University of Southern California

Viterbi School of Engineering

EE352
Computer Organization and Architecture

CPU Organization
Basic Pipelining Techniques

References:

1) Textbook
2) Mark Redekopp’s slide series

Shahin Nazarian

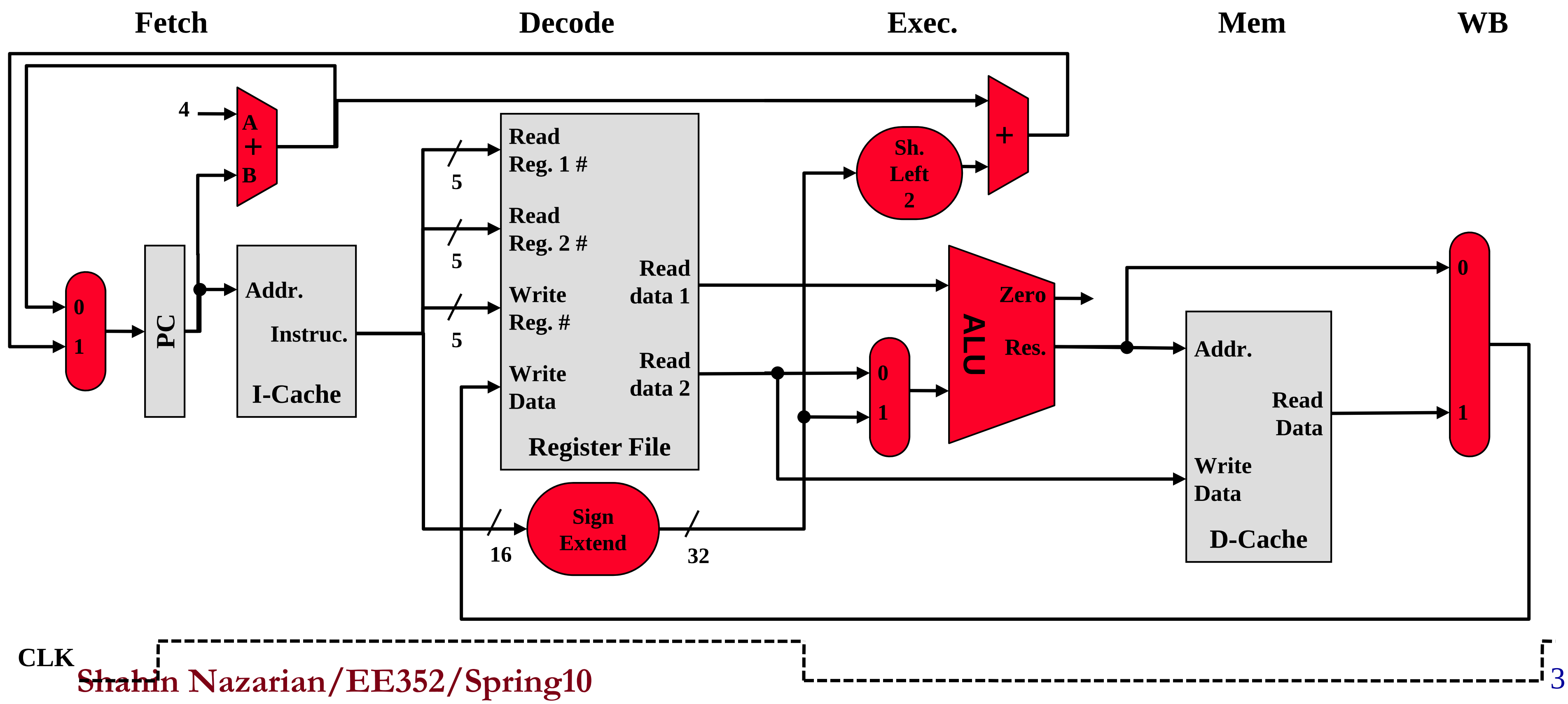
Spring 2010

Datapath Organization

- What components are needed and how should they be organized to fetch, decode, and execute instructions
  - We'll examine a simplified datapath to implement a subset of MIPS instructions (add, sub, and, or, xor, lw, sw, bne, beq)
- Datapath organization affects CPI

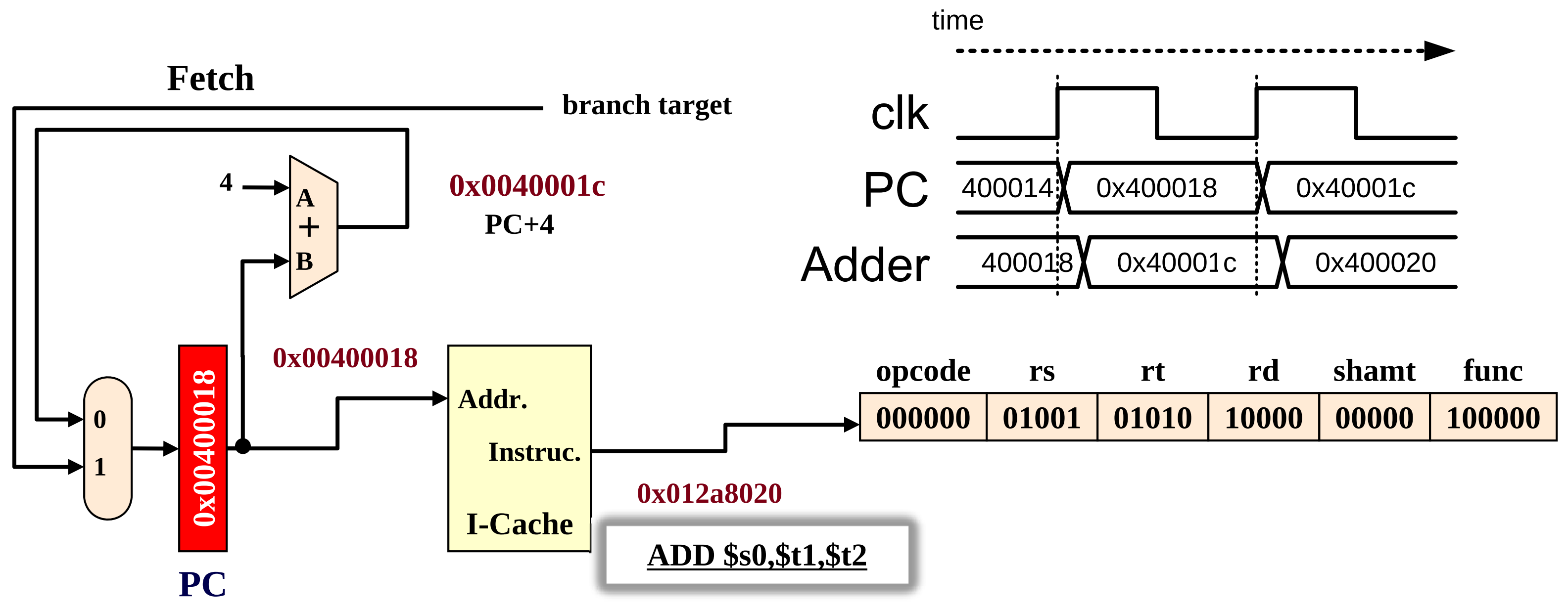
Single Cycle CPU Datapath

- Each instruction will execute in one LONG clock cycle
- To understand the whole datapath we'll walk through it in five phases (Fetch, Decode, Execute, Memory, Writeback)



Fetch

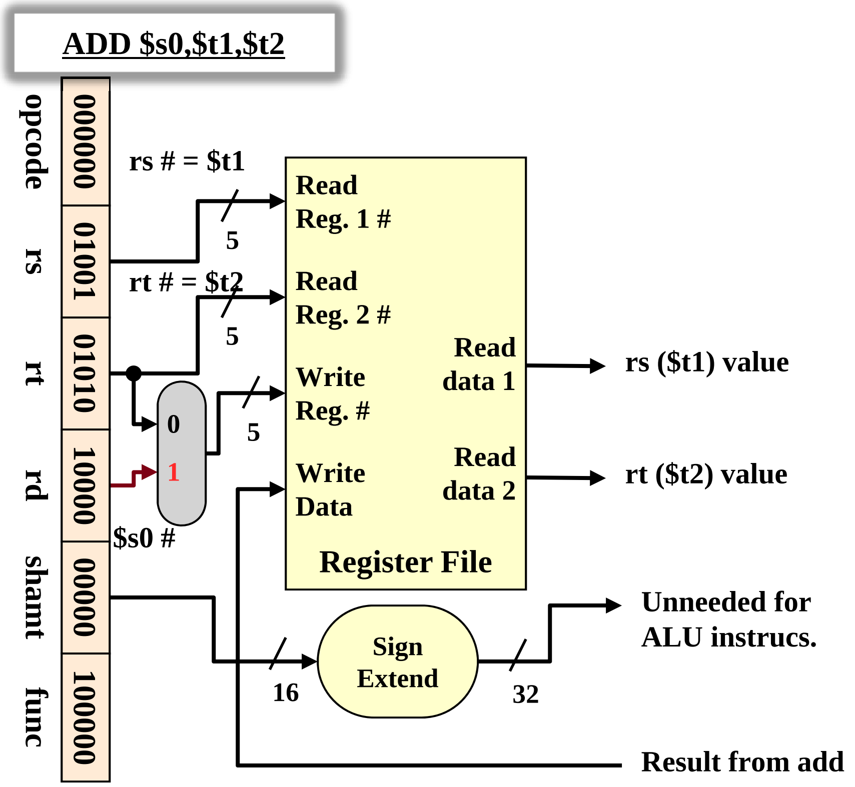
- Address in PC is used to fetch instruction while it is also incremented by 4 to point to the next instruction
- Remember, the PC doesn't update until the end of the clock cycle / beginning of next cycle
- Mux provides a path for branch target addresses

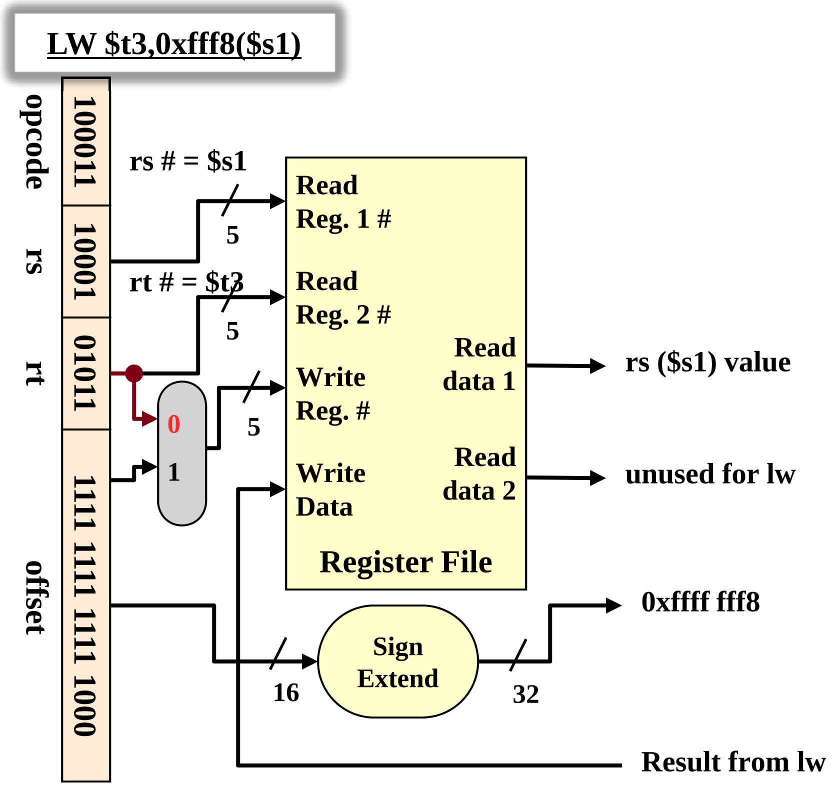


opcode	rs	rt	rd	shamt	func
000000	01001	01010	10000	00000	100000

Decode

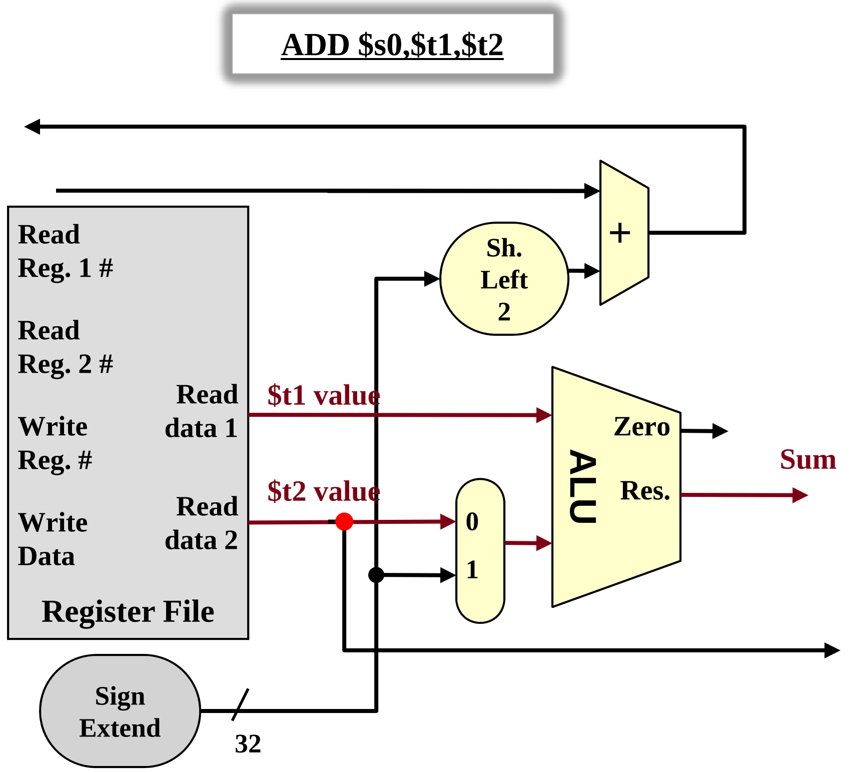
- Opcode is decoded (not shown)
- Fields of instruction are used to access registers and generate offsets

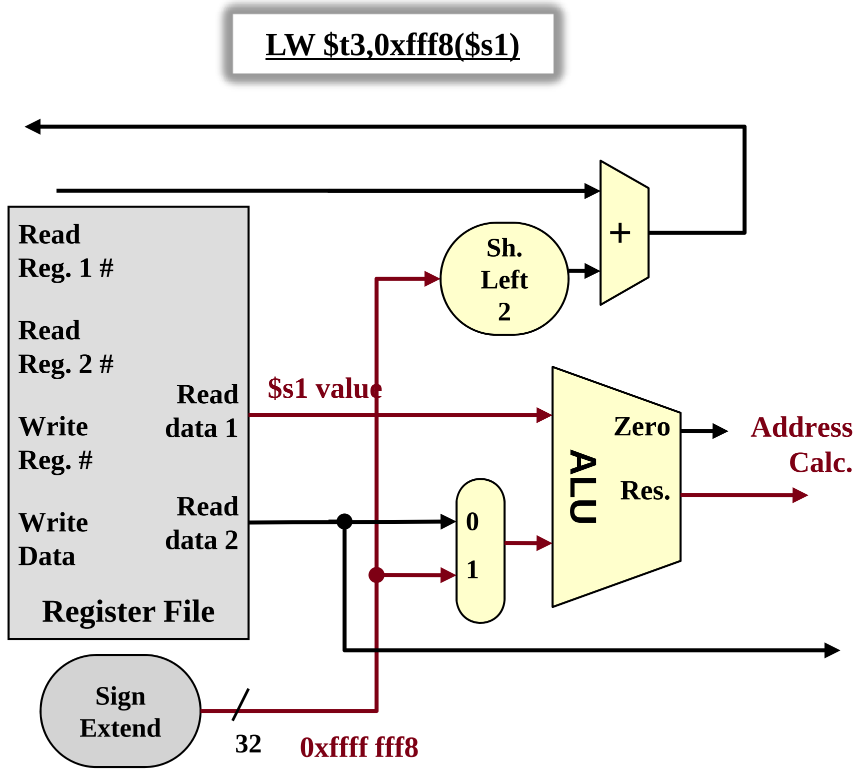




ALU (Exec.) Phase

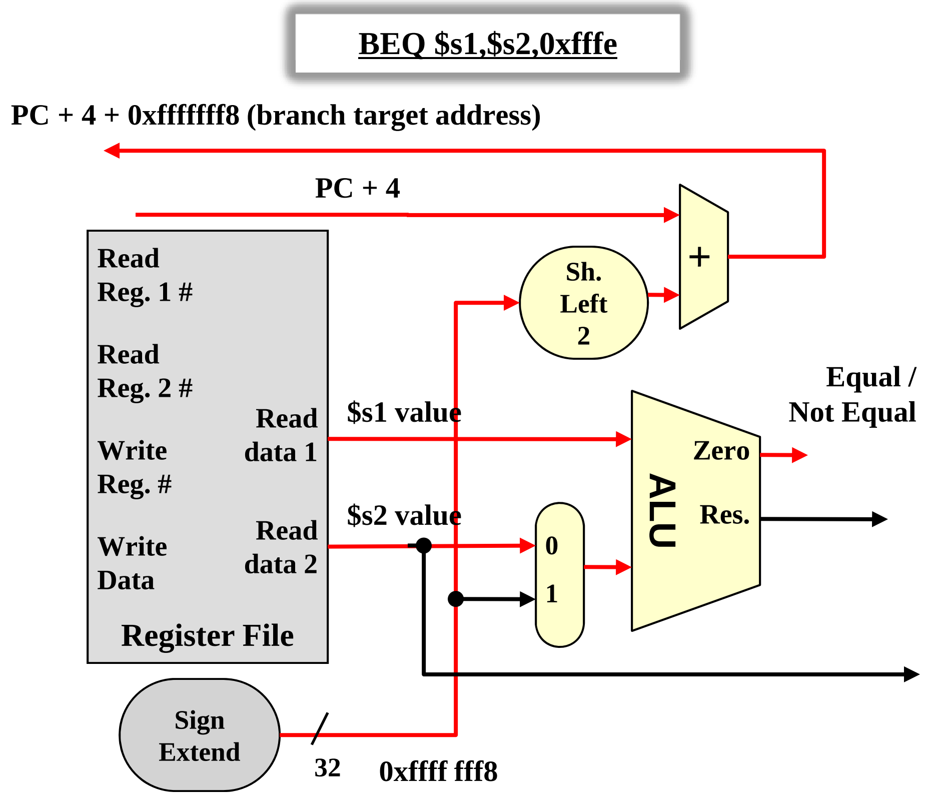
- ADD and LW/SW use the ALU to calculate result or address for LW/SW access





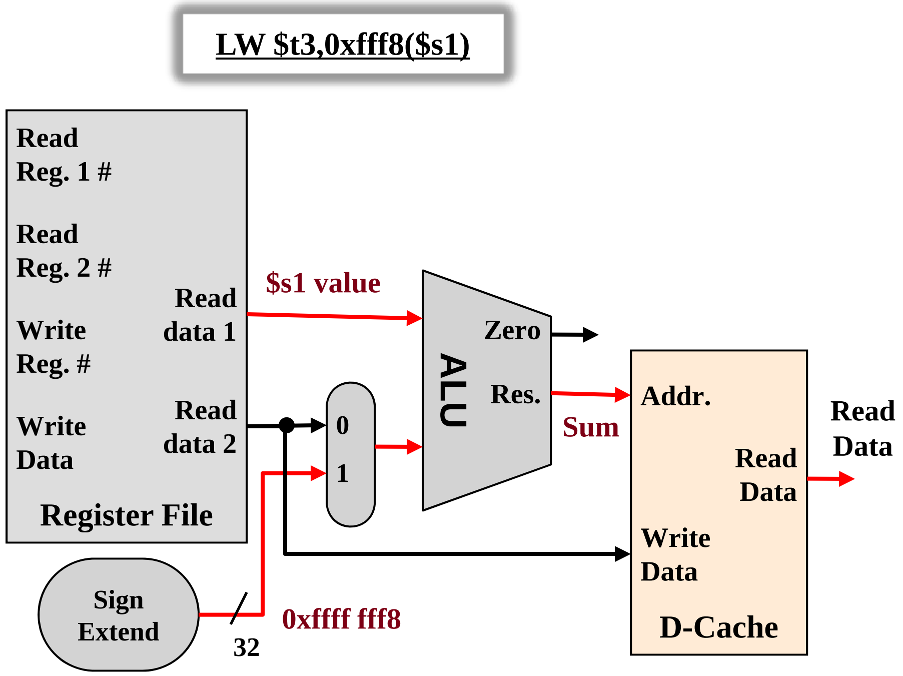
ALU (Exec.) Phase

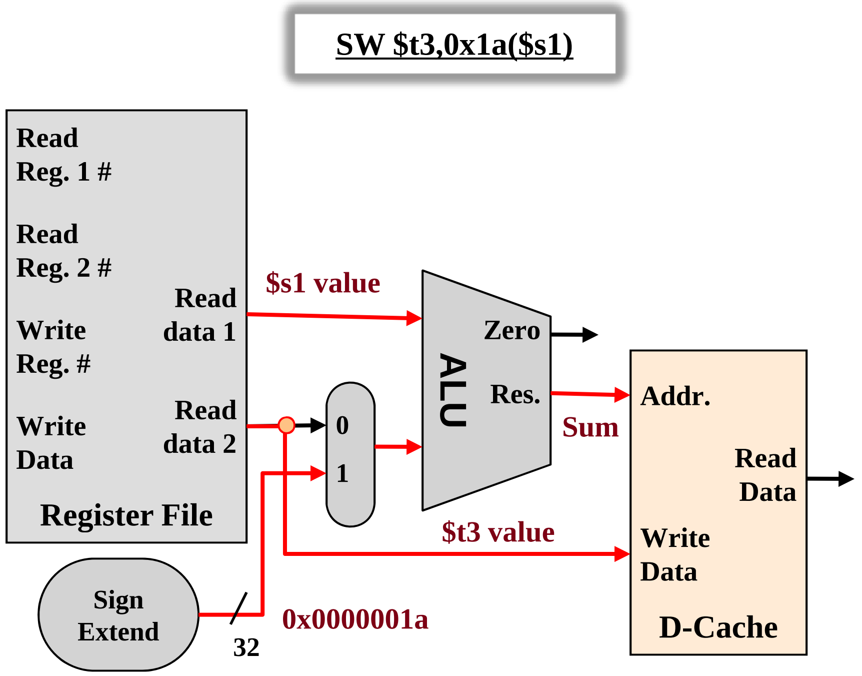
- Branch instructions compare the operands ($s1, $s2) and uses the separate adder to find the branch target address



Memory Phase

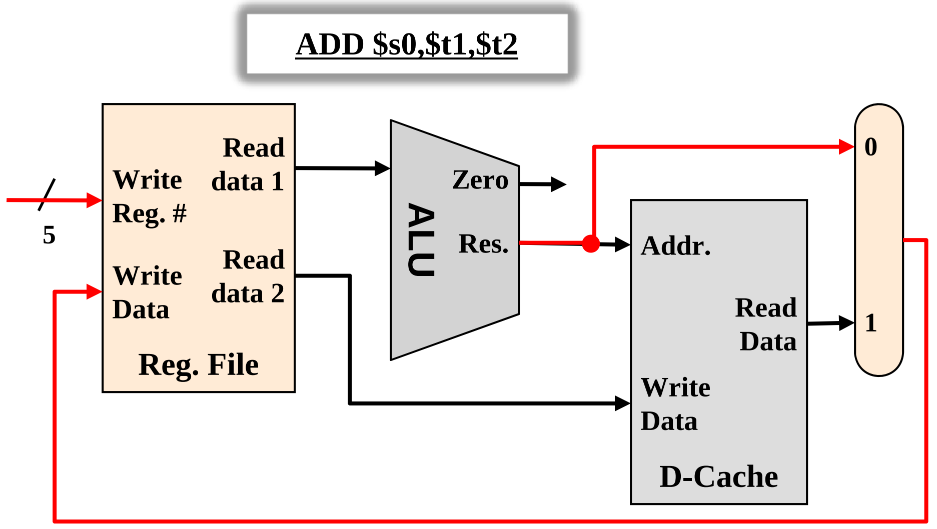
- LW uses address calculation from exec. phase and reads the data cache
- SW calculates address and uses data from register file to perform the write

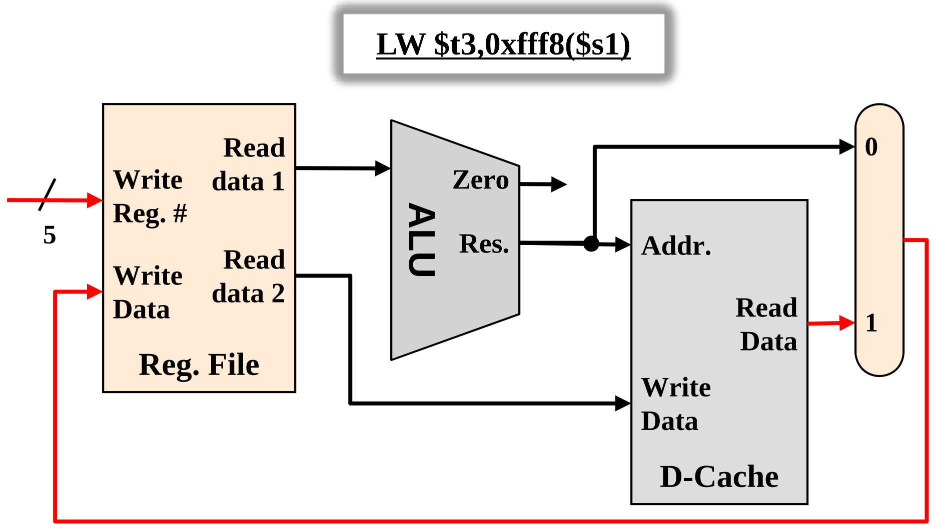




Writeback Phase

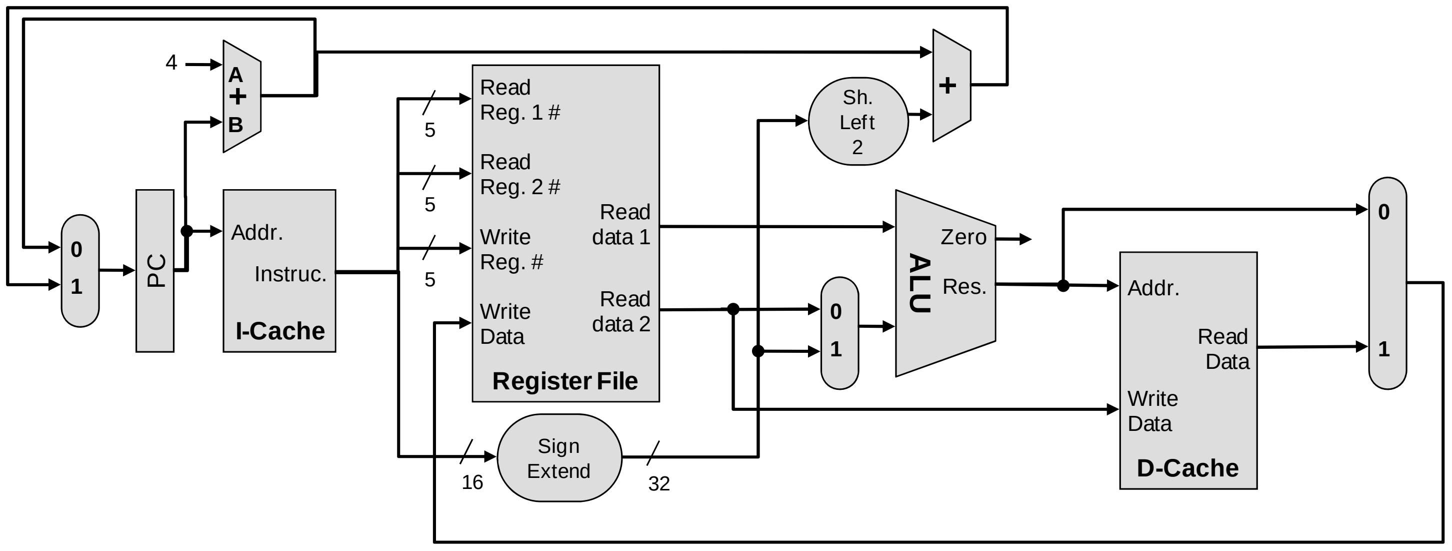
- Mux selects either the output of the ALU (for arithmetic and logic instructions) or the output of the data cache for LW instructions





Single-Cycle Performance

- Since we are making all instructions fit within a single clock cycle, we must set the cycle time equal to the LONGEST instruction
  - Probably a LW for our limited CPU (fetch, decode, address calc., data access, writeback)
  - What if cache miss?
- Each piece of the datapath requires only a small period of the overall instruction execution (clock cycle) time yielding low utilization of the HW's actual capabilities



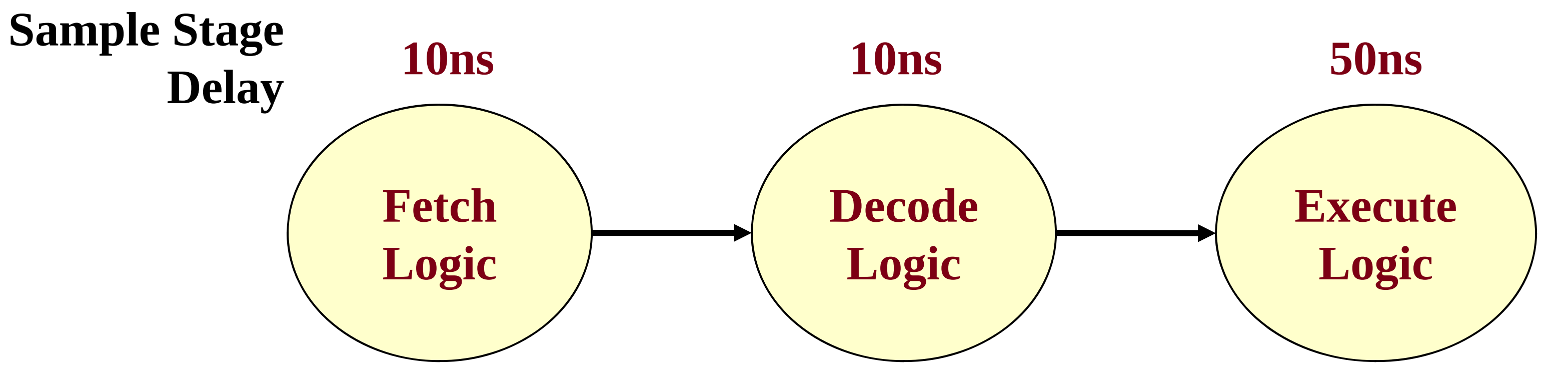
Pipelining

- Overlapped execution of multiple instructions
- Natural breakdown into stages
  - Fetch, Decode, Execute
- Fetch an instruction, while decoding another, while executing another

	F	D	E
Clock 1	Inst. 1		
Clock 2	Inst. 2	Inst. 1	
Clock 3	Inst. 3	Inst. 2	Inst. 1
Clock 4	Inst. 4	Inst. 3	Inst. 2
Clock 5	Inst. 5	Inst. 4	Inst. 3

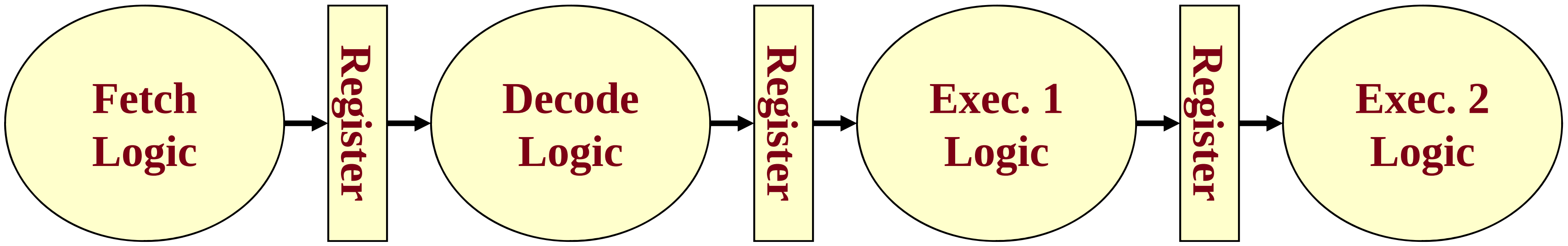
Issues with Pipelining

- No sharing of HW/logic resources between stages
  - Can't have a single cache (both I & D) because each is needed to fetch one instruction while another accesses data]
- Prevent signals in one stage (instruc.) from flowing into another stage (instruc.) and becoming convoluted
- Balancing stage delay
  - Clock period = longest stage
  - In example below, clock period = 50ns means 150ns delay for only 70ns of logic delay



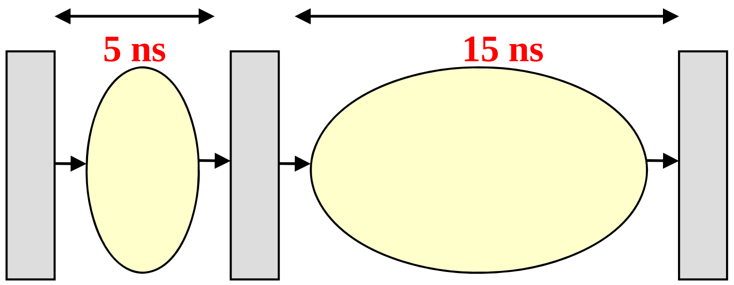
Resolution of Pipelining Issues

- No sharing of HW/logic resources between stages
  - For full performance, no feedback (stage i feeding back to stage i-k)
  - If two stages need a HW resource, replicate the resource in both stages (e.g. an I- AND D-cache)
- Prevent signals from one stage (instruc.) from flowing into another stage (instruc.) and becoming convoluted
  - Place registers between stages (outputs change once per clock)
  - Registers act as barrier wall to signals until next edge
- Balancing stage delay [Important!!!]
  - Balance or divide long stages (See next slides)

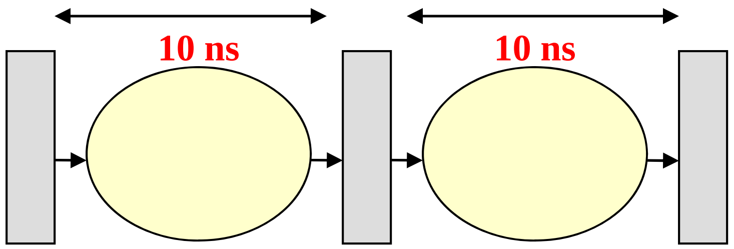


Balancing Pipeline Stages

- Clock period must equal the *LONGEST* delay from register to register
  - In Example 1, clock period would have to be set to 15ns [66 MHz], meaning total time through pipeline = 30ns for only 20 ns of logic
- Could try to balance delay in each stage
  - Example 2: Clock period = 10ns [100 MHz], meaning total time through pipeline = 20ns



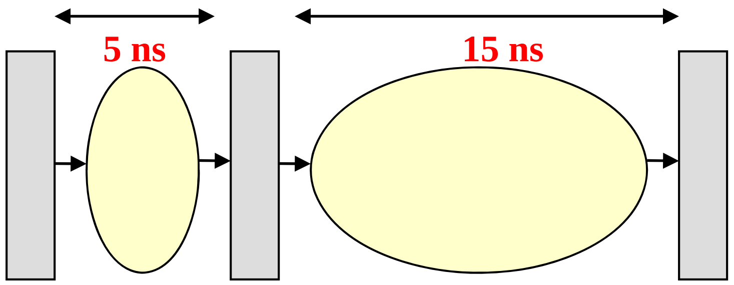
Ex. 1: Unbalanced stage delay
Clock Period = 15ns



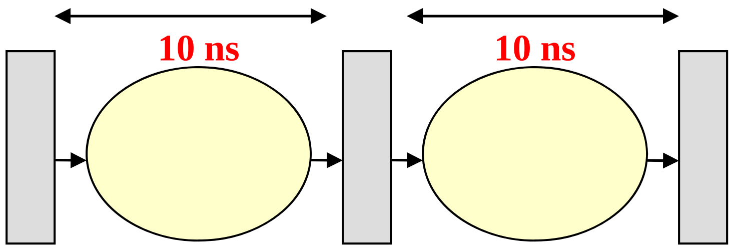
Ex. 2: Balanced stage delay
Clock Period = 10ns (150% speedup)

Pipelining Effects on Clock Period

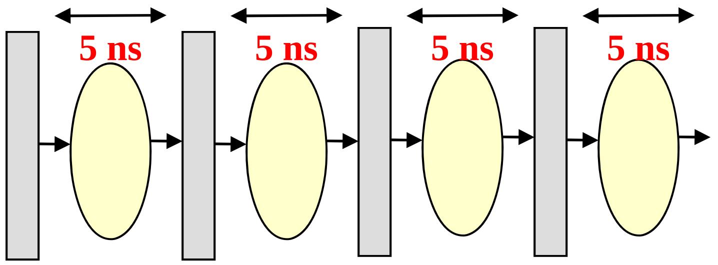
- Rather than just try to balance delay we could consider making more stages
  - Divide long stage into multiple stages
  - In Example 3, clock period could be 5ns [200 MHz]
  - Time through the pipeline (latency) is still 20 ns, but we've doubled our throughput (1 result every 5 ns rather than every 10 or 15 ns)
  - Note: There is a small time overhead to adding a pipeline register/stage (i.e. can't go crazy adding stages)



Ex. 1: Unbalanced stage delay
Clock Period = 15ns



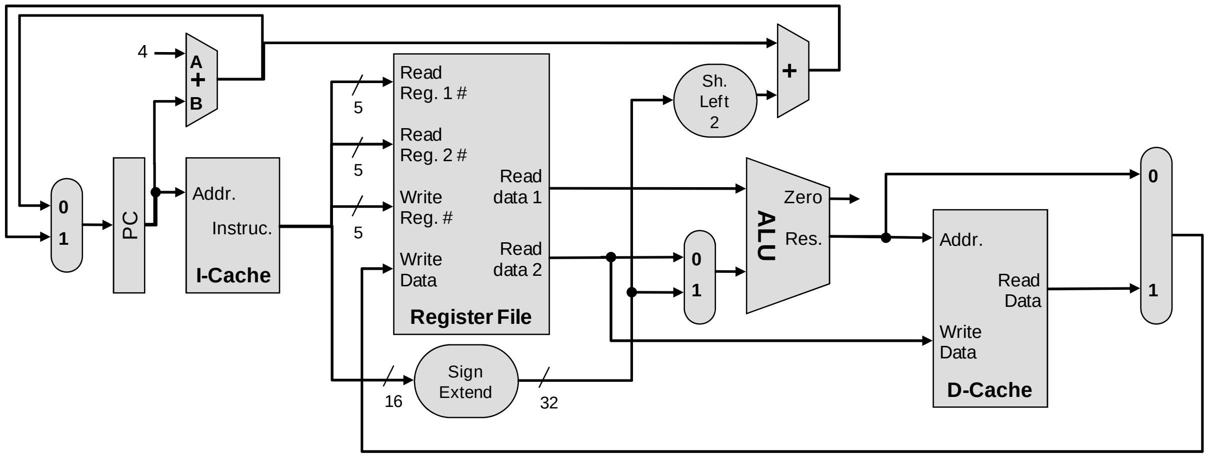
Ex. 2: Balanced stage delay
Clock Period = 10ns (150% speedup)



Ex. 3: Break long stage into multiple stages
Clock period = 5 ns (300% speedup)

Feed-Forward Issues

- CISC (Complex Instruction Set Computer) instructions often perform several ALU and memory operations per instructions
  - MOVE.W (A0)+,$8(A0,D1) [M68000/Coldfire ISA]
    - 3 Adds (post-increment, disp., index)
    - 4 Memory operations (2 I-Fetch + 1 read + 1 write)
  - This makes pipelining hard because of multiple uses of ALU and memory
- Redesign the Instruction Set Architecture to better support pipelining (MIPS was designed with pipelining in mind)

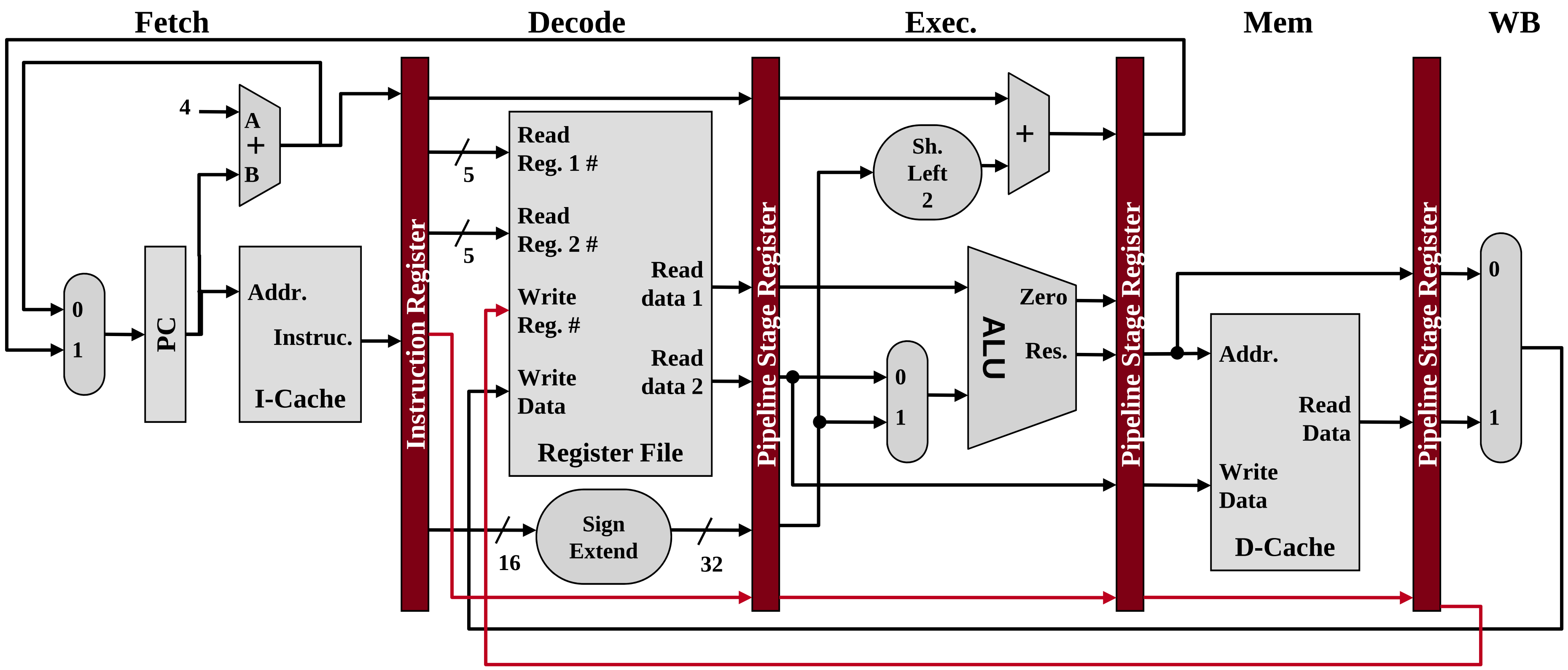


Sample 5-Stage Pipeline

- Examine the basic operations that need to be performed by our instruction classes
  - LW: I-Fetch, Decode/Reg. Fetch, Address Calc., Read Mem., Write to Register
  - SW: I-Fetch, Decode/Reg. Fetch, Address Calc., Write Mem.
  - ALUop: I-Fetch, Decode/Reg. Fetch, ALUop, Write to Reg.
  - Bxx: I-Fetch, Decode/Reg. Fetch, Compare (Subtract), Update PC
- These suggest a 5-stage pipeline:
  - I-Fetch, Decode/Reg. Fetch, ALU (Exec.), Memory, Reg. Writeback

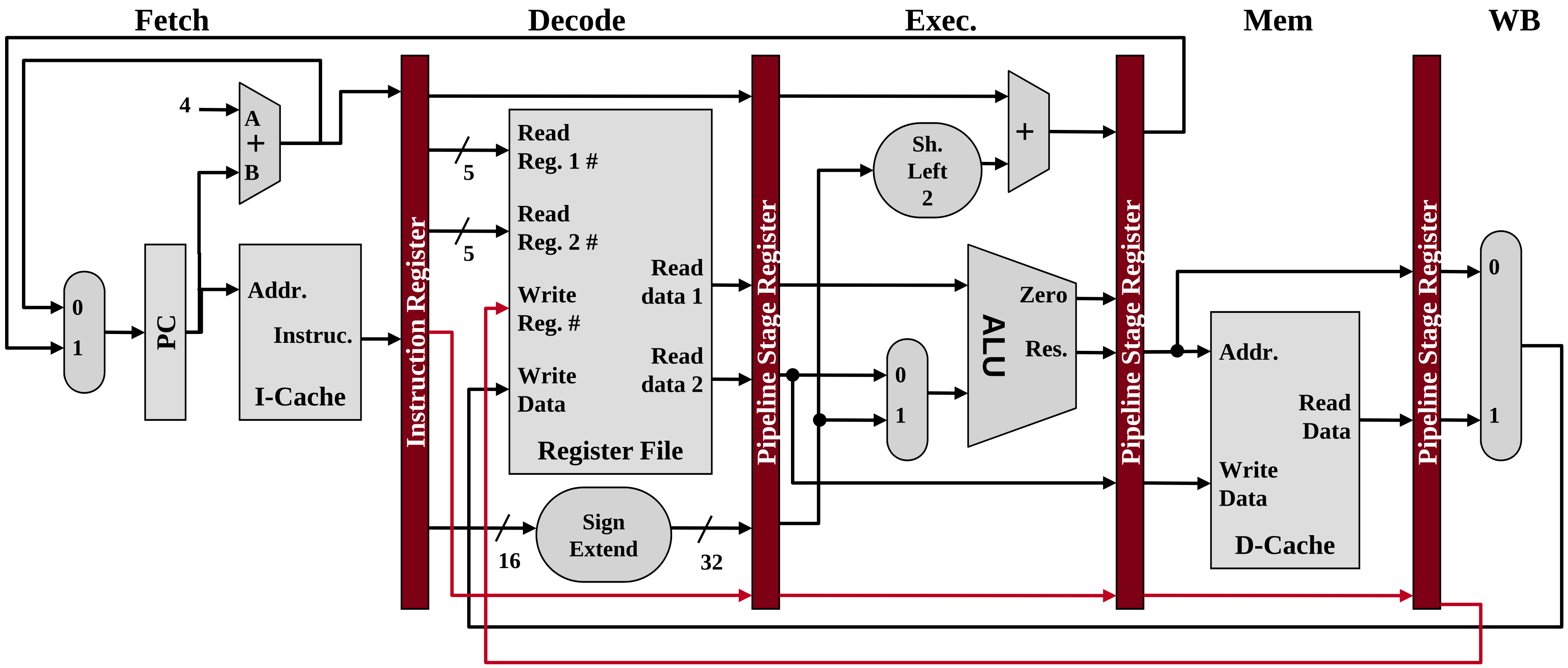
Basic 5 Stage Pipeline

- Same structure as single cycle but now broken into 5 stages
- Pipeline stage registers store intermediate outputs of each stage for the next stage and act as a barrier from signals from different stages intermixing



Basic 5 Stage Pipeline

- All control signals needed for an instruction in the following stages are generated in the decode stage and shipped with the instruction
  - Since writeback doesn't occur until final stage, write register # is shipped with the instruction through the pipeline and then used at the end
  - Register File can read out the current data being written if read reg # = write reg #

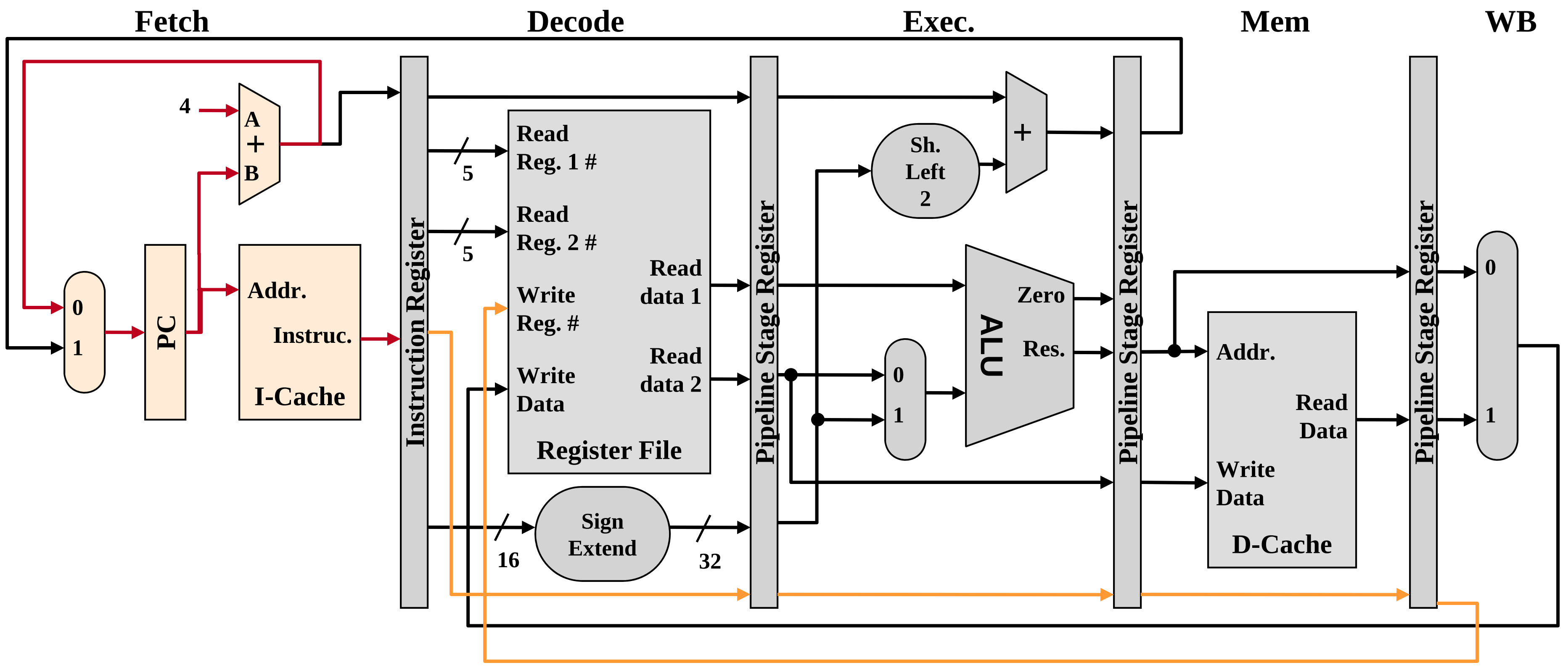


Sample Instructions

Instruction
LW $t1,4($s0)
ADD $t4,$t5,$t6
BEQ $a0,$a1,LOOP

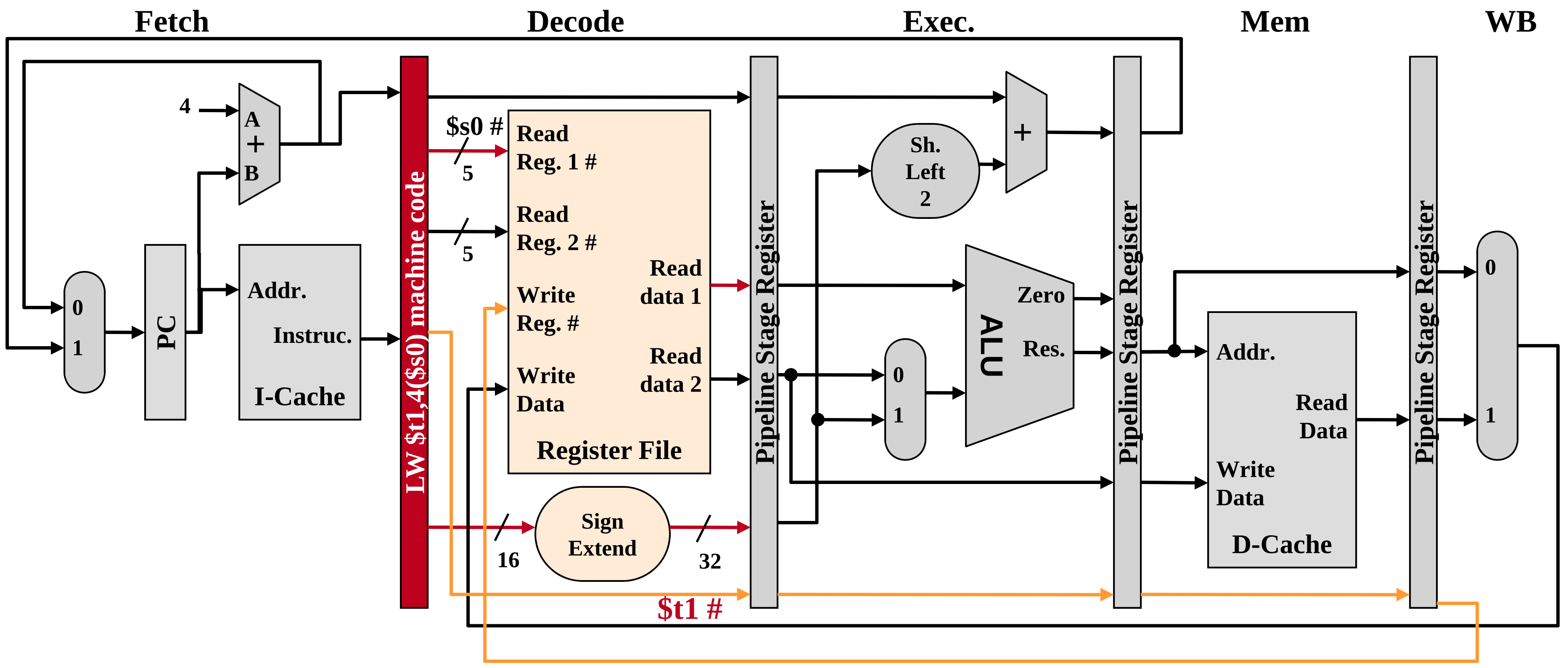
For now let’s assume we just execute one at a time though that’s not how a pipeline works (multiple instructions are executed at one time)

LW $t1,4($s0): Fetch



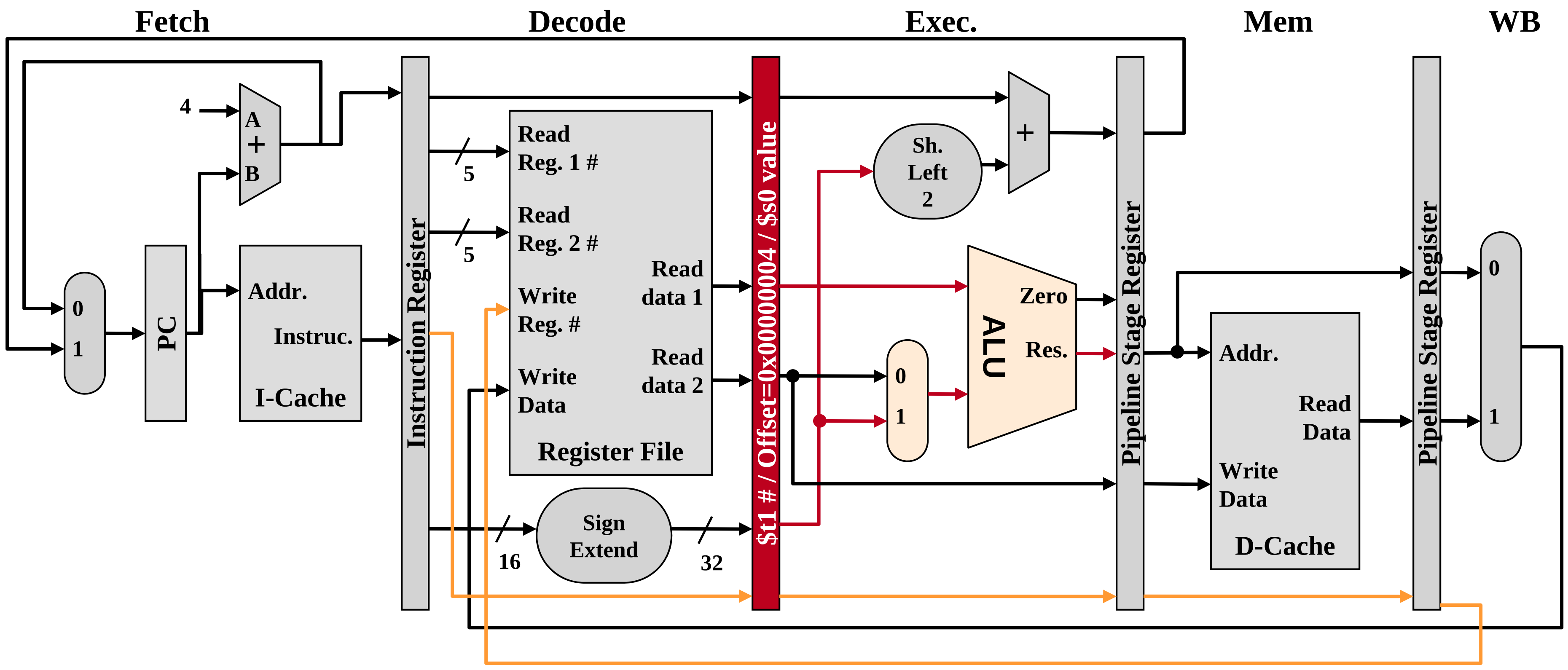
Fetch LW and increment PC

LW $t1,4($s0): Decode



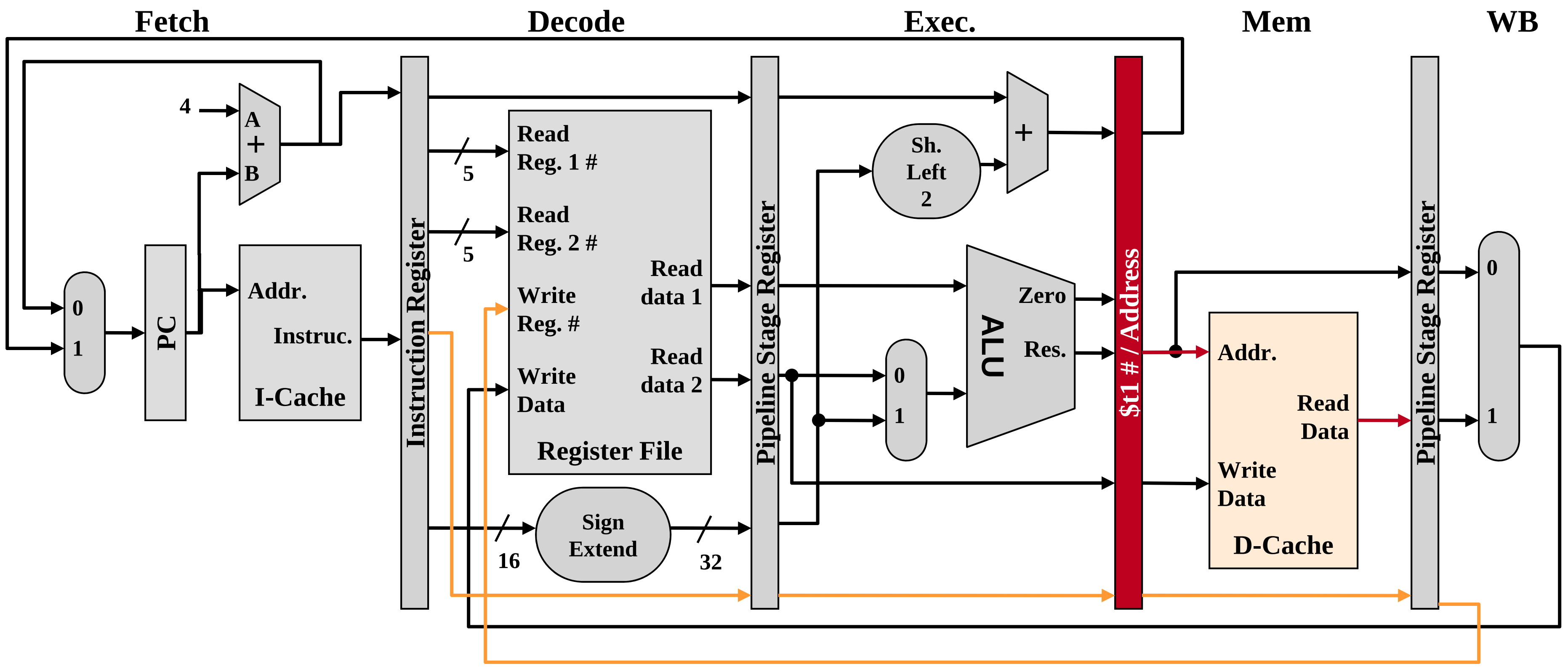
Decode instruction and fetch operands

LW $t1,4($s0): Execute



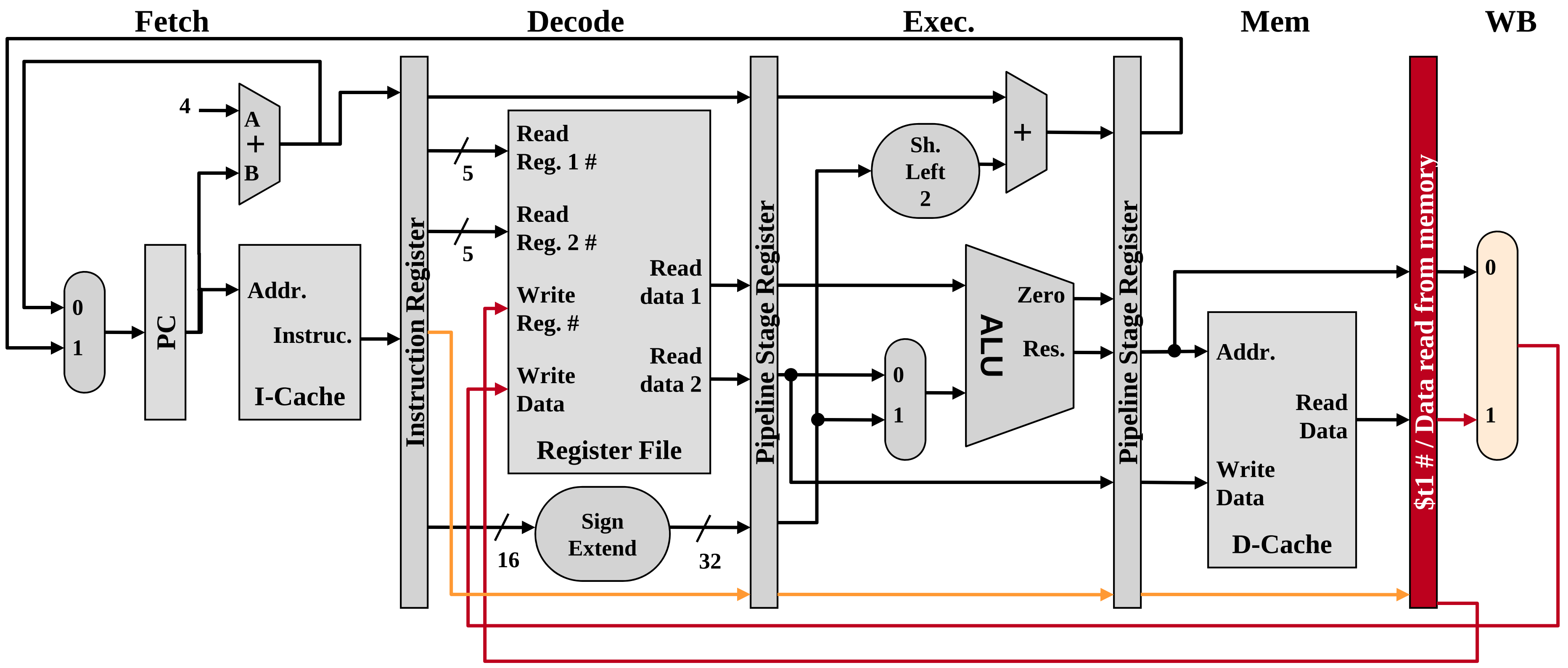
Add offset 4 to
$s0 value

LW $t1,4($s0): Memory



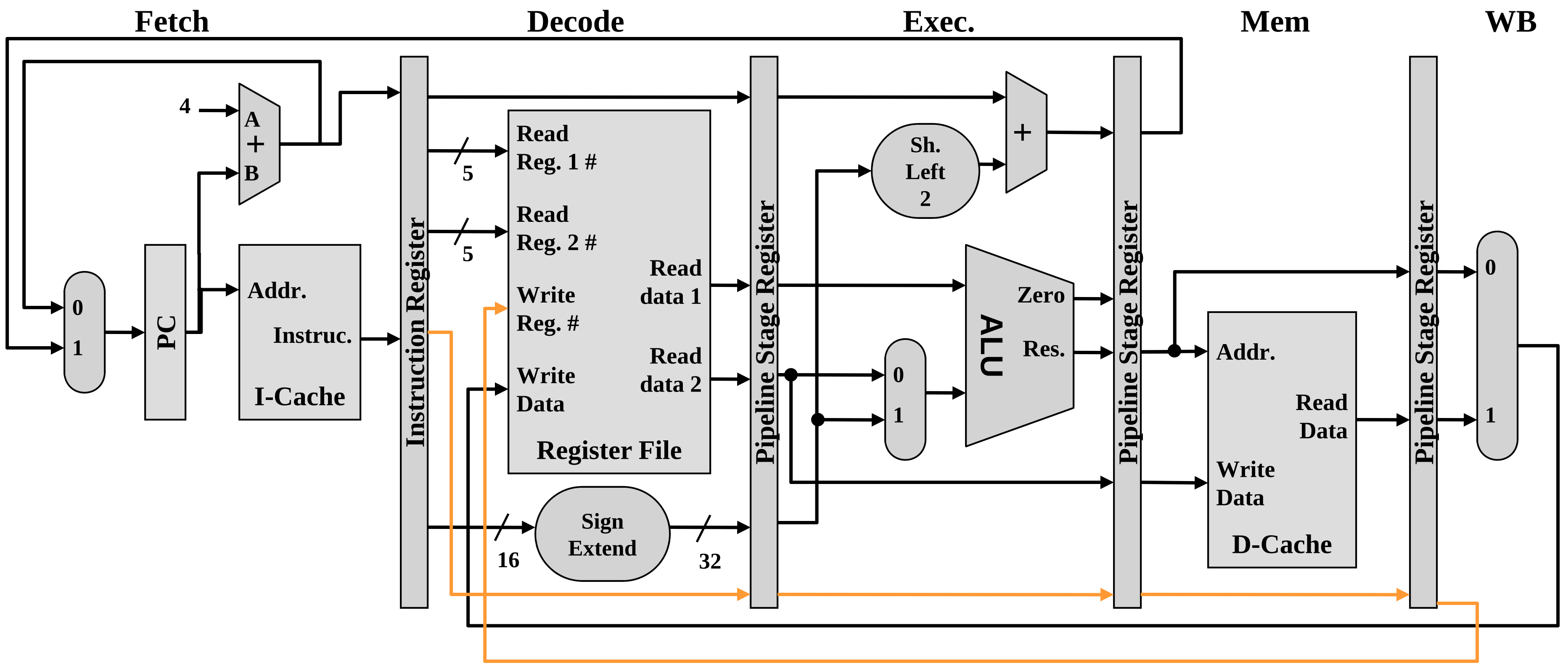
Read word from memory

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