

# 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.

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

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

# jlr 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.

$$R[rd] = \text{imm} \ll 12$$

# 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.

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

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>

# Additional Info(miniRV.circ)

Set ROM Attributes as:

- Address Bit Width - Configure based on the subsequent program size and your understanding
- Data Bit Width: 32
- Line size: Single
- Allow misaligned?: No

**sum.hex and mem.hex take about 6000 cycles, and PC should stop at <halt> and a0(R10) = 0 at that point (check where is <halt> in sum.txt/mem.txt)**

RAM Attributes

- Address Bit Width - Configure based on the subsequent program size and your understanding
- Data Bit Width: 32
- Enables: Use byte enables
- Ram type: non volatile
- Use clear pin: No
- Trigger: Rising Edge
- Asynchronous read: Yes
- Read write control: Use byte enables
- Data bus implementation: Separate data bus for read and write

# Additional Info(miniRV.circ)

- To upload test case file(for example, sum.hex), right-click the ROM component, select Load Image, then choose the sum.hex file. This process will **load the contents described in sum.hex sequentially into the ROM.**
- In fact, the .hex file contains not only the program's instruction sequence but also the data processed by the program. The program accesses this data via memory access instructions, **so we also need to load the .hex file into RAM.**
- Consequently, the ROM contains the data processed by the program, while the RAM holds the instruction sequence. However, the program's inherent functionality ensures it will not access the instruction sequence in RAM via memory access instructions. It also prevents erroneously retrieving data from ROM and executing it as instructions. Thus, this dual loading does not compromise the program's correct execution.

# 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.
- For the processor, components like the RGB Video are called peripheral devices, or "peripherals" for short. In fact, how to access peripherals is part of the ISA specification. Specifically, in RISC-V, accessing peripherals is done through "Memory-mapped I/O". The essence of this method is to determine whether the processor accesses memory or peripherals based on the range of memory access addresses.

- Specifically for the RGB Video, according to the above configuration, a pixel data occupies 3 bytes. However, for ease of processing, we can treat it as 4 bytes. In this way, the size of the pixel data stored in the entire screen is  $256 \times 256 \times 4B = 256KB$ .
- We stipulate that the addresses corresponding to each pixel data on the screen are contiguous; therefore, we need to **allocate a contiguous address range for the pixel data of the entire screen, such as [0x20000000, 0x20040000).**
- When the target address of a memory access instruction falls within this range, the corresponding instruction will access the RGB Video instead of the RAM.

- To implement the function of "determining the access target based on the memory access address range", you **need to add an address decoder module** to the processor's data path. This module takes the memory access address as input and outputs two control signals: **isVGA** and **isMem**. Among them, isVGA is active when the memory access address falls within the above range; otherwise, isMem is active. Then, these two control signals can be used to control the access behavior of the corresponding components.
- For the RGB Video, its write operation needs to be controlled by the isVGA signal. That is, the RGB Video can only be written when the current instruction is a store instruction and the address falls within the above range. For simplicity, we stipulate that the program can only write pixels to the screen through the sw instruction, so the data to be written by sw can be directly connected to the RGB Video. Finally, we also need to consider the connection of the X and Y coordinates.

- In fact, because the addresses corresponding to pixel data are contiguous, given an address within the RGB Video range, it is easy for us to get the X and Y coordinates of the pixel corresponding to that address.
- For example, the address 0x20000000 corresponds to the pixel in row 0, column 0, while the address 0x20000408 corresponds to the pixel in row 1, column 2. For the RAM, its write operation also needs to be controlled by the isMem signal to avoid erroneously writing to the RAM when accessing the RGB Video. The above is just a brief introduction to the principle of peripheral access.

# Optional Practice

- Add the **RGB Video** component to the processor's data path, then load and run the **vga.hex** program.
- Data range for RGB Video: [0x20000000, 0x20040000)
- 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.

# Additional Info of optional vga.hex task

## RGB Video Attributes

- Cursor: No Cursor
- Reset Behavior Asynchronous
- Color Model: 888 RGB (24 bit)
- Width: 256
- Height: 256

In the end, RGB Video should show the logo of “One Student One chip”:



# Q&A

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