


CS61C: Great Ideas in Computer Architecture (Machine Structure)

Instructors: Rosalie Fang, Charles Hong, Jero Wang

Lecture feedback:

<https://tinyurl.com/fyr-feedback>

Announcements

- Midterm!!!
 - Friday 7/14, **5-7PM** PT
 - Logistics on Ed
 - Lecture 1-11 are in scope!
 - Tomorrow 3-5PM: Review session #2, focused on RISC-V, SDS & FSM
 - Good luck! 
- Project 2
 - 2B due next Tuesday
 - Quite helpful for understanding & utilizing calling convention; would recommend at least attempting Task 6 before the midterm!
- No labs tomorrow
- No OH on Friday

Recall... Great Idea #1: Abstraction (Layers of Representation/Interpretation)

High Level Language Program (e.g., C)

```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

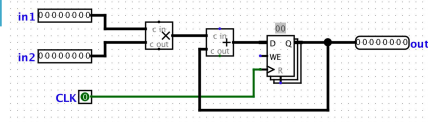
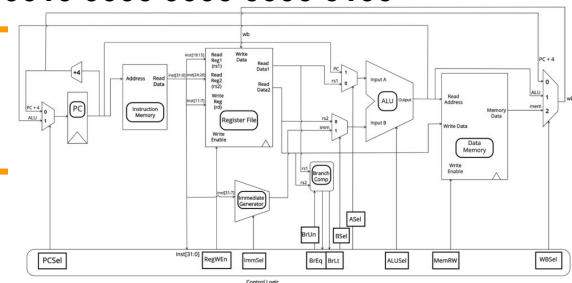
| *Assembler*

Machine Language Program (RISC-V)

```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 01000
1010 1110 0001 0010 0000 0000 0000 00000
1010 1101 1110 0010 0000 0000 0000 0100
```

Hardware Architecture Description (e.g., block diagrams)

Logic Circuit Description (Circuit Schematic Diagrams)



CS 61C Hardware Roadmap

Higher level languages, C, RISC-V, Machine code

Processor (CPU)

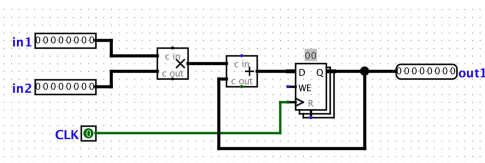
- **Datapath**
- Control Logic
- Pipelining

Synchronous Digital Systems

- Combinatorial Logic
- Sequential Logic

Transistors

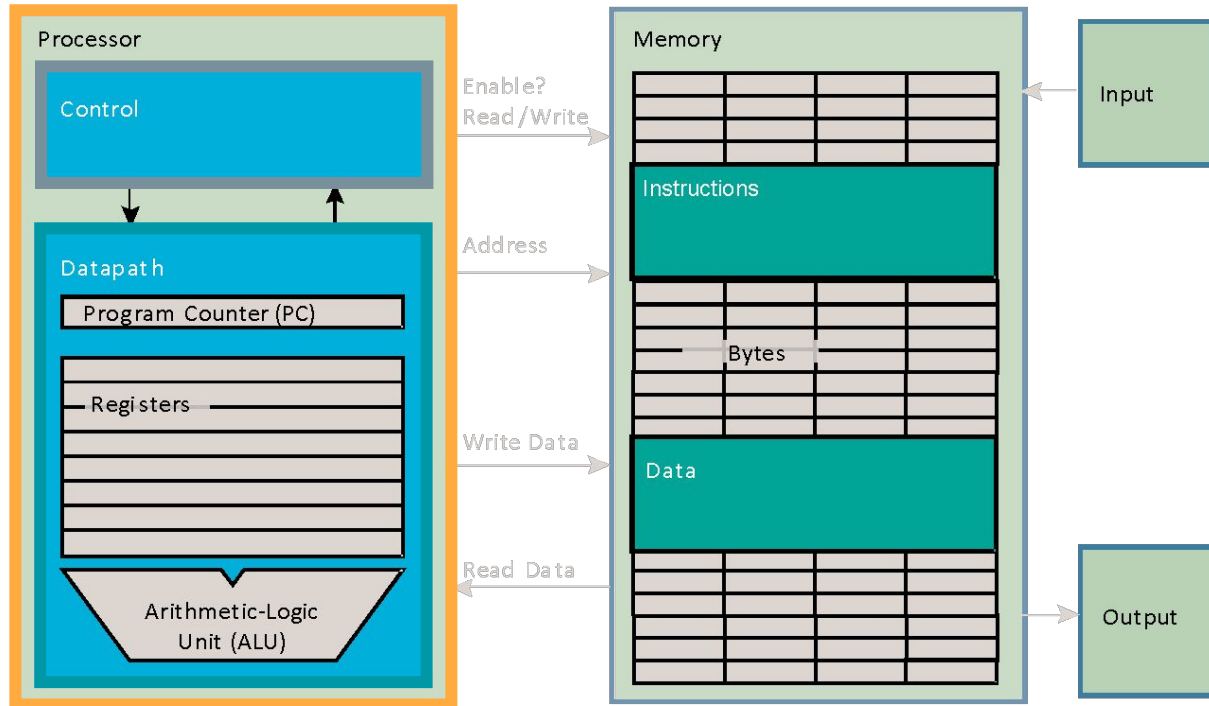
Today, we're building a circuit that can run every base RISC-V instruction!



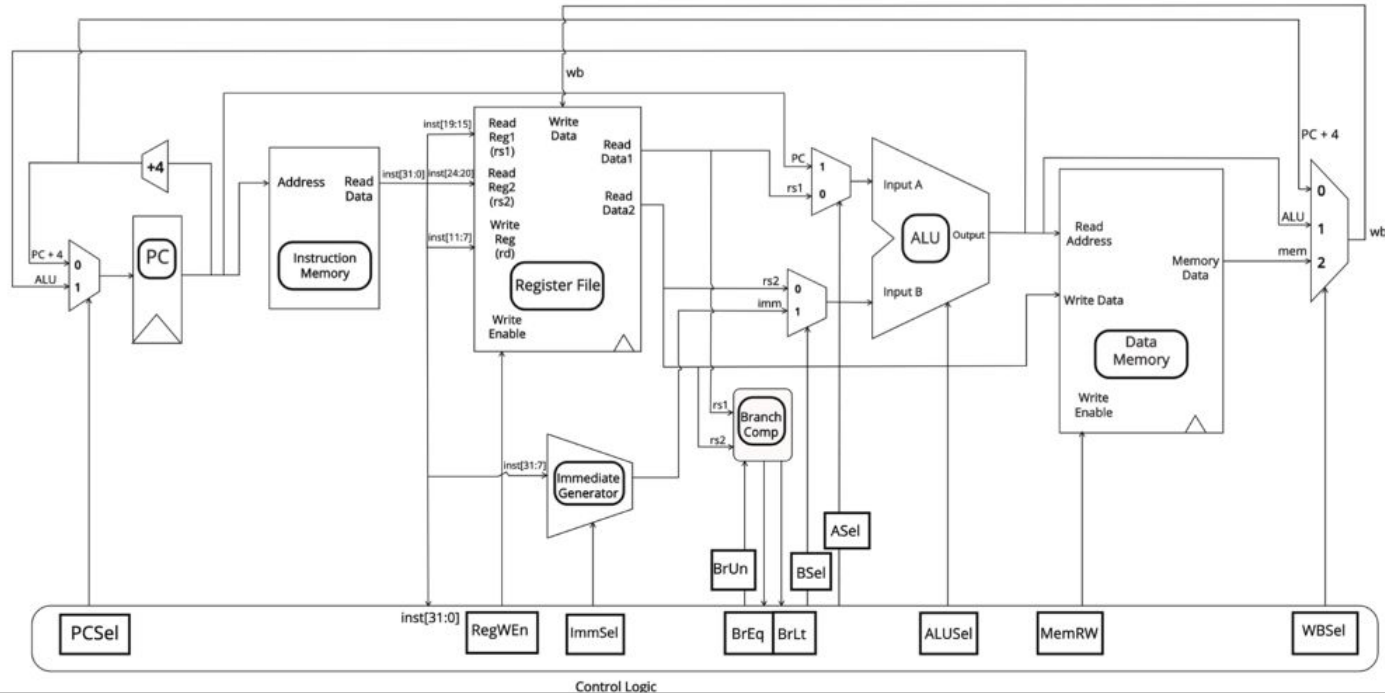
How do we build a Single-Core Processor?

Processor (CPU): the active part of the computer that does all the work (data manipulation and decision-making).

- **Datapath** (“the brawn”): portion of the processor that contains hardware necessary to perform operations required by the processor.
- **Control** (“the brain”): portion of the processor (also in hardware) that tells the datapath what needs to be done.



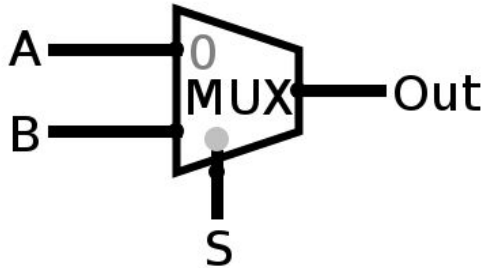
A little taste on what you're going to be able to do after this lecture... (& in project 3!)



Review: Subcircuits

Combinatorial Logic: Multiplexer (MUX)

- Intuitively:
 - If S is 0, output the value in A.
 - If S is 1, output the value in B.
 - S is the “select” bit
- Usually drawn like this:

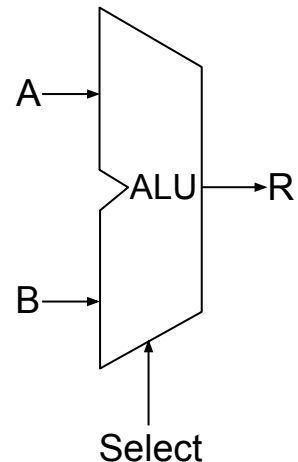
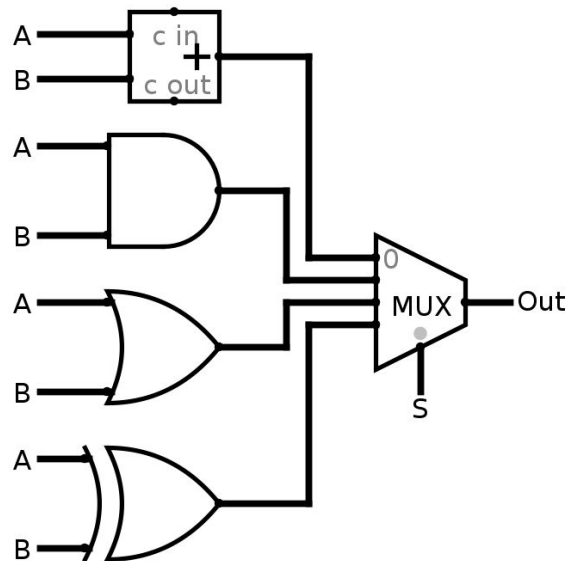


S	A	B	Out
0	0	0	0
1	0	0	0
0	0	1	0
1	0	1	1
0	1	0	1
1	1	0	0
0	1	1	1
1	1	1	1

$$\text{Out} = (A)(!S) + (B)(S)$$

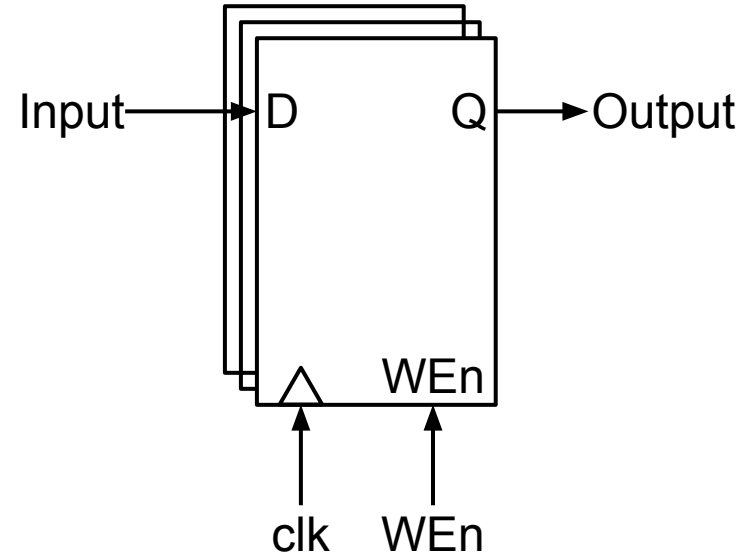
Combinatorial Logic: Arithmetic Logic Unit (ALU)

- Inputs:
 - 32-bit input A
 - 32-bit input B
 - 2-bit operation selector S
- Outputs:
 - 32-bit result R
- Behavior:
 - If $S=0b00$, set $R=A+B$ (addition)
 - If $S=0b01$, set $R=A \& B$ (bitwise AND)
 - If $S=0b10$, set $R=A | B$ (bitwise OR)
 - If $S=0b11$, set $R=A \wedge B$ (bitwise XOR)



Sequential Logic: Registers

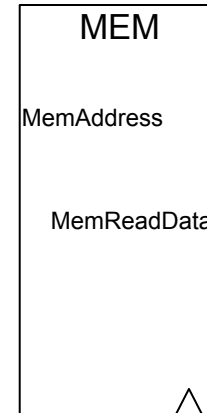
- Inputs:
 - n -bit value D
 - One-bit clock value
 - One-bit write-enable
- Outputs:
 - n -bit value Q
- Behavior:
 - If $WEn=1$, then on the rising edge of the clock, set $Q=D$
 - At all other times, do nothing



Sequential Logic: Read-Only Memory

- Behavior:
 - MemAddress identifies an address in memory.
 - Reading from memory: Set MemReadData to the value at that address.

Inputs	MemAddress	32 bits
	Clock	1 bit
Output	MemReadData	32 bits

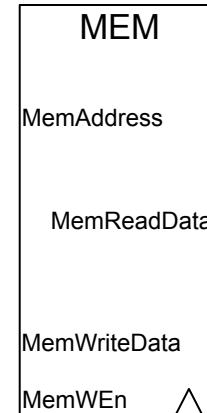


Sequential Logic: Read/Write Memory

- Behavior:

- MemAddress identifies an address in memory.
- Reading from memory: Set MemReadData to the value at that address.
- Writing to memory: If MemWEn=1, then on the next rising edge of the clock, write the value in MemWriteData at that address.

Inputs	MemAddress	32 bits
	MemWriteData	32 bits
	MemWEn	1 bit
	Clock	1 bit
Output	MemReadData	32 bits



Datapath for **add**

List of RISC-V Instructions

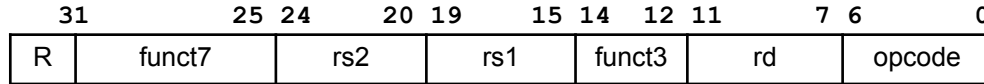
Instruction	Name	Type
add	ADD	R
sub	SUBtract	R
and	bitwise AND	R
or	bitwise OR	R
xor	bitwise XOR	R
sll	Shift Left Logical	R
srl	Shift Right Logical	R
sra	Shift Right Arithmetic	R
slt	Set Less Than (signed)	R
sltu	Set Less Than (Unsigned)	R
addi	ADD Immediate	I
andi	bitwise AND Immediate	I
ori	bitwise OR Immediate	I
xori	bitwise XOR Immediate	I

Instruction	Name	Type
slli	Shift Left Logical Immediate	I*
srli	Shift Right Logical Immediate	I*
srai	Shift Right Arithmetic Immediate	I*
slti	Set Less Than Immediate (signed)	I
sltiu	Set Less Than Immediate (Unsigned)	I
lb	Load Byte	I
lbu	Load Byte (Unsigned)	I
lh	Load Half-word	I
lhu	Load Half-word (Unsigned)	I
lw	Load Word	I

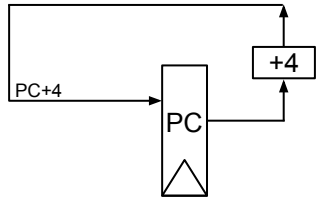
Instruction	Name	Type
sb	Store Byte	S
sh	Store Half-word	S
sw	Store Word	S
beq	Branch if Equal	B
bge	Branch if Greater or Equal (signed)	B
bgeu	Branch if Greater or Equal (Unsigned)	B
blt	Branch if Less Than (signed)	B
bltu	Branch if Less Than (Unsigned)	B
bne	Branch if Not Equal	B
jal	Jump And Link	J
jalr	Jump And Link Register	I
auipc	Add Upper Immediate to PC	U
lui	Load Upper Immediate	U

add instruction

- What's the **add** workflow?
 - **rs1 + rs2**: Add the values in registers rs1 and rs2
 - **rd = rs1 + rs2**: Store the result in register **rd**
 - **PC = PC + 4**: Increment the program counter



Stage 1: Instruction Fetch (IF)

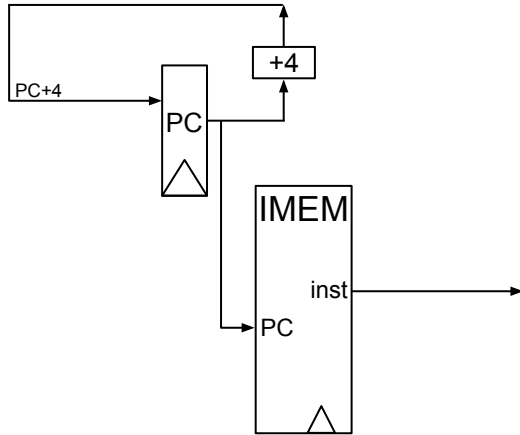


State (sequential)
component!

Here's a circuit that adds 4 to **PC** on each clock cycle.

PC+4 is written to the register on the next rising edge.

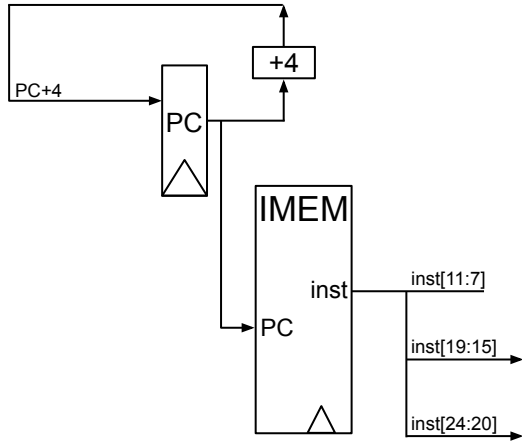
Stage 1: Instruction Fetch (IF)



Let's find out what instruction is at the address in **PC**.

We can input the address into a memory block, and retrieve a 32-bit instruction.

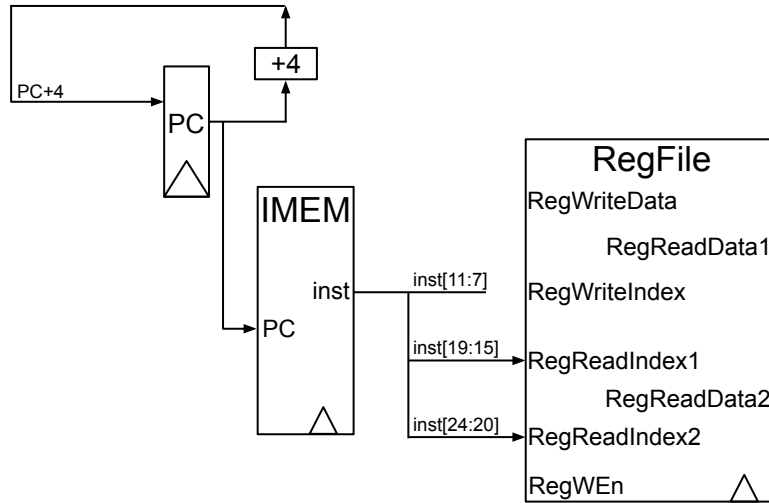
Stage 2: Instruction Decode (ID)



Let's split up the instruction bits into some useful fields.

Now we have the 5-bit values **rs1**, **rs2**, and **rd**.

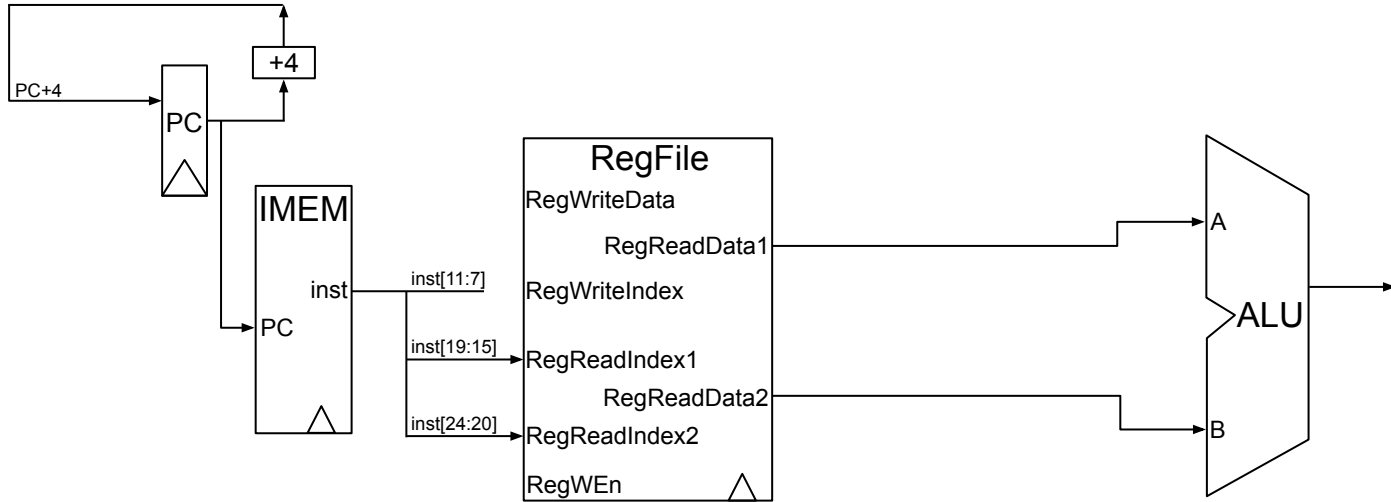
Stage 2: Instruction Decode (ID), Register Read



Let's find out what values are in the **rs1** and **rs2** registers.

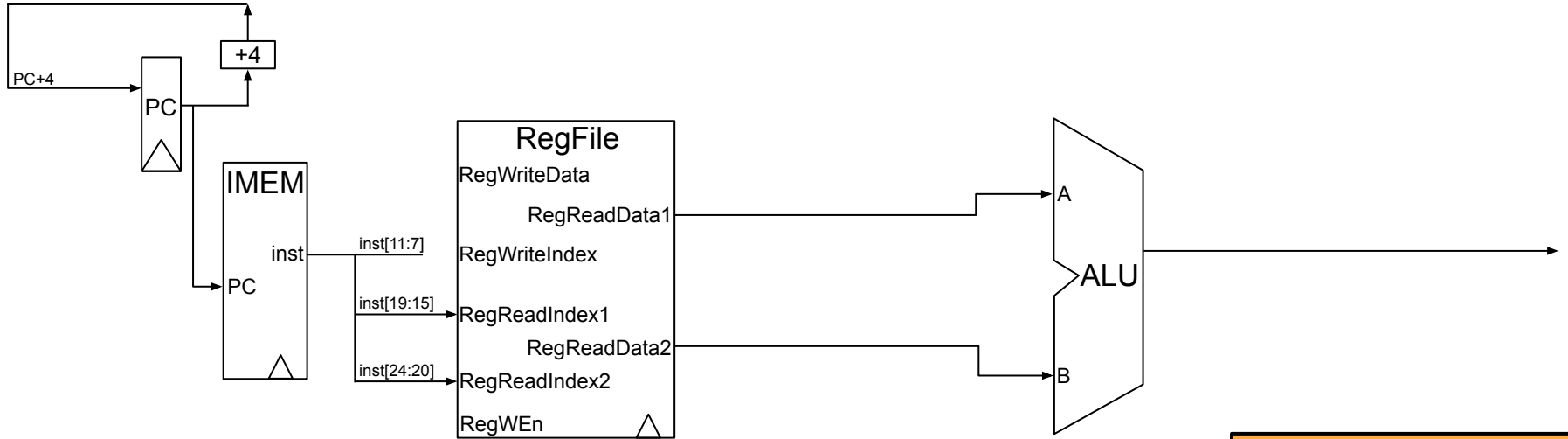
We can input the register numbers into the RegFile and retrieve 32-bit values stored in those registers.

Stage 3: Execute



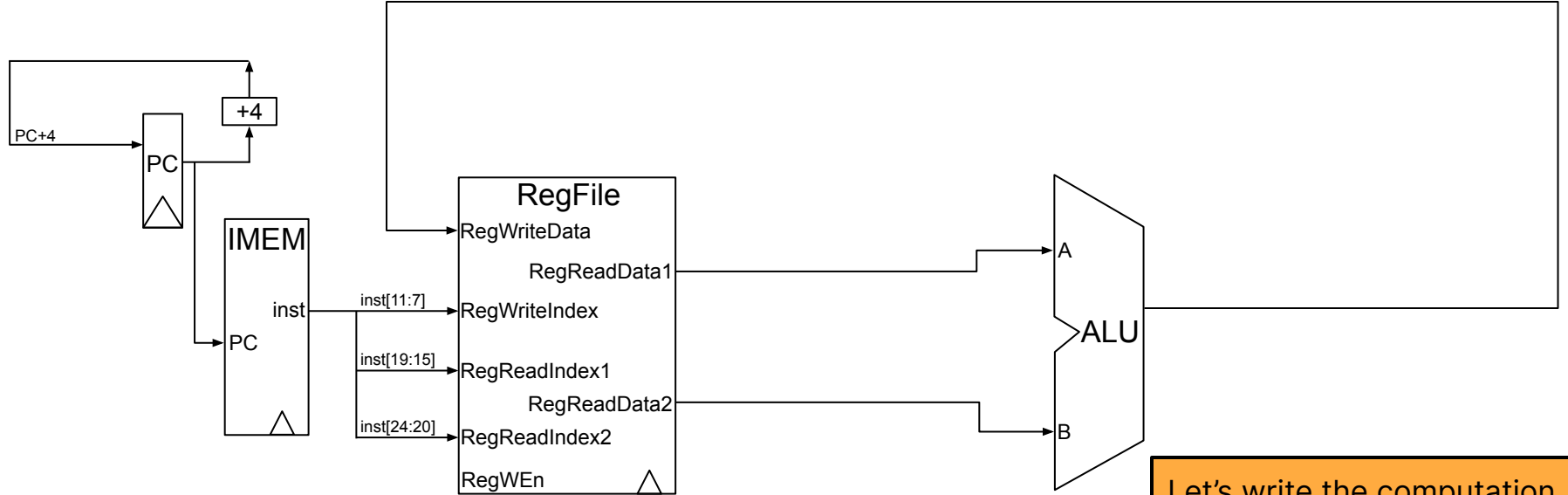
Let's add the 32-bit values stored in **rs1** and **rs2** together.

Stage 4: Memory



We don't need to read from or write to memory, so we can do nothing in this stage.

Stage 5: Register Write



Let's write the computation result back to register `rd`.

Note that we also set `RegWriteIndex` to `rd`.

Five-Stage Datapath: **add**

1. Instruction Fetch (IF)

- Look up the 32-bit instruction at address PC, using instruction memory IMEM
- Add 4 to PC for the next cycle

2. Instruction Decode (ID), Register Read

- Extract fields from the 32-bit instruction
- Read data from the registers, using the RegFile

3. Execute

- Perform calculation on register data, using the Arithmetic Logic Unit (ALU)

4. Memory

- Read or write to memory (not needed for **add**)

5. Register Write

- Write the result of calculation to a register, using the RegFile

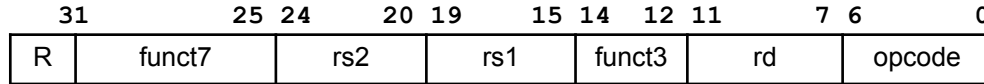
Why Five-Stage Datapath?

- We broke up the processor circuit into five stages
- Why is this useful?
 - A single logic block that runs the entire instruction is bulky and inefficient
 - Smaller stages are easier to design
 - Modularity: Easier to change one stage without affecting the other stages

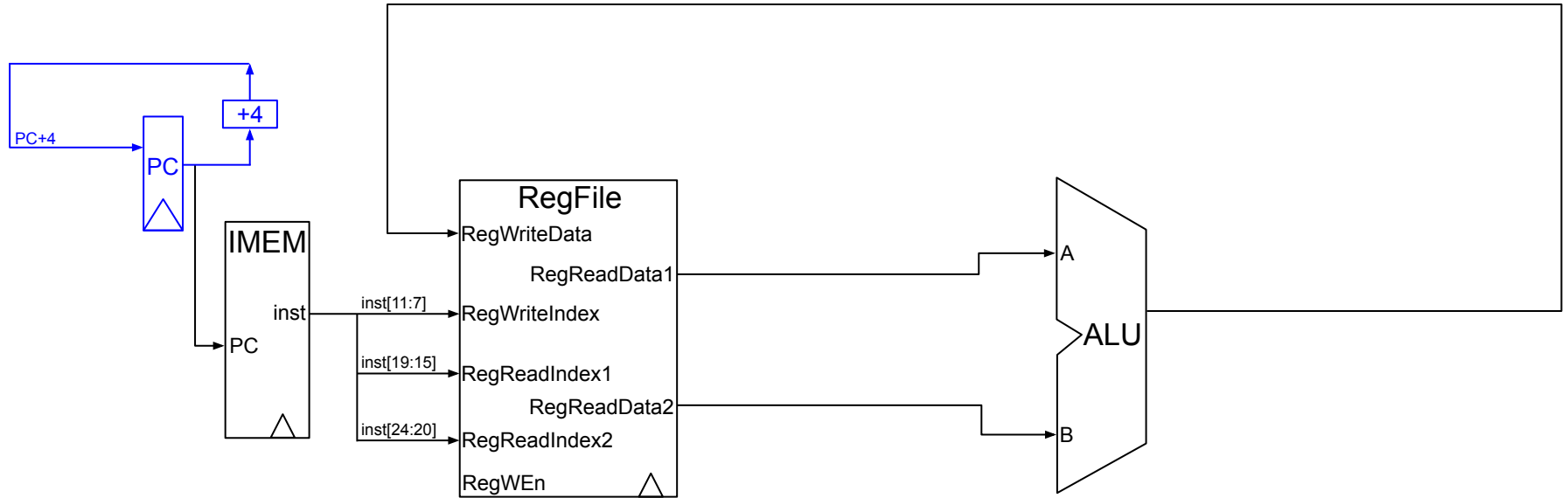
Datapath for **sub**

sub instruction

- What's the **sub** workflow?
 - **rs1 - rs2**: Add the values in registers rs1 and rs2
 - **rd = rs1 - rs2**: Store the result in register **rd**
 - **PC = PC + 4**: Increment the program counter

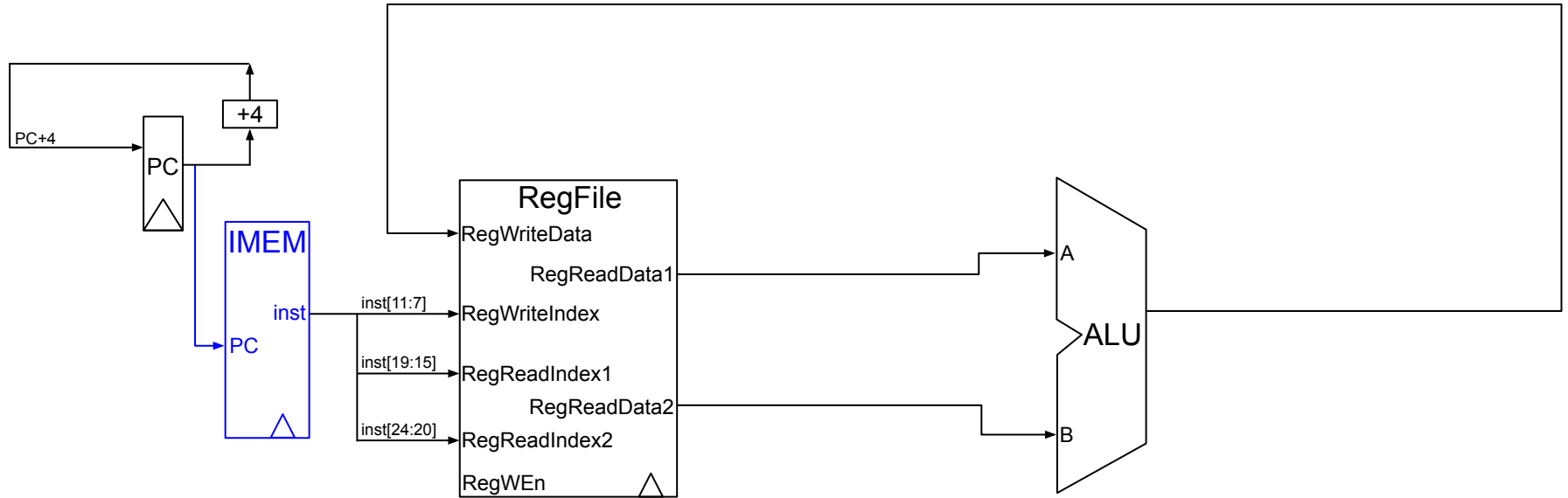


Stage 1: Instruction Fetch (IF)



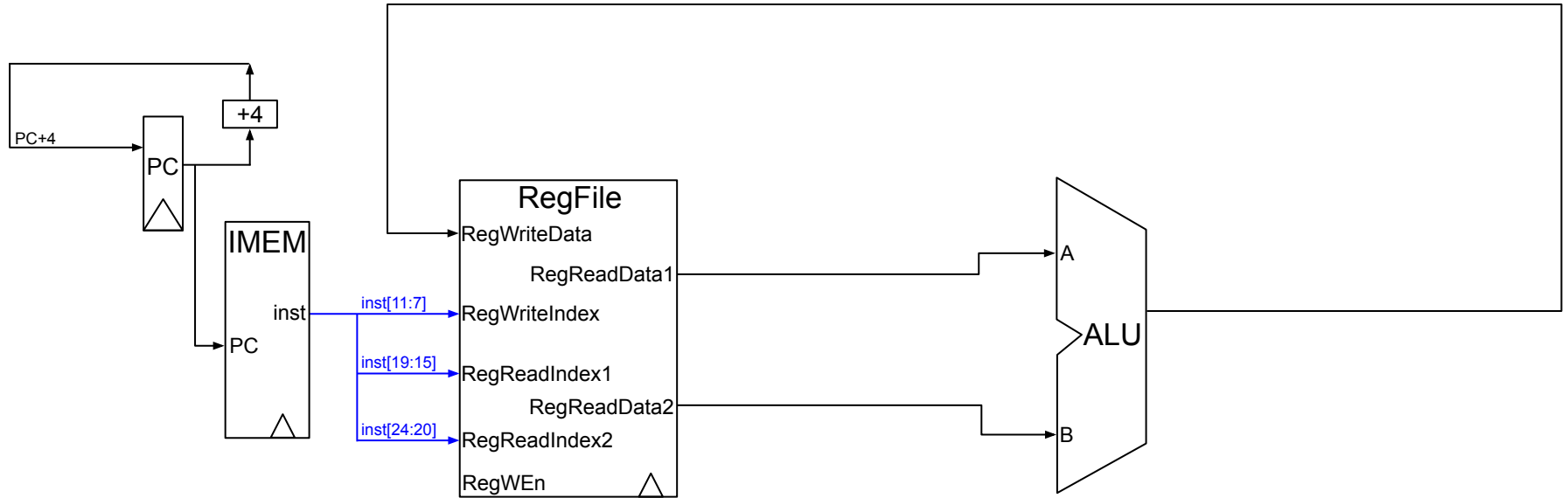
Same as the `add` instruction.
Compute $PC+4$ for the next instruction.

Stage 1: Instruction Fetch (IF)



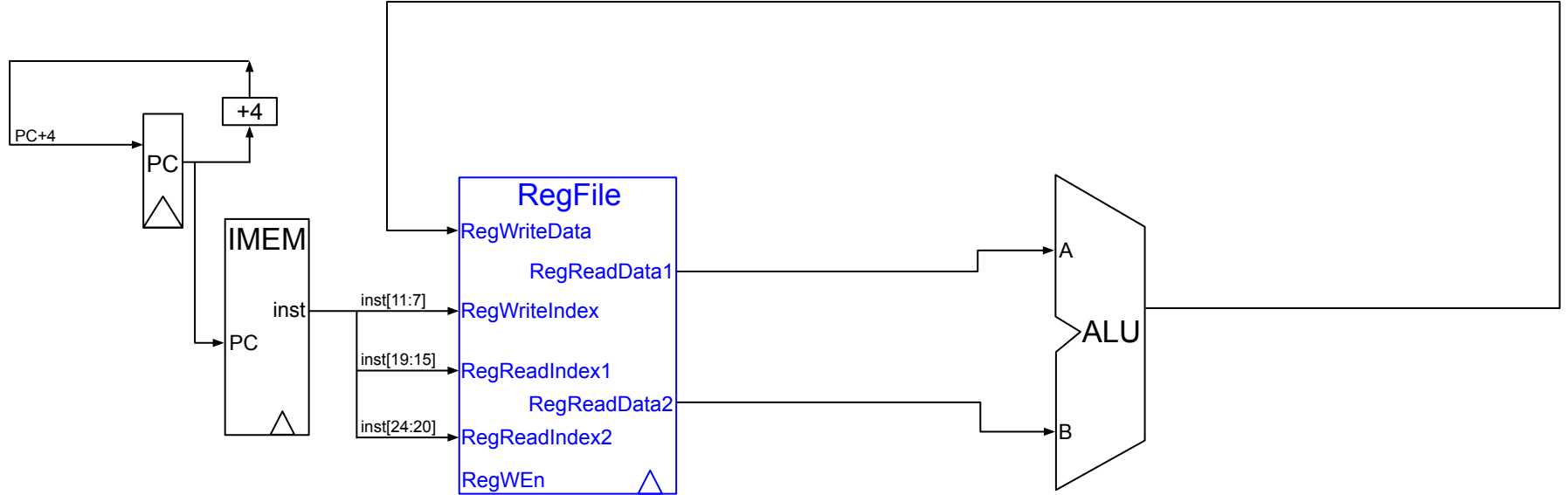
Same as the **add** instruction.
Fetch the 32-bit instruction
from IMEM.

Stage 2: Instruction Decode (ID)



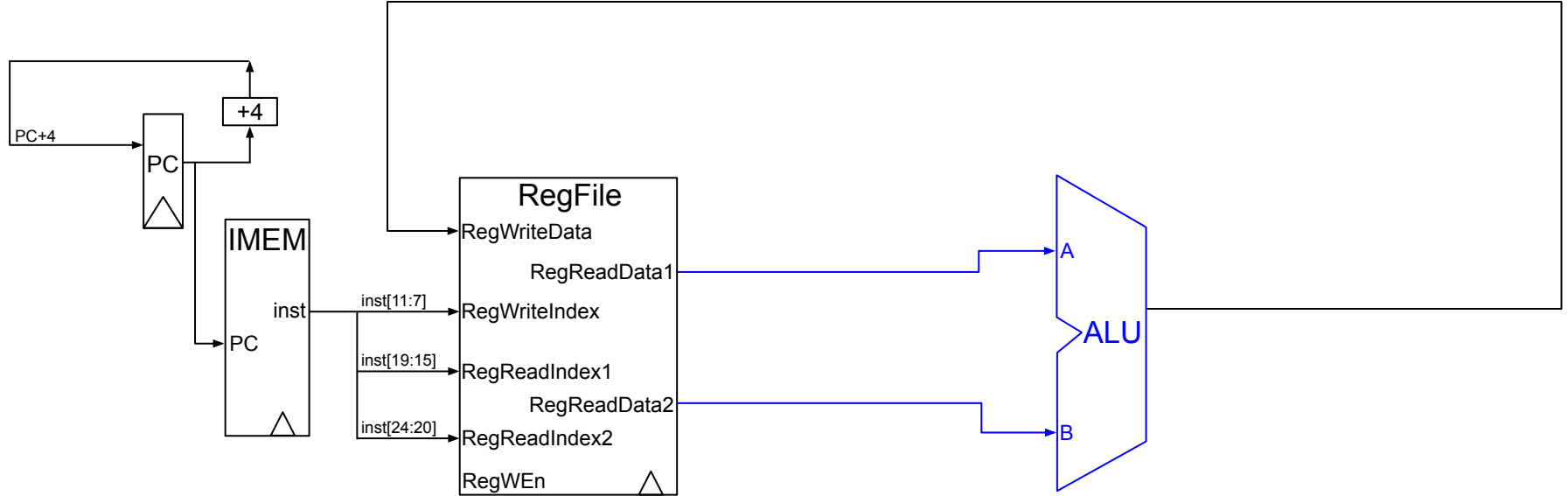
Same as the **add** instruction.
Extract fields from the 32-bit instruction.

Stage 2: Instruction Decode (ID), Register Read



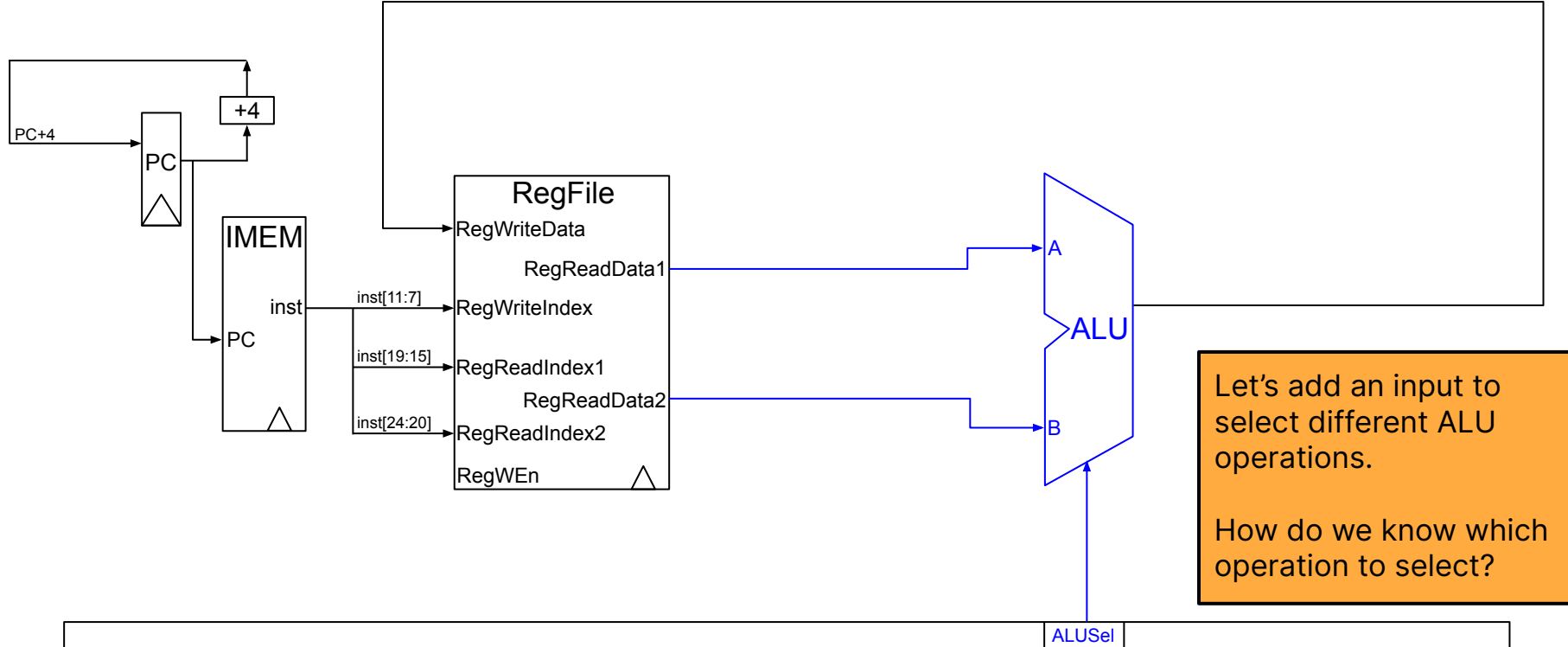
Same as the `add` instruction.
Read data from registers.

Stage 3: Execute

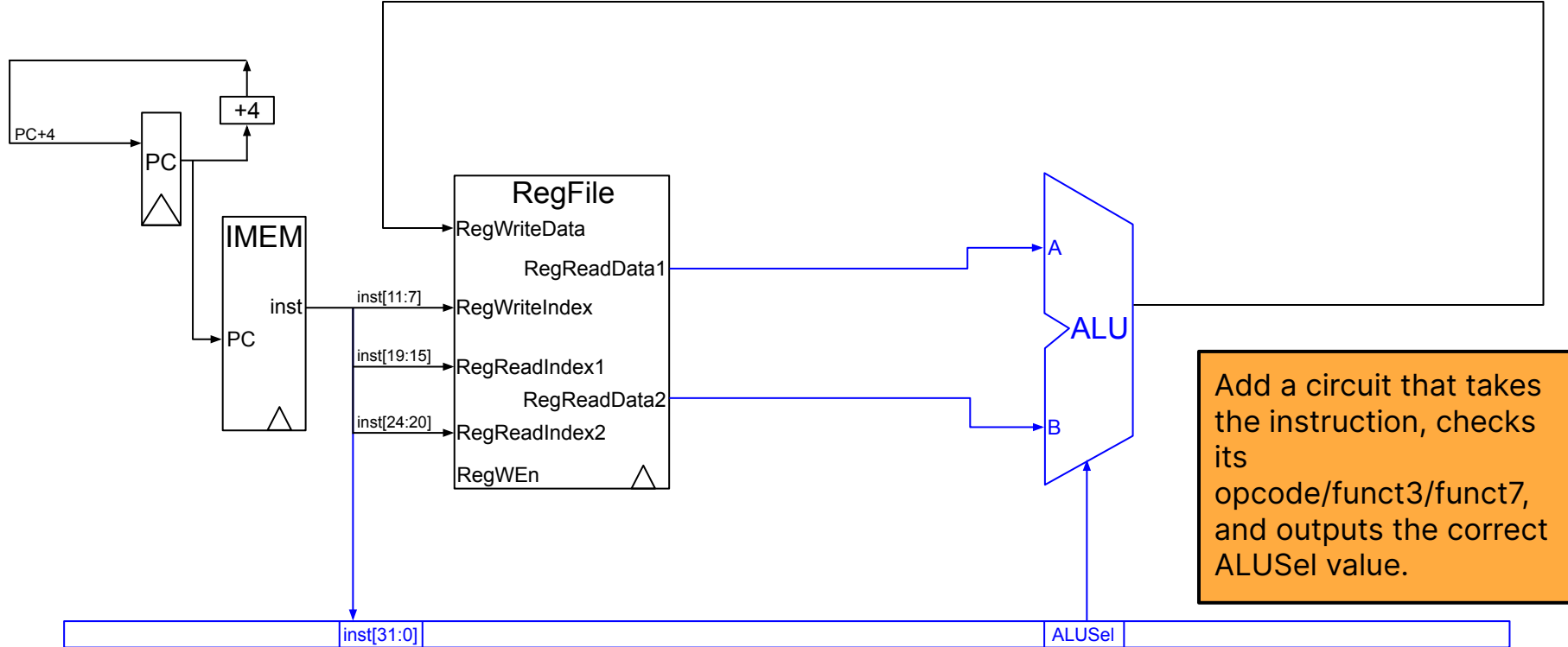


Different from the **add** instruction! We want to compute $A - B$, not $A + B$.

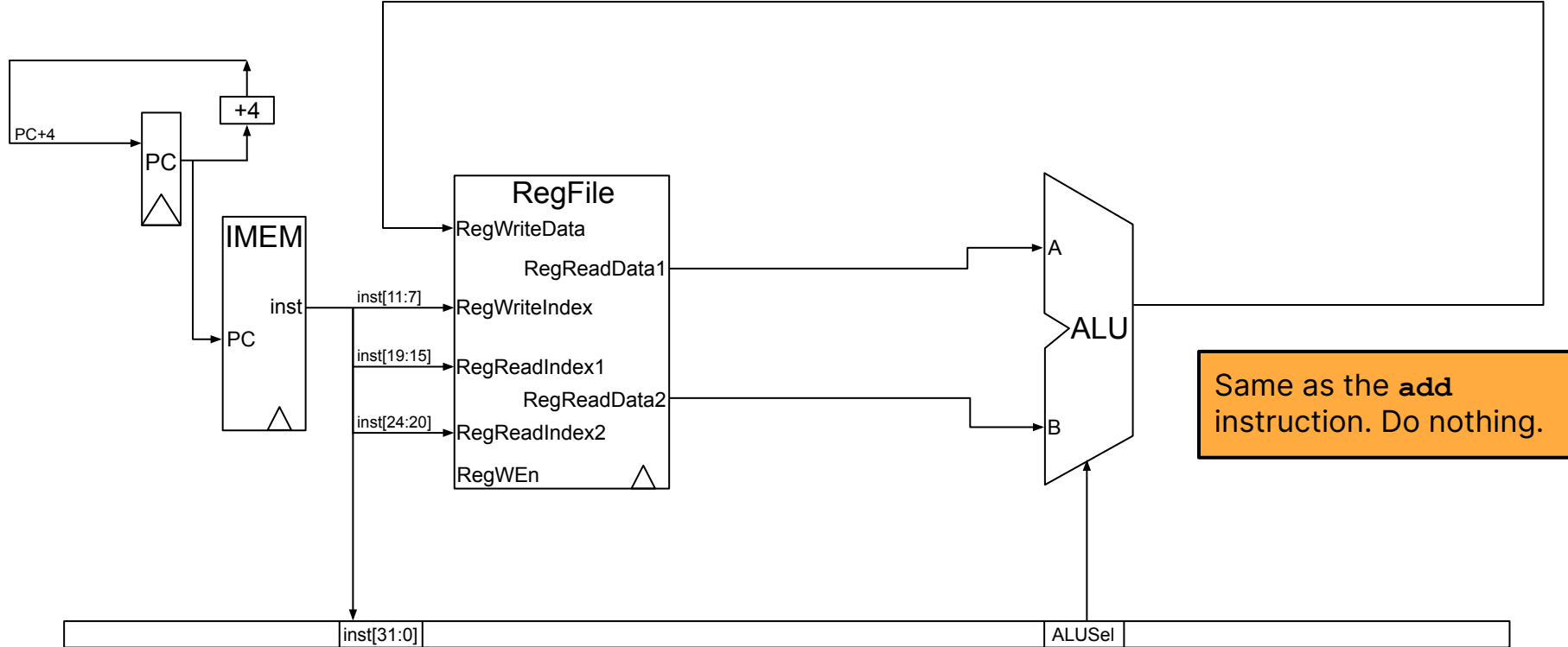
Stage 3: Execute



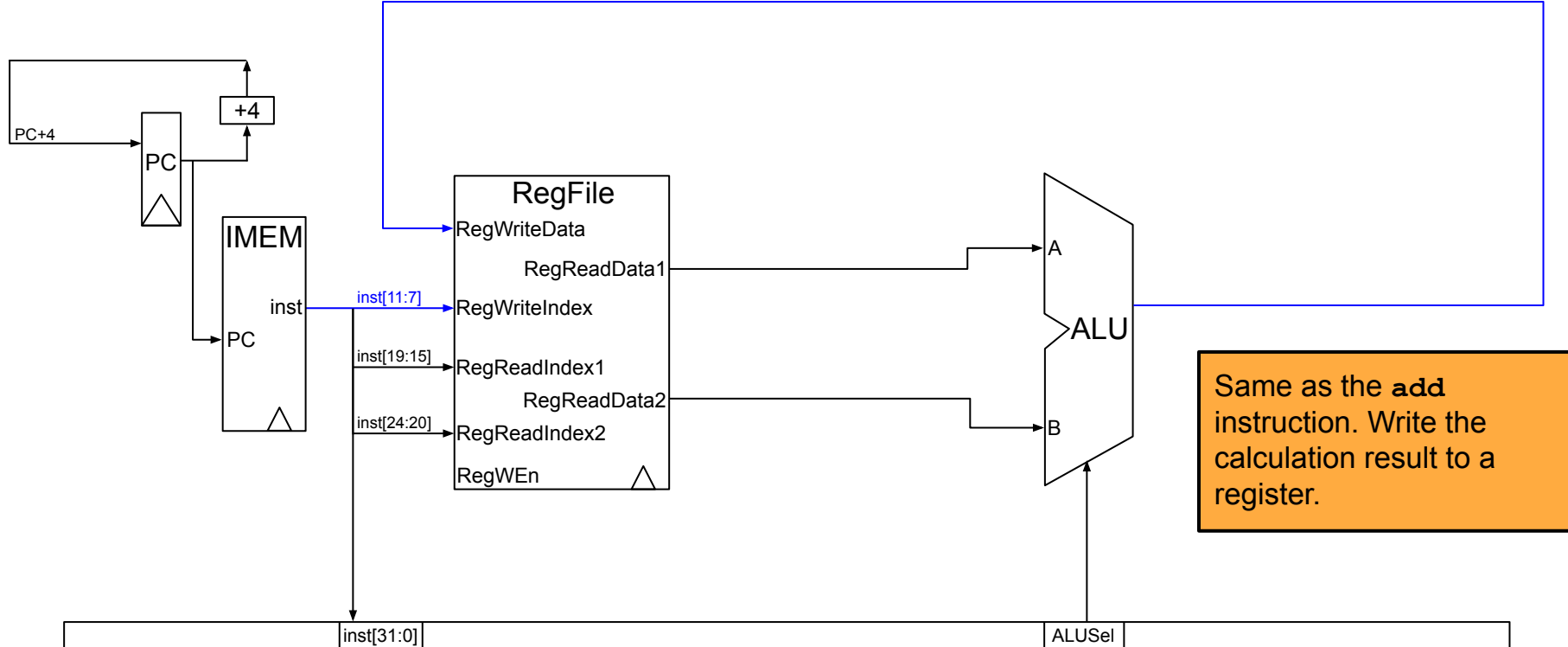
Stage 3: Execute



Stage 4: Memory



Stage 5: Register Write



New Subcircuit: Control Logic

- New subcircuit in our datapath: control logic
- Input: 32-bit instruction
- Output: Values that change how the datapath behaves
 - Example: ALUSel: What operation should the ALU perform?
 - Example: RegWEn: Should we write a value to the RegFile?
 - More control logic outputs as we see more instructions



Datapath for All R-type Instructions

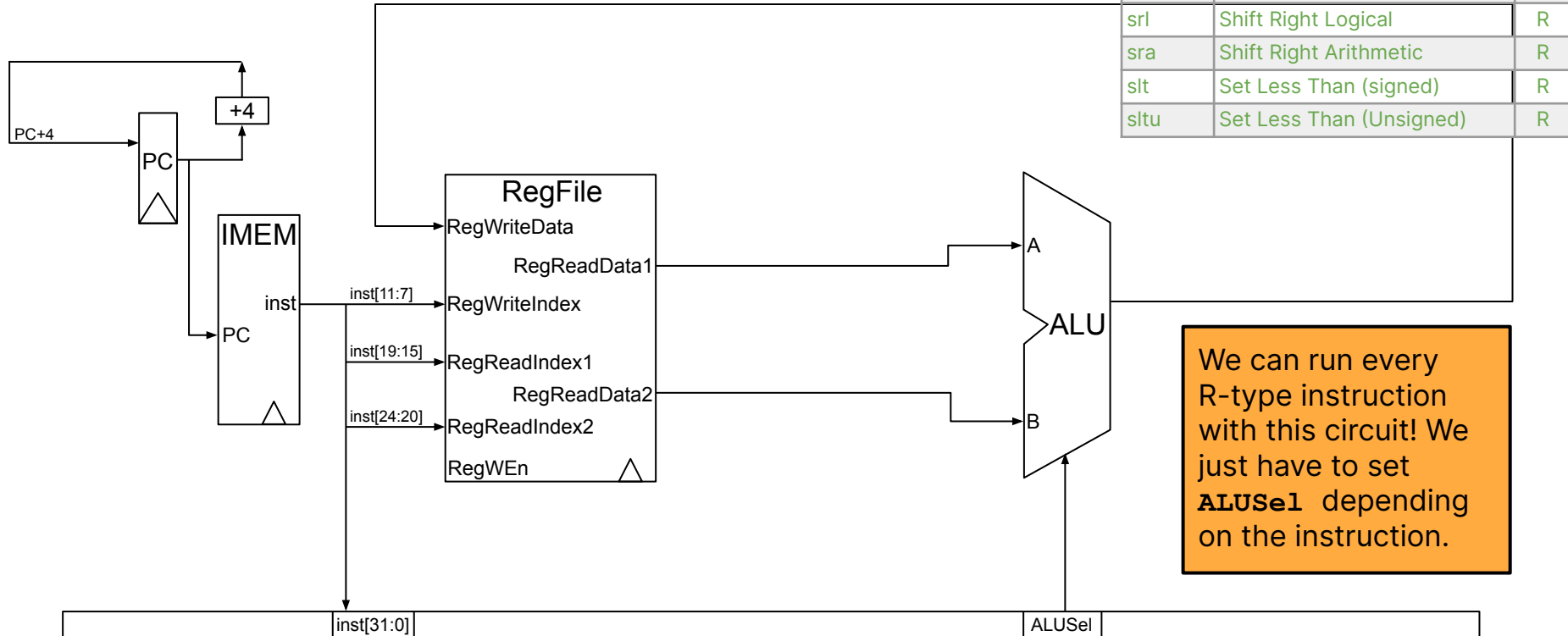
List of RISC-V Instructions

Instruction	Name	Type
add	ADD	R
sub	SUBtract	R
and	bitwise AND	R
or	bitwise OR	R
xor	bitwise XOR	R
sll	Shift Left Logical	R
srl	Shift Right Logical	R
sra	Shift Right Arithmetic	R
slt	Set Less Than (signed)	R
sltu	Set Less Than (Unsigned)	R
addi	ADD Immediate	I
andi	bitwise AND Immediate	I
ori	bitwise OR Immediate	I
xori	bitwise XOR Immediate	I

Instruction	Name	Type
slli	Shift Left Logical Immediate	I*
srli	Shift Right Logical Immediate	I*
srai	Shift Right Arithmetic Immediate	I*
slti	Set Less Than Immediate (signed)	I
sltiu	Set Less Than Immediate (Unsigned)	I
lb	Load Byte	I
lbu	Load Byte (Unsigned)	I
lh	Load Half-word	I
lhu	Load Half-word (Unsigned)	I
lw	Load Word	I

Instruction	Name	Type
sb	Store Byte	S
sh	Store Half-word	S
sw	Store Word	S
beq	Branch if Equal	B
bge	Branch if Greater or Equal (signed)	B
bgeu	Branch if Greater or Equal (Unsigned)	B
blt	Branch if Less Than (signed)	B
bltu	Branch if Less Than (Unsigned)	B
bne	Branch if Not Equal	B
jal	Jump And Link	J
jalr	Jump And Link Register	I
auipc	Add Upper Immediate to PC	U
lui	Load Upper Immediate	U

R-Type Instruction Datapath



Datapath for **addi**

List of RISC-V Instructions

Instruction	Name	Type
add	ADD	R
sub	SUBtract	R
and	bitwise AND	R
or	bitwise OR	R
xor	bitwise XOR	R
sll	Shift Left Logical	R
srl	Shift Right Logical	R
sra	Shift Right Arithmetic	R
slt	Set Less Than (signed)	R
sltu	Set Less Than (Unsigned)	R
addi	ADD Immediate	I
andi	bitwise AND Immediate	I
ori	bitwise OR Immediate	I
xori	bitwise XOR Immediate	I

Instruction	Name	Type
slli	Shift Left Logical Immediate	I*
srli	Shift Right Logical Immediate	I*
srai	Shift Right Arithmetic Immediate	I*
slti	Set Less Than Immediate (signed)	I
sltiu	Set Less Than Immediate (Unsigned)	I
lb	Load Byte	I
lbu	Load Byte (Unsigned)	I
lh	Load Half-word	I
lhu	Load Half-word (Unsigned)	I
lw	Load Word	I

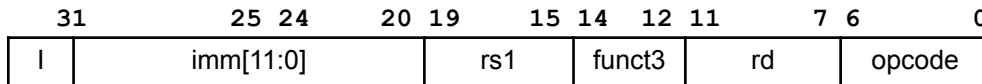
Instruction	Name	Type
sb	Store Byte	S
sh	Store Half-word	S
sw	Store Word	S
beq	Branch if Equal	B
bge	Branch if Greater or Equal (signed)	B
bgeu	Branch if Greater or Equal (Unsigned)	B
blt	Branch if Less Than (signed)	B
bltu	Branch if Less Than (Unsigned)	B
bne	Branch if Not Equal	B
jal	Jump And Link	J
jalr	Jump And Link Register	I
auipc	Add Upper Immediate to PC	U
lui	Load Upper Immediate	U

addi instruction

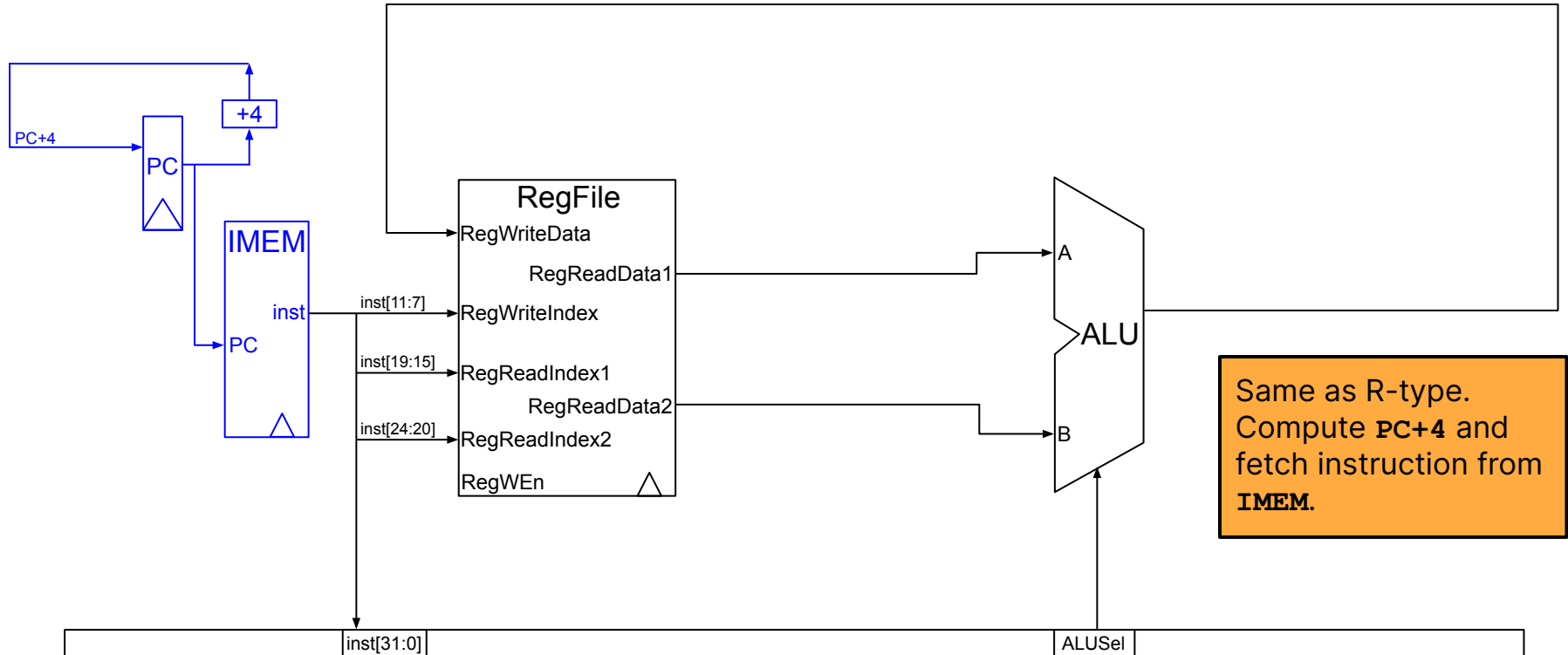
addi rd rs1 imm	ADD Immediate	rd = rs1 + imm
-----------------	---------------	----------------

- What's the **add** workflow?

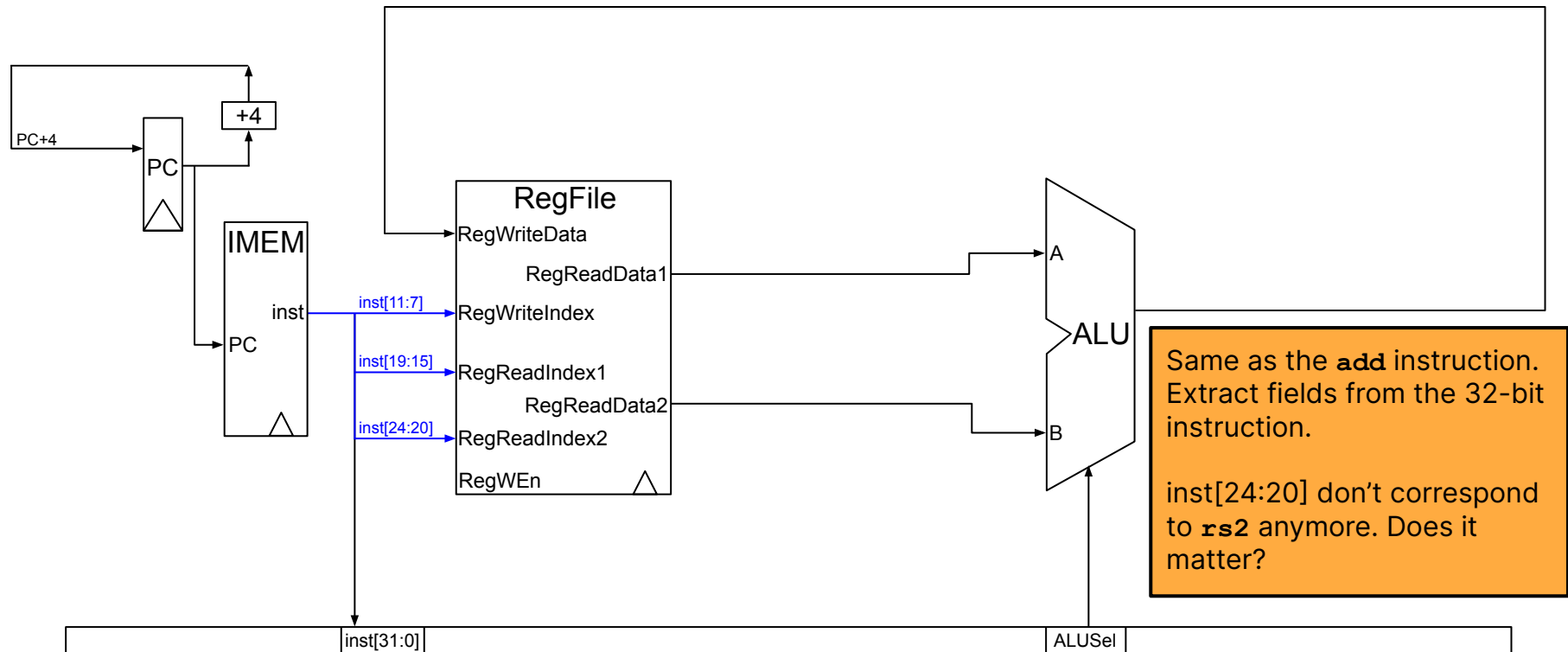
- **rs1 + imm**: Add the values in register rs1 to the immediate
- **rd = rs1 + imm**: Store the result in register **rd**
- **PC = PC + 4**: Increment the program counter



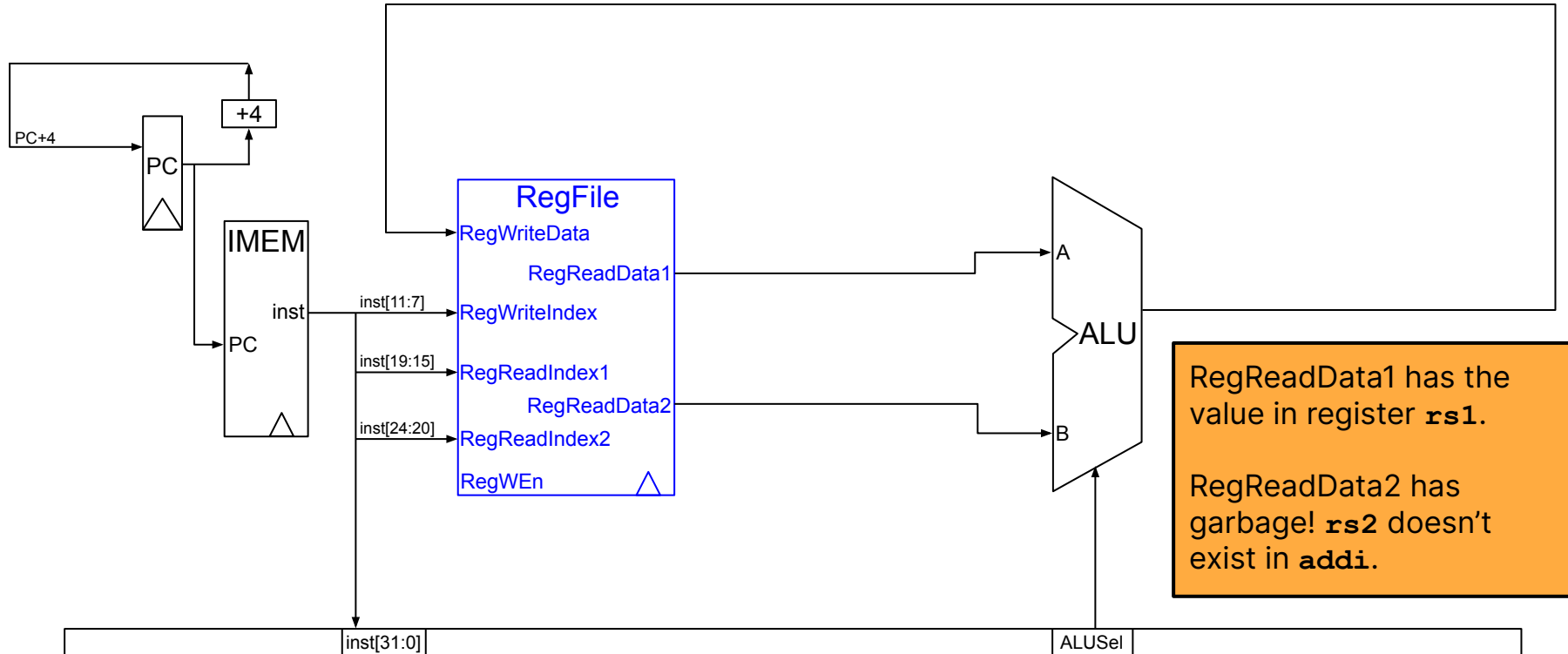
Stage 1: Instruction Fetch (IF)



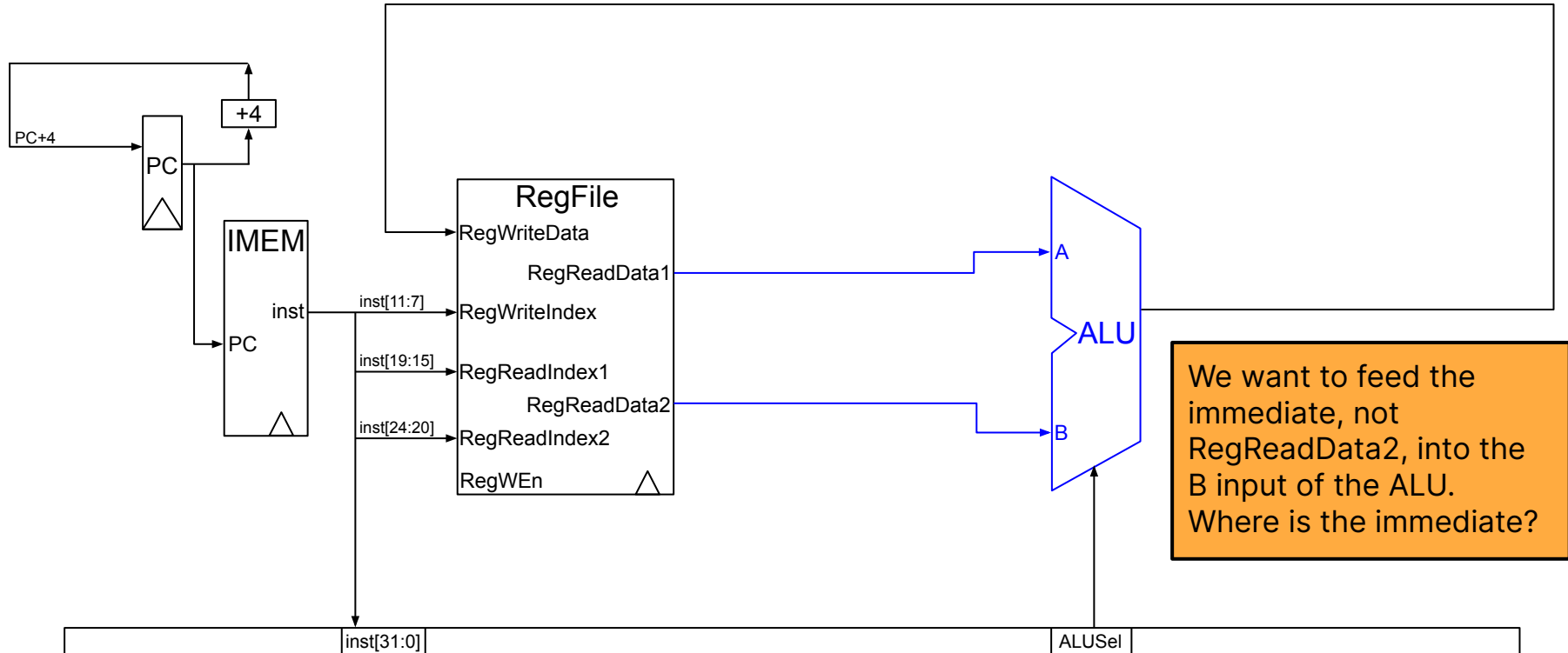
Stage 2: Instruction Decode (ID)



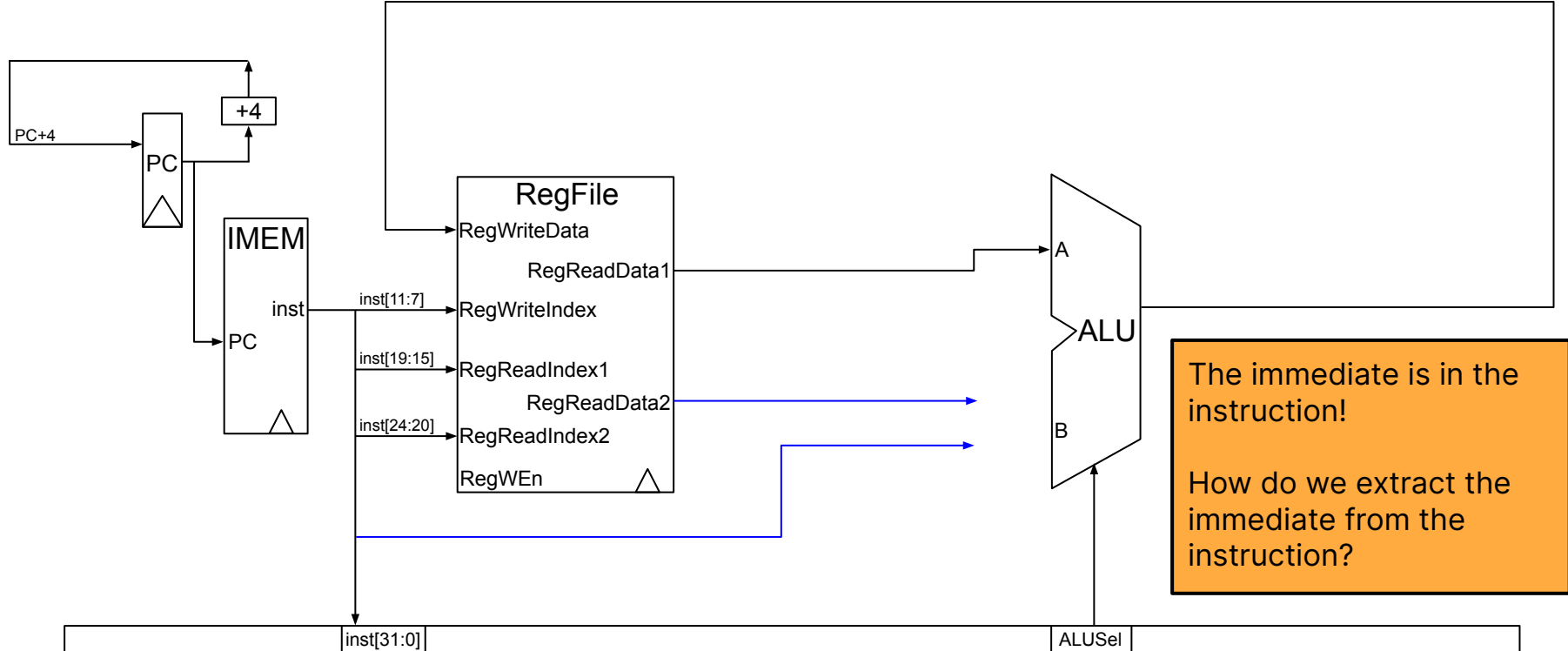
Stage 2: Instruction Decode (ID), Register Read



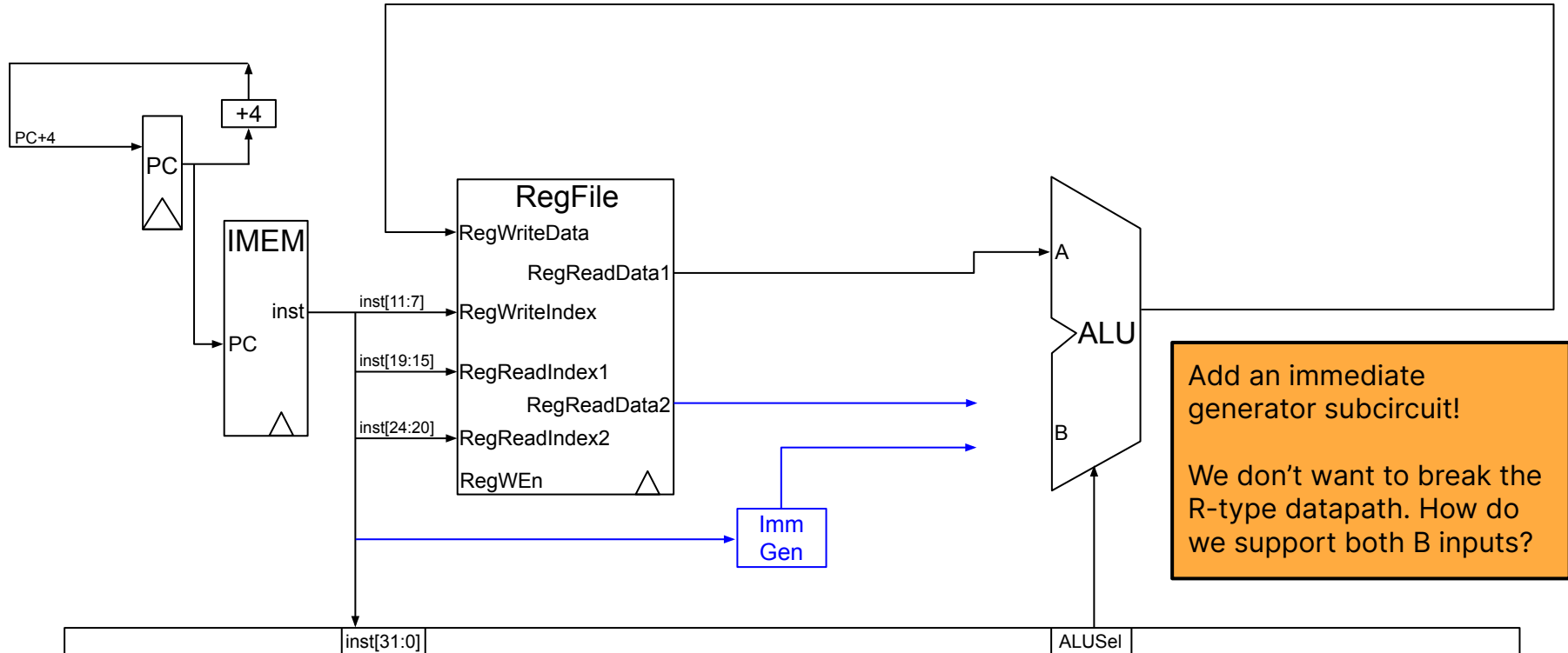
Stage 3: Execute



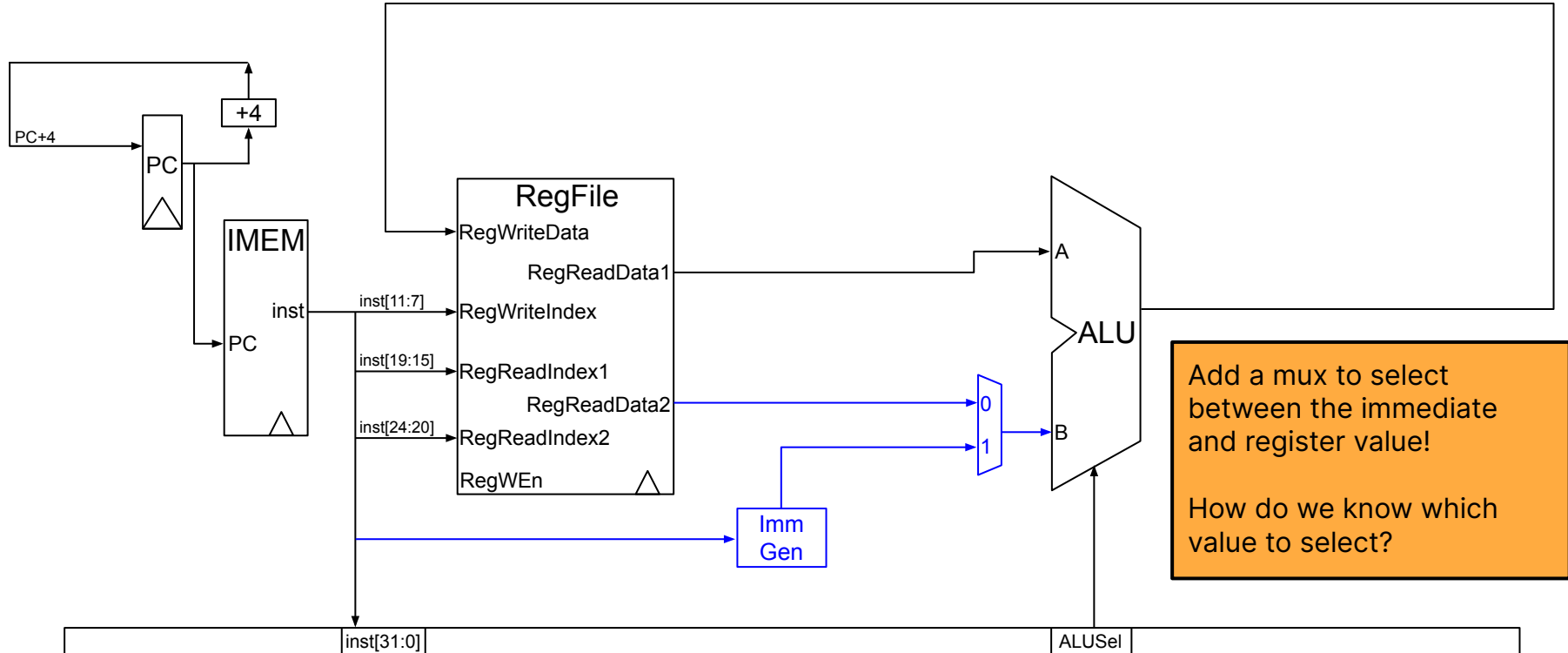
Stage 3: Execute



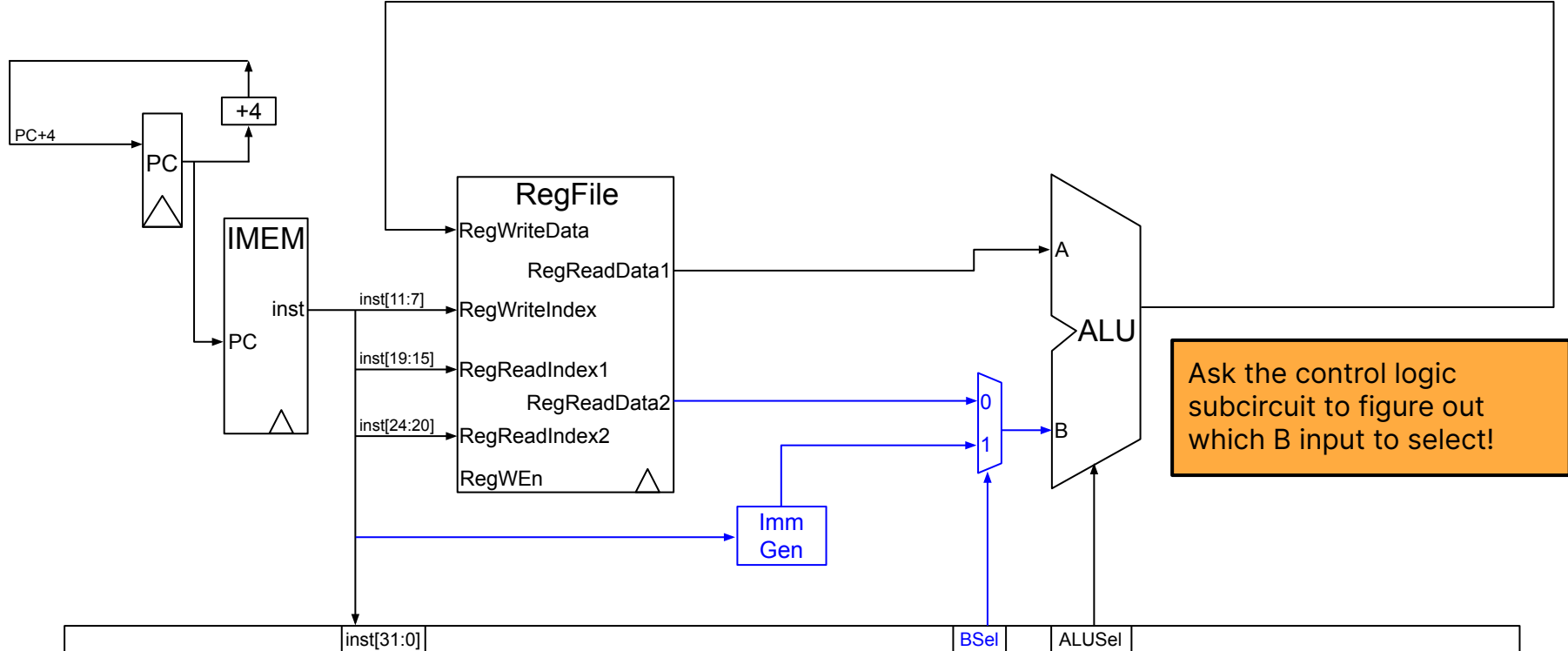
Stage 3: Execute



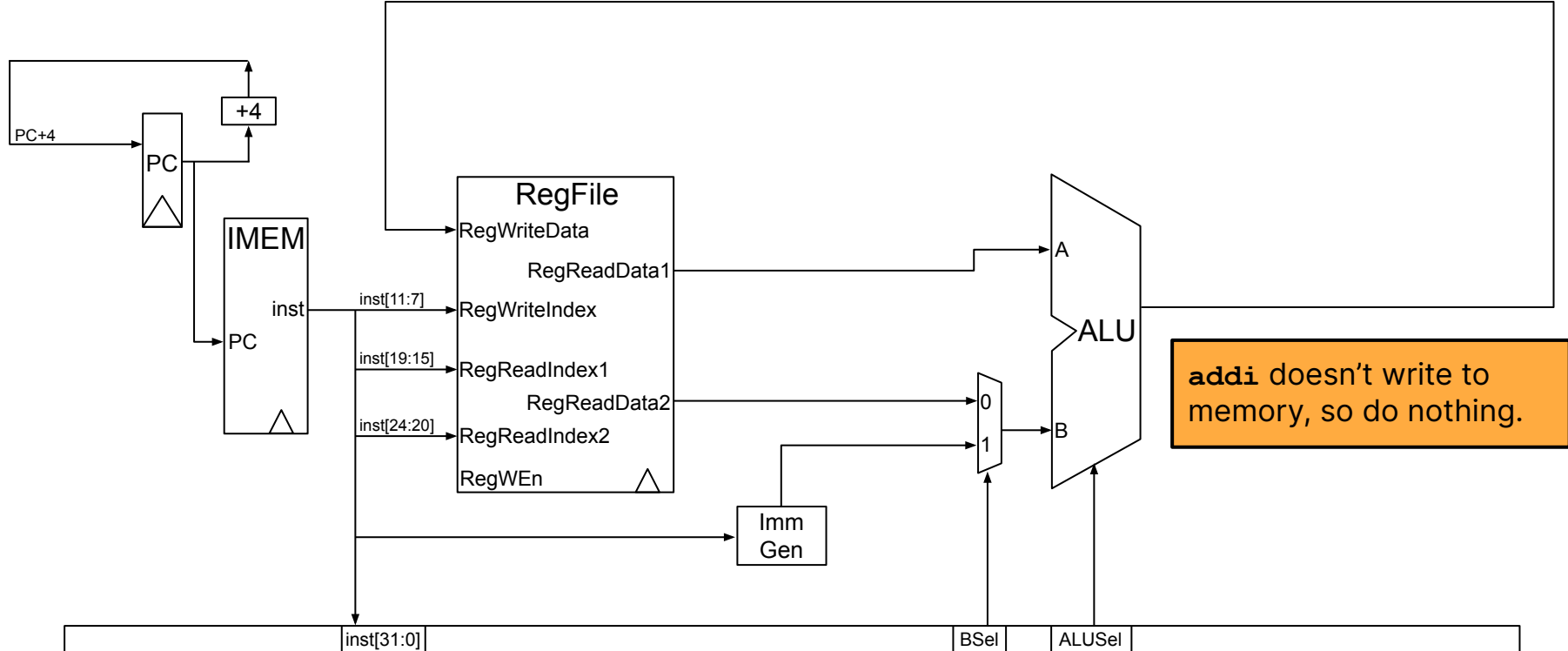
Stage 3: Execute



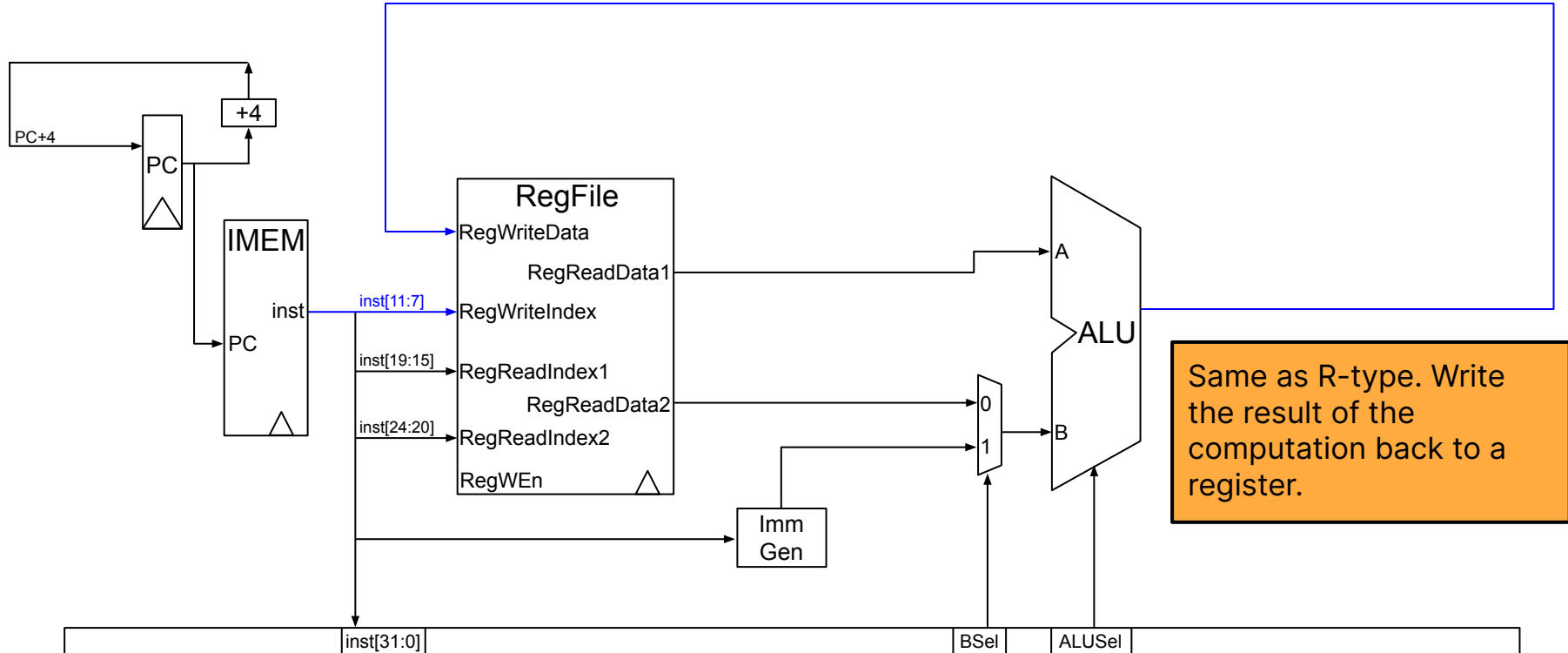
Stage 3: Execute



Stage 4: Memory



Stage 5: Register Write

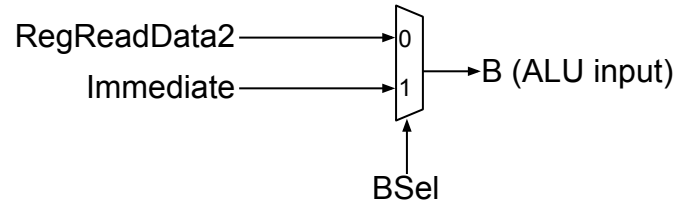


New Subcircuit: Immediate Generator

- New subcircuit in our datapath: immediate generator
- Input: 32-bit instruction
 - There's another input, ImmSel, which we'll see later
- Output: 32-bit immediate value
 - Extract the immediate bits from the instruction
 - Extend the immediate to 32 bits (usually sign-extend)

New Control Signal: BSel

- Chooses which value to send to the B input of the ALU
 - BSel=0: Send data in register `rs2` to ALU
 - BSel=1: Send immediate to ALU
- Control logic subcircuit decodes instruction and outputs appropriate BSel



Datapath for I-type Instructions

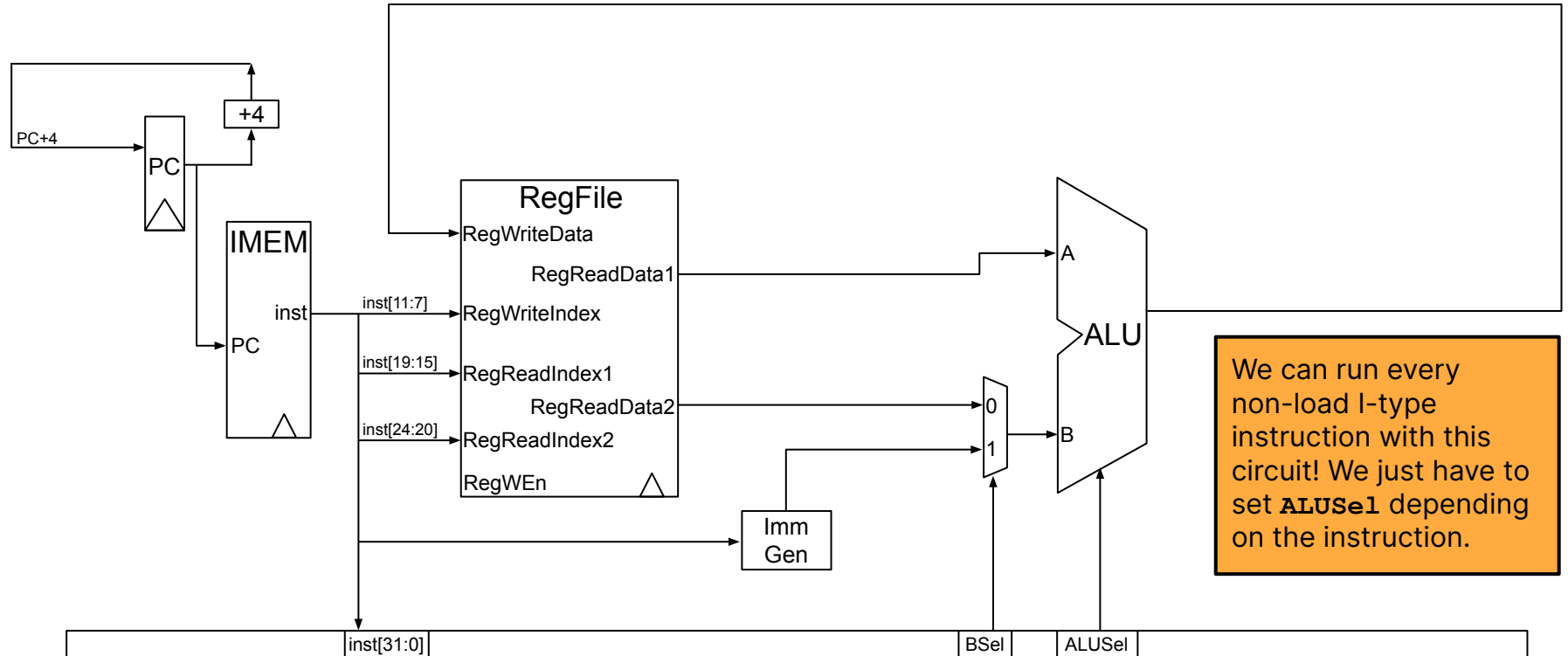
List of RISC-V Instructions

Instruction	Name	Type
add	ADD	R
sub	SUBtract	R
and	bitwise AND	R
or	bitwise OR	R
xor	bitwise XOR	R
sll	Shift Left Logical	R
srl	Shift Right Logical	R
sra	Shift Right Arithmetic	R
slt	Set Less Than (signed)	R
sltu	Set Less Than (Unsigned)	R
addi	ADD Immediate	I
andi	bitwise AND Immediate	I
ori	bitwise OR Immediate	I
xori	bitwise XOR Immediate	I

Instruction	Name	Type
slli	Shift Left Logical Immediate	I*
srli	Shift Right Logical Immediate	I*
srai	Shift Right Arithmetic Immediate	I*
slti	Set Less Than Immediate (signed)	I
sltiu	Set Less Than Immediate (Unsigned)	I
lb	Load Byte	I
lbu	Load Byte (Unsigned)	I
lh	Load Half-word	I
lhu	Load Half-word (Unsigned)	I
lw	Load Word	I

Instruction	Name	Type
sb	Store Byte	S
sh	Store Half-word	S
sw	Store Word	S
beq	Branch if Equal	B
bge	Branch if Greater or Equal (signed)	B
bgeu	Branch if Greater or Equal (Unsigned)	B
blt	Branch if Less Than (signed)	B
bltu	Branch if Less Than (Unsigned)	B
bne	Branch if Not Equal	B
jal	Jump And Link	J
jalr	Jump And Link Register	I
auipc	Add Upper Immediate to PC	U
lui	Load Upper Immediate	U

R-Type and I-Type Instruction Datapath



Datapath for Loads

List of RISC-V Instructions

Instruction	Name	Type
add	ADD	R
sub	SUBtract	R
and	bitwise AND	R
or	bitwise OR	R
xor	bitwise XOR	R
sll	Shift Left Logical	R
srl	Shift Right Logical	R
sra	Shift Right Arithmetic	R
slt	Set Less Than (signed)	R
sltu	Set Less Than (Unsigned)	R
addi	ADD Immediate	I
andi	bitwise AND Immediate	I
ori	bitwise OR Immediate	I
xori	bitwise XOR Immediate	I

Instruction	Name	Type
slli	Shift Left Logical Immediate	I*
srli	Shift Right Logical Immediate	I*
srai	Shift Right Arithmetic Immediate	I*
slti	Set Less Than Immediate (signed)	I
sltiu	Set Less Than Immediate (Unsigned)	I
lb	Load Byte	I
lbu	Load Byte (Unsigned)	I
lh	Load Half-word	I
lhu	Load Half-word (Unsigned)	I
lw	Load Word	I

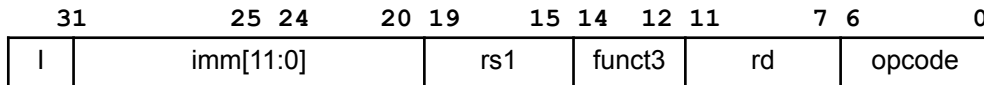
Instruction	Name	Type
sb	Store Byte	S
sh	Store Half-word	S
sw	Store Word	S
beq	Branch if Equal	B
bge	Branch if Greater or Equal (signed)	B
bgeu	Branch if Greater or Equal (Unsigned)	B
blt	Branch if Less Than (signed)	B
bltu	Branch if Less Than (Unsigned)	B
bne	Branch if Not Equal	B
jal	Jump And Link	J
jalr	Jump And Link Register	I
auipc	Add Upper Immediate to PC	U
lui	Load Upper Immediate	U

Load instructions

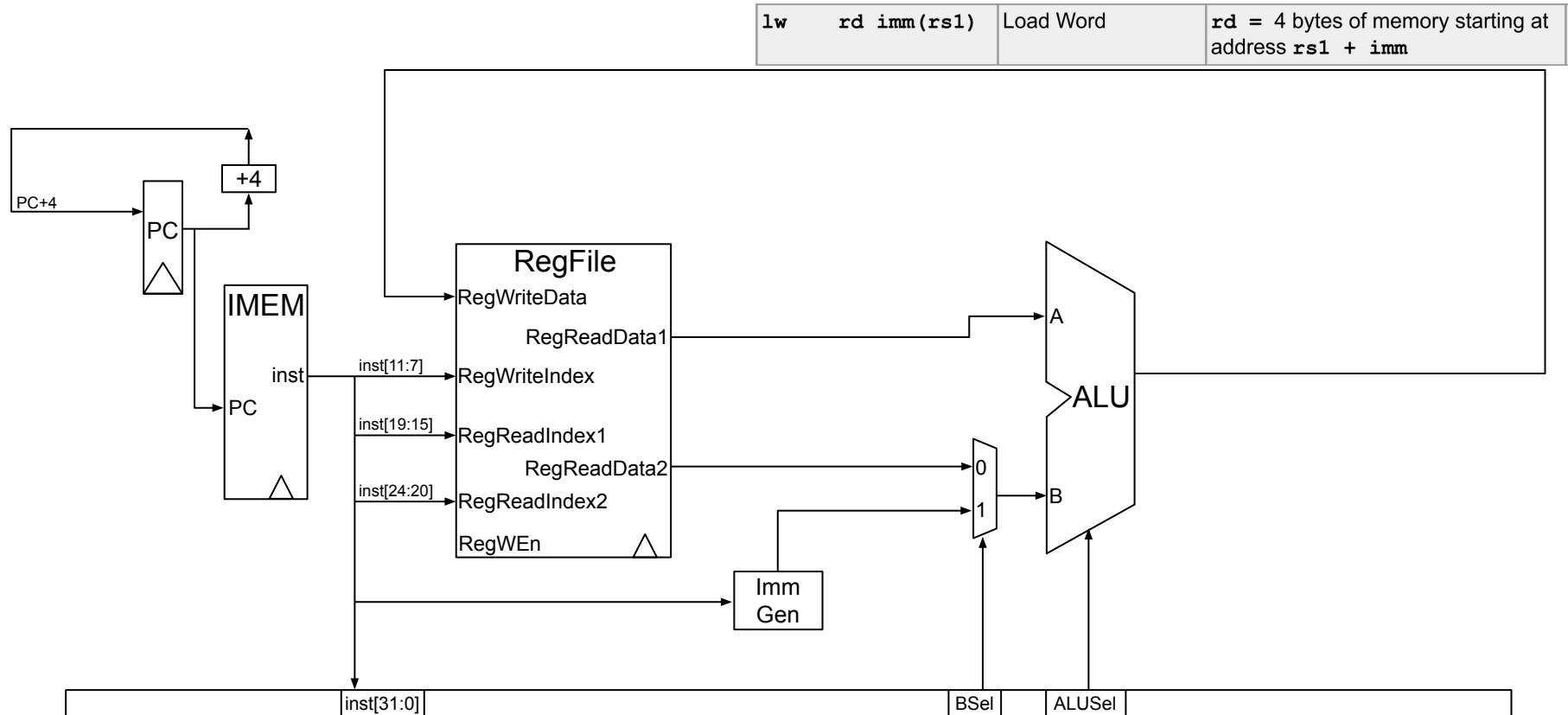
lw	rd imm(rs1)	Load Word	rd = 4 bytes of memory starting at address rs1 + imm
-----------	--------------------	-----------	--

- What is the **lw** workflow?

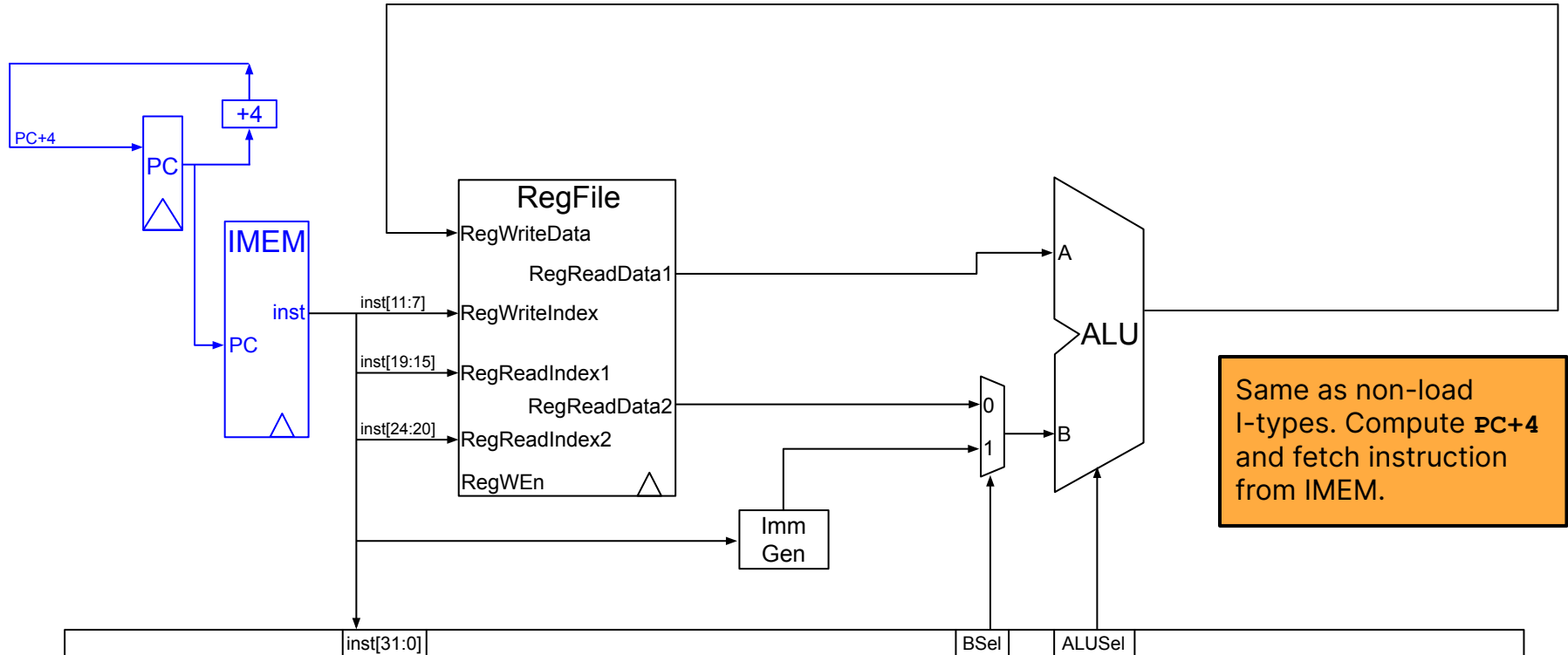
- Basics: read instruction, parse instruction, etc.
- **imm + rs1**: Add the value in register **rs1** to the immediate in the instruction (just like non-load l-types)
- **Read Mem[imm + rs1]**: The result of the addition is the memory address to load from. Load data from this memory address.
- **rd = Mem[imm + rs1]**: write the data to a register
- **PC = PC + 4**: Increment the program counter



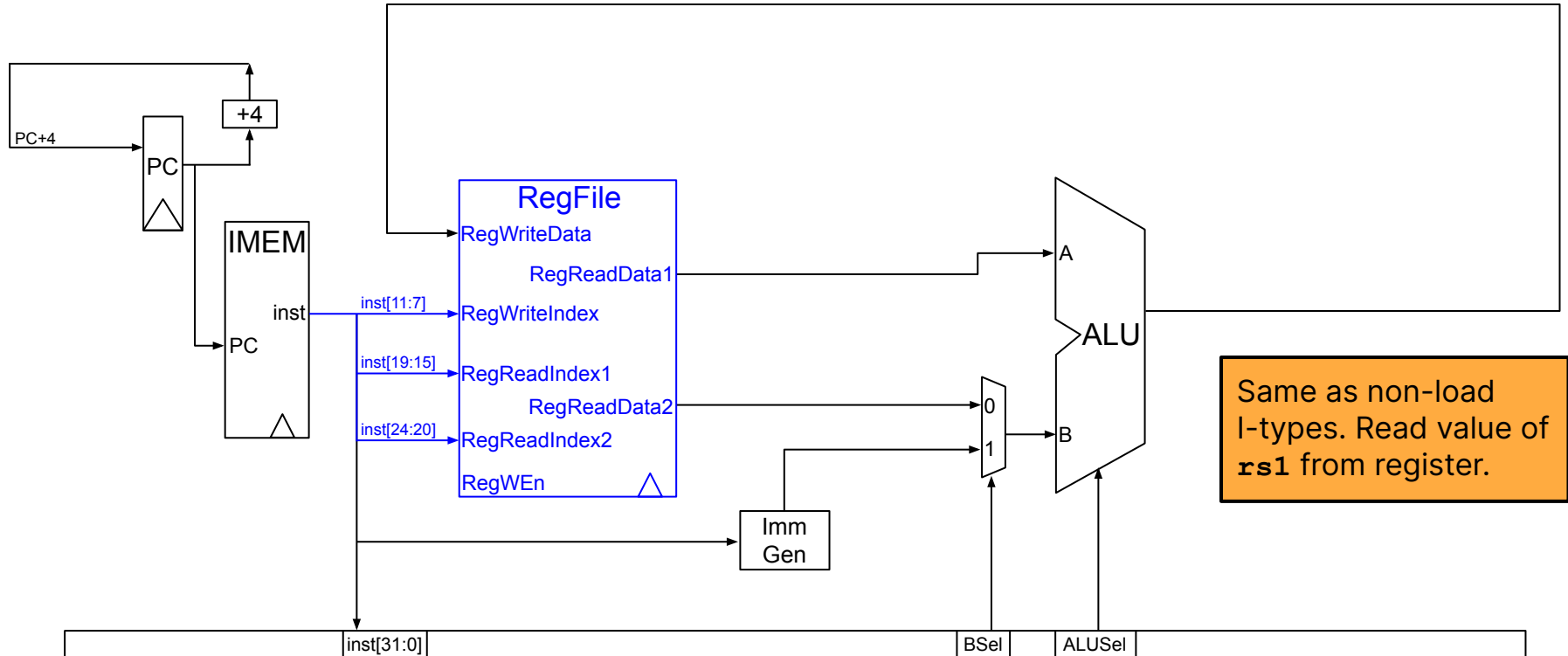
Discussion: What changes?



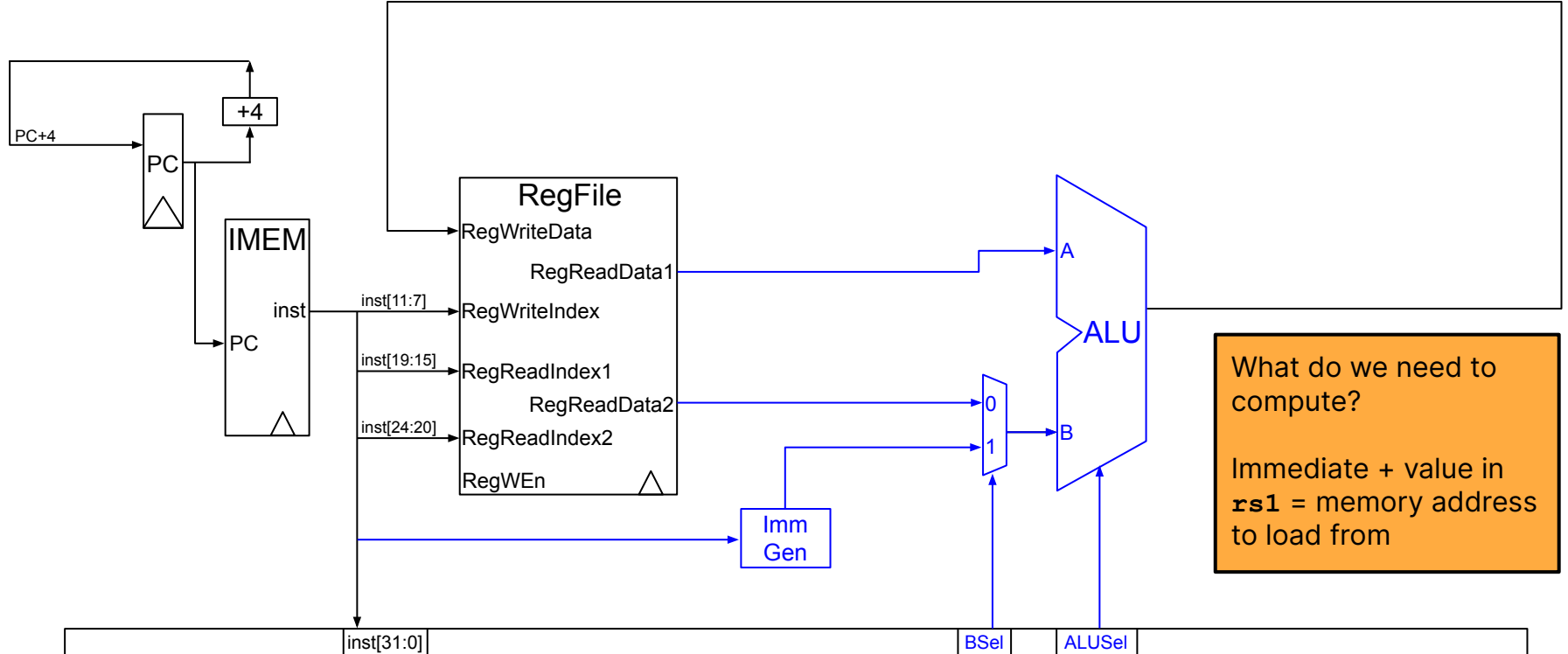
Load Stage 1: Instruction Fetch (IF)



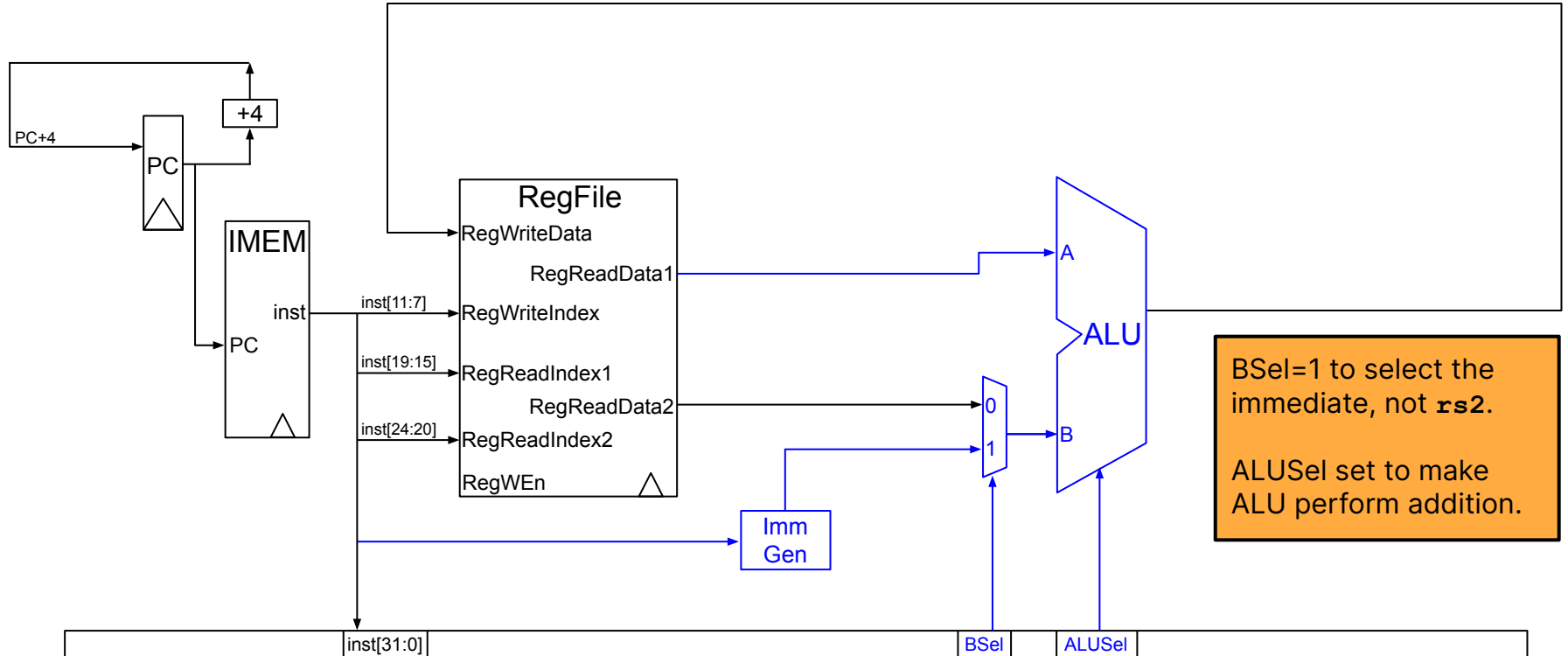
Load Stage 2: Instruction Decode (ID), Register Read



Load Stage 3: Execute

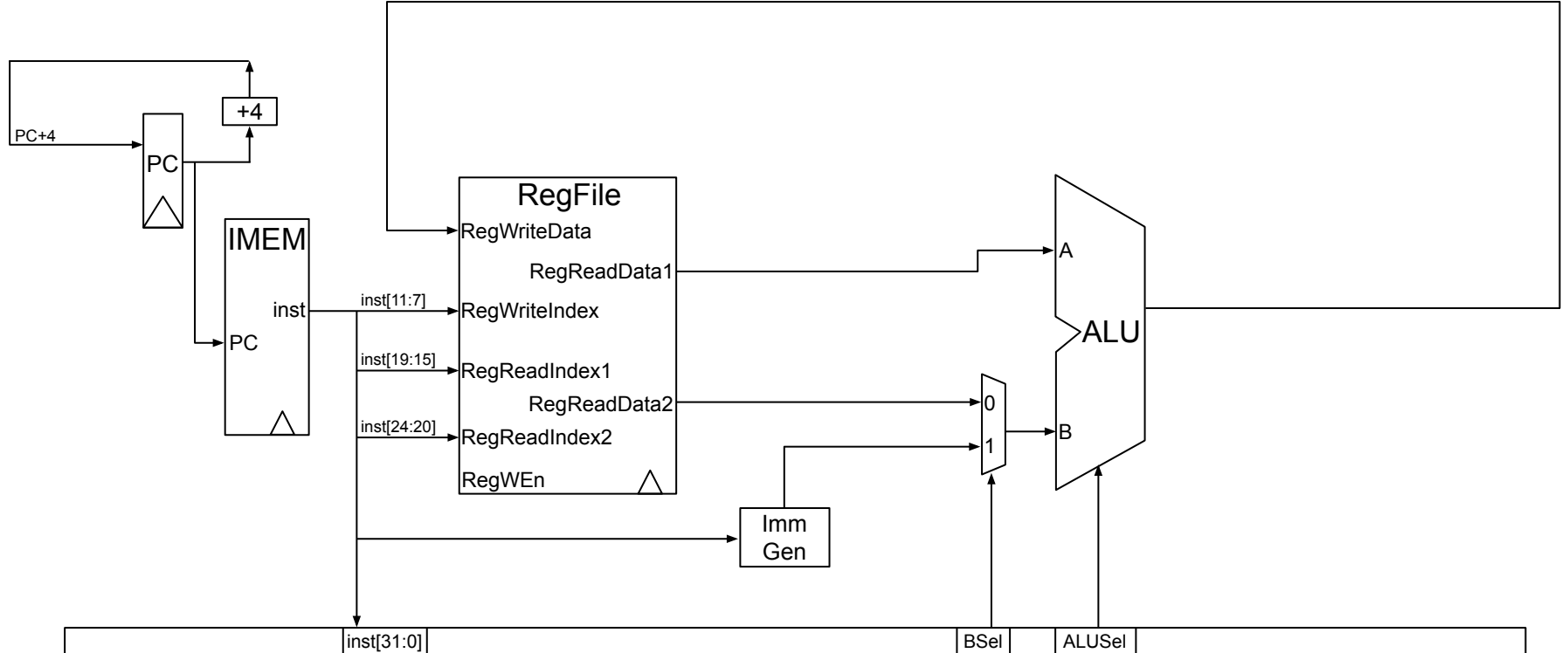


Load Stage 3: Execute



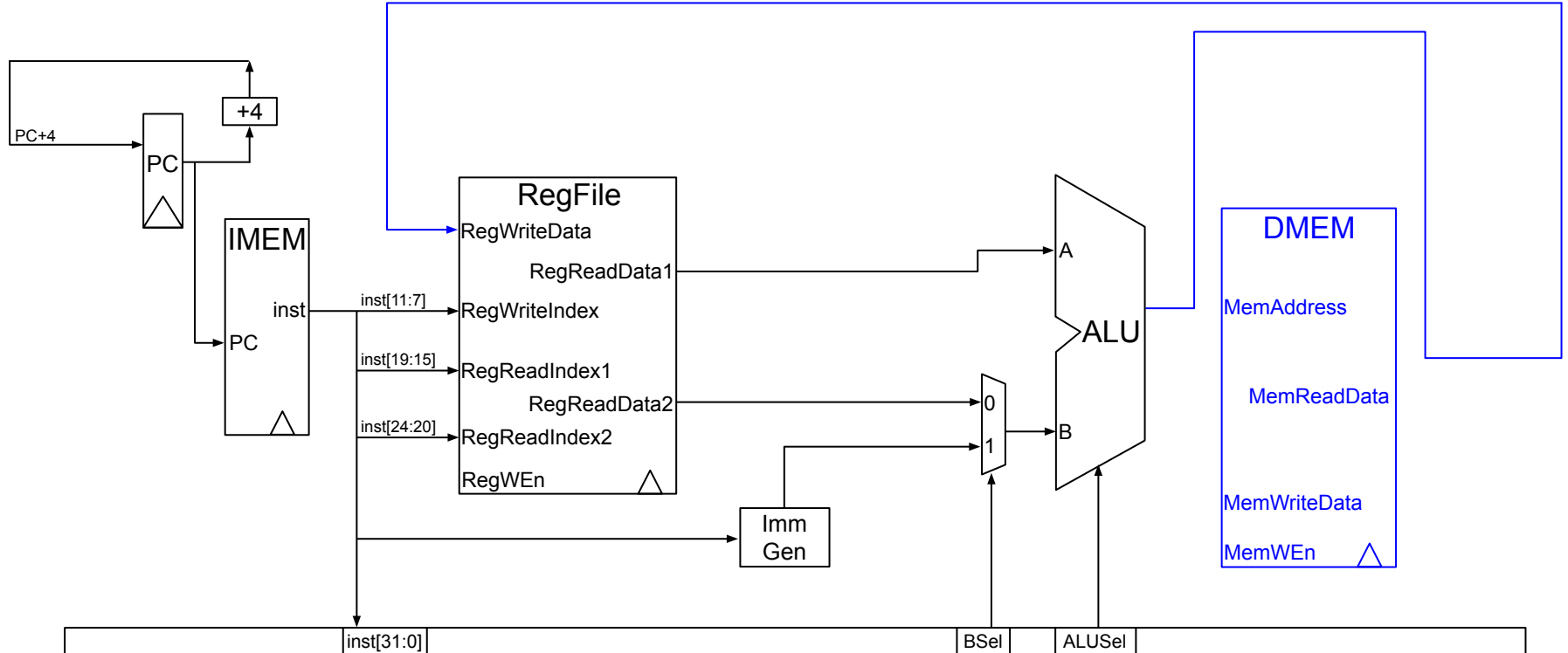
We need a memory component!

Load Stage 4: Memory



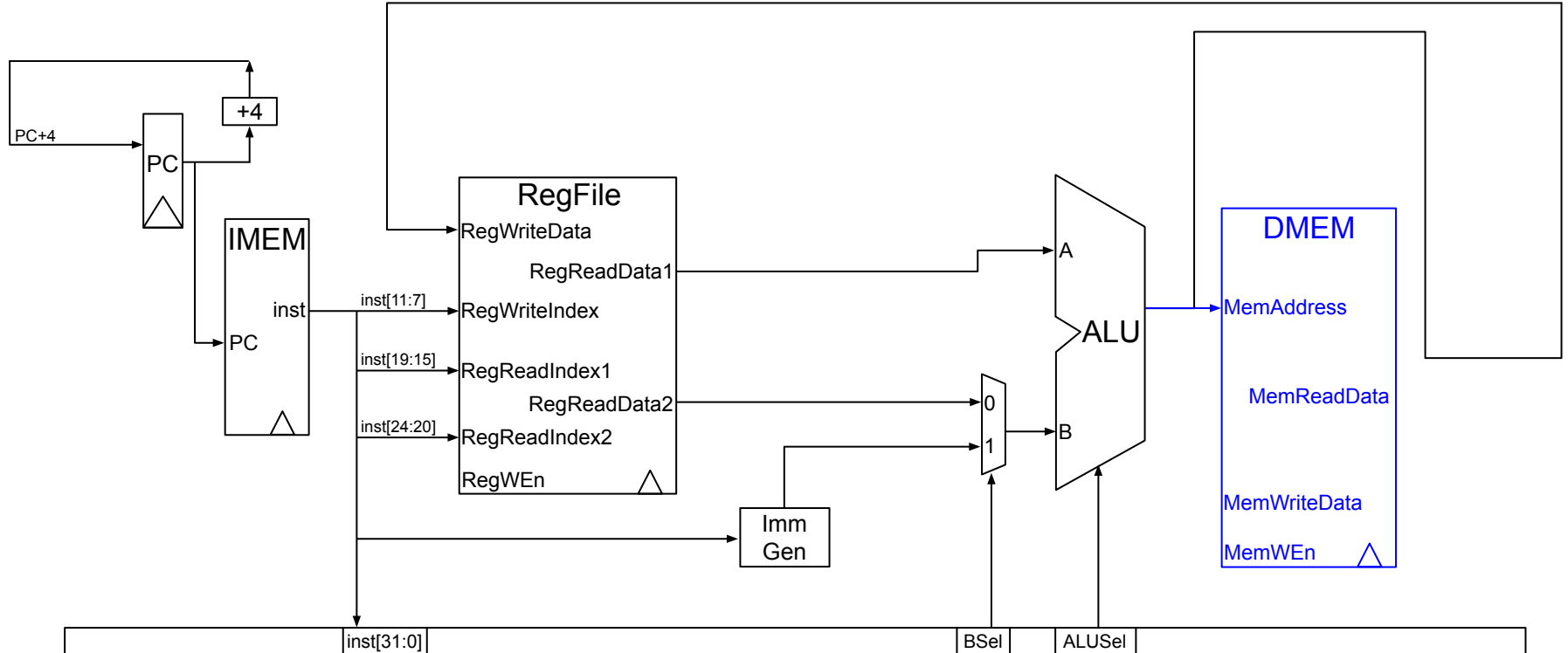
Load Stage 4: Memory

Here's a memory block. What's the address we want to load from?



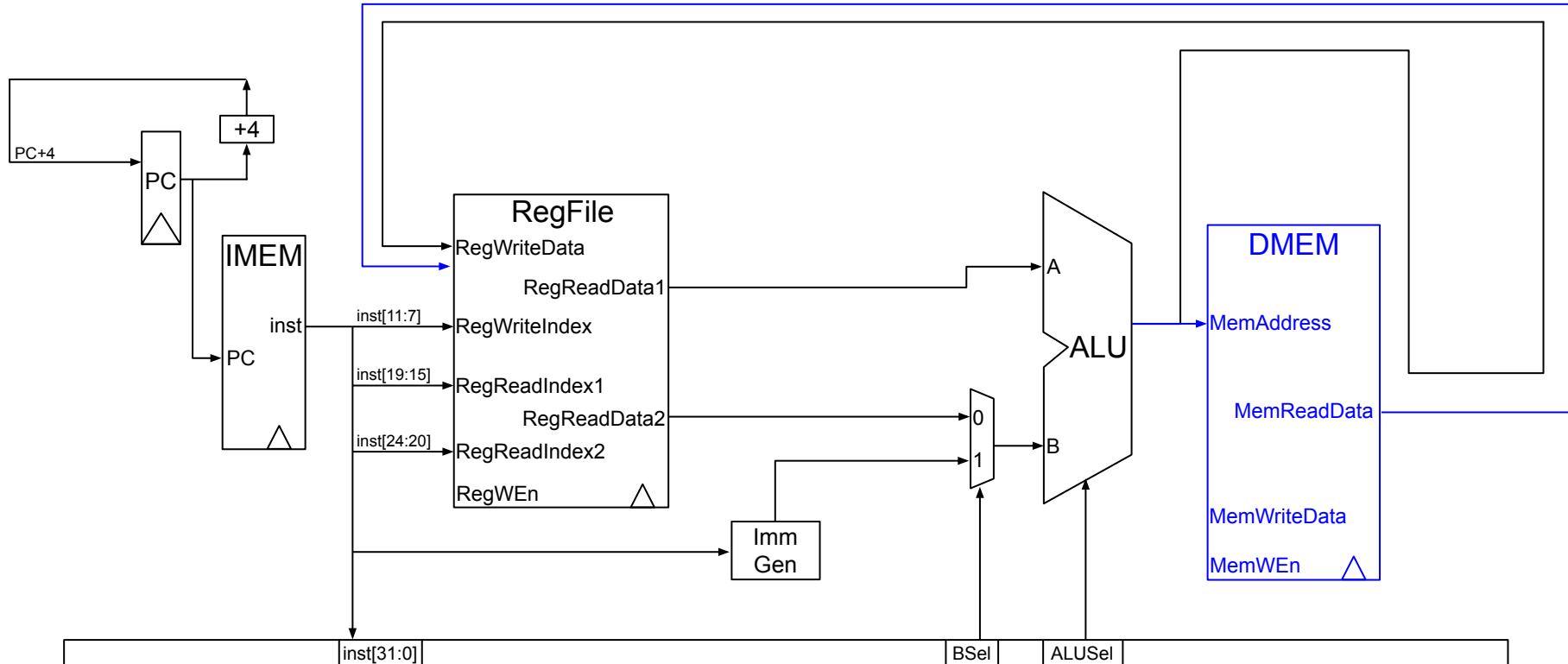
Load Stage 4: Memory

We computed the address with the ALU!
Where does the data we read go?



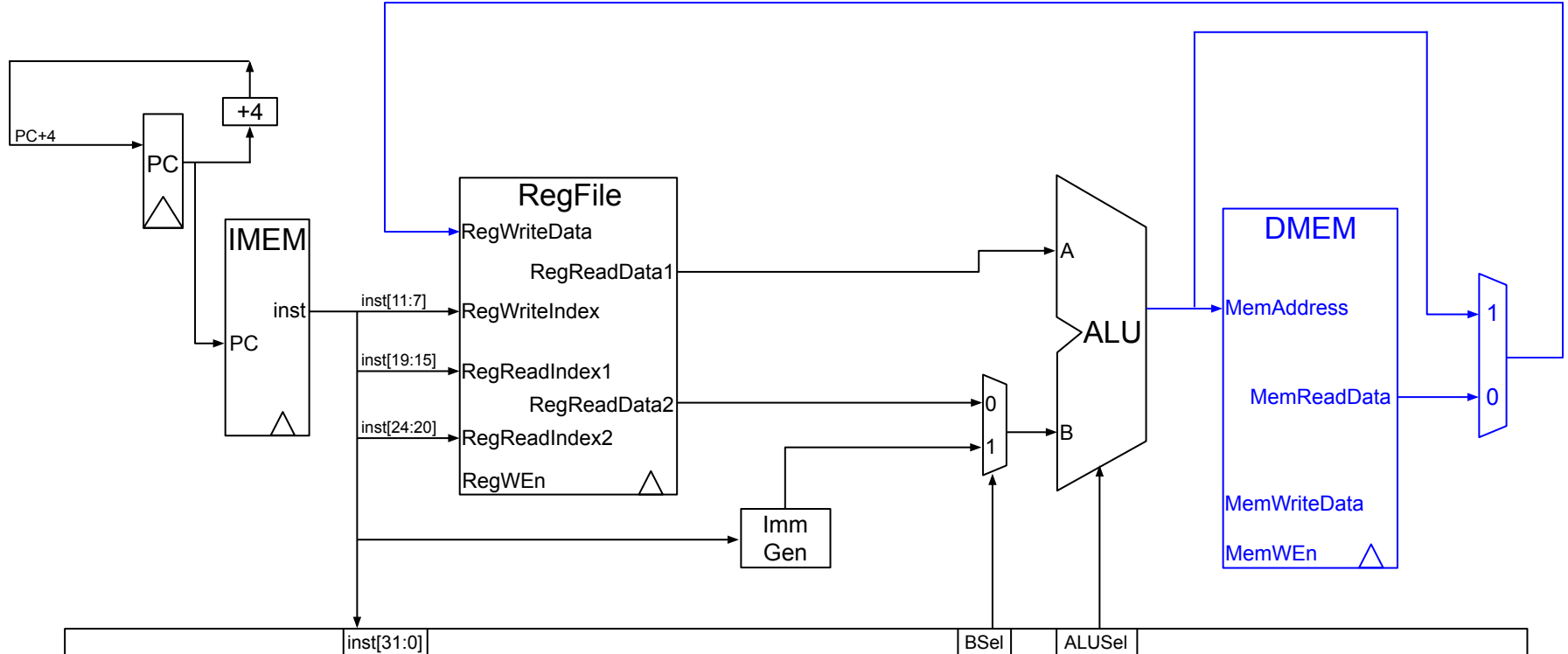
Load Stage 5: Register Write

We want to write the data from memory to a register. How do we support writing both data from memory and ALU output to a register?



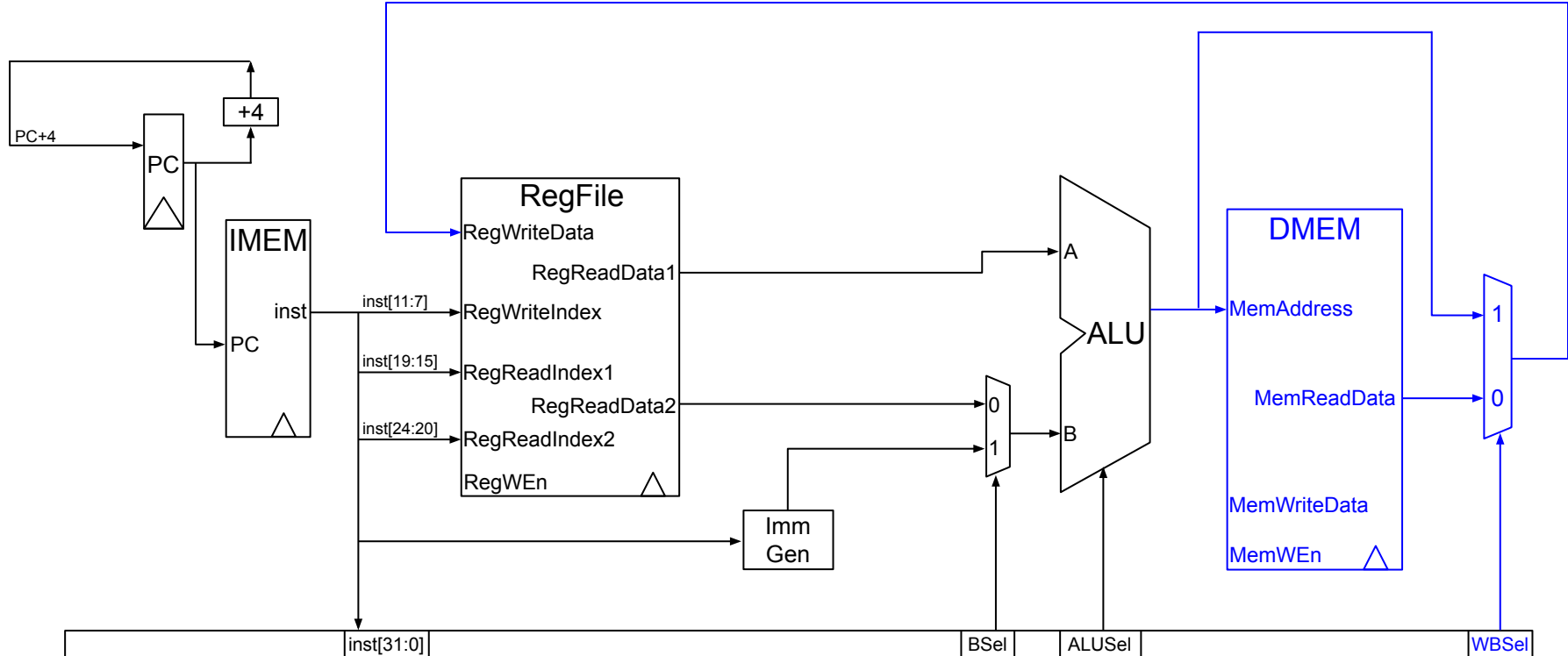
Load Stage 5: Register Write

Add a mux to choose which value to write back to the register!
How do we know which value to write back?



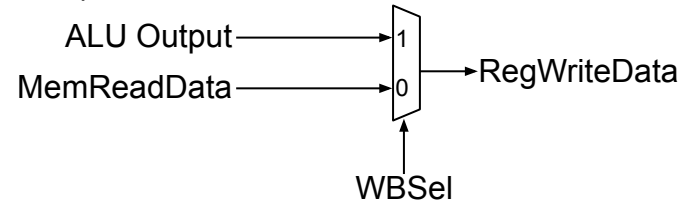
Add a new control signal: **WBSe1**
(write-back select)

Load Stage 5: Register Write

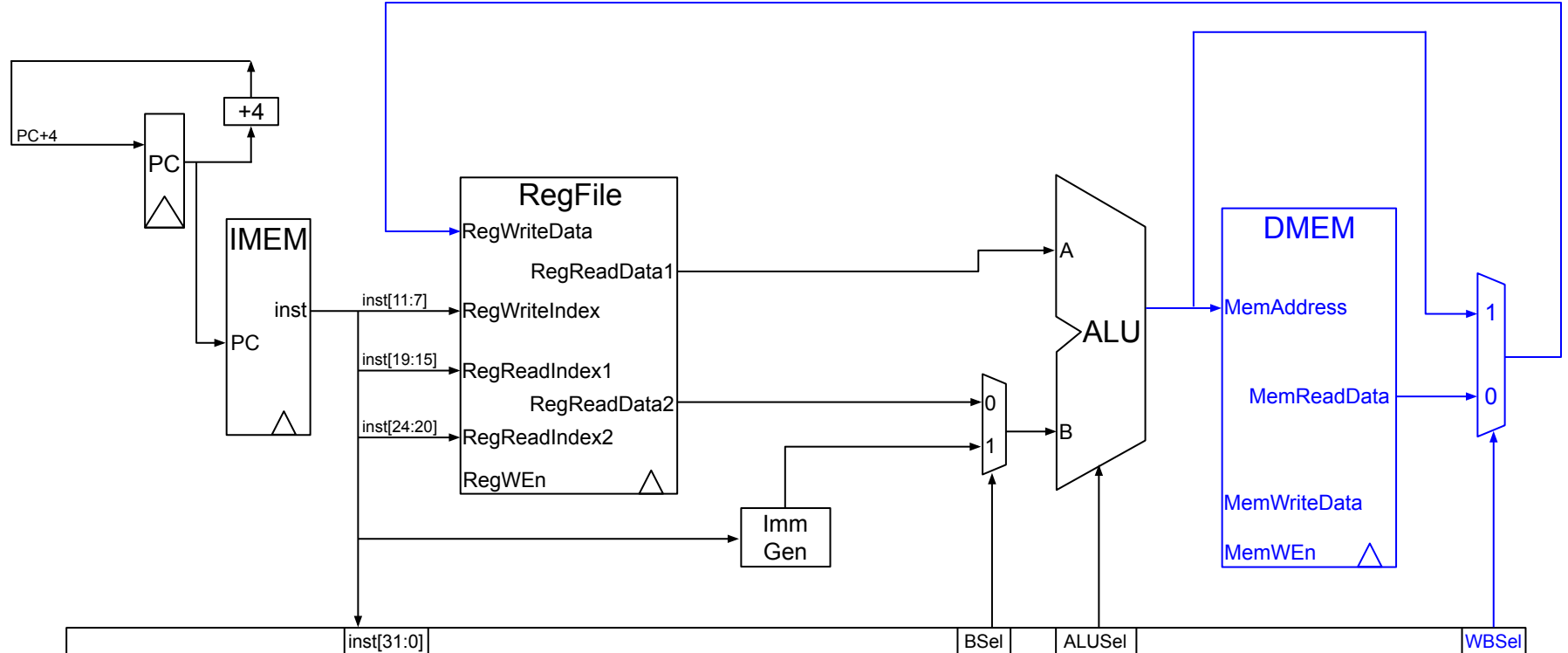


New Control Signal: WBSel

- Write-back select
- Chooses which value to write back to the register
 - WBSel=0: Write the data read from memory (MemReadData) to the register
 - WBSel=1: Write the ALU output to the register
 - There's another WBSel value, which we'll see later
- Control logic subcircuit decodes instruction and outputs appropriate WBSel



Does this work for lw, lh, lb, and lbu?



Partial Loads

- Different types of loads:
 - **lw** (load word): Read 4 bytes from memory
 - **lh** (load half-word): Read 2 bytes from memory, sign-extend to 4 bytes
 - **lhu** (load half-word unsigned): Read 2 bytes from memory, zero-extend to 4 bytes
 - **lb** (load byte): Read 1 byte from memory, sign-extend to 4 bytes
 - **lbu** (load byte unsigned): Read 1 byte from memory, sign-extend to 4 bytes
- Some additional circuitry needed to extract the correct bytes from memory
 - You'll implement a partial load subcircuit in Project 3B!

Datapath for Stores

List of RISC-V Instructions

Instruction	Name	Type
add	ADD	R
sub	SUBtract	R
and	bitwise AND	R
or	bitwise OR	R
xor	bitwise XOR	R
sll	Shift Left Logical	R
srl	Shift Right Logical	R
sra	Shift Right Arithmetic	R
slt	Set Less Than (signed)	R
sltu	Set Less Than (Unsigned)	R
addi	ADD Immediate	I
andi	bitwise AND Immediate	I
ori	bitwise OR Immediate	I
xori	bitwise XOR Immediate	I

Instruction	Name	Type
slli	Shift Left Logical Immediate	I*
srli	Shift Right Logical Immediate	I*
srai	Shift Right Arithmetic Immediate	I*
slti	Set Less Than Immediate (signed)	I
sltiu	Set Less Than Immediate (Unsigned)	I
lb	Load Byte	I
lbu	Load Byte (Unsigned)	I
lh	Load Half-word	I
lhu	Load Half-word (Unsigned)	I
lw	Load Word	I

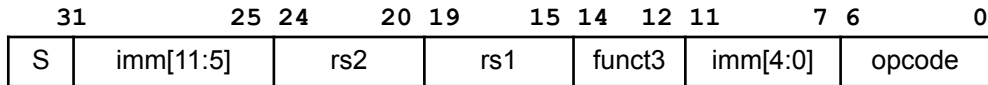
Instruction	Name	Type
sb	Store Byte	S
sh	Store Half-word	S
sw	Store Word	S
beq	Branch if Equal	B
bge	Branch if Greater or Equal (signed)	B
bgeu	Branch if Greater or Equal (Unsigned)	B
blt	Branch if Less Than (signed)	B
bltu	Branch if Less Than (Unsigned)	B
bne	Branch if Not Equal	B
jal	Jump And Link	J
jalr	Jump And Link Register	I
auipc	Add Upper Immediate to PC	U
lui	Load Upper Immediate	U

Store instructions

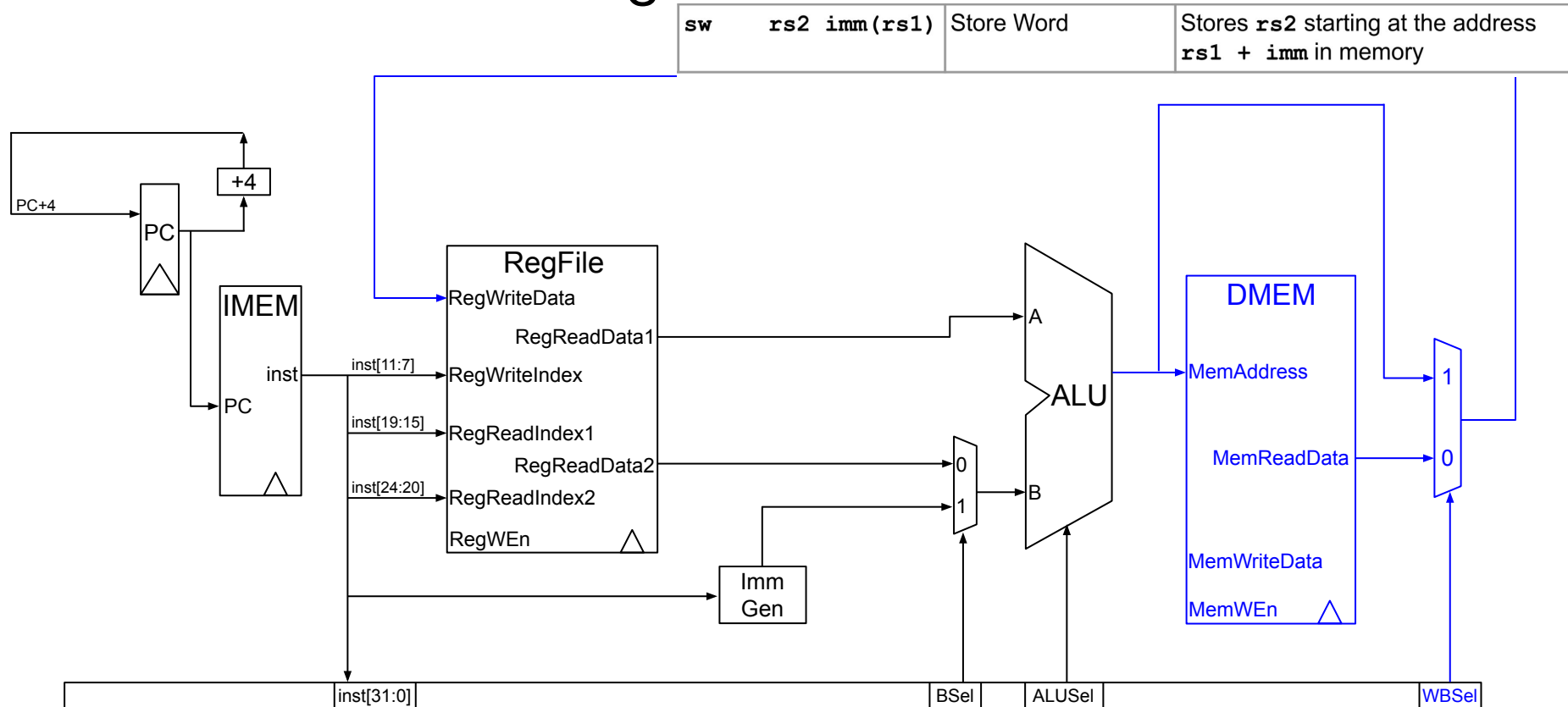
sw rs2 imm(rs1)	Store Word	Stores rs2 starting at the address rs1 + imm in memory
-------------------------------	------------	--

- What is the **sw** workflow?

- Basics: read instruction, parse instruction, etc.
- imm + rs1**: Add the value in register **rs1** to the immediate in the instruction
- Mem[imm + rs1] = rs2**: The result of the addition is the memory address to write to.
Write the value in **rs2** into the given memory address
- PC = PC + 4**: Increment the program counter

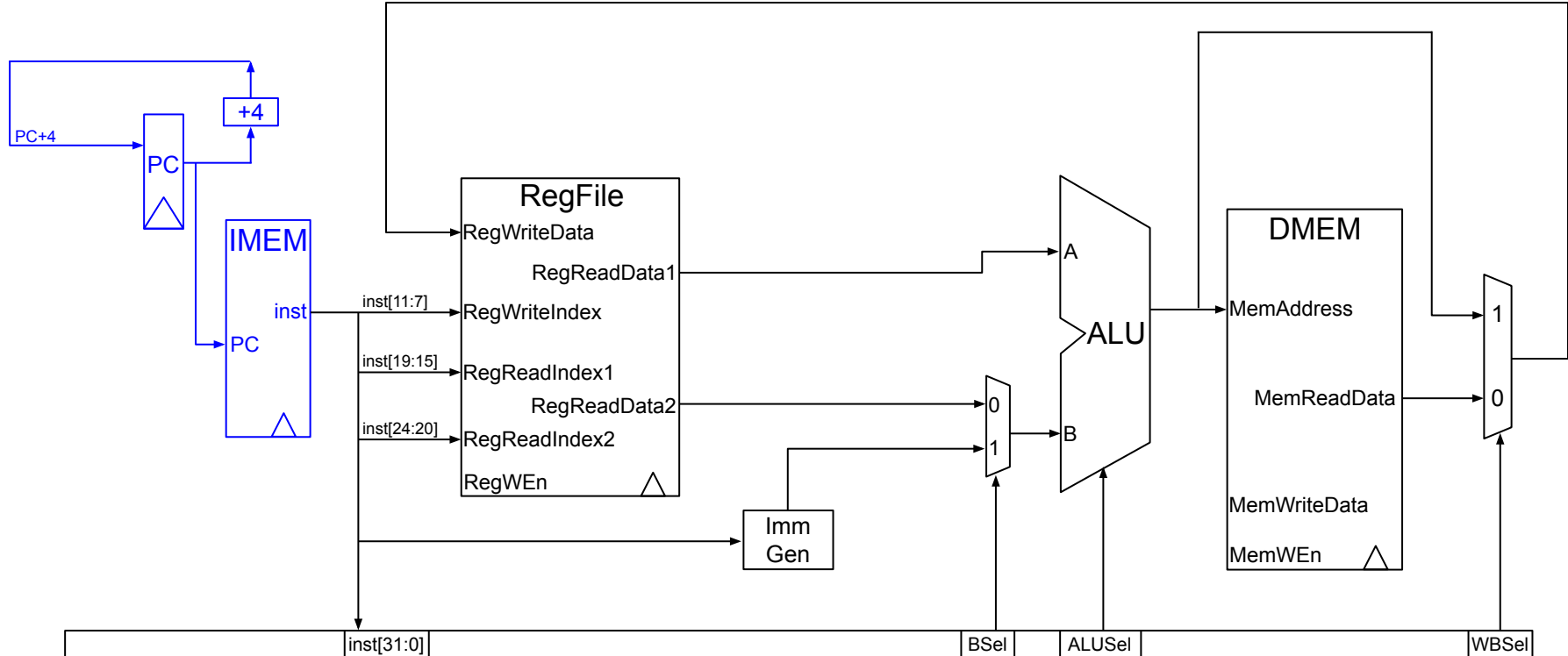


Discussion: What changes?



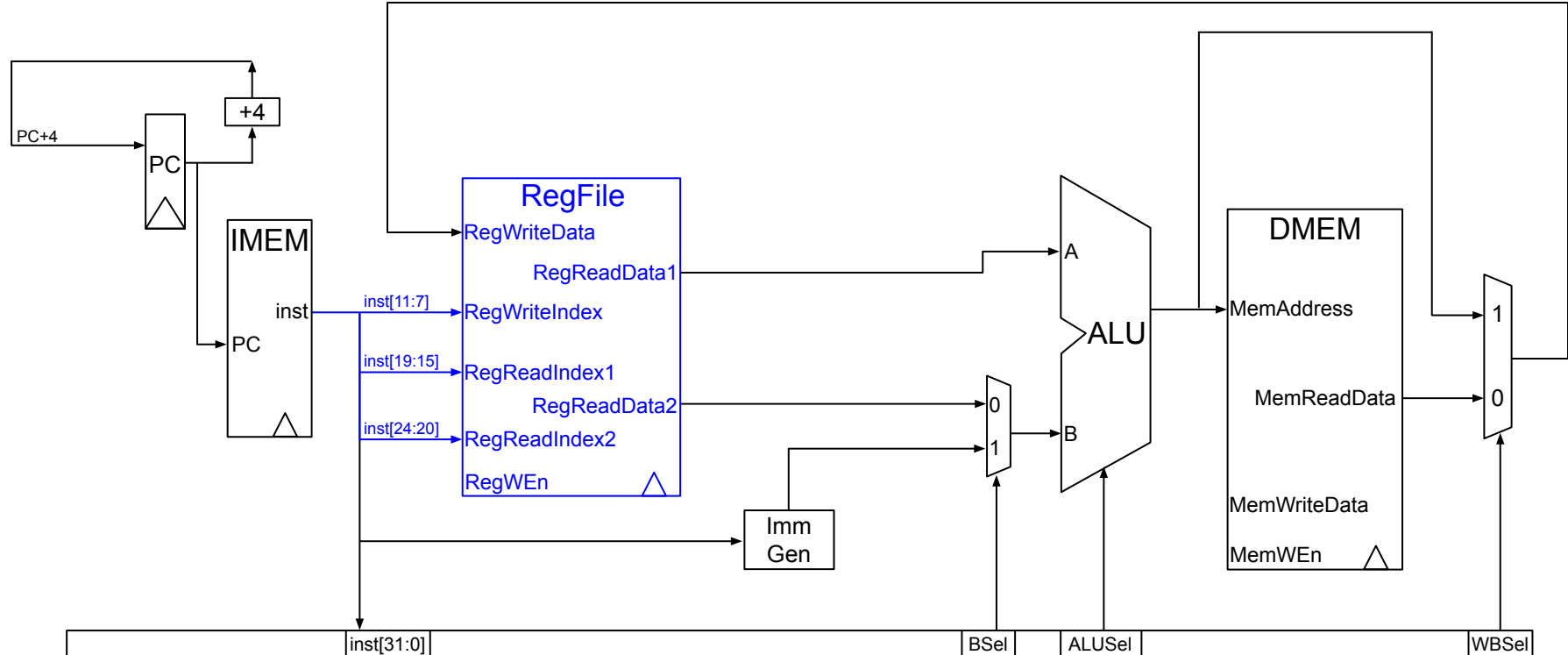
As before, we read the instruction from **IMEM**.

Store Stage 1: Instruction Fetch (IF)



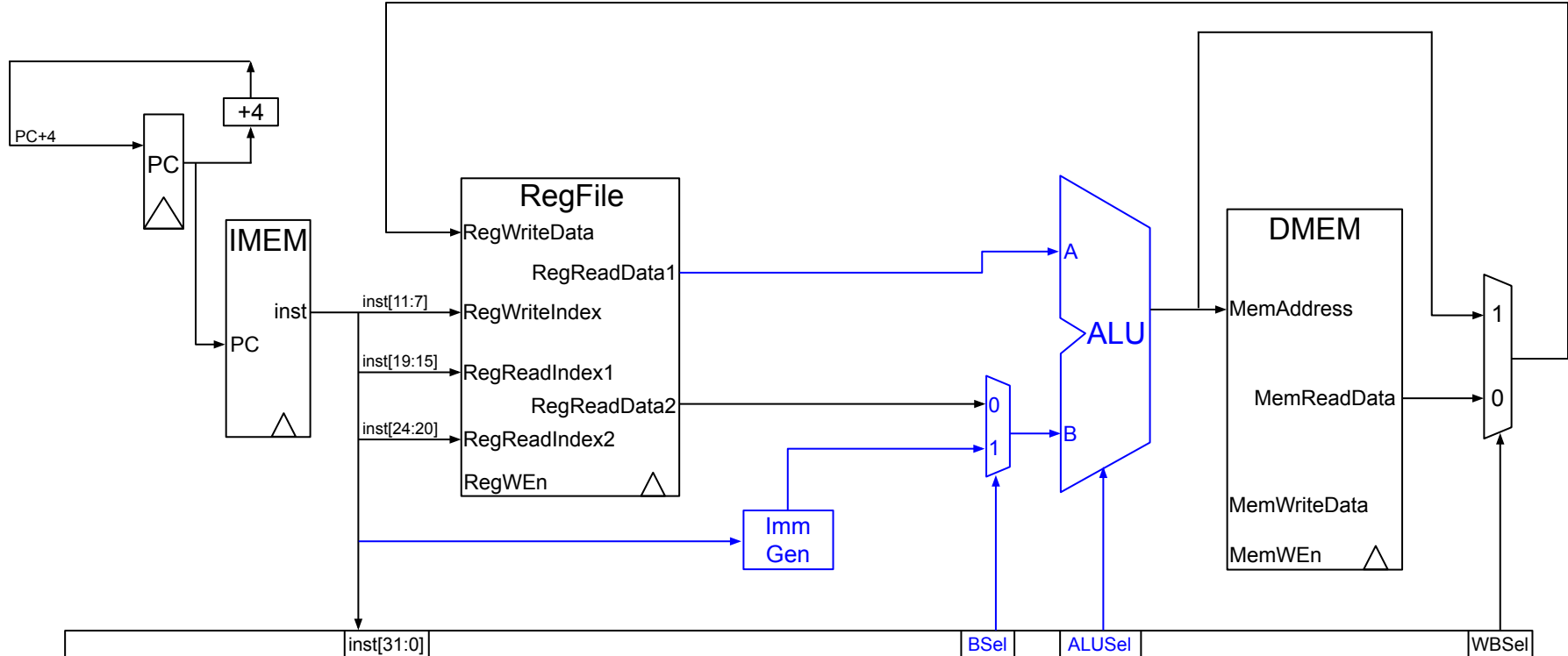
Store Stage 2: Instruction Decode (ID)

Note that we have to read values from both **rs1** and **rs2**!



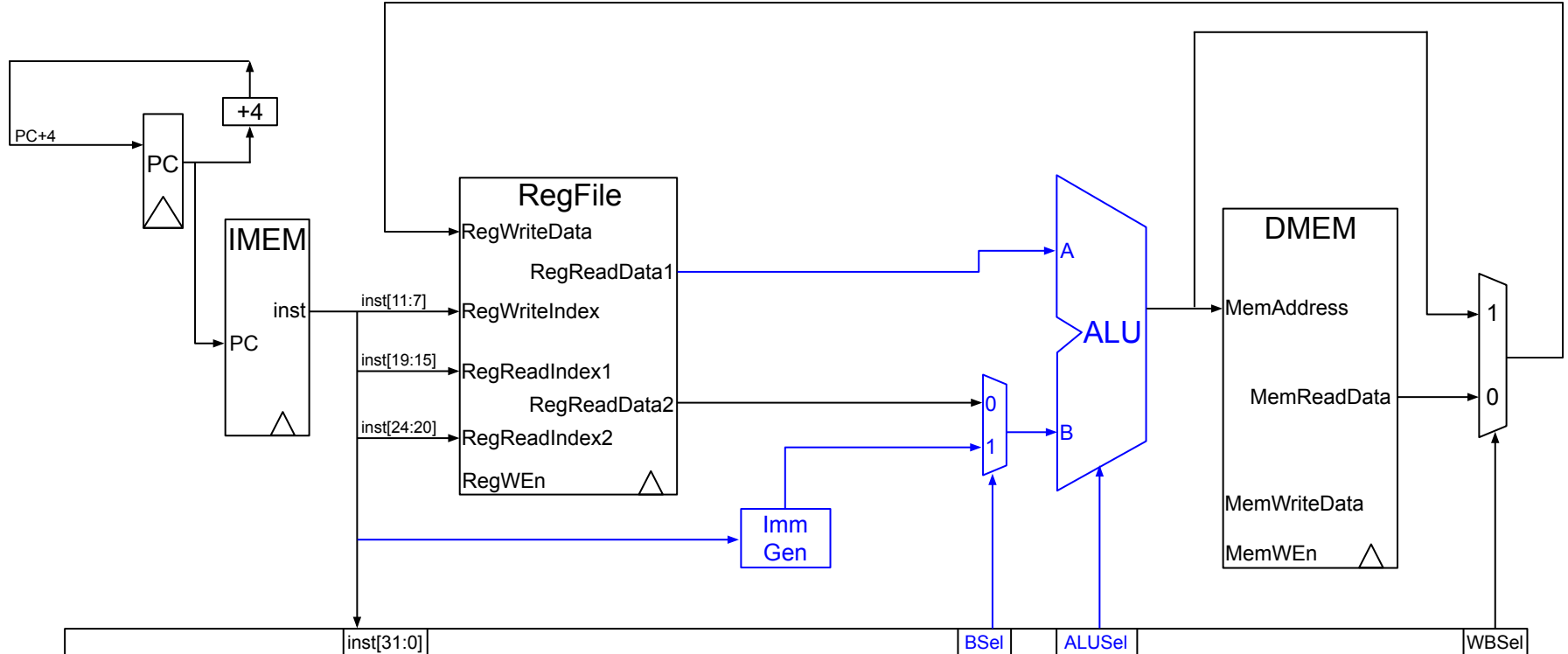
Store Stage 3: Execute

What are we computing?
Address = value in **rs1** + immediate
Bsel = 1!



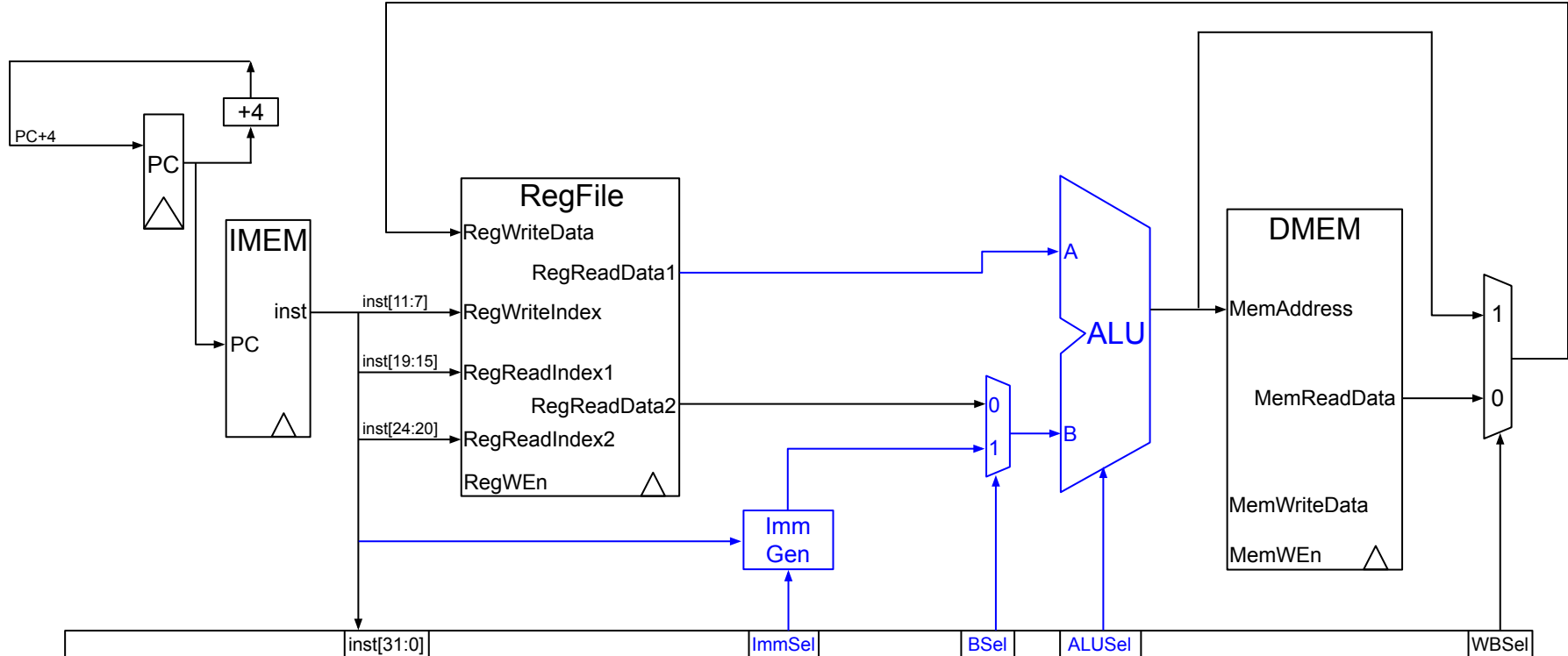
Store Stage 3: Execute

Problem: I-type immediates are in inst[31:20]. S-type immediates are in inst[31:25, 11:7]. How to support both?



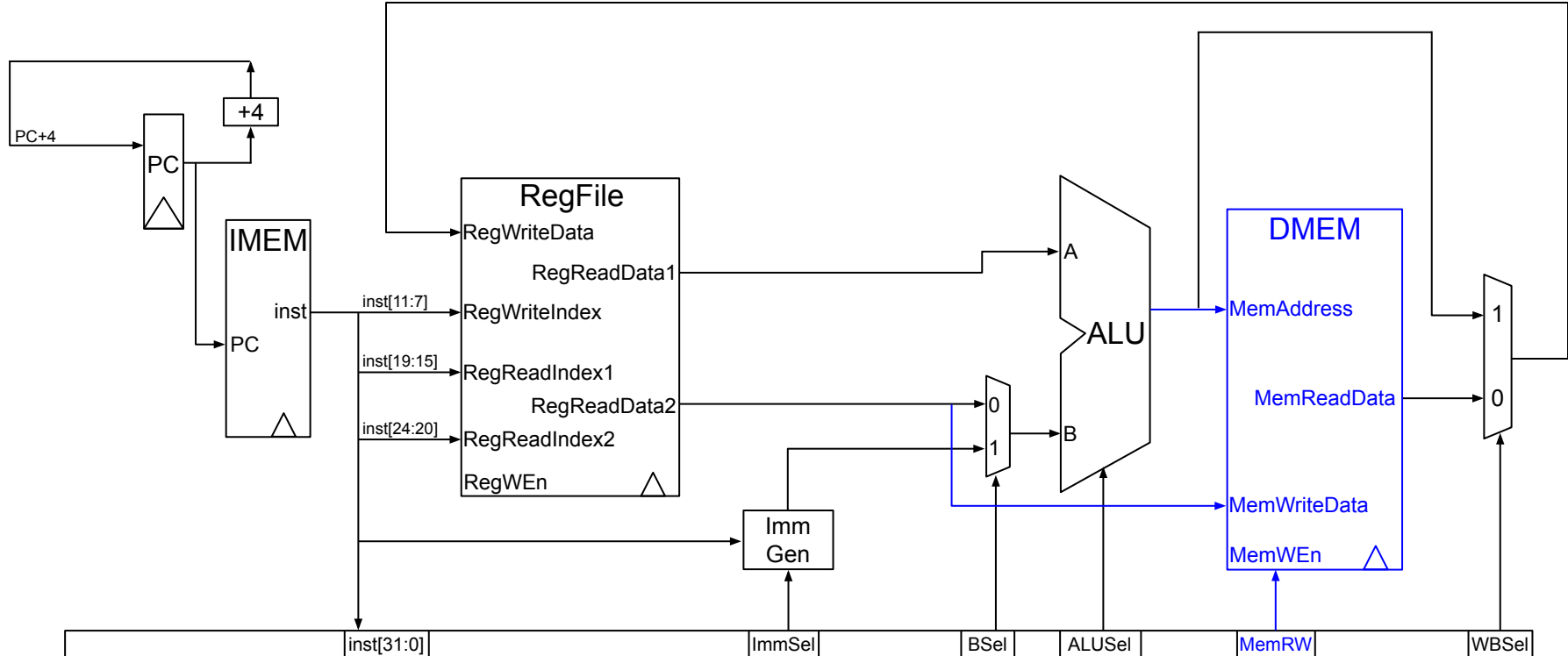
Store Stage 3: Execute

Add a new control signal, **ImmSel** (immediate select). Update immediate generator to support I-types and S-types.



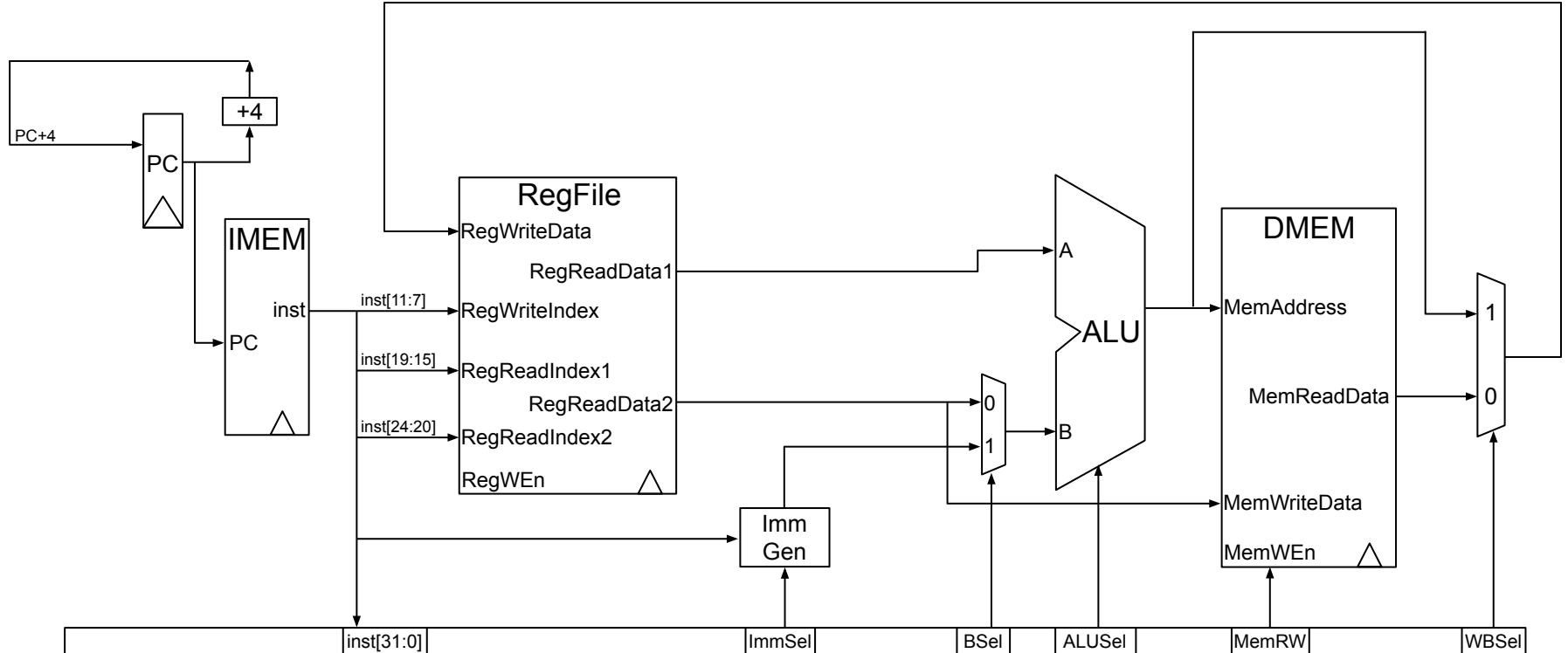
Store Stage 4: Memory

How do we tell when we want to write to memory? Add a **MemWE_n** (memory write-enable) control signal!



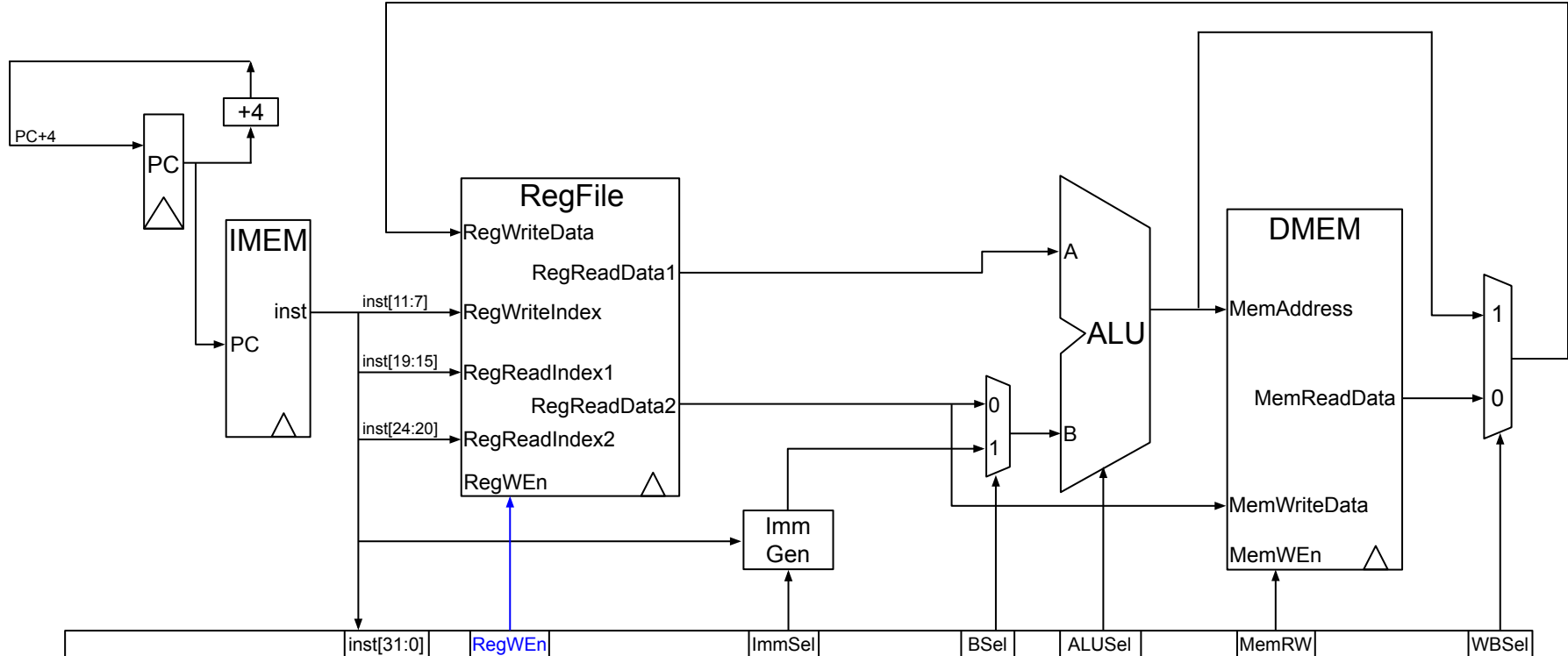
Store Stage 5: Register Write

What are we writing back to the register? Nothing! Store instructions don't modify register values.



Store Stage 5: Register Write

How do we tell we don't want to write to a register? Add a **RegWEn** (register write-enable) control signal!



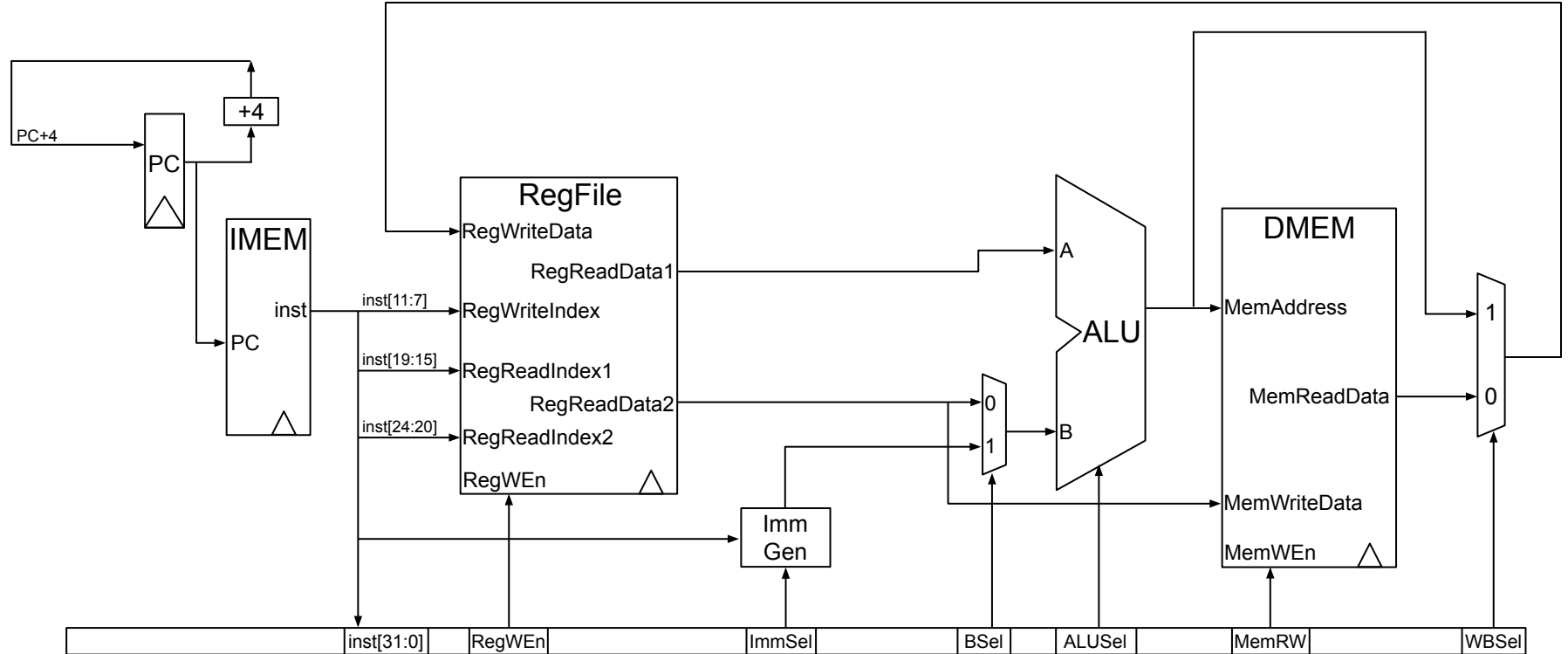
New Control Signal: ImmSel

- Immediate select
- Choose which bits of the instruction to use for the immediate
 - One ImmSel value for each instruction type
- Only used by the immediate generator

New Control Signals: MemWEn, RegWEn

- Memory write-enable, register write-enable
- Chooses whether to modify memory/registers for this instruction
 - MemWEn=0: Do not modify memory
 - MemWEn=1: Write data to memory
 - RegWEn=0: Do not modify registers
 - RegWEn=1: Write value to register
- Control logic subcircuit decodes instruction and outputs appropriate WEn
 - Example: For store instructions, MemWEn=1 and RegWEn=0
 - Example: For R-type instructions, MemWEn=0 and RegWEn=1

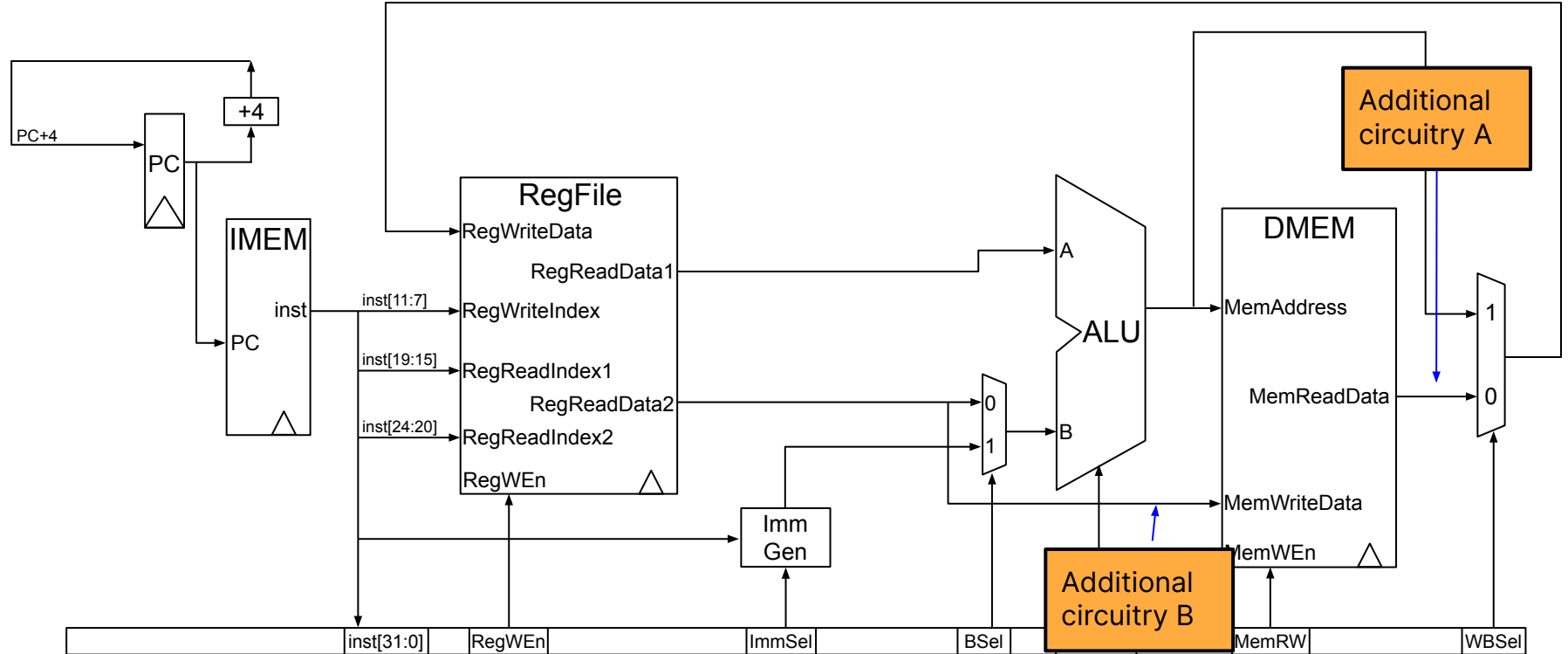
Does this circuit handle sw, sh, and sb?



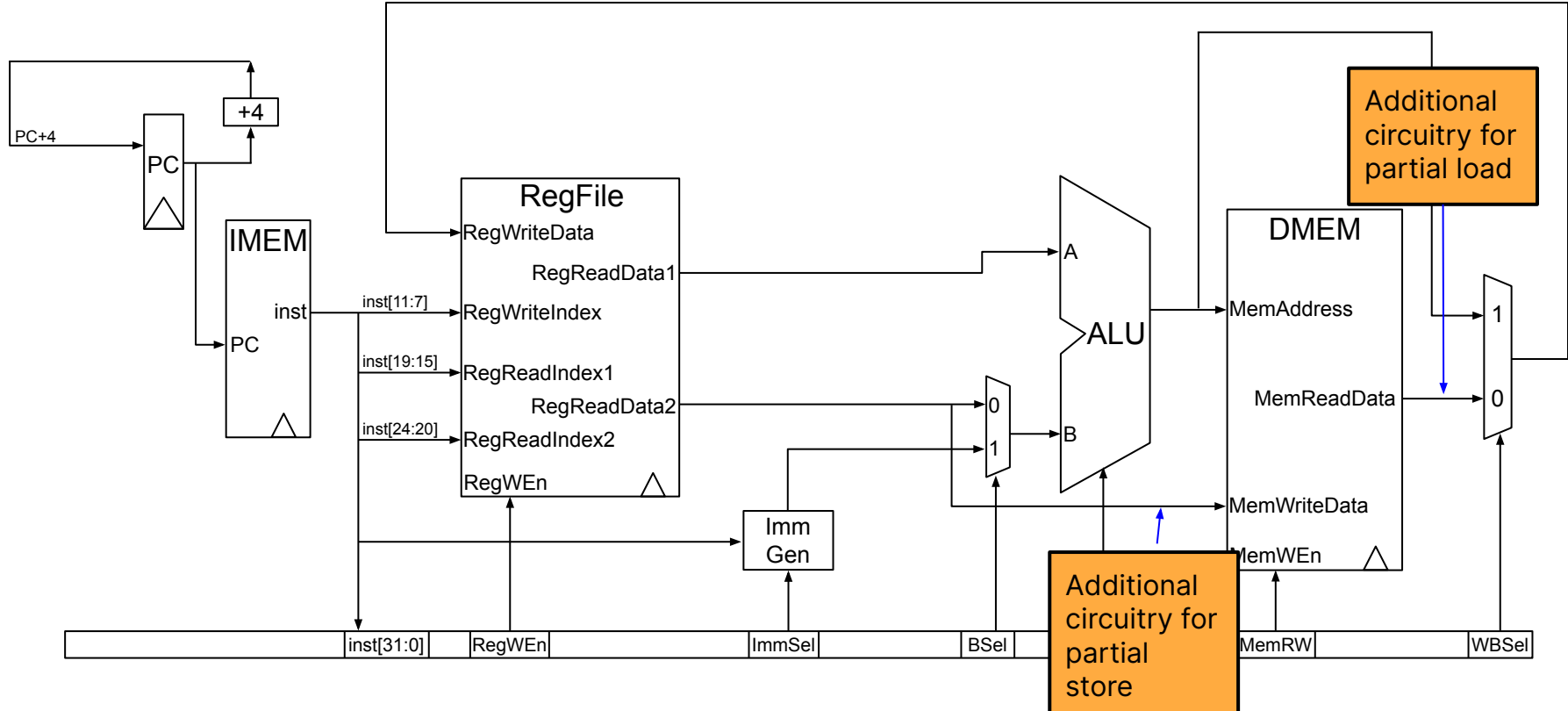
Partial Stores

- Different types of stores:
 - **sw** (store word): Write 4 bytes to memory
 - **sh** (store half-word): Write lowest 2 bytes to memory
 - **sb** (store byte): Write lowest byte to memory
- Some additional circuitry needed to send the correct bytes to memory
 - You'll implement a partial store subcircuit in Project 3B!

Datapath with partial loads and stores



Datapath with partial loads and stores



Datapath for Branches

List of RISC-V Instructions

Instruction	Name	Type
add	ADD	R
sub	SUBtract	R
and	bitwise AND	R
or	bitwise OR	R
xor	bitwise XOR	R
sll	Shift Left Logical	R
srl	Shift Right Logical	R
sra	Shift Right Arithmetic	R
slt	Set Less Than (signed)	R
sltu	Set Less Than (Unsigned)	R
addi	ADD Immediate	I
andi	bitwise AND Immediate	I
ori	bitwise OR Immediate	I
xori	bitwise XOR Immediate	I

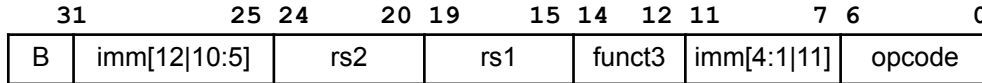
Instruction	Name	Type
slli	Shift Left Logical Immediate	I*
srli	Shift Right Logical Immediate	I*
srai	Shift Right Arithmetic Immediate	I*
slti	Set Less Than Immediate (signed)	I
sltiu	Set Less Than Immediate (Unsigned)	I
lb	Load Byte	I
lbu	Load Byte (Unsigned)	I
lh	Load Half-word	I
lhu	Load Half-word (Unsigned)	I
lw	Load Word	I

Instruction	Name	Type
sb	Store Byte	S
sh	Store Half-word	S
sw	Store Word	S
beq	Branch if EQual	B
bge	Branch if Greater or Equal (signed)	B
bgeu	Branch if Greater or Equal (Unsigned)	B
blt	Branch if Less Than (signed)	B
bltu	Branch if Less Than (Unsigned)	B
bne	Branch if Not Equal	B
jal	Jump And Link	J
jalr	Jump And Link Register	I
auipc	Add Upper Immediate to PC	U
lui	Load Upper Immediate	U

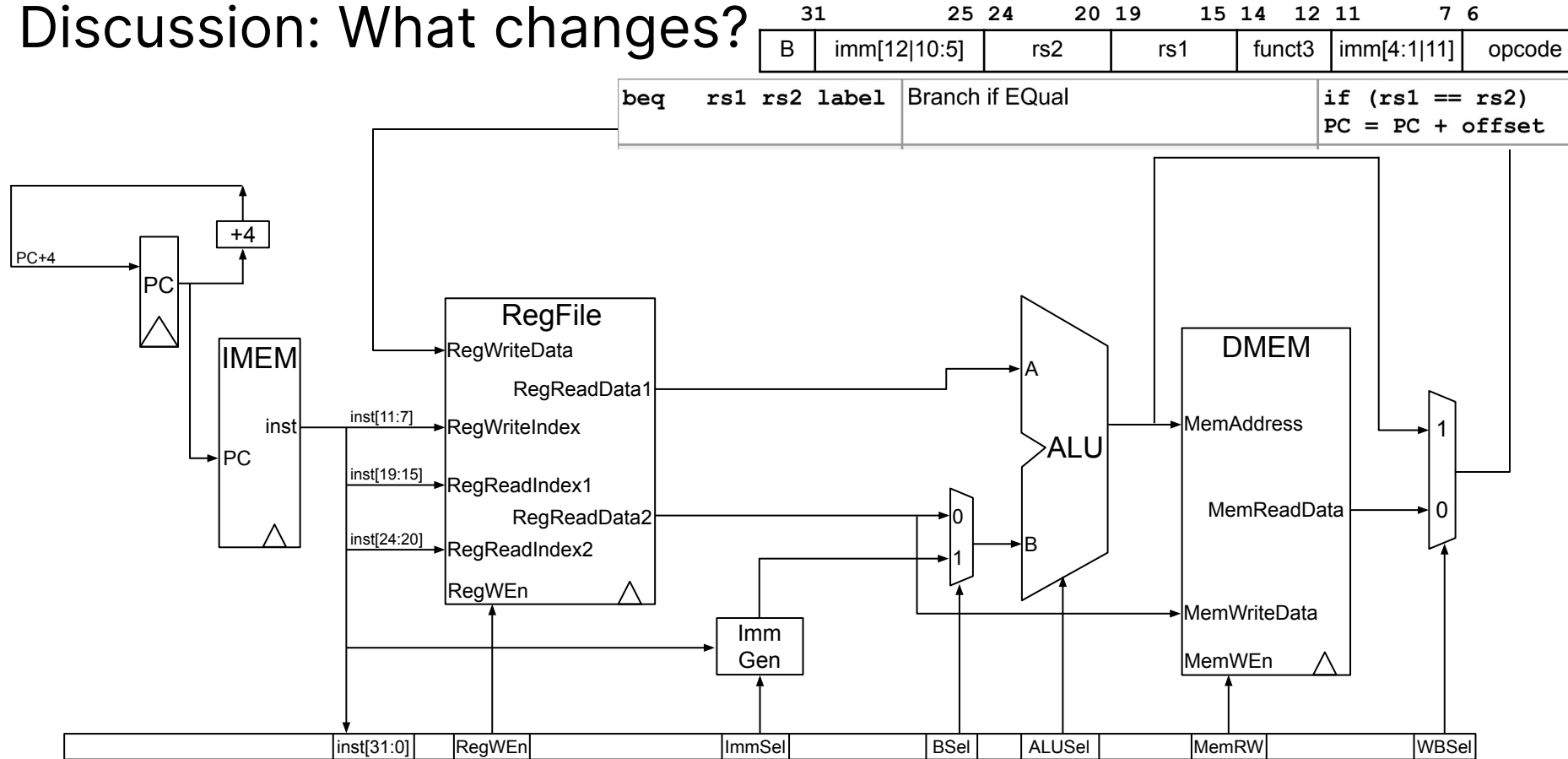
Branch instructions

<code>beq rs1 rs2 label</code>	Branch if EQual	<code>if (rs1 == rs2)</code> <code>PC = PC + offset</code>
--------------------------------	-----------------	---

- What is the **beq** workflow?
 - Basics: read instruction; parse instruction, etc.
 - **rs1 == rs2?**: Compare **rs1** and **rs2**
 - If the comparison is true: set **PC = PC + immediate**
 - If the comparison is false: set **PC = PC + 4**

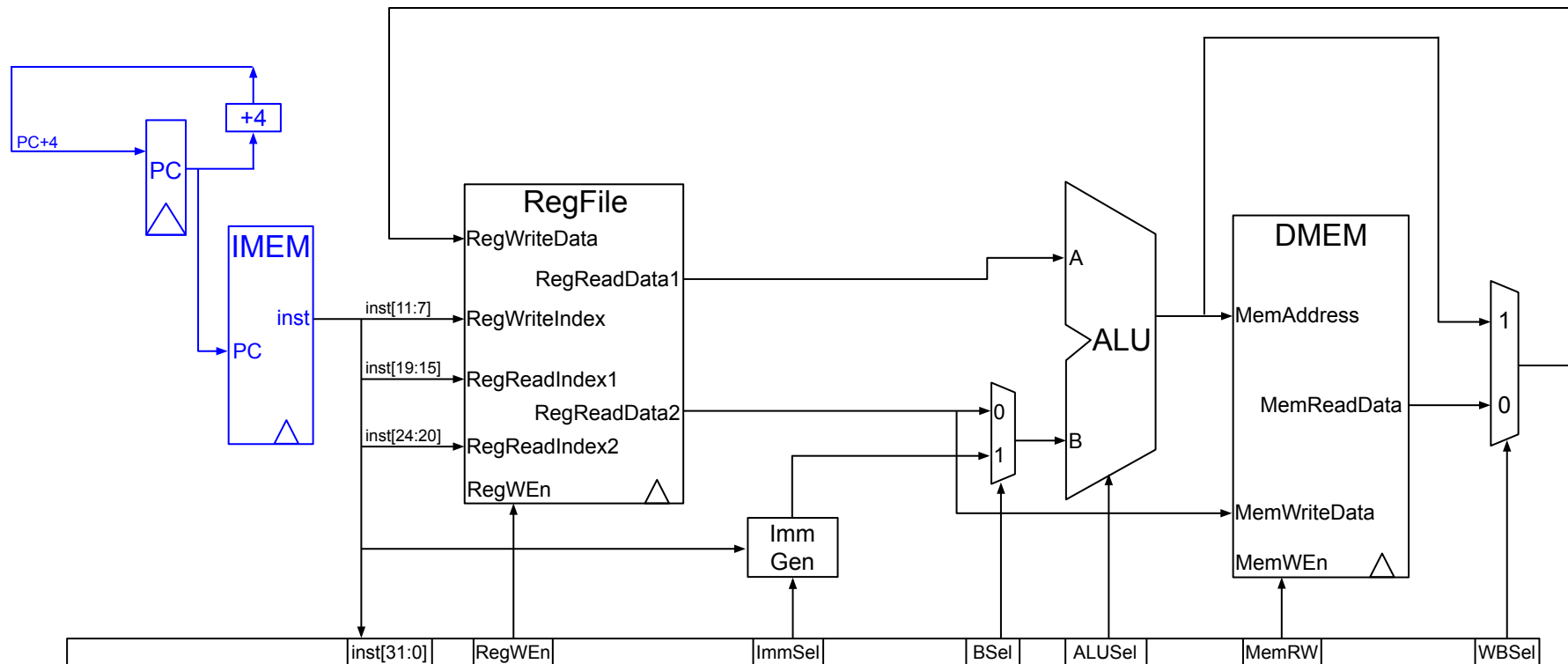


Discussion: What changes?



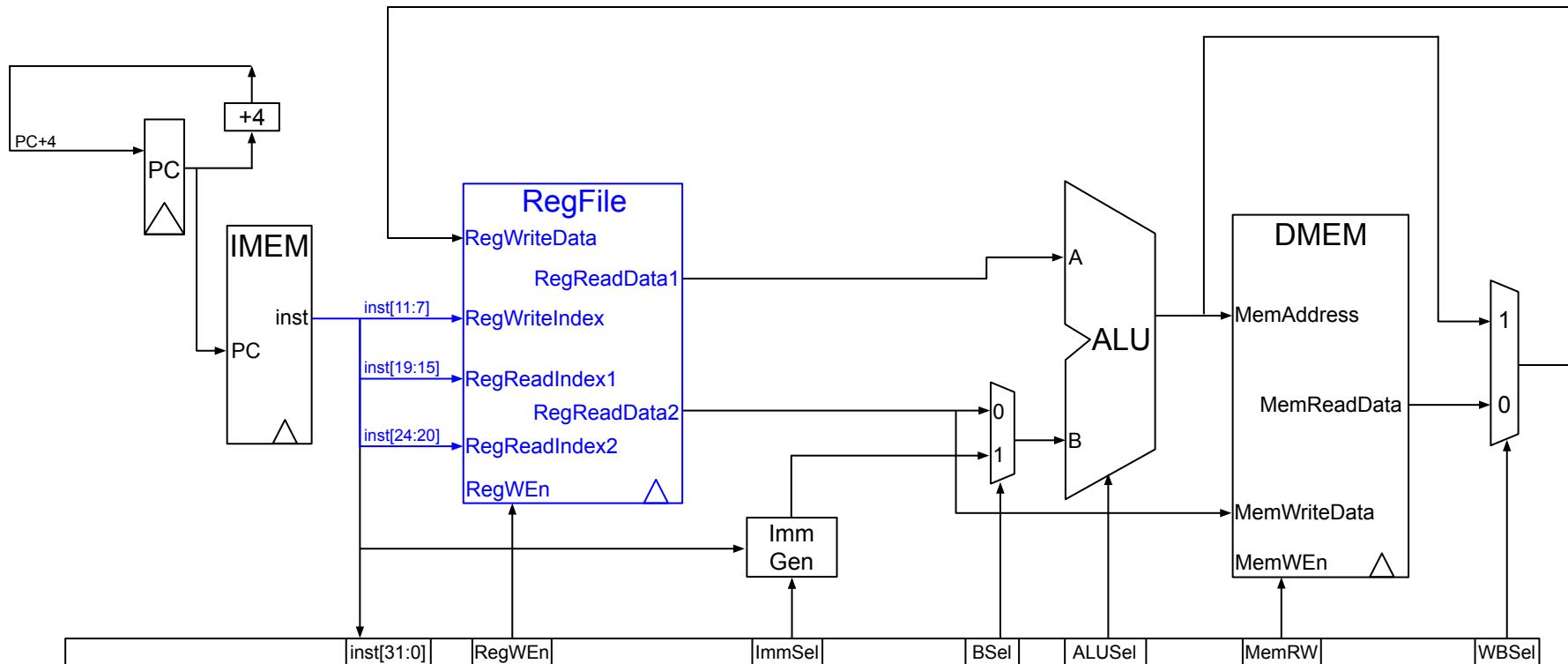
Branch Stage 1: Instruction Fetch (IF)

As before, we read the instruction from IMEM. We'll modify the PC logic later, in the write-back stage.



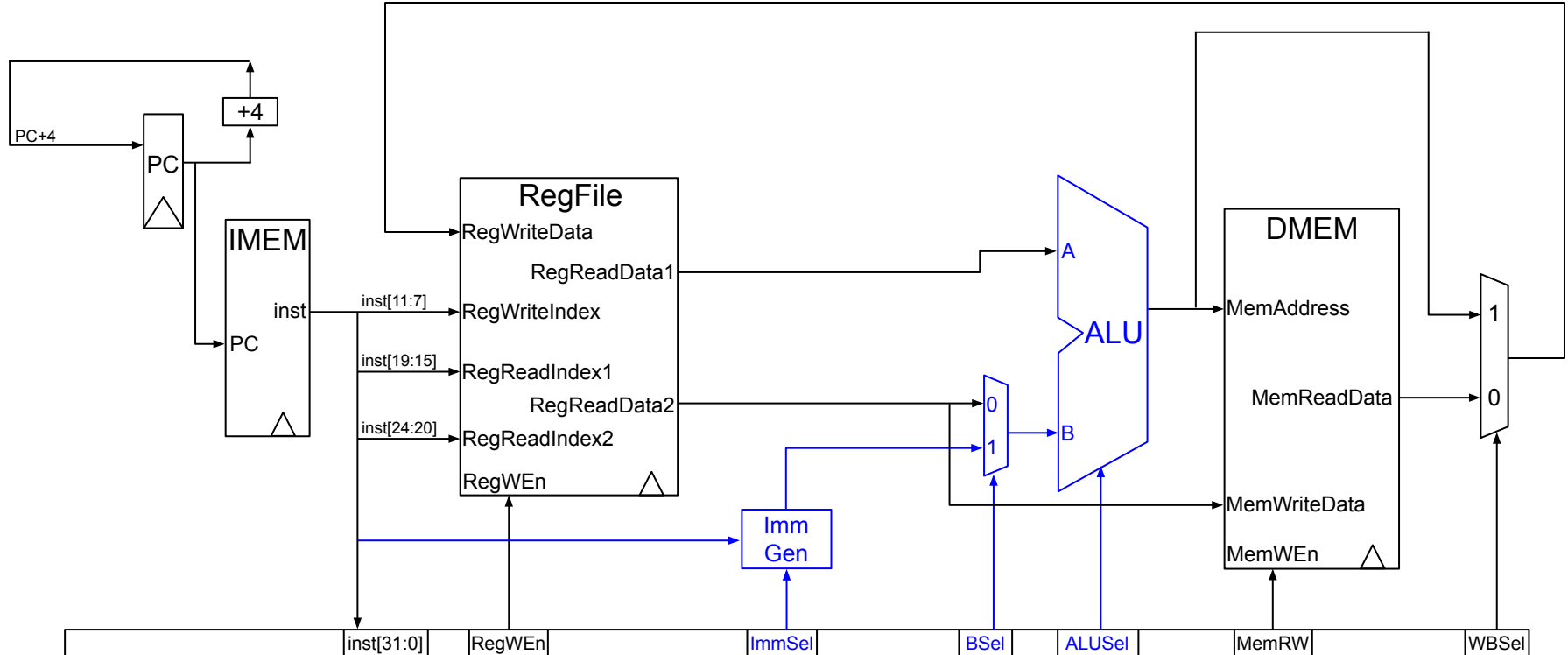
Branch Stage 2: Instruction Decode (ID)

Note that we have to read values from both **rs1** and **rs2**!



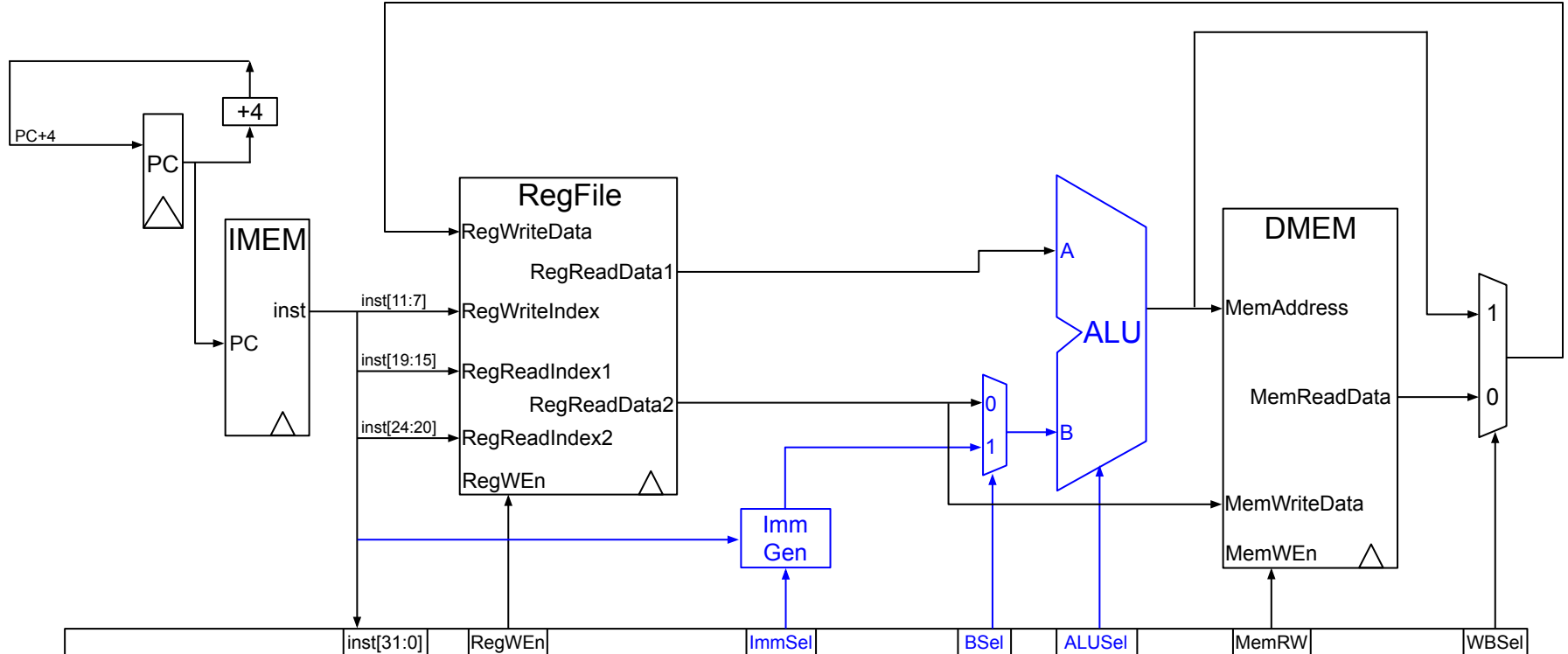
Branch Stage 3: Execute

Note that we have a new type (B-type), so we need to update the immediate generator and ImmSel.



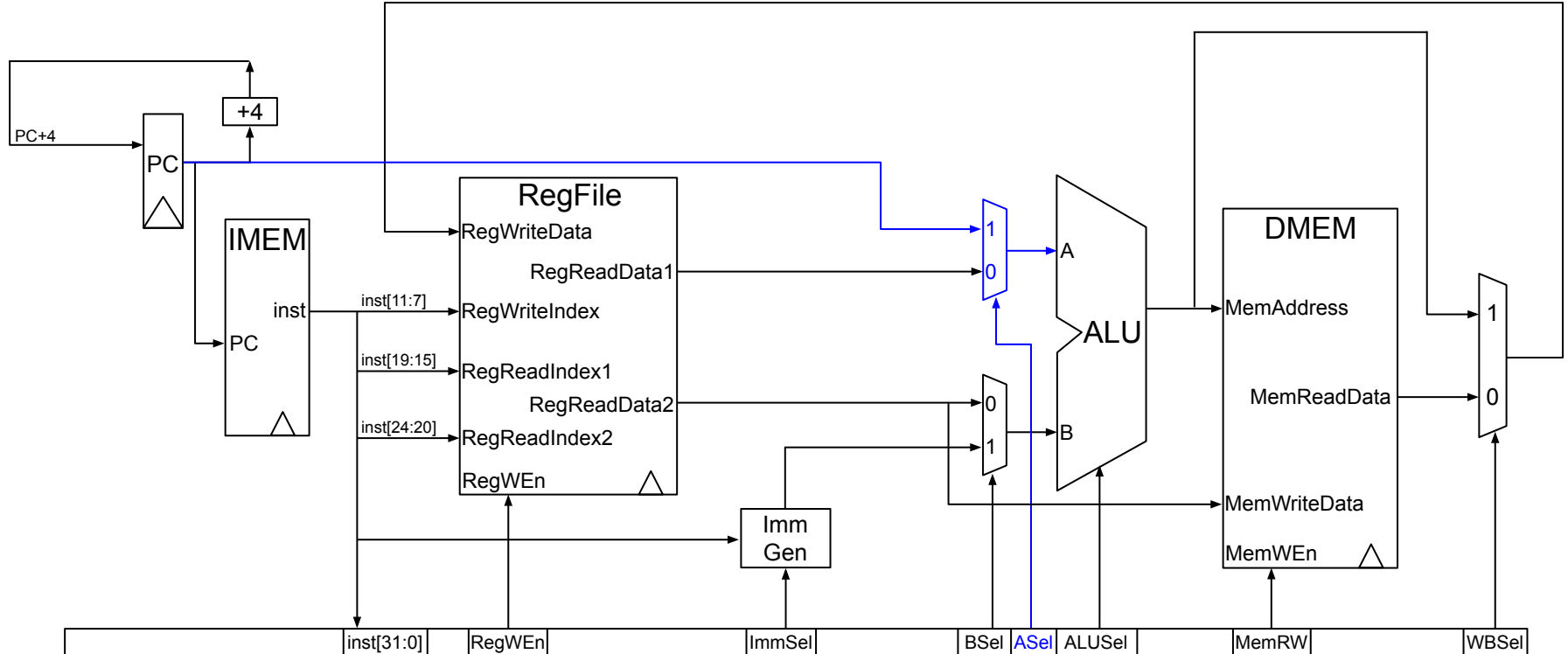
Branch Stage 3: Execute

If the branch is taken, we have to compute PC + immediate. How do we send PC to the ALU?



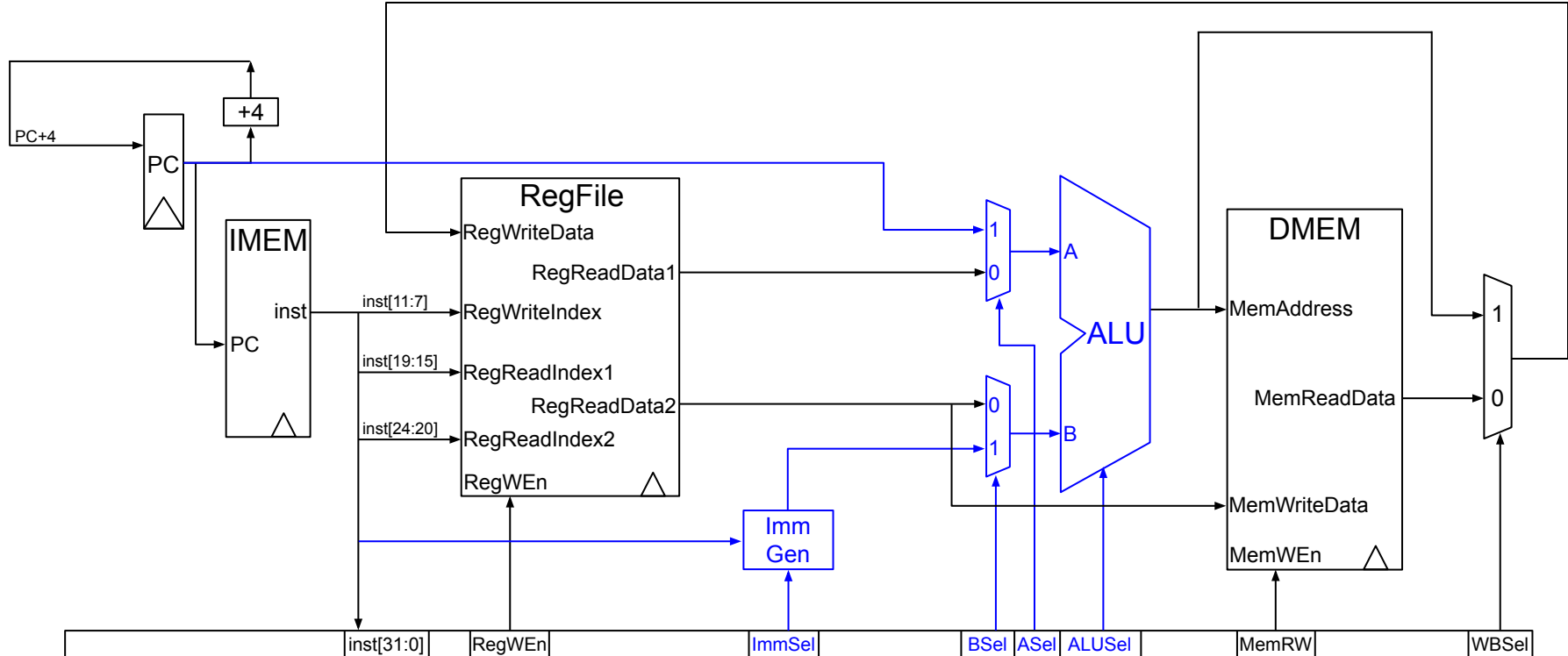
Branch Stage 3: Execute

Add a new mux and control signal (ASel) so we don't break any existing instructions.



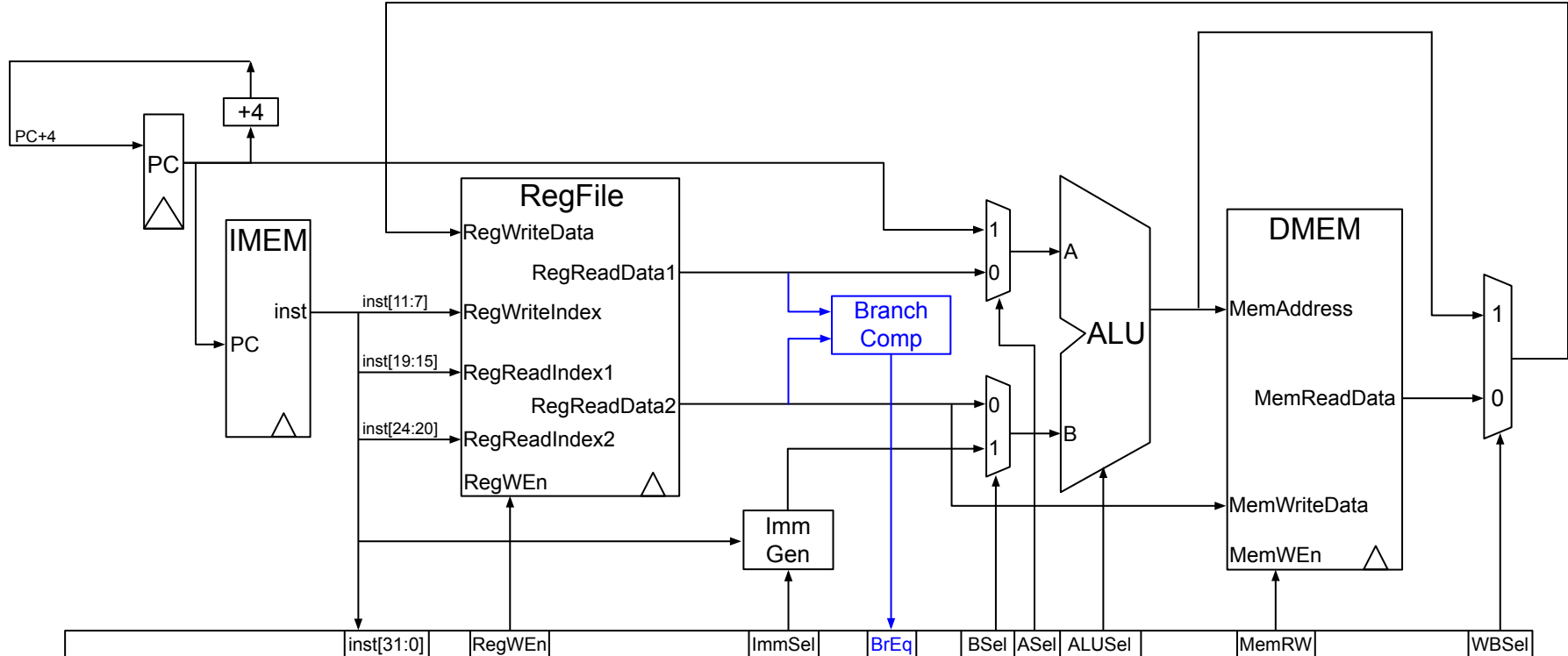
Branch Stage 3: Execute

We've computed PC + immediate, but we didn't compare the register values, and we ran out of hardware! Let's add some more.



Branch Stage 3: Execute

The branch comparator takes in register values, and sends the comparison result to the control logic subcircuit.



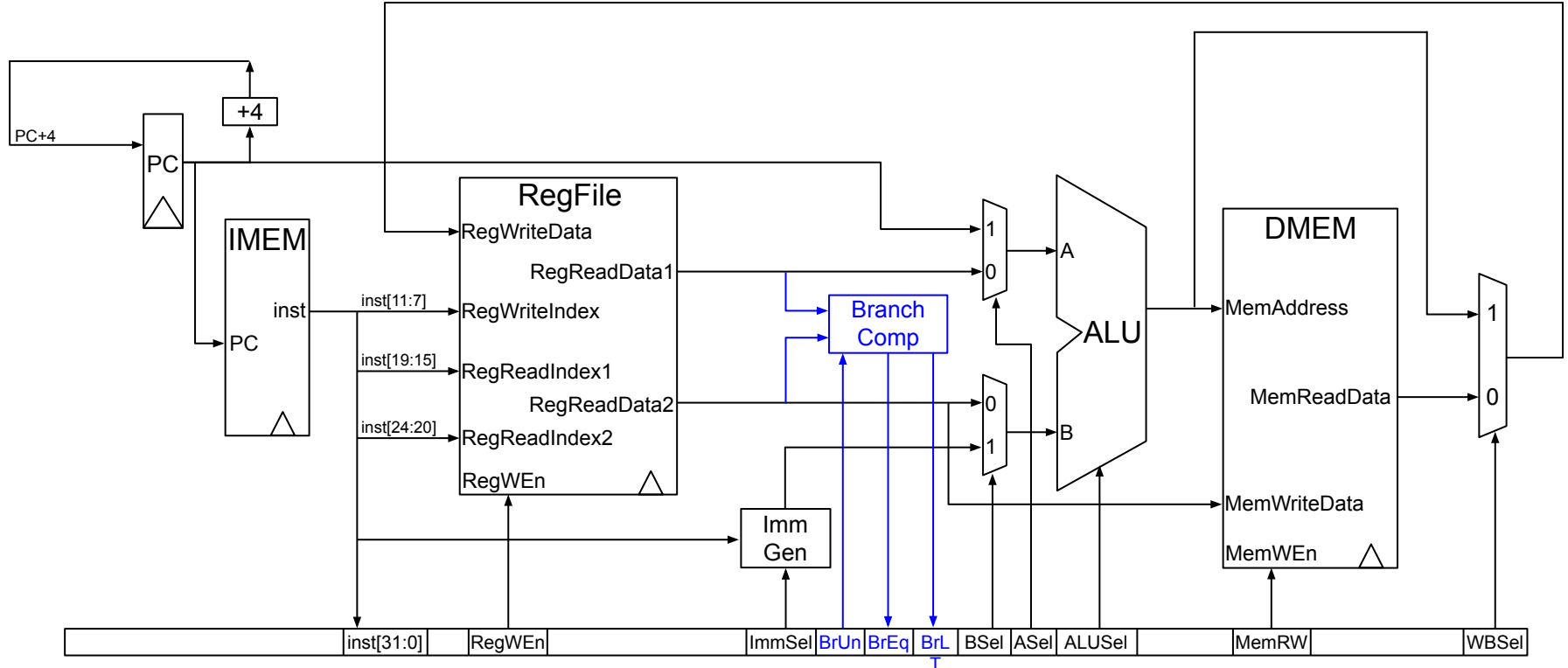
What else should we achieve with the branch comparator?

Instruction	Name	Description
beq rs1 rs2 label	Branch if Equal	if (rs1 == rs2) PC = PC + offset
bge rs1 rs2 label	Branch if Greater or Equal (signed)	if (rs1 >= rs2) PC = PC + offset
bgeu rs1 rs2 label	Branch if Greater or Equal (Unsigned)	
blt rs1 rs2 label	Branch if Less Than (signed)	if (rs1 < rs2) PC = PC + offset
bltu rs1 rs2 label	Branch if Less Than (Unsigned)	
bne rs1 rs2 label	Branch if Not Equal	if (rs1 != rs2) PC = PC + offset

We need to do “less than” comparisons, and unsigned comparisons!
How do we know whether we compare values as if they’re signed or unsigned?

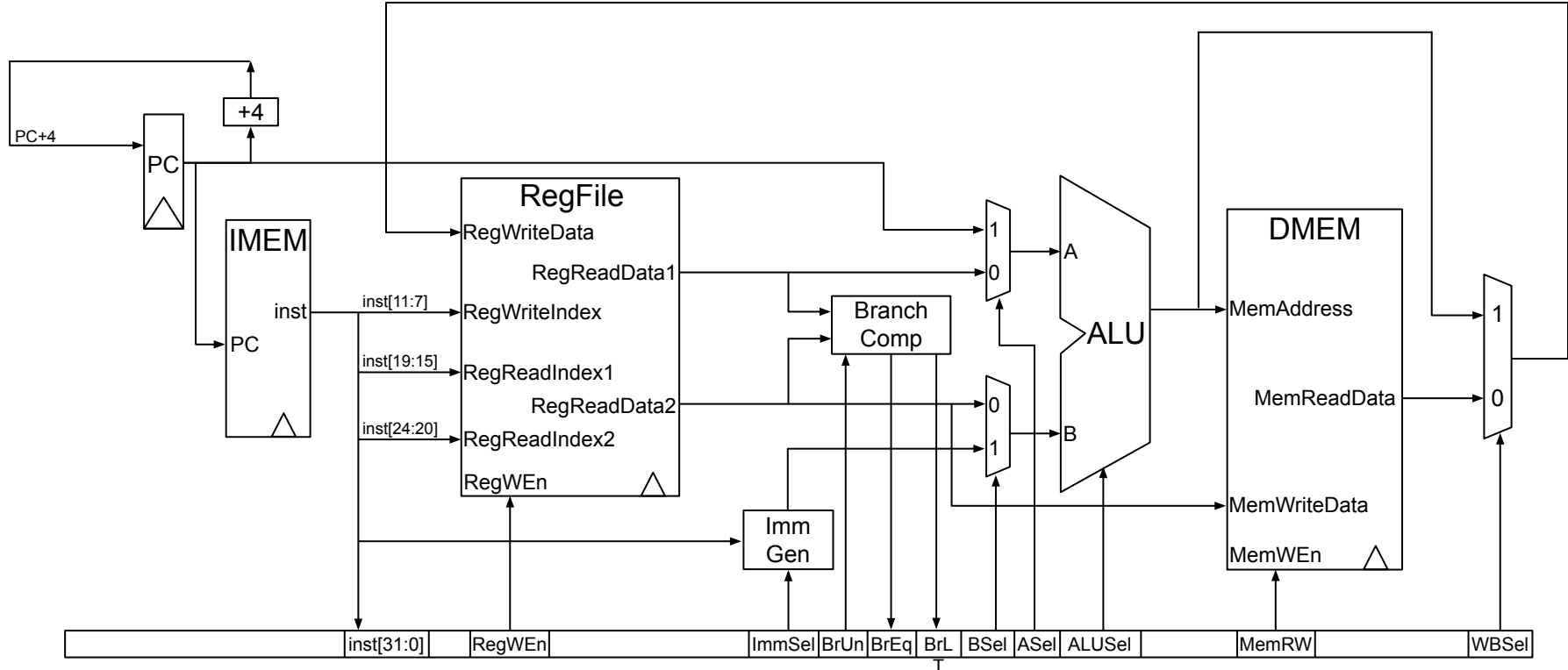
Branch Stage 3: Execute

The branch comparator takes in register values, and sends the comparison result to the control logic subcircuit.



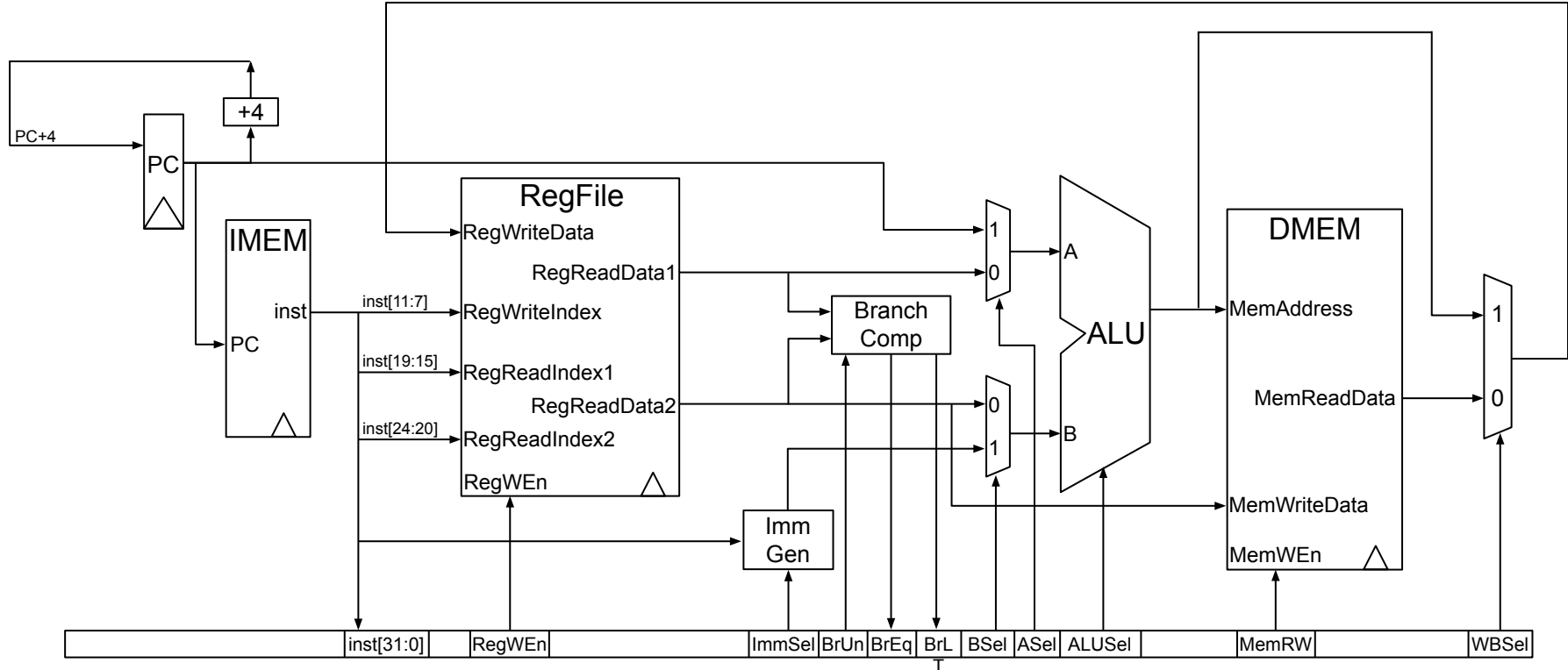
Branch Stage 4: Memory

Branch instructions don't write to memory, so nothing to do here. Don't forget to set **MemWEn=0** to disable writing to memory.



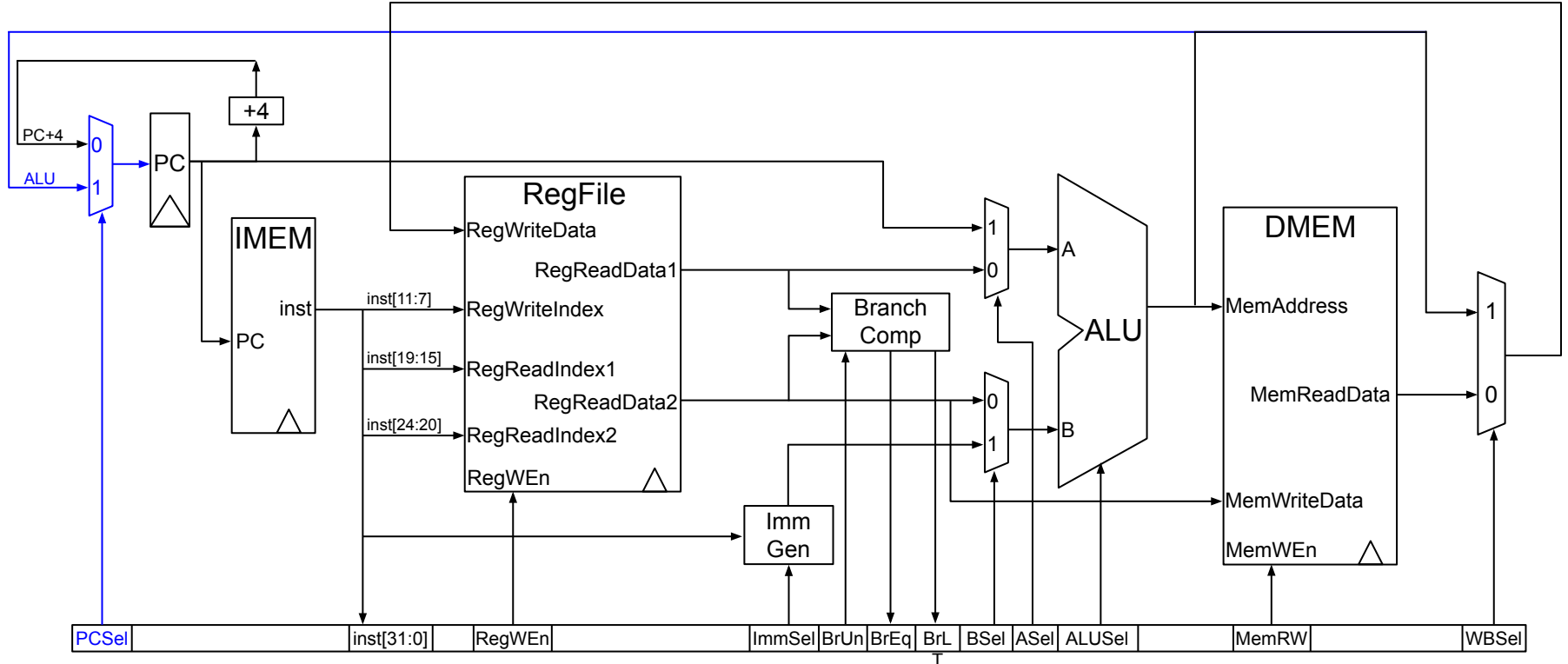
Branch Stage 5: Register Write

Branch instructions don't write to registers in the RegFile, but they might write PC = PC + immediate.



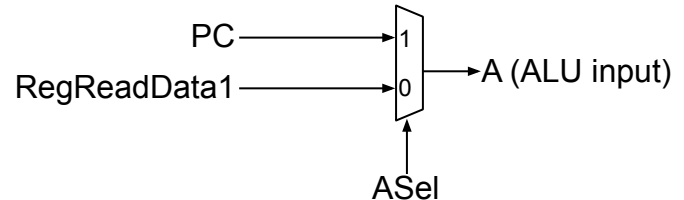
Branch Stage 5: Register Write

Add a mux and control signal (PCSel) to write either PC+4 or PC+imm (ALU output) back to PC.



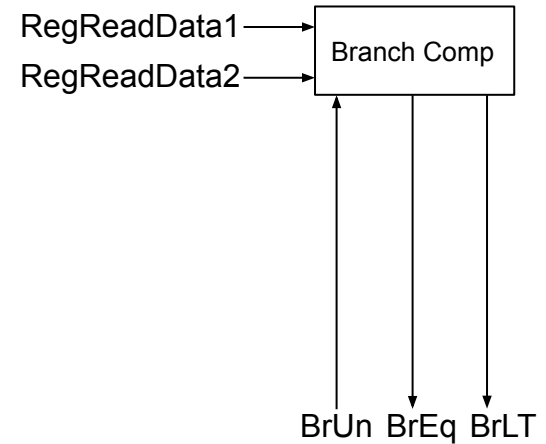
New Control Signal: ASel

- Chooses which value to send to the A input of the ALU
 - ASel=0: Send data in register `rs1` to ALU
 - ASel=1: Send PC to ALU
- Control logic subcircuit decodes instruction and outputs appropriate ASel
 - Example: For R-type instructions, set ASel=0
 - Example: For B-type instructions, set ASel=1



New Subcircuit: Branch Comparator

- New subcircuit in our datapath: branch comparator
- Inputs
 - RegReadData1 and RegReadData2: Register values
 - BrUn: New control signal. Is the branch instruction doing a signed or unsigned comparison?
- Outputs
 - BrEq: Is RegReadData1 == RegReadData2?
 - BrLt: Is RegReadData1 < RegReadData2?
 - BrLt performs a signed or unsigned comparison depending on the input BrUn



New Control Signal: PCSel

- Chooses how to update PC on the next cycle
 - PCSel=0: Set the next PC = current PC + 4
 - PCSel=1: Set the next PC = current PC + immediate (ALU output)
- Control logic subcircuit uses instruction and branch comparator output to compute PCSel
 - Example: **blt** instruction, and BrLt=1. The branch is taken, so PCSel=1.
 - Example: **beq** instruction, and BrEq=0. The branch is not taken, so PCSel=0.
 - Example: **addi** instruction. There's no branch to take, so PCSel=0.

