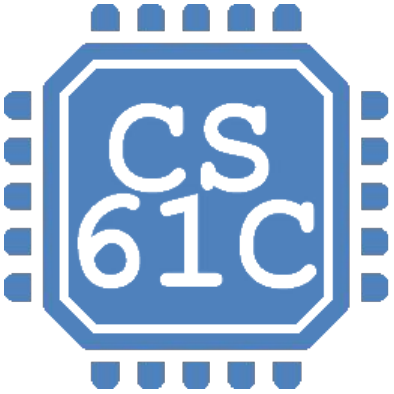
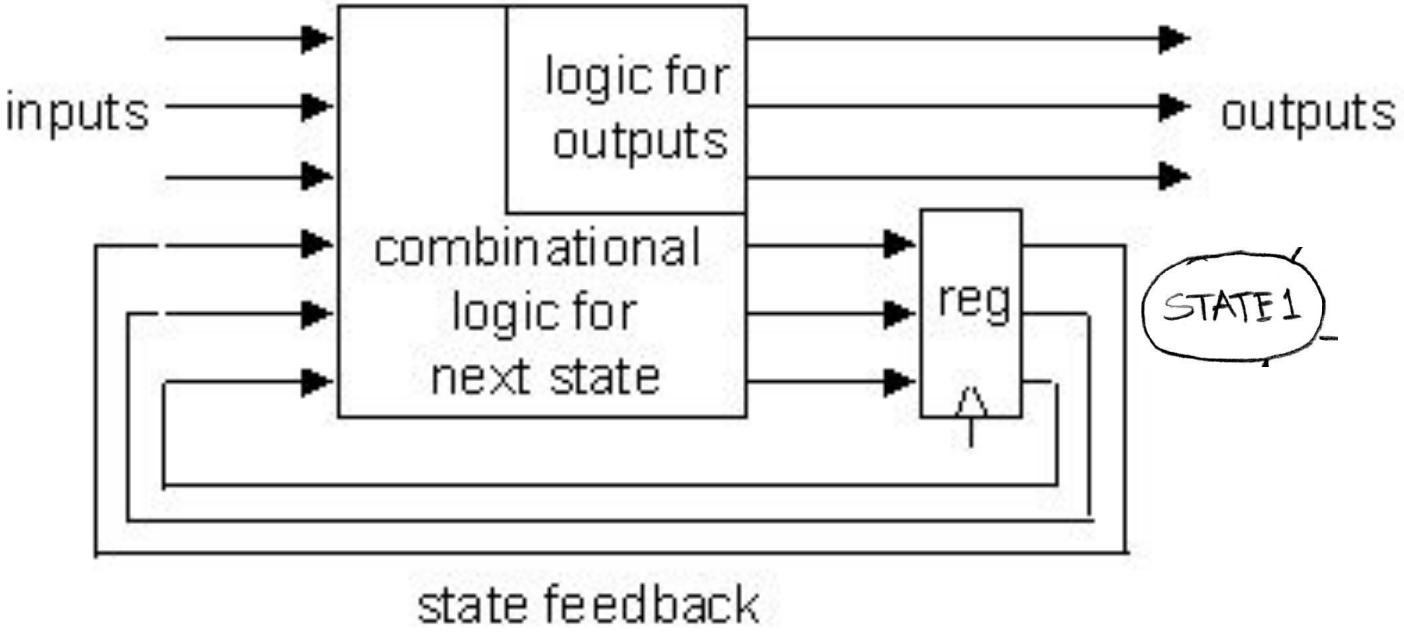



# Great Ideas in Computer Architecture

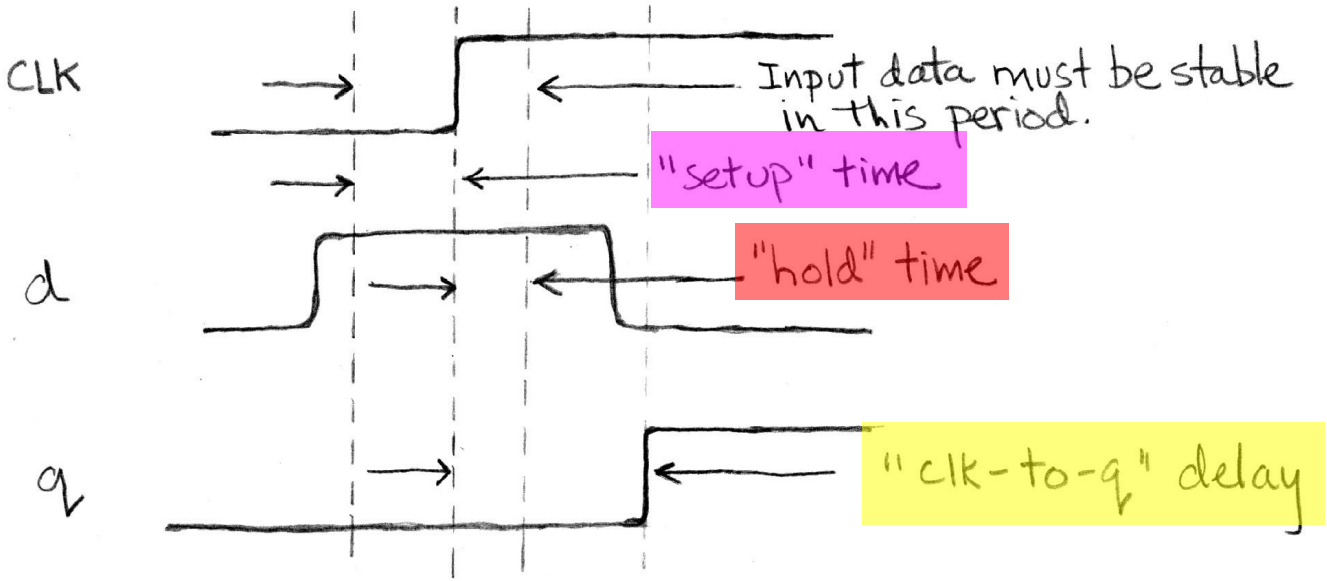
## *RISC-V CPU Datapath, Control Intro*

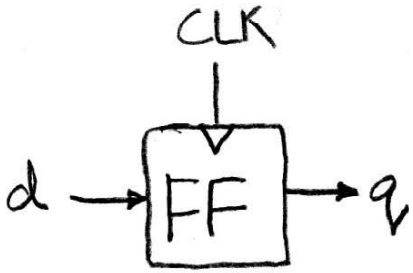
**Instructor:** Steven Ho

# Review -- SDS and Sequential Logic

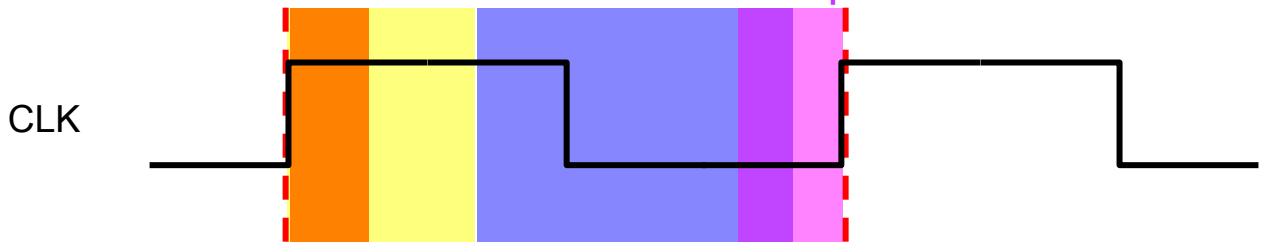


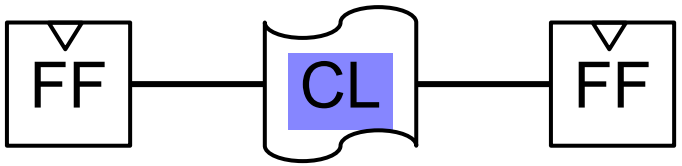
# Review -- Timing



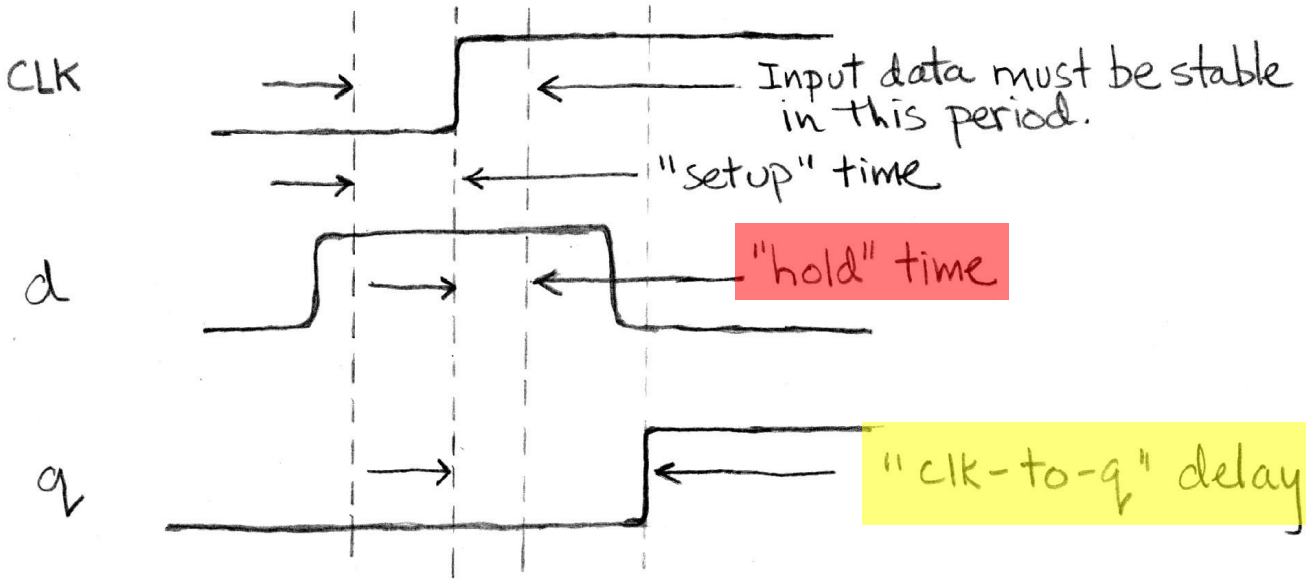

Setup Violation

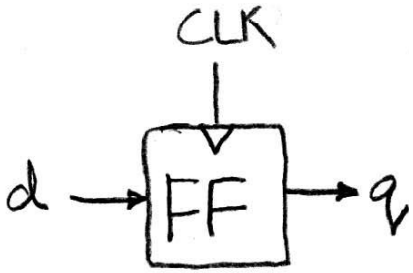


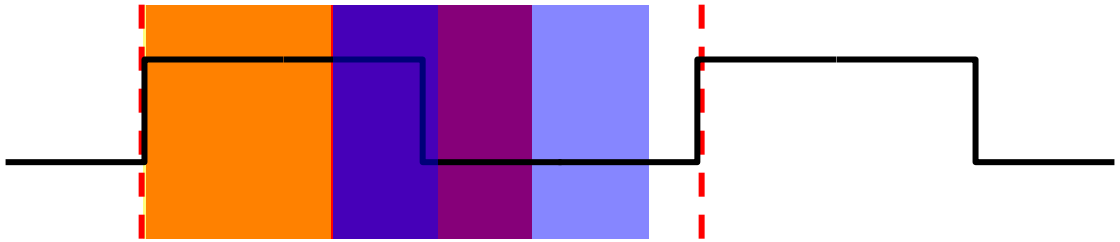

Clk-to-q + longest CL + setup time <= Clk period

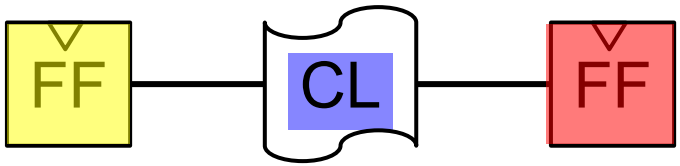
# Review -- Timing



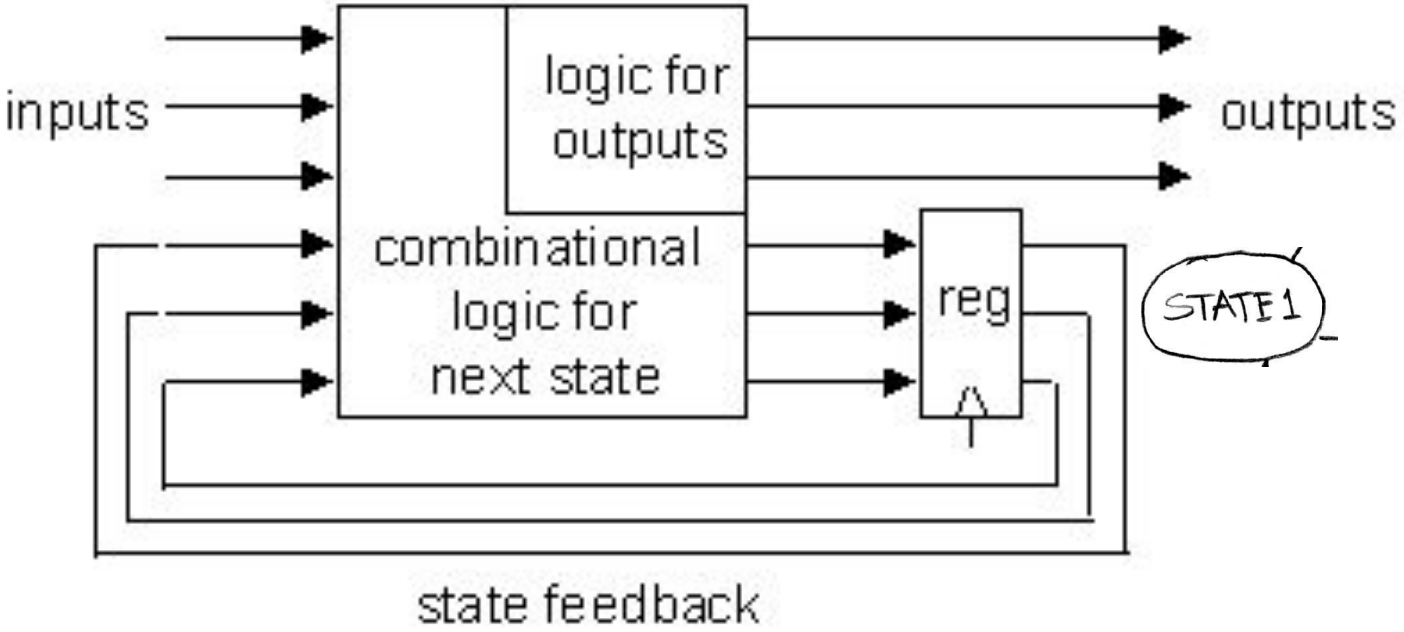

Hold Time Violation




Clk-to-q + shortest CL >= hold time

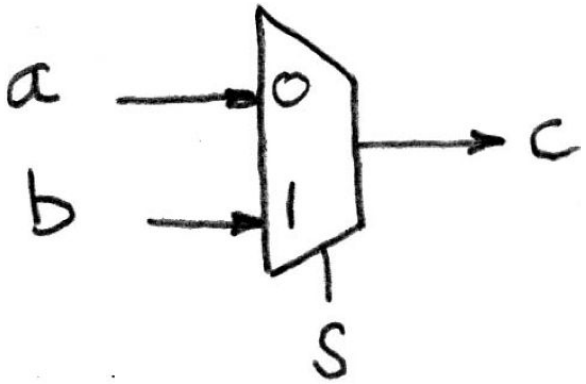
# Review -- SDS and Sequential Logic




Critical Path = Clk-to-q + longest CL + setup time between ANY two registers

# Review -- Combinational Logic

- Hardware is permanent. Always do everything you might want
- Use MUXes to pick from among input
  - S input bits selects one of $2^S$ inputs




- Ex: ALU

# Great Idea #1: Levels of Representation & Interpretation

**Higher-Level Language Program (e.g. C)**

*Compiler*

**Assembly Language Program (e.g. RISC-V)**

*Assembler*

**Machine Language Program (RISC-V)**

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw      $t0, 0($2)
lw      $t1, 4($2)
sw      $t1, 0($2)
sw      $t0, 4($2)
```

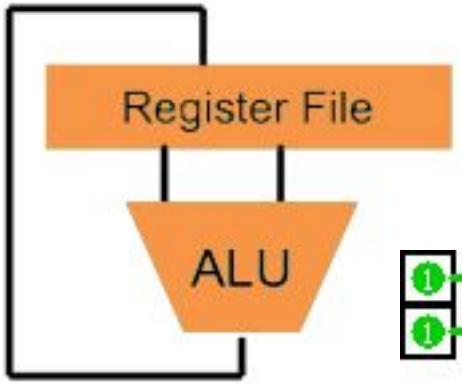
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

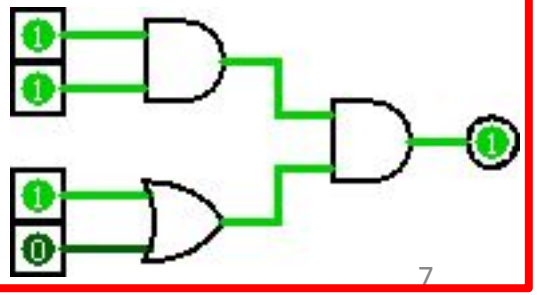
*Machine Interpretation*

**Hardware Architecture Description (e.g. block diagrams)**

*Architecture Implementation*

**Logic Circuit Description (Circuit Schematic Diagrams)**



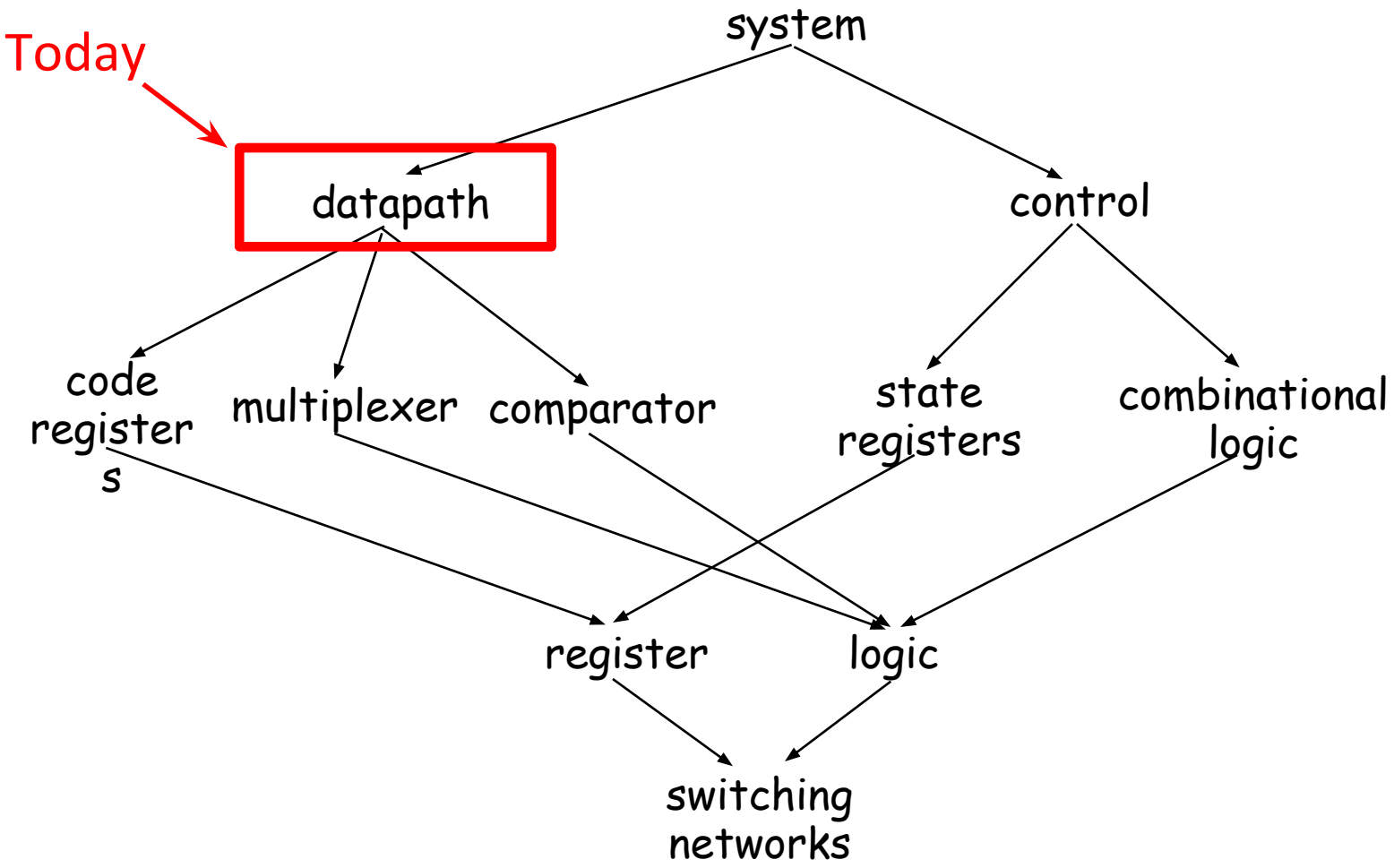




We are here

# Agenda

- Datapath Overview
- Assembling the Datapath Part 1
- Administrivia
- Processor Design Process
- Assembling the Datapath Part 2

# Hardware Design Hierarchy



# The Processor

- ***Processor (CPU):*** Instruction Set Architecture (ISA) implemented **directly in hardware**
  - *Datapath:* part of the processor that contains the hardware necessary to perform operations required by the processor ("the brawn")
  - *Control:* part of the processor (also in hardware) which tells the datapath what needs to be done ("the brain")

# Executing an Instruction

Very generally, what steps do you take (order matters!) to figure out the effect/result of the next RISC-V instruction?

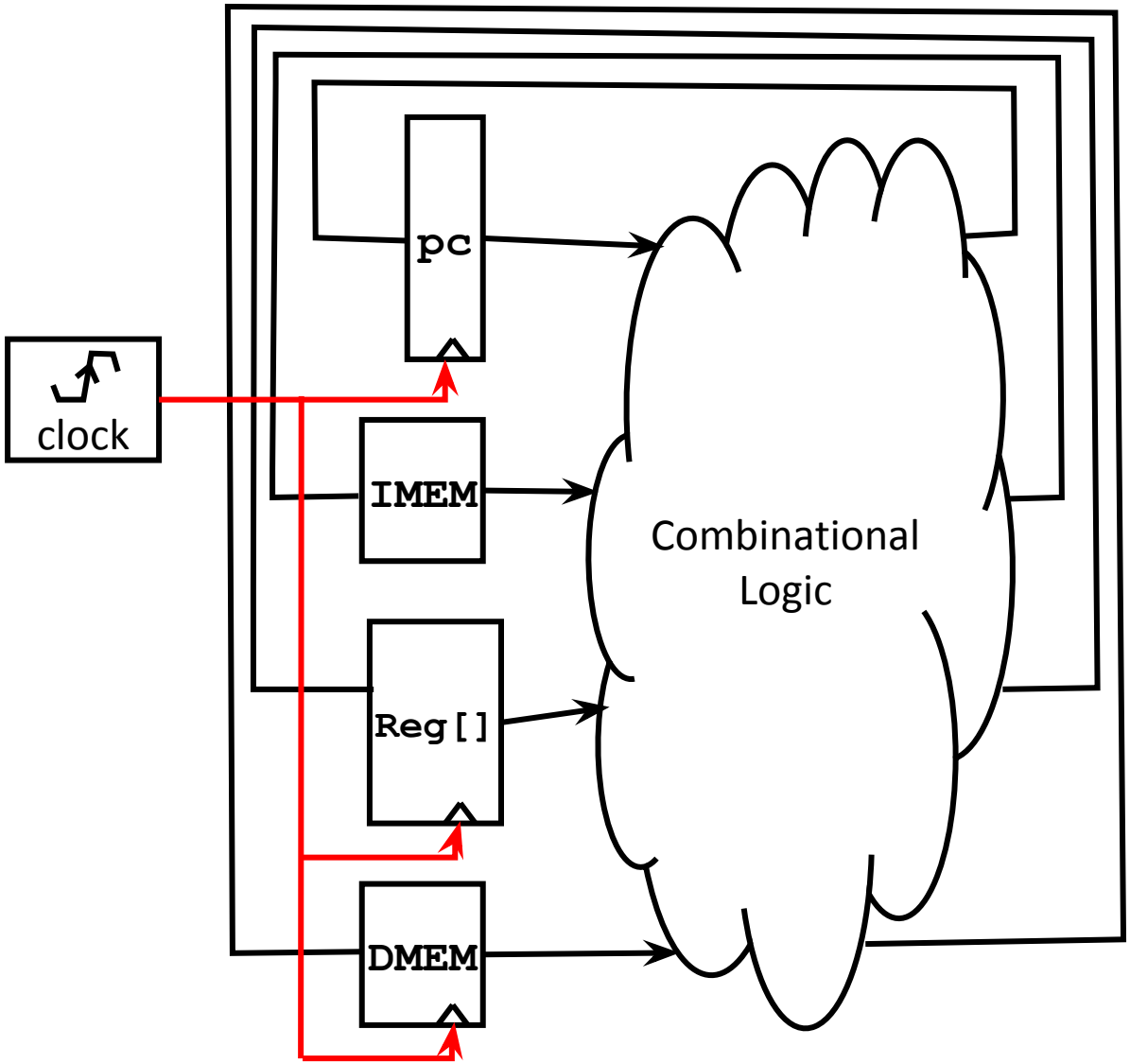
- Get the instruction `add s0,t0,t1`
- What instruction is it? `add`
- Gather data read `R[t0],R[t1]`
- Perform operation calc `R[t0]+R[t1]`
- Store result save into `s0`

# State Required by RV32I ISA

Each instruction reads and updates this state during execution:

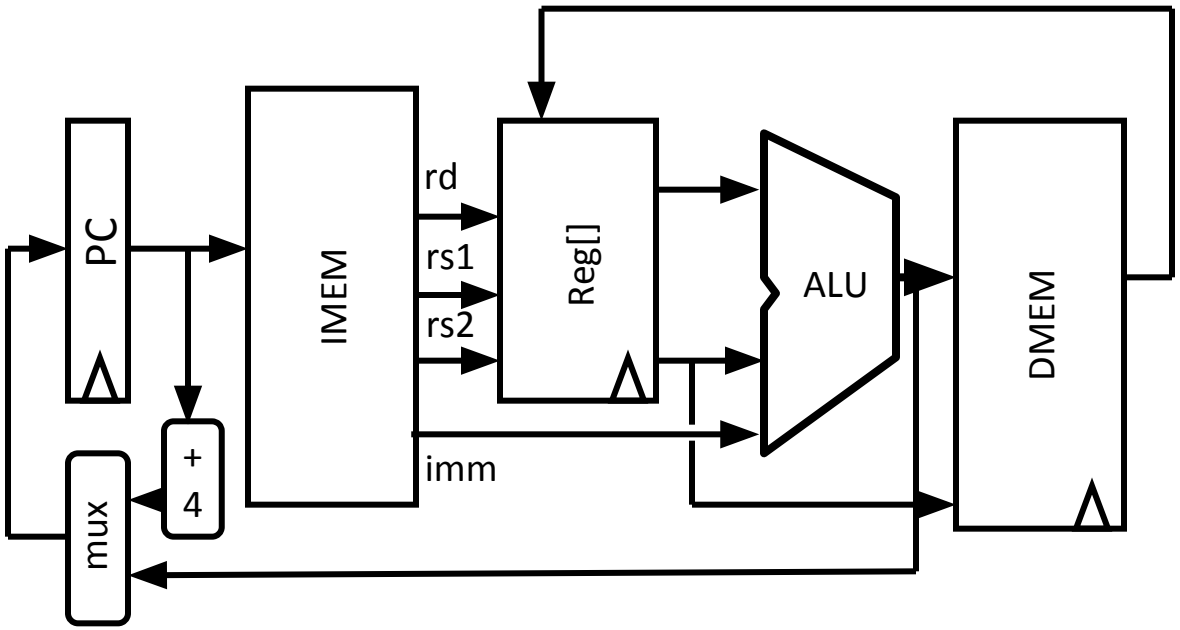
- Registers (**`x0..x31`**)
  - Register file (or *regfile*) **`Reg`** holds 32 registers x 32 bits/register: **`Reg[0].. Reg[31]`**
  - First register read specified by *rs1* field in instruction
  - Second register read specified by *rs2* field in instruction
  - Write register (destination) specified by *rd* field in instruction
  - **`x0`** is always 0 (writes to **`Reg[0]`** are ignored)
- Program Counter (**`PC`**)
  - Holds address of current instruction
- Memory (**`MEM`**)
  - Holds both instructions & data, in one 32-bit byte-addressed memory space
  - We'll use separate memories for instructions (**`IMEM`**) and data (**`DMEM`**)
    - *Later we'll replace these with instruction and data caches*
  - Instructions are read (*fetched*) from instruction memory (assume **`IMEM`** read-only)
  - Load/store instructions access data memory

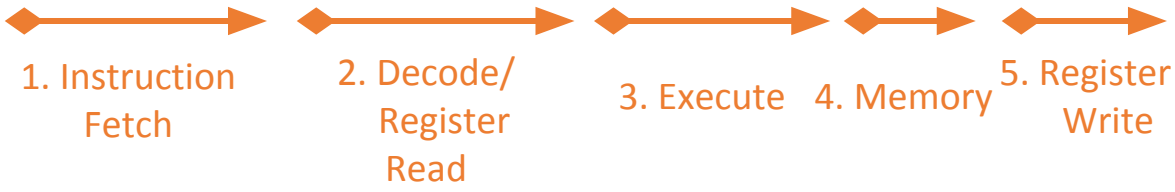
# One-Instruction-Per-Cycle RISC-V Machine

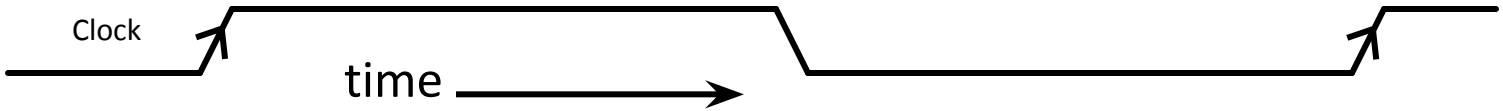



- On every tick of the clock, the computer executes one instruction
- Current state outputs drive the inputs to the combinational logic, whose outputs settles at the values of the state before the next clock edge
- At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle

# Basic Phases of Instruction Execution



# Why Five Stages?

- Could we have a different number of stages?
  - Yes, and other architectures do
- So why does RISC-V have five if instructions tend to idle for at least one stage?
  - The five stages are the union of all the operations needed by all the instructions
  - There is one instruction that uses all five stages: *load* (`lw`/`lb`)

# Agenda

- Datapath Overview
- Assembling the Datapath Part 1
- Administrivia
- Processor Design Process
- Assembling the Datapath Part 2

# Implementing the **add** instruction

| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |
|---|---|---|---|---|---|---|

**`add rd, rs1, rs2`**

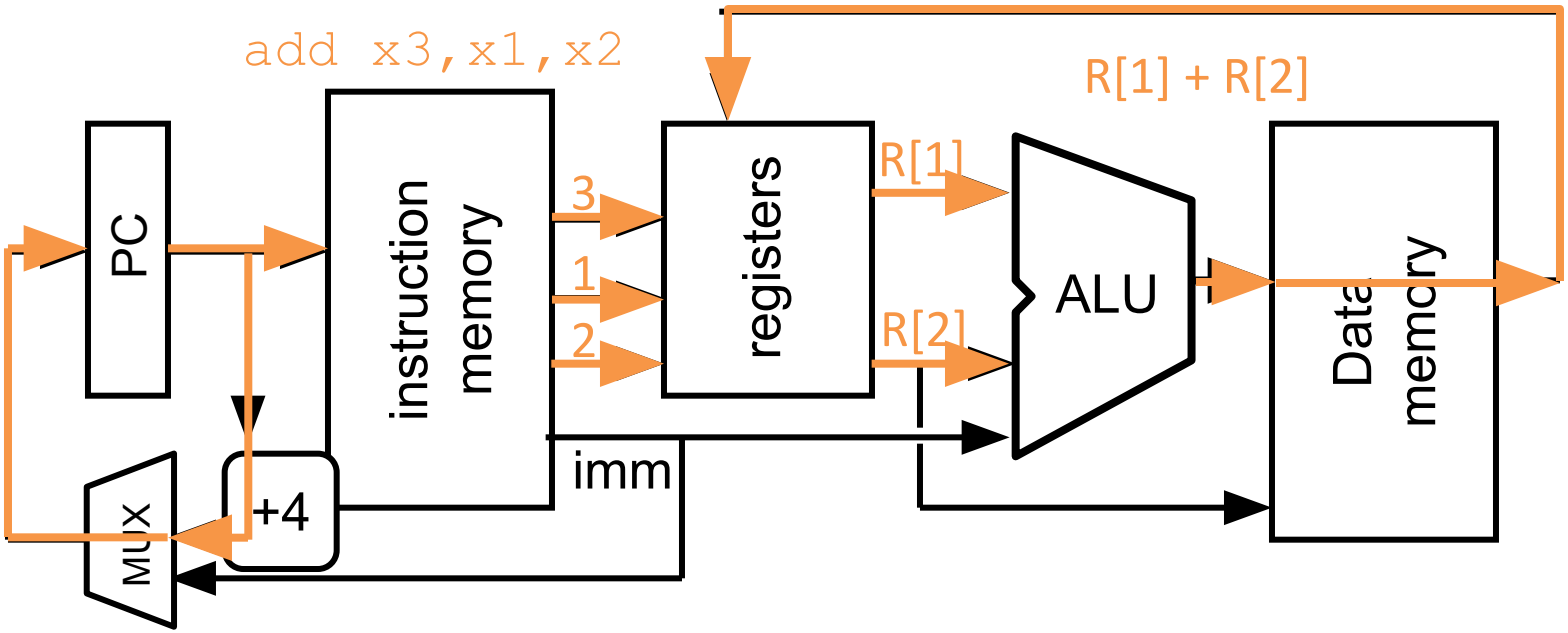
- Instruction makes two changes to machine's state:
  - **`Reg[rd] = Reg[rs1] + Reg[rs2]`**
  - **`PC = PC + 4`**

# Datapath Walkthroughs (1/3)

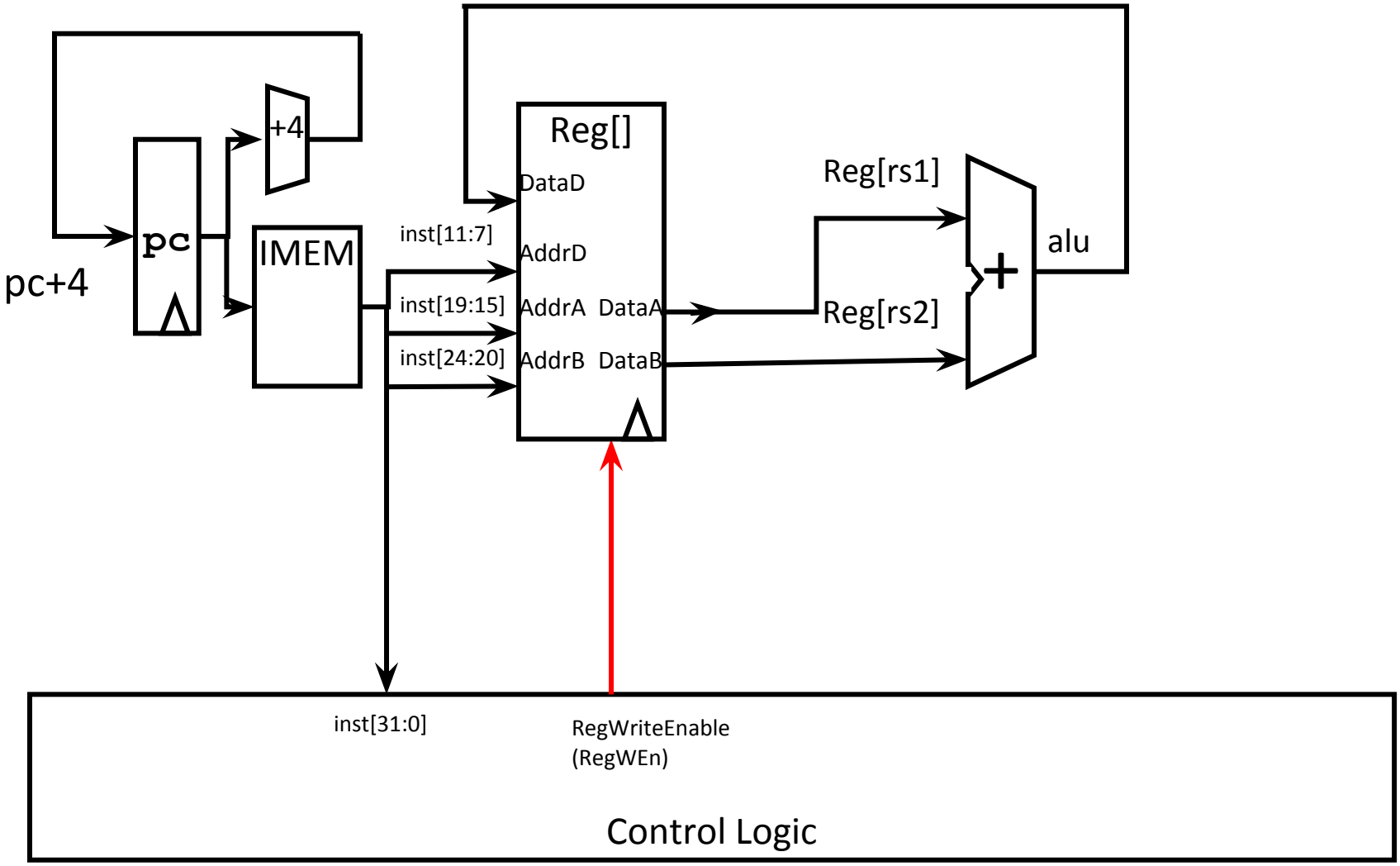
- `add x3,x1,x2` # r3 = r1+r2

1) IF: fetch this instruction, increment PC
2) ID: decode as `add`
   then read `R[1]` and `R[2]`
3) EX: add the two values retrieved in ID
4) MEM: idle (not using memory)
5) WB: write result of EX into `R[3]`

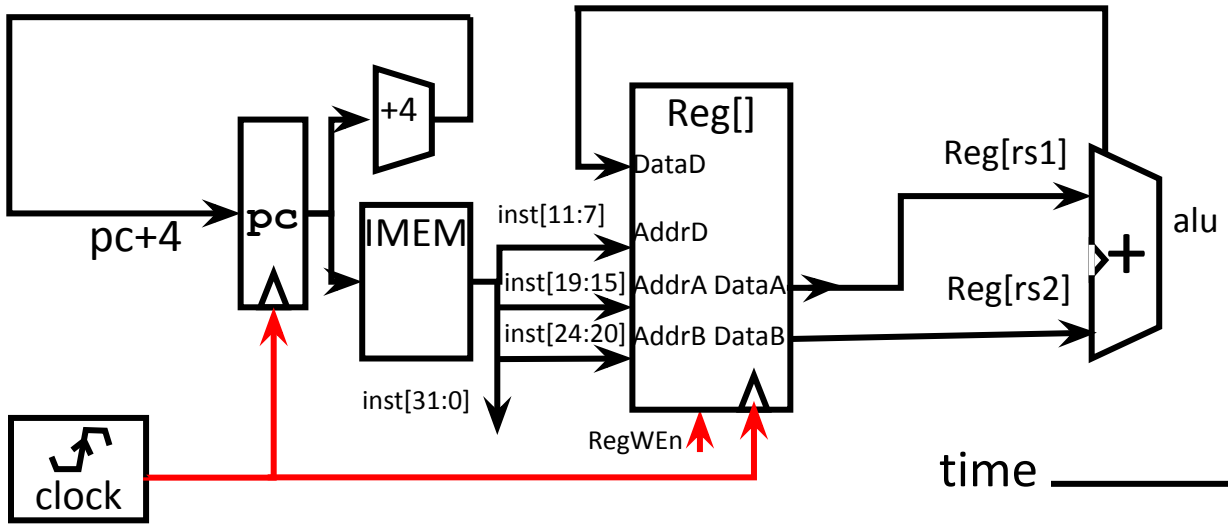
# Example: `add` Instruction

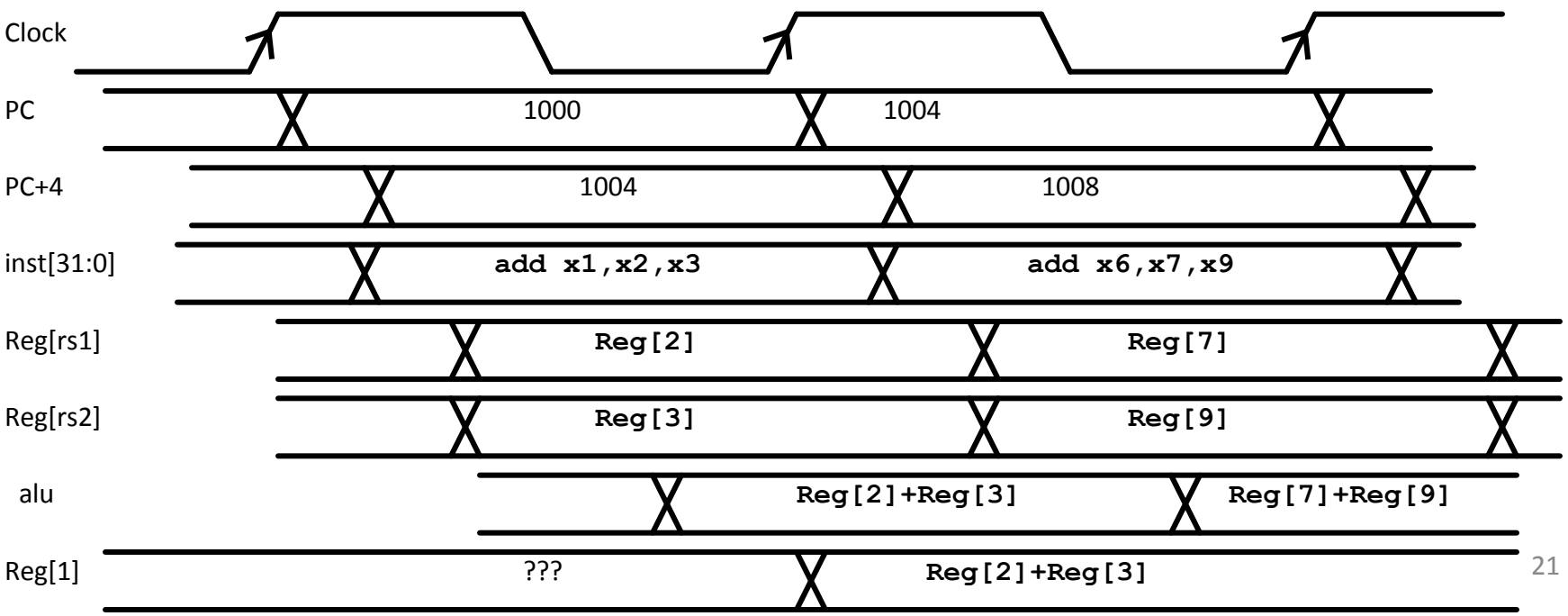


# Datapath for **add**



# Timing Diagram for **add**



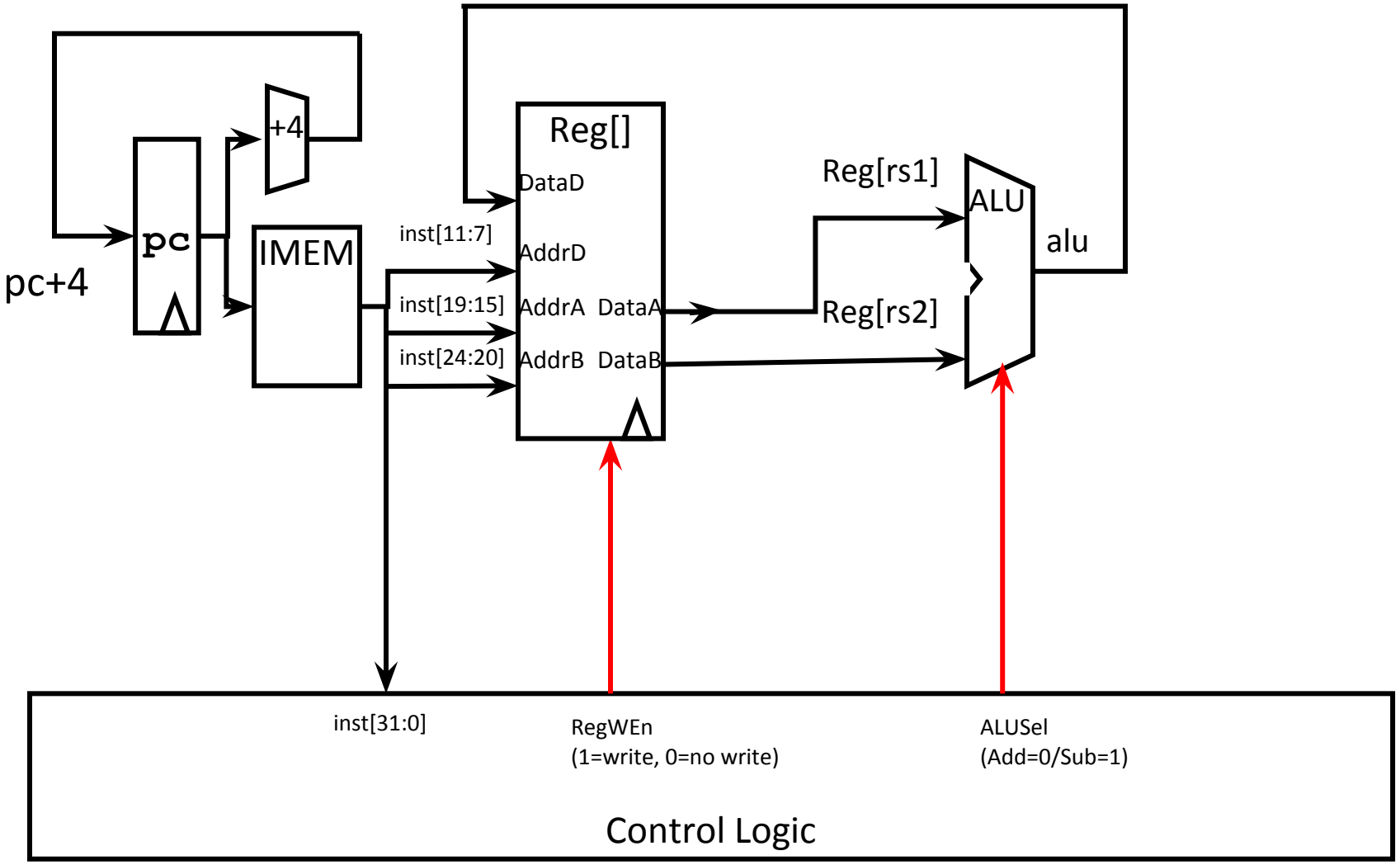
# Implementing the **`sub`** instruction

| | | | | | | |
|---|---|---|---|---|---|---|
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | SUB |

**`sub rd, rs1, rs2`**

- Almost the same as add, except now have to subtract operands instead of adding them
- **`inst[30]`** selects between add and subtract

# Datapath for `add/sub`



# Implementing other R-Format instructions

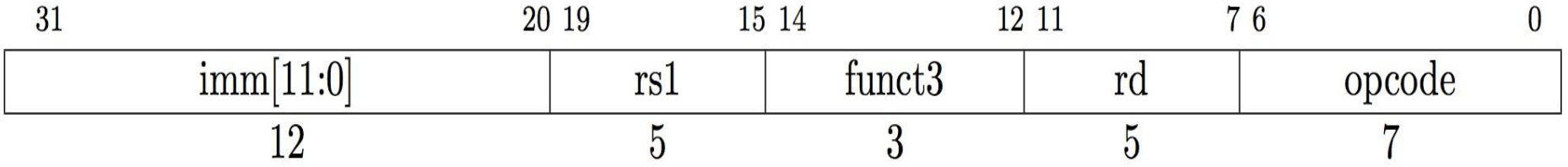
| | | | | | | |
|---|---|---|---|---|---|---|
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | SUB |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 | SLL |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 | SLT |
| 0000000 | rs2 | rs1 | 011 | rd | 0110011 | SLTU |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | XOR |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 | SRL |
| 0100000 | rs2 | rs1 | 101 | rd | 0110011 | SRA |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 | OR |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | AND |

- All implemented by decoding funct3 and funct7 fields and selecting appropriate ALU function

# Implementing the `addi` instruction

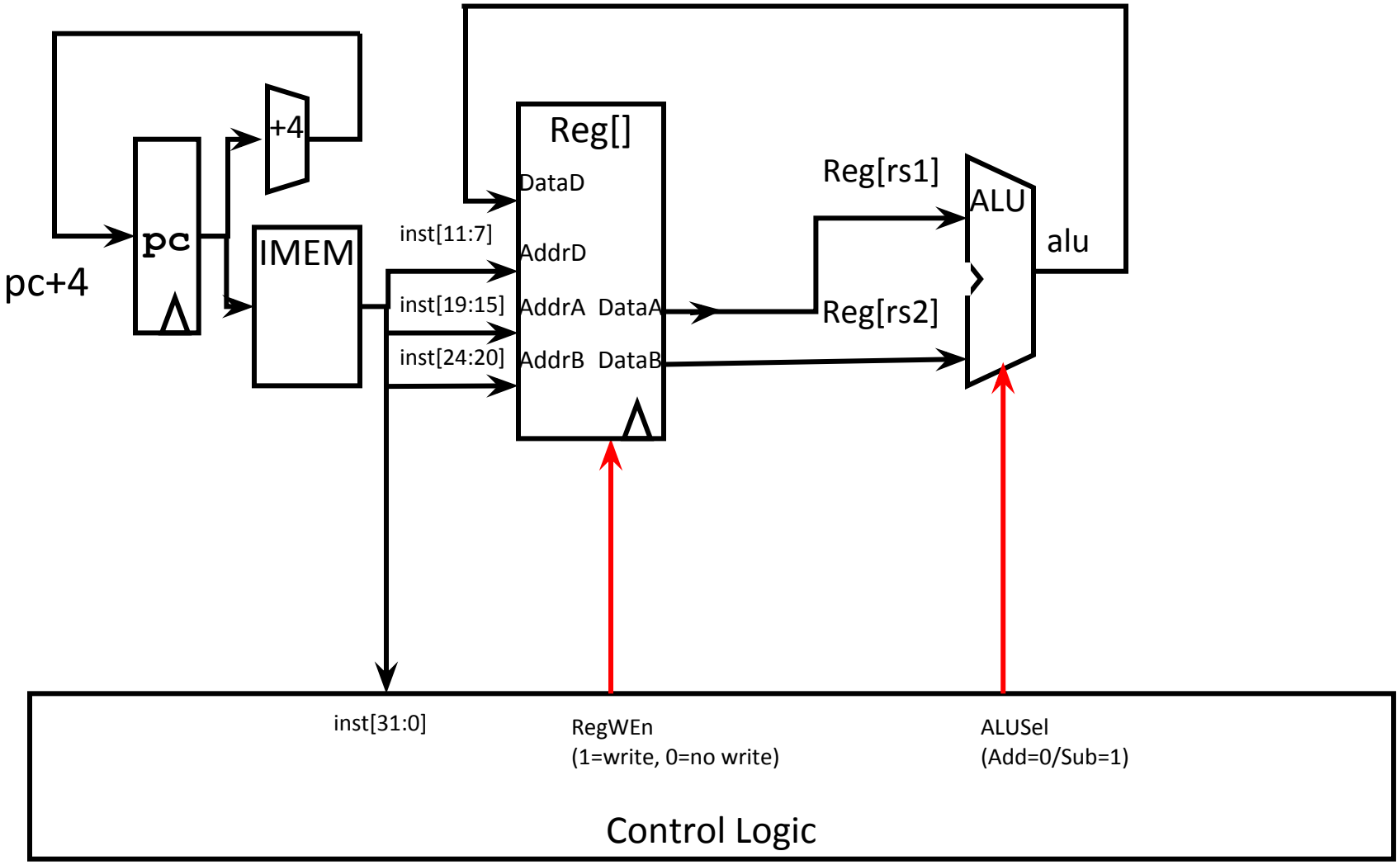
- RISC-V Assembly Instruction:

```
addi   x15,x1,-50
```

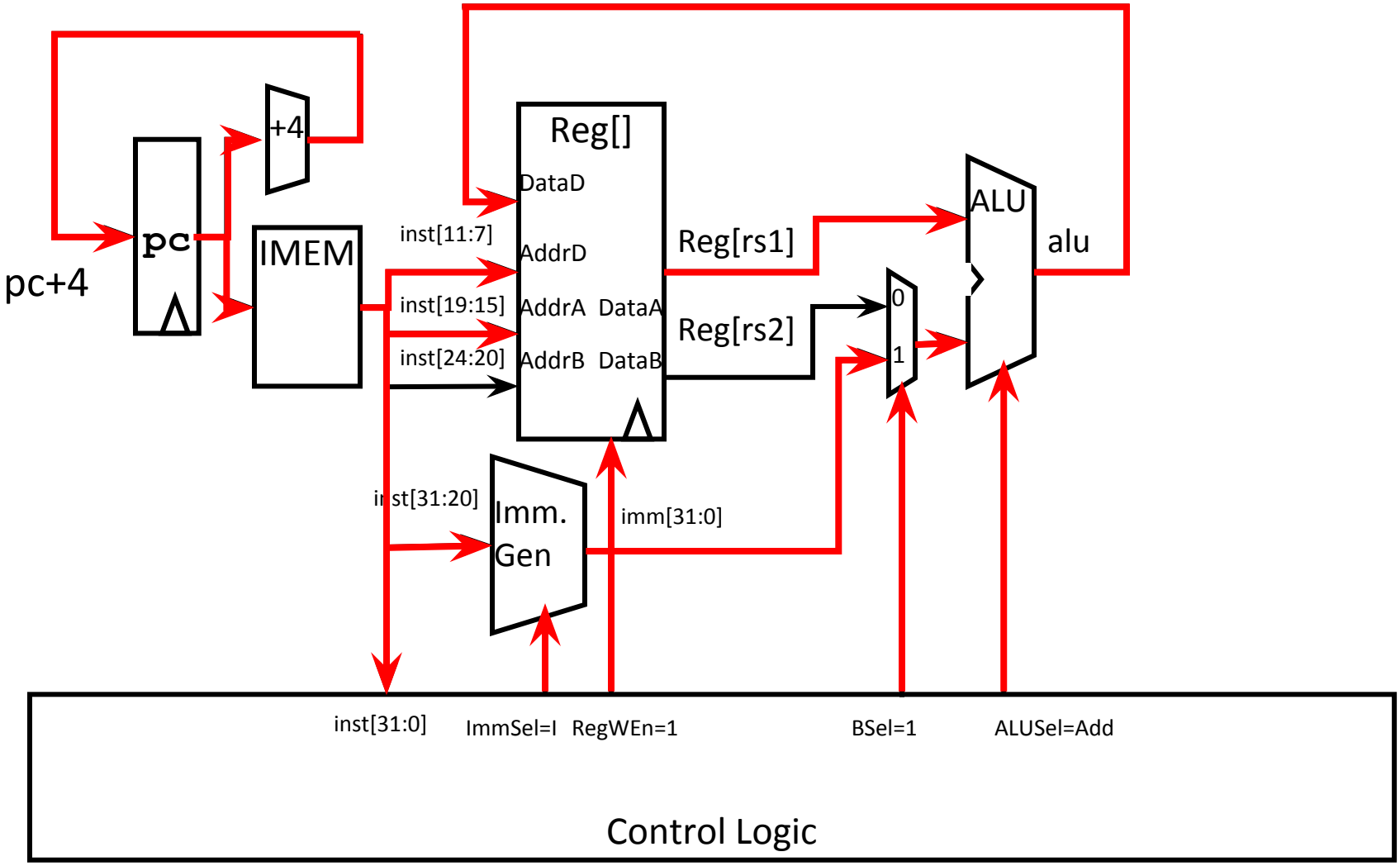



| 111111001110 | 00001 | 000 | 01111 | 0010011 |
|---|---|---|---|---|
| imm=-50 | rs1=1 | ADD | rd=15 | OP-Imm |

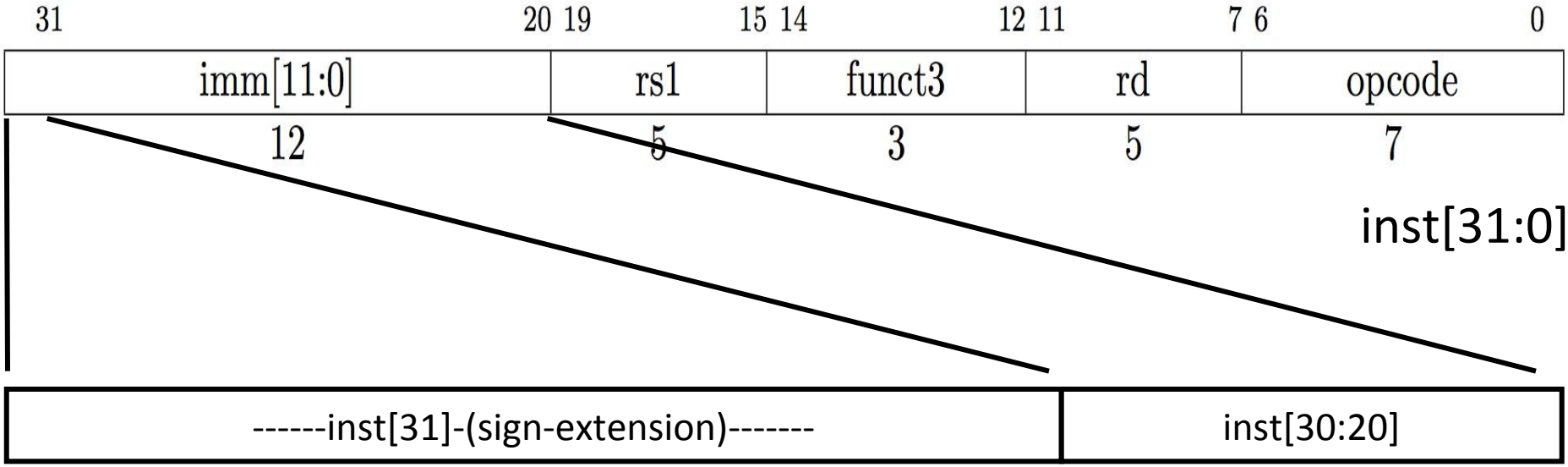
# Datapath for **add/sub**

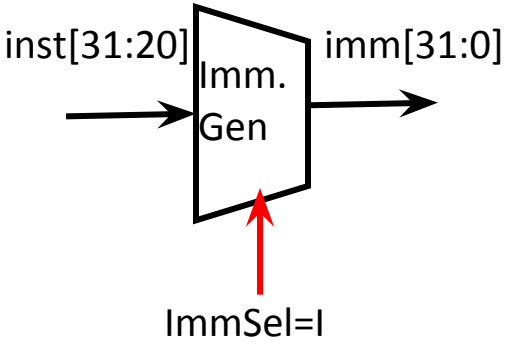


# Adding `addi` to datapath



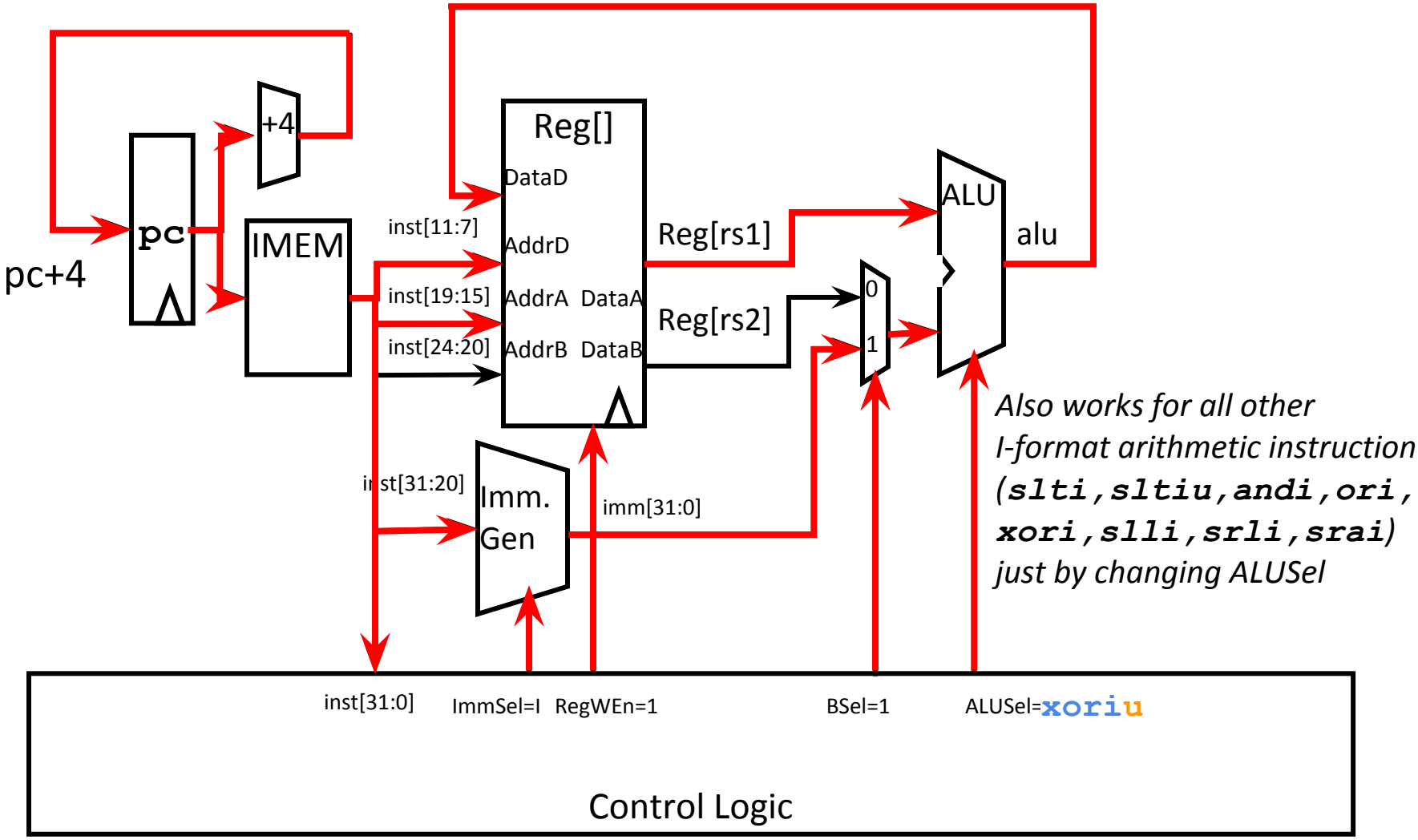
# I-Format immediates




- High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
- Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])

# Adding `addi` to datapath



# Agenda

- Datapath Overview
- Assembling the Datapath Part 1
- Administrivia
- Processor Design Process
- Assembling the Datapath Part 2

# Administrivia

- Project 2-2 due Friday (7/13)
  - Project Party tonight 4-6p in Soda 405 and 411!
- Homework 4 released; 3/4 due Monday (7/16)
- Guerilla Session on Wednesday, 4-6p, Cory 540AB
  - SDS, FSM, and Single-Cycle Datapath
- Project 3 will be released Thursday
  - Project party on 7/13, 4-6p (intended for students to get help starting on proj3/finishing lab 6)
  - Project party 7/20, 4-6p to finish up project 3
- Homework 2 grades are on glookup
  - Hw0/Hw1 not yet updated, but if you still didn't get credit for hw2, please change your emails to match!

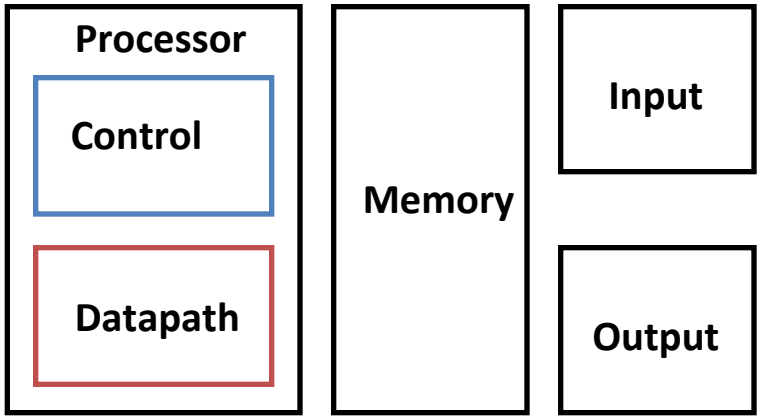
# Agenda

- Datapath Overview
- Assembling the Datapath Part 1
- Administrivia
- Processor Design Process
- Assembling the Datapath Part 2

# Processor Design Process

- Five steps to design a processor:
  1. Analyze instruction set → datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath components to meet the requirements
  4. Analyze implementation of each instruction to determine setting of control points that affect the register transfer
  5. Assemble the control logic
     - Formulate Logic Equations
     - Design Circuits

Now

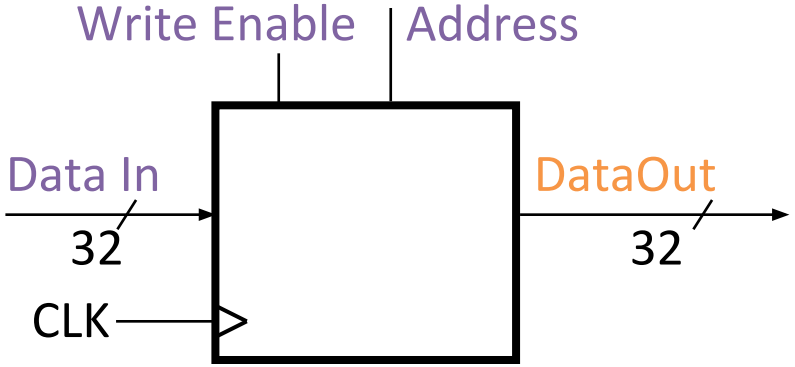


# Step 1: Requirements of the Instruction Set

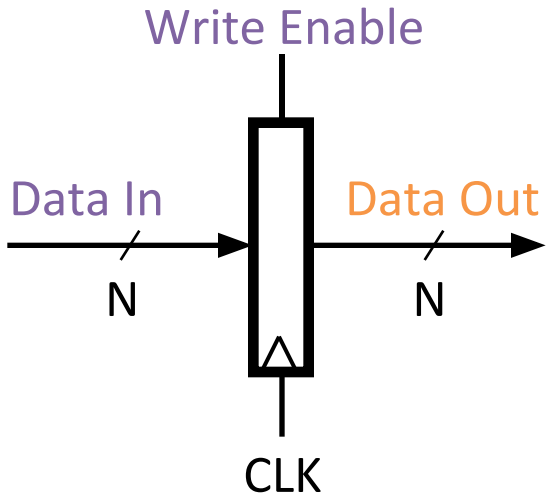
- Memory (MEM)
  - Instructions & data (separate: in reality just caches)
  - Load from and store to
- Registers (32 32-bit regs)
  - Read *rs1* and *rs2*
  - Write *rd*
- PC
  - Add 4 (+ maybe extended immediate)
- Add/Sub/OR unit for operation on register(s) or extended immediate
  - Compare if registers equal?

# Storage Element: Idealized Memory

- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- Memory access:
  - Read: Write Enable = 0, data at Address is placed on Data Out
  - Write: Write Enable = 1, Data In written to Address
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read, behaves as a combinational logic block: Address valid → Data Out valid after "access time"

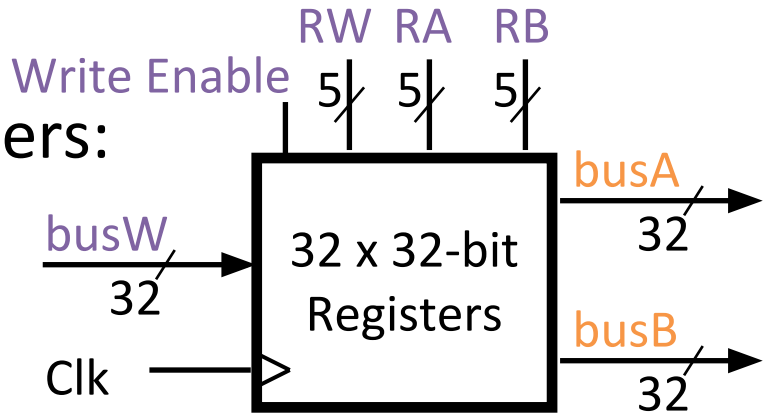


# Storage Element: Register




- Similar to D flip-flop except:
  - N-bit input and output buses
  - Write Enable input
- Write Enable:
  - De-asserted (0): Data Out will not change
  - Asserted (1): Data In value placed onto Data Out after CLK trigger

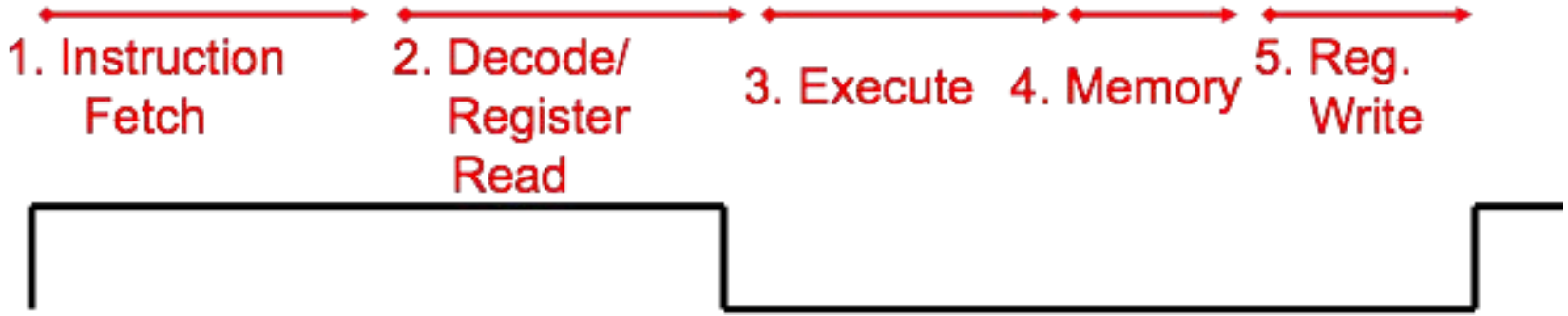
# Storage Element: Register File




- *Register File* consists of 32 registers:
  - Output buses busA and busB
  - Input bus busW
- Register selection
  - Place data of register RA (number) onto busA
  - Place data of register RB (number) onto busB
  - Store data on busW into register RW (number) when Write Enable is 1
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read, behaves as a combinational logic block: RA or RB valid → busA or busB valid after "access time"

# Step 2: CPU Clocking

- For each instruction, how do we control the flow of information through the datapath?
- Single Cycle CPU: All stages of an instruction completed within one long clock cycle
  - Clock cycle sufficiently long to allow each instruction to complete all stages without interruption within one cycle



# Step 3: Assembling the Datapath

- Assemble datapath to meet ISA requirements
  - Exact requirements will change based on ISA
  - Here we must examine *each instruction* of RISC
- The datapath is all of the hardware components and wiring necessary to carry out ALL of the different instructions
  - Make sure all components (e.g. RegFile, ALU) have access to all necessary signals and buses
  - Control will make sure instructions are properly executed (the decision making)

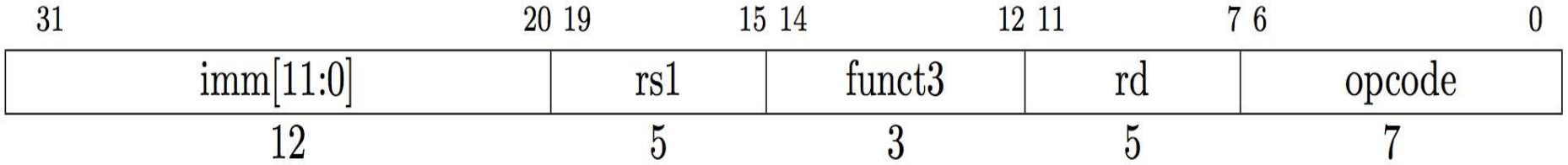
# Agenda

- Datapath Overview
- Assembling the Datapath Part 1
- Administrivia
- Processor Design Process
- Assembling the Datapath Part 2

# Implementing Load Word instruction

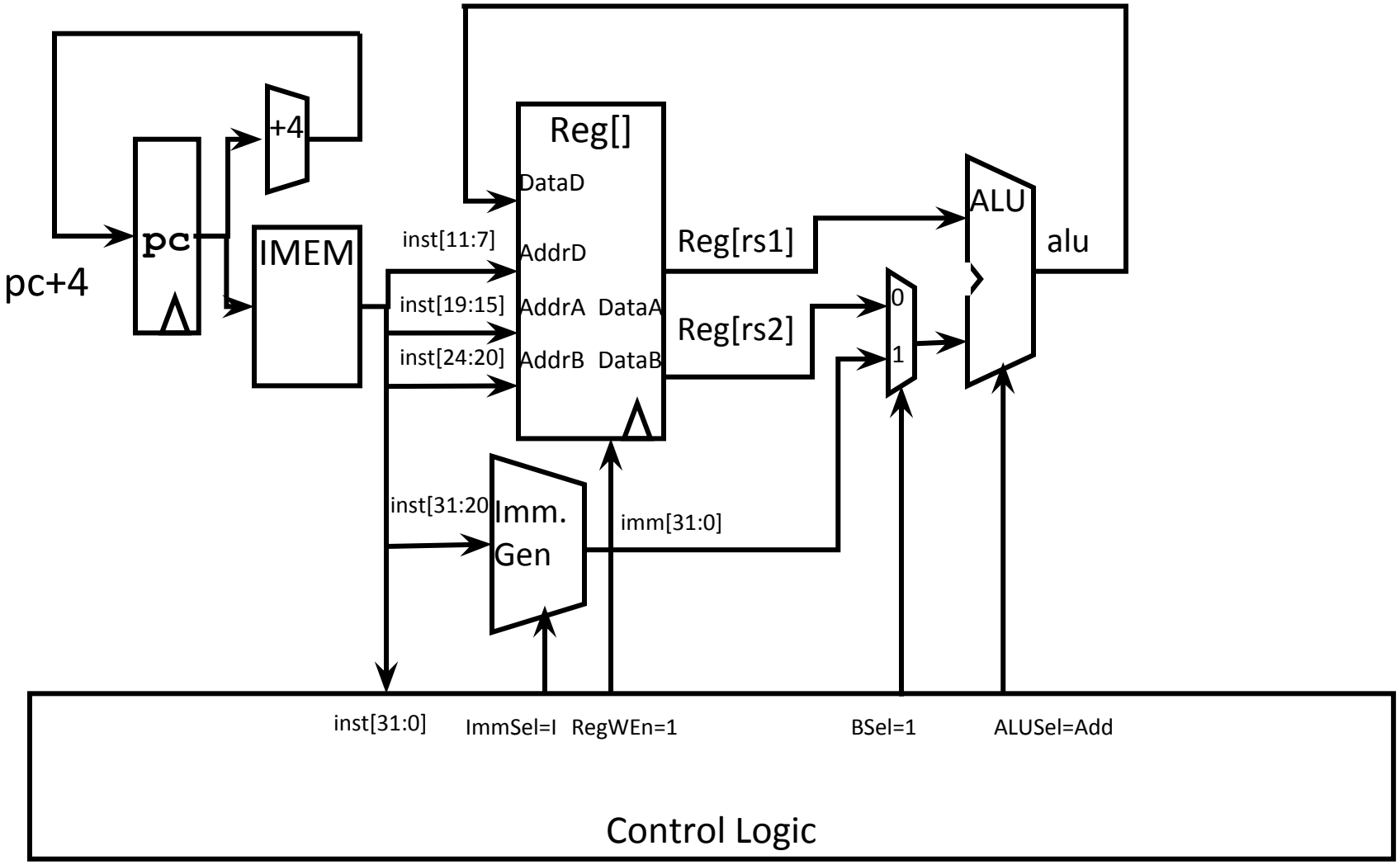
- RISC-V Assembly Instruction:

```
lw x14, 8(x2)
```

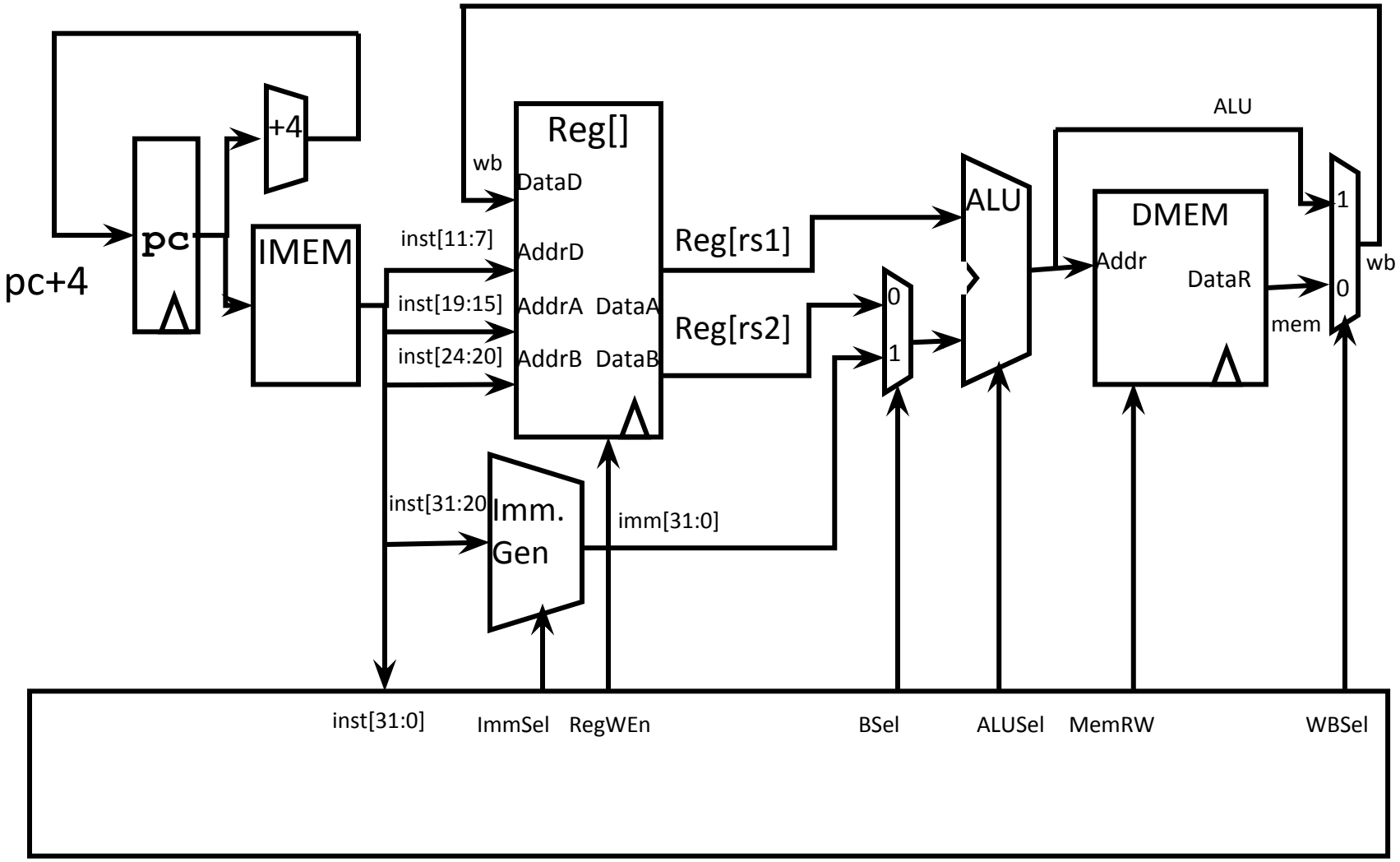



| 000000001000 | 00010 | 010 | 01110 | 0000011 |
|---|---|---|---|---|
| imm=+8 | rs1=2 | LW | rd=14 | LOAD |

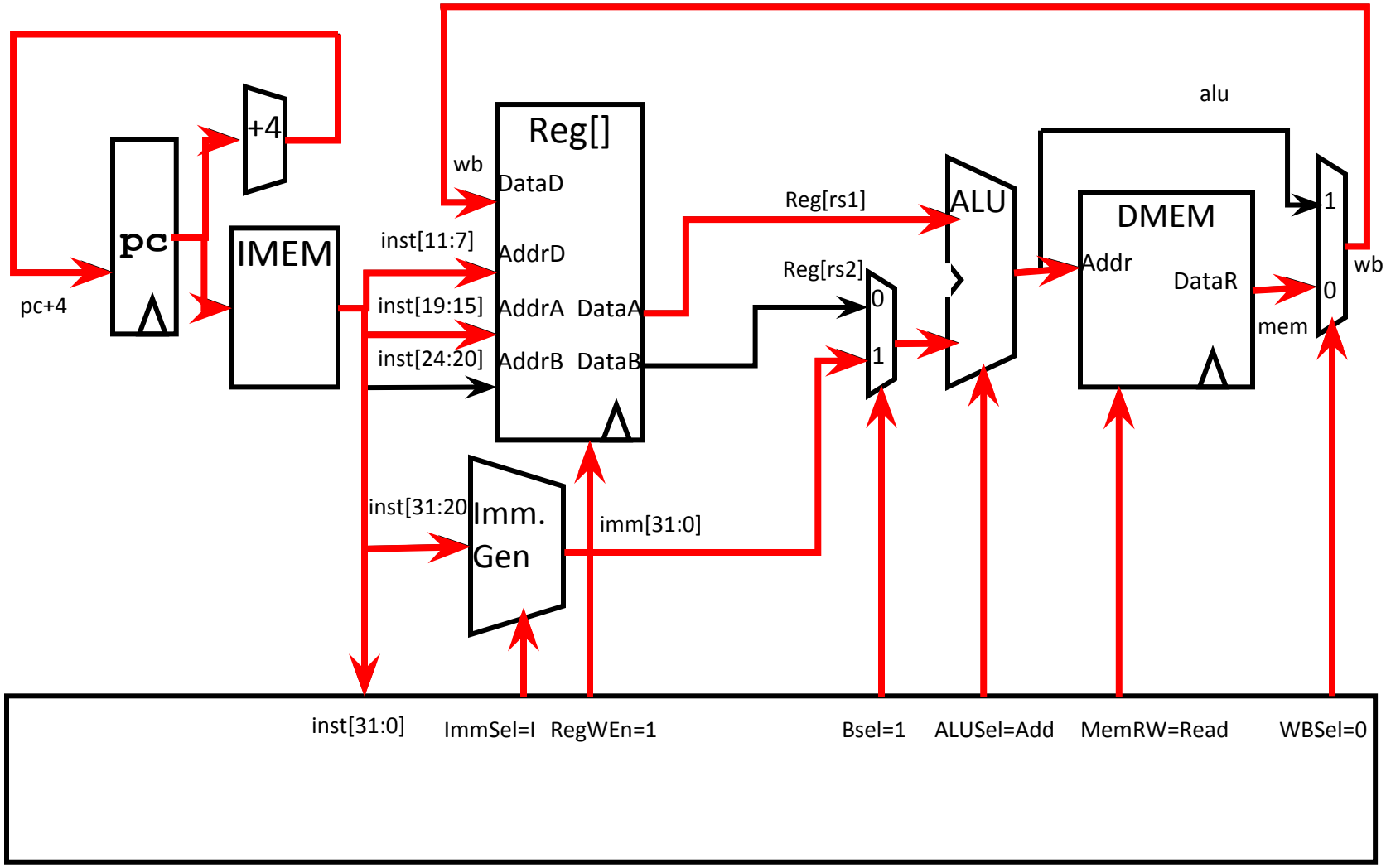
# Adding `addi` to datapath



# Adding `lw` to datapath



# Adding `lw` to datapath



# All RV32 Load Instructions

| | | | | | |
|---|---|---|---|---|---|
| imm[11:0] | rs1 | 000 | rd | 0000011 | LB |
| imm[11:0] | rs1 | 001 | rd | 0000011 | LH |
| imm[11:0] | rs1 | 010 | rd | 0000011 | LW |
| imm[11:0] | rs1 | 100 | rd | 0000011 | LBU |
| imm[11:0] | rs1 | 101 | rd | 0000011 | LHU |

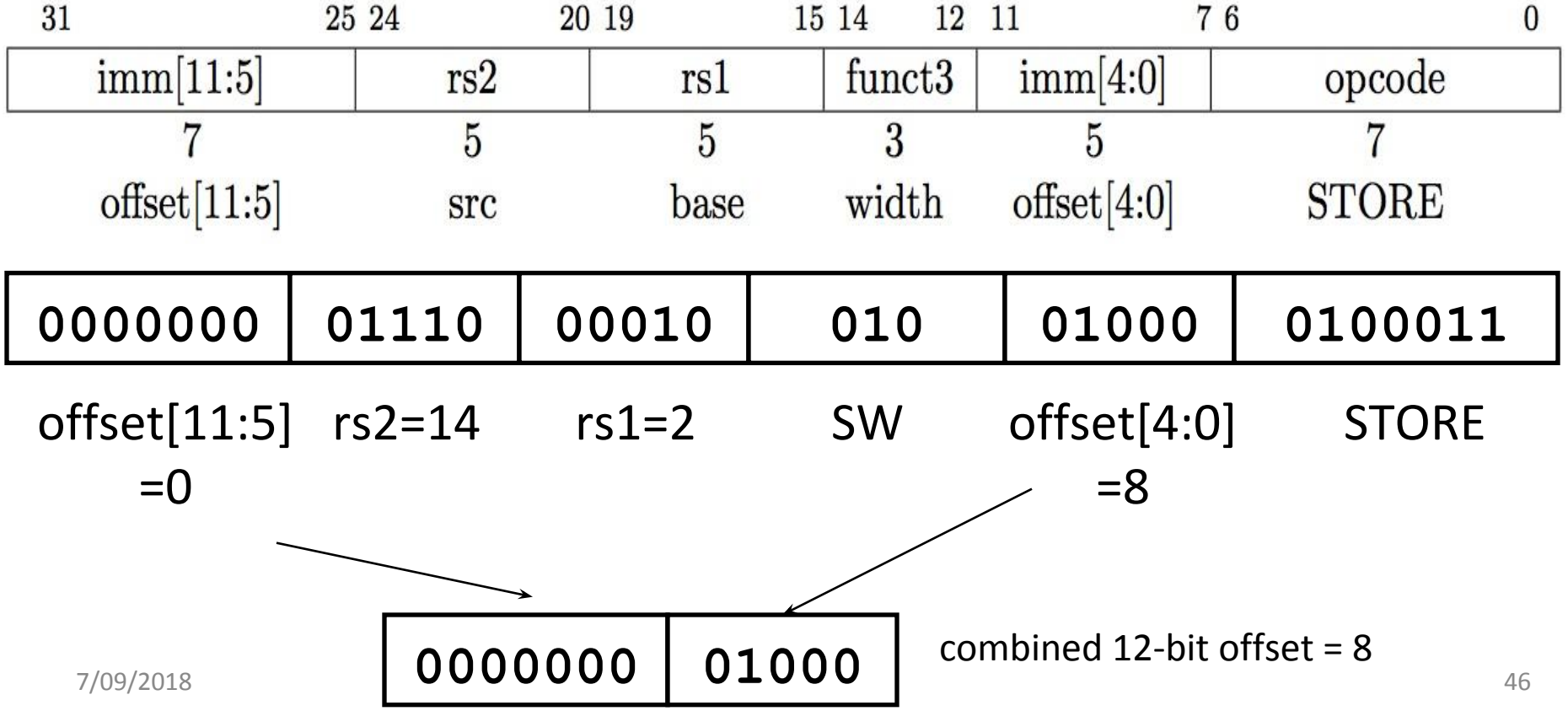
funct3 field encodes size and signedness of load data

- Supporting the narrower loads requires additional circuits to extract the correct byte/halfword from the value loaded from memory, and sign- or zero-extend the result to 32 bits before writing back to register file.

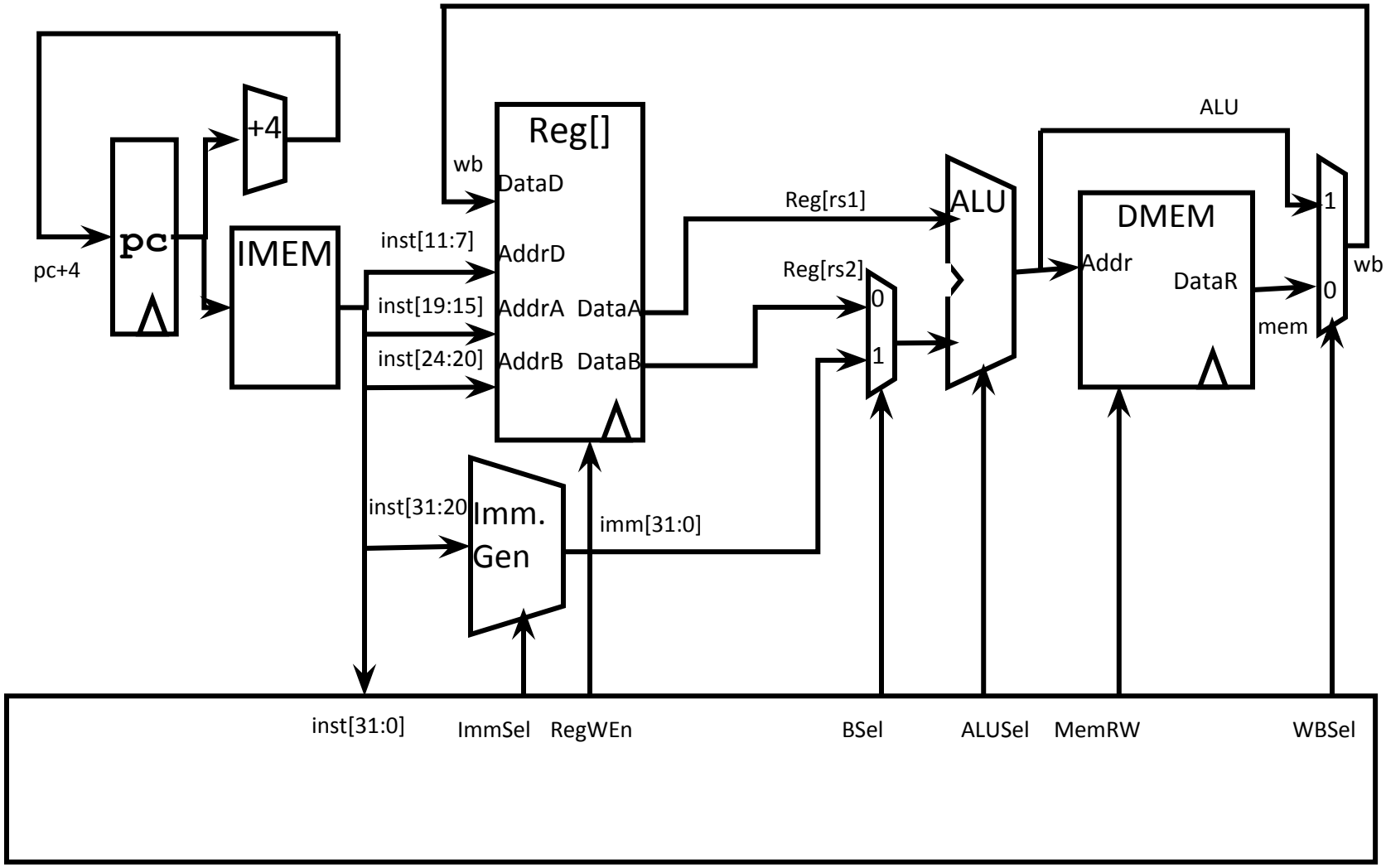
# Implementing Store Word instruction

- RISC-V Assembly Instruction:

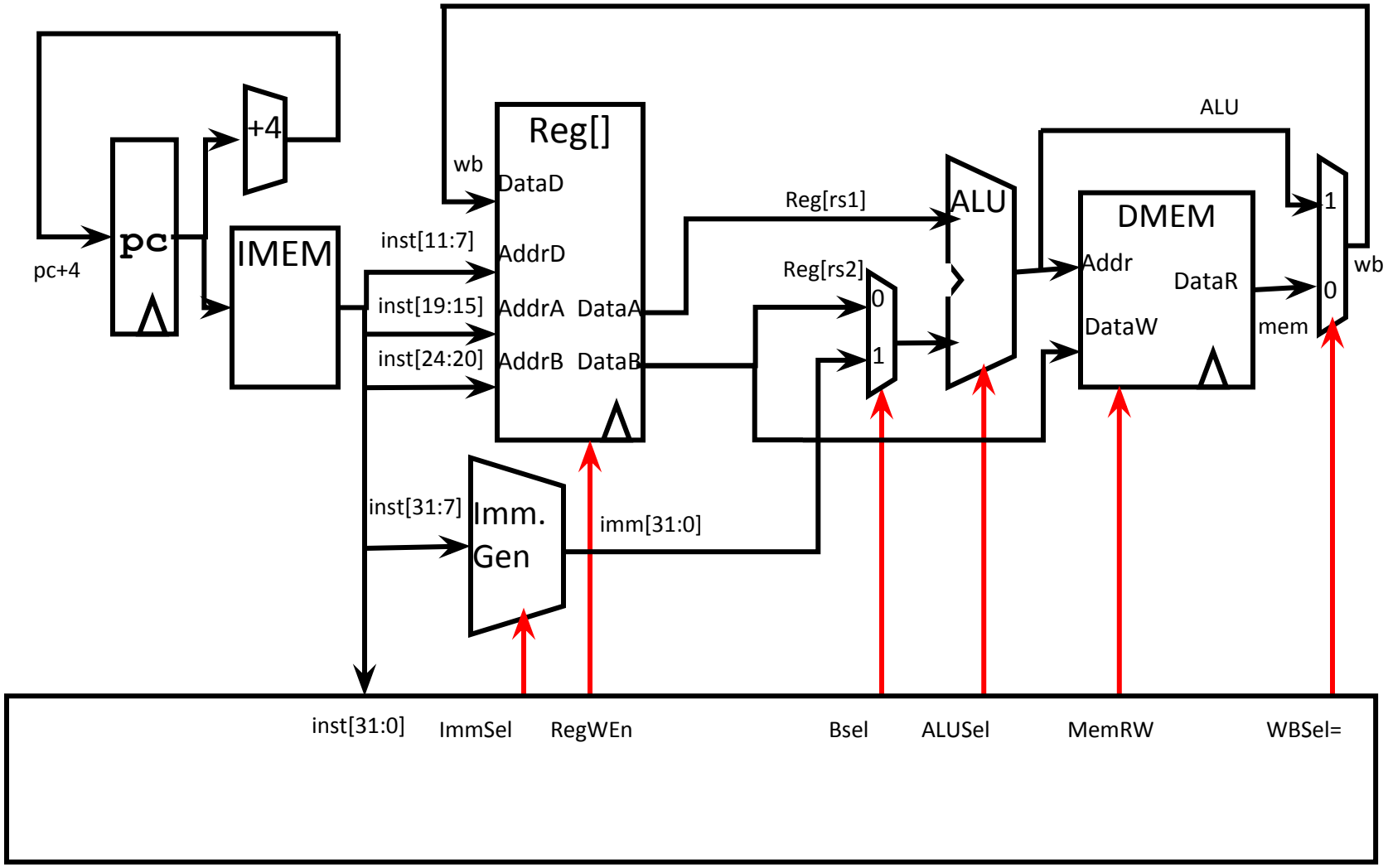
```
sw x14, 8(x2)
```



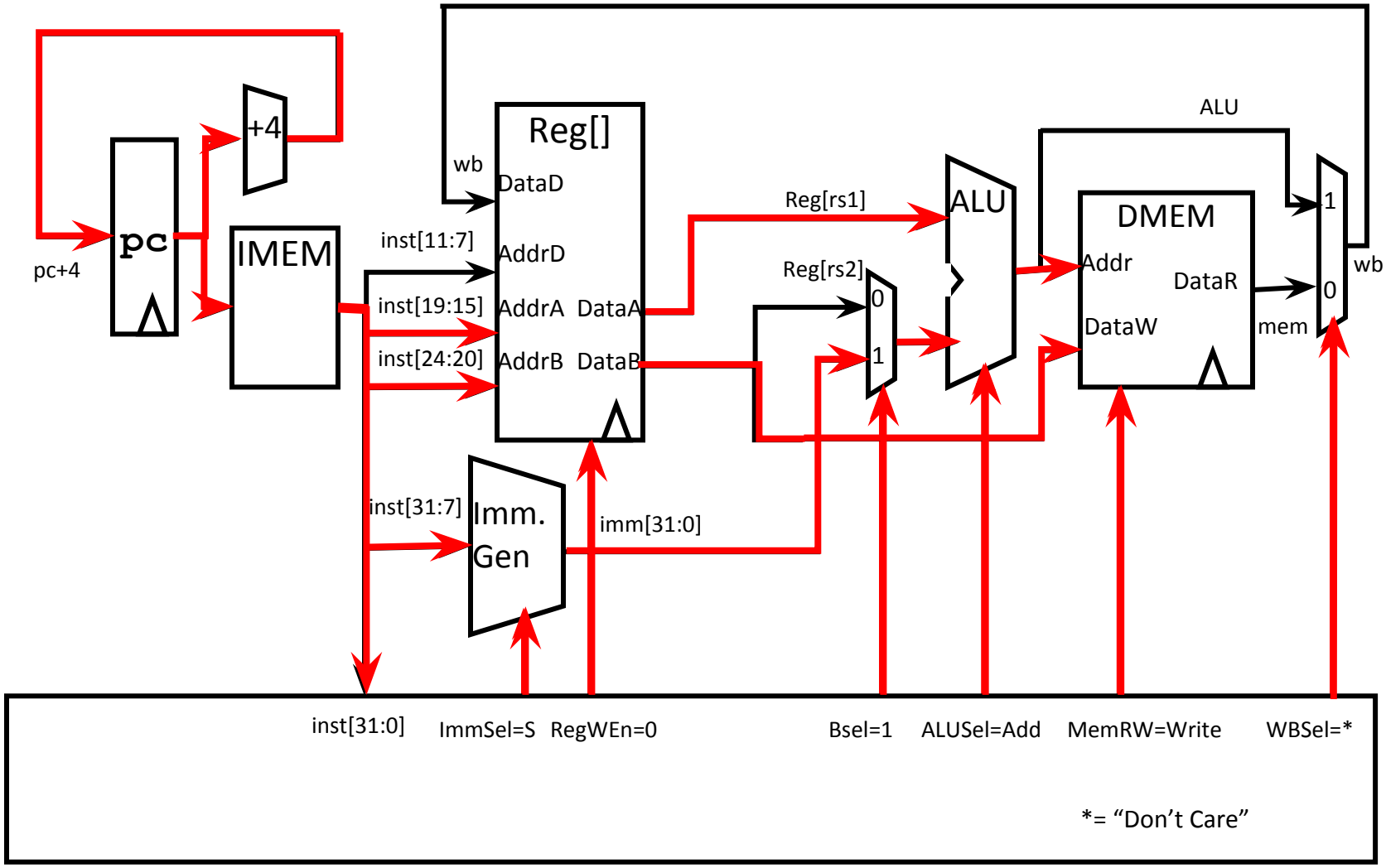
# Adding `lw` to datapath



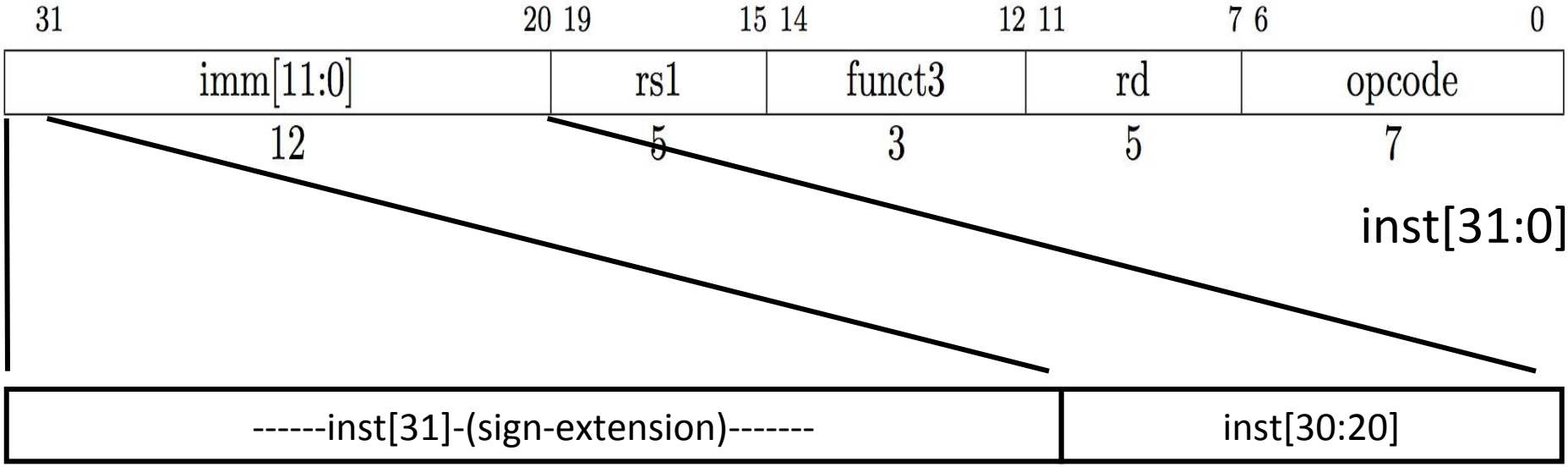
# Adding `sw` to datapath

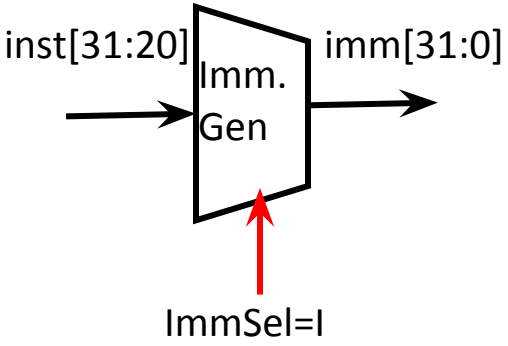


# Adding `sw` to datapath



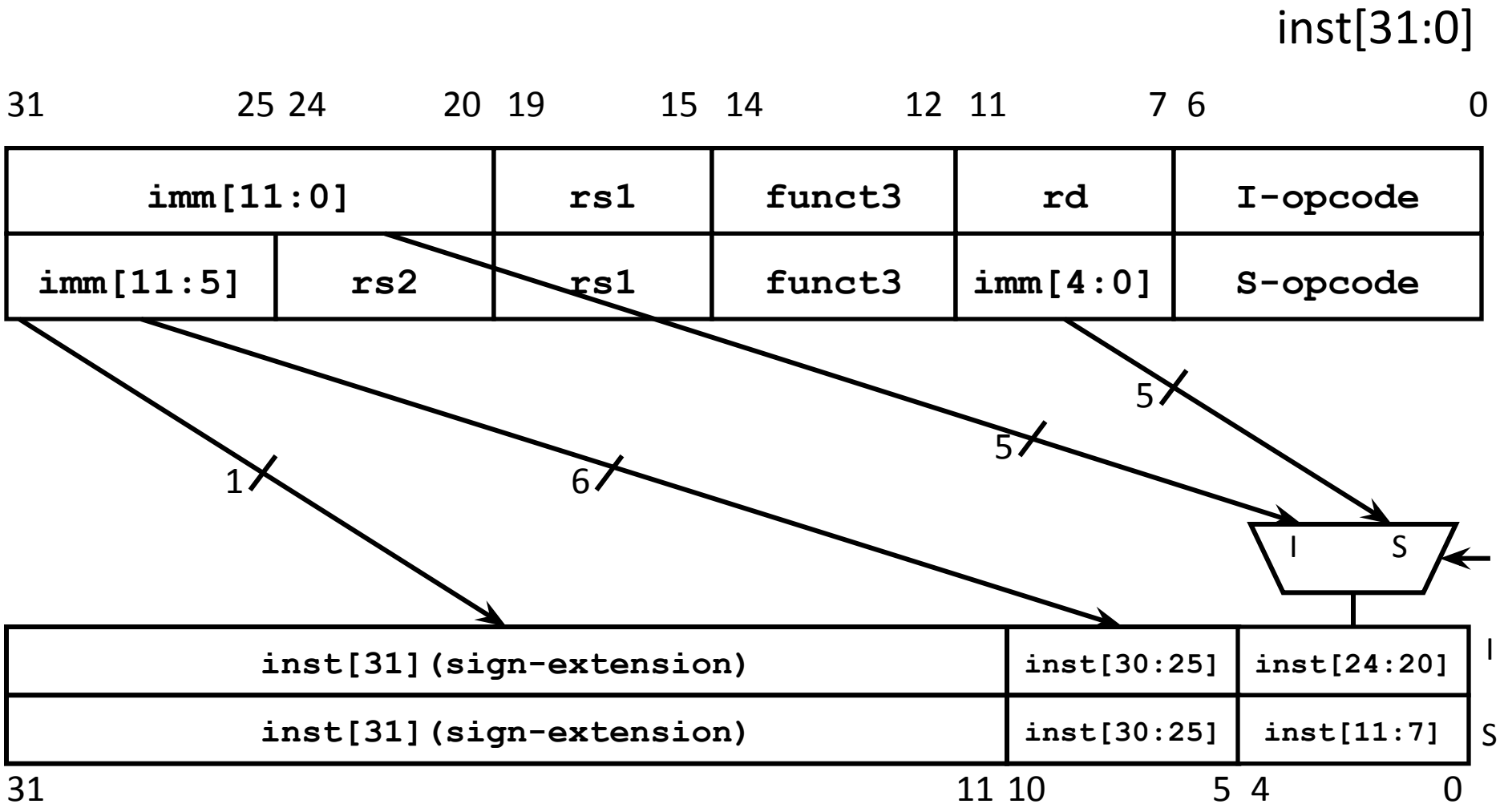
# I-Format immediates



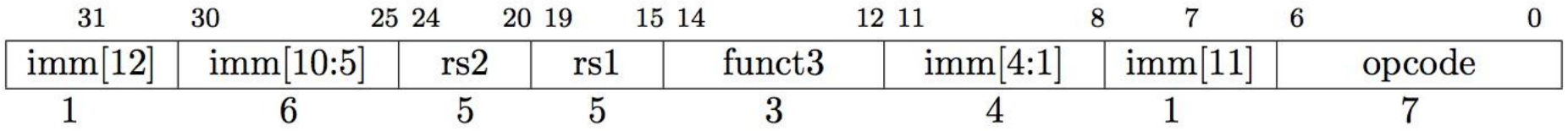

- High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
- Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])

# I & S Immediate Generator



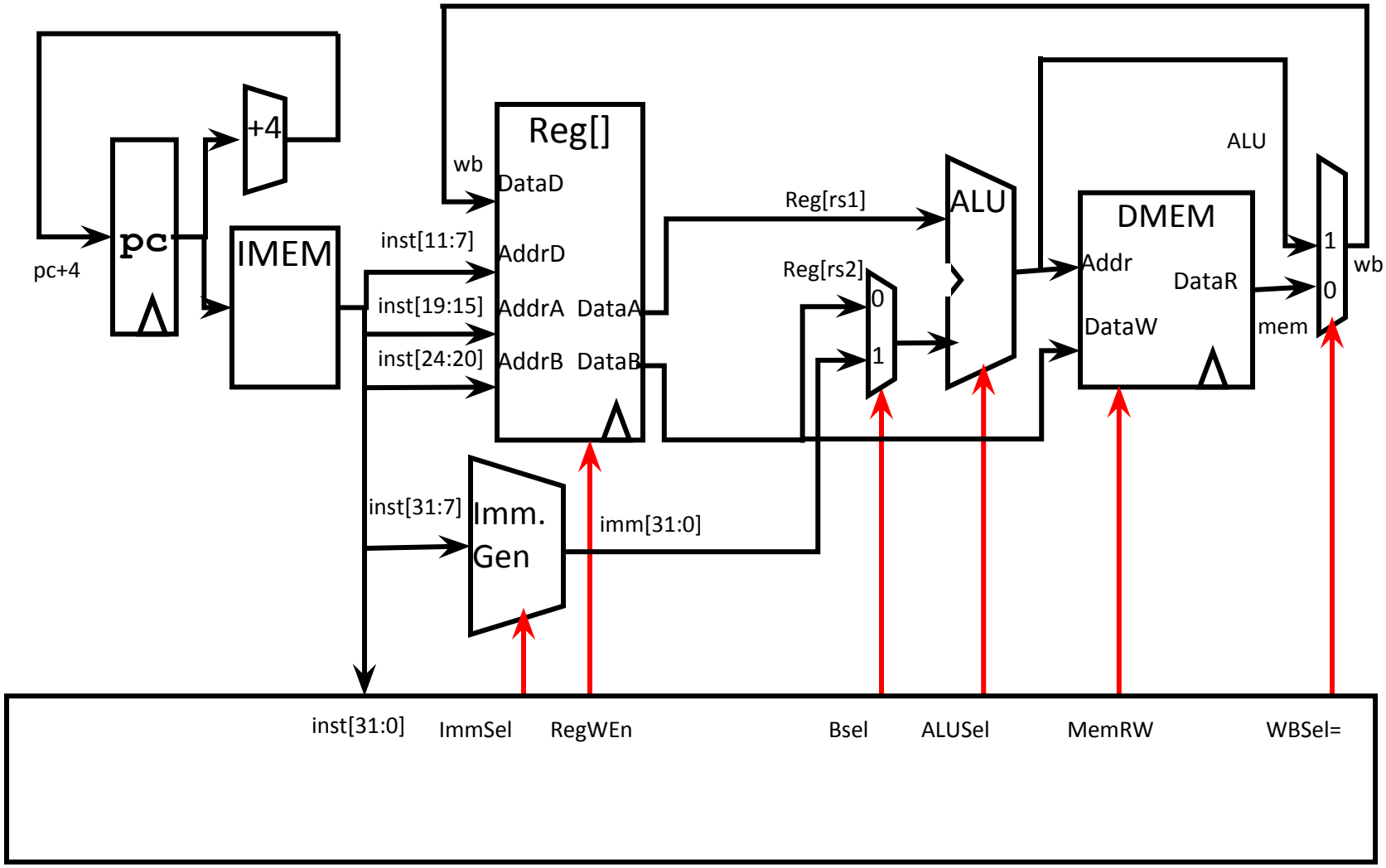

- Just need a 5-bit mux to select between two positions where low five bits of immediate can reside in instruction
- Other bits in immediate are wired to fixed positions in instruction

# Implementing Branches

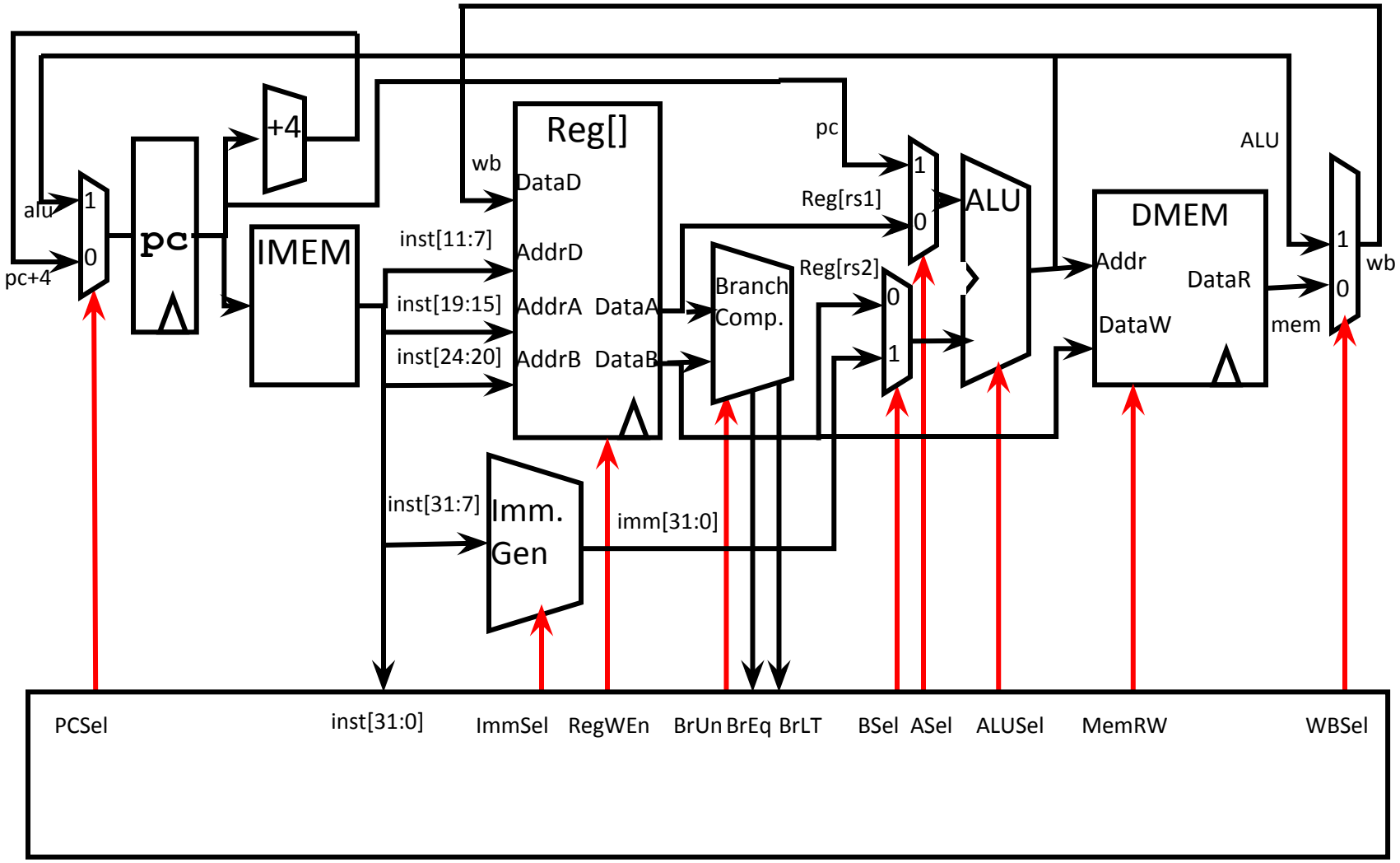



- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode *even* 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

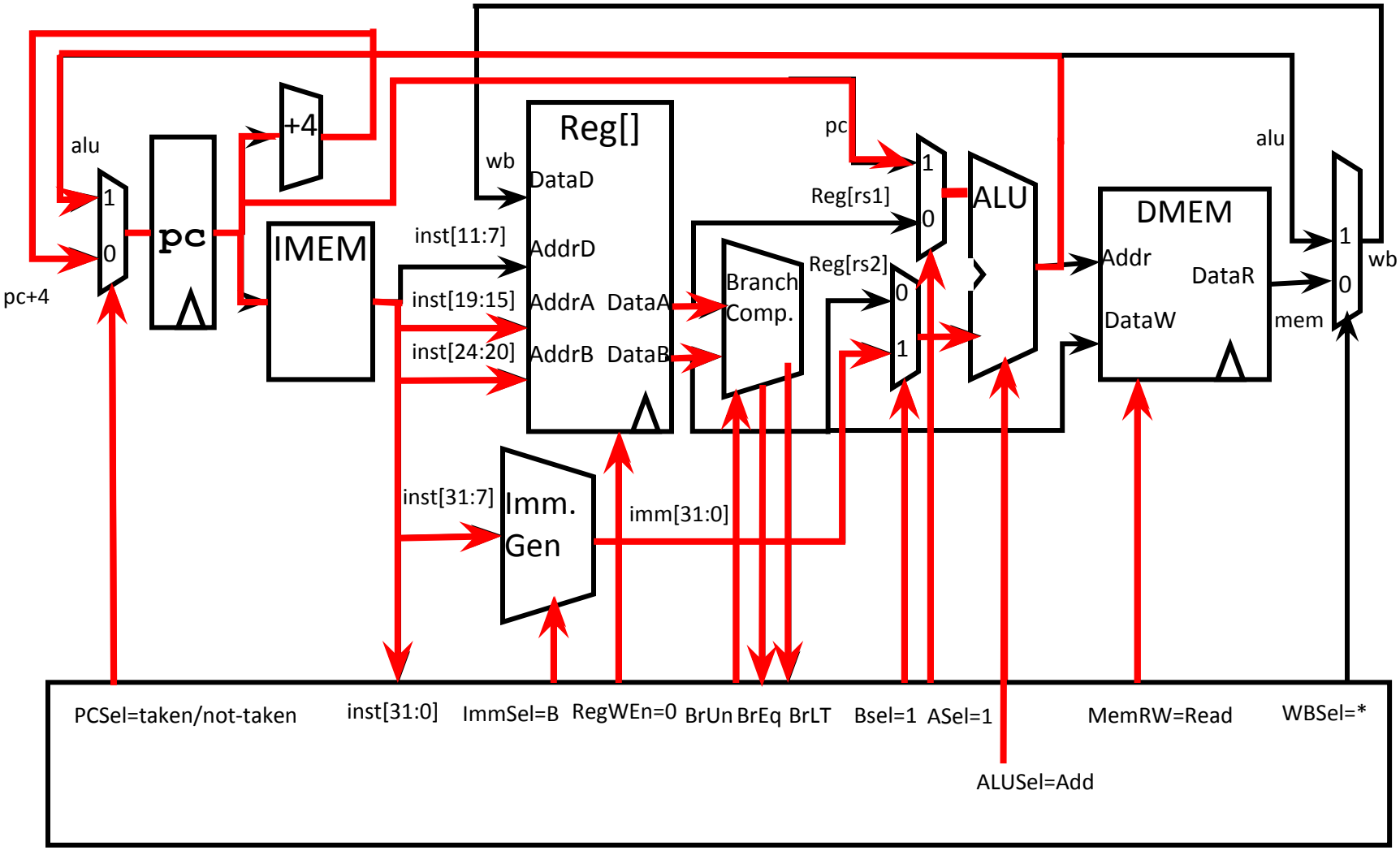
# Adding `sw` to datapath



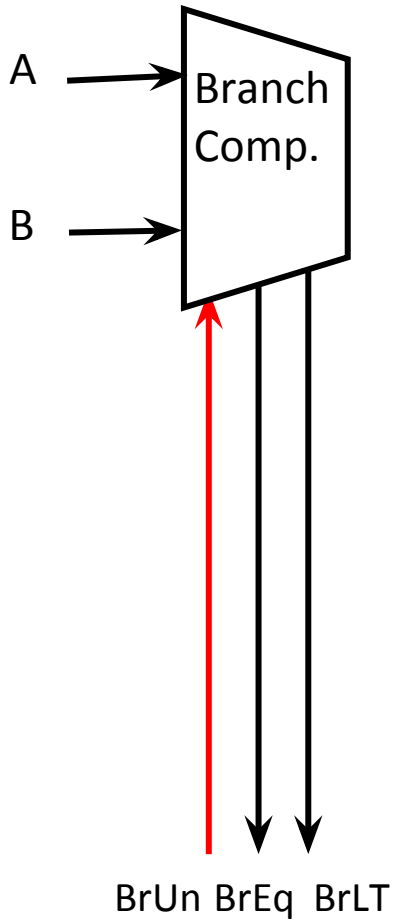
# Adding branches to datapath



# Adding branches to datapath



# Branch Comparator



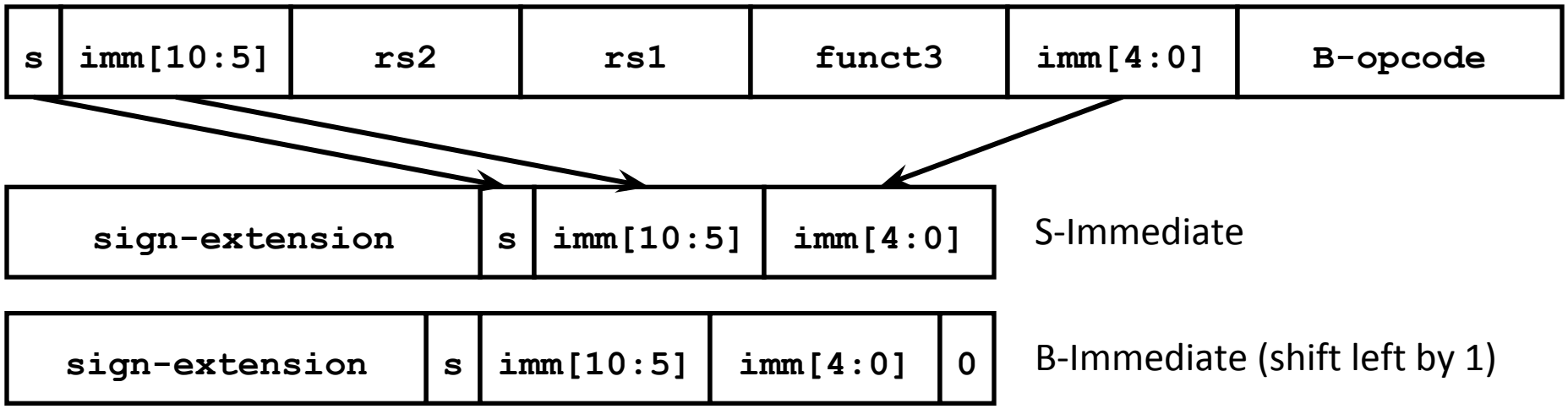

- BrEq = 1, if A=B
- BrLT = 1, if A < B
- BrUn =1 selects unsigned comparison for BrLT, 0=signed

- BGE branch: A >= B, if !(A<B)

# Break!

# Multiply Branch Immediates by Shift?

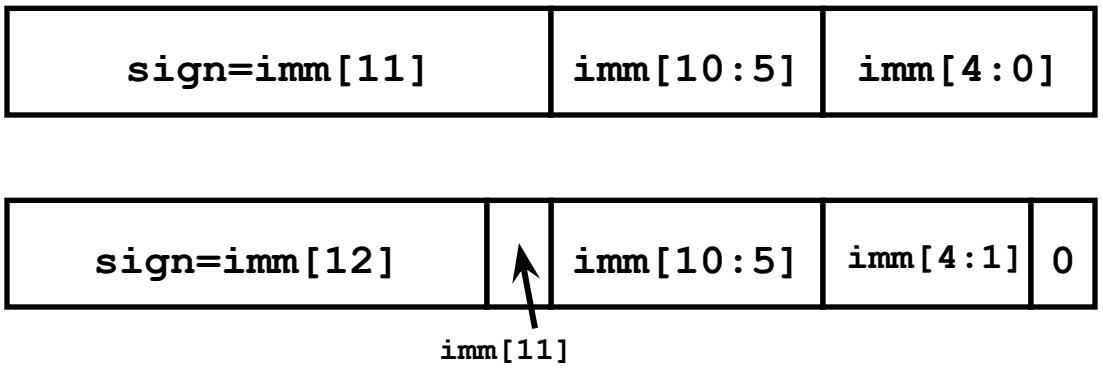
- 12-bit immediate encodes PC-relative offset of -4096 to +4094 bytes in multiples of 2 bytes
- Standard approach: treat immediate as in range -2048..+2047, then shift left by 1 bit to multiply by 2 for branches




Each instruction immediate bit can appear in one of two places in output immediate value – so need one 2-way mux per bit

# RISC-V Branch Immediates

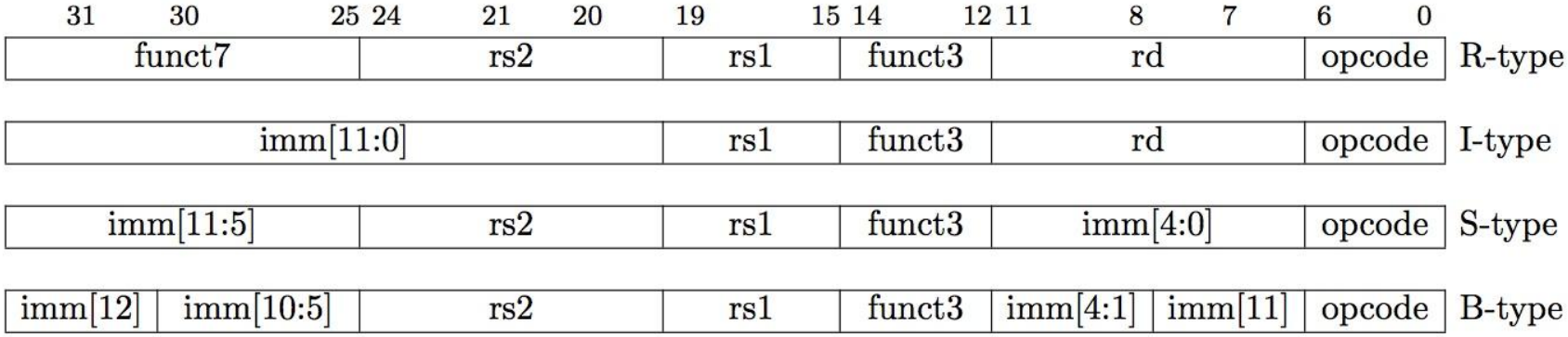
- 12-bit immediate encodes PC-relative offset of -4096 to +4094 bytes in multiples of 2 bytes
- RISC-V approach: keep 11 immediate bits in fixed position in output value, and rotate LSB of S-format to be bit 12 of B-format



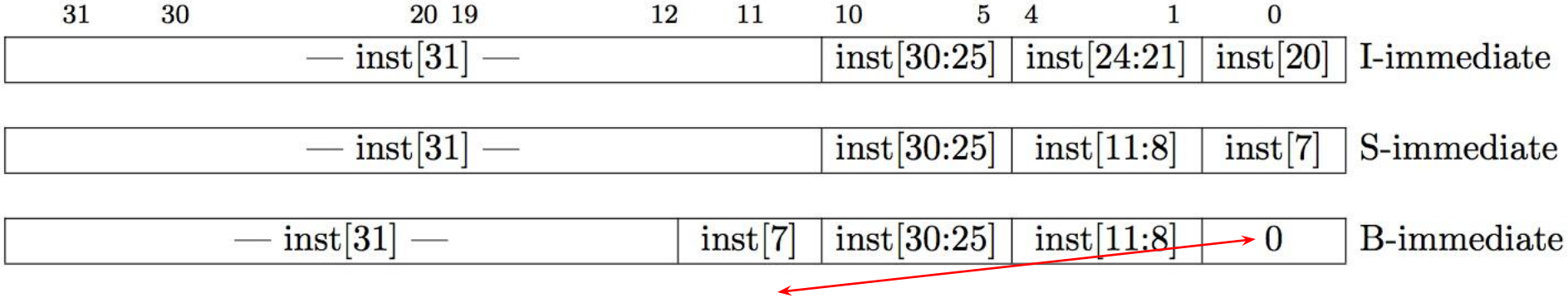

Only one bit changes position between S and B, so only need a single-bit 2-way mux

# RISC-V Immediate Encoding

Instruction Encodings, inst[31:0]



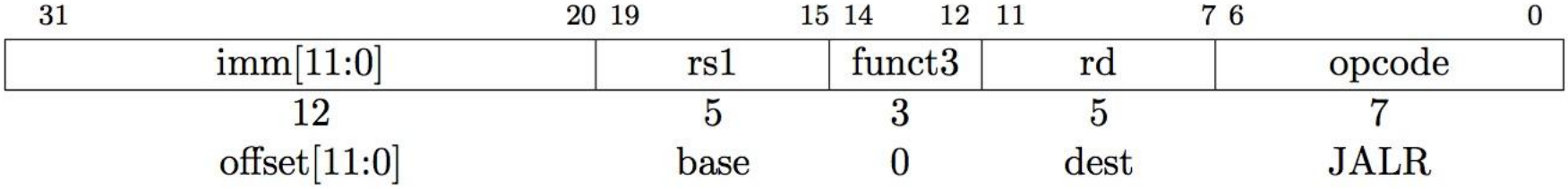

32-bit immediates produced, imm[31:0]




Upper bits sign-extended from inst[31] always

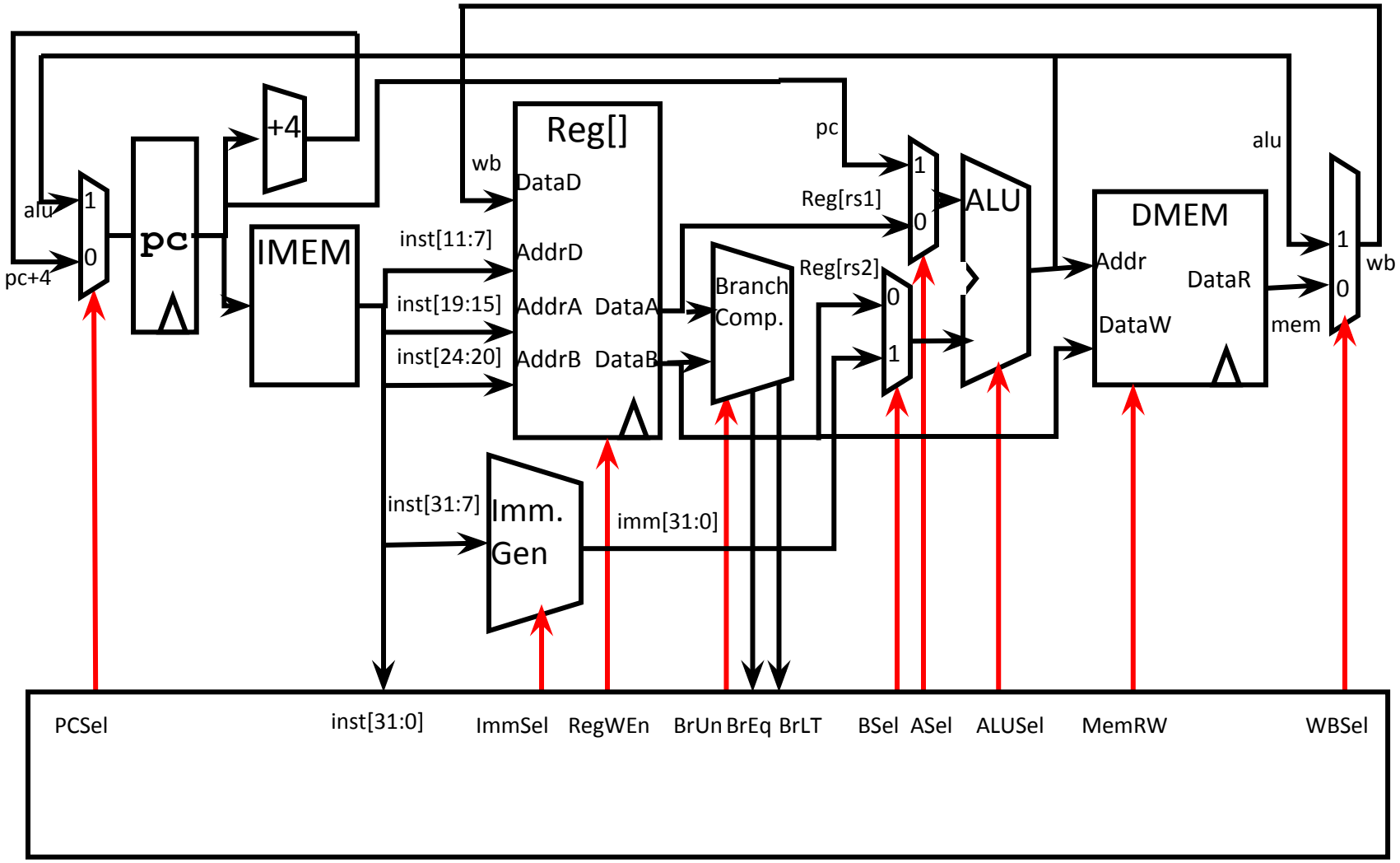
Only bit 7 of instruction changes role in immediate between S and B

# Implementing **JALR** Instruction (I-Format)

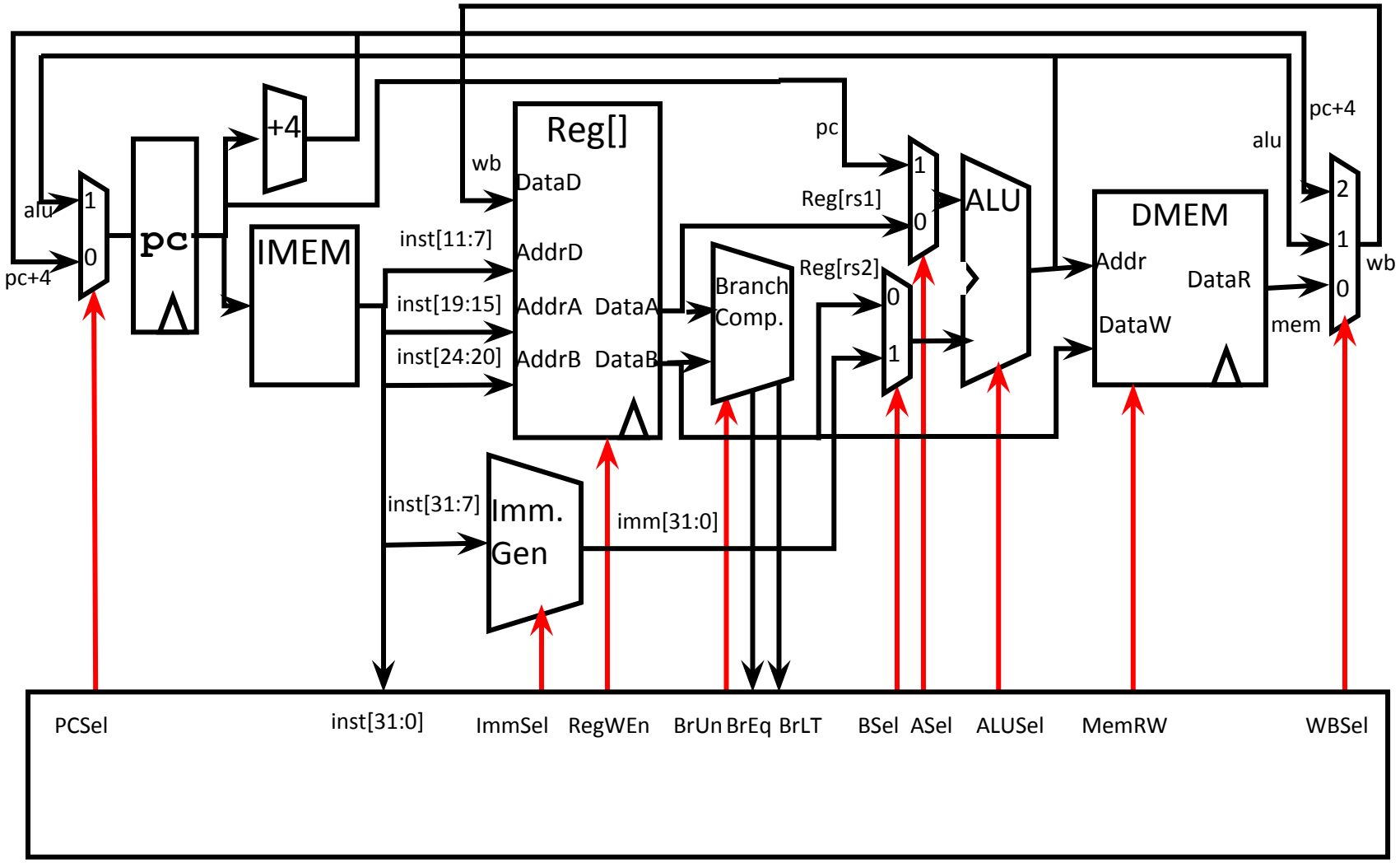



- JALR rd, rs, immediate
  - Writes PC+4 to Reg[rd] (return address)
  - Sets PC = Reg[rs1] + immediate
  - Uses same immediates as arithmetic and loads
    - ***no*** multiplication by 2 bytes

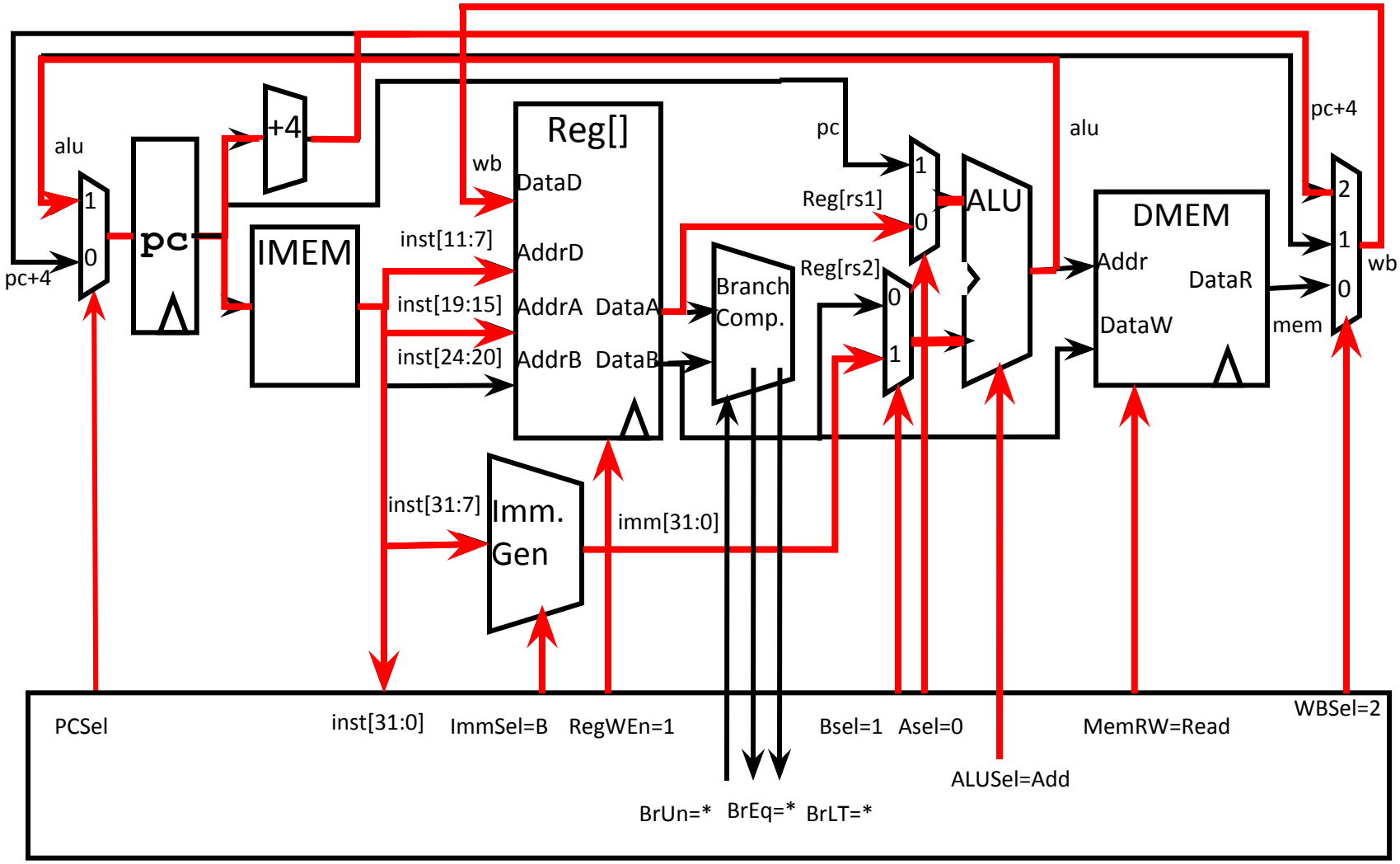
# Adding branches to datapath



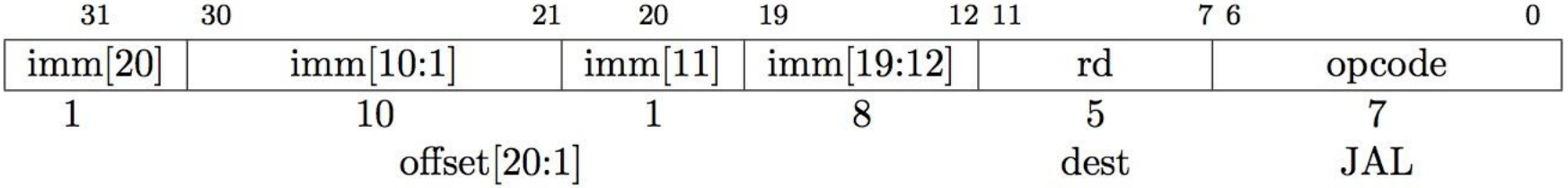
# Adding `jalr` to datapath



# Adding `jalr` to datapath

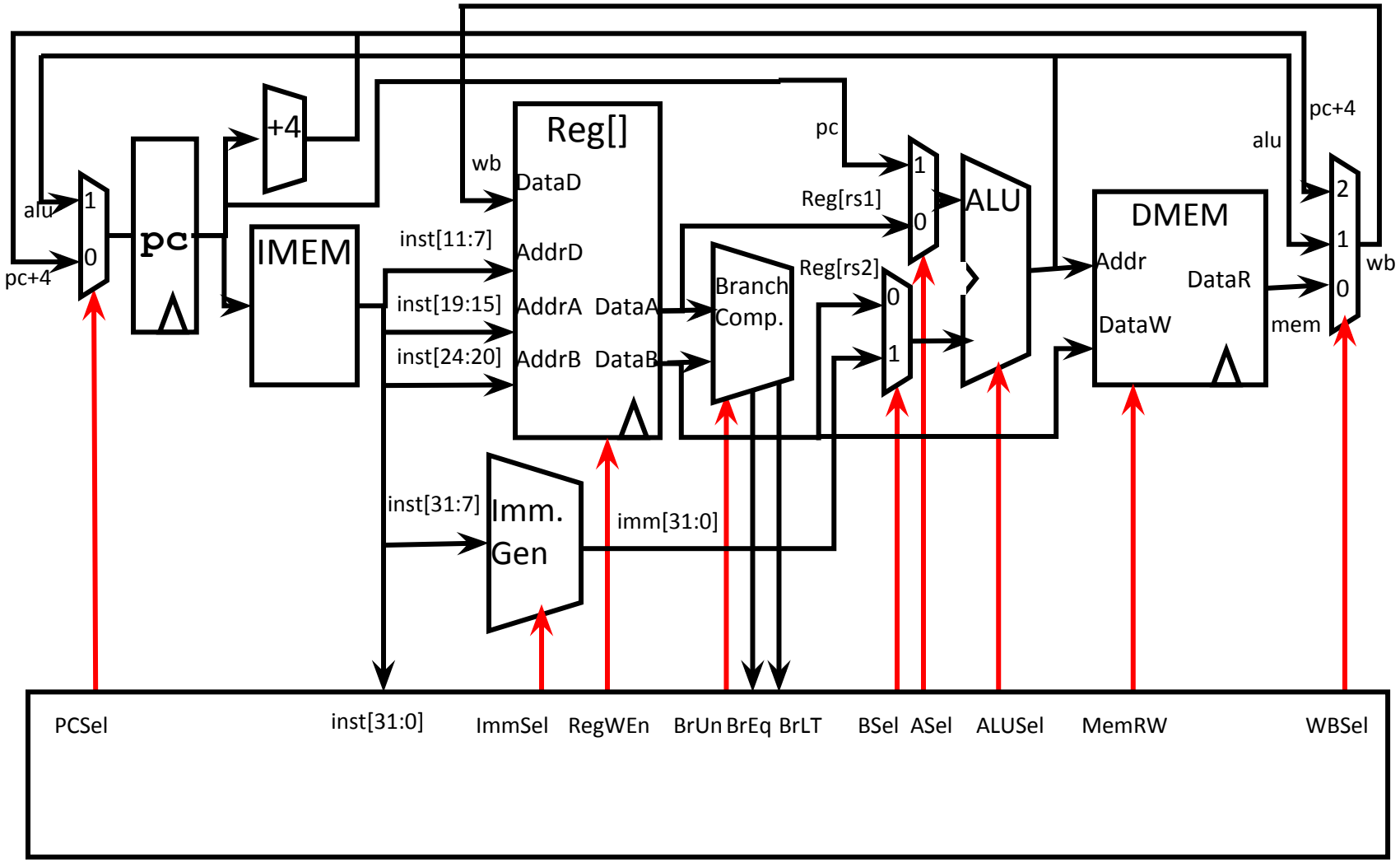


# Implementing `jal` Instruction

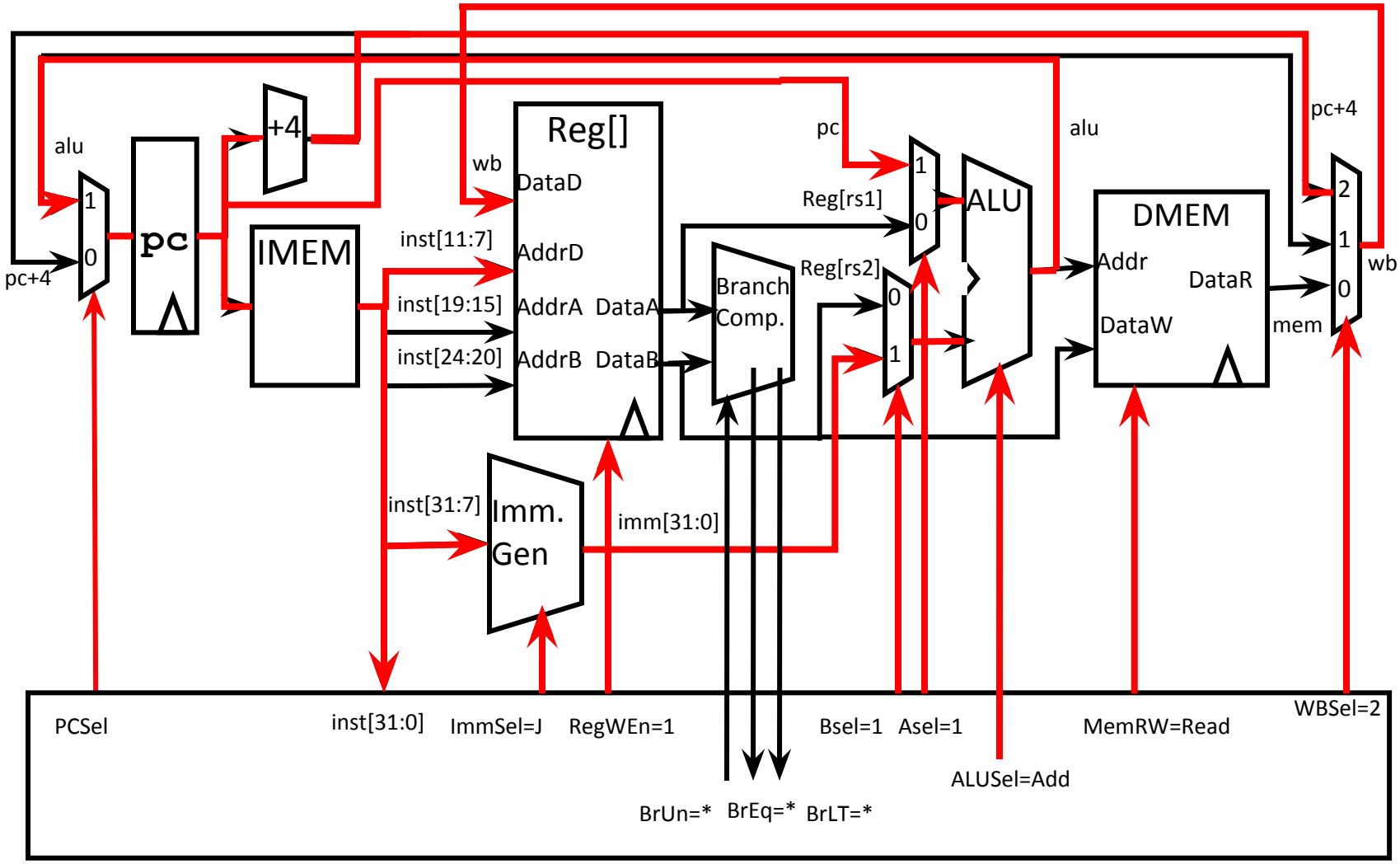



- JAL saves PC+4 in Reg[rd] (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

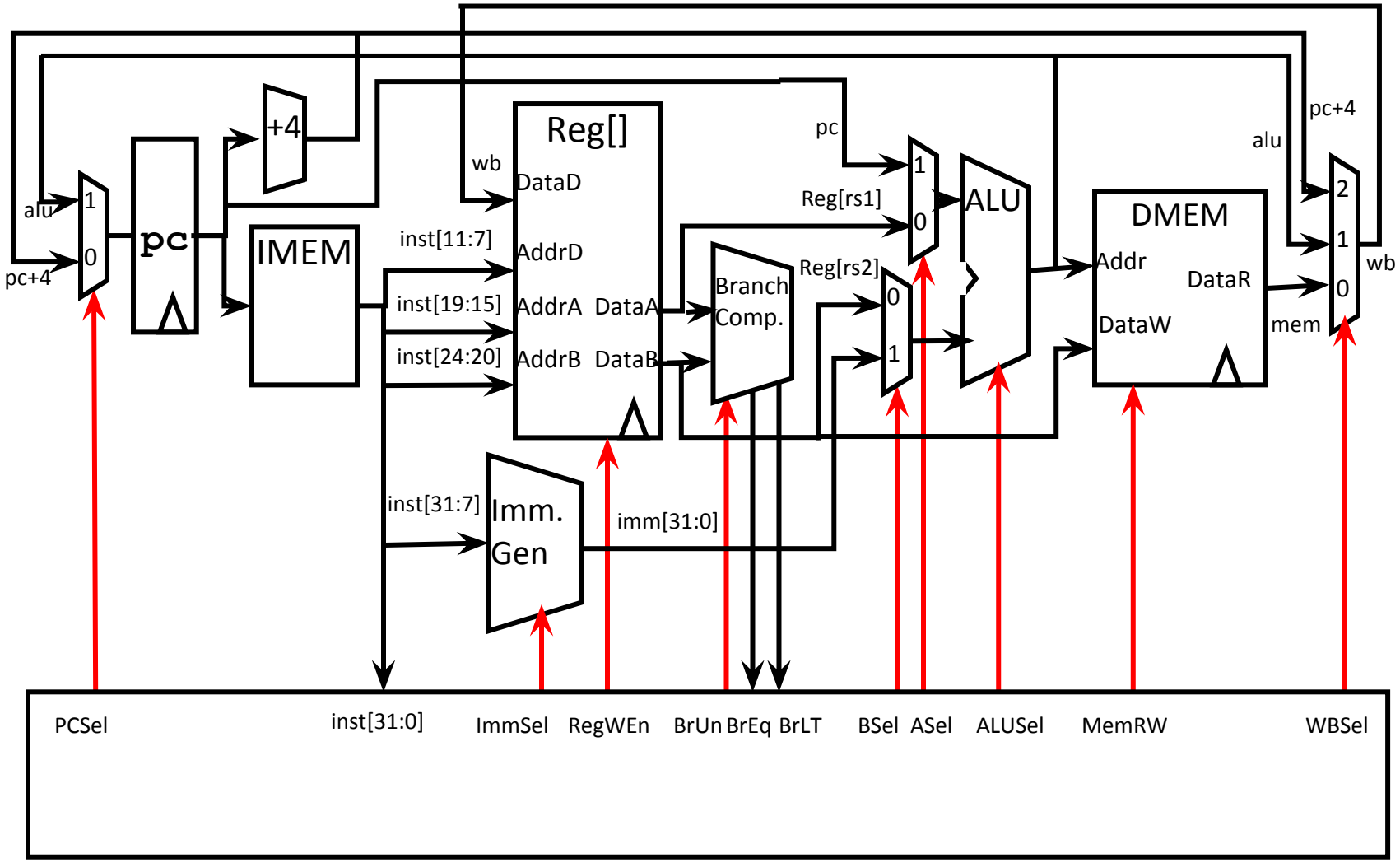
# Adding `jal` to datapath



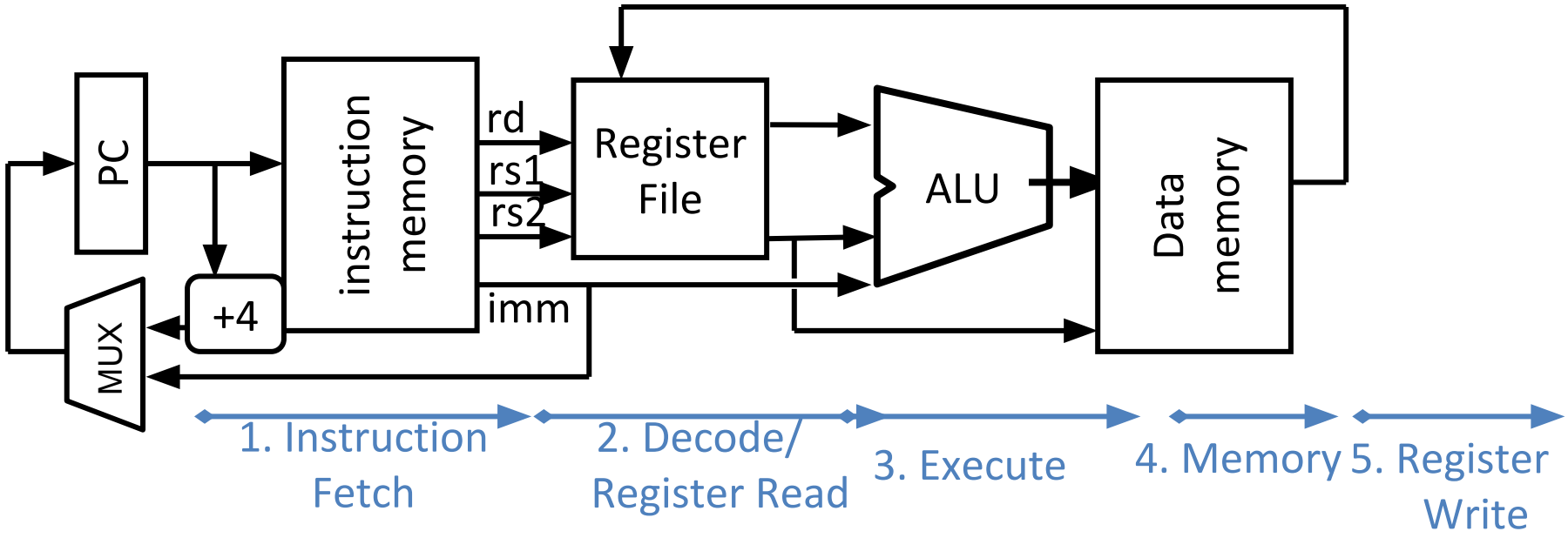
# Adding `jal` to datapath



# Single-Cycle RISC-V RV32I Datapath



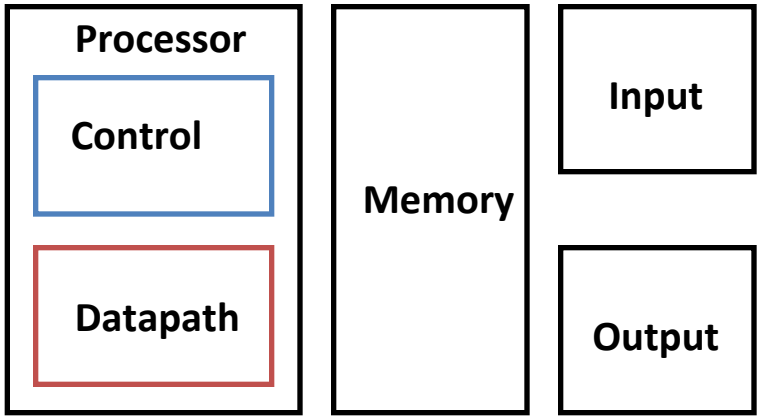
**Question:** Which of the following RISC-V instructions is active in the *most* stages?




**(A)** `lw` **(B)** `jal` **(C)** `beq` **(D)** `add`

# Processor Design Process

- Five steps to design a processor:
  1. Analyze instruction set → datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points that affect the register transfer
  5. Assemble the control logic
     - Formulate Logic Equations
     - Design Circuits



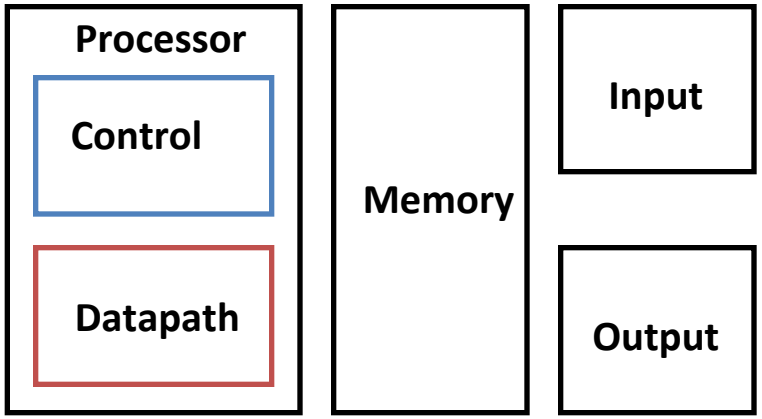

Now

# Control

- Need to make sure that correct parts of the datapath are being used for each instruction
  - Have seen *control signals* in datapath used to select inputs and operations
  - For now, focus on what value each control signal should be for each instruction in the ISA
    - Next lecture, we will see how to implement the proper combinational logic to implement the control

# Summary (1/2)

- Five steps to design a processor:
  1) Analyze instruction set → datapath requirements
  2) Select set of datapath components & establish clock methodology
  3) Assemble datapath meeting the requirements
  4) Analyze implementation of each instruction to determine setting of control points that effects the register transfer
  5) Assemble the control logic
     - Formulate Logic Equations
     - Design Circuits



# Summary (2/2)

- Determining control signals
  - Any time a datapath element has an input that changes behavior, it requires a control signal (e.g. ALU operation, read/write)
  - Any time you need to pass a different input based on the instruction, add a **MUX** with a control signal as the selector (e.g. next PC, ALU input, register to write to)
- Your control signals will change based on your exact datapath
- Your datapath will change based on your ISA

# And in Conclusion, …

- Universal datapath
  - Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions
- 5 Phases of execution
  - IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases
- Controller specifies how to execute instructions
  - what new instructions can be added with just most control?