

RISC-V Processor Design

Part 1: The Datapath

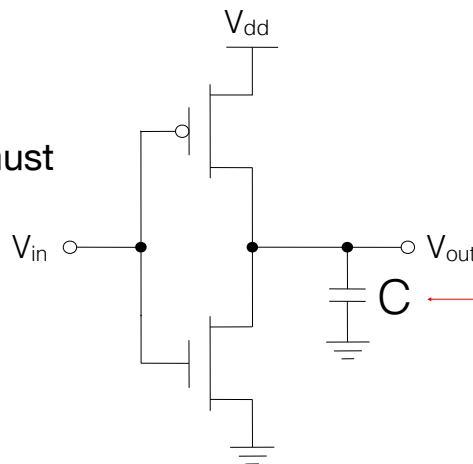
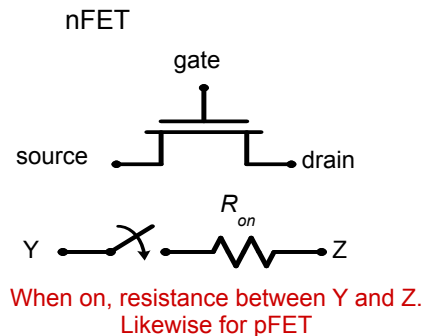
Outline:

1. Finish up CMOS circuits (nasty realities)
2. How to build a processor

Nasty Realities: Delays in CMOS circuits

More physically realistic model:

1. Transistors are not perfect switches
 - A. They leak when off
 - B. They have finite resistance when on
2. All circuit nodes have capacitance
 - To change their voltage level must displace charge

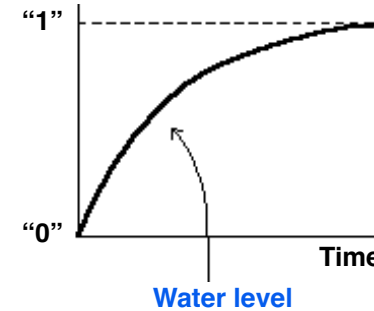
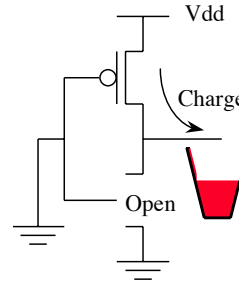


Represents the sum of all the capacitance at the output of the inverter and everything to which it connects: (drains, wires, transistor-gate capacitance of next gate(s))

Transistors as water valves

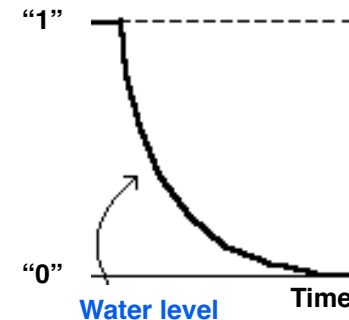
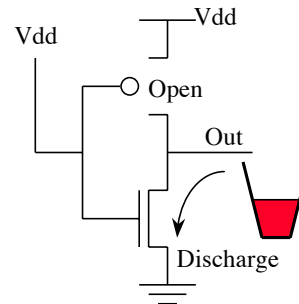
If **electrons** are water molecules, **transistor resistance** like pipe diameters, and **capacitors** are buckets ...

A “on” p-FET fills up the capacitor with charge.



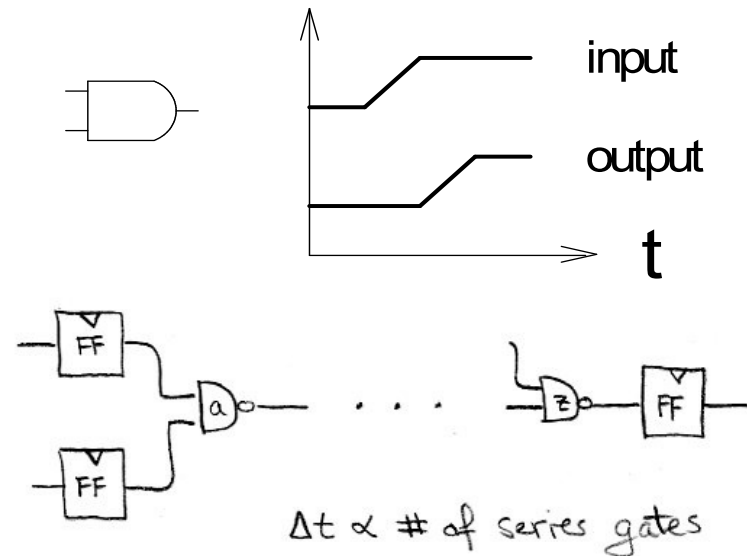
$$\tau \propto R \cdot C$$

A “on” n-FET empties the bucket.

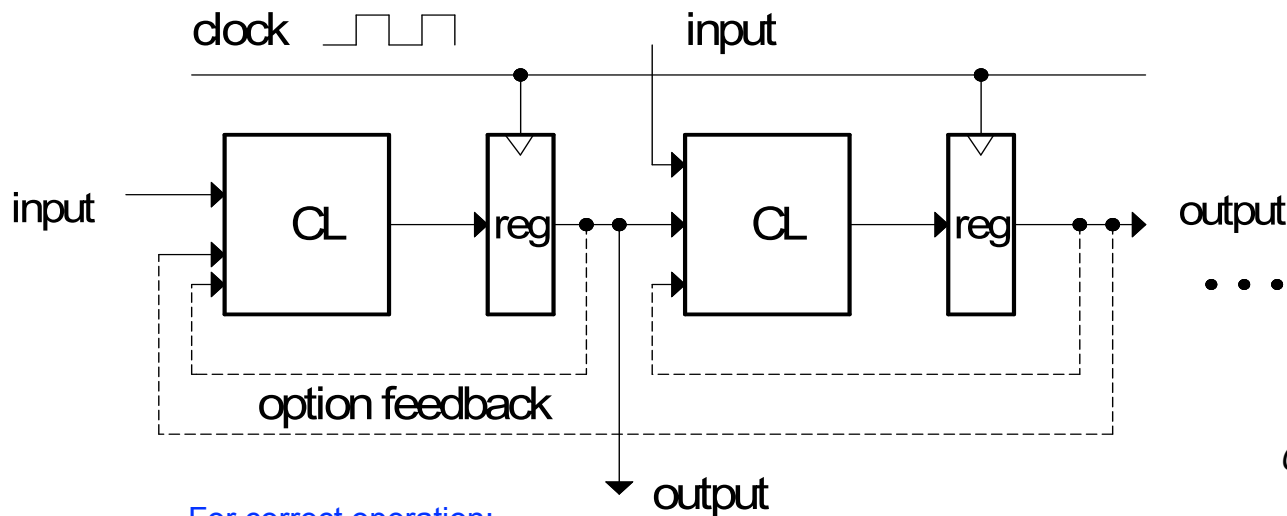


Consequences

- For every logic gate, delay from input change to output change
- The exact amount of the delay depends on:
 - type of gate, how many other gates it's output connects to, IC process details
- For cascaded gates, delay accumulates
- Remember, flip-flops also have details and timing constraints: $\tau_{clk-to-q}$ and τ_{setup}



Therefore, in General ...



For correct operation:

$$T \geq \tau_{\text{clk} \rightarrow Q} + \tau_{\text{CL}} + \tau_{\text{setup}}$$

for all paths.

The worst case path is called the “critical path”

What can we do to reduce T (increase frequency)?

More nasty realities: CMOS circuits use electrical energy (consume power)

Energy is the ability to do work (joules).

Power is rate of expending energy (watts).

Energy Efficiency: energy per operation

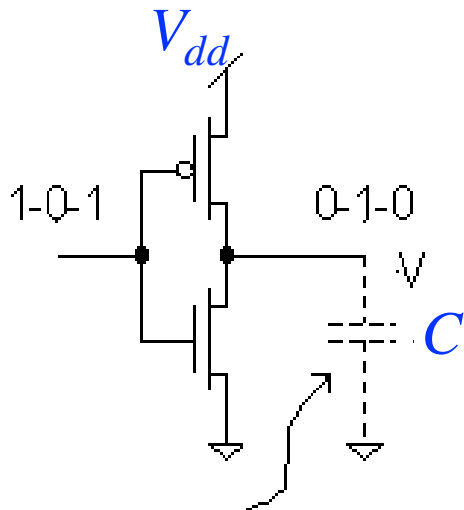
$$P = \frac{dE}{dt}$$

- **Handheld and portable** (battery operated):
 - ❑ Energy Efficiency - limits battery life
 - ❑ Power - limited by heat
- **Infrastructure and servers** (connected to power grid):
 - ❑ Energy Efficiency - dictates operation cost
 - ❑ Power - heat removal contributes to TCO

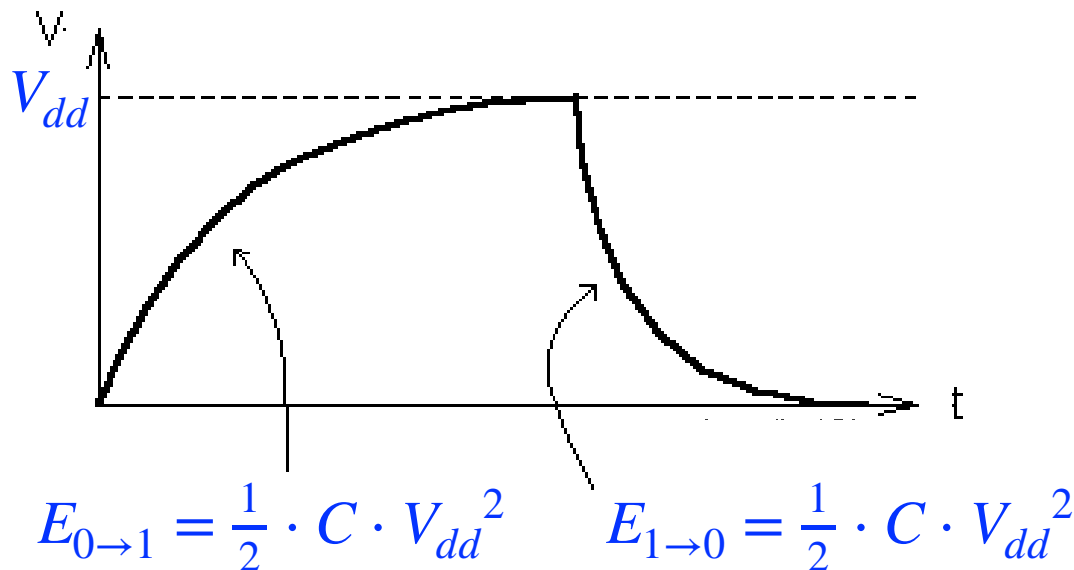


Switching Energy: Fundamental Physics

Every logic transition dissipates energy.



Models inputs to other gates & wire capacitance



Chip-Level “switching” Power

$$P = dE / dt$$

$$P_{sw} = 1/2 \alpha C V_{dd}^2 F$$

“activity factor”, average percentage of capacitance switching per cycle (~ number of nodes to switch)

Total chip capacitance to be switched

Clock Frequency

Reducing power consumption or improving energy efficiency

$$P_{sw} = 1/2 \alpha C V_{dd}^2 F$$

- Power proportional to F . Can reduce power by reducing frequency. But that doesn't improve energy efficiency (just spreads computation over longer time)
- Energy efficiency:
 - $E_{sw} \propto V_{dd}^2$ but $\tau_{logic} \propto V_{dd}$
 - Therefore can improve energy efficiency by lowering supply voltage and making up for less performance by using parallelism

Great Idea #1: Abstraction

```
lw  t0, t2, 0
lw  t1, t2, 4
sw  t1, t2, 0
sw  t0, t2, 4
```

High Level Language
Program (e.g., C)

Compiler

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

Assembler

Machine Language
Program (RISC-V)

*Machine
Interpretation*

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

*Architecture
Implementation*

Logic Circuit Description
(Circuit Schematic Diagrams)

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

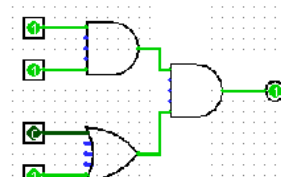
Anything can be represented
as a *number*,
i.e., data or instructions

```
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
```

Register File

ALU

We are here!



Welcome To Marionette Time

- The next couple of lectures are going to describe how we build a complete processor
 - Consists of two pieces:
 - The **Datapath** that implements the computation
 - The **Control Logic** that looks at the instruction and tells the datapath what to do
- Think of it like a Marionette
 - The puppet is the datapath
 - The strings are pulled the control logic
- Today we will mostly focus on the puppet



Recap: Complete RV32I ISA

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20:10:11:19:12]				rd	1101111	JAL
imm[11:0]				rd	1100111	JALR
imm[12:10:5]	rs2	rs1	000	imm[4:1:11]	1100011	BEQ
imm[12:10:5]	rs2	rs1	001	imm[4:1:11]	1100011	BNE
imm[12:10:5]	rs2	rs1	100	imm[4:1:11]	1100011	BLT
imm[12:10:5]	rs2	rs1	101	imm[4:1:11]	1100011	BGE
imm[12:10:5]	rs2	rs1	110	imm[4:1:11]	1100011	BLTU
imm[12:10:5]	rs2	rs1	111	imm[4:1:11]	1100011	BGEU
imm[11:0]				rd	0000011	LB
imm[11:0]				rd	0000011	LH
imm[11:0]				rd	0000011	LW
imm[11:0]				rd	0000011	LBU
imm[11:0]				rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]				rd	0010011	ADDI
imm[11:0]				rd	0010011	SLTI
imm[11:0]				rd	0010011	SLTIU
imm[11:0]				rd	0010011	XORI
imm[11:0]				rd	0010011	ORI
imm[11:0]				rd	0010011	ANDI

0000000	shamt	rs1	001	rd	0010011	SLLI	
0000000	shamt	rs1	101	rd	0010011	SRLI	
0100000	shamt	rs1	101	rd	0010011	SRAI	
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	
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
0000000000000			00000	000	00000	1110011	ECALL
0000000000001			00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW	
csr		rs1	0	rd	1110011	CSRRS	
csr		rs1	011	rd	1110011	CSRRC	
csr		zimm	101	rd	1110011	CSRRWI	
csr		zimm	110	rd	1110011	CSRRSI	
csr		zimm	111	rd	1110011	CSRRCI	

Not in CS61C

“State” Required by RV32I ISA

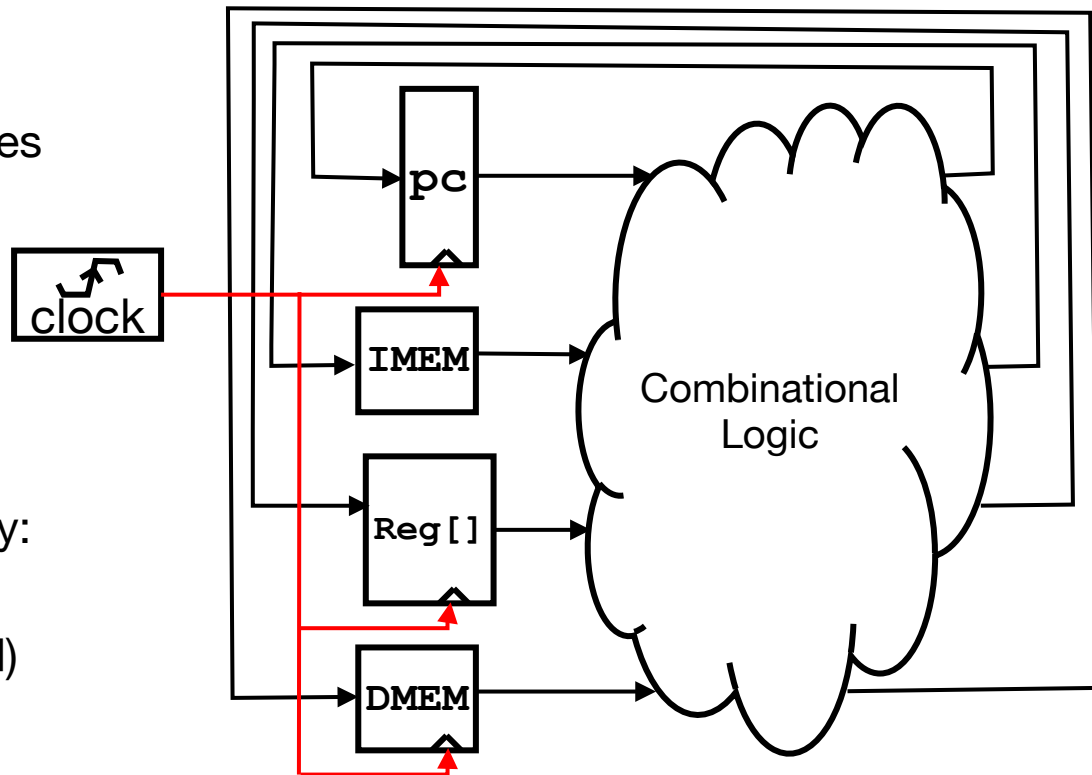
Each instruction reads and updates this state during execution:

- 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

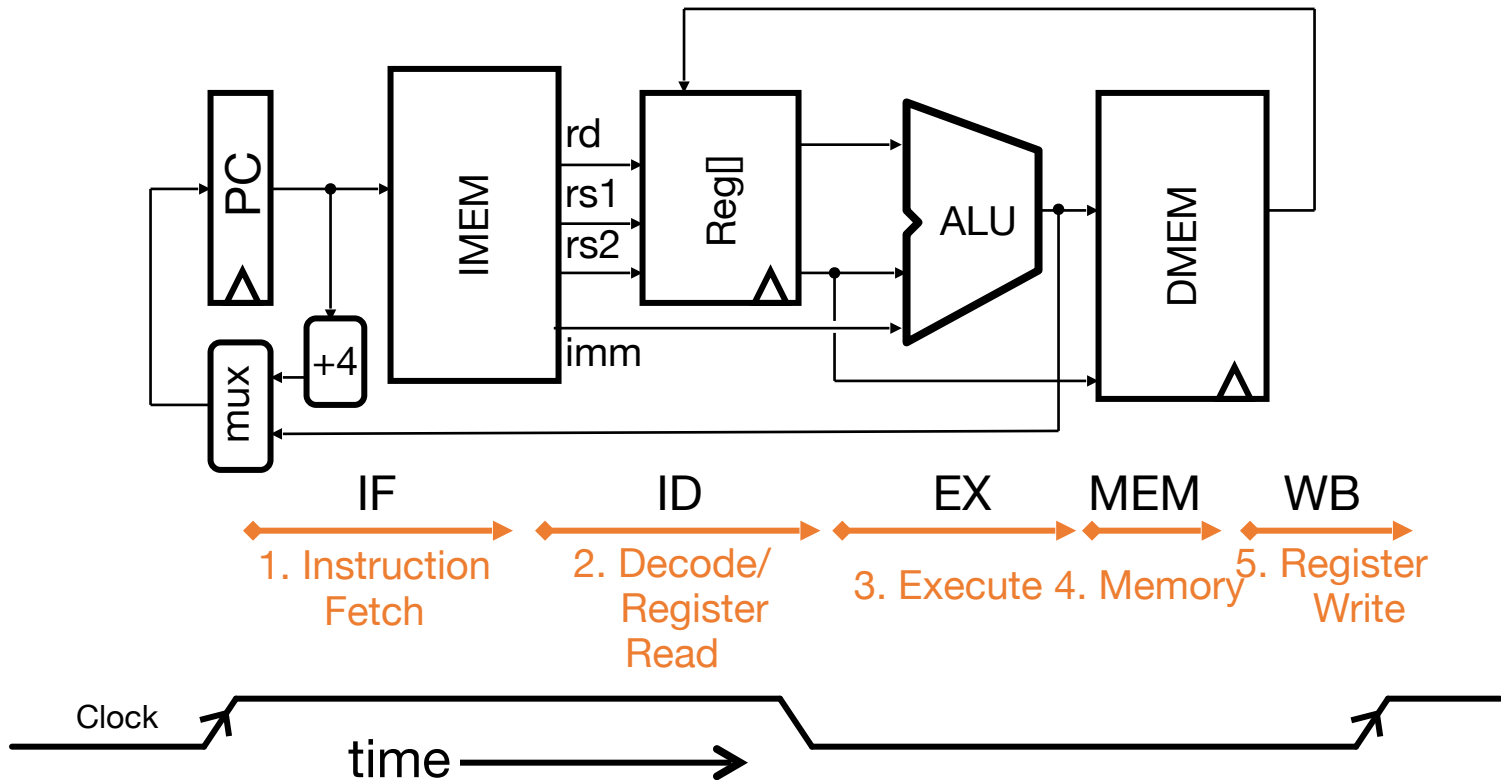
One-Instruction-Per-Cycle RISC-V Machine

On every tick of the clock, the processor executes one instruction

1. Current state outputs drive the inputs to the combinational logic, whose outputs settle at the values of the state before the next clock edge
2. At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle
3. Separate instruction/data memory: For simplification, memory is asynchronous read (not clocked), but synchronous write (is clocked)



Basic Phases of Instruction Execution



Implementing the `add` instruction

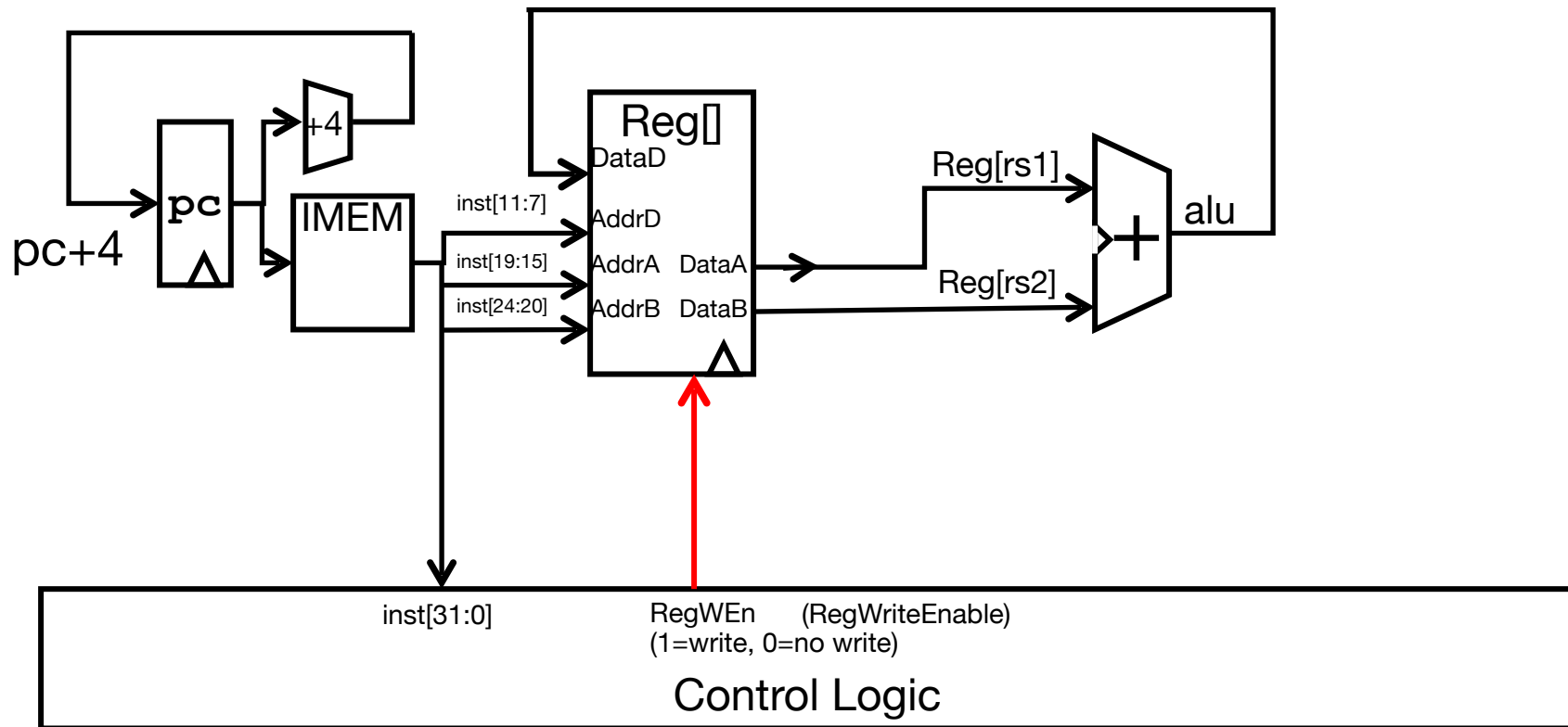
0000000	rs2	rs1	000	rd	0110011	ADD
---------	-----	-----	-----	----	---------	-----

- Instruction makes two changes to machine's state:

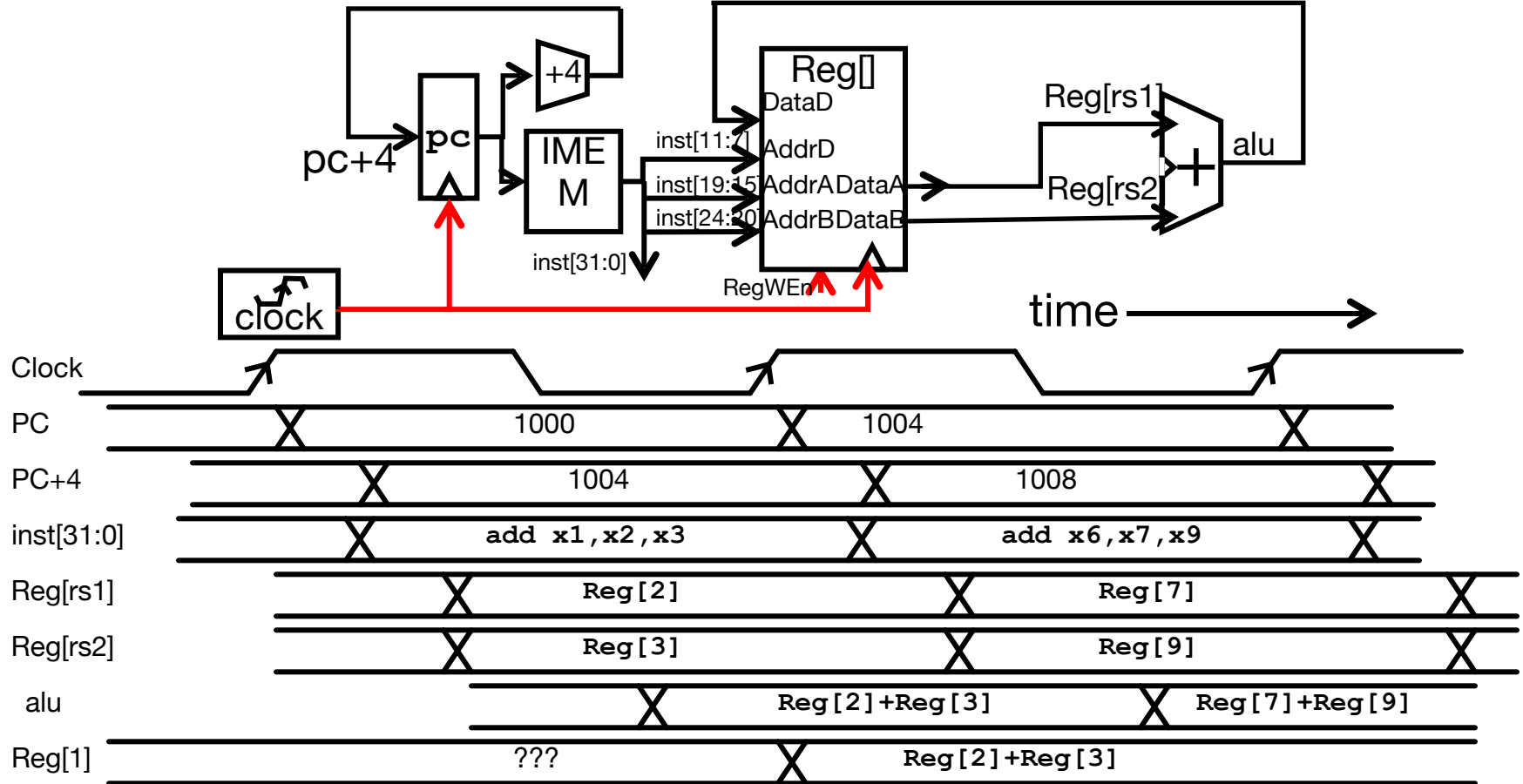
$\text{Reg}[\text{rd}] = \text{Reg}[\text{rs1}] + \text{Reg}[\text{rs2}]$

$\text{PC} = \text{PC} + 4$

Datapath for add



Timing Diagram for add

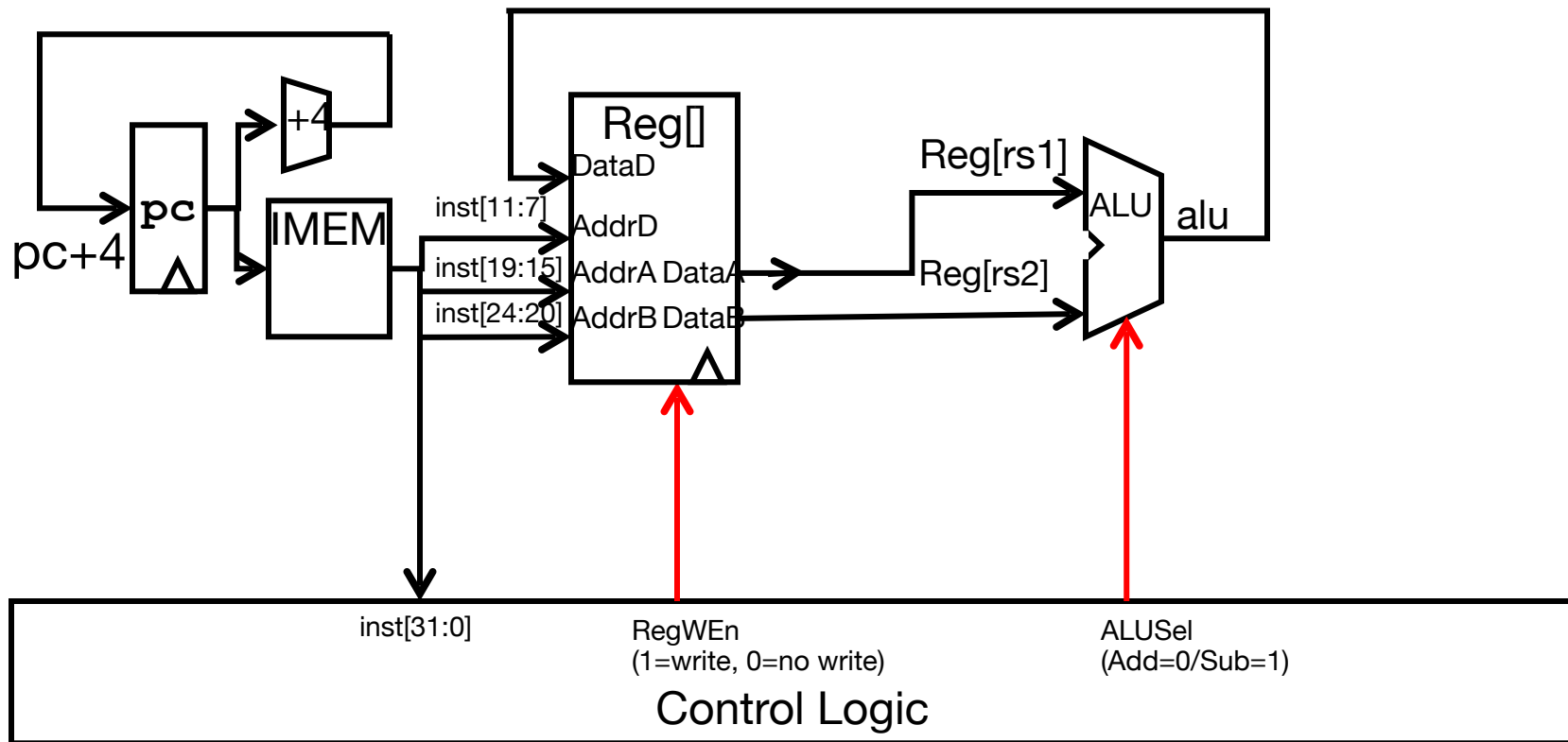


Implementing the **sub** instruction

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

- 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



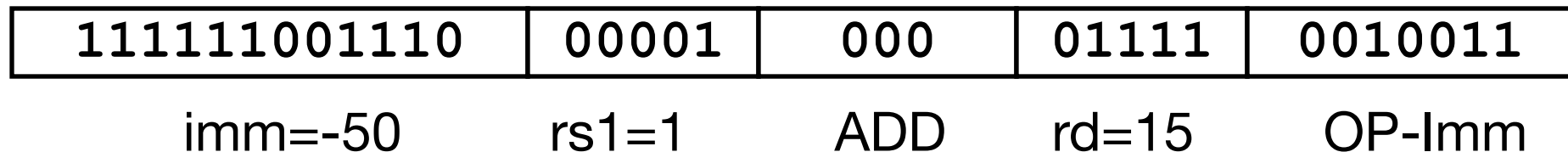
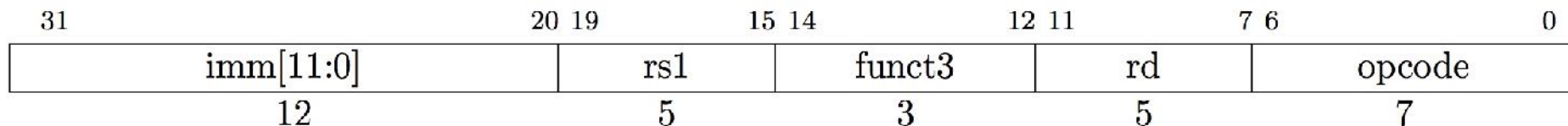
Implementing other R-Format instructions

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

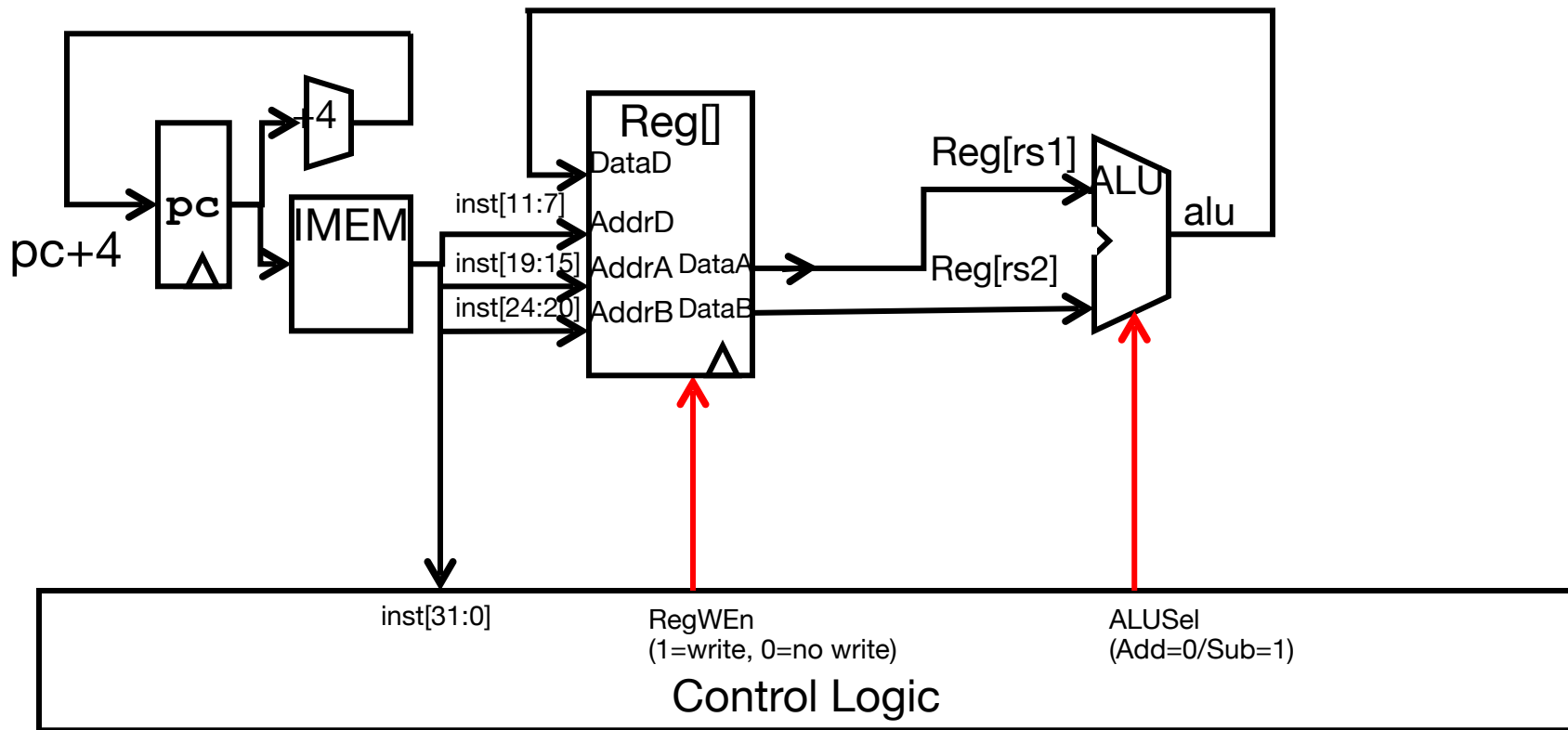
Implementing the `addi` instruction

- RISC-V Assembly Instruction:

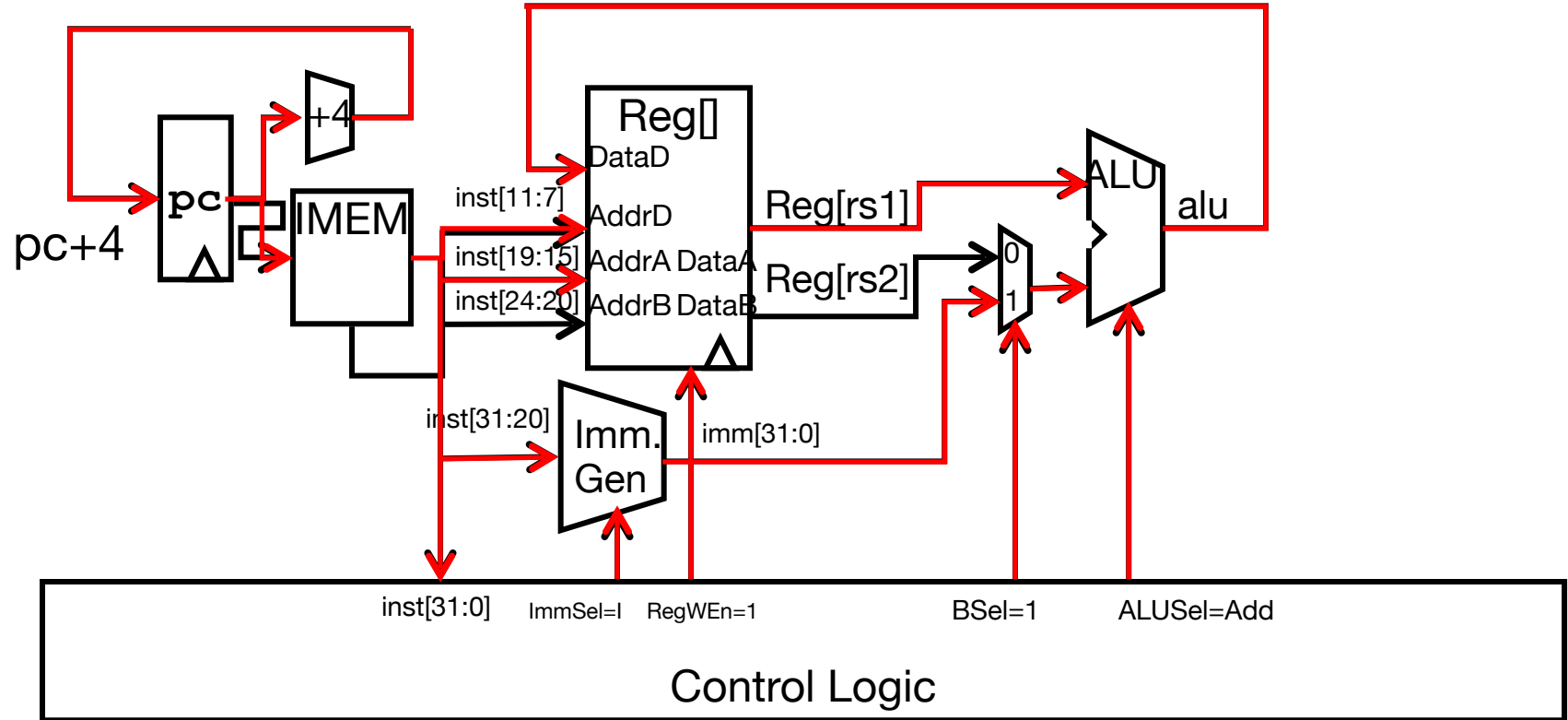
`addi x15, x1, -50`



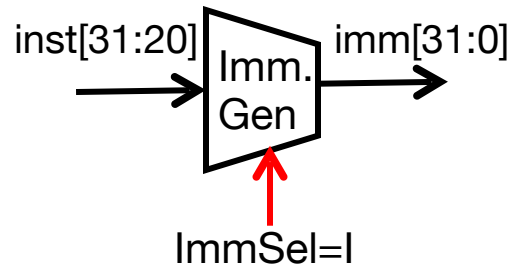
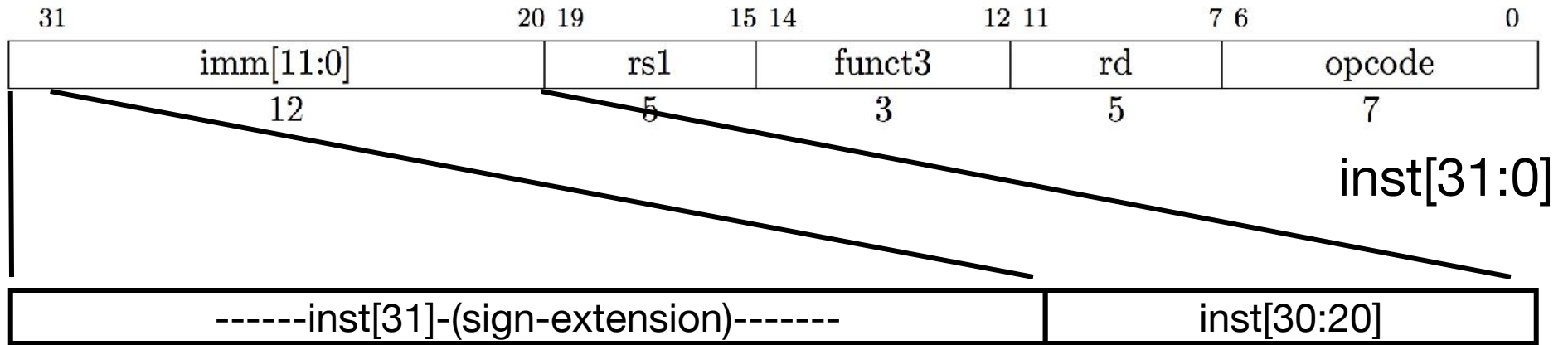
Datapath for add/sub



Adding `addi` to datapath

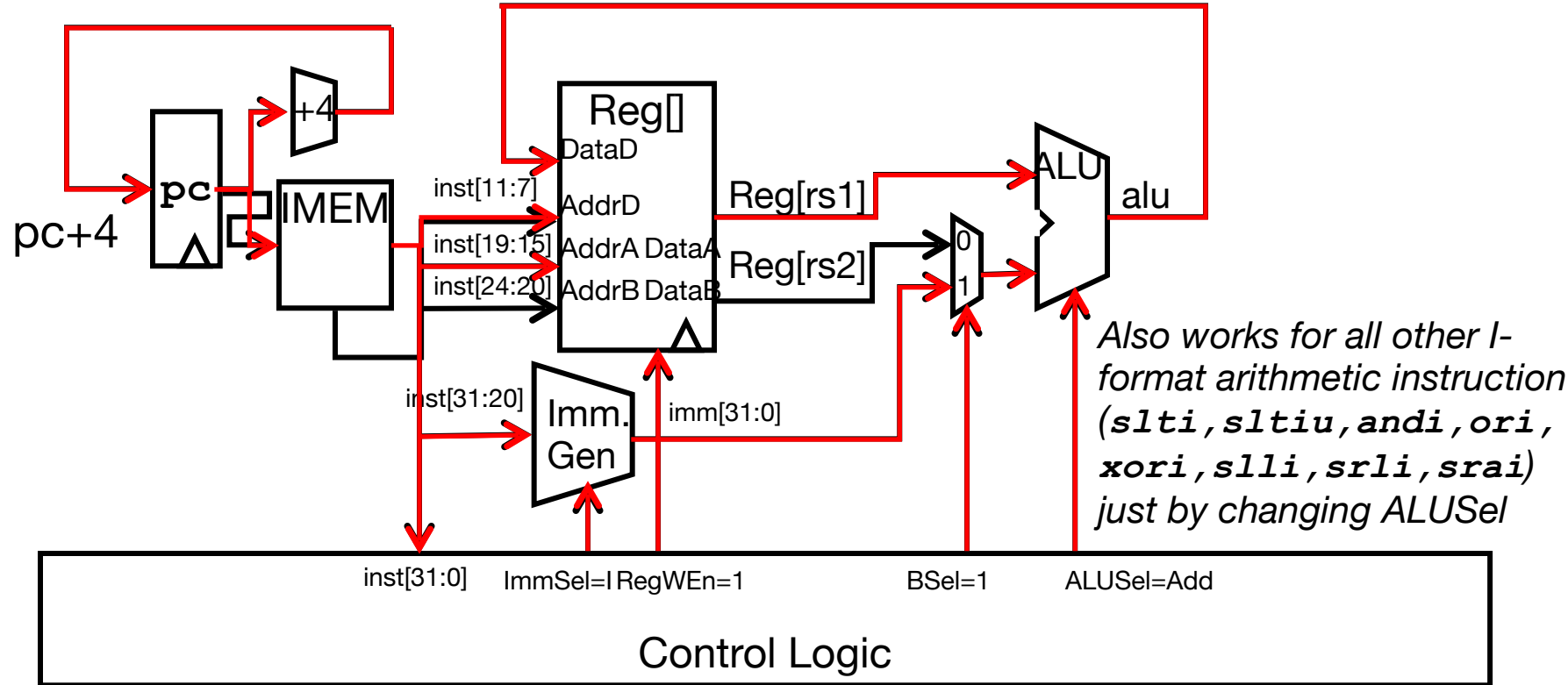


I-Format immediates



- 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 ($imm[31:12]$)

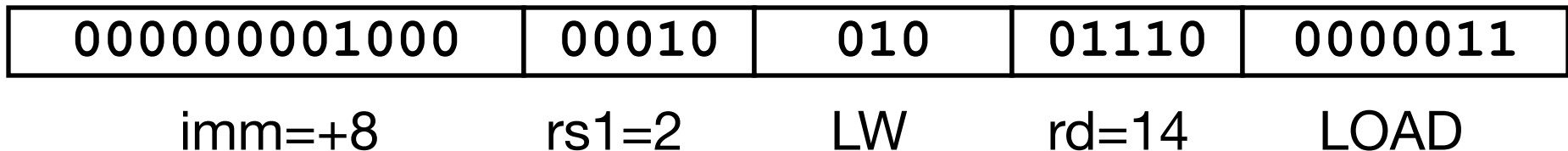
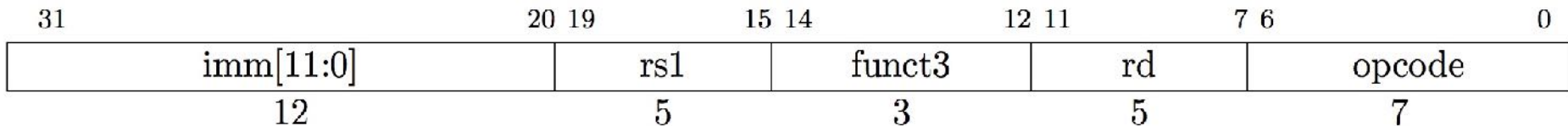
Adding `addi` to datapath



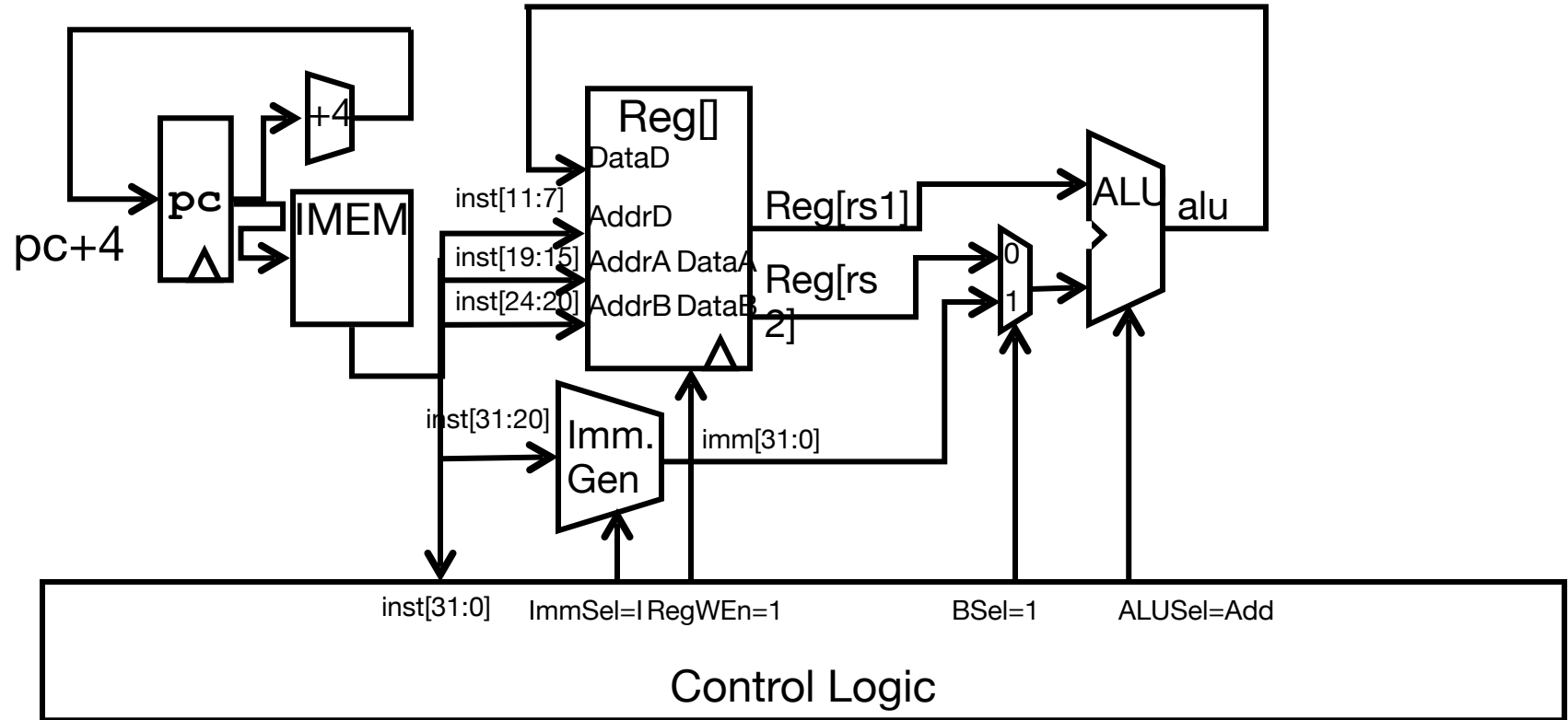
Implementing Load Word instruction

- RISC-V Assembly Instruction:

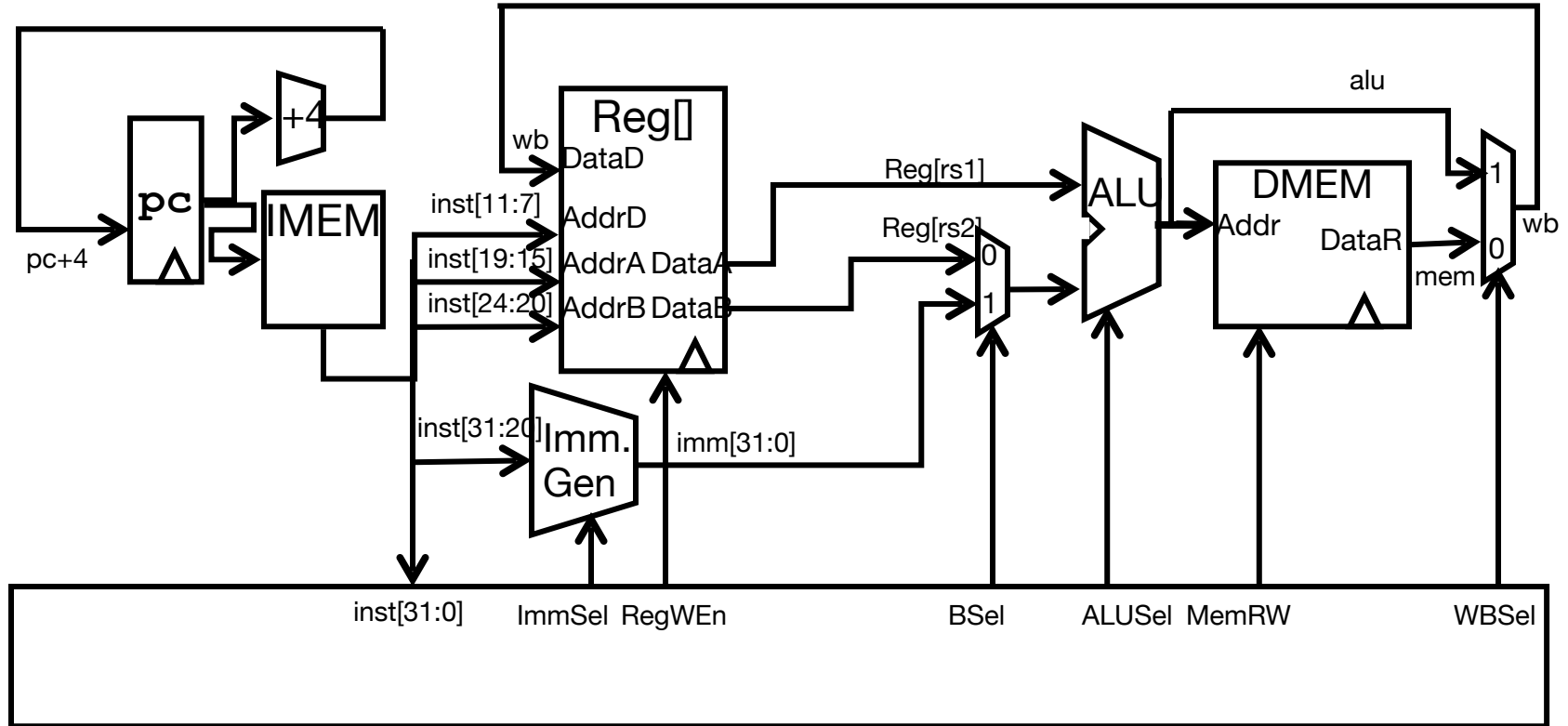
lw x14, 8(x2)



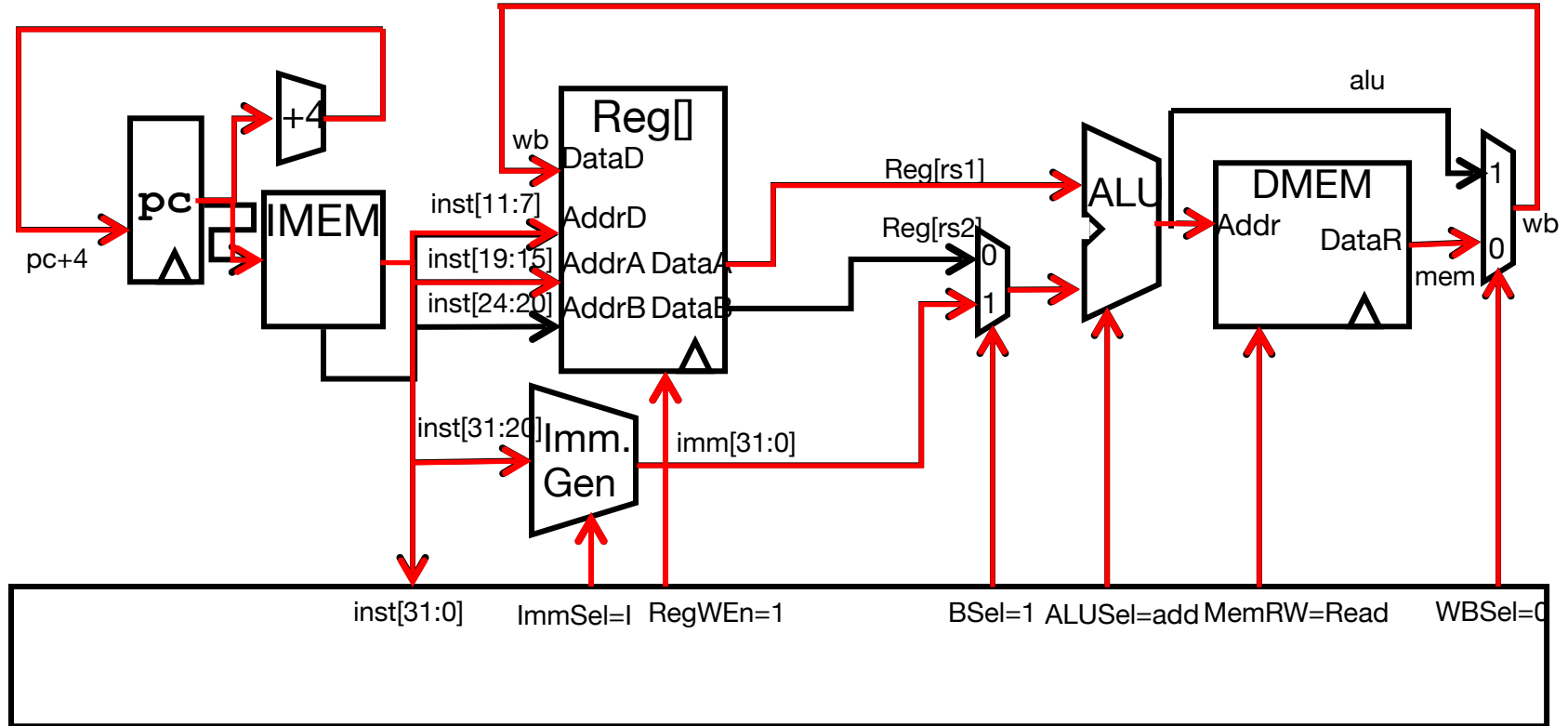
Adding `addi` to datapath



Adding 1w to datapath



Adding 1w 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

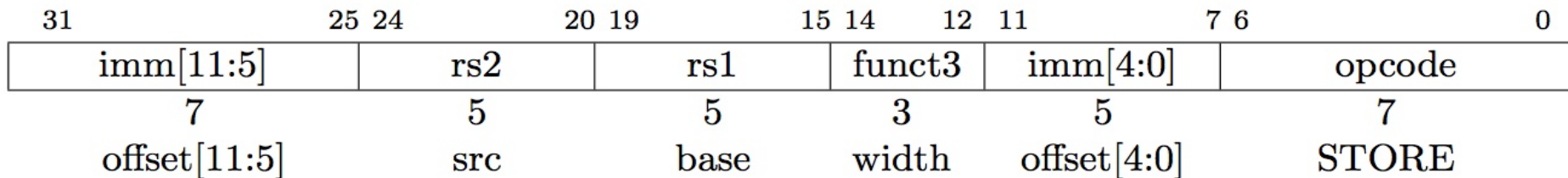
↑
funct3 field encodes size
and signedness of load
data

- 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.

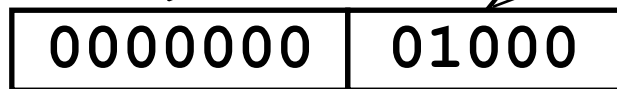
Implementing Store Word instruction

- RISC-V Assembly Instruction:

sw x14, 8(x2)

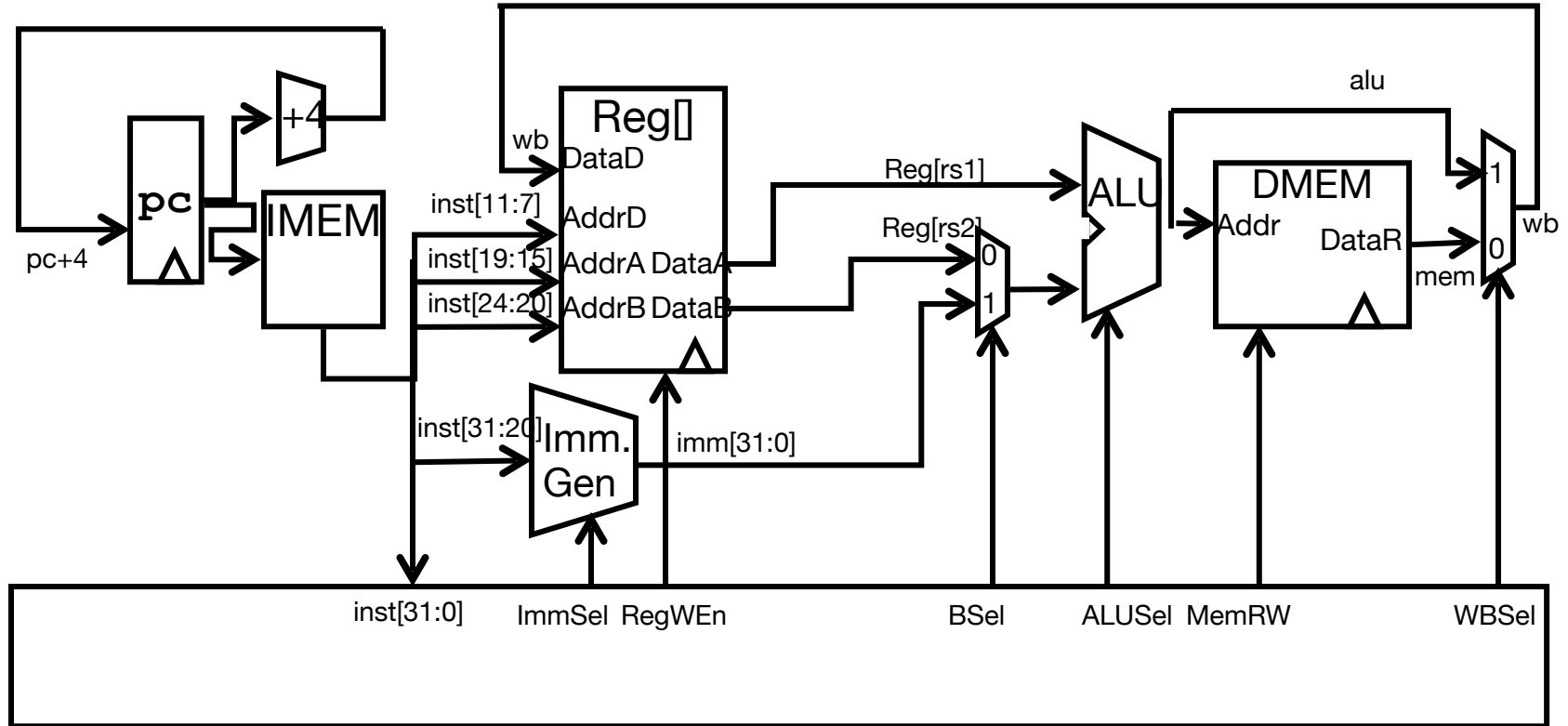


offset[11:5] = 0 rs2=14 rs1=2 SW offset[4:0] = 8 STORE

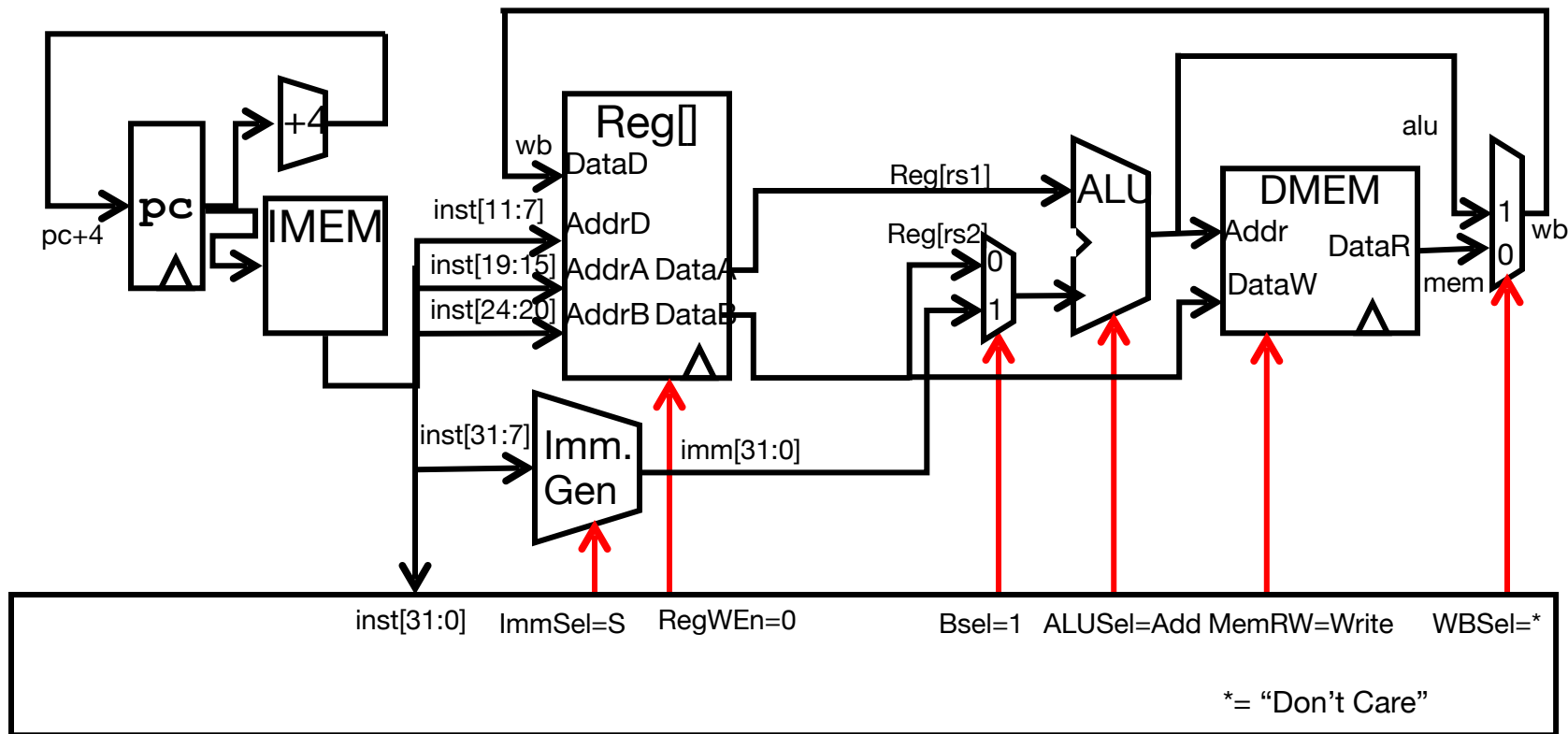


combined 12-bit offset = 8

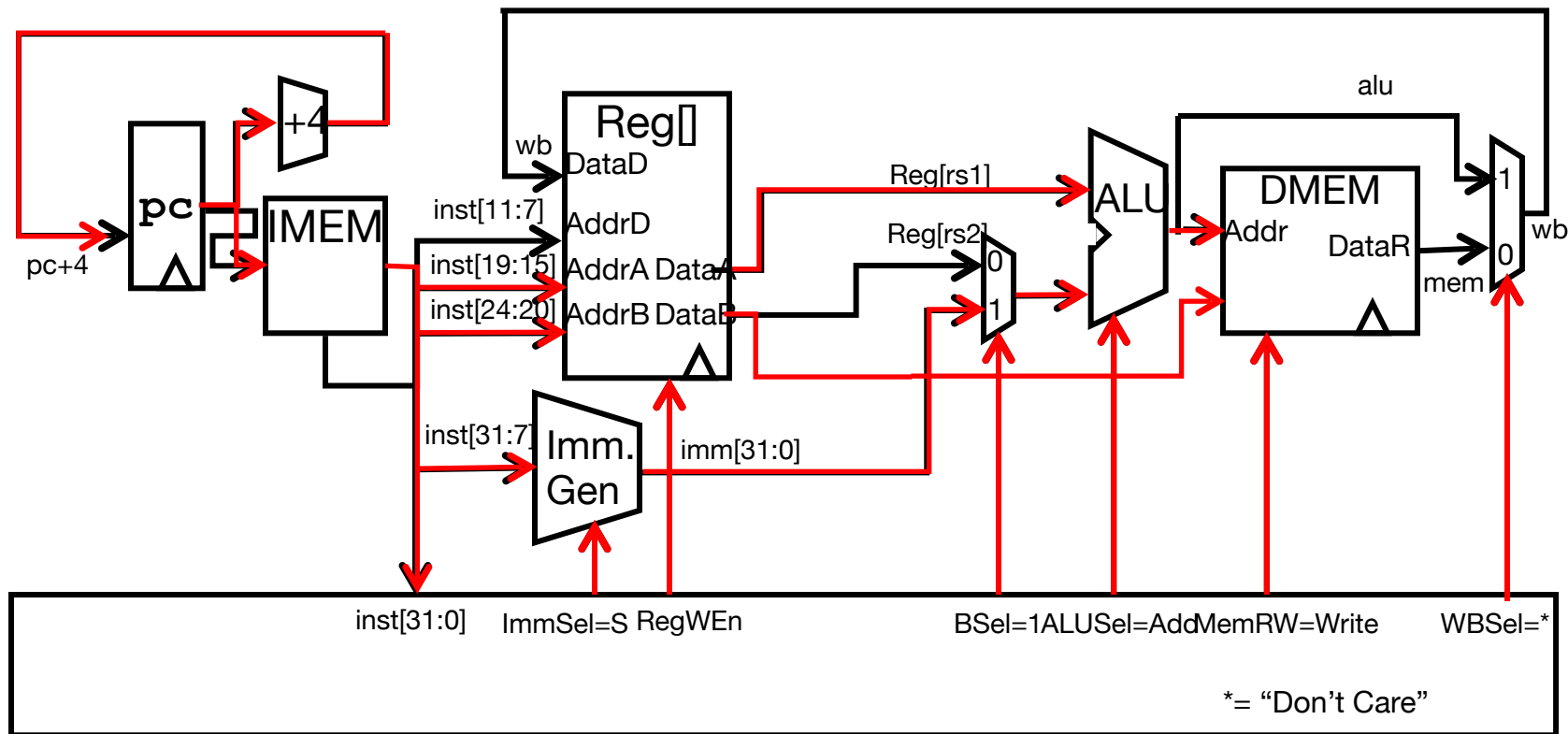
Adding 1w to datapath



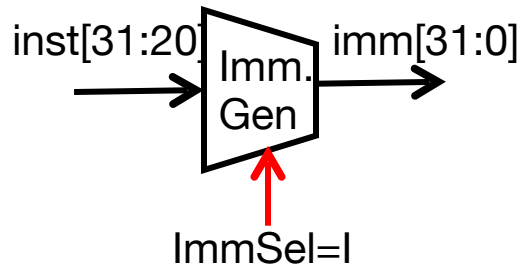
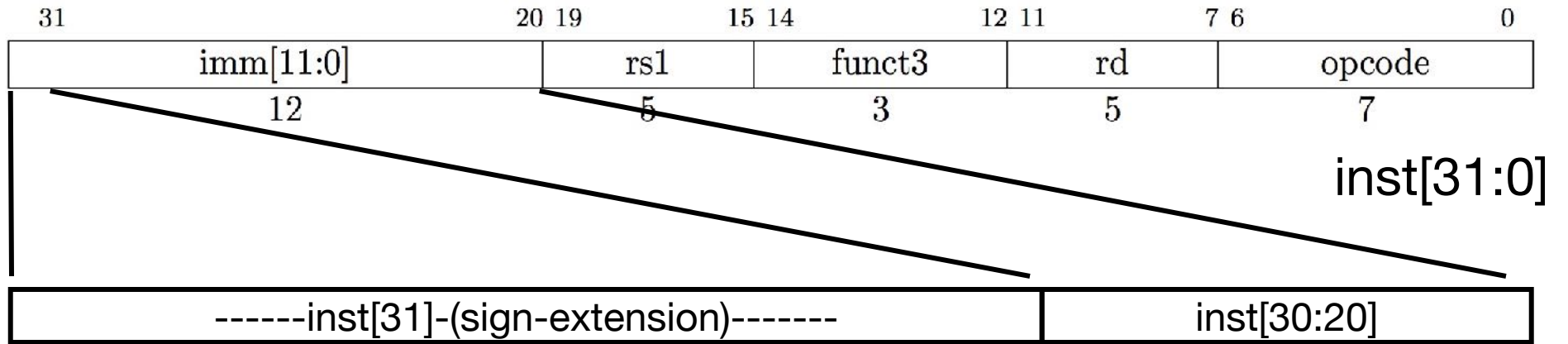
Adding **sw** to datapath



Adding **sw** to datapath

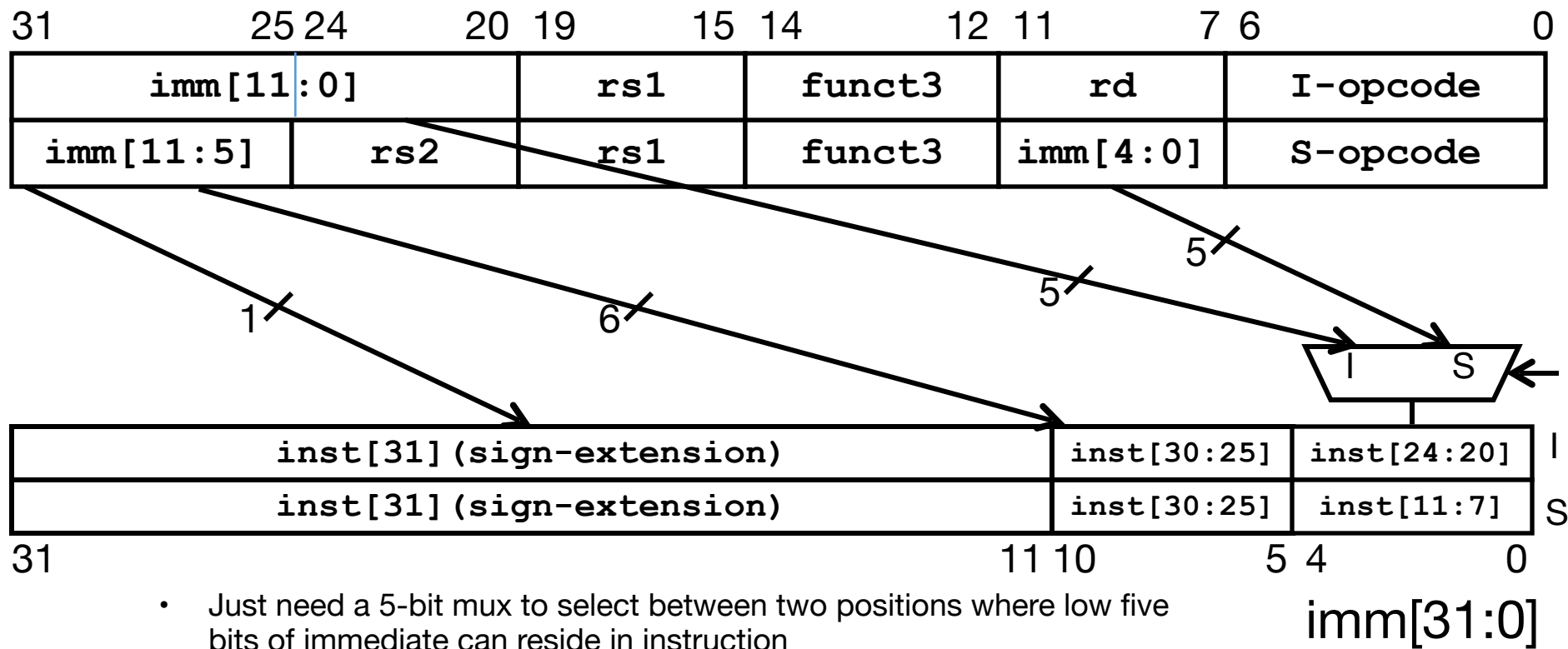


I-Format immediates



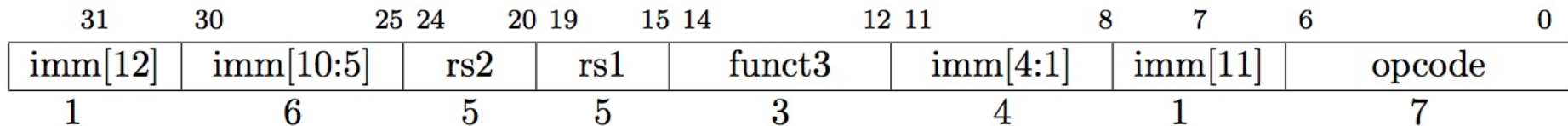
- High 12 bits of instruction (**inst[31:20]**) copied to low 12 bits of immediate (**imm[11:0]**)
- Immediate is sign-extended by copying value of **inst[31]** to fill the upper 20 bits of the immediate value (**imm[31:12]**)

I & S Immediate Generator



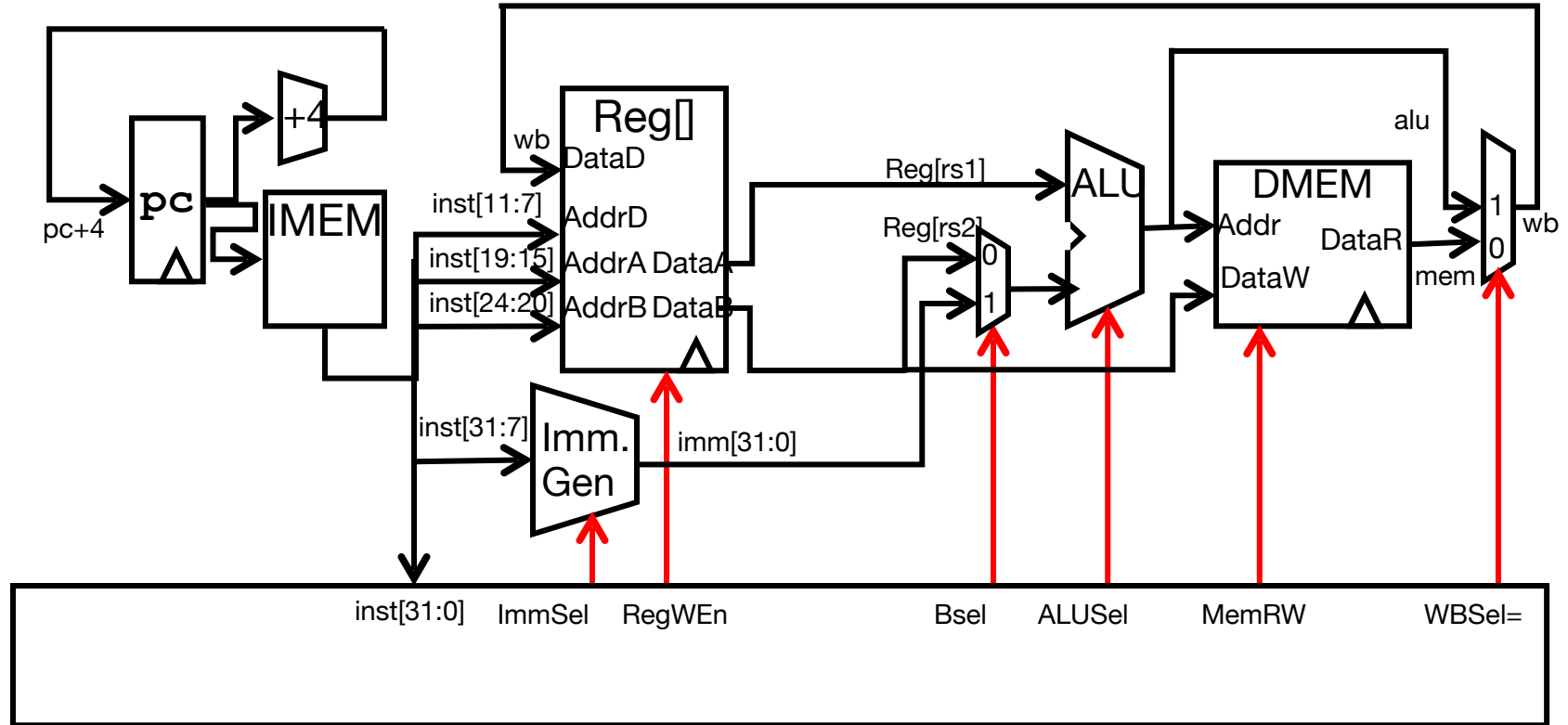
- Just need a 5-bit mux to select between two positions where low five bits of immediate can reside in instruction
- Other bits in immediate are wired to fixed positions in instruction

Implementing Branches

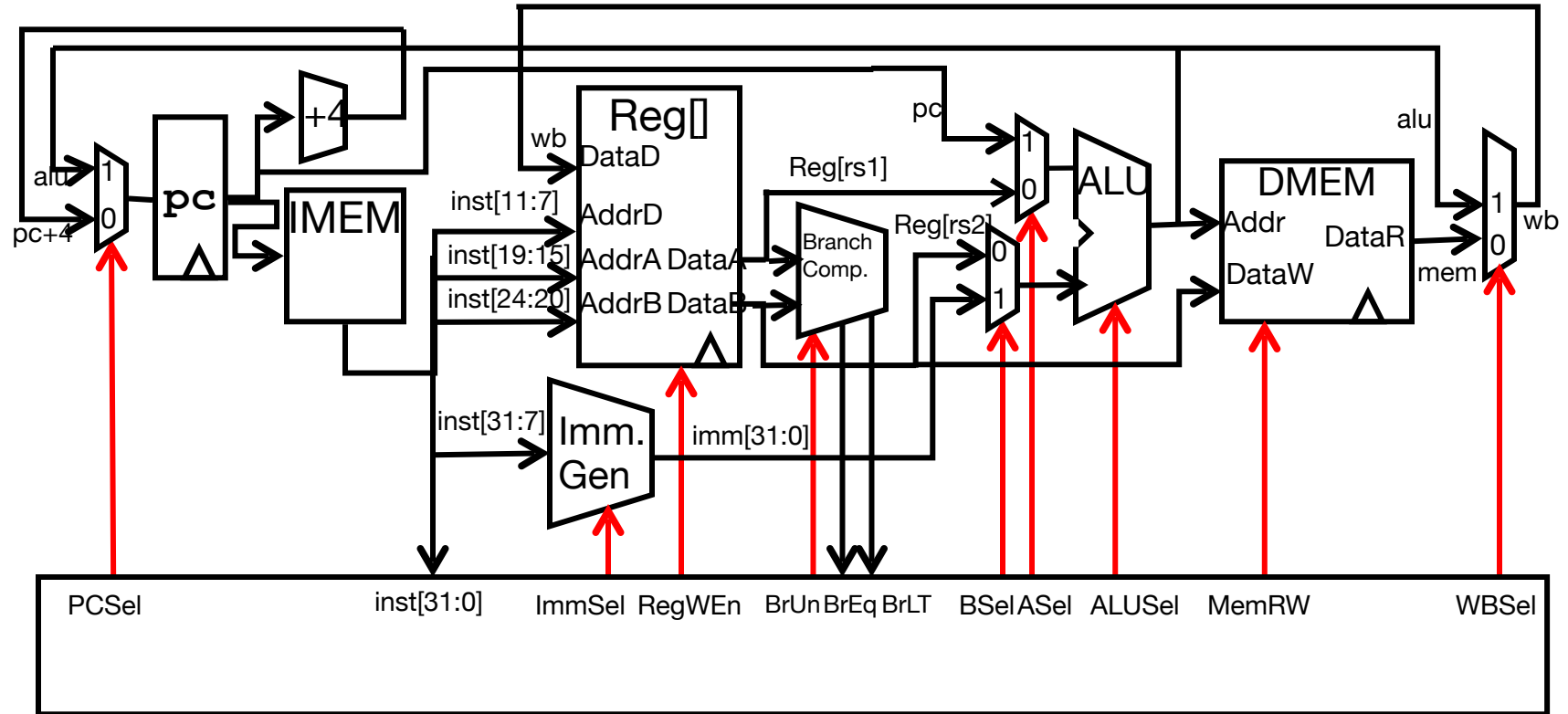


- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode *even* 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

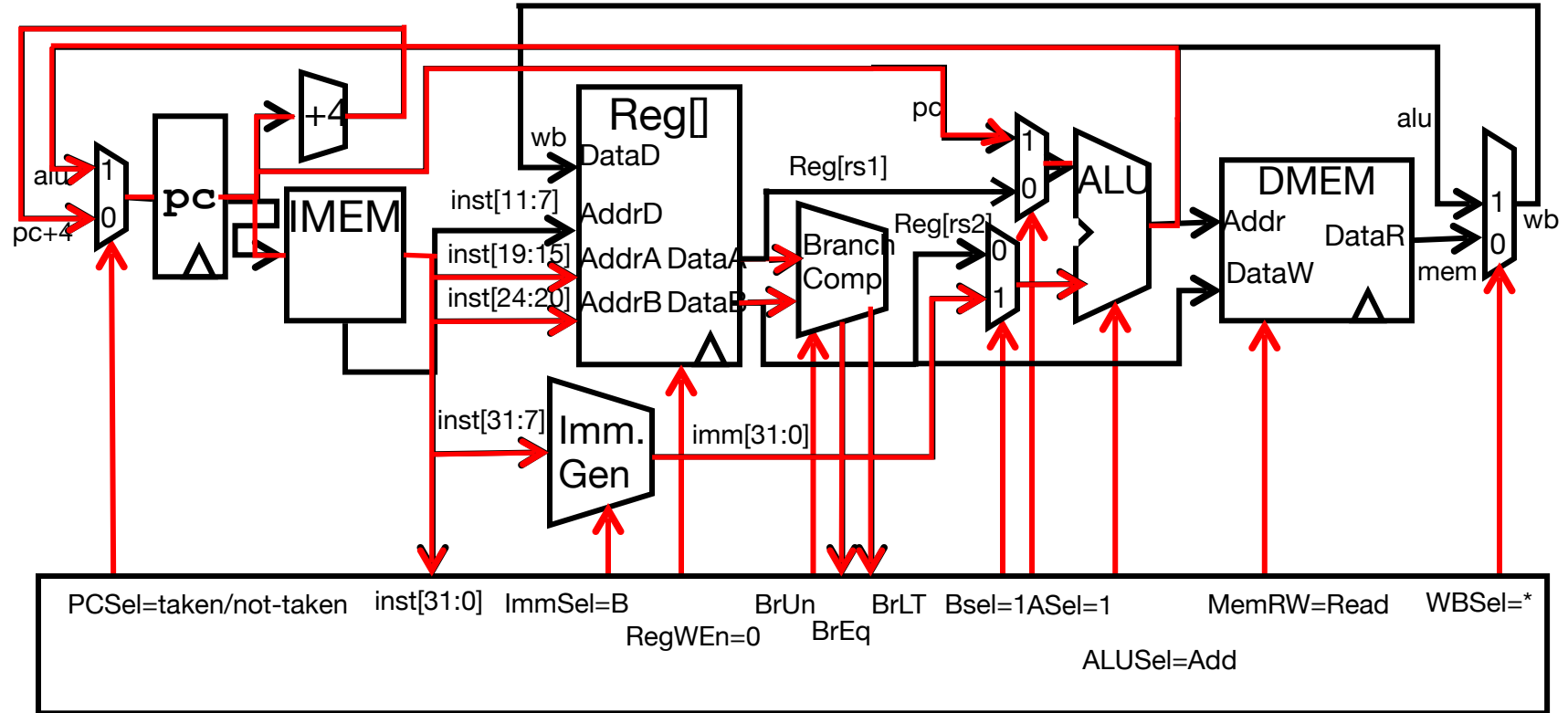
Adding **sw** to datapath



Adding branches to datapath

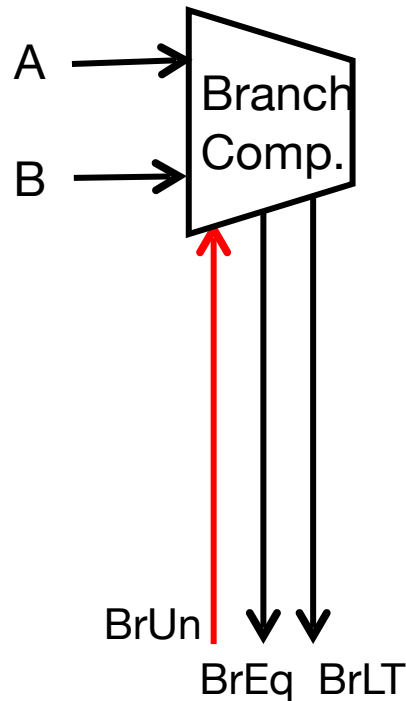


Adding branches to datapath

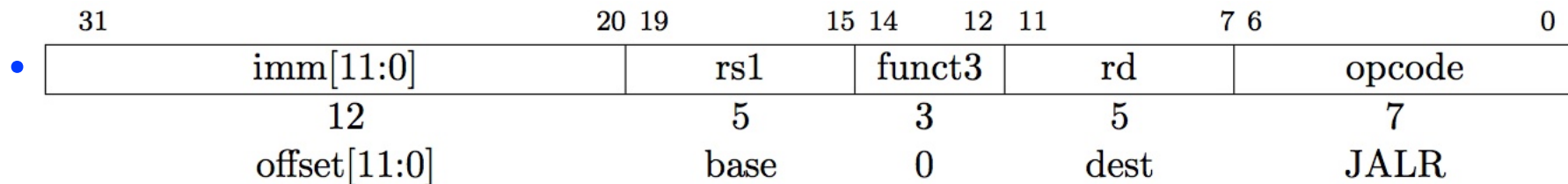


Branch Comparator

- $\text{BrEq} = 1$, if $A=B$
- $\text{BrLT} = 1$, if $A < B$
- $\text{BrUn} = 1$ selects unsigned comparison for BrLT , 0=signed
- BGE branch: $A \geq B$, if $\neg(A < B)$

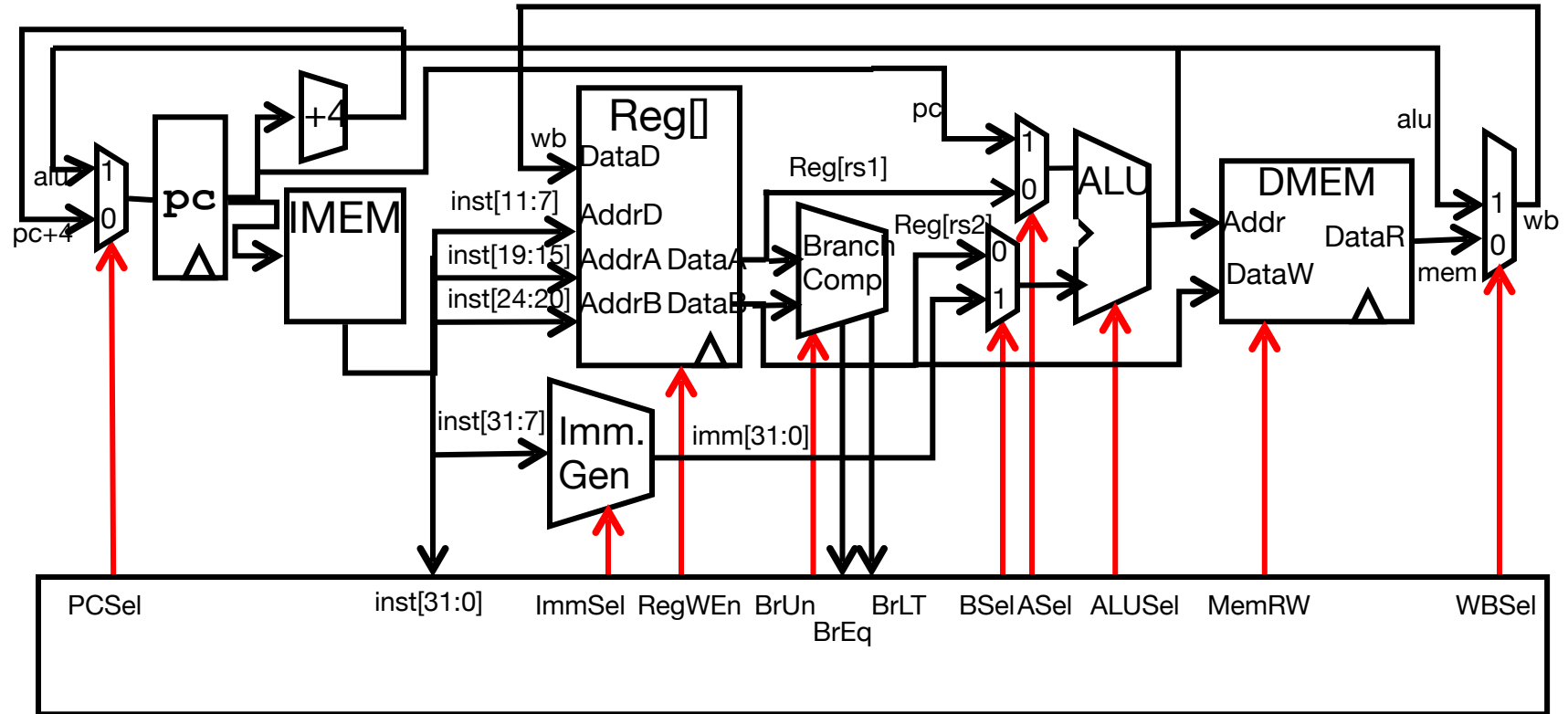


Implementing JALR Instruction (I-Format)

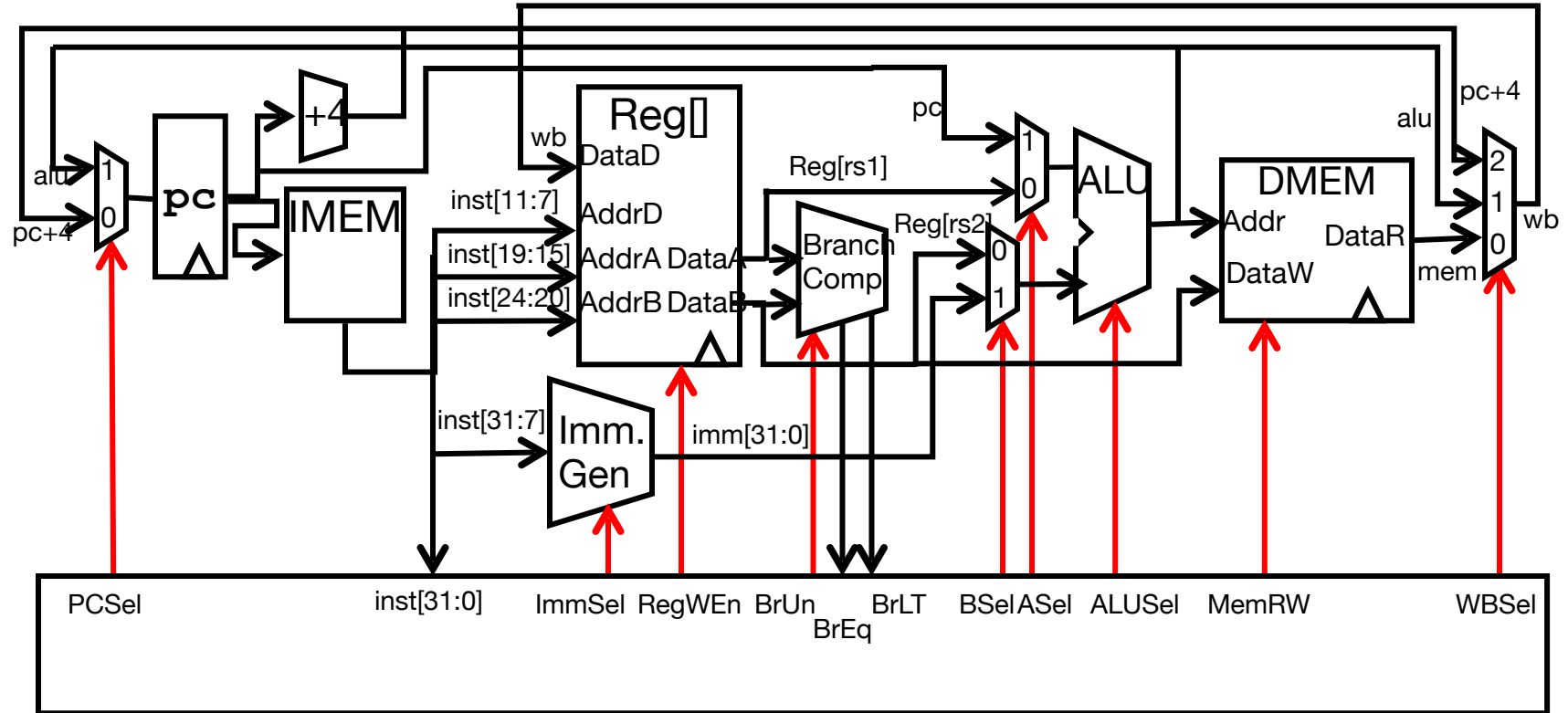


- Sets $PC = \text{Reg}[\text{rs1}] + \text{immediate}$
- Uses same immediates as arithmetic and loads
 - **no** multiplication by 2 bytes

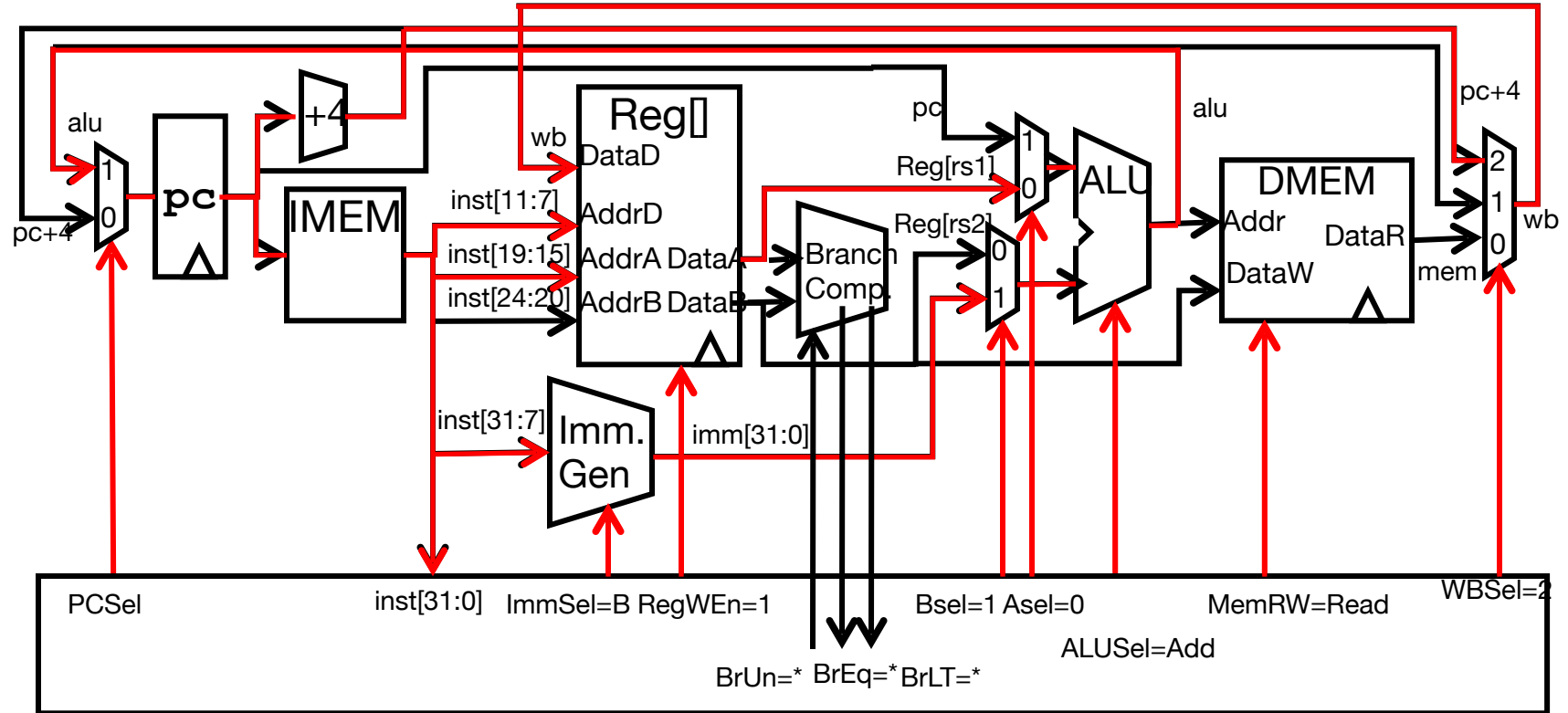
Adding branches to datapath



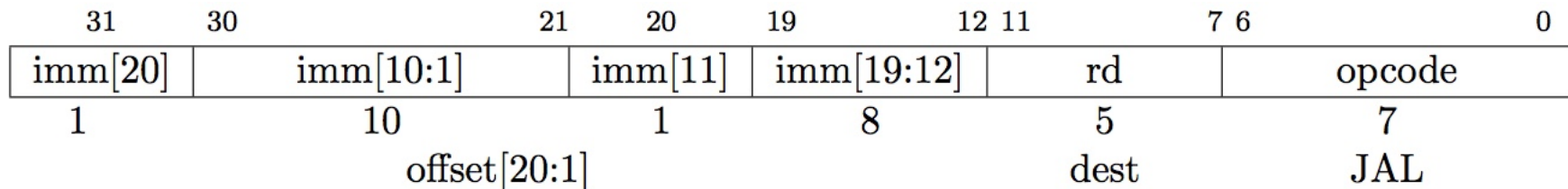
Adding jalr to datapath



Adding jalr to datapath

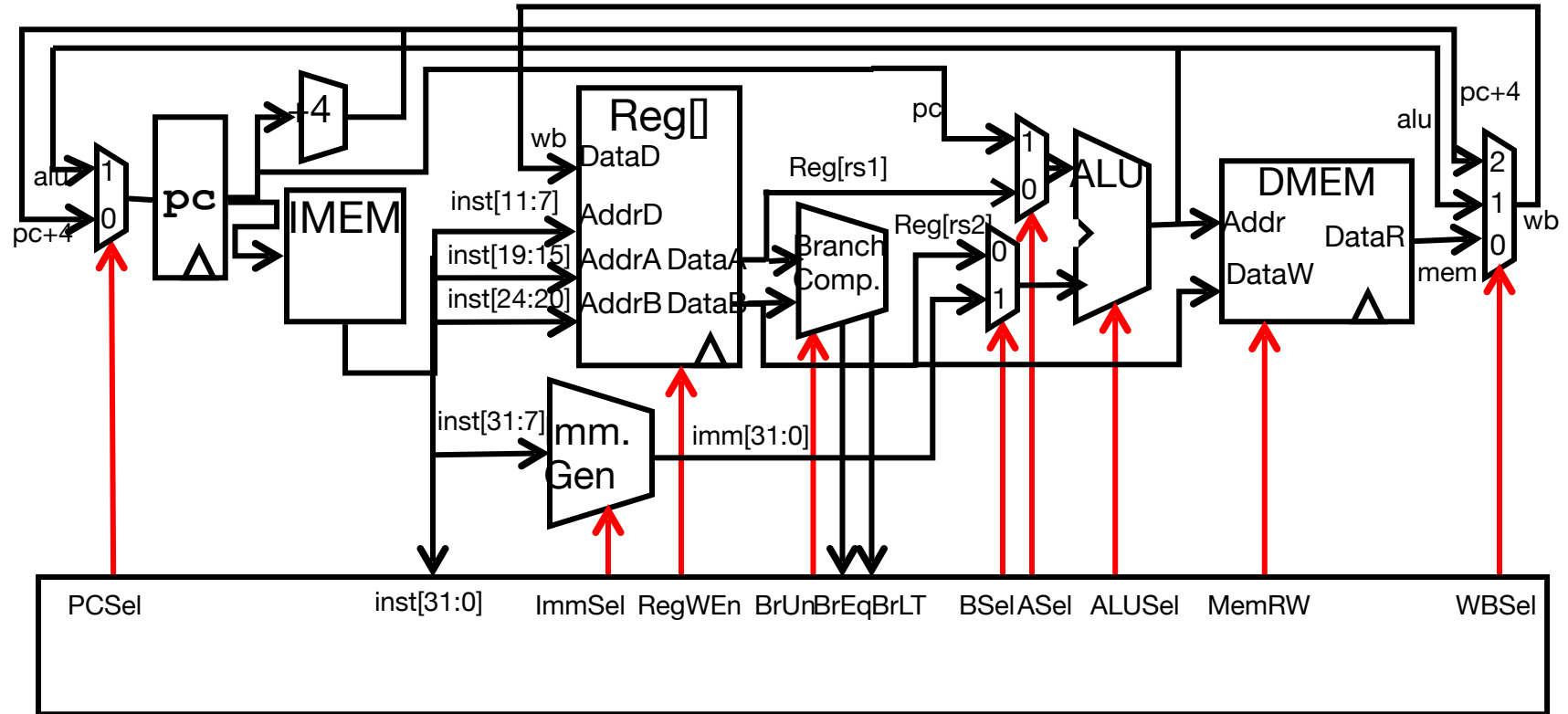


Implementing jal Instruction

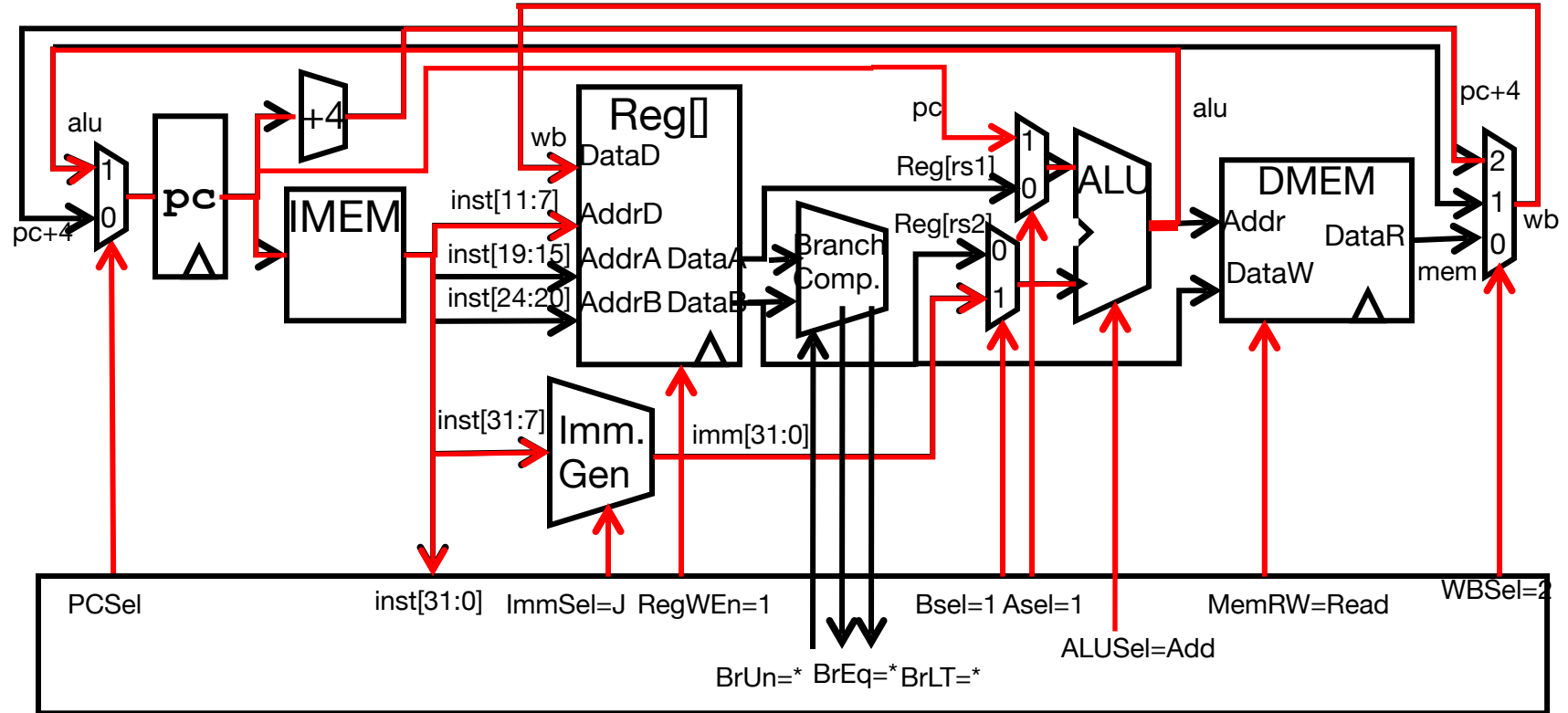


- JAL saves PC+4 in Reg[rd] (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
 - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

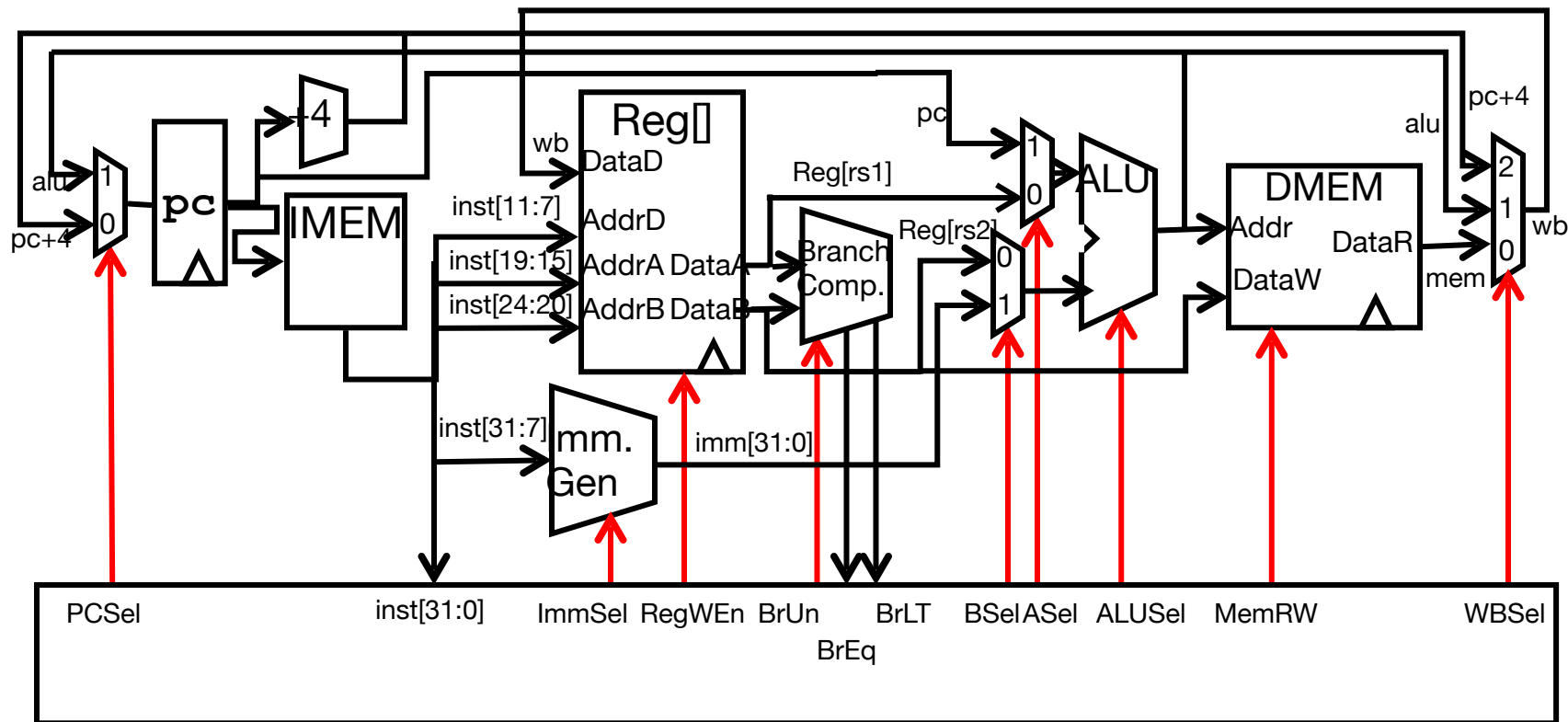
Adding jal to datapath



Adding jal to datapath



Single-Cycle RISC-V RV32I Datapath



And in Conclusion, ...

- Universal datapath
 - Capable of executing all RISC-V instructions in one cycle each
 - datapath is the “union” of all the units used by all the instructions. Muxes provide the options.
 - 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