EECS 151/251A
Spring 2019
Digital Design and Integrated Circuits

Instructor:
John Wawrzynek

Lecture 13

Project Introduction

- You will design and optimize a RISC-V processor
- Phase 1: Design and demonstrate a processor
- Phase 2:
  - ASIC Lab – implement cache memory and generate complete chip layout
  - FPGA Lab – Add video display and graphics accelerator

Today discuss how to design the processor

What is RISC-V?

- Fifth generation of RISC design from UC Berkeley
- A high-quality, license-free, royalty-free RISC ISA specification
- Experiencing rapid uptake in both industry and academia
- Supported by growing shared software ecosystem
- Appropriate for all levels of computing system, from micro-controllers to supercomputers
  - 32-bit, 64-bit, and 128-bit variants (we're using 32-bit in class, textbook uses 64-bit)
- Standard maintained by non-profit RISC-V Foundation

https://riscv.org/specifications/

Foundation Members (60+)

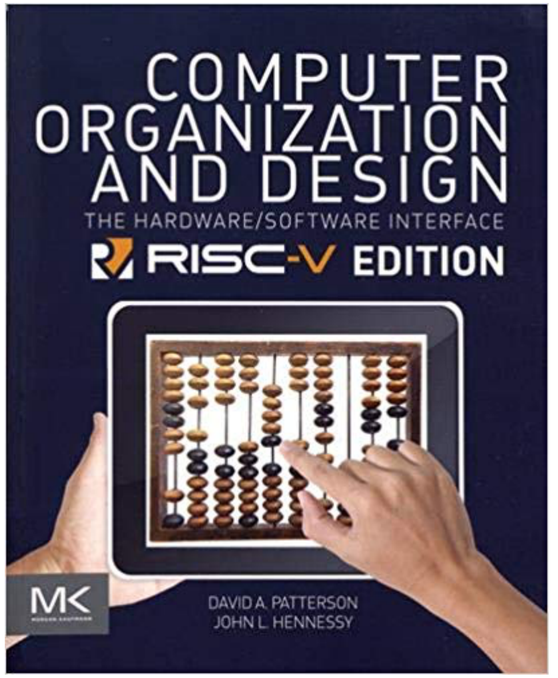
Platinum:

Gold, Silver, Auditors:

Rumble
Development

Instruction Set Architecture (ISA)

- Job of a CPU (*Central Processing Unit*, aka *Core*): execute *instructions*
- Instructions: CPU's primitives operations
  - Instructions performed one after another in sequence
  - Each instruction does a small amount of work (a tiny part of a larger program).
  - Each instruction has an *operation* applied to *operands,*
  - and might be used change the sequence of instruction.
- CPUs belong to "families," each implementing its own set of instructions
- CPU's particular set of instructions implements an *Instruction Set Architecture* (*ISA*)
  - Examples: ARM, Intel x86, MIPS, RISC-V, IBM/Motorola PowerPC (old Mac), Intel IA64, ...



If you need more info on processor organization.

Complete RV32I ISA

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	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]		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
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]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI

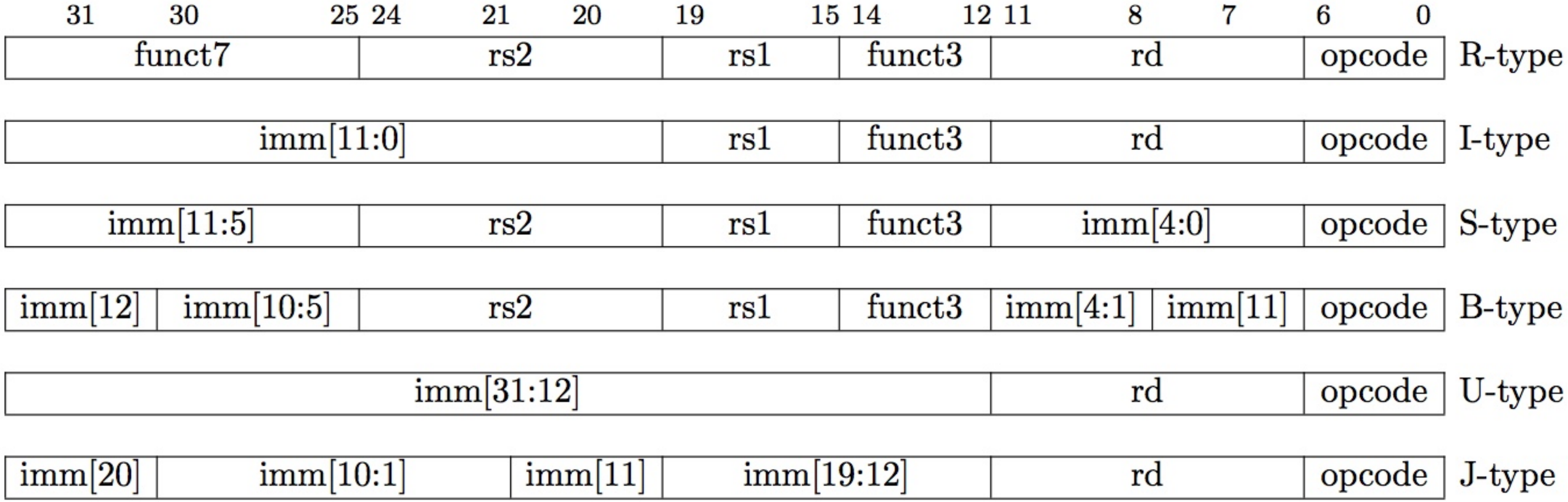
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	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
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRRW *
csr			rs1	010	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 EECS151/251A

** implemented in the ASIC project*

Summary of RISC-V Instruction Formats

Binary encoding of machine instructions. Note the common fields.



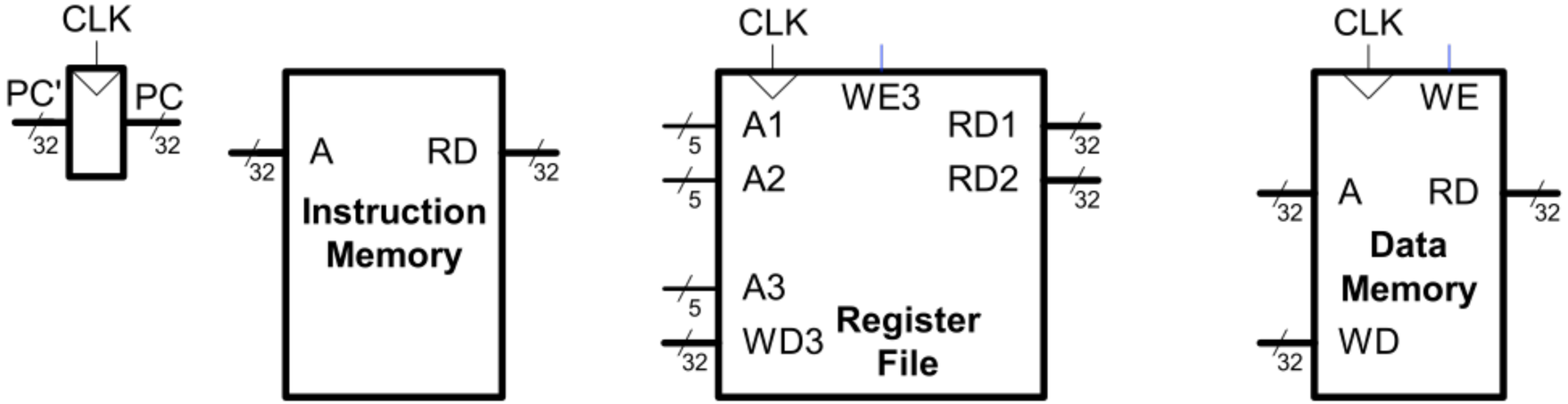
“State” Required by RV32I ISA

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

RISC-V State Elements

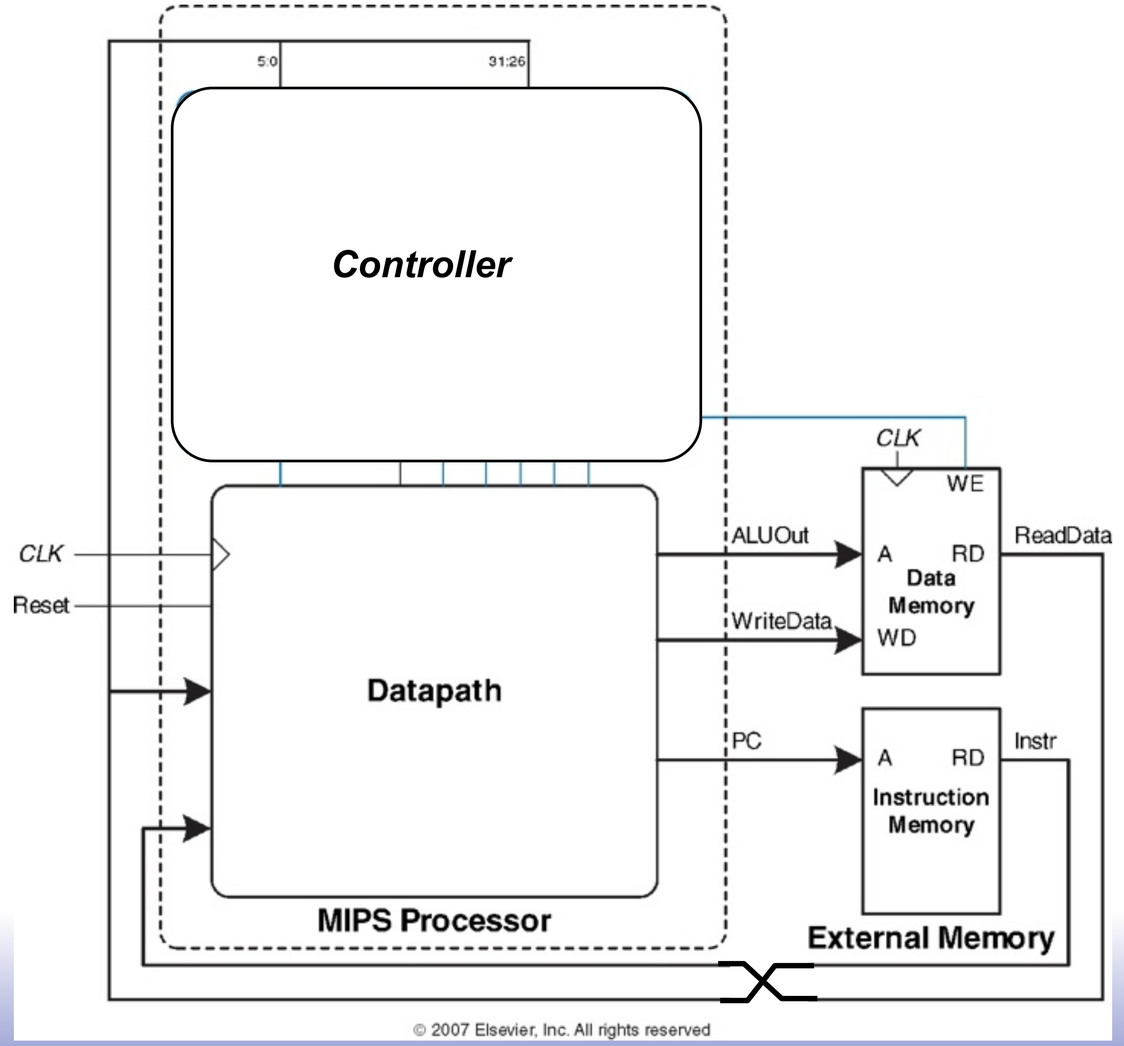
- State encodes everything about the execution status of a processor:
  - **`PC`** register
  - 32 registers
  - Memory



Note: for these state elements, clock is used for write but not for read (asynchronous read, synchronous write).

RISC-V Microarchitecture Oganization

Datapath + Controller + External Memory



© 2007 Elsevier, Inc. All rights reserved

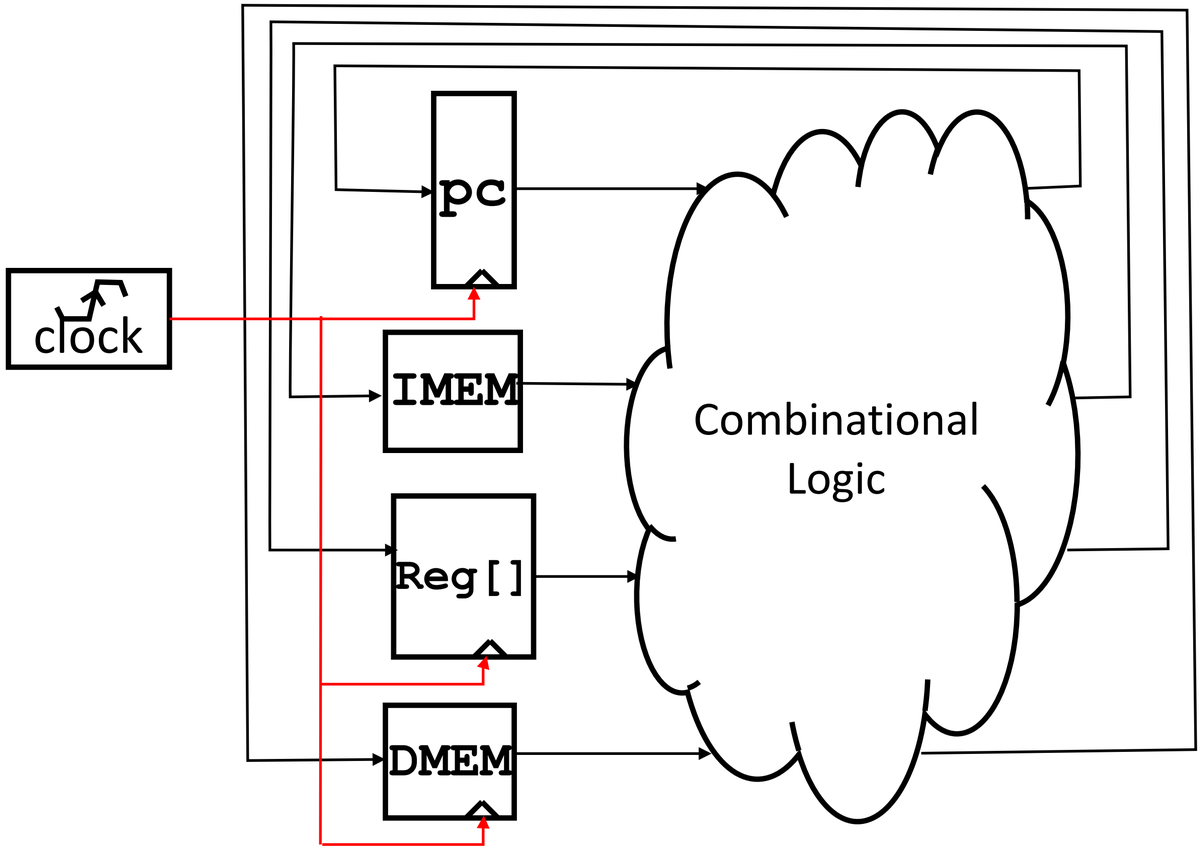
Microarchitecture

Multiple implementations for a single architecture:

- *Single-cycle*
  - Each instruction executes in a single clock cycle.
- *Multicycle*
  - Each instruction is broken up into a series of shorter steps with one step per clock cycle.
- *Pipelined (variant on "multicycle")*
  - Each instruction is broken up into a series of steps with one step per clock cycle
  - Multiple instructions execute at once by overlapping in time.
- *Superscalar*
  - Multiple functional units to execute multiple instructions at the same time
- *Out of order...*
  - Hey, who says we have to follow the program exactly....

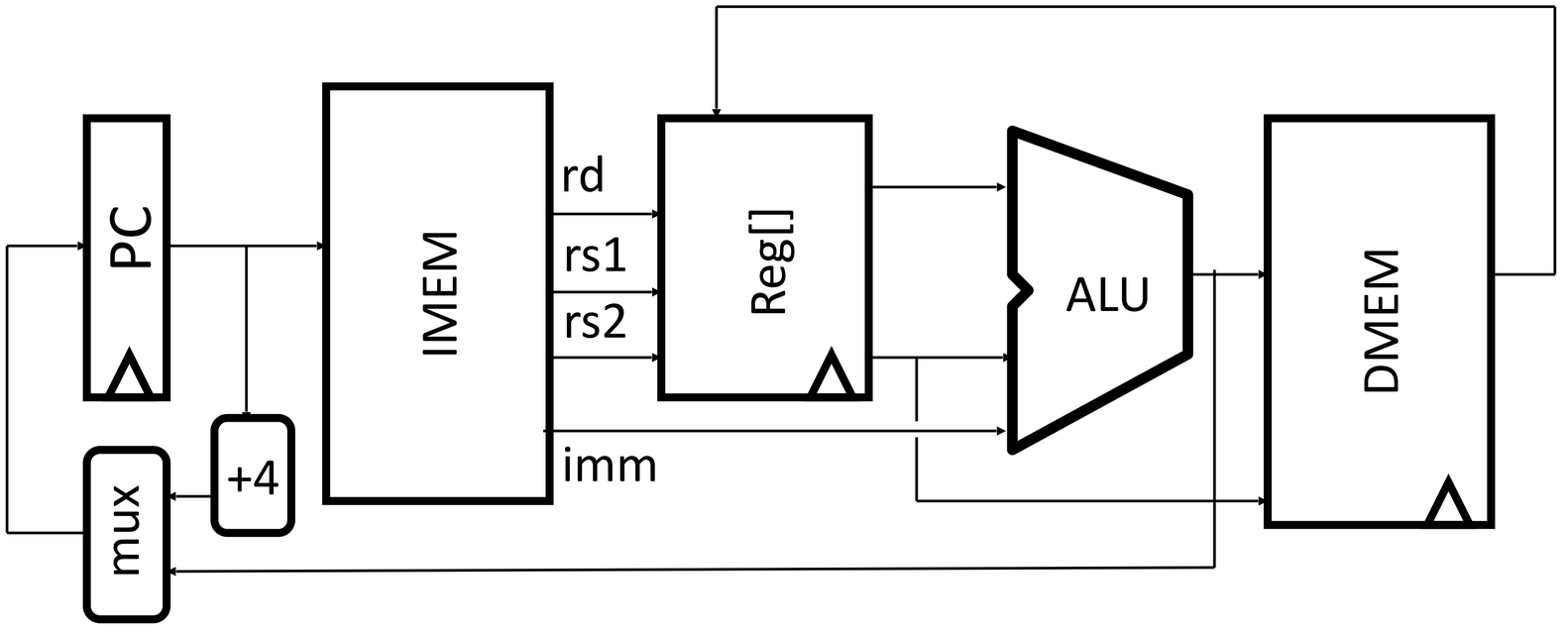
First Design: One-Instruction-Per-Cycle RISC-V Machine

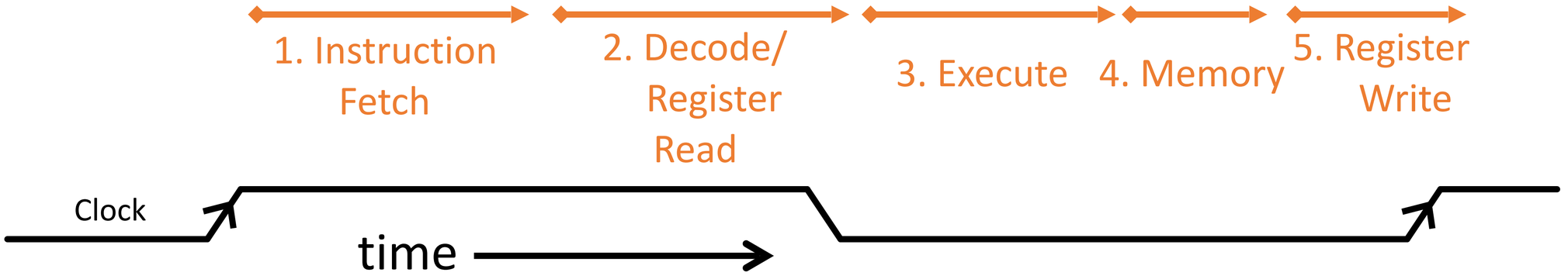
On every tick of the clock, the computer executes one instruction



1. Current state outputs drive the inputs to the combinational logic, whose outputs settles 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 (next instruction)

Basic Phases of Instruction Execution





Implementing the **`add`** instruction

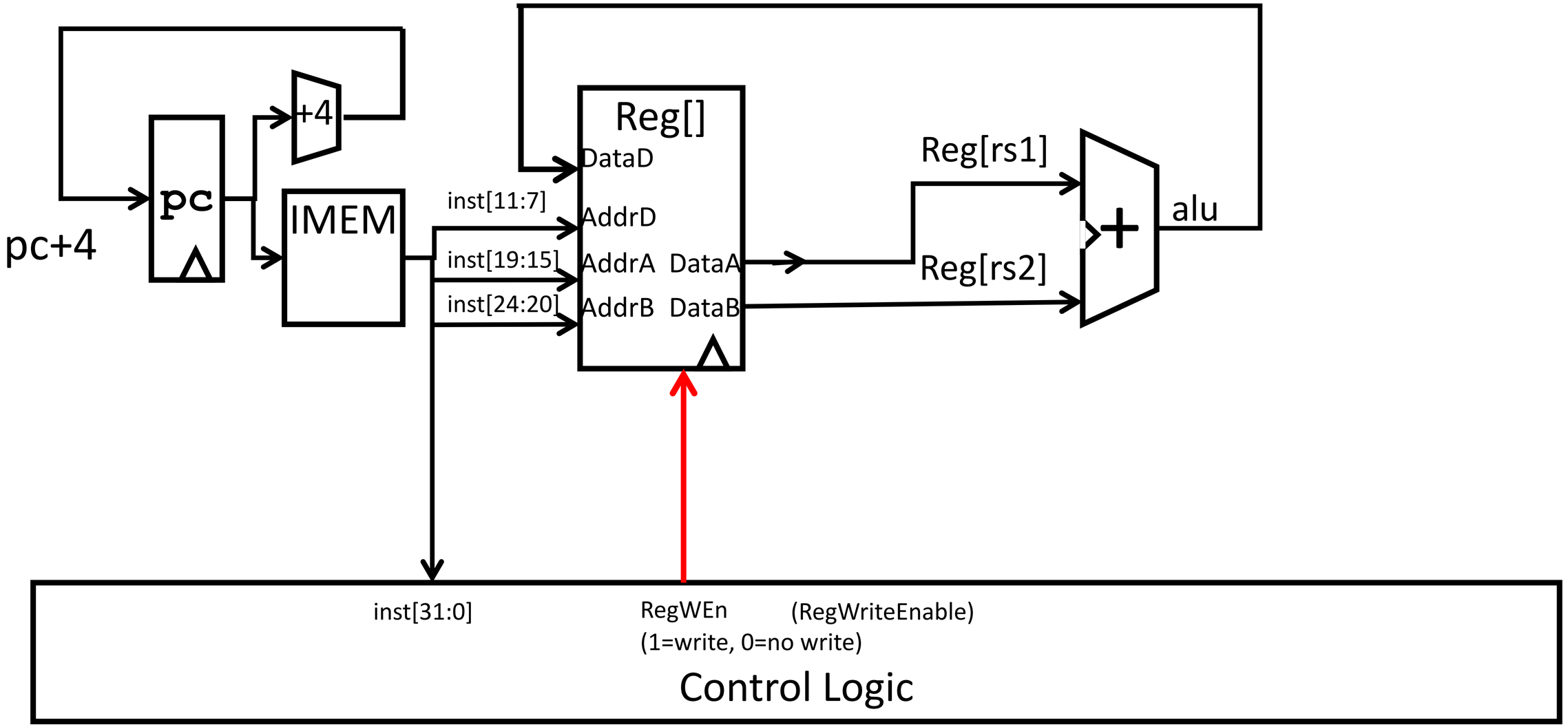
0000000	rs2	rs1	000	rd	0110011	ADD

`add rd, rs1, rs2`

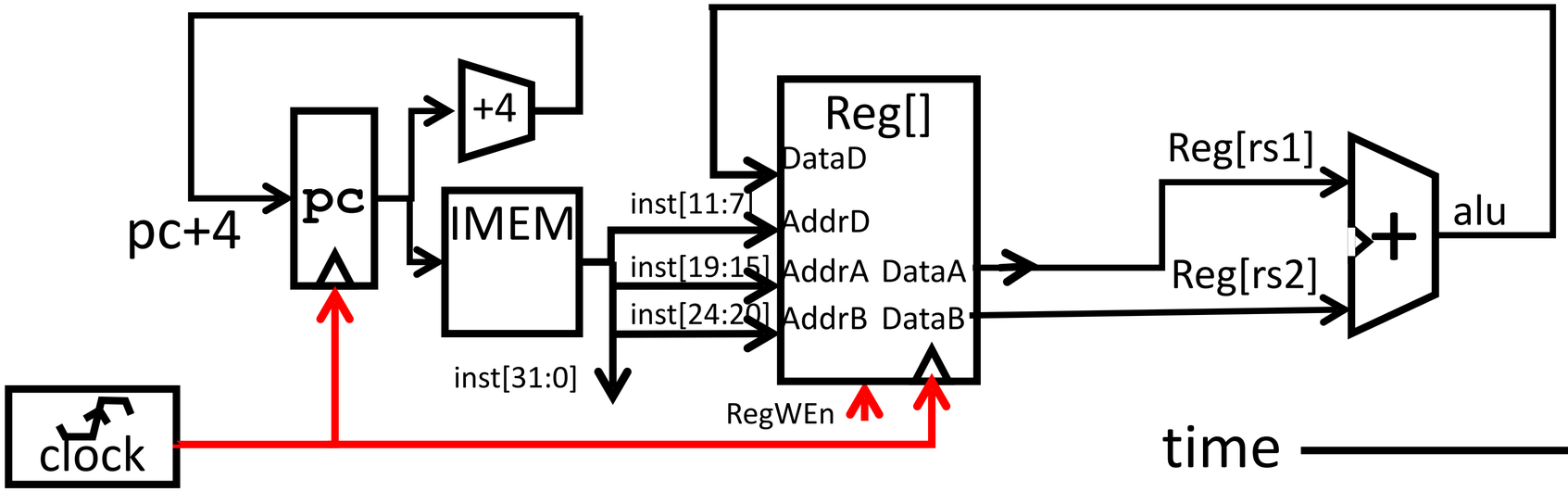
- Instruction makes two changes to machine's state:

```
Reg[rd] = Reg[rs1] + Reg[rs2]
PC = PC + 4
```

Datapath for **add**



Timing Diagram for **add**



time

Clock			
PC	1000	1004	
PC+4	1004	1008	
inst[31:0]	`add x1,x2,x3`	`add x6,x7,x9`	
Reg[rs1]	`Reg[2]`	`Reg[7]`	
Reg[rs2]	`Reg[3]`	`Reg[9]`	
alu	`Reg[2]+Reg[3]`	`Reg[7]+Reg[9]`	
Reg[1]	???	`Reg[2]+Reg[3]`	

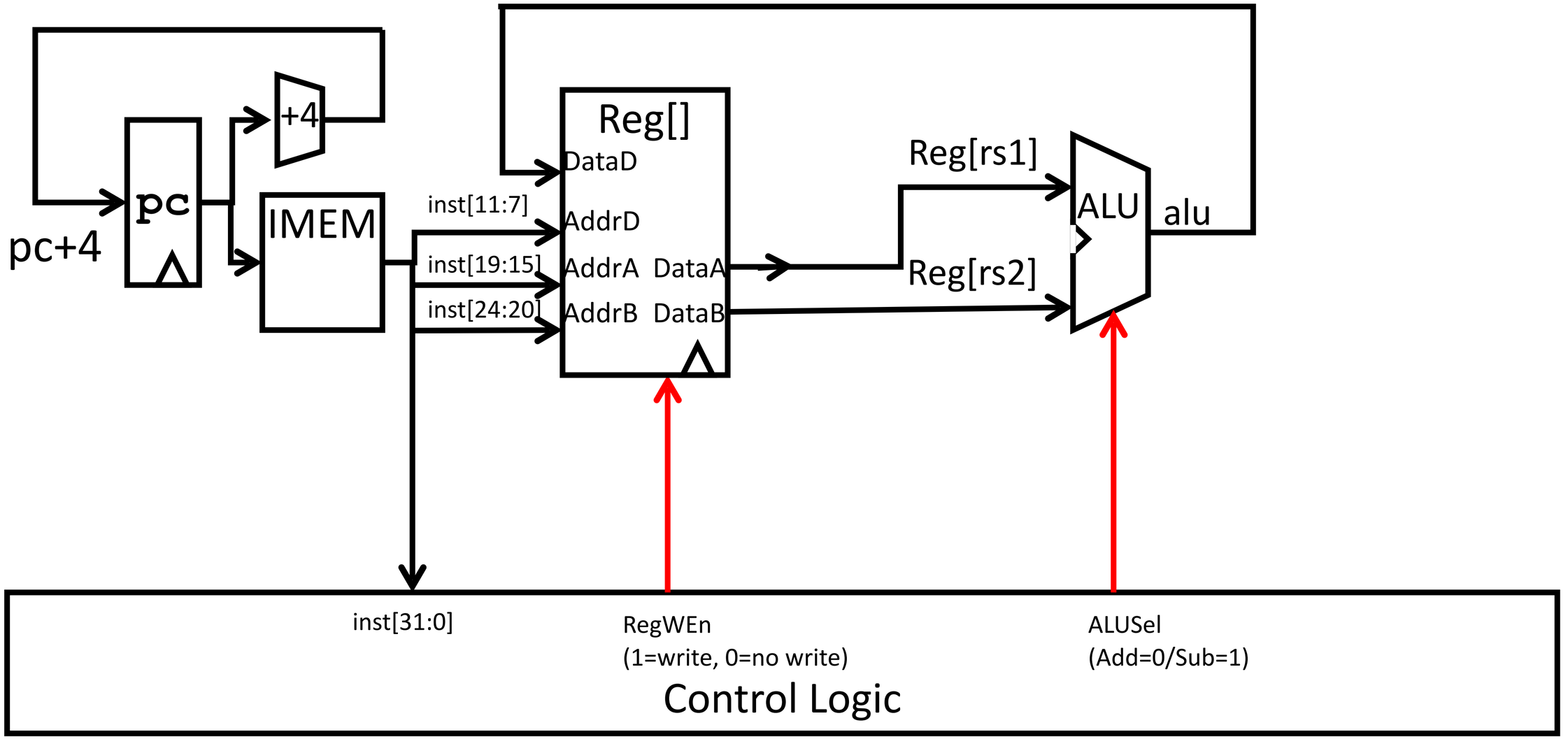
Implementing the **`sub`** instruction

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

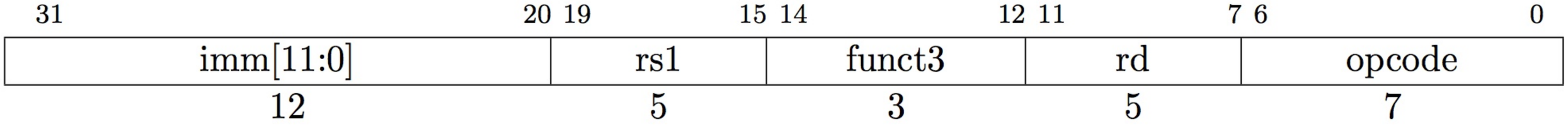
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110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

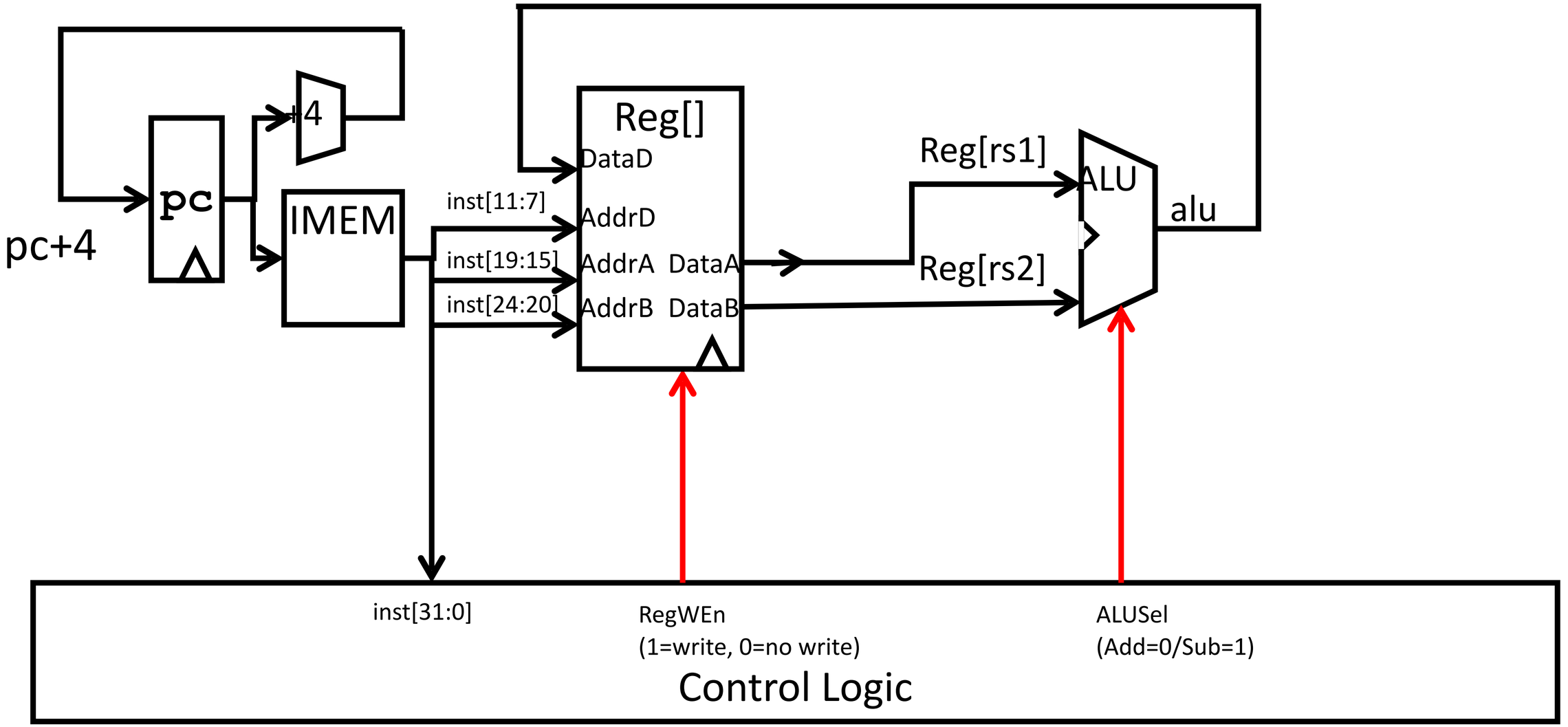
- RISC-V Assembly Instruction:

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

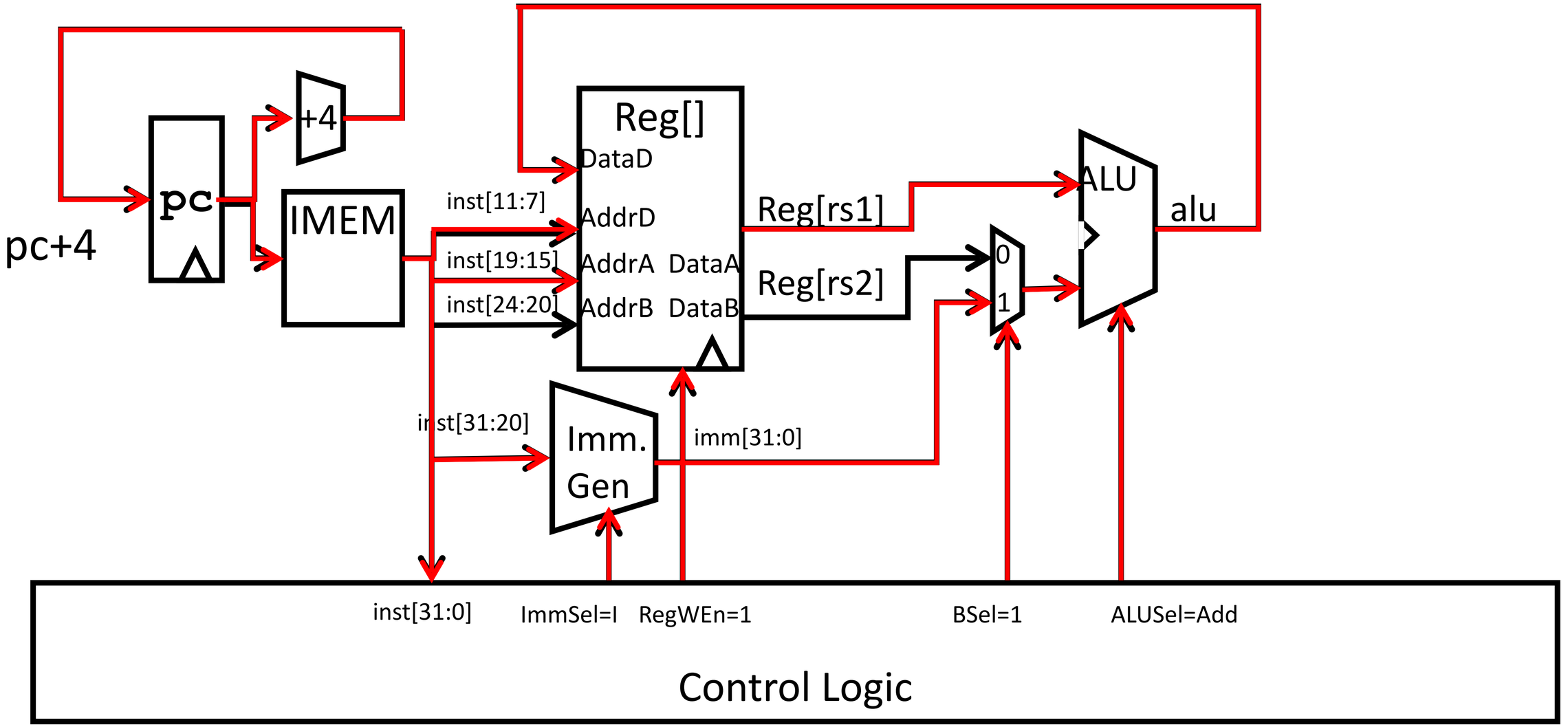


111111001110	00001	000	01111	0010011
imm=-50	rs1=1	ADD	rd=15	OP-Imm

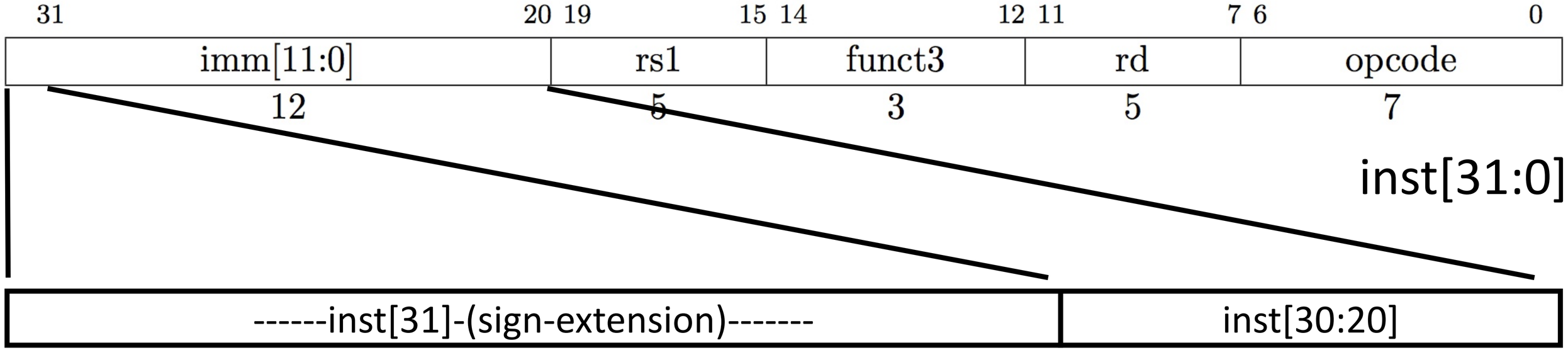
Datapath for **add/sub**

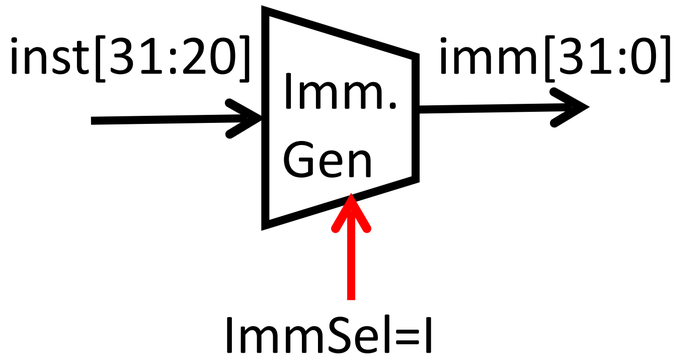


Adding **addi** to datapath



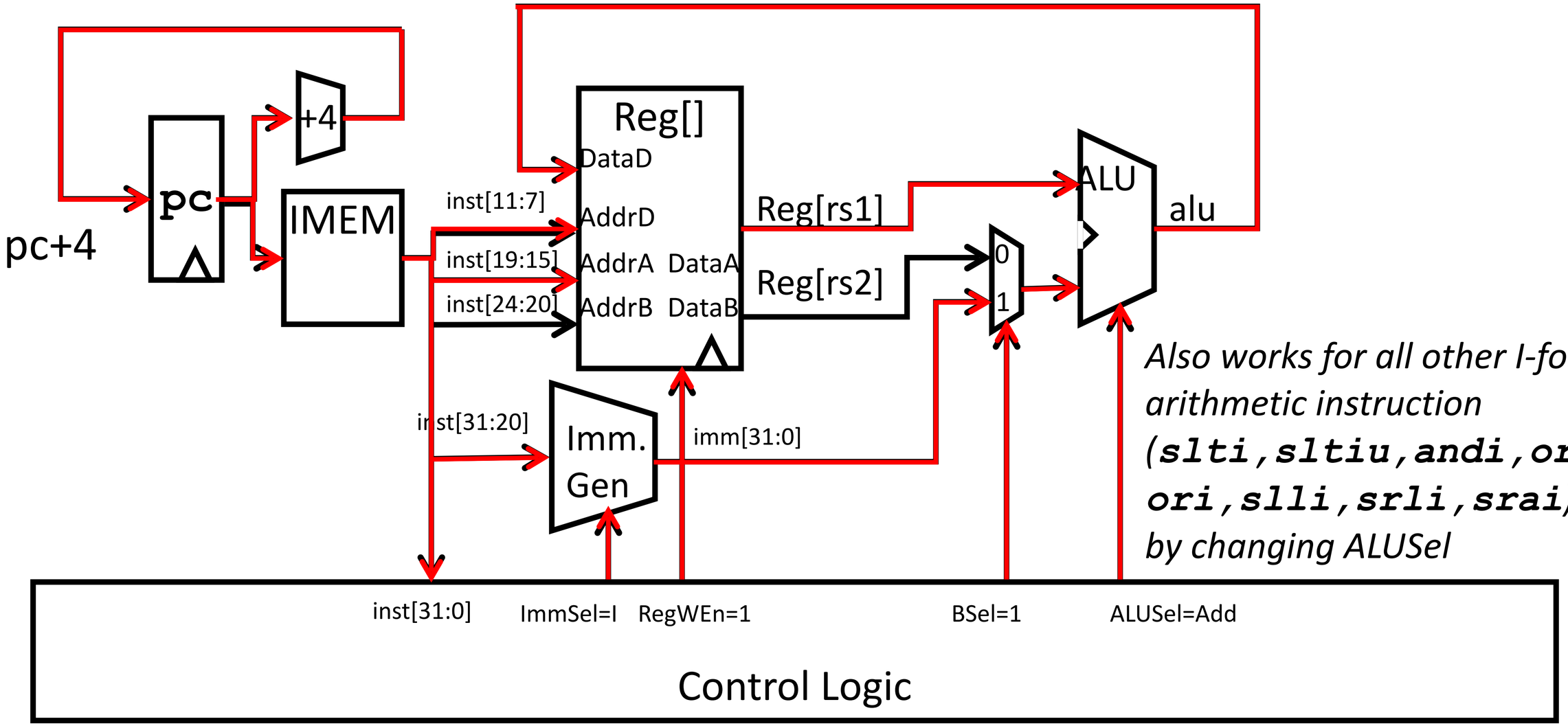
I-Format immediates





- 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])

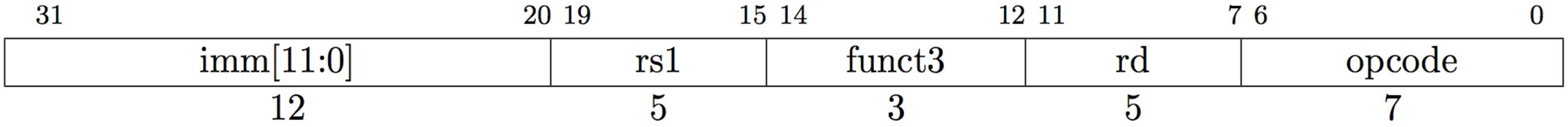
Adding **`addi`** to datapath



Implementing Load Word instruction

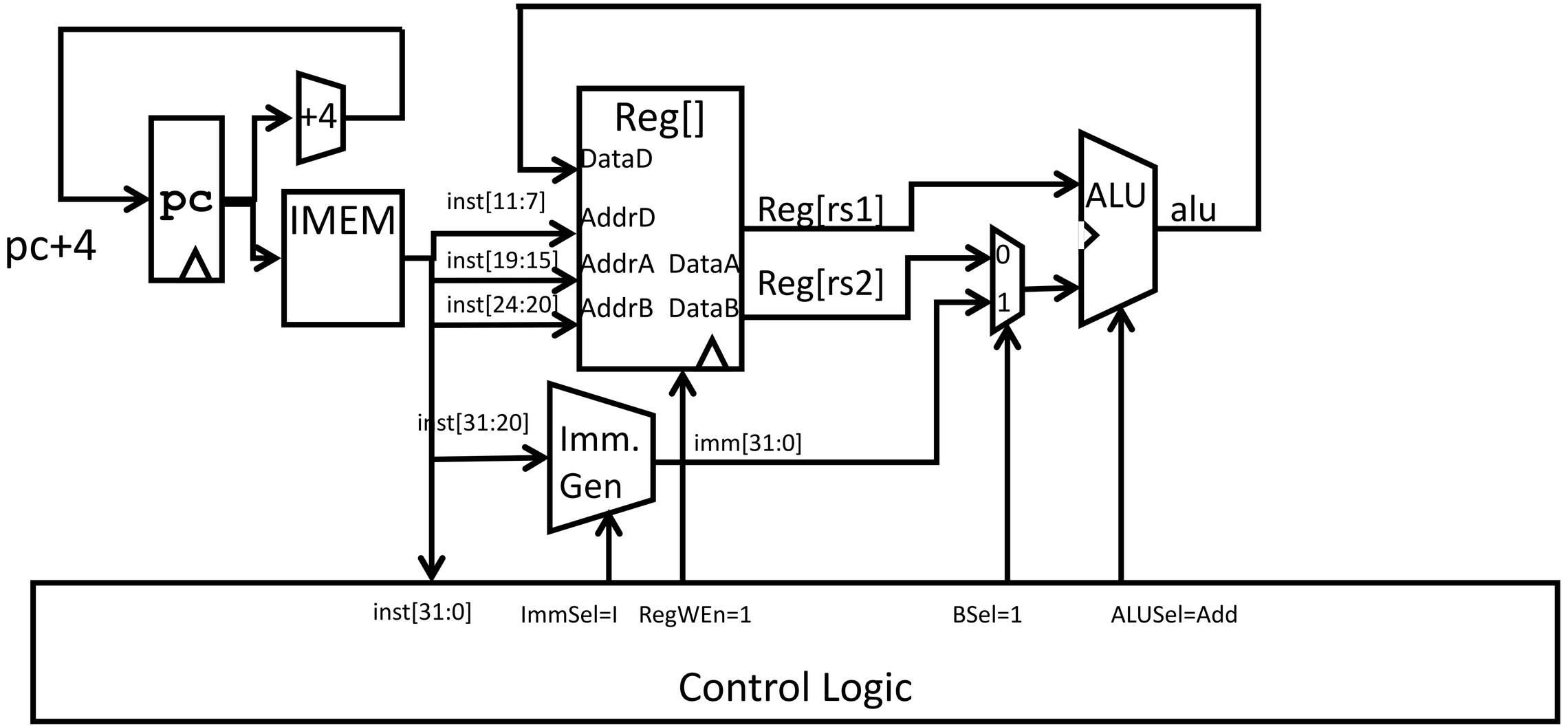
- RISC-V Assembly Instruction:

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

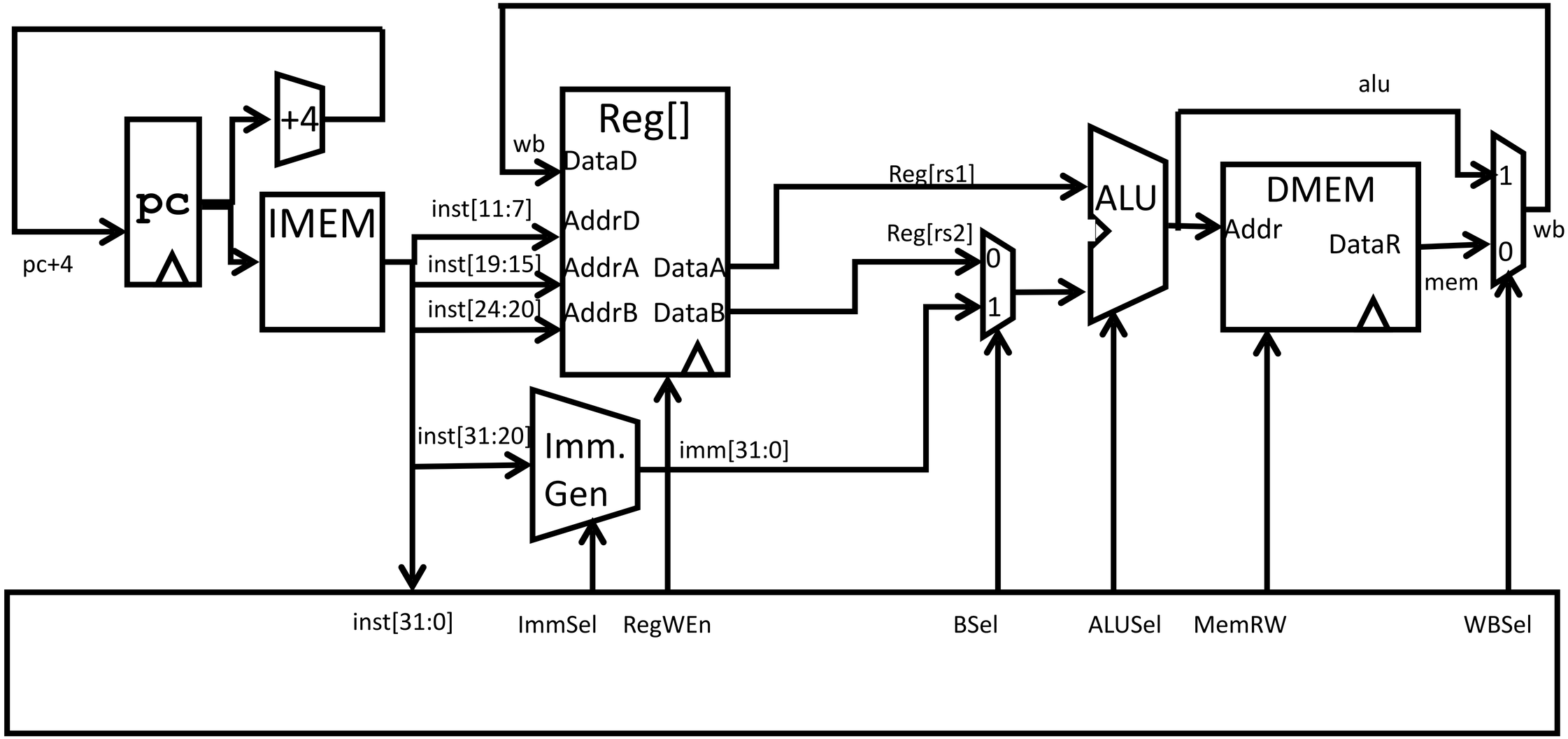


000000001000	00010	010	01110	0000011
imm=+8	rs1=2	LW	rd=14	LOAD

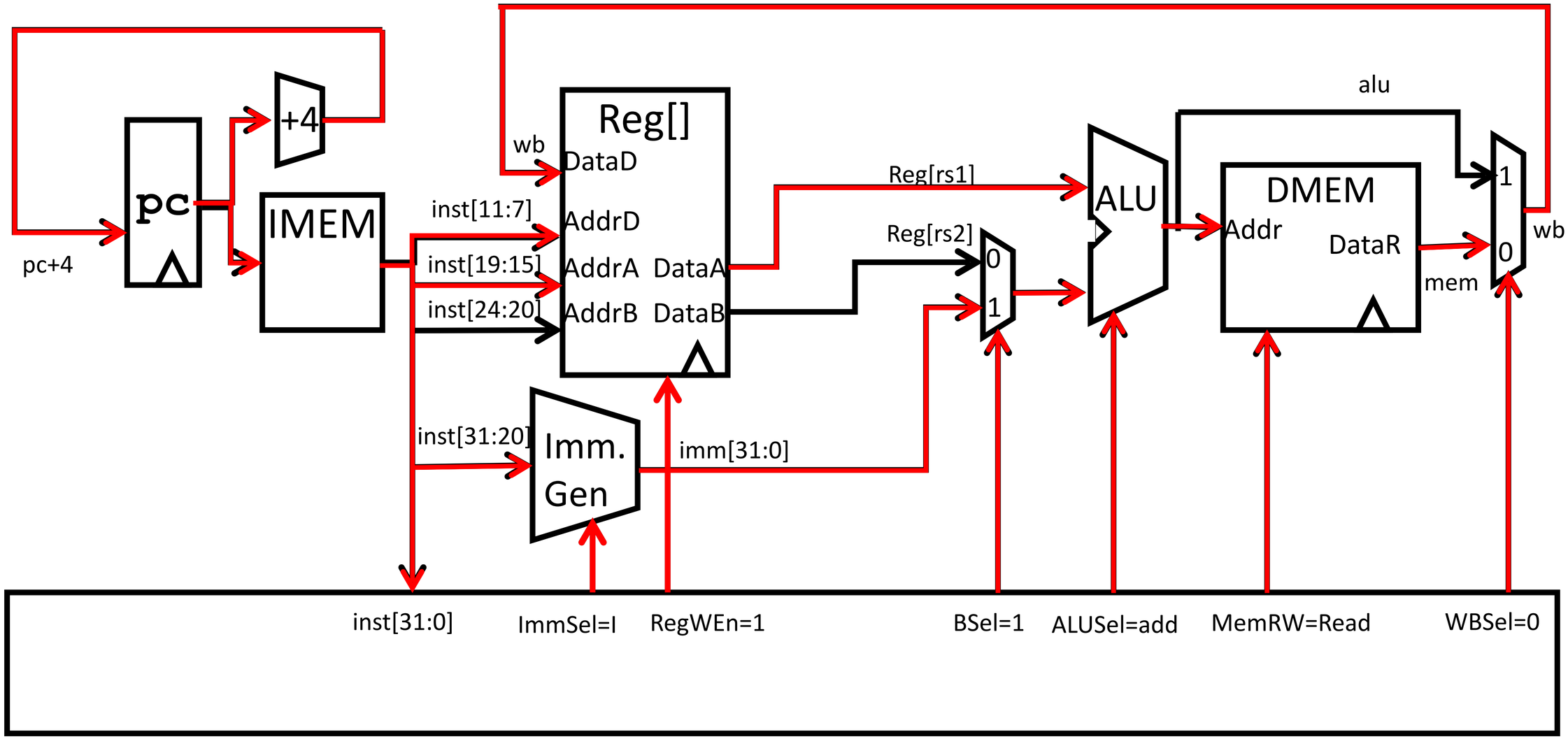
Adding `addi` to datapath



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

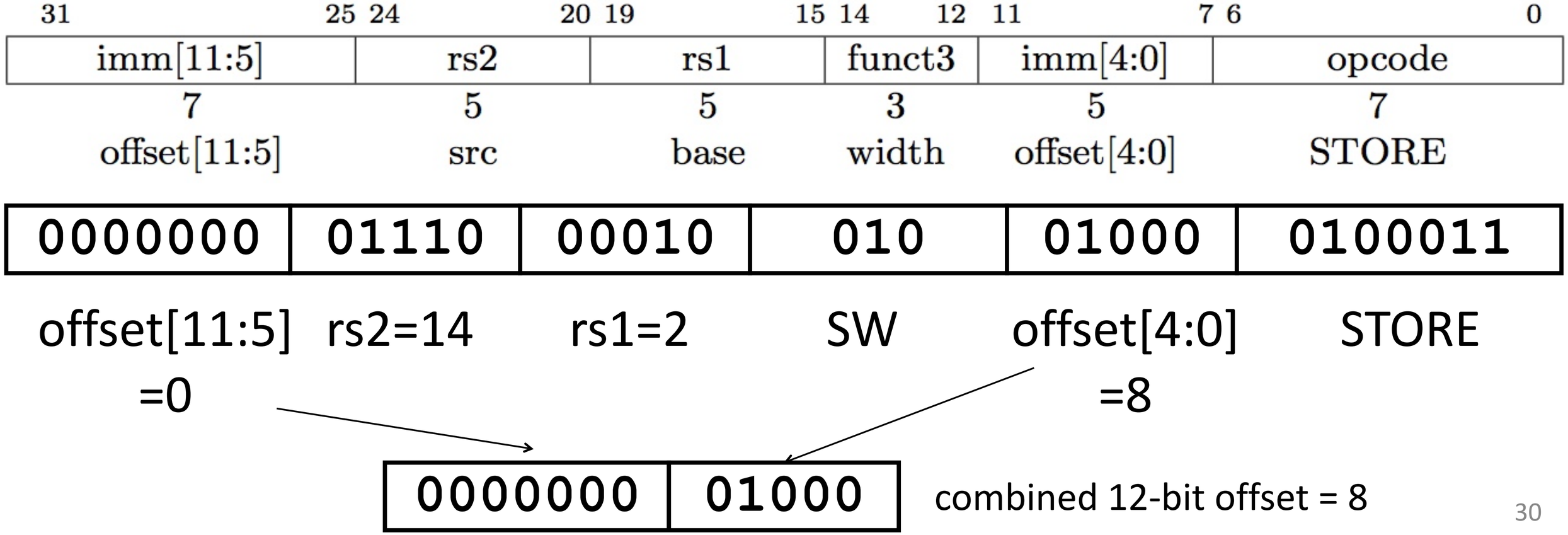
funct3 field encodes size and signedness of load data

- 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.

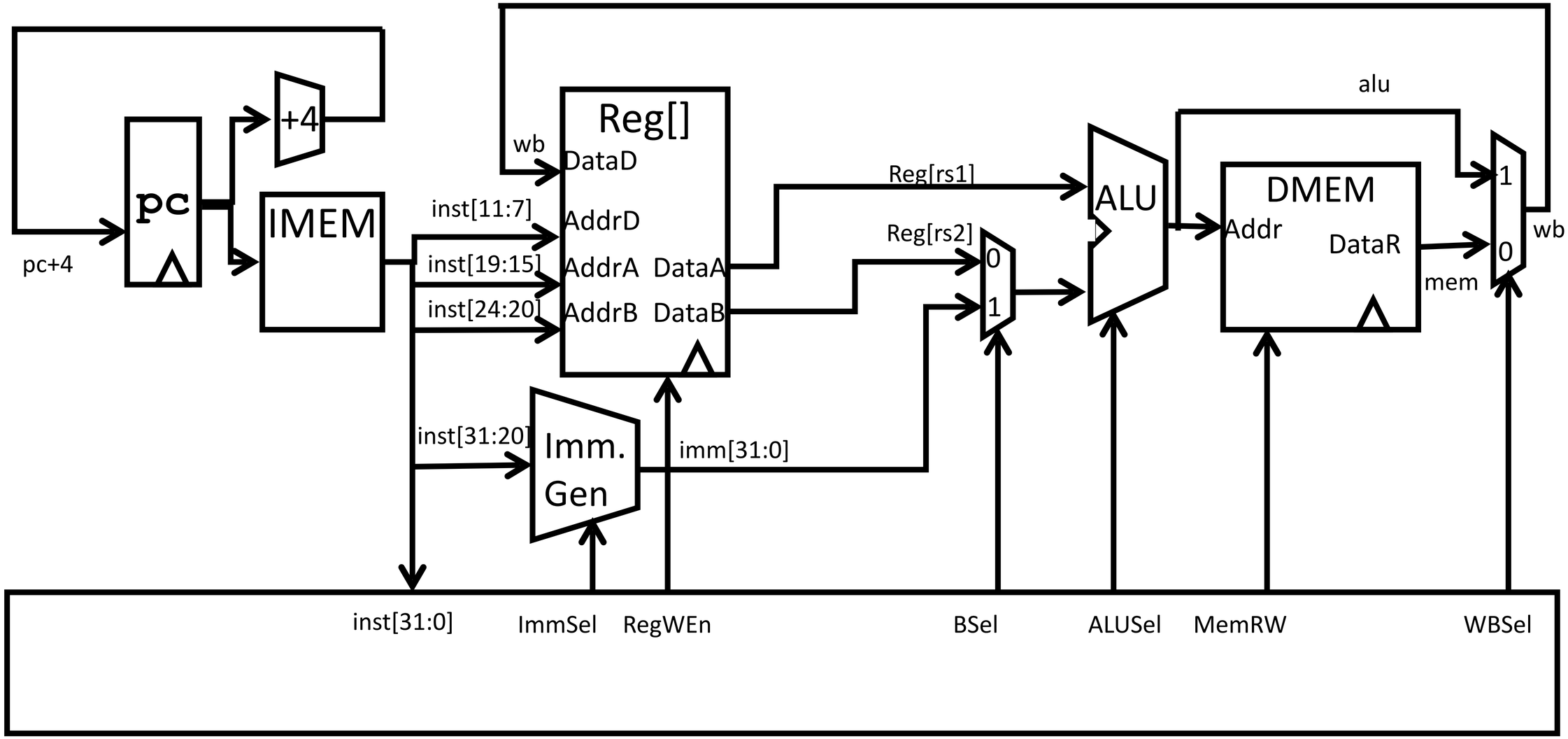
Implementing Store Word instruction

- RISC-V Assembly Instruction:

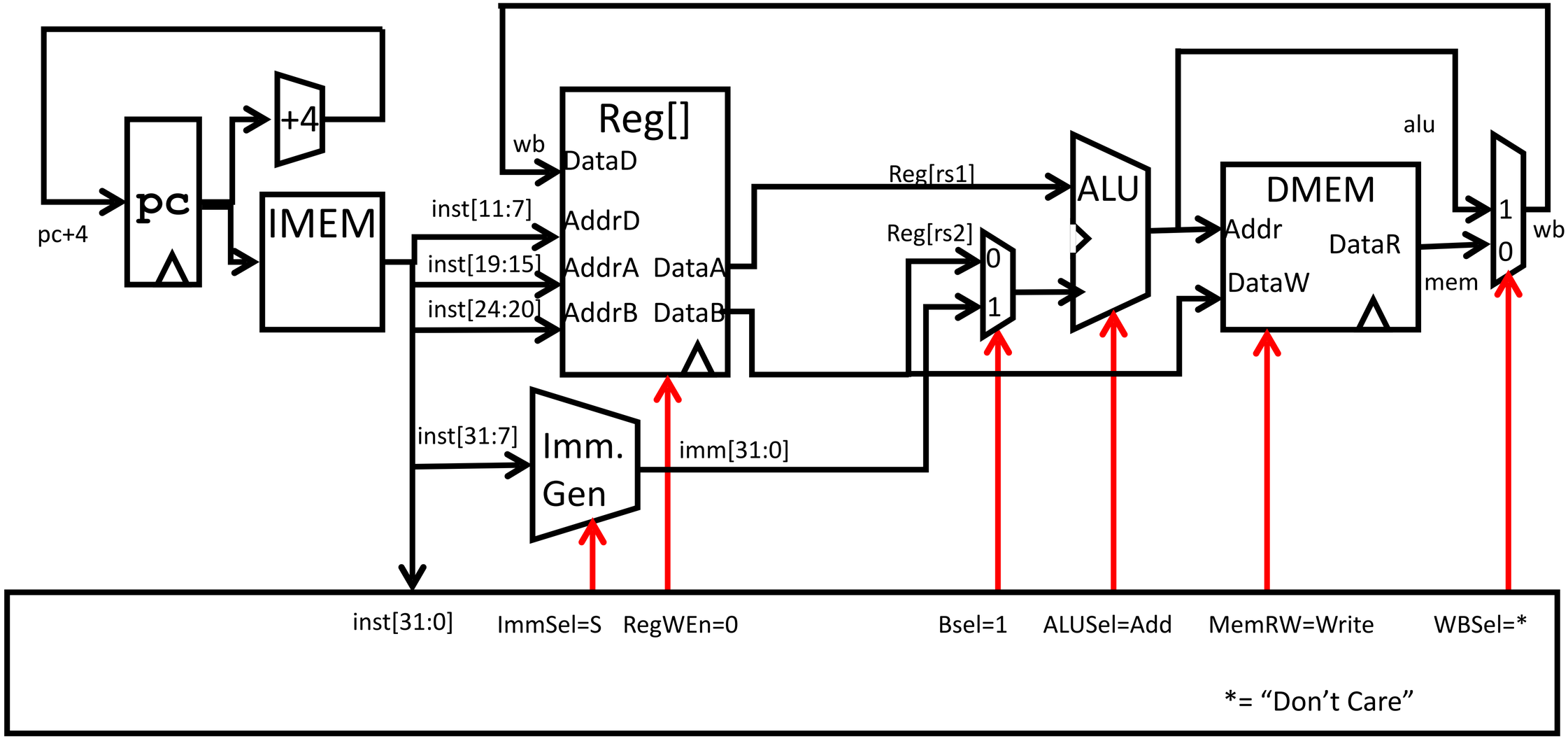
```
sw x14, 8(x2)
```



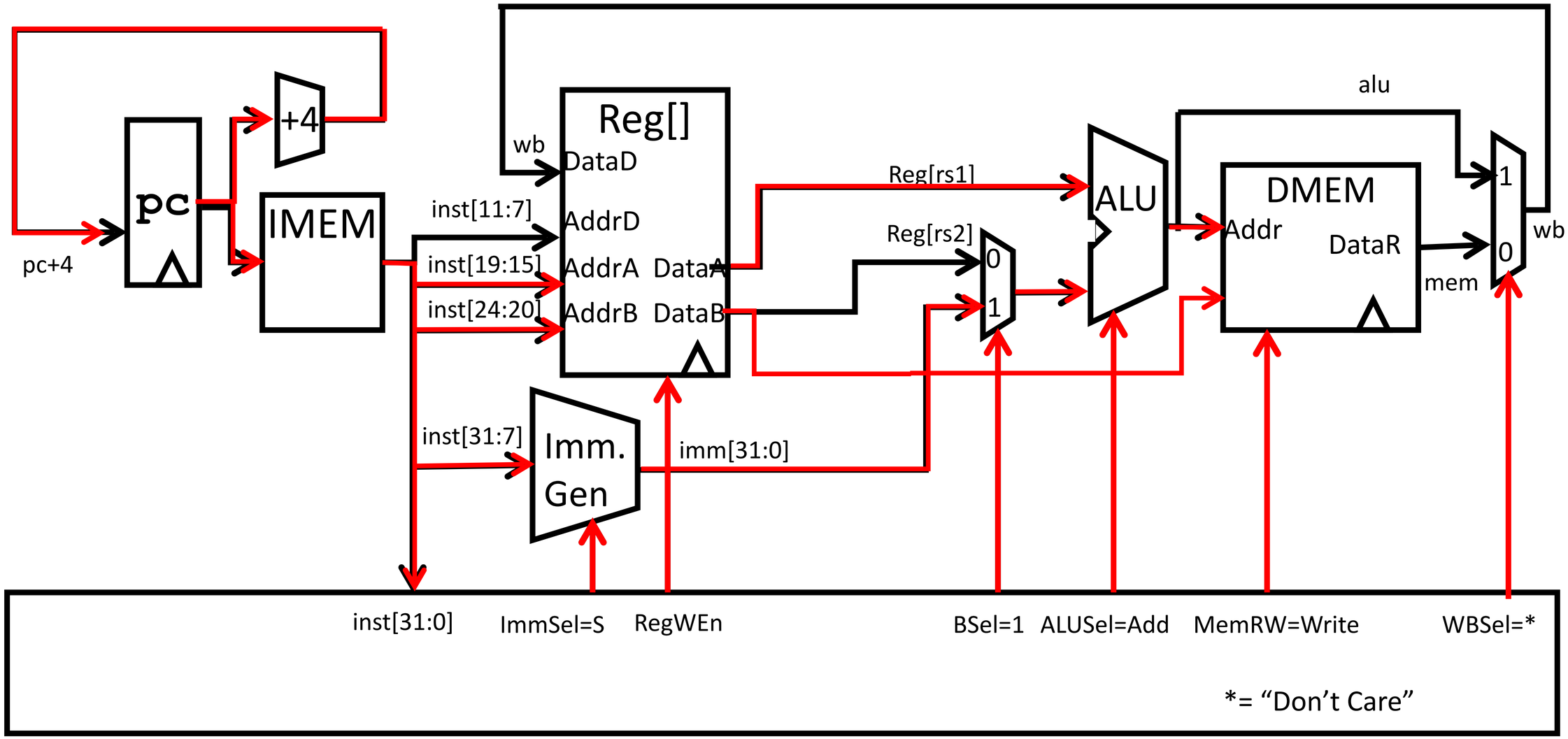
Adding `lw` to datapath



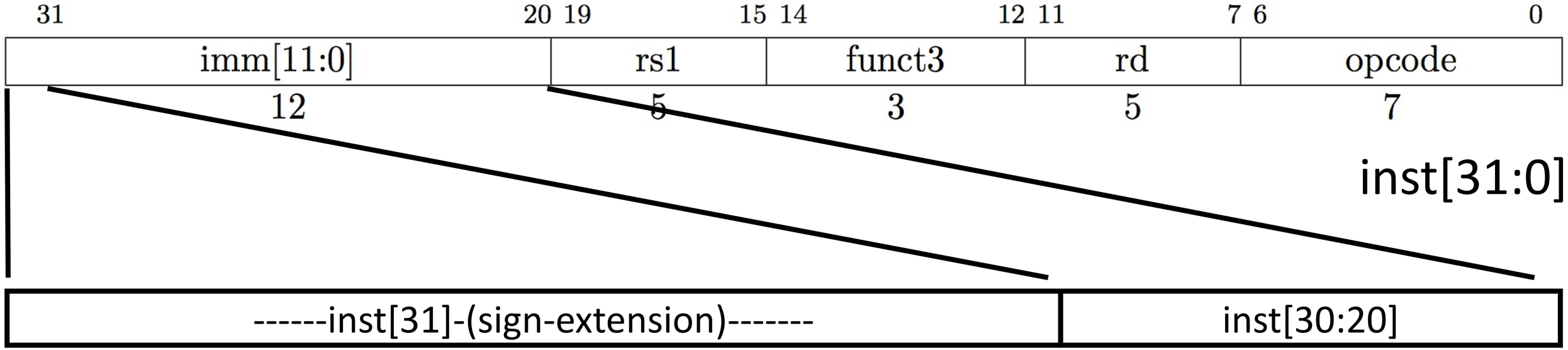
Adding **sw** to datapath

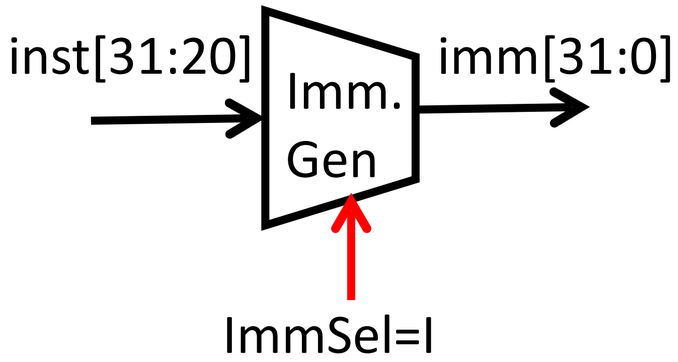


Adding `sw` to datapath



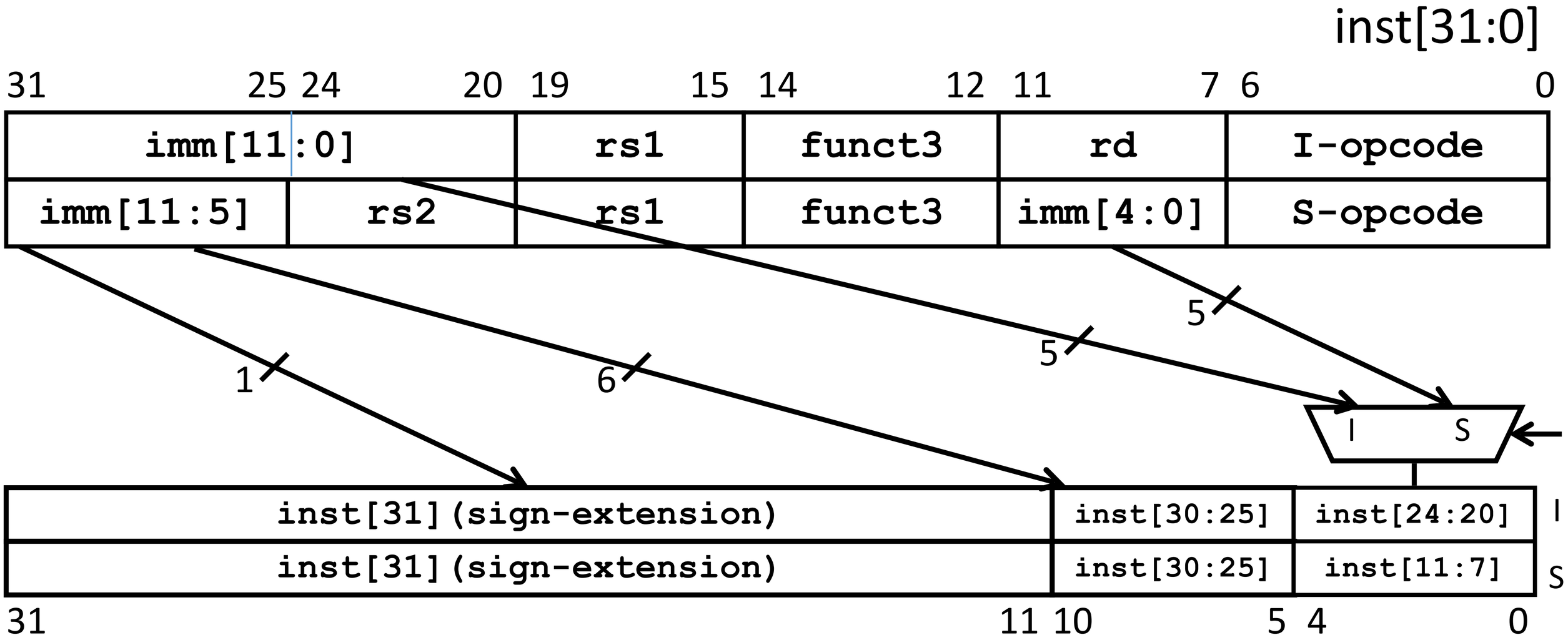
I-Format immediates





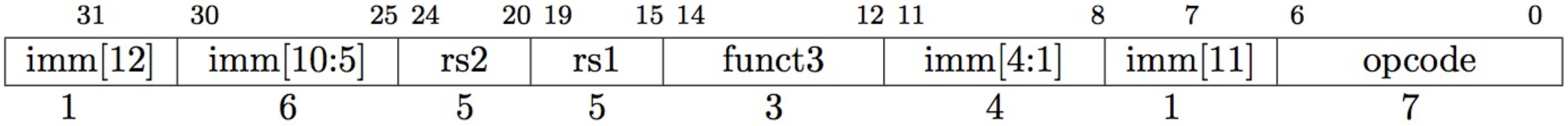
- High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
- Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])

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

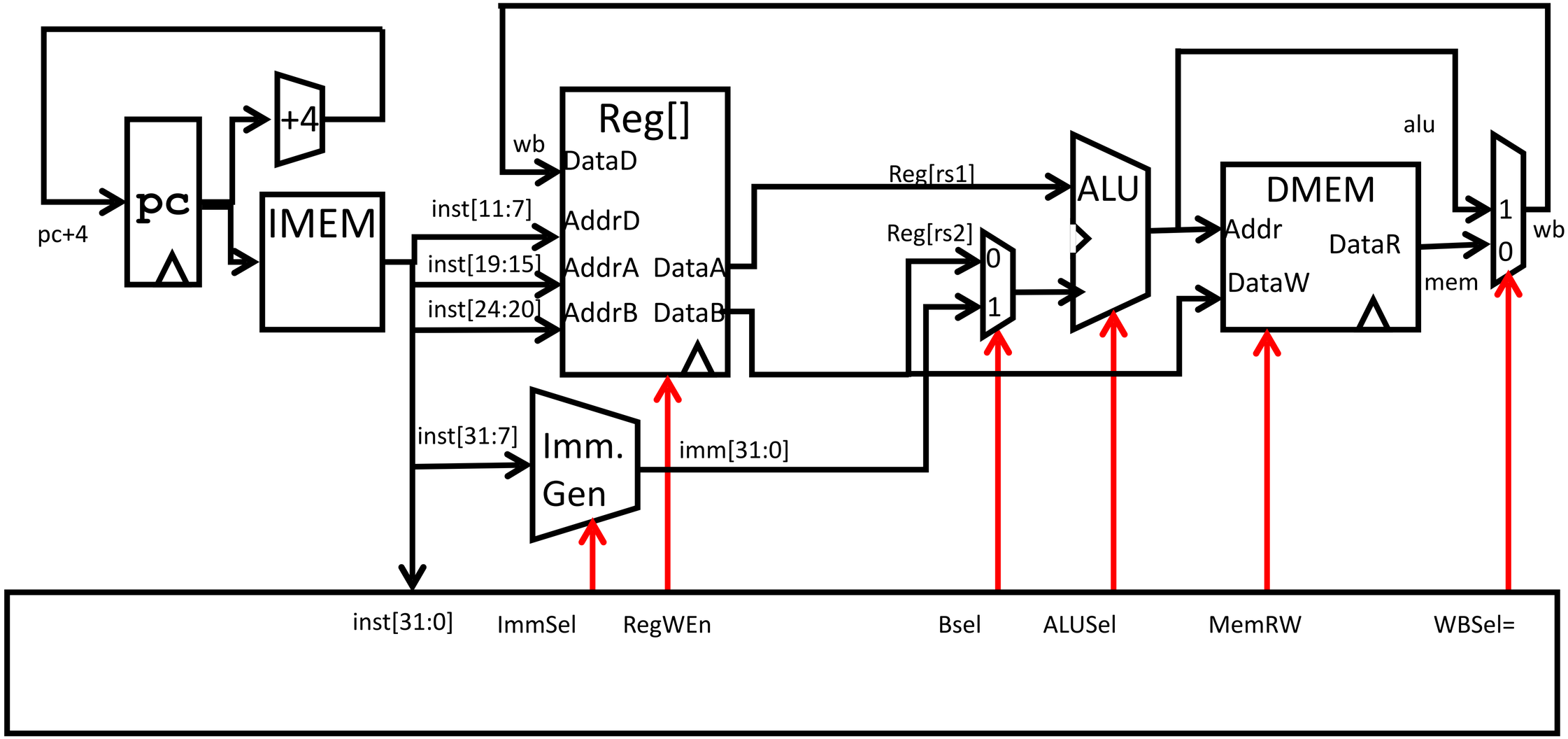
Implementing Branches



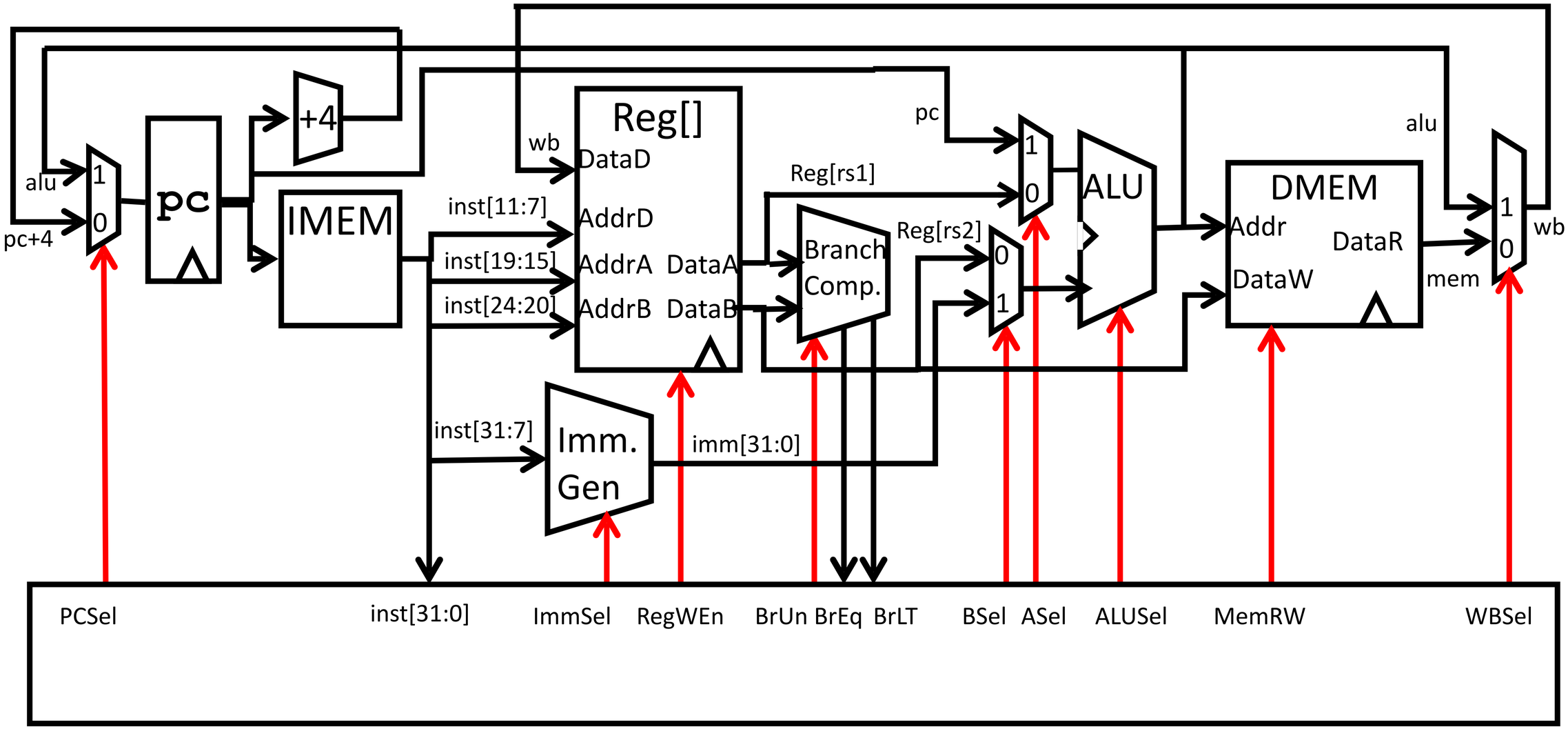
Example: ***if rs1 = rs2 then pc ← pc + offset***

- 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)

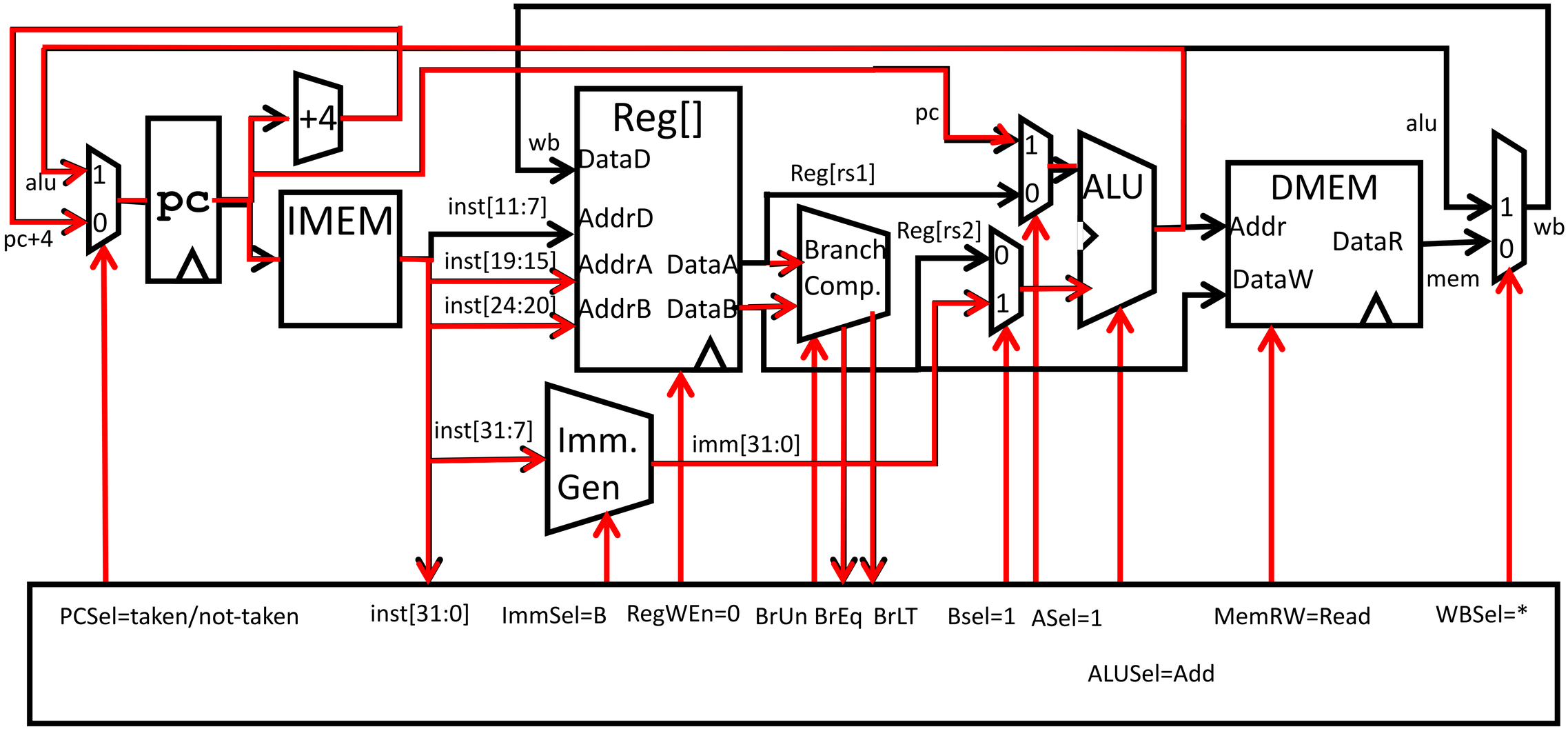
Adding **sw** to datapath



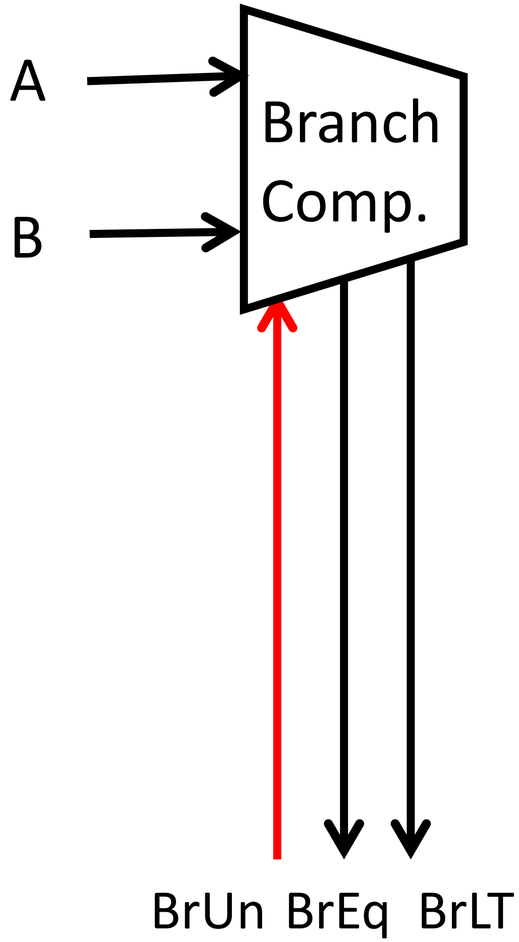
Adding branches to datapath



Adding branches to datapath

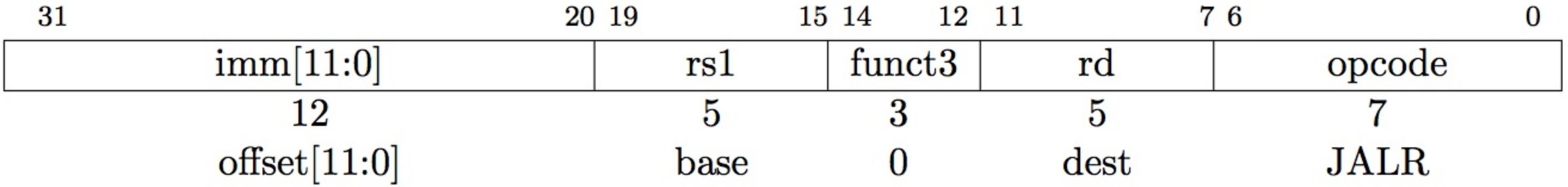


Branch Comparator



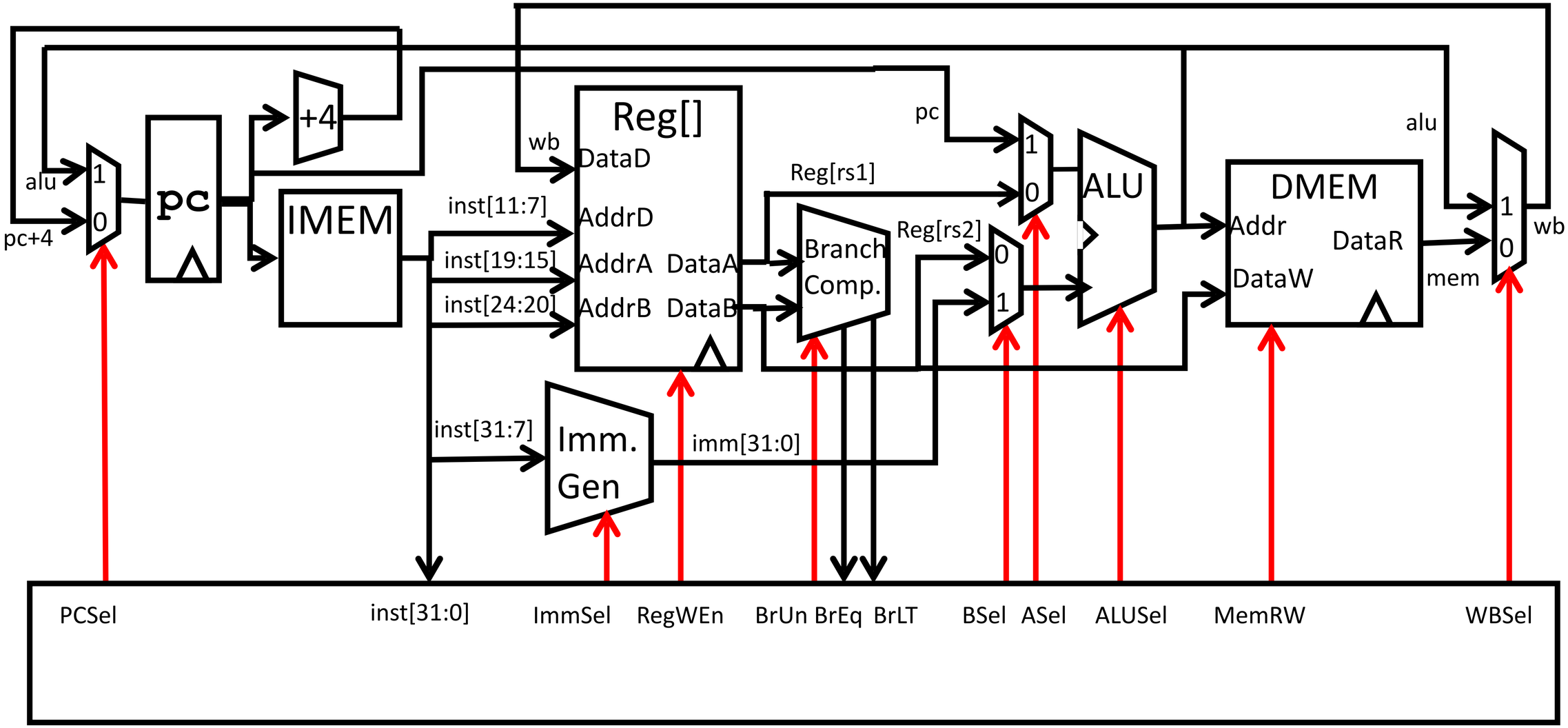
- BrEq = 1, if A=B
- BrLT = 1, if A < B
- BrUn =1 selects unsigned comparison for BrLT, 0=signed
- BGE branch: A >= B, if !(A<B)

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

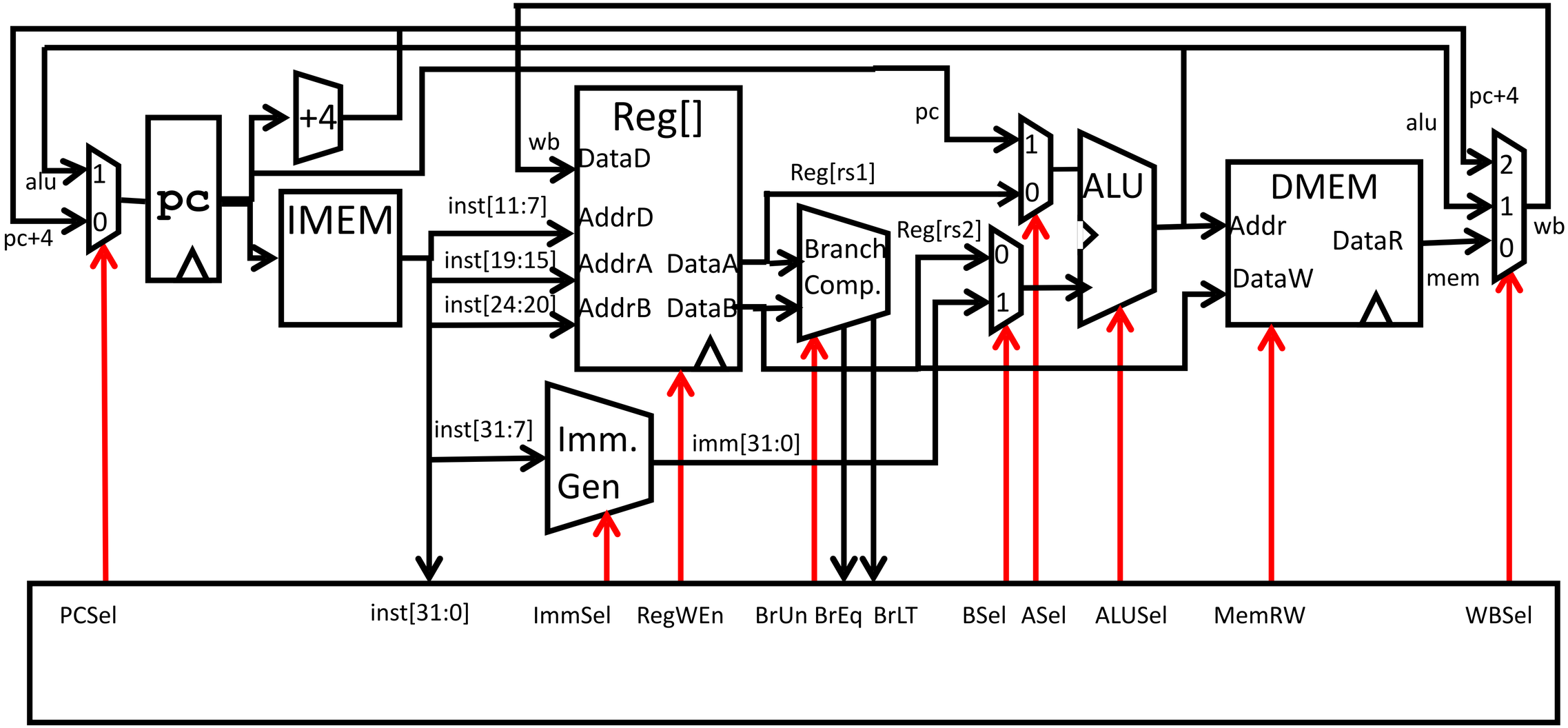


- JALR rd, rs, immediate
  - Writes PC+4 to Reg[rd] (return address)
  - Sets PC = Reg[rs1] + immediate
  - Uses same immediates as arithmetic and loads
    - ***no*** multiplication by 2 bytes

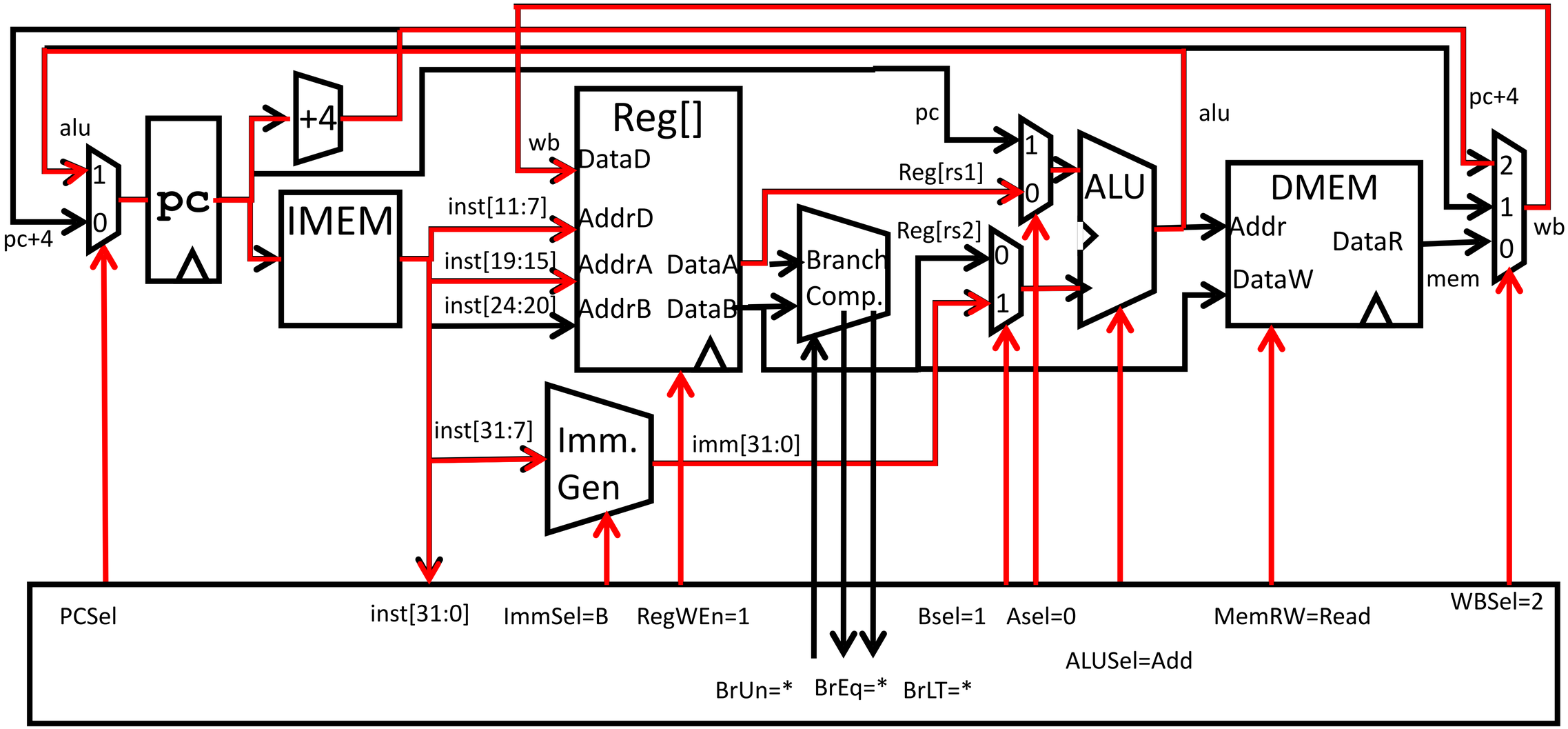
Adding branches to datapath



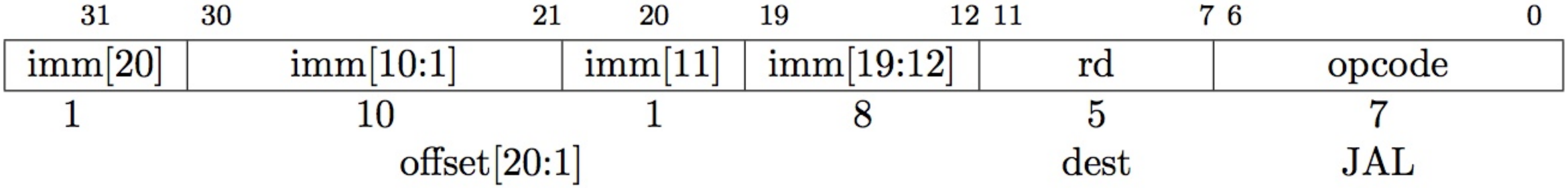
Adding `jalr` to datapath



Adding `jalr` to datapath

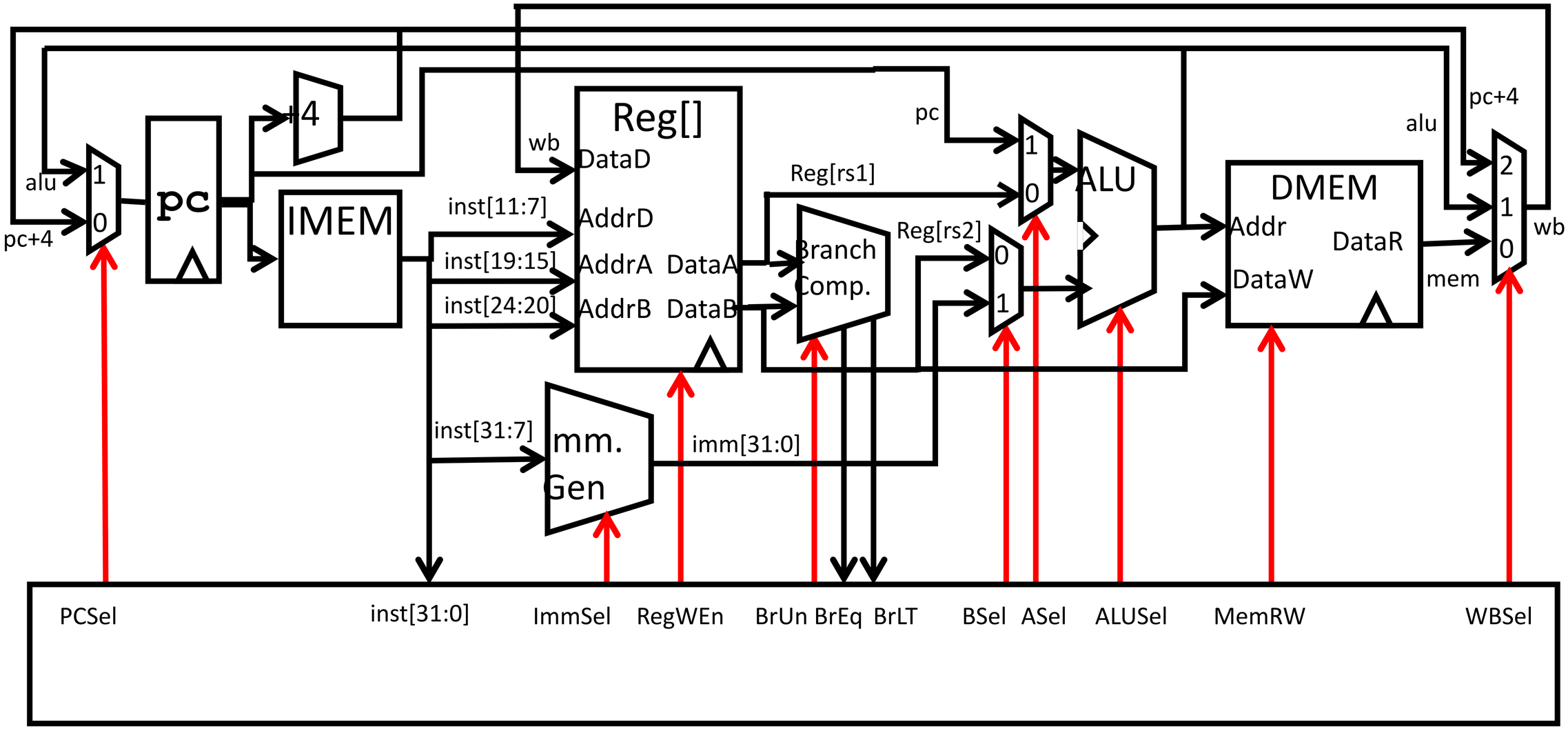


Implementing `jal` Instruction

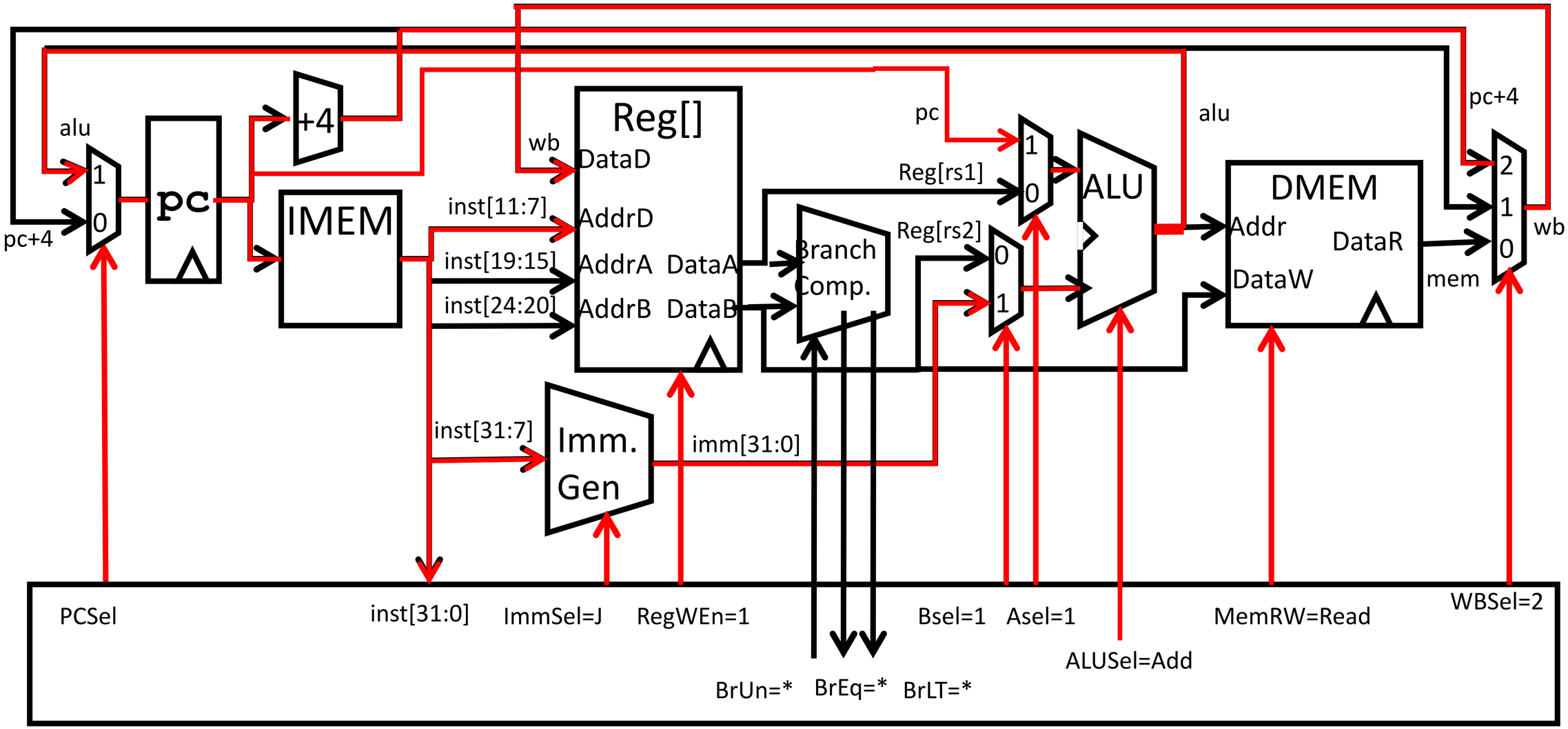


- JAL saves PC+4 in Reg[rd] (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

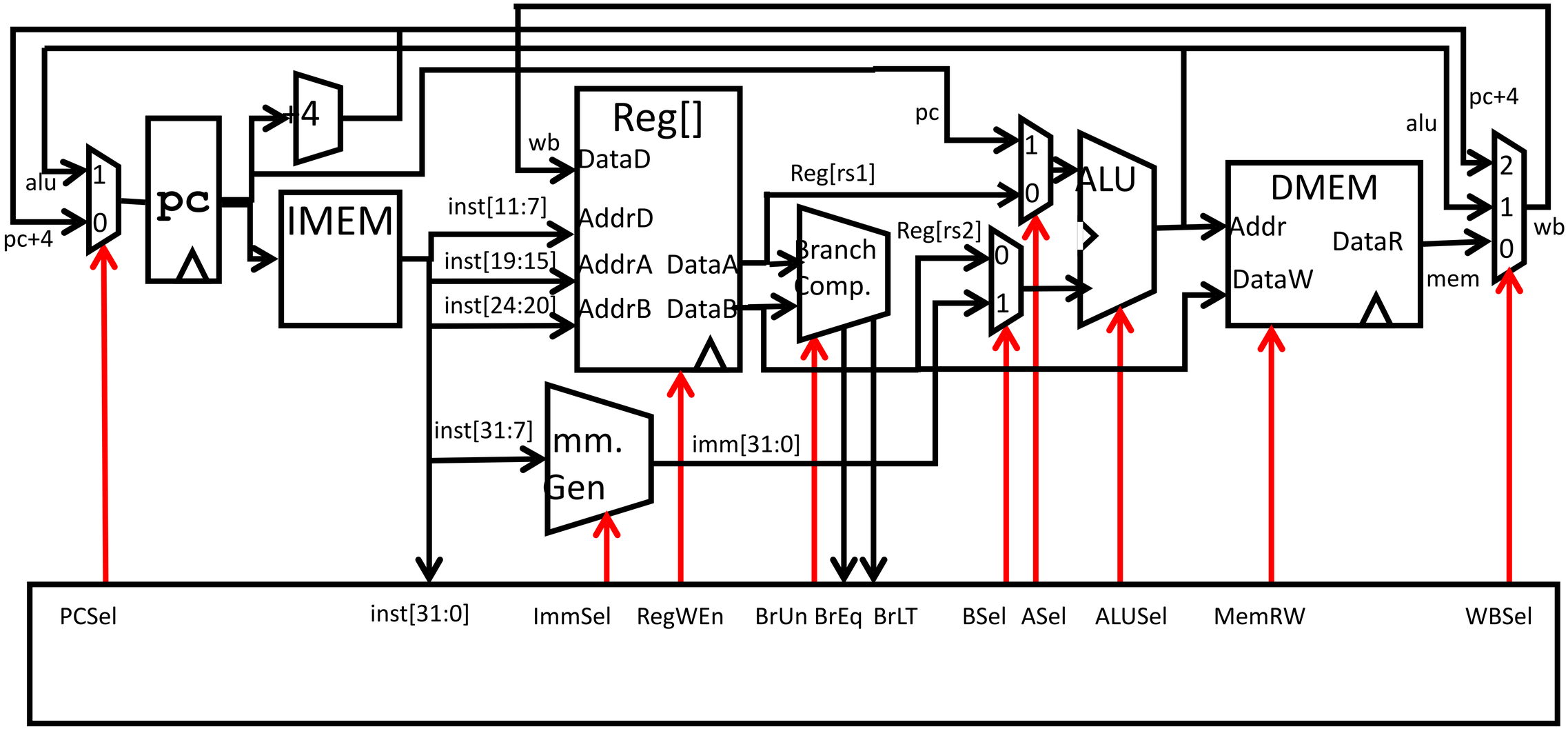
Adding `jal` to datapath



Adding `jal` to datapath



Single-Cycle RISC-V RV32I Datapath



Controller Implementation:

- Control logic works really well as a case statement...

```
always @* begin
    op = instr[26:31];
    imm = instr[15:0]; ...

    reg_dst = 1'bx;   // Don't care
    reg_write = 1'b0; // Do care, side effecting
    ...
    case (op)
     6'b000000: begin reg_write = 1; ... end
     ...
```

Processor Pipelining

Review: Processor Performance (The Iron Law)

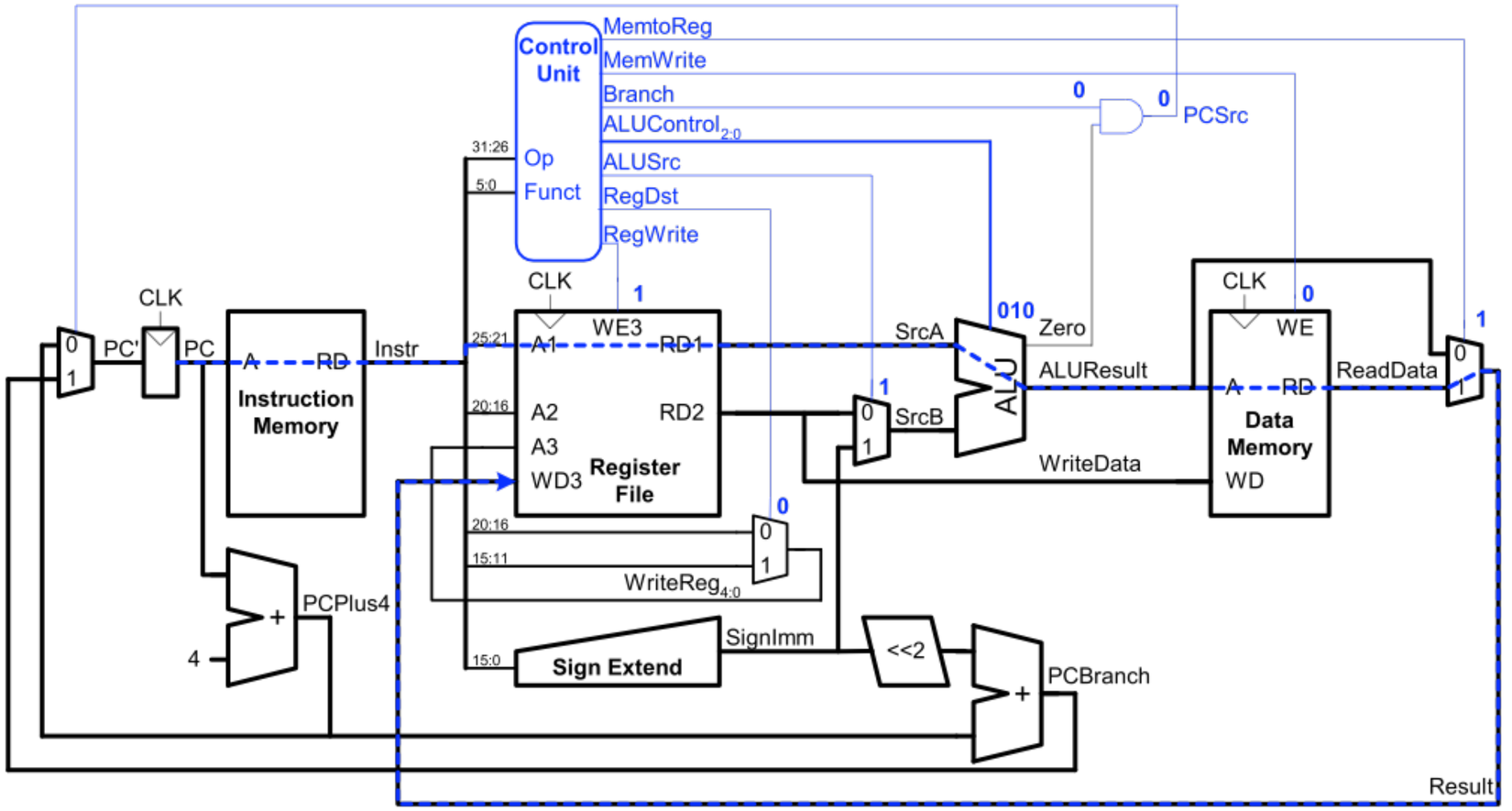
Program Execution Time

= (# instructions)(cycles/instruction)(seconds/cycle)

= # instructions x CPI x T_C

Single-Cycle Performance

- T_C *is limited by the critical path (*`lw`*)*



Single-Cycle Performance

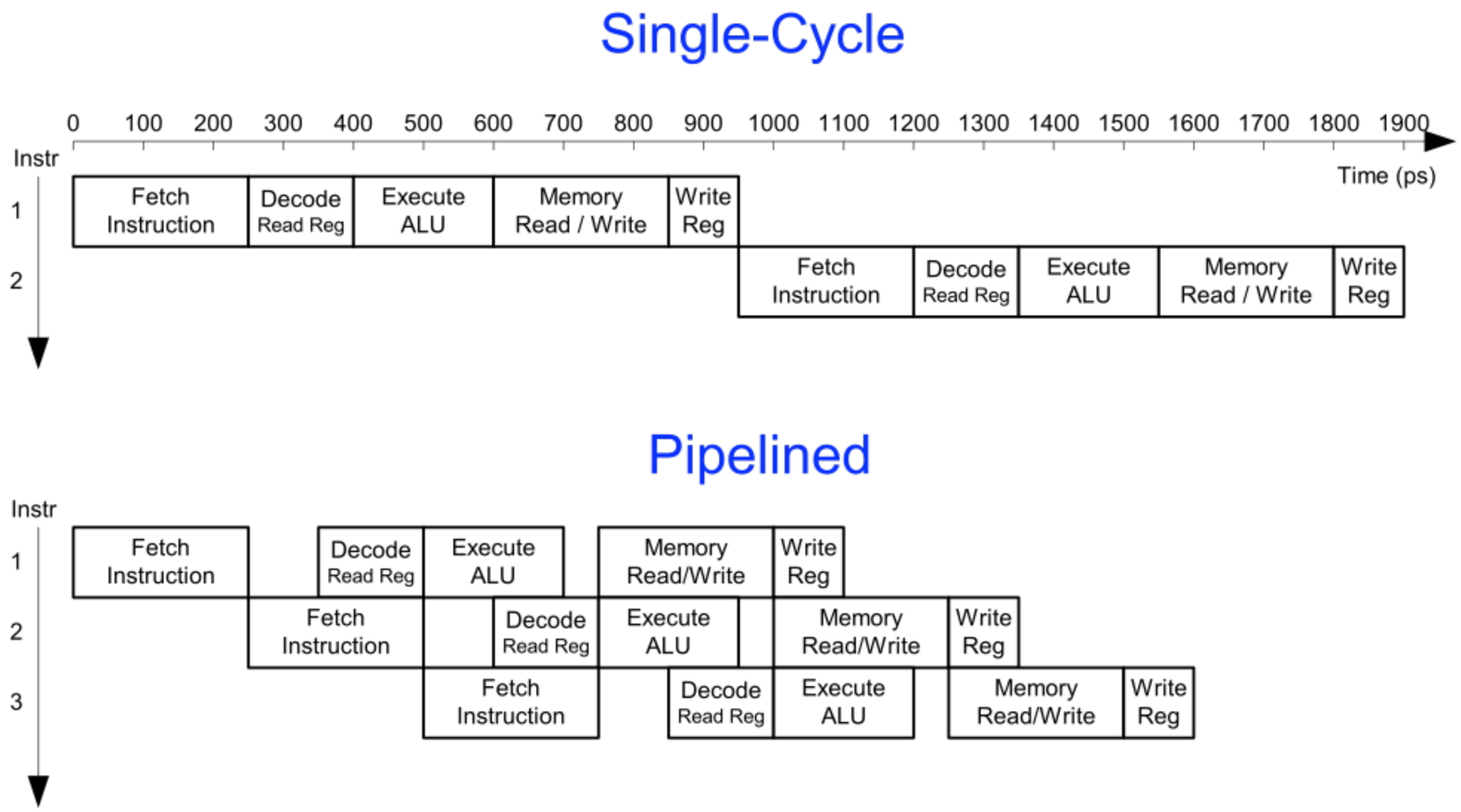
- *Single-cycle critical path:*
  $T_c = t_{q\_PC} + t_{mem} + max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$

- *In most implementations, limiting paths are:*
  - *memory, ALU, register file.*
  - $T_c = t_{q\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$

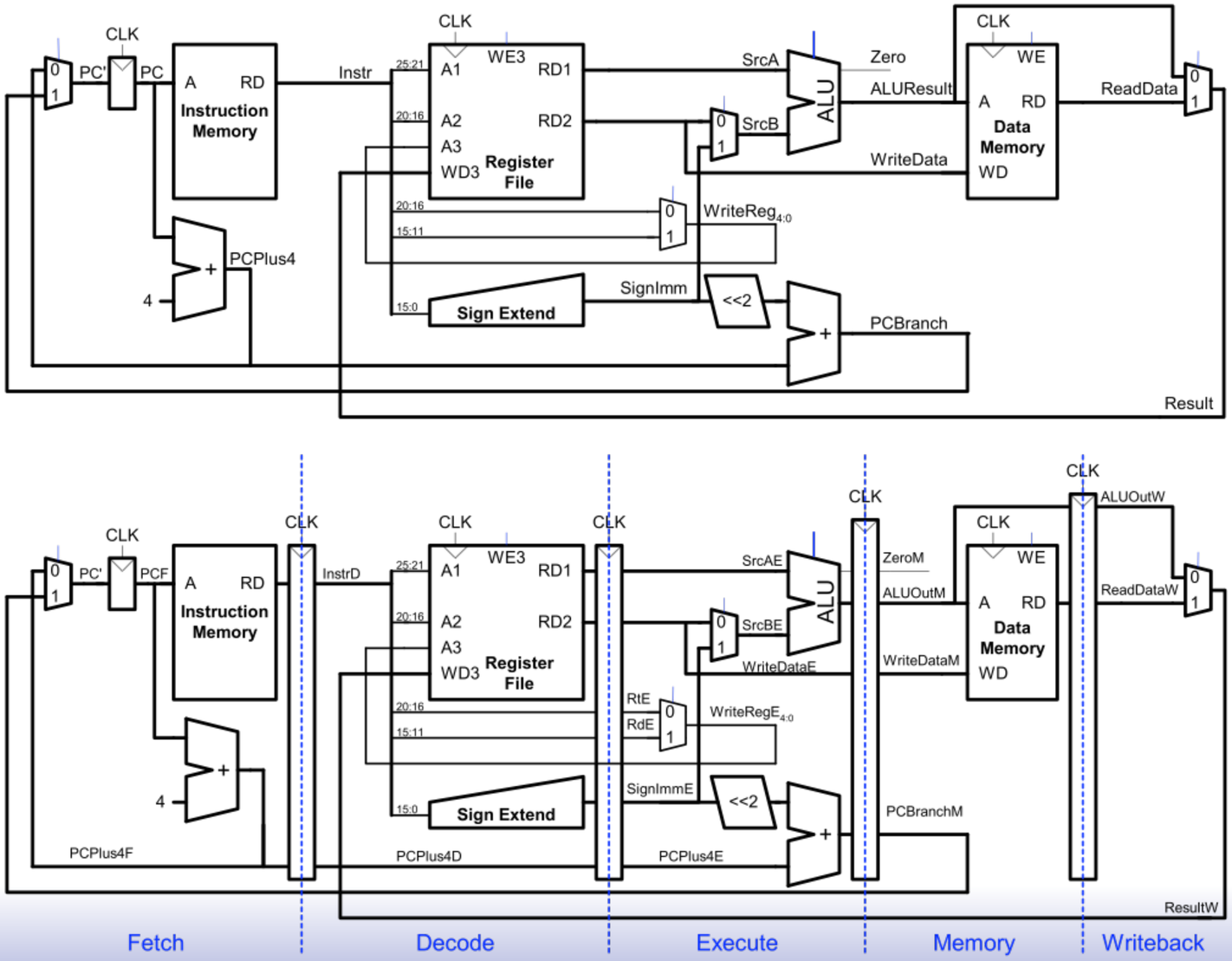
Pipelined Processor

- *Temporal parallelism*
- *Divide single-cycle processor into 5 stages:*
  - *Fetch*
  - *Decode*
  - *Execute*
  - *Memory*
  - *Writeback*
- *Add pipeline registers between stages*

Single-Cycle vs. Pipelined Performance

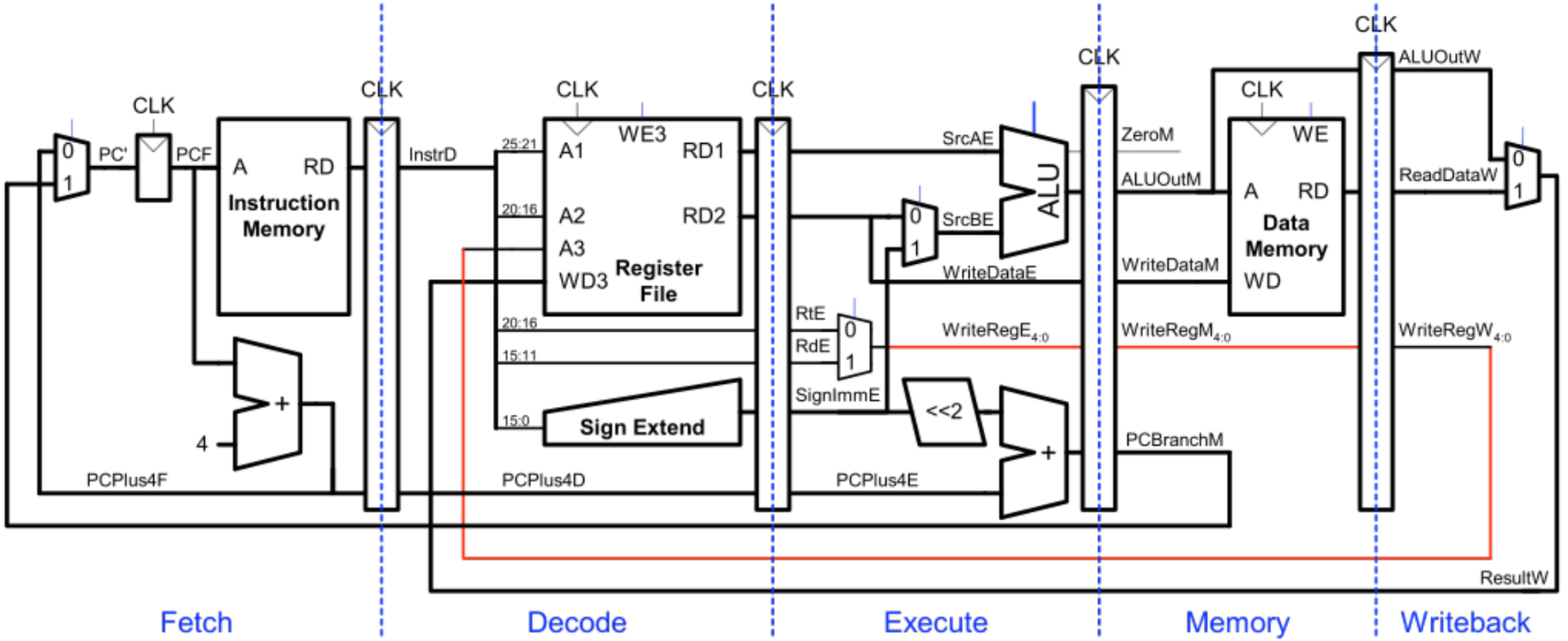


Single-Cycle and Pipelined Datapath

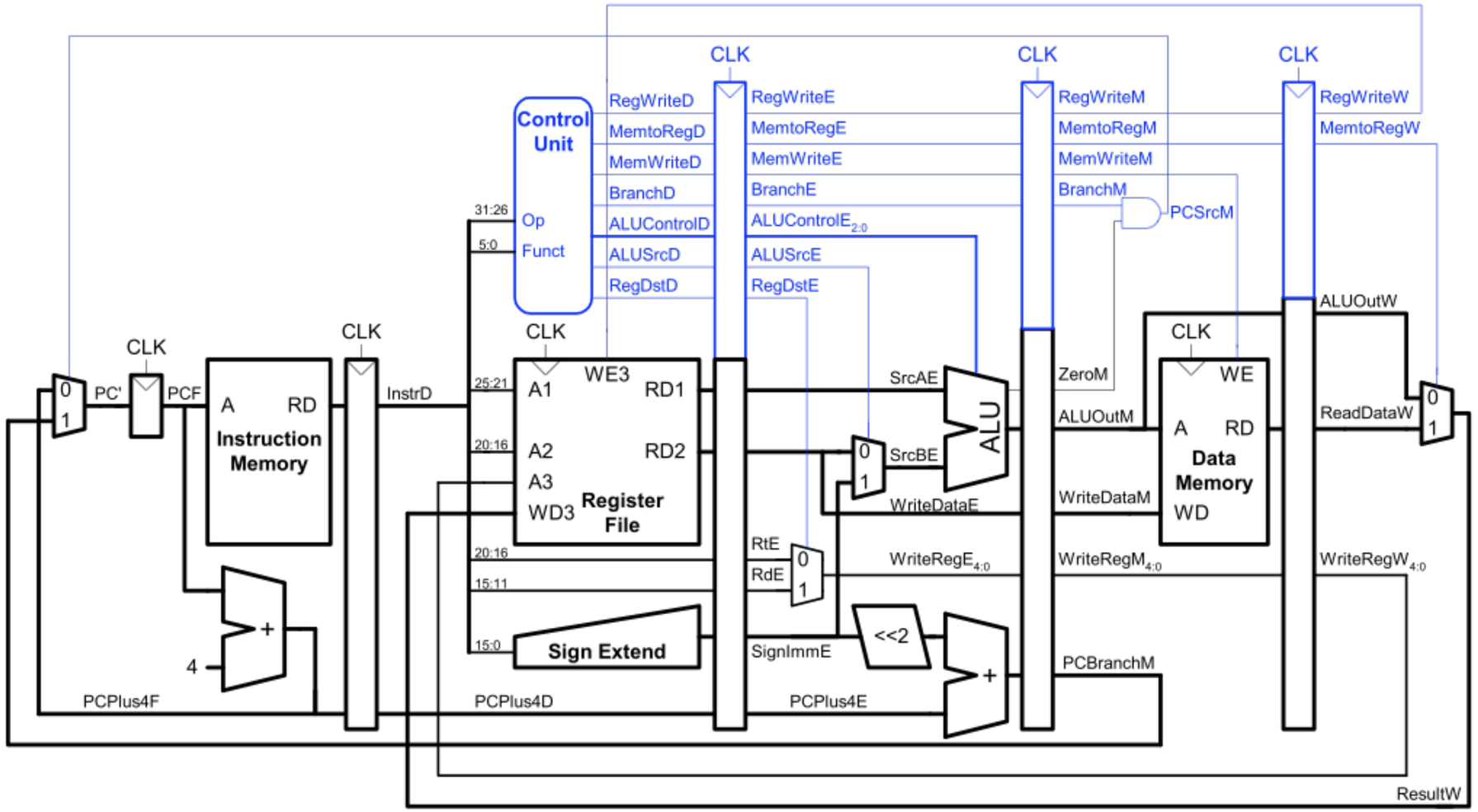


Corrected Pipelined Datapath

- *WriteReg must arrive at the same time as Result*



Pipelined Control



Same control unit as single-cycle processor

Control delayed to proper pipeline stage

Pipeline Hazards

- Occurs when an instruction depends on results from previous instruction that hasn't completed.
- Types of hazards:
  - **Data hazard:** register value not written back to register file yet
  - **Control hazard:** next instruction not decided yet (caused by branches)

Processor Pipelining

Deeper pipeline example.

IF1	**IF2**	**ID**	**X1**	**X2**	**M1**	**M2**	**WB**
	IF1	**IF2**	**ID**	**X1**	**X2**	**M1**	**M2**	**WB**

Deeper pipelines => less logic per stage => high clock rate.

But

Deeper pipelines => more hazards => more cost and/or higher CPI.*

Cycles per instruction might go up because of unresolvable hazards.

Remember, Performance = # instructions X Frequency$_{clk}$ / CPI

**Many designs included pipelines as long as 7, 10 and even 20 stages (like in the Intel Pentium 4). The later "Prescott" and "Cedar Mill" Pentium 4 cores (and their Pentium D derivatives) had a 31-stage pipeline.*

How about shorter pipelines ... Less cost, less performance

3-Stage Pipeline

3-Stage Pipeline (used for FPGA/ASIC project)

The blocks in the datapath with the greatest delay are: IMEM, ALU, and DMEM. Allocate one pipeline stage to each:

Use PC register as address to IMEM and retrieve next instruction. Instruction gets stored in a pipeline register, also called "instruction register", in this case.

Use ALU to compute result, memory address, or branch target address.

Access data memory or I/O device for load or store. Allow for setup time for register file write.

Most details you will need to work out for yourself. Some details to follow ... In particular, let's look at hazards.

3-stage Pipeline

Data Hazard

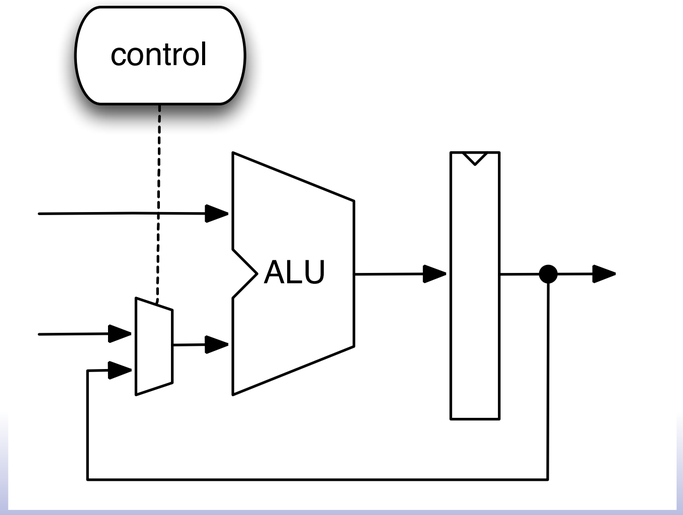
add x5, x3, x4	I	X	M		
add x7, x6, x5		I	X	M	

reg 5 value needed here!

reg 5 value updated here

The fix:

Selectively forward ALU result back to input of ALU.



- *Need to add mux at input to ALU, add control logic to sense when to activate. Check reference for details.*

3-stage Pipeline

Load Hazard

lw x5, offset(x4)	I	X	M		
add x7, x6, x5		I	X	M	

value needed here!

Memory value known here. It is written into the regfile on this edge.

The fix: Delay the dependent instruction by one cycle to allow the load to complete, send the result of load directly to the ALU (and to the regfile). No delay if not dependent!

lw x5, offset(x4)	I	X	M		
add x7, x6, x5		I	nop	nop	
add x7, x6, x5			I	X	M

3-stage Pipeline

Control Hazard

beq x1, x2, L1	I	X	M		
add x5, x3, x4		I	X	M	
add x6, x1, x2			I	X	M
L1: sub x7, x6, x5				I	X

but needed here!

branch address ready here

The fix:

*Several Possibilities:**

1. *Always delay fetch of instruction after branch*
2. *Assume branch "not taken", continue with instruction at PC+4, and correct later if wrong.*
3. *Predict branch taken or not based on history (state) and correct later if wrong.*

1. *Simple, but all branches now take 2 cycles (lowers performance)*
2. *Simple, only some branches take 2 cycles (better performance)*
3. *Complex, very few branches take 2 cycles (best performance)*

* *MIPS defines "branch delay slot", RISC-V doesn't*

Predict “not taken”

Branch address ready at end of X stage:

- *If branch “not taken”, do nothing.*
- *If branch “taken”, then kill instruction in I stage (about to enter X stage) and fetch at new target address (PC)*

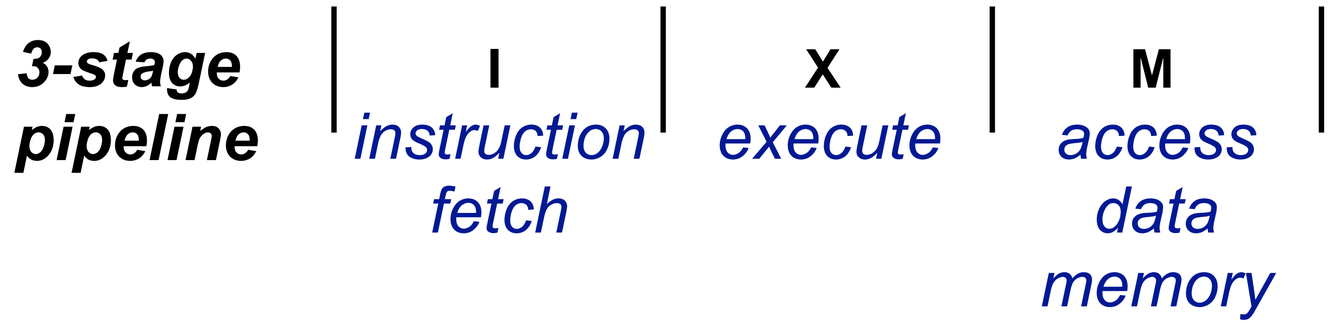
bneq x1, x1, L1	**I**	**X**	**M**		
add x5, x3, x4		**I**	**X**	**M**	
add x6, x1, x2			**I**	**X**	**M**
L1: sub x7, x6, x5				**I**	**X**

Not taken

beq x1, x1, L1	**I**	**X**	**M**		
add x5, x3, x4		**I**	**nop**	**nop**	
L1: sub x7, x6, x5			**I**	**X**	**M**

Taken

EECS151 Project CPU Pipelining Summary



- Pipeline rules:
  - Writes/reads to/from DMem are clocked on the leading edge of the clock in the “M” stage
  - Writes to RegFile at the end of the “M” stage
  - Instruction Decode and Register File access is up to you.
- Branch: predict “not-taken”
- Load: 1 cycle delay/stall on dependent instruction
- Bypass ALU for data hazards
- More details in upcoming spec