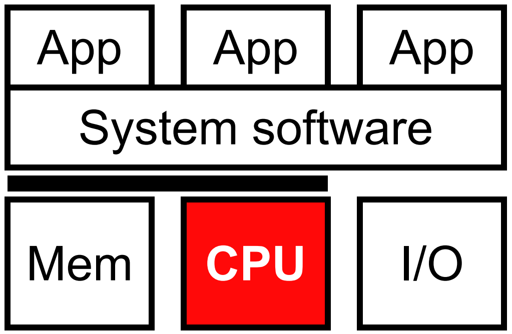
Computer Architecture

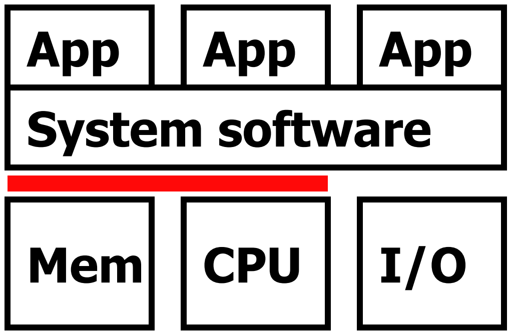
Lec 4a: Single-Cycle Datapath

This Lecture: Single-Cycle Datapath



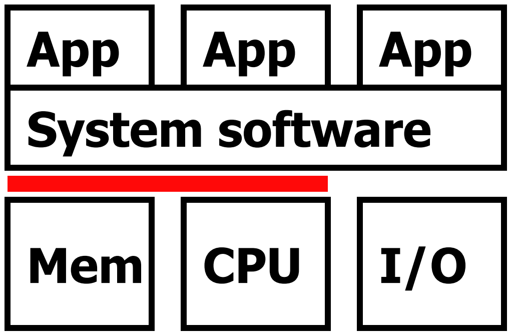
- Overview of ISAs
- Datapath storage elements
- MIPS Datapath
- MIPS Control

Review: ISA



- App/OS are software … execute on hardware
- HW/SW interface is **ISA (instruction set architecture)**
  - A **"contract"** between SW and HW
  - Encourages compatibility, allows SW/HW to evolve independently
  - **Functional definition** of HW storage locations & operations
    - Storage locations: registers, memory
    - Operations: add, multiply, branch, load, store, etc.
  - **Precise description** of how to invoke & access them
    - Instructions (bit-patterns hardware interprets as commands)

MIPS: A Real ISA



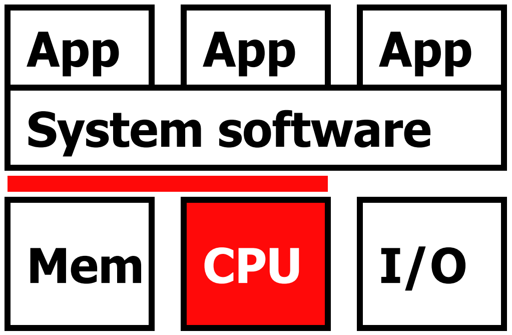
- **MIPS**: example of real ISA
  - 32/64-bit operations
  - 32-bit insns
  - 64 registers
    - 32 integer, 32 floating point
  - ~100 different insns
  - Full OS support

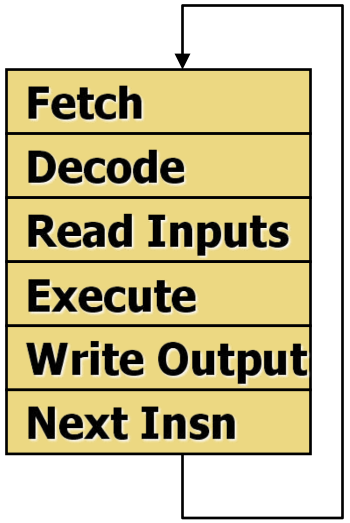
Example code is MIPS, but all ISAs are pretty similar

```
          .data
array:  .space 100
sum:      .word 0
          .text

array_sum:
          li $5, 0
          la $1, array
          la $2, sum
array_sum_loop:
          lw $3, 0($1)
          lw $4, 0($2)
          add $4, $3, $4
          sw $4, 0($2)
          addi $1, $1, 1
          addi $5, $5, 1
          li $6, 100
          blt $5, $6, array_sum_loop
```

Review: Insn Execution Model



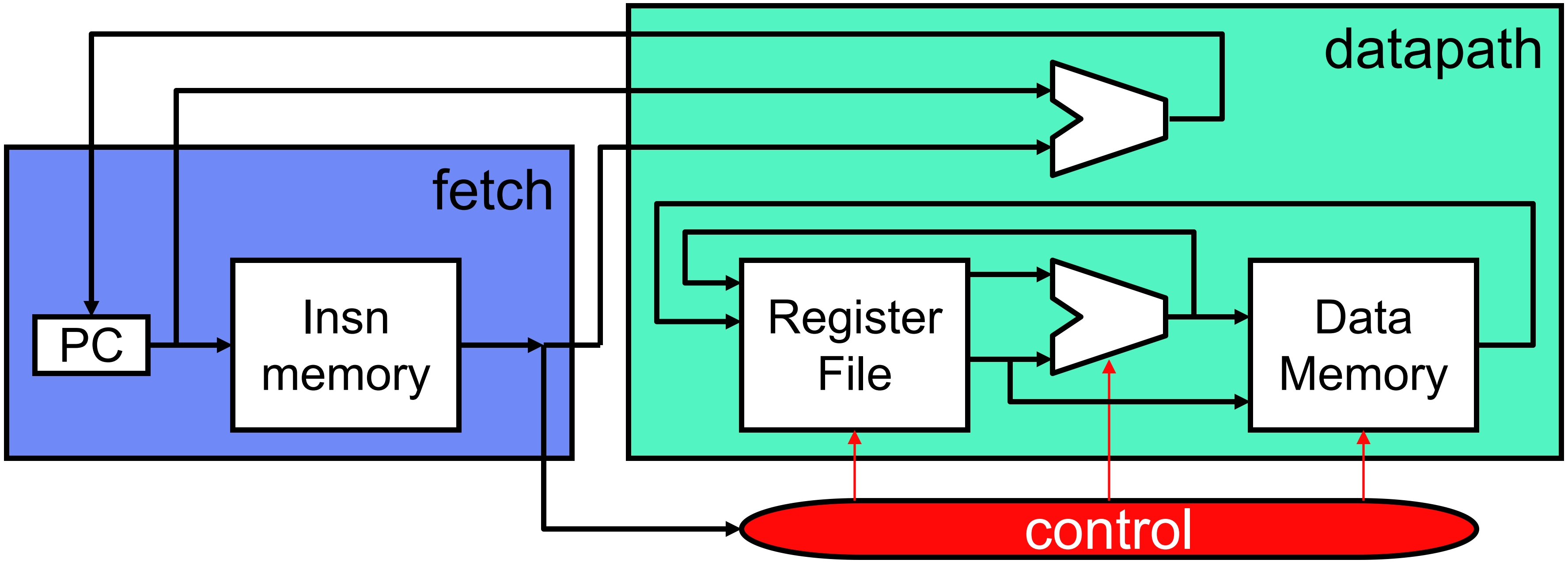


Instruction → Insn

- The computer is just finite state machine
  - **Registers** (few of them, but fast)
  - **Memory** (lots of memory, but slower)
  - **Program counter** (next insn to execute)
    - Sometimes called "instruction pointer"
- A computer executes **instructions**
  - **Fetches** next instruction from memory
  - **Decodes** it (figure out what it does)
  - **Reads** its **inputs** (registers & memory)
  - **Executes** it (adds, multiply, etc.)
  - **Write** its **outputs** (registers & memory)
  - **Next insn** (adjust the program counter)
- **Program is just "data in memory"**
  - Makes computers programmable ("universal")

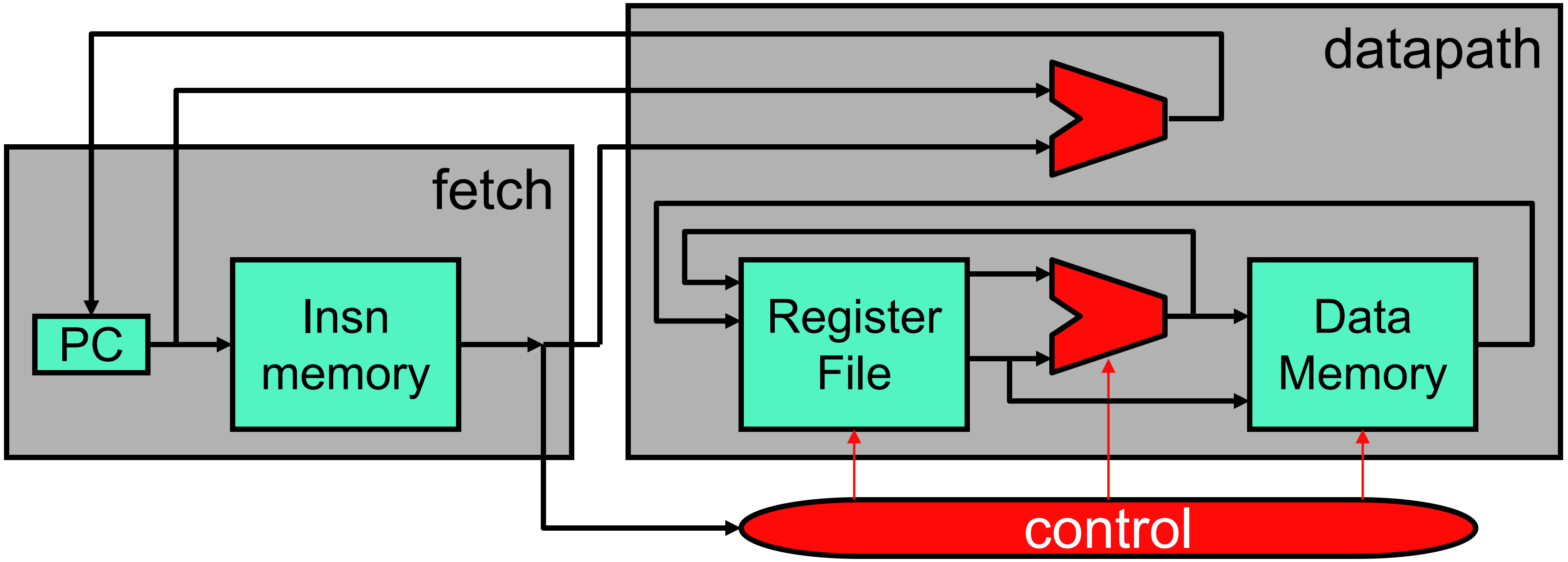
Implementing an ISA

Implementing an ISA



- **Datapath**: performs computation (registers, ALUs, etc.)
  - ISA specific: can implement every insn (single-cycle: in one pass!)
- **Control**: determines which computation is performed
  - Routes data through datapath (which regs, which ALU op)
- **Fetch**: get insn, translate opcode into control
- **Fetch** → **Decode** → **Execute** "cycle"

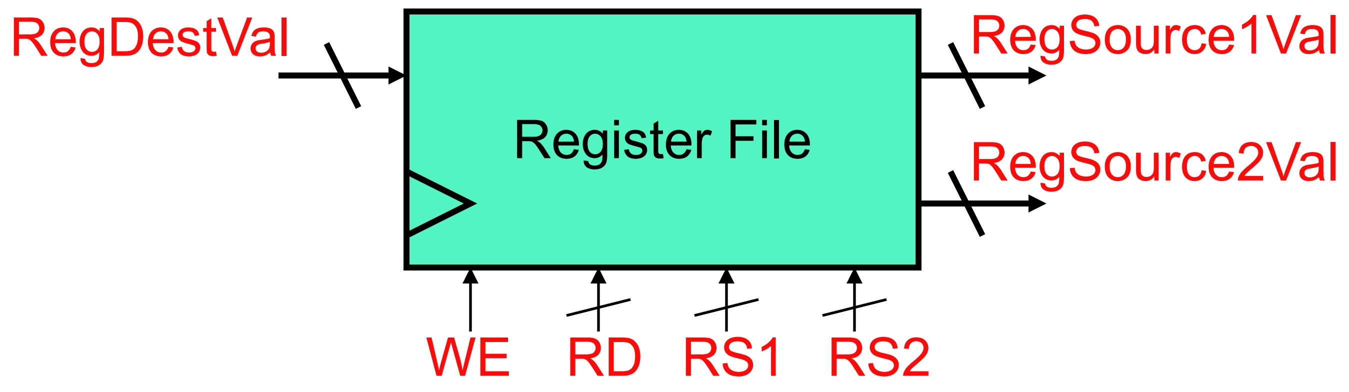
Two Types of Components



- **Purely combinational**: stateless computation
  - ALUs, muxes, control
  - Arbitrary Boolean functions
- **Combinational/sequential**: storage
  - PC, insn/data memories, register file
  - Internally contain some combinational components

Datapath Storage Elements

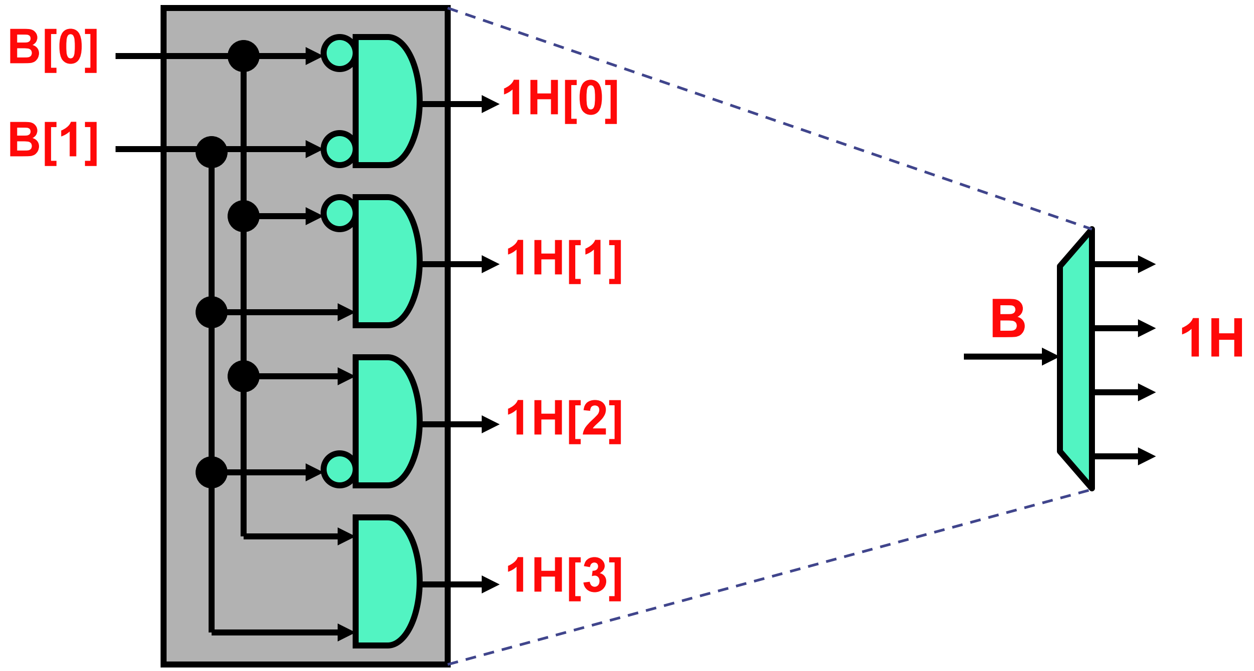
Register File



- **Register file**: M N-bit storage words
  - Multiplexed input/output: data buses write/read "random" word
- **"Port"**: set of buses for accessing a random word in array
  - Data bus (N-bits) + address bus ($\log_2$M-bits) + optional WE bit
  - P ports = P parallel and independent accesses
- MIPS integer register file
  - 32 32-bit words, two read ports + one write port

Decoder

- **Decoder**: converts binary integer to “1-hot” representation
  - Binary representation of 0…2^N–1: N bits
  - 1 hot representation of 0…2^N–1: 2^N bits
    - J represented as J^th^ bit 1, all other bits zero
  - Example below: 2-to-4 decoder



Decoder in Verilog (1 of 2)

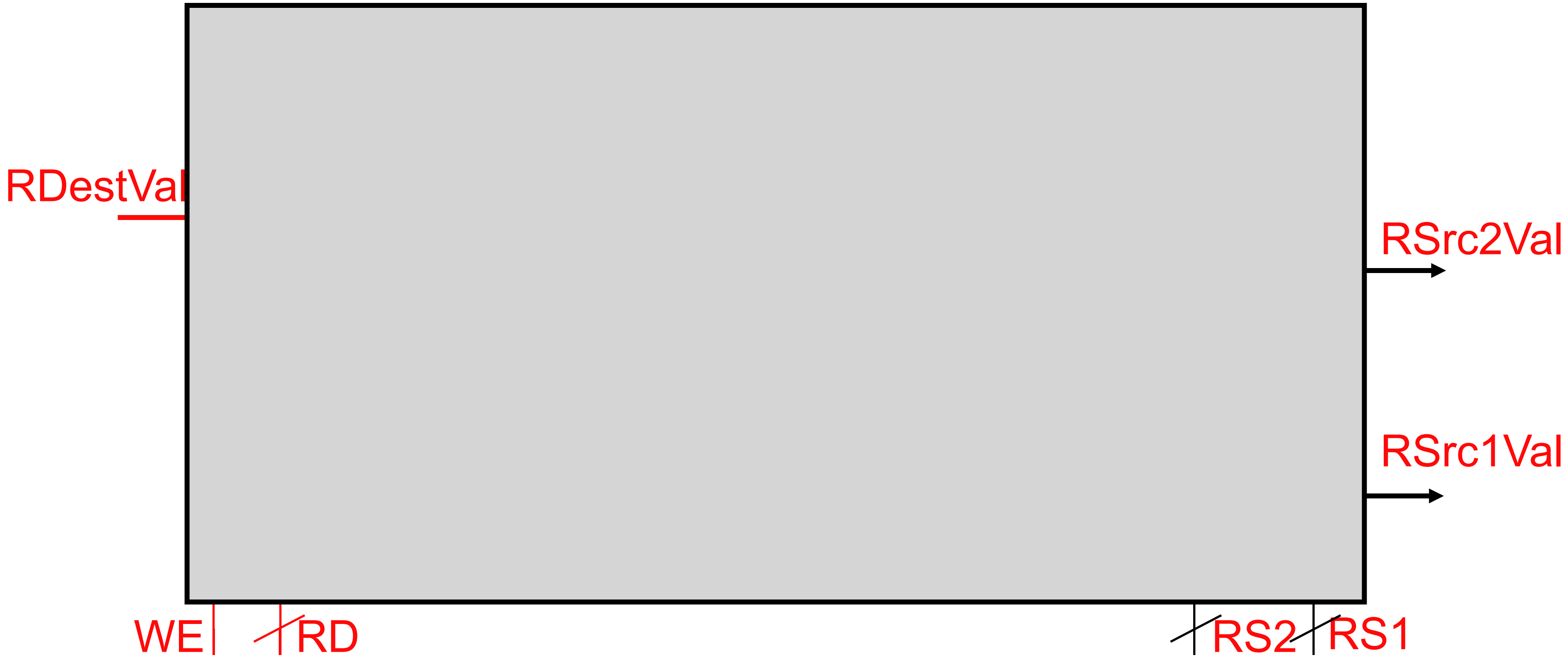
```
module decoder_2_to_4 (binary_in, onehot_out);
   input [1:0] binary_in;
   output [3:0] onehot_out;
   assign onehot_out[0] = (~binary_in[0] & ~binary_in[1]);
   assign onehot_out[1] = (~binary_in[0] & binary_in[1]);
   assign onehot_out[2] = (binary_in[0] & ~binary_in[1]);
   assign onehot_out[3] = (binary_in[0] & binary_in[1]);
endmodule
```

- Is there a simpler way?

Decoder in Verilog (2 of 2)

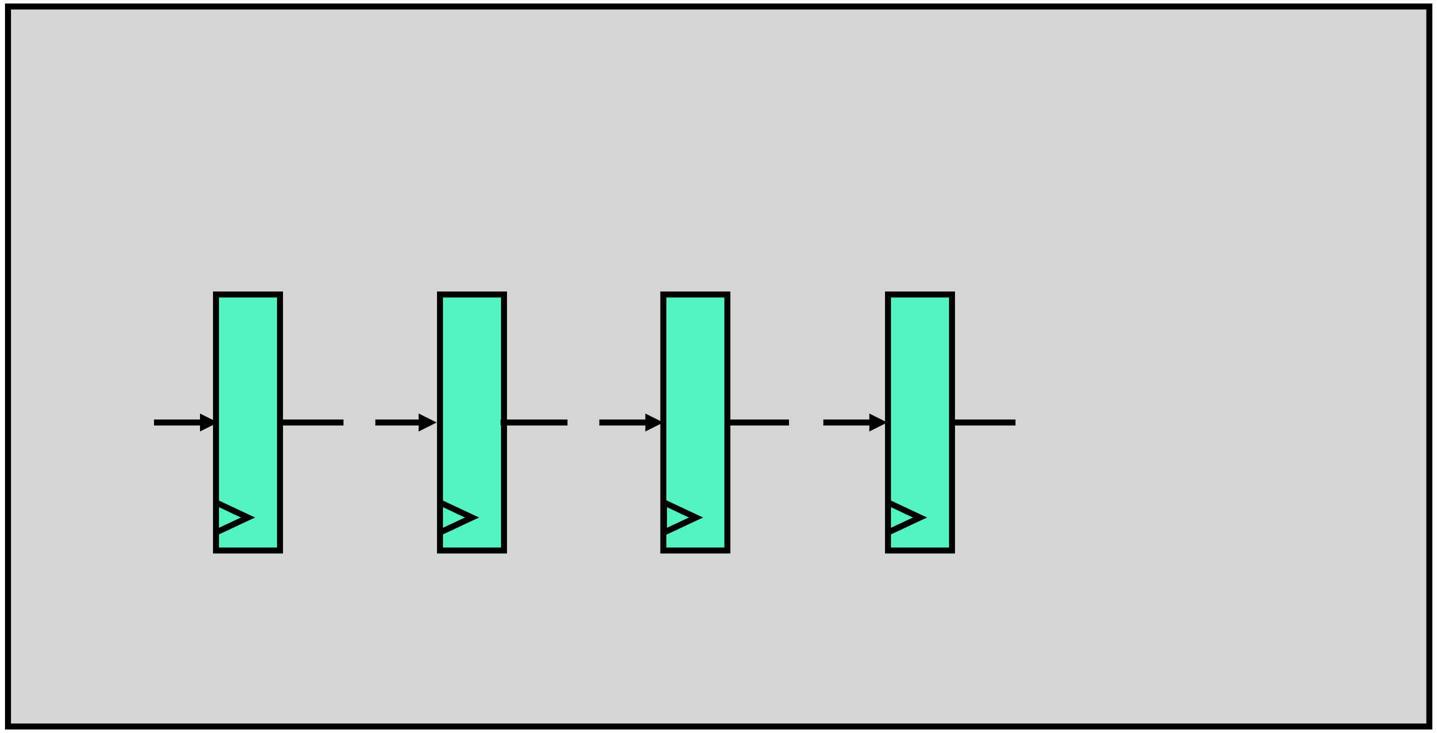
```
module decoder_2_to_4 (binary_in, onehot_out);
   input [1:0] binary_in;
   output [3:0] onehot_out;
   assign onehot_out[0] = (binary_in == 2’d0);
   assign onehot_out[1] = (binary_in == 2’d1);
   assign onehot_out[2] = (binary_in == 2’d2);
   assign onehot_out[3] = (binary_in == 2’d3);
endmodule
```

Register File Interface



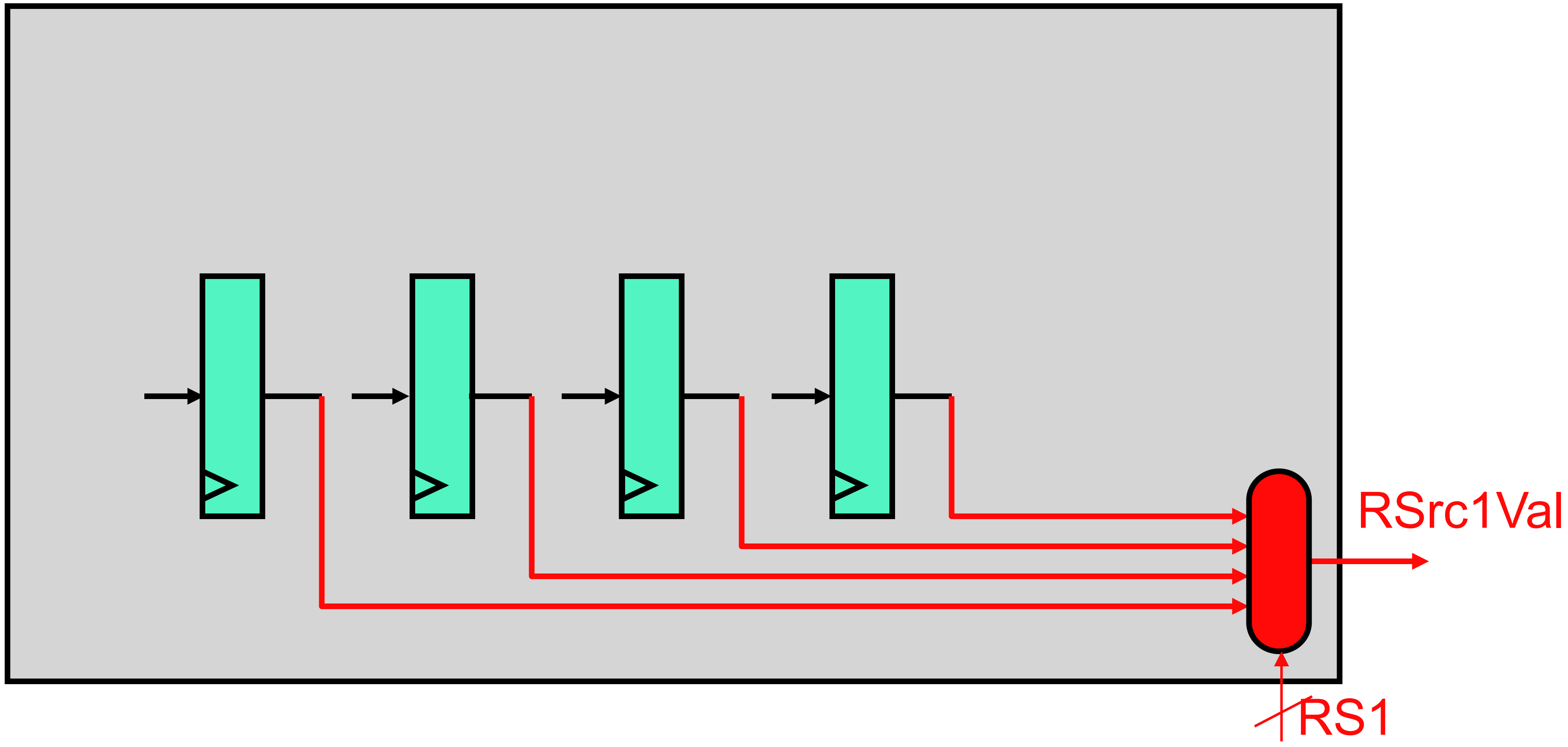
- Inputs:
  - RS1, RS2 (reg. sources to read), RD (reg. destination to write)
  - WE (write enable), RDestVal (value to write)
- Outputs: RSrc1Val, RSrc2Val (value of RS1 & RS2 registers)

Register File: Four Registers



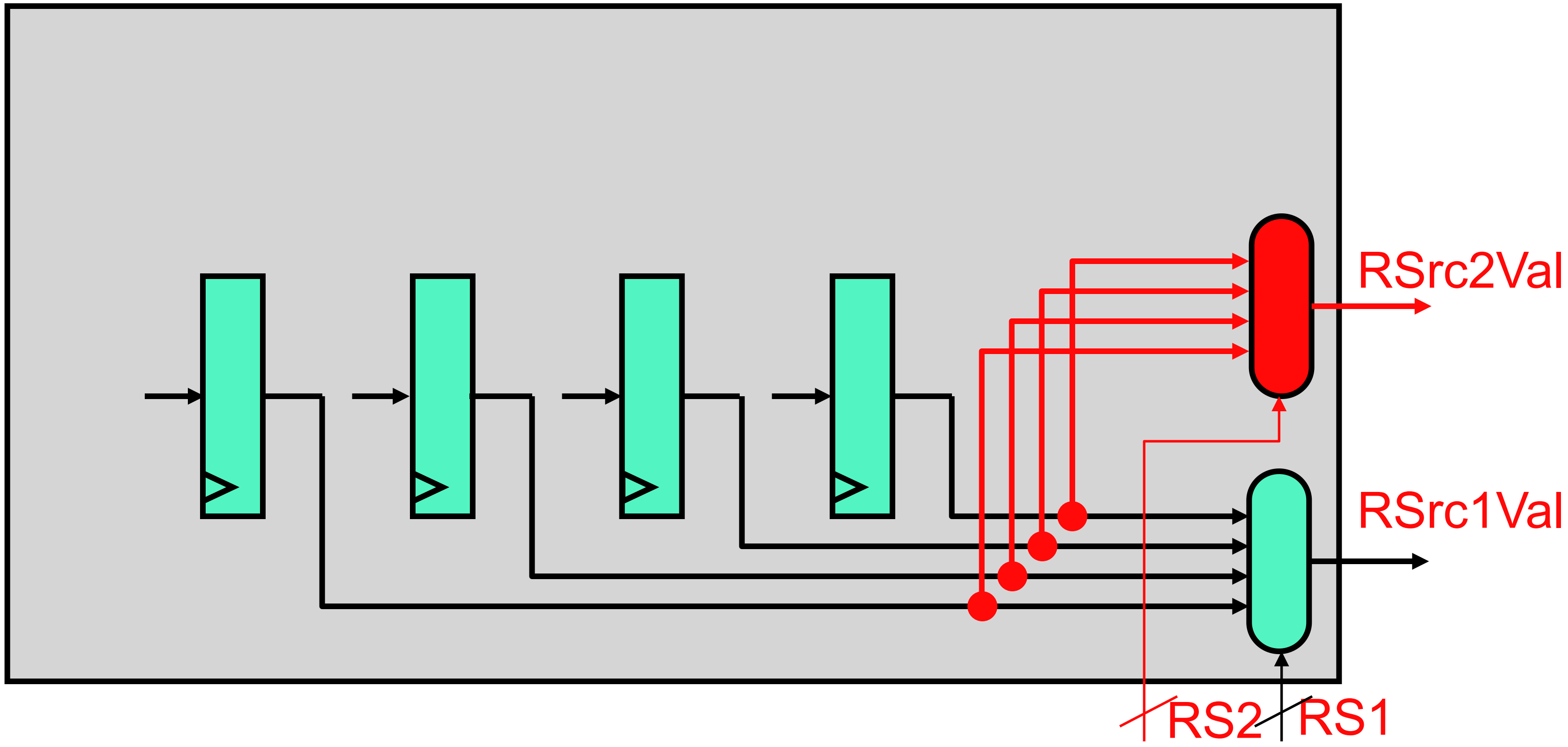
- Register file with four registers

Add a Read Port



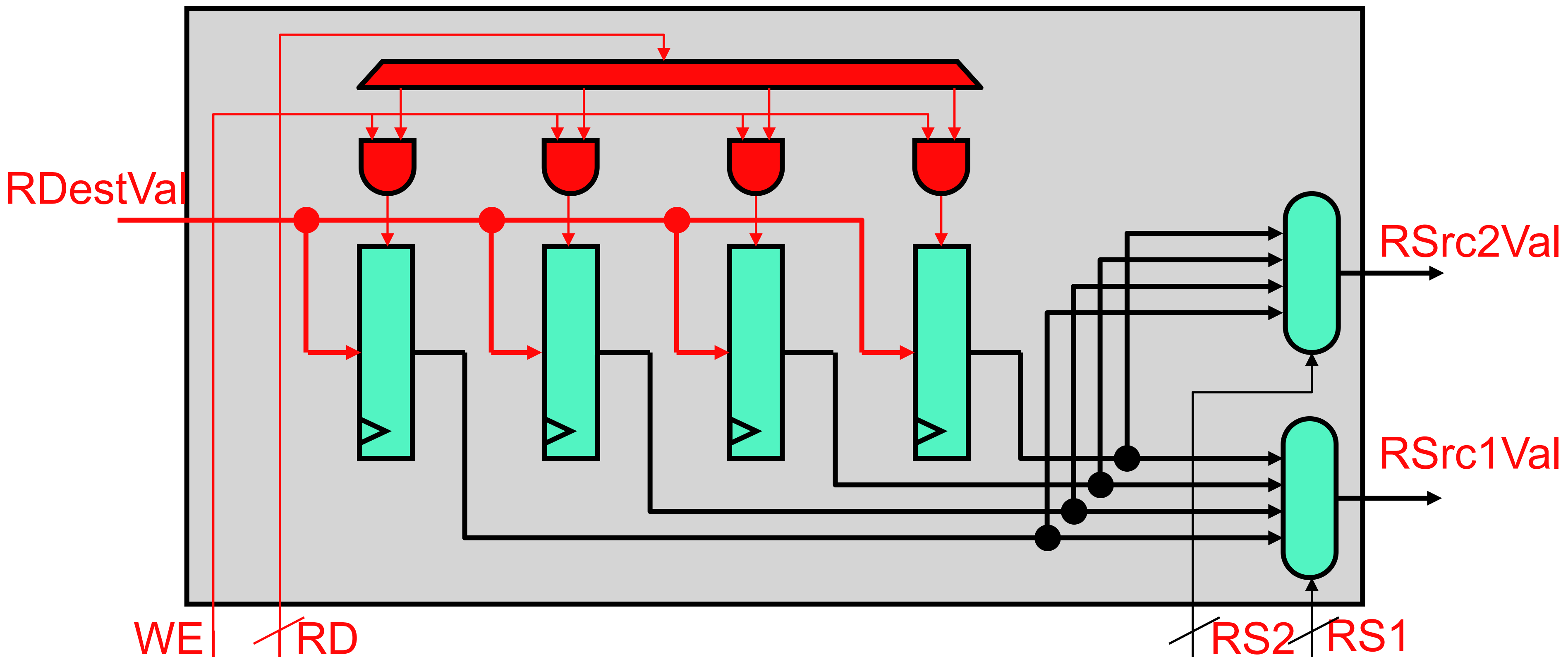
- Output of each register into 4to1 mux (RSrc1Val)
  - RS1 is select input of RSrc1Val mux

Add Another Read Port



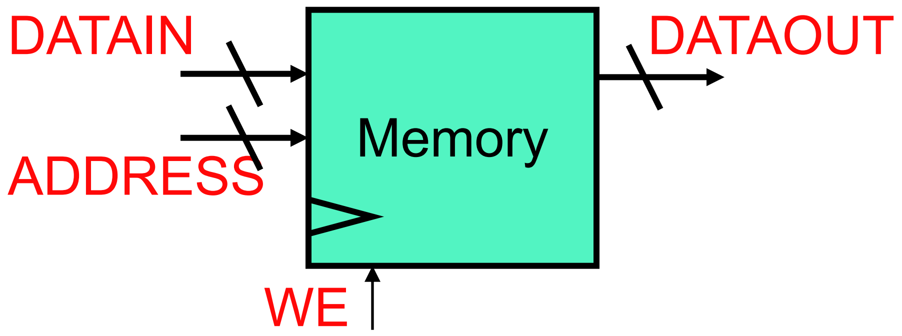
- Output of each register into another 4to1 mux (RSrc2Val)
    - RS2 is select input of RSrc2Val mux

Add a Write Port



- Input RegDestVal into each register
  - Enable only one register's WE: (Decoded RD) & (WE)
- What if we needed two write ports?

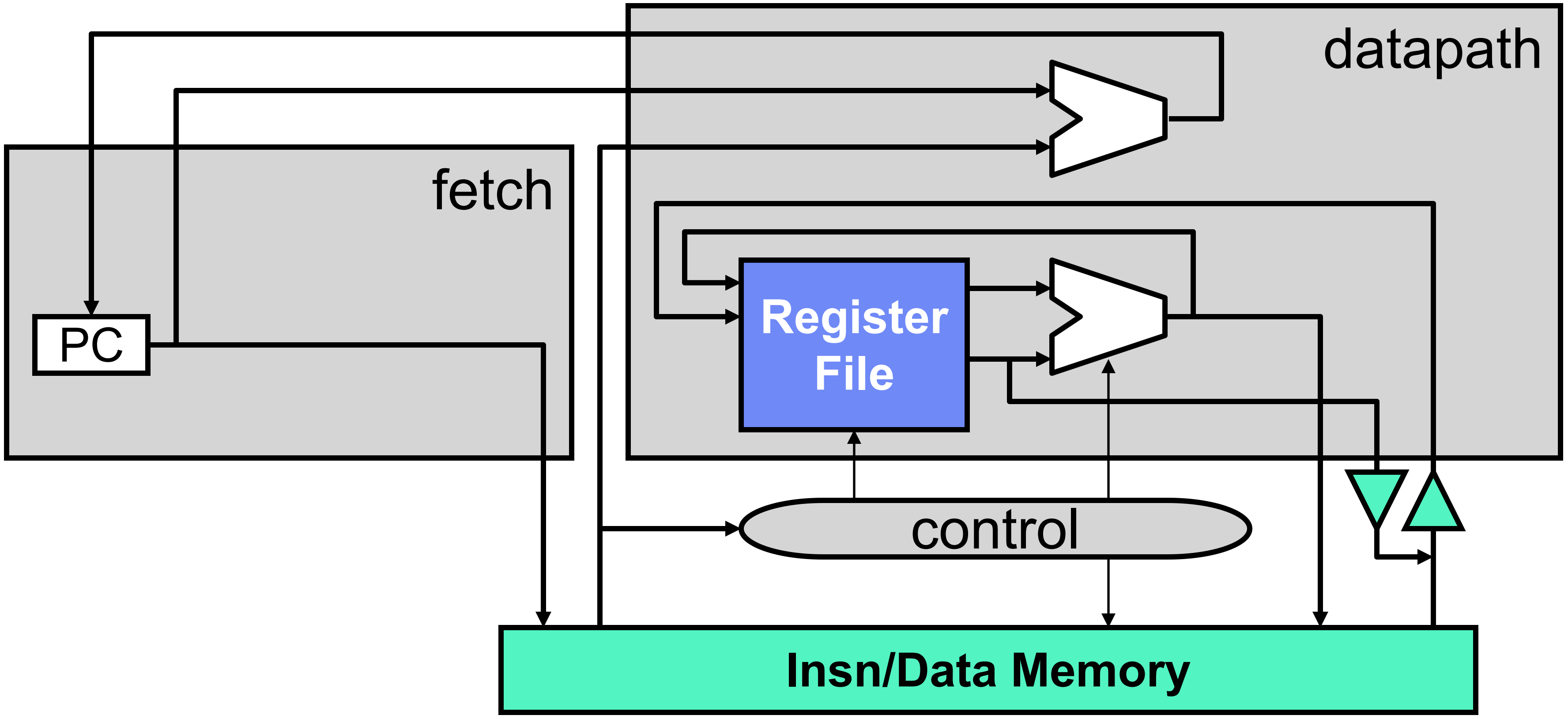
Another Useful Component: Memory



- Register file: M N-bit storage words
  - Few words (< 256), many ports, dedicated read and write ports
- **Memory**: M N-bit storage words, yet not a register file
  - Many words (> 1024), few ports (1, 2), shared read/write ports
- Leads to different implementation choices
  - Lots of circuit tricks and such
  - Larger memories typically only 6 transistors per bit

MIPS Datapath

Unified vs Split Memory Architecture



- **Unified architecture**: unified insn/data memory
- **“Harvard” architecture**: split insn/data memories

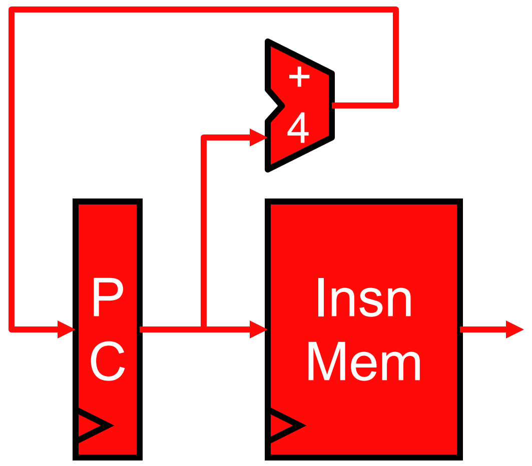
Datapath for MIPS ISA

- MIPS: 32-bit instructions, registers are $0, $2… $31
- Consider only the following instructions

```
add $1,$2,$3        $1 = $2 + $3        (add)
addi $1,$2,3        $1 = $2 + 3         (add immed)
lw $1,4($3)         $1 = Memory[4+$3]   (load)
sw $1,4($3)         Memory[4+$3] = $1   (store)
beq $1,$2,PC_relative_target  (branch equal)
j absolute_target             (unconditional jump)
```

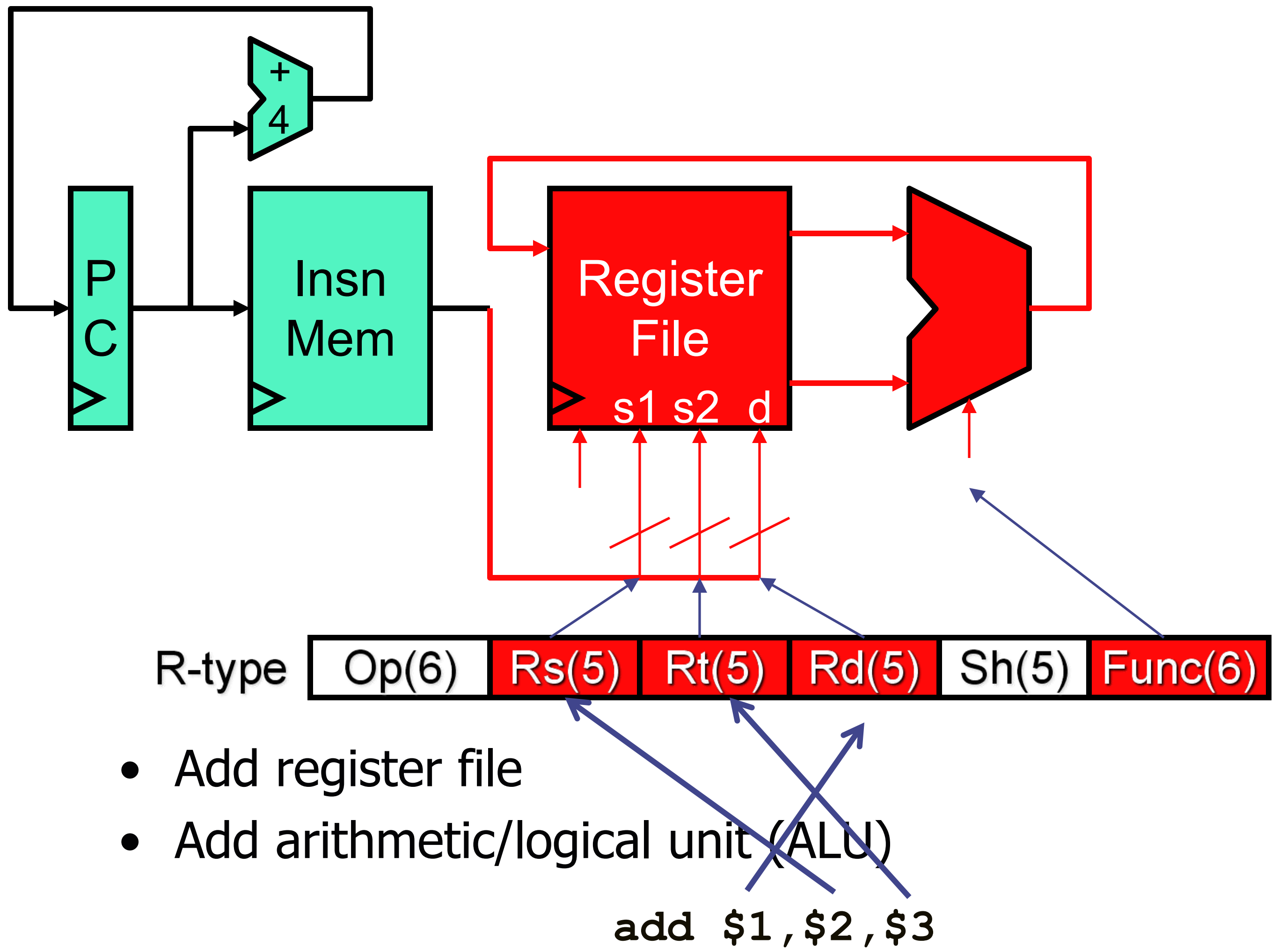
- Why only these?
  - Most other instructions are the same from datapath viewpoint
  - The ones that aren't are left for you to figure out ☺

Start With Fetch



- PC and instruction memory (split insn/data architecture, for now)
- A +4 incrementer computes default next instruction PC

First Instruction: **add**

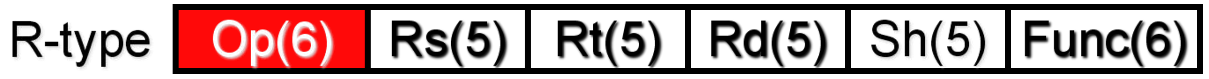


- Add register file
- Add arithmetic/logical unit (ALU)

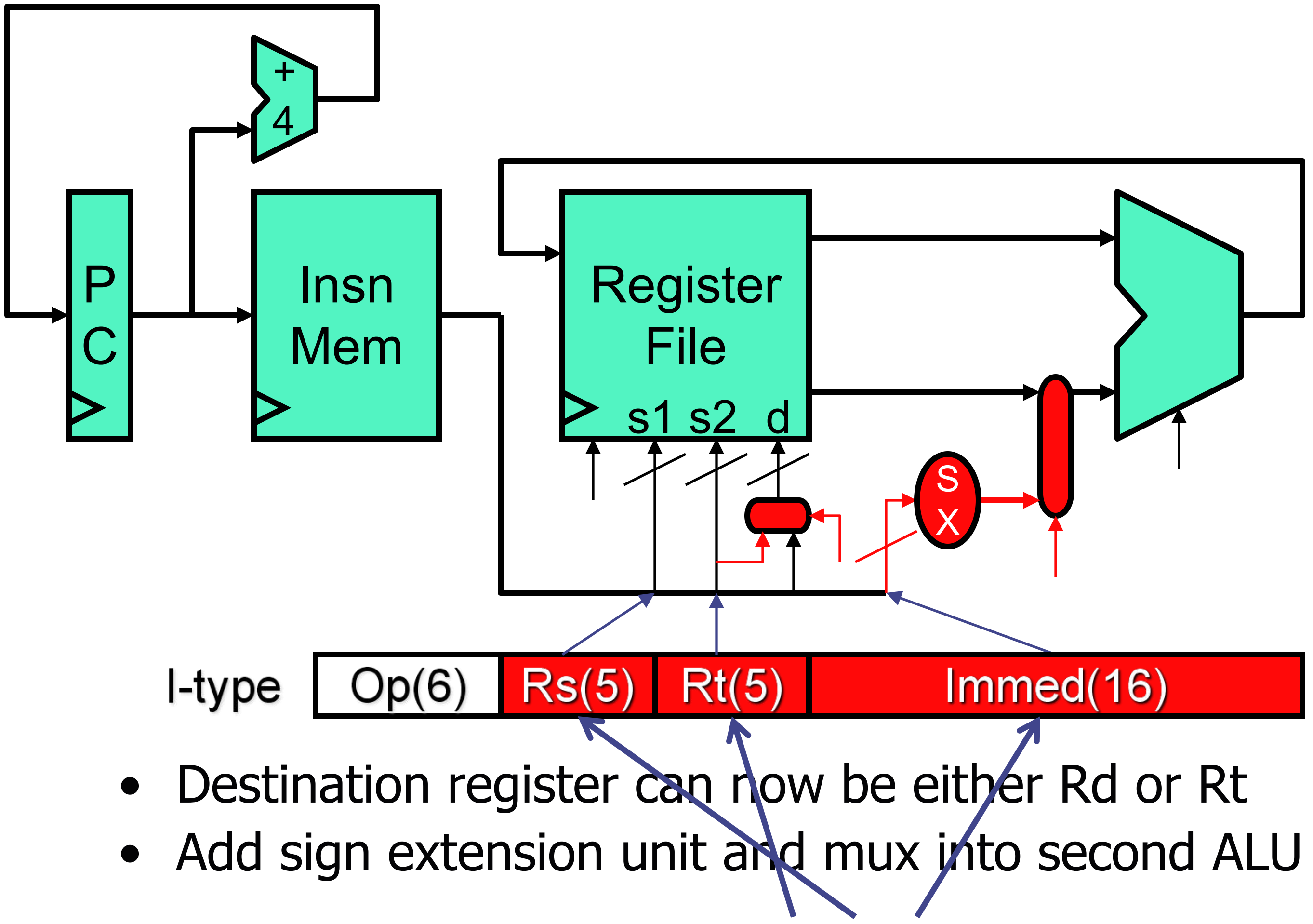
Wire Select in Verilog

- How to pull out individual fields of an insn? **Wire select**

```
wire [31:0] insn;
wire [5:0] op = insn[31:26];
wire [4:0] rs = insn[25:21];
wire [4:0] rt = insn[20:16];
wire [4:0] rd = insn[15:11];
wire [4:0] sh = insn[10:6];
wire [5:0] func = insn[5:0];
```



Second Instruction: **addi**



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

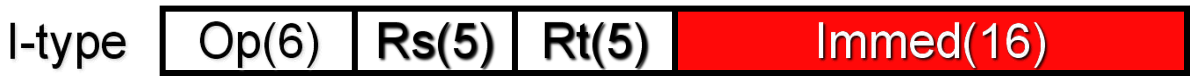
```
addi $1,$2,8
```

Verilog Wire Concatenation

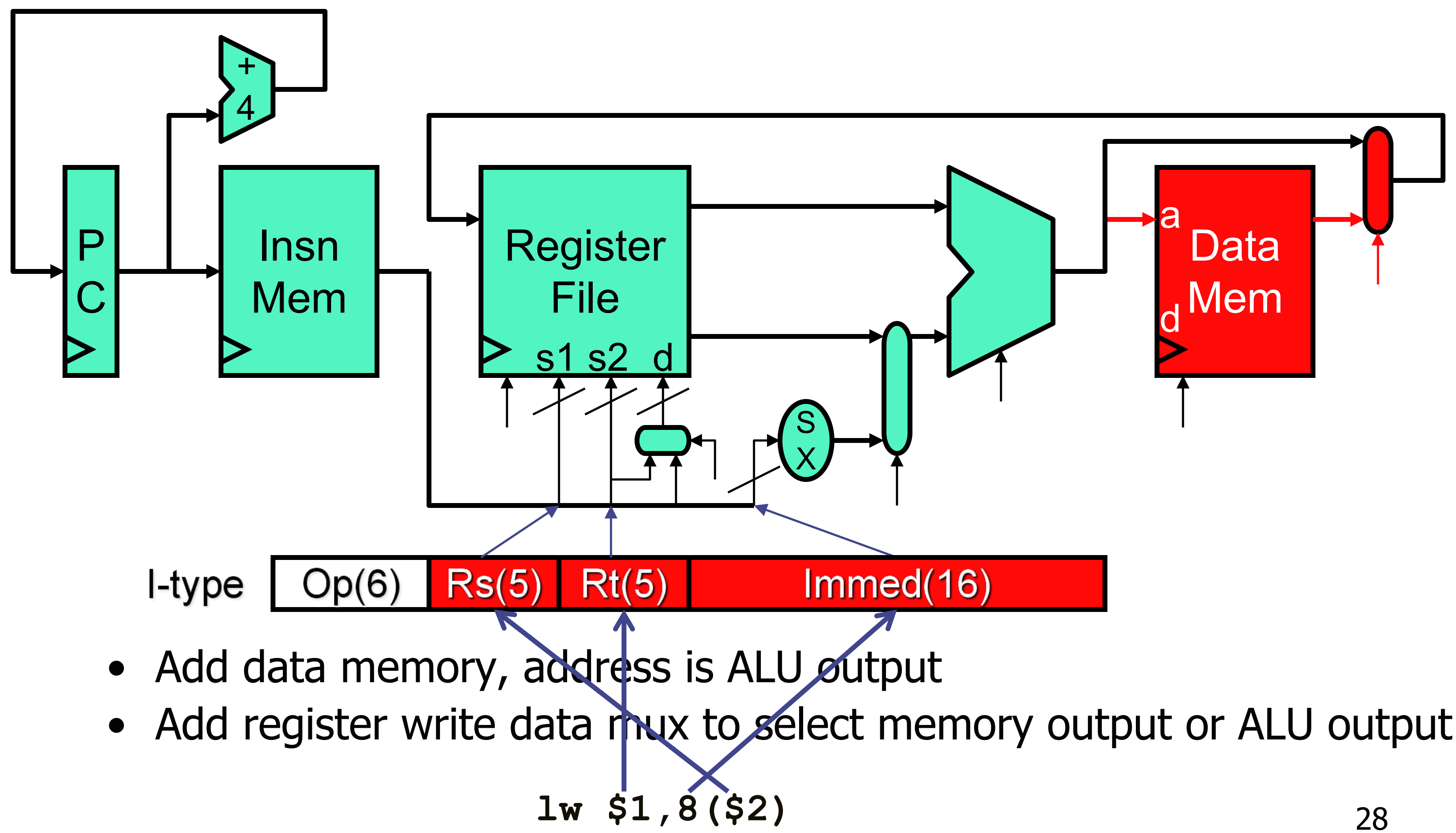
- Recall two Verilog constructs
  - **Wire concatenation**: `{bus0, bus1, … , busn}`
  - **Wire repeat:** `{repeat_x_times{w0}}`

- How do you specify sign extension? **Wire concatenation**

```
wire [31:0] insn;
wire [15:0] imm16 = insn[15:0];
wire [31:0] sximm16 = {{16{imm16[15]}}, imm16};
```

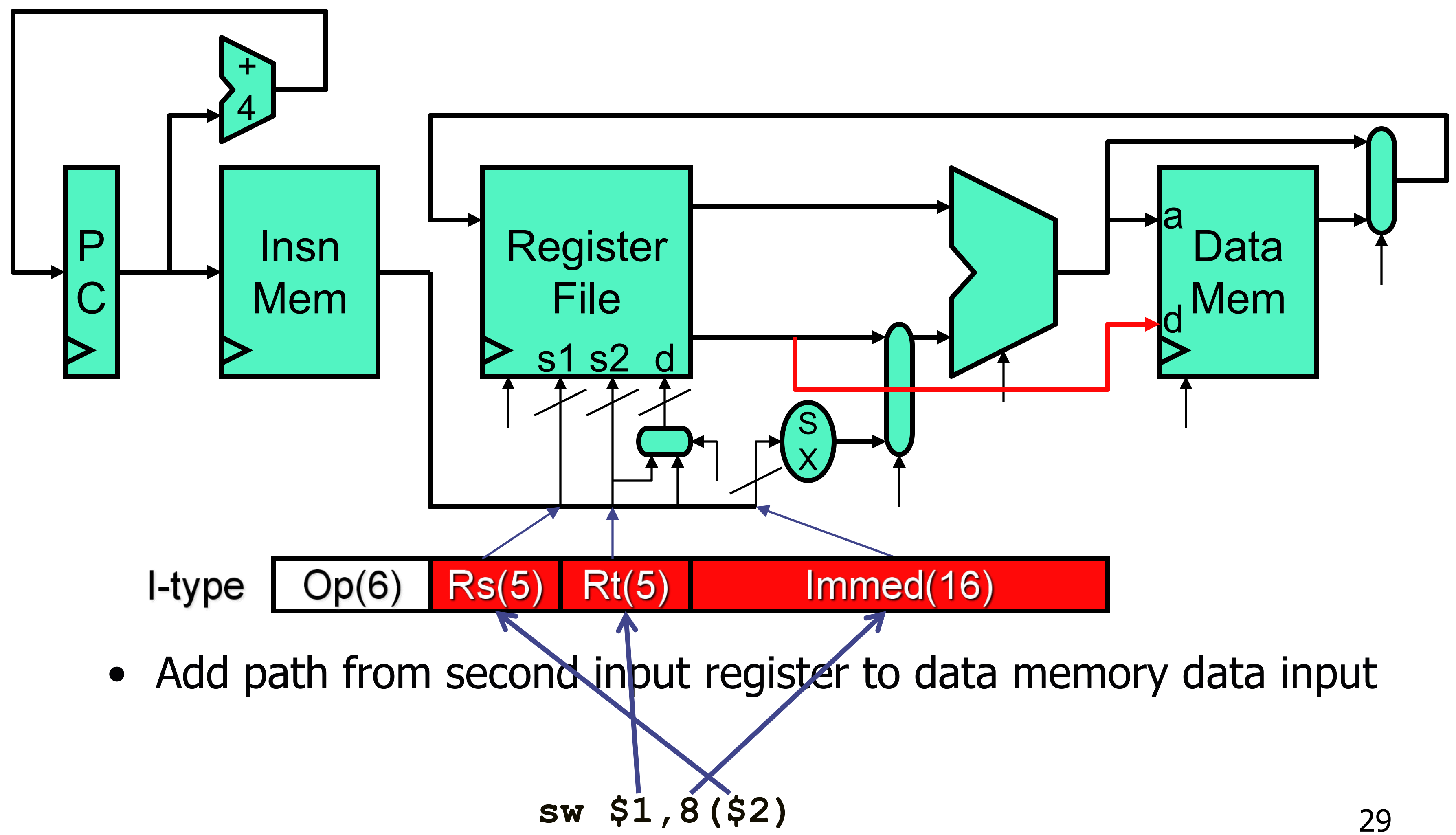


Third Instruction: **lw**



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

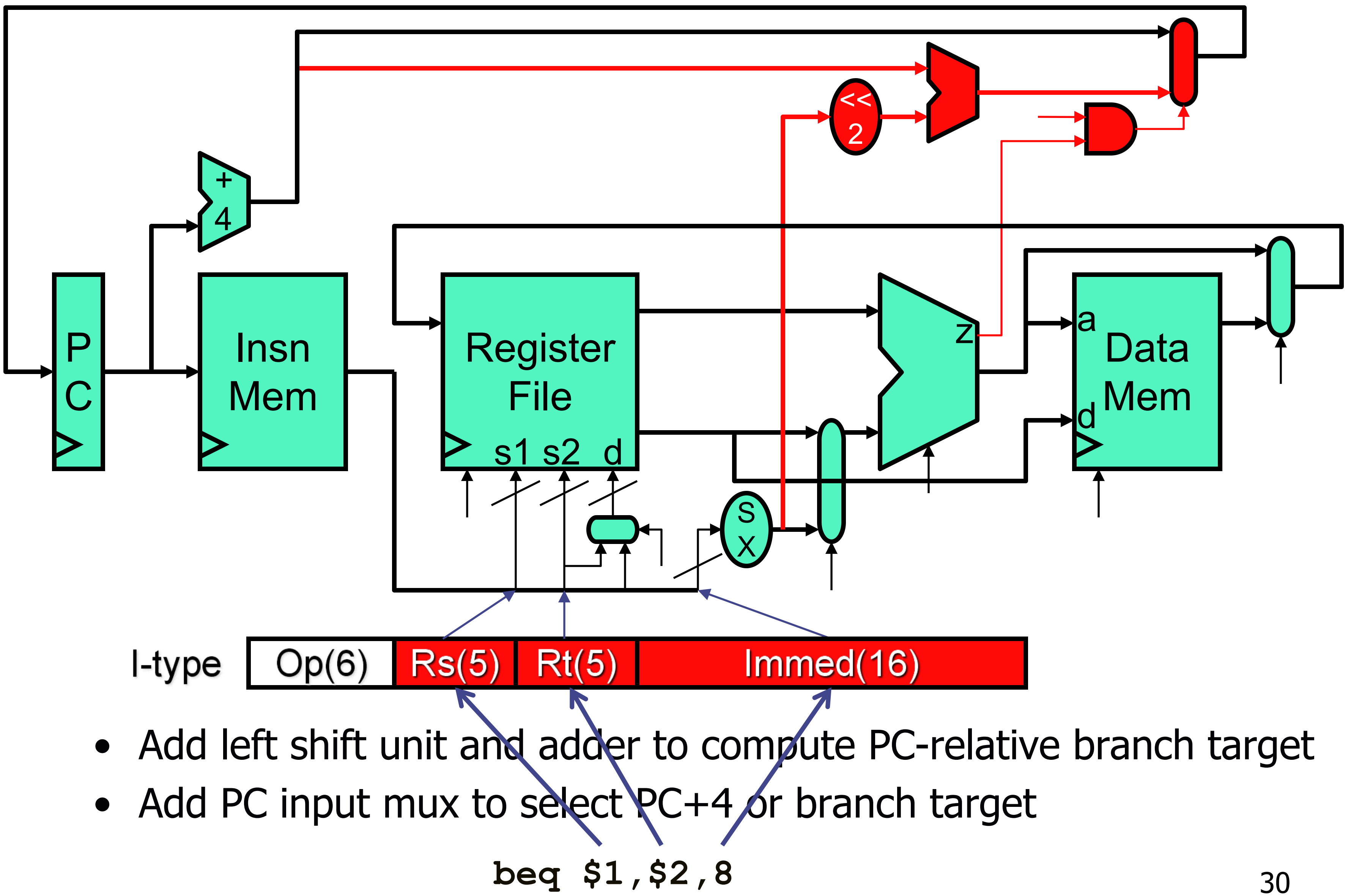
Fourth Instruction: **sw**



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

```
sw $1,8($2)
```

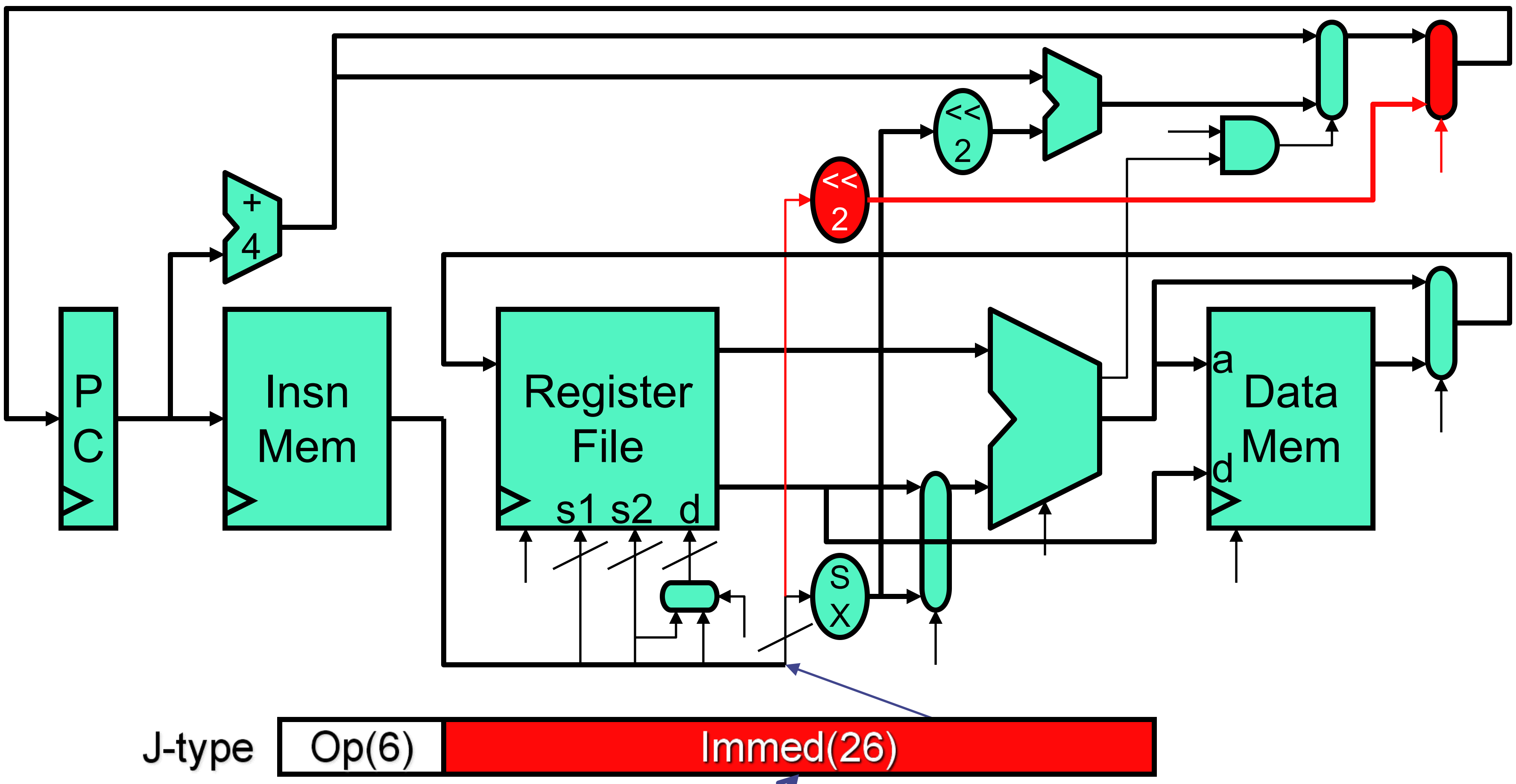
Fifth Instruction: **beq**



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

```
beq $1,$2,8
```

Sixth Instruction: j

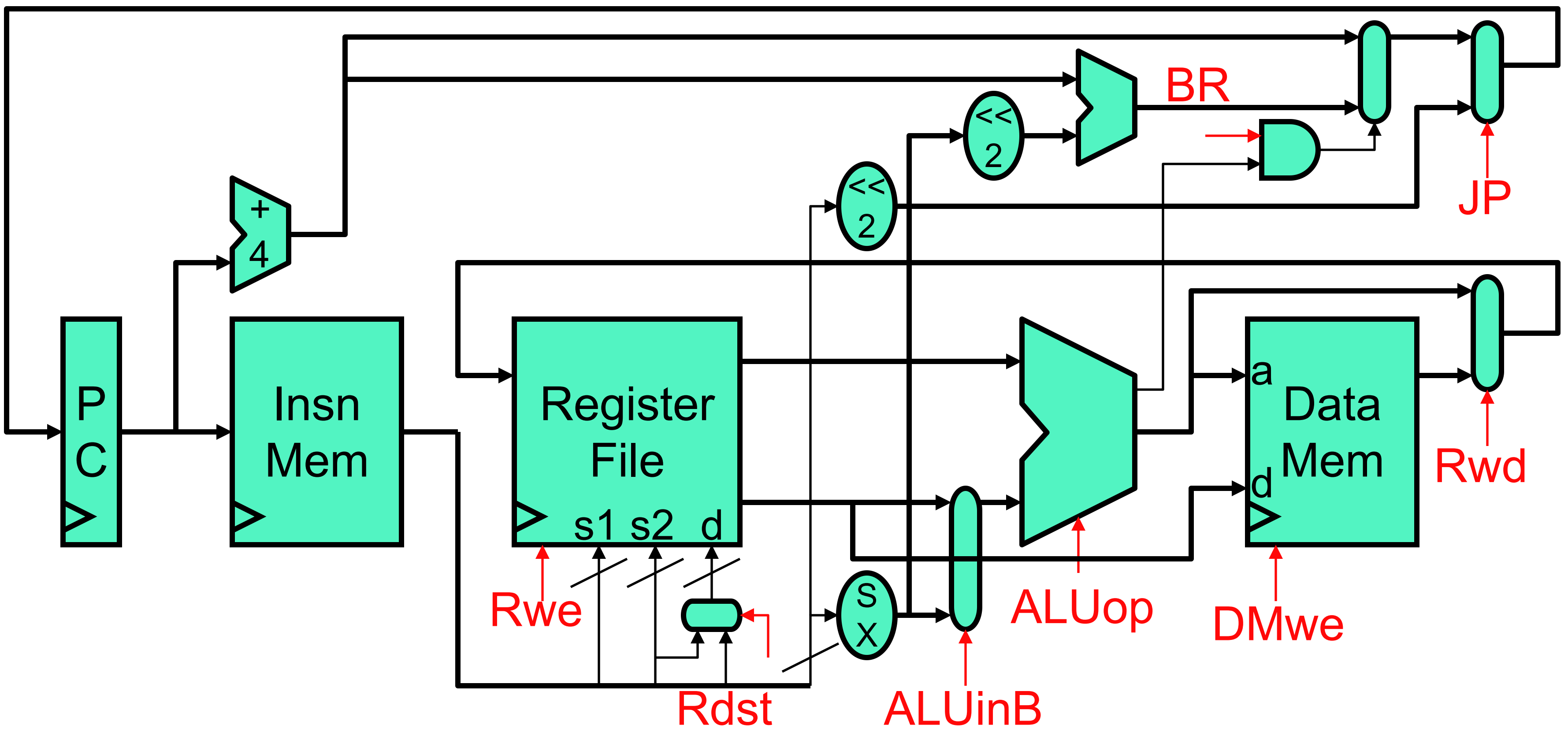


- Add shifter to compute left shift of 26-bit immediate
- Add additional PC input mux for jump target

```
j 100
```

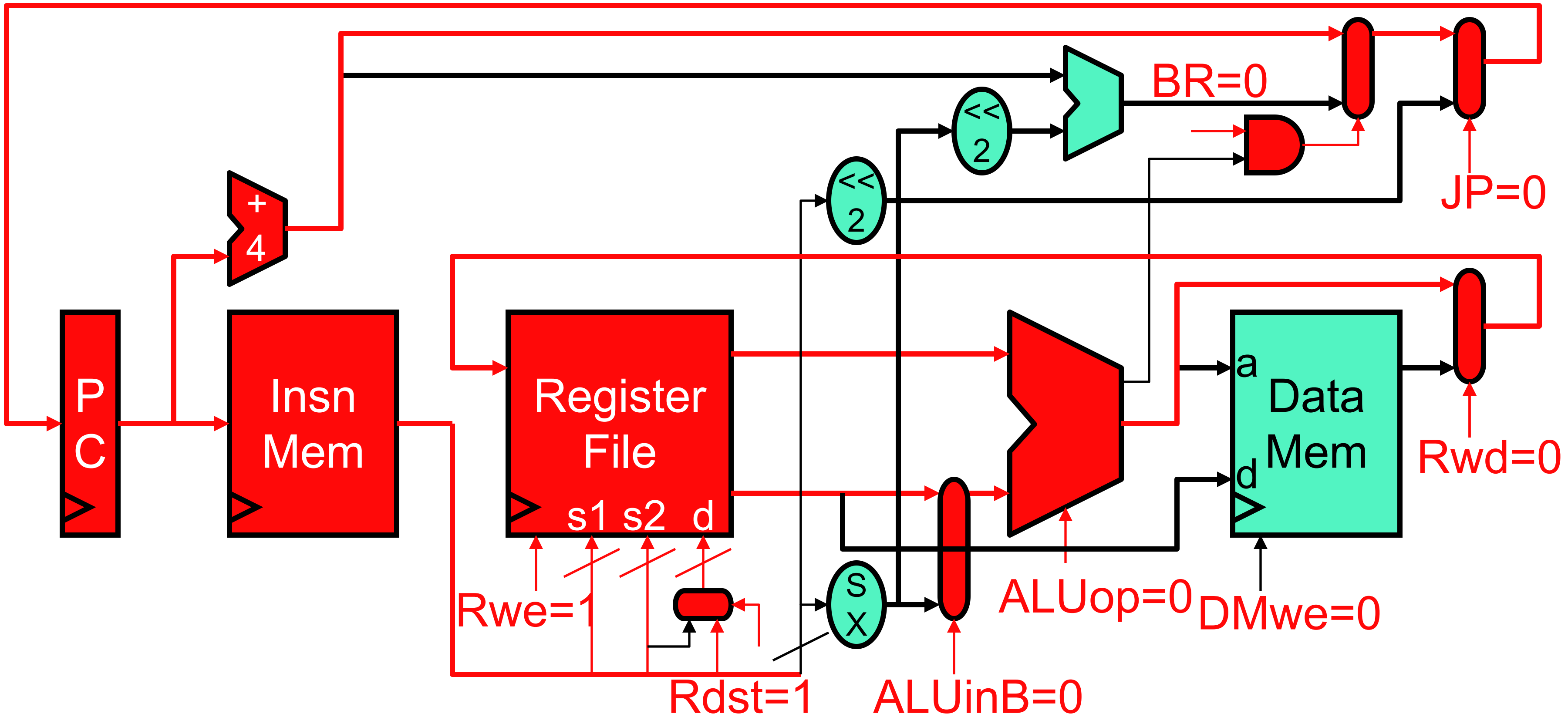
MIPS Control

What Is Control?



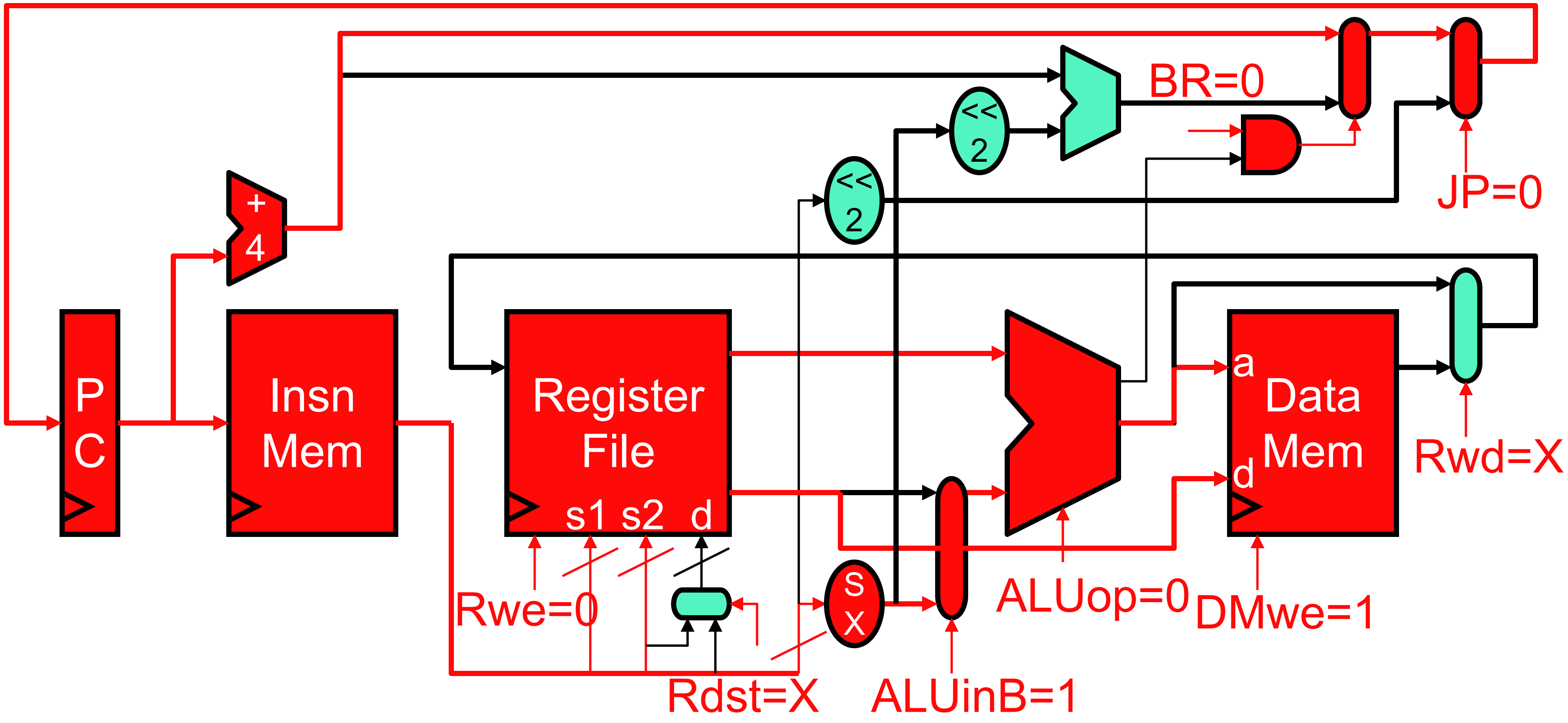
- 8 signals control flow of data through this datapath
  - MUX selectors, or register/memory write enable signals
  - A real datapath has 300-500 control signals

Example: Control for **add**



```
add $4, $3, $4
```

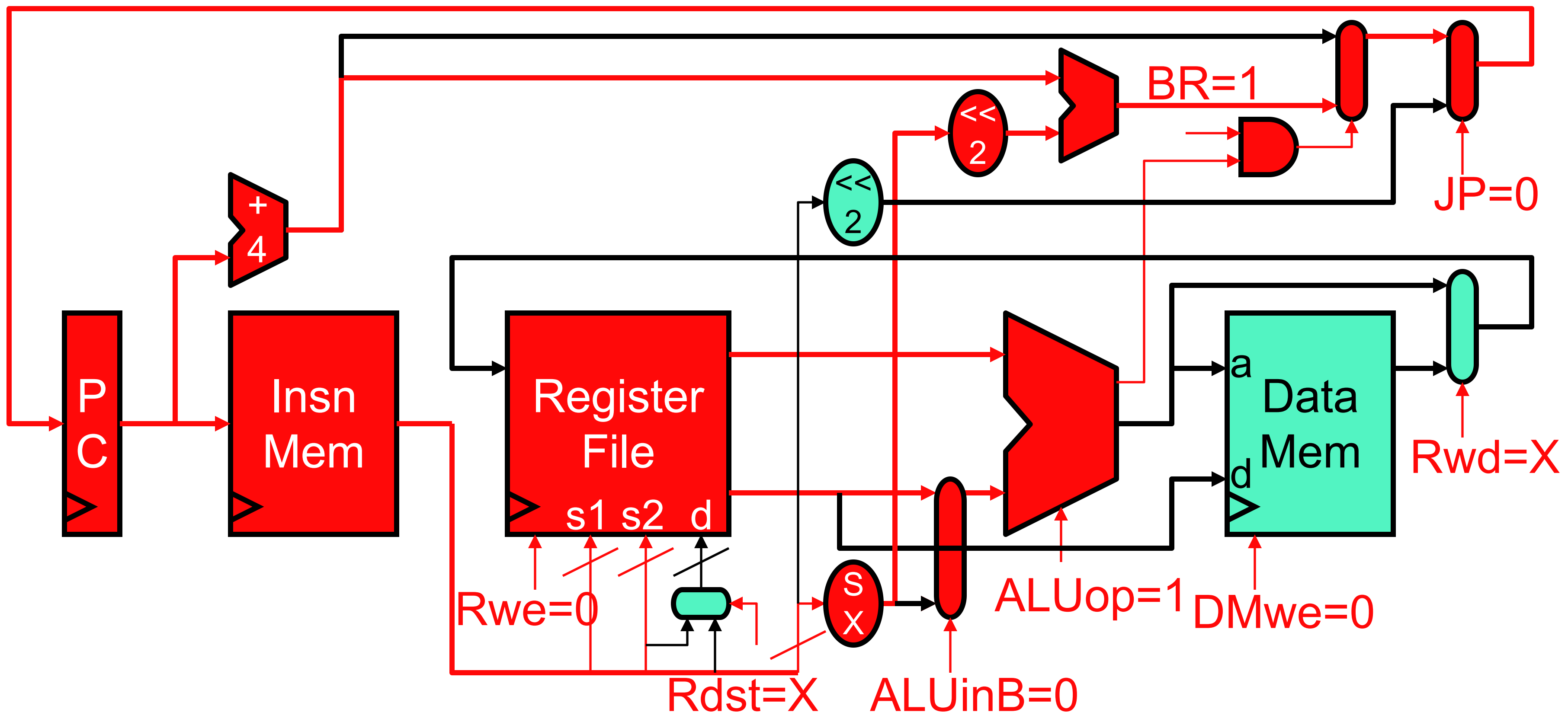
Example: Control for **sw**



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

`sw $4, 0($2)`

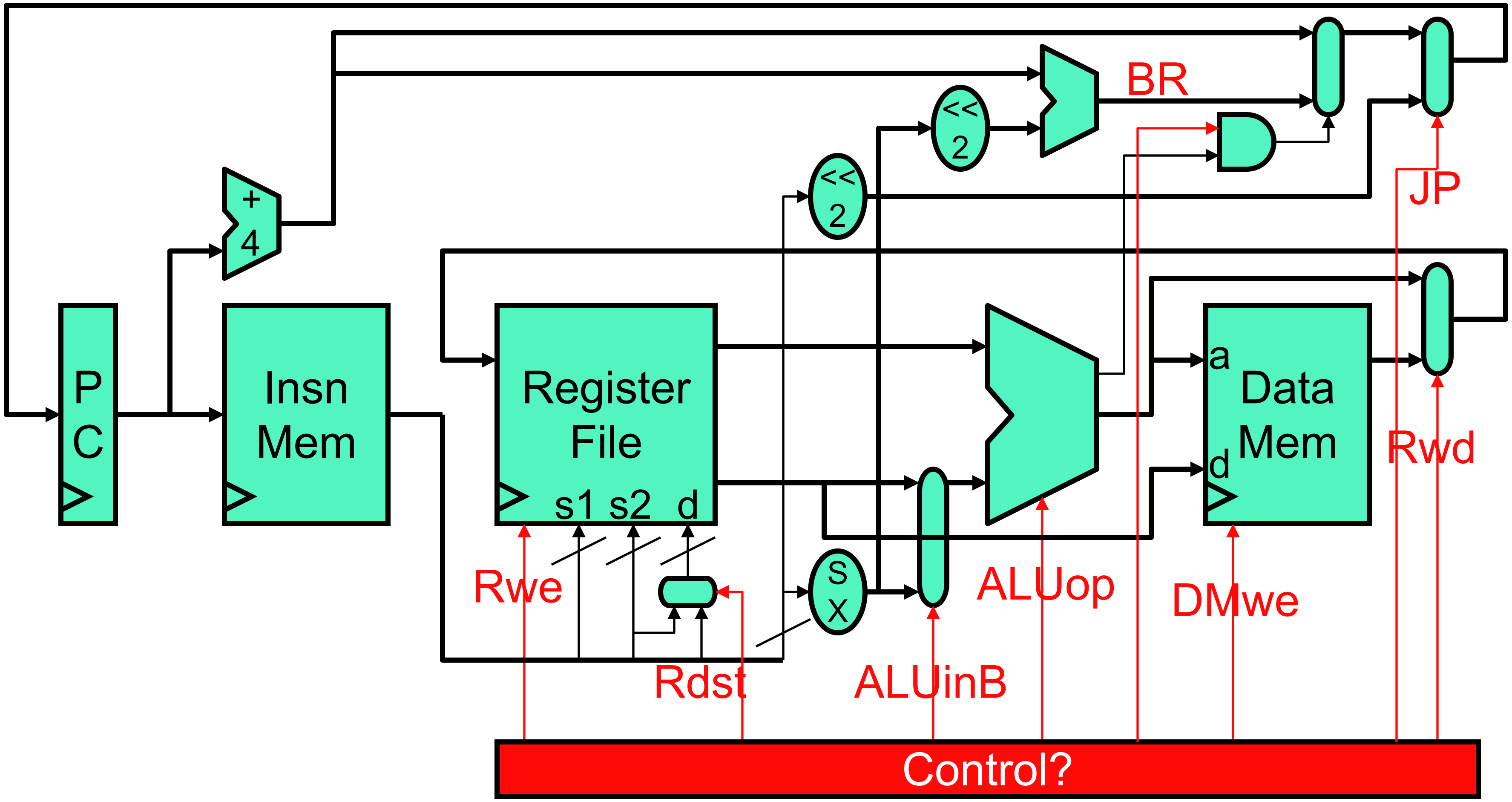
Example: Control for **beq**



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

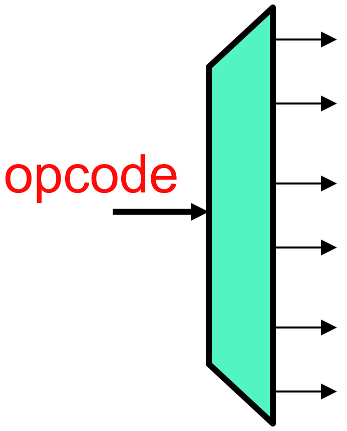
```
beq $5, $6, array_sum_loop
```

How Is Control Implemented?



Control Implementation: ROM

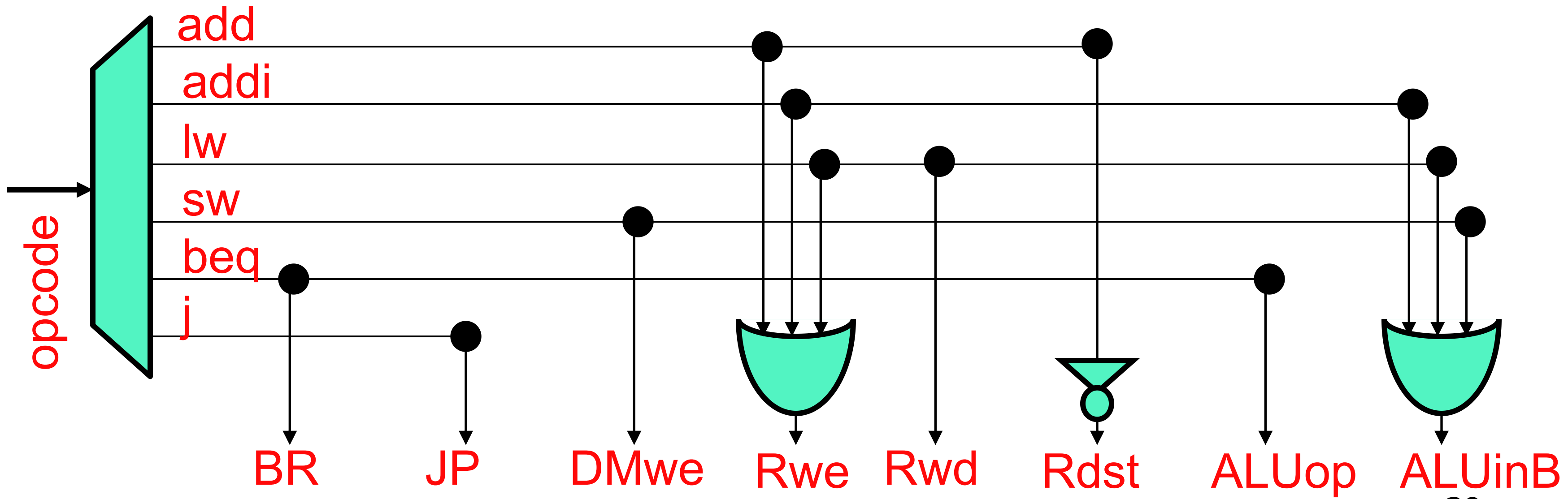
- **ROM (read only memory)**: like a RAM but unwritable
  - Bits in data words are control signals
  - Lines indexed by opcode
  - Example: ROM control for 6-insn MIPS datapath
  - X is "don't care"



	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

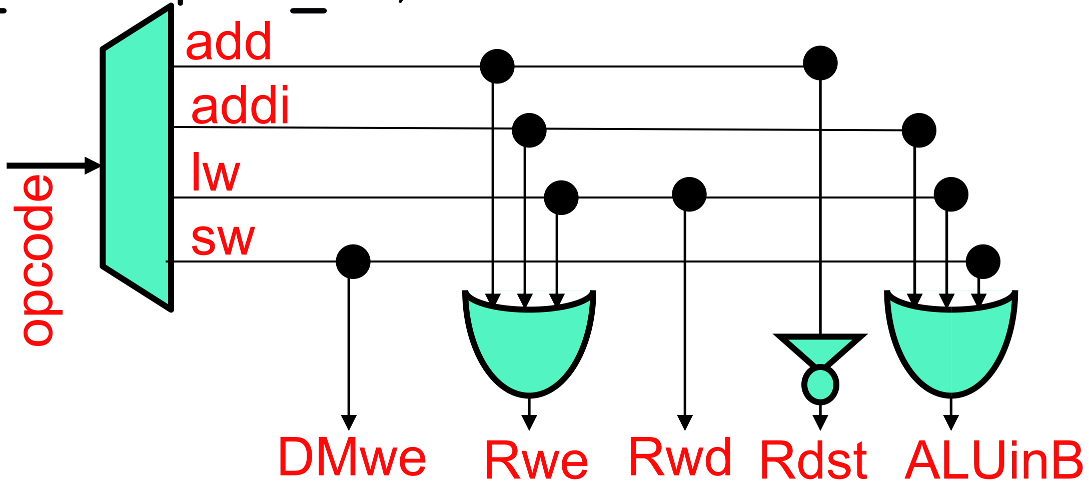
Control Implementation: Logic

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



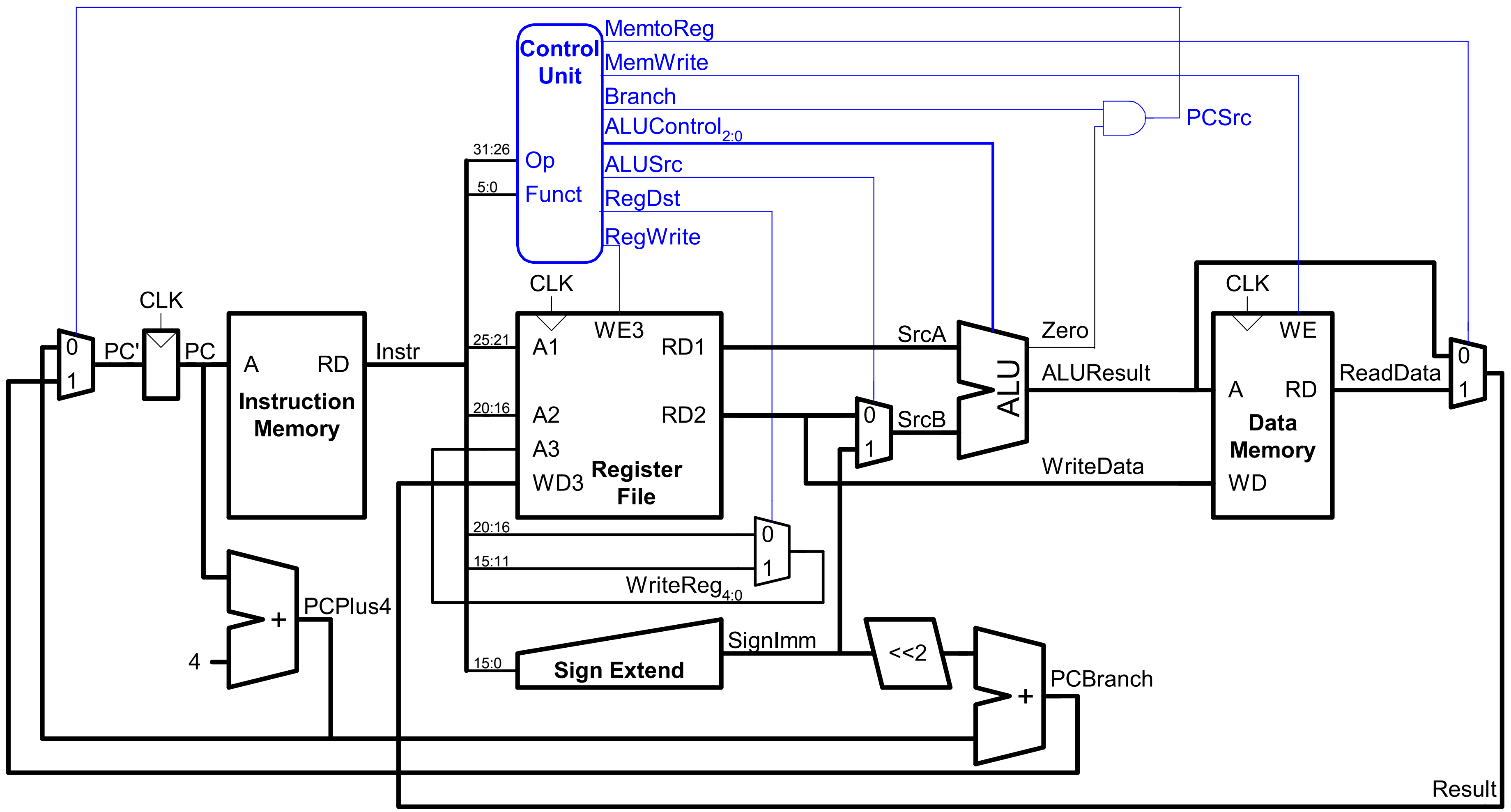
Control Logic in Verilog

```
wire [31:0] insn;
wire [5:0] func = insn[5:0]
wire [5:0] opcode = insn[31:26];
wire is_add = ((opcode == 6'h00) & (func == 6'h20));
wire is_addi = (opcode == 6'h0F);
wire is_lw = (opcode == 6'h23);
wire is_sw = (opcode == 6'h2A);
wire ALUinB = is_addi | is_lw | is_sw;
wire Rwe = is_add | is_addi | is_lw;
wire Rwd = is_lw;
wire Rdst = ~is_add;
wire DMwe = is_sw;
```



Single-Cycle Performance

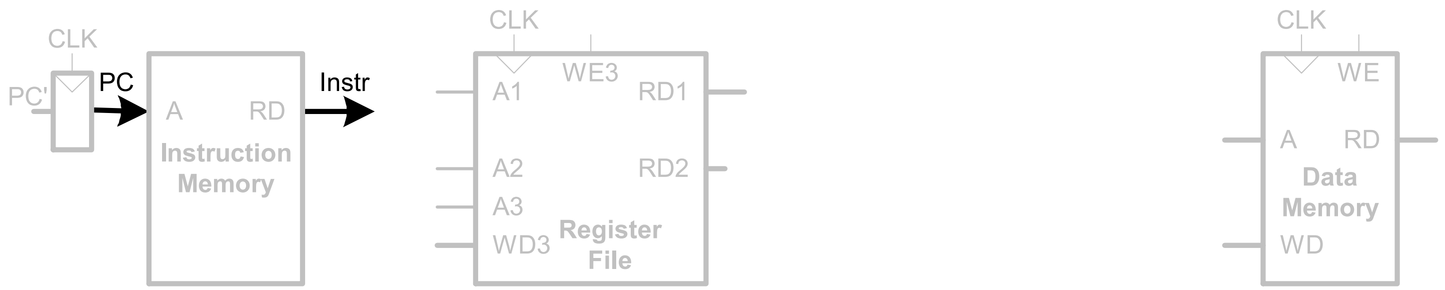
Single-Cycle Datapath Performance



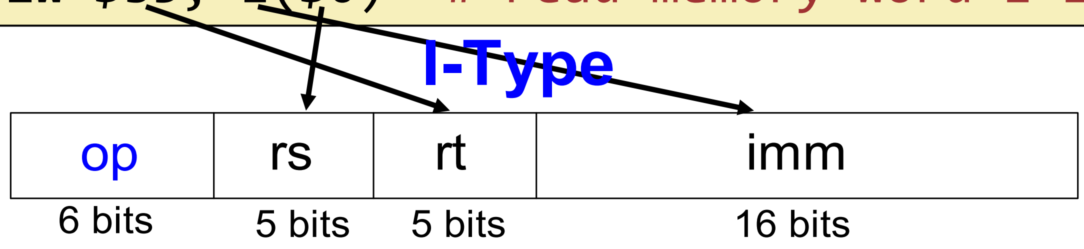
Single-cycle processor. Harris and Harris, Chapter 7.3.

Example: Single-Cycle Datapath: `lw` fetch

- ***STEP 1:*** **Fetch instruction**

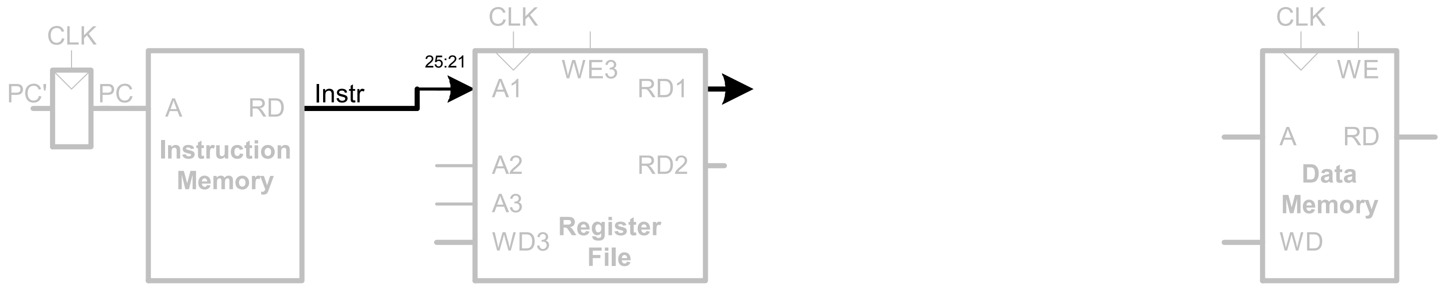


```
lw $s3, 1($0)   # read memory word 1 into $s3
```



Single-Cycle Datapath: `lw` register read

- *STEP 2:* **Read source operands from register file**



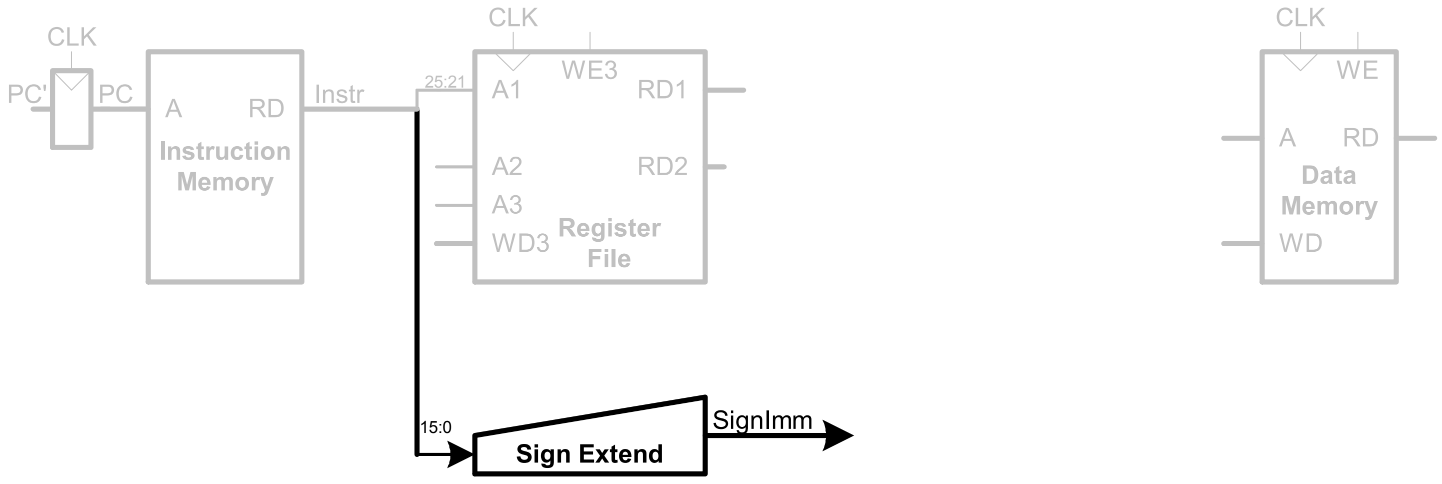
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` immediate

- ***STEP 3:*** **Sign-extend the immediate**



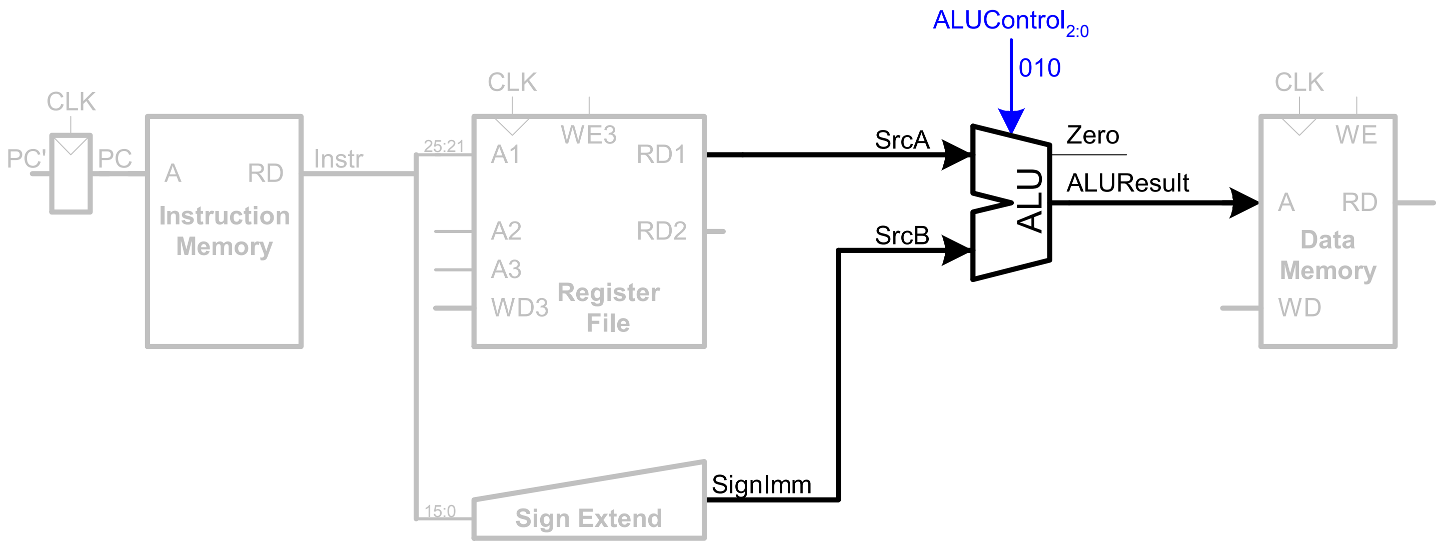
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` address

- ***STEP 4:*** **Compute the memory address**



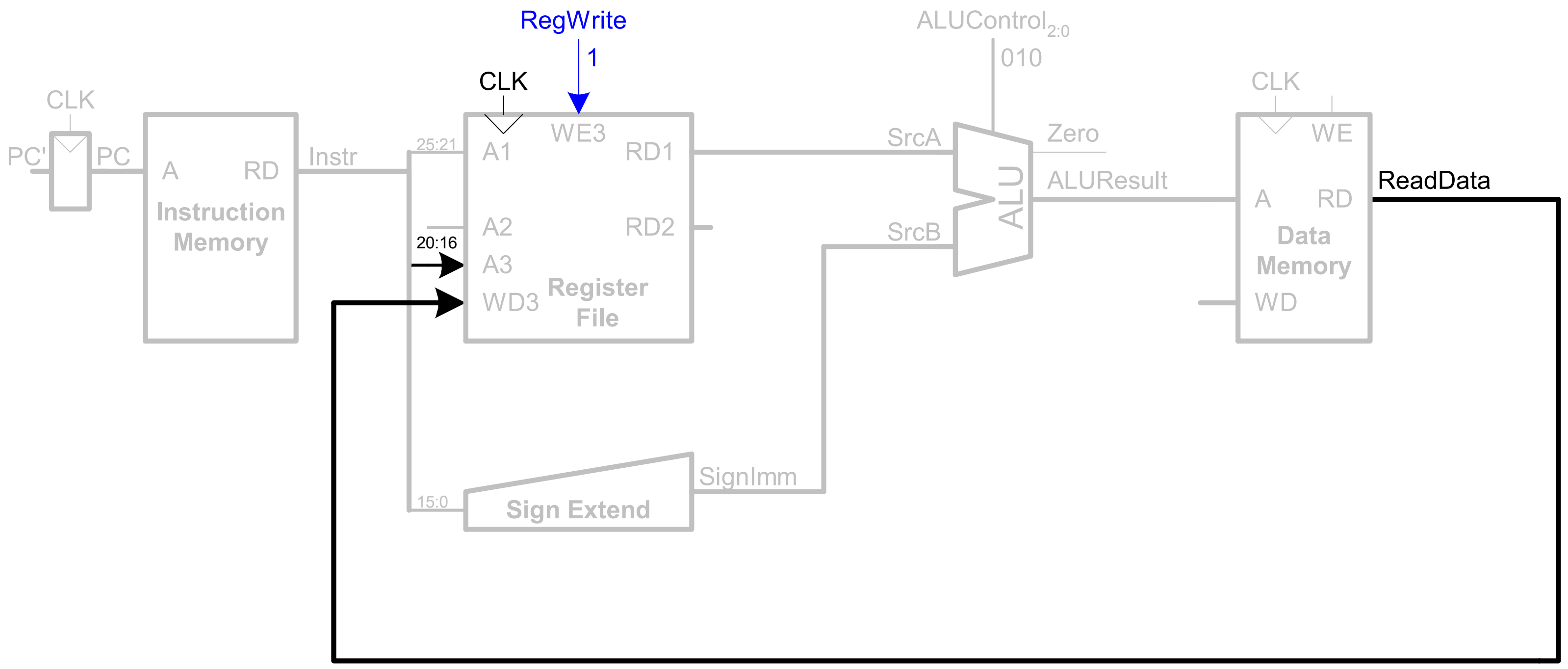
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` memory read

- ***STEP 5:*** **Read from memory and write back to register file**



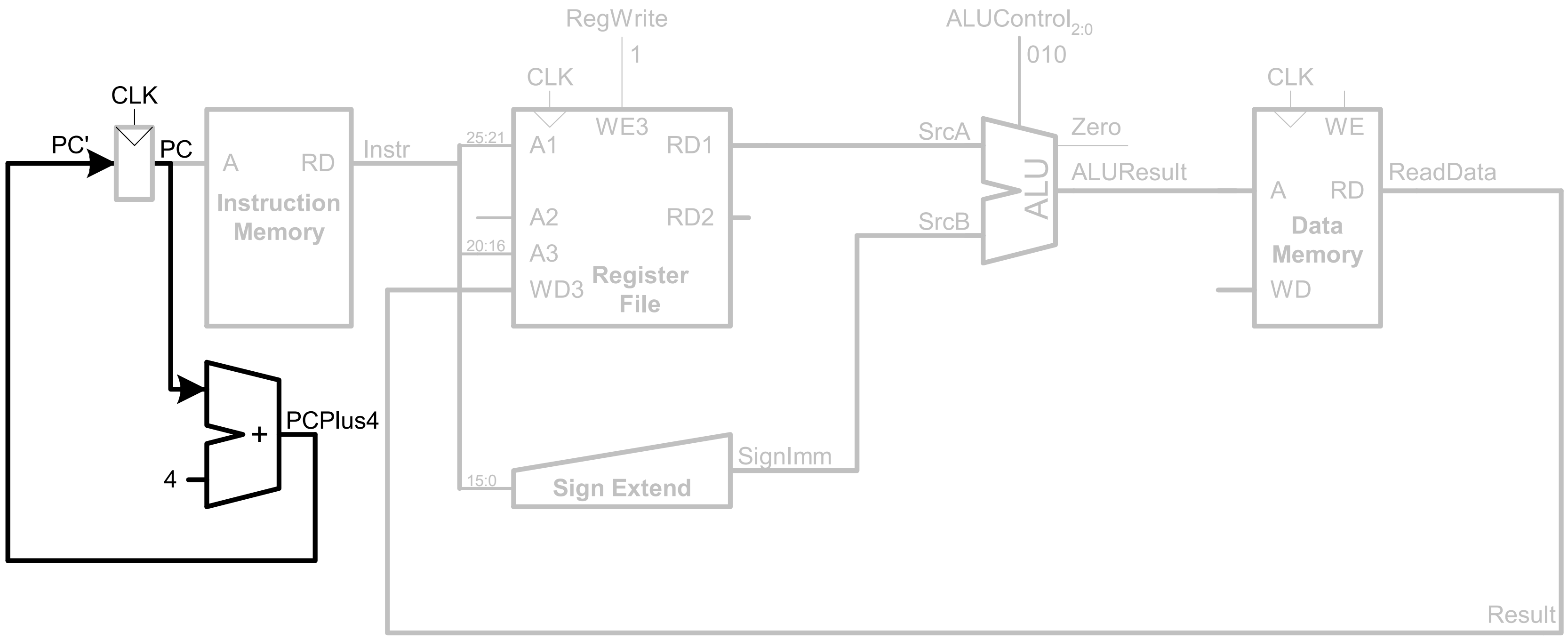
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` PC increment

- ***STEP 6:*** **Determine address of next instruction**

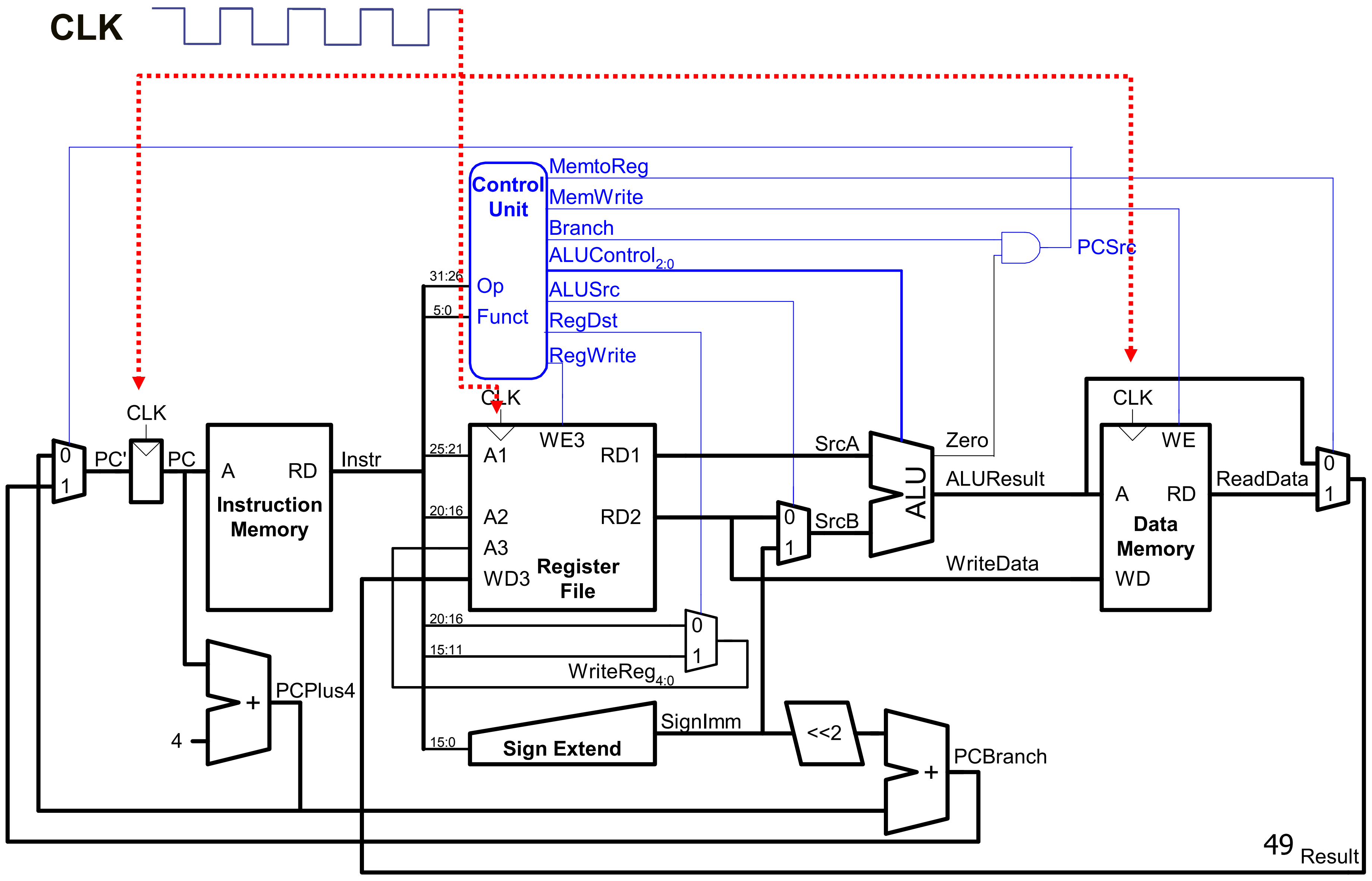


```
lw $s3, 1($0)  # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle CPU



Evaluating the Single-Cycle Microarchitecture

Performance Analysis Basics

- Execution time of an instruction
  - {CPI} x {clock cycle time}
    - CPI: Number of cycles it takes to execute an instruction

- Execution time of a program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - **{# of instructions} x {Average CPI} x {clock cycle time}**

A Single-Cycle Microarchitecture: Analysis

- Every instruction takes 1 cycle to execute
  - CPI (Cycles per instruction) is strictly 1

- How long each instruction takes is determined by how long the slowest instruction takes to execute
  - Even though many instructions do not need that long to execute

- Clock cycle time of the microarchitecture is determined by how long it takes to complete the slowest instruction
  - Critical path of the design is determined by the processing time of the slowest instruction

What is the Slowest Instruction to Process?

- Let's go back to the basics

- All six phases of the instruction processing cycle take a *single machine clock cycle* to complete

- Fetch
- Decode
- Evaluate Address
- Fetch Operands
- Execute
- Store Result

1. Instruction fetch (IF)
2. Instruction decode and register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)

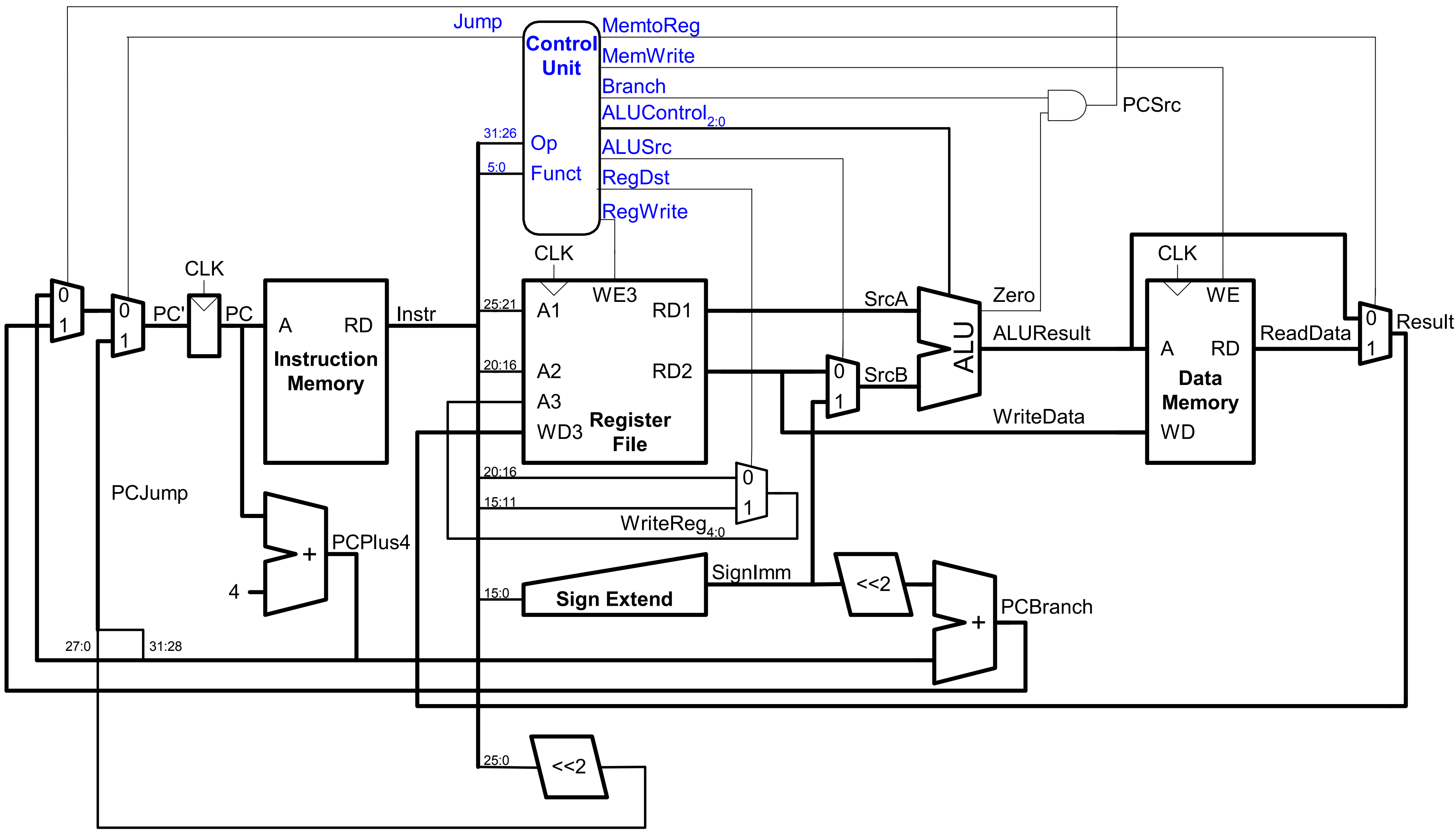
- Do each of the above phases take the same time (latency) for all instructions?

Example Single-Cycle Datapath Analysis

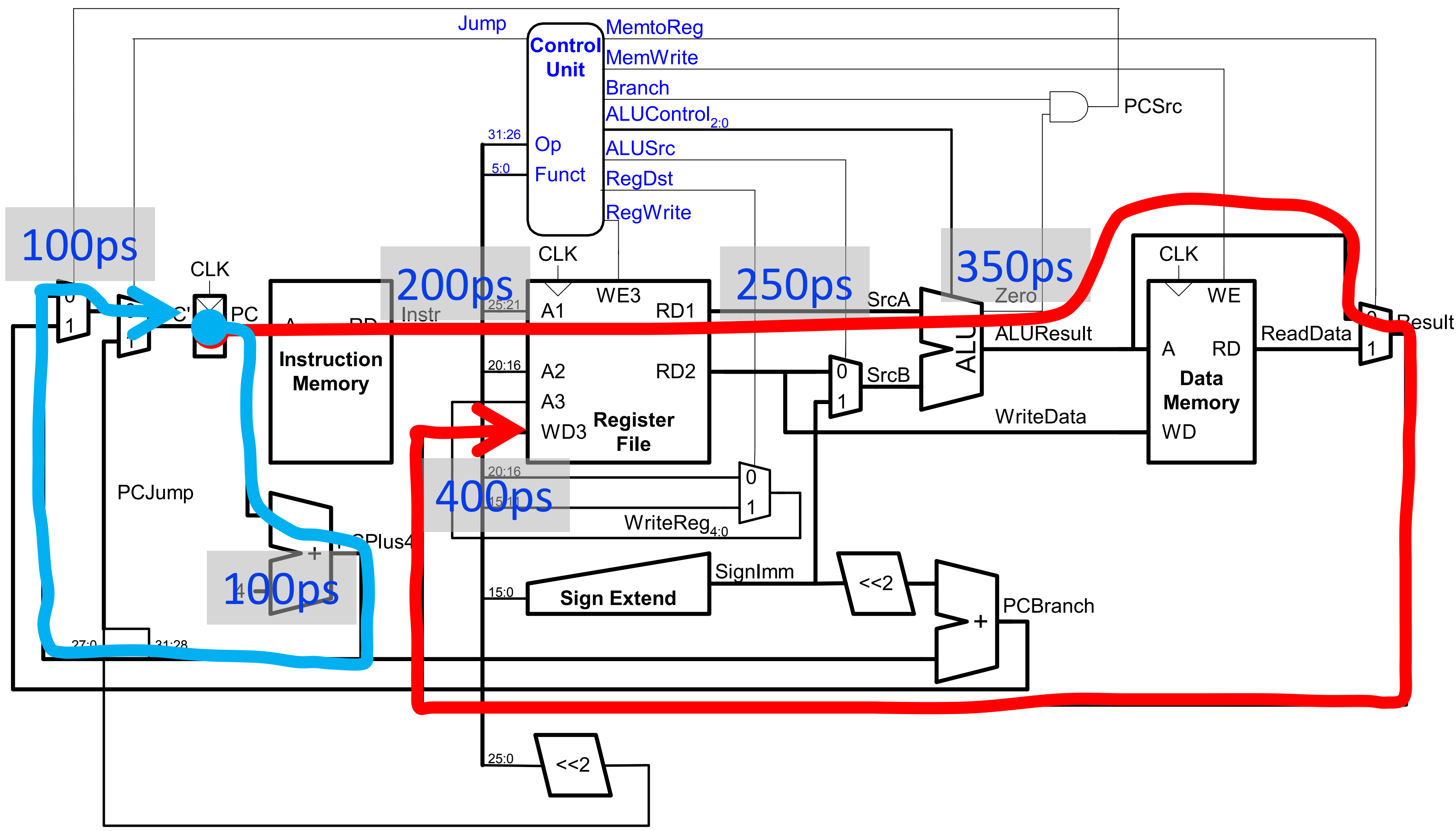
- Assume (for the design in the previous slide)
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

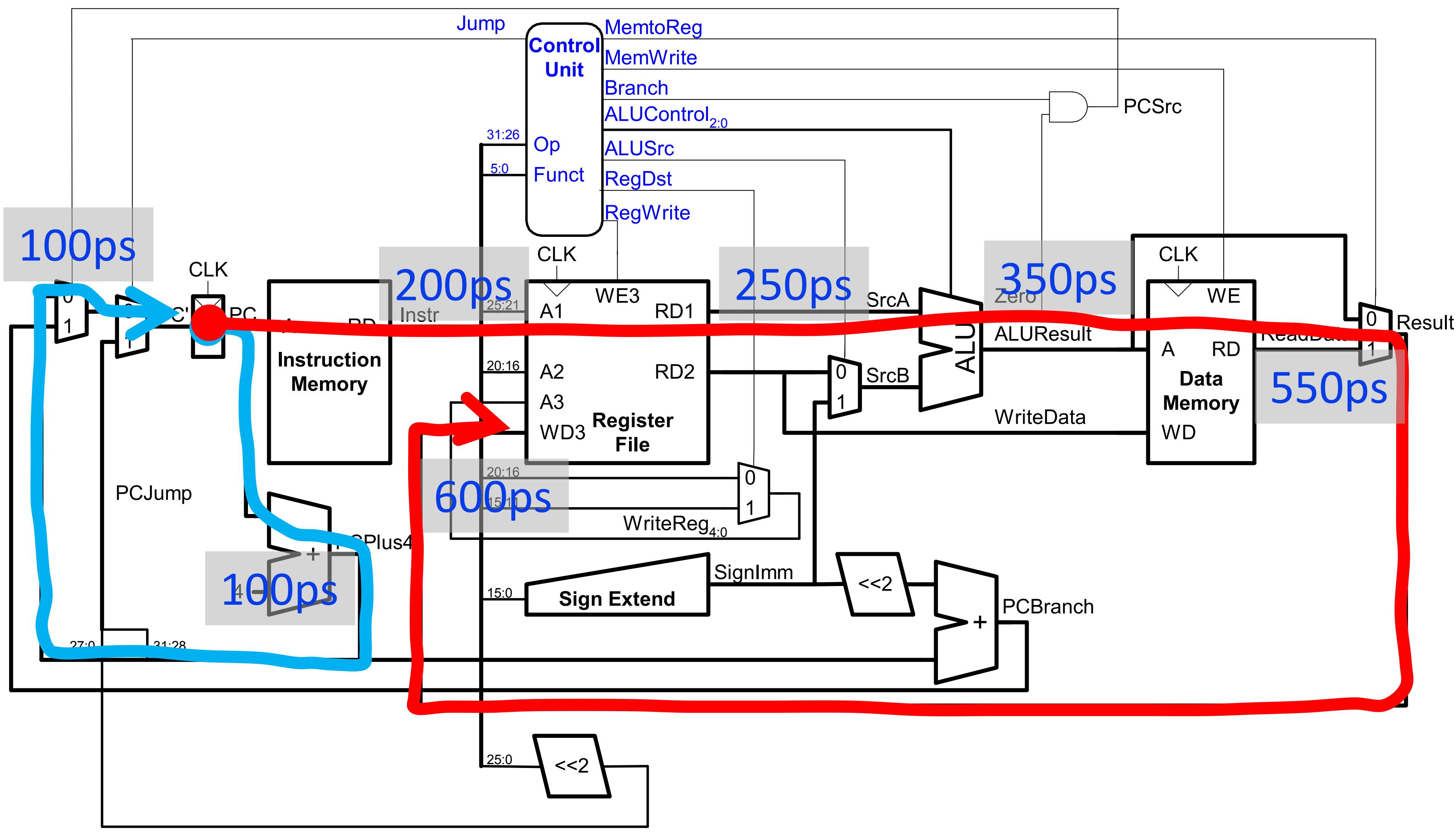
Let’s Find the Critical Path



R-Type and I-Type ALU

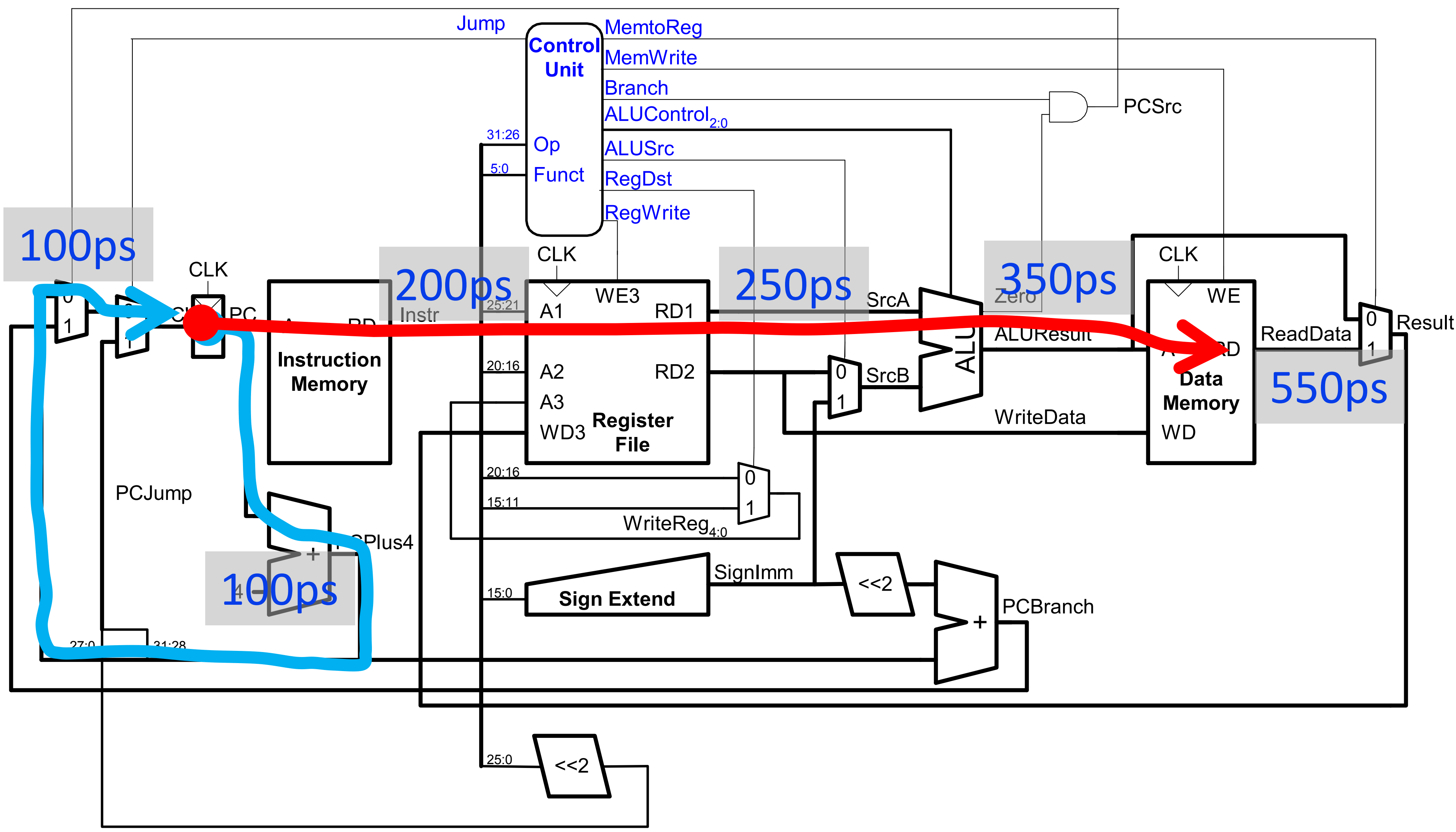


LW



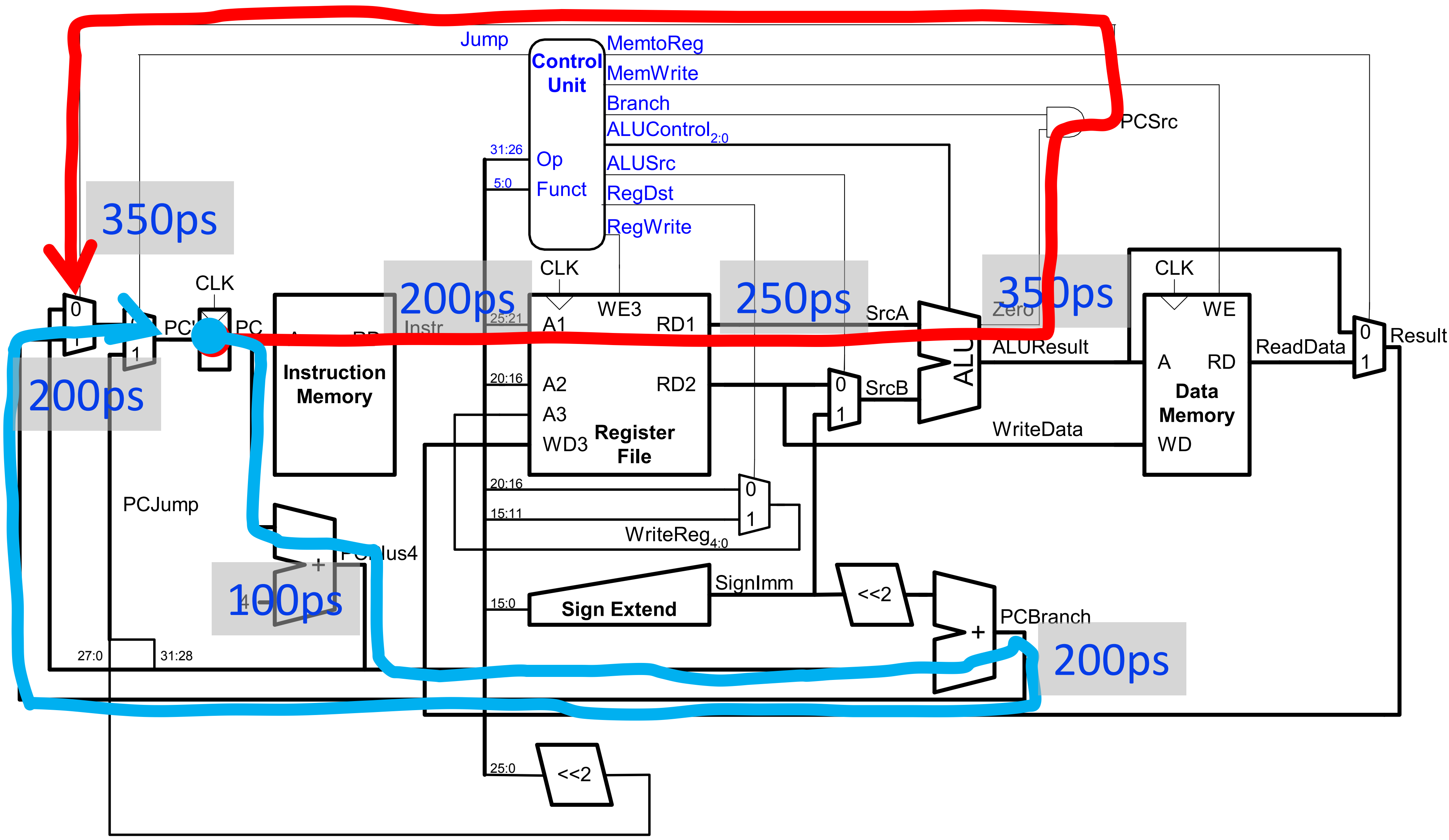
Jump
Control Unit
MemtoReg
MemWrite
Branch
ALUControl$_{2:0}$
ALUSrc
RegDst
RegWrite
Op
Funct
31:26
5:0
PCSrc
100ps
200ps
250ps
350ps
550ps
600ps
100ps
CLK
PC'
PC
A
RD
Instr
Instruction Memory
25:21
20:16
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
SrcA
SrcB
Zero
ALU
ALUResult
WriteData
WE
A
RD
Data Memory
WD
ReadData
Result
0
1
20:16
15:11
WriteReg$_{4:0}$
PCJump
PCPlus4
4
+
15:0
Sign Extend
SignImm
<<2
PCBranch
27:0
31:28
25:0

SW

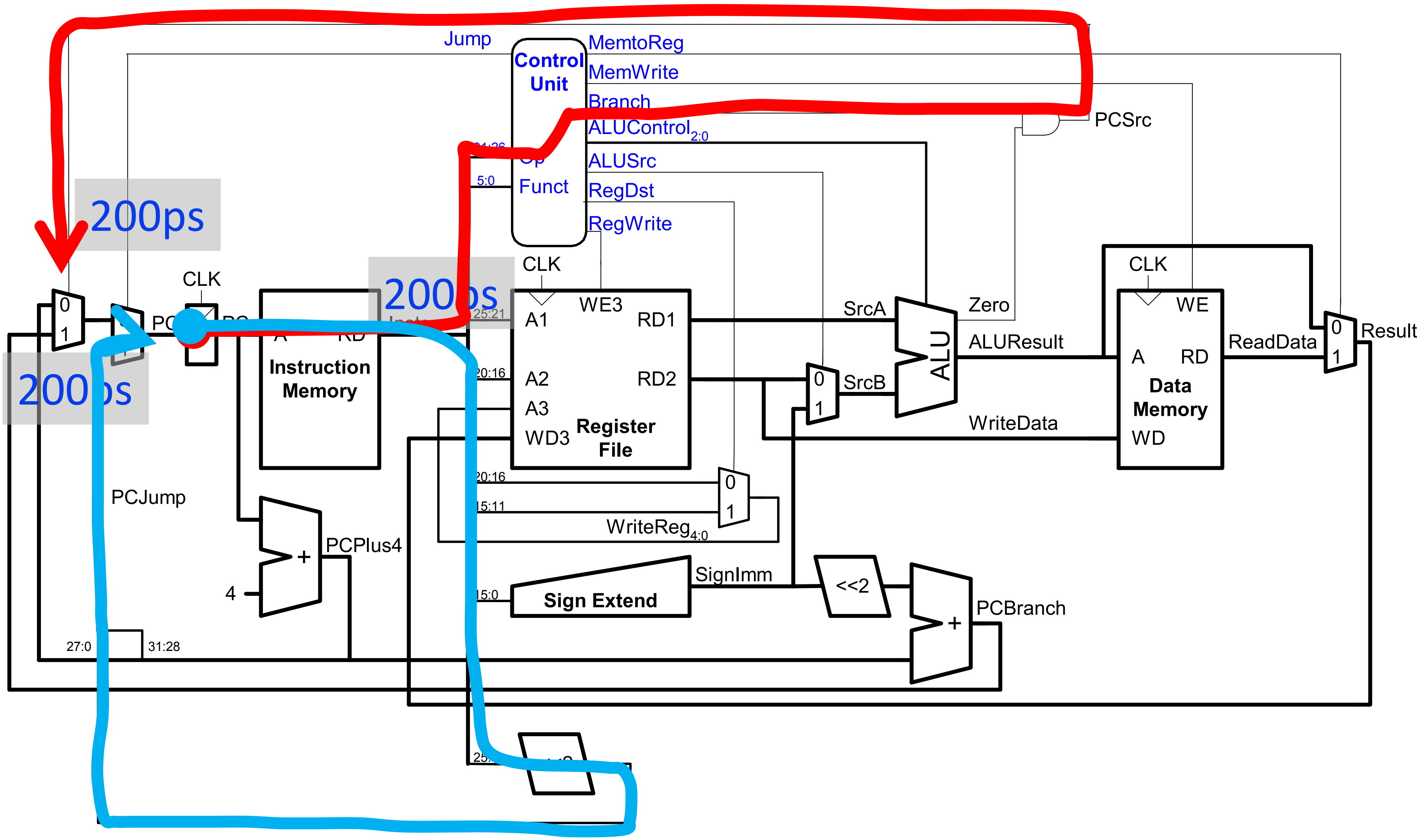


Jump
Control Unit
MemtoReg
MemWrite
Branch
ALUControl2:0
ALUSrc
RegDst
RegWrite
Op
Funct
31:26
5:0
PCSrc
CLK
100ps
200ps
250ps
350ps
550ps
100ps
PC
Instr
25:21
20:16
15:11
15:0
25:0
27:0
31:28
Instruction Memory
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
SrcA
SrcB
ALU
Zero
ALUResult
WriteData
Data Memory
WE
RD
WD
ReadData
Result
WriteReg4:0
SignImm
Sign Extend
<<2
PCBranch
PCPlus4
PCJump
0
1
+

Branch Taken



Jump



Jump
Control Unit
MemtoReg
MemWrite
Branch
ALUControl$_{2:0}$
ALUSrc
RegDst
RegWrite
PCSrc
Op
Funct
31:26
5:0
200ps
200ps
200ps
CLK
PC
Instr
A
RD
Instruction Memory
25:21
20:16
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
SrcA
SrcB
Zero
ALU
ALUResult
WriteData
A
RD
WE
Data Memory
WD
ReadData
Result
0
1
20:16
15:11
WriteReg$_{4:0}$
PCJump
PCPlus4
4
+
15:0
Sign Extend
SignImm
<<2
PCBranch
27:0
31:28
25:0

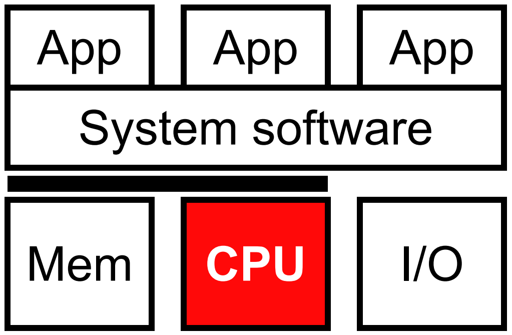
Single Cycle Arc: Complexity

- Contrived
  - All instructions run as slow as the slowest instruction

- Inefficient
  - All instructions run as slow as the slowest instruction
  - Must provide worst-case combinational resources in parallel as required by any instruction
  - Need to replicate a resource if it is needed more than once by an instruction during different parts of the instruction processing cycle

- Not easy to optimize/improve performance
  - Optimizing the common case does not work (e.g. common instructions)
  - Need to optimize the worst case all the time

Summary



- Overview of ISAs
- Datapath storage elements
- MIPS Datapath
- MIPS Control