CMPEN431
Computer Architecture
Fall 2017

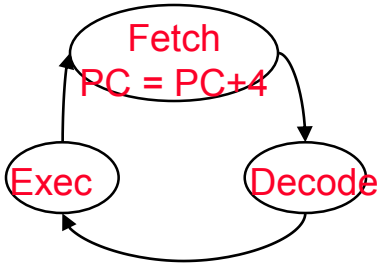
The Single Cycle Processor, Part A

Mahmut Taylan Kandemir (www.cse.psu.edu/~kandemir)

[Adapted from *Computer Organization and Design, 5th Edition*,
Patterson & Hennessy, © 2014, Morgan Kaufmann]

The Processor: Datapath & Control

- Our implementation of the MIPS is simplified
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- Generic implementation
  - computers are state machines!
  - 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

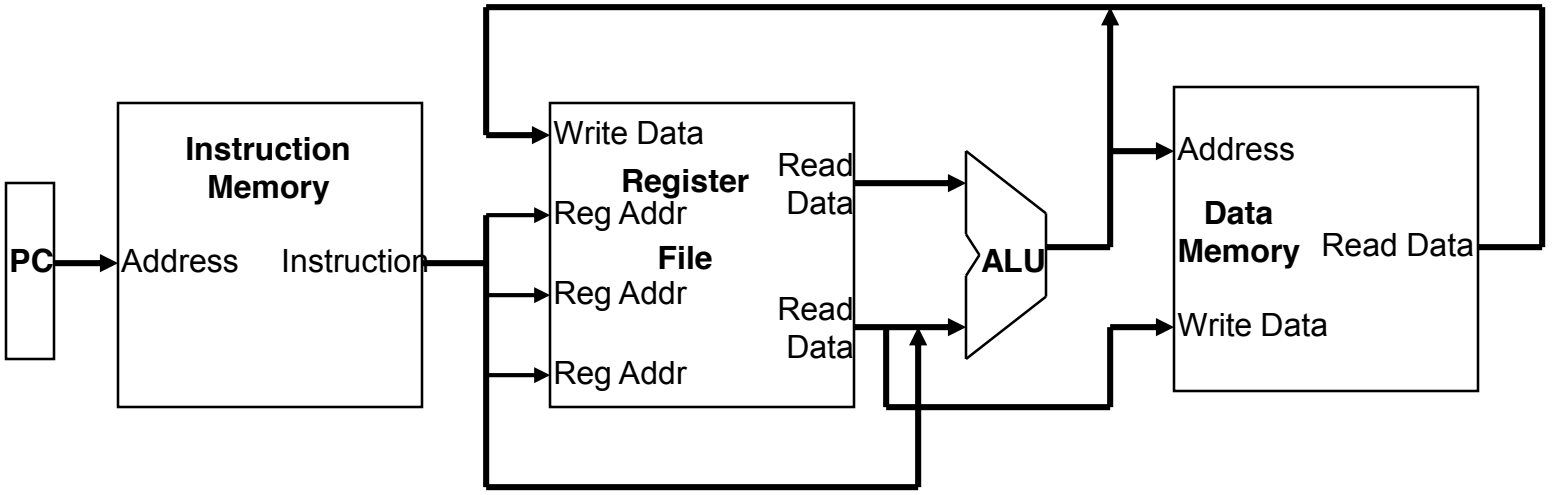


- All instructions (except `j`) use the (integer) ALU after reading the registers

How is the ALU used?
memory-reference? arithmetic? control flow?

Abstract Implementation, First Version

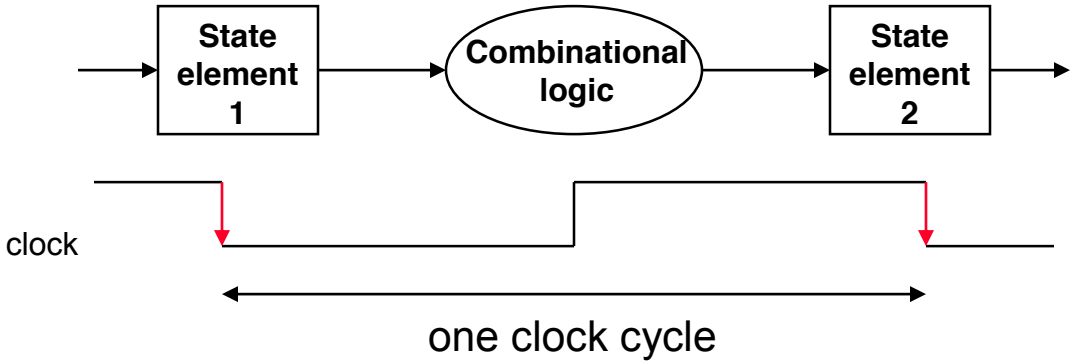
- Two types of functional units:
  - elements that operate on data values (**combinational**)
  - elements that contain state (**sequential**)



- Single cycle operation
- Split memory model – one memory for instructions (really L1I$) and one memory (really L1D$) for data

Aside: Clocking Methodologies

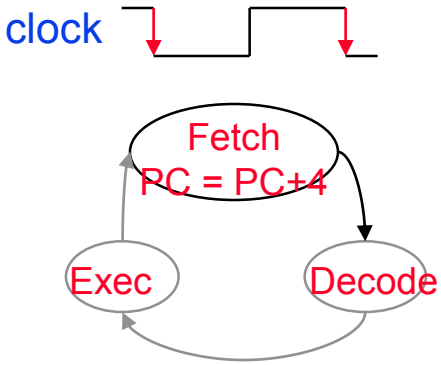
- The clocking methodology defines when data in a state element is valid and stable relative to the clock
  - State elements – a memory element such as a register
  - Edge-triggered – all state changes occur on a clock edge
- Typical execution
  - read contents of state elements -> send values through combinational logic -> write results to one or more state elements

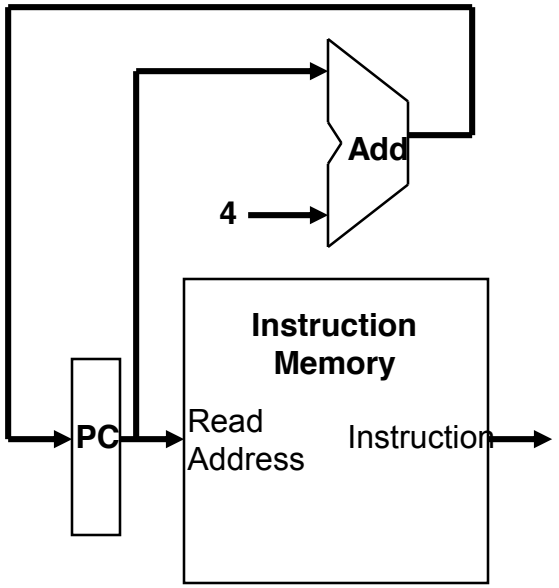


- Writes to state elements occur only when both the write control is asserted and the clock edge occurs

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

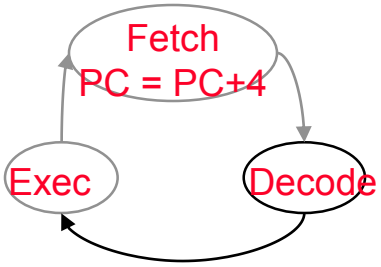




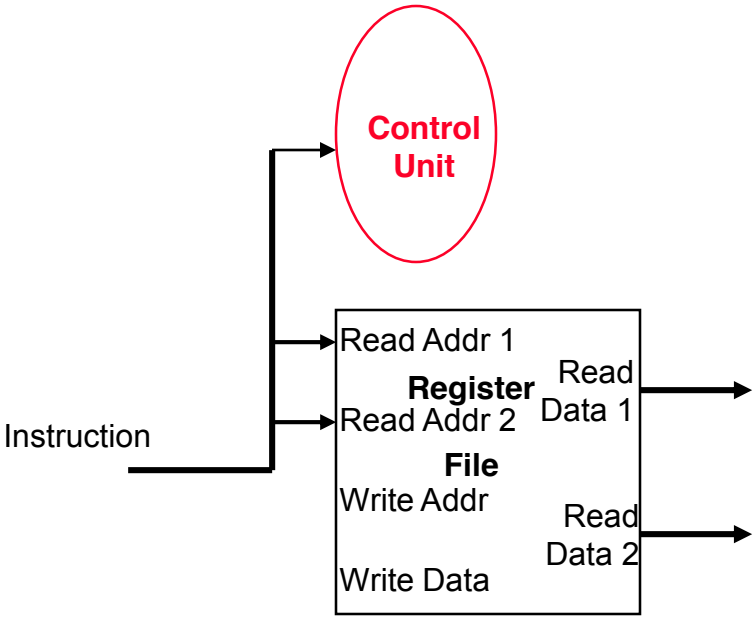
  - MIPS instructions are each four bytes long, so the PC should be incremented by four to read the next instruction in sequence
  - PC is updated every clock cycle, so it does *not* need an explicit write control signal just a clock signal

Decoding Instructions

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



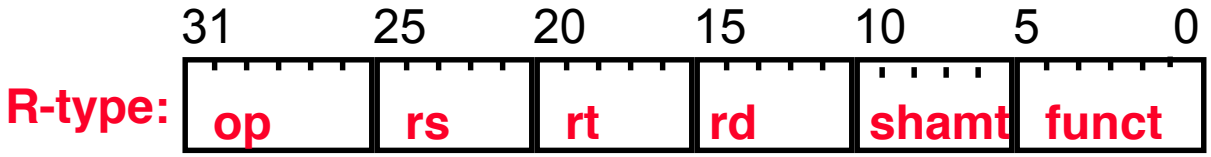
and



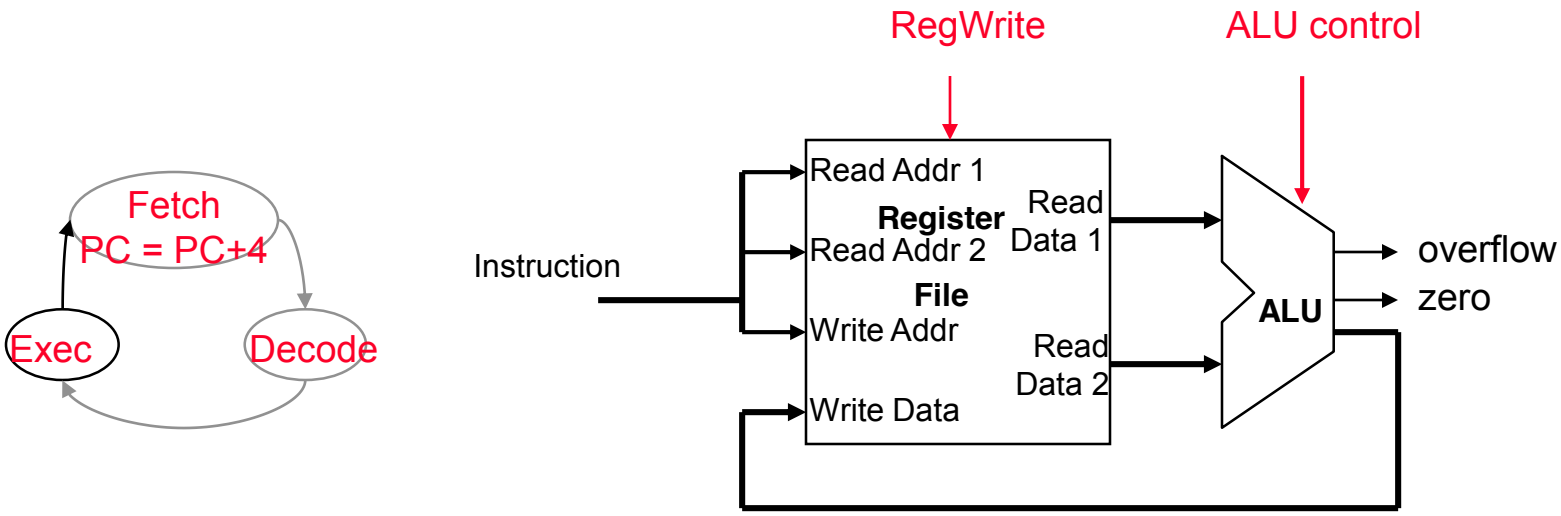
  - reading two values from the RF
    - RF addresses are contained in the instruction
    - Can read registers before the instruction is fully decoded, then ignore what you don't need. Benefits? Costs?

Executing R Format Operations

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



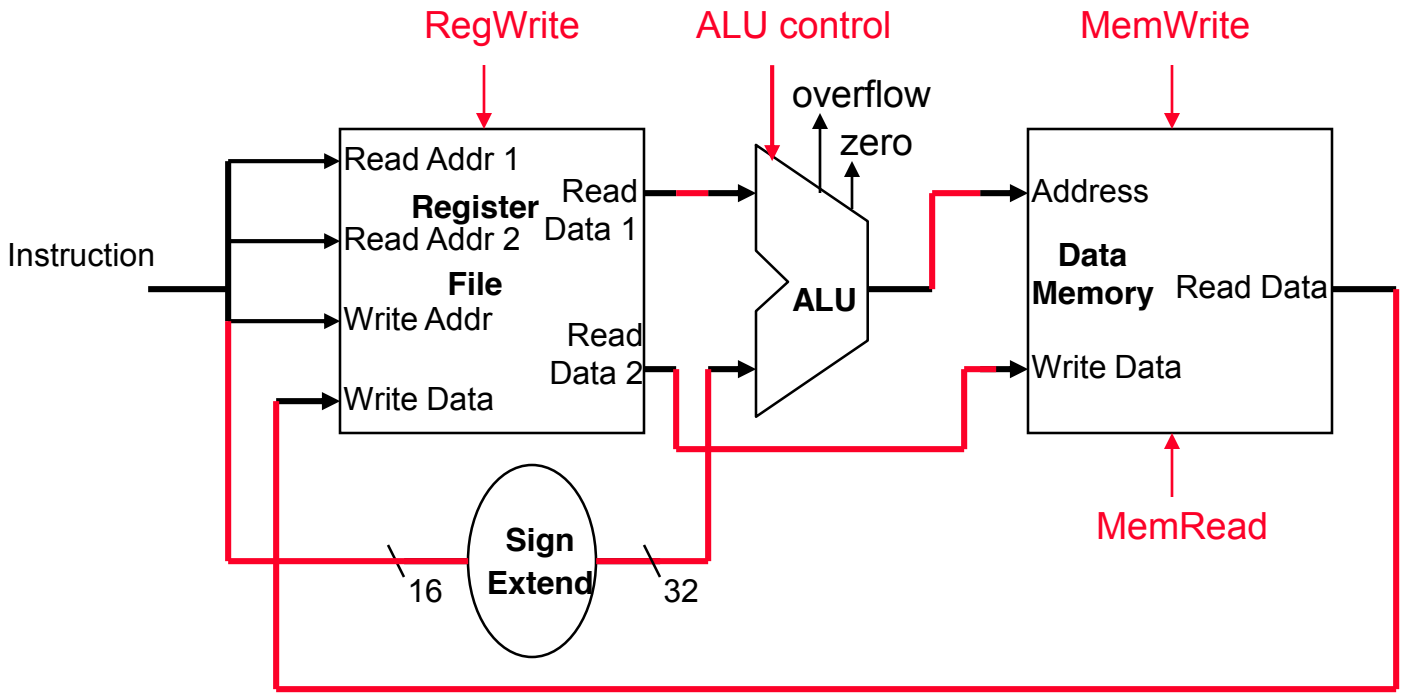
- perform operation (op and funct) on values in rs and rt
- store the result back into the RF (into location rd)



- Note that RF is not written every cycle (e.g., `sw`), so we need an explicit write control signal (RegWrite) for RF
- You can read from two registers at a time

Executing Load and Store Operations

- Load and store operations involves
  - compute memory address by adding the base register (read from the RF during decode) to the 16-bit signed-extended offset field in the instruction
  - store value (read from the RF during decode) written to the Data Memory
  - load value, read from the Data Memory, written to the RF



Accessing Data Memory

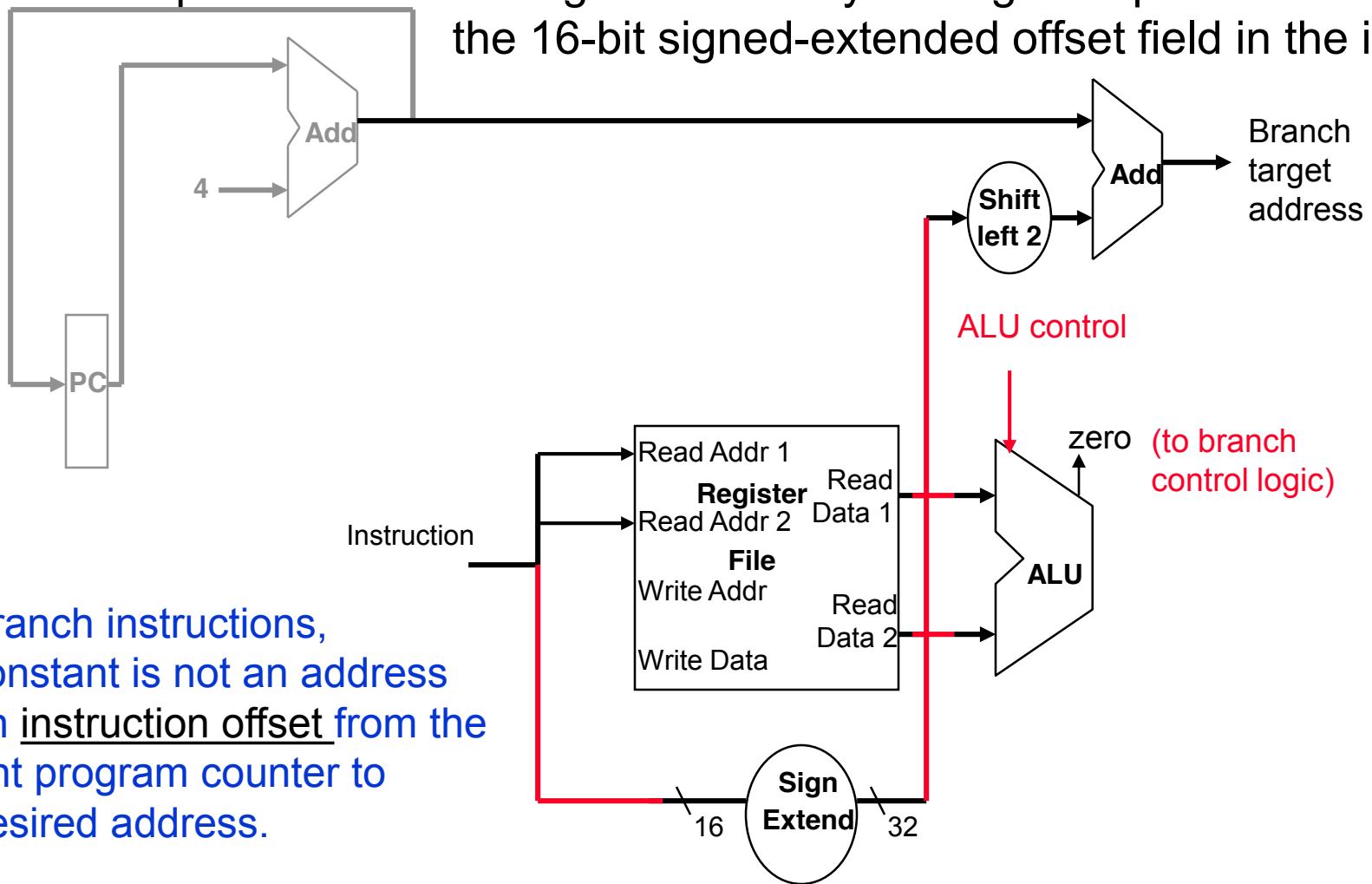
- For an instruction like lw $t0, –4($sp), the base register $sp is added to the *sign-extended* constant to get a data memory address.
- This means the ALU must accept *either* a register operand for arithmetic instructions, *or* a sign-extended immediate operand for lw and sw.
- We'll add a multiplexer, controlled by ALUSrc, to select either a register operand (0) or a constant operand (1).

Executing Branch Operations

- Fetch the instruction, like beq $at, $0, offset, from memory.
- Read the source registers, $at and $0, from the register file.
- Compare the values by subtracting them in the ALU.
- If the subtraction result is 0, the source operands were equal and the PC should be loaded with the target address, PC + 4 + (offset x 4).
- Otherwise the branch should not be taken, and the PC should just be incremented to PC + 4 to fetch the next instruction sequentially.

Executing Branch Operations

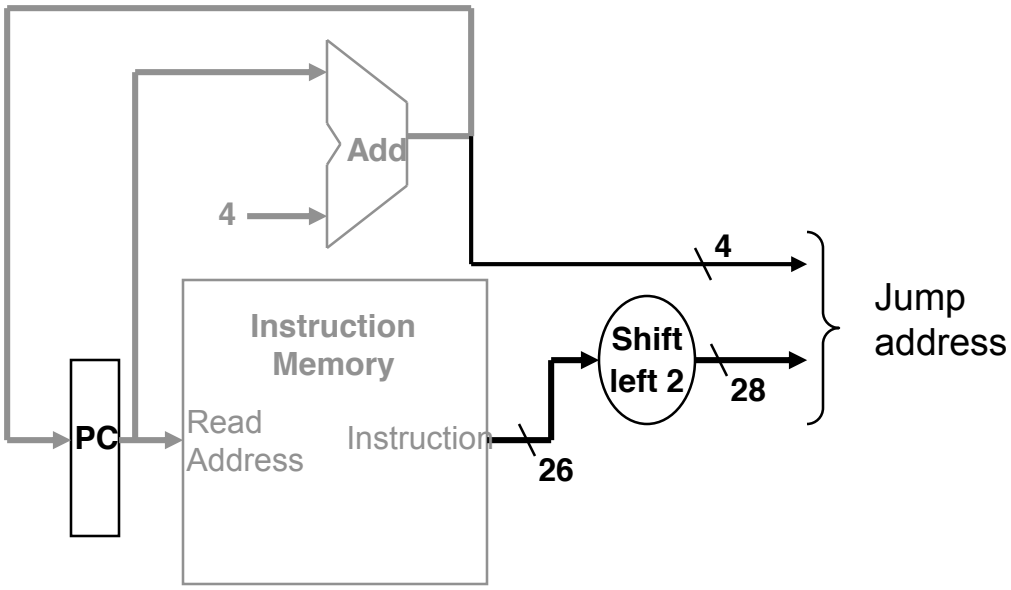
- Branch operations involves
  - compare the operands read from the RF during decode for equality (`zero` ALU output)
  - compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instr



For branch instructions, the constant is not an address but an instruction offset from the current program counter to the desired address.

Executing Jump Operations

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

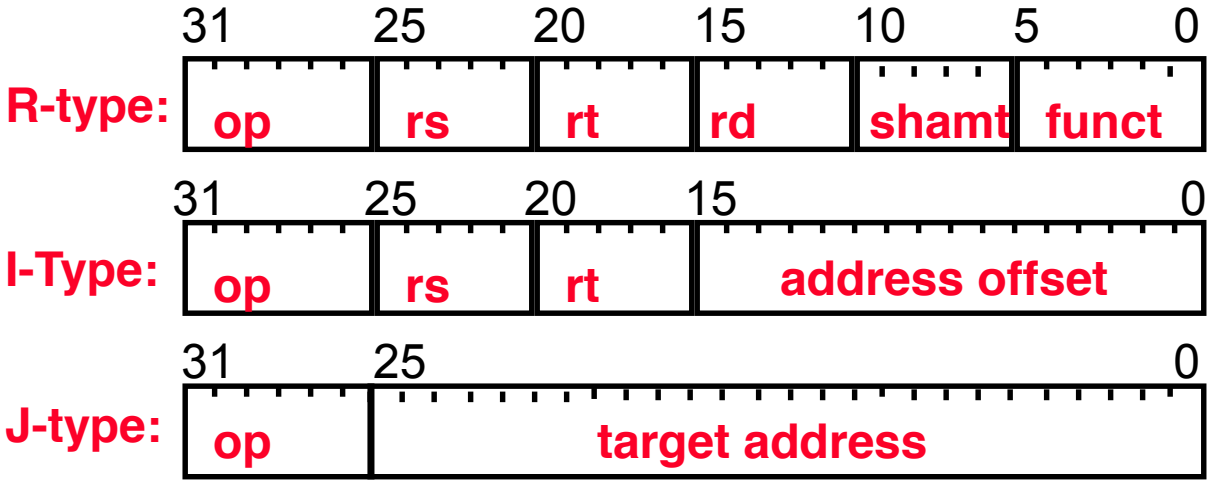


Creating a Single Datapath from the Parts

- 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 RF and Data Memory
- Cycle time is determined by length of the **longest** path

Adding the Control

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

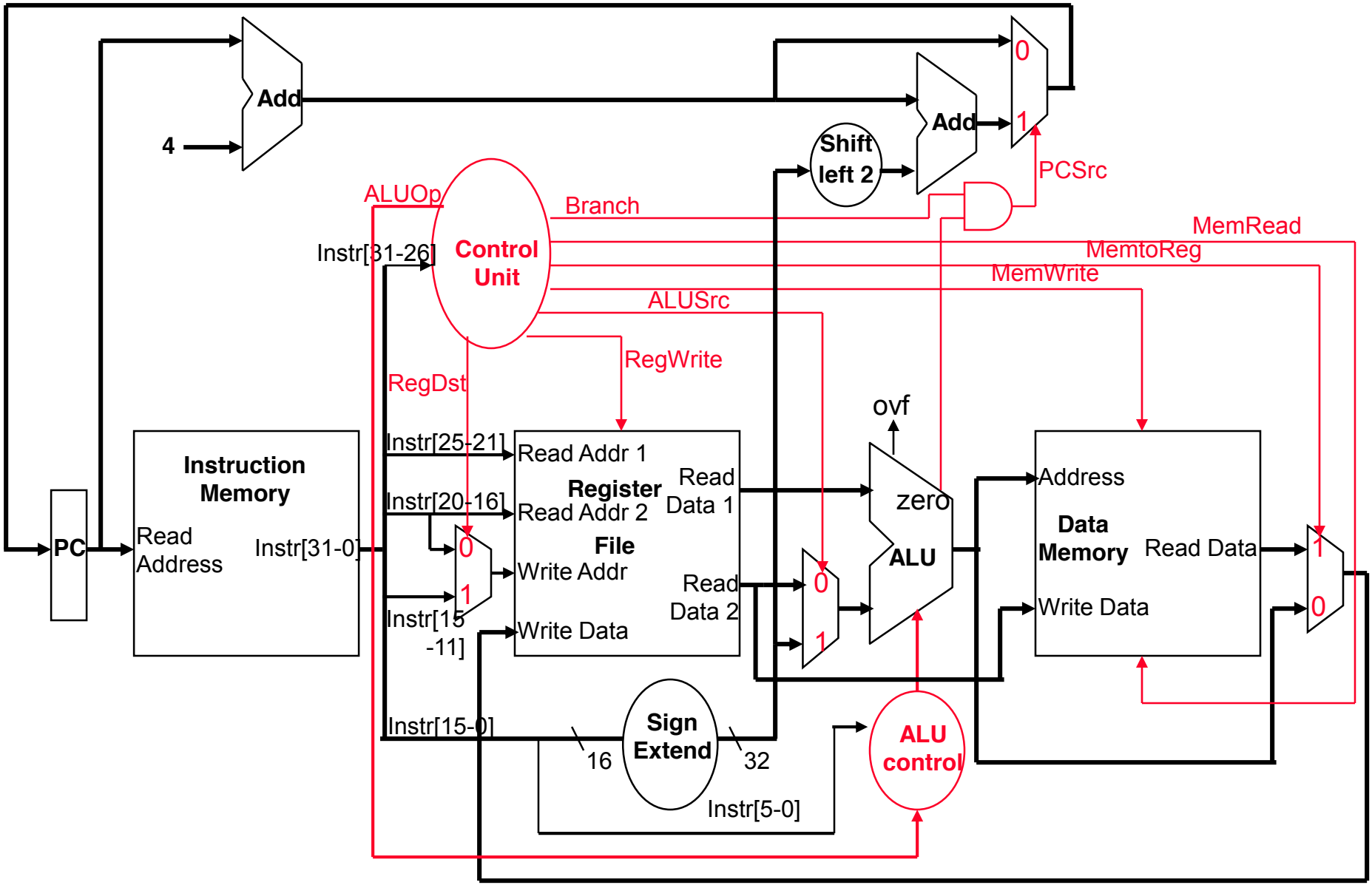


- Observations
  - op field always in bits 31-26
  - addr of registers to be read are always specified by the rs field (bits 25-21) and rt field (bits 20-16); for lw and sw, rs is the base register
  - addr. of register to be written is in one of two places – in rt (bits 20-16) for lw; in rd (bits 15-11)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

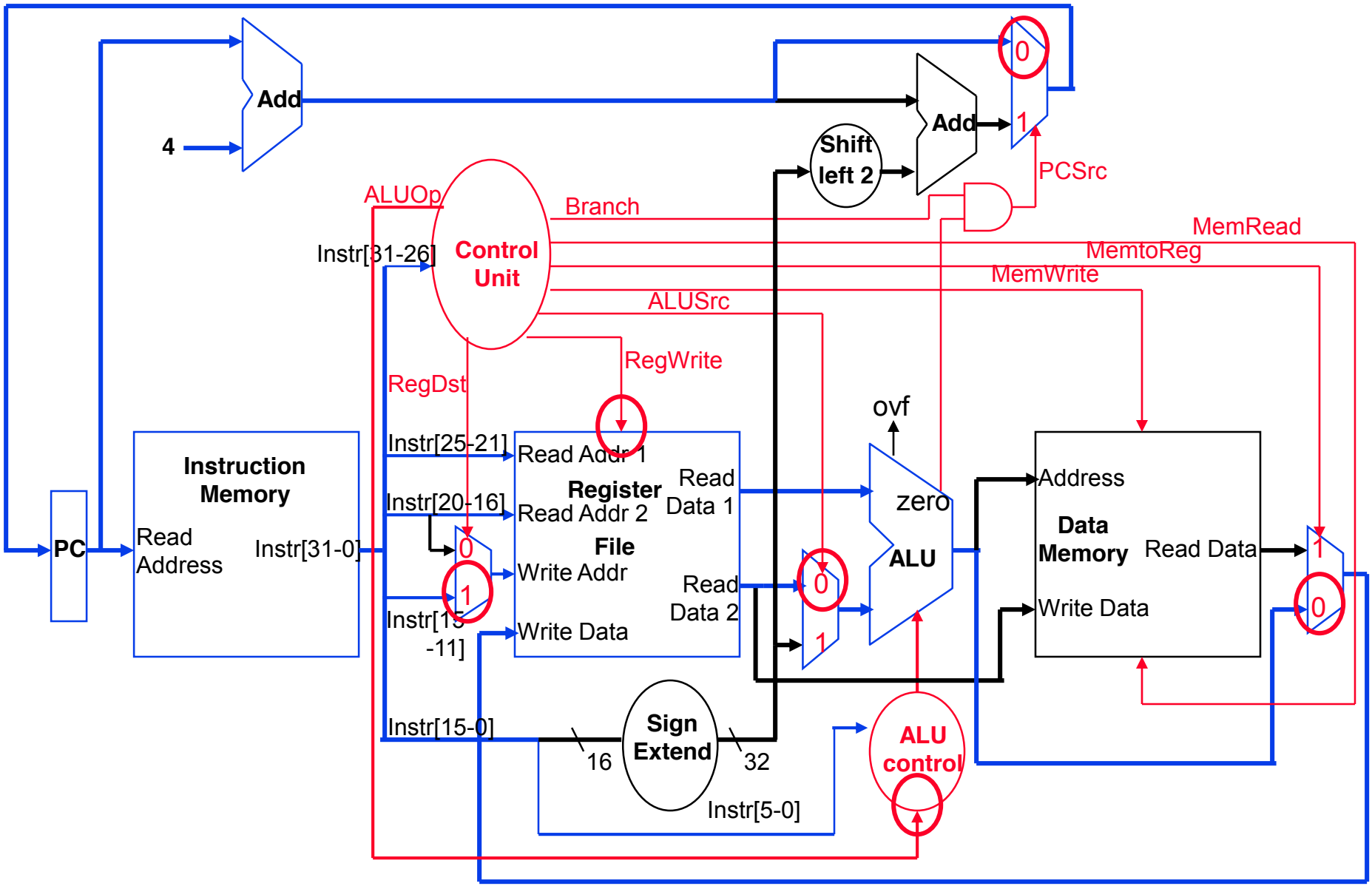
Control

- The control unit is responsible for setting all the control signals so that each instruction is executed properly.
  - The control unit's input is the 32-bit instruction word.
  - The outputs are signals to the datapath.
- Most of the signals can be generated from the instruction opcode alone, and not the entire 32-bit word.
- To illustrate the relevant control signals, we will show the route that is taken through the datapath by R-type, lw, sw and beq instructions.

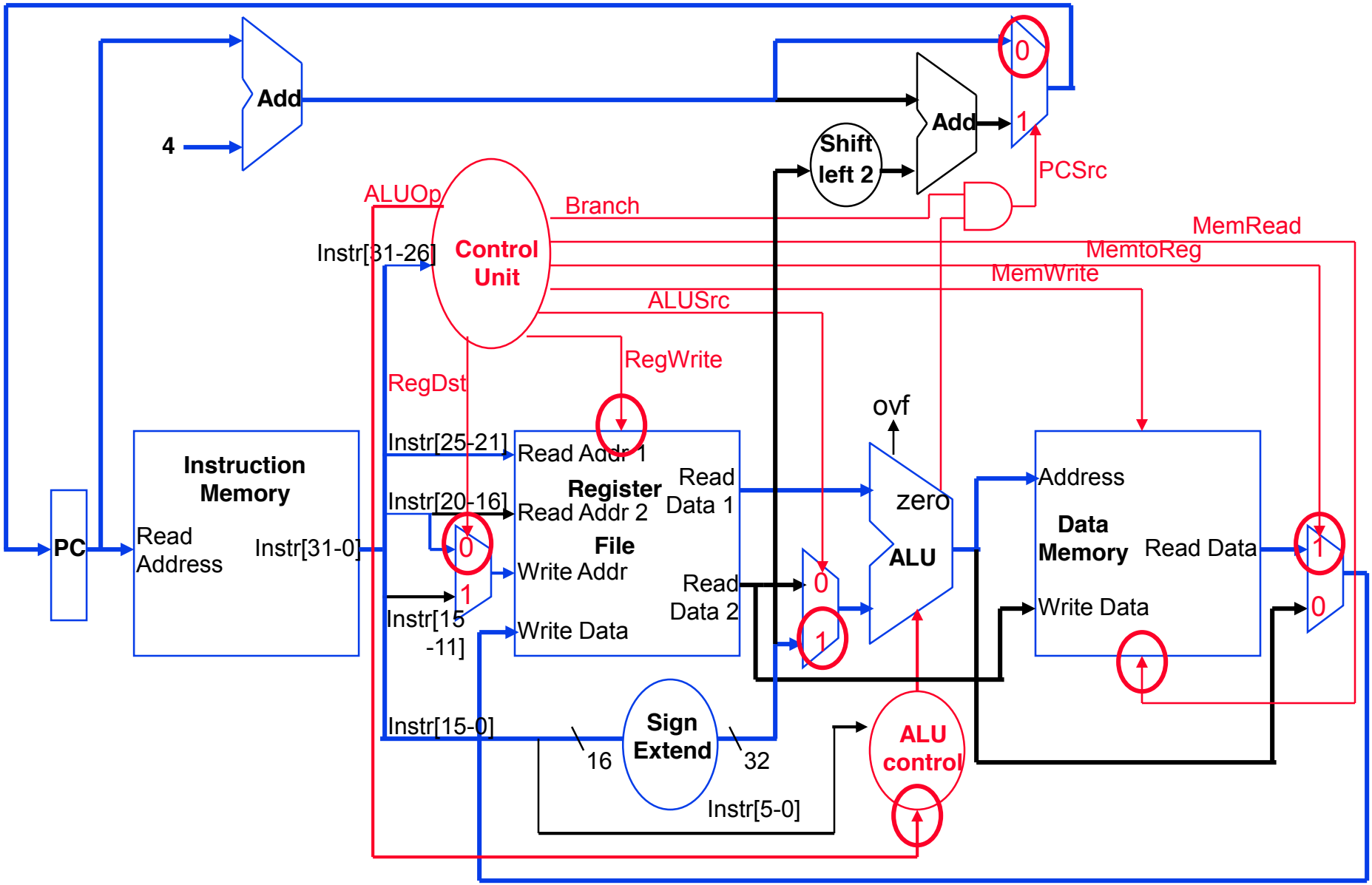
Single Cycle Datapath with Control Unit



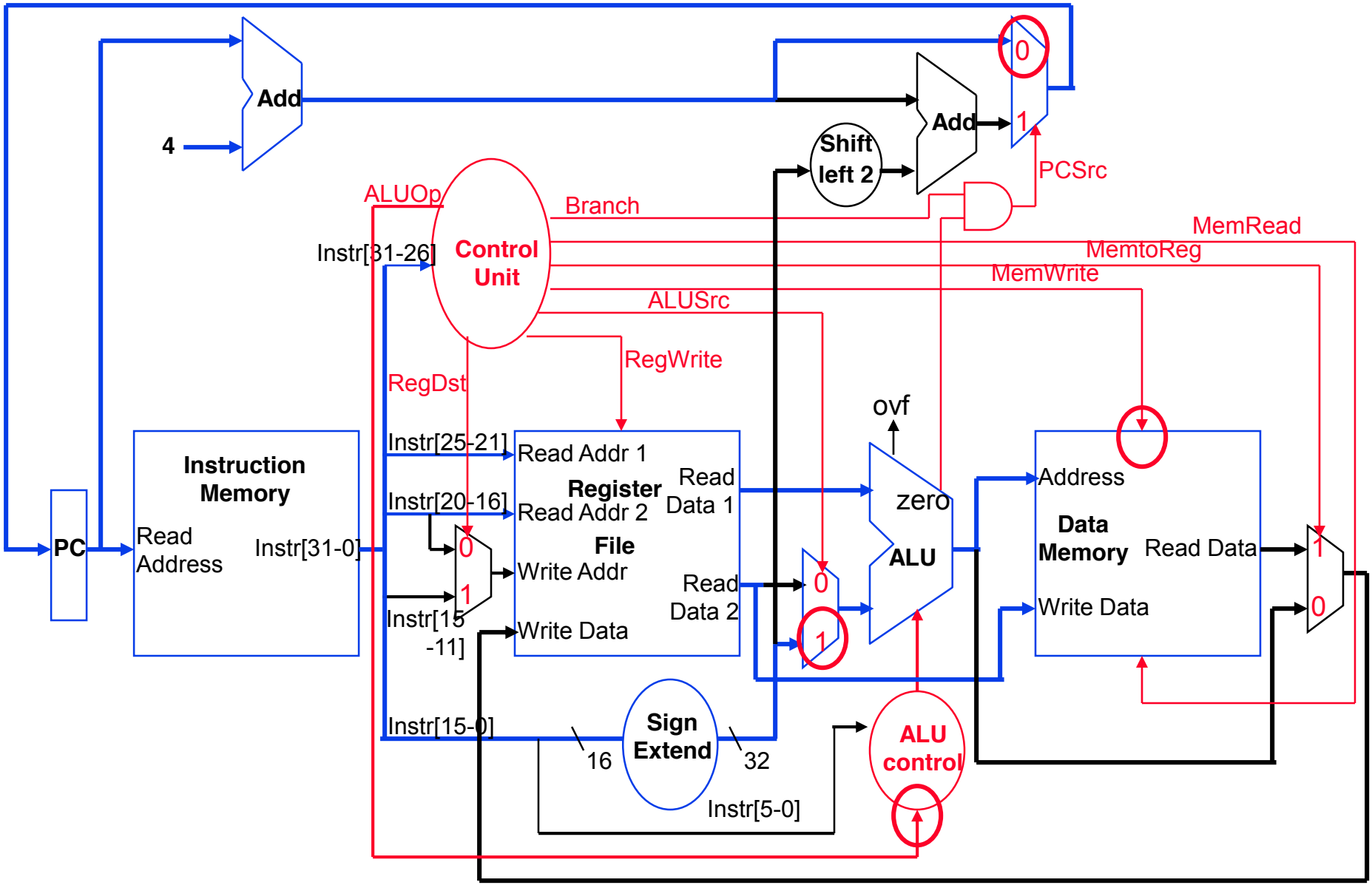
R-type Instruction Data/Control Flow



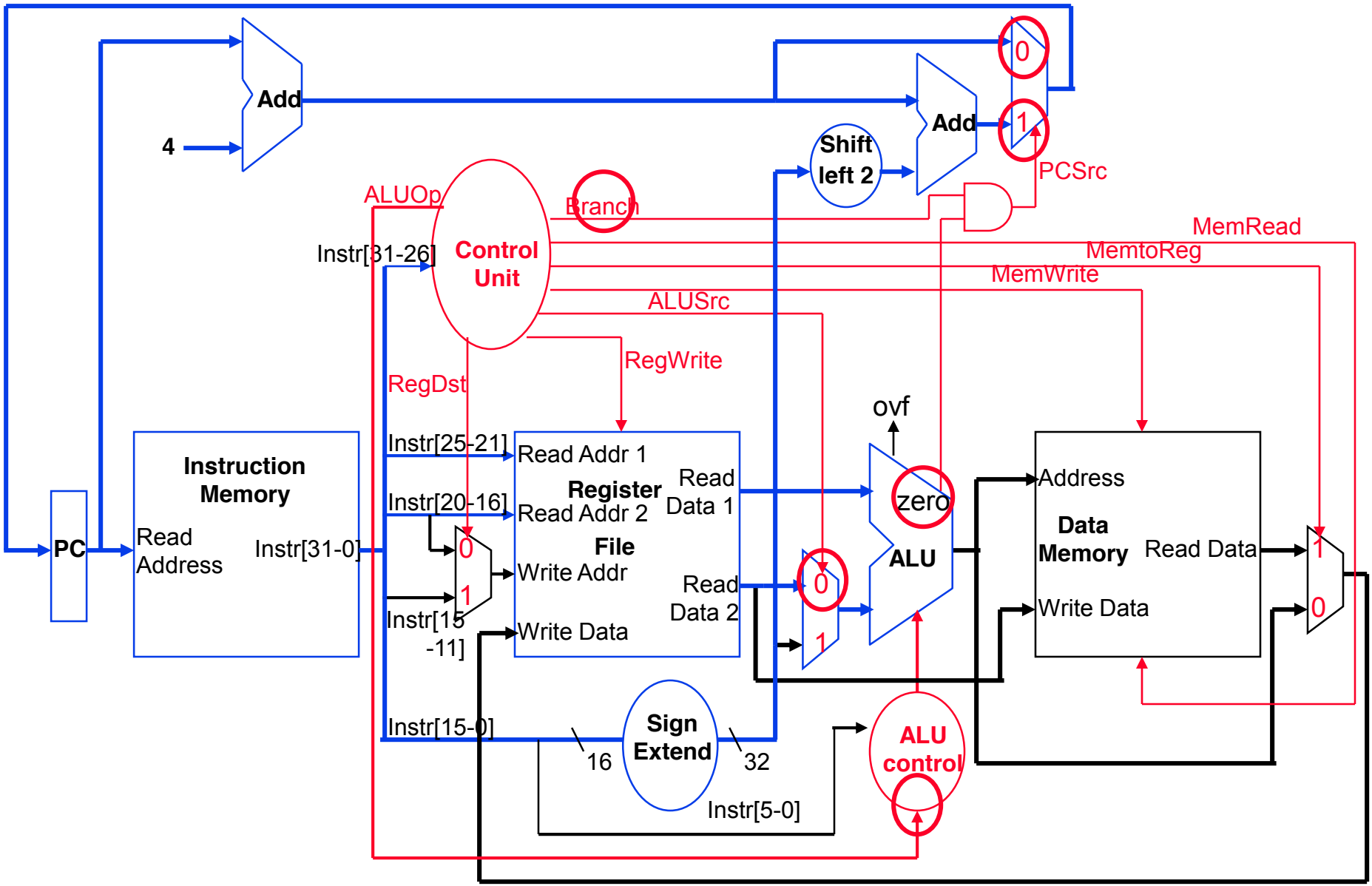
Load Word Instruction Data/Control Flow



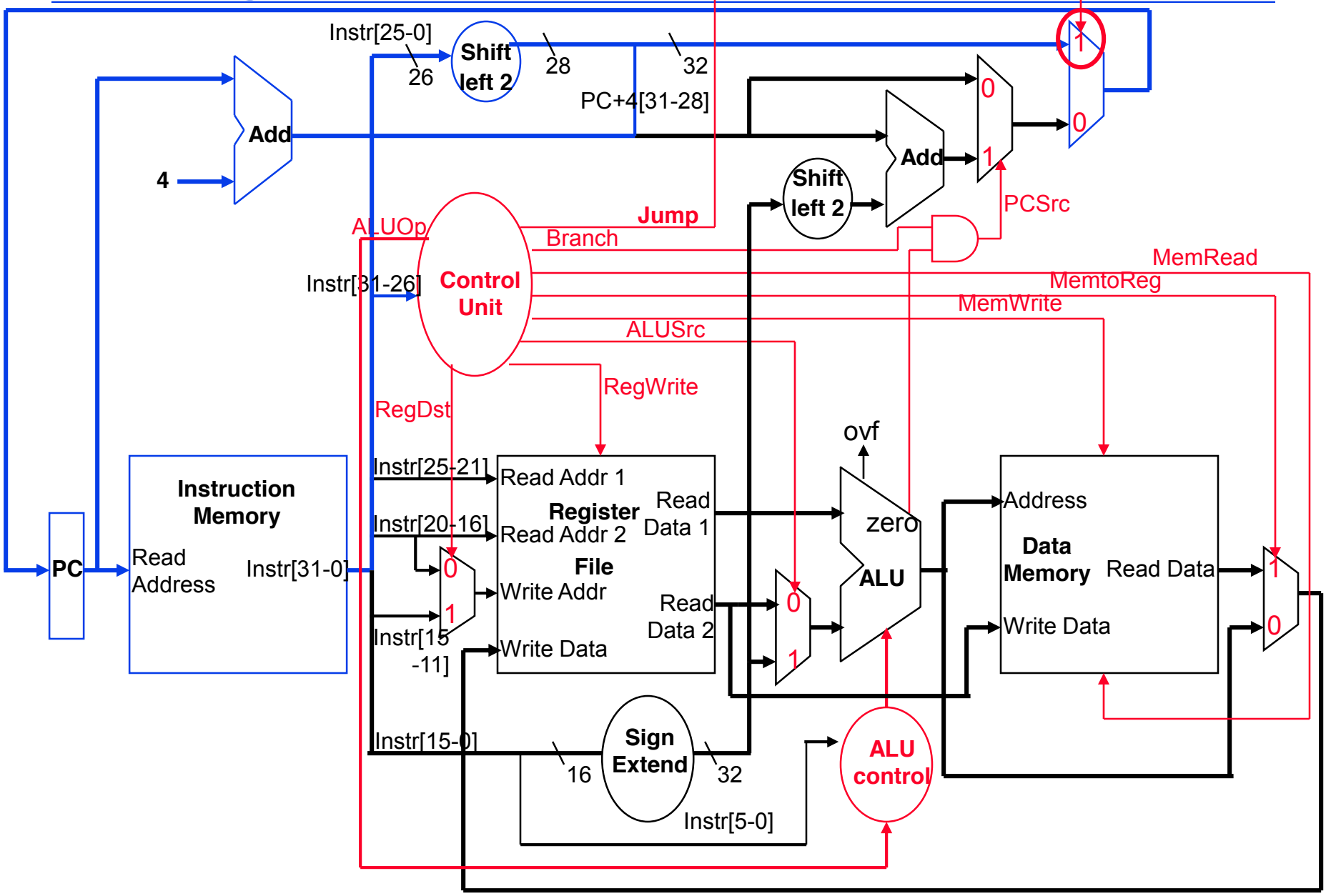
Store Word Instruction Data/Control Flow



Branch Instruction Data/Control Flow



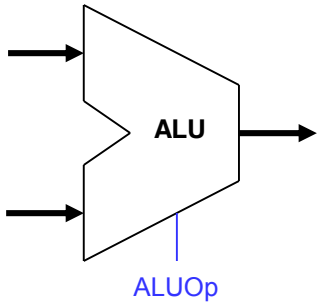
Adding the Jump Instruction



ALUOp

- Here's a simple ALU with five operations, selected by a 3-bit control signal ALUOp.

ALUOp	Function
000	and
001	or
010	add
110	subtract
111	slt



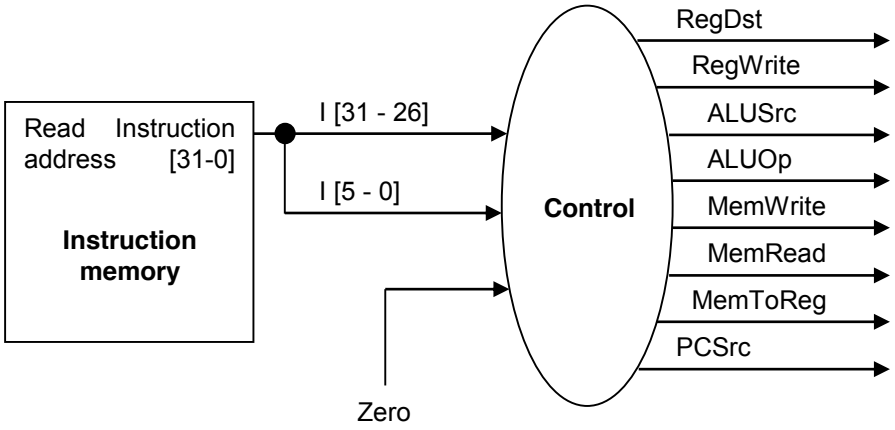
Control Signal Table

Operation	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemRead	MemToReg
add	1	1	0	010	0	0	0
sub	1	1	0	110	0	0	0
and	1	1	0	000	0	0	0
or	1	1	0	001	0	0	0
slt	1	1	0	111	0	0	0
lw	0	1	1	010	0	1	1
sw	X	0	1	010	1	0	X
beq	X	0	0	110	0	0	X

- sw and beq are the only instructions that do not write any registers.
- lw and sw are the only instructions that use the constant field. They also depend on the ALU to compute the effective memory address.
- ALUOp for R-type instructions depends on the instructions' func field.
- The PCSrc control signal (not listed) should be set if the instruction is beq *and* the ALU's Zero output is true.

Generating Control Signals

- The control unit needs 13 bits of inputs.
  - Six bits make up the instruction's opcode.
  - Six bits come from the instruction's func field.
  - It also needs the Zero output of the ALU.
- The control unit generates 10 bits of output, corresponding to the signals mentioned on the previous slide.
- You can build the actual circuit by using big K-maps, big Boolean algebra, or big circuit design programs.
- The textbook presents a slightly different control unit.



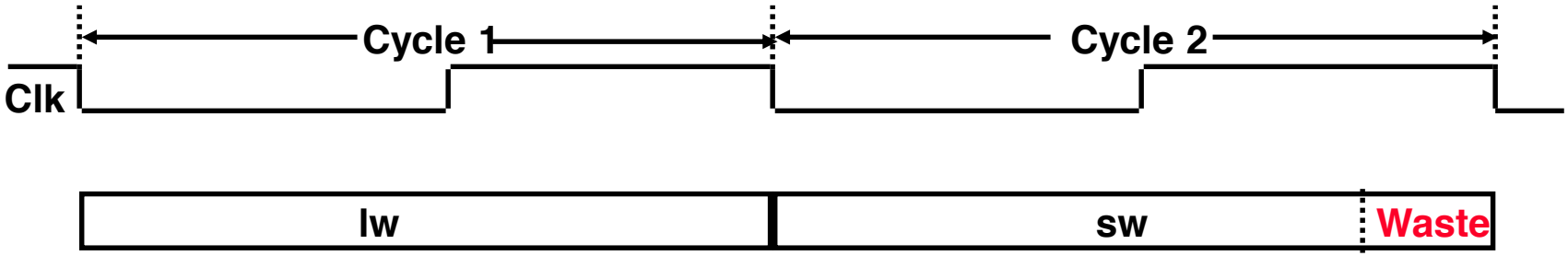
Instruction Critical Paths

- 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	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 (FP) 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

- **Single cycle datapath is simple and easy to understand**

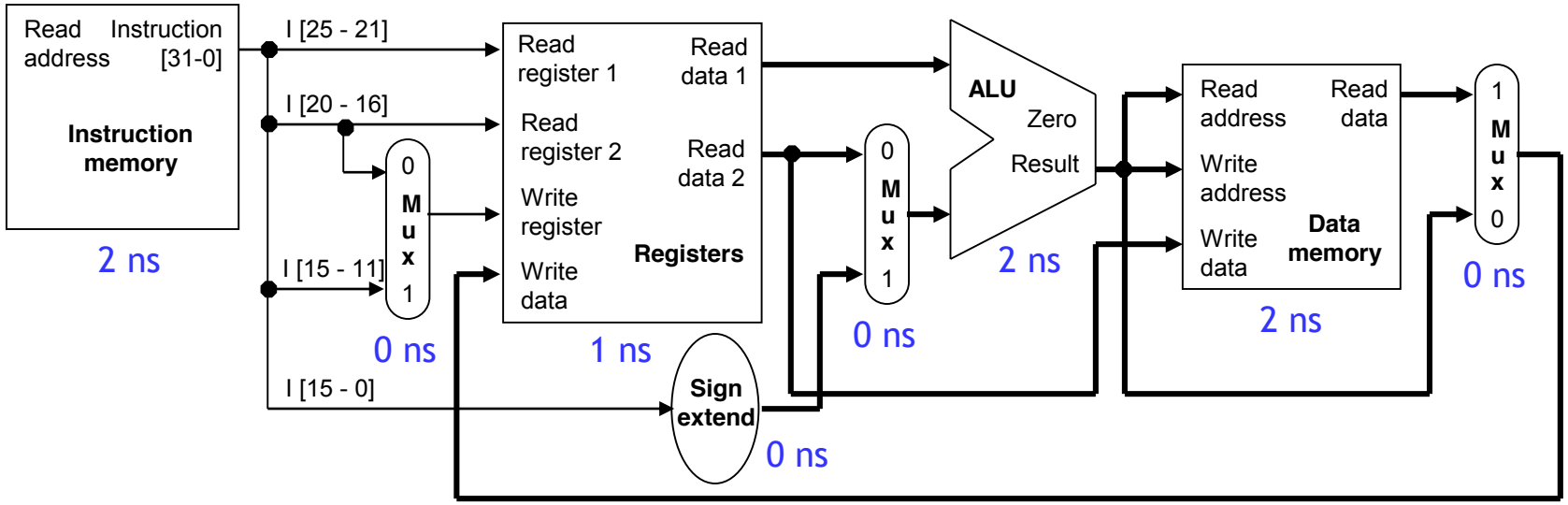
Summary

- A datapath contains all the functional units and connections necessary to implement an instruction set architecture.
  - For our single-cycle implementation, we use two separate memories, an ALU, some extra adders, and lots of multiplexers.
  - MIPS is a 32-bit machine, so most of the buses are 32-bits wide.
- The control unit tells the datapath what to do, based on the instruction that's currently being executed.
  - Our processor has ten control signals that regulate the datapath.
  - The control signals can be generated by a combinational circuit with the instruction's 32-bit binary encoding as input.
- Later, we'll see how this simple architecture can be made faster

CCT Problem

- If all instructions must complete within one clock cycle, then the cycle time has to be large enough to accommodate the *slowest* instruction.
- For example, lw $t0, –4($sp) needs 8ns, assuming the delays shown here.

reading the instruction memory	2ns	8ns
reading the base register $sp	1ns	
computing memory address $sp-4	2ns	
reading the data memory	2ns	
storing data back to $t0	1ns	

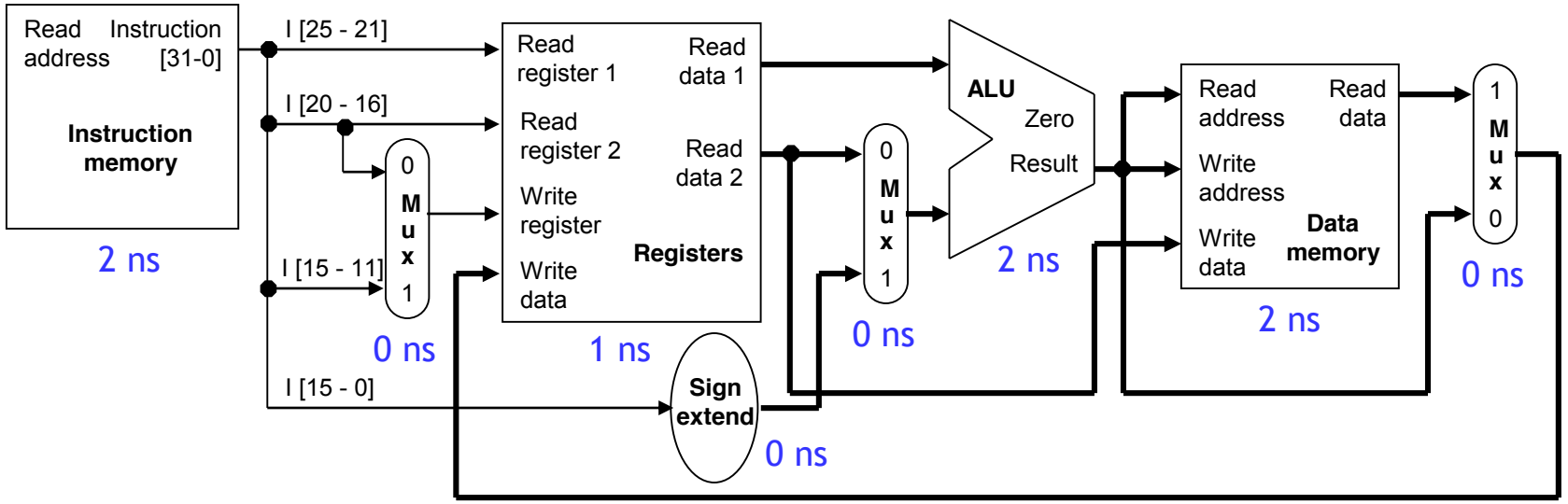


CCT Problem

- If we make the cycle time 8ns then *every* instruction will take 8ns, even if they don't need that much time.
- For example, the instruction add $s4, $t1, $t2 really needs just 6ns.

reading the instruction memory	2ns
reading registers $t1 and $t2	1ns
computing $t1 + $t2	2ns
storing the result into $s0	1ns

} 6ns



Is this really bad?

- With these same component delays, a sw instruction would need 7ns, and beq would need just 5ns.
- Let's consider the **gcc** instruction mix:

Instruction	Frequency
Arithmetic	48%
Loads	22%
Stores	11%
Branches	19%

- With a single-cycle datapath, each instruction would require 8ns.
- But if we could execute instructions as fast as possible, the average time per instruction for gcc would be:

$$(48\% \times 6ns) + (22\% \times 8ns) + (11\% \times 7ns) + (19\% \times 5ns) = 6.36ns$$

- The single-cycle datapath is about 1.26 times slower!

A multistage approach to instruction execution

- We've informally described instructions as executing in several steps.

1. Instruction fetch and PC increment.
2. Reading sources from the register file.
3. Performing an ALU computation.
4. Reading or writing (data) memory.
5. Storing data back to the register file.

- What if we made these stages explicit in the hardware design?

Performance benefits

- Each instruction can execute only the stages that are necessary.
  - Arithmetic
  - Load
  - Store
  - Branches
- This would mean that instructions complete as soon as possible, instead of being limited by the slowest instruction.

Proposed execution stages

1. Instruction fetch and PC increment
2. Reading sources from the register file
3. Performing an ALU computation
4. Reading or writing (data) memory
5. Storing data back to the register file

Performance?

- Things are simpler if we assume that each "stage" takes one clock cycle.
  - This means instructions will require multiple clock cycles to execute.
  - But, since a single stage is fairly simple, the cycle time can be very low.

Story so far…

- Single cycle
  - CPI = 1
  - Long CCT

CCT=Clock Cycle Time

Tradeoff

- Multi cycle
  - CPI > 1 (depends on instruction type)
  - Short CCT

- Common point:
  - At any given cycle, one instruction is executing

- Next step:
  - Try to execute multiple instructions in the same cycle

How Can We Make It Faster?

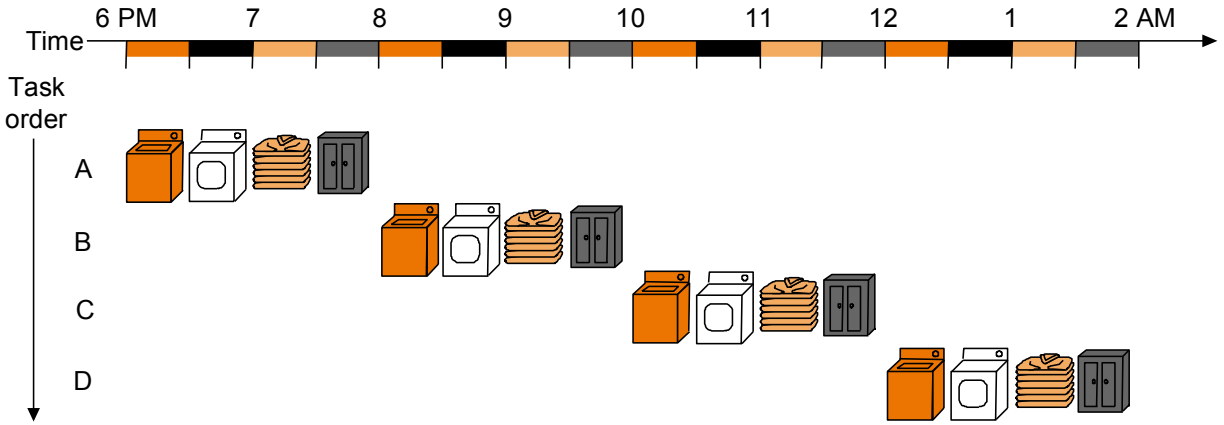
- Start fetching and executing the next instruction before the current one has completed
  - **Pipelining** – (all?) modern processors are pipelined for performance
  - Remember *the* performance equation:

    CPU time = CPI * CC * IC

- Under *ideal* conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages
  - A five stage pipeline is nearly five times faster because the CC is nearly five times faster

- Fetch (and execute) more than one instruction at a time
  - Superscalar processing – stay tuned

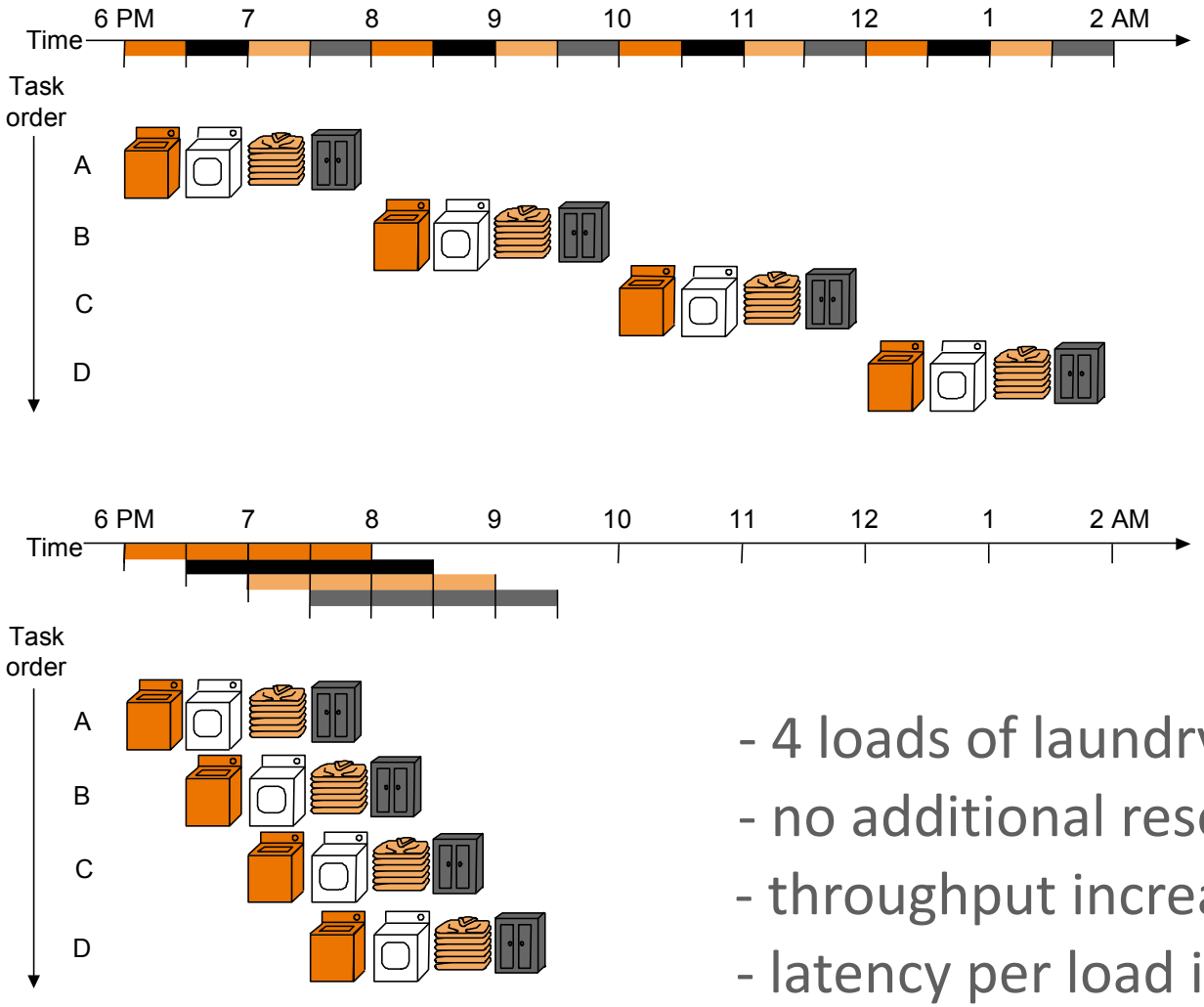
The Laundry Analogy



- "place one dirty load of clothes in the washer"
- "when the washer is finished, place the wet load in the dryer"
- "when the dryer is finished, take out the dry load and fold"
- "when folding is finished, ask your roommate (??) to put the clothes away"

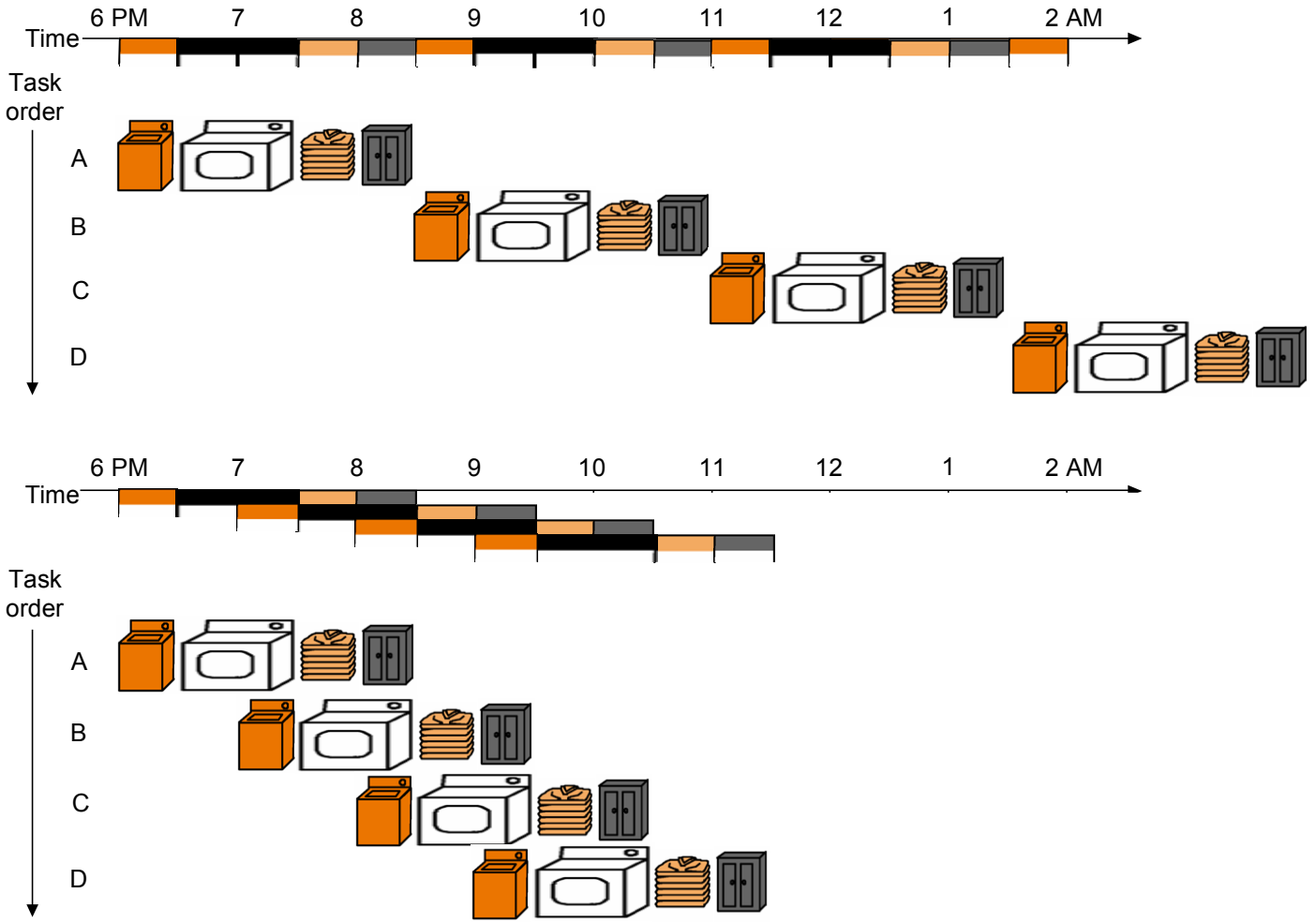
- steps to do a load are sequentially dependent
- no dependence between different loads
- different steps do not share resources

Pipelining Multiple Loads of Laundry



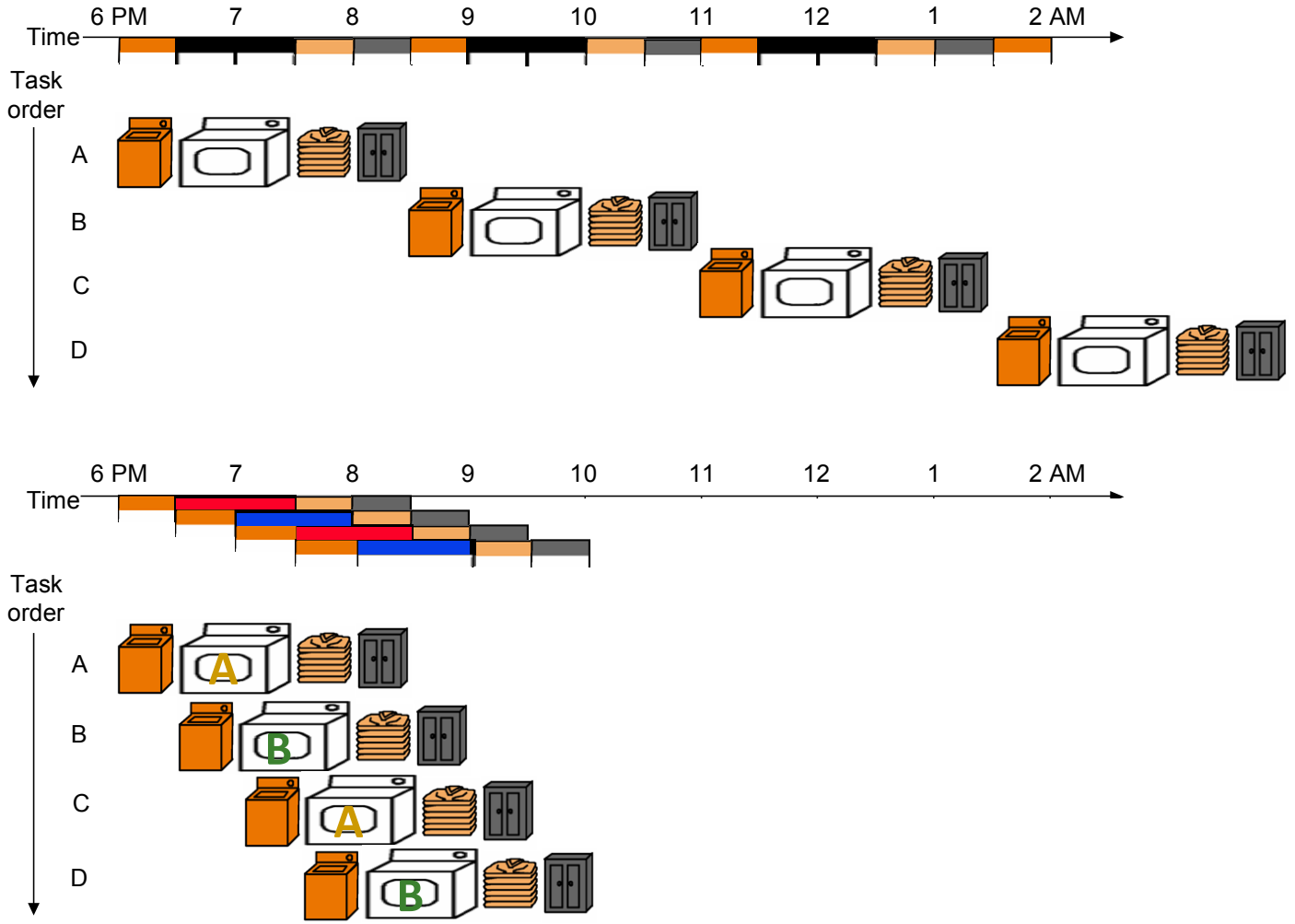
- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

Pipelining Multiple Loads of Laundry



the slowest step decides throughput

Pipelining Multiple Loads of Laundry

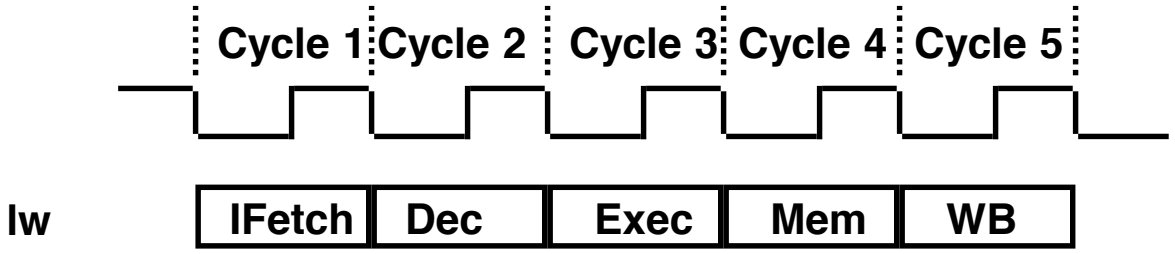


Throughput restored (2 loads per hour) using 2 dryers

Pipelining: Basic Idea

- Analogy:
  - Laundry, assembly line processing
- Idea:
  - Divide the instruction processing cycle into distinct “stages” of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a different instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput (1/CPI)

The Five Stages of Load Instruction



IFetch: Instruction Fetch and Update PC

Dec: Registers Fetch and Instruction Decode

Exec: Execute R-type; calculate memory address

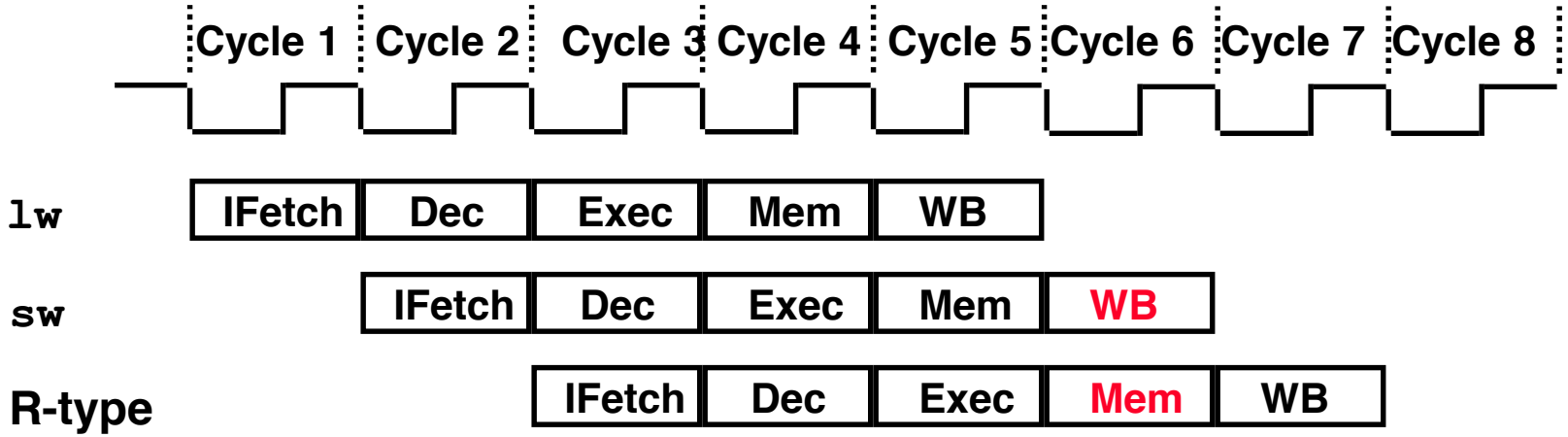
Mem: Read/write the data from/to the Data Memory

WB: Write the result data into the register file

- Single cycle – each stage is used once in each cycle
  - One active instruction per cycle (a looonnng cycle)
- Pipelined – each stage is used in each cycle
  - Multiple active instructions per cycle (a short cycle)

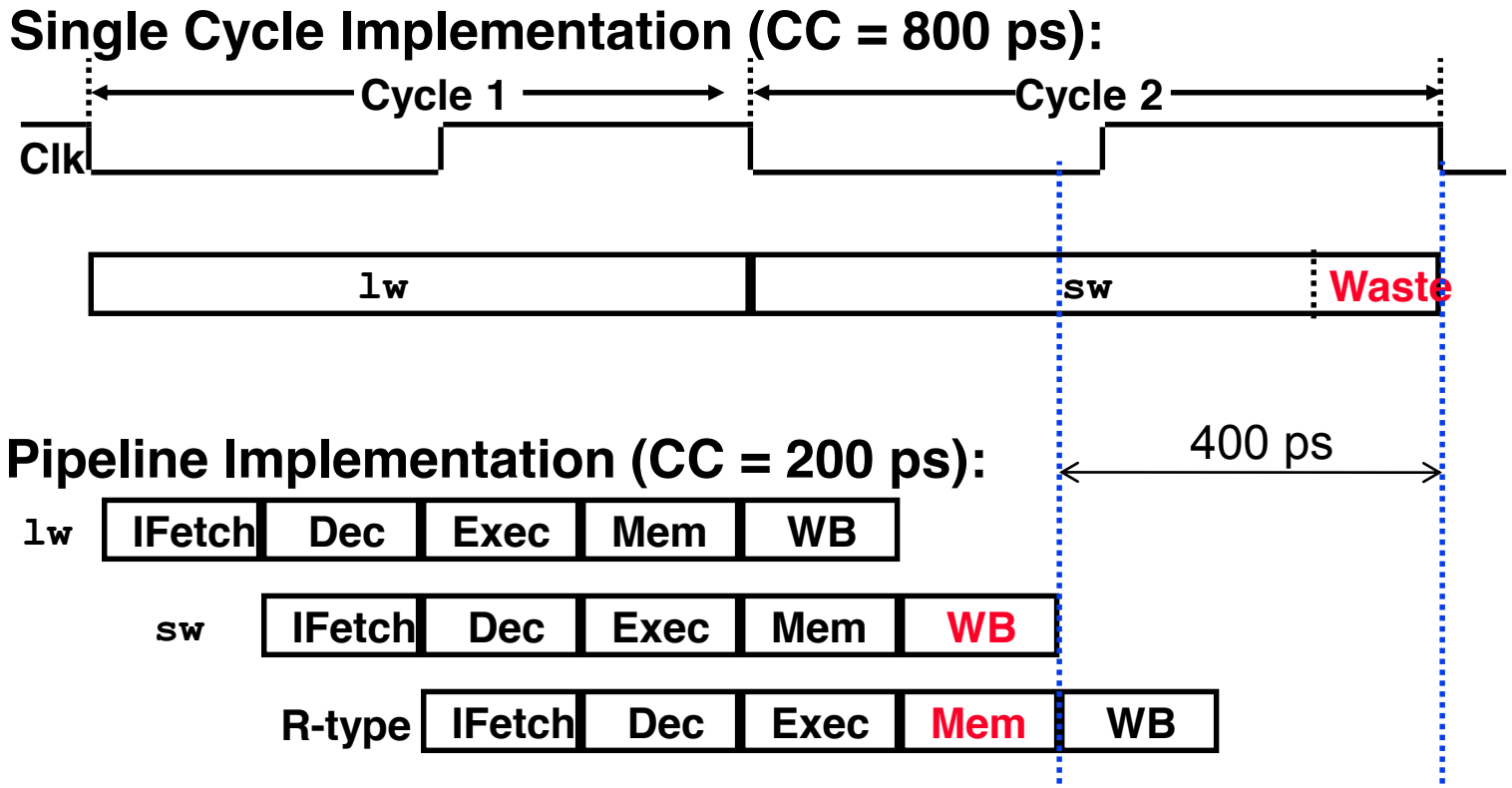
A Pipelined MIPS Processor

- Start the next instruction before the current one has completed
  - 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 and may increase slightly



- clock cycle (pipeline stage time) is limited by the **slowest** stage
  - for some stages don't need the whole clock cycle (e.g., WB)
- for some instructions, some stages are wasted cycles (i.e., nothing is done during that cycle for that instruction)

Single Cycle versus Pipeline



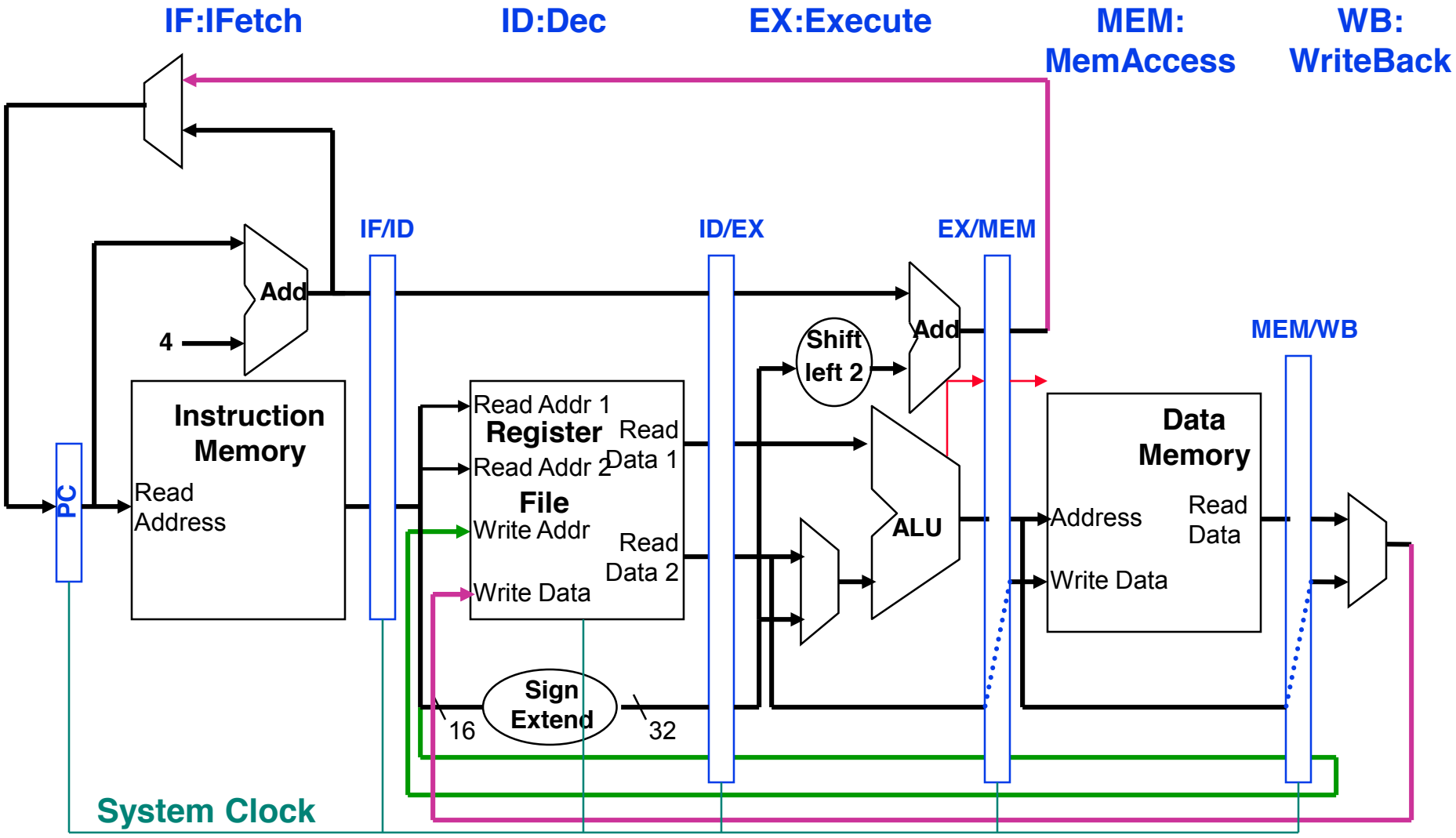
- How long does each take to complete 1,000,000 adds ?

Pipelining the MIPS ISA – What Makes It Easy?

- All instructions are the same length (32 bits)
  - can fetch in the 1st stage and decode in the 2nd stage
- Only a few instruction formats (three) with symmetry across formats
  - can begin reading register file in 2nd stage (before instruction is fully decoded) even if it turns out we don't need it
- 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

MIPS Pipeline Datapath Additions/Mods

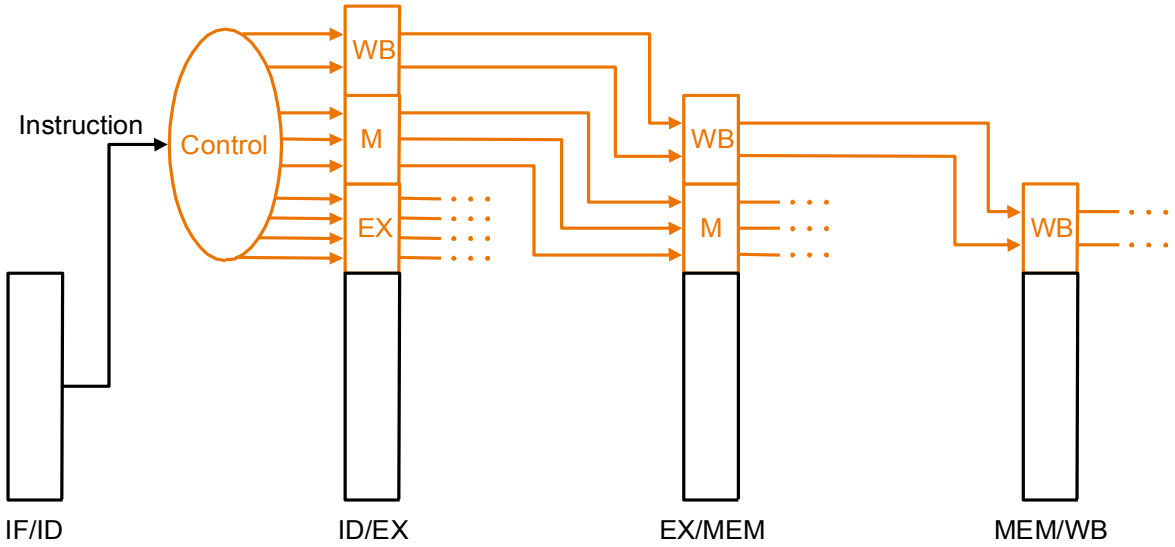
- State Registers between each pipeline stage to isolate them



Control Signals in a Pipeline

- For a given instruction
  - same control signals as single-cycle, but
  - control signals required at different cycles, depending on stage

⇒ decode once using the same logic as single-cycle and buffer control signals until consumed



⇒ or carry relevant "instruction word/field" down the pipeline and decode locally within each or in a previous stage

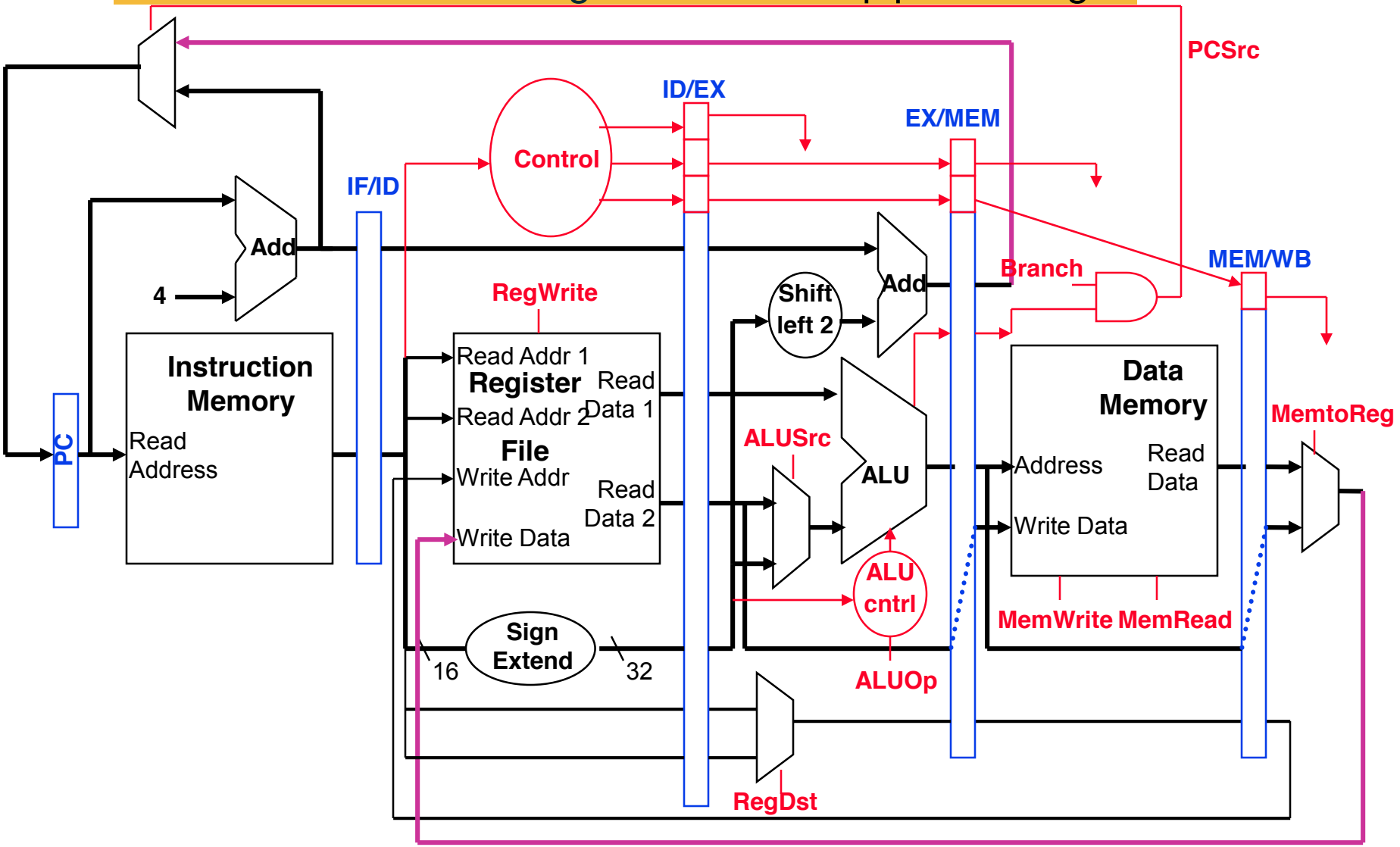
Pipeline Control

- IF Stage: read Instr Memory (always asserted) and write PC (on System Clock)
- ID Stage: decoding so no control signals ready yet

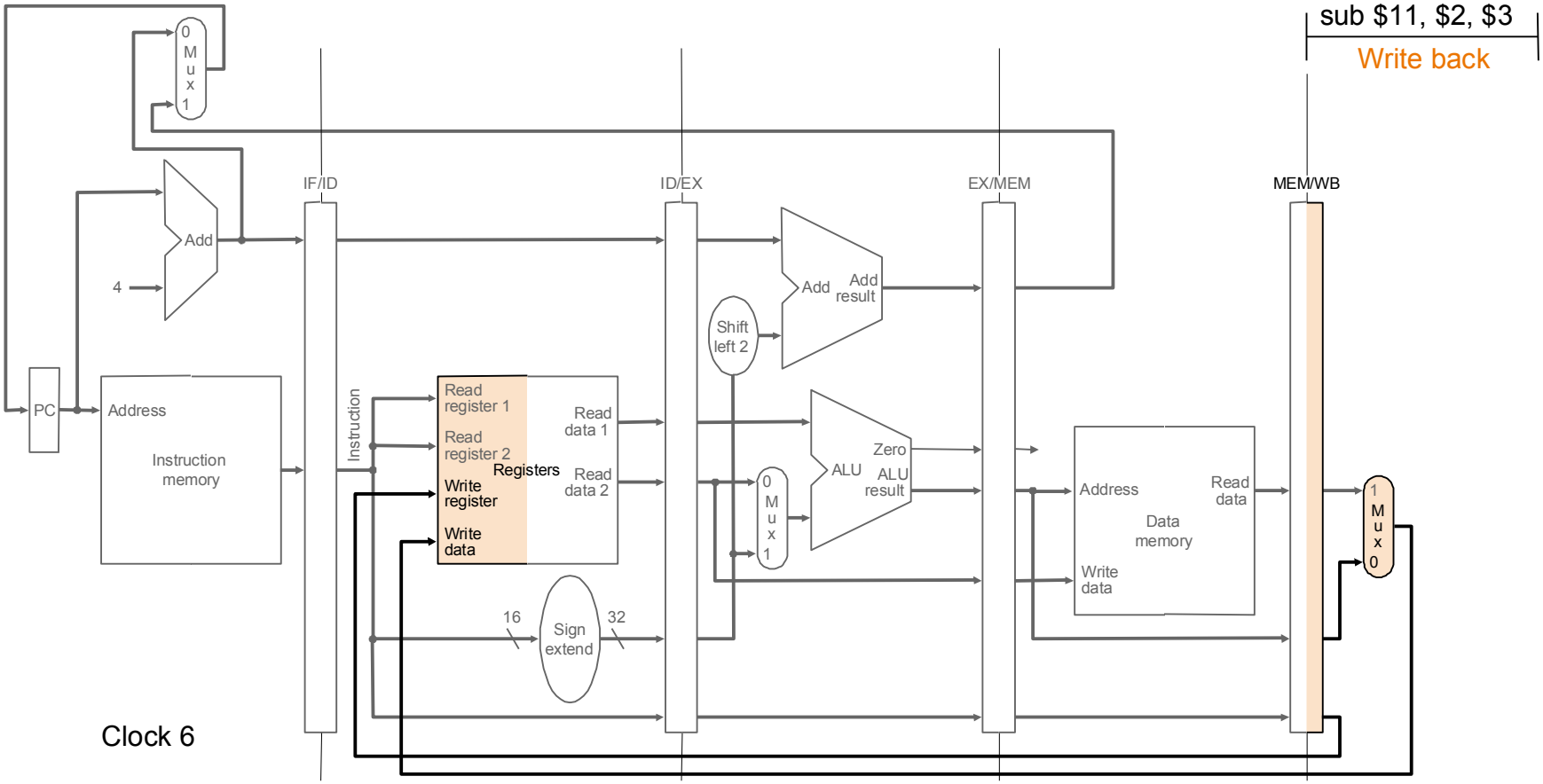
	EX Stage				MEM Stage			WB Stage	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Brch	Mem Read	Mem Write	Reg Write	Mem toReg
R	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

MIPS Pipeline Control Path Modifications

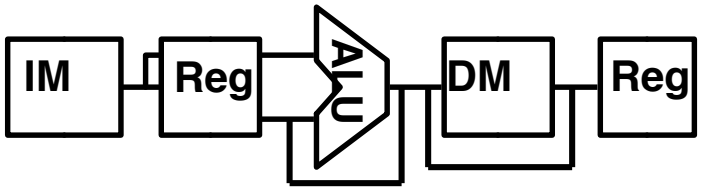
- All control signals can be determined during Decode
  - and held in the state registers between pipeline stages



Pipelined Operation Example

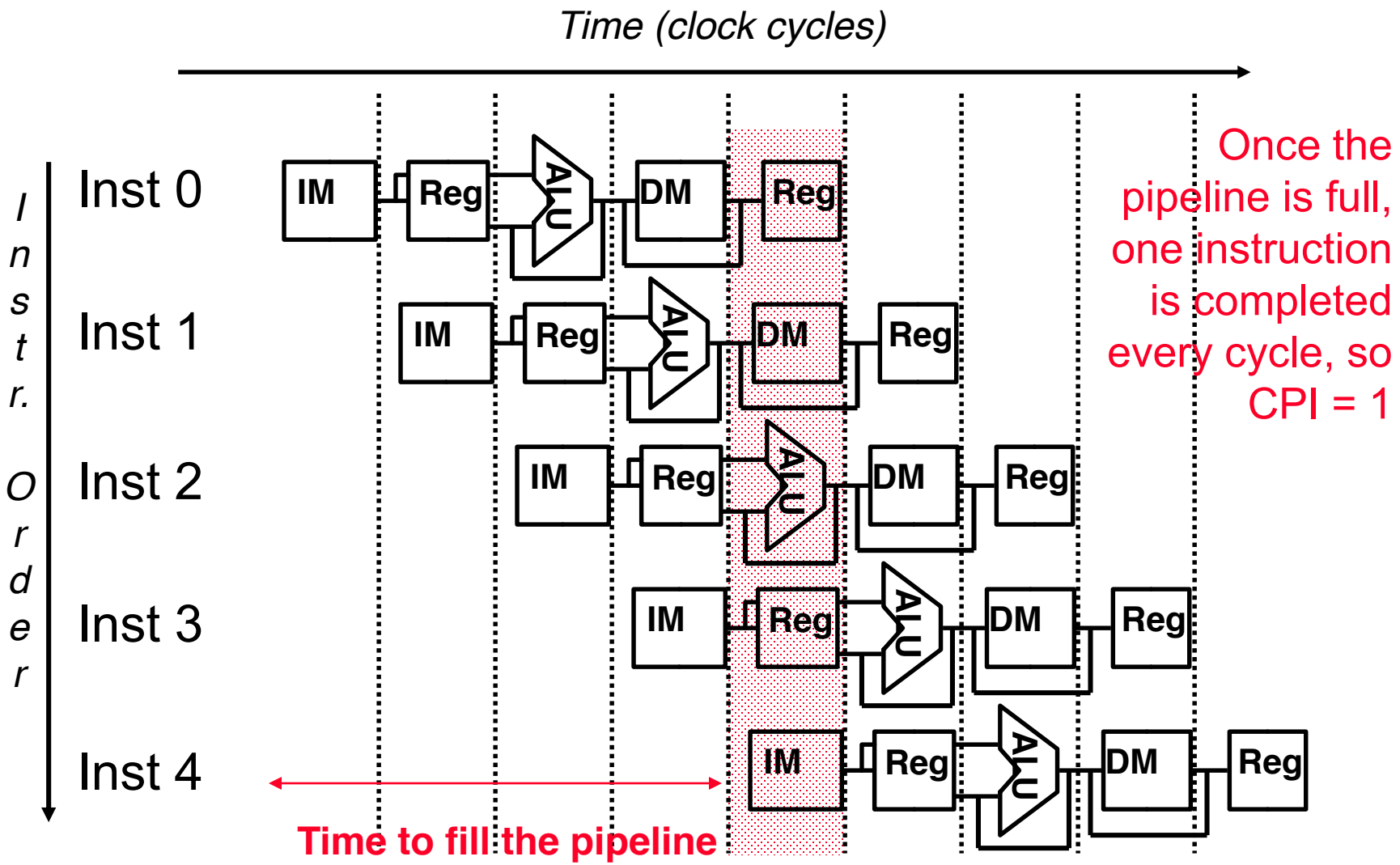


Graphically Representing MIPS Pipeline



- Can help with answering questions like:
  - How many cycles does it take to execute this code?
  - What is the ALU doing during cycle 4?
  - Is there a hazard, why does it occur, and how can it be fixed?

Why Pipeline? For Performance!

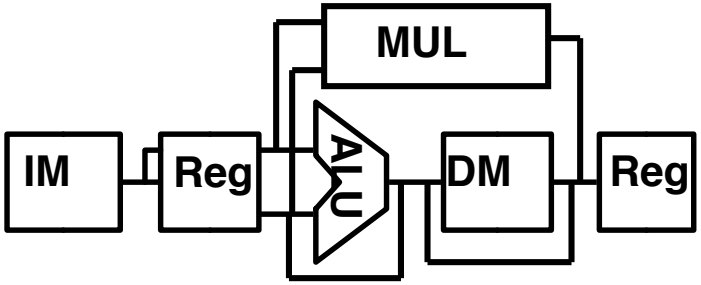


Pipelining the MIPS ISA – What Makes It Hard?

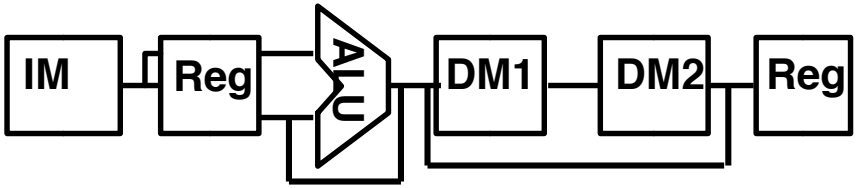
- Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt 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
- Dependencies backward in time cause hazards
- Can usually resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards

Other Pipeline Structures Are Possible

- What about the (slow) multiply operation?
  - Make the clock twice as slow or …
  - let it take two cycles (since it doesn't use the DM stage)



- What if the data memory access is twice as slow as the instruction memory?
  - make the clock twice as slow or …
  - let data memory access take two cycles (and keep the same clock rate)



Summary

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

Reading Assignment

- Read Sections 4.1, 4.2, 4.3 and 4.4 from HP