合肥工業大學

《系统硬件综合设计》
报告

学生姓名 刘嘉伟
学　　号 2018213106
专业班级 计算机创新实验 18-1 班
指导教师 阙夏
院系名称 计算机与信息学院

2021 年　1 月　15 日

一、实验目的

设计并实现一个多周期流水 CPU

- 三种类型的指令各若干条
- MIPS、ARM、RISC-V 等类型 CPU 都可以
- 下载到 FPGA 上进行验证

二、实验平台

Vivado 2020.2

三、实验设计与过程

我设计的 CPU 指令集是 RISC-V32I，该指令集总共包含 47 条指令，我实现了其中的 37 条。

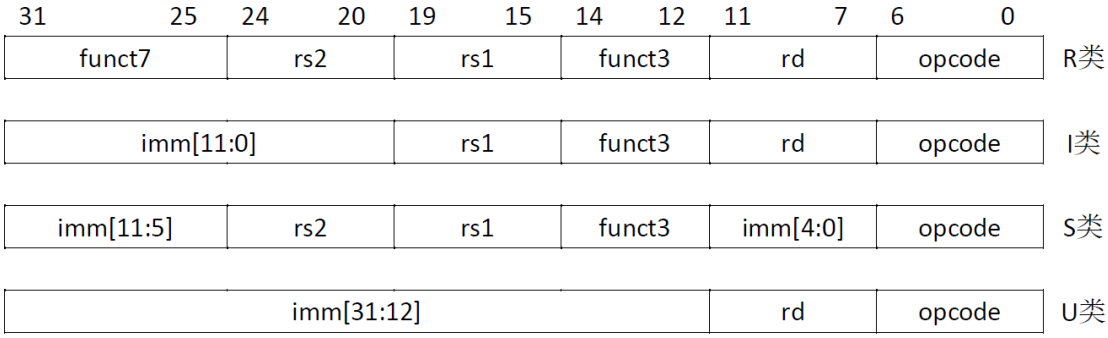
3.1 RISC-V 32I 指令格式

有四种核心指令格式（R/I/S/U），所有的指令长度都是固定的 32 位，并且在存储器中必须 4 字节边界对齐。

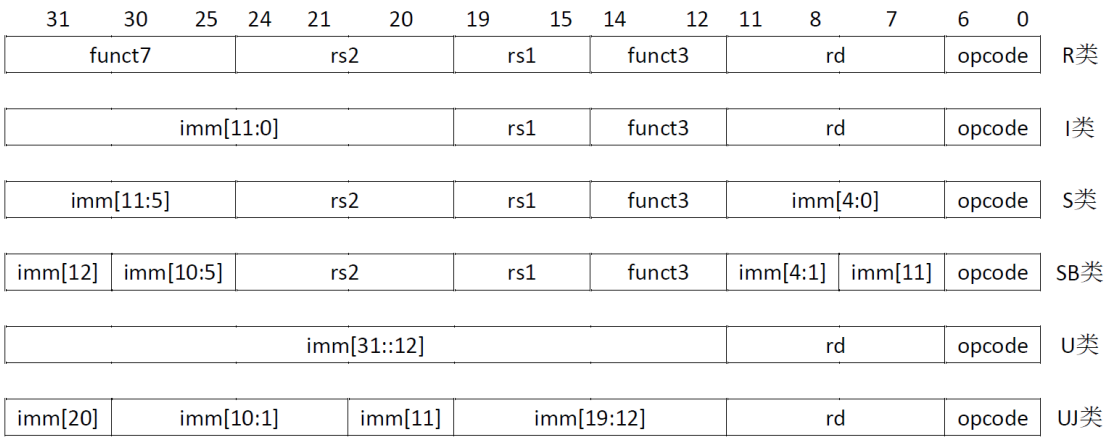
其中 opcode 字段指明了该条指令的操作类型，一般对于一大类指令，再根据 funct3 或/和 funct7 字段来确定具体指令（某些指令直接由 opcode 确定）

rs1，rs2，rd 字段，分别指源操作数 1、源操作数 2、目的操作数的寄存器编号，在所有的格式中，这三者位置是固定的，方便指令译码。

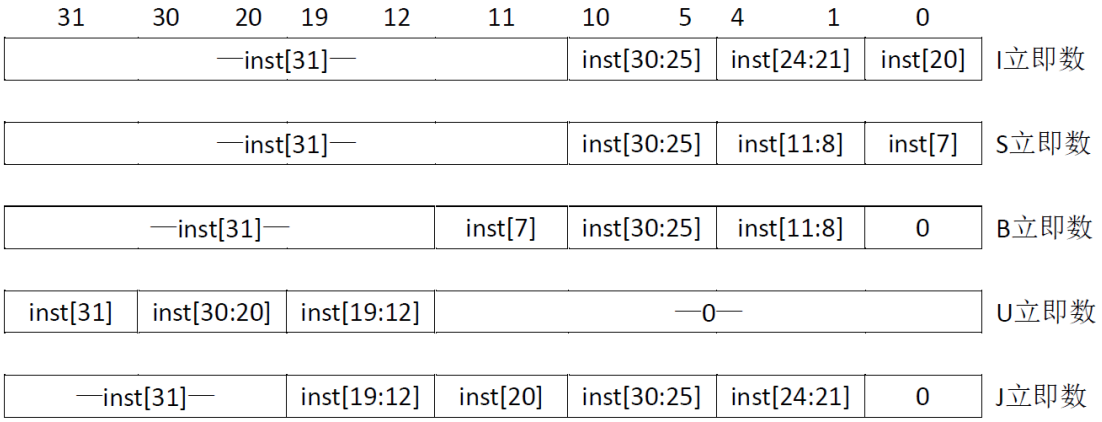
另外，所有立即数的符号位总是在指令的第 31 位，以加速符号扩展电路。



为了方便对立即数的处理，增加了两种指令格式变种，由 S 类派生出的 SB 类，由 U 类派生出的 UJ 类



除了 R 类指令，剩下五类指令中都包含立即数，使用立即数前应该对立即数进行符号拓展（32 位），得到以下 5 类立即数



其中 inst[x]表示，立即数的这一位是指令中的第 x 位(0-31)，符号拓展总是在指令的第 31 位。

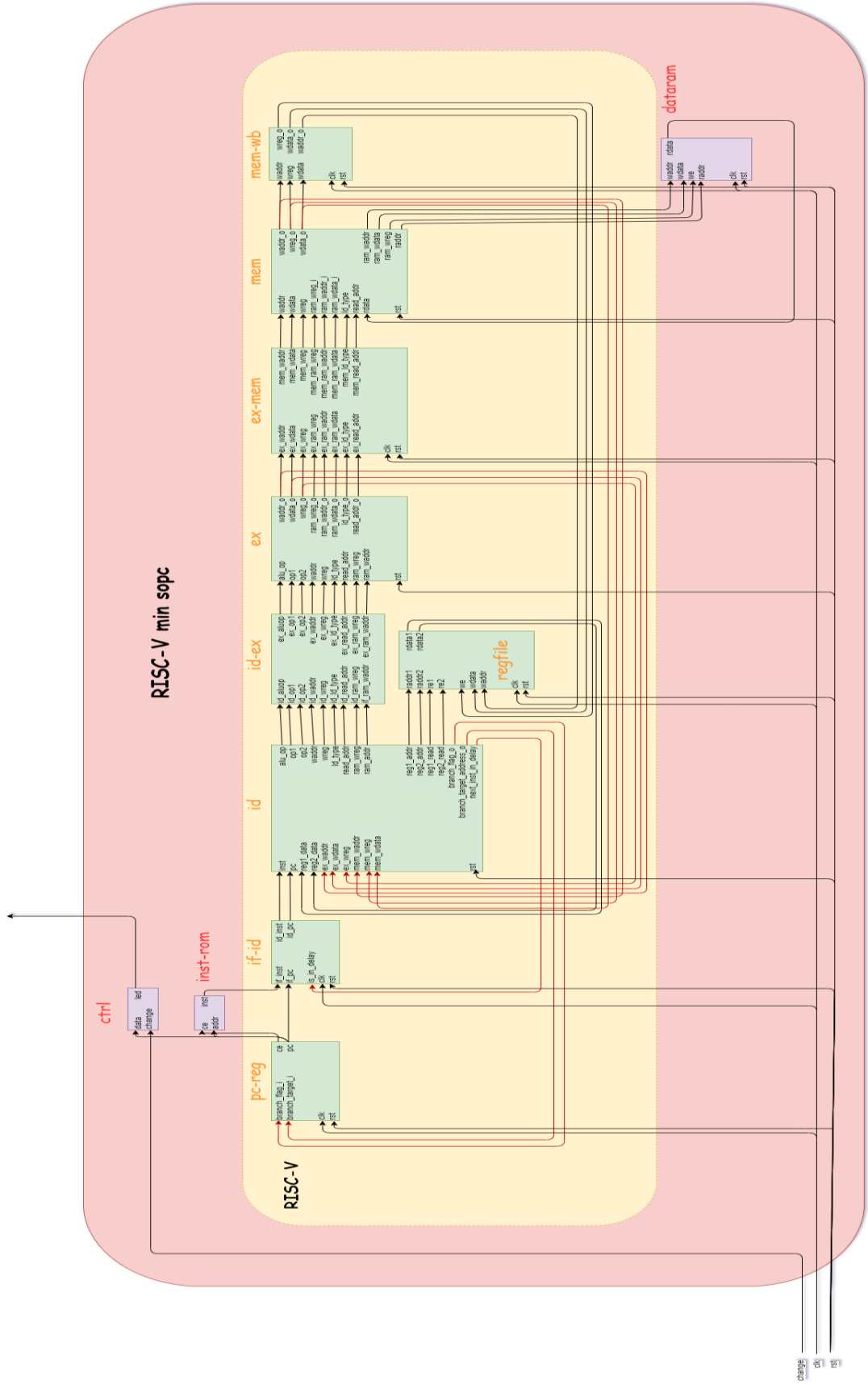
S 和 SB 格式唯一的区别在于，在 SB 格式中，12 位立即数字段用于编码 2 的倍数的分支偏移量。（SB 立即数的第 0 位固定为 0）

U 和 UJ 格式唯一的区别在于，20 位立即数被左移 12 位以生成 U 立即数，而被左移 1 位以生成 J 立即数。

3.2 计划实现的指令列表如下

指令	类型	Inst[31:25]	Inst[24:20]	Inst[19:15]	Inst[14:12]	Inst[11:7]	Inst[6:0]
LUI	U	imm[31:12]				rd	0110111
AUIPC	U	imm[31:12]				rd	0010111
JAL	J	imm[20|10:1|11|19:12]				rd	1101111
JALR	I	imm[11:0]		rs1	000	rd	1101111
BEQ	B	imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011
BNE		imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011
BLT		imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011
BGE		imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011
BLTU		imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011
BGEU		imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011
LB	I	imm[11:0]		rs1	000	rd	0000011
LH		imm[11:0]		rs1	001	rd	0000011
LW		imm[11:0]		rs1	010	rd	0000011
LBU		imm[11:0]		rs1	100	rd	0000011
LHU		imm[11:0]		rs1	101	rd	0000011
SB	S	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011
SH		imm[11:5]	rs2	rs1	001	imm[4:0]	0100011
SW		imm[11:5]	rs2	rs1	010	imm[4:0]	0100011
ADDI	I	imm[11:0]		rs1	000	rd	0010011
SLTI		imm[11:0]		rs1	010	rd	0010011
SLTIU		imm[11:0]		rs1	011	rd	0010011
XORI		imm[11:0]		rs1	100	rd	0010011
ORI		imm[11:0]		rs1	110	rd	0010011
ANDI		imm[11:0]		rs1	111	rd	0010011
SLLI		0000000	rs2	rs1	001	rd	0010011
SRLI		0000000	rs2	rs1	101	rd	0010011
SRAI		0100000	rs2	rs1	101	rd	0010011
ADD	R	0000000	rs2	rs1	000	rd	0110011
SUB		0100000	rs2	rs1	000	rd	0110011
SLL		0000000	rs2	rs1	001	rd	0110011
SLT		0000000	rs2	rs1	010	rd	0110011
SLTU		0000000	rs2	rs1	011	rd	0110011
XOR		0000000	rs2	rs1	100	rd	0110011
SRL		0000000	rs2	rs1	101	rd	0110011
SRA		0100000	rs2	rs1	101	rd	0110011
OR		0000000	rs2	rs1	110	rd	0110011
AND		0000000	rs2	rs1	111	rd	0110011

3.3 整体设计图


RISC-V min sopc
RISC-V
pc-reg
inst-rom
if-id
id
id-ex
ex
ex-mem
mem
mem-wb
regfile
dataram
ctrl

3.4 常量定义

预先在模块 defines.v 中定义实验中使用的常量，方便阅读。

```
`define RstEnable           1'b1
`define RstDisable          1'b0
`define ZeroWord            32'h00000000
`define WriteEnable   1'b1
`define WriteDisable        1'b0
`define ReadEnable          1'b1
`define ReadDisable   1'b0
`define ChipEnable          1'b1
`define ChipDisable   1'b0

`define I_OP        7'b0010011
`define R_OP            7'b0110011
`define B_OP            7'b1100011
`define LUI_OP          7'b0110111
`define AUIPC_OP    7'b0110111
`define S_OP            7'b0100011
`define JAL_OP          7'b1101111
`define JALR_OP             7'b1100111
`define LOAD_OP             7'b0000011



`define ALU_NO_OP       4'b1111
`define ADD    4'b0000
`define SLL    4'b0001
`define SLT    4'b0010
`define SLTU 4'b0011
`define XOR    4'b0100
`define SRL    4'b0101
`define OR      4'b0110
`define AND    4'b0111
`define SUB    4'b1000
`define LUI    4'b1001
`define SB      4'b1010
`define SH      4'b1011
`define SW     4'b1100
`define SRA    4'b1101
`define JAL    4'b1110

`define ADDI 4'b0000
`define SLLI 4'b0001
```

```
`define SLTI 4'b0010
`define SLTIU 4'b0011
`define XORI 4'b0100
`define SRLI 4'b0101
`define ORI 4'b0110
`define ANDI 4'b0111
`define SRAI 4'b1101

`define Branch 1'b1
`define NotBranch 1'b0
`define BEQ    3'b000
`define BNE    3'b001
`define BLT    3'b100
`define BGE    3'b101
`define BLTU 3'b110
`define BGEU 3'b111

`define LB      3'b000
`define LW      3'b010
`define LH      3'b001
`define LBU    3'b100
`define LHU    3'b101

//指令存储器 inst_rom
`define InstAddrBus 31:0
`define InstBus 31:0
`define InstMemNum 131071
`define InstMemNumLog2 17

//通用寄存器 regfile
`define RegAddrBus 4:0
`define RegBus 31:0
`define SelBus 2:0
`define OpBus 3:0

`define NOPRegAddr 5'b00000
```

在该模块中定义了数据总线宽度，数据地址线宽度等一系列宽度，还定义了各种使能信号，如芯片使能、读使能，写使能等。里面还定义了各类指令的指令码 opcode，以及指令码的子类型编码 funct3 和 funct7。

3.5 各模块具体设计与编写

pc_reg.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟信号
input	[0:0]	rst	复位信号
input	[0:0]	branch_flag_i	是否跳转标志
input	[31:0]	branch_target_i	跳转目标地址
output	[31:0]	pc	下一条指令地址
output	[0:0]	ce	是否取指信号

该模块是 pc 模块，输入跳转标志和跳转的目标地址来进行指令的跳转，并且内部的 PC 每个时钟周期自增 4，因为 CPU 是按字节存储，每条指令一个字，占 4 个字节，所以每次都要递增 4。当跳转标志为真时，PC 将跳转至目标地址，否则自增 4。具体代码如下：

```
module pc_reg(
    input wire clk,
    input wire rst,

    input wire branch_flag_i,
    input wire[`InstAddrBus] branch_target_i,

    output reg[`InstAddrBus] pc,
    output reg ce
);

    always @ (posedge clk) begin
        if (ce == `ChipDisable) begin
            pc <= 32'h00000000;
        end else begin
            if (branch_flag_i == `Branch) begin
                pc <= branch_target_i;
                pc[0] <= 0;
                pc[1] <= 0;
            end else begin
                pc <= pc + 4'h4;
            end
        end
    end

    always @ (posedge clk) begin
        if (rst == `RstEnable) begin
            ce <= `ChipDisable;
        end else begin
            ce <= `ChipEnable;
        end
    end

endmodule
```

if_id.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟信号
input	[0:0]	rst	复位信号
input	[0:0]	is_in_delay	该条指令是否处于延迟槽
input	[31:0]	if_pc	该条指令地址
input	[31:0]	if_inst	该条指令值
output	[31:0]	id_pc	指令地址输出到译码模块
output	[0:0]	id_inst	指令值输出到译码模块

该模块是取指和译码的中间模块，将输入的指令值和指令地址送到译码模块，其中的时钟信号是用于控制流水，使每个时钟周期向后处理一步，也就是流水，后面的中间模块的时钟的作用都是为了控制流水，在后面就不一一解释。

如果译码阶段发现了转移指令，那么当前处于取指阶段的指令就处于延迟槽中，也就是一条无效指令，所以为了方便，当在译码阶段检测出当前指令是一条转移或跳转指令的时候，将会发送一个延迟信号到本模块，当本模块处于延迟槽时，指令无效，所以在这个模块将下一步输送的指令设为无效指令，到后续模块如果遇到无效指令自然不会去处理执行，所以就实现了转移指令的延迟槽处理。但这有个细节需要注意，在这个模块中，进行的判断是如果指令不处于延迟槽的话才会将原始指令输送到译码阶段，所以这就造成了一个问题，CPU 刚开始工作的时候延迟槽信号没有数据输送过来，信号是高阻状态，将会无法传递指令。为了解决这个问题，我设置了一个开关 open，初始阶段 open 为 true，为传递指令增加一个判断条件，如果 open==true，就可以直接传递指令到下一模块，然后将 open 关掉，open=false，经过一个周期后，延迟槽信号都会出现值，所以就可以直接通过 is_in_delay 进行判断是否处于延迟槽了。open 信号也就没有作用了。代码如下：

```
module if_id(

    input wire clk,
    input wire rst,
    input wire is_in_delay,

    input wire[`InstAddrBus] if_pc,
    input wire[`InstBus] if_inst,
    output reg[`InstAddrBus] id_pc,
    output reg[`InstBus] id_inst
);

    reg open = 1'b1;
    always @ (posedge clk) begin
        if (rst == `RstEnable) begin
            id_pc <= `ZeroWord;
            id_inst <= `ZeroWord;
        end else begin
            id_pc <= if_pc;
            if (is_in_delay == 1'b0 || open == 1'b1) id_inst <= if_inst;
            else id_inst <= 32'b0;
            open <= 1'b0;
        end
    end

endmodule
```

id			
输入/输出	宽度	信号名	说明
input	[0:0]	rst	复位信号
input	[31:0]	inst	指令值
input	[31:0]	pc	当前指令的地址
input	[31:0]	reg1_data	寄存器端口 1 的值
input	[31:0]	reg2_data	寄存器端口 2 的值
input	[0:0]	ex_wreg	ex 模块数据前推读信号
input	[31:0]	ex_wdata	ex 模块数据前推数据
input	[4:0]	ex_waddr	ex 模块数据前推地址
input	[0:0]	mem_wreg	mem 模块数据前推读信号
input	[31:0]	mem_wdata	mem 模块数据前推数据
input	[4:0]	mem_waddr	mem 模块数据前推地址
output	[0:0]	wreg	寄存器写信号
output	[4:0]	waddr	写寄存器地址
output	[31:0]	op1	操作数 1
output	[31:0]	op2	操作数 2
output	[3:0]	alu_op	运算类型
output	[4:0]	reg1_addr	寄存器读地址 1
output	[4:0]	reg2_addr	寄存器读地址 2
output	[0:0]	reg1_read	寄存器端口 1 读使能
output	[0:0]	reg2_read	寄存器端口 2 读使能
output	[0:0]	branch_flag_o	跳转信号
output	[31:0]	branch_target_address_o	跳转地址
output	[0:0]	next_inst_in_delay	下条指令是否处于延迟槽
output	[0:0]	ram_wreg	ram 写使能
output	[31:0]	ram_addr	ram 写地址
output	[31:0]	read_addr	ram 读地址
output	[2:0]	ld_type	load 指令的类型

该模块是译码模块，非常重要的一个模块，处理指令的译码，为后续运算提供基础。该模块的设计比较复杂，我将分指令来介绍工作原理。

I 类型指令：

该类型指令是移位或加减运算指令，操作数 1 是位于 rs1 的寄存器值，操作数 2 是立即数 imm，立即数的译码如下：

```
`I_OP: begin
    reg1_read <= `ReadEnable;
    waddr <= rd;
    wreg <= `WriteEnable;
    if (fun3 == 3'b101) begin
        tmp <= inst[30];
    end
    alu_op <= {tmp, fun3};
    case ({tmp, fun3})
        `ADDI: imm <= {20'h0,inst[31:20]};
        `SLTI: imm <= {20'h0,inst[31:20]};
        `SLTIU:imm <= {20'h0,inst[31:20]};
        `XORI: imm <= {20'h0,inst[31:20]};
        `ORI: imm <= {20'h0,inst[31:20]};
        `ANDI: imm <= {20'h0,inst[31:20]};
        `SLLI: imm <= {27'h0,inst[24:20]};
        `SRLI: imm <= {27'h0,inst[24:20]};
        `SRAI: imm <= {27'h0,inst[24:20]};
    endcase
end
```

R 类型指令：

R 类型指令与 I 类型指令的作用相似，唯一不同的是 I 类型指令的第二操作数是 imm 立即数，而 R 类型指令的操作数是位于 rs2 的寄存器的值。由于两者的运算是一致的，因此 alu_op 的赋值是一致的，能简化代码。

```
`R_OP: begin
    reg1_read <= `ReadEnable;
    reg2_read <= `ReadEnable;
    waddr <= rd;
    wreg <= `WriteEnable;
    alu_op <= {inst[30],fun3};
end
```

B 类型指令：

B 类型指令是条件转移指令，需要读取 rs1 和 rs2 寄存器中的值 ，然后依据具体的 funct3 的值来判断运算的类型：

```
case (fun3)
    `BEQ: if (num1 == num2) branch_flag_o <= `Branch;
    `BNE: if (num1 != num2) branch_flag_o <= `Branch;
    `BLT: if ($signed(num1) < $signed(num2)) branch_flag_o <= `Branch;
    `BGE: if ($signed(num1) >= $signed(num2)) branch_flag_o <= `Branch;
    `BLTU: if (num1 < num2) branch_flag_o <= `Branch;
    `BGEU: if (num1 >= num2) branch_flag_o <= `Branch;
endcase
```

num1 和 num2 是读取的 rs1 和 rs2 寄存器的值，原本只需要用 op1 和 op2 就能进行判断，但是不知道为什么在用 op1 或 op2 的时候，仿真的时候就会直接停止，所以我又定义了两个变量与 op1 和 op2 类型相同，赋值语句也相同。

```
if((ex_wreg == `WriteEnable) && (ex_waddr == reg1_addr)) begin
    num1 <= ex_wdata;
end else if((mem_wreg == `WriteEnable) && (mem_waddr == reg1_addr)) begin
    num1 <= mem_wdata;
end else begin
    num1 <= reg1_data;
end

if((ex_wreg == `WriteEnable) && (ex_waddr == reg2_addr)) begin
    num2 <= ex_wdata;
end else if((mem_wreg == `WriteEnable) && (mem_waddr == reg2_addr)) begin
    num2 <= mem_wdata;
end else begin
    num2 <= reg2_data;
end
```

LUI 指令：

该指令是将立即数左移 12 位，然后保存到 rd 的寄存器中。实现方案很简单。由于不需要读取 rs1 和 rs2，所以两者的读使能都是 0。在这我使用了个小技巧，先看 op2 的赋值语句：

```
//读取寄存器2的数据
always @ (*) begin
    if (rst == `RstEnable) begin
        op2 <= `ZeroWord;
    end else begin
        if((ex_wreg == `WriteEnable) && (ex_waddr == reg2_addr) && (reg2_read == `ReadEnable)) begin
            op2 <= ex_wdata;
        end else if((mem_wreg == `WriteEnable) && (mem_waddr == reg2_addr) && (reg2_read == `ReadEnable))begin
            op2 <= mem_wdata;
        end else if (reg2_read == `ReadEnable) begin
            op2 <= reg2_data;
        end else begin
            op2 <= imm;
        end
    end
end
```

op2 首先判断是否有数据前推的要处理，在这肯定没有，因为数据前推是在需要读取寄存器的值的时候才进行，这里不读取寄存器。然后如果寄存器的第二个读使能位 1，就将对于的地址的寄存器内的值赋给 op2，否则直接将立即数赋给 op2，在这就利用这个 imm，将待保存的数据放在 imm 中，下个时钟周期会直接通过 op2 将保存的数据传到执行阶段，然后再执行阶段执行相应的逻辑就可以了。在后面的一些指令译码也用到了该特性，在这同意解释，后面就不进行解释了。代码如下：

```
`LUI_OP: begin
    wreg <= `WriteEnable;
    imm <= { inst[31:12], {12{1'b0}} };
    waddr <= rd;
    alu_op <= `LUI;
end
```

AUIPC 指令：

该指令与 LUI 指令类似，不过保存的不是立即数，而是以立即数作为偏移量的 pc 的值。逻辑与 LUI 指令相同。

```
`AUIPC_OP: begin
    wreg <= `WriteEnable;
    imm <= { inst[31:12], {12{1'b0}} } + pc;
    waddr <= rd;
    alu_op <= `LUI;
end
```

S 类型指令：

S 类型指令将 rs2 寄存器的值存放到以 rs1 寄存器为基址的，imm 为偏移量的 ram 中。需要读取两个寄存器的值，然后计算出保存的地址，放到 ram_addr 中传递下去，然后在执行阶段根据 fun3 子类型进一步判断是保存的位数是 8 位还是 16 位，进行截取。代码如下：

```
`S_OP: begin
    case (fun3)
        3'b000: alu_op <= `SB;
        3'b001: alu_op <= `SH;
        3'b010: alu_op <= `SW;
    endcase
    reg1_read <= `ReadEnable;
    reg2_read <= `ReadEnable;
    ram_wreg <= `WriteEnable;
    imm <= { {21{inst[31]}}, inst[30:25], inst[11:7] };
    if((ex_wreg == `WriteEnable) && (ex_waddr == reg1_addr)) begin
        num1 <= ex_wdata;
    end else if((mem_wreg == `WriteEnable) && (mem_waddr == reg1_addr)) begin
        num1 <= mem_wdata;
    end else begin
        num1 <= reg1_data;
    end
    ram_addr <= num1 + imm;
end
```

JAL 指令：

这是转移指令，目标地址是以 pc 为基址，imm 为偏移量。

```
`JAL_OP: begin
    wreg <= `WriteEnable;
    branch_flag_o <= `Branch;
    waddr <= rd;
    imm <= pc + 32'd4;
    next_inst_in_delay <= 1'b1;
    branch_target_address_o <= pc + { {12{inst[31]}}, inst[19:12], inst[20], inst[30:21], 1'b0 };
    alu_op <= `JAL;
end
```

JALR 指令：

与 JAL 指令类似，不同点在于基址为 rs1 寄存器的值，由于作用相同，所以操作符与 JAL 设置一样的，节省一个类型。

```
`JALR_OP: begin
    wreg <= `WriteEnable;
    branch_flag_o <= `Branch;
    waddr <= rd;
    reg1_read <= `ReadEnable;
    imm <= pc + 32'd4;
    next_inst_in_delay <= 1'b1;
    if((ex_wreg == `WriteEnable) && (ex_waddr == reg1_addr)) begin
        num1 <= ex_wdata;
    end else if((mem_wreg == `WriteEnable) && (mem_waddr == reg1_addr)) begin
        num1 <= mem_wdata;
    end else begin
        num1 <= reg1_data;
    end
    branch_target_address_o <= num1 + { {21{inst[31]}}, inst[30:20] };
    alu_op <= `JAL;
end
```

LOAD 指令：

LOAD 指令加载 ram 中的值到寄存器中，ram 地址为 num1 寄存器中的值加 imm 立即数。

```
`LOAD_OP: begin
    wreg <= `WriteEnable;
    waddr <= rd;
    ld_type <= fun3;
    reg1_read <= `ReadEnable;
    if((ex_wreg == `WriteEnable) && (ex_waddr == reg1_addr)) begin
        num1 <= ex_wdata;
    end else if((mem_wreg == `WriteEnable) && (mem_waddr == reg1_addr)) begin
        num1 <= mem_wdata;
    end else begin
        num1 <= reg1_data;
    end
    read_addr <= num1 + { {21{inst[31]}}, inst[30:20] };
end
default: begin
end
```

寄存器 1 的读取：

```
//读取寄存器1的数据
always @ (*) begin
    if (rst == `RstEnable) begin
        op1 <= `ZeroWord;
    end else begin
        if((ex_wreg == `WriteEnable) && (ex_waddr == reg1_addr) && (reg1_read == `ReadEnable)) begin
            op1 <= ex_wdata;
        end else if((mem_wreg == `WriteEnable) && (mem_waddr == reg1_addr) && (reg1_read == `ReadEnable))begin
            op1 <= mem_wdata;
        end else if (reg1_read == `ReadEnable) begin
            op1 <= reg1_data;
        end else begin

        end
    end
end
```

ex.v			
输入/输出	宽度	信号名	说明
input	[0:0]	rst	复位信号
input	[3:0]	alu_op	操作符
input	[0:0]	wreg	寄存器写使能
input	[4:0]	waddr	寄存器写地址
input	[31:0]	op1	操作数 1
input	[31:0]	op2	操作数 2
input	[0:0]	ram_wreg	ram 写使能
input	[31:0]	ram_waddr	ram 写地址
input	[2:0]	ld_type	load 指令类型
input	[31:0]	read_addr	ram 读地址
output	[4:0]	waddr_o	寄存器写地址
output	[31:0]	wdata_o	寄存器写数据
output	[0:0]	wreg_o	寄存器写使能
output	[0:0]	ram_wreg_o	ram 写使能
output	[31:0]	ram_waddr_o	ram 写地址
output	[31:0]	ram_wdata_o	ram 写数据
output	[2:0]	ld_type_o	load 指令类型
output	[31:0]	read_addr_o	ram 读取地址

该模块是运算模块，进行各种指令的运算，需要注意的是进行有符号运算的时候使用了$signed，可以直接进行有符号拓展。核心代码：

```
//直接计算结果
always@(*) begin
    if (rst == `RstEnable) begin
        AluOut <= `ZeroWord;
    end
    else begin
        case (alu_op)
            `ADD: AluOut = op1 + op2;
            `SUB: AluOut = op1 - op2;
            `AND: AluOut = op1 & op2;
            `OR : AluOut = op1 | op2;
            `XOR: AluOut = op1 ^ op2;
            `SLT: AluOut = ($signed(op1) < $signed(op2)) ? 32'b1 : 32'b0;
            `SLTU: AluOut = (op1 < op2) ? 32'b1 : 32'b0;
            `SLL: AluOut = op1 << op2[4:0];
            `SRL: AluOut = op1 >> op2[4:0];
            `SRA: AluOut = $signed(op1) >>> op2[4:0];
            `LUI: AluOut = op2;
            `SH: RamData = op2[15:0];
            `SB: RamData = op2[7:0];
            `SW: RamData = op2;
            `JAL: AluOut = op2;
            default: AluOut = 32'hxxxxxxxx;
        endcase
    end
end
```

regfile.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟信号
input	[0:0]	rst	复位信号
input	[4:0]	raddr1	读端口 1 地址
input	[4:0]	raddr2	读端口 2 地址
input	[0:0]	re1	读端口 1 使能
input	[0:0]	re2	读端口 2 使能
input	[31:0]	wdata	写数据
input	[4:0]	waddr	写地址
input	[0:0]	we	写使能
output	[31:0]	rdata1	读端口 1 数据
output	[31:0]	rdata2	读端口 2 数据

该模块是寄存器模块，负责读取和写回寄存器的数据。在该模块处理了数据相关的冲突，如果需要读取的代码是刚要写入的代码，那么就直接将待写的数据送到输出。代码如下：

```
always @ (*) begin
    if(rst == `RstEnable) begin
        rdata1 <= `ZeroWord;
    end else if(raddr1 == 5'b0) begin
        rdata1 <= `ZeroWord;
    end else if((raddr1 == waddr) && (we == `WriteEnable)
                && (re1 == `ReadEnable)) begin
        rdata1 <= wdata;
    end else if(re1 == `ReadEnable) begin
        rdata1 <= regs[raddr1];
    end else begin
        rdata1 <= `ZeroWord;
    end
end

always @ (*) begin
    if(rst == `RstEnable) begin
          rdata2 <= `ZeroWord;
  end else if(raddr2 == 5'b0) begin
        rdata2 <= `ZeroWord;
  end else if((raddr2 == waddr) && (we == `WriteEnable)
                && (re2 == `ReadEnable)) begin
      rdata2 <= wdata;
  end else if(re2 == `ReadEnable) begin
      rdata2 <= regs[raddr2];
  end else begin
      rdata2 <= `ZeroWord;
  end
end
```

mem.v			
输入/输出	宽度	信号名	说明
input	[0:0]	rst	复位信号
input	[4:0]	waddr	寄存器写地址
input	[0:0]	wreg	寄存器写使能
input	[31:0]	wdata	寄存器写数据
input	[2:0]	ld_type	load 指令子类型
input	[31:0]	read_addr	ram 读取地址
input	[31:0]	rdata	ram 读取到的数据
input	[31:0]	ram_wdata_i	写回 ram 的数据
input	[31:0]	ram_waddr_i	写回 ram 地址
input	[0:0]	ram_wreg_i	ram 写使能
output	[31:0]	waddr_o	寄存器写地址
output	[0:0]	wreg_o	寄存器写使能
output	[31:0]	wdata_o	寄存器写数据
output	[31:0]	ram_waddr	写回 ram 的地址
output	[31:0]	ram_wdata	ram 写数据
output	[0:0]	ram_wreg	ram 写使能
output	[31:0]	raddr	ram 读取地址

该模块是存储模块，负责将数据写到 ram 中，或者从 ram 中读取数据保存到寄存器中。如果是读取数据 load 指令的话，需要判断子类型 ld_type，分有符号和无符号，字节、半字和字读取。核心代码如下：

```
always @ (*) begin
    if(rst == `RstEnable) begin
        waddr_o <= `NOPRegAddr;
        wreg_o <= `WriteDisable;
        wdata_o <= `ZeroWord;

        ram_waddr <= `ZeroWord;
        ram_wdata <= `ZeroWord;
        ram_wreg <= `WriteDisable;
    end else begin
        raddr <= read_addr;
        waddr_o <= waddr;
        wreg_o <= wreg;
        case (ld_type)
            `LB: wdata_o <= { {25{rdata[7]}},rdata[6:0] };
            `LW: wdata_o <= rdata;
            `LH: wdata_o <= { {17{rdata[15]}},rdata[14:0] };
            `LBU: wdata_o <= { 24'h0,rdata[7:0] };
            `LHU: wdata_o <= { 16'h0,rdata[15:0] };
            default: wdata_o <= wdata;
        endcase
        ram_waddr <= ram_waddr_i;
        ram_wdata <= ram_wdata_i;
        ram_wreg <= ram_wreg_i;
    end    //if
end      //always
```

dataram.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟信号
input	[0:0]	rst	复位信号
input	[0:0]	we	写使能
input	[31:0]	wdata	写数据
input	[31:0]	waddr	写地址
input	[31:0]	raddr	读地址
output	[31:0]	rdata	读数据

该模块是内存模块，内存数据的读取和写入。这个模块需要注意的是，由于内存只分配了 4096 个字（12 位），因此如果读取的地址或写入的地址高 20 位不全为 0 的话，就是无效地址不进行任何处理。代码如下：

```
reg [`RegBus] ram [0:4095];
wire waddr_valid = ( waddr[31:12]==20'h0 );
wire raddr_valid = ( raddr[31:12]==20'h0 );

initial begin
    ram[1] = 32'h00005678;
end

always @ (posedge clk) begin
    if ((rst == `RstDisable) && we == `WriteEnable && waddr_valid) begin
        ram[waddr] <= wdata;
    end
end


always @ (*) begin
    if (rst == `RstEnable || !raddr_valid) begin
        rdata <= `ZeroWord;
    end else begin
        rdata <= ram[raddr];
    end
end
```

inst_rom.v			
输入/输出	宽度	信号名	说明
input	[0:0]	ce	使能信号
input	[31:0]	addr	读取地址
output	[31:0]	inst	输出的指令值

该模块是指令存储器，通过输入的指令地址输出对应指令。代码如下：

```
module inst_rom(

    input wire ce,
    input wire[`InstAddrBus] addr,
    output reg[`InstBus] inst

);

    reg[`InstBus]  inst_mem[0:`InstMemNum-1];

    initial $readmemb ( "L:/documents/vivado-workspace/cpu/cpu.srcs/sources_1/new/inst_rom.data", inst_mem);

    always @ (*) begin
        if (ce == `ChipDisable) begin
            inst <= `ZeroWord;
      end else begin
            inst <= inst_mem[addr[`InstMemNumLog2+1:2]];
        end
    end

endmodule
```

ctrl.v			
输入/输出	宽度	信号名	说明
input	[31:0]	data	指令的值
input	[0:0]	change	开关信号
led	[15:0]	led	输出的 led 信号

该模块是 FPGA 的连接控制模块，我连接到 FPGA 的是指令的值，连接到的器件是 16 个 led 灯，但是指令是 32 位的，所以还有个开关连接到 change 信号上，控制高低电平，高电平就显示高 16 位，低电平就显示低 16 位，输出的 led 就是 FPGA 上展示的高低 16 位指令值。代码如下：

```
module ctrl(
    input wire[`RegBus] data,
    input wire change,
    output reg[15:0] led
);

    always @ (*) begin
        if (change == 1'b0) begin
            led <= data[15:0];
        end else begin
            led <= data[31:16];
        end
    end
endmodule
```

riscv.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟信号
input	[0:0]	rst	复位信号
input	[31:0]	inst	输入的指令值
input	[31:0]	rdata	输入的 ram 数据
output	[0:0]	ce	inst_rom 使能信号
output	[31:0]	inst_addr	指令地址
output	[31:0]	ram_wdata	ram 写数据
output	[31:0]	ram_waddr	ram 写地址
output	[0:0]	ram_wreg	ram 写使能
output	[31:0]	raddr	ram 读取地址

该模块是 risc-v cpu 的综合模块，实例化各个模块并连接各个模块的线。代码略。

riscv_min_sopc			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟信号
input	[0:0]	rst	复位信号
input	[0:0]	change	高低电平
output	[15:0]	led	输出的 led 信号

该模块是整合模块，连接 CPU 和指令寄存器和内存 ram。输入和输出都连接到 FPGA 上来测试。clk 信号连接一个按钮，rst 信号连接一个开关，change 信号连接一个开关，led 信号连接 16 个 led 灯。代码如下：

```
module riscv_min_sopc(
    input wire clk,
    input wire rst,
    input wire change,
    output wire[15:0] led
);

    wire ce;

    //指令，传到crtl模块进行输出，也传到cpu模块进行执行
    wire[`InstBus] inst;
    wire[`InstAddrBus] inst_addr;

    wire ram_wreg;
    wire[`RegBus] ram_waddr;
    wire[`RegBus] rdata;
    wire[`RegBus] ram_wdata;
    wire[`RegBus] raddr;

    riscv riscv0(
        .clk(clk), .rst(rst),
        .inst(inst), .inst_addr(inst_addr), .ce(ce),
        .rdata(rdata) ,.ram_waddr(ram_waddr), .ram_wdata(ram_wdata), .ram_wreg(ram_wreg), .raddr(raddr)
    );

    inst_rom inst_rom0(
        .ce(ce), .addr(inst_addr), .inst(inst)
    );

    //控制输出模块
    ctrl ctrl0(
        .clk(clk), .rst(rst),
        .data(inst), .change(change), .led(led)
    );

    dataram dataram0(
        .rst(rst), .clk(clk), .we(ram_wreg), .wdata(ram_wdata), .waddr(ram_waddr), .raddr(raddr), .rdata(rdata)
    );

endmodule
```

四、实验结果

仿真

我设计了 12 条指令对该 CPU 进行测试，具体测试指令如下：

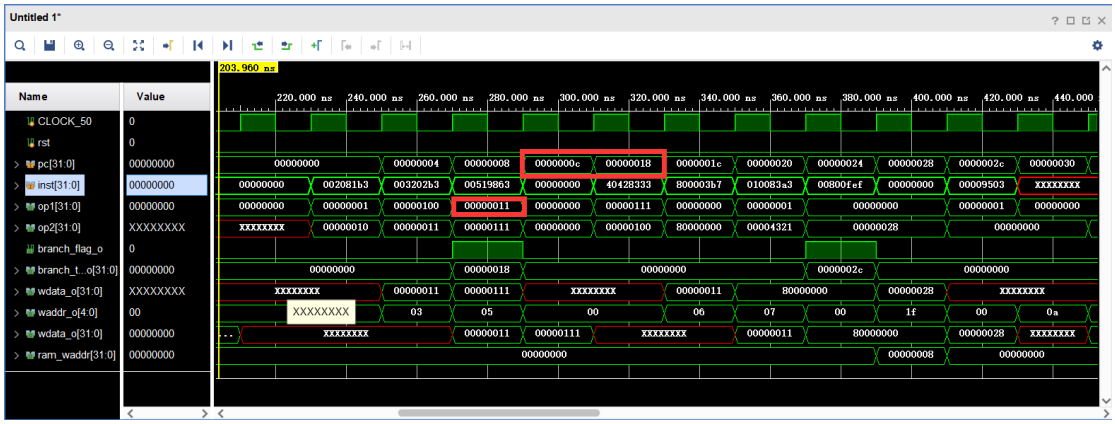
```
//////////////////      流水测试      //////////////////////
00000000001000001000000110110011     rs1+rs2->reg3    (11)
00000000001100100000001010110011     rs3+rs4->reg5    (111)
00000000000100010001100001100011     BNE rs3 rs5 16
01000000010000101000001100110011     jump
01000000010000101000001100110011     jump
01000000010000101000001100110011     jump
01000000010000101000001100110011     rs5-rs4->reg6    (011)
10000000000000000000001110110111     lui: imm->reg7
00000000100000001000001110100011     SB: reg2[7:0] -> ram[reg1+imm]
00000001000000000000111111101111     jal: pc+4->reg31, jump->pc+8
00000001000000000000111111101111     jump
00000000000000001001010100000011     ld ram[1]->reg[10]
```

寄存器的初始化如下：

```
initial begin
    regs[1] = 32'h0001;
    regs[2] = 32'h0010;
    regs[3] = 32'h0000;
    regs[4] = 32'h0100;
    regs[5] = 32'h0000;
    regs[16] = 32'h4321;
end
```

该测试代码涵盖了所有类型的指令，并且测试了流水的数据冲突，以及转移指令。

仿真结果：



第一条指令将 reg[1]+reg[2]送到 reg[3]存储，第二条指令将 reg[3]+reg[4]送到 reg[5]中存储，由于原本 reg[3]放的是 0，在图中可以看到取得的 reg[3]是 00000011，而不是 0，说明数据相关的冲突得到了解决。

第三条指令是转移指令，比较 reg[3]和 reg[5]中的值，如果不相等则转移到以 pc 位基址，imm 为偏移量的地址去，这里 imm 是 16，此时 reg[3]为 00000011，而 reg5 为 0，所以肯定会转移，在图中圈起来的地方可以看到，地址有 0000000c 变成了 00000018，而 0000000c 下方的指令值 inst 变成了 0，说明转移成功。

中间几条指令是保存指令就不解释，等下看 regfile 的值就能知道是否保存。下面看倒数第三条指令，这条指令是 jal 指令，跳转到 pc+imm 处，并将 pc+4 保存到寄存器中去，一般返回地址都是保存在第 31 个，所以我就保存到第 31 寄存器中。在图中可以看到，在 pc 位 00000028 的位置下方，inst 的值为 0，说明跳转成功，而跳转是加 8，所以 pc 的值看起来并没有变换。

接下来执行最后一条指令 load 指令，将 ram[1]中的值存储到 reg[10]中，存储的是半字。

```
initial begin
    ram[1] = 32'h00005678;
end
```

我们将 ram[1]初始化为上述的值，半字也就是 5678，reg[10]中的内容将会是 5678。

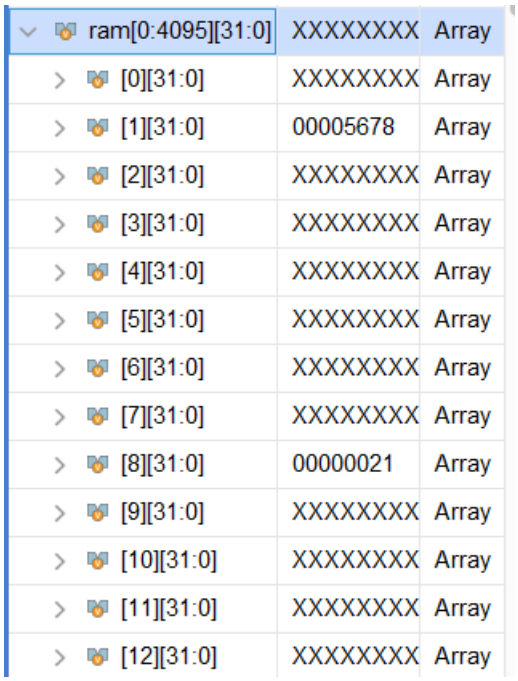
下面来看 regfile 中的值：

regs[0:31][31:0]	XXXXXXXX	Array	[12][31:0]	XXXXXXXX	Array
[0][31:0]	XXXXXXXX	Array	[13][31:0]	XXXXXXXX	Array
[1][31:0]	00000001	Array	[14][31:0]	XXXXXXXX	Array
[2][31:0]	00000010	Array	[15][31:0]	XXXXXXXX	Array
[3][31:0]	00000011	Array	[16][31:0]	00004321	Array
[4][31:0]	00000100	Array	[17][31:0]	XXXXXXXX	Array
[5][31:0]	00000111	Array	[18][31:0]	XXXXXXXX	Array
[6][31:0]	00000011	Array	[19][31:0]	XXXXXXXX	Array
[7][31:0]	80000000	Array	[20][31:0]	XXXXXXXX	Array
[8][31:0]	XXXXXXXX	Array	[21][31:0]	XXXXXXXX	Array
[9][31:0]	XXXXXXXX	Array	[22][31:0]	XXXXXXXX	Array
[10][31:0]	00005678	Array	[23][31:0]	XXXXXXXX	Array
[11][31:0]	XXXXXXXX	Array	[24][31:0]	XXXXXXXX	Array

[26][31:0]	XXXXXXXX	Array
[27][31:0]	XXXXXXXX	Array
[28][31:0]	XXXXXXXX	Array
[29][31:0]	XXXXXXXX	Array
[30][31:0]	XXXXXXXX	Array
[31][31:0]	00000028	Array

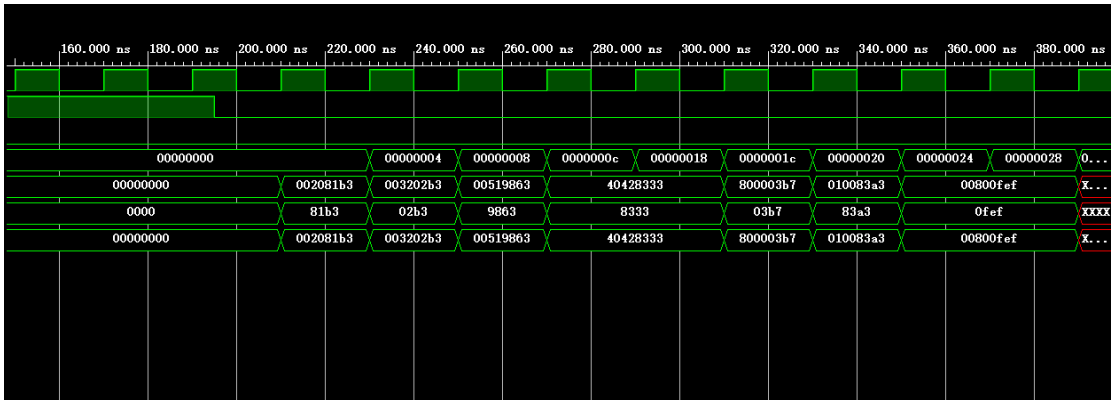
可以看到 reg[10]的值是 5678，reg[31]的值是 28，也就是 24+4，也是正确的。

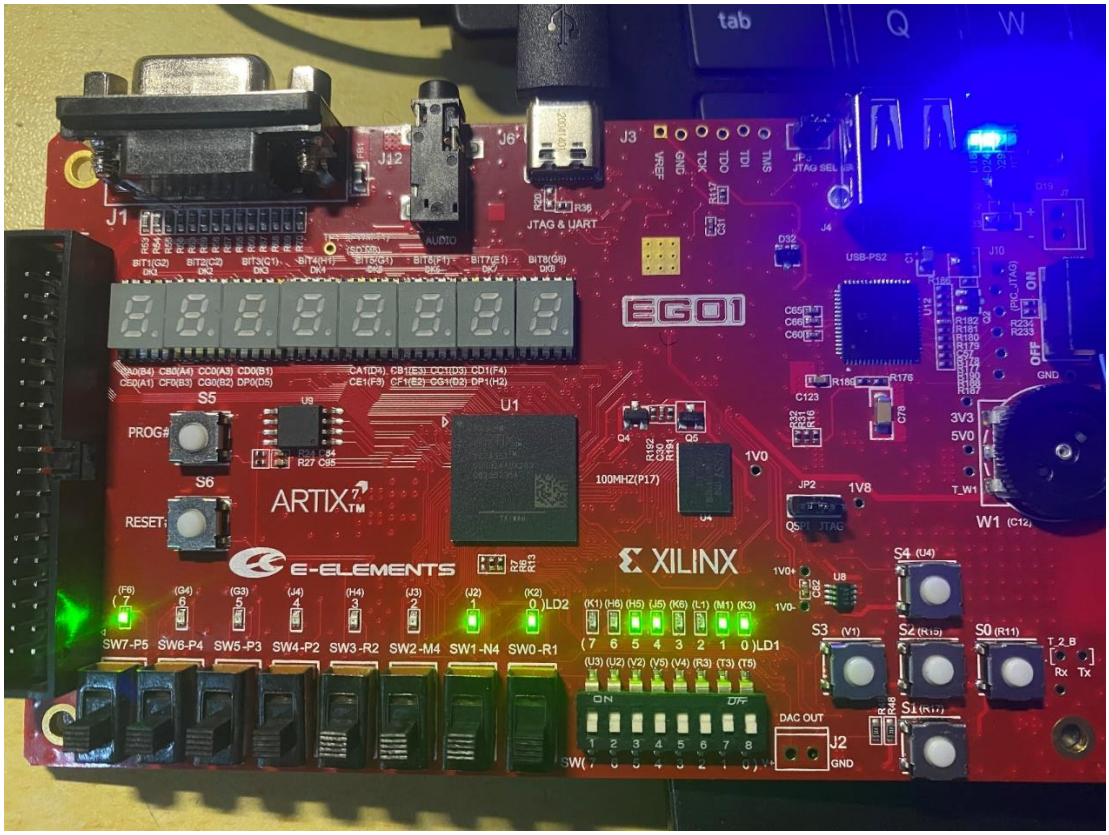
最后看一下 ram 中的数据，SB 将 reg[16]保存到 ram 中，类型是字节，

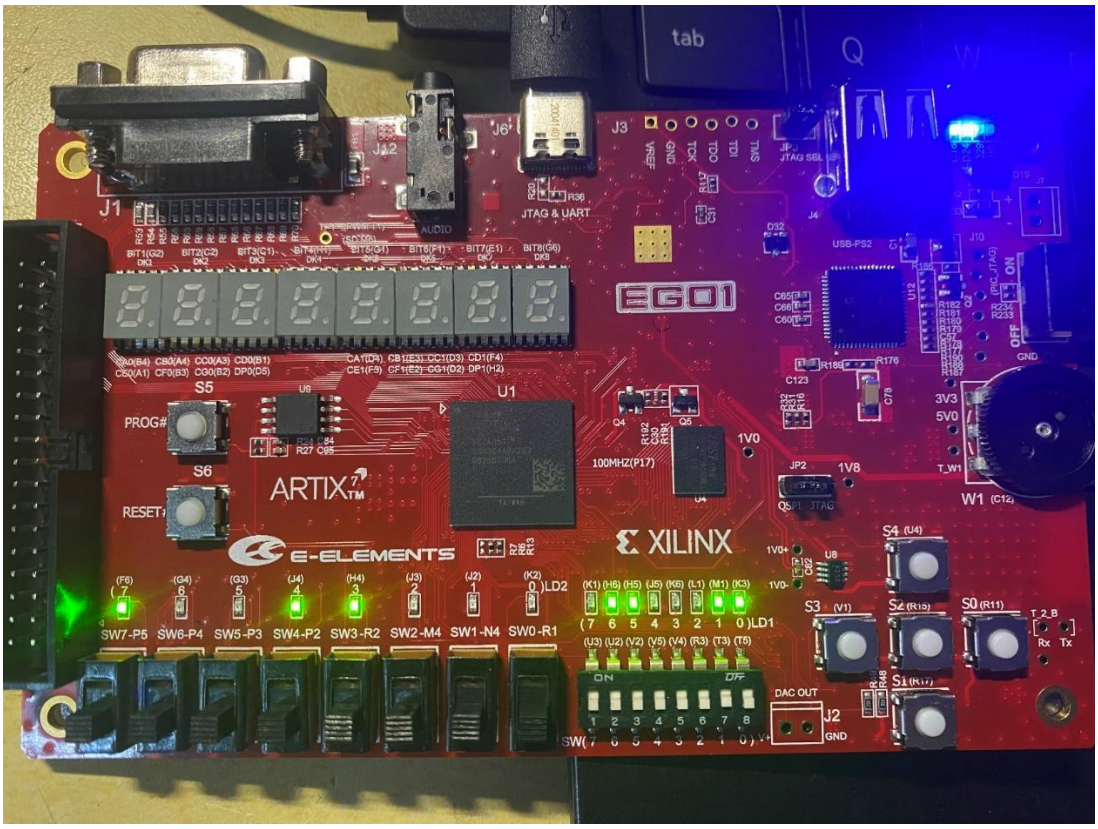


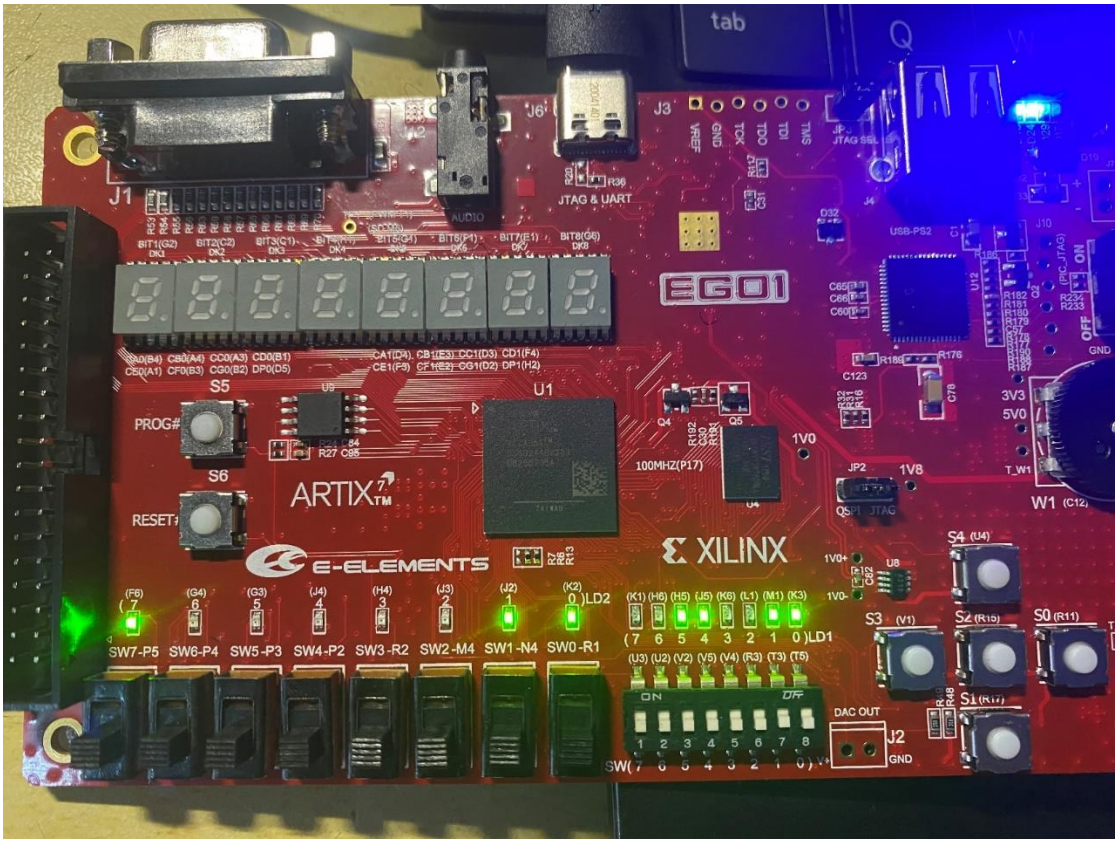
看到 ram[8]是 21，结果正确。

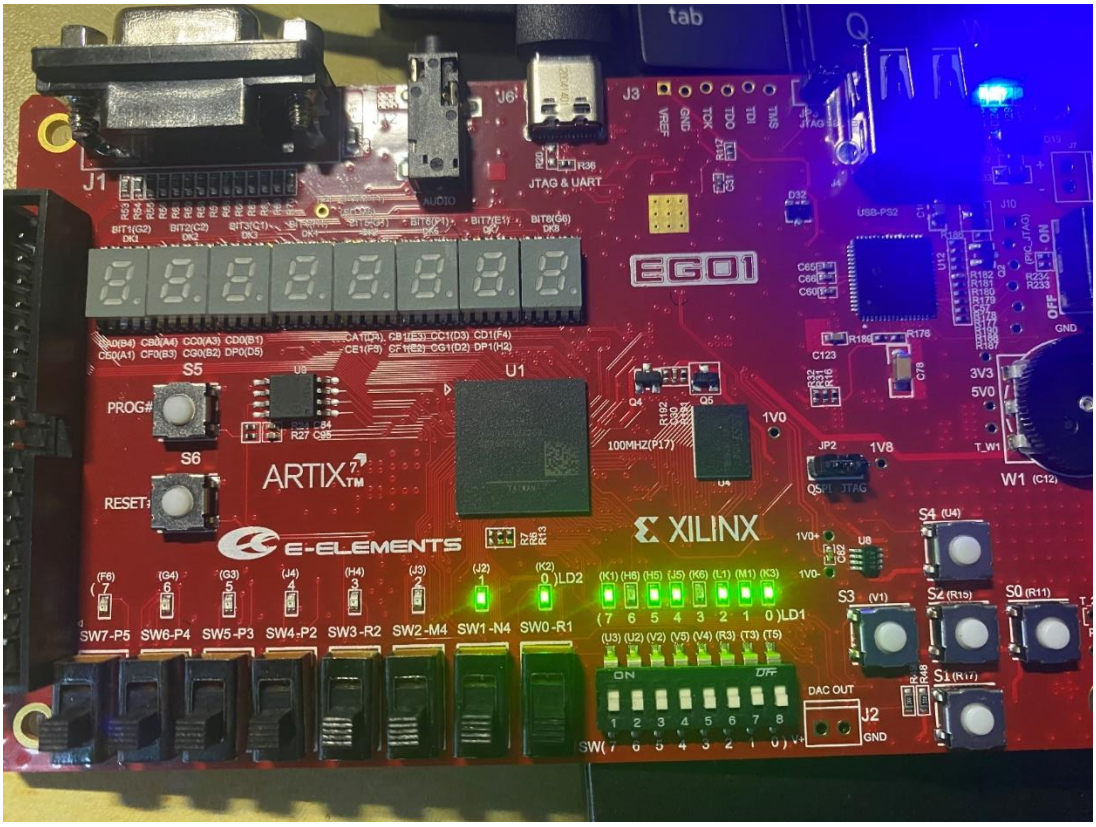
FPGA 验证：

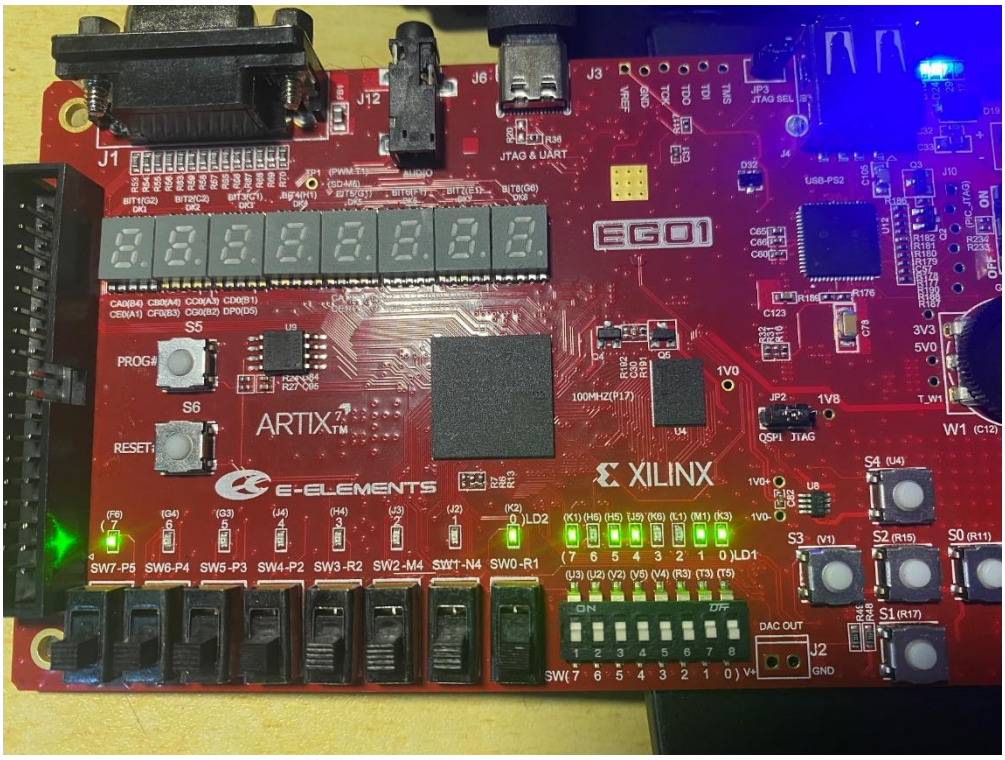

160.000 ns
180.000 ns
200.000 ns
220.000 ns
240.000 ns
260.000 ns
280.000 ns
300.000 ns
320.000 ns
340.000 ns
360.000 ns
380.000 ns
00000000
00000004
00000008
0000000c
00000018
0000001c
00000020
00000024
00000028
00000000
002081b3
003202b3
00519863
40428333
800003b7
010083a3
00800fef
0000
81b3
02b3
9863
8333
03b7
83a3
0fef
00000000
002081b3
003202b3
00519863
40428333
800003b7
010083a3
00800fef


EGO1
ARTIX7
XILINX
E-ELEMENTS
JTAG & UART
100MHZ(P17)
PROG#
RESET
SW7-P5
SW6-P4
SW5-P3
SW4-P2
SW3-R2
SW2-M4
SW1-N4
SW0-R1
DAC OUT


EGO1
ARTIX7
E-ELEMENTS
XILINX
JTAG & UART
100MHZ(P17)
PROG#
RESET
S5
S6
S4 (U4)
S3 (V1)
S2 (R15)
S0 (R11)
S1 (R17)
SW7-P5
SW6-P4
SW5-P3
SW4-P2
SW3-R2
SW2-M4
SW1-N4
SW0-R1
LD2
LD1
DAC OUT
J2
GND
3V3
5V0
1V0
1V8
W1 (C12)
JTAG SEL
QSPI JTAG
USB-PS2
Rx
Tx


EG01
ARTIX 7
E-ELEMENTS
XILINX
JTAG & UART
AUDIO
USB-PS2
PROG#
RESET
S5
S6
U1
100MHZ(P17)
1V0
1V8
3V3
5V0
W1
JP2
JTAG
SW7-P5
SW6-P4
SW5-P3
SW4-P2
SW3-R2
SW2-M4
SW1-N4
SW0-R1
LD2
LD1
DAC OUT
J2
GND
S4 (U4)
S3 (V1)
S2 (R15)
S0 (R11)
S1 (R17)


EG01
ARTIX7
E-ELEMENTS
XILINX
PROG#
RESET
S5
S6
U1
100MHZ(P17)
JTAG & UART
SW7-P5 SW6-P4 SW5-P3 SW4-P2 SW3-R2 SW2-M4 SW1-N4 SW0-R1
LD2
LD1
DAC OUT
J1
J2
J3
GND
3V3
5V0
1V0
1V8


EGO1
ARTIX7
XILINX
E-ELEMENTS
JTAG & UART
PROG#
RESET
SW7-P5
SW6-P4
SW5-P3
SW4-P2
SW3-R2
SW2-M4
SW1-N4
SW0-R1
LD2
LD1
DAC OUT
3V3
5V0
1V0
1V8
W1
S0
S1
S2
S3
S4
S5
S6

五、实验总结

这个 CPU 设计对我的收获非常巨大，不仅让我重温并更加深刻理解了计算机组成原理的知识，还再一次熟悉掌握了 Verilog 硬件描述语言，并且还学习了 FPGA 的使用。

刚开始的时候我完全不知道该如何下手写这个 CPU，觉得那个自己动手写 CPU 也太多了，肯定看不完看不懂，但是被逼无奈，不看更不会写。于是我就花了整整三天时间把这本书看到了乘除模块的编写，边看边敲代码理解。到了乘除模块不太看得懂也不太愿意看了。于是就停下打算就去验收了算了。但是看到很多人都是这样验收的，觉得这样就算我不看这个书，直接把代码复制过去验收，也能验收啊，那我不是白看了书吗？所以我又觉得不甘心，这样去验收肯定不会有很好的成绩。

那天是星期三，看到他们都去验收了，我就很心急，当天晚上就开始自己写一个 CPU，然后网上就去找 RISC-V 的指令集，开始自己写一个 CPU。然后就写出了这个 CPU。花了我一天一夜的时间，从星期三晚上九点开始写，写到星期四晚上十二点，中间只睡了五个小时，上了一节课。之后验收前还花了一晚上时间增加了几条指令。当自己写完 CPU 的时候，心里是很高兴的，也是很自豪的，这是我自己写出来的 CPU！虽然流水线与那本教程上的是一样的，但是内部的逻辑其实都是我自己的，并且很多教程上麻烦的代码都被我自己给改进改的简单了。

总的来说，这个 CPU 花费了我大量的时间，也让我学到了大量的知识。当初学习计算机组成原理的时候完全不懂数据前推是什么概念，只知道处理冲突有这个方法，考试的时候写这个就对了，写完这个 CPU 后就理解了数据前推的作用和原理。在我设计的这个 CPU 中其实还有一点不完整，就是 LOAD 指令的数据冲突作用，没有考虑进去，因为当时写的时候没想到会有冲突。不过我也想到了一种解决方案，下面描述一下该解决方案。由于 load 指令会将数据加载到寄存器中，而回写到寄存器还需要一个时钟周期，所以如果下一个周期有指令读取写的数据的话，是不能正确读取的。这时候可以做如下修改：在译码阶段可以直接将 load 指令需要的参数直接送到 mem 模块（记得一个词叫旁路技术，不知道这是不是），然后从 mem 模块接一根数据线到译码模块，作用就是数据前推，设置一个标志，如果需要数据前推就直接使用 mem 传来的数据，而不使用读取到的数据。我认为这个解决方案是可行的。

最后，这个报告也花了我一整天的时间，画那个电路图找了很多工具，最后是用 draw.io 绘制的。