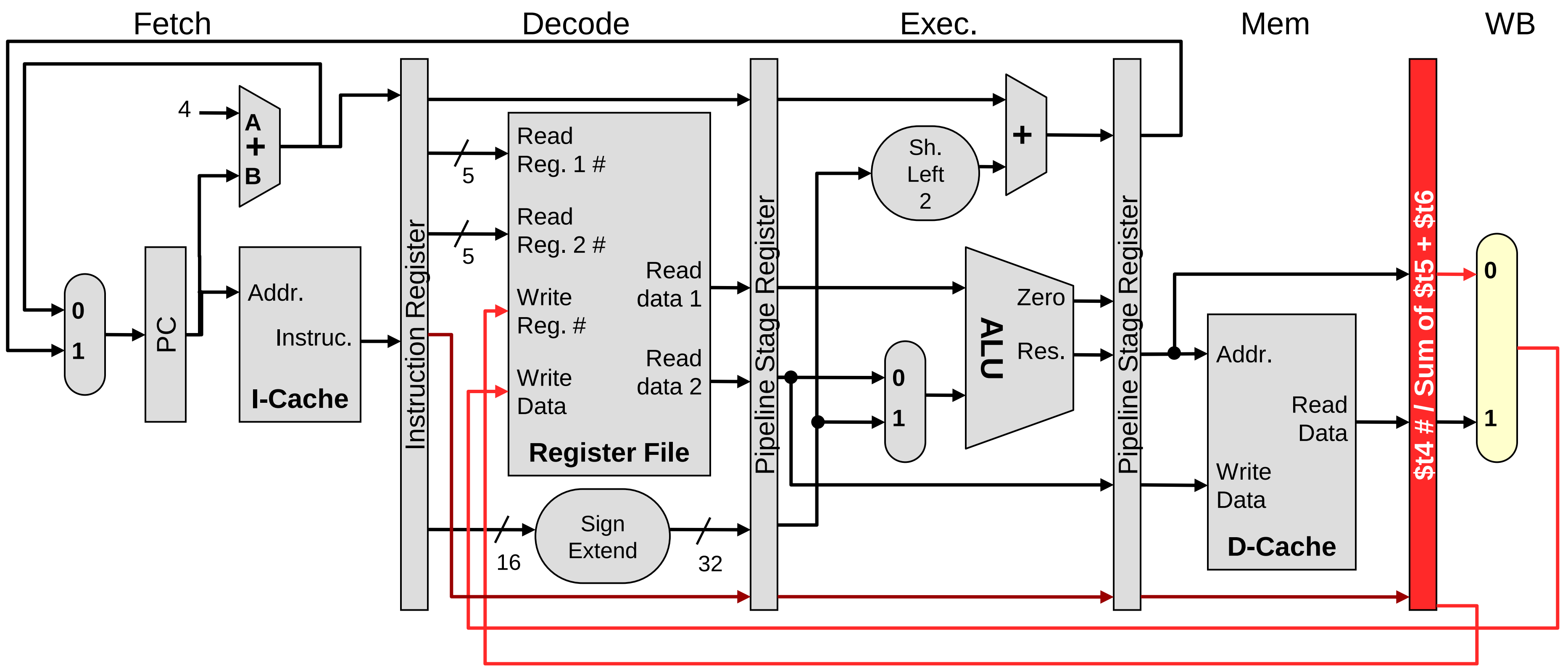
Add $t5 + $t6

ADD $t4,$t5,$t6: Memory



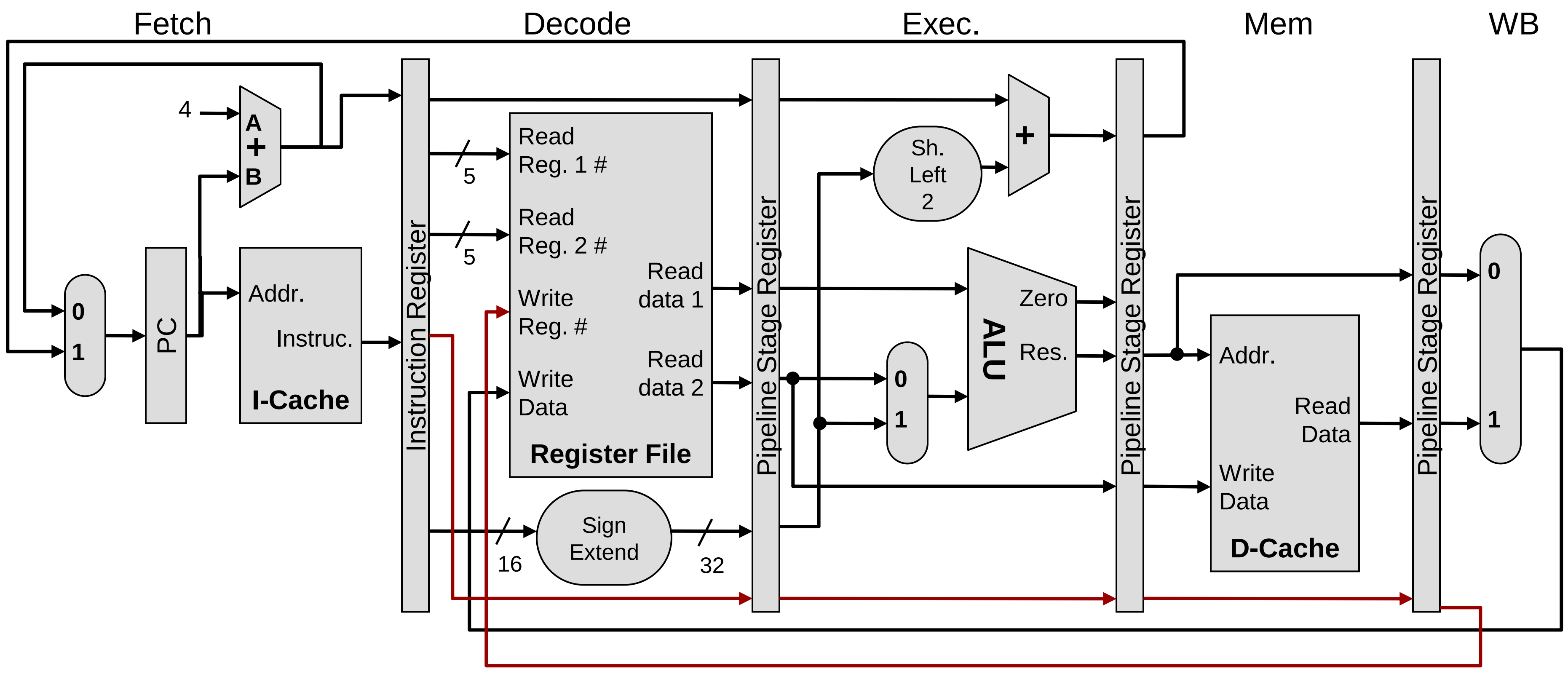
Just pass sum through

ADD $t4,$t5,$t6: Writeback



Write sum to $t4

ADD $t4,$t5,$t6



Fetch ADD

Decode instruction and fetch operands

Add $t5 + $t6

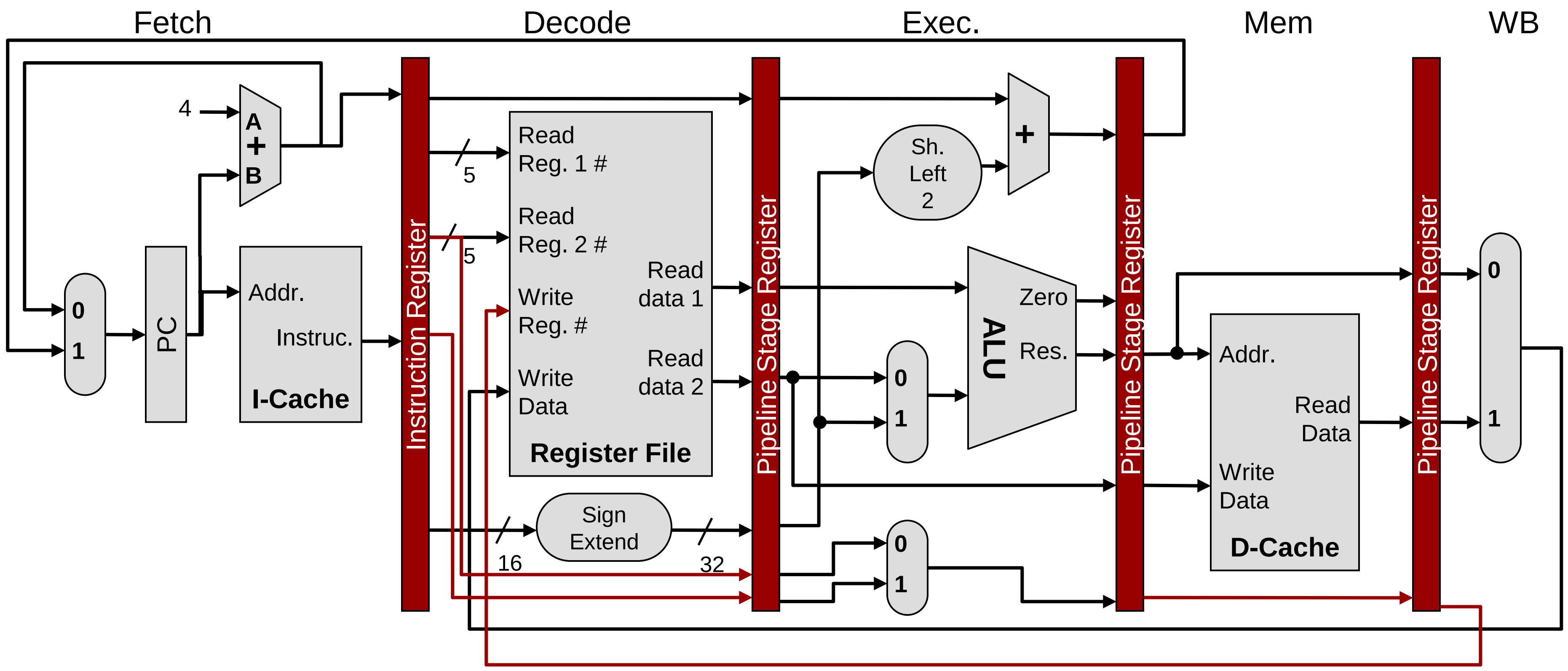
Just pass sum through

Write sum to $t4

OLD PIPELINING

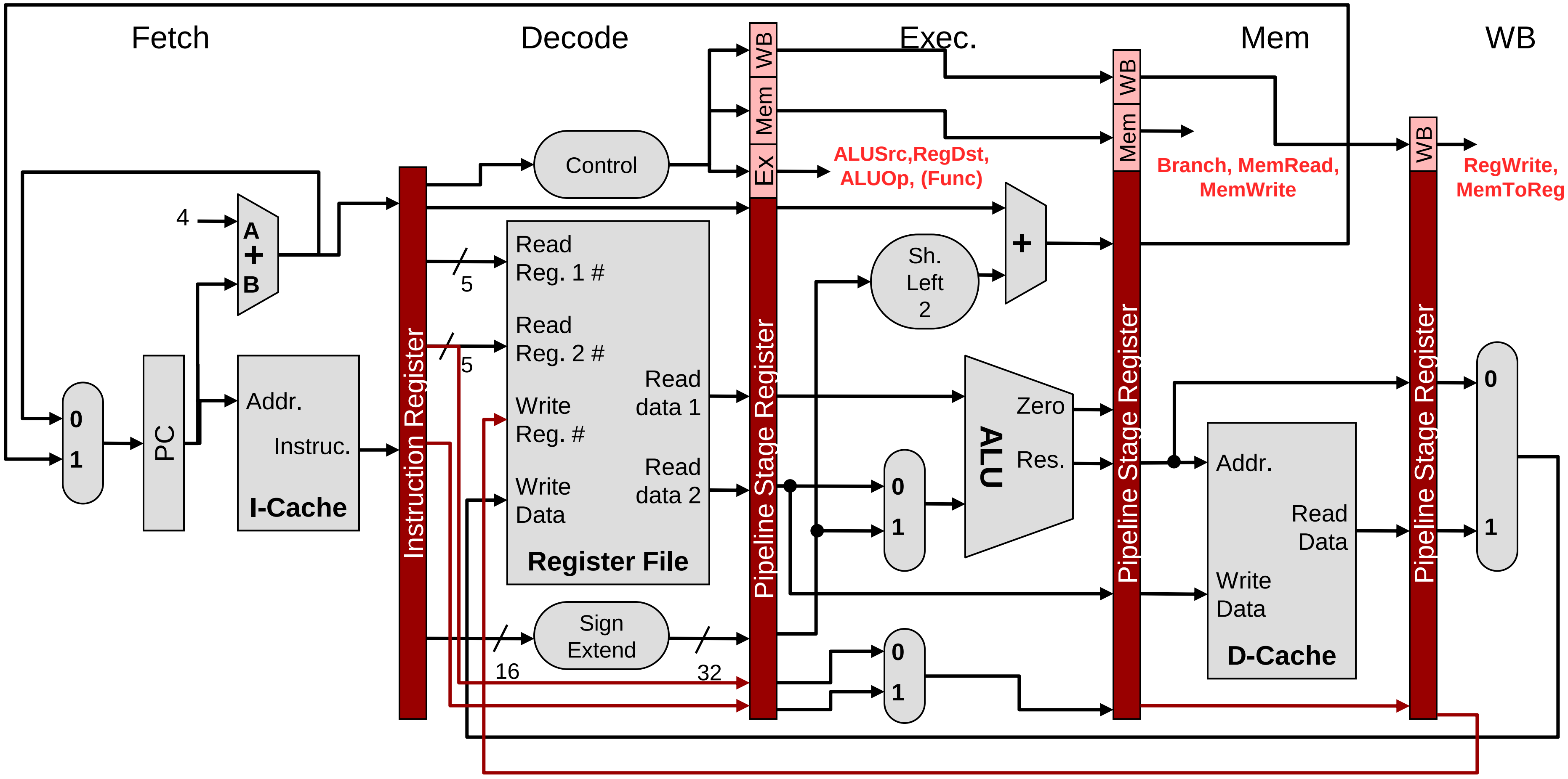
Basic 5 Stage Pipeline

- Control is generated in the decode stage and passed along to consuming stages through stage registers



Basic 5 Stage Pipeline

- Control is generated in the decode stage and passed along to consuming stages through stage registers


Fetch
Decode
Exec.
Mem
WB
Control
WB
Mem
Ex
ALUSrc,RegDst, ALUOp, (Func)
Branch, MemRead, MemWrite
RegWrite, MemToReg
4
A
+
B
0
1
PC
Addr.
Instruc.
I-Cache
Instruction Register
Read Reg. 1 #
Read Reg. 2 #
Write Reg. #
Write Data
Read data 1
Read data 2
Register File
5
5
Sign Extend
16
32
Pipeline Stage Register
Sh. Left 2
+
ALU
Zero
Res.
Addr.
Read Data
Write Data
D-Cache