

CS61C: Great Ideas in Computer Architecture (Machine Structure)

Instructors: Rosalie Fang, Charles Hong, Jero Wang

Lecture feedback:

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

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)

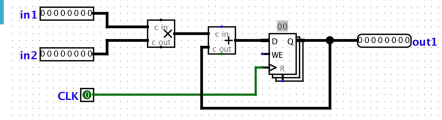
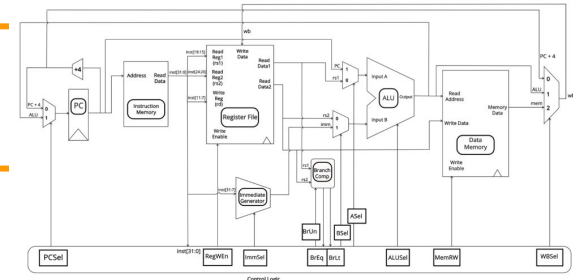
```
1000 1101 1110 0010 0000 0000 0000 0000  
1000 1110 0001 0000 0000 0000 0000 0100  
1010 1110 0001 0010 0000 0000 0000 0000  
1010 1101 1110 0010 0000 0000 0000 0100
```

**Hardware Architecture
Description**

(e.g., block diagrams)



Logic Circuit Description
(Circuit Schematic Diagrams)



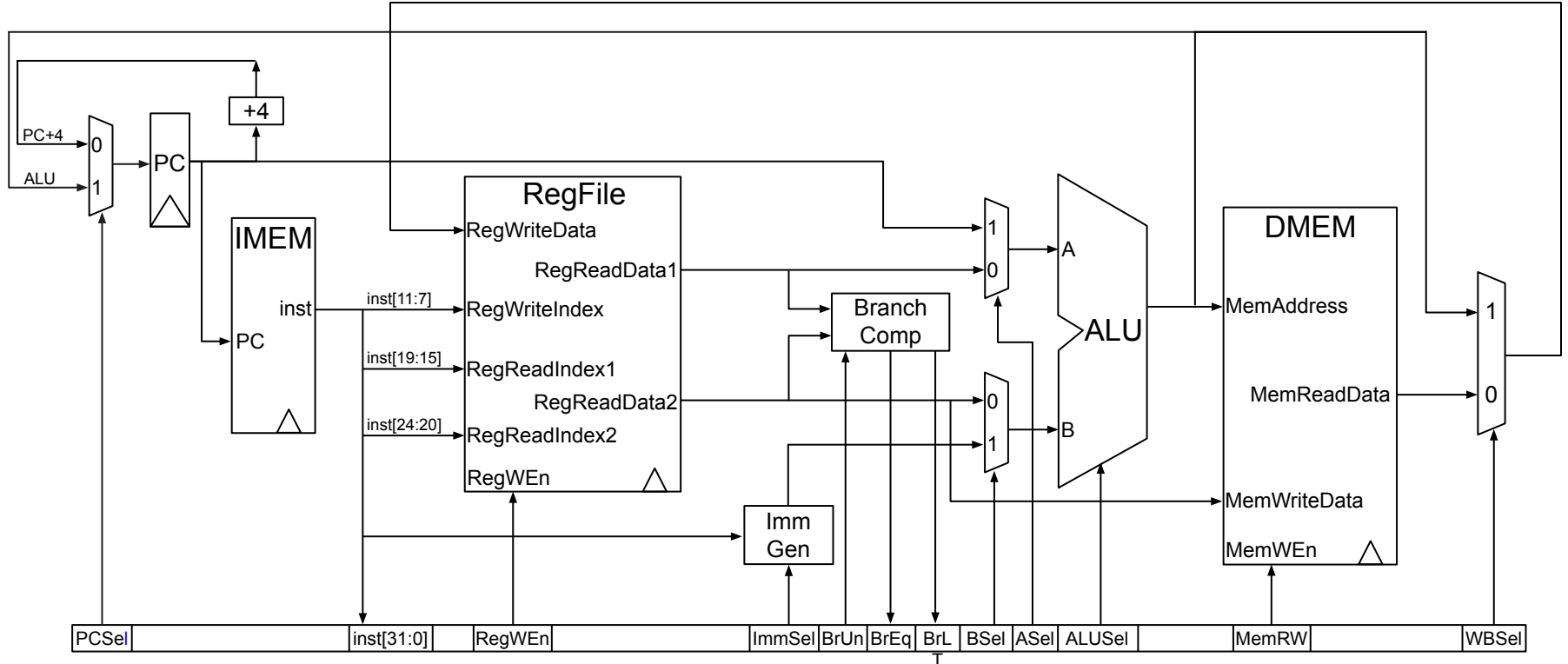
Last time...

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

Datapath So Far



Datapath for jalr

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

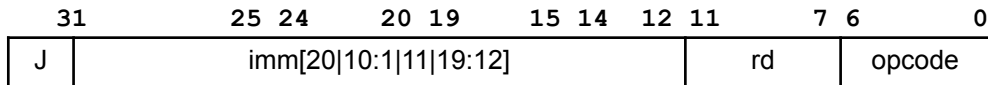
Instruction	Name	Type
slli	Shift Left Logical Immediate	I*
srli	Shift Right Logical Immediate	I*
srai	Shift Right Arithmetic Immediate	I*
slti	Set Less Than Immediate (signed)	I
sltiu	Set Less Than Immediate (Unsigned)	I
lb	Load Byte	I
lbu	Load Byte (Unsigned)	I
lh	Load Half-word	I
lhu	Load Half-word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

jalr instructions

jalr rd rs1 imm	Jump And Link Register	$rd = PC + 4$ $PC = rs1 + imm$
-----------------	------------------------	-----------------------------------

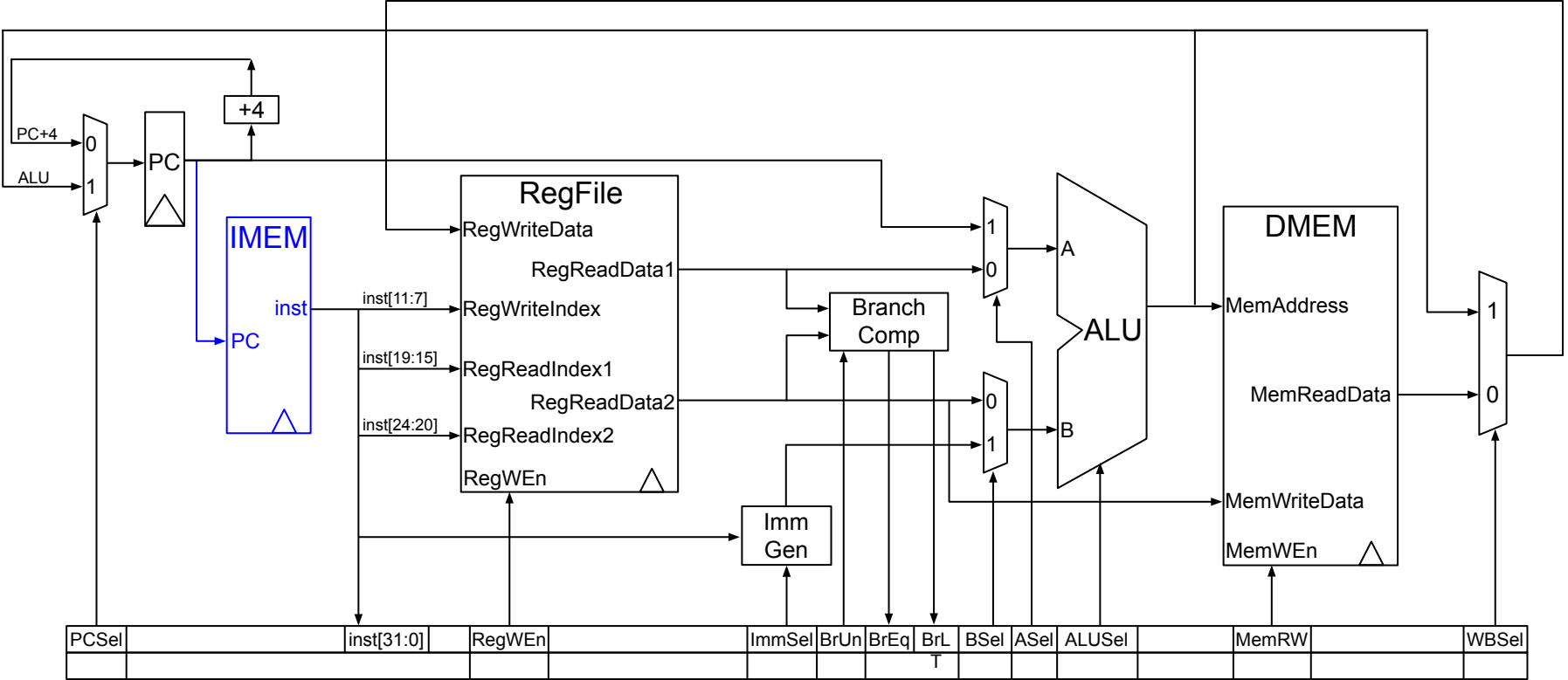
- What is the **jalr** workflow?
 - Basics: read instruction; parse instruction, etc.
 - **rd = PC + 4**: Write PC+4 into the **rd** register
 - **rs1 + imm**: Add the value in register **rs1** to the immediate in the instruction
 - **PC = rs1 + imm**: The result of that addition is the new PC



jalr Stage 1: Instruction Fetch (IF)

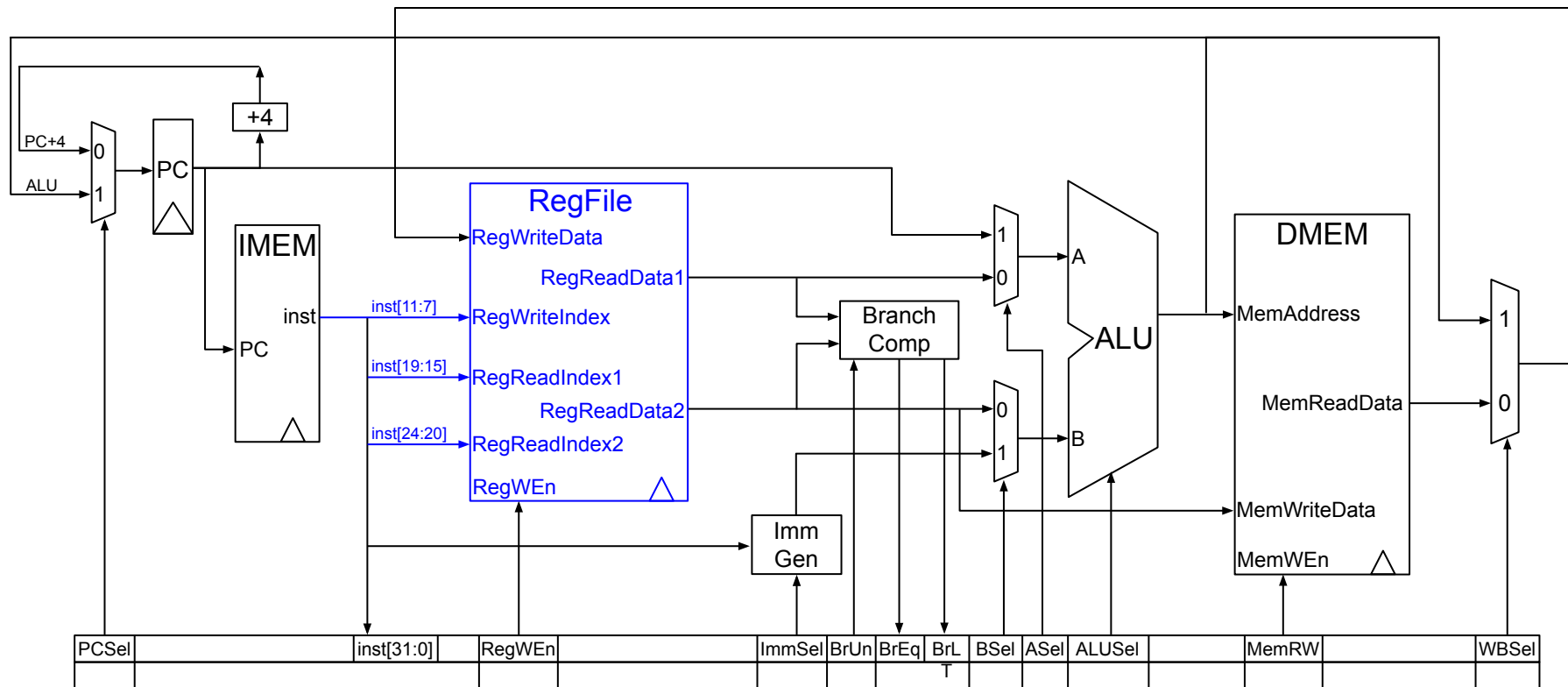
As before, we fetch the instruction from IMEM.

jalr rd rs1 imm	Jump And Link Register	rd = PC + 4 PC = rs1 + imm
-----------------	------------------------	-------------------------------



We read the value of **rs1**.

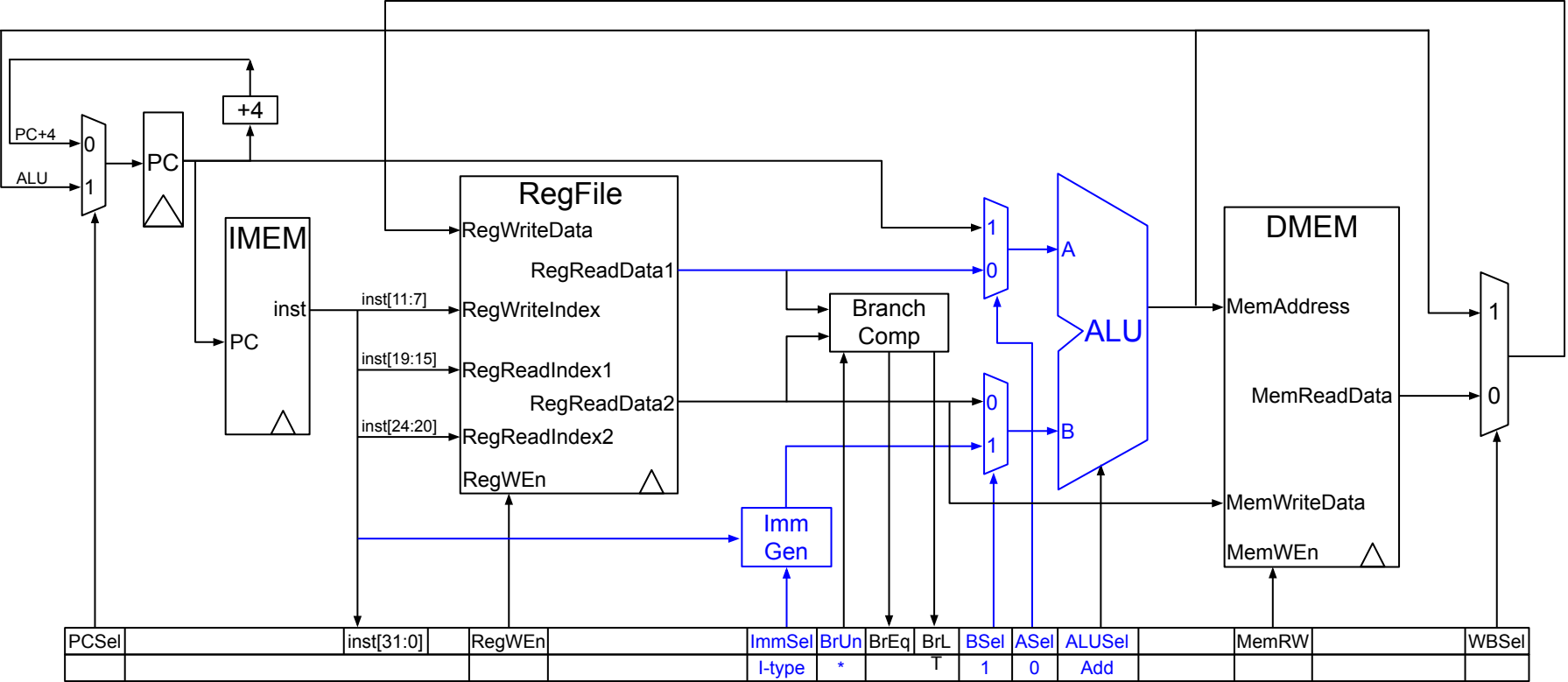
```
rd = PC + 4
PC = rs1 + imm
```



jalr Stage 3: Execute

What should the control signals be?
Note that we don't care about the value of BrUn, so we can write a star * (don't care).

jalr rd rs1 imm	Jump And Link Register	rd = PC + 4 PC = rs1 + imm
-----------------	------------------------	-------------------------------



What should the control signals be?

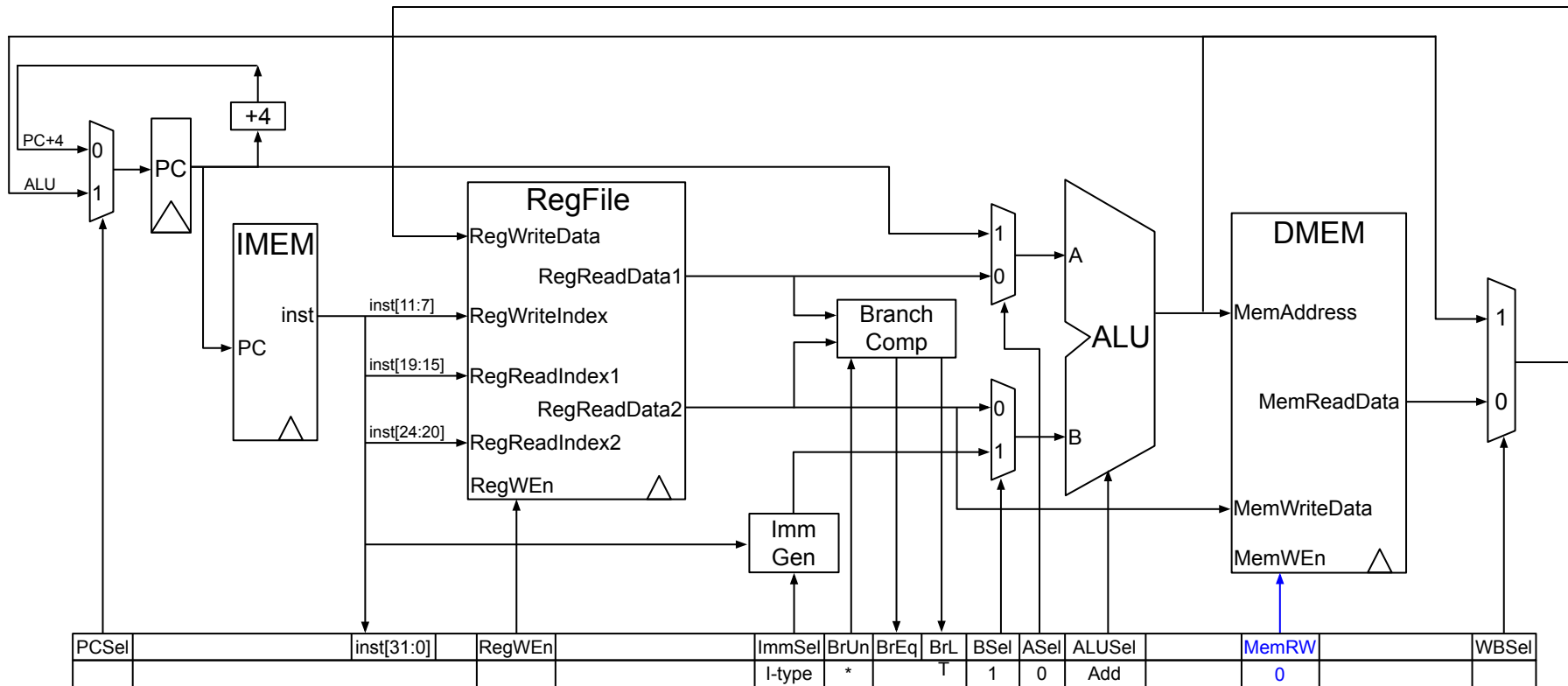
jalr Stage 4: Memory

jalr rd rs1 imm

Jump And Link Register

rd = PC + 4

PC = rs1 + imm

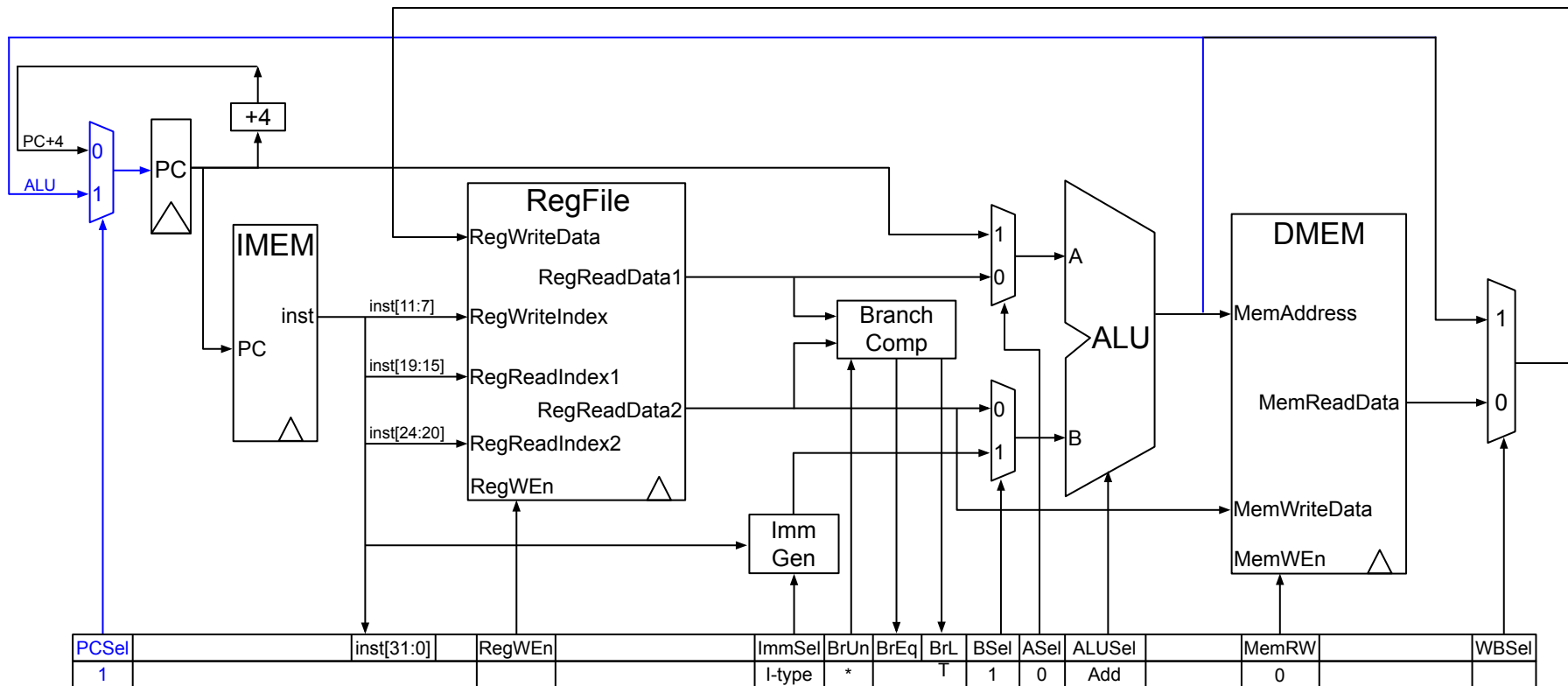


jalr Stage 5: Register Write

We just used the ALU to calculate $PC + imm$. We want this in PC, so set $PCSel = 1$.

```
jalr    rd, rs1, imm
```

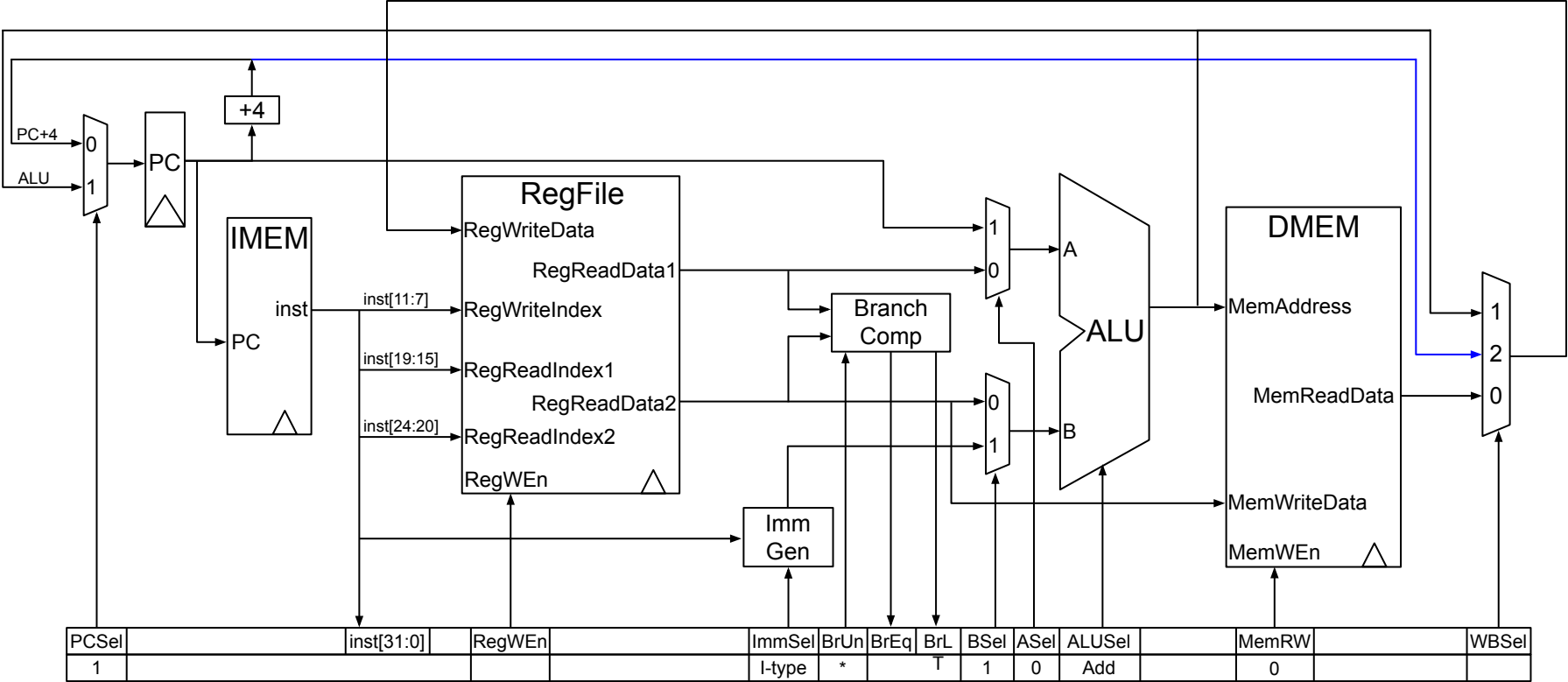
Jump And Link Register

$$rd = PC + 4$$
$$PC = rs1 + imm$$


jalr Stage 5: Register Write

We want to write PC+4 into register **rd**. We already have this value, but we need to add wires to write it back.

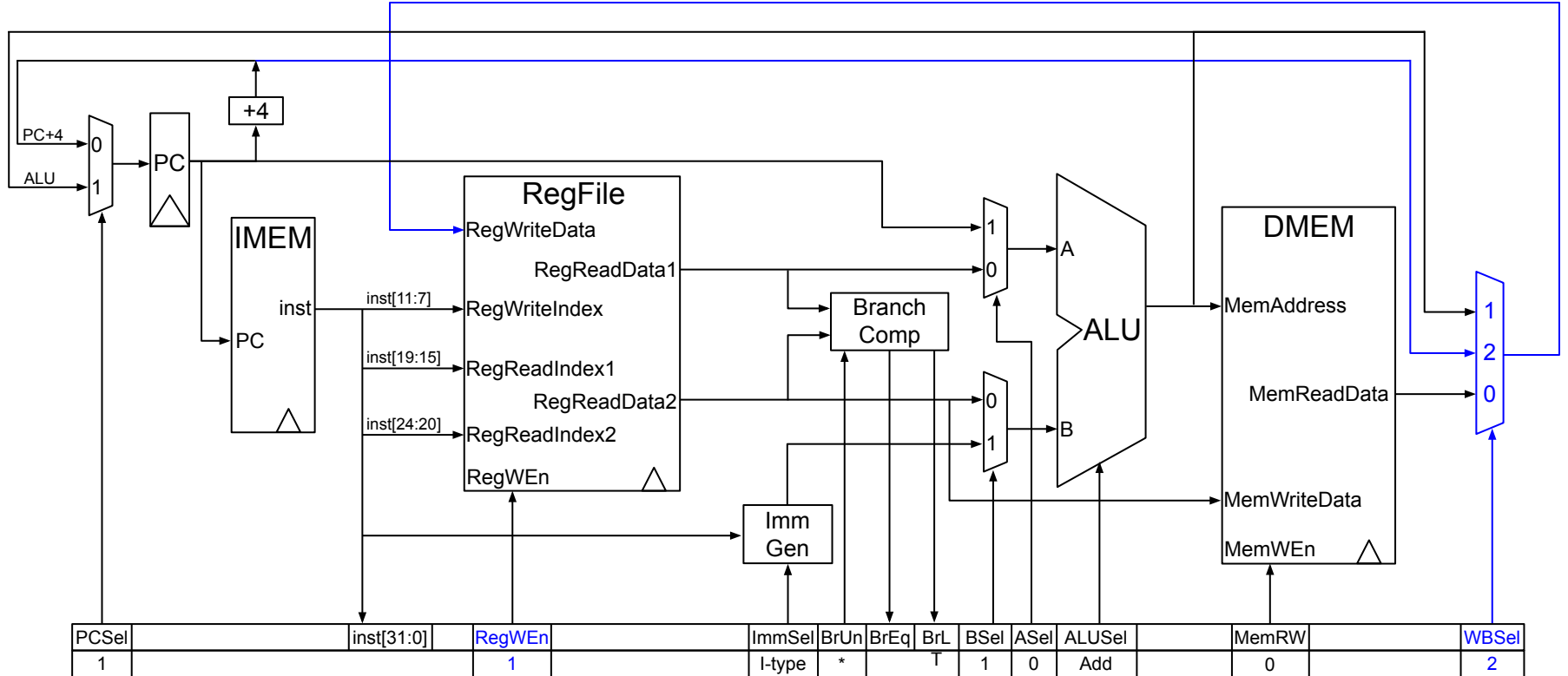
jalr rd rs1 imm	Jump And Link Register	rd = PC + 4 PC = rs1 + imm
-----------------	------------------------	-------------------------------



Don't forget control signals!

jalr Stage 5: Register Write

jalr rd rs1 imm	Jump And Link Register	rd = PC + 4 PC = rs1 + imm
-----------------	------------------------	-------------------------------



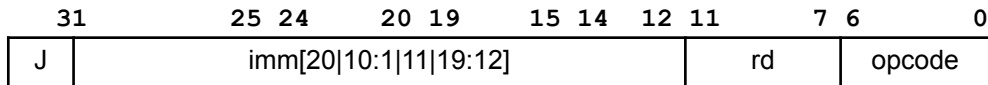
Datapath for jal

jal instructions

jal rd label	Jump And Link	rd = PC + 4 PC = PC + offset
-----------------	---------------	---------------------------------

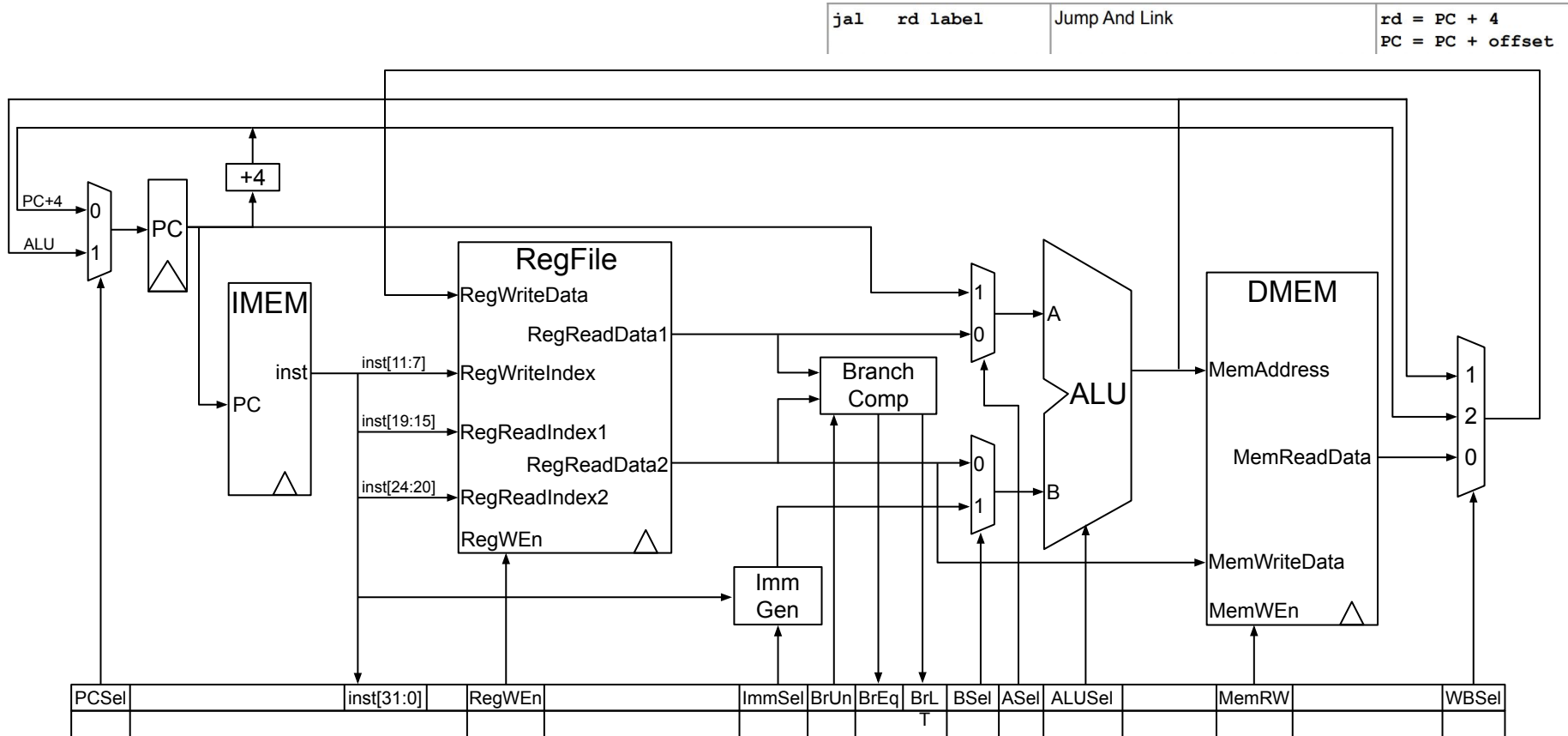
- What is the **jal** workflow?

- Basics: read instruction; parse instruction, etc.
- rd = PC + 4**: Write PC+4 into the **rd** register
- PC + imm**: Add the value in register **rs1** to the immediate in the instruction
- PC = PC + imm**: The result of that addition is the new PC

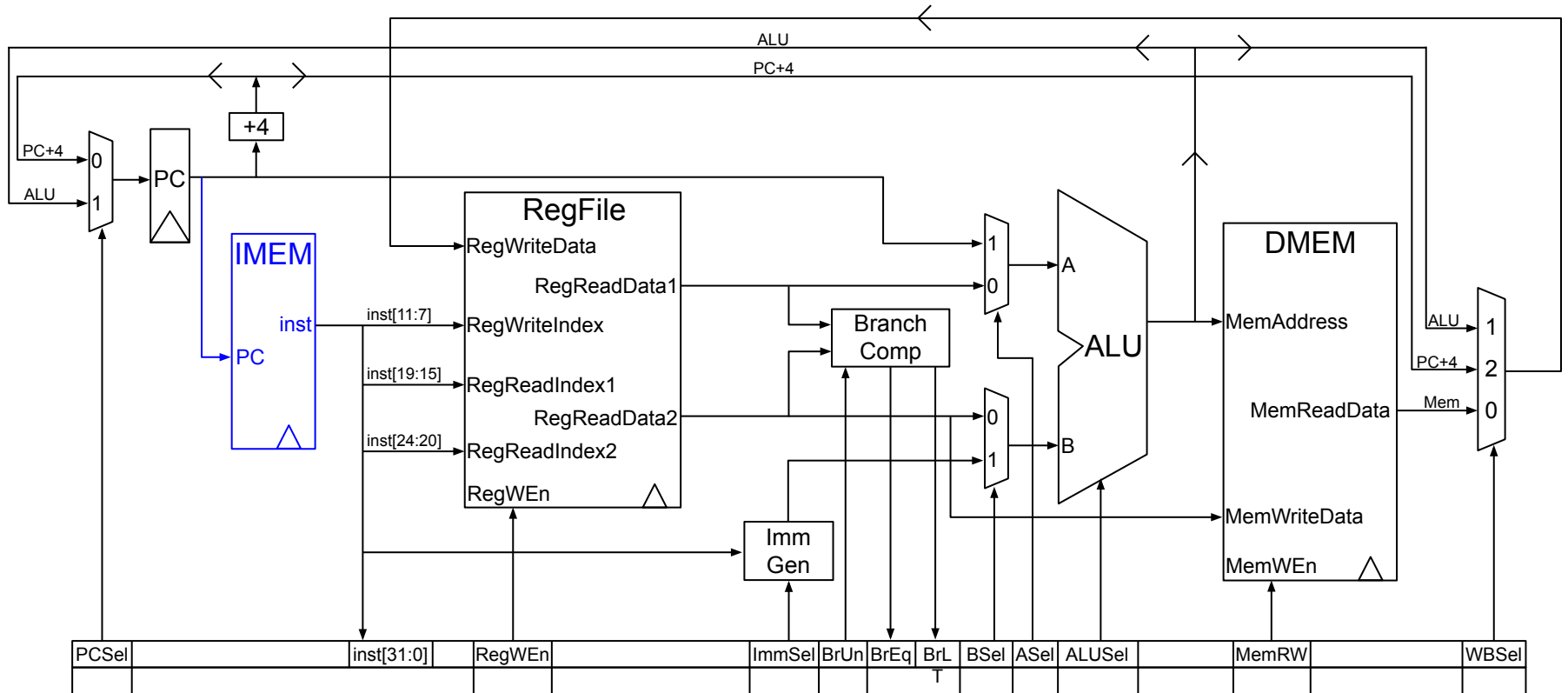


Do we need new hardware?

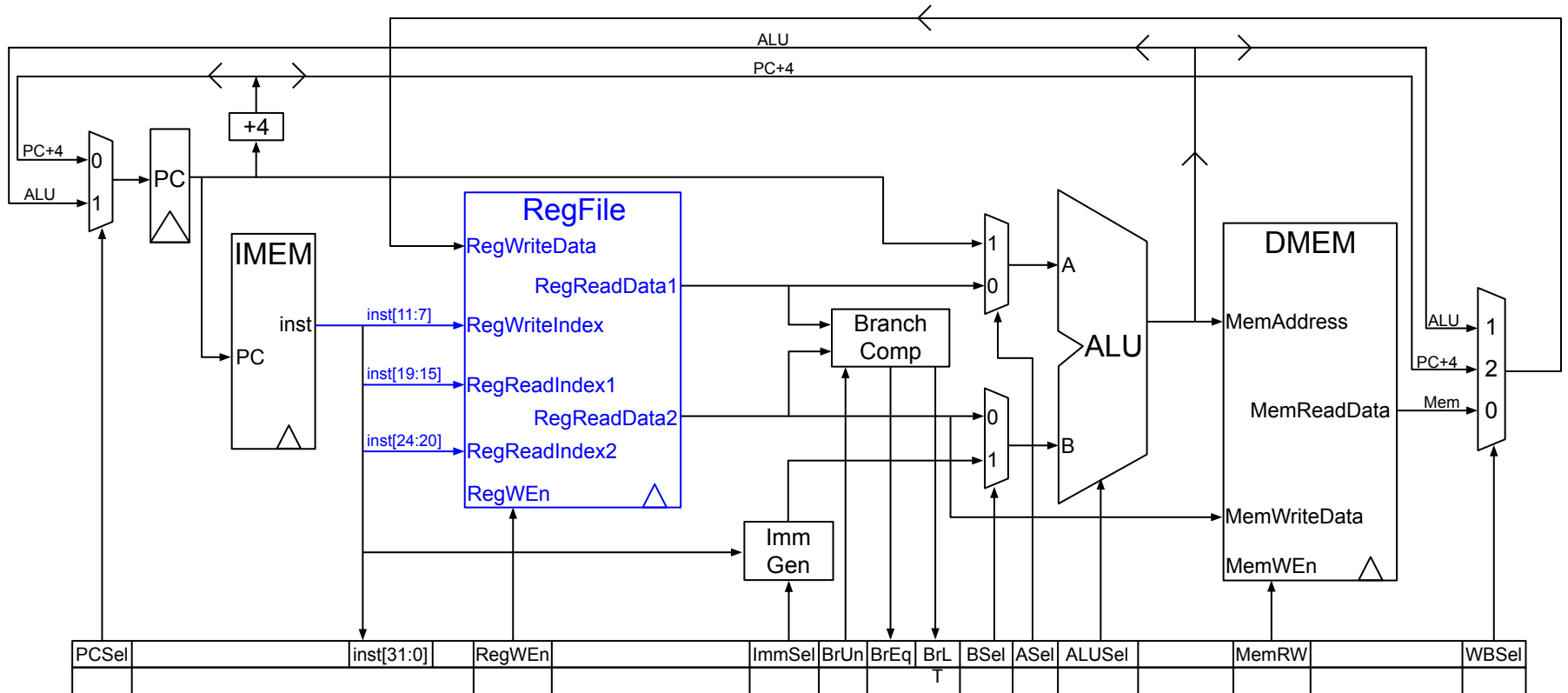
Discussion: What gets changed?



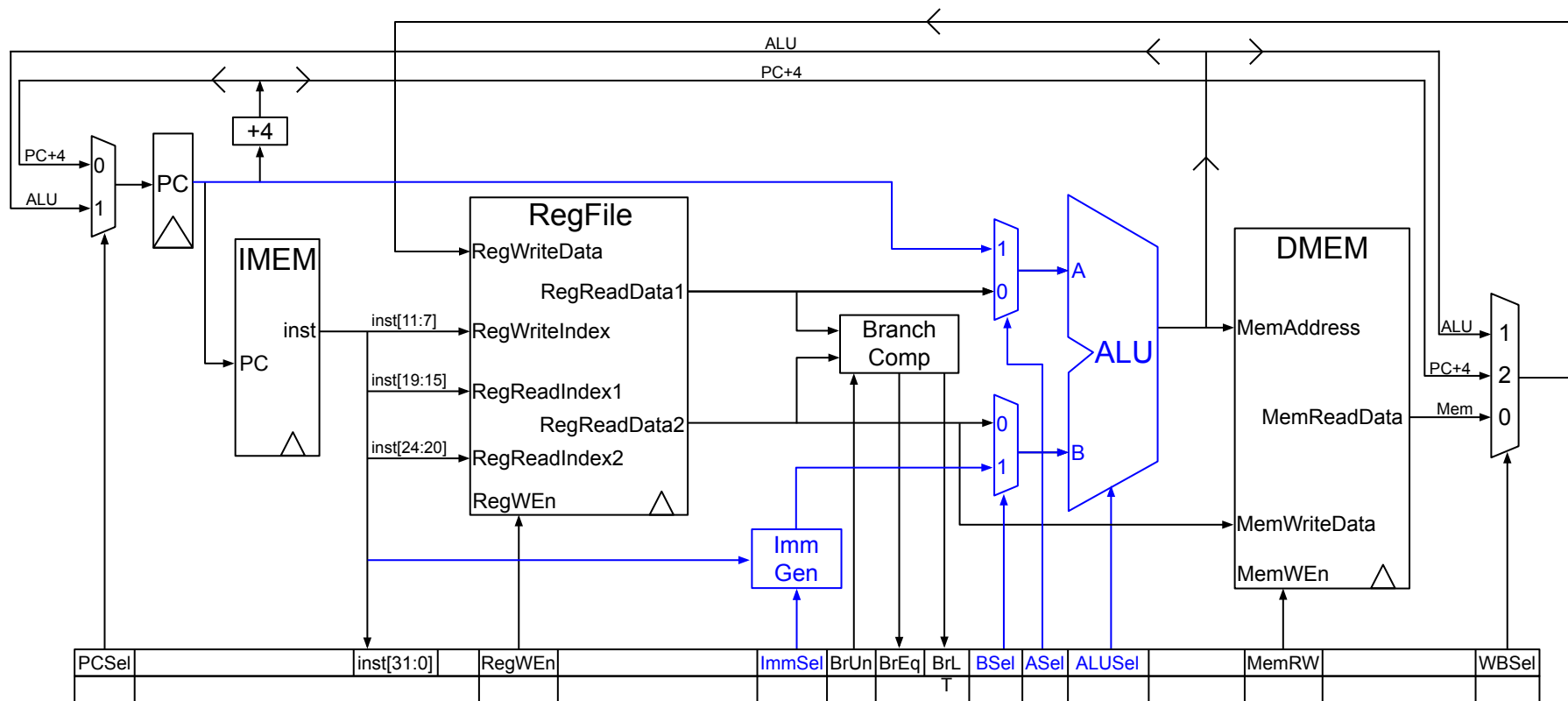
Your turn to fill in all the control signals!



Your turn to fill in all the control signals!

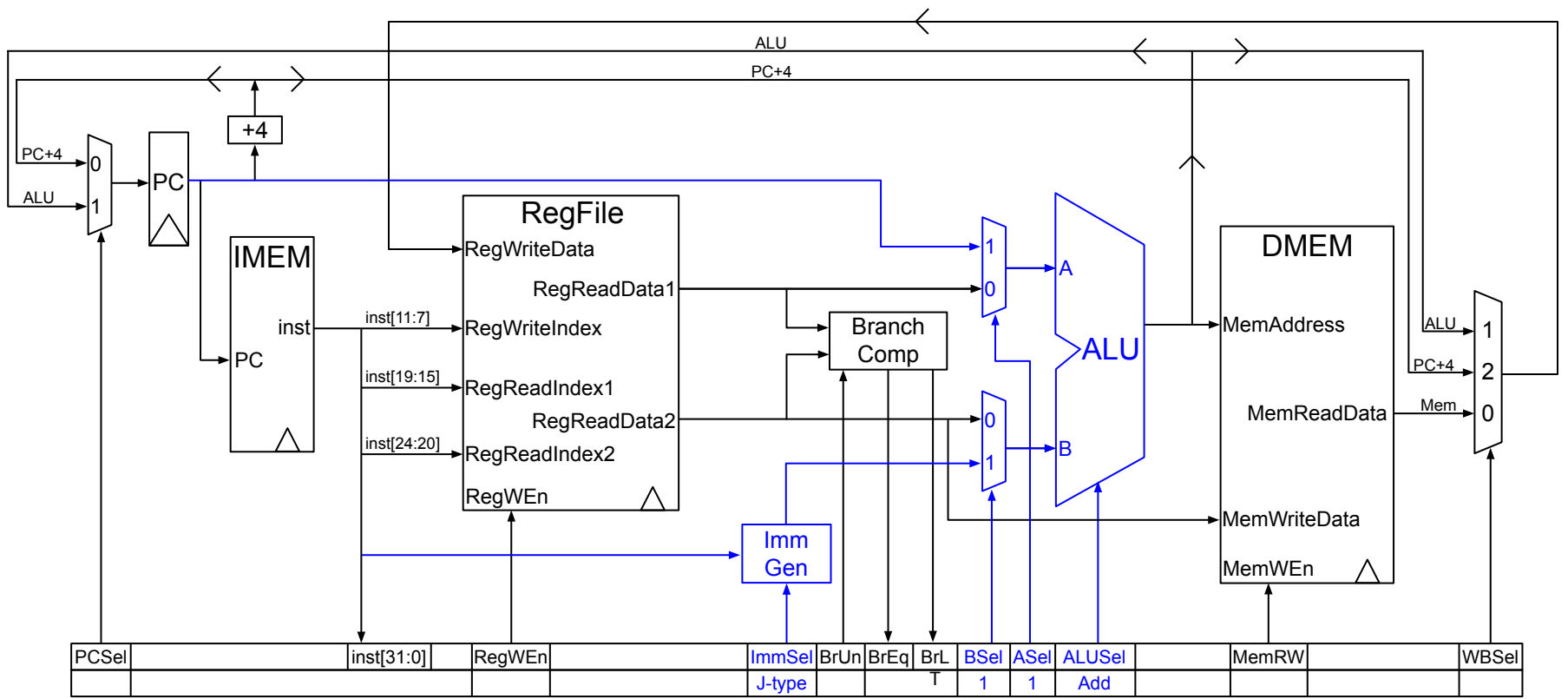


Your turn to fill in all the control signals!



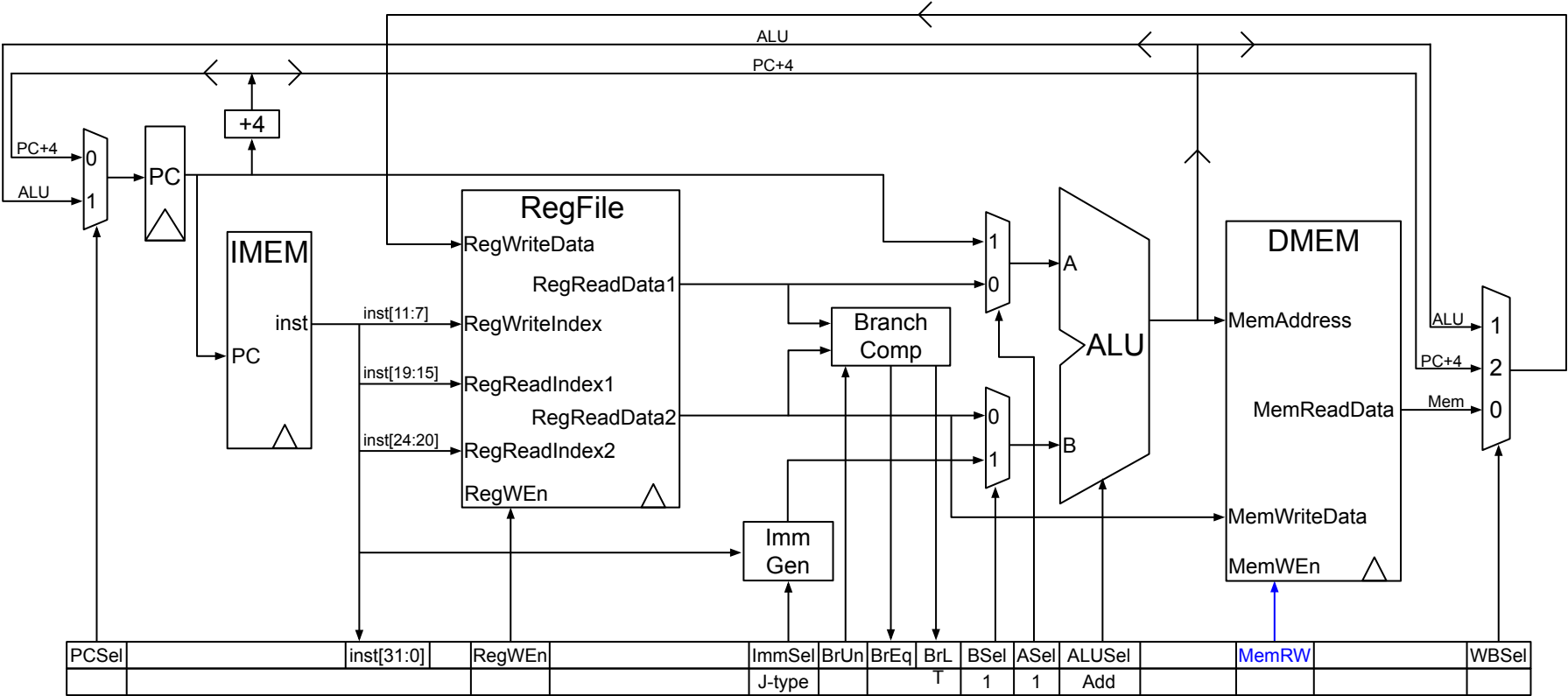
jal Stage 3: Execute

Your turn to fill in all the control signals!



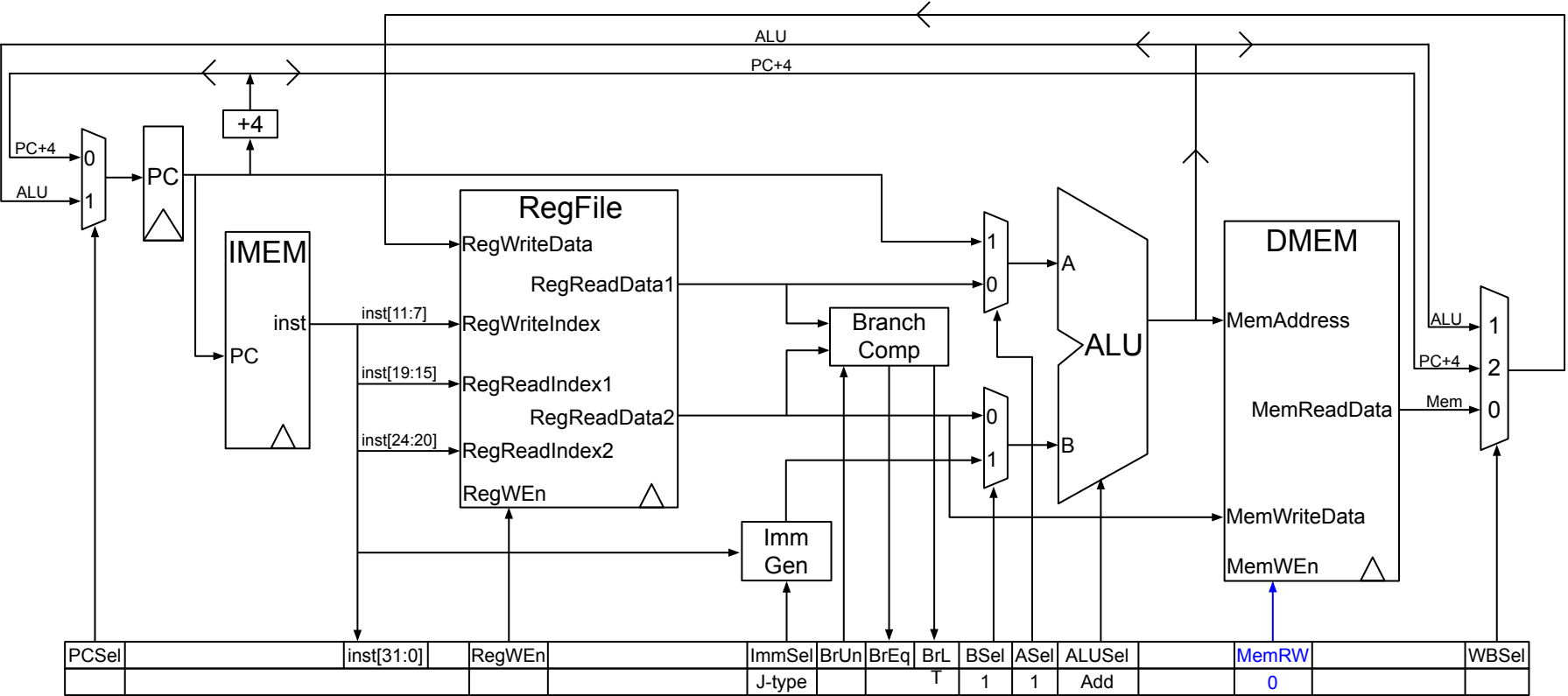
jal Stage 4: Memory

Your turn to fill in all the control signals!



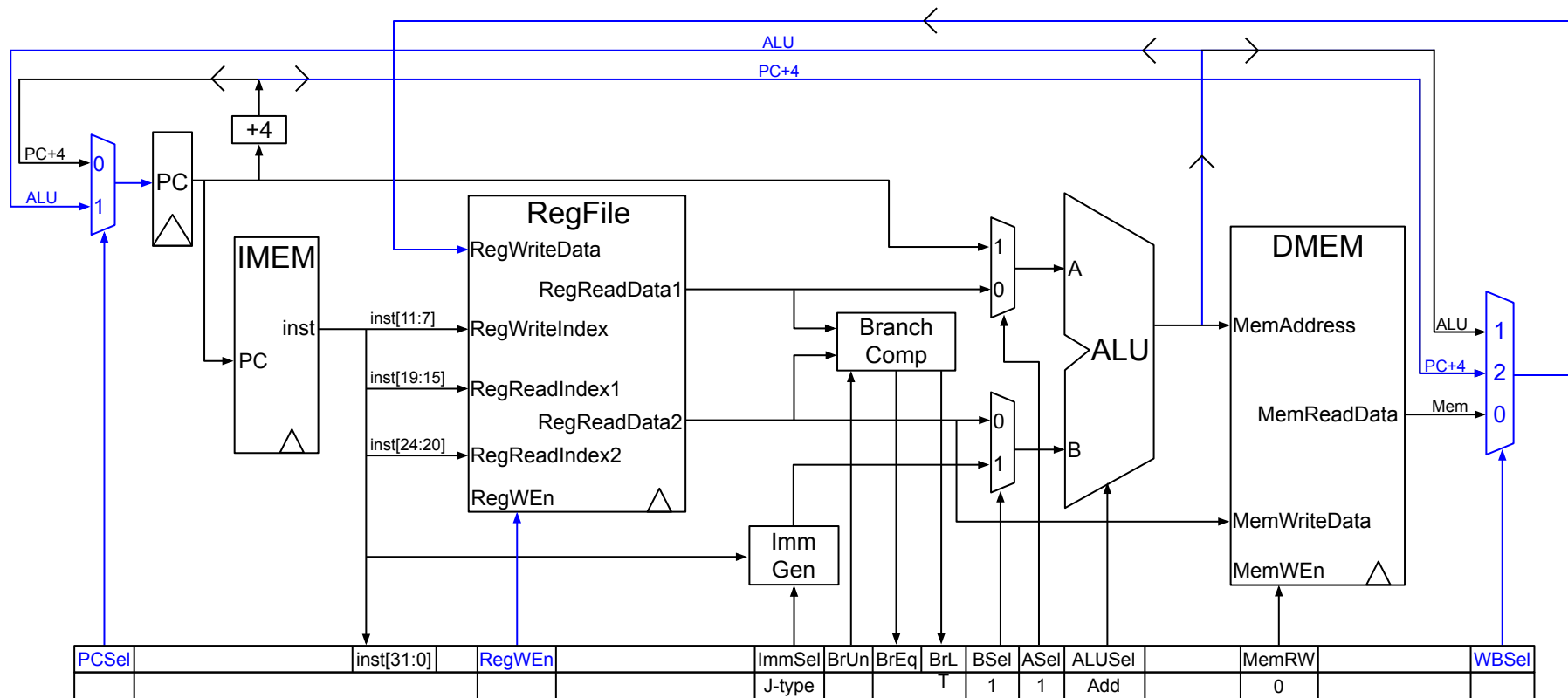
jal Stage 4: Memory

Your turn to fill in all the control signals!



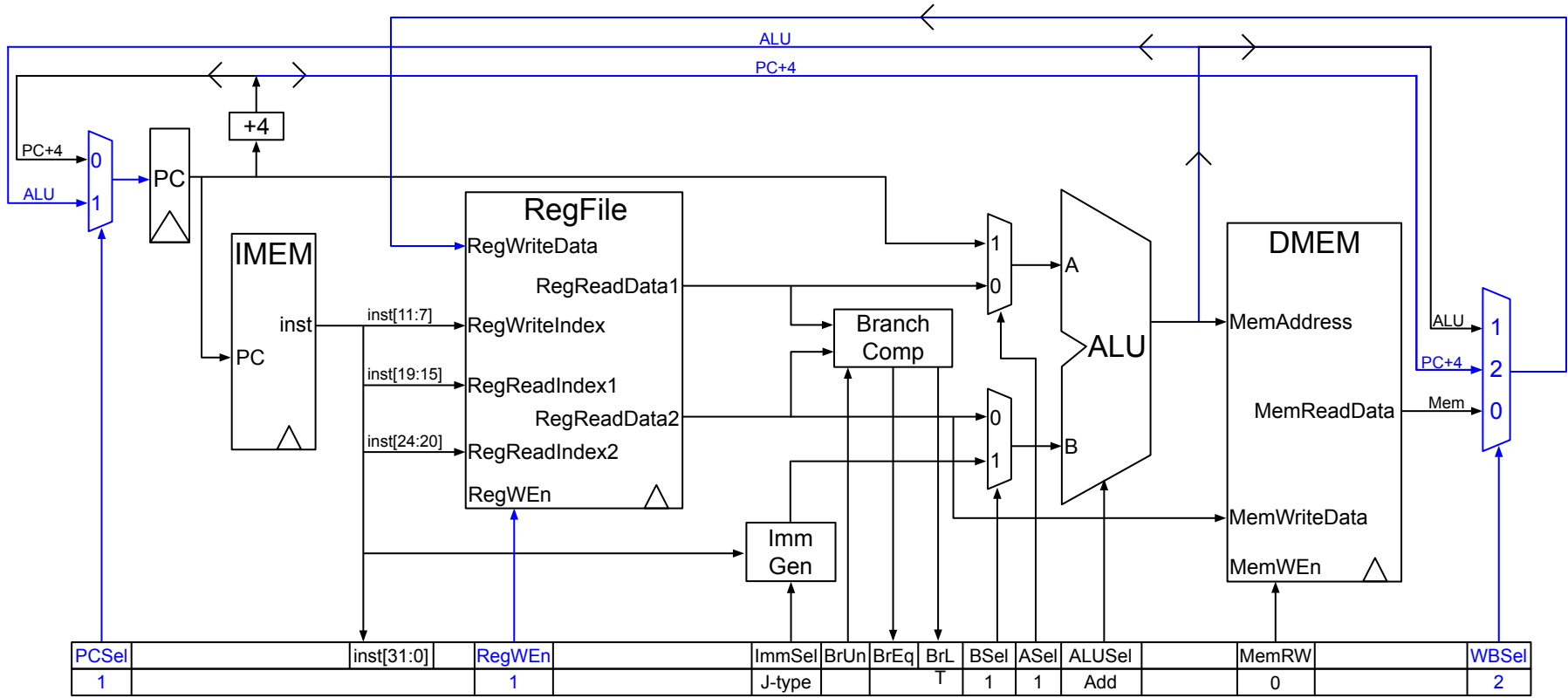
jal Stage 5: Register Write

Your turn to fill in all the control signals!



jal Stage 5: Register Write

Your turn to fill in all the control signals!



Datapath for U-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

U-type instructions

- What is the workflow for lui?

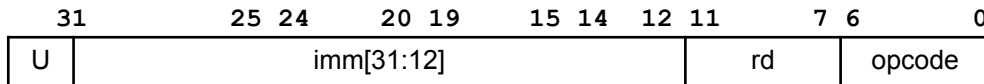
- $imm = immu \ll 12$: shift immediate by 12
- $rd = imm$: the result is the value we put in **rd**

lui rd immu	Load Upper Immediate	$imm = immu \ll 12$ $rd = imm$
----------------	----------------------	-----------------------------------

- What is the workflow for auipc?

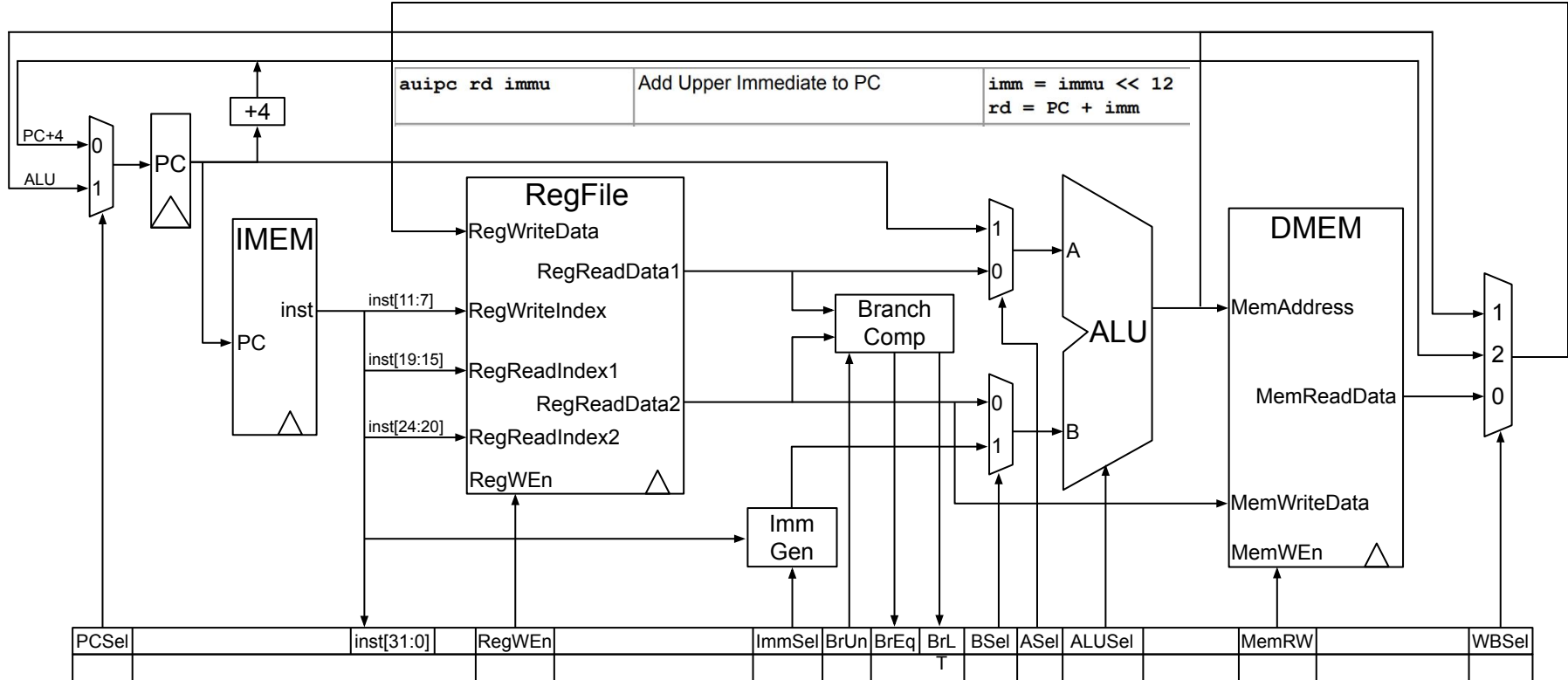
- $imm = immu \ll 12$: shift immediate by 12
- $PC + imm$: add the new immediate to PC
- $rd = PC + imm$: the result is the value we put in register **rd**

auipc rd immu	Add Upper Immediate to PC	$imm = immu \ll 12$ $rd = PC + imm$
---------------	---------------------------	--



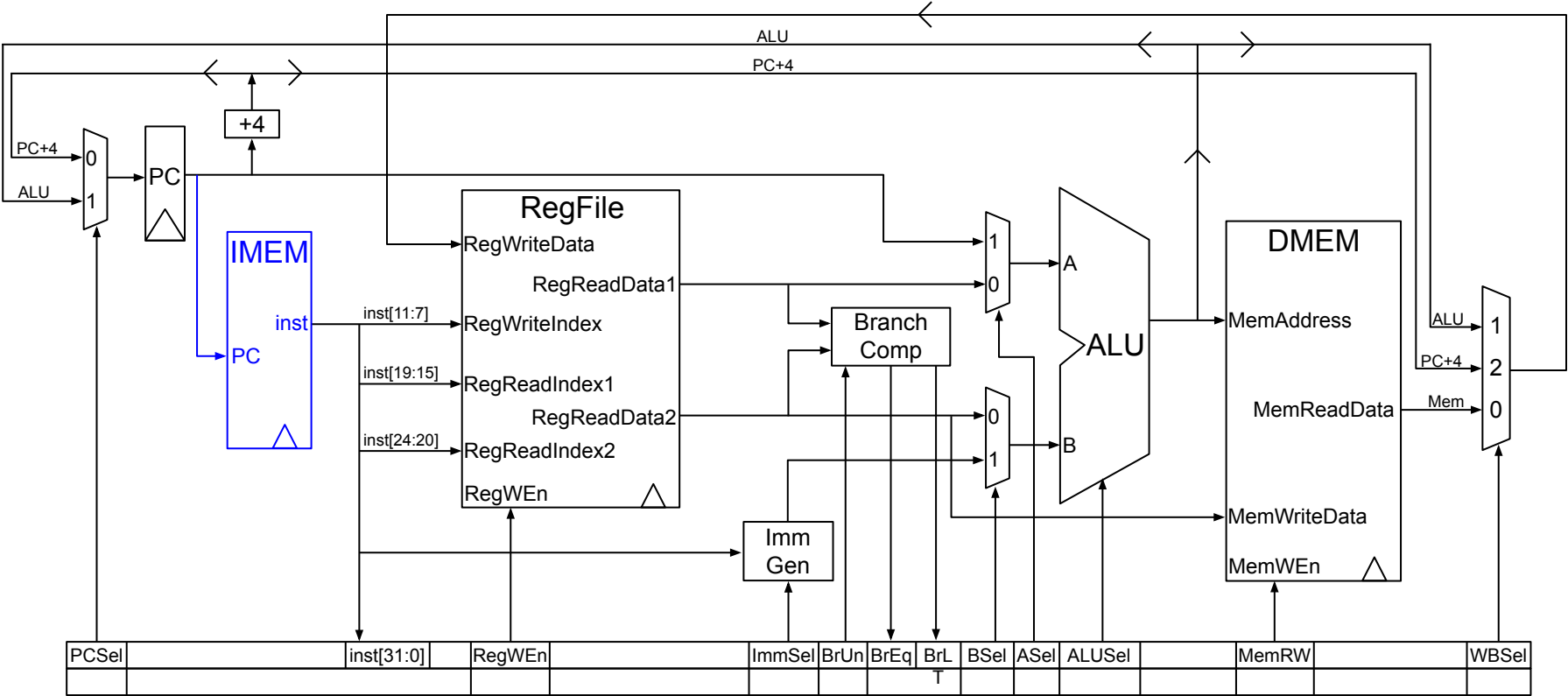
Discussion: What gets changed?

How to we implement the control signals?
Where do we implement the shift, where do we implement the addition?

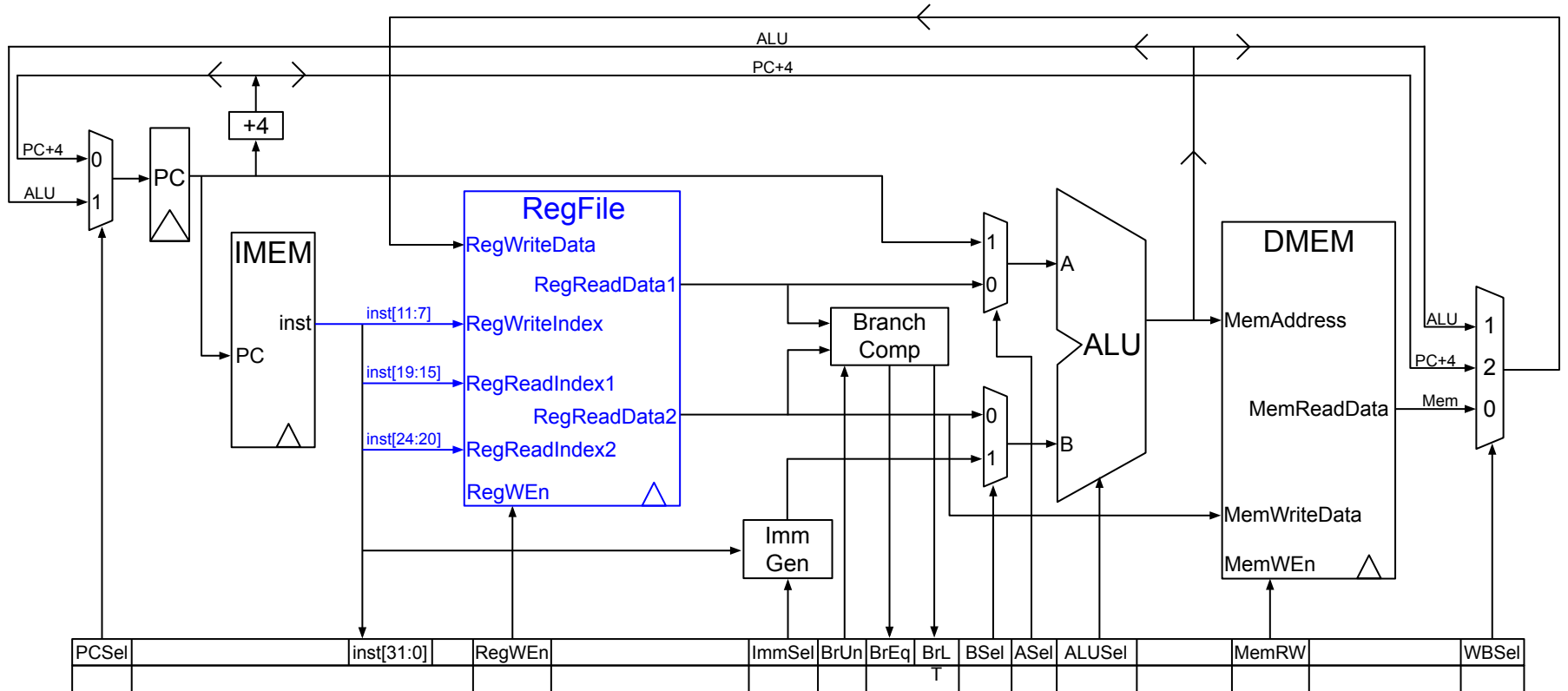


U-type Stage 1: Instruction Fetch (IF)

Your turn to fill in all the control signals!

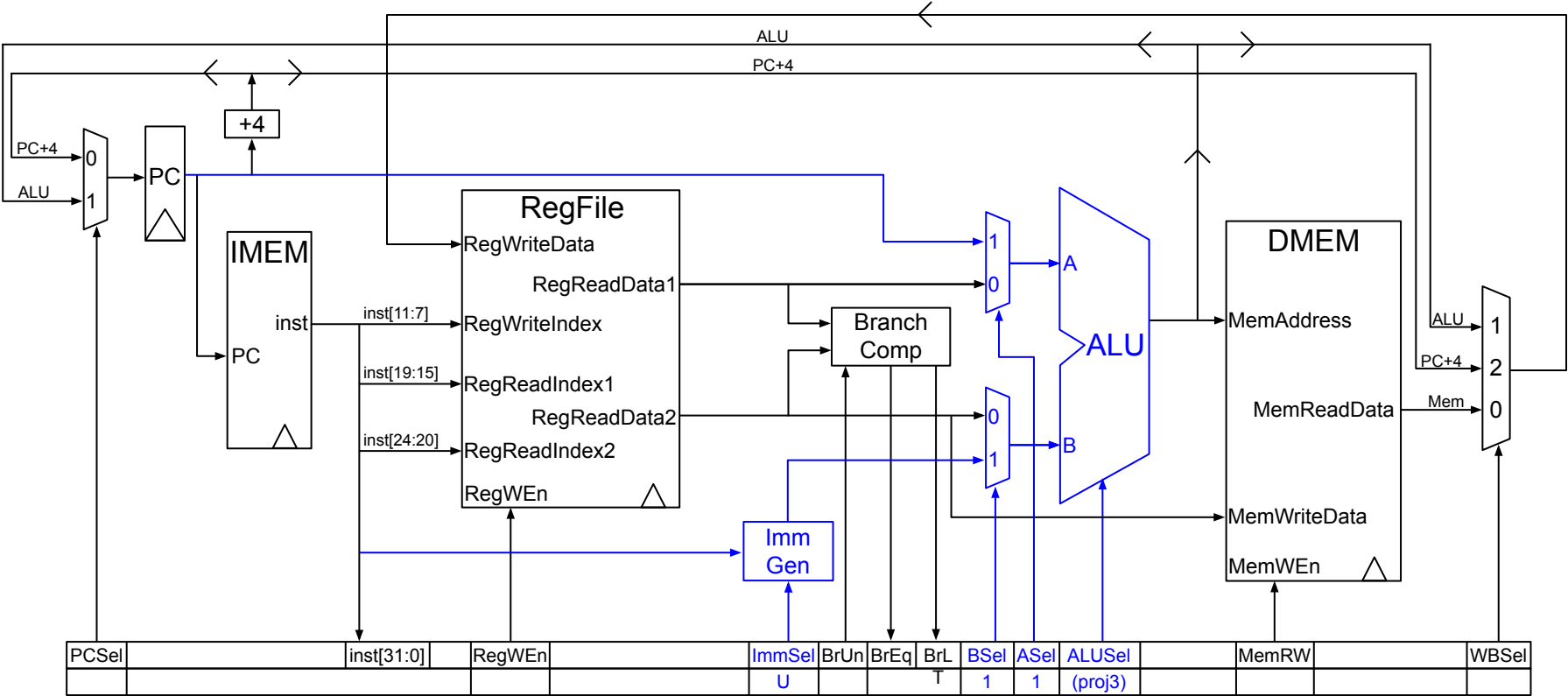


Your turn to fill in all the control signals!



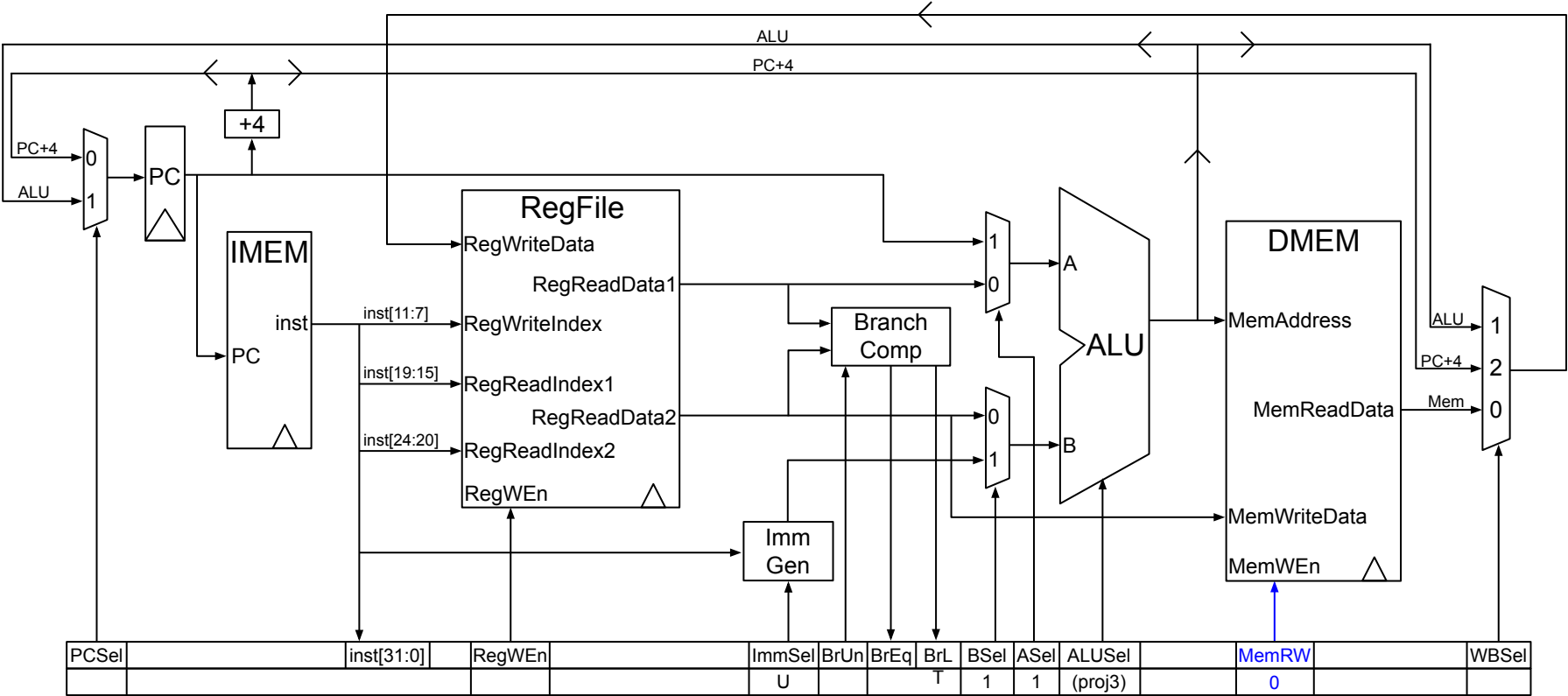
U-type Stage 3: Execute

Your turn to fill in all the control signals!



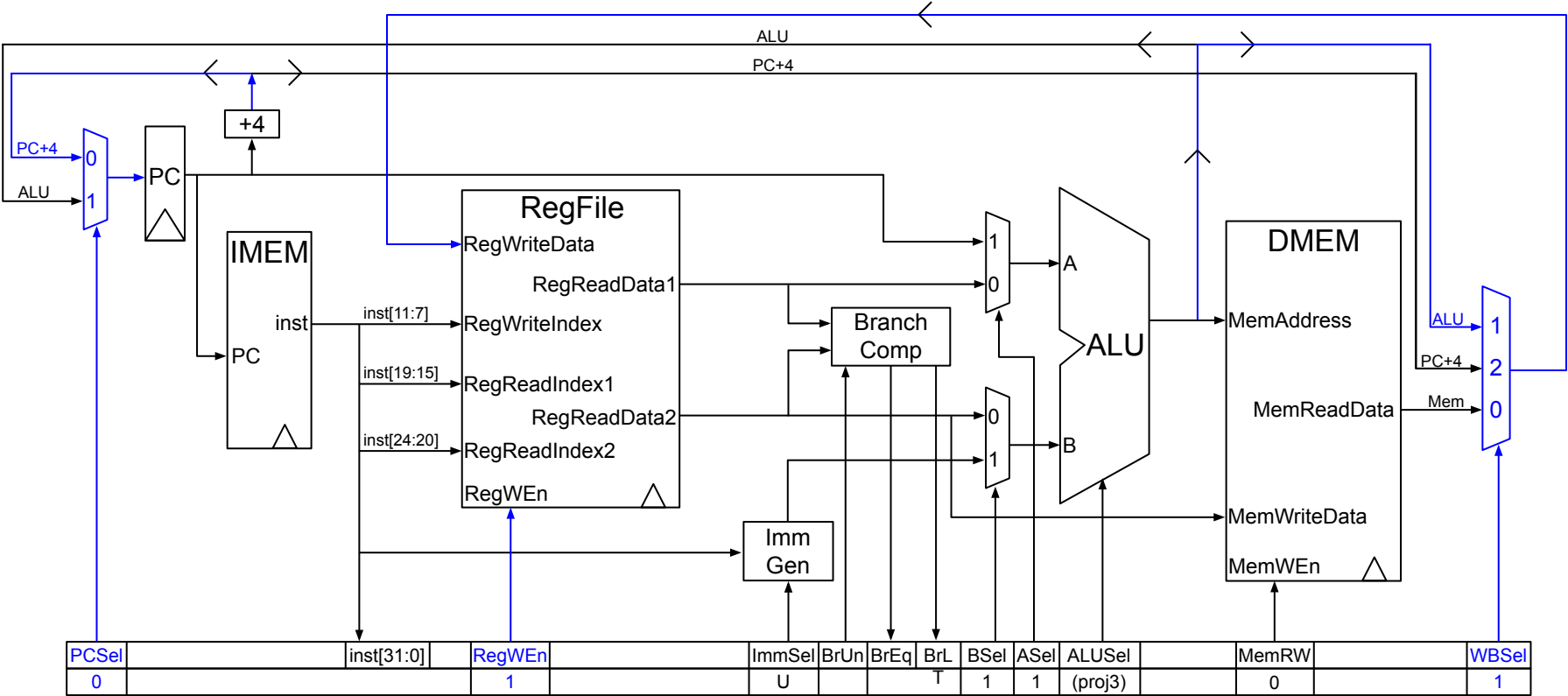
U-type Stage 4: Memory

Your turn to fill in all the control signals!



U-type Stage 5: Register Write

Your turn to fill in all the control signals!



Timing Instructions

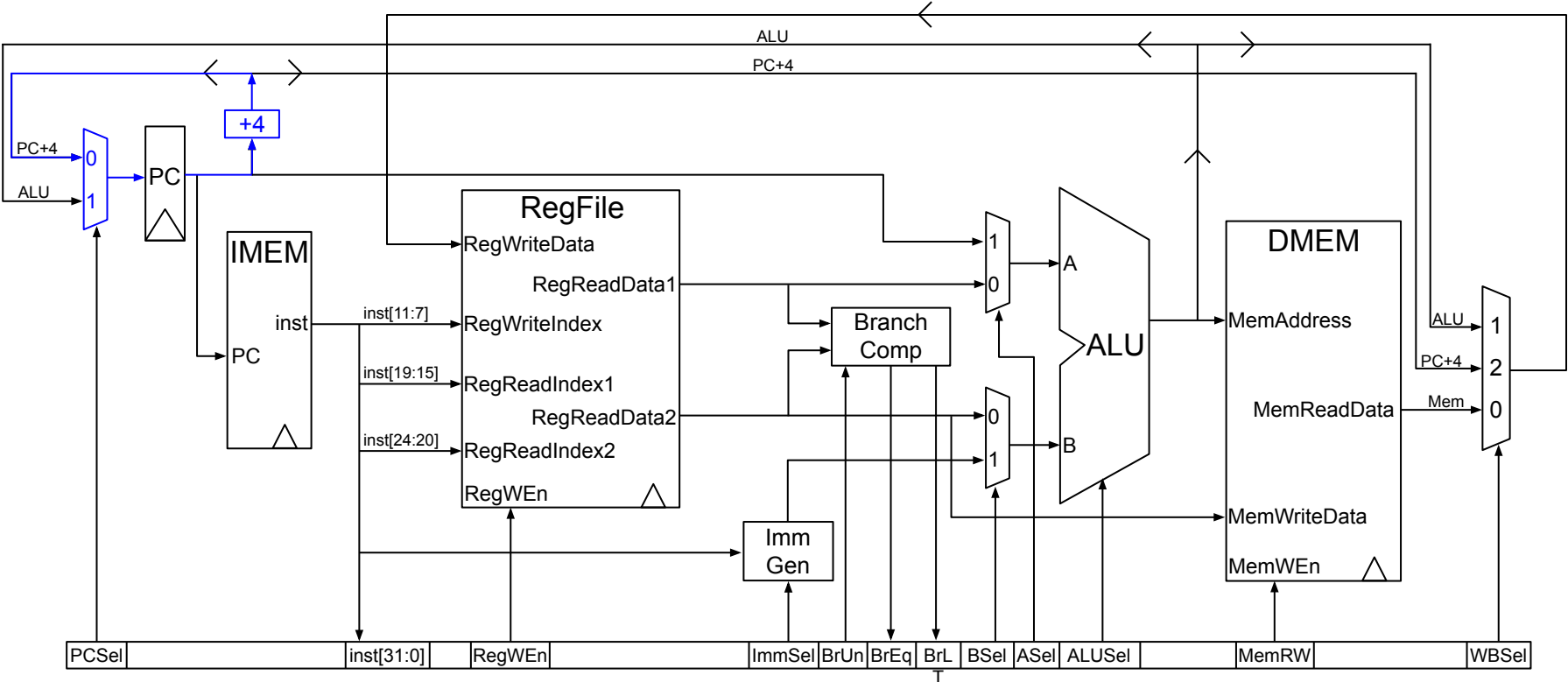
Timing Instructions

- How many instructions can we execute per second?
 - Depends on clock period (one instruction per clock cycle)
 - Clock period depends on critical path
 - Critical path in the datapath: longest path between two registers
 - Reminder that critical path is calculated from state element to state element!
 - In the datapath, that means the PC register, RegFile write, and DMEM write!

Timing 1w

Path #1 from register (PC) to register (PC).

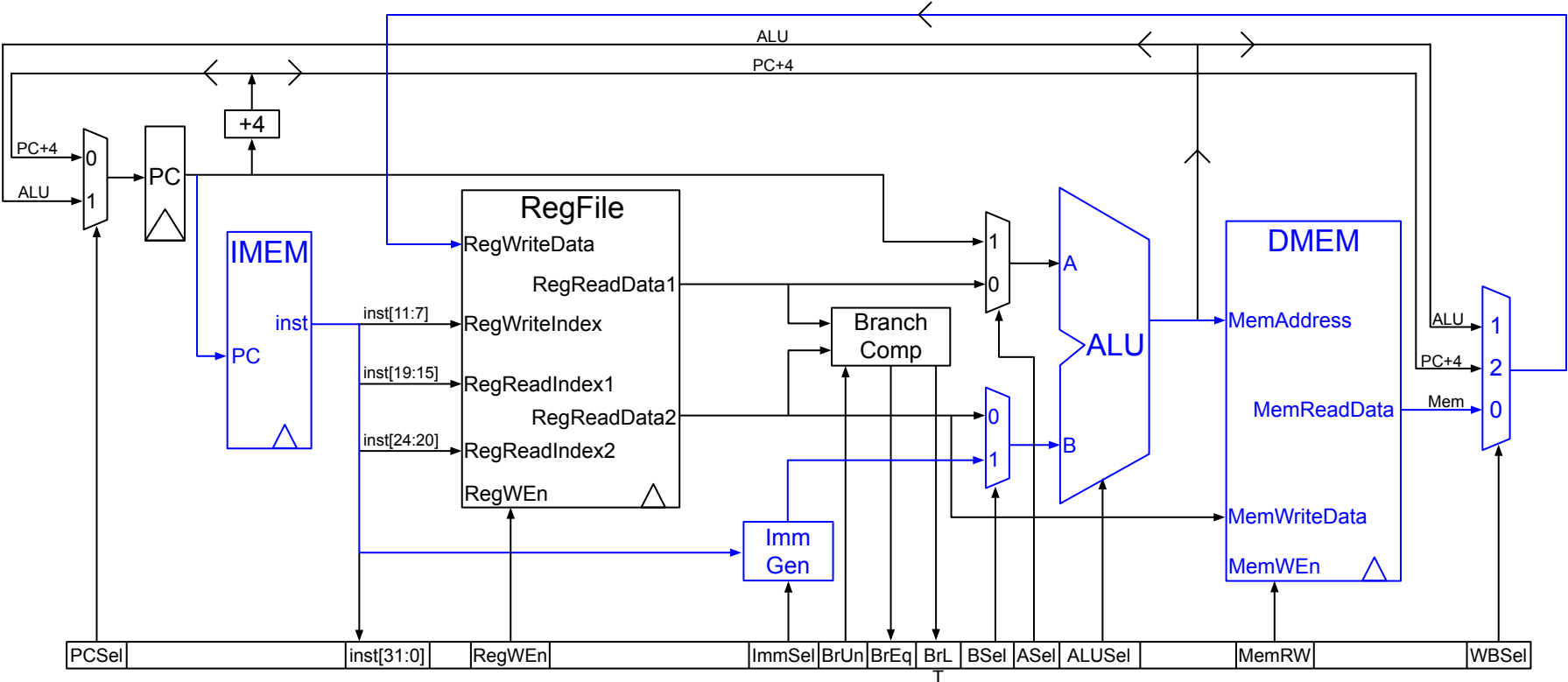
Delays: adder + mux



Timing 1w

Path #2 from register (PC) to register (RegFile).

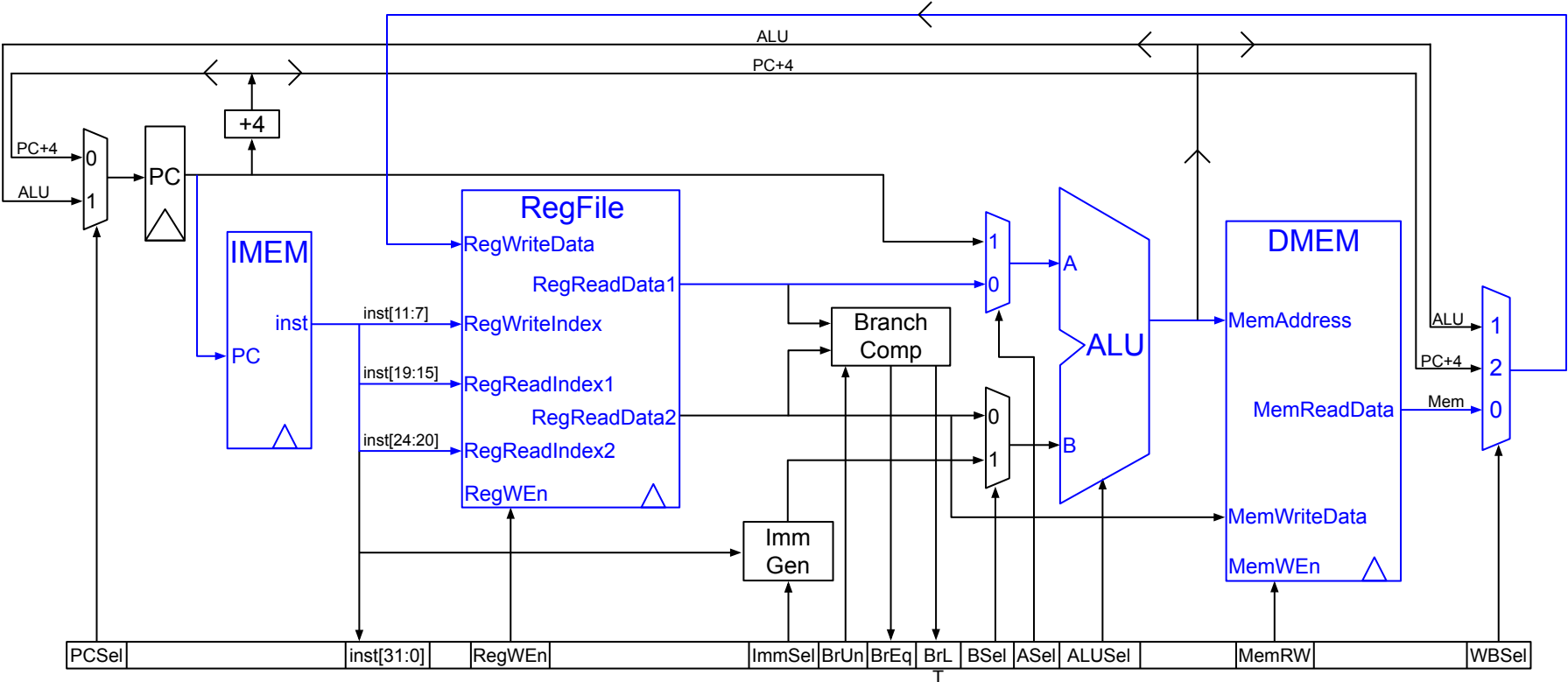
Delays: IMEM + ImmGen + mux + ALU + DMEM + mux



Timing 1w

Path #3 from register (PC) to register (RegFile).

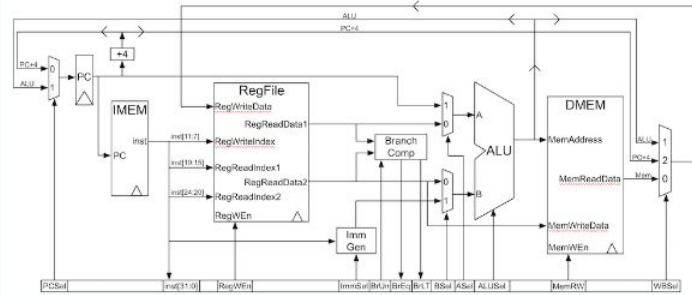
Delays: IMEM + RegFile + mux + ALU + DMEM + mux



Timing 1w

- Three paths between registers
 - Path #1 delay: adder + mux
 - Path #2 delay: IMEM + ImmGen + mux + ALU + DMEM + mux
 - Path #3 delay: IMEM + RegFile + mux + ALU + DMEM + mux
- Minimum clock period: clk-to-q time + longest delay + setup time
 - In practice, this is around 800ps
 - Maximum clock frequency: $1/800\text{ps} = 1.25 \text{ GHz} = 1.25 \text{ billion instructions per second}$

What is the critical path delay of addi?



$t_{clk-q} + t_{Add} + t_{MEM} + t_{Reg} + t_{BMux} + t_{ALU} + t_{DMEM} + t_{Mux} + t_{Setup}$

0%

$t_{clk-q} + t_{MEM} + \max(t_{Reg}, t_{Imm}) + t_{ALU} + 3t_{Mux} + t_{Setup}$

0%

$t_{clk-q} + t_{MEM} + \max(t_{Reg}, t_{Imm}) + t_{ALU} + 3t_{Mux} + t_{DMEM}$

0%

None of the above

0%



Summary Datapath; What's Next

CS 61C Hardware Roadmap

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

Processor (CPU)

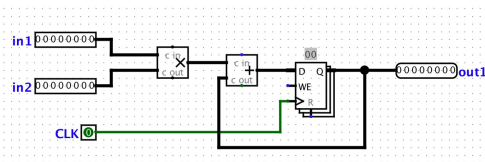
- Datapath
- **Control Logic**
- **Pipelining**

Synchronous Digital Systems

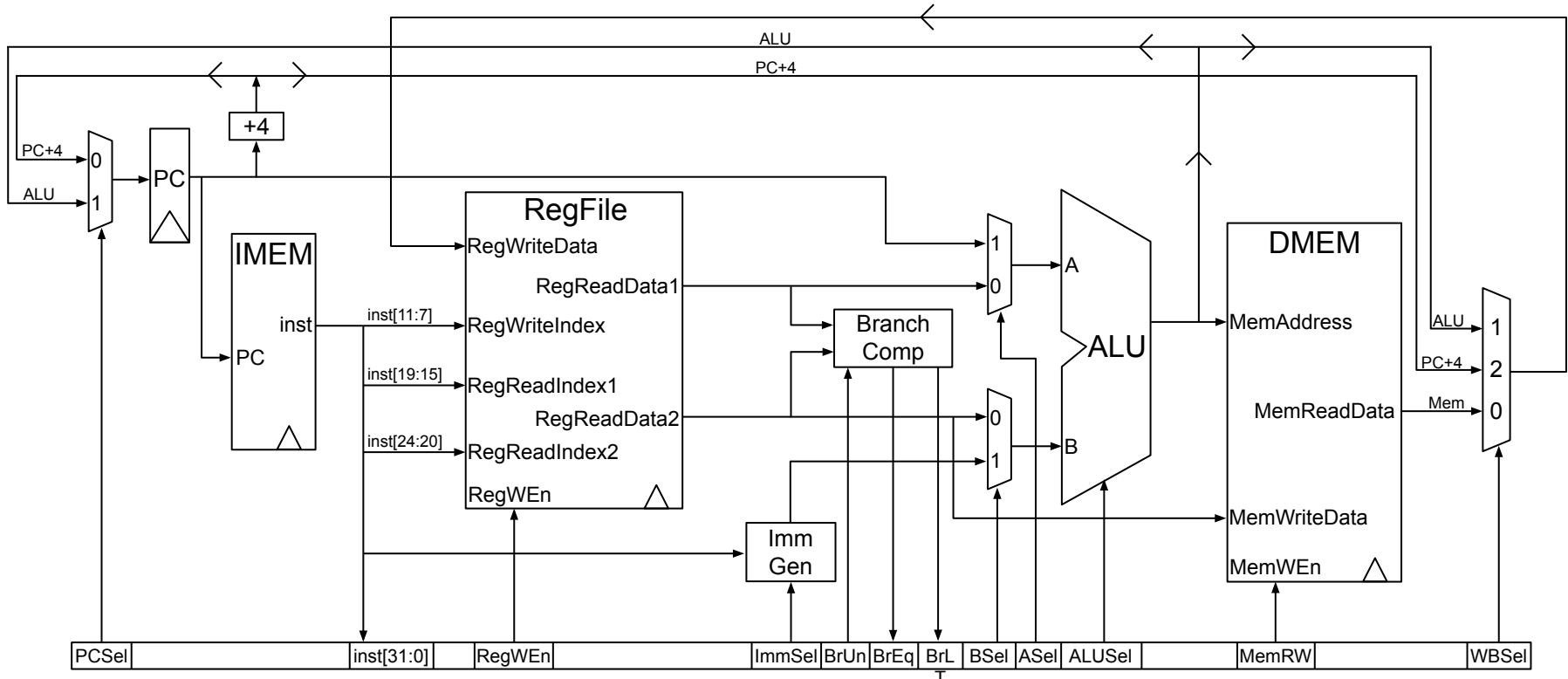
- Combinatorial Logic
- Sequential Logic

Transistors

We just finished building a circuit that can run every base RISC-V instruction!



Single-Cycle RISC-V Datapath



CS 61C Hardware Roadmap



Processor (CPU)

- Datapath
- **Control Logic**
- Pipelining

Synchronous Digital Systems

- Combinatorial Logic
- Sequential Logic

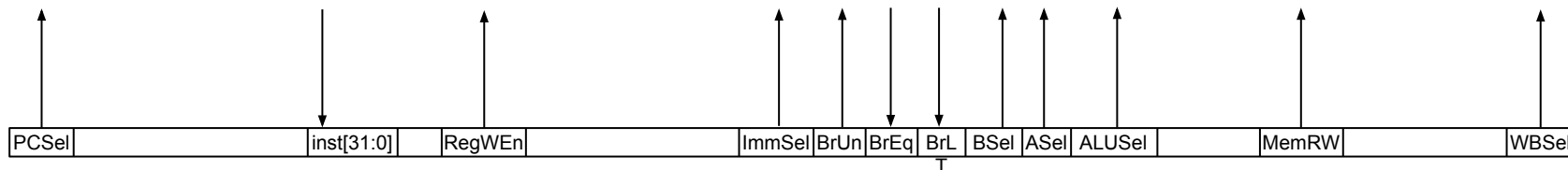
Transistors

Next, we'll design the control logic subcircuit.

Control Logic Signals

Review: Control Logic

- Inputs
 - 32-bit instruction
 - Results of branch comparator (BrLt and BrEq)
- 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?



Control Signal: ImmSel

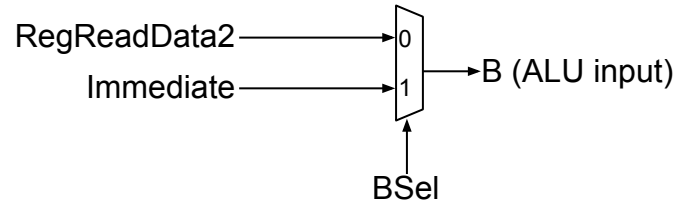
- Immediate select
- Choose which bits of the instruction to use for the immediate
 - One ImmSel value for each instruction type
- Only used by the immediate generator

Control Signal: BrUn

- Branch unsigned
- Chooses whether the branch comparator should perform a signed or unsigned comparison
 - BrUn=0: Perform a signed comparison
 - BrUn=1: Perform an unsigned comparison
- Only used by the branch comparator
- Control logic subcircuit decodes instruction and outputs appropriate BrUn
 - Example: For **b1t** instructions, set BrUn=0
 - Example: For **b1tu** instructions, set BrUn=1
 - Example: For R-type instructions, set BrUn=* (we don't care if it's 0 or 1)

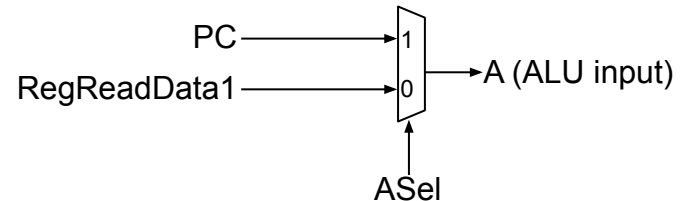
Control Signal: BSel

- 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
 - Example: For R-type instructions, set BSel=0
 - Example: For I-type instructions, set BSel=1



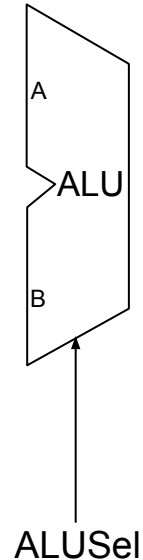
Control Signal: ASel

- 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



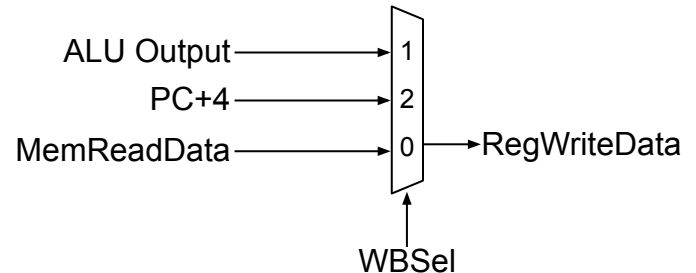
Control Signal: ALUSel

- Chooses what computation the ALU should do
 - Examples: add, subtract, bitwise AND, etc.
 - One ALUSel value for each ALU operation
- Control logic subcircuit decodes instruction and outputs appropriate ALUSel
 - Example: For `ori` instructions, set ALUSel = bitwise OR
 - Example: For load/store instructions, set ALUSel = add (so we can add the value in `rs1` + immediate = address to load/store from)
 - Example: For `jal` instructions, set ALUSel = add (so we can add PC + immediate = address to jump to)



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
 - WBSel=2: Write PC+4 to the register
- Control logic subcircuit decodes instruction and outputs appropriate WBSel
 - Example: For load instructions, set WBSel=0
 - Example: For R-type and non-load I-type instructions, set WBSel=1
 - Example: For jal instructions, set WBSel=2

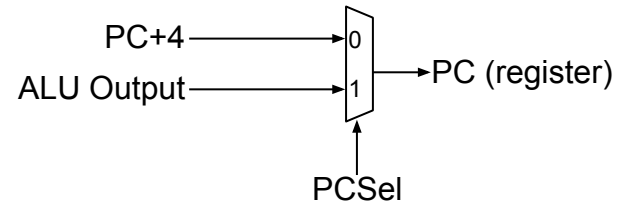


Control Signals: MemWEn, RegWEn

- 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

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: **b1t** 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.
 - Example: **ja1** instruction. PCSel=1.



Implementing Control Logic

Control Logic Truth Table

inst	BrEq	BrLt	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemWEn	RegWEn	WBSEL
add	*	*	+4	*	*	Reg	Reg	Add	0	1	ALU
sub	*	*	+4	*	*	Reg	Reg	Sub	0	1	ALU
R-type	*	*	+4	*	*	Reg	Reg	(op)	0	1	ALU
addi	*	*	+4	I	*	Reg	Imm	Add	0	1	ALU
lw	*	*	+4	I	*	Reg	Imm	Add	0	1	Mem
sw	*	*	+4	S	*	Reg	Imm	Add	1	0	*
beq	0	*	+4	B	*	PC	Imm	Add	0	0	*
beq	1	*	ALU	B	*	PC	Imm	Add	0	0	*
bne	0	*	ALU	B	*	PC	Imm	Add	0	0	*
blt	*	1	ALU	B	0	PC	Imm	Add	0	0	*
bltu	*	1	ALU	B	1	PC	Imm	Add	0	0	*
jalr	*	*	ALU	I	*	Reg	Imm	Add	0	1	PC+4
jal	*	*	ALU	J	*	PC	Imm	Add	0	1	PC+4
auipc	*	*	+4	U	*	PC	Imm	Add	0	1	ALU

Determining Instruction From Bits

- Which bits of the instruction are needed to uniquely identify the instruction?
 - Opcode: `inst[6:0]`
 - Funct3: `inst[14:12]`
 - Funct7: `inst[31:25]`
 - Can we use fewer bits?

Instruction	Opcode	Funct3	Funct7
<code>add</code>	011 0011	000	000 0000
<code>sub</code>	011 0011	000	010 0000
<code>and</code>	011 0011	111	000 0000
<code>or</code>	011 0011	110	000 0000
<code>xor</code>	011 0011	100	000 0000
<code>sll</code>	011 0011	001	000 0000
<code>srl</code>	011 0011	101	000 0000
<code>sra</code>	011 0011	101	010 0000
<code>slt</code>	011 0011	010	000 0000
<code>sltu</code>	011 0011	011	000 0000

Instruction	Opcode	Funct3	Funct7
<code>addi</code>	001 0011	000	
<code>andi</code>	001 0011	111	
<code>ori</code>	001 0011	110	
<code>xori</code>	001 0011	100	
<code>slli</code>	001 0011	001	000 0000
<code>srl</code>	001 0011	101	000 0000
<code>srai</code>	001 0011	101	010 0000
<code>slti</code>	001 0011	010	
<code>sltiu</code>	001 0011	011	

Instruction	Opcode	Funct3
<code>lb</code>	000 0011	000
<code>lbu</code>	000 0011	100
<code>lh</code>	000 0011	001
<code>lhu</code>	000 0011	101
<code>lw</code>	000 0011	010
<code>sb</code>	010 0011	000
<code>sh</code>	010 0011	001
<code>sw</code>	010 0011	010

Instruction	Opcode	Funct3
<code>beq</code>	110 0011	000
<code>bge</code>	110 0011	101
<code>bgeu</code>	110 0011	111
<code>blt</code>	110 0011	100
<code>bltu</code>	110 0011	110
<code>bne</code>	110 0011	001
<code>jal</code>	110 1111	
<code>jalr</code>	110 0111	000
<code>auipc</code>	001 0111	
<code>lui</code>	011 0111	

Determining Instruction From Bits

- Note that some bits are the same for all RISC-V 32-bit base instructions
 - Opcode: `inst[6:2]` (lower two bits `inst[1:0]` are always the same)
 - Funct3: `inst[14:12]`
 - Funct7: `inst[30]` (all other bits of funct7 are always the same)
 - 9 bits are needed to uniquely identify the instruction

Instruction	Opcode	Funct3	Funct7
add	011 0011	000	000 0000
sub	011 0011	000	010 0000
and	011 0011	111	000 0000
or	011 0011	110	000 0000
xor	011 0011	100	000 0000
sll	011 0011	001	000 0000
srl	011 0011	101	000 0000
sra	011 0011	101	010 0000
slt	011 0011	010	000 0000
sltu	011 0011	011	000 0000

Instruction	Opcode	Funct3	Funct7
addi	001 0011	000	
andi	001 0011	111	
ori	001 0011	110	
xori	001 0011	100	
slli	001 0011	001	000 0000
srl	001 0011	101	000 0000
srai	001 0011	101	010 0000
slti	001 0011	010	
sltiu	001 0011	011	

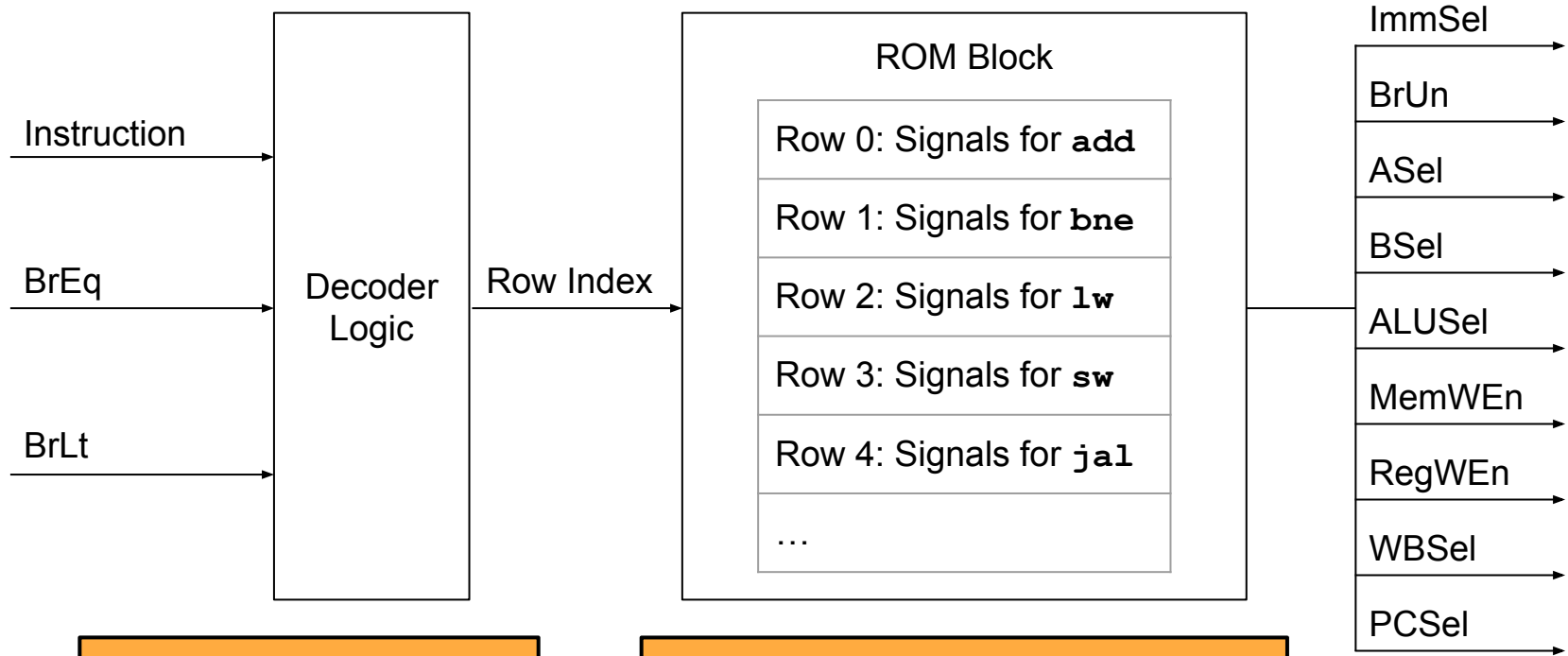
Instruction	Opcode	Funct3
lb	000 0011	000
lbu	000 0011	100
lh	000 0011	001
lhu	000 0011	101
lw	000 0011	010
sb	010 0011	000
sh	010 0011	001
sw	010 0011	010

Instruction	Opcode	Funct3
beq	110 0011	000
bge	110 0011	101
bgeu	110 0011	111
blt	110 0011	100
bltu	110 0011	110
bne	110 0011	001
jal	110 1111	
jalr	110 0111	000
auipc	001 0111	
lui	011 0111	

Implementing Control Logic

- Control logic is purely combinatorial logic
 - No registers, no clocks
- Approach 1: ROM
 - ROM = read-only memory
 - Store a table of control signals for each instruction in memory
 - Use the bits of the input to index into the table and select a row of signals to output
- Approach 2: Combinatorial logic gates
 - Use logic gates to compute signals from the bits of the input

Approach 1: ROM



Use the bits of the input to pick a row index.

Then, the ROM block outputs the hard-coded control signals on that row.

Approach 2: Logic Gates

- Can we write a Boolean expression for BrUn?
- Recall: BrUn determines if a branch comparison is signed
 - BrUn must be 1 for **bgeu** and **bltu**
 - BrUn must be 0 for **bge** and **blt**
 - BrUn can be either 0 or 1 for everything else
- Notice bit 13 of the instruction
 - This bit is 1 for **bgeu** and **bltu**
 - This bit is 0 for **bge** and **blt**
- $\text{BrUn} = \text{inst}[13]$

Instruction	Opcode <code>inst[6:0]</code>	Funct3 <code>inst[14:12]</code>
beq	110 0011	000
bge	110 0011	101
bgeu	110 0011	111
blt	110 0011	100
bltu	110 0011	110
bne	110 0011	001

Approach 2: Logic Gates

- Can we write a Boolean expression telling us if an instruction is **add**?
 - 1 if the input instruction is **add**, and 0 otherwise
 - Can be used later to compute control logic signals
 - Example: RegWEn is 1 if the instruction is **add**, or if the instruction is **sub**, or...
- Using sum-of-products:
 - $\text{add} = !i[30] \cdot !i[14] \cdot !i[13] \cdot !i[12] \cdot !i[6] \cdot i[5] \cdot i[4] \cdot !i[3] \cdot !i[2] \cdot i[1] \cdot i[0]$

Instruction	Opcode <code>inst[6:0]</code>	Funct3 <code>inst[14:12]</code>	Funct7 <code>inst[31:25]</code>
add	011 0011	000	000 0000

Comparing Implementations

- Approach 1: ROM
 - Regular structure
 - Can be easily reprogrammed to fix errors or add instructions (just update the table of signals)
 - Popular when designing control logic manually
- Approach 2: Combinatorial logic gates
 - Today, chip designers use logic synthesis tools to convert truth tables to networks of gates

Introduction to Pipelining

Measuring Performance

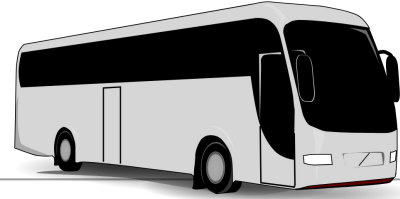
At one time, most of the components are idle

Instruction Timing

- Clock period: 800ps
 - Determined by longest path (lw)
 - Frequency: $1/800\text{ps} = 1.25\text{ GHz}$
 - 1.25 billion instructions per second
- Can we improve our datapath performance?
 - What does it mean to improve performance?
 - Quicker response time = one job finishes faster?
 - More jobs per unit time?
 - Longer battery life?

instr	IF	ID	EX	MEM	WB	Total
add	200ps	100ps	200ps		100ps	600ps
beq	200ps	100ps	200ps			500ps
jal	200ps	100ps	200ps		100ps	600ps
lw	200ps	100ps	200ps	200ps	100ps	800ps
sw	200ps	100ps	200ps	200ps		700ps

Analogy: Transportation



	Sports Car	Bus
Passenger Capacity	2	50
Travel Speed	200 mph	50 mph
Gas Mileage	5 mpg	2 mpg

For a 50 mile trip (assuming instant return trips)...

	Sports Car	Bus
Travel Time	15 min	60 min
Time for 100 passengers	750 min (50 trips)	120 min (2 trips)
Gallons per passenger	5 gallons	0.5 gallons

Performance in Computers

Transportation	Computer
Trip Time	Program execution time (Example: Time to update display)
Time for 1 passenger	Latency (Example: Seconds needed to complete one singular instruction)
Time for 100 passengers	Throughput (Example: Number of server requests handled per hour)
Gallons per passenger	Energy per task (Example: How many movies you can watch per battery charge)

Optimizing Time Per Program

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- “Iron Law” of processor performance
- Three components for optimizing the time it takes to execute a program

Optimizing Time Per Program

$$\frac{\text{Time}}{\text{Program}} = \boxed{\frac{\text{Instructions}}{\text{Program}}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- Instruction per program determined by:
 - Task
 - Algorithm (e.g. $O(n^2)$ vs. $O(n)$)
 - Programming language
 - Compiler
 - Instruction set architecture (ISA)

Optimizing Time Per Program

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \boxed{\frac{\text{Cycles}}{\text{Instruction}}} \times \frac{\text{Time}}{\text{Cycle}}$$

- Average clock cycles per instruction (CPI) determined by:
 - Instruction set architecture (ISA)
 - Processor implementation (microarchitecture)
 - Example: For the single-cycle RISC-V processor we made, $\text{CPI} = 1$
 - Example: For complex instructions (e.g. strcpy), $\text{CPI} > 1$
 - Example: We'll see $\text{CPI} < 1$ processors soon

Optimizing Time Per Program

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \boxed{\frac{\text{Time}}{\text{Cycle}}}$$

- Time per cycle (1/frequency) determined by:
 - Processor microarchitecture (critical path through logic gates)
 - Technology (how many transistors fit in the space)
 - Power budget (lower voltages reduce transistor speed)

Optimizing Time Per Program

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

	Processor A	Processor B
# Instructions	1 million	1.5 million
Average CPI	2.5	1
Clock rate f	2.5 GHz	2 GHz
Execution time	1 ms	0.75 ms

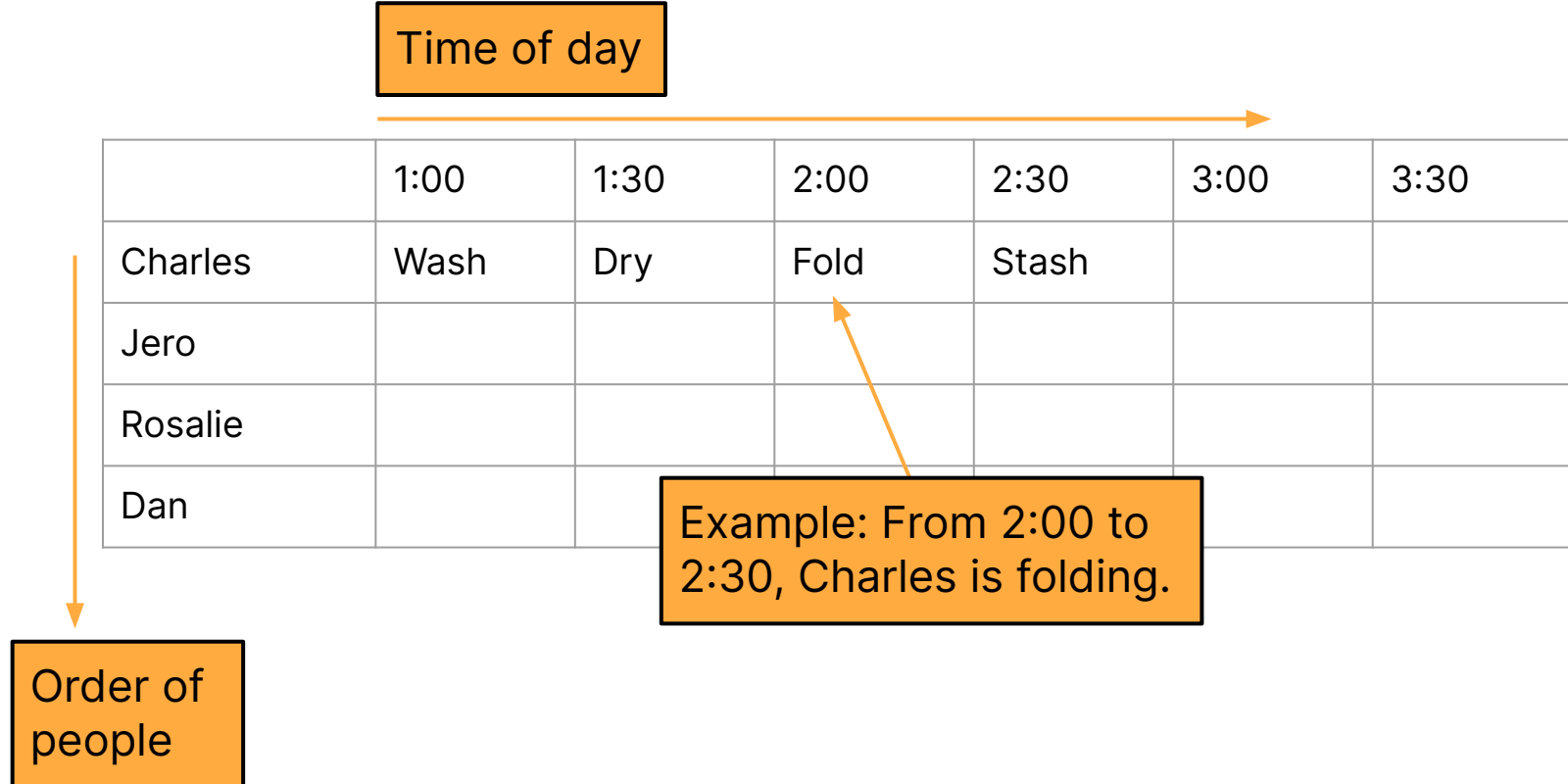
Processor B is faster for this task, despite executing more instructions and having a slower clock rate!

Pipelining Analogy

Pipelining Analogy: Laundry

- Charles, Jero, Rosalie, Dan each have one load of clothes to wash, dry, fold, and put away
 - W: Washer takes 30 minutes
 - D: Dryer takes 30 minutes
 - F: Folding clothes on the table takes 30 minutes
 - S: Stashing clothes in drawers takes 30 minutes

Laundry Timing Diagram



Sequential Laundry


	1:00	1:30	2:00	2:30	3:00	3:30	4:00	4:30	5:00	5:30	6:00	6:30	7:00	7:30	8:00	8:30
Charles	Wash	Dry	Fold	Stash												
Jero					Wash	Dry	Fold	Stash								
Rosalie									Wash	Dry	Fold	Stash				
Dan													W	D	F	S

How long for all four people to do laundry? **8 hours**

How long for one person to do laundry? **2 hours**

Sequential Laundry

	1:00	1:30	2:00	2:30	3:00	3:30	4:00	4:30	5:00	5:30	6:00	6:30	7:00	7:30	8:00	8:30
Charles	Wash	Dry	Fold	Stash												
Jero					Wash	Dry	Fold	Stash								
Rosalie									Wash	Dry	Fold	Stash				
Dan													W	D	F	S



Idea: When one person is washing, no one's using the dryer.

Pipelined Laundry

	1:00	1:30	2:00	2:30	3:00	3:30	4:00
Charles	Wash	Dry	Fold	Stash			
Jero		Wash	Dry	Fold	Stash		
Rosalie			Wash	Dry	Fold	Stash	
Dan				Wash	Dry	Fold	Stash

How long for all four people to do laundry? **3.5 hours**

How long for one person to do laundry? **2 hours** (same as before)

Pipelined Laundry

	1:00	1:30	2:00	2:30	3:00	3:30	4:00
Charles	Wash	Dry	Fold	Stash			
Jero		Wash	Dry	Fold	Stash		
Rosalie			Wash	Dry	Fold	Stash	
Dan				Wash	Dry	Fold	Stash

One person uses all the equipment (e.g. washer, dryer) over time.

At one time, different people are using different resources simultaneously.

Pipelined Laundry

W	D	F	S								
	W	D	F	S							
		W	D	F	S						
			W	D	F	S					
				W	D	F	S				
					W	D	F	S			
						W	D	F	S		
							W	D	F	S	
								W	D	F	S
									W	D	F

Ideal speedup is 4x (four things happening at once), but limited by startup (“fill the pipeline”) and shutdown (“drain the pipeline”) costs.

At the start and end, some resources are unused.

Pipelined Laundry

	1:00	1:30	2:00	2:30	3:00	3:30	4:00
Charles	Wash	Dry	Fold	Stash			
Jero		Wash	Dry	Fold	Stash		
Rosalie			Wash	Dry	Fold	Stash	
Dan				Wash	Dry	Fold	Stash

What if drying only took 20 minutes now?

Dryer finishes early, but has to wait for the folding to finish.

Pipeline rate is **limited by the slowest pipeline stage**

Pipelining Laundry: Observations

- Latency (time it takes to finish a single task) is unchanged
 - It still took 2 hours for one person to do laundry
- Throughput (number of jobs finished per hour) increases
 - Non-pipelined: 4 people took 8 hours = 0.5 people per hour
 - Pipelined: 4 people took 3.5 hours = 1.14 people per hour
- Maximum throughput speedup = number of stages
 - If 4 people are using resources at every time, we could have 4x speedup
 - Limited by cost of filling and draining pipeline: not all resources used at the start and end
- Pipeline rate limited by slowest pipeline stage
 - Everyone needs to wait for the slowest stage to finish before moving to the next stage
 - Balancing the length of each pipeline stage is good for speedup