EECS151 : Introduction to Digital Design and ICs

Lecture 8 – RISC-V Datapath and Control

Bora Nikolić and Sophia Shao

Dual-Core RISC-V 64Bit K210 AI Board – Kendryte KD233

The Kendryte K210 is a system-on-chip (SoC) that integrates machine vision and machine hearing. Using TSMC's ultra-low-power 28-nm advanced process with dual-core 64-bit processors for better power efficiency, stability and reliability.

https://www.analoglamb.com/product/dual-core-risc-v-64bit-k210-ai-board-kendryte-kd233/

Review

- State machines:
  - Specify circuit function
  - Draw state transition diagram
  - Write down symbolic state-transition table
  - Assign encodings (bit patterns) to symbolic states
  - Code as Verilog behavioral description
- RISC-V processor
  - A large state machine
  - Datapath + control
  - Reviewed R-format instructions

Implementing other R-Format instructions

0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

- All implemented by decoding funct3 and funct7 fields and selecting appropriate ALU function

Instruction Encoding

- Instructions are encoded to simplify logic
  - **`sub`** and **`sra`** differ in Inst[30] from **`add`** and **`srl`**
- **`RV64I`** widens registers (XLEN=64)
- Additional instructions manipulate 32-bit values, identified by a suffix W
  - **`ADDW`, `SUBW`**
  - RV64I opcode field for 'W' instructions is **`0111011`** (**`0110011`** for RV32I)

32b

0000000	rs2	rs1	000	rd	0111011	**addw**
0000000	rs2	rs1	000	rd	0110011	**add**

64b

I-Format Instruction Layout

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

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

All RV32 I-format Arithmetic Instructions

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

The same Inst[30] immediate bit is used to distinguish “shift right logical” (SRLI) from “shift right arithmetic” (SRAI)

“Shift-by-immediate” instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

Implementing I-Format - `addi` instruction

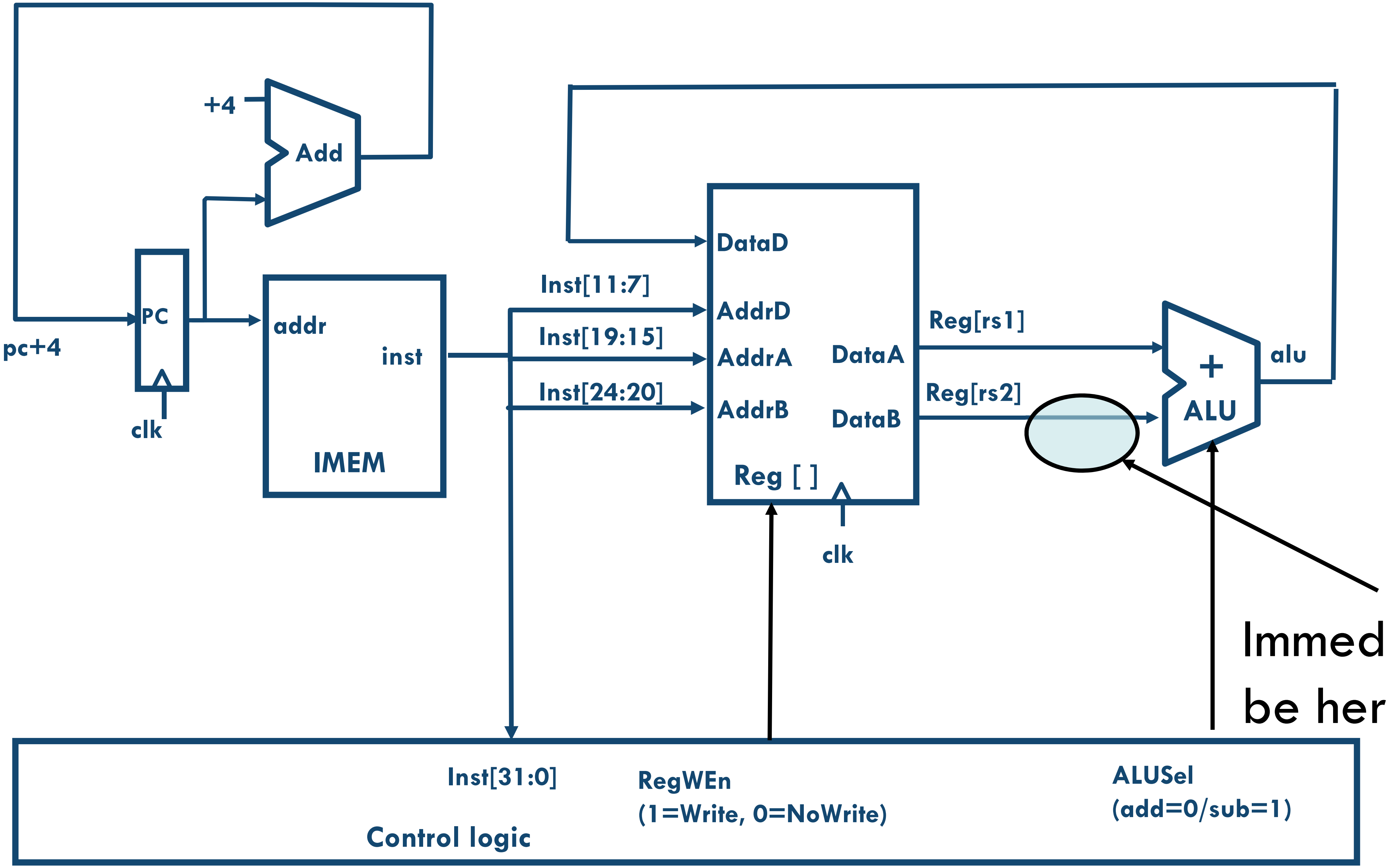
- **RISC-V Assembly Instruction – add immediate:**

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

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7

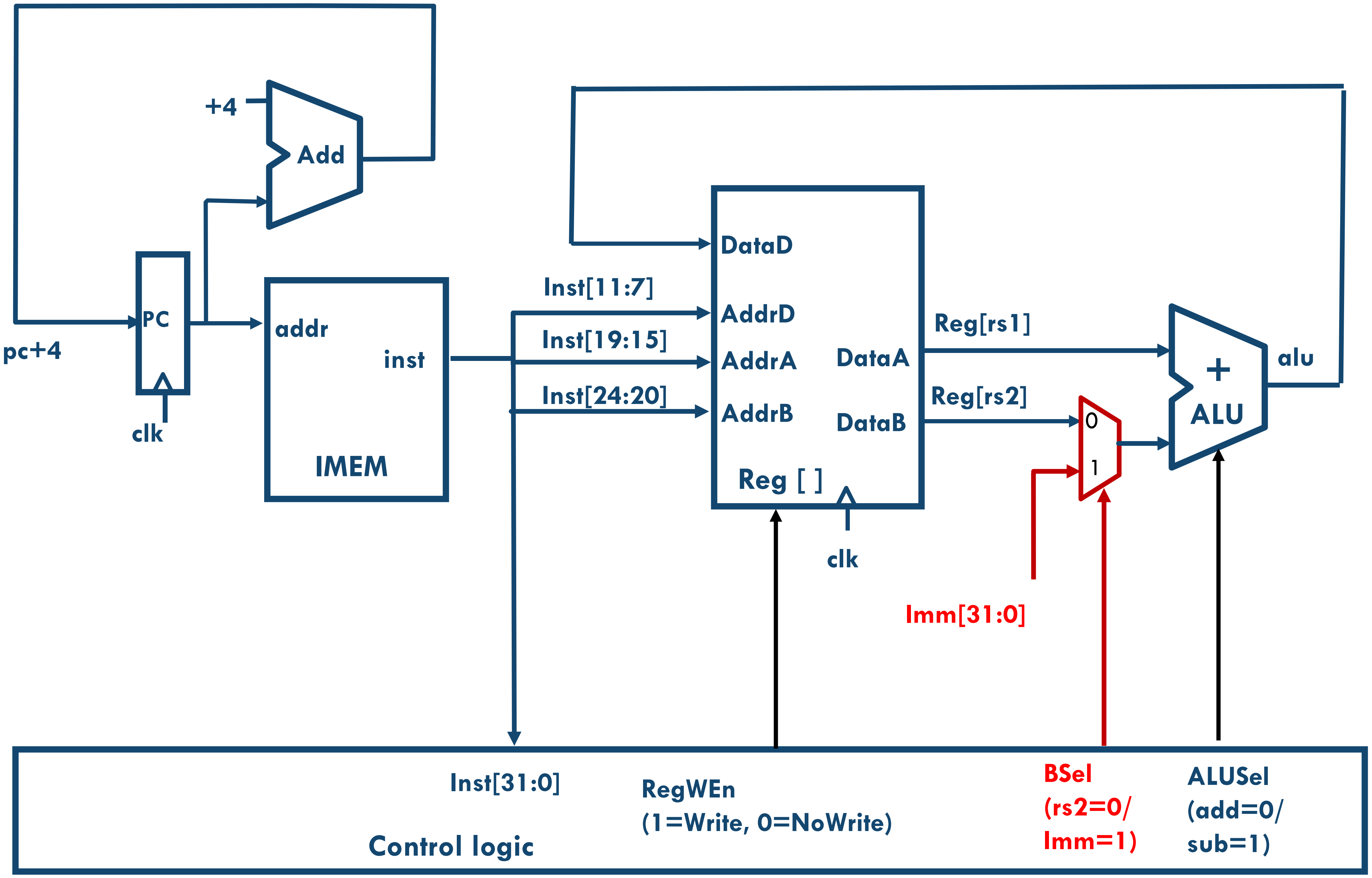
111111001110	00001	000	01111	0010011
imm=-50	rs1=1	add	rd=15	OP-Imm

Datapath for add/sub

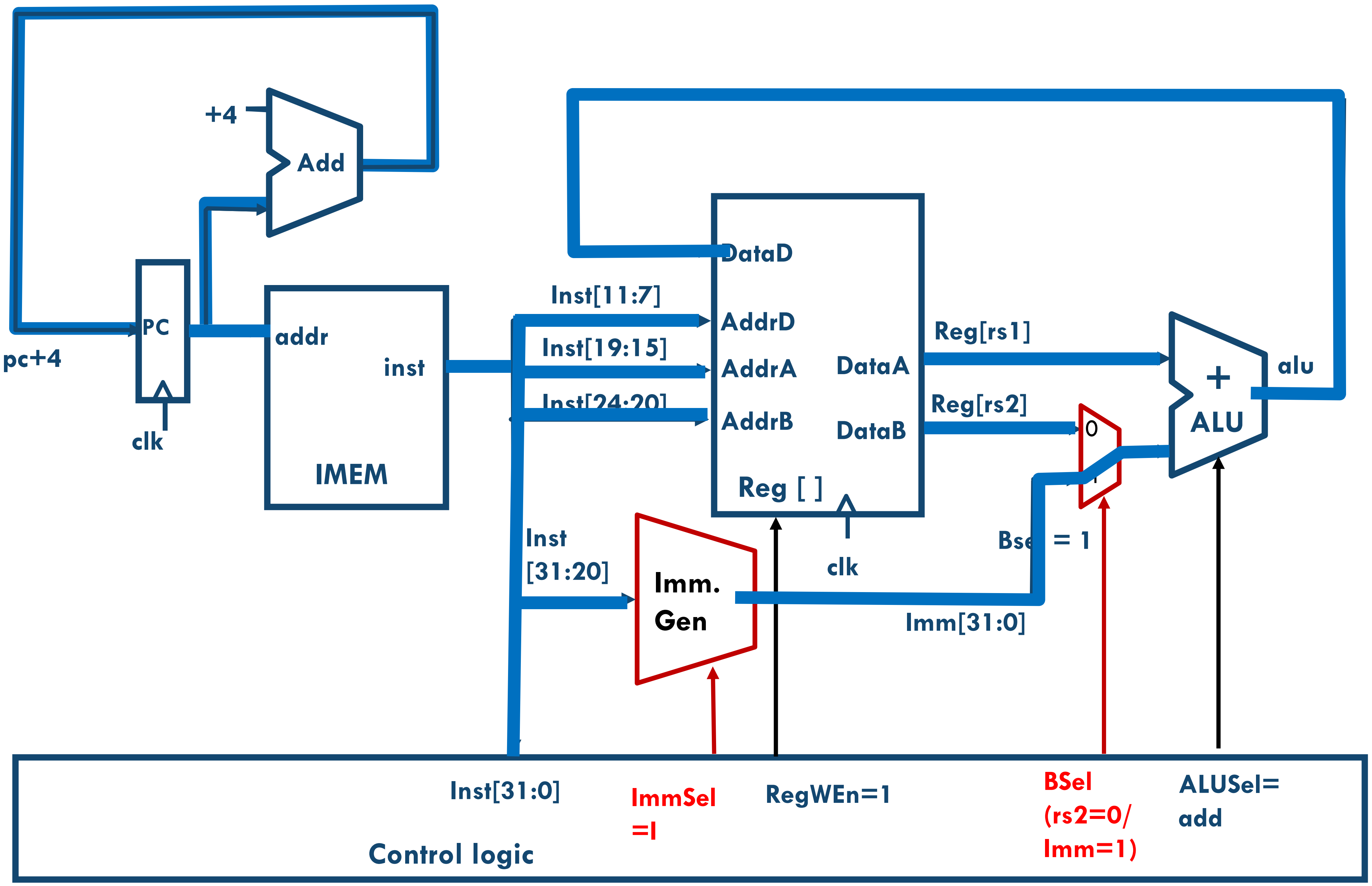


Immediate should be here

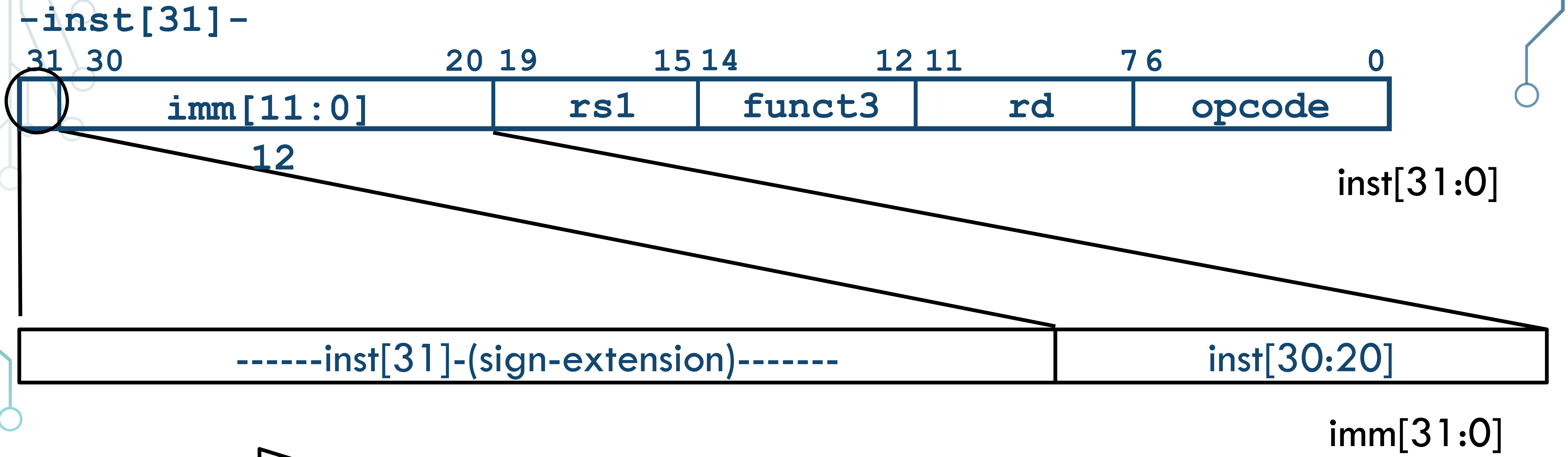
Adding `addi` to Datapath

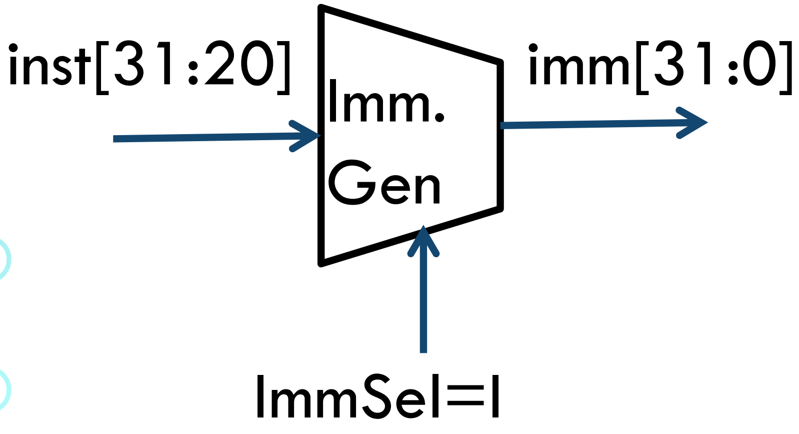


Adding `addi` to Datapath



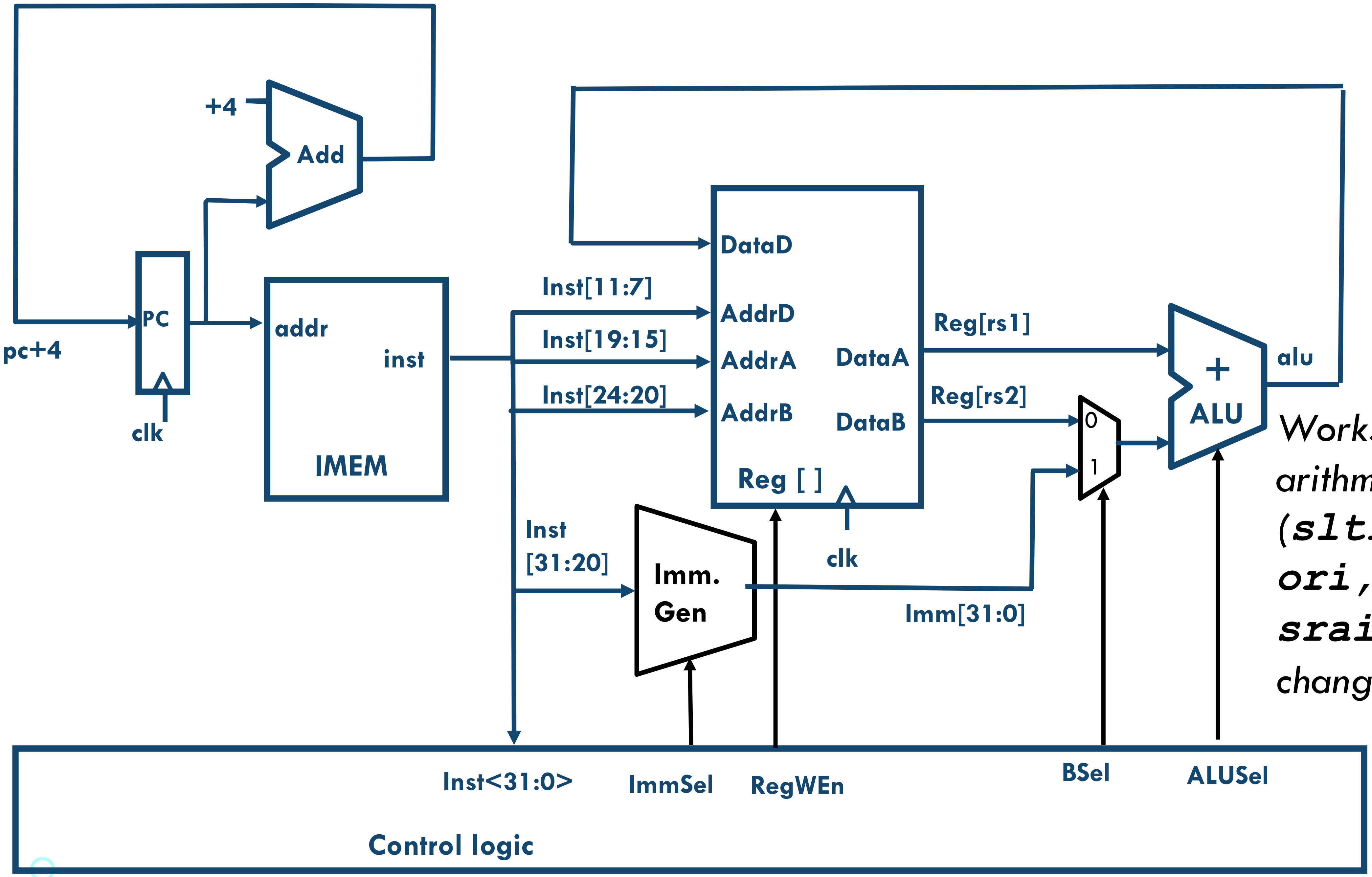
I-Format immediates





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

R+I Datapath



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

Load Instructions are also I-Type

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

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

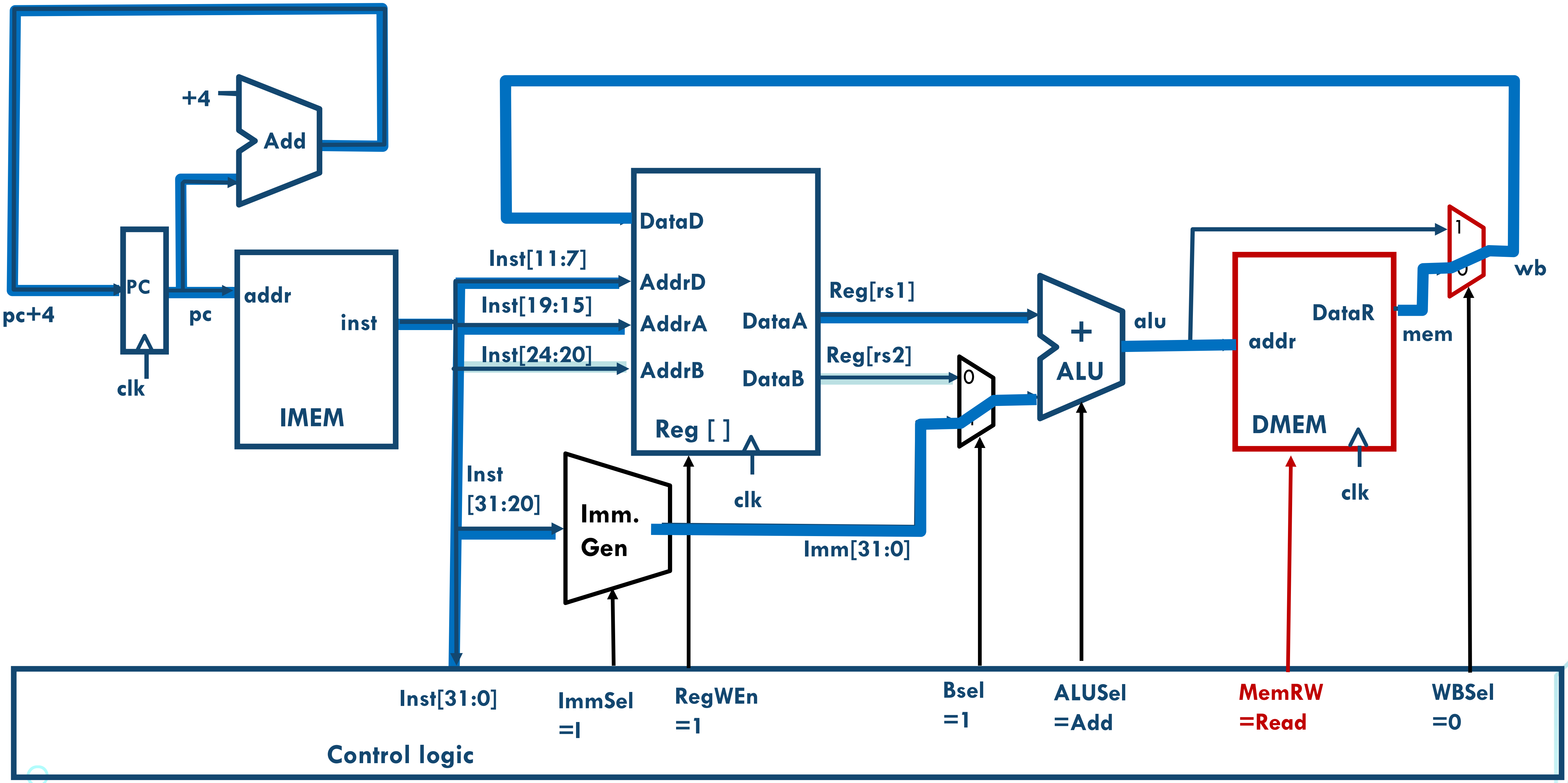
Add lw to Datapath

- RISC-V Assembly Instruction (I-type): `lw x14, 8(x2)`

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD
31 20	19 15	14 12	11 7	6 0
000000001000	00010	010	01110	0000011
imm= +8	rs1=2	LW	rd=14	LOAD

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

Adding `lw` to Datapath



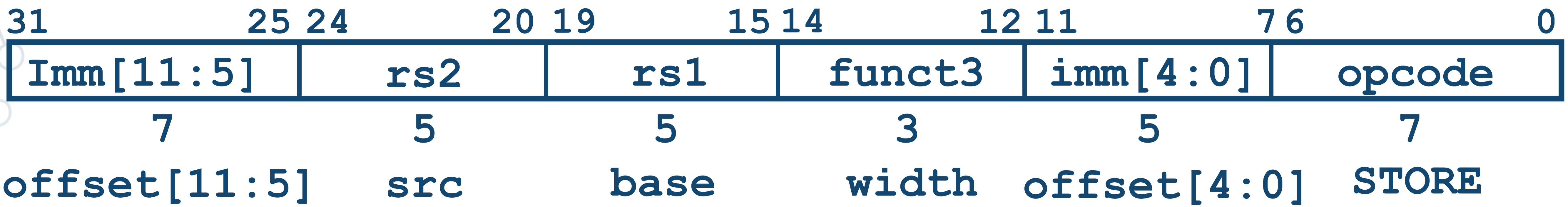
All RV32 Load Instructions

imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	001	rd	0000011	lh
imm[11:0]	rs1	010	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu

funct3 field encodes size and 'signedness' of load data

- Supporting the narrower loads requires additional logic to extract the correct byte/halfword from the value loaded from memory, and sign- or zero-extend the result to 32 bits before writing back to register file.
  - It is just a mux for load extend, similar to sign extension for immediates

S-Format Used for Stores

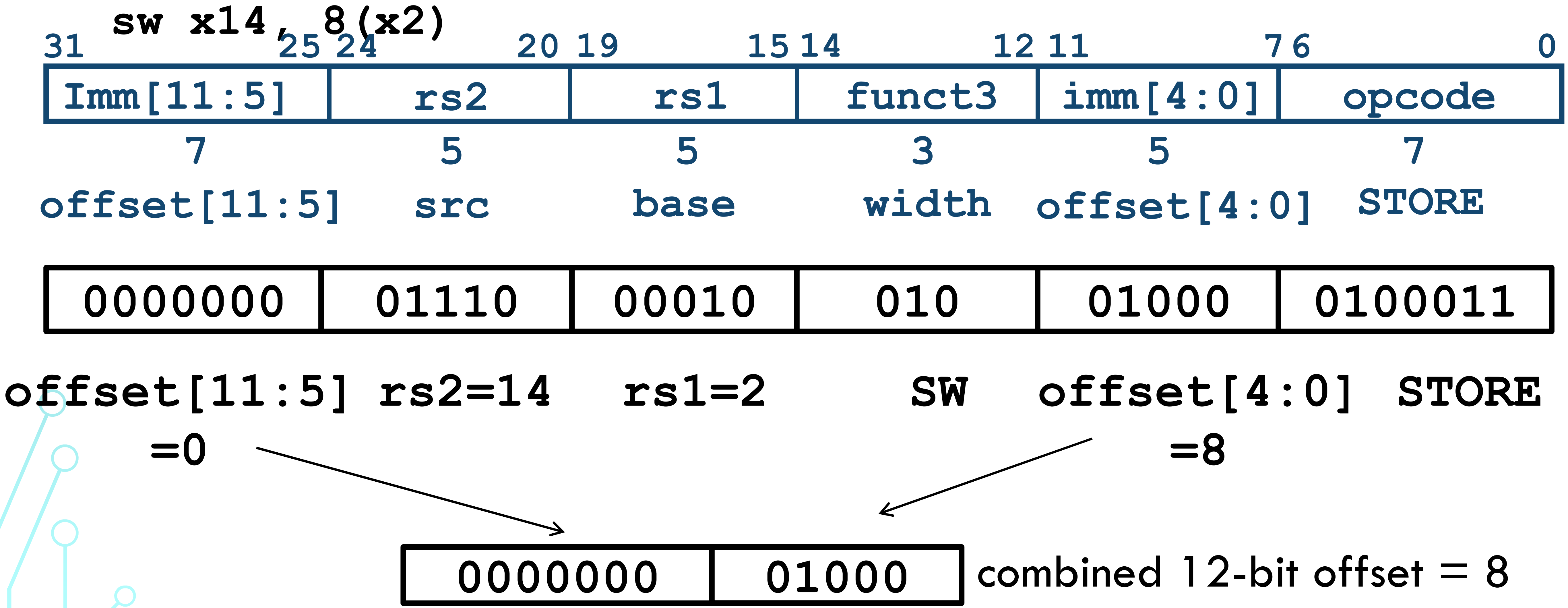


31 25	24 20	19 15	14 12	11 7	6 0
Imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

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

Adding `sw` Instruction

- sw: Reads two registers, rs1 for base memory address, and rs2 for data to be stored, as well immediate offset!



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

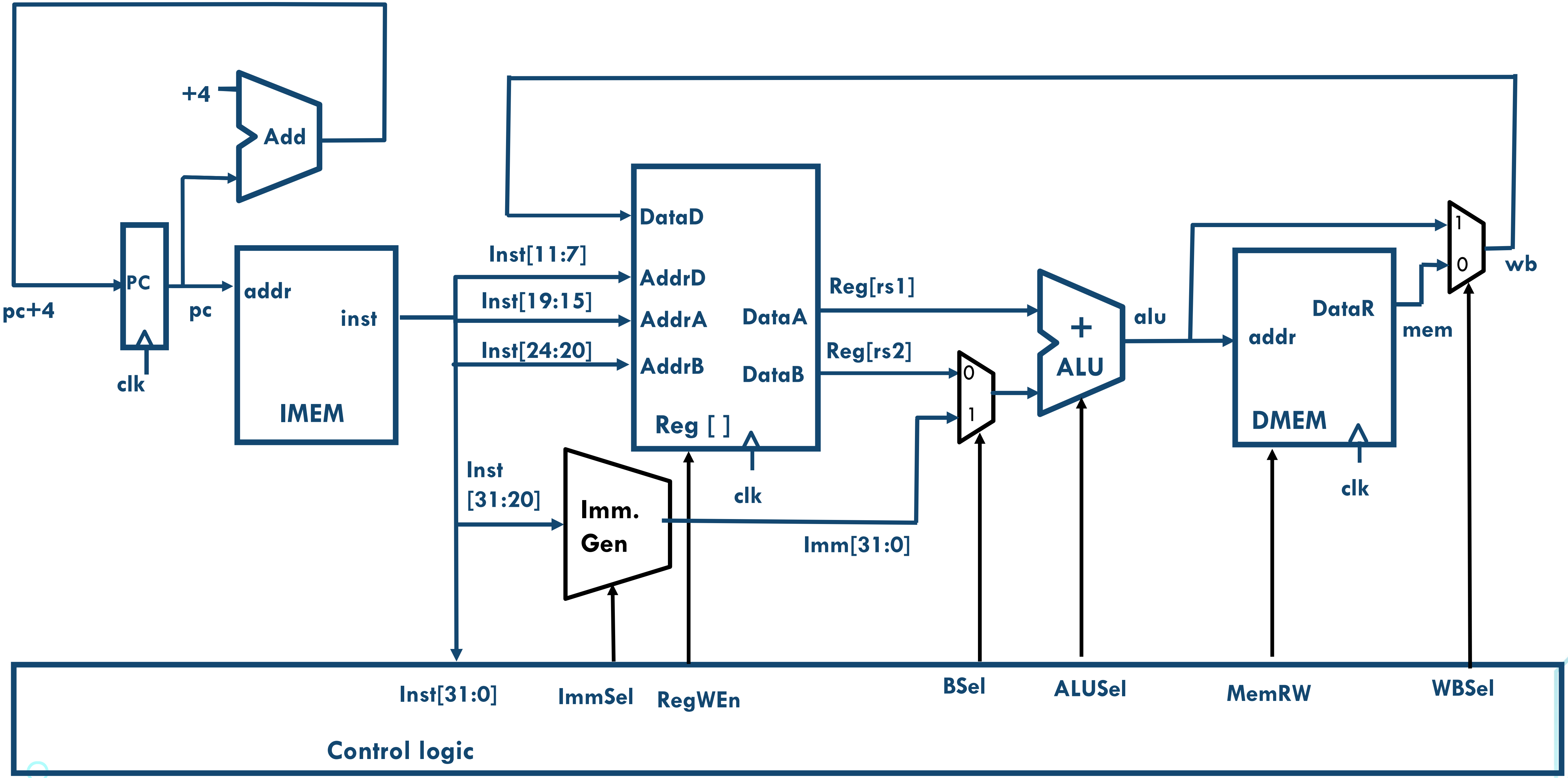
31 25	24 20	19 15	14 12	11 7	6 0
Imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

0000000	01110	00010	010	01000	0100011
offset[11:5] =0	rs2=14	rs1=2	SW	offset[4:0] =8	STORE

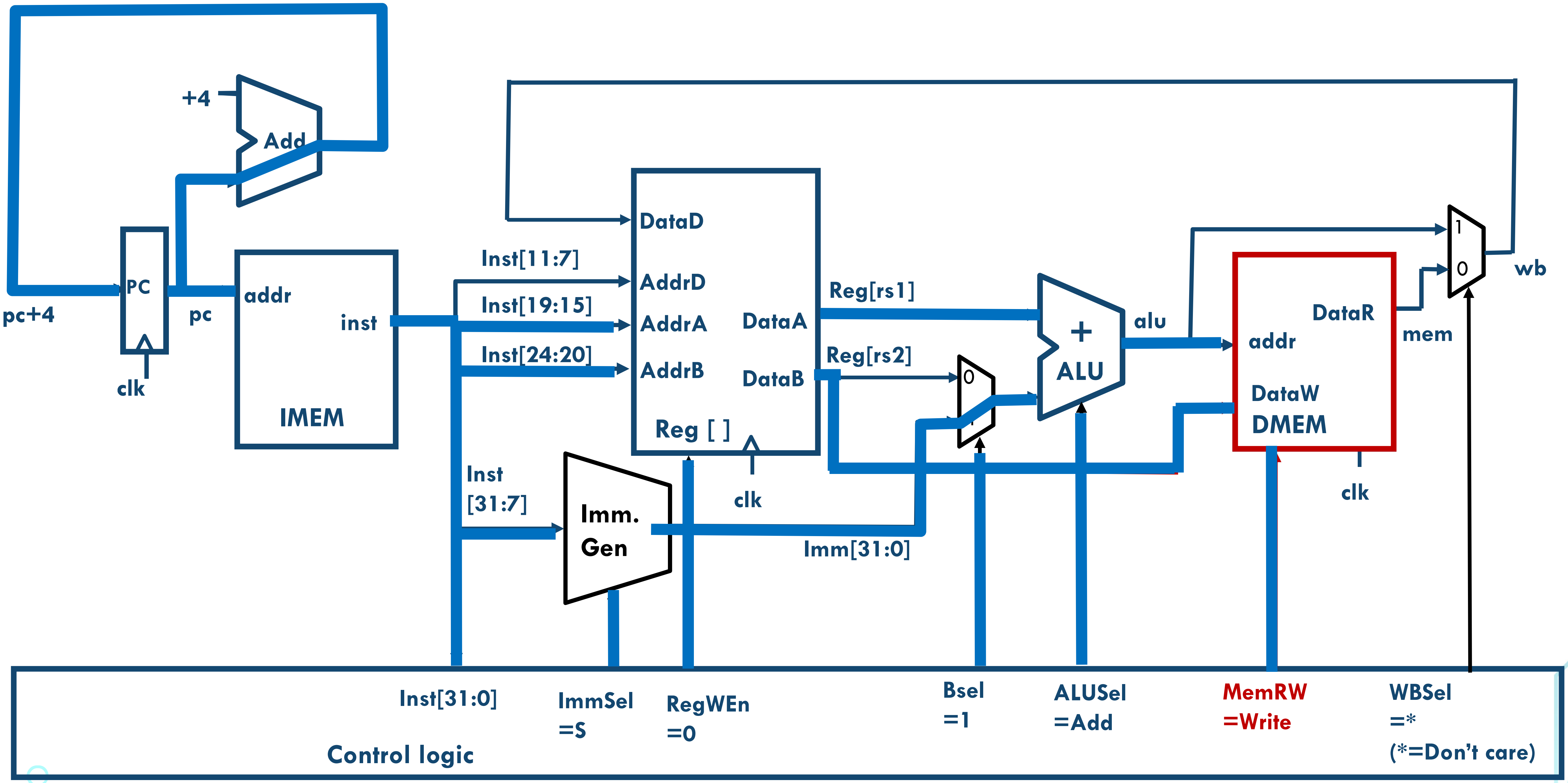
0000000	01000

combined 12-bit offset = 8

Datapath with `lw`



Adding sw to Datapath



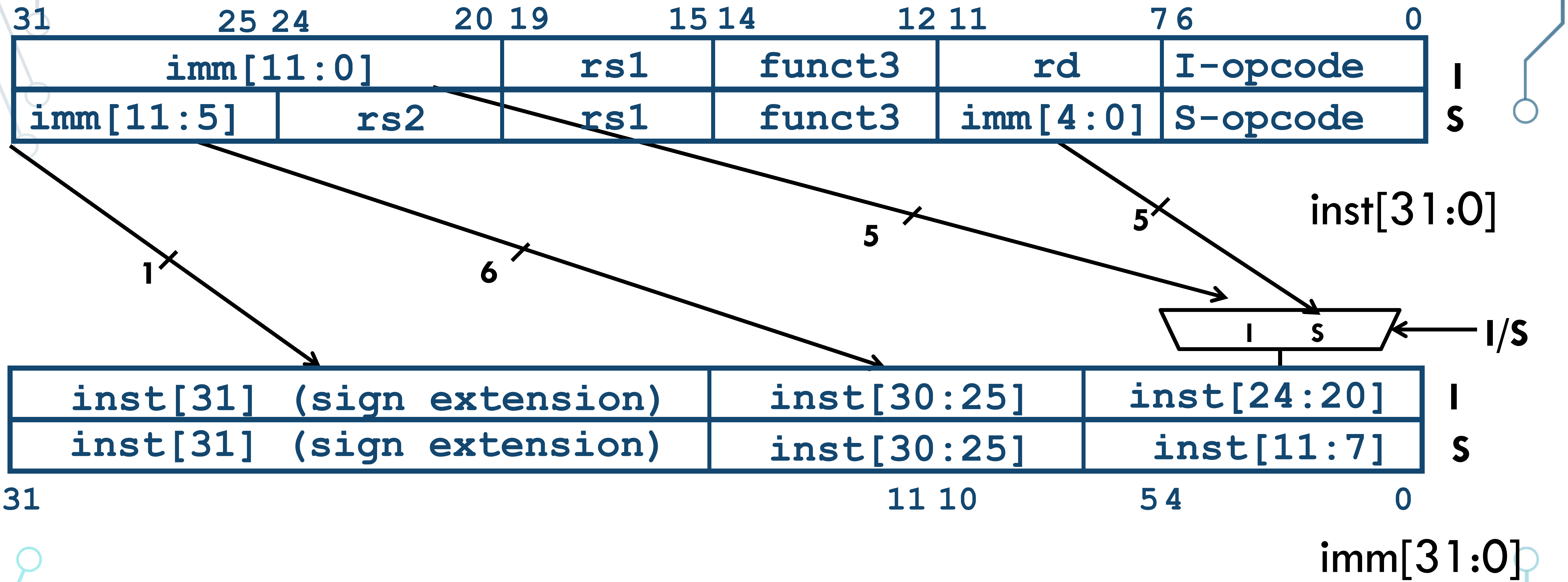
All RV32 Store Instructions

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

width

- Store byte, halfword, word

I+S Immediate Generation



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

Administrivia

- Midterm 1: October 2, in class
  - Covers material up to today
  - One double-sided page of hand-written notes
  - Review session on Monday
- Midterm 2: November 6

B-Format - RISC-V Conditional Branches

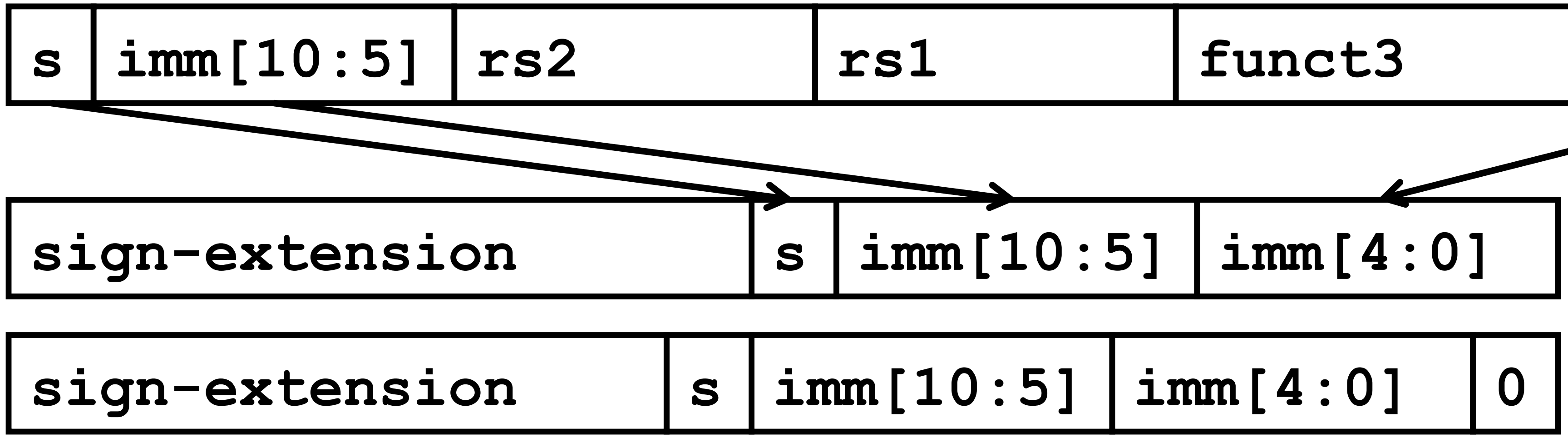
- E.g., **`BEQ x1, x2, Label`**
- Branches read two registers but don't write a register (similar to stores)
- How to encode label, i.e., where to branch to?

RISC-V Feature, n×16-bit instructions

- Extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 16-bits in length
- To enable this, RISC-V scales the branch offset by 2 bytes even when there are no 16-bit instructions
- Reduces branch reach by half and means that ½ of possible targets will be errors on RISC-V processors that only support 32-bit instructions (as used in this class)
- RISC-V conditional branches can only reach $\pm 2^{10} \times$ 32-bit instructions on either side of PC

RISC-V Branch Immediates

- 12-bit immediate encodes PC-relative offset of -4096 to +4094 bytes in multiples of 2 bytes
- RISC-V approach: keep 11 immediate bits in fixed position in output value, and rotate LSB of S-format to be bit 12 of B-format



s	imm[10:5]	rs2	rs1	funct3	imm[4:0]	B-opcode

sign-extension	s	imm[10:5]	imm[4:0]	
sign-extension	s	imm[10:5]	imm[4:0]	0

S-Immediate

B-Immediate (shift left by 1)

Only one bit changes position between S and B, so only need a single-bit 2-way mux

RISC-V Immediate Encoding

Instruction encodings, inst[31:0]

31 30 25	24 20	19 15	14 12	11 8 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type

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

31 25 24 12	11	10 5	4 1	0	
-inst[31]-		inst[30:25]	inst[24:21]	inst[20]	I-imm.
-inst[31]-		inst[30:25]	inst[11:8]	inst[7]	S-imm.
-inst[31]-	inst[7]	inst[30:25]	inst[11:8]	0	B-imm.

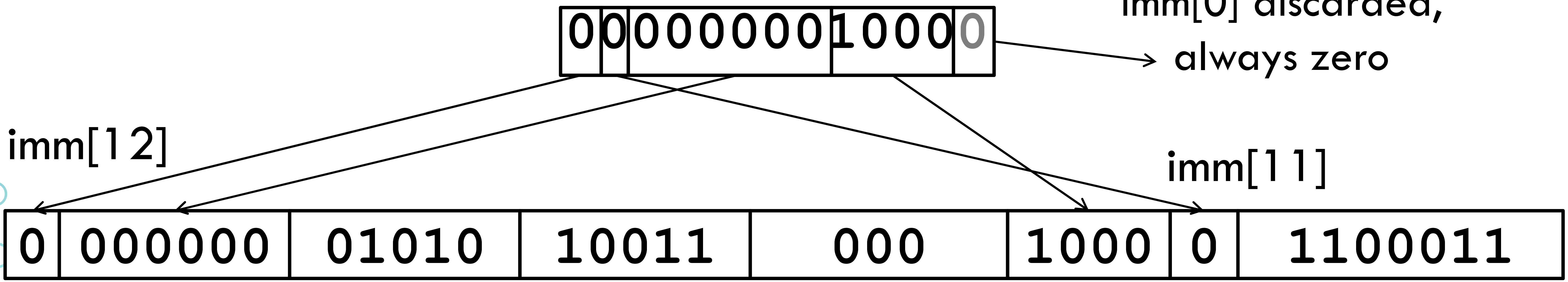
Upper bits sign-extended from inst[31] always

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

Branch Example, complete encoding

beq x19,x10, offset = 16 bytes

13-bit immediate, imm[12:0], with value 16



0	0	000000	1000	0

imm[0] discarded, always zero

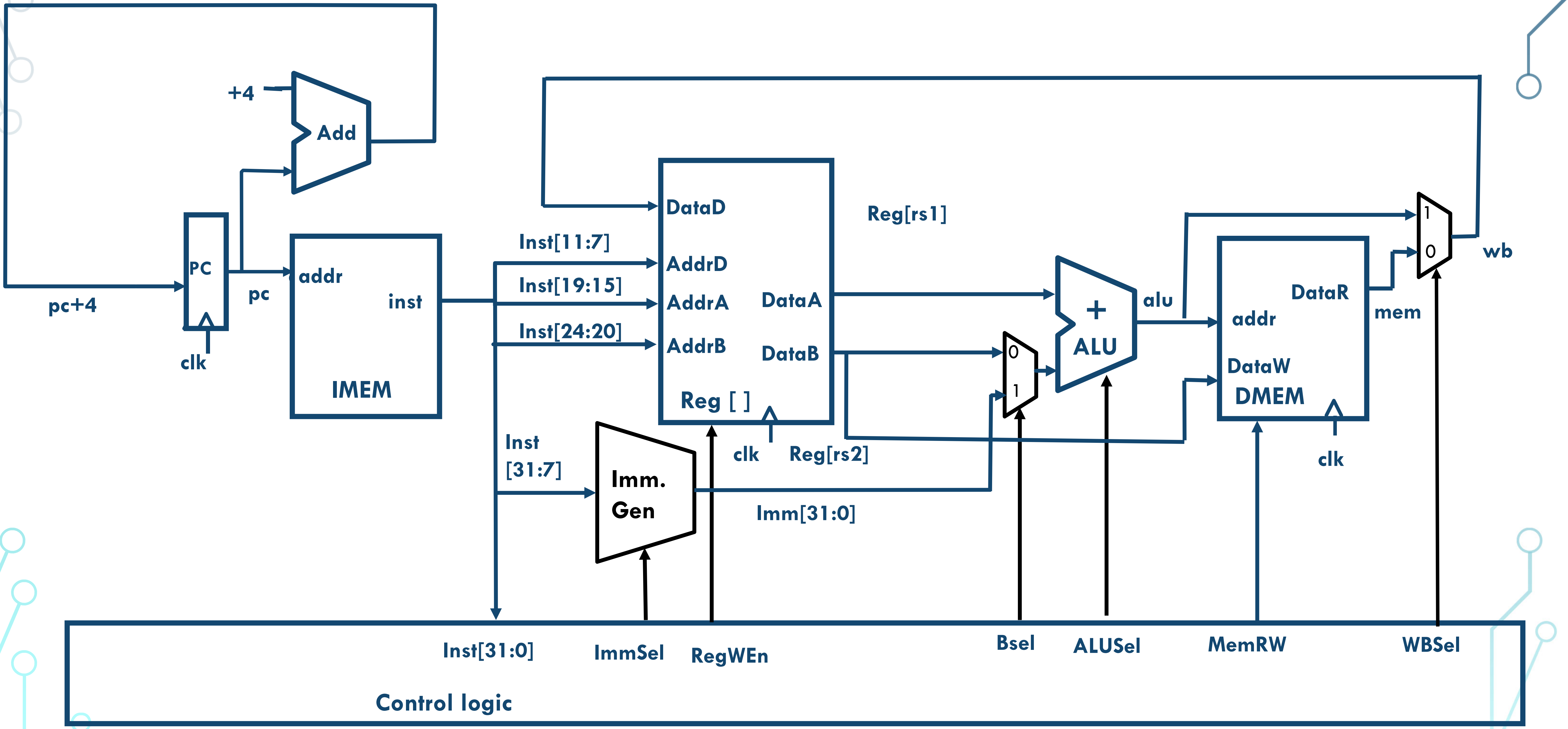
imm[12]						imm[11]	
0	000000	01010	10011	000	1000	0	1100011
	imm[10:5]	rs2=10	rs1=19	BEQ	imm[4:1]		BRANCH

Implementing Branches

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		rs2	rs1	funct3	offset[4:1|11]		BRANCH

- B-format is mostly same as S-format, with two register sources (rs1/rs2) and a 12-bit immediate
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode *even* 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

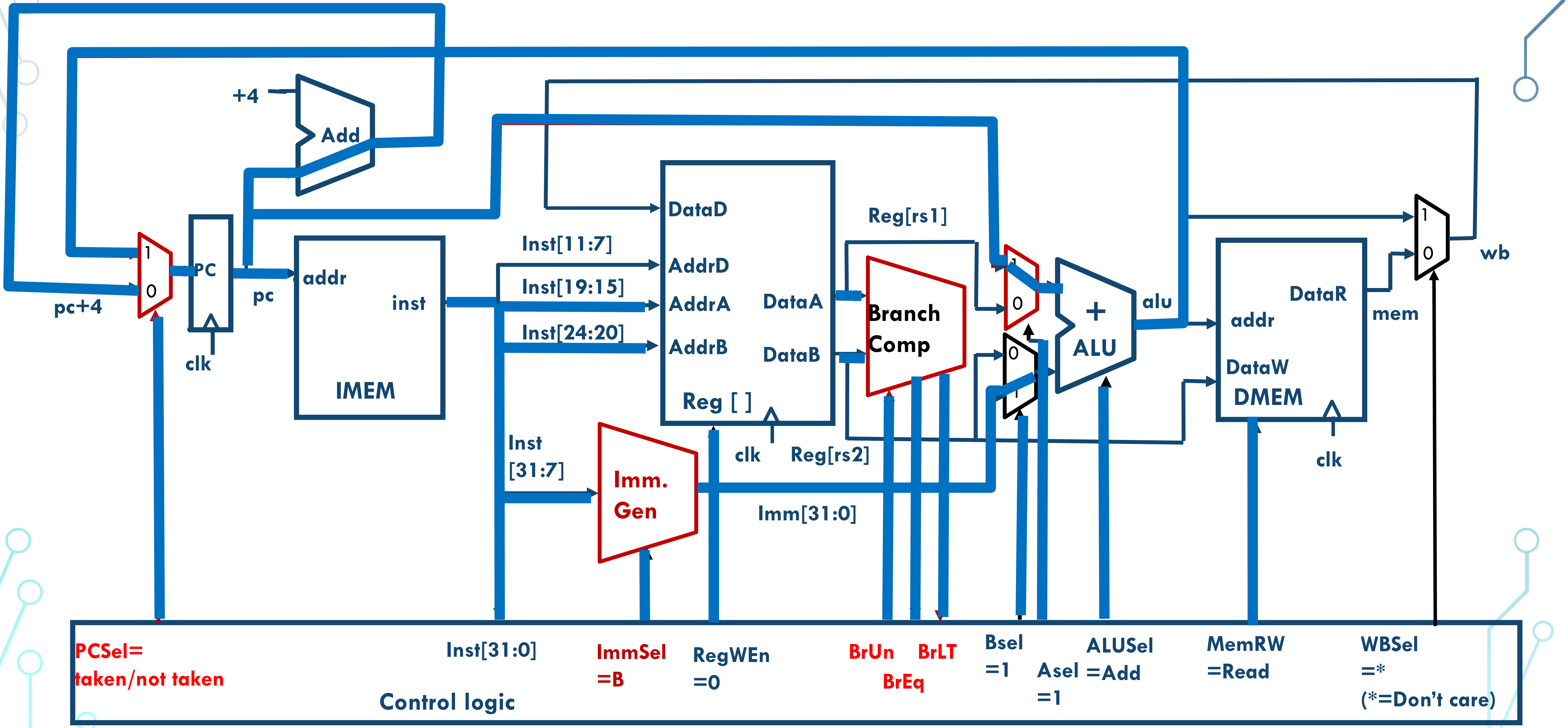
Datapath So Far



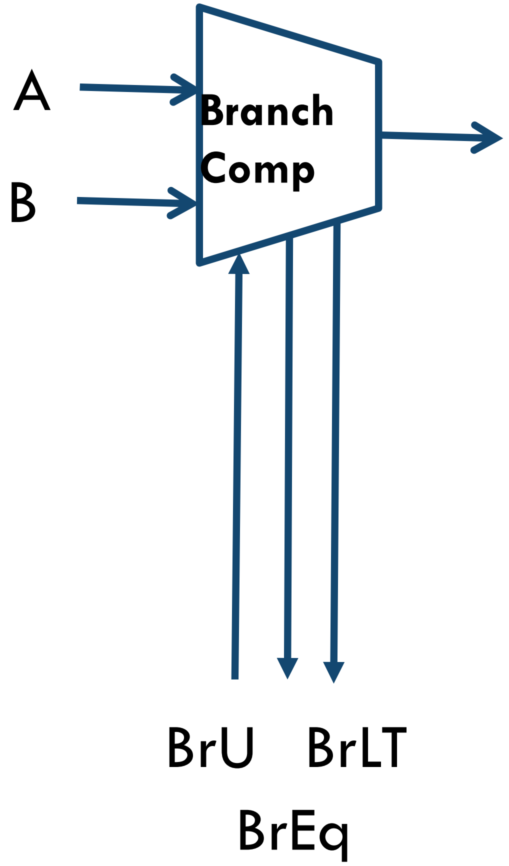
To Add Branches

- Different change to the state:
  - $\mathbf{PC} = \begin{cases} \mathbf{PC + 4,} & \mathbf{branch\ not\ taken} \\ \mathbf{PC + immediate,} & \mathbf{branch\ taken} \end{cases}$
- 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

Adding Branches



Branch Comparator



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

All RISC-V Branch Instructions

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

Peer Instruction

1) We should use the main ALU to evaluate branches in order to save some gates in small cores

2) Program counter is a register

3) The ALU is a sequential element

123
FFF
FFT
FTF
FTT
TFF
TFT
TTF
TTT

J-Format for Jump Instructions

31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
offset[20:1]				dest	JAL

- JAL 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

JALR Instruction (I-Format)

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	func3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

- JALR rd, rs, immediate
  - Writes PC+4 to rd (return address)
  - Sets PC = rs1 + immediate (and sets the LSB to 0)
  - Uses same immediates as arithmetic and loads
    - *no* multiplication by 2 bytes
    - In contrast to branches and JAL

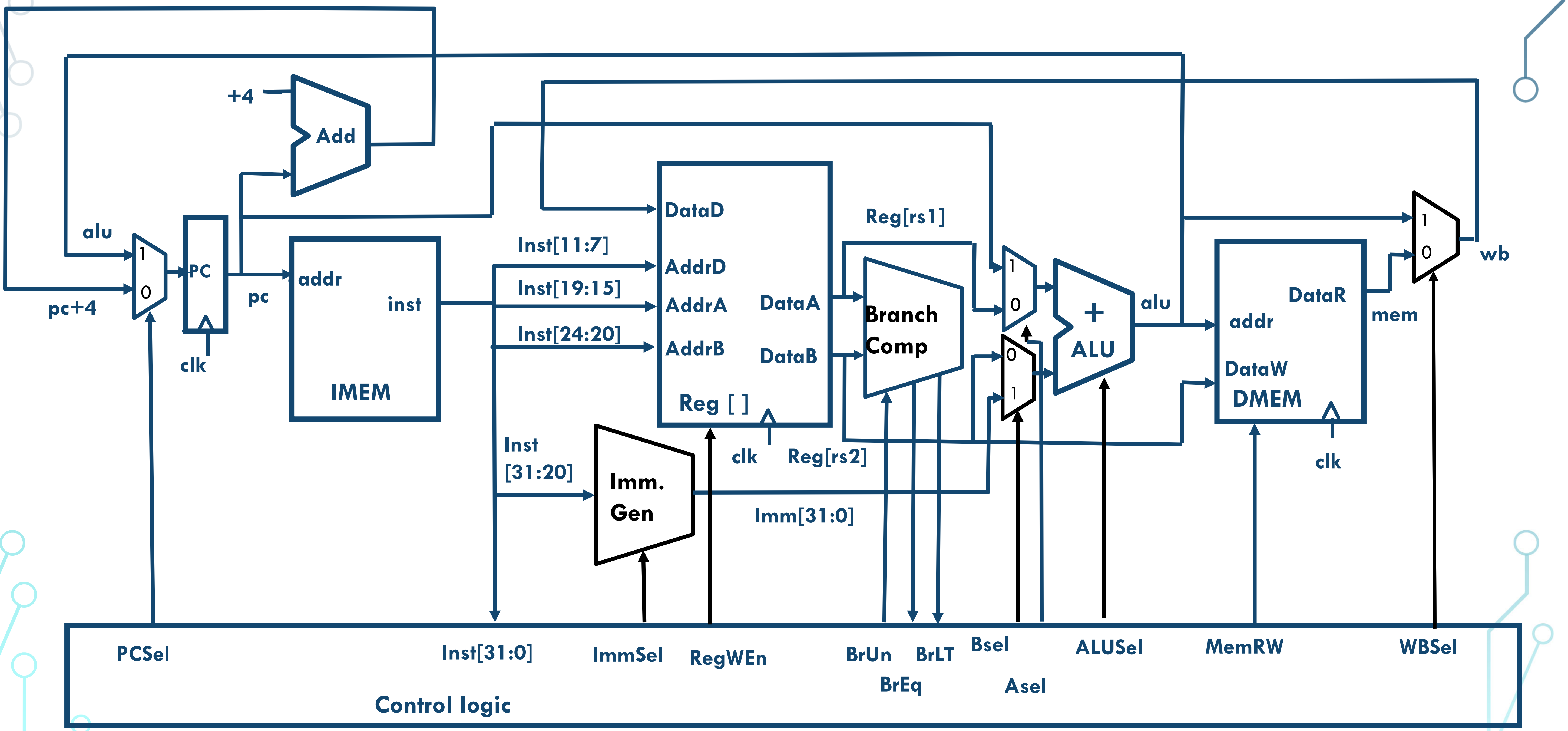
Otherwise, we would have yet another new encoding

Let’s Add JALR (I-Format)

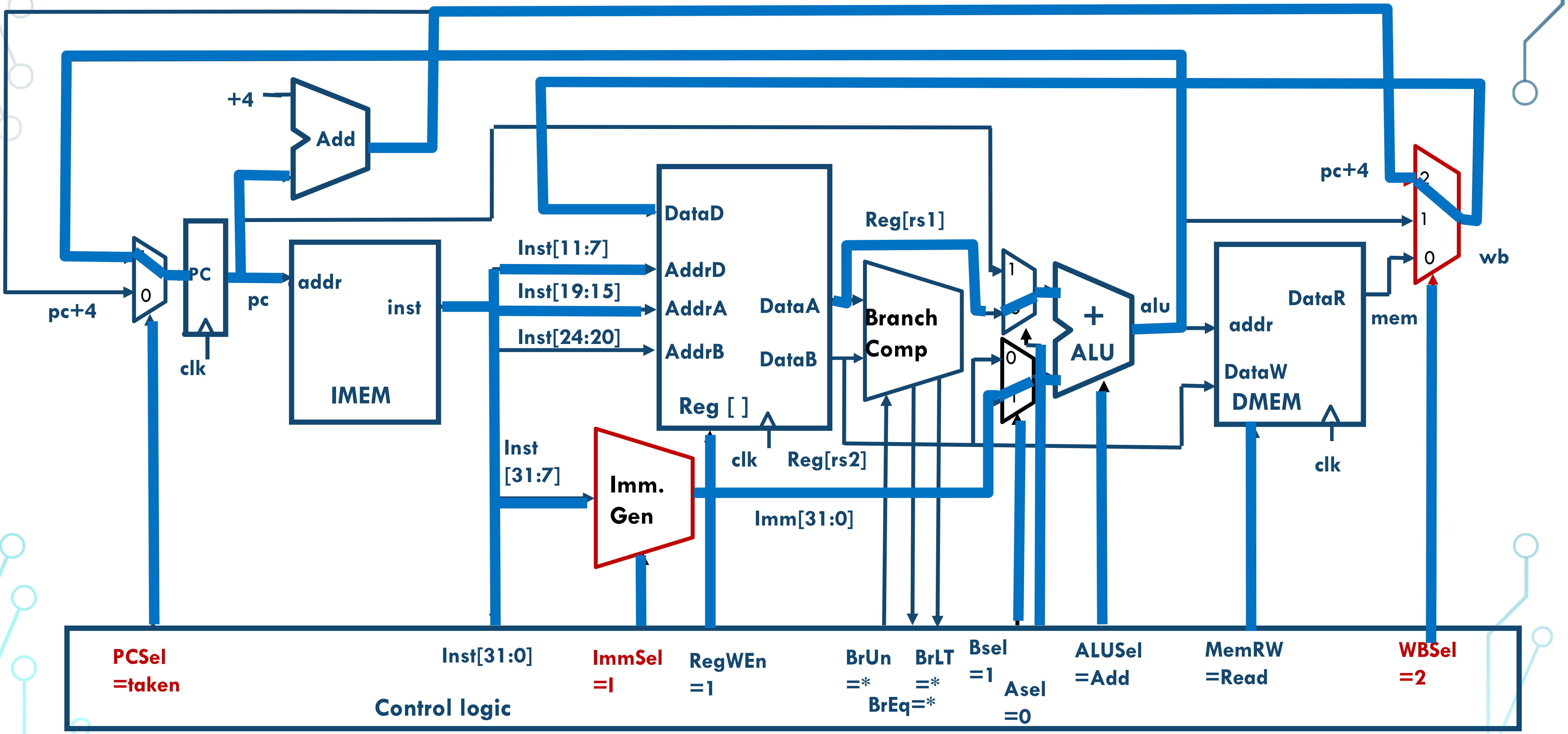
31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	func3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

- JALR rd, rs, immediate
- Two changes to the state
  - Writes PC+4 to rd (return address)
  - Sets PC = rs + immediate
  - Uses same immediates as arithmetic and loads
    - *no* multiplication by 2 bytes
    - LSB is ignored

Datapath So Far, with Branches



Adding **JALR**

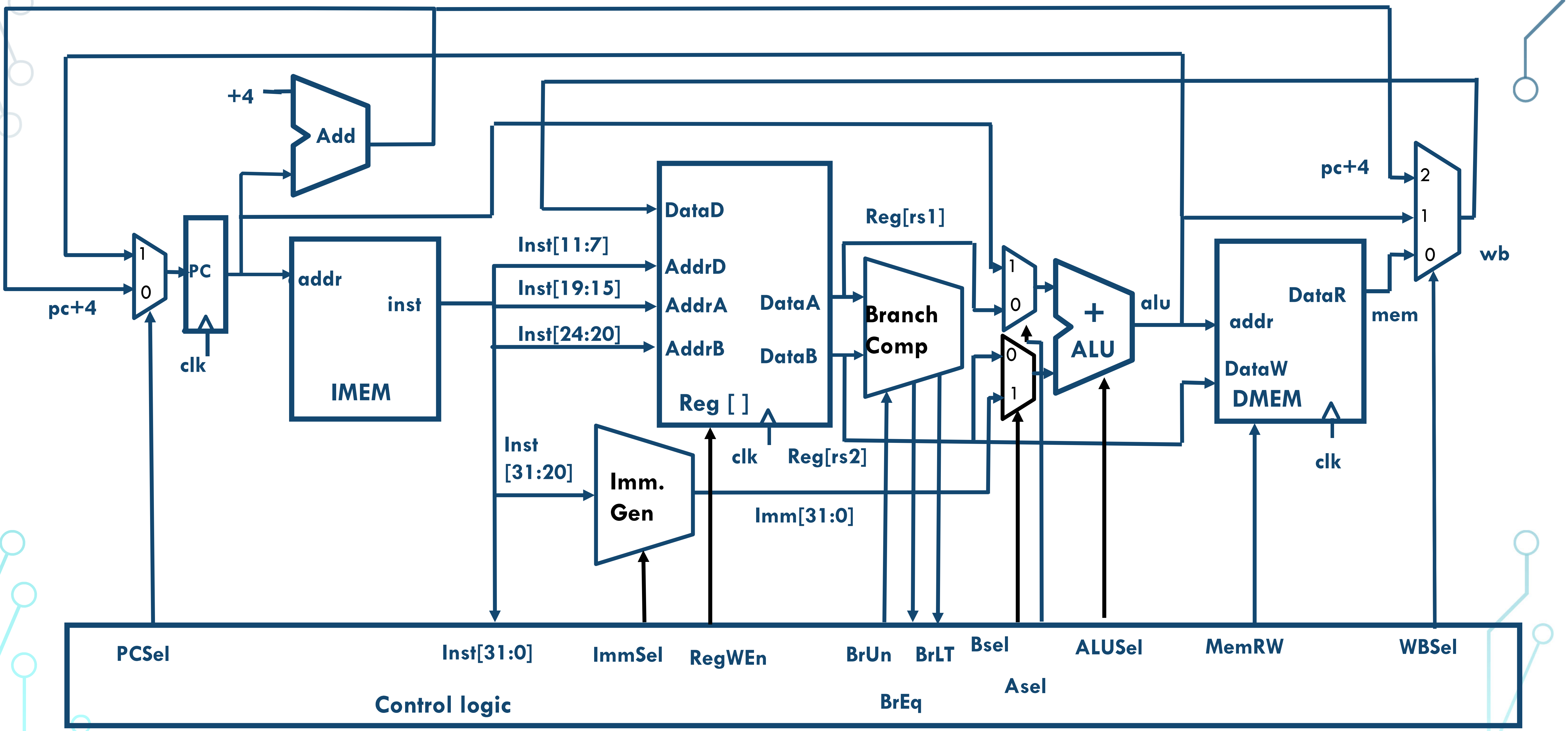

+4
Add
PC
clk
pc
pc+4
0
addr
inst
IMEM
Inst[11:7]
Inst[19:15]
Inst[24:20]
DataD
AddrD
AddrA
AddrB
DataA
DataB
Reg [ ]
clk
Reg[rs1]
Reg[rs2]
Inst
[31:7]
Imm.
Gen
Imm[31:0]
Branch
Comp
1
0
0
1
+
ALU
alu
addr
DataR
DataW
DMEM
clk
mem
pc+4
2
1
0
wb
PCSel
=taken
Inst[31:0]
ImmSel
=I
RegWEn
=1
BrUn
=*
BrLT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

Adding **JAL**

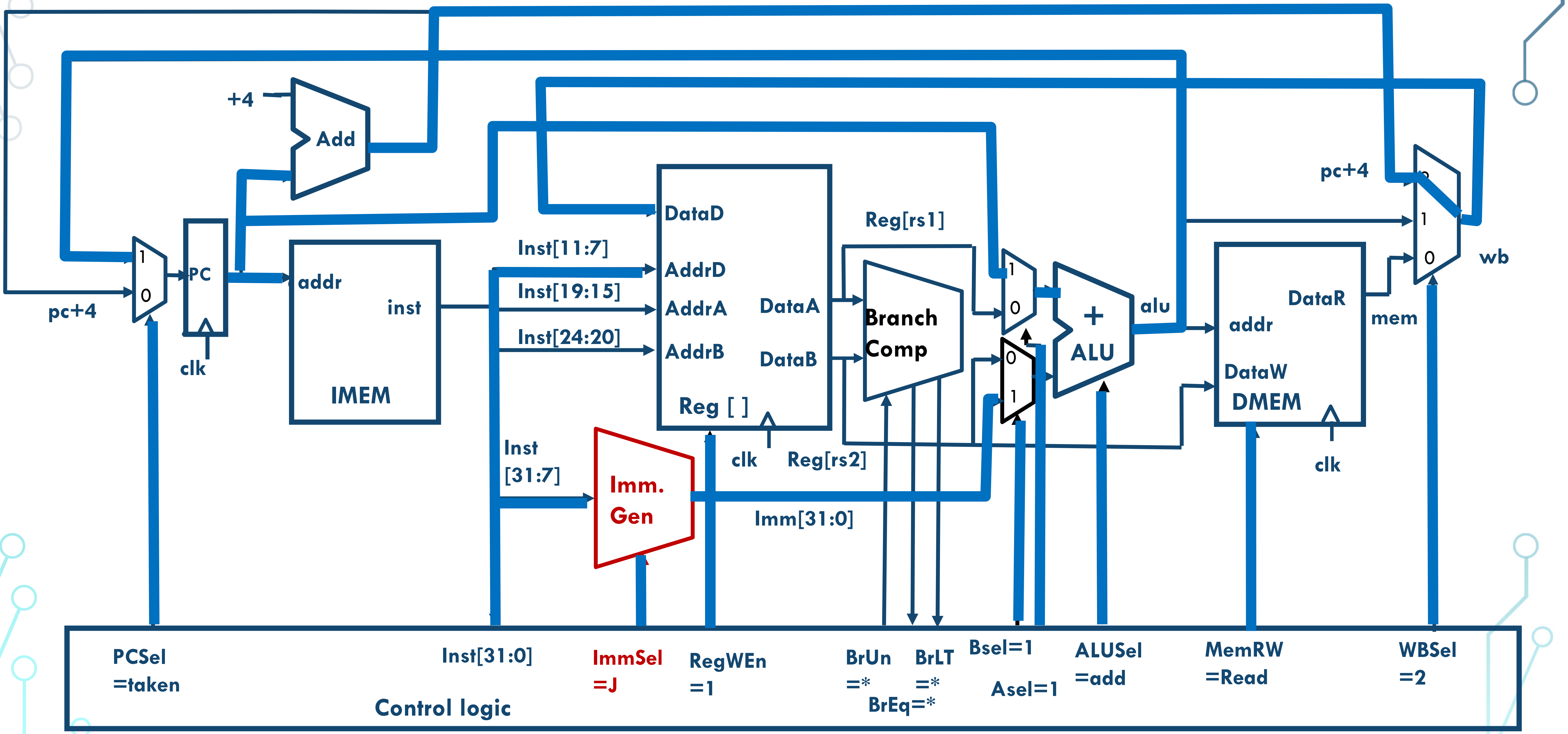
31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
offset[20:1]				dest	JAL

- JAL saves PC+4 in register rd (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within ±2^{19} locations, 2 bytes apart
  - ±2^{18} 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

Datapath with **JALR**



Adding **JAL**



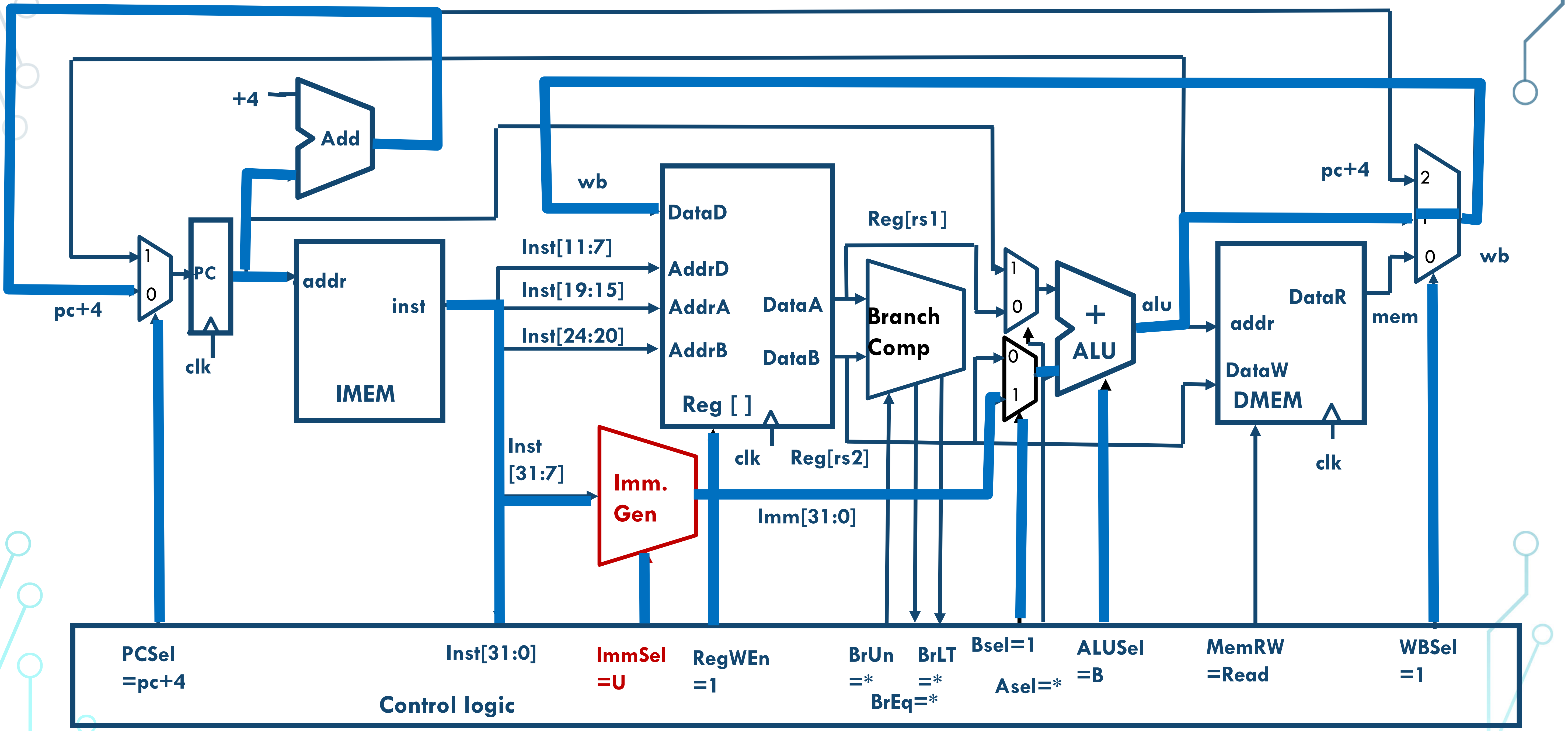
+4
Add
pc+4
1
0
PC
clk
addr
inst
IMEM
Inst[11:7]
Inst[19:15]
Inst[24:20]
DataD
AddrD
AddrA
AddrB
DataA
DataB
Reg [ ]
clk
Inst
[31:7]
Imm.
Gen
Reg[rs1]
Reg[rs2]
Imm[31:0]
Branch
Comp
1
0
0
1
+
ALU
alu
addr
DataR
DataW
DMEM
clk
mem
pc+4
1
0
wb
PCSel
=taken
Inst[31:0]
ImmSel
=J
RegWEn
=1
BrUn
=*
BrLT
=*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

U-Format for “Upper Immediate” Instructions

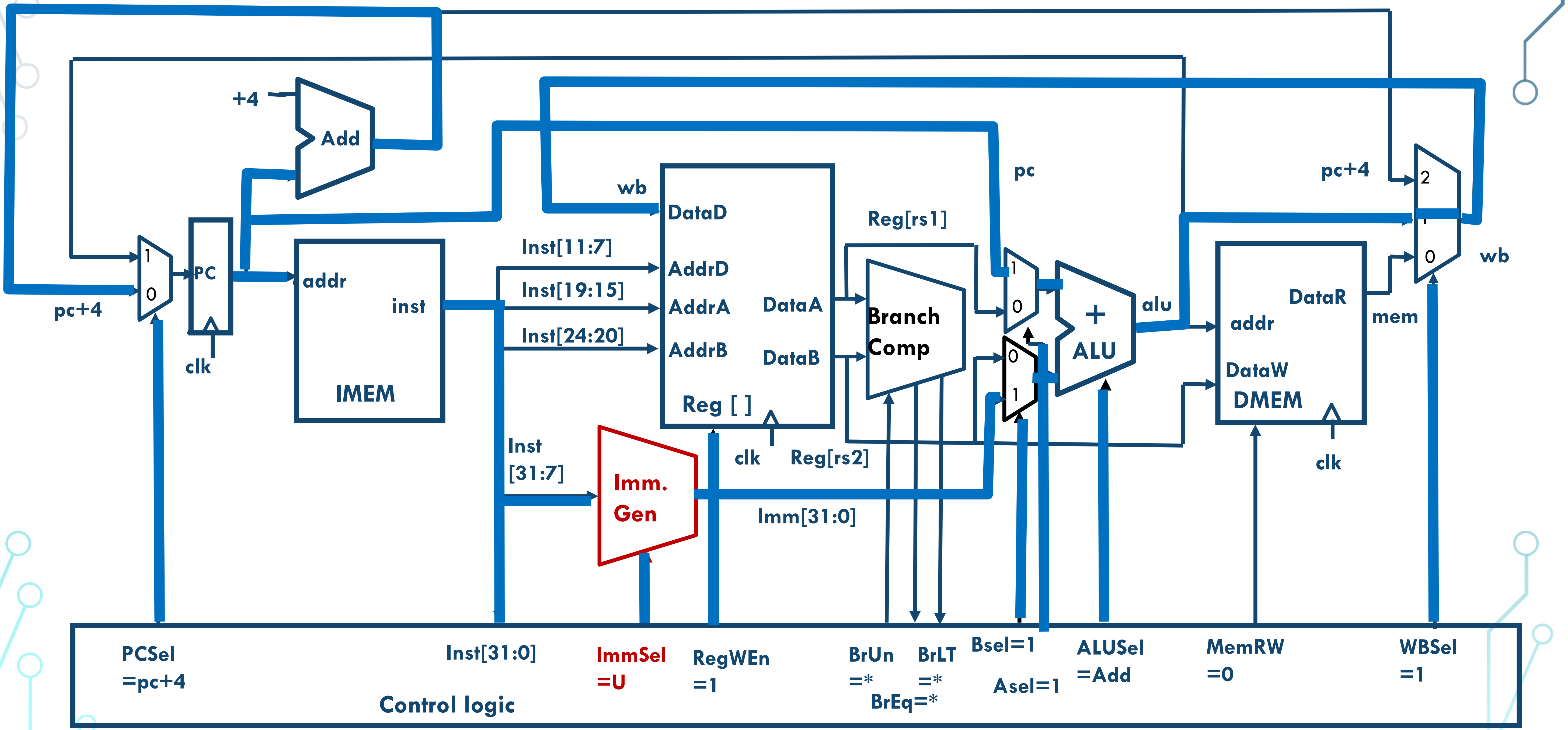
31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

- 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
  - AUIPC – Add Upper Immediate to PC

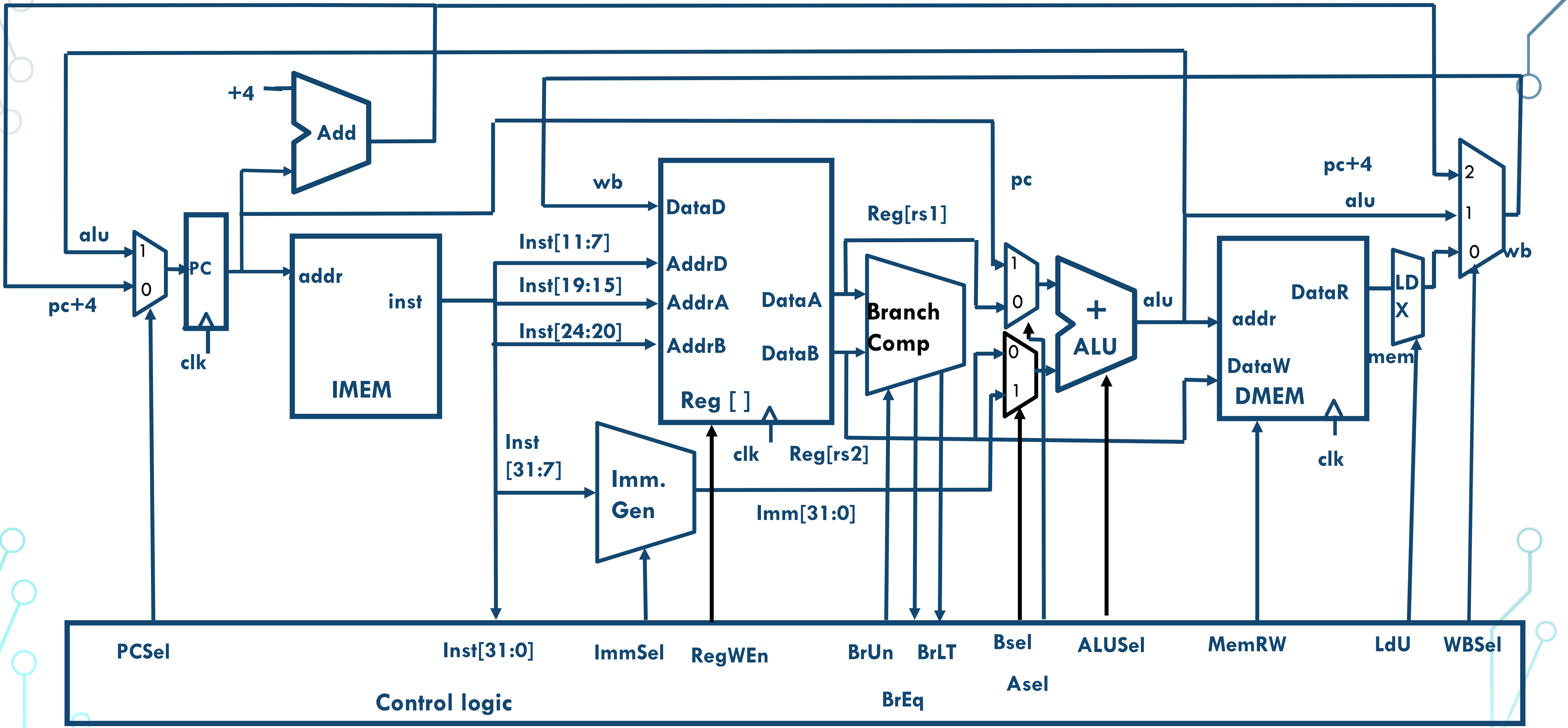
Implementing `LUI`



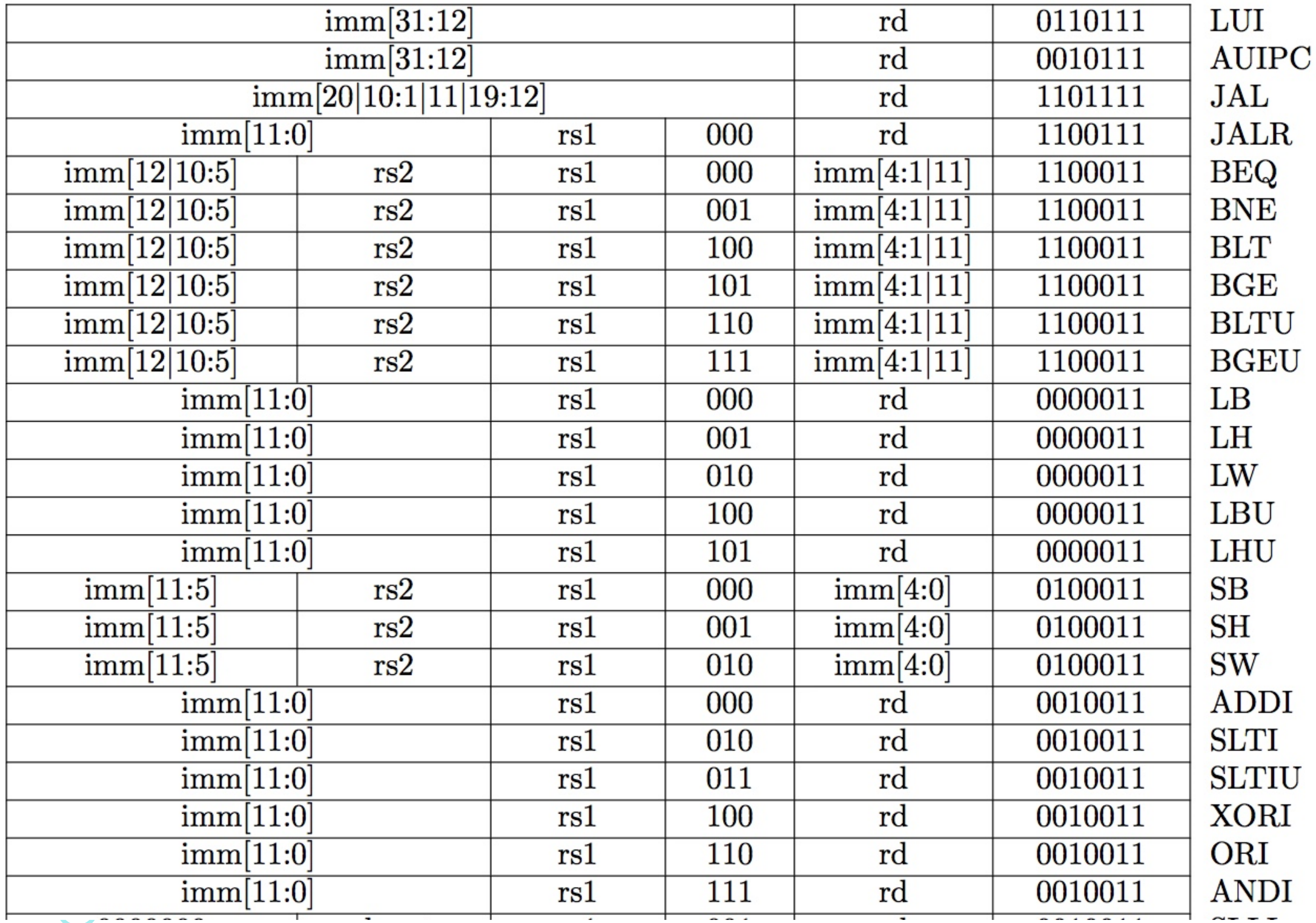
Implementing AUIPC



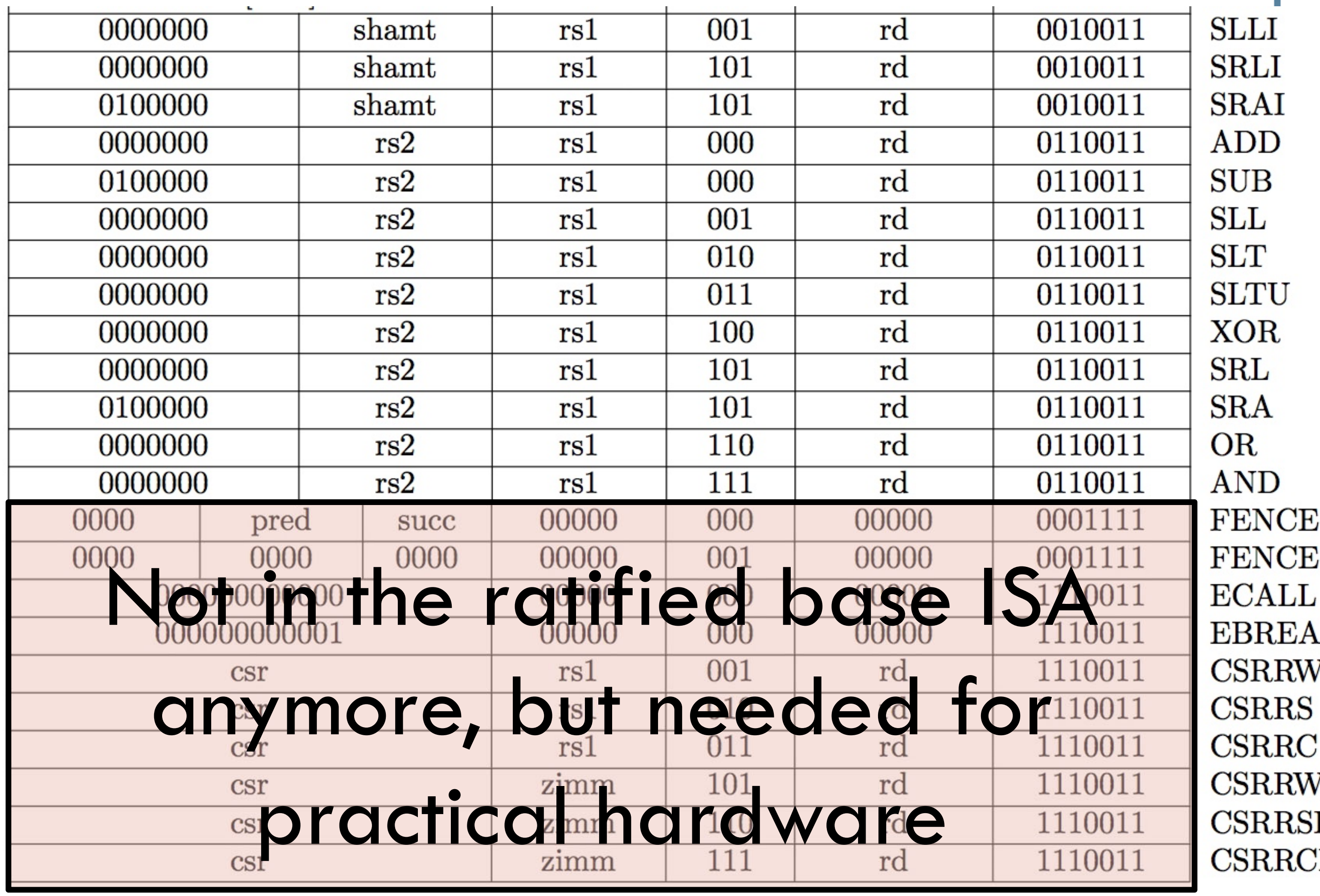
Complete RV32I Datapath!



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
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREA
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRW
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRC

Not in the ratified base ISA anymore, but needed for practical hardware

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

Summary of RISC-V Instruction Formats

31 30 25	24 21 20	19 15	14 12	11 8 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11]]		imm[19:12]		rd	opcode	J-type

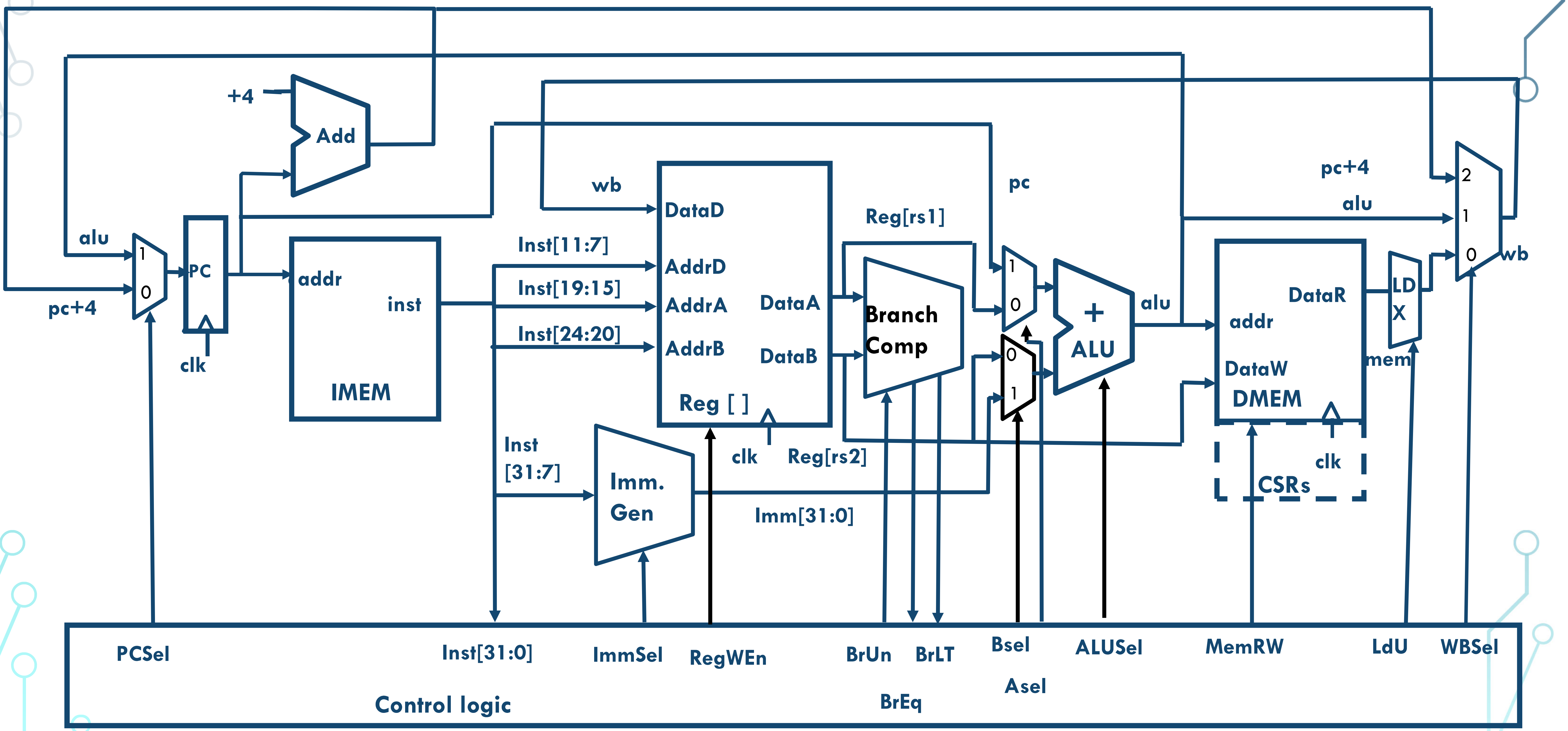
Control and Status Registers (CSRs)

- 4096 CSRs in a separate address space

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW CSRRS	dest	SYSTEM

- **`csrrw`** reads the old value of CSR, zero-extends and writes to **`rd`**
- Initial value of **`rs1`** is written to CSR
- Pseudo-instructions: **`csrr, csrw (csrrw x0, csr, rs1)`**

Add the `csrrw`!



Summary

- We have covered the implementation of the base ISA for RV32I
- Implementation suggested is straightforward, yet there are modalities in how to implement it well at gate level.
- Still need control
- Single-cycle datapath is slow – need to pipeline it