inst.eecs.berkeley.edu/~eecs151

EECS151 : Introduction to Digital Design and ICs

Lecture 9 – RISC-V ISA, Pipelining

Bora Nikolić

August 20, 2021, Tom's hardware
Tesla Packs 50 Billion Transistors Onto D1 Dojo Chip Designed to Conquer Artificial Intelligence Training

D1 delivers 362 TeraFLOPs of power.
Called the D1, the chip resembles a part of the Dojo supercomputer used to train AI models inside Tesla HQ, which are later deployed in various applications. The D1 chip is a product of TSMC's manufacturing efforts, forged in a 7nm semiconductor node. Packing over 50 billion transistors, the chip boasts a huge die size of 645mm^2.

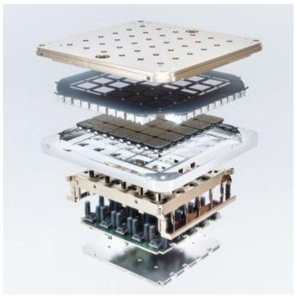


Image credit: Dennis Hong / Twitter

1

Review

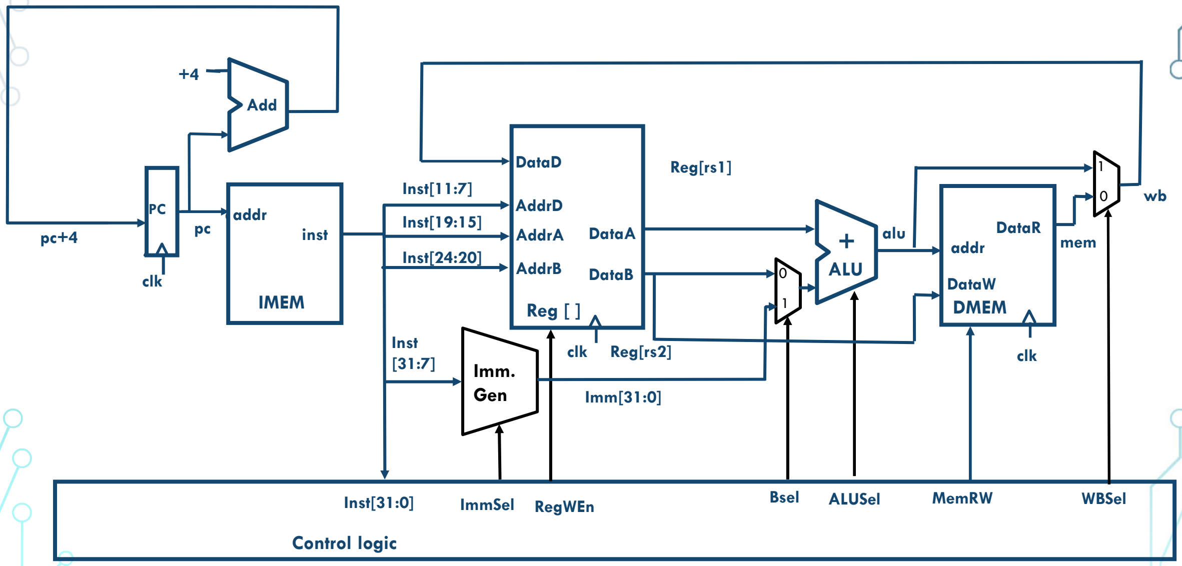
- RISC-V ISA
  - Open, with increasing adoption
- RISC-V processor
  - A large state machine
  - Datapath + control
  - Reviewed R-, I-, S-format instructions and corresponding datapath elements

2

RISC-V
B-Format Instructions

3

Datapath So Far (R-, I-, S Instruction Types)



4

B-Format - RISC-V Conditional Branches

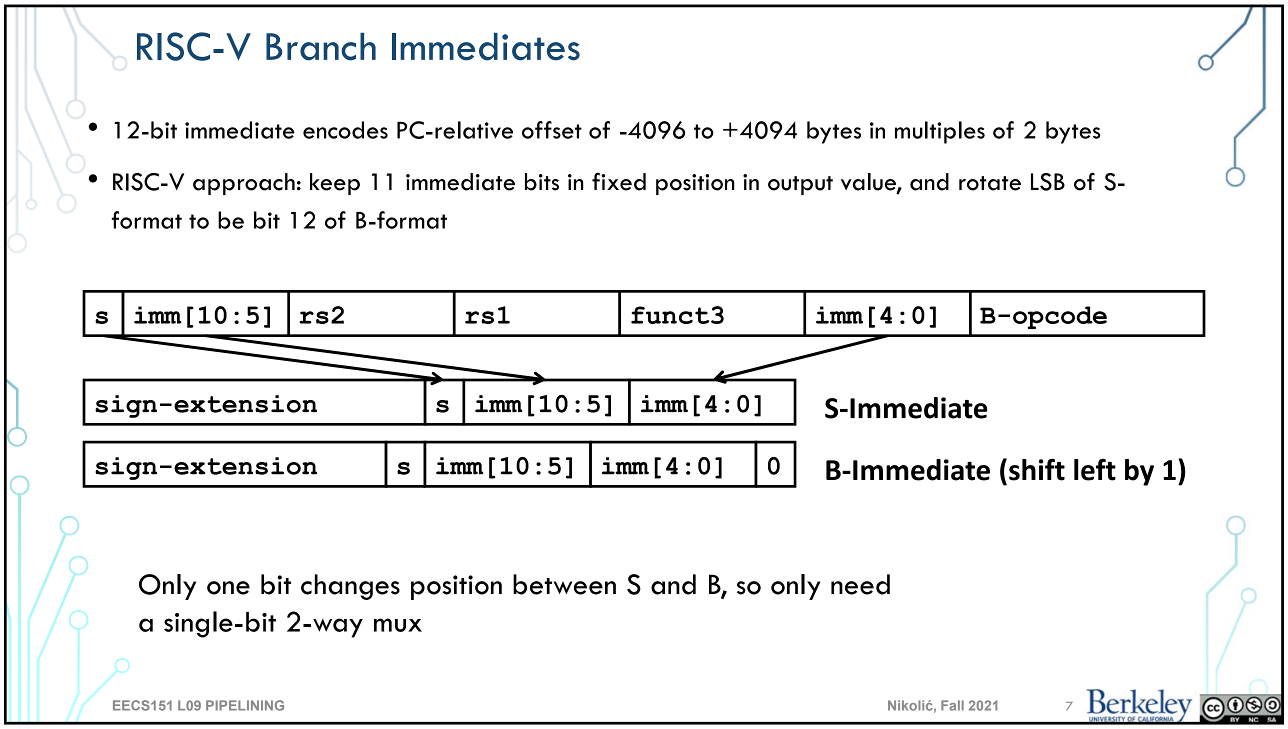
- E.g., `BEQ x1, x2, Label`
- Branches read two registers but don't write a register (similar to stores)
- How to encode label, i.e., where to branch to?

5

RISC-V Feature, n×16-bit instructions

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

6



RISC-V Branch Immediates

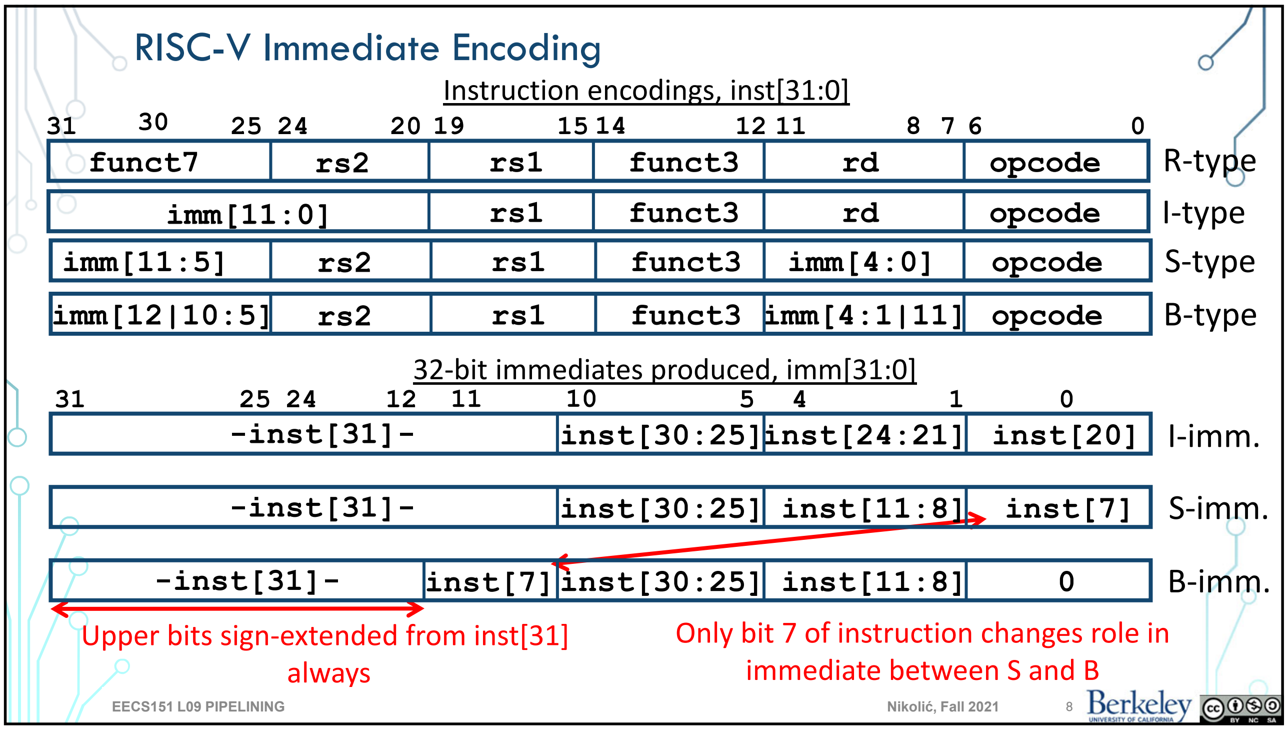
- 12-bit immediate encodes PC-relative offset of -4096 to +4094 bytes in multiples of 2 bytes
- RISC-V approach: keep 11 immediate bits in fixed position in output value, and rotate LSB of S-format to be bit 12 of B-format

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

sign-extension	s	imm[10:5]	imm[4:0]		S-Immediate
sign-extension	s	imm[10:5]	imm[4:0]	0	B-Immediate (shift left by 1)

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

7



RISC-V Immediate Encoding

Instruction encodings, inst[31:0]

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

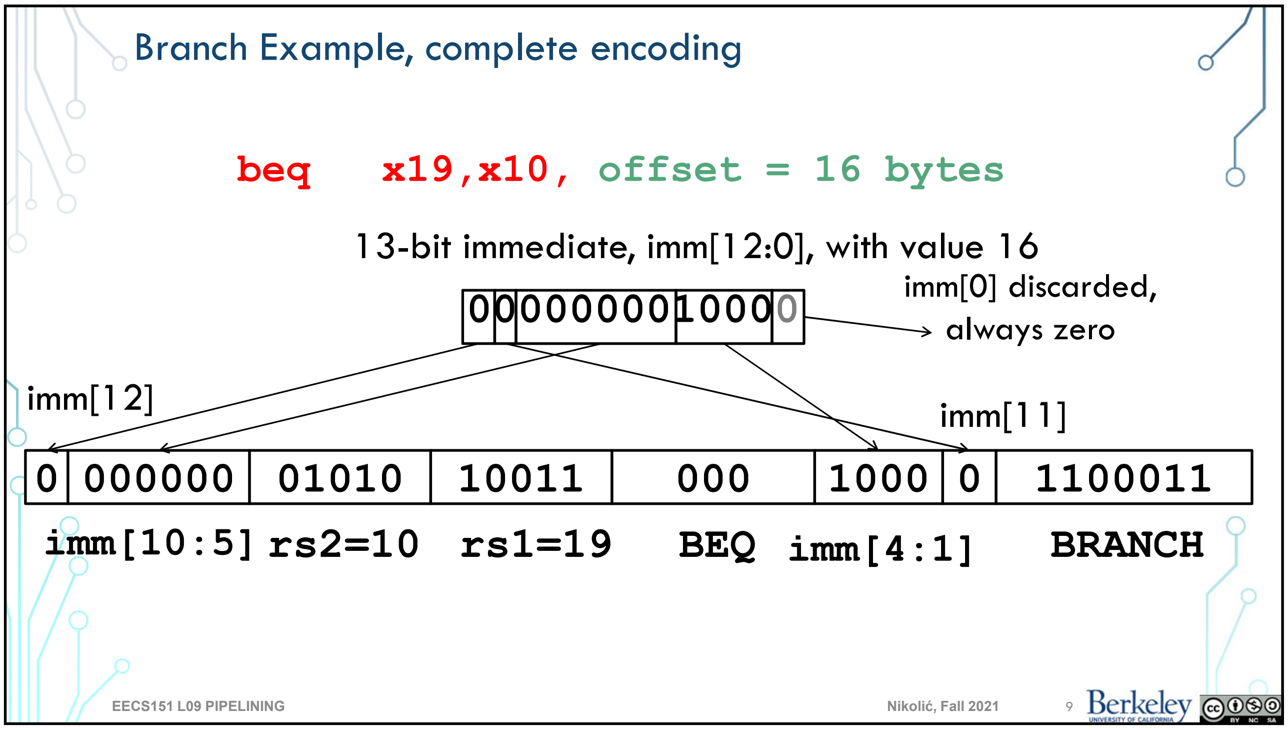
32-bit immediates produced, imm[31:0]

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

Upper bits sign-extended from inst[31] always

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

8



Branch Example, complete encoding

`beq x19,x10, offset = 16 bytes`

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

`0000000010000`

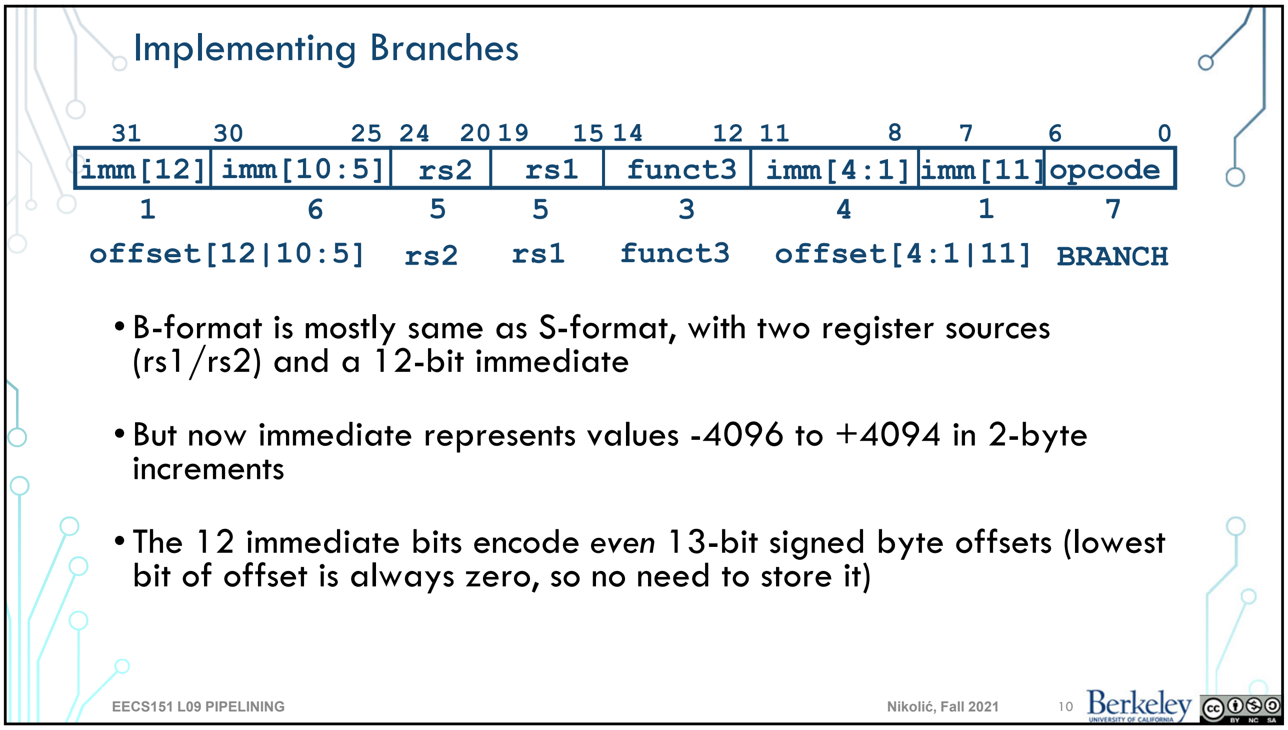
imm[0] discarded, always zero

imm[12]

imm[11]

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

9

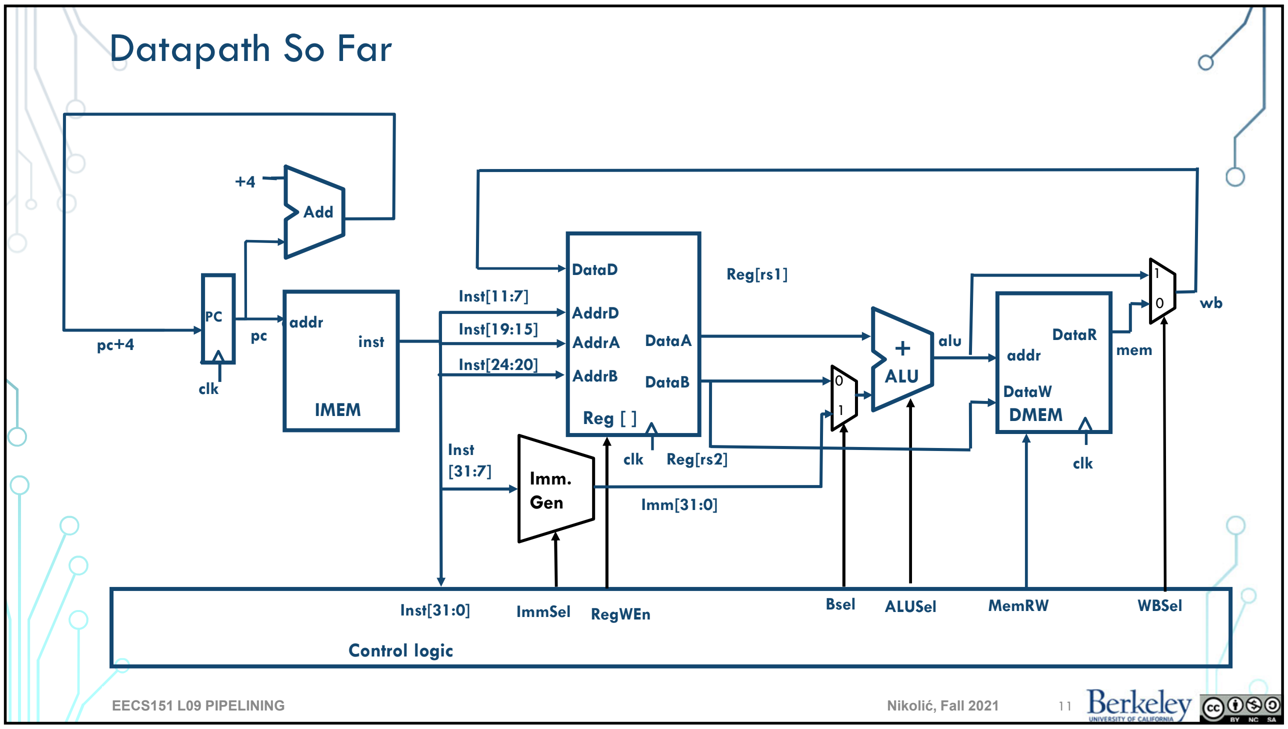


Implementing Branches

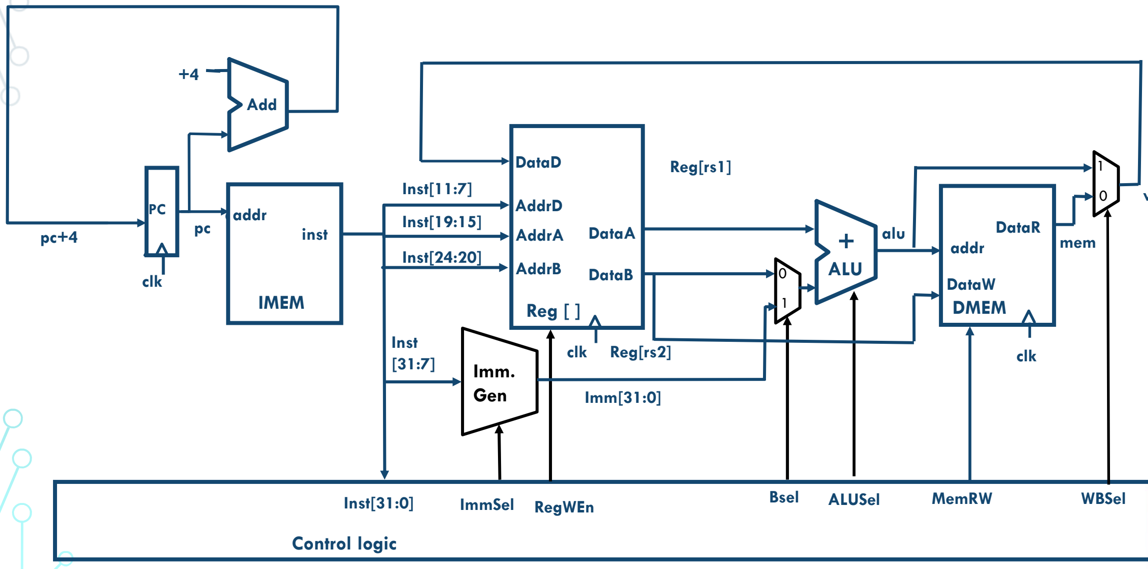
31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		rs2	rs1	funct3	offset[4:1|11]		BRANCH

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

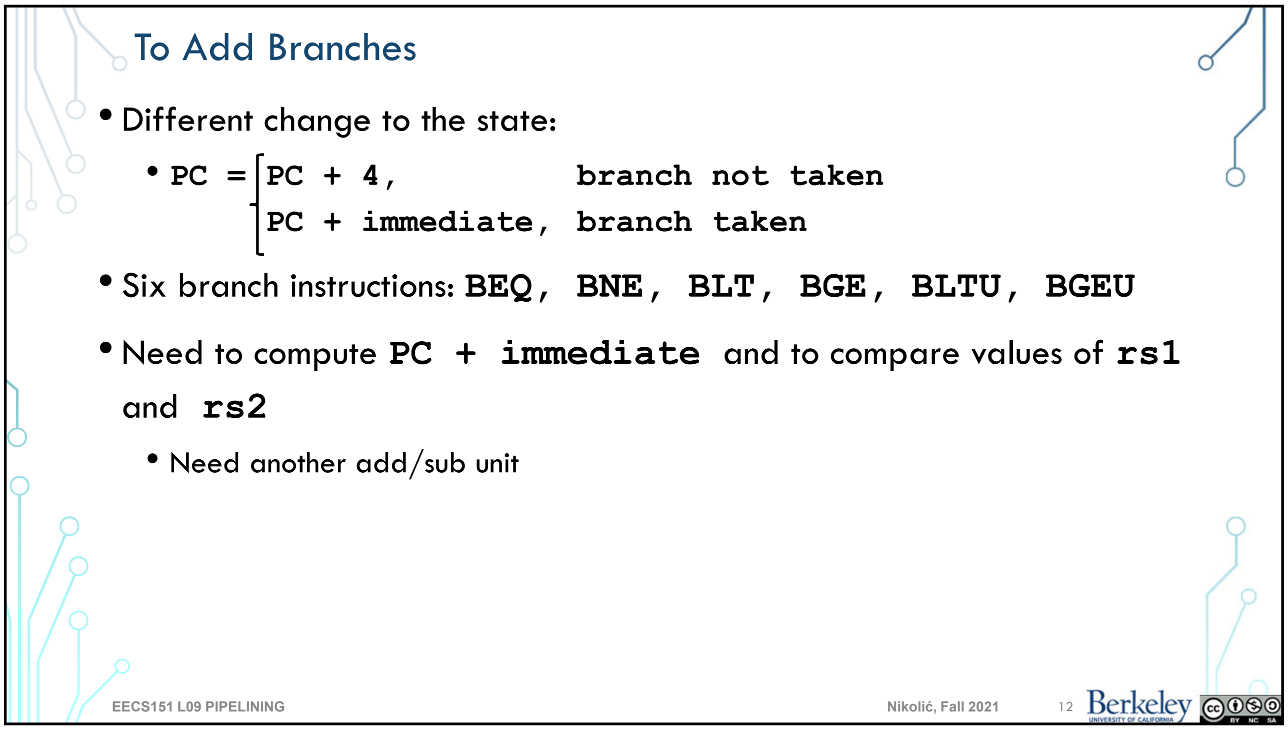
10



Datapath So Far



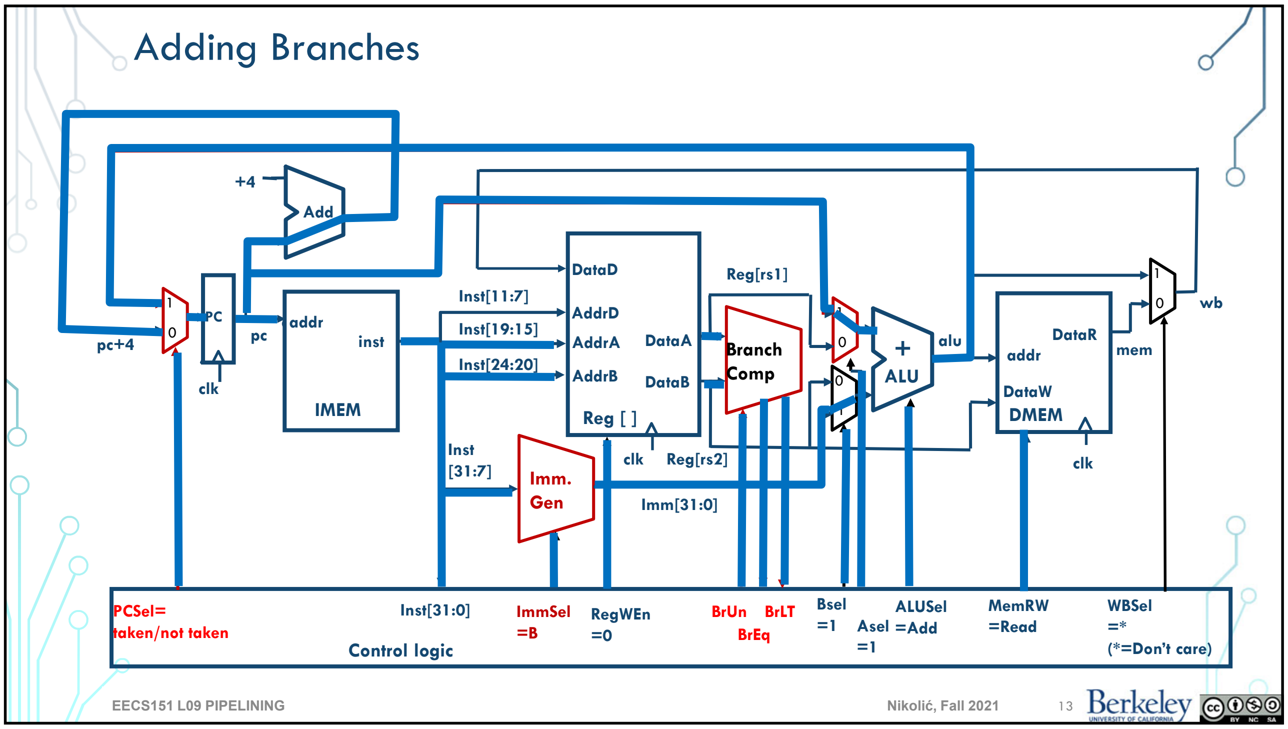
11



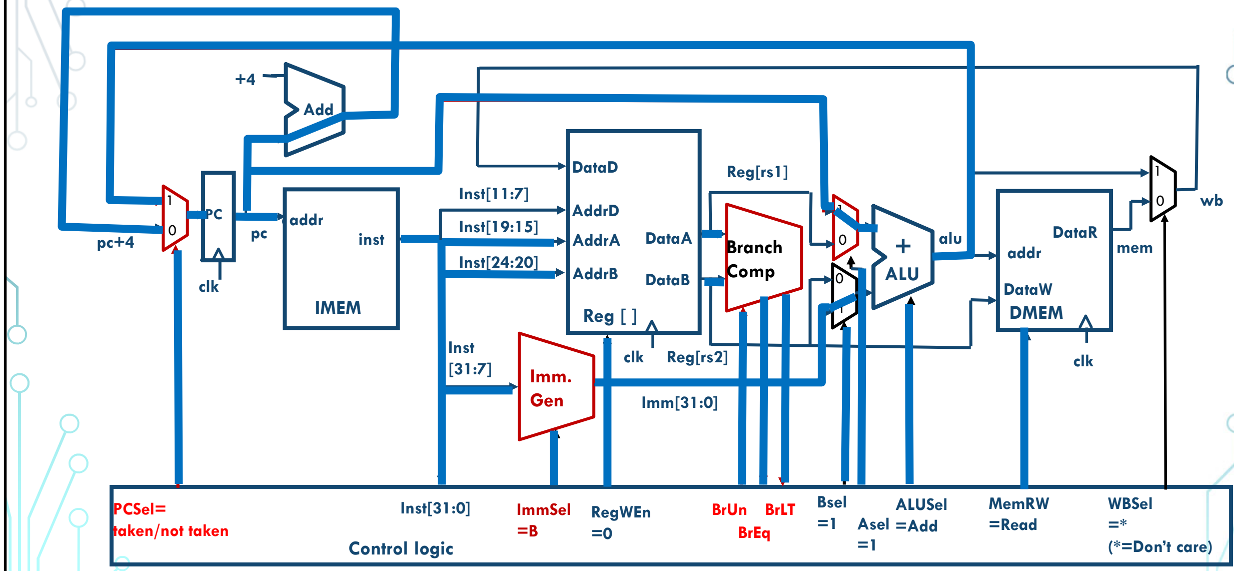
To Add Branches

- Different change to the state:
  - **PC** = $\begin{cases} \texttt{PC + 4,} & \texttt{branch not taken} \\ \texttt{PC + immediate,} & \texttt{branch taken} \end{cases}$
- Six branch instructions: **BEQ, BNE, BLT, BGE, BLTU, BGEU**
- Need to compute **PC + immediate** and to compare values of **rs1** and **rs2**
  - Need another add/sub unit

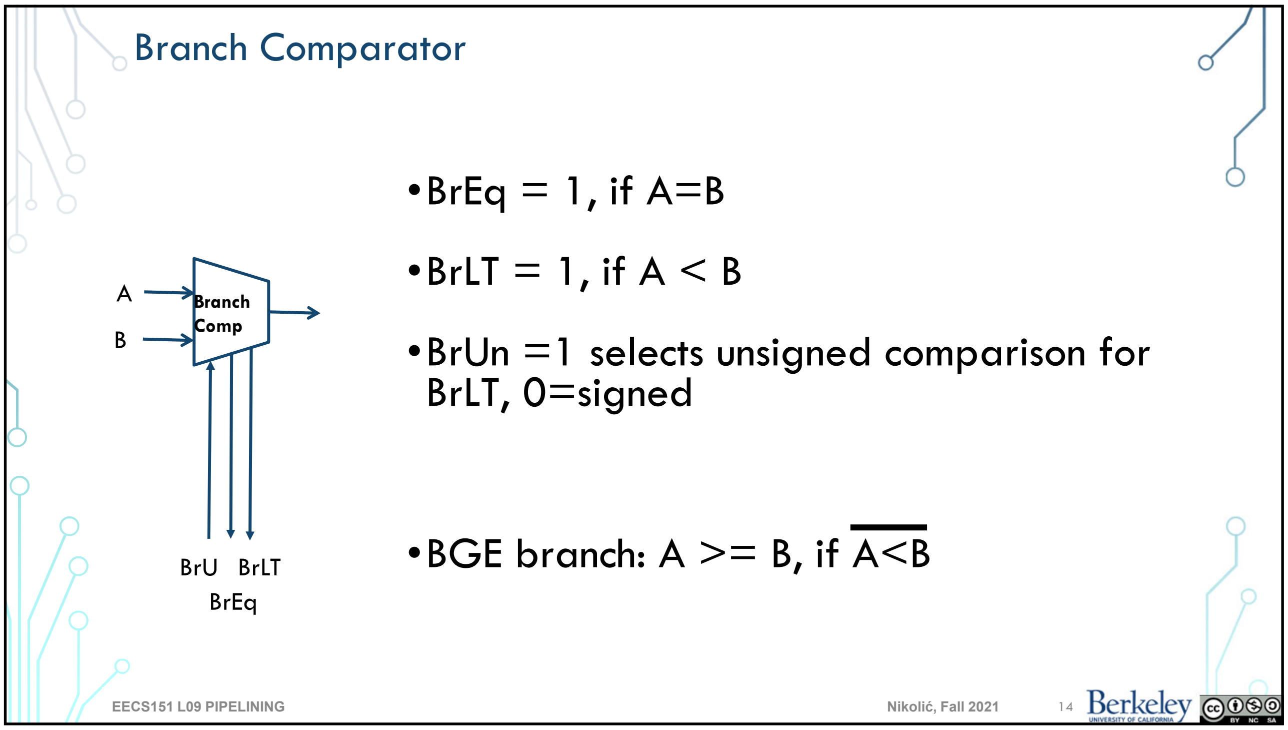
12



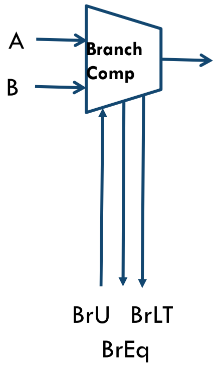
Adding Branches



13



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 $\overline{A<B}$

14

All RISC-V Branch Instructions

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

15

Administrivia

- Homework 4 is due next Monday
  - No new homework this week
  - Homework 5 will be posted next week, due after the midterm
- Lab 5 this week
  - No lab next week
  - Lab 6 (last) after the midterm
- Midterm 1on October 7, 7-8:30pm

16

RISC-V
J-Format Instructions

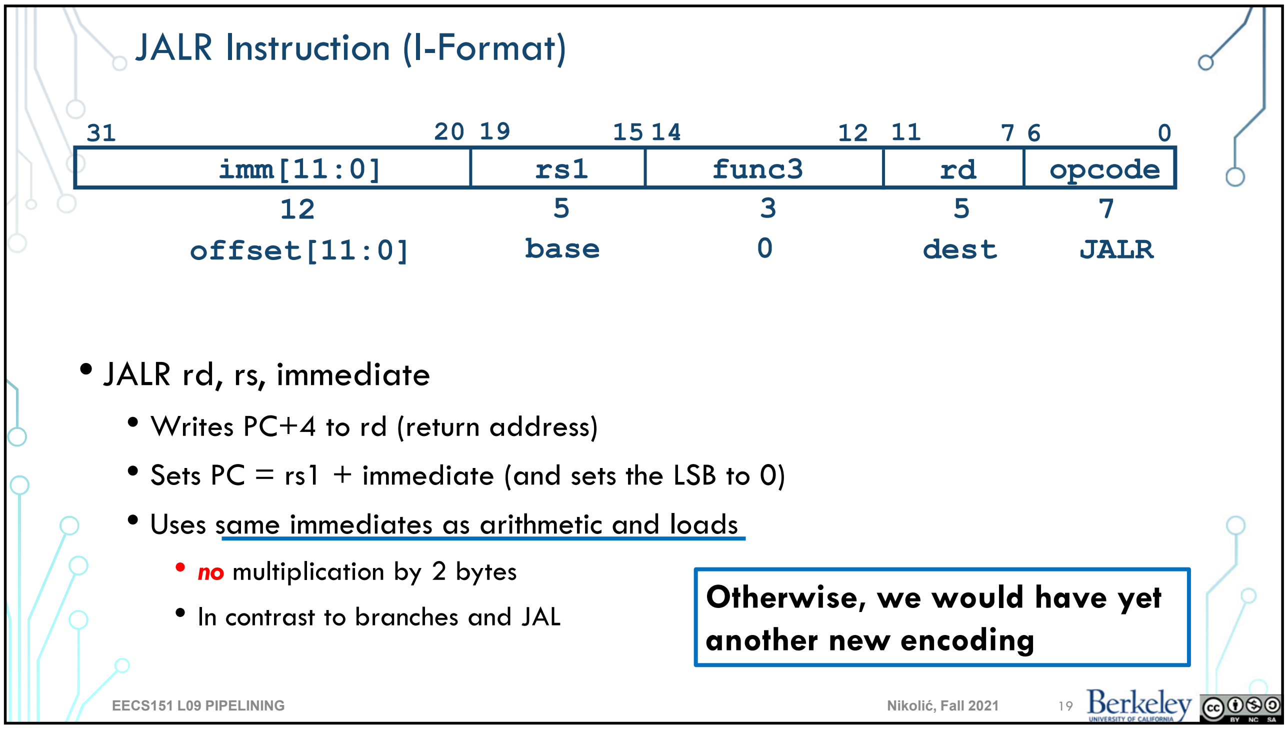
17

J-Format for Jump Instructions

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

- JAL saves PC+4 in register rd (the return address)
  - Assembler "`j`" jump is pseudo-instruction, uses JAL but sets `rd=x0` to discard return address
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

18



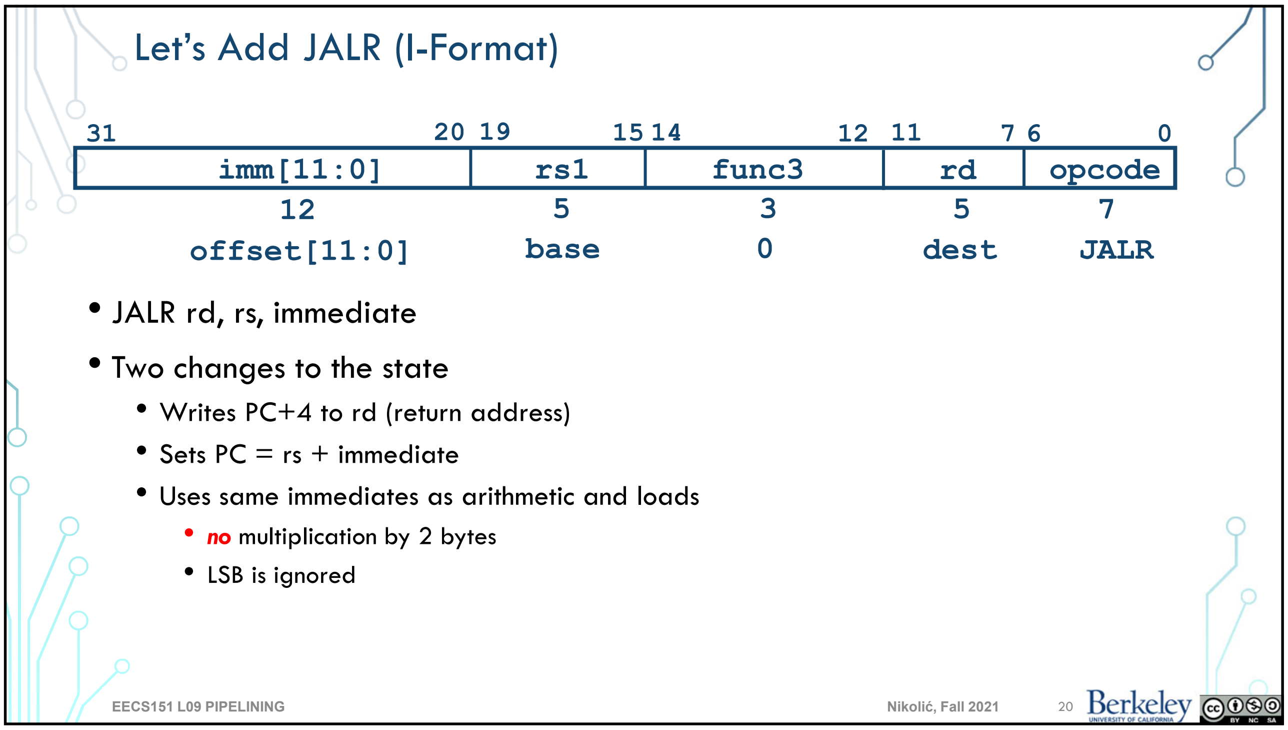
JALR Instruction (I-Format)

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

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

Otherwise, we would have yet another new encoding

19



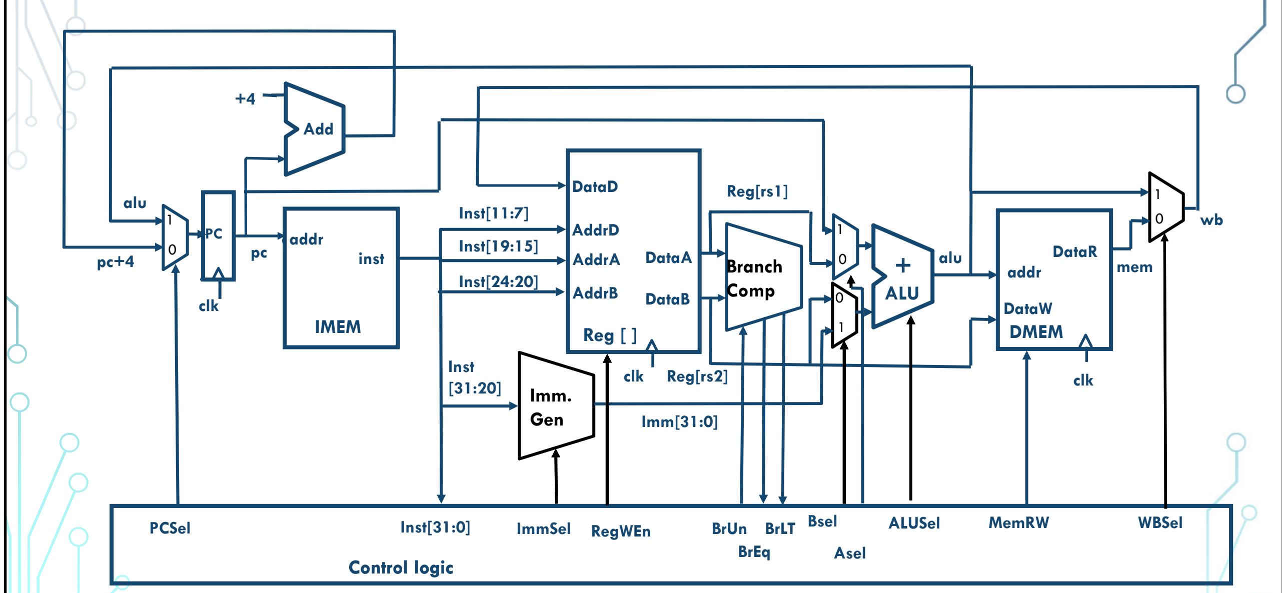
Let's Add JALR (I-Format)

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

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

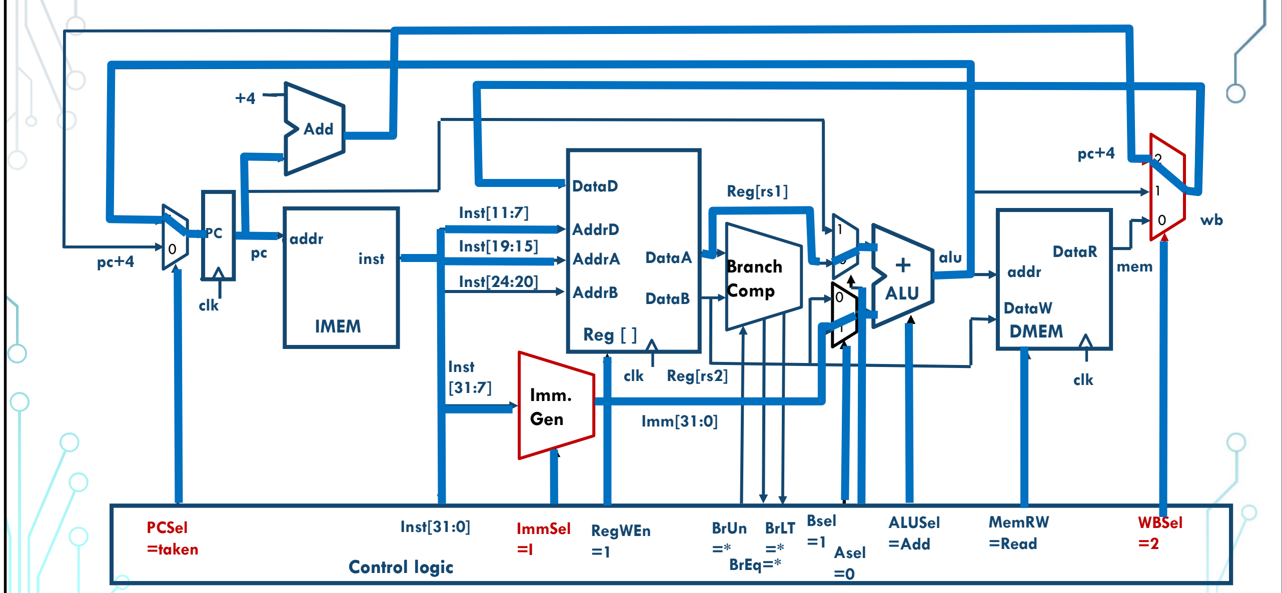
20

Datapath So Far, with Branches



21

Adding JALR



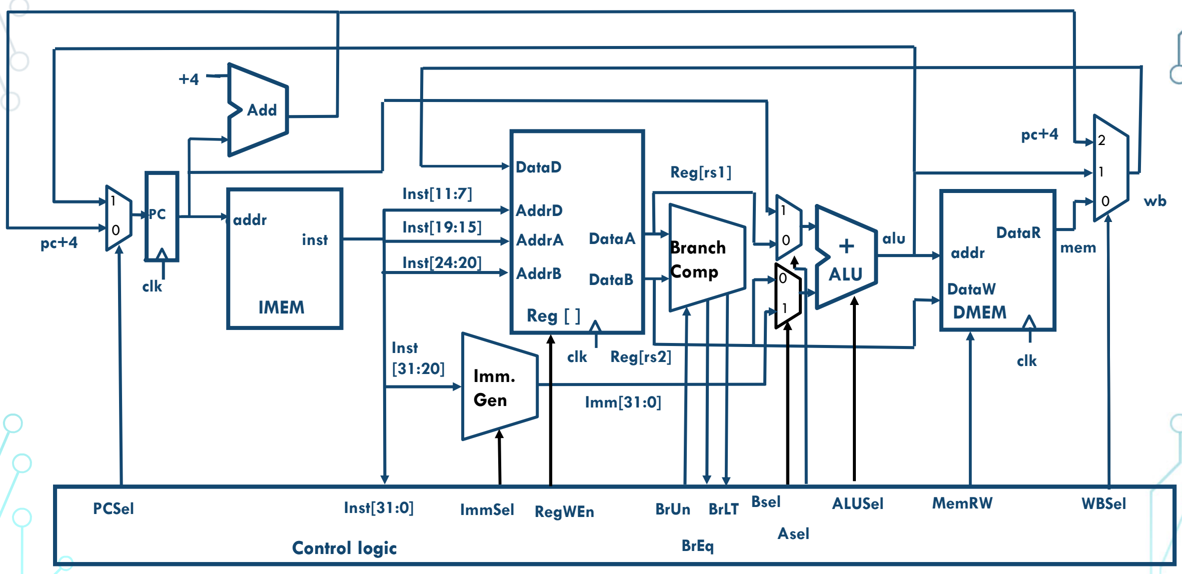
22

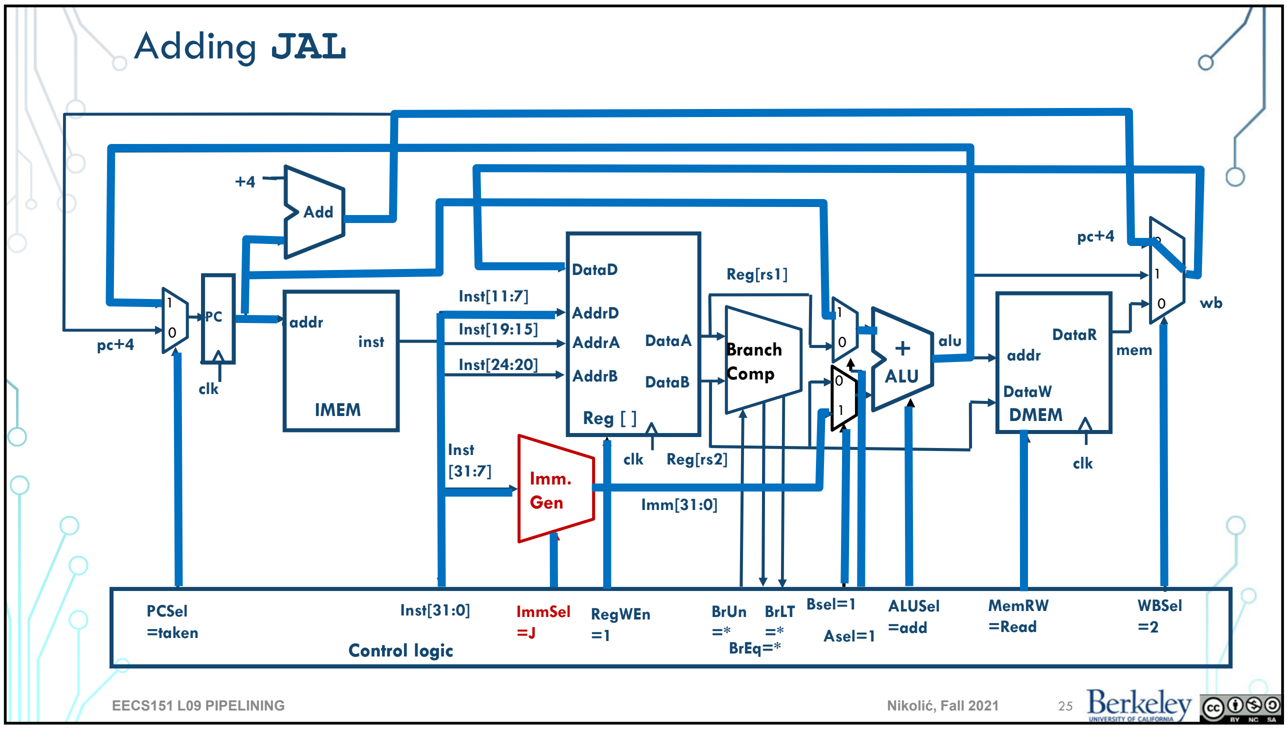
Adding **JAL**

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

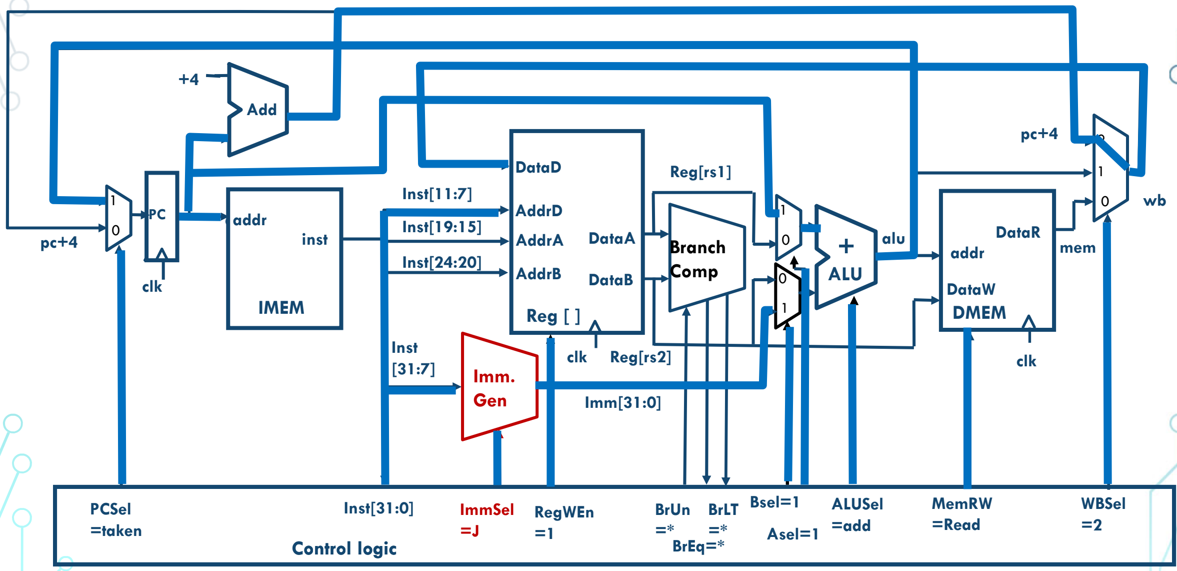
- JAL saves PC+4 in register rd (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

Datapath with **JALR**

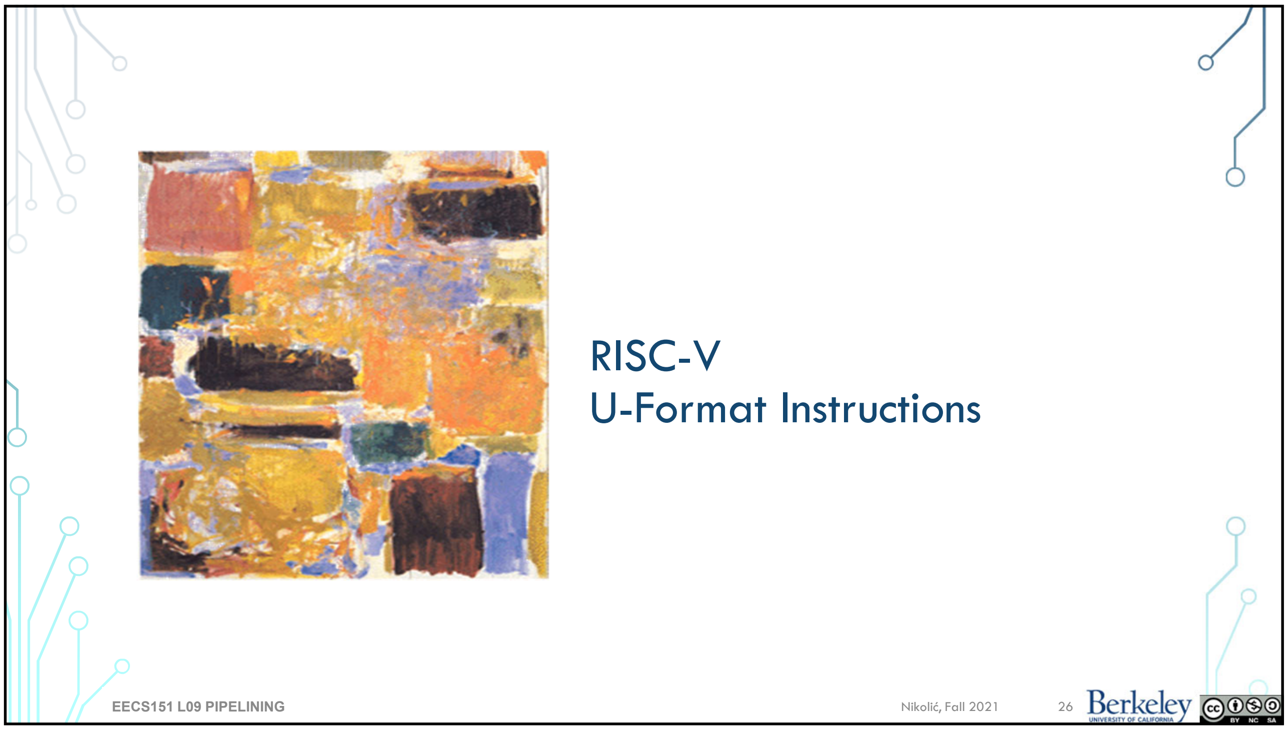




Adding **JAL**



25



RISC-V
U-Format Instructions

26

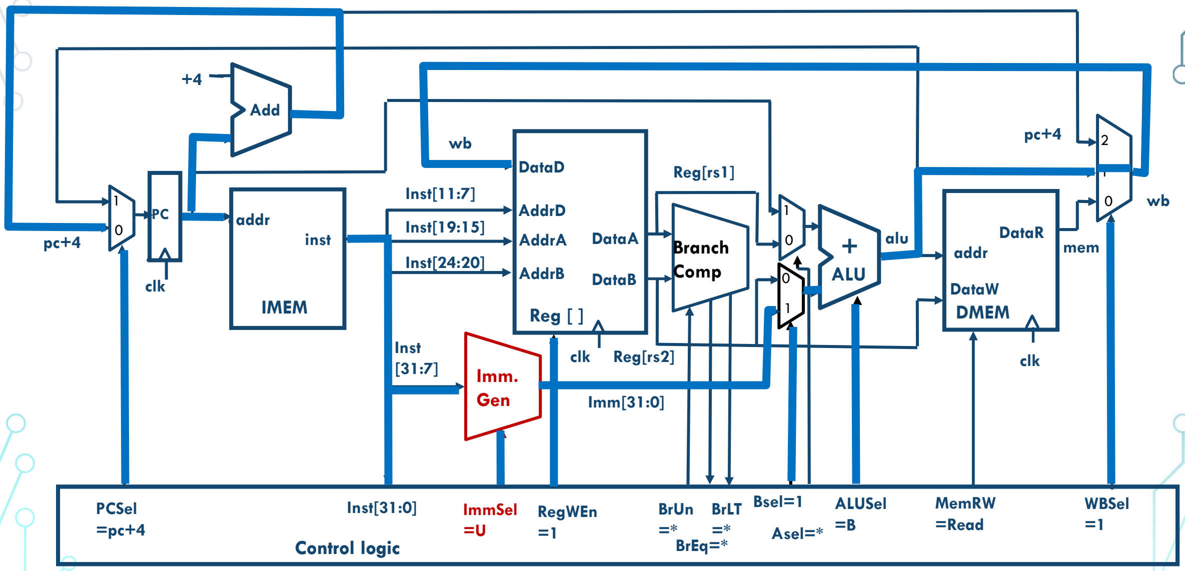
U-Format for "Upper Immediate" Instructions

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

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

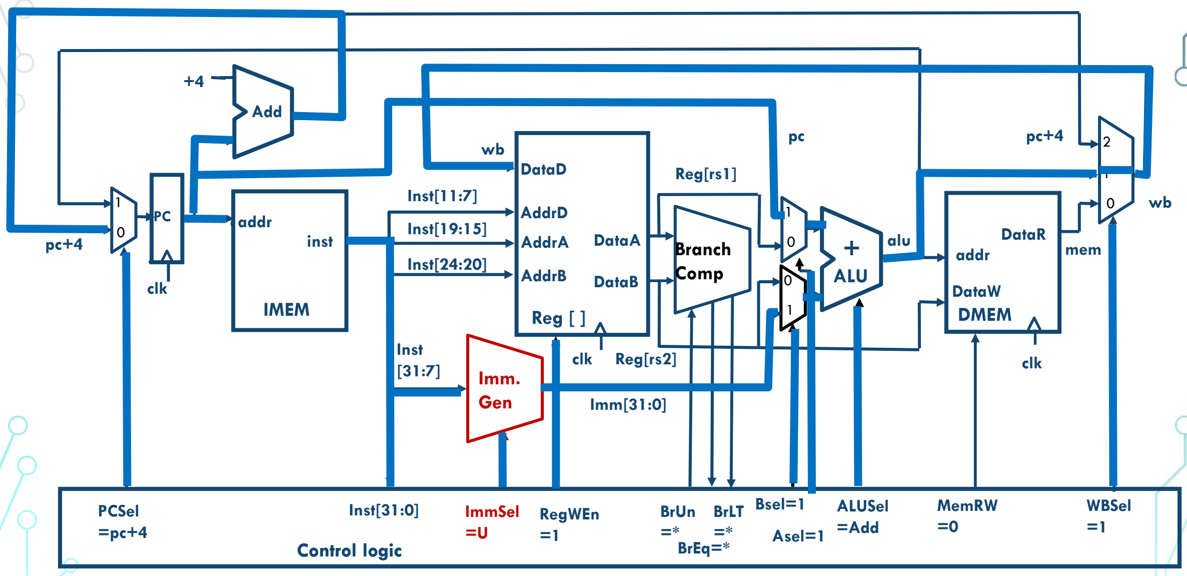
27

Implementing **`lui`**



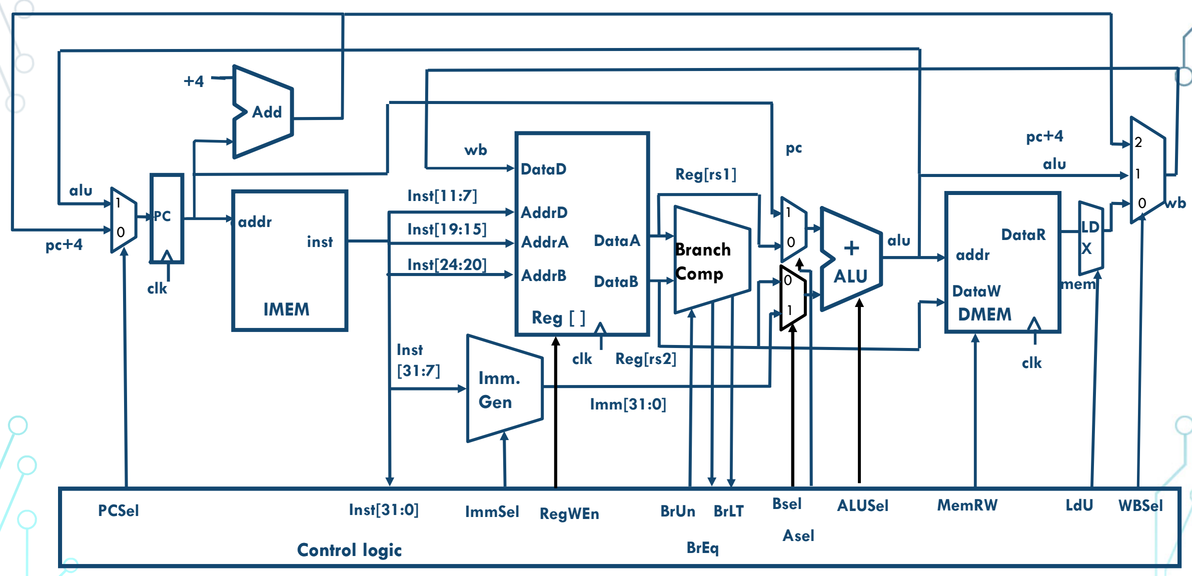
28

Implementing auipc



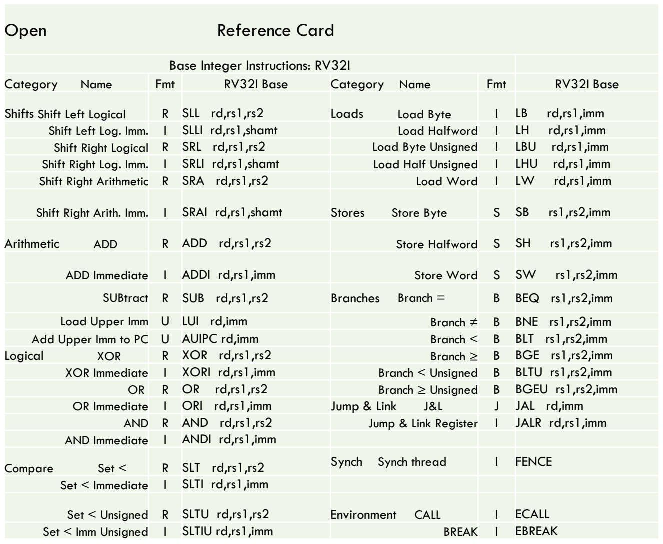
29

Complete RV32I Datapath!



30

Recap: Complete RV32I ISA



Open Reference Card

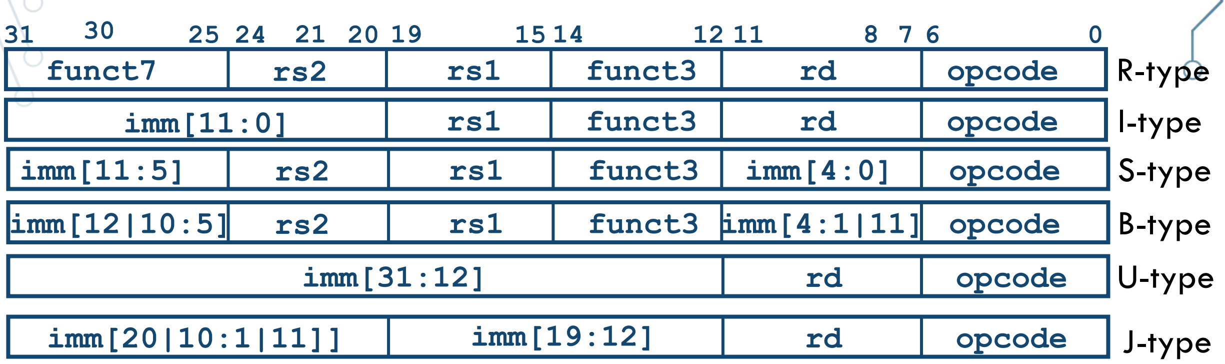
Base Integer Instructions: RV32I

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
	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
Logical	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	Set <	R	SLT rd,rs1,rs2	Synch	Synch thread	I	FENCE
	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

- 40 instructions are enough to run any C program

31

Summary of RISC-V Instruction Formats



31 30 25 24 21 20 19 15 14 12 11 8 7 6 0

funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11]]		imm[19:12]		rd	opcode	J-type

32

Control and Status Registers (CSRs)

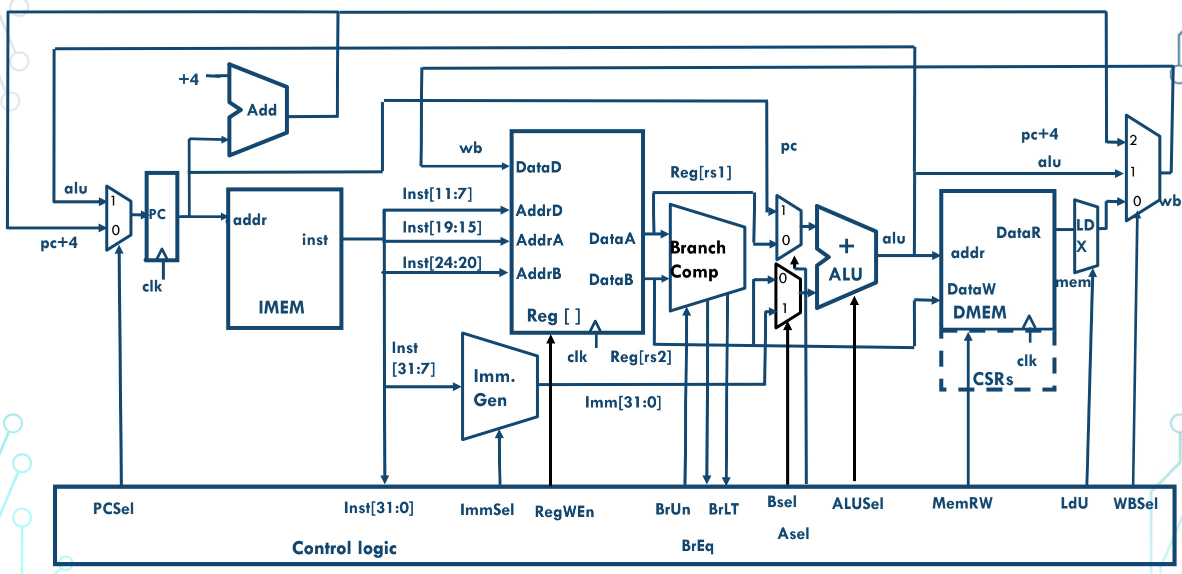
- 4096 CSRs in a separate address space

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

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

33

Add the **csrrw**!

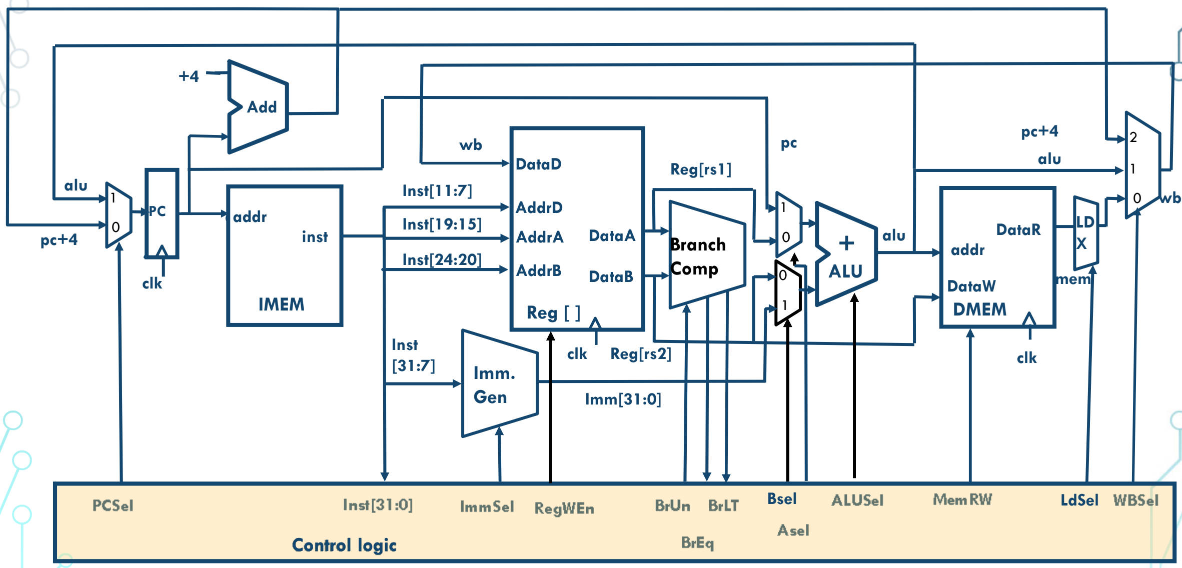


34

RISC-V Control Logic

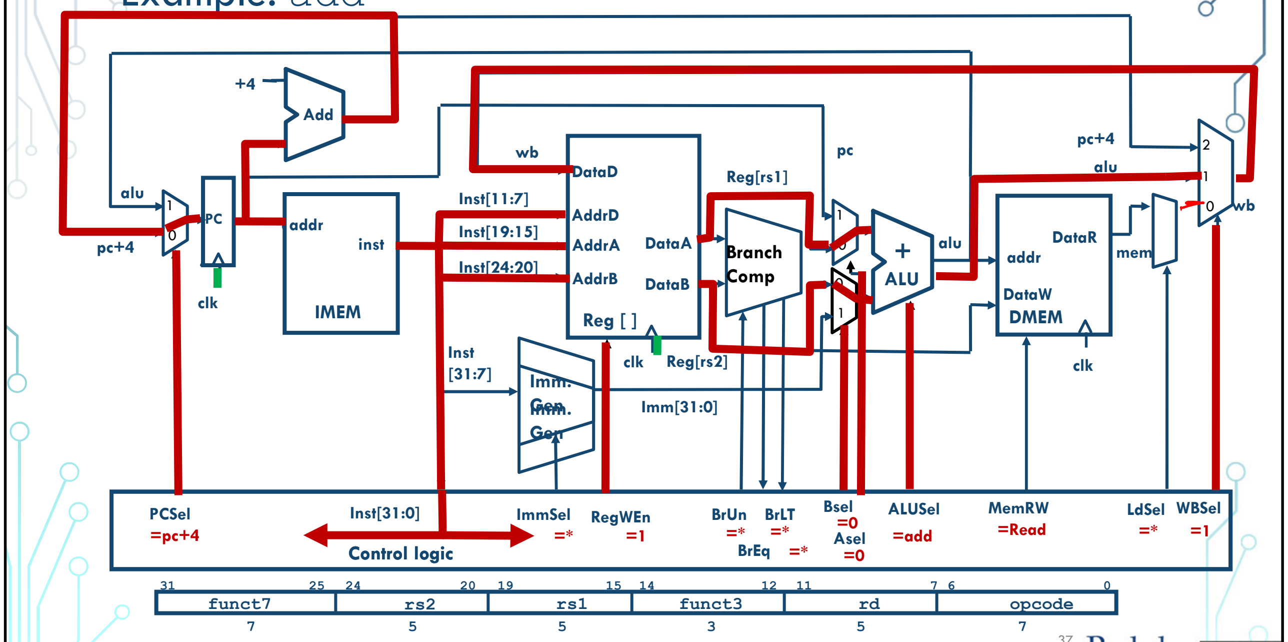
35

Complete RV32I Datapath with Control



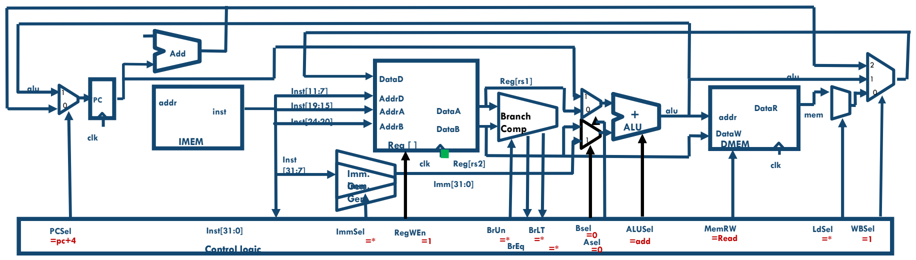
36

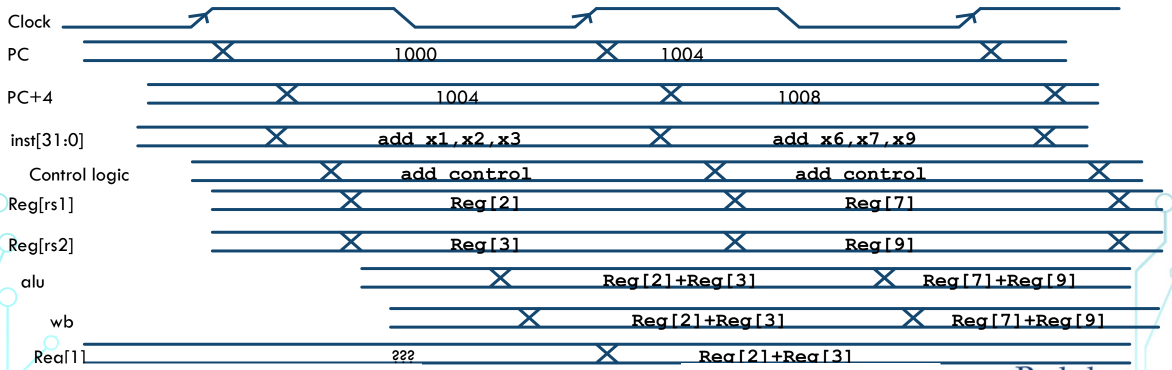
Example: add



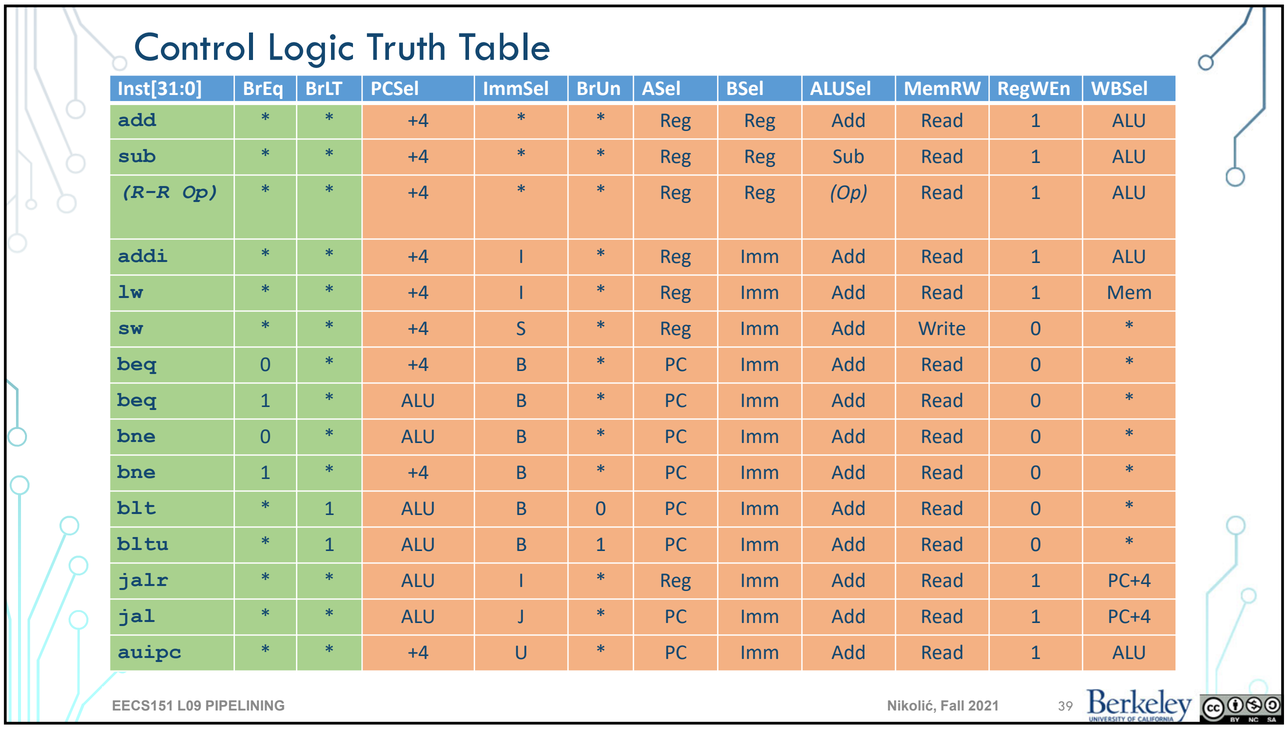
37

add Execution





38

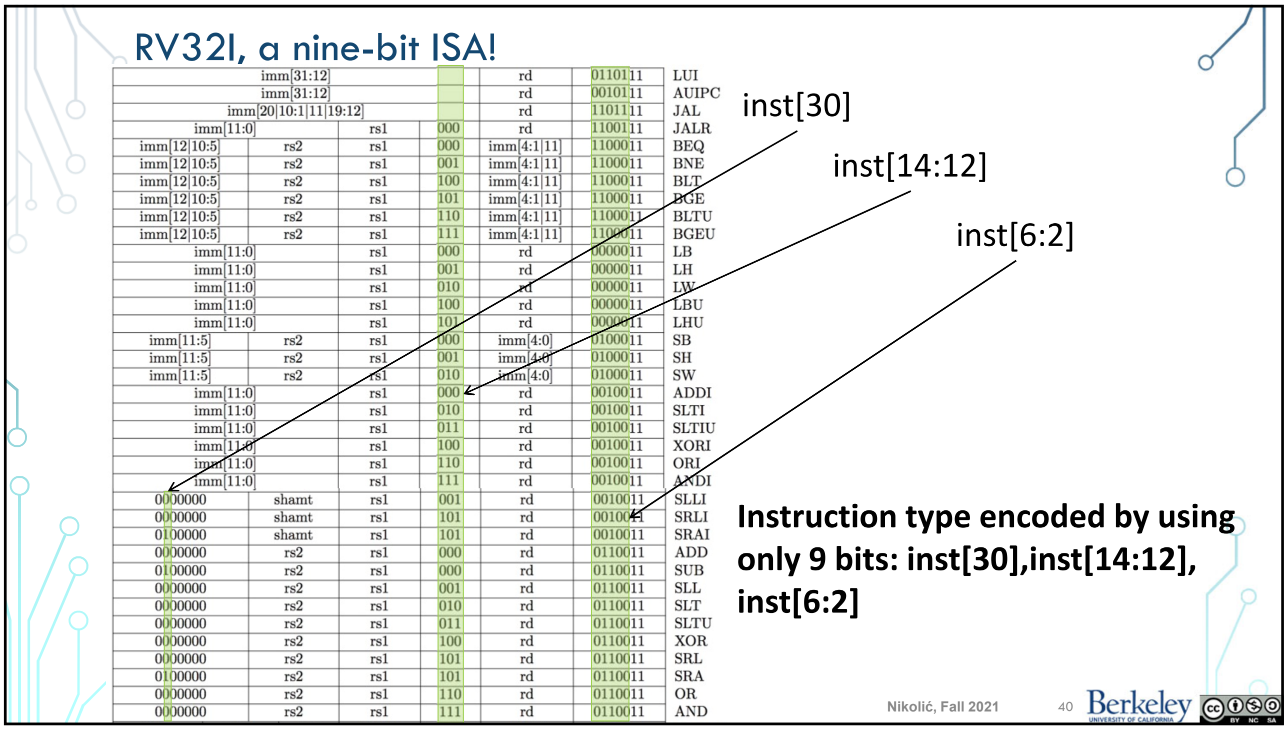


Control Logic Truth Table

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

EECS151 L09 PIPELINING   Nikolić, Fall 2021   39

39



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

inst[30]

inst[14:12]

inst[6:2]

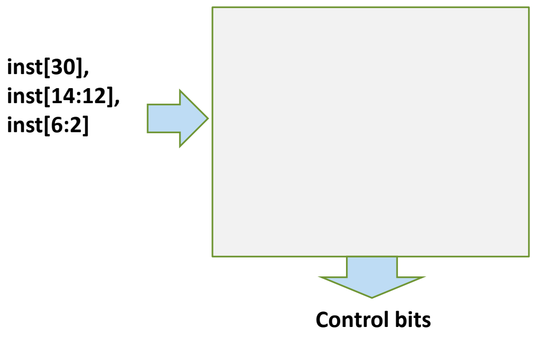
Instruction type encoded by using only 9 bits: inst[30],inst[14:12], inst[6:2]

Nikolić, Fall 2021   40

40

Control Realization Options

- ROM
  - “Read-Only Memory”
  - Regular structure
  - Can be easily reprogrammed
    - fix errors
    - add instructions
- Combinatorial Logic
  - Start from a truth table
  - More compact, faster
  - Use synthesis tools



41

Combinational Logic Control

- Decoder is typically hierarchical
  - First decode opcode, and figure out instruction type
  - E.g. branches are Inst[6:2] = 11000
  - Then determine the actual instruction
  - Inst[30] + Inst[14:12]
- Modularity helps simplify and speed up logic
  - Narrows problem space for logic synthesis

42

Combinational Logic Control

- Simple 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

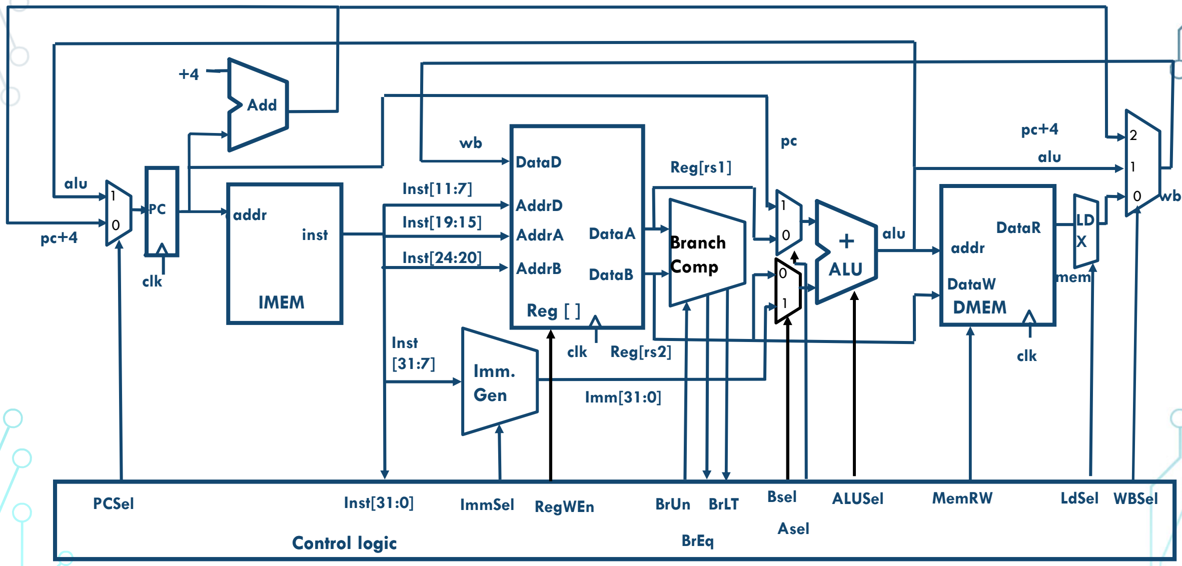
inst[14:13]

inst[12]	00	01	11	10
0				
1				

- **How to decode whether BrUn is 1?**
  - **BrUn = Inst [13] • Branch**
  - **Branch = Inst[6] • Inst[5] • !Inst[4] • !Inst[3] • !Inst[2]**

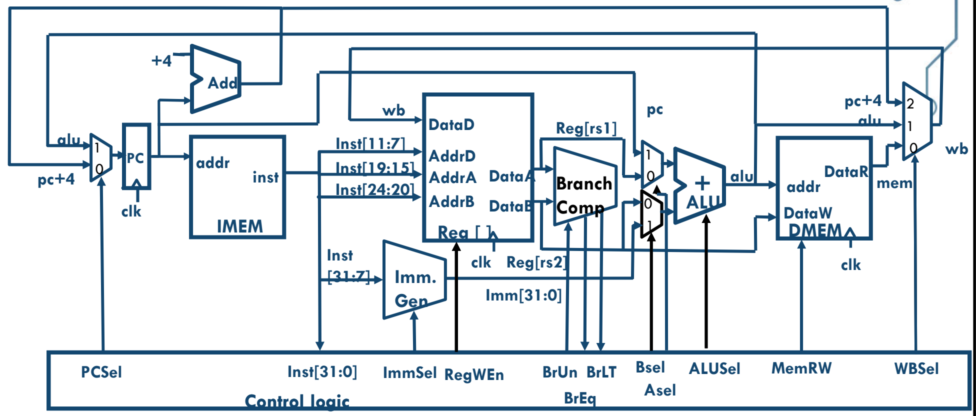
43

Complete RV32I Datapath with Control



44

Peer Instruction(s): Critical Path yellkey: crime



Critical path for a addi

R[rd] = R[rs1]+imm

1) $t_{clk\text{-}q} + t_{Add} + t_{IMEM} + t_{Reg} + t_{BComp} + t_{ALU} + t_{DMEM} + t_{mux} + t_{Setup}$

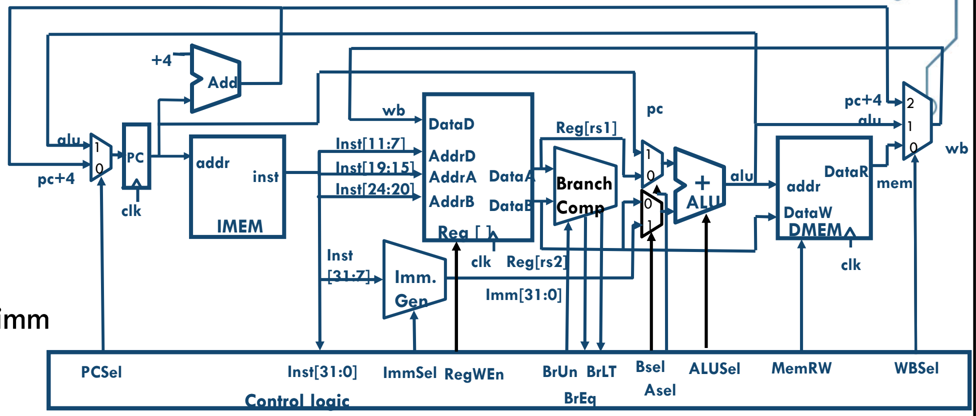
2) $t_{clk\text{-}q} + t_{IMEM} + \max\{t_{Reg}, t_{Imm}\} + t_{ALU} + 2t_{mux} + t_{Setup}$

3) $t_{clk\text{-}q} + t_{IMEM} + \max\{t_{Reg}, t_{Imm}\} + t_{ALU} + 3t_{mux} + t_{DMEM} + t_{Setup}$

4) None of the above

45

Peer Instruction(s): Critical Path yellkey: physical



Critical path for a addi

R[rd] = R[rs1]+imm

1) $t_{clk\text{-}q} + t_{Add} + t_{IMEM} + t_{Reg} + t_{BComp} + t_{ALU} + t_{DMEM} + t_{mux} + t_{Setup}$

2) $t_{clk\text{-}q} + t_{IMEM} + \max\{t_{Reg}, t_{Imm}\} + t_{ALU} + 2t_{mux} + t_{Setup}$

3) $t_{clk\text{-}q} + t_{IMEM} + \max\{t_{Reg}, t_{Imm}\} + t_{ALU} + 3t_{mux} + t_{DMEM} + t_{Setup}$

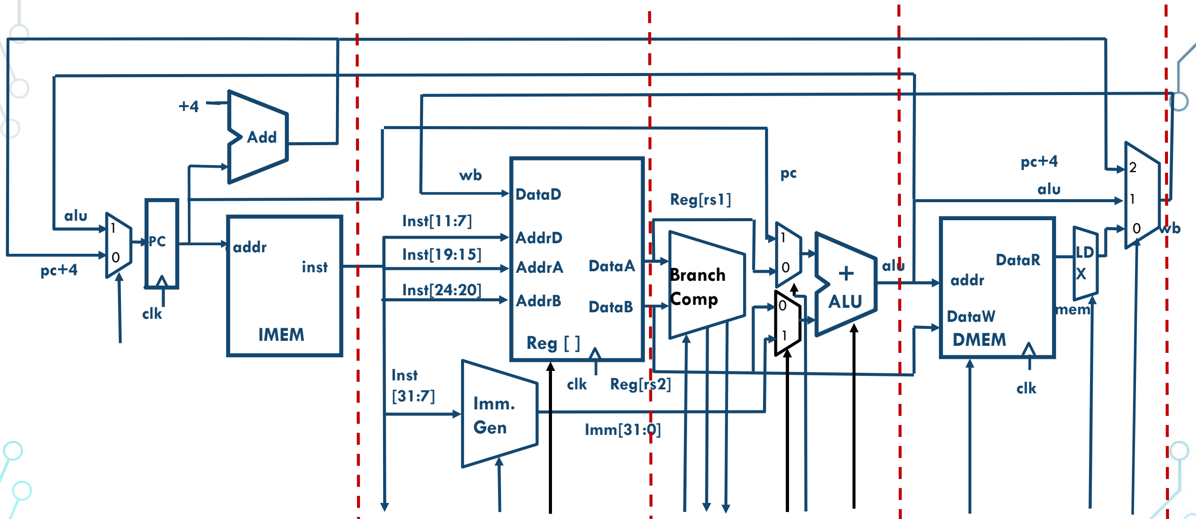
4) None of the above

46

Pipelining

EECS151 L09 PIPELINING

Complete RV32I Datapath with Control



Instruction Fetch
(F)

Instruction Decode/
Register Read
(D)

ALU Execute
(X)

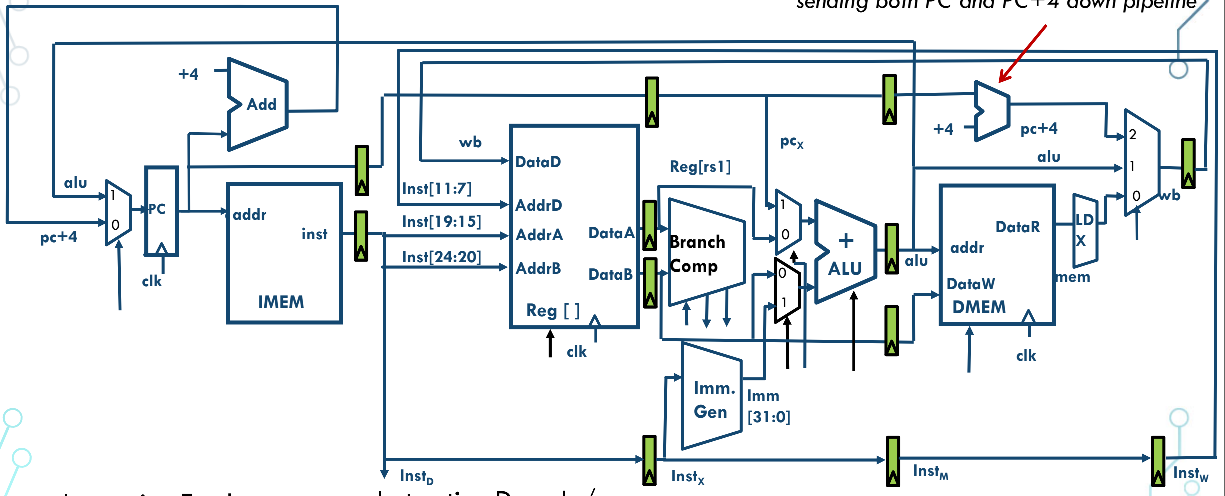
Memory Access
(M)

Write
Back
(W)

EECS151 L09 PIPELINING

Pipelining RV32I Datapath

Recalculate PC+4 in M stage to avoid sending both PC and PC+4 down pipeline



Instruction Fetch
(F)

Instruction Decode/
Register Read
(D)

ALU Execute
(X)

Memory Access
(M)

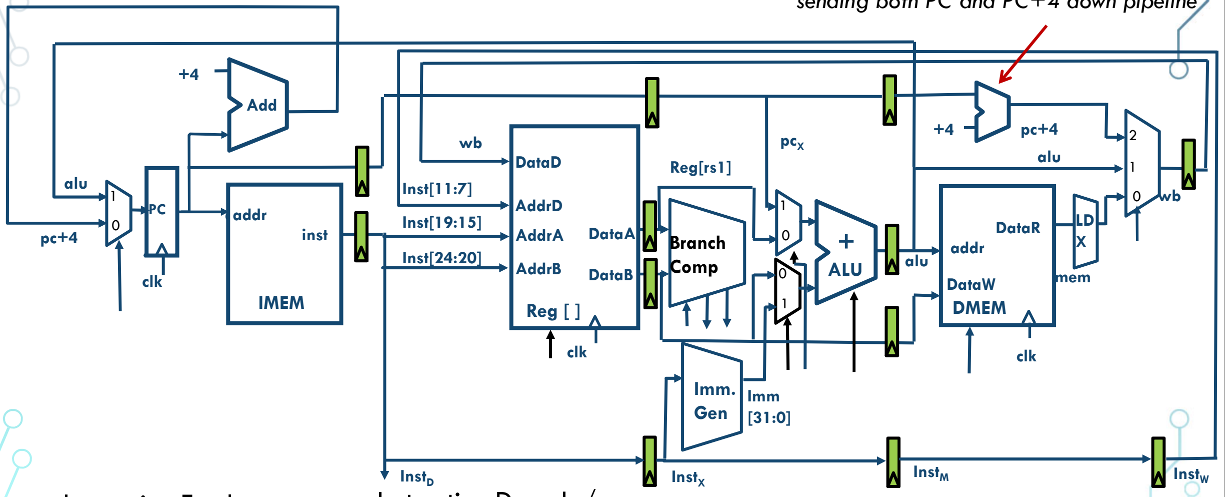
Write
Back
(W)

instruction along with data

:tly in each stage

Pipelining RV32I Datapath

Recalculate PC+4 in M stage to avoid sending both PC and PC+4 down pipeline



Instruction Fetch
(F)

Instruction Decode/
Register Read
(D)

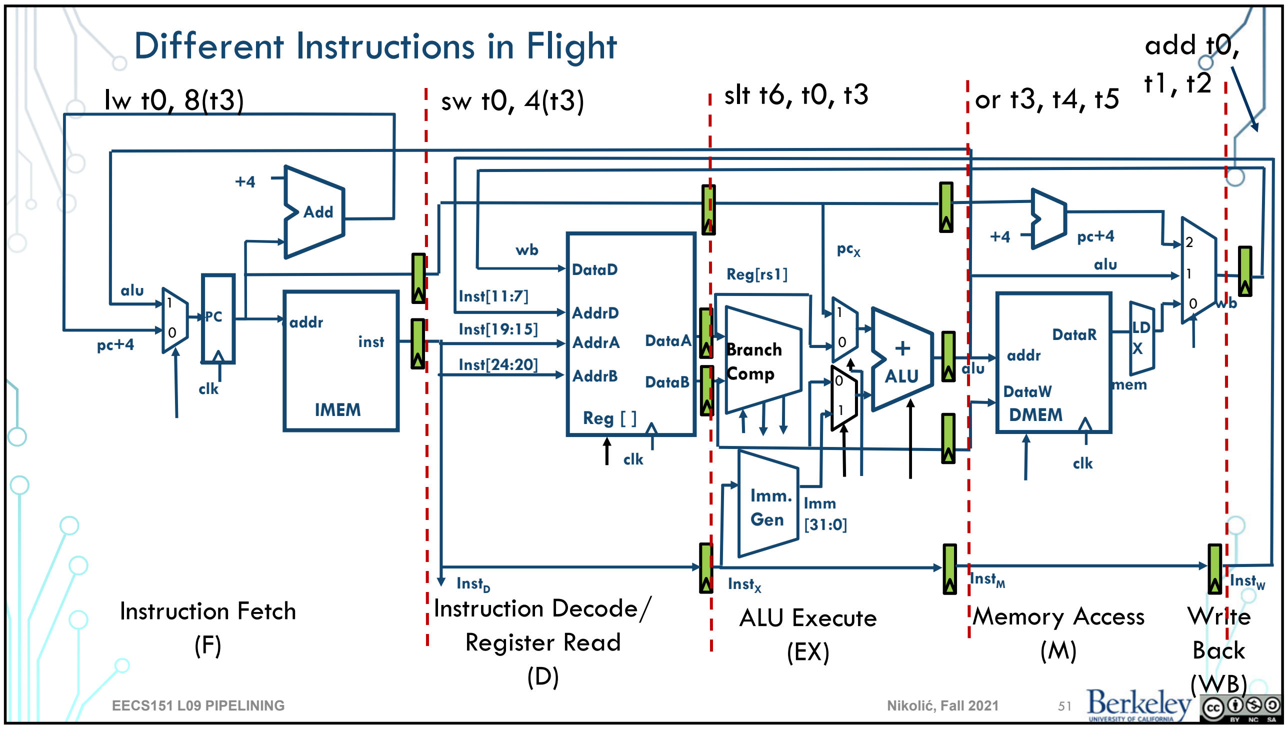
ALU Execute
(X)

Memory Access
(M)

Write
Back
(W)

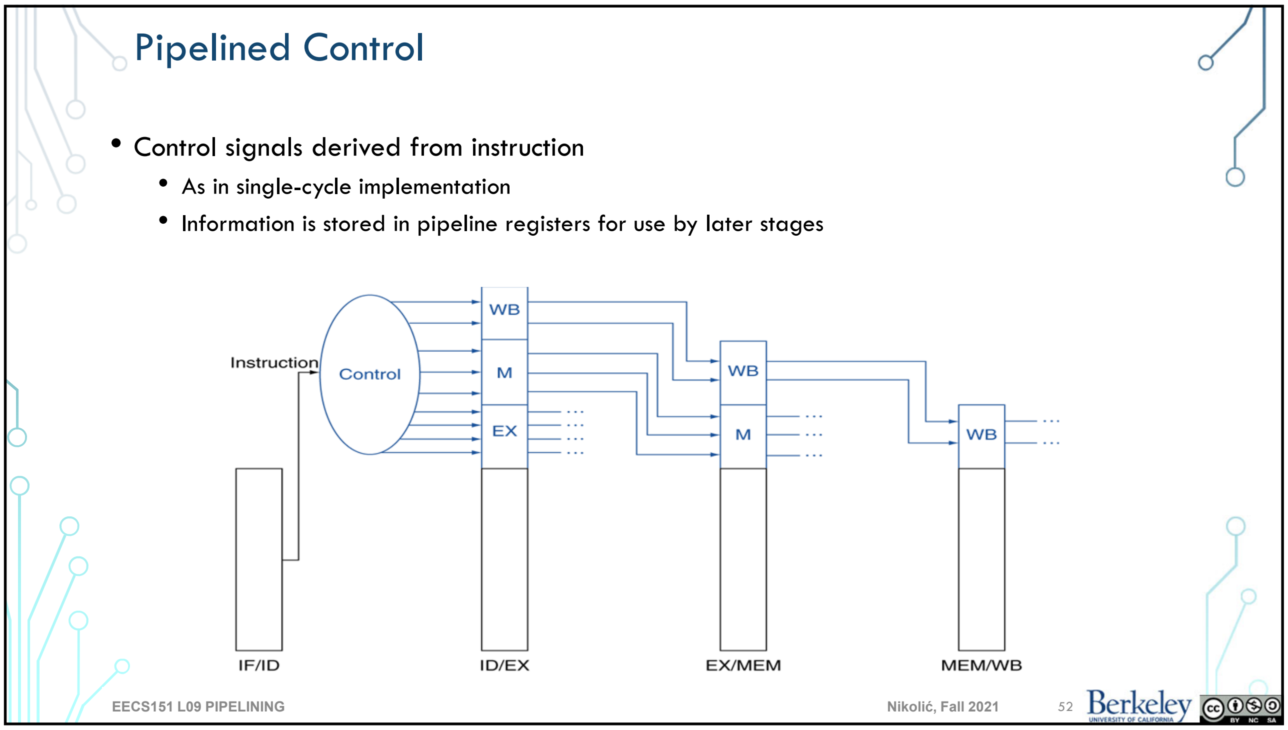
instruction along with data

:tly in each stage



Different Instructions in Flight

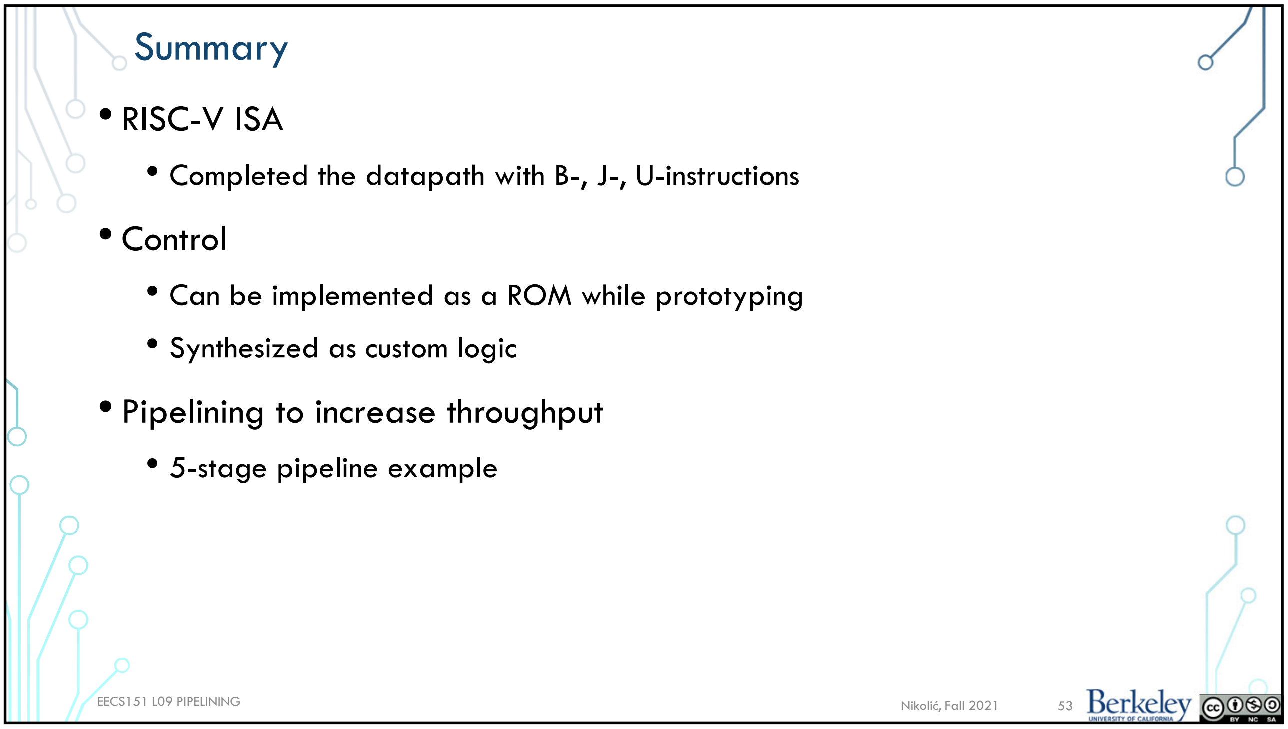
51



Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
  - Information is stored in pipeline registers for use by later stages

52



Summary

- RISC-V ISA
  - Completed the datapath with B-, J-, U-instructions
- Control
  - Can be implemented as a ROM while prototyping
  - Synthesized as custom logic
- Pipelining to increase throughput
  - 5-stage pipeline example