

## 第一步：学习RISC-V指令系统

[illegible][illegible]

- |        |   |  |
|--------|---|--|
| R-type | <ul style="list-style-type: none"> <li>– ALU Op: ADD/SUB, SLT, SLTU, (SR), ...</li> </ul>   | <ul style="list-style-type: none"> <li>– Register-Register Op (雙寄存器存取模式)</li> <li>– Store Op (儲存)</li> <li>– Register-Imm Op (雙寄存器存取-立即數操作)</li> </ul> |
| J-type | <ul style="list-style-type: none"> <li>– ALU Op: ADDI/SUBI, SLTI, ANDI, ORI, XORI, (SR), ...</li> <li>– Jump JALUR</li> <li>– Load</li> </ul> | <ul style="list-style-type: none"> <li>– J-type (J, LJUI)</li> <li>– Unconditional Jump (无条件跳轉)</li> <li>– J-type, I-type</li> </ul>                   |
| I-type | <ul style="list-style-type: none"> <li>– LUI, (ALURPC) ...</li> </ul>   | <ul style="list-style-type: none"> <li>– Conditional Jump (有条件跳轉)</li> <li>– R-type</li> </ul>   |
| S-type | <ul style="list-style-type: none"> <li>– JAL</li> </ul>   | <ul style="list-style-type: none"> <li>– Load/Store</li> </ul>   |
| B-type | <ul style="list-style-type: none"> <li>– Branch: BGE, BNE, BLT, ...</li> </ul>  | <ul style="list-style-type: none"> <li>– Load I-type</li> <li>– Store: S-type</li> </ul>   |
| S-type | <ul style="list-style-type: none"> <li>– Store</li> </ul>   |  |

对这37条整型指令进行理解

U	指令名	指令	二进制指令	作用
	LUI	lui x1, 0xffff	000011111111111111110000101011	将20位立即数的值左移12位（低12位补零）成为一个32位数，并其写回x1。这一指令主要是为了在寄存器中存入比较大的数值。比如，要想在寄存器x1中存入一个数，可以用li和add指令实现（add x1,x0,100），但这个数的范围有限（-2048~2047），因为add指令的立即数部分只有12位。能表示最大的无符号数就是0xffff十进制4095）。对应的有符号数的范围则为-2048到2047。当立即数超出这个范围，则需要用LUI指令。
	AUIPC	auipc x2,0xffff	000000001111111111111111000100010111	将立即数的值左移12位（低12位补零）成为一个32位数，再加上该指令的PC值，再将结果写回x2。这条指令与LUI类似，可能是为了实现任意的地址跳转？
	JAL	jal x3,label1 addi x5,x0,5	00000000100000000000000011110111 	PC+4的结果送rd但不送入PC，然后计算下一条指令地址。转移地址采用相对寻址。转移地址为当前指令地址（即PC），偏移量为立即数imm20经符号扩展后的值的2倍。在实际的指令中，跳转的目标往往使用汇编程序中的label，汇编器会自动将label所在地址计算后填入偏移量就写回偏移。
	ADDI	addi x4,x0,4 addi x5,x0,5 addi x6,x0,6 addi x7,x0,7	00000000100001000000000111110111	jal指令使用12位立即数（有符号数）作为偏移量，与操作数寄存器r1+PC的值相加，然后将结果的最低有效位返回。jal指令将下一条指令的PC（即当前指令PC+4）的值写入其结果寄存器rd。
	BNE	addi x1,x0,1 addi x1,x1,x1 addi x2,x0,2 bne x1,x2,label1	11111110001000001000100011000111 11111111111100	bne指令只有在操作数寄存器r1中的数值与操作数寄存器r2的数值不相等时，才会跳转，跳转地址为offset的有符号扩展和最低位补0（即乘以2）的偏移量加上BNE指令的地址。
	BNEI	addi x1,x0,1 label1: addi x1,x1,x1 addi x2,x0,2 bne x1,x2,label1	1111111000100000100011000111000111	bne指令只有在操作数寄存器r1中的数值与操作数寄存器r2的数值不相等时，才会跳转，跳转地址为offset的有符号扩展和最低位补0（即乘以2）的偏移量加上BNEI指令的地址。
	BTL	addi x1,x0,1 label1: addi x1,x1,x1 addi x2,x0,2 btl x1,x2,label1	1111111000100000100011000111000111	btl指令只有在操作数寄存器r1中的数值小于操作数寄存器r2的数值时（有符号数），才会跳转，跳转地址为offset的有符号扩展和最低位补0（即乘以2）的偏移量加上BTL指令的地址。
	BGE	addi x1,x0,-1 label1: addi x1,x1,x1 x2,x0,-2 bge x1,x2,label1	111111100010000010110011000111	bge指令只有在操作数寄存器r1中的数值大于或等于操作数寄存器r2的数值时（有符号数），才会跳转，跳转地址为offset的有符号扩展和最低位补0（即乘以2）的偏移量加上BGE指令的地址。
	BLTU			无符号数
	BGEUI			无符号数
	SB			将操作数寄存器r2中的8位数据，写回存储器
	SH			将操作数寄存器r2中的16位数据，写回存储器
	SW	lui x1,0xd0f11 addi x1,x1,0x7f sw x1,12,x0		将操作数寄存器r2中的32位数据，写回存储器
	LB	addi x1,0xd0f12 addi x1,x1,0x7f sw x1,12,x0 lb x2,12,x0 lb x3,13,x0 lb x4,14,x0 lb x5,15,x0		从指令从存储器中读出一个8位的数据，进行符号位扩展后写回寄存器rd。
	LH	addi x1,0xd0f12 addi x1,x1,0x7f sw x1,12,x0 lh x2,12,x0 lh x3,13,x0 lh x4,14,x0 lh x5,15,x0		从指令从存储器中读出一个16位的数据，进行符号位扩展后写回寄存器rd。
	LW	lui x1,0xd0f11 addi x1,x1,0x7f sw x1,0,x0 lw x2,0,x0		从指令从存储器中读出一个32位的数据，写回寄存器rd。
	LBUI			无符号数
	LHUI			无符号数
	SLT			
	SLTU			
	SLTI			
	SLTIU			
	SRL	addi x1,x0,4 x2,x0,0x0 srl x3,x2,x1		逻辑右移（SRL）根据寄存器（rs2）中的移位量对寄存器（rs1）中的值执行逻辑右移，左边空出的位补0，并存储在（rd）寄存器中。 
	SRA			
	SLLI			
	SRLI			
	SRAI			
	ADDI	addi x2,x1,x1		add指令将寄存器（rs1）与寄存器（rs2）中的值相加，并写回（rd）寄存器中。
	SUB	sub x3,x2,x1		sub指令将寄存器（rs1）与寄存器（rs2）中的值相减，并写回（rd）寄存器中。
	ADDI	addi x1,x0,-1		add指令将操作数寄存器rs1的数值加上12位立即数进行加法操作，结果写回寄存器rd。
	XOR			
	OR			
	AND			
	XORI			

**第二步：单周期处理器的设计**

cpu包括控制单元和数据通路

## 初步实现六条指令

- JALR
- BEQ
- LW
- SW
- ADD
- ADDI

### 6条指令的功能总结

[illegible]

### 6条指令的功能总结

31	25	24	23	19	15	14	12	11	7	6	0
imm[11:0]		r2	r0	r0	r0	imm[5:2]	r0	imm[11:1]		rW	

**SW** Store word (字写入内存)。功能：MEM[GP[R] $\times$ 4+imm] = GP[R] $\times$ 2

31	25	24	23	19	15	14	12	11	7	6	0
imm[11:0]		r2	000	r0	imm[5:2]		imm[11:1]		ADDR		

**ADDI** 立即数加法。功能：GP[R] $\times$ 2 = GP[R] $\times$ 1 + sign extended (带符号扩展)  
12-bit immediate (立即数)

31	25	24	23	19	15	14	12	11	7	6	0
imm[11:0]		r2	000	0000	r0	imm[5:2]		imm[11:1]		ADDR	

**ADD** 加法。功能：GP[R] $\times$ 2 = GP[R] $\times$ 1 + GP[R] $\times$ 2

- Instruction fetch (IF, 指令获取)
- Instruction decode and register operand fetch (ID/RF, 指令译码与寄存器操作数获取)
- Execute/evaluate memory address (EX/WG, 执行/计算内存地址)
- Memory operand fetch (MEM, 计算数据获取)
- Writeback result (WB, 写回结果)

[illegible]

指令	使用方法																		
对 rom 进行初始化	只需要将二进制的指令写入到文件 "rom_binary_file.txt" 中。																		
从 rom 数据存储器中读出的数据	<p>由于 rom 指令中指令字元的书写方式，所以读取数据存储器的设计也应该支持多位数读写，根据输入给 rom 的 17 位信号来确读数据类型。这里的 17 位信号，其数据类型为 rom 的 16mc 信号。</p> <table><tr><th>RW_type</th><th>对应信号</th><th>该信号写</th></tr><tr><td>000</td><td>ls,lsb</td><td>读取或写入一个字，读取时将 16mc 信号扩展为 32bit</td></tr><tr><td>001</td><td>ls,ls</td><td>读取或写入两个字，读取时将 16mc 信号扩展为 32bit</td></tr><tr><td>010</td><td>ls,ls,2w</td><td>读取或写入 4 个字节</td></tr><tr><td>100</td><td>ls,ls,2w</td><td>读取或写入，将 16 位无符号字扩展为 32bit</td></tr><tr><td>101</td><td>ls,ls</td><td>读取两个字，将 16 位无符号字扩展为 32bit</td></tr></table>	RW_type	对应信号	该信号写	000	ls,lsb	读取或写入一个字，读取时将 16mc 信号扩展为 32bit	001	ls,ls	读取或写入两个字，读取时将 16mc 信号扩展为 32bit	010	ls,ls,2w	读取或写入 4 个字节	100	ls,ls,2w	读取或写入，将 16 位无符号字扩展为 32bit	101	ls,ls	读取两个字，将 16 位无符号字扩展为 32bit
RW_type	对应信号	该信号写																	
000	ls,lsb	读取或写入一个字，读取时将 16mc 信号扩展为 32bit																	
001	ls,ls	读取或写入两个字，读取时将 16mc 信号扩展为 32bit																	
010	ls,ls,2w	读取或写入 4 个字节																	
100	ls,ls,2w	读取或写入，将 16 位无符号字扩展为 32bit																	
101	ls,ls	读取两个字，将 16 位无符号字扩展为 32bit																	

说明 时钟信号 寄存器1 寄存器2 读使能 目标寄存器 寄存器的数据 寄存器1的数据 寄存器2的数据	
位1至位10数据	
操作数1 操作数2 控制信号 计算结果中的标志 标志 运算结果	<p>(1) 加法运算还输出给，检测结果就是，当时，最左边的位为0，正数时，最左边的位为1。</p> <p>具体每一位有以下四种情况</p> <p>a 正数、正数、负数，则</p> <p>b 负数、正数、正数，则</p> <p>c 正数 负数、负数，则</p> <p>d 负数 正数、正数，则</p> <p>(2) 逻辑运算，用逻辑实现即可。</p> <p>(3) 小于等于、等于运算，根据加法的运算结果，判断是否小于、等于。</p> <p>(4) 移位操作，这里</p> <pre>         output &lt;= (code)         input [1:0]mov,         output [6:0]opcode         output [2:0]func3,         output func3,         output [4:0]r1,         output [4:0]r2,         output [4:0]rd,         output [1:0]imm       </pre>
时钟周期中的pc值 值 输出的指令	<pre>         wire l_type;         wire U_type;         wire l_type;         wire R_type;       </pre>

[illegible]

```

reg [31:0] Wv_data; // 字节拼接
wire [31:0] Rd_data_H; // 半字拼接

assign Rd_data=ram[addr[31:2]]; // 读基址????????
always@(*)
begin
    case(addr[1:0])
        0://通过地址的最后两位确定数据的读法? ? ?
            Zb00=Wv_data_B-[Rd_data[31:8].div[7:0]];
            Zb01=Wv_data_B-[Rd_data[31:16].div[7:0].Rd_data[7:0]];
            Zb10=Wv_data_B-[Rd_data[31:24].div[7:0].Rd_data[15:0]];
            Zb11=Wv_data_B-[div[7:0].Rd_data[23:0]];
            endcase
    end

//半字拼接, addr[1] 确定拼接顺序
assign Wv_data_H={addr[1]} ? {div[15:0].Rd_data[15:0]} :
{Rd_data[31:16].div[15:0]};

//根据写类型, 选择写入的数据
//根据写地址, 则为1b 读取或写入32bit
//等于2'b00 将mem03b 置为32bit 数据为字节拼接; 等于2'b01 16bit 扩展为
32bit 数据为半字拼接;

assign Wv_data={RW_type[1:0]==2'b00} ? Wv_data_B : {RW_type[1:0]==
2'b01} ? Wv_data_H : Wv_data;
//上升沿写入数据
//读数据
begin
    if(W_en) //如果写使能 wr_data 写入地址为addr[19:2]的mem数据区
        读
    else
        读;
    if(R1) : read;
    if(R2) : read;
    //读数据

reg [7:0] Rd_data_B;
wire [15:0] Rd_data_H;

wire [31:0] Rd_data_B_ext;
wire [31:0] Rd_data_H_ext;

```

```

        assign Rd_data, {
            begin
                case(addr[1:5])
                    7'b0001_Rd_data_BuH_data[7:0];
                    7'b0101_Rd_data_BuH_data[15:8];
                    7'b1001_Rd_data_BuH_data[23:16];
                    7'b1101_Rd_data_BuH_data[31:24];
                endcase
            end

            assign Rd_data_H={addr[1:1] ? Rd_data[31:16] : Rd_data[15:0];

            //将data符号扩展为32bit, new_type2是用符号判断是否无符号数
            assign Rd_data_B_ext=(WV_type2 ? {24'd0,Rd_data,B} :

```

	pc_newpcimm6, jalrd, pc_newdata[431]imm6,	rst_n 1bit input 32bit output 下一个时钟周期pc值 pc_new 32bitb output更新后的pc值	读出的指令	output [4:0]b1, output [4:0]b2, output [31:0]imm6 output [31:0]imm6 };
数据通路	数据通路就是将以上的几个关键的部分进行连接，为了形成完整的数据通路，还需要加一些数据选择器，作用就是对不同的数据的来源进行选择，并输出到指定的模块。	信号名 位置 输入输出	输入输出	write 1_type; write 1_type; write 1_type; write 5_type; write [31:0]_imm6; write [31:0]_imm6; write [31:0]_imm6; write [31:0]_imm6; write [31:0]_imm6; write [31:0]_imm6; assign opcodes=
	比如：写入寄存器的数据来源于5个，分别是：ALU的运算结果，数据寄存器读出的数据，pc+4值的立即数，auipc的pc+imm6，	data 1bit input ALU的运算结果数据选择器的控制信号 RegWrite 1bit input 寄存器写的数据使能控制信号 lui 1bit input lui指令标志，写入寄存器的数据选择器的控制信号 ui_type 1bit input ui_type指令标志，写入寄存器的数据选择器的控制信号 p2c 1bit input p2c指令标志，选择p2c数据选择器的控制信号，同时也是写入寄存器的数据选择器的控制		assign opcodes=
	由的结果加上一个 pc+4 4+imm6,Read_data4+imm6, 另外，还需要加入两个加法器，分别计算 pc+4和pc+imm6	指令 控制字 bneq 1bit input bneq指令标志，判断是否跳转的控制信号 bneq 1bit input bneq指令标志，判断是否跳转的控制信号 blt 1bit input blt指令标志，判断是否跳转的控制信号 bge 1bit input bge指令标志，判断是否跳转的控制信号 bltu 1bit input bltu指令标志，判断是否跳转的控制信号 bgeu 1bit input bgeu指令标志，判断是否跳转的控制信号 ALUctrl 4bit input ALU的控制信号，决定ALU进行什么运算 Read_imm6 32bit input 从数据寄存器读出的数据，作为写入寄存器的数据来源之一 imm6_addr 32bit input 指令存储器的地址 Wr_imm6 32bit input 数据寄存器写的结果 ALU_result 32bit output ALU的运算结果，作为数据寄存器读出的（写）地址 opcode7 1bit input 7位操作码 func3 1bit input func3 func7 1bit input func7		assign opcodes=

控制线	解释	信号名 位置				说明	endmodule
		opcode	7bit	input	7位操作码		
子控制器	由不同类型的指令所经过的数据通路不同,所以需要有控制信号控制数据通路,使得数据经过正确的通路,得到正确的运算结果。	Memread	1bit	output	数据存储器读使能	子控制器的控制信号	
	本设计将控制线分为两级控制,主控制器产生大一级控制信号,子控制器是ALU控制器,产生控制ALU进行正确运算的信号。	ALUOp	2bit	input	子控制器的控制信号		
		MemtoReg	1bit	output	写回寄存器的数据选择器控制信号		
		Memwrite	1bit	output	数据存储器写使能		
		ALUSrc	1bit	output	ALU的数据来源寄存器数据选择器控制信号		
		RegWrite	1bit	output	寄存器的写使能控制信号		
		Isr	1bit	output	Isr指令标志,写回寄存器的数据选择器的控制信号		
		Uj_type	1bit	output	Uj-type指令标志,写回寄存器的数据选择器的控制信号		
		jalr	1bit	output	jalr指令标志,选择r <sub>2</sub> 的数据选择器的控制信号,同时也是写回寄存器的数据选择器的控制信号		
		jalr	1bit	output	jalr指令标志,选择r <sub>2</sub> 的数据选择器的控制信号,同时也是写回寄存器的数据选择器的控制信号		
		bneq	1bit	output	bneq指令标志,判断是否跳转的控制信号		
		bne	1bit	output	bne指令标志,判断是否跳转的控制信号		
		blt	1bit	output	blt指令标志,判断是否跳转的控制信号		
		bge	1bit	output	bge指令标志,判断是否跳转的控制信号		
		bgtu	1bit	output	bgtu指令标志,判断是否跳转的控制信号		
		bgeu	1bit	output	bgeu指令标志,判断是否跳转的控制信号		
		RegW_type	3bit	output	数据寄存器端的读(写)类型		
子控制器	子控制器根据主控制器产生的ALUop信号,结合func和func3信号来产生ALUin信号。	ALUOp	2bit	input	主控制器产生的ALUop信号	子控制器产生的ALUop信号	
	ALUOp 子控制器的操作	func3	3bit	input	func3		
	00 加法运算	func7	7bit	input	func7		
	01 R-type指令,根据func和func3判断运算类型	ALUctl	4bit	output			
	01 R-type指令,根据func和func3判断运算类型						
	11 条件跳转指令,根据func3判断运算类型						
	ALUctl信号设置如下:						
	ALUctl 运算类型						
	0000 加法运算 (真的是0001???)						
	0011 加法运算						
	0100 逻辑左运算						
	0101 逻辑左运算						
	0110 逻辑左运算						
	0111 逻辑左运算						
	1000 小于等于						
	1001 小于等于						
	1100 移位运算						
	1101 移位运算						
	1110 移位运算						

ok, 上面步骤完成之后, 会拥有以下文件

alluv	2022/1/15 20:04	V 2/31
control.v	2022/1/15 20:51	V 2/31
data_memory.v	2022/1/15 14:01	V 2/31
datapath.v	2022/1/15 20:17	V 2/31
define.v	2022/1/15 14:36	V 2/31
instr_decoder.v	2022/1/15 14:40	V 2/31
instr_memory.v	2022/1/15 13:03	V 2/31
mutex.v	2022/1/15 20:06	V 2/31
pc_reg.v	2022/1/15 16:17	V 2/31
registers.v	2022/1/15 14:17	V 2/31
rom_binary_file.txt	2022/1/15 12:47	308x256

指令寄存器, 指令译码器

数据存储器, 数据寄存器

计算单元

数据通路, 多路选择器

nc寄存器

控制器、计算控制器

但是他们现在是分散的模块，需要将他们连接起来

RISC-V核心-----将数据通路和控制信号连接

信号名	位置	输入/输出	说明
clk	1bit	input	系统时钟
		input	复位信号
rst_n	1bit		
instr	32bit	input	来自指令存储器的指令
rdg_mem_data	32bit	input	来自数据存储器读的数据
rom_addr	8bit	output	指令存储器的地址线
Wt_mem_data	32bit	input	写入数据存储器数据
rs_en	1bit	output	数据存储器读使能信号
Wt_en	1bit	output	数据存储器写使能信号
ram_addr	32bit	output	数据存储器读(写)地址
RW_type	3bit	output	数据存储器读写类型

▣ <b>alias</b>	2022/1/6 20:04	V20H
▣ <b>control</b>	2022/1/6 20:51	V20H
▣ <b>data_memory_v</b>	2022/1/6 14:01	V20H
▣ <b>datapath_v</b>	2022/1/6 20:17	V20H
▣ <b>define_v</b>	2022/1/6 14:36	V20H
▣ <b>instr_decoder_v</b>	2022/1/6 16:48	V20H
▣ <b>instr_memory_v</b>	2022/1/6 13:03	V20H
▣ <b>mutex</b>	2022/1/6 20:06	V20H
▣ <b>pc_mux_v</b>	2022/1/6 16:17	V20H
▣ <b>registers_v</b>	2022/1/6 14:17	V20H
▣ <b>reset_v</b>	2022/1/6 21:02	V20H
▣ <b>stop_mux_v</b>	2022/1/6 21:03	V20H
▣ <b>rom_binary_file_test</b>	2022/1/6 12:47	无版本

加入这两个文件之后相当于，将控制器、数据选择器、数据存储器、指令寄存器结合了起来。

完成了单周期cpu的理论设计, 下面开始尝试仿真

```
** Error: (vlog-1366) D:\models\single_cycle_cpu\sn.v(284): near "(": syntax
error, unexpected '('.
```

```

801 // 如果不小心错了1组,那就没有40000这个AUU.CTL了
802 // 根据判断所添加的时候,发现他的加法AUU.CTL错了这个80001
803 // assign ADD_overflow(AUU_CTL:=+40000 & -431) & -431; // 正数+负数=负数, 则溢出
804 // ((AUU_CTL:=+80000) & 431) & 431; // 正数+正数=正数, 则溢出
805 // ((AUU_CTL:=+80000) & 431) & -431; // 正数+负数=负数, 则溢出
806 // ((AUU_CTL:=+80000) & -431) & -431; // 正数+正数=正数
807 // ((AUU_CTL:=+80000) & 431) & 431; // 负数+负数=正数, 则溢出

```







