Project4 Verilog 完成单周期 CPU 开发

一、总体设计

1、顶层模块

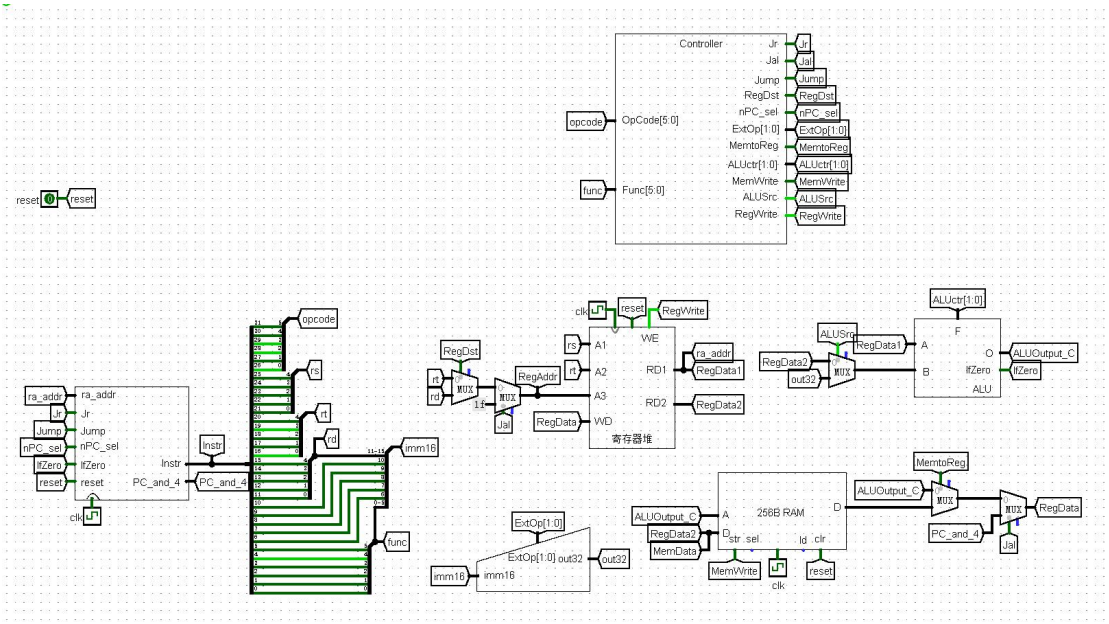

图 1　Logisim 顶层模块图

```
21  module mips(
22     input clk,reset
23      );
24
25      wire ifzero,  npc_sel,  jump,  jr,  jal,  regdst,  memtoreg,  memwrite,  alusrc,  regwrite;
26      wire [31:0] memout,  memaddr,  memdata,  aluoprand_a,  aluoprand_b,  aluoutput_c,  ra_addr,  instr,
27                  pc_and_4,  regdata, regdatal, regdata2,  out32;
28      wire [15:0] imm16;
29      wire [5:0] opcode,  func;
30      wire [4:0] rs,  rt,  rd,  regaddr;
31      wire [3:0] aluctr;
32      wire [1:0] extop;
33
34      //IFU
35      assign opcode[5:0] = instr[31:26], rs[4:0] = instr[25:21], rt[4:0] = instr[20:16],
36             rd[4:0] = instr[15:11], func[5:0] = instr[5:0], imm16[15:0] = instr[15:0],
37             ra_addr[31:0] = regdatal[31:0];
38      //ALU
39      assign aluoprand_a[31:0] = regdatal[31:0];
40      //DM
41      assign memdata[31:0] = regdata2[31:0];
42      assign memaddr[31:0] = aluoutput_c[31:0];
43
44      IFU ifu(
45      .clk(clk),// 1
46      .reset(reset),// 1
47      .ifzero(ifzero),// 1
48      .npc_sel(npc_sel),// 1
49      .jump(jump),// 1
50      .jr(jr),// 1
51      .ra_addr(ra_addr),// 32
```

```
52
53      .instr(instr),// 32
54      .pc_and_4(pc_and_4)// 32
55      );
56
57      control controller(
58      .opcode(opcode),// 6
59      .func(func),// 6
60
61      .jr(jr),// 1
62      .jal(jal),// 1
63      .jump(jump),// 1
64      .regdst(regdst),// 1
65      .npc_sel(npc_sel),// 1
66      .extop(extop),// 2
67      .memtoreg(memtoreg),// 1
68      .aluctr(aluctr),// 4
69      .memwrite(memwrite),// 1
70      .alusrc(alusrc),// 1
71      .regwrite(regwrite)// 1
72      );
73
74      GRF grf(
75      .rs(rs),// 5
76      .rt(rt),// 5
77      .regaddr(regaddr),// 5
78      .clk(clk),// 1
79      .reset(reset),// 1
80      .regwrite(regwrite),// 1
81      .regdata(regdata),// 32
82      .pc_and_4(pc_and_4),// 32
83
84      .regdata1(regdata1),// 32
85      .regdata2(regdata2)// 32
86      );
87
88      EXT ext(
89      .imm16(imm16),// 16
90      .extop(extop),// 2
91
92      .out32(out32)//32
93      );
94
95      ALU alu(
96      .aluctr(aluctr),// 4
97      .aluoprand_a(aluoprand_a),// 32
98      .aluoprand_b(aluoprand_b),// 32
99
100     .aluoutput_c(aluoutput_c),// 32
101     .ifzero(ifzero)// 1
102     );
103
104     DM dm(
105     .memwrite(memwrite),// 1
106     .clk(clk),// 1
107     .reset(reset),// 1
108     .memdata(memdata),// 32
109     .memaddr(memaddr),// 32
110     .pc_and_4(pc_and_4),// 32
111
112     .memout(memout)// 32
113     );
114
115     MuxRegRd muxregin(
116     .rt(rt),// 5
117     .rd(rd),// 5
118     .jal(jal),// 1
119     .regdst(regdst),// 1
120
121     .regaddr(regaddr)// 5
122     );
123
124     MuxALUNum muxaluin(
125     .out32(out32),// 32
126     .regdata2(regdata2),// 32
127     .alusrc(alusrc),// 1
128
129     .aluoprand_b(aluoprand_b)// 32
130     );
131
132     MuxRegData muxregdata(
133     .aluoutput_c(aluoutput_c),// 32
134     .memout(memout),// 32
135     .pc_and_4(pc_and_4),// 32
136     .jal(jal),// 1
137     .memtoreg(memtoreg),// 1
138
139     .regdata(regdata)// 32
140     );
141 endmodule
```

图 2 顶层模块代码

2、顶层多选器

```
module MuxRegRd(
   input [4:0] rt,rd,                          //输入寄存器地址
   input regdst,jal,                           //输入控制信号
   output  [4:0] regaddr                       //输出寄存器地址
    );
      assign regaddr = jal?32'h0000001f:(regdst?rd:rt);
endmodule

module MuxALUNum(
   input [31:0] regdata2,out32,                //输入立即数和寄存器值
   input alusrc,                               //输入控制信号
   output [31:0] aluoprand_b                   //输出ALU操作数
    );


   assign   aluoprand_b = alusrc?out32:regdata2;


endmodule

module MuxRegData(
   input [31:0] memout,aluoutput_c,pc_and_4,   //输入存储器值和ALU运算结果和pc_and_4
   input memtoreg,jal,                         //输入控制信号
   output [31:0] regdata                       //输出输入寄存器堆的值
    );


   assign regdata = jal?pc_and_4:(memtoreg?memout:aluoutput_c);

endmodule
```

图 3 顶层多选器代码

表 1 MuxRegRd 模块接口

信号	方向	描述
rt	I	5 位寄存器地址输入信号
rd	I	5 位寄存器地址输入信号
regdst	I	寄存器地址选择信号
jal	I	寄存器地址选择信号，判断当前指令是否为 jal
regaddr	O	5 位经选择后的寄存器地址输出信号

表 2 MuxALUNum 模块接口

信号	方向	描述
regdata2	I	32 位读取的寄存器的运算数输入信号
out32	I	32 位经扩展后的运算数输入信号
alusrc	I	运算数选择信号
aluoprand_b	O	32 位经选择后的运算数输出信号，为 ALU 的操作数

表 3 MuxRegData 模块接口

信号	方向	描述
memout	I	32 位读取的 DM 的数据输入信号
aluoutput_c	I	32 位 ALU 运算结果的数据输入信号
pc_and_4	I	32 位当前指令地址加 4 后的输入信号
memtoreg	I	数据选择信号，判断是否从 DM 中读数据至 GRF
jal	I	数据选择信号，判断当前指令是否为 jal
regdata	O	32 位经选择后的数据输出信号，为 GRF 的写入数据

表 4 顶层多选器功能定义

序号	多选器模块名称	功能描述
1	MuxRegRd	jal 有效则输出 0x1f，否则 regdst 有效则输出 rd,否则 输出 rt
2	MuxALUNum	alusrc 有效则输出 out32，否则 输出 regdata2
3	MuxRegData	jal 有效则输出 pc_and_4, 否则 memtoreg 有效则输出 memout，否则 输出 aluoutput_c

二、模块定义

1、IFU

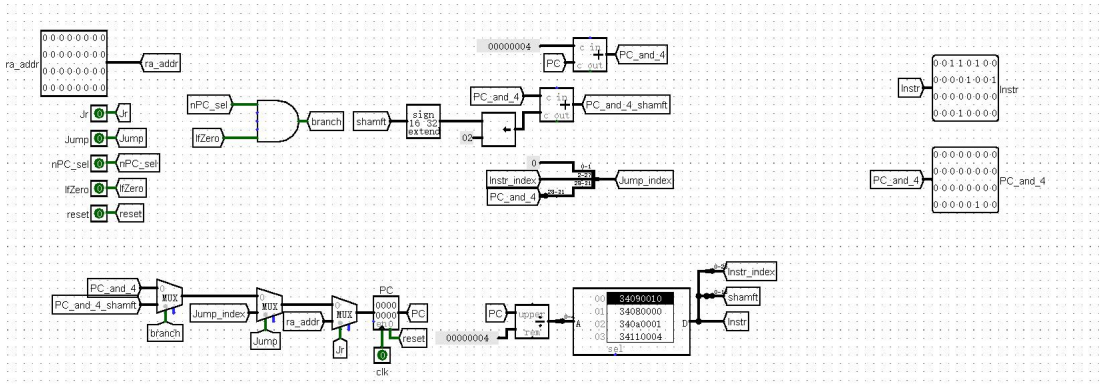

图 4　Logisim IFU 电路图

```
21  module IFU(
22     input jr,jump,npc_sel,ifzero,reset,clk,                //信号输入
23     input [31:0] ra_addr,
24     output [31:0] instr ,pc_and_4                          //信号输出
25      );
26
27      reg [31:0] IM[0:1023],PC = 32'h00003000;              //构建IM和PC，注意IM是32bit×1024字
28      wire [31:0] pc_and_four, pc_and_four_and_shamft, shamft_ext, jump_index;
29      wire [25:0] instr_index;
30      wire [15:0] shamft;
31      wire branch;
32
33
34      assign pc_and_four = PC + 4;//连接PC+4
35      assign instr_index = IM[PC[11:2]][25:0];//连接地址位移
36      assign shamft = IM[PC[11:2]][15:0];//连接地址偏移量
37      assign shamft_ext = {{14{shamft[15]}},{shamft[15:0]},{2'b00}};//拓展地址偏移量
38      assign pc_and_four_and_shamft = pc_and_four + shamft_ext;//得出偏移后的地址
39      assign branch = ifzero && npc_sel;//是否分支
40      assign jump_index = {pc_and_4[31:28],instr_index[25:0],2'b00};
41
42      assign pc_and_4 = pc_and_four;                         //连接PC+4
43      assign instr = IM[PC[11:2]];                           //连接Instr
44
45
46      initial
47      begin
48      $readmemh("code.txt",IM);
49      end
50
51      always @(posedge clk)                                  //时钟上升时,完成PC
52      begin
53        if(reset)                                            //同步复位
54        begin
55           PC <= 32'h00003000;
56        end
57        else                                                 //不复位
58        begin
59        //$display("%h %h\n",PC,IM[PC[11:2]]);
60           PC <= jr? ra_addr : (jump? jump_index : (branch? pc_and_four_and_shamft : pc_and_four));
61        end
62      end
63
64  endmodule
65
```

图 5　IFU 代码

（1）基本描述

IFU 主要作用是完成取指令功能。IFU 内部包含 PC、IM 以及其他相关逻辑操作。IFU 除了能执行顺序取指令外，还能根据指令的执行情况决定 PC 接下来的操作是顺序取指令还是转移取指令。

（2）模块接口

表 5 IFU 模块接口

信号名	方向	描述
jr	I	判断当前指令是否为 jr 指令的标志
jump	I	判断当前指令是否为跳转指令的标志
npc_sel	I	判断当前指令是否为 beq 指令的标志 1：当前指令为 beq 指令 0：当前指令非 beq 指令
ifzero	I	判断 ALU 计算结果是否为 0 的标志 1：计算结果为 0 0：计算结果非 0
reset	I	判断 PC 是否复位的信号 1：复位 0：无效
clk	I	时钟信号
ra_addr	I	32 位指令地址输入信号
instr	O	32 位当前的 MIPS 指令
pc_and_4	O	32 位当前指令地址加 4 后的输出信号

（3）功能定义

表 6 IFU 功能定义

序号	功能名称	功能描述
1	复位	当 reset 信号有效时 PC 设置成 0x0000_3000
2	取指令	根据 PC 指定的地址从 IM 中取出指令
3	计算下一条指令地址	*PC 取地址为 4 字节即一个字，所以 PC 的低 2 位地址可以省略 jr 有效则 PC←ra_addr，否则 jump 有效则 PC←PC[31:28]||instr_index||0^2，否则 ifzero 和 npc_sel 都有效则 PC←PC+4+sign_ext(offset||0^2)否则 PC←PC+4

2、GRF

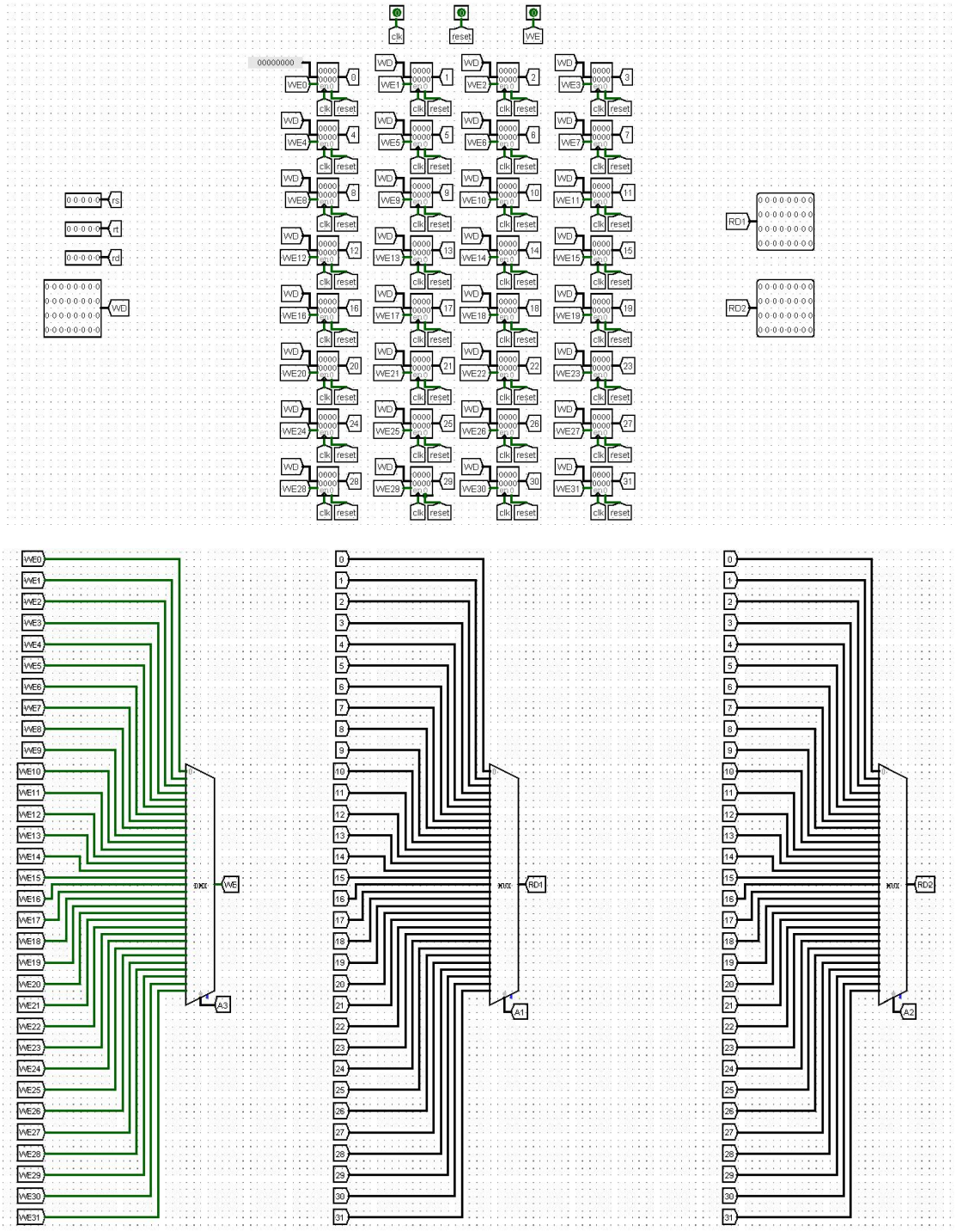

图 6　Logisim GRF 电路图

```
21  module GRF(
22     input [4:0] rs,rt,regaddr,                          //输入寄存器地址
23     input reset,clk,regwrite,                           //输入置位和时钟和使能信号
24     input [31:0] regdata,pc_and_4,                      //输入写入数据
25     output [31:0] regdata1,regdata2                     //输出读出数据
26      );
27      reg [31:0] GRF[0:31];                             //初始化寄存器堆
28
29      assign regdata1 = GRF[rs];                          //连接寄存器
30      assign regdata2 = GRF[rt];                          //连接寄存器
31
32      integer i;
33      initial
34      begin
35        for(i = 0 ; i < 32 ;i = i + 1)                  //初始化寄存器
36           GRF[i] <= 0;
37      end
38
39     always @ (posedge clk)                             //时钟上升沿工作
40      begin
41        if(reset)                                       //如果复位
42        begin
43           for(i = 0 ; i < 32 ;i = i + 1)               //全部归零
44           GRF[i] <= 0;
45        end
46        else if(regwrite)                         //否则如果不复位，写信号有效和非0号存储器的话则写入数据
47        begin
48         $display("@%h: $%d <= %h", pc_and_4 - 4, regaddr,regdata);
49           GRF[regaddr] <= regaddr == 0? 0 : regdata;
50        end
51      end
52
53  endmodule
```

图 7　GRF 代码

（1）基本描述

GRF 的作用主要是提供相应的寄存器以供指令完成相应的操作。GRF 内部包含 32 个 32 位寄存器，分别对应 0-31 号寄存器，其中 0 号寄存器的读取结果均为 0，还包含选择读写寄存器的逻辑操作电路。

（2）模块接口

表 7 GRF 模块接口

信号名	方向	描述
rs	I	5 位寄存器读地址输入信号
rt	I	5 位寄存器读地址输入信号
regaddr	I	5 位寄存器写地址输入信号
reset	I	复位信号 1：复位 0：无效
clk	I	时钟信号
regwrite	I	寄存器写使能信号
regdata	I	32 位寄存器写数据
pc_and_4	I	32 位当前指令地址加 4 后的输入信号
regdata1	O	32 位 rs 地址对应的寄存器的数据的输出信号
regdata2	O	32 位 rt 地址对应的寄存器的数据的输出信号

（3）功能定义

表 8 GRF 功能定义

序号	功能名称	描述
1	读寄存器	读出 rs、rt 地址对应寄存器中所存储的数据到 regdata1、regdata2
2	写寄存器	当 regwrite 信号有效且时钟上升沿到来时，将 regdata 写入 regaddr 所对应的寄存器中

3、ALU

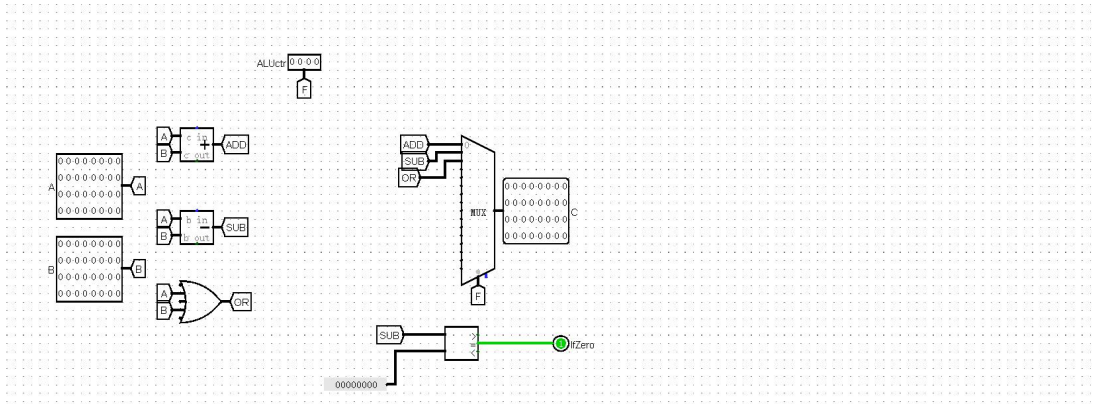

图 8 Logisim ALU 电路图

```
21  module ALU(
22     input [3:0] aluctr,                                          //控制信号输入
23     input [31:0] aluoprand_a,aluoprand_b,                                    //操作数输入
24     output  [31:0] aluoutput_c ,                                  //结果输出
25     output  ifzero                                          //零标记输出
26      );
27      parameter [3:0] ADDU = 4'b0000 ,SUBU = 4'b0001 , ORI = 4'b0010;      //定义运算参数
28      wire [31:0] addu,subu,ori;
29
30      assign ifzero = aluoprand_a - aluoprand_b == 0? 1 : 0;
31      assign addu = aluoprand_a + aluoprand_b;
32      assign subu = aluoprand_a - aluoprand_b;
33      assign ori = aluoprand_a | aluoprand_b;
34
35      assign aluoutput_c = aluctr == ADDU? addu:(aluctr == SUBU? subu:(aluctr == ORI? ori : 0));
36
37
38
39  endmodule
40
```

图 9 ALU 代码

（1）基本描述

ALU 的主要作用是完成算术逻辑指令或者其他指令所需要的算术逻辑操作。ALU 内部包含加法、减法、与或非等基本的算数逻辑部件。

（2）模块接口

表 9 ALU 模块接口

信号名	方向	描述
aluctr	I	4 位输入信号，ALU 的功能选择信号： 0000：ALU 进行加法运算 0001：ALU 进行减法运算 0010：ALU 进行或运算
aluoprand_a	I	32 位输入信号，ALU 的第一个操作数
aluoprand_b	I	32 位输入信号，ALU 的第二个操作数
aluoutput_c	O	32 位输出信号，ALU 的计算结果
ifzero	O	通过减运算输出的值和常数 0 作比较判断输入两值是否

		相等的信号

（3）功能定义

表 10 ALU 功能定义

序号	功能名称	描述
1	加运算	aluoutput_c = aluoprand_a + aluoprand_b
2	减运算	aluoutput_c = aluoprand_a - aluoprand_b
3	或运算	aluoutput_c = aluoprand_a | aluoprand_b

4、EXT

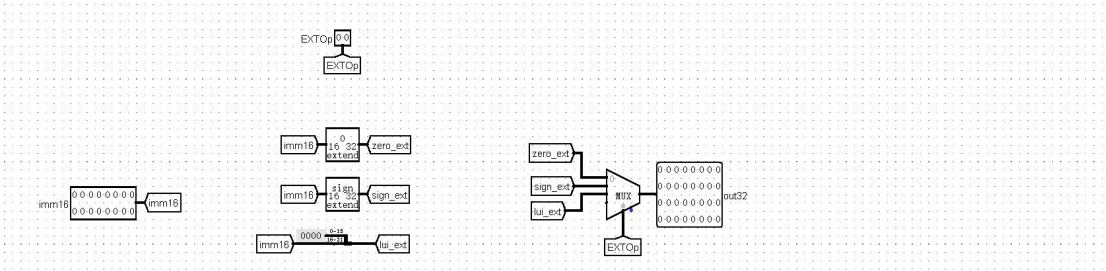

图 10 Logisim EXT 电路图

```
21  module EXT(
22     input [15:0] imm16,                                          //立即数输入
23     input [1:0] extop,                                           //控制信号输入
24     output [31:0] out32                                  //输出
25      );
26      parameter [1:0] zero_ext = 2'b00 ,sign_ext = 2'b01 ,lui_ext = 2'b10;      //定义扩展参数
27
28      wire [31:0] zeroext,signext,luiext;
29
30      assign zeroext = {{16{1'b0}},{imm16[15:0]}};
31      assign signext = {{16{imm16[15]}},{imm16[15:0]}};
32      assign luiext = {{imm16[15:0]},{16{1'b0}}};
33      assign out32 = extop == zero_ext? zeroext :(extop == sign_ext? signext:(extop == lui_ext? luiext:0));
34
35
36  endmodule
```

图 11　EXT 代码

（1）基本描述

EXT 的主要作用是完成对输入到其中的 16 位数据的符号扩展、零扩展以及将输入的 16 位数据加载到高位等操作。其内部包含与完成符号扩展、零扩展以及讲输入的 16 位数据加载到高位等操作的相关逻辑部件。

（2）模块接口

表 11 EXT 模块接口

信号名	方向	描述
imm16	I	输入 EXT 内部需要被扩展的 16 位数据
extop	I	输入数据进行扩展的方式的选择信号： 00：将 imm16 进行零扩展到 32 位 01：将 imm16 进行符号扩展到 32 位 10：将 imm16 加载到高位，低位补 0
out32	O	imm16 进行扩展后的数据输出

（3）功能定义

表 12 EXT 功能定义

序号	功能名称	描述
1	零扩展	将 imm16 进行高位补 0 扩展到 32 位
2	符号扩展	将 imm16 进行符号扩展到 32 位
3	加载到高位	将 imm16 加载到高位，低位补 0

5、DM

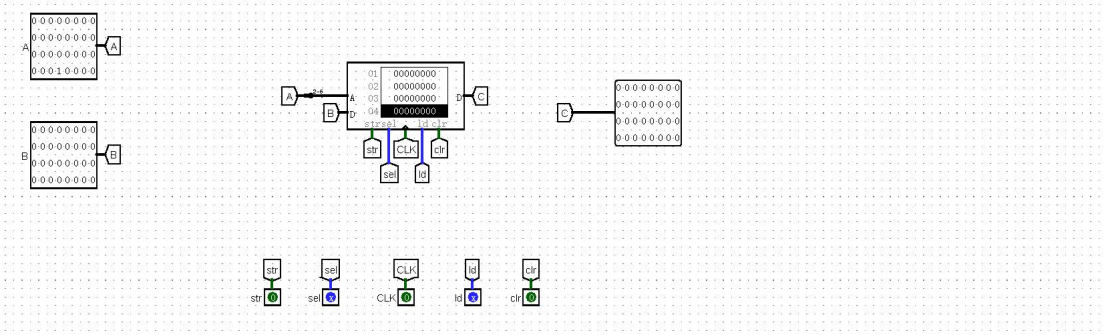

图 12 Logisim DM 电路图

```
21  module DM(
22     input [31:0] memaddr,memdata,pc_and_4,                    //输入地址信号和数据信号
23     input memwrite,clk,reset,                                 //输入写控制信号和时钟和清零信号
24     output [31:0] memout                                      //输出数据
25      );
26     reg [31:0] DM[0:1023];
27
28     assign memout = DM[memaddr[11:2]];                        //连接输出
29
30     integer i;
31     initial                                                   //初始化数据存储器
32     begin
33        for(i = 0;i < 1024 ;i = i + 1)
34           DM[i] <= 0;
35     end
36
37     always @ (posedge clk)                                    //在时钟上升沿
38     begin
39        if(reset)                                              //复位信号全部置0
40        begin
41           for(i = 0;i < 1024 ;i = i + 1)
42           DM[i] <= 0;
43        end
44        else if(memwrite)                                      //否则如果存数信号有效就存数
45        begin
46        $display("@%h: *%h <= %h",pc_and_4 - 4, memaddr,memdata);//别忘-4
47           DM[memaddr[11:2]] <= memdata;
48        end
49     end
50
51  endmodule
```

图 13 DM 代码

（1）基本描述

DM 的主要作用是作为 CPU 执行程序时的临时数据存储媒介。内部包含存储芯片及相关逻辑通路。

（2）模块接口

表 13 DM 模块接口

信号名	方向	描述
memaddr	I	32 位输入信号，操作存储器的地址
memdata	I	32 位输入信号，为写入数据的输入
pc_and_4	I	32 位当前指令地址加 4 后的输入信号
memwrite	I	读写控制信号 1：写信号 0：读信号
clk	I	时钟信号
reset	I	复位信号
memout	O	32 位输出信号，输出存储器操作地址输入对应的数据

（3）功能定义

表 14 DM 功能定义

序号	功能名称	描述
1	读	根据输入的存储器地址读出数据
2	写	根据输入的存储器地址，写入输入的数据
3	复位	当 reset 有效时将存储器所有数据置 0

三、Controller 设计

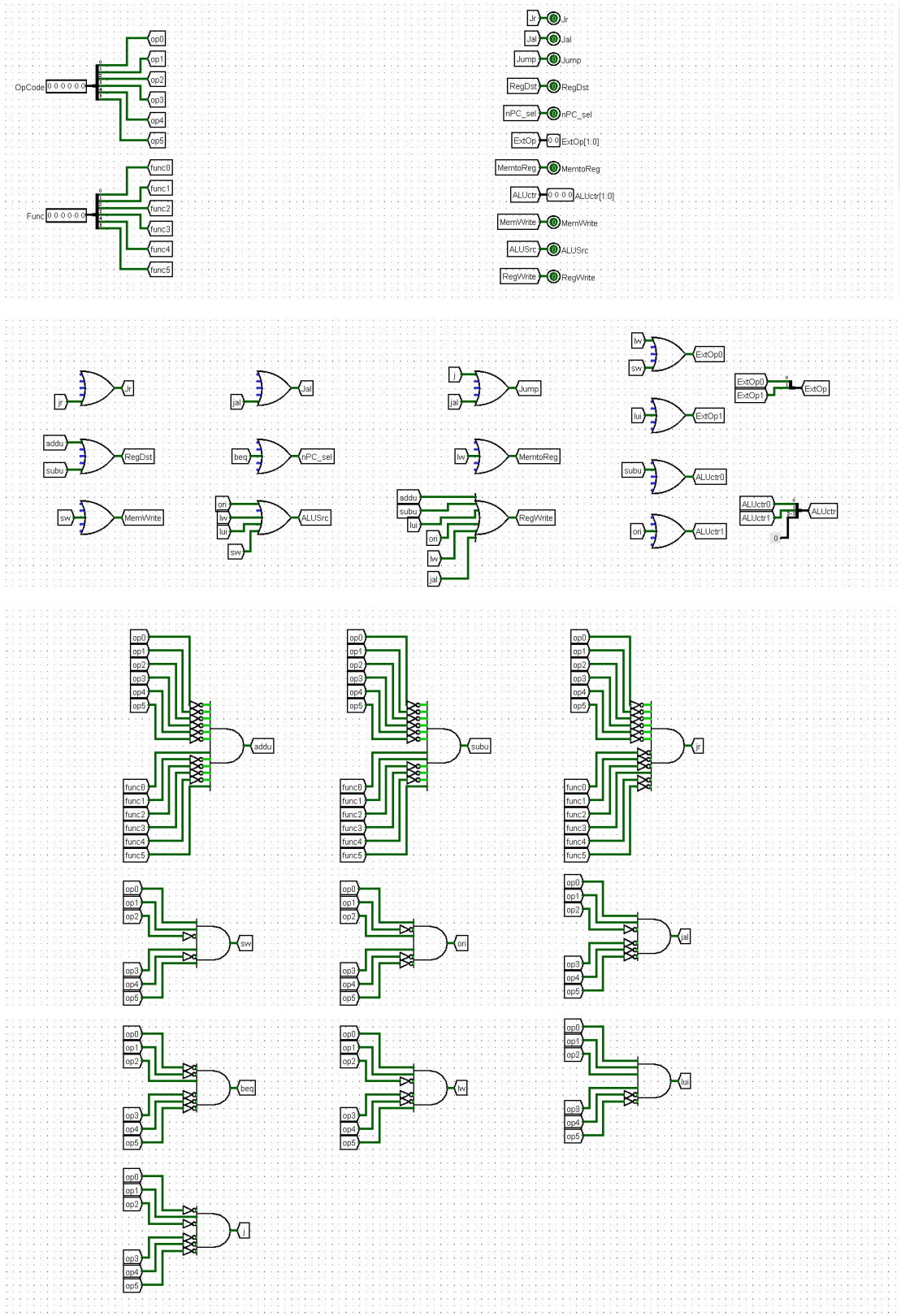

图 14 Logisim Controller 电路图

```
module control(
   input [5:0] opcode,func,
   output  jr ,jal ,jump ,regdst ,npc_sel ,memtoreg ,memwrite ,alusrc ,regwrite ,
   output  [1:0] extop ,
   output  [3:0] aluctr
    );

   parameter [5:0] RCLASS = 6'b000000, // R类指令 或 nop
                   ADDU = 6'b100001,   //ADDU
                   SUBU = 6'b100011,   //SUBU
                   ORI = 6'b001101,    //ORI
                   LW = 6'b100011,     //LW
                   SW = 6'b101011,     //SW
                   BEQ = 6'b000100,    //BEQ
                   LUI = 6'b001111,    //LUI
                   JAL = 6'b000011,    //JAL
                   JR = 6'b001000,     //JR
                   J = 6'b000010;      //J
   wire addu_and, subu_and, ori_and, lw_and, sw_and, beq_and, lui_and, jal_and, jr_and, j_and;

   //and logic
   assign addu_and = opcode == RCLASS? (func == ADDU? 1 : 0) : 0;
   assign subu_and = opcode == RCLASS? (func == SUBU? 1 : 0) : 0;
   assign ori_and = opcode == ORI? 1 : 0;
   assign lw_and = opcode == LW? 1 : 0;
   assign sw_and = opcode == SW? 1 : 0;
   assign beq_and = opcode == BEQ? 1 : 0;
   assign lui_and = opcode == LUI? 1 : 0;
   assign jal_and = opcode == JAL? 1 : 0;
   assign jr_and = opcode == RCLASS? (func == JR? 1 : 0) : 0;
   assign j_and = opcode == J? 1 : 0;
   //or_logic
   assign jr = jr_and;
   assign jal = jal_and;
   assign jump = j_and || jal_and;
   assign regdst = addu_and || subu_and;
   assign npc_sel = beq_and;
   assign memtoreg = lw_and;
   assign memwrite = sw_and;
   assign alusrc = ori_and || lw_and || lui_and || sw_and;
   assign regwrite = addu_and || subu_and || lui_and || ori_and || lw_and || jal_and;
   assign extop = lui_and? 2 :((lw_and || sw_and)? 1 : 0);
   assign aluctr = ori_and? 2 :(subu_and ? 1 : 0);

endmodule
```

图 15 Controller 代码

（1）基本描述

控制器的主要作用是译码，即将每一条机器指令中包含的信息，转化为给 CPU 各部分的控制信号。其内部主要包含与或门阵列，与逻辑部分的功能是识别，将输入的机器码识别为相应的指令。或逻辑部分的功能是生成，根据输入的指令不同，产生不同的控制信号。

（2）模块接口

表 15 Controller 模块接口

信号名	方向	描述
opcode	I	机器指令的操作码部分
func	I	机器指令的函数码部分
jr	O	JR 指令标志
jal	O	JAL 指令标志
jump	O	JUMP 指令标志
regdst	O	GRF 写地址控制
npc_sel	O	BEQ 指令标志
memtoreg	O	DM 输出数据控制标志
memwrite	O	DM 写入数据控制标志
alusrc	O	ALU 操作数控制标志
regwrite	O	GRF 写入数据控制标志
extop	O	EXT 扩展方式控制标志
aluctr	O	ALU 运算控制标志

（3）单周期真值表

表 16 Controller 单周期真值表

	op code	func	jr	jal	jump	reg dst	npc _sel	mem toreg	mem write	alusrc	reg write	extop	aluctr
addu	000000	100001	0	0	0	1	0	0	0	0	1	X	0
subu	000000	100011	0	0	0	1	0	0	0	0	1	X	1
ori	001101	N/A	0	0	0	0	0	0	0	1	1	0	2
lw	100011	N/A	0	0	0	0	0	1	0	1	1	1	0
sw	101011	N/A	0	0	0	X	0	0	1	1	0	1	0
beq	000100	N/A	0	0	0	X	1	0	0	0	0	X	0
j	000010	N/A	0	0	1	X	0	0	0	X	0	X	0
jr	000000	001000	1	0	0	X	0	0	0	X	0	X	0
jal	000011	N/A	0	1	1	0	0	0	0	X	1	X	0
lui	001111	N/A	0	0	0	0	0	0	0	1	1	2	0

四、测试 CPU

```
1  ori $t1,$zero,0x0010     # ori 测试程序要实现：zero（0）寄存器中的数据与立即数0x00000010进行或运算，结果存储在t1(9)寄存器中
2  ori $t0,$zero,0x0000     # ori 测试程序要实现：zero（0）寄存器中的数据与立即数0x00000000进行或运算，结果存储在t0(8)寄存器中
3  ori $t2,$zero,0x0001     # ori 测试程序要实现：zero（0）寄存器中的数据与立即数0x00000001进行或运算，结果存储在t2(10)寄存器中
4  ori $s1,$zero,0x0004     # ori 测试程序要实现：zero（0）寄存器中的数据与立即数0x00000004进行或运算，结果存储在s1(17)寄存器中
5  lui $s0,0xffff           # lui 测试程序要实现：立即数0xffff 加载到s0（16）寄存器高位
6  sw $s0,0($s1)            # sw 测试程序要实现：把s0(16)寄存器中的数据（1word），存储到s1(17)的值再加上偏移量0所指向的RAM中
7
8  loop:
9  beq $t0,$t1,loop_end     # beq 测试程序要实现：判断t0(8)的值和t1(9)的值是否相等，相等跳转loop_end
10 nop                      # nop 测试程序不用实现任何指令，默认PC <- PC+4
11
12 jal test                 # jal 测试程序要实现：跳转至test并把 PC + 4存至ra(31)中
13
14 addu $t0,$t0,$t2         # addu 测试程序要实现：t0(8)寄存器中的值加上t2(10)寄存器中的值后将结果存到t0(8)寄存器中
15 j loop                   # j 测试程序要实现：跳转至loop
16
17 test:
18 lw $s0,0($s1)            # lw 测试程序要实现：把s1(17)寄存器中的值再加上偏移量0所指向的RAM中的数据取出来并存到s0(16)寄存器中
19 subu $s0,$s0,$t1         # subu 测试程序要实现：把s0(16)寄存器中的值减去t1(9)寄存器中的值后将结果存到s0(16)寄存器中
20 sw $s0,0($s1)            # sw 测试程序要实现：把s0(16)寄存器中的数据（1word），存储到s1(17)的值再加上偏移量0所指向的RAM中
21 jr $ra                   # jr 跳转至ra(31)中存储的地址
22
23 loop_end:
24 sw $t1,8($s1)            # sw 测试程序要实现：把t1(9)寄存器中的数据（1word），存储到s1(17)的值再加上偏移量8所指向的RAM中
```

图 16 code 测试代码

Bkpt	Address	Code	Basic	Source
	0x00003000	0x34090010	ori $9,$0,0x00000010	1: ori $t1,$zero,0x0010 # ori 测试程序要实现：zero（0）寄存器中的数据与立即数0x00000010进行或运算，结果存储在t1(9,
	0x00003004	0x34080000	ori $8,$0,0x00000000	2: ori $t0,$zero,0x0000 # ori 测试程序要实现：zero（0）寄存器中的数据与立即数0x00000000进行或运算，结果存储在t0(8,
	0x00003008	0x340a0001	ori $10,$0,0x00000001	3: ori $t2,$zero,0x0001 # ori 测试程序要实现：zero（0）寄存器中的数据与立即数0x00000001进行或运算，结果存储在t2(1.
	0x0000300c	0x34110004	ori $17,$0,0x00000004	4: ori $s1,$zero,0x0004 # ori 测试程序要实现：zero（0）寄存器中的数据与立即数0x00000004进行或运算，结果存储在s1(1.
	0x00003010	0x3c10ffff	lui $16,0x0000ffff	5: lui $s0,0xffff # lui 测试程序要实现：立即数0xffff 加载到s0（16）寄存器高位
	0x00003014	0xae300000	sw $16,0x00000000($17)	6: sw $s0,0($s1) # sw 测试程序要实现：把s0(16)寄存器中的数据（1word），存储到s1(17)的值再加上偏移量0所指向.
	0x00003018	0x11090008	beq $8,$9,0x00000008	9: beq $t0,$t1,loop_end # beq 测试程序要实现：判断t0(8)的值和t1(9)的值是否相等，相等跳转loop_end
	0x0000301c	0x00000000	nop	10: nop # nop 测试程序不用实现任何指令，默认PC <- PC+4
	0x00003020	0x0c000c0b	jal 0x0000302c	12: jal test # jal 测试程序要实现：跳转至test并把 PC + 4存至ra(31)中
	0x00003024	0x010a4021	addu $8,$8,$10	14: addu $t0,$t0,$t2 # addu 测试程序要实现：t0(8)寄存器中的值加上t2(10)寄存器中的值后将结果存到t0(8)寄存器中
	0x00003028	0x08000c06	j 0x00003018	15: j loop # j 测试程序要实现：跳转至loop
	0x0000302c	0x8e300000	lw $16,0x00000000($17)	18: lw $s0,0($s1) # lw 测试程序要实现：把s1(17)寄存器中的值再加上偏移量0所指向的RAM中的数据取出来并存到s0(1.
	0x00003030	0x02098023	subu $16,$16,$9	19: subu $s0,$s0,$t1 # subu 测试程序要实现：把s0(16)寄存器中的值减去t1(9)寄存器中的值后将结果存到s0(16)寄存器.
	0x00003034	0xae300000	sw $16,0x00000000($17)	20: sw $s0,0($s1) # sw 测试程序要实现：把s0(16)寄存器中的数据（1word），存储到s1(17)的值再加上偏移量0所指向.
	0x00003038	0x03e00008	jr $31	21: jr $ra # jr 跳转至ra(31)中存储的地址
	0x0000303c	0xae290008	sw $9,0x00000008($17)	24: sw $t1,8($s1) # sw 测试程序要实现：把t1(9)寄存器中的数据（1word），存储到s1(17)的值再加上偏移量8所指向.

图 17 code 测试机械码

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000000	0xfffeff00	0x00000000	0x00000010	0x00000000	0x00000000	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

图 18 MARS 模拟结果之 DM

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000010
$t1	9	0x00000010
$t2	10	0x00000001
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xfffeff00
$s1	17	0x00000004
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00001800
$sp	29	0x00002ffc
$fp	30	0x00000000
$ra	31	0x00003024
pc		0x00003040
hi		0x00000000
lo		0x00000000

图 19 MARS 模拟结果之 GRF

Address: Columns: auto Address Radix: Hexadecimal Value Radix: Hexadecimal

	0	1
0x0	00000000	FFFEFF00
0x2	00000000	00000010
0x4	00000000	00000000
0x6	00000000	00000000
0x8	00000000	00000000
0xA	00000000	00000000
0xC	00000000	00000000
0xE	00000000	00000000
0x10	00000000	00000000
0x12	00000000	00000000
0x14	00000000	00000000
0x16	00000000	00000000
0x18	00000000	00000000
0x1A	00000000	00000000
0x1C	00000000	00000000
0x1E	00000000	00000000
0x20	00000000	00000000
0x22	00000000	00000000
0x24	00000000	00000000
0x26	00000000	00000000
0x28	00000000	00000000
0x2A	00000000	00000000
0x2C	00000000	00000000
0x2E	00000000	00000000

图 20 ISIM 仿真结果之 DM

	0	1
0x0	00000000	00000000
0x2	00000000	00000000
0x4	00000000	00000000
0x6	00000000	00000000
0x8	00000010	00000010
0xA	00000001	00000000
0xC	00000000	00000000
0xE	00000000	00000000
0x10	FFFEFF00	00000004
0x12	00000000	00000000
0x14	00000000	00000000
0x16	00000000	00000000
0x18	00000000	00000000
0x1A	00000000	00000000
0x1C	00000000	00000000
0x1E	00000000	00003024

图 21 ISIM 仿真结果之 GRF

jal test:

图 22 ISIM 仿真结果之波形

五、思考题

1、根据你的理解，在下面给出的 DM 的输入示例中，地址信号 addr 位数为什么是[11:2]而不是[9:0]？这个 addr 信号又是从哪里来的？

文件	模块接口定义

dm.v	dm(clk,reset,MemWrite,addr,din,dout);
	input clk; //clock
	input reset; //reset
	input MemWrite; //memory write enable
	input [11:2] addr; //memory's address for write
	input [31:0] din; //write data
	output [31:0] dout; //read data

MIPS 中内存地址按字节编址，地址信号为 4 的倍数，而设置的 CPU 内存按字编址，所以取[11:2]相当于除以 4，即把字节的地址转成字的地址。addr 信号从 ALU 模块中来。

2、在相应的部件中，reset 的优先级比其他控制信号（不包括 clk 信号）都要高，且相应的设计都是同步复位。清零信号 reset 是针对哪些部件进行清零复位操作？这些部件为什么需要清零？

reset 是对 GRF 和 DM 进行清零复位操作。因为首指令的地址是 0x000030000，所以 PC 不能进行清零复位操作。而 GRF 和 DM 是存储数据的单元，复位时应全部清零。

3、列举出用 Verilog 语言设计控制器的几种编码方式（至少三种），并给出代码示例。

宏定义 define：

```
`define lw = opcode == 100011? 1 : 0;
```

三目运算符+assign：

```
assign lw = opcode == 100011? 1 : 0;
```

case 选择：

```
always @ (*)begin
    case(opcode)
        6’b000000:
            case(func)
                6’b100011:
```

```
                addu <= 1;
                endcase
        6’b100011:
            lw <= 1;
    endcase
end
```

4、根据你所列举的编码方式，说明他们的优缺点。

宏定义 define:

优点：组合逻辑描述，清晰直观，便于理解

缺点：多位信号如 extop 的判断繁琐

case 选择：

优点：清晰直观，便于添加指令

缺点：写在 always 块中，与 controller 纯组合逻辑描述有出入，可能会产生未知错误

三木运算符+assign:

优点：能很好地描述 logisim 中的 controller 的与或逻辑部分，描述的结构一致

缺点：多位信号如 extop 的判断繁琐

5、C 语言是一种弱类型程序设计语言。C 语言中不对计算结果溢出进行处理，这意味着 C 语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持 C 语言，MIPS 指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的 Operation 部分 。

查阅资料得到 addi 和 addiu、add 和 addu 的 operation 部分如下：

addi:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + sign_extend(immediate)
if temp32 ≠ temp31 then
SignalException(IntegerOverflow)
```

else

GPR[rt] ← temp

endif

addiu:

temp ← GPR[rs] + sign_extend(immediate)

GPR[rt]← temp

add:

temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)

if temp32 ≠ temp31 then

SignalException(IntegerOverflow)

else

GPR[rd] ← temp

endif

addu:

temp ← GPR[rs] + GPR[rt]

GPR[rd] ← temp

可以看出在忽略溢出的情况下，addi 和 addiu 执行的操作都是

temp ← GPR[rs] + sign_extend(immediate)

GPR[rt]← temp

add 和 addu 执行的操作都是

temp ← GPR[rs] + GPR[rt]

GPR[rd] ← temp

所以在忽略溢出的情况下，addi 和 addiu、add 和 addu 是等价的

6、根据自己的设计说明单周期处理器的优缺点。

优点：每个单周期执行一条指令，不会像多周期一样产生冒险，设计简单

缺点：因为每个单周期执行一条指令，故处理器的周期由处理器支持的指令集中的指令的最长路径决定，会导致程序执行效率低

7、简要说明 jal、jr 和堆栈的关系。

jal 与 jr 成对出现，每次执行 jal 指令，都会把下一条指令的地址 PC+4 存进 ra 中，相当于压栈，然后跳转；而每次执行 jr 指令，程序则跳转回 ra 存储的地址对应的指令，相当于出栈。意即当调用多次 jal 与 jr 时，我们总会把之前的 ra 值存在内存中，然后 jal 存进 ra 对应的地址总是最后的地址，即存进栈顶元素，而使用 jr 时，则跳转至当前的 ra 对应的地址，即取出栈顶元素，符合栈的 LIFO 原则。