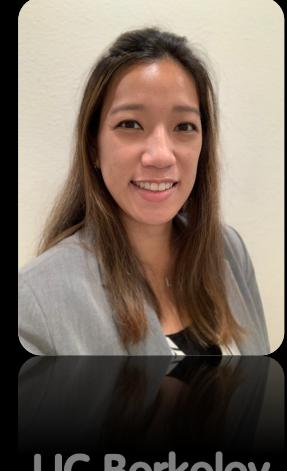




UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture (a.k.a. Machine Structures)



UC Berkeley
Teaching Professor
Lisa Yan

Systems Lunch on the Glade!
Today! Friday, October 7, 12pm
Dan, Lisa, and
Prof. Natacha Crooks (CS162)
Bring your own lunch
and come join us!

RISC V Datapath II



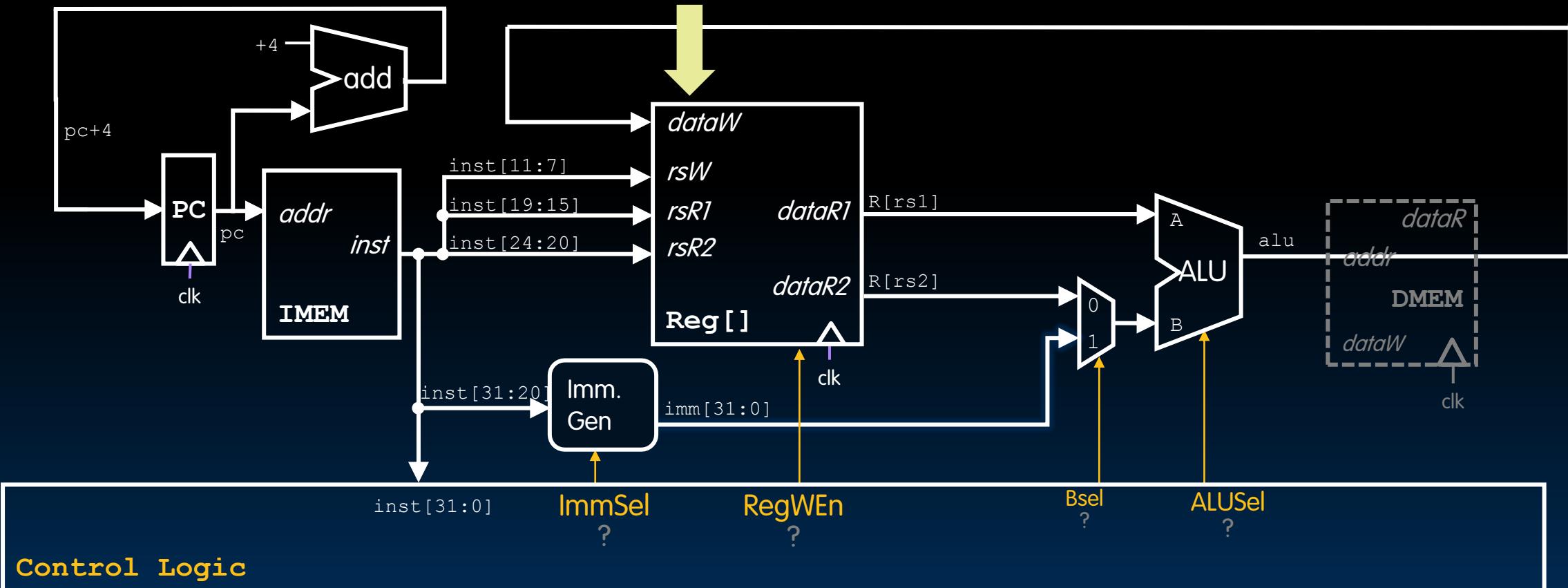
Our Datapath So Far: Arithmetic/Logic

Review

Edge-triggered write.

If RegWEn=1, RegFile updated with input to dataW on the next rising-clock edge.

Bold white wires carry data.
Thin orange wires are control logic.



Implementing Loads

- Implementing Loads
- Implementing Stores
- Adding B-Type (Branches)
- Designing the Immediate Generation Block, Part 2
- Adding Jumps `jal`, `jalr`
- Adding U-Types
- And in Conclusion...

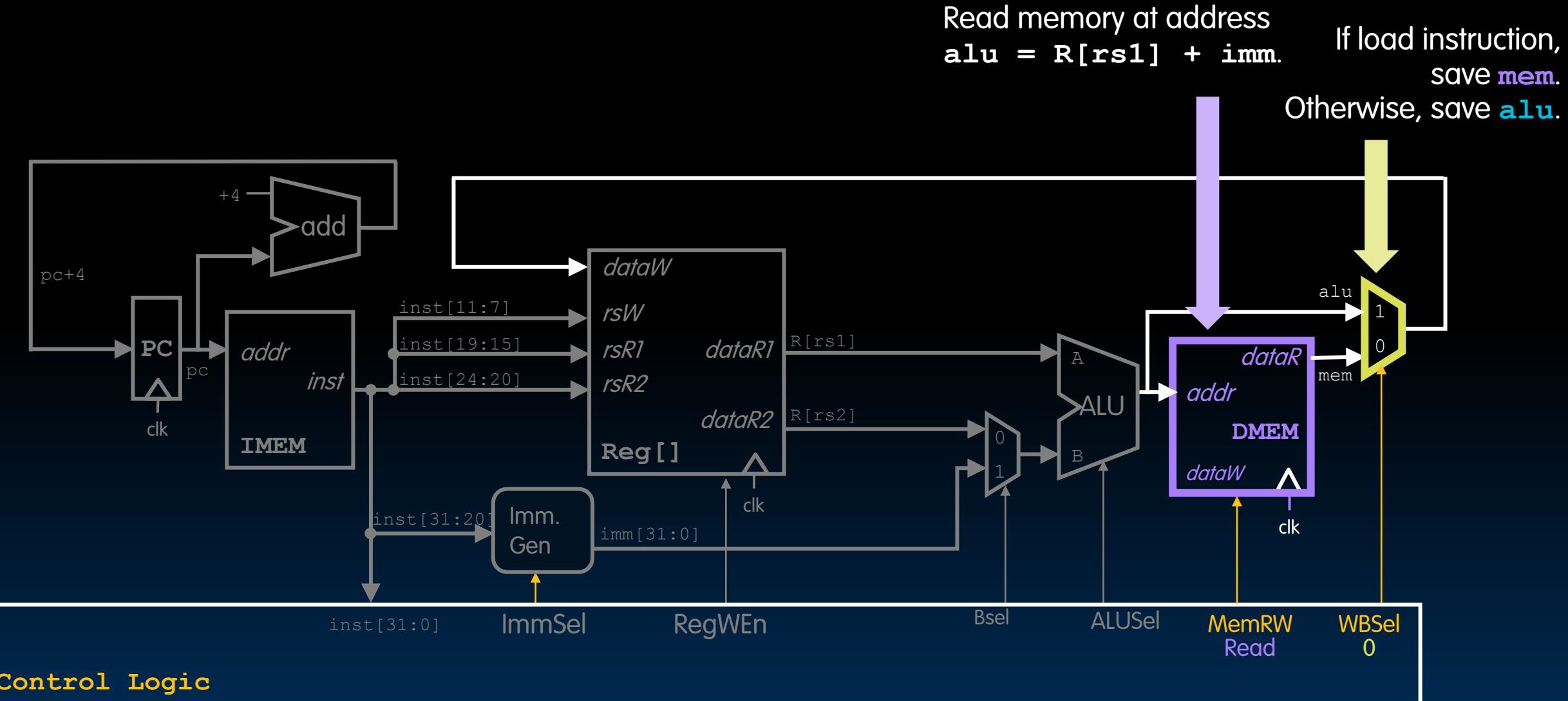
Implementing the `lw` Instruction

- **lw uses l-Format:** `lw x14, 8(x2)`

					31	20	19	15	14	12	11	7	6	0	
					imm[11:0]		rs1		funct3		rd		opcode		
					000000001000		00010		010		01110		0000011		
					12	5	5	3	5	7	LOAD				
"load word"															

- Similar datapath to addi, but creates an address (not the final value stored).
$$\text{addr} = (\text{Base register } \text{rs1}) + (\text{sign-extended } \text{imm offset})$$
 - State element access now includes a memory read!
 - DMEM (read word at address **addr**)
 - RegFile **Reg[rs1]** # read; **Reg[rd]** # write
 - PC **PC = PC + 4**

Saving Memory Read to RegFile



Lighting the LOAD Datapath

Load uses all five stages of the datapath!

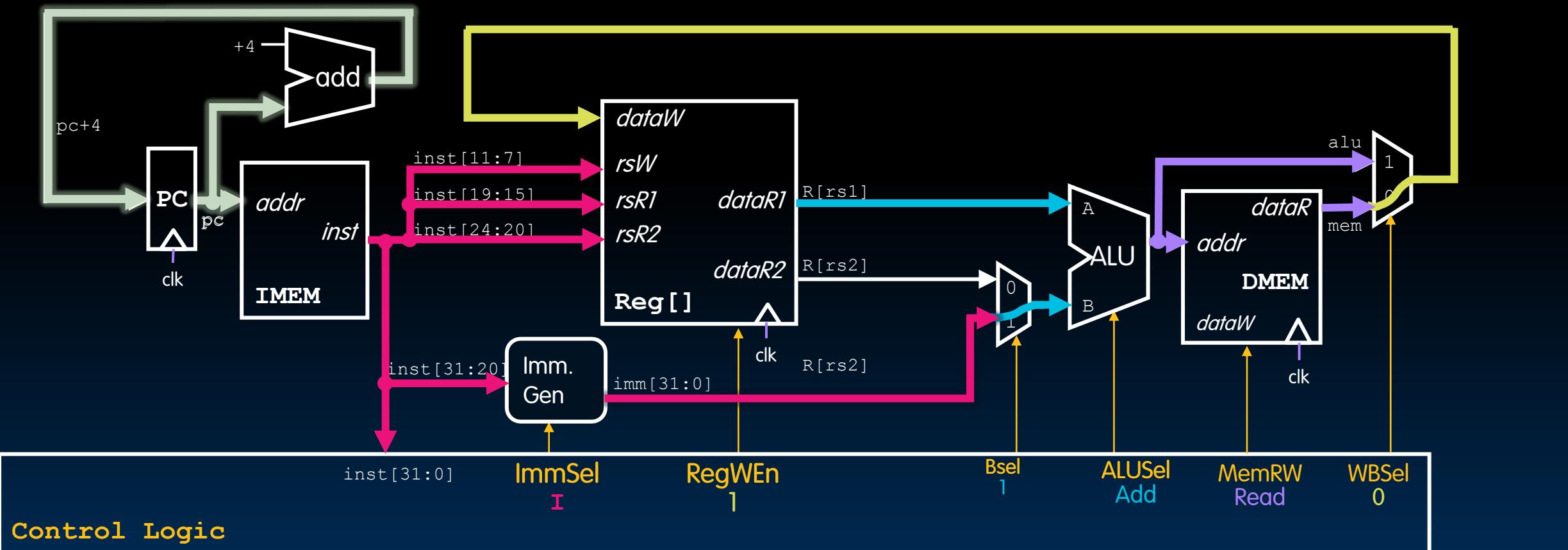
Increment PC to next instruction.

Immediate Generation Block builds a 32-bit immediate **imm**.

ALU computes address **alu** = $R[\text{rs1}] + \text{imm}$.

Read memory at address **alu**.

Write loaded memory value **mem** to register.



RV32I Can Load Different Widths

imm[11:0]	rs1	funct3	rd	opcode	
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

- To support narrower loads (lb, lh, lbu, lhu):
 - Load 32-bit word from memory;
 - Add additional logic to extract correct byte or halfword; and
 - Sign- or zero-extend result to 32 bits to write into RegFile.
 - Can be implemented with MUX + and a few gates.

For simplicity, we'll omit this wiring in lecture. Check out Project 3!

Implementing Stores

- Implementing Loads
- Implementing Stores
- Adding B-Type (Branches)
- Designing the Immediate Generation Block, Part 2
- Adding Jumps `jal`, `jalr`
- Adding U-Types
- And in Conclusion...

Implementing the sw Instruction

- S-Format:

sw x14, 36(x2)



- New Immediate Format:

0000001	00100
---------	-------

- addr = (Base register **rs1**) + (sign-extended **imm** offset)

- State Elements Accessed:

- DMEM (write R[**rs2**] to word at address **addr**)

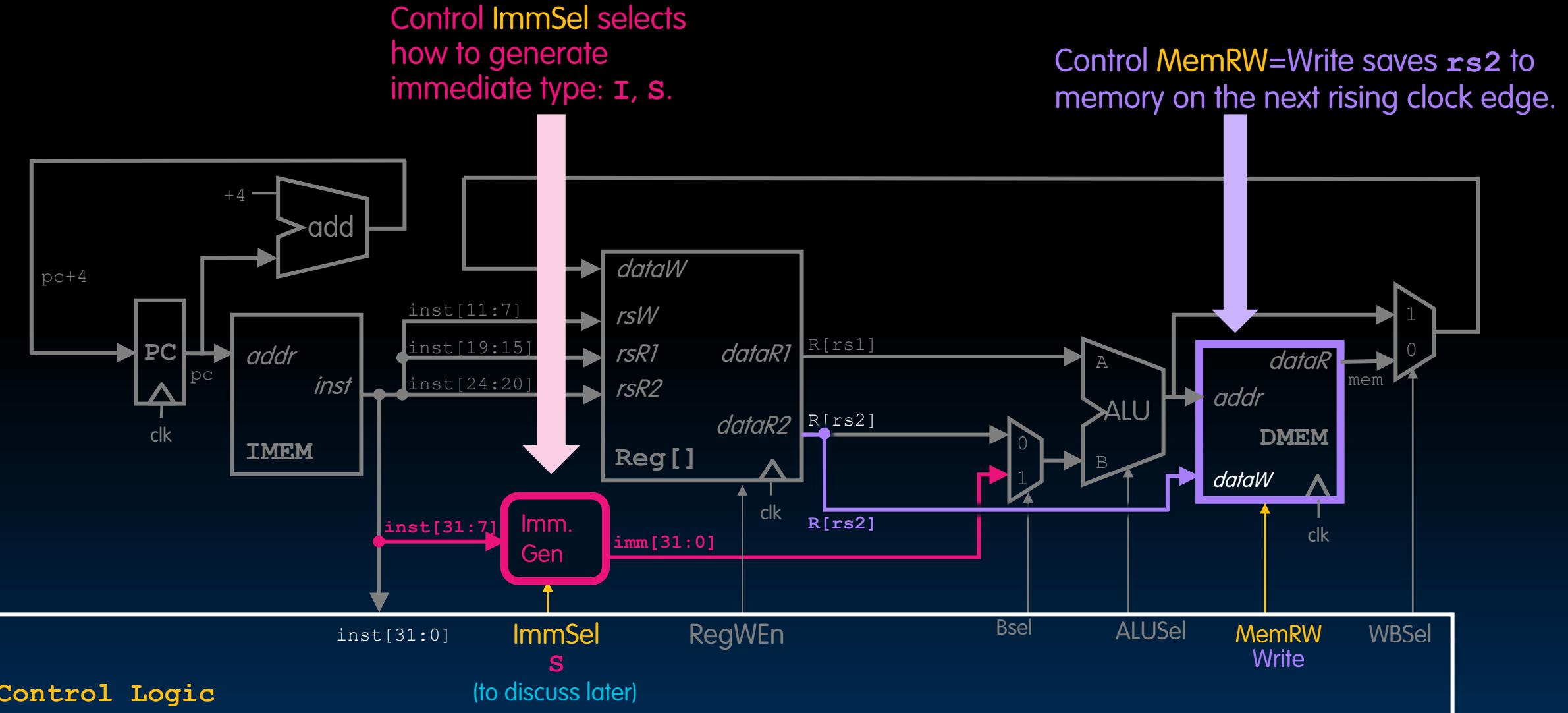
- RegFile R[**rs1**] (base address), R[**rs2**] (value to store)

- PC $PC = PC + 4$

No
RegFile
write!

Garcia, Yan

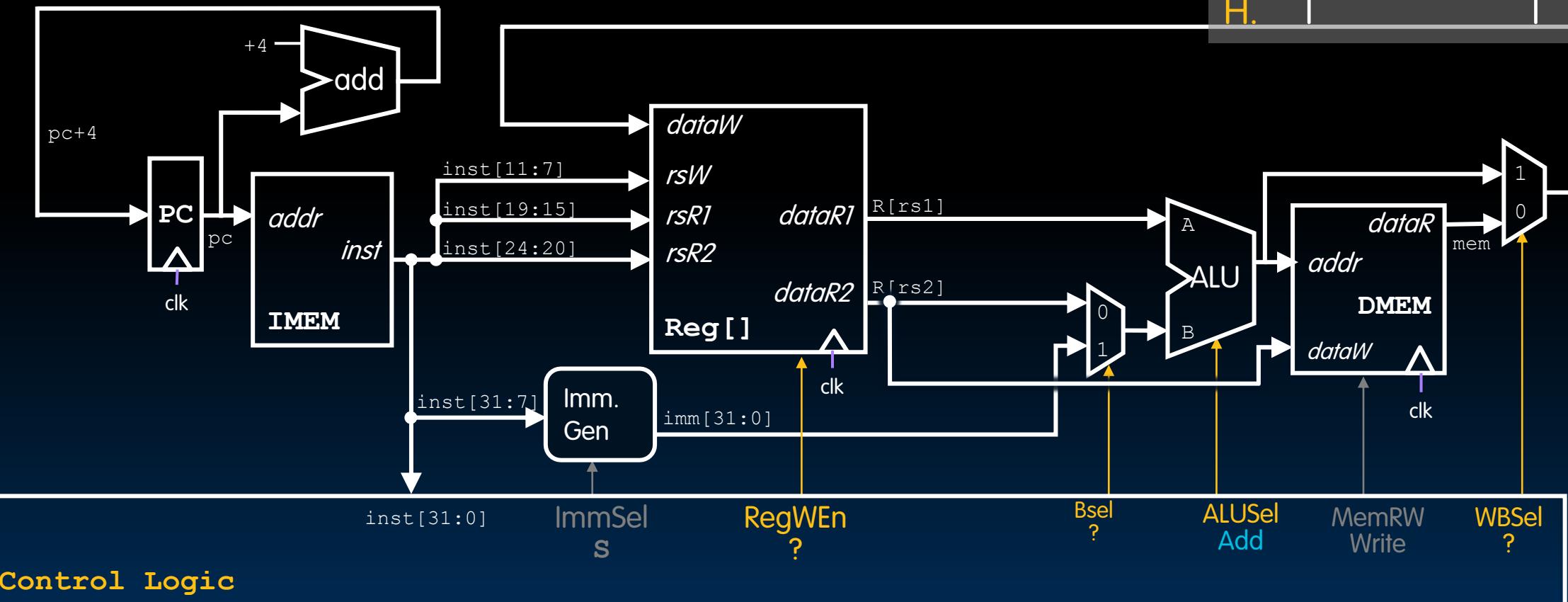
Updated Blocks for sw



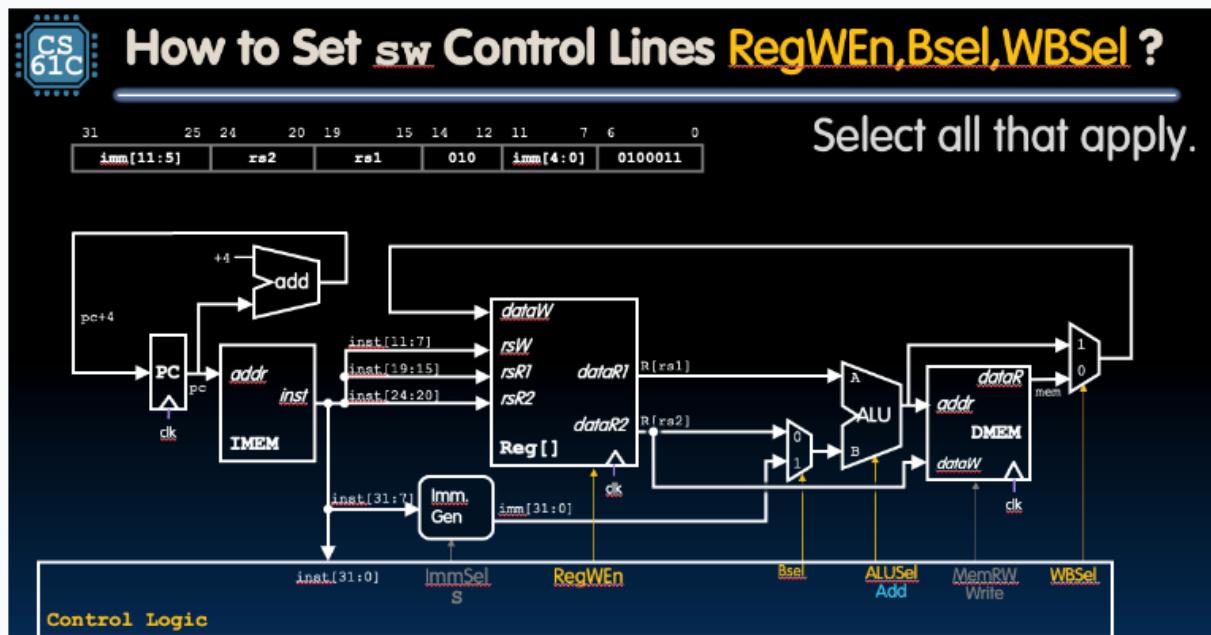
How to Set sw Control Lines?



	RegWEn	Bsel	WBSel
A.	Read(0)	0	0
B.	0	0	1
C.	0	1	0
D.	0	1	1
E.	Write(1)	0	0
F.	1	0	1
G.	1	1	0
H.	1	1	1



Datapath Design II



Total Results: 0

Lighting the STORE Datapath

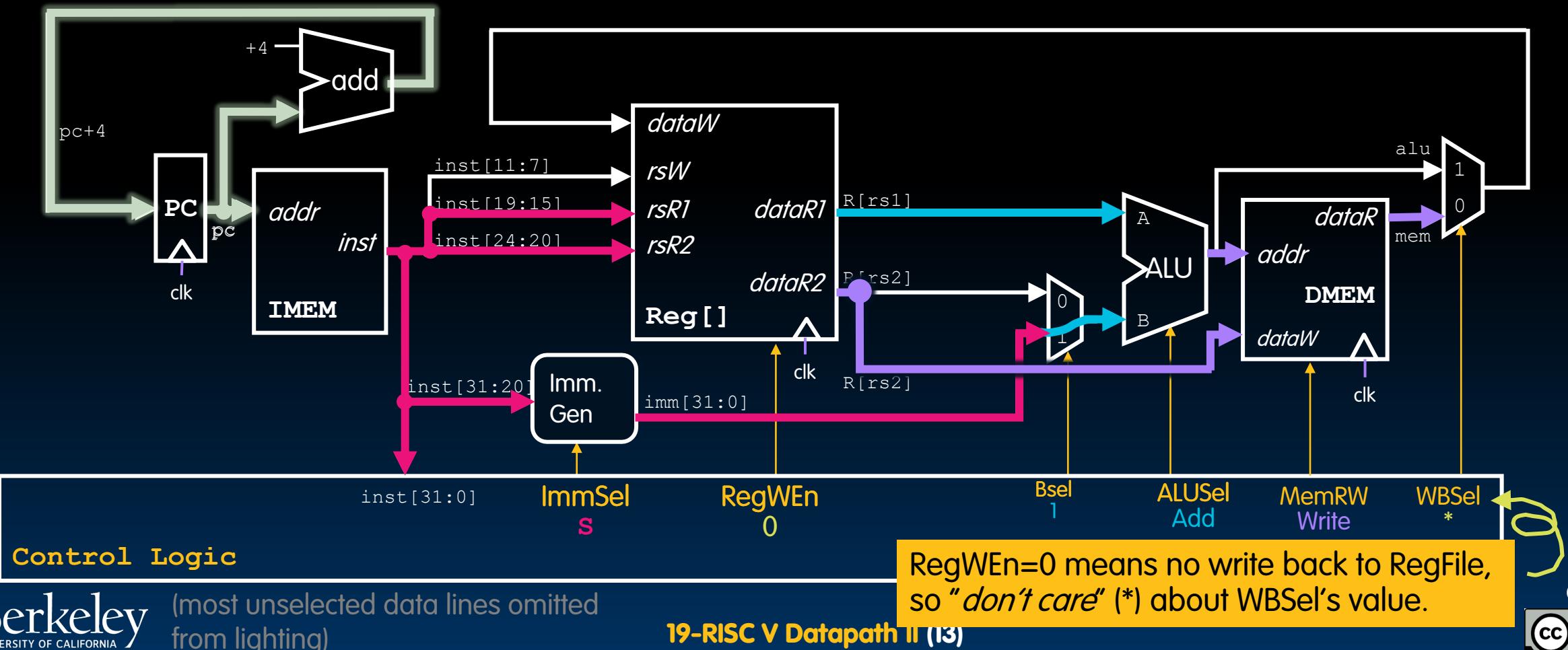
	RegWEn	Bsel	WBSel
C.	0	1	0
D.	0	1	1

Increment PC to next instruction.

Build **imm** from S-type instruction.

ALU computes address **alu** = **R[rs1]** + **imm**.

Write memory at address **alu**.

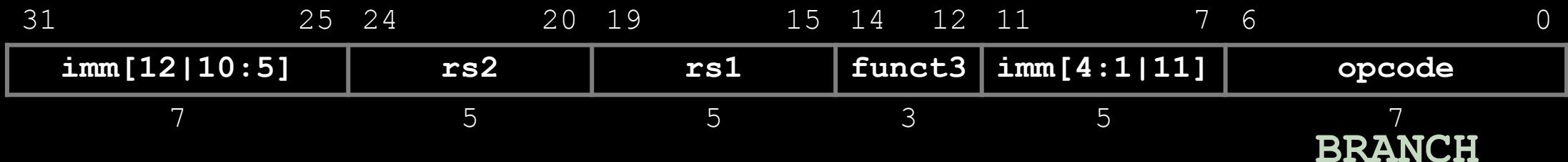


Adding B-Type (Branches)

- Implementing Loads
- Implementing Stores
- Adding B-Type (Branches)
- Designing the Immediate Generation Block, Part 2
- Adding Jumps `jal`, `jalr`
- Adding U-Types
- And in Conclusion...

B-Format: Conditional Branch

- B-Format (textbook: SB-Type) close to S-Format:
opname rs1 , rs2 , Label



- New Immediate Format!
- PC state element now conditionally changes:
 - RegFile $R[rs1], R[rs2]$ (read only, for branch comparison)
 - PC $PC = PC + imm$ (if branch taken)
 $PC = PC + 4$ (otherwise, not taken)

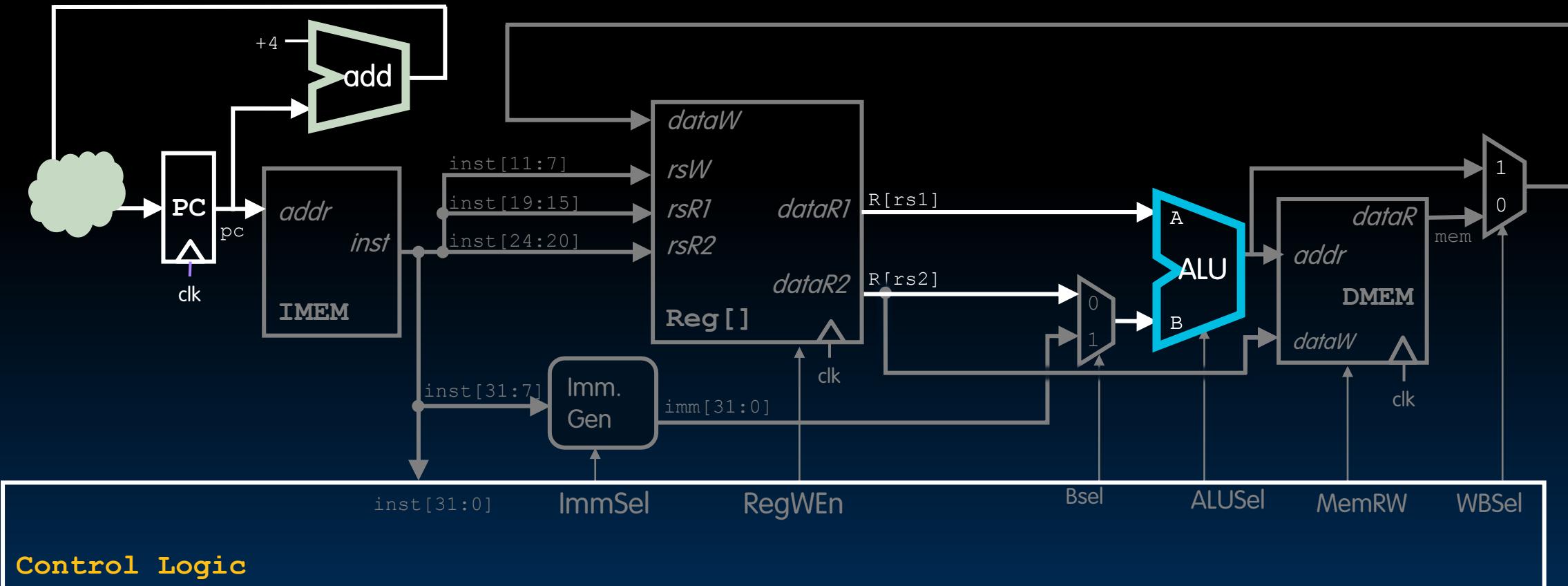
What do Branches Need to Compute?

$PC = PC + imm$?

$PC = PC + 4$

Compare $R[rs1], R[rs2]$?

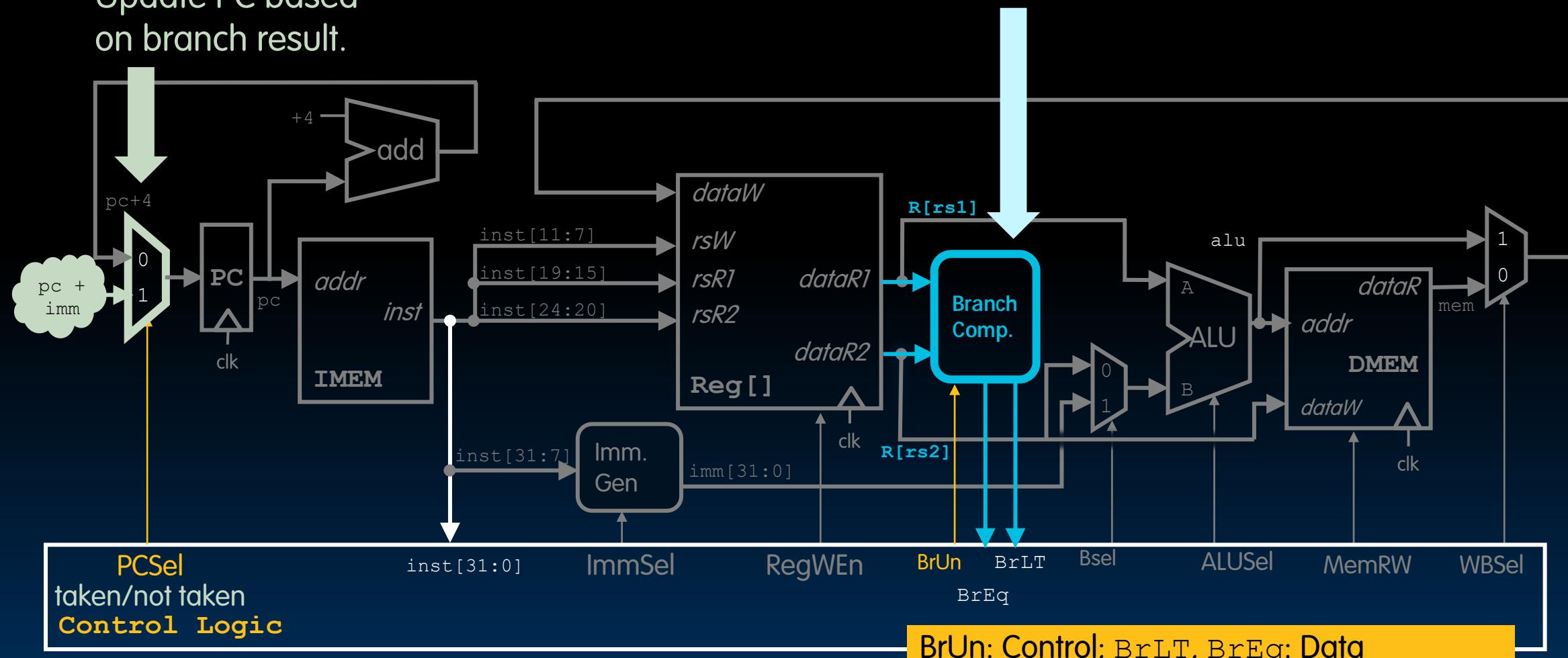
Only one ALU accessible in the same clock cycle.
Need more hardware!!



The Branch Comparator Block

Compare R[rs1], R[rs2],
feed result to Control Logic.

Update PC based
on branch result.



The Branch Comparator Block

- The Branch Comparator is a combinational logic block.

- Input:
 - Two data busses **A** and **B** (datapath $R[rs1]$ and $R[rs2]$, respectively)
 - BrUn** ("Branch Unsigned") control bit

- Control Logic:**

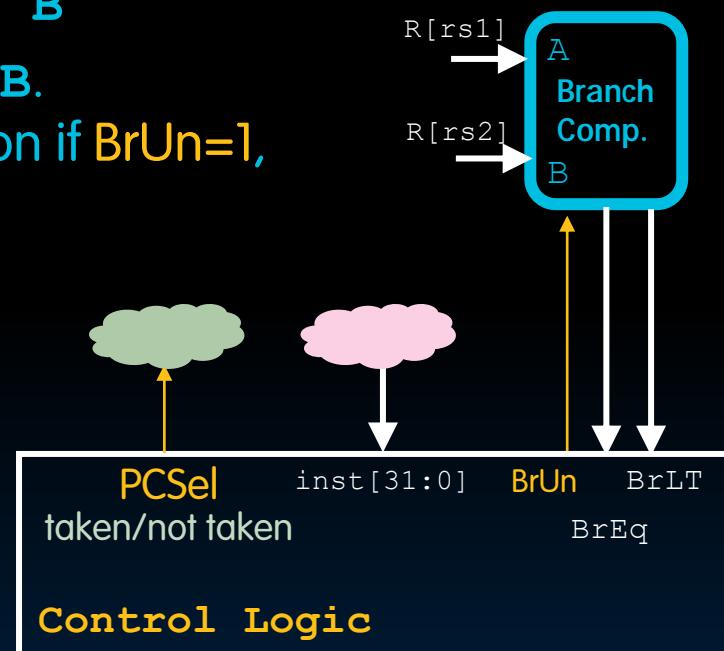
- Set **BrUn** based on current instruction, $inst[31:0]$.
- Set **PCSel** based on branch flags **BrLT**, **BrEq**.

- Examples:**

- blt:**
 - If **BrLT=1** and **BrEq=0**, then **PCSel=taken**.
- bge:** $(A \geq B) = \overline{A < B}$
 - If **BrLT=0**, then **PCSel=taken**.

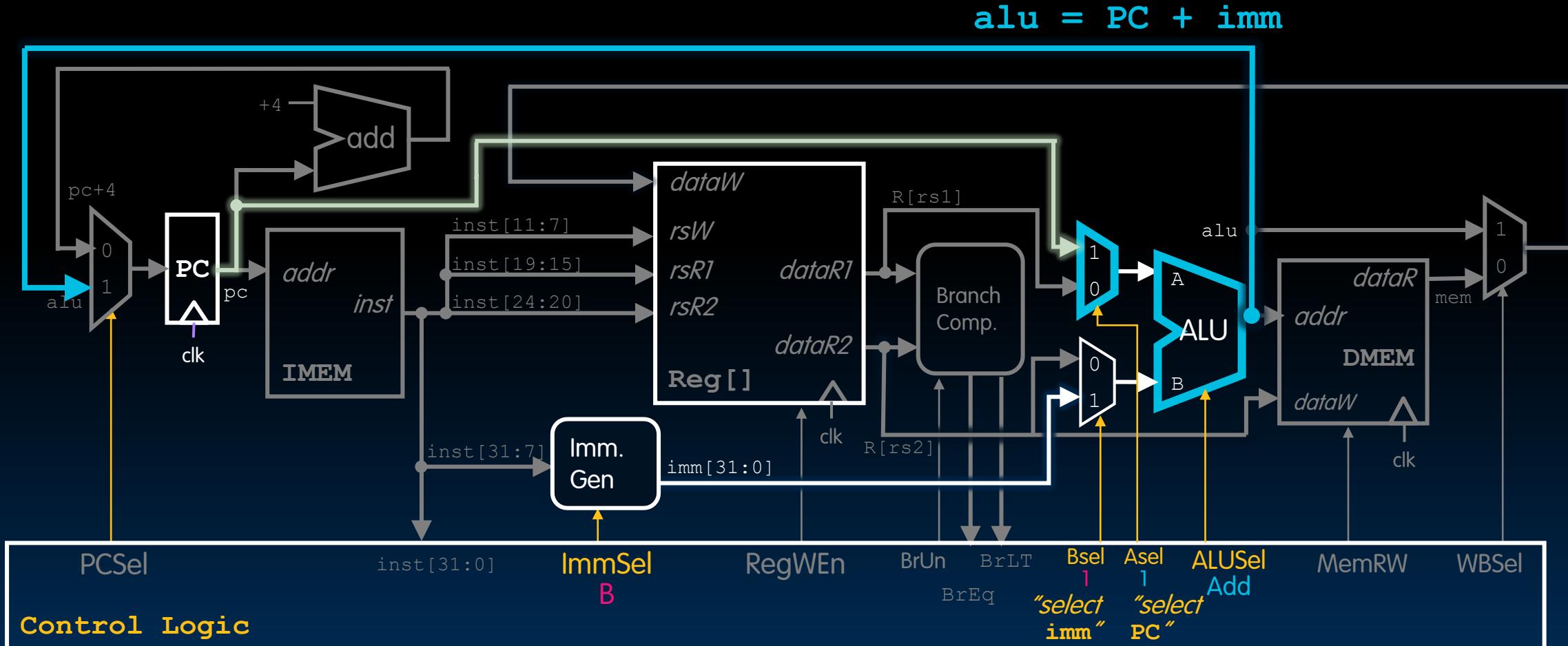
- Output:

- BrEq** flag: 1 if $A == B$
- BrLT** flag: 1 if $A < B$.
Unsigned comparison if $BrUn=1$, signed otherwise.



The ALU computes PC + Imm

- To compute the address to branch to, use the ALU:



Lighting the Branch Datapath

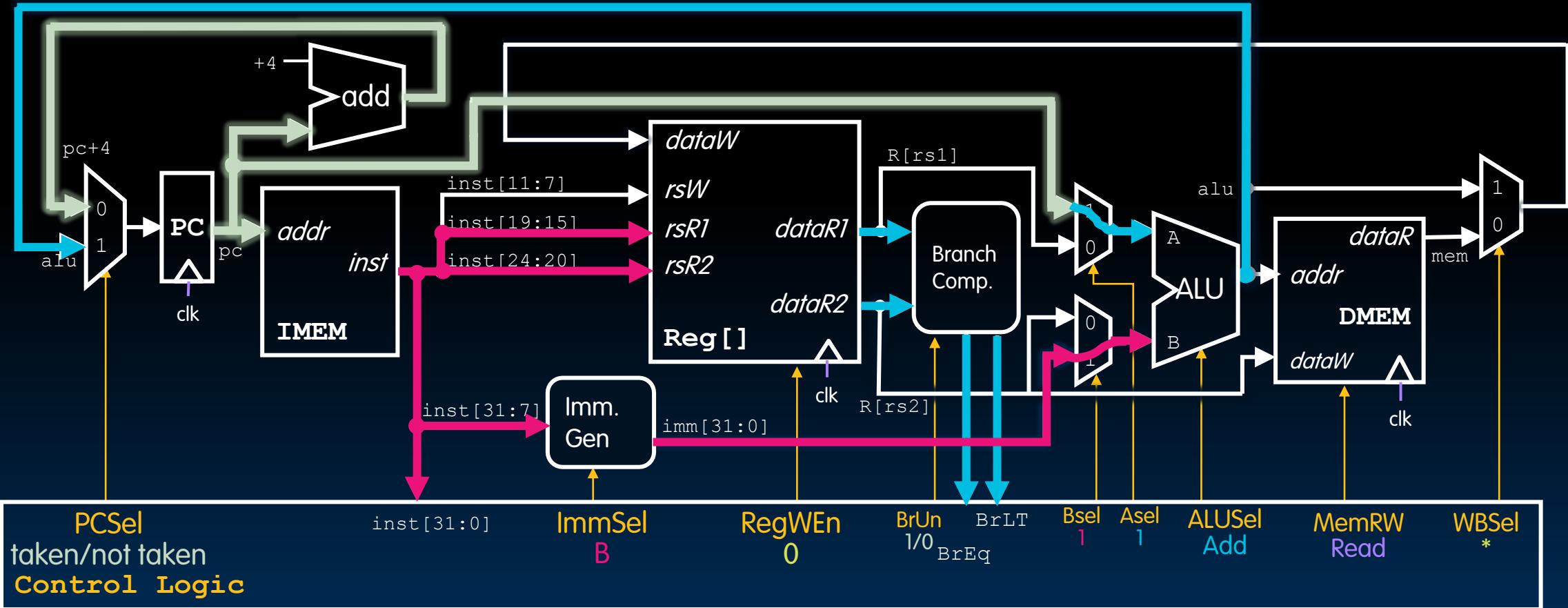
If PCSel=taken, update PC to **ALU output**. Else, update to next instruction **PC + 4**.

Build `imm` from B-type instruction.

- Compute branch; feed to Control.
 - Compute PC + imm.

Don't write to memory.

Don't write to
registers.



Designing the Immediate Generation Block, Part 2

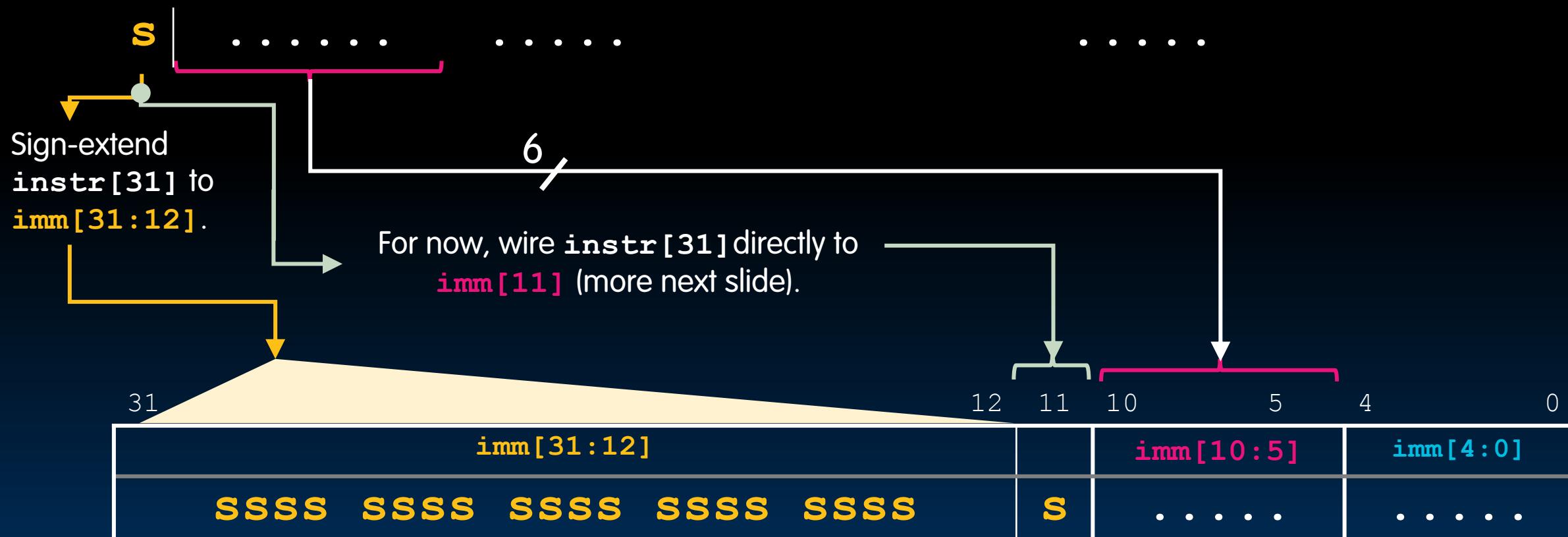
- Implementing Loads
- Implementing Stores
- Adding B-Type (Branches)
- Designing the Immediate Generation Block, Part 2
- Adding Jumps `jal`, `jalr`
- Adding U-Types
- And in Conclusion...

I-Type and S-Type Immediates



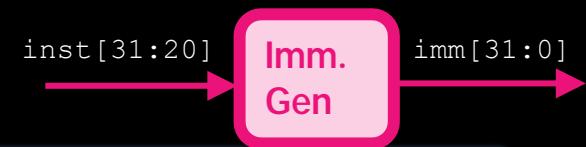
Instruction `inst[31:0]`

	31	30		25	24		20	19		15	14	12	11		7	6		0
I-Type		imm[11 10:5]		imm[4:0]		rs1		funct3		rd					I-OPCODE			
S-Type		imm[11 10:5]			rs2		rs1		funct3		imm[4:0]				S-OPCODE			



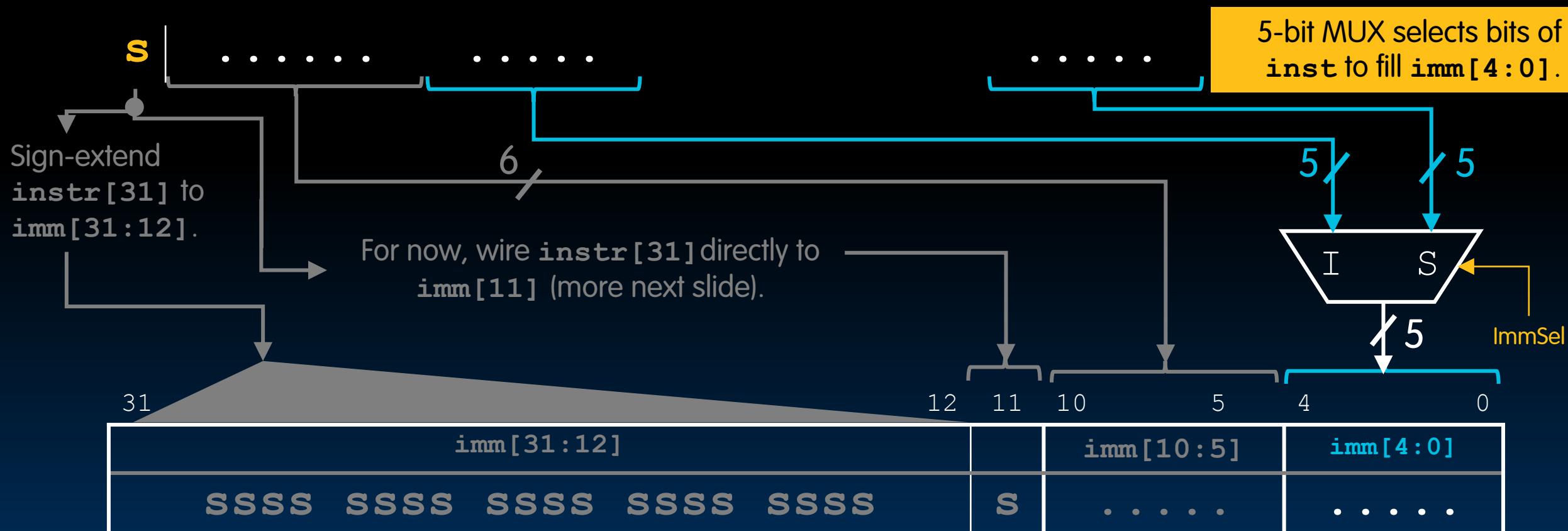
Berkeley Immediate imm[31:0]

I-Type and S-Type Immediates



Instruction $\text{inst}[31:0]$

	31	30	25	24	20	19	15	14	12	11	7	6	0
I-Type			imm[11 10:5]		imm[4:0]		rs1	funct3		rd			I-OPCODE
S-Type			imm[11 10:5]			rs2	rs1	funct3		imm[4:0]			S-OPCODE





B-Type Immediates



Instruction `inst[31:0]`

The diagram illustrates the MIPS instruction format across three rows: I-Type, S-Type, and B-Type. The columns represent bit positions from 31 to 0. The fields are as follows:

- I-Type:** imm[11:5], imm[4:0], rs1, funct3, rd, I-OPCODE
- S-Type:** imm[11|10:5], rs2, rs1, funct3, imm[4:0], S-OPCODE
- B-Type:** imm[12|10:5], imm[4:1|11], B-OPCODE

Below the rows, there are two boxes with arrows pointing downwards:

- A purple box containing the letter **S**, with an arrow pointing to the bottom-left.
- A yellow box containing a small square with a dot inside, with an arrow pointing to the bottom-right.

instr[31] is always
the sign bit.

MUX for `imm[11:0]`

- S: **instr[31]**
 - B: **instr[7]**

MUX for imm[0]:

- S: **instr[7]**
 - B: 0 (implicit 0;
half-words → bytes)



Berkeley Immediate imm[31:0]

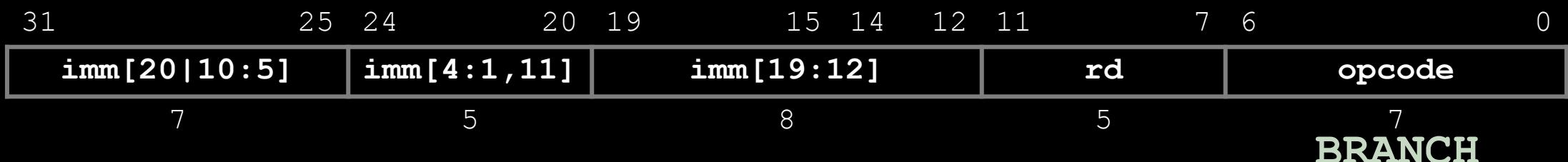
Adding Jumps `jal, jalr`

- Implementing Loads
- Implementing Stores
- Adding B-Type (Branches)
- Designing the Immediate Generation Block, Part 2
- Adding Jumps `jal, jalr`
- Adding U-Types
- And in Conclusion...

J-Format: Unconditional Jump

- J-Format:

jal rd, Label



- New immediate format! (see Project 3)
- State Elements updated:

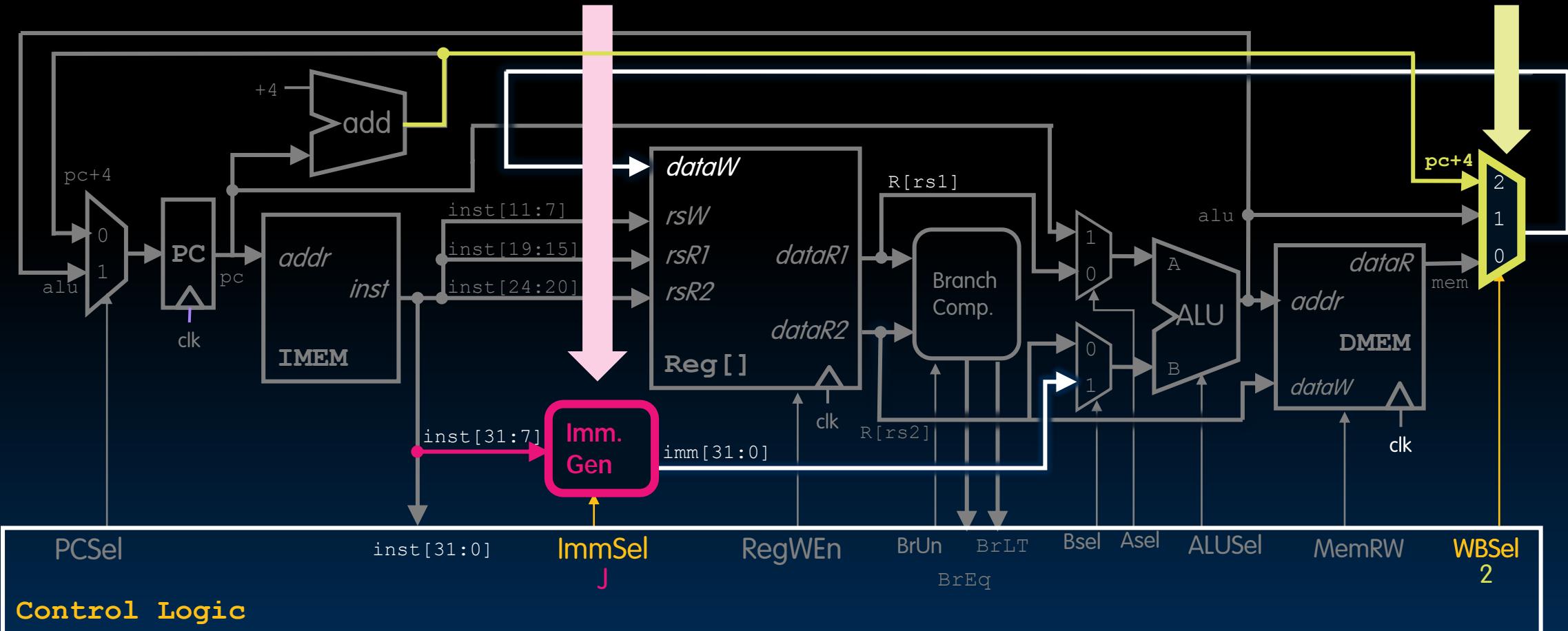
- PC $PC = PC + \text{imm}$ (unconditional PC-relative jump)
- RegFile $rd = PC + 4$ } Save return address to RegFile destination register.

Block Updates Needed for JAL

$$\begin{aligned} \text{PC} &= \text{PC} + \text{imm} \\ \text{rd} &= \text{PC} + 4 \end{aligned}$$

Immediate Generation Block
needs to support J-Types: 20-bit half-words \rightarrow byte offset.

To save $\text{rd} = \text{PC} + 4$,
WBSel now controls
a 3-input MUX.



Lighting the JAL Datapath

$$\begin{aligned} \text{PC} &= \text{PC} + \text{imm} \\ \text{rd} &= \text{PC} + 4 \end{aligned}$$

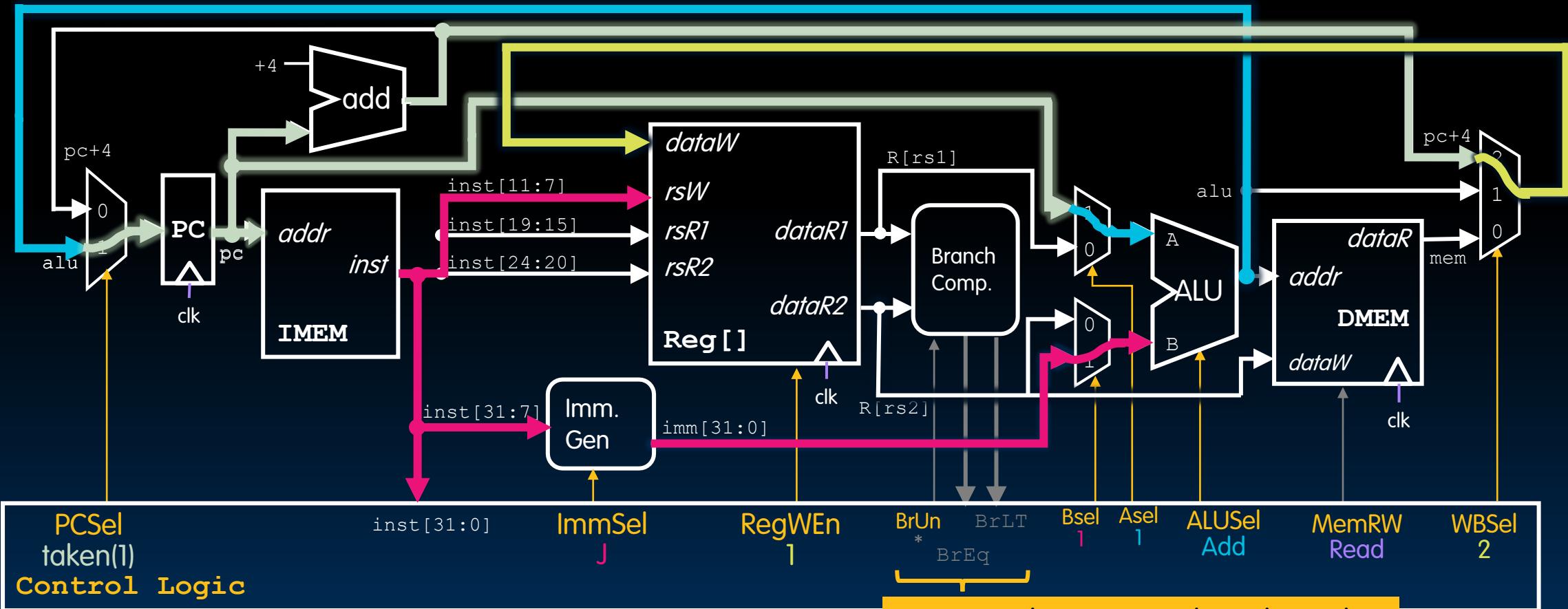
- Feed PC into blocks.
- Write ALU output to PC.

Generate byte offset **imm**
for 20-bit PC-relative jump.

Compute **PC + imm**.

Write **PC + 4** to
destination register.

Don't write to memory.



I-Format Instruction Layout: jalr

- `jalr` uses I-Format:

jalr rd,rs1,imm



- ## ▪ Two changes to state:

- PC $PC = R[rs1] + imm$ (absolute addressing)
 - RegFile $R[rd] = PC + 4$

- I-Format means jalr uses the same immediates as arithmetic/loads!

- In other words, `imm` is already a byte offset.

Control ImmSel is based on instruction format, not instruction. So far: I, S, B, J

Lighting the JALR Datapath

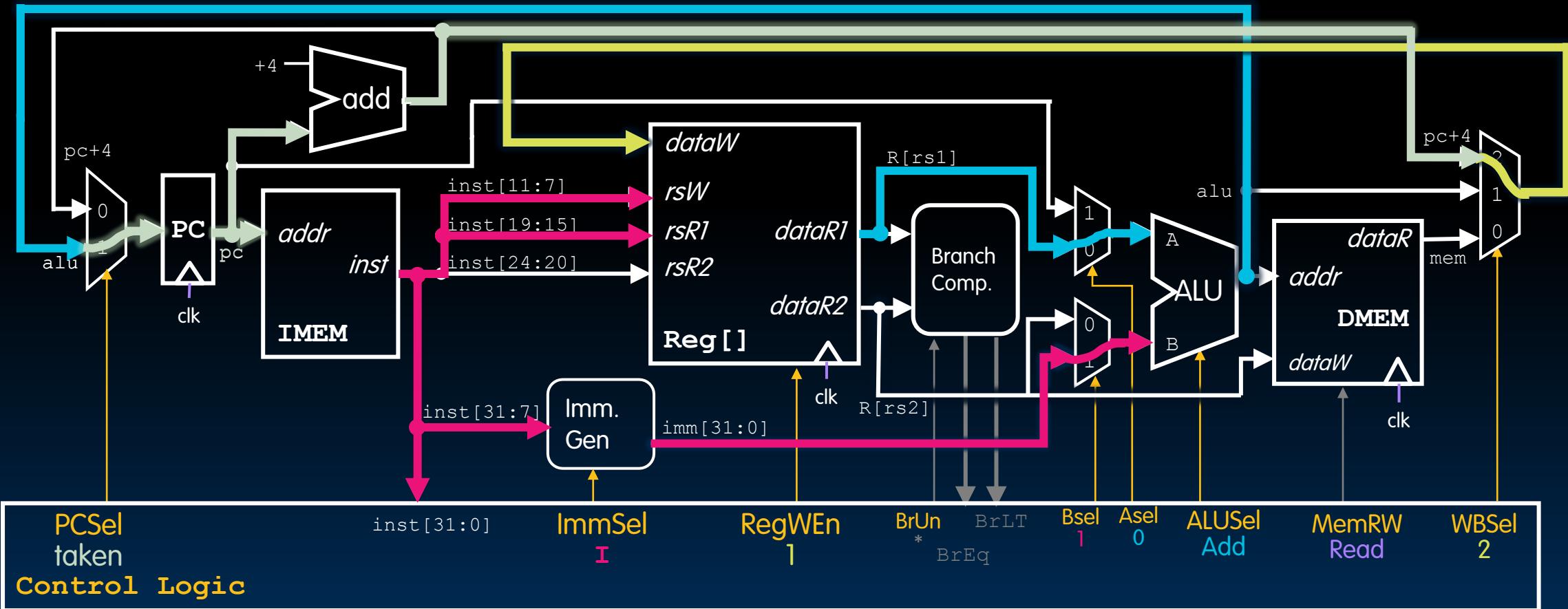
$$\begin{aligned} \text{PC} &= R[\text{rs1}] + \text{imm} \\ R[\text{rd}] &= \text{PC} + 4 \end{aligned}$$

- Feed PC into blocks.
- Write ALU output to PC.

Generate 12-bit **imm**
(I-Format).

Compute **rs1 + imm**.

Write **PC + 4** to destination register.
Don't write to memory.



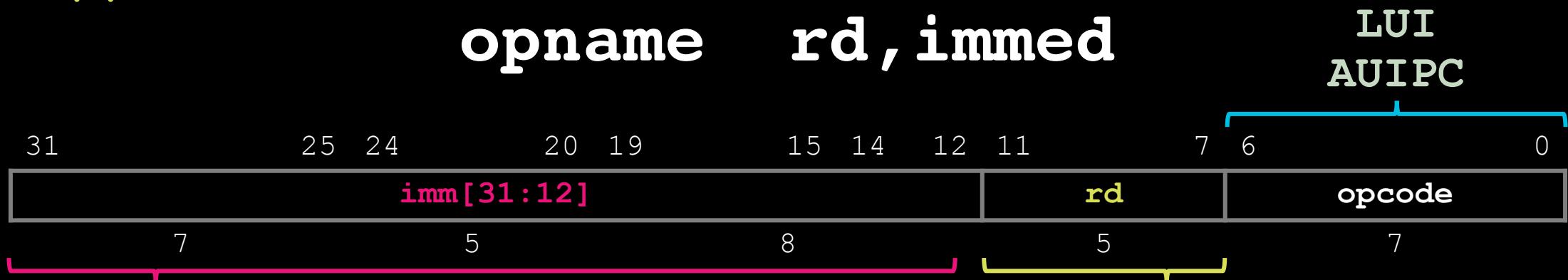
Adding U-Types

- Implementing Loads
- Implementing Stores
- Adding B-Type (Branches)
- Designing the Immediate Generation Block, Part 2
- Adding Jumps `jal`, `jalr`
- Adding U-Types
- And in Conclusion...

- “Upper Immediate” instructions:

opname rd , immed

LUI
AUIPC



Immediate represents *upper 20 bits* of a 32-bit immediate **imm**.

"Destination" Register

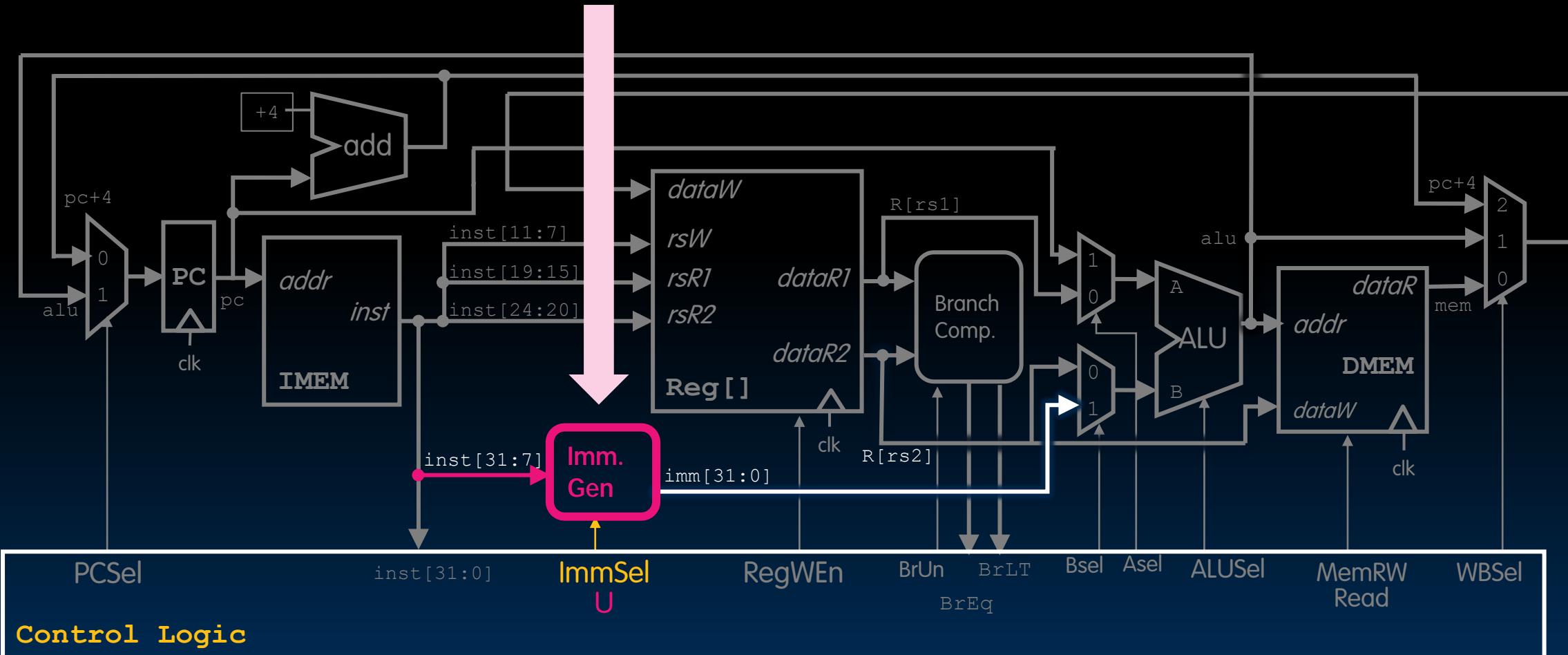
- New immediate format! (see Project 3)

- Used for two instructions:

- lui**: Load Upper Immediate
- auipc**: Add Upper Immediate to PC
- Both increment PC to next instruction and save to destination register.

U-Format Block update: Immediates

Generate **imm** with
upper 20 bits. (U-format)



Lighting the LUI Datapath

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

$$R[\text{rd}] = \text{imm}$$

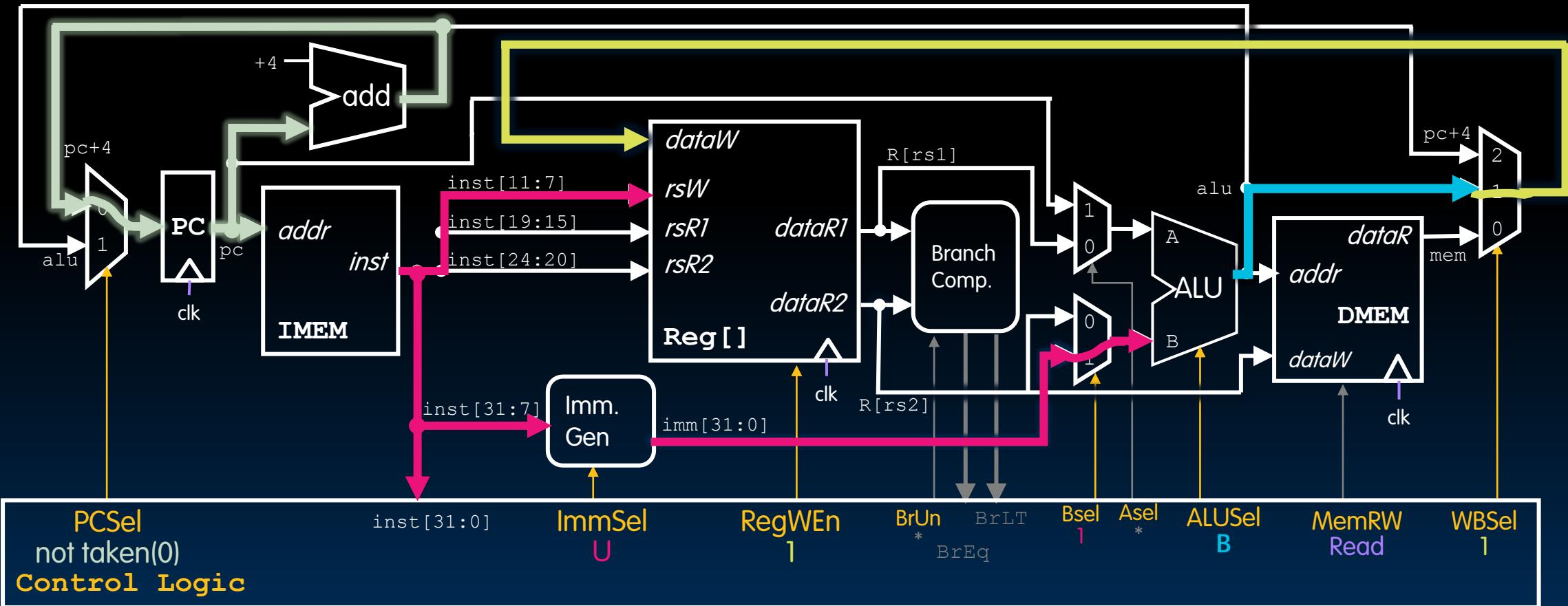
Increment PC to next instruction.

Generate **imm** with upper 20 bits. (U-format)

Grab only imm!
(ALUSel=B)

Write result to destination register.

Don't write to memory.



Lighting the AUIPC Datapath

$$\begin{aligned} \text{PC} &= \text{PC} + 4 \\ \text{R[rd]} &= \text{PC} + \text{imm} \end{aligned}$$

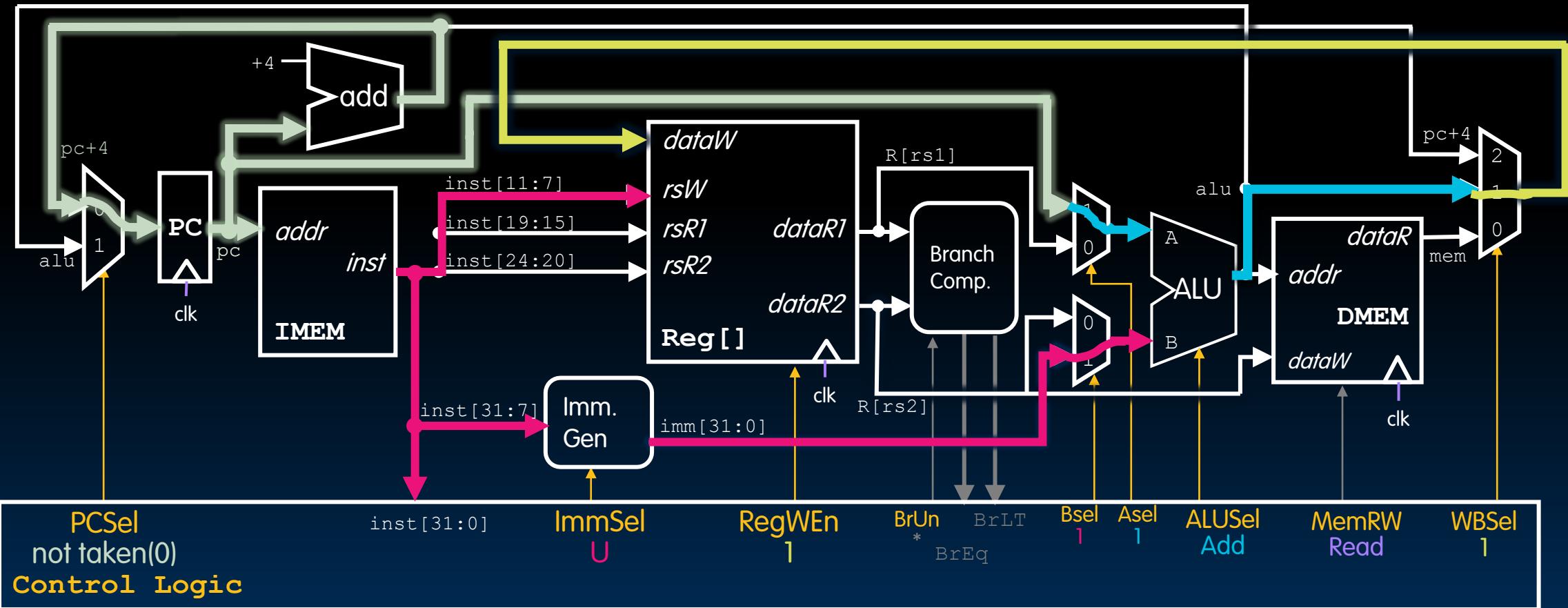
Increment PC to next instruction.

Generate **imm** with upper 20 bits. (U-format)

Add PC + imm!

Write result to destination register.

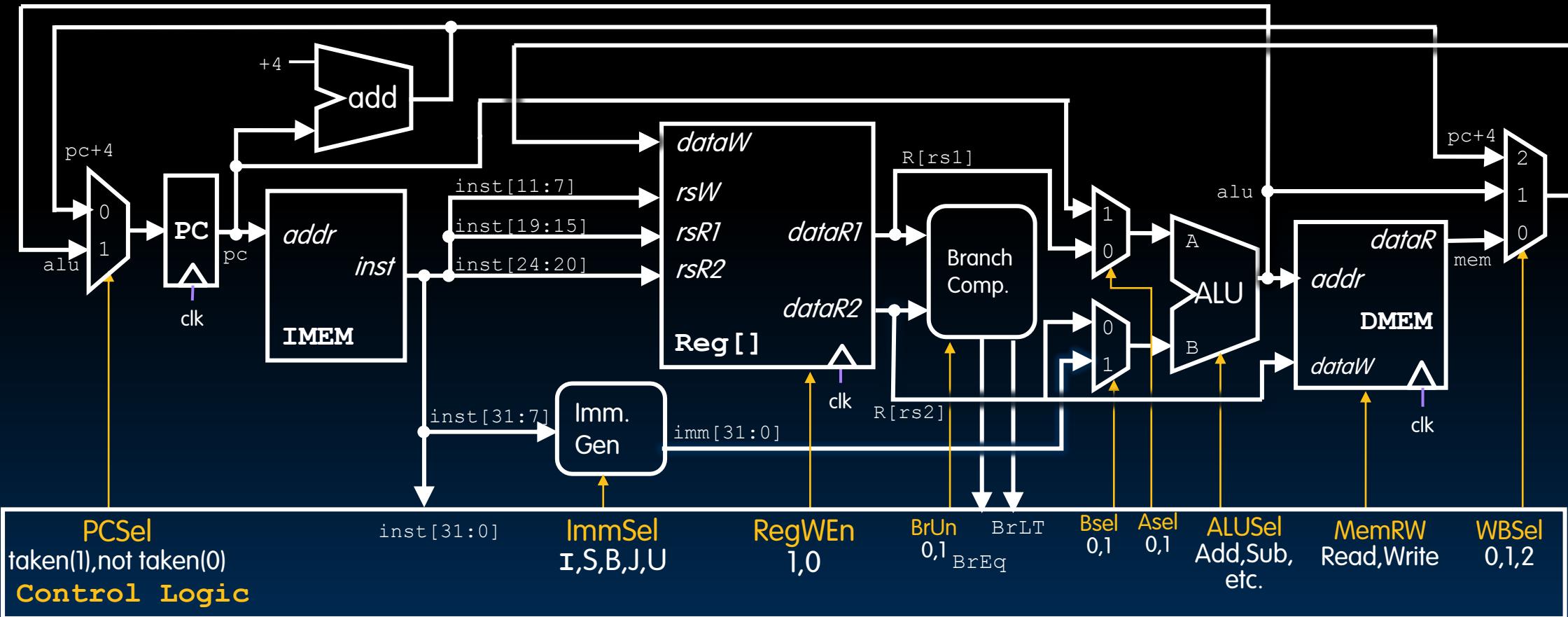
Don't write to memory.



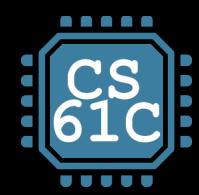
And in Conclusion...

- Implementing Loads
- Implementing Stores
- Adding B-Type (Branches)
- Designing the Immediate Generation Block, Part 2
- Adding Jumps `jal`, `jalr`
- Adding U-Types
- And in Conclusion...

Complete RV32I Datapath!



Garcia, Yan



Have a Great Weekend!