

# Computer Architecture 1

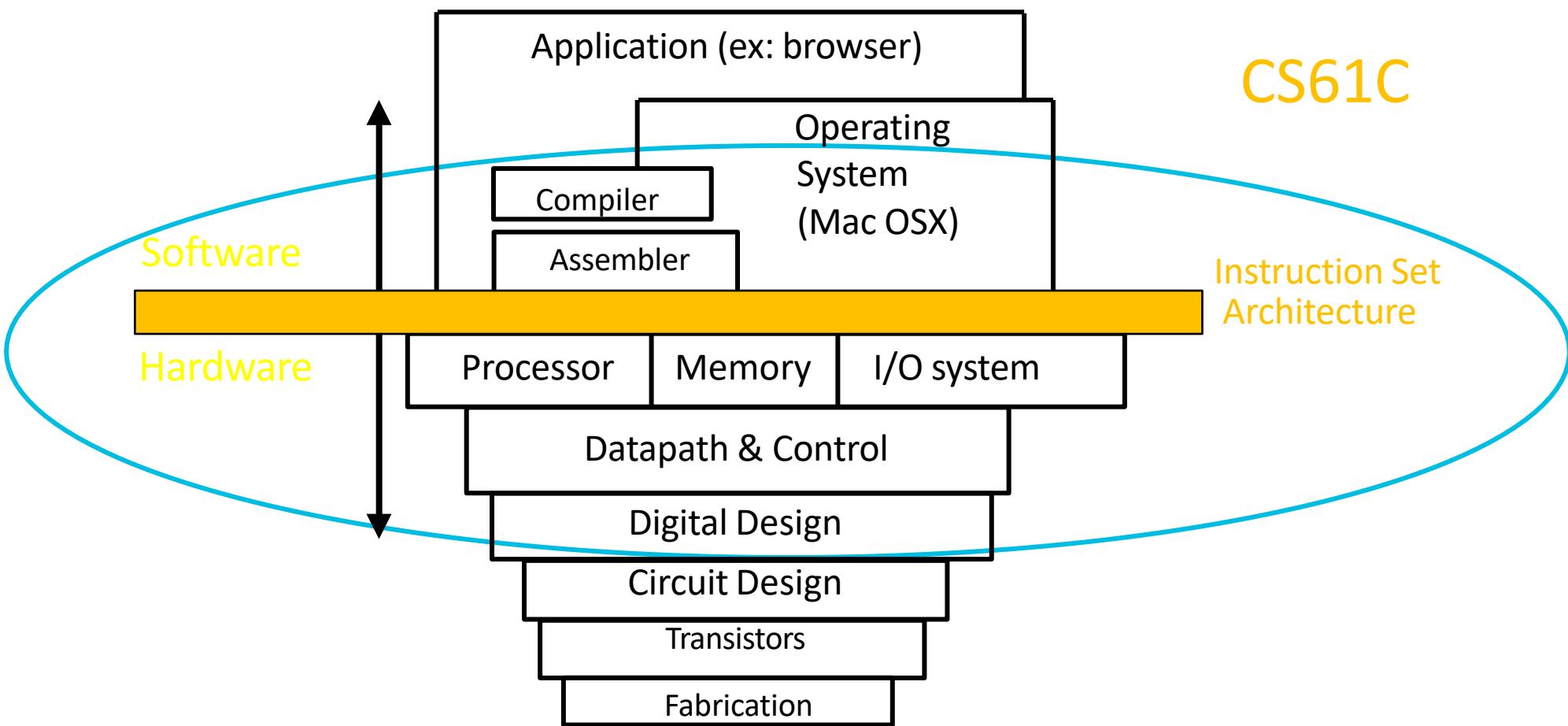
Computer Organization and Design  
THE HARDWARE/SOFTWARE INTERFACE

[Adapted from *Computer Organization and Design, RISC-V Edition*, Patterson & Hennessy, © 2018, MK]

[Adapted from *Great ideas in Computer Architecture (CS 61C)* lecture slides, Garcia and Nikolic, © 2020, UC Berkeley]

# RISC-V Processor Design

# Machine Structures



# New-School Machine Structures

## Software

Parallel Requests

Assigned to computer

e.g., Search "Cats"

## Parallel Threads

Assigned to core e.g., Lookup, Ads

## Parallel Instructions

>1 instruction @ one time

e.g., 5 pipelined instructions

## Parallel Data

>1 data item @ one time

e.g., Add of 4 pairs of words

## Hardware descriptions

All gates work in parallel at same time

Harness  
Parallelism &  
Achieve High  
Performance

## Hardware



Warehouse Scale Computer

Smart Phone



Computer

Core

Memory

(Cache)

Input/Output

Exec. Unit(s)

Functional Block(s)

$A_0 + B_0$

$A_1 + B_1$

Main Memory

Logic Gates



# Great Idea #1: Abstraction (Levels of Representation/Interpretation)

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

Compiler

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

Assembler

Machine Language  
Program (RISC-V)

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

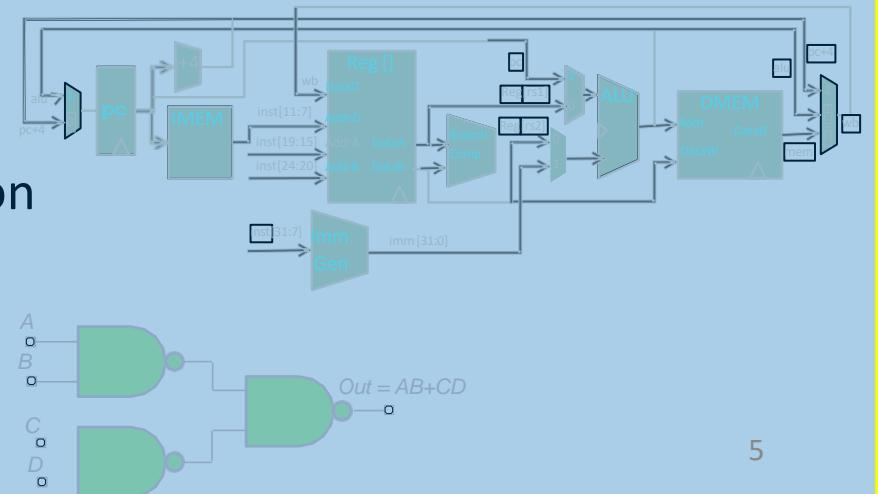
```
lw    x3, 0(x10)  
lw    x4, 4(x10)  
sw    x4, 0(x10)  
sw    x3, 4(x10)
```

1000	ii	01	1110	0010	0000	0000	0000	0000	0000	0000
1000	1110	0001	0000	0000	0000	0000	0000	0100		
1010	1110	0001	0010	0000	0000	0000	0000	0000	0000	
1010	1101	1110	0010	0000	0000	0000	0000	0100		

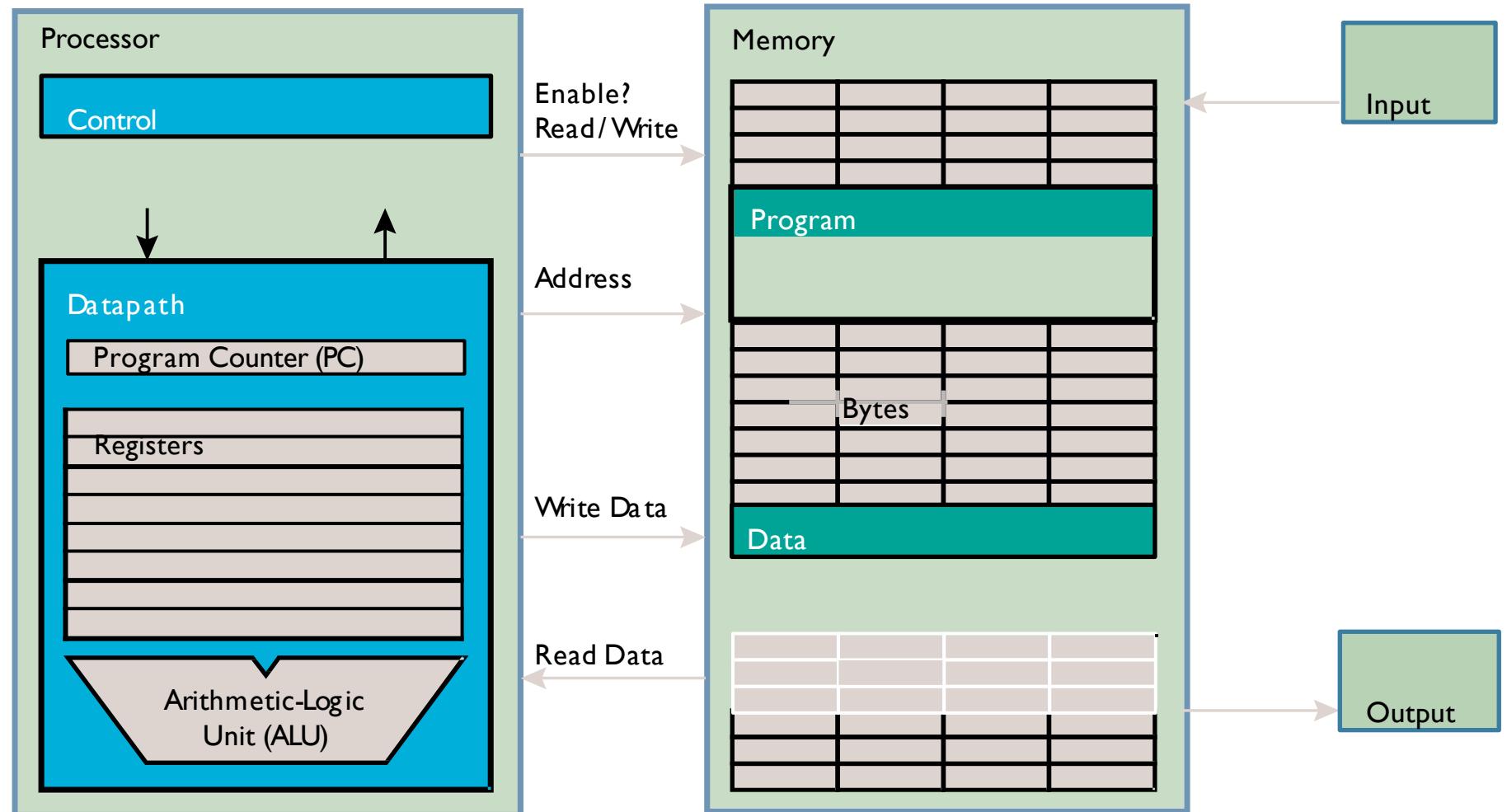
Hardware Architecture Description  
(e.g., block diagrams)

Architecture Implementation

Logic Circuit Description  
(Circuit Schematic Diagrams)



# Our Single-Core Processor So Far...



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

# Need to Implement All RV32I Instructions



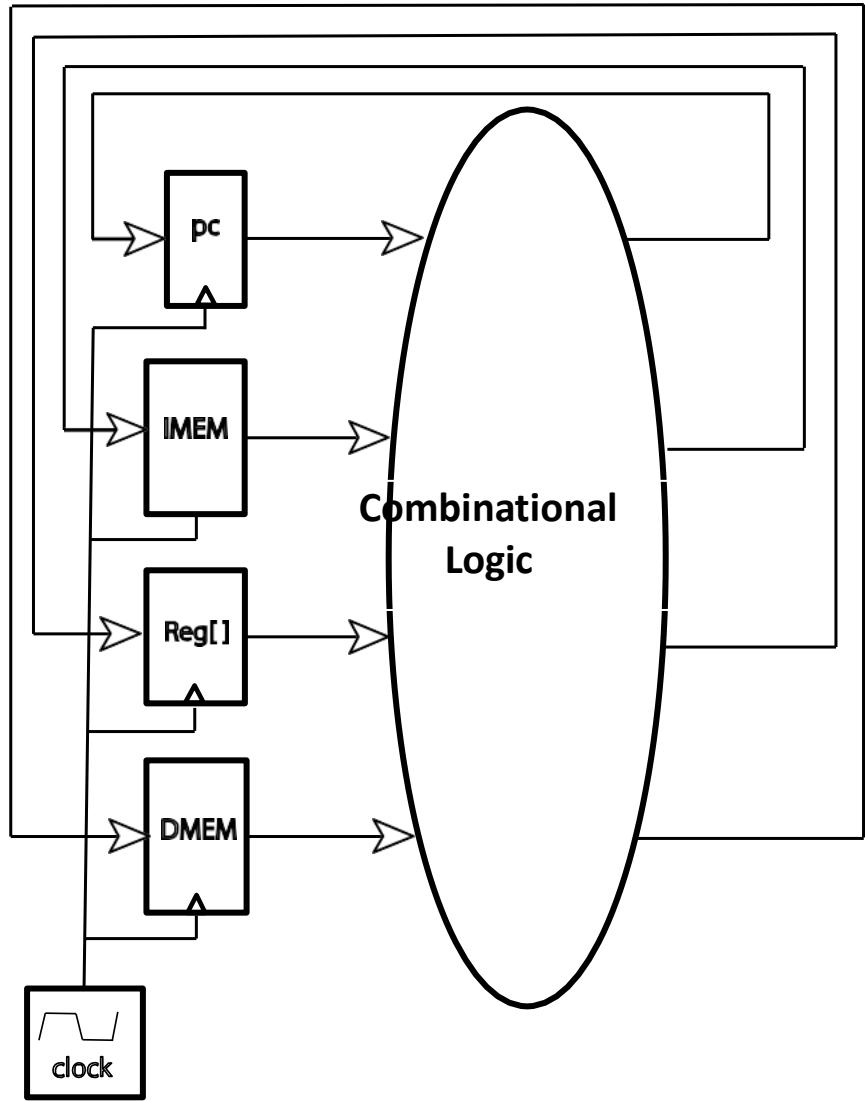
Open Reference Card							
Base Integer Instructions: RV32I							
Category	Name	Fmt	RV32I Base	Category	Name	Fmt	RV32I Base
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2	Loads	Load Byte	I	LB rd,rs1,imm
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt		Load Halfword	I	LH rd,rs1,imm
	Shift Right Logical	R	SRL rd,rs1,rs2		Load Byte Unsigned	I	LBU rd,rs1,imm
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt		Load Half Unsigned	I	LHU rd,rs1,imm
	Shift Right Arithmetic	R	SRA rd,rs1,rs2		Load Word	I	LW rd,rs1,imm
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	Stores	Store Byte	S	SB rs1,rs2,imm
Arithmetic	ADD	R	ADD rd,rs1,rs2		Store Halfword	S	SH rs1,rs2,imm
	ADD Immediate	I	ADDI rd,rs1,imm		Store Word	S	SW rs1,rs2,imm
	SUBtract	R	SUB rd,rs1,rs2	Branches	Branch =	B	BEQ rs1,rs2,imm
Logical	Load Upper Imm	U	LUI rd,imm		Branch ≠	B	BNE rs1,rs2,imm
	Add Upper Imm to PC	U	AUIPC rd,imm		Branch <	B	BLT rs1,rs2,imm
	XOR	R	XOR rd,rs1,rs2		Branch ≥	B	BGE rs1,rs2,imm
	XOR Immediate	I	XORI rd,rs1,imm		Branch < Unsigned	B	BLTU rs1,rs2,imm
	OR	R	OR rd,rs1,rs2		Branch ≥ Unsigned	B	BGEU rs1,rs2,imm
	OR Immediate	I	ORI rd,rs1,imm	Jump & Link	J&L	J	JAL rd,imm
	AND	R	AND rd,rs1,rs2		Jump & Link Register	I	JALR rd,rs1,imm
	AND Immediate	I	ANDI rd,rs1,imm				
	Compare	R	SLT rd,rs1,rs2	Synch	Synch thread	I	FENCE
	Set <	R	SLT rd,rs1,rs2				
	Set < Immediate	I	SLTI rd,rs1,imm				
	Set < Unsigned	R	SLTU rd,rs1,rs2	Environment	CALL	I	ECALL
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm		BREAK	I	EBREAK



Not in  
6IC

# **Building a RISC-V Processor**

# One-Instruction-Per-Cycle RISC-V Machine



- On every tick of the clock, the computer executes one instruction
- Current state outputs drive the inputs to the combinational logic, whose outputs settle at the values of the state before the next clock edge
- At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle

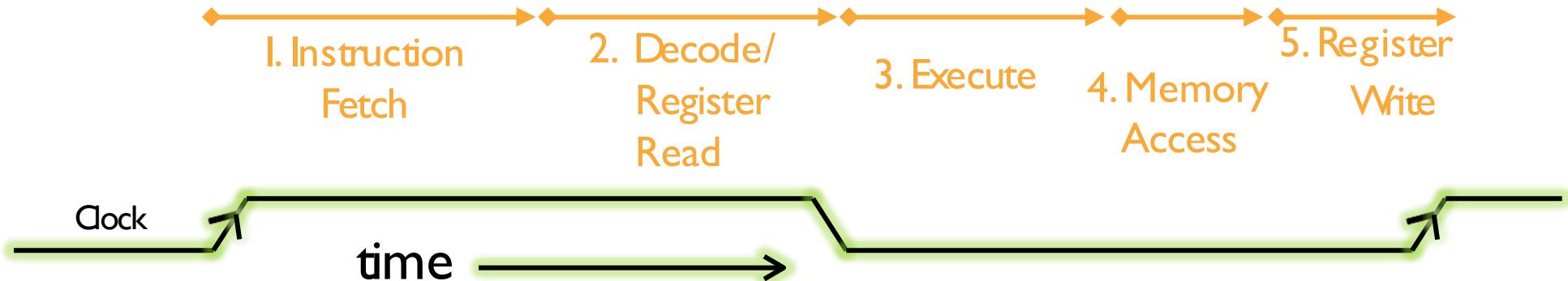
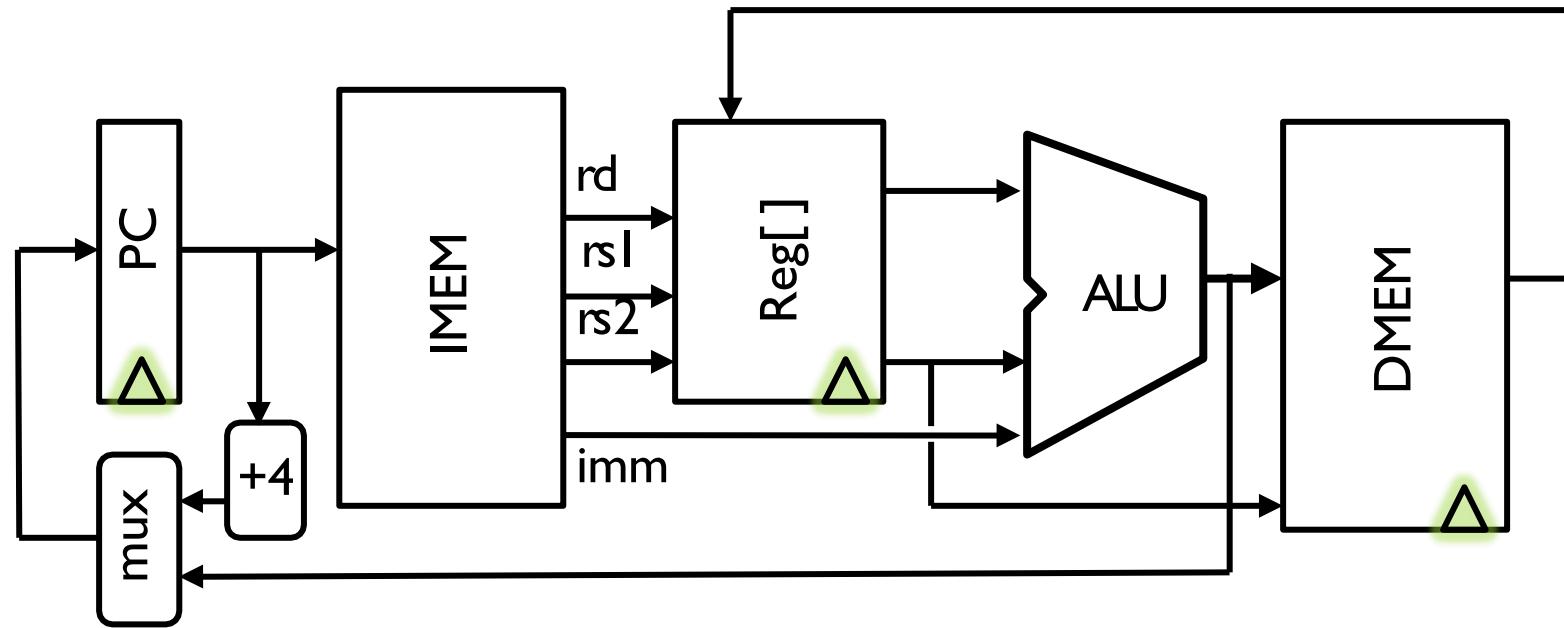
# Stages of the Datapath : Overview

- 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 the Datapath

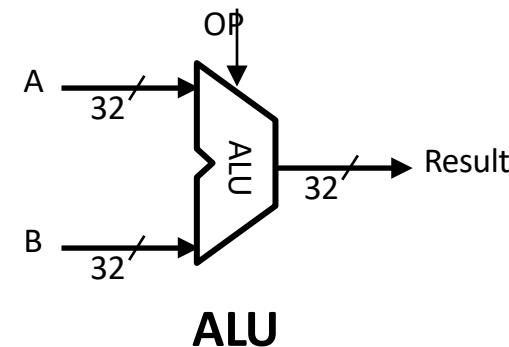
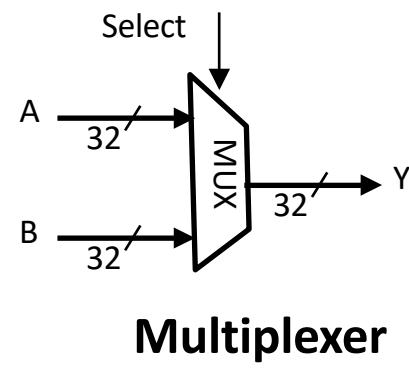
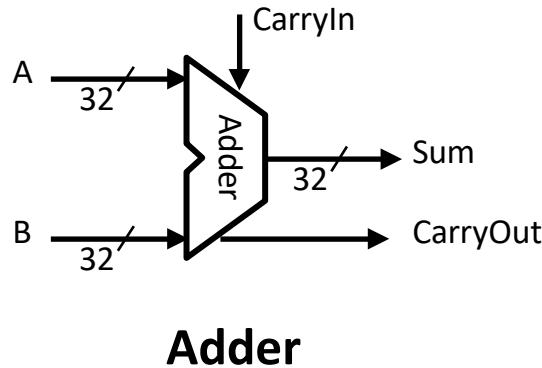
- Stage 1: *Instruction Fetch (IF)*
- Stage 2: *Instruction Decode (ID)*
- Stage 3: *Execute (EX) - ALU (Arithmetic-Logic Unit)*
- Stage 4: *Memory Access (MEM)*
- Stage 5: *Write Back to Register (WB)*

# Basic Phases of Instruction Execution



# Datapath Components: Combinational

- Combinational elements



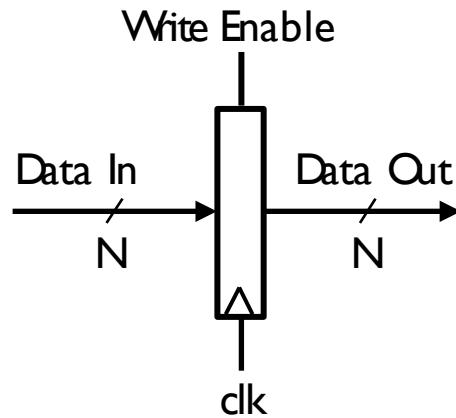
- Storage elements + clocking methodology
- Building blocks

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

- Register

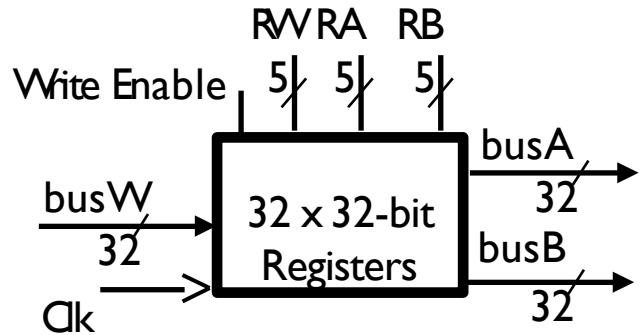
- Write Enable:

- Low (or deasserted) (0):  
Data Out will not change
- Asserted (1): Data Out will become Data In on  
positive edge of clock



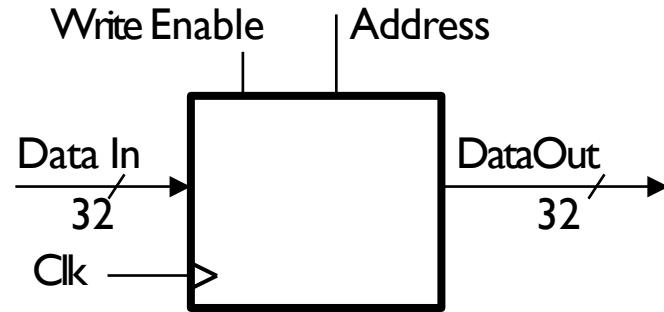
# Datapath Elements: State and Sequencing (2/3)

- Register file (regfile, RF) consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
- 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)

- “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 (1/2)

Each instruction during execution reads and updates the state of :

(1) Registers, (2) Program counter, (3) Memory

- 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

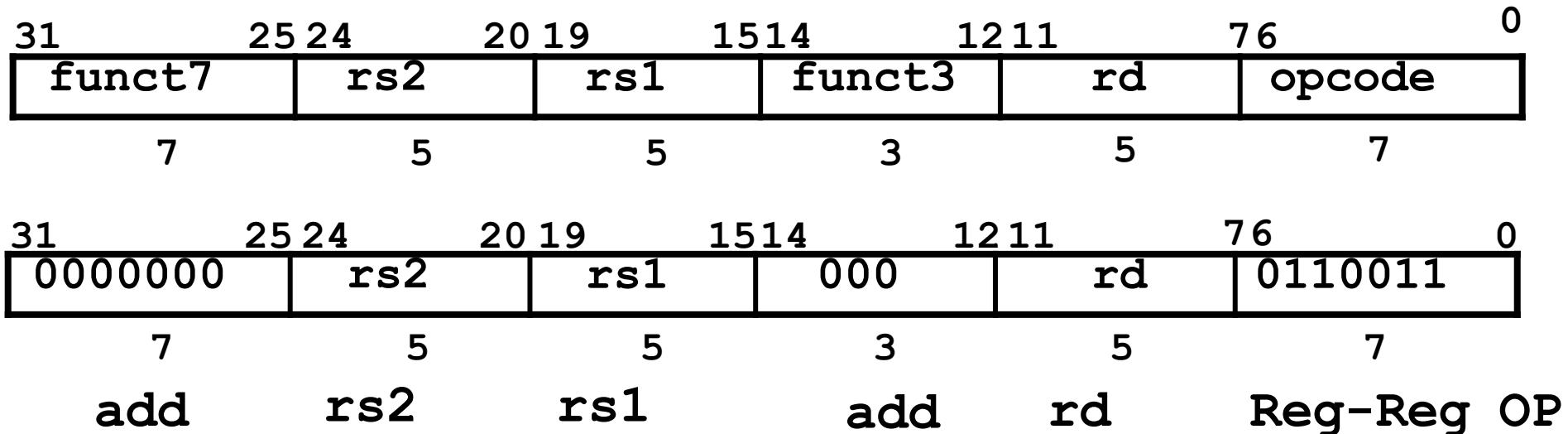
# State Required by RV32I ISA (2/2)

- 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

# R-Type Add Datapath



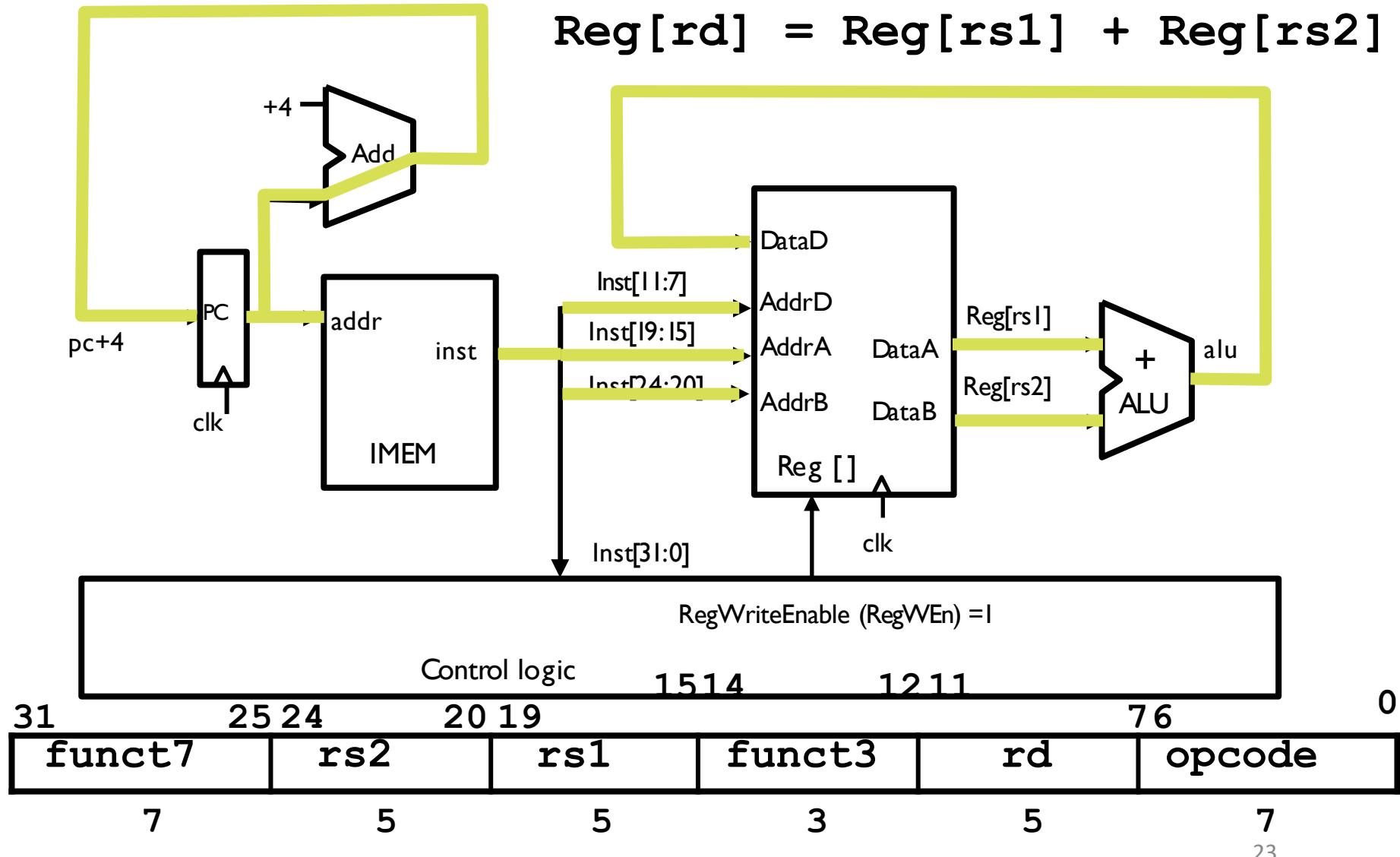
# Implementing the add instruction



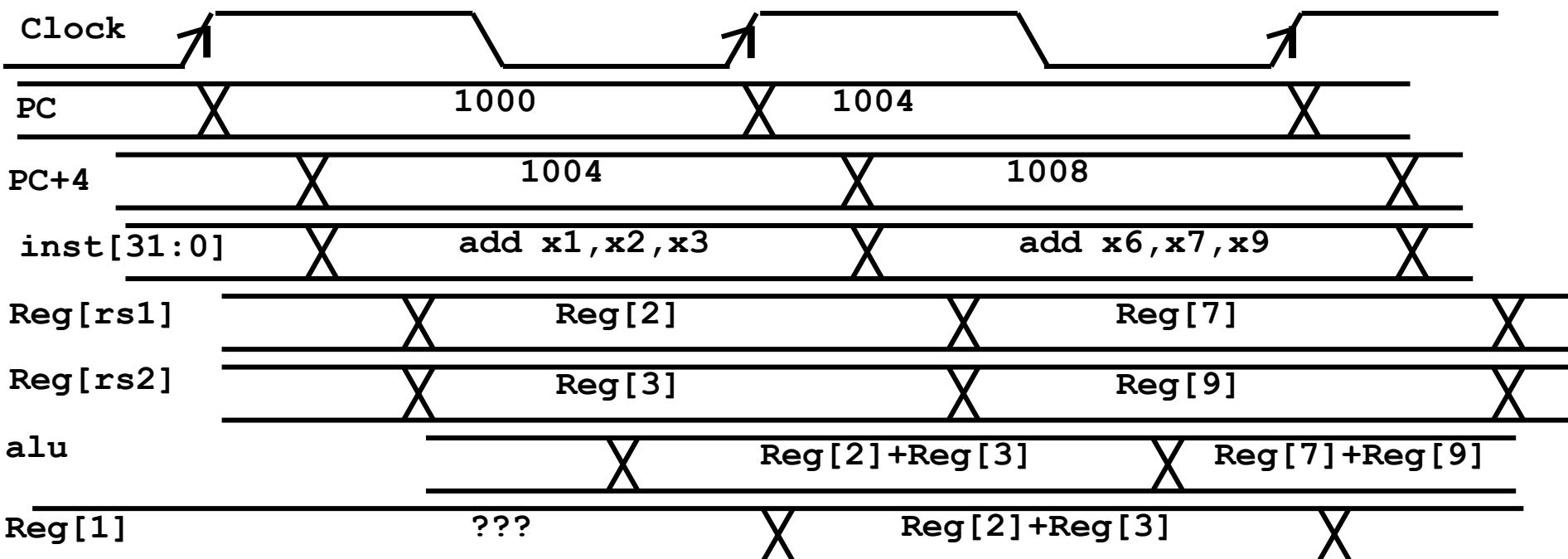
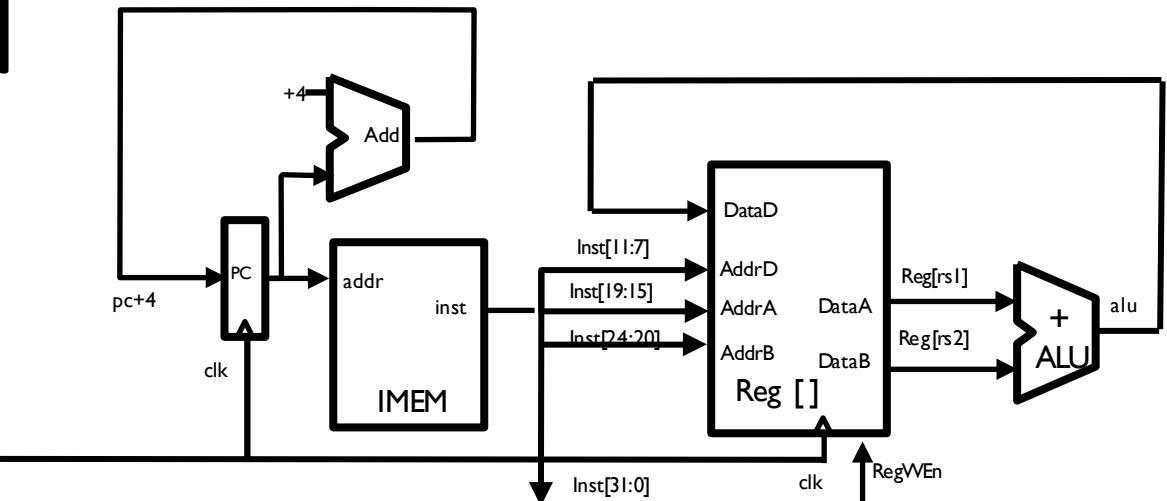
**add rd, rs1, rs2**

- Instruction makes two changes to machine's state:
  - $\text{Reg}[rd] = \text{Reg}[rs1] + \text{Reg}[rs2]$
  - $\text{PC} = \text{PC} + 4$

# Datapath for add



# Timing Diagram for add



# **Sub Datapath**

# Implementing the sub instruction

0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub

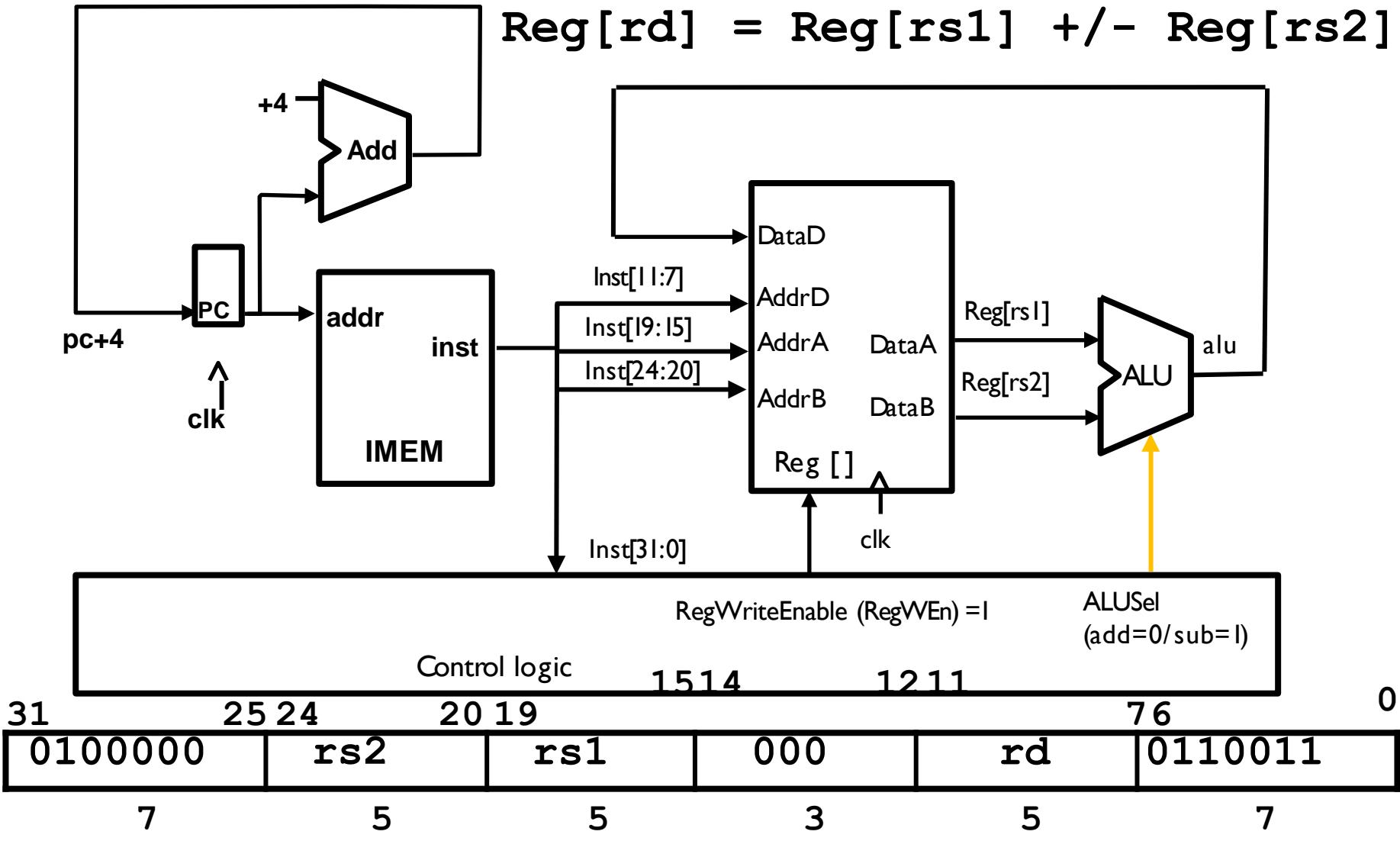
**sub rd, rs1, rs2**

- 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

$$PC = PC + 4$$

$$Reg[rd] = Reg[rs1] +/- Reg[rs2]$$



# 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

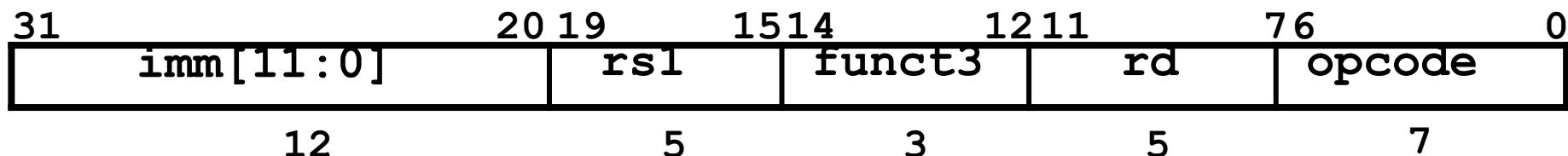
All implemented by decoding funct3 and funct7 fields  
and selecting appropriate ALU function

# Datapath With Immediates

# Implementing I-Format - addi instruction

- RISC-V Assembly Instruction:

**addi x15,x1,-50**



111111001110	00001	000	01111	0010011
--------------	-------	-----	-------	---------

imm=-50

rs1=1

add

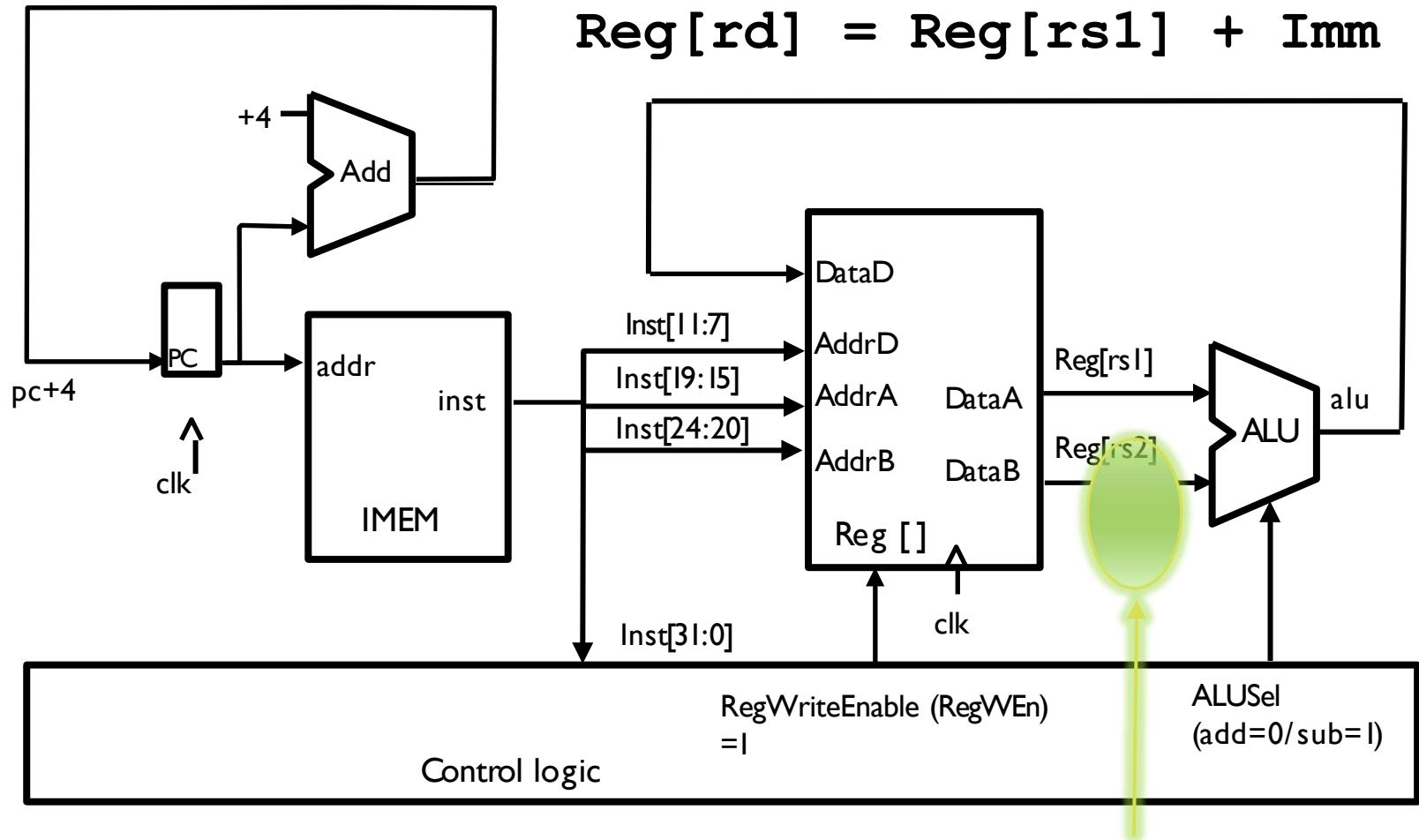
rd=15

OP-Imm

# Datapath for add/sub

$$\text{PC} = \text{PC} + 4$$

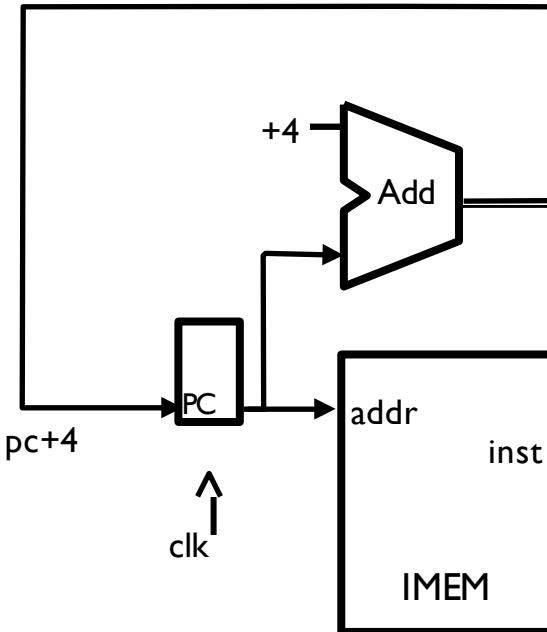
$$\text{Reg}[rd] = \text{Reg}[rs1] + \text{Imm}$$



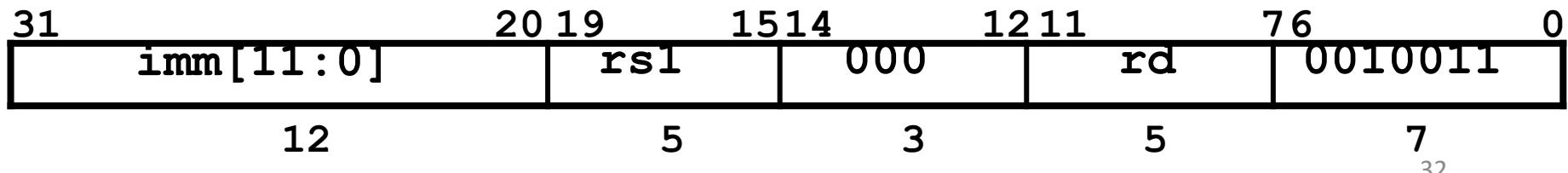
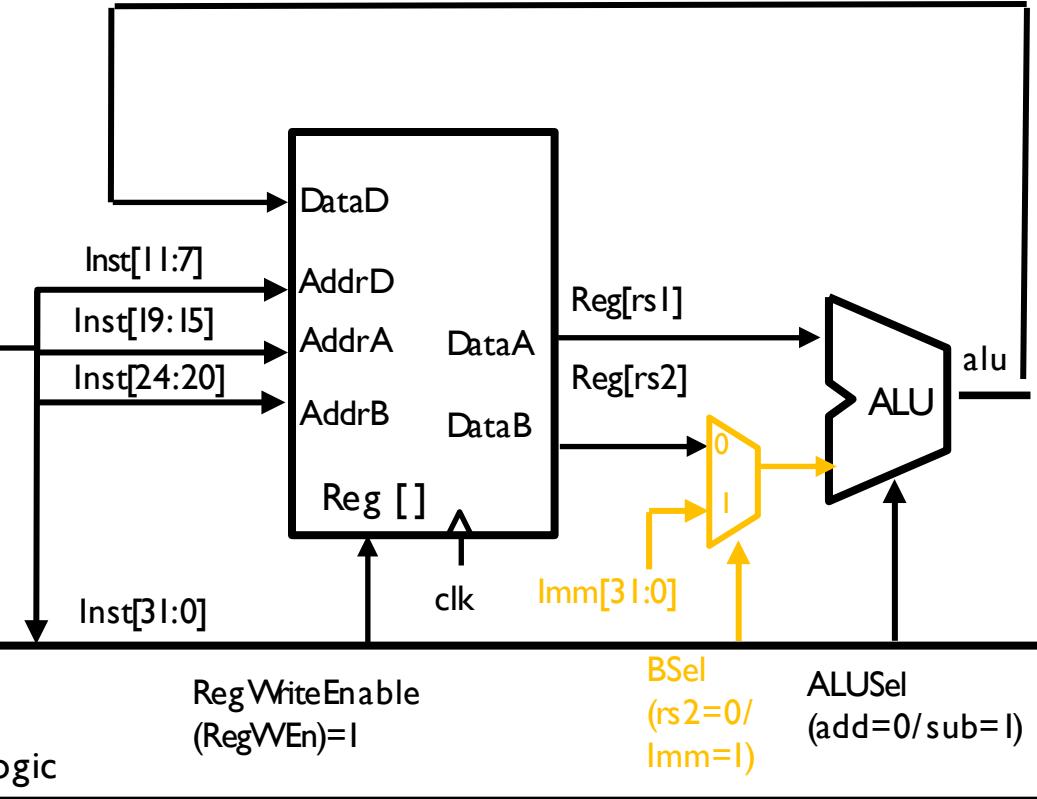
Immediate should  
be here

# Adding **addi** to Datapath

$$\text{PC} = \text{PC} + 4$$

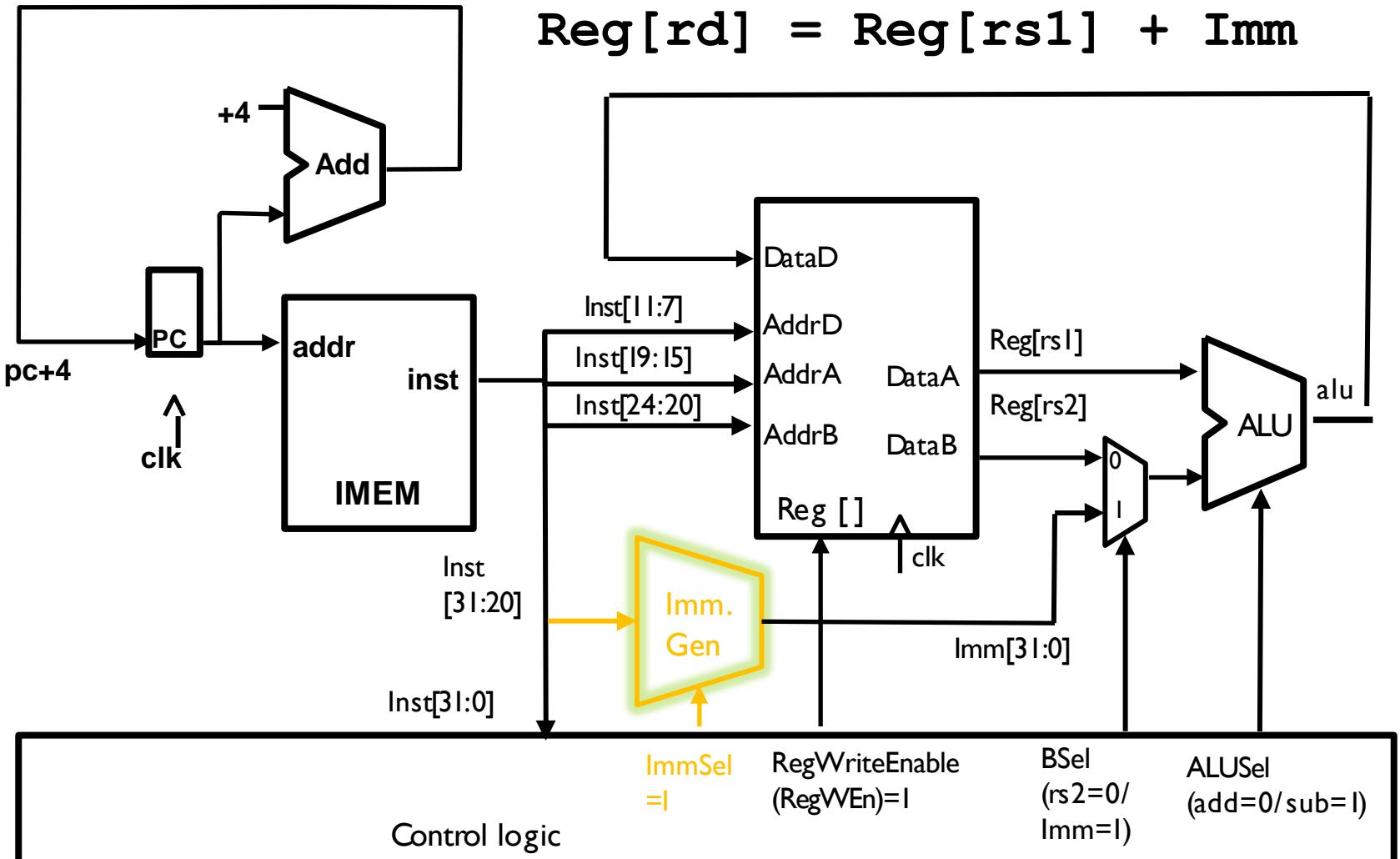


$$\text{Reg}[rd] = \text{Reg}[rs1] + \text{Imm}$$



# Adding **addi** to Datapath

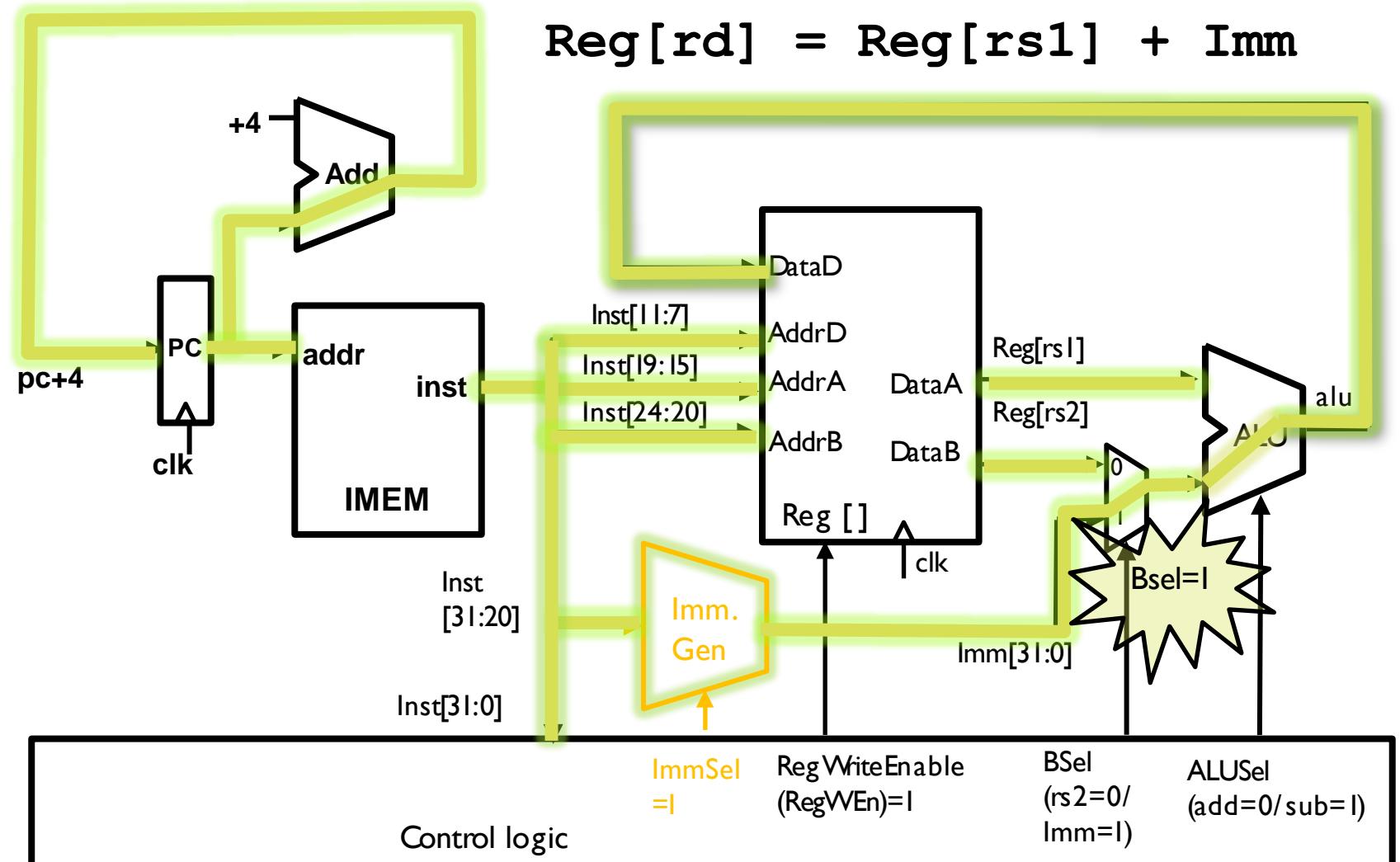
$$\text{PC} = \text{PC} + 4$$



# Adding **addi** to Datapath

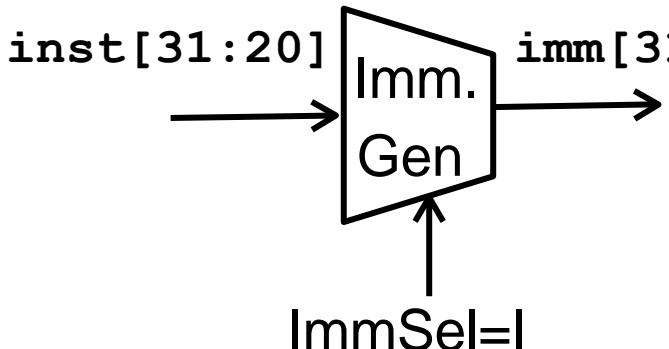
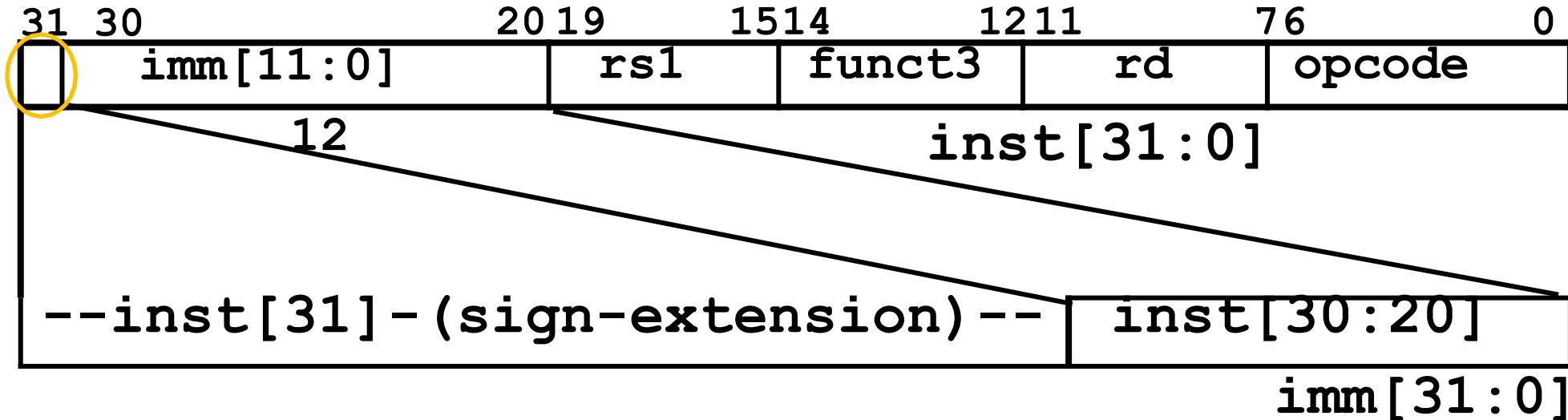
$$PC = PC + 4$$

$$Reg[rd] = Reg[rs1] + Imm$$



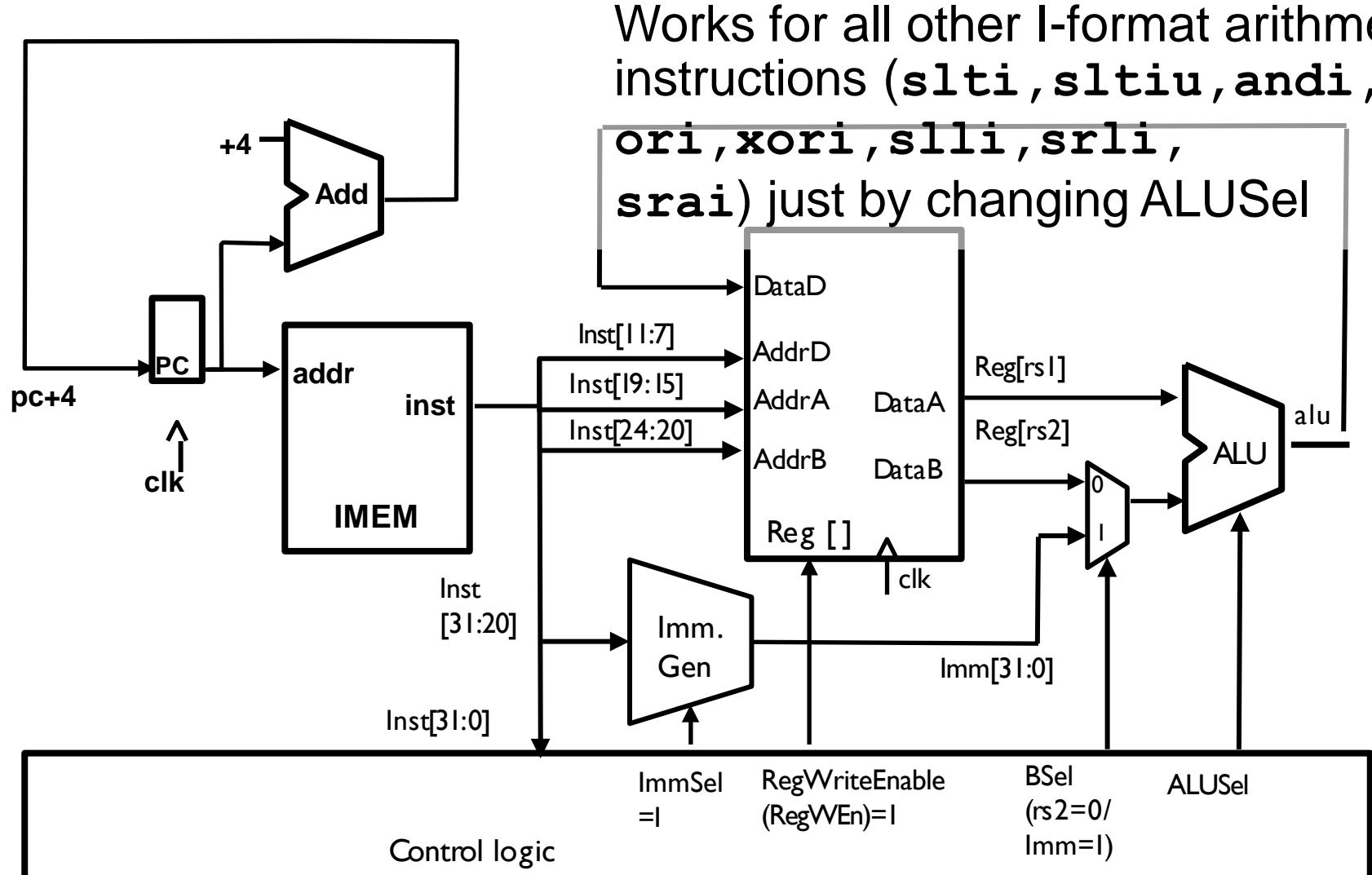
# I-Format Immediates

-inst[31]-



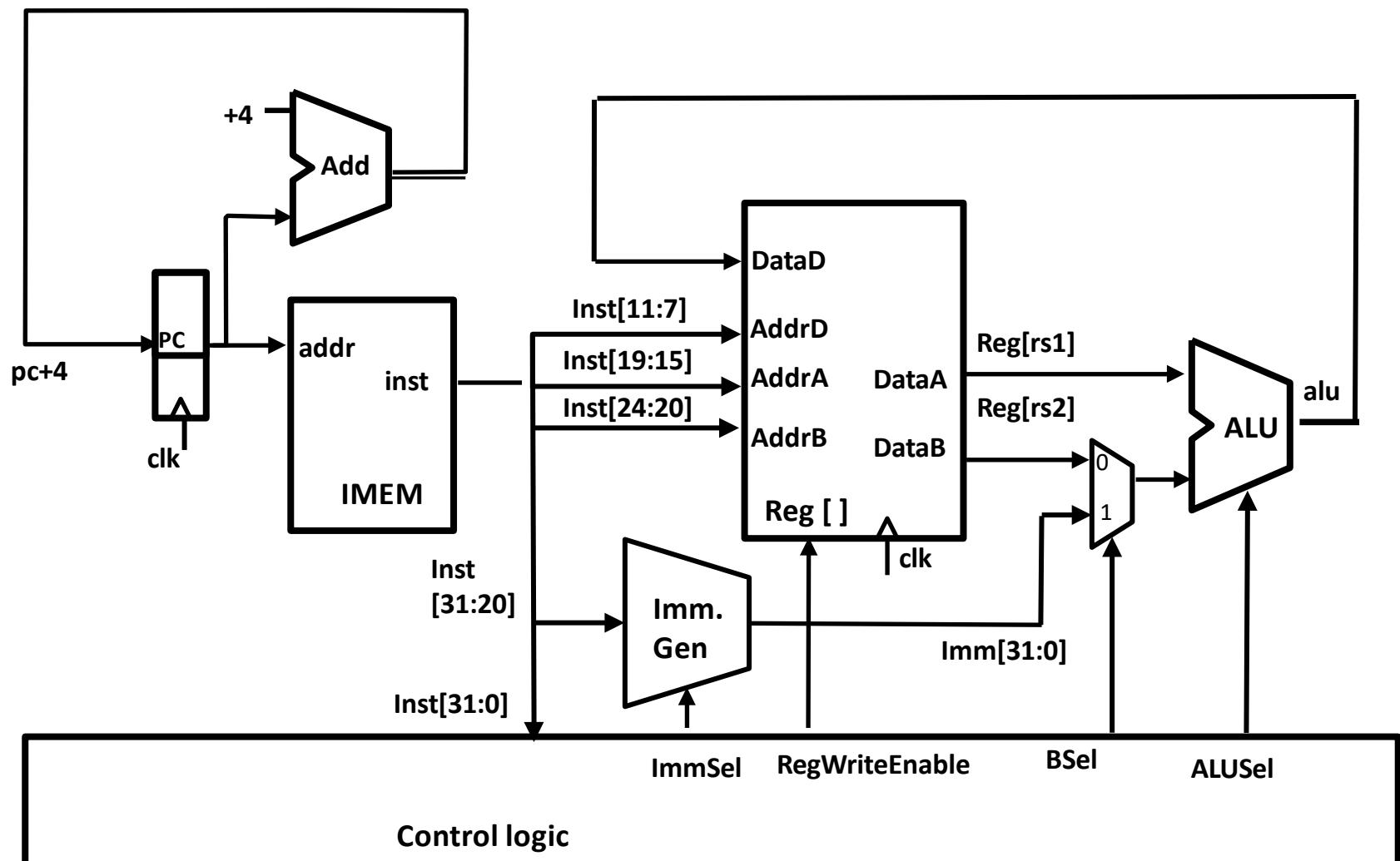
- High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
- Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])

# Adding **addi** to Datapath



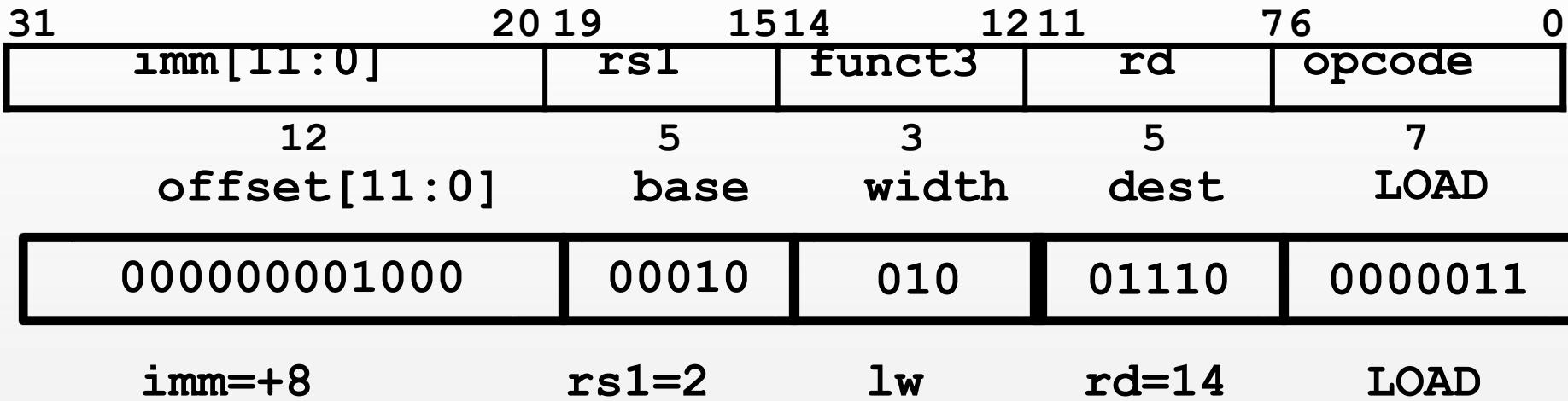
# **Supporting Loads**

# R+I Arithmetic/Logic Datapath



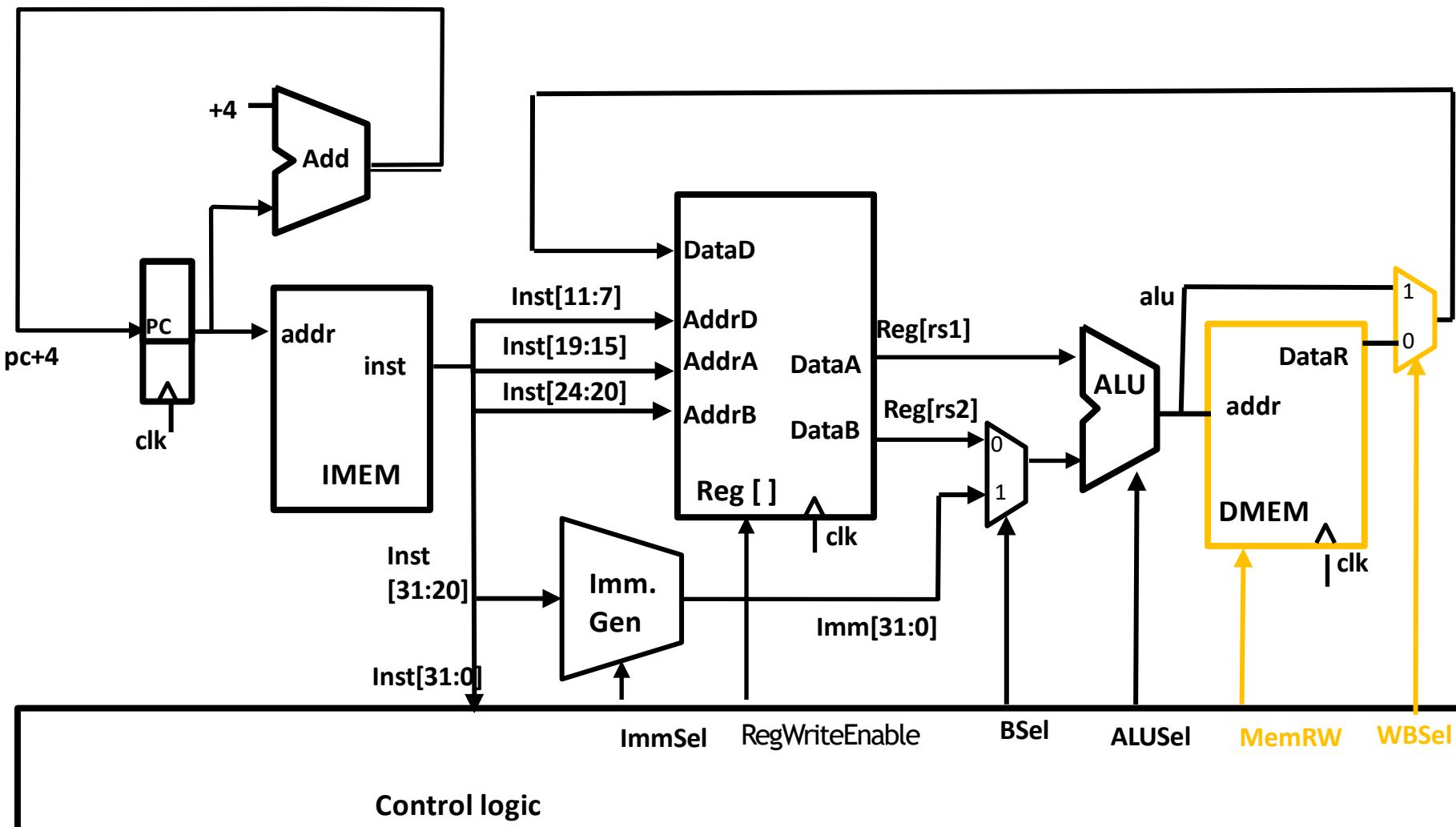
# Add **lw**

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

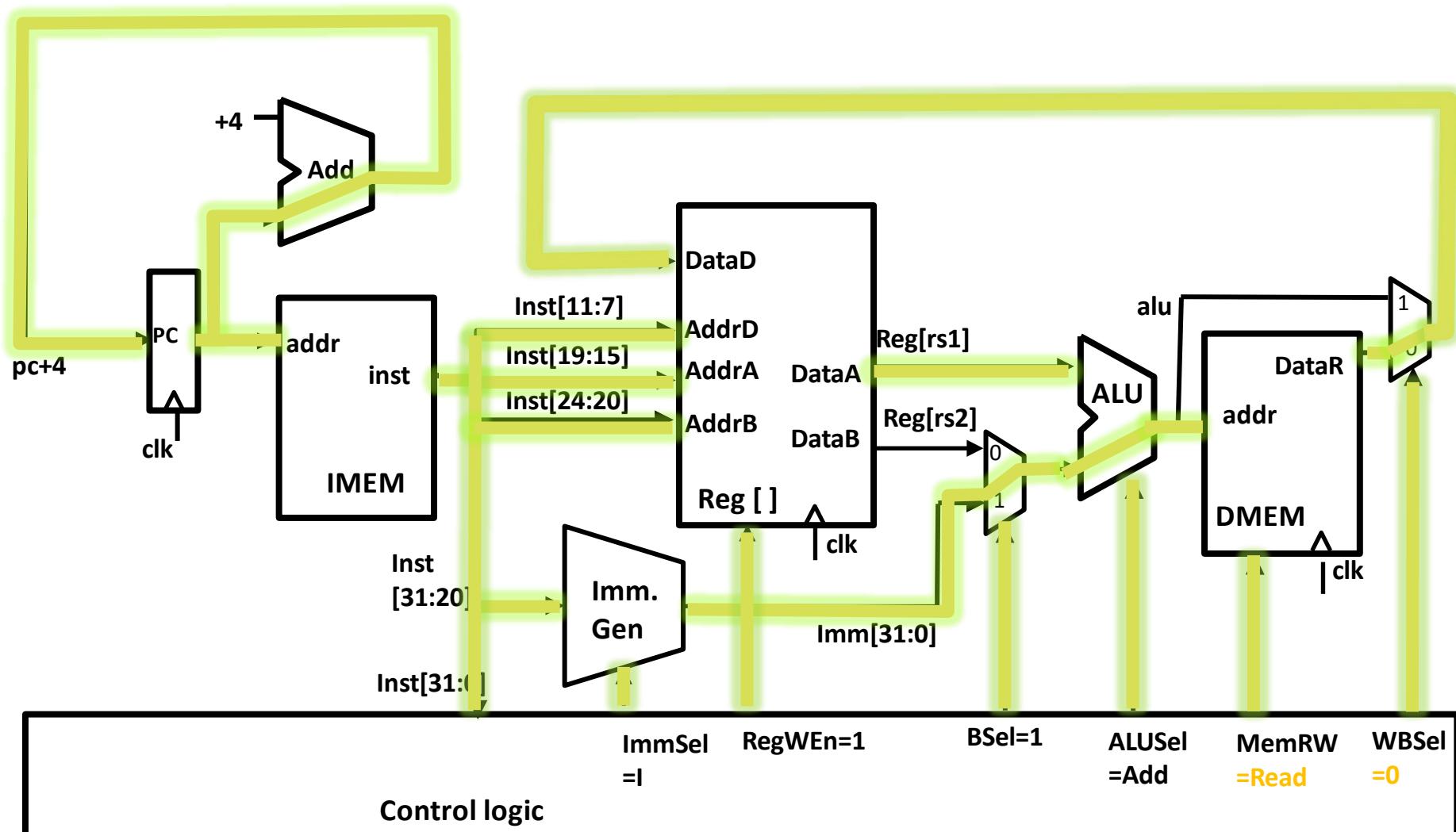


- The 12-bit signed immediate is added to the base address in register **rs1** to form the memory address
  - This is very similar to the add-immediate operation but used to create address not to create final result
- The value loaded from memory is stored in register **rd**

# R+I Arithmetic/Logic Datapath



# R+I Arithmetic/Logic 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

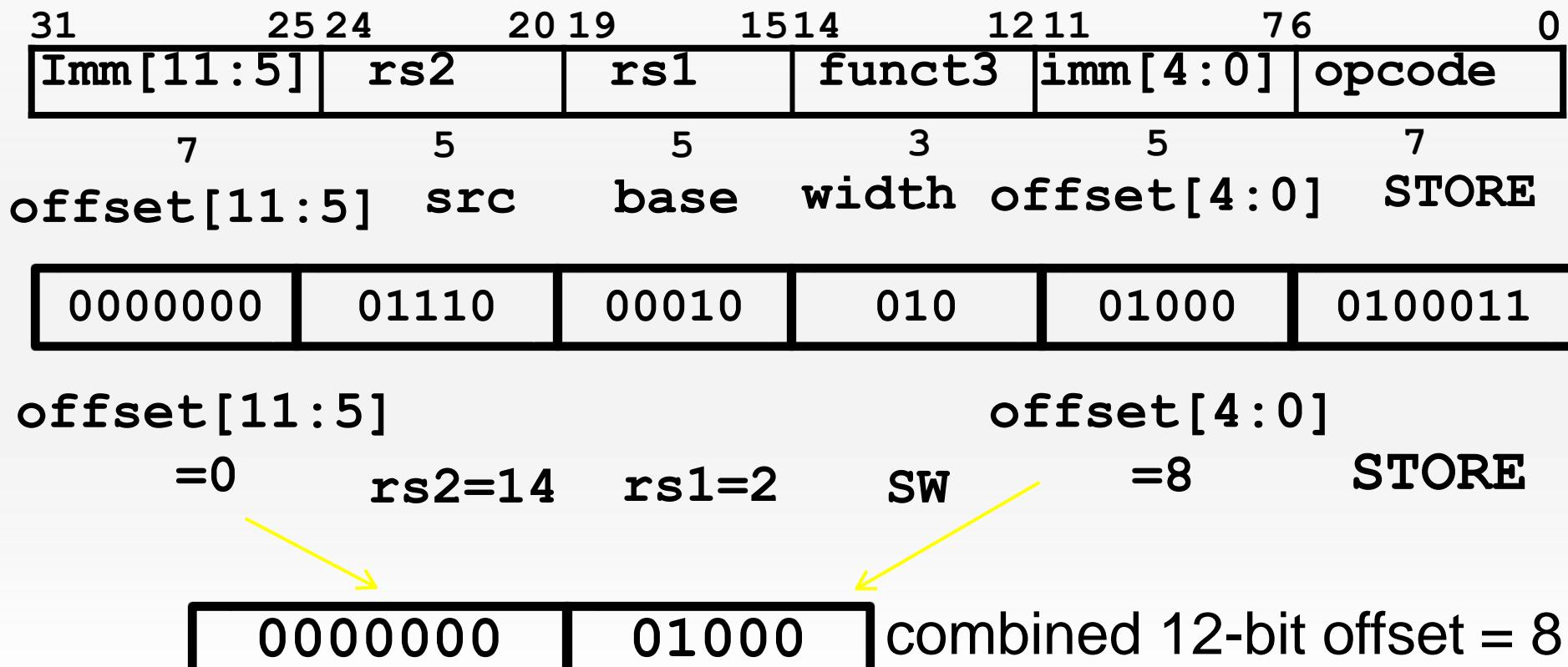
- Supporting the narrower loads requires additional logic to extract the correct byte/halfword from the value loaded from memory, and sign- or zero-extend the result to 32 bits before writing back to register file.
  - It is just a mux + a few gates

# **Datapath for Stores**

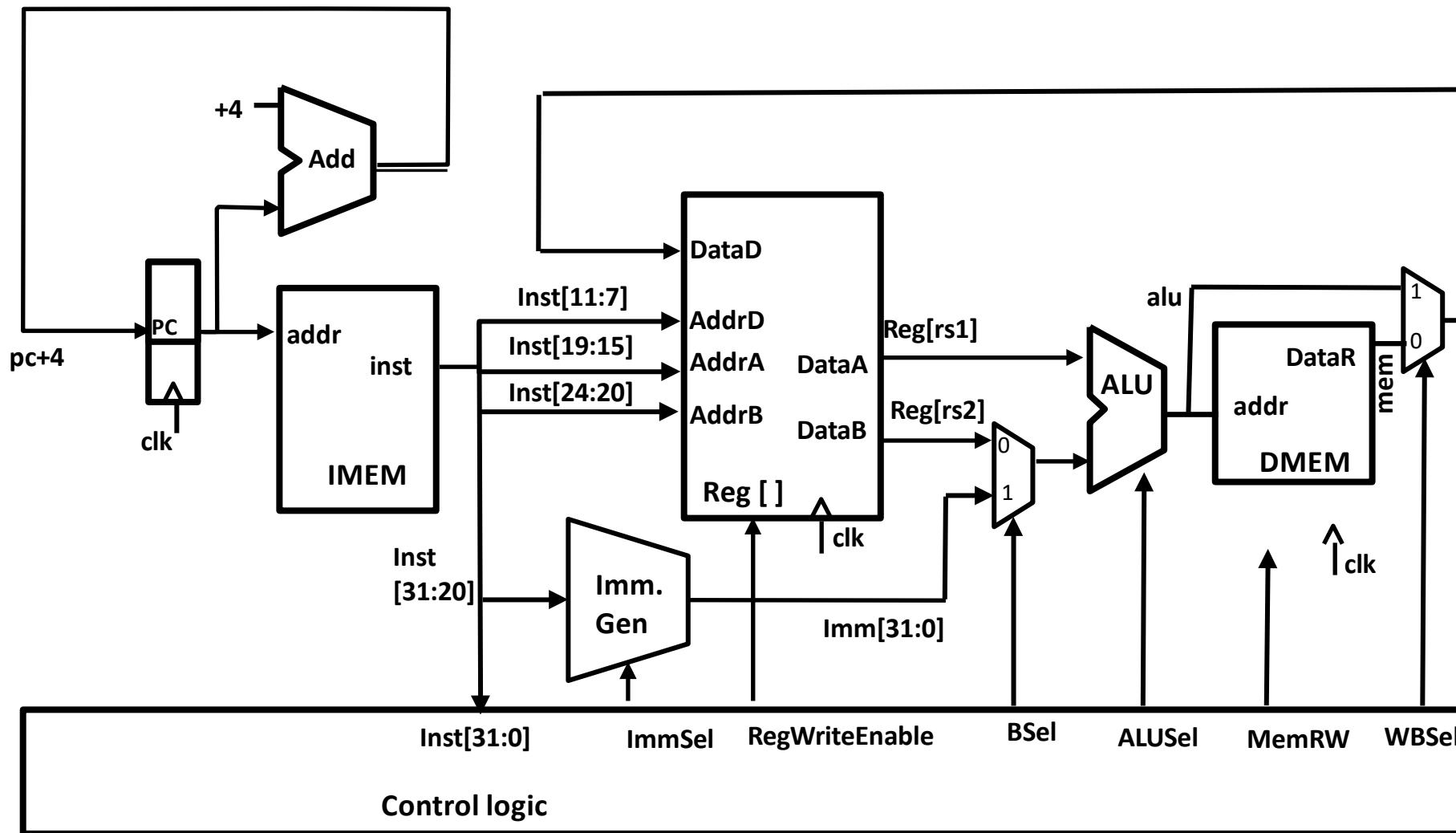
# Adding **sw** instruction

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

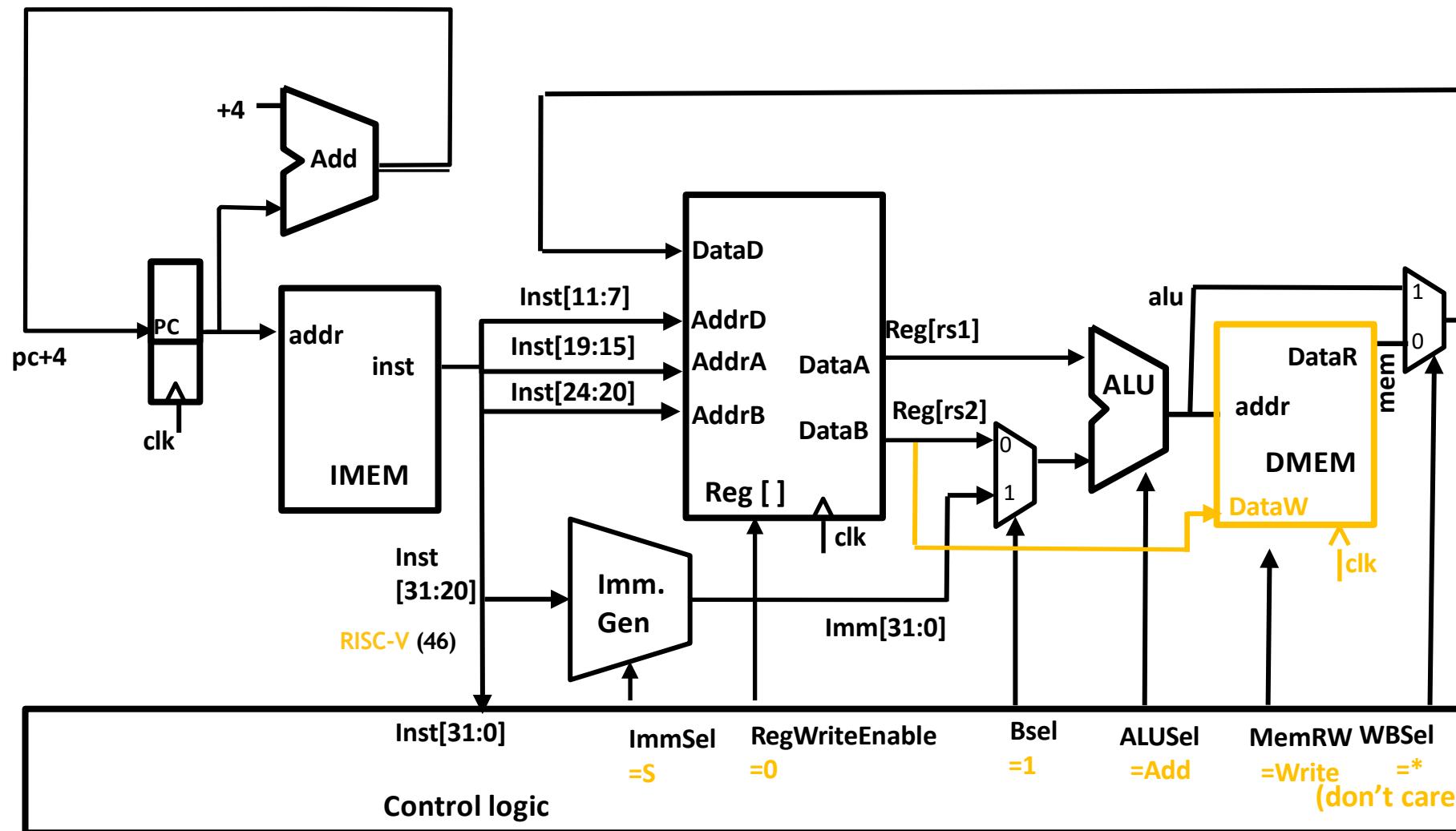
**sw x14 , 8(x2)**



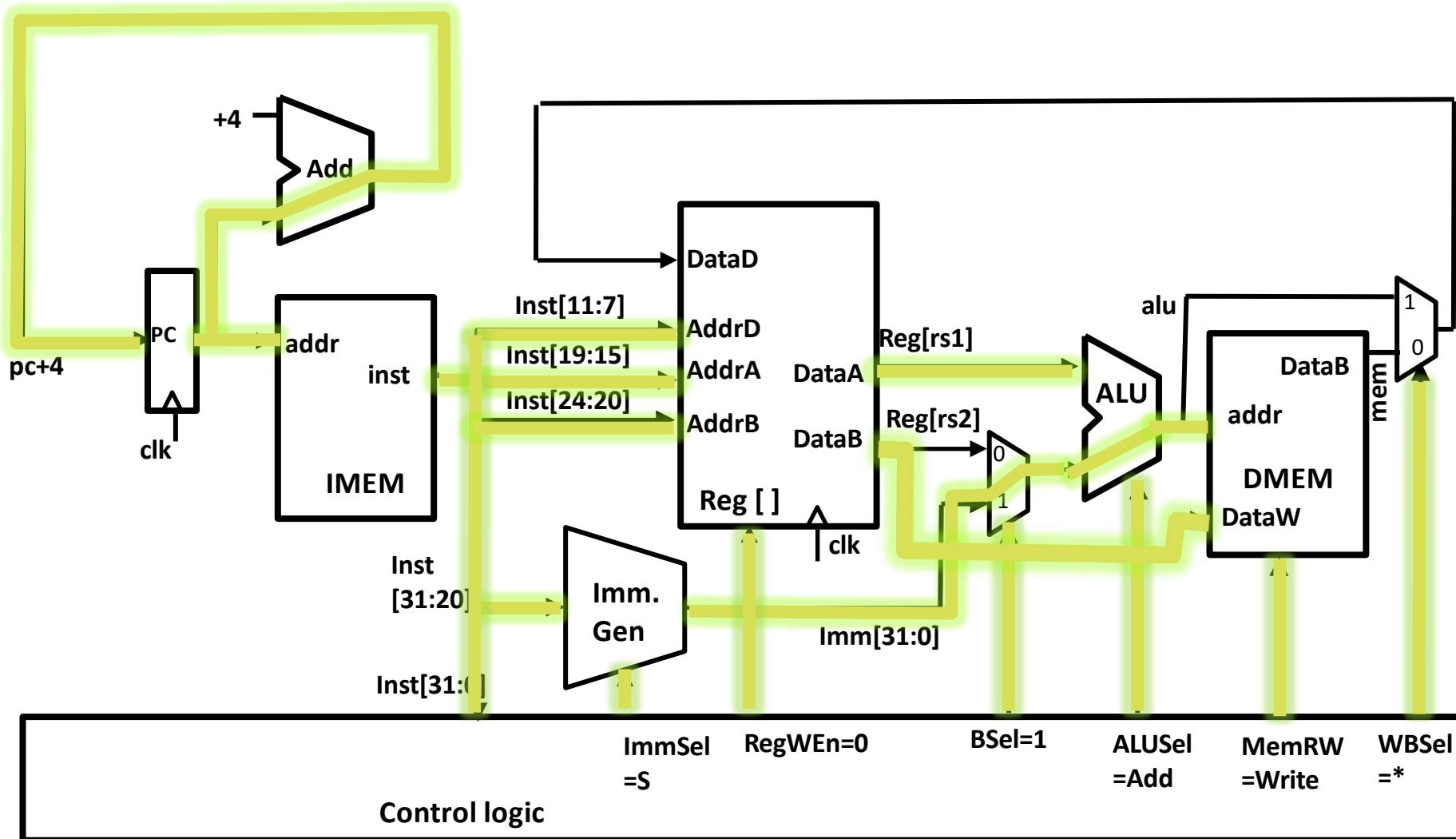
# Datapath with $lw$



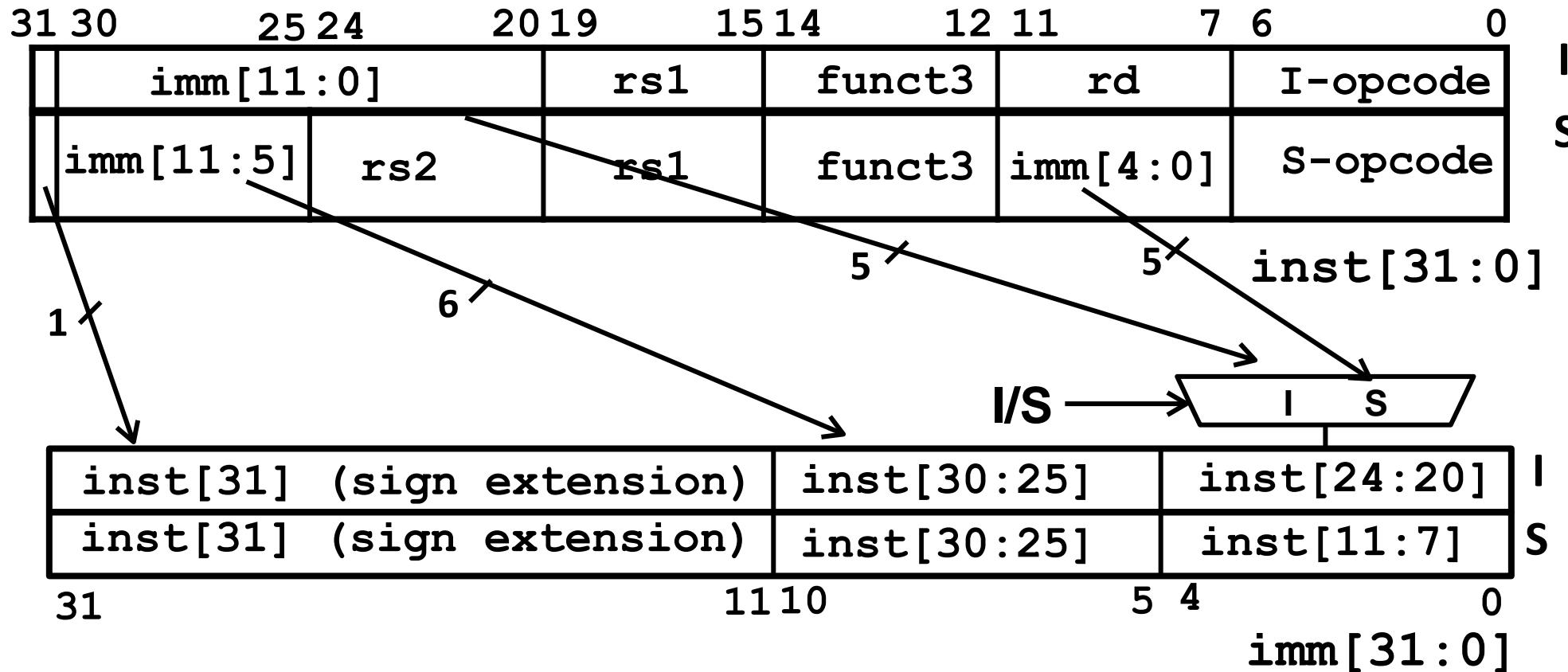
# Adding **sw** to Datapath



# Adding **sw** to Datapath



# 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

# All RV32 Store Instructions

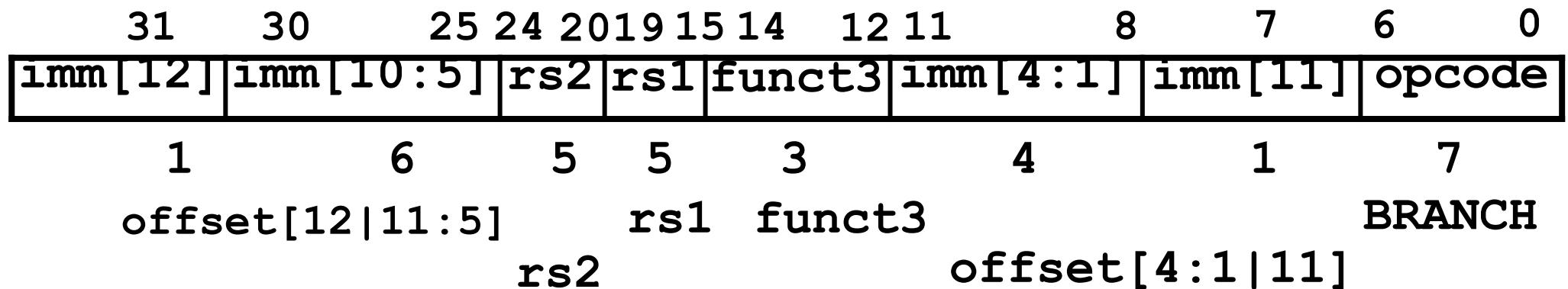
- Store byte writes the low byte to memory
- Store halfword writes the lower two bytes to memory

Imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	<b>sb</b>
Imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	<b>sh</b>
Imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	<b>sw</b>

**width**

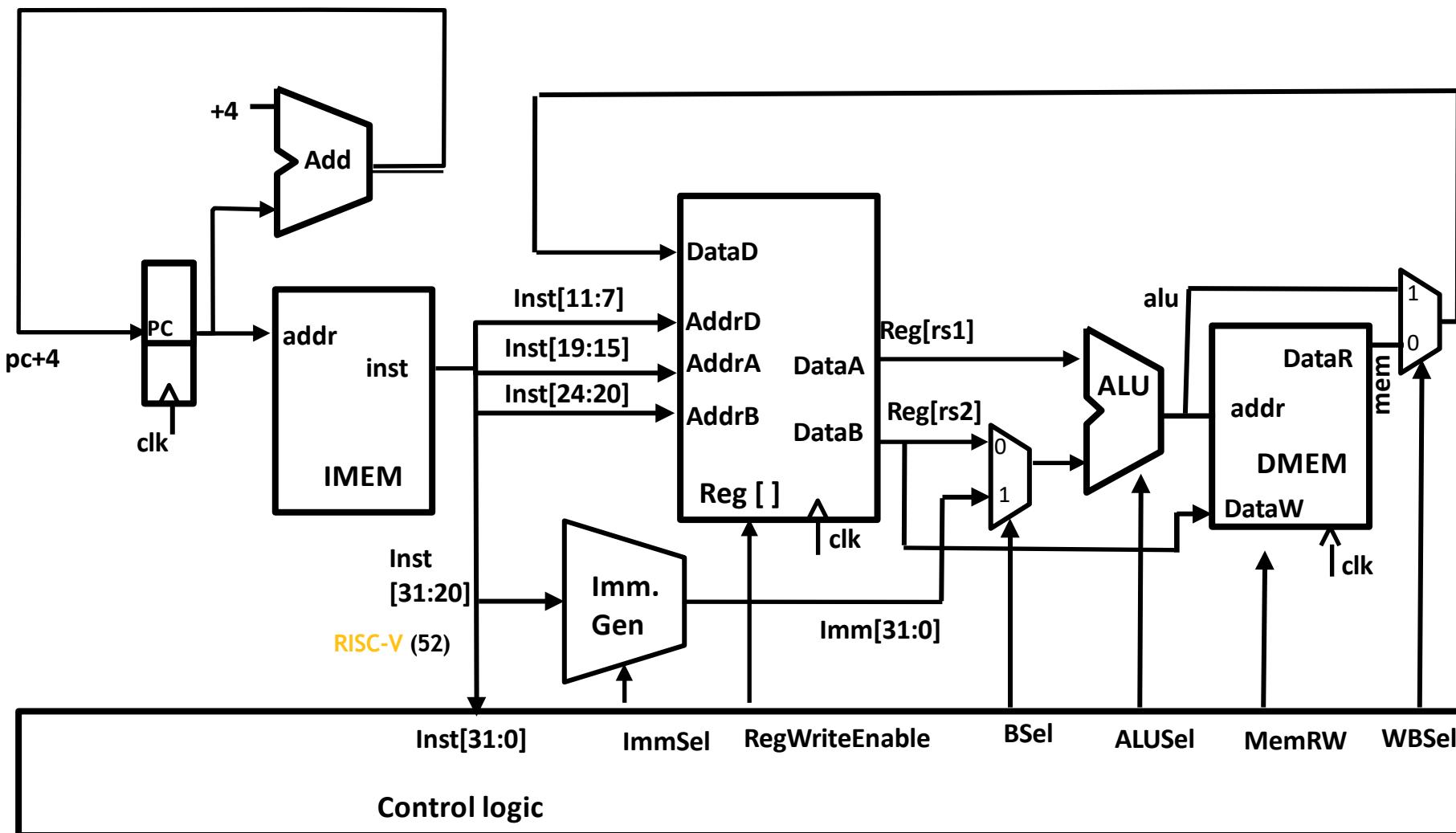
# **Implementing Branches**

# RISC-V B-Format for Branches



- B-format is mostly same as S-Format, with two register sources (**rs1/rs2**) and a 12-bit immediate **imm [12:1]**
- 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



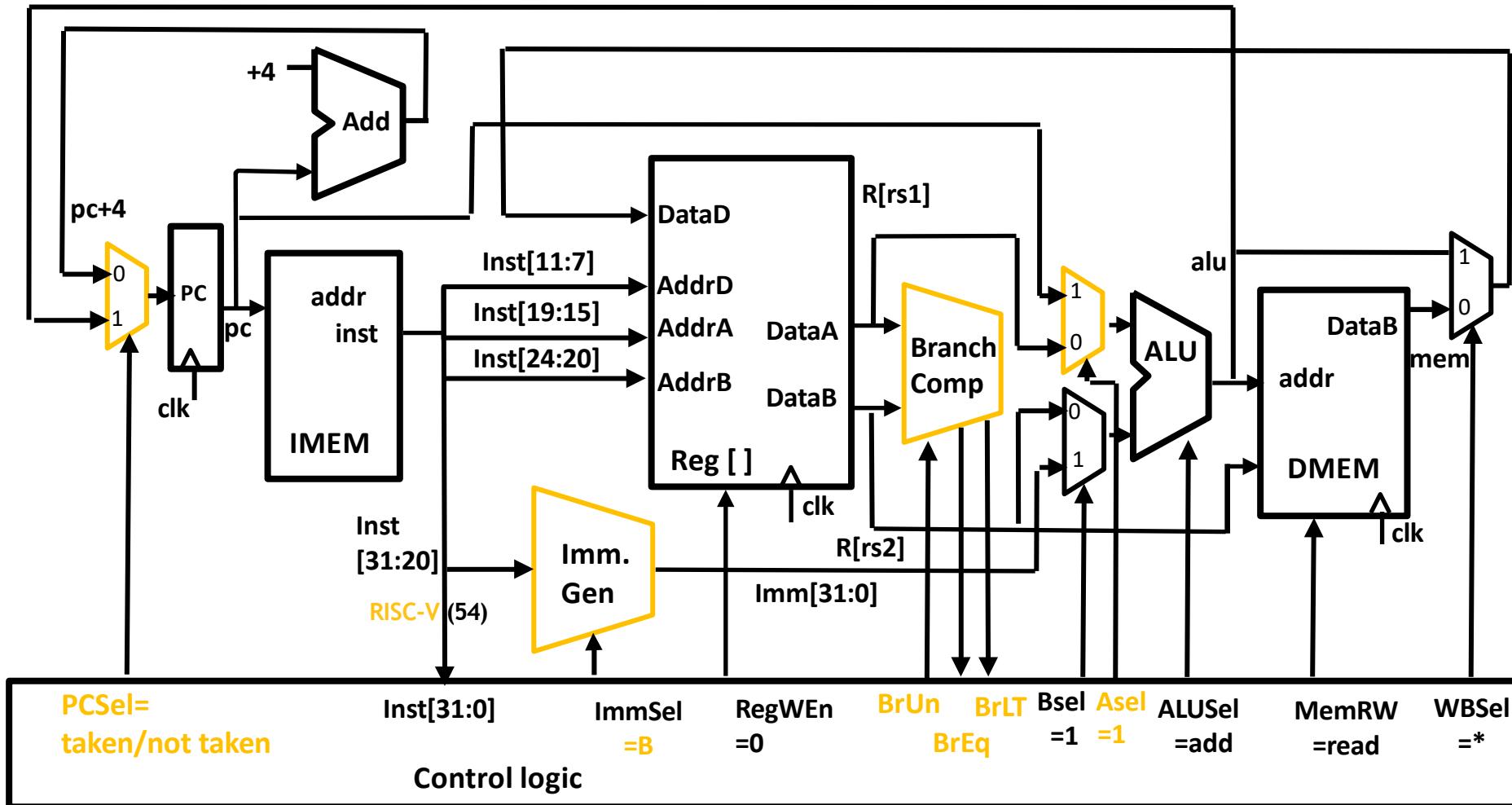
# Add Branches

- Different change to the state:

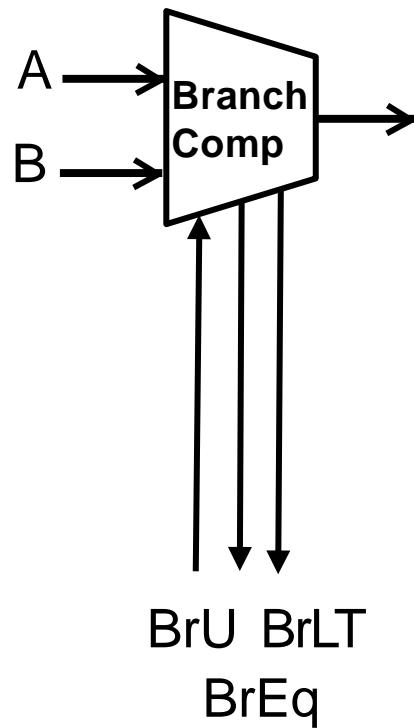
$$\text{PC} = \begin{cases} \text{PC} + 4, & \text{branch not taken} \\ \text{PC} + \text{immediate}, & \text{branch taken} \end{cases}$$

- 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



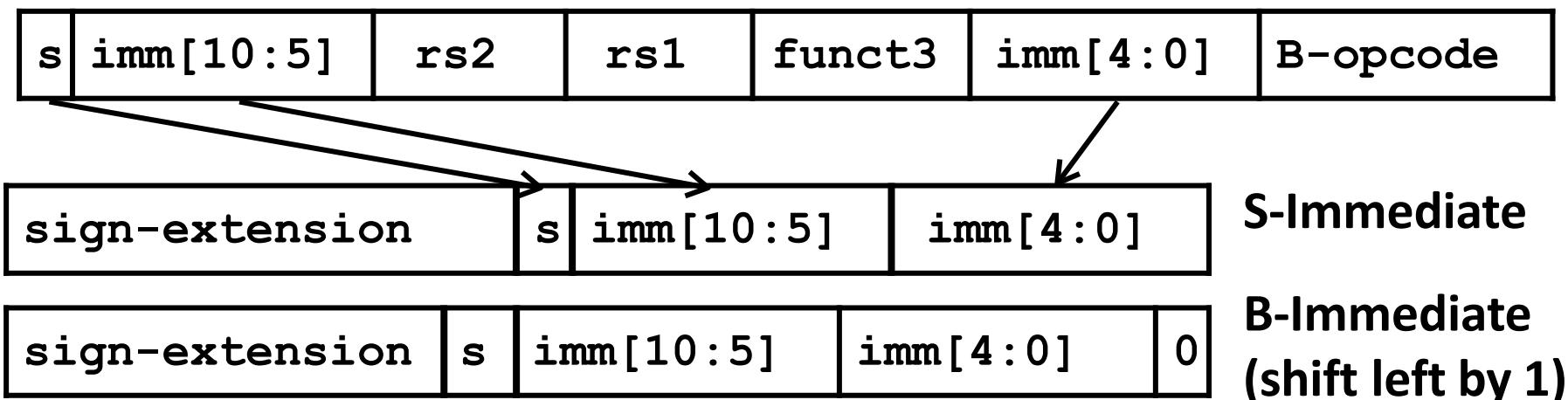
- **BrEq** = 1, if  $A=B$
- **BrLT** = 1, if  $A < B$
- **BrUn** = 1 selects unsigned comparison for **BrLT**, 0=signed

**BGE** branch:  $A \geq B$ , if  $\overline{A < B}$

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

# Branch Immediates (In Other ISAs)

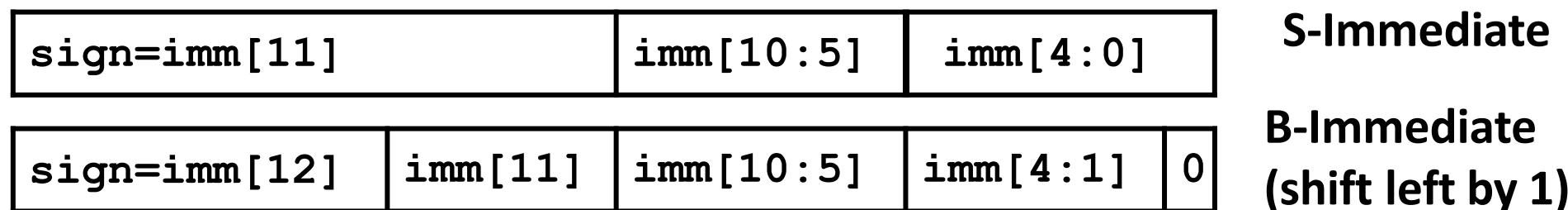
- 12-bit immediate encodes PC-relative offset of -4096 to +4094 bytes in multiples of 2 bytes
- Standard approach: Treat immediate as in range -2048..+2047, then shift left by 1 bit to multiply by 2 for branches



Each instruction immediate bit can appear in one of two places in output immediate value – so need one 2-way mux per bit

# Branch Immediates (In Other ISAs)

- 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**



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

# RISC-V Immediate Encoding

## Instruction encodings, $\text{inst}[31:0]$

31	30	25 24	20 19	15 14	12 11	8 7 6	0	
		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

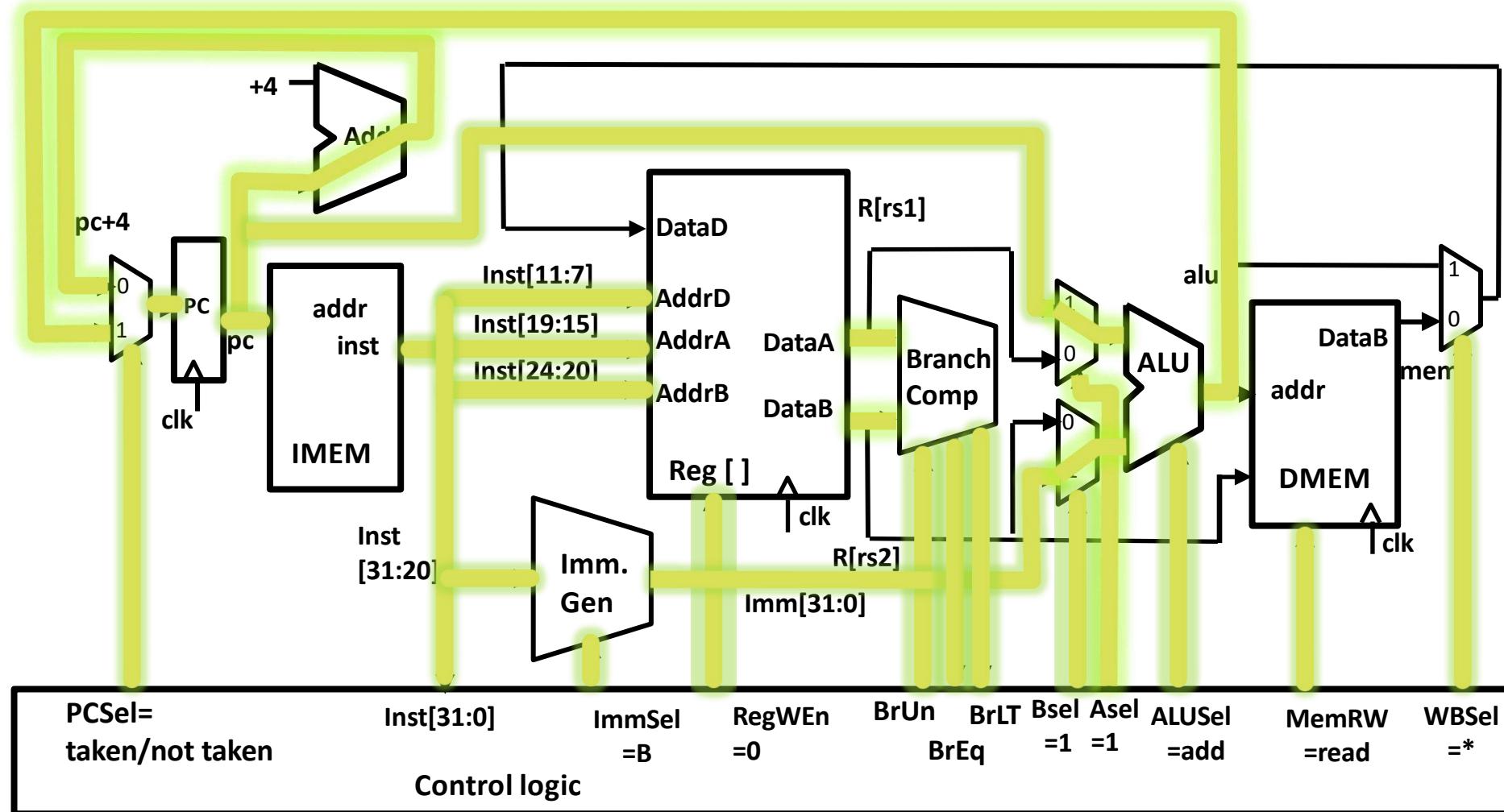
## 32-bit immediates produced, $\text{imm}[31:0]$

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

Upper bits sign-extended from  $\text{inst}[31]$  always

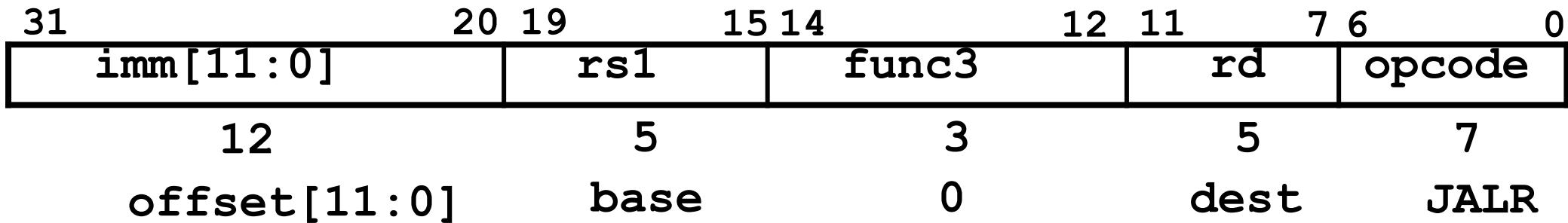
Only bit 7 of instruction changes role in immediate between S and B

# Lighting Up Branch Path



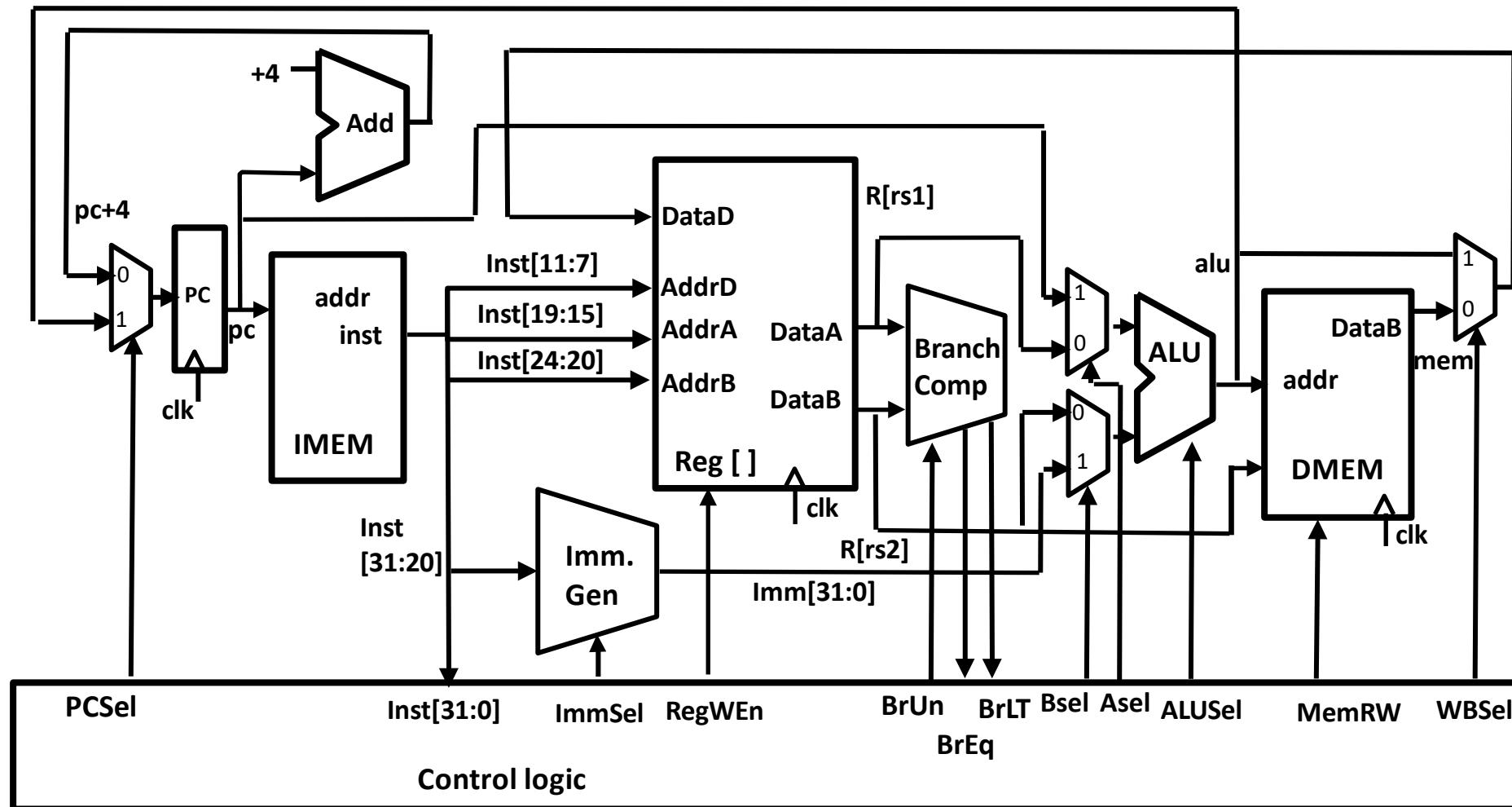
# **Adding JALR to Datapath**

# Let's Add JALR (I-Format)

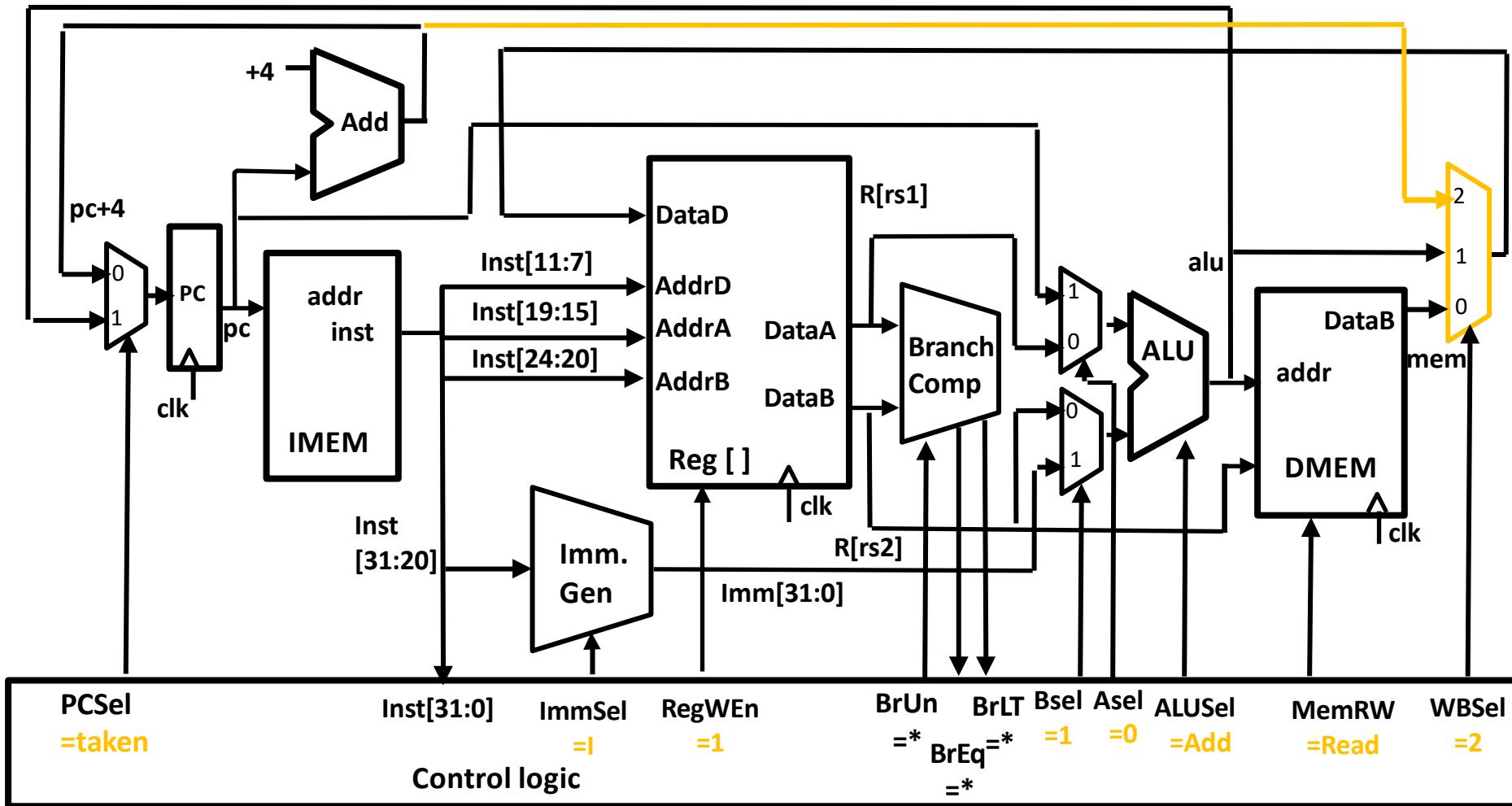


- **JALR rd, rs, immediate**
- Two changes to the state
  - Writes PC+4 to **rd** (return address)
  - Sets PC = **rs1 + immediate**
  - Uses same immediates as arithmetic and loads
    - **no** multiplication by 2 bytes
  - LSB is ignored

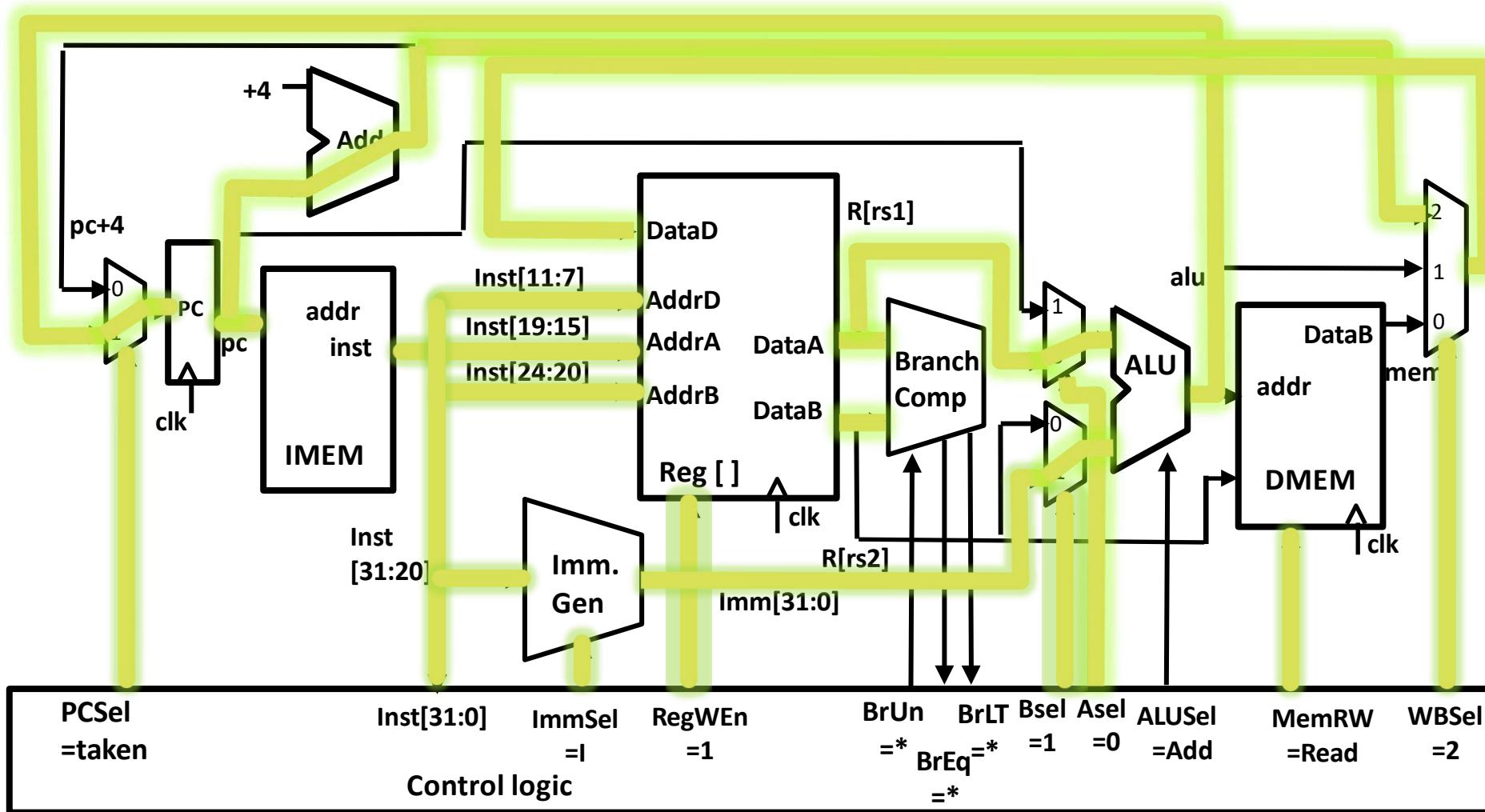
# Datapath So Far, With Branches



# Datapath With JALR

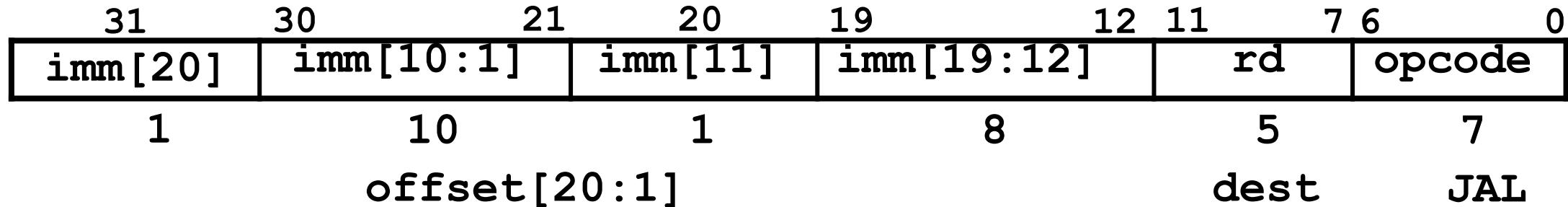


# Datapath With JALR



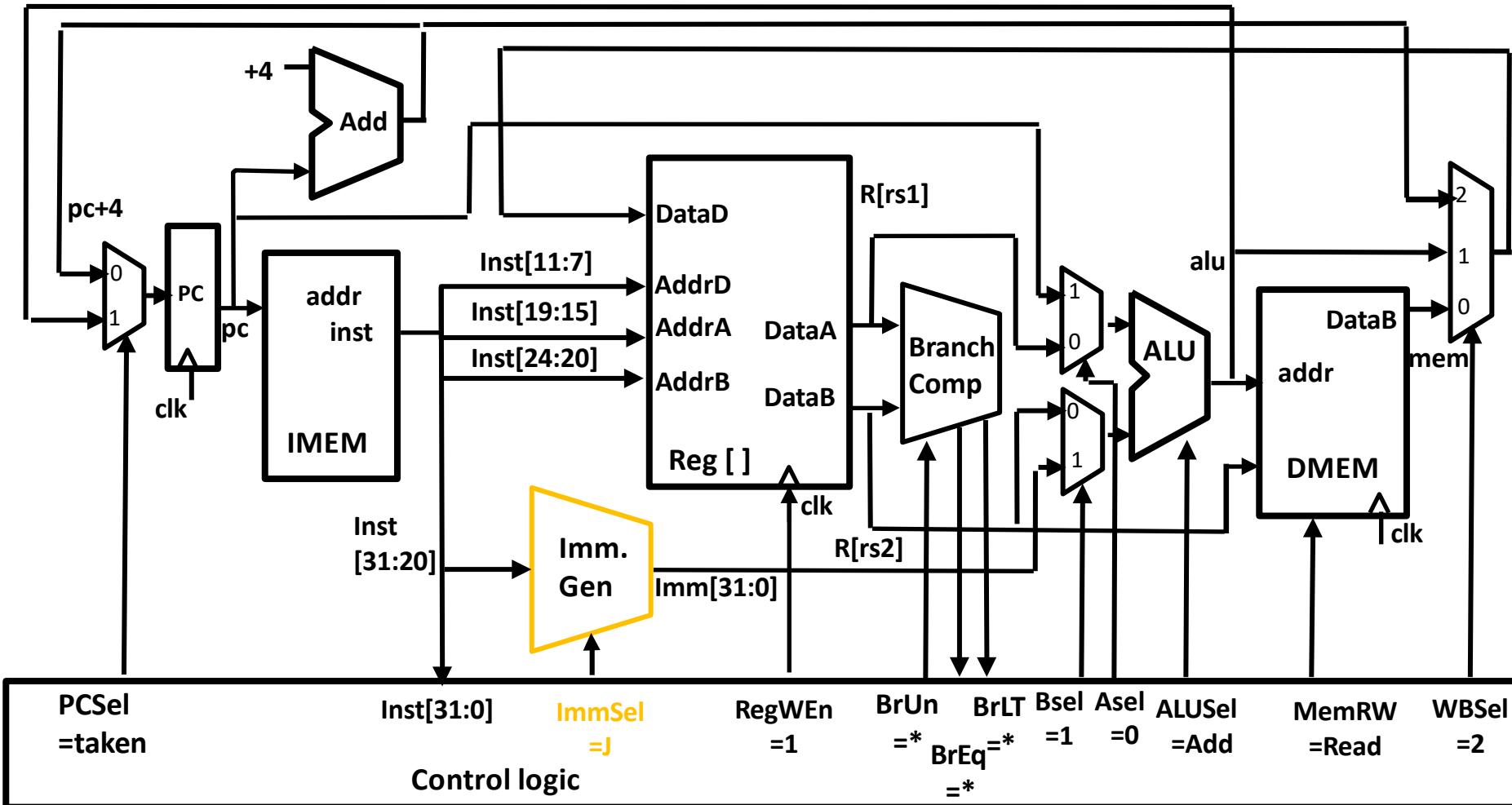
# Adding JAI

# JFormat for Jump Instructions

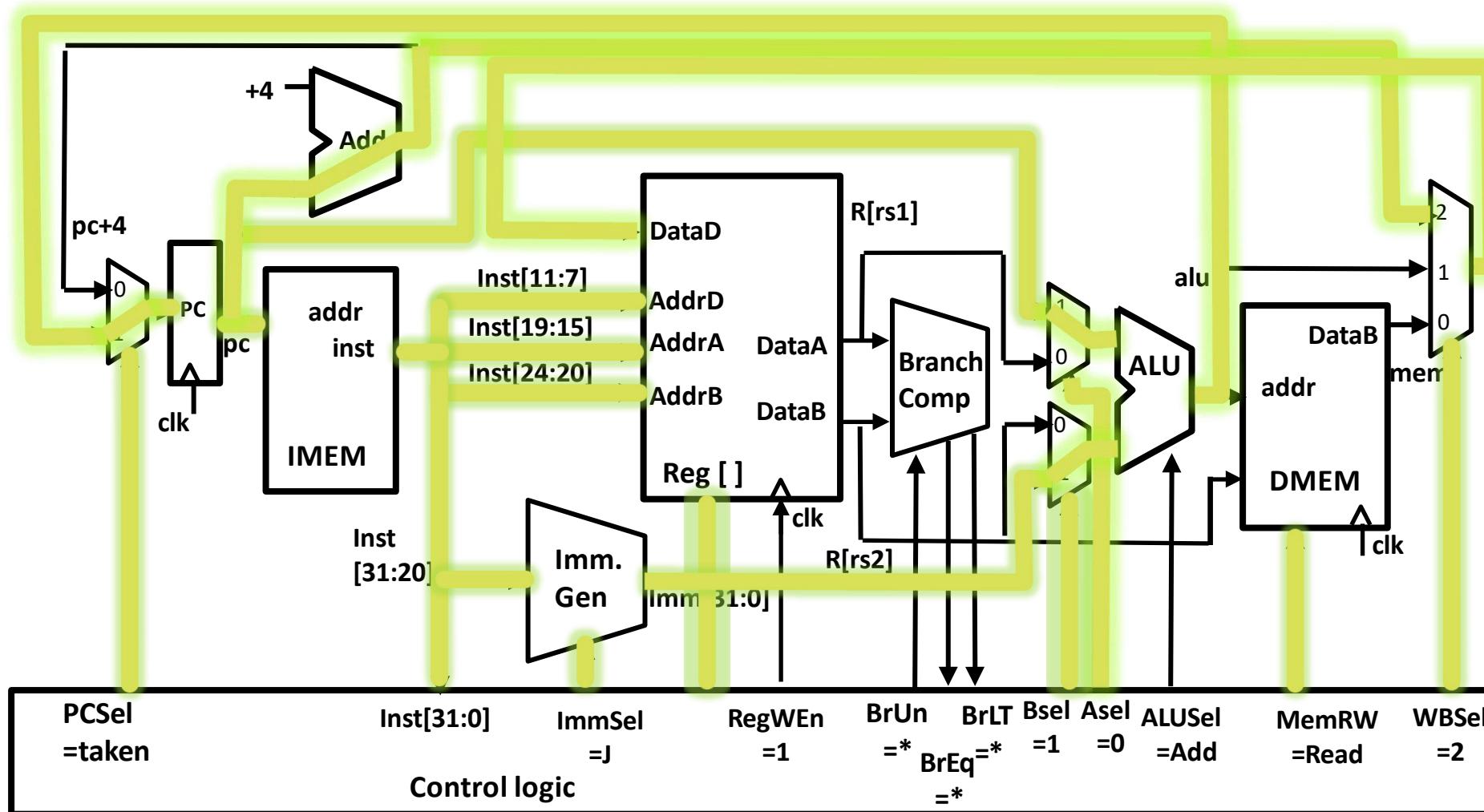


- Two changes to the state
  - `jal` saves PC+4 in register **rd** (the return address)
  - Set PC = PC + offset (PC-relative jump)
- Target somewhere within  $\pm 2^{19}$  locations, 2 bytes apart
  - $\pm 2^{18}$  32-bit instructions
- Immediate encoding optimized similarly to branch instruction to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

# Datapath with JAL

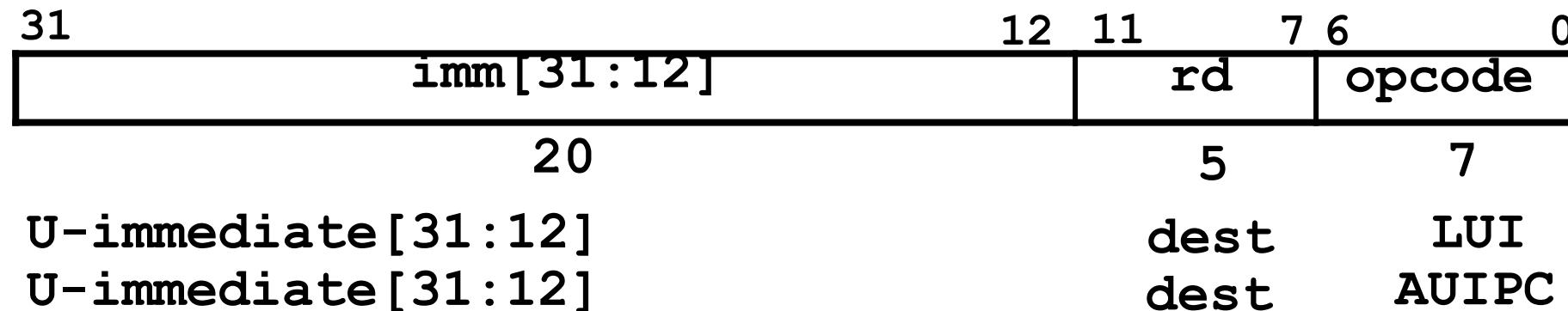


# Light Up JAL Path



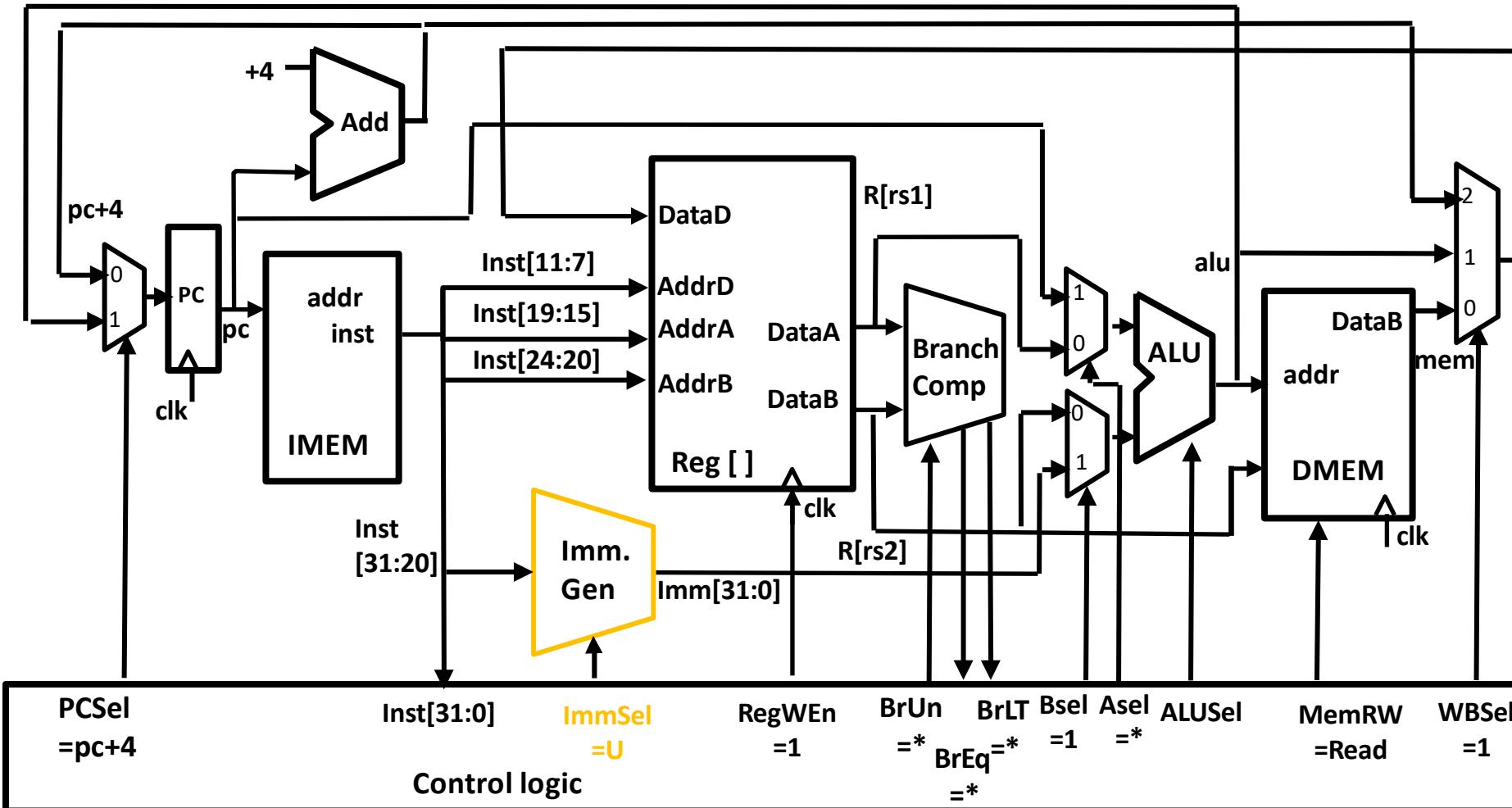
# **Adding U-Types**

# U-Format for “Upper Immediate” Instructions

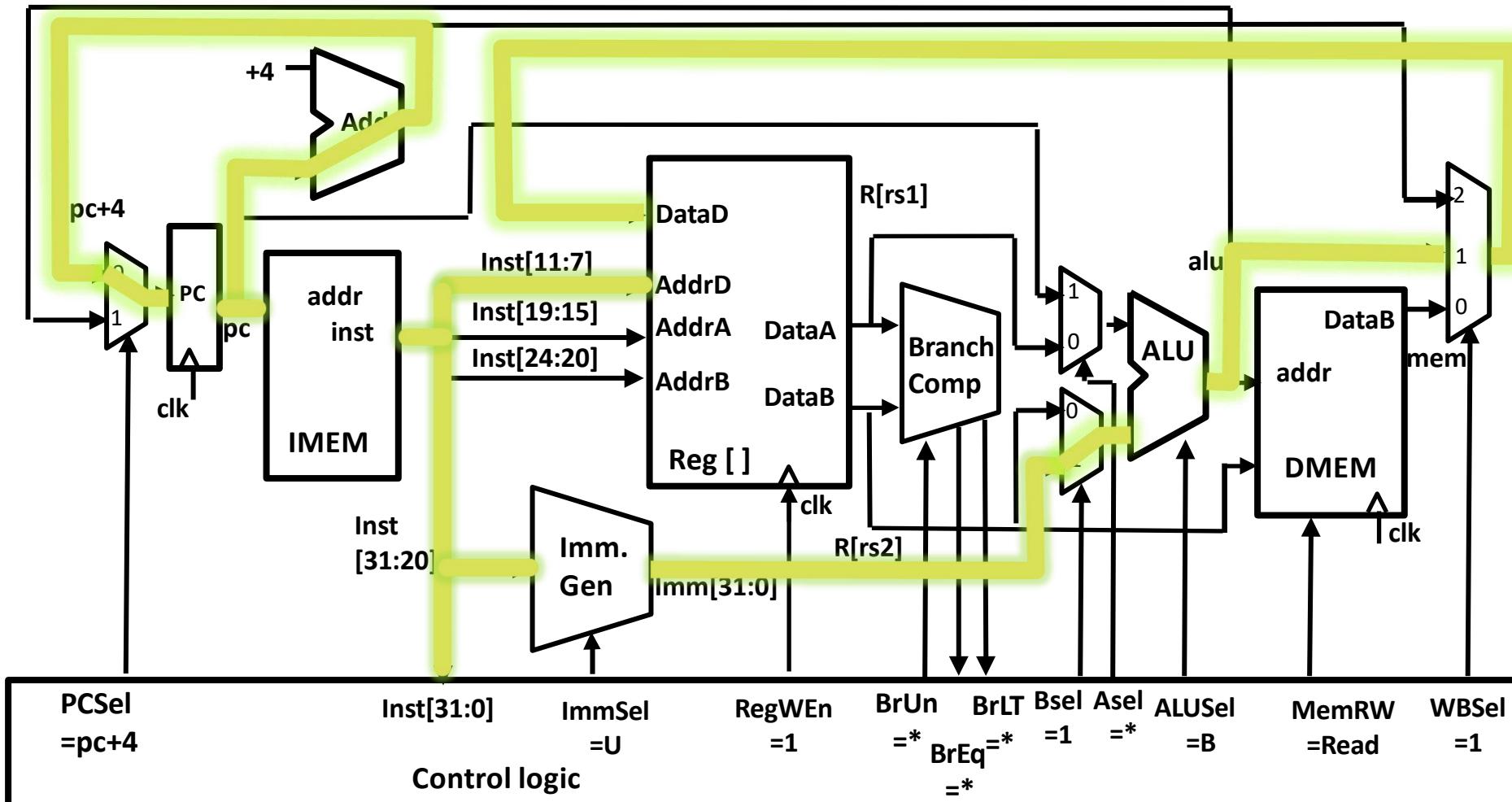


- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, **rd**
- Used for two instructions
  - **lui** – Load Upper Immediate
  - **auipc** – Add Upper Immediate to PC

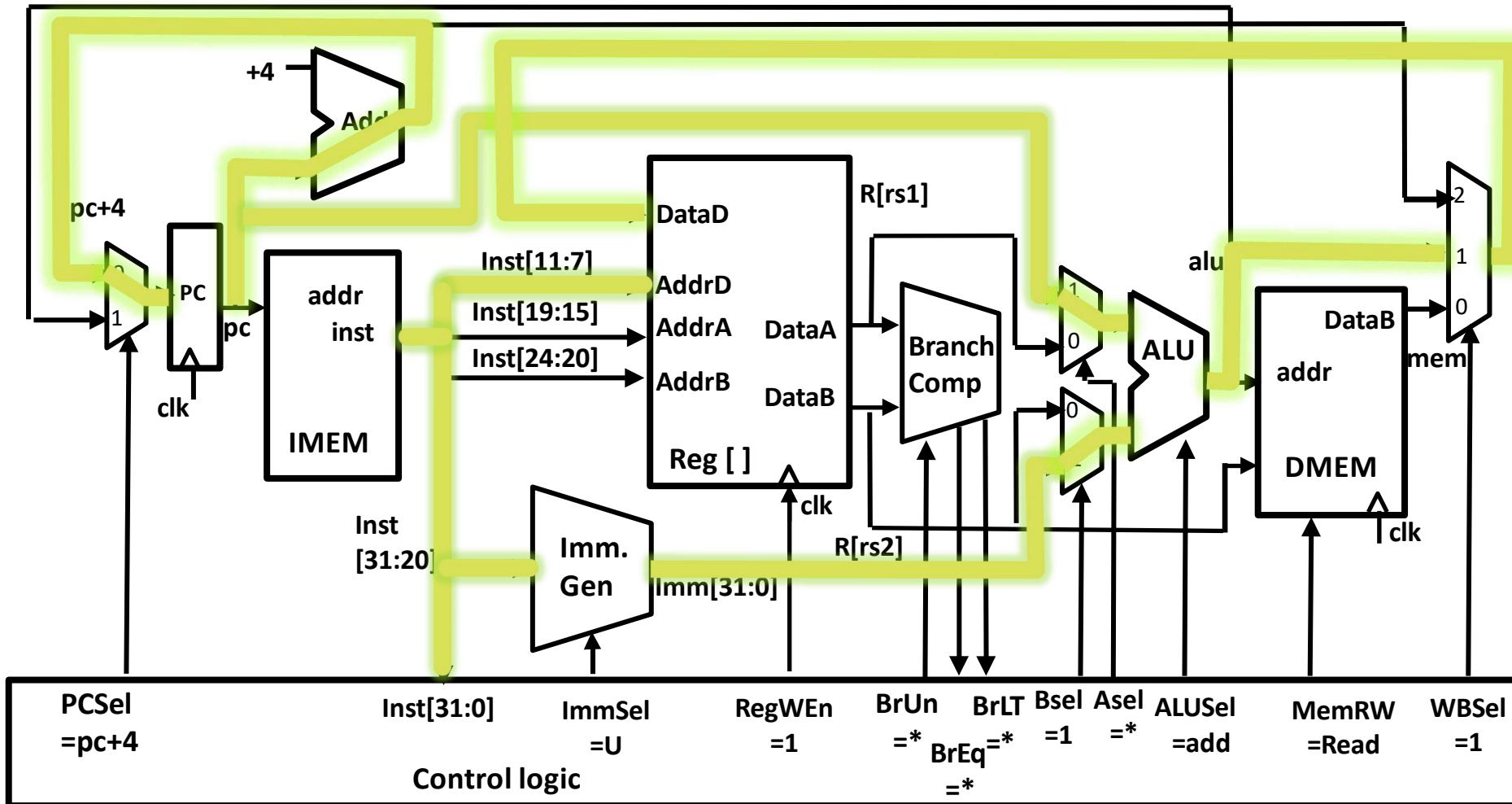
# Datapath With LUI,AUIPC



# Lighting Up LUI

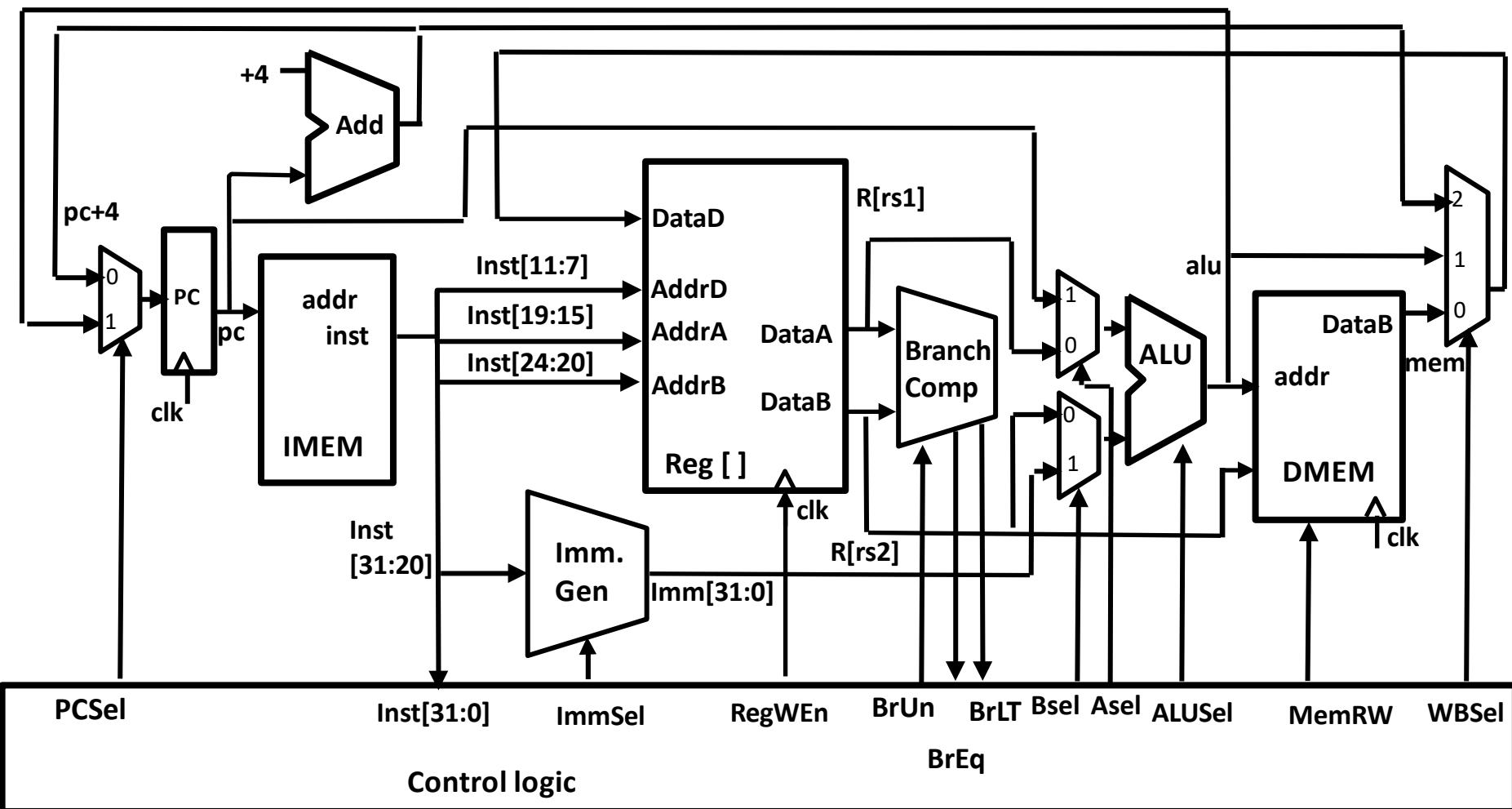


# Lighting Up AUIPC



**“And In Conclusion...”**

# Complete RV32I Datapath!





# Complete RV32I ISA!

## Open RISC-V Reference Card

Base Integer Instructions: RV32I						
Category	Name	Fmt	RV32I Base	Category	Name	Fmt
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2	Loads	Load Byte	I
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt		Load Halfword	I
	Shift Right Logical	R	SRL rd,rs1,rs2		Load Byte Unsigned	I
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt		Load Half Unsigned	I
	Shift Right Arithmetic	R	SRA rd,rs1,rs2		Load Word	I
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	Stores	Store Byte	S
Arithmetic	ADD	R	ADD rd,rs1,rs2		Store Halfword	S
	ADD Immediate	I	ADDI rd,rs1,imm		Store Word	S
	SUBtract	R	SUB rd,rs1,rs2	Branches	Branch =	B
	Load Upper Imm	U	LUI rd,imm		Branch ≠	B
	Add Upper Imm to PC	U	AUIPC rd,imm		Branch <	B
Logical	XOR	R	XOR rd,rs1,rs2		Branch ≥	B
	XOR Immediate	I	XORI rd,rs1,imm		Branch < Unsigned	B
	OR	R	OR rd,rs1,rs2		Branch ≥ Unsigned	B
	OR Immediate	I	ORI rd,rs1,imm	Jump & Link	J&L	J
	AND	R	AND rd,rs1,rs2		Jump & Link Register	I
	AND Immediate	I	ANDI rd,rs1,imm			
Compare	Set <	R	SLT rd,rs1,rs2	Synch	Synch thread	I
	Set < Immediate	I	SLTI rd,rs1,imm			
	Set < Unsigned	R	SLTU rd,rs1,rs2	Environment	CALL	I
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm		BREAK	I

Not in  
61C

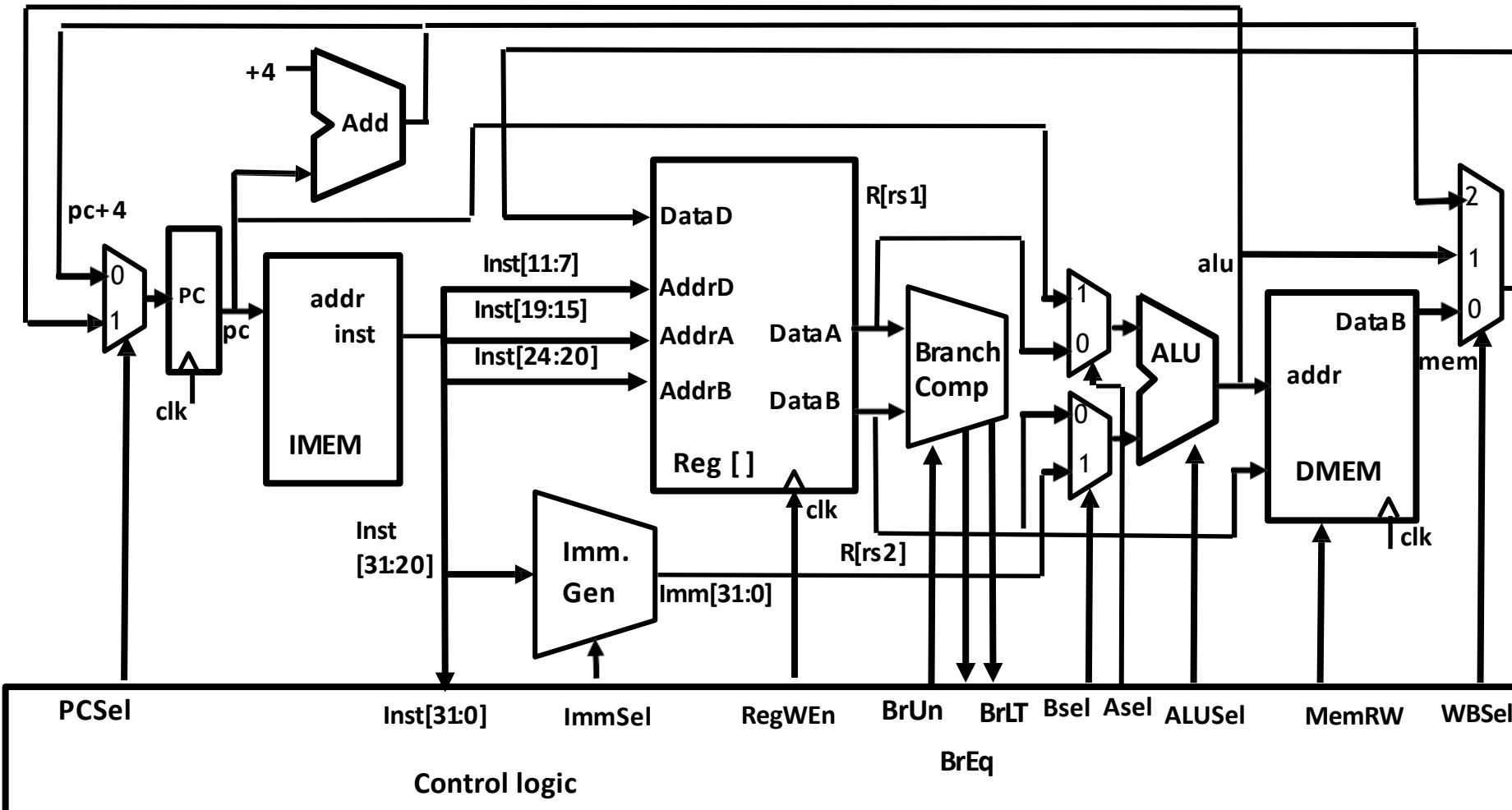
# Review

---

- 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
  - We still need to design it

# **Control and Status Registers**

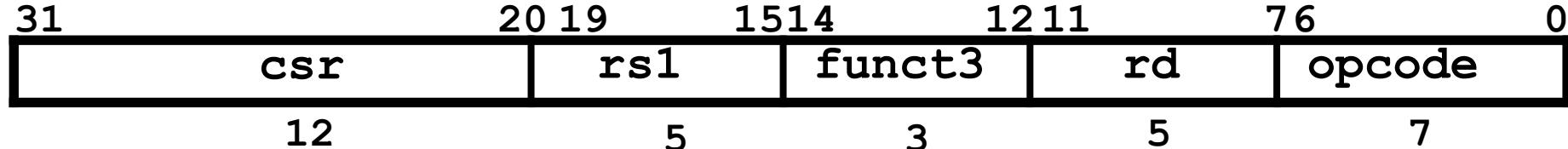
# Complete Single-Cycle RV32I Datapath!



# Control and Status Registers

- Control and status registers (CSRs) are separate from the register file ( $x0-x31$ )
  - Used for monitoring the status and performance
  - There can be up to 4096 CSRs
- Not in the base ISA, but almost mandatory in every implementation
  - ISA is modular
  - Necessary for counters and timers, and communication with peripherals

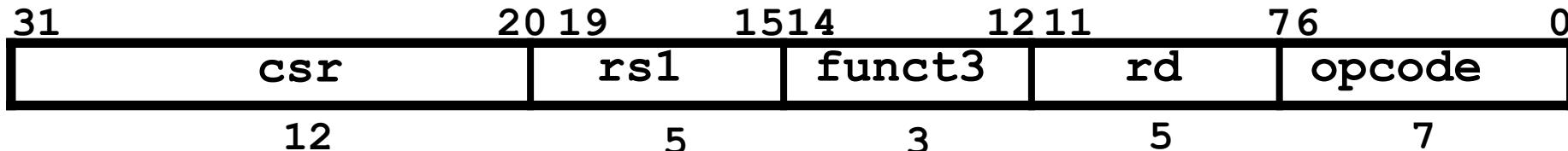
# CSR Instructions



source/dest	source	instr	rd	SYSTEM
	uimm[4:0]			1110011

Instr.	rd	rs	Read CSR?	Write CSR?
<b>csrrw</b>	<b>x0</b>	—	no	yes
<b>csrrw</b>	<b>!x0</b>	—	yes	yes
<b>csrrs/c</b>	—	<b>x0</b>	yes	no
<b>csrrs/c</b>	—	<b>!x0</b>	yes	yes

# CSR Instructions



source/dest      source    instr      rd      SYSTEM  
uimm[4:0] ← Zero-extended to 32b

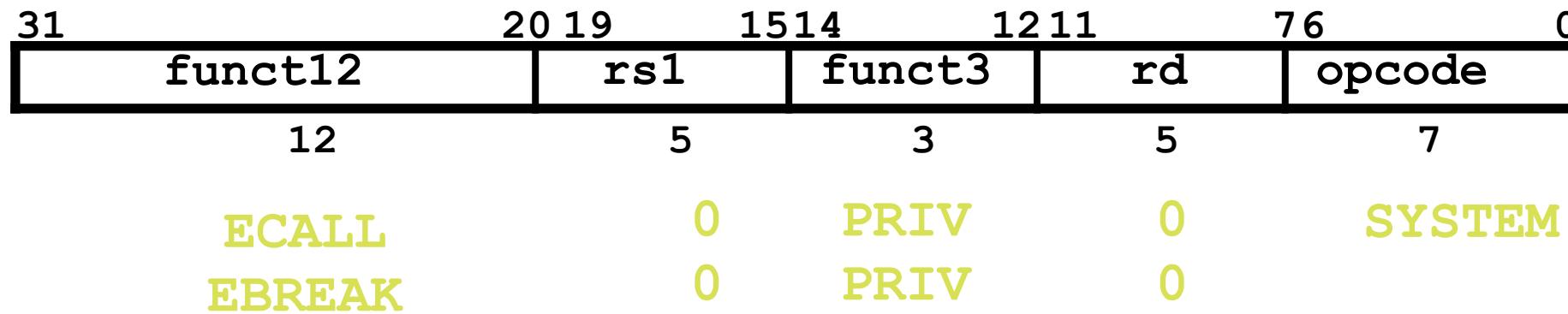
Instr.	rd	uimm	Read CSR?	Write CSR?
<b>csrrwi</b>	<b>x0</b>	—	no	yes
<b>csrrwi</b>	<b>!x0</b>	—	yes	yes
<b>csrrs/ci</b>	—	0	yes	no
<b>csrrs/ci</b>	—	!0	yes	yes

# CSR Instruction Example

- The CSRRW (Atomic Read/Write CSR) instruction ‘atomically’ swaps values in the CSRs and integer registers.
  - We will see more on ‘atomics’ later
- CSRRW reads the previous value of the CSR and writes it to integer register **rd**. Then writes **rs1** to CSR
- Pseudoinstruction **csrw csr, rs1** is  
**csrrw x0, csr, rs1**
  - **rd=x0**, just writes **rs1** to CSR
- Pseudoinstruction **csrwi csr, uimm** is  
**csrrwi x0, csr, uimm**
  - **rd=x0**, just writes **uimm** to CSR
- Hint Use write enable and clock...

# System Instructions

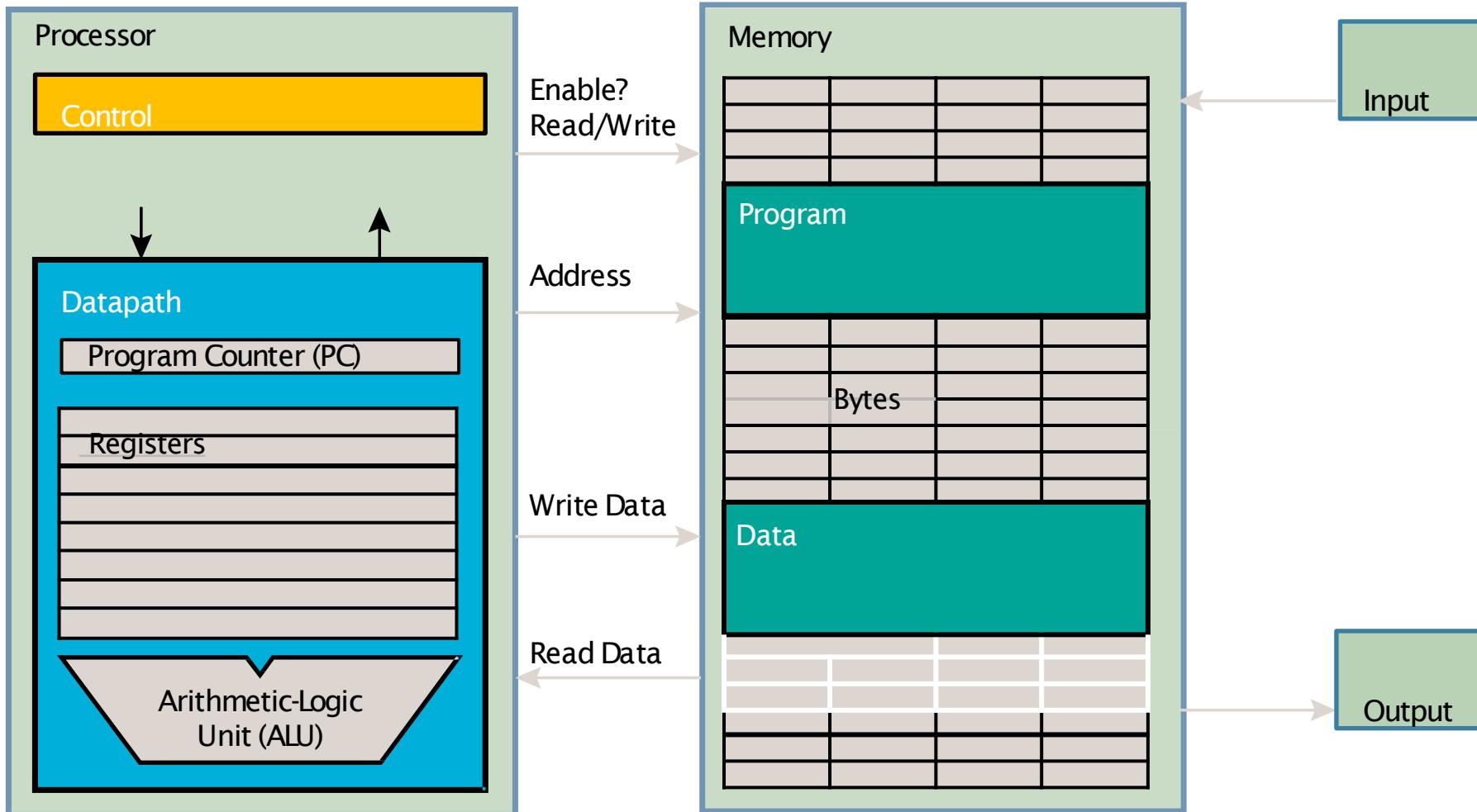
- **ecall** – (I-format) makes requests to supporting execution environment (OS), such as system calls (**syscalls**)
- **ebreak** – (I-format) used e.g. by debuggers to transfer control to a debugging environment



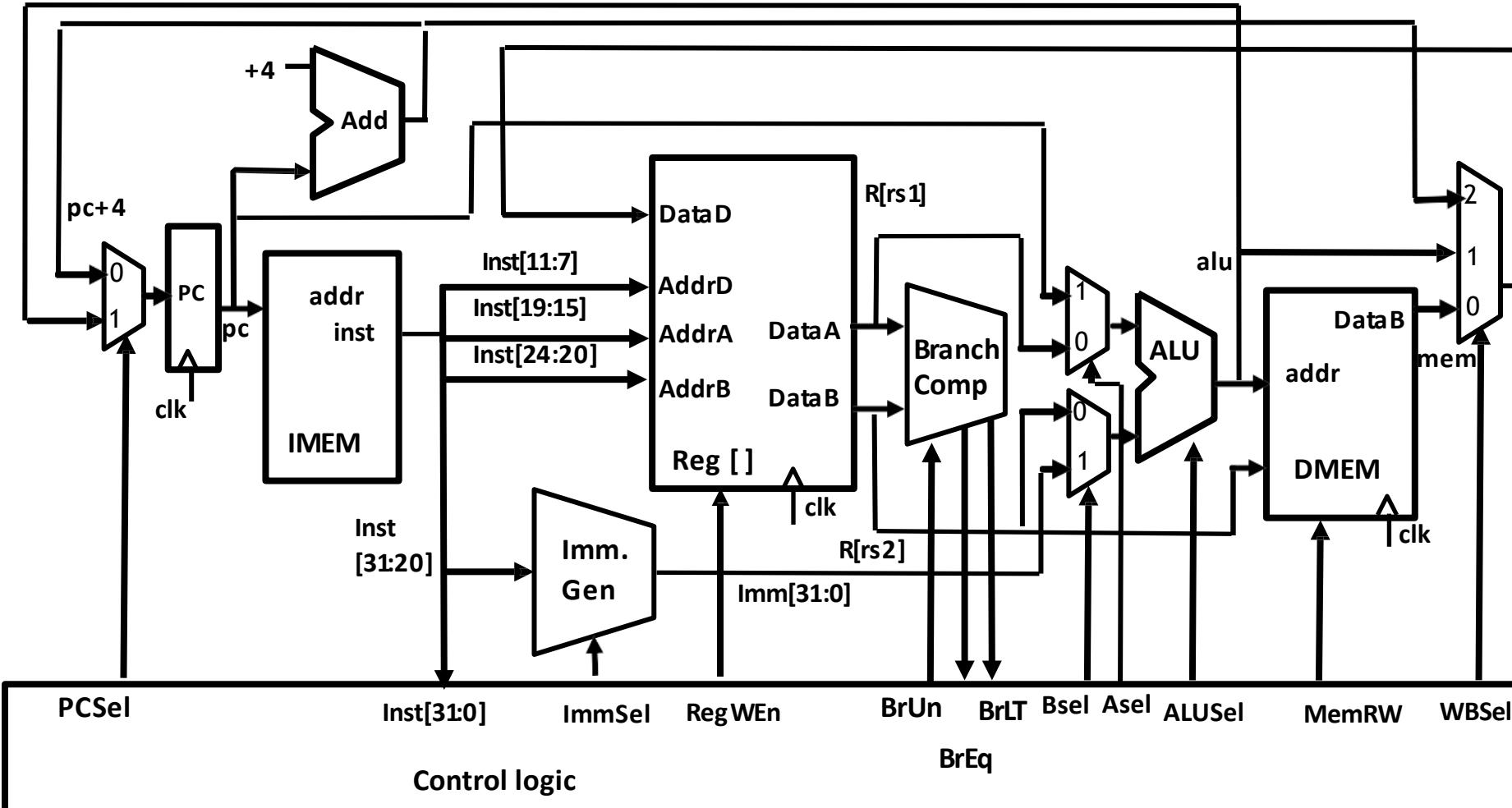
- **fence** – sequences memory (and I/O) accesses as viewed by other threads or co-processors

# Datapath Control

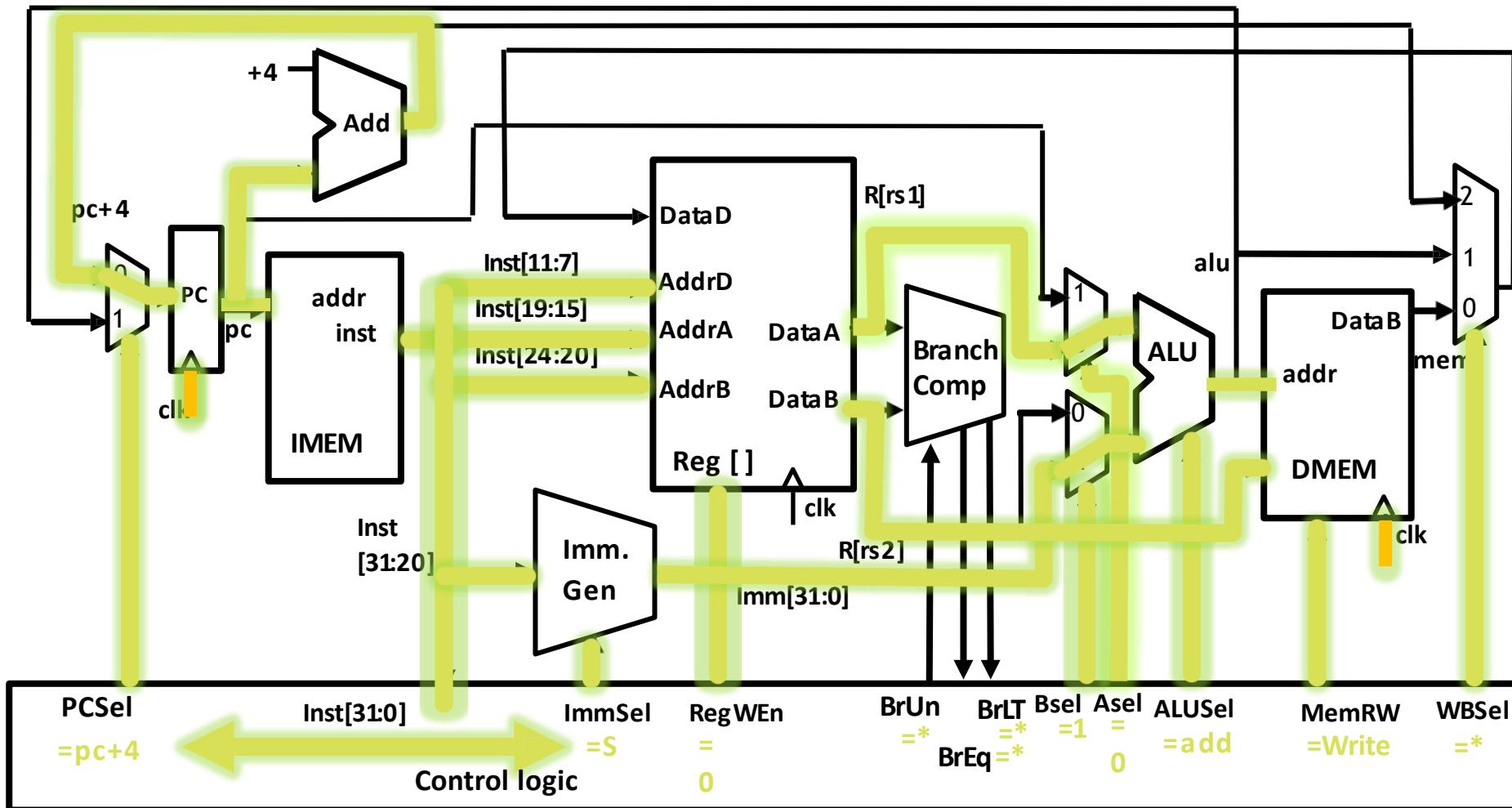
# Our Single-Core Processor



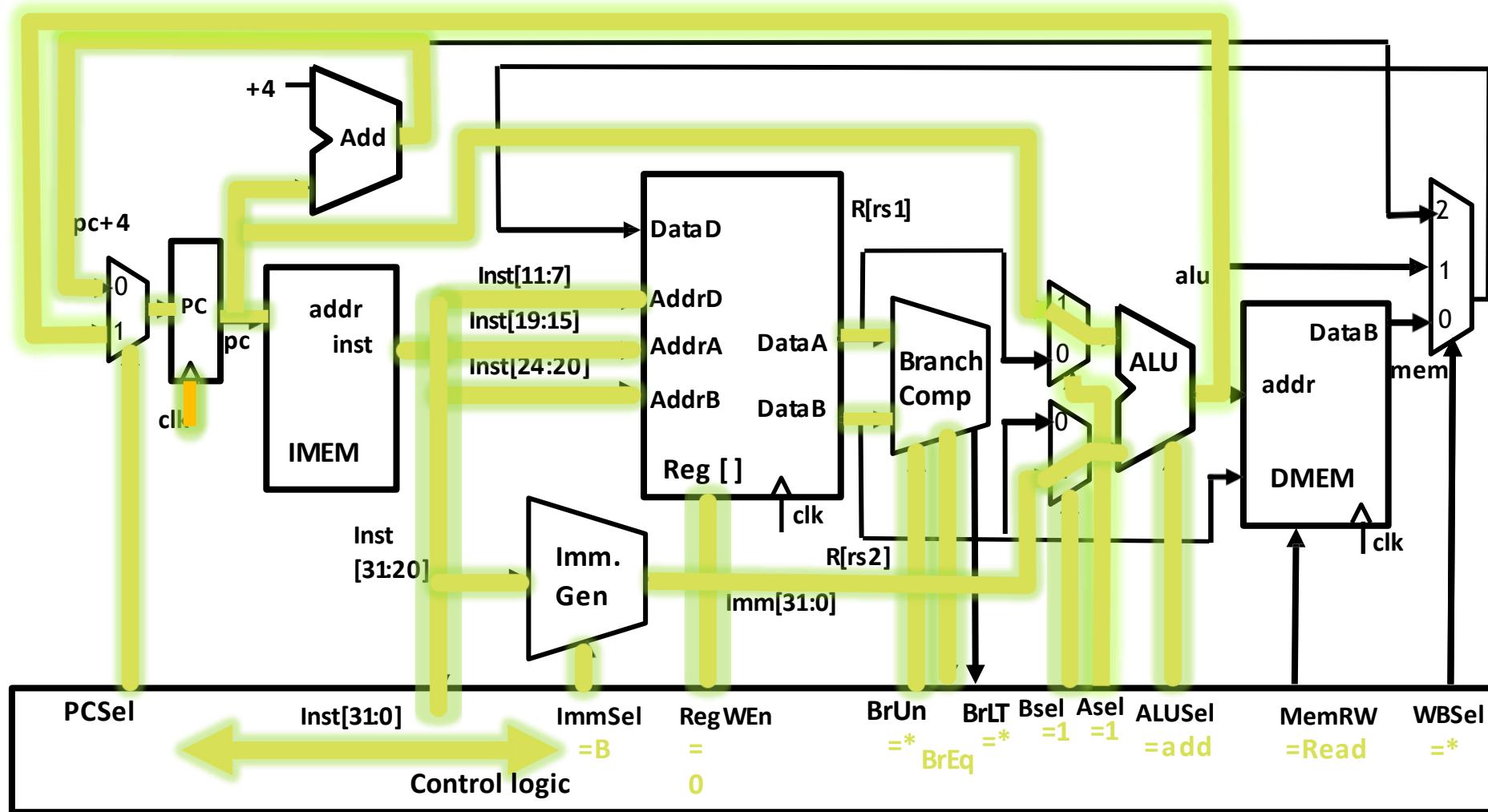
# Single-Cycle RV32I Datapath and Control



# Example: sw

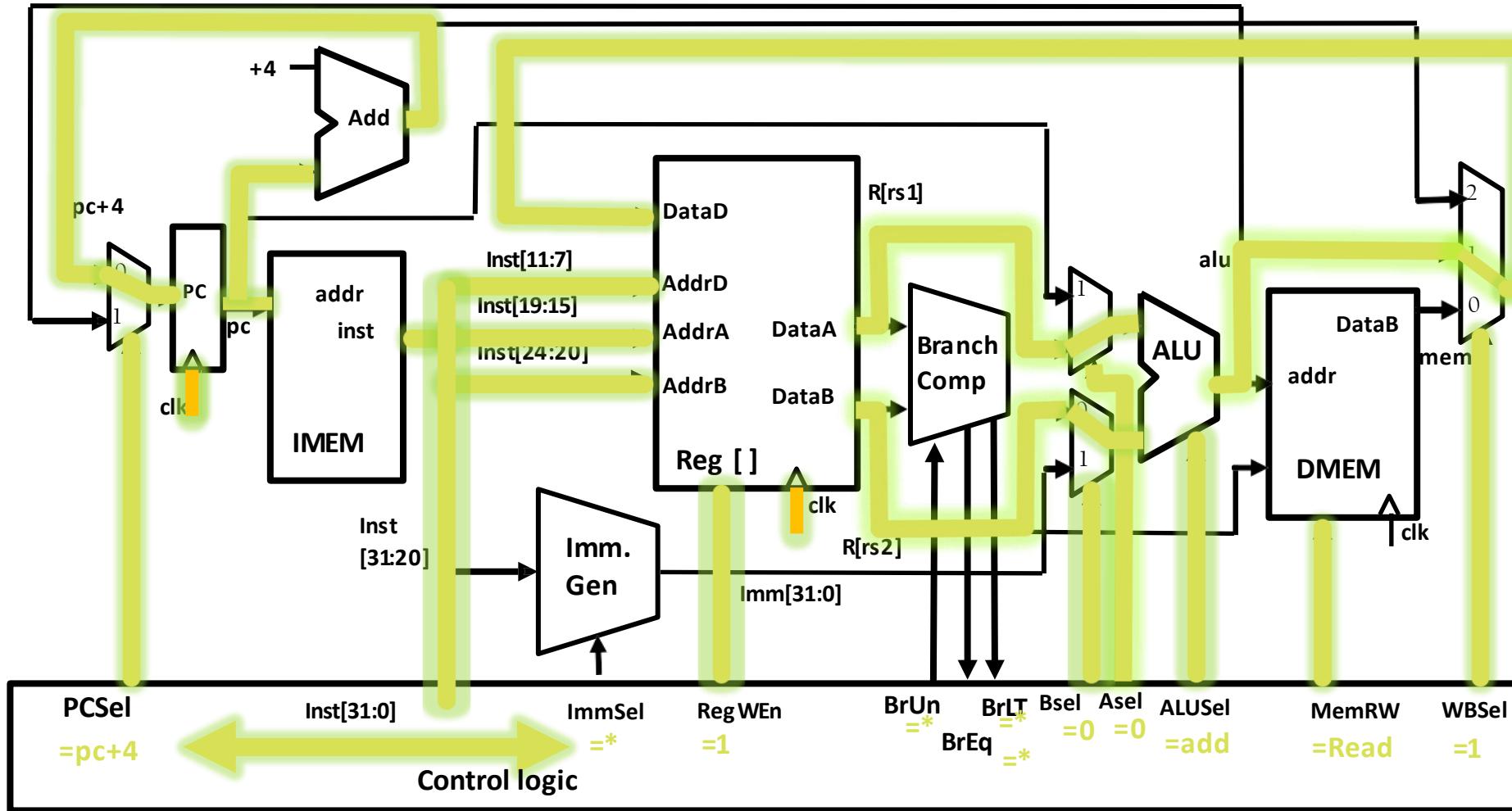


# Example: **beq**

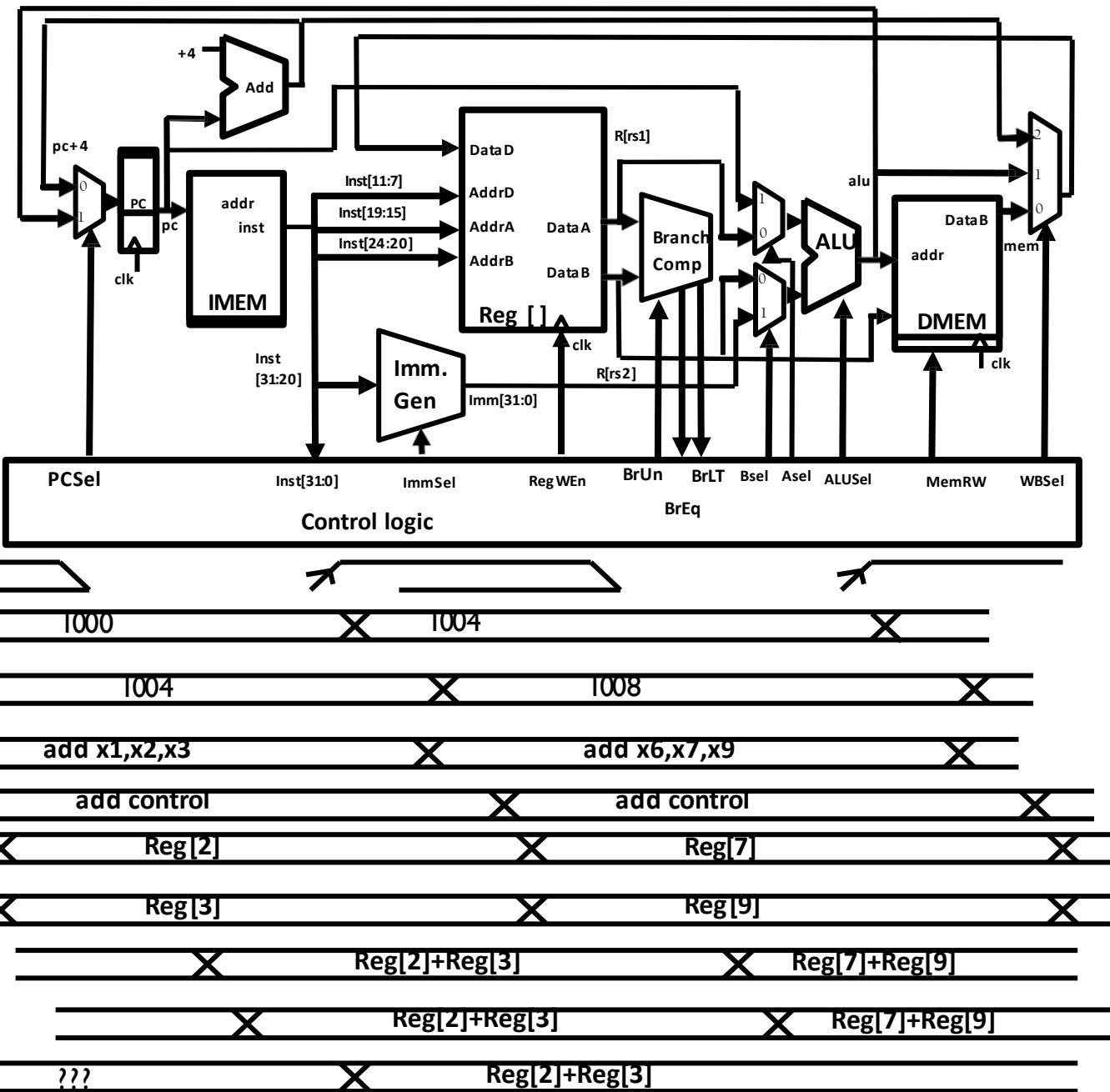


# **Instruction Timing**

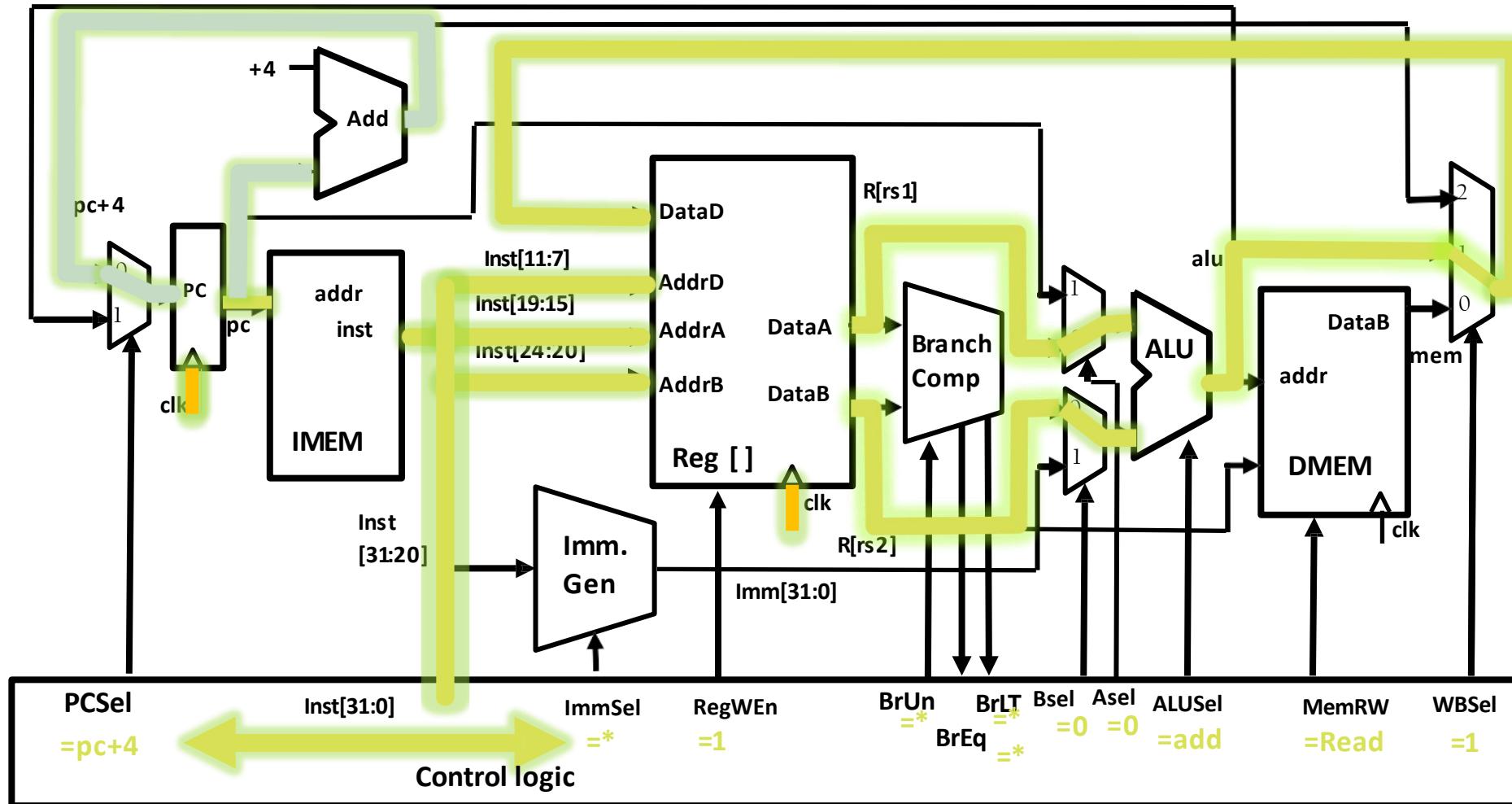
# Example: add



# Add Execution

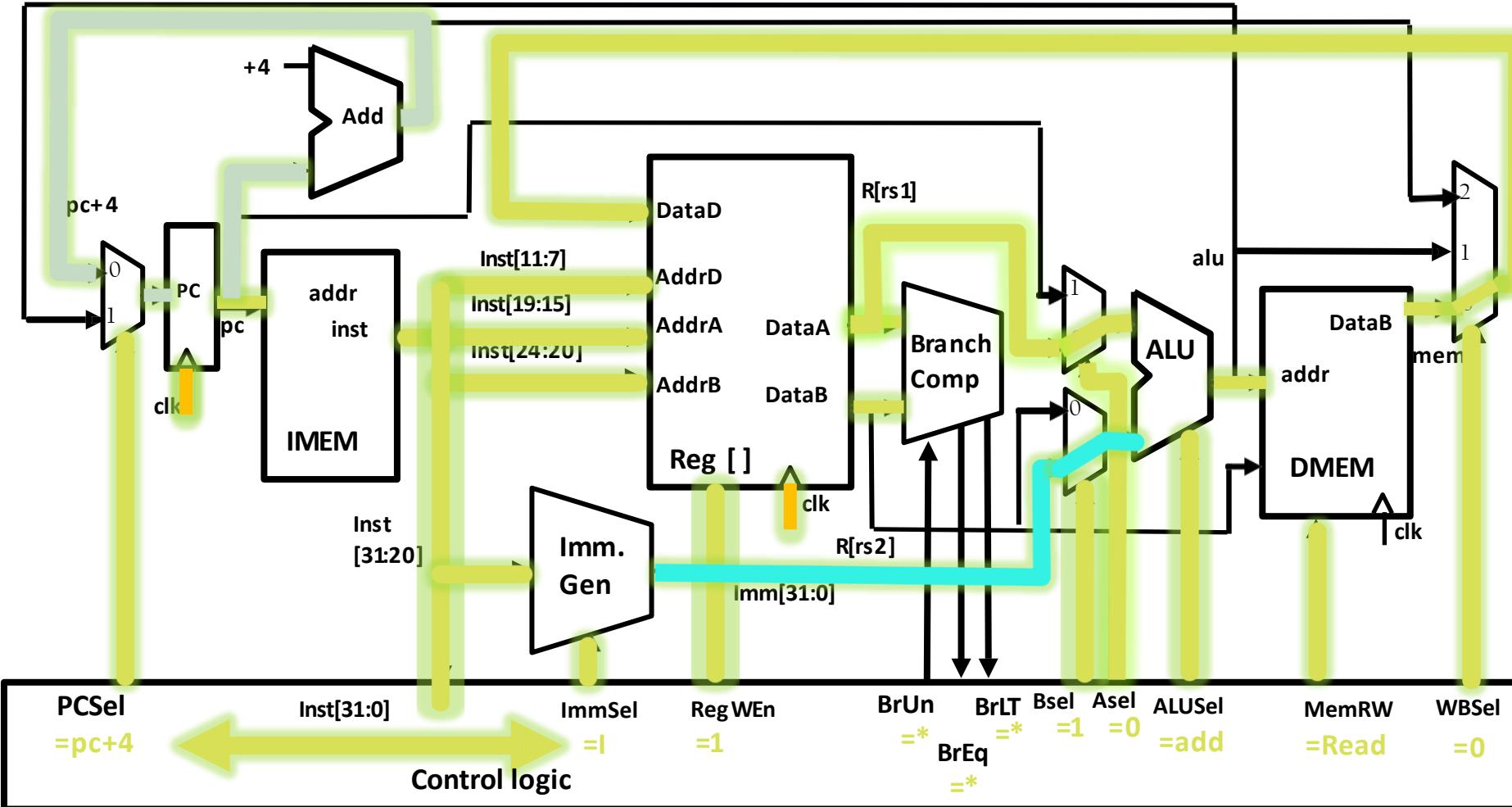


# Example: add timing



$$\begin{aligned}
 \text{Critical path} &= t_{\text{clk-q}} + \max \{ t_{\text{Add}} + t_{\text{mux}}, t_{\text{MEM}} + t_{\text{Reg}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} \} + t_{\text{setup}} \\
 &= t_{\text{clk-q}} + t_{\text{MEM}} + t_{\text{Reg}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} + t_{\text{setup}}
 \end{aligned}$$

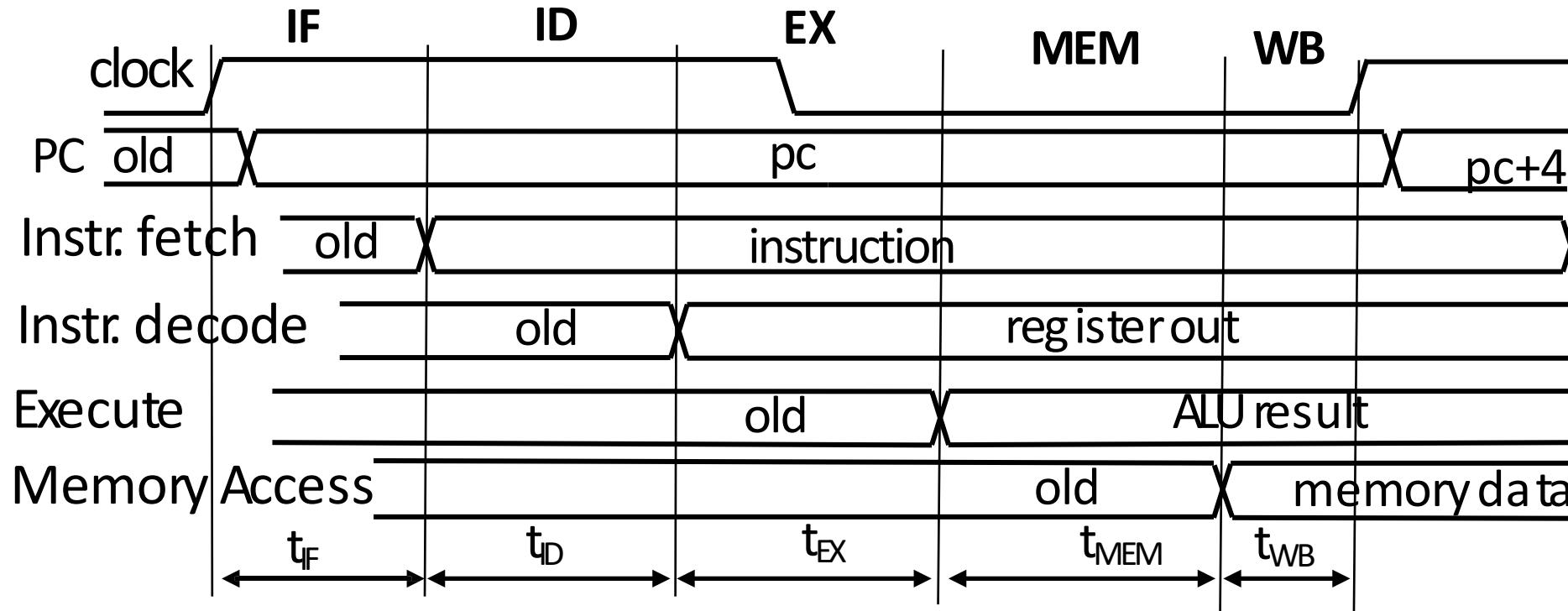
# Example: lw



$$\text{Critical path} = t_{\text{clk-q}} + \max \{ t_{\text{Add}} + t_{\text{mux}}, t_{\text{IMEM}} + t_{\text{Imm}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{DMEM}} + t_{\text{mux}}, \\ t_{\text{IMEM}} + t_{\text{Reg}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{DMEM}} + t_{\text{mux}} \} + t_{\text{setup}}$$

# Instruction Timing

IF-MEM	Reg Read	ALU	D-MEM	Reg W	Total
200 ps	100 ps	200 ps	200 ps	100 ps	<b>800 ps</b>



# Instruction Timing

Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X		X	600ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

- Maximum clock frequency
  - $f_{max} = 1/800\text{ps} = 1.25 \text{ GHz}$
- Most blocks idle most of the time
  - E.g.  $f_{max,ALU} = 1/200\text{ps} = 5 \text{ GHz!}$

# Control Logic Design

# Control Logic Truth Table

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

# Control Realization Options

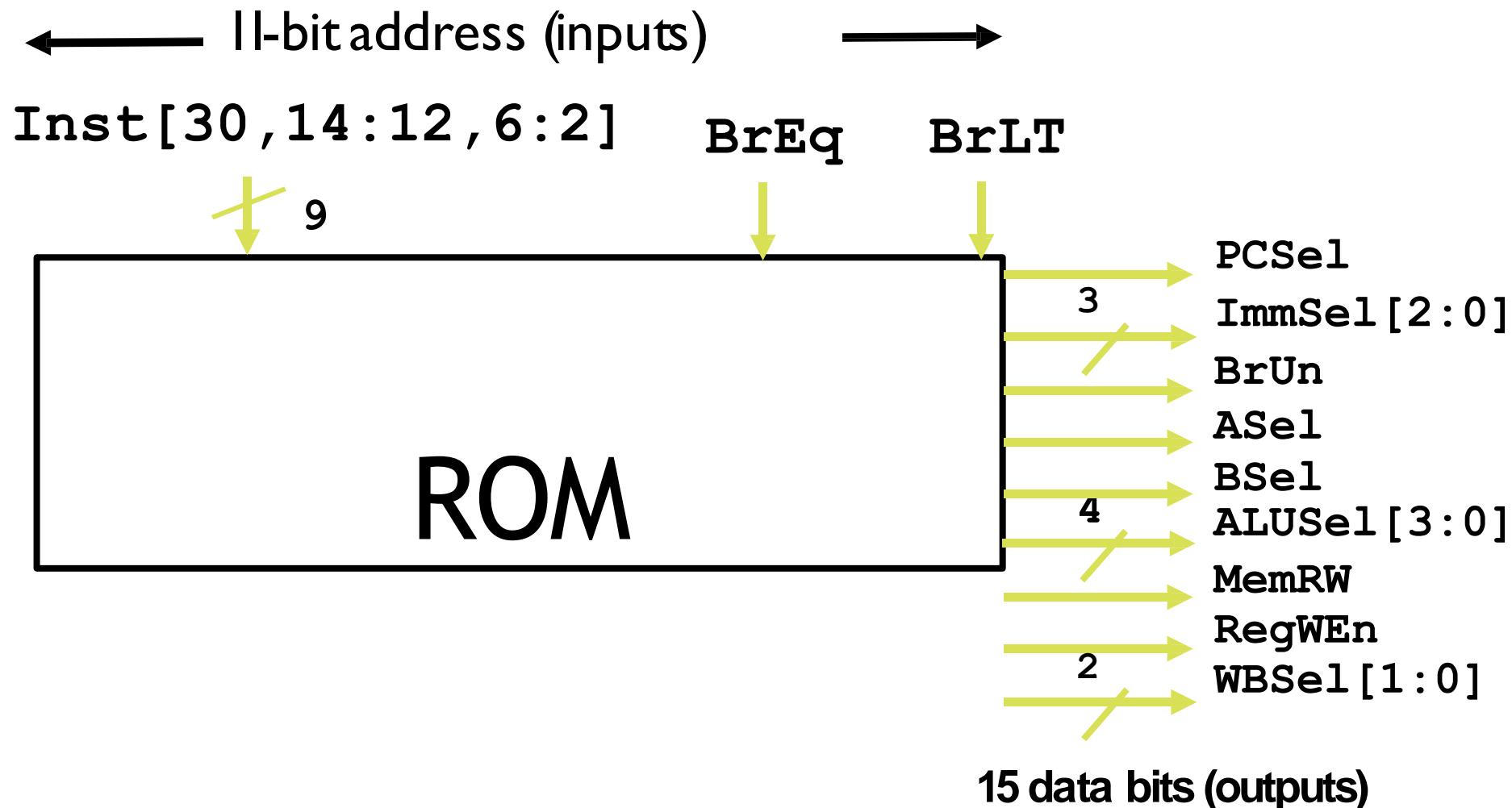
- ROM
  - “Read-Only Memory”
  - Regular structure
  - Can be easily reprogrammed
    - fix errors
    - add instructions
  - Popular when designing control logic manually
- Combinatorial Logic
  - Today, chip designers use logic synthesis tools to convert truth tables to networks of gates

# RV32I, A Nine-Bit ISA!

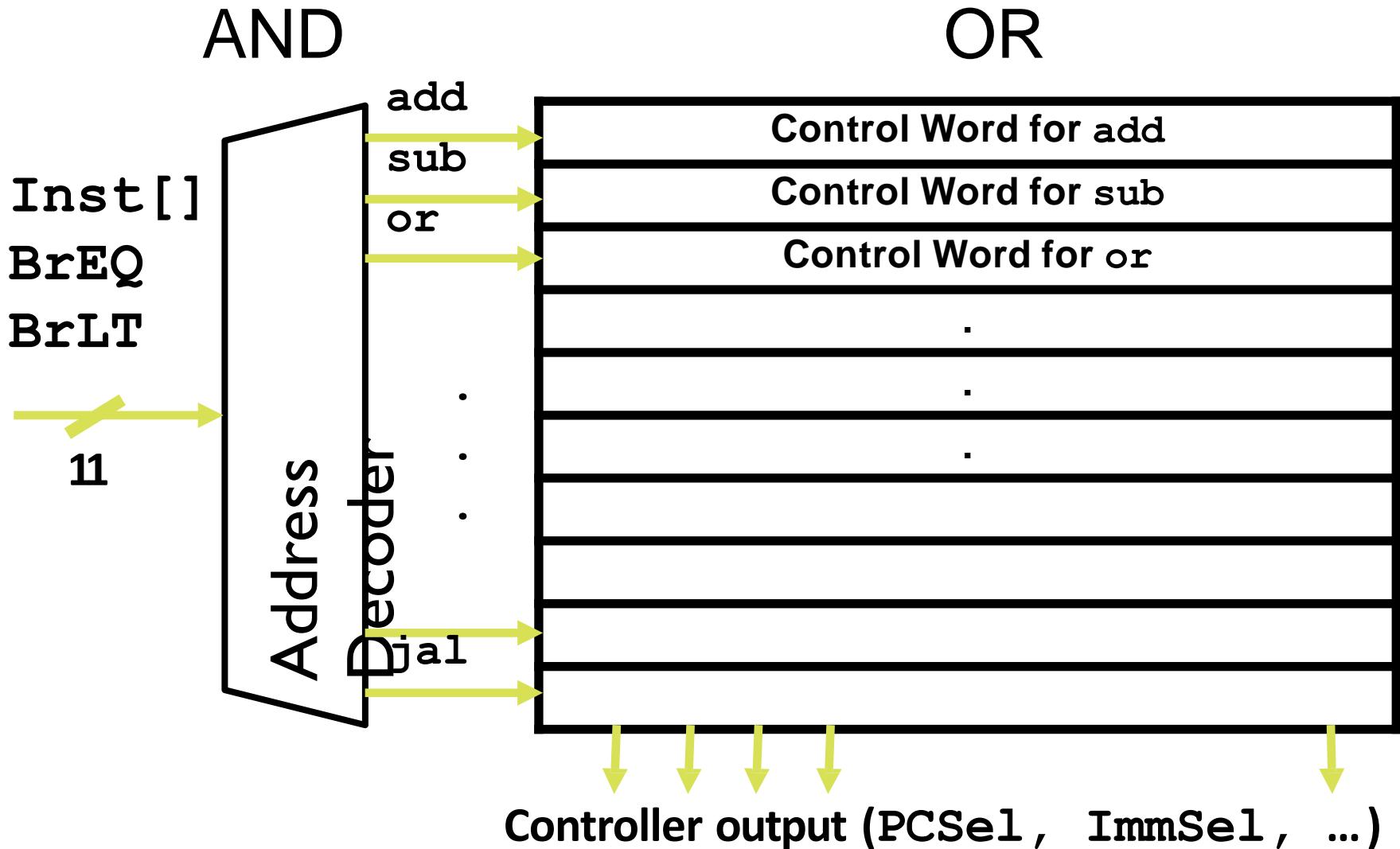
imm[31:12]			rd	0110	LUI
imm[31:12]			rd	0010	AUIPC
imm[20:10:1] 11 11 19:12			rd	1101	JAL
imm[11:0]	rs1	000	rd	1100	JALR
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	BGEU
imm[11:0]	rs1	000	rd	0000	LB
imm[11:0]	rs1	001	rd	0000	LH
imm[11:0]	rs1	010	rd	0000	LW
imm[11:0]	rs1	100	rd	0000	LBU
imm[11:0]	rs1	101	rd	0000	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	SW
imm[11:0]	rs1	000	rd	0010	ADDI
imm[11:0]	rs1	010	rd	0010	SLTI
imm[11:0]	rs1	011	rd	0010	SLTIU
imm[11:0]	rs1	100	rd	0010	XORI
imm[11:0]	rs1	110	rd	0010	ORI
imm[11:0]	rs1	111	rd	0010	ANDI
000000	shamt	rs1	001	rd	SLLI
000000	shamt	rs1	101	rd	SRLI
000000	shamt	rs1	101	rd	SRAT
000000	rs2	rs1	000	rd	ADD
000000	rs2	rs1	000	rd	SUB
000000	rs2	rs1	001	rd	SLI
000000	rs2	rs1	010	rd	SLT
000000	rs2	rs1	011	rd	SLTU
000000	rs2	rs1	100	rd	XOR
000000	rs2	rs1	101	rd	SRL
000000	rs2	rs1	110	rd	SPA
000000	rs2	rs1	111	rd	OR
fm	pred	succ	rs1	000	AND
0000000000000000			00000	000	FENCE
0000000000000001			00000	000	ECALL
			00000	000	EBREAK

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

# ROM-based Control



# ROM Controller Implementation



# Combinational Logic Control

- Simplest example: BrUn

			inst[14:12]	inst[6:2]		
imm[12:10:5]	rs2	rs1	000	imm[4:1][11]	1100011	BE Q
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

- How to decode whether BrUn is 1?

BrUn = Inst [13] • Branch

# Control Logic to Decode add

$\text{add} = i[30] \cdot i[14] \cdot i[13] \cdot i[12] \cdot \text{R-type}$   
inst[30] inst[14:12] inst[6:2]

000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SRLI
010000	shamt	rs1	101	rd	0010011	SRAI
000000	rs2	rs1	000	rd	0110011	ADD
010000	rs2	rs1	000	rd	0110011	SUB
000000	rs2	rs1	001	rd	0110011	SLL
000000	rs2	rs1	010	rd	0110011	SLT
000000	rs2	rs1	011	rd	0110011	SLTU
000000	rs2	rs1	100	rd	0110011	XOR
000000	rs2	rs1	101	rd	0110011	SRL
010000	rs2	rs1	101	rd	0110011	SRA
000000	rs2	rs1	110	rd	0110011	OR
000000	rs2	rs1	111	rd	0110011	AND

R-type =  $\overline{i[6]} \cdot i[5] \cdot i[4] \cdot \overline{i[3]} \cdot \overline{i[2]} \cdot \text{RV32I}$

RV32I =  $i[1] \cdot i[0]$

**“And In  
Conclusion...”**

# Call home, we've made HW/SW contact!

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

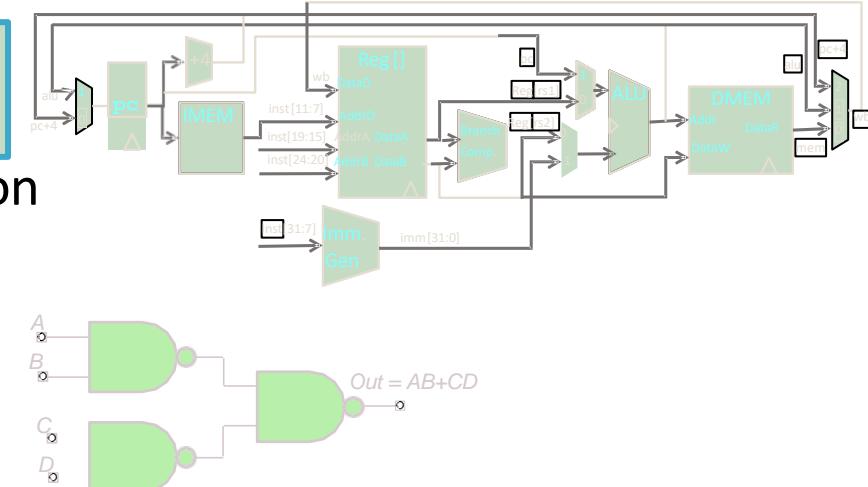
Machine Language  
Program (RISC-V)

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

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

| Architecture Implementation

Logic Circuit Description  
(Circuit Schematic Diagrams)



# “And In conclusion...”

- We have built a processor!
  - Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions
  - Critical path changes
- 5 Phases of execution
  - IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases
- Controller specifies how to execute instructions
  - Implemented as ROM or logic