EECS 151/251A
Spring 2021
Digital Design and Integrated Circuits

Instructor:
John Wawrzynek

Lecture 14: RISC-V Part 2

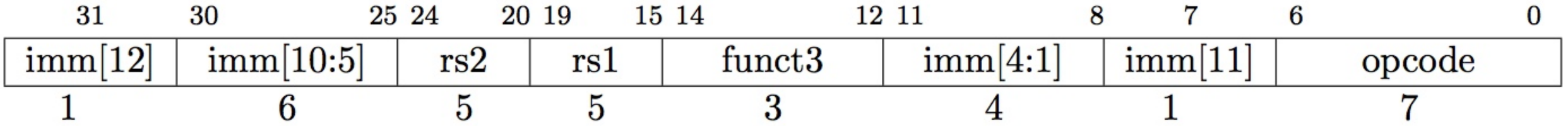
Announcements

- Virtual Front Row today, 3/4:
  - *Tony Kam*
  - *Ben Tait*
  - *Robin Chu*
  - *Neil Kulkarni*
  - *Robert Puccinelli*
- HW6 posted (due Monday)
- Midterm Reminder
  - Format TBD
- No HW next week

	2/25	Circuit Timing Part 2 (slides)(video)	
7	3/2	RISC-V Microarchitecture and Implementation	Discussion 7
	3/4	RISC-V Part 2	
8	3/9	Exam 1 Review	Discussion 8
	3/11	No Class - Exam 6-9PM	
9	3/16	Power and Energy	Discussion 9
	3/18	Memory Blocks 1	
10	3/23	Spring Recess	
	3/25	Spring Recess	

Implementing Branches

Uses the "B-type" instruction format



- RISC-V Assembly Instruction, example:

  `beq rs1, rs2, label`

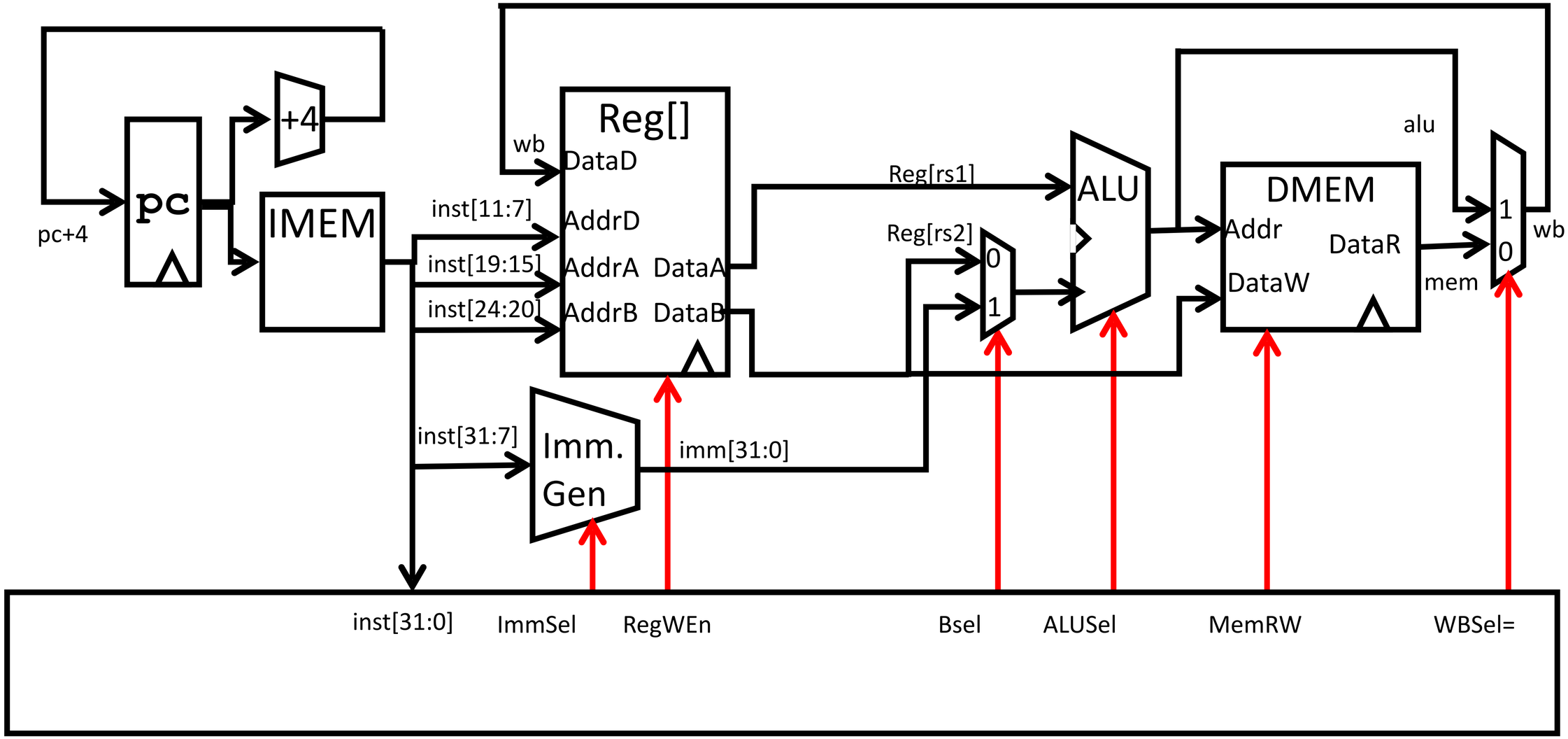
  `if rs1==rs2 pc ← pc + offset // offset computed by compiler/assembler and stored in the immediate field(s)`

  example:

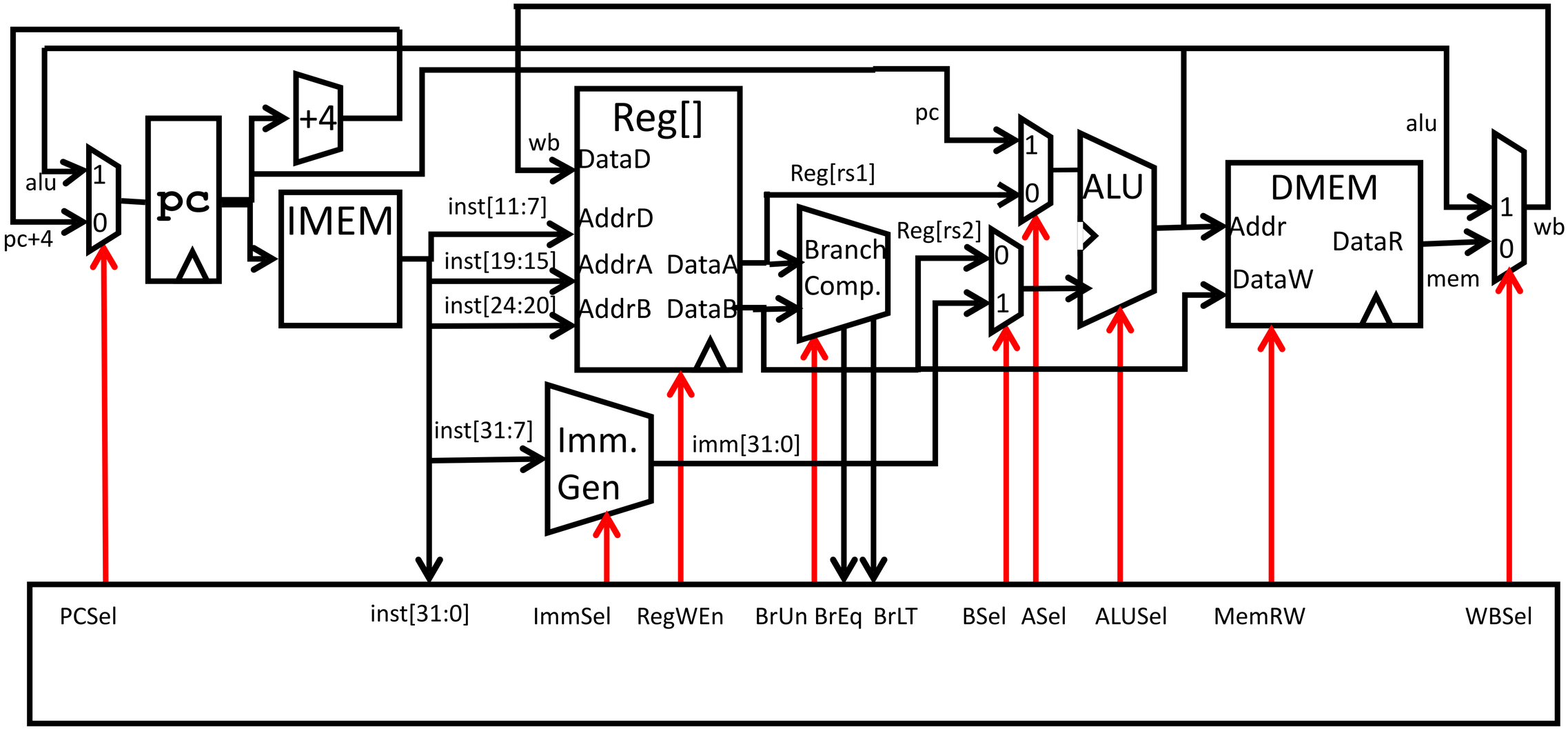
  `beq x1, x2, L1`

- 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)

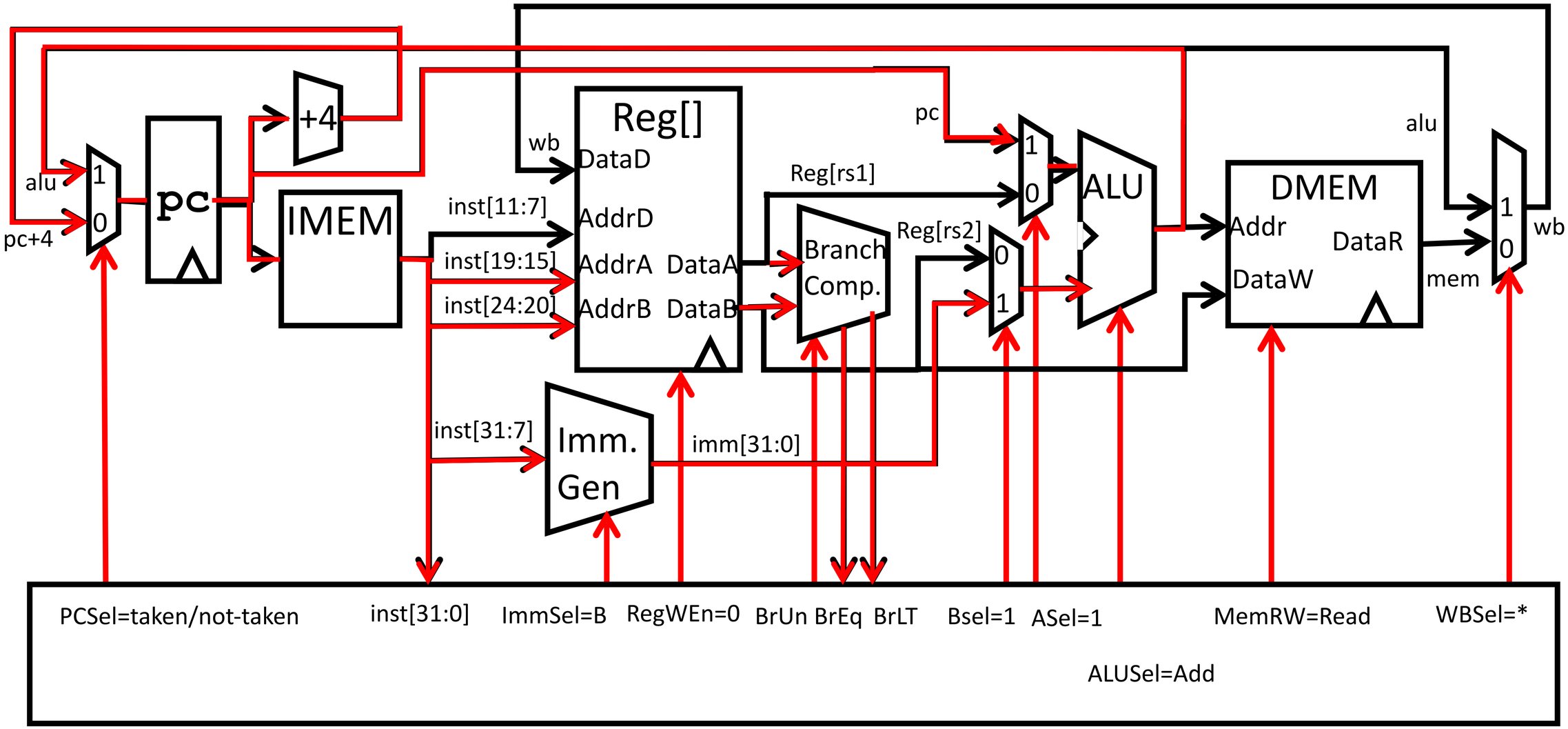
Review: Adding **sw** to datapath



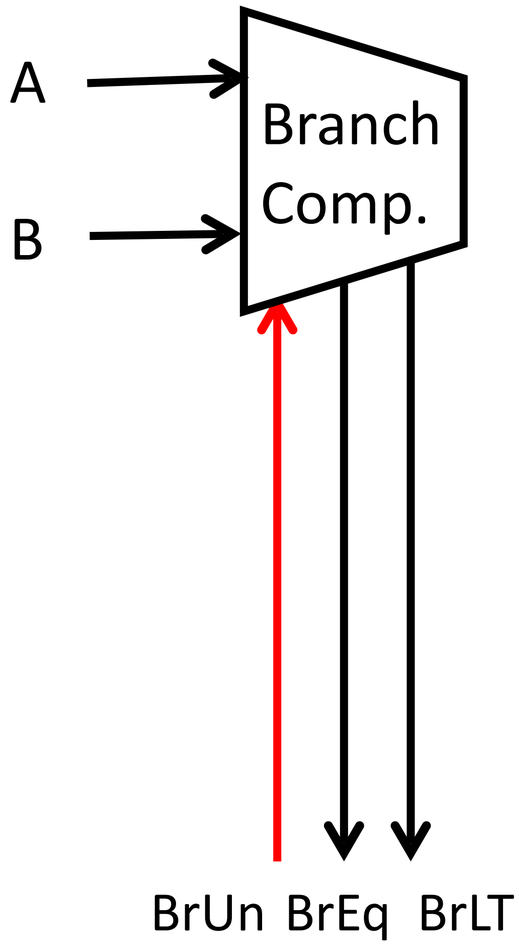
Adding branches to datapath



Adding branches to datapath



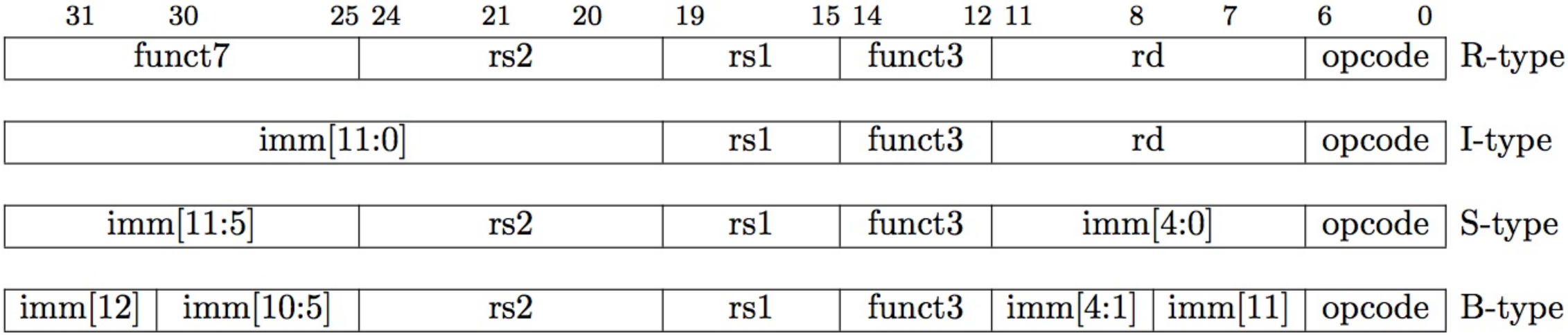
Branch Comparator



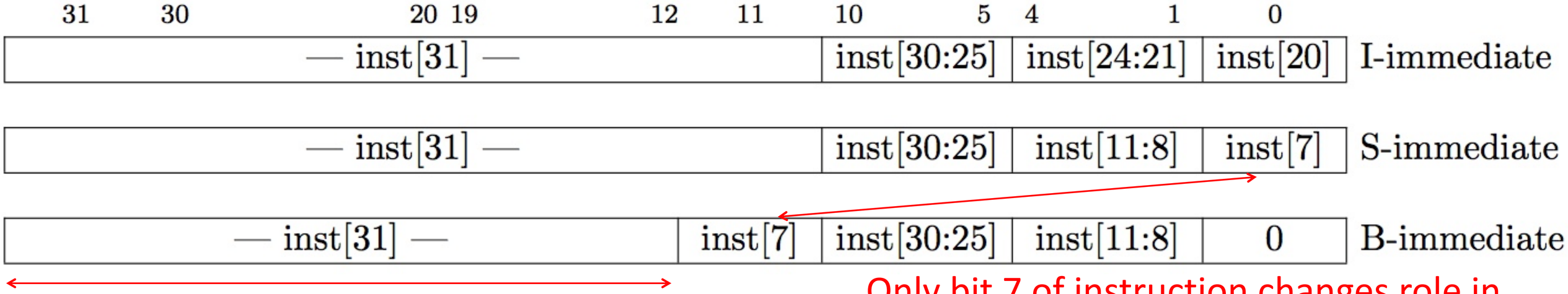
- BrEq = 1, if A=B
- BrLT = 1, if A < B
- BrUn =1 selects unsigned comparison for BrLT, 0=signed
- BGE branch: A >= B, if !(A<B)

RISC-V Immediate Encoding

Instruction Encodings, inst[31:0]



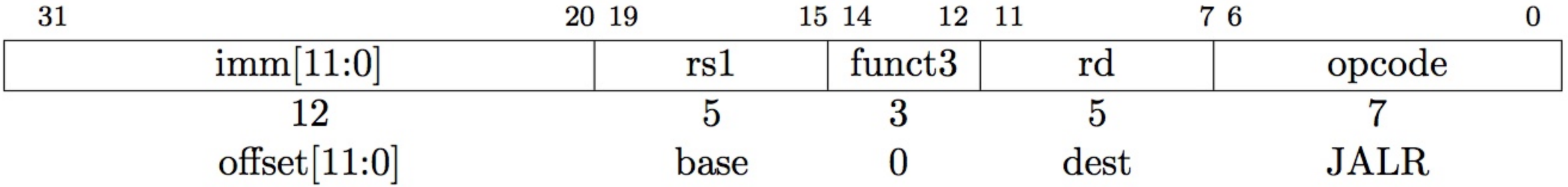
32-bit immediates produced, imm[31:0]



Upper bits sign-extended from inst[31] always

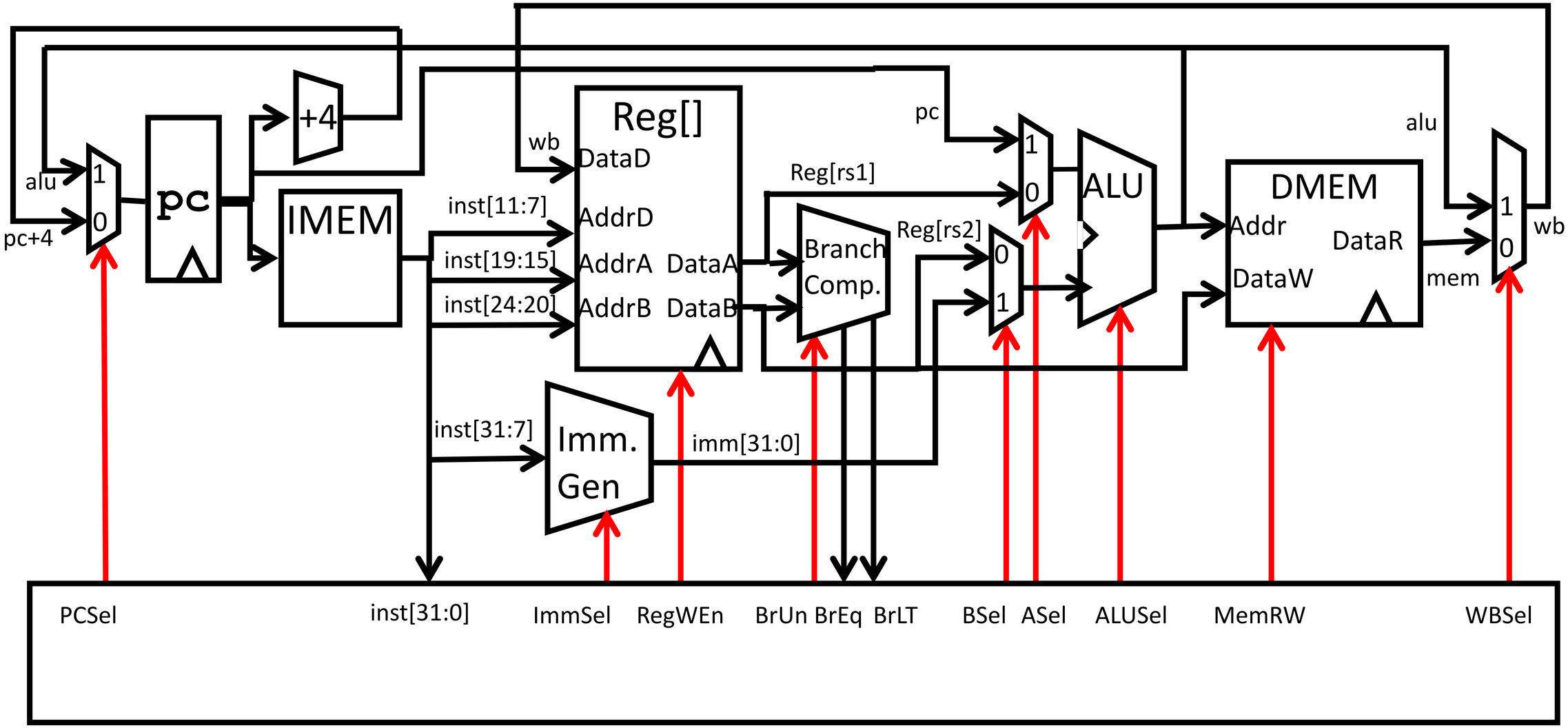
Only bit 7 of instruction changes role in immediate between S and B

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

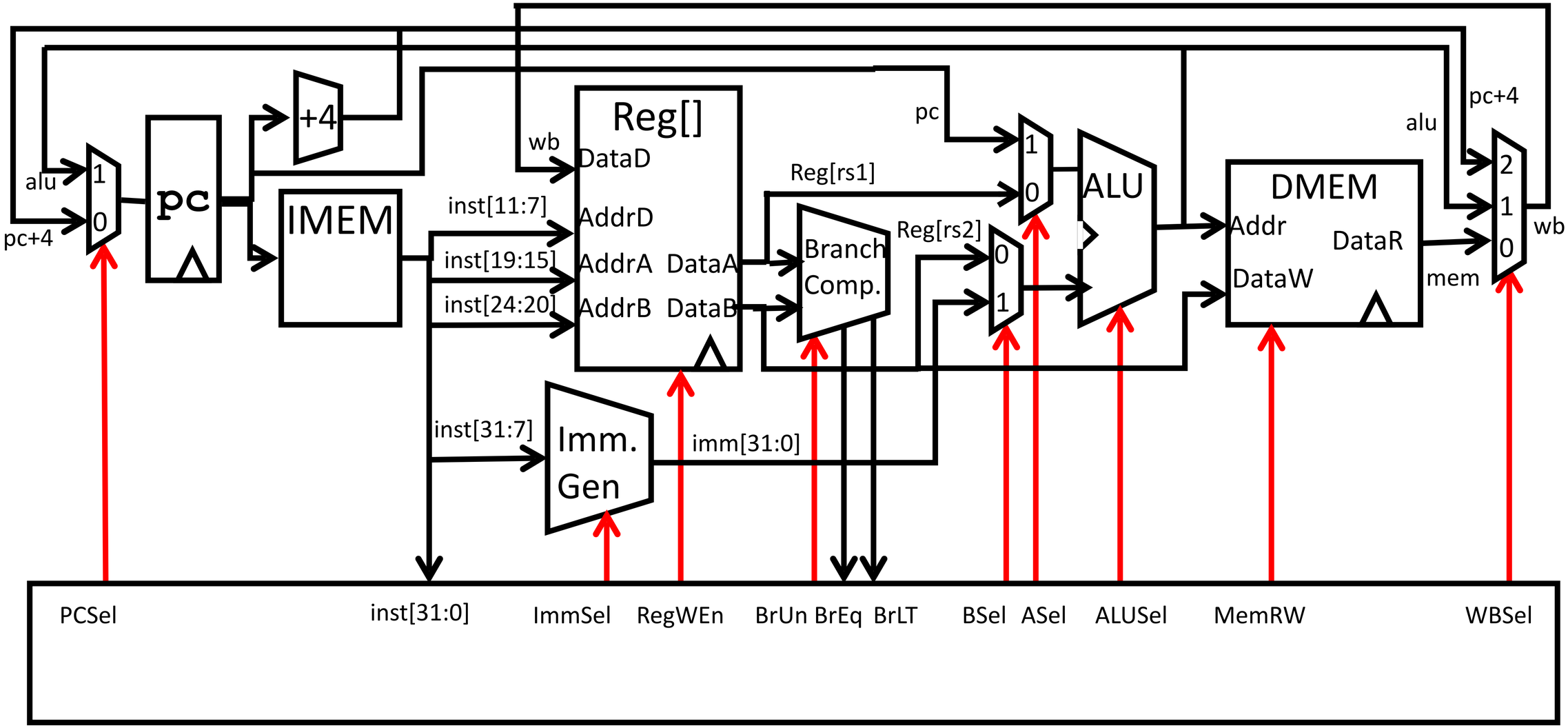


- JALR rd, rs, immediate
  - Writes PC+4 to Reg[rd] (return address)
  - Sets PC = Reg[rs1] + offset
  - Uses same immediates as arithmetic and loads
    - ***no*** multiplication by 2 bytes

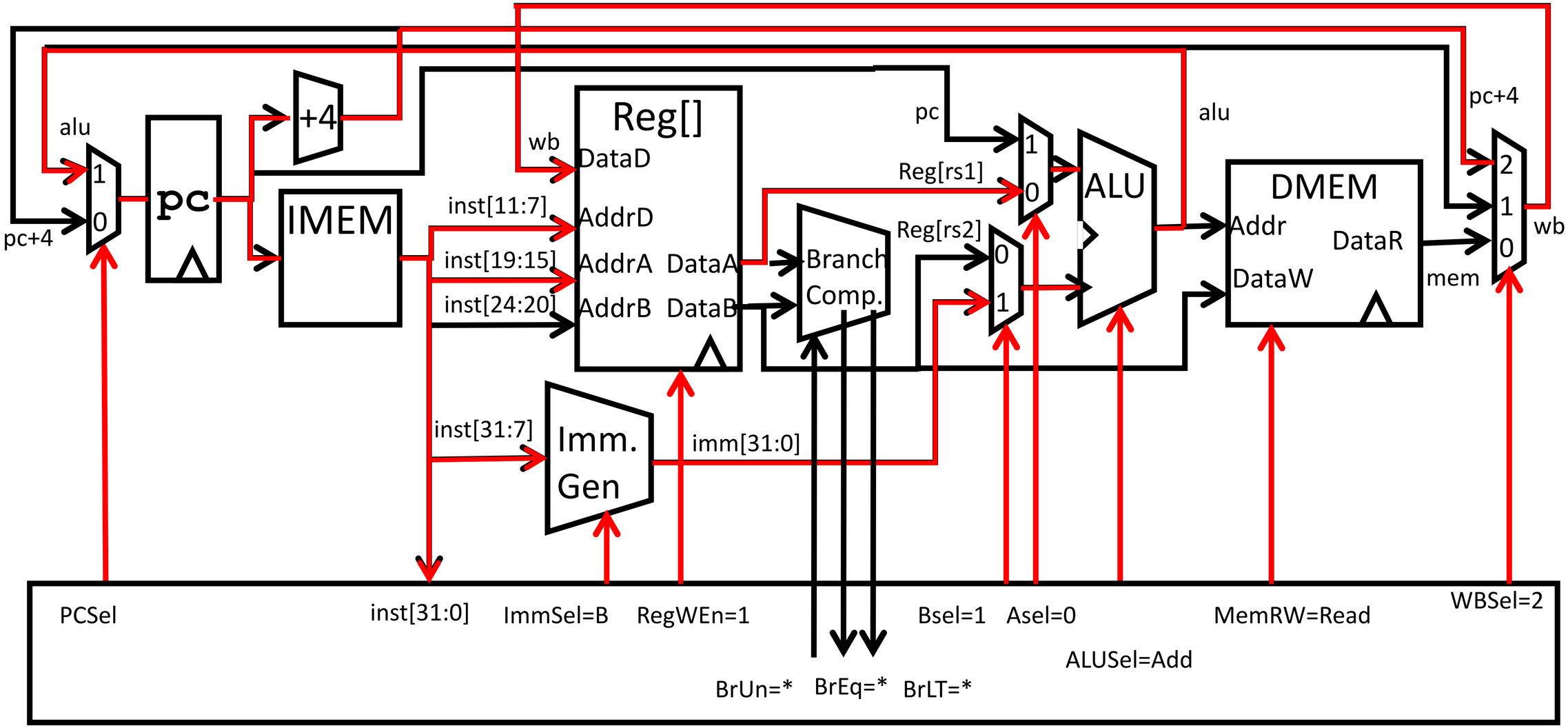
Review: Adding branches to datapath



Adding `jalr` to datapath

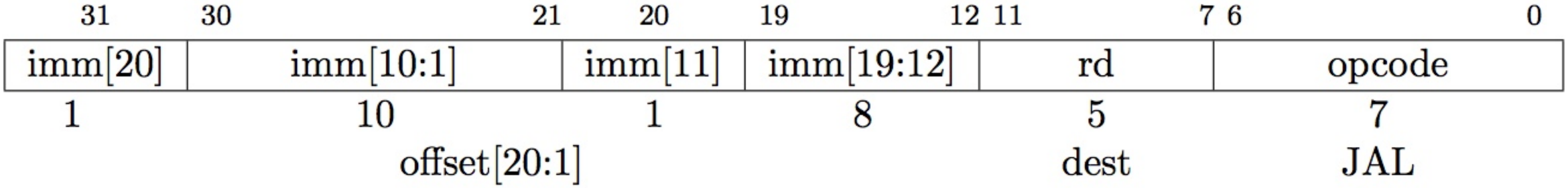


Adding `jalr` to datapath



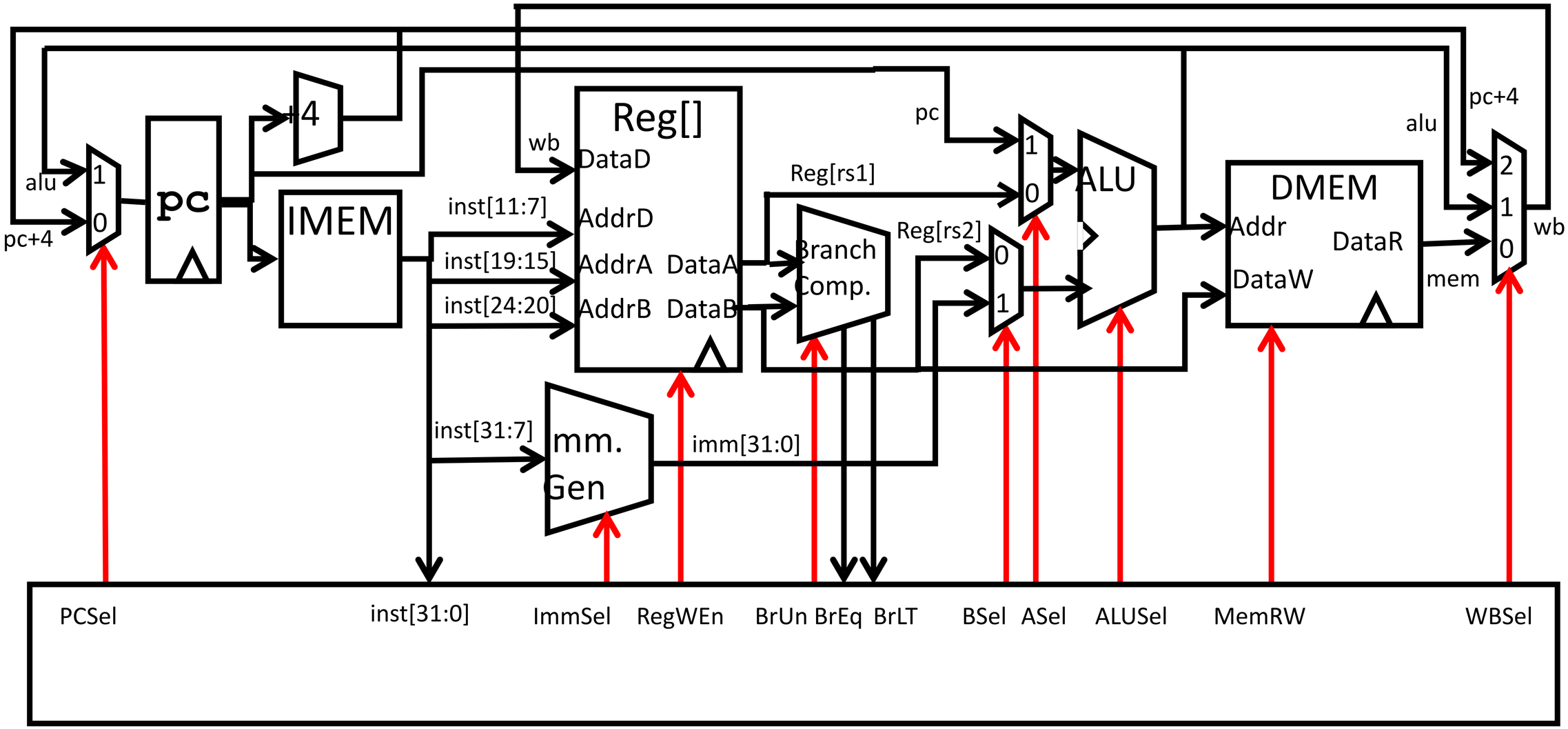
Implementing `jal` Instruction

Uses the "J-type" instruction format

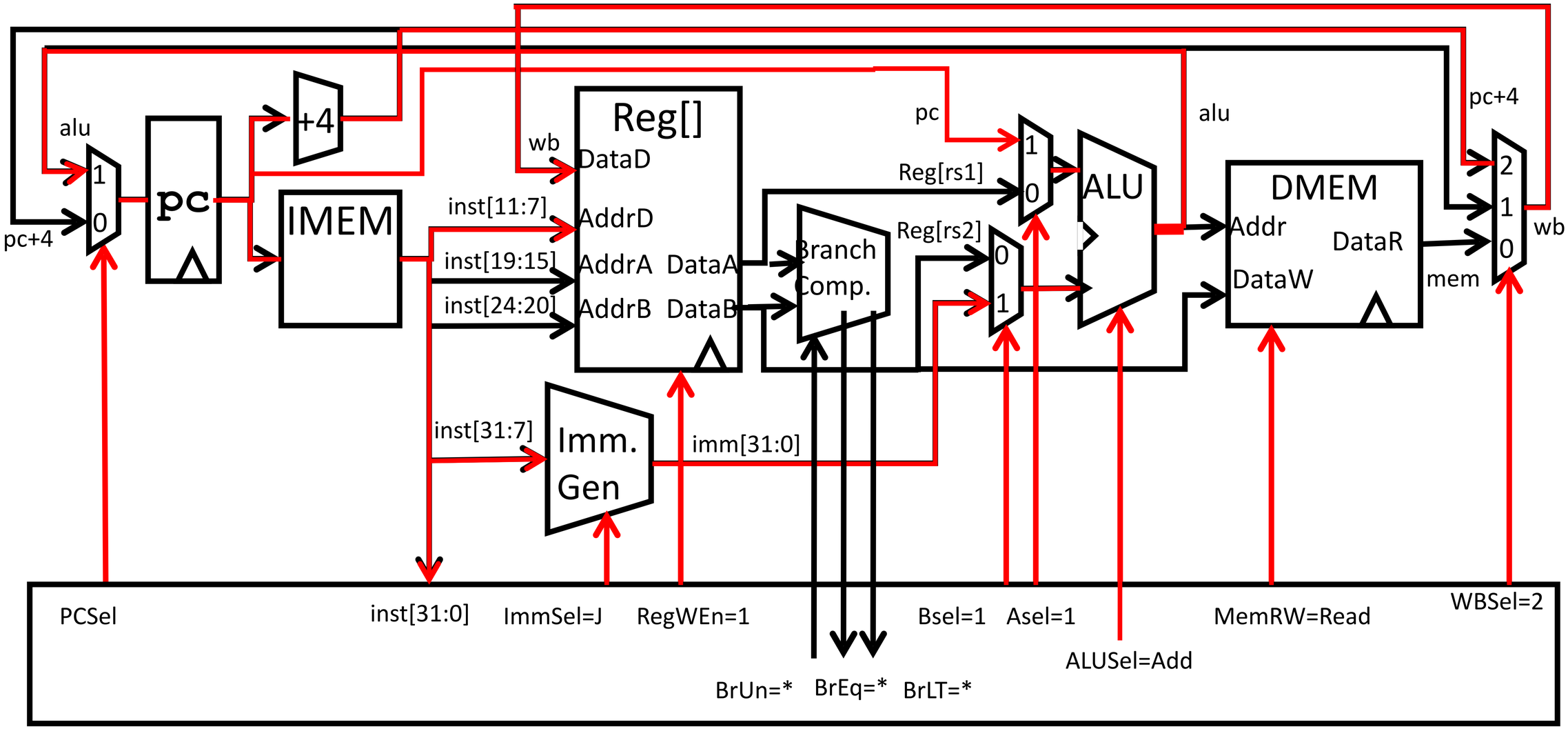


- JAL saves PC+4 in Reg[rd] (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

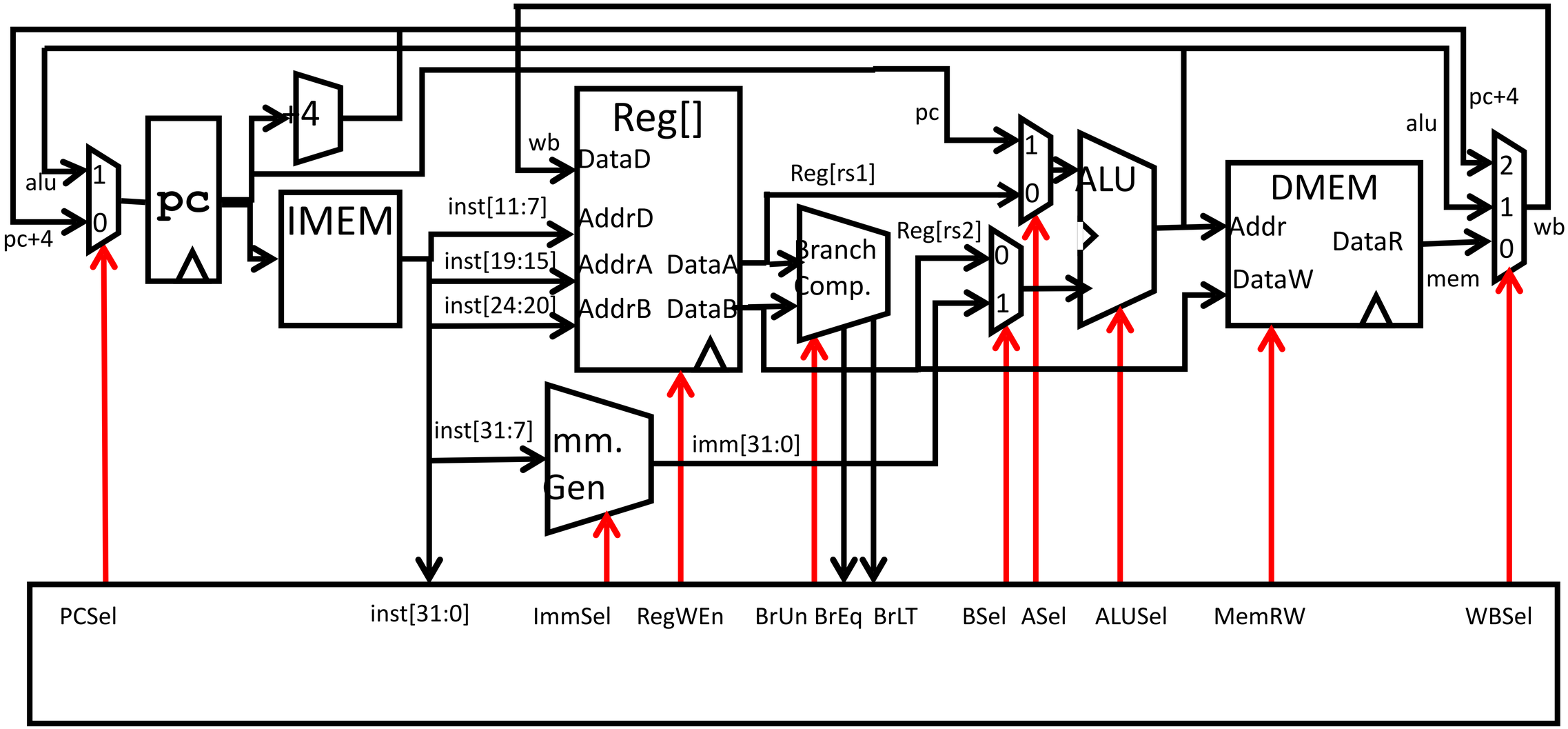
Adding `jal` to datapath



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; // By default don't write
    ...
    case (op)
     6'b000000: begin reg_write = 1; ... end
     ...
```

Processor Pipelining

Review: Processor Performance

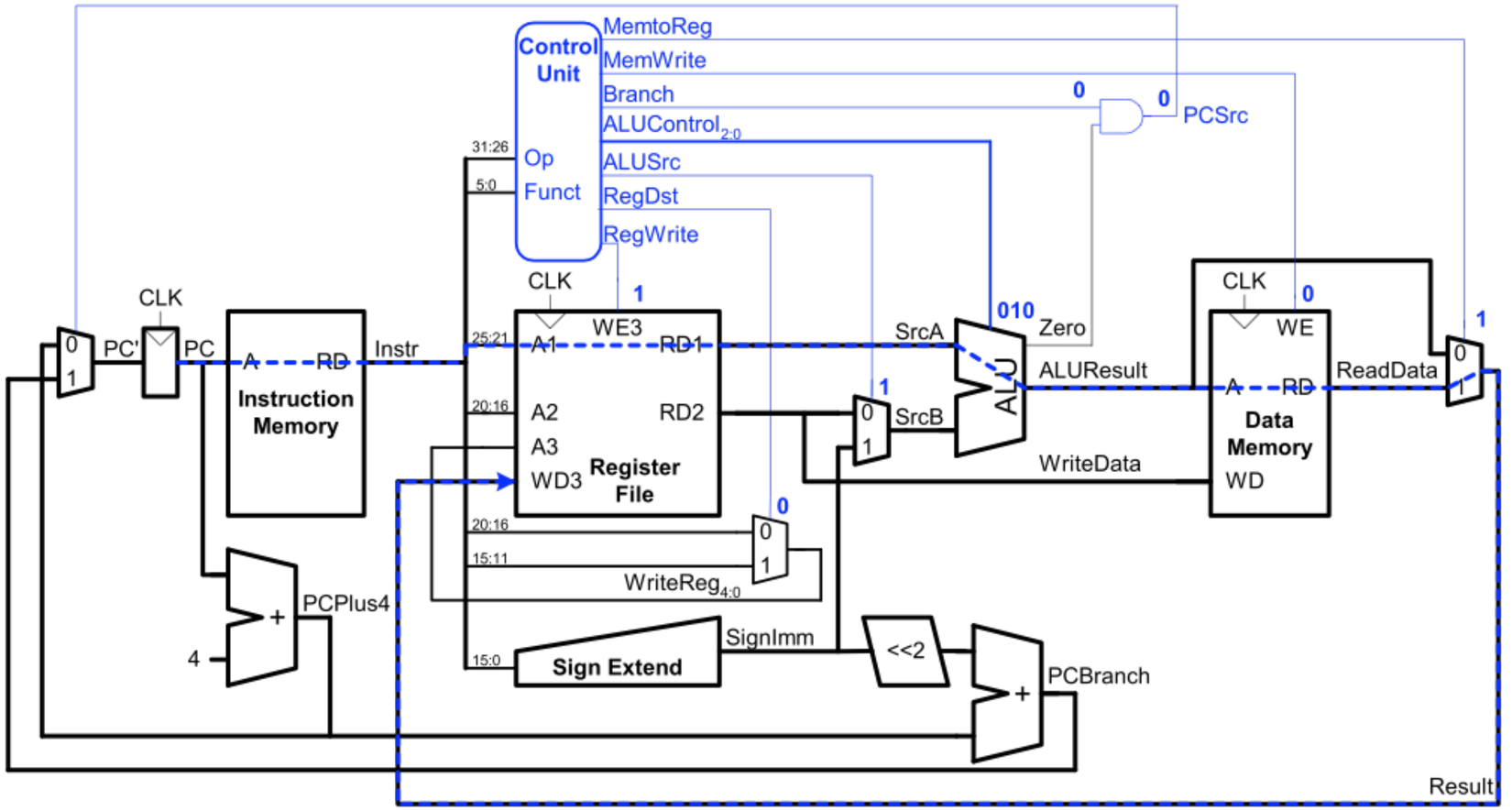
Program Execution Time

= (# 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:*
  $T_c = t_{q\_PC} + t_{mem} + max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$

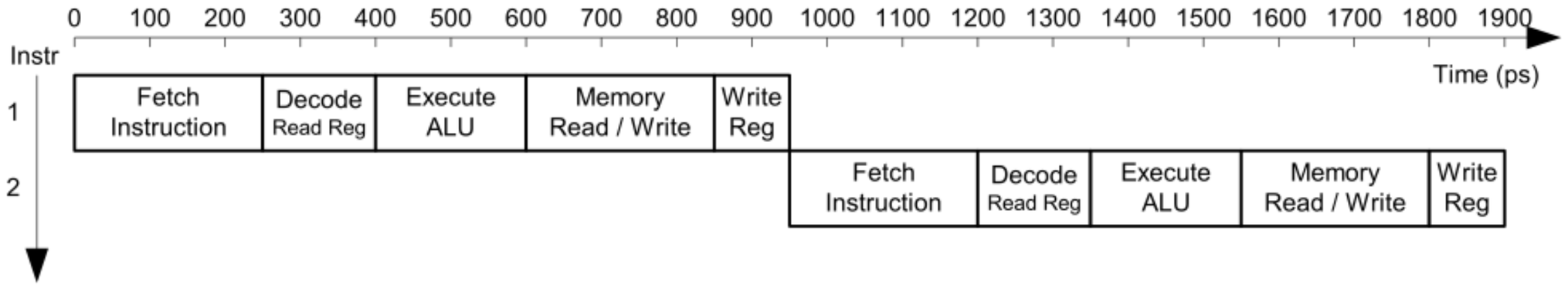
- *In most implementations, limiting paths are:*
  - *memory, ALU, register file.*
  - $T_c = t_{q\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$

Pipelined Processor

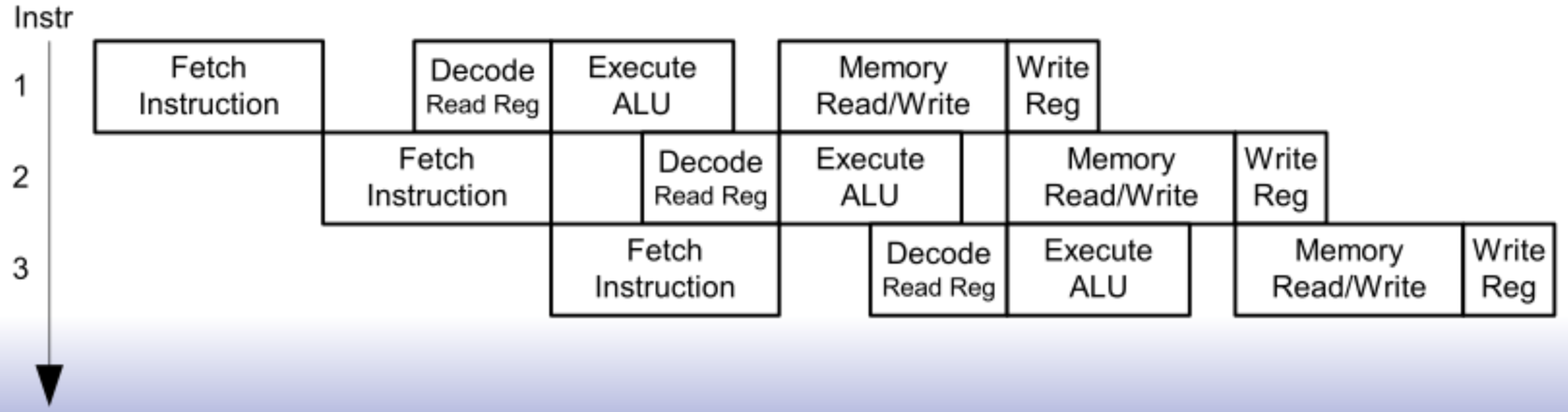
- *Use temporal parallelism*
- *Divide single-cycle processor into 5 stages:*
  - *Fetch*
  - *Decode*
  - *Execute*
  - *Memory*
  - *Writeback*
- *Add pipeline registers between stages*

Single-Cycle vs. Pipelined Performance

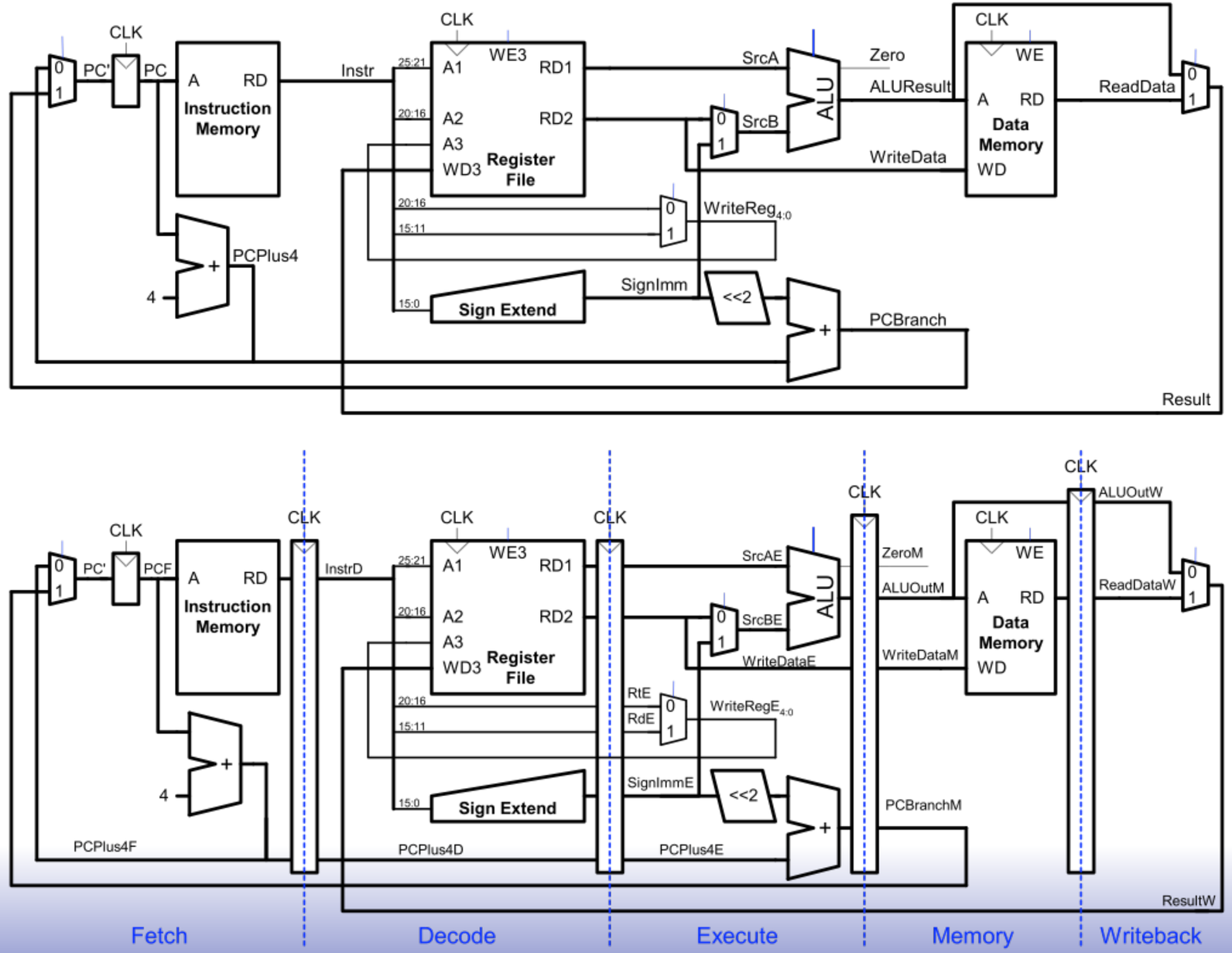
Single-Cycle



Pipelined

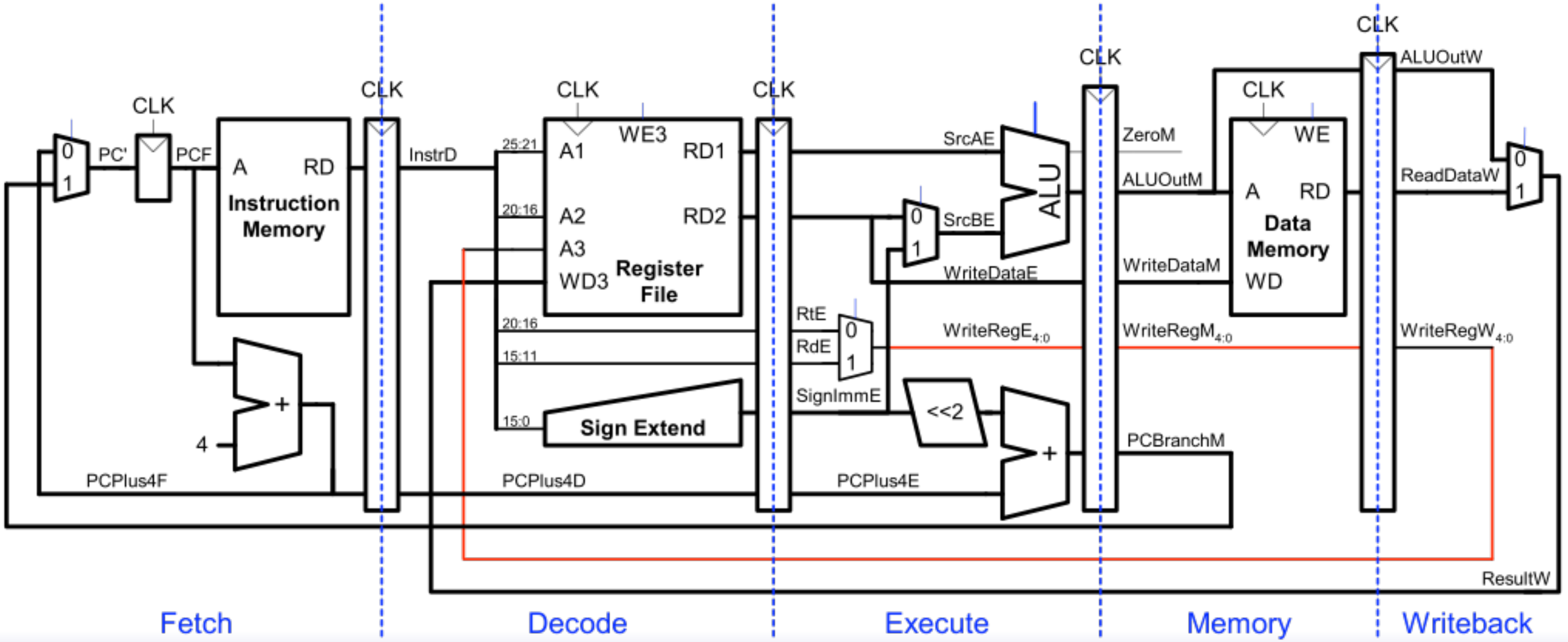


Single-Cycle and Pipelined Datapath

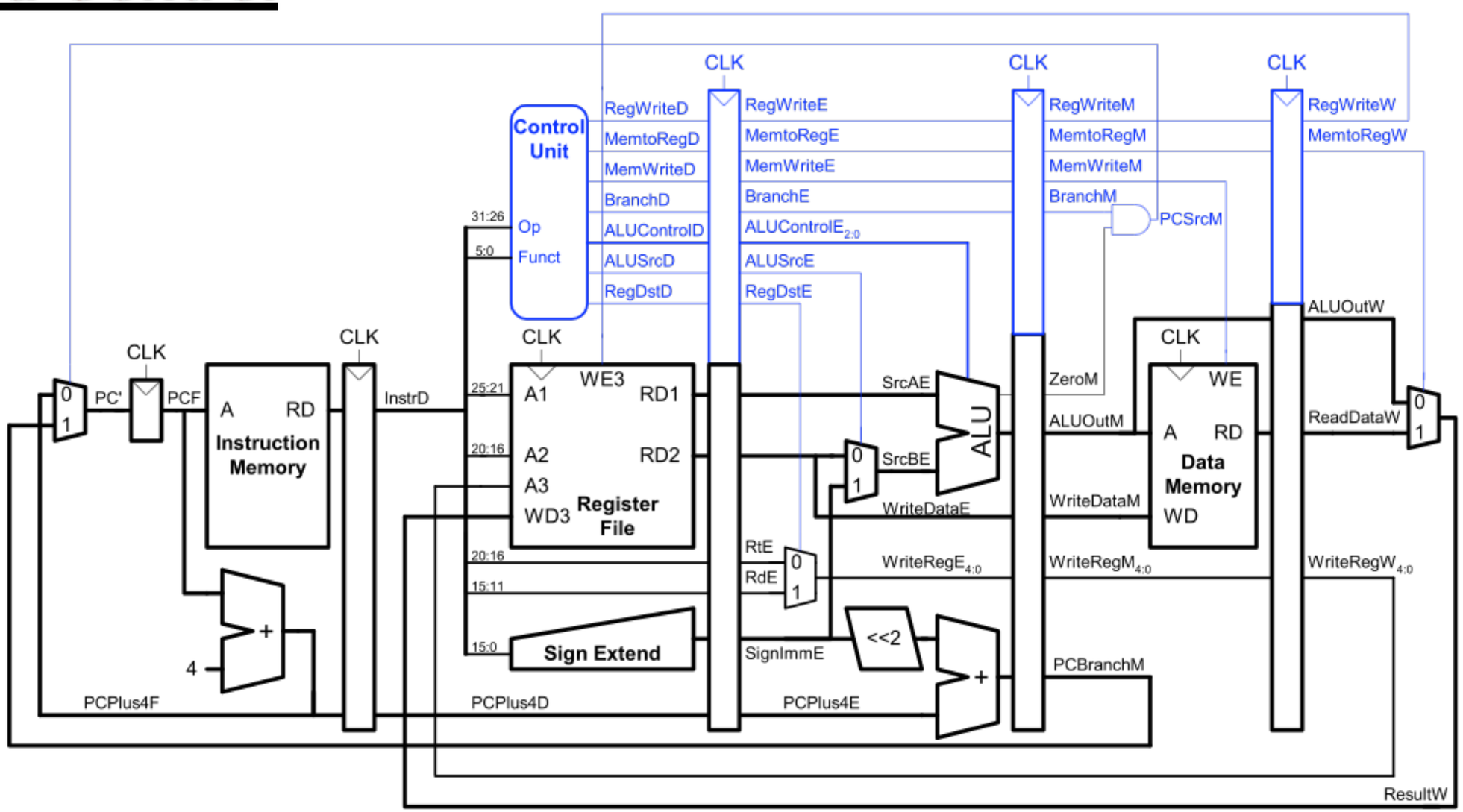


Corrected Pipelined Datapath

- *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)

We need to design ways to avoid hazards, else we pay the price in CPI (cycles per instruction) and processor performance suffers.

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 (but higher cost efficiency)

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

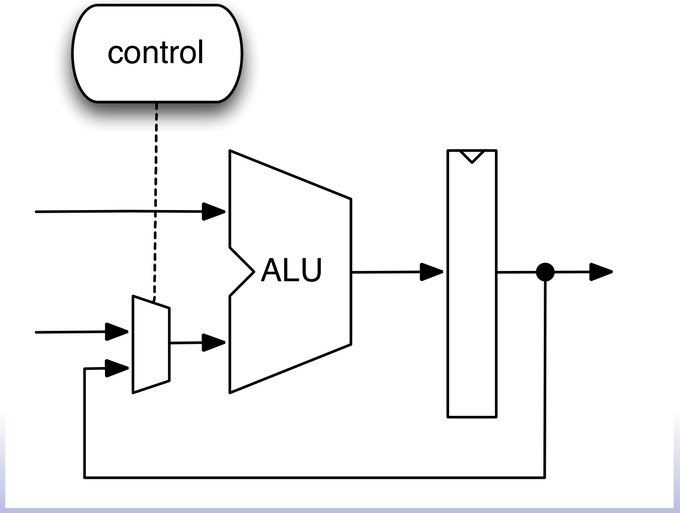
add x5, x3, x4	I	X	M		
add x7, x6, x5		I	X	M	

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!

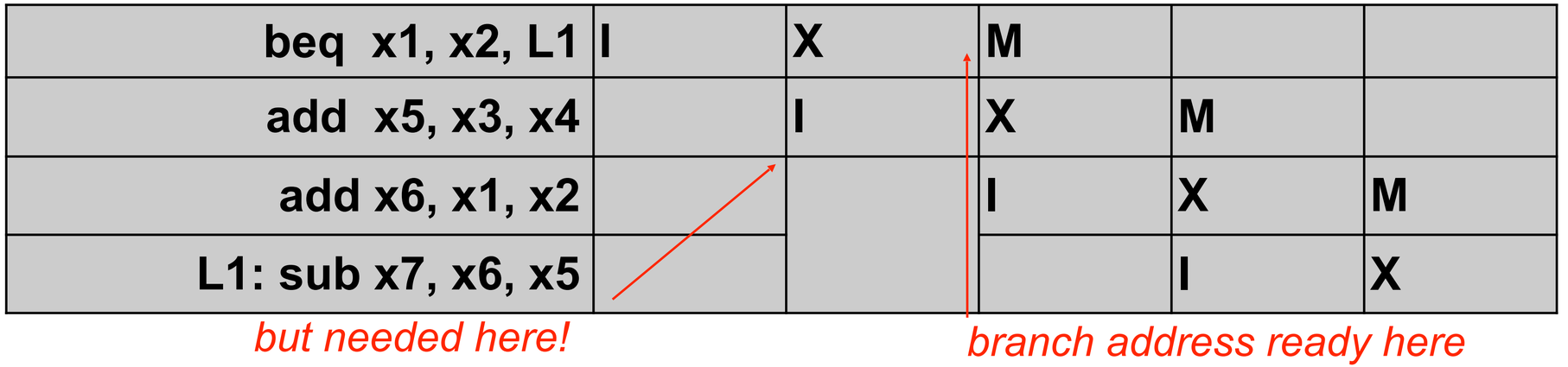
Memory value known here. It is written into the regfile on this edge.

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



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

3-stage pipeline	**I** *instruction fetch*	**X** *execute*	**M** *access data memory*

- 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

End of Lecture 14