

# miniRV - a subset of RISC-V with 8 instructions

Xiaoke Su

Dec 19, 2025

# Review sISA

Instruction	Meaning	Opcode
add rd, rs1, rs2	add rs1+rs2 together, and write to rd	00
li rd, X	load the immediate value X to rd, fill high bits with zeros	10
bner0 rs2, X	if rs2 not equal to R0, PC jumps to X	11

# Now we try to know the real RISC-V

- RTFM: Read The Friendly [Manual](#) of RISC-V
- Read chapters on RV32I and RV32E
- Questions:
  - What is the bit width of PC?
  - How many GPRs in total? What's their bit width?
  - What is the difference of RISC-V R[0] and sISA R[0]?
  - What is the bit width of instruction? How many types of instructions?
  - How many bits needed to represent a GPR? Why?
  - What is the type of *add*? How to decode it?
  - What is the difference between RV32E and RV32I ?

# miniRV

- miniRV: (we defined) a subset of RISC-V, it is still an ISA
- PC is initialized as 0, each time PC + 4(not PC + 1)
  - PC is 32 bits
- Number of GPRs = Number of GPRs defined in RV32E
  - 16 registers in RV32E, each with 32 bits
- Supporting 8 insts: ***add, addi, lui, lw, lbu, sw, sb, jalr***
- Others are same as RV32I

# Registers of RV32E

- 16 GPRs, x0-x15
- x0 is a dedicated zero register
  - x0 is always zero

Register Name	Alias	Role
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary register
x6	t1	Temporary register
x7	t2	Temporary register
x8	s0/fp	Saved register / Frame pointer
x9	s1	Saved register
x10	a0	Function argument / Return value
x11	a1	Function argument / Return value
x12	a2	Function argument
x13	a3	Function argument
x14	a4	Function argument
x15	a5	Function argument

# addi, jalr

- Read manual and find descriptions of *addi* and *jalr*
- Answer the questions:
  - What do *addi* and *jalr* do?
  - What are the inst types and formats of *addi* and *jalr*?
    - Hint: Instruction Set Listings(Page 554)

# addi rd, rs1, imm

Adds a register and a sign-extended immediate value.

$$R[rd] = R[rs1] + \text{imm}$$

31		20	19	15	14	12	11	7	6	0		TYPE
	imm[11:0]		rs1		000		rd		0010011	ADDI		I
	imm[11:0]		rs1		000		rd		1100111	JALR		I

# jalr rd, imm(rs1)

Jump and Link Register: Used for function returns or indirect jumps.

$$R[rd] = PC + 4; PC = (R[rs1]+imm) \& \sim 1$$

# Test your design (*addi, jalr*)

00000000 <\_start>:

**0: 01400513** addi a0,zero,20

**4: 010000e7** jalr ra,16(zero) # 10 <fun>

**8: 00c000e7** jalr ra,12(zero) # c <halt>

0000000c <halt>:

**c: 00c00067** jalr zero,12(zero) # c <halt>

00000010 <fun>:

**10: 00a50513** addi a0,a0,10

**14: 00008067** jalr zero,0(ra)

# add rd, rs1, rs2

Adds two registers and stores the result in the destination register.

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

31	25	24	20	19	15	14	12	11	7	6	0		TYPE
	0000000		rs2		rs1		000		rd		0110011	ADD	R
	imm[31:12]							rd			0110111	LUI	U

# lui rd, imm

Load Upper Immediate: Places a 20-bit immediate in the top 20 bits of the register, clearing the bottom 12.

$$R[rd] = imm \ll 12$$

# Test your design (*add, lui*)

- Write some instructions in ROM and test

# lw rd, imm(rs1)

Load Word: Loads a 32-bit value from memory into a register.

$$\text{addr} = R[\text{rs1}] + \text{imm}; R[\text{rd}] = M[\text{addr}][31:0]$$

31	25	24	20	19	15	14	12	11	7	6	0		TYPE
imm[11:0]			rs1		010		rd		0000011		LW	I	
imm[11:5]			rs2		rs1		010		imm[4:0]		0100011	SW	S

# sw rs2, imm(rs1)

Store Word: Stores a 32-bit value from a register into memory.

$$\text{addr} = R[\text{rs1}] + \text{imm}; M[\text{addr}] = R[\text{rs2}][31:0]$$

# Test your design (*lw*, *sw*)

- Write some instructions in ROM and test

# **lbu rd, imm(rs1)**

**Load Byte Unsigned:** Loads 8 bits from memory and zero-extends them to 32 bits.

$$\text{addr} = R[\text{rs1}] + \text{imm}; R[\text{rd}] = \{24'b0, M[\text{addr}][7:0]\}$$

31		20	19	15	14	12	11	7	6	0		TYPE
	imm[11:0]		rs1		100		rd		0000011	LBU		I

**Hint:**

- First set 4-byte data 0x12345678 in RAM
- Load the data by ***lw*** (assume raddr is a)and make sure the rdata is 0x12345678
- Then load the data by 4 ***lbu*** instructions from addr a, a+1, a+2, a+3
- The result should be 0x78(addr a), 0x56, 0x34, 0x12(addr a+3)

# sb rs2, imm(rs1)

Store Byte: Stores the lowest 8 bits of a register into memory.

$$\text{addr} = R[\text{rs1}] + \text{imm}; M[\text{addr}][7:0] = R[\text{rs2}][7:0]$$

31	25	24	20	19	15	14	12	11	7	6	0		TYPE
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB	S						

Hint:

- First set 4-byte data 0x12345678 in RAM
- Load the data by ***lw*** (assume raddr is a) and make sure the rdata is 0x12345678
- Then store the data 0x90(a+3), 0xab, 0xcd, 0xef(a) by 4 ***sb*** instructions into addr a+3, a+2, a+1, a+0 respectively.
  - Before performing the stores, an ***addi*** instruction can be used with the zero register to write an immediate value into the destination register (similar to the effect of the ***li*** instruction in the sISA)
- Finally, use an ***lw*** instruction to read back the updated data, the expected result is 0x90abcdef.

# Practice

- Build miniRV CPU
- We RECOMMEND you to do step by step:
  1. Build miniRV CPU in Logisim-evolution first
  2. Implement miniRV CPU using Verilog
  3. Implement miniRV CPU using C and do difftesting
- For some students who are very confident in Verilog, you may skip the first logisim step

# Practice Hint(must read)

- Download **GPR.circ** to help you accelerate your register file design
- You can use test cases(**sum.hex, mem.hex**) to test your 8-inst miniRV CPU
  - Use ROM component to construct your inst ROM, edit contents of ROM and open sum.hex (in Logisim)
  - Remember to use **DPI-C**(Google it) to read program (in Verilog)
  - Transfer to .bin: `tail -n +2 sum.hex | sed -e 's/.*: //' -e 's/ \n/g' | sed -e 's/\(..\)\(..\)\(..\)\(..\)/\4 \3 \2 \1/' | xxd -r -p > sum.bin`
- All files have been uploaded to:
  - <https://trello.com/b/RtfpCYmh/chip-design-bootcamp-2025>

# Optional – Adding Graphics Display Functionality to your miniRV CPU (in logisim)

- Add the **RGB Video** component to the processor's data path, then load and run the **vga.hex** program.
- The expected running time of this program is 628,000 cycles, and you may need to wait 1 to 2 minutes(Logisim version 4.0.0). In Logisim 3.8.0, the vga.hex program required 1 to 2 hours to run, so we recommend using the new version of Logisim.
- If your implementation is correct, you will see the "One Student One Chip" logo displayed in the RGB Video component when the program finishes running.

# Q&A

Reference: <https://ysyx.oscc.cc/docs/en/2407> (Stage F Chapter 6 &  
Stage E Chapter 4)