EECS151/251A
Introduction to Digital Design and ICs

Lecture 9:
RISC-V Datapath and Control II

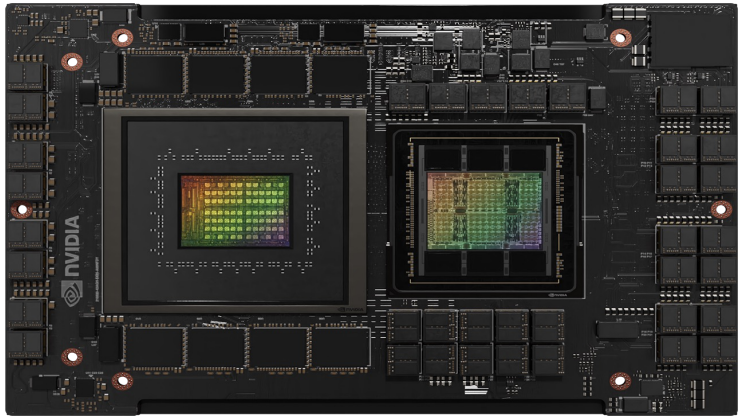
Sophia Shao

NVIDIA Grace Hopper Superchip

NVIDIA Grace CPU is the first NVIDIA data center CPU, and it is built from the ground up to create HPC and AI superchips. The NVIDIA Grace CPU uses Arm Neoverse V2 CPU cores to deliver leading per-thread performance, while providing higher energy efficiency than traditional CPUs.

NVIDIA Hopper is the ninth-generation NVIDIA data center GPU and is designed to deliver order-of-magnitude improvements for large-scale AI and HPC applications compared to previous NVIDIA Ampere GPU generations.

NVIDIA Grace Hopper fuses an NVIDIA Grace CPU and an NVIDIA Hopper GPU into a single superchip via NVIDIA NVLink C2C, a 900 GB/s total bandwidth chip-to-chip interconnect.



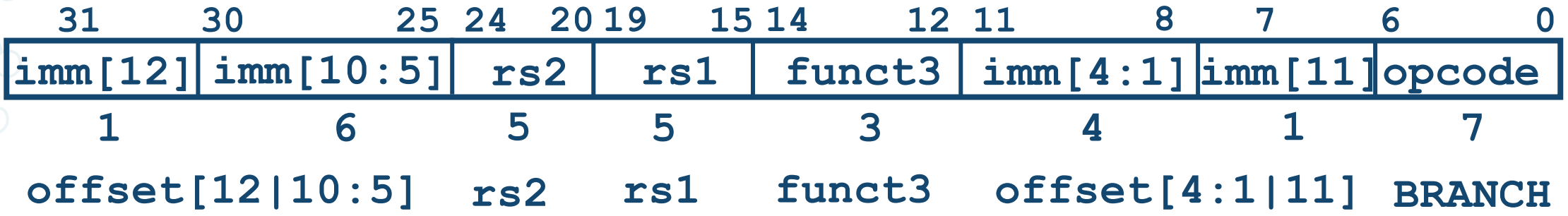
https://nvdam.widen.net/s/qjzrmfdn2j/nvidia-grace-hopper-superchip-architecture-whitepaper-v1.0

- **RISC-V Datapath & Control**
  - R-type
  - I-type
  - S-type
  - **B-type**
  - **J-type**
  - **U-type**
  - **Control Logic**

B-Format - RISC-V Conditional Branches

- E.g., **`BEQ x1, x2, Label`**
- Branches read two registers but don't write a register (similar to stores)
- How to encode label, i.e., where to branch to?

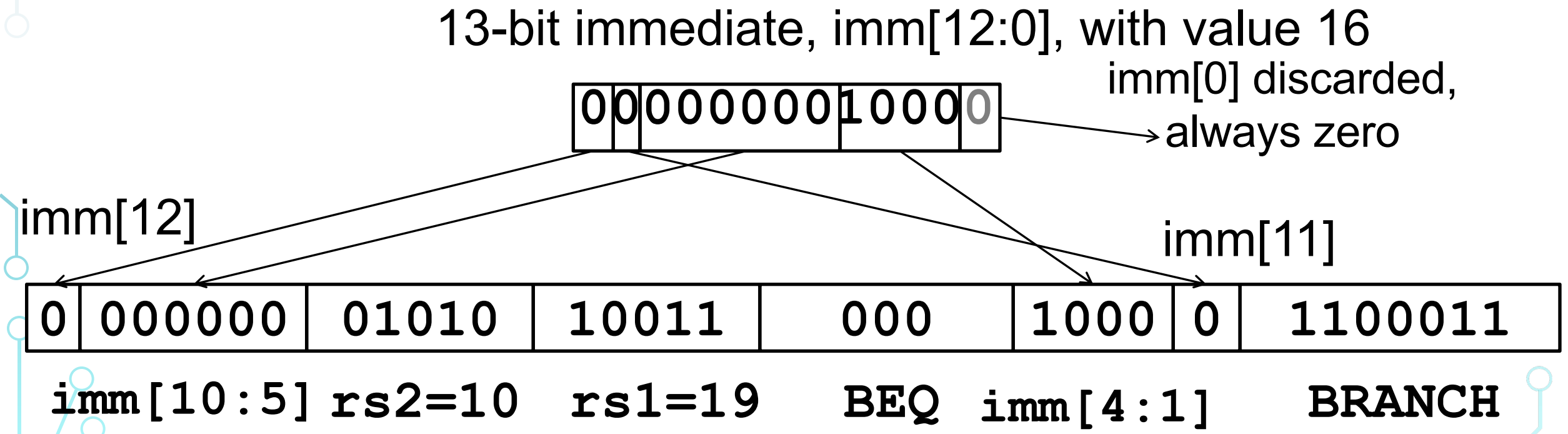
Implementing Branches



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

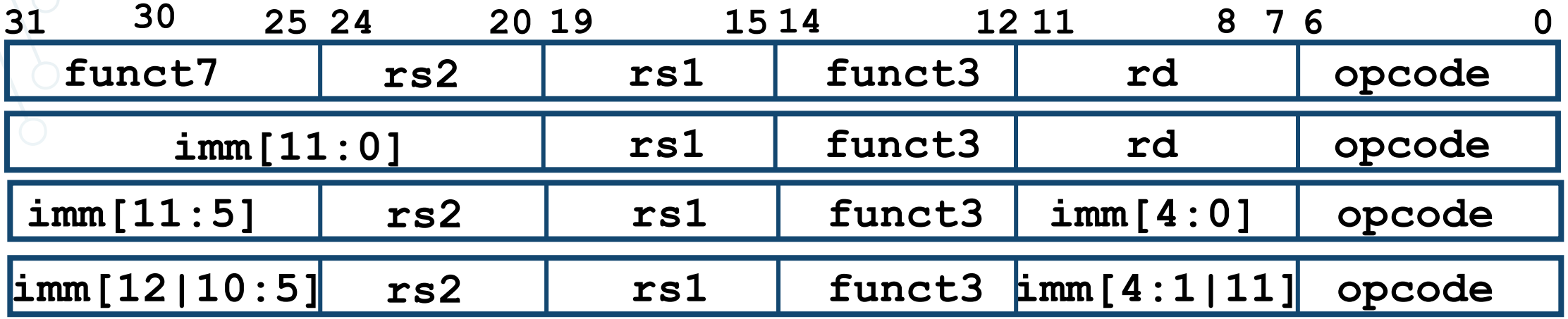
Branch Example, complete encoding

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

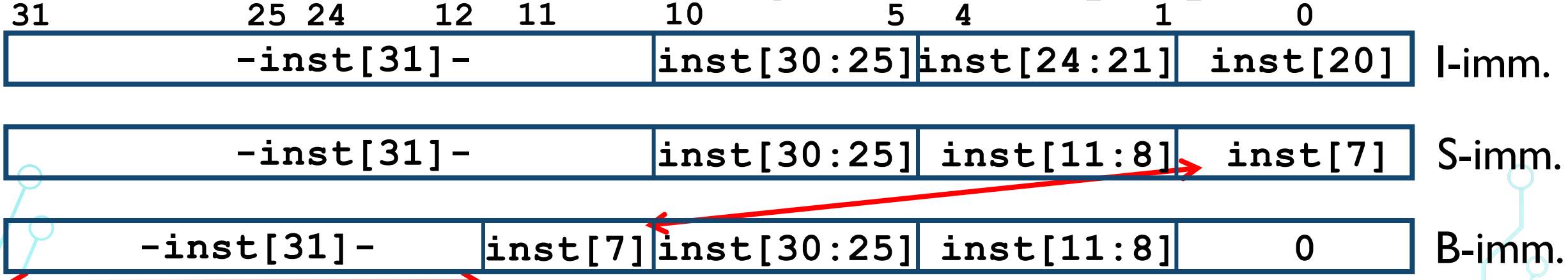


RISC-V Immediate Encoding

Instruction encodings, inst[31:0]



32-bit immediates produced, imm[31:0]



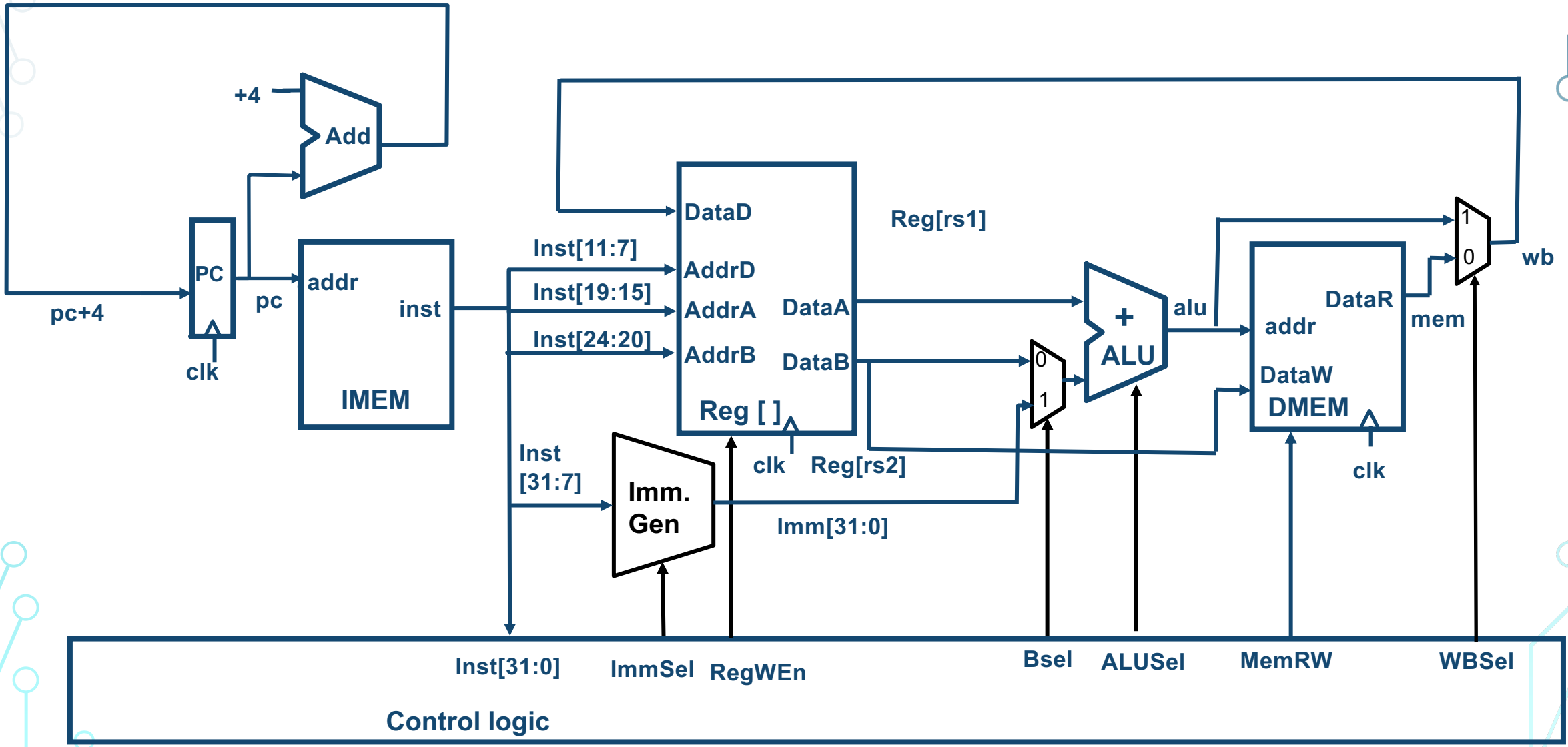
Upper bits sign-extended from inst[31] always

Only bit 7 of instruction changes role in immediate between S and B

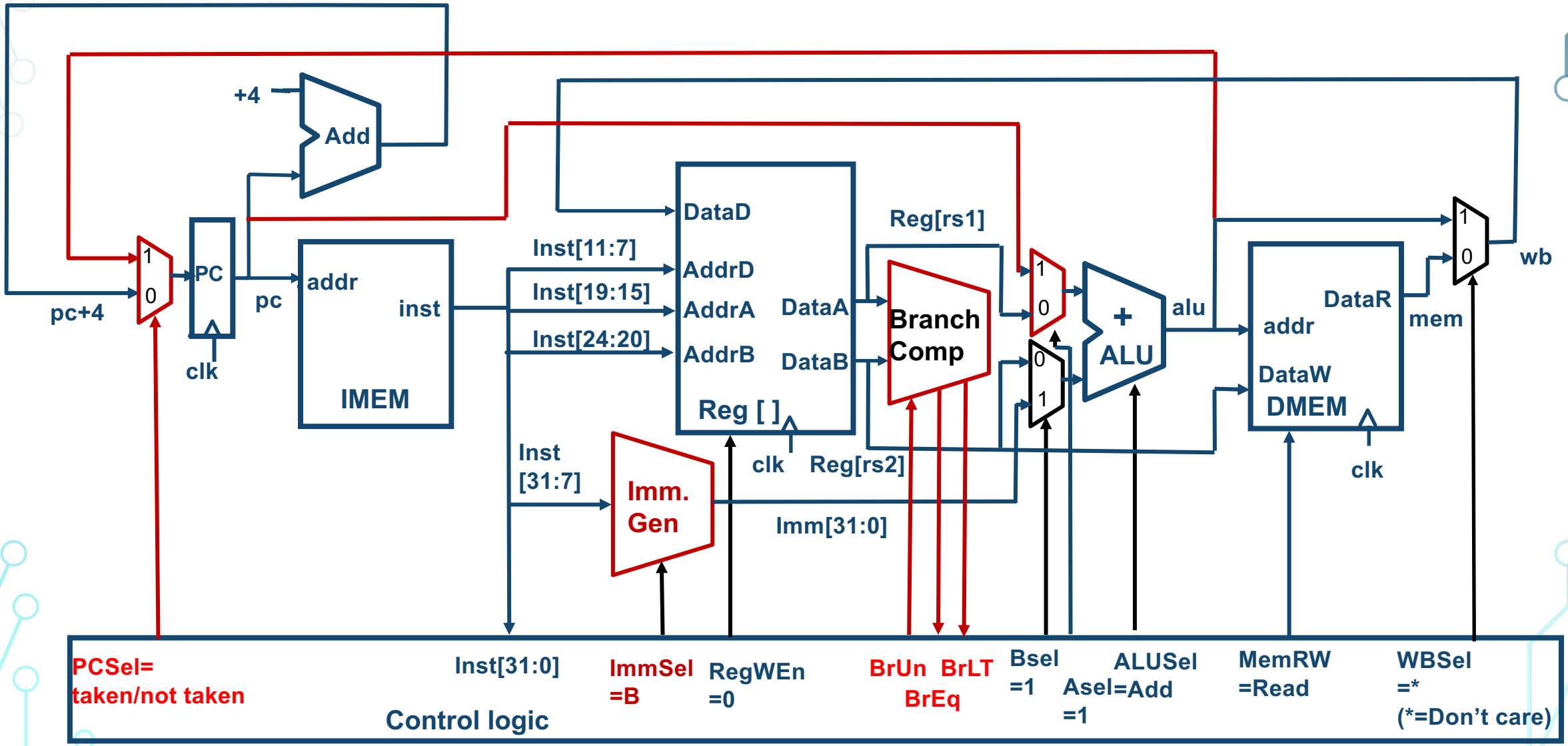
To Add Branches

- Different change to the state:
  - `PC = PC + 4,` `branch not taken`
    `PC + immediate,` `branch taken`
- Six branch instructions: **`BEQ, BNE, BLT, BGE, BLTU, BGEU`**
- Need to compute **`PC + immediate`** and to compare values of **`rs1`** and **`rs2`**
  - Need another add/sub unit

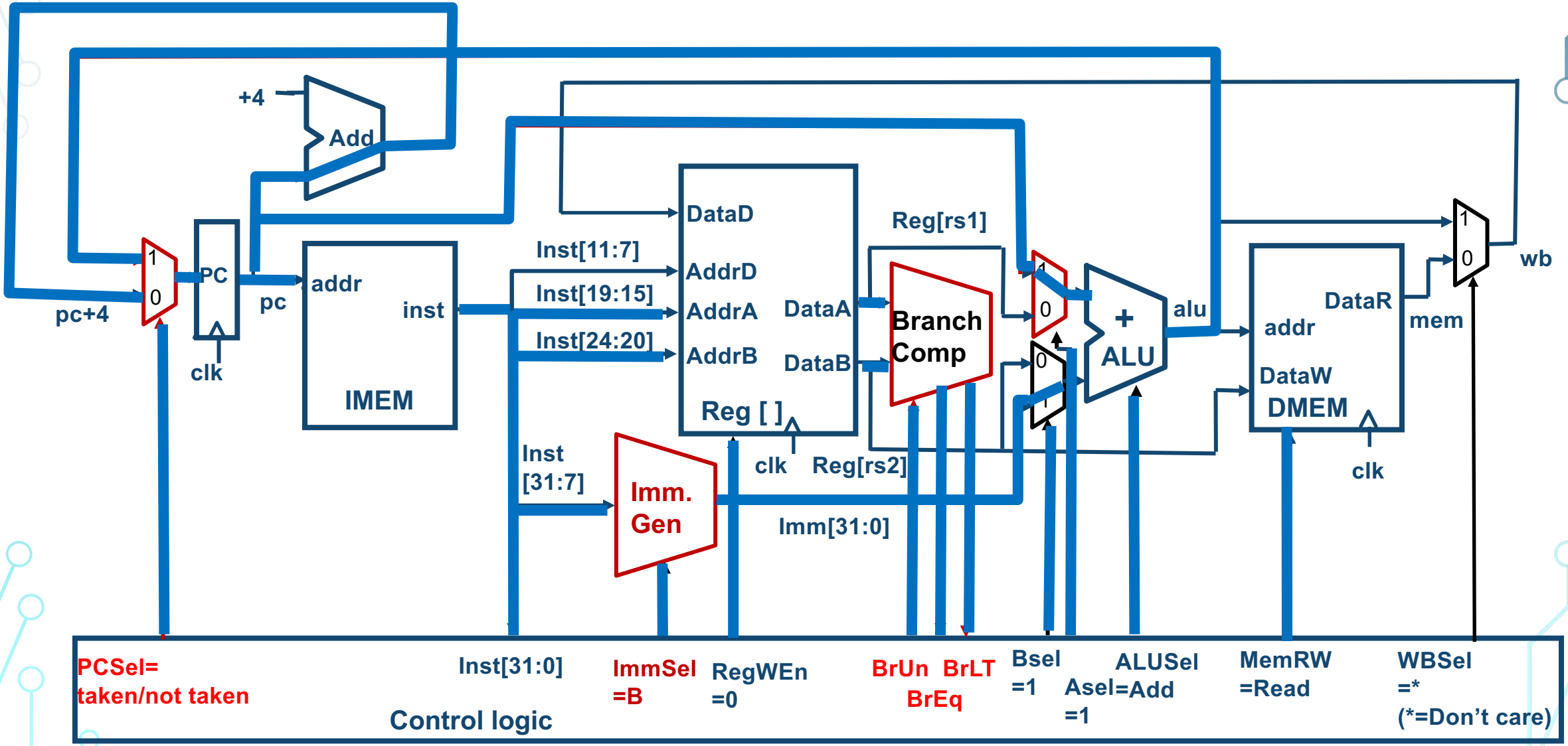
Datapath So Far



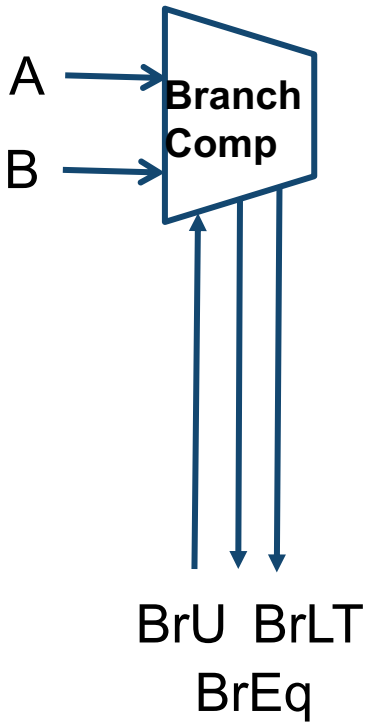
Adding Branches



Adding Branches



Branch Comparator



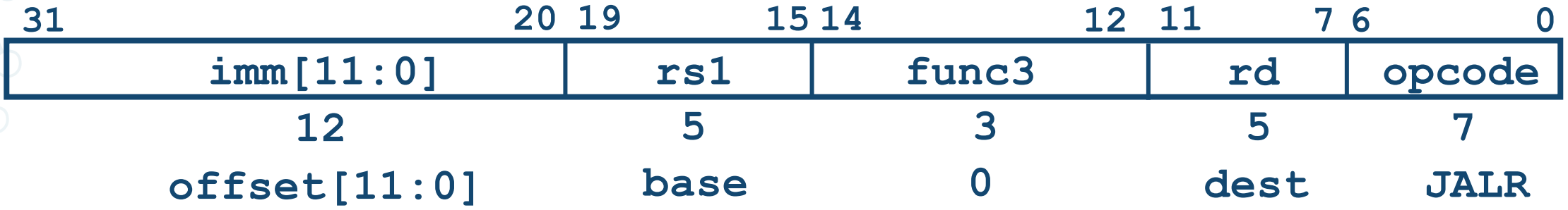
- BrEq = 1, if A=B
- BrLT = 1, if A < B
- BrUn =1 selects unsigned comparison for BrLTU, 0=signed
- BGE branch: A >= B, if $\overline{A<B}$

All RISC-V Branch Instructions

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

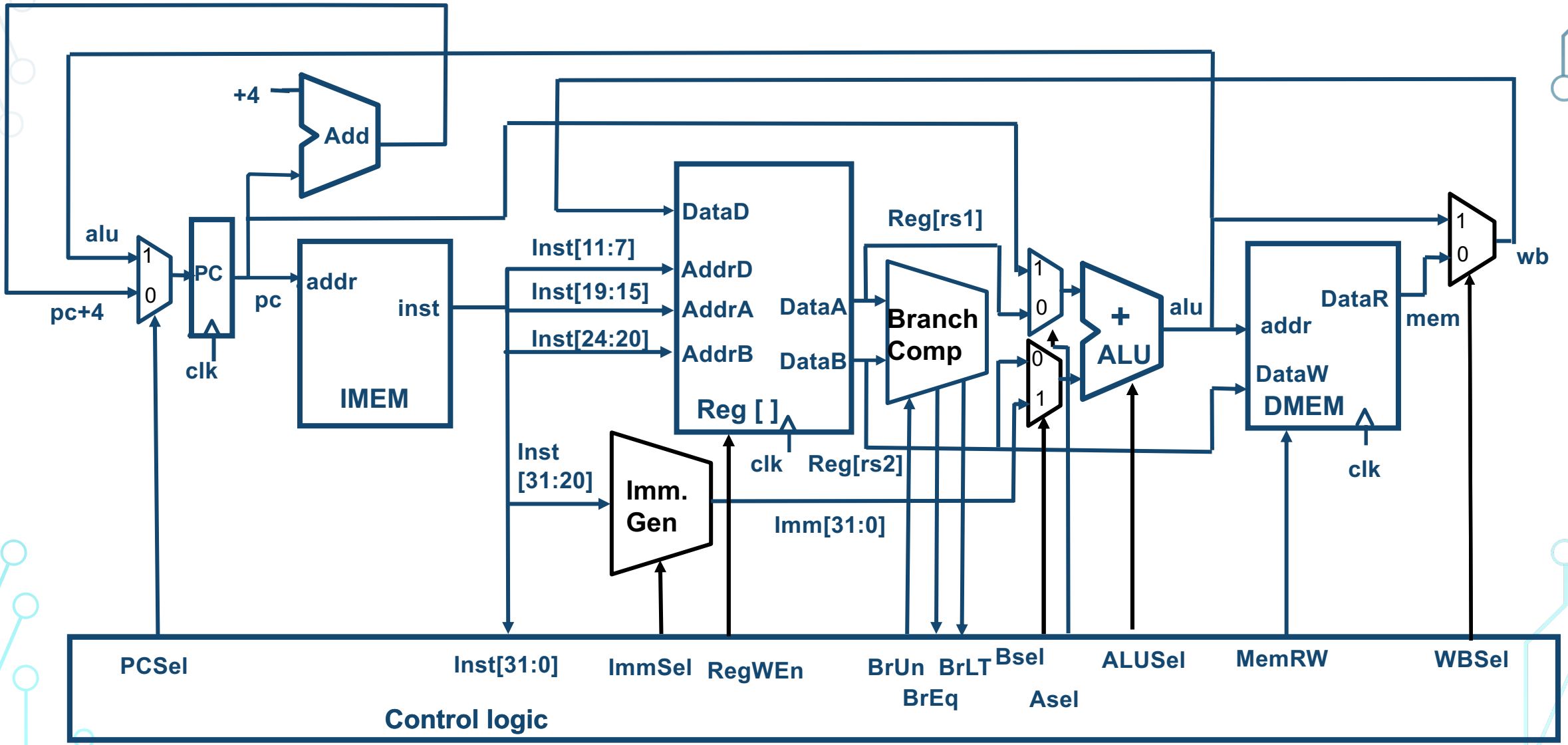
- **RISC-V Datapath & Control**
  - R-type
  - I-type
  - S-type
  - B-type
  - **J-type**
  - **U-type**
  - **Control Logic**

JALR Instruction (I-Format)

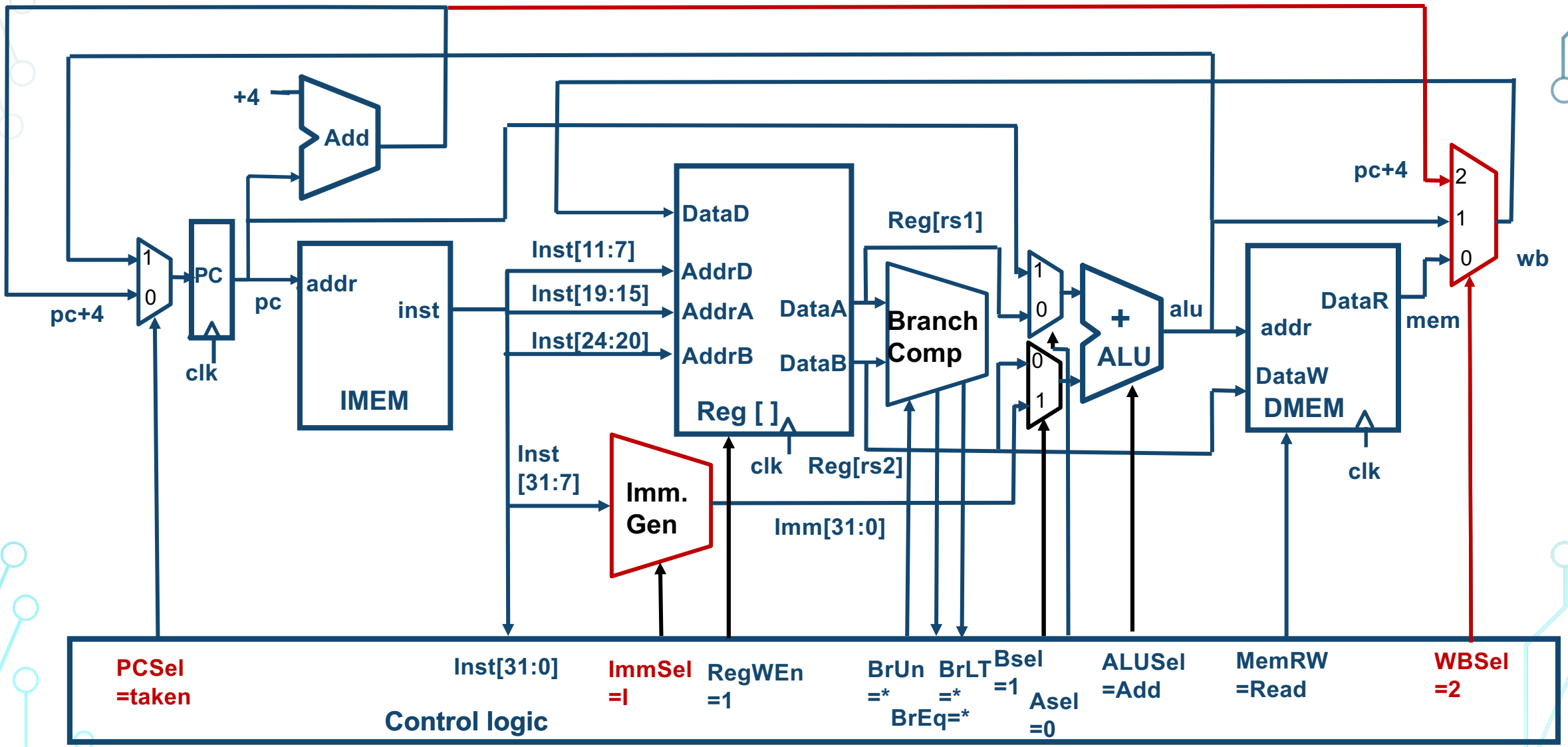


- JALR rd, rs, immediate
  - R[rd] = PC + 4; PC = Reg[rs1] + imm;
    - Writes PC+4 to rd (return address)
    - Sets PC = rs1 + immediate
  - Uses same immediates as arithmetic and loads
    - ***no*** multiplication by 2 bytes
    - In contrast to branches and JAL

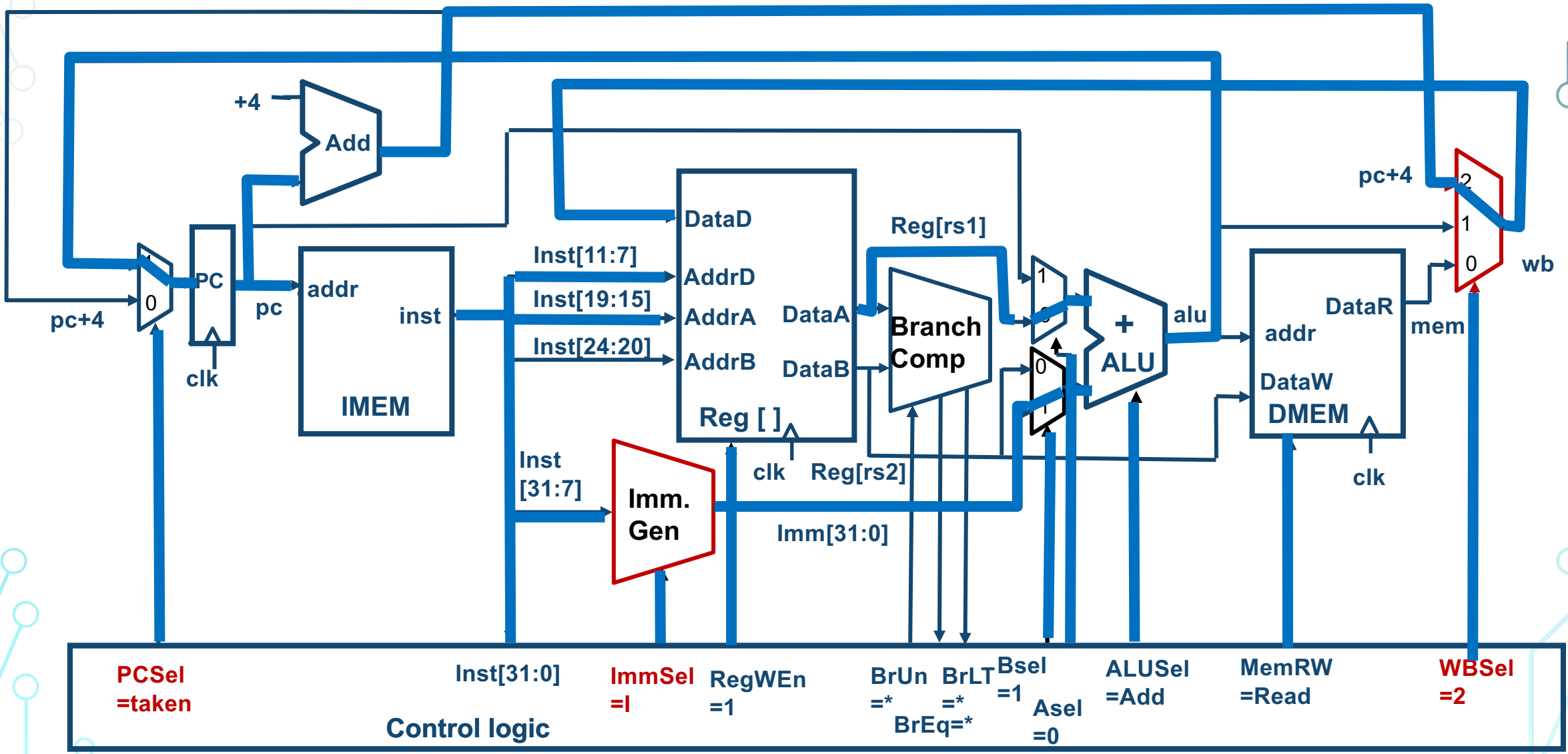
Datapath So Far, with Branches



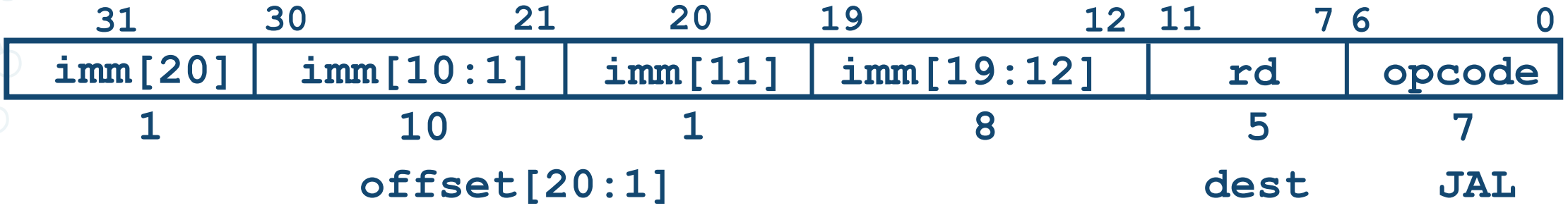
Adding JALR



Adding JALR

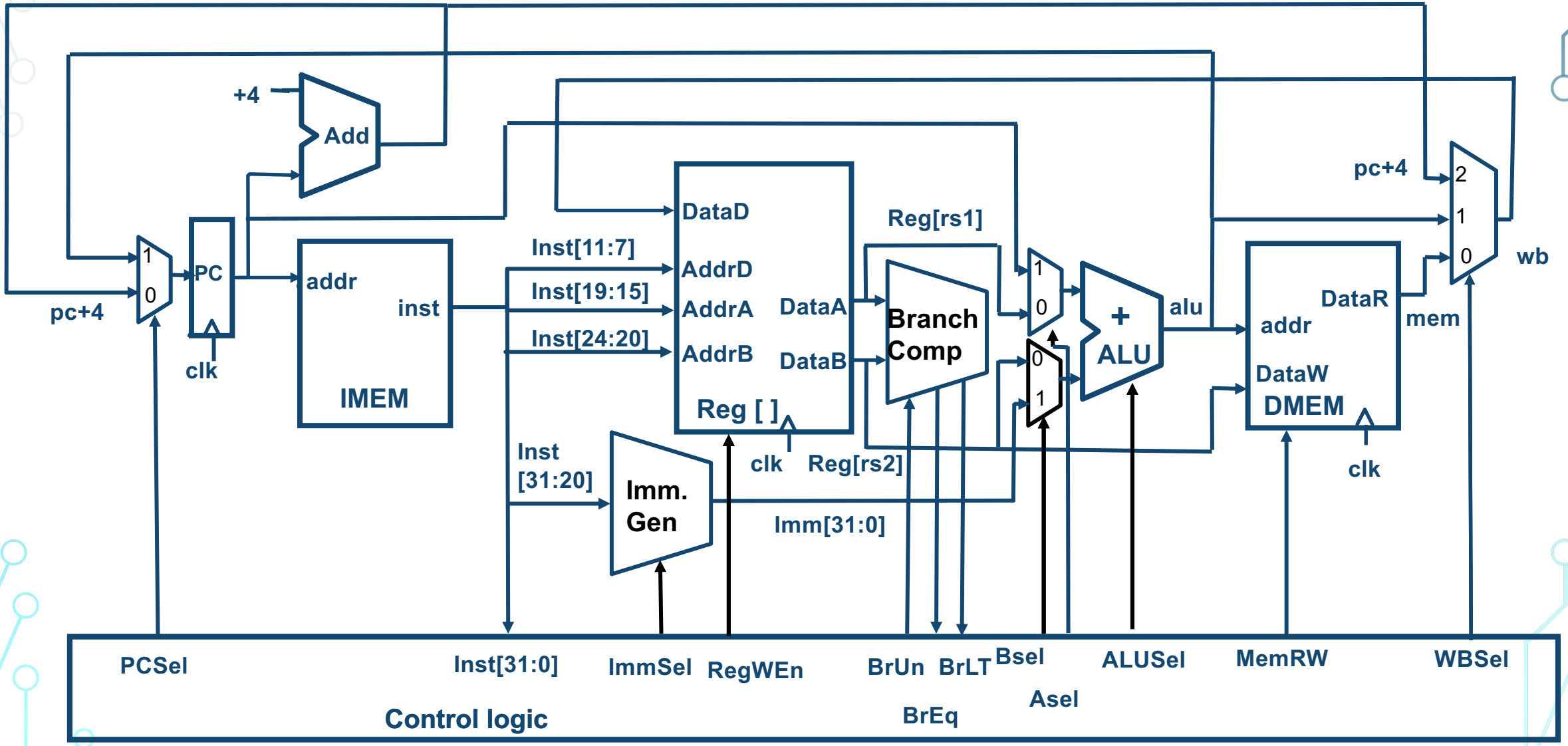

+4
Add
pc+4
0
PC
clk
pc
addr
inst
IMEM
Inst[11:7]
Inst[19:15]
Inst[24:20]
DataD
AddrD
AddrA
AddrB
DataA
DataB
Reg [ ]
clk
Reg[rs1]
Reg[rs2]
Branch Comp
1
0
+
ALU
alu
pc+4
2
1
0
wb
addr
DataR
DataW
DMEM
clk
mem
Inst [31:7]
Imm. Gen
Imm[31:0]
PCSel =taken
Inst[31:0]
ImmSel =I
RegWEn =1
BrUn =*
BrLT =*
BrEq=*
Bsel =1
Asel =0
ALUSel =Add
MemRW =Read
WBSel =2
Control logic

J-Format for Jump Instructions

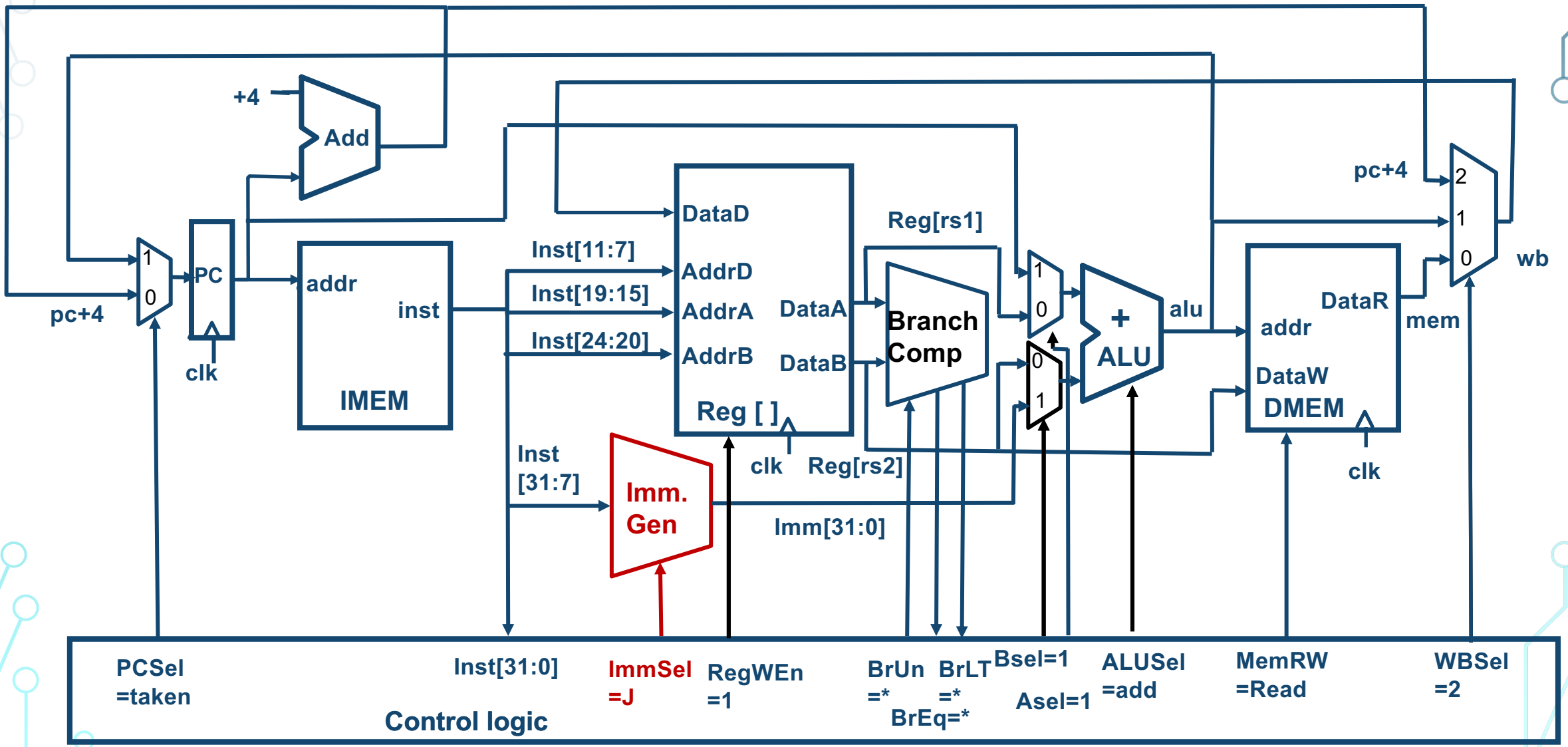


- JAL:
  - R[rd] = PC + 4; PC = PC + imm;
  - saves PC+4 in register rd (the return address)
    - Assembler "`j`" jump is pseudo-instruction, uses JAL but sets `rd=x0` to discard return address
  - Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart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

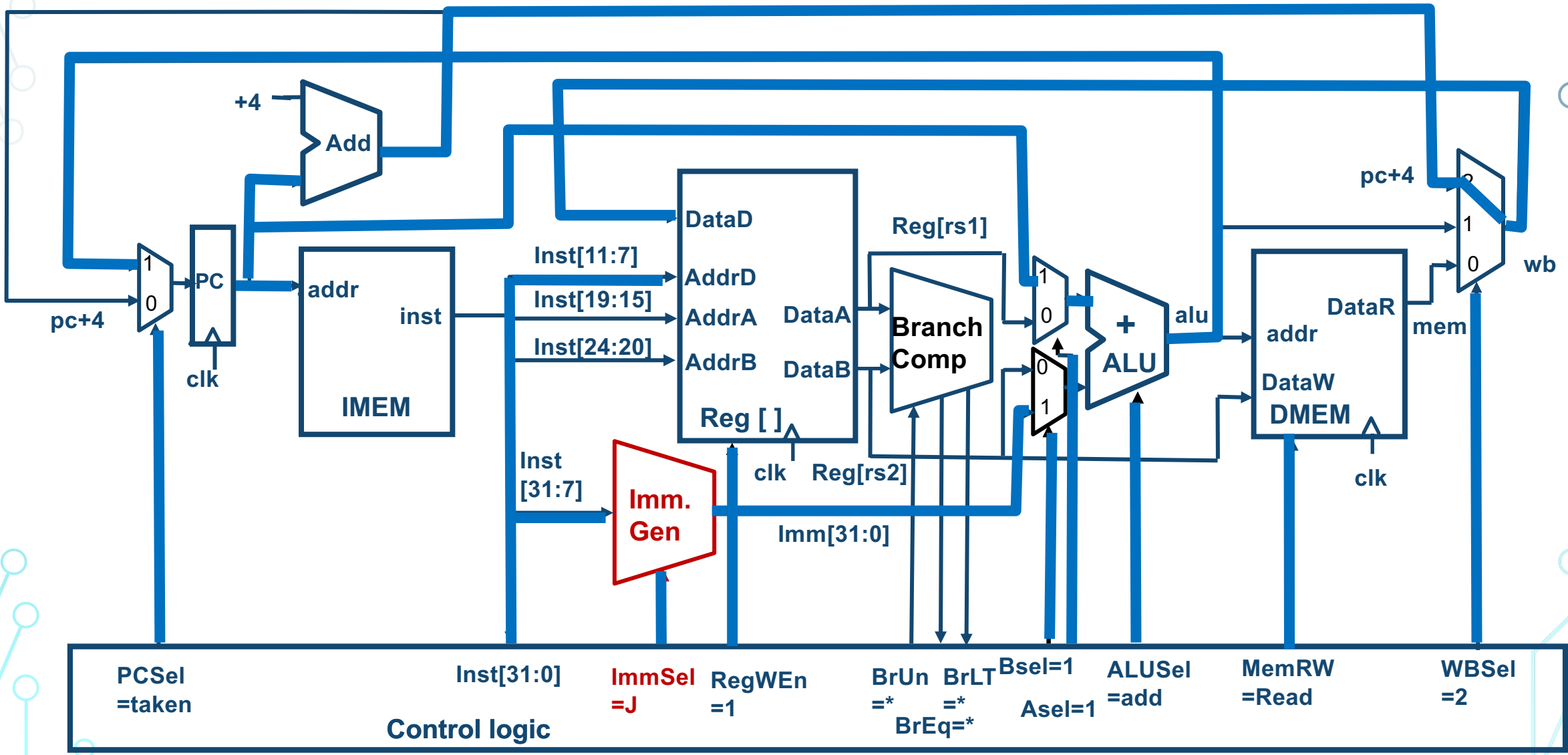
Datapath with JALR



Adding JAL

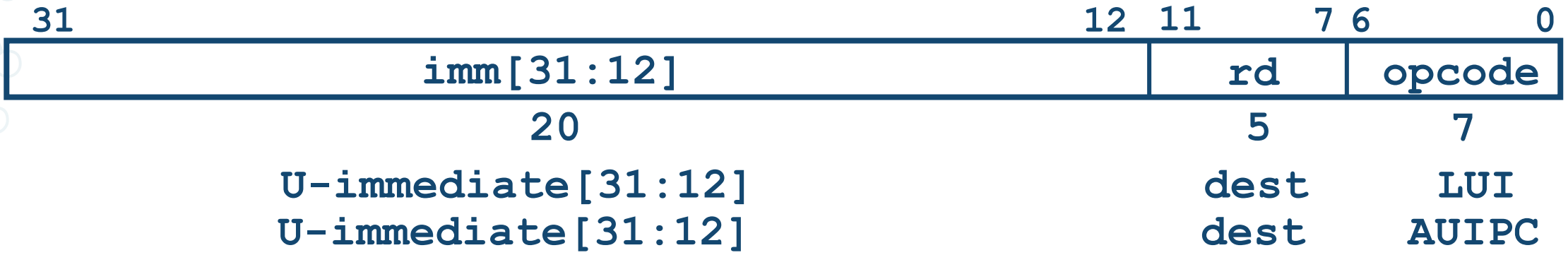

+4
Add
pc+4
1
0
PC
clk
addr
inst
IMEM
Inst[11:7]
Inst[19:15]
Inst[24:20]
DataD
AddrD
AddrA
AddrB
DataA
DataB
Reg [ ]
clk
Reg[rs1]
Reg[rs2]
Branch Comp
1
0
0
1
+
ALU
alu
pc+4
2
1
0
wb
addr
DataR
DataW
DMEM
clk
mem
Inst [31:7]
Imm. Gen
Imm[31:0]
PCSel =taken
Inst[31:0]
ImmSel =J
RegWEn =1
BrUn =*
BrLT =*
BrEq=*
Bsel=1
Asel=1
ALUSel =add
MemRW =Read
WBSel =2
Control logic

Adding JAL


+4
Add
pc+4
1
0
PC
clk
addr
inst
IMEM
Inst[11:7]
Inst[19:15]
Inst[24:20]
DataD
AddrD
AddrA
AddrB
DataA
DataB
Reg [ ]
clk
Reg[rs1]
Reg[rs2]
Branch Comp
Inst [31:7]
Imm. Gen
Imm[31:0]
1
0
0
1
+
ALU
alu
addr
DataR
DataW
DMEM
clk
mem
pc+4
1
0
wb
PCSel =taken
Inst[31:0]
Control logic
ImmSel =J
RegWEn =1
BrUn =*
BrLT =*
BrEq=*
Bsel=1
Asel=1
ALUSel =add
MemRW =Read
WBSel =2

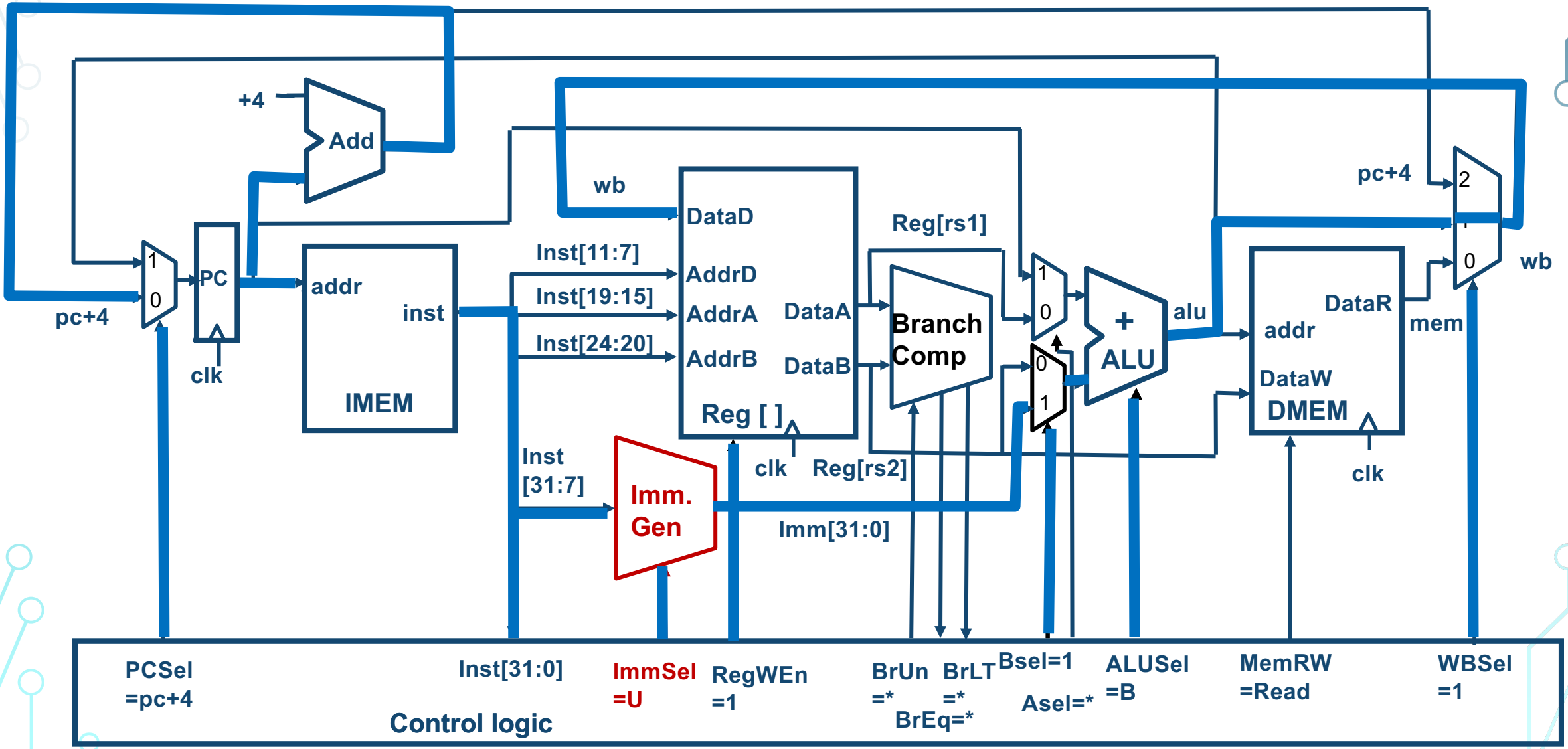
- **RISC-V Datapath & Control**
  - R-type
  - I-type
  - S-type
  - B-type
  - J-type
  - **U-type**
  - **Control Logic**

U-Format for “Upper Immediate” Instructions

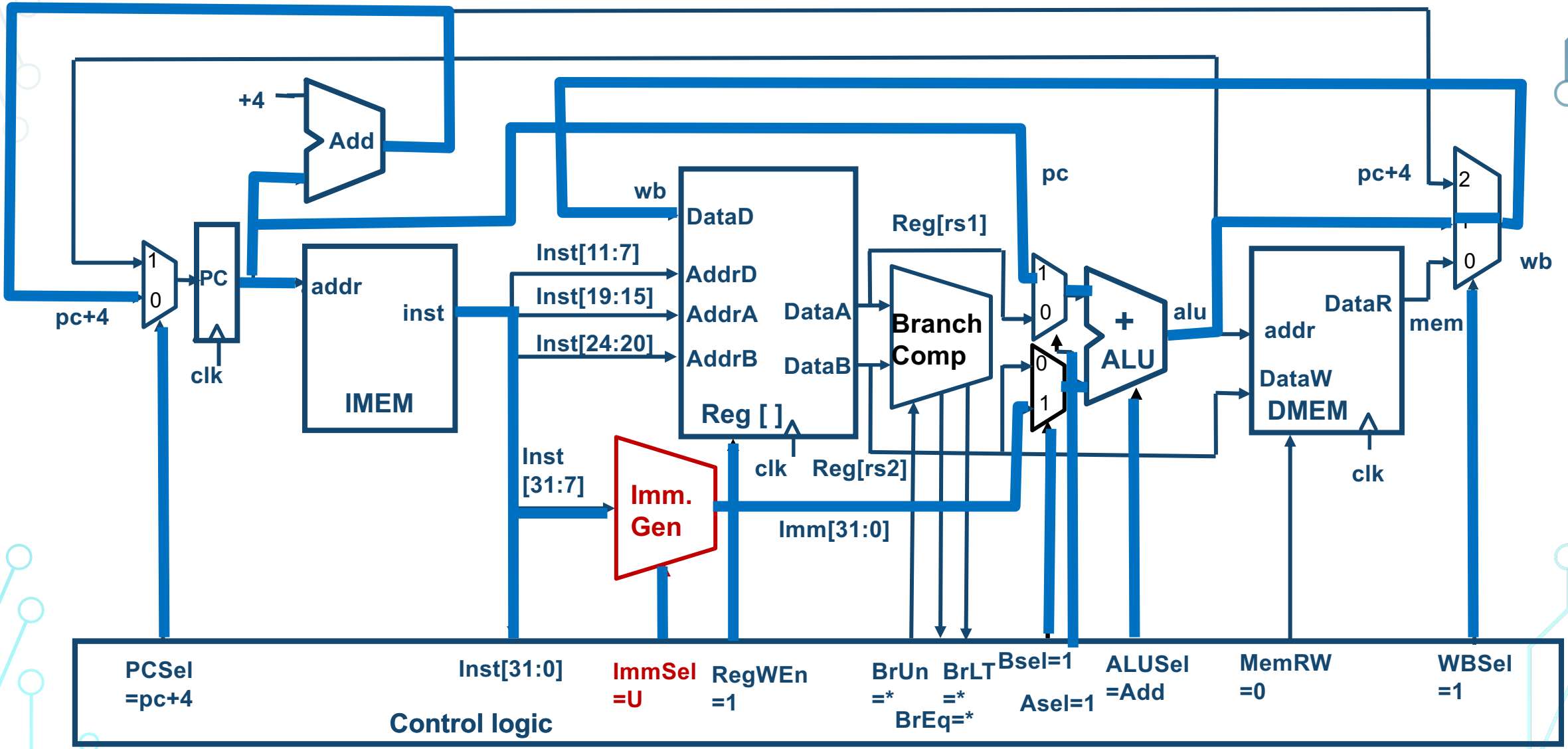


- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
  - LUI – Load Upper Immediate
    - Reg[rd] = {imm, 12b’0}
  - AUIPC – Add Upper Immediate to PC
    - Reg[rd] = PC + {imm, 12’b0}

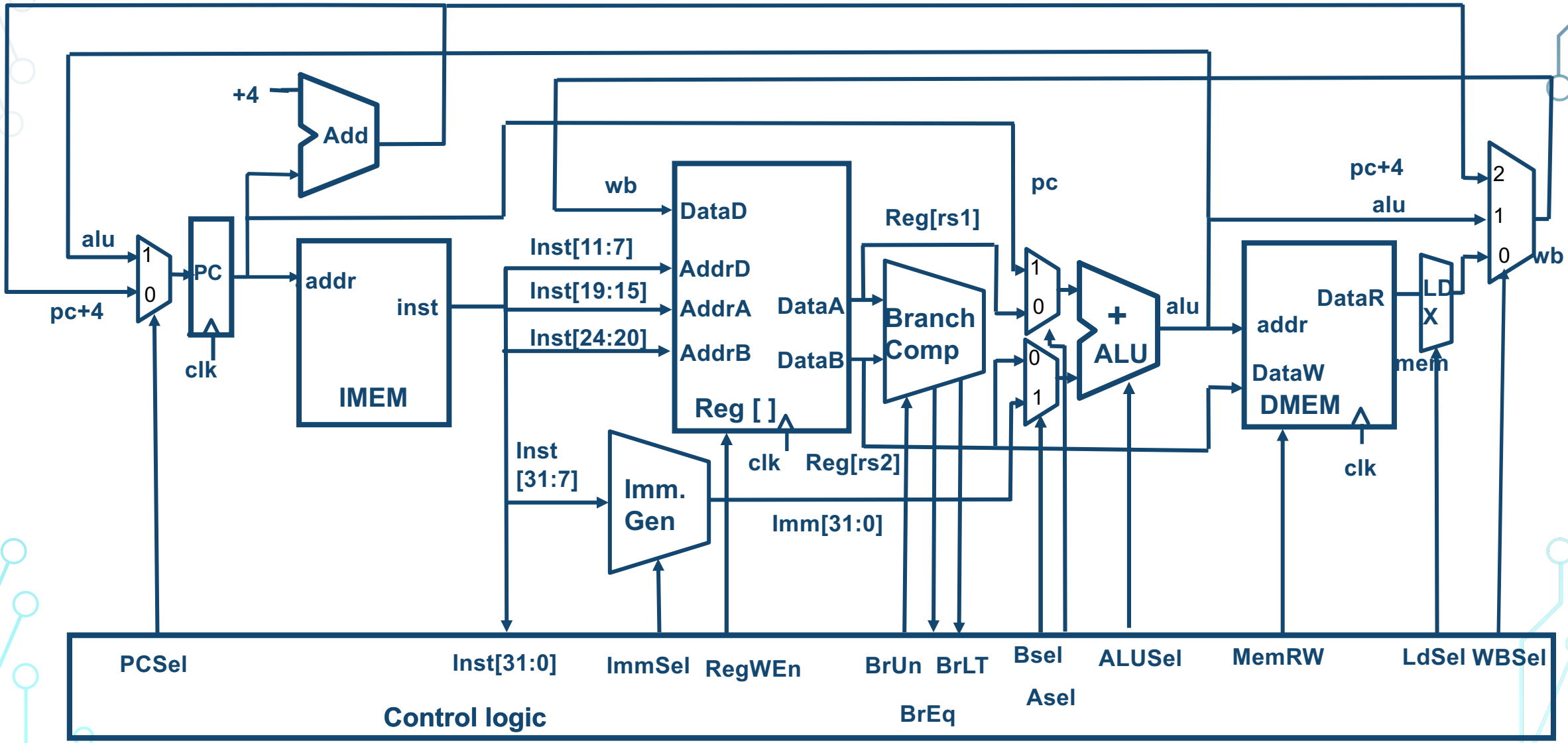
Implementing LUI



Implementing AUIPC



Complete RV32I Datapath!



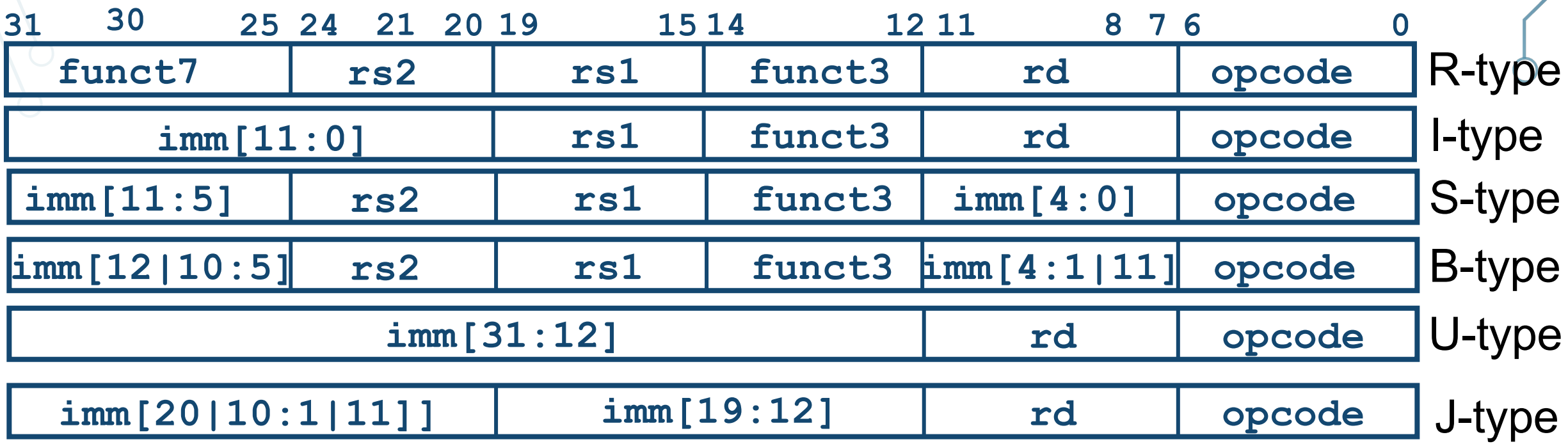
Recap: Complete RV32I ISA

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

- RV32I has 47 instructions
- 37 instructions are enough to run any C program

Summary of RISC-V Instruction Formats

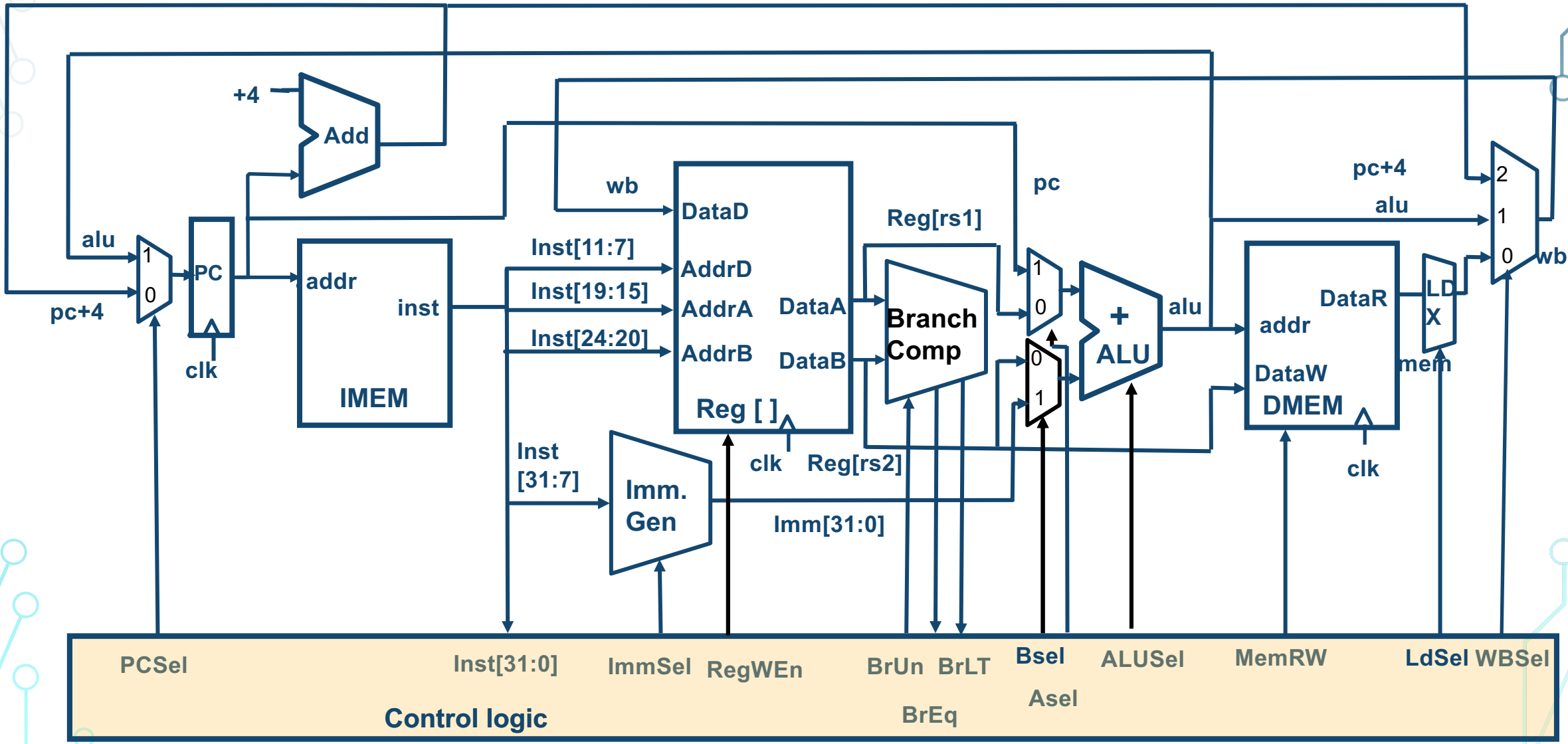


Administrivia

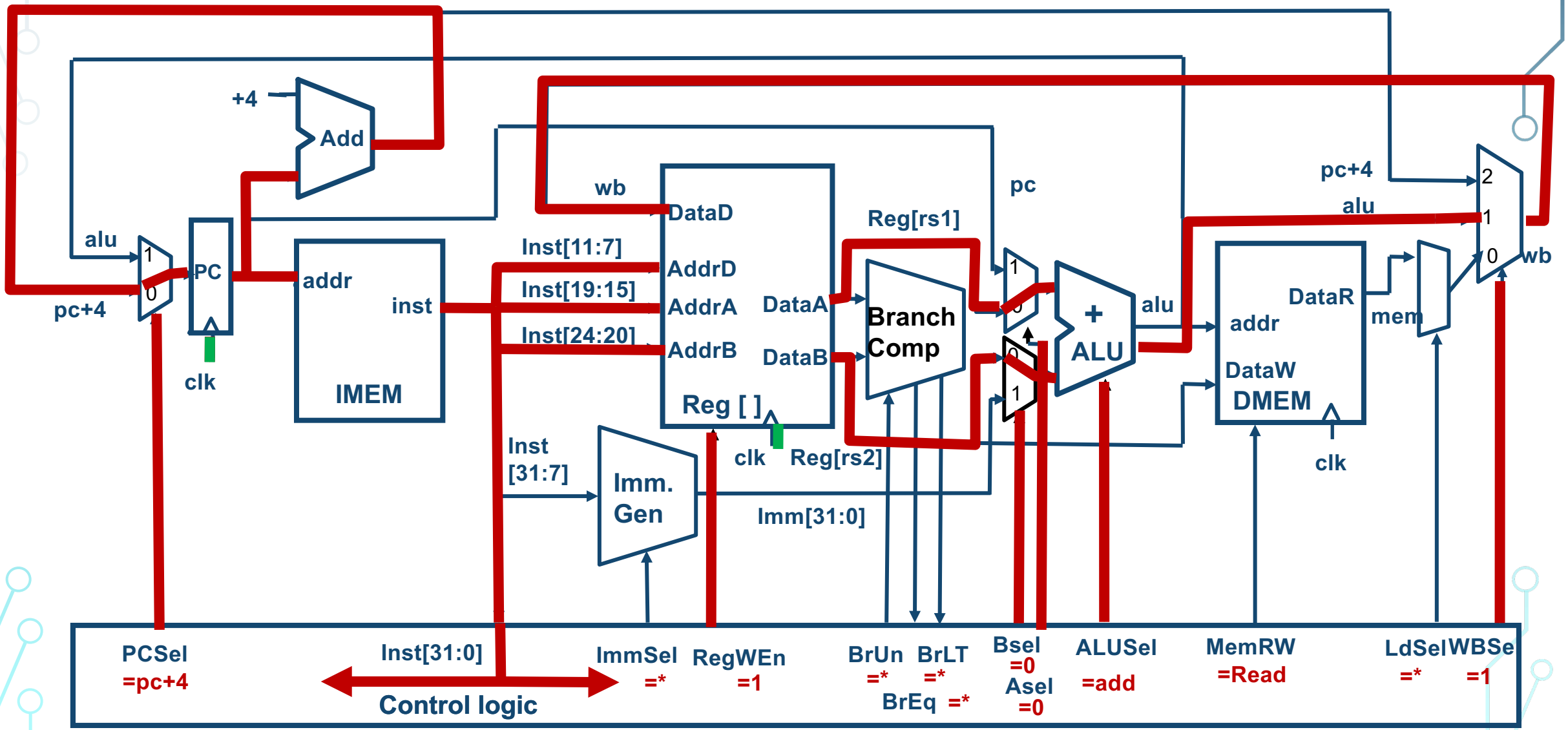
- Lab 4 starts this week.
  - 2-week lab
- HW 3 due this week.
  - HW 4 out

- **RISC-V Datapath & Control**
  - R-type
  - I-type
  - S-type
  - B-type
  - J-type
  - U-type
  - **Control Logic**

Complete RV32I Datapath with Control

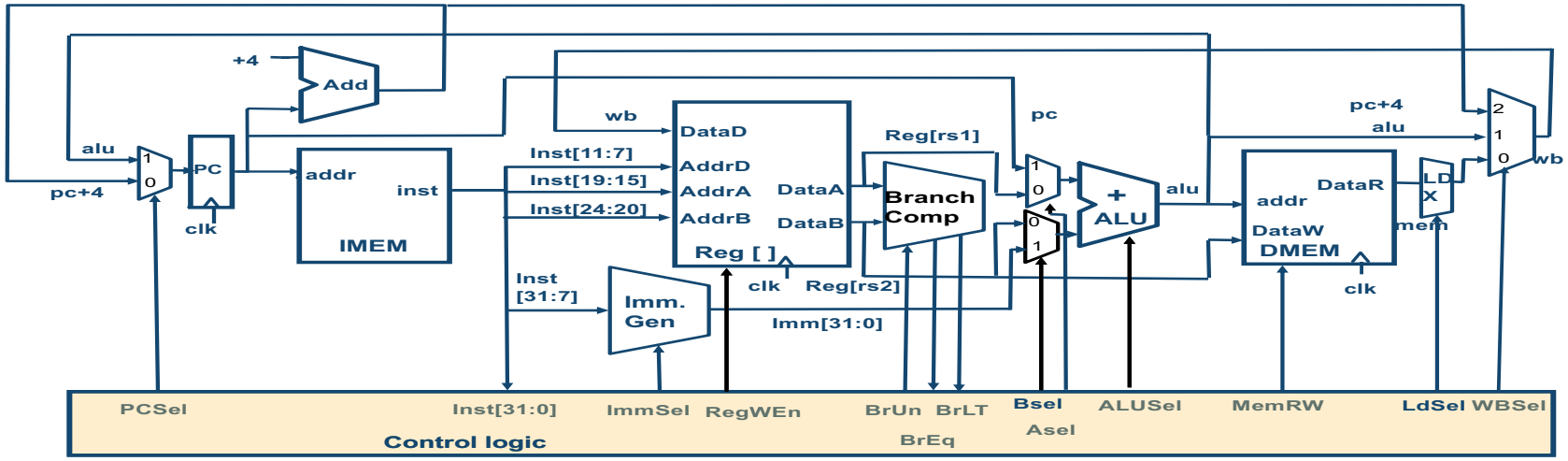


Example: add


+4
Add
alu
pc+4
PC
clk
addr
inst
IMEM
wb
DataD
AddrD
AddrA
AddrB
Reg [ ]
clk
DataA
DataB
Inst[11:7]
Inst[19:15]
Inst[24:20]
Inst
[31:7]
Imm.
Gen
Imm[31:0]
Reg[rs1]
Reg[rs2]
pc
Branch
Comp
+
ALU
alu
addr
DataW
DMEM
DataR
clk
mem
pc+4
alu
wb
PCSel
=pc+4
Inst[31:0]
Control logic
ImmSel
=*
RegWEn
=1
BrUn
=*
BrLT
=*
BrEq
=*
Bsel
=0
Asel
=0
ALUSel
=add
MemRW
=Read
LdSel
=*
WBSel
=1


31
25
24
20
19
15
14
12
11
7
6
0
funct7
rs2
rs1
funct3
rd
opcode
7
5
5
3
5
7

add Execution



Clock			
PC	1000	1004	
PC+4	1004	1008	
inst[31:0]	add x1,x2,x3	add x6,x7,x9	
Control logic	add control	add control	
Reg[rs1]	Reg[2]	Reg[7]	
Reg[rs2]	Reg[3]	Reg[9]	
alu	Reg[2]+Reg[3]	Reg[7]+Reg[9]	
wb	Reg[2]+Reg[3]	Reg[7]+Reg[9]	
Reg[1]	???	Reg[2]+Reg[3]	

Control Logic Truth Table

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

RV32I, a nine-bit ISA!

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

inst[30]

inst[14:12]

inst[6:2]

Instruction type encoded using only 9 bits inst[30],inst[14:12], inst[6:2]

Control Realization Options

- ROM
  - “Read-Only Memory”
  - Regular structure
  - Can be easily reprogrammed
    - fix errors
    - add instructions
- Combinatorial Logic
  - Decoder is typically hierarchical
    - First decode opcode, and figure out instruction type
    - E.g. branches are Inst[6:2] = 11000
    - Then determine the actual instruction
    - Inst[30] + Inst[14:12]
  - Modularity helps simplify and speed up logic
    - Narrow problem space for logic synthesis

Combinational Logic Control

- Simplest example: BrUn

inst[14:12] inst[6:2]

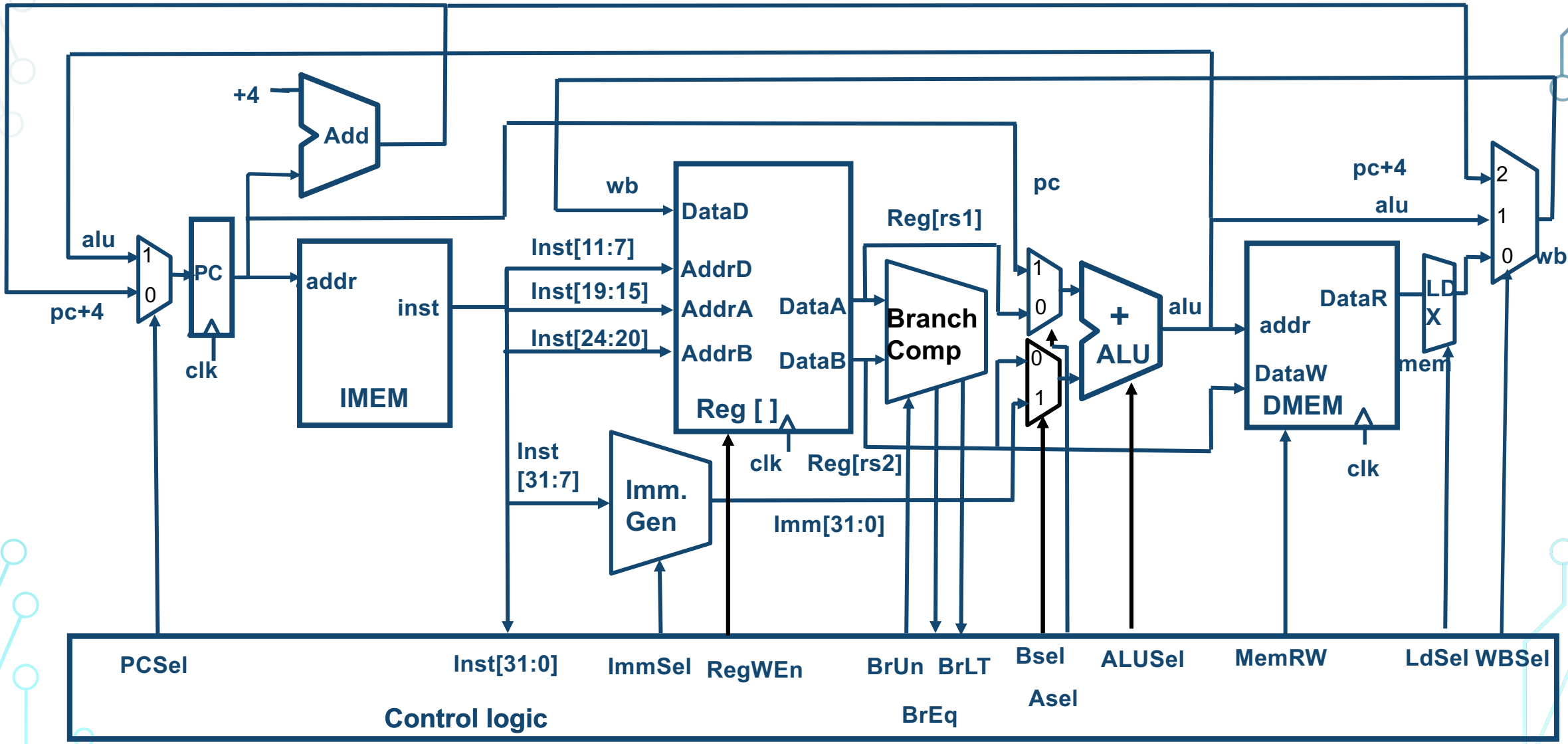
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

inst[14:13]

inst[12]	00	01	11	10
0				
1				

- **How to decode whether BrUn is 1?**
  - **Branch = Inst[6] • Inst[5] • !Inst[4] • !Inst[3] • !Inst[2]**
  - **BrUn = Inst [13] • Branch**

Complete RV32I Datapath with Control



Summary

- We have covered the implementation of the base ISA for RV32I!!!
  - Get yourself familiar with the ISA Spec.
- Instruction type:
  - R-type
  - I-type
  - S-type
  - B-type
  - J-type
  - U-type
- Implementation suggested is straightforward, yet there are modalities in how to implement it well at gate level.
- Single-cycle datapath is slow – need to pipeline it