中国科学技术大学计算机学院

《数字电路实验》报告

实验题目：综合实验-简易 cpu 的实现

学生姓名：舒文炫

学生学号：PB18000029

完成日期：12.23

计算机实验教学中心制

2020 年 09 月

【实验题目】

综合实验-简易 cpu 的实现

【实验目的】

- 熟练掌握前面实验中的所有知识点
- 熟悉几种常用通信接口的工作原理及使用
- 独立完成具有一定规模的功能电路设计

【实验环境】

- logisim
- vivado2020

【题目设计及原因】

本次实验我没有写实验文档里面提供的那几题，我是考虑自行尝试搭建一个简单的 cpu，因为我之前一直对 cpu 的具体实现比较感兴趣，对这一小块单元是如何能做到这么多的功能很好奇，但是由于之前没接触过关于底层硬件的课程，理解这些内容有些困难，但是现在学习了这门课，也逐渐对数字电路熟悉起来了，所以我想趁这个机会，尝试搭建一个简单的 cpu，一来这个单元综合性很高，可以很好的运用到前面所学的知识，达到巩固的目的，二来，也满足一下自己的好奇心，去深入探索计算机的内部工作原理

考虑到如果直接使用 vivado 写 verilog 代码可能调试起来比较困难，而且对于这种大电路，直接上手代码可能比较难有一个全局观，所以这次实验计划分两个步骤进行。

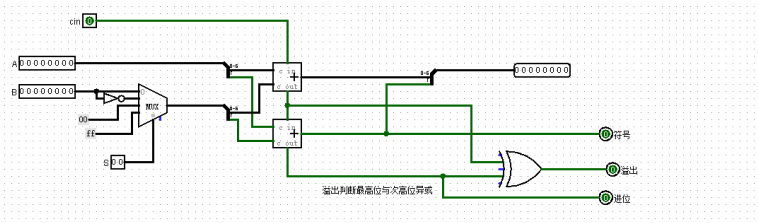
- 步骤一：在 logisim 上实现简易 cpu 的模型
- 步骤二：在这个模型的基础上，对照使用 vivado 实现

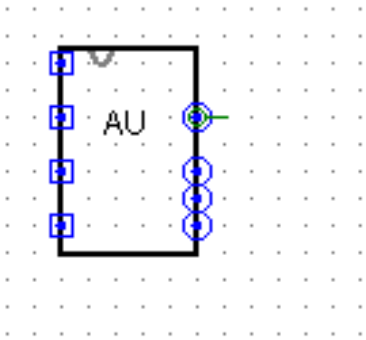
【实验过程】

logisim 实现

cpu 即中央处理器，其需要完成的功能有算数运算，逻辑运算，指令控制，数据通路连接，数据寄存等一系列功能，要搭建一块 cpu，也就需要一步步实现这些功能，最后将其组合成一个稳定的系统

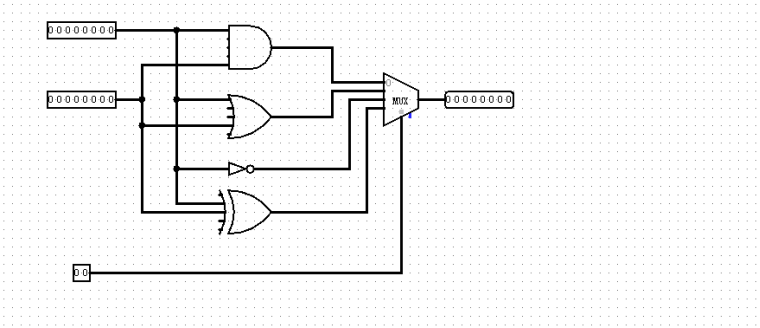
首先是算数运算单元 AU

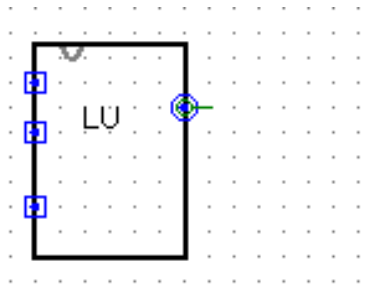




这一部分需要实现两个数的相加相减等运算，这里我所有的数据都是 8bit 的，考虑到溢出的情况，需要进行单独的判断，以防出现错误结果。这里我判断溢出，使用的原理是如果最高进位与次高进位不同，则发生了溢出，所以这里我把数据拆分成两部分，第一部分可以得到次高进位，第二部分得到最高进位，然后两个结果做异或即可得到是否溢出，输出信号符号表示运算结果的符号，也就是结果的最高位，进位就是运算的进位信息。这里通过四选一选择器实现运算单元运算功能选择，有相加，相减，不变，-1.

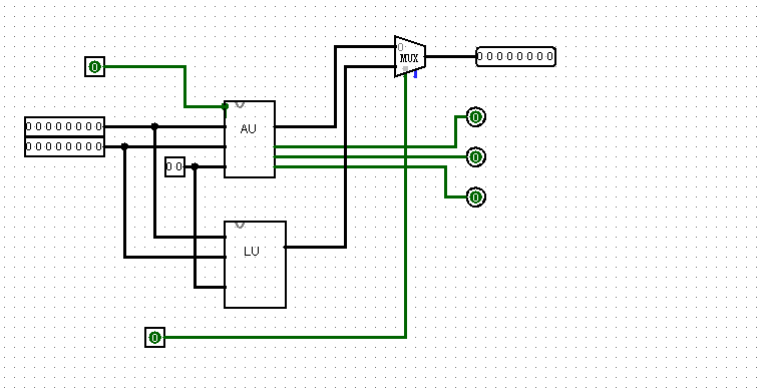
然后是逻辑运算单元 LU

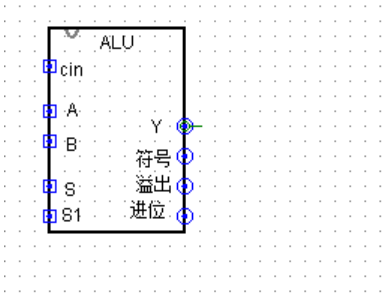




这部分实现两个数的逻辑运算，有与，或，非，异或这四种基本的逻辑运算，通过四选一选择器选择进行何种运算

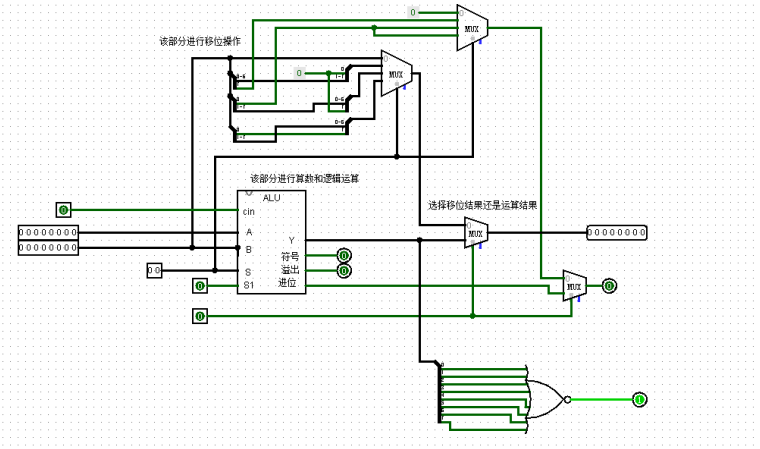
然后将这两个单元组合在一起，就是需要的运算单元 ALU

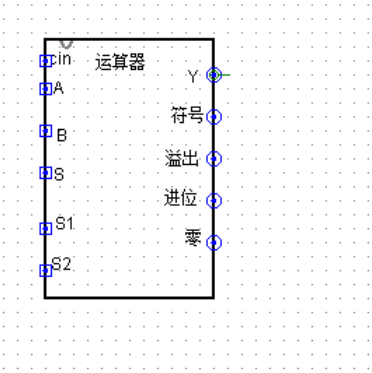




说是组合，但也不能就简简单单连在一起，这里需要注意到算术运算和逻辑运算并不是同时进行的，这一模块主要就是把这两种功能放在一起，需要根据实际情况进行选择是算术运算还是逻辑运算，这里 s1 就是做这个工作，s1=0 进行算术运算，s1=1 进行逻辑运算

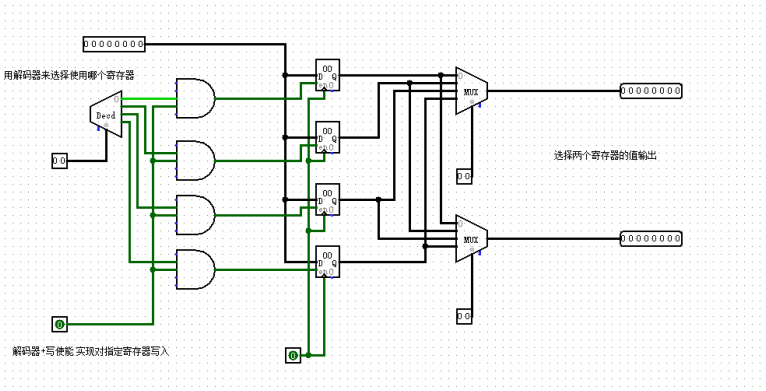
考虑到计算机里面还有移位的操作，这里还需要把这个操作也综合进来，这样的运算器基本就完整了，至于乘法器，除法器，实现起来过于复杂，但是我们知道这些都可以用加法和减法实现，所以对于我这个简易 cpu 而言已经足够了

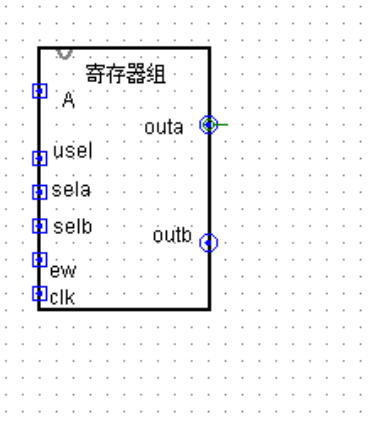




移位操作的思想很简单，使用两个分线器叉开一位对接就可以了，这里我实现了左移，右移，循环移位等操作，这里选择功能的信号复用了运算器选择功能信号，因为不可能同时去做两种运算，复用可以使电路简单一点。这里我考虑到后面可能需要做循环操作还增加了零这个信号，就是判断输出结果是否为零，这个信号可以作为循环终止条件，比较方便

下面是寄存器组 regs

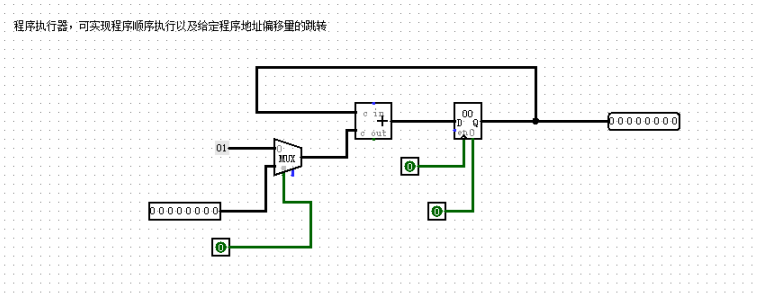


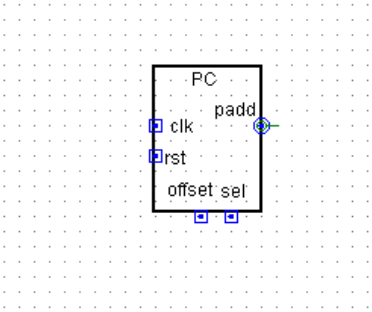


寄存器组是一个很重要的部分，可以对运算结果进行暂存，从 RAM 中读写数据，都需要有寄存器作为中间者。这里我就只实现了 4 个寄存器组成的寄存器组，实际上用更多的寄存器也没什么问题，但是就我这次实验而言，用 4 个就够了，再多就显得复杂了，而且原理都一样的。这里需要给出选择信号 usel，选择对哪个寄存器操作，然后是对寄存器的写入使能信号 ew，表示是否可以写入，时钟信号 clk，最后输出结果，也需要选择从哪个寄存器读出，这里对两个输出都分别要给一个 sela，selb，这样寄存器组就完成了

然后进入到控制器部分

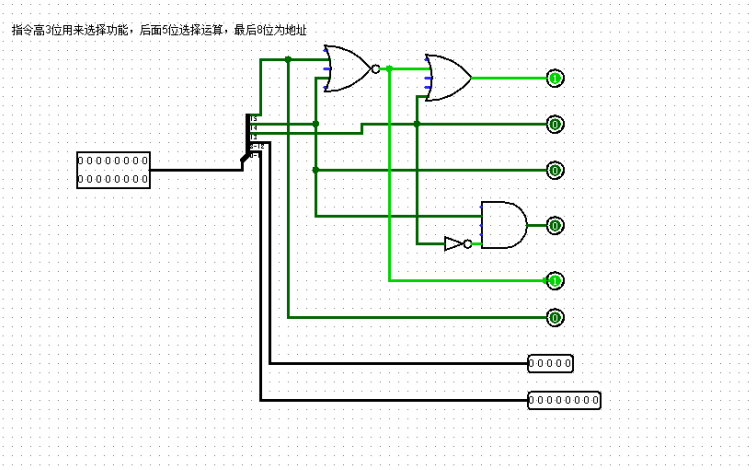
首先是程序计数器，计算机执行指令是一条一条执行的，执行完上一条指令后需要跳转到下一条指令的地址，然后才能接着执行，这也就是程序计数器的作用

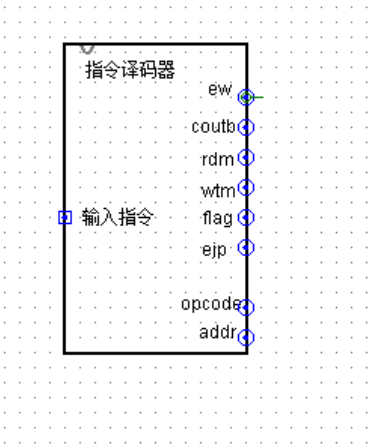




这一部分因为是计数器，之前实验里面也接触过，所以画出来比较简单，但是这里不单单是循环计数就够了，有时候下指令地址并不在紧接着的那个地址，所以这里如果需要发生跳转，还需要给一个跳转的偏移量，不发生跳转，就每隔一个时钟周期往后 +1 就可以了

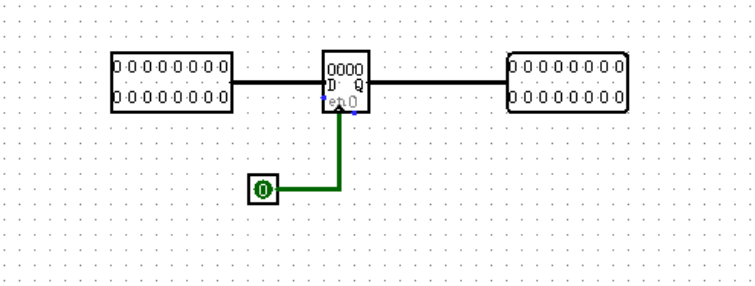
有了指令地址，那么现在需要考虑对于指令本体的设计，目前主流的是 mips 架构，不过对于本实验而言，我只是试着实现一些基本的功能，用不上那么强大的指令集，而且强大的指令集也意味着实现起来的复杂，所以这里就实现一套精简指令集，指令是 16 位的，下面是指令译码器

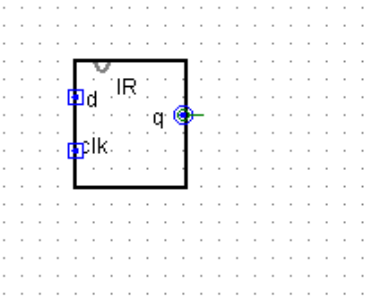




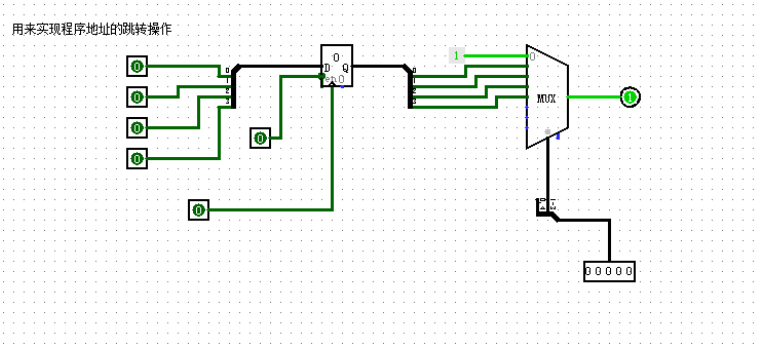
16 位的指令主要分为 3 个部分，前三位选择操作，后八位是地址，中间五位用来选择运算。这里面操作分为两个寄存器的操作，一个寄存器与常数的操作，读写内存操作，跳转操作。两个寄存器操作比如将寄存器 A 的值赋给寄存器 B，又或者是寄存器 A 的值与寄存器 B 相加，五种操作所需要的编码最少就是三位，所以这里用前三位选择操作。中间五位主要就是关于运算器部分，如何选择了。后八位的地址包括了寄存器地址选择，以及立即数，这个立即数就是用来直接进行常量操作的

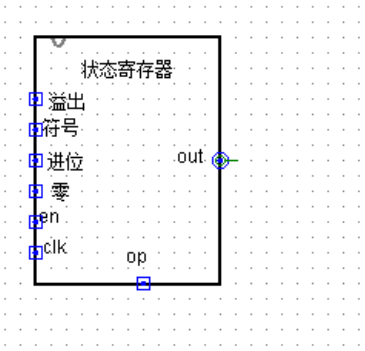
然后是指令寄存器，用来读取指令暂存备用，实际上也就是一个寄存器，我这里的指令设计成 16 位的



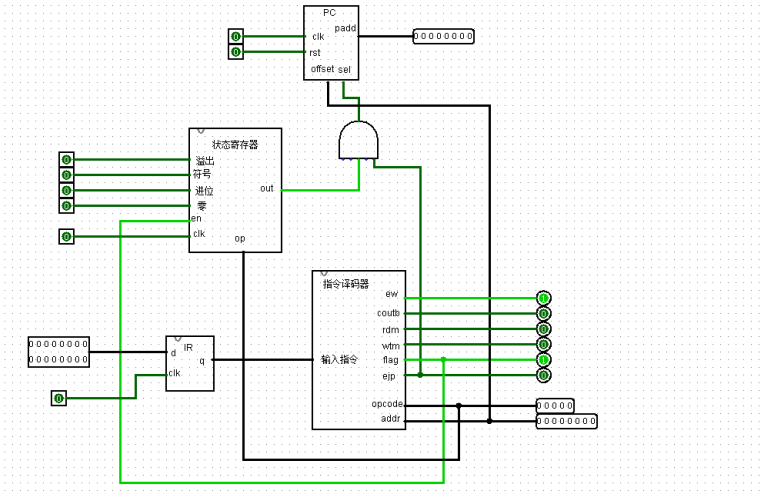


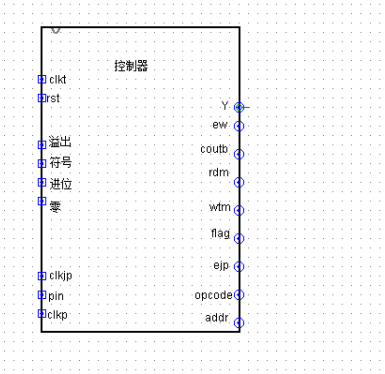
最后还需要状态寄存器，这一部分输入的是前面电路运算结果产生的各种状态比如溢出，零，进位，符号之类的，对于这些状态需要进行不同的操作。比如说递减一个数，以其变为零作为循环终止条件，这时如果有零的状态传达过来，就需要进行处理了，比如程序的地址需要发生跳转，为了使电路简洁，复用前面中间五位来进行特定状态的选择



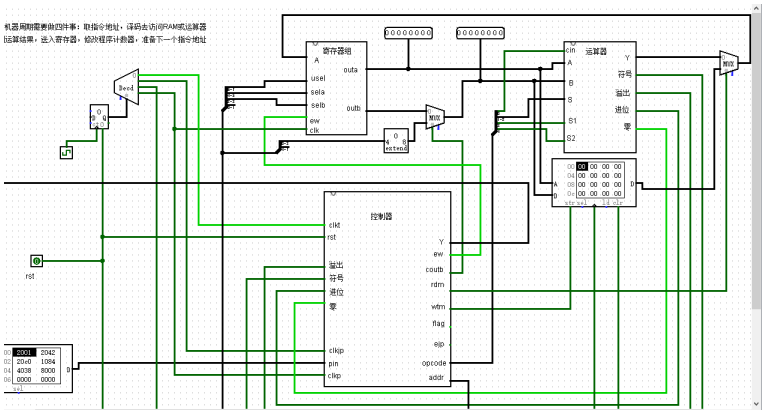


将上述模块组合起来，就是我们需要的控制器





最后加上数据通路，也就是把对应的接口连接起来，以及 ROM,RAM 就得到 cpu 了，这样就能运行一些简单的指令了。



这里面需要注意的是，一个机器周期需要依次完成四件事，这些必须按照顺序进行，不然无法产生正确结果，首先取出指令地址，经过译码器，然后按照译码结果，取出 RAM 中对应地址的值或者选择寄存器进行运算，然后运算结果保存，最后跳转到下一条指令。

verilog 实现

有了前面电路作为参考，再用 verilog 实现就稍微简单一些了

首先是 AU

```
module au(
input cin,
input [7:0]a,b,
input [1:0]s,
output [7:0] y,
output sign,ov,ca1
    );
    reg [7:0]rb;
    wire[7:0] ca;
    always@(s)
    begin
    case(s)
    2'b00:
    rb=b;
    2'b01:
    rb=~b;
    2'b10:
    rb=8'h00;
     2'b11:
    rb=8'hff;
    endcase
    end
```

```
add1 inst_1(.a(a[0]),.b(rb[0]),.cin(cin),.sum(y[0]),.cout(ca[0]));
add1 inst_2(.a(a[1]),.b(rb[1]),.cin(ca[0]),.sum(y[1]),.cout(ca[1]));
add1 inst_3(.a(a[2]),.b(rb[2]),.cin(ca[1]),.sum(y[2]),.cout(ca[2]));
add1 inst_4(.a(a[3]),.b(rb[3]),.cin(ca[2]),.sum(y[3]),.cout(ca[3]));
add1 inst_5(.a(a[4]),.b(rb[4]),.cin(ca[3]),.sum(y[4]),.cout(ca[4]));
add1 inst_6(.a(a[5]),.b(rb[5]),.cin(ca[4]),.sum(y[5]),.cout(ca[5]));
add1 inst_7(.a(a[6]),.b(rb[6]),.cin(ca[5]),.sum(y[6]),.cout(ca[6]));
add1 inst_8(.a(a[7]),.b(rb[7]),.cin(ca[6]),.sum(y[7]),.cout(ca[7]));
assign sign=y[7];
assign ca1=ca[7];
assign ov=ca[7]^ca[6];
endmodule

module add1(
input a,b,cin,
output sum,cout
);
assign sum=a^b^cin;
assign cout=a&b|a&cin|b&cin;
endmodule
```

这里就通过 case 语句选择参与运算的 B 是什么，然后后面用 8 个一位全加器实现 8 位数的加法器，这样做方便我产生各种信号

```
module lu(
input[7:0] a,b,
input[1:0] s,
output reg[7:0] y
    );
 always@(*)
 begin
 case(s)
 0:y=a&b;
 1:y=a|b;
 2:y=~a;
 3:y=a^b;
 endcase
 end
endmodule
```

LU 相对而言就比较容易了，也是一样的通过 case 语句选择与，或，非，异或这四种运算

```
module shifter(
input [7:0] b,
input [1:0]s,
output reg[7:0] y,
output reg ca
    );
 always@(*)
 begin
 case(s)
 0:
 begin
 y=b;
 ca=0;
 end
 1:
 begin
 ca=b[7];
 y={b[6:0],1'b0};
 end
 2:
 begin
 ca=b[0];
 y={1'b0,b[7:1]};
 end
```

```
 ca=b[0];
 y={b[0],b[7:1]};
 end
 endcase
 end

endmodule
```

然后是移位器部分，这一部分也是主要通过 case 语句选择，移位我使用向量拼接的方法实现

```
module alu(
input cin,
input [7:0] a,b,
input [1:0]s,
input s1,
output [7:0] y,
output sign,ov,ca
    );
 wire [7:0] y1,y2;
 au inst_1(.a(a),.b(b),.s(s),.cin(cin),.sign(sign),.y(y1),.ov(ov),.ca1(ca));
 lu inst_2(.a(a),.b(b),.s(s),.y(y2));
 assign y=(s1==0)?y1:y2;
endmodule
```

调用上面创建的模块，用一个选择信号选择算术运算还是逻辑运算，就构成了 ALU

```
module algori(
input[7:0] a,b,
input[1:0] s,
input s1,cin,s2,
output[7:0] y,
output sign,ca,ov,ze
);
wire [7:0]y1,y2;
wire ca1,ca2;
alu inst_1(.a(a),.b(b),.y(y1),.s(s),.s1(s1),.cin(cin),.sign(sign),.ca(ca1),.ov(ov));
assign ze=(y1==0)?1:0;
shifter inst_2(.b(b),.s(s),.y(y2),.ca(ca2));
assign ca=(s2==0)?ca1:ca2;
assign y=(s2==0)?y1:y2;
endmodule
```

最后加上移位器部分，就构成了完整的运算器，这里面输出的信号 ca 是进位，sign 是符号，ov 是溢出，ze 是零

```
module regs(
input[1:0] usel,sela,selb,
input[7:0] a,
input  ew,clk,rst,
output reg [7:0] outa,outb
    );
 reg[7:0] reg1,reg2,reg3,reg4;
always@(posedge clk or posedge rst)
begin
if(rst)
   begin
    reg1=0;
    reg2=0;
    reg3=0;
    reg4=0;
   end
else
begin
if(ew)
   begin
    case(usel)
    0:reg1=a;
    1:reg2=a;
    2:reg3=a;
```

```
    case(usel)
    0:reg1=a;
    1:reg2=a;
    2:reg3=a;
    3:reg4=a;
    endcase
    end
  else
  ;
   end
   end
always@(*)
begin
case(sela)
 0:outa=reg1;
 1:outa=reg2;
 2:outa=reg3;
 3:outa=reg4;
 endcase
 end


 always@(*)
begin
case(selb)
```

```
  always@(*)
begin
case(selb)
  0:outb=reg1;
  1:outb=reg2;
  2:outb=reg3;
  3:outb=reg4;
  endcase
  end

endmodule
```

这一部分是寄存器模块，定义了四个寄存器，在时钟信号到来时，如果 ew 为真也就是

能写入，就把 a 写入到选定寄存器中，之后把寄存器的值输出到 outa 和 outb

```
module id(
input[15:0] instru,
output ew,coutb,rdm,wtm,flag,ejp,
output [4:0] opcode,
output [7:0] addr
    );
assign ew=~(instru[15]|instru[14])|instru[13];
assign coutb=instru[13];
assign rdm=instru[14];
assign wtm=instru[14]&~instru[13];
assign flag=~(instru[15]|instru[14]);
assign ejp=instru[15];
assign opcode=instru[12:8];
assign addr=instru[7:0];
endmodule
```

这一部分是指令译码器，按照 logisim 中所画电路写出来的

```
module ir(
input clk,
input [15:0]d,
output reg[15:0]q
    );
always@(posedge clk)
    q=d;
endmodule
```

这一部分是指令寄存器，比较简单不多说明了

```
module sr(
input sign,ov,ca,ze,clk,en,
input [4:0] op,
output out
    );
reg rout;
reg [3:0] q;
always@(posedge clk)
   begin
   if(en)
   q={ze,ca,sign,ov};
   else
   q=q;
   end
always@(*)
    begin
    case(op[3:1])
    3'b000: rout=1;
    3'b001: rout=q[0];
    3'b010: rout=q[1];
    3'b011: rout=q[2];
    3'b100: rout=q[3];
    default: rout=1;
    endcase
    end
```

这一部分是状态寄存器，主要功能就是存储电路当前的状态比如溢出，零，进位之类的，进而根据这些状态，cpu 选择进行相应的操作，比如指令地址的跳转

```
module pc(
input [7:0] offset,
input clk, rst, sel,
output reg[7:0] padd
    );
wire [7:0] offset1;
assign offset1=(sel==0)?8'h01:offset;
always@(posedge clk or posedge rst)
begin
if(rst)
    padd=8'h00;
else
    padd=padd+offset1;
end


endmodule
```

这一部分是程序计数器，用来实现指令地址的跳转的操作，偏移量要么为 1，也就是顺序执行，要么为前面产生的一个偏移量，表示向前或向后偏移多少

```
module control(
input clkt, clkjp, clkp,
input rst, sign, ov, ca, ze,
input [15:0] pin,
output [7:0]y,
output ew, outb, rdm, wtm, flag, ejp, sel1,
output [4:0] opcode,
output [7:0] addr
    );
wire out1;
wire [15:0] qpin;
assign sel1=out1&ejp;
pc inst_1( .clk(clkt), .rst(rst), .offset(addr), .sel(sel1), .padd(y));
sr inst_2( .sign(sign), .ov(ov), .ca(ca), .ze(ze), .en(flag), .clk(clkjp), .op(opcode), .out(out1));
ir inst_3( .clk(clkp), .d(pin), .q(qpin));
id inst_4( .instru(qpin), .ew(ew), .outb(outb), .rdm(rdm), .wtm(wtm), .flag(flag), .ejp(ejp), .opcode(opcode), .addr(addr));

endmodule
```

将他们的对应接口接在一起就得到了控制器

这一部分是程序计数器，用来实现指令地址的跳转的操作，偏移量要么为 1，也就是顺序执行，要么为前面产生的一个偏移量，表示向前或向后偏移多少

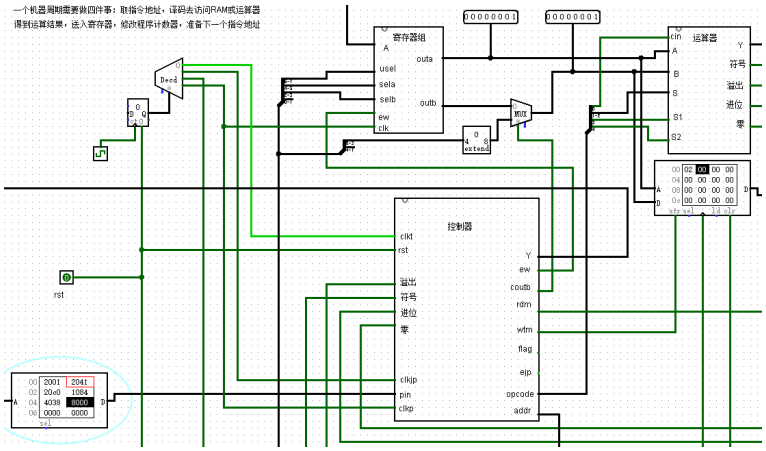
```
always@(posedge clk or posedge rst)
begin
    if(rst)
    count=0;
    else
    count=count+1;
end
assign clkt=(count==0)?1:0;
assign clkjp=(count==1)?1:0;
assign clkram=(count==2)?1:0;
assign clkp=(count==3)?1:0;
control inst_1(
.clkt(clkt), .clkjp(clkjp), .clkp(clkp), .rst(rst),
.sign(sign), .ov(ov), .ca(ca), .ze(ze), .pin(pin), .y(y),
.ew(ew), .outb(outb), .rdm(rdm), .wtm(wtm), .addr(addr),
.opcode(opcode)
);
dist_mem_gen_0 inst_2( a(y), spo(pin));
regs inst_3( a(y1), .sel(addr[7:6]), .sela(addr[5:4]), .selb(addr[3:2]), .clk(clkp), .ew(ew), .outa(outa), .outb(outb), .rst(rst));
assign outb1={4'b0000,addr[3:0]};
assign outb1=(outb==0)?outb:outb1;
algori inst_4( a(outa), b(outb1), s(opcode[2:1]), .cin(opcode[0]), .s1(opcode[3]), .s2(opcode[4]), .y(y2), .sign(sign), .ov(ov),
.ca(ca), .ze(ze));
dist_mem_gen_1 inst_5( .clk(clkram), .a(outa), .d(outb1), .we(wtm), .spo(data));
assign y1=(rdm==0)?y2:data;
```

最后把控制器和运算器接在一起，并接上 ROM，RAM，就构成一块简易的 cpu 了，这里指令周期的划分，我采用的是 2 位的 count 信号，产生脉冲，去驱动对应的模块，RAM,ROM 使用了 IP 核

【调试分析】

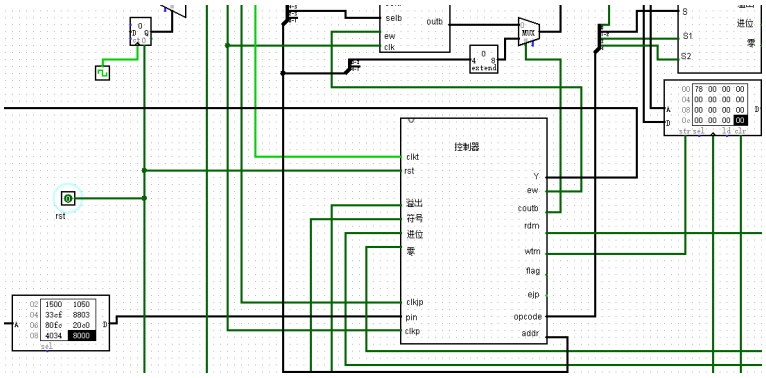
logisim 部分

这里我直接展示成品的电路的运行，在 ROM 中依次存入 2001 2041 20c0 1084 4038 8000 这个表示进行 1+1 运算，首先在第一个寄存器存入值 1，第二个寄存器存入值 1，第三个寄存器存入值 0，表示运算结果存到 RAM 的零号地址，最后终止



运算结果如上，可以看见这个 cpu 正常运行

对于稍微复杂的程序，比如考察循环语句的实现，在 ROM 中依次存入 2000 2040 1500 1050 33cf 8803 80fc 20c0 4034 8000 这个表示从 1 加到 f 也就是 15，运行 cpu 得到下面结果

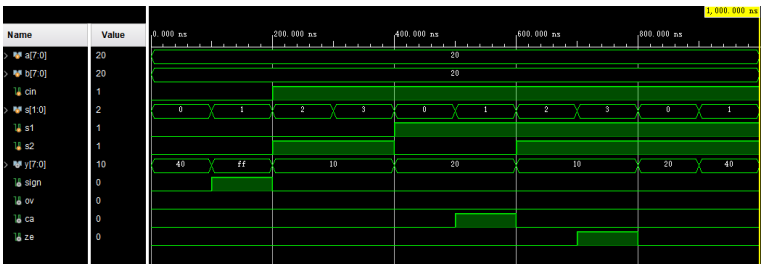


运算结果 78，注意这里是十六进制数，所以换成十进制数为 120，正是我们要的结果

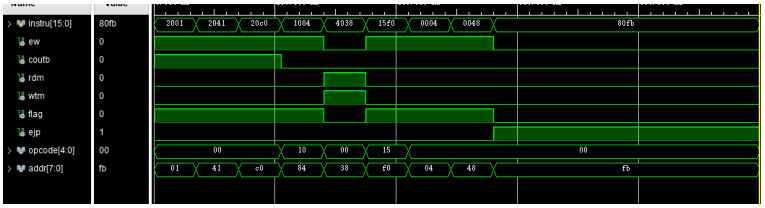
verilog 部分

这一部分在调试起来是比较困难的，只能通过仿真，查看其波形来看电路是否实现正确，不过仿真正确也不见得就一定能在所有情况下正确，不过这里也没办法 orz

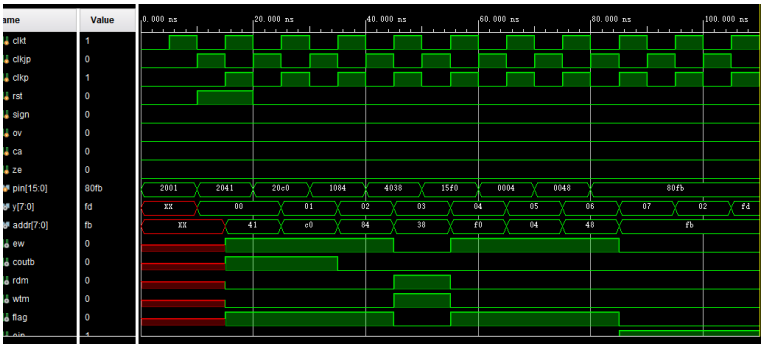
这里我给基本所有的模块都写了仿真文件来测试，下面展示几个仿真的结果



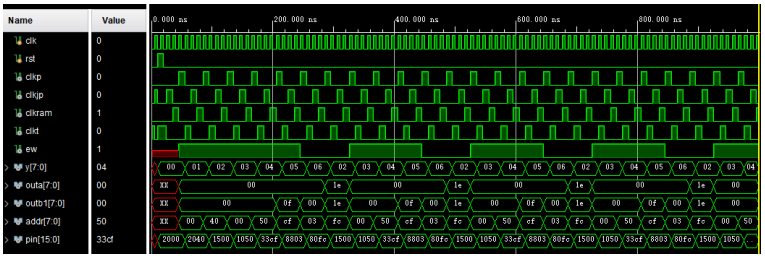
上面是对运算器的仿真结果，输入数据 a，b 都为 20，s 循环 0 到 3 用来选择具体的运算，s1 在算数运算和逻辑运算之间选择，s2 在前面的运算和移位运算中选择。仿真波形的输出与预期一致



这是对指令译码器的仿真，输入是我任意给定的一列指令，然后看输出的波形，波形与预期一致



对控制器的仿真，按照顺序产生一系列时钟信号，按顺序执行，指令也是任意输入的指令，主要看输出的波形是否能对上预期，这里也是符合预期的



这是对 cpu 的仿真，我在 ROM 中初始化的代码就是从一加到 15，也就是需要循环，这里可以看到指令在运行的过程中可以正常跳转，符合预期

【总结与思考】

毕竟这也算是第一次搭建一个比较有规模的电路，最大的问题就是一开始并不知道从哪里下手，不过在把整个大电路拆分成很多比较容易的小模块之后，每完成一个模块就有一个模块的进展，这种体验是非常好的，完成从无到有的转变，积沙成塔，最终电路可以正常运行，相当的有成就感

这次实验我选择的是搭建一个简易的 cpu，主要是想增进一下对 cpu 的理解，同时又可以综合运用前面所学的各种知识，收获还是很大的

当然这次实验还是存在很多可以改进的地方，比如我的这个 cpu 的功能实际上还是算比较简陋的，可以实现一些简单的程序，但是需要自己手动解码，把程序翻译成这里面的指令码，然后运行，可以考虑搭建 mips 架构，不过这些都留待后面学计组的时候再说吧，对于这门课而言，应该是足够的了