inst.eecs.berkeley.edu/~eecs151

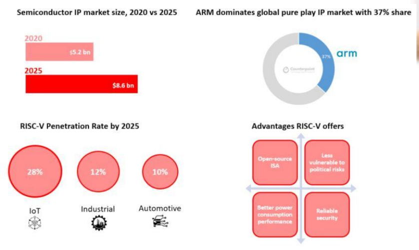
EECS151 : Introduction to Digital Design and ICs

Lecture 8 – RISC-V ISA

Bora Nikolić

September 21, 2021, EET Asia
RISC-V to Shake Up $8.6B Semiconductor IP Market

RISC-V is now a rising star in the industry, largely due to its open-source advantage, better power consumption performance promise, reliable security functions and lower political risk impact yet.



https://www.eetasia.com/risc-v-to-shake-up-8-6b-semiconductor-ip-market/

1

Review

- Finite state machines: Common example of sequential logic
  - Moore's machine: Output depends only on the current state
  - Mealy's machine: Output depends on the current state and the input
- Large state machines can be factored
- Common Verilog patterns for FSMs
- Common job interview questions ☺

2

Building a RISC-V Processor

3

Berkeley RISC-V ISA

www.riscv.org

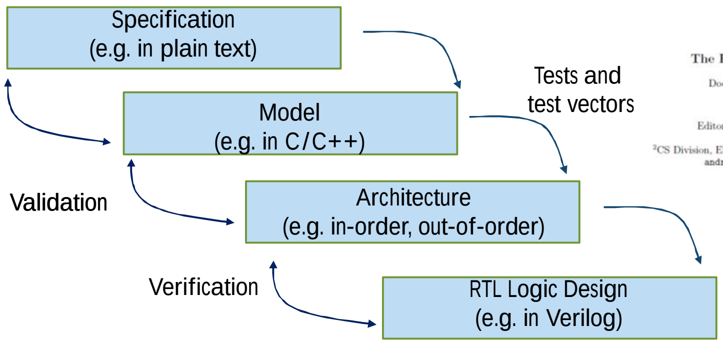
- An open, license-free ISA
  - Runs GCC, LLVM, Linux distributions, …
  - RV32, RV64, and RV128 variants for 32b, 64b, and 128b address spaces
- Originally developed for teaching classes at Berkeley, now widely adopted
- Base ISA only ~40 integer instructions
- Extensions provide full general-purpose ISA, including IEEE-754/2008 floating-point
- Designed for extension, customization
- Developed at UC Berkeley, now maintained by RISC-V Foundation
- Open and commercial implementations
- RISC-V ISA, datapath, and control covered in CS61C; summarized here

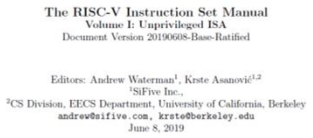
THE LINLEY GROUP 2016 ANALYSTS' CHOICE BEST TECHNOLOGY

4

RISC-V Processor Design

- Spec: Unprivileged ISA, RV32I (and a look at RV64I)

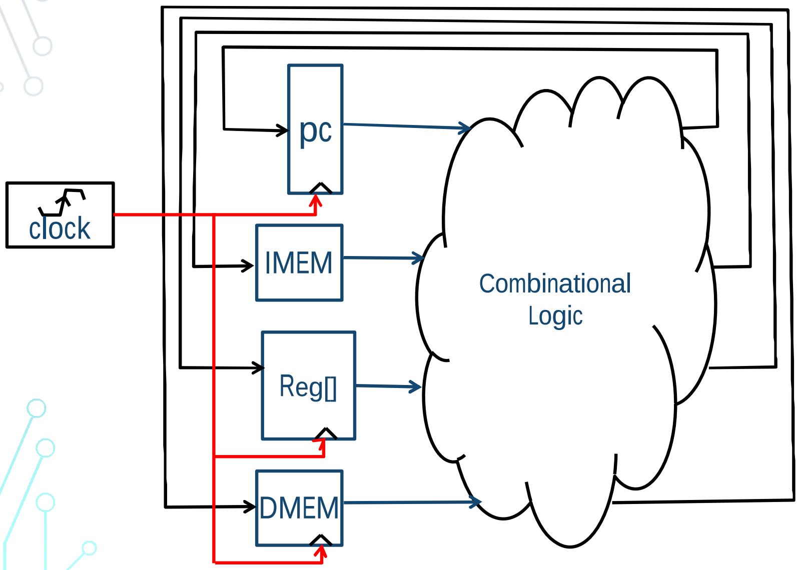



The RISC-V Instruction Set Manual
Volume I: Unprivileged ISA
Document Version 20190608-Base-Ratified

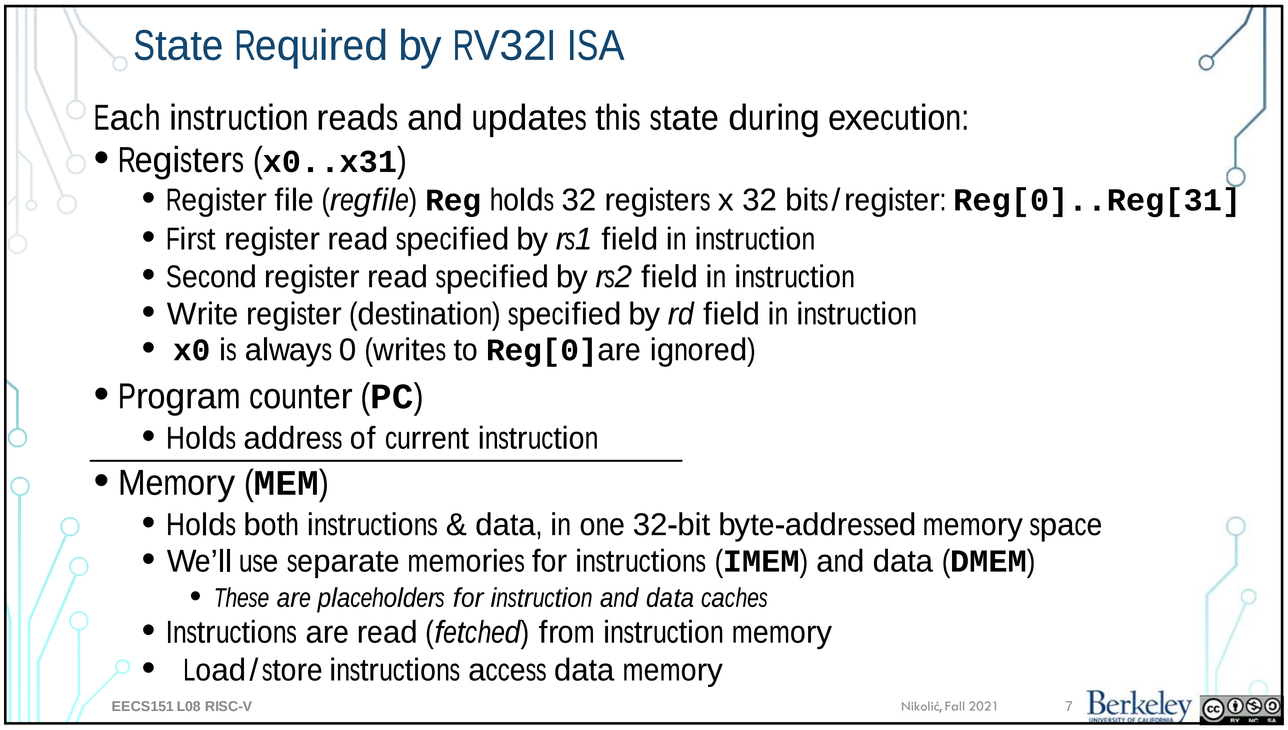
Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu
June 8, 2019

- Tests provided as a part of the project
- Architecture: Single-cycle and pipelined in-order processor
  - Expanded from CS61C

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

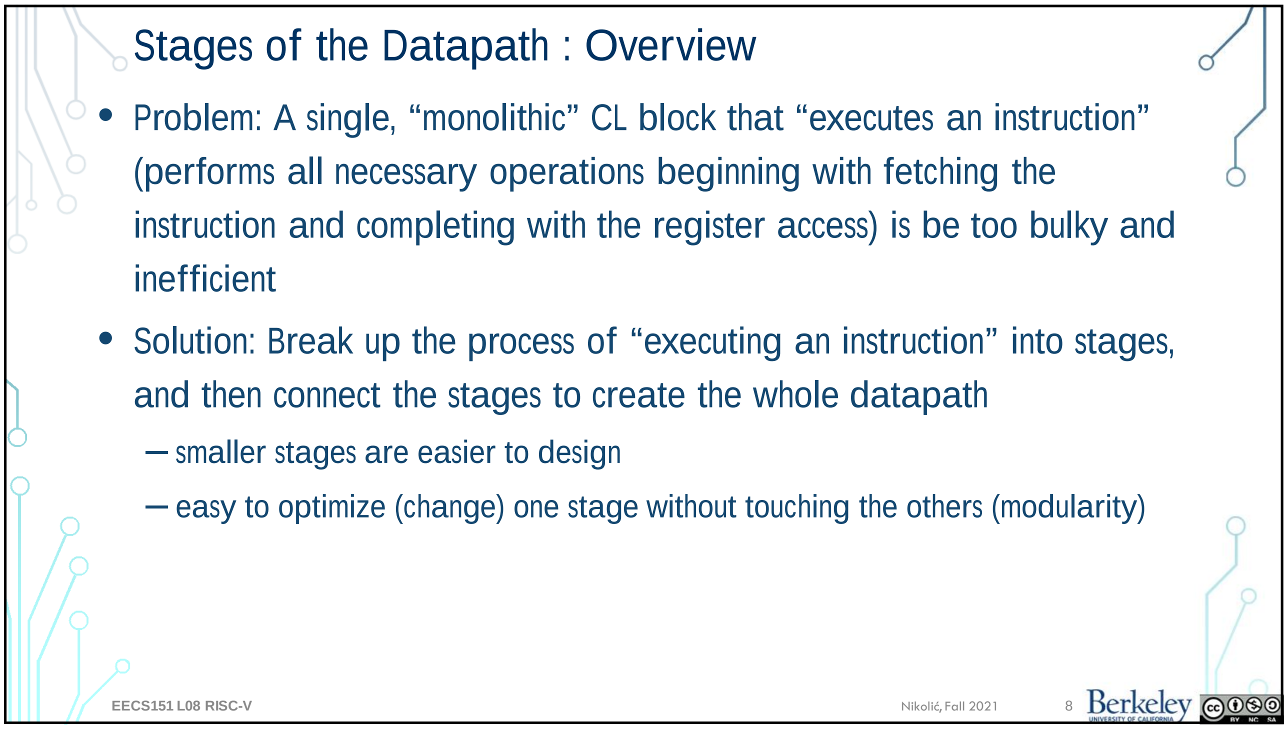


State Required by RV32I ISA

Each instruction reads and updates this state during execution:

- Registers (**`x0..x31`**)
  - Register file (*regfile*) **`Reg`** holds 32 registers x 32 bits/register: **`Reg[0]..Reg[31]`**
  - First register read specified by *rs1* field in instruction
  - Second register read specified by *rs2* field in instruction
  - Write register (destination) specified by *rd* field in instruction
  - **`x0`** is always 0 (writes to **`Reg[0]`** are ignored)
- Program counter (**`PC`**)
  - Holds address of current instruction
- Memory (**`MEM`**)
  - Holds both instructions & data, in one 32-bit byte-addressed memory space
  - We'll use separate memories for instructions (**`IMEM`**) and data (**`DMEM`**)
    - *These are placeholders for instruction and data caches*
  - Instructions are read (*fetched*) from instruction memory
  - Load/store instructions access data memory

7



Stages of the Datapath : Overview

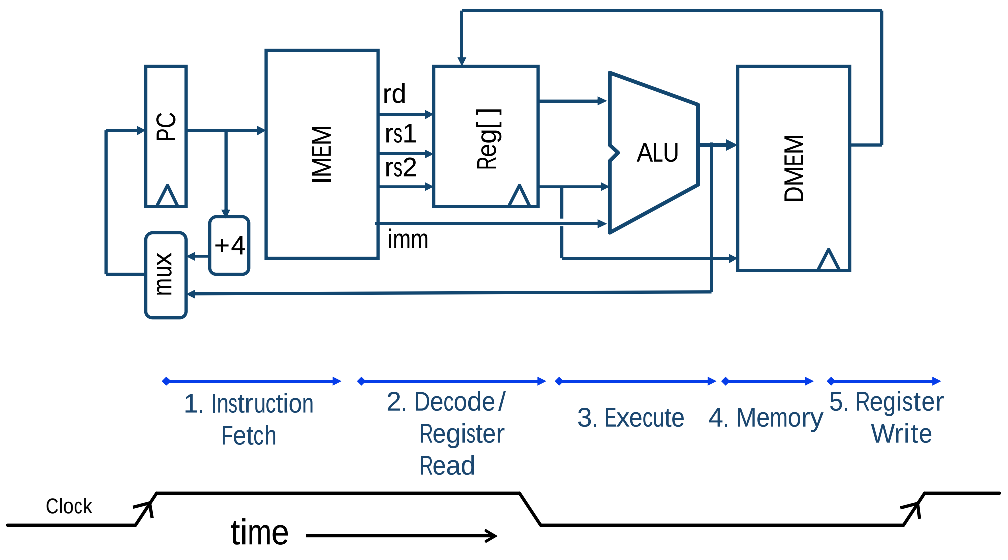
- Problem: A single, "monolithic" CL block that "executes an instruction" (performs all necessary operations beginning with fetching the instruction and completing with the register access) is be too bulky and inefficient
- Solution: Break up the process of "executing an instruction" into stages, and 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)

8

Five Stages of the Datapath

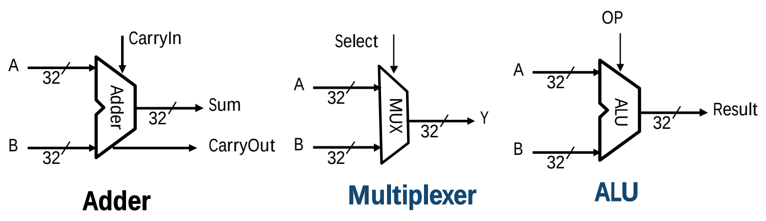
- Stage 1: *Instruction Fetch (IF)*
- Stage 2: *Instruction Decode (ID)*
- Stage 3: *Execute (EX) - ALU* (Arithmetic-Logic Unit)
- Stage 4: *Memory Access (MEM)*
- Stage 5: *Write Back to Register (WB)*

Basic Phases of Instruction Execution



Datapath Components: Combinational

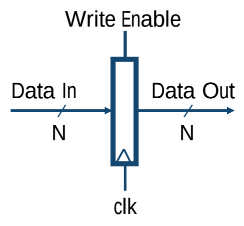
- Combinational Elements



- Storage Elements + Clocking Methodology
- Building Blocks

Datapath Elements: State and Sequencing (1/4)

- Register

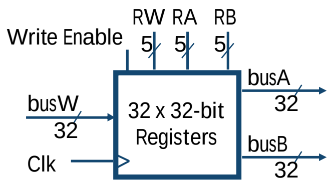


```
always @(posedge clk)
   if (wen) dataout <= datain;
endmodule
```

- Write Enable:
  - Negated (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 (2/4)

- Register file (regfile, RF) consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
  - x0 is wired to 0

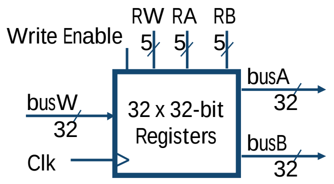


- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
- Clock input (clk)
  - Clk input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid ⇒ busA or busB valid after “access time.”

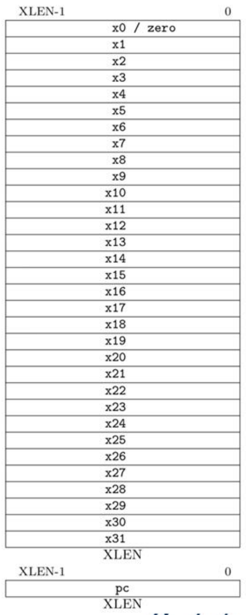
Datapath Elements: State and Sequencing (3/4)

- Reg file in Verilog

```
module rv32i_regs (
    input clk, wen,
    input [4:0] rw,
    input [4:0] ra,
    input [4:0] rb,
    input [31:0] busw,
    output [31:0] busa,
    output [31:0] busb
);
    reg [31:0] regs [0:30];
    always @(posedge clk)
        if (wen) regs[rw] <= busw;
    assign busa = (ra == 5’d0) ? 32’d0: regs[ra];
    assign busb = (rb == 5’d0) ? 32’d0: regs[rb];
endmodule
```



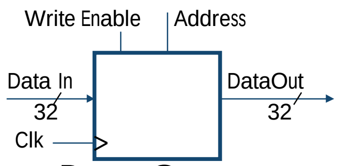
x0 is zero



- How does RV64I register file look like?

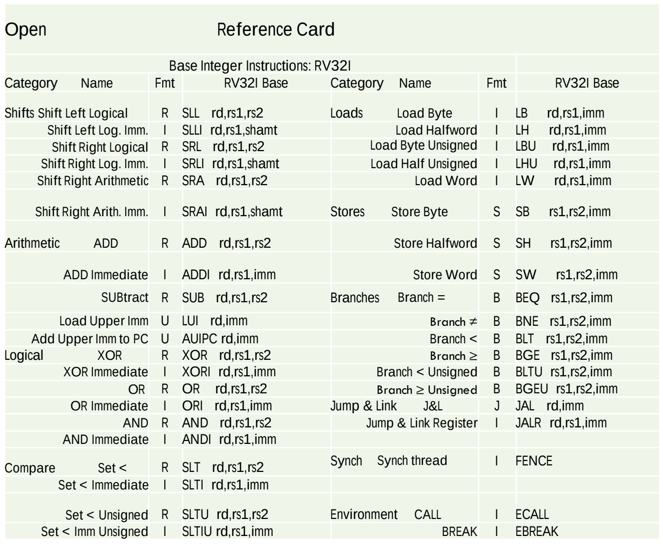
Datapath Elements: State and Sequencing (4/4)

- "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 ⇒ Data Out valid after "access time"
- Real memory later in the class



15

Review: 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

- Need datapath and control to implement these instructions

16

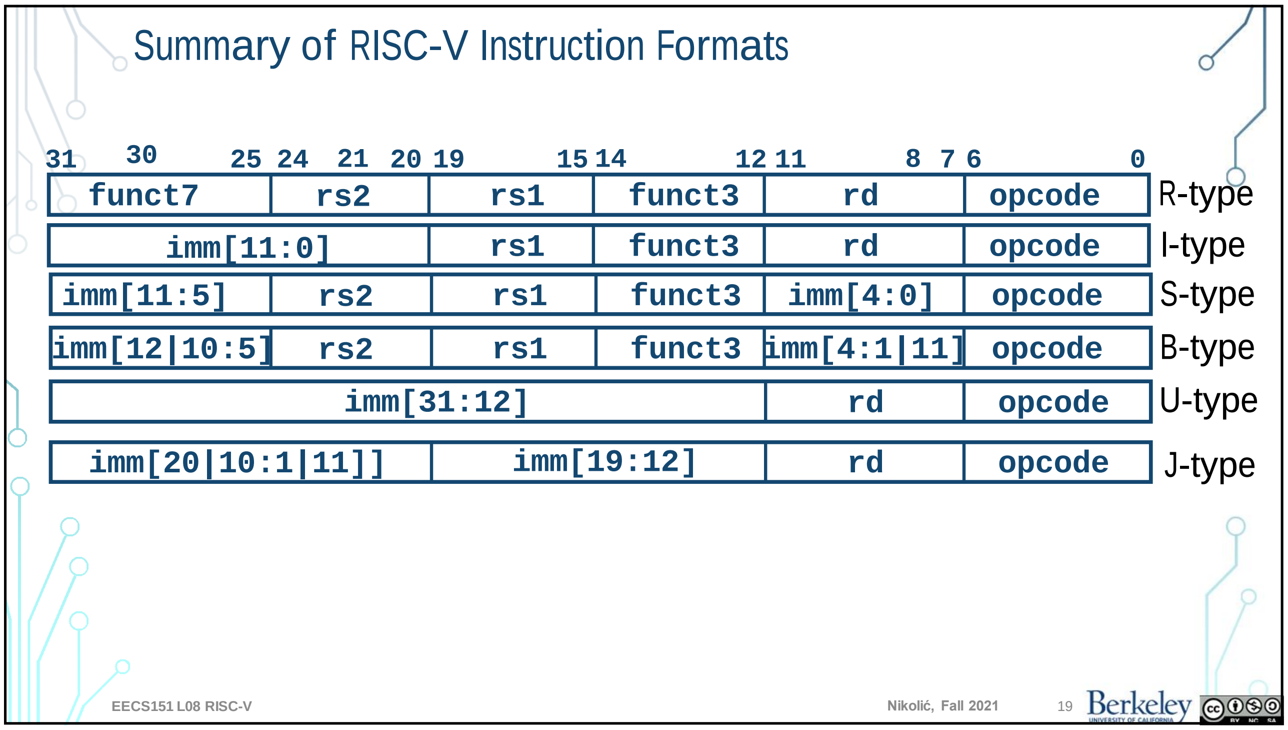
Quiz

1) We should use the main ALU to compute PC=PC+4 in order to save some gates
2) The ALU is a sequential element
3) Program counter is a register

www.yellkey.com / picture

123
FFF
FFT
FTF
FTT
TFF
TFT
TTF
TTT

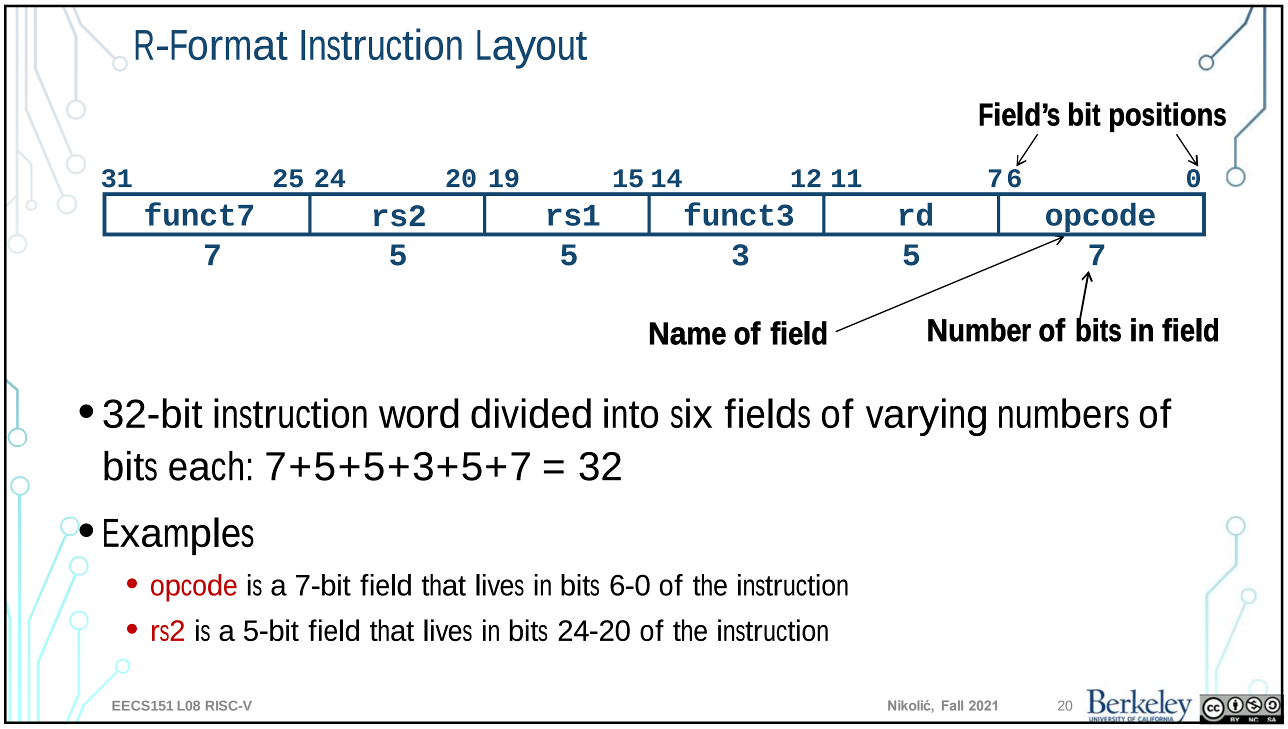
R-Format Instructions: Datapath



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

19



R-Format Instruction Layout

Field's bit positions

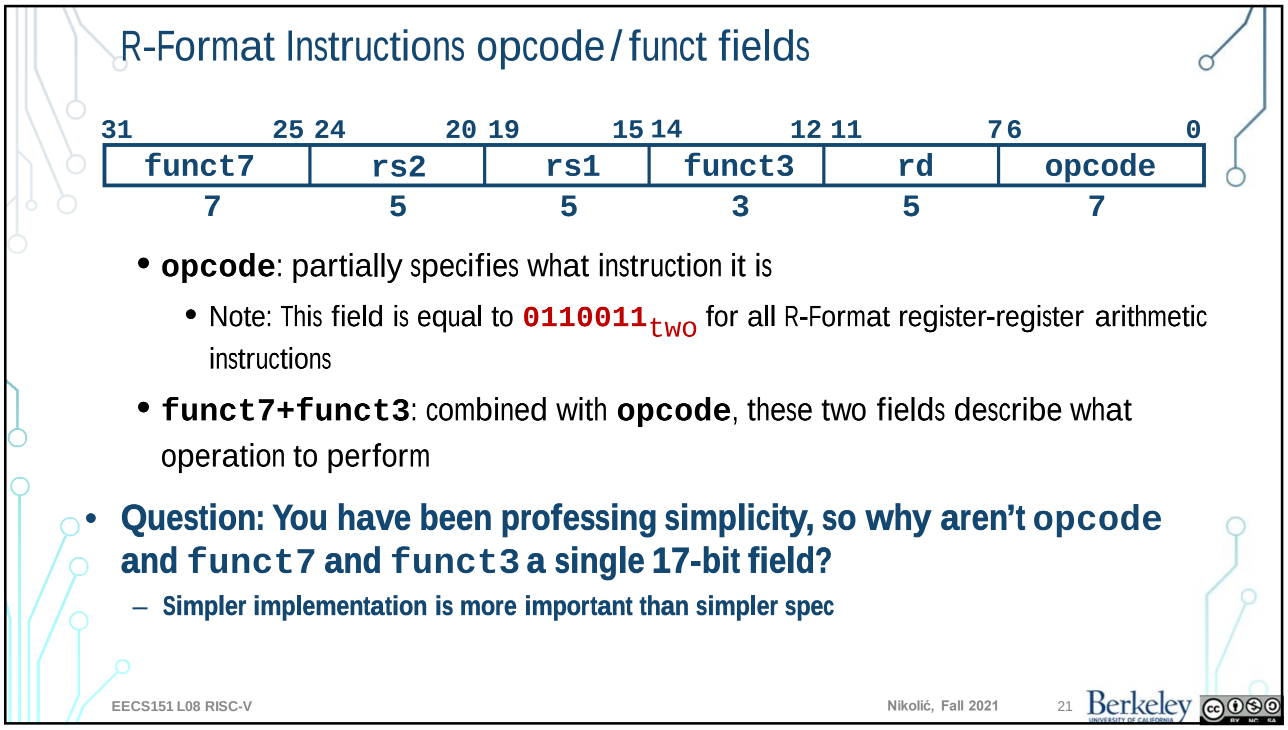
31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7

Name of field

Number of bits in field

- 32-bit instruction word divided into six fields of varying numbers of bits each: 7+5+5+3+5+7 = 32
- Examples
  - opcode is a 7-bit field that lives in bits 6-0 of the instruction
  - rs2 is a 5-bit field that lives in bits 24-20 of the instruction

20



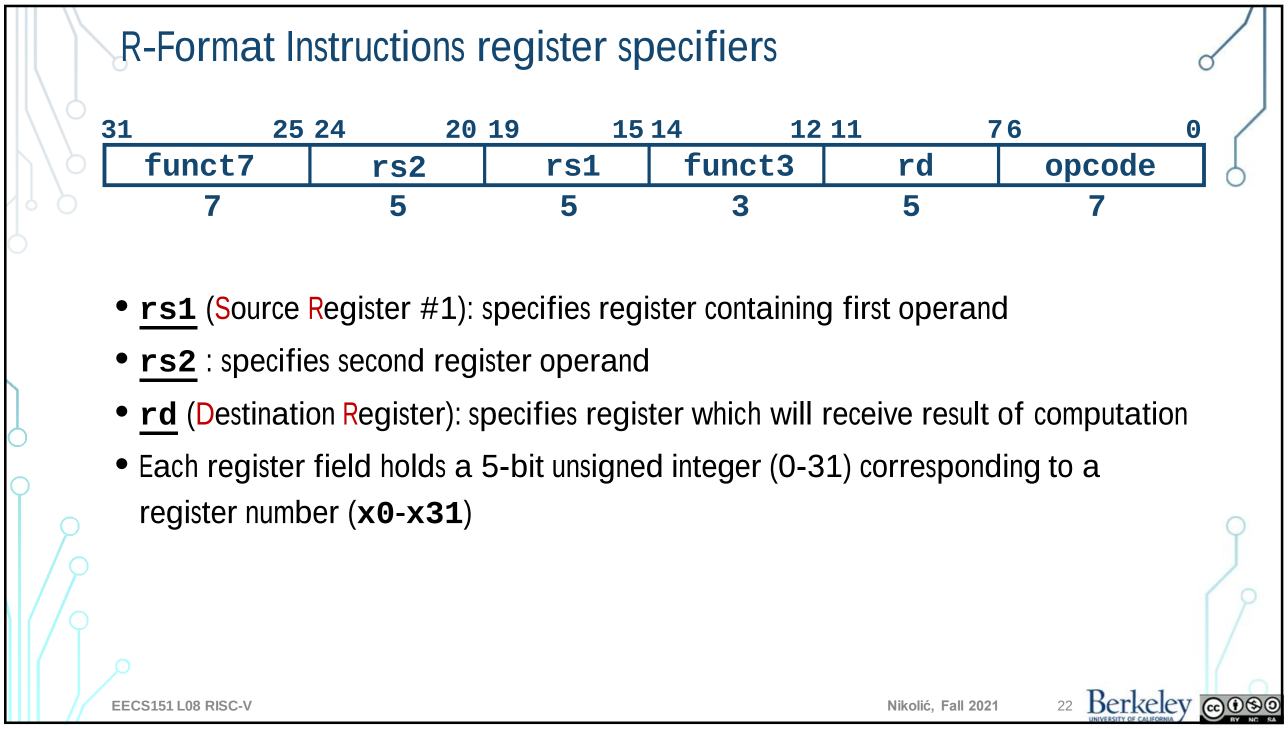
R-Format Instructions opcode / funct fields

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

- **`opcode`**: partially specifies what instruction it is
  - Note: This field is equal to **`0110011`**$_{\texttt{two}}$ for all R-Format register-register arithmetic instructions
- **`funct7+funct3`**: combined with **`opcode`**, these two fields describe what operation to perform

- **Question: You have been professing simplicity, so why aren't `opcode` and `funct7` and `funct3` a single 17-bit field?**
  – **Simpler implementation is more important than simpler spec**

21

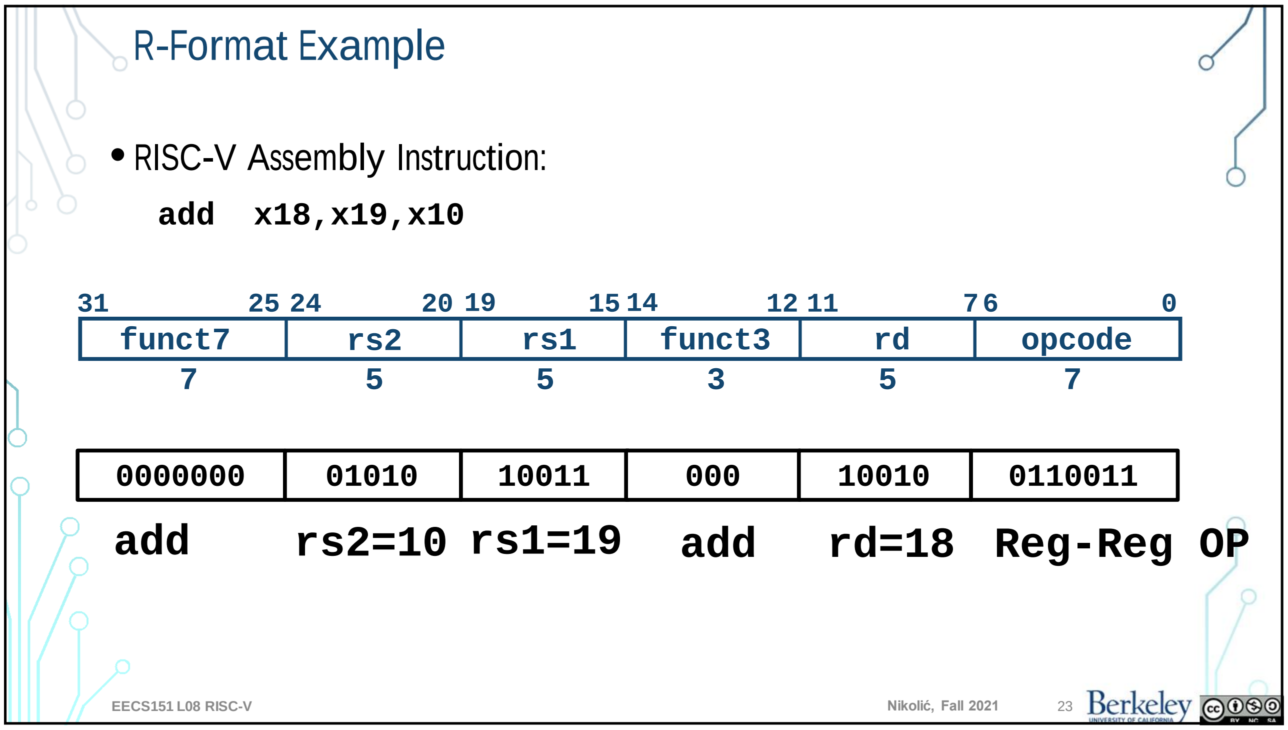


R-Format Instructions register specifiers

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

- **`rs1`** (Source Register #1): specifies register containing first operand
- **`rs2`** : specifies second register operand
- **`rd`** (Destination Register): specifies register which will receive result of computation
- Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (**`x0-x31`**)

22

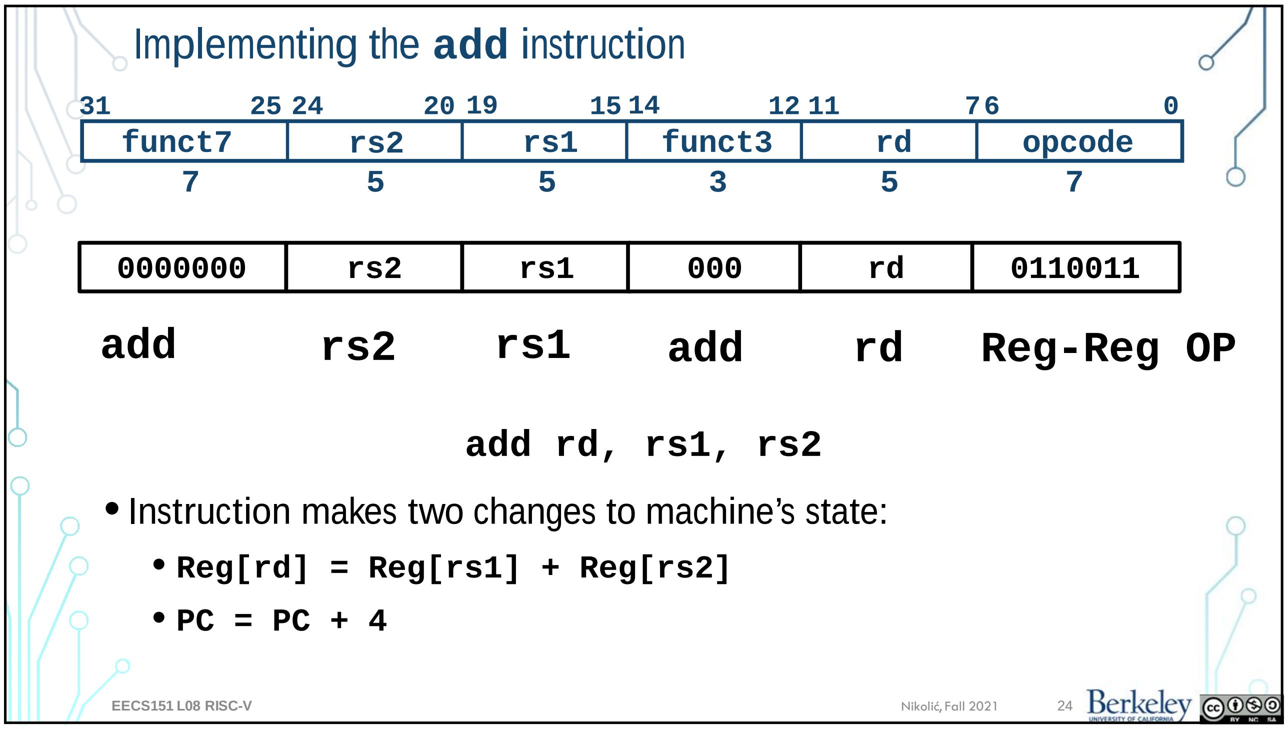


R-Format Example

- RISC-V Assembly Instruction:

  `add  x18,x19,x10`

31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	01010	10011	000	10010	0110011
add	rs2=10	rs1=19	add	rd=18	Reg-Reg OP



Implementing the **add** instruction

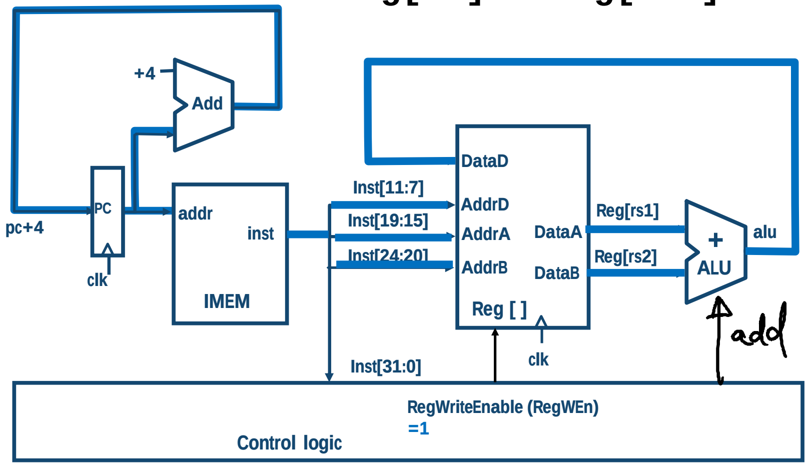
31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	rs2	rs1	000	rd	0110011
add	rs2	rs1	add	rd	Reg-Reg OP

`add rd, rs1, rs2`

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

Datapath for add

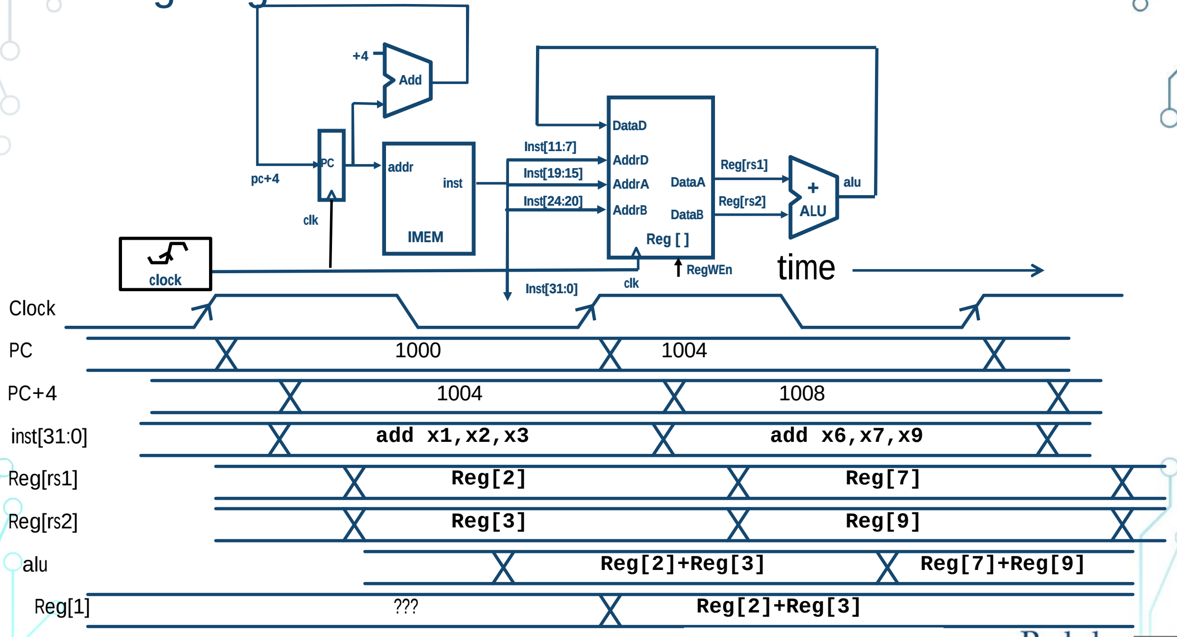
PC = PC + 4   Reg[rd] = Reg[rs1] + Reg[rs2]



31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	000	rd	opcode
add	5	5	add	5	Reg-Reg OP

25

Timing Diagram for **add**



26

Implementing the **sub** instruction

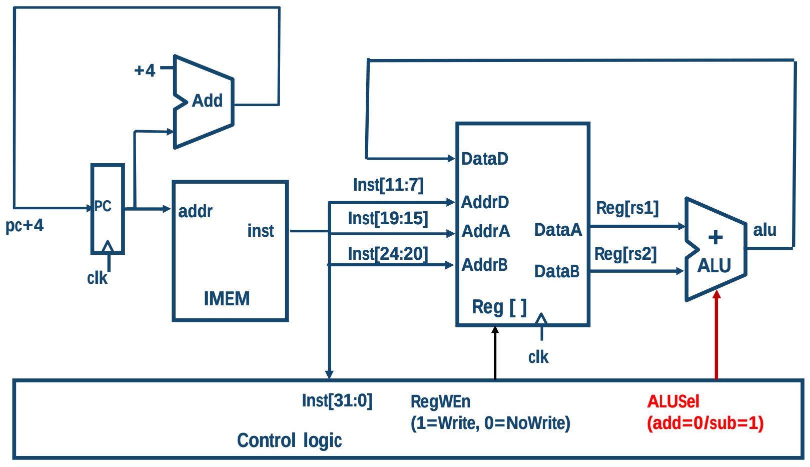
31 ... 25	24 ... 20	19 ... 15	14 ... 12	11 ... 7	6 ... 0	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub

`sub rd, rs1, rs2`

- Almost the same as add, except now have to subtract operands instead of adding them
- **`inst[30]`** selects between add and subtract

27

Datapath for `add/sub`



28

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

Administrivia

- Homework 3 is due next Monday
  - Homework 4 will be posted this week, due before midterm 1
- Lab 4 this week
- Lab 5 next week
- Midterm 1on October 7, 7-8:30pm

I-Format Instructions: Datapath

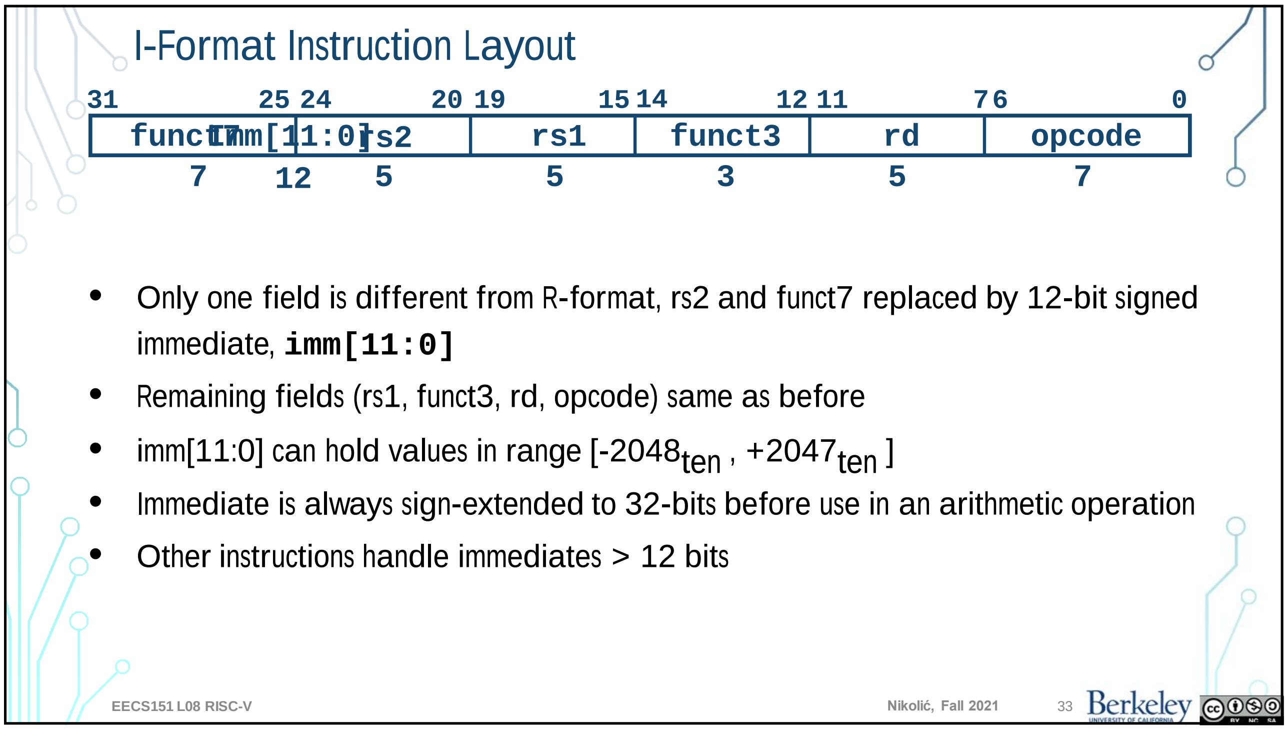
Instruction Encoding

- Instructions are encoded to simplify logic
  - **`sub`** and **`sra`** differ in Inst[30] from **`add`** and **`srl`**
- **`RV64I`** widens registers (XLEN=64)
- Additional instructions manipulate 32-bit values, identified by a suffix W
  - **`ADDW`**, **`SUBW`**
  - RV64I opcode field for 'W' instructions is **`0111011`** (**`0110011`** for RV32I)

0000000	rs2	rs1	000	rd	0110011	addw
0000000	rs2	rs1	000	rd	0111011	add

32b

64b



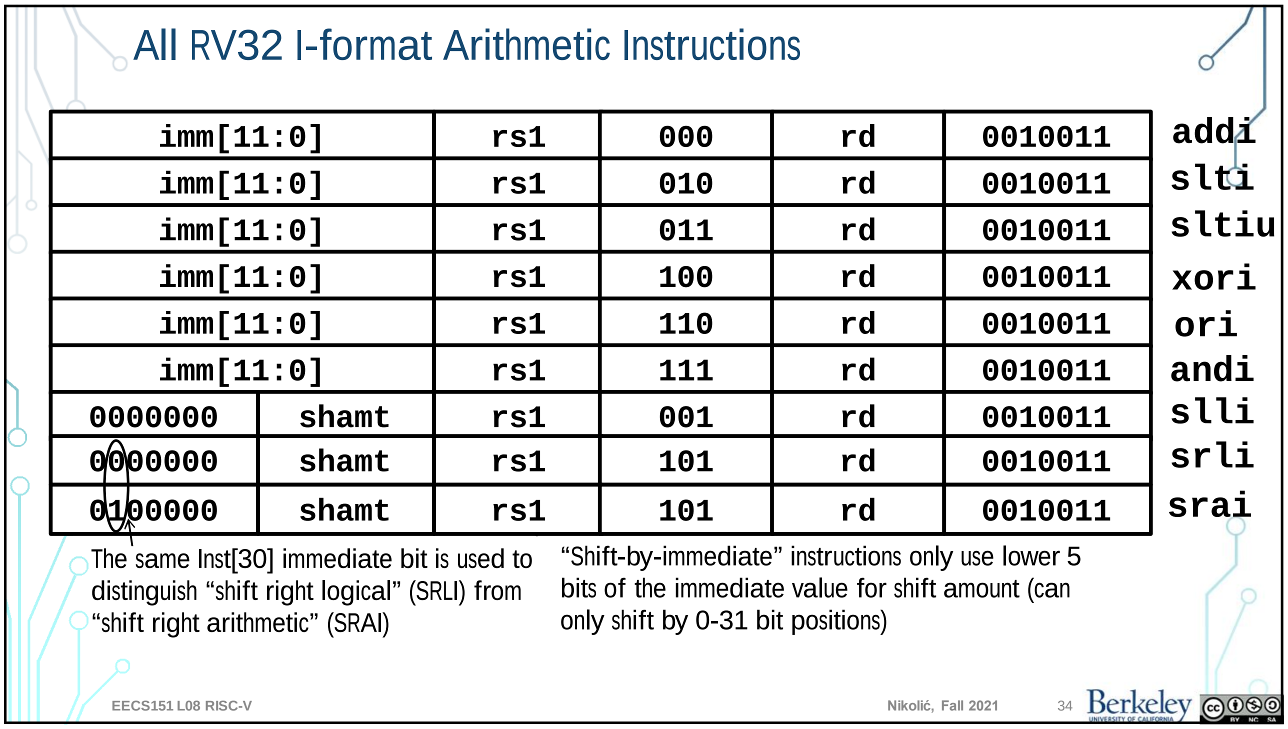
I-Format Instruction Layout

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

- Only one field is different from R-format, rs2 and funct7 replaced by 12-bit signed immediate, **imm[11:0]**
- Remaining fields (rs1, funct3, rd, opcode) same as before
- imm[11:0] can hold values in range [-2048$_{\text{ten}}$, +2047$_{\text{ten}}$]
- Immediate is always sign-extended to 32-bits before use in an arithmetic operation
- Other instructions handle immediates > 12 bits

EECS151 L08 RISC-V Nikolić, Fall 2021 33 Berkeley

33



All RV32 I-format Arithmetic Instructions

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

The same Inst[30] immediate bit is used to distinguish "shift right logical" (SRLI) from "shift right arithmetic" (SRAI)

"Shift-by-immediate" instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

EECS151 L08 RISC-V Nikolić, Fall 2021 34 Berkeley

34

Implementing I-Format - **`addi`** instruction

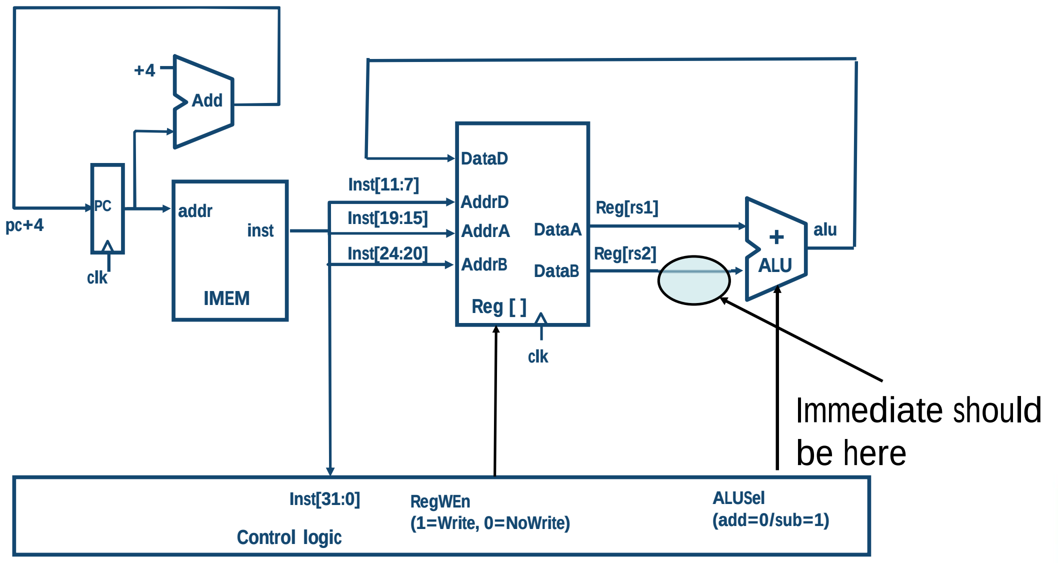
- RISC-V Assembly Instruction – add immediate:

  `addi x15,x1,-50`

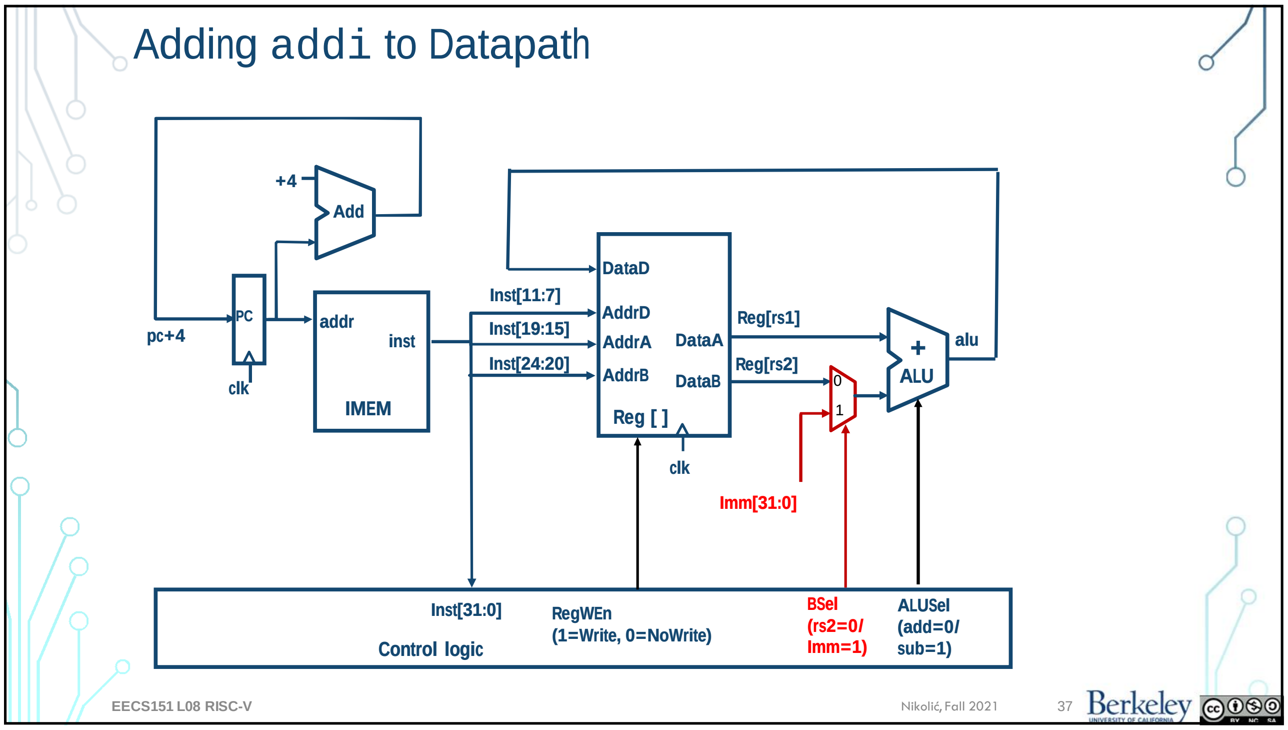
31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
111111001110	00001	000	01111	0010011
imm=-50	rs1=1	add	rd=15	OP-Imm

35

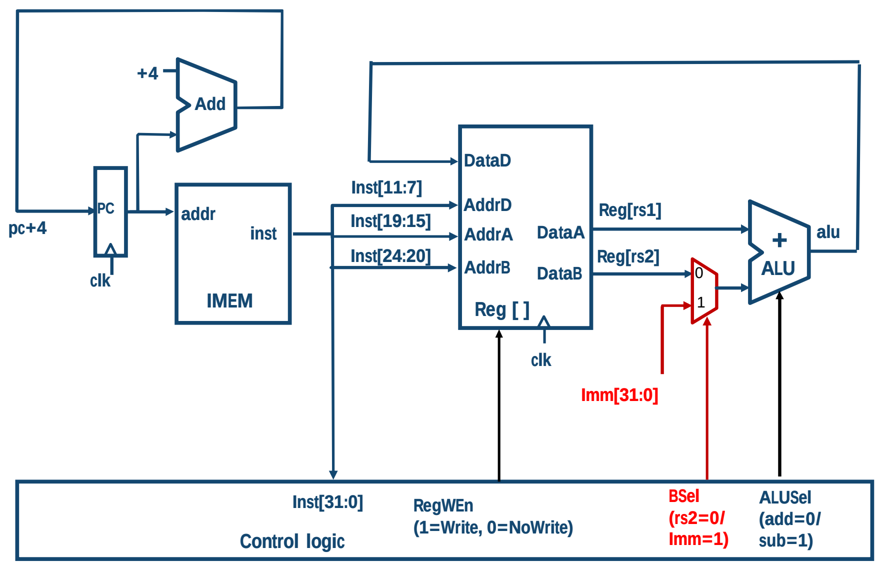
Datapath for **`add/sub`**



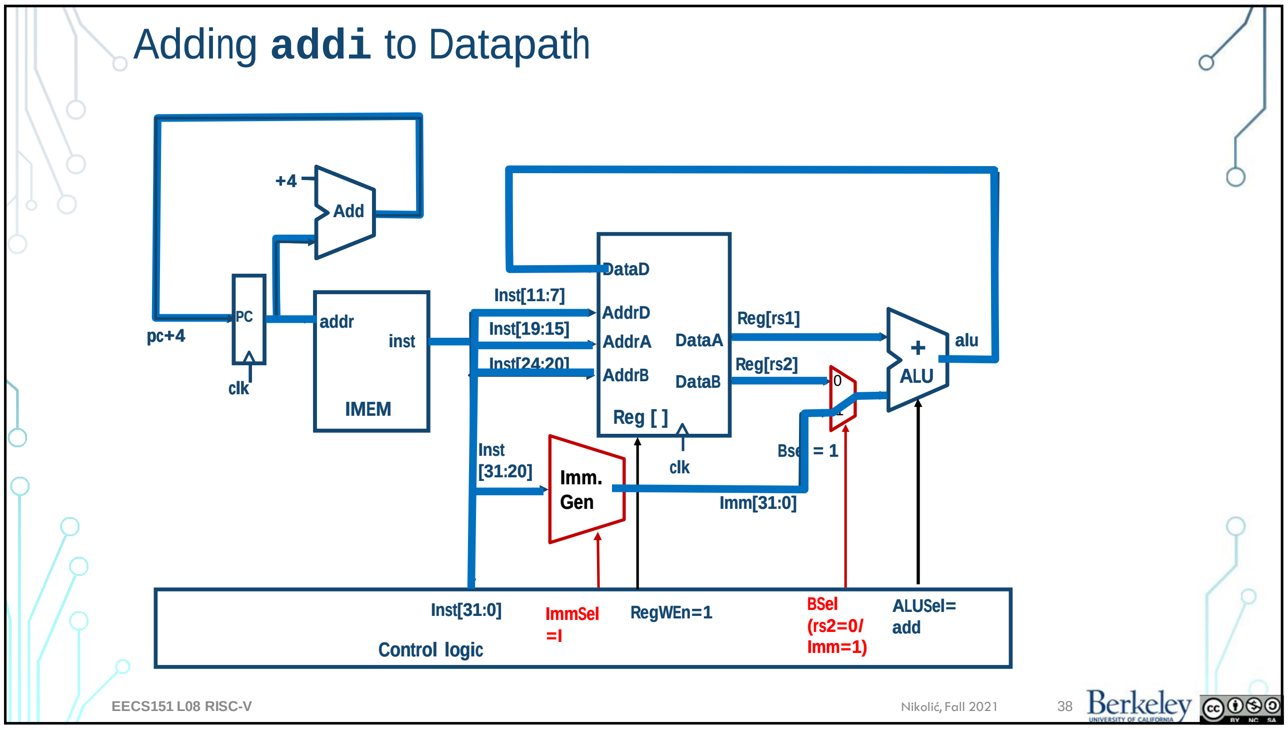
36



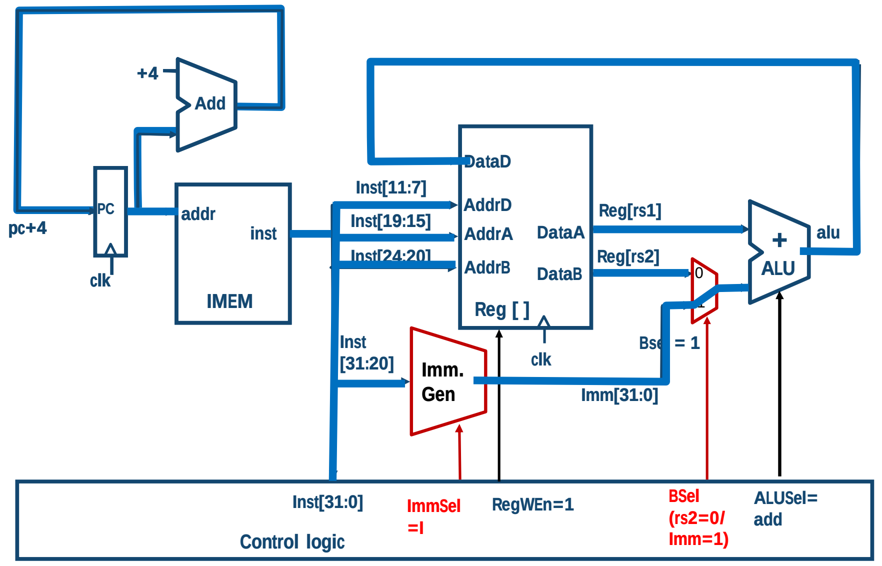
Adding addi to Datapath



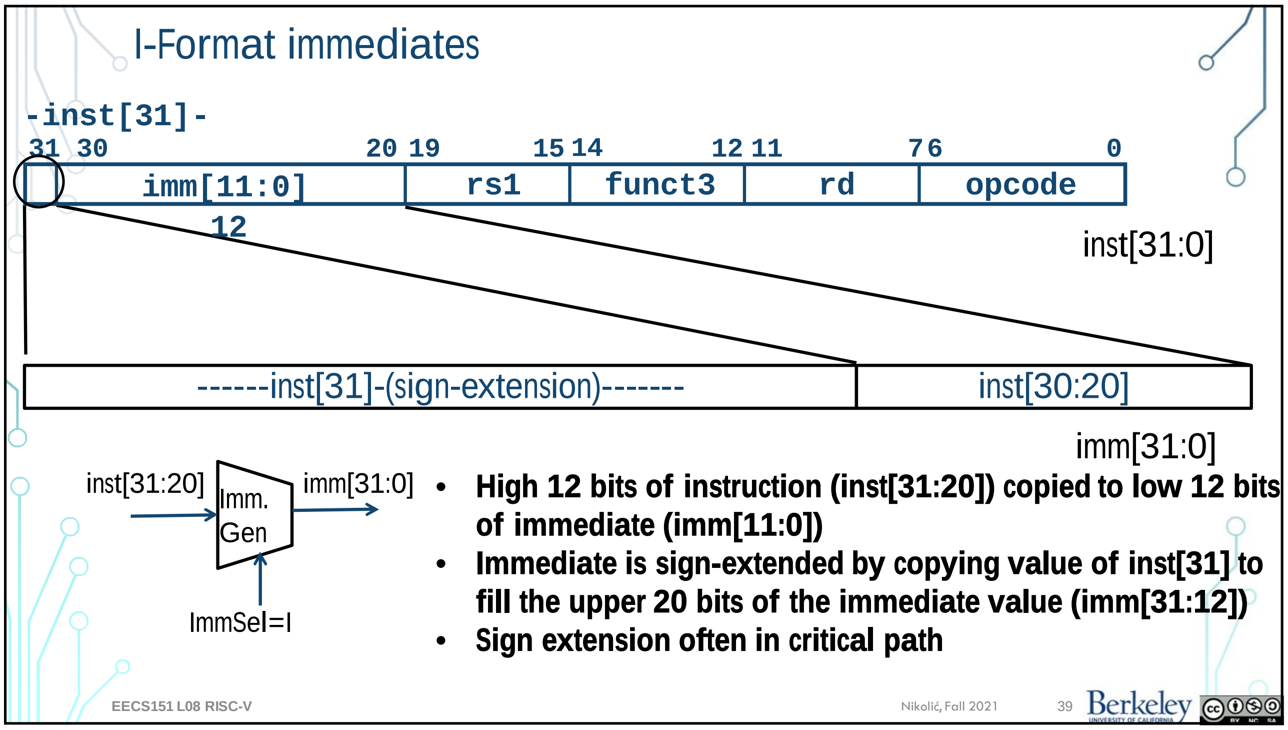
37



Adding addi to Datapath



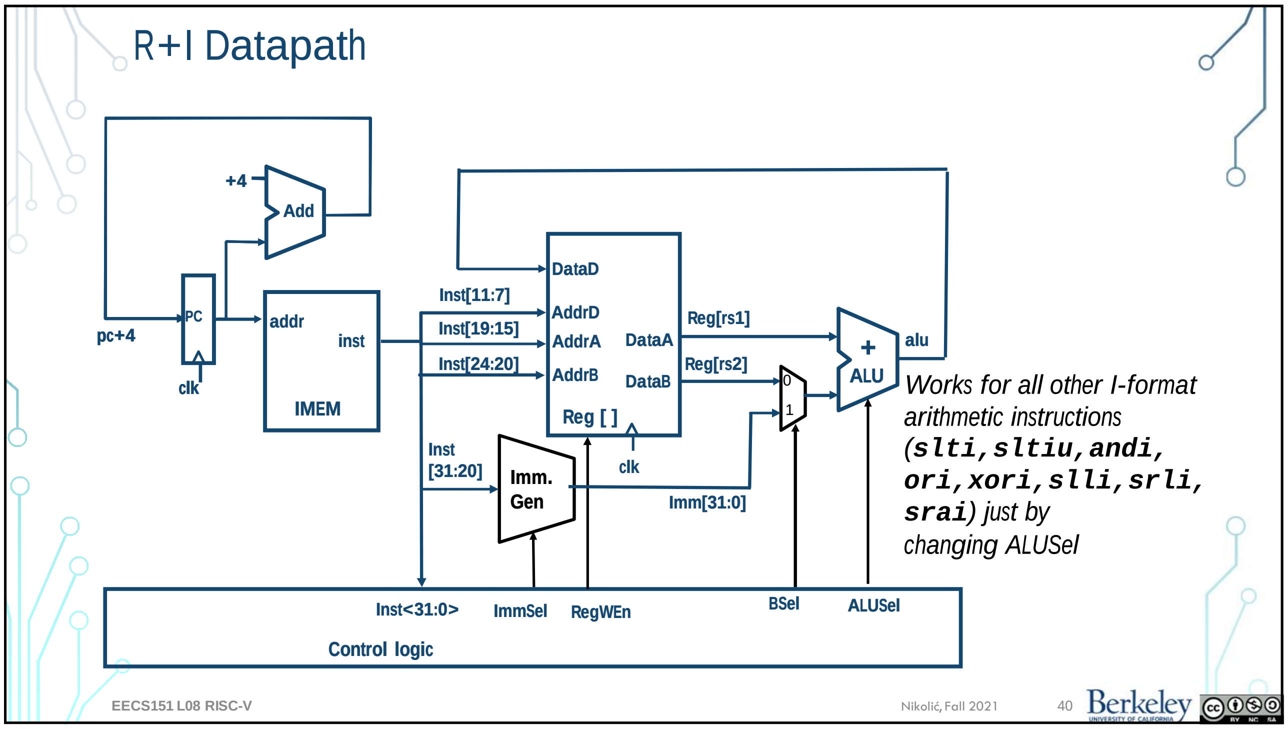
38



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])**
- **Sign extension often in critical path**

39



R+I Datapath

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

40

Add **lw** to Datapath

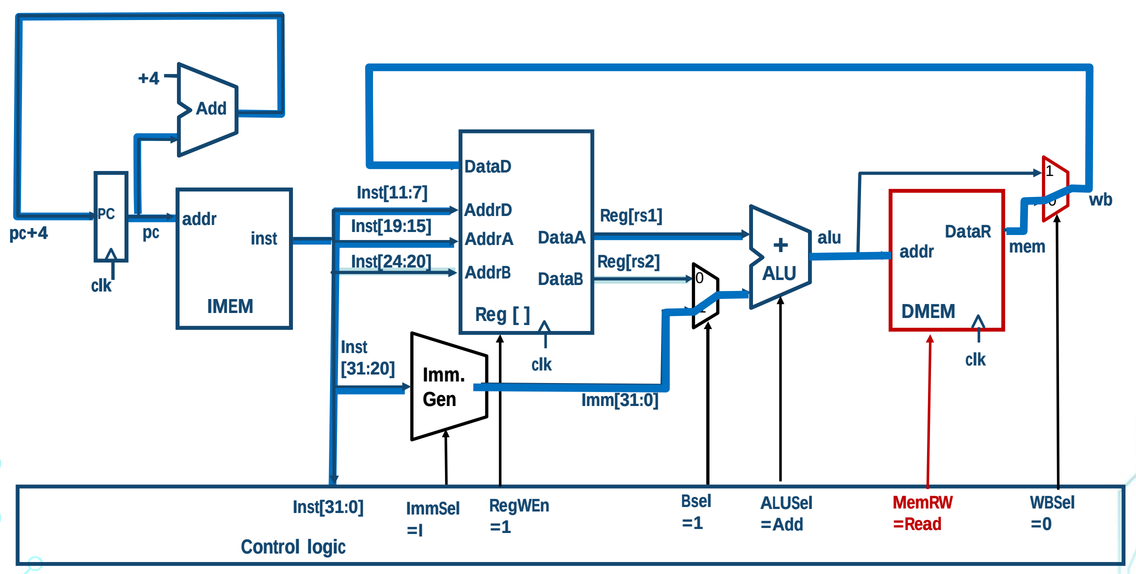
- RISC-V Assembly Instruction (I-type): **lw x14, 8(x2)**

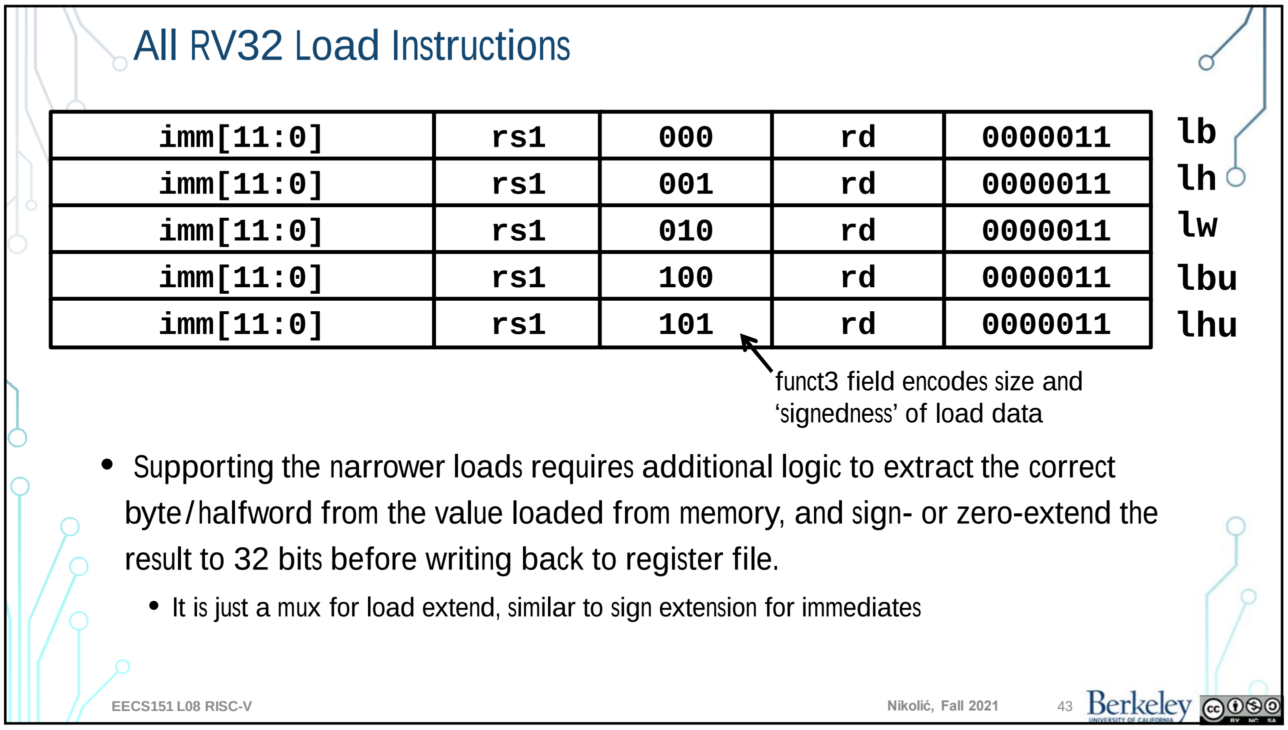
31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

31 20	19 15	14 12	11 7	6 0
000000001000	00010	010	01110	0000011
imm= +8	rs1=2	LW	rd=14	LOAD

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

Adding lw to Datapath





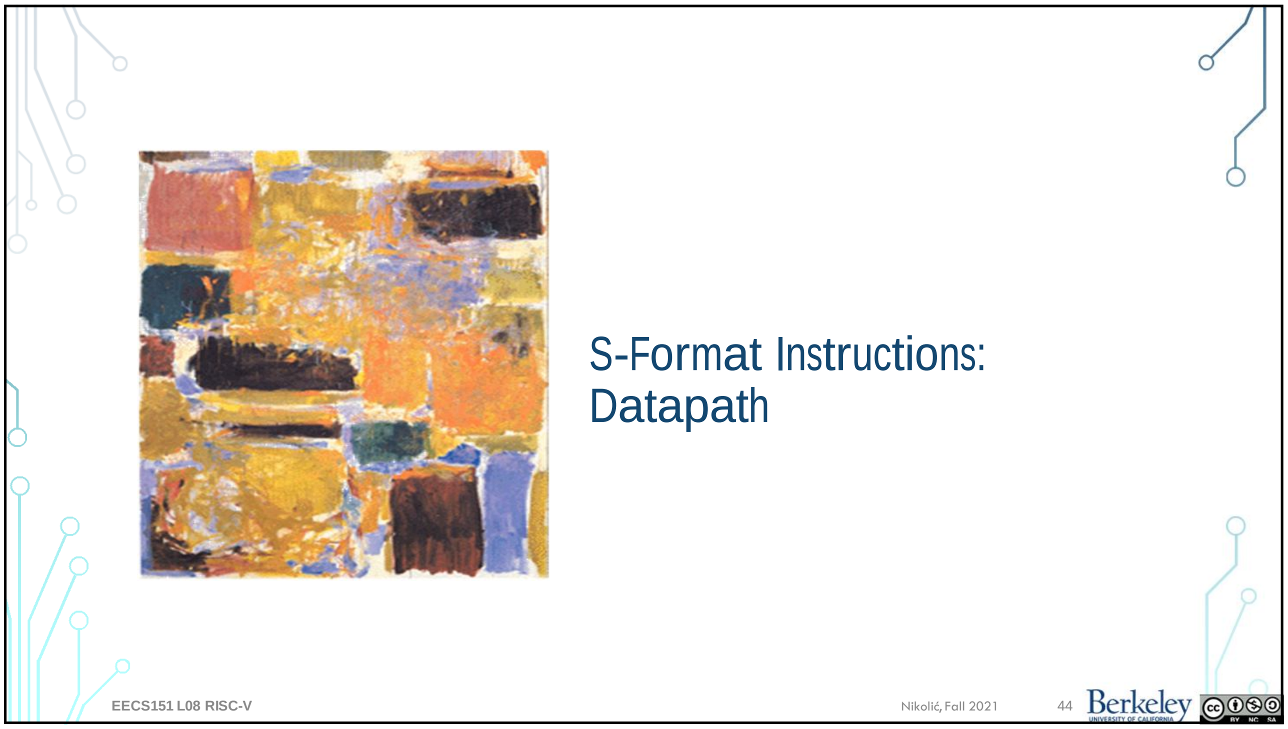
All RV32 Load Instructions

imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	001	rd	0000011	lh
imm[11:0]	rs1	010	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu

funct3 field encodes size and 'signedness' of load data

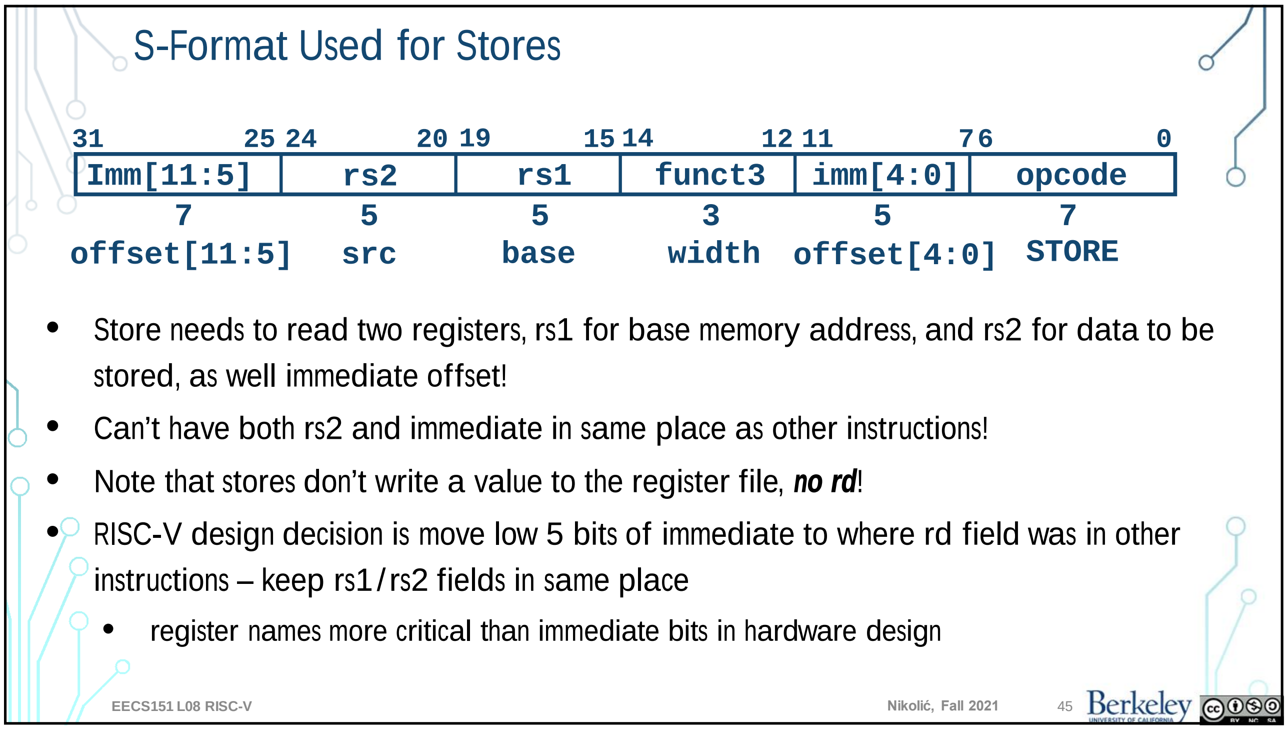
- Supporting the narrower loads requires additional logic to extract the correct byte/halfword from the value loaded from memory, and sign- or zero-extend the result to 32 bits before writing back to register file.
  - It is just a mux for load extend, similar to sign extension for immediates

43



S-Format Instructions: Datapath

44

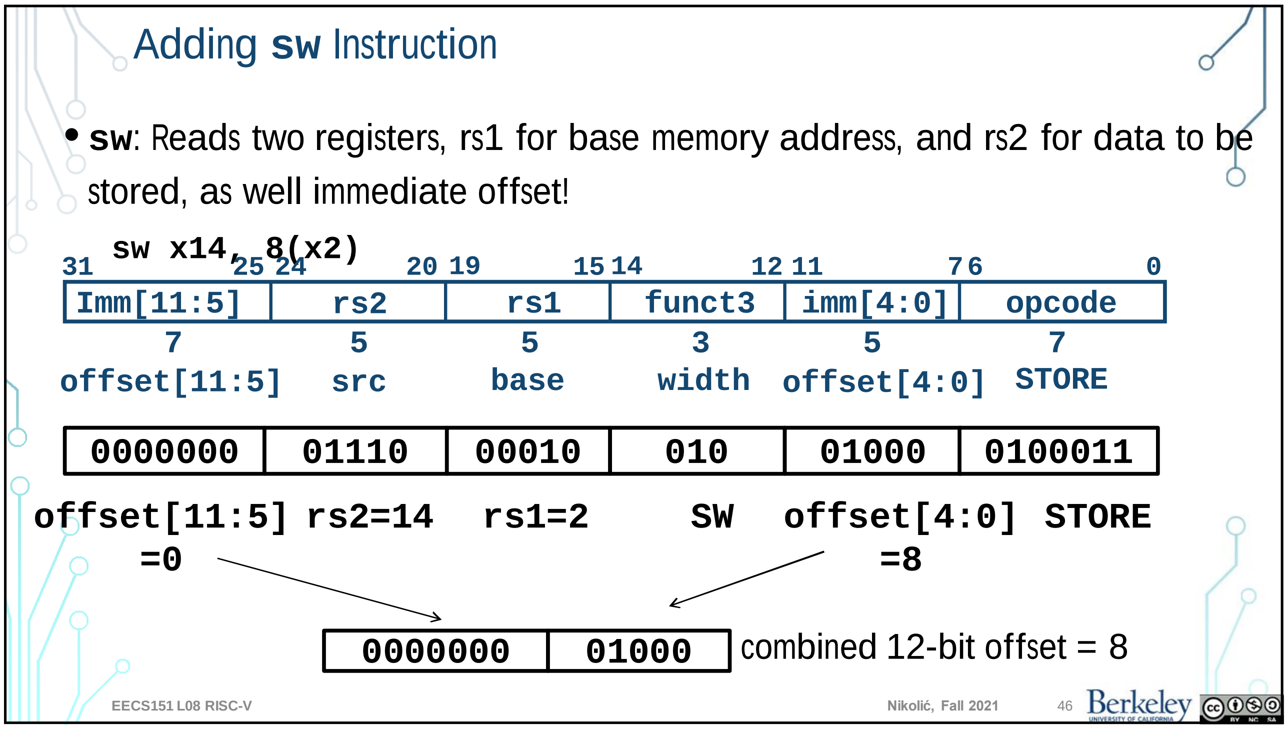


S-Format Used for Stores

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

- Store needs to read two registers, rs1 for base memory address, and rs2 for data to be stored, as well immediate offset!
- Can't have both rs2 and immediate in same place as other instructions!
- Note that stores don't write a value to the register file, ***no rd***!
- RISC-V design decision is move low 5 bits of immediate to where rd field was in other instructions – keep rs1/rs2 fields in same place
  - register names more critical than immediate bits in hardware design

45



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)
```

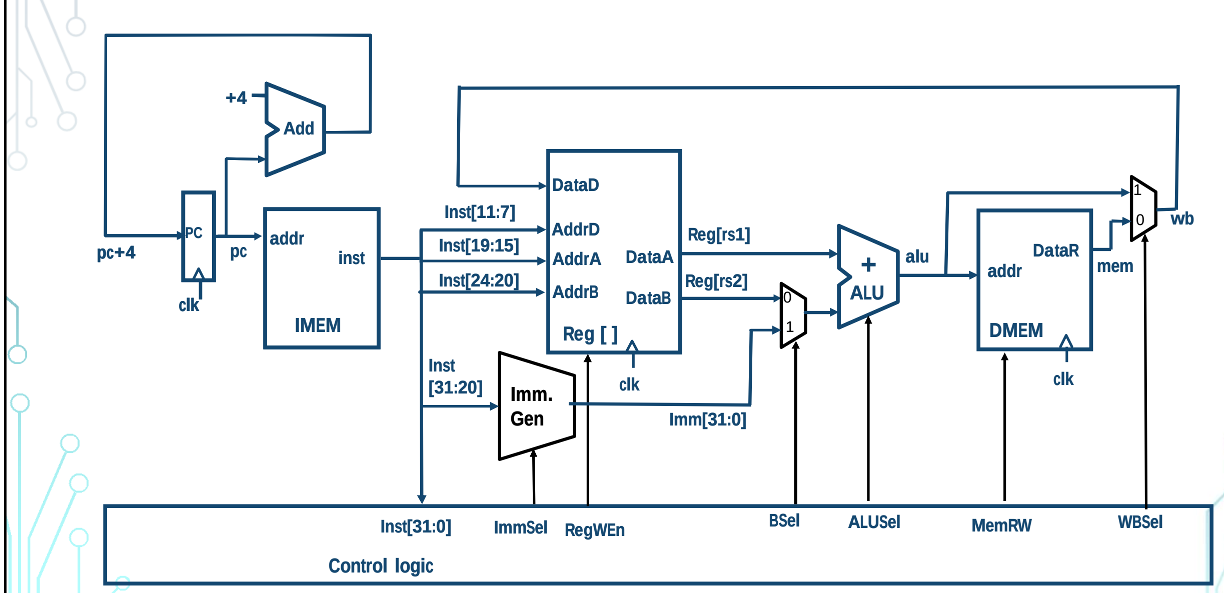
31 25	24 20	19 15	14 12	11 7	6 0
Imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE
0000000	01110	00010	010	01000	0100011
offset[11:5] =0	rs2=14	rs1=2	SW	offset[4:0] =8	STORE

0000000	01000

combined 12-bit offset = 8

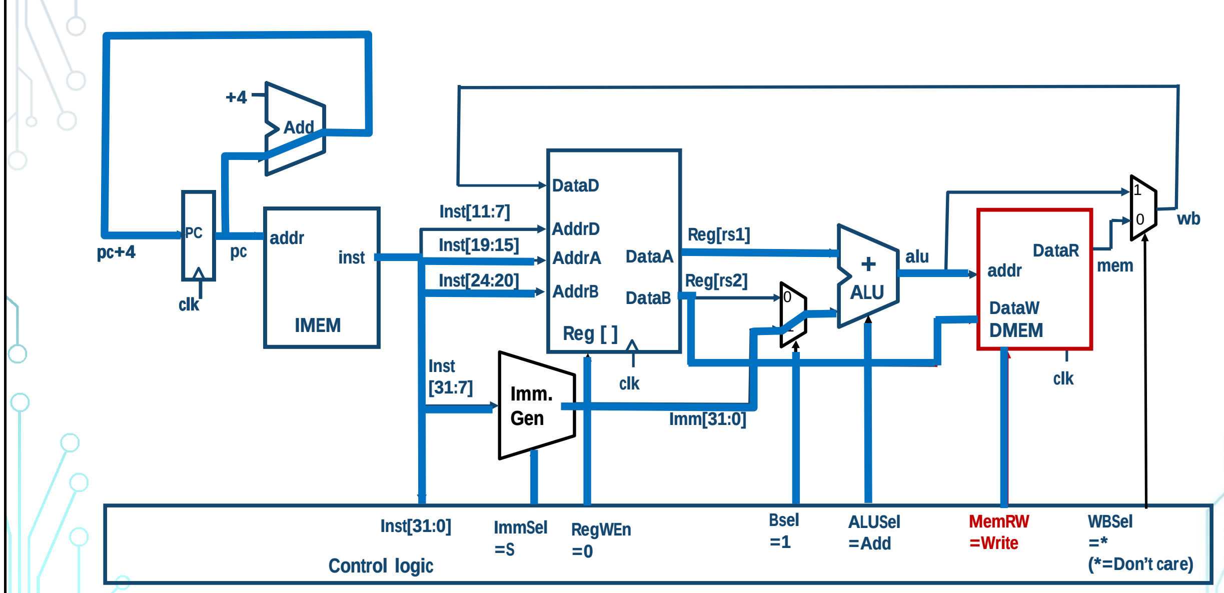
46

Datapath with lw

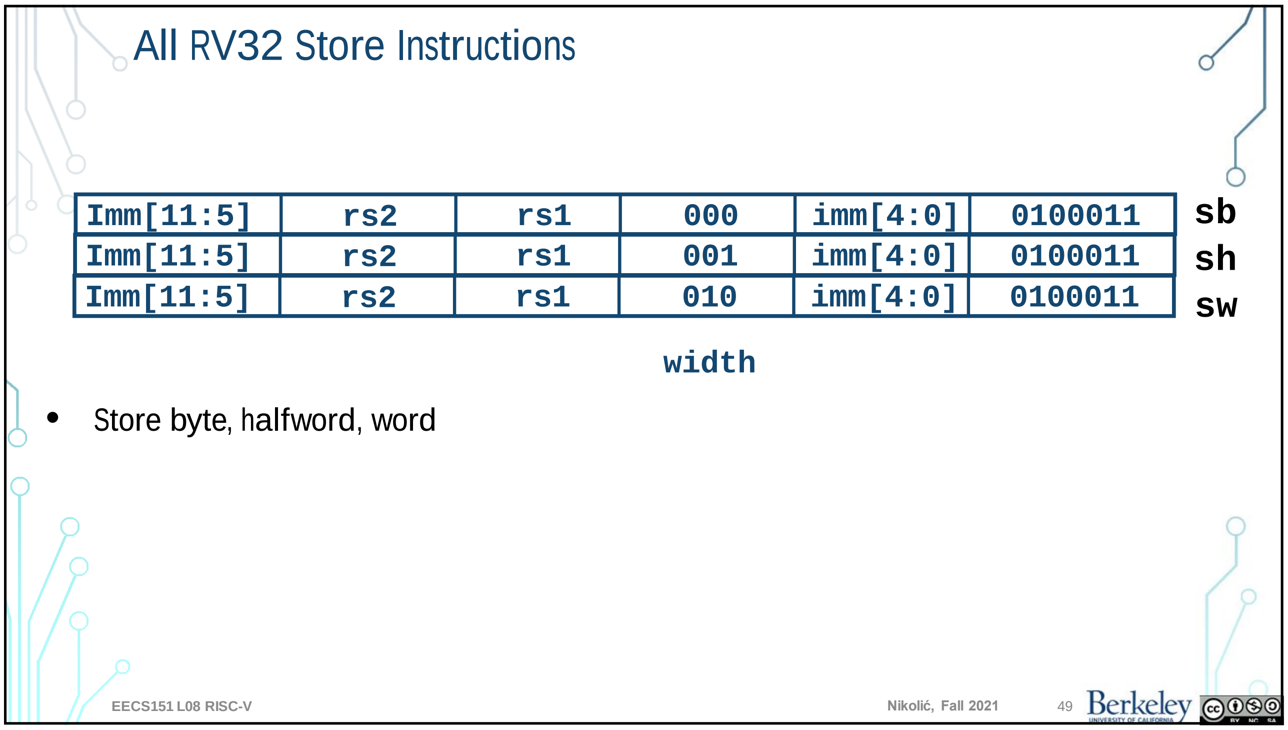


47

Adding sw to Datapath



48



All RV32 Store Instructions

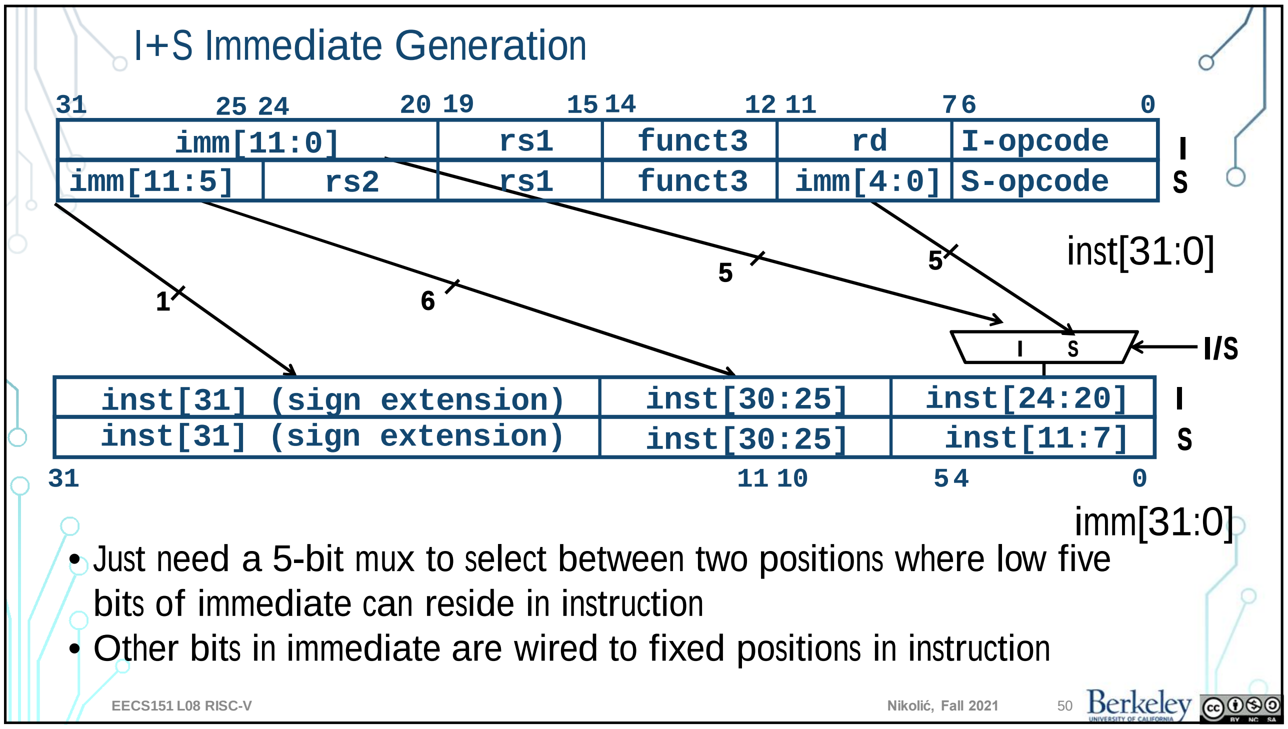
Imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	sb
Imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	sh
Imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw

width

- Store byte, halfword, word

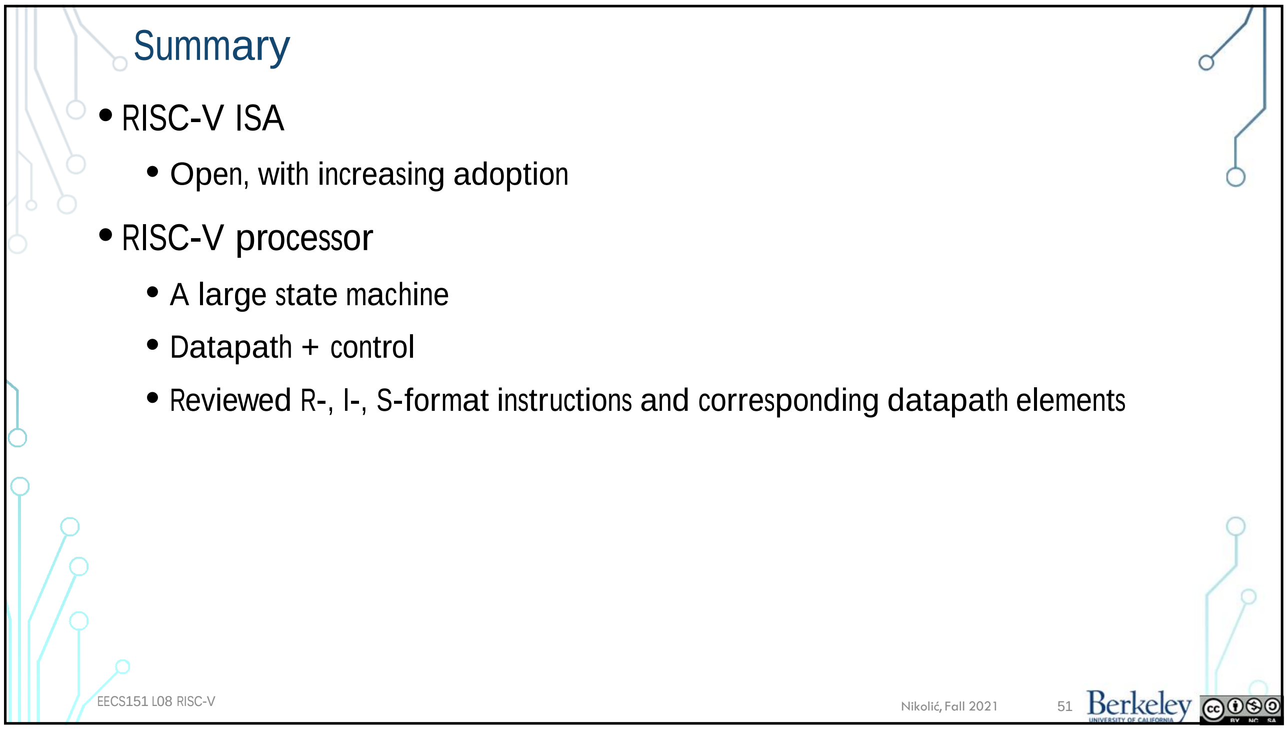
49

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

50



Summary

- 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

51