Bridge of Life Education

SOC Design
Processor Single Cycle

Jiin Lai

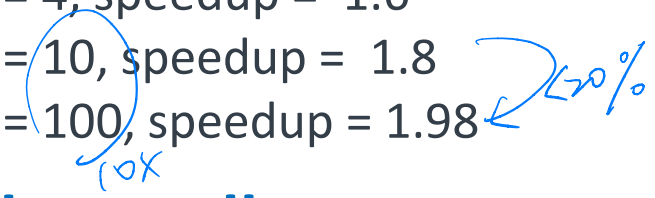
Topics

1. The Reasoning of RISC Processor Design
2. ISA & Single Cycle Processor
3. Multi-Cycle Processor
4. Pipelined Processor
5. Superscalar

Amdahl's Law

$$speedup = \frac{time_{without\_enhancement}}{time_{with\_enhancement}} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

- Diminishing returns
  - Incremental improvements in speedup gained by enhancing just one portion of our design diminish as improvements are made. Eventually, we reach the limit `1/(1-Fraction_enh)`.
  - E.g., If Fraction_enh = 0.5
    Speedup_enh = 2, speedup = 1.33
    Speedup_enh = 4, speedup = 1.6
    Speedup_enh = 10, speedup = 1.8
    Speedup_enh = 100, speedup = 1.98



How do we allocate our resources?

Making the Common-case Fast

- Not every improvement equal
- Adding an enhancement may disadvantage the common case:
  - Less time can be spent optimizing the common case.
  - Accelerating complex operations may require the cycle-time to be extended. This will slow all operations.
  - Resources (transistors, power) are redirected to less important (less often used) features.
- We use **benchmarks** to determine what the common cases are and to help guide the design process.
- Common Techniques used in processor design
  - Locality
  - Speculation
  - Prediction
  - Indirection*
  - Parallelism
  - Specialization

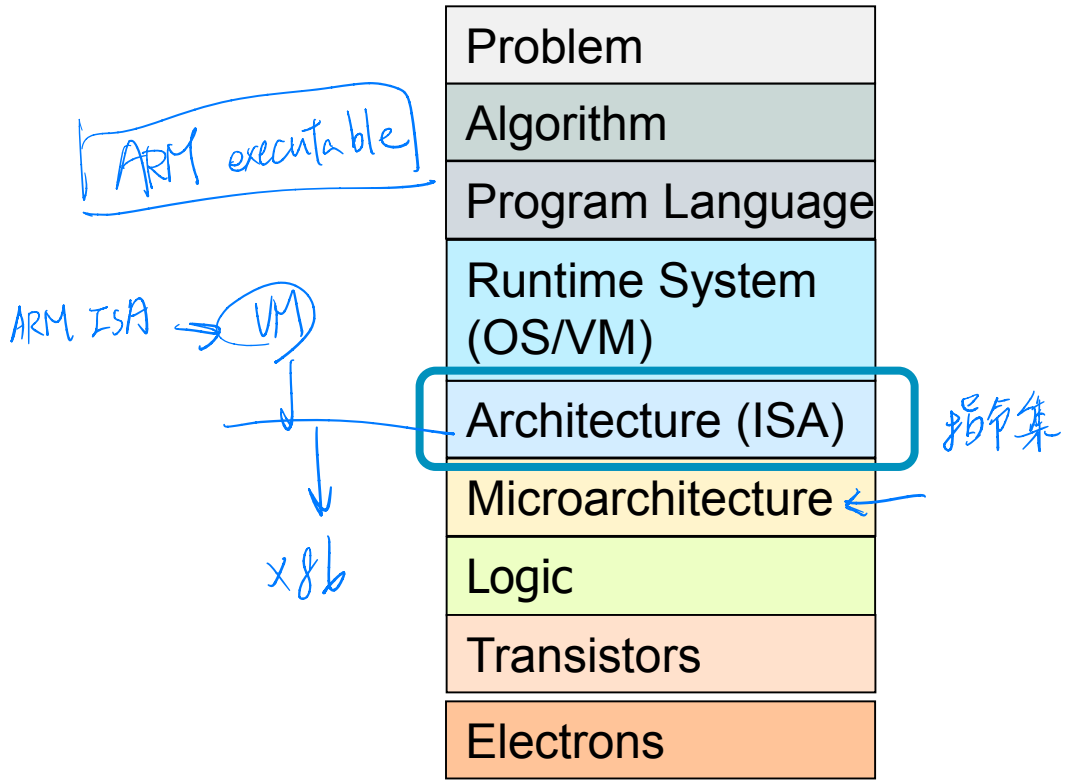
Instruction Set Architecture (ISA)

- ISA is typically seen as the contract between software and hardware.

Instruction Set Architecture

- instructions
- registers
- memory addressing
- addressing modes
- etc.

The computing abstraction stack



The RISC Approach

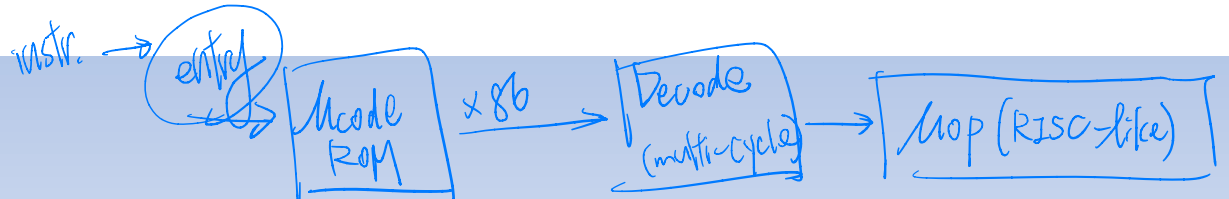
- The RISC approach aims to ensure that we **make the common-case fast** by carefully selecting the most useful instructions and addressing modes, etc.
- Instructions are designed to make good use of the register file.
- A RISC ISA is designed to ensure a simple high-performance implementation is possible.

Common features of RISC instruction sets:

- Fixed length instruction encodings (or a small number of easily decoded formats)
- Each instruction follows similar steps when being executed.
- Access to data memory is restricted to special load/store instructions (a so-called **load/store architecture**).

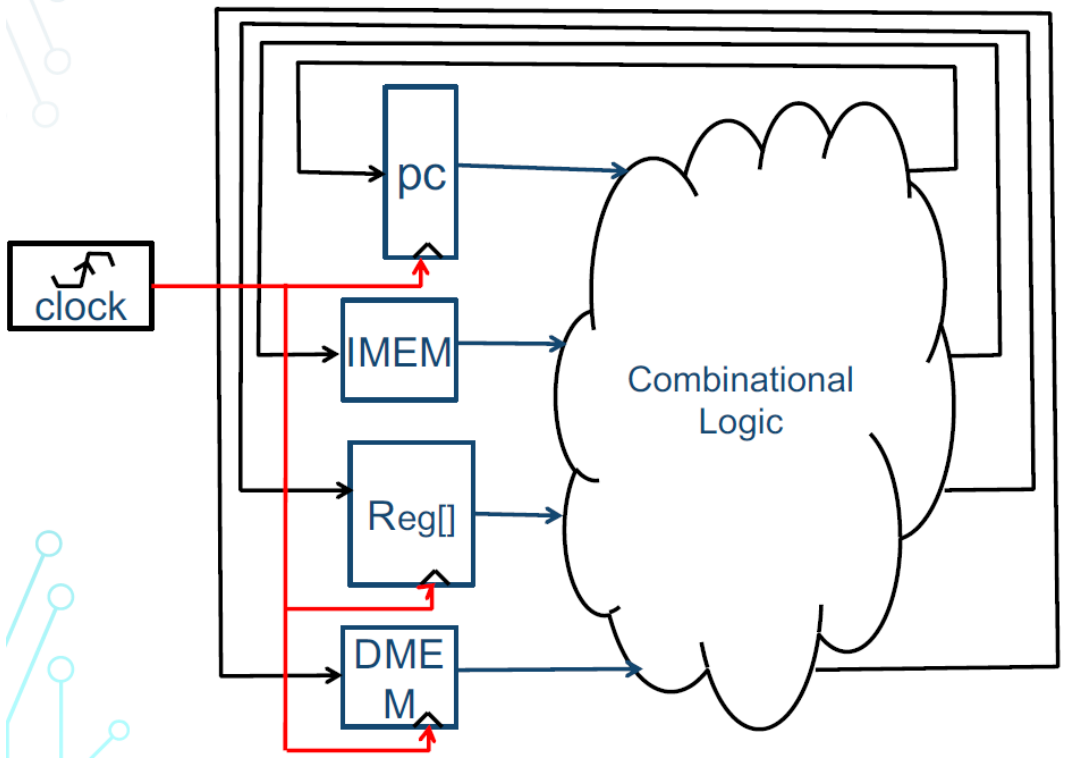
The CISC approach

- A single instruction can be used to do all of the loading, evaluating and storing operations.
- To minimize the number of instructions per program. However, that increases the number of cycles per instruction.



One-Instruction-Per-Cycle RISC-V Machine

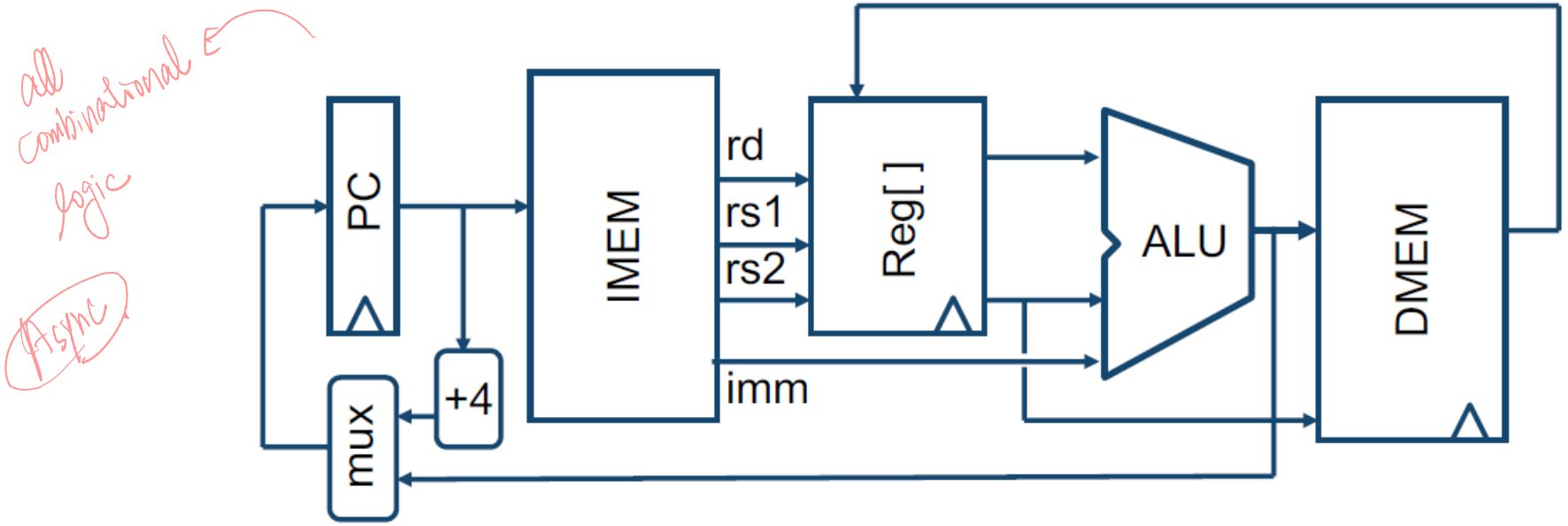
- Instruction is executed in one clock
- Current state drive the inputs to the combinational logic, whose outputs settles before the next clock edge
- At the rising clock edge, all the state elements are updated with the combinational logic outputs
- Separate instruction/data memory: memory is asynchronous read (not clocked), but synchronous write (is clocked)

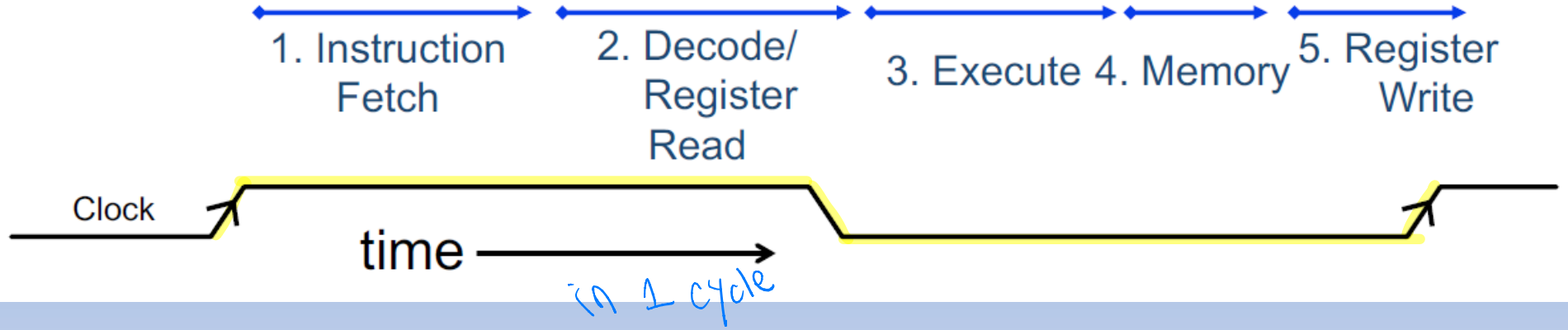


Processor States

- Registers (**x0..x31**)
- Program counter (**PC**)
- Memory (**IMEM, DMEM**)
  - Instructions are read (*fetched*) from IMEM
  - Load/store instructions access DMEM

Basic Phases of Instruction Execution

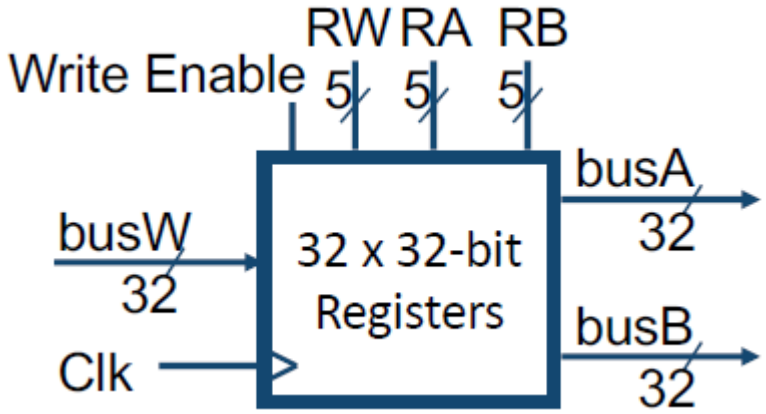




Datapath Components: Register Files (x0- x31)

rs1, rs2, rd
read write

- Register file (regfile, RF) consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
  - x0 is wired to 0
- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
- Clock input (clk)
  - Clk input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid => busA or busB valid after “access time.”

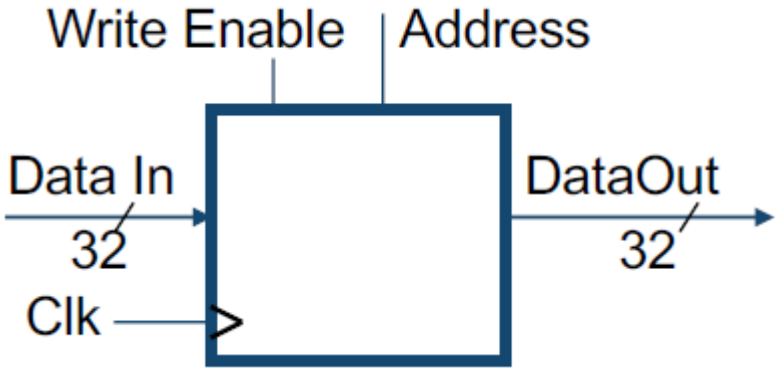



```
module rv32i_regs (
        input clk, wen,
        input [4:0] rw,
        input [4:0] ra,
        input [4:0] rb,
        input [31:0] busw,
        output [31:0] busa,
        output [31:0] busb
);
        reg [31:0] regs [0:31];
        always @(posedge clk)
                if (wen) regs[rw] <= busw;
        assign busa = (ra == 5'd0) ? 32'd0: regs[ra];
        assign busb = (rb == 5'd0) ? 32'd0: regs[rb];
endmodule
```

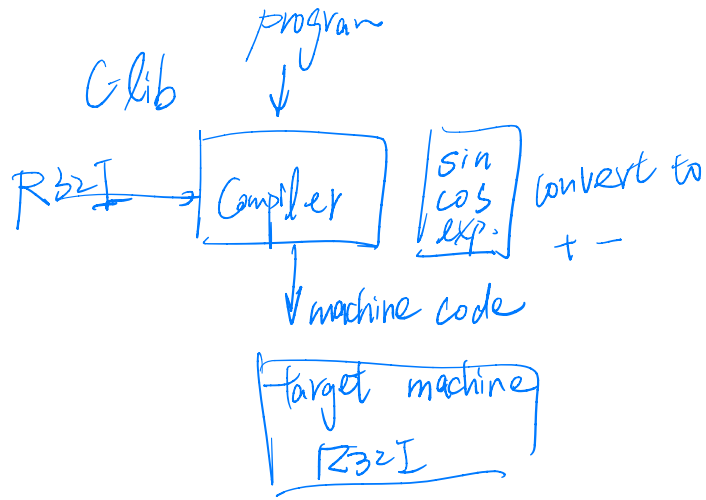
Datapath Component: Memory

- "Magic" memory
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is found by:
  - For Read: Address selects the word to put on Data Out
  - For Write: Set Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:

    Address valid => Data Out valid after "access time"

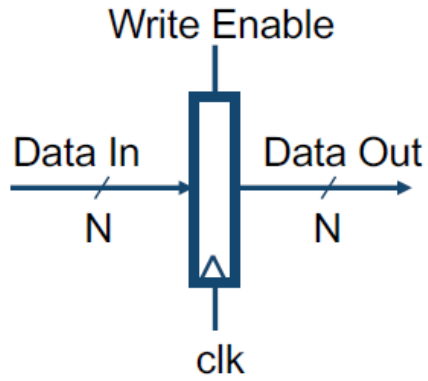


Datapath Components – ALU, MUX,

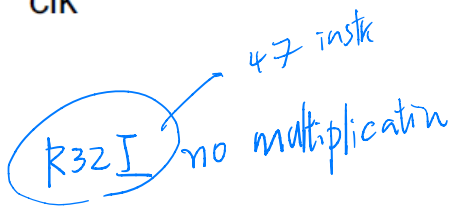


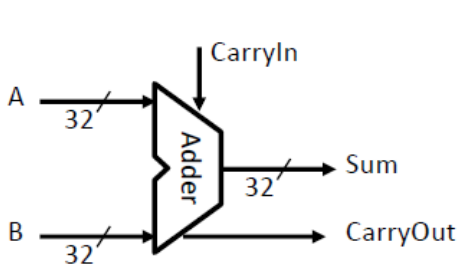
Registers

```
always @(posedge clk)
        if (wen) dataout <= datain;
endmodule
```

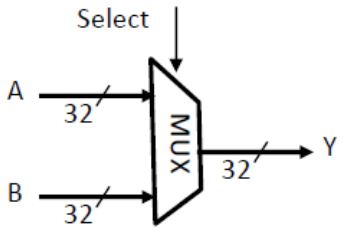


Combinational Elements – No clock

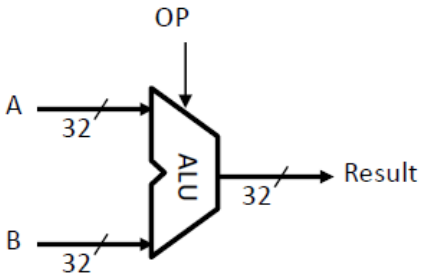




Adder



Multiplexer



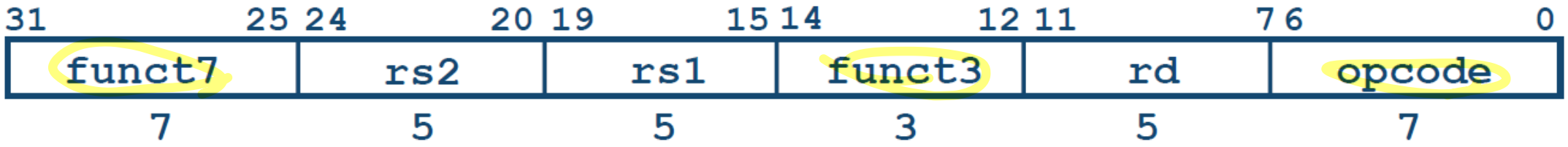
ALU

RISC-V Datapath & Control

- **R-type – Register-to-Register**
- **I-type - Immediate**
- **S-type**
- **B-type**
- **J-type**
- **U-type**
- **Control Logic**

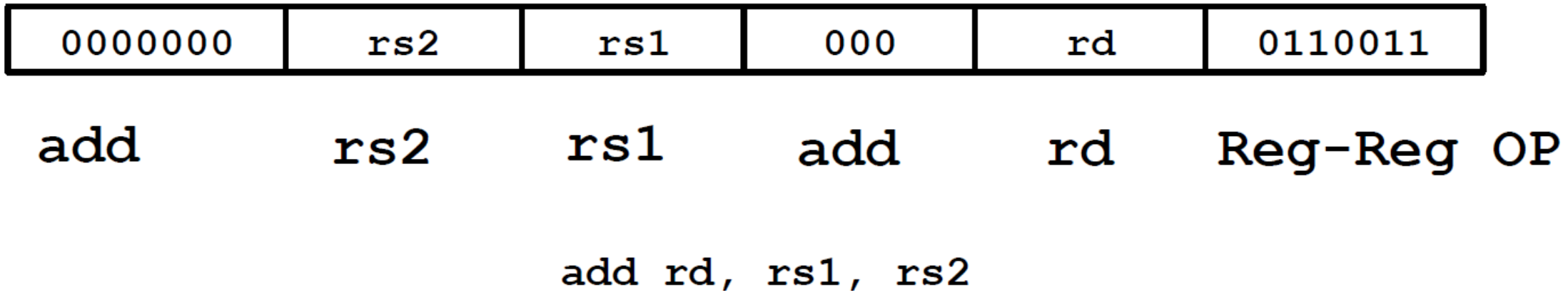
R-Format Instructions opcode/funct fields

- **opcode**: partially specifies what instruction it is
  - Note: This field is equal to **0110011** for all R-Format **register-register arithmetic** instructions
- **funct7+funct3**: combined with **opcode**, describe what operation to perform



- **rs1** (Source Register #1): specifies register containing first operand
- **rs2** : specifies second register operand
- **rd** (Destination Register): specifies register which will receive result of computation
- Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (**x0**-**x31**)

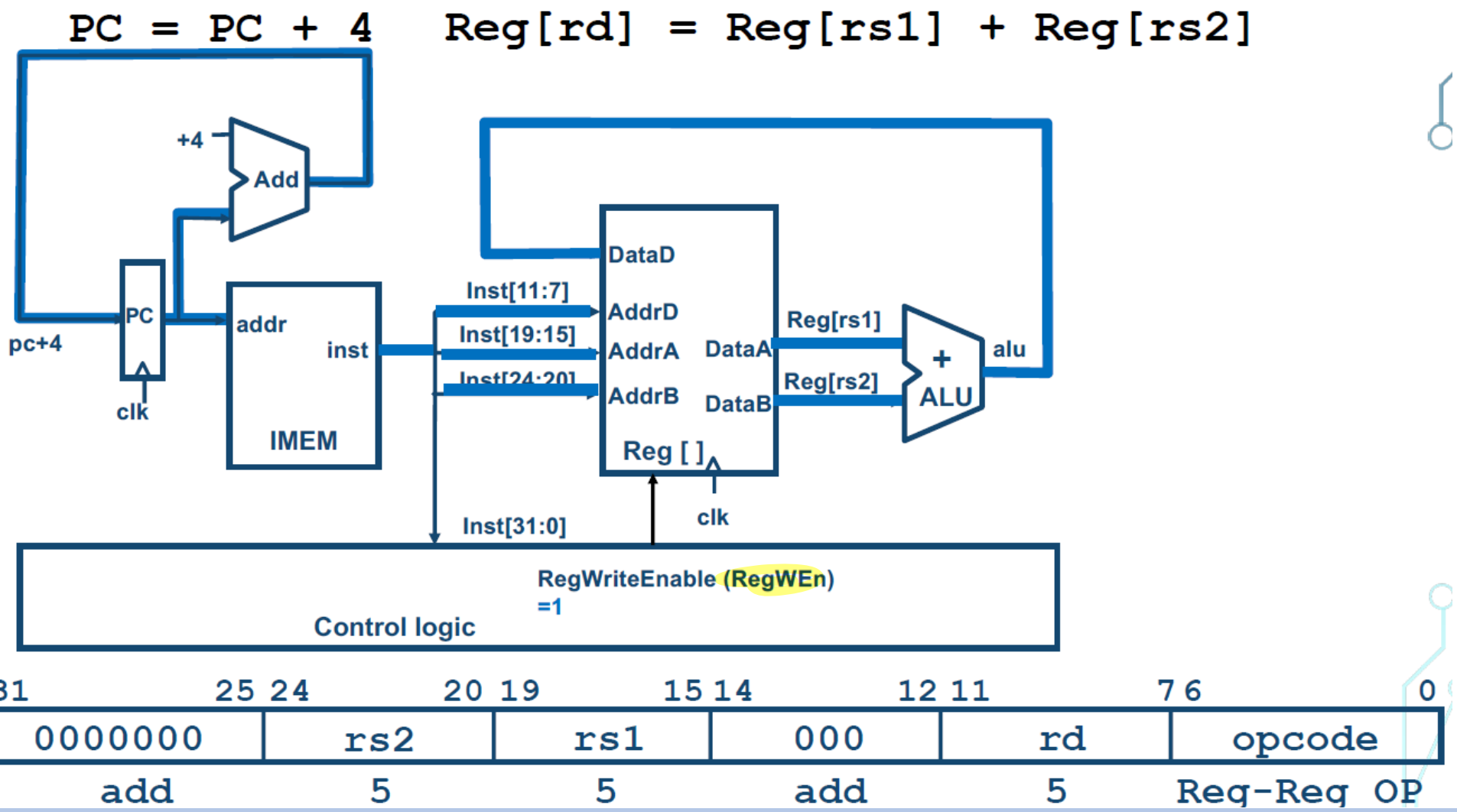
Implementing the **add** instruction



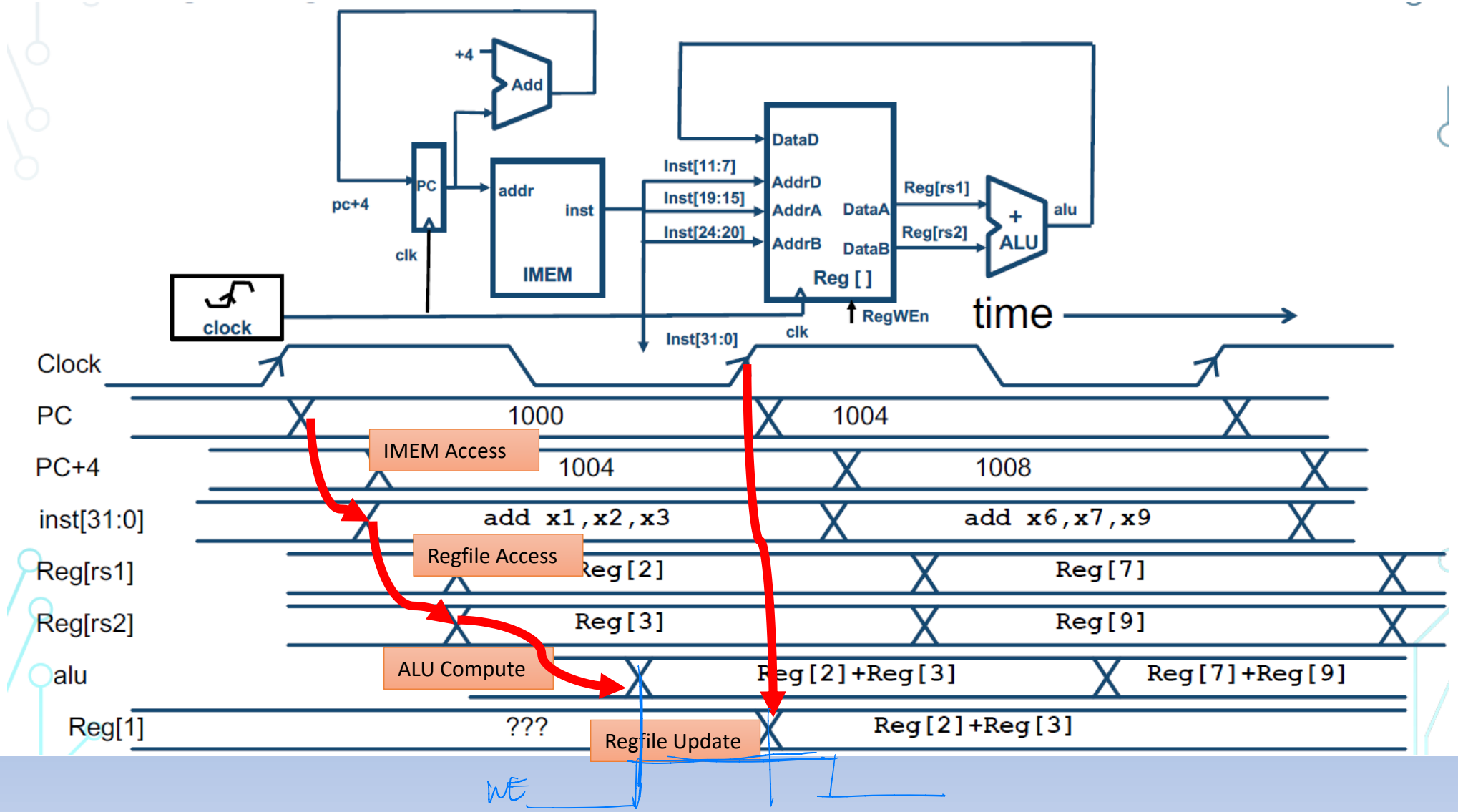
Instruction makes two changes to machine's state:

- **Reg[rd] = Reg[rs1] + Reg[rs2]**
- **PC = PC + 4** Byte

Datapath for **add**

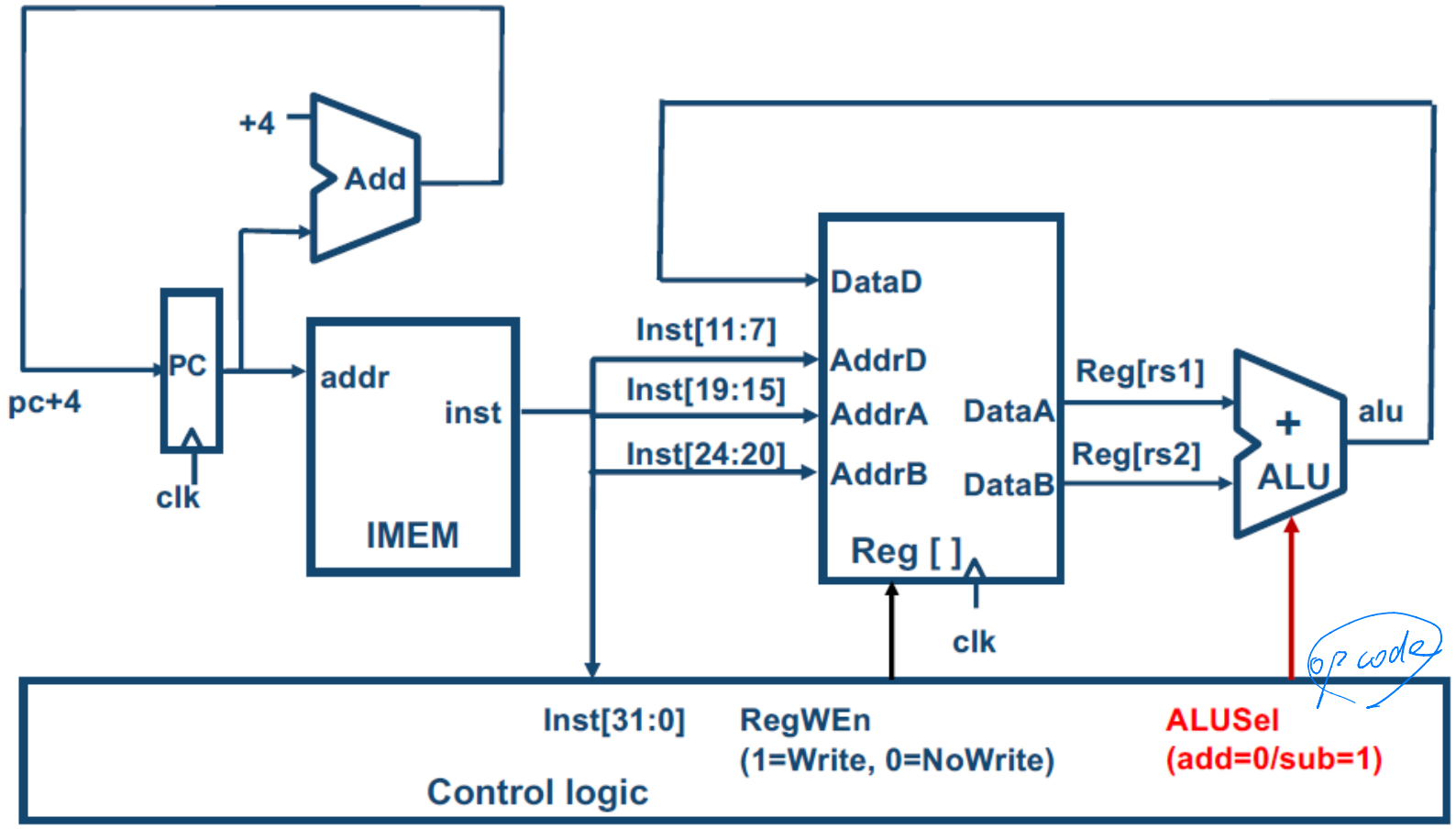


Timing Waveform for add



Datapath for add/sub

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub



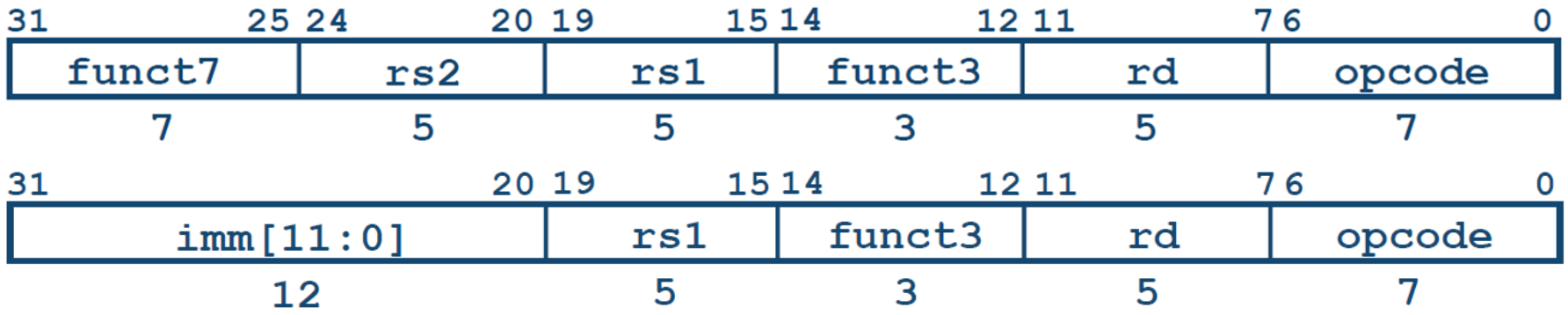
Implementing other R-Format instructions

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 generate **ALUSel** (Decoding Logic)

op code

I-Format Instruction Layout

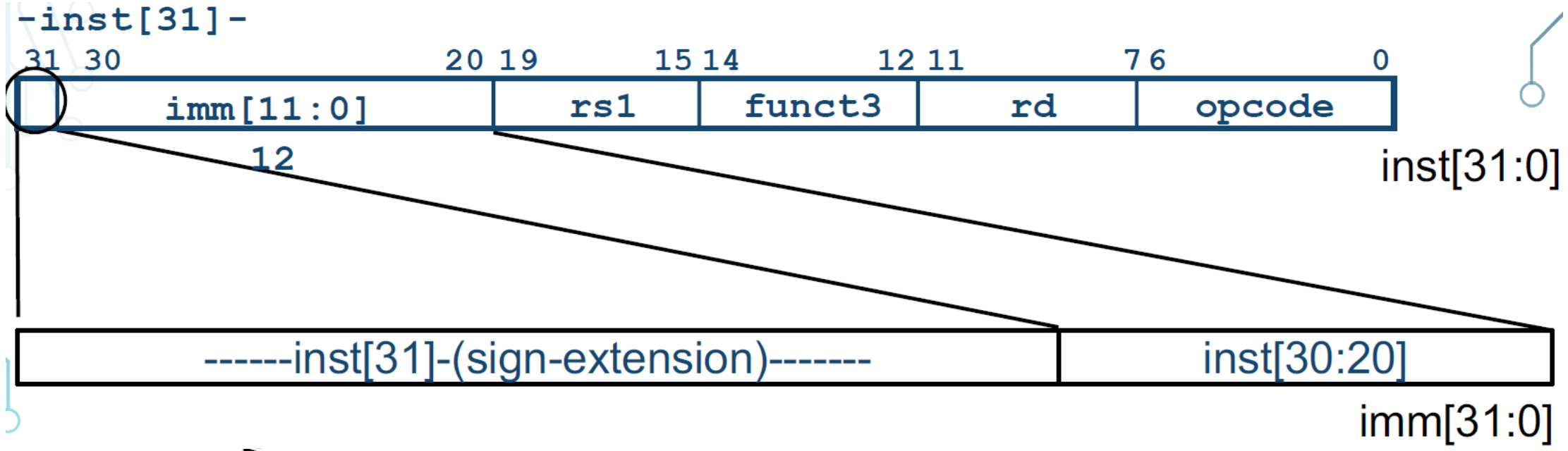


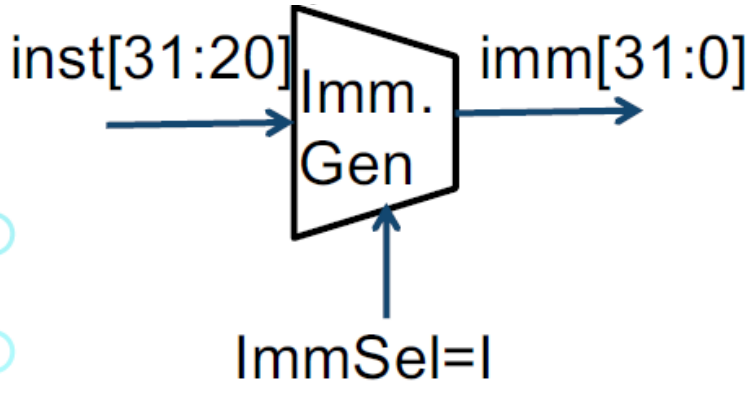
- rs2 and funct7 replaced by 12-bit signed immediate, **imm[11:0]**
- Remaining fields (rs1, funct3, rd, opcode) same as before
- imm[11:0] can hold values in range [-2048 , +2047]
- Immediate is always sign-extended to 32-bits before use in an arithmetic operation
- Other instructions handle immediate > 12 bits

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

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

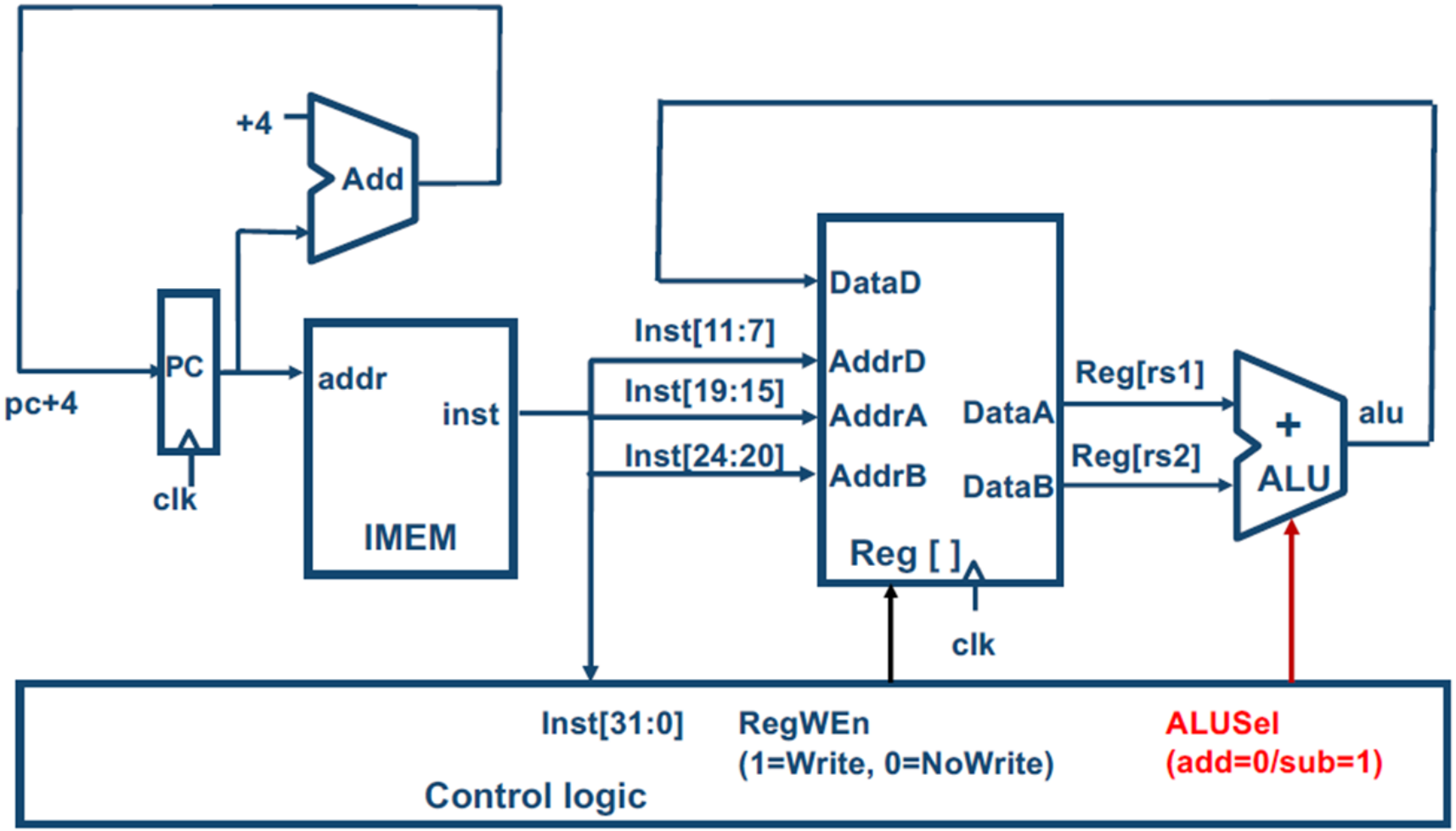
Immediate is sign-extended



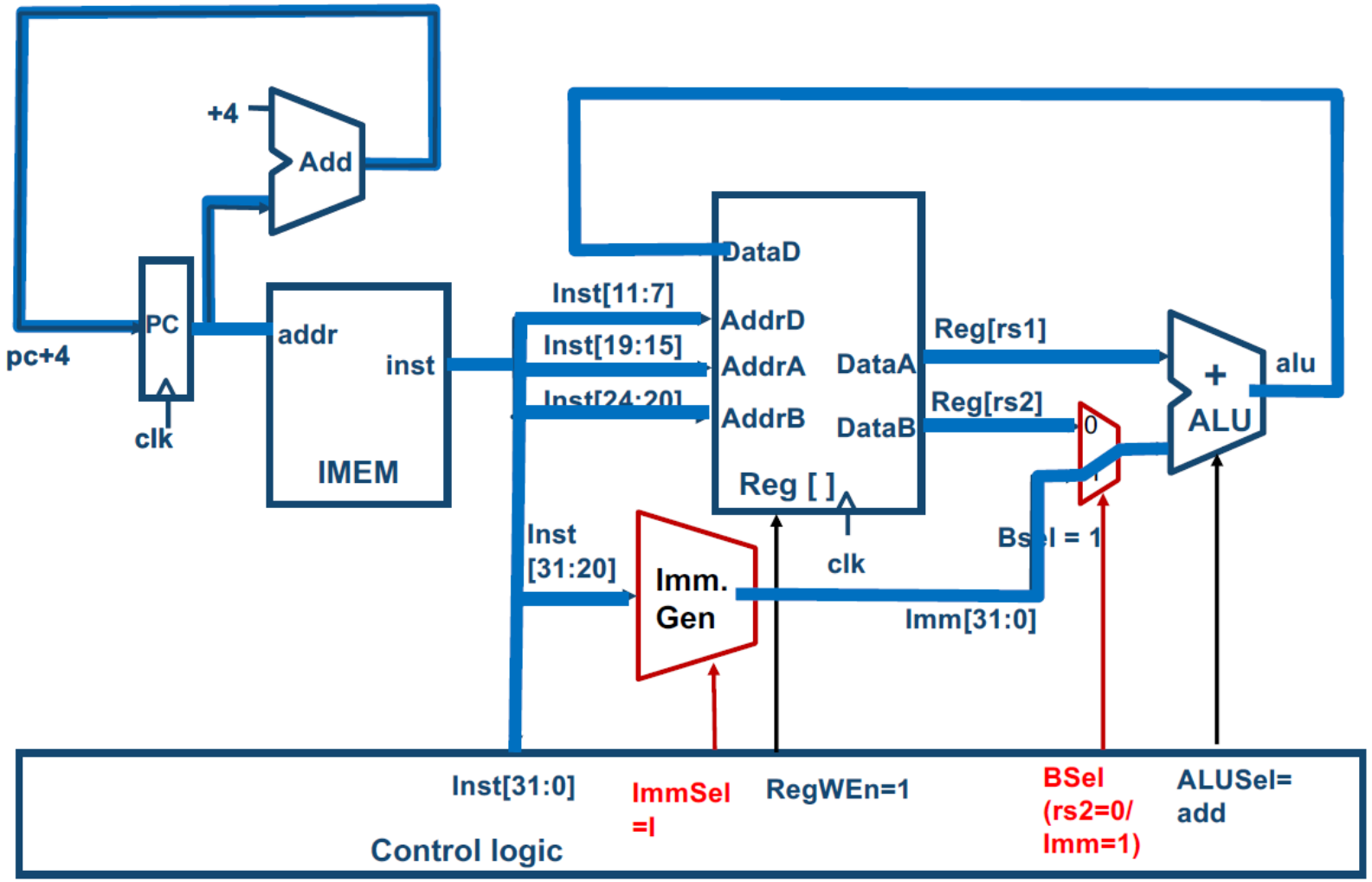


- **High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])**
- **Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])**

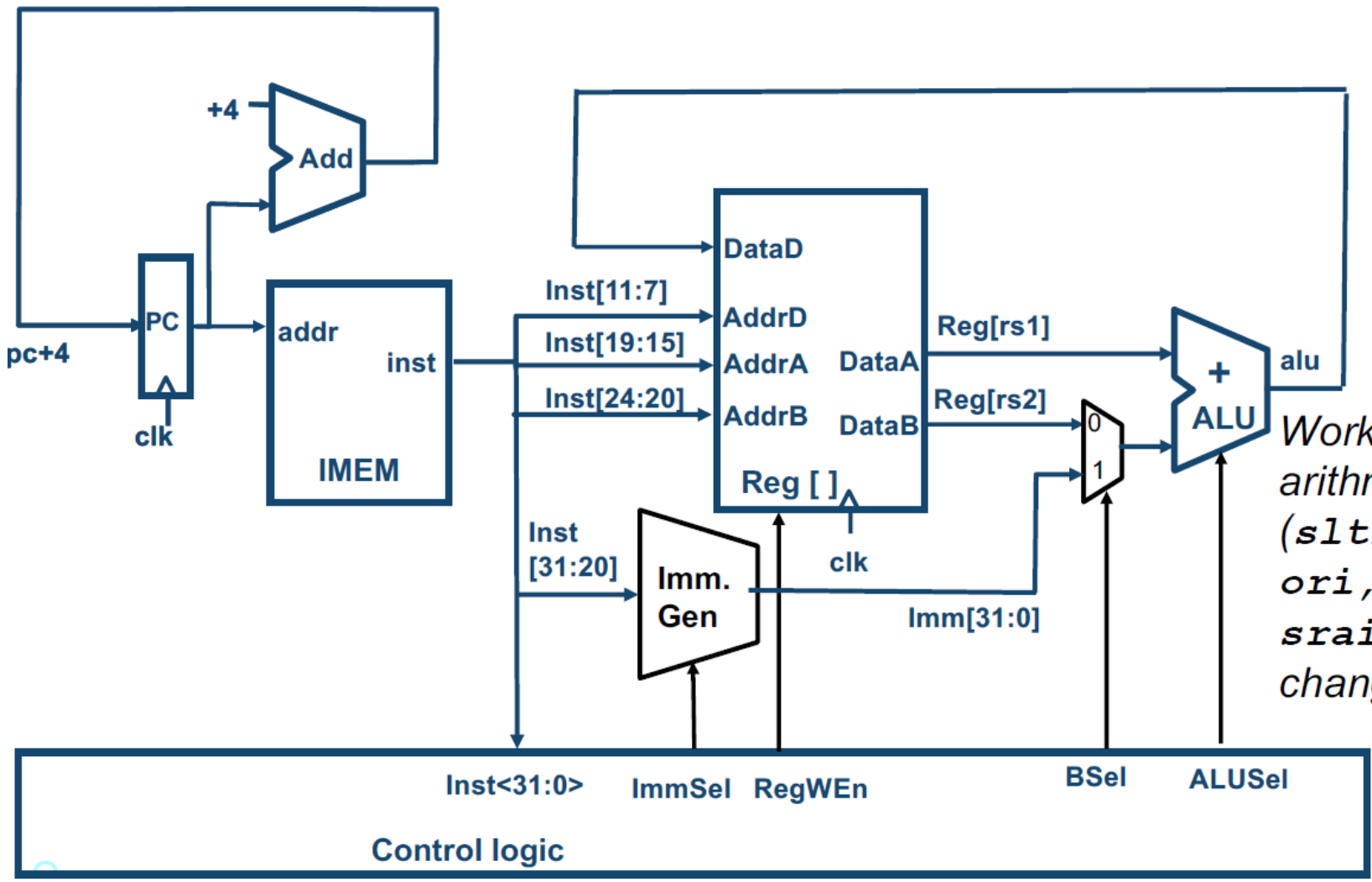
Datapath for addi / subi - Where to change the datapath ?



Datapath for addi / subi - Where to change the datapath ?

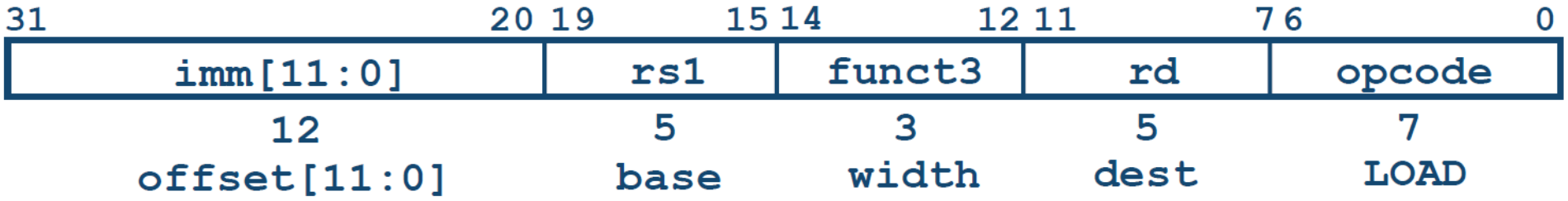


R+I Datapath



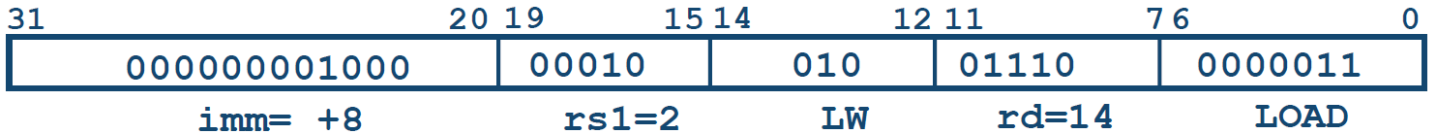
Works for all other I-format arithmetic instructions (`slti`, `sltiu`, `andi`, `ori`, `xori`, `slli`, `srli`, `srai`) just by changing ALUSel

Load Instruction is I-Type

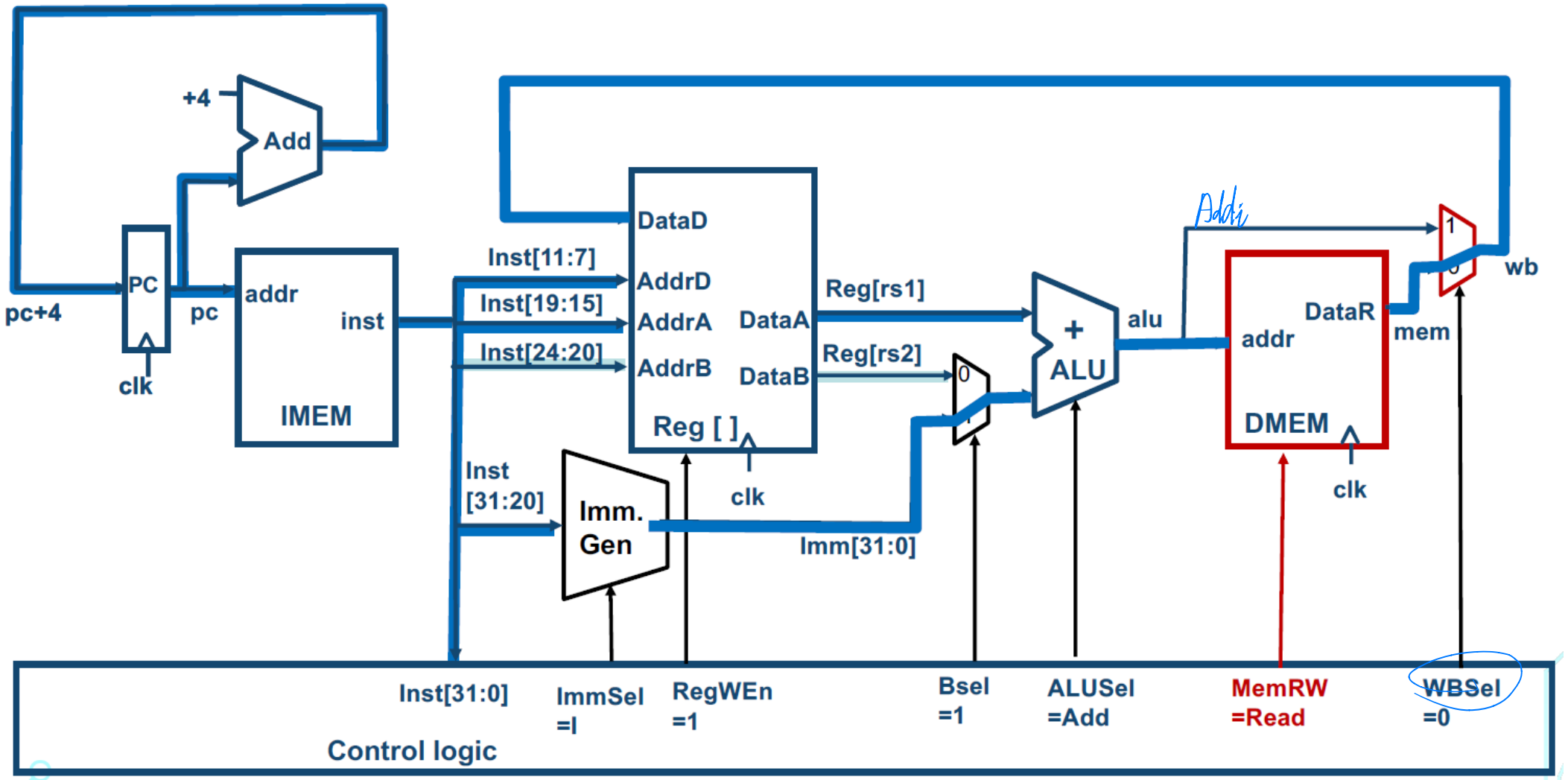


- Reg[rd] = Mem[Reg[rs1] + offset]
  - The 12-bit signed immediate is added to the base address in register rs1 to form the memory address
  - This is very similar to the add-immediate operation but used to create address not to create final result
- The value loaded from memory is stored in register rd

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



Add lw into Datapath



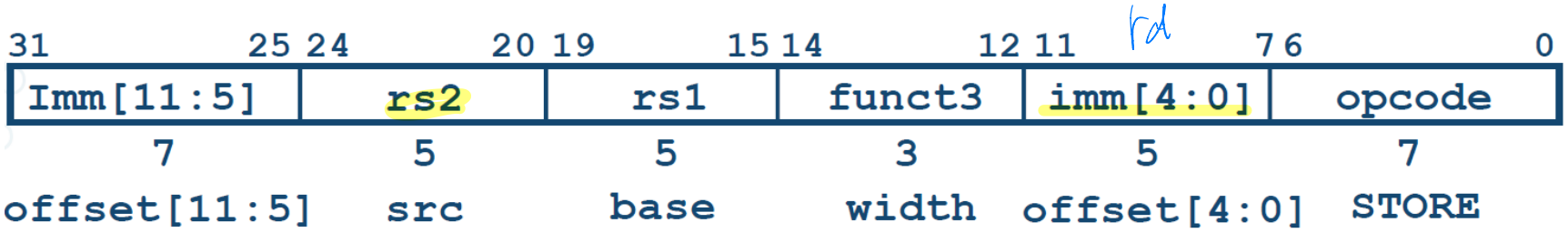
All RV32 Load Instructions

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

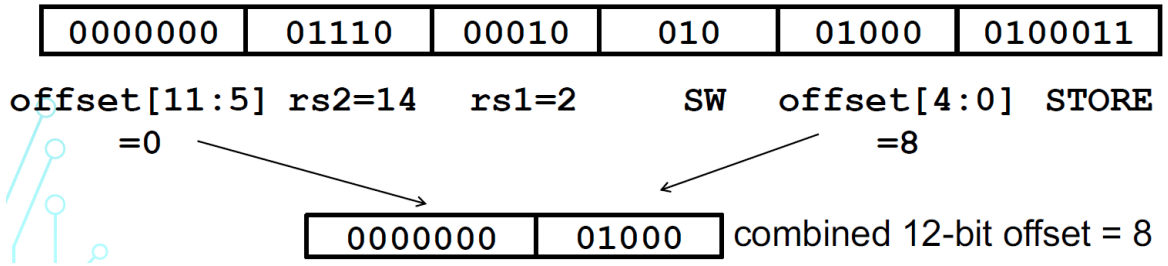
- lbu: load unsigned byte; lh: load halfword (halfword == 16bits == 2bytes)
- Supporting the narrower loads requires additional logic to
  - extract the correct byte/halfword from the value loaded from memory, and
  - sign- or zero-extend the result to 32 bits before writing back to register file.
  - It is just a mux for load extend, similar to sign extension for immediates

S-Format for Stores

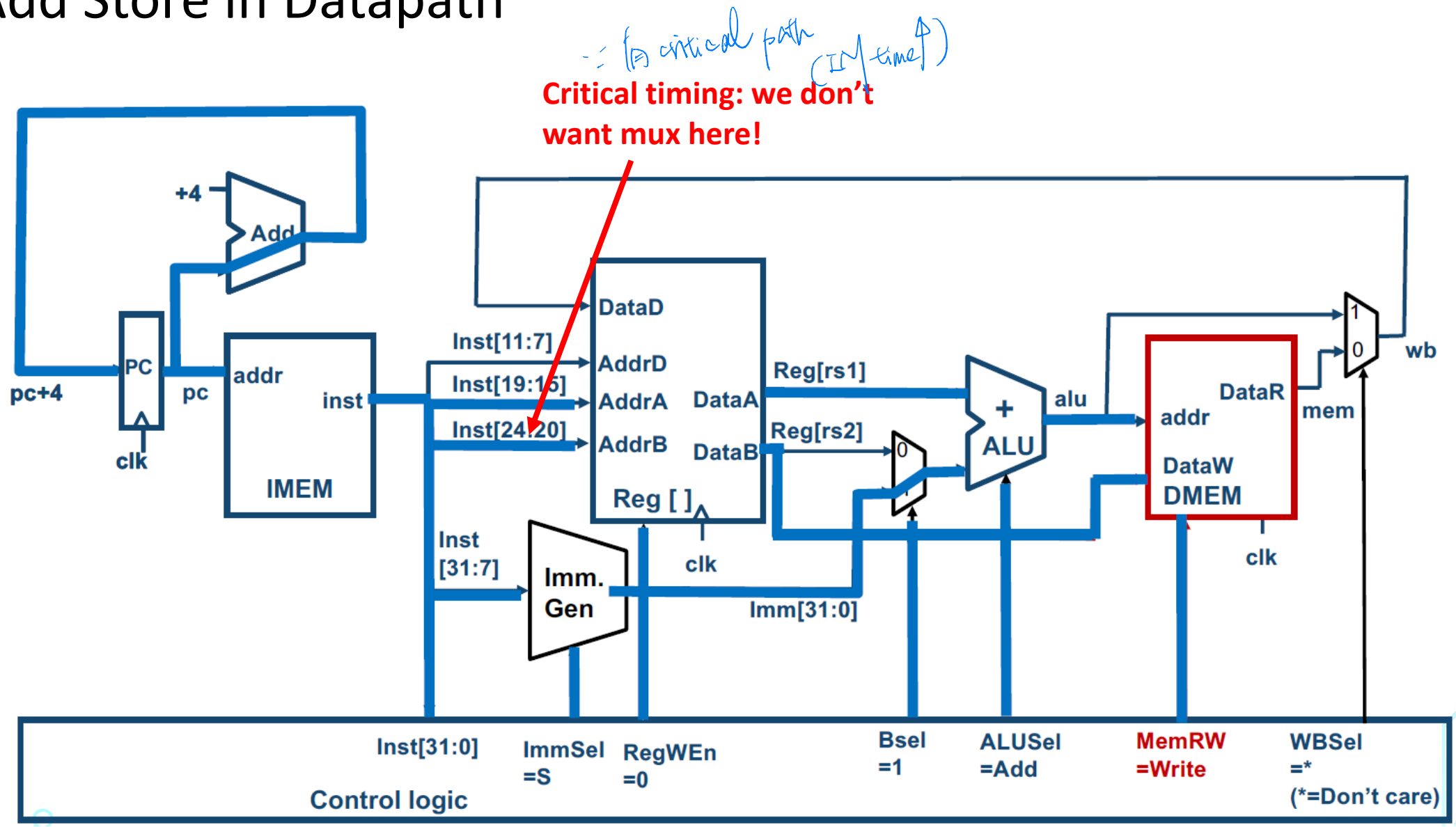


- Mem [Reg[rs1] + offset] = Reg[rs2]
  - Store needs to read two registers, rs1 for base memory address, and rs2 for data to be stored, as well immediate offset!
  - Note that stores don't write a value to the register file, ***no rd***!
- Immediate in two parts:
  - Can't have both rs2 and immediate in same place as other instructions!
  - RISC-V design decision is move low 5 bits of immediate to where rd field was in other instructions – keep rs1/rs2 fields in same place
    - **register names more critical than immediate bits in hardware design**

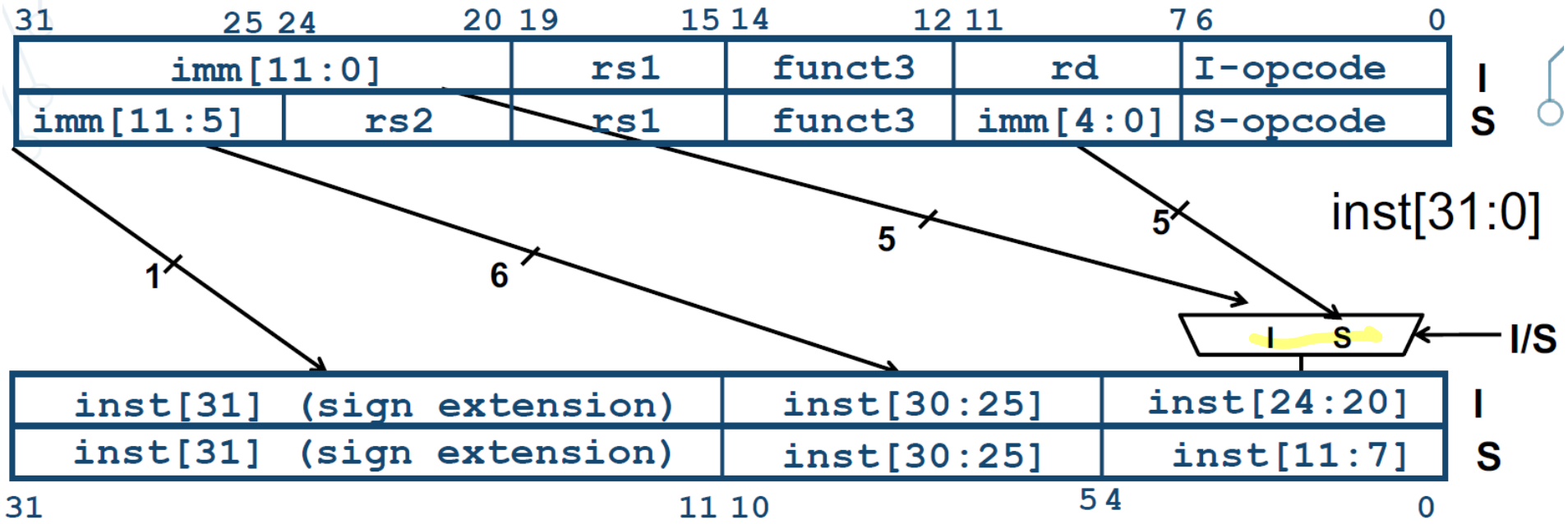
```
sw x14, 8(x2)
```



Add Store in Datapath

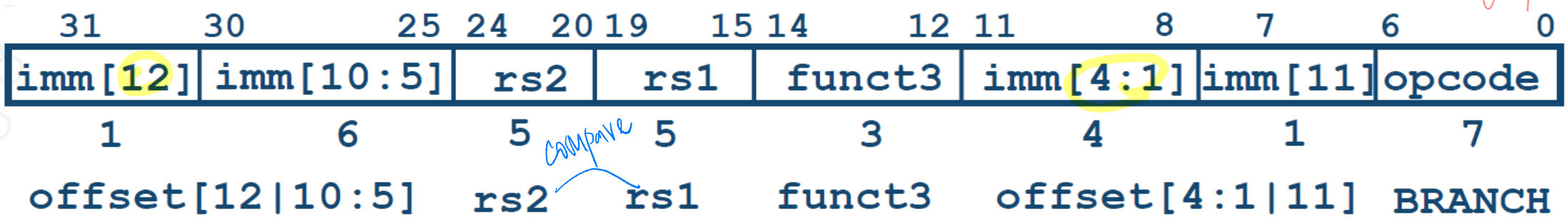


I + S Immediate Generation



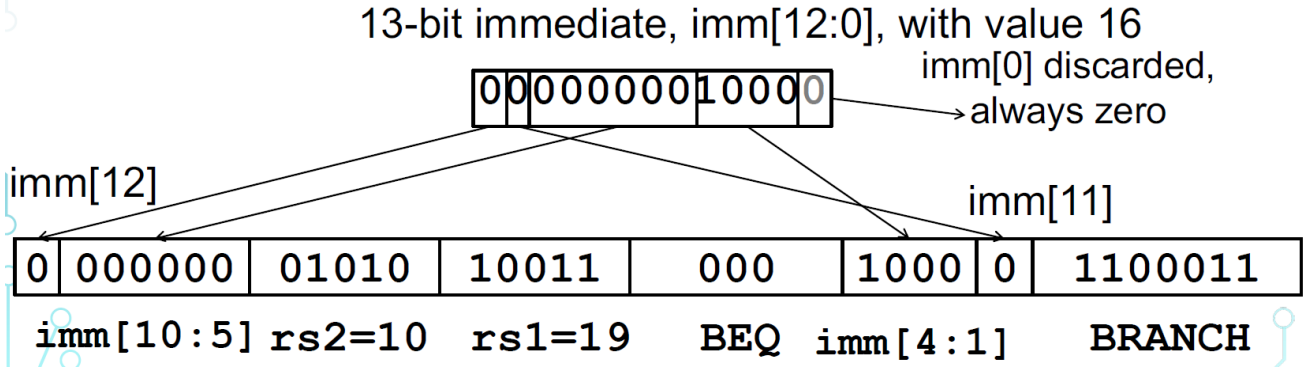
- Just need a 5-bit mux to select between two positions where low five bits of immediate can reside in instruction
- Other bits in immediate are wired to fixed positions in instruction

B-Format – Conditional Branch

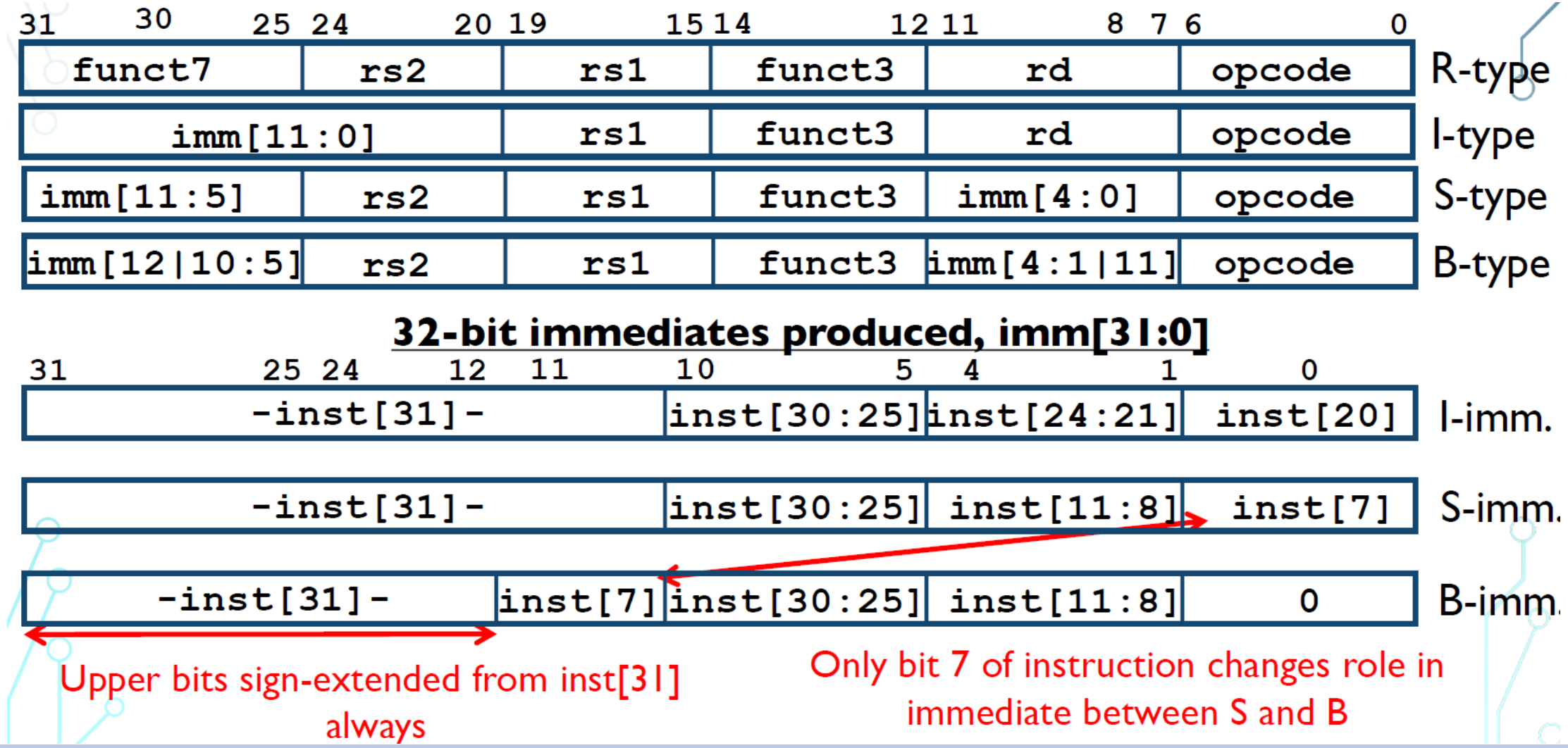


- B-format is similar to S-format, with two register sources (rs1/rs2) and a 12-bit immediate
- The 12 immediate bits encode 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)
- But now immediate represents values -2^12 to +2^12 byte increments

```
beq    x19,x10,  offset = 16 bytes
```



RISC-V Immediate Encoding

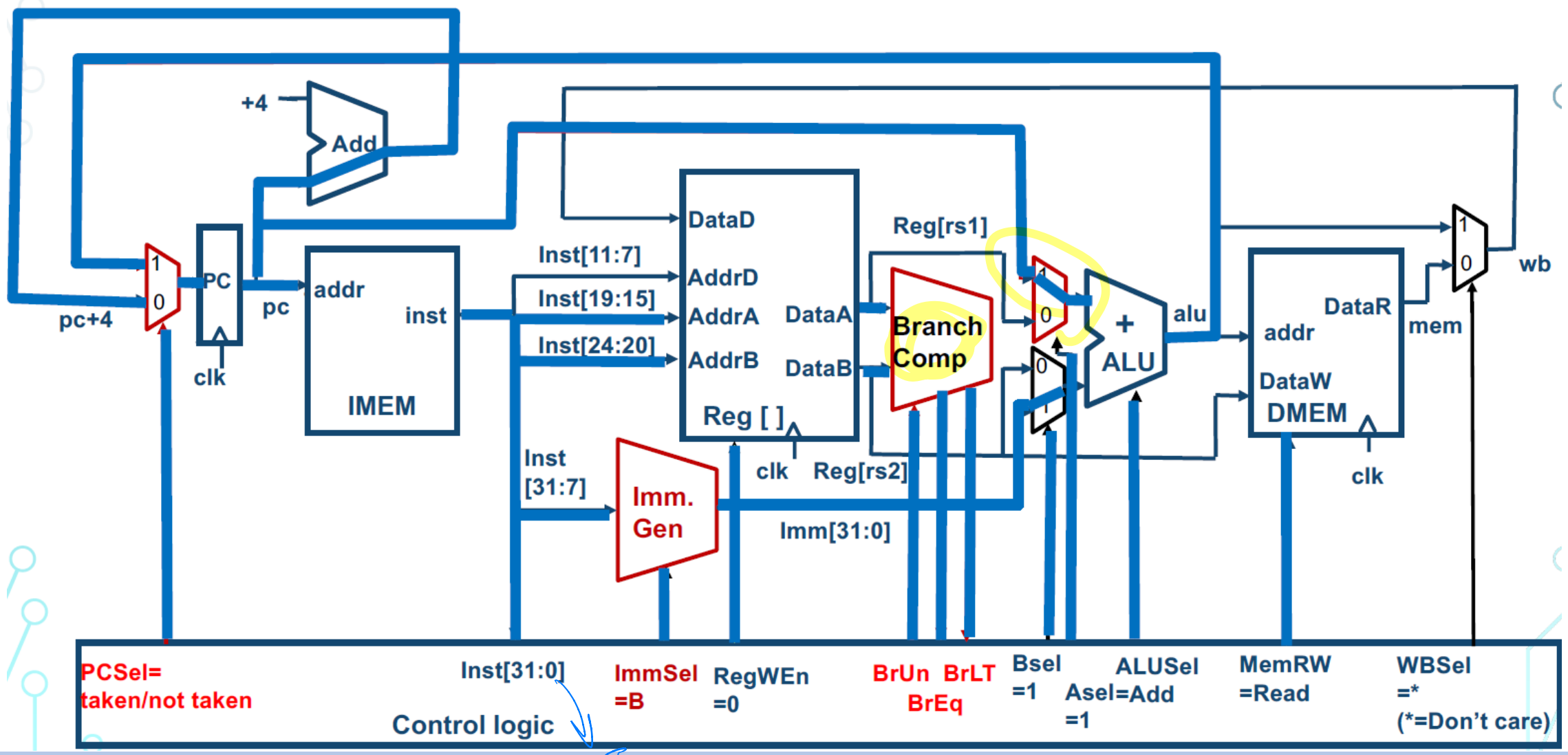


To implement Branches

- Change on PC
  - PC = PC + 4 branch not take
  - PC = PC + immediate branch taken (new)
- Compare values of rs1 and rs2
- Six branch instructions: BEQ, BNE, BLT, BGE, BLTU, BGEU

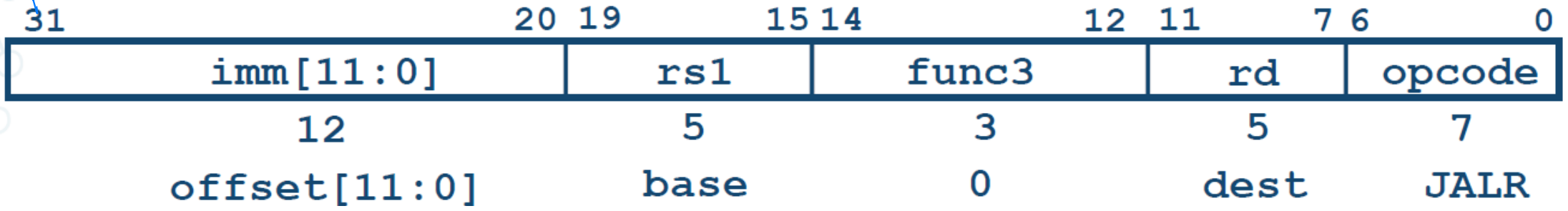
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

Add Branches



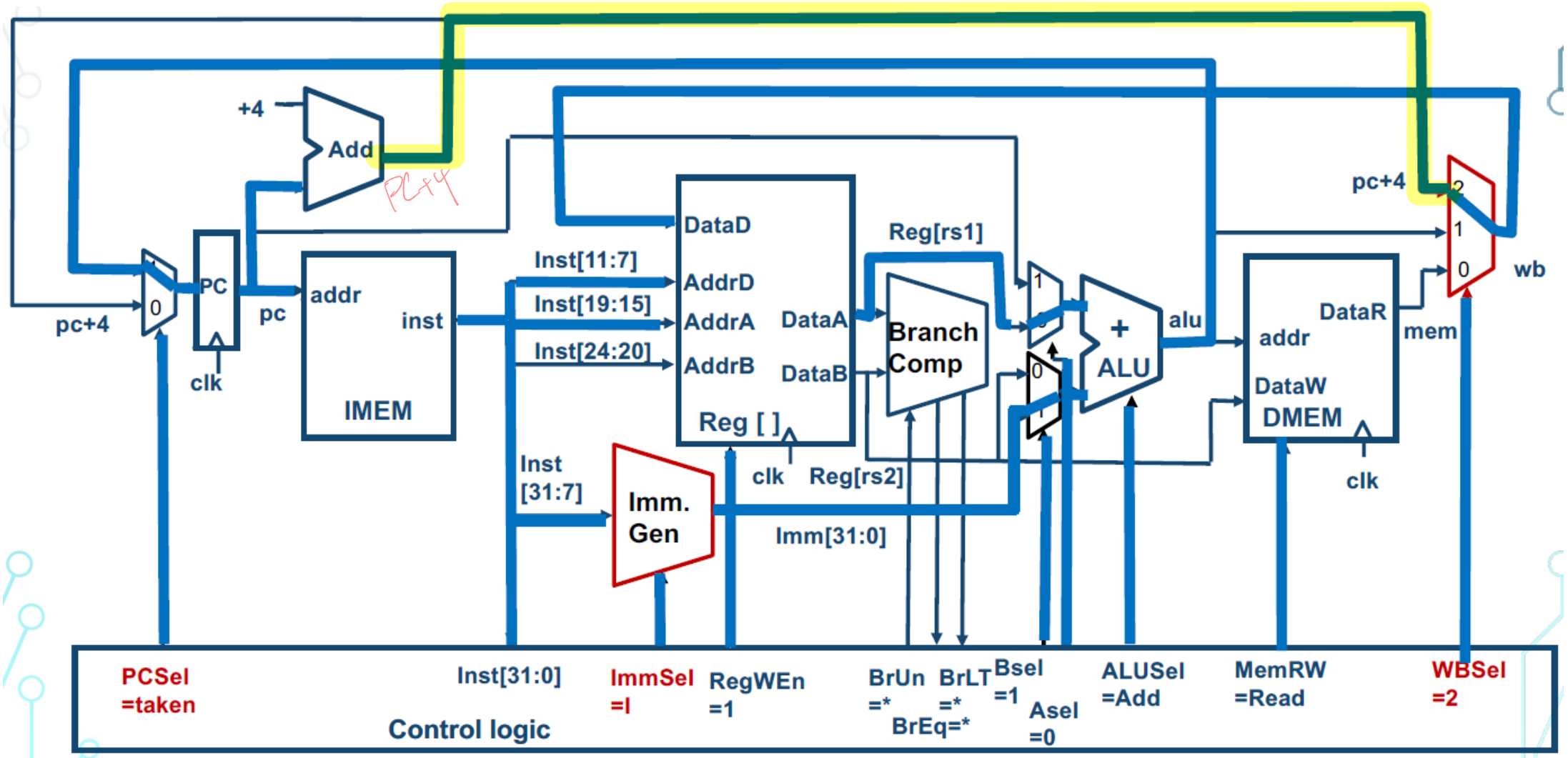
JALR Instruction (I-Format)

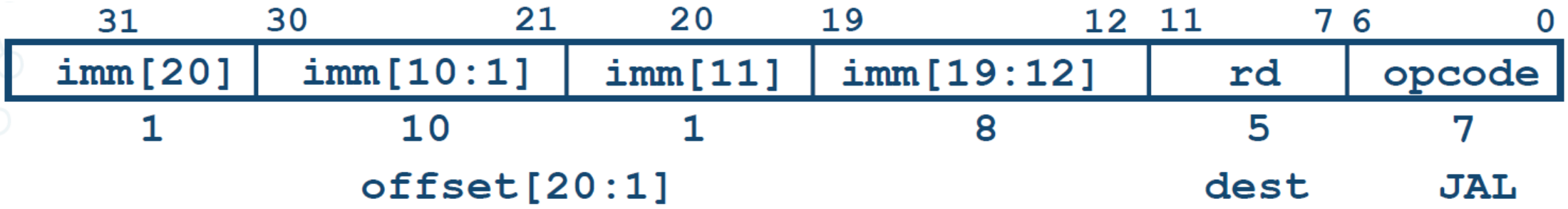
Jump & Link



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

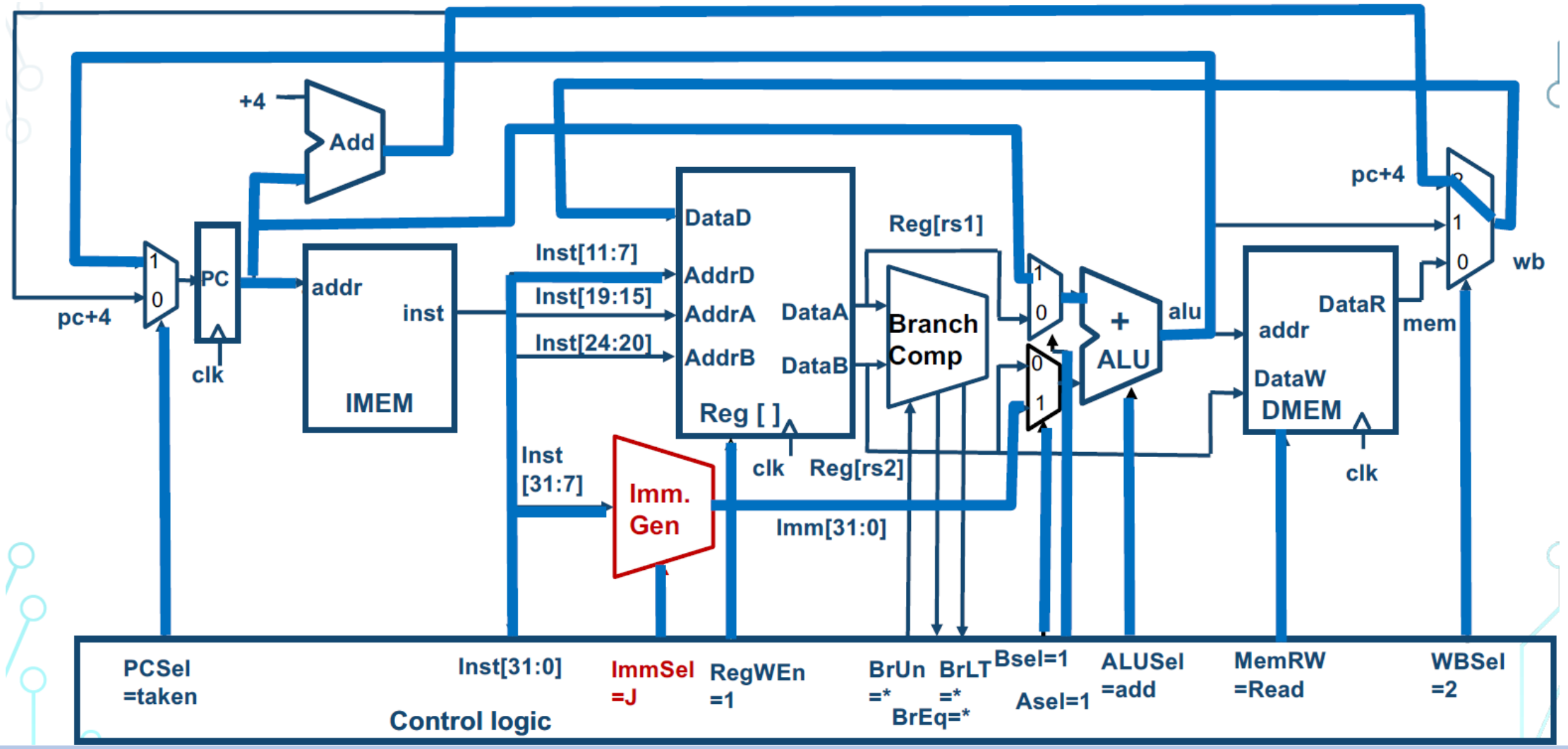
Add JALR



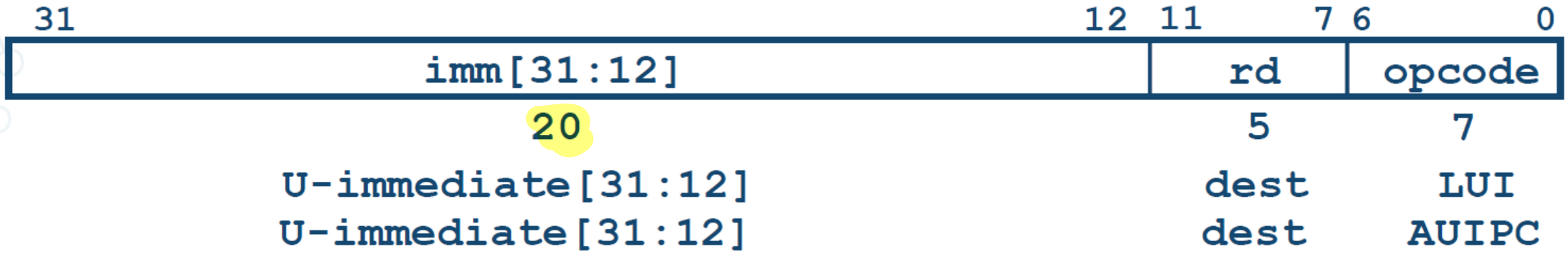


- JAL:
  - R[rd] = PC + 4; PC = PC + imm;
  - saves PC+4 in register rd (the return address)
    - Assembler "**j**" jump is pseudo-instruction, uses JAL but sets rd=x0 to discard return address
  - Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{\wedge 18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

Add JAL

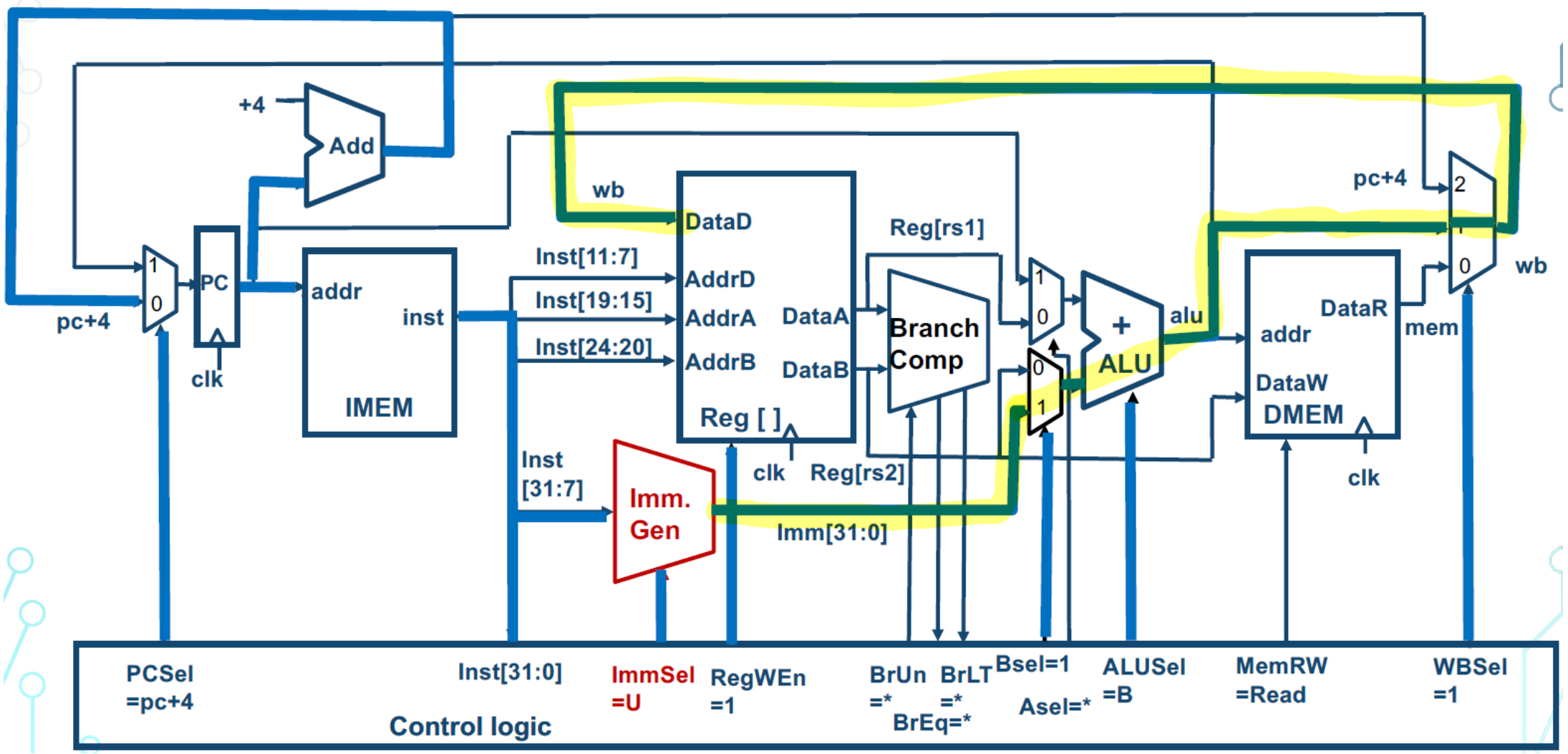


U-Format for “Upper Immediate” Instructions

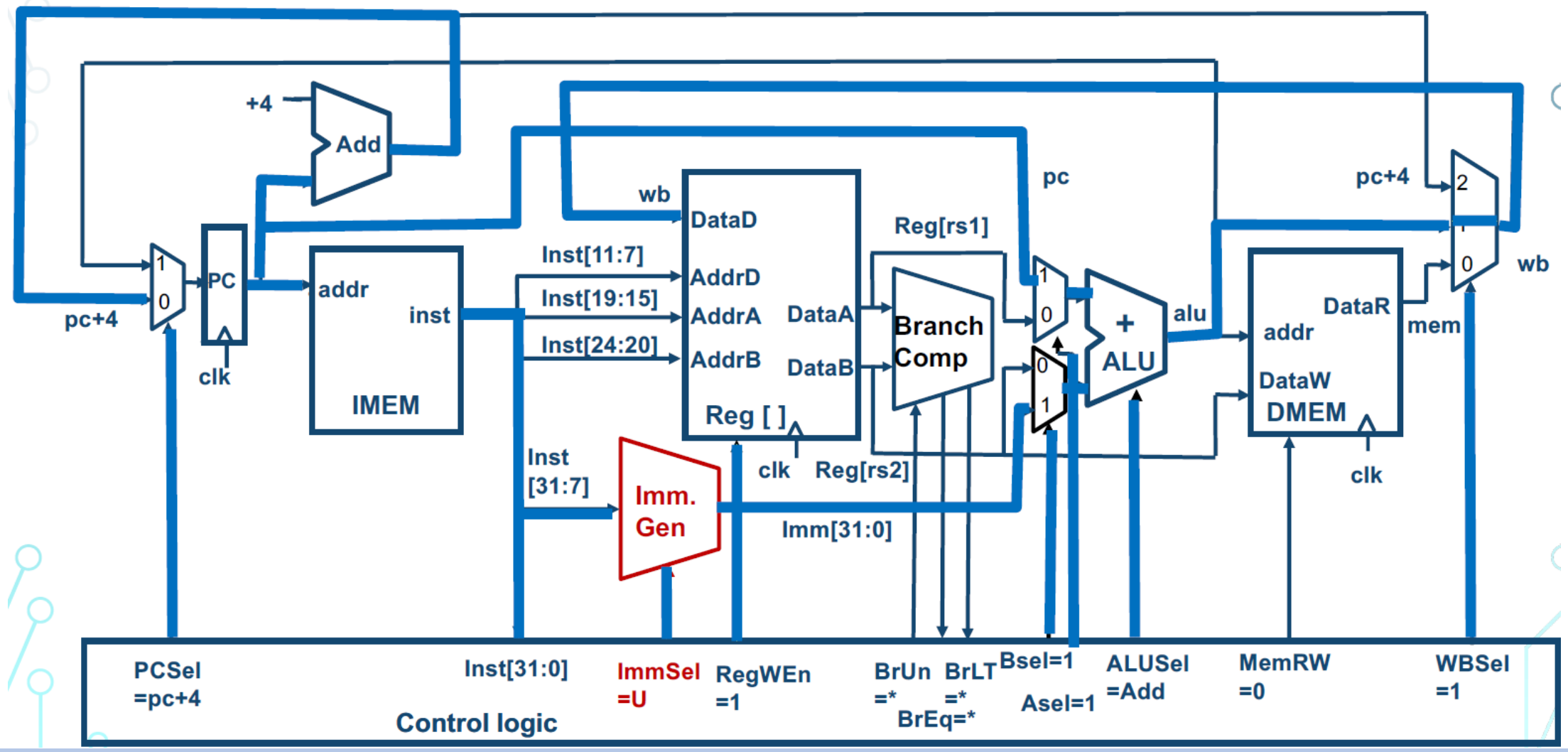


- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
  - LUI – Load Upper Immediate
    - Reg[rd] = {imm, 12b’0}
  - AUIPC – Add Upper Immediate to PC
    - Reg[rd] = PC + {imm, 12’b0}

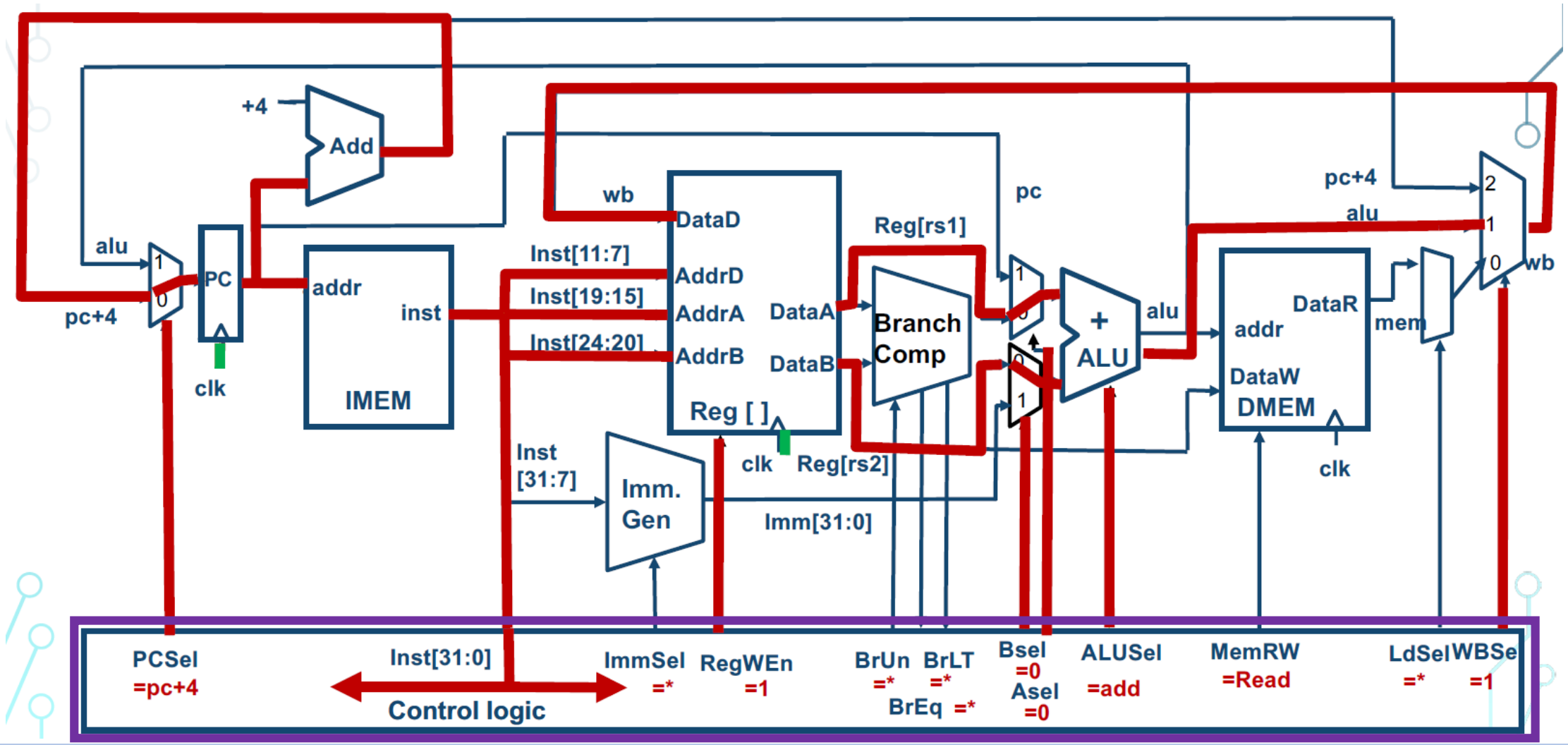
Implement LUI



Implement AUIPC



Control Logic Implementation



Truth Table

Pure combinational logic

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	+4	*	*	Reg	Reg	Add	Read	1	ALU
sub	*	*	+4	*	*	Reg	Reg	Sub	Read	1	ALU
(R-R Op)	*	*	+4	*	*	Reg	Reg	*(Op)*	Read	1	ALU
addi	*	*	+4	I	*	Reg	Imm	Add	Read	1	ALU
lw	*	*	+4	I	*	Reg	Imm	Add	Read	1	Mem
sw	*	*	+4	S	*	Reg	Imm	Add	Write	0	*
beq	0	*	+4	B	*	PC	Imm	Add	Read	0	*
beq	1	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	0	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	1	*	+4	B	*	PC	Imm	Add	Read	0	*
blt	*	1	ALU	B	0	PC	Imm	Add	Read	0	*
bltu	*	1	ALU	B	1	PC	Imm	Add	Read	0	*
jalr	*	*	ALU	I	*	Reg	Imm	Add	Read	1	PC+4
jal	*	*	ALU	J	*	PC	Imm	Add	Read	1	PC+4
auipc	*	*	+4	U	*	PC	Imm	Add	Read	1	ALU