Chapter 4: The Processor

Ngo Lam Trung

(trungnl@soict.hut.edu.vn)

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK
And M.J. Irwin's presentation, PSU 2008]

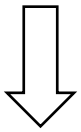
Review: MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands
- Good design demands good compromises
  - three instruction formats

Review

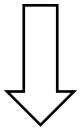
High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Assembly language program (for MIPS)

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```



Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000011000000110000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Performance metric

CPU time = CPI * CC * IC

CPI: cycle per instruction
CC: clock cycle
IC: instruction count

How to improve?

- IC:
- CC:
- CPI:

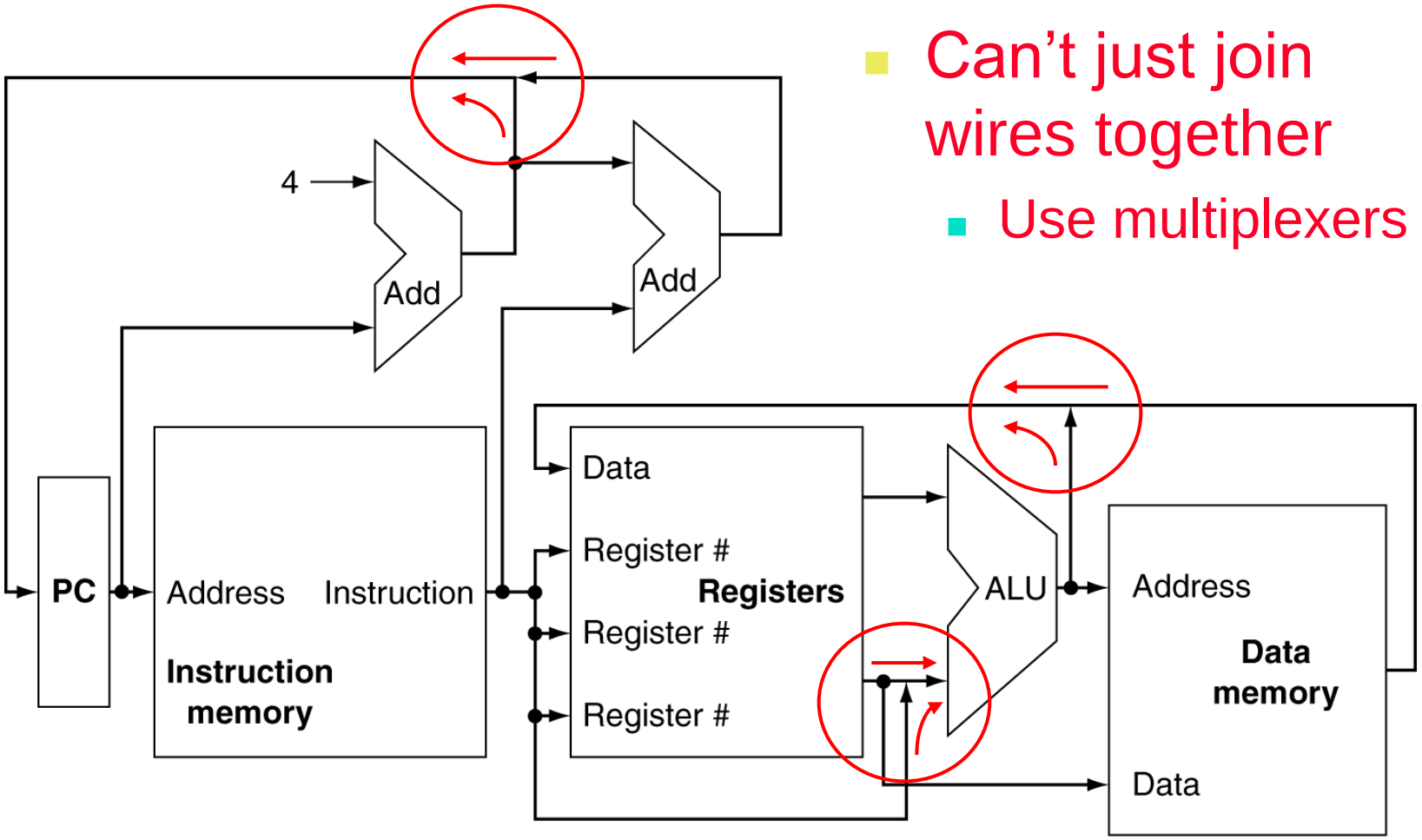
In this chapter

- Implementation of data path
- How to get CPI < 1

Overview

- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`
- Implementation of real CPU with other instructions are similar to the simplified version (theoretically!)

CPU Overview

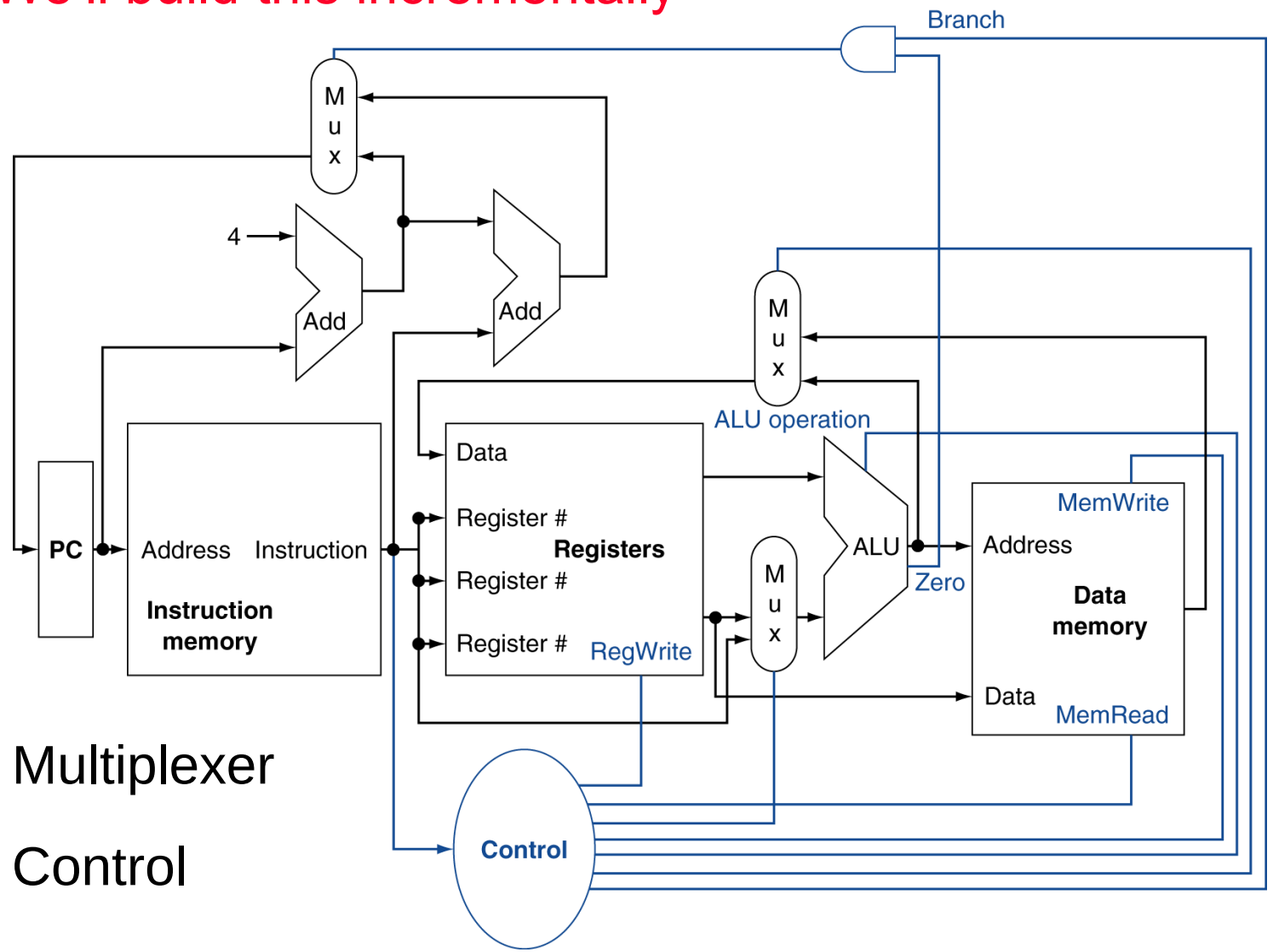


- Can't just join wires together
  - Use multiplexers

- Simple MIPS implementation

CPU implementation with MUXes and Control

We'll build this incrementally



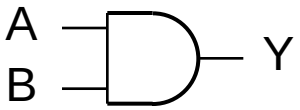
- Multiplexer
- Control

Logic Design Basics

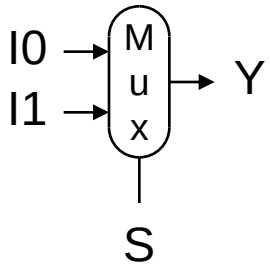
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

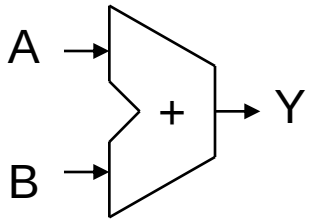
- AND-gate
  - Y = A & B



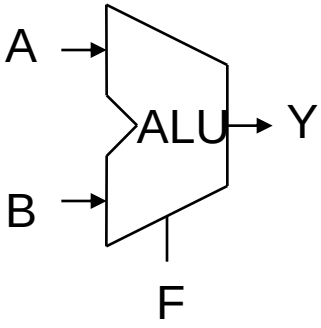
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

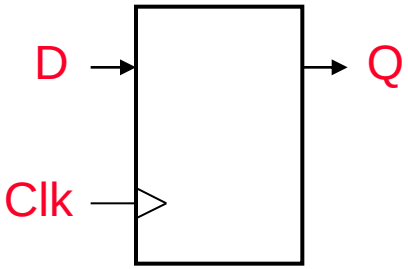


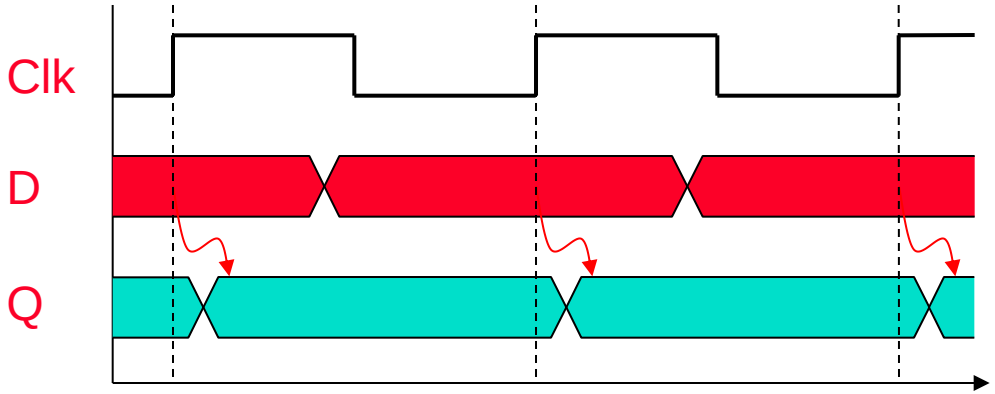
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

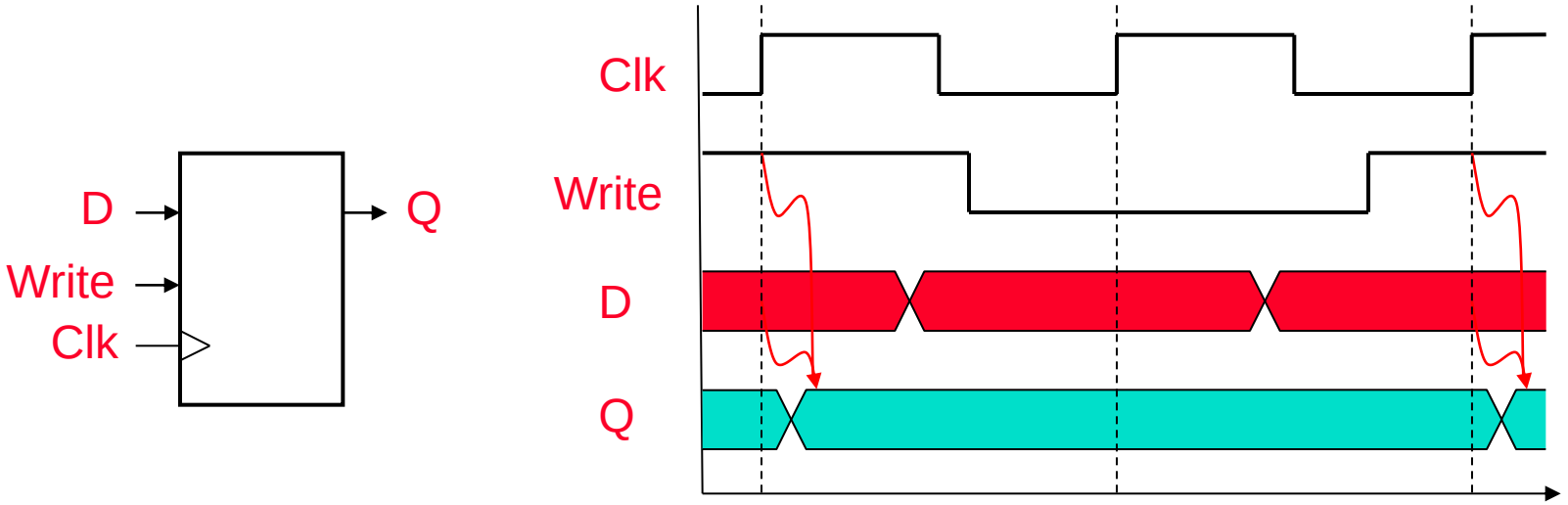




Sequential Elements

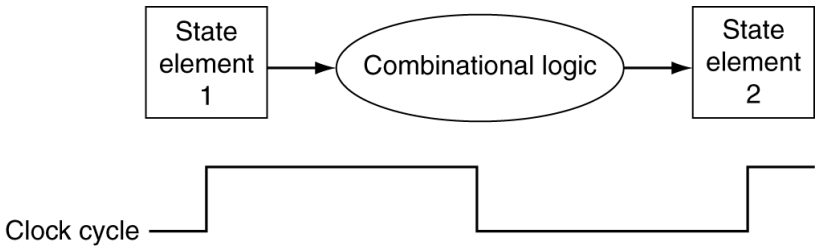
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

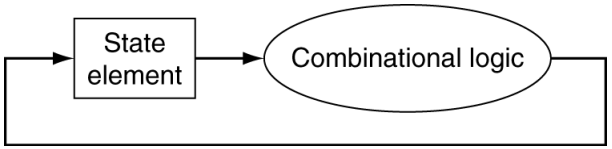
➔ assumption: Data is clocked in with CLK automatically when there is no explicit Write control signal



Clocking Methodology

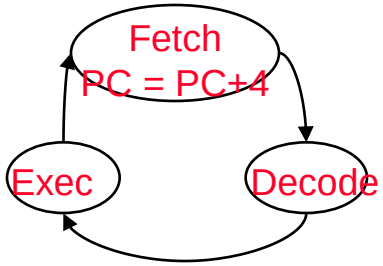
- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period
- State (sequential) elements: require clock





General instruction cycle

- Simplified MIPS ISA
  - memory-reference instructions: **`lw, sw`**
  - arithmetic-logical instructions: **`add, sub, and, or, slt`**
  - control flow instructions: **`beq, j`**
- Generic implementation
  - use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
  - decode the instruction (and read registers)
  - execute the instruction

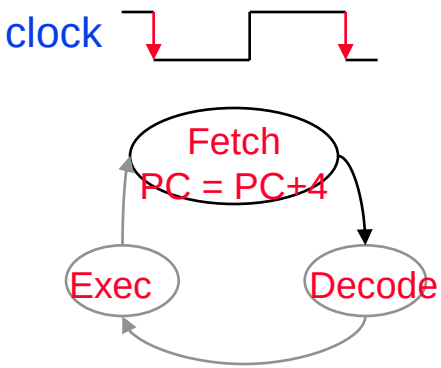


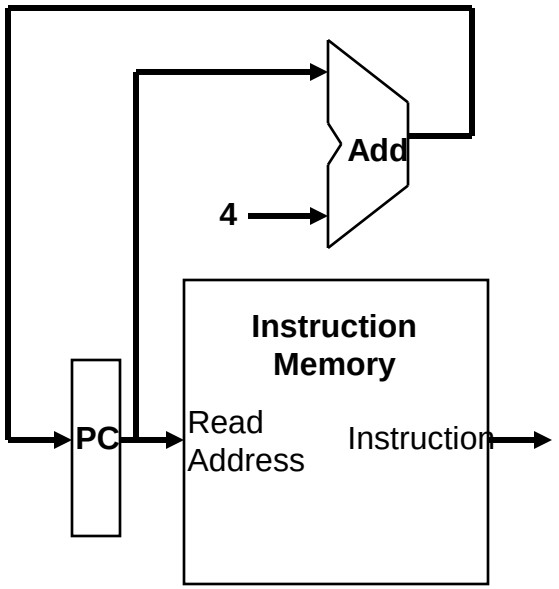
- All instructions (except **`j`**) use the ALU after reading the registers

How? memory-reference? arithmetic? control flow?

Fetching Instructions

- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - updating the PC value to be the address of the next (sequential) instruction

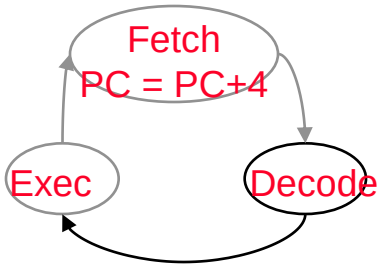




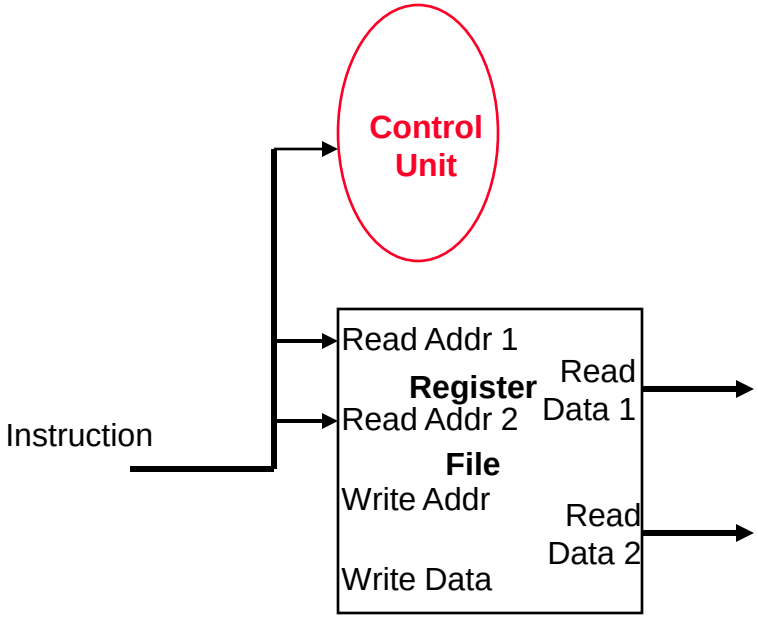
  - PC is updated every clock cycle, so it does not need an explicit write control signal just a clock signal
  - Reading from the Instruction Memory is a combinational activity, so it doesn't need an explicit read control signal

Decoding Instructions

- Decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit



and

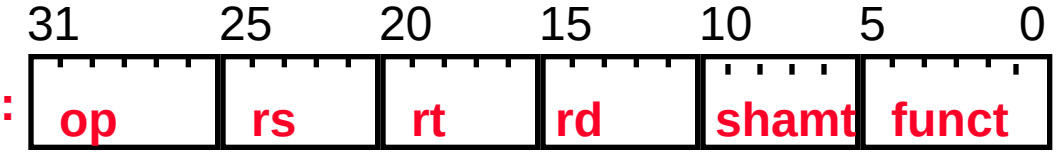


- Example: reading two values from the Register File

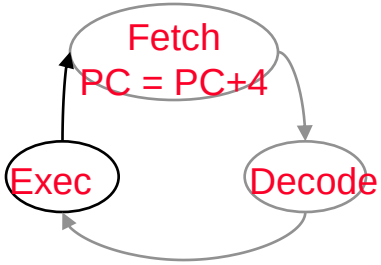
  →Register File addresses are contained in the instruction

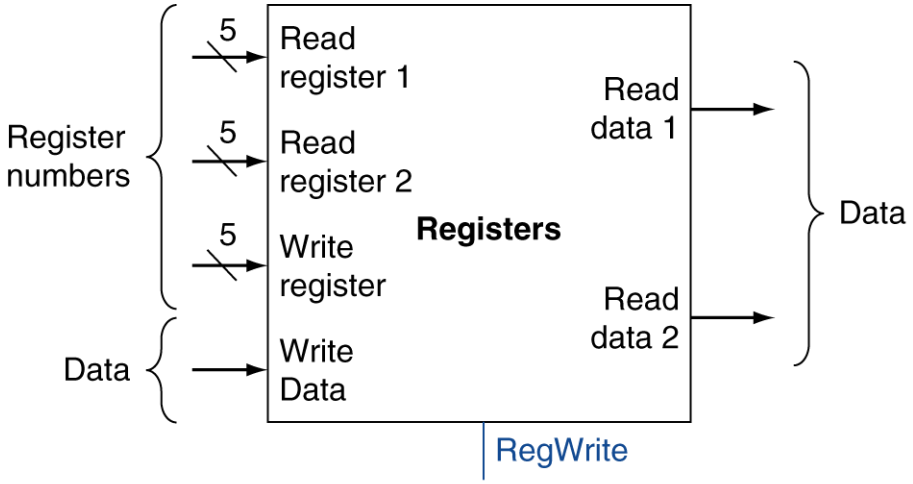
Executing R Format Operations

- R format operations (**`add, sub, slt, and, or`**)

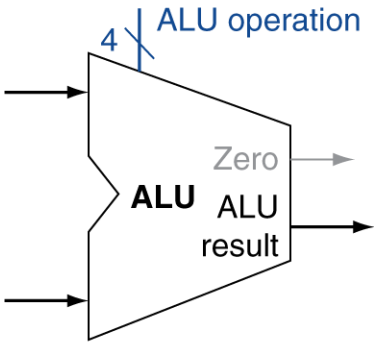


- read two register operands rs and rt
- perform operation (op and funct) on values in rs and rt
- store the result back into the Register File (into location rd)





a. Registers

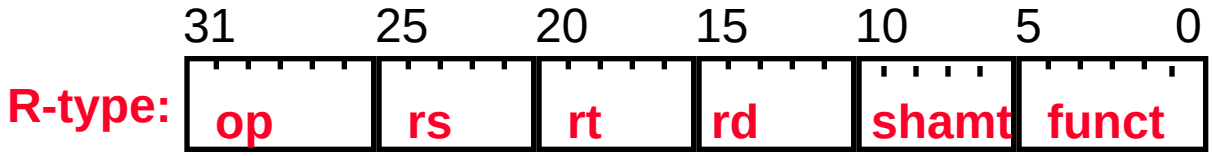


b. ALU

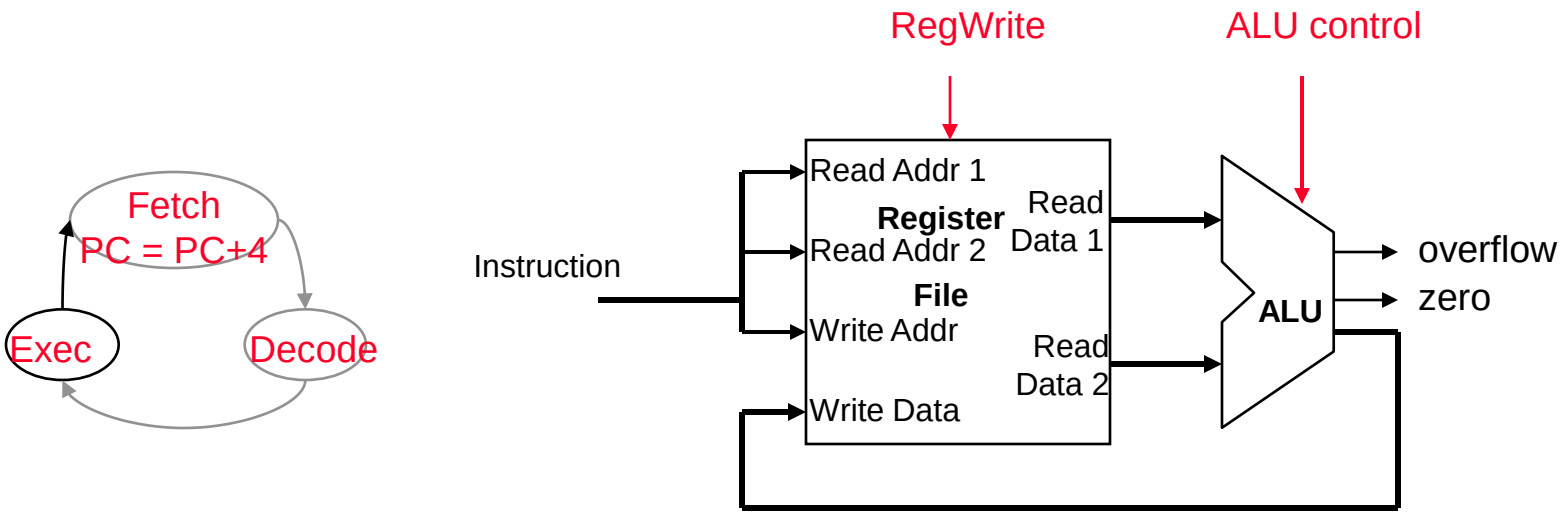
Draw connection between a and b to form the execution unit?

Executing R Format Operations

- R format operations (**`add, sub, slt, and, or`**)



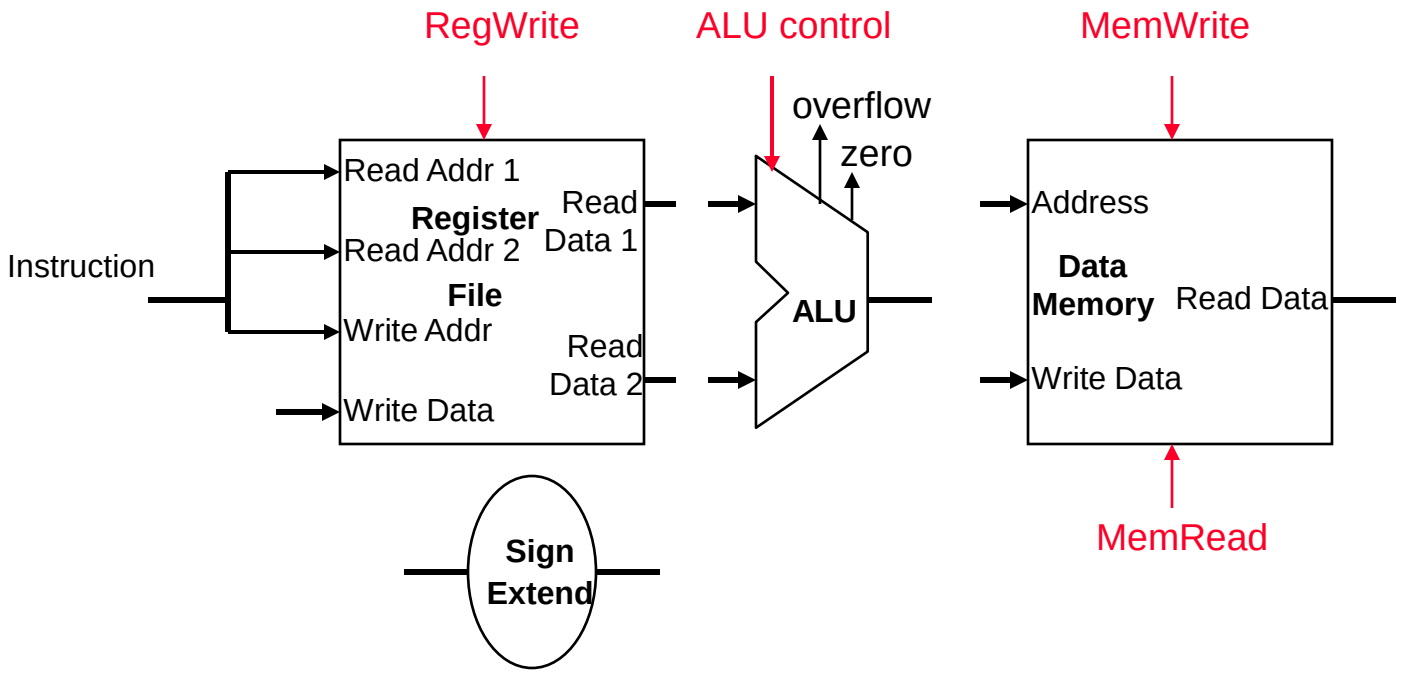
- read two register operands rs and rt
- perform operation (op and funct) on values in rs and rt
- store the result back into the Register File (into location rd)



- Note that Register File is not written every cycle (e.g. **`sw`**), so we need an explicit write control signal for the Register File

Executing Load and Store Operations

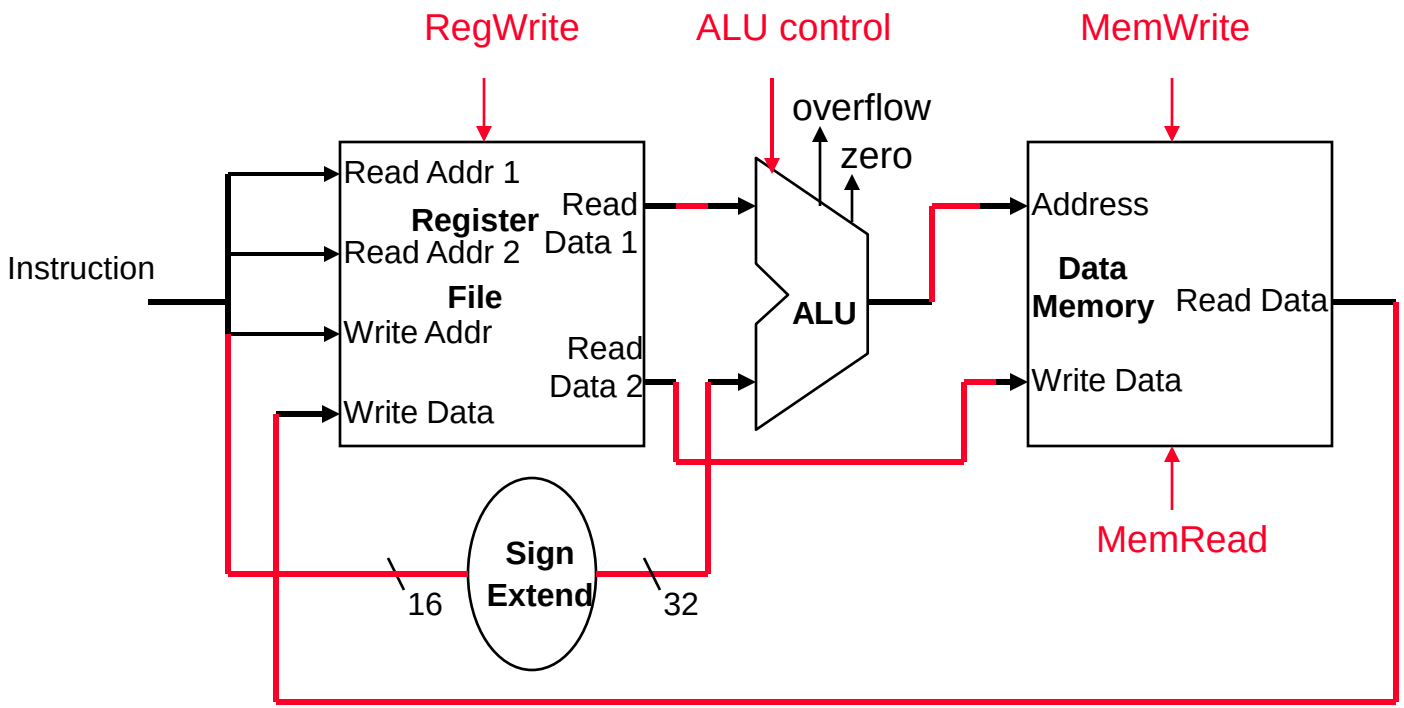
- Load and store operations involves
  - read register operands (including one base register)
  - compute memory address by adding the base to the 16-bit signed-extended offset field in the instruction
  - store: read from the Register File, write to the Data Memory
  - load: read from the Data Memory, write to the Register File



Draw necessary connections to form execution unit?

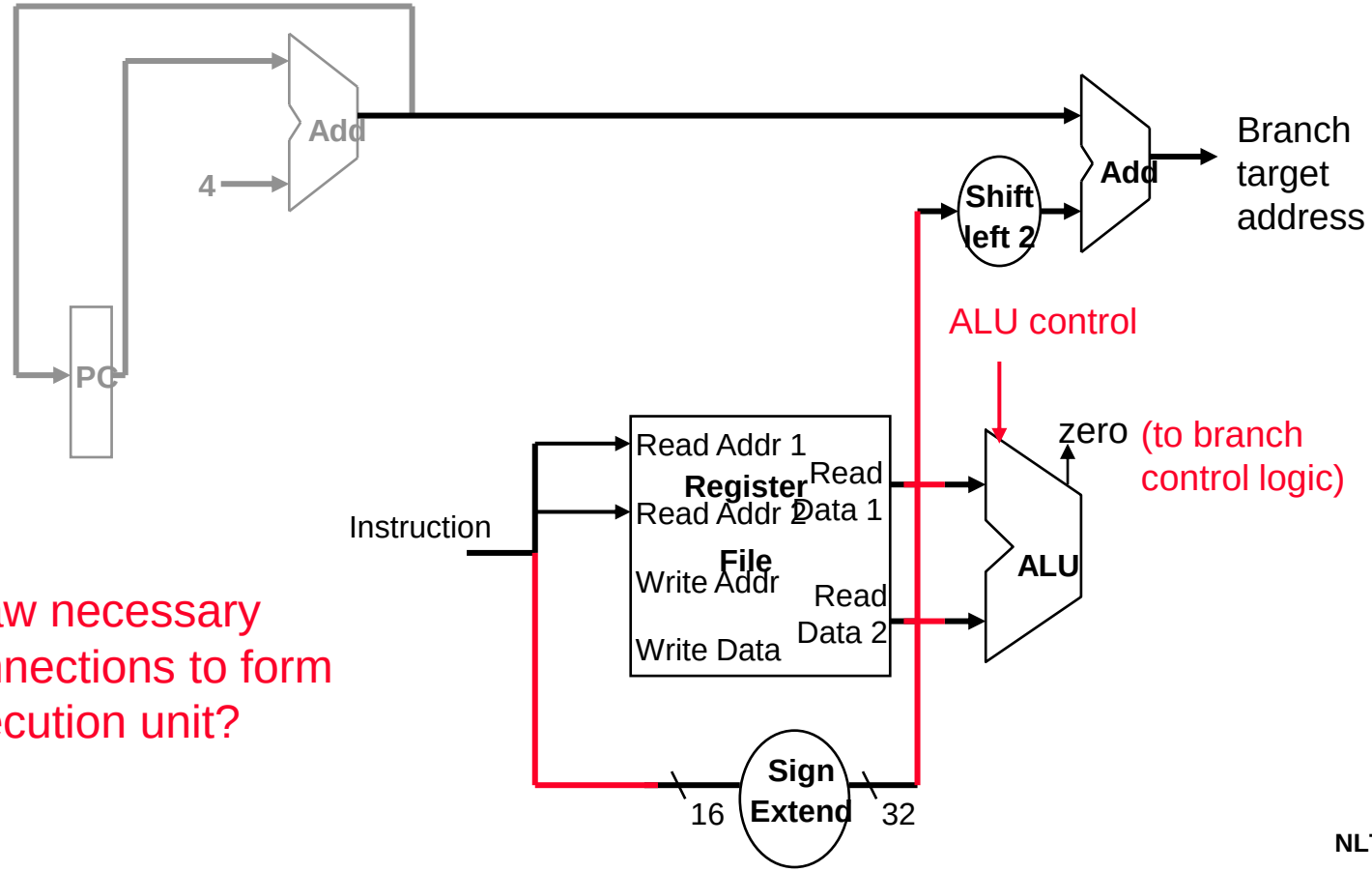
Executing Load and Store Operations

- Load and store operations involves
  - read register operands (including one base register)
  - compute memory address by adding the base to the 16-bit signed-extended offset field in the instruction
  - store: read from the Register File, write to the Data Memory
  - load: read from the Data Memory, write to the Register File



Executing Branch Operations

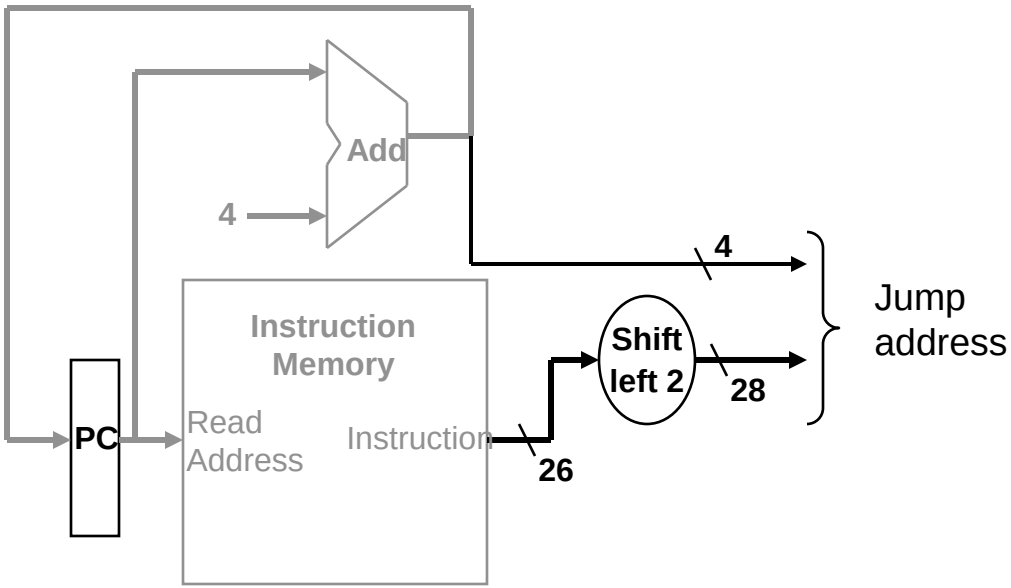
- Branch operations involves
  - read register operands
  - compare the operands (subtract, check `zero` ALU output)
  - compute the branch target address: adding the updated PC to the 16-bit signed-extended offset field in the instr



Draw necessary connections to form execution unit?

Executing Jump Operations

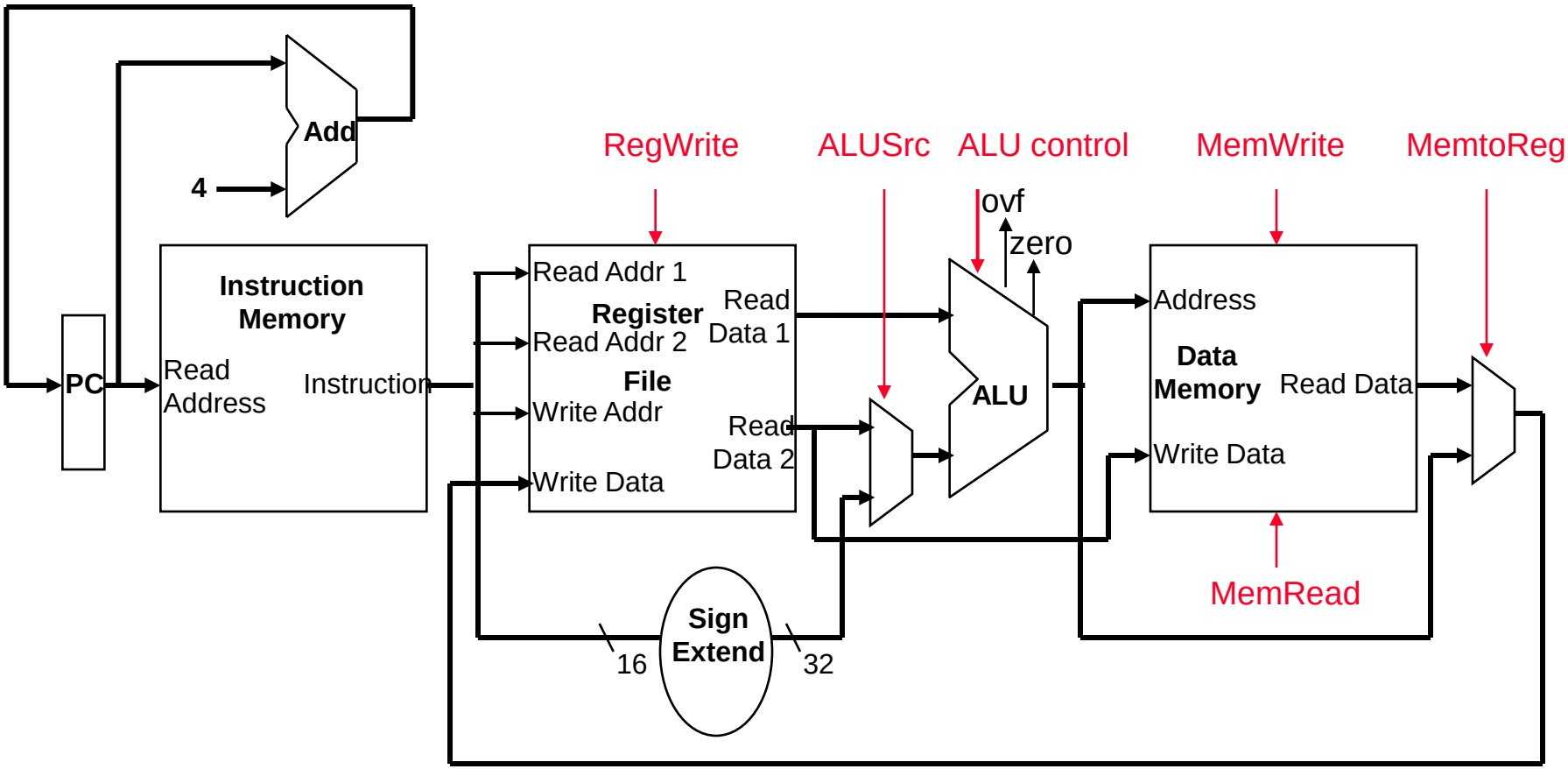
- Jump operation involves
  - keep 4 highest bits of PC
  - replace the lower 28 bits of the PC by
    - the lower 26 bits of the fetched instruction shifted left by 2 bits



Creating a Single Datapath from the Parts

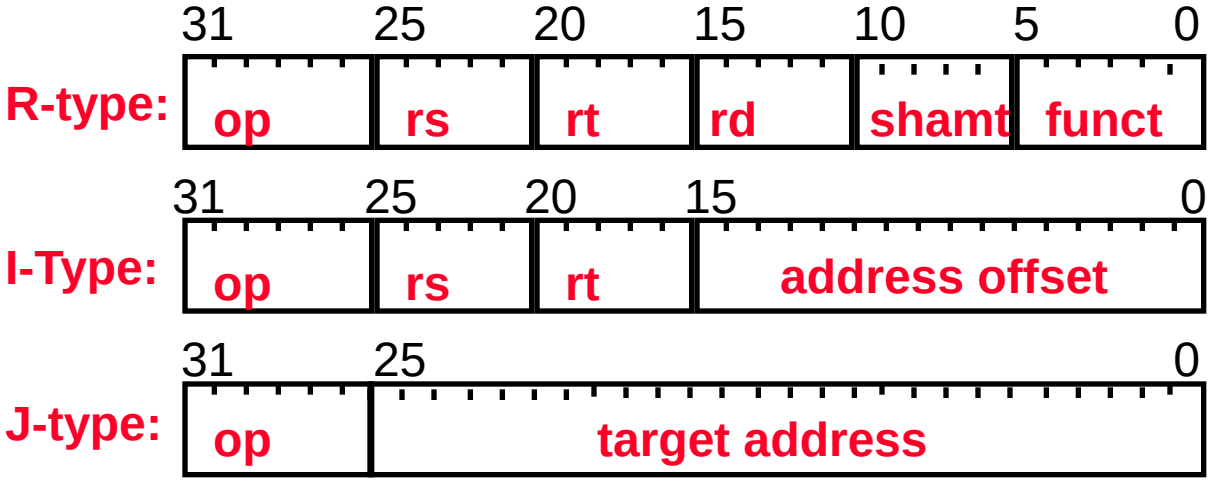
- Assemble the datapath segments and add control lines and multiplexors as needed
- Single cycle design – fetch, decode and execute each instructions in one clock cycle
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
  - multiplexors needed at the input of shared elements with control lines to do the selection
  - write signals to control writing to the Register File and Data Memory
- Cycle time is determined by length of the longest path

Fetch, R, and Memory Access Portions



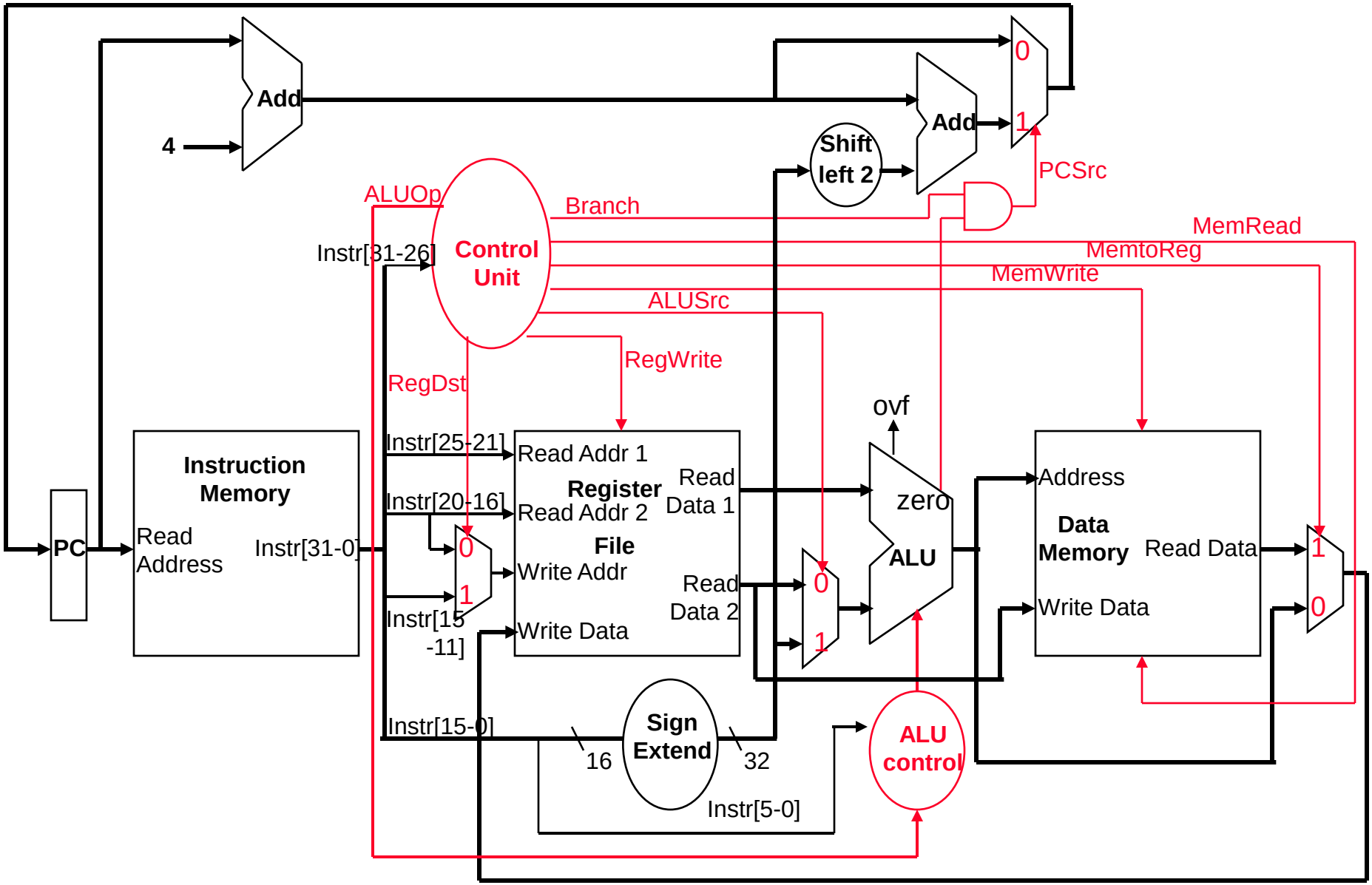
Adding the Control

- Selecting the operations to perform (ALU, Register File and Memory read/write)
- Controlling the flow of data (multiplexor inputs)

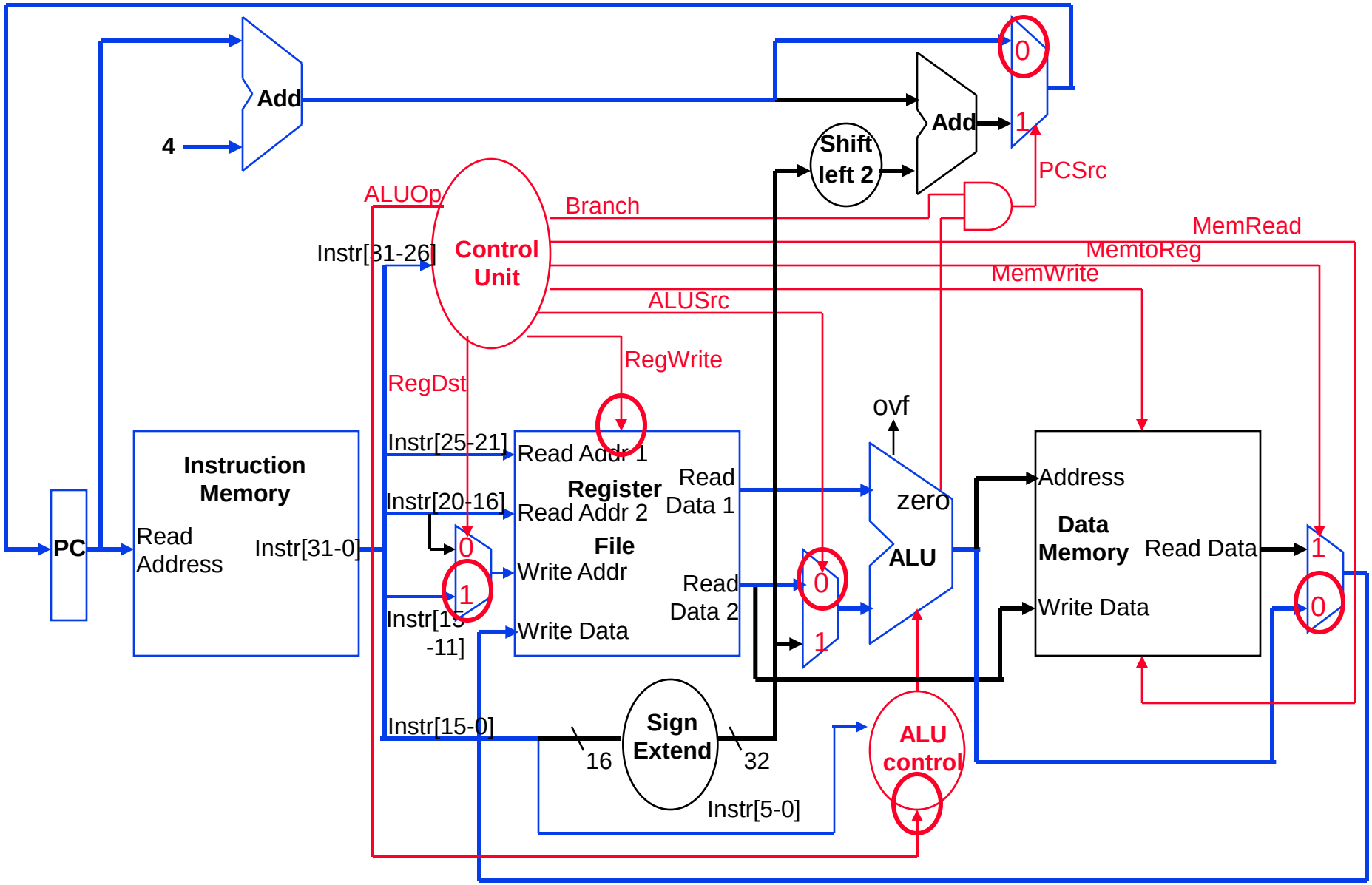


- Observations
  - op field always in bits 31-26
  - addr of registers to be read are always specified by the rs field (bits 25-21) and rt field (bits 20-16); for lw and sw rs is the base register
  - addr. of register to be written is in one of two places – in rt (bits 20-16) for lw; in rd (bits 15-11) for R-type instructions
  - offset for beq, lw, and sw always in bits 15-0

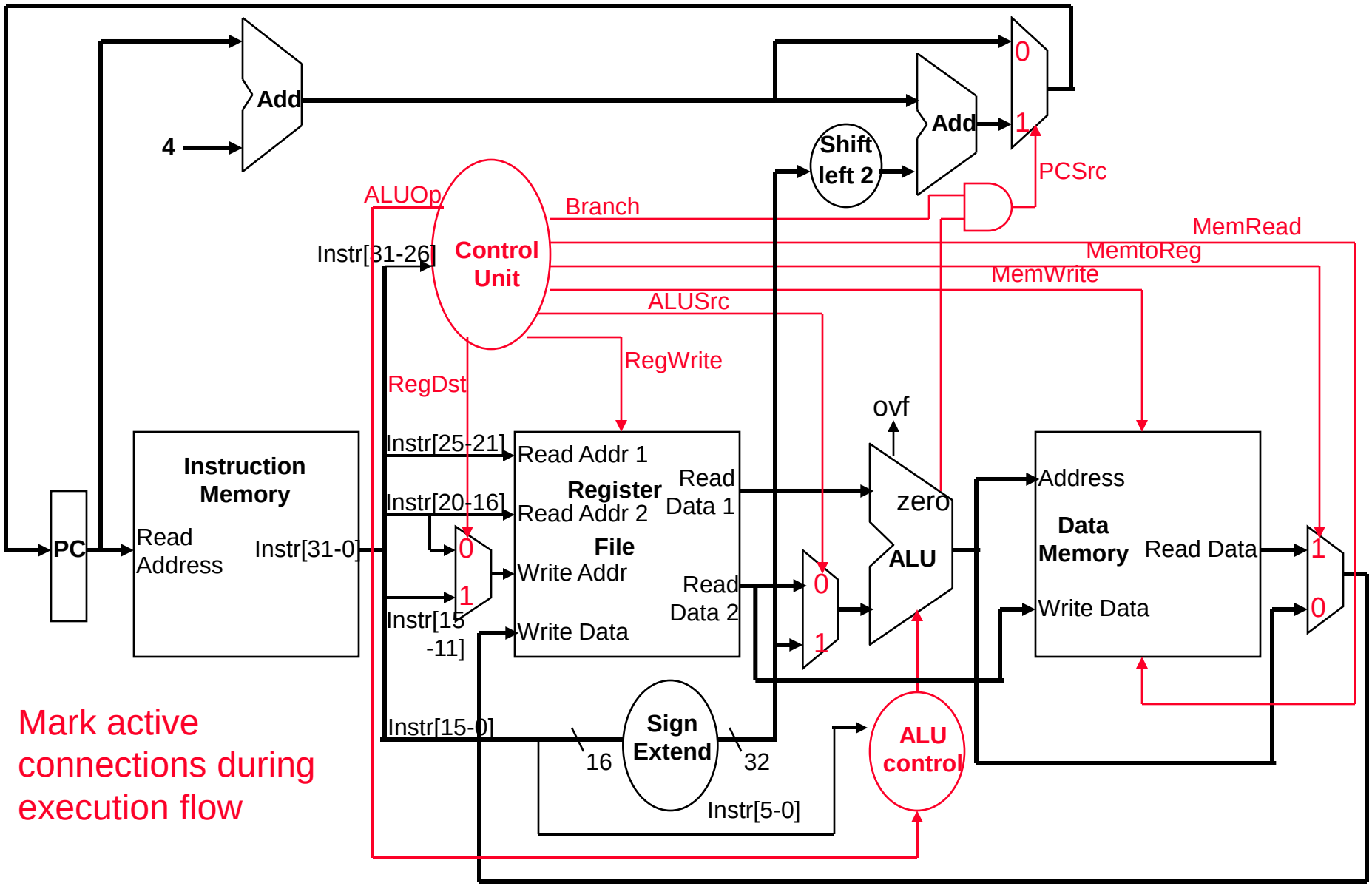
Single Cycle Datapath with Control Unit



R-type Instruction Data/Control Flow

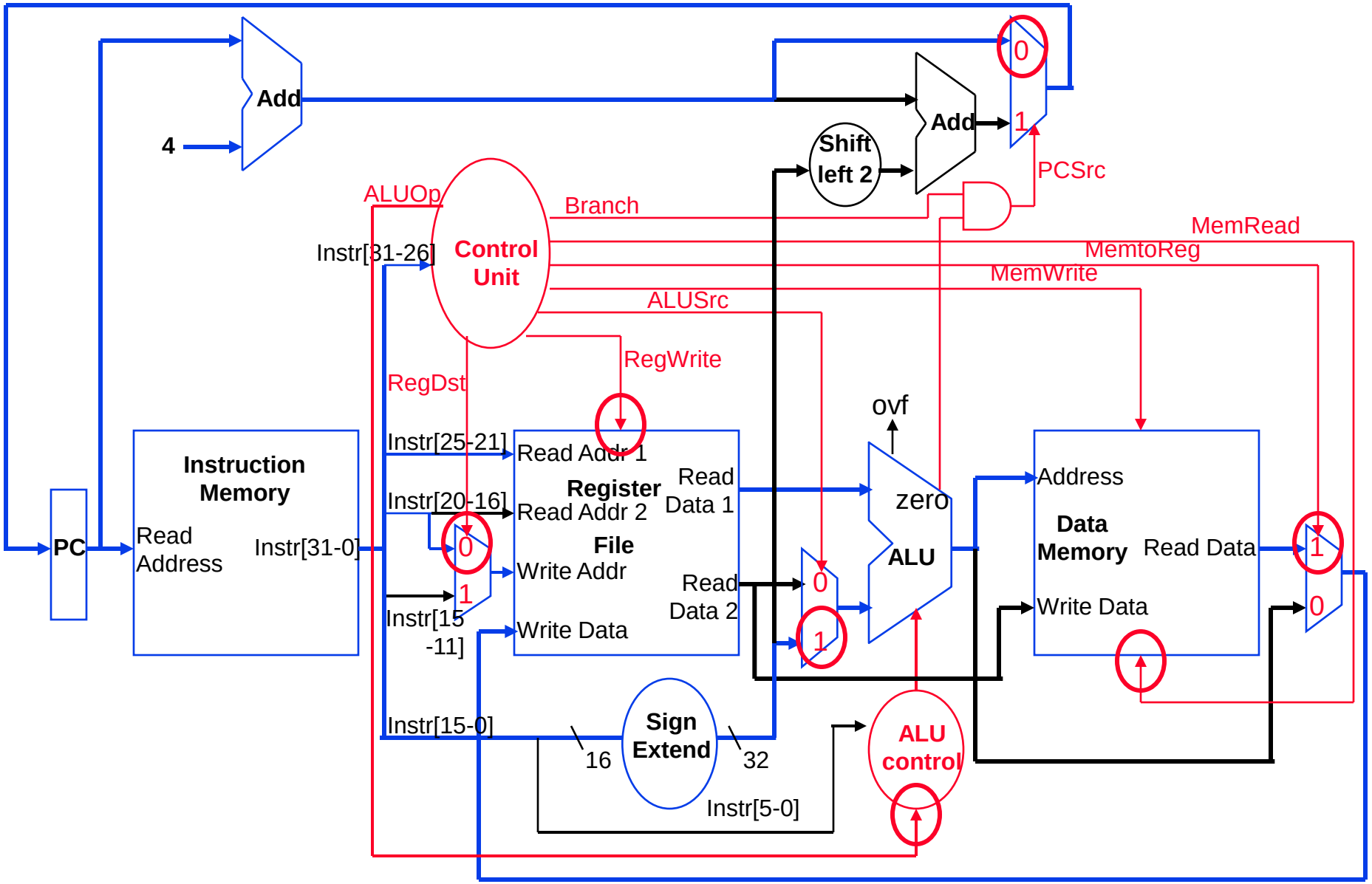


Load Word Instruction Data/Control Flow

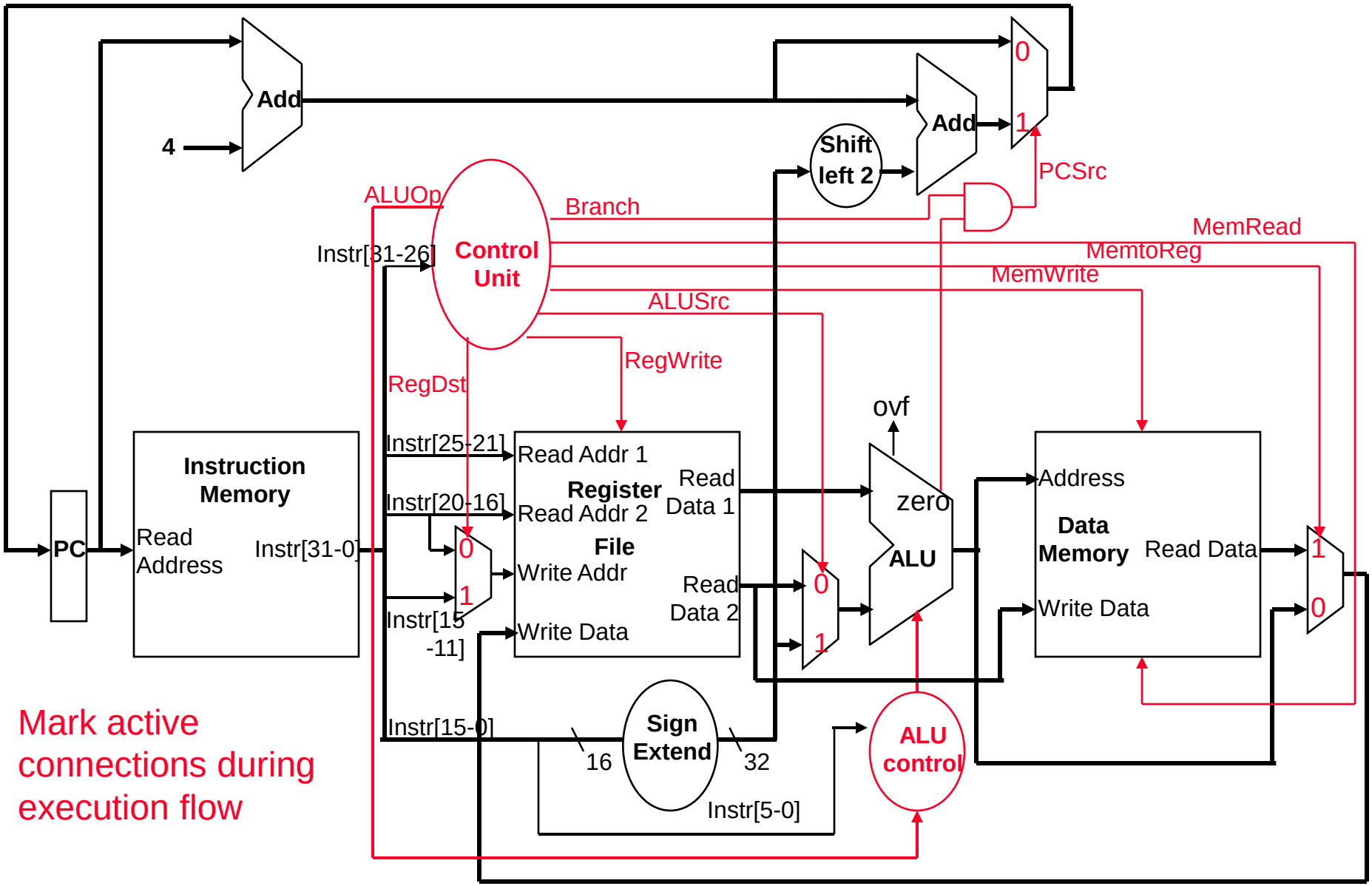


Mark active connections during execution flow

Load Word Instruction Data/Control Flow

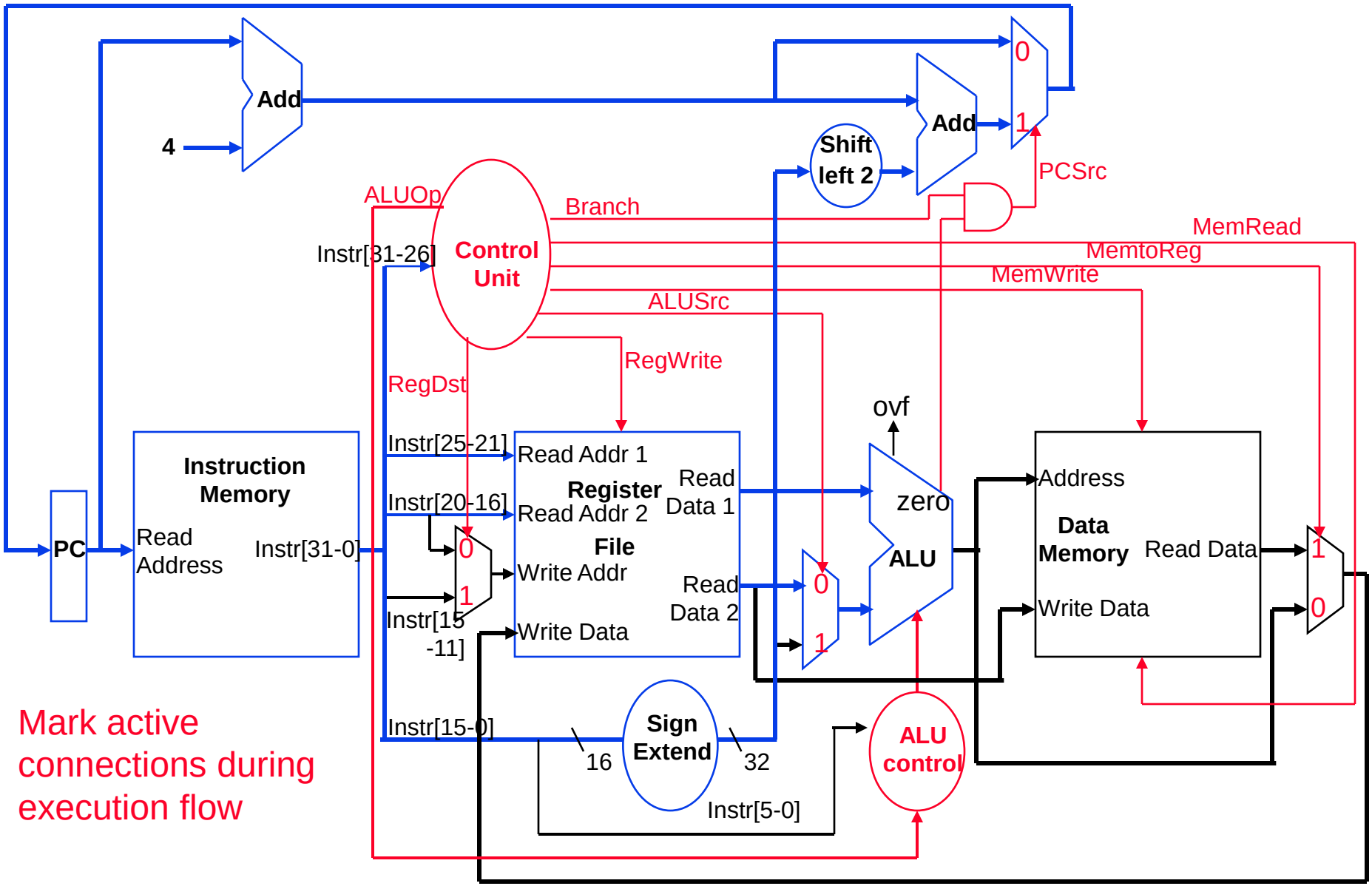


Branch Instruction Data/Control Flow



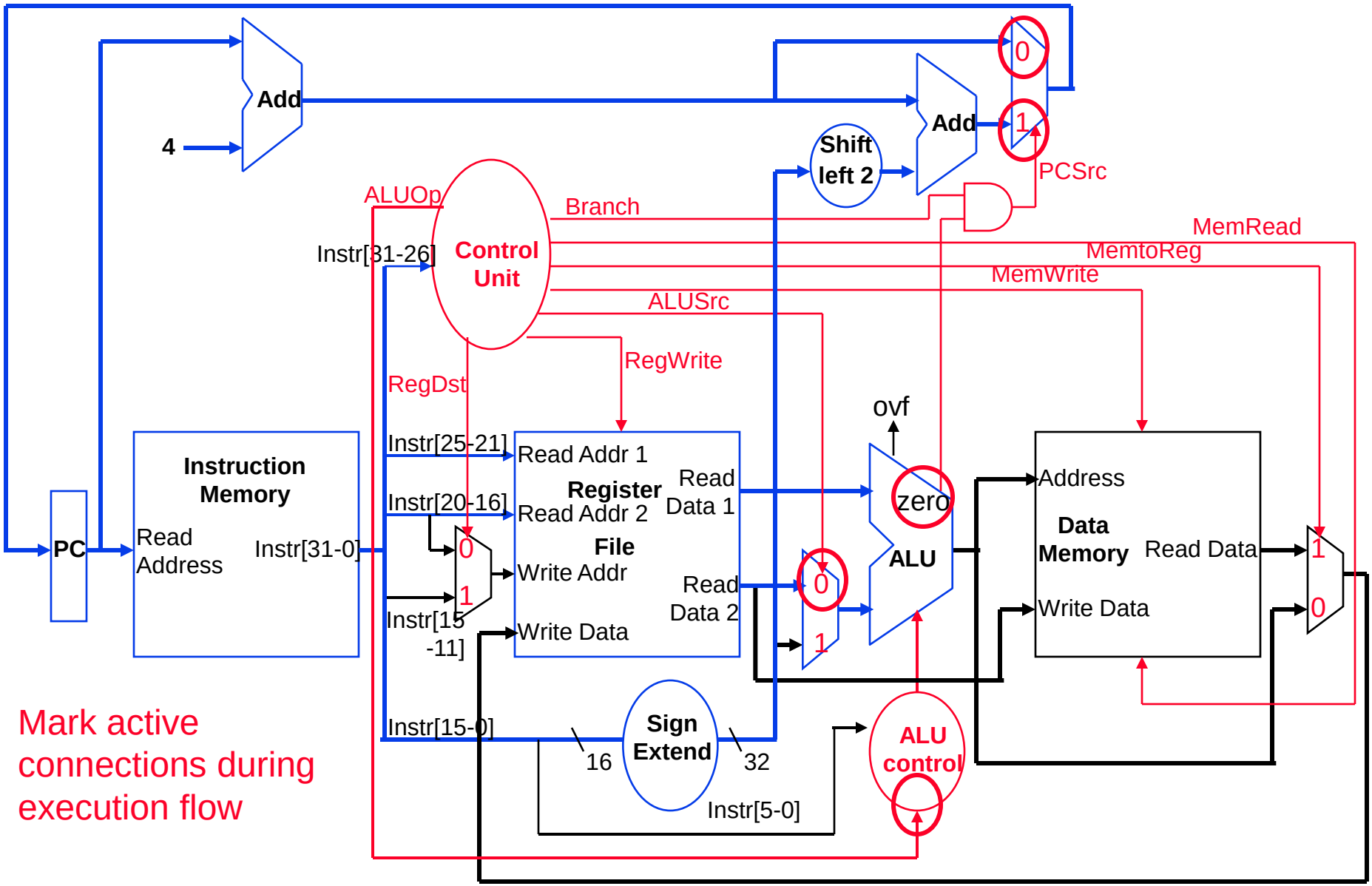
Mark active connections during execution flow

Branch Instruction Data/Control Flow



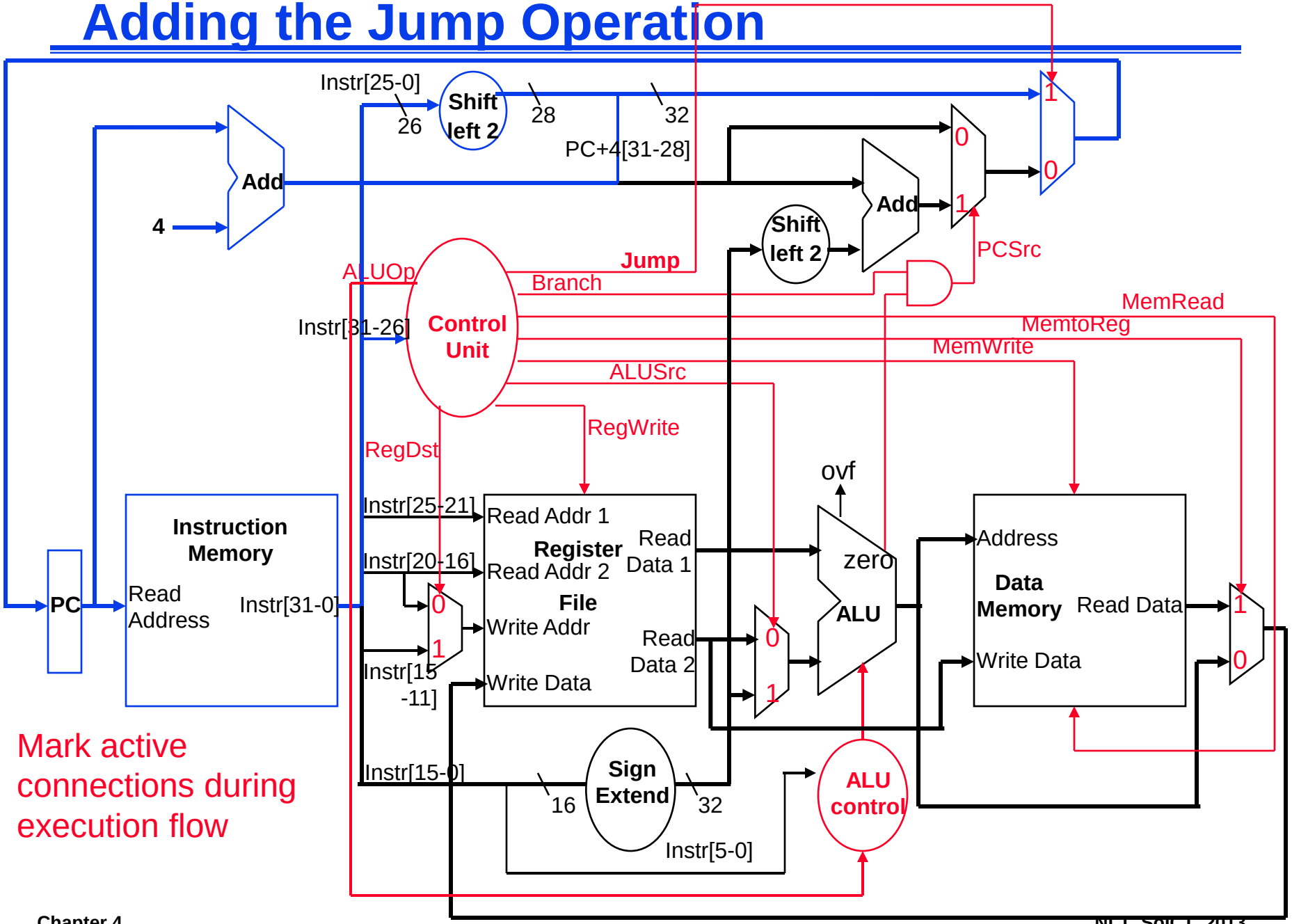
Mark active connections during execution flow

Branch Instruction Data/Control Flow



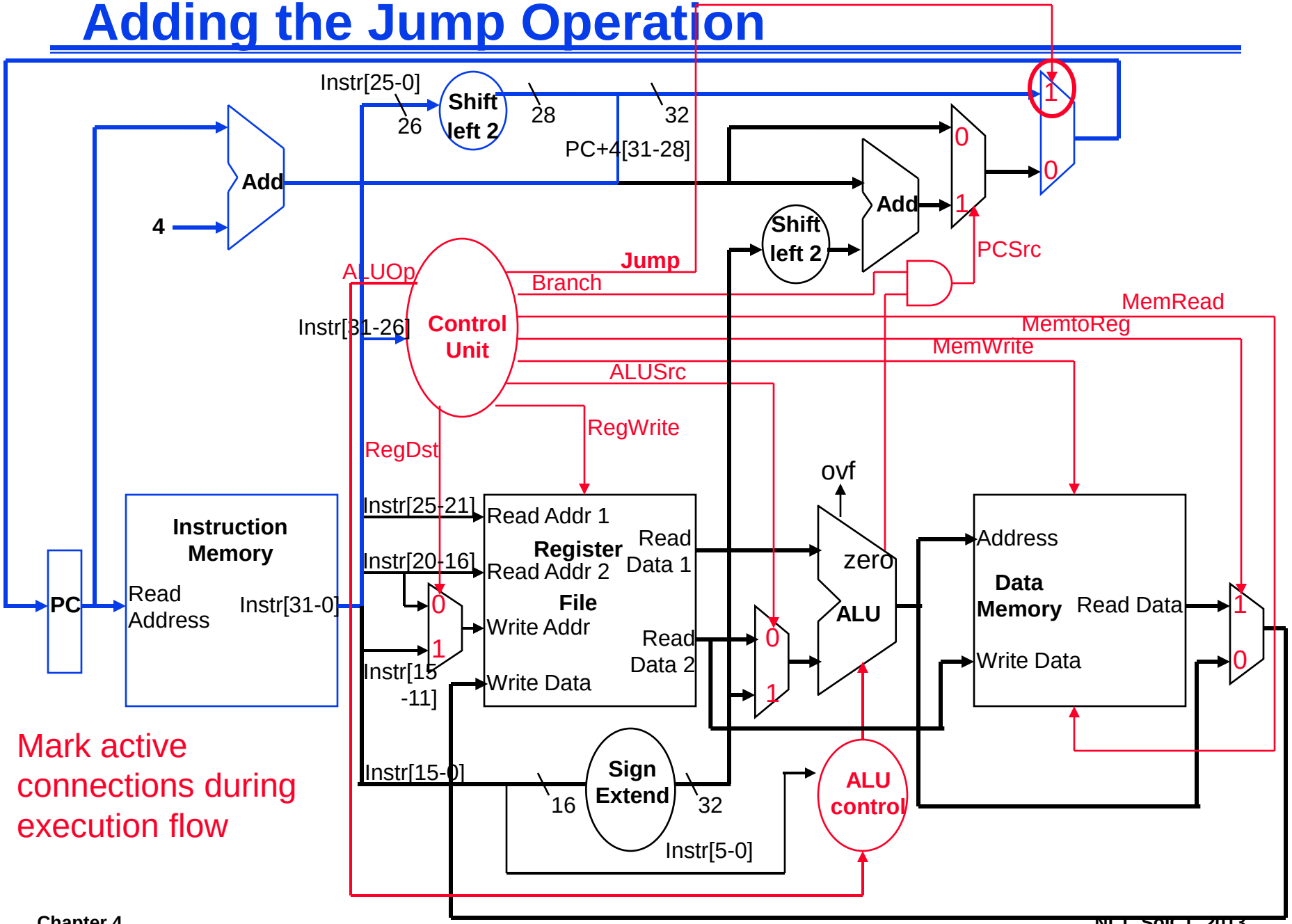
Mark active connections during execution flow

Adding the Jump Operation



Mark active connections during execution flow

Adding the Jump Operation



Mark active connections during execution flow

Instruction Times (Critical Paths)

- What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times except:
  - Instruction and Data Memory (200 ps)
  - ALU and adders (200 ps)
  - Register File access (reads or writes) (100 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type						
load						
store						
beq						
jump						

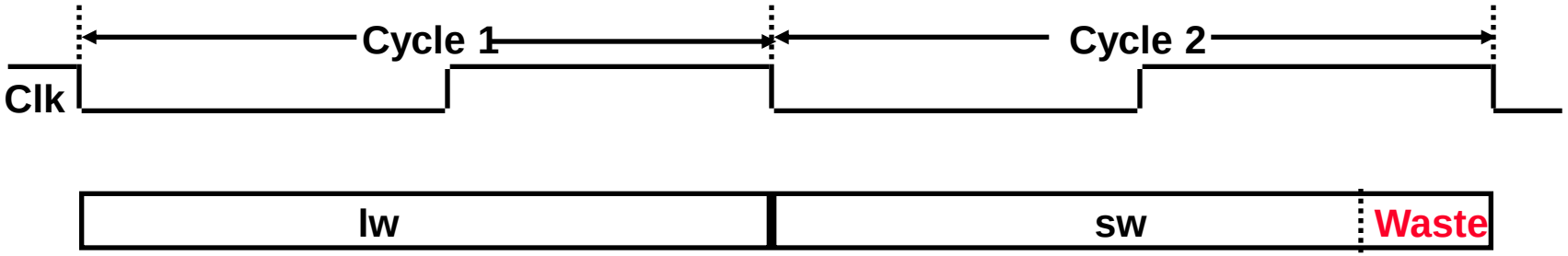
Instruction Critical Paths for Single cycle CPU

- What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times except:
  - Instruction and Data Memory (200 ps)
  - ALU and adders (200 ps)
  - Register File access (reads or writes) (100 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	200	100	200		100	600
load	200	100	200	200	100	800
store	200	100	200	200		700
beq	200	100	200			500
jump	200					200

Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  - especially problematic for more complex instructions like floating point multiply



- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

but

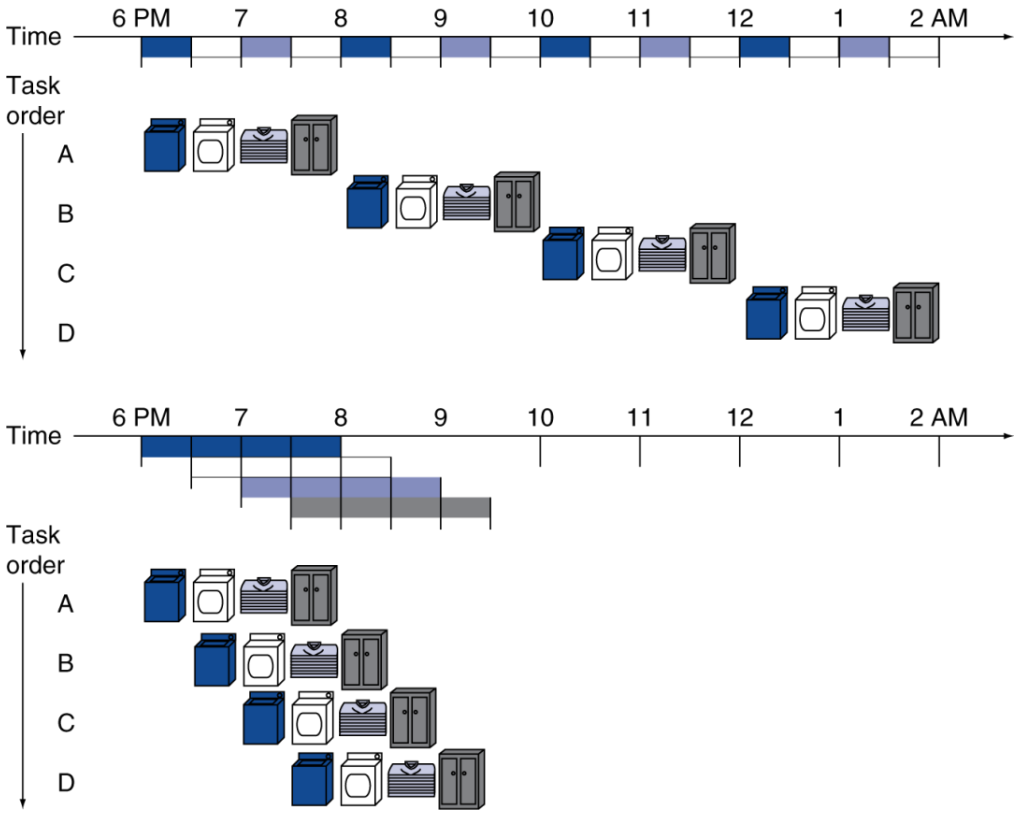
- Is simple and easy to understand

How Can We Make It Faster?

- Divide instruction cycles into smaller cycles
- Executing instructions in parallel
  - With only one CPU?
- Pipelining:
  - Start fetching and executing the next instruction before the current one has completed
  - Overlapping execution

Example: laundry work

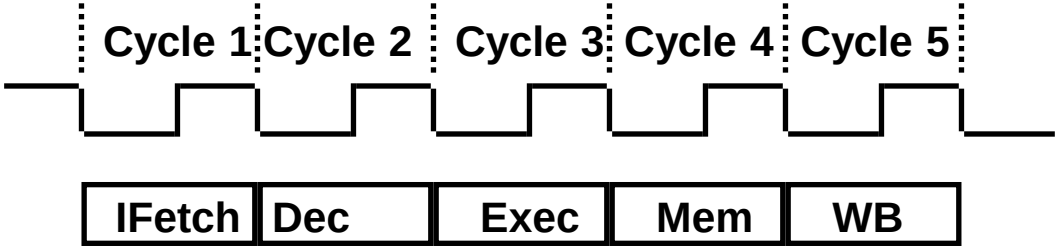
- Pipelined laundry boots performance up to 4 times



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

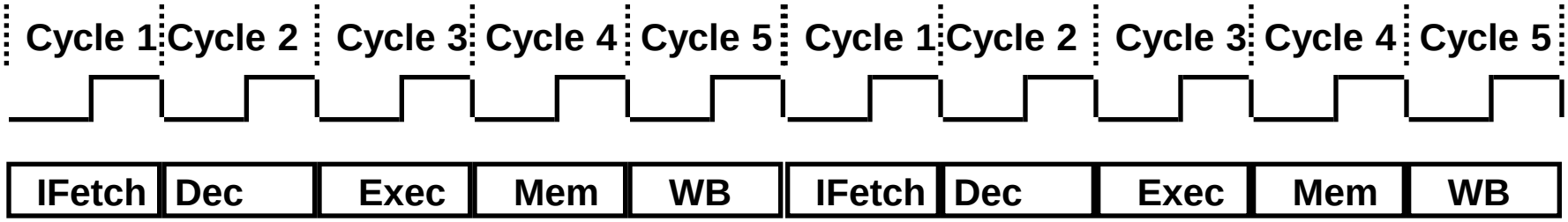
MIPS Pipeline

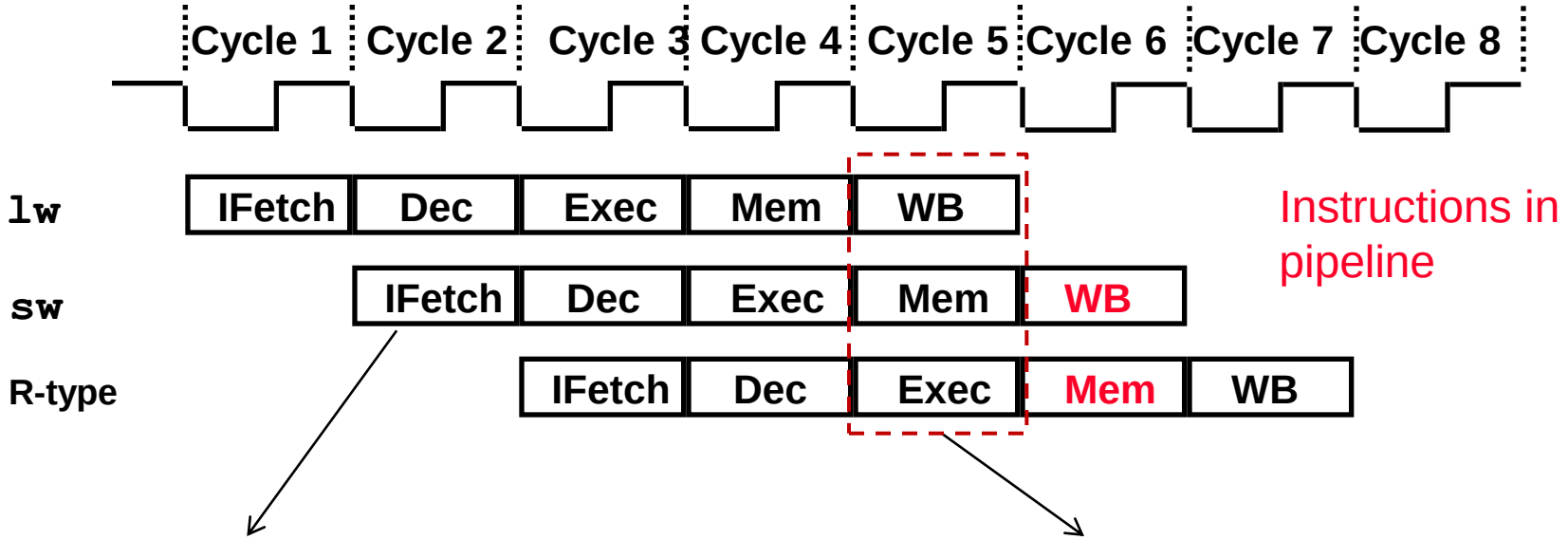
- Five stages, one step per stage
  - IFetch: Instruction Fetch and Update PC
  - Dec: Registers Fetch and Instruction Decode
  - Exec: Execute R-type; calculate memory address
  - Mem: Read/write the data from/to the Data Memory
  - WB: Write the result data into the register file



Execution time for a single instruction is always 5 cycles, regardless of instruction operation

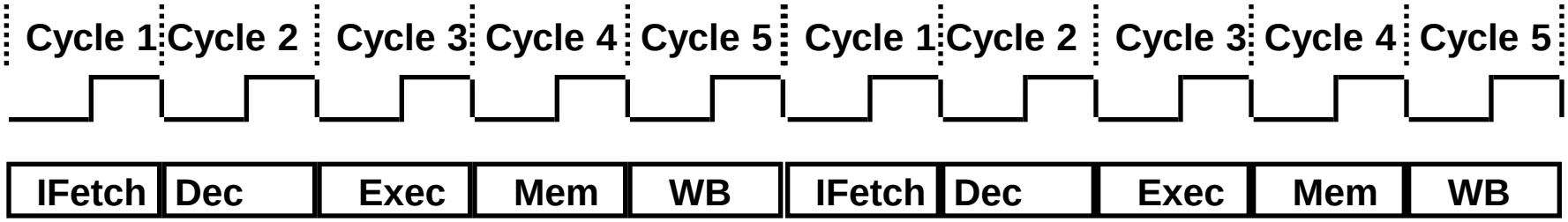
Instruction pipeline





Start fetching and executing the next instruction before the current one has completed

More than one instruction are executed at a time


Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
IFetch
Dec
Exec
Mem
WB
IFetch
Dec
Exec
Mem
WB

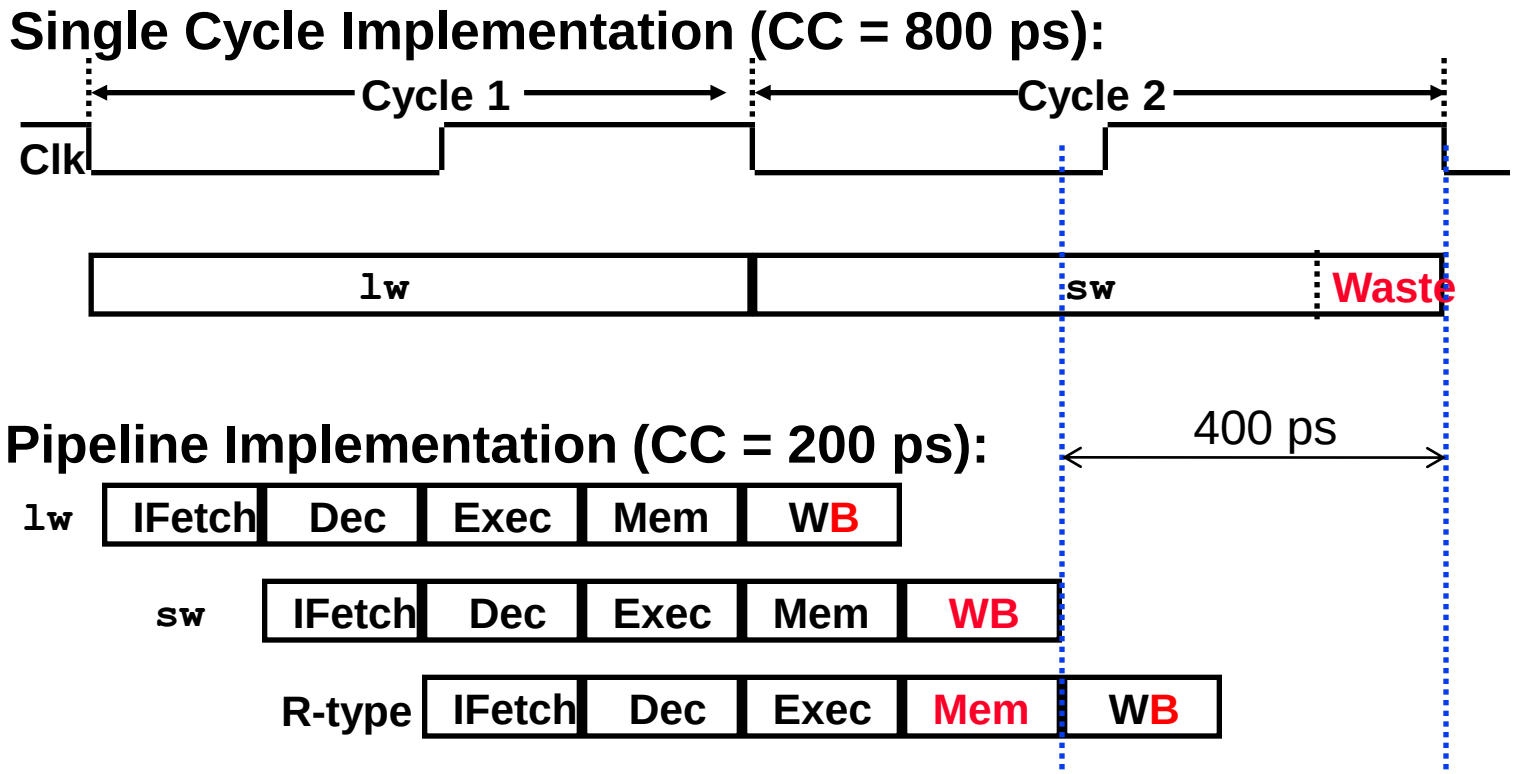
Pipeline performance

- All modern processors are pipelined for performance
  - Remember *the* performance equation:
    CPU time = CPI * CC * IC
- Under *ideal* conditions (balance) and with a large number of instructions:
  - A five stage pipeline is nearly five times faster because the CC is nearly five times faster

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

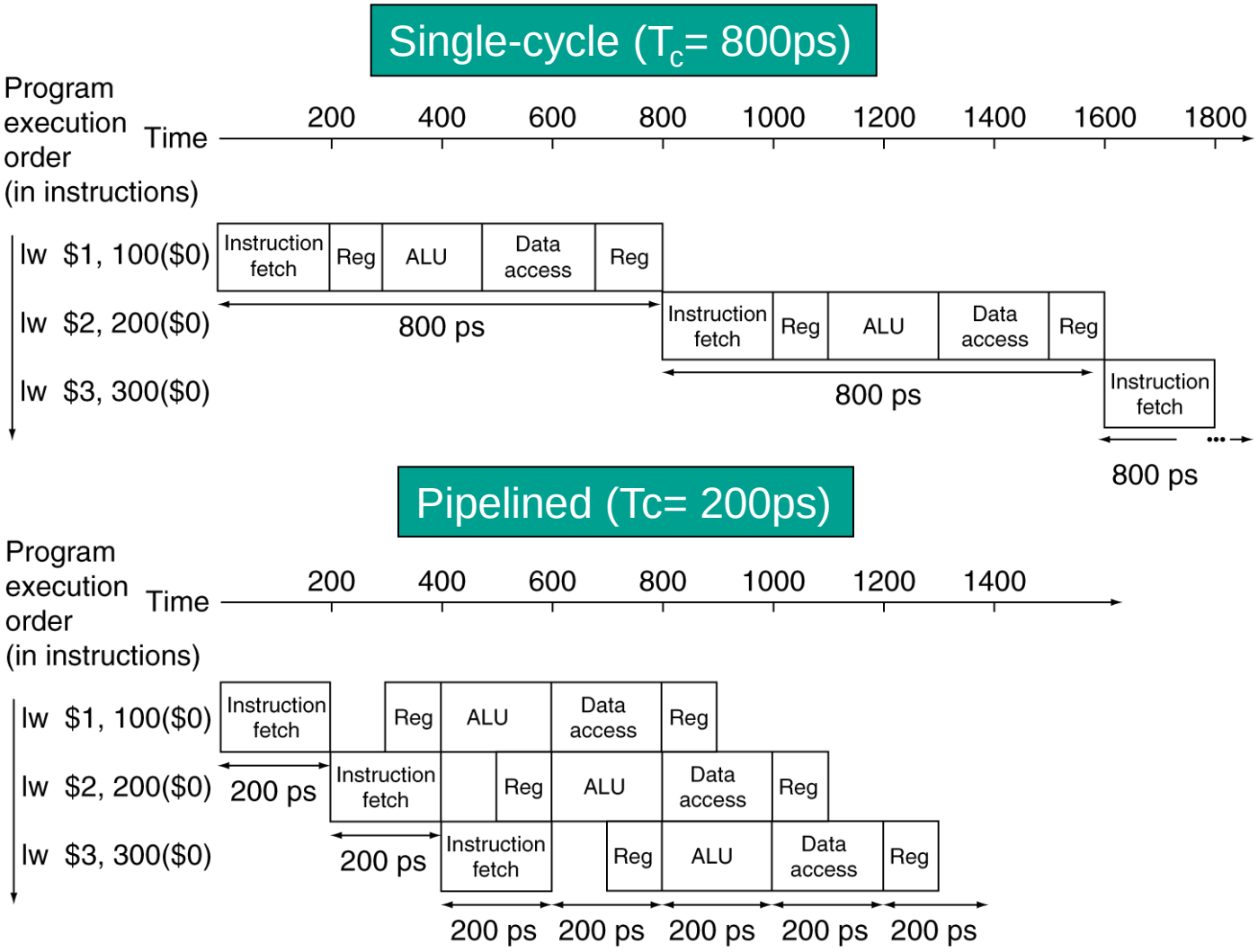
  - improves throughput - total amount of work done in a given time
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced
- In reality, speedup is less because of imbalance and overhead

Single Cycle versus Pipeline



- To complete an entire instruction in the pipelined case takes 1000 ps (as compared to 800 ps for the single cycle case). Why ?
- How long does each take to complete 1,000,000 adds ?

Example with lw instructions



Pipelining the MIPS ISA

- What makes it easy: advantages of MIPS ISA
  - all instructions are the same length (32 bits)
    - can fetch in the 1st stage and decode in the 2nd stage
  - few instruction formats (three) with symmetry across formats
    - can begin reading register file in 2nd stage
  - memory operations occur only in loads and stores
    - can use the execute stage to calculate memory addresses
  - each instruction writes at most one result (i.e., changes the machine state) and does it in the last few pipeline stages (MEM or WB)
  - operands must be aligned in memory so a single data transfer takes only one data memory access

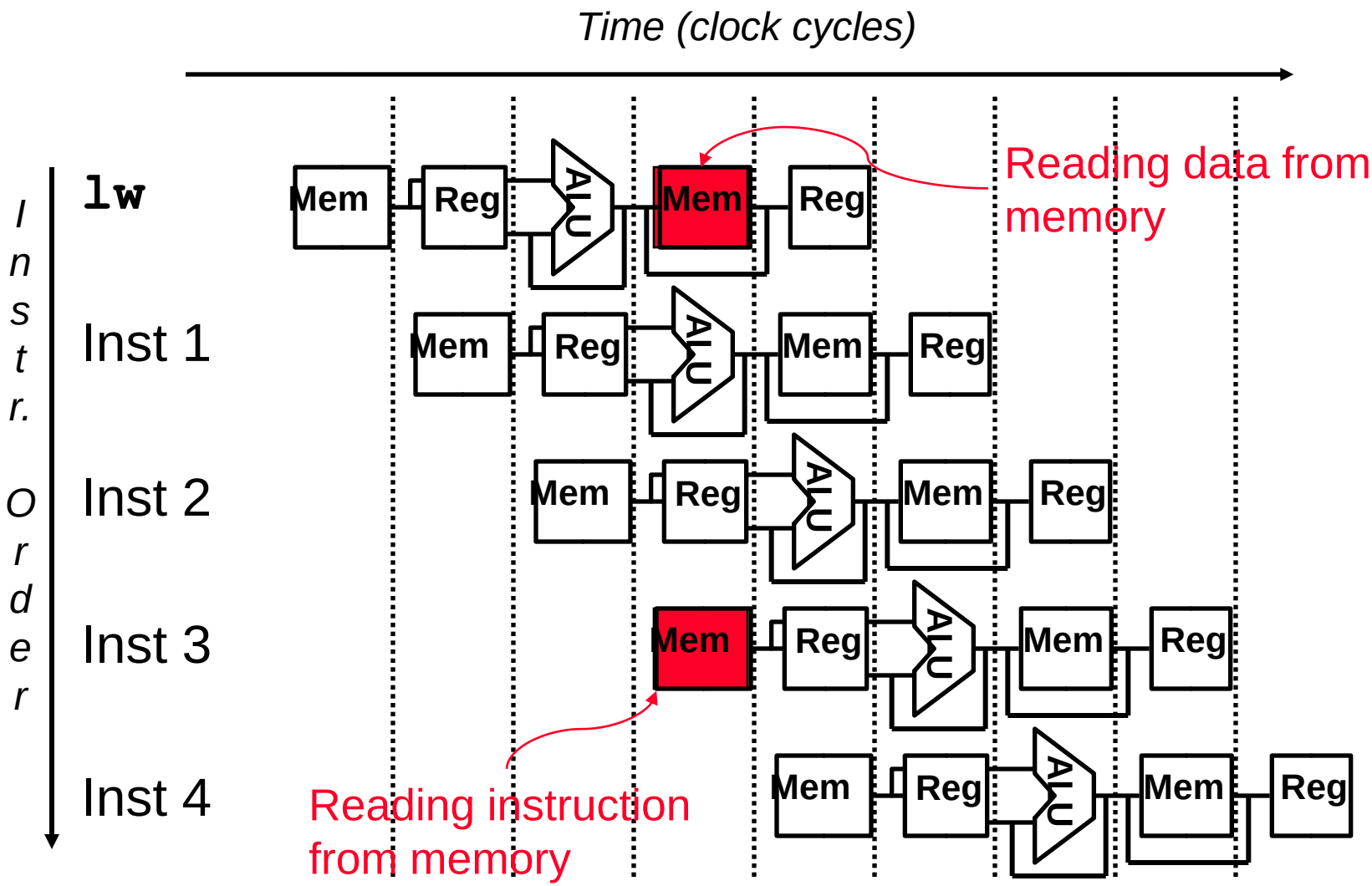
Pipeline hazards

- Pipeline can lead us into troubles!!!
- Hazards: situations that prevent starting the next instruction in the next cycle
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch and jump instructions, exceptions
- In most cases, hazard can be solved simply by waiting
  - but we need better solutions to take advantages of pipeline

Structural hazard

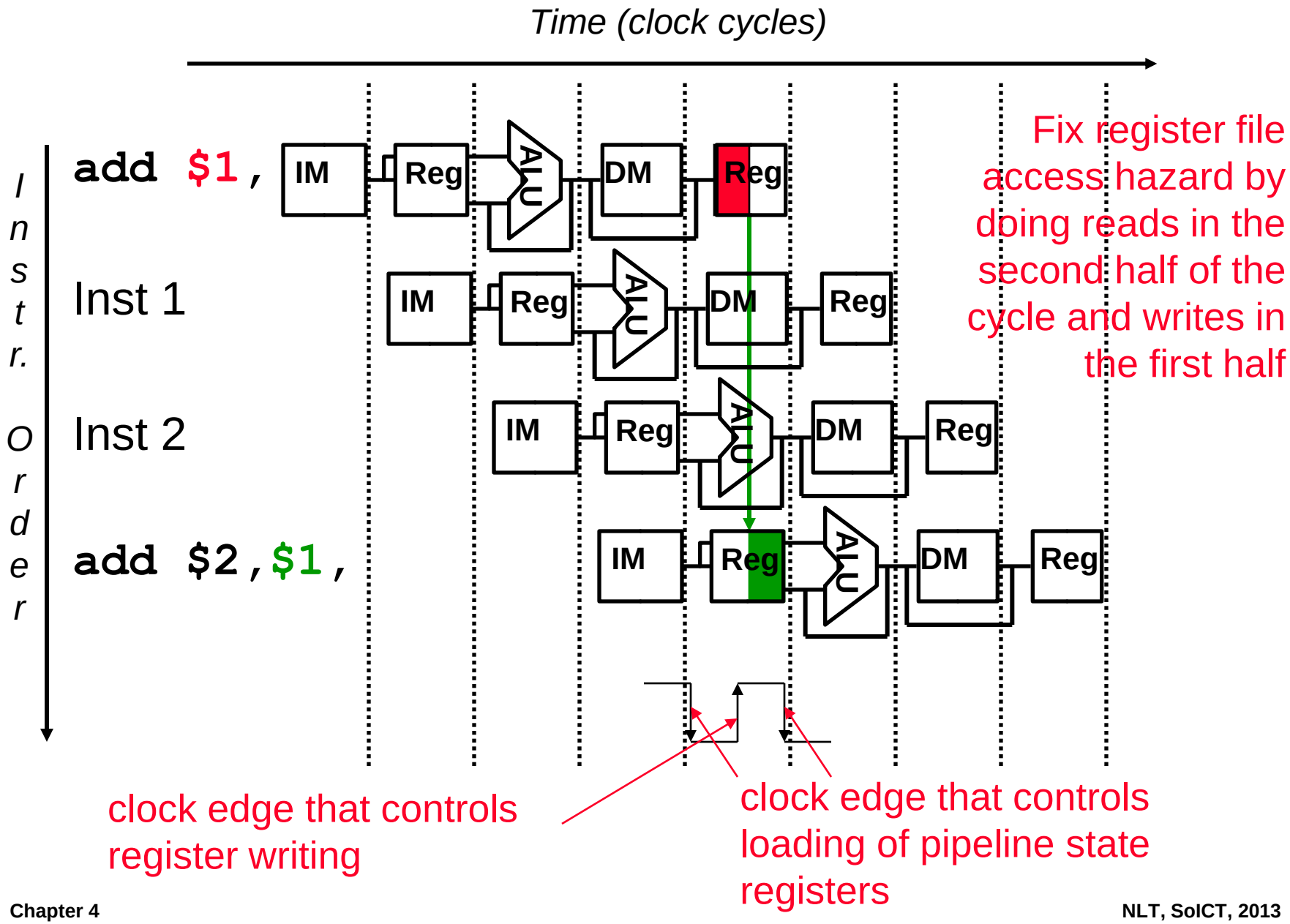
- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

A Single Memory Would Be a Structural Hazard



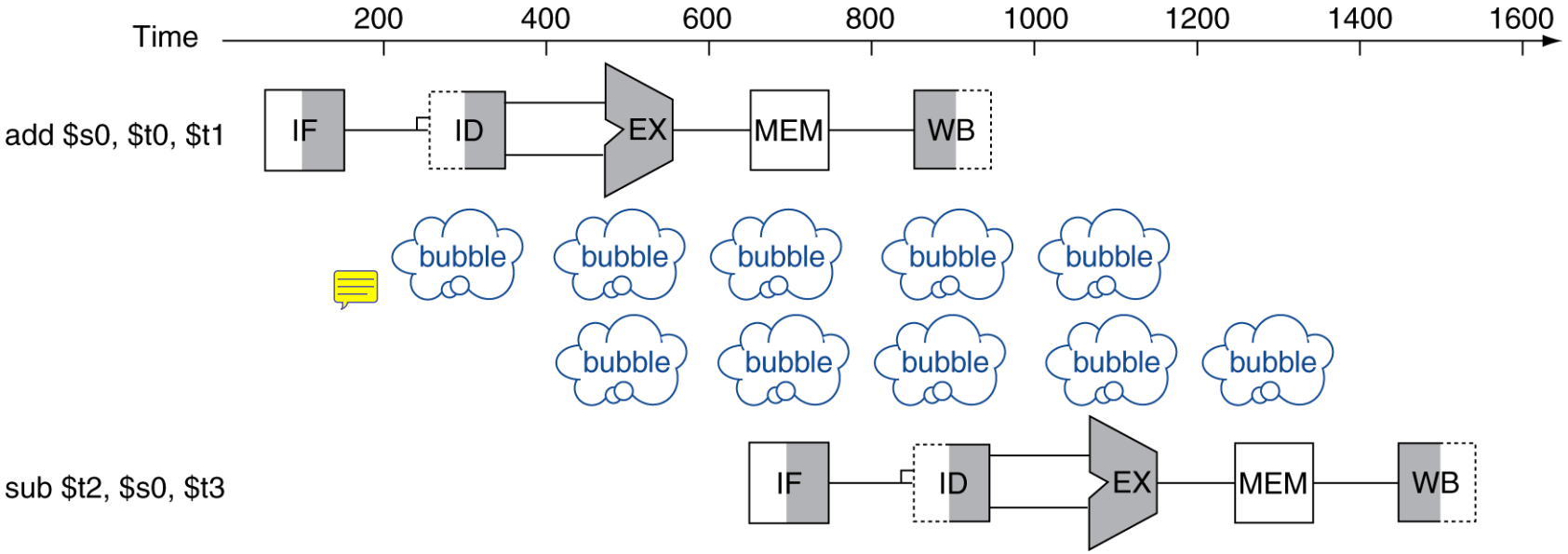
- Fix with separate instr and data memories (I$ and D$)

How About Register File Access?



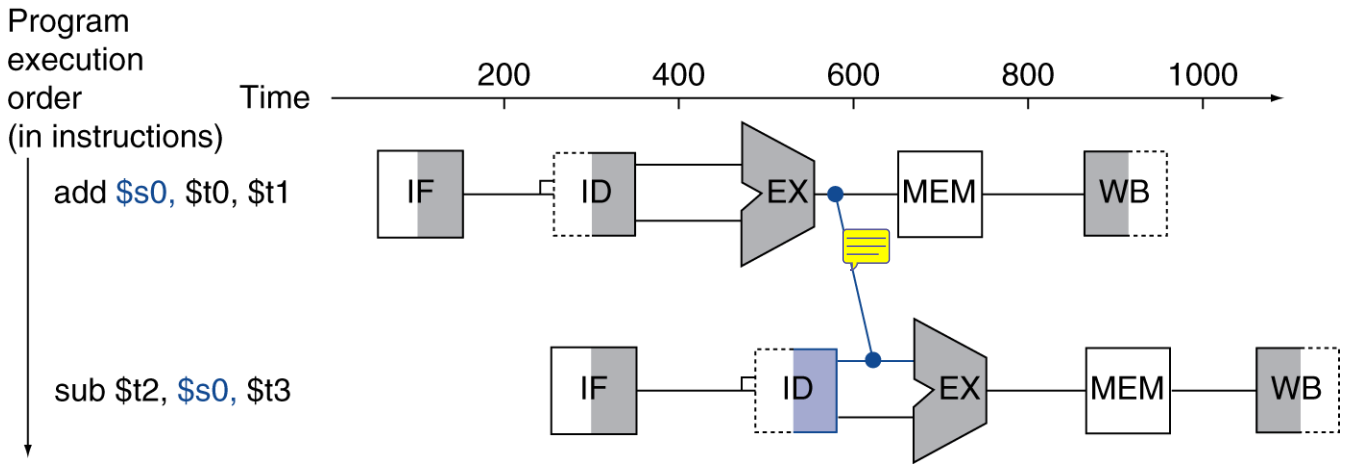
Data hazard

- An instruction depends on completion of data access by a previous instruction
  - `add  $s0, $t0, $t1`
    `sub  $t2, $s0, $t3`



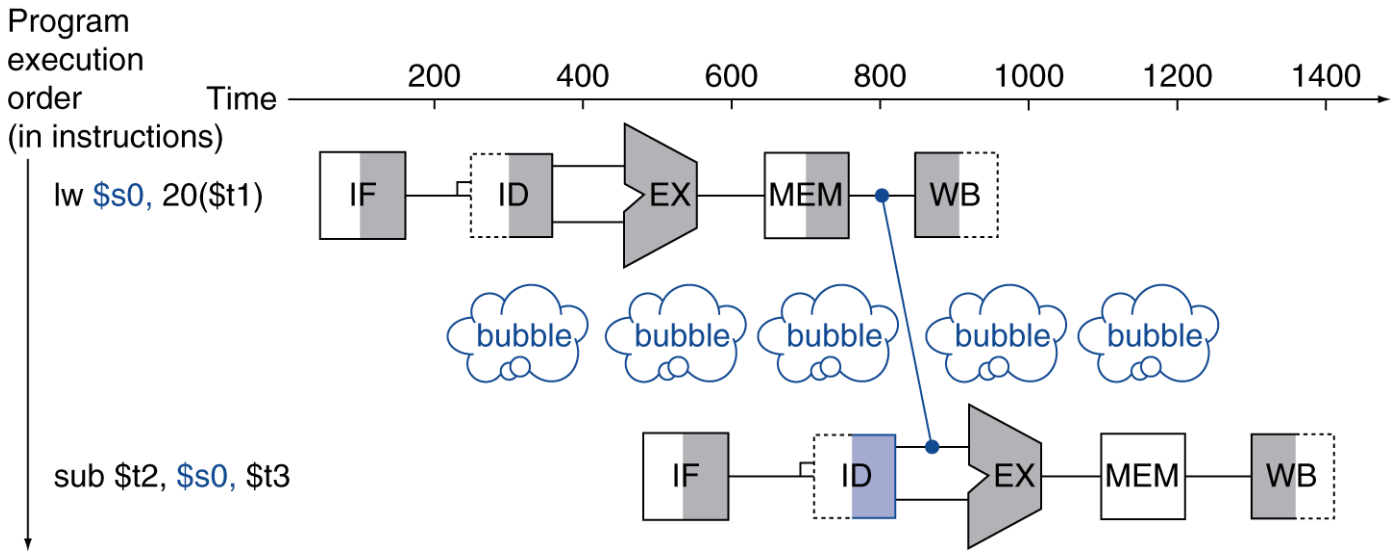
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra connections in the datapath



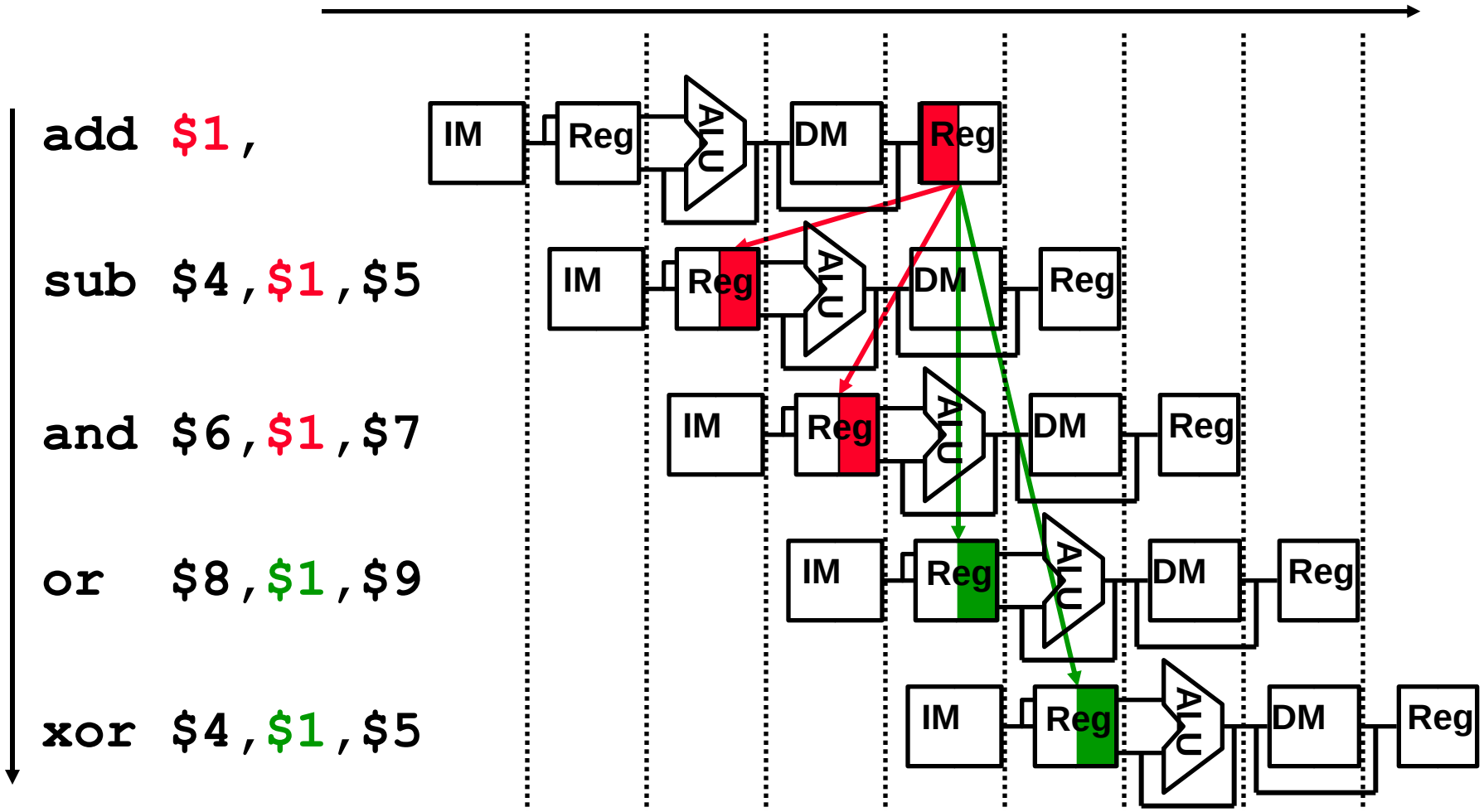
Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!



Example

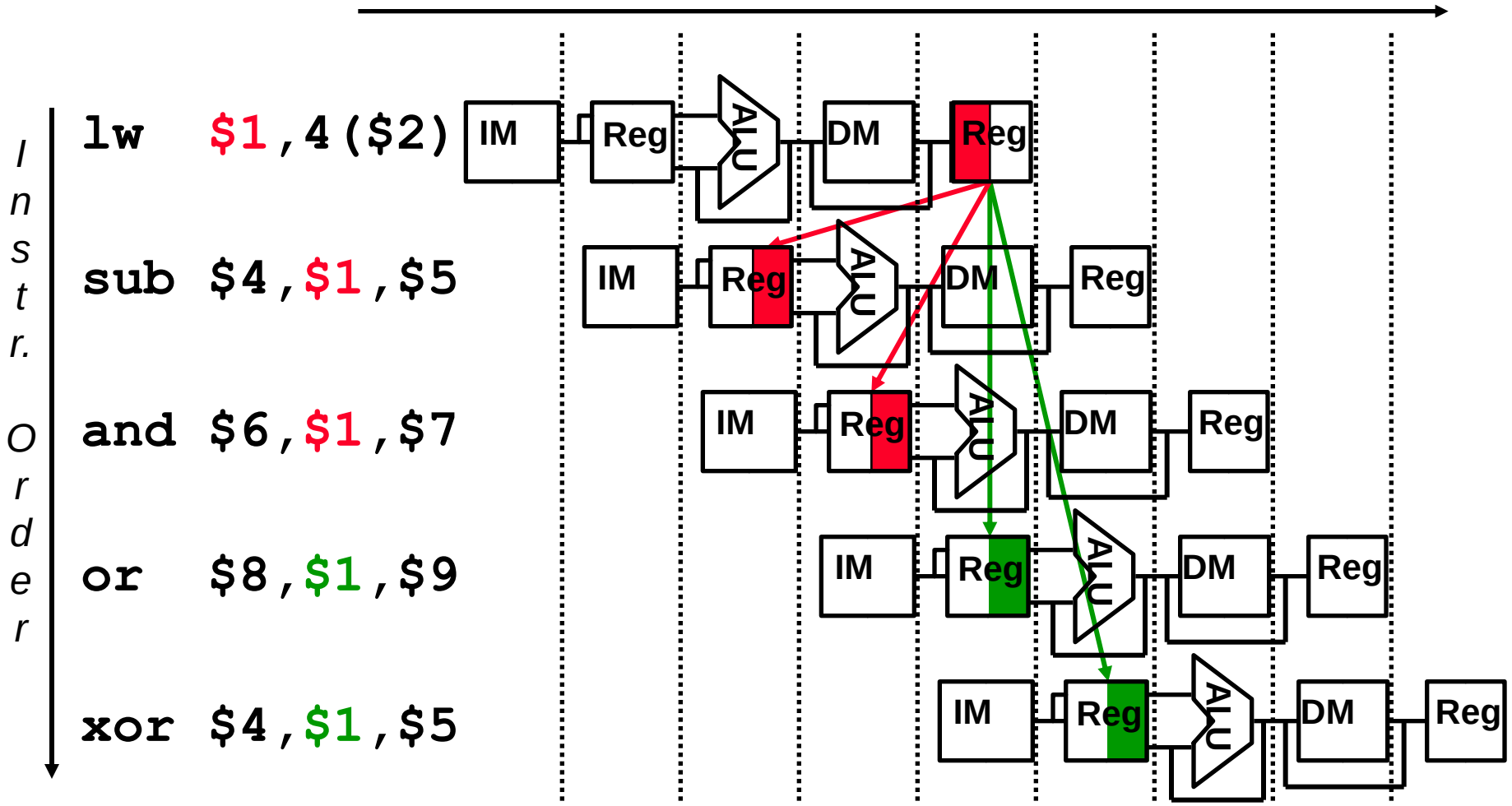
- Dependencies backward in time cause hazards



- Read before write data hazard

Example

- Dependencies backward in time cause hazards

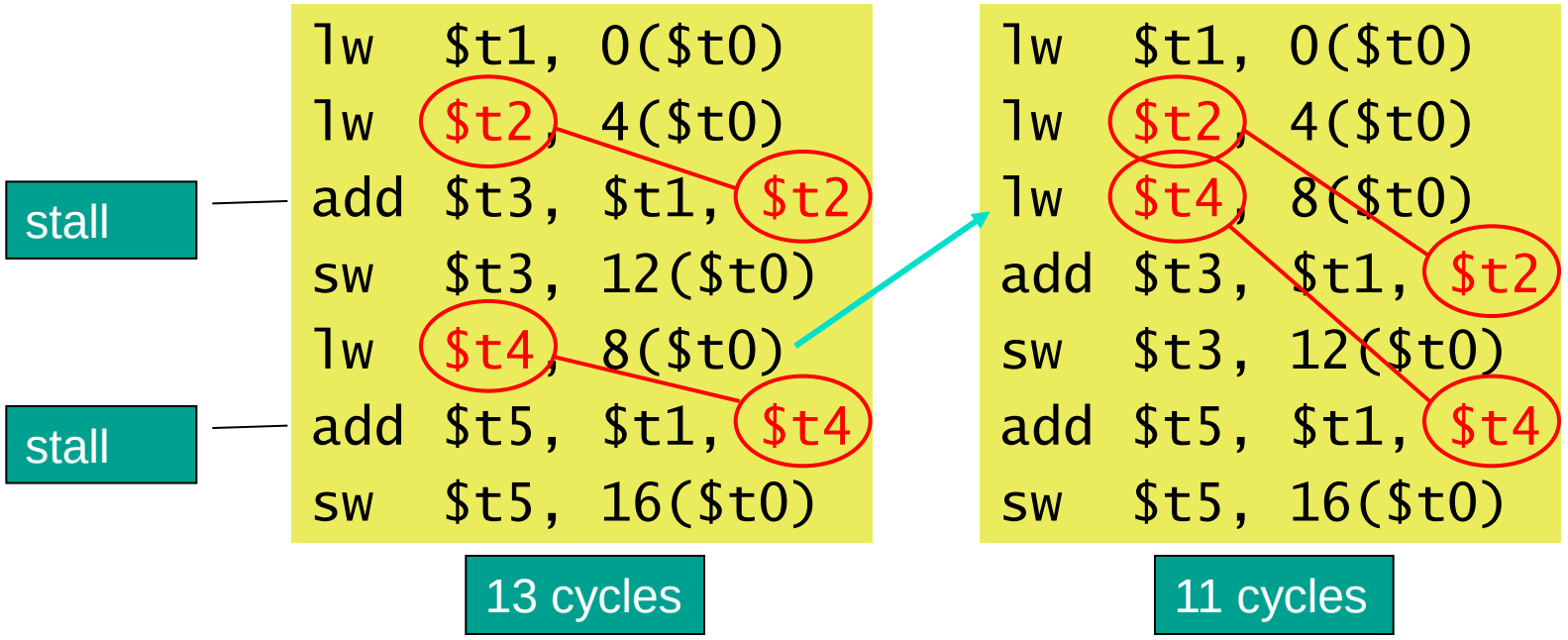


- Load-use data hazard

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code:

```
A = B + E;
C = B + F;
```

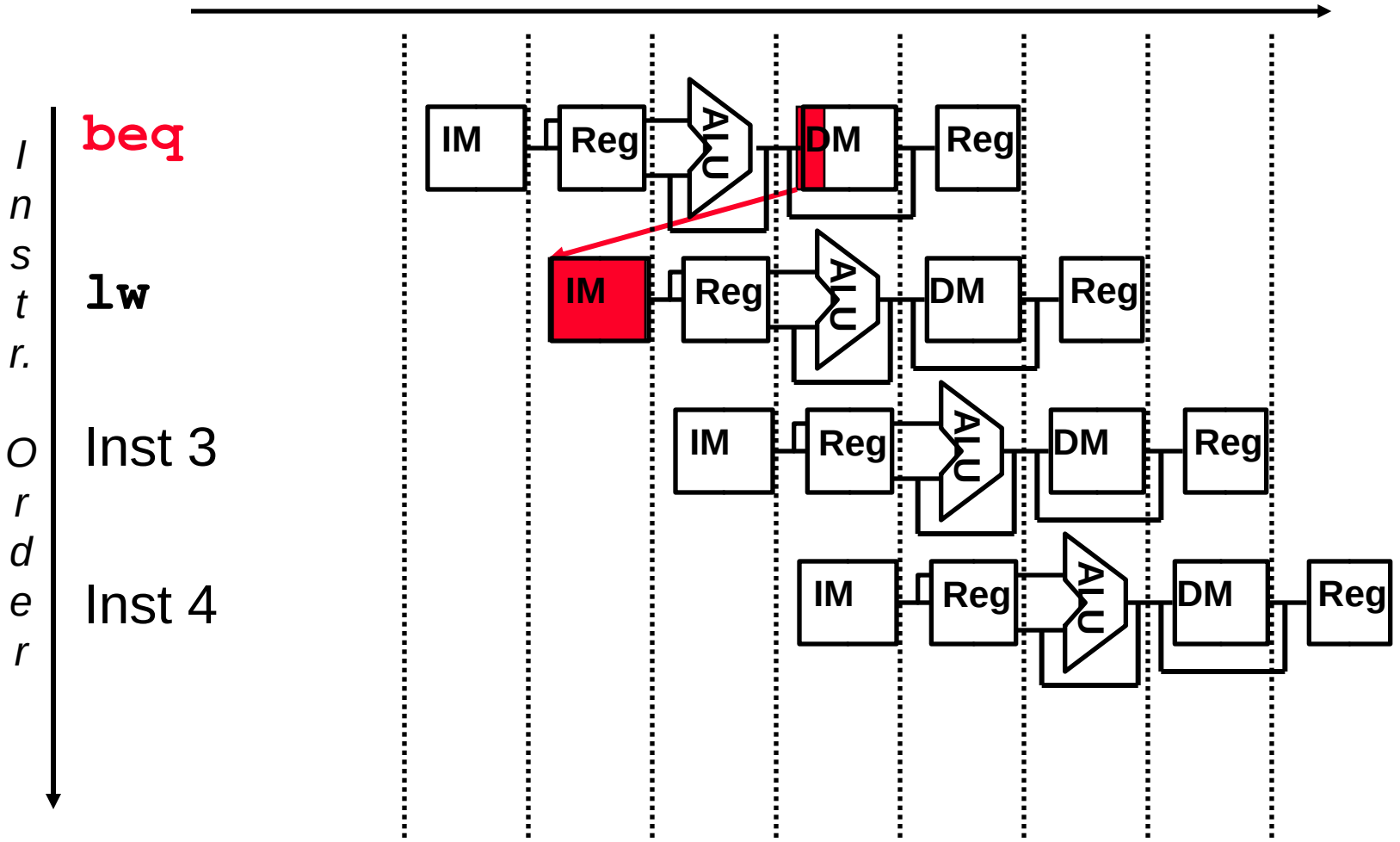


Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

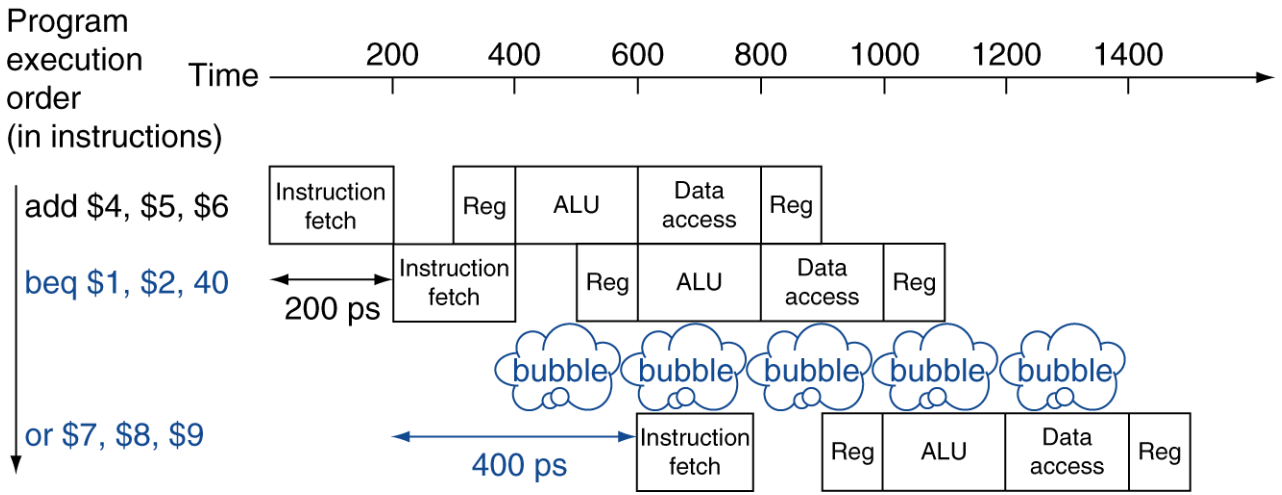
Branch Instructions Cause Control Hazards

- Dependencies backward in time cause hazards



Stall on Branch

- Naïve approach: Wait until branch outcome determined before fetching next instruction



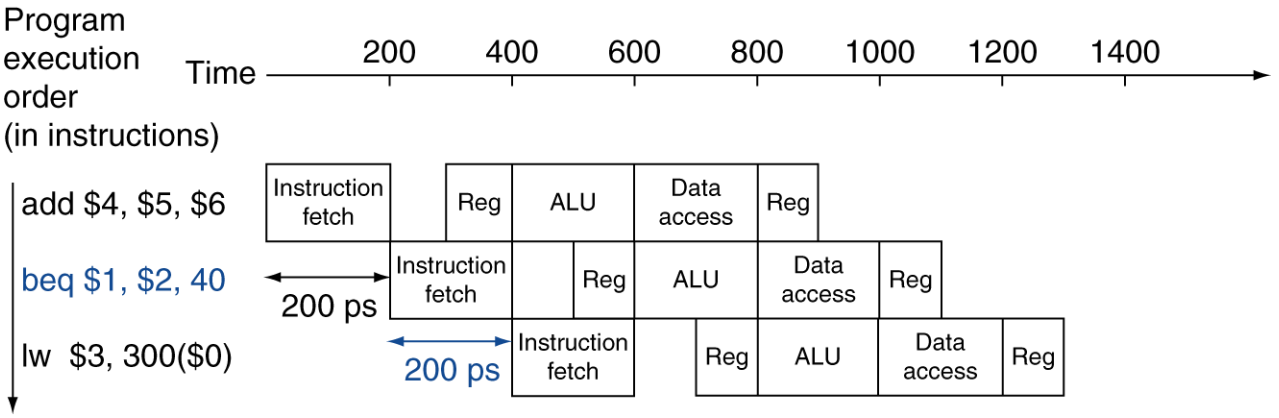
Performance affect: with 17% of instructions in SPECint2006 are branches, if each branch take one cycle for the stall, then performance will be 17% slower. (CPI = 1.17)

Branch Prediction

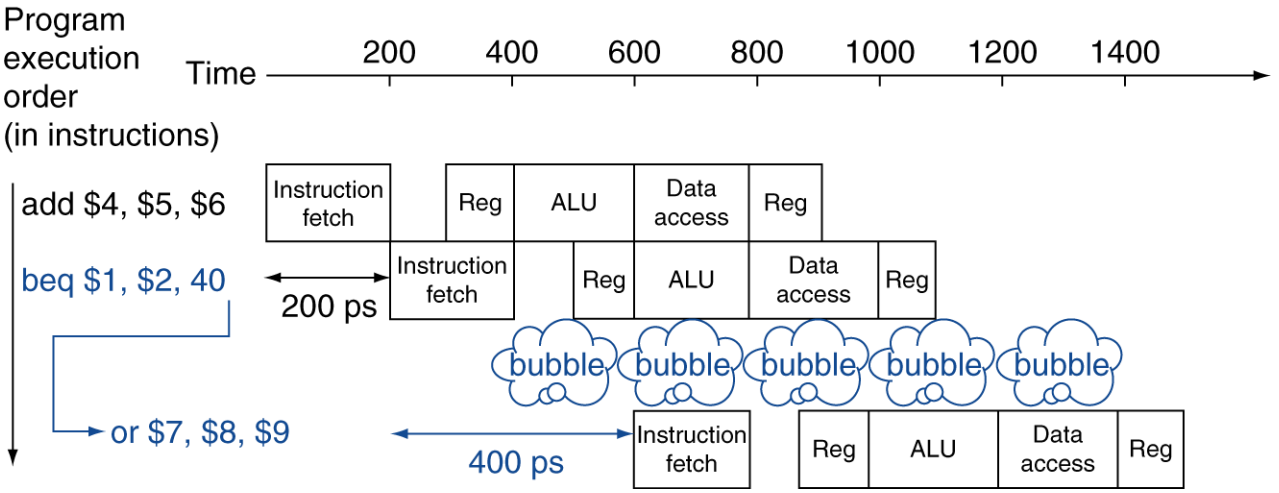
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



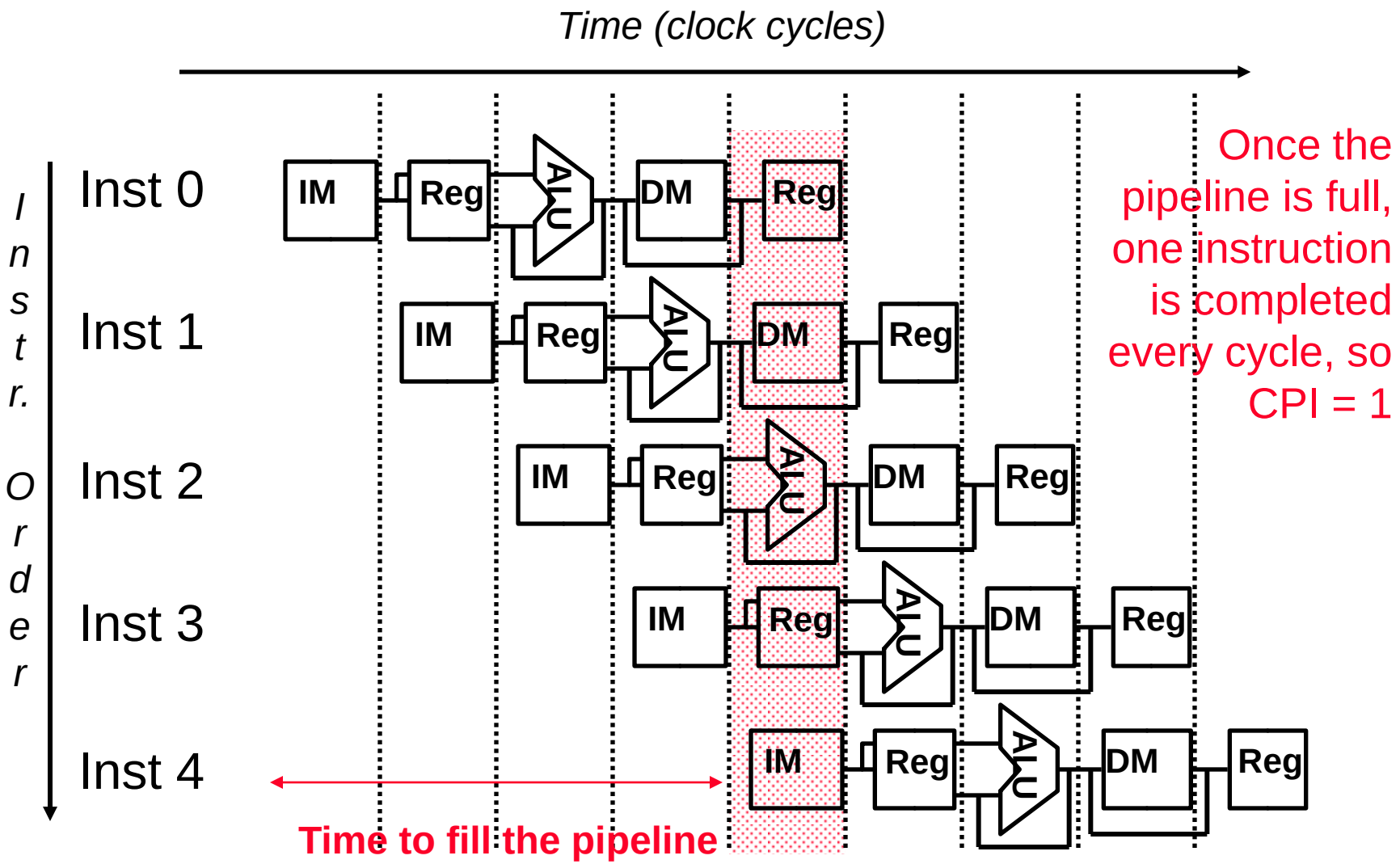
Prediction incorrect



More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history
  - As good as > 90% accuracy

Summary: Why Pipeline? For Performance!



Summary

- All modern day processors use pipelining
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Potential speedup: a CPI of 1 and a fast CC
- Pipeline rate limited by slowest pipeline stage
  - Unbalanced pipe stages makes for inefficiencies
  - The time to "fill" pipeline and time to "drain" it can impact speedup for deep pipelines and short code runs
- Must detect and resolve hazards
  - Stalling negatively affects CPI (makes CPI less than the ideal of 1)

Example

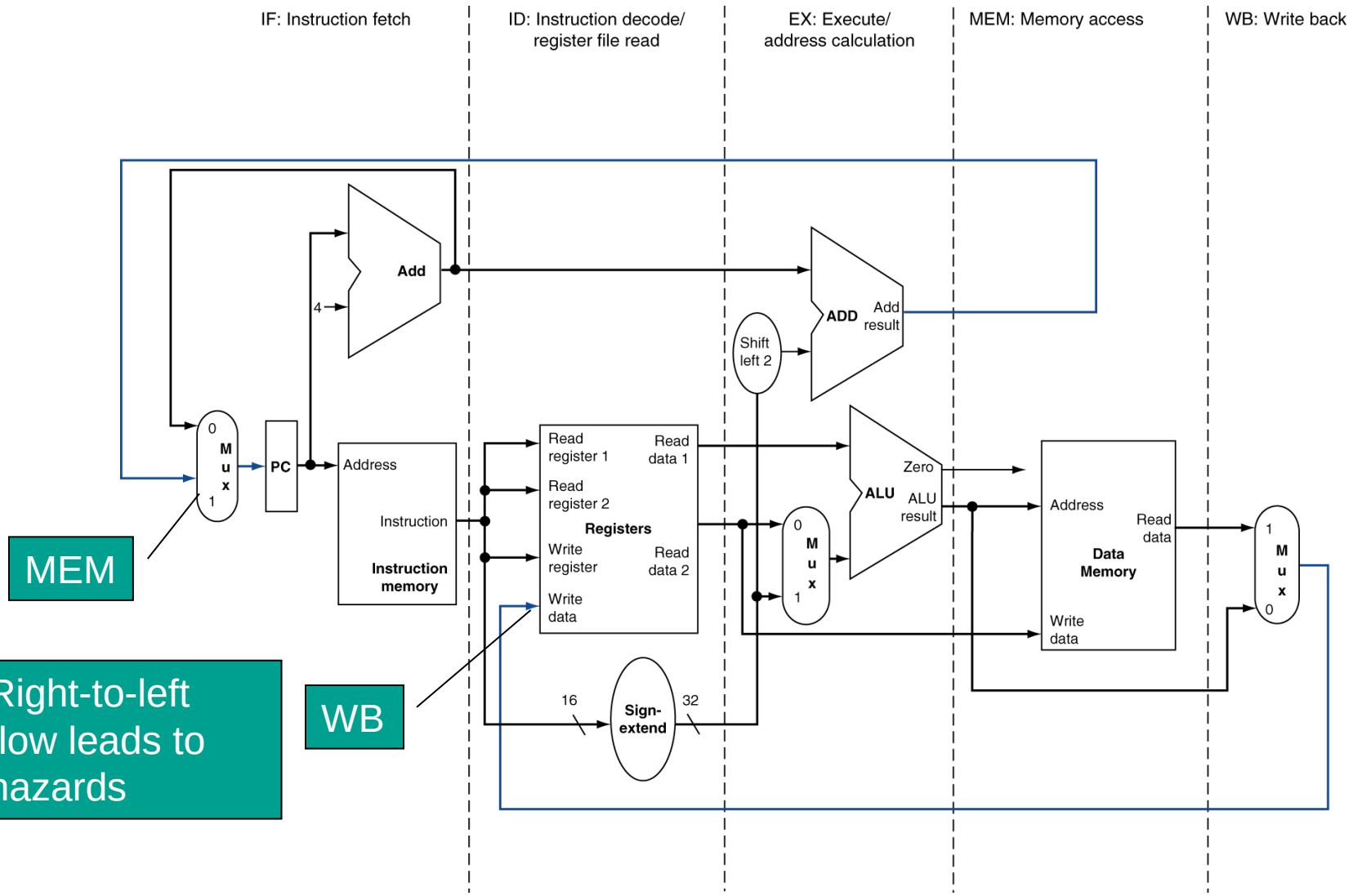
- Detect stall and bubble in the code snippets below

```
lw   $t0,0($t0)
add  $t1,$t0,$t0
```

```
add  $t1,$t0,$t0
addi $t2,$t0,#5
addi $t4,$t1,#5
```

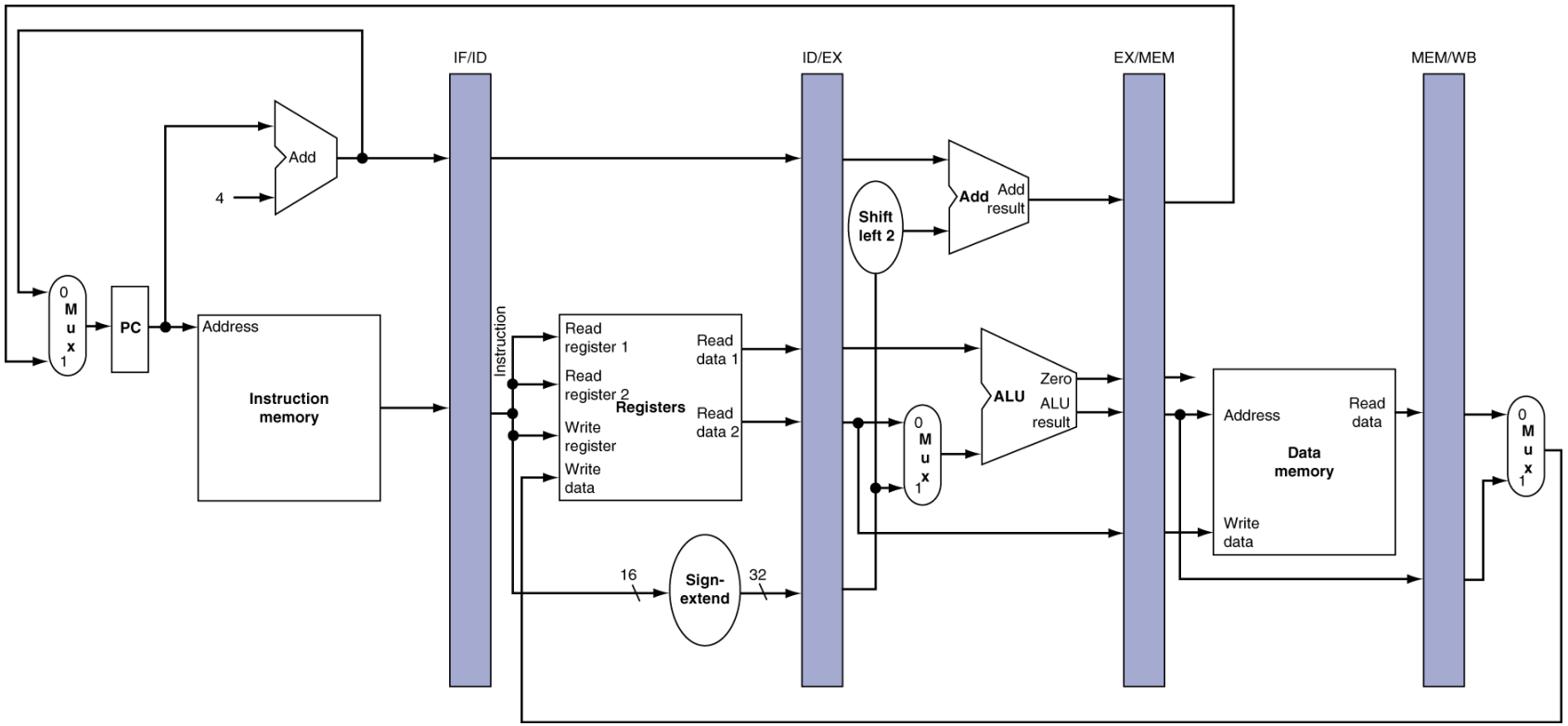
```
addi $t1,$t0,#1
addi $t2,$t0,#2
addi $t3,$t0,#2
addi $t3,$t0,#4
addi $t5,$t0,#5
```

MIPS Pipelined Datapath



Pipeline registers

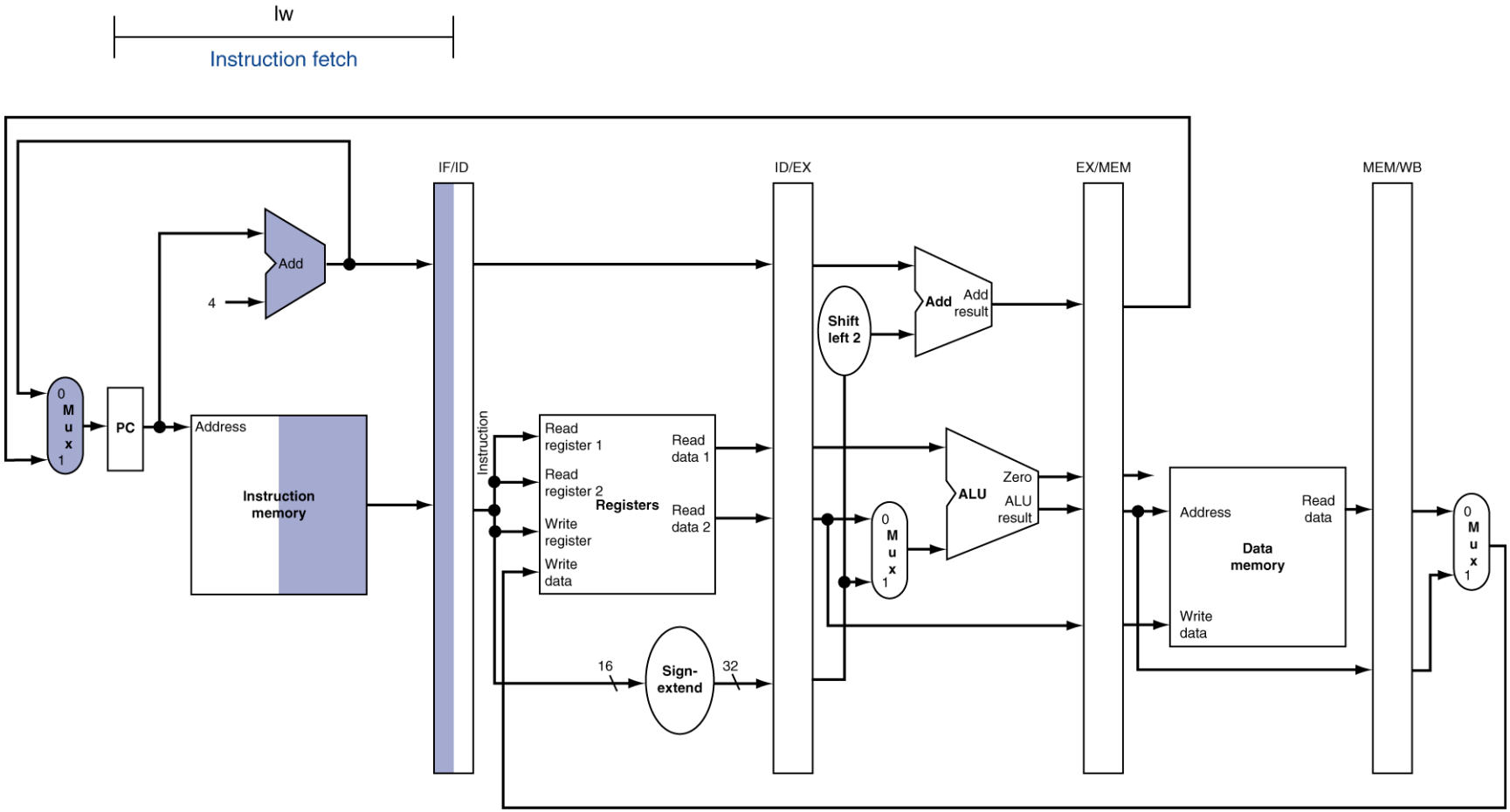
- Need registers between stages
  - To hold information produced in previous cycle



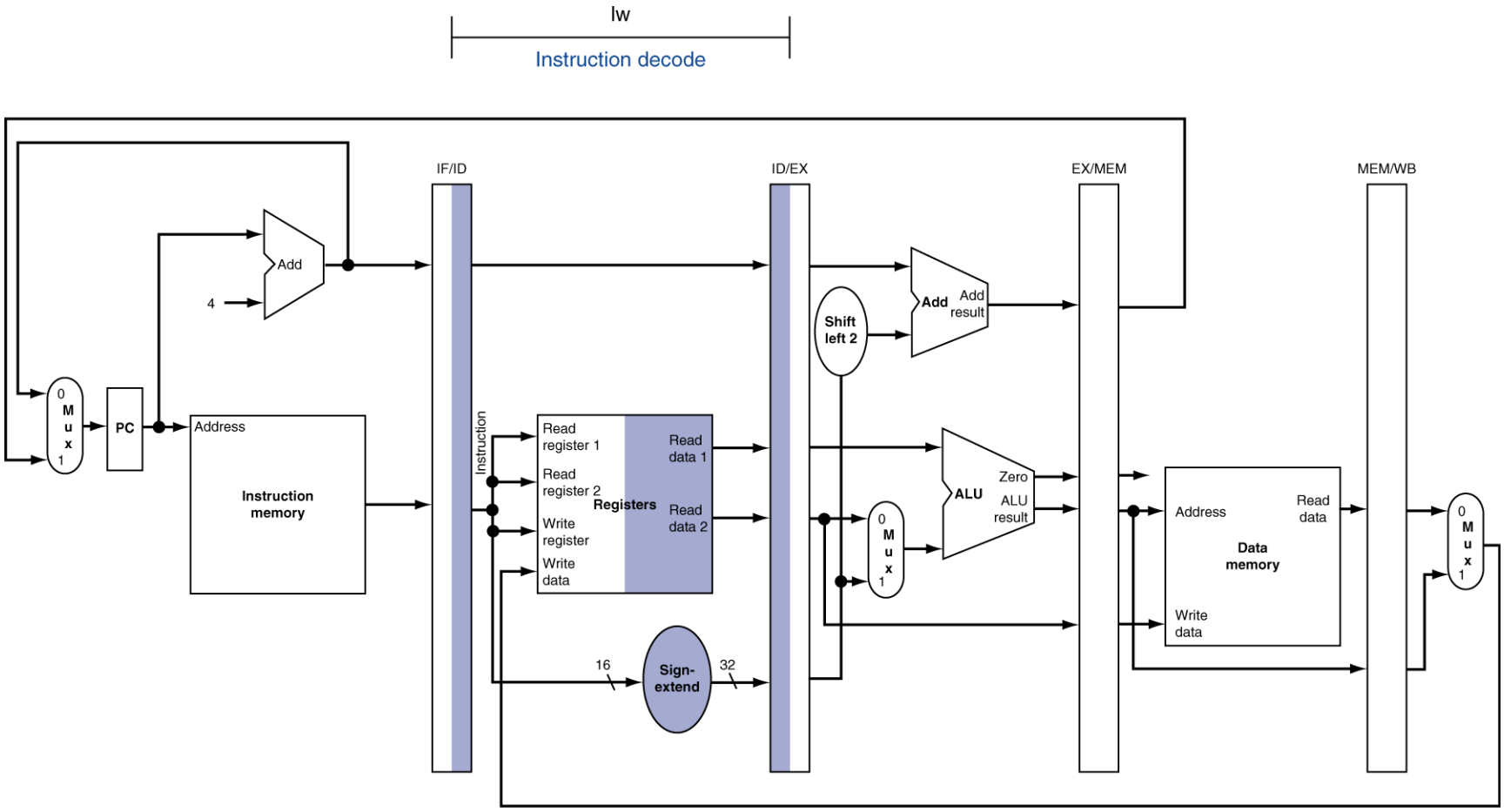
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store

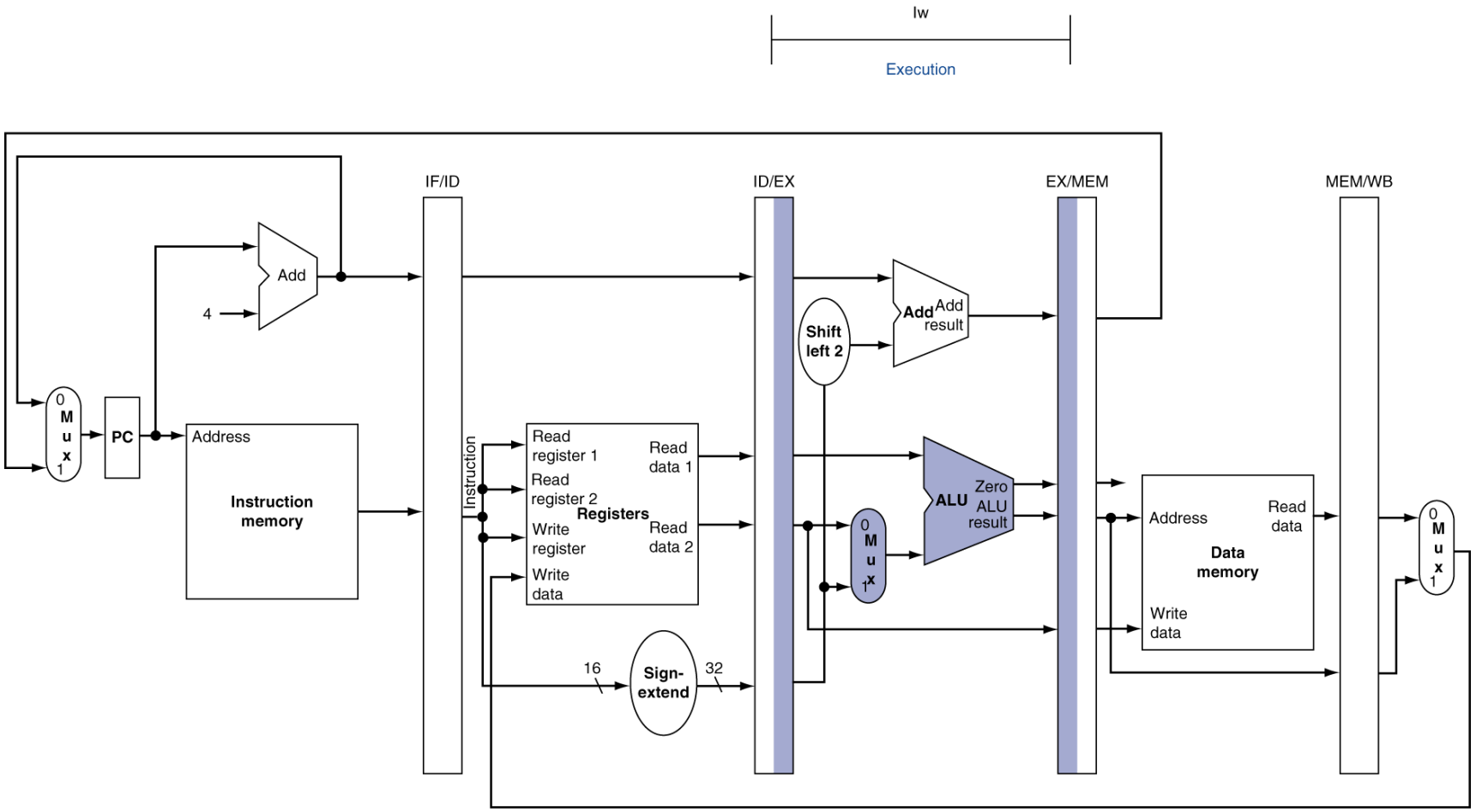
IF for Load, Store, …



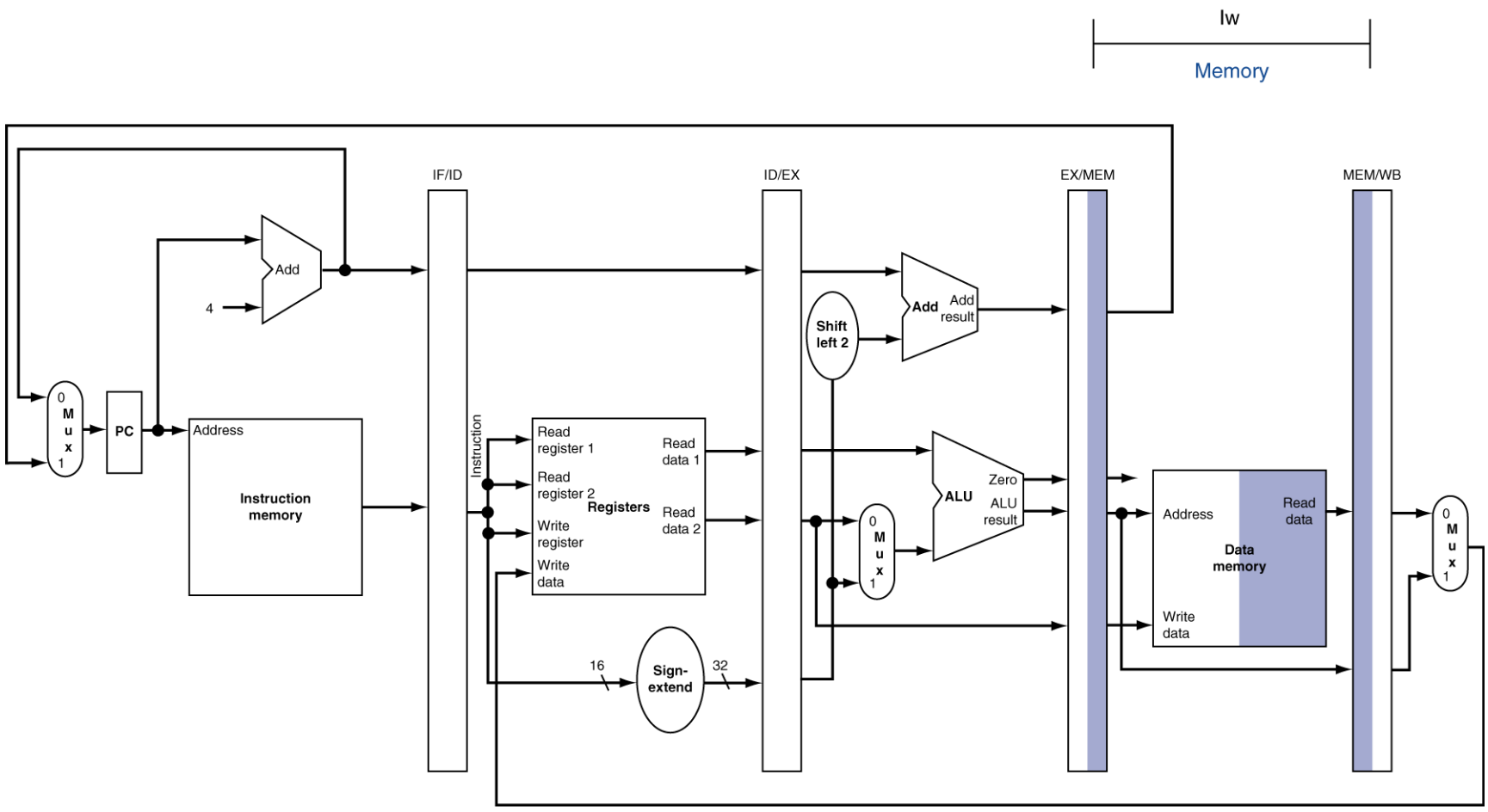
ID for Load, Store, …



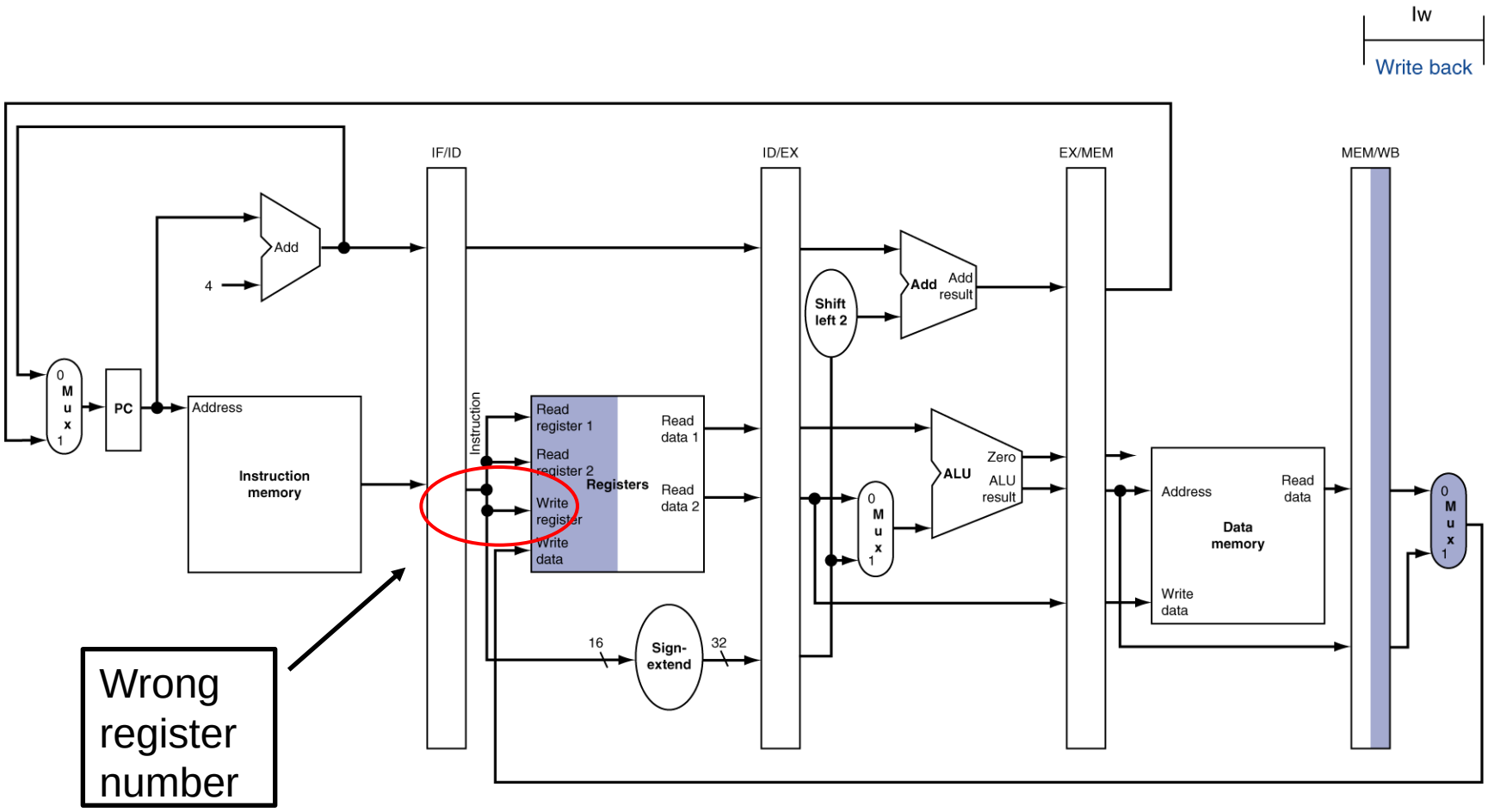
EX for Load


lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32

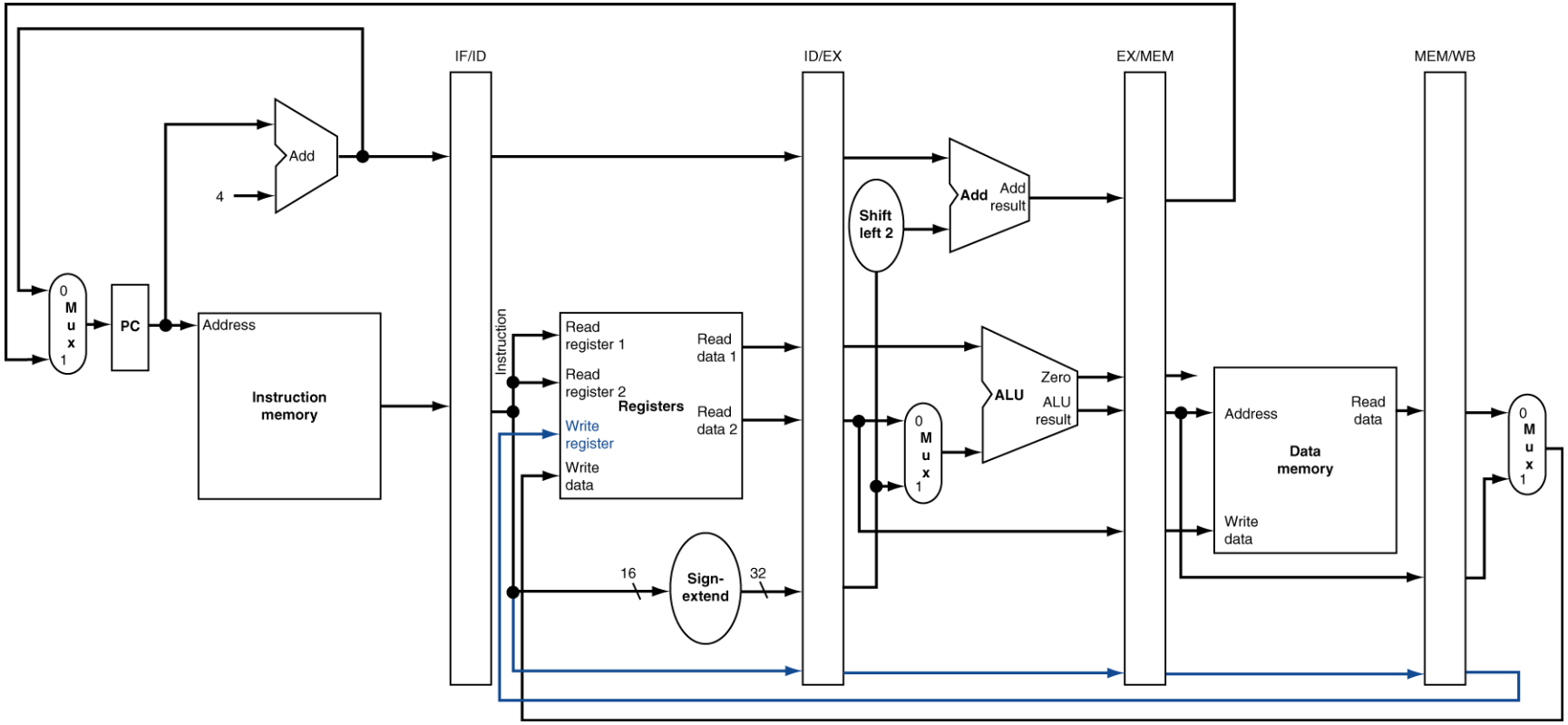
MEM for Load


lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
16
Sign-extend
32
Address
Data memory
Read data
Write data

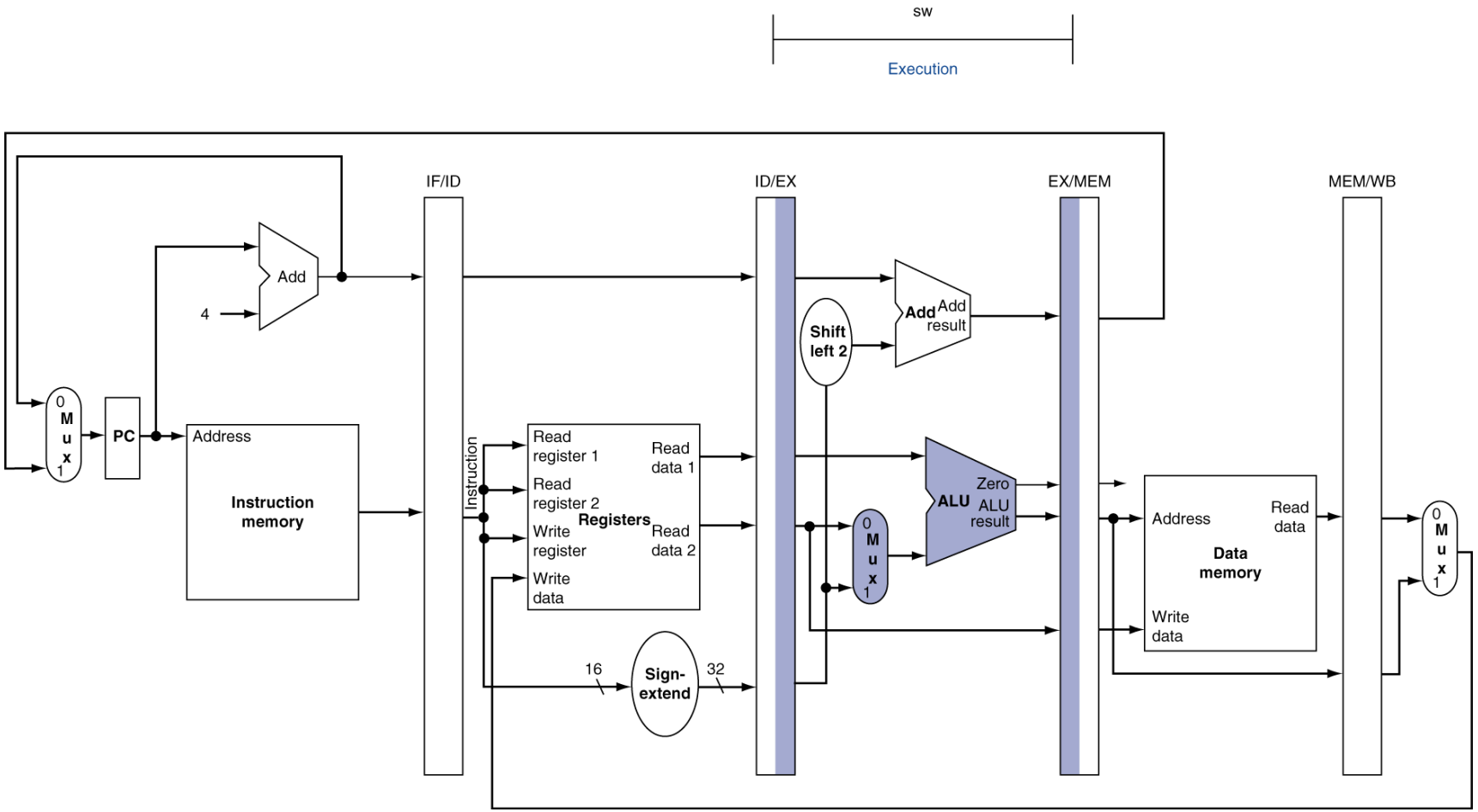
WB for Load


lw
Write back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32
Wrong register number

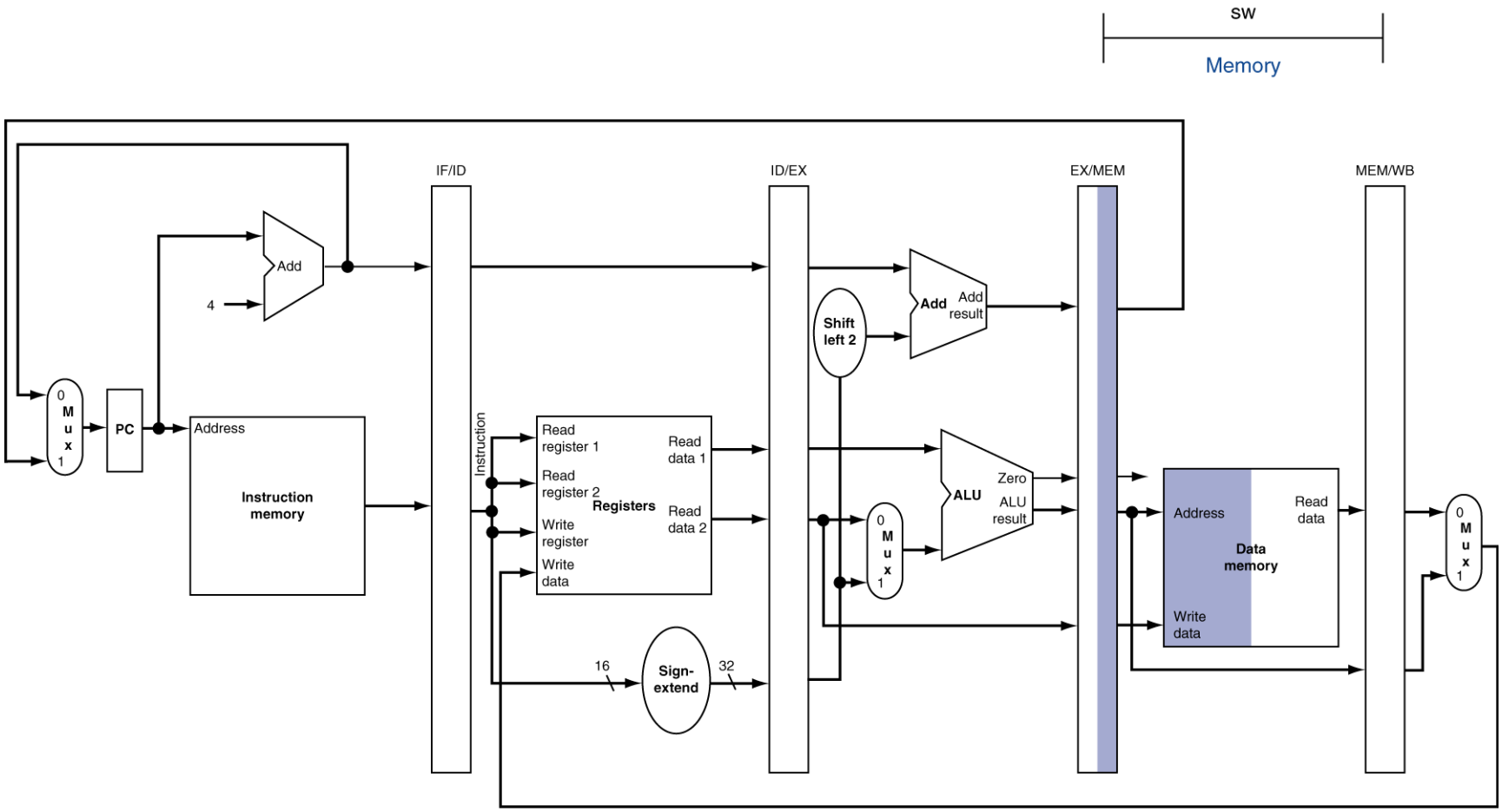
Corrected Datapath for Load



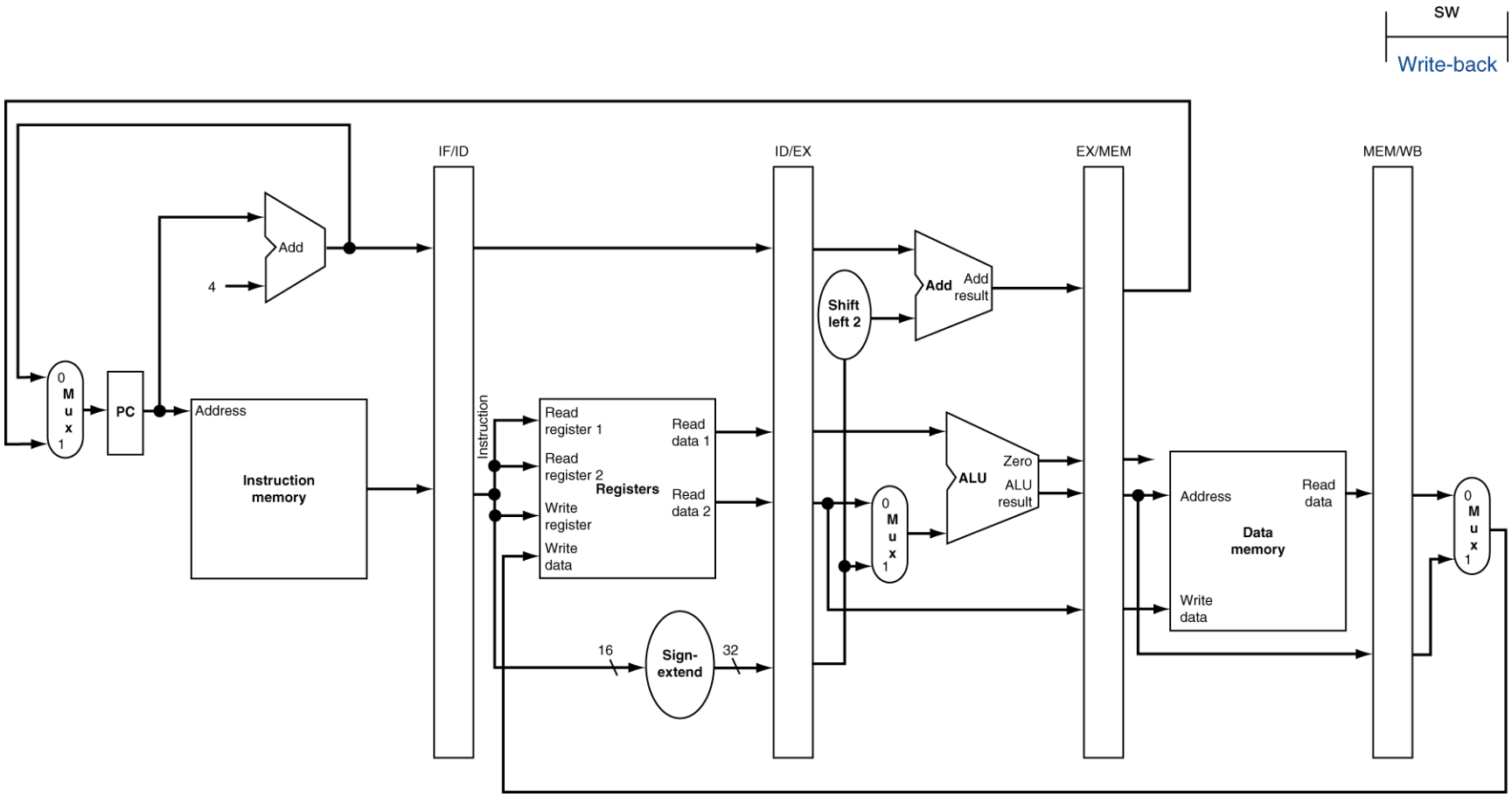
EX for Store


sw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
16
Sign-extend
32

MEM for Store

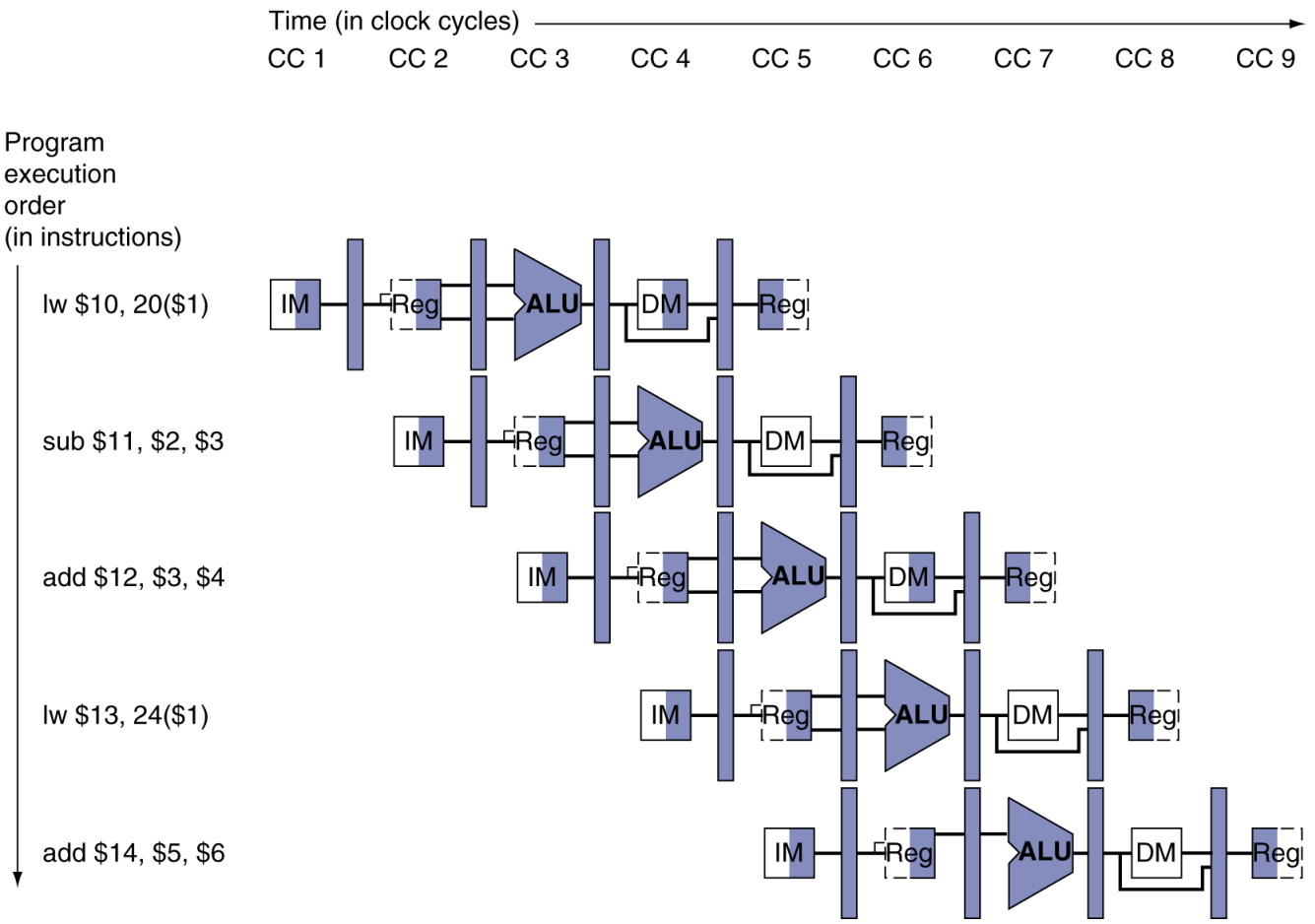


WB for Store


SW
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data

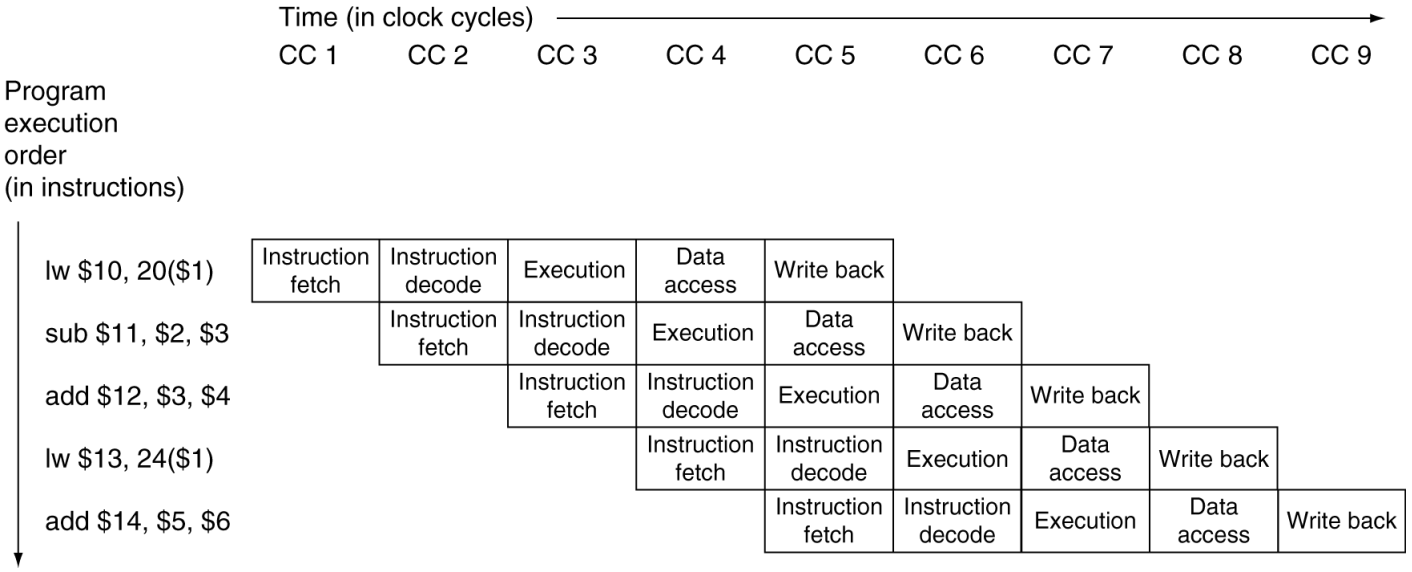
Multi-Cycle Pipeline Diagram

- Form showing resource usage



Multi-Cycle Pipeline Diagram

- Traditional form



Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back