Build sCPU in Logisim

Xiaoke Su

Dec 17, 2025

Review the program

- 0: li r0, 10 # 10 in decimal
- 1: li r1, 0
- 2: li r2, 0
- 3: li r3, 1
- 4: add r1, r1, r3
- 5: add r2, r2, r1
- 6: bner0 r1, 4
- 7: bner0 r3, 7

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

Review sISA

```
7  6 5  4 3   2 1   0
+----+----+-----+-----+
| 00 | rd | rs1 | rs2 | R[rd]=R[rs1]+R[rs2]
+----+----+-----+-----+
| 10 | rd |     imm   | R[rd]=imm
+----+----+-----+-----+
| 11 |   addr    | rs2 | if (R[0]!=R[rs2]) PC=addr
+----+----------+-----+
```

li rd, imm

- load the immediate value imm to R[rd], fill high bits with zeros
- R[rd] = imm

- 8 bits
  - opcode = 00 = inst[7:6]
  - rd = inst[5:4]
  - imm = inst[3:0]

What do we need for *li*?

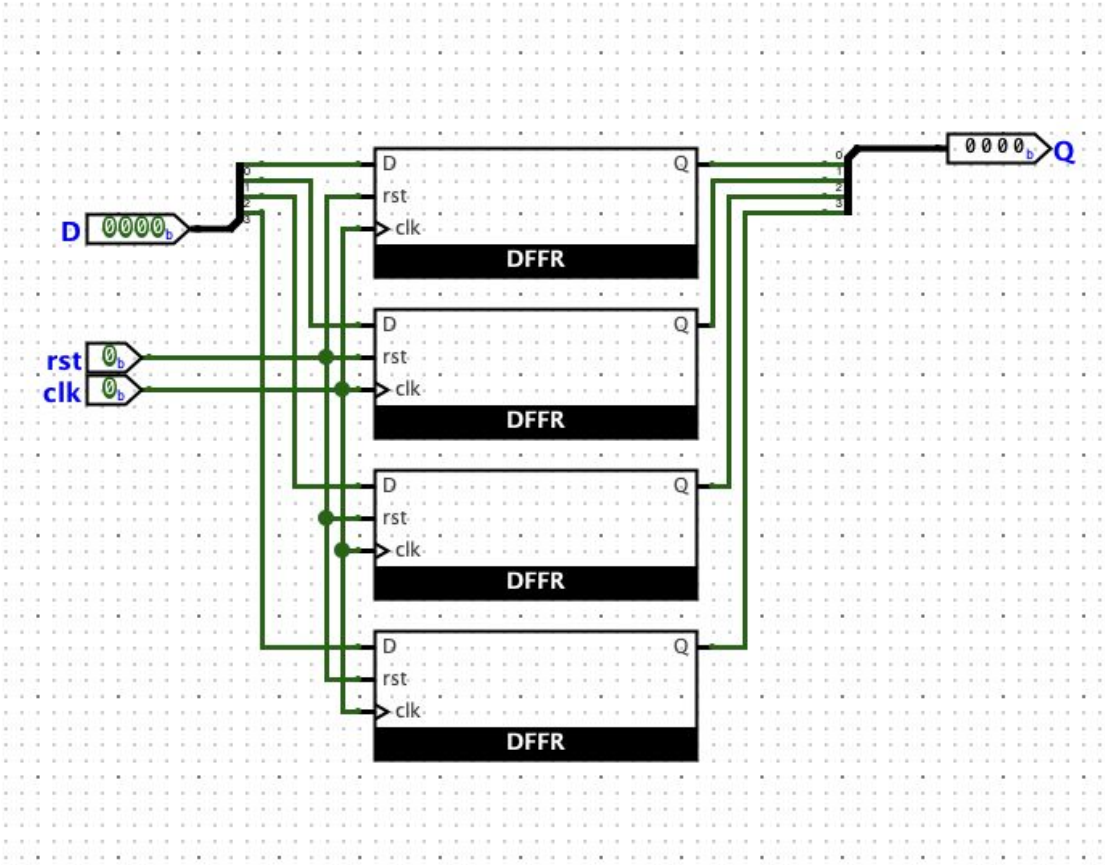
- PC
- ROM
- Decoder
- Register File(GPR)

- Clock

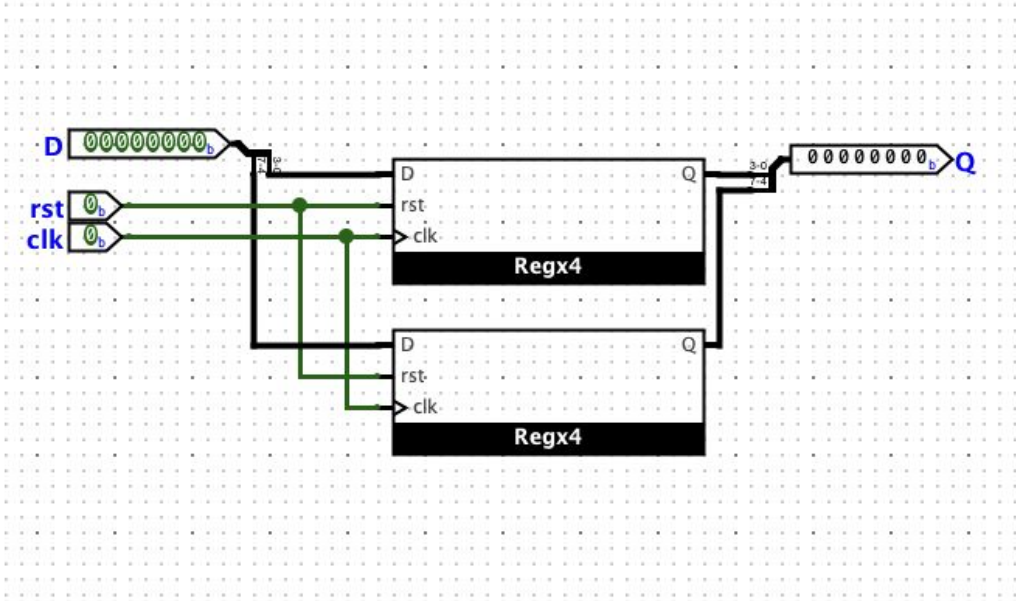
Register

- Try to build register in Logisim for this 8-bit CPU
  - Q1: How many registers the CPU should have?
    - 4 GPRs, and PC
  - Q2: How many bits should each register have?
    - GPR: 8 bits
    - PC: 4 bits
- Hint: Using D Flip-Flop
  - And build 8-bit register with two 4-bit registers

Register (implementation might be varied)



4-bit register



8-bit register

PC

- Store the current instruction address
- next PC = PC + 1

- 4-bit register
- adder
  - To get PC + 1

Instruction ROM

- Get the corresponding instruction of input PC
- 8-bit multiplexer(8 instructions, so 8 choose 1)

- Input: PC (4-bit)
- Output: instruction (8-bit)

Decoder

- Translate instruction

- Input: instruction (8-bit)
- Output: rd, imm, opcode/wen(write enable)

Register File

- Set of GPRs
  - sCPU: 4 GPRs, each 8-bit
    - We need four 8-bit registers
- Input
  - read/write
    - if write, we also need wdata
  - the index of register
    - To let CPU know which register you want to process
- Output:
  - the value of the register, rdata

Try to test your design (li)

- **0: li r0, 10 # 10 in decimal**
- **1: li r1, 0**
- **2: li r2, 0**
- **3: li r3, 1**
- 4: add r1, r1, r3
- 5: add r2, r2, r1
- 6: bner0 r1, 4
- 7: bner0 r3, 7

Have all registers been initialized correctly?

add rd, rs1, rs2

- add R[rs1]+R[rs2] together, and write to R[rd]
- R[rd] = R[rs1]+R[rs2]

- Decoder (rs1, rs2)
- ALU
  - To get = R[rs1]+R[rs2]

ALU

- Execute R[rs1] + R[rs2]
- 8-bit adder

Try to test your design (add)

- 0: li r0, 10 # 10 in decimal
- 1: li r1, 0
- 2: li r2, 0
- 3: li r3, 1
- **4: add r1, r1, r3**
- **5: add r2, r2, r1**
- 6: bner0 r1, 4
- 7: bner0 r3, 7

Have all registers been calculated correctly?

bner0 rs2, addr

- if rs2 not equal to R0, PC jumps to addr

- PC (need MUX to select next PC address)
  - PC + 1 OR addr
- Comparator (compare rs2 with R0)
- Register File (wen = 0, no need to write data)

Try to test your design (bner0)

- 0: li r0, 10 # 10 in decimal
- 1: li r1, 0
- 2: li r2, 0
- 3: li r3, 1
- 4: add r1, r1, r3
- 5: add r2, r2, r1
- **6: bner0 r1, 4**
- **7: bner0 r3, 7**

Is PC being updated correctly?

Review and test all 8 instructions

- 0: li r0, 10 # 10 in decimal
- 1: li r1, 0
- 2: li r2, 0
- 3: li r3, 1
- 4: add r1, r1, r3
- 5: add r2, r2, r1
- 6: bner0 r1, 4
- 7: bner0 r3, 7

PC	R0	R1	R2	R3
0	0	0	0	0
1	10	0	0	0
2	10	0	0	0
3	10	0	0	0
4	10	0	0	1
5	10	1	0	1
6	10	1	1	1
4	10	1	1	1
5	10	2	1	1
…	…	…	…	…

Re-examining the CPU Design

- Process of adding a new instruction
  - Analyze the intended behavior of the instruction
  - Then, according to this behavior, add the required components along the direction of data flow
    - The paths through which data flows in the CPU, together with the associated components on those paths, are referred to as the CPU's **data path**, **including GPR, adder, memory, etc.**

Re-examining the CPU Design

- When data paths of multiple instructions conflict, additional circuits must be introduced to control how data flows.
  - This ensures that each instruction can complete operations in compliance with the ISA specification.
  - These additional circuits belong to the CPU's control logics.
  - The signals that determine the behavior of the control logic are called **control signals**.

Control Signal

Instruction	next PC	wen	wdata	rdata1
li	PC + 1	1	imm	X
add	PC + 1	1	R[rs1]+R[rs2]	rs1
bner0	PC + 1 or ADDR	0	X	0

- X means it could be anything
  - li will not read data
  - bner0 will not write data
- So whatever they choose, it doesn't affect results

Control Signal

Instruction	next PC	wen	wdata	rdata1
li	PC + 1	1	imm	X
add	PC + 1	1	R[rs1]+R[rs2]	rs1
bner0	PC + 1 or ADDR	0	X	0

- The values of control signals are related to the type of instruction
- The instruction type determines the behavior of the control logic, and determines the direction of data flow in the data path, ultimately realizing the functionality specified by the instruction

Practice 1

- Write a sequence of instructions to compute the **sum of odd numbers** less than 10, i.e., **1 + 3 + 5 + 7 + 9**.
- Then, try executing this instruction sequence **on the sCPU** you designed and verify whether the result matches the expected outcome.
- After completing this task, what insights do you gain about the computer's "stored-program" concept?

Practice 2

- Try to add one more instruction ***out rs*** for sISA.
  - After executing ***out rs***, the value of **R[rs]** should be output to the 7-segment display in hexadecimal format. You may choose the encoding of this instruction yourself.
- Then, implement the ***out*** instruction in the sCPU and modify the sequence sum program so that, after the result is computed, it can be displayed on the 7-segment display.
- Hint: use the 7-segment display in Logisim output section

Q&A

Reference: https://ysyx.oscc.cc/docs/en/2407 (Stage F Chapter 5)