

# **ECE/CS 250**

# **Computer Architecture**

## **Summer 2024**

### Processor Design: Datapath and Control

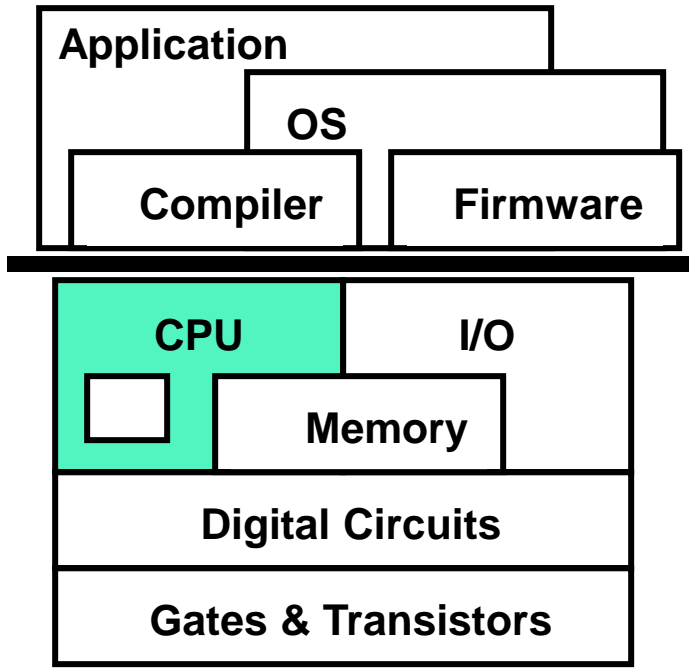
Tyler Bletsch  
Duke University

Slides are derived from work by  
Daniel J. Sorin (Duke), Amir Roth (Penn)

# Where We Are in This Course Right Now

- So far:
  - We know what a computer architecture is
  - We know what kinds of instructions it might execute
  - We know how to perform arithmetic and logic in an ALU
- Now:
  - We learn how to design a processor in which the ALU is just one component
  - Processor must be able to fetch instructions, decode them, and execute them
  - There are many ways to do this, even for a given ISA
- Next:
  - We learn how to design memory systems

# This Unit: Processor Design



- Datapath components and timing
  - Registers and register files
  - Memories (RAMs)
- Mapping an ISA to a datapath
- Control
- Exceptions

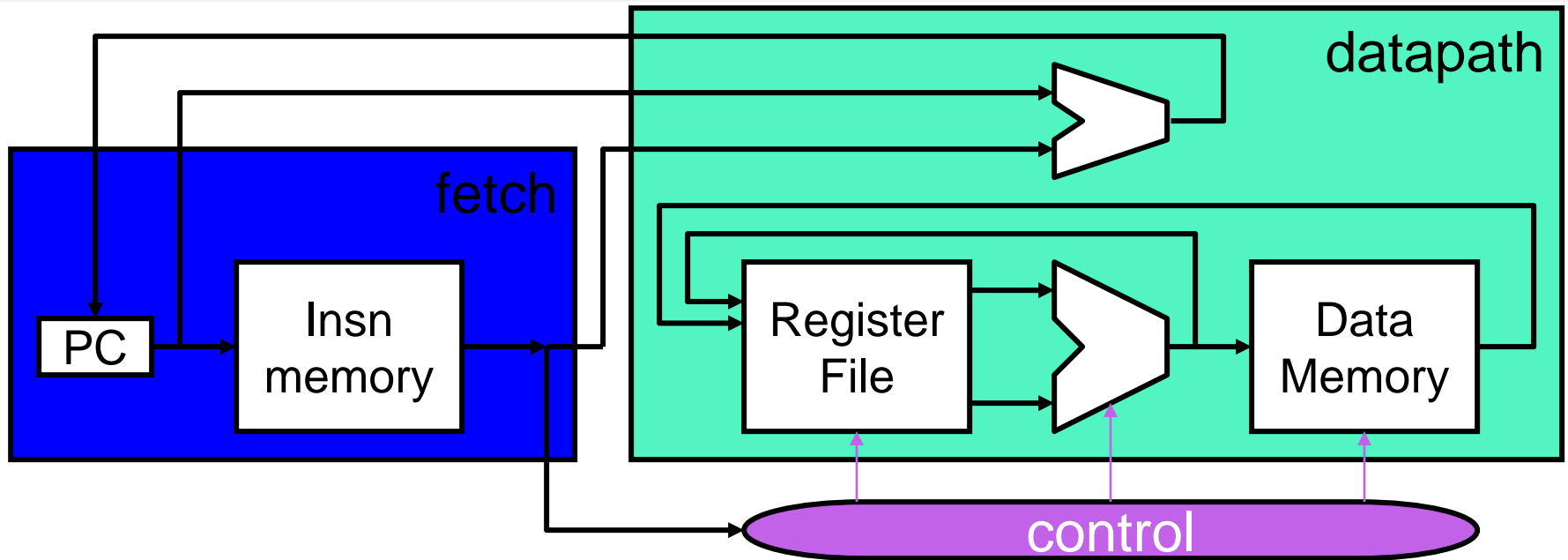
# Readings

- Patterson and Hennessy
  - Chapter 4: Sections 4.1-4.4
- Read this chapter carefully
  - It has many more examples than I can cover in class

# So You Have an ALU...

- **Important reminder:** a processor is just a big finite state machine (FSM) that interprets some ISA
- Start with one instruction
  - add** \$3, \$2, \$4
    - ALU performs just a small part of execution of instruction
    - You have to read and write registers
    - You have have to fetch the instruction to begin with
- What about loads and stores?
  - Need some sort of memory interface
- What about branches?
  - Need some hardware for that, too

# Datapath and Control



- **Datapath**: registers, memories, ALUs (computation)
- **Control**: which registers read/write, which ALU operation
- **Fetch**: get instruction, translate into control
- Processor Cycle: **Fetch** → **Decode** → **Execute**

# Building a Processor for an ISA

- Fetch is pretty straightforward
  - Just need a register (called the Program Counter or PC) to hold the next address to fetch from instruction memory
  - Provide address to instruction memory → instruction memory provides instruction at that address
- Let's start with the datapath
  1. Look at ISA
  2. Make sure datapath can implement every instruction

# Datapath for MIPS ISA

- Consider only the following instructions

`add $1,$2,$3`

`addi $1,$2,<value>`

`lw $1,4($3)`

`sw $1,4($3)`

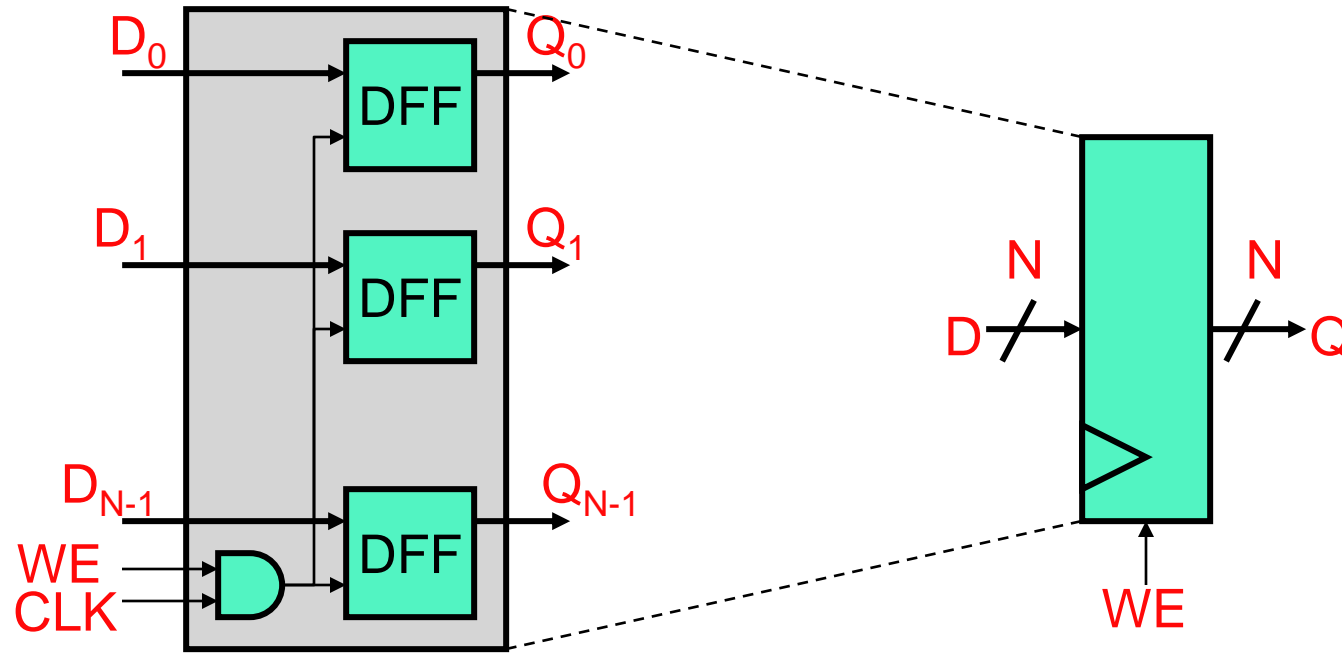
`beq $1,$2,PC_relative_target`

`j Absolute_target`

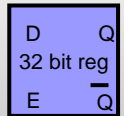
- Why only these?
  - Most other instructions are similar from datapath viewpoint
  - I leave the ones that aren't for you to figure out



# Review: A Register



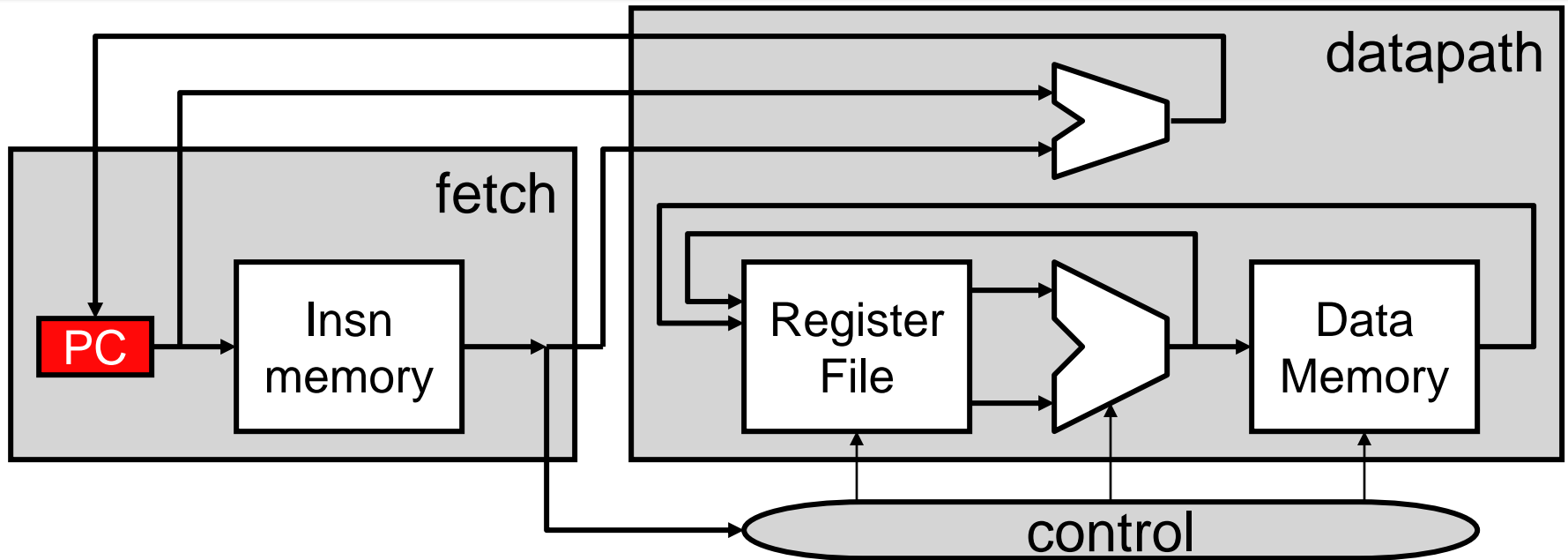
=



Note: Above is the "classic" register we learned before; we're just introducing a new symbol for the same thing

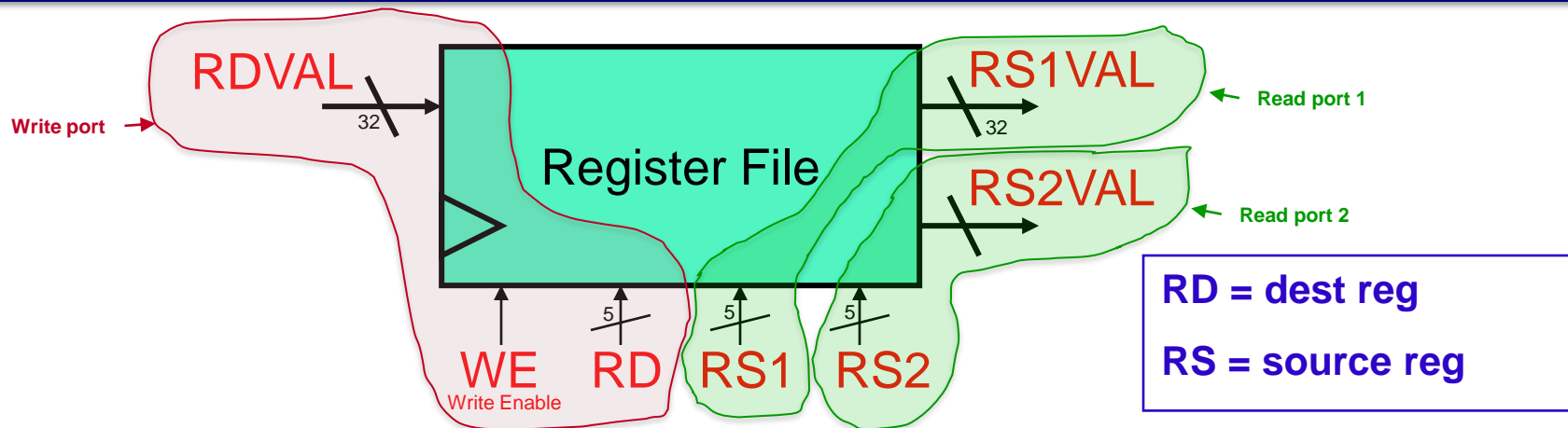
- **Register:** DFF array with shared clock, write-enable (WE)
  - Notice: both a clock and a WE ( $\text{DFF}_{\text{WE}} = \text{clock} \ \& \ \text{register}_{\text{WE}}$ )
  - Convention I: clock represented by wedge
  - Convention II: if no WE, DFF is written on every clock

# Uses of Registers



- A single register is good for some things
  - PC: program counter
  - Other things which aren't the ISA registers (more later in semester)

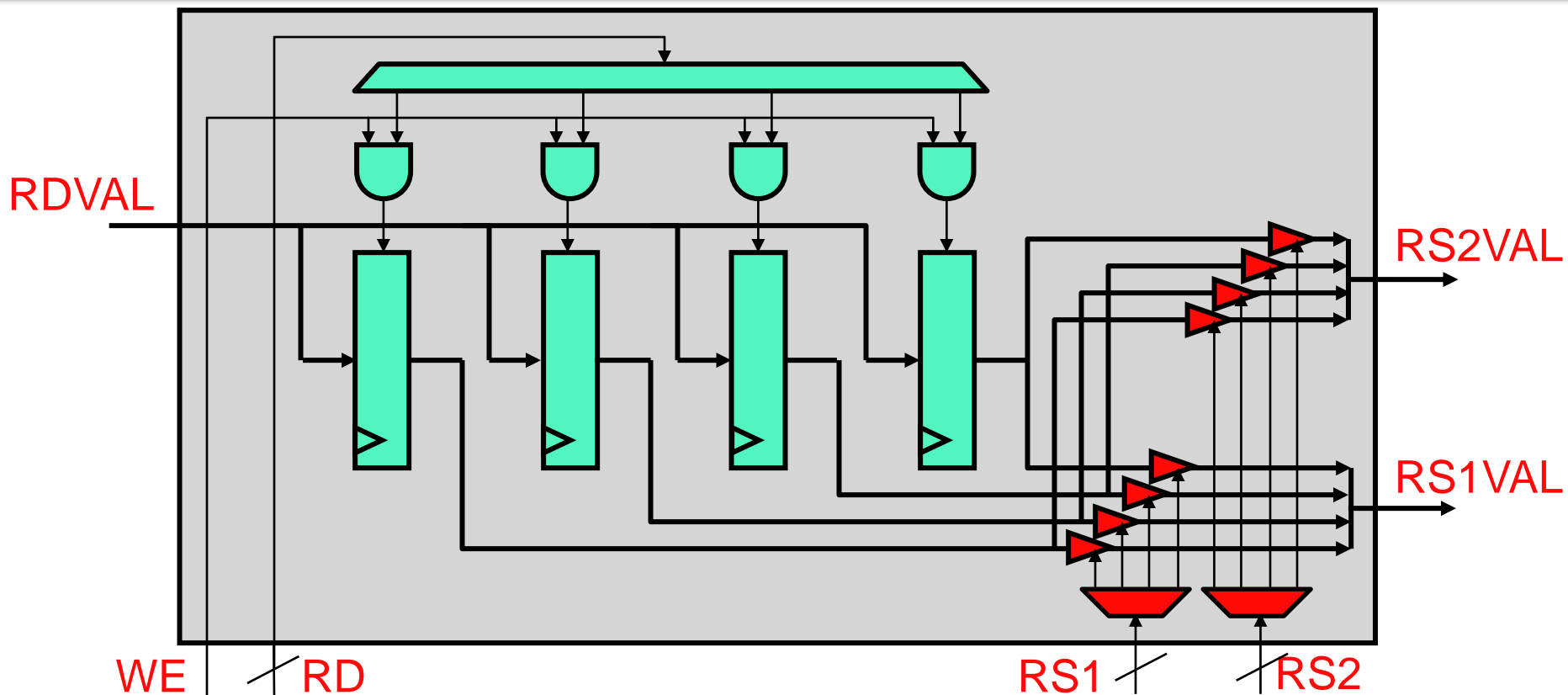
# What About the ISA Registers?



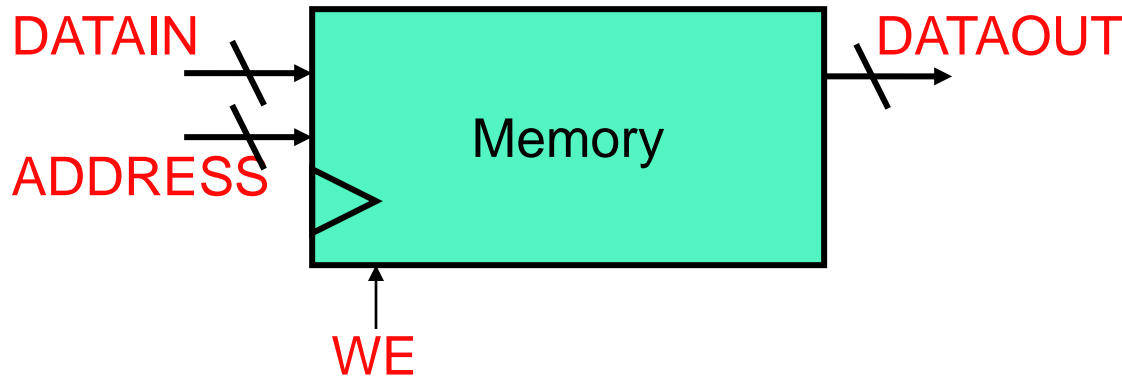
- **Register file:** the ISA ("architectural", "visible") registers
  - Two read "ports" + one write "port"
  - Maximum number of reads/writes in single instruction (R-type)
- **Port:** wires for accessing an array of data
  - Data bus: width of data element (MIPS: 32 bits)
  - Address bus: width of  $\log_2$  number of elements (MIPS: 5 bits)
  - Write enable: if it's a write port
  - M ports = M parallel and independent accesses

Reminder

# Register File With Tri-State Read Ports

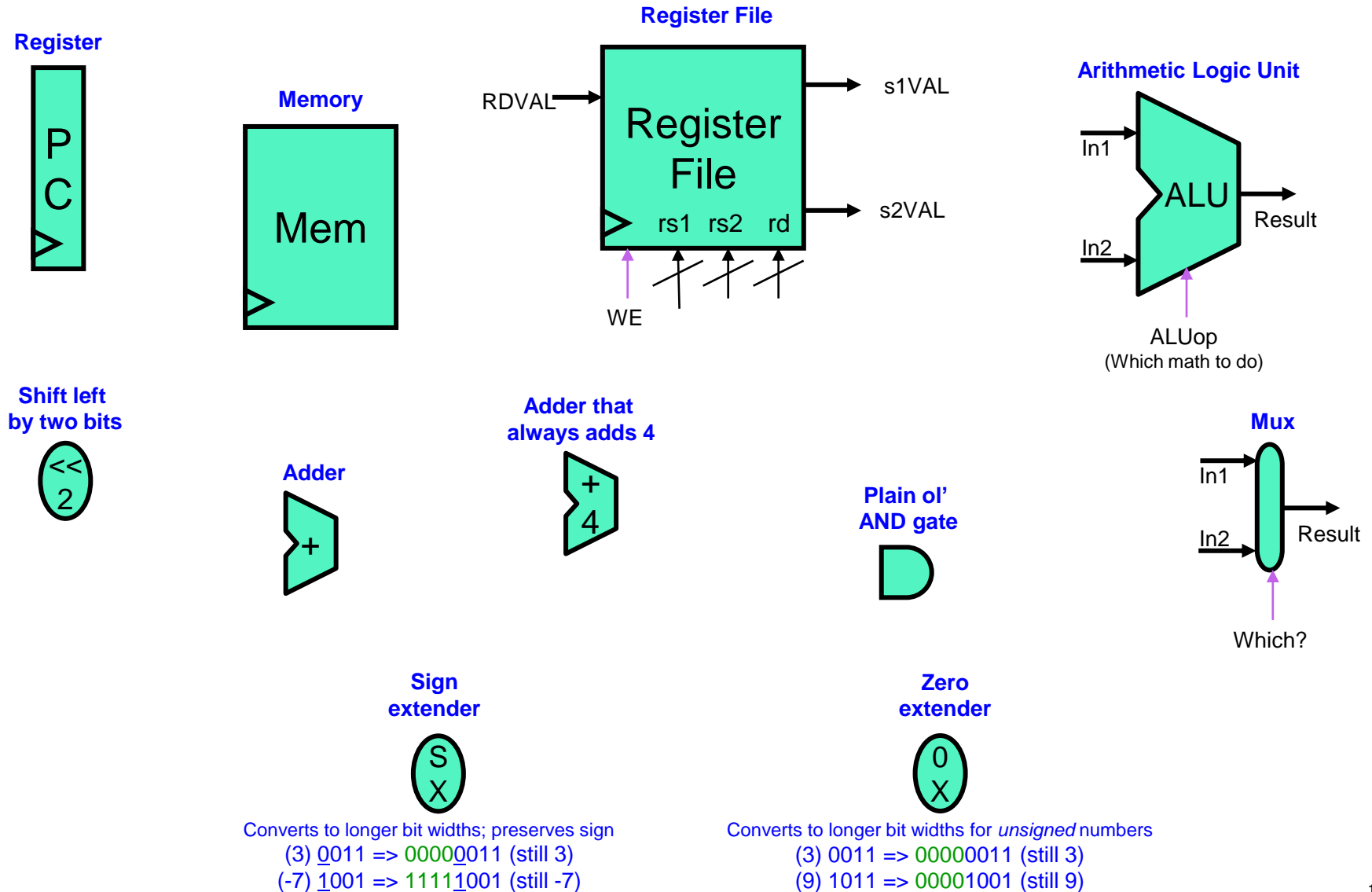


# Another Useful Component: Memory



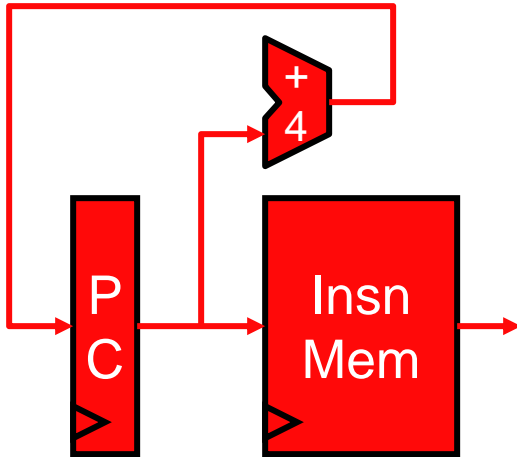
- **Memory**: where instructions and data reside
  - One read/write "port": one access per cycle, either read **or** write
    - One address bus
  - One input data bus for writes, one output data bus for reads
- Actually, a more traditional definition of memory is
  - One input/output data bus
  - No clock → asynchronous "strobe" instead

# Dramatis Personae



# Let's Build A MIPS-like Datapath

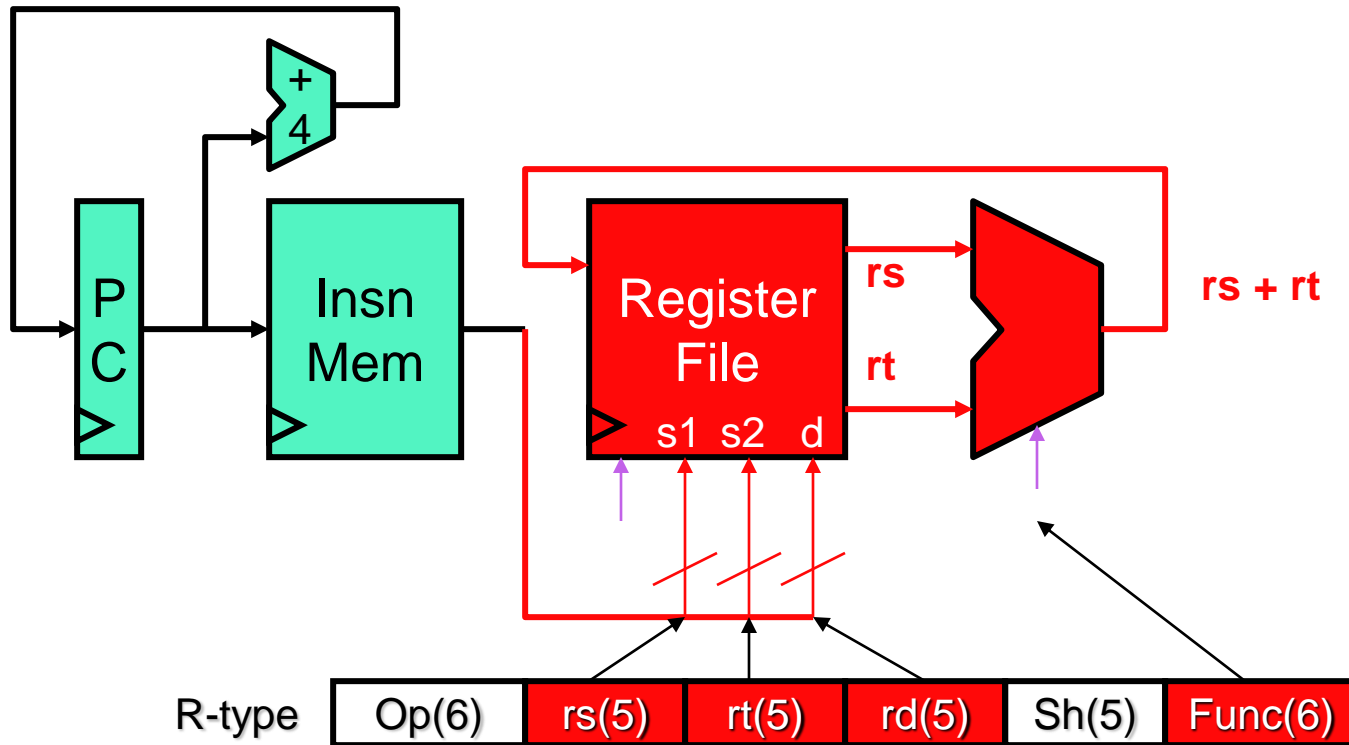
# Start With Fetch



- PC and instruction memory
- A +4 incrementer computes default next instruction PC
  - Why +4 (and not +1)? What will it be for 16-bit Duke 250/16?

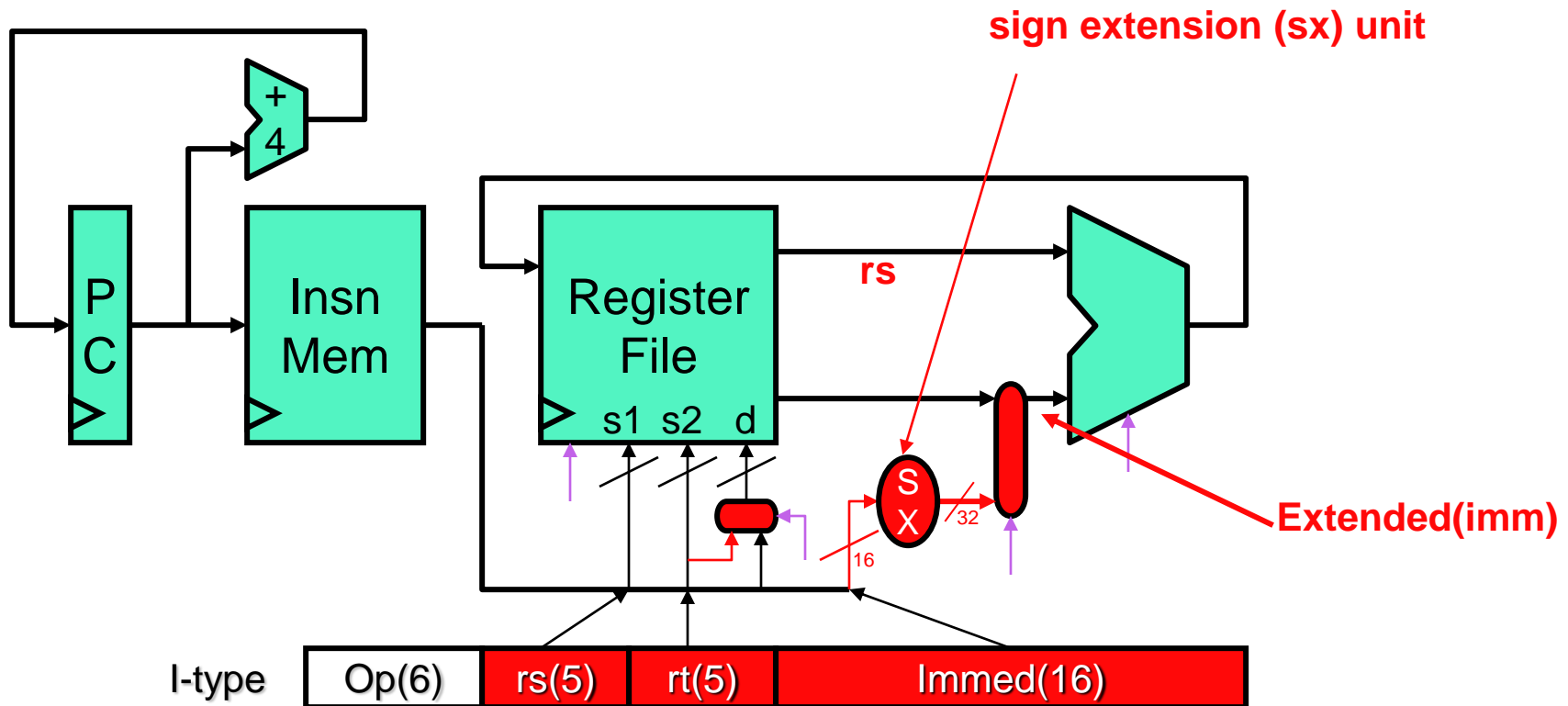


# First Instruction: add \$rd, \$rs, \$rt



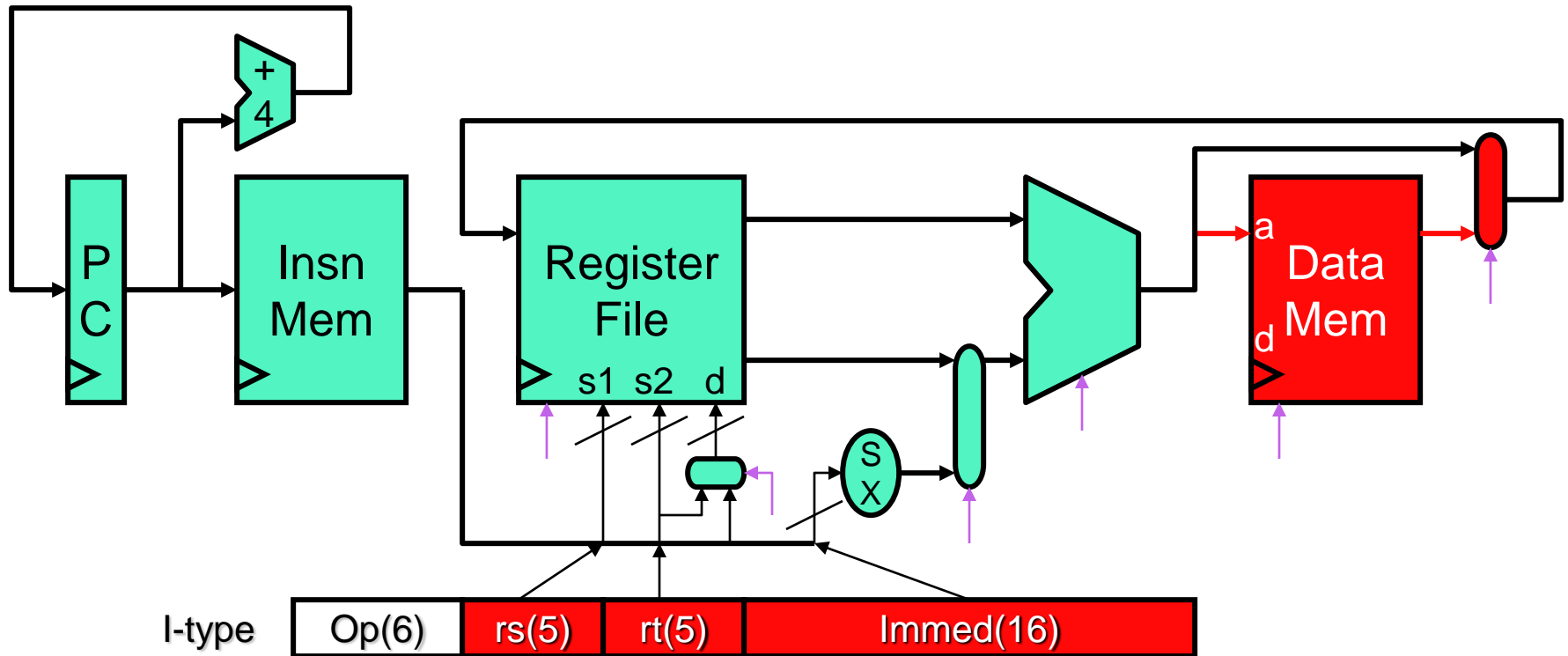
- Add register file and ALU

# Second Instruction: addi \$rt, \$rs, imm



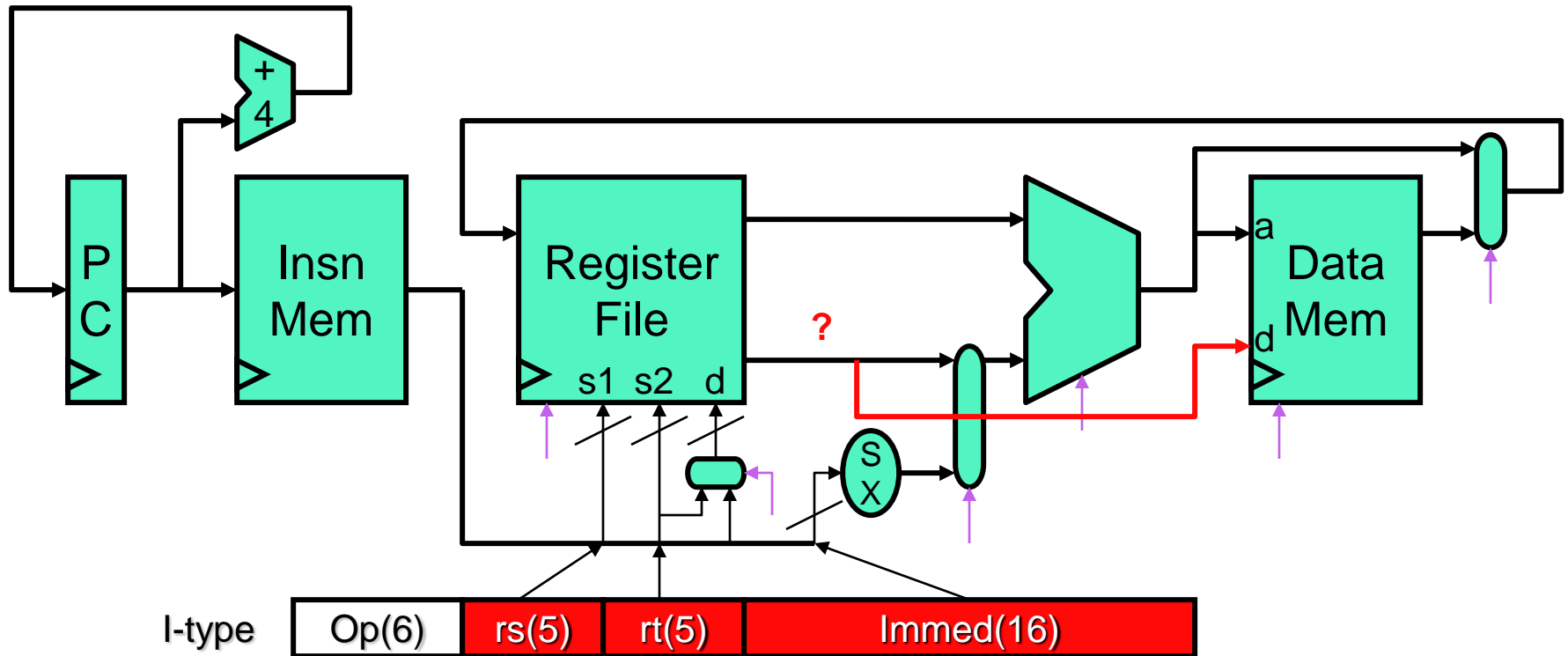
- Destination register can now be either rd or rt
- Add sign extension unit and mux into second ALU input

# Third Instruction: lw \$rt, imm(\$rs)



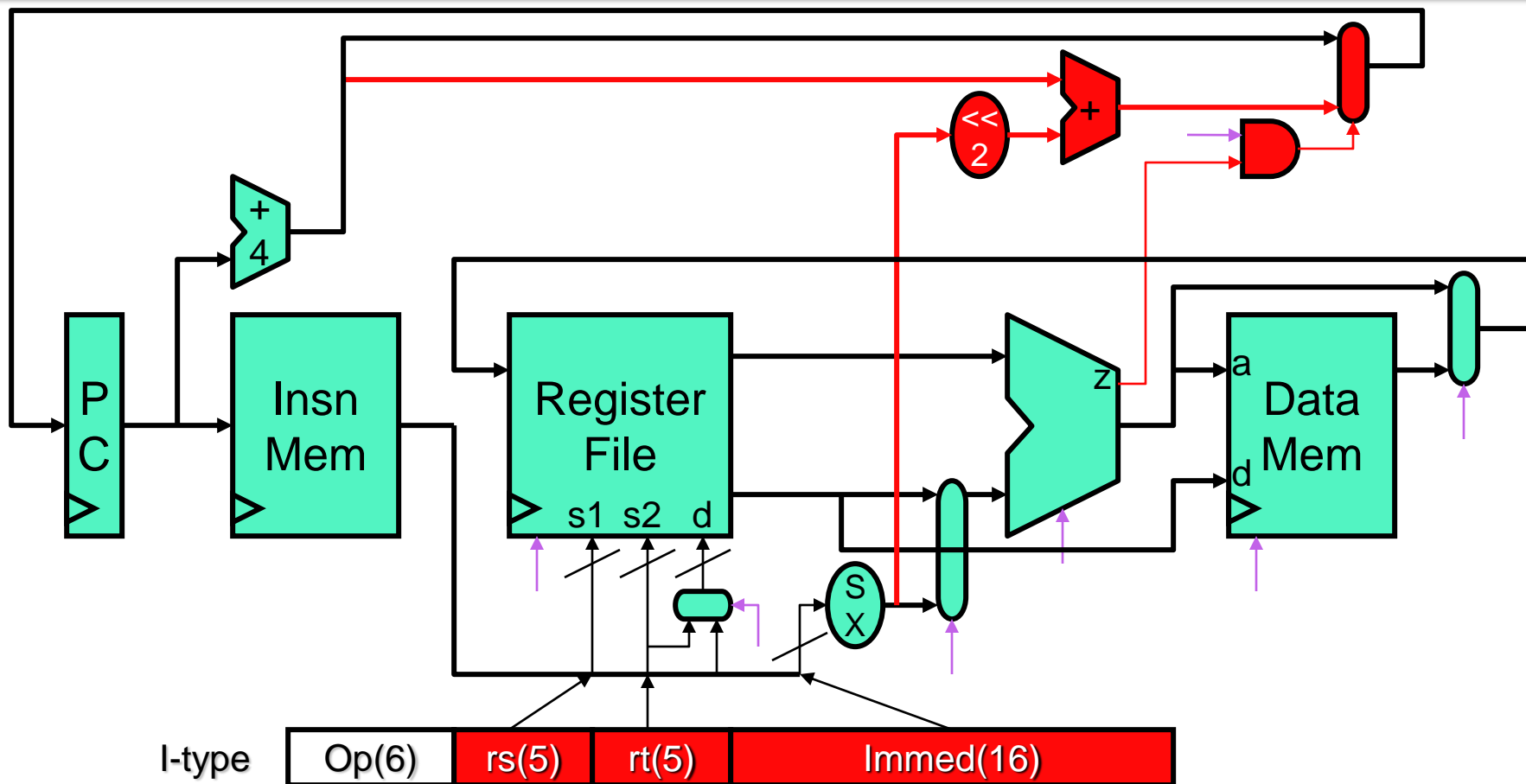
- Add data memory, address is ALU output ( $rs+imm$ )
- Add register write data mux to select memory output or ALU output

# Fourth Instruction: sw \$rt, imm(\$rs)



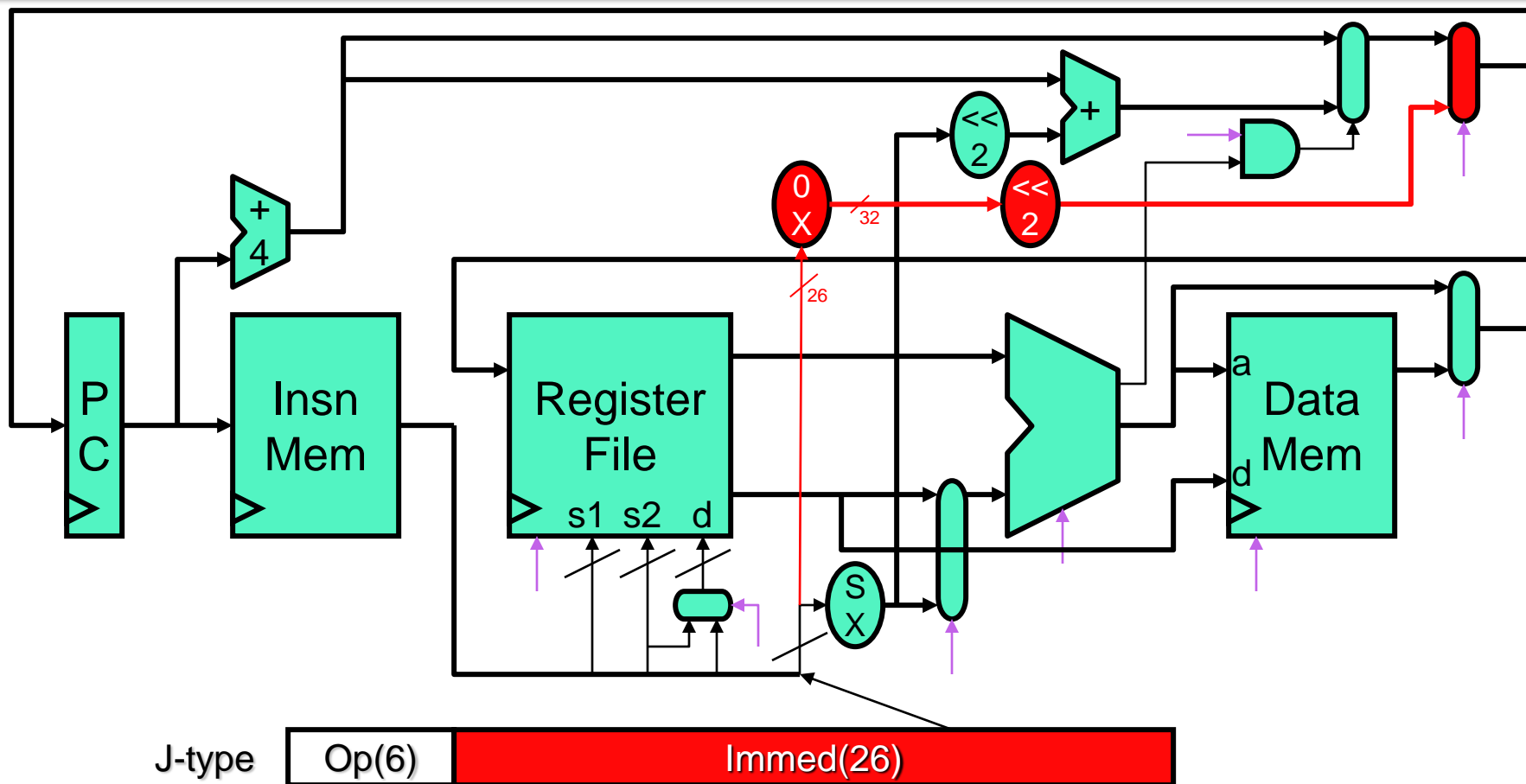
- Add path from second input register to data memory data input
- Disable RegFile's WE signal

# Fifth Instruction: beq \$1, \$2, target



- Add left shift unit (why?) and adder to compute PC-relative branch target
- Add mux to do what?

# Sixth Instruction: j



- Add shifter to compute left shift of 26-bit immediate
- Add additional PC input mux for jump target

# Seventh, Eight, Ninth Instructions

- Are these the paths we would need for all instructions?

**sll** \$1,\$2,4 // shift left logical

- Like an arithmetic operation, but need a shifter too

**slt** \$1,\$2,\$3 // set less than (slt)

- Like subtract, but need to write the condition bits, not the result
  - Need zero extension unit for condition bits
  - Need additional input to register write data mux

**jal** absolute\_target // jump and link

- Like a jump, but also need to write PC+4 into \$ra (\$31)
  - Need path from PC+4 adder to register write data mux
  - Need to be able to specify \$31 as an implicit destination

**jr** \$31 // jump register

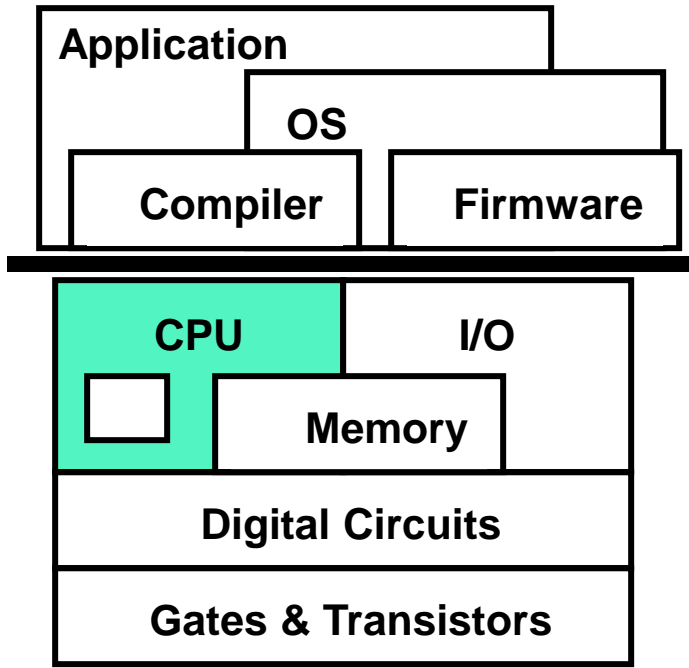
- Like a jump, but need path from register read to PC write mux

# Clock Timing

- Must deliver clock(s) to avoid races
- Can't write and read same value at same clock edge
  - Particularly a problem for RegFile and Memory
- May create multiple clock edges (from single input clock) by using buffers (to delay clock) and inverters
- For Homework 4 (the Duke 250/16 CPU):
  - Keep the clock SIMPLE and GLOBAL
  - You may need to do the PC on *falling* edge and everything else on *rising* edge
    - Changing clock edges in this way will separate PC++ from logic
    - Otherwise, if the PC changes *while* the operation is occurring, the instruction bits will change before the answer is computed -> *non-deterministic behavior* ☹
    - Note: A cheap way to make something trigger on the other clock edge is to NOT the clock on the way into that component

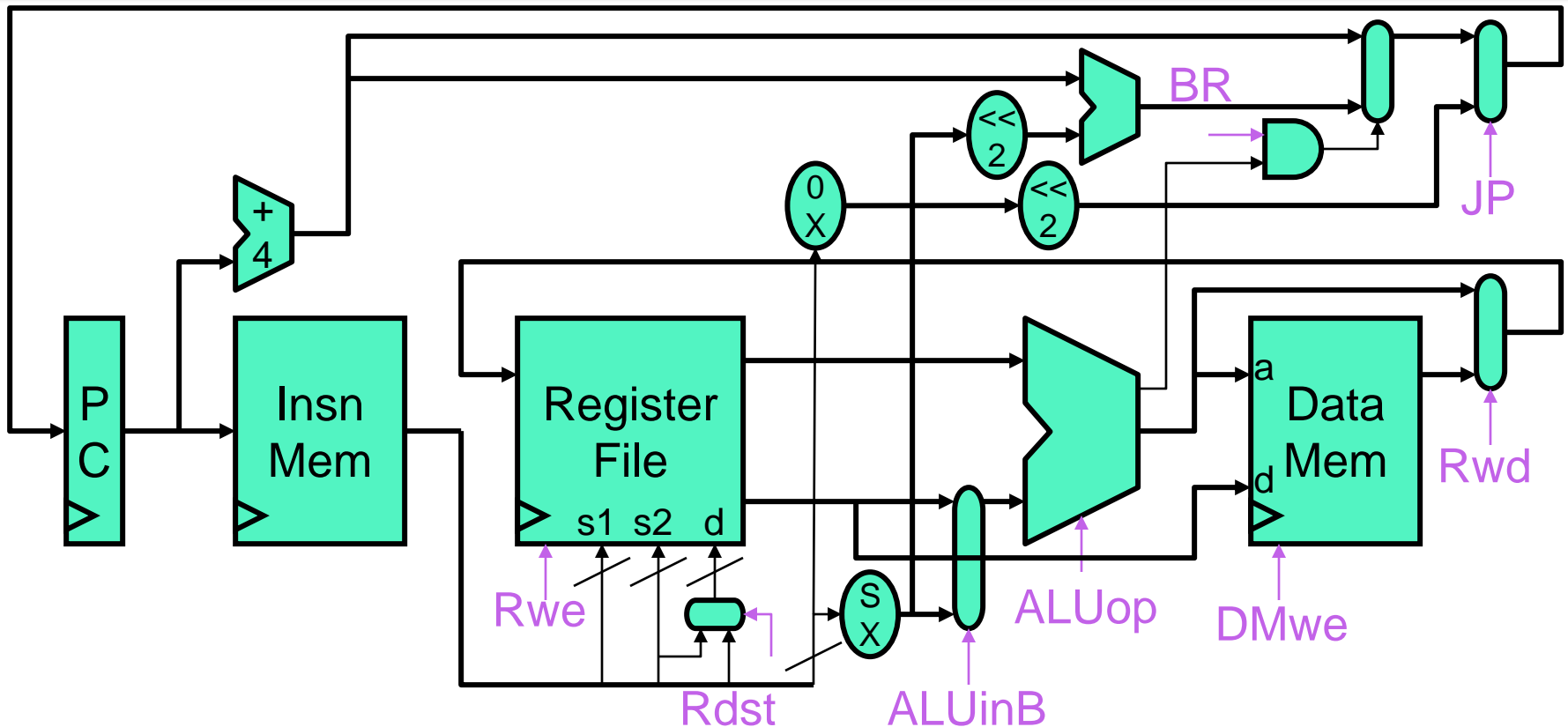


# This Unit: Processor Design



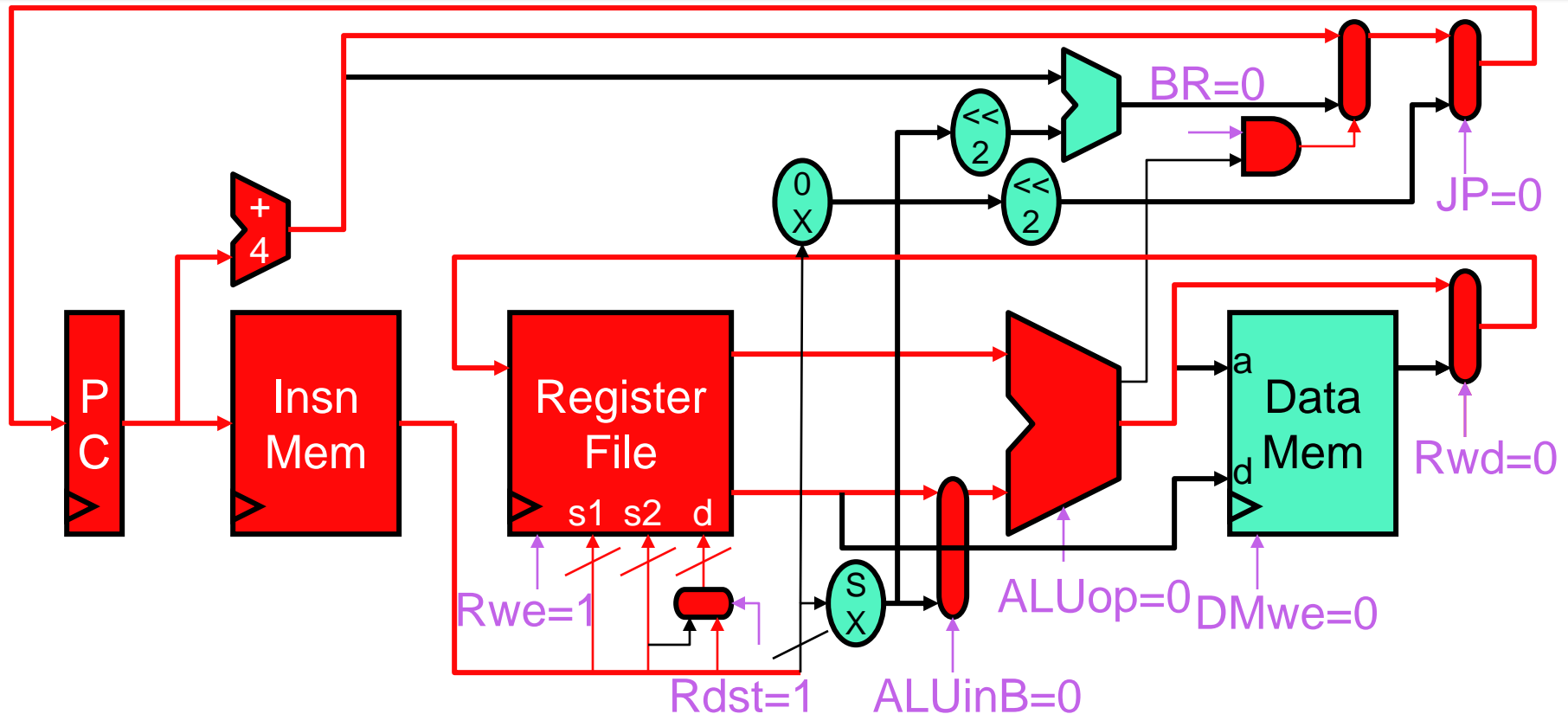
- Datapath components and timing
  - Registers and register files
  - Memories (RAMs)
  - Clocking strategies
- Mapping an ISA to a datapath
- **Control**
- Exceptions

# ↑ What Is Control? ↑



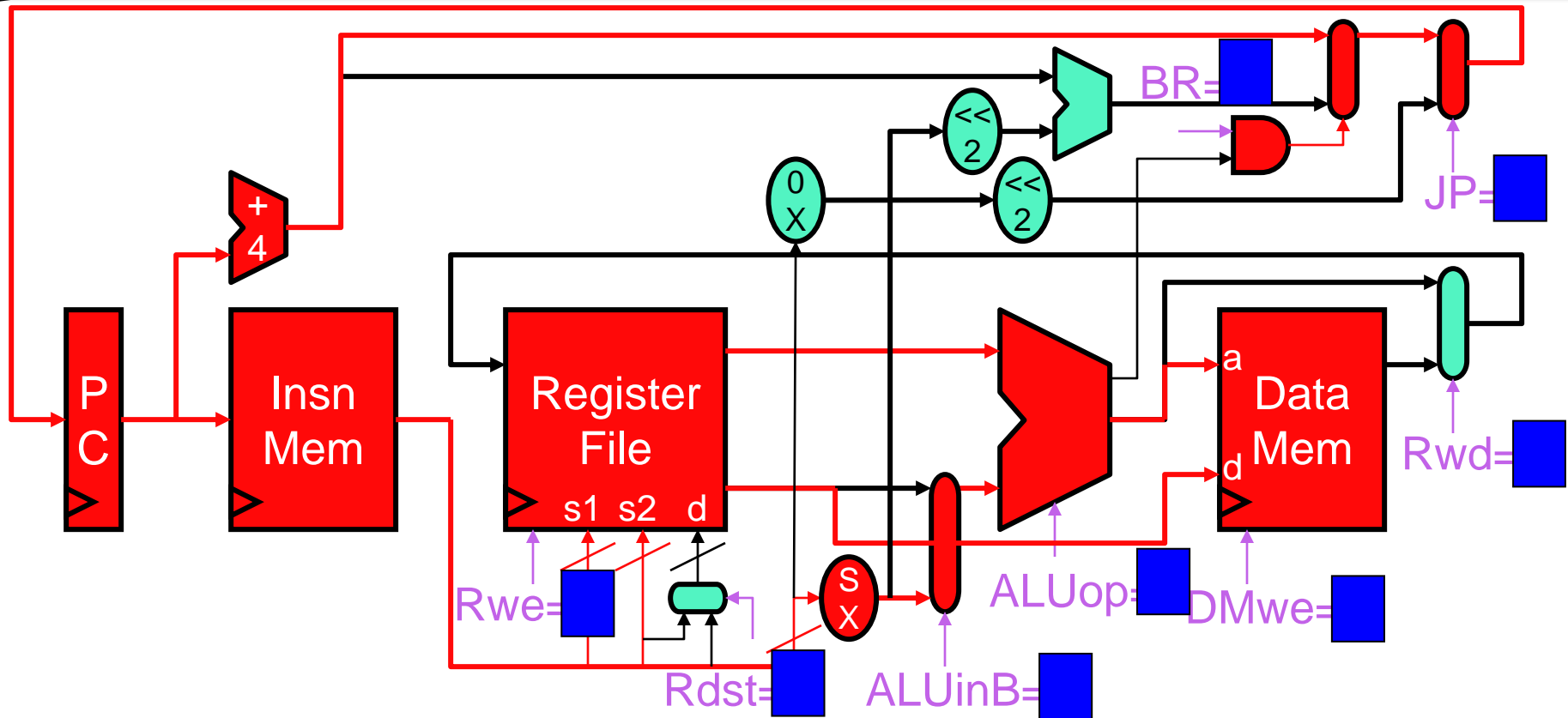
- 9 signals control flow of data through this datapath
  - MUX selectors, or register/memory write enable signals
  - Datapath of current microprocessor has 100s of control signals

# Example: Control for add



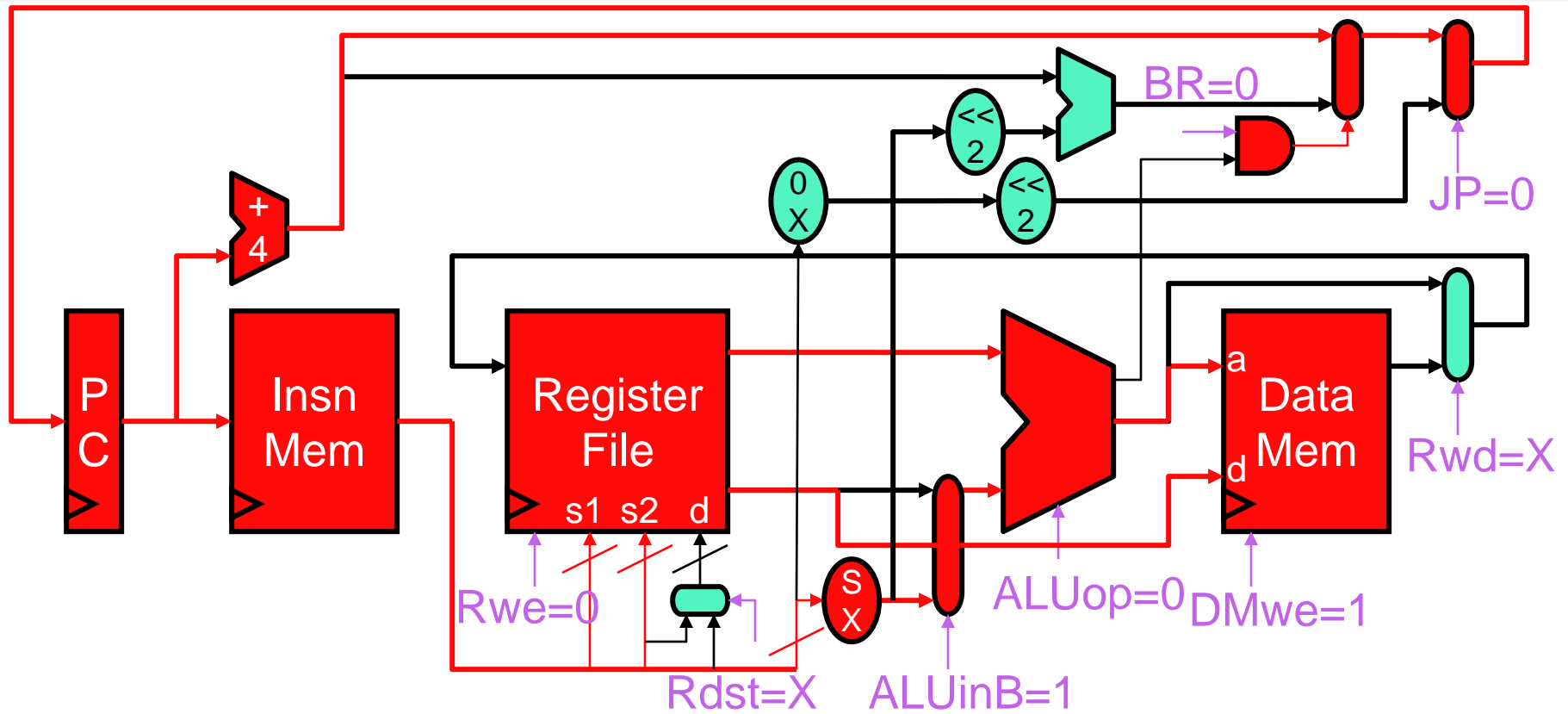
- **Rwe**: Register Write Enable
- **Rdst**: Register Destination chooser
- **ALUinB**: ALU input B chooser
- **ALUOp**: ALU operation (multi-bit)
- **DMwe**: Data Memory Write Enable
- **Rwd**: Register Write Data chooser
- **BR**: Branch?
- **JP**: Jump?

# Example: Control for sw



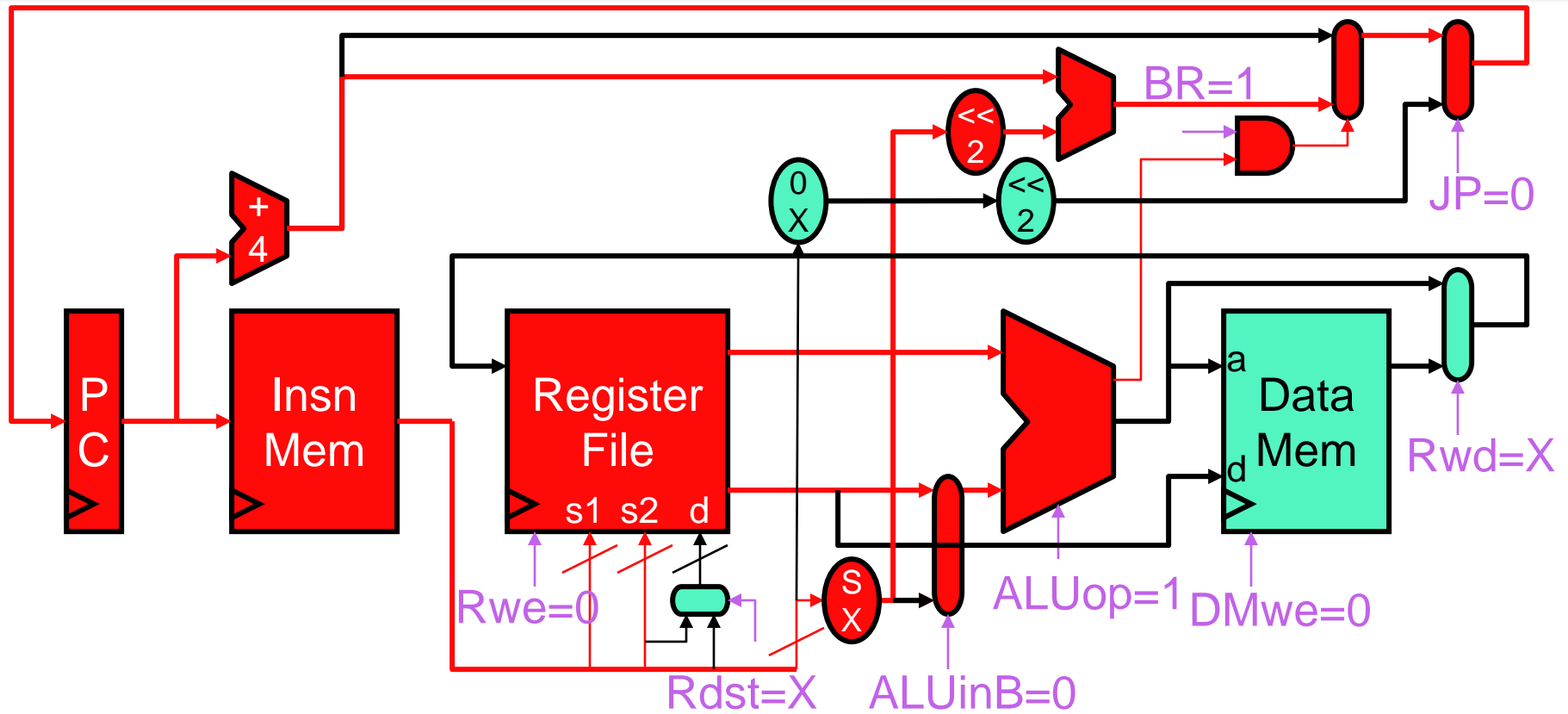
- Difference between a `sw` and an `add` is 5 signals
  - 3 if you don't count the X ("don't care") signals

# Example: Control for sw



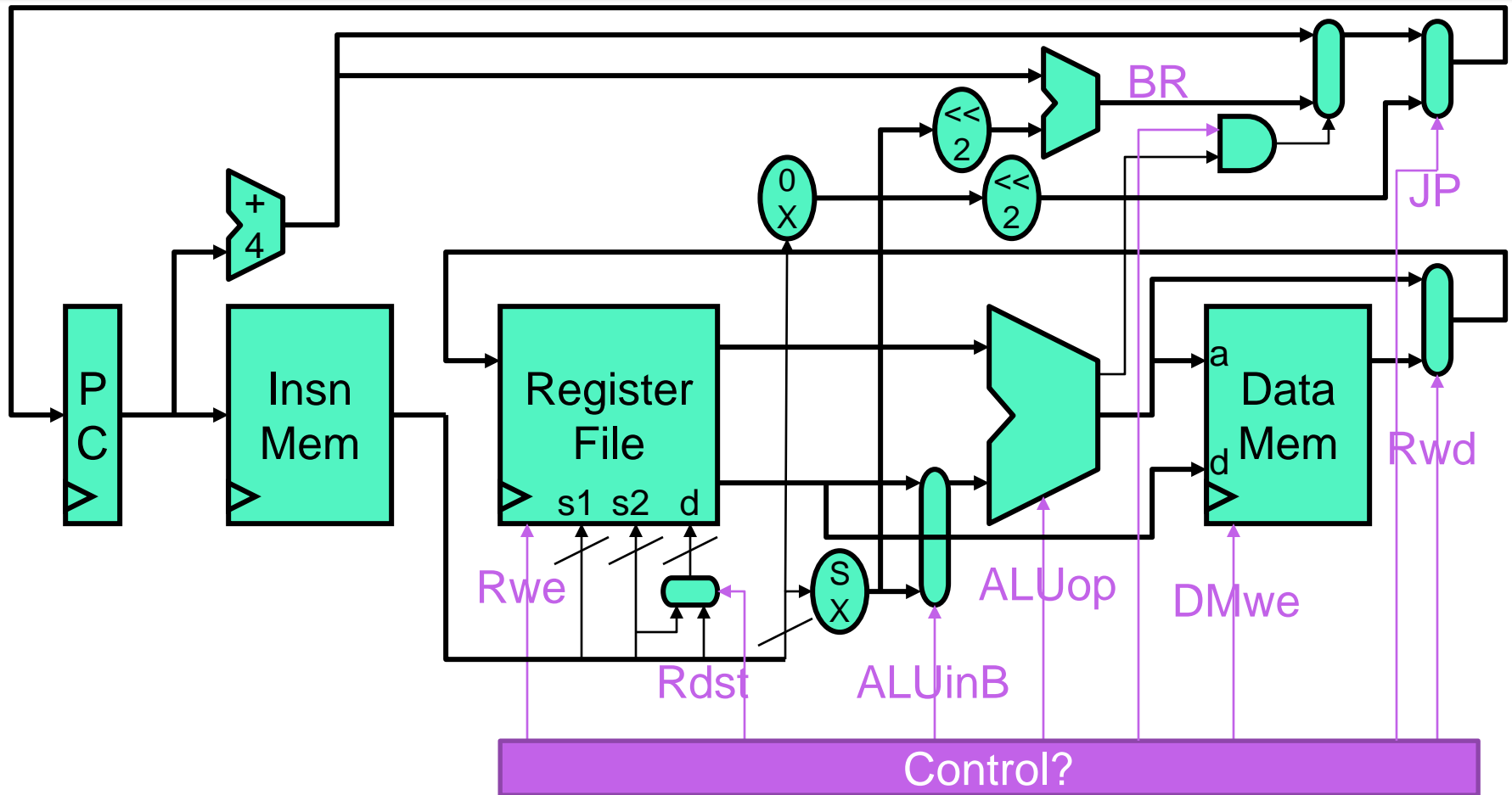
- Difference between a sw and an add is 5 signals
  - 3 if you don't count the X ("don't care") signals

# Example: Control for beq \$1, \$2, target



- Difference between a store and a branch is only 4 signals

# How Is Control Implemented?



# Implementing Control

- Each instruction has a unique set of control signals
  - Most signals are function of opcode
  - Some may be encoded in the instruction itself
    - E.g., the ALUop signal is some portion of the MIPS Func field
  - + Simplifies controller implementation
  - Requires careful ISA design
- Options for implementing control
  1. Use instruction type to look up control signals in a table
  2. Design combinational logic whose outputs are control signals
  - Either way, goal is same: turn instruction into control signals



# Control Implementation: ROM

- **ROM (read only memory)**: like a RAM but unwritable
  - Bits in data words are control signals
  - Lines indexed by opcode
- Example: ROM control for our simple datapath

		BR	JP	ALUinB	ALUop	DMwe	Rwe	Rdst	Rwd
→	add	0	0	0	0	0	1	1	0
→	addi	0	0	1	0	0	1	0	0
→	lw	0	0	1	0	0	1	0	1
→	sw	0	0	1	0	1	0	0	0
→	beq	1	0	0	1	0	0	0	0
→	j	0	1	0	0	0	0	0	0

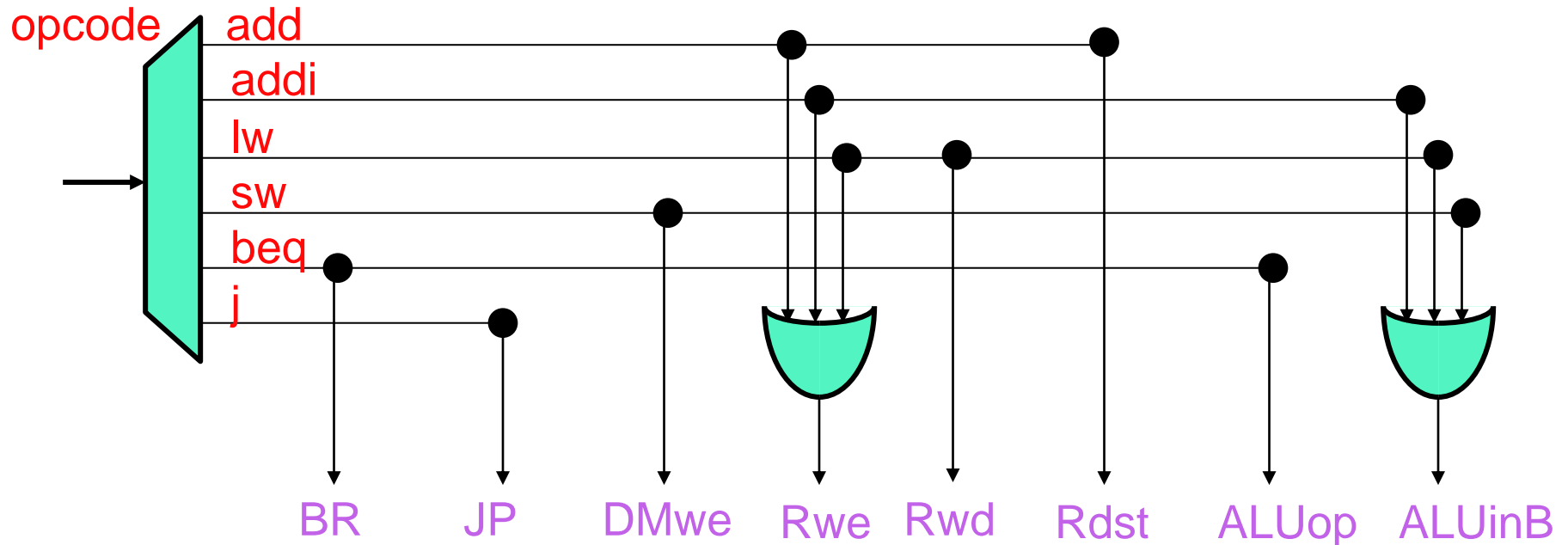
opcode →

# ROM vs. Combinational Logic

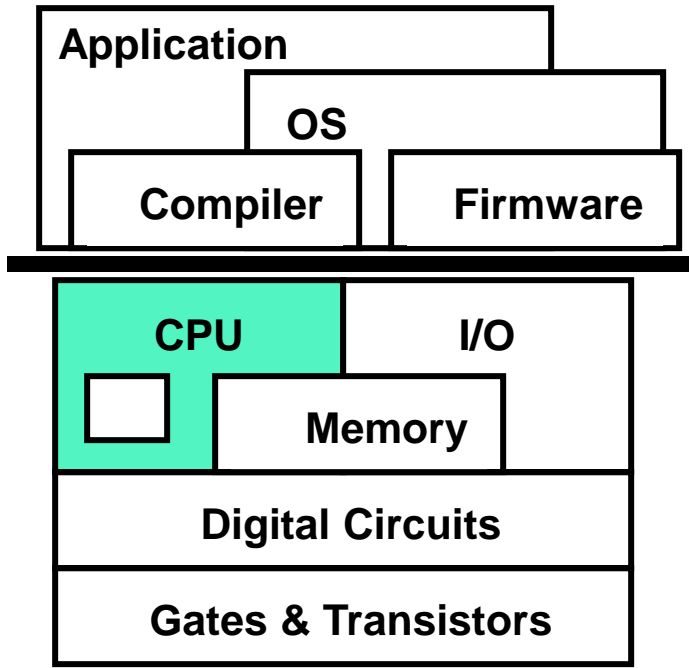
- A control ROM is fine for 6 insns and 9 control signals
- A real machine has 100+ insns and 300+ control signals
  - Even “RISC”s have lots of instructions
  - 30,000+ control bits (~4KB)
    - Not huge, but hard to make fast
      - Control must be faster than datapath
- Alternative: **combinational logic**
  - It’s that thing we know how to do! *Nice!*
  - Exploits observation: many signals have few 1s or few 0s

# Control Implementation Combinational Logic with a Decoder (one-hot representation)

- Example: combinational logic control for our simple datapath



# This Unit: Processor Design



- Datapath components and timing
  - Registers and register files
  - Memories (RAMs)
  - Clocking strategies
- Mapping an ISA to a datapath
- Control
- Exceptions

# Exceptions

- **Exceptions and interrupts**

- Infrequent (exceptional!) events
  - I/O, divide-by-0, illegal instruction, page fault, protection fault, ctrl-C, ctrl-Z, timer
- Handling requires intervention from operating system
  - End program: divide-by-0, protection fault, illegal insn, ^C
  - Fix and restart program: I/O, page fault, ^Z, timer
- Handling should be transparent to application code
  - Don't want to (can't) constantly check for these using insns
  - Want "Fix and restart" equivalent to "never happened"

# Exception Handling

- What does exception handling look like to software?
  - When exception happens...
  - Control transfers to OS at pre-specified exception handler address
  - OS has privileged access to registers user processes do not see
    - These registers hold information about exception
    - Cause of exception (e.g., page fault, arithmetic overflow)
    - Other exception info (e.g., address that caused page fault)
    - PC of application insn to return to after exception is fixed
  - OS uses privileged (and non-privileged) registers to do its “thing”
  - OS returns control to user application
- Same mechanism available programmatically via SYSCALL

# MIPS Exception Handling

- MIPS uses registers to hold state during exception handling
  - These registers live on “coprocessor 0”
  - \$14: EPC (holds PC of user program during exception handling)
  - \$13: exception type (SYSCALL, overflow, etc.)
  - \$8: virtual address (that produced page/protection fault)
  - \$12: exception mask (which exceptions trigger OS)
- Exception registers accessed using two **privileged** instructions **mfc0**, **mtc0**
  - Privileged = user process can’t execute them
  - mfc0: move (register) from coprocessor 0 (to user reg)
  - mtc0: move (register) to coprocessor 0 (from user reg)
- Privileged instruction **rfe** restores user mode
  - Kernel executes this instruction to restore user program

# MIPS Exception Handling

- MIPS uses registers to hold state during exception handling
  - These registers live on “coprocessor 0”
  - \$14: EPC (holds PC of user program during exception handling)
  - \$13: exception type (SYSCALL, overflow, etc.)
  - \$8: virtual address (that produced exception)
  - \$12: exception mask (what exceptions are enabled)
- Exception instructions
  - `li` (load immediate) – load into user register
  - `sw` (store word) – store into coprocessor 0 (to user reg)
  - `mfhc` (move from coprocessor 0) – move (register) to coprocessor 0 (from user reg)
- Privileged instruction `rfw` restores user mode
  - Kernel executes this instruction to restore user program

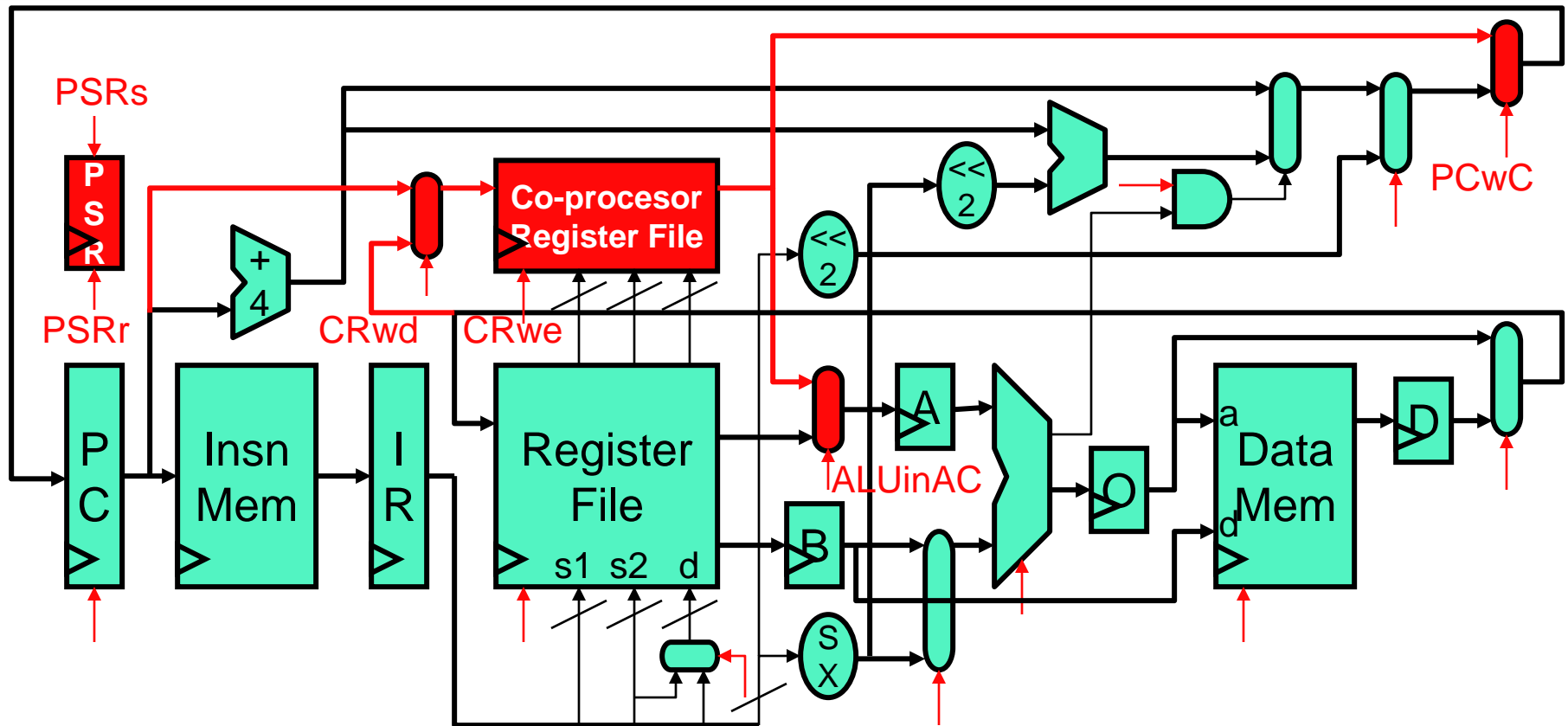
**DON'T GET TOO OBSESSED ABOUT HOW EXACTLY MIPS DOES THIS – FOCUS ON THE BIG PICTURE AND WHAT MUST HAPPEN IN GENERAL**



# Implementing Exceptions

- Why do architects care about exceptions?
  - Because we use datapath and control to implement them
  - More precisely... to implement aspects of exception handling
    - Recognition of exceptions
    - Transfer of control to OS
    - Privileged OS mode
- Later in semester, we'll talk more about exceptions (b/c we need them for I/O)

# Datapath with Support for Exceptions



- Co-processor register (CR) file needn't be implemented as RF
  - Independent registers connected directly to pertinent muxes
- **PSR (processor status register):** in privileged mode?

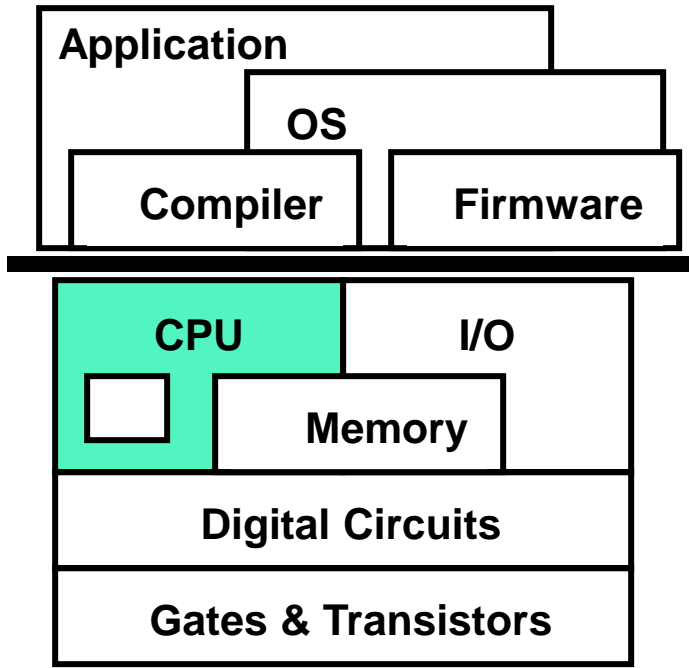
# Summary

- We now know how to build a fully functional processor
- But ...
  - We're still treating memory as a black box (actually two green boxes, to be precise)
  - Our fully functional processor is slow. Really, really slow.

# “Single-Cycle” Performance

- Useful metric: cycles per instruction (CPI)
- + Easy to calculate for single-cycle processor:  $CPI = 1$ 
  - Seconds/program = (insns/program) \* 1 CPI \* (N seconds/cycle)
  - ICQ: How many cycles/second in 3.8 GHz processor?
- Slow!
  - Clock period must be elongated to accommodate longest operation
    - In our datapath: lw
    - Goes through five structures in series: insn mem, register file (read), ALU, data mem, register file again (write)
  - No one will buy a machine with a slow clock
    - Not even your grandparents!
    - Biggest issue: data memory itself is sloooooooooooooooooooooooooooooow
- Next up: Speed up data memory!
- Later on: Faster processor cores!

# This Unit: Processor Design



- Datapath components and timing
  - Registers and register files
  - Memories (RAMs)
  - Clocking strategies
- Mapping an ISA to a datapath
- Control

**Next up: Memory Systems**