

Computer Architecture

Lecture 02 – How to Build a RISC-V Processor (ISA & MicroArch)

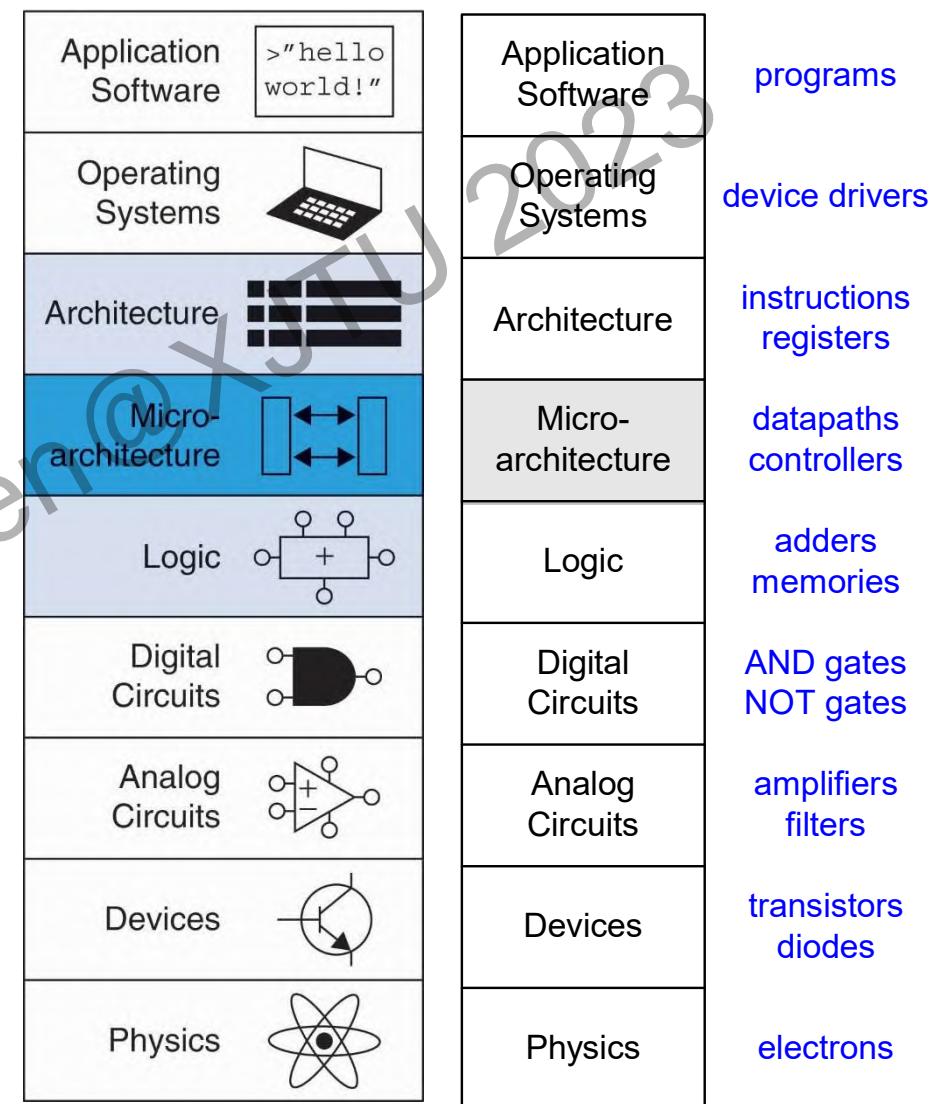
Pengju Ren

Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University

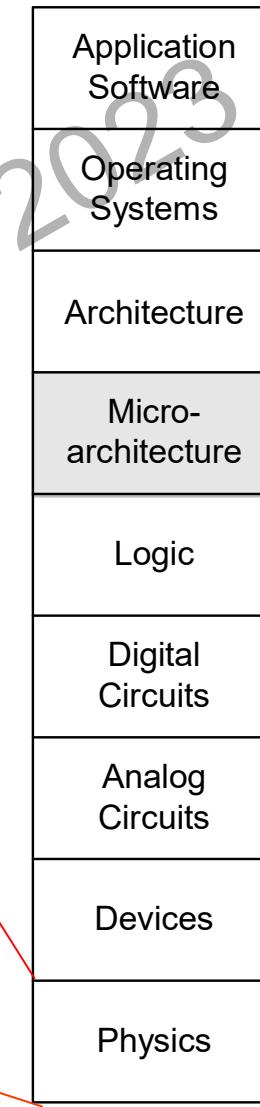
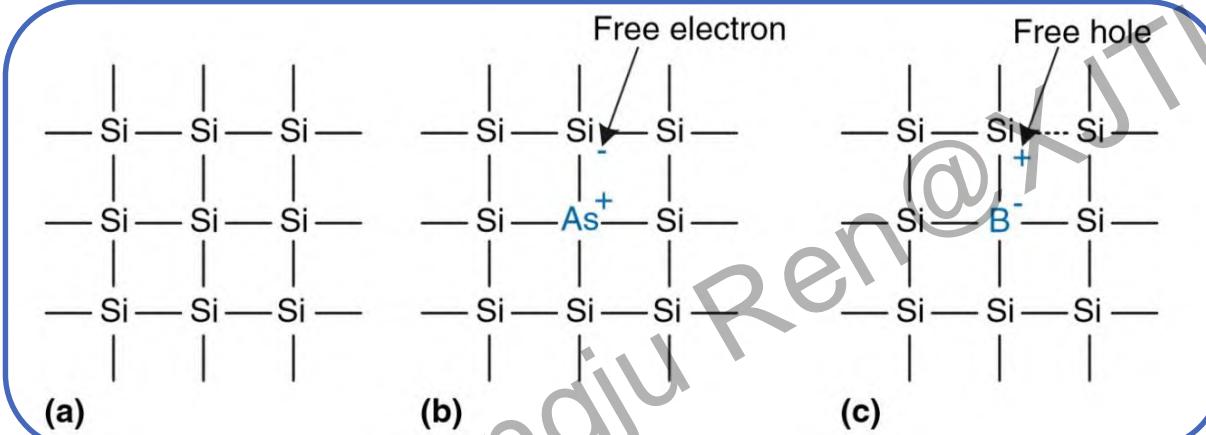
<http://gr.xjtu.edu.cn/web/pengjuren>

Recap: How to design a processor

- Single-Cycle Processor
- Multicycle Processor
- Pipeline

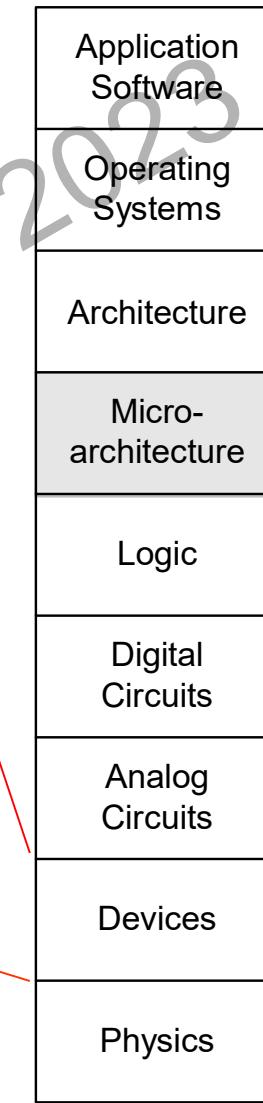
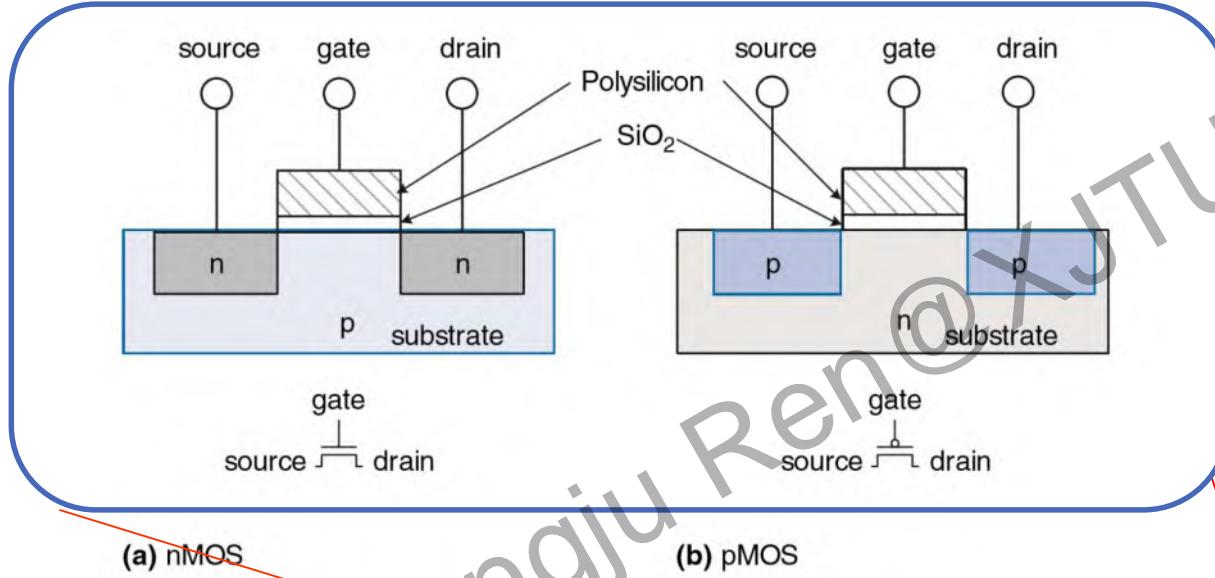


Recap(Digital Logic and Circuits)



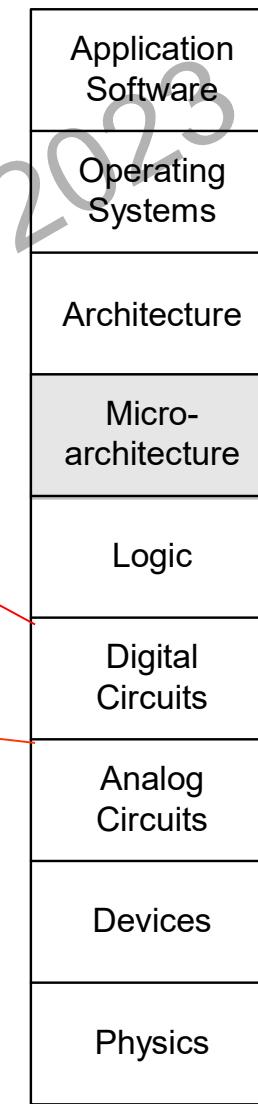
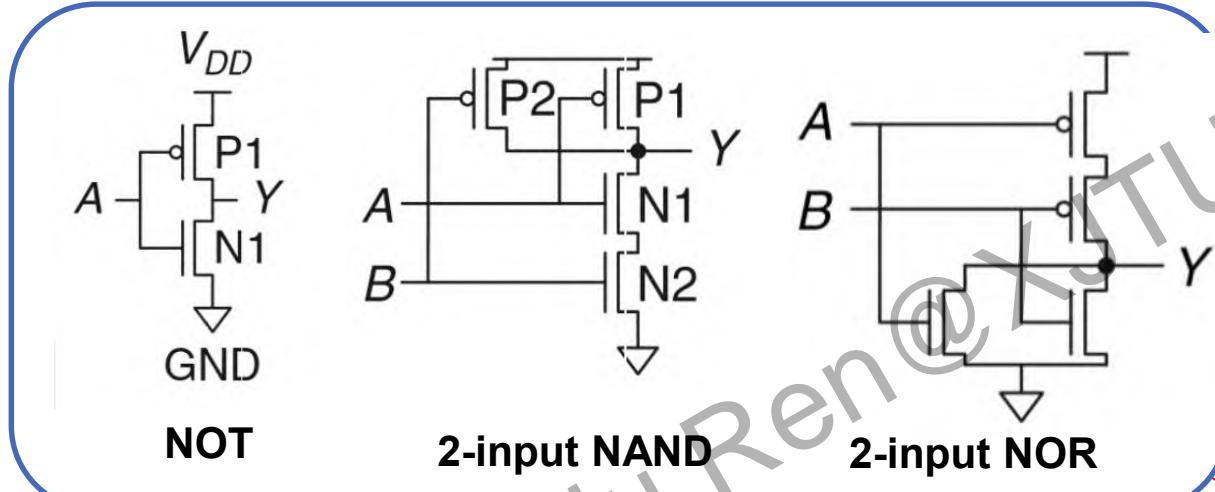
programs
device drivers
instructions
registers
datapaths
controllers
adders
memories
AND gates
NOT gates
amplifiers
filters
transistors
diodes
electrons

Recap(Digital Logic and Circuits)



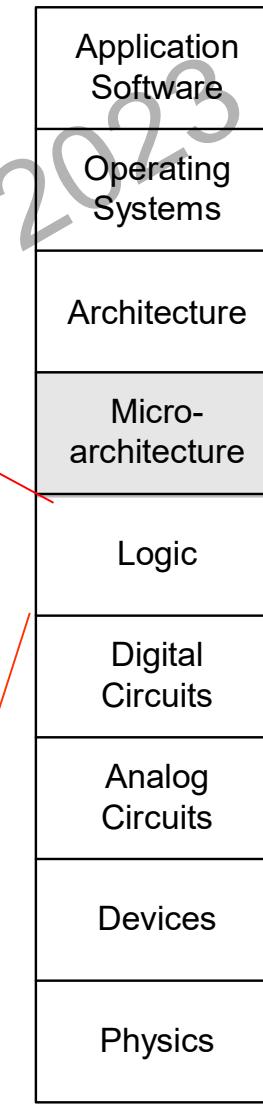
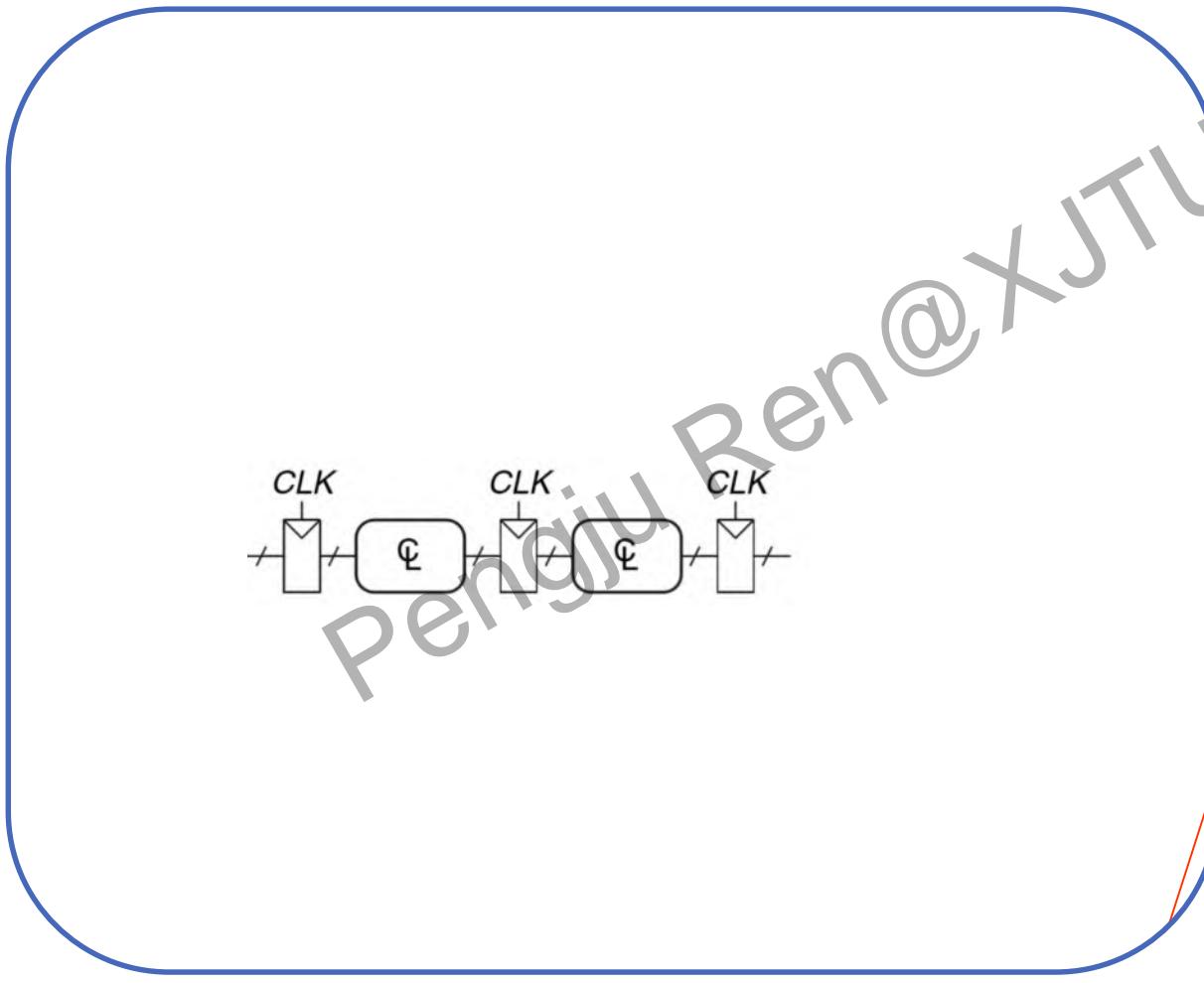
programs
device drivers
instructions
registers
datapaths
controllers
adders
memories
AND gates
NOT gates
amplifiers
filters
transistors
diodes
electrons

Recap(Digital Logic and Circuits)

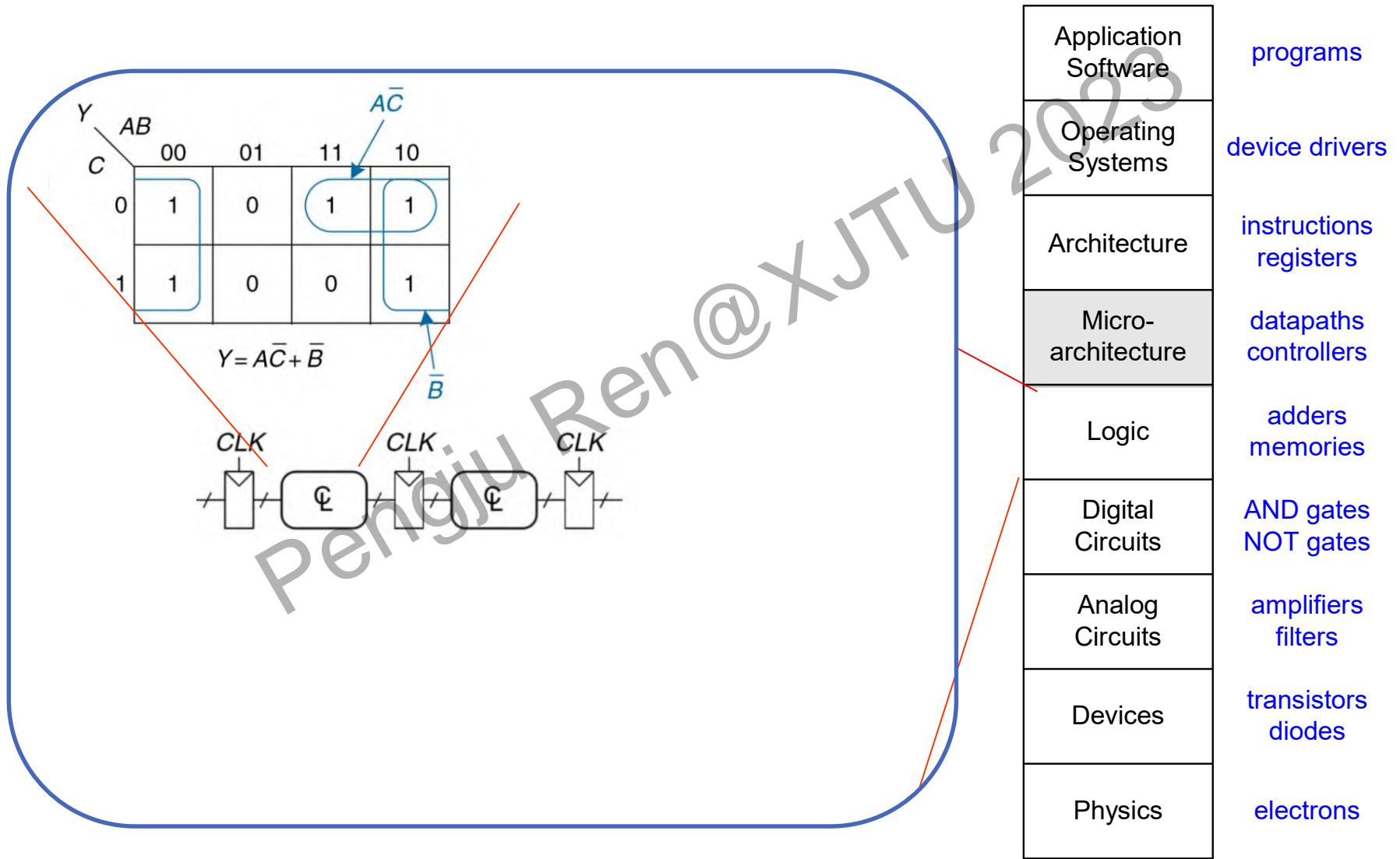


programs
device drivers
instructions
registers
datapaths
controllers
adders
memories
AND gates
NOT gates
amplifiers
filters
transistors
diodes
electrons

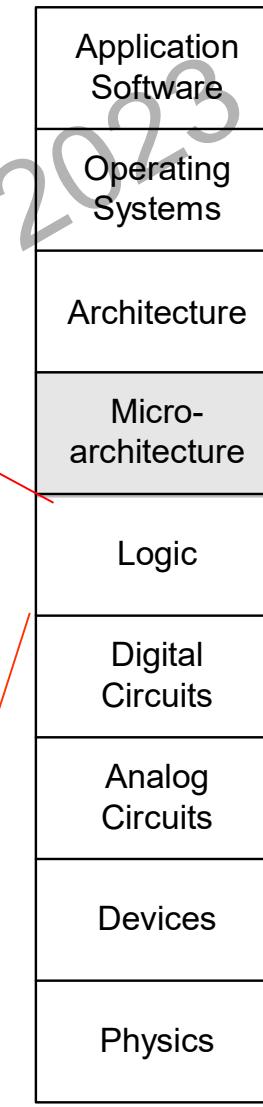
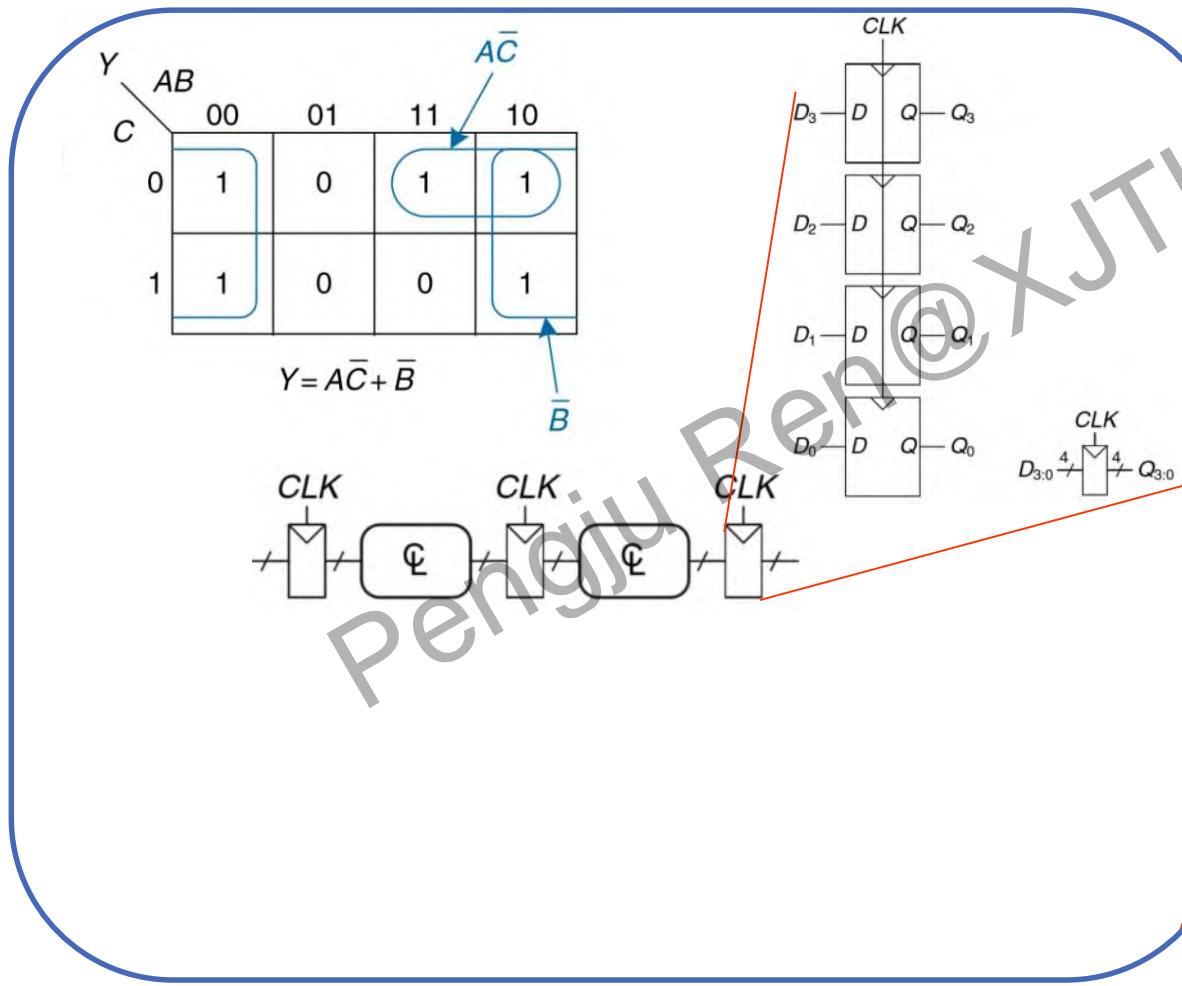
Recap(Digital Logic and Circuits)



Recap(Digital Logic and Circuits)

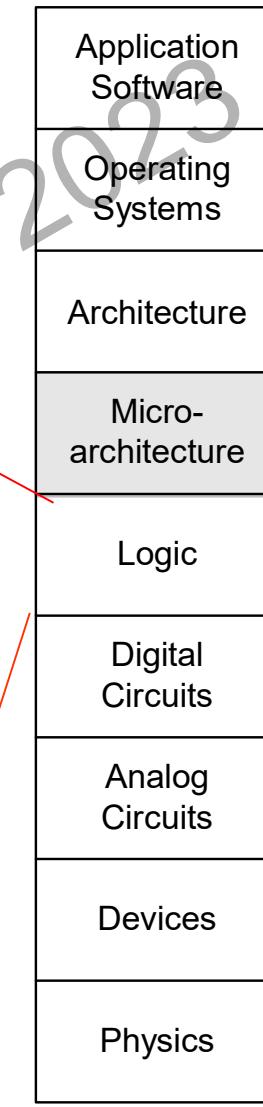
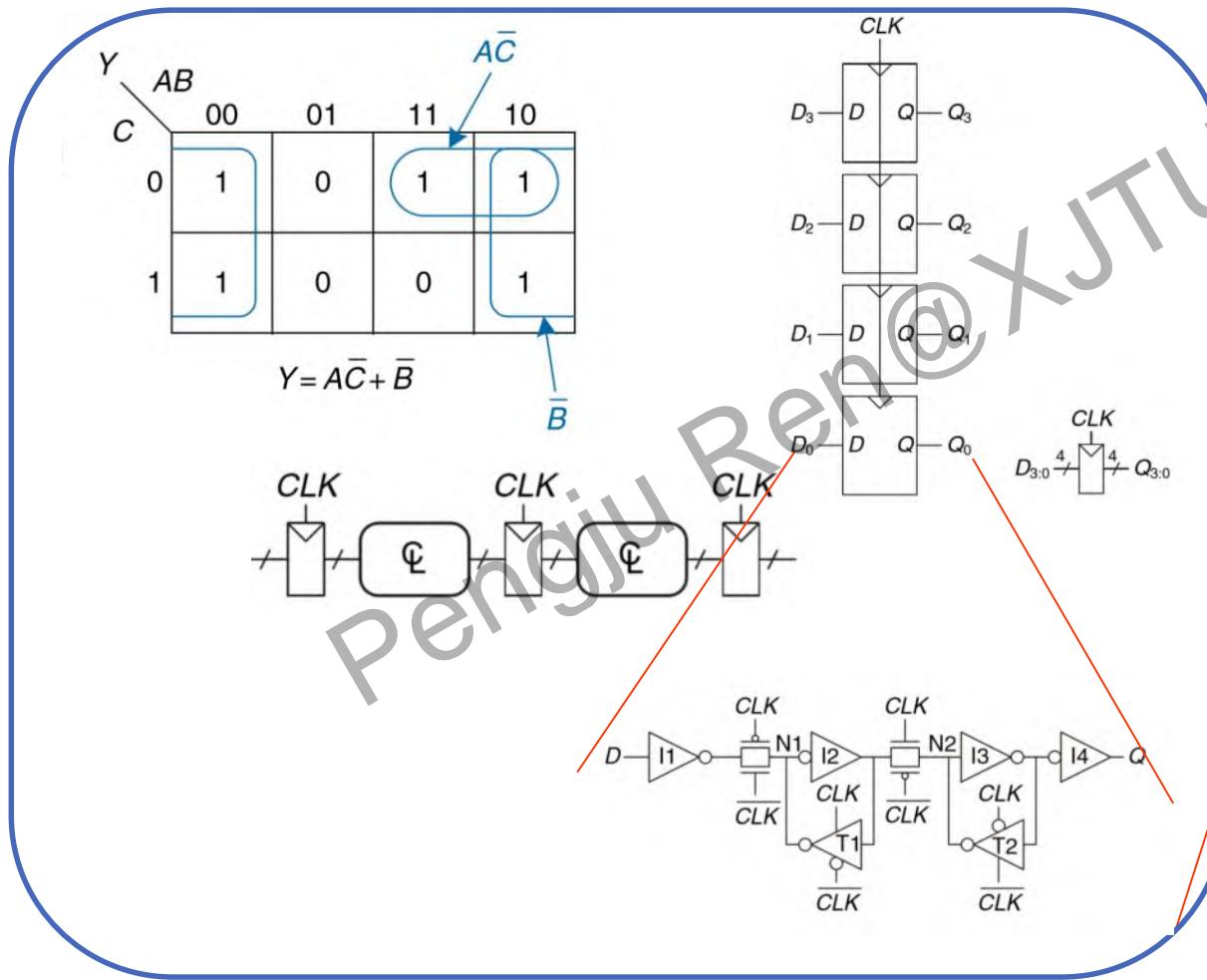


Recap(Digital Logic and Circuits)



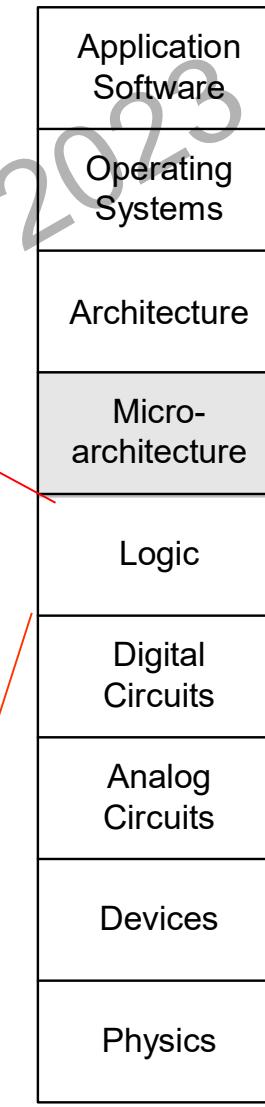
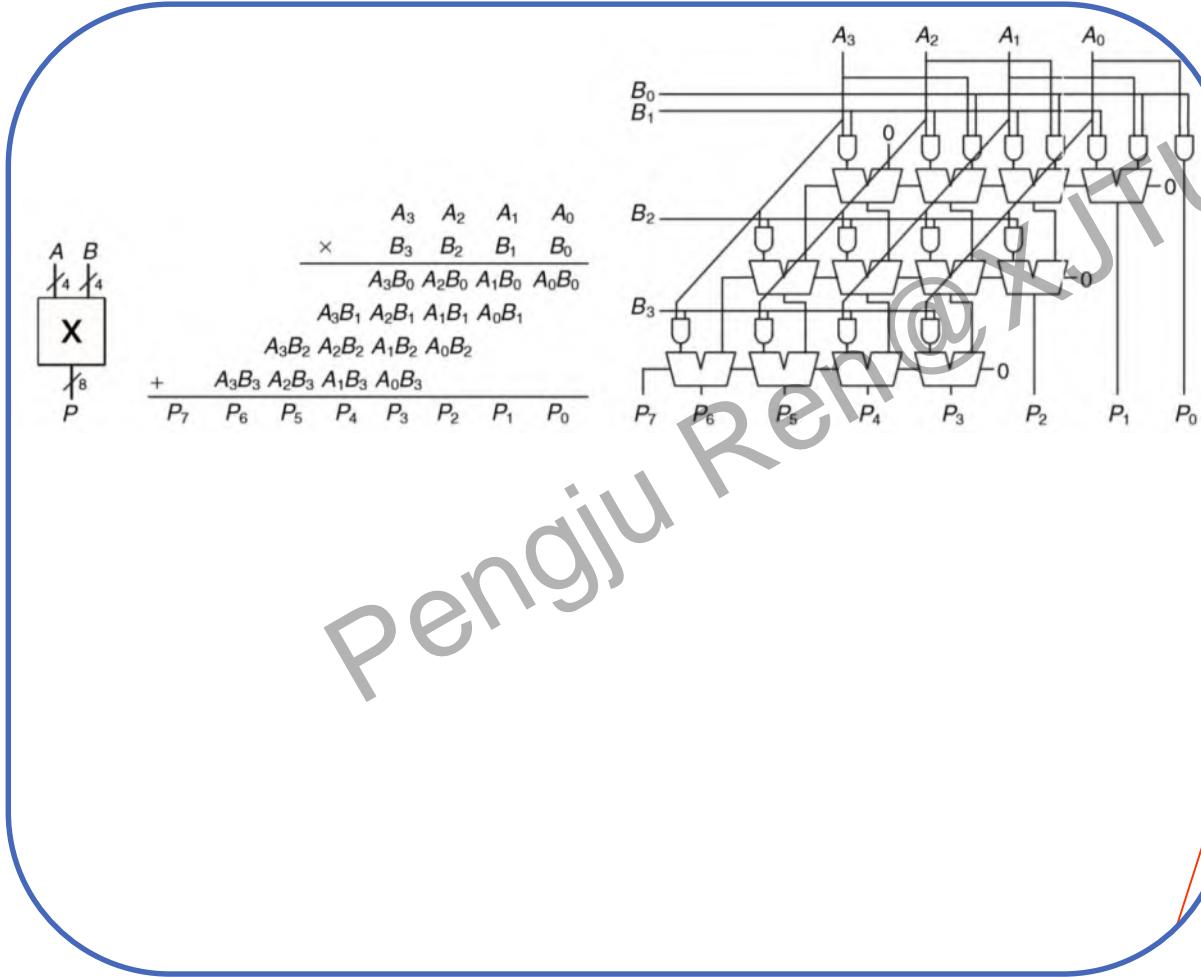
programs
device drivers
instructions
registers
datapaths
controllers
adders
memories
AND gates
NOT gates
amplifiers
filters
transistors
diodes
electrons

Recap(Digital Logic and Circuits)



programs
device drivers
instructions
registers
datapaths
controllers
adders
memories
AND gates
NOT gates
amplifiers
filters
transistors
diodes
electrons

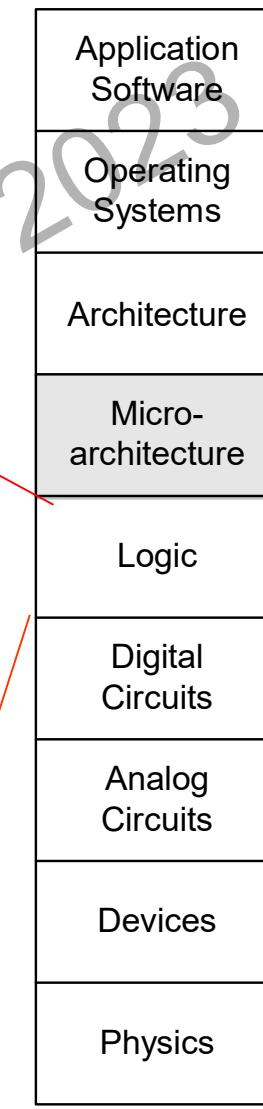
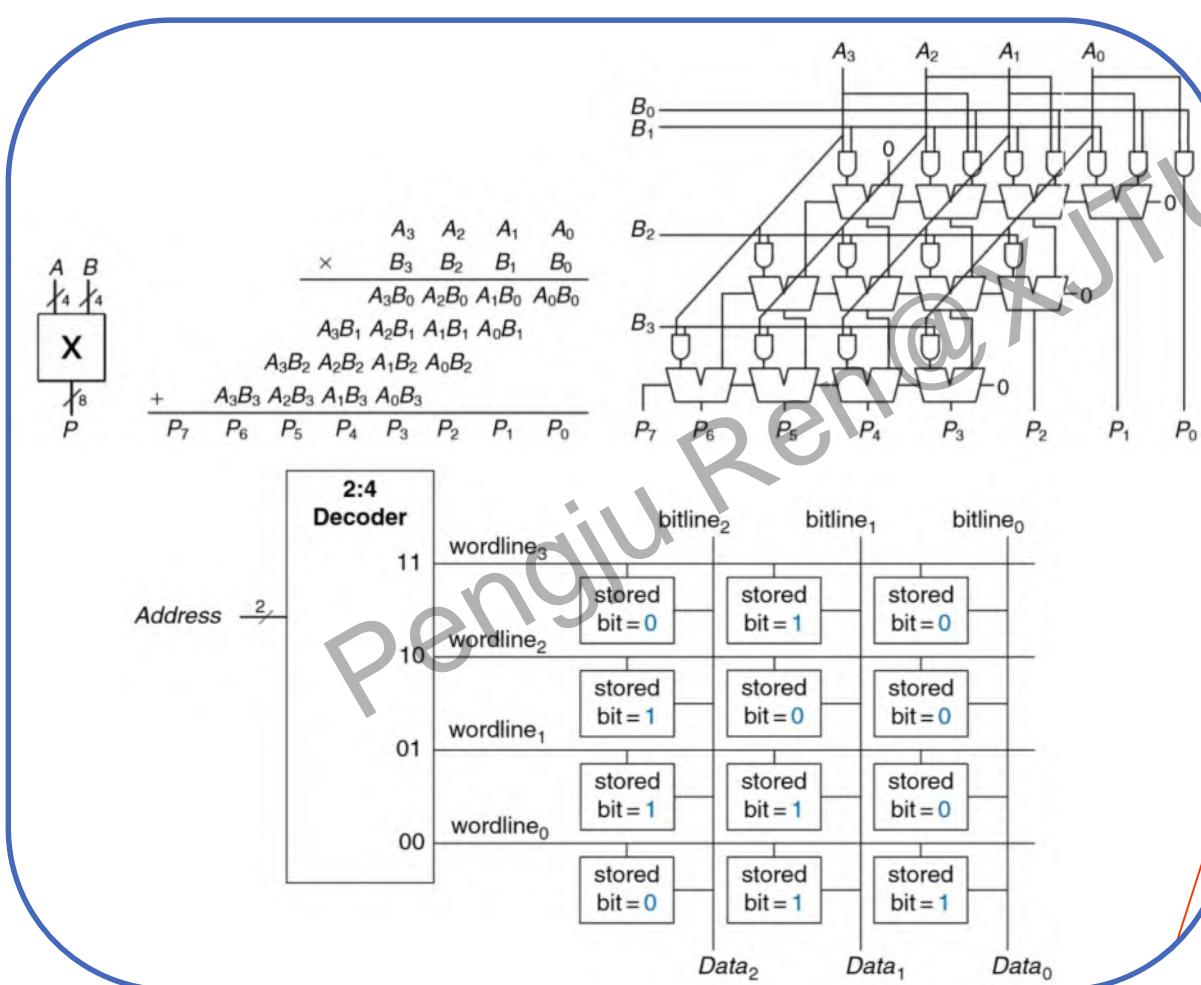
Recap(Digital Logic and Circuits)



Corresponding components for each layer:

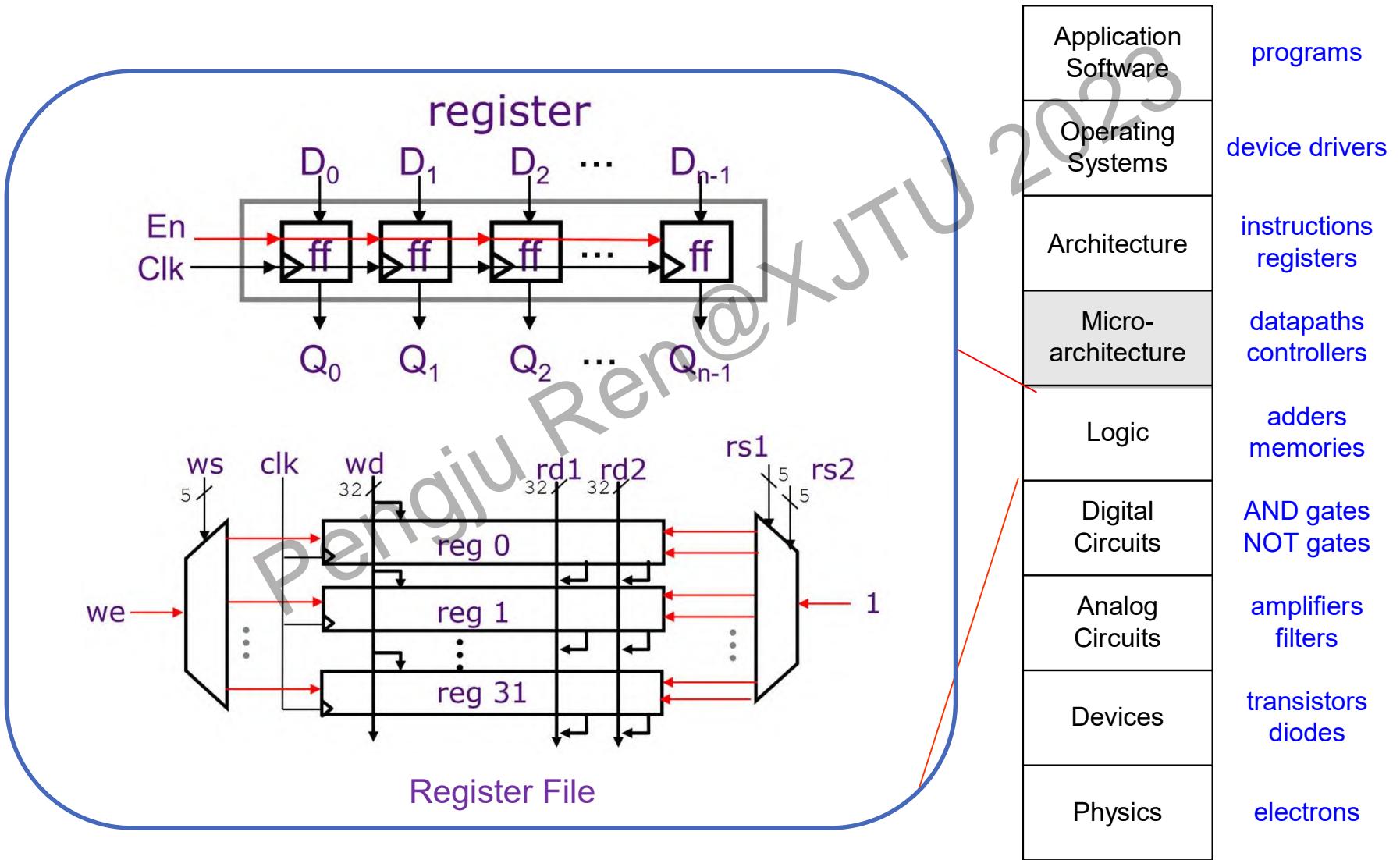
- programs
- device drivers
- instructions registers
- datapaths controllers
- adders memories
- AND gates NOT gates
- amplifiers filters
- transistors diodes
- electrons

Recap(Digital Logic and Circuits)

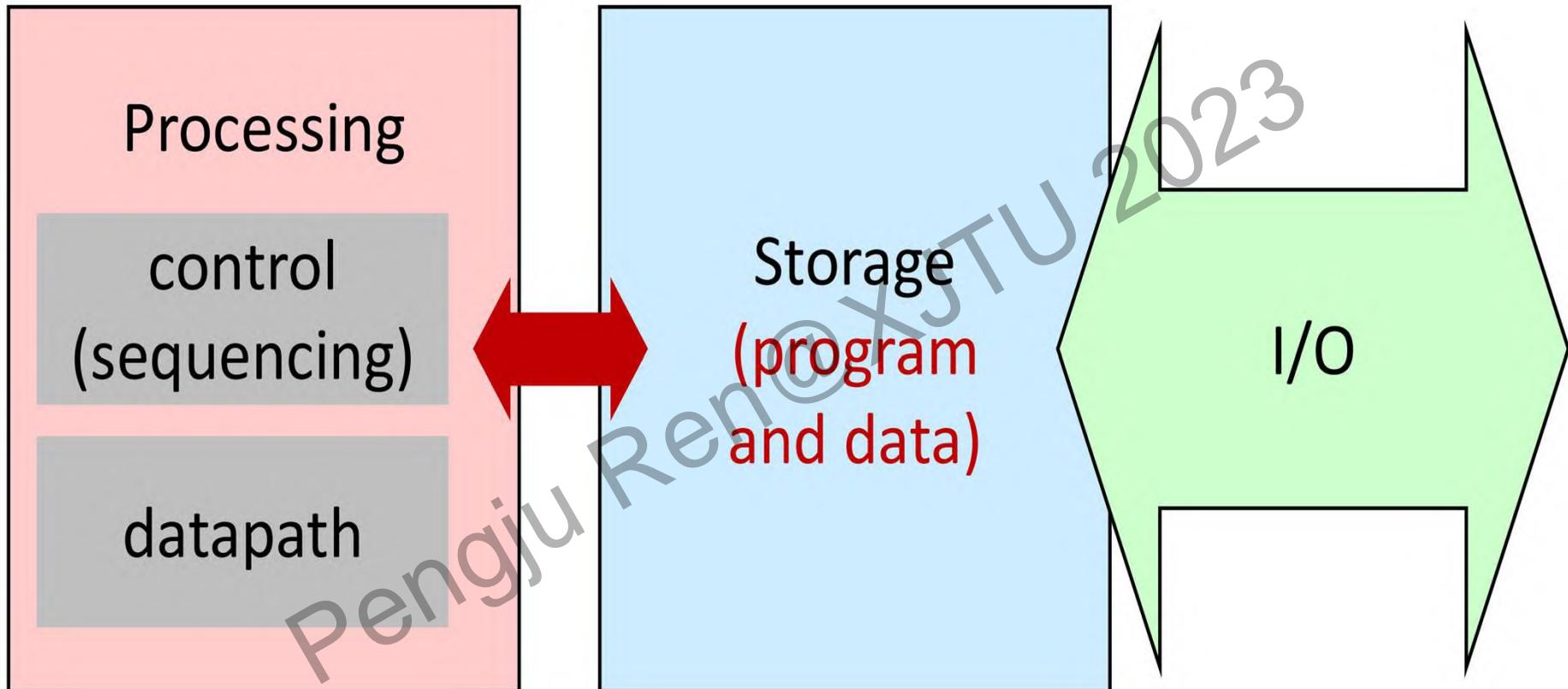


programs
device drivers
instructions
registers
datapaths
controllers
adders
memories
AND gates
NOT gates
amplifiers
filters
transistors
diodes
electrons

Recap(Digital Logic and Circuits)

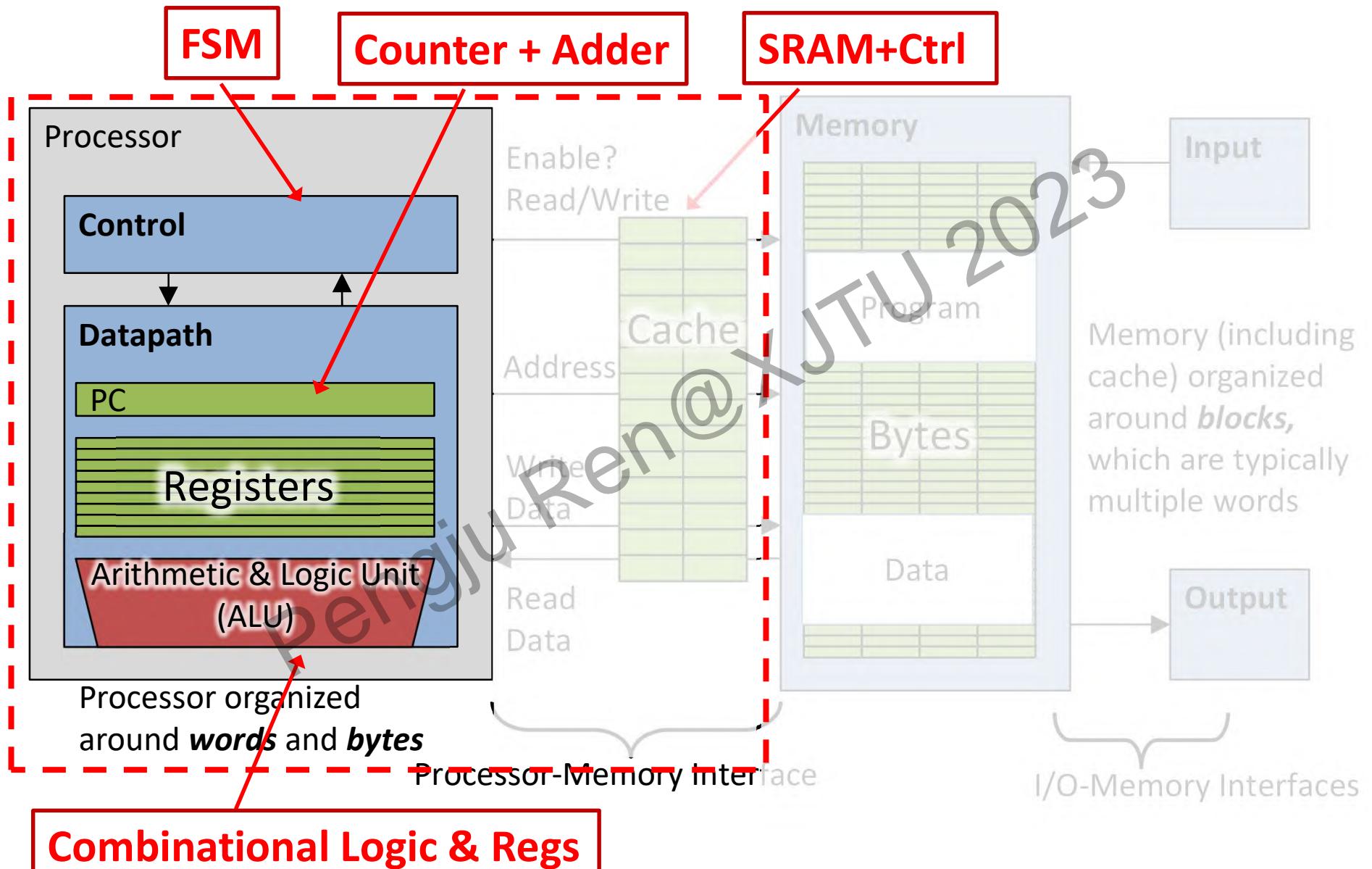


Abstract View of a Simple Processor



Having program stored as data is an extremely important step
in the evolution of computer architectures

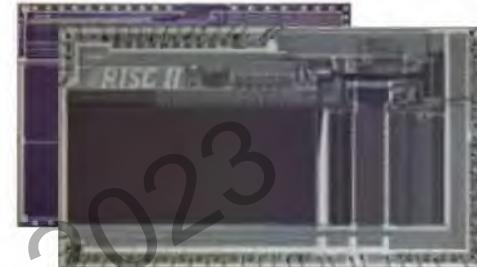
Abstract View of a Simple Processor



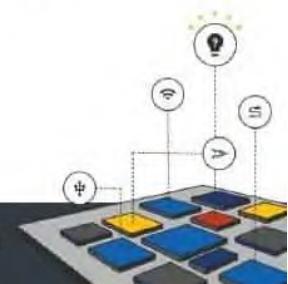
What is RISC-V ? And Why?

What is RISC-V

RISC-V is a high-quality, license-free, royalty-free ISA



- 5th Generation RISC design from UC Berkeley
- A high-quality, license-free, royalty-free RISC ISA specification
- Standard maintained by non-profit RISC-V Foundation
- Multiple proprietary and open-source core implementations
- Supported by growing software ecosystem
 - binutils/gcc/FreeBSD mainlined, Linux/glibc submitted to upstream
- Appropriate for all levels of computing system, from microcontrollers to supercomputers



RISC-V



Berkeley
Architecture
Research

D R A P E R bluespec



D O V E R

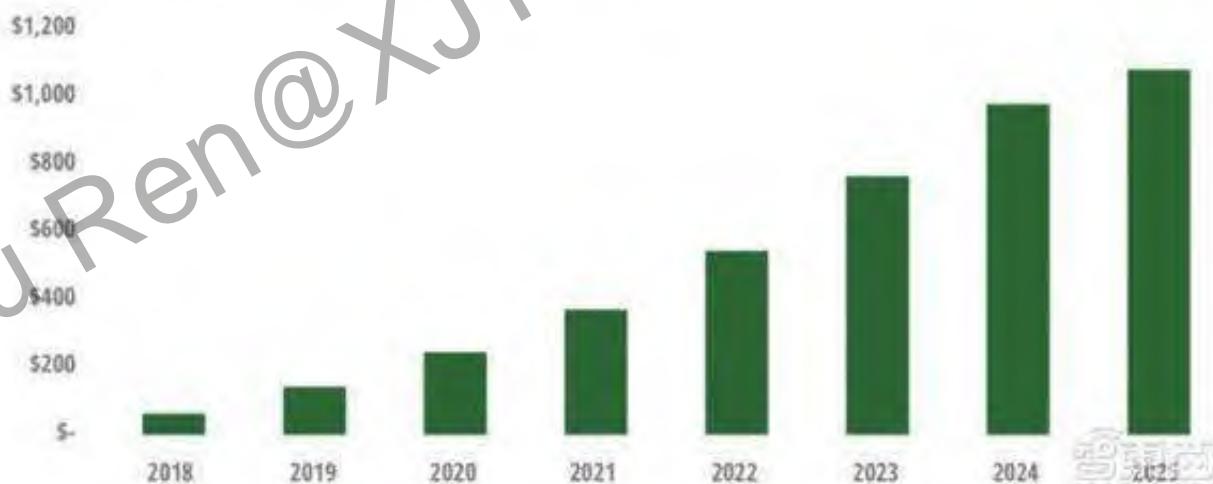
“三分天下有其一！”

2k+ RISC-V Members
across 70 Countries



RISC-V revenue is on track for exponential growth

Total RISC-V market revenue, 2018–2025 (US\$ millions)



- 截至2021年12月，RISC-V International基金会集团成员比年初增长了130%，达到24278名，企业成员来自70多个国家，数量增长至2000多家；
- 已有近100款不同门类及型号的RISC-V芯片应用于云端、移动、高性能计算和机器学习，全球范围内产出的RISC-V核累计超过20亿颗

RISC-V ISA

- New fifth-generation RISC design from UC Berkeley
- Realistic & complete ISA, but open & small
- Not over-architected for a certain implementation style
- Both 32-bit (RV32) and 64-bit (RV64) address-space variants
- Designed for multiprocessing
- Efficient instruction encoding
- Easy to subset/extend for education/research
- RISC-V spec available on Foundation website and github
- Increasing momentum with industry adoption

This course uses 32bit-version (RV32) for demonstration

How to Build a RISC-V Processor

Each instruction reads and updates this state during execution:

- **Registers ($x_0 \dots x_{31}$)**

- 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
- x_0 is always 0 (writes to **Reg [0]** are ignored)

- **Program counter (PC)**

- Holds address of current instruction

- **Memory (MEM)**

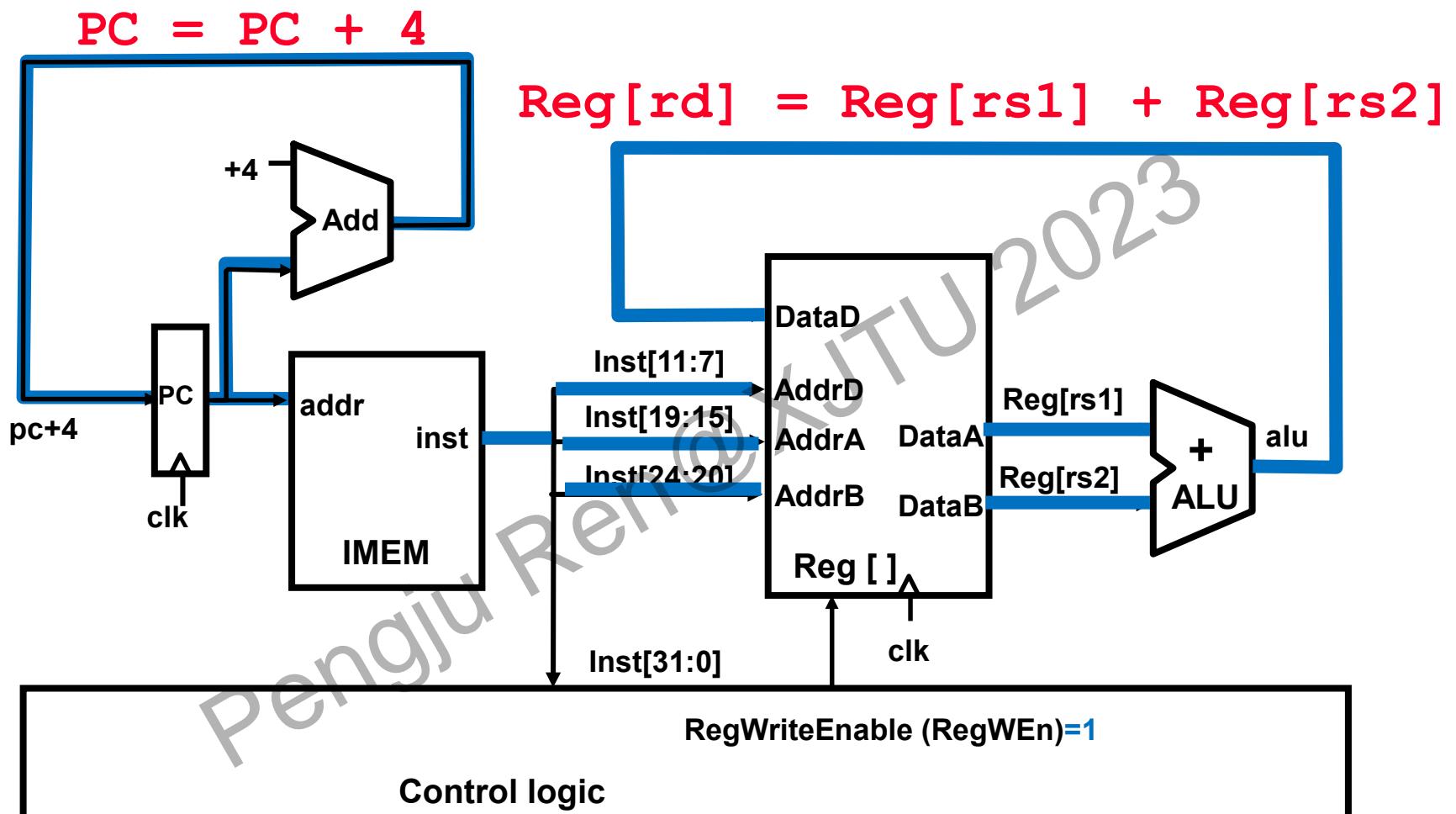
- Holds both instructions & data, in one 32-bit byte-addressed memory space
- We'll use separate memories for instructions (**IMEM**) and data (**DMEM**)
 - *These are placeholders for instruction and data caches*
- Instructions are read (*fetched*) from instruction memory
- Load/store instructions access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

Recap: RISC-V Instruction Encoding(ISA)

	31	27	26	25	24	20	19	15	14	12	11	7	6	0
R		funct7			rs2	rs1		funct3		rd		Opcode		
I		imm[11:0]				rs1		funct3		rd		Opcode		
S	imm[11:5]				rs2	rs1	funct3		imm[4:0]		opcode			
SB	imm[12 10:5]				rs2	rs1	funct3		imm[4:1 11]		opcode			
U		imm[31:12]								rd		opcode		
UJ		imm[20 10:1 11 19:12]								rd		opcode		

- **R-Format:** instructions using 3 register inputs
 - add, xor, mul — arithmetic/logical ops
- **I-Format:** instructions with immediates, loads
 - addi, lw, jalr, slli
- **S-Format:** store instructions: sw, sb
- **SB-Format:** branch instructions: beq, bge
- **U-Format:** instructions with upper immediates
 - lui, auipc — upper immediate is 20-bits
- **UJ-Format:** jump instructions: jal

Implementing the add instruction



31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	000	rd	Reg-Reg	OP

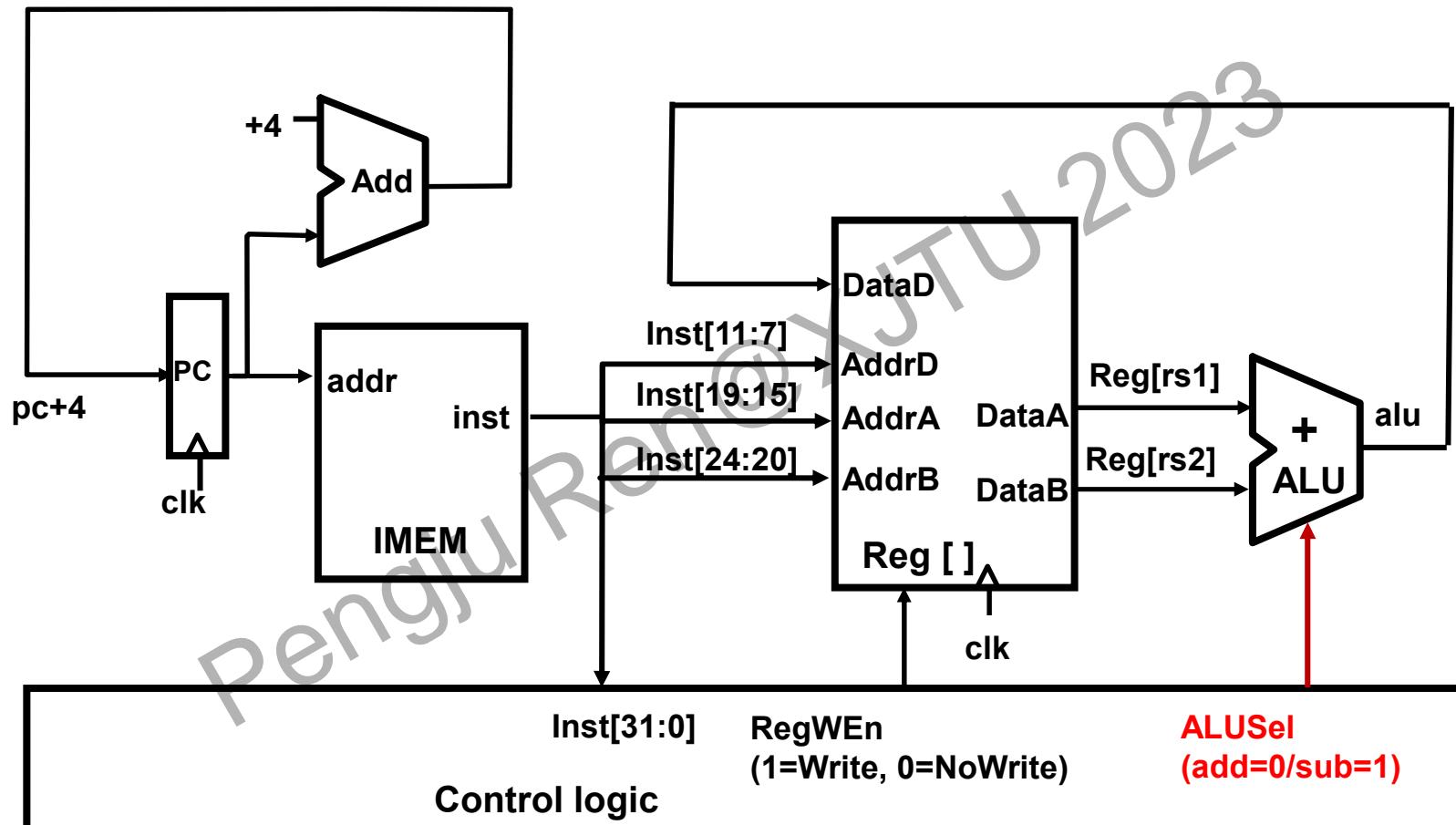
Implementing the sub instruction

31	2524	2019	1514	1211	76	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

31	2524	21 2019	1514	1211	8 7 6	0	R-Type
funct7	rs2	rs1	funct3	rd	opcode		
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

Implementing other R-Format instructions

31	2524	21	2019	1514	1211	8	7	6	0	R-Type
funct7	rs2	rs1		funct3	rd	rd	rd	rd	rd	
0000000				add R[rd]=R[rs1]+R[rs2]	000	rd	01	10011		add
0100000				sub R[rd]=R[rs1]-R[rs2]	000	rd	01	10011		sub
0000000				sll R[rd]=R[rs1]<<R[rs2]	001	rd	01	10011		sll
0000000				sltu R[rd]=(R[rs1]<R[rs2])? 1:0	010	rd	01	10011		slt
0000000				xor R[rd]=R[rs1]^R[rs2]	011	rd	01	10011		sltu
0000000				srl R[rd]=R[rs1]>>R[rs2] (Shift right)	100	rd	01	10011		xor
0000000				sra R[rd]=R[rs1]>>R[rs2] (Shift right Arithmetic)	101	rd	01	10011		srl
0100000				or R[rd]=R[rs1] R[rs2]	101	rd	01	10011		sra
0000000				and R[rd]=R[rs1] & R[rs2]	110	rd	01	10011		or
0000000					111	rd	01	10011		and

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

Recap: RISC-V Instruction Encoding(ISA)

	31	27	26	25	24	20	19	15	14	12	11	7	6	0
R		funct7			rs2		rs1		funct3		rd		Opcode	
I			imm[11:0]			rs1		funct3		rd		Opcode		
S		imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		
SB	imm[12 10:5]			rs2		rs1		funct3	imm[4:1 11]			opcode		
U		imm[31:12]								rd		opcode		
UJ	imm[20 10:1 11 19:12]									rd		opcode		

- **R-Format:** instructions using 3 register inputs
 - add, xor, mul — arithmetic/logical ops
- **I-Format:** instructions with immediates, loads
 - addi, lw, jalr, slli
- **S-Format:** store instructions: sw, sb
- **SB-Format:** branch instructions: beq, bge
- **U-Format:** instructions with upper immediates
 - lui, auipc — upper immediate is 20-bits
- **UJ-Format:** jump instructions: jal

Recap: RISC-V Instruction Encoding(ISA)

	31	27	26	25	24	20	19	15	14	12	11	7	6	0
R		funct7			rs2		rs1		funct3		rd		Opcode	
I			imm[11:0]			rs1		funct3		rd		Opcode		
S		imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		
SB	imm[12 10:5]			rs2		rs1		funct3	imm[4:1 11]			opcode		
U		imm[31:12]								rd		opcode		
UJ	imm[20 10:1 11 19:12]									rd		opcode		

- **R-Format:** instructions using 3 register inputs
 - add, xor, mul — arithmetic/logical ops
- **I-Format:** instructions with immediates, loads
 - addi, lw, jalr, slli

addi $R[rd]=R[rs1]+imm$

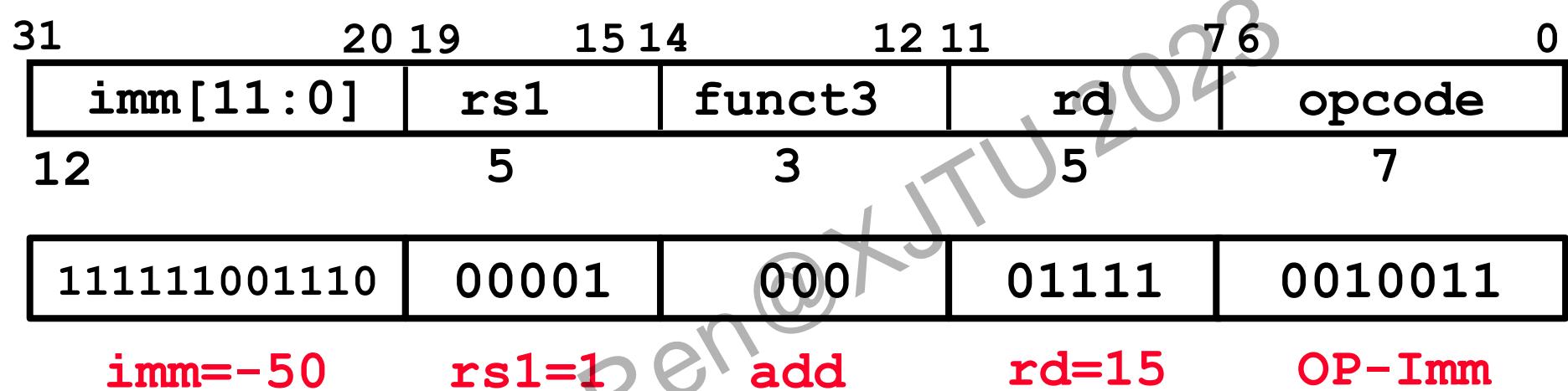
lw $R[rd]=\{32'bM[](31), M[R[rs1]+imm]\}$

jalr $R[rd]=PC+4; PC=R[rs1]+imm$ (jump and link register for case/switch)

slli $R[rd]=R[rs1]<<imm$ (shift left Immediate(word))

* UJ-Format: jump instructions: jal

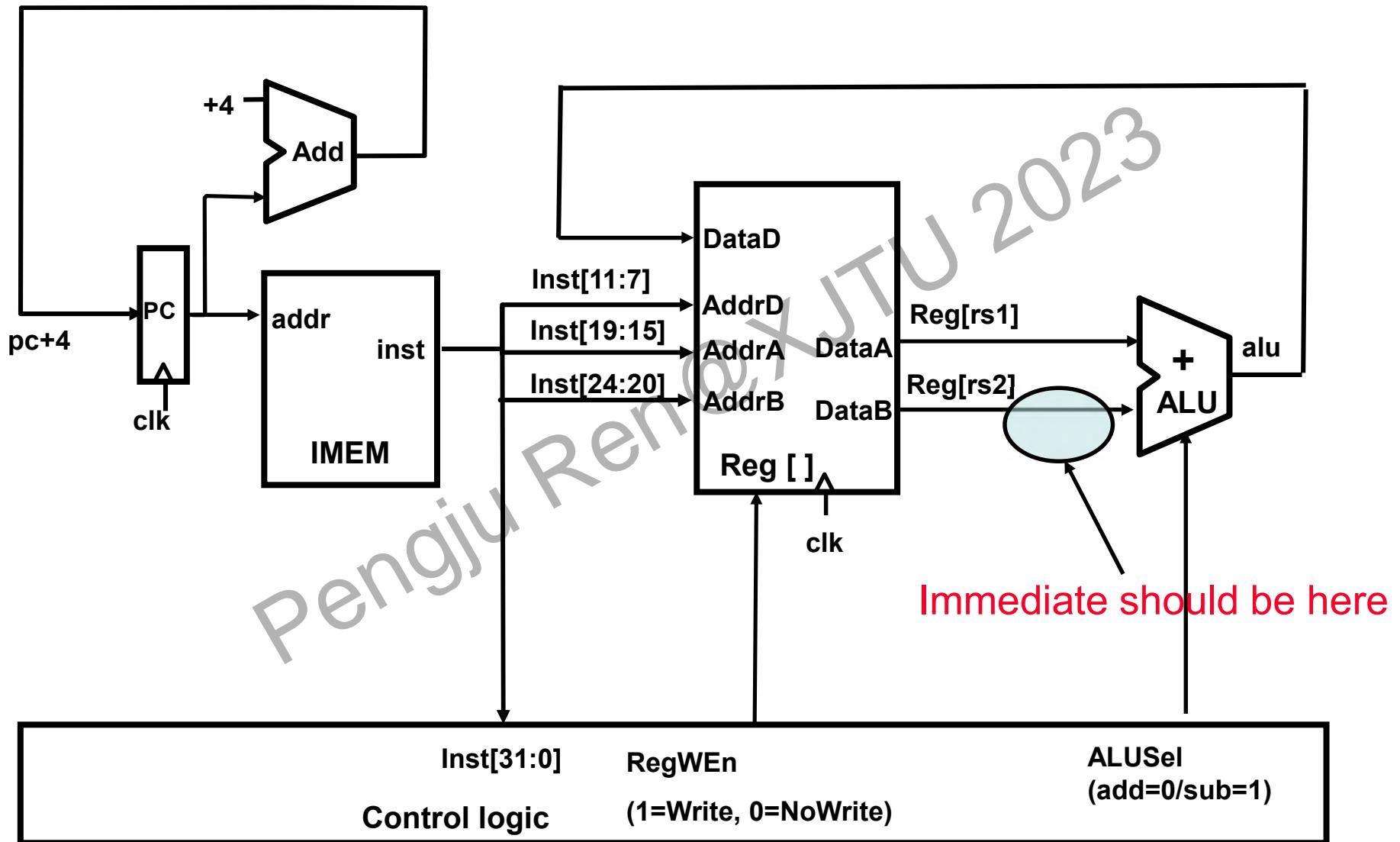
Implementing I-Format - addi instruction



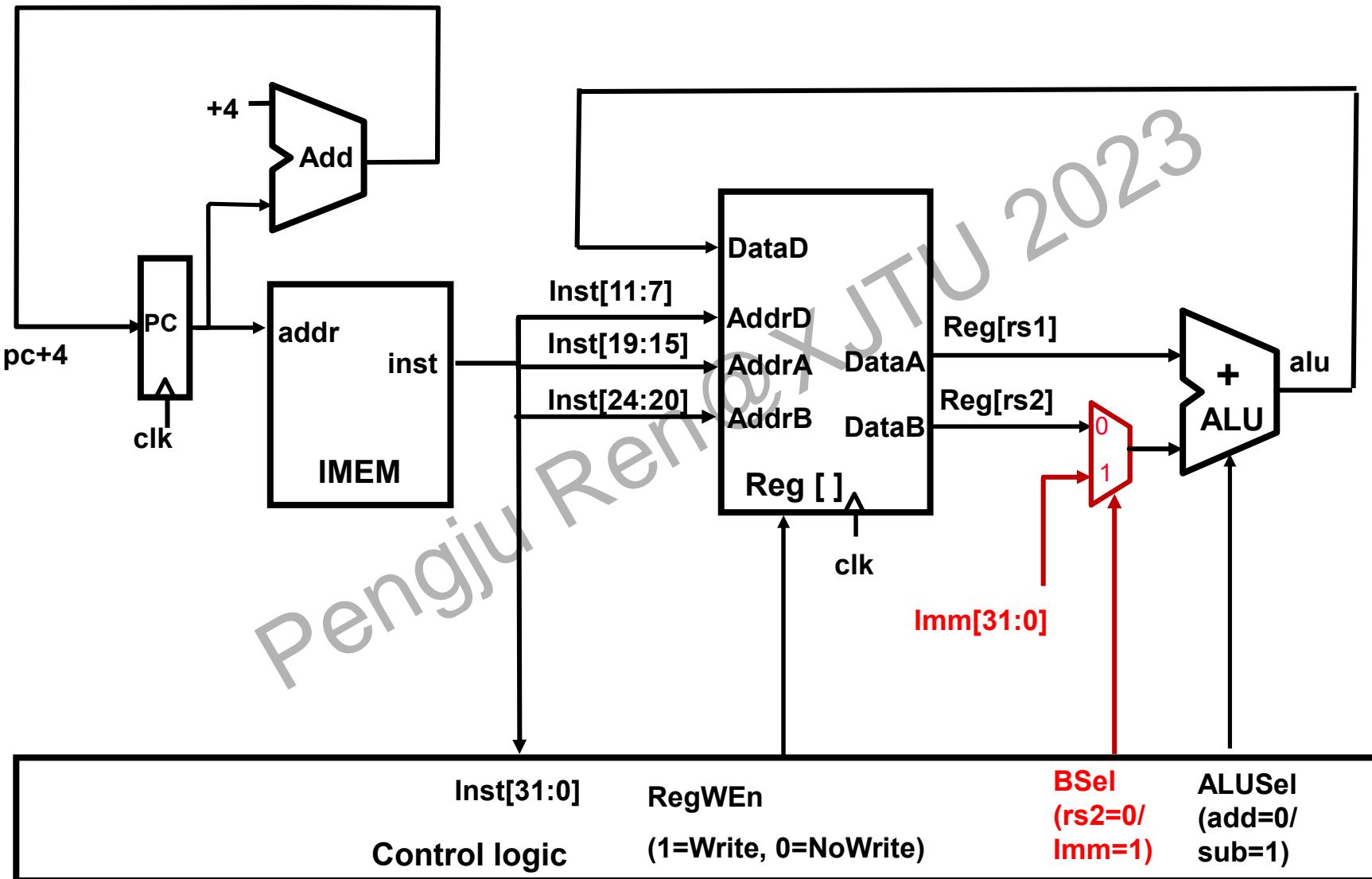
- RISC-V Assembly Instruction:

addi x15, x1, -50

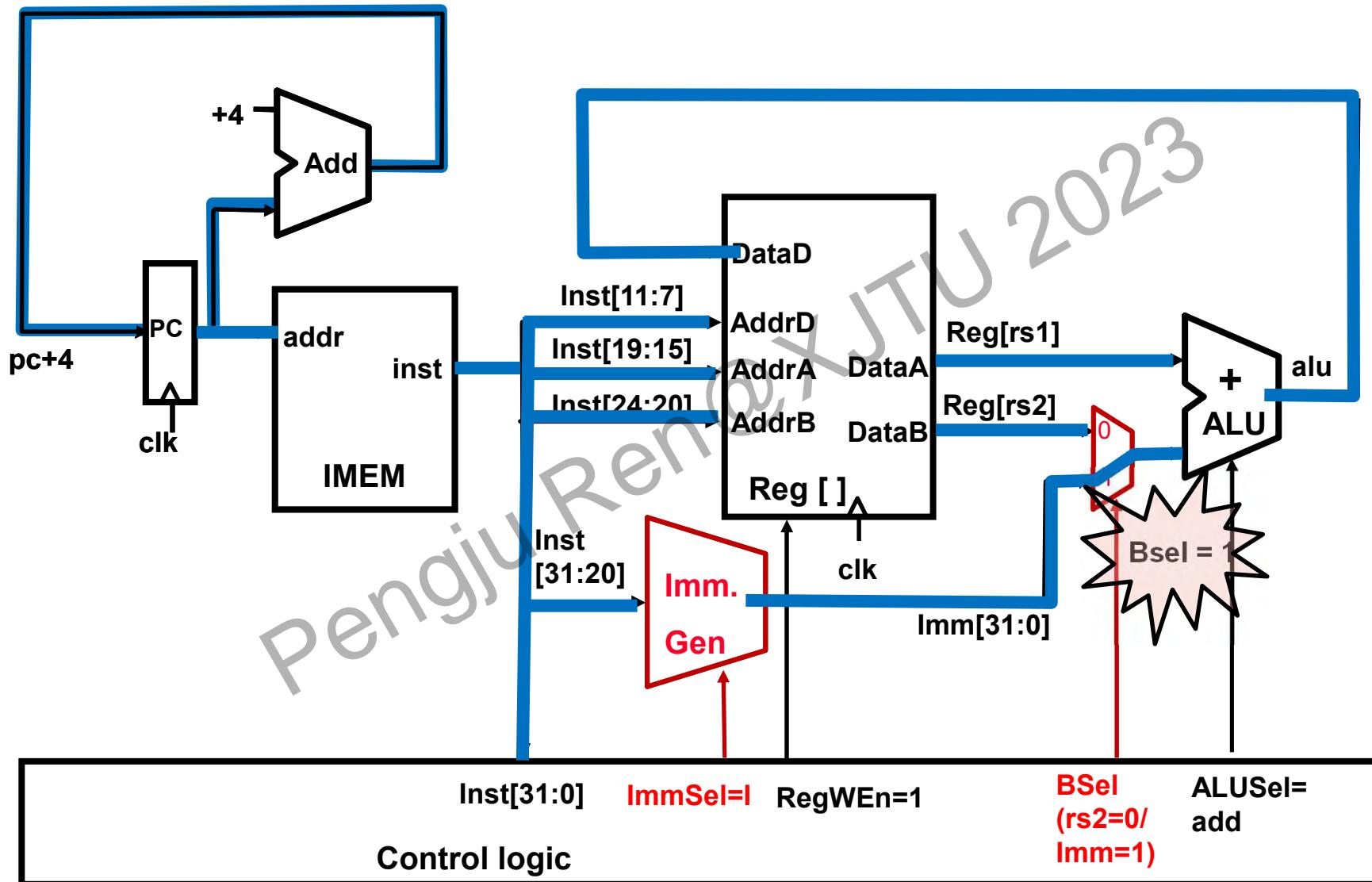
Datapath for add/sub



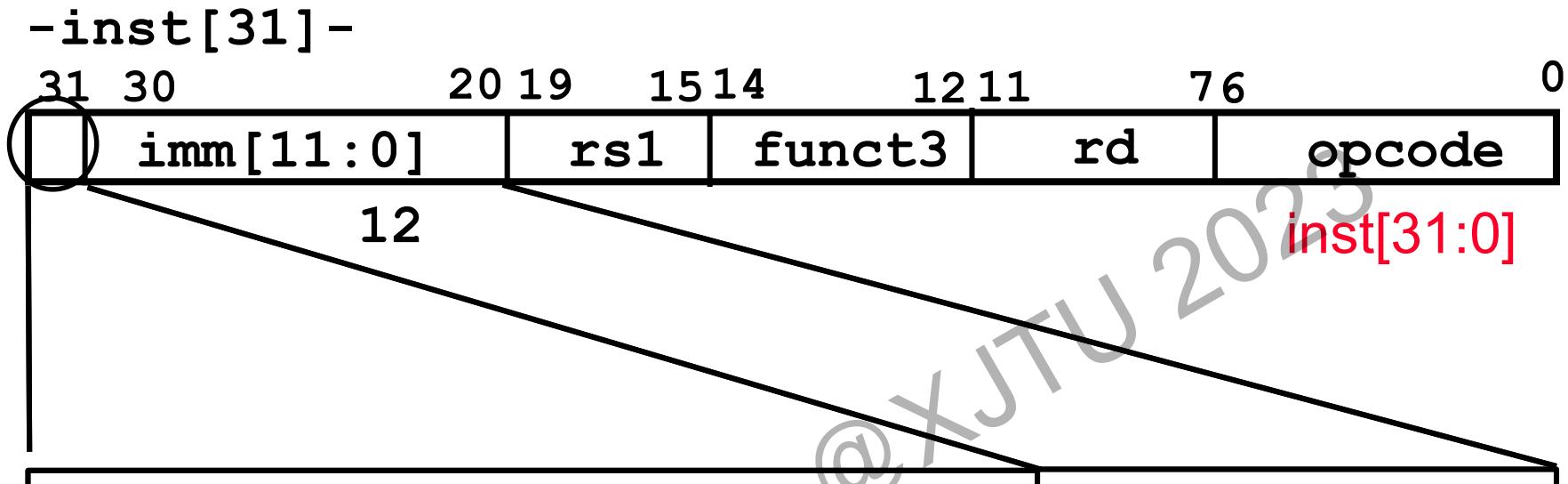
Adding addi to Datapath(1)



Adding addi to Datapath(2)

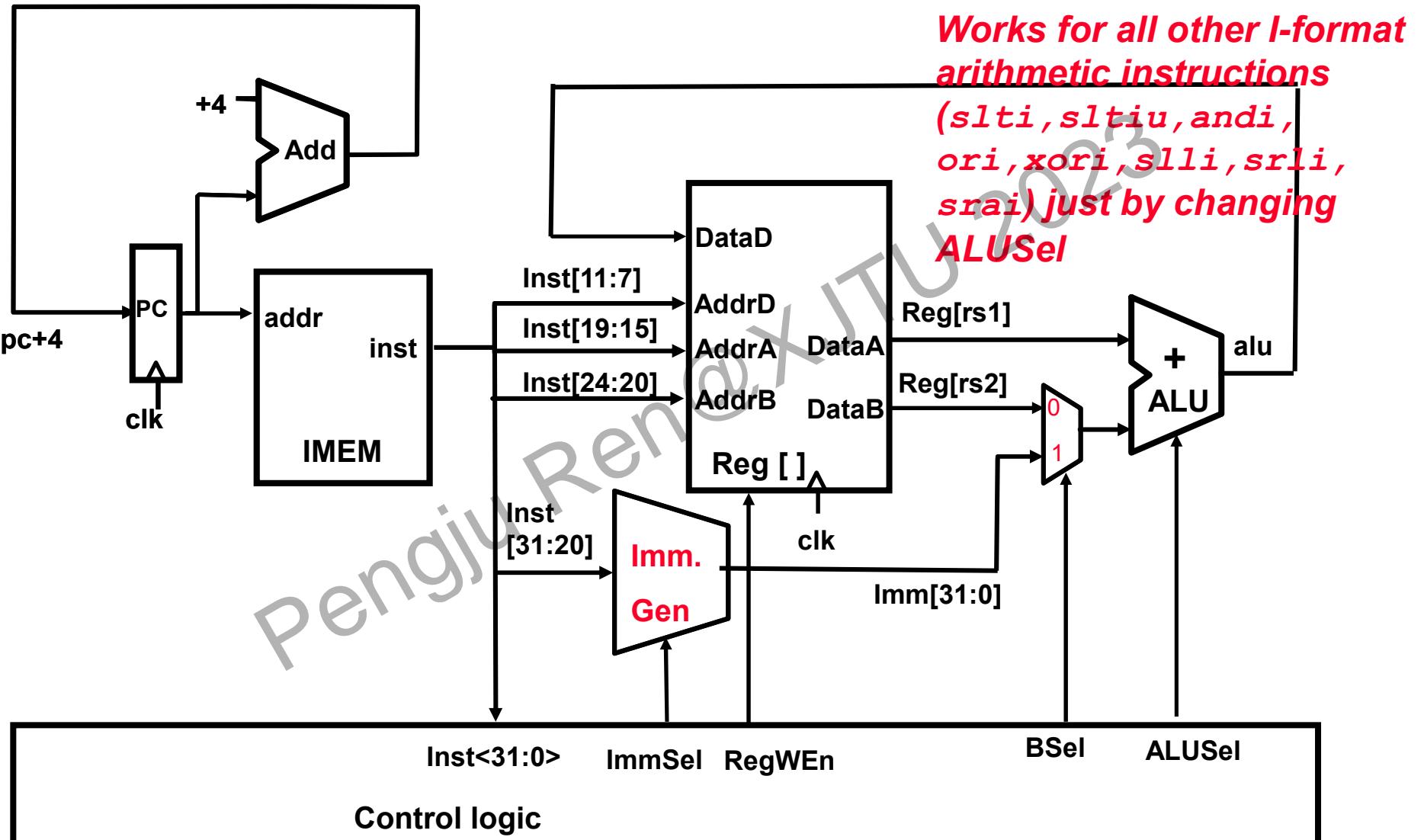


I-Format immediates



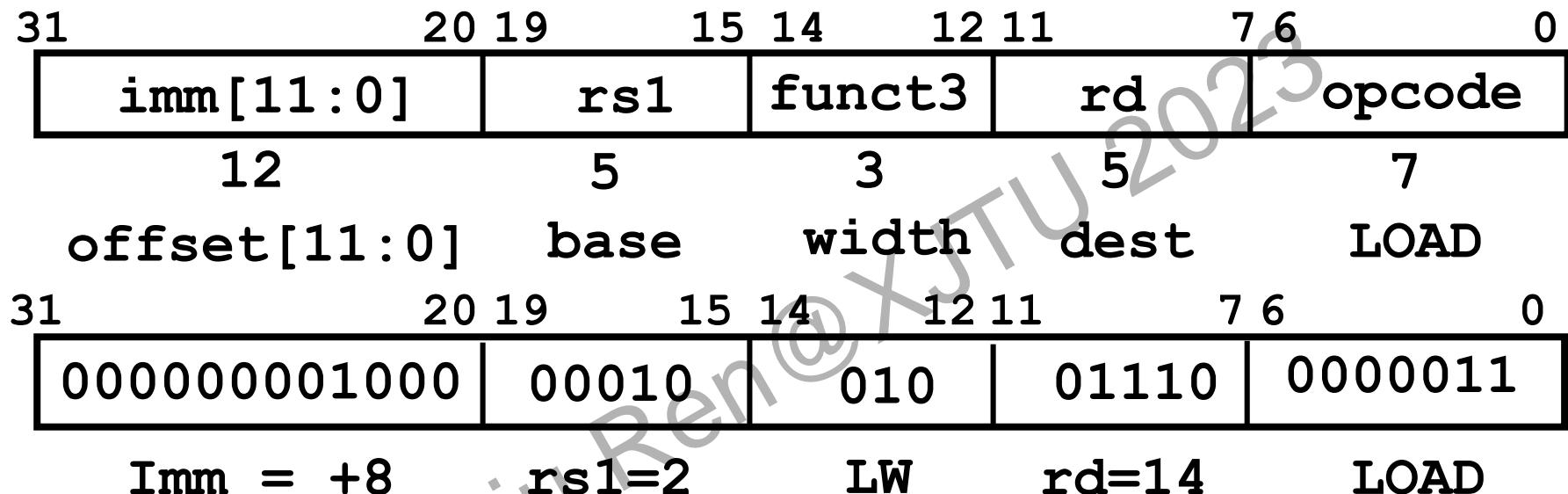
- inst[31:20]
- Imm. Gen
- ImmSel=1
- imm[31:0]
- High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
 - Immediate is **sign-extended** by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])

R+I Datapath



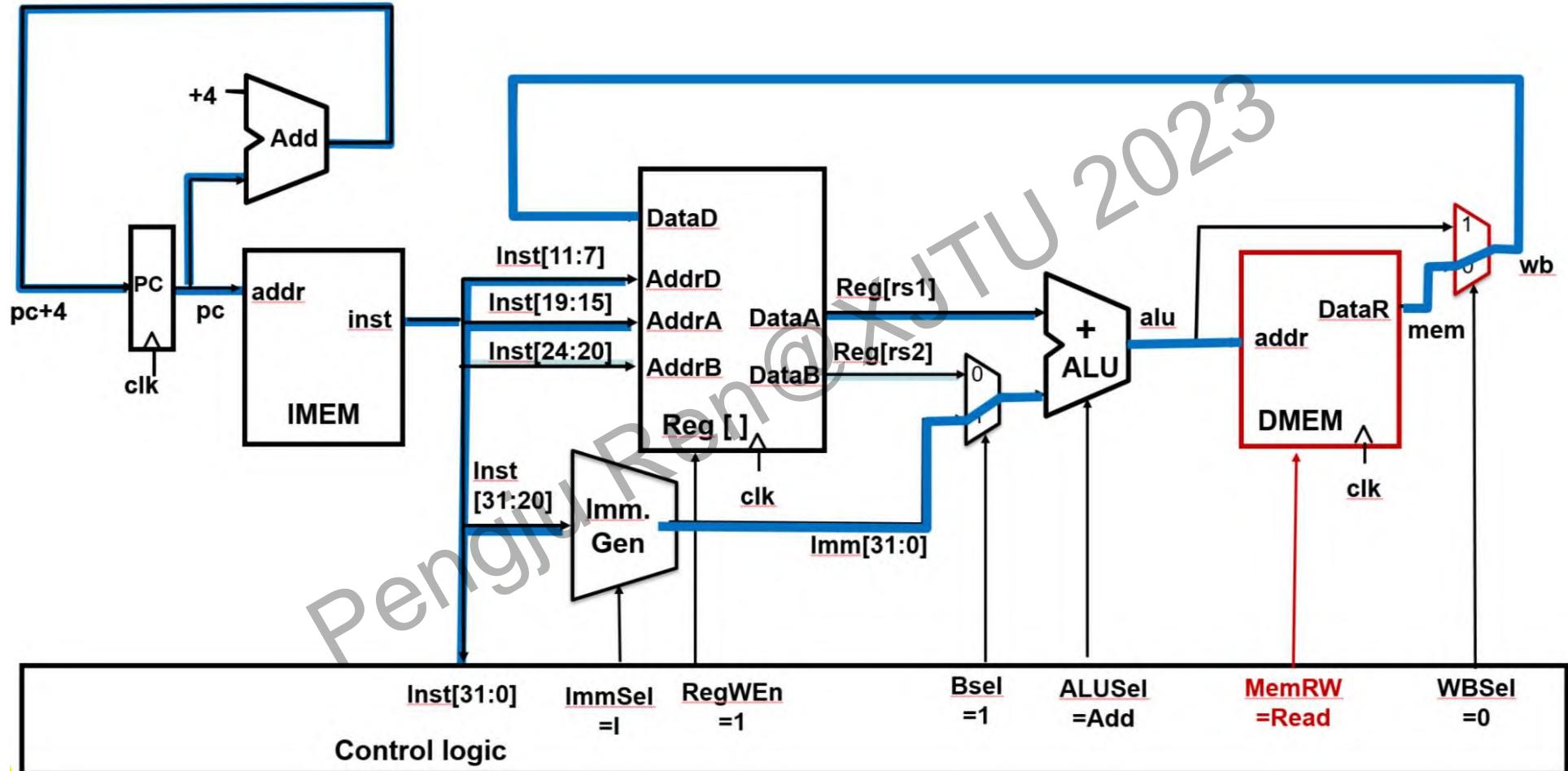
Add lw

- 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

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

Recap: RISC-V Instruction Encoding(ISA)

	31	27	26	25	24	20	19	15	14	12	11	7	6	0
R		funct7			rs2		rs1		funct3		rd		Opcode	
I				imm[11:0]				rs1		funct3		rd		Opcode
S		imm[11:5]			rs2		rs1		funct3		imm[4:0]		opcode	
SB		imm[12 10:5]			rs2		rs1		funct3		imm[4:1 11]		opcode	
U				imm[31:12]							rd		opcode	
UJ				imm[20 10:1 11 19:12]							rd		opcode	

- **R-Format:** instructions using 3 register inputs
 - add, xor, mul — arithmetic/logical ops
- **I-Format:** instructions with immediates, loads
 - addi, lw, jalr, slli
- **S-Format: store instructions:** sw, sb
- **SB-Format:** branch instructions: beq, bge
- **U-Format:** instructions with upper immediates
 - lui, auipc — upper immediate is 20-bits
- **UJ-Format:** jump instructions: jal

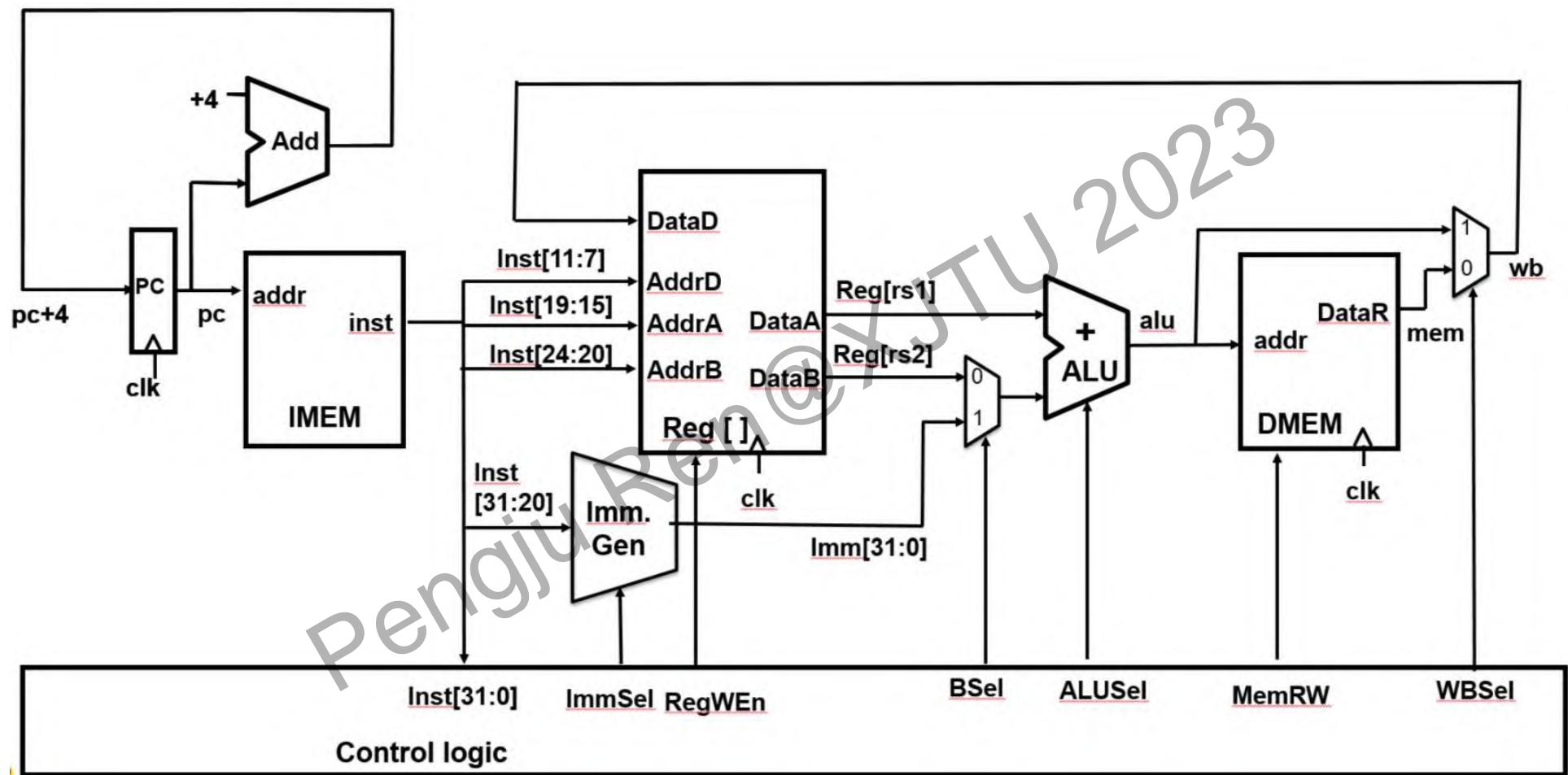
Adding sw Instruction

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

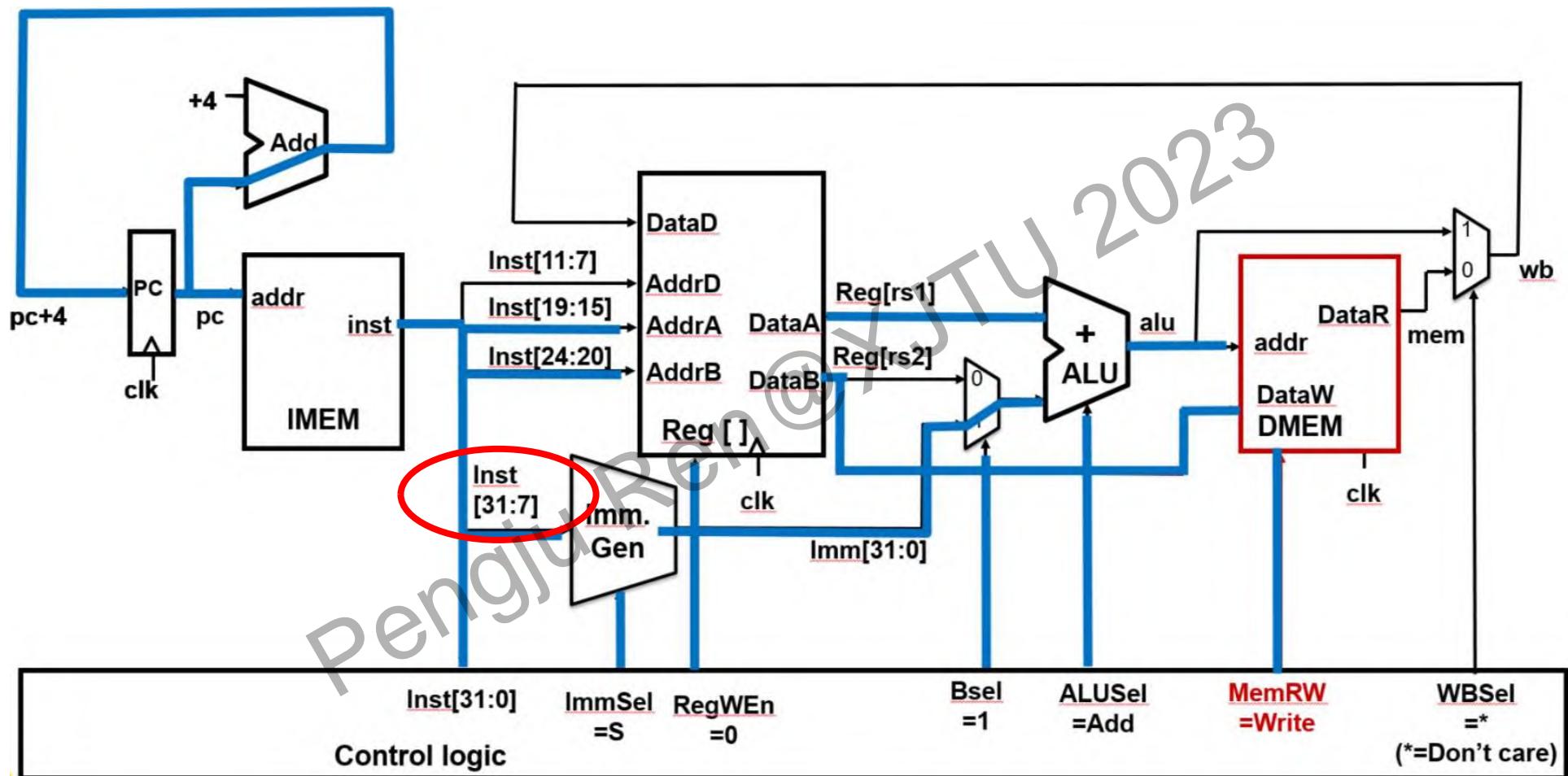
sw x14, 8(x2)

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				

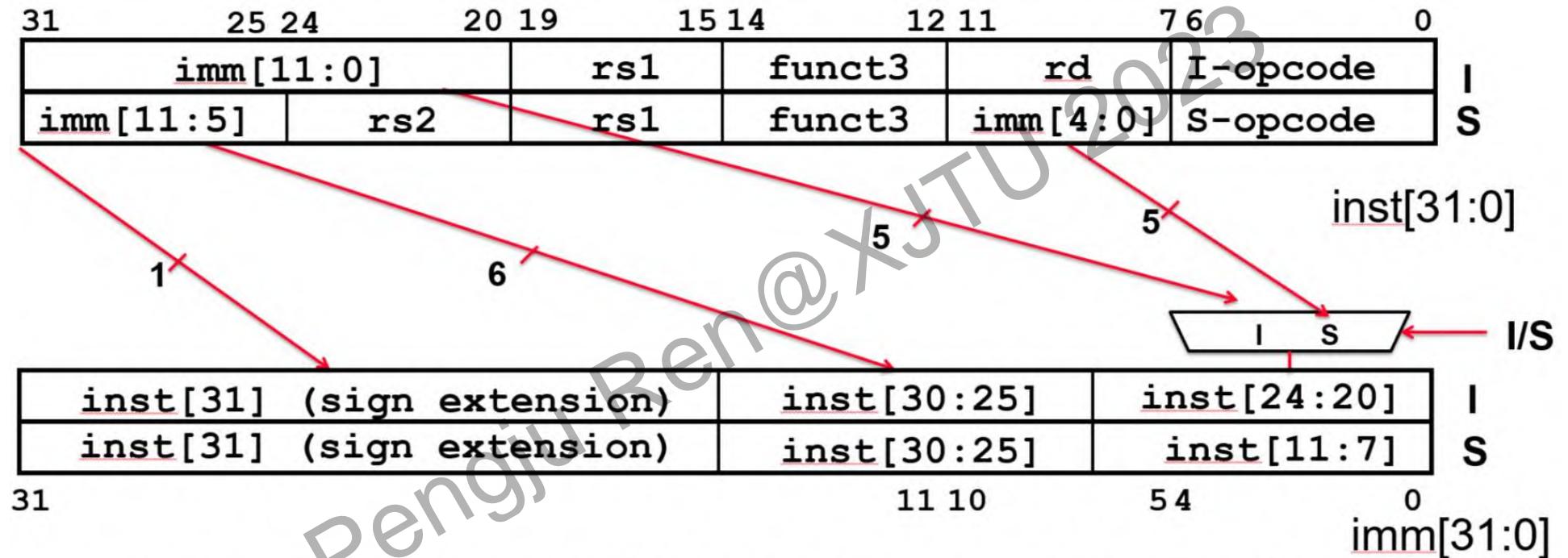
Datapath with 1w



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

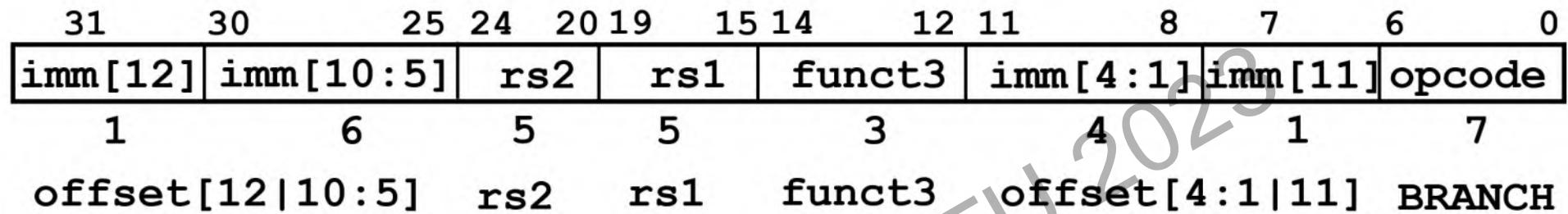
Control Flow Instructions

- C-Code

```
{ code A }  
if X==Y then  
    { code B }  
else  
    { code C }  
{ code D }
```

basic blocks (1-way in, 1-way out, all or nothing)

Implementing (Conditional) Branches

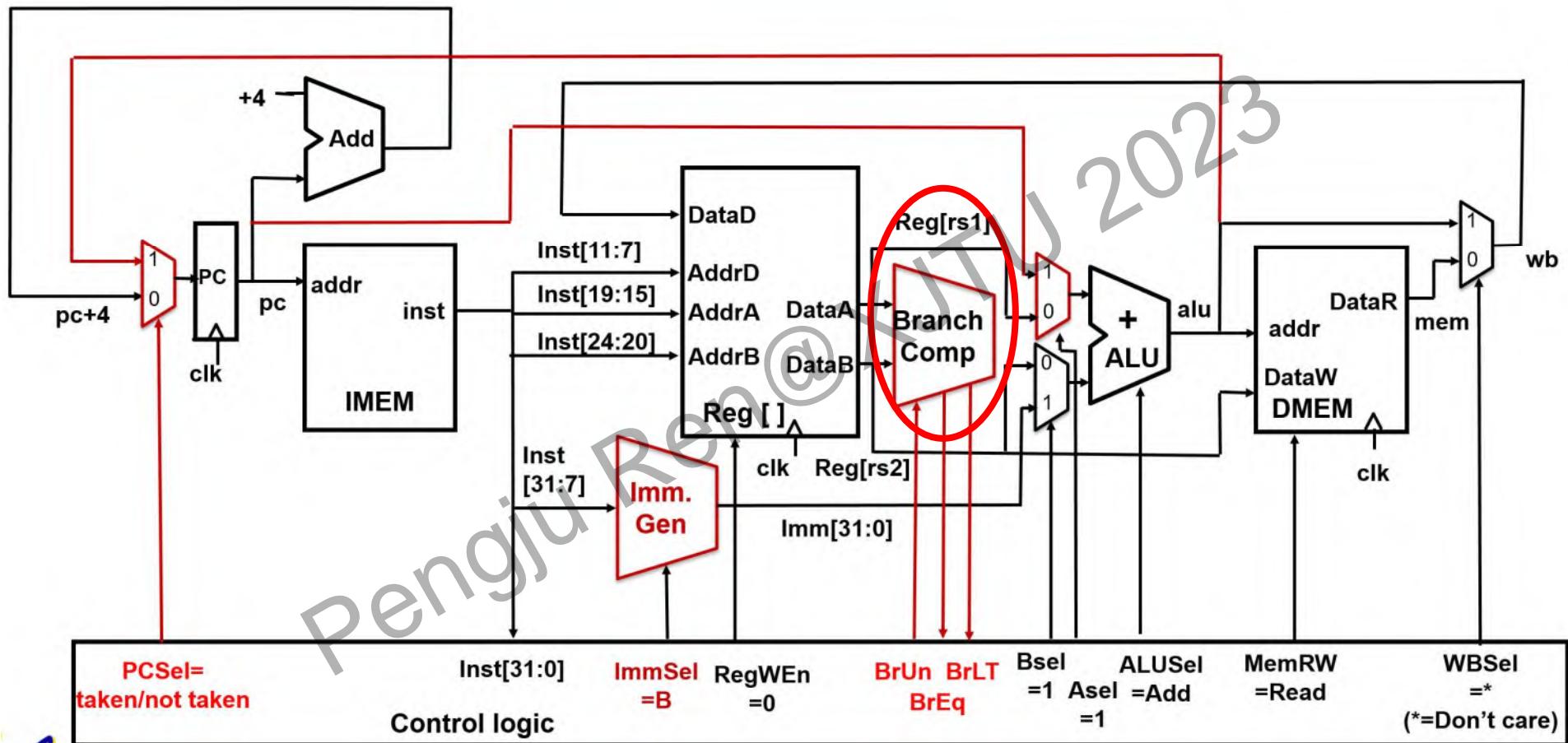


- B-format is **PC-relative addressing** (typically used for loops, *if-else*, *while*, *for*), take two register sources (rs1/rs2) to compare, and a 12-bit immediate(+/-) to specify an address to go to
- Notice that, Memory is “**byte-addressed**”, Instructions are “**word-aligned**”(4-bytes). Immediate represents values -4096 to +4094 in 2-byte increments to support 16-bit compressed instructions
- The 12 immediate bits encode *even* 13-bit signed byte offsets (**lowest bit of offset is always zero**, so no need to store it)

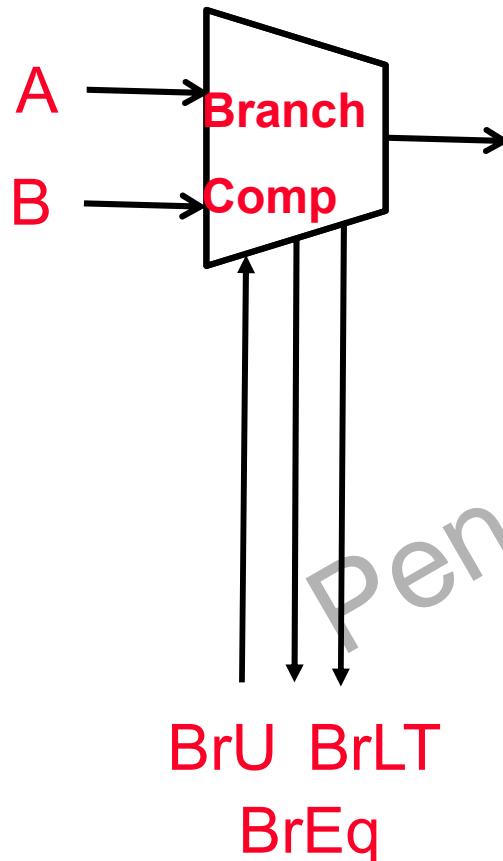
To Add Branches

- Different change to the state:
 - $\text{PC} = \text{PC} + 4$, *branch not taken*
 - $\text{PC} + \text{immediate}$, *branch taken*
- Six branch instructions: BEQ, BNE, BLT, BGE, BLTU, BGEU
- Need to compute $\text{PC} + \text{immediate}$ and to compare values of rs1 and rs2
 - But have only one ALU – need more hardware

Adding Branches



Branch Comparator



- $\text{BrEq} = 1$, if $A=B$
- $\text{BrLT} = 1$, if $A < B$
- $\text{BrUn} = 1$ selects unsigned comparison for BrLT , $0 = \text{signed}$
- $\text{BGE} = 1$, if $A \geq B$

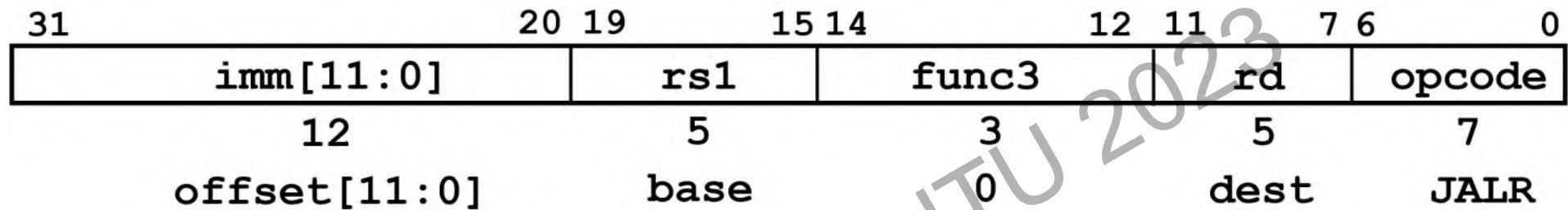
RISC-V Immediate Encoding

Instruction encodings, $\text{inst}[31:0]$													0	
31	30	25	24	20	19	15	14	12	11	8	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode			R-type	
						imm[11:0]		rs1		funct3		rd	opcode	I-type
imm[11:5]		rs2		rs1		funct3			imm[4:0]		opcode		S-type	
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode			B-type	

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

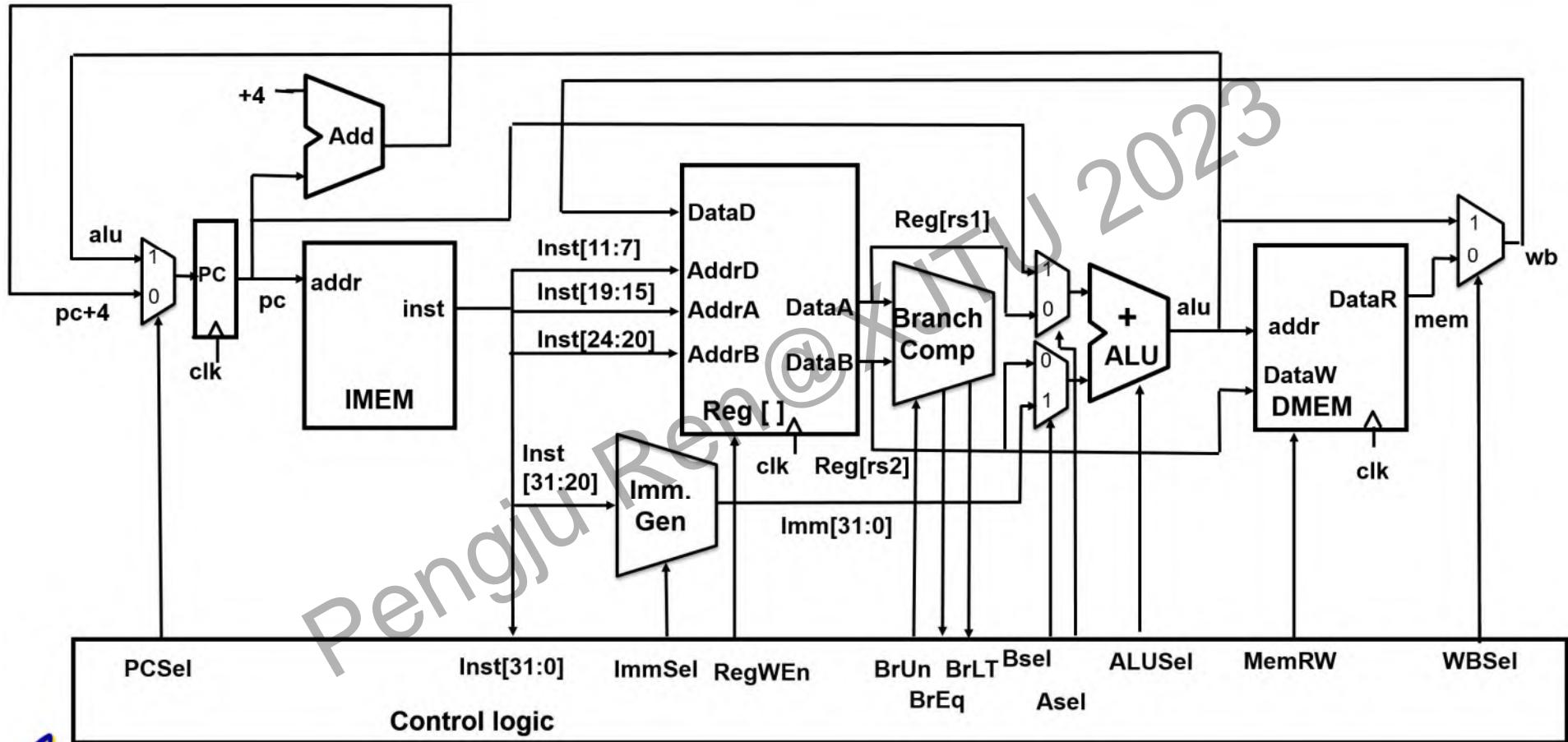
Upper bits sign-extended from **inst[31]** always Only bit 7 of instruction changes role in immediate between S and B

Let's Add JALR (I-Format)

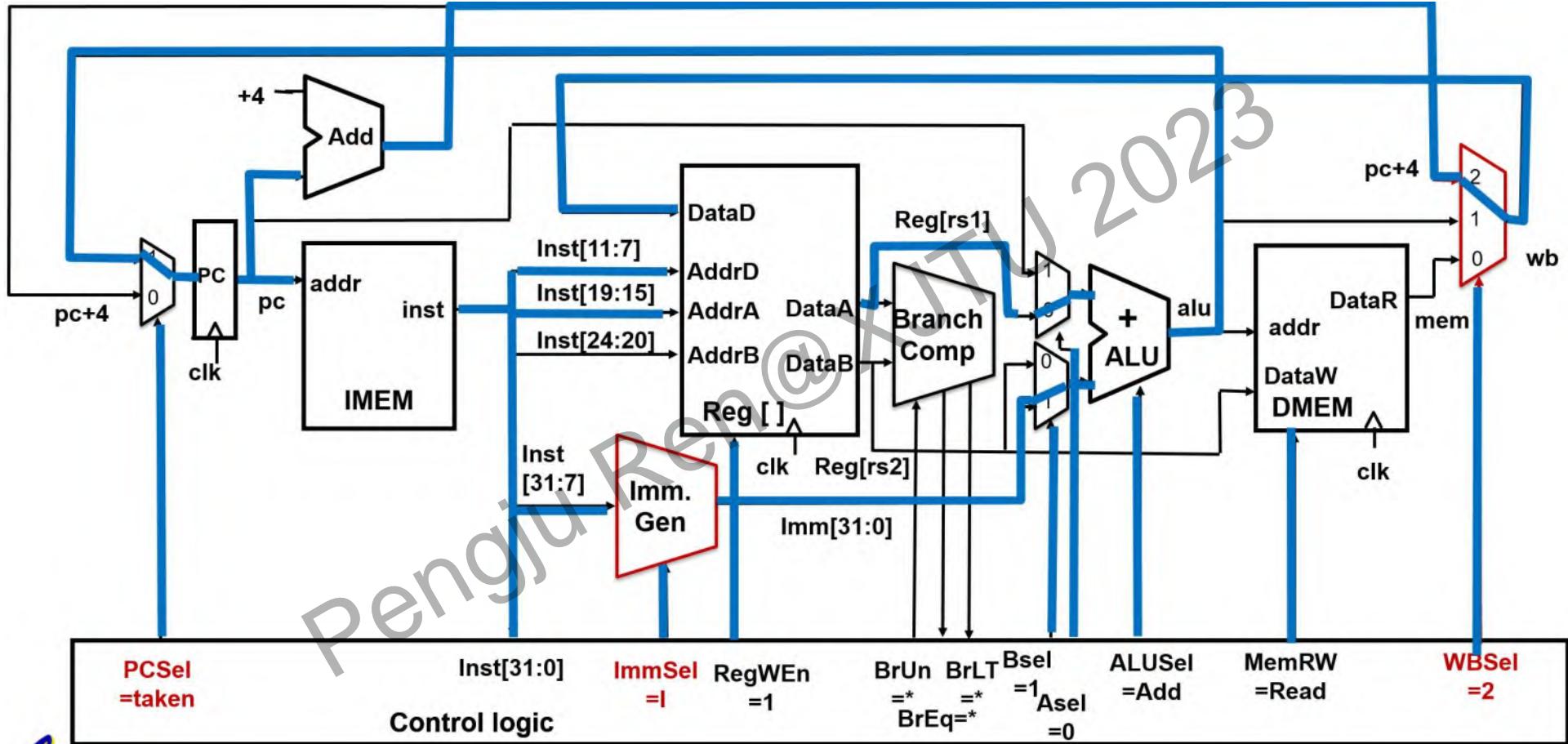


- JALR rd, rs, immediate
- Two changes to the state
 - Writes PC+4 to rd (return address)
 - Sets PC = rs + immediate (branch addr table)
 - **Uses same immediates as arithmetic and loads**
 - **no** multiplication by 2 bytes
 - LSB is ignored

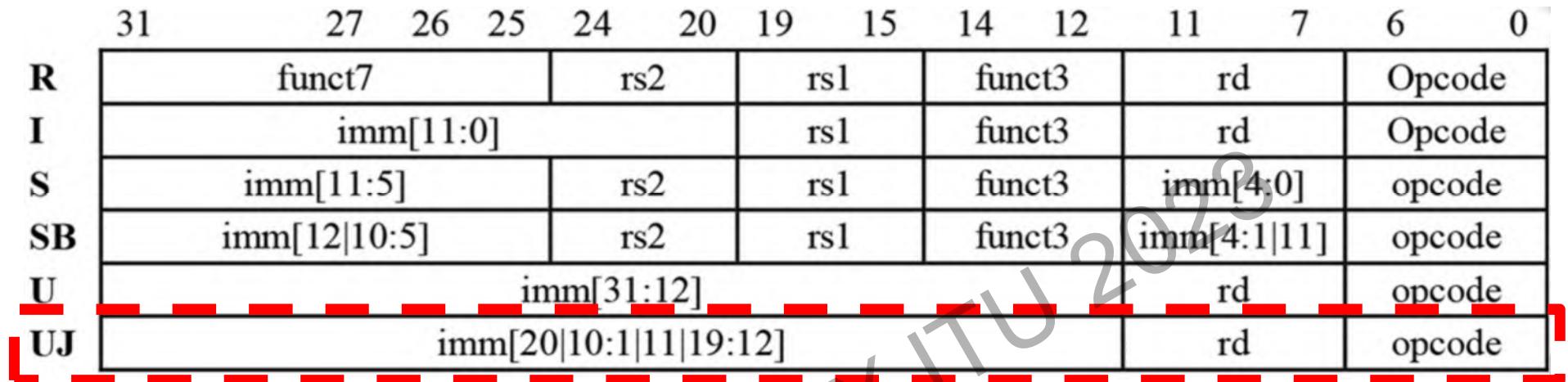
Datapath So Far, with Branches



Adding JALR ($R[rd] = PC+4$; $PC = R[rs1] + imm$)

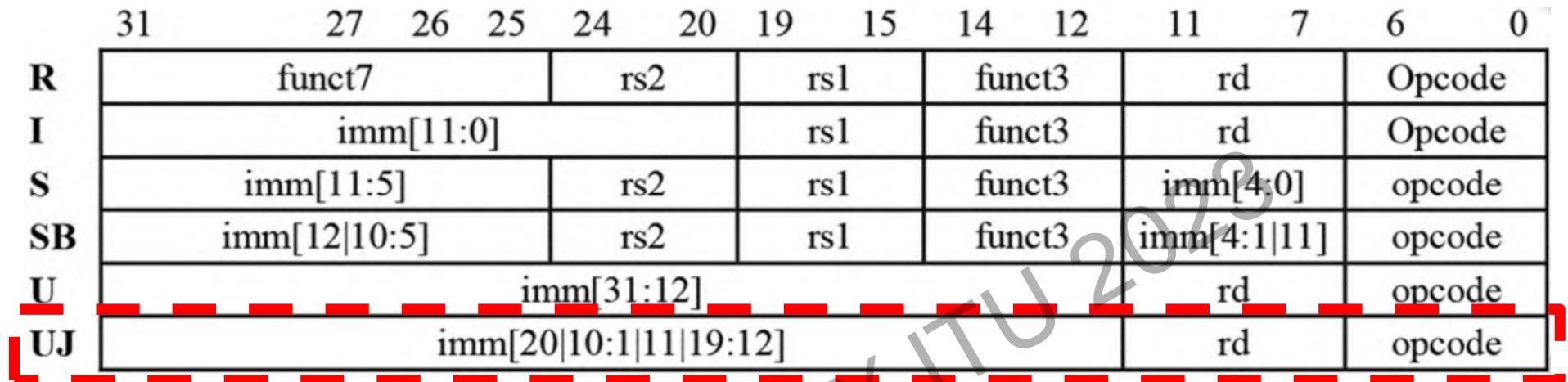


Recap: RISC-V Instruction Encoding(ISA)



- **R-Format:** instructions using 3 register inputs
 - add, xor, mul — arithmetic/logical ops
- **I-Format:** instructions with immediates, loads
 - addi, lw, jalr, slli
- **S-Format:** store instructions: sw, sb
- **SB-Format:** branch instructions: beq, bge
- **U-Format:** instructions with upper immediates
 - lui, auipc — upper immediate is 20-bits
- **UJ-Format:** jump instructions: jal

Recap: RISC-V Instruction Encoding(ISA)



- **R-Format:** instructions using 3 register inputs

— add, sub, mul — arithmetic/logical ops

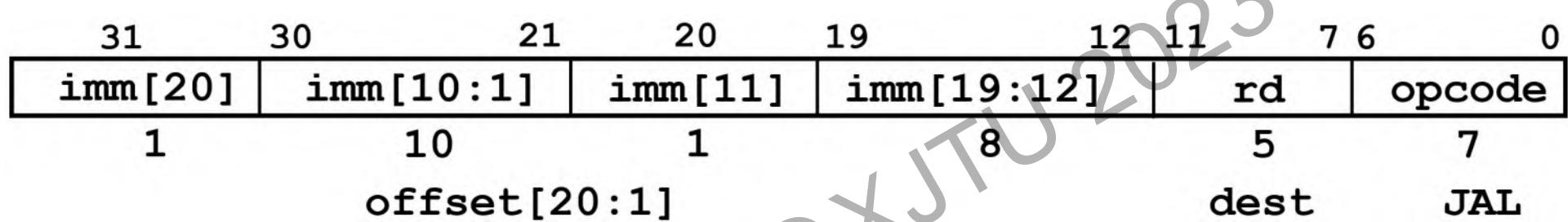
jal R[rd]=PC+4; PC=PC+{imm,1'b0} (jump and link)

- **U-Format:** instructions with upper immediates

— lui, auipc — upper immediate is 20-bits

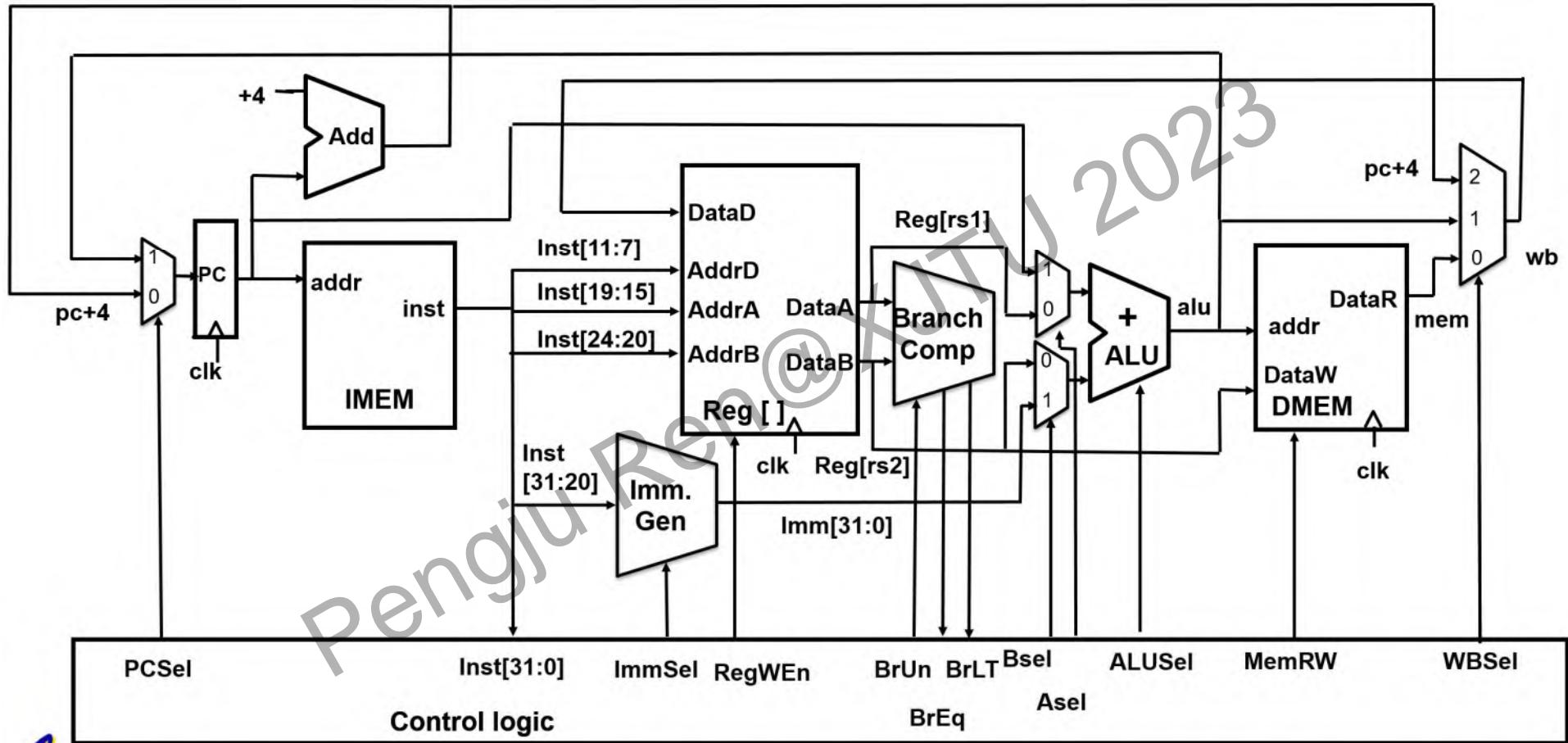
- **UJ-Format:** jump instructions: jal

Adding JAL

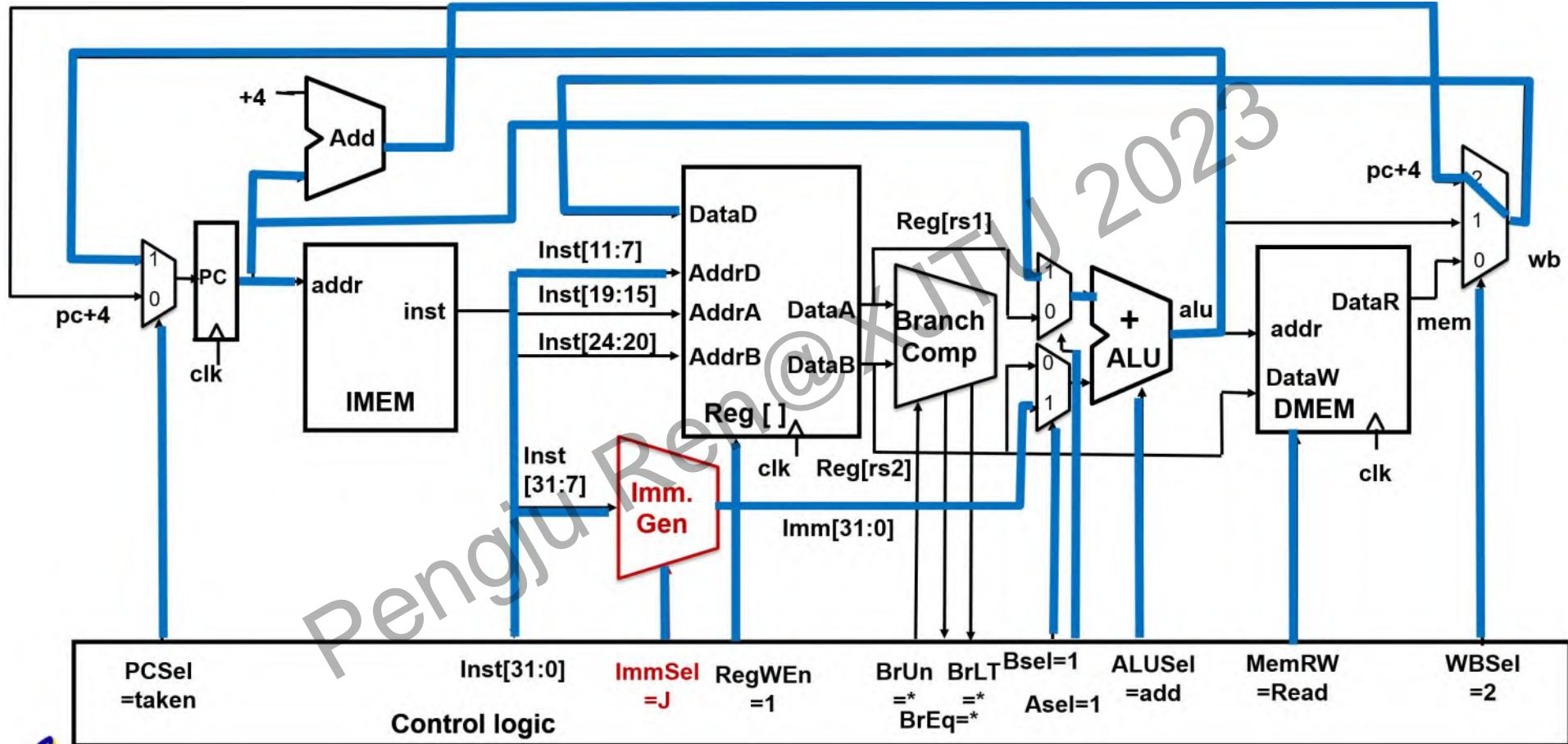


- JAL saves PC+4 in register rd (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
 - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

Datapath with JAL



Adding JAL ($R[rd] = PC+4$; $PC = PC + \{imm, 1b'0\}$)



Dealing with large immediates

How to deal with 32-bit immediates (for Calculation and PC-offset)?

- **I-type instructions only give us 12-bits**
- **UJ-type instructions only give us 20-bits**

Solution: Need a new instruction format for dealing with the rest of the 20 bits.

This instruction should deal with:

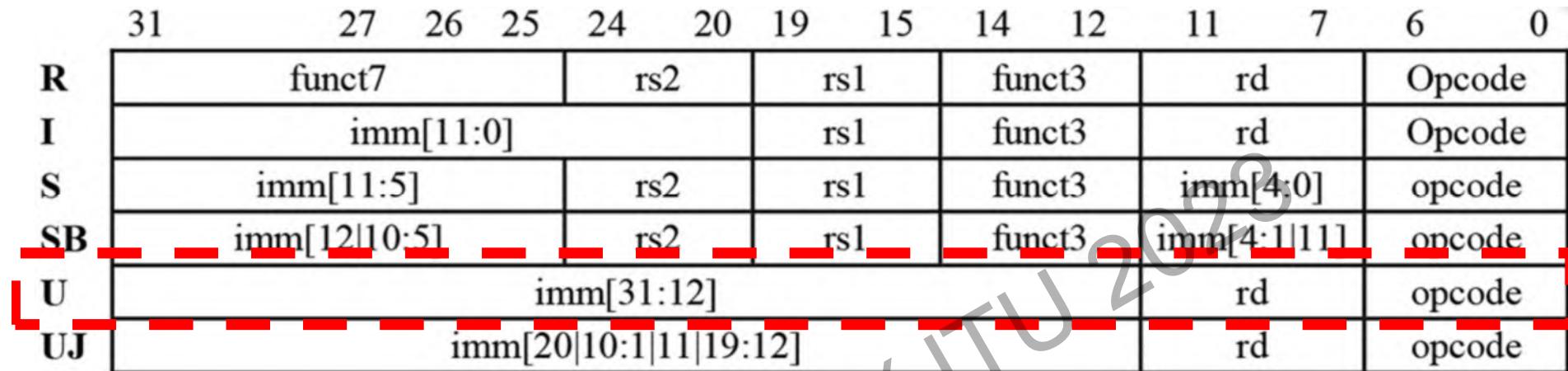
- a destination register to put the 20 bits into
- the immediate of 20 bits
- the instruction opcode

Recap: RISC-V Instruction Encoding(ISA)

	31	27	26	25	24	20	19	15	14	12	11	7	6	0
R		funct7		rs2		rs1		funct3		rd		Opcode		
I		imm[11:0]				rs1		funct3		rd		Opcode		
S		imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		
SB		imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode		
U			imm[31:12]							rd		opcode		
UJ			imm[20 10:1 11 19:12]							rd		opcode		

- **R-Format:** instructions using 3 register inputs
 - add, xor, mul — arithmetic/logical ops
- **I-Format:** instructions with immediates, loads
 - addi, lw, jalr, slli
- **S-Format:** store instructions: sw, sb
- **SB-Format:** branch instructions: beq, bge
- **U-Format:** instructions with upper immediates
 - lui, auipc — upper immediate is 20-bits
- **UJ-Format:** jump instructions: jal

Recap: RISC-V Instruction Encoding(ISA)



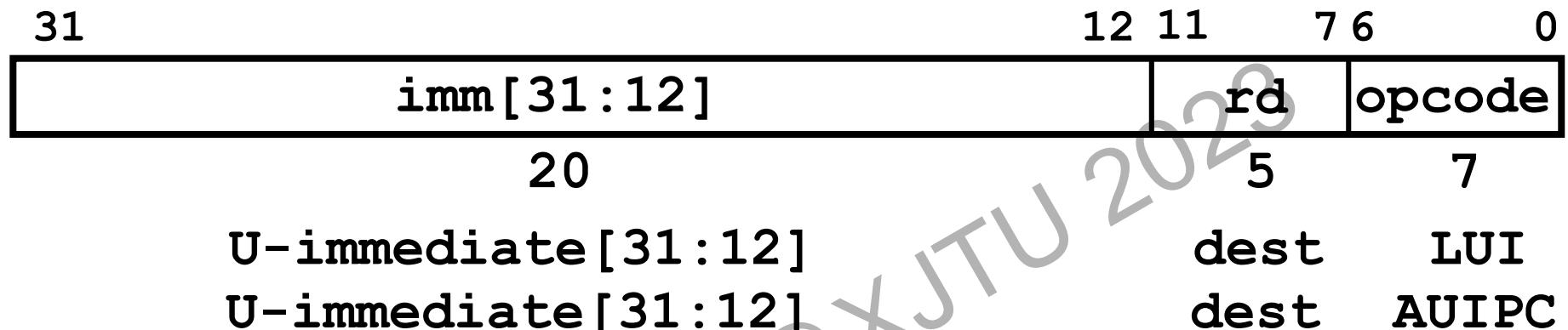
- **R-Format:** instructions using 3 register inputs

— add, sub, mul — arithmetic/logical ops

lui $R[rd]=\{32'bimm<31>,imm,12'b0\}$ (**Load Upper Immediate**)
auipc $R[rd]=PC+\{imm,12'b0\}$ (**Add Upper Immediate to PC**)

- **U-Format:** instructions with upper immediates
 - lui, auipc —upper immediate is 20-bits
- **UJ-Format:** jump instructions: jal

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

Usage of lui and AUIPC

Describe the function of the following instructions

`lui rY, BigImmediate`

`addi rX, rY, BigImmediate`

$$rY = \{\text{imm}[31:12], 12b'0\}$$

$$rX = rY + \{\text{imm}[11:0]\}$$

`AUIPC rs, labelFarAway`

`JALR rs, $ra, labelFarAway`

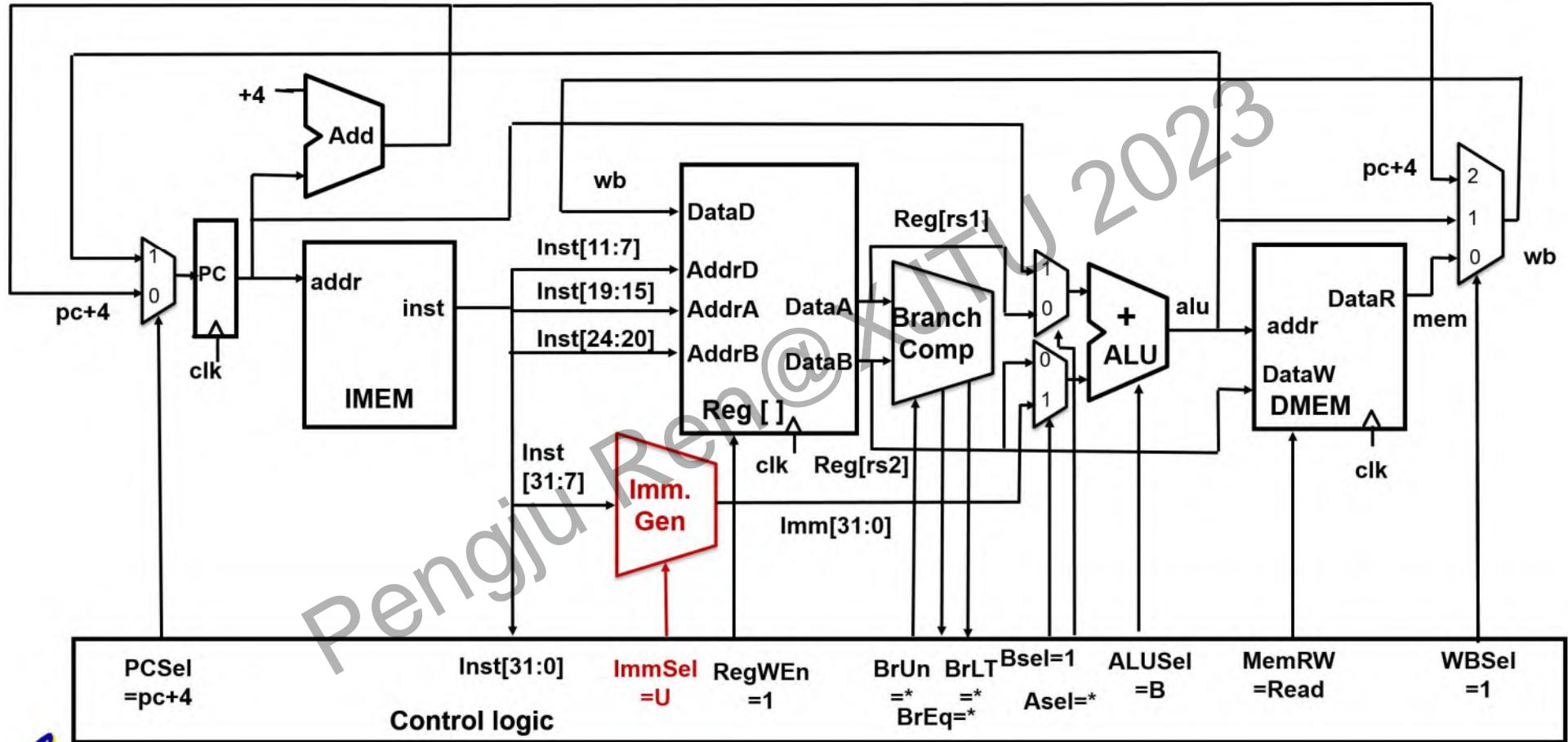
$$rs = PC + \{\text{imm}[31:12], 12b'0\}$$

$$PC = rs + \{\text{imm}[11:0]\}$$

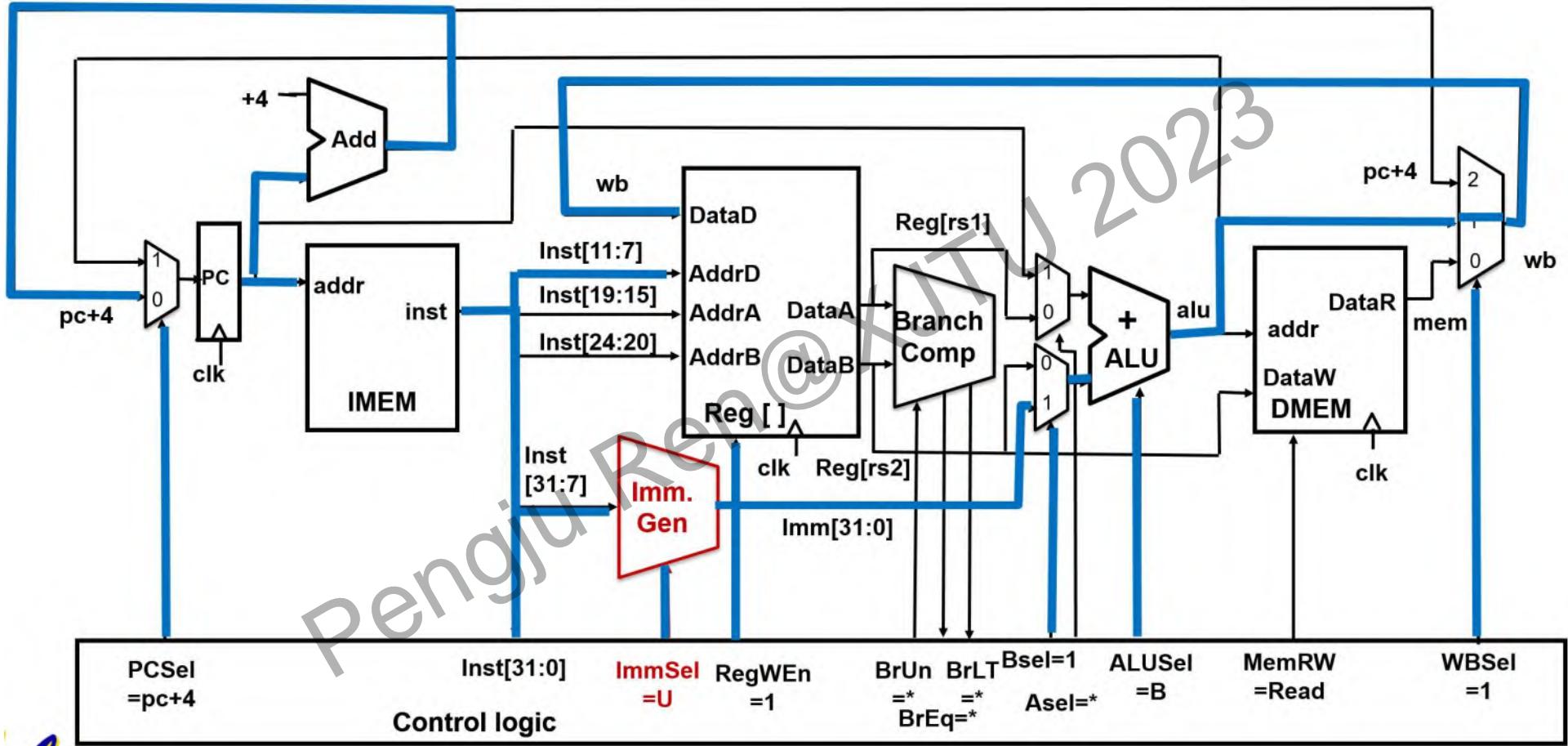
$$ra = PC + 4$$

`lui` writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits

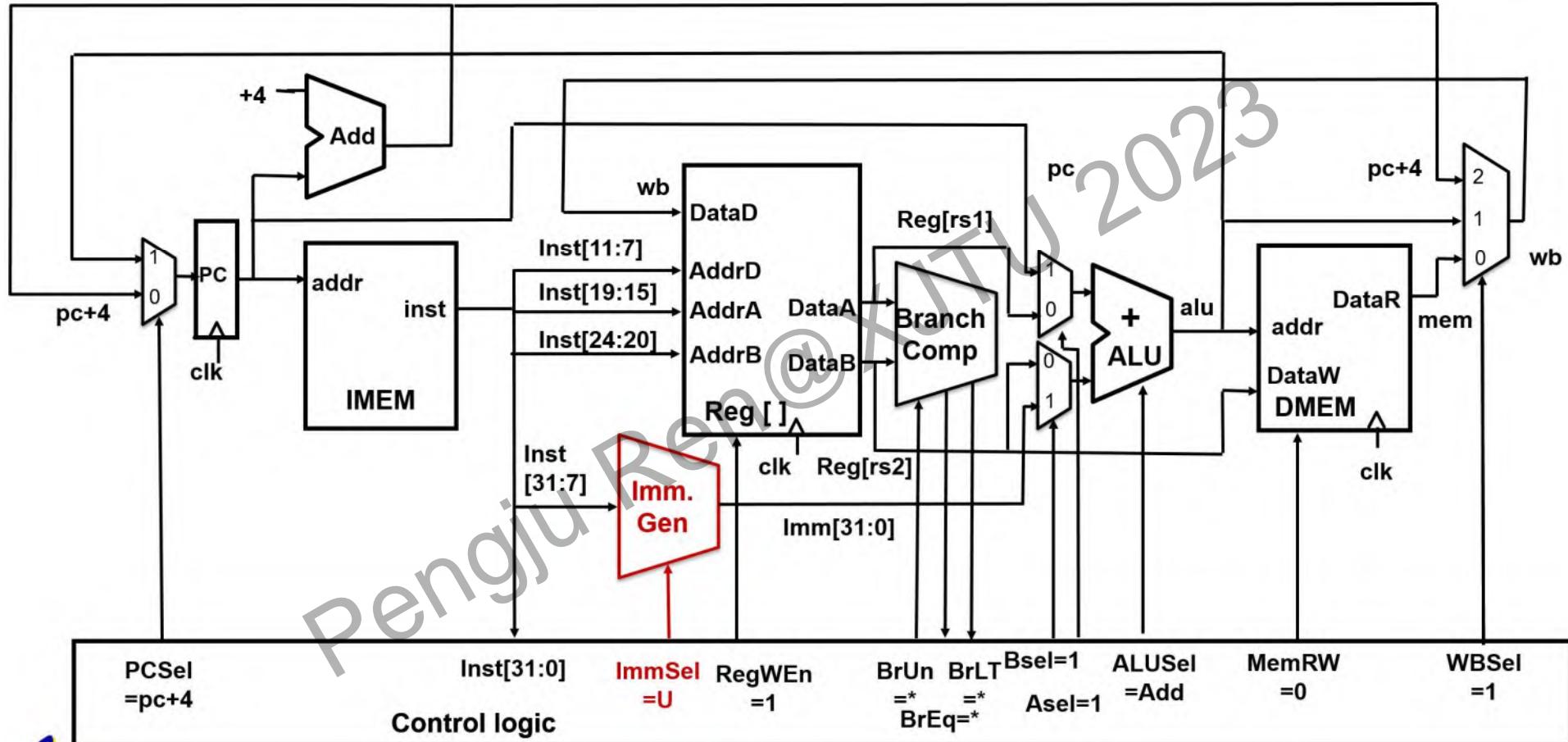
Implementing LUI ($R[rd] = \{imm, 12'b0\}$)



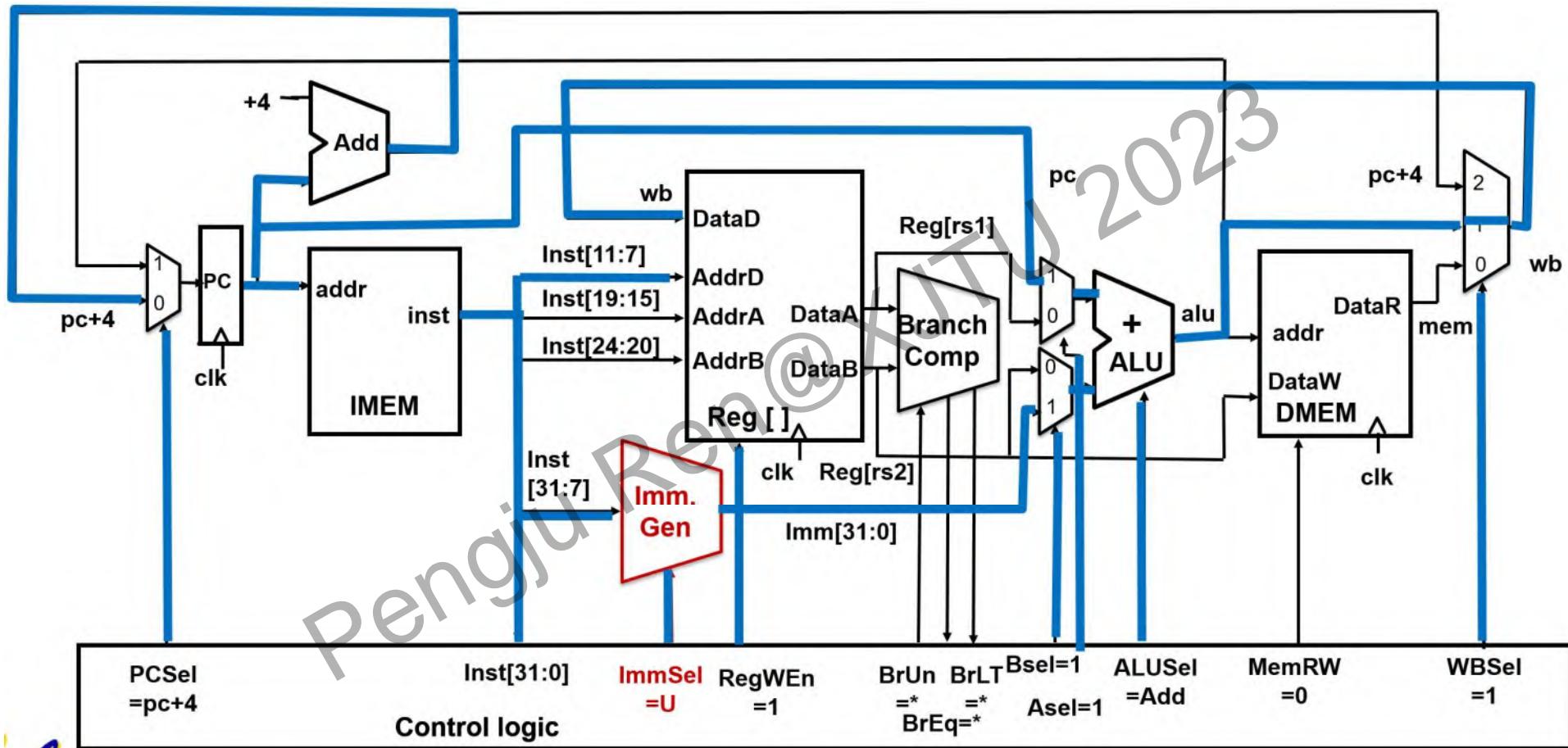
Implementing LUI ($R[rd] = \{imm, 12'b0\}$)



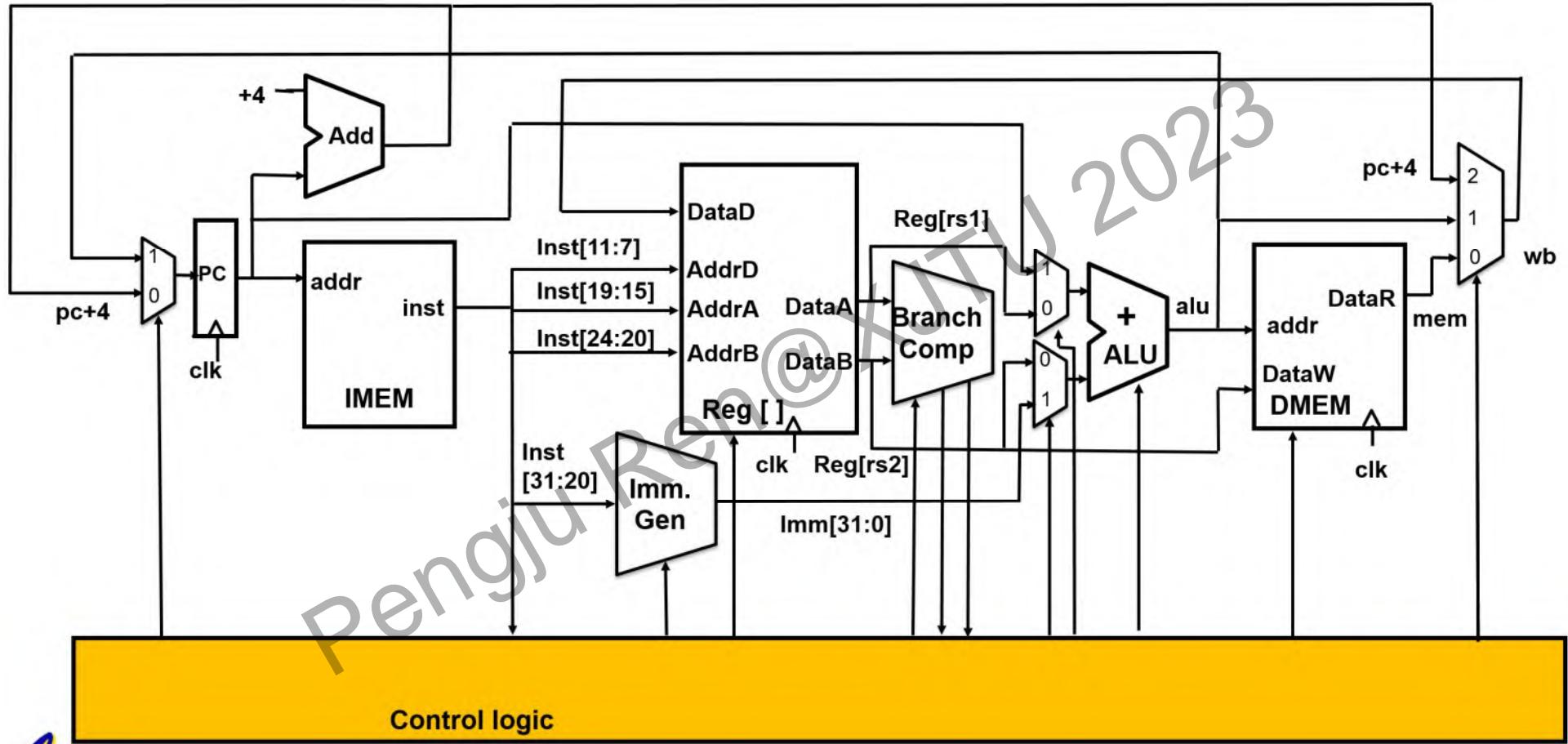
Implementing AUIPC ($R[rd] = PC + \{imm, 12'b0\}$)



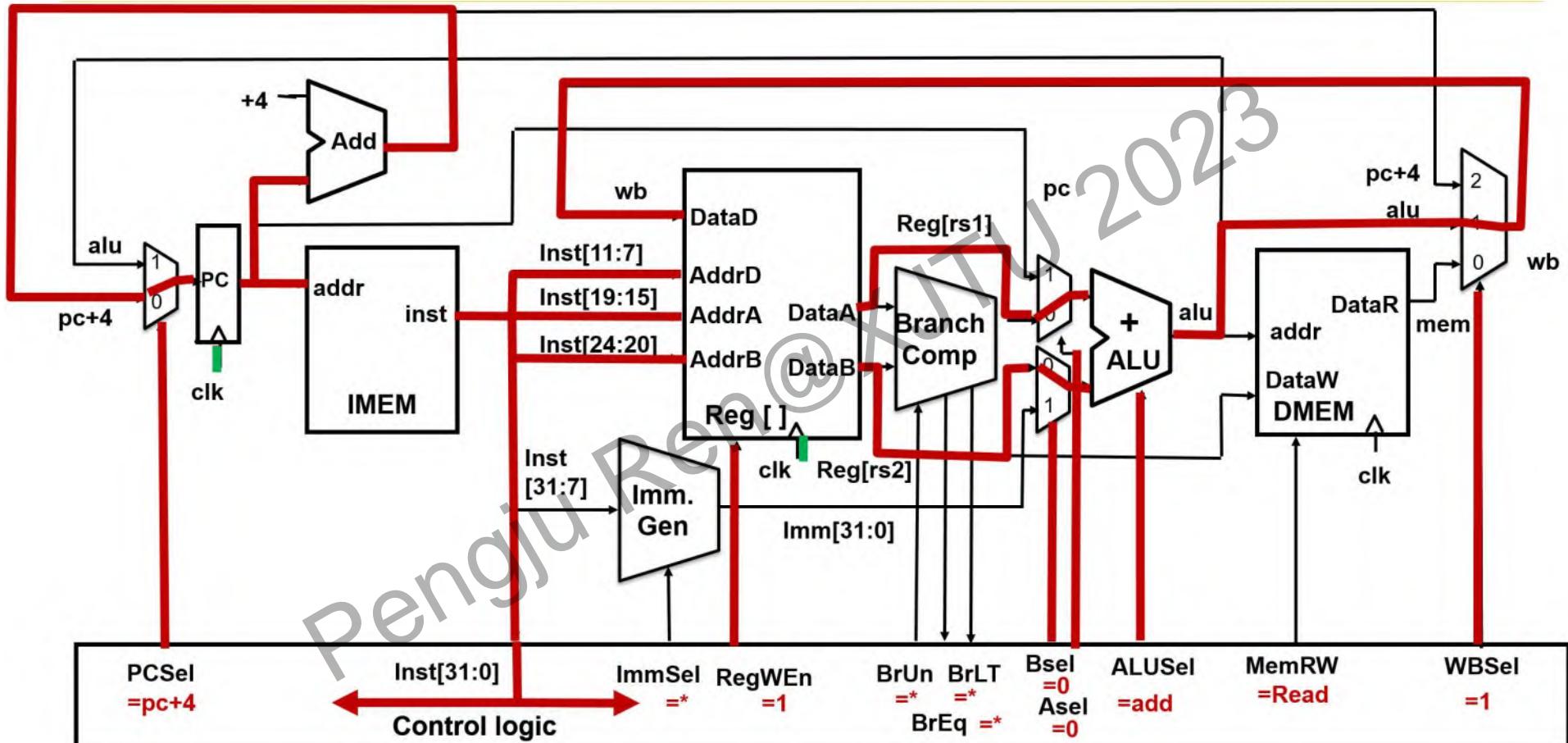
Implementing AUIPC ($R[rd] = PC + \{imm, 12'b0\}$)



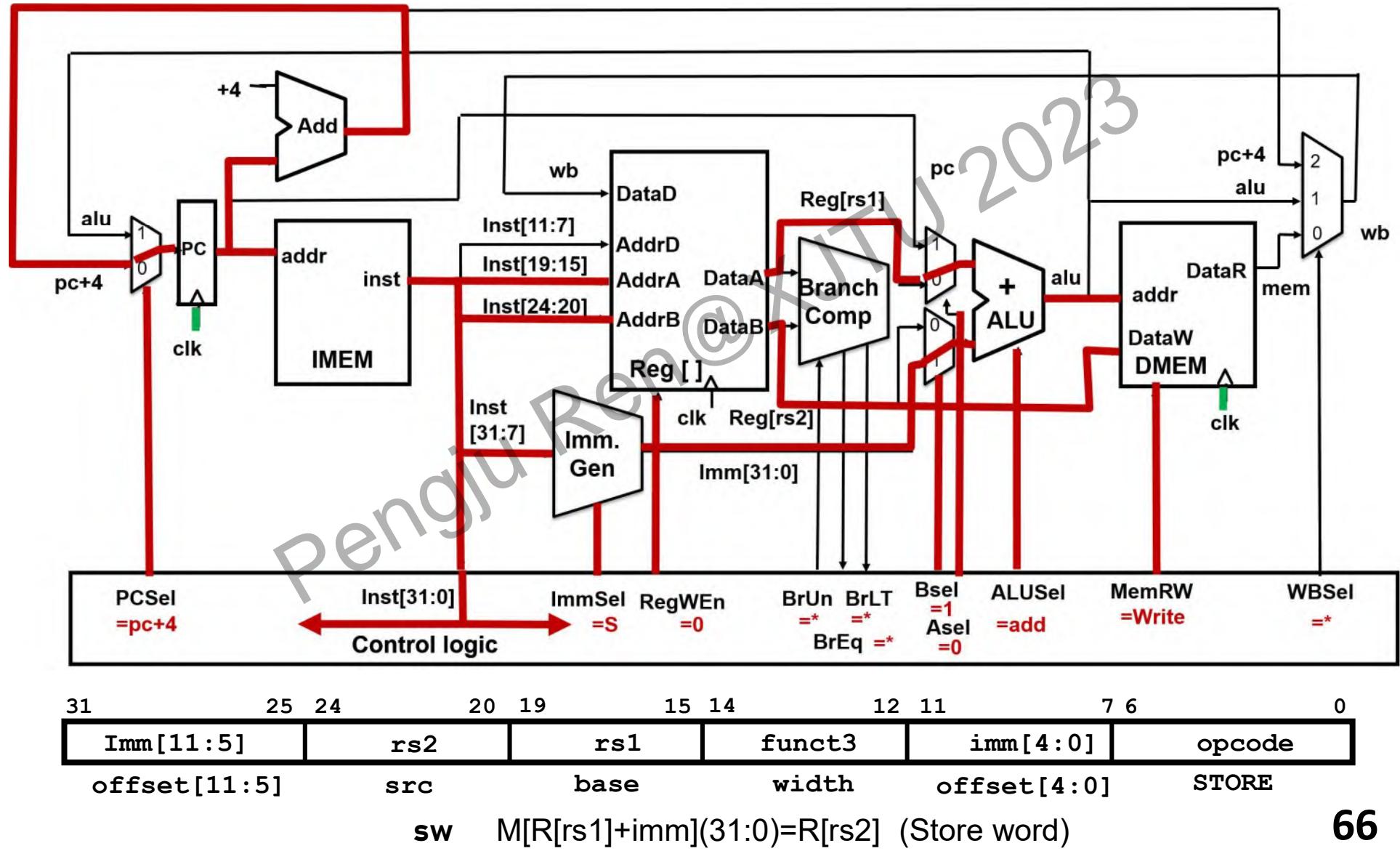
Single-Cycle RV32I Datapath and Control



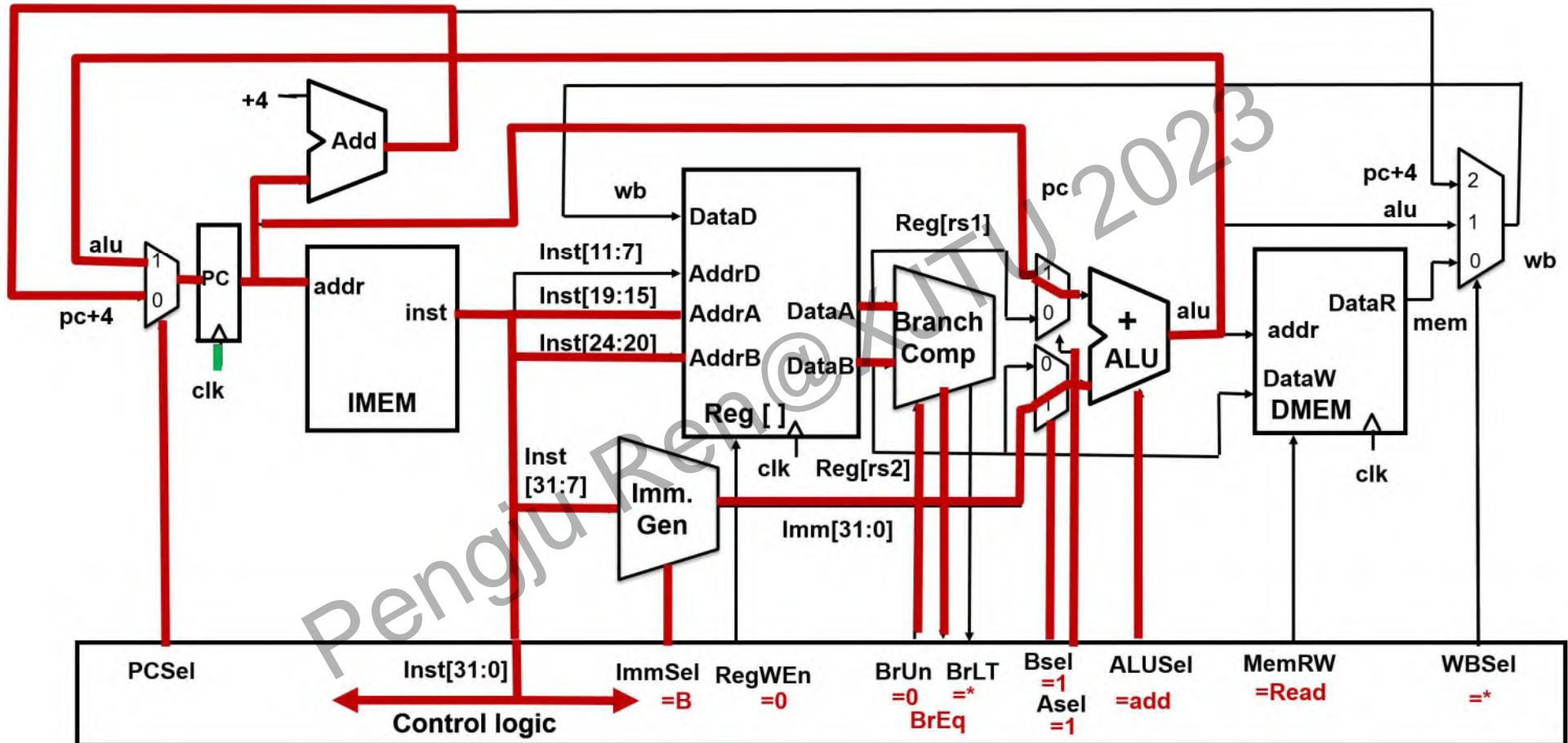
Example: add



Example: sw



Example: beq



beq if($R[rs1] == R[rs2]$) $PC = PC + \{imm.1'b0\}$ (Branch Equal)

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
<i>(R-R Op)</i>	*	*	+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

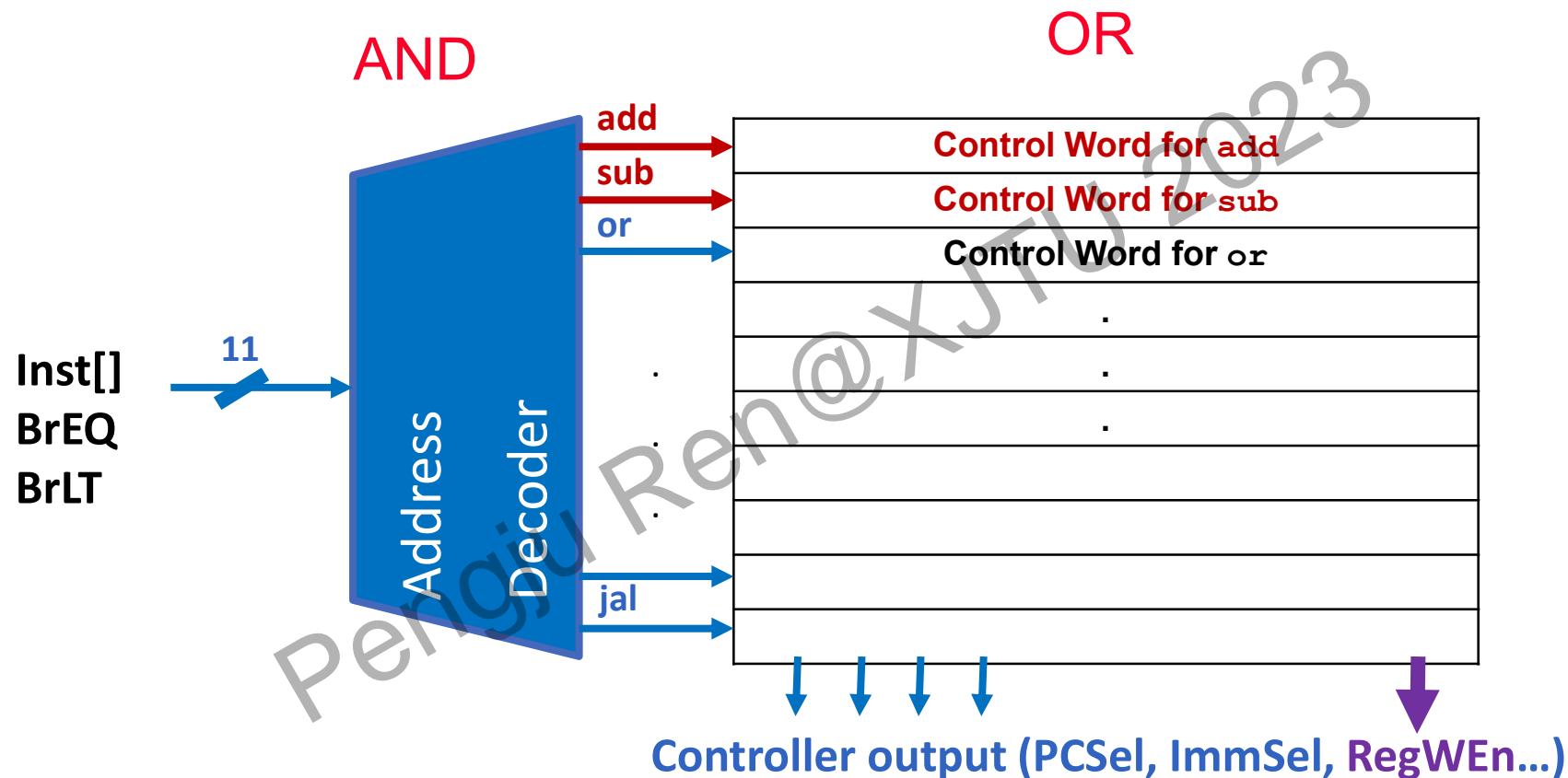
■ ROM

- “Read-Only Memory”
- Regular structure
- Can be easily reprogrammed
 - fix errors
 - add instructions
- Popular when designing control logic manually

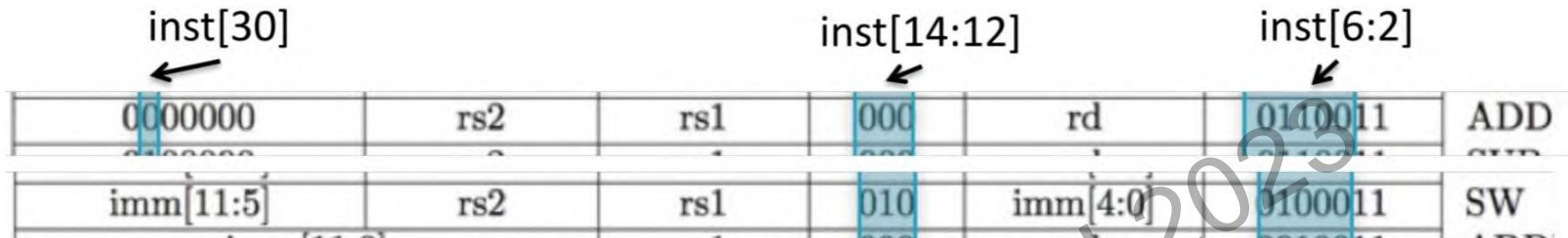
■ Combinatorial Logic

- Today, chip designers use logic synthesis tools to convert truth tables to networks of gates

ROM Controller Implementation



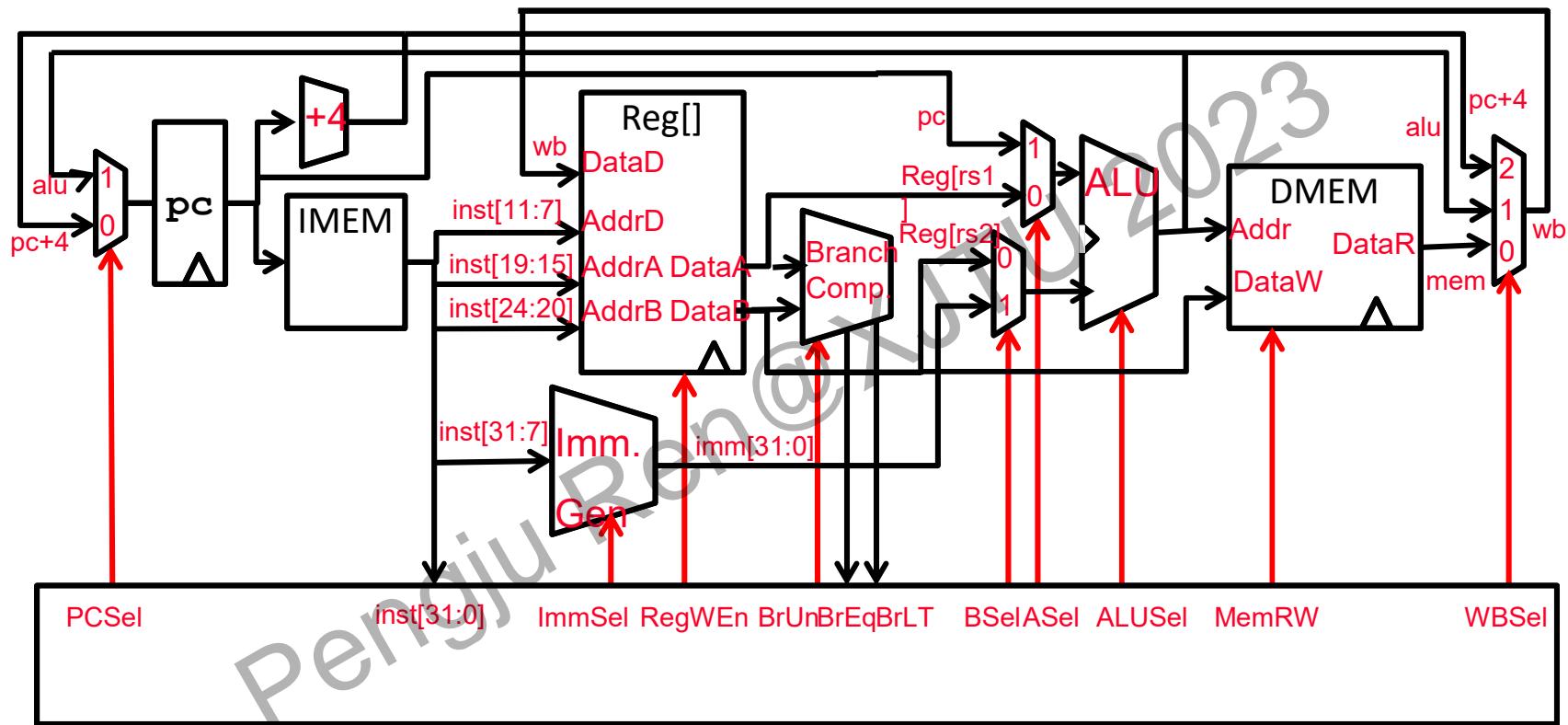
Combinatorial Logic Controller Implementation



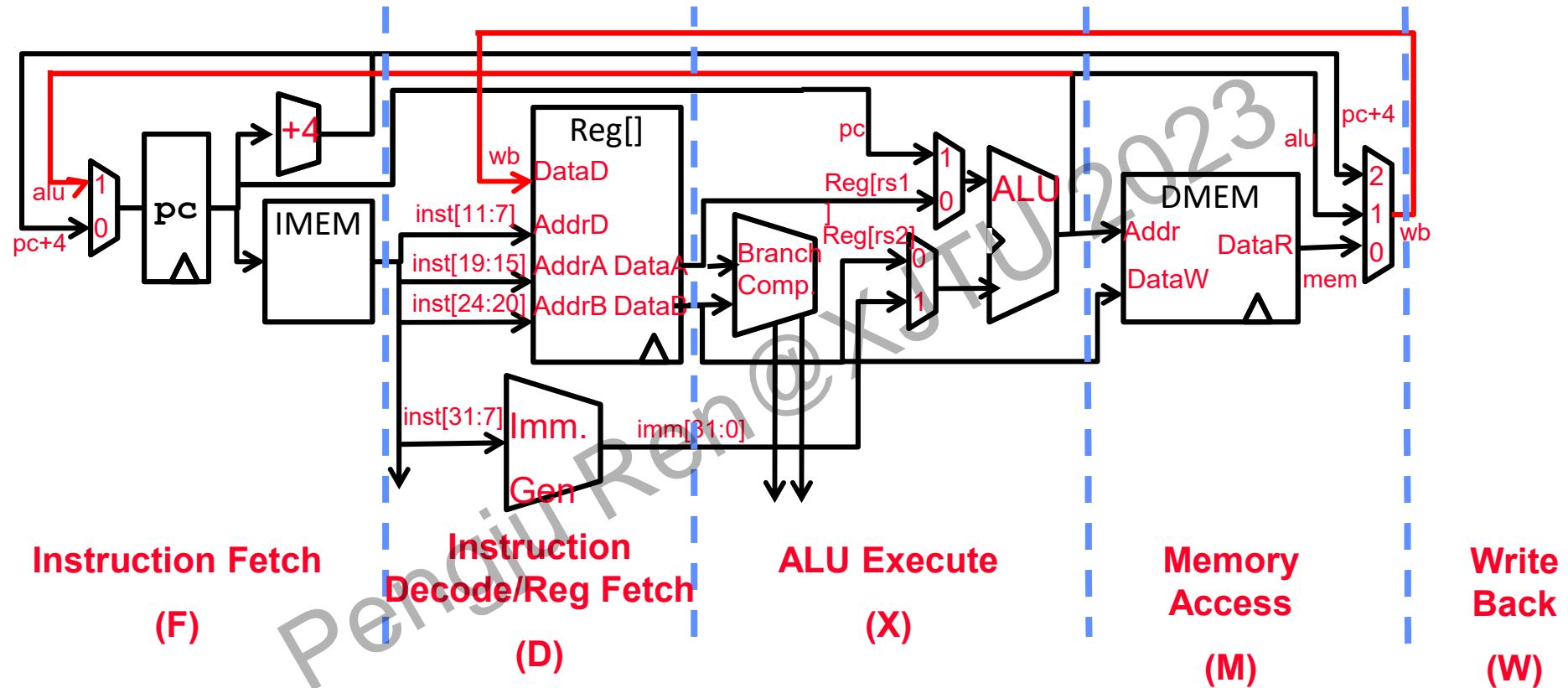
- add= $\overline{\text{inst}[2]} \cdot \overline{\text{inst}[3]} \cdot \overline{\text{inst}[4]} \cdot \overline{\text{inst}[5]} \cdot \overline{\text{inst}[6]} \cdot \overline{\text{inst}[12]} \cdot \overline{\text{inst}[13]} \cdot \overline{\text{inst}[14]} \cdot \overline{\text{inst}[30]}$
- sw= $\overline{\text{inst}[2]} \cdot \overline{\text{inst}[3]} \cdot \overline{\text{inst}[4]} \cdot \overline{\text{inst}[5]} \cdot \overline{\text{inst}[6]} \cdot \overline{\text{inst}[12]} \cdot \overline{\text{inst}[13]} \cdot \overline{\text{inst}[14]}$

AND Controller Logic										
	2	3	4	5	6	12	13	14	30	unused
add	xor(i[2], 1) • xor(i[3], 1) • xor(i[4], 0) • xor(i[5], 0) • xor(i[6], 1) • xor(i[12], 1) • xor(i[13], 1) • xor(i[14], 1) • (xor(i[30], 1) + 0)									
sw	xor(i[2], 1) • xor(i[3], 1) • xor(i[4], 1) • xor(i[5], 0) • xor(i[6], 1) • xor(i[12], 1) • xor(i[13], 1) • xor(i[14], 1) • (xor(i[30], 1) + 1)									
...										

Single-Cycle RISC-V RV32I Datapath

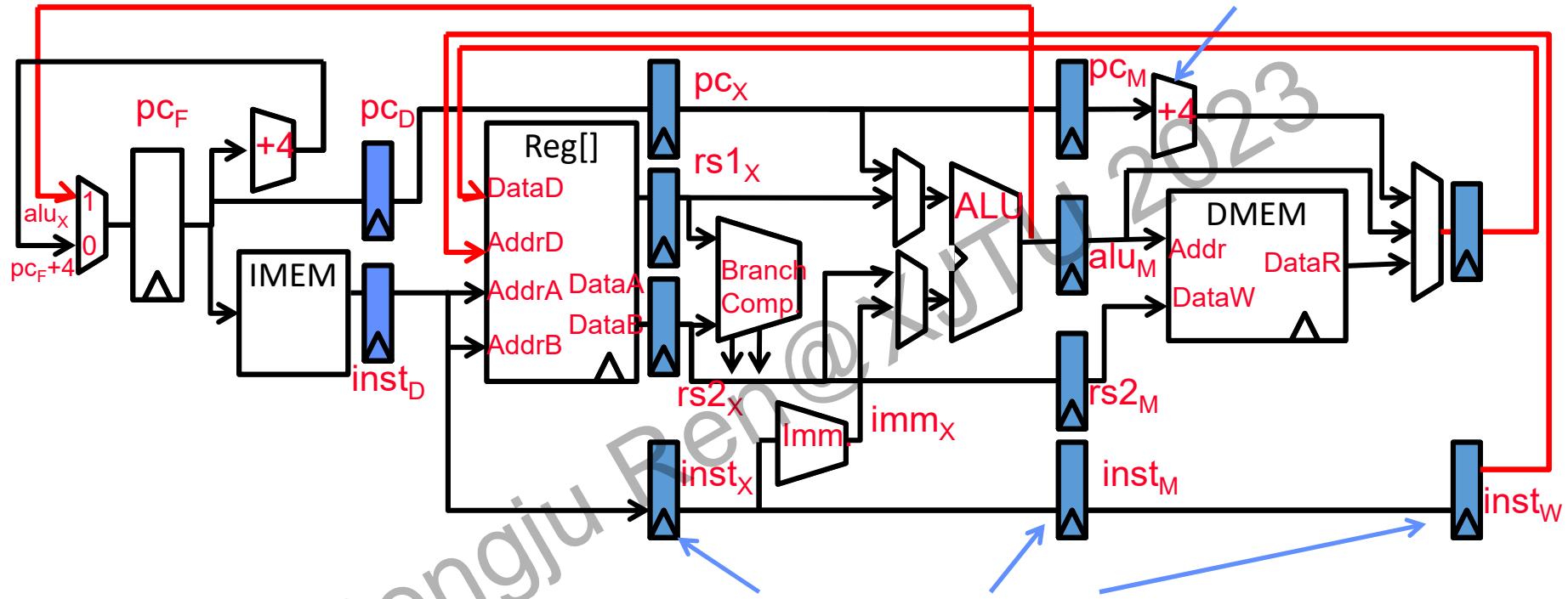


Pipelining RISC-V RV32I Datapath



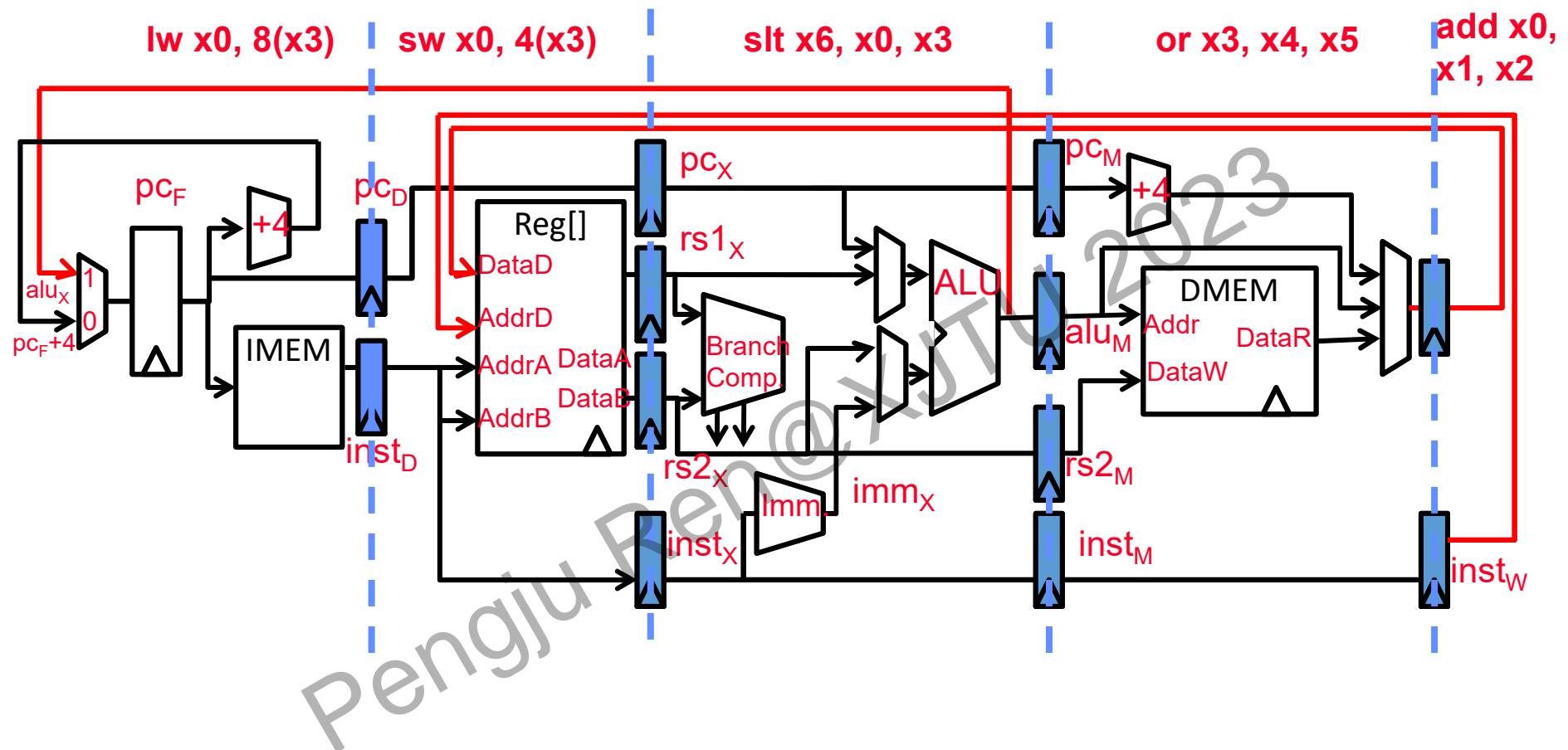
Pipelined RISC-V RV32I Datapath

① Recalculate PC+4 in M stage to make sure sending the right number to rd

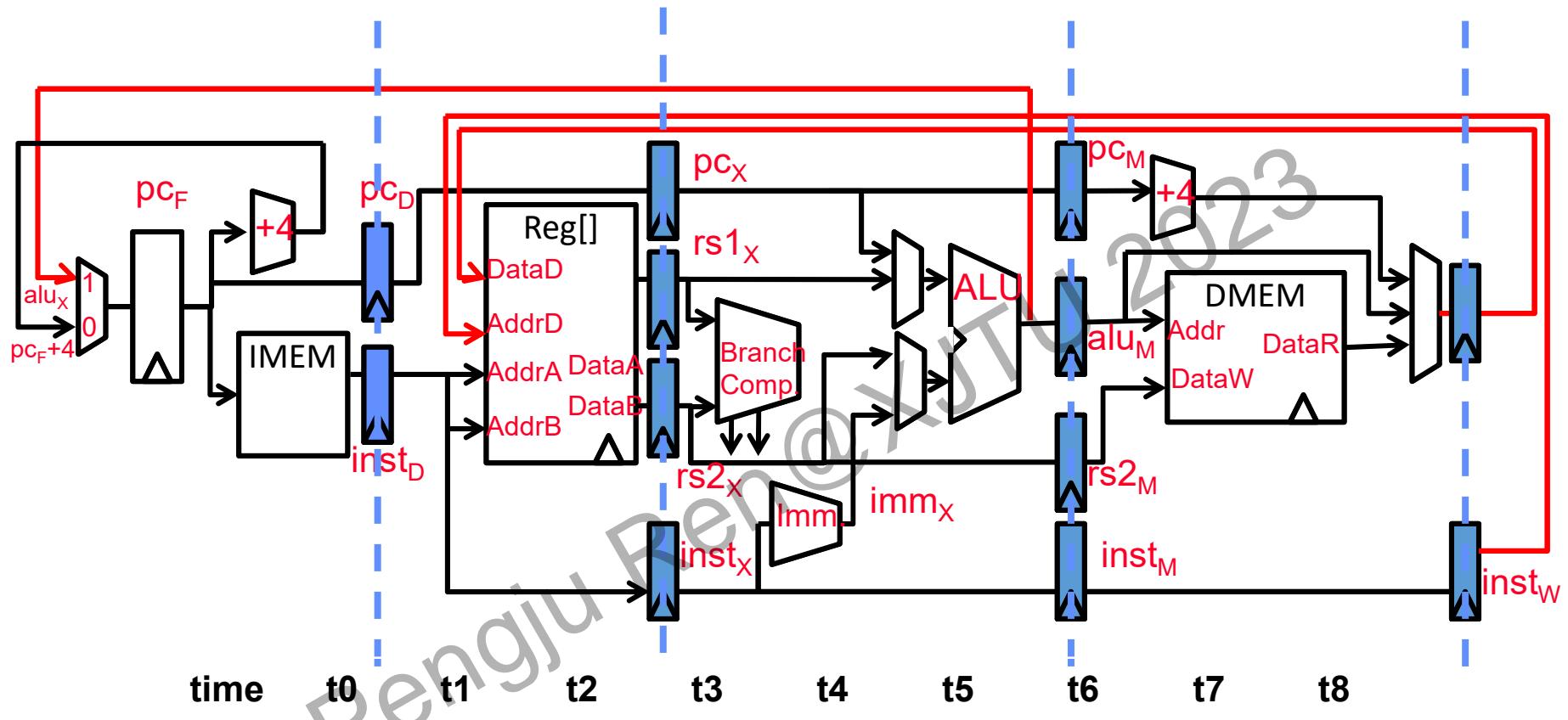


② Must pipeline instruction along with data, so control operates correctly in each stage

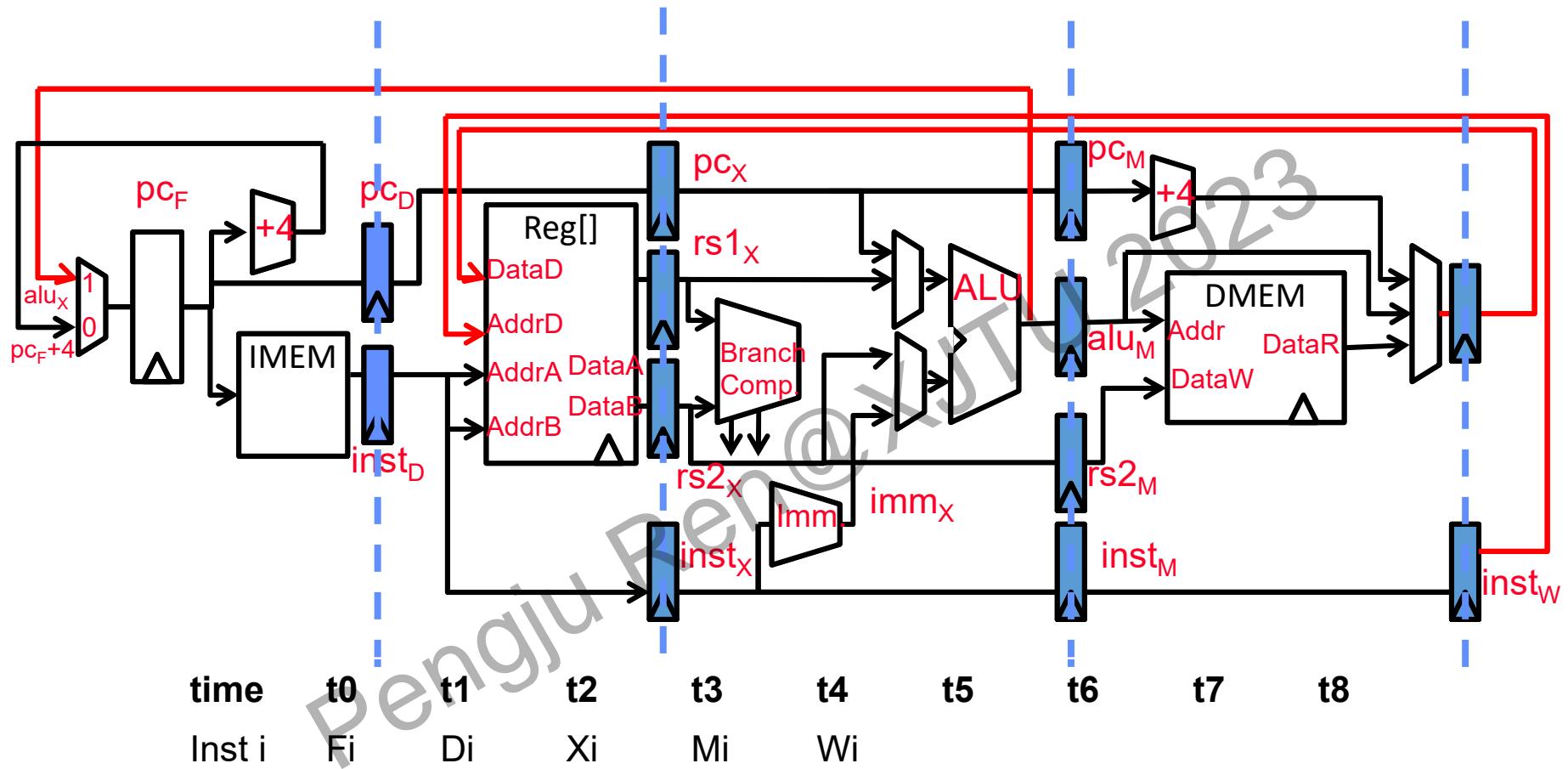
Each stage operates on different instruction



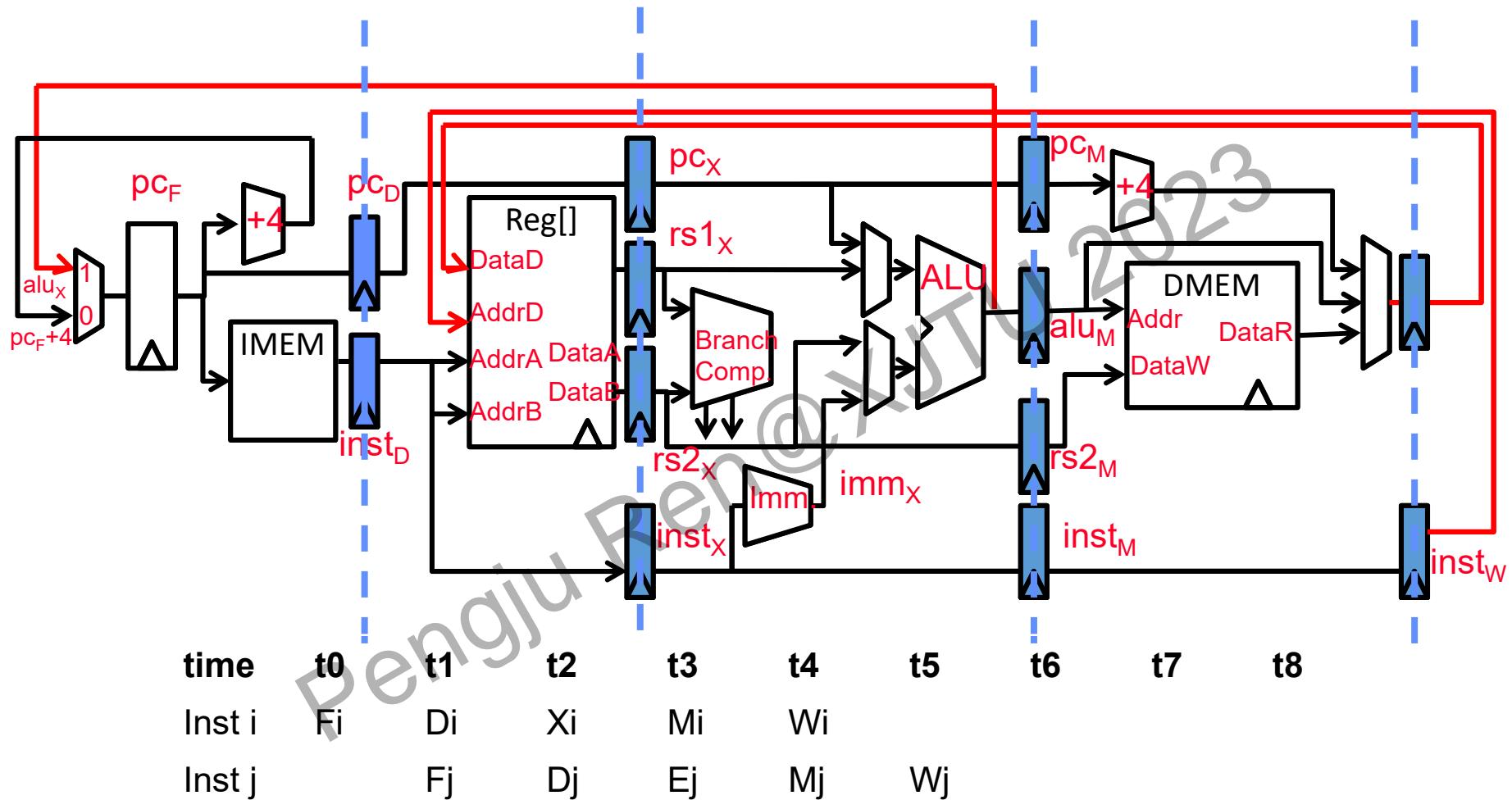
Pipeline-Instruction flow diagram



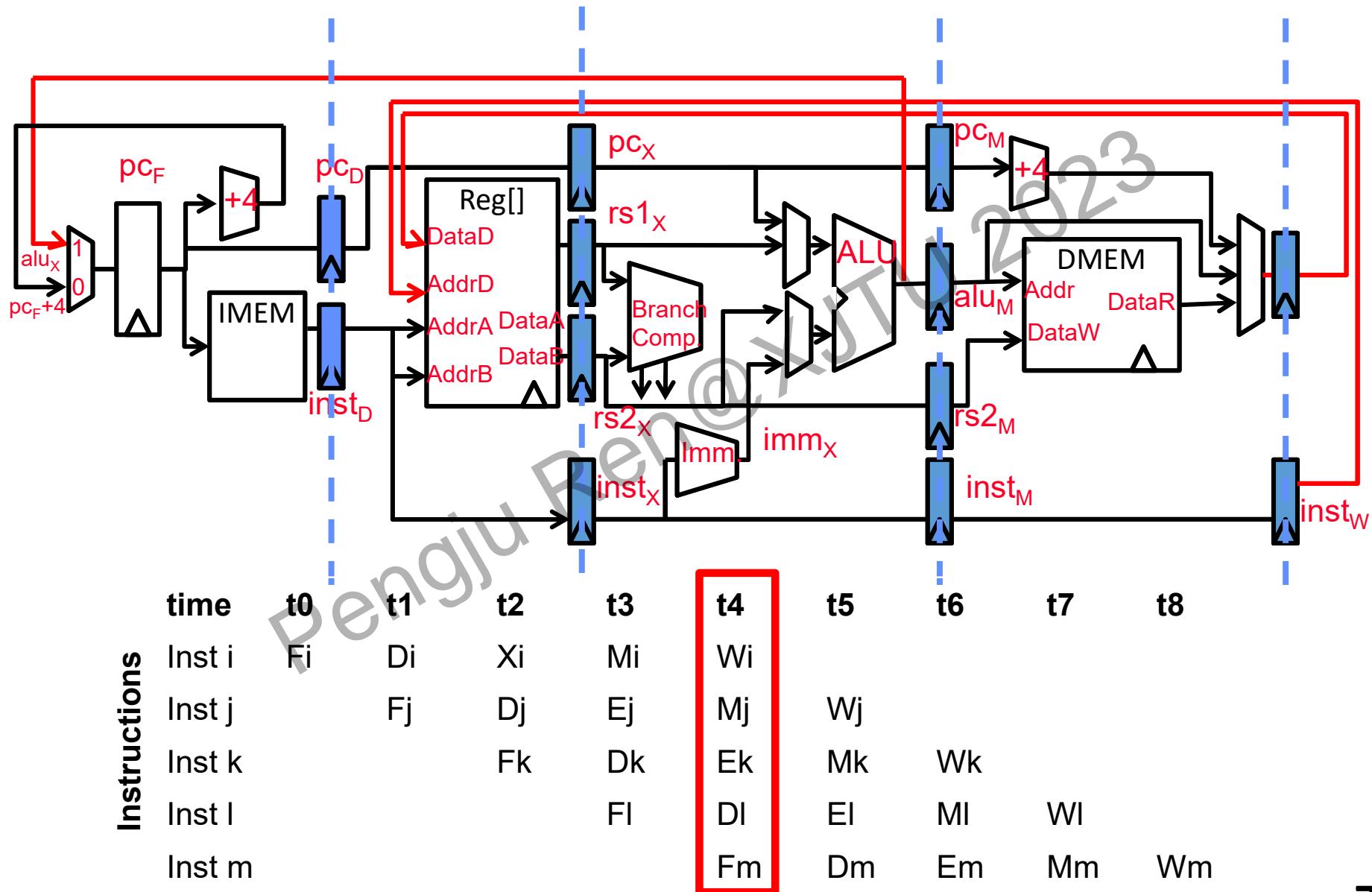
Pipeline-Instruction flow diagram



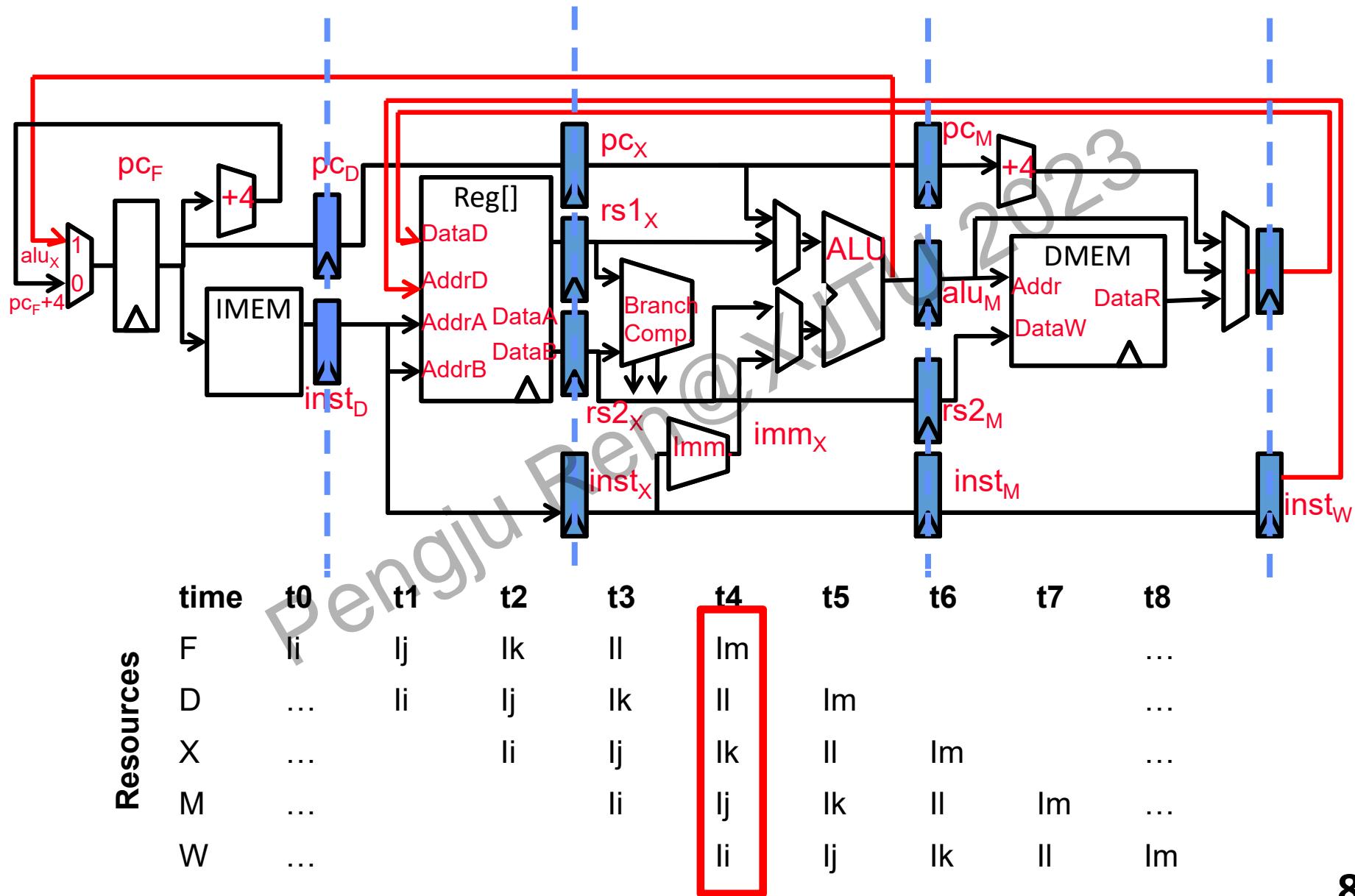
Pipeline-Instruction flow diagram



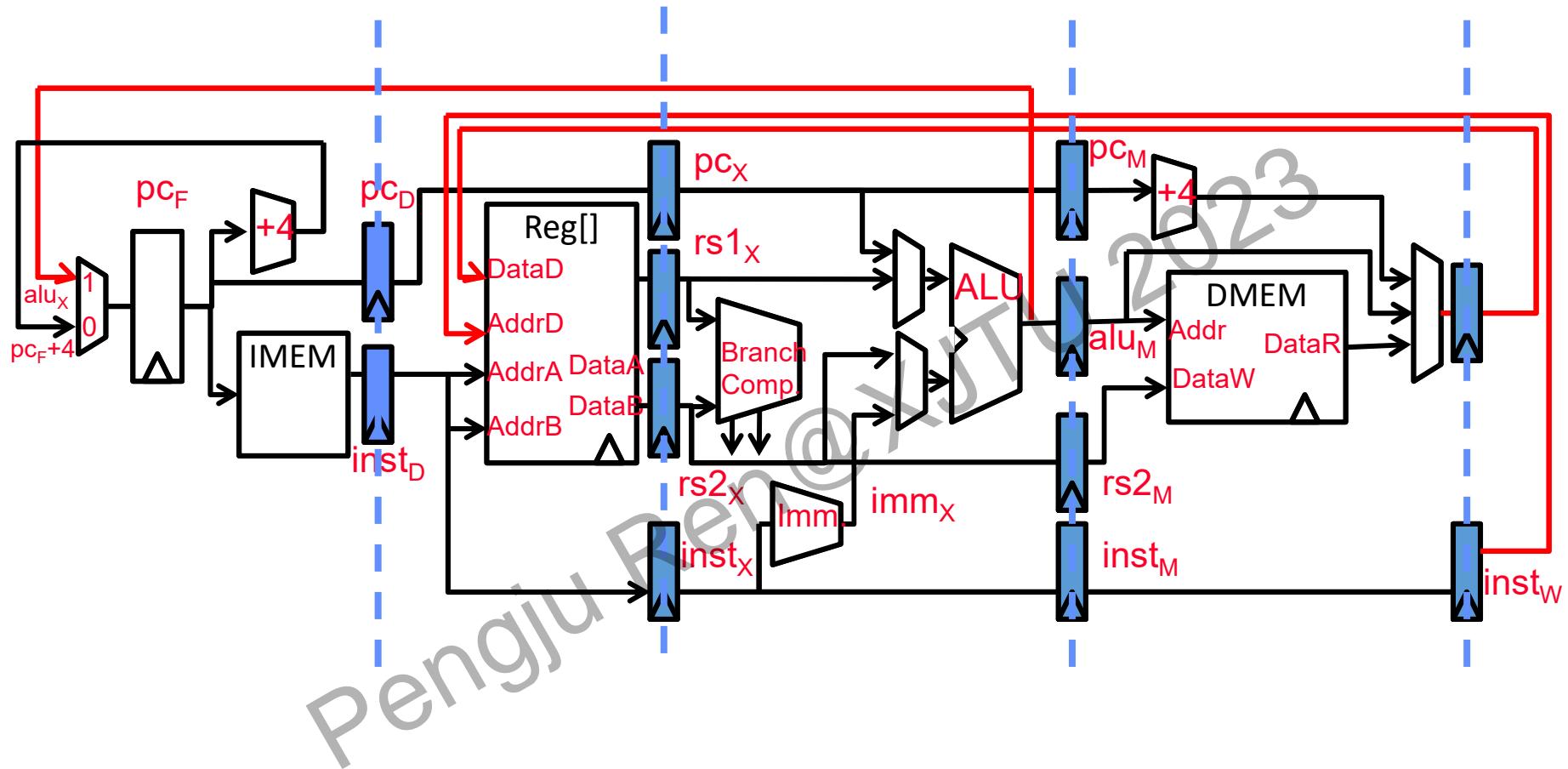
Pipeline-Instruction flow diagram



Pipeline-Instruction flow diagram



The Pipelined version of the datapath



*Next Lecture : Pipeline and Hazard
(Instruction level parallel)*

Acknowledgements

- Some slides contain material developed and copyright by:
 - Arvind (MIT)
 - Krste Asanovic (MIT/UCB)
 - Joel Emer (Intel/MIT)
 - James Hoe (CMU)
 - David Patterson (UCB)
 - David Wentzlaff (Princeton University)
- MIT material derived from course 6.823
- UCB material derived from course CS252 and CS 61C