CS 110
Computer Architecture
Lecture 10:
Datapath

Instructors:

Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/20s/

School of Information Science and Technology SIST

ShanghaiTech University
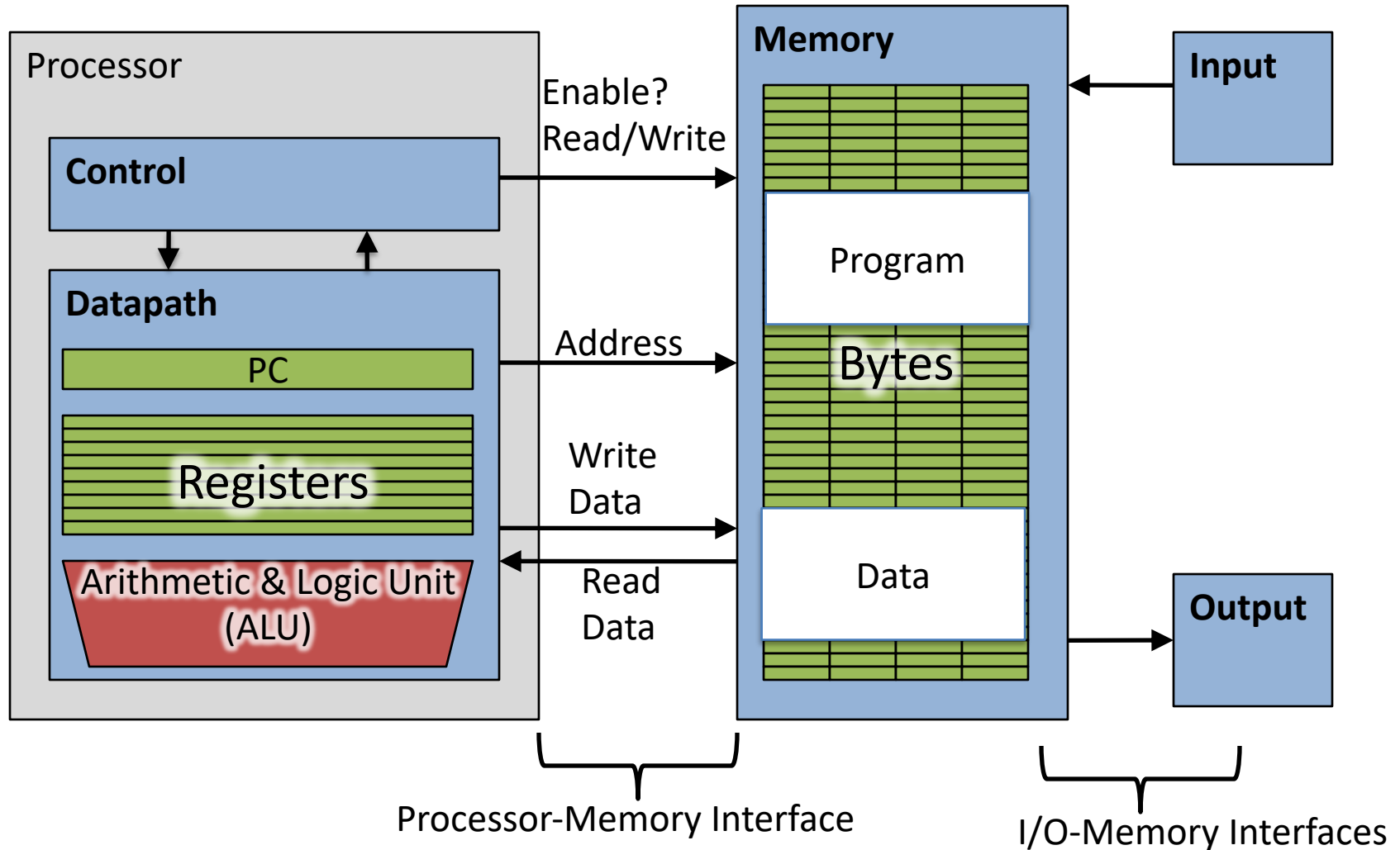
Slides based on UC Berkley's CS61C

Admin

- Project 1.1 due very soon!

- Start early with project 1.2…

- Be careful not to publicly post your HW or project code on gitlab!

- Do not make merge requests to the framework…

Review

- Timing constraints for Finite State Machines
  - Setup time, Hold Time, Clock to Q time
- Use muxes to select among inputs
  - S control bits selects from 2^S inputs
  - Each input can be n-bits wide, independent of S
  - Can implement muxes hierarchically
- ALU can be implemented using a mux
  - Coupled with basic block elements
 - Adder/ Substractor & AND & OR & shift

Components of a Computer

Processor

Control

Datapath

PC

Registers

Arithmetic & Logic Unit (ALU)

Enable?
Read/Write

Address

Write
Data

Read
Data

Memory

Program

Bytes

Data

Input

Output

Processor-Memory Interface

I/O-Memory Interfaces

The CPU

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

- Datapath: portion of the processor that contains hardware necessary to perform operations required by the processor

- Control: portion of the processor (also in hardware) that tells the datapath what needs to be done

# One-Instruction-Per-Cycle RISC-V Machine
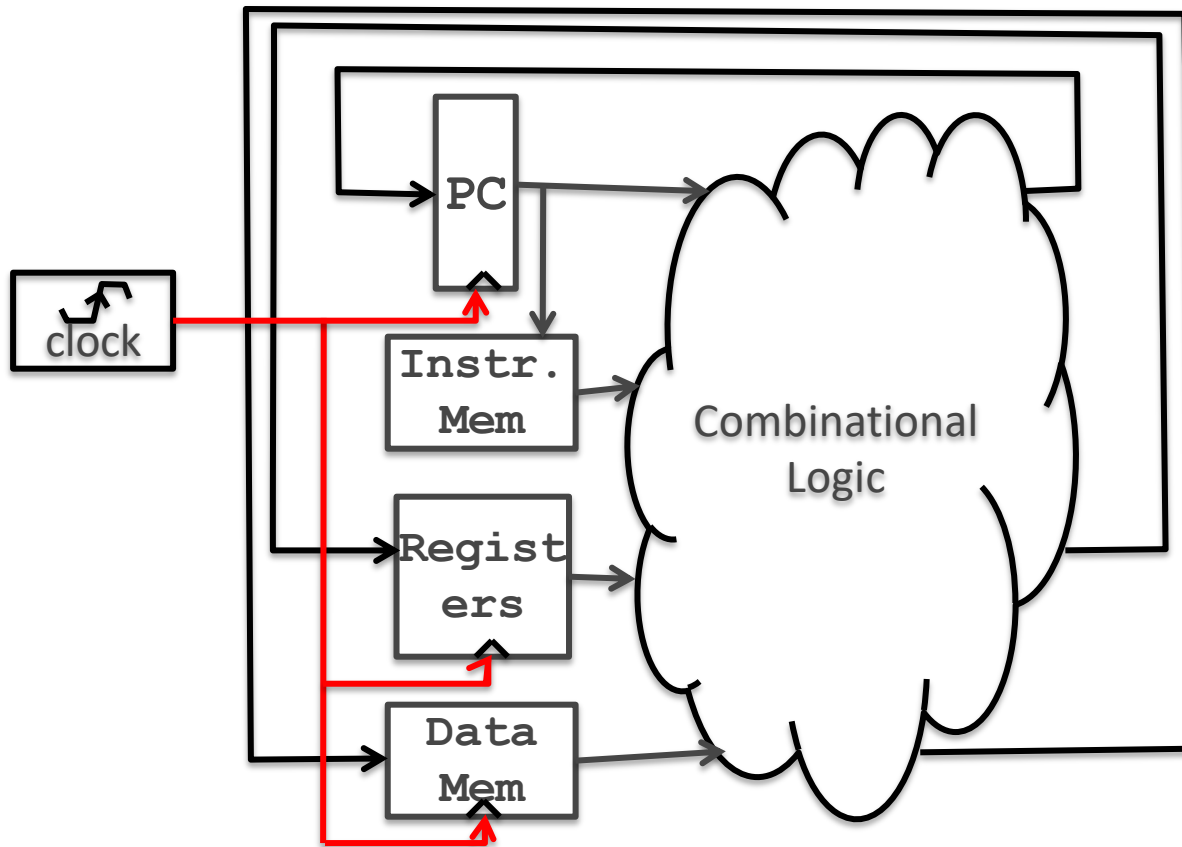
- One clock tick =>
  one instruction

- Current state outputs
  => inputs to
  combinational logic =>
  outputs settle at the
  values of state before
  next clock edge

- Rising clock edge:
  – all state elements
    are updated with
    combinational logic
    outputs
  – execution moves to
    next clock cycle

What is special about Instruction Memory?   **Why is Instruction Memory special?**

Datapath and Control

- Datapath designed to support data transfers required by instructions

- Controller causes correct transfers to happen
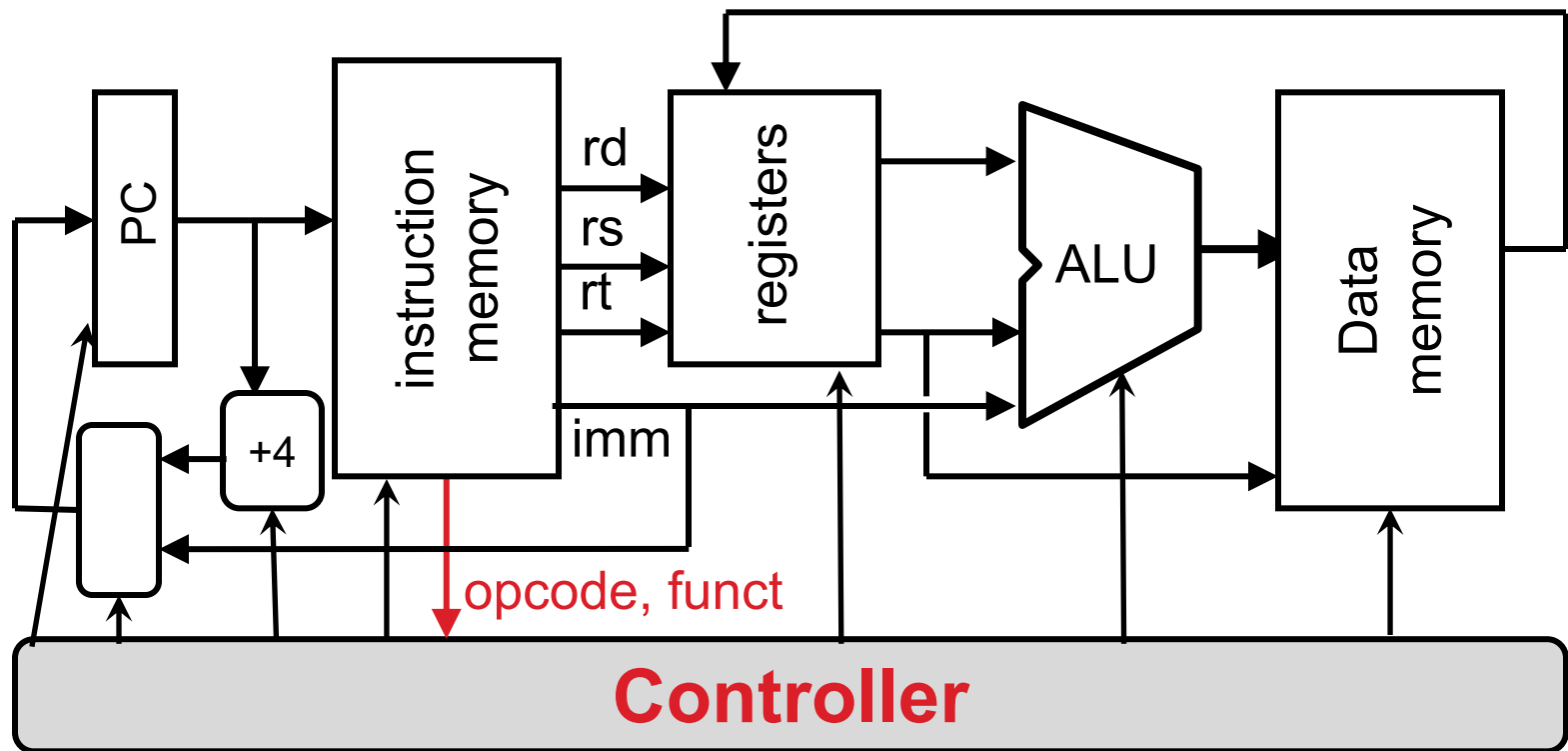
Stages of the Datapath : Overview

- Problem: a single, "monolithic" block that "executes an instruction" (performs all necessary operations beginning with fetching the instruction) would be too bulky and inefficient

- Solution: break up the process of "executing an instruction" into stages, and then connect the stages to create the whole datapath
  - smaller stages are easier to design
  - easy to optimize (change) one stage without touching the others (modularity)

Five Stages of Instruction Execution

- Stage 1: Instruction Fetch (IF)

- Stage 2: Instruction Decode (ID)

- Stage 3: Execute (EX): ALU (Arithmetic-Logic Unit)

- Stage 4: Memory Access (MEM)

- Stage 5: Register Write (WB)

Stages of Execution on Datapath

1. Instruction Fetch
2. Decode/ Register Read
3. Execute
4. Memory
5. Register Write
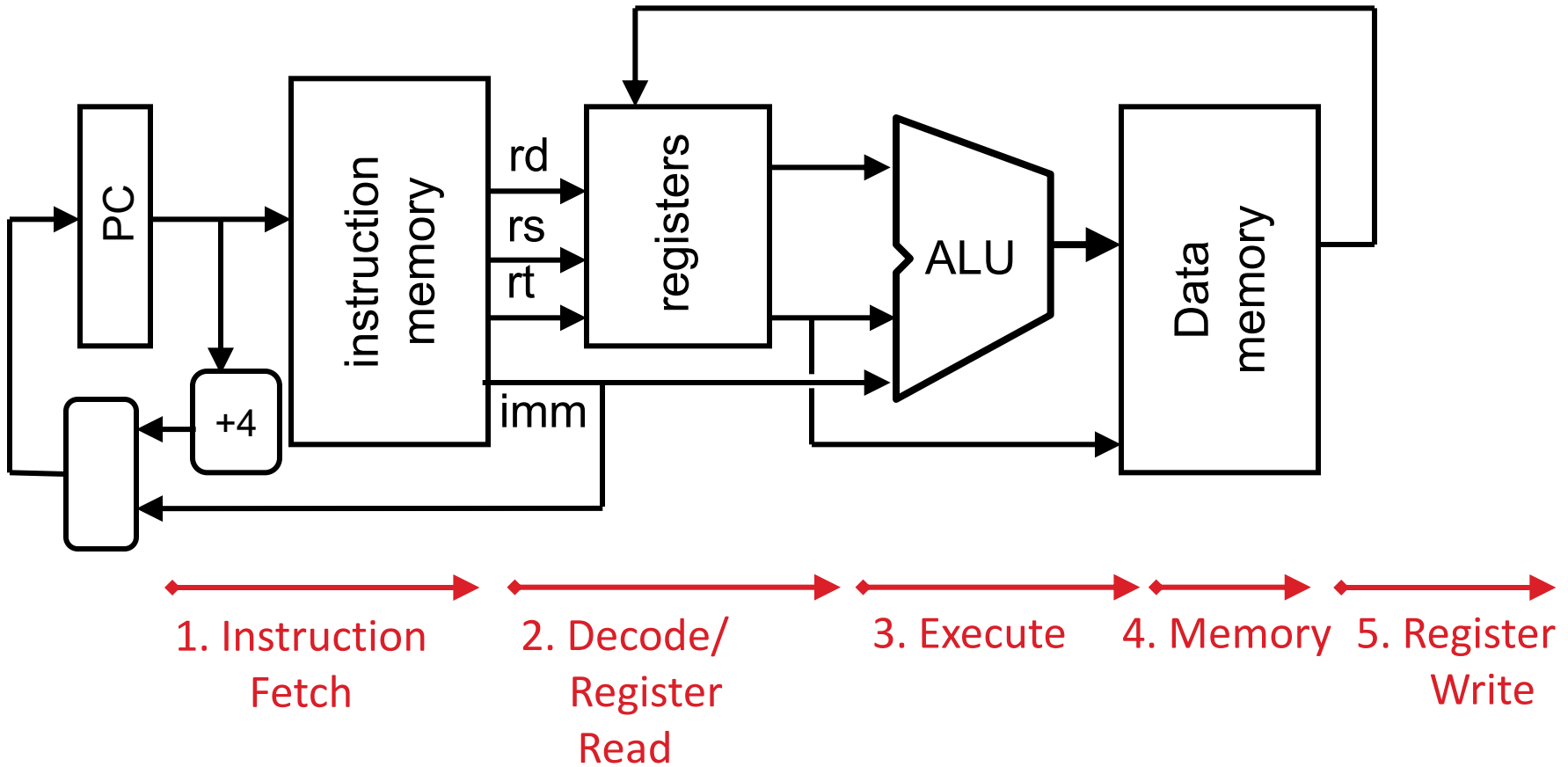
Stages of Execution (1/5)

- There is a wide variety of RISC-V instructions: so what general steps do they have in common?

- Stage 1: Instruction Fetch

  – no matter what the instruction, the 32-bit instruction word must first be fetched from memory (the cache-memory hierarchy)

  – also, this is where we Increment PC
  (that is, PC = PC + 4, to point to the next instruction: byte addressing so + 4)

Stages of Execution (2/5)

- Stage 2: Instruction Decode
  - upon fetching the instruction, we next gather data from the fields (decode all necessary instruction data)
  - first, read the opcode to determine instruction type and field lengths
  - second, (at the same time!) read in data from all necessary registers
    - for add, read two registers
    - for addi, read one register
  - third, generate the immediates

Stages of Execution (3/5)

- Stage 3: ALU (Arithmetic-Logic Unit)
  - the real work of most instructions is done here: arithmetic (+, -, *, /), shifting, logic (&, |)

  - what about loads and stores?
    - lw   t0, 40(t1)
    - the address we are accessing in memory = the value in t1 PLUS the value 40
    - so we do this addition in this stage
 - also does stuff for other instructions...

Stages of Execution (4/5)

- Stage 4: Memory Access
  - actually only the load and store instructions do anything during this stage; the others remain idle during this stage or skip it all together
  - since these instructions have a unique step, we need this extra stage to account for them
  - as a result of the cache system, this stage is expected to be fast

Stages of Execution (5/5)

- Stage 5: Register Write
  - most instructions write the result of some computation into a register
  - examples: arithmetic, logical, shifts, loads, jumps
  - what about stores, branches?
    - don't write anything into a register at the end
    - these remain idle during this fifth stage or skip it all together
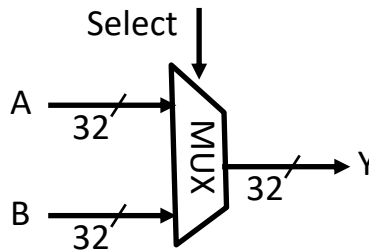
Stages of Execution on Datapath

1. Instruction Fetch
2. Decode/ Register Read
3. Execute
4. Memory
5. Register Write
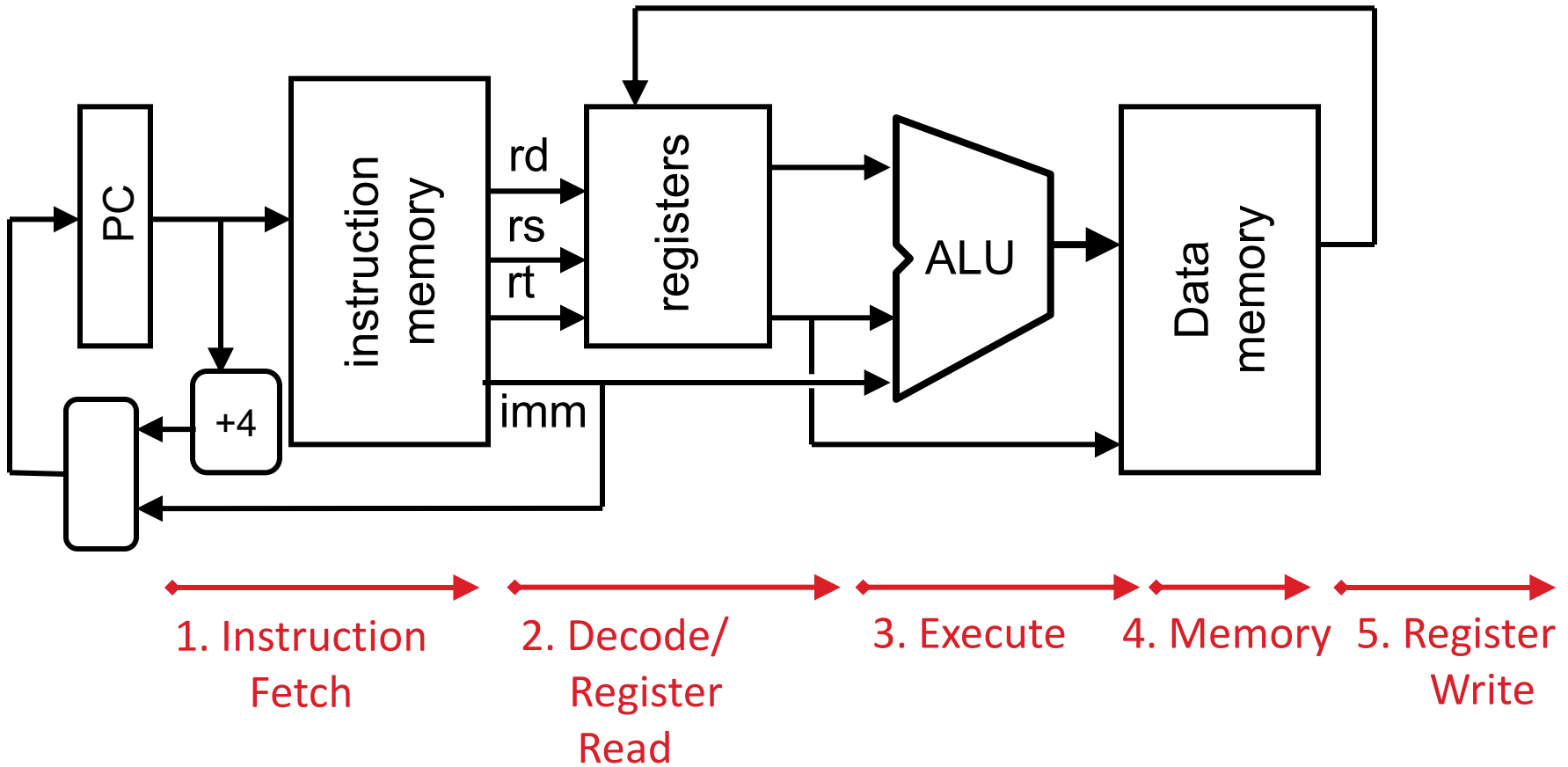
Datapath Components: Combinational
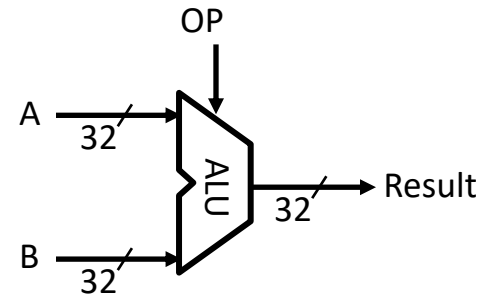
- Combinational Elements
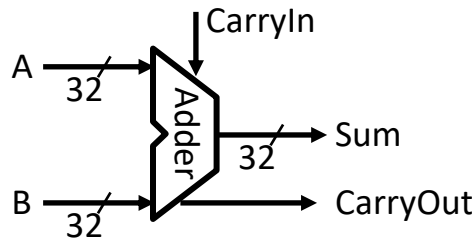
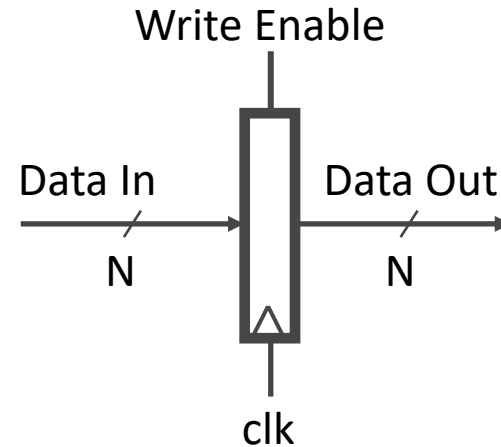

Adder   **Multiplexer**   **ALU**

- Storage Elements + Clocking Methodology
- Building Blocks

# Datapath Elements: State and Sequencing (1/3)

- Register

- Write Enable:

  - Negated (or deasserted) (0):
    Data Out will not change

  - Asserted (1): Data Out will become Data In on
    positive edge of clock

Write Enable

Data In → N → Data Out → N

clk

Datapath Elements: State and Sequencing (2/3)

- Register file (regfile, RF) consists of 32 registers
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
  - In one clock cycle can read two registers and write another!

Memory Size of Register File?
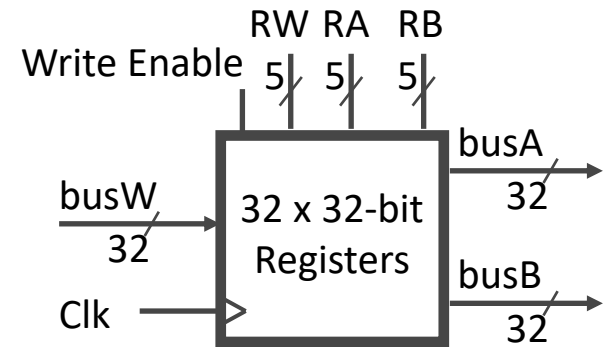
- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1

- Clock input (clk)
  - Clk input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid $\Rightarrow$ busA or busB valid after "access time."

Datapath Elements: State and Sequencing (3/3)

Write Enable   Address

Data In

32

Clk

DataOut

32

- "Magic" Memory
  - One input bus: Data In
  - One output bus: Data Out
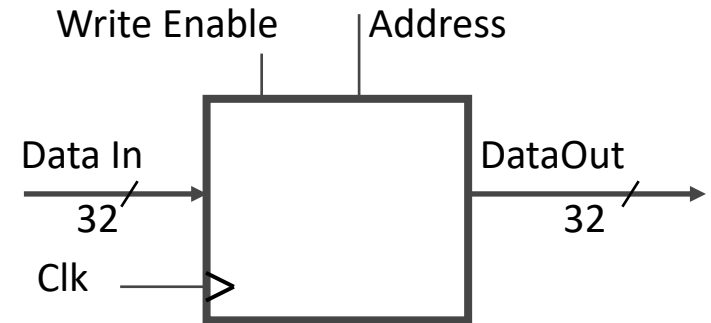
- Memory word is found by:
  - For Read: Address selects the word to put on Data Out
  - For Write: Set Write Enable = 1: address selects the memory word to be written via the Data In bus

- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block: Address valid $\Rightarrow$ Data Out valid after "access time"

State Required by RV32I ISA

Each instruction reads and updates this state during execution:

- Registers (`x0..x31`)
  - Register file (*regfile*) `Reg` holds 32 registers x 32 bits/register: `Reg[0]..Reg[31]`
  - First register read specified by *rs1* field in instruction
  - Second register read specified by *rs2* field in instruction
  - Write register (destination) specified by *rd* field in instruction
  - `x0` is always 0 (writes to `Reg[0]` are ignored)

- Program Counter (`PC`)
  - Holds address of current instruction

- Memory (`MEM`)
  - Holds both instructions & data, in one 32-bit byte-addressed memory space
  - We'll use separate memories for instructions (IMEM) and data (DMEM)
    - *These are placeholders for instruction and data caches*
  - Instructions are read (*fetched*) from instruction memory (assume `IMEM` read-only)
  - Load/store instructions access data memory
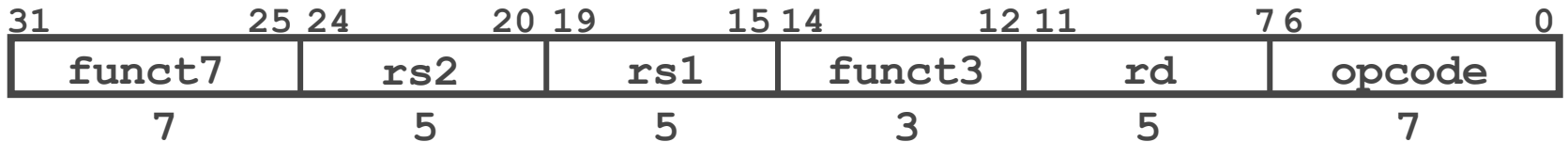
Review: Complete RV32I ISA

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

Not in CA

- Need datapath and control to implement these instructions

Implementing the **add** instruction

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

0000000	rs2	rs1	000	rd	0110011

add   **rs2**   **rs1**   **add**   **rd**   **Reg-Reg OP**

`add rd, rs1, rs2`

- Instruction makes two changes to machine's state:
  - `Reg[rd] = Reg[rs1] + Reg[rs2]`
  - `PC = PC + 4`

Datapath for add
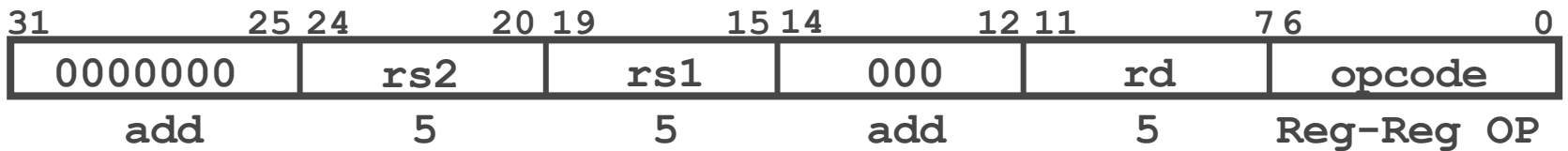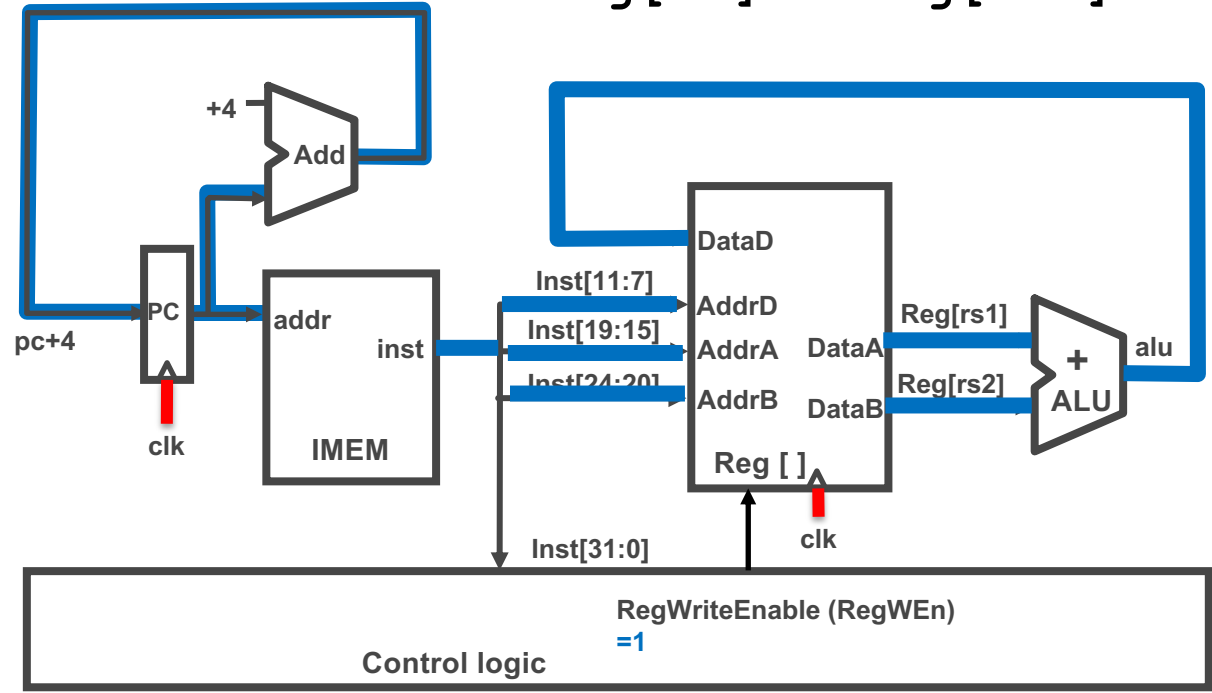
Timing Diagram for **add**
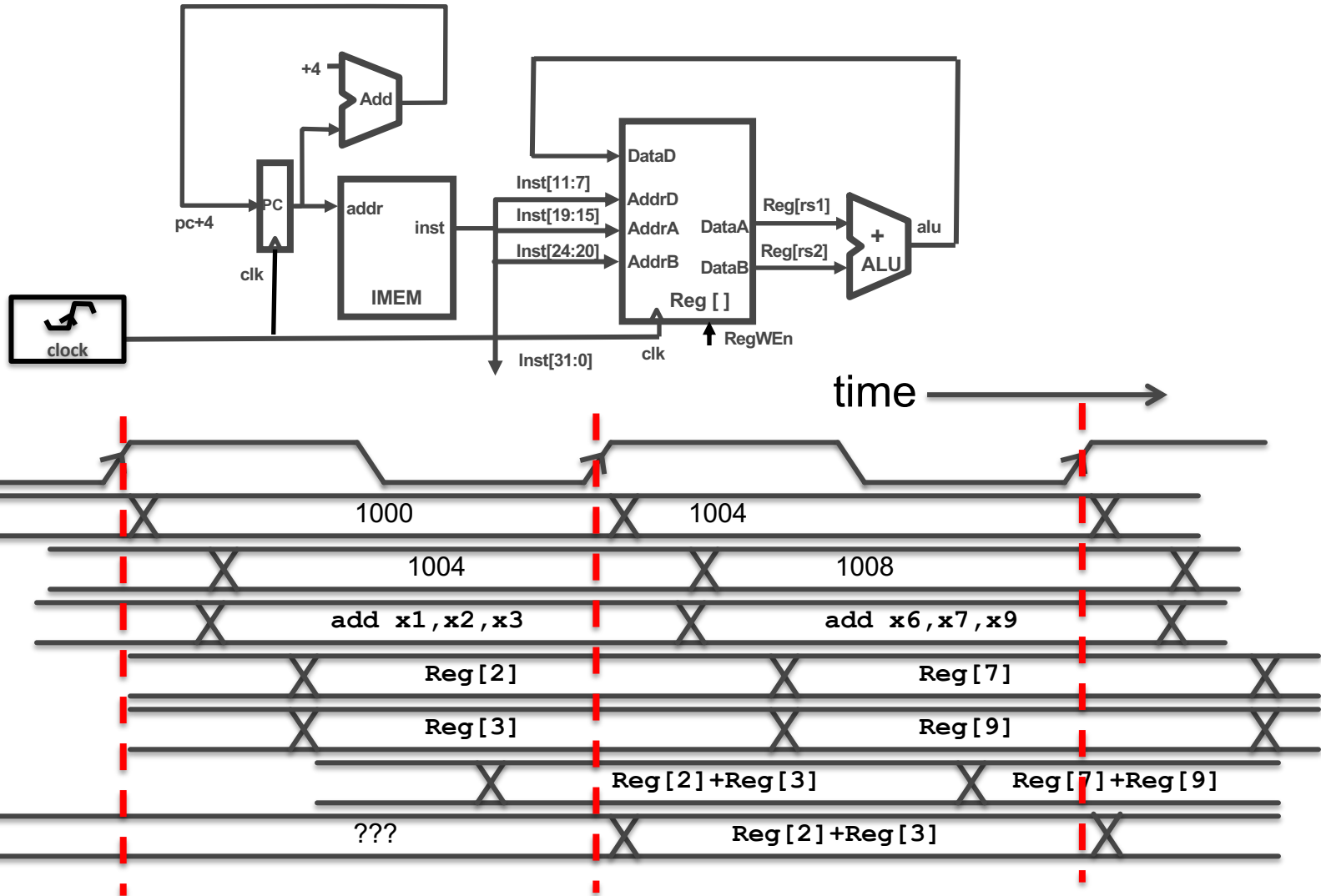
Implementing the **sub** instruction

31	25	24	20	19	15	14	12	11	7	6	0	
0000000		rs2		rs1		000		rd		0110011		add
0100000		rs2		rs1		000		rd		0110011		sub

sub rd, rs1, rs2

- Almost the same as add, except now have to subtract operands instead of adding them

- **inst[30]** selects between add and subtract

Datapath for `add/sub`
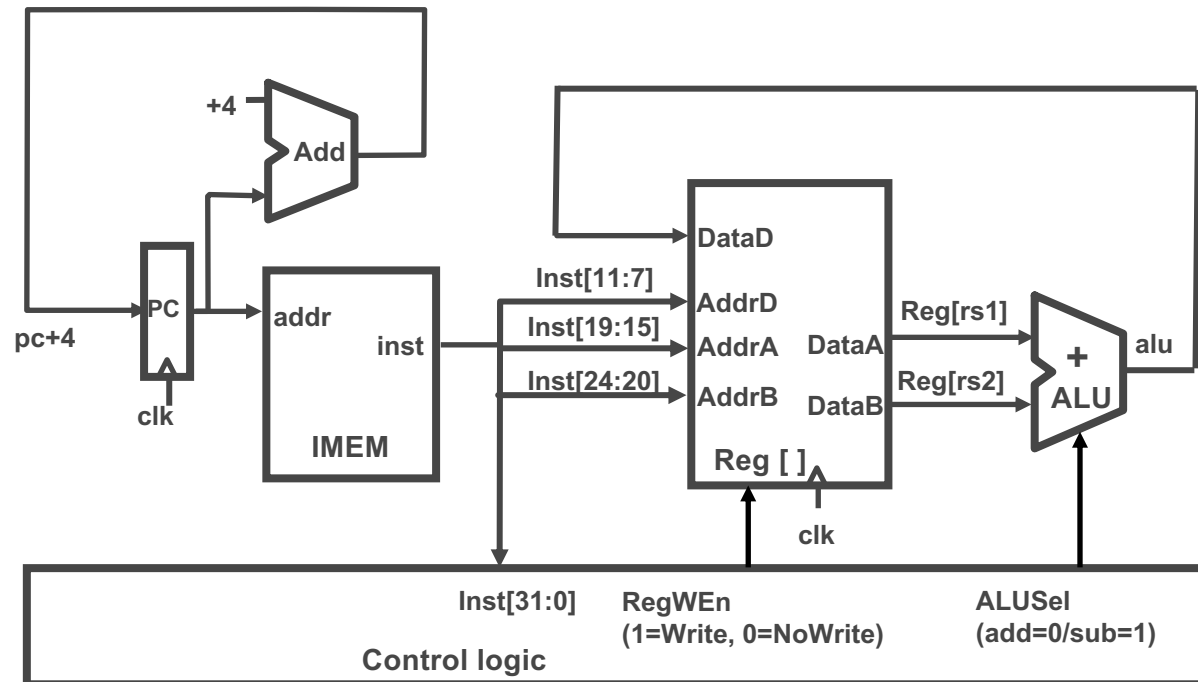
Implementing other R-Format instructions

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

TA Discussion

Video Anqi Pang:

https://robotics.shanghaitech.edu.cn/static/ca2020/Discussion_10_DatapathAnqiPan.mp4

Q & A

Quiz

Quiz

- Select the statements that are TRUE:

A. The Clk->Q delay is not important for the Datapath.
B. The Datapath for add and sub are identical – the only difference is that the controller is signaling the ALU which instruction to execute.
C. The result of an instruction is written into the destination register as soon as it is ready.
D. The controller is getting the instruction during the fetch stage.
E. The datapath introduced so far contains two adders.

CS 110
Computer Architecture
Lecture 10:
Datapath
Video 2: I & S

Instructors:

Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/20s/

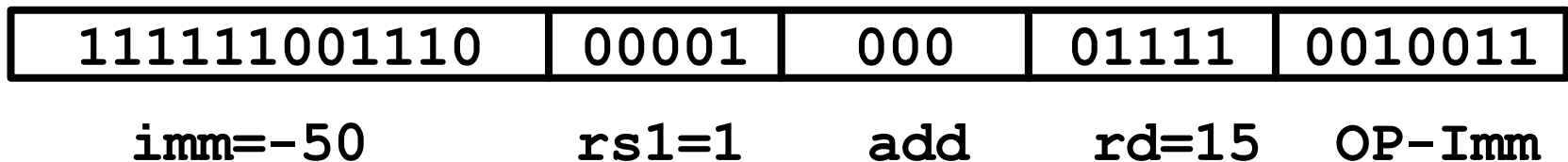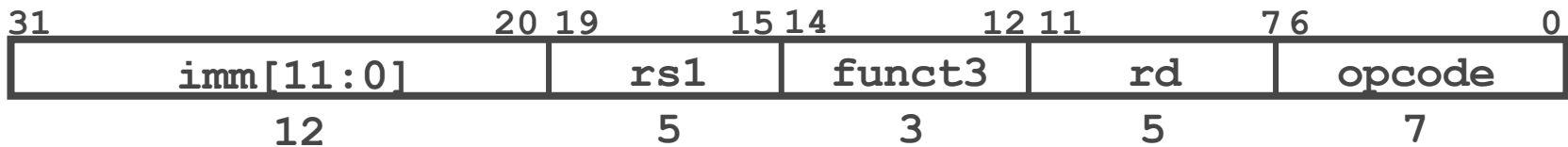**School of Information Science and Technology SIST**

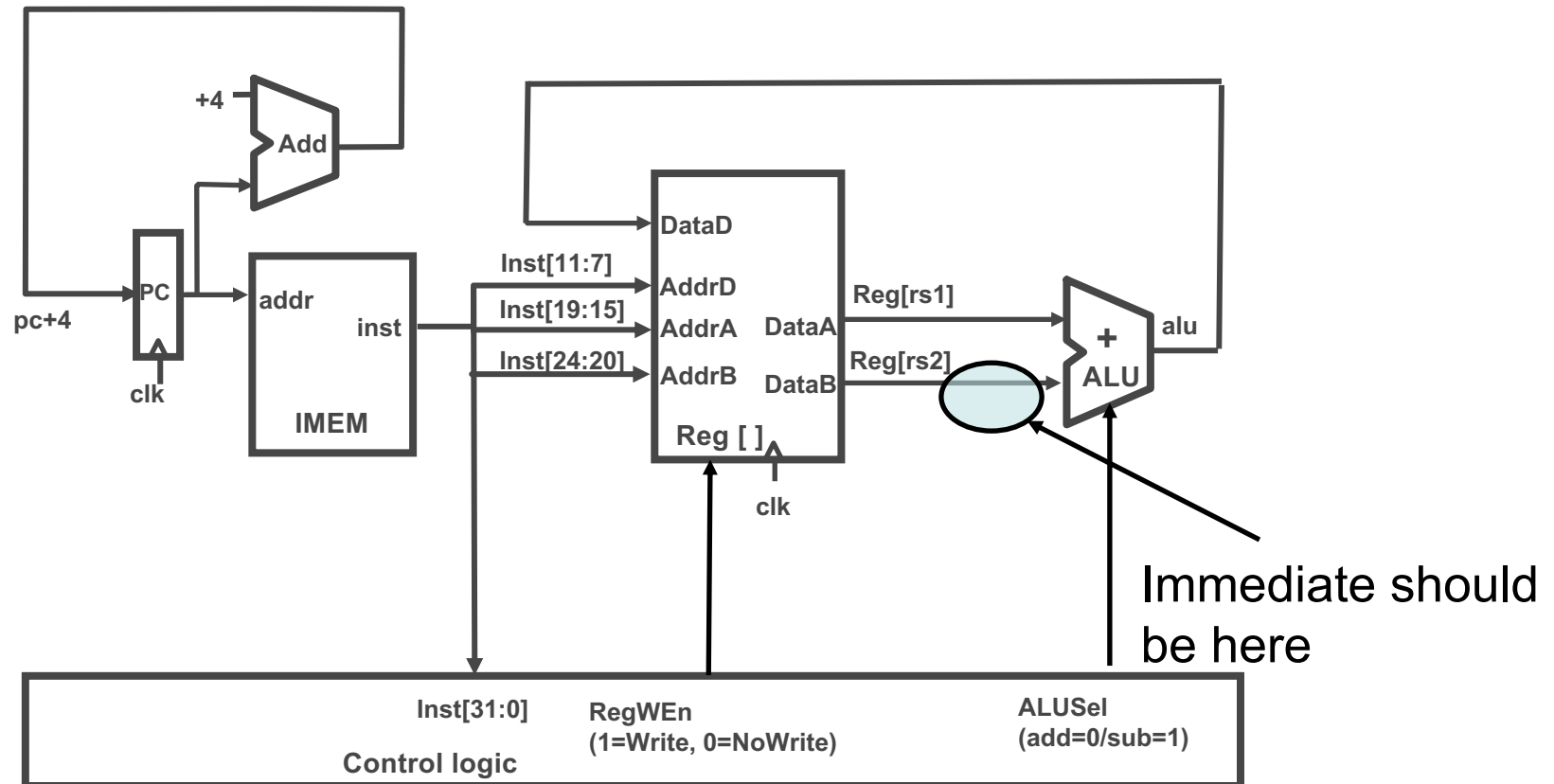**ShanghaiTech University**

Slides based on UC Berkley's CS61C

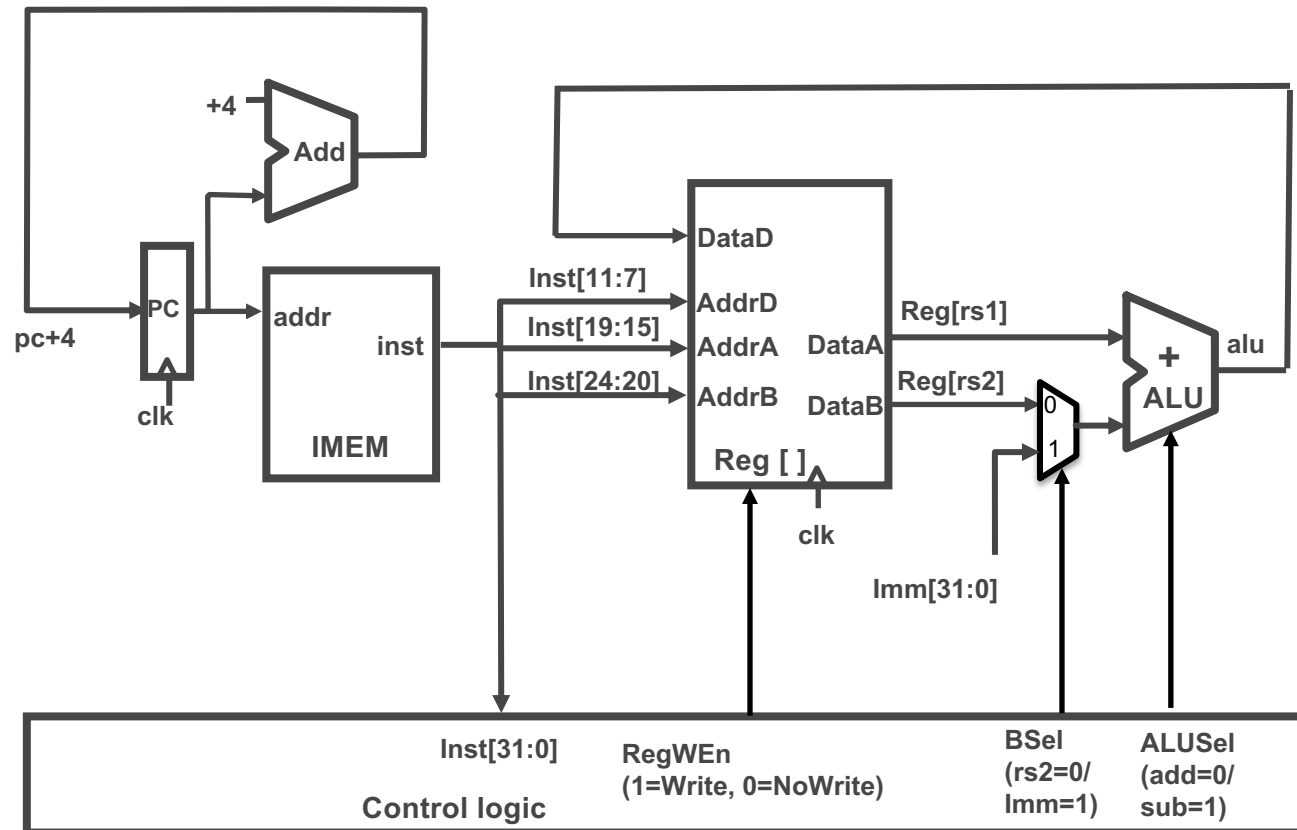# Implementing I-Format - `addi` instruction

- **RISC-V Assembly Instruction:**

  `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`

Adding `addi` to Datapath

Adding `addi` to Datapath

I-Format immediates

```
-inst[31]-
31 30              20 19      15 14      12 11      7 6        0
  imm[11:0]           rs1        funct3      rd        opcode
       12
```

inst[31:0]

| ------inst[31]-(sign-extension)------- | inst[30:20] |
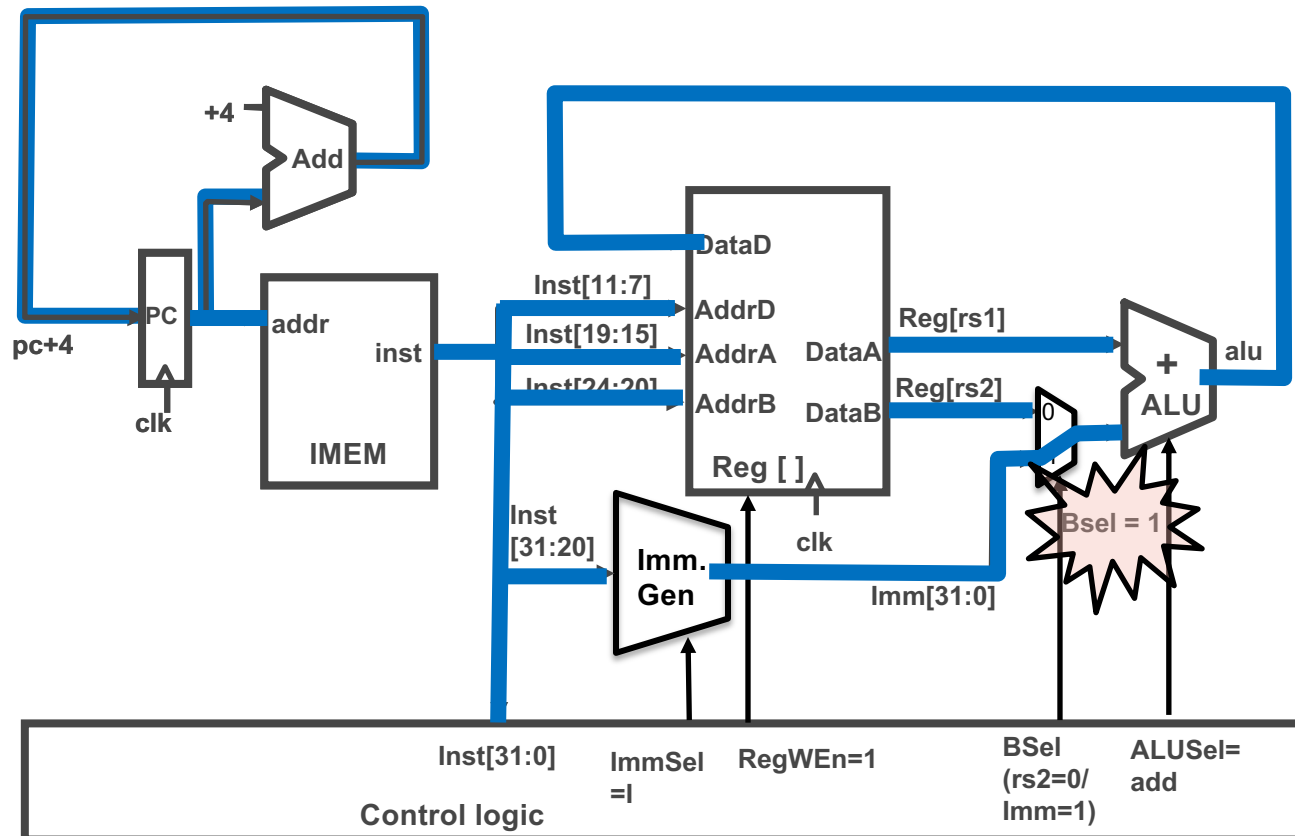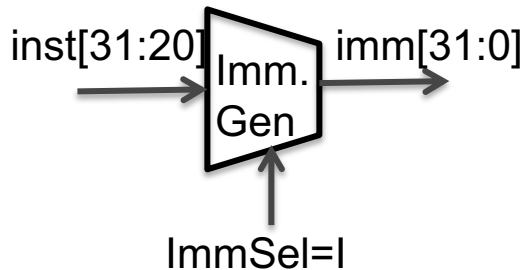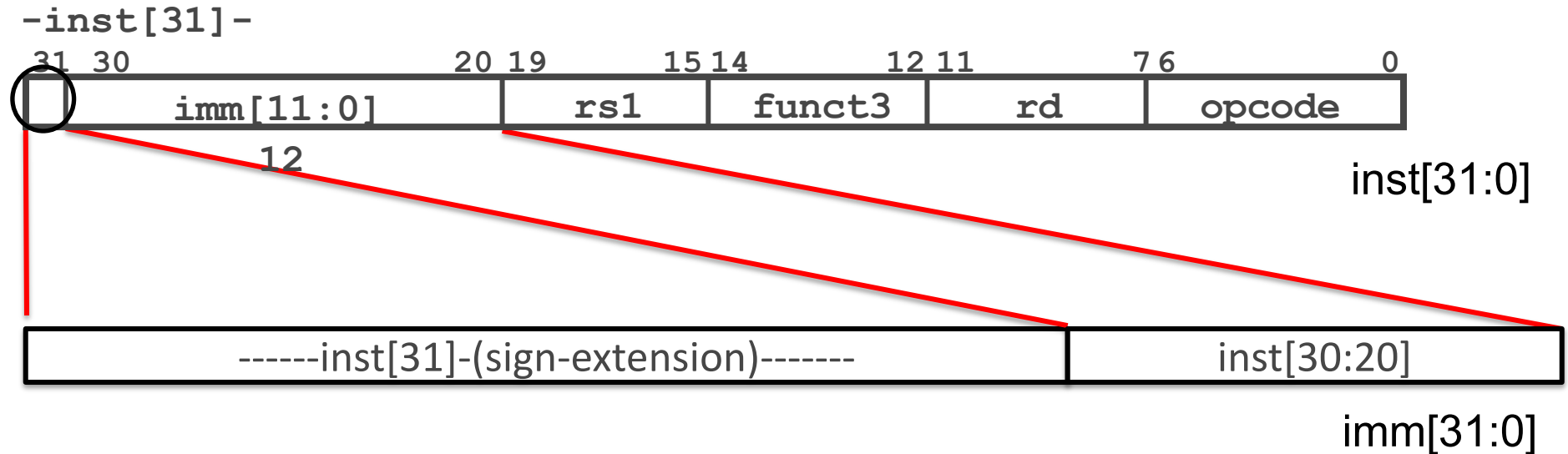
imm[31:0]

- 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])
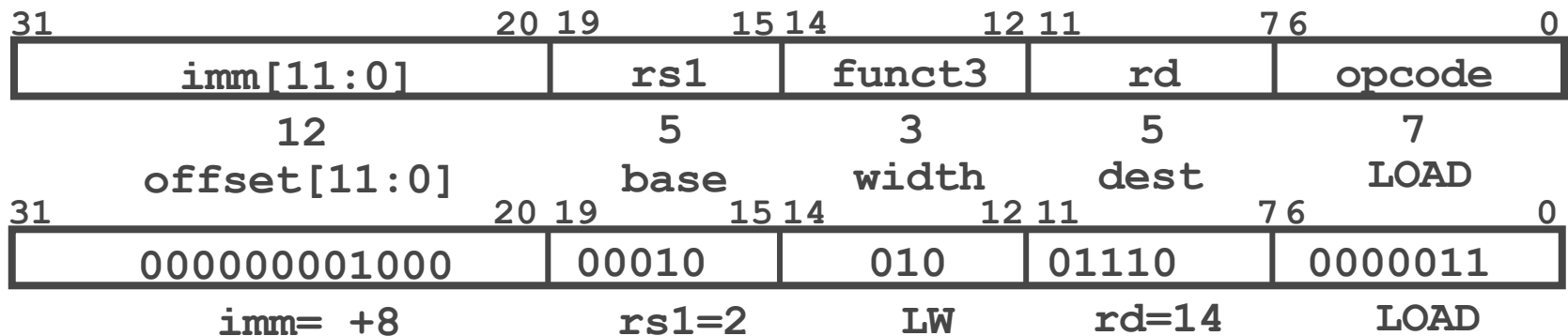
R+I Datapath

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

Question

9

	123
A:	FFF
B:	FFT
C:	FTF
D:	FTT
E:	TFF
F:	TFT
G:	TTF
H:	TTT

1) Program counter is a register

2) We should use the main ALU to compute PC=PC+4 in order to save some gates

3) The ALU is a synchronous state element

Add `lw`

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

31 imm[11:0] 20	19 rs1 15 14	funct3 12 11	rd 7 6	opcode 0
12 offset[11:0]	5 base	3 width	5 dest	7 LOAD
31 000000001000 20	19 00010 15 14	010 12 11	01110 7 6	0000011 0
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
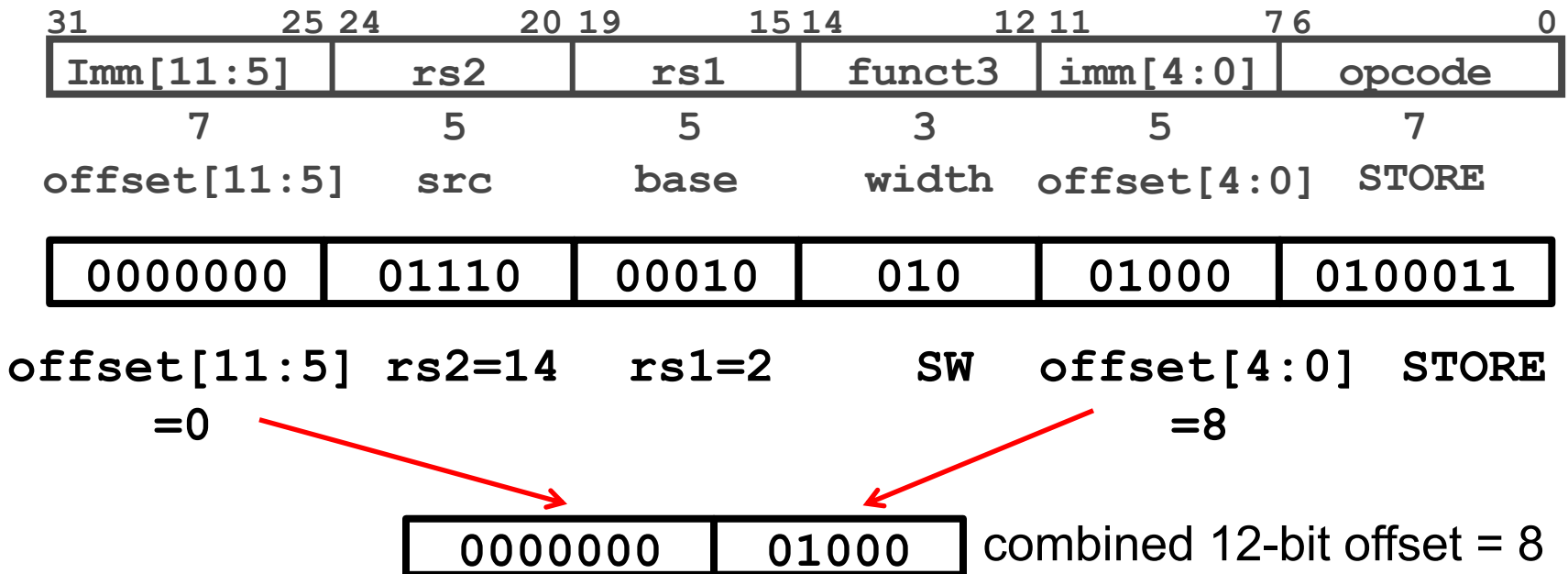
Adding lw to Datapath
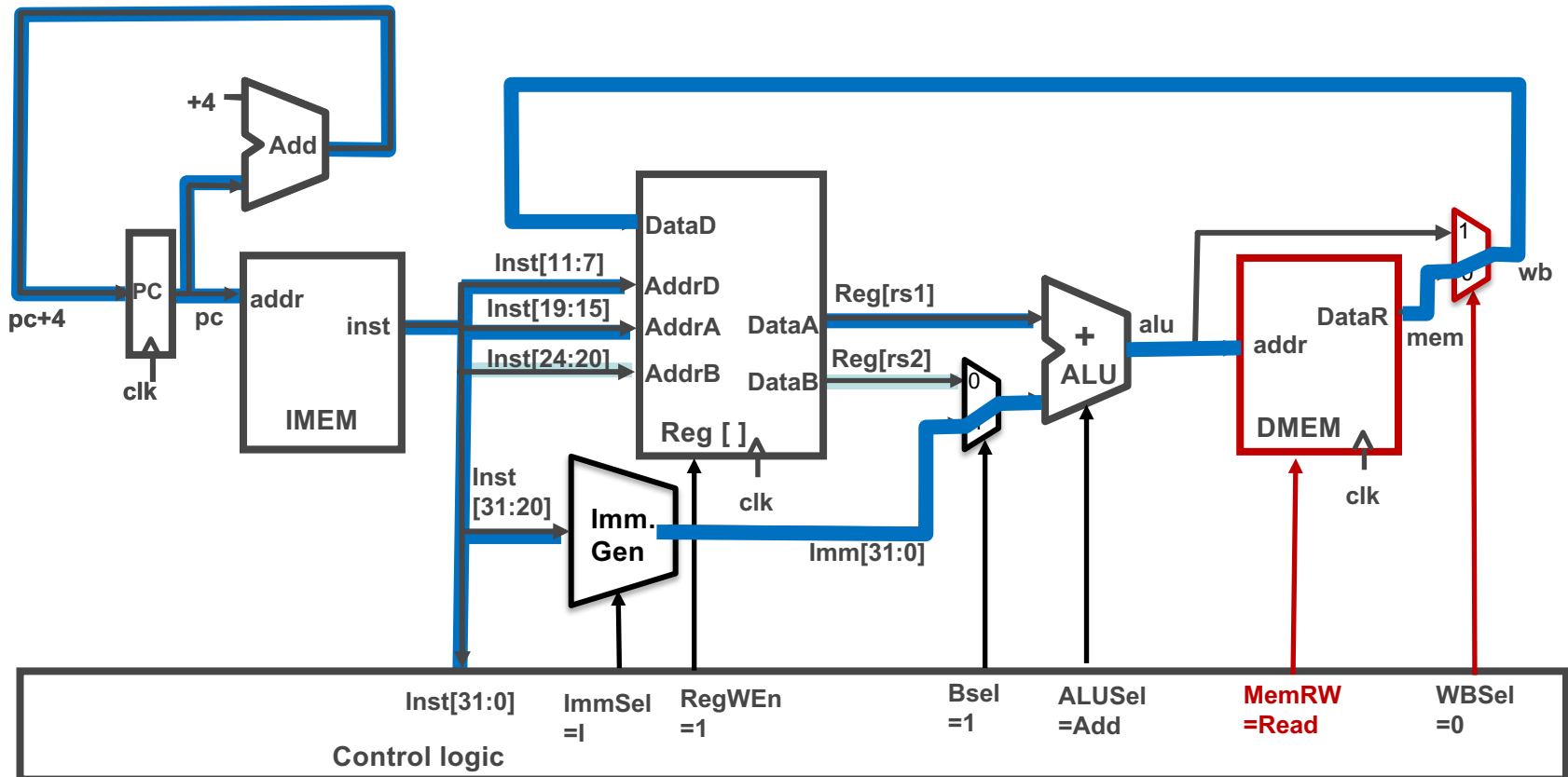
All RV32 Load Instructions

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

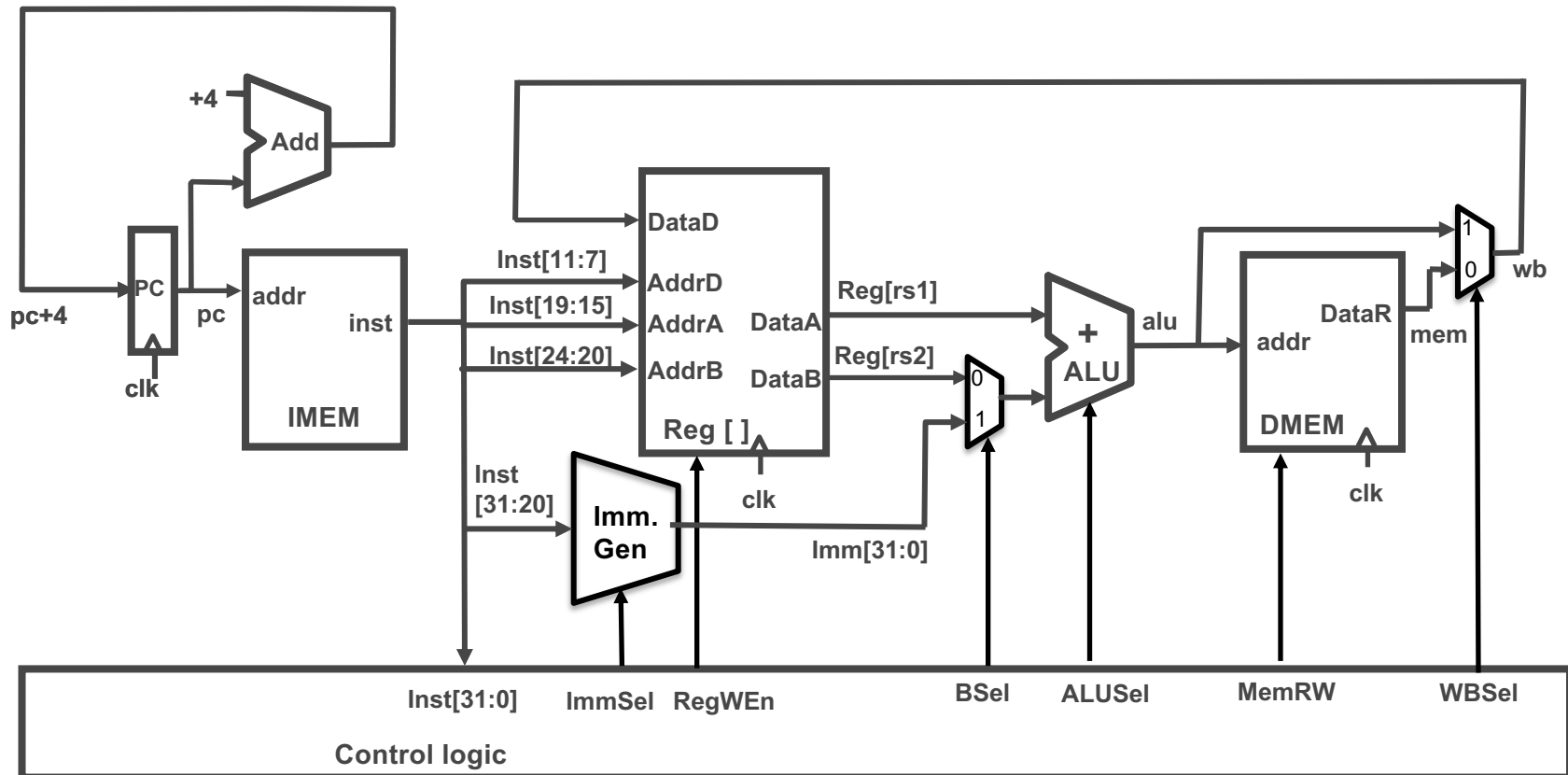funct3 field encodes size and 'signedness' of load data

- 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 mod
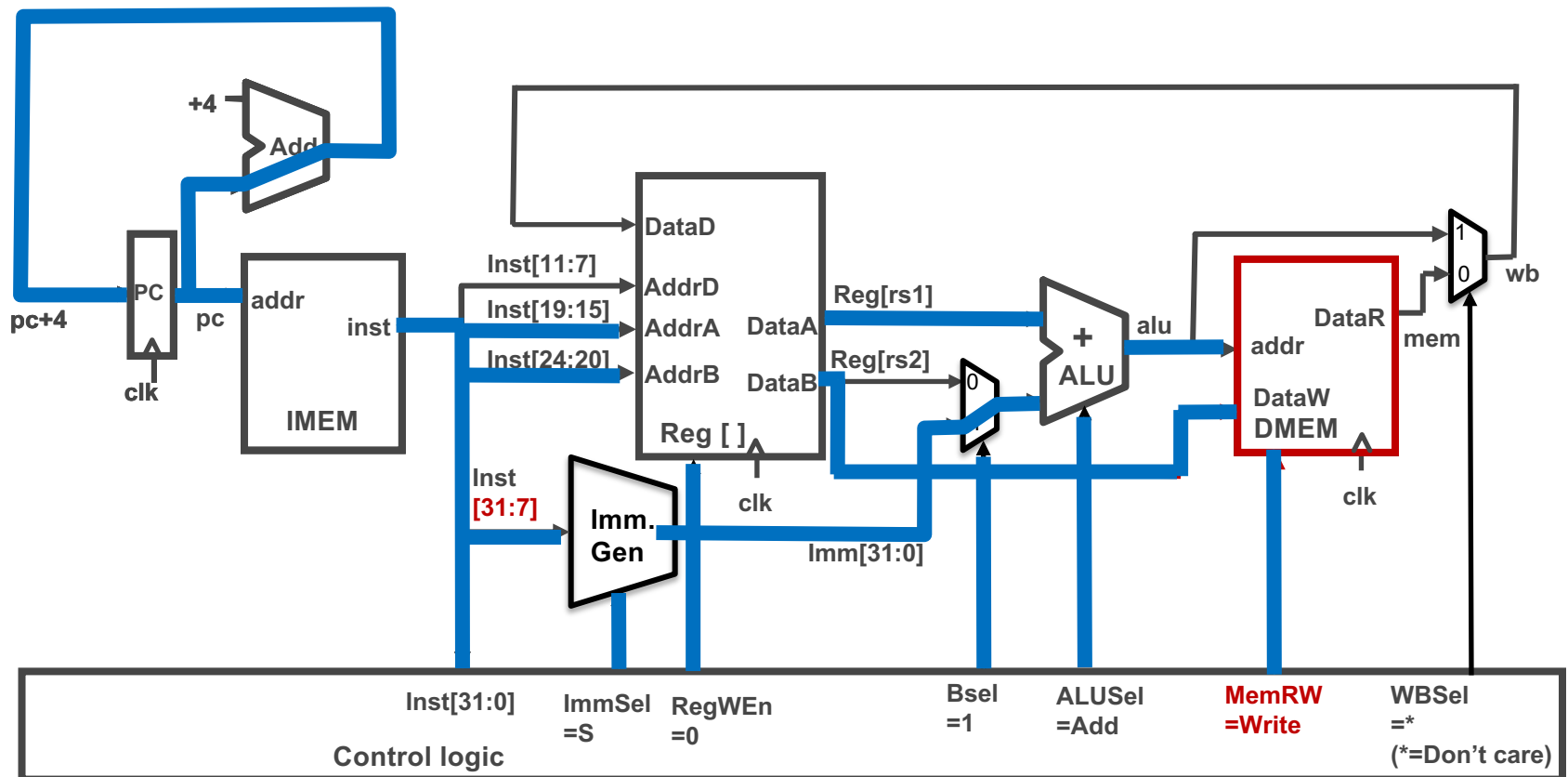
Adding `sw` Instruction

- 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] rs2=14   rs1=2   SW   offset[4:0] STORE
**   =0   =8**

0000000	01000

combined 12-bit offset = 8

Datapath with `lw`

Adding sw to Datapath

I+S Immediate Generation

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

inst[31:0]

5

5

1

6

I   S   **I/S**

31	11 10	5 4	0	
inst[31] (sign extension)	inst[30:25]	inst[24:20]		**I**
inst[31] (sign extension)	inst[30:25]	inst[11:7]		**S**

imm[31:0]
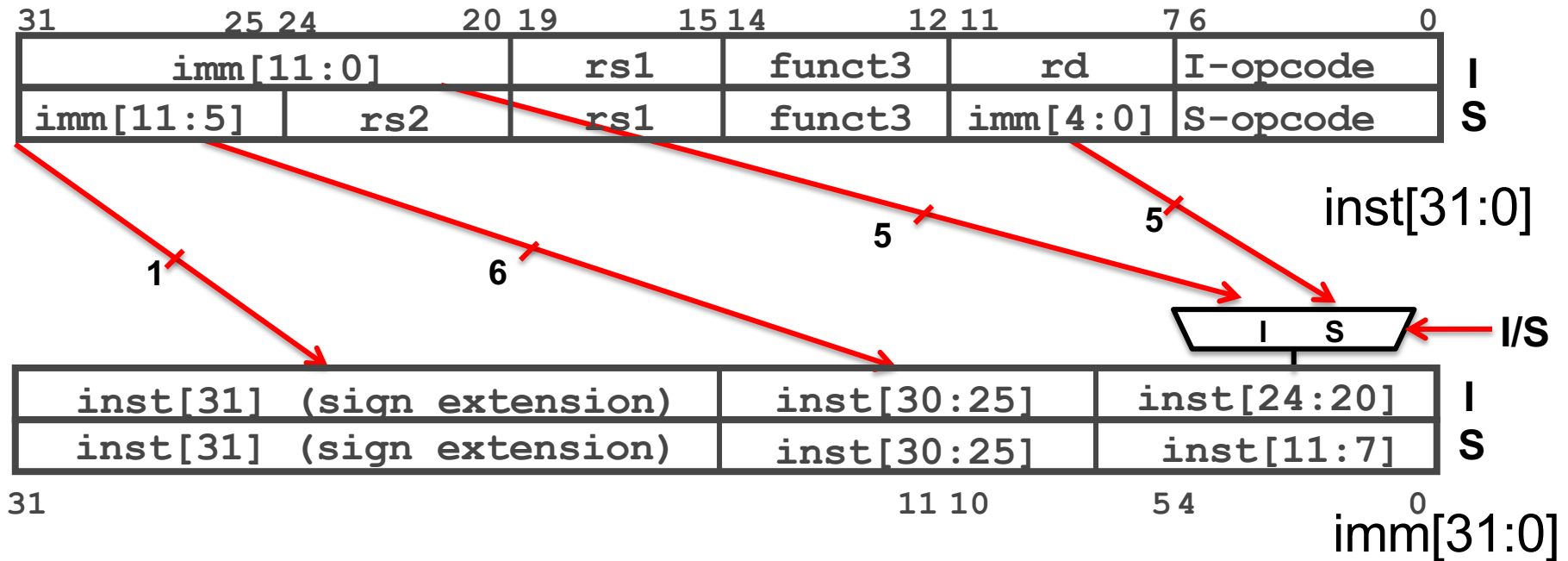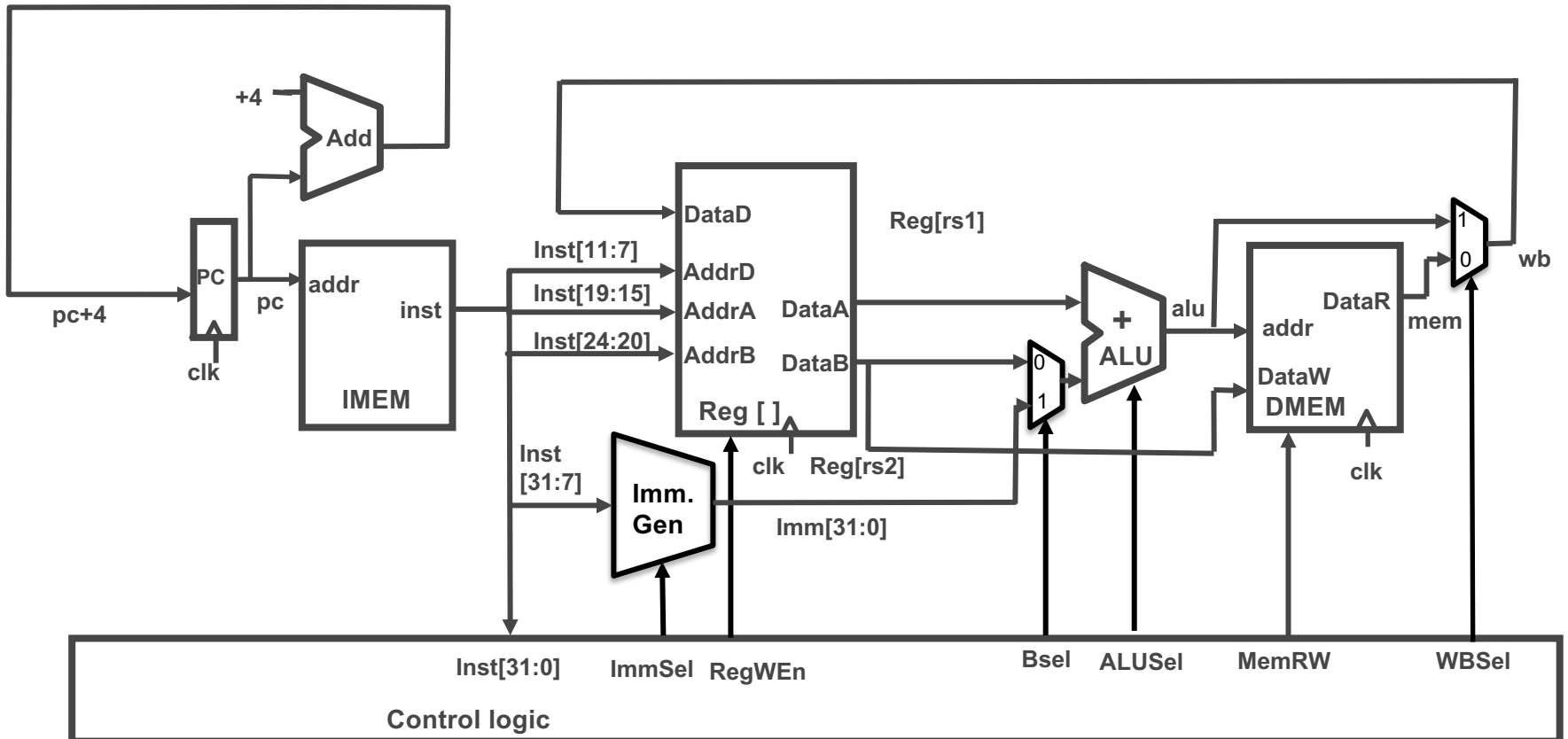
- Just need a 5-bit mux to select between two positions where low five bits of immediate can reside in instruction
- Other bits in immediate are wired to fixed positions in instruction

Datapath So Far

CS 110
Computer Architecture
Lecture 10:
Datapath
Video 3: Branches

Instructors:

Sören Schwertfeger & Chundong Wang

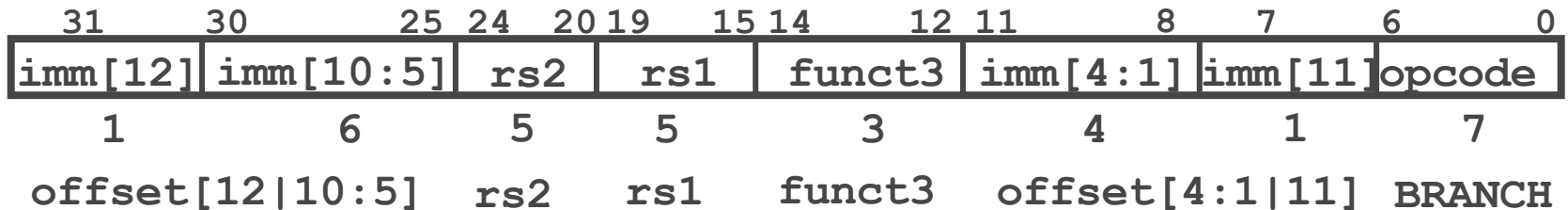https://robotics.shanghaitech.edu.cn/courses/ca/20s/

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Implementing Branches

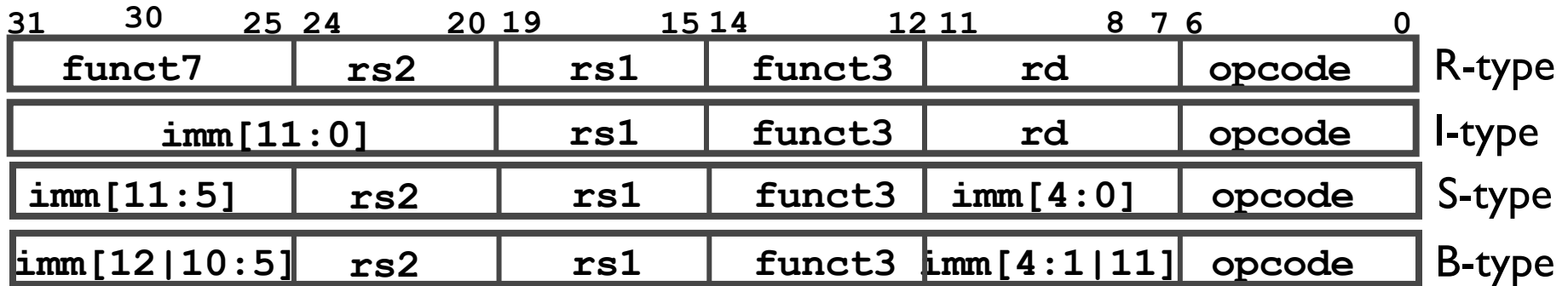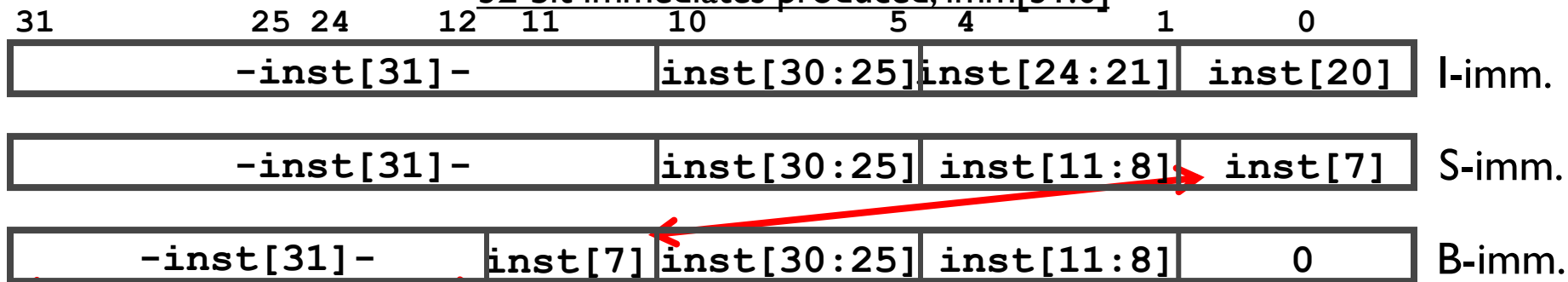| 31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		rs2	rs1	funct3	offset[4:1|11]		BRANCH

- 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)

RISC-V Immediate Encoding

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]

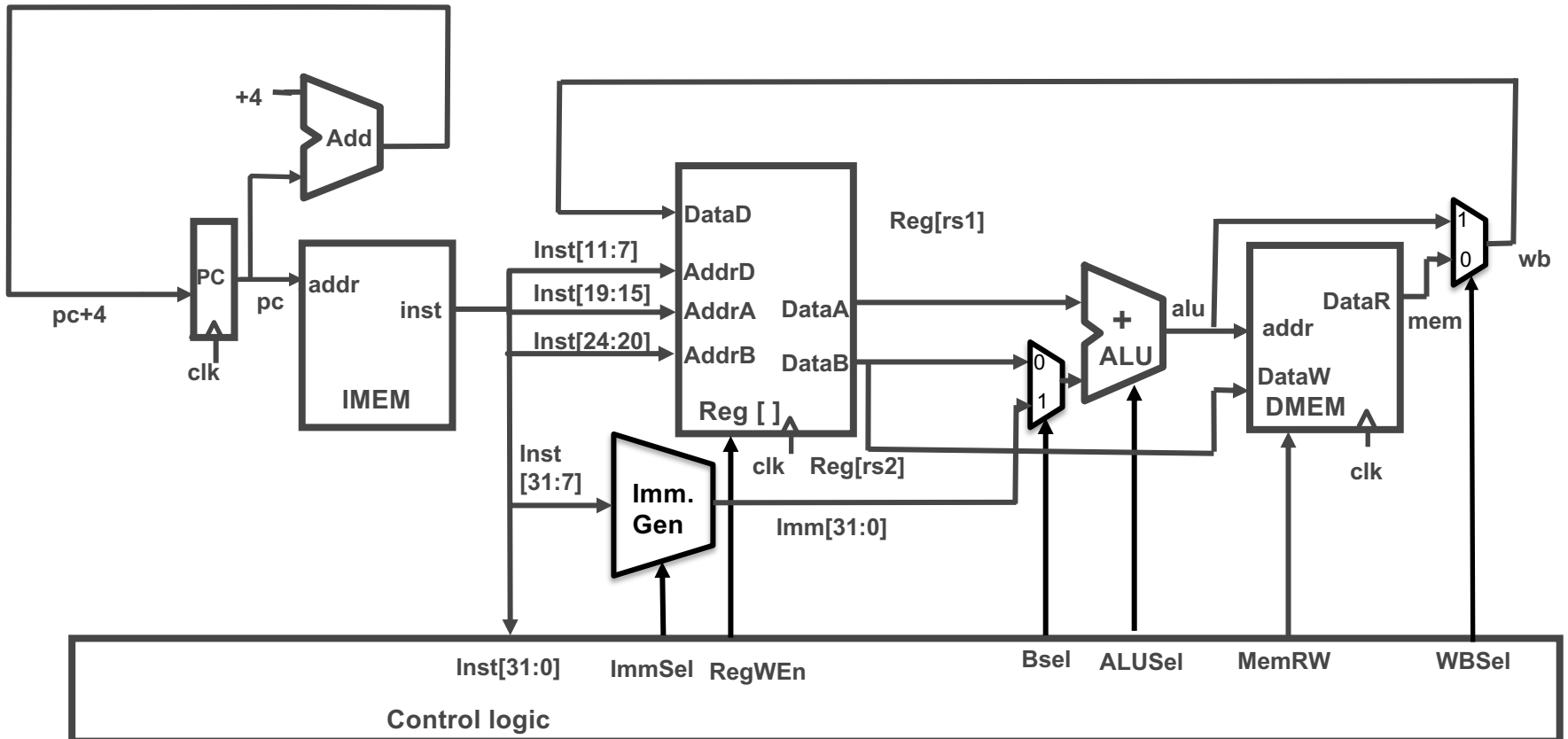| 31	25 24	12	11	10	5 4	1	0	
-inst[31]-				inst[30:25]	inst[24:21]		inst[20]	I-imm.
-inst[31]-				inst[30:25]	inst[11:8]		inst[7]	S-imm.
-inst[31]-			inst[7]	inst[30:25]	inst[11:8]		0	B-imm.

Upper bits sign-extended from inst[31] always

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

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

Datapath So Far

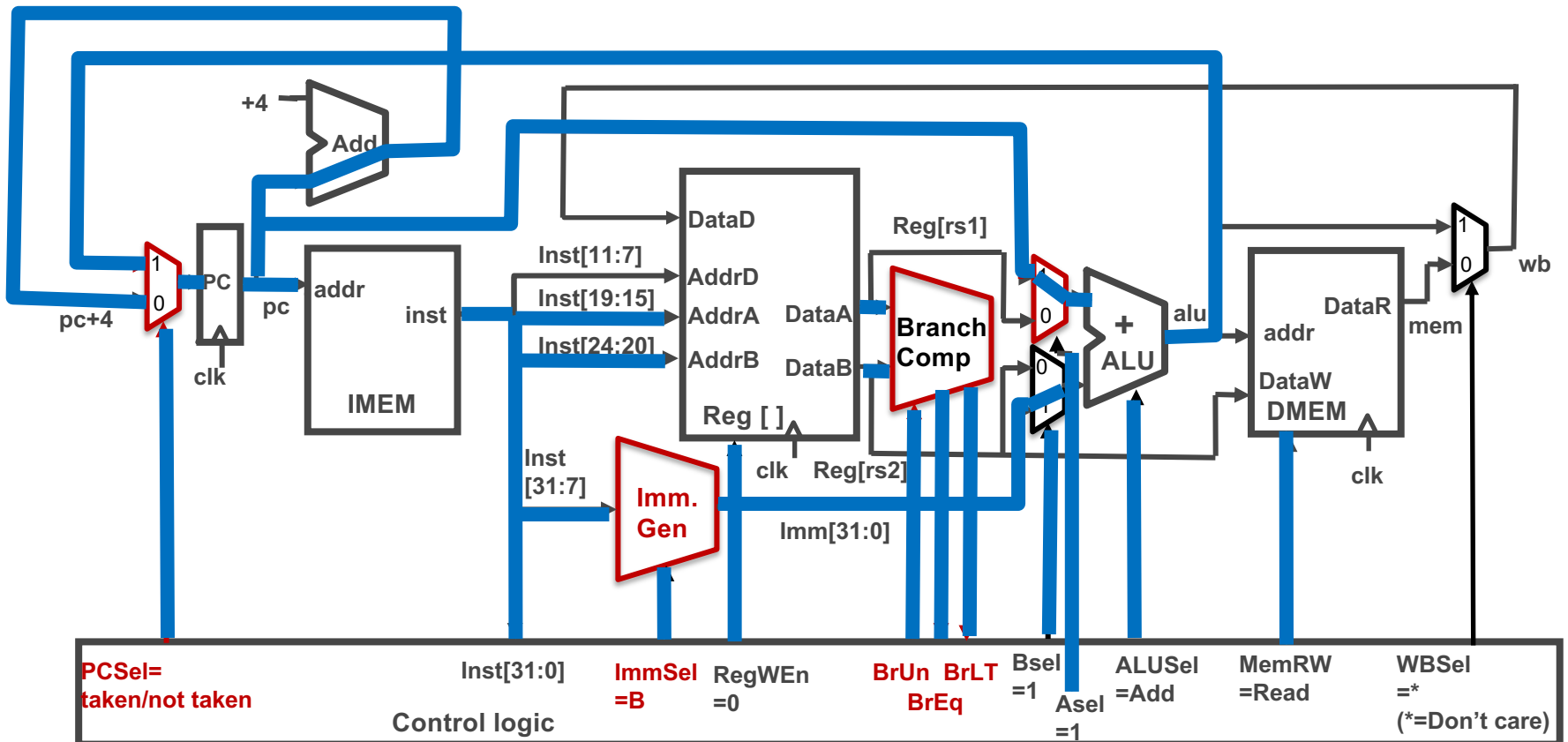# Branches

- Different change to the state:

$$- \text{PC} = \begin{cases} \text{PC + 4,} & \text{branch not taken} \\ \text{PC + immediate,} & \text{branch taken} \end{cases}$$
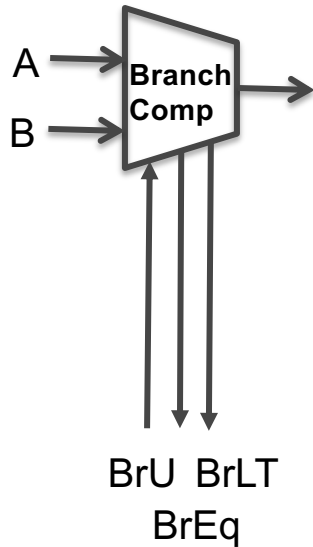
- Six branch instructions:
  `BEQ, BNE, BLT, BGE, BLTU, BGEU`

- Need to compute `PC + immediate` and to compare values of `rs1` and `rs2`
  - But have only one ALU – need more hardware

Adding Branches

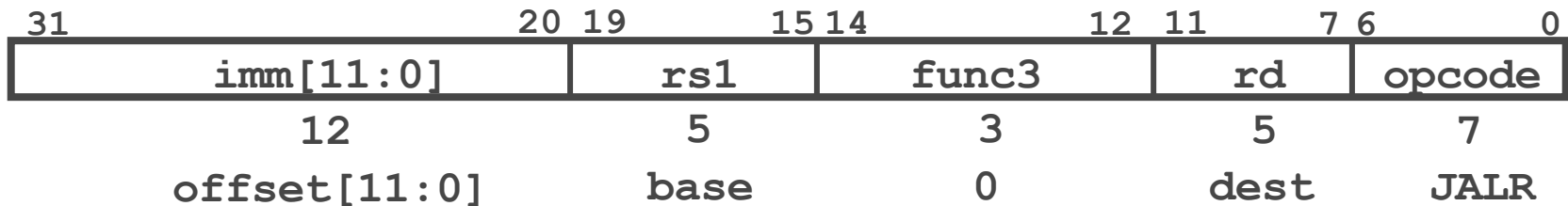# Branch Comparator

- BrEq = 1, if A=B

- BrLT = 1, if A < B

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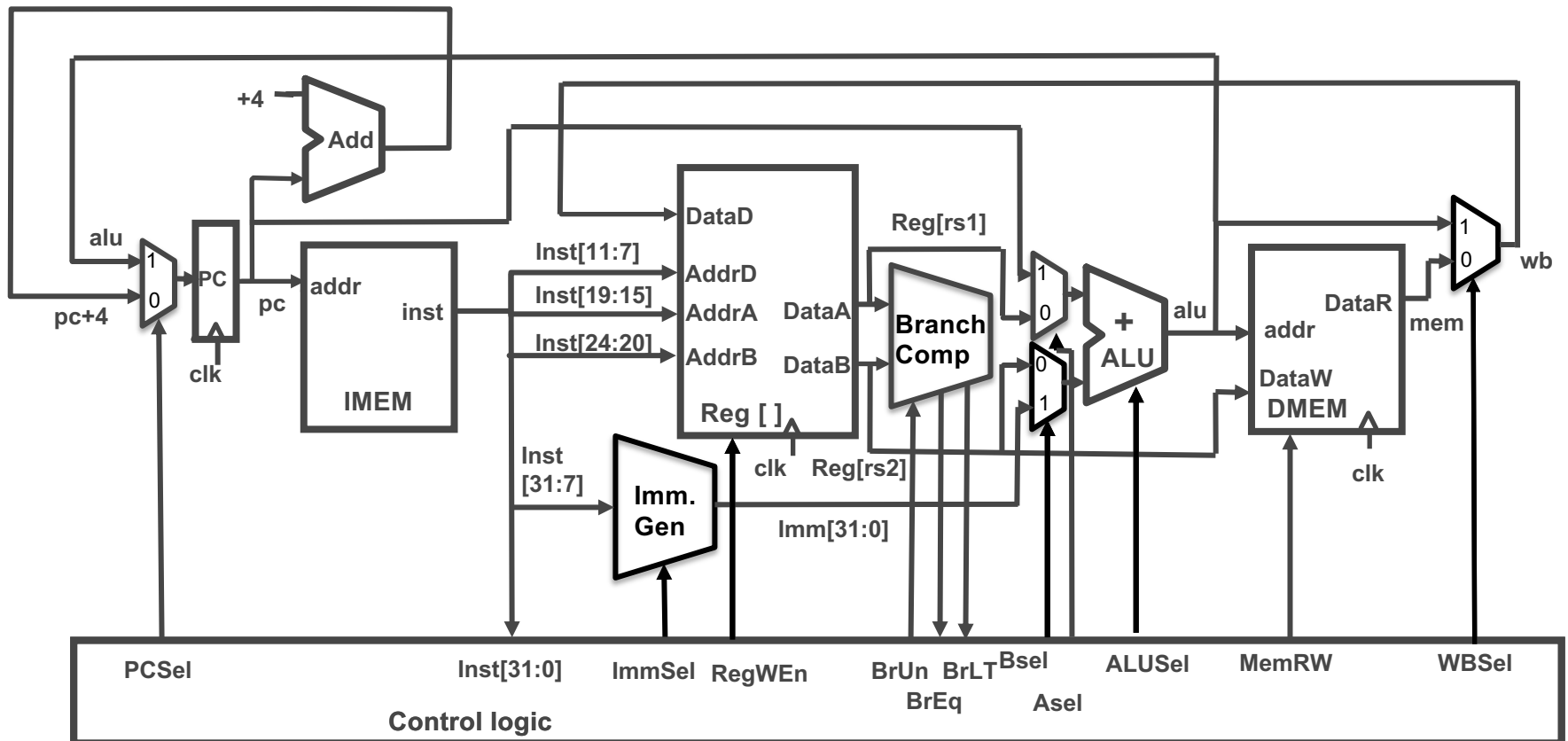
- BGE branch: A >= B, if $\overline{A<B}$

$$\overline{A<B} = !(A<B)$$

Let's Add JALR (I-Format)

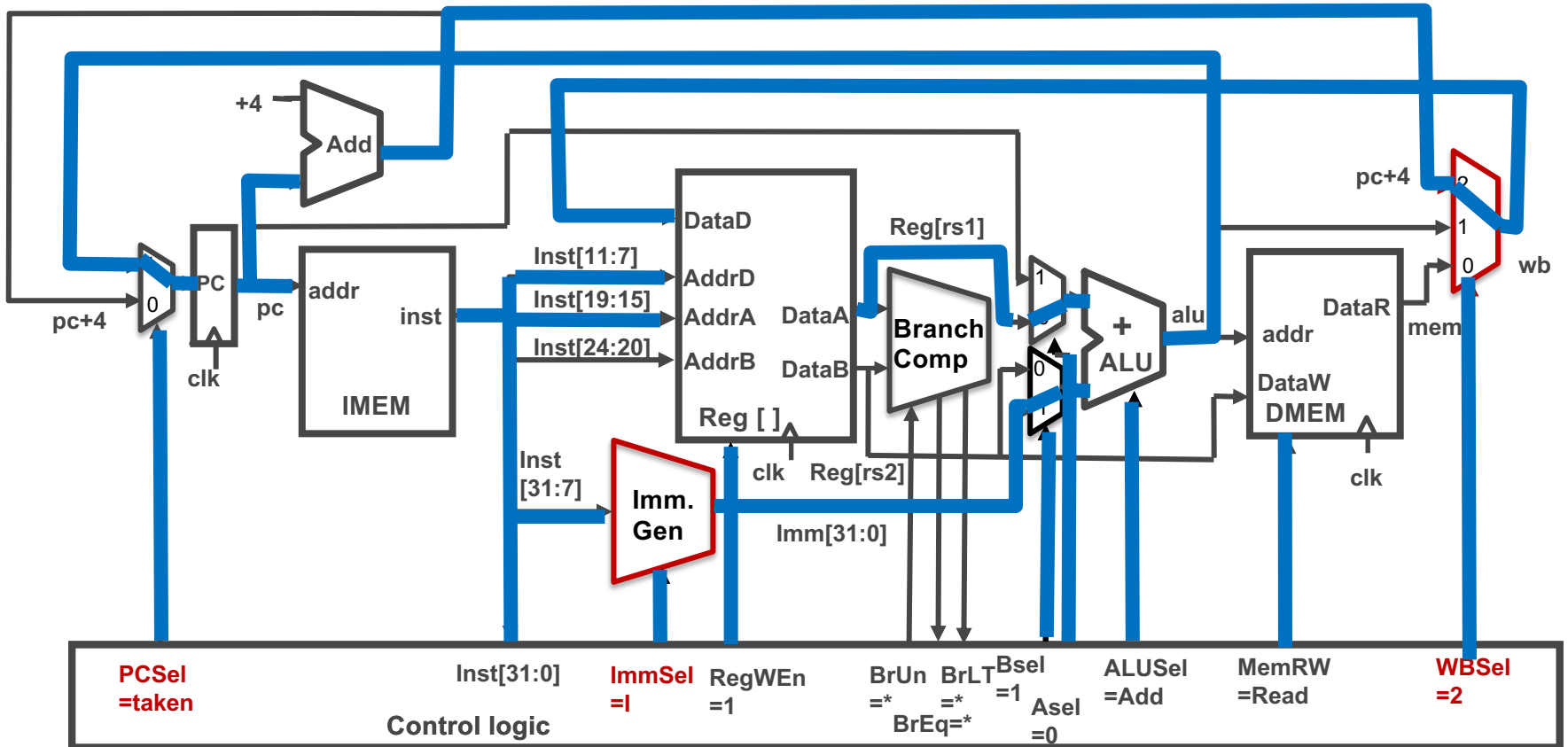| 31	20 19	15 14	12 11	7 6	0
imm[11:0]		rs1	func3	rd	opcode
12		5	3	5	7
offset[11:0]		base	0	dest	JALR

- 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
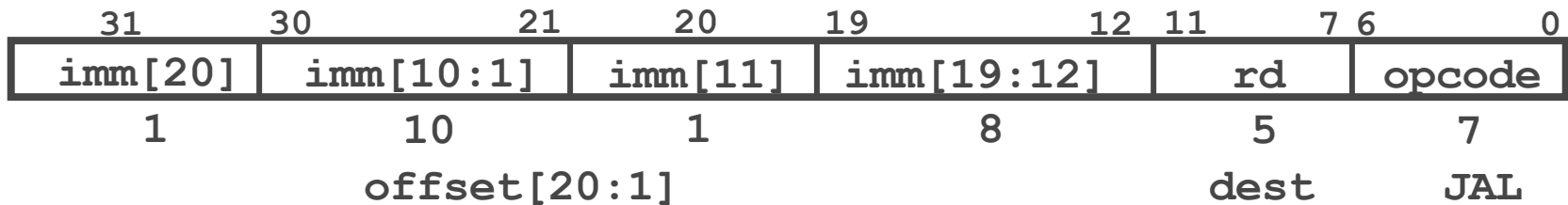   - *no* multiplication by 2 bytes
    - LSB is ignored

Datapath So Far, with Branches

Adding JALR

58

Adding JAL

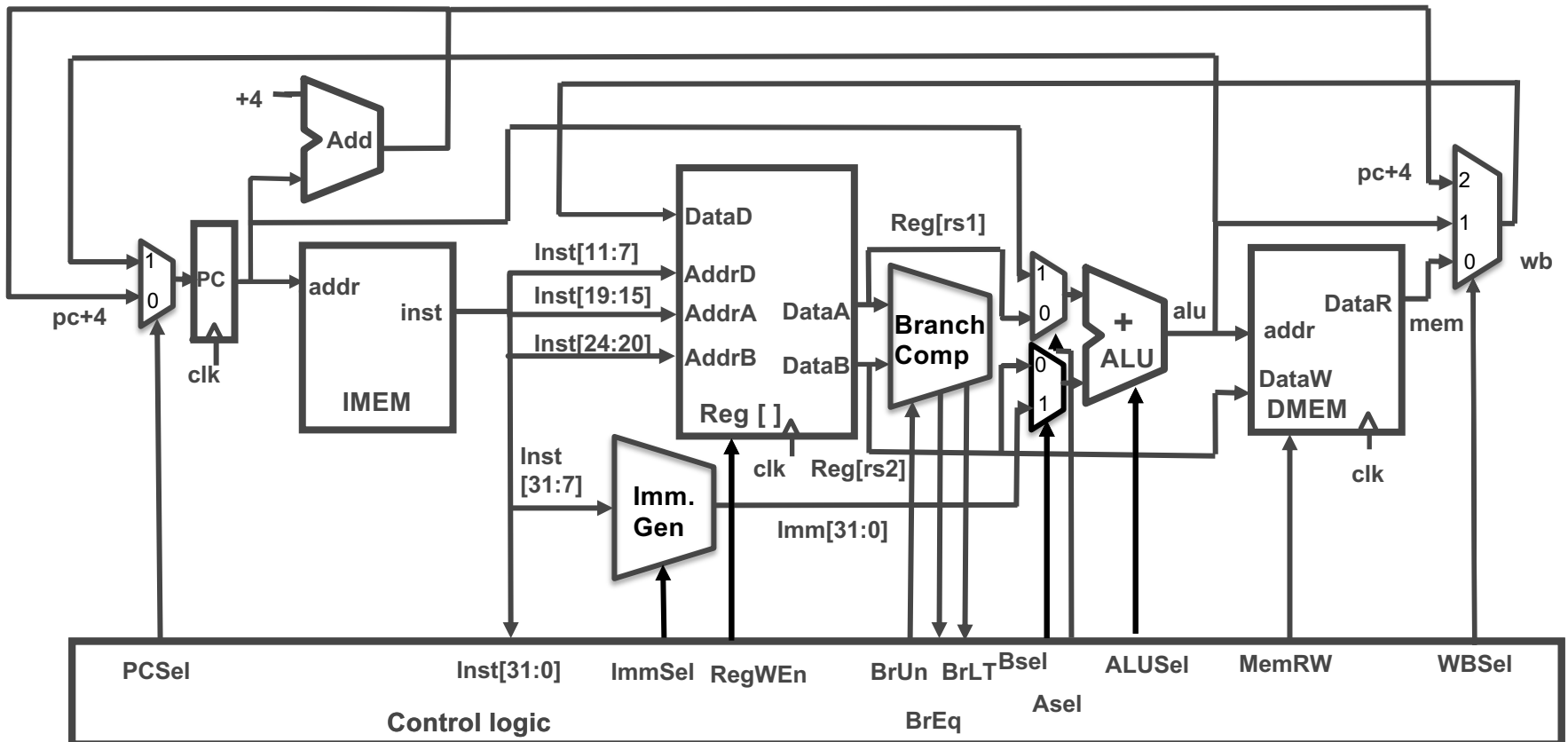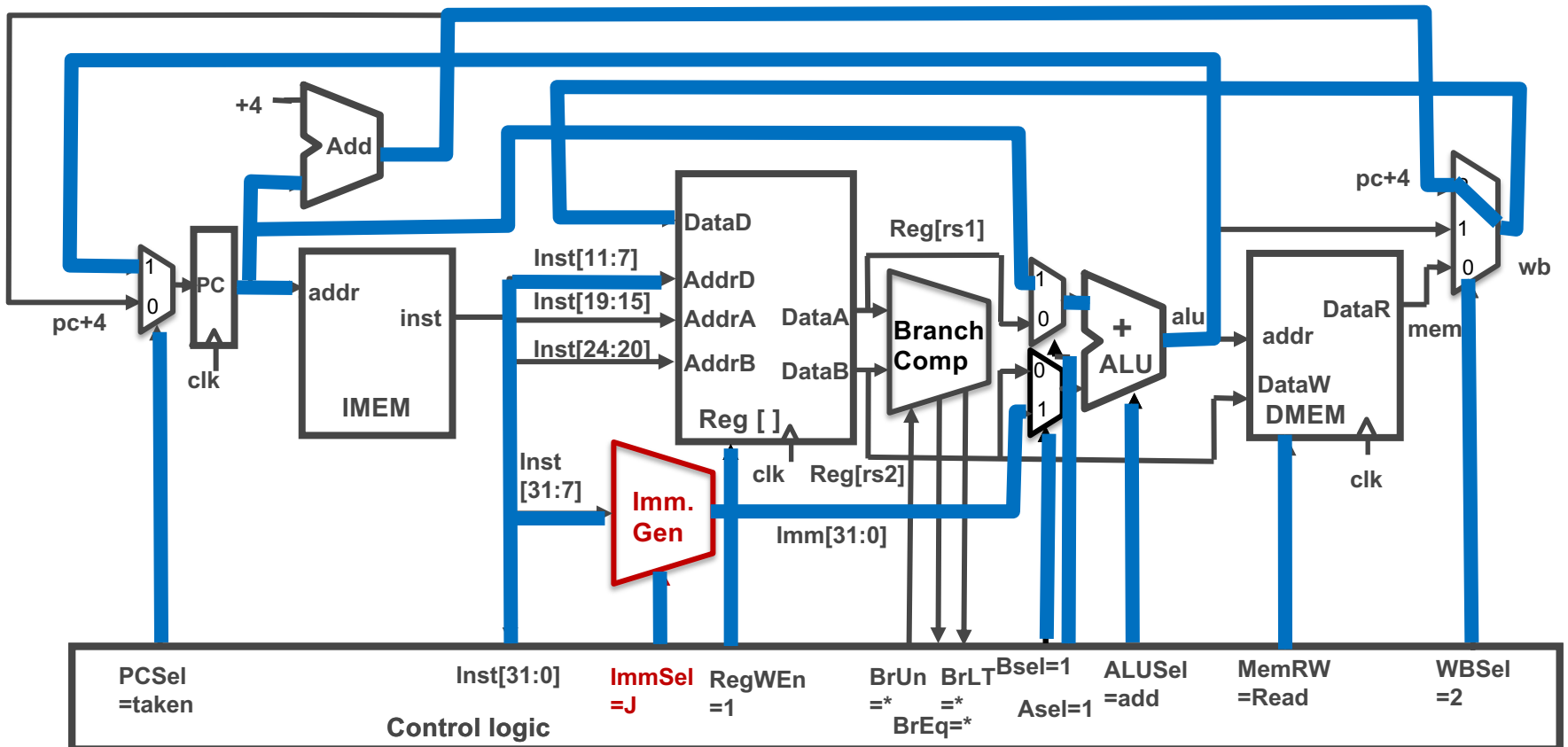| 31	30	21	20	19	12	11	7	6	0
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	
1	10		1	8		5		7	

<div align="center">offset[20:1]   dest   JAL</div>

- JAL saves PC+4 in register rd (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm2^{19}$ locations, 2 bytes apart
  - $\pm2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

Datapath with JALR

Adding JAL

U-Format for "Upper Immediate" Instructions

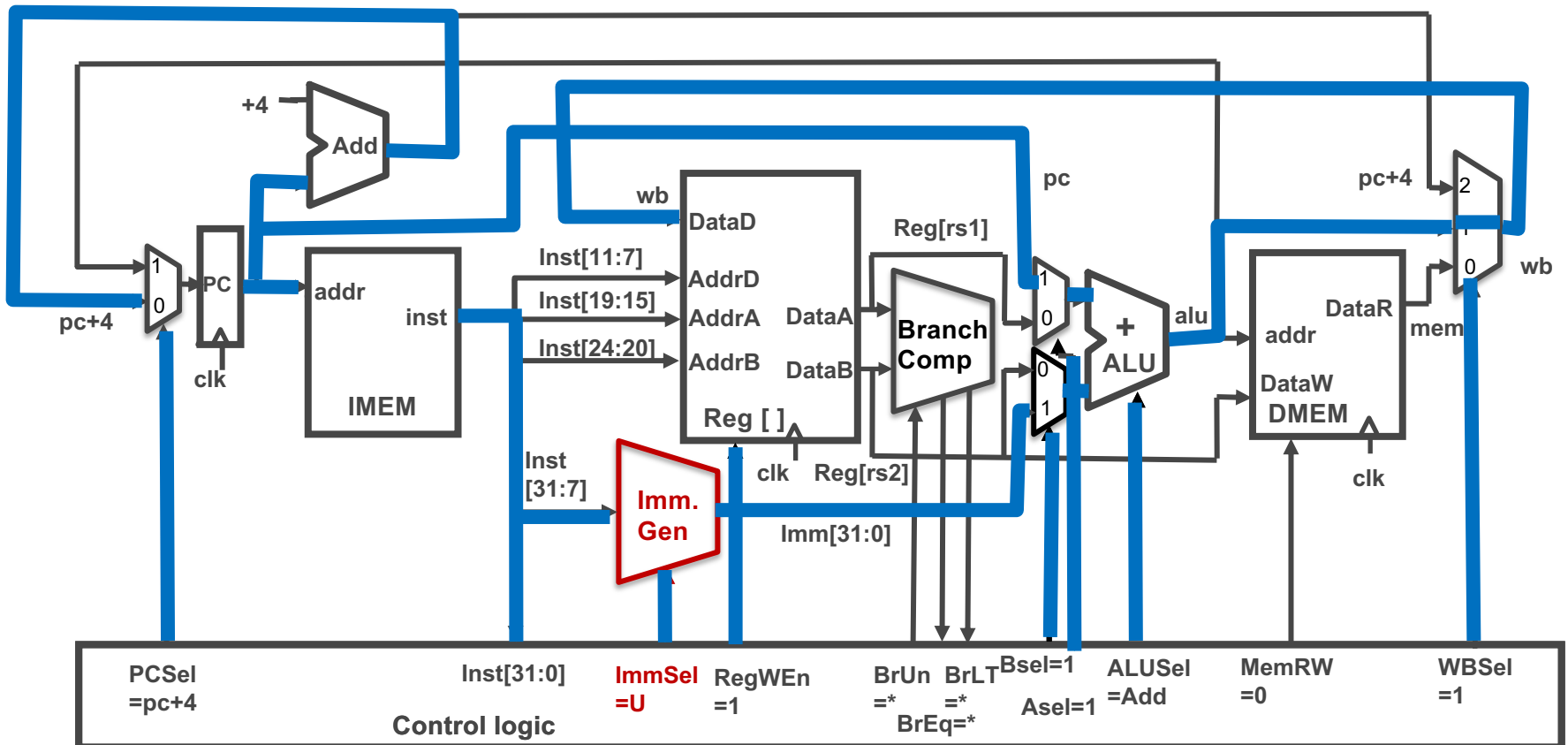| 31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC
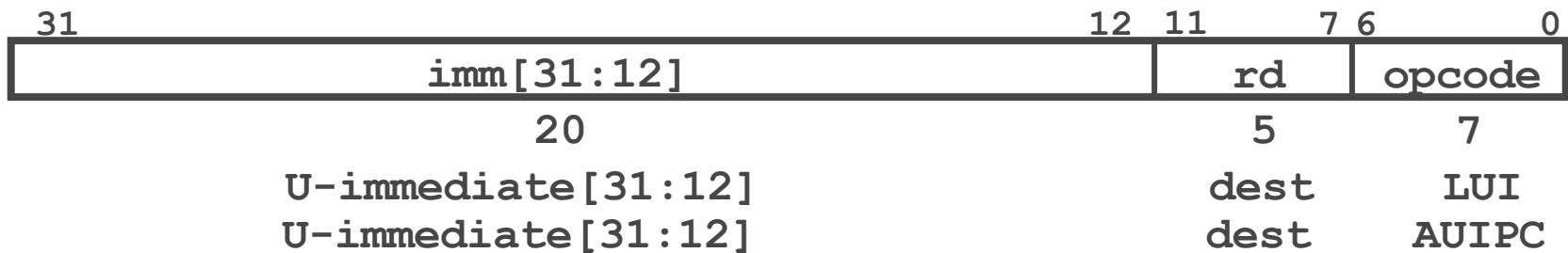
- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
  - LUI – Load Upper Immediate
  - AUIPC – Add Upper Immediate to PC

# Implementing LUI
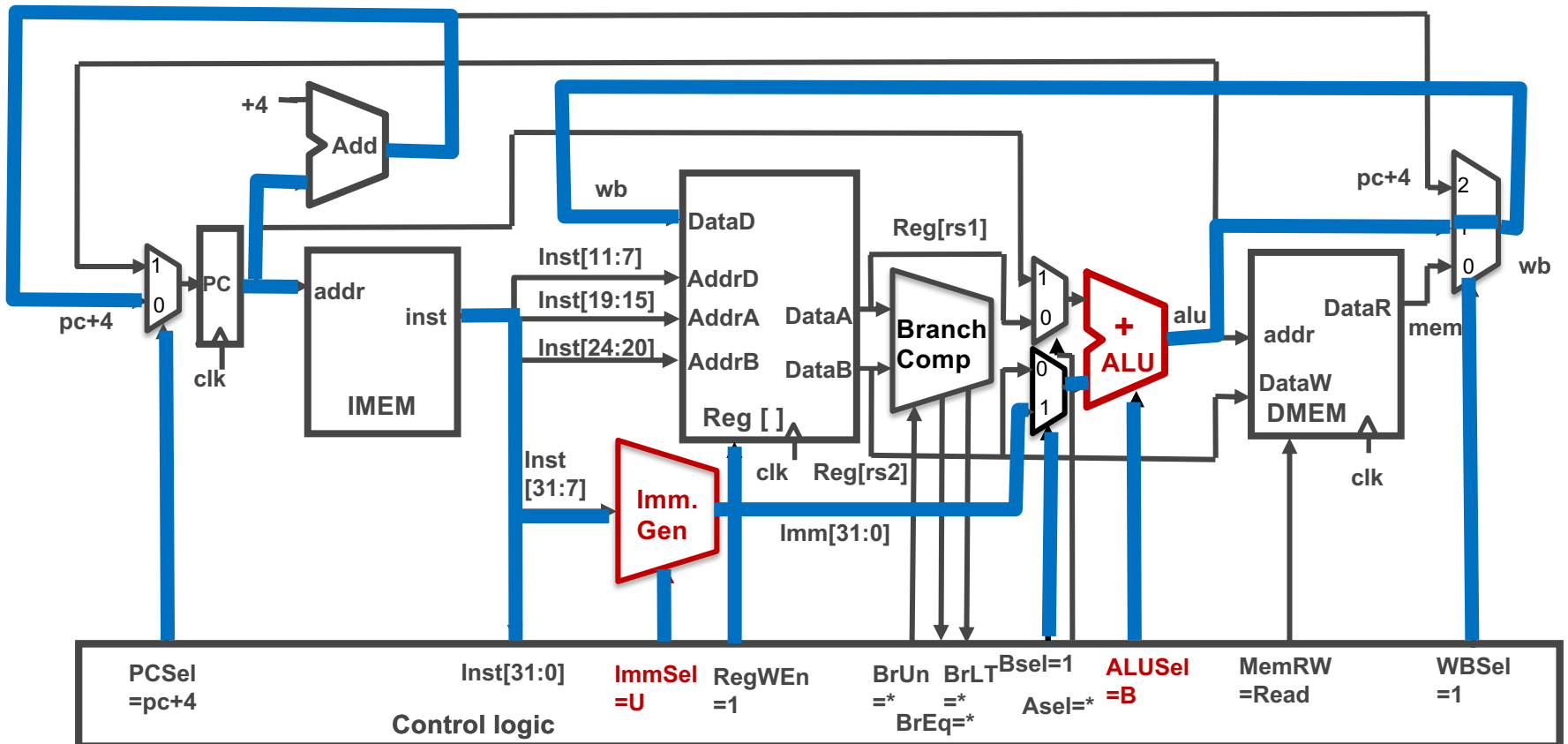
Implementing `AUIPC`

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	EBREAK
csr		rs1	001	rd	1110011	CSRRW
csr		rs1		rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRW
csr		zimm	110	rd	1110011	CSRRS
csr		zimm	111	rd	1110011	CSRRC

Not in CA

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

Complete RV32I Datapath!
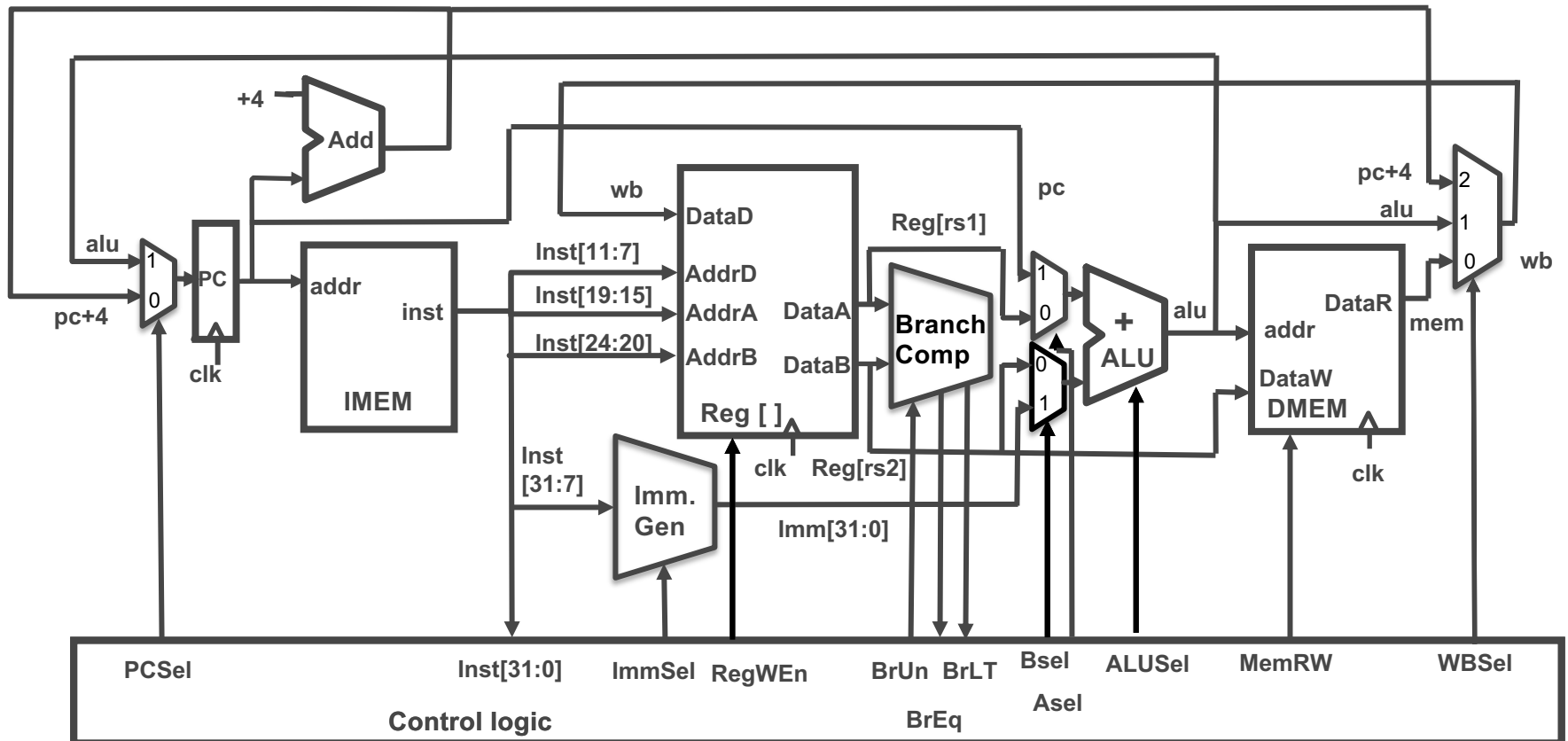
"And In conclusion…"

- We have designed a complete datapath
  - Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions

- 5 Phases of execution
  - IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases

- Controller specifies how to execute instructions
  - New instructions can be added with just control?

Question

- Select the statements that are TRUE:

A. Instructions that don't need certain stages (e.g. Memory stage) can run with a higher clock speed.

B. Control signals are usually connected to a mux.

C. The datapath from this lecture is single cycle – so it only contains combinatorial logic elements.

D. For some instructions, certain control signals are undefined.

E. All I-type instructions sign-extend the immediate.