Computer Architecture 1

Computer Organization and Design

T H E H A R D W A R E / S O F T W A R E I N T E R F A C E

[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 Nikolíc, © 2020, UC Berkeley]

RISC-V Processor Design

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

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

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

Compiler

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

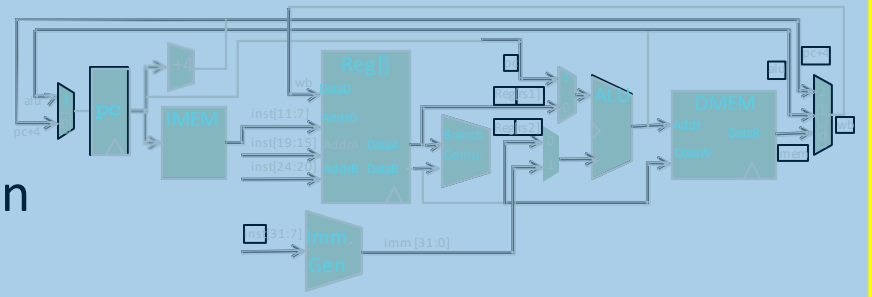
```
lw      x3, 0(x10)
lw      x4, 4(x10)
sw      x4, 0(x10)
sw      x3, 4(x10)
```

Assembler

Machine Language Program (RISC-V)

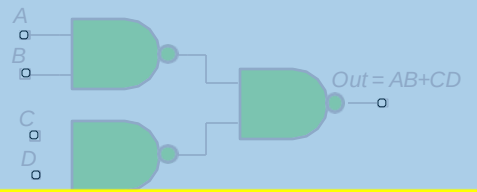
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

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

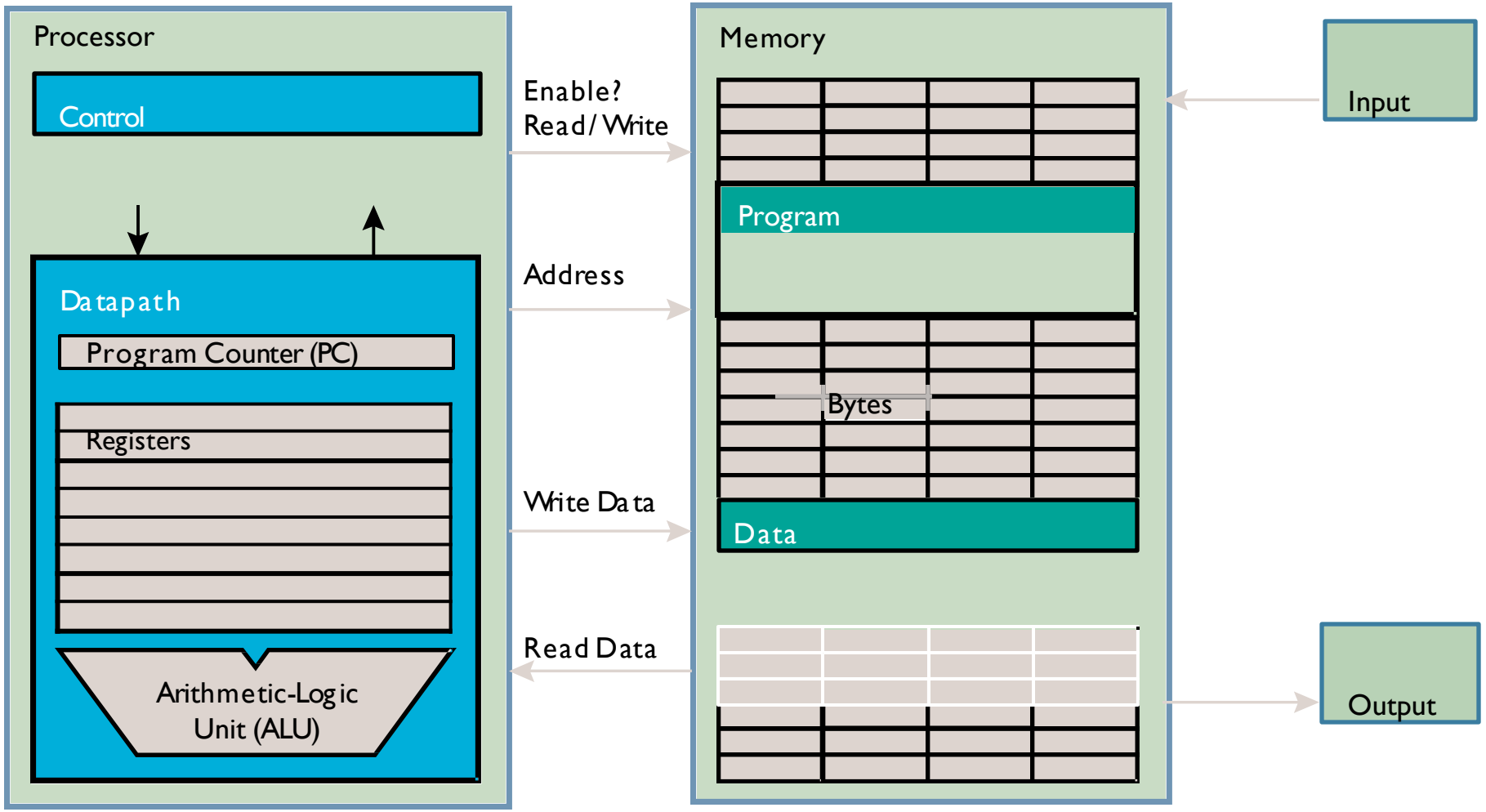


Architecture Implementation

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

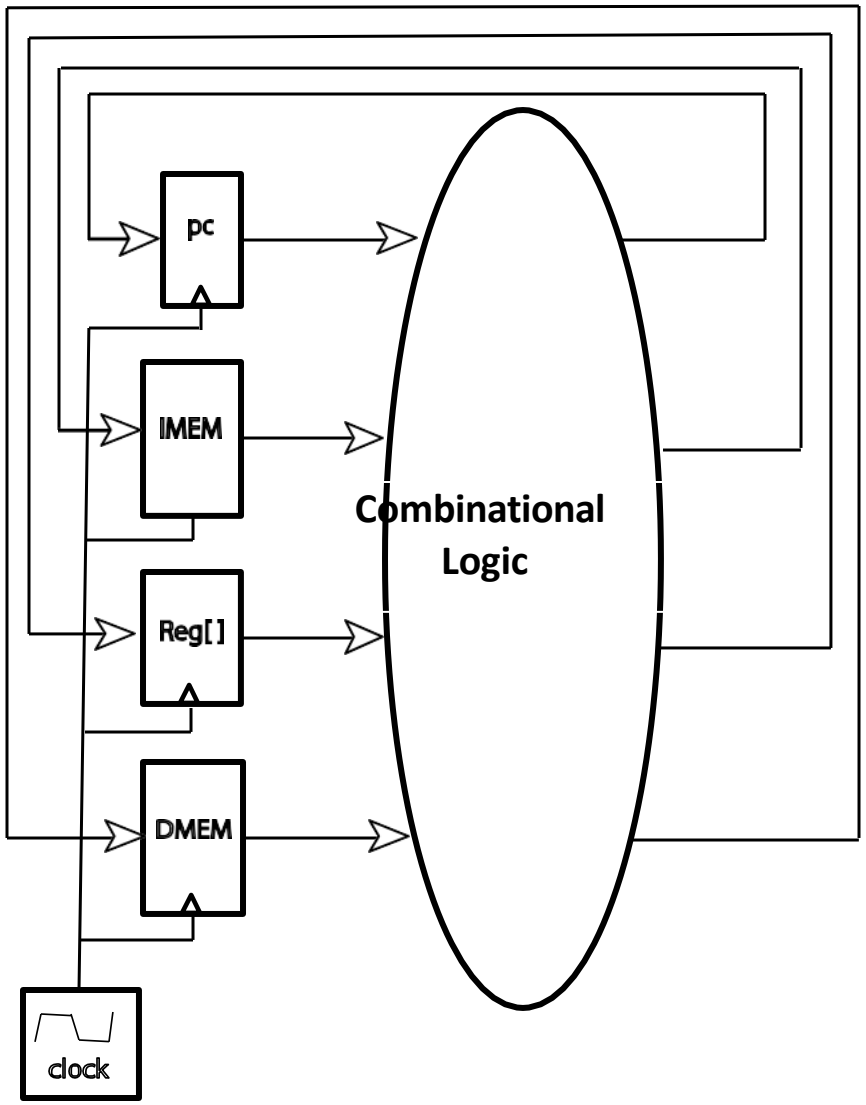
Base Integer Instructions: RV32I

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

Not in 61C

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 settles 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

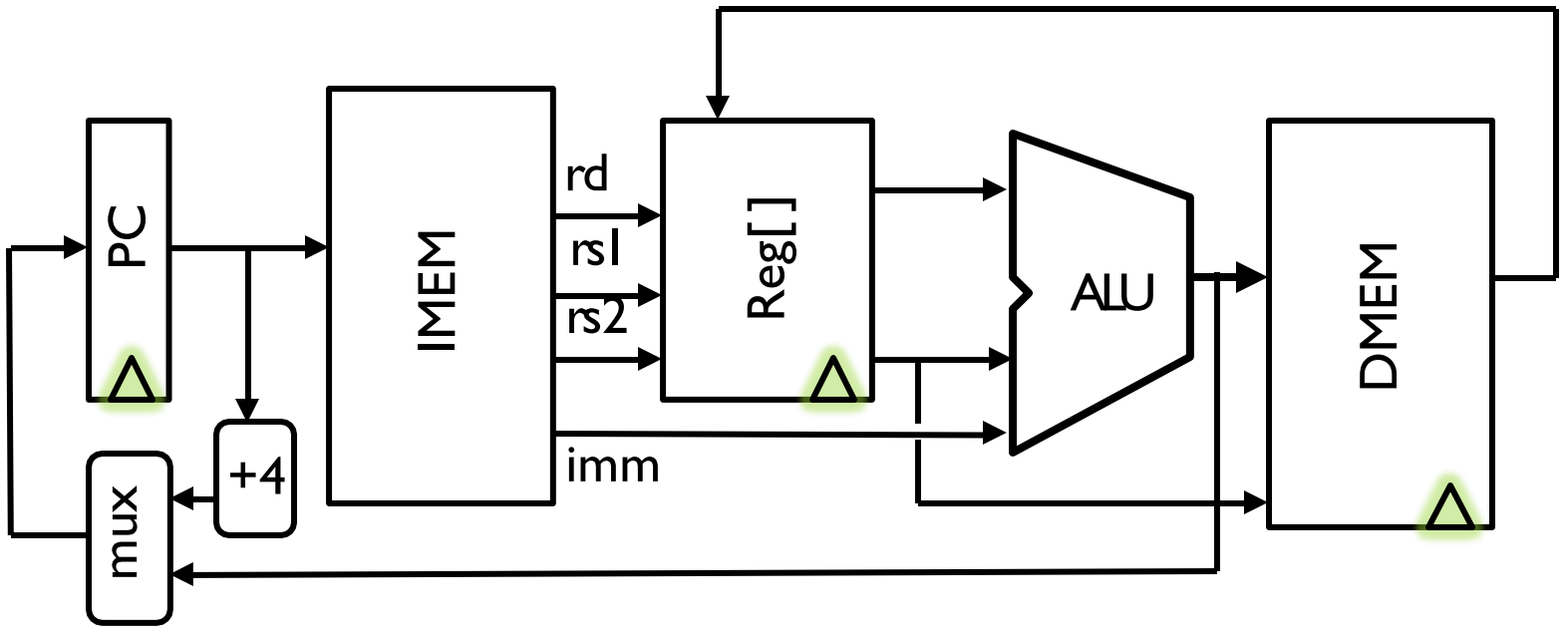
Stages of the Datapath : Overview

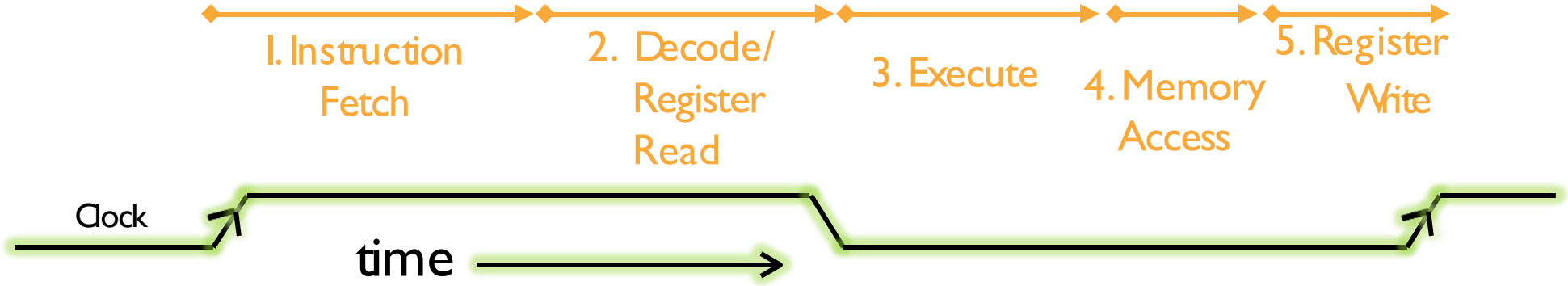
- Problem: a single, "monolithic" block that "executes an instruction" (performs all necessary operations beginning with fetching the instruction) would be too bulky and inefficient
- Solution: break up the process of "executing an instruction" into stages, and then connect the stages to create the whole 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)***

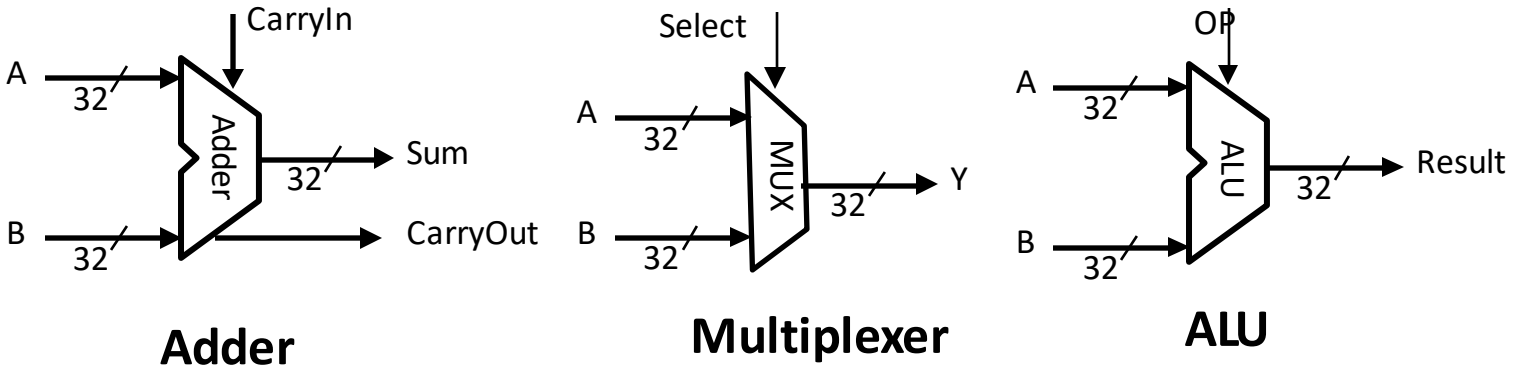
Basic Phases of Instruction Execution





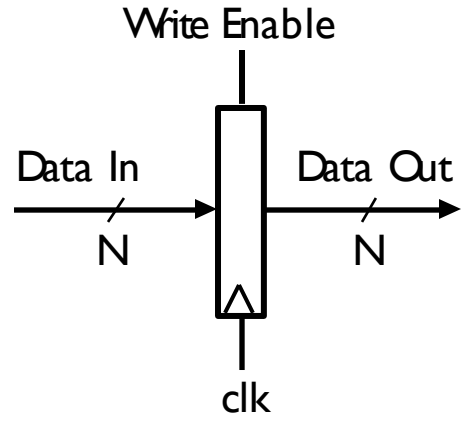
Datapath Components: Combinational

- Combinational elements



- Storage elements + clocking methodology
- Building blocks

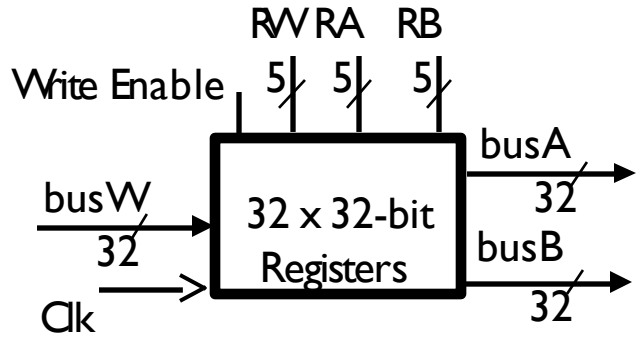
Datapath Elements: State and Sequencing (1/3)



- 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

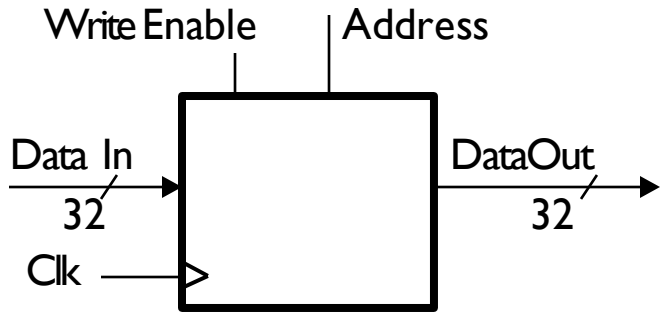
Datapath Elements: State and Sequencing (2/3)

- 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

Review: R-Type Instructions

31 30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0
R-format : ALU					
[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]
7	5	5	3	5	7
func7	rs2	rs1	func3	rd	opcode
0000000	rs2	rs1	000 : ADD	rd	0110011:OP-R
0100000	rs2	rs1	000 : SUB	rd	0110011:OP-R
0000000	rs2	rs1	001 : SLL	rd	0110011:OP-R
0000000	rs2	rs1	010 : SLT	rd	0110011:OP-R
0000000	rs2	rs1	011 : SLTU	rd	0110011:OP-R
0000000	rs2	rs1	100 : XOR	rd	0110011:OP-R
0000000	rs2	rs1	101 : SRL	rd	0110011:OP-R
0100000	rs2	rs1	101 : SRA	rd	0110011:OP-R
0000000	rs2	rs1	110 : OR	rd	0110011:OP-R
0000000	rs2	rs1	111 : AND	rd	0110011:OP-R

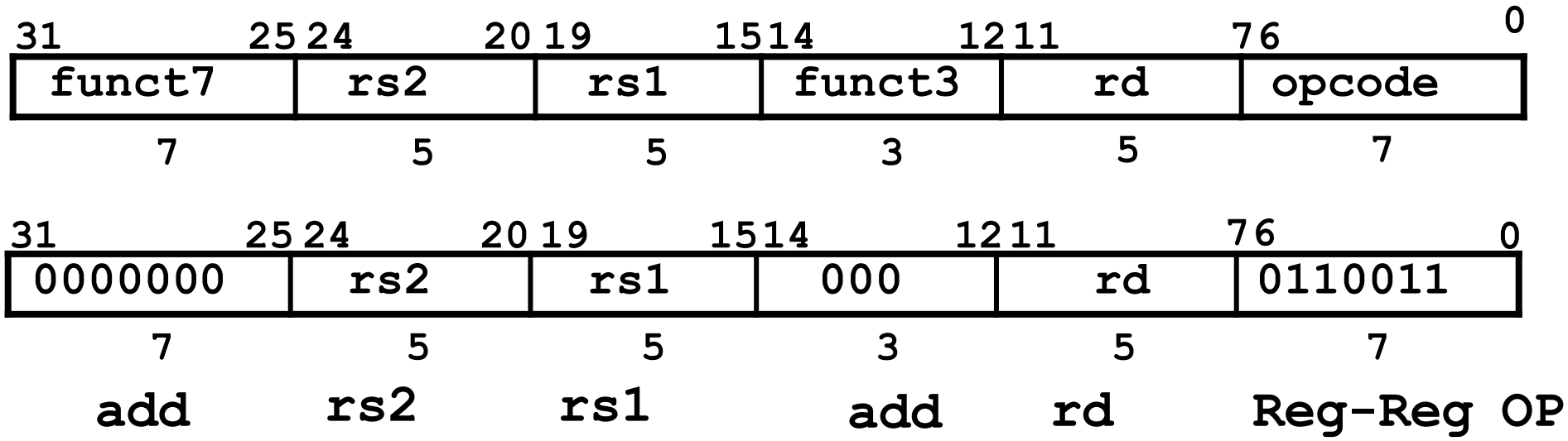
- E.g. Addition/subtraction **`add rd, rs1, rs2`**

  `R[rd] = R[rs1] + R[rs2]`

  `sub rd, rs1, rs2`

  `R[rd] = R[rs1] - R[rs2]`

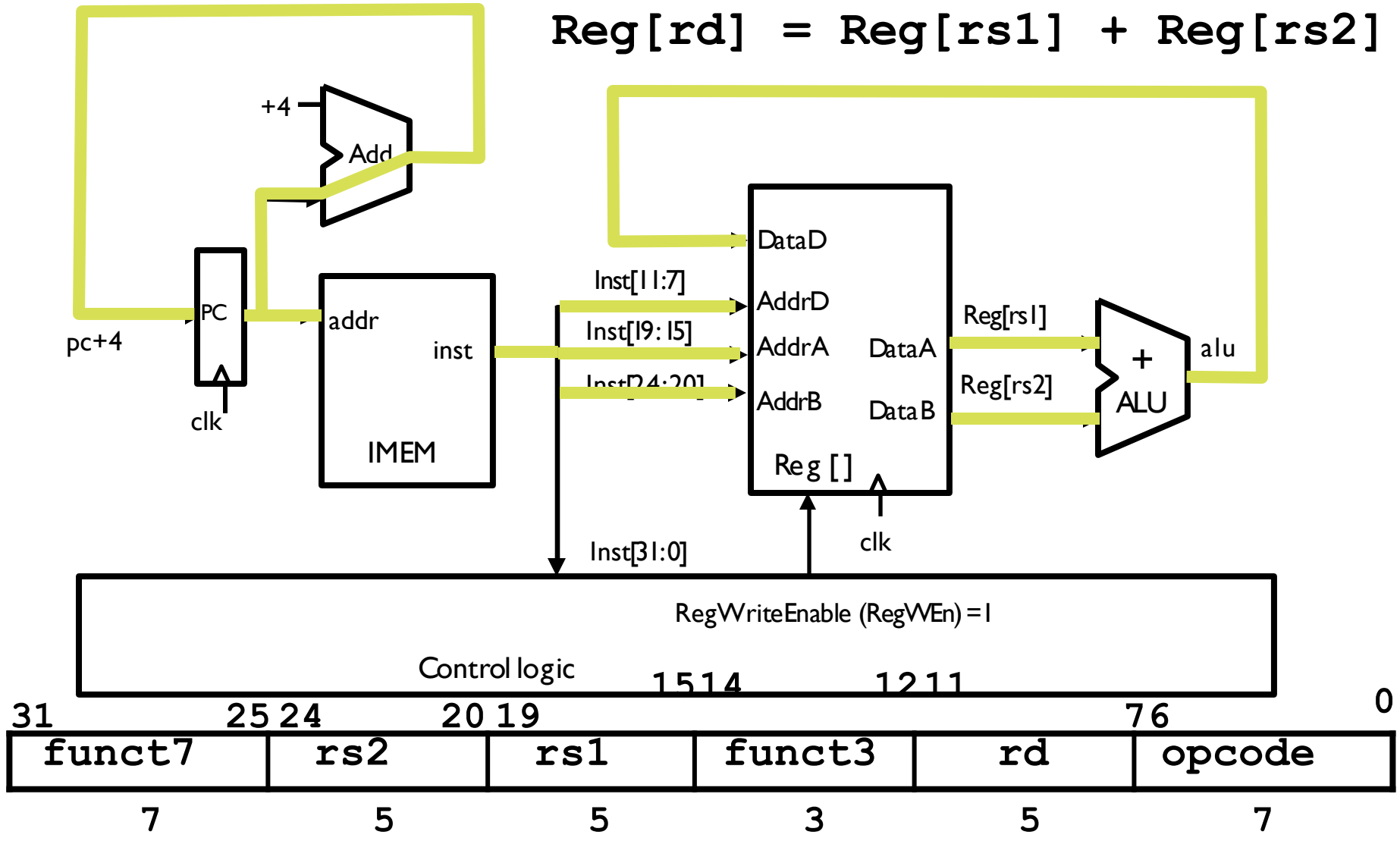
Implementing the add instruction



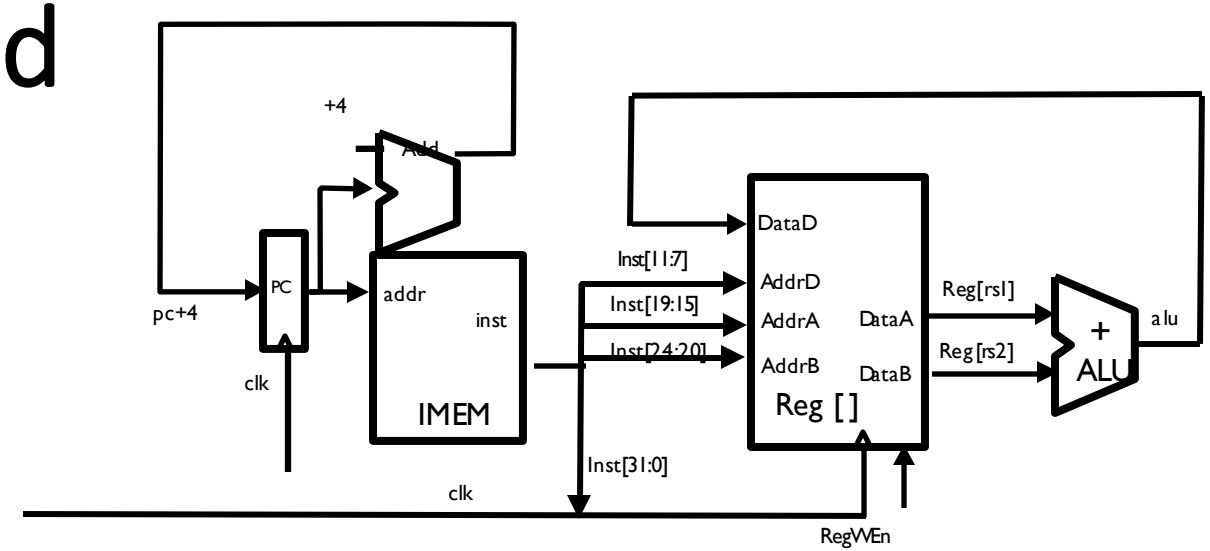
`add rd, rs1, rs2`

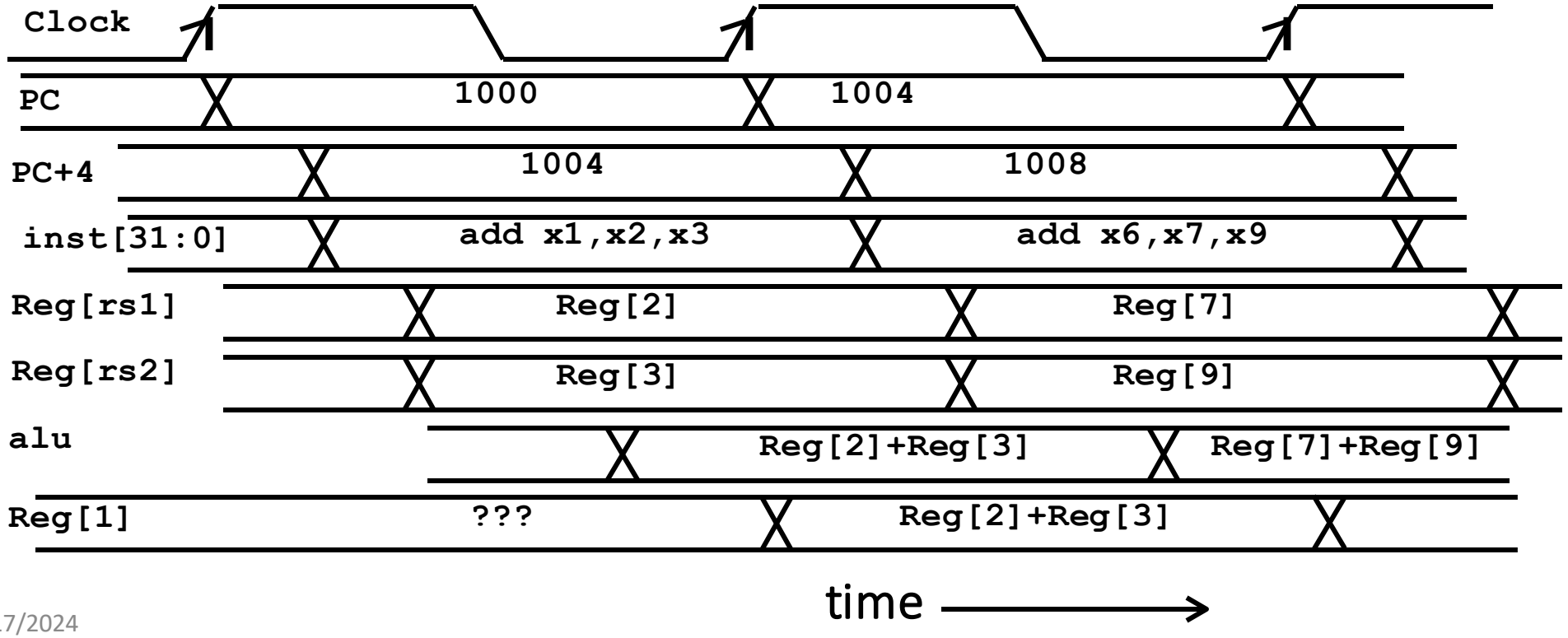
- Instruction makes two changes to machine's state:
  - **`Reg[rd] = Reg[rs1] + Reg[rs2]`**
  - **`PC = PC + 4`**

Datapath for add



Timing Diagram for add





Sub Datapath

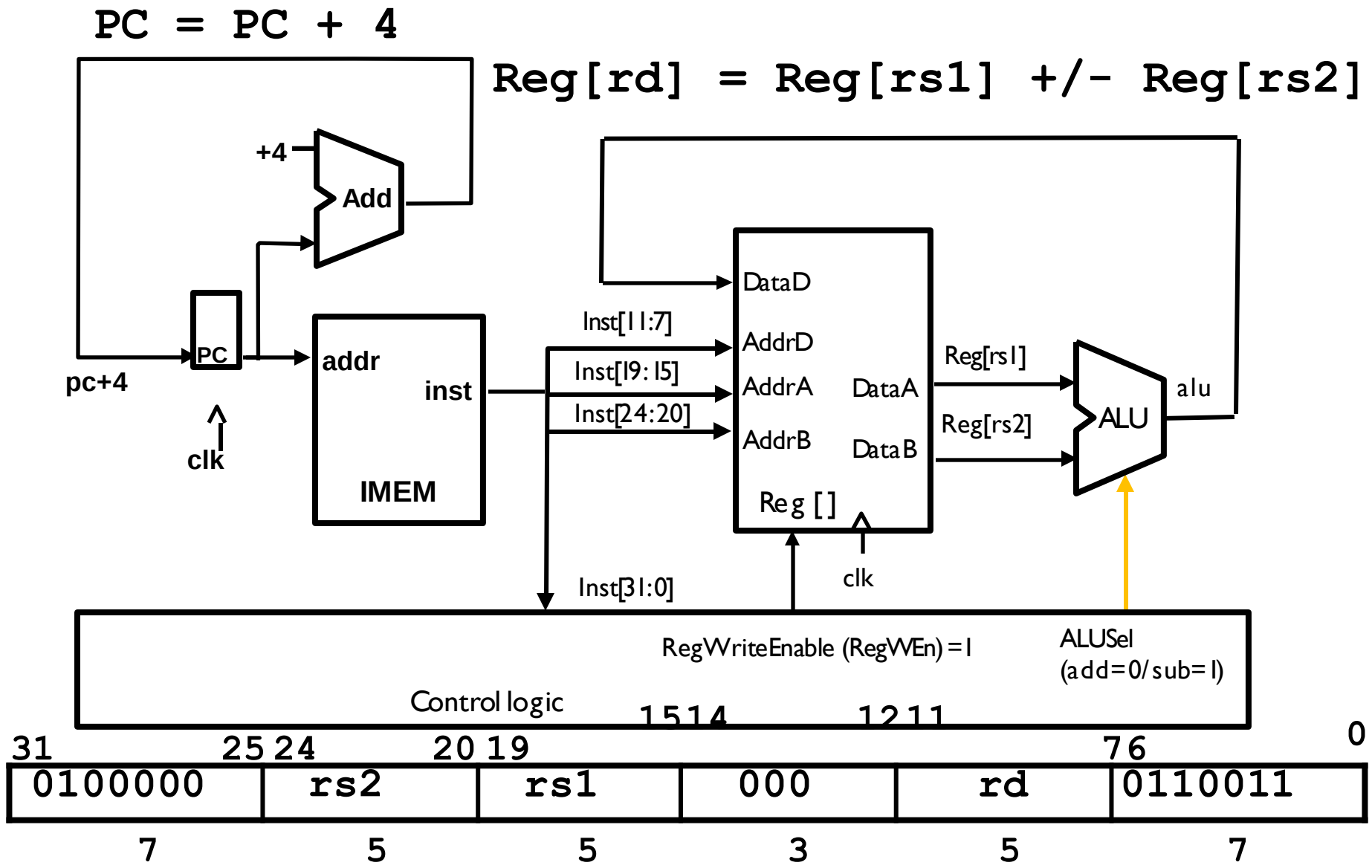
Implementing the sub instruction

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



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

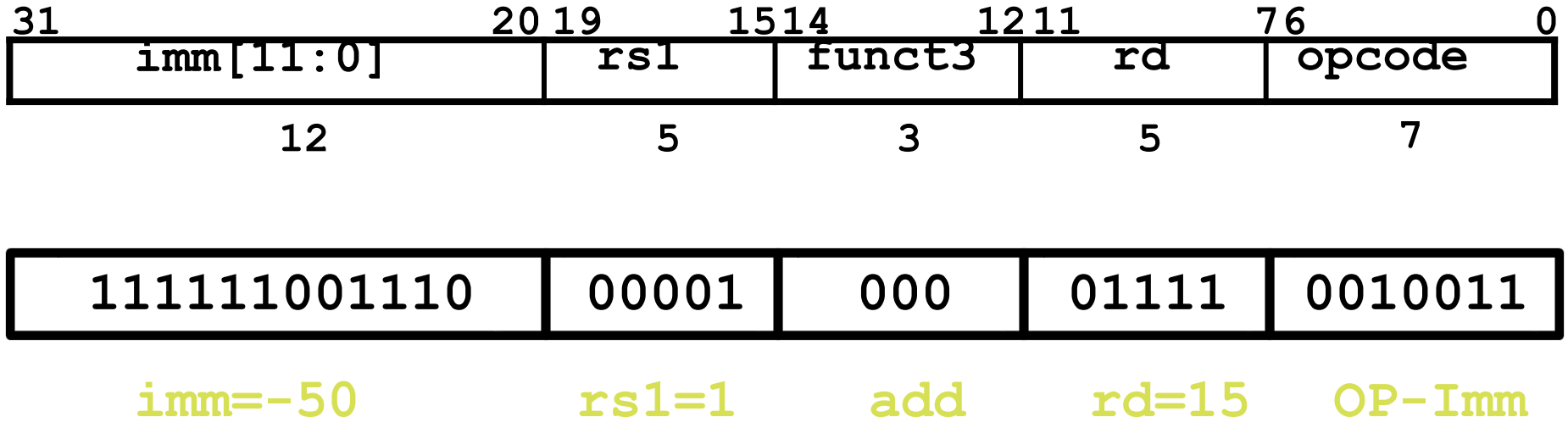
All implemented by decoding funct3 and funct7 fields and selecting appropriate ALU function

Datapath With Immediates

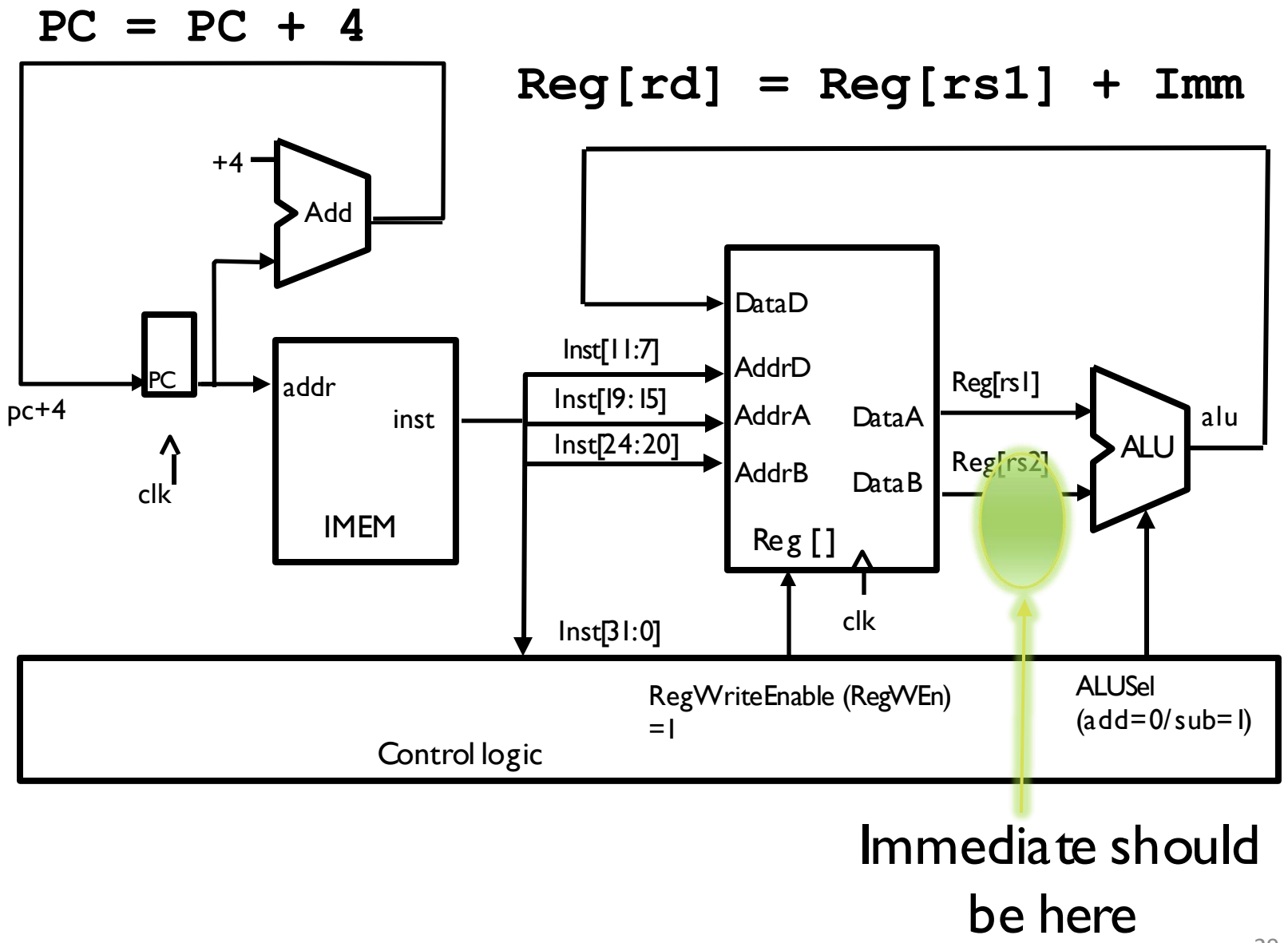
Implementing I-Format - addi instruction

- RISC-V Assembly Instruction:

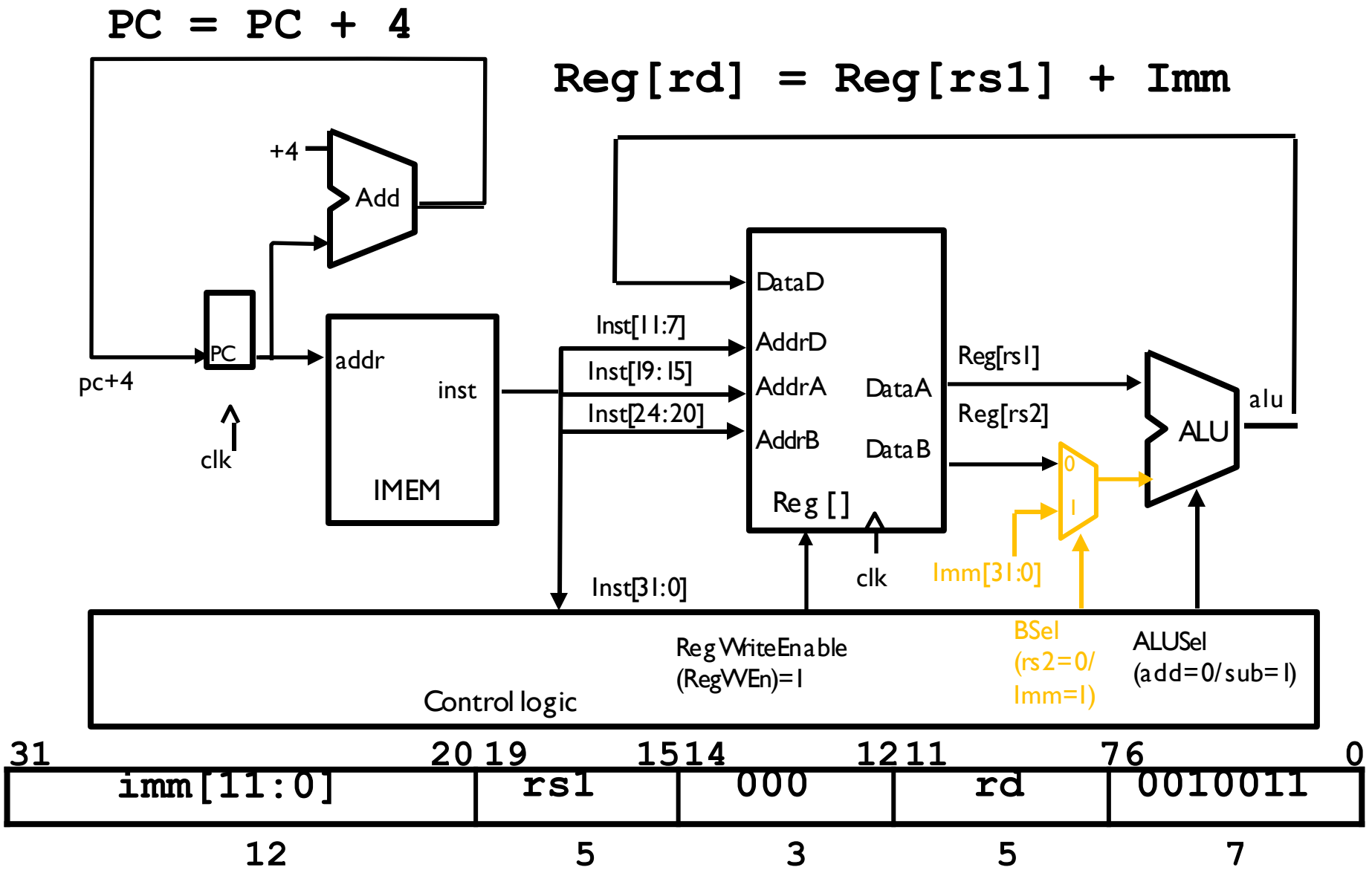
```
addi   x15,x1,-50
```



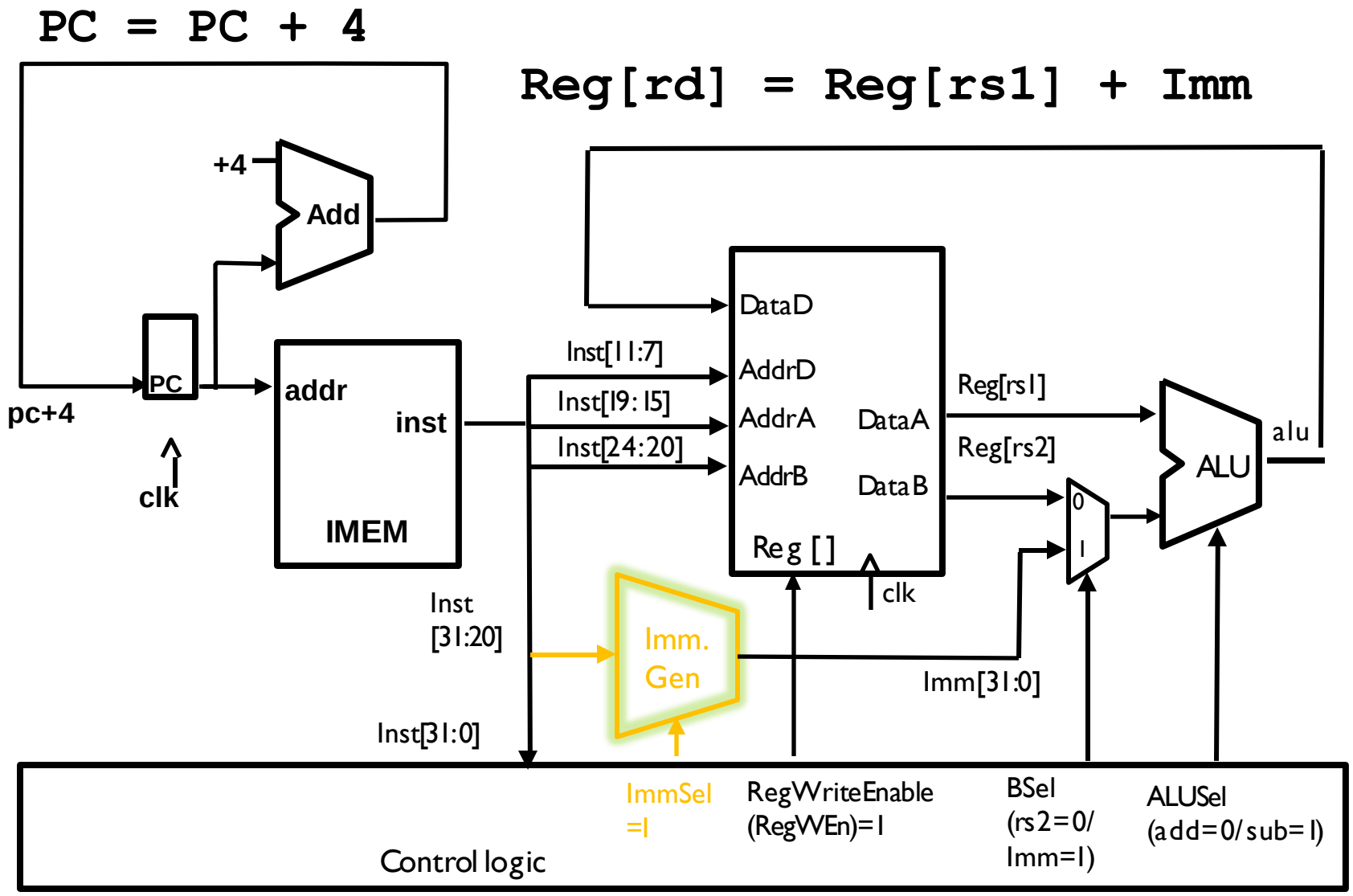
Datapath for add/sub



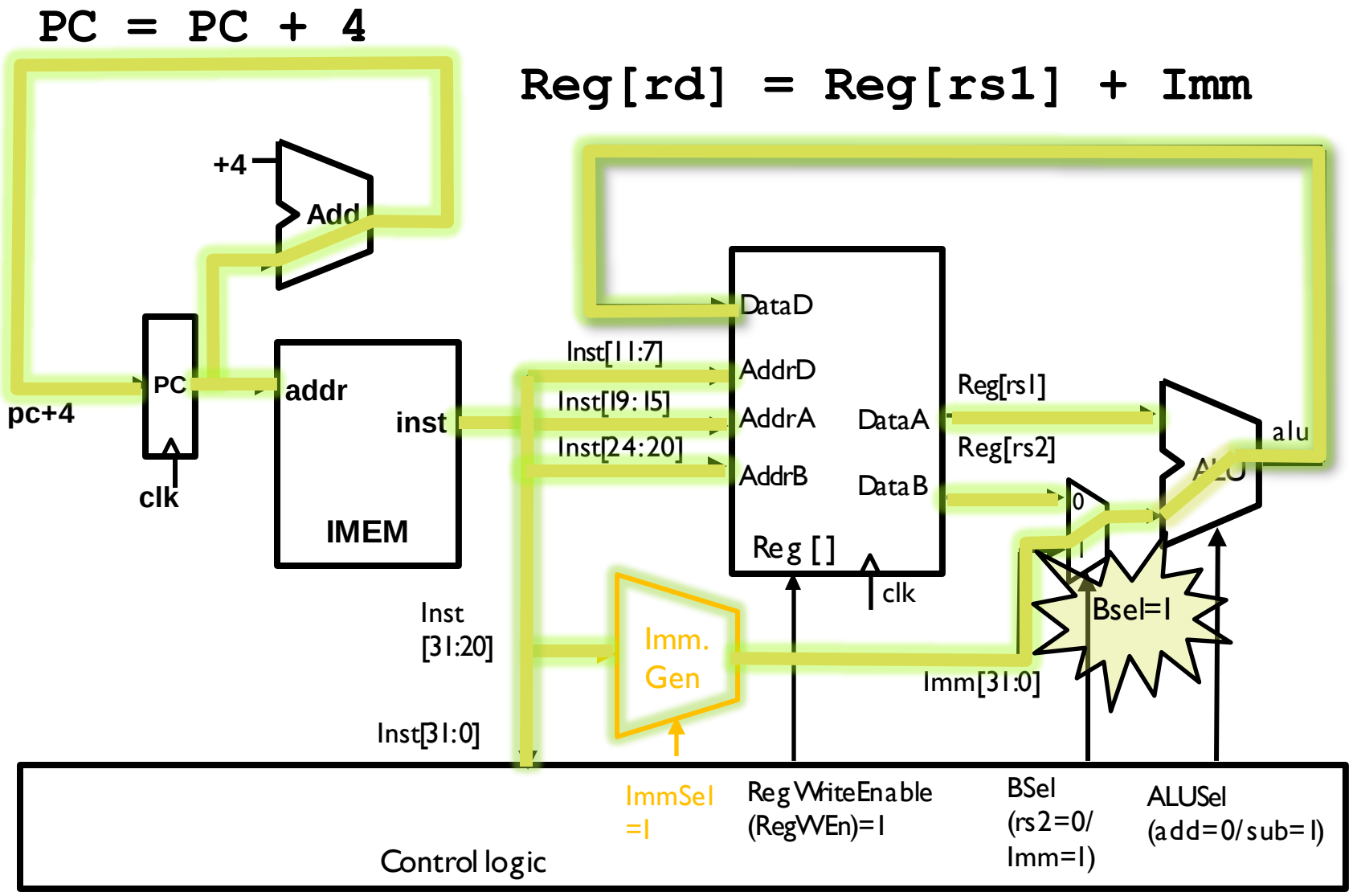
Adding **addi** to Datapath



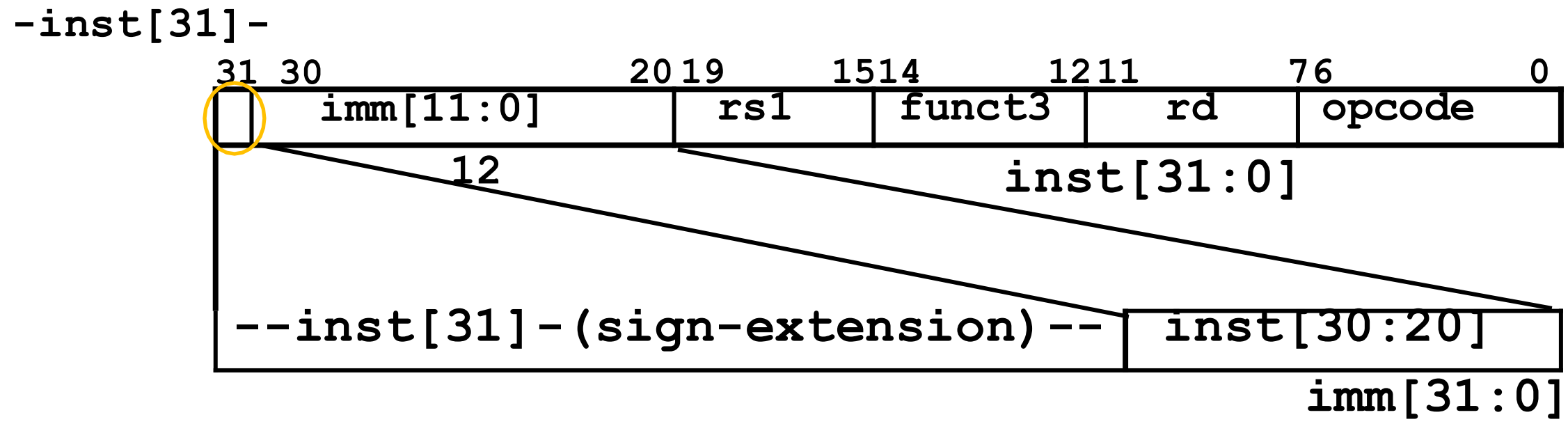
Adding **addi** to Datapath

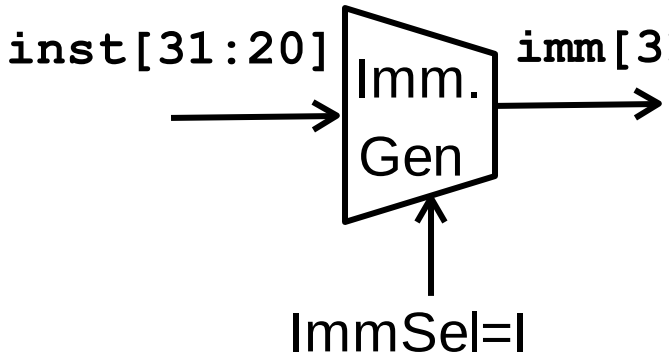


Adding **addi** to Datapath



I-Format Immediates

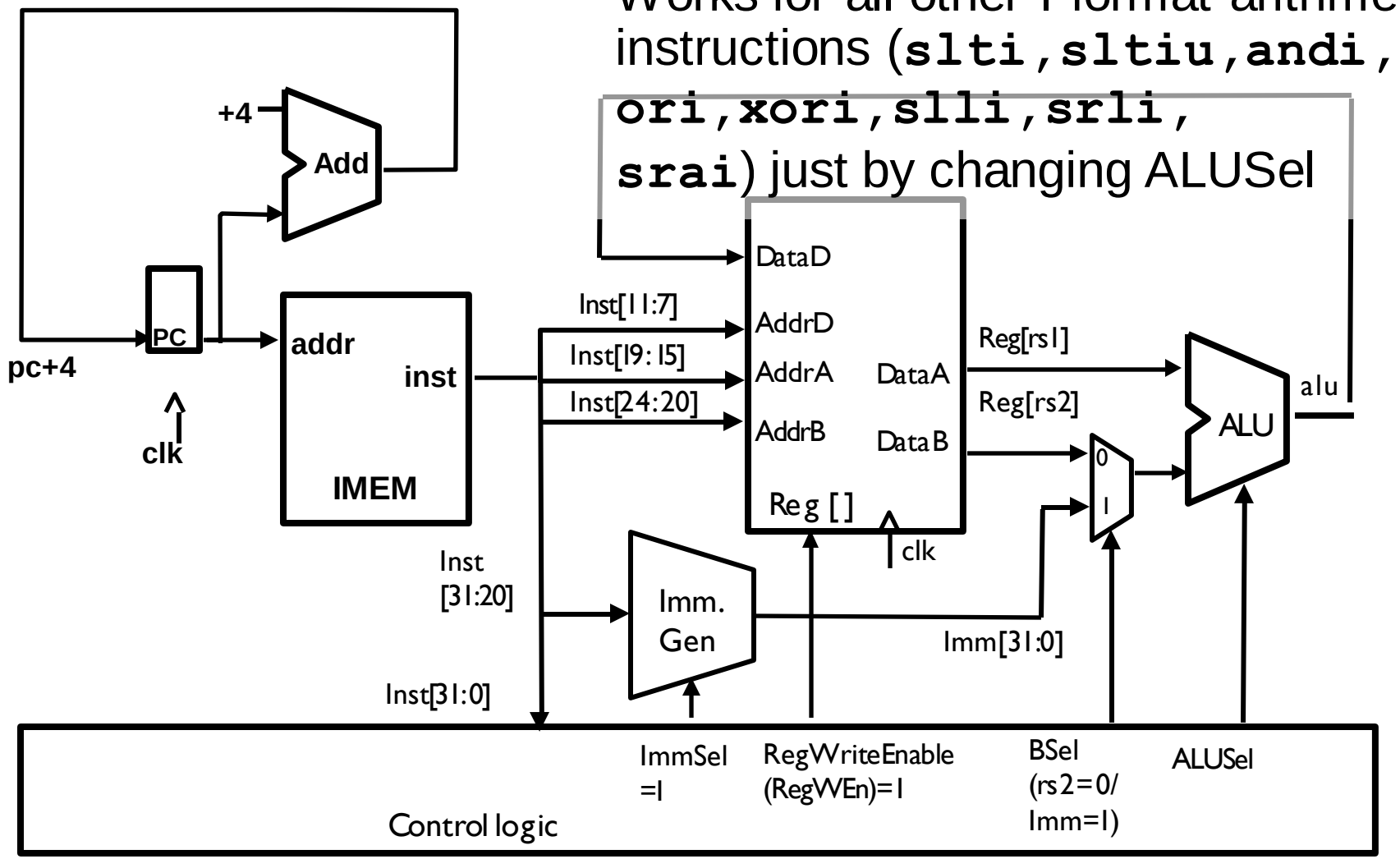




- High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
- Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])

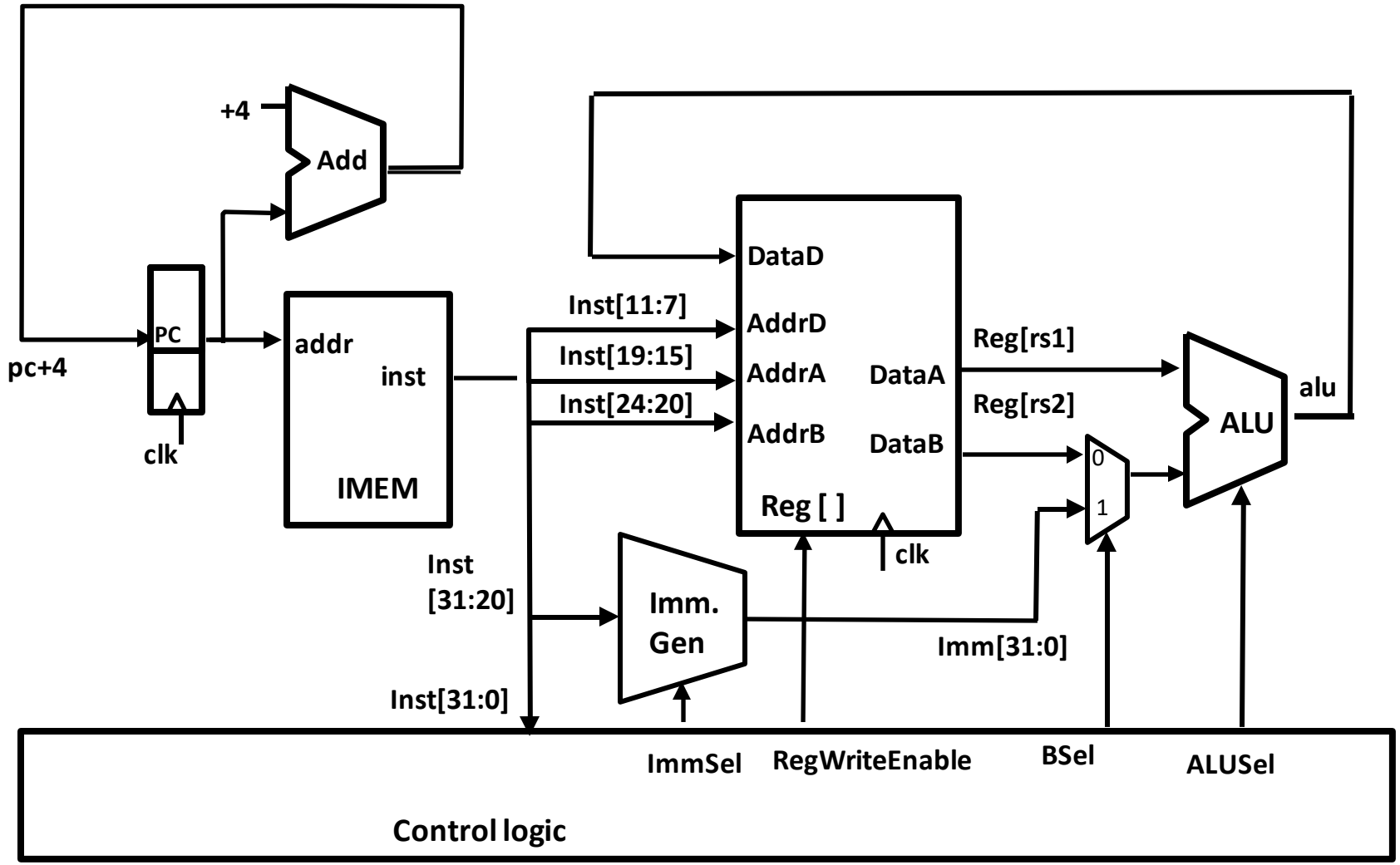
Adding **`addi`** to Datapath

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



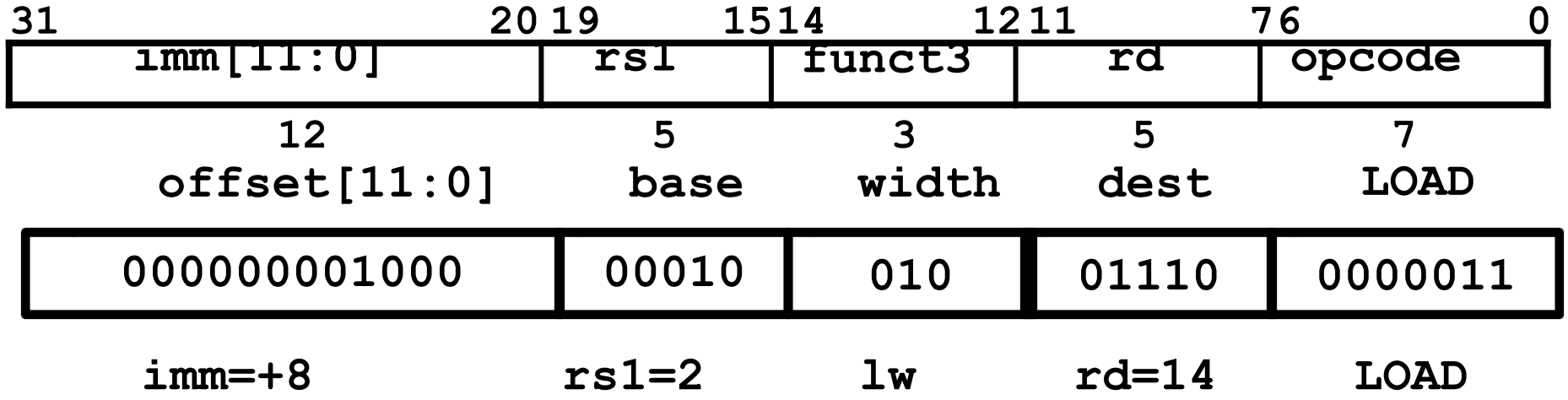
Supporting Loads

R+I Arithmetic/Logic Datapath



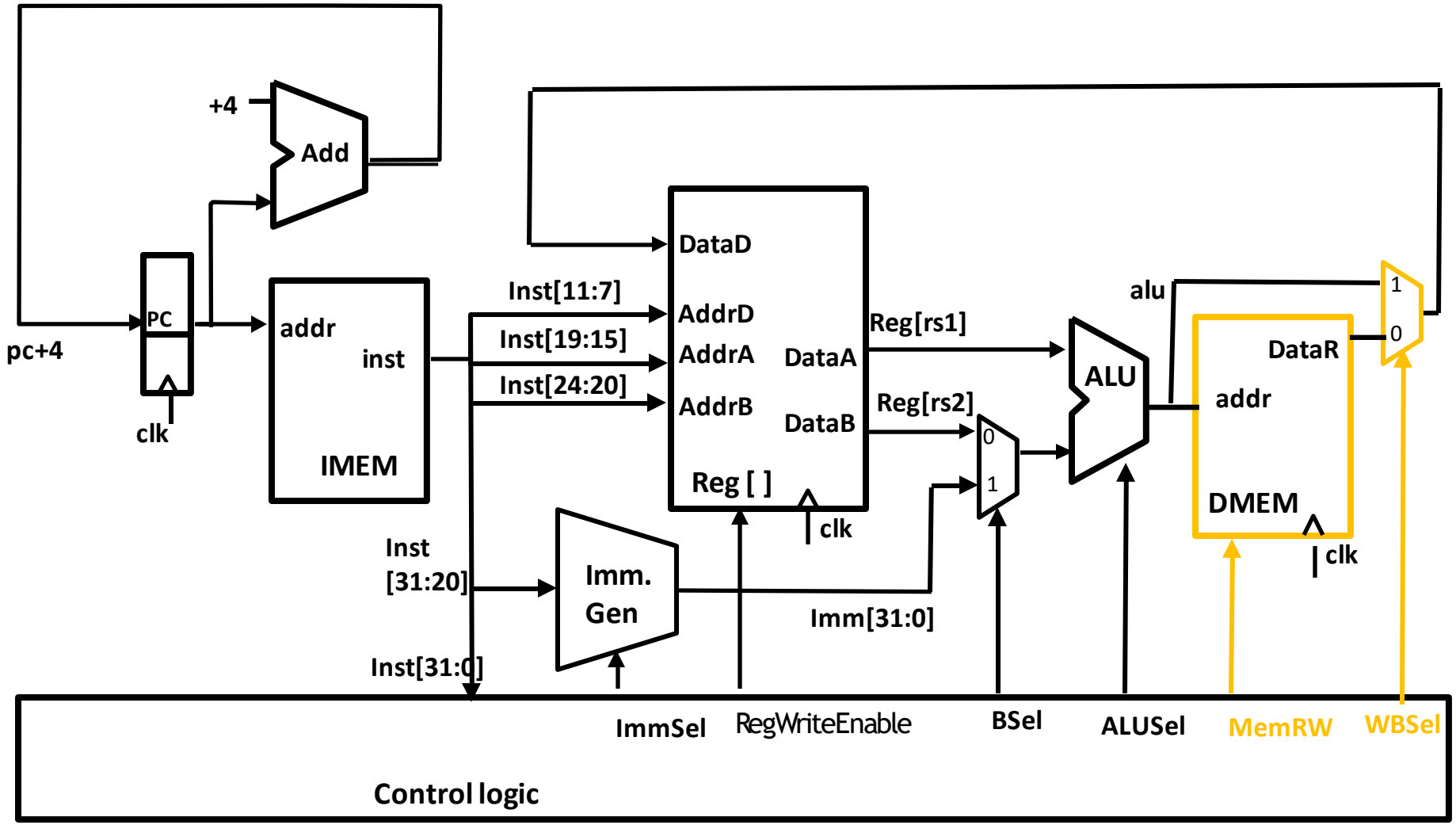
Add `lw`

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

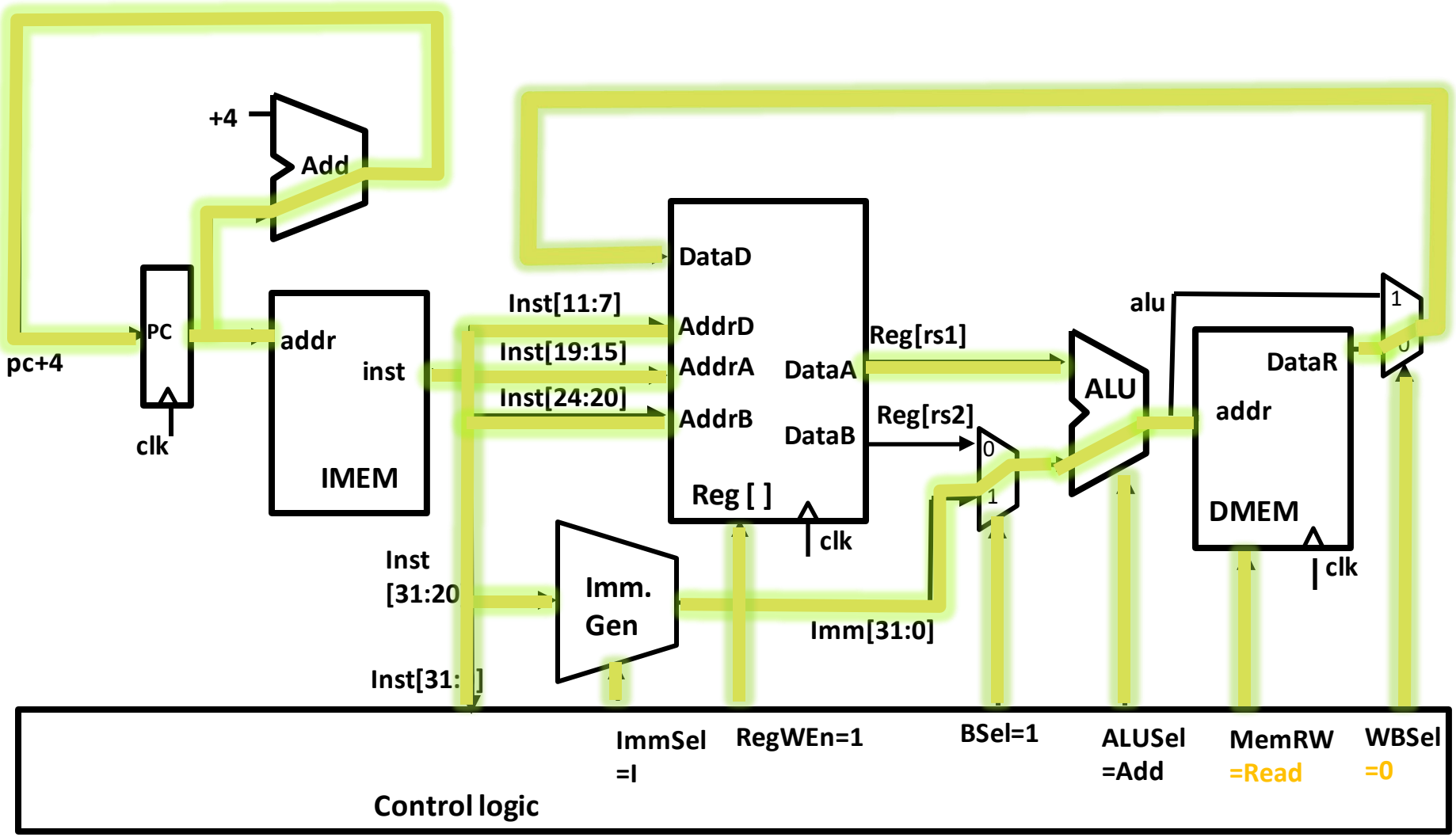


- The 12-bit signed immediate is added to the base address in register **`rs1`** to form the memory address
  - This is very similar to the add-immediate operation but used to create address not to create final result
- The value loaded from memory is stored in register **`rd`**

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

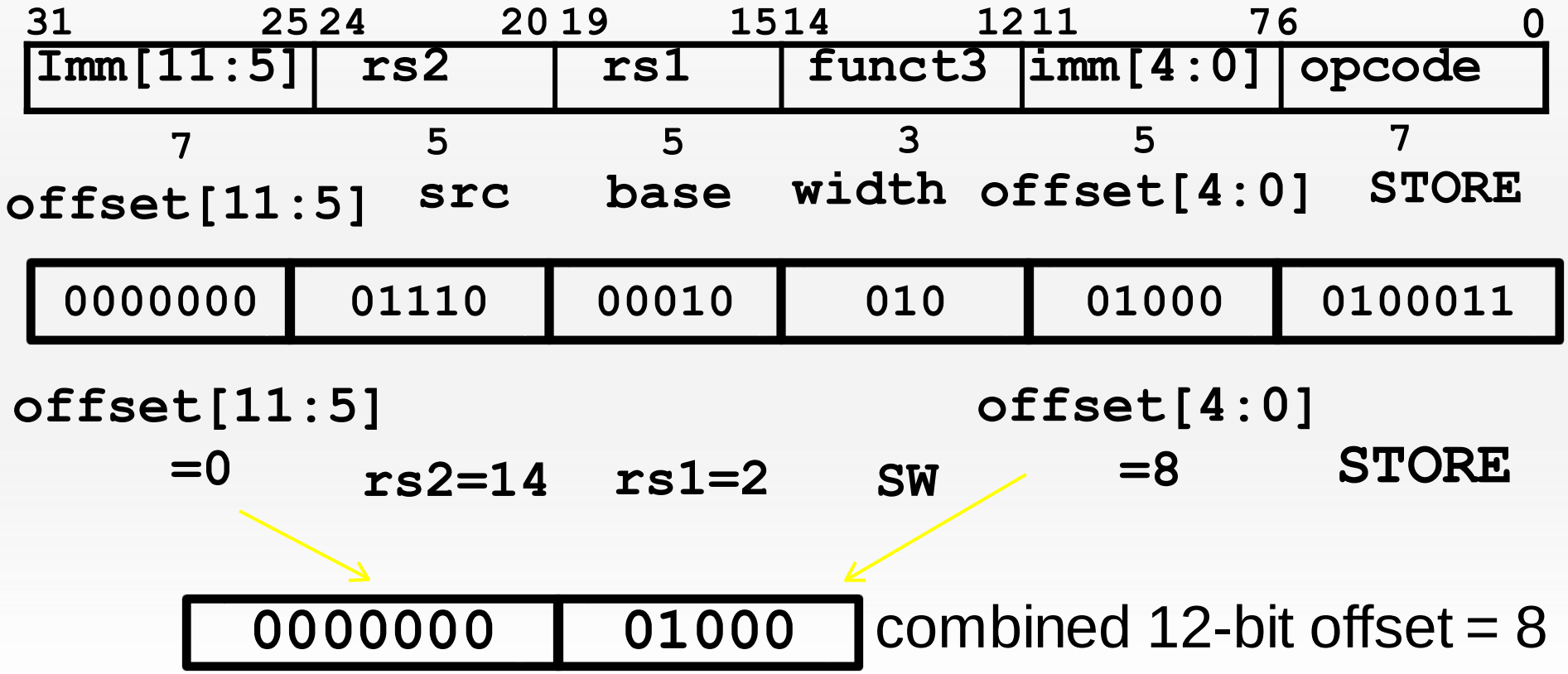
- Supporting the narrower loads requires additional logic to extract the correct byte/halfword from the value loaded from memory, and sign- or zero-extend the result to 32 bits before writing back to register file.
  - It is just a mux + a few gates

Datapath for Stores

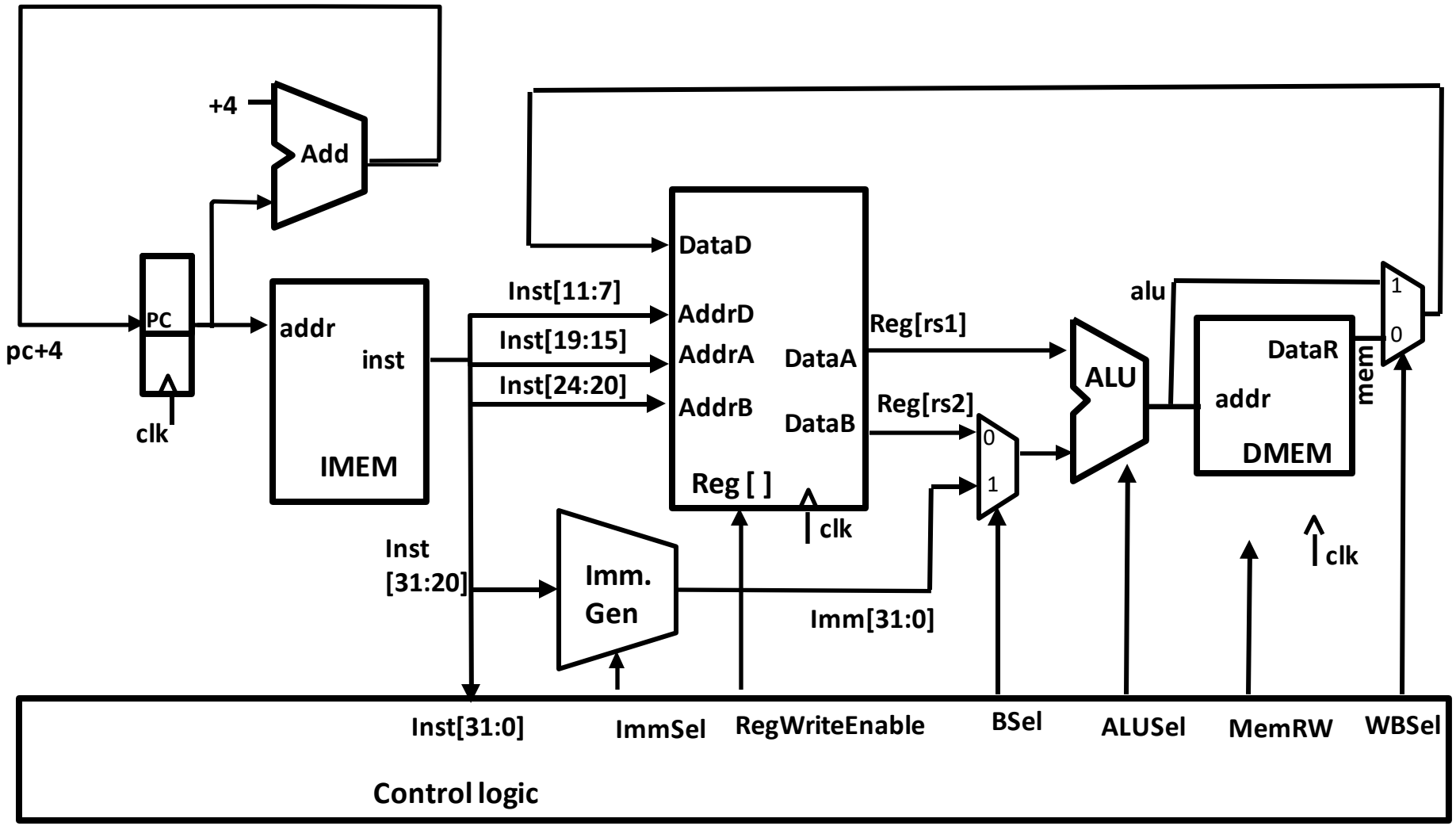
Adding **sw** instruction

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

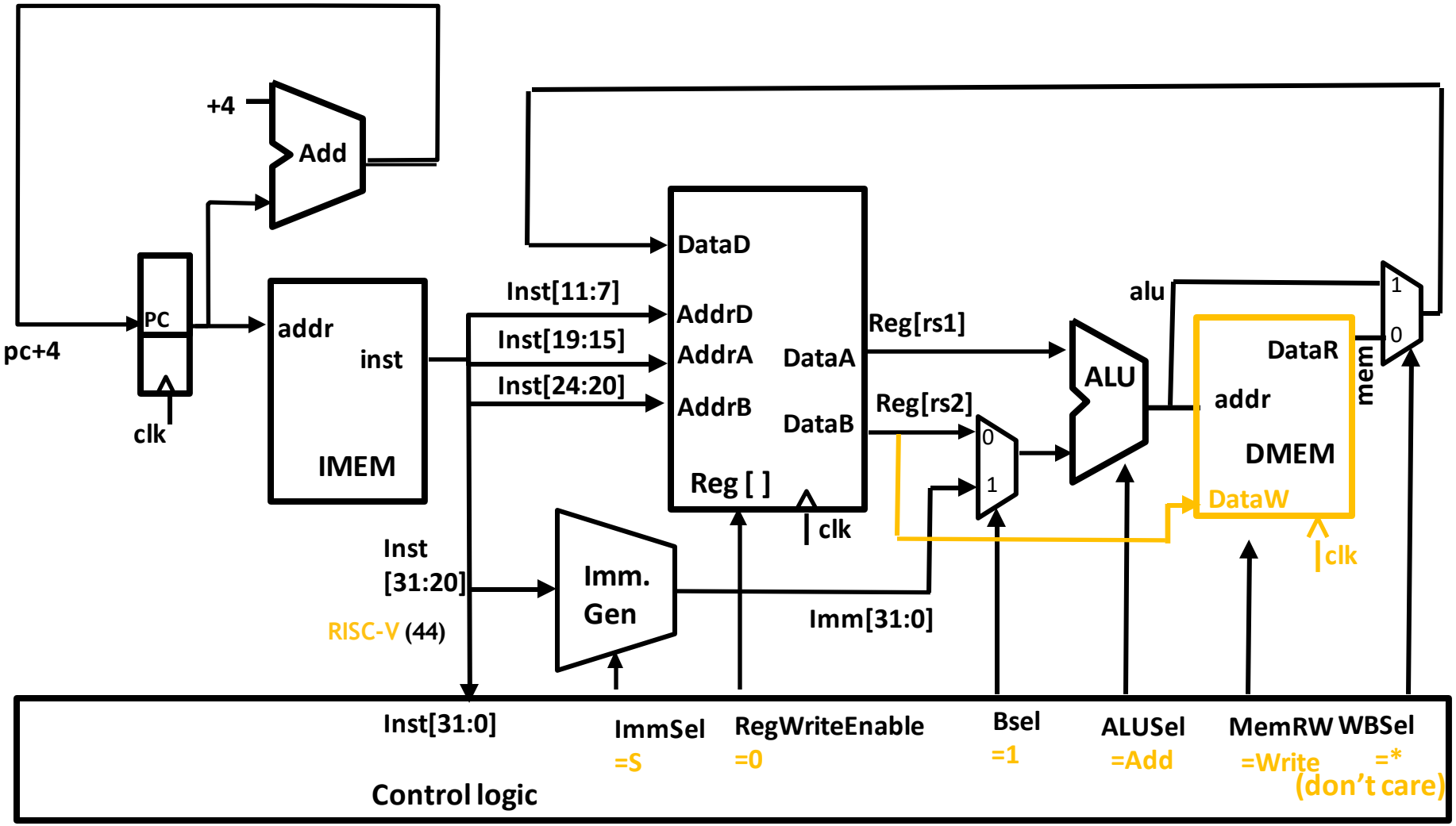
```
sw x14, 8(x2)
```



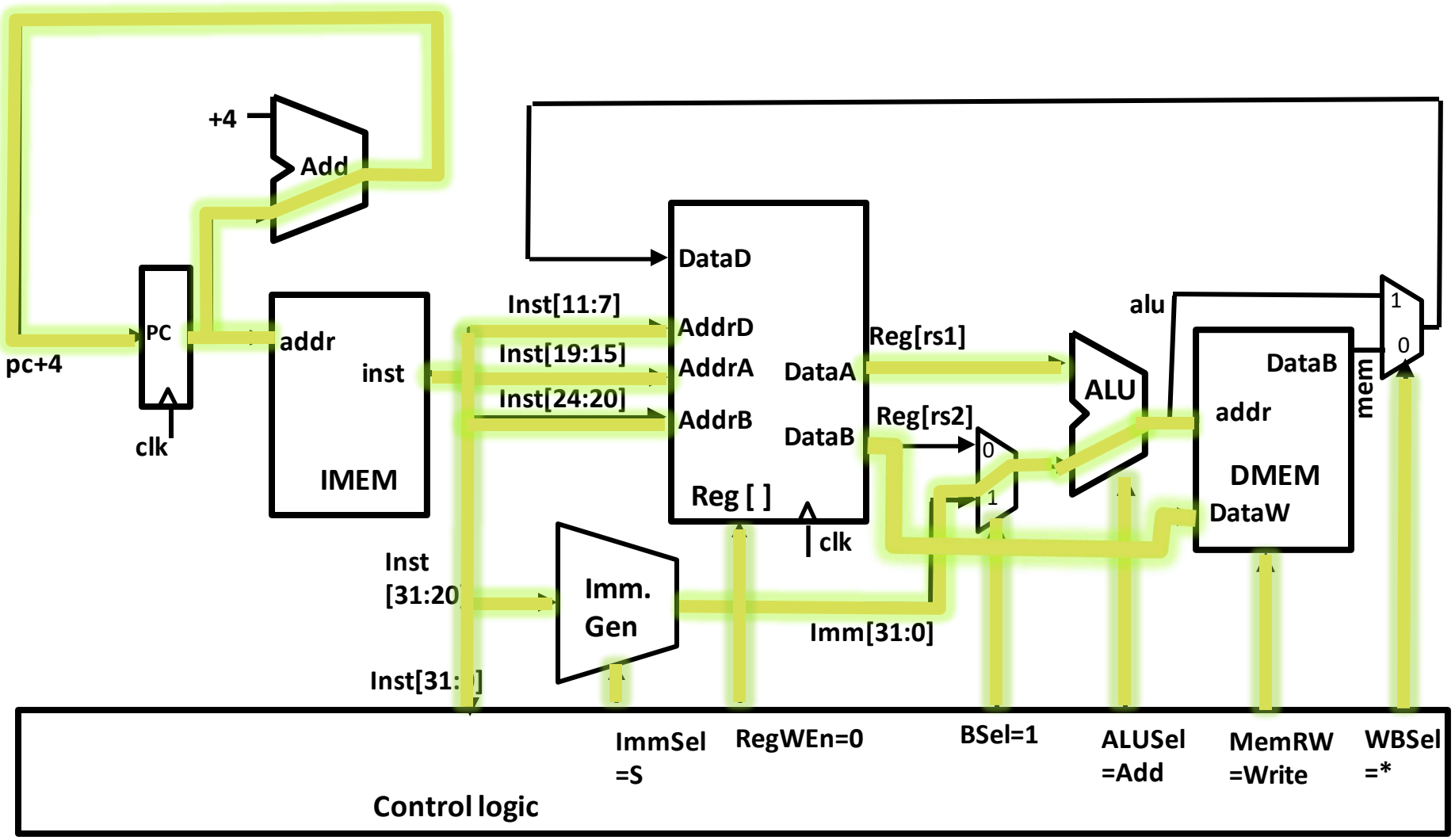
Datapath with `lw`



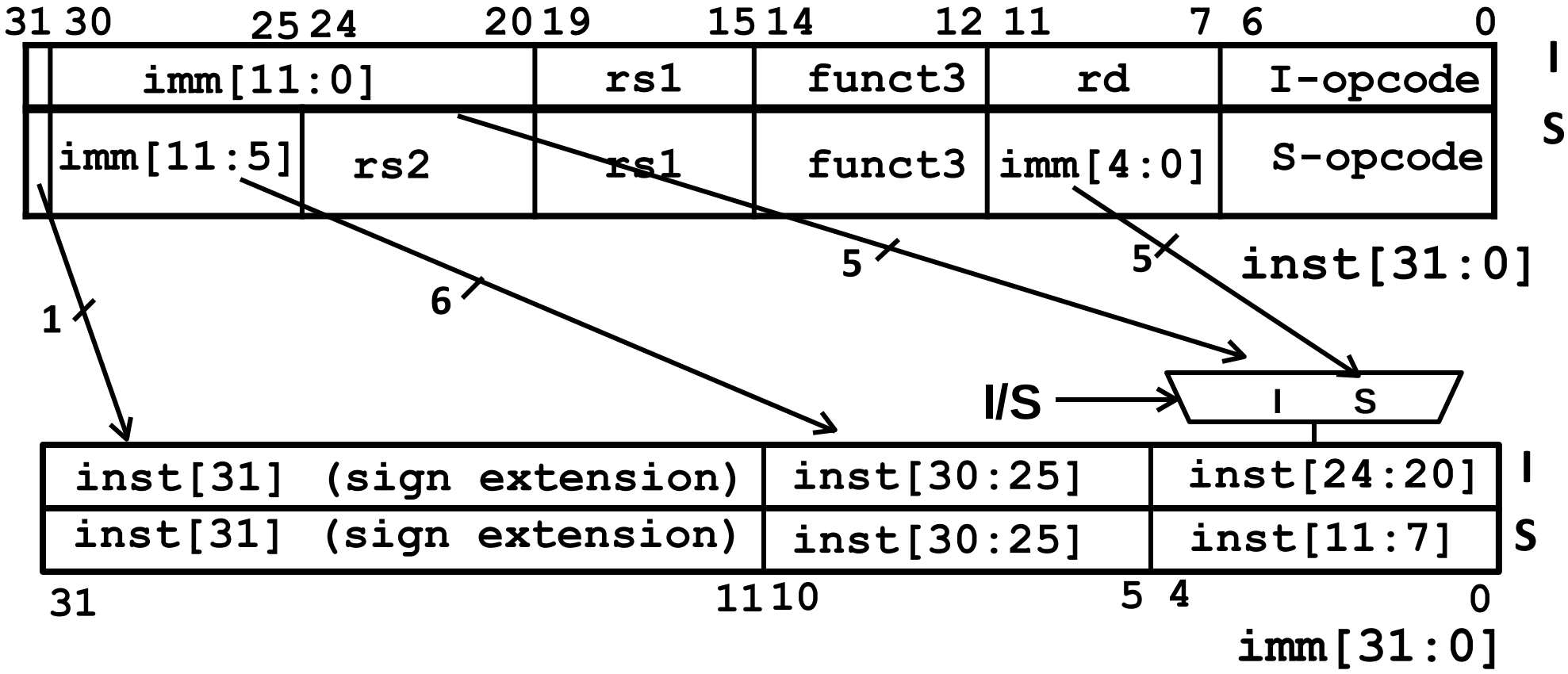
Adding **sw** to Datapath



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

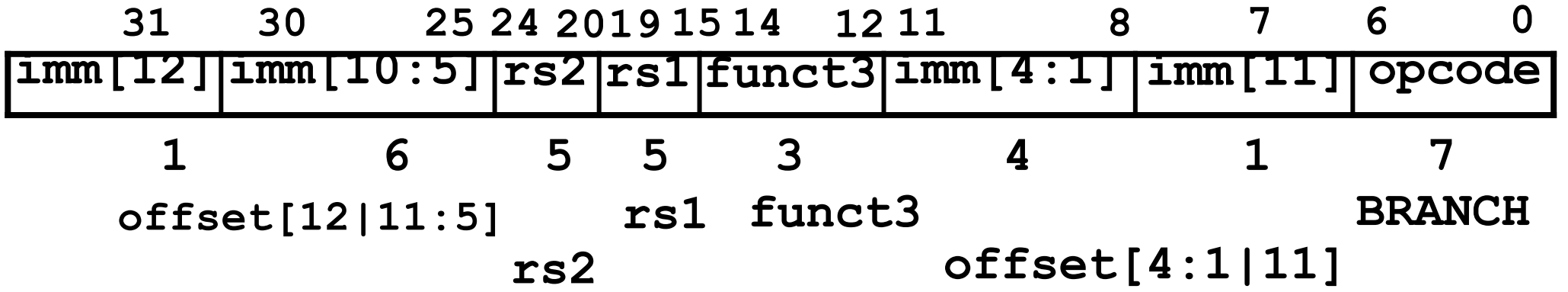
- Store byte writes the low byte to memory
- Store halfword writes the lower two bytes to memory

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

width

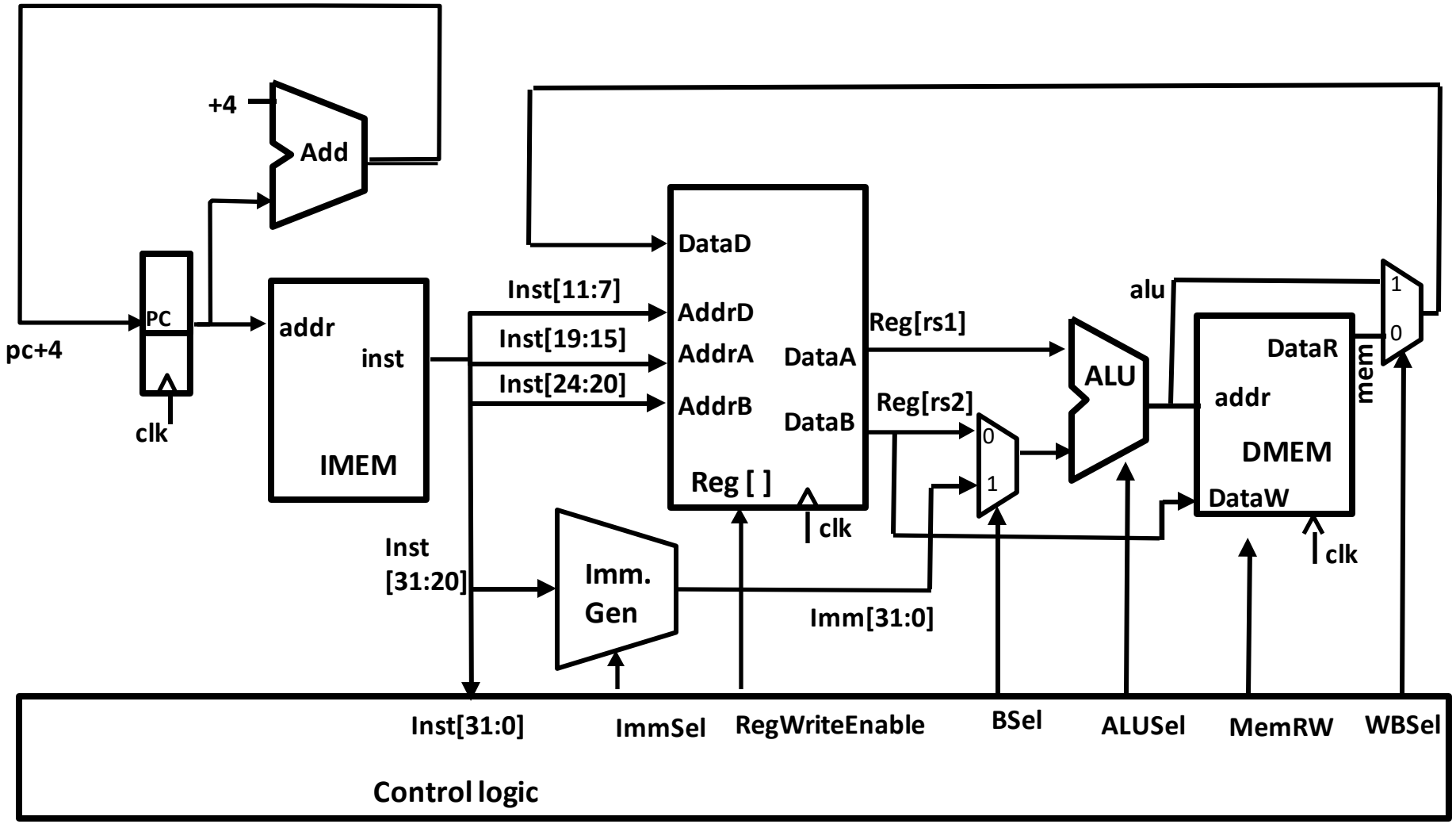
Implementing Branches

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

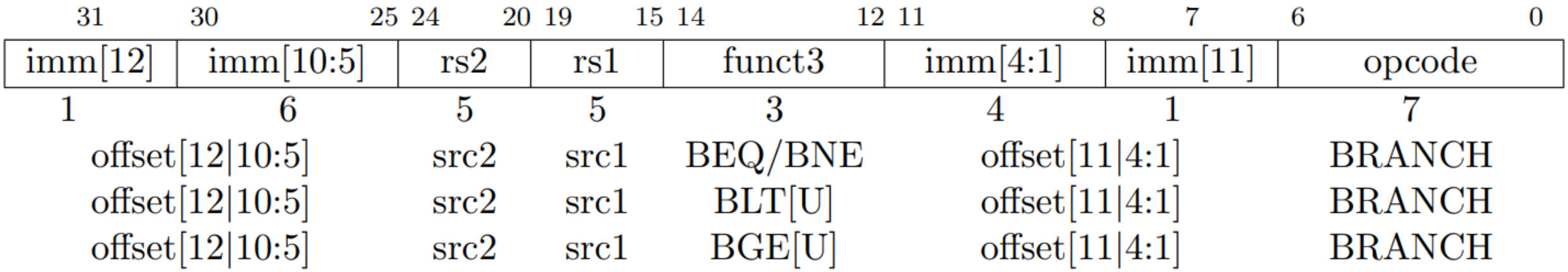


To Add Branches

- Different change to the state:

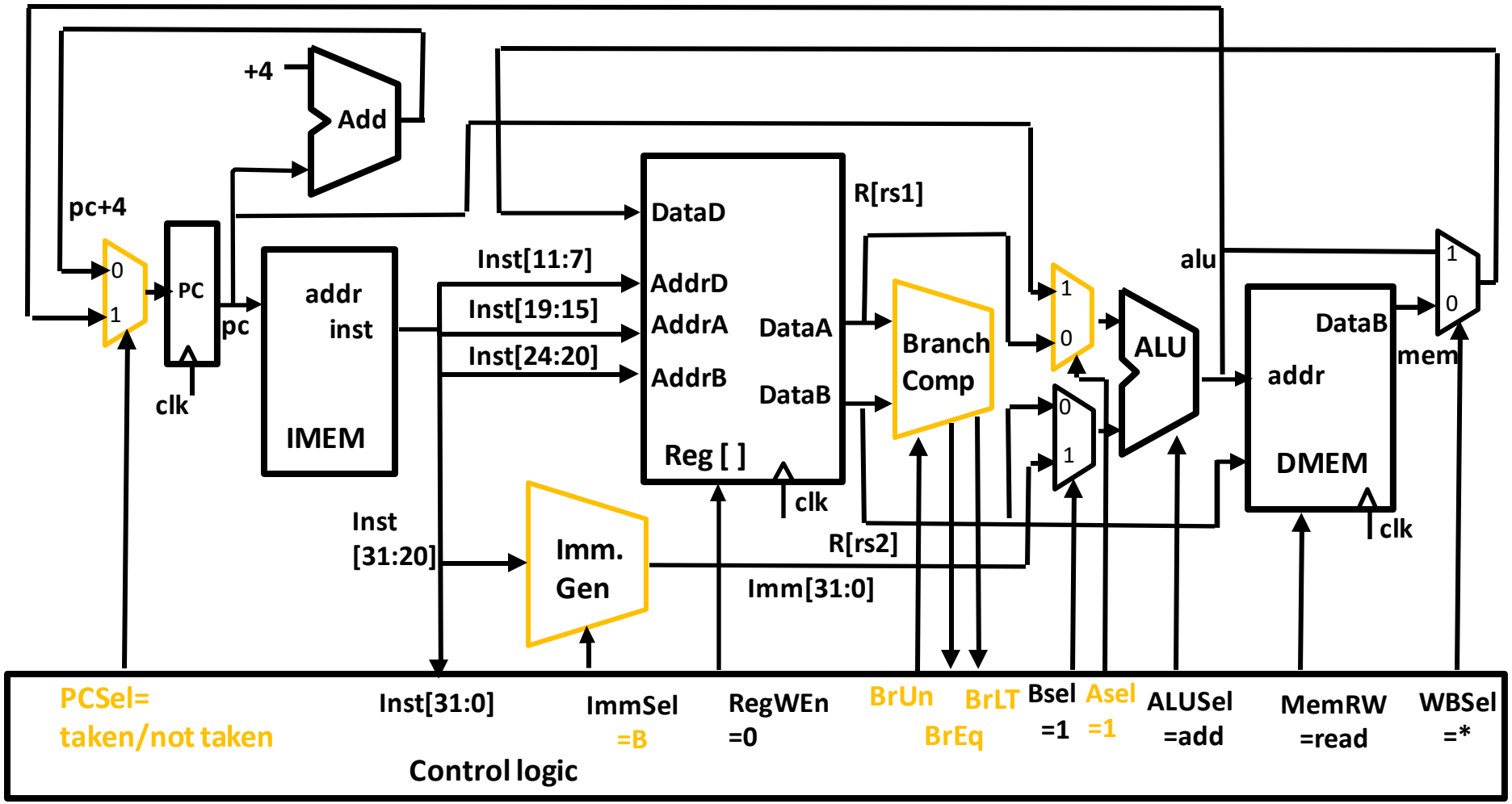
$$\texttt{PC} = \begin{cases} \texttt{PC + 4}, & \textbf{branch not taken} \\ \texttt{PC + immediate}, & \textbf{branch taken} \end{cases}$$

- 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

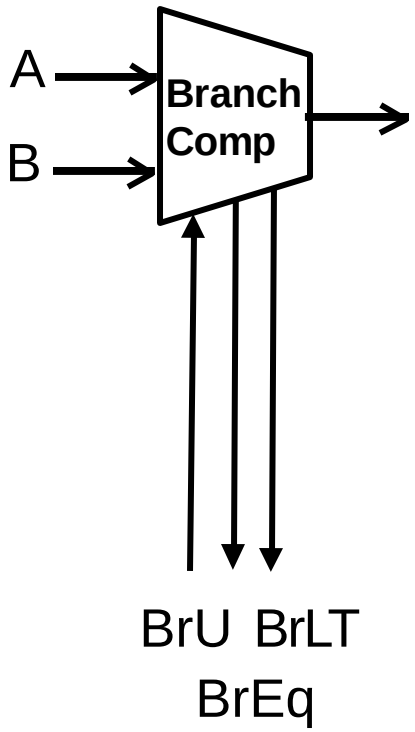


31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

Adding Branches



Branch Comparator



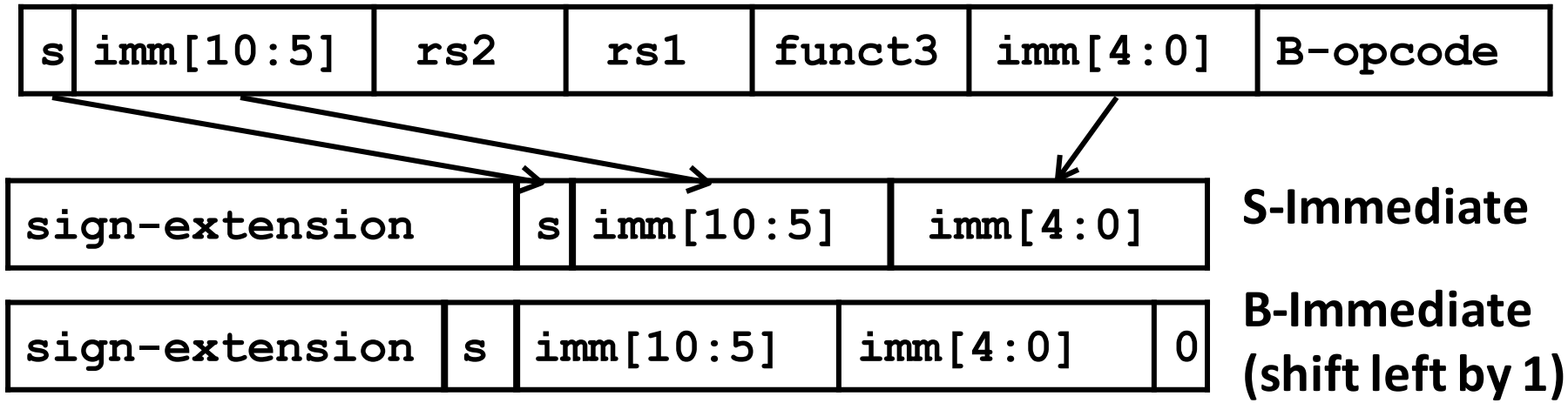
- **`BrEq = 1`**, if A=B
- **`BrLT = 1`**, if A<B
- **`BrUn`** = 1 selects unsigned comparison for **`BrLT`**, 0=signed

`BGE` branch: A >= B, if $\overline{A<B}$

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

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



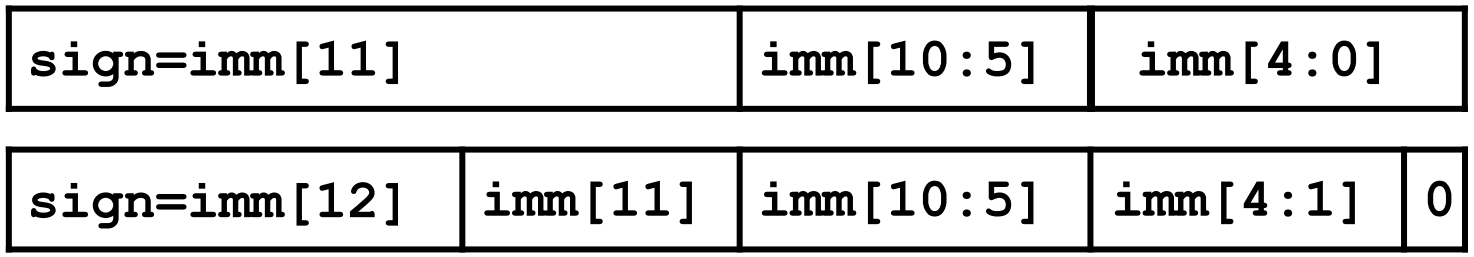
Each instruction immediate bit can appear in one of two places in output immediate value – so need one 2-way mux per bit

Branch Immediates (In RISC -V ISAs)

- 12-bit immediate encodes PC-relative offset of -4096 to +4094 bytes in multiples of 2 bytes

NOTE : Page. 116 (RISC V Textbook -> get extra information)

- 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-Immediate

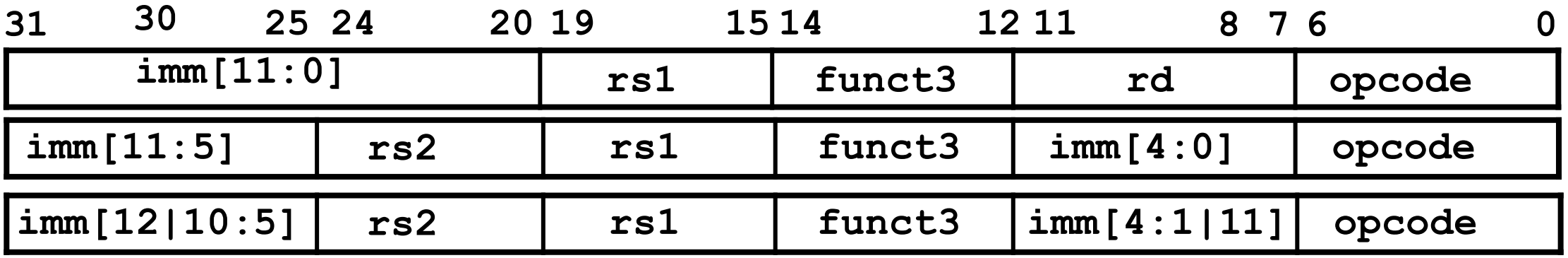
B-Immediate (shift left by 1)

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

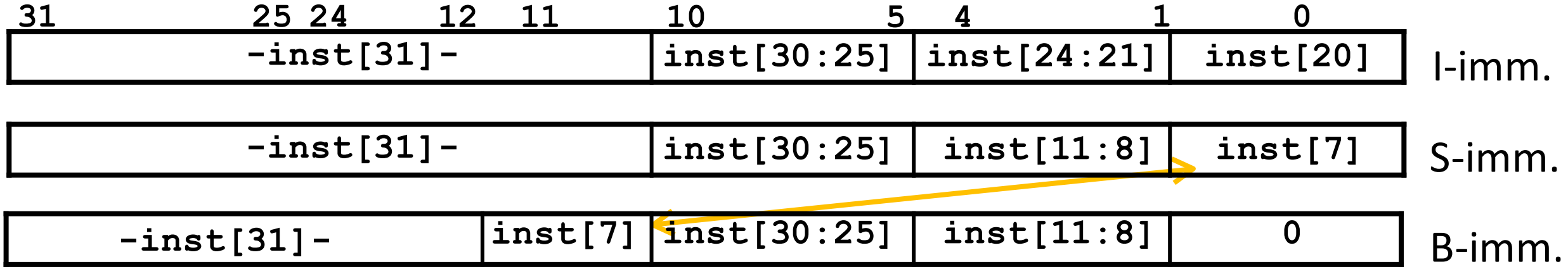
The RISC-V architects wanted to support the possibility of instructions that are only 2 bytes long, so the branch instructions represent the number of *halfwords* between the branch and the branch target

RISC-V Immediate Encoding

Instruction encodings, inst[31:0]



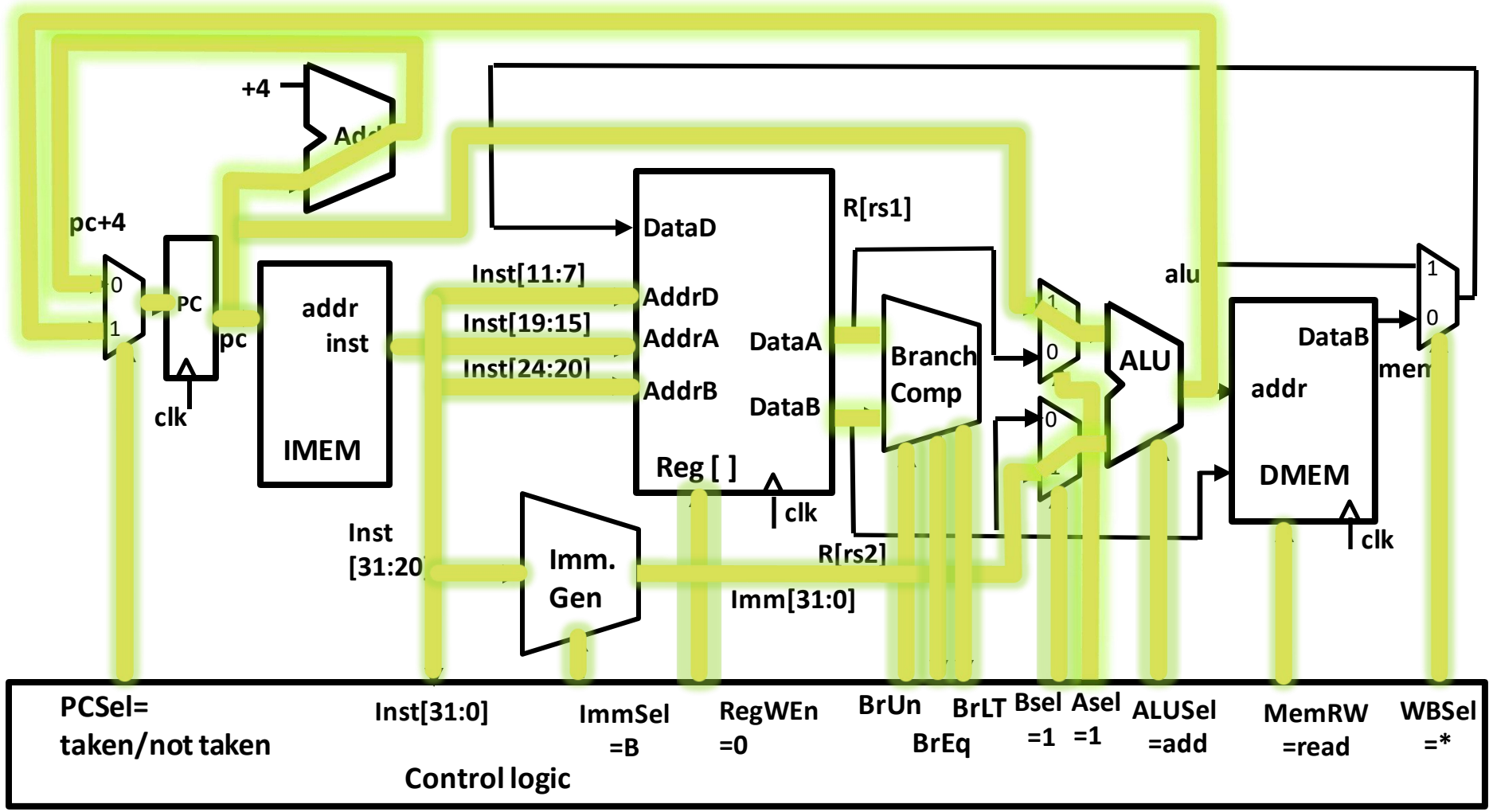
32-bit immediates produced, imm[31:0]



Upper bits sign-extended from `inst[31]` always

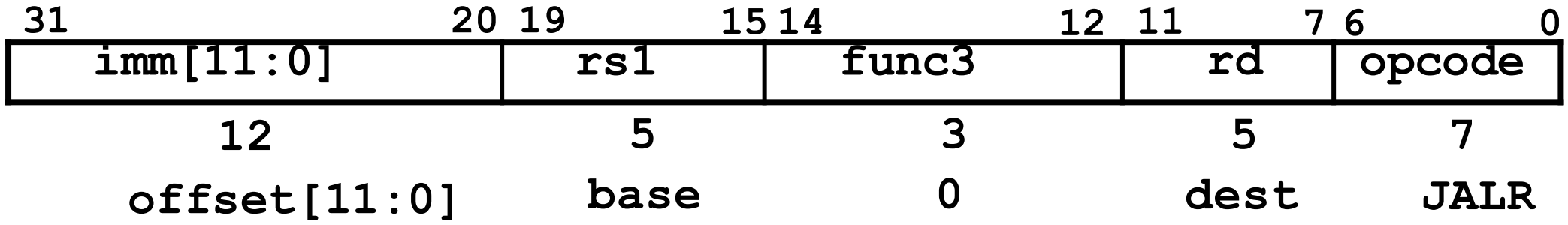
Only bit 7 of instruction changes role in immediate between S and B

Lighting Up Branch Path



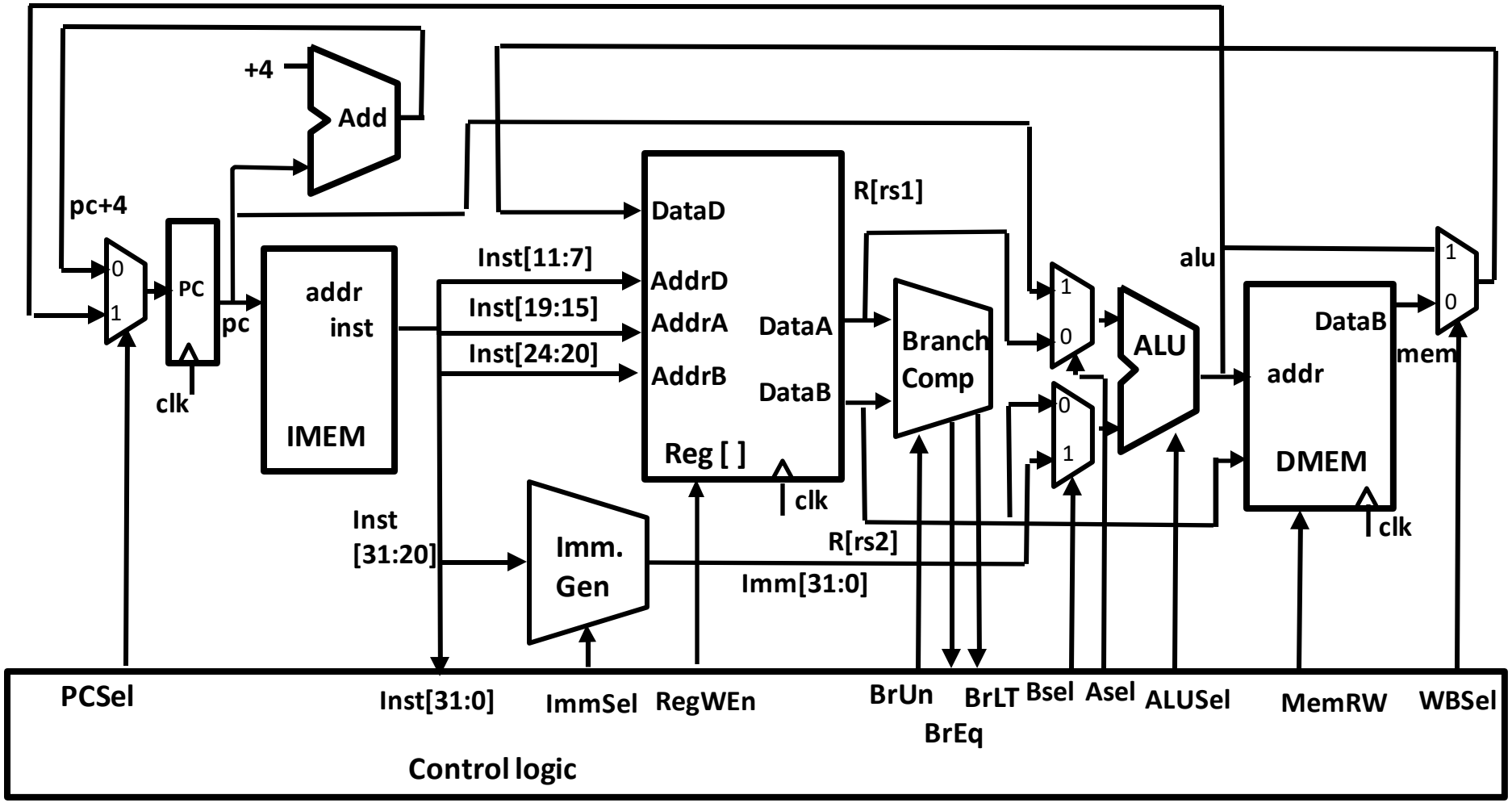
Adding JALR to Datapath

Let's Add **JALR** (I-Format)

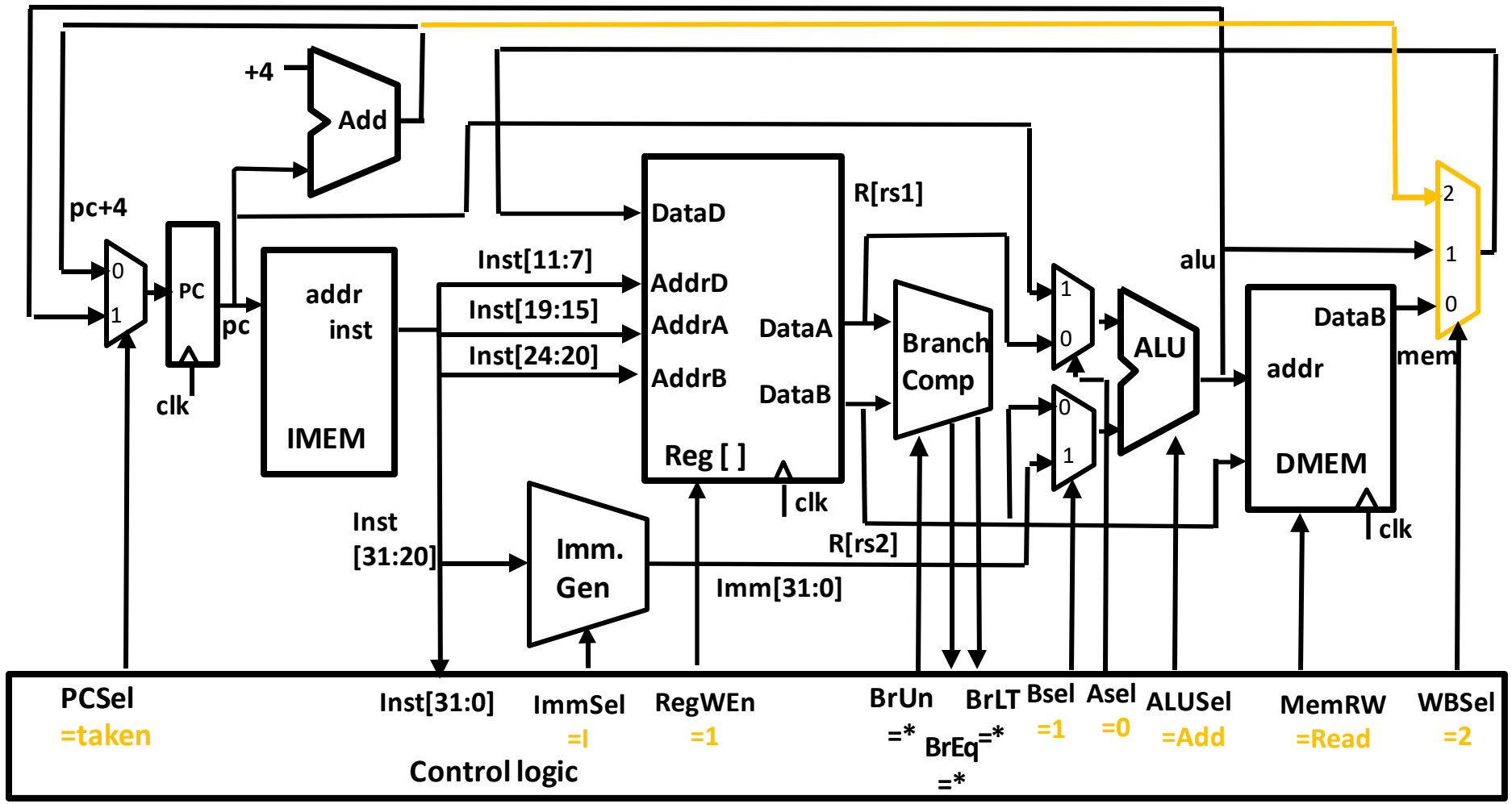


- **`JALR rd, rs, immediate`**
- Two changes to the state
  - Writes PC+4 to **`rd`** (return address)
  - Sets PC = **`rs1 + immediate`**
  - Uses same immediates as arithmetic and loads
    - ***no*** multiplication by 2 bytes
    - LSB is ignored

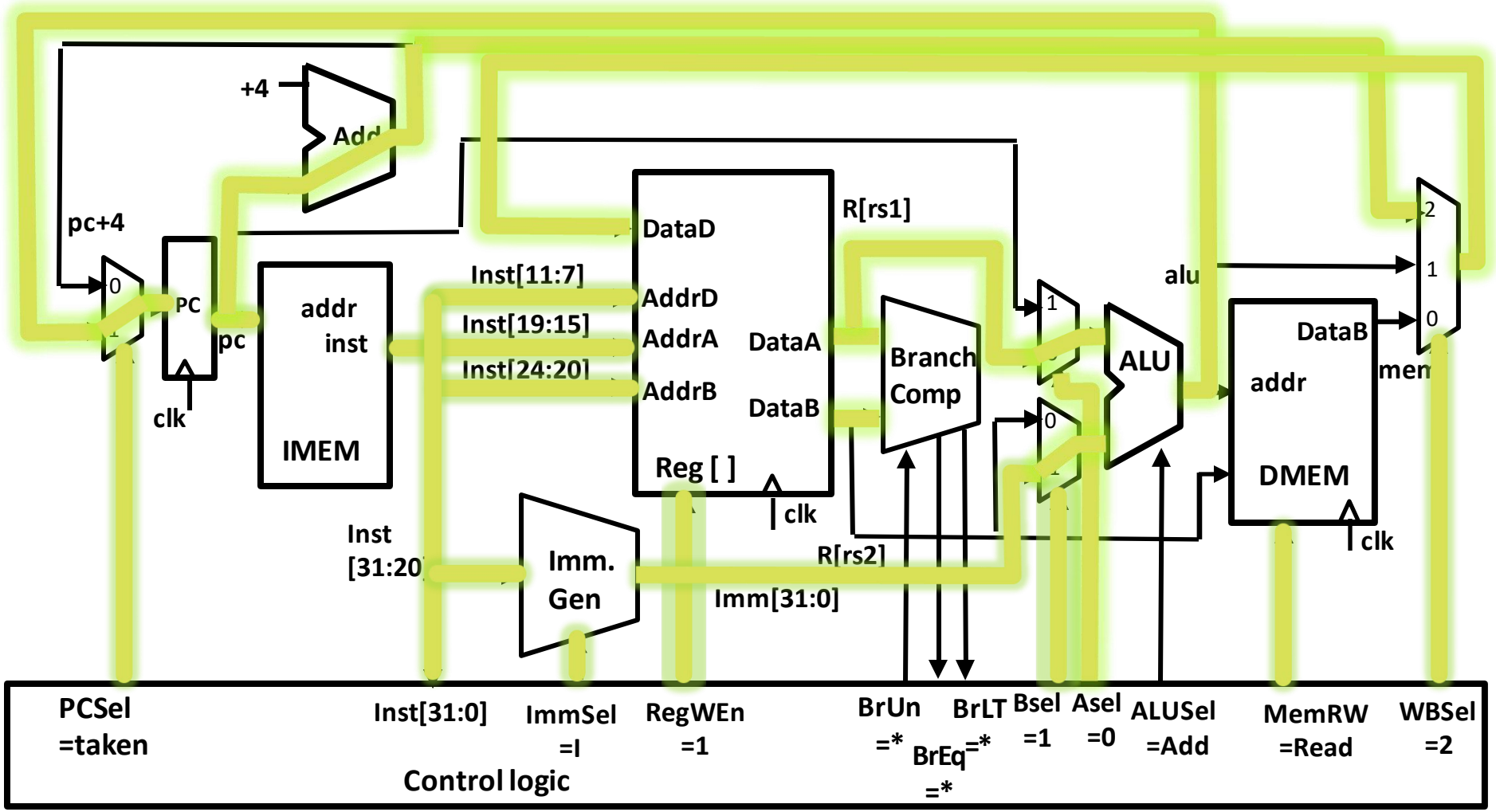
Datapath So Far, With Branches



Datapath With **JALR**

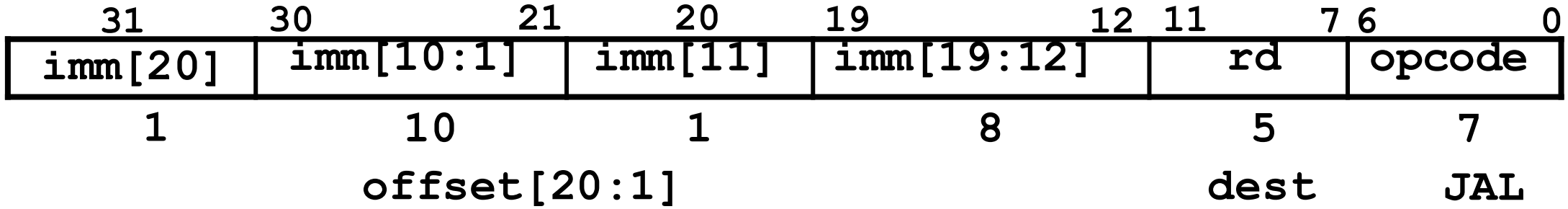


Datapath With **JALR**



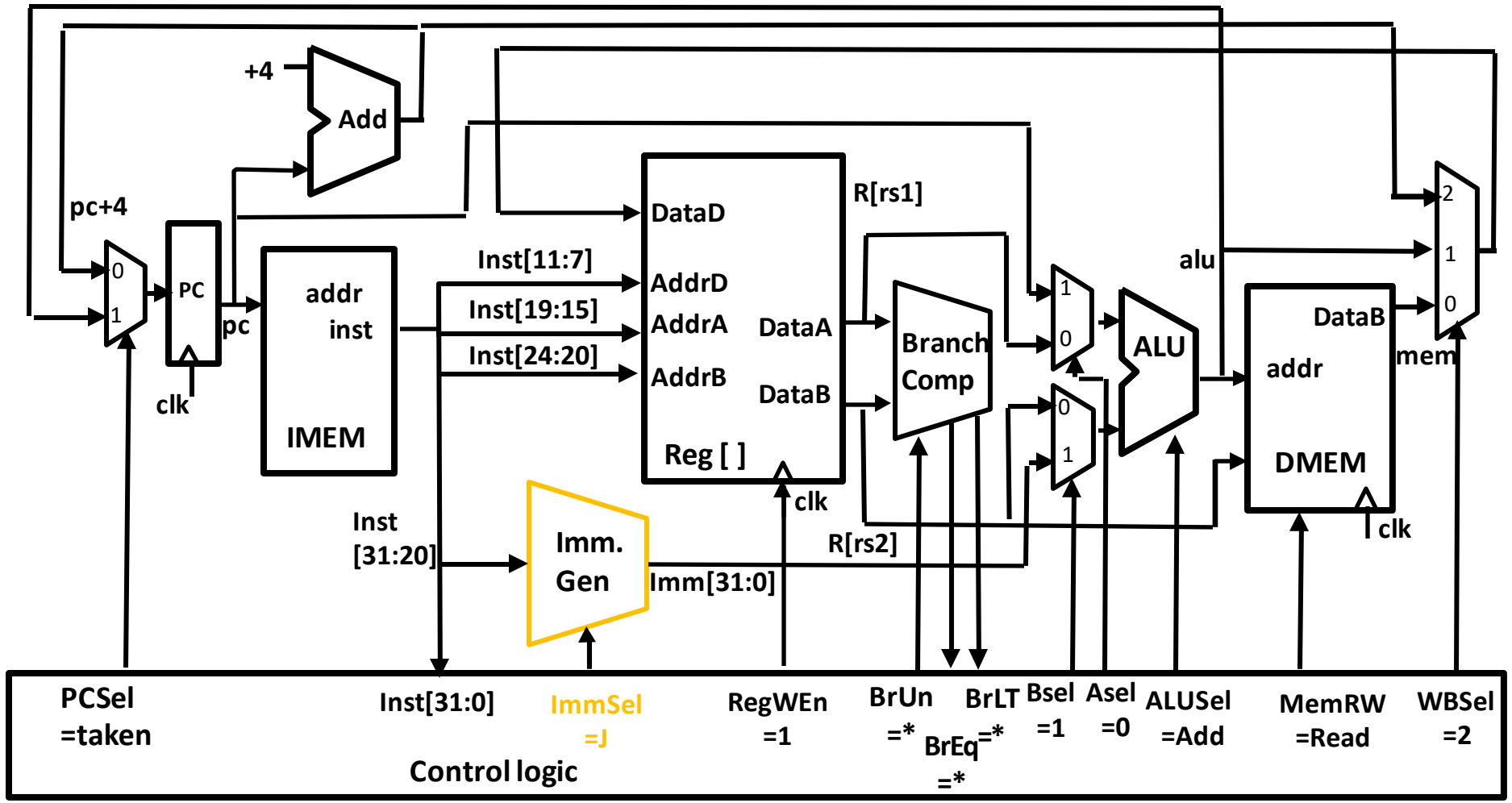
Adding JAL

J-Format for Jump Instructions

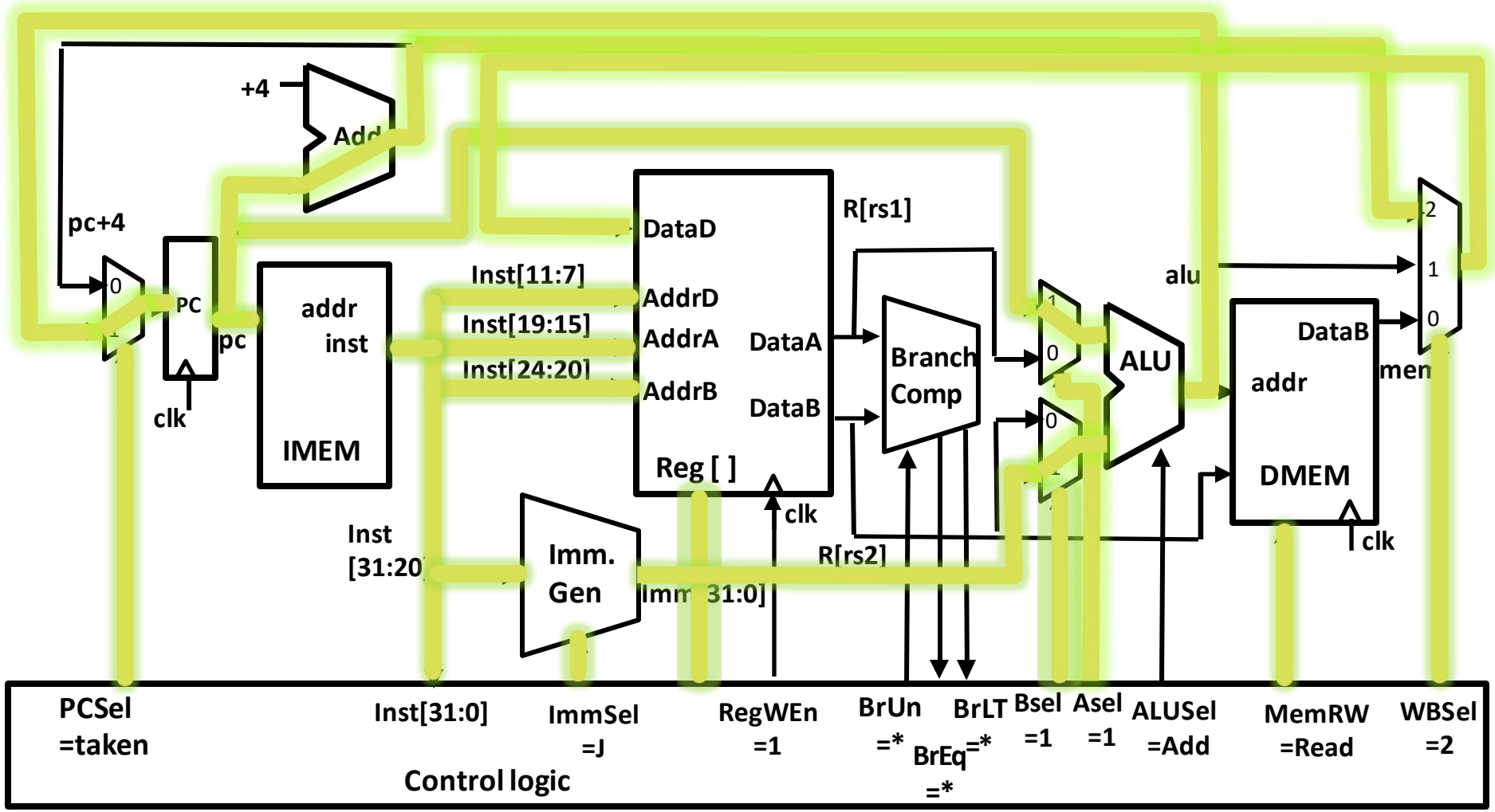


- Two changes to the state
  - **`jal`** saves PC+4 in register **`rd`** (the return address)
  - Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction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

Datapath with JAL

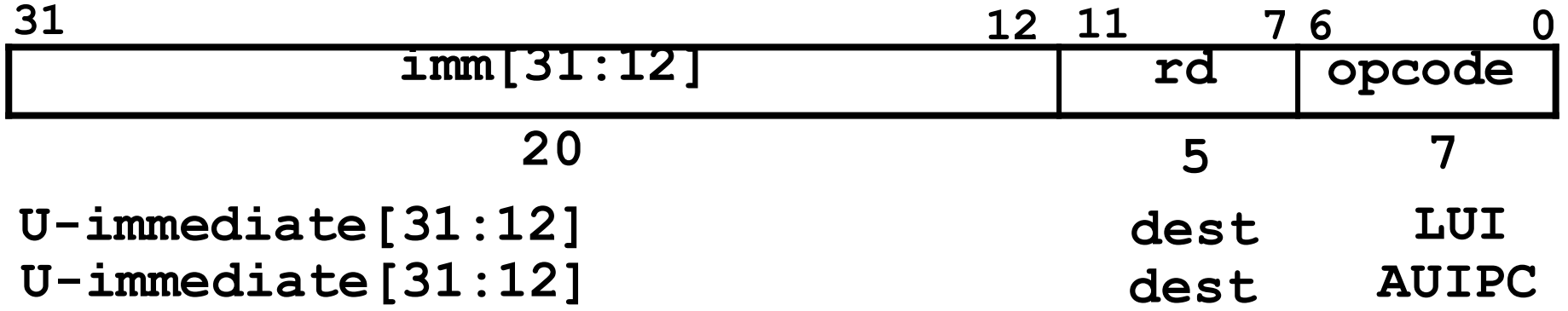


Light Up JAL Path



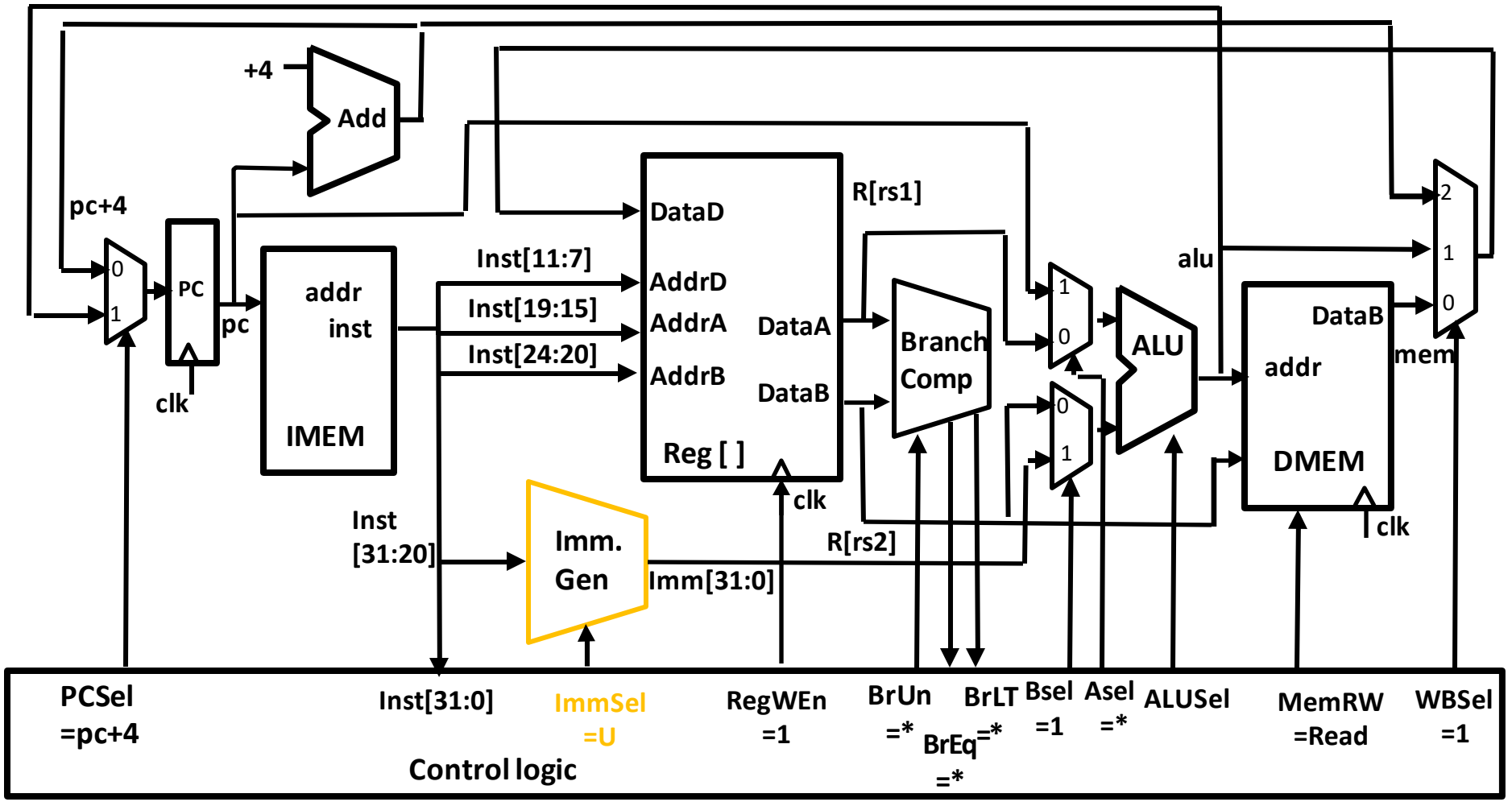
Adding U-Types

U-Format for "Upper Immediate" Instructions

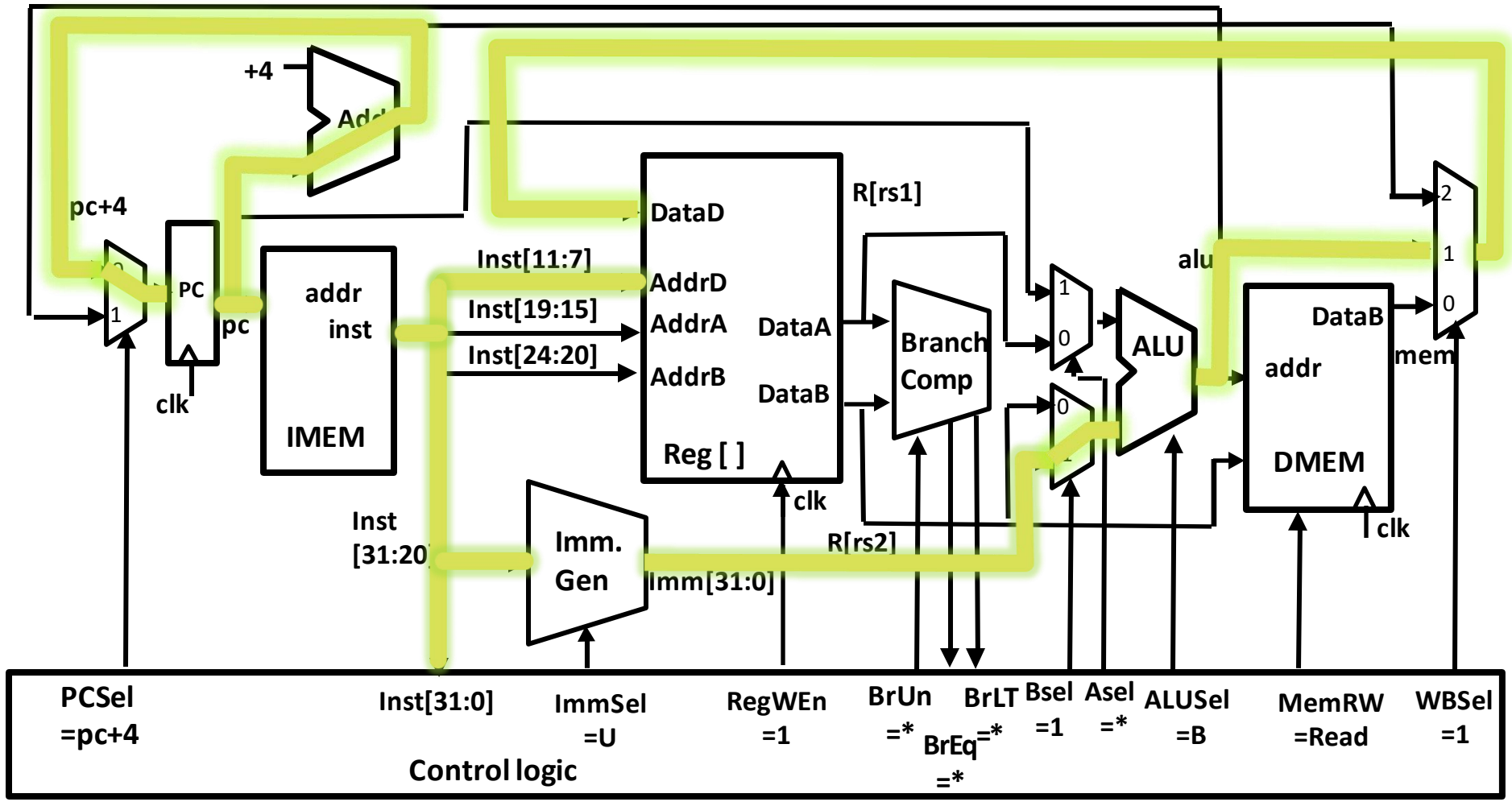


- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, **`rd`**
- Used for two instructions
  - **`lui`** – Load Upper Immediate
  - **`auipc`** – Add Upper Immediate to PC

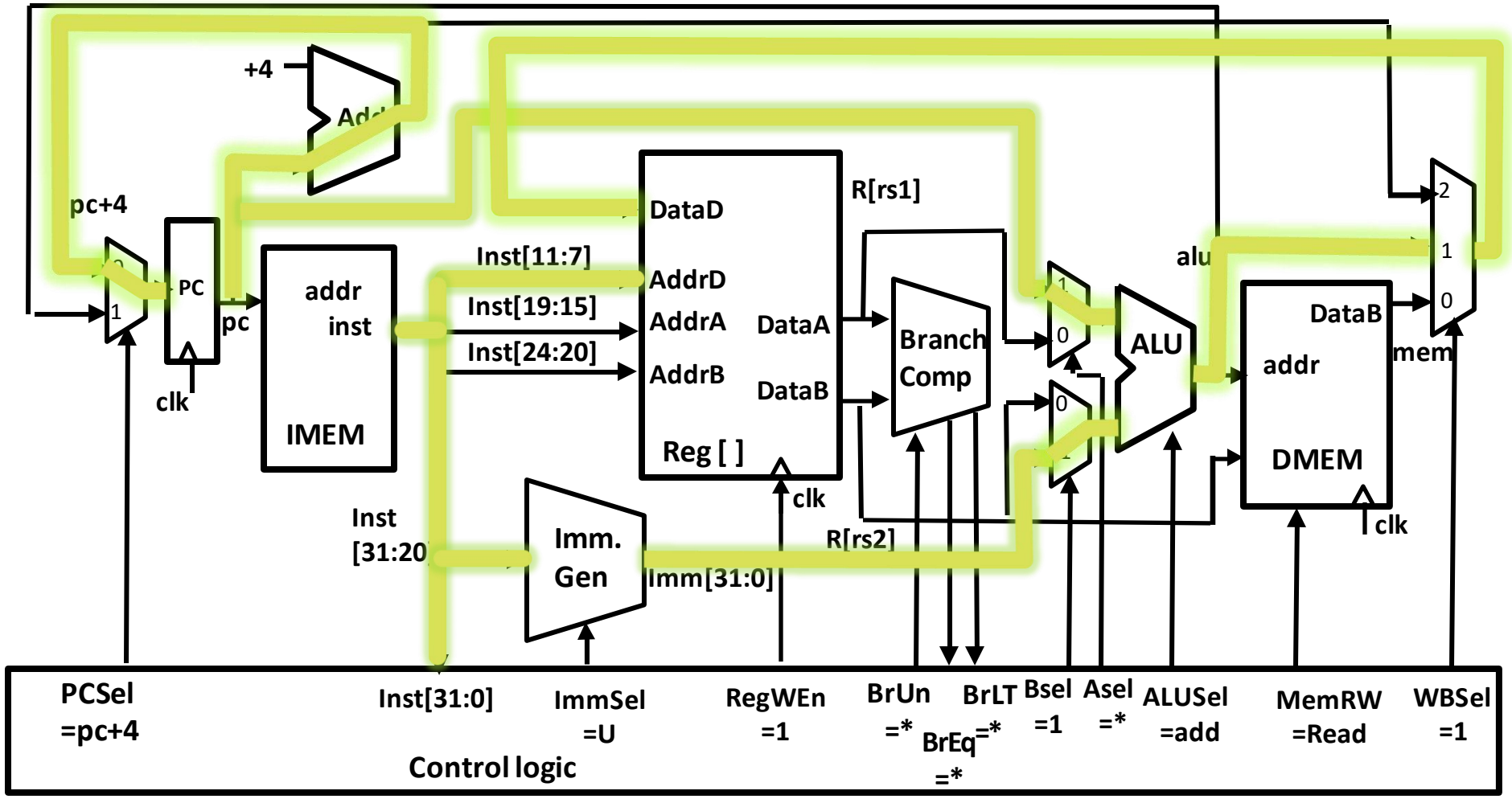
Datapath With LUI, AUIPC



Lighting Up **LUI**

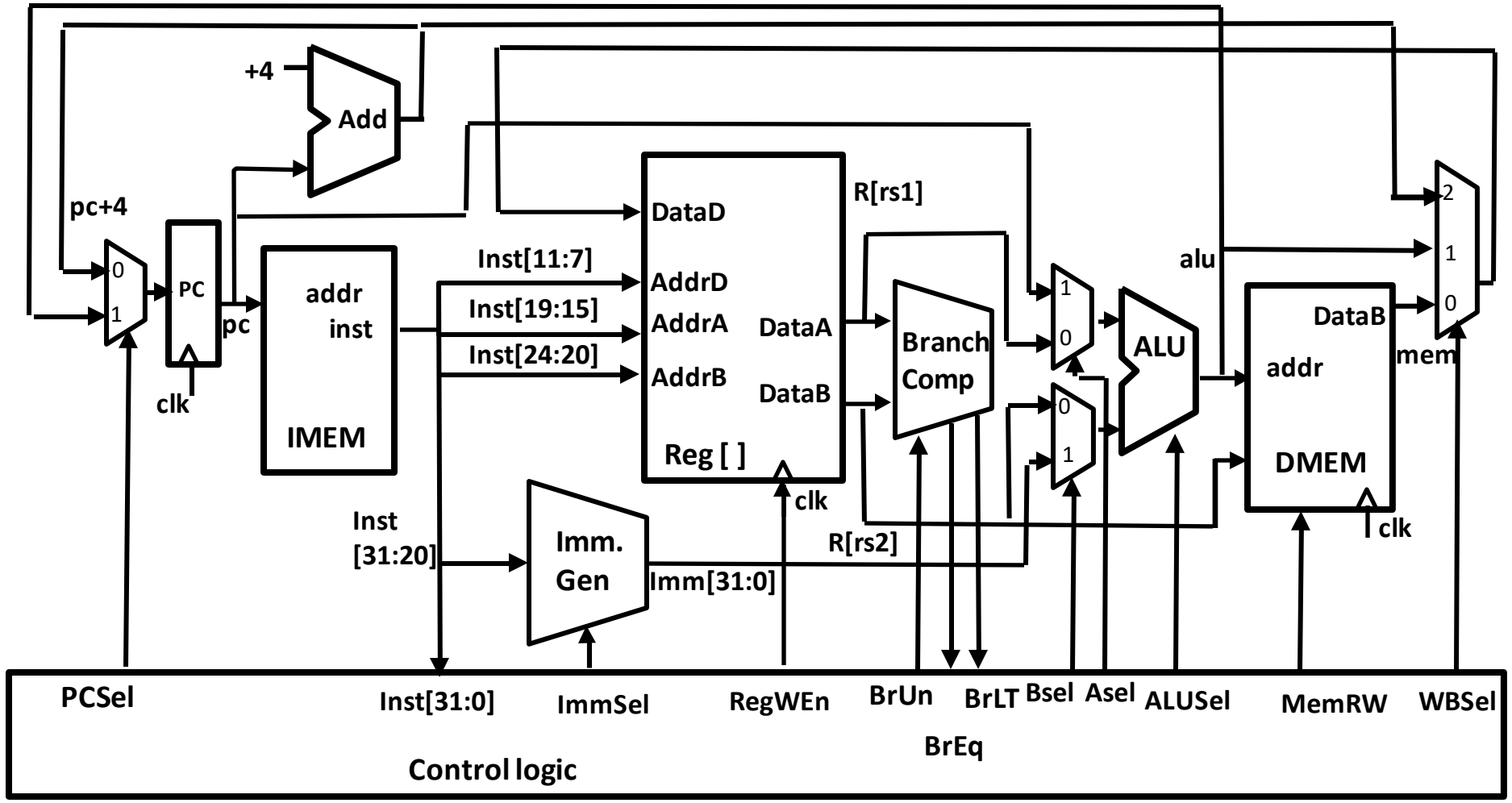


Lighting Up **AUIPC**



“And In Conclusion…”

Complete RV32I Datapath!



Complete RV32I ISA!

Open RISC-V Reference Card

Base Integer Instructions: RV32I

Category	Name	Fmt	RV32I Base
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt
	Shift Right Logical	R	SRL rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm
	Add Upper Imm to PC	U	AUIPC rd,imm
Logical	XOR	R	XOR rd,rs1,rs2
	XOR Immediate	I	XORI rd,rs1,imm
	OR	R	OR rd,rs1,rs2
	OR Immediate	I	ORI rd,rs1,imm
	AND	R	AND rd,rs1,rs2
	AND Immediate	I	ANDI rd,rs1,imm
Compare	Set <	R	SLT rd,rs1,rs2
	Set < Immediate	I	SLTI rd,rs1,imm
	Set < Unsigned	R	SLTU rd,rs1,rs2
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm

Category	Name	Fmt	RV32I Base
Loads	Load Byte	I	LB rd,rs1,imm
	Load Halfword	I	LH rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm
	Load Half Unsigned	I	LHU rd,rs1,imm
	Load Word	I	LW rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm
	Store Halfword	S	SH rs1,rs2,imm
	Store Word	S	SW rs1,rs2,imm
Branches	Branch =	B	BEQ rs1,rs2,imm
	Branch ≠	B	BNE rs1,rs2,imm
	Branch <	B	BLT rs1,rs2,imm
	Branch ≥	B	BGE rs1,rs2,imm
	Branch < Unsigned	B	BLTU rs1,rs2,imm
	Branch ≥ Unsigned	B	BGEU rs1,rs2,imm
Jump & Link	J&L	J	JAL rd,imm
	Jump & Link Register	I	JALR rd,rs1,imm
Synch	Synch thread	I	FENCE
Environment	CALL	I	ECALL
	BREAK	I	EBREAK

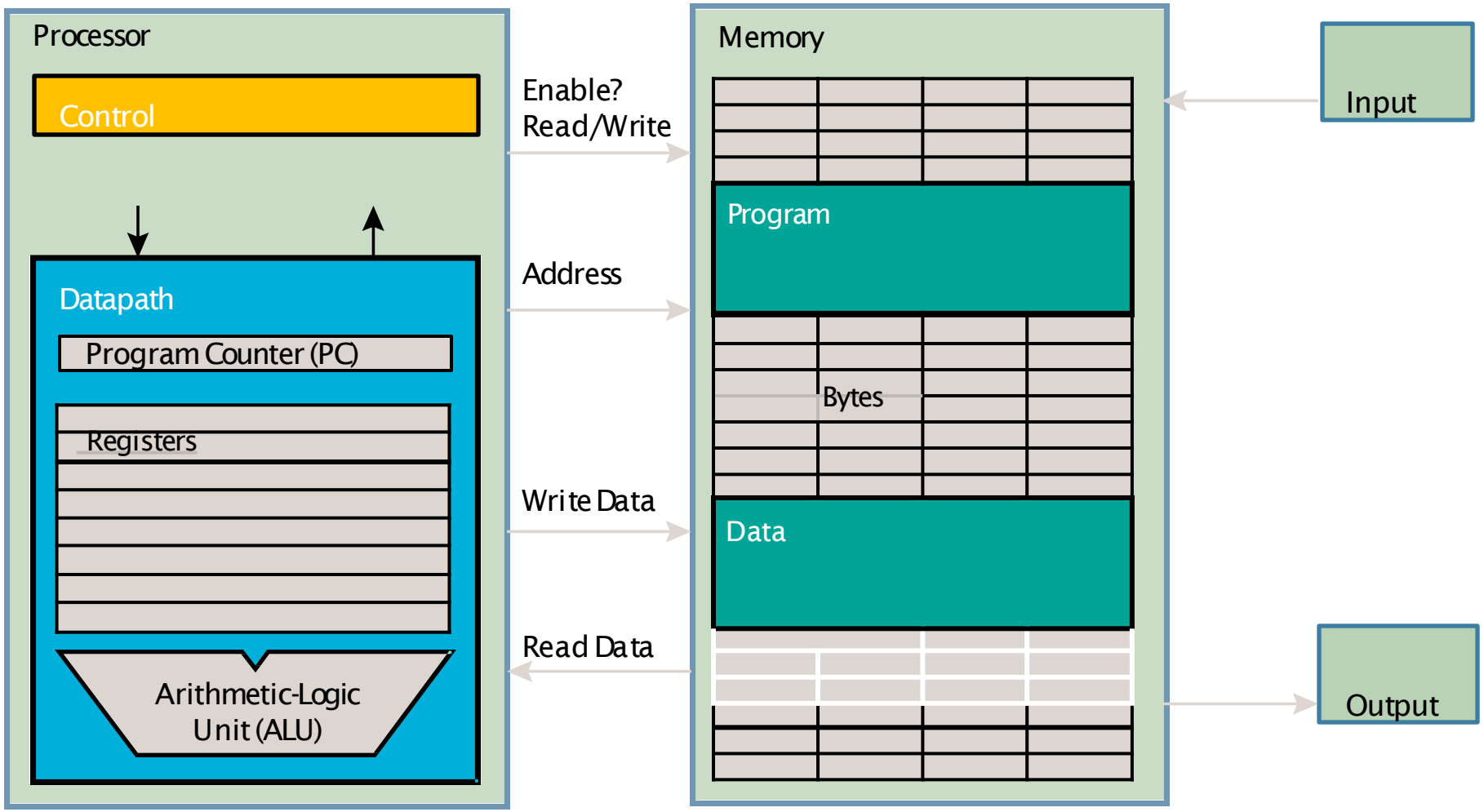
Not in 61C

Review

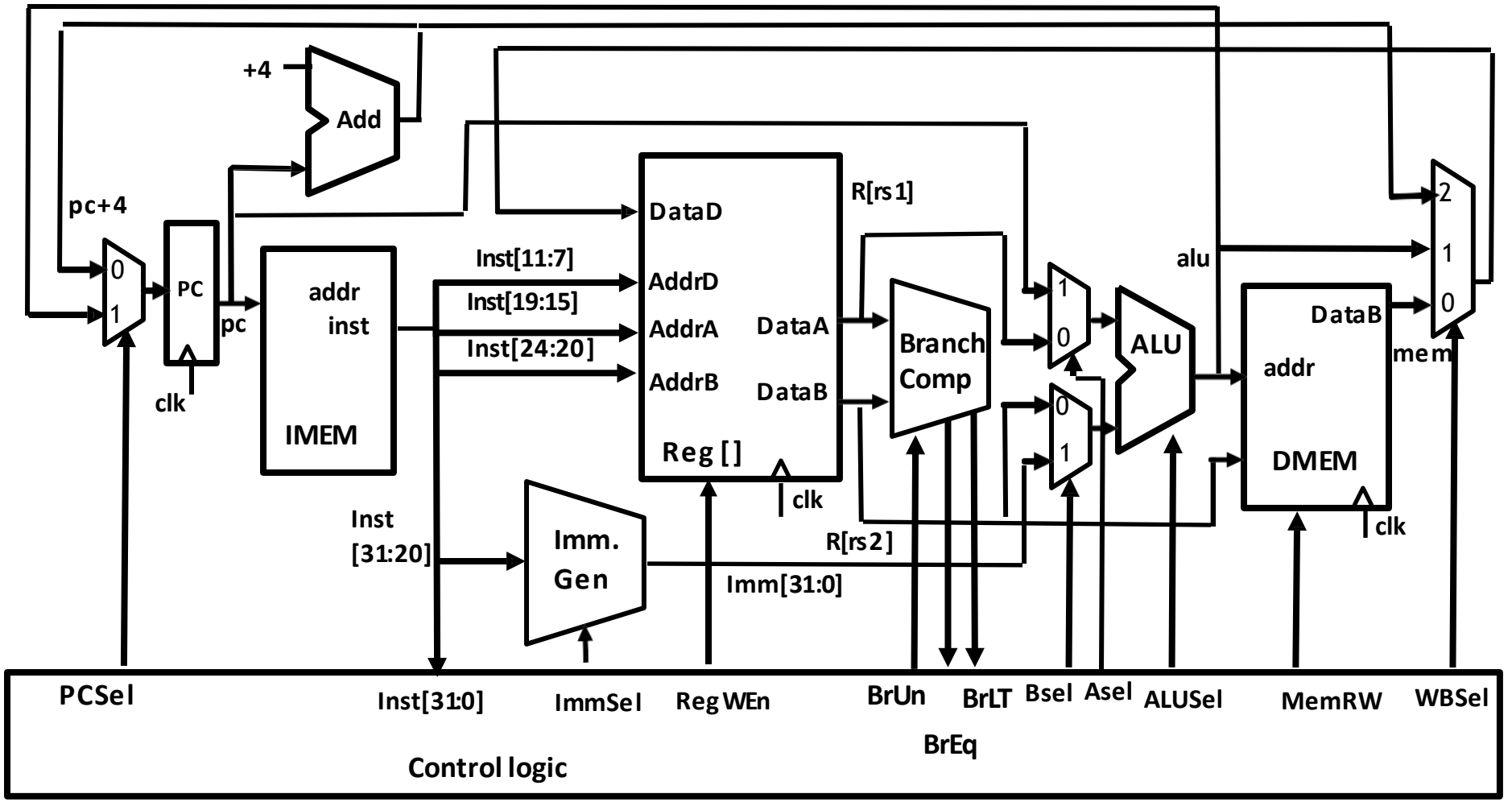
- We have designed a complete datapath
  - Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions
- 5 Phases of execution
  - IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases
- Controller specifies how to execute instructions
  - We still need to design it

Datapath
Control

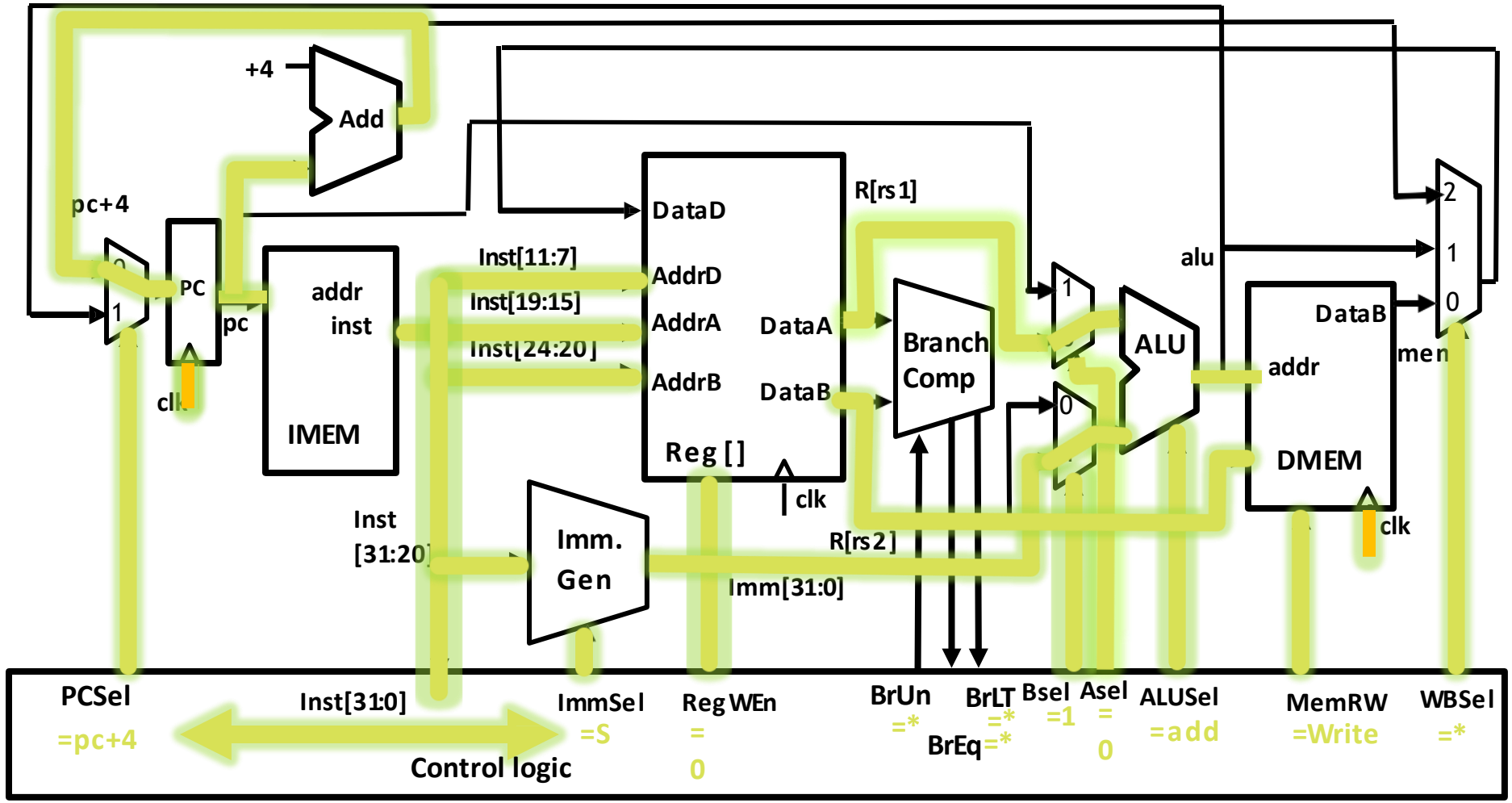
Our Single-Core Processor



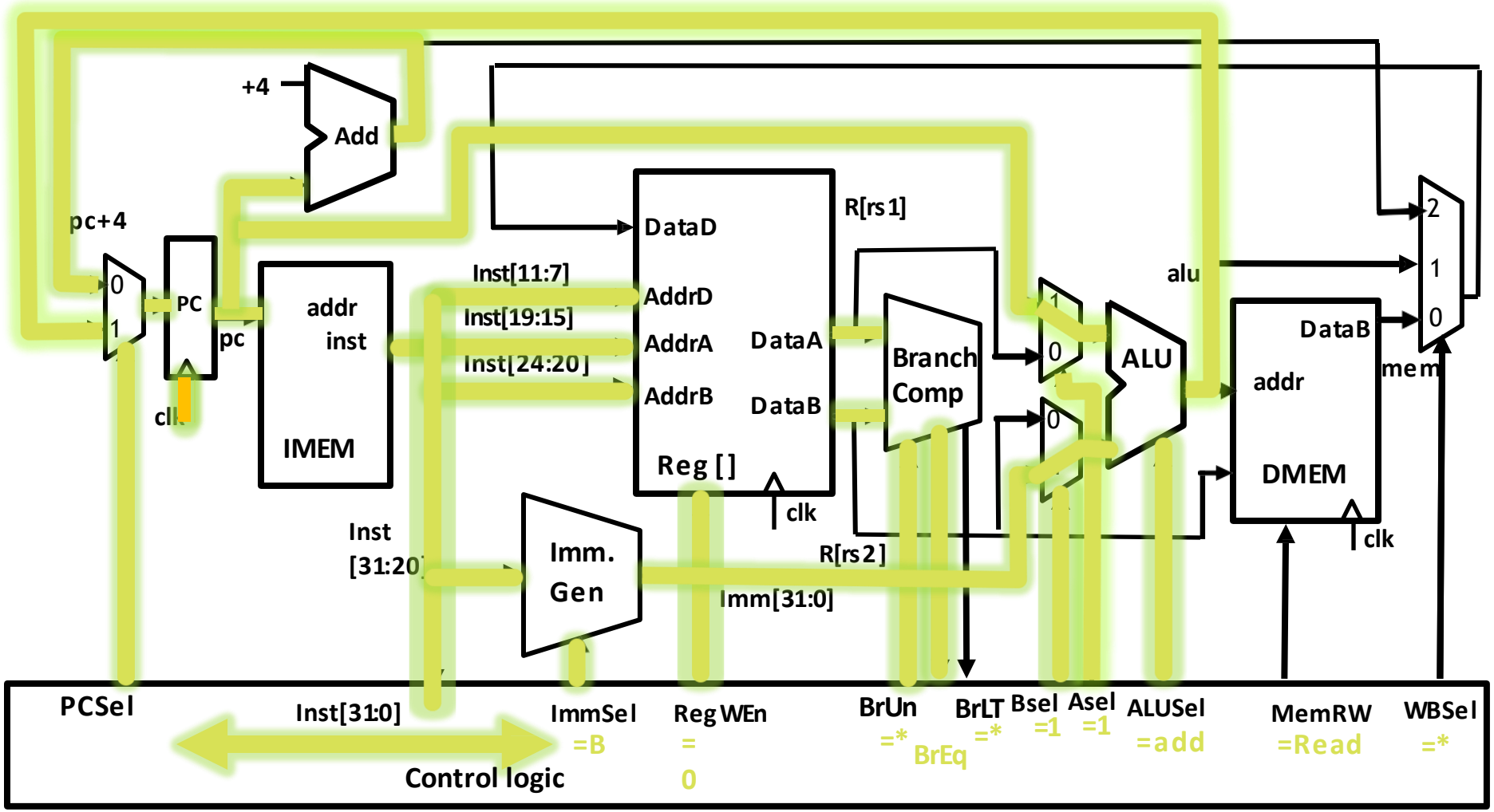
Single-Cycle RV32I Datapath and Control



Example: `sw`

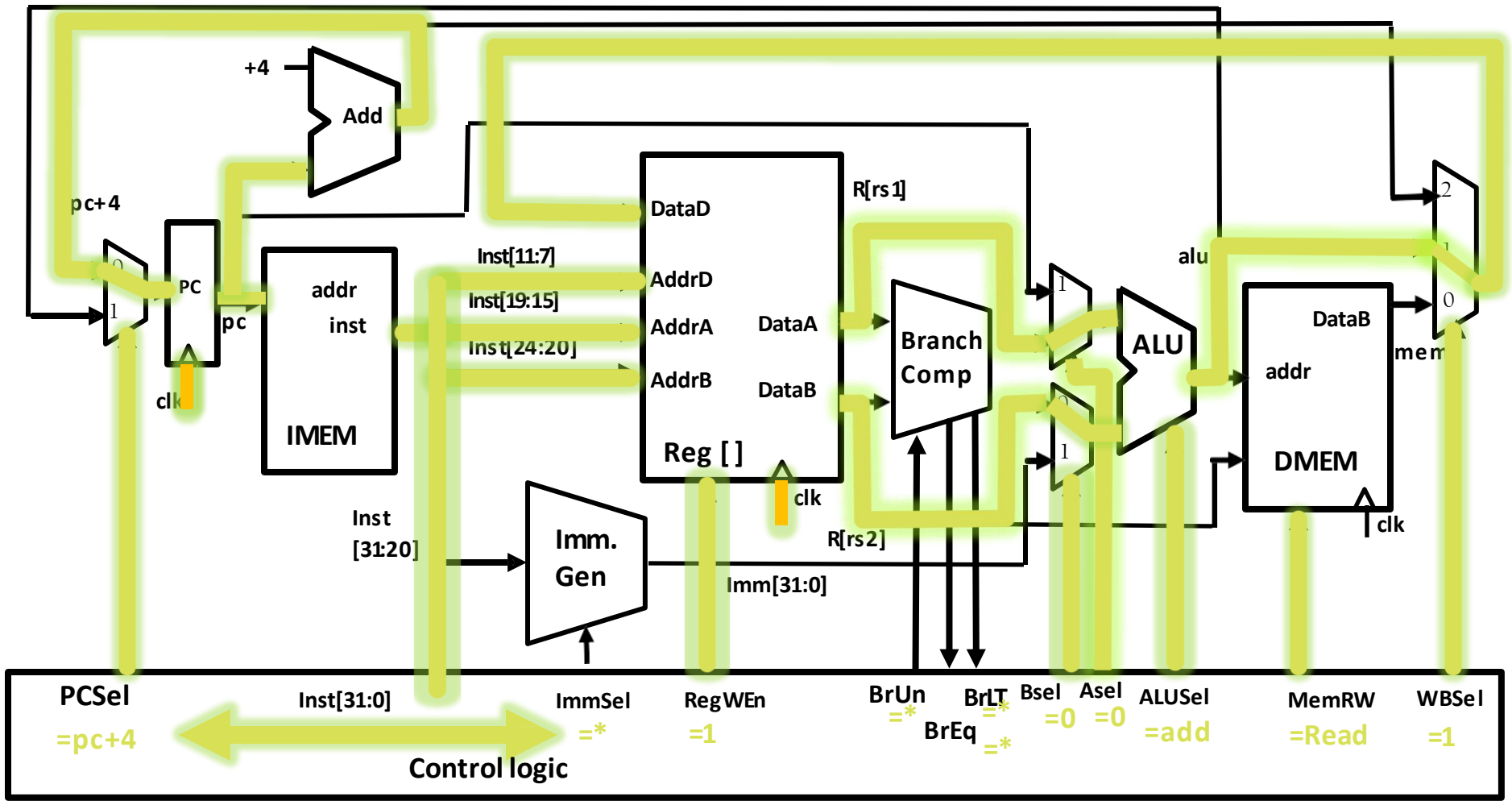

+4
Add
pc+4
PC
pc
addr
inst
IMEM
clk
Inst[11:7]
Inst[19:15]
Inst[24:20]
DataD
AddrD
AddrA
AddrB
DataA
DataB
Reg [ ]
clk
R[rs1]
Branch Comp
1
0
ALU
alu
2
1
0
DataB
addr
DMEM
clk
men
Inst
[31:20]
Imm.
Gen
Imm[31:0]
R[rs2]
PCSel
=pc+4
Inst[31:0]
Control logic
ImmSel
=S
RegWEn
=
0
BrUn
=*
BrEq
BrLT
=*
=*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

Example: **beq**

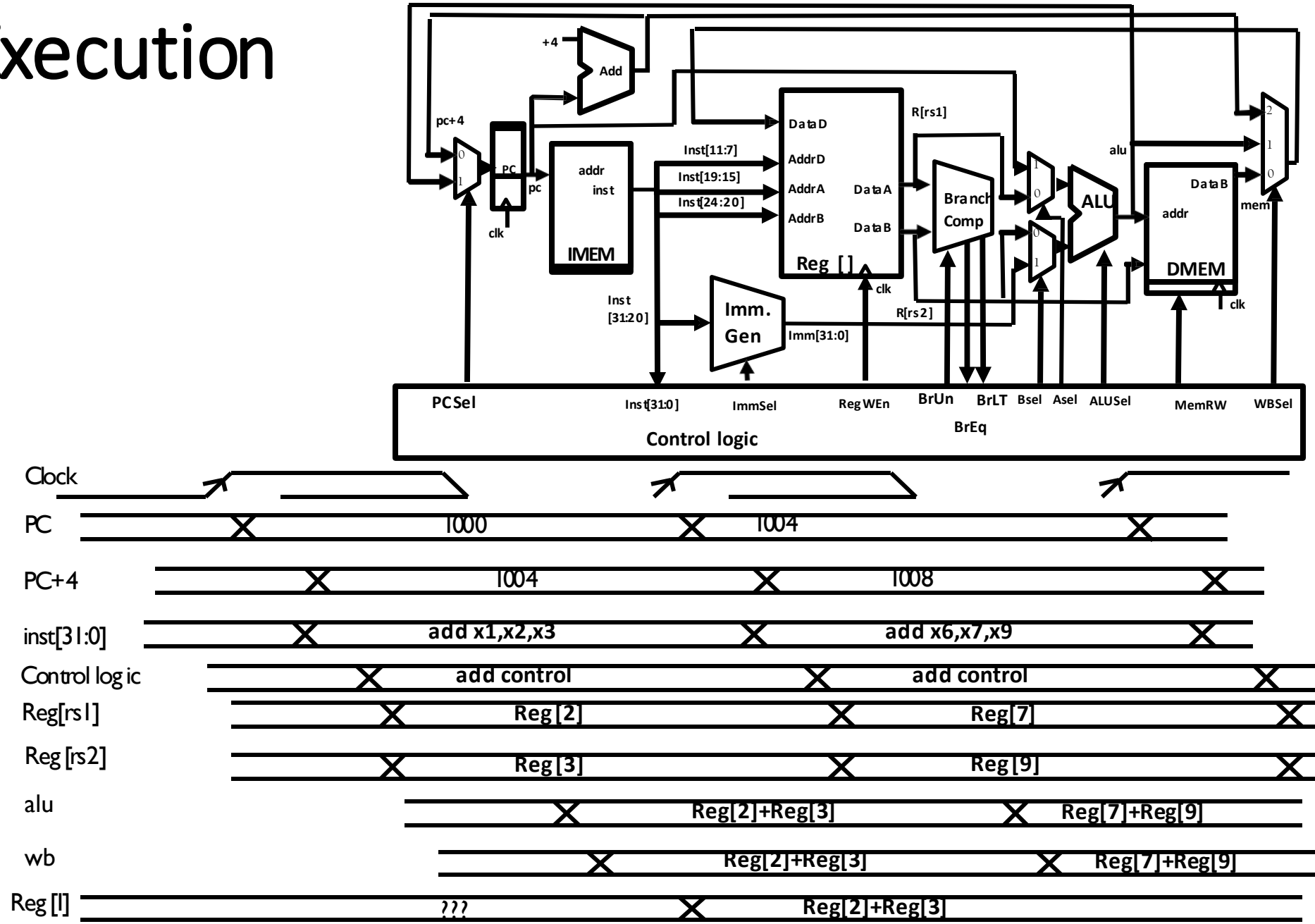

+4
Add
pc+4
0
1
PC
clk
pc
addr
inst
IMEM
Inst[11:7]
Inst[19:15]
Inst[24:20]
Inst
[31:20]
DataD
AddrD
AddrA
AddrB
DataA
DataB
Reg [ ]
clk
R[rs1]
R[rs2]
Imm.
Gen
Imm[31:0]
Branch
Comp
1
0
0
ALU
alu
DataB
addr
DMEM
clk
mem
2
1
0
PCSel
Inst[31:0]
Control logic
ImmSel
=B
Reg WEn
=
0
BrUn
=*
BrEq
BrLT
=*
Bsel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

Instruction Timing

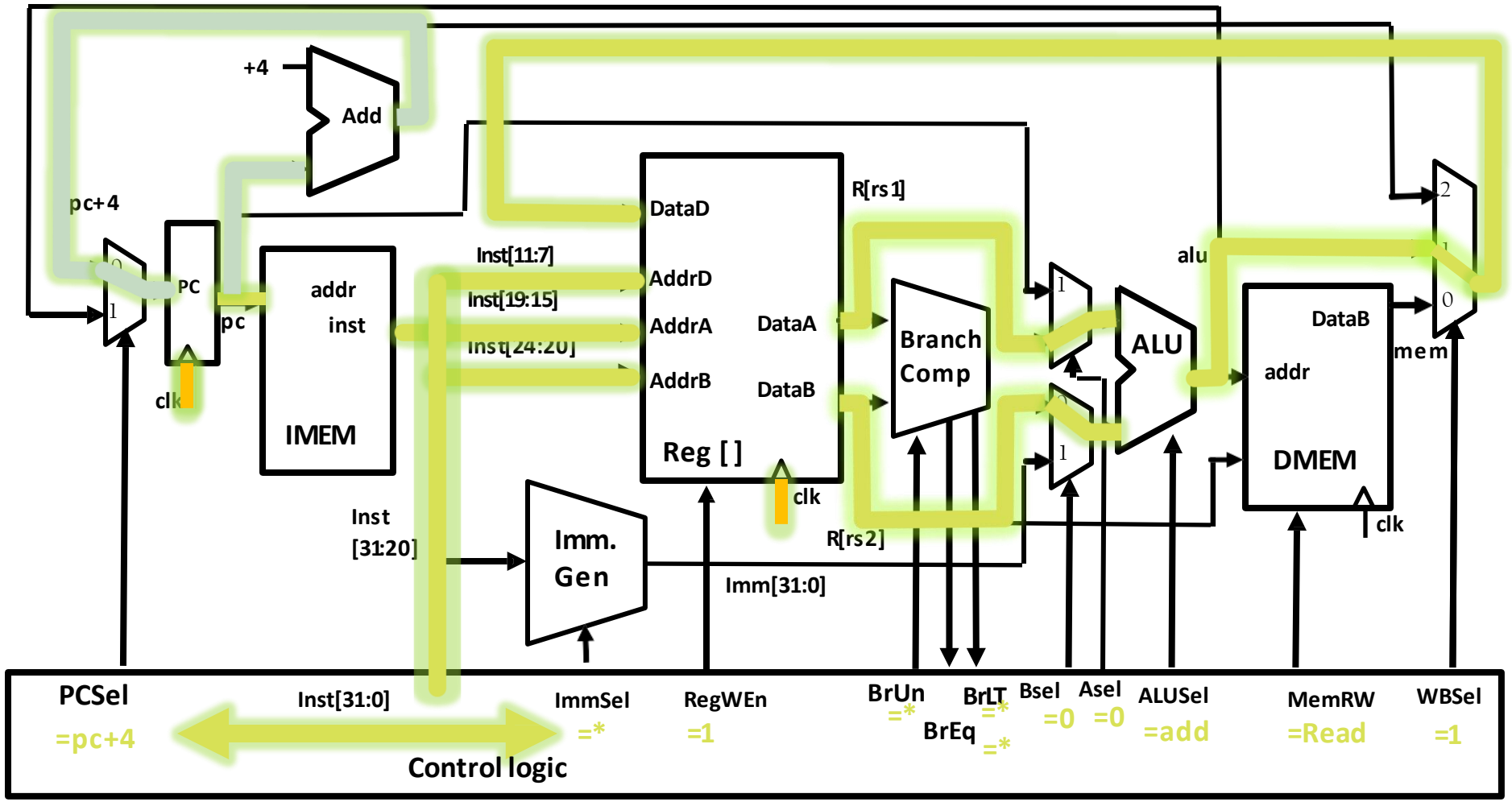
Example: **add**


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

Add Execution



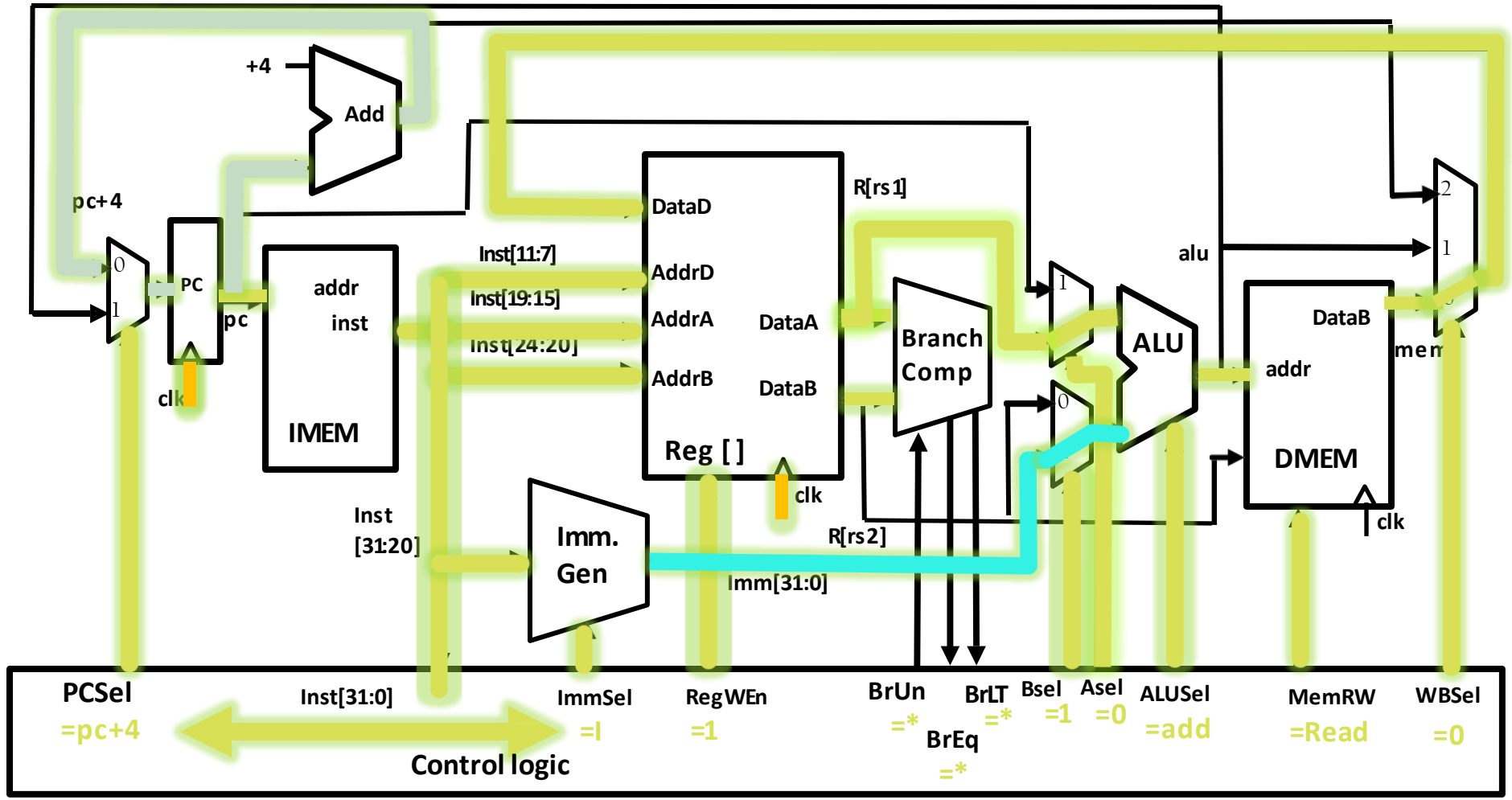
Example: **add** timing



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

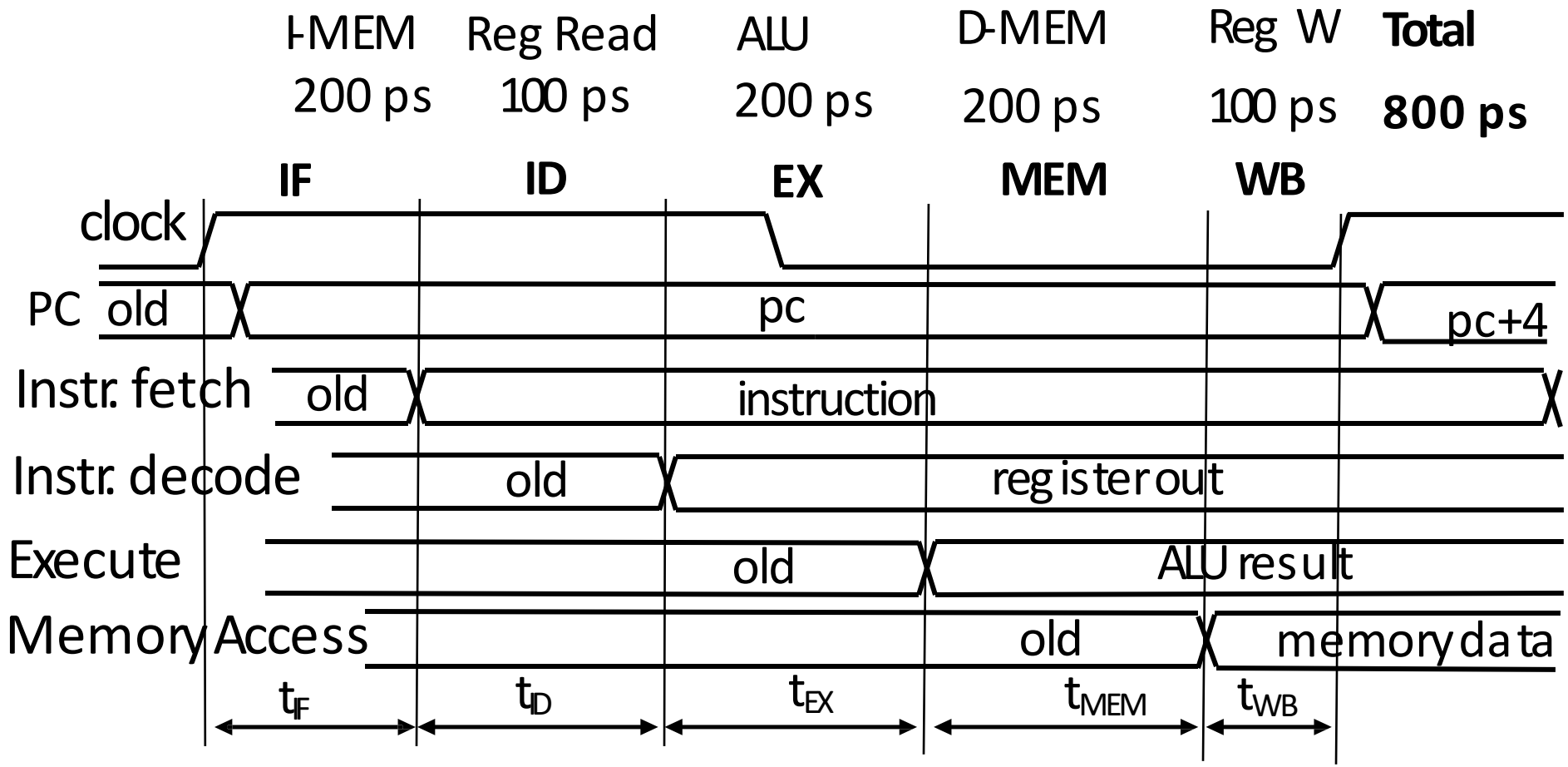
$= t_{clk\text{-}q} + t_{IMEM} + t_{Reg} + t_{mux} + t_{ALU} + t_{mux} + t_{setup}$

Example: `lw`



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

Instruction Timing



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

Control Logic Design

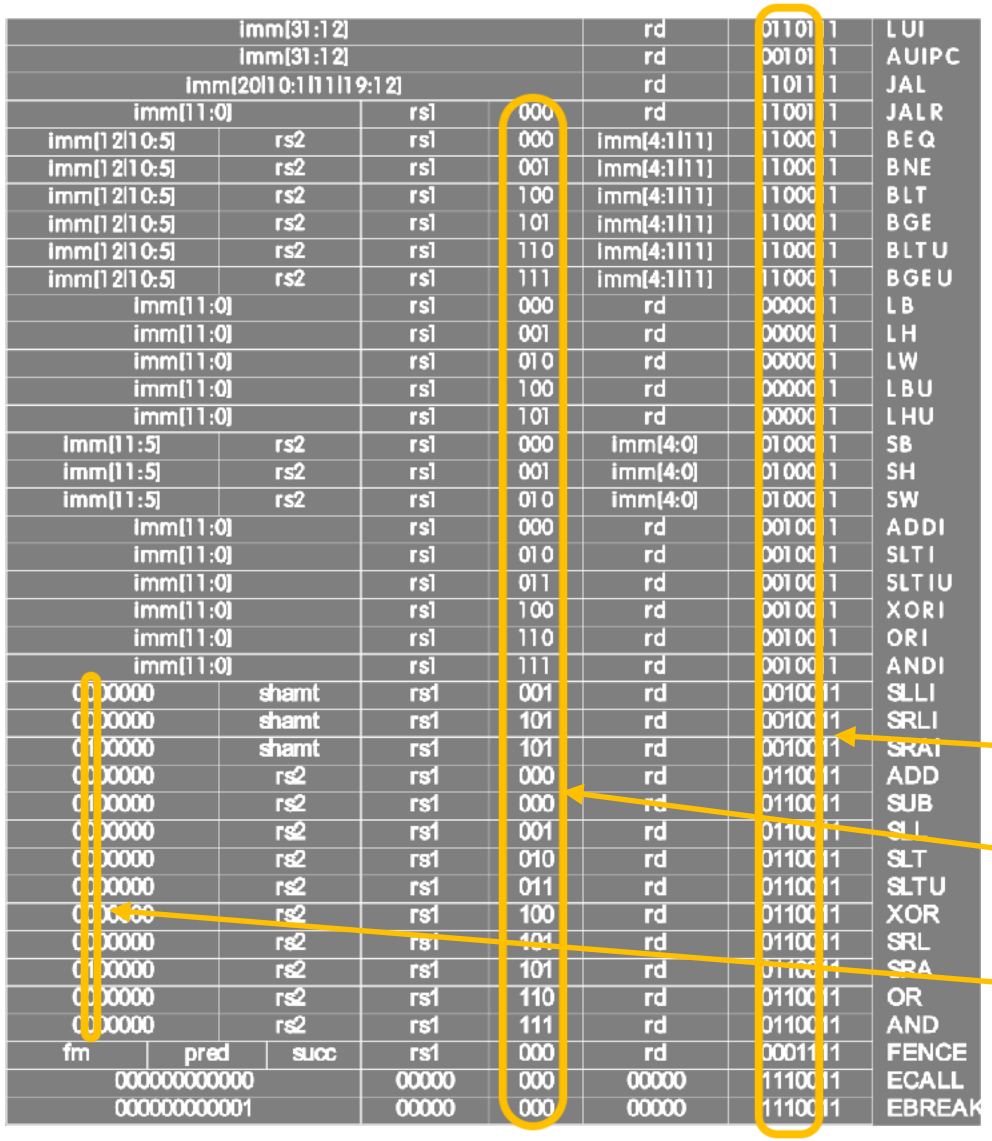
Control Logic Truth Table

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

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	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
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

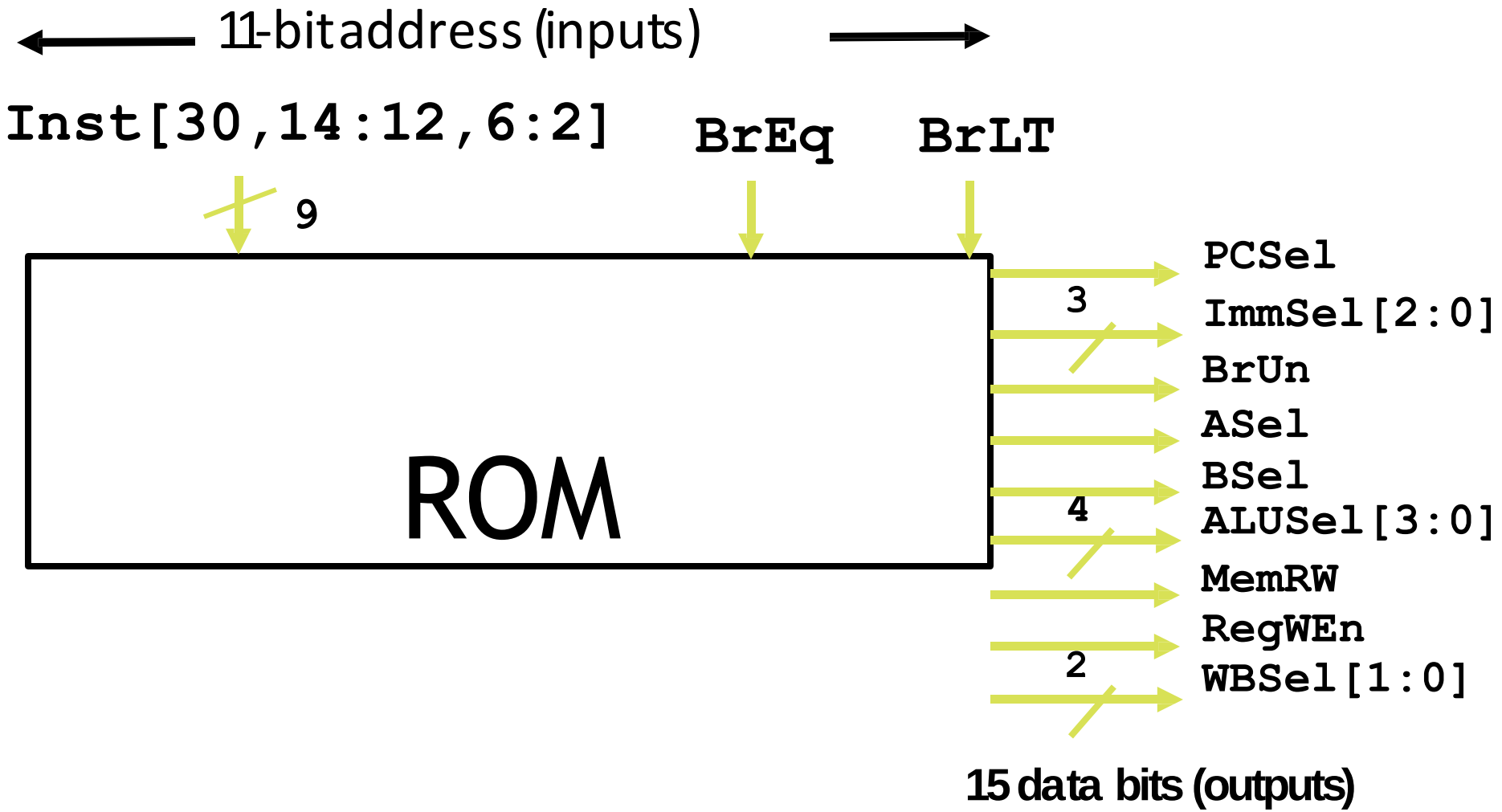
- Instruction type encoded using only 9 bits:
- **`inst[30]`, `inst[14:12]`, `inst[6:2]`**

`inst[6:2]`

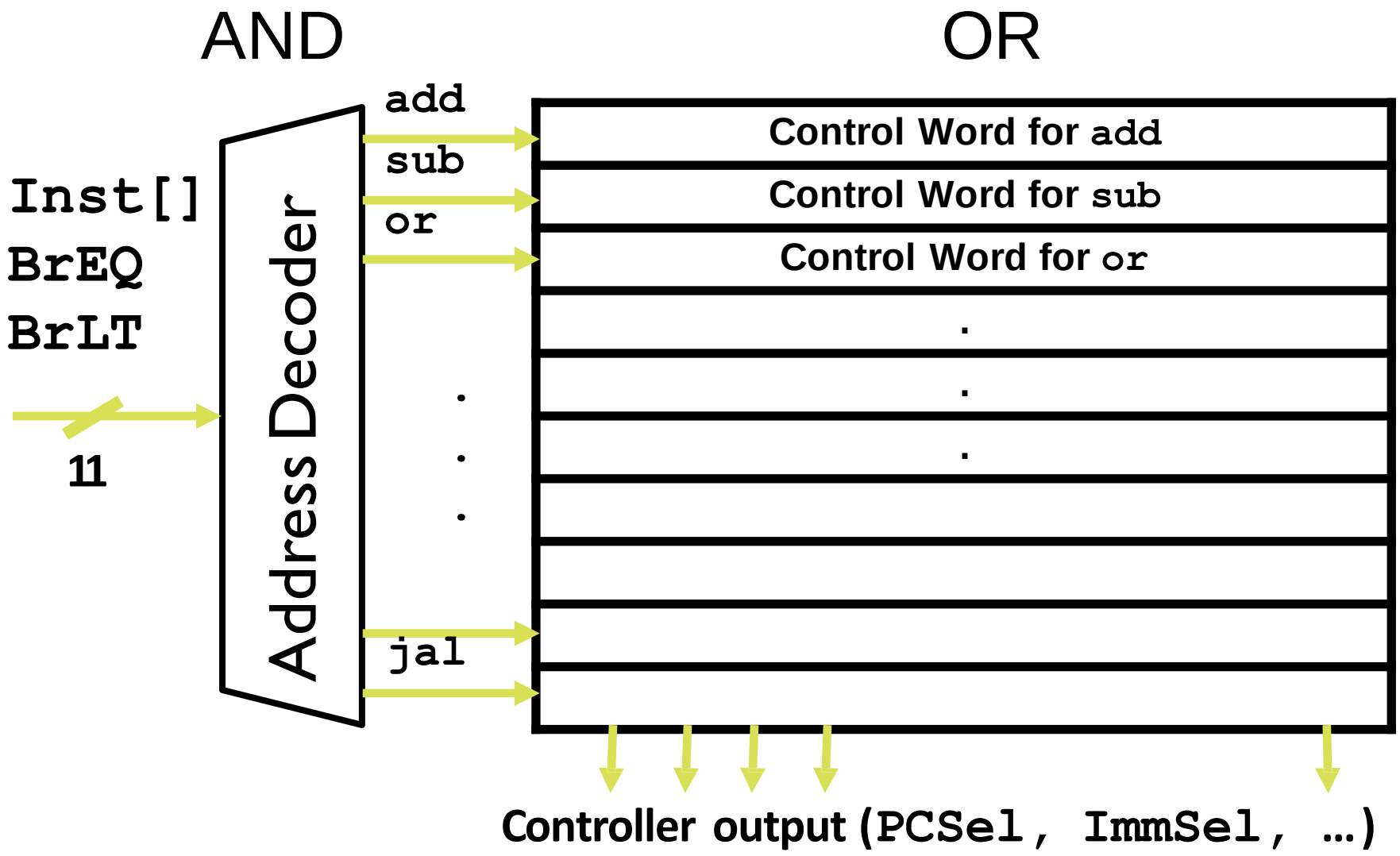
`inst[14:12]`

`inst[30]`

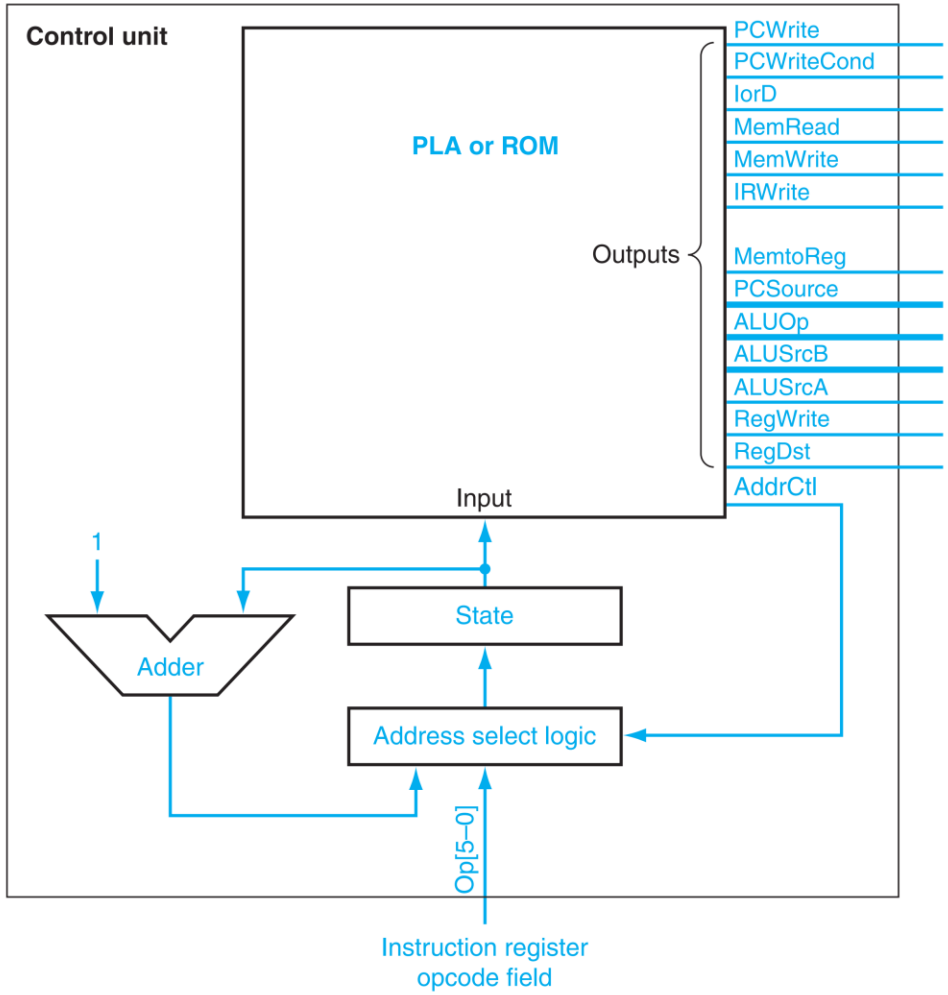
ROM-based Control

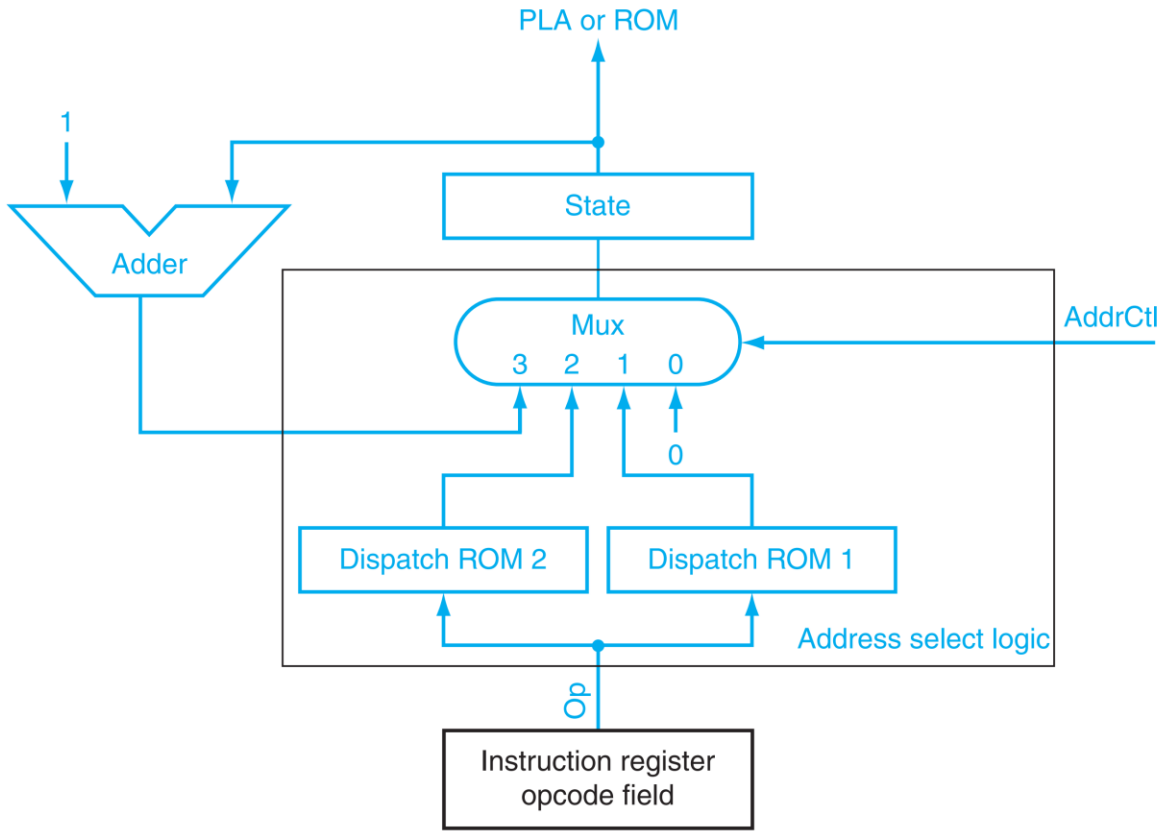


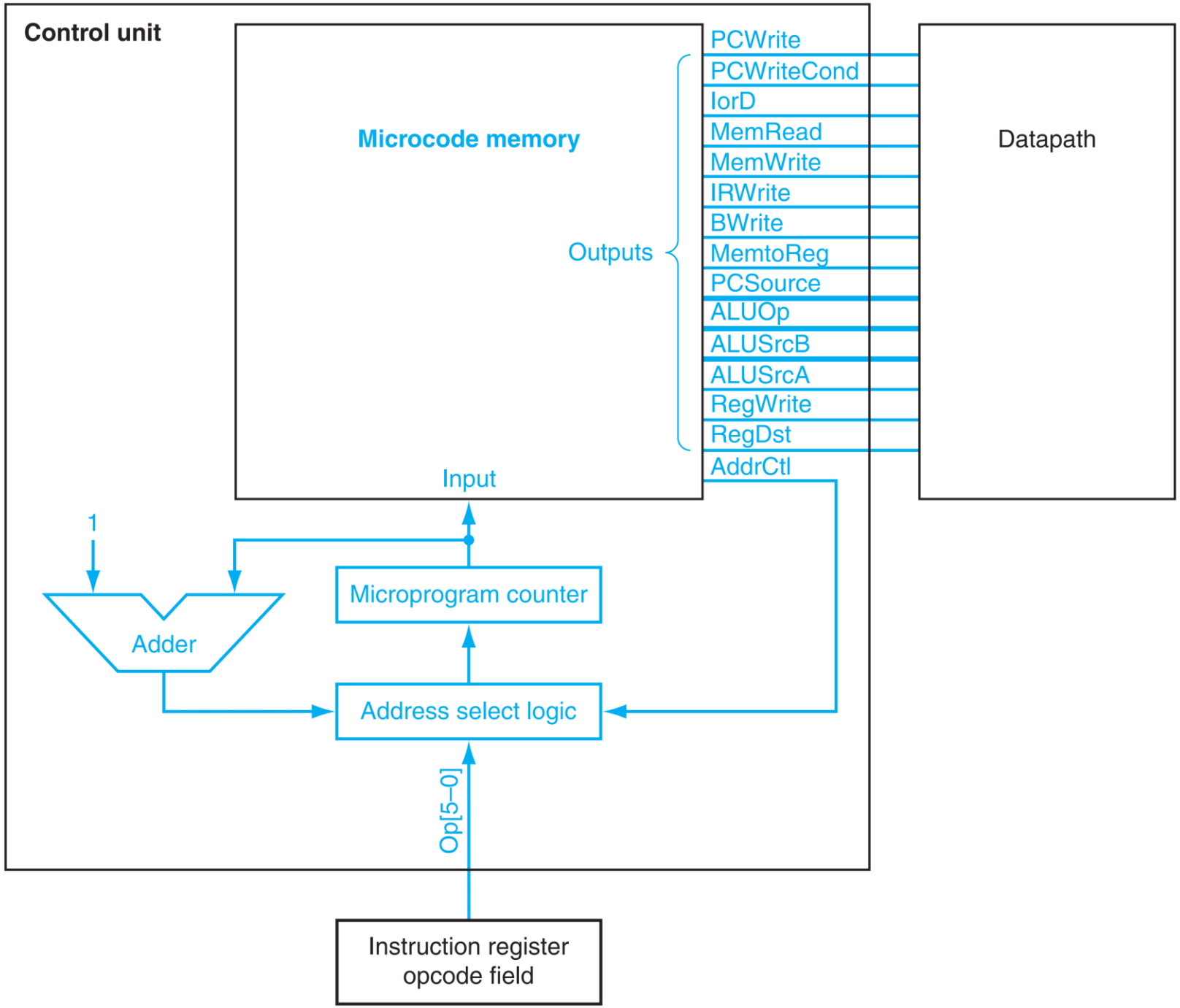
ROM Controller Implementation



The control unit using an explicit counter to compute the next state



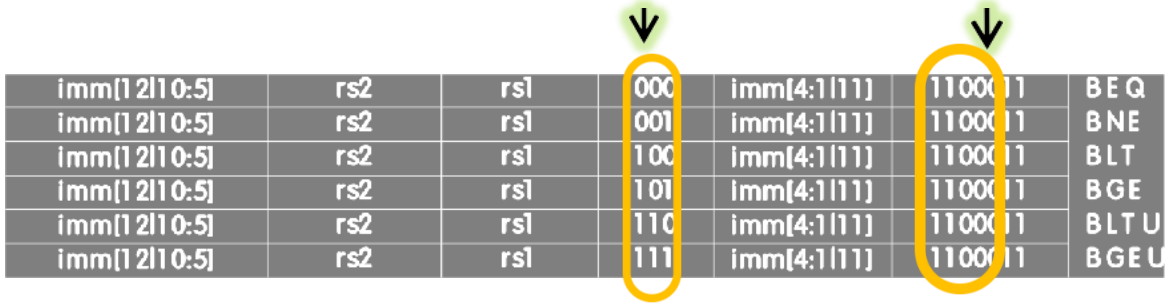



Control unit
Microcode memory
Outputs
PCWrite
PCWriteCond
IorD
MemRead
MemWrite
IRWrite
BWrite
MemtoReg
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
AddrCtl
Datapath
Input
1
Adder
Microprogram counter
Address select logic
Op[5–0]
Instruction register
opcode field

Combinational Logic Control

- Simplest example: **`BrUn`**

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



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

- How to decode whether BrUn is 1?

`BrUn = Inst [13] • Branch`

Control Logic to Decode add

$$\text{add} = i[30] \bullet i[14] \bullet i[13] \bullet i[12] \bullet \text{R-type}$$

inst[30] inst[14:12] inst[6:2]

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

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

$$\text{RV32I} = i[1] \bullet i[0]$$

"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)

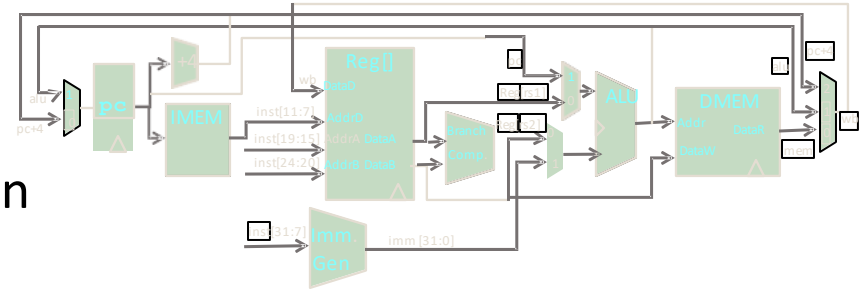
```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

Assembler

Machine Language Program (RISC-V)

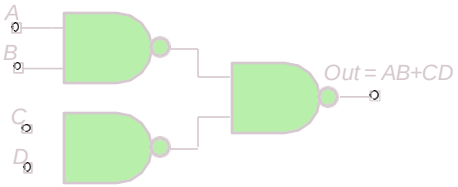
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

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



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