



EECS 151/251A

Spring 2023

**Digital Design and Integrated
Circuits**

Instructor:

John Wawrzynek

Lecture 13: RISC-V Part 1

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 branch predictor?

Lec13 and 14 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:

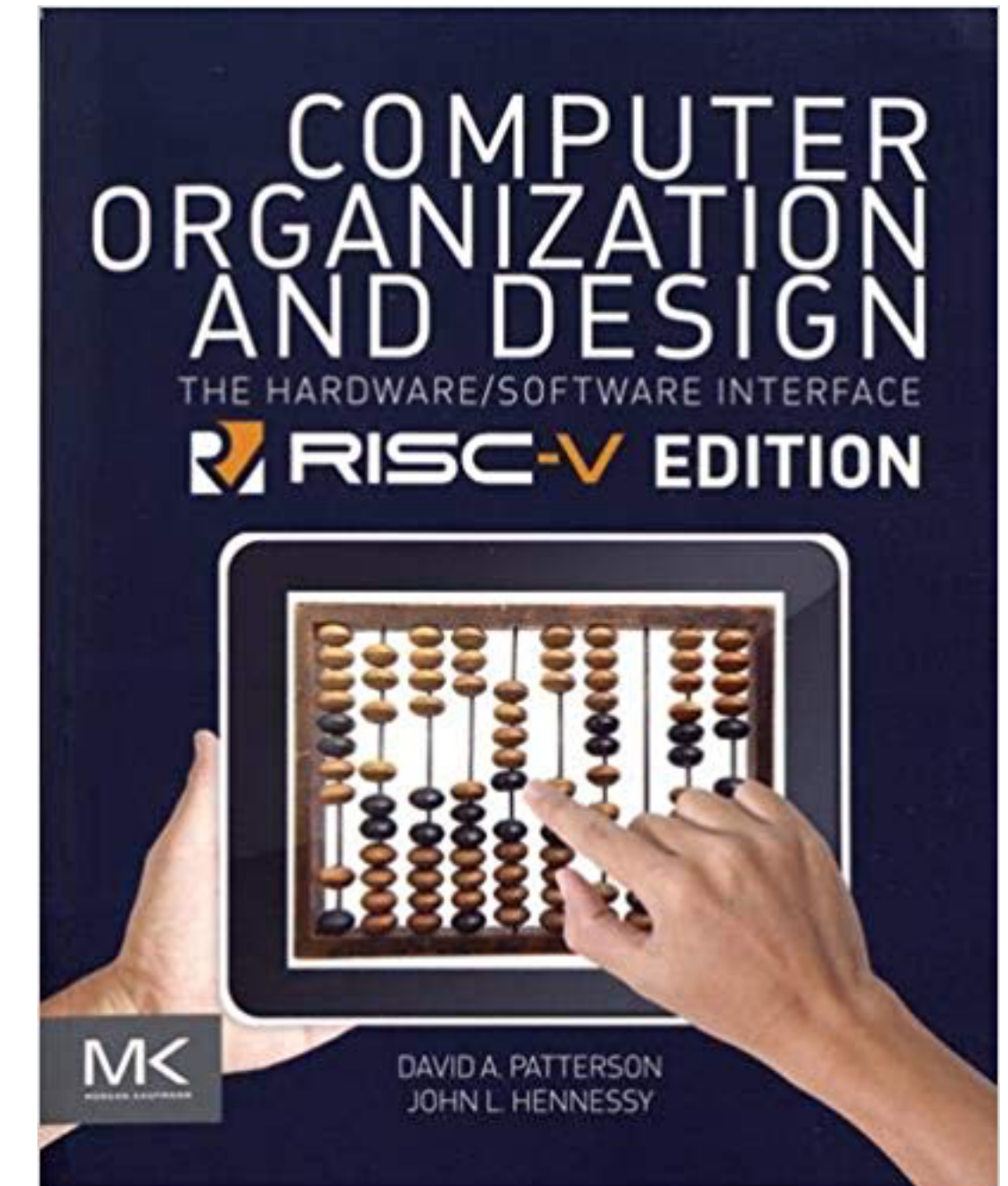


Gold, Silver, Auditors:



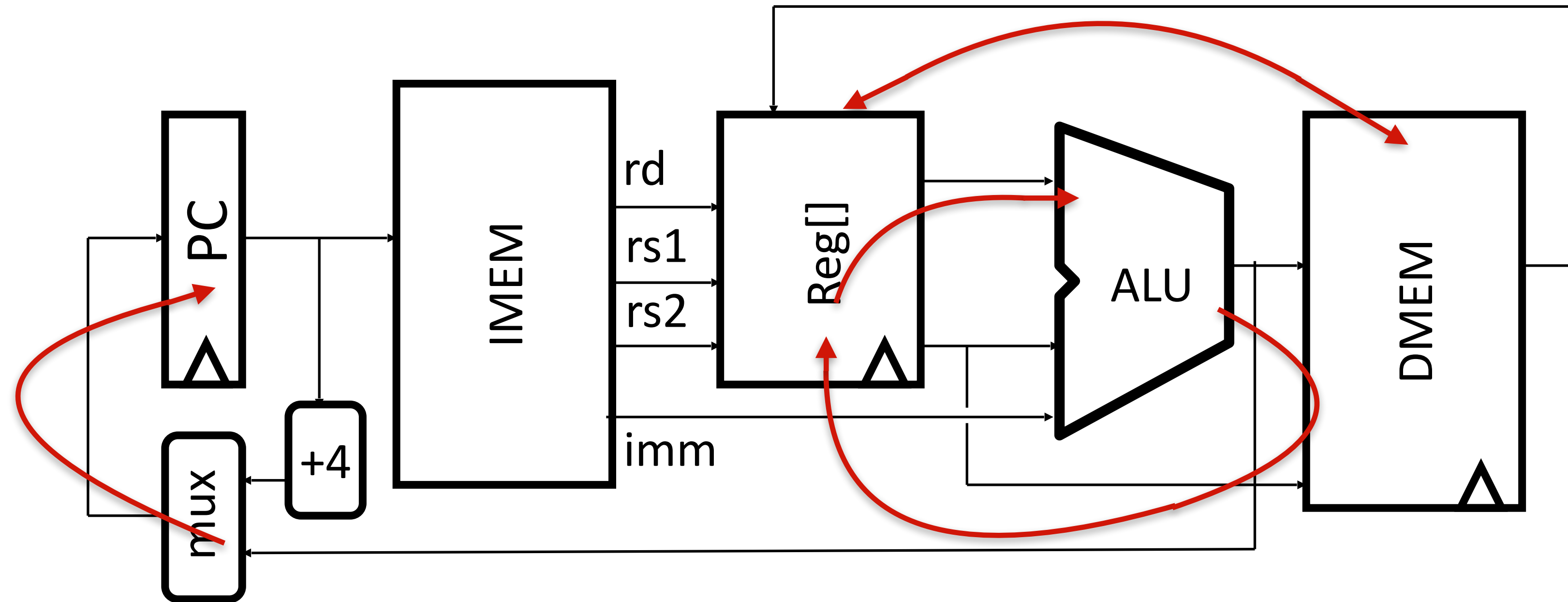
Instruction Set Architecture (ISA)

- 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.

RISC Processor Instructions in Brief



- *Compilers generate machine instructions to execute your programs in the following way:*
- Load/Store instructions move operands between main memory (cache hierarchy) and core register file.
- Register/Register instructions perform arithmetic and logical operations on register file values as operands and result returned to register file.
- Register/Immediate instructions perform arithmetic and logical operations on register file value and constants.
- Branch instructions are used for looping and if-then-else (data dependent operations).
- Jumps are used for function call and return.

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 EEC5151/251A

* implemented in the ASIC project

Summary of RISC-V Instruction Formats

Binary encoding of machine instructions. Note the common fields.

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0	
funct7		rs2			rs1		funct3		rd			opcode		R-type	
imm[11:0]						rs1		funct3		rd			opcode		I-type
imm[11:5]			rs2			rs1		funct3		imm[4:0]			opcode		S-type
imm[12]	imm[10:5]		rs2			rs1		funct3		imm[4:1]	imm[11]		opcode		B-type
imm[31:12]										rd			opcode		U-type
imm[20]	imm[10:1]			imm[11]		imm[19:12]				rd			opcode		J-type

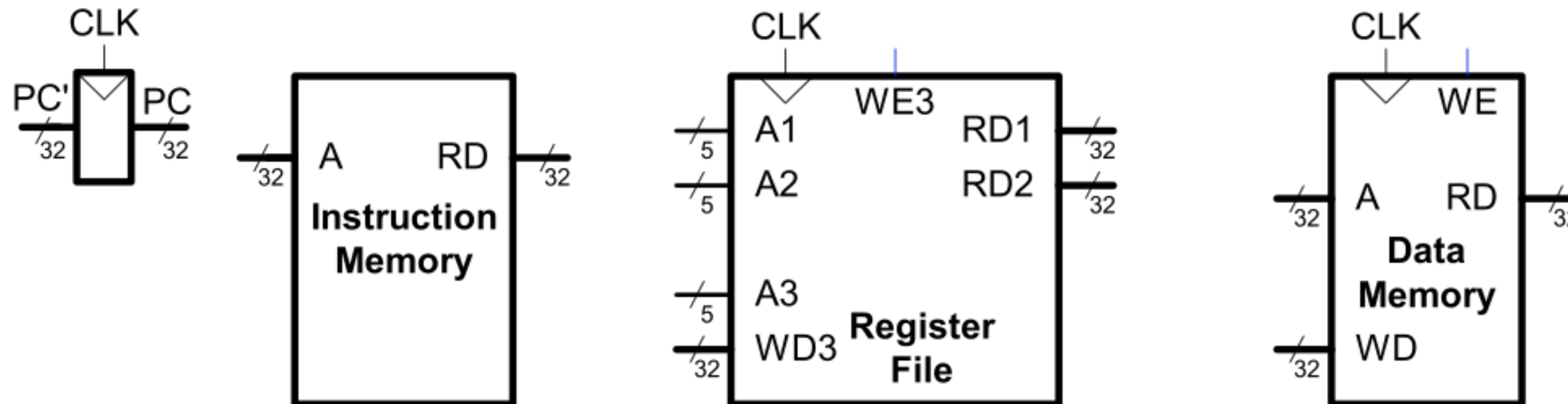
“State” Required by RV32I ISA

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

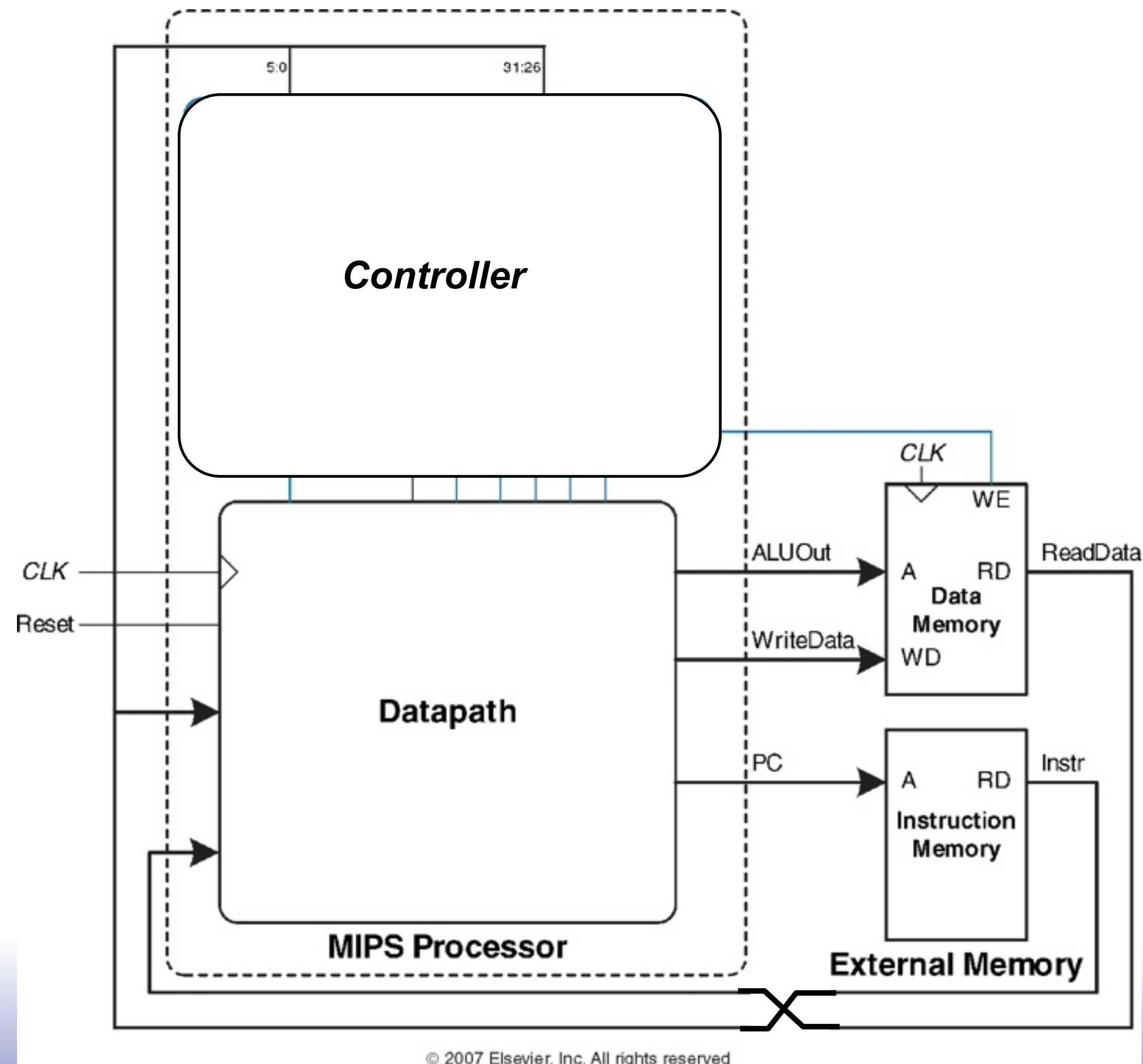
- State encodes everything about the execution status of a processor:
 - PC register
 - 32 registers
 - Memory



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

RISC-V Microarchitecture Organization

Datapath + Controller + External Memory



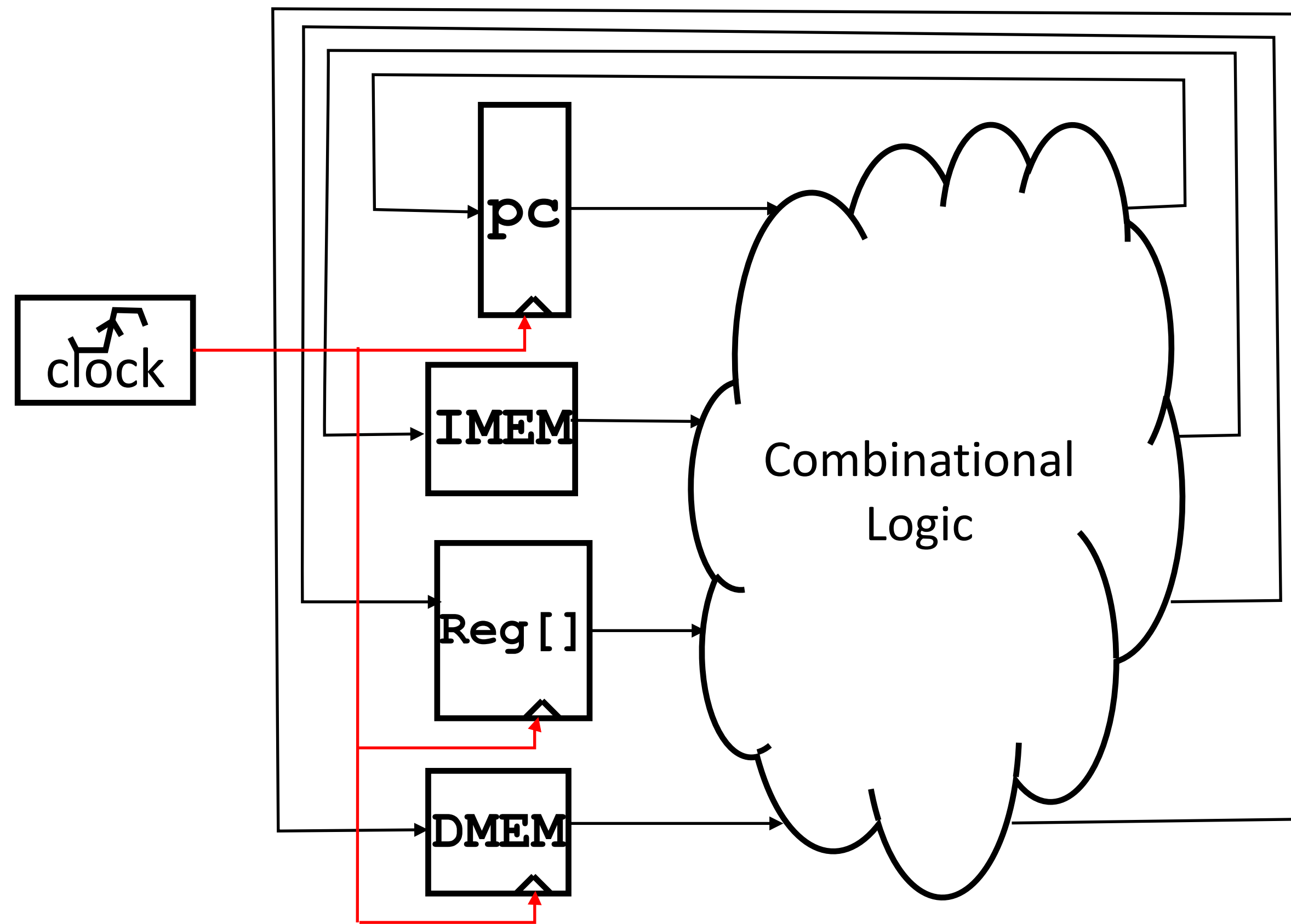
Microarchitecture

Multiple implementations for a single instruction set 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...*
 - Instructions are reordered by the hardware

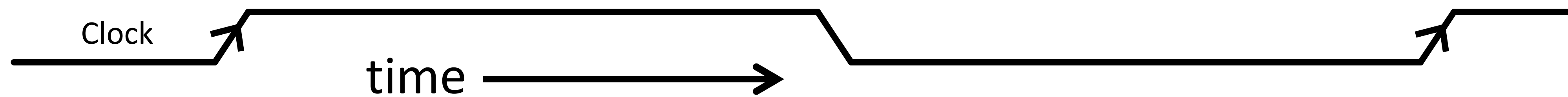
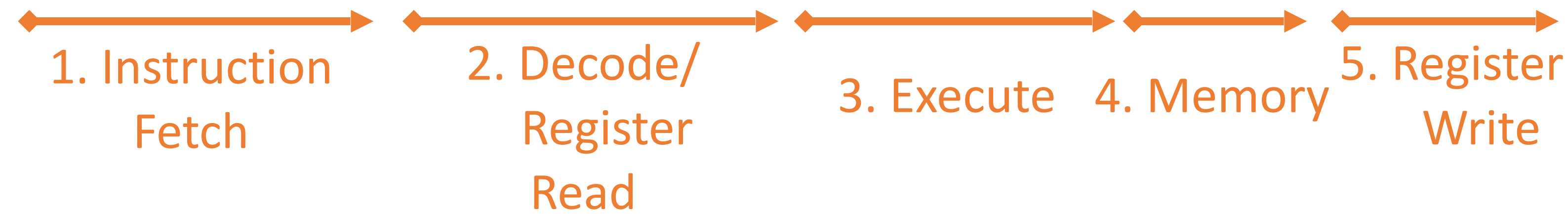
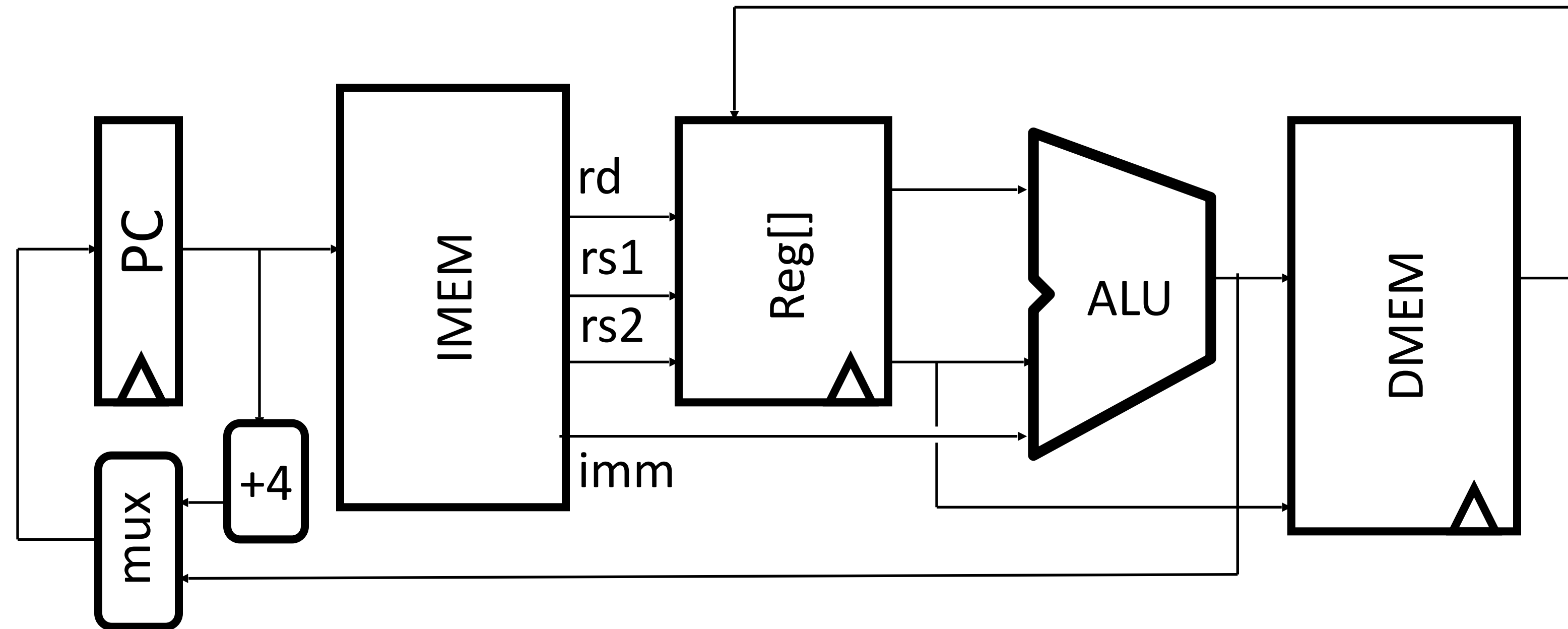
First Design: One-Instruction-Per-Cycle RISC-V Machine

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

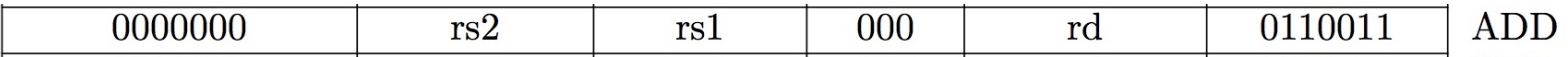


1. Current state outputs drive the inputs to the combinational logic, whose outputs settles at the values of the state before the next clock edge
2. At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle (next instruction)

Basic Phases of Instruction Execution



Implementing the `add` instruction



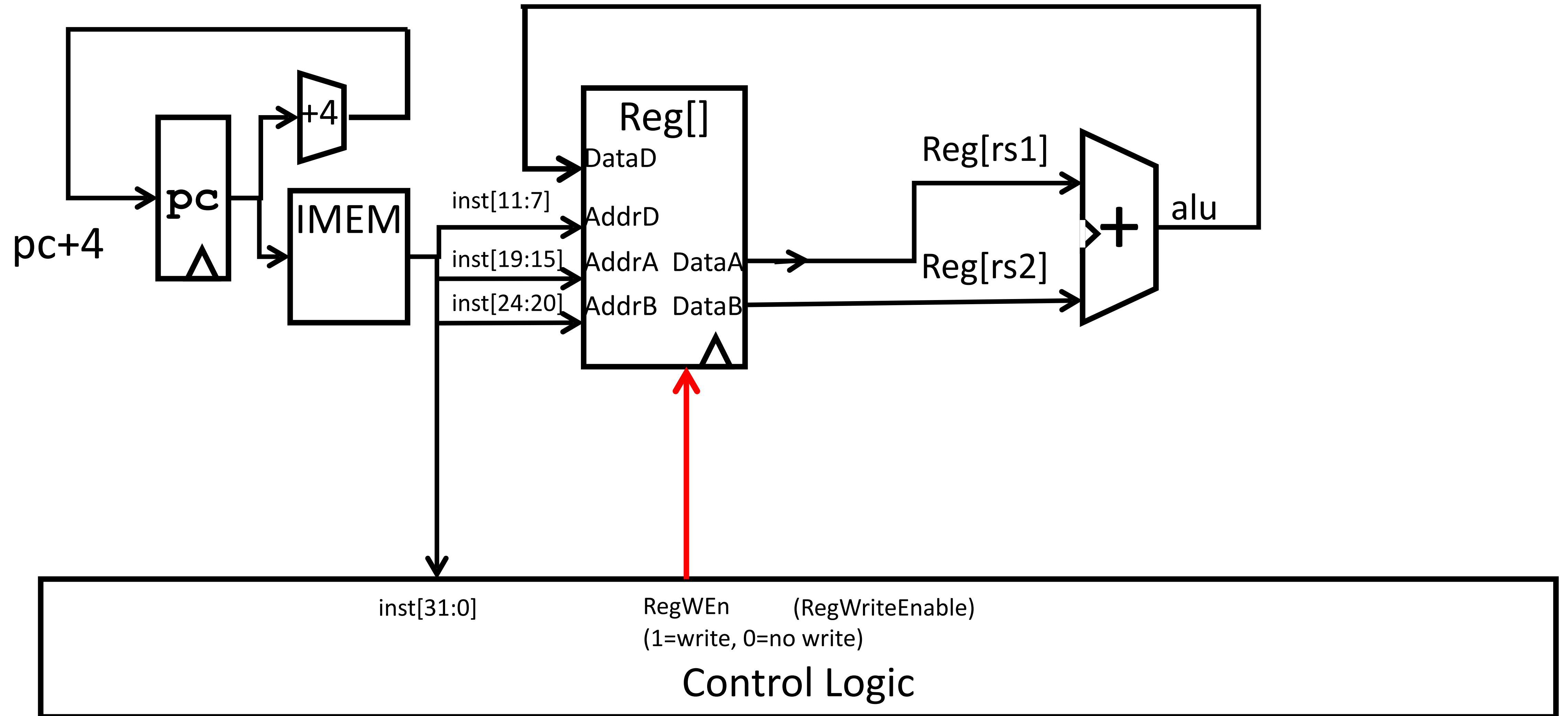
`add rd, rs1, rs2`

- Instruction makes two changes to machine's state:

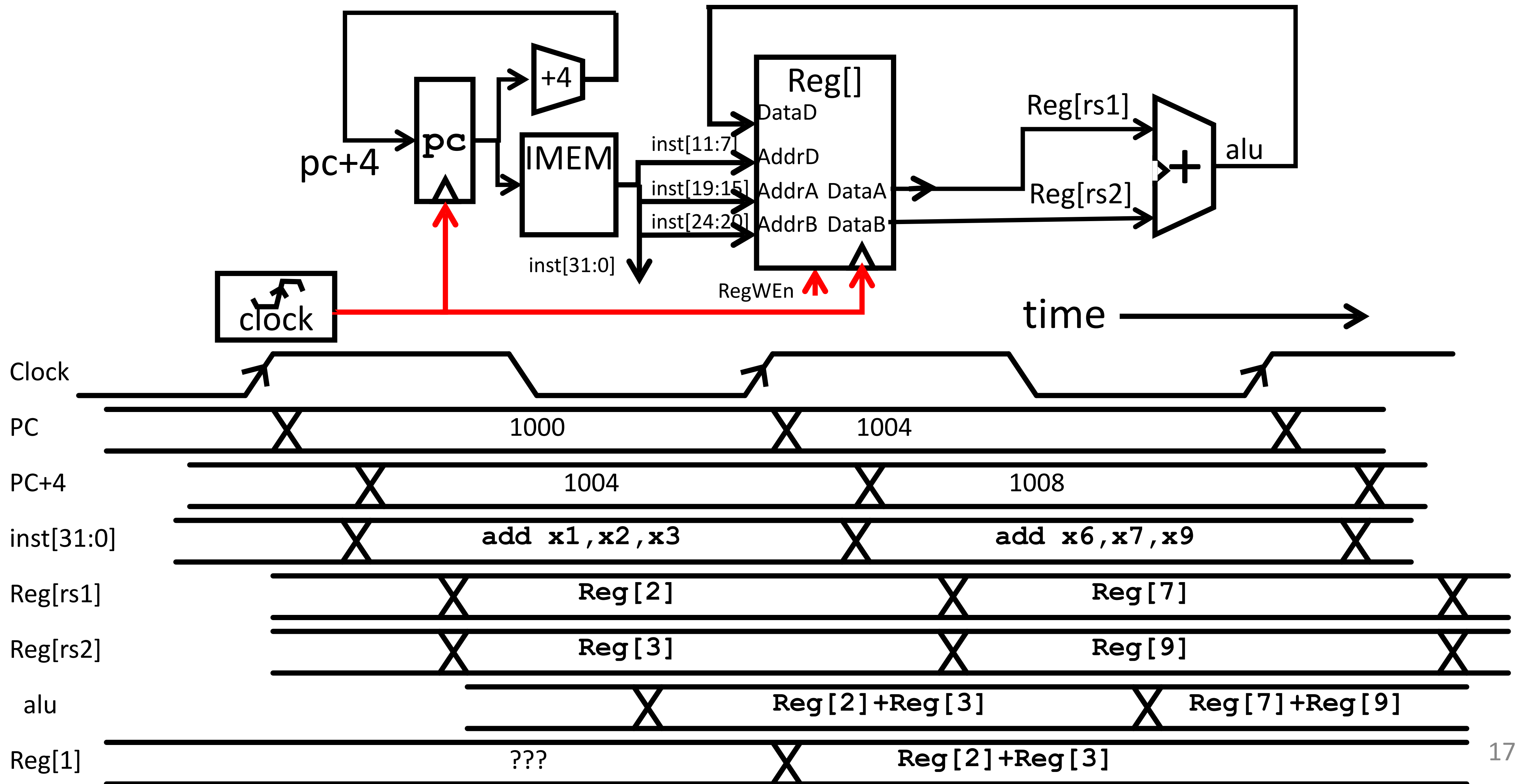
$$Reg[rd] = Reg[rs1] + Reg[rs2]$$

$$PC = PC + 4$$

Datapath for add



Timing Diagram for add



Implementing the `sub` instruction

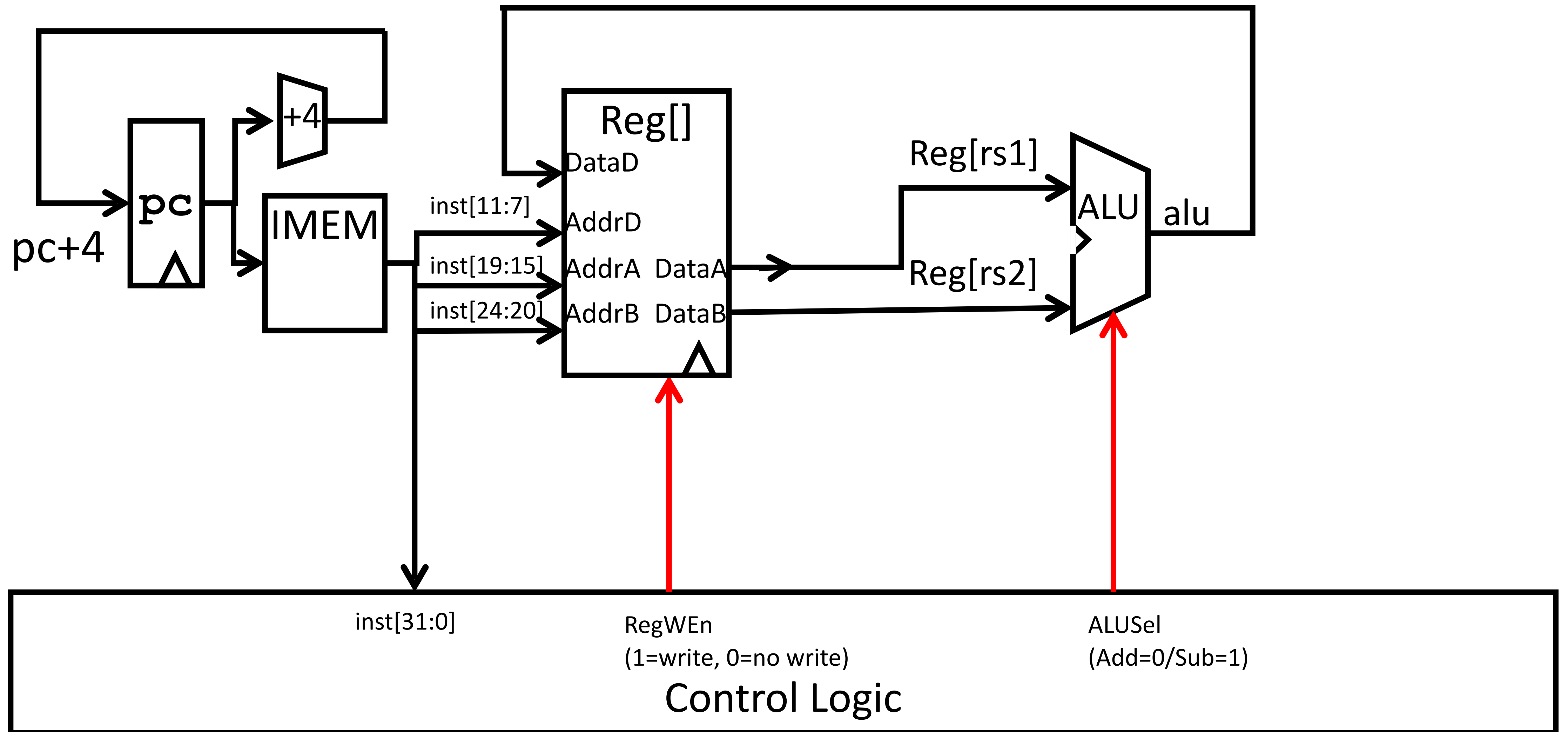
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB

`sub rd, rs1, rs2`

$$Reg[rd] = Reg[rs1] - Reg[rs2]$$

- 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

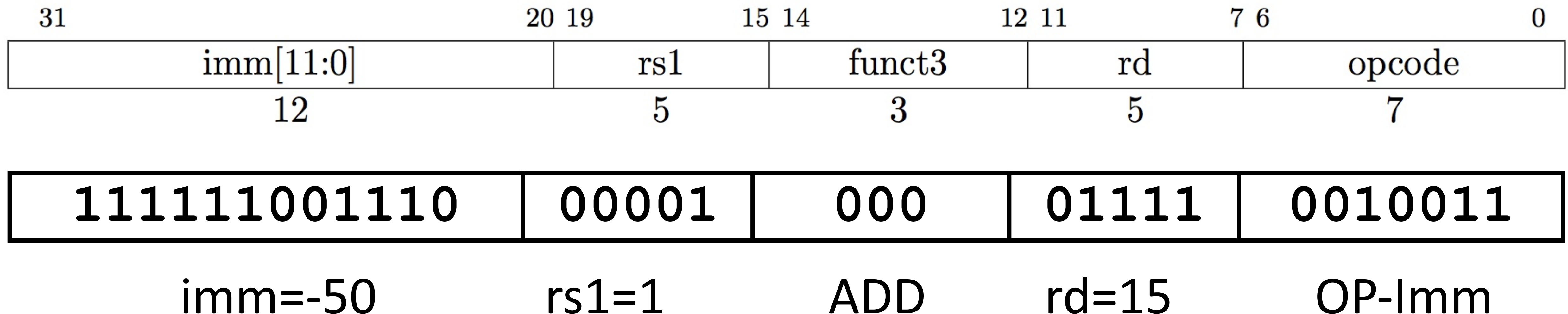
- RISC-V Assembly Instruction: *Uses the “I-type” instruction format*

`addi rd, rs1, integer`

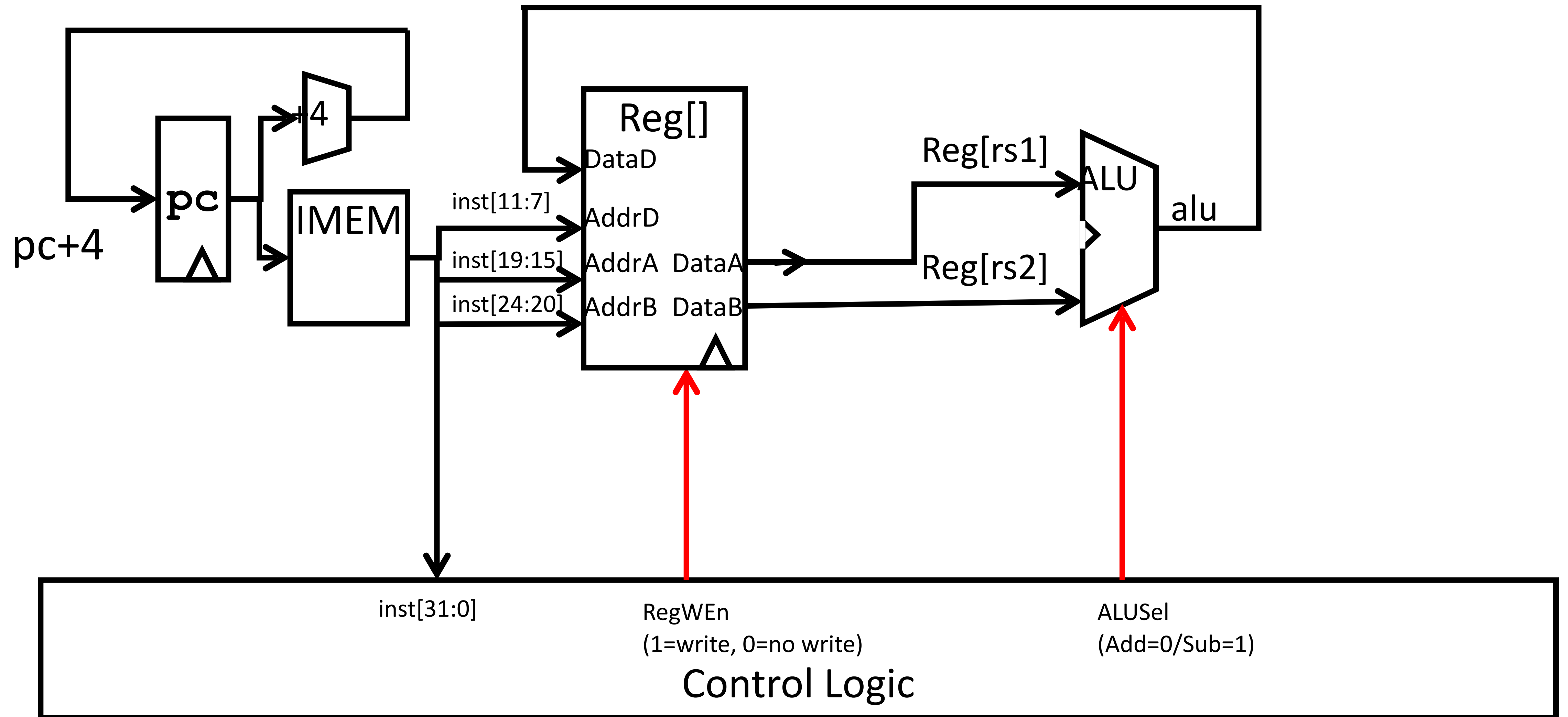
$Reg[rd] = Reg[rs1] + sign_extend(immediate)$

example:

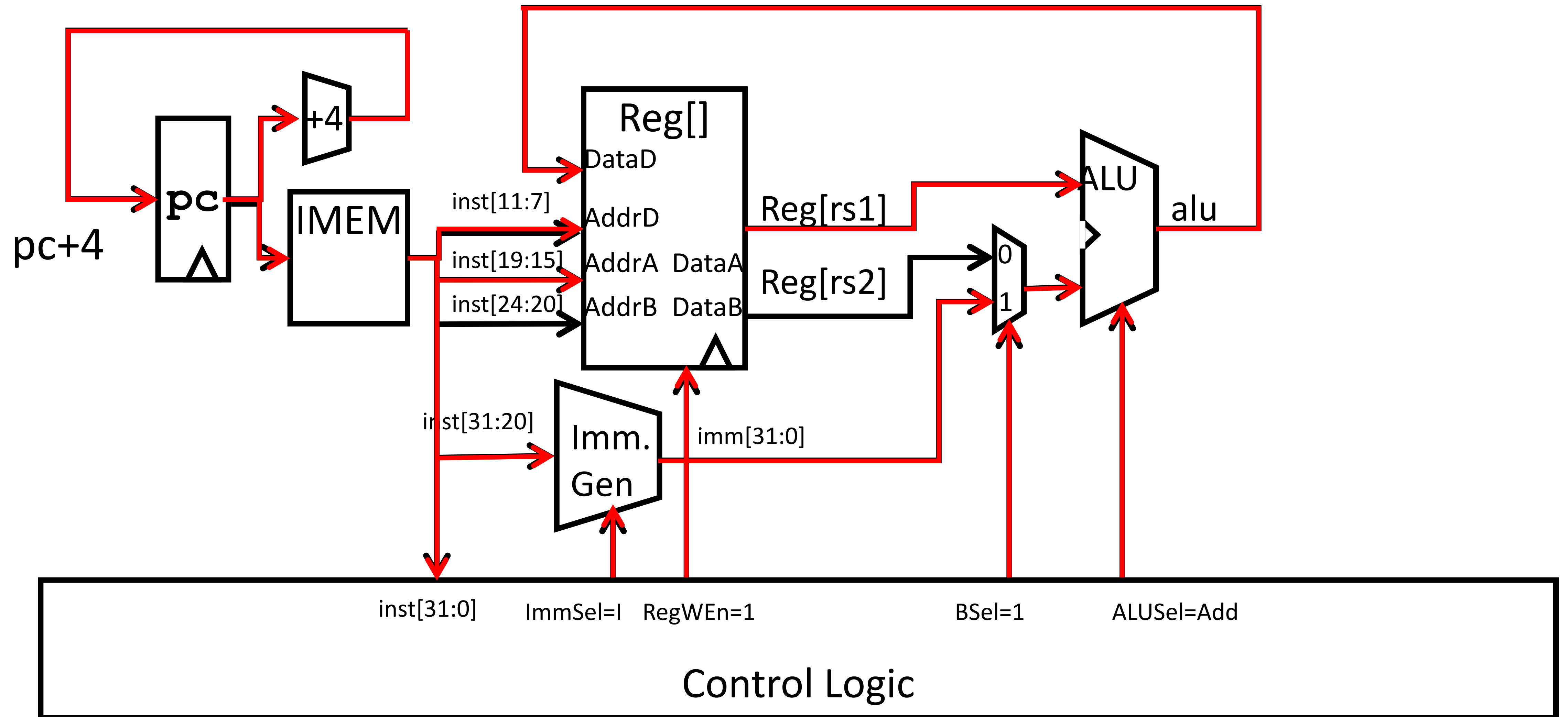
`addi x15, x1, -50`



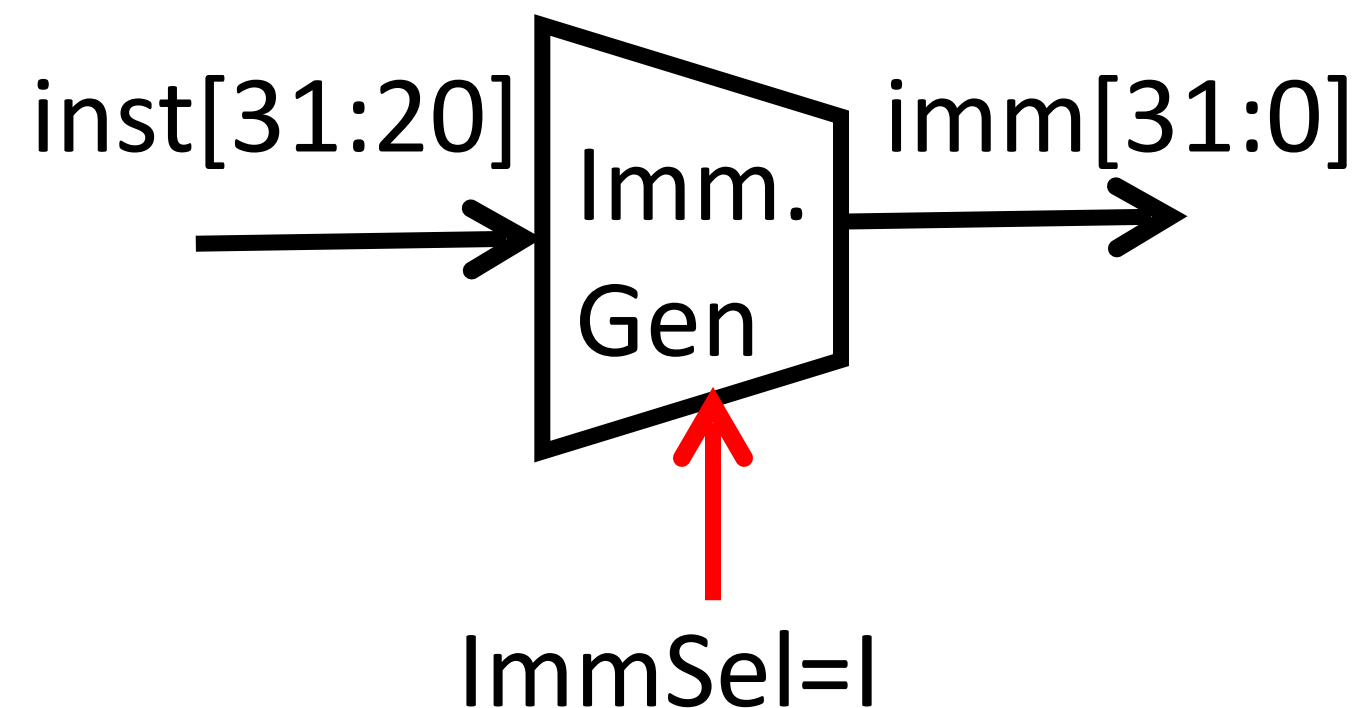
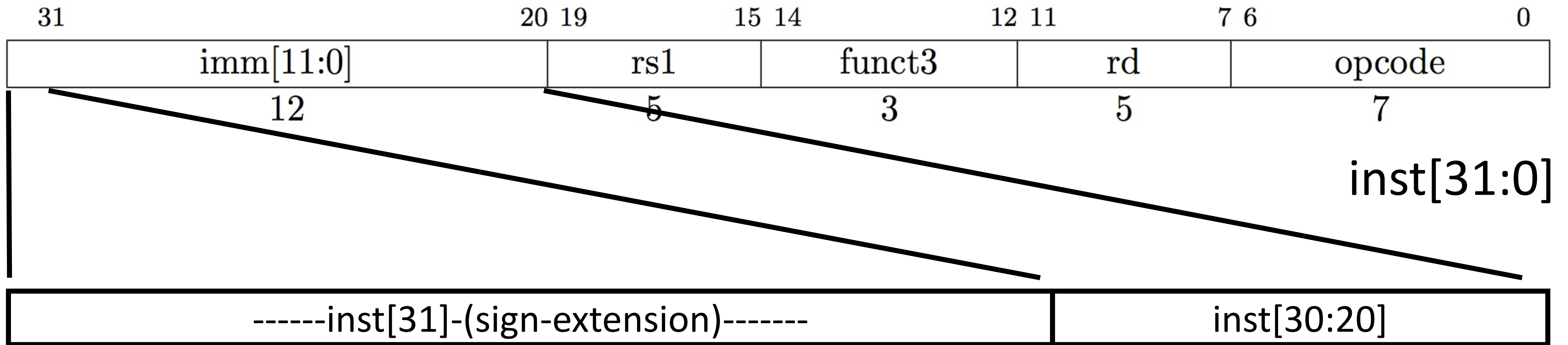
Review: Datapath for add/sub



Adding `addi` to datapath

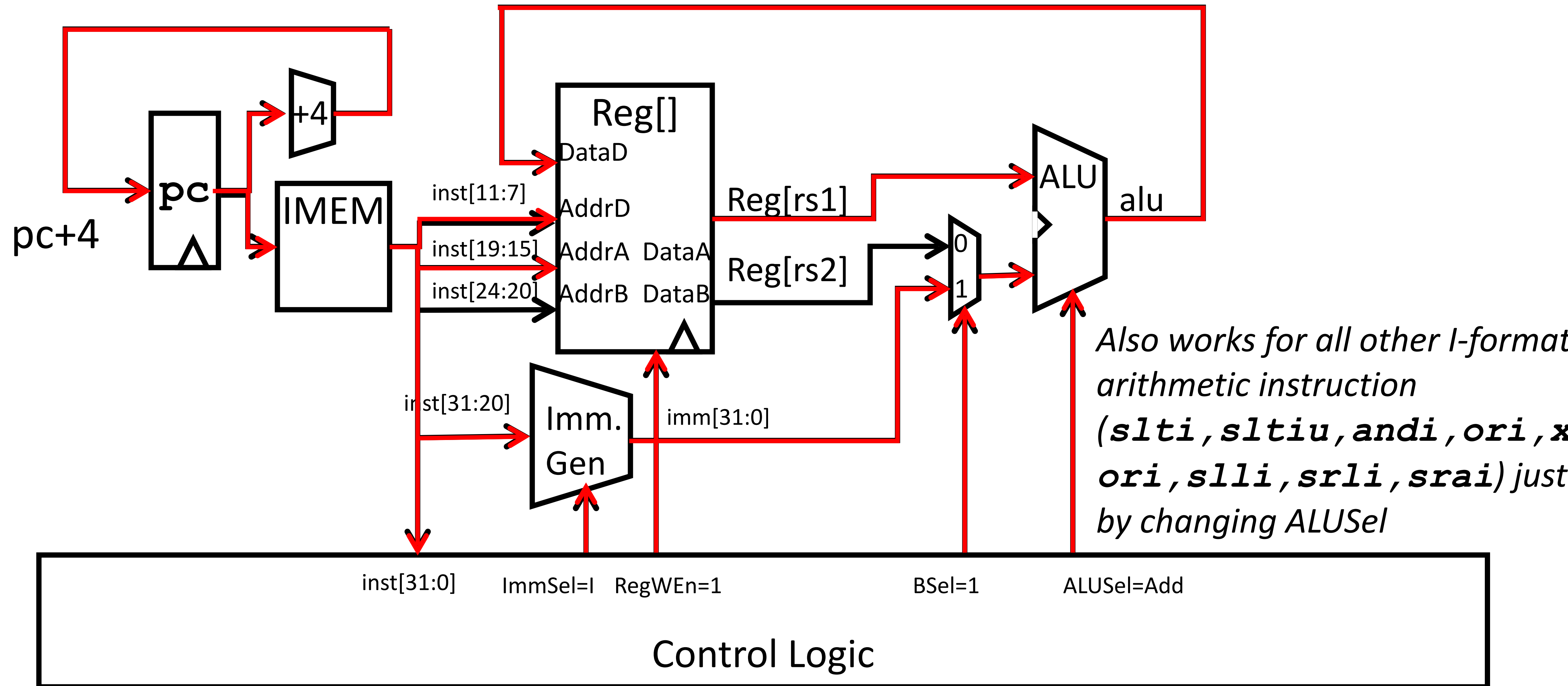


I-type Format immediates



- 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

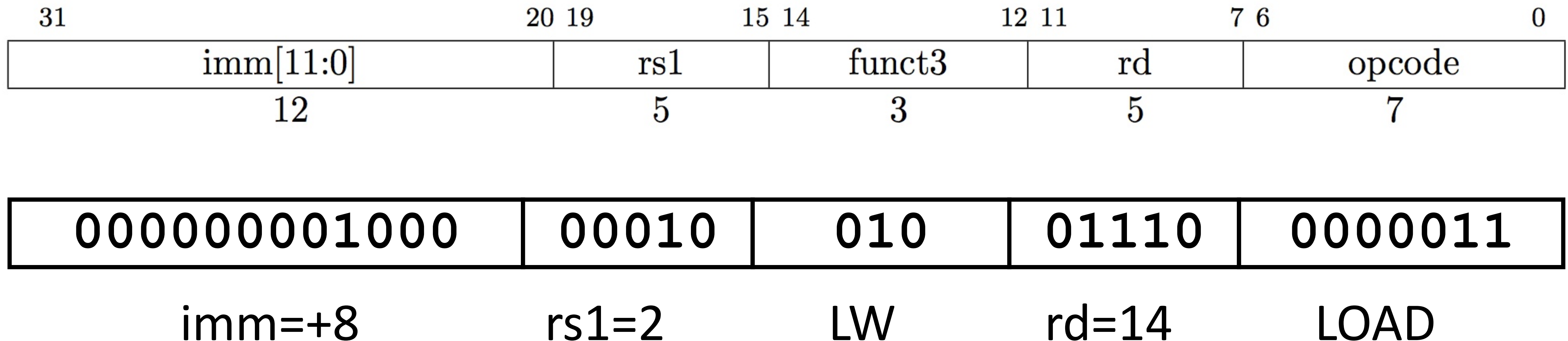
- RISC-V Assembly Instruction: *Also uses the “I-type” instruction format*

lw rd, integer(rs1)

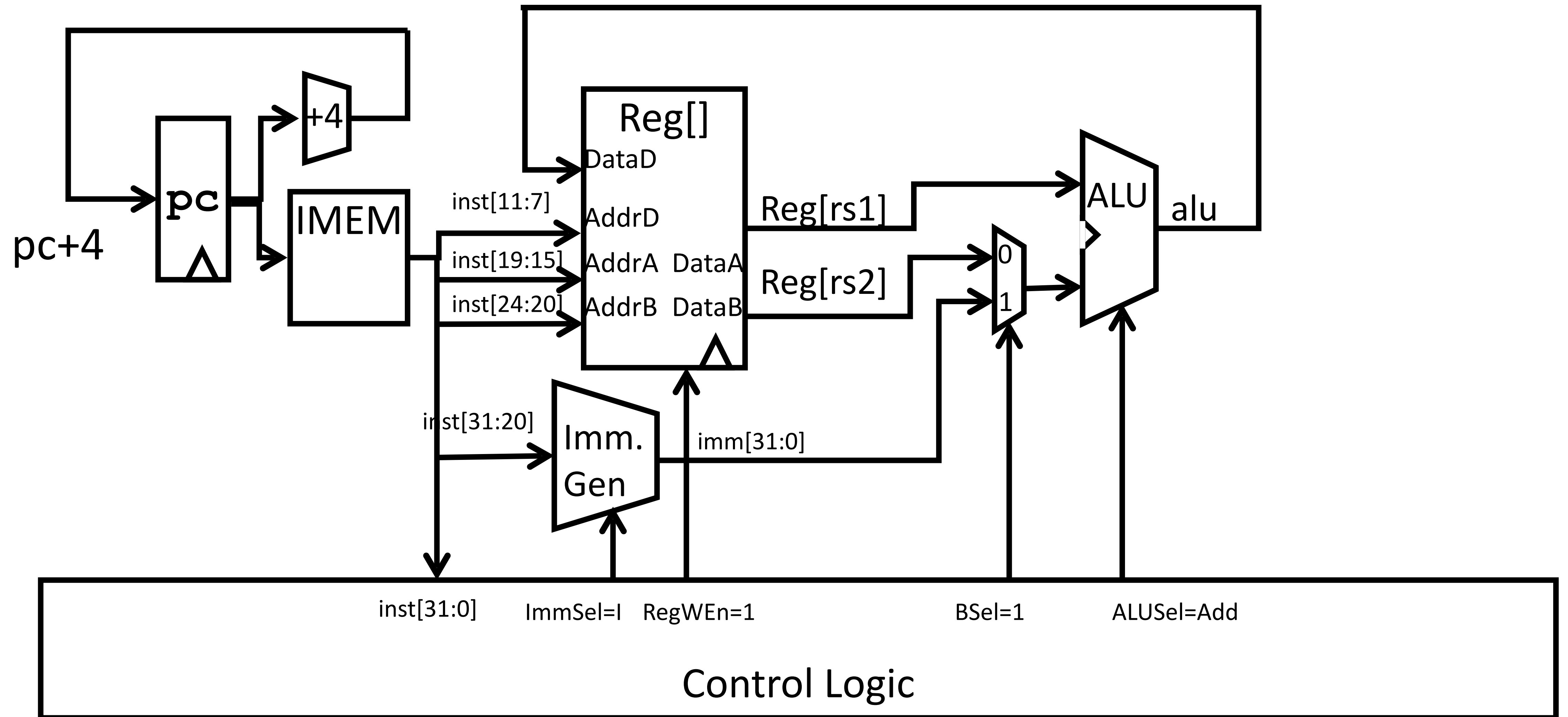
$Reg[rd] = DMEM[Reg[rs1] + sign_extend(immediate)]$

example:

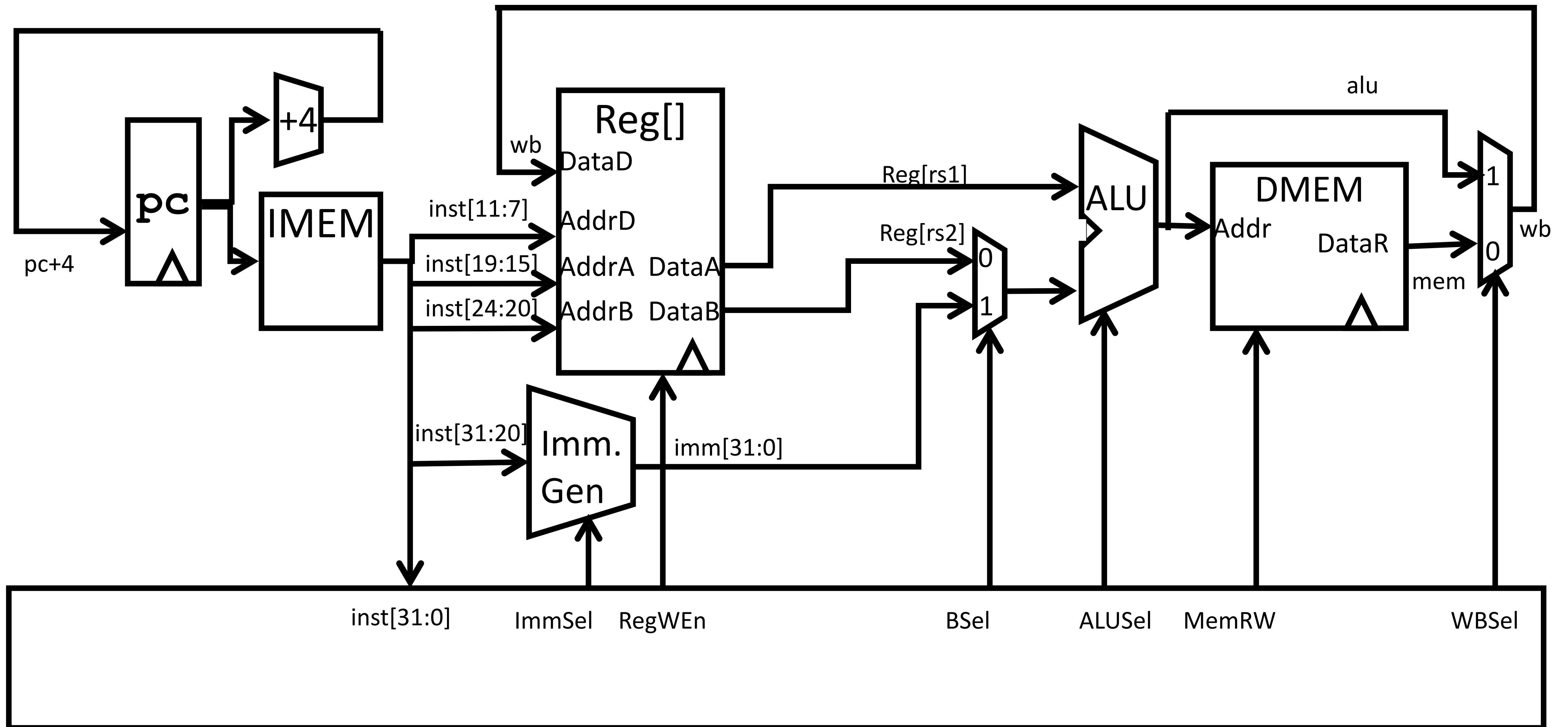
lw x14, 8(x2)



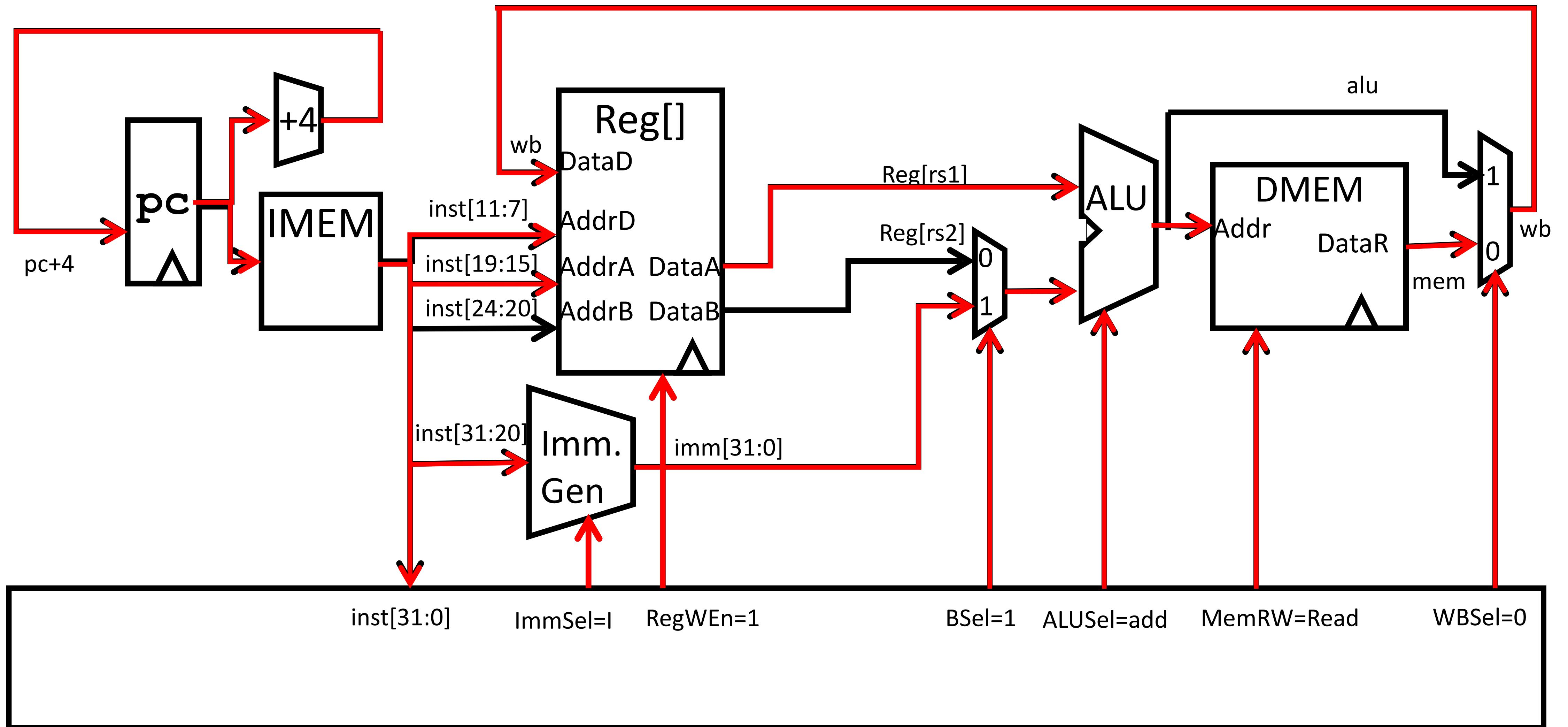
Review: Adding `addi` to datapath



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

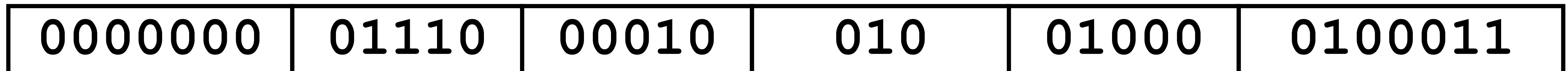
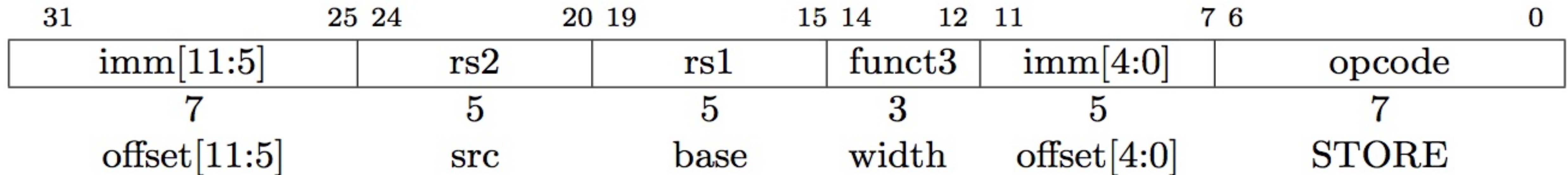
- RISC-V Assembly Instruction: *Uses the "S-type" instruction format*

sw rs2, integer(rs1)

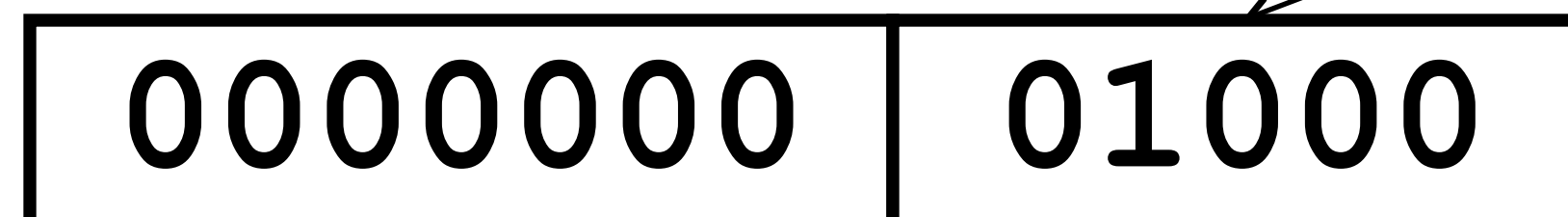
$DMEM[Reg[rs1] + sign_extend(immediate)] = Reg[rs2]$

example:

sw x14, 8(x2)

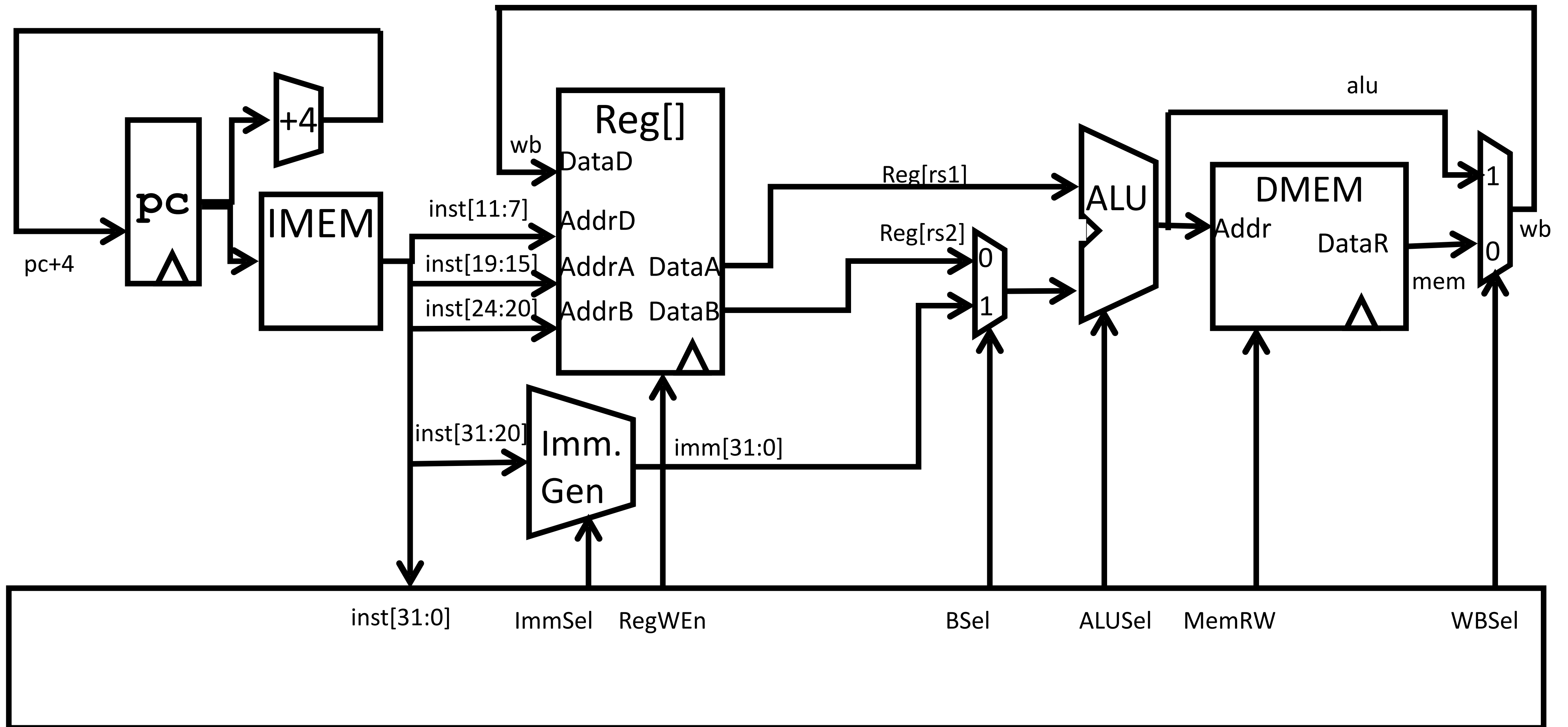


offset[11:5] = 0 rs2=14 rs1=2 SW offset[4:0] = 8 STORE

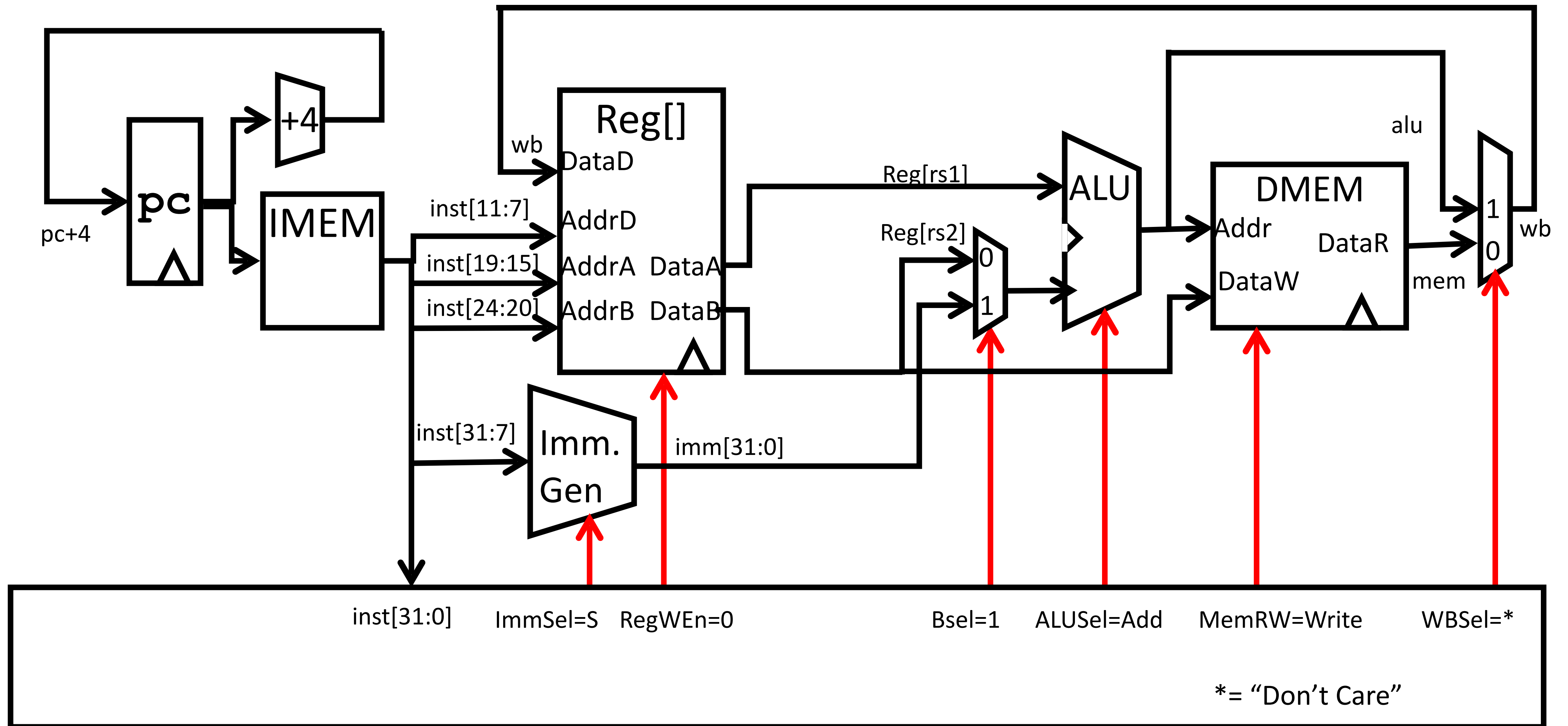


combined 12-bit offset = 8

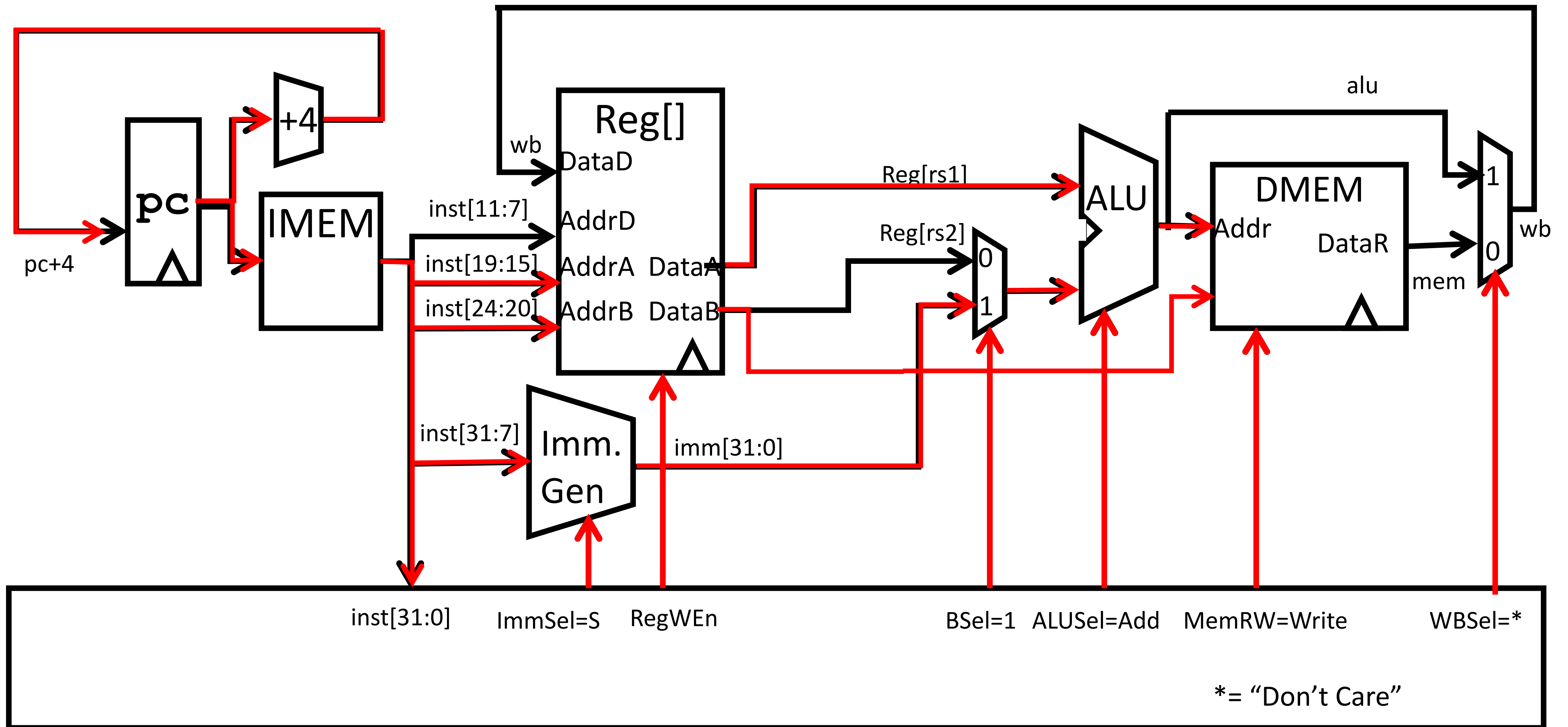
Review: Adding `lw` to datapath



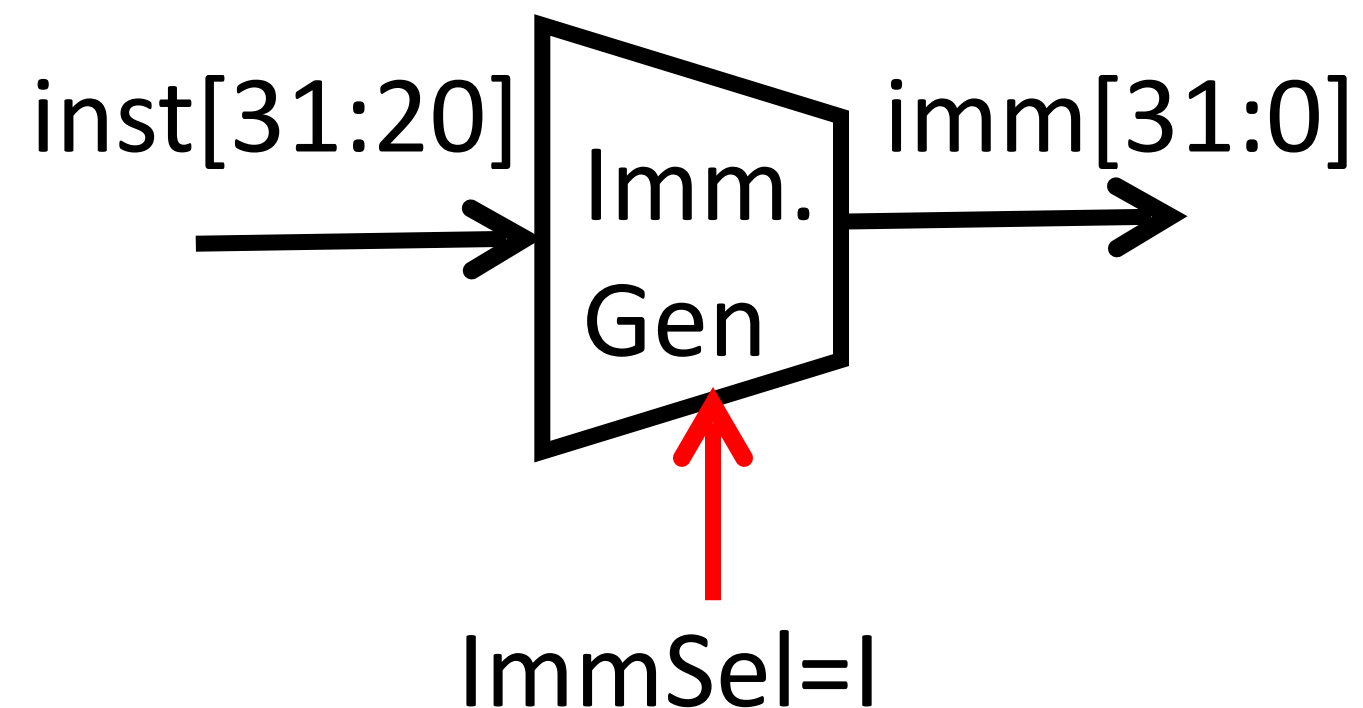
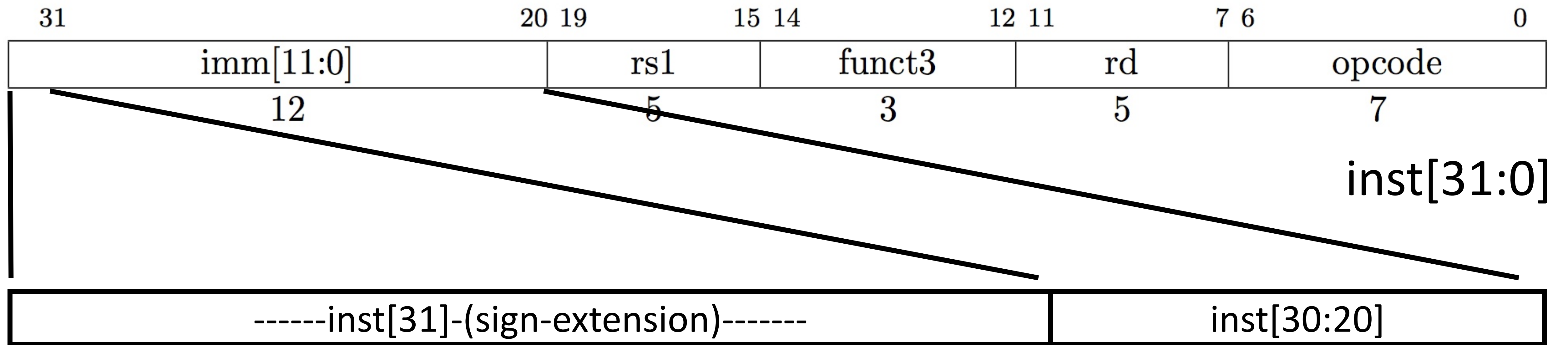
Adding sw to datapath



Adding *sw* to datapath

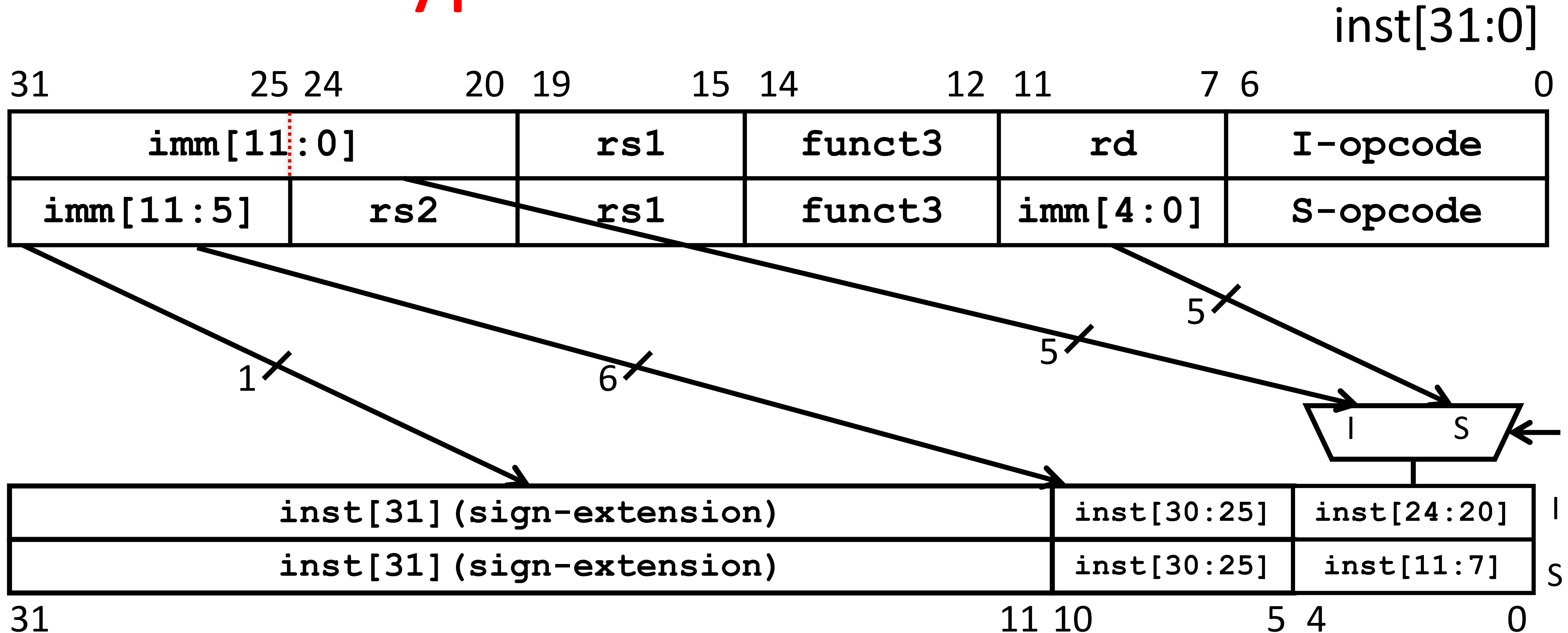


Review: I-Format immediates



- 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 -type 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

End of Lecture 13