ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2025

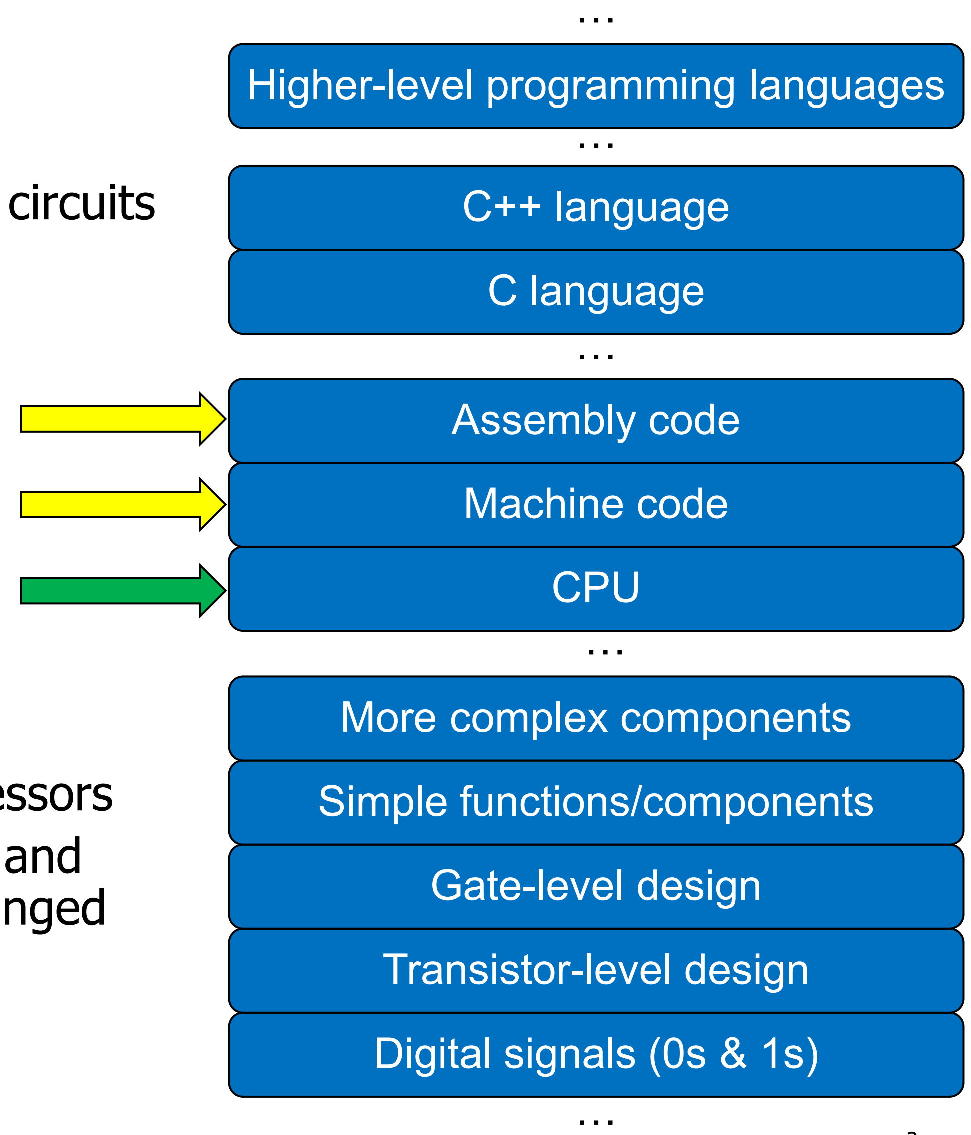
Datapath

Yiran Chen and Hai “Helen” Li
Duke University

Slides are derived from work by
Rabih Younes, John Board, Andrew Hilton, and Tyler Bletsch (Duke)

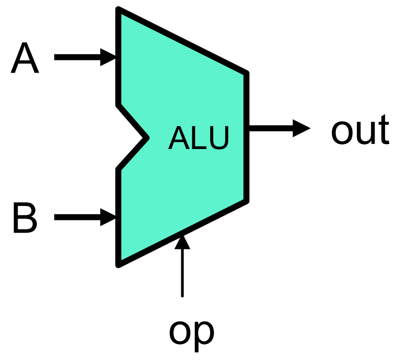
Big Picture of Where We Are

- Start of semester: Digital Logic
  - Combinational + sequential digital circuits
  - Building blocks of digital design
- Most recently: ISA
  - Assembly
  - Lowest-level software
- Now: Datapath
  - Where they meet
  - Hardware implementation of processors
  - Where machine code is processed and registers + memory states are changed

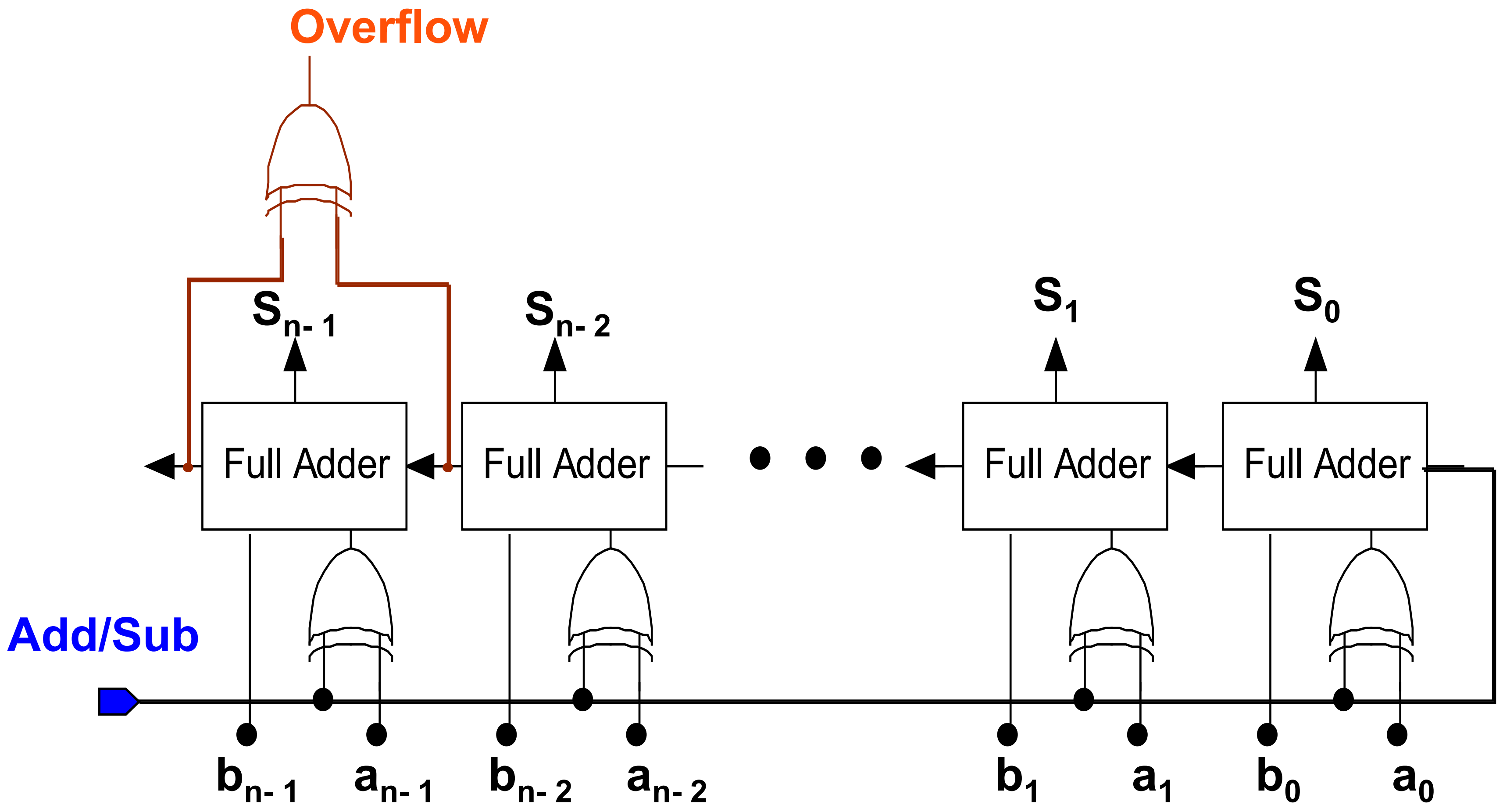


Necessary CPU Ingredient: The ALU

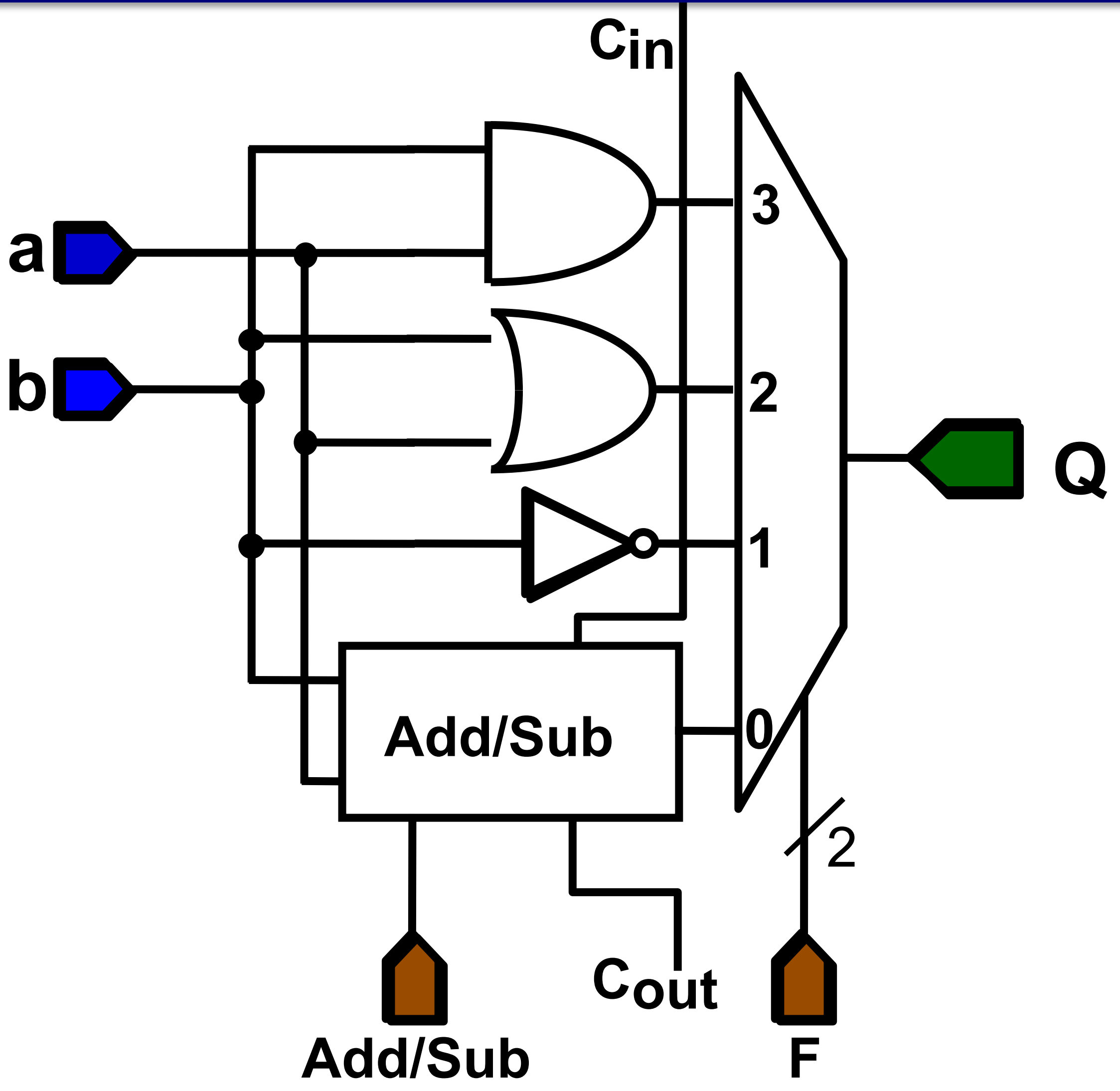
- **ALU: Arithmetic/Logic Unit**
  - Performs any supported math or logic operation on two inputs
  - Which operation is chosen by a third input



An Add/Subtract with Overflow Detection

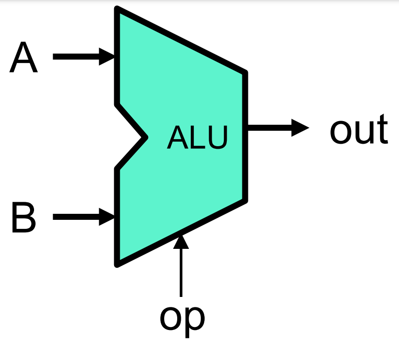


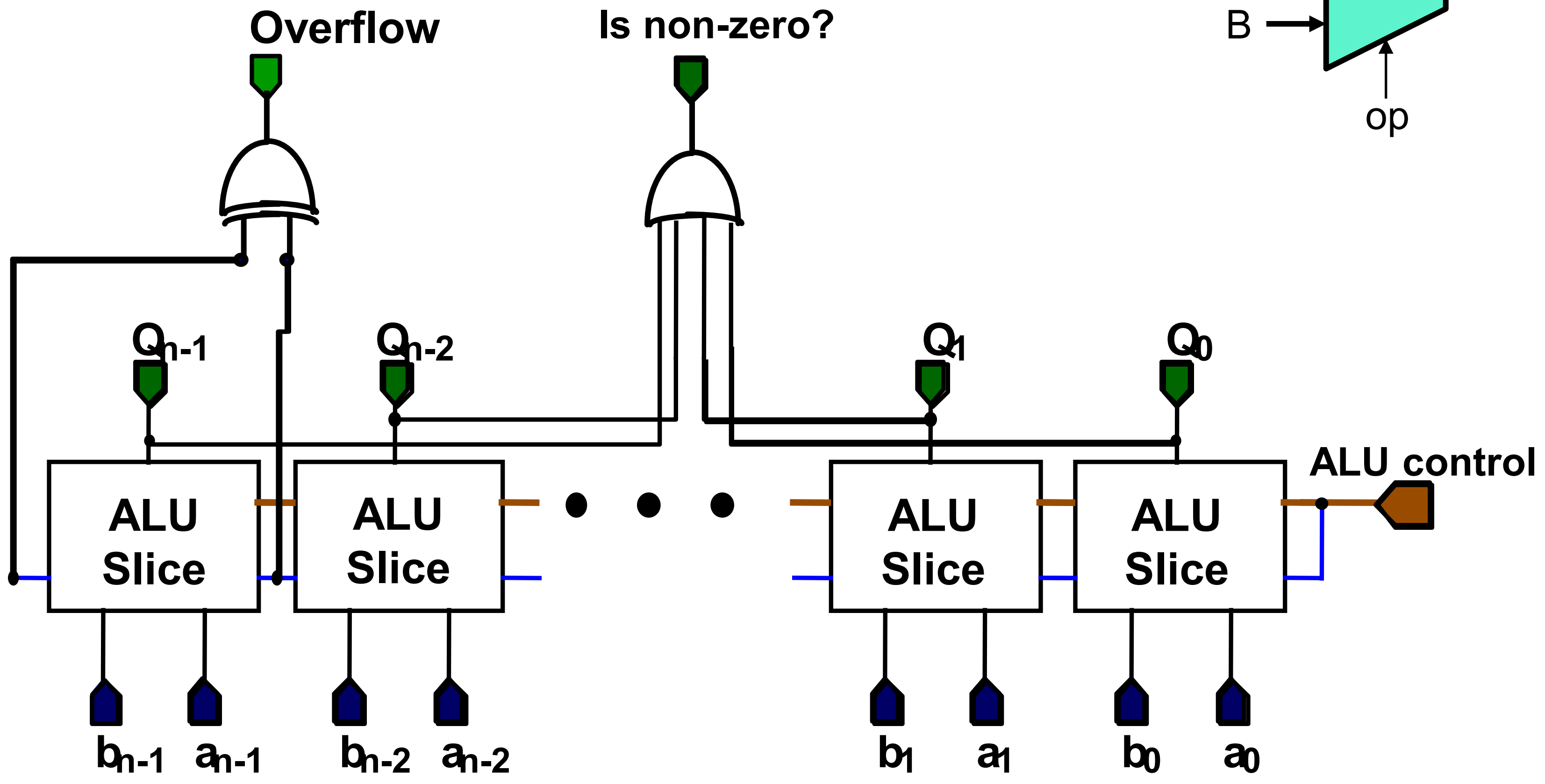
An Example of an ALU Slice (i.e., 1-bit ALU)



A/S	F	Q
0	0	a + b
1	0	a - b
-	1	NOT b
-	2	a OR b
-	3	a AND b

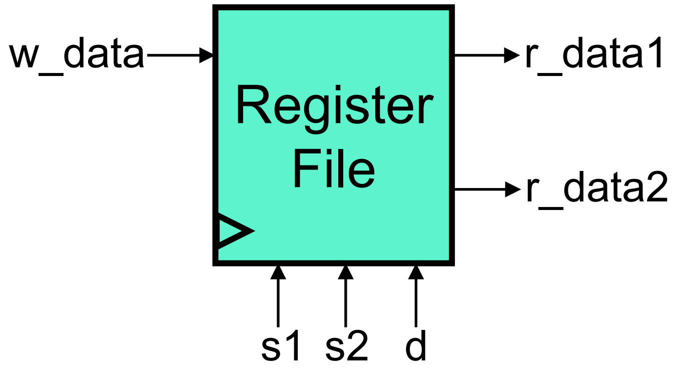
The ALU (n-bit ALU)





Another CPU Ingredient: RegFile

- **RegFile: Register File**
  - Where operands are fetched and results are stored
  - We don’t want to keep going to memory back and forth
    - RegFile is way faster



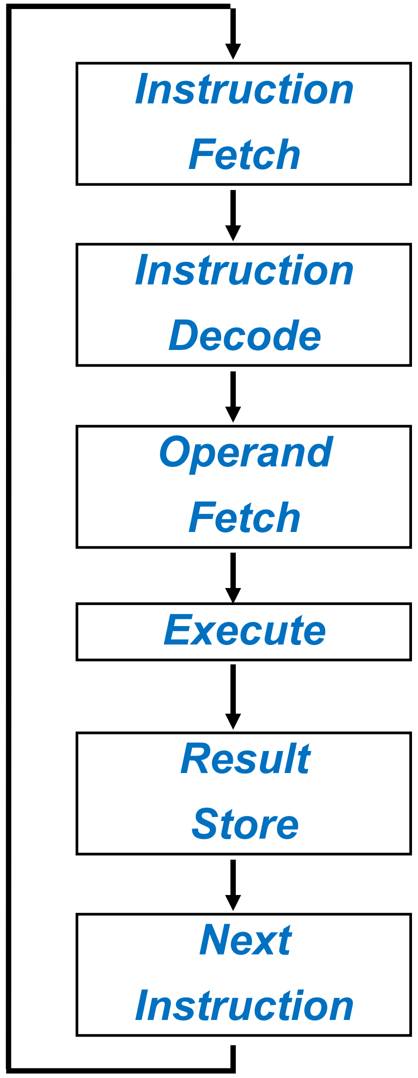
Now, Let's Implement A MIPS-Like Datapath

- We'll only accommodate for the following MIPS ISA insns:

```
add $1,$2,$3
addi $1,$2,50
lw $1,4($3)
sw $1,4($3)
beq $1,$2,PC_relative_target
j absolute_target
```

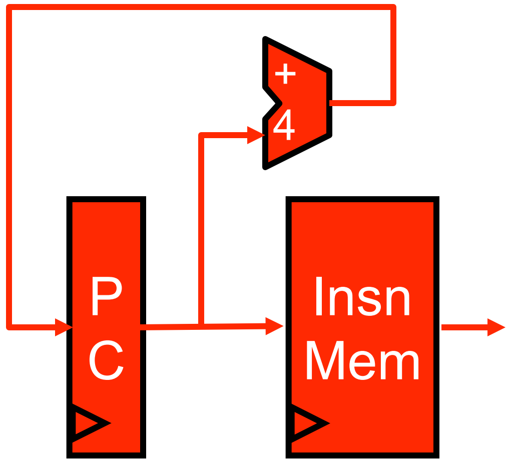
- Why only these?
  - Most other instructions are the same from a datapath viewpoint
  - The ones that aren't are left for you to figure out

Remember The von Neumann Model?



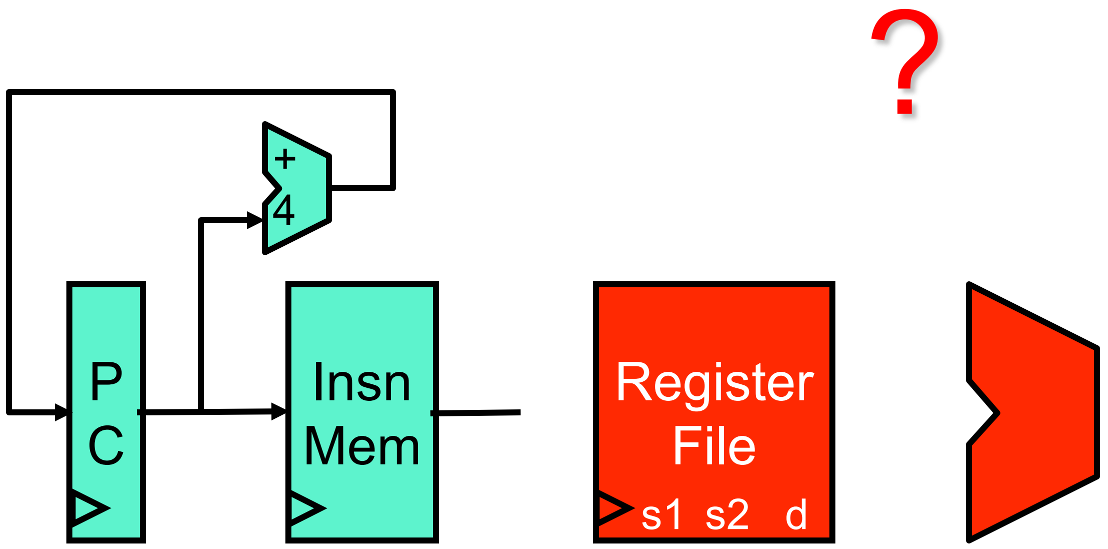
- Instruction Fetch:
  Read instruction bits from memory
- Decode:
  Figure out what those bits mean
- Operand Fetch:
  Read registers (+ mem to get sources)
- Execute:
  Do the actual operation (e.g., add the numbers)
- Result Store:
  Write result to register or memory
- Next Instruction:
  Figure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

Let’s Start With Insn Fetch



- Same for all instructions
- PC **register** and instruction memory
  - Details of Insn Mem (IMem): later…
  - For now, just assume a bunch of DFFs
- A “+4” incrementer computes default next instruction PC
- Processor has main clock
  - At every clock cycle, fetch a new insn based on PC

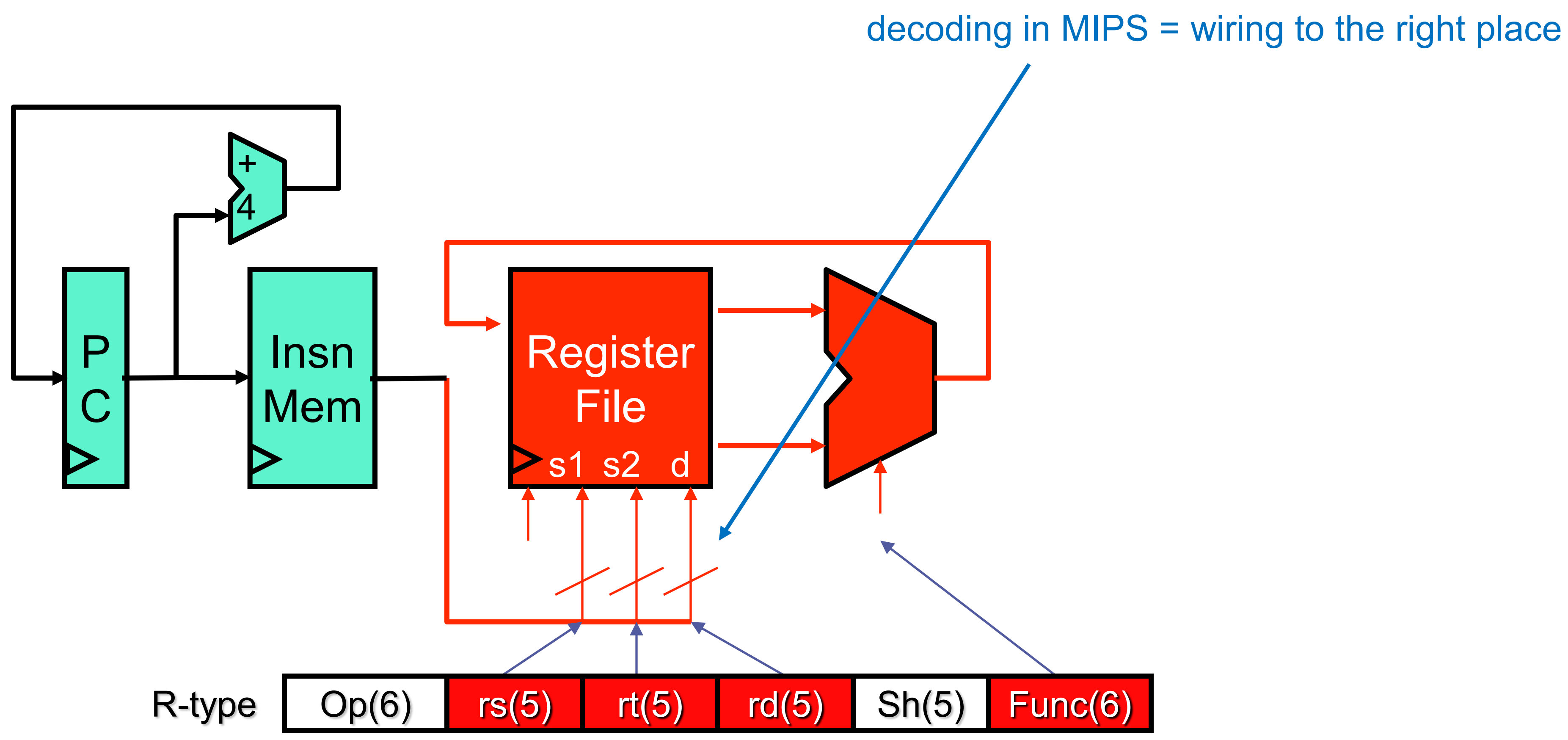
First Instruction: add $rd, $rs, $rt (e.g., `add $1,$2,$3`)



R-type	Op(6)	rs(5)	rt(5)	rd(5)	Sh(5)	Func(6)

```
add $d,$s,$t
```

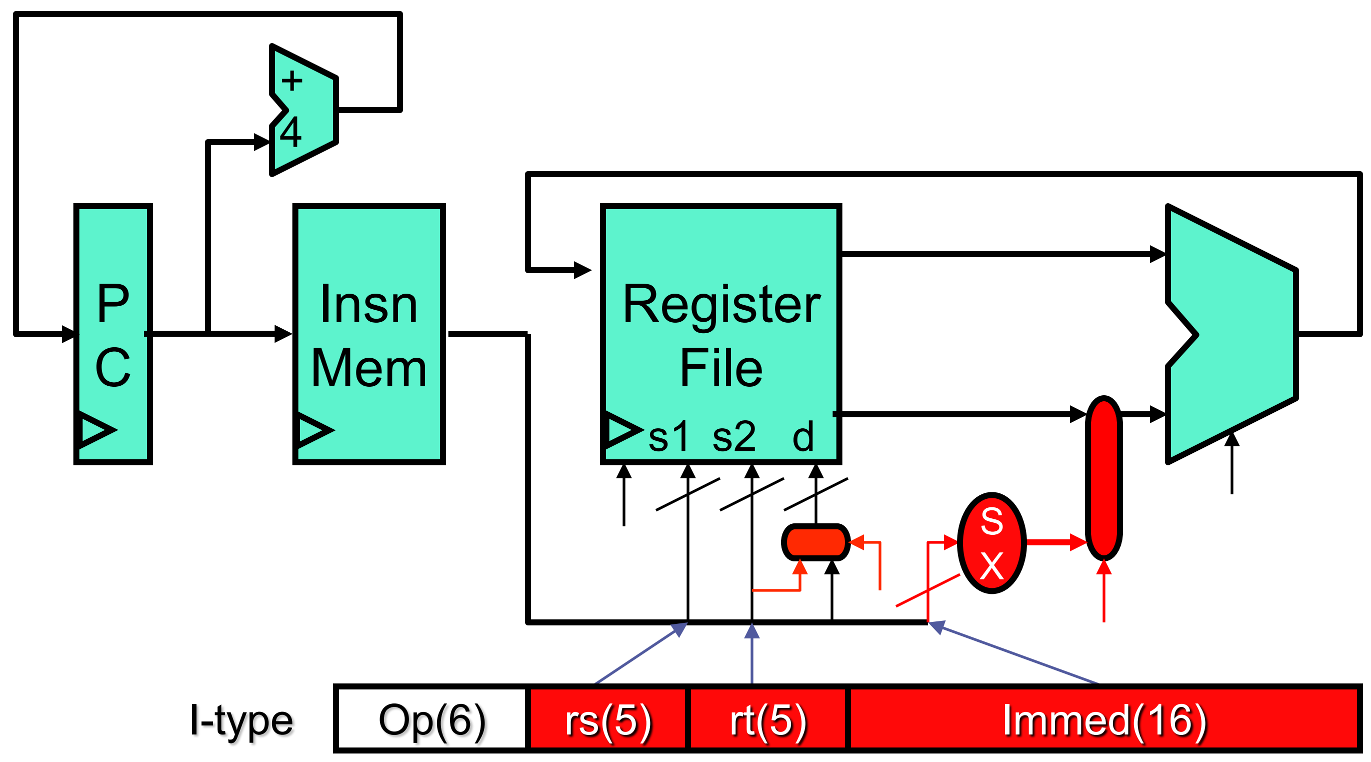
First Instruction: add $rd, $rs, $rt (e.g., `add $1,$2,$3`)



- Add register file and ALU
- *Control (unlabeled arrows) discussed later*

Second Instruction: addi $rt, $rs, imm (e.g., `addi $1,$2,50`)

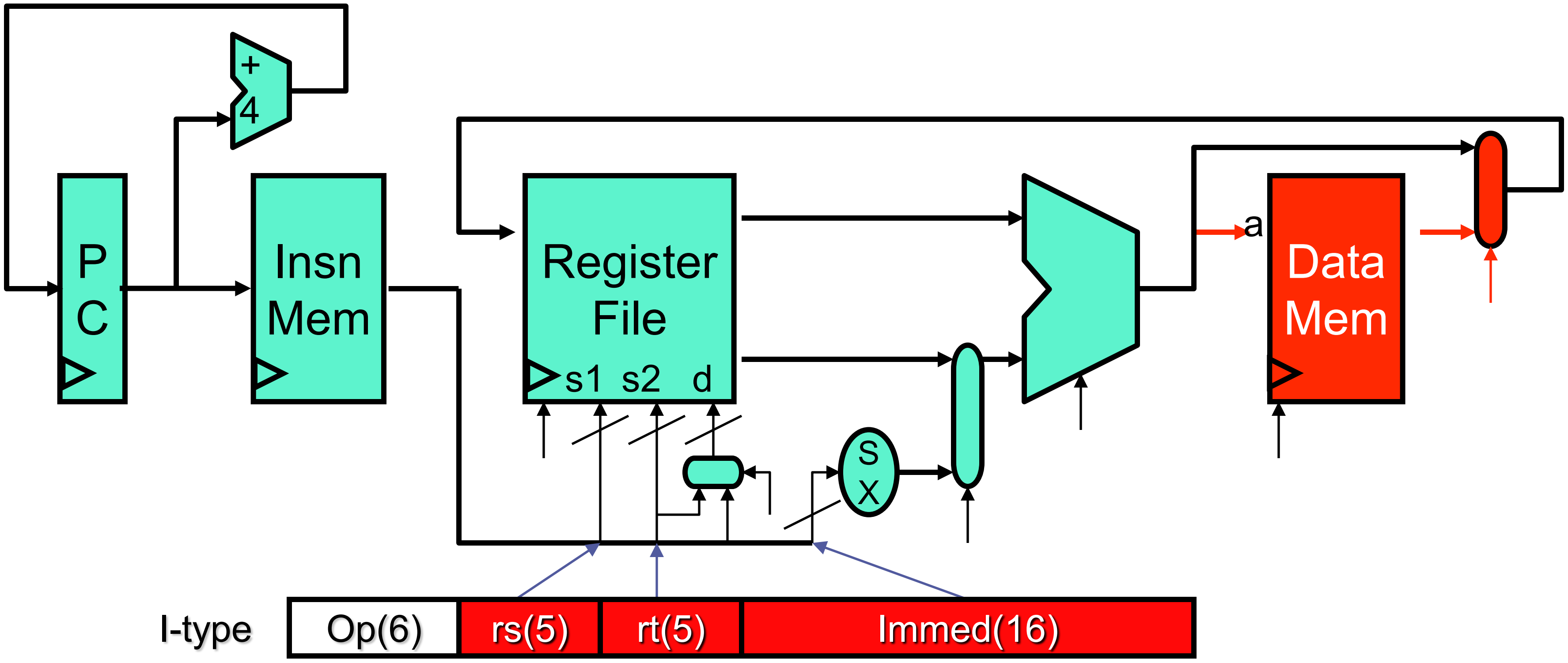
`addi $t,$s,imm`



- Destination register can now be either Rd (for R-type) or Rt (for I-type)
  - → use mux to accommodate for both
  - Add sign extension (SX) unit (adds 0's or 1's accordingly at the beginning to make it 32 bits), and mux into second ALU input

Third Instruction: lw $rt, imm($rs) (e.g., `lw $1,4($3)`)

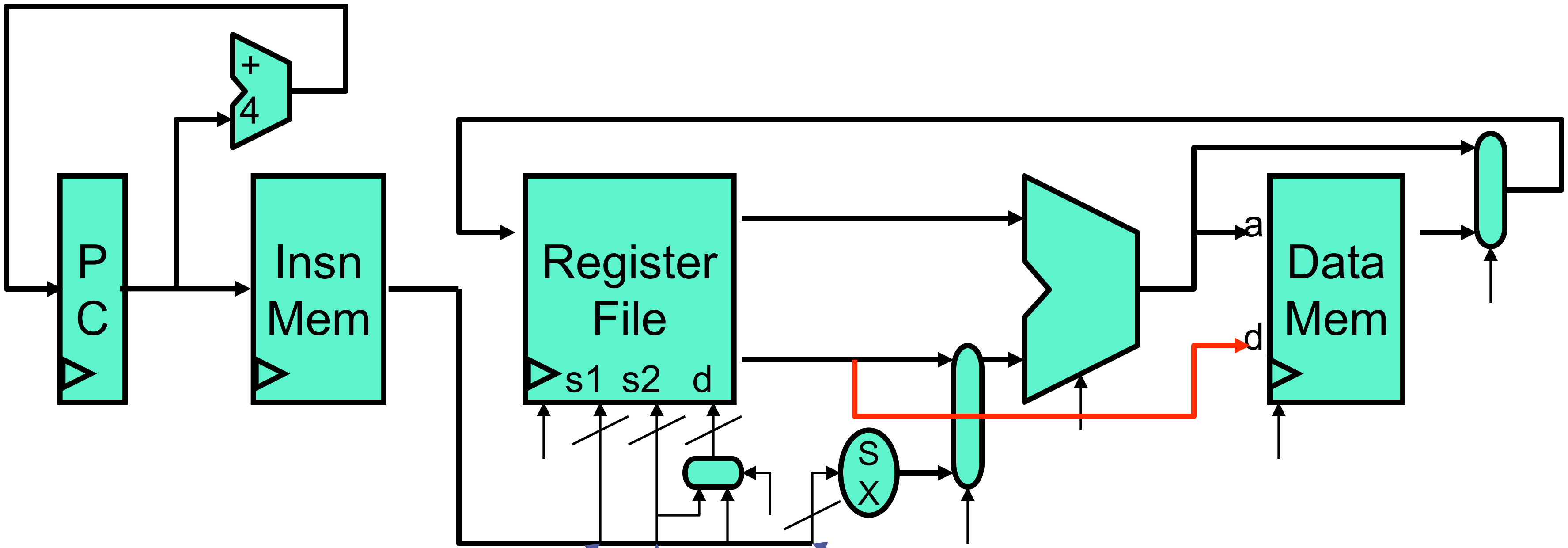
`lw $t,imm($s)`



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

Fourth Instruction: sw $rt, imm($rs) (e.g., `sw $1,4($3)`)

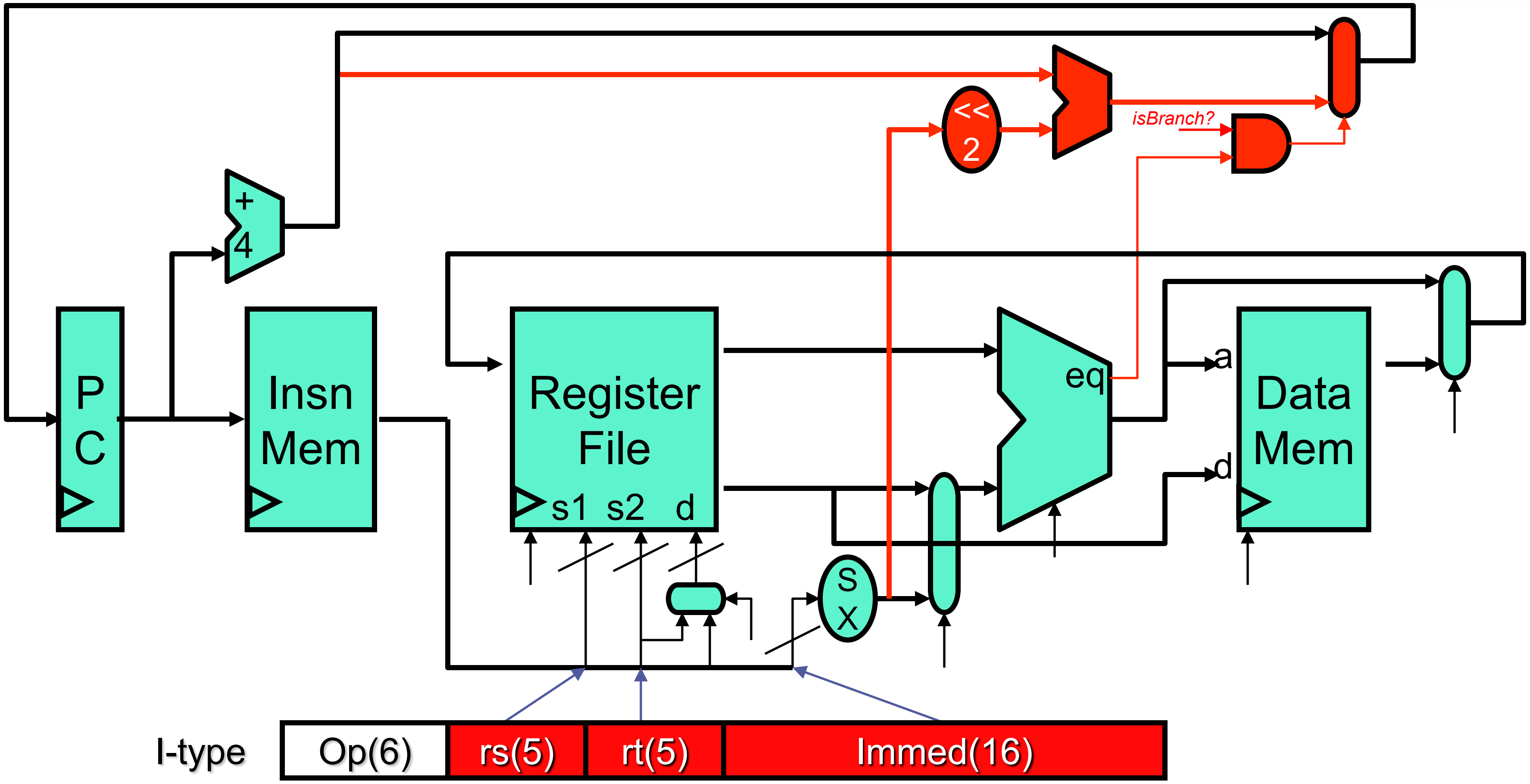
`sw $t,imm($s)`



I-type	Op(6)	rs(5)	rt(5)	Immed(16)

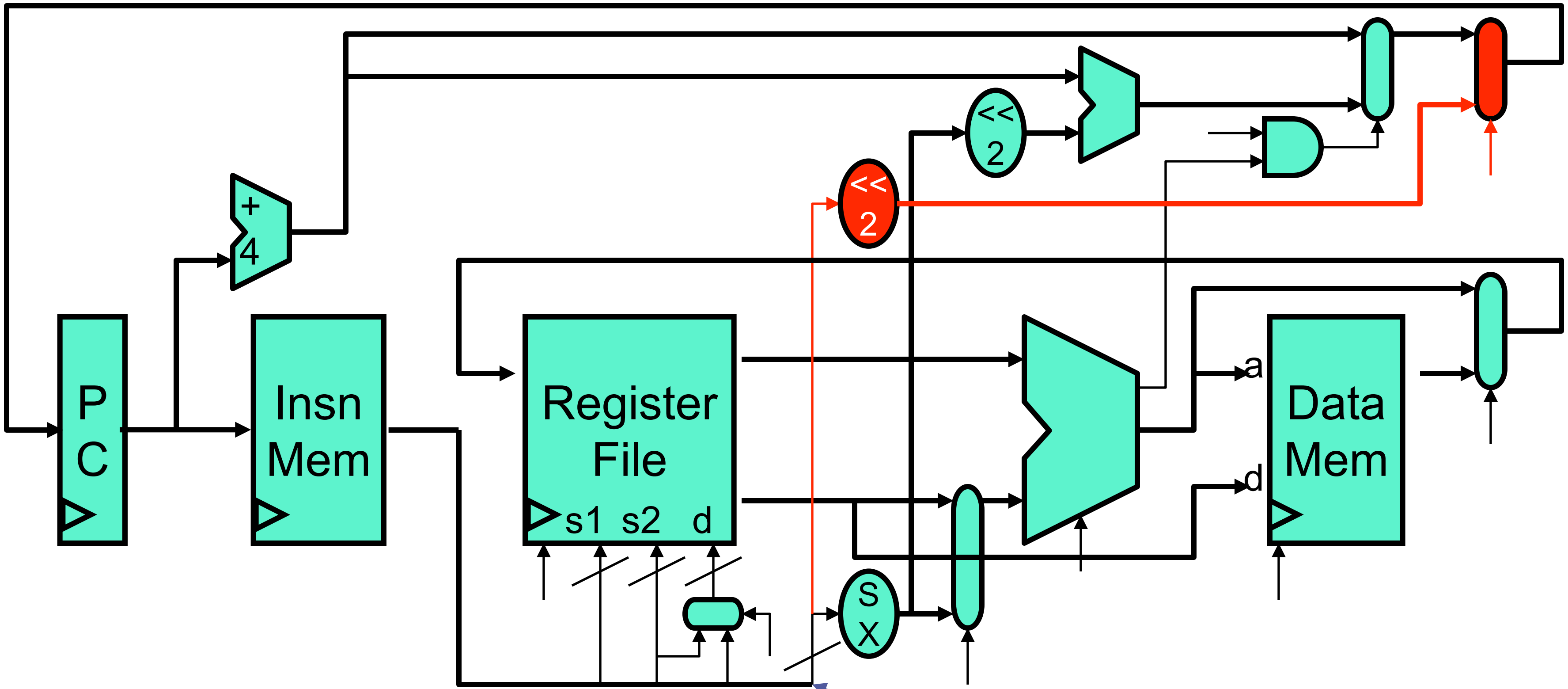
- Add path from second input register to data memory data input

Fifth Instruction: beq $rs, $rt, relative_target (e.g., `beq $1,$2,LOOP`)



- Add left shift unit (<<2) to bring back "00" we don't store and adder to compute PC-relative branch target
- Add PC input mux to select PC+4 or branch target
- Note that shifting by a fixed amount is very simple (just wires)

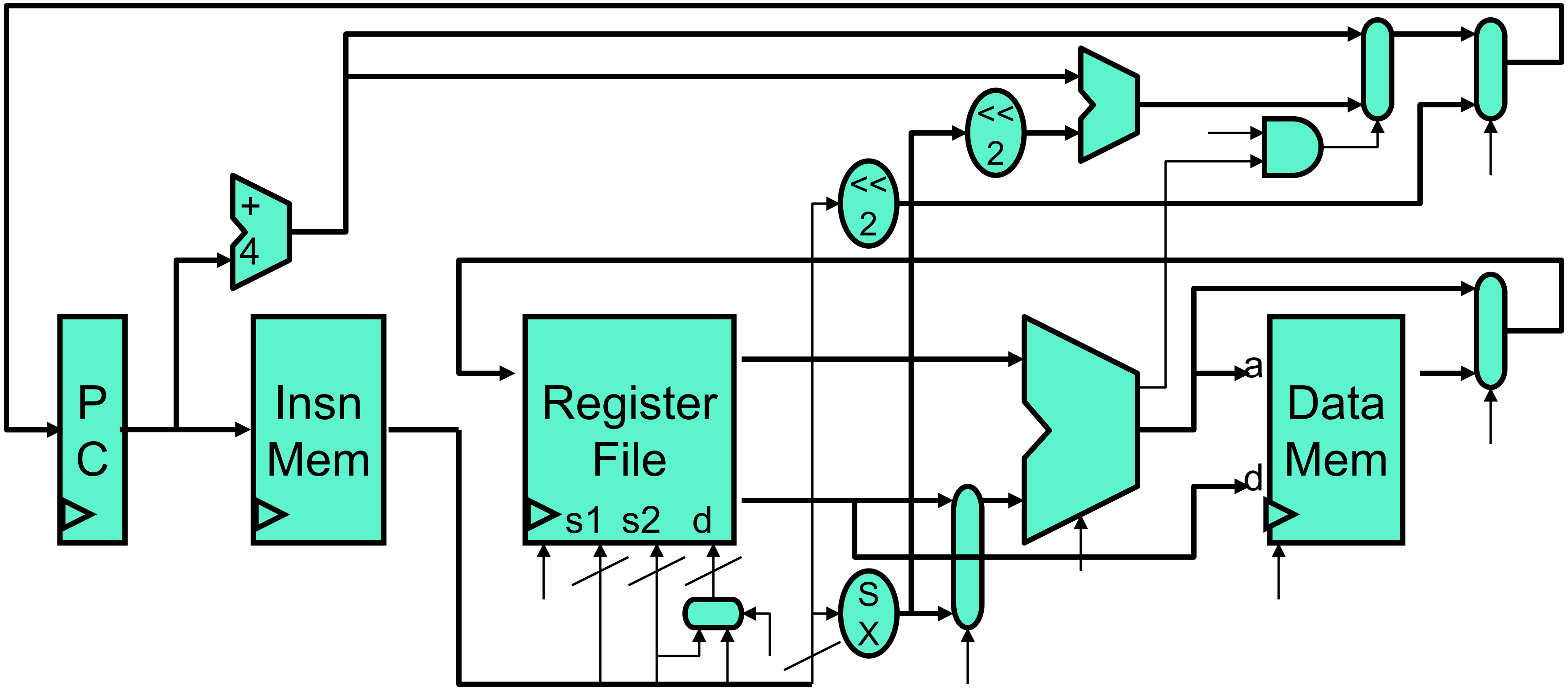
Sixth Instruction: j absolute_target (e.g., `j FUNCTION`)



J-type	Op(6)	Immed(26)

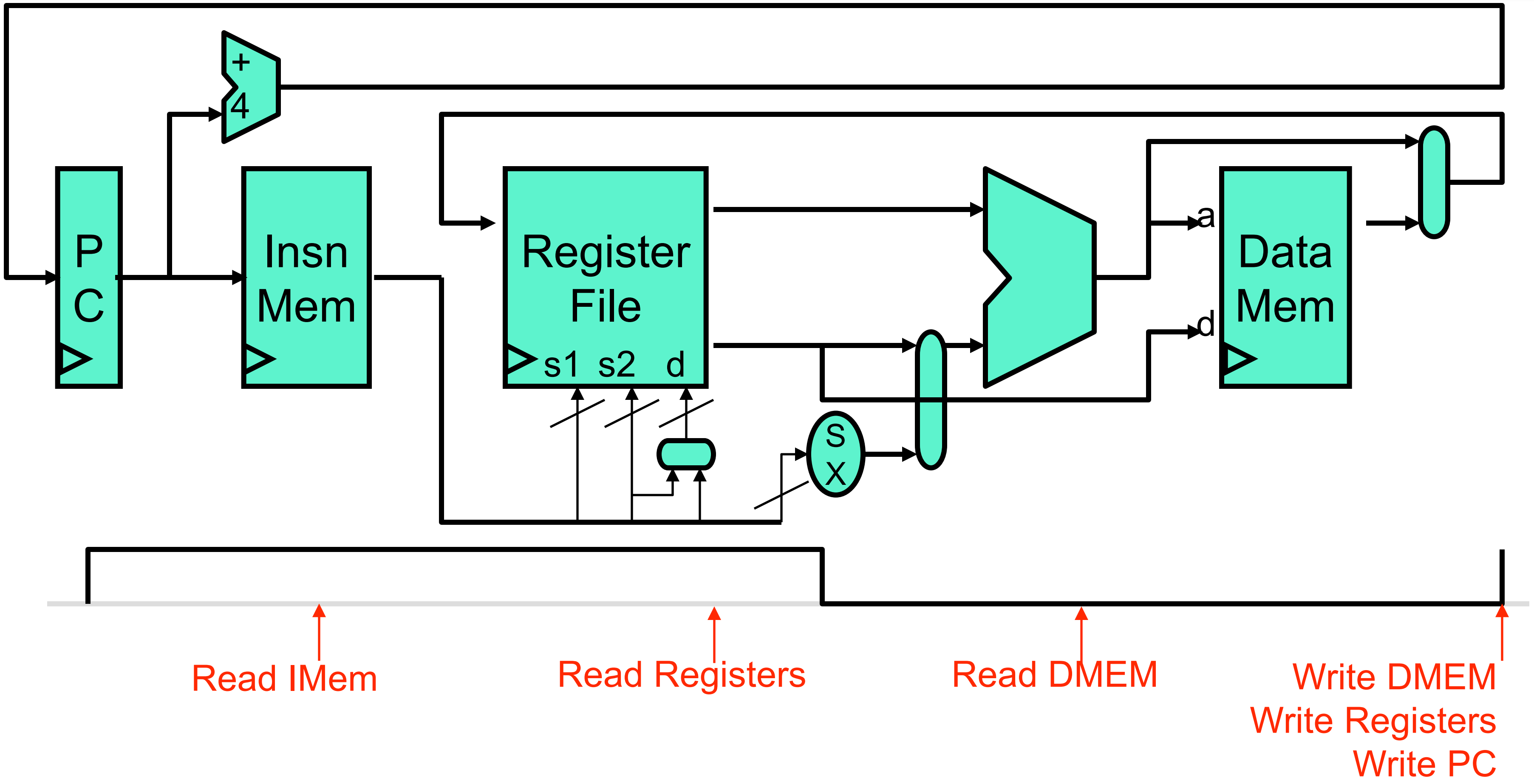
- Add shifter to compute left shift of 26-bit immediate
- Add additional PC input mux for jump target

Practice with More Instructions



- Pick other MIPS instructions and contemplate how to add them
  - jal (J-type)
  - jr (R-type)
  - …

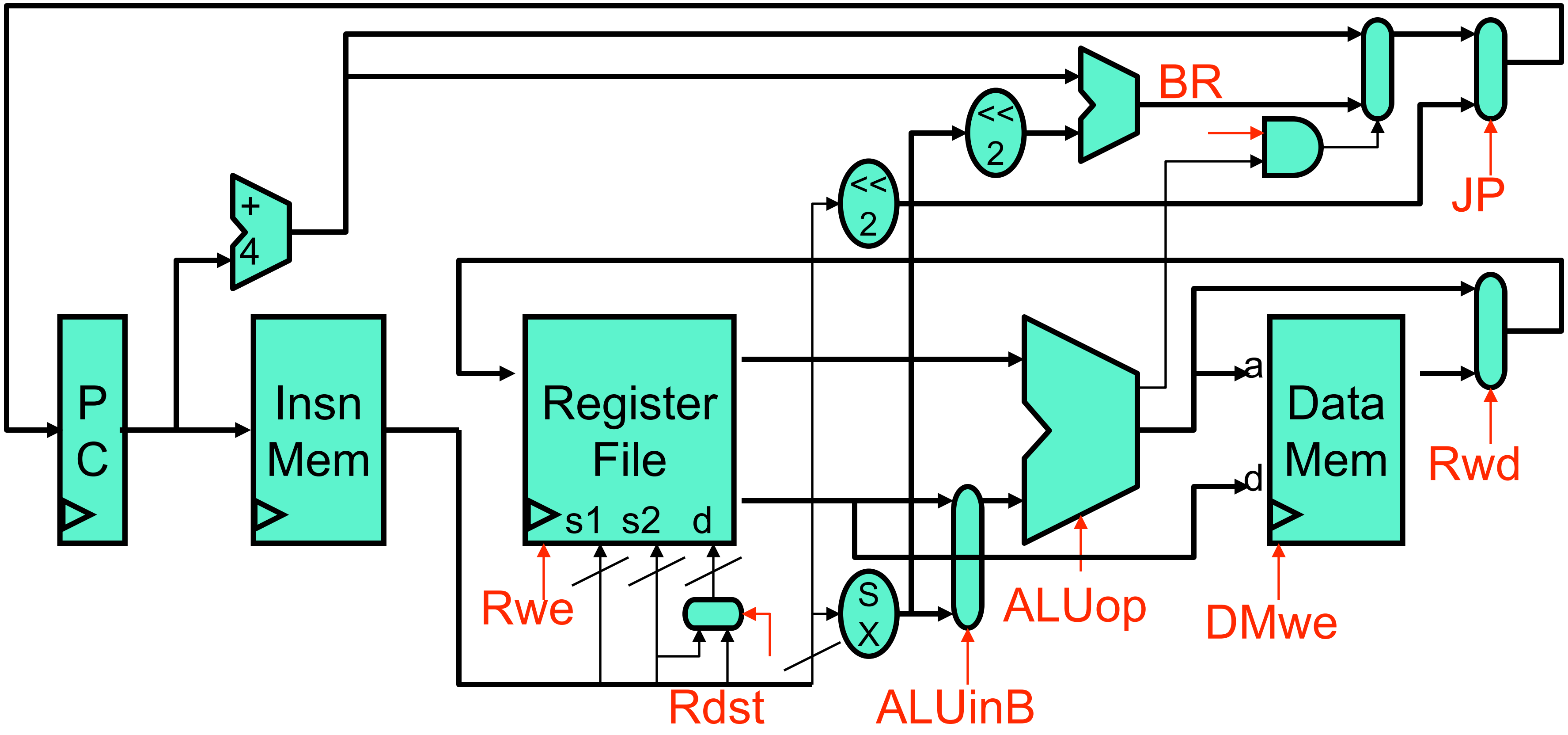
Datapath Timing



- Works because writes (PC, RegFile, DMem) are independent
- *We'll discuss data dependencies later*

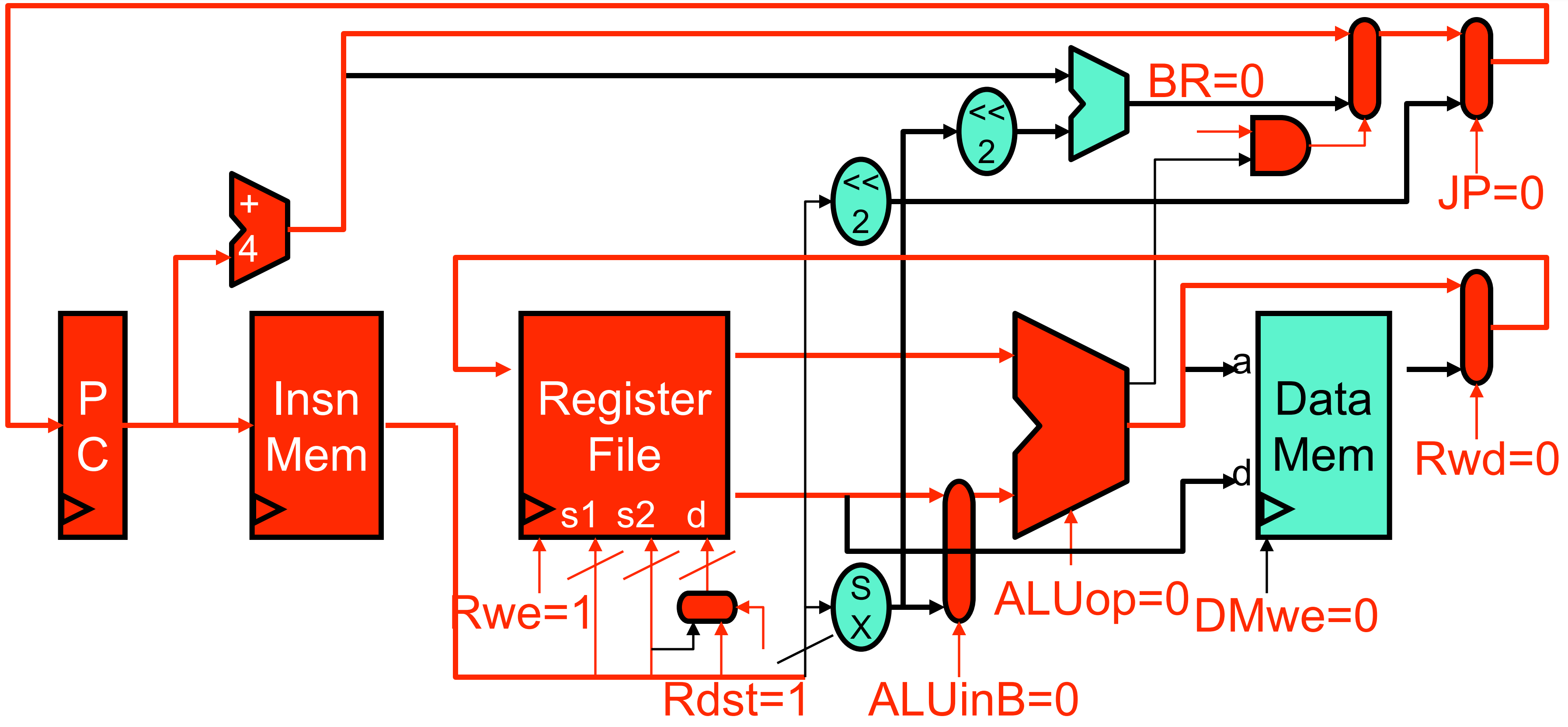
Datapath Control

What Is Control?



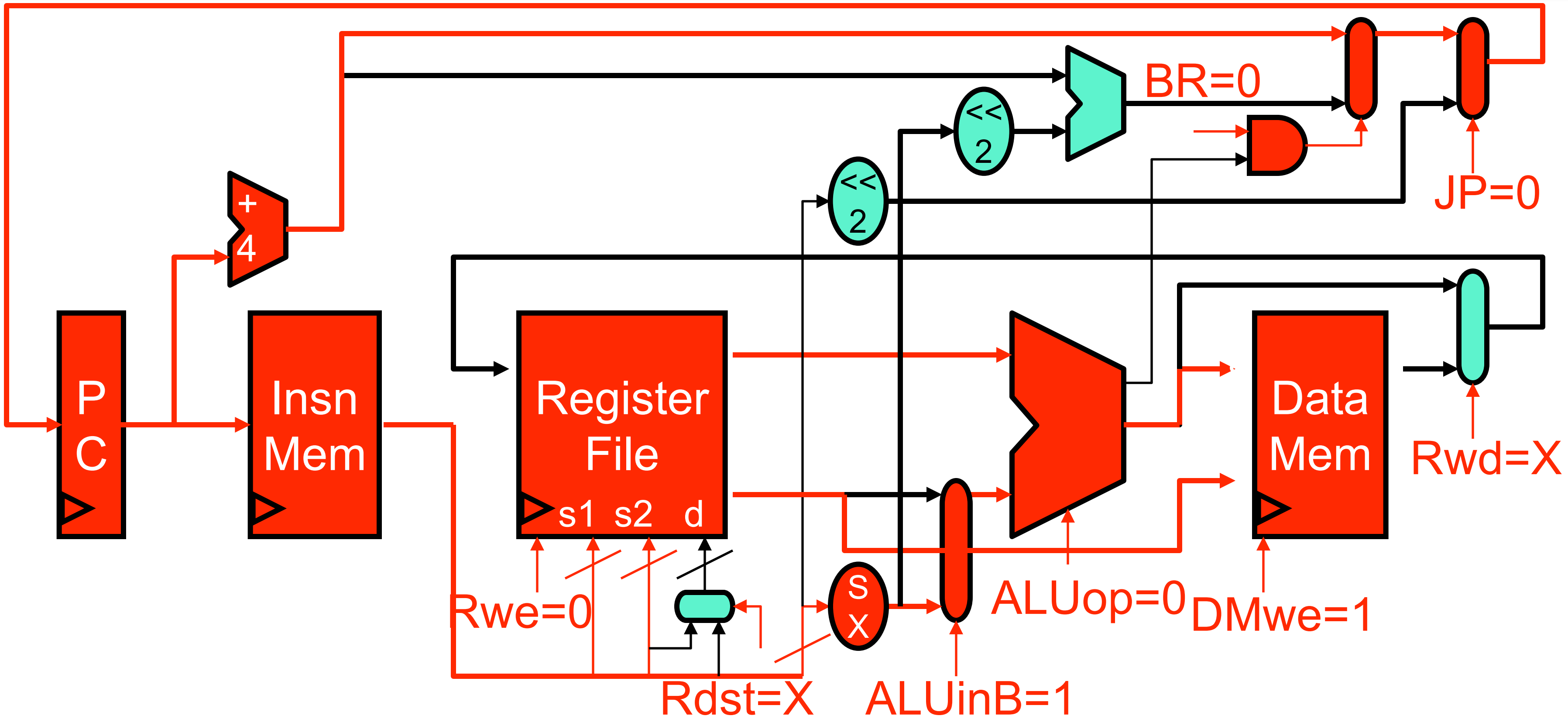
- 8 signals control flow of data through this datapath
  - MUX selectors, or register/memory write enable ("we") signals
  - A real datapath has hundreds of control signals

Example: Control for add



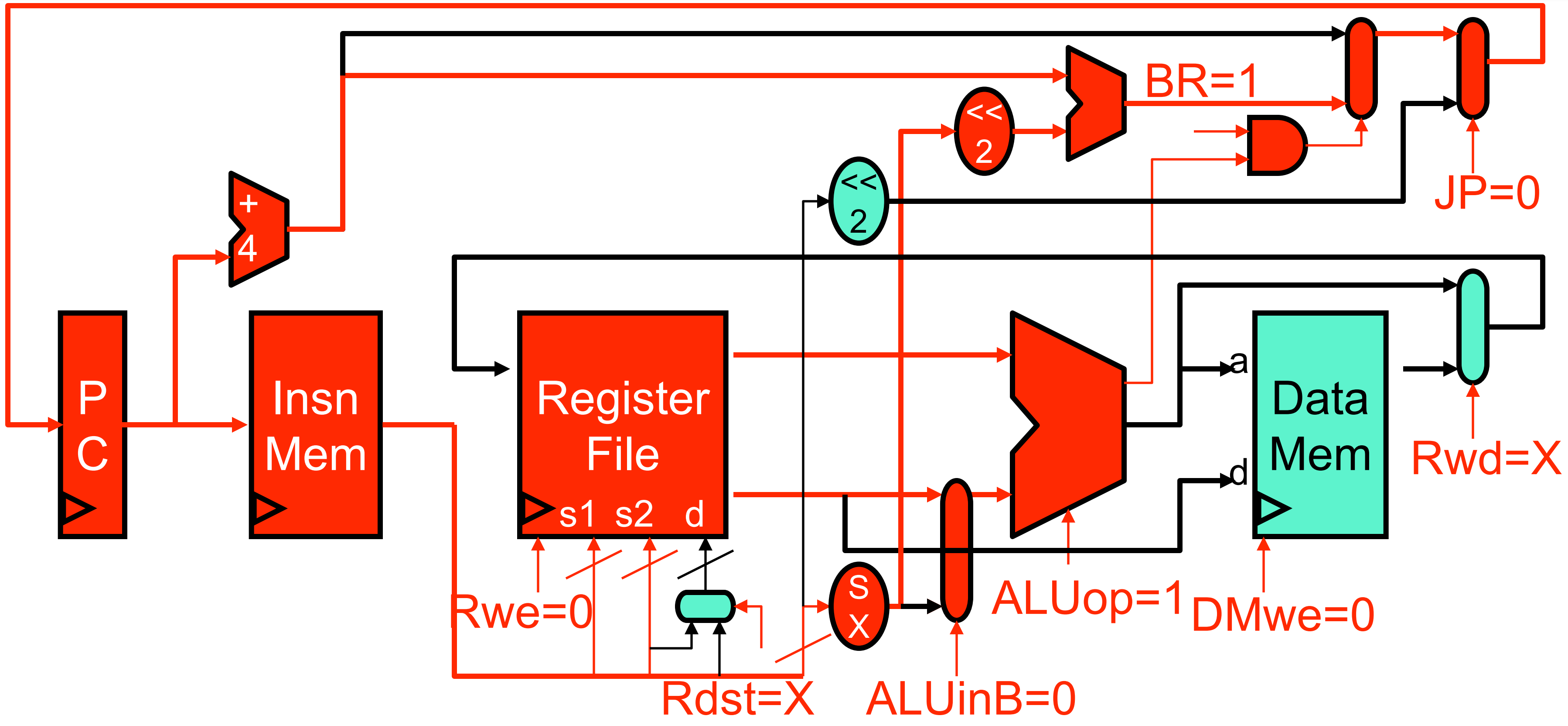
- Note that, for muxes, we're assuming that a select value of '0' selects the left input (for Rdst) or the top one (for the remaining muxes)
- Also note that, for ALUop, we're assuming that '0' is for the addition operation while '1' is for the equality check operation

Example: Control for sw

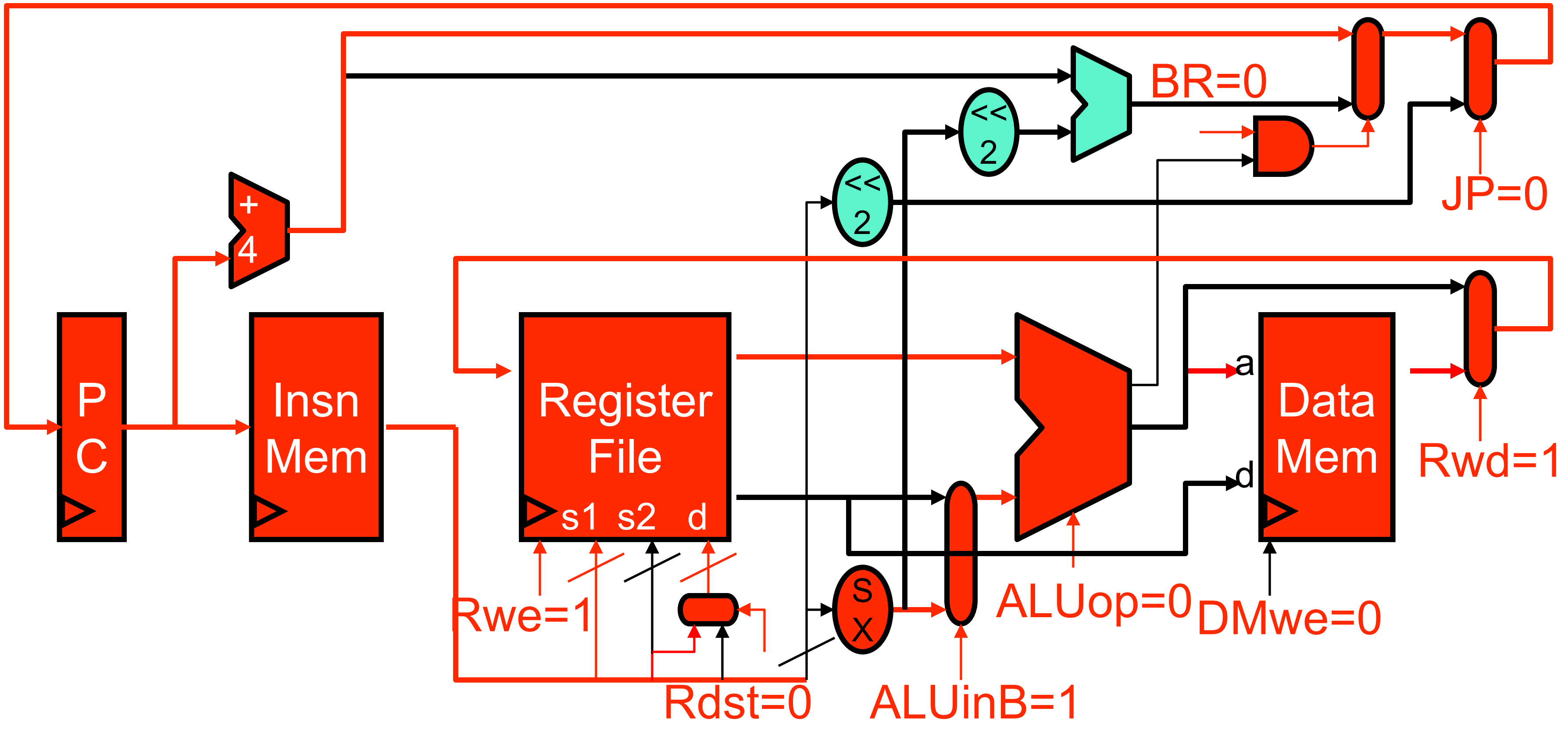


- 'X' stands for "don't care"
  - We don't case whether the signal is a '0' or a '1'
  - The signal can be treated as either a '0' or a '1' when making decisions on designing its circuitry

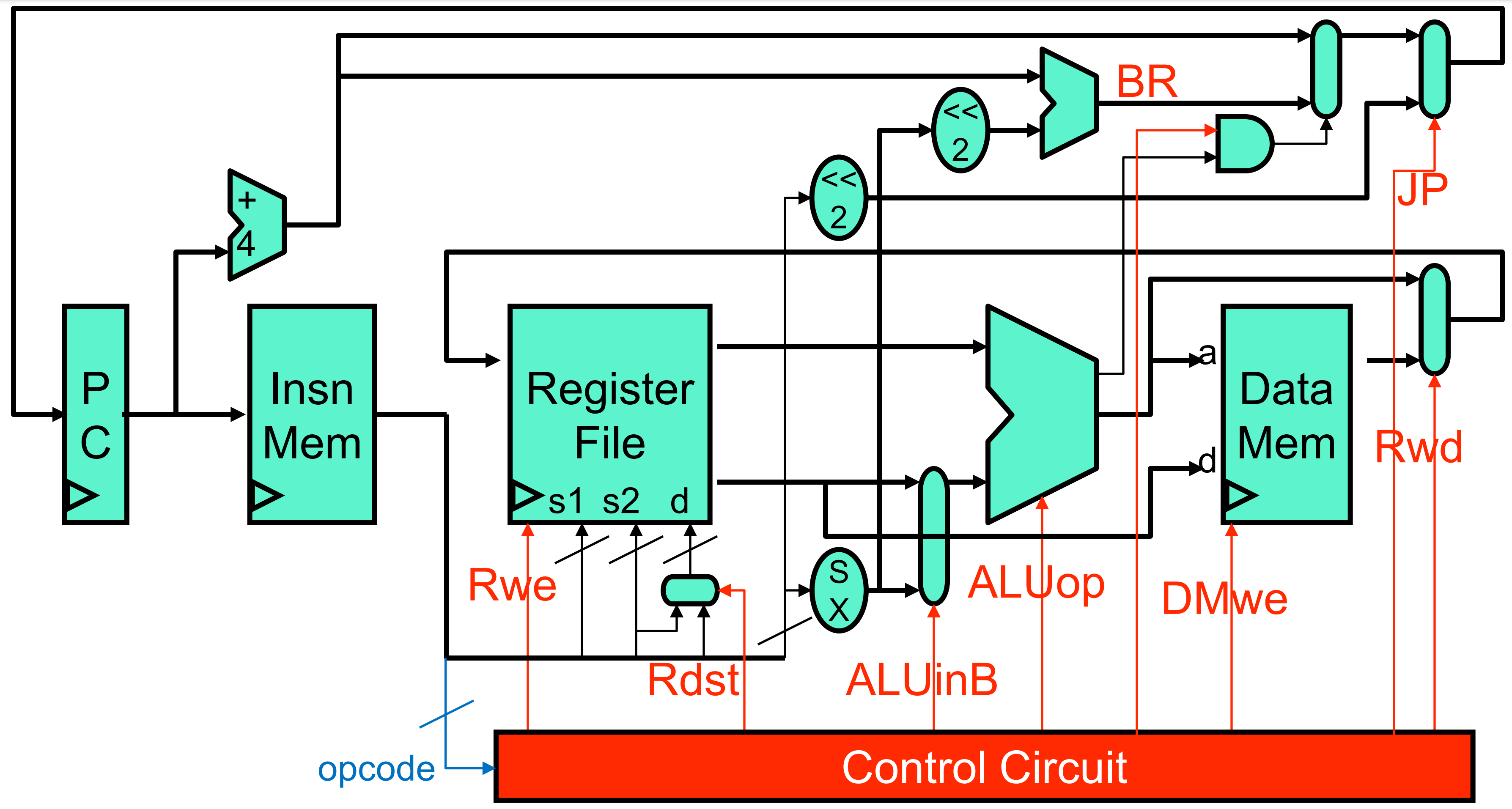
Example: Control for beq



Example: Control for lw



How Is Control Implemented?



Implementing Control

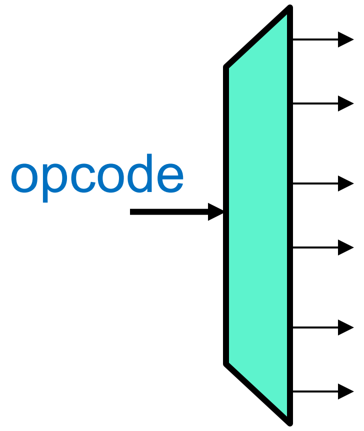
- Each insn has a unique set of control signals
  - Most are function of opcode
  - Some may be encoded in the instruction itself
    - E.g., the ALUop signal is some portion of the MIPS Func field
    + Simplifies controller implementation
    - Requires careful ISA design

- In our datapath, the control unit has the following interface:
  - Inputs: 6 bits for the opcode
  - Outputs: the 8 control signal bits
- → We can implement the control unit using the 3-step method
  1. Truth table
  2. Boolean expressions for every output (could simplify)
  3. Translate into a circuit
- *OR we could do it in other ways that could be more optimal for our desired metrics*

Control Implementation Using a ROM

- **ROM (read-only memory)**: like rows of bits
  - Bits in data words are control signals
  - Lines indexed by opcode

  - Example: ROM control for our 6-insn MIPS datapath:



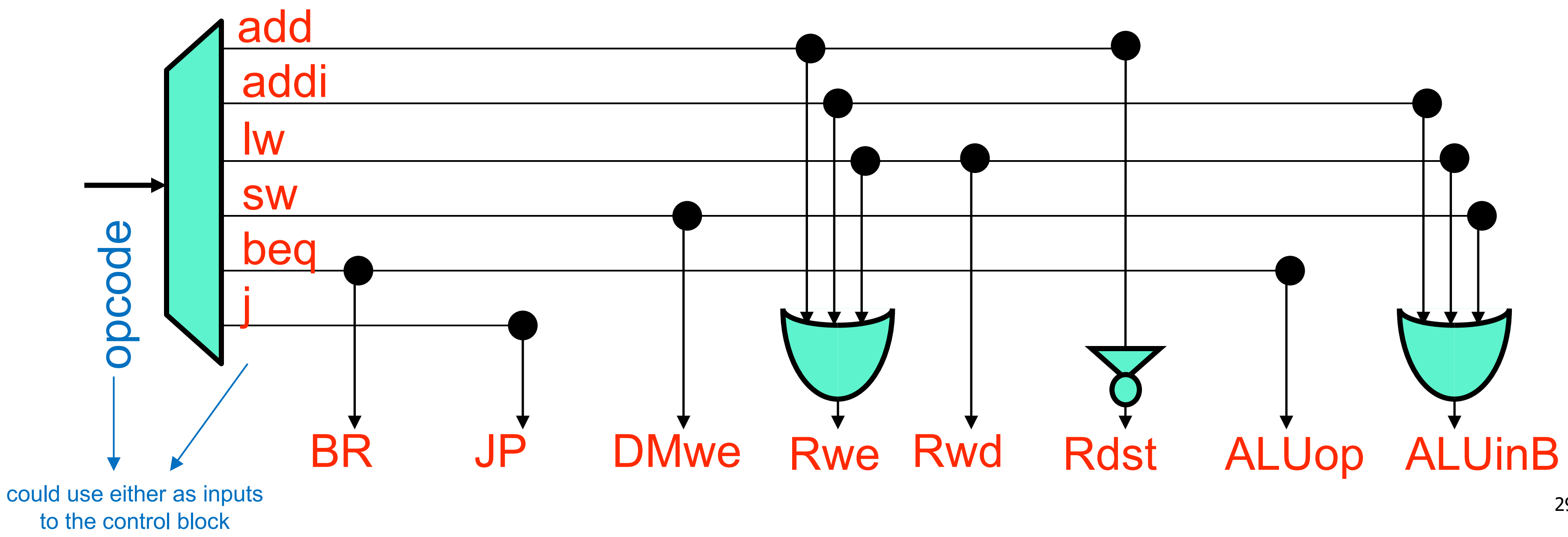
	BR	JP	ALUinB	ALUop	DMwe	Rwe	Rdst	Rwd
add	0	0	0	0	0	1	1	0
addi	0	0	1	0	0	1	0	0
lw	0	0	1	0	0	1	0	1
sw	0	0	1	0	1	0	X	X
beq	1	0	0	1	0	0	X	X
j	X	1	X	X	0	0	X	X

Control Implementation Using "Random Logic"

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

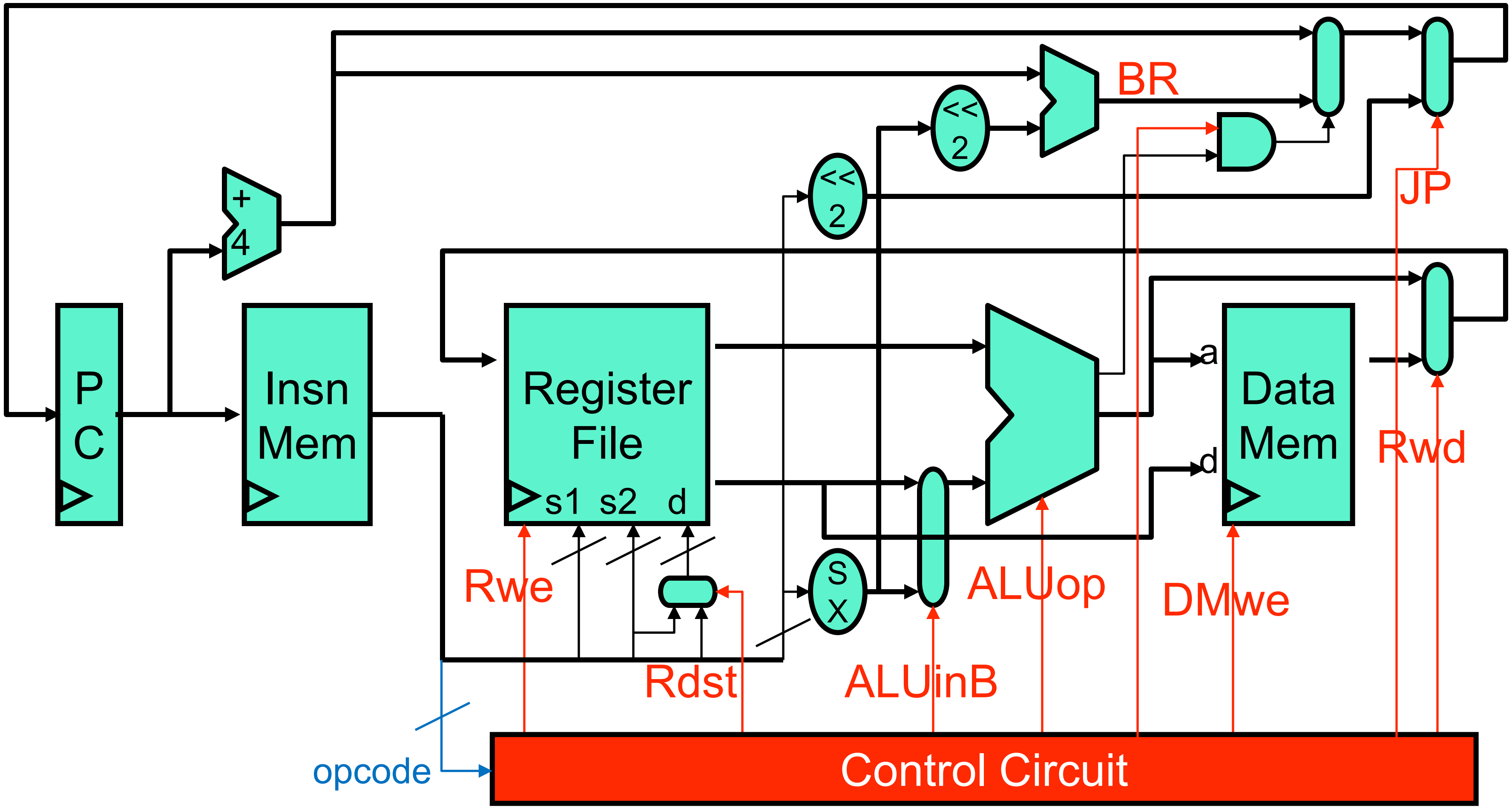
Yes, "random logic" is a very misleading name for this concept.

- Example: random logic control for our 6-insn MIPS datapath:



Performance

"Single-Cycle Datapath" Performance



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

Performance

- Three components to performance (in terms of speed):

  Number of instructions

  x Cycles per instruction (CPI)

  x Clock Period (1 / Clock frequency)

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

$\downarrow$

Insns/Program: determined by compiler + ISA

Micro-Architectural Factors

- Micro-architecture:
  - The details of how the ISA is implemented
  - Impacts CPI and Clock frequency

- Often will look at fixed program, and consider **MIPS**
  - *Million Instructions Per Second*
  - MIPS = IPC * Frequency (in MHz)
  - IPC = Instruction Per Cycle (1 / CPI)
  - Gives a "bigger is better" number

The use of "MIPS" to mean "Millions of Instructions Per Second" has nothing to do with the CPU architecture also called "MIPS", which stands for "Microprocessor without Interlocked Pipeline Stages".

$$\underset{\textbf{(IPC)}}{\frac{\text{Instructions}}{\text{Cycle}}} \times \underset{\textbf{(Frequency)}}{\frac{\text{Cycles}}{\text{Second}}} = \underset{\substack{\textbf{(Throughput)} \\ \text{(instead of speed)}}}{\frac{\overset{\text{(Million)}}{\text{Instructions}}}{\text{Second}}}$$

“Best” IPC

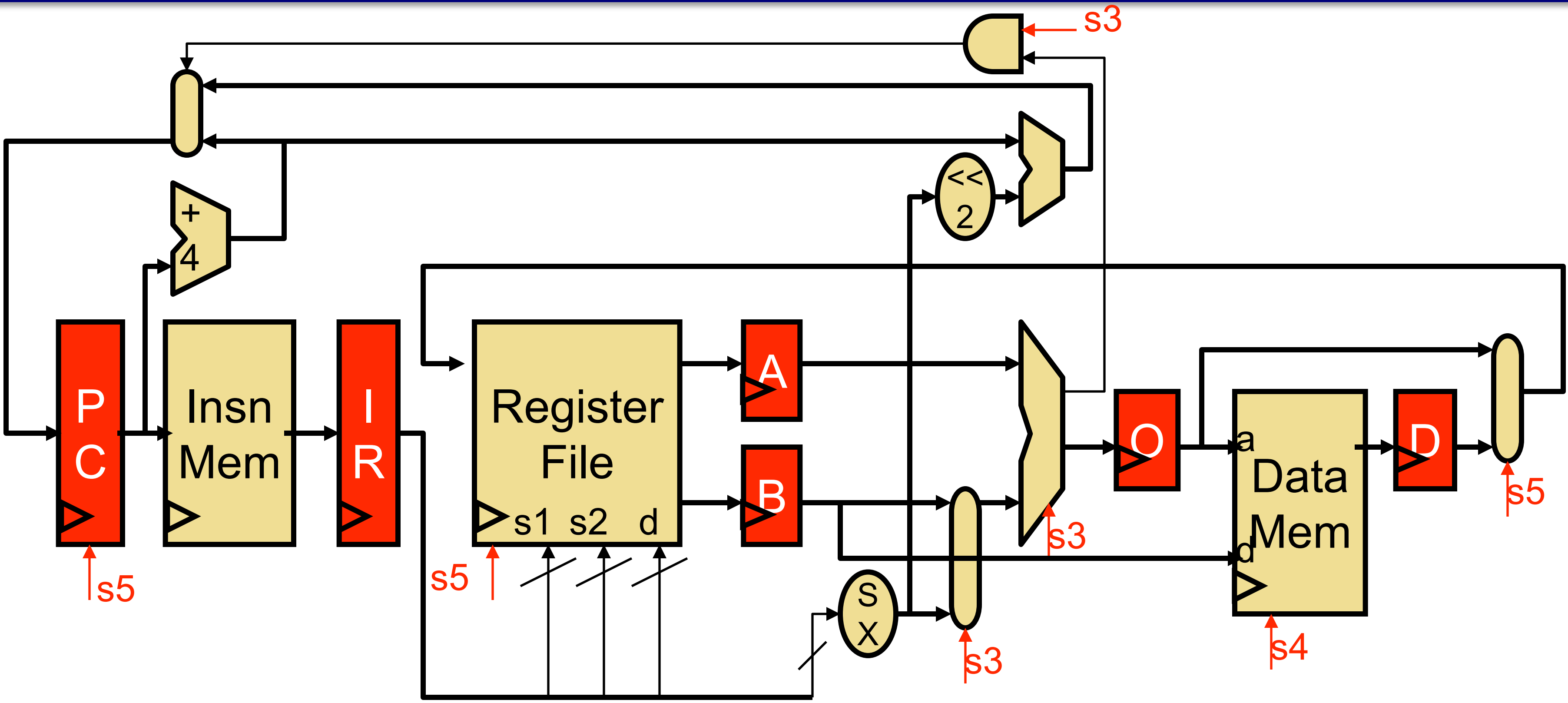
- For now, the best we can do is: IPC=1 (CPI=1)
  - Do 1 instruction every cycle
- Later:
  - Real processors can do multiple instructions at once!
  - Potentially: IPC>1! (CPI<1!)
  - Best possible IPC depends on design

Other Metrics

- 1990s: Performance at all cost
  - Actually, more like “higher clock frequency” at all cost
- Now: Care about other things
  - **Energy** (electric bill, battery life)
  - **Power** (cooling, also impacts energy)
  - Area (chip cost)
  - Reliability (tolerance of transient faults: e.g., charge particle strikes)
  - **Performance/Watt** (throughput divided by power consumption)
  - …

- Speaking of performance…
  - Making a processor takes time (years) and money (millions)
  - Want to know it will perform well before you finish
    - If it’s wrong, doing it all over is painful
  - **Performance can be simulated in software using different modeling techniques to approximate components’ metrics**

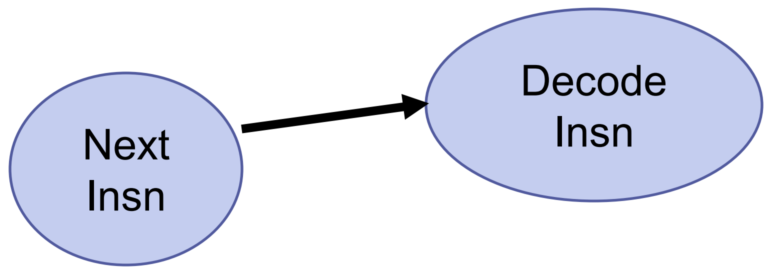
One Alternative to Single-Cycle DP: Multi-Cycle DP



- Cuts datapath into multiple stages (5 here), isolated using registers
- **FSM** control "walks" insns thru stages (by staging control signals)

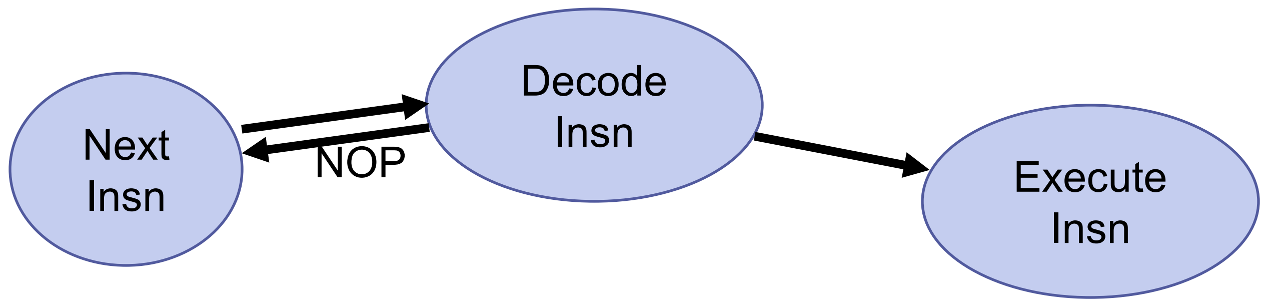
+ **Insns can bypass stages and exit early**

Multi-Cycle Datapath FSM



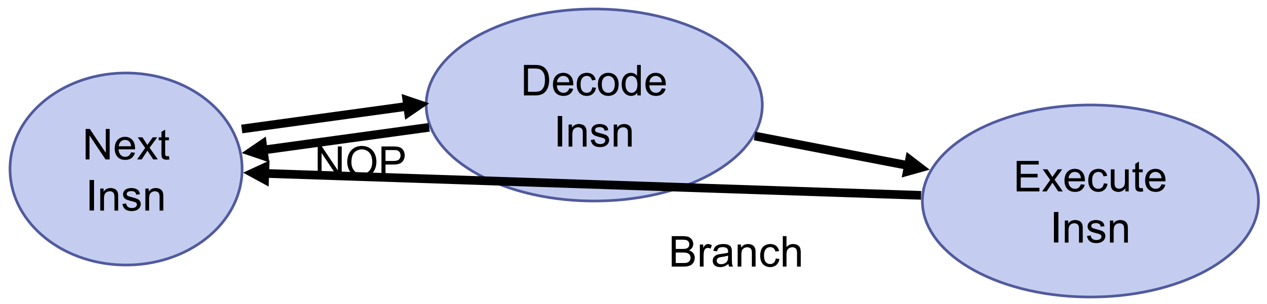
- First state: Get a New Instruction
  - Output signals to fetch (e.g., read enable IMEM)
  - Next State: Always Decode

Multi-Cycle Datapath FSM



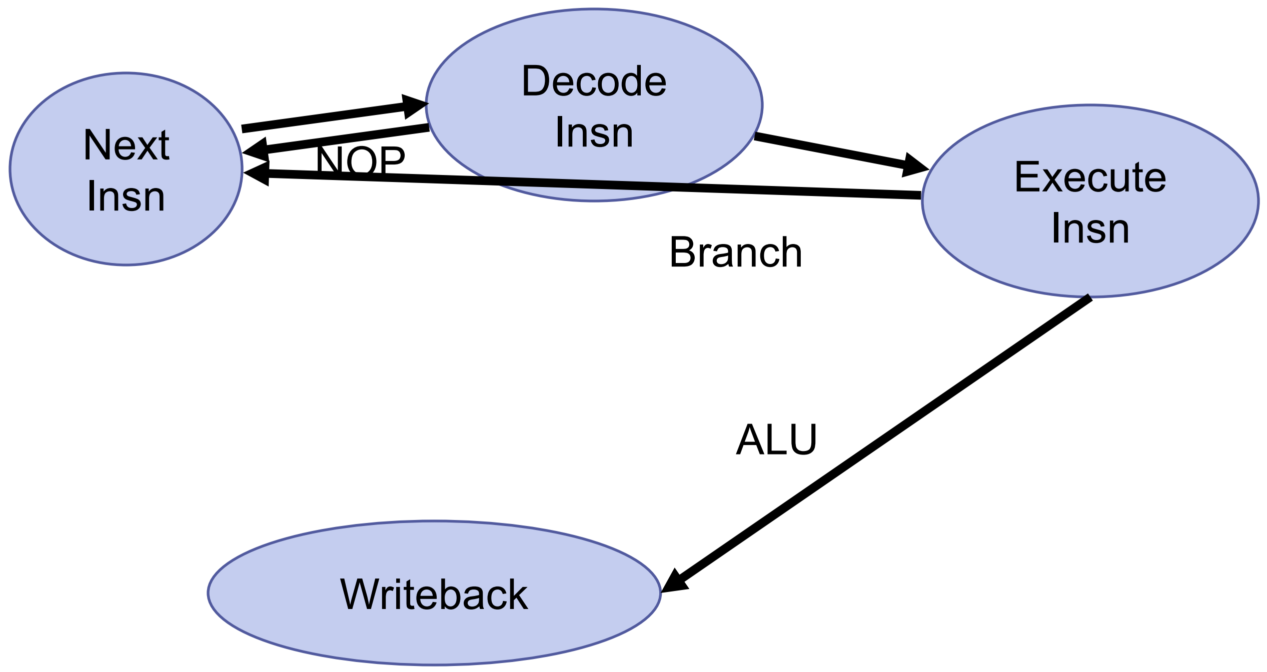
- Second State: Decode
  - Output signals to decode instruction (RdEn RegFile)
  - Go to Next Insn if NOP (this is a MIPS insn meaning "no operation")
  - Otherwise Execute

Multi-Cycle Datapath FSM



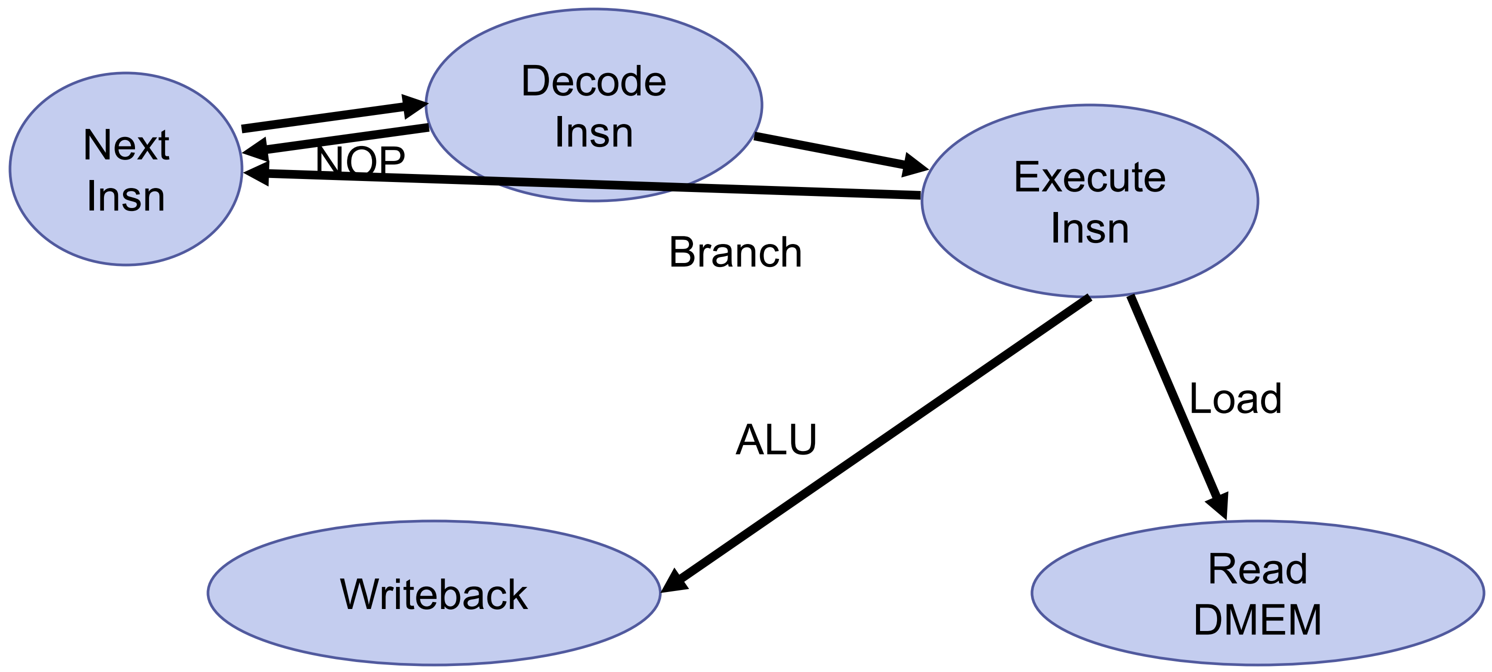
- Execute State
  - Execute Insn (varies by insn type)
  - Next State: Also depends on insn type
    - Branches: Next Insn

Multi-Cycle Datapath FSM



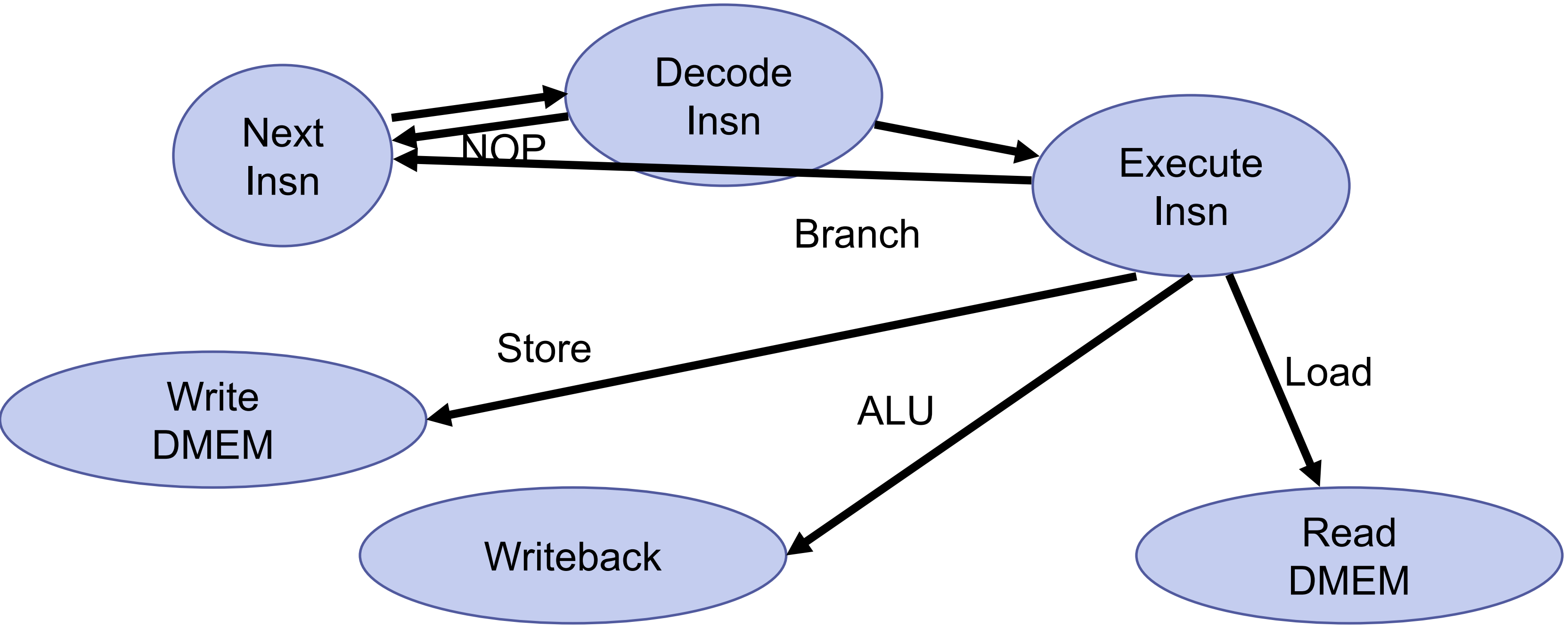
- Execute State
  - Execute Insn (varies by insn type)
  - Next State: Also depends on insn type
    - ALU op: write register

Multi-Cycle Datapath FSM



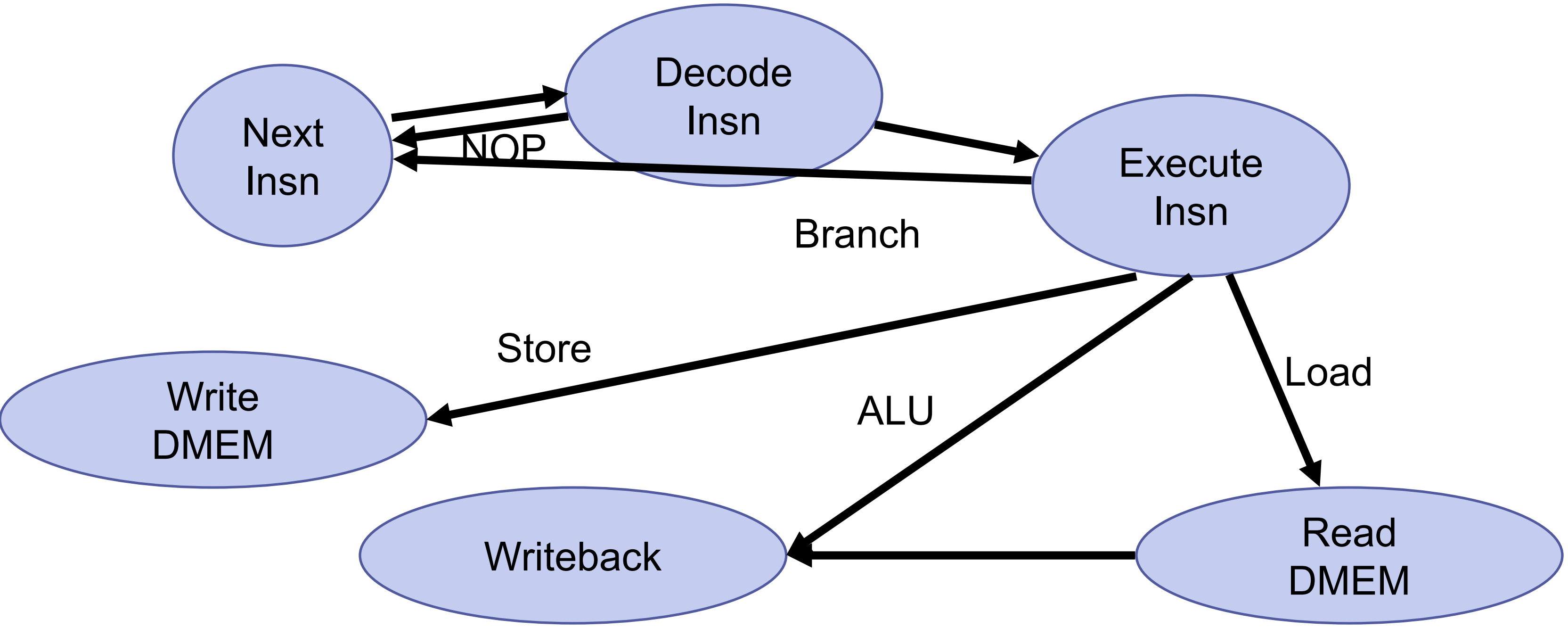
- Execute State
  - Execute Insn (varies by insn type)
  - Next State: Also depends on insn type
    - Load: Read Memory

Multi-Cycle Datapath FSM



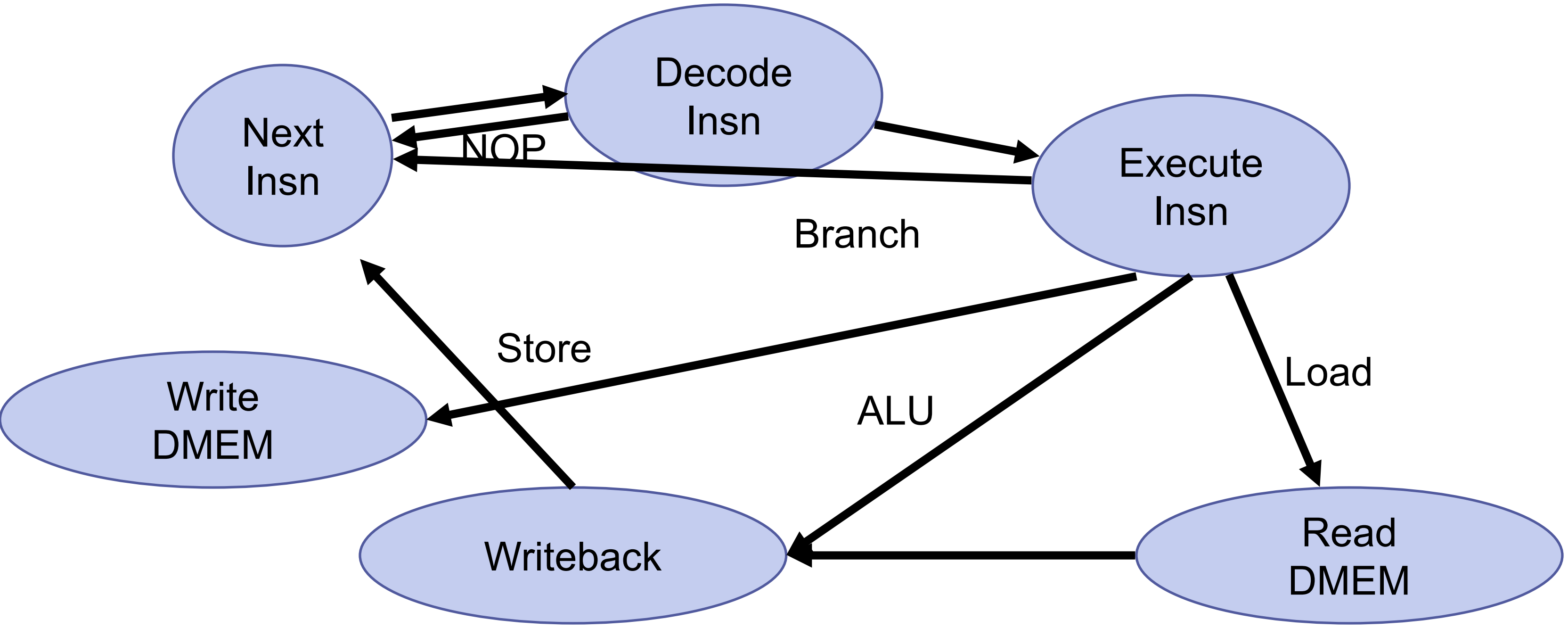
- Execute State
    - Execute Insn (varies by insn type)
    - Next State: Also depends on insn type
        - Store: Write Memory

Multi-Cycle Datapath FSM



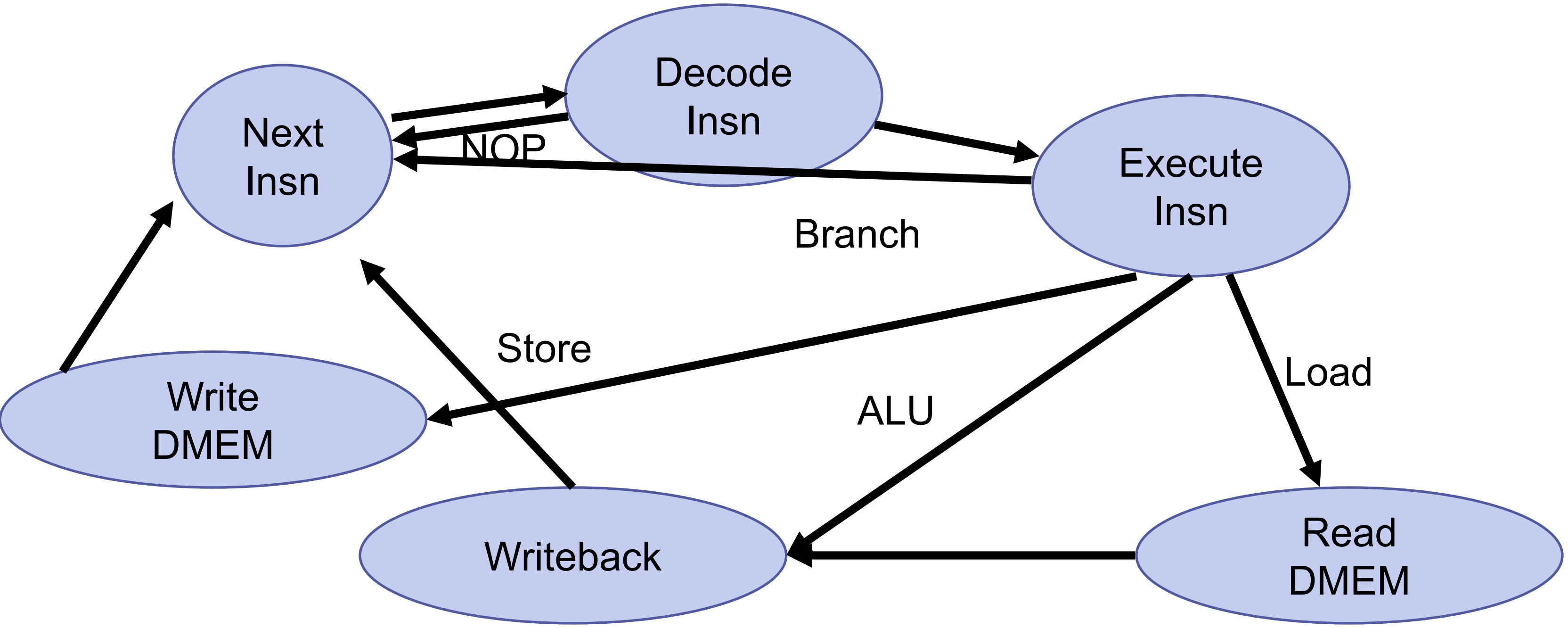
- Read DMEM State
    - Control signals enable DMEM Read
    - Next state is writeback

Multi-Cycle Datapath FSM



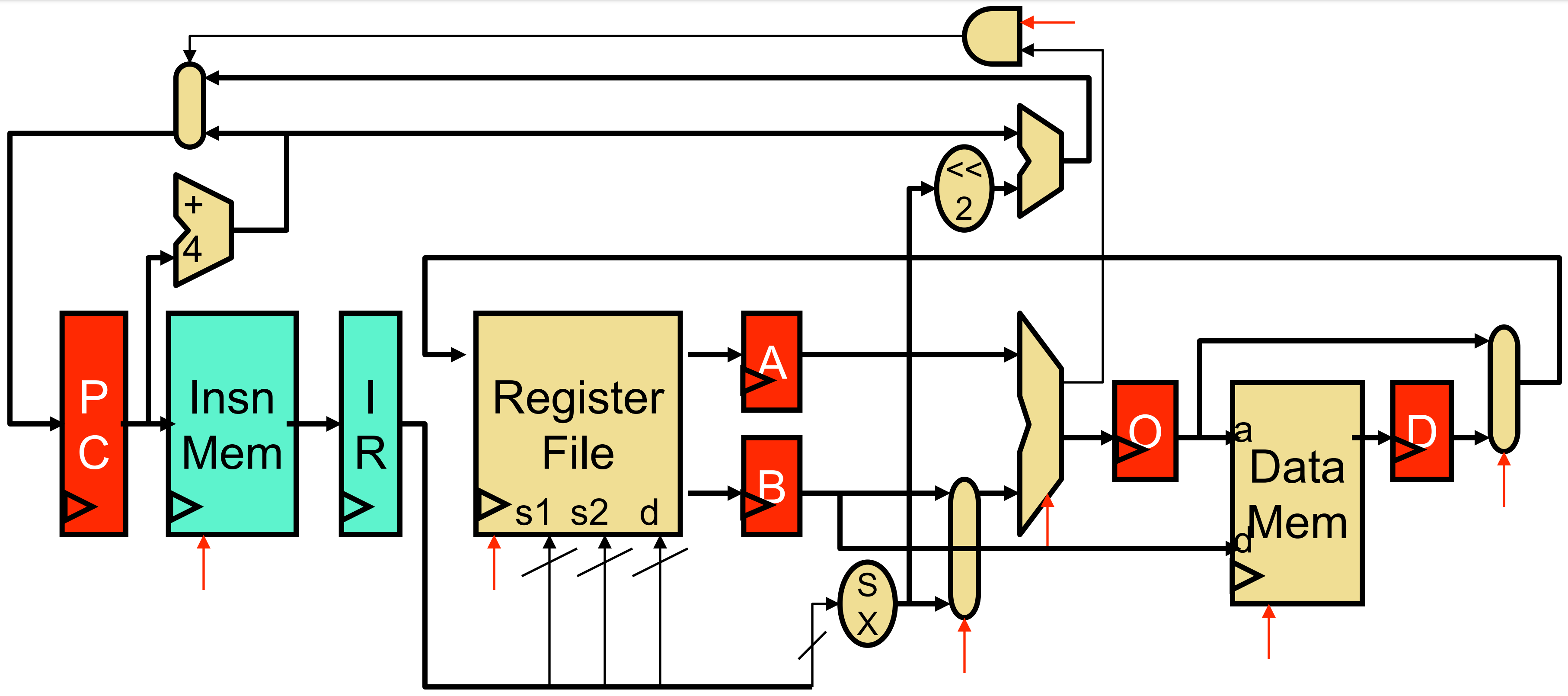
- Writeback state
  - Control signals enable regfile write
  - Next state: Next Insn

Multi-Cycle Datapath FSM



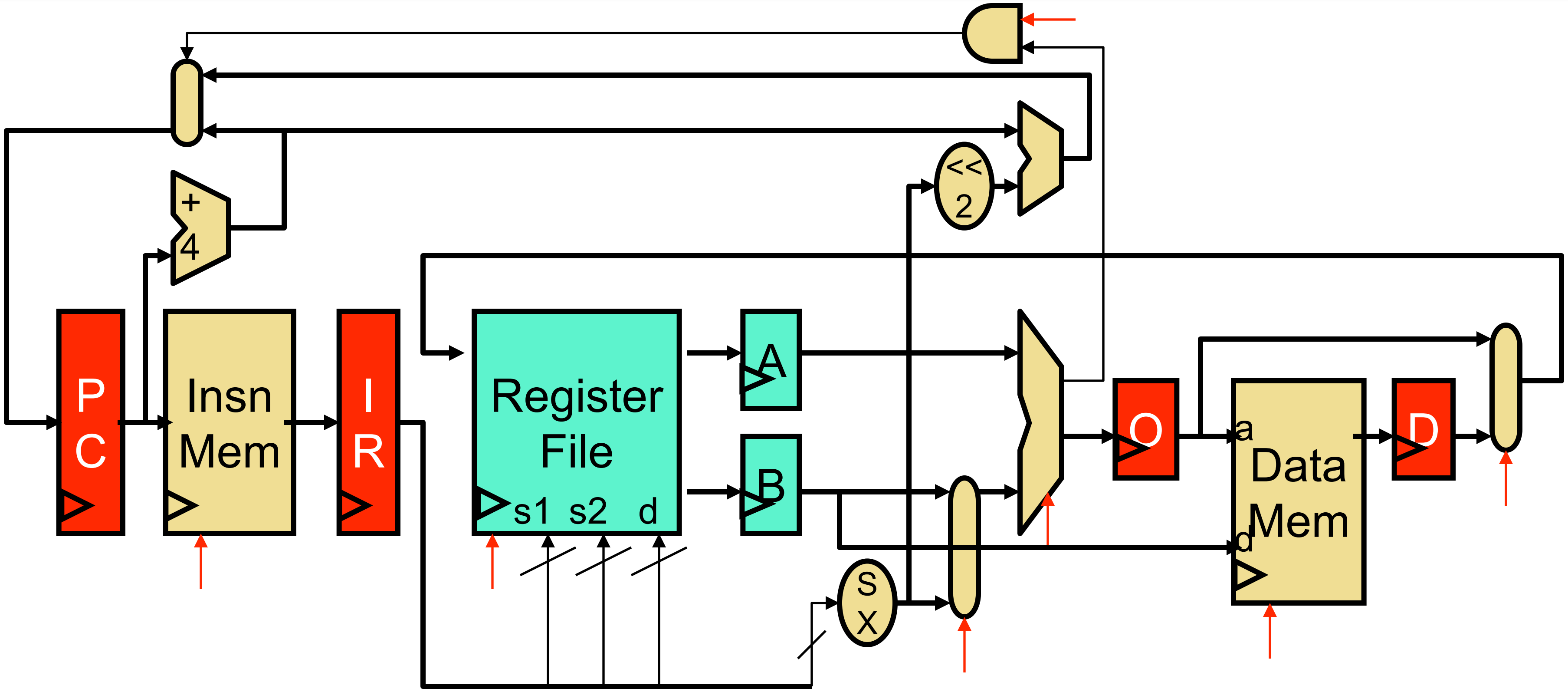
- Write DMEM state
  - Control signals enable memory write
  - Next state: Next Insn

Multi-Cycle Datapath Example: Add



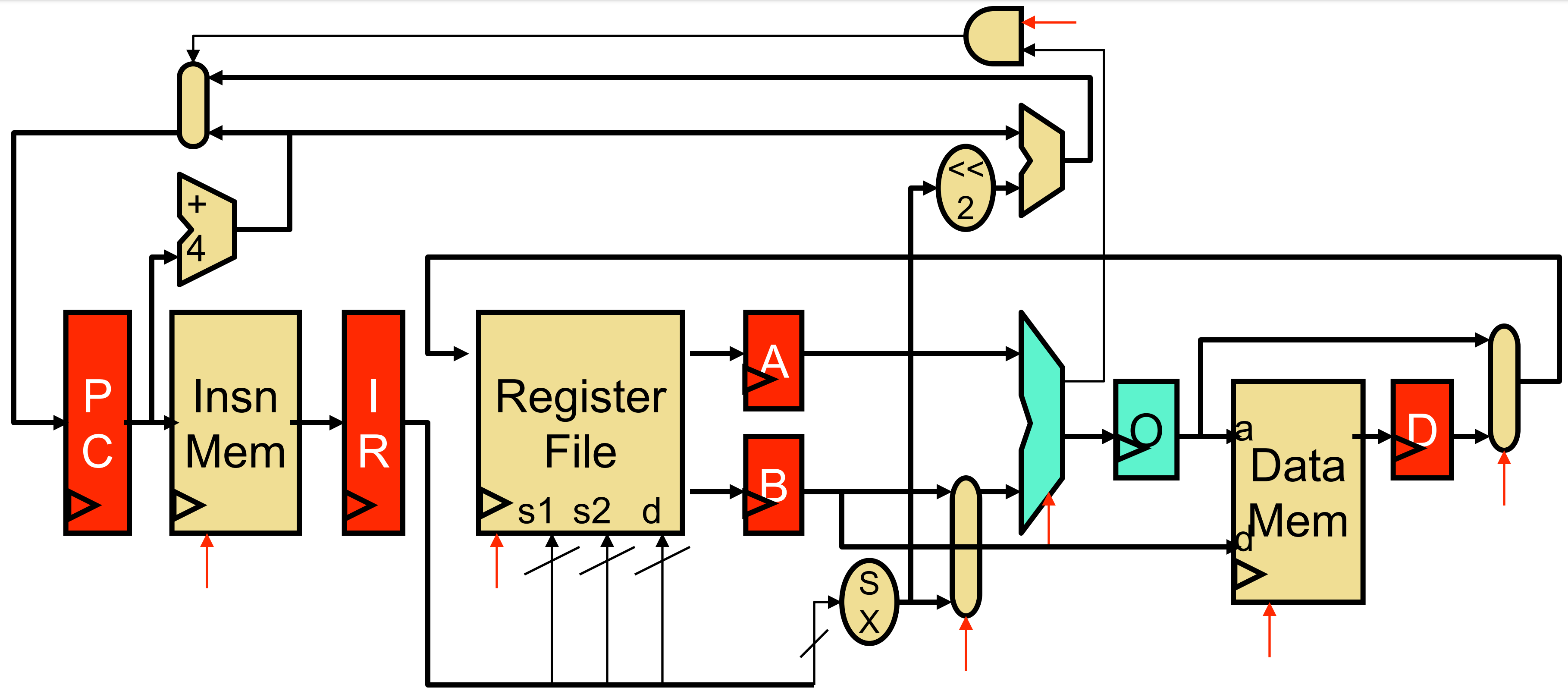
- Example: Add
  - Cycle 1: Read IMEM

Multi-Cycle Datapath Example: Add



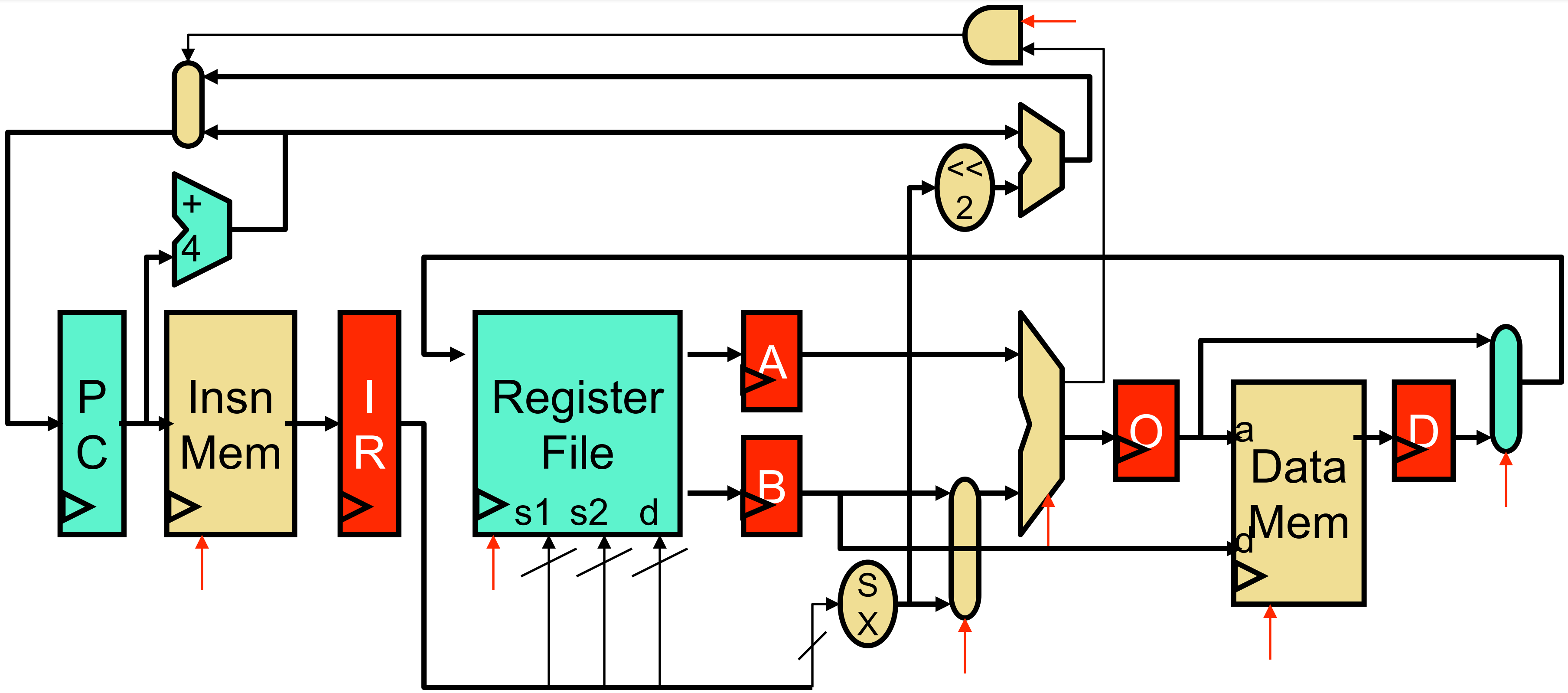
- Example: Add
  - Cycle 1: Read IMEM
  - Cycle 2: Decode + Read RF

Multi-Cycle Datapath Example: Add



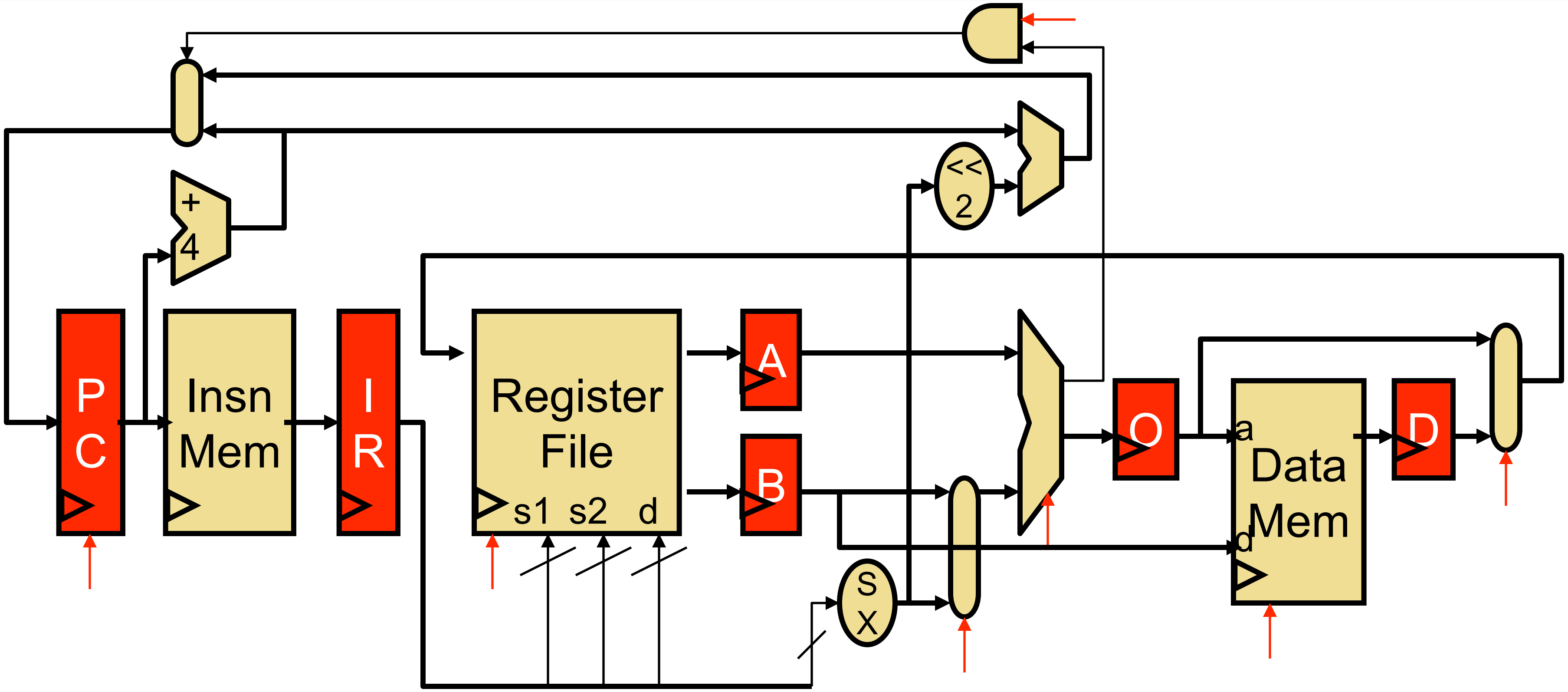
- Example: Add
  - Cycle 1: Read IMEM
  - Cycle 2: Decode + Read RF
  - Cycle 3: ALU

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 Datapath CPI

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

Multi-Cycle Datapath 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 Datapath 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 + 0.15 * 4 +

Multi-Cycle Datapath CPI

- CPI depends on instructions
  - Branches / Jumps: 3 cycles
  - ALU: 4 cycles
  - Stores: 4 cycles
  - Loads: 5 cycles

- Overall CPI is weighted average

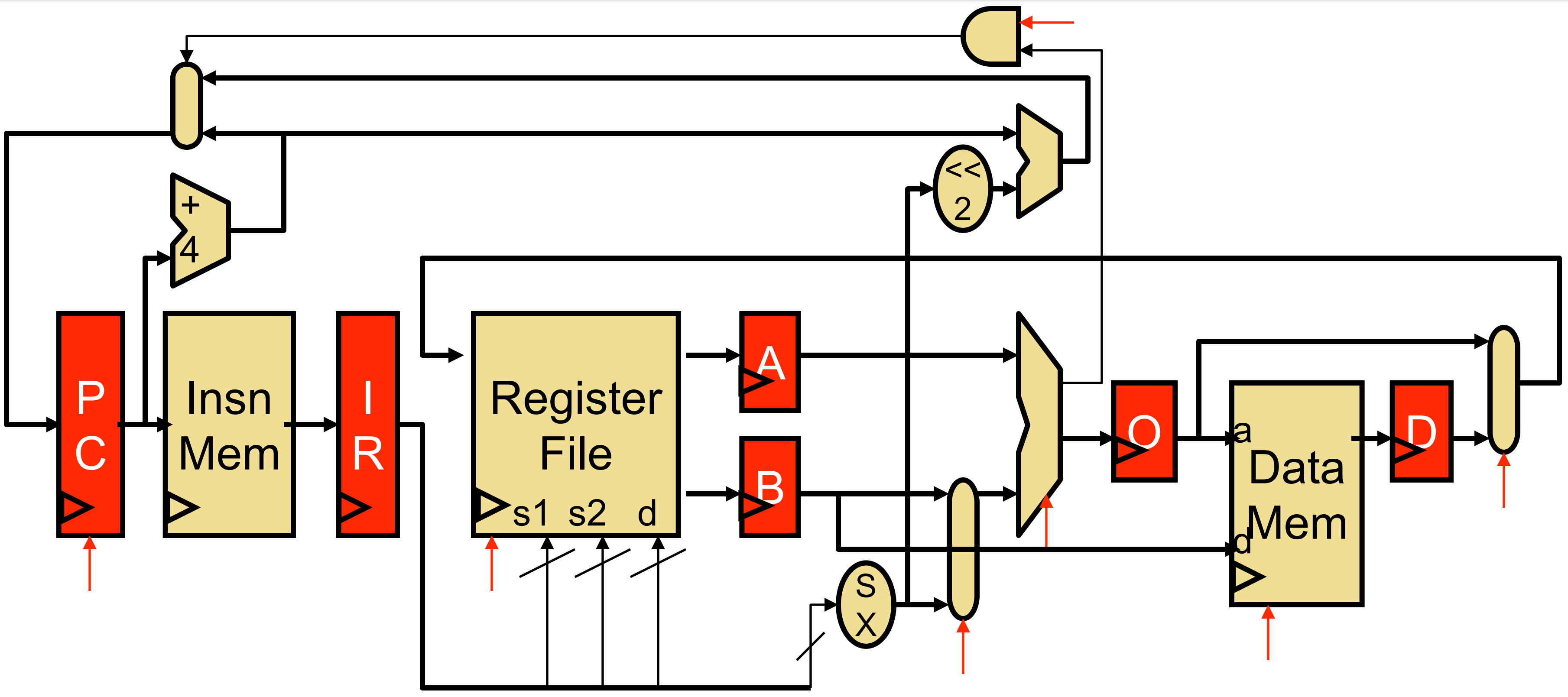
- Example:
  - 20% loads, 15% stores, 20% branches, 45% ALU

CPI = 0.20 * 5 + 0.15 * 4 + 0.20 * 3 + 0.45 * 4 = **4.0**

Multi-Cycle Datapath Performance

- Assuming clk period is 50ns…
- Single-cycle dp:
  - Clock period = 50ns, CPI = 1
  - Performance = 50 ns/insn
- Multi-cycle dp:
  - Clock period = 50ns/5 = 10ns
  - CPI = (0.2*3+0.2*5+0.6*4) = 4
  - Performance = 40 ns/insn
- But wait…

Multi-Cycle Datapath Performance



- We did not just cut up existing logic into 5 pieces
- We also added logic (registers)
- We also have to accommodate for the slowest stage
- → Clock period is not 1/5 of single cycle dp, but longer

Multi-Cycle Datapath Performance

- Assuming clk period is 50ns…

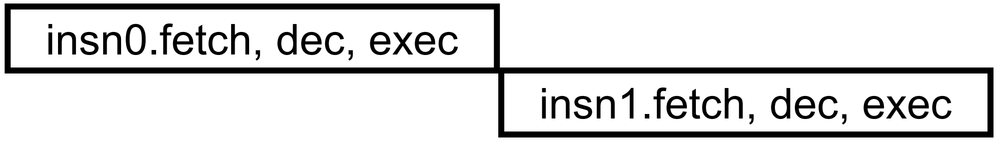
- Single-cycle
  - Clock period = 50ns, CPI = 1
  - Performance = 50 ns/insn

- Multi-cycle
  - Clock period = **12ns** (new approximation)
  - CPI = (0.2*3+0.2*5+0.6*4) = 4
  - Performance = **48 ns/insn**

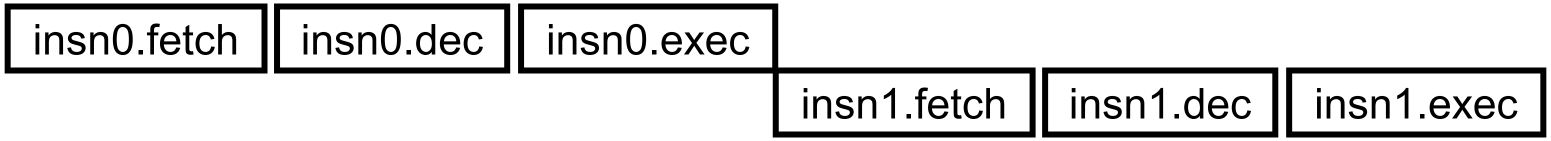
- Better, but not as exciting…
  - Can we do better?
  - Can we have low CPI AND high clock frequency?
  - *Also, don't forget about area and other measures*

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 (coming next)**