

ECE 590.03 Fundamentals of Computer Systems and Engineering

Datapaths

Admin

- Homework
 - Homework 3: out this weekend, assembly
 - Going forwards: no **process** or **variable**
- Reading:
 - Chapter 4 (4.1-4.4 for now)
 - (Maybe review 1.4)

What did we do last time?

- Who can remind us what we did last time?

When last I saw you all..

- MIPS Assembly
 - Functions: call (jal), return (jr)
 - Calling convention
 - Saving/restoring on the stack

Now confluence of MIPS + digital logic

- Start of semester: Digital Logic
 - Building blocks of digital design
- Most recently: MIPS assembly, ISA
 - Lowest level software
- Now: where they meet...
 - Datapaths: hardware implementation of processors
 - By the way: homework 4 = build a datapath
 - With some components from the TAs...

Datapath for MIPS ISA

- Consider only the following instructions

`add $1,$2,$3`

`addi $1,2,$3`

`lw $1,4($3)`

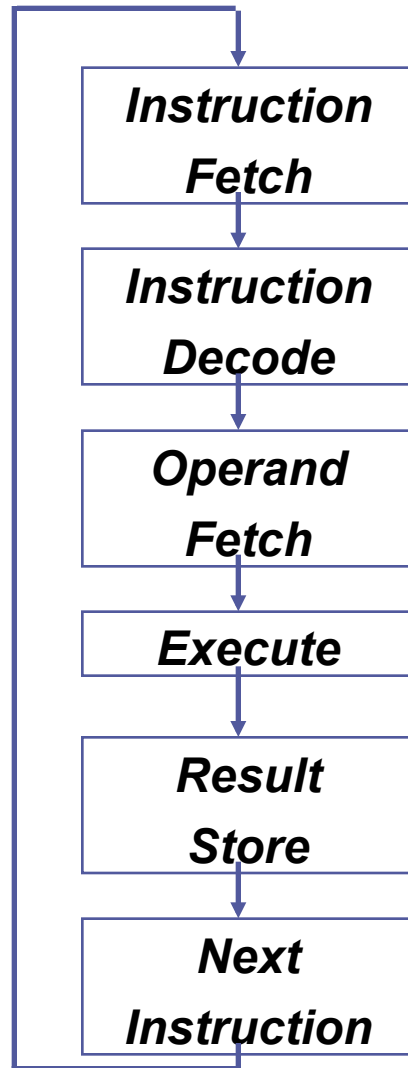
`sw $1,4($3)`

`beq $1,$2,PC_relative_target`

`j absolute_target`

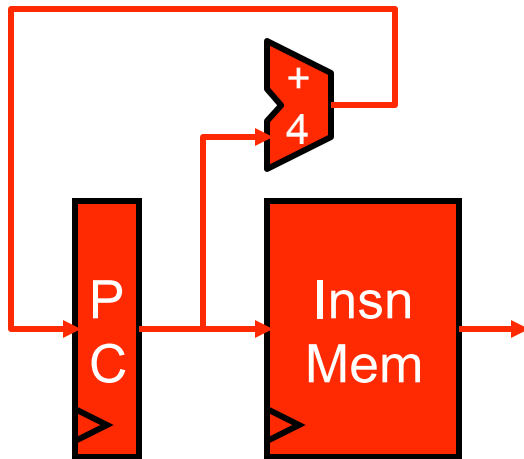
- Why only these?
 - Most other instructions are the same from datapath viewpoint
 - The one's that aren't are left for you to figure out

Remember The von Neumann Model?



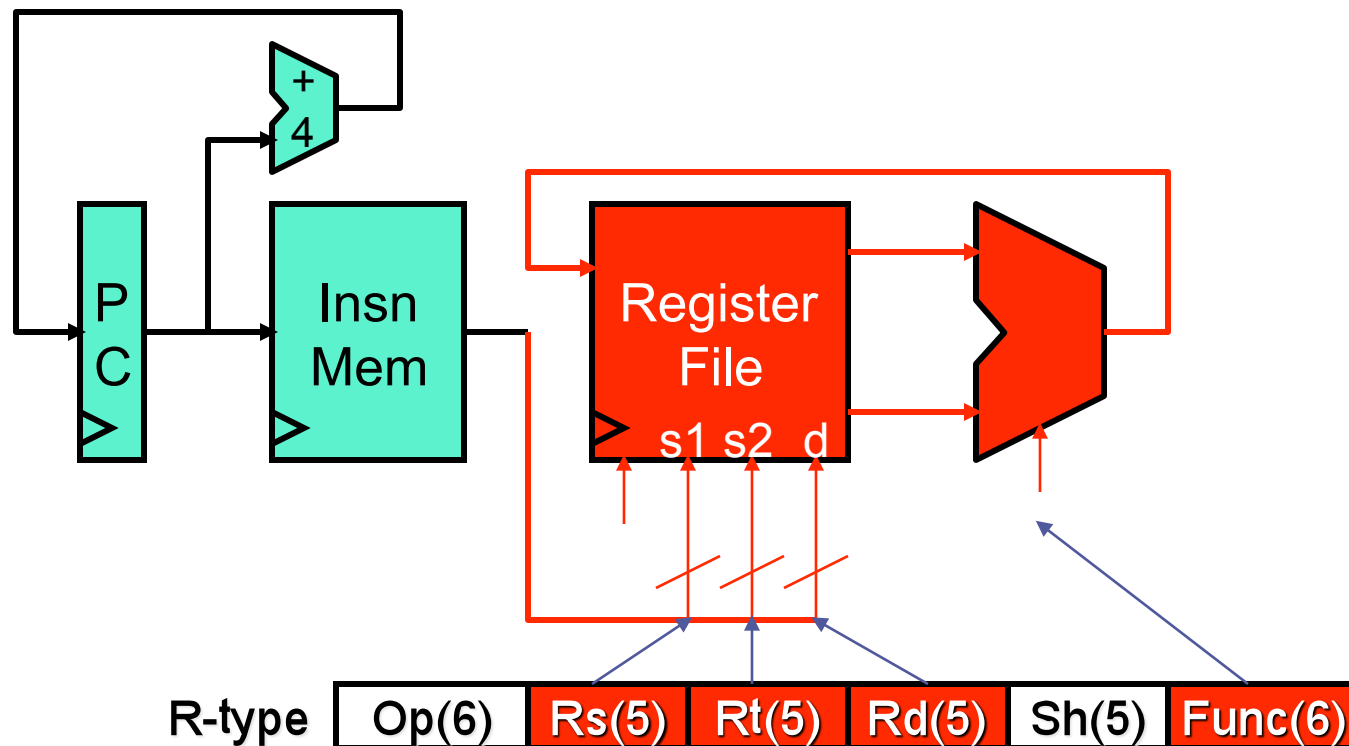
- **Instruction Fetch:**
Read instruction bits from memory
- **Decode:**
Figure out what those bits mean
- **Operand Fetch:**
Read registers (+ mem to get sources)
- **Execute:**
Do the actual operation (e.g., add the #s)
- **Result Store:**
Write result to register or memory
- **Next Instruction:**
Figure out mem addr of next insn, repeat

Start With Fetch



- Same for all instructions (don't know insn yet)
- PC and instruction memory
- A +4 incrementer computes default next instruction PC
 - Details of Insn Mem: later...

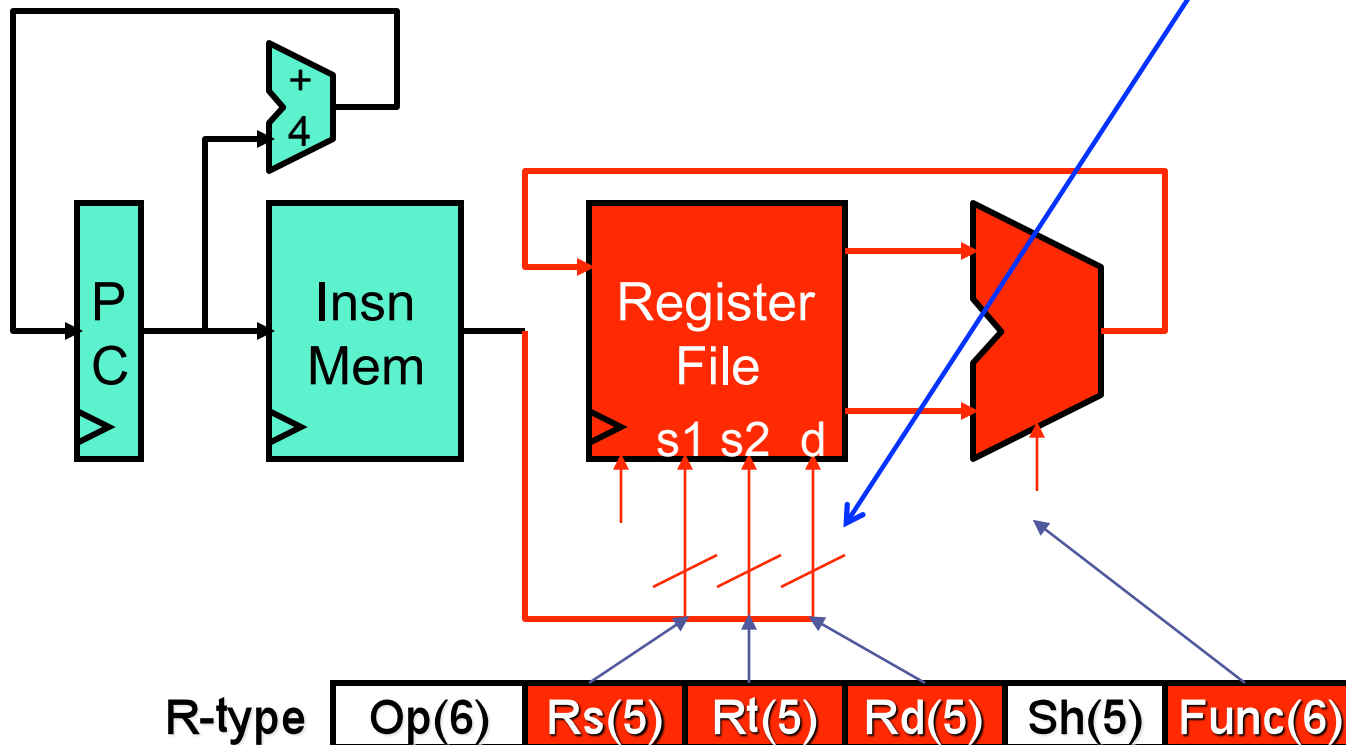
First Instruction: add



- Add register file and ALU

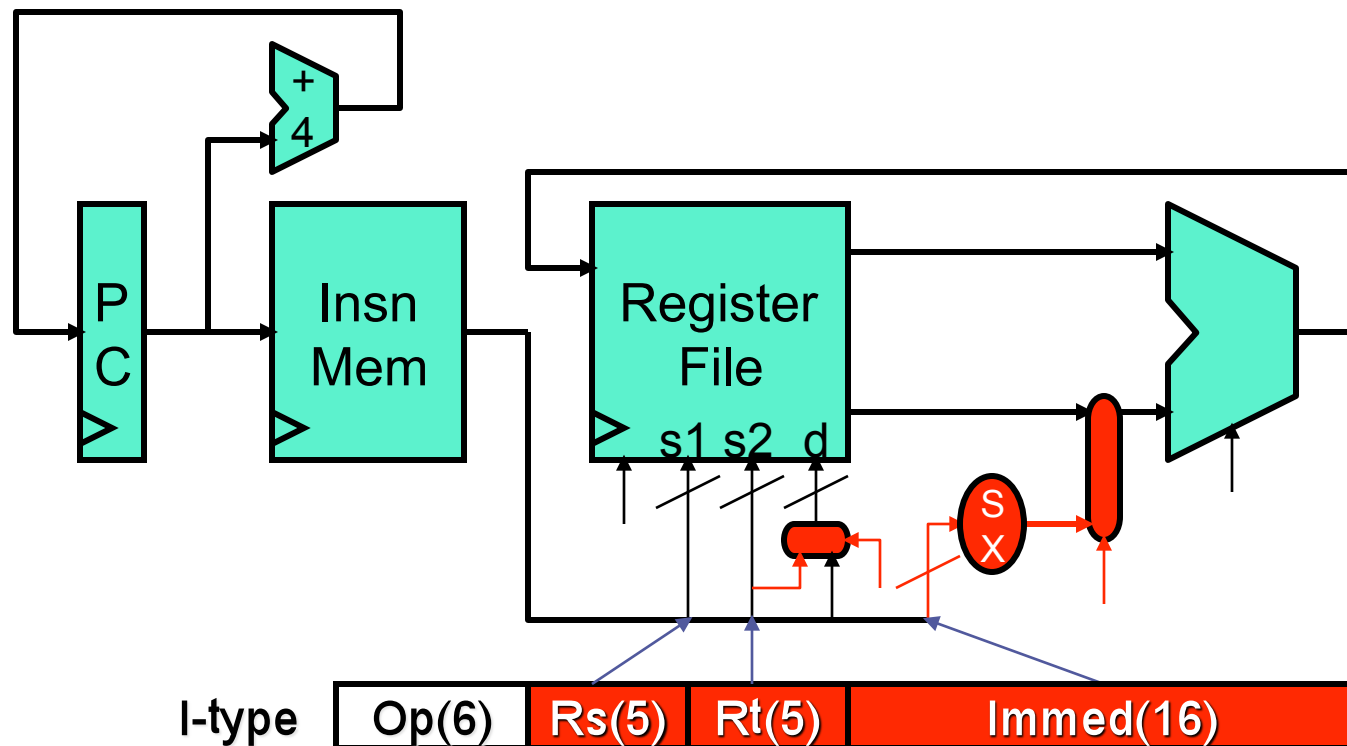
First Instruction: add

Decoding: Very easy in MIPS



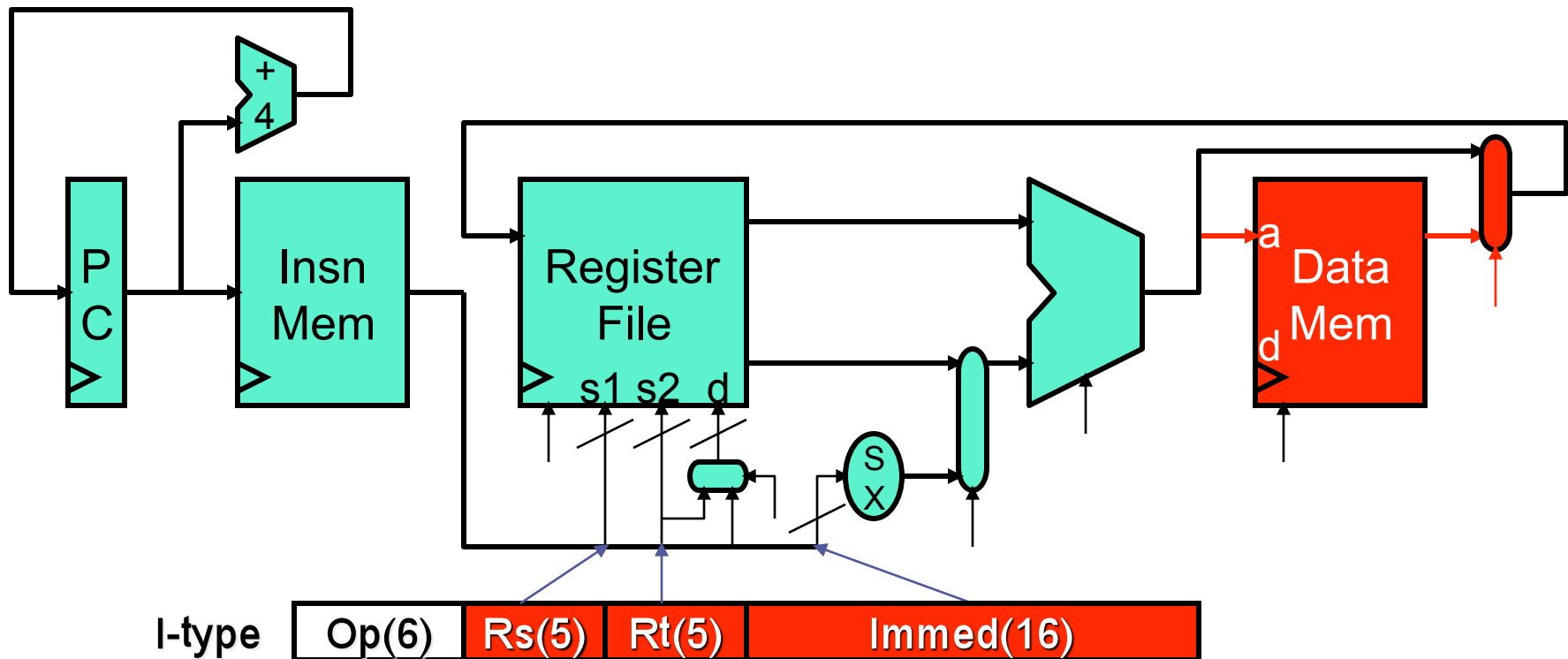
- Add register file and ALU

Second Instruction: addi



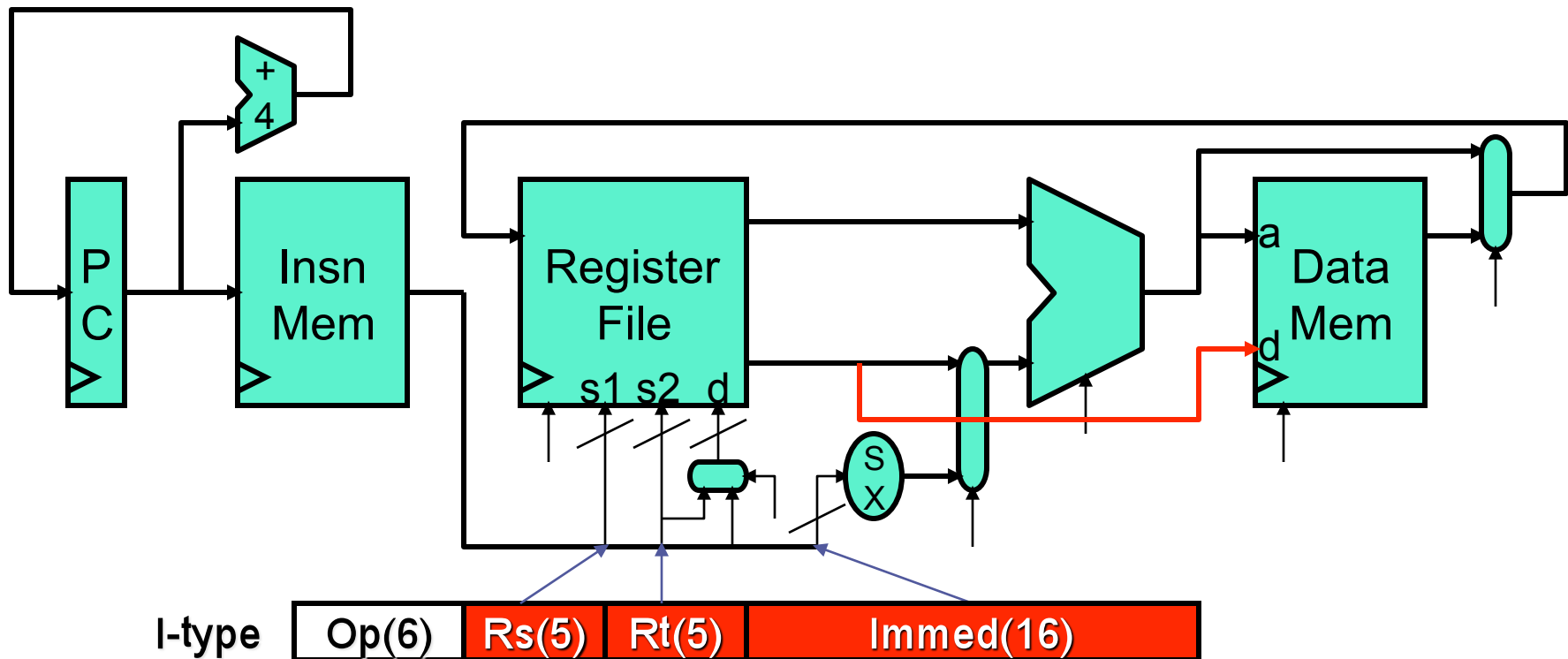
- Destination register can now be either Rd or Rt
- Add sign extension unit and mux into second ALU input

Third Instruction: lw



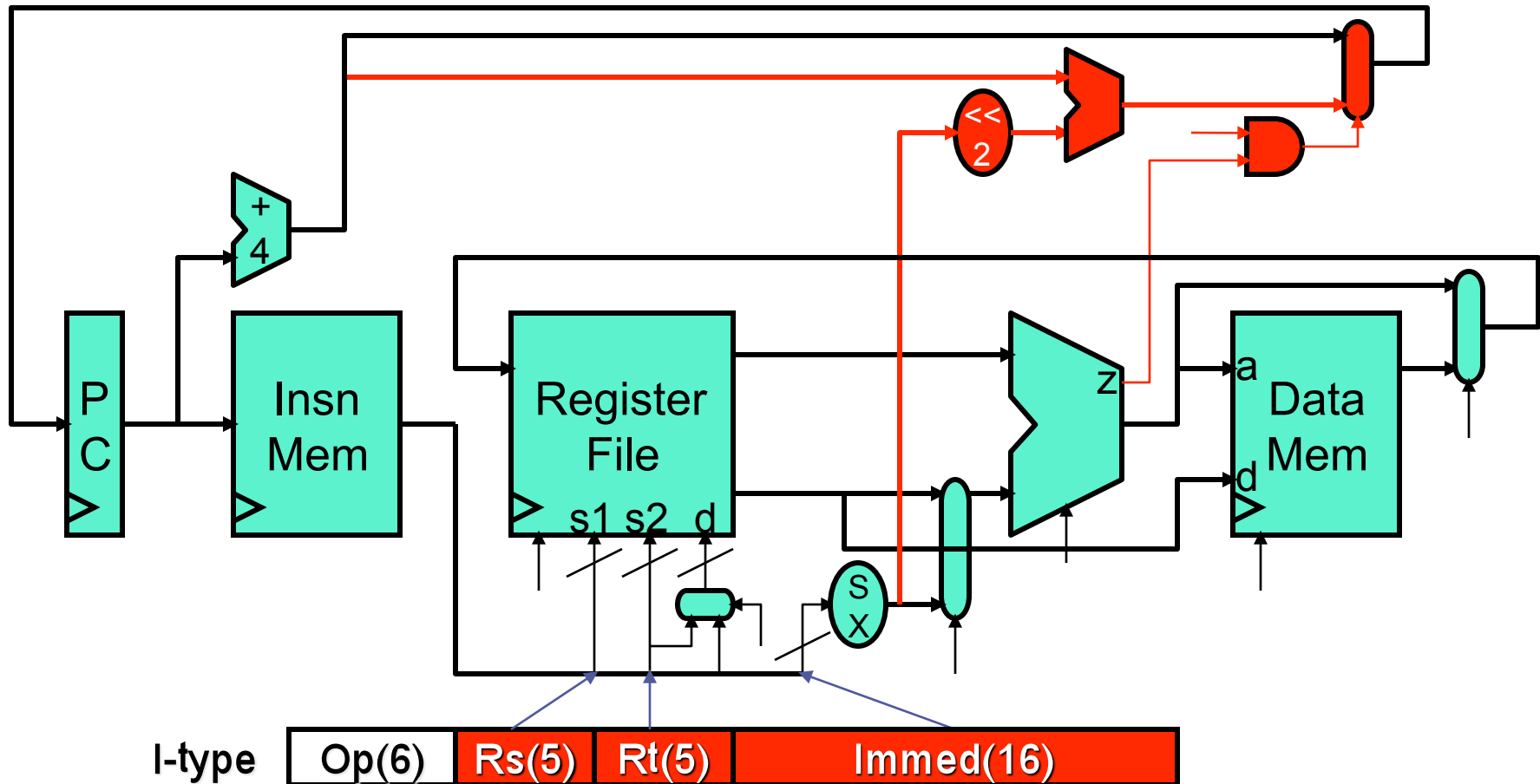
- Add data memory, address is ALU output
- Add register write data mux to select memory output or ALU output

Fourth Instruction: sw



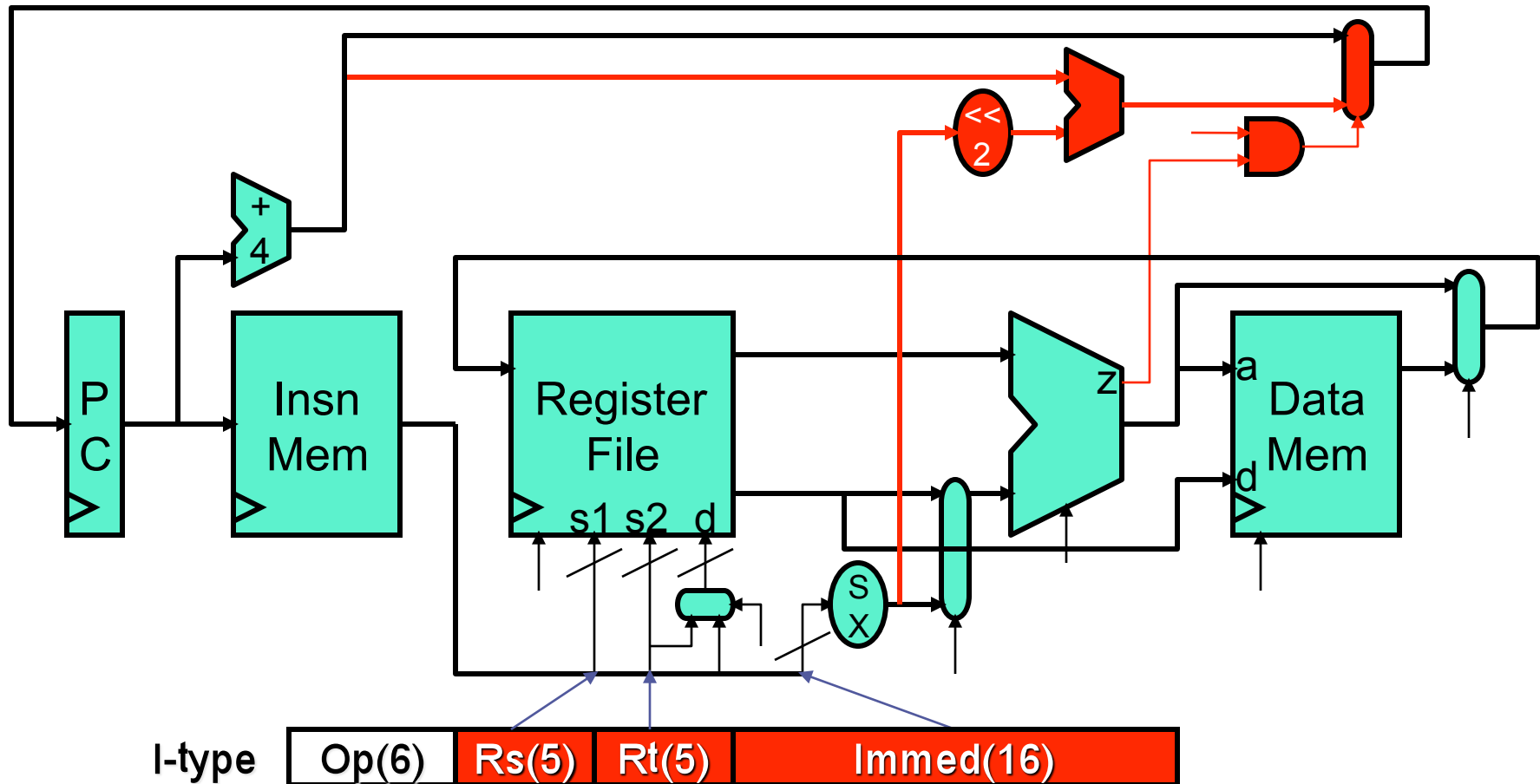
- Add path from second input register to data memory data input

Fifth Instruction: beq



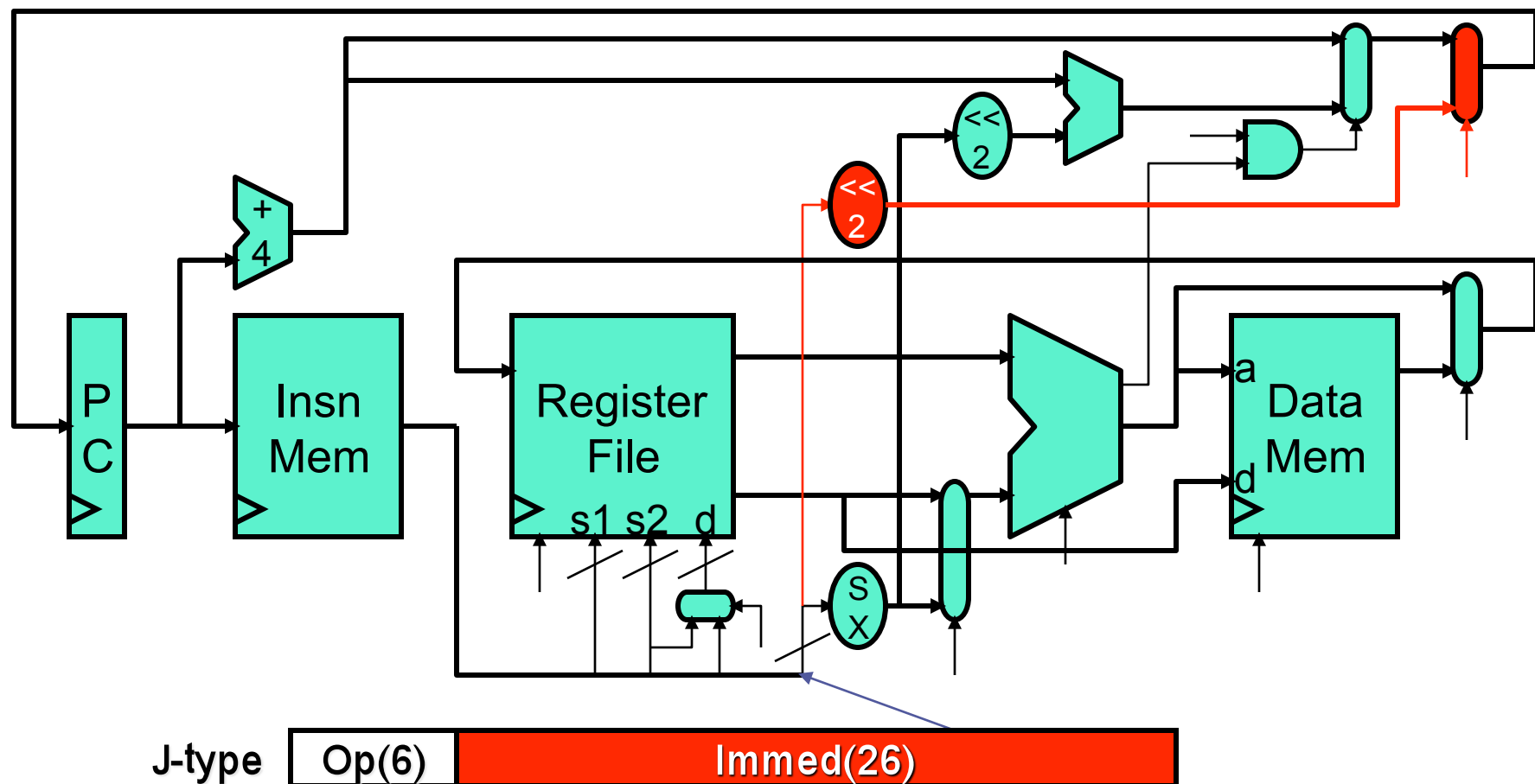
- Add left shift unit and adder to compute PC-relative branch target
- Add PC input mux to select PC+4 or branch target

Fifth Instruction: beq



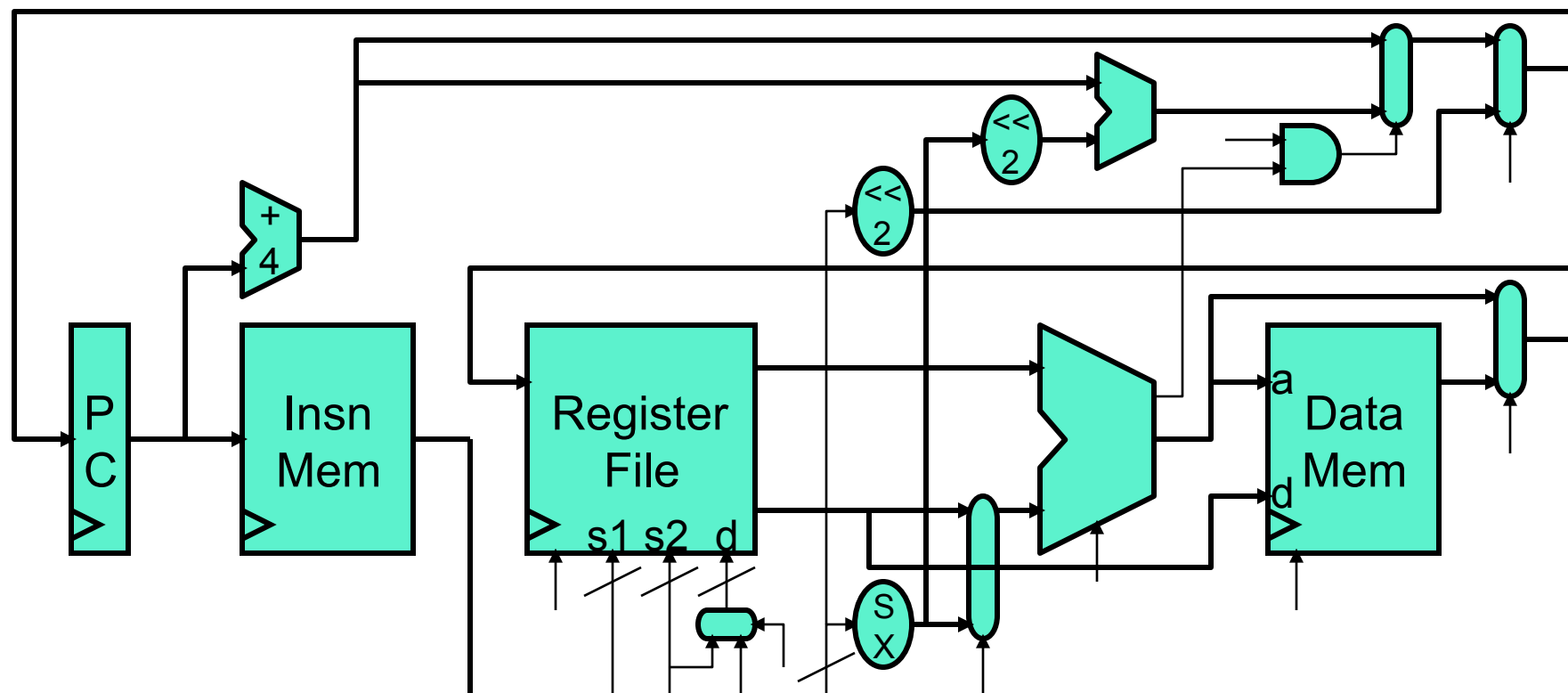
- Add left shift unit and adder to compute PC-relative branch target
- Note: shift by fixed amount very simple

Sixth Instruction: j



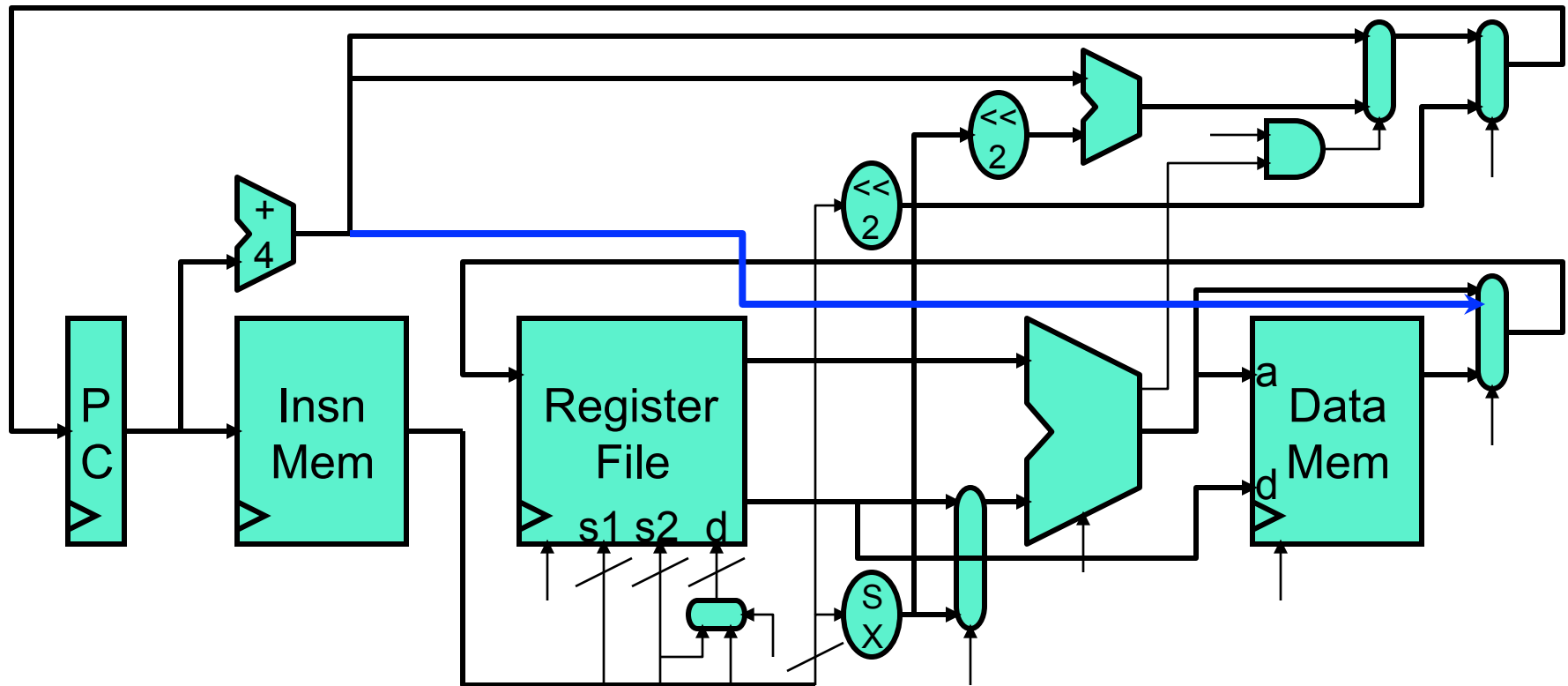
- Add shifter to compute left shift of 26-bit immediate
- Add additional PC input mux for jump target

More Instructions...



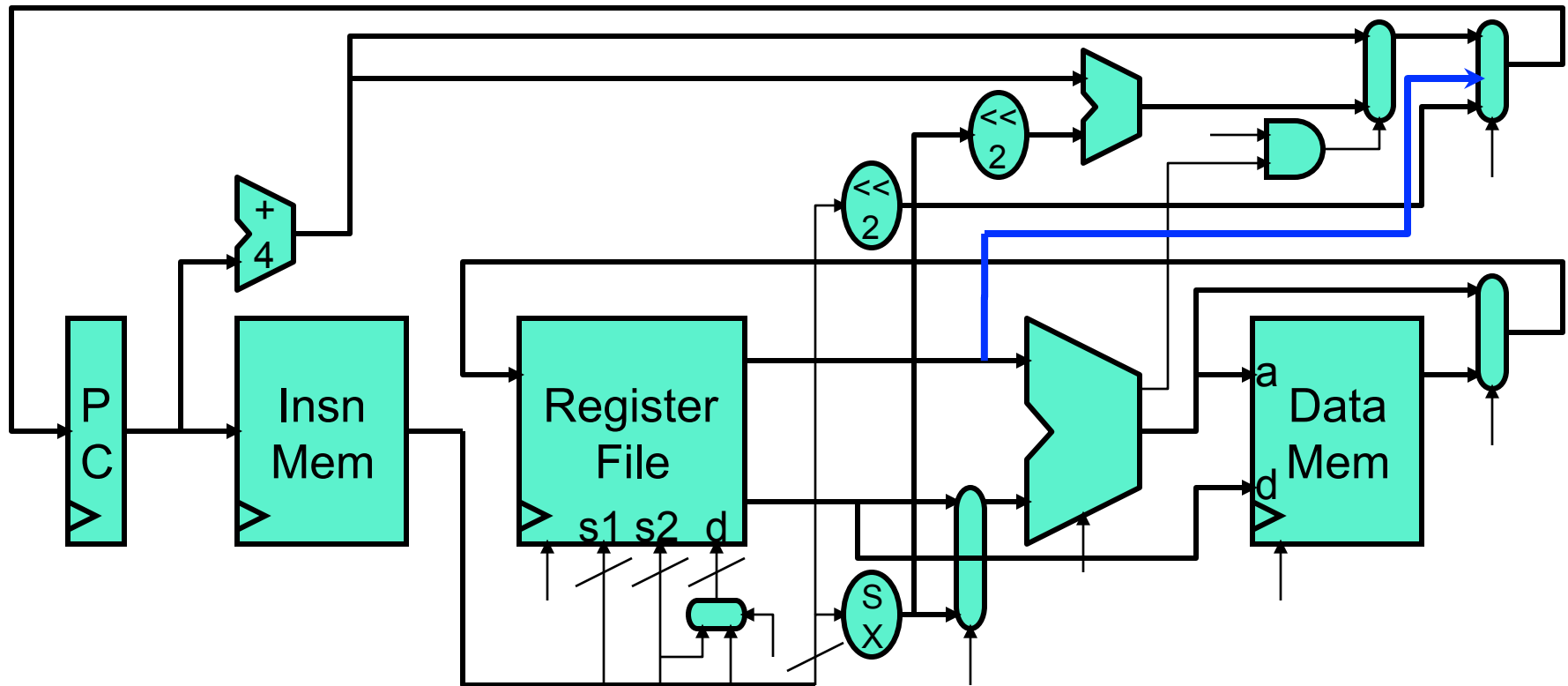
- Figure out datapath modifications for
 - jal (J-type)
 - jr (R-type)

Jal



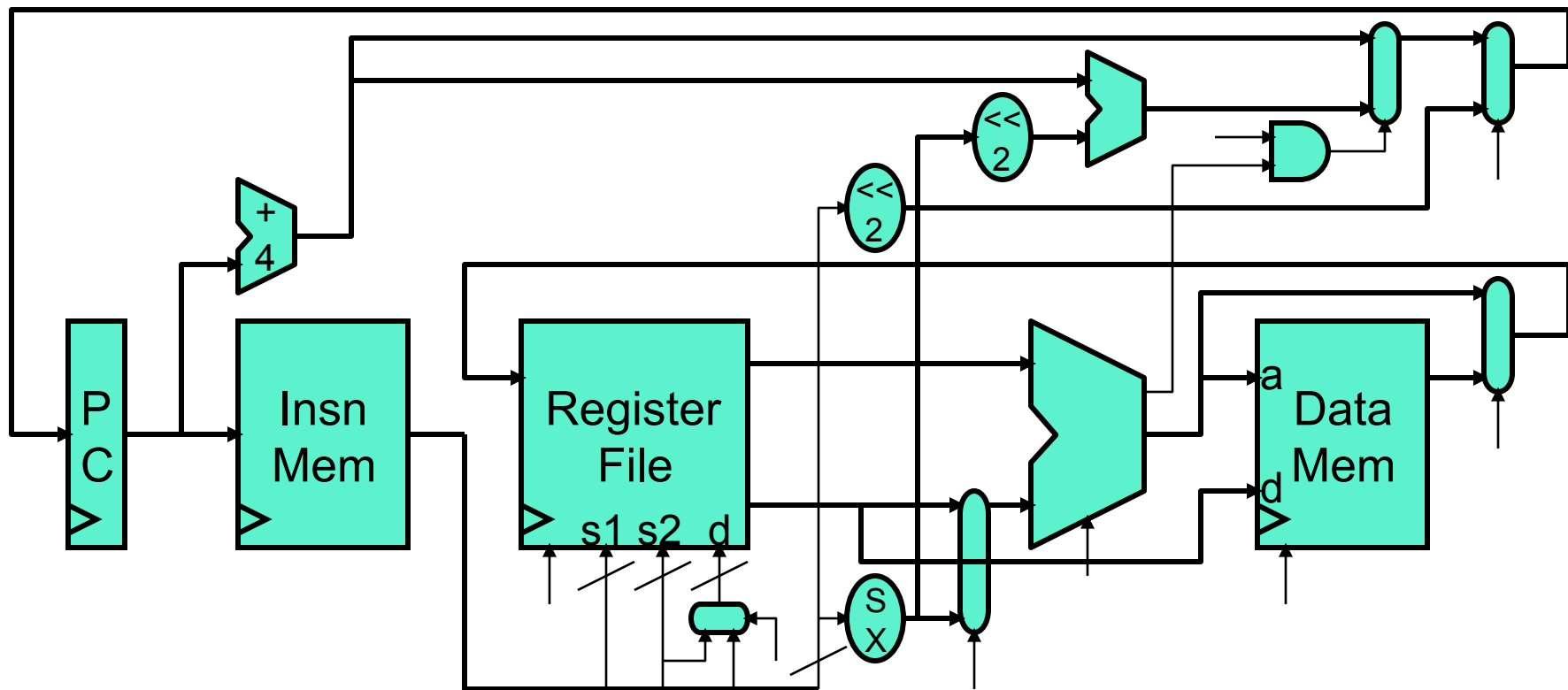
- For jal, need to get PC+4 to RF write mux

JR



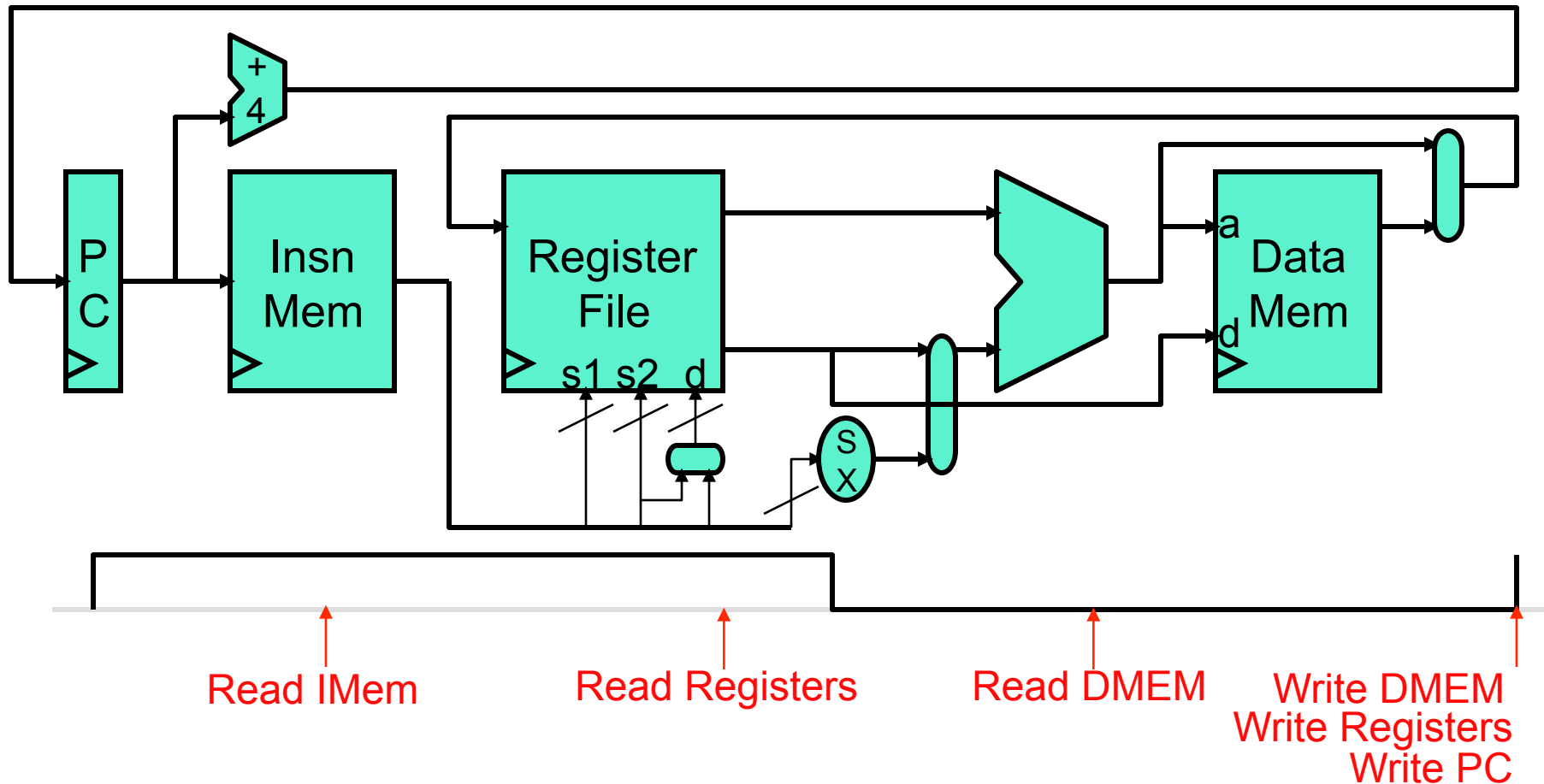
- For JR need to get RF read value to next PC mux

Good practice: Try other insns



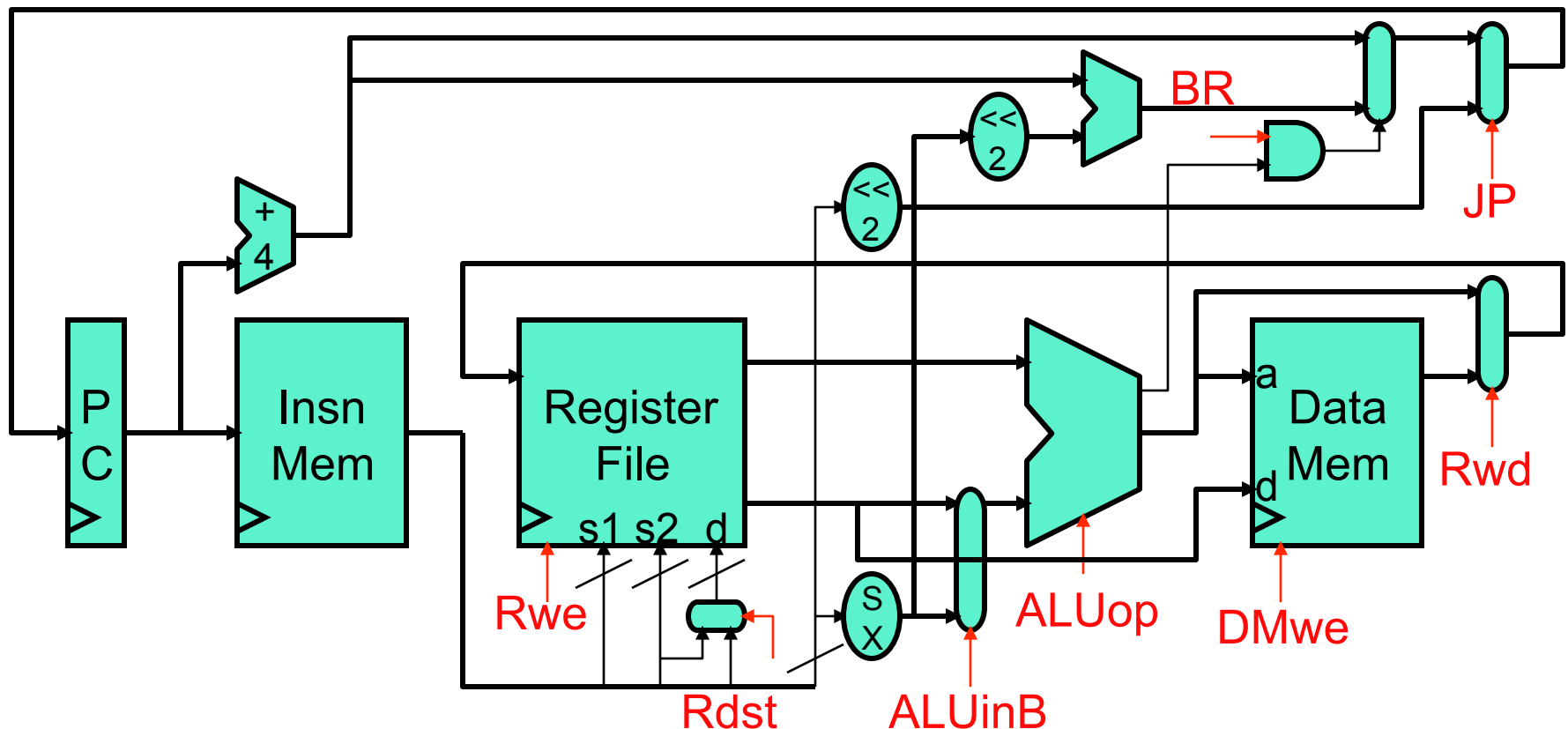
- Pick other MIPS instructions, contemplate how to add

“Continuous Read” Datapath Timing



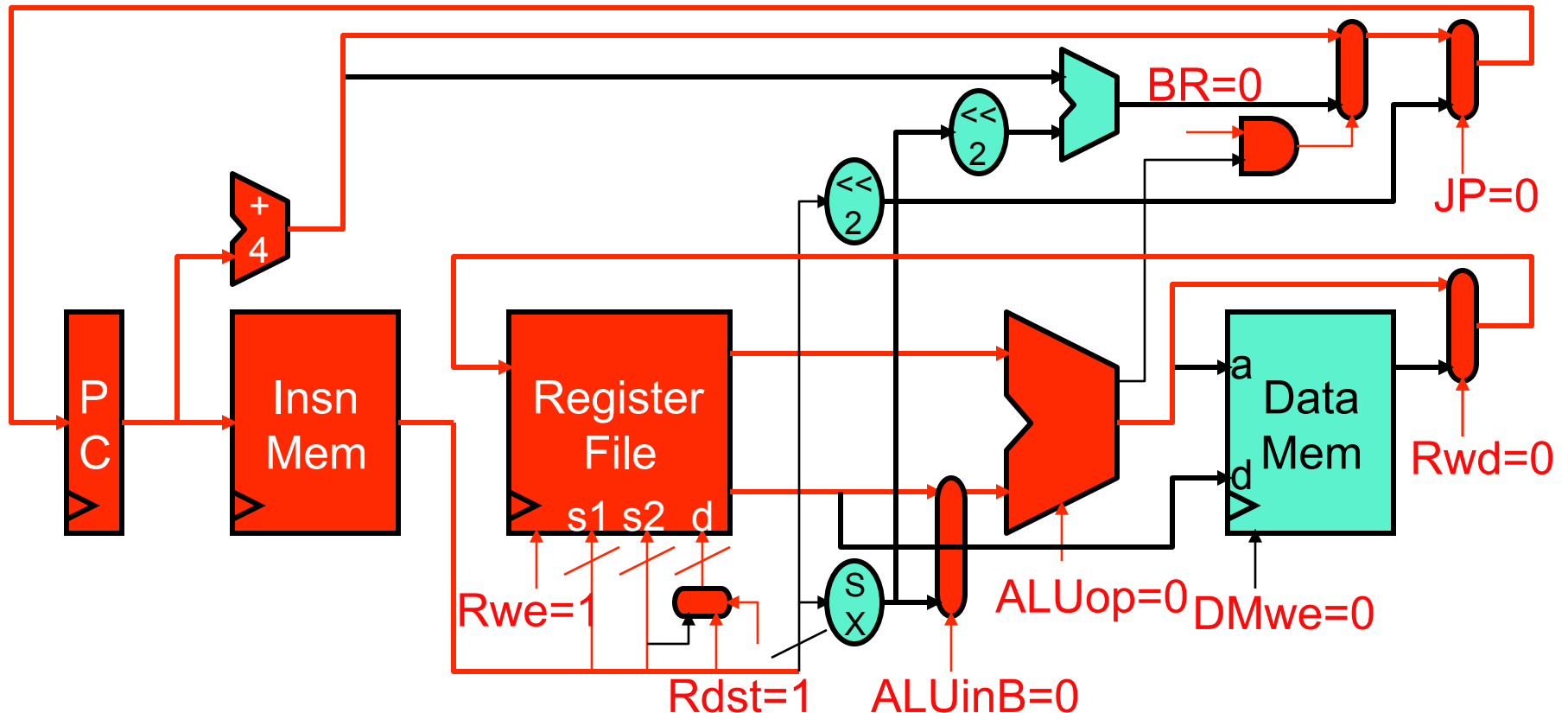
- Works because writes (PC, RegFile, DMem) are independent
- And because no read logically follows any write

What Is Control?



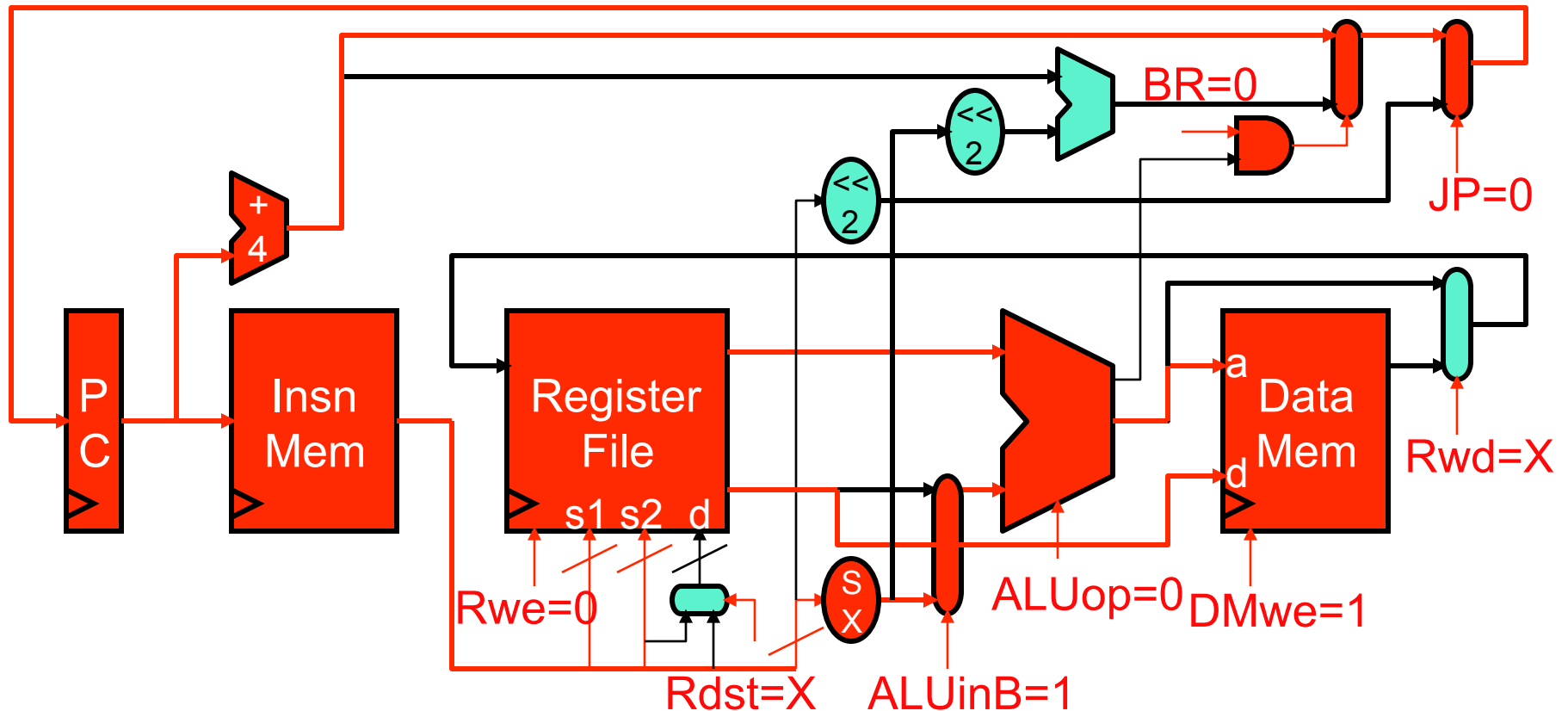
- 9 signals control flow of data through this datapath
 - MUX selectors, or register/memory write enable signals
 - A real datapath has 300-500 control signals

Example: Control for add



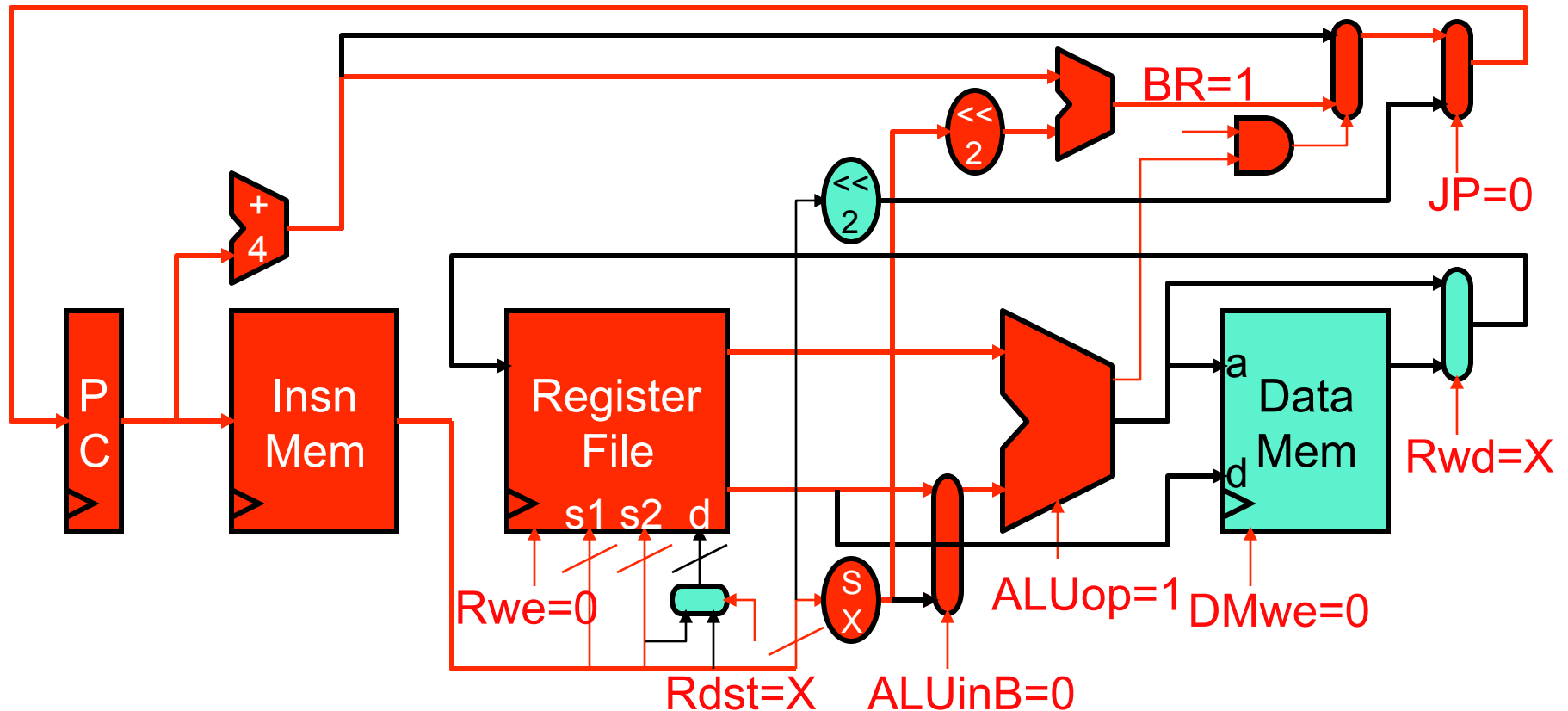
- Control for an instruction:
 - Values of all control signals to correctly execute it

Example: Control for sw



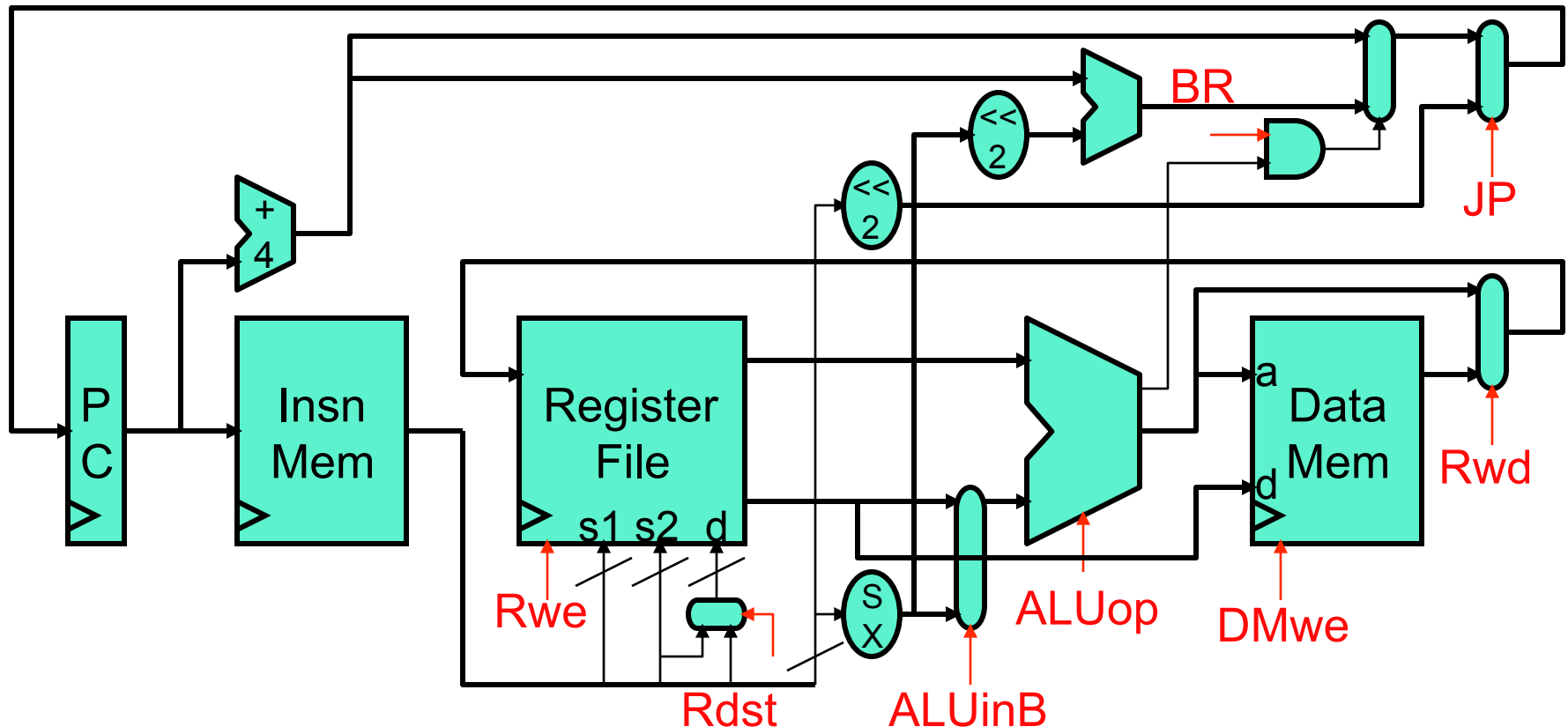
- Difference between **sw** and **add** is 5 signals
 - 3 if you don't count the X (don't care) signals

Example: Control for beq



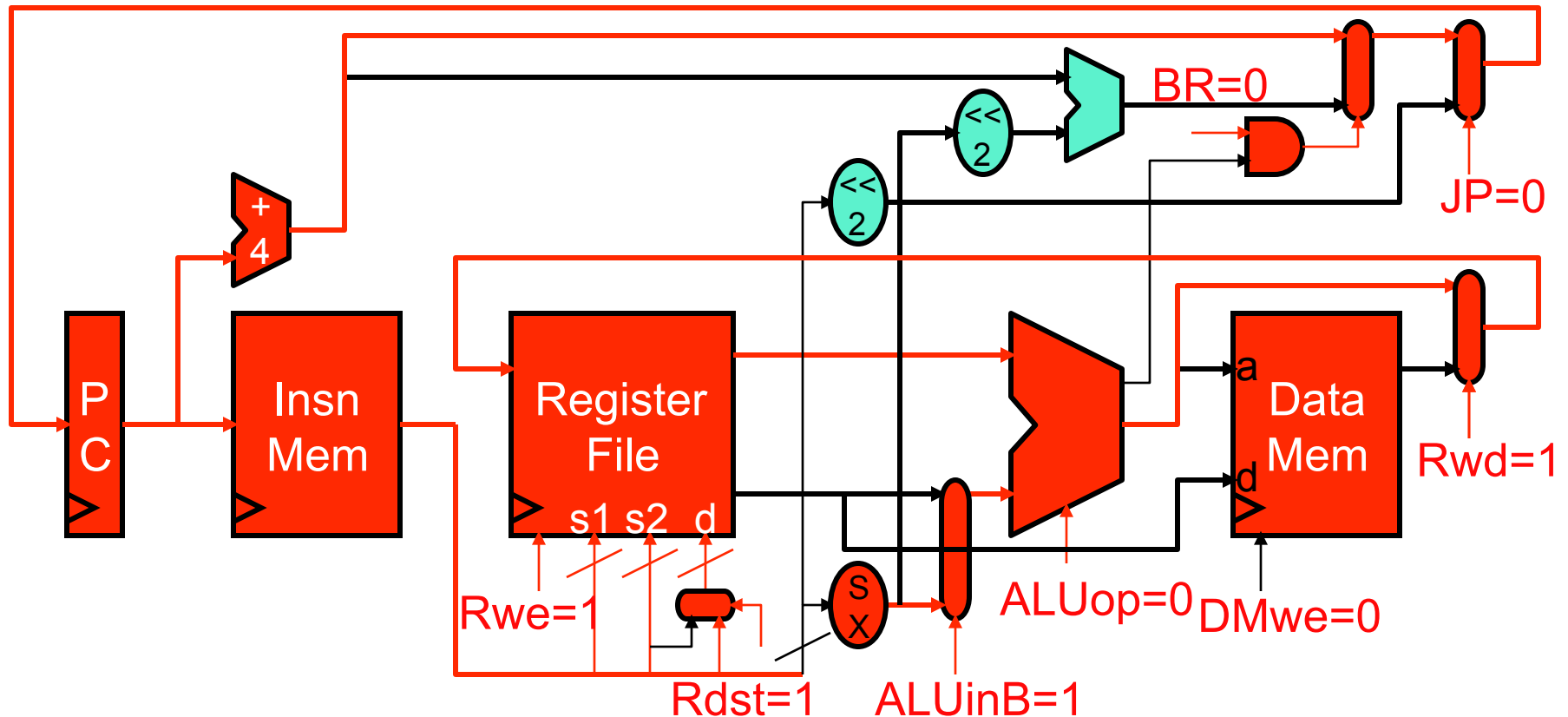
- Difference between **sw** and **beq** is only 4 signals

You all figure LW

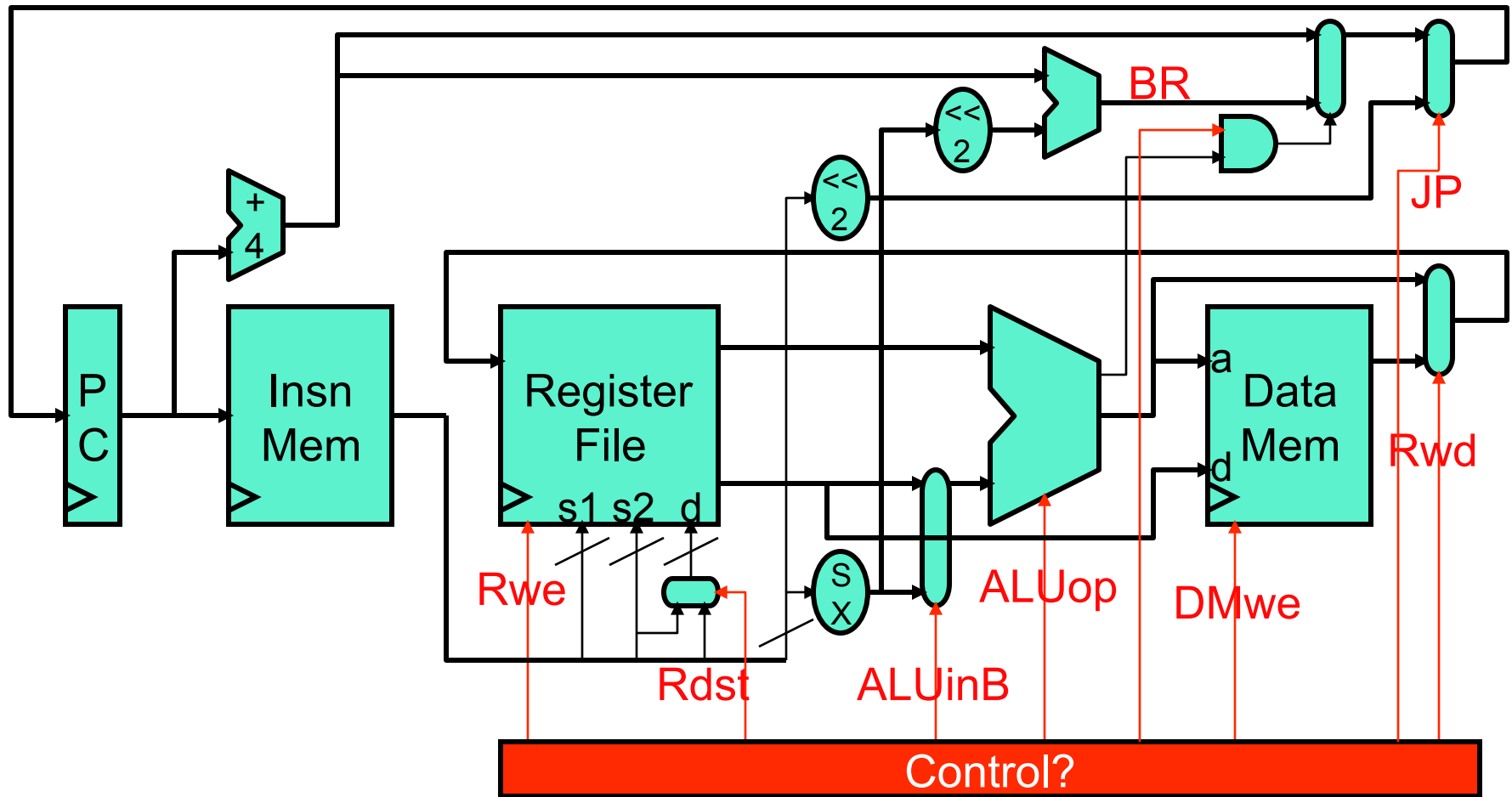


- How would these control signals be set for LW?

Example: Control for LW



How Is Control Implemented?

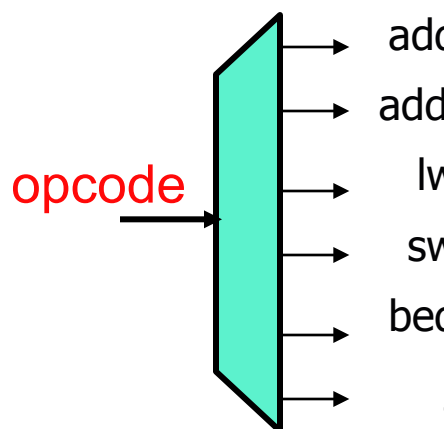


Implementing Control

- Each insn has a unique set of control signals
 - Most 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

Control Implementation: ROM

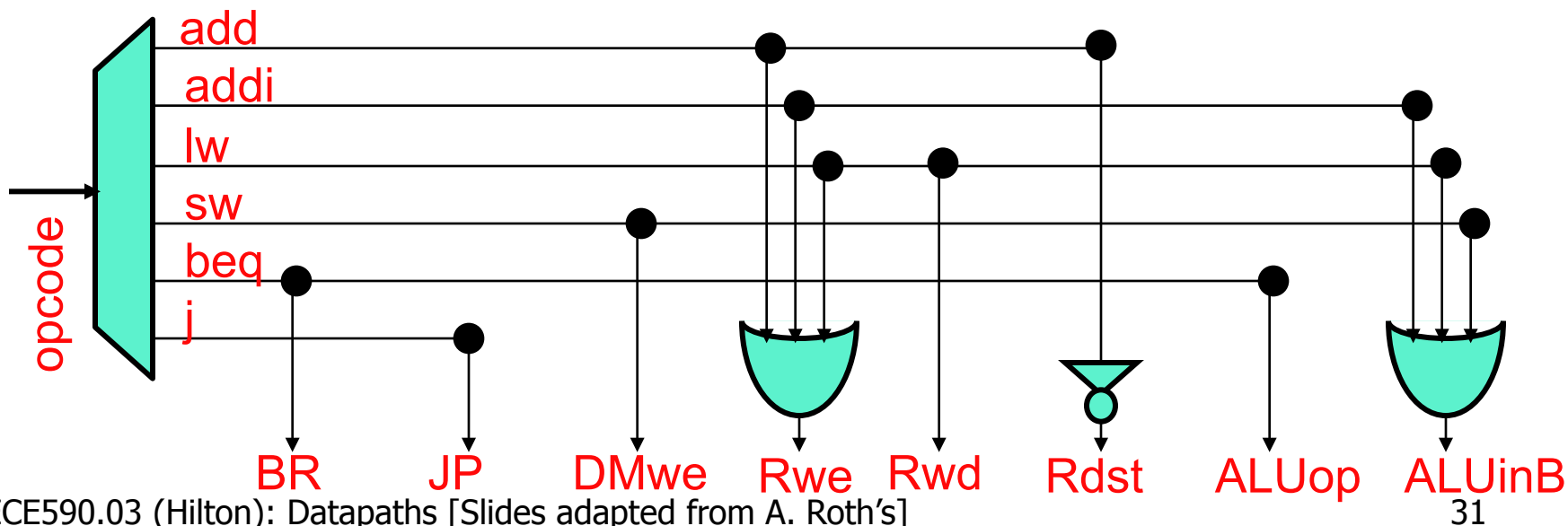
- **ROM (read only memory)**: think rows of bits
 - Bits in data words are control signals
 - Lines indexed by opcode
 - Example: ROM control for 6-insn MIPS datapath
 - X is “don’t care”



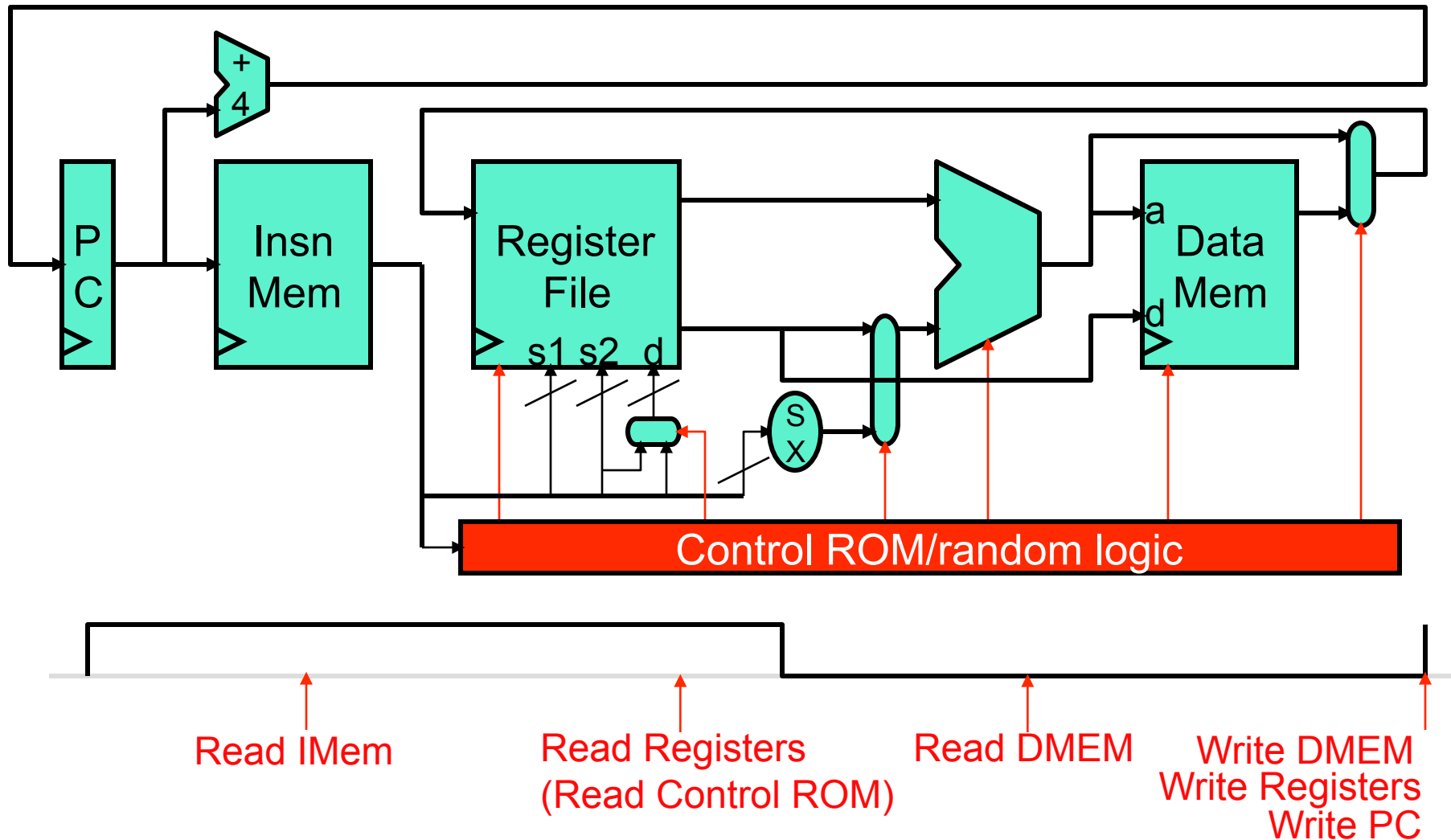
	BR	JP	ALUinB	ALUop	DMwe	Rwe	Rdst	Rwd
add	0	0	0	0	0	1	0	0
addi	0	0	1	0	0	1	1	0
lw	0	0	1	0	0	1	1	1
sw	0	0	1	0	1	0	X	X
beq	1	0	0	1	0	0	X	X
j	0	1	0	0	0	0	X	X

Control Implementation: Random Logic

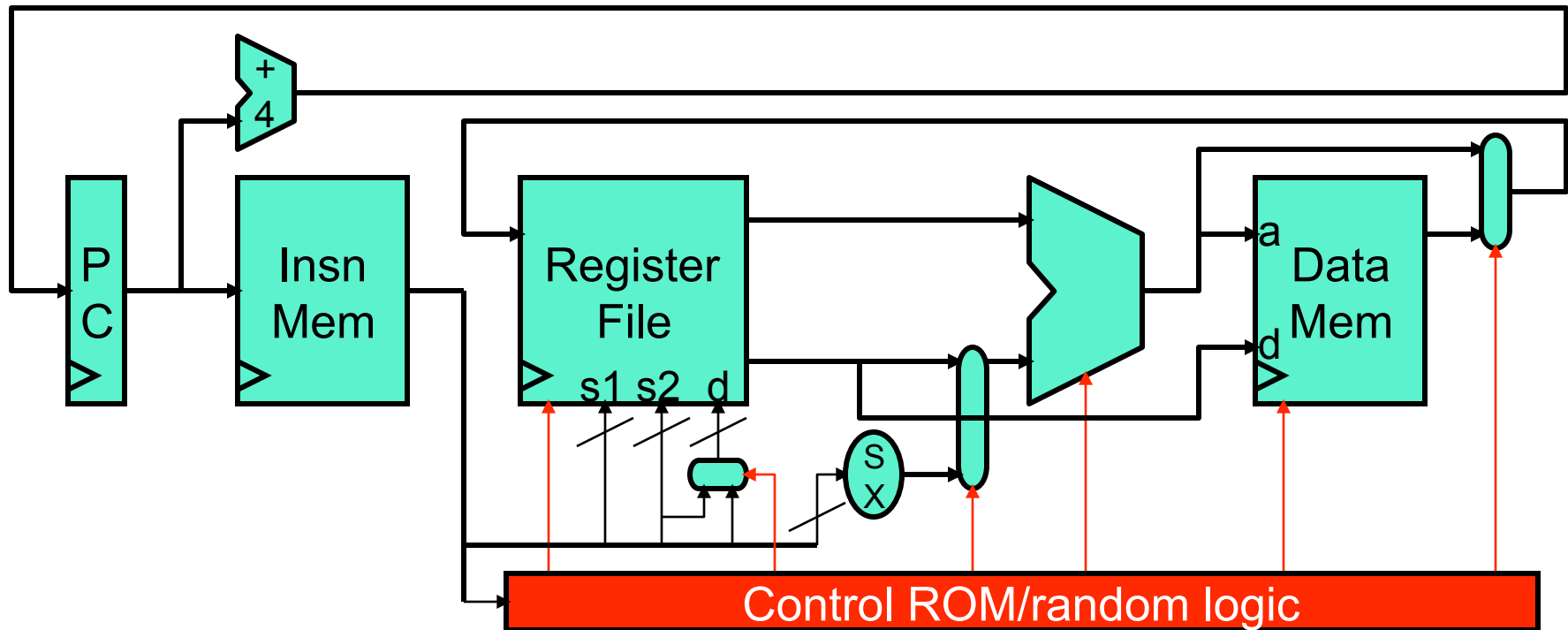
- Real machines have 100+ insns 300+ control signals
 - 30,000+ control bits (~4KB)
 - Not huge, but hard to make faster than datapath (important!)
- Alternative: **random logic** (random = 'non-repeating')
 - Exploits the observation: many signals have few 1s or few 0s
 - Example: random logic control for 6-insn MIPS datapath



Datapath and Control Timing



Single-Cycle Datapath Performance



- Goes against make common case fast (MCCF) principle
 - + Low Cycles Per Instruction (**CPI**): 1
 - Long clock period: to accommodate slowest insn

Interlude: Performance

- Previous slide alludes to something new: **Performance**
 - Don't just want it to work...
 - But want it to go fast!

- Three components to performance:

Number of instructions
x Cycles per instruction (CPI)
x Clock Period (1 / Clock frequency)

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}} = \frac{\text{Seconds}}{\text{Program}}$$

Interlude: Performance

- Three components to performance:

Number of instructions <- **Compiler's Job**

x Cycles per instruction (CPI)

x Clock Period (1 / Clock frequency)

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}} = \frac{\text{Seconds}}{\text{Program}}$$

- Insns/Program: determined by compiler + ISA
 - Generally assume fixed program when do **micro-architecture**

Micro-architectural factors

- Micro-architecture:
 - The details of how the ISA is implemented
 - Affects CPI and Clock frequency
- Often will look at fixed program, and consider MIPS
 - Million Instructions Per Second
 - $\text{MIPS} = \text{IPC} * \text{Frequency (in MHz)}$
 - $\text{IPC} = \text{Instruction Per Cycle (1 / CPI)}$
 - Gives “Bigger is better” number

$$\frac{\text{Instructions}}{\text{Cycle}} \times \frac{\text{Cycles}}{\text{Second}} = \frac{\text{Instructions}}{\text{Second}}$$

(IPC) (Frequency) (Throughput)

“Best” IPC

- For now, best we can do: $IPC = 1$ ($CPI = 1$)
 - Do 1 instruction every cycle
- Later:
 - Real processors can do multiple instructions at once!
 - Potentially: $IPC < 1$!
 - Best possible IPC depends on design

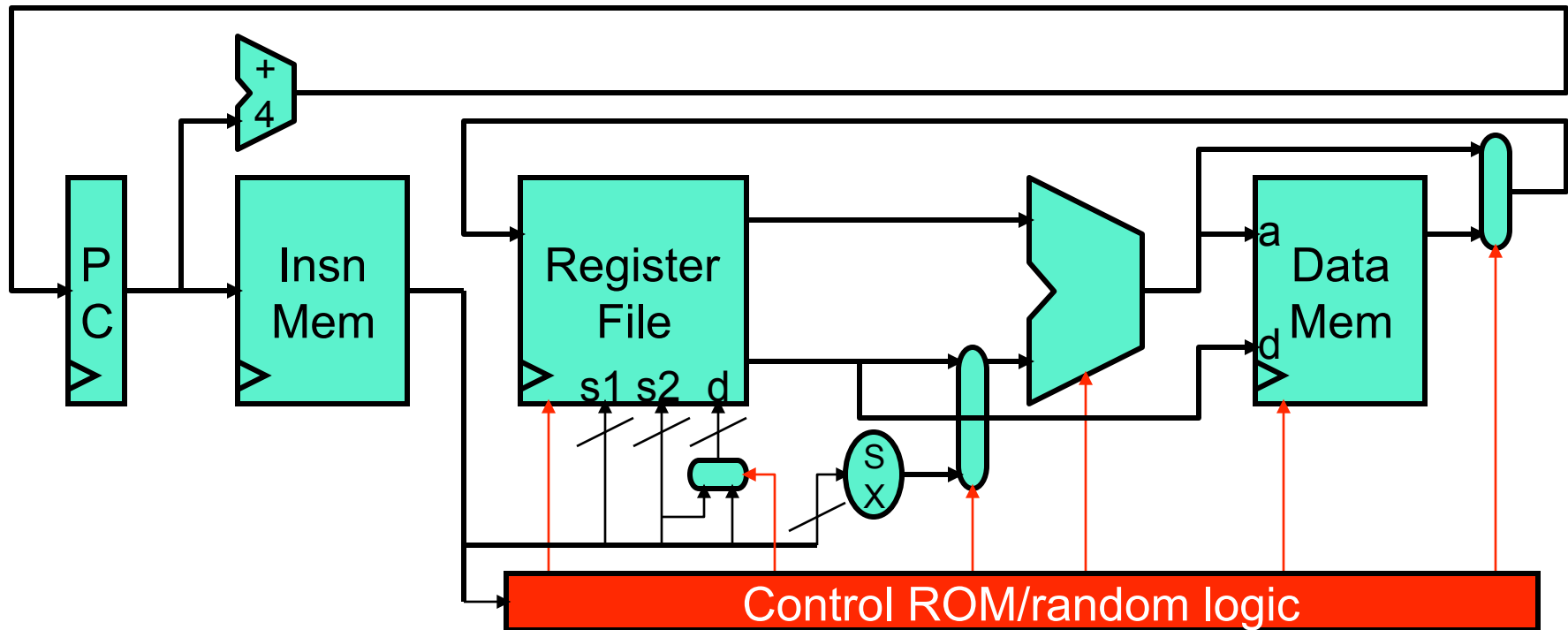
Performance vs

- 1990s: Performance at all cost
 - Actually more “clock frequency” at all cost...
- Now: Care about other things
 - **Energy** (electric bill, battery life)
 - **Power** (cooling, also affects energy)
 - Area (chip cost)
 - Reliability (tolerance of transient faults: e.g., charge particle strikes)
 - ...
- Important metric these days “Performance / Watt”
 - Throughput divided by power consumption
 - Why?

Performance Modeling and Analysis

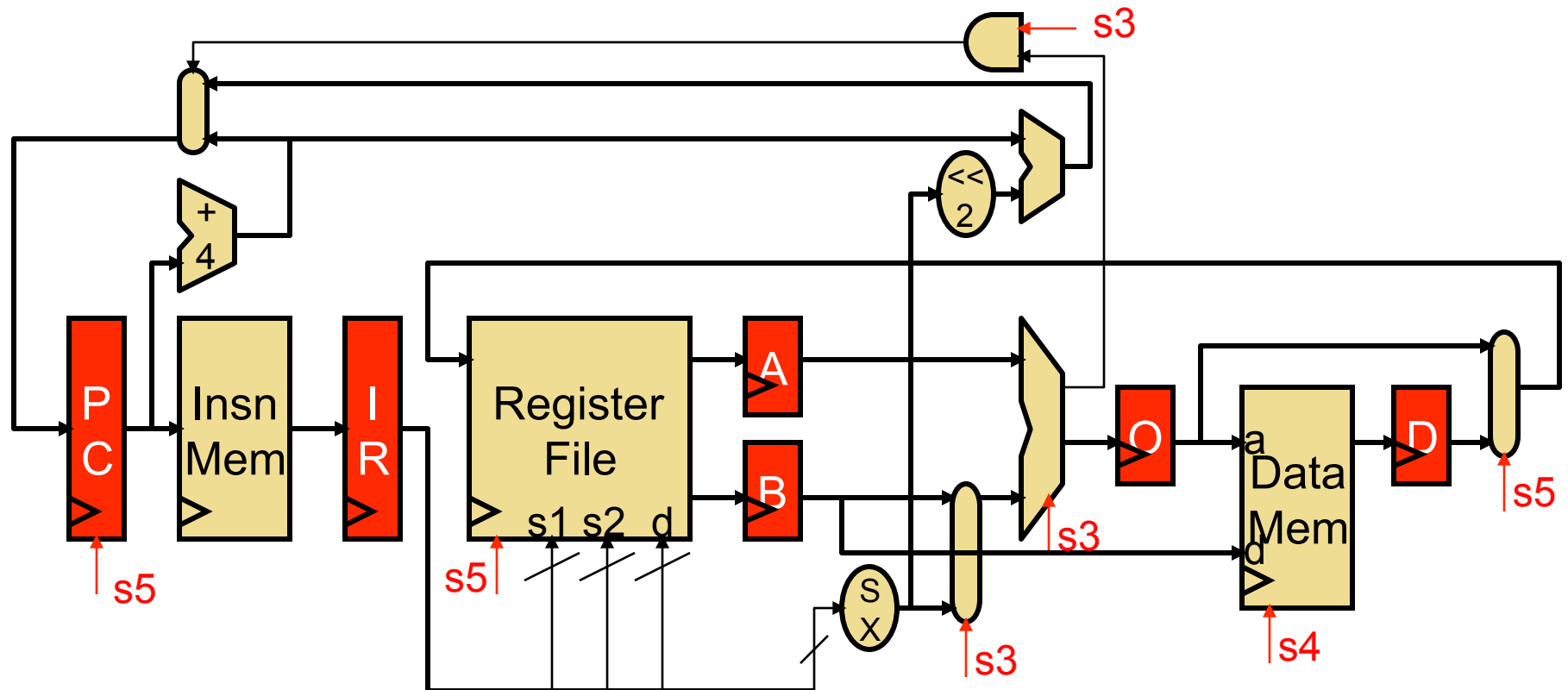
- Speaking of performance
 - Making a processor takes time (years) and money (millions)
 - Want to know it will perform well before you finish
 - If its wrong, doing it all over is painful...
 - Performance can be simulated in software
 - Estimate what IPC will be
 - Guide design
- This is my old job by the way...

Single-Cycle Datapath Performance



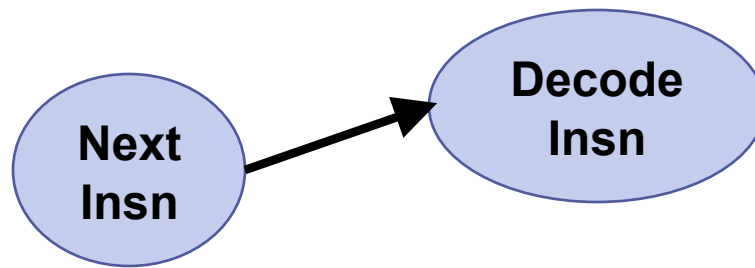
- Goes against make common case fast (MCCF) principle
 - + Low Cycles Per Instruction (**CPI**): 1
 - Long clock period: to accommodate slowest insn

Alternative: Multi-Cycle Datapath



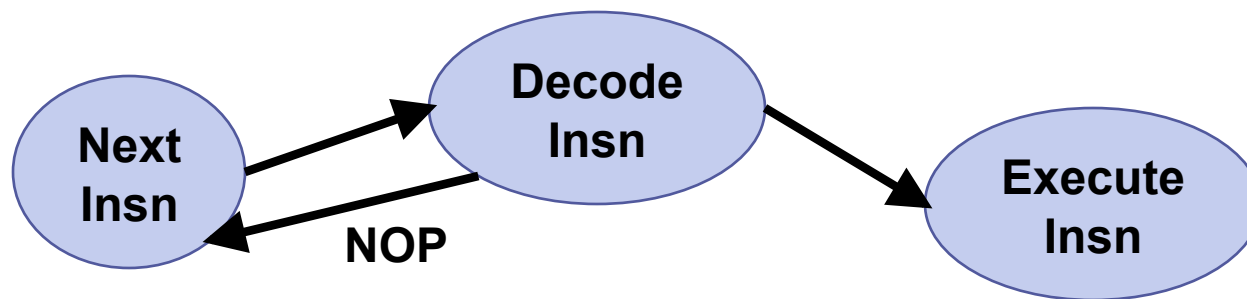
- **Multi-cycle datapath:** attacks high clock period
 - Cut datapath into multiple stages (5 here), isolate using FFs
 - **FSM** control “walks” insns thru stages (by staging control signals)
 - + Insns can bypass stages and exit early

Multi-cycle Datapath FSM



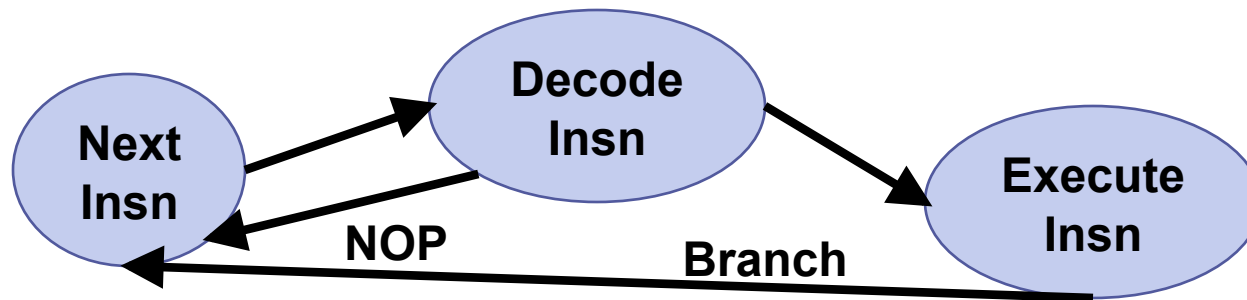
- First state: Get a New Instruction
 - Output signals to fetch (e.g., read enable IMEM)
 - Next State: Always Decode

Multi-cycle Datapath FSM



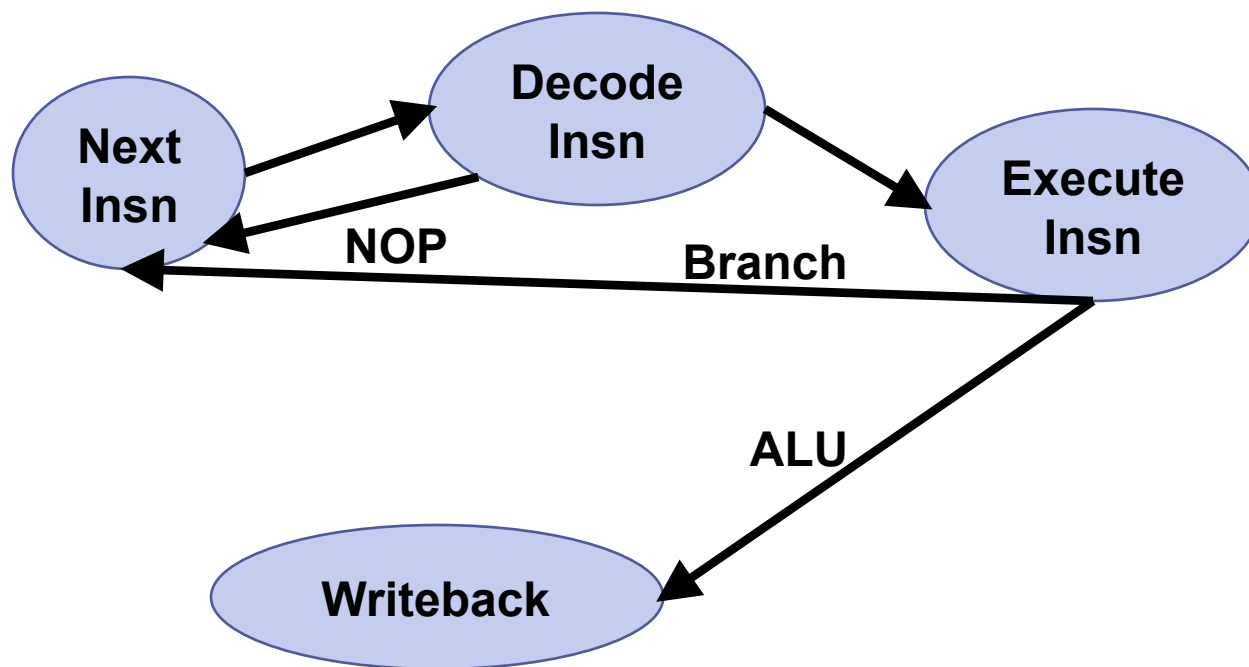
- Second State: Decode
 - Output signals to decode instruction (RdEn RegFile)
 - Go to Next Insn if NOP
 - Otherwise Execute

Multi-cycle Datapath FSM



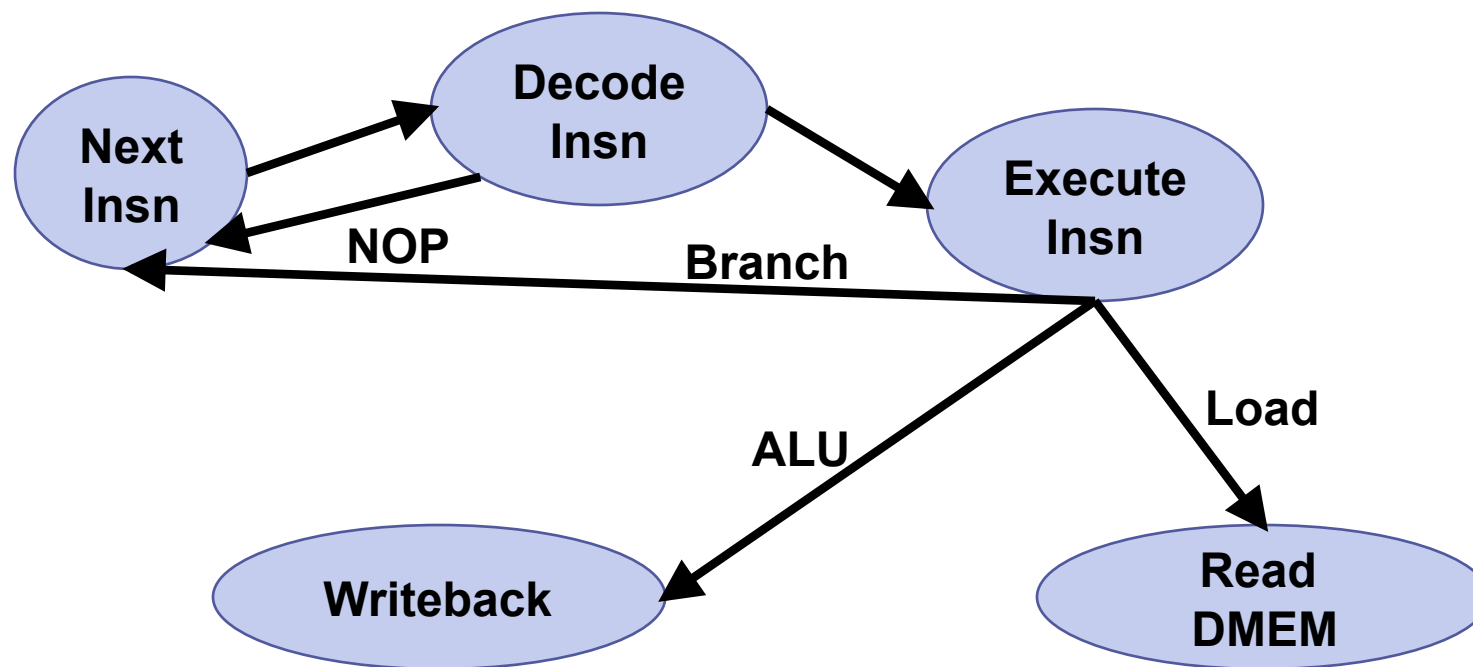
- Execute State
 - Execute Insn (varies by insn type)
 - Next State: Also depends on insn type
 - Branches: Next Insn

Multi-cycle Datapath FSM



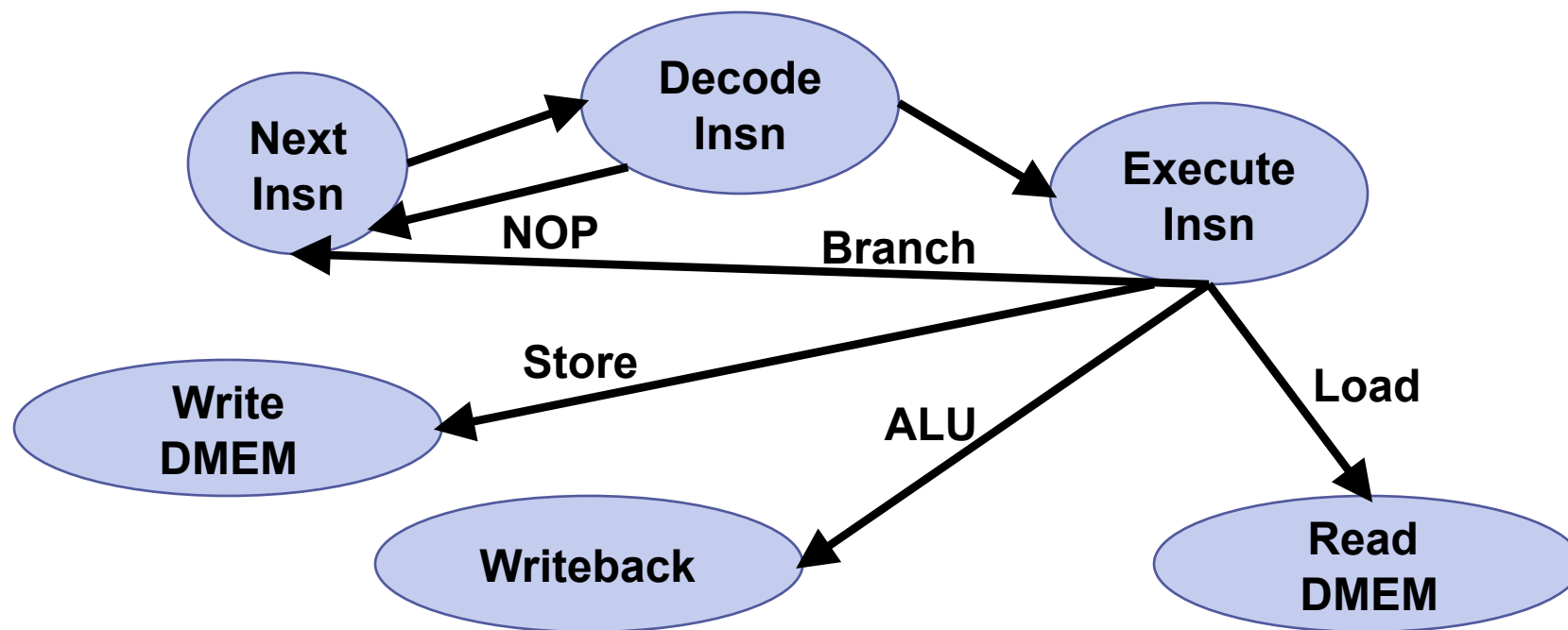
- Execute State
 - Execute Insn (varies by insn type)
 - Next State: Also depends on insn type
 - ALU op: write register

Multi-cycle Datapath FSM



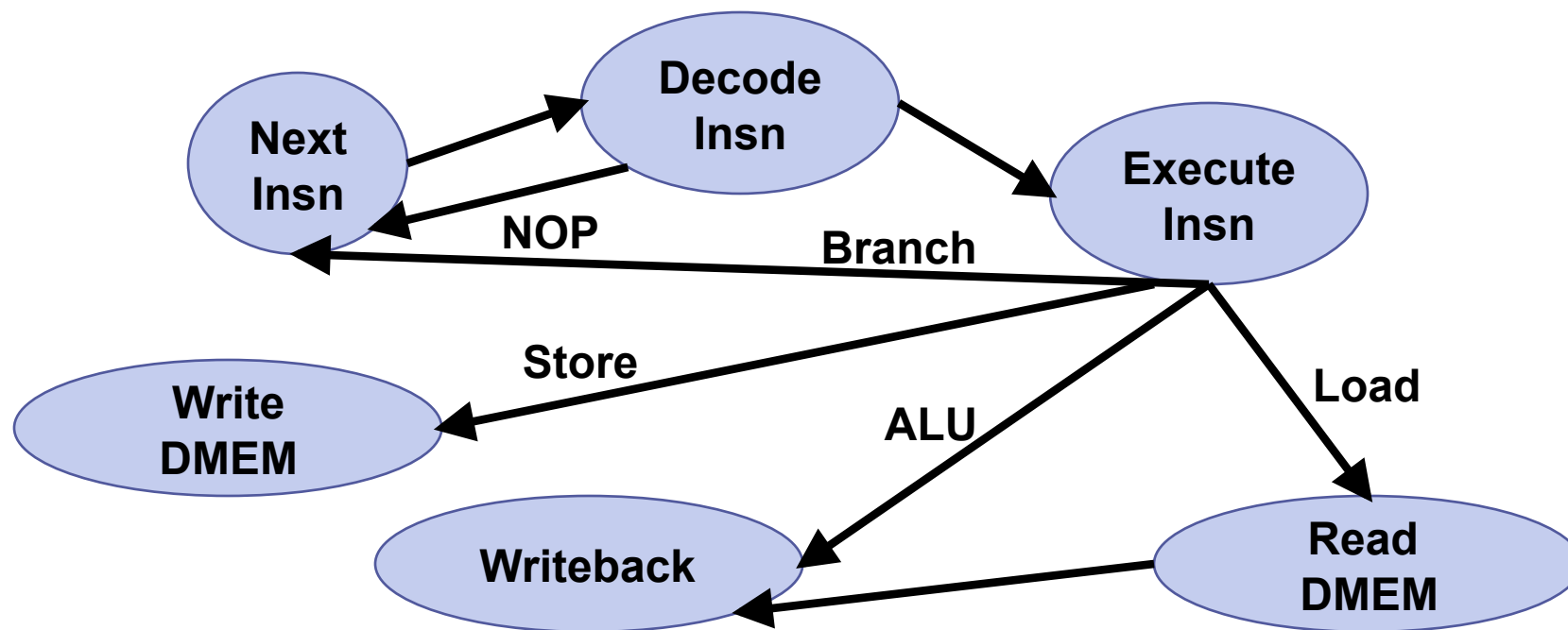
- **Execute State**
 - Execute Insn (varies by insn type)
 - Next State: Also depends on insn type
 - Load: Read Memory

Multi-cycle Datapath FSM



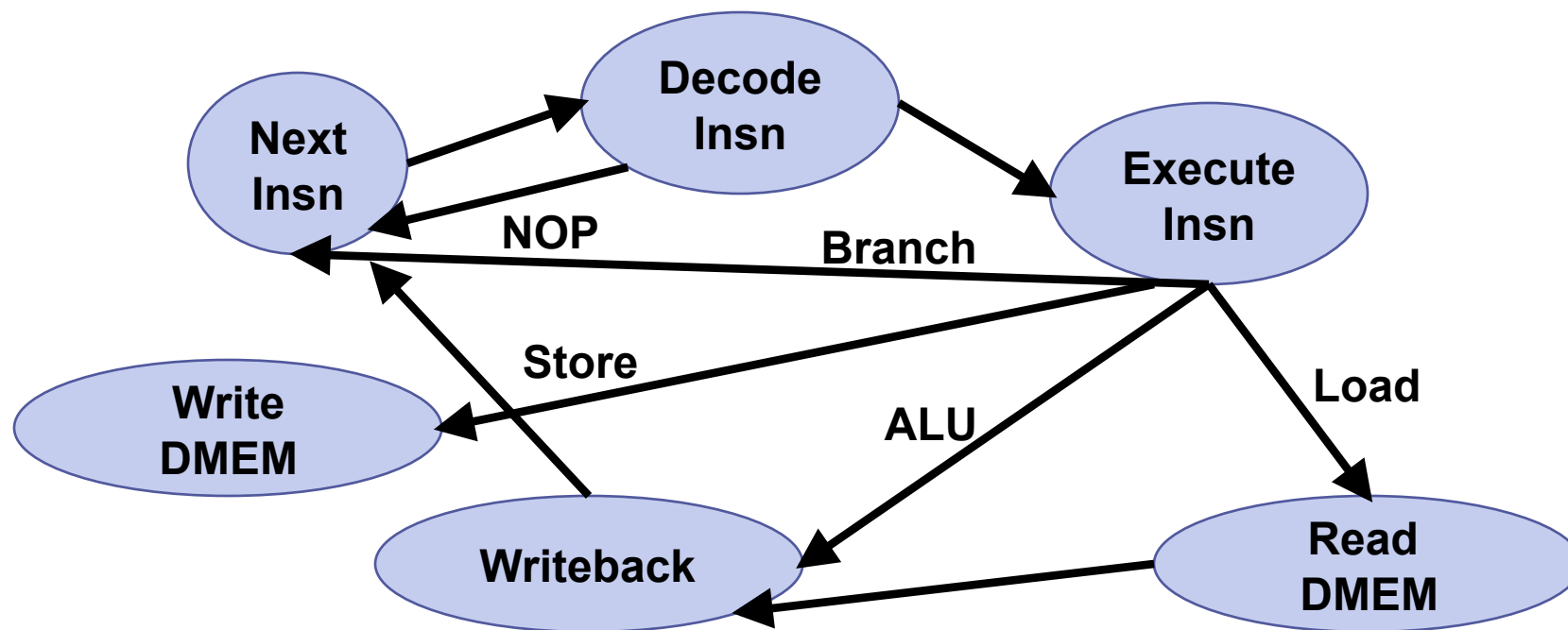
- **Execute State**
 - Execute Insn (varies by insn type)
 - Next State: Also depends on insn type
 - Store: Write Memory

Multi-cycle Datapath FSM



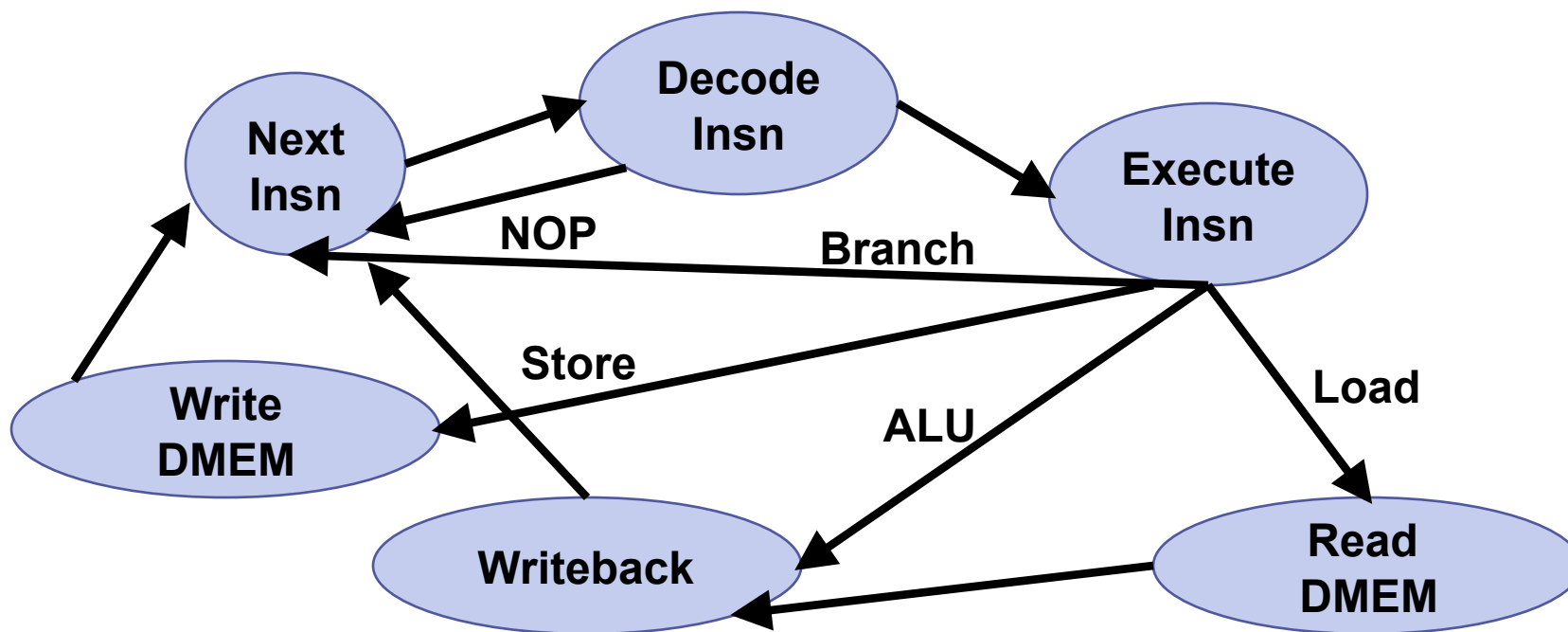
- Read DMEM State
 - Control signals enable DMEM Read
 - Next state is writeback

Multi-cycle Datapath FSM



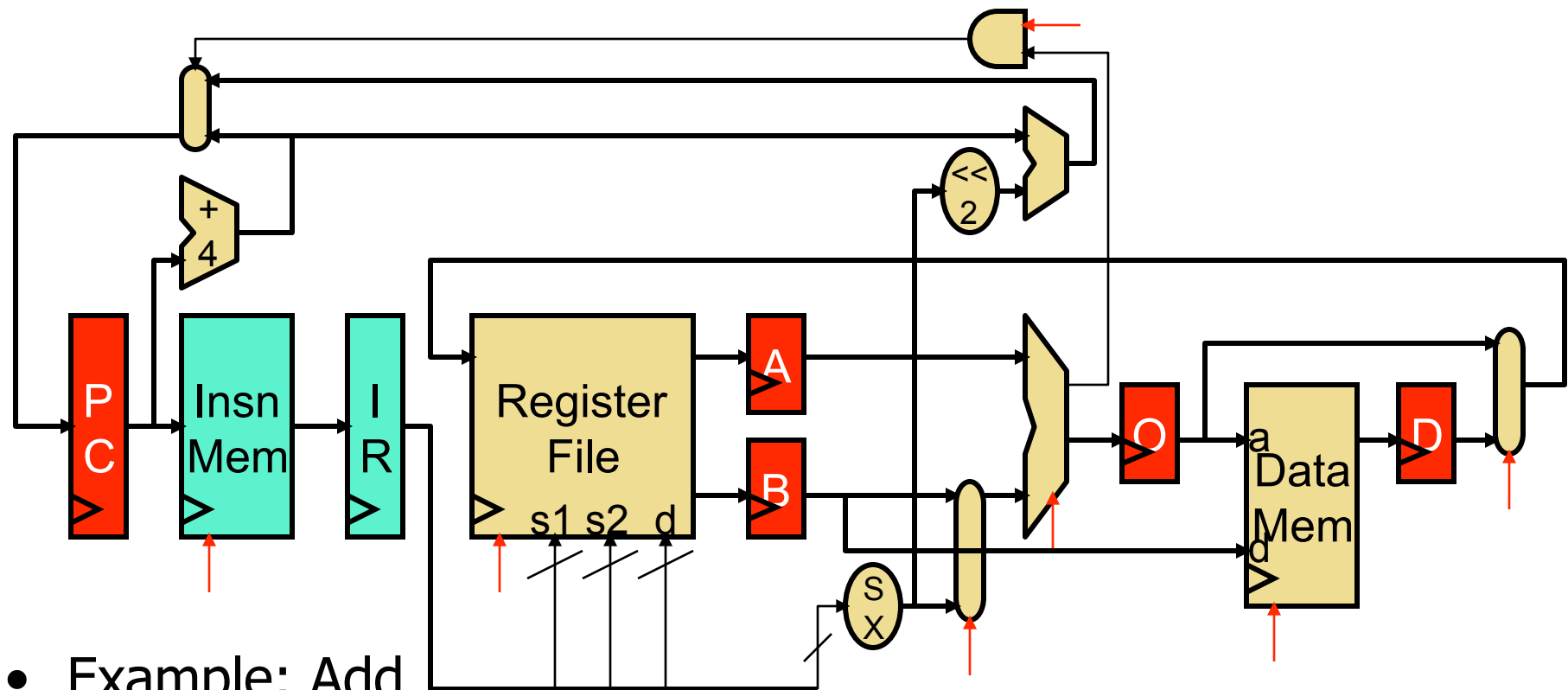
- Writeback state
 - Control signals enable regfile write
 - Next state: Next Insn

Multi-cycle Datapath FSM



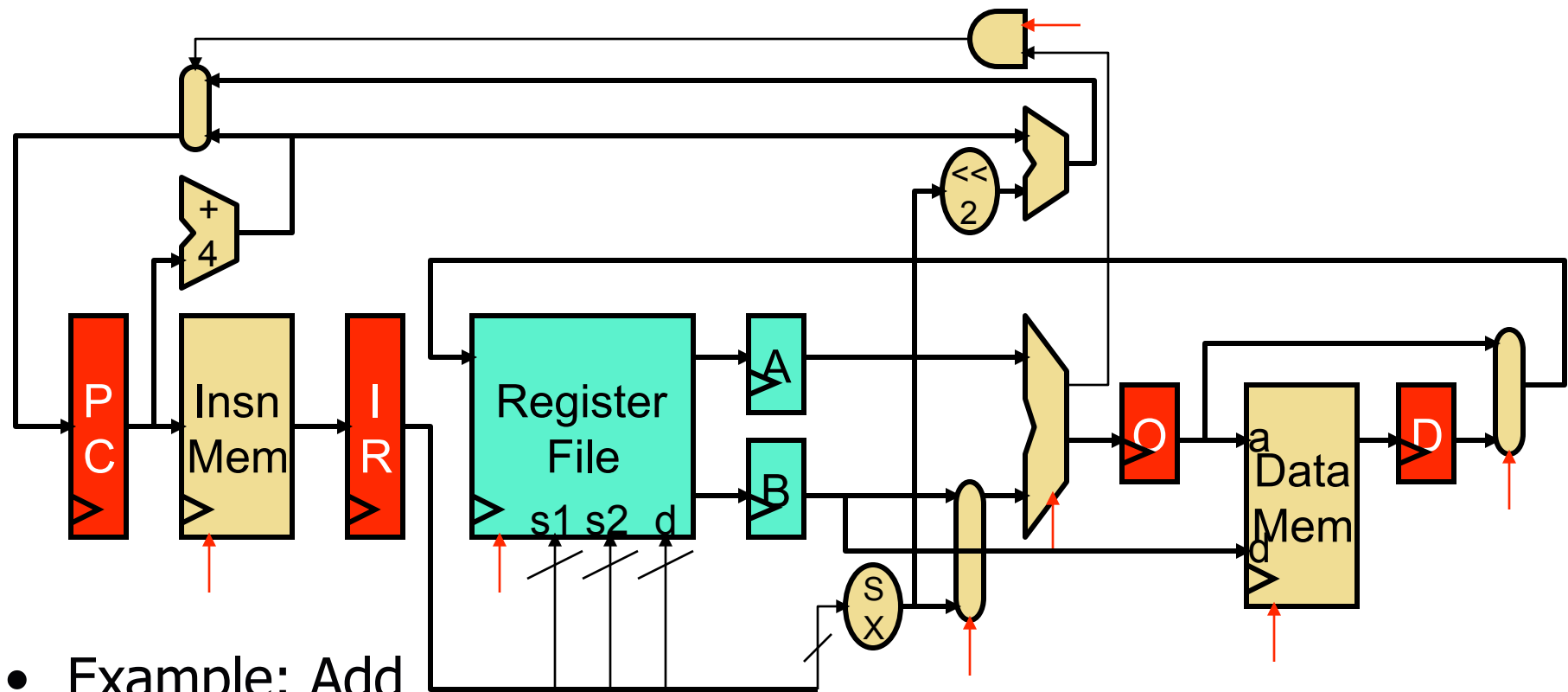
- Write DMEM state
 - Control signals enable memory write
 - Next state: Next Insn

Multi-Cycle Datapath Example: Add



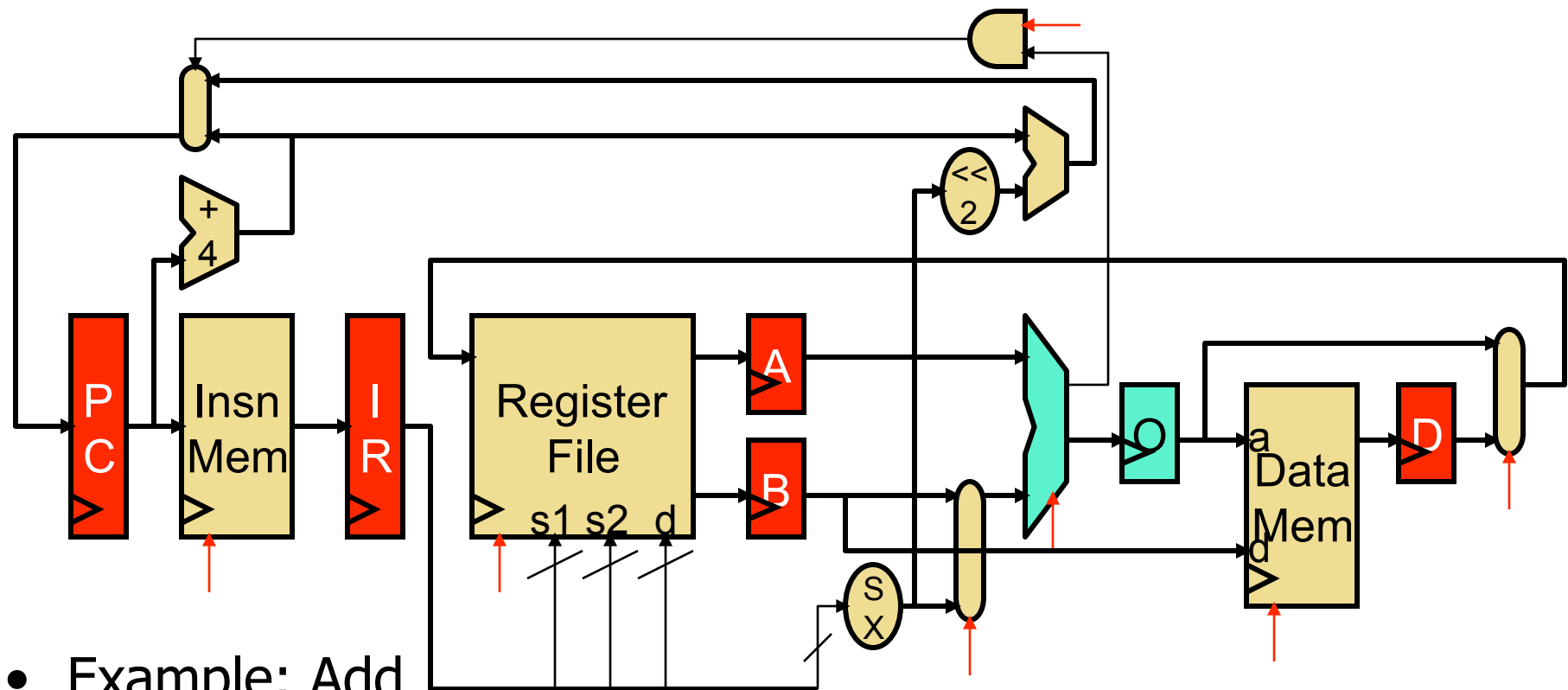
- Example: Add
 - Cycle 1: Read IMEM

Multi-Cycle Datapath Example: Add



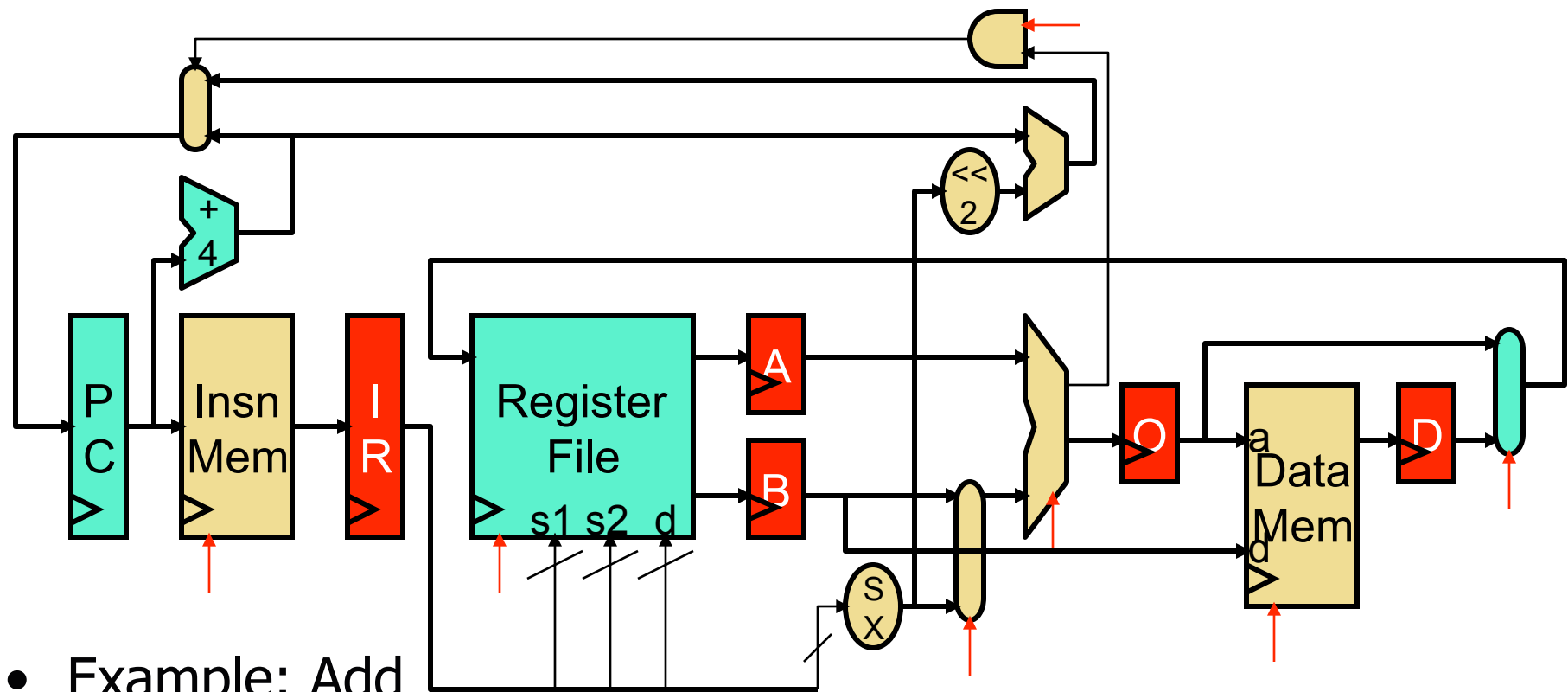
- Example: Add
 - Cycle 1: Read IMEM
 - Cycle 2: Decode + Read RF

Multi-Cycle Datapath Example: Add



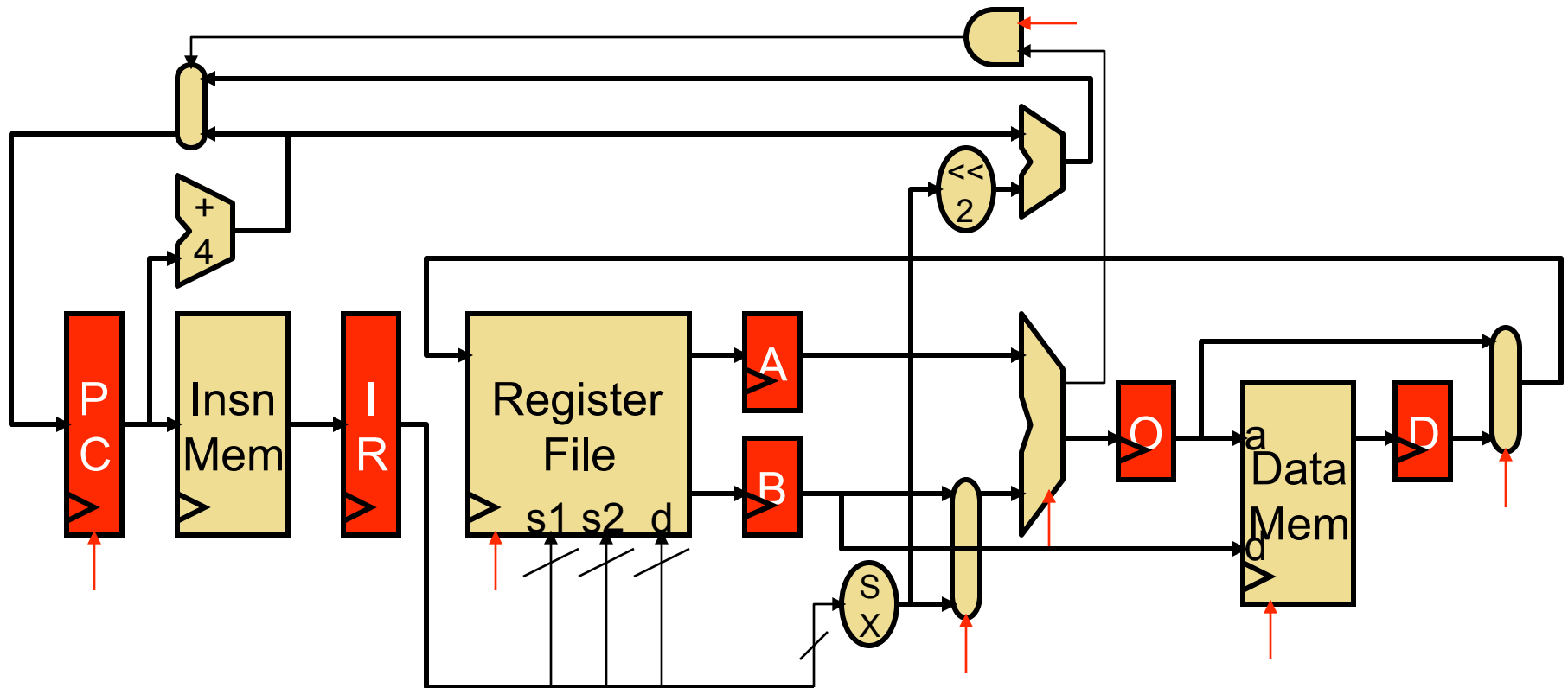
- Example: Add
 - Cycle 1: Read IMEM
 - Cycle 2: Decode + Read RF
 - Cycle 3: ALU

Multi-Cycle Datapath Example: Add



- Example: Add
 - Cycle 1: Read IMEM
 - Cycle 2: Decode + Read RF
 - Cycle 3: ALU
 - Cycle 4: Writeback + Increment PC

Multi-Cycle Datapath Performance



- Opposite performance split of single-cycle datapath
 - + Short clock period
 - **High CPI**

Multi-cycle Data-path CPI

- CPI depends on instructions
 - Branches / Jumps: 3 cycles
 - ALU: 4 cycles
 - Stores: 4 cycles
 - Loads: 5 cycles
- Overall CPI is weighted average
- Example:
 - 20% loads, 15% stores, 20% branches, 45% ALU

Multi-cycle Data-path CPI

- CPI depends on instructions
 - Branches / Jumps: 3 cycles
 - ALU: 4 cycles
 - Stores: 4 cycles
 - **Loads: 5 cycles**
 - Overall CPI is weighted average
 - Example:
 - **20% loads**, 15% stores, 20% branches, 45% ALU
- CPI= $0.20 * 5 +$

Multi-cycle Data-path CPI

- CPI depends on instructions
 - Branches / Jumps: 3 cycles
 - ALU: 4 cycles
 - **Stores: 4 cycles**
 - Loads: 5 cycles
 - Overall CPI is weighted average
 - Example:
 - 20% loads, **15% stores**, 20% branches, 45% ALU
- $$\text{CPI} = 0.20 * 5 + 0.15 * 4 +$$

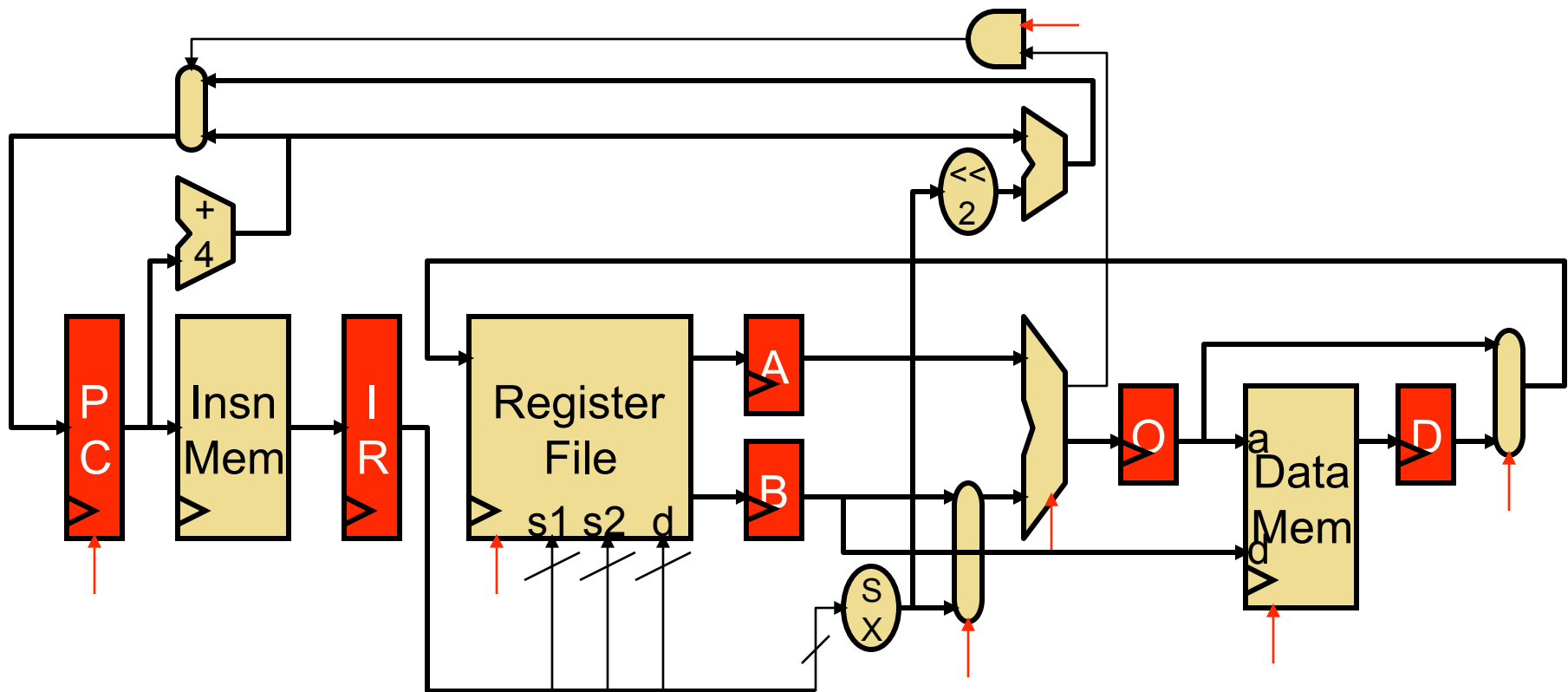
Multi-cycle Data-path CPI

- CPI depends on instructions
 - Branches / Jumps: 3 cycles
 - ALU: 4 cycles
 - Stores: 4 cycles
 - Loads: 5 cycles
 - Overall CPI is weighted average
 - Example:
 - 20% loads, 15% stores, 20% branches, 45% ALU
- $$\text{CPI} = 0.20 * 5 + 0.15 * 4 + 0.20 * 3 + 0.45 * 4 = \mathbf{4.0}$$

Multi-cycle Datapath Performance

- Single-cycle
 - Clock period = 50ns, CPI = 1
 - Performance = 50 ns/insn
- Multi-cycle
 - Clock period = 10ns
 - $\text{CPI} = (0.2 \cdot 3 + 0.2 \cdot 5 + 0.6 \cdot 4) = 4$
 - Performance = 40 ns/insn
- But wait...

Multi-Cycle Datapath Performance



- Did not just cut up existing logic into 5 pieces
- Also added logic (flip flops)
 - So clock period not 1/5 of single cycle, but slightly longer

Multi-cycle Datapath Performance

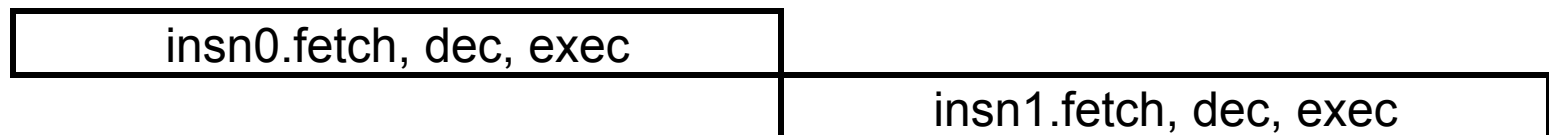
- Single-cycle
 - Clock period = 50ns, CPI = 1
 - Performance = 50 ns/insn
- Multi-cycle
 - Clock period = **12ns**
 - $\text{CPI} = (0.2 \cdot 3 + 0.2 \cdot 5 + 0.6 \cdot 4) = 4$
 - Performance = **48 ns/insn**
- Better, but not as exciting...
 - Can we do better still?
 - Have our cake (low CPI) and eat it too (high clock frequency)?

Clock Period and CPI

- Single-cycle datapath

- + Low CPI: 1

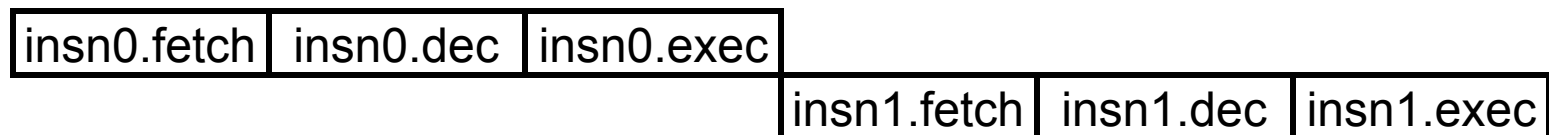
- Long clock period: to accommodate slowest insn



- Multi-cycle datapath

- + Short clock period

- High CPI



- Can we have both low CPI and short clock period?

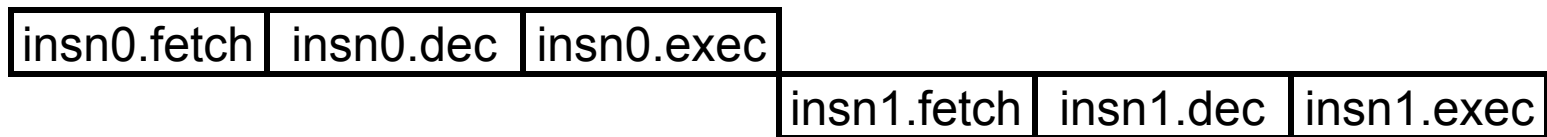
- No good way to make a single insn go faster

- + Insn latency doesn't matter anyway ... insn throughput matters

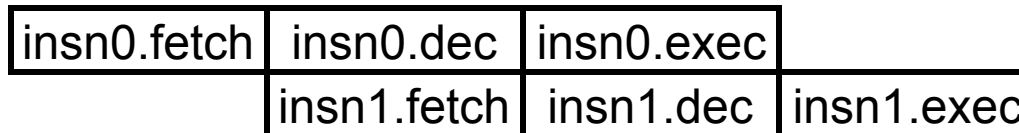
- Key: **exploit inter-insn parallelism**

Pipelining

- **Pipelining**: important performance technique
 - **Improves insn throughput rather than insn latency**
 - **Exploits parallelism at insn-stage level to do so**
 - Begin with multi-cycle design



- When insn advances from stage 1 to 2, next insn enters stage 1



- Individual insns take same number of stages
 - + **But insns enter and leave at a much faster rate**
 - Physically breaks “atomic” VN loop ... but must maintain illusion
- Revisit at end of semester (hopefully)

Summary

- Datapaths:
 - Single Cycle
 - What do we need?
 - Control
 - How control is implemented
 - Multi-cycle
 - Faster clock (yay!)
 - Worse CPI (boooo)
 - Performance:
 - IPC
 - Performance / Watt
 - CPU Performance Equation
 - Pipelining
 - Teaser for later!