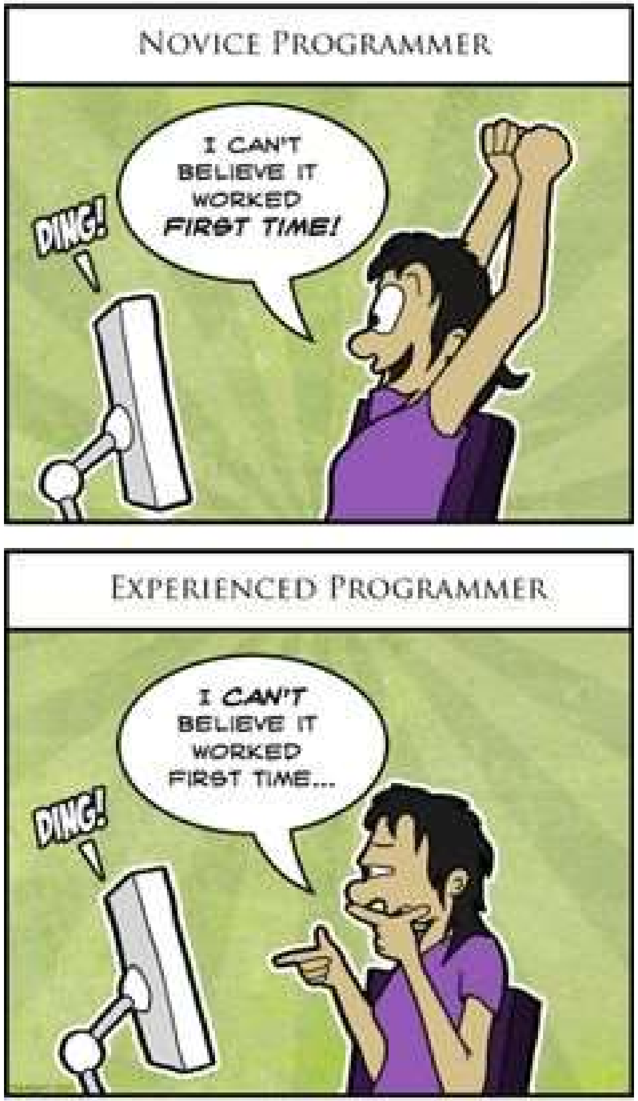

NOVICE PROGRAMMER
I CAN'T BELIEVE IT WORKED FIRST TIME!
DING!
EXPERIENCED PROGRAMMER
I CAN'T BELIEVE IT WORKED FIRST TIME...
DING!

https://i.pinimg.com/236x/9f/7c/42/9f7c42a12a59e2706b144d62d9b67f4e.jpg

Lecture 4:
Basic Processor Architecture

Some slides adapted from Mary Jane Irwin at Penn State University for *Computer Organization and Design*, Patterson & Hennessy, © 2005

Review: CPU Performance

- execution time = seconds / program

$$\frac{Instructions}{program} \times \frac{cycles}{Instruction} \times \frac{seconds}{cycle}$$

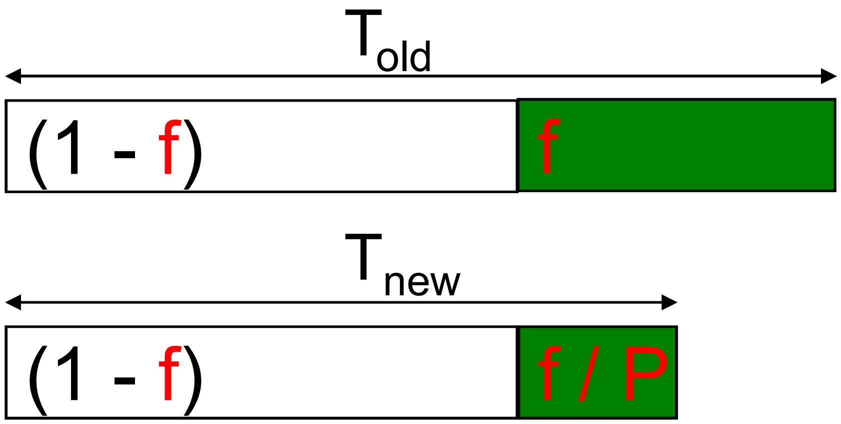
- programmer
- algorithms
- ISA
- compilers

- microarchitecture
- system architecture

- microarchitecture pipeline depth
- circuit design
- technology

Review: Amdahl's Law

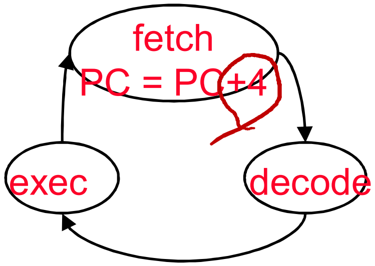
- law of diminishing returns
  - ✓ make the common case faster ✓
  - ✓ speedup = Perf_{new} / Perf_{old} = T_{old} / T_{new} = $$\frac{1}{(1-f)+\frac{f}{P}}$$
- example
  - ✓ supposing floating point instructions are improved to run 2X but comprise only 10% of actual instructions, what's the speedup?



Processor: Datapath & Control

- our implementation of a processor is simplified
  - memory-reference instructions: `lw, sw`
  - arithmetic/logical instructions: `add, sub, and, or, slt`
  - branch 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


fetch
PC = PC+4
decode
exec

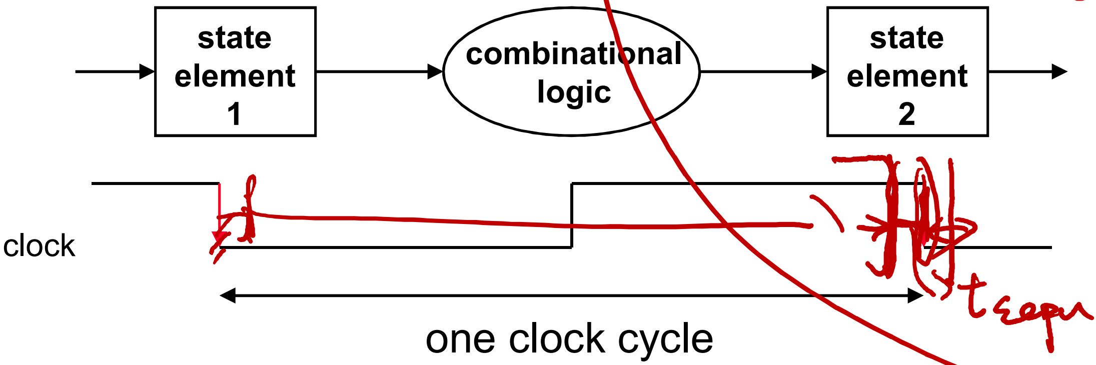
- all instructions (except `j`) use the ALU after reading the registers

Recap: Instruction Execution

- PC → instruction memory, fetch instruction
- register numbers → register file, read registers
- depending on instruction class
  - use ALU to calculate
    - arithmetic result
    - memory address for load/store
    - branch target address
  - access data memory for load/store
  - PC ← target address or PC + 4

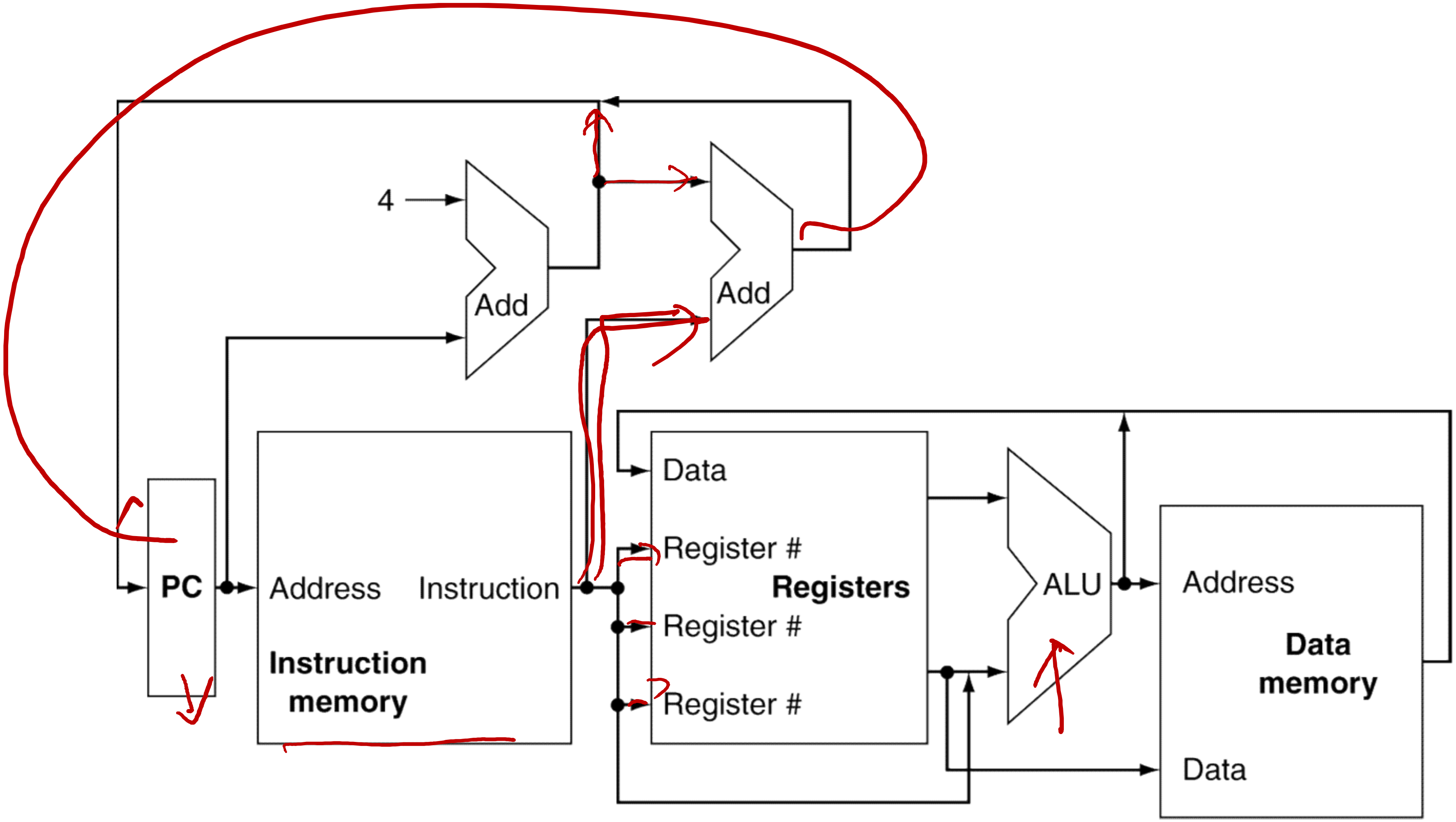
Clocking Methodologies

- clocking methodology defines when signals can be read and when they are written
  - edge-triggered methodology
- typical execution
  - read contents of state elements
  - send values through combinational logic
  - write results to one or more state elements

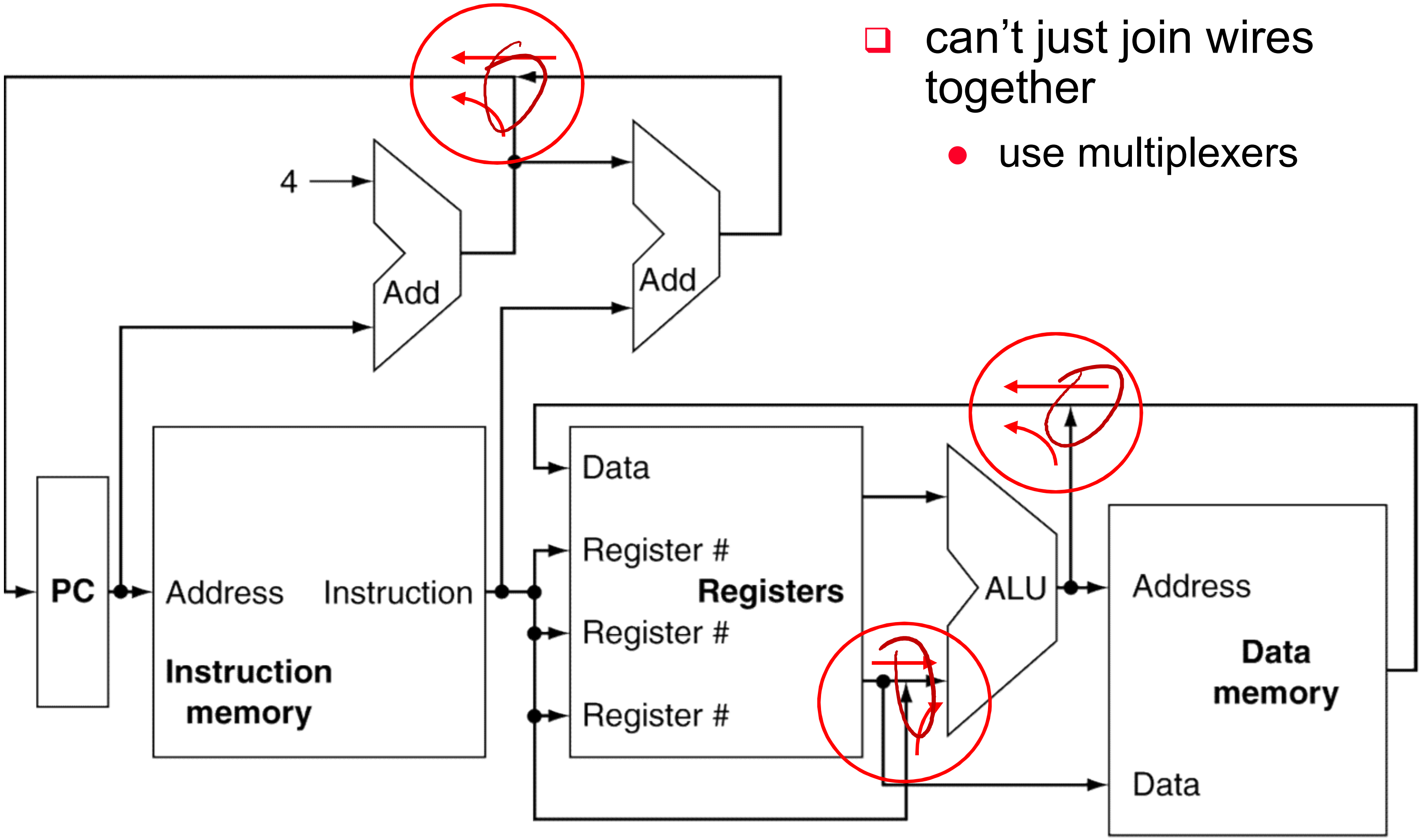


- assumes state elements are written on every clock cycle; if not, need explicit write control signal
  - write occurs only when both the write control is asserted and the clock edge occurs

Overview (Simplified example circuit)



Multiplexers



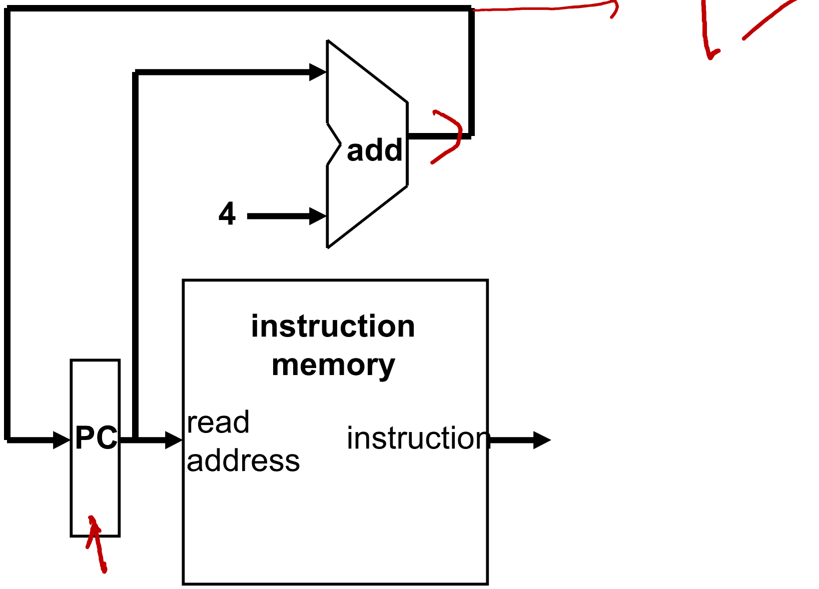
- can’t just join wires together
  - use multiplexers

Building a Datapath

- datapath
  - elements that process data and addresses in the CPU
    - registers, ALUs, mux’s, memories, …
- we will build a MIPS datapath incrementally
  - refining the overview design

Fetching Instructions

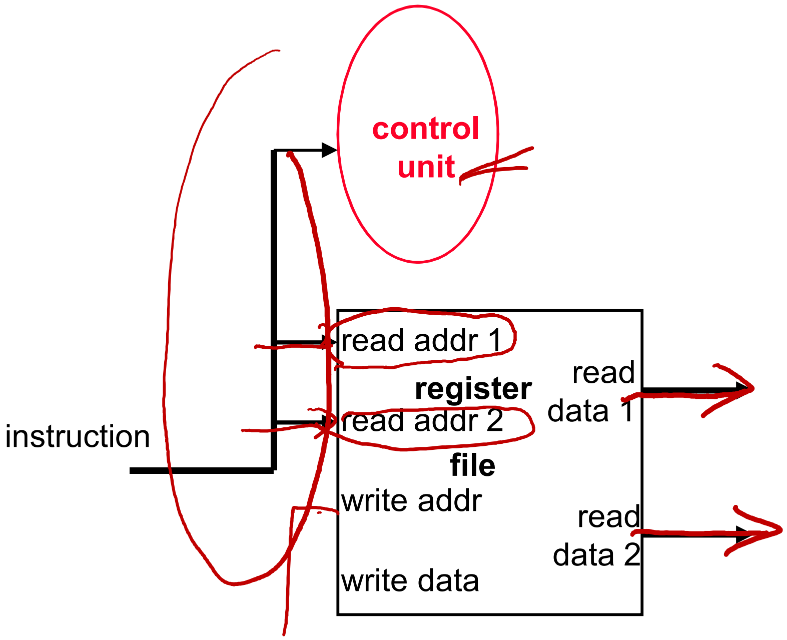
- fetching instructions involves
  - reading the instruction from instruction memory
  - updating PC to hold the address of the next instruction



- PC is updated every cycle, so it does not need an explicit write control signal
- instruction memory is read every cycle, so it doesn't need an explicit read control signal

Decoding Instructions

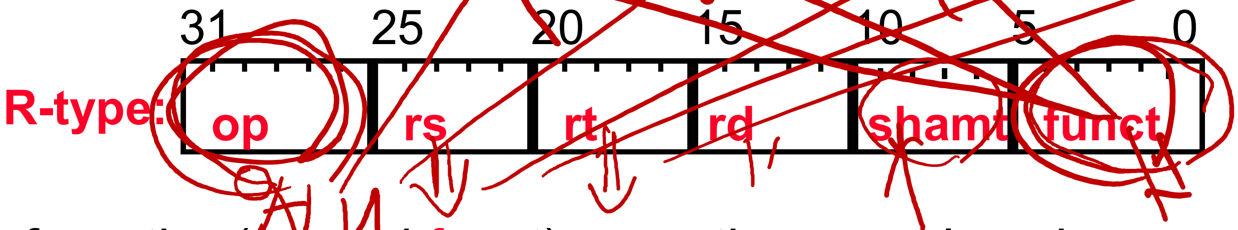
- decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit



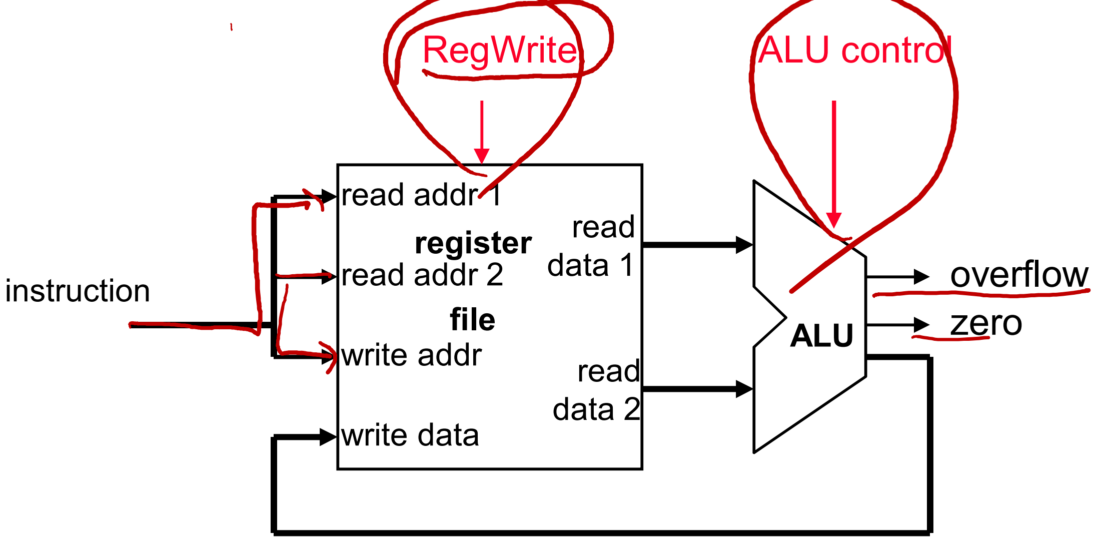
  - reading two values from register file
    - register file addresses are contained in the instruction

Executing R Format Operations

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



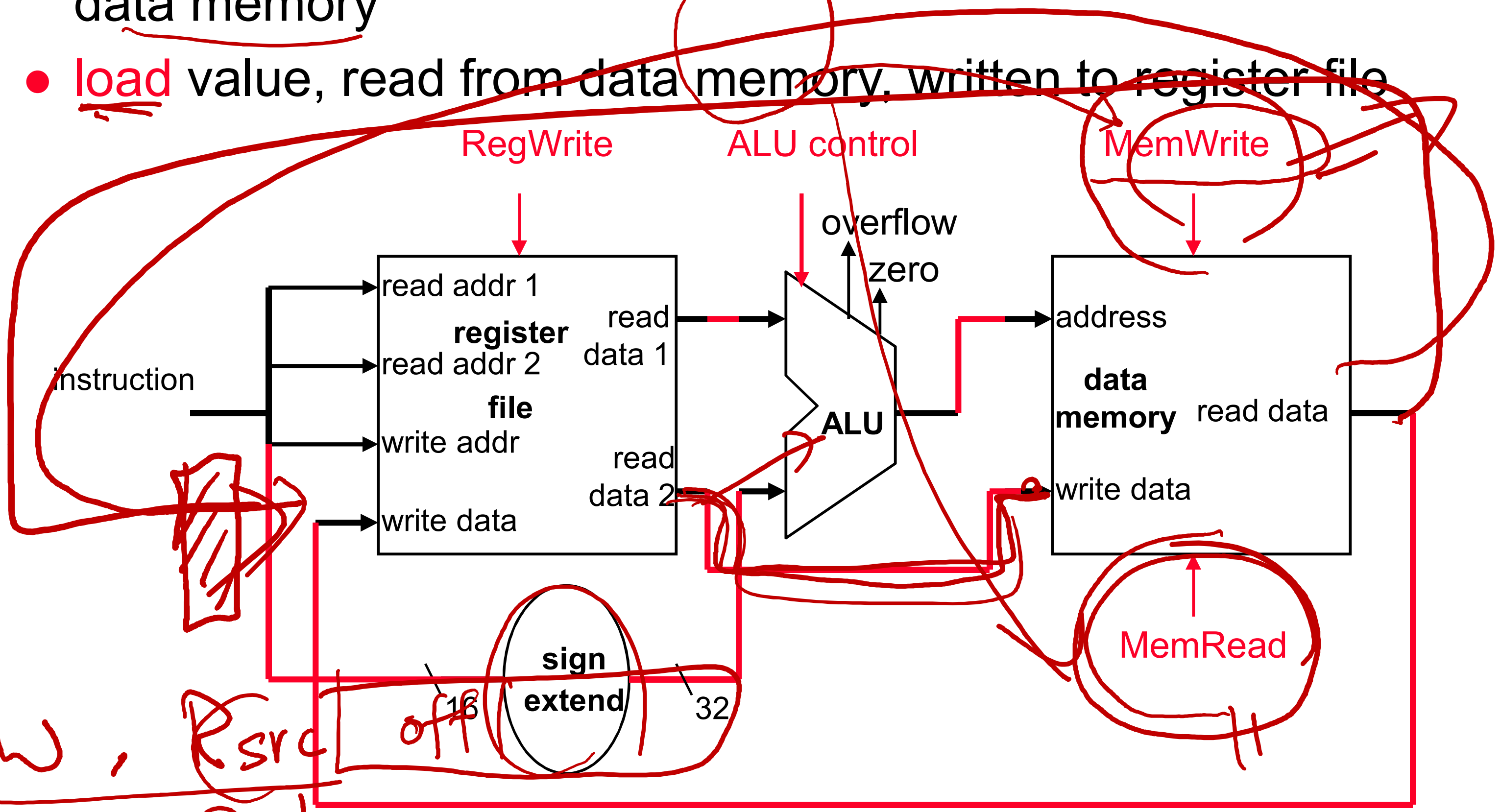
- perform the (op and funct) operation on values in rs and rt
- store the result back into register file (into location rd)



- register file is not written every cycle (e.g. **sw**), so we need an explicit write control signal for register file

Executing Load and Store Operations

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

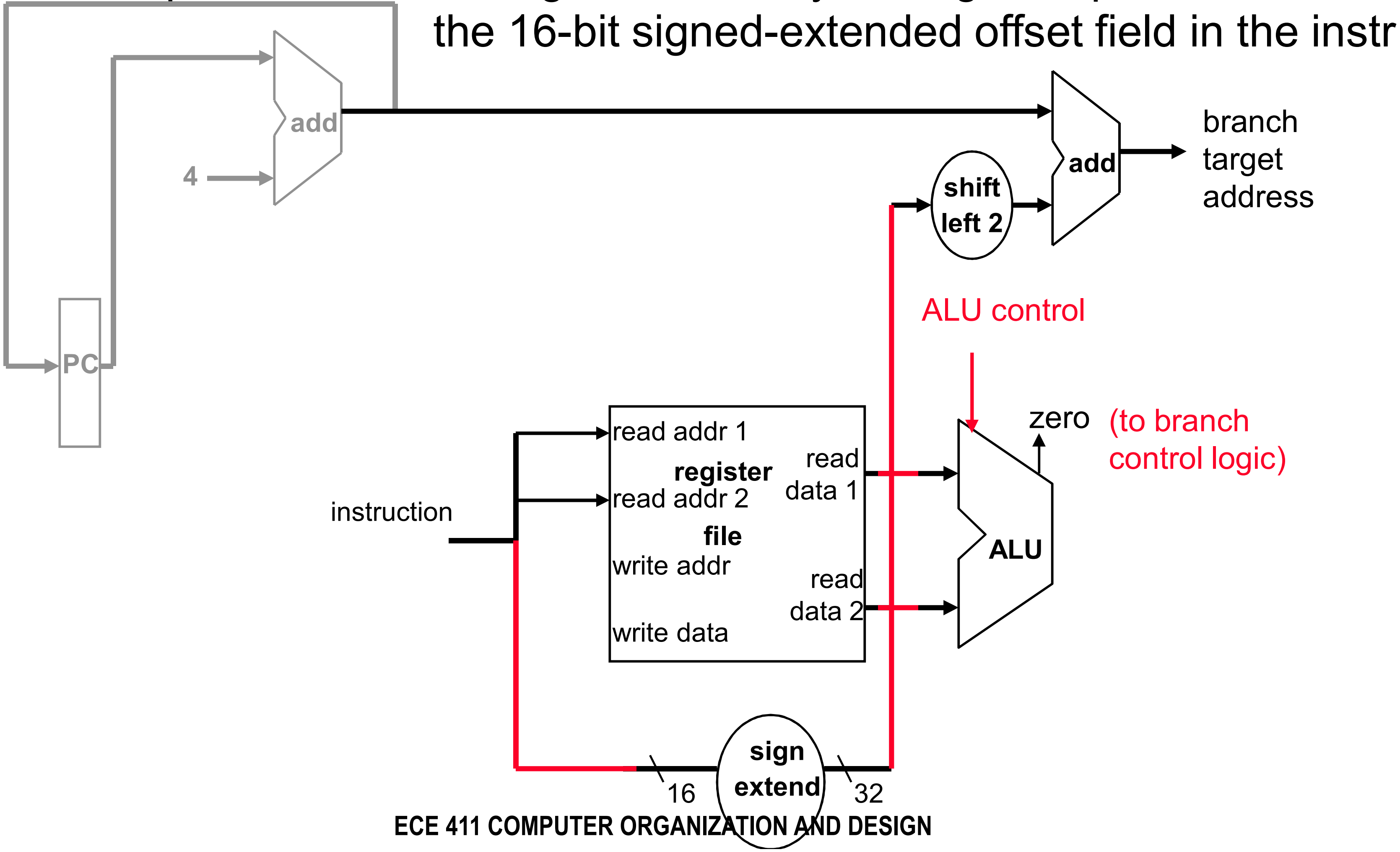


SW , Rsrc

LD . Rdst

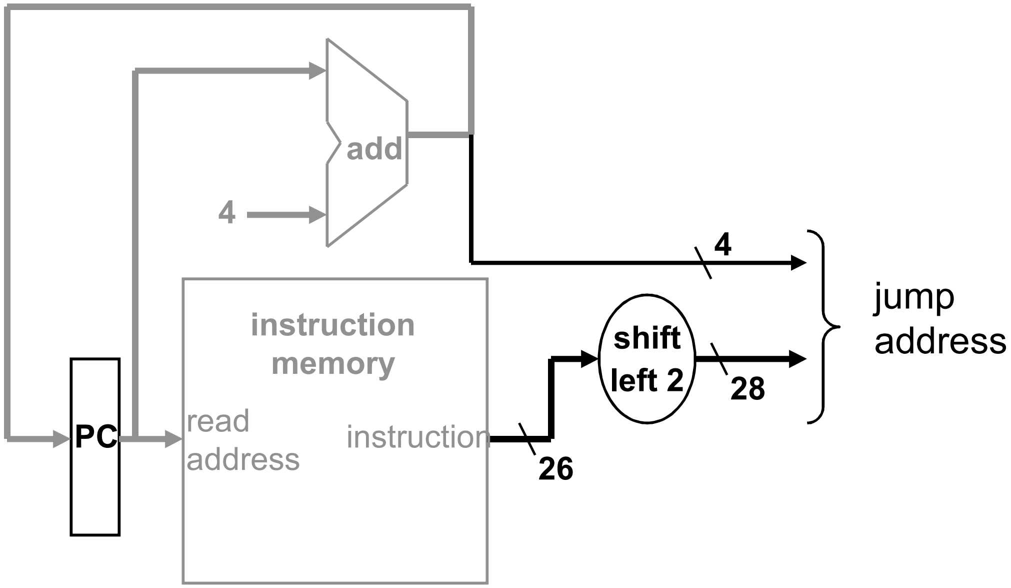
Executing Branch Operations

- branch operations involves
  - compare the operands read from register file 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



Executing Jump Operations

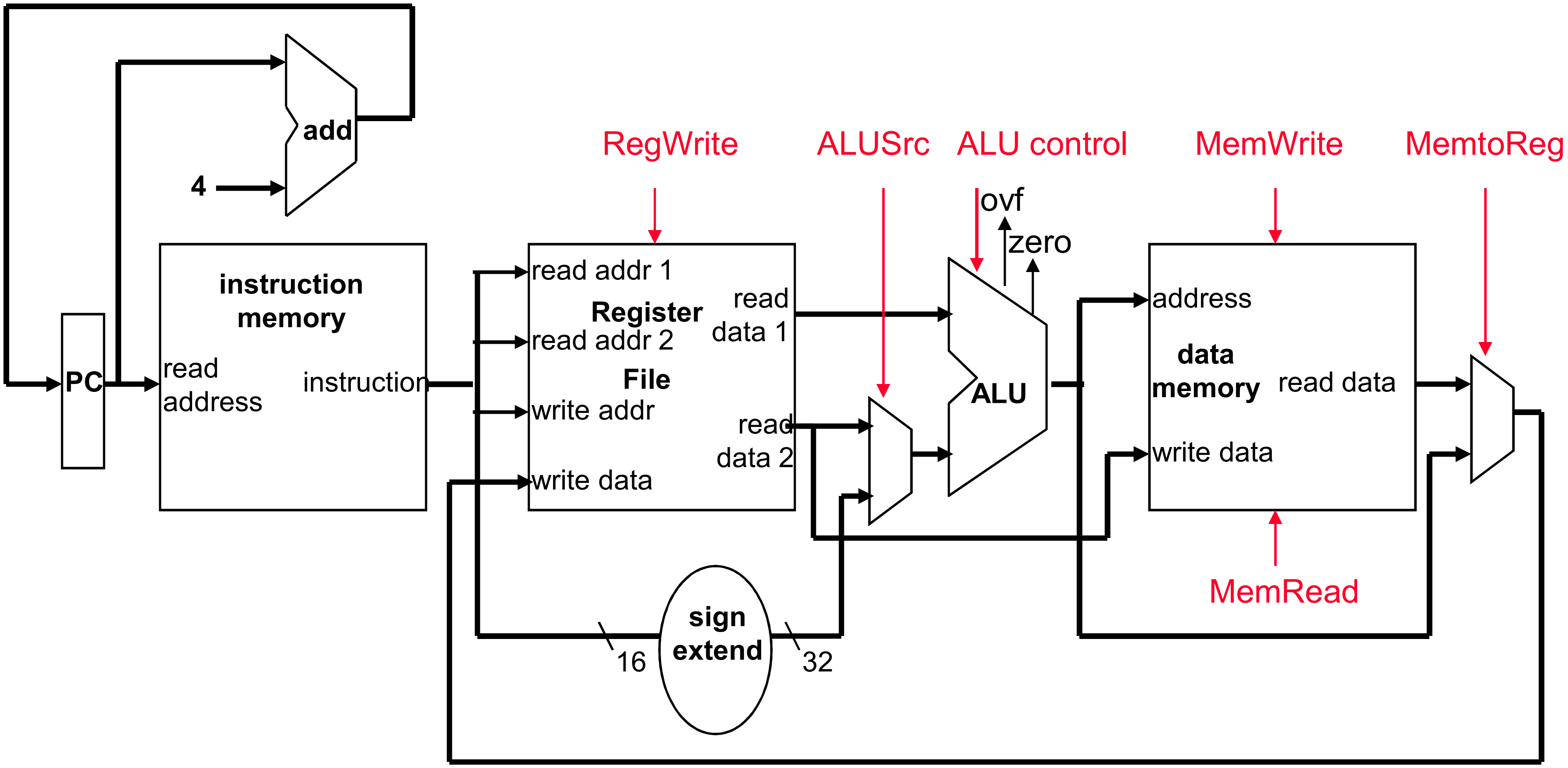
- jump operation involves
  - replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits



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 register file and data memory
- cycle time is determined by length of the longest path

Control signals for Fetch, Reg, and Memory

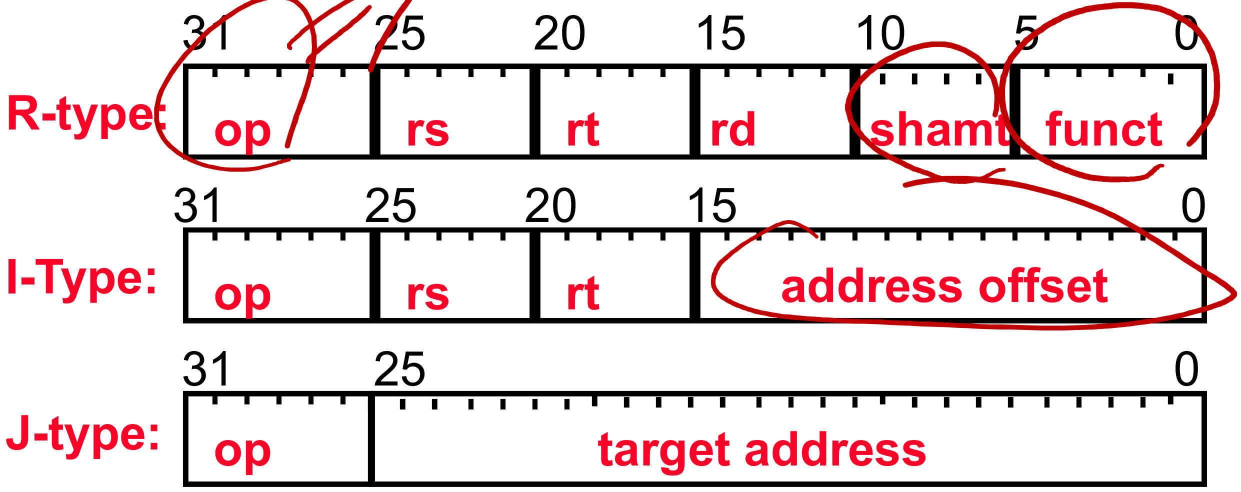


what determines the values needed on these control signals?

Adding the Control

skip now
need to see

- selecting the operations to perform (ALU, register file and memory read/write)
- controlling the flow of data (multiplexor inputs)



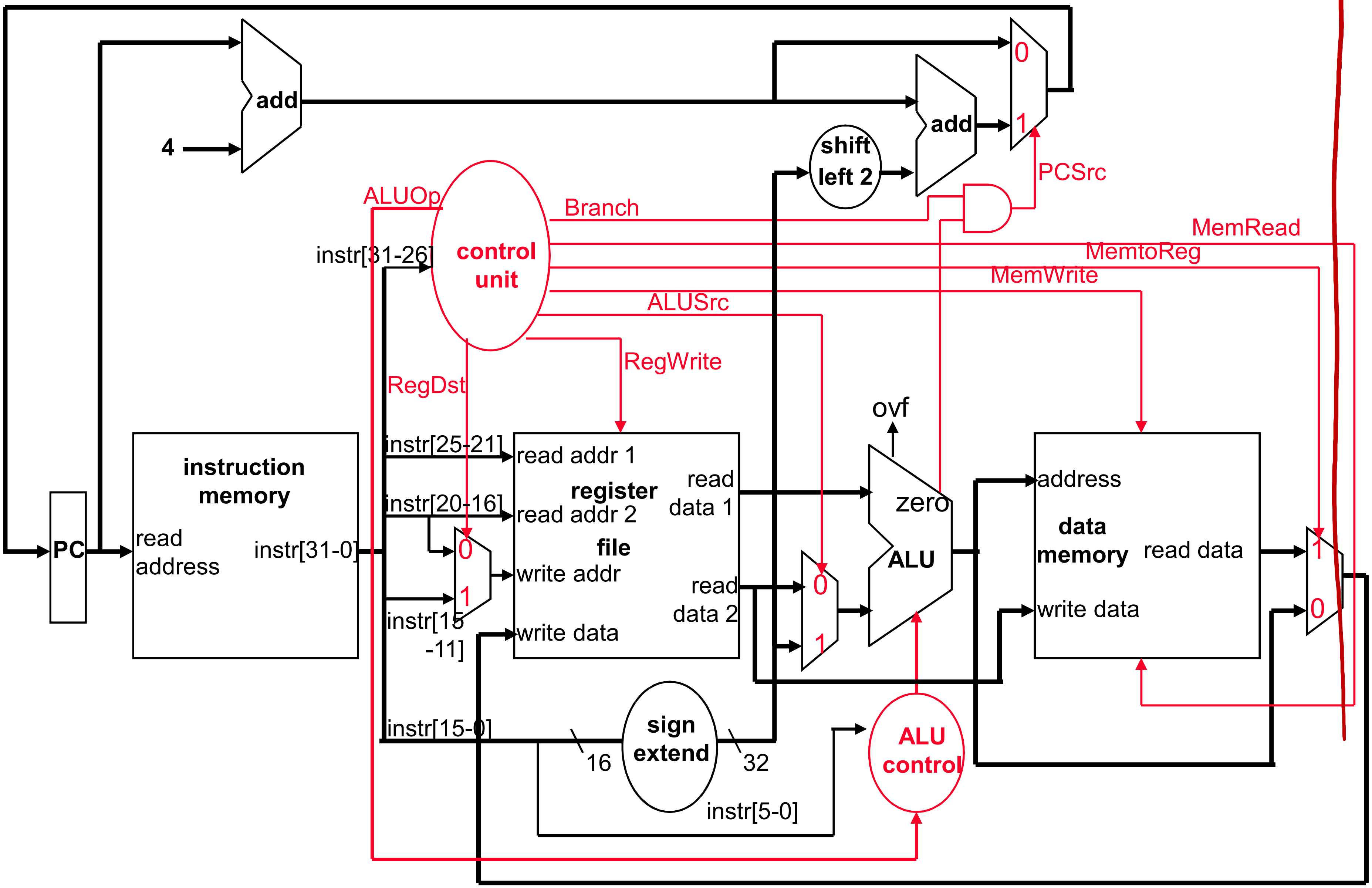
R-type: 31 op 25 rs 20 rt 15 rd 10 shamt 5 funct 0

I-Type: 31 op 25 rs 20 rt 15 address offset 0

J-type: 31 op 25 target address 0

- 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

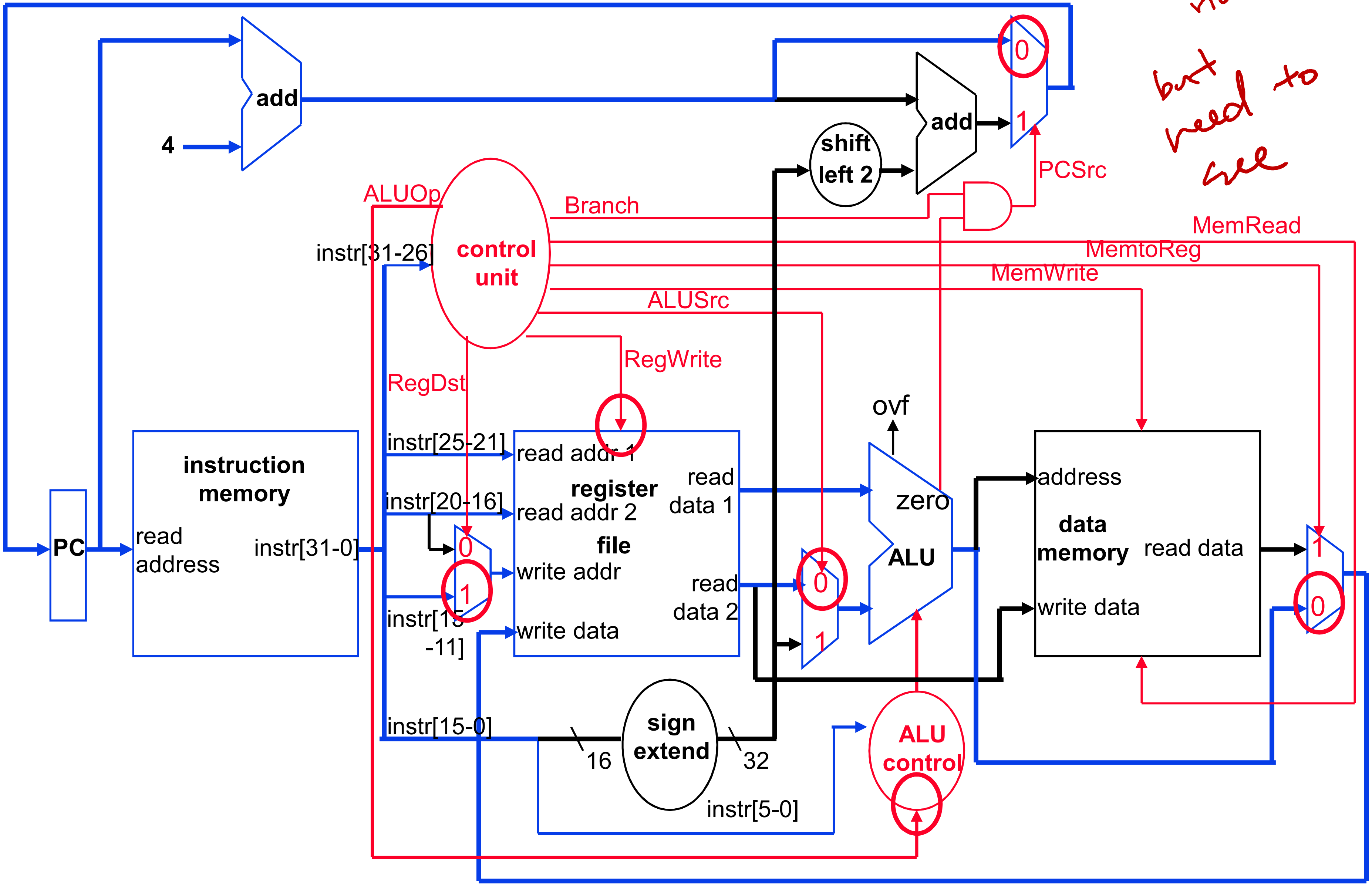
Single Cycle Datapath w/ Control Unit



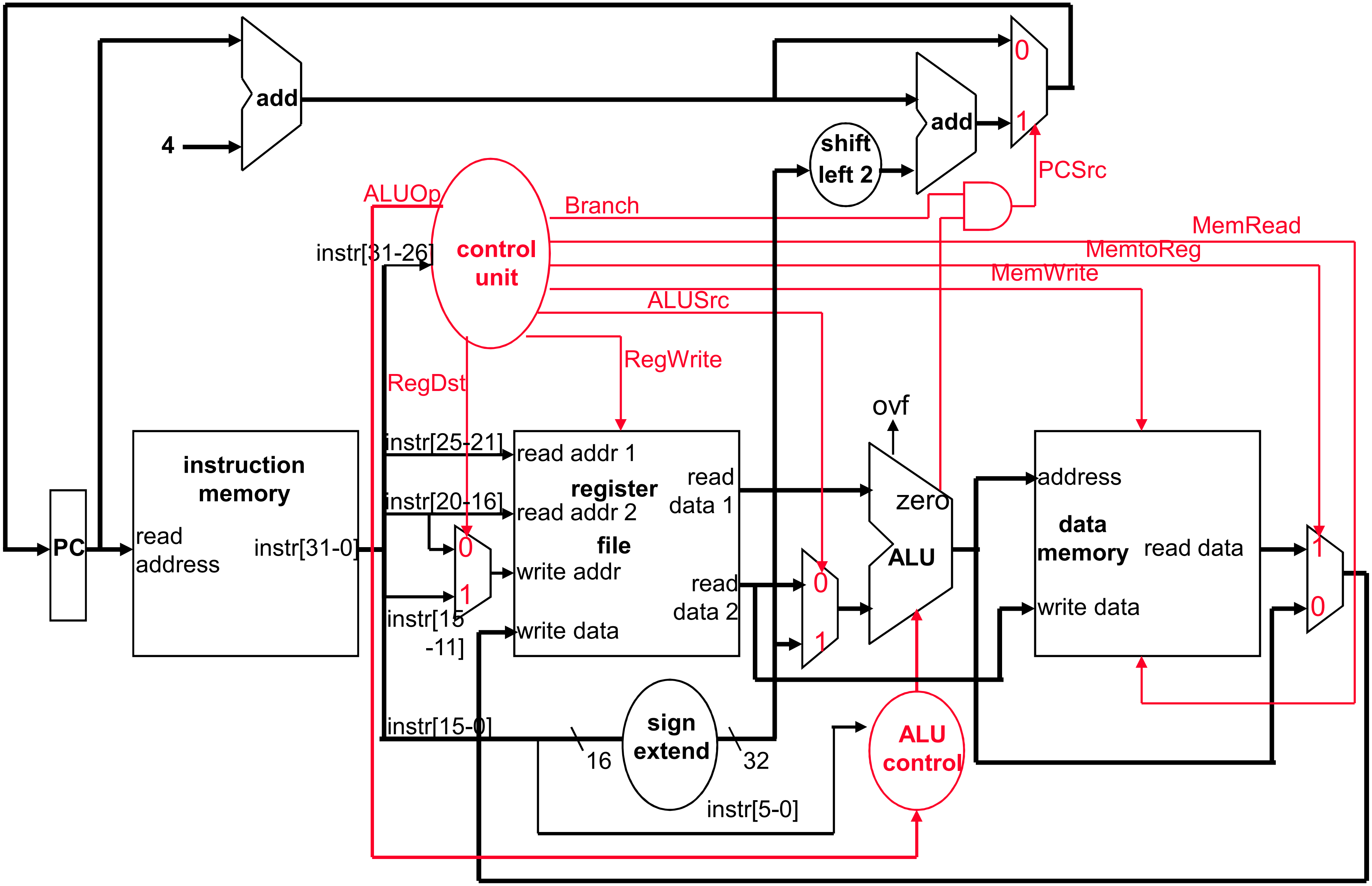
R-type Instruction Data/Control Flow

skip this now

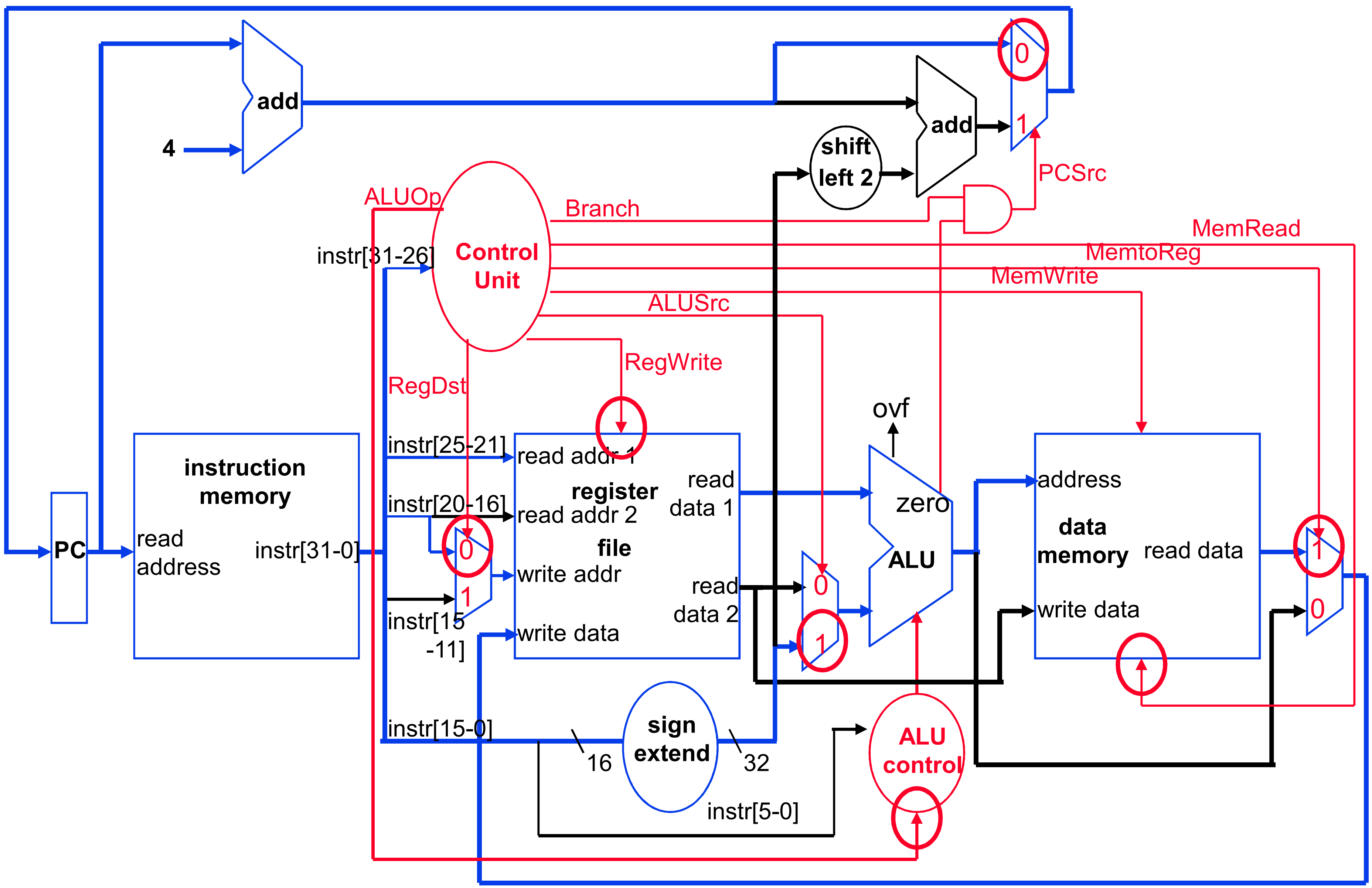
but need to see



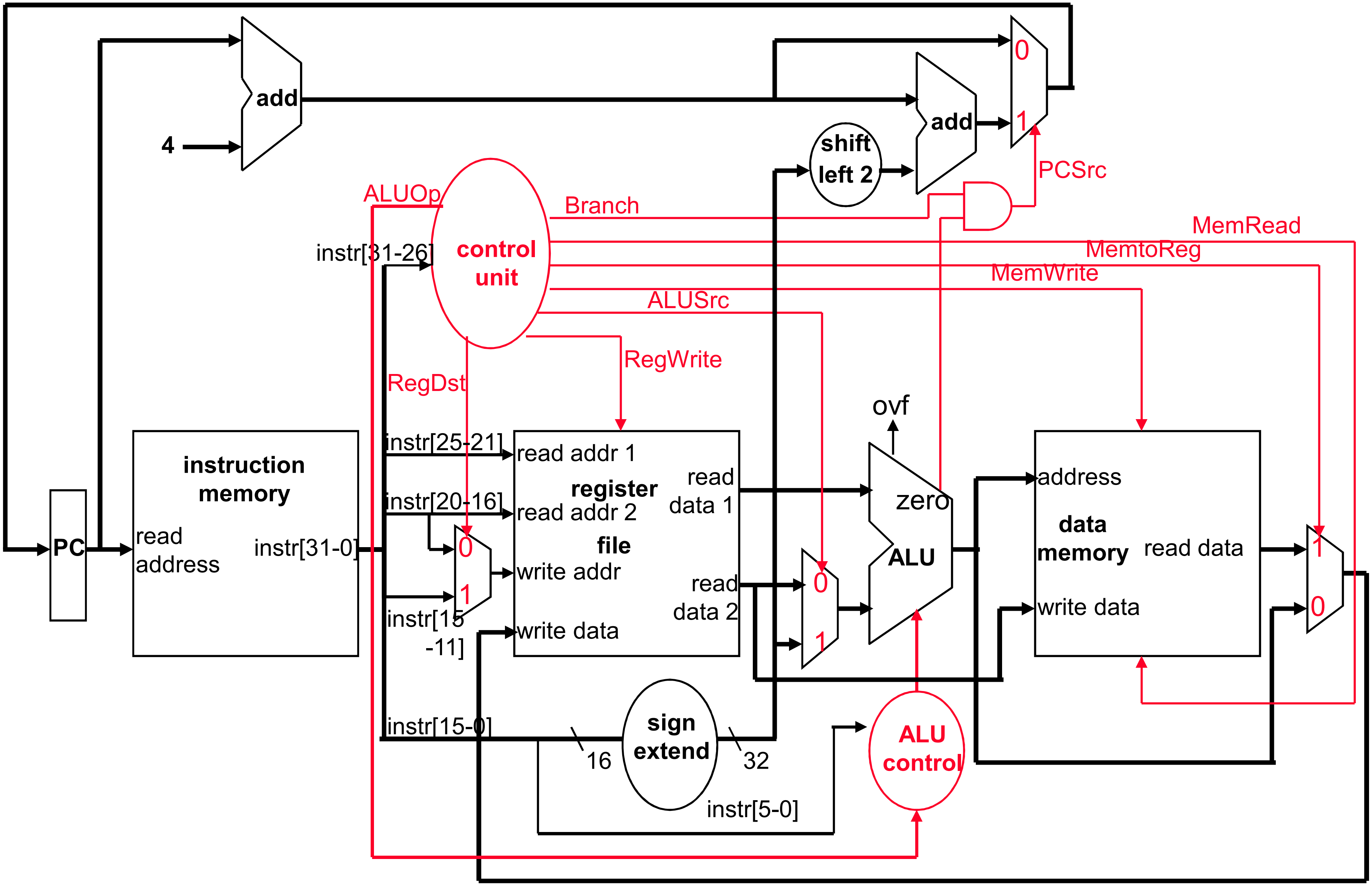
Load Word Instruction Data/Control Flow



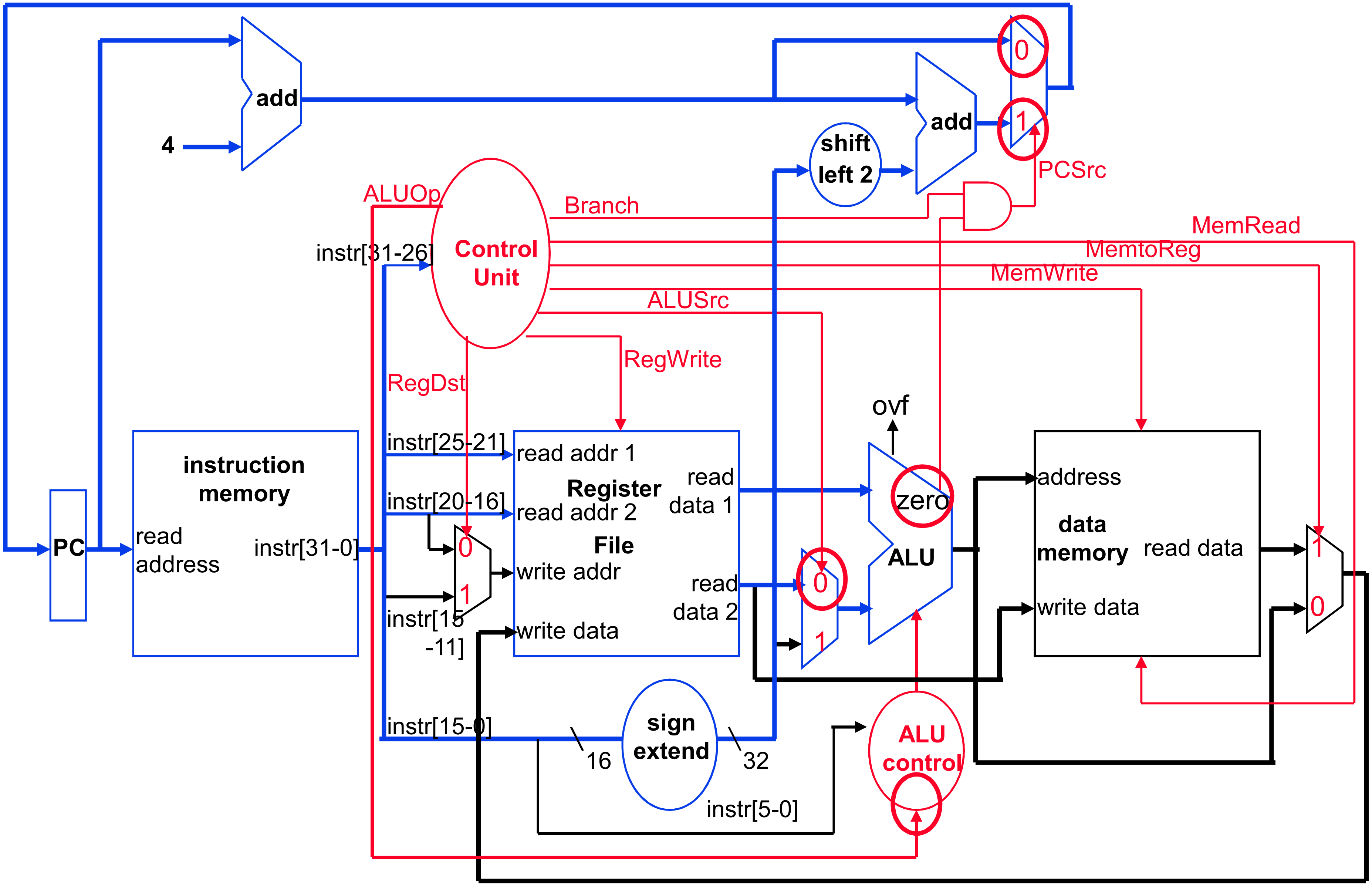
Load Word Instruction Data/Control Flow



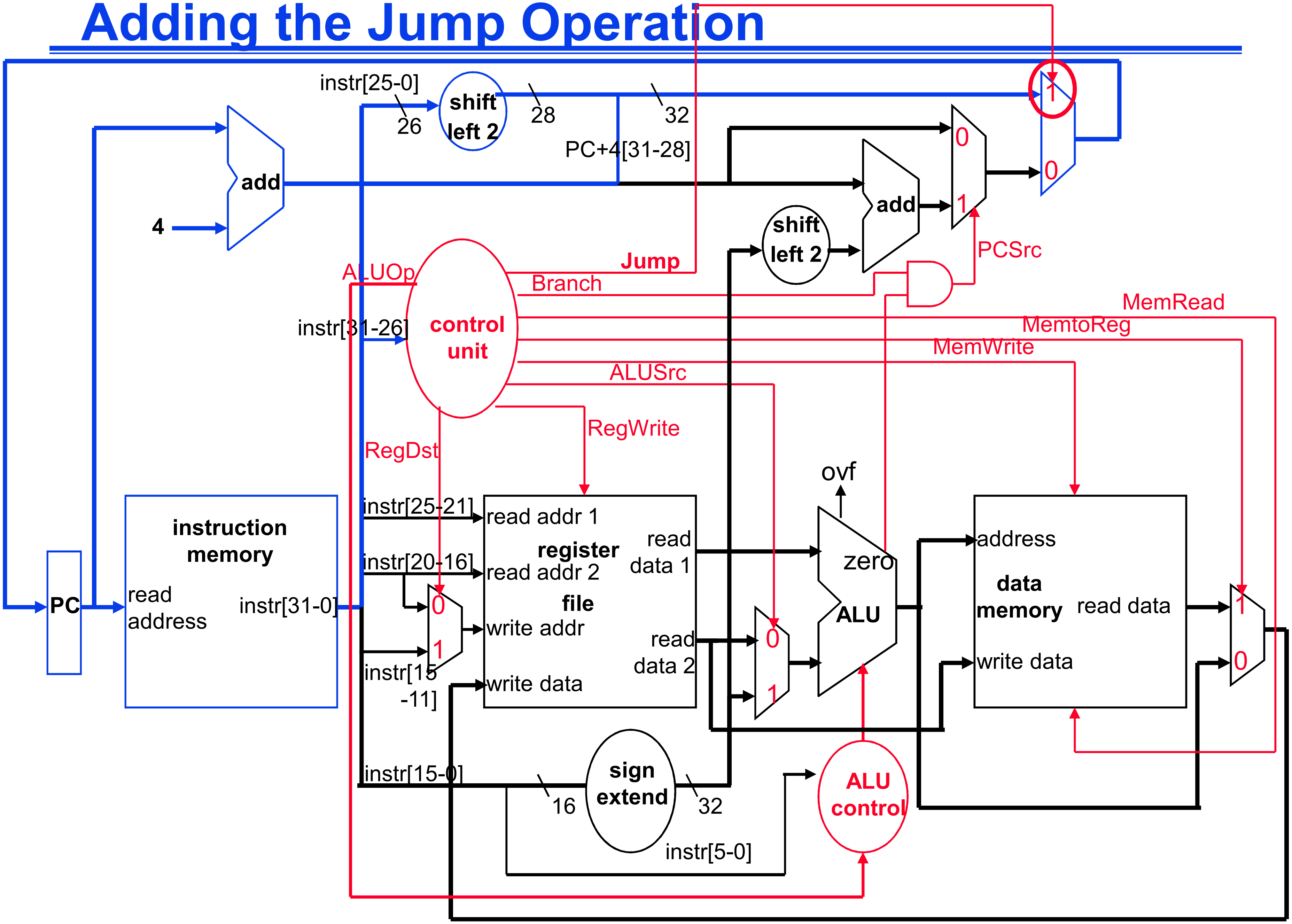
Branch Instruction Data/Control Flow



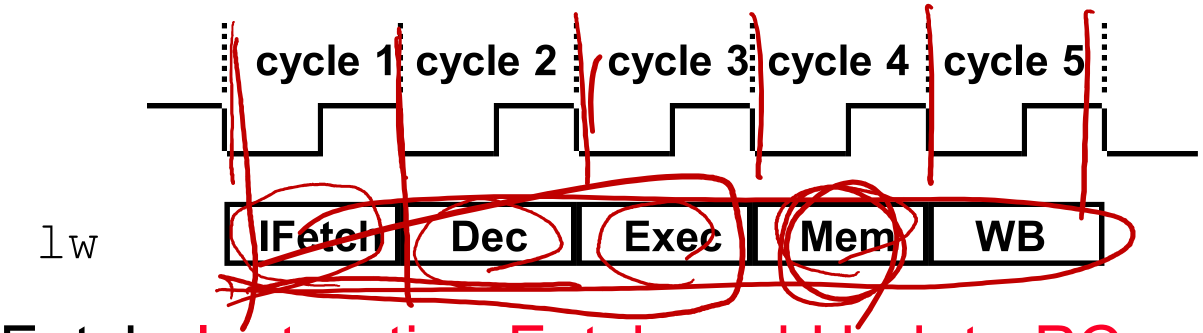
Branch Instruction Data/Control Flow



Adding the Jump Operation



Instruction Type vs # of Required Cycles

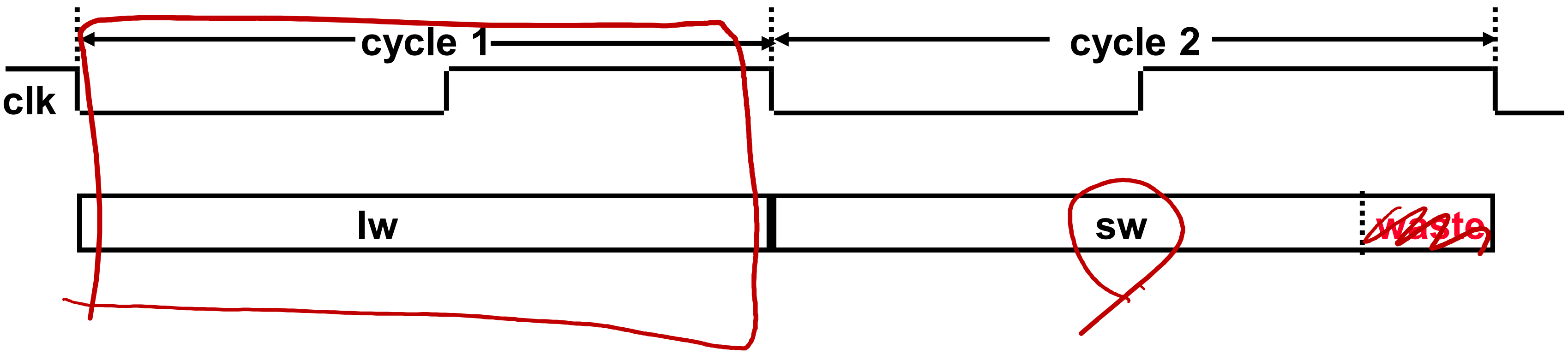


- ❑ IFetch: Instruction Fetch and Update PC
- ❑ Dec: Instruction Decode, Register Read, Sign Extend Offset
- ❑ Exec: Execute R-type; Calculate Memory Address; Branch Comparison; Branch and Jump Completion
- ❑ Mem: Memory Read; Memory Write Completion; R-type Completion (RegFile write)
- ❑ WB: Memory Read Completion (RegFile write)

→ *INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!*

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

Multicycle Datapath Approach

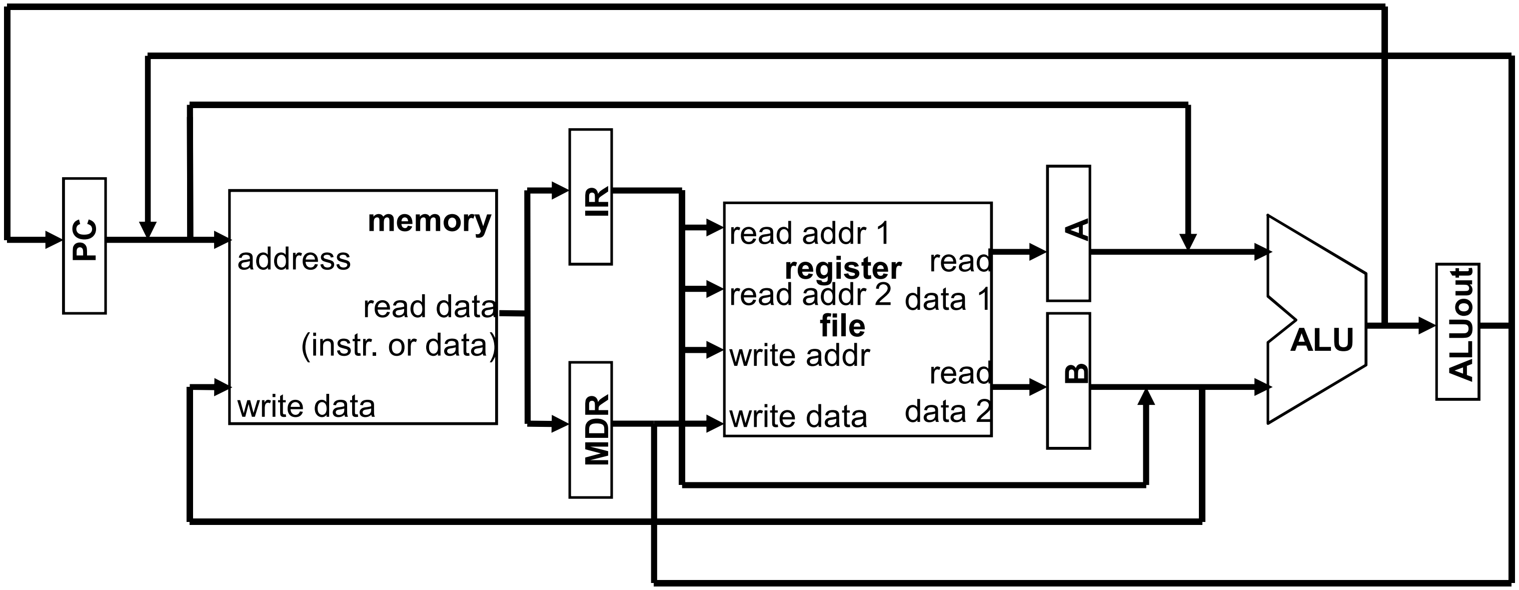
Look at the detail

- let an instruction take more than 1 clock cycle to complete
  - break up instructions into steps where each *step* takes a cycle while trying to
    - balance the amount of work to be done in each step
    - restrict each cycle to use only one major functional unit
  - not every instruction takes the *same* number of clock cycles

- in addition to faster clock rates, multicycle allows functional units that can be used more than once per instruction as long as they are used on *different* clock cycles, as a result
  - only need one memory – but only one memory access per cycle
  - need only one ALU/adder – but only one ALU operation per cycle

Multicycle Datapath Approach, con't

- at the end of a cycle you can have a look at this
  - store values needed in a later cycle by the current instruction in an internal register (not visible to the programmer); all (except IR) hold data only b/w a pair of adjacent clock cycles (no write control signal needed)



IR – Instruction Register

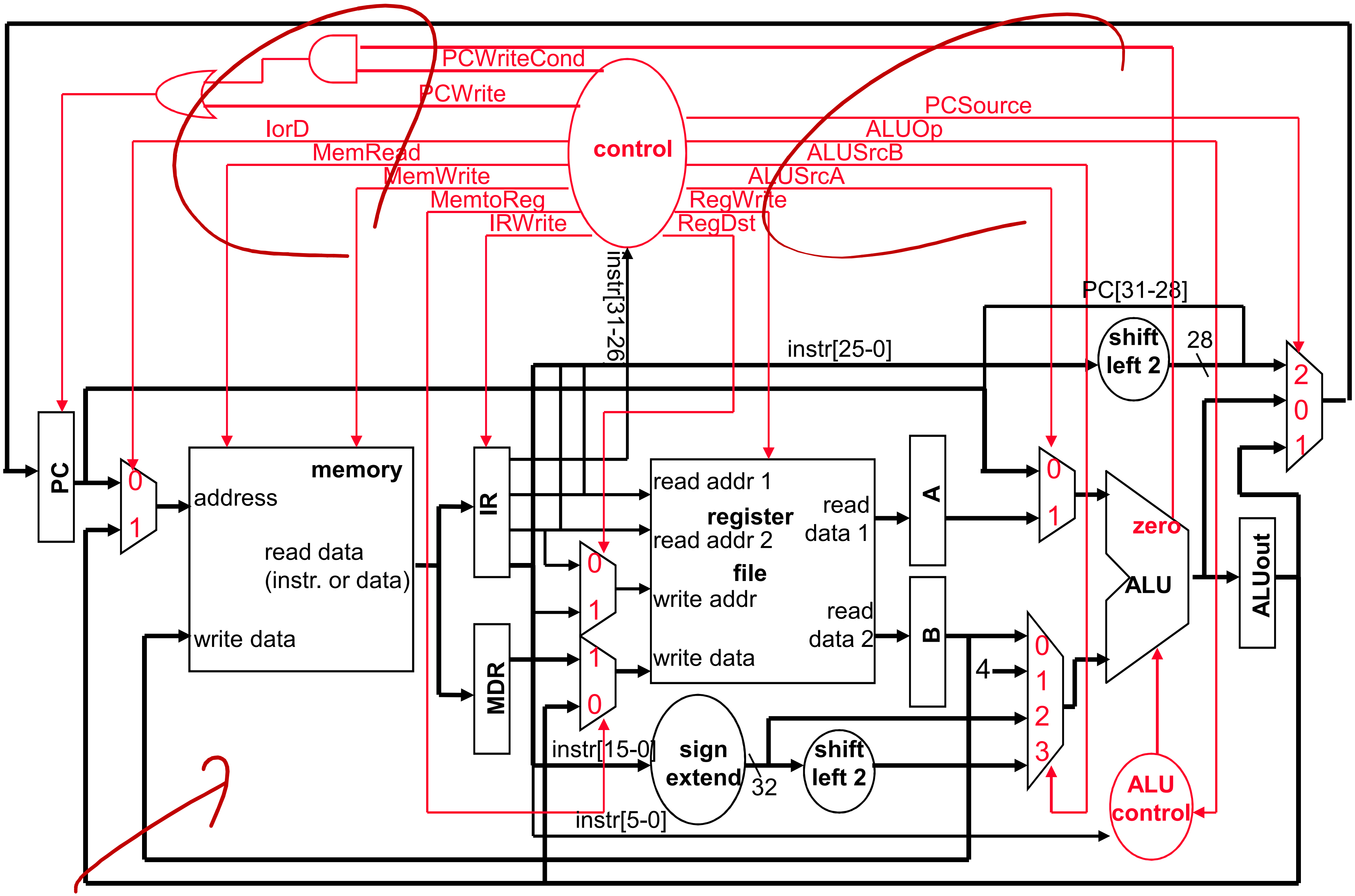
MDR – Memory Data Register

A, B – regfile read data registers

ALUout – ALU output register

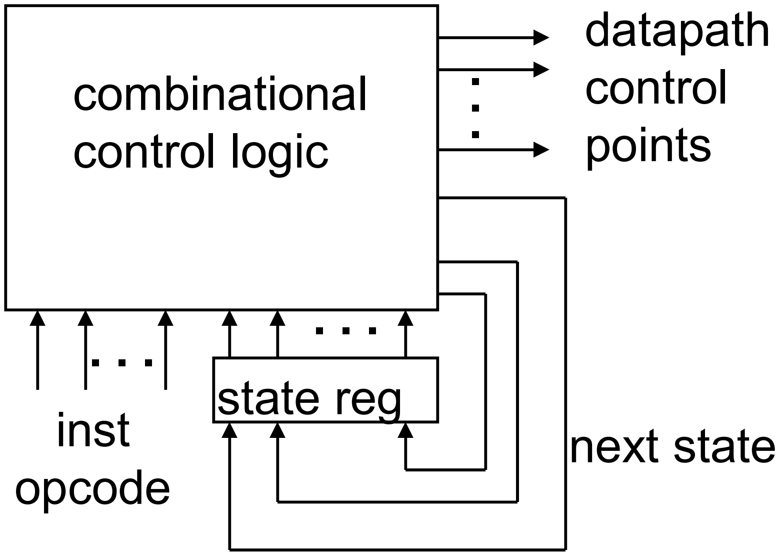
  - data used by subsequent instructions are stored in programmer visible registers (i.e., register file, PC, or memory)

The Multicycle Datapath w/ Control Signals



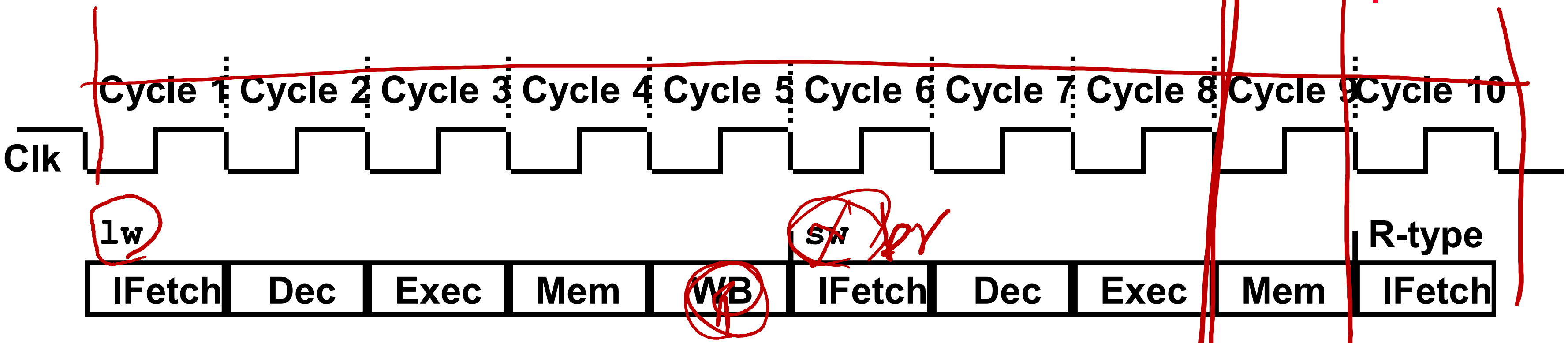
Multicycle Control Unit

- multicycle datapath control signals are not determined solely by the bits in the instruction
    - e.g., op code bits tell what operation the ALU should be doing, but *not* what instruction cycle is to be done next
- must use a finite state machine (FSM) for control
    - a set of states (current state stored in state register)
    - next state function (determined by current state and the input)
    - output function (determined by current state and the input)



Multicycle Advantages & Disadvantages

- uses the clock cycle efficiently – the clock cycle is timed to accommodate the slowest instruction step

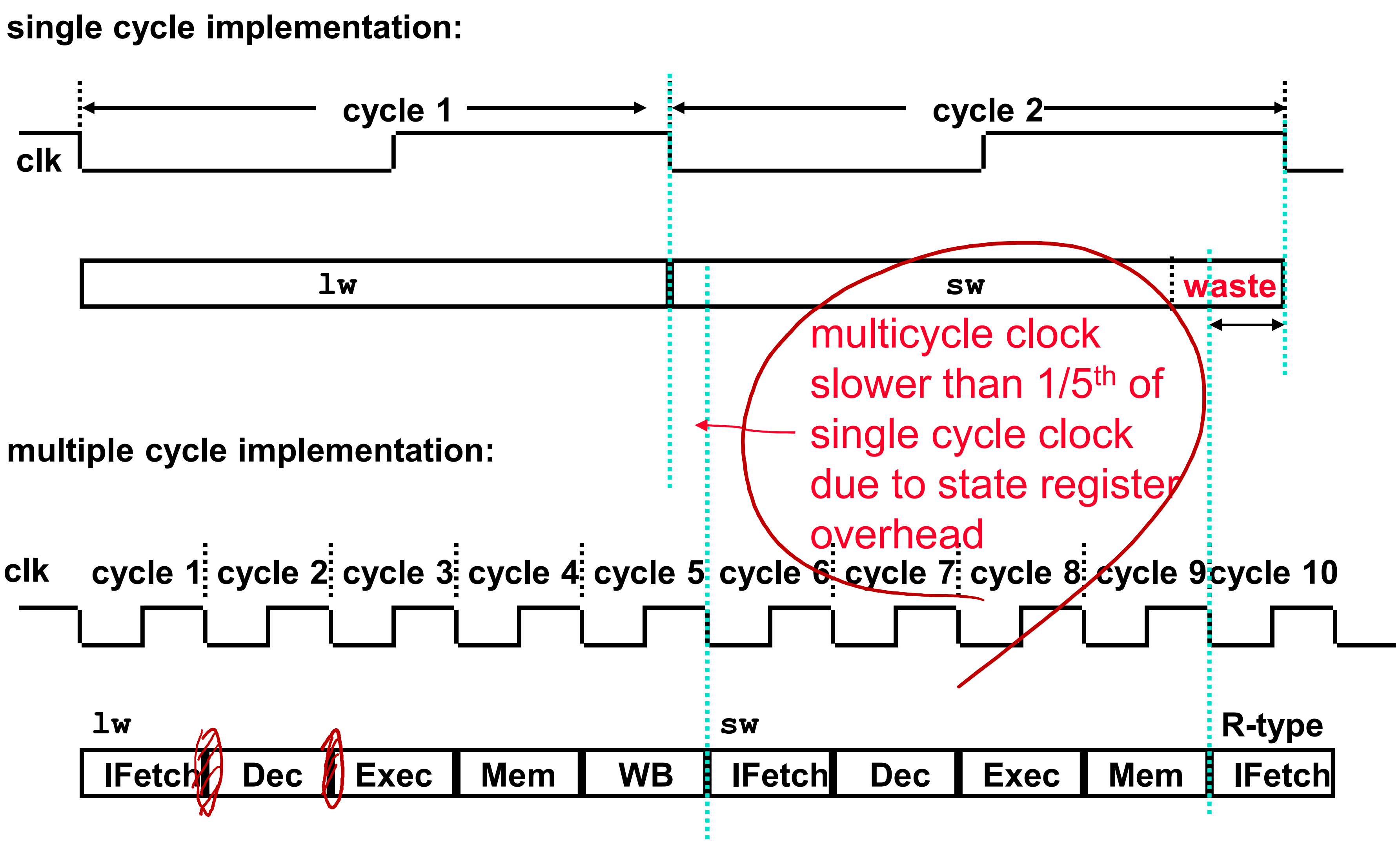


- multicycle implementations allow functional units to be used more than once per instruction as long as they are used on different clock cycles

but

- requires additional internal state registers, more muxes, and more complicated (FSM) control

Single Cycle vs. Multiple Cycle Timing



A Hypothetical Data Path Timing Analysis

- critical path: a path through combinational circuit that takes as long or longer than any other.

	I Fetch	Decode, R-Read	ALU	PC update	D Memory	R-Write	Total (ns)
Add	1	1	.9	-	-	.8	3.7
Load	1	1	.9	-	1	.8	4.7
Store	1	1	.9	-	1	-	3.9
beq	1	1	.9	.1	-	-	3.0

clock cycle time

$= 4.7\text{ns} + t_{setup} + t_{cq}$

T_{cycle}

A Hypothetical Data Path Timing Analysis

- critical path: a path through combinational circuit that takes as long or longer than any other

clock cycle time

$= 4.7\text{ns} + t_{setup} + t_{cq}$

	I Fetch	Decode, R-Read	ALU	PC update	D Memory	R-Write	Total (ns)
Add	1	1	.9	-	-	.8	3.7
Load	1	1	.9	-	1	.8	4.7
Store	1	1	.9	-	1	-	3.9
beq	1	1	.9	.1	-	-	3.0

Cycle Time vs. CPI

- goal: balance amount of work done each cycle
  - ✓ load needs 5 cycles
  - ✓ add and store need 4
  - ✓ beq needs 3

singlecycle

multi-cycles

	I Fetch	Decode, R-Read	ALU	PC update	D Memory	R-Write	Total (ns)
Add	1	1	.9	-	-	.8	3.7
Load	1	1	.9	-	1	.8	4.7 ns
Store	1	1	.9	-	1	-	3.9
beq	1	1	.9	.1	-	-	3.0

Will multicycle design be faster?

- let’s assume t_{setup} + t_{cq} time for registers = 0.1 ns
  - ✓ single cycle design:
    - ○ clock cycle time = 4.7 + 0.1 = 4.8 ns
    - ○ time/inst = 1 cycle/inst × 4.8 ns/cycle = 4.8 ns/inst
  - ✓ multicycle design:
    - ○ clock cycle time = 1.0 + 0.1 = 1.1
    - ○ time/inst = CPI × 1.1 ns/cycle (depends on the types or mixture of instructions!)

	I Fetch	Decode, R-Read	ALU	PC update	D Memory	R-Write	Total (ns)
Add	1	1	.9	-	-	.8	3.7
Load	1	1	.9	-	1	.8	4.7
Store	1	1	.9	-	1	-	3.9
beq	1	1	.9	.1	-	-	3.0

Will multicycle design be faster?

- let's assume t_{setup} + t_{cq} time for the register = 0.1 ns
  - ✓ single cycle design:
    - ○ clock cycle time = 4.7 + 0.1 = 4.8 ns
    - ○ time/inst = 1 cycle/inst × 4.8 ns/cycle = 4.8 ns/inst
  - ✓ multicycle design:
    - ○ clock cycle time = 1.0 + 0.1 = 1.1
    - ○ time/inst = CPI × 1.1 ns/cycle

⇒ 4.1×1.1

	Cycles needed	Instruction frequency
Add	4	60%
Load	5	20%
Store	4	10%
beq	3	10%

4×0.6+5×0.2 . . . .

CPI

= 4×0.6+5×0.2+4×0.1+3×0.1

= 2.4+1.0+0.4+0.3

= 4.1

A More Extreme Example

You can do this on your own

- calculation assumptions:
  - ✓ most instructions take 10 nanoseconds (ns)
  - ✓ but multiply instruction takes 40ns
  - ✓ multiplies are 10% of all instructions
- how much faster is a multi-cycle data path over a single-cycle data path?
  - ✓ for simplicity, assume t_{setup} + t_{cq} =0
    - ○ single-cycle time/inst = 40ns
    - ○ multi-cycle with 10ns clock, time/inst = (1 × 0.9 + 4 × 0.1) × 10 = 13ns

Announcement

- next lecture: pipeline (overview)
  - ✓ Ch. 4.5 – 4.6 (HP1)
- MP assignment
  - ✓ MP1 checkpoint 1 due on 1/28 5pm