Lab 2简单ALU、存储IP设计

ALU设计

实验目的

1. 了解流水线PipeLine的设计原理

- 流水线设计思想

  ✍需要一个顶层模块，顶层模块的输入输出和第一个流水子模块一致

  有几个流水段就设计几个子模块

这里以3个子模块a, b, c为例

顶层模块的输入给子模块a

子模块a的输出给子模块b

子模块b的输出给子模块c

子模块c的输出给顶层模块的输出

```
module pipline(
    input clk,
    input rst_n,
    input [7:0] data_in,
    input in_en,
    output out_en,
    output [7:0] data_out
    );

    wire [7:0] data_a, data_b, data_c;
    wire in_en_a, in_en_b, in_en_c;
    wire out_en_a, out_en_b, out_en_c;
    assign in_en_a = in_en;
    assign in_en_b = out_en_a;
    assign in_en_c = out_en_b;

    sequence_A sq_a(.clk(clk), .rst_n(rst_n), .data_in(data_in), .in_en(in_en_a), .out_en(out_en_a), .data_out(data_a));
    sequence_B sq_b(.clk(clk), .rst_n(rst_n), .data_in(data_a ), .in_en(in_en_b), .out_en(out_en_b), .data_out(data_b));
    sequence_C sq_c(.clk(clk), .rst_n(rst_n), .data_in(data_b ), .in_en(in_en_c), .out_en(out_en_c), .data_out(data_c));

    assign out_en = out_en_c;
    assign data_out = data_c;

endmodule
```

$y=(x_1+x_2+x_3+x_4)\div 4+15$

$m=x_1+x_2+x_3+x_4$

$n=m\div 4$

$y=n+15$

2. 了解算术逻辑单元ALU的设计原理

根据控制信号设置ALU执行对应的操作运算

3. 熟悉并运用Verilog语言设计ALU

4. 熟悉并运用Verilog语言设计流水线全加器

5. 学习Verilog不同形式的编程方式，理解assign和always的区别

assign相当于一根导线，只能针对wire等线网型赋值，是组合逻辑

always则可以根据@事件，选择是组合逻辑还是时序逻辑

always@(*)，是组合逻辑，内部采用阻塞赋值，使用reg型变量

always@(posedge clk)等，是时序逻辑，内部采用非阻塞赋值，使用reg型变量

简单ALU设计

实验要求

$F_{2:0}$	功能	$F_{2:0}$	功能
000	A + B(Unsigned)	100	$\overline{A}$
001	A - B	*101*	*SLT*
010	A AND B	110	未使用
011	A OR B	111	未使用

表 1: 算数运算控制码及功能

本次实验实现一个**具有 N 位输入和 N 位输出**的算数逻辑单元的电路符号。算术逻辑单元根据执行哪个功能的控制信号F（左图所示），执行对应功能后输出N 位结果。并将N位结果通过7段数码管（无小数点）进行显示

原理图如下：

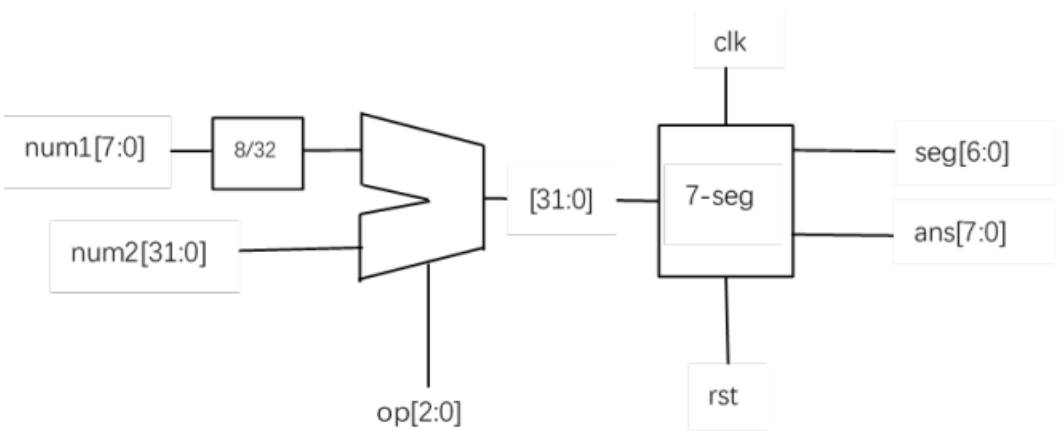


图 2: ALU 实验原理图

实验思路

ALU模块：接受N位的输入A和B，按照操作码进行运算N位输出结果

数码管显示模块：接受N位的输入，然后每4位交付给单个数码管显示模块进行显示→输出对应的数码管显示编码

数码管顶层模块负责指定位选码和数字，底层模块负责根据数字指定段选码

顶层模块：接受N位输入A和B，输出结果对应的数码管显示编码（每个结果对应8个编码）

设计代码

- ALU模块

```Verilog
`timescale 1ns / 1ps

module aluDesign #(parameter N = 32)(
      input [N-1:0]A,
      input [N-1:0]B,
      input [2:0]func,
      output [N-1:0]res
);
   assign res = (func == 3'b0)?A+B:
         (func == 3'b001)?A-B:
         (func == 3'b010)?A&B:
         (func == 3'b011)?A|B:
         (func == 3'b100)?~A:
         (func == 3'b101)?A<B:0;
endmodule
```

- 数码管显示顶层模块

```Verilog
`timescale 1ns / 1ps

//这里设参数比较难选，就设置板上只有8个数码管，那么结果最多是32位
module digitshow (
    input clk,
    input rst,
    input [31:0] res,
```

```
    output reg[7:0]loc_sel,//数码管位选码
    output [6:0]seg_sel//数码管段选码
);
    reg [2:0] loc_cycle;
    reg [3:0] temp;

    always @(posedge clk or negedge rst) begin//一个clk输出一个4位
        if (~rst) begin
            loc_cycle=0;
        end
        else begin
            case (loc_cycle)
                3'b000:begin
                    loc_sel<=0;
                    temp<=res[3:0];
                end
                3'b001:begin
                    loc_sel<=1;
                    temp<=res[7:4];
                end
                3'b010:begin
                    loc_sel<=2;
                    temp<=res[11:8];
                end
                3'b011:begin
                    loc_sel<=3;
                    temp<=res[15:12];
                end
                3'b100:begin
                    loc_sel<=4;
                    temp<=res[19:16];
                end
```

```
                3'b101:begin
                    loc_sel=5;
                    temp=res[23:20];
                end
                3'b110:begin
                    loc_sel<=6;
                    temp<=res[27:24];
                end
                default:begin
                    loc_sel<=7;
                    temp<=res[31:28];
                end
            endcase
            loc_cycle=loc_cycle+1;
        end
    end
    sevendigit  sevendigit_inst (
            .digit(temp),
            .seg(seg_sel)
        );
endmodule
```

Verilog

- 数码管显示输出段选码模块

```
`timescale 1ns / 1ps

module sevendigit(
    input [3:0]digit,
    output [6:0] seg
);
assign seg=(digit==4'b0000)?7'b000_0001://abcdefg,共阳极
           (digit==4'b0001)?7'b100_1111:
```

```
            (digit==4'b0010)?7'b001_0010:
            (digit==4'b0011)?7'b000_0110:
            (digit==4'b0100)?7'b100_1100:
            (digit==4'b0101)?7'b010_0100:
            (digit==4'b0110)?7'b010_0000:
            (digit==4'b0111)?7'b000_1111:
            (digit==4'b1000)?7'b000_0000:
            (digit==4'b1001)?7'b000_0100:
            (digit==4'b1010)?7'b000_1000:
            (digit==4'b1011)?7'b110_0000:
            (digit==4'b1100)?7'b011_0001:
            (digit==4'b1101)?7'b100_0010:
            (digit==4'b1110)?7'b011_0000:
            (digit==4'b1111)?7'b011_1000:7'b111_1111;
endmodule
```

Verilog

- 顶层模块

```
`timescale 1ns / 1ps

module alutop #(parameter N = 32)(
    input clk,
    input rst,

    input [N-1:0]A,
    input [N-1:0]B,
    input [2:0]func,

    output [7:0]loc_sel,//数码管位选码———8个
    output [6:0]seg_sel//数码管段选码———7段
);
```

```
    wire [N-1:0]res;

    aluDesign # (.N(N)) lab1_aluDesign_inst (
        .A(A),
        .B(B),
        .func(func),
        .res(res)
    );

    digitshow  lab1_digitshow_inst (
        .clk(clk),
        .rst(rst),
        .res(res),
        .loc_sel(loc_sel),
        .seg_sel(seg_sel)
    );

endmodule
```

Verilog

测试代码

```
`timescale 1ns / 1ps

module alutop_tb;

    // Parameters
    localparam  N = 32;

    //Ports
    reg  clk;
    reg  rst;
    reg [N-1:0] A;
```

```
    reg [N-1:0] B;
    reg [2:0] func;
    wire [7:0] loc_sel;
    wire [6:0] seg_sel;

    integer  i=0;
    initial begin
        rst=0;
        clk=1;
        A=25;
        B=31;
        func=3'b000;
        #10;
        rst=1;
    end
    alutop # (
        .N(N)
    )alutop_inst (
        .clk(clk),
        .rst(rst),
        .A(A),
        .B(B),
        .func(func),
        .loc_sel(loc_sel),
        .seg_sel(seg_sel)
    );
    always #5  clk = ! clk ;
    always #80 begin
        if (i==0) begin
            #10;
            i=i+1;
        end
        A=A+2;
```

```
            B=B-1;
            func=func+1;
        end
endmodule
```

Verilog

带流水线的全加器实现

https://blog.csdn.net/Bunny9__/article/details/117967149书签：阻塞流水线实现加法器_Bunny9__的博客-CSDN博客

流水线简介

流水线的设计**实际上是把规模较大、层次较多的组合逻辑电路分为几级**，在每一级**插入寄存器组**并暂存中间数据。

K级流水线就是**从组合逻辑的输入到输出恰好有K个寄存器组（分为K级，每一级都有一个寄存器组）**，上一级的输出是下一级的输入

采用非阻塞赋值

```
always @ (posedge clk)
    pipe1_data  <=  din;

always @ (posedge clk)
    pipe2_data  <=  pipe1_data;

always @ (posedge clk)
    pipe3_data  <=  pipe2_data;


assign  dout = pipe3_data;
```

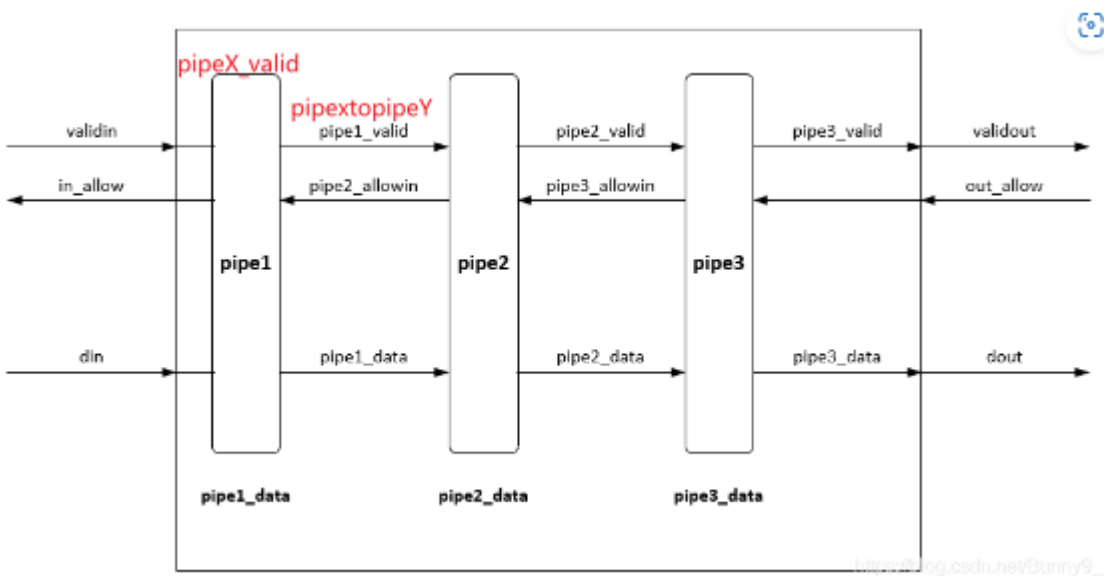
pipeX_data用来暂存数据

阻塞流水线

暂停流水：意味着**一旦后面的流水线被阻塞，前面的流水线也立刻被阻塞**，暂存原有数据——**暂存数据是通过带使能的触发器实现的➡只要使能信号无效，那么内部存储数据就会保存**

对某一级流水线而言，它会向后一级发送一个“**下一时刻我有数据传递给你pipeX_to_pipeY_valid**”的请求，也会向前一级发送一个“**下一个时刻我可以接收数据pipeX_allowin**”的反馈，然后它也会收到从后一级送来的“**是否可以接收数据pipeY_allowin**”的反馈，也会收到从前一级送来的“**是否有数据传递过来pipeX_to_pipeY_valid**”的请求。如果某一级流水线当前时刻有数据并且想在下一时刻传递给后一级流水线，但是后一级说它不能接收，那么该级流水线在下一时刻就要保持当前时刻的数据，就产生了阻塞。

✍也就是说pipeX_allowin和pipeX_to_pipeY_valid用来流水线寄存器之间信号的传递➡下一级流水是否可以接收数据，上一级是否有有效数据传递



- pipeX_valid：**当前流水级是否存在有效的数据，高有效**。在**需要清空流水线的时候**不需要把数据域的值置为无效，**只需要将valid拉低，代表数据无效，可以节约逻辑资源**
- pipeX_allowin：第X级传给第X-1级的状态，是否可以接收上一级的数据
- pipeX_ready_go：描述第X级的状态，1表示第X级的处理任务已经完成，可以传给X+1级。在实现多周期任务的时候，在最终结果没有得到的时候，**可以将这个信号拉低表示暂停住了**
- pipeX_to_pipeX+1_valid：从第X级传递给第X+1级，1表示下一时刻第X级有数据传递给第X+1级

✍pipeX_valid可以控制是否清空流水线

pipeX_ready_go可以控制是否阻塞流水线

- 流水线的阻塞和非阻塞

非阻塞就是直接赋值，如下图

```
always @(posedge clk) begin
    pipe1_data <= datain;
end

always @(posedge clk) begin
    pipe2_data <= pipe1_data;
end

always @(posedge clk) begin
    pipe3_data <= pipe2_data;
end

assign dataout = pipe3_data;

endmodule
```

阻塞则需要在每个流水段前设置一个阀门，只有满足一定条件，才可以打开该阀门，实现数据的传递

设置pipeX_valid来表示当前流水级上是有效数据

设置pipeX_allowin表示第X级流水可以接受前一级流水送来的数据

设置pipeX_to_pipeY_valid表示正在有数据从X级流水传至Y级

设置pipeX_ready_go表示X级流水上的数据已经准备好送往下一级流水

每一级正在往下一级传递的有效信号=这一级上有有效数据且正准备往下一级传

$$pipeX\ \ to\ \ pipeY\ \ valid = pipeX\ \ validspace\&spacepipeX\ \ ready\ \ go$$

每一级的允许数据进入信号=要么这一级上无有效数据要么是正准备走下一级也可以接收

$$pipeX\ \ allowin = \sim pipeX\ \ valid|(pipeX\ \ ready\ \ gospace\&spacepipeY\ \ allowin)$$

当允许进入信号有效时，可以用级之间正在传递信号赋值给该级数据有效

$$if spacepipeX\ \ allowinspacethenspacepipeX\ \ valid = pipeX'\ \ to\ \ pipeX\ \ valid$$

当允许进入信号有效，且级之间也正在传，那么就可以打通数据通路

$$if spacepipeX\ \ allowinspace\&spacepipeX'\ \ to\ \ pipeX\ \ validspacethenspacepipeX\ \ data = pipeX'\ \ data$$

```
// pipeline stage2
wire pipe2_allowin;
wire pipe2_ready_go;
wire pipe2_to_pipe3_valid;

assign pipe2_ready_go = ……
assign pipe2_allowin = ! pipe2_valid || pipe2_ready_go && pipe3_allowin ;
assign pipe2_to_pipe3_valid = pipe2_valid && pipe2_ready_go ;
always @(posedge clk or negedge rst) begin
    if ( ~rst ) begin
        pipe2_valid <= 1'b0;
    end
    else if ( pipe2_allowin ) begin //可以接受，那么表示这一级上可以进行数据传递，是否有效就可以传递
        pipe1_valid <= pipe1_to_pipe2_valid;
    end     pipe2
    if ( pipe1_to_pipe2_valid && pipe2_allowin ) begin
        pipe2_data <= pipe1_data ;
    end
end
```

```
module stallable_pipeline # (parameter WIDTH = 100 )(
    input clk,
    input rst,
    input validin,//表示有有效数据输入
    input [WIDTH-1:0]datain,
    input out_allow,//表示允许数据输出
    output validout,//表示有有效数据输出
    output [WIDTH-1:0]dataout
);
    reg pipe1_valid;//表示第一级流水上存在有效数据
    reg [WIDTH -1:0] pipe1_data ;
    reg pipe2_valid ;//表示第二级流水上存在有效数据
    reg [WIDTH -1:0] pipe2_data ;
    reg pipe3_valid ;//表示第三级流水上存在有效数据
    reg [WIDTH -1:0] pipe3_data ; // pipeline stage1
```

```
    //第一级流水段
    wire pipe1_allowin;//表示允许数据送至第一级流水段，允许进入的前提是这一级上无有效数据，或者准备好送往下一级且下一级可以接收
    wire pipe1_ready_go;//表示第一级上的数据将送往下一级
    wire pipe1_to_pipe2_valid;//表示正在有数据送往下一级，前提是这一级上有数据且已经准备好送往下一级

    assign pipe1_ready_go = ……
    assign pipe1_allowin = ! pipe1_valid || pipe1_ready_go && pipe2_allowin ;
    assign pipe1_to_pipe2_valid = pipe1_valid && pipe1_ready_go ;
    always @(posedge clk or negedge rst) begin
        if ( ~rst ) begin
            pipe1_valid <= 1'b0;
        end
        else if ( pipe1_allowin ) begin //可以接受，那么表示这一级上可以进行数据传递，是否有效就可以传递
            pipe1_valid <= validin ;
        end
        if ( validin && pipe1_allowin ) begin
            pipe1_data <= datain ;
        end
    end


    // pipeline stage2
    wire pipe2_allowin;
    wire pipe2_ready_go;
    wire pipe2_to_pipe3_valid;
```

```
    assign pipe2_ready_go = ……
    assign pipe2_allowin = ! pipe2_valid || pipe2_ready_go && pipe3_allowin ;
    assign pipe2_to_pipe3_valid = pipe2_valid && pipe2_ready_go ;
    always @(posedge clk or negedge rst) begin
        if ( ~rst ) begin
            pipe2_valid <= 1'b0;
        end
        else if ( pipe2_allowin ) begin //可以接受，那么表示这一级上可以进行数据传递，是否有效就可以传
递
            pipe2_valid <= pipe1_to_pipe2_valid;
        end
        if ( pipe1_to_pipe2_valid && pipe2_allowin ) begin
            pipe2_data <= pipe1_data ;
        end
    end
```

```
// pipeline stage3
wire pipe3_allowin;
wire pipe3_ready_go;

assign pipe3_ready_go = ……
assign pipe3_allowin = ! pipe3_valid || pipe3_ready_go && out_allow ;
always @(posedge clk or negedge rst) begin
    if ( ~rst ) begin
        pipe3_valid <= 1'b0;
    end
    else if ( pipe3_allowin ) begin //可以接受，那么久表示这一级上有有效数据
        pipe3_valid <= pipe2_to_pipe3_valid;
    end
    if ( pipe2_to_pipe3_valid && pipe3_allowin ) begin
```

```
            pipe3_data <= pipe2_data ;
        end
    end
    assign validout = pipe3_valid && pipe3_ready_go ;
    assign dataout  = pipe3_data ;
endmodule
```

Verilog

- 四级流水的八位全加器实现——非阻塞

流水全加器的实现思想是根据流水线级数划分等位的加数相加，比如四级流水的八位全加器则第一级最低两位相加得到相加结果和进位1；第二级是[3:2]位相加并加上进位1，同一级结果连接得到结果和进位2；第三级是[5:4]位相加并加上进位2与二级结果连接，依次类推……

- 设计代码

```
`timescale 1ns / 1ps


module lab1_8bit_adder(
    input clk,

    input [7:0]A,
    input [7:0]B,

    output reg out_en,
    output reg [7:0]res,
    output reg C
);
    reg c1,c2,c3;
    reg [1:0]r1;
    reg [3:0]r2;
    reg [5:0]r3;

    reg [7:0] A1,B1,A2,B2,A3,B3,A4,B4;
```

```
    always @(posedge clk) begin //用非阻塞赋值
            A1<=A;
            B1<=B;
            {c1,r1}<={1'b0, A1[1:0]} + {1'b0, B1[1:0]};
    end
    always @(posedge clk) begin
            A2<=A1;
            B2<=B1;
            {c2,r2}<={ {1'b0, A2[3:2]} + {1'b0, B2[3:2]} + c1, r1 };
    end
    always @(posedge clk) begin
            A3<=A2;
            B3<=B2;
            {c3,r3}<={ {1'b0, A3[5:4]} + {1'b0, B3[5:4]} + c2, r2 };
    end
    always @(posedge clk) begin
            A4<=A3;
            B4<=B3;
            {C,res}<={ {1'b0, A4[7:6]} + {1'b0, B4[7:6]} + c3, r3 };
            out_en<=1;
    end
endmodule
```

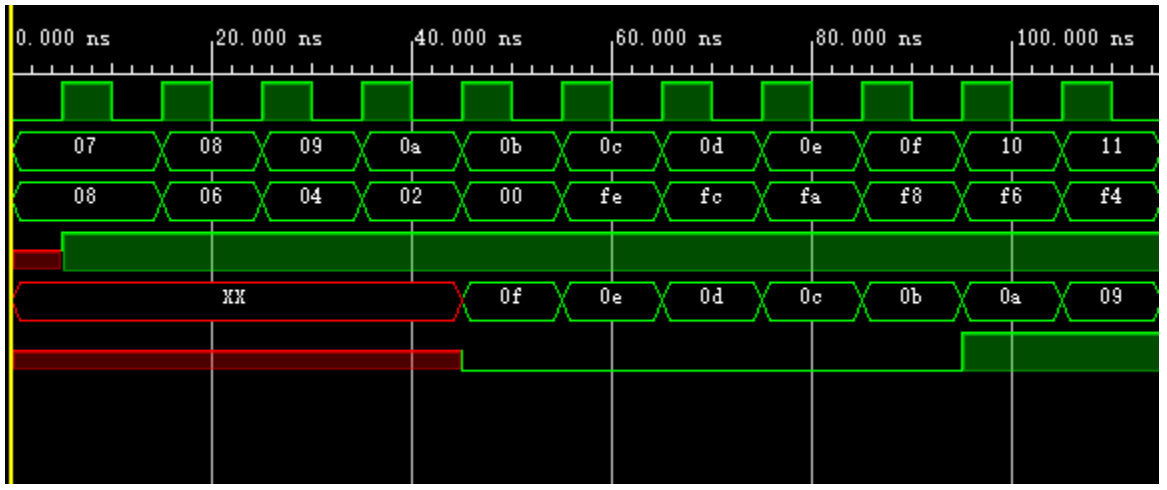
Verilog

- 测试代码

```
`timescale 1ns / 1ps


module fullAdder_tb;

    // Parameters
```

```
    //Ports
    reg  clk;
    reg [7:0] A;
    reg [7:0] B;
    wire  out_en;
    wire [7:0] res;
    wire  C;

    initial begin
        A=7;
        B=8;
        clk=0;
    end
    always @(posedge clk) begin
        #10;
        A=A+1;
        B=B-2;
    end
    lab1_8bit_adder  lab1_8bit_adder_inst (
        .clk(clk),
        .A(A),
        .B(B),
        .out_en(out_en),
        .res(res),
        .C(C)
    );

    always #5  clk = ! clk ;

endmodule
```

Verilog



- 四级流水的八位全加器实现——阻塞
  - 设计代码

```
`timescale 1ns / 1ps

module lab1_fullAdder_clog(
        input clk,
        input rst,
        input stop,

        input validin,
        input allowout,

        input [7:0]A,
        input [7:0]B,

        output validout,
        output reg [7:0]res,
        output reg C
);

    wire pipe1_ready_go,pipe2_ready_go,pipe3_ready_go,pipe4_ready_go;
    wire pipe1_allowin,pipe2_allowin,pipe3_allowin,pipe4_allowin;
    wire pipe1_to_pipe2_valid,pipe2_to_pipe3_valid,pipe3_to_pipe4_valid;
```

```
reg valid_1;
reg [1:0]res_1;
reg c1;
reg [7:0]input_1_A;
reg [7:0]input_1_B;

assign pipe1_ready_go=~stop;
assign pipe1_allowin=~valid_1|pipe1_ready_go&&pipe2_allowin;
assign pipe1_to_pipe2_valid=pipe1_ready_go&&valid_1;

always @(posedge clk) begin
    if (~rst) begin
        valid_1<=1'b0;
    end
    if (pipe1_allowin) begin
        valid_1<=validin;
        input_1_A<=A;
        input_1_B<=B;
    end
    if (pipe1_allowin&&validin) begin
        {c1,res_1}<={1'b0,input_1_A[1:0]}+{1'b0,input_1_B[1:0]};
    end
end

reg valid_2;
reg [3:0]res_2;
reg c2;
reg [7:0]input_2_A;
reg [7:0]input_2_B;

assign pipe2_ready_go=~stop;
assign pipe2_allowin=~valid_2|pipe2_ready_go&&pipe3_allowin;
```

```
    assign pipe2_to_pipe3_valid=valid_2&&pipe2_ready_go;
    always @(posedge clk) begin
        if (~rst) begin
            valid_2<=1'b0;
        end
        if (pipe2_allowin) begin
            valid_2<=pipe1_to_pipe2_valid;
            input_2_A<=input_1_A;
            input_2_B<=input_1_B;
        end
        if (pipe1_to_pipe2_valid&&pipe2_allowin) begin
            {c2,res_2}<={{1'b0,input_2_A[3:2]}+{1'b0,input_2_B[3:2]}+c1,res_1};
        end
    end

    reg valid_3;
    reg [5:0]res_3;
    reg c3;
    reg [7:0]input_3_A;
    reg [7:0]input_3_B;

    assign pipe3_ready_go=~stop;
    assign pipe3_allowin=~valid_3|pipe3_ready_go&&pipe4_allowin;
    assign pipe3_to_pipe4_valid=valid_3&&pipe3_ready_go;
    always @(posedge clk) begin
        if (~rst) begin
            valid_3<=1'b0;
        end
        if (pipe3_allowin) begin
            valid_3<=pipe2_to_pipe3_valid;
            input_3_A<=input_2_A;
            input_3_B<=input_2_B;
        end
```

```
        if (pipe2_to_pipe3_valid&&pipe3_allowin) begin
            {c3,res_3}<={{1'b0,input_3_A[5:4]}+{1'b0,input_3_B[5:4]}+c2,res_2};
        end
    end

    reg valid_4;
    reg [7:0]input_4_A;
    reg [7:0]input_4_B;

    assign pipe4_ready_go=~stop;
    assign pipe4_allowin=~valid_4|pipe4_ready_go&&allowout;
    assign validout=valid_4&&pipe4_ready_go;
    always @(posedge clk) begin
        if (~rst) begin
            valid_4<=1'b0;
        end
        if (pipe4_allowin) begin
            valid_4<=pipe3_to_pipe4_valid;
            input_4_A<=input_3_A;
            input_4_B<=input_3_B;
        end
        if (pipe4_allowin&&pipe3_to_pipe4_valid) begin
            {C,res}<={{1'b0,input_4_A[7:6]}+{1'b0,input_4_B[7:6]}+c3,res_3};
        end
    end
endmodule
```

Verilog

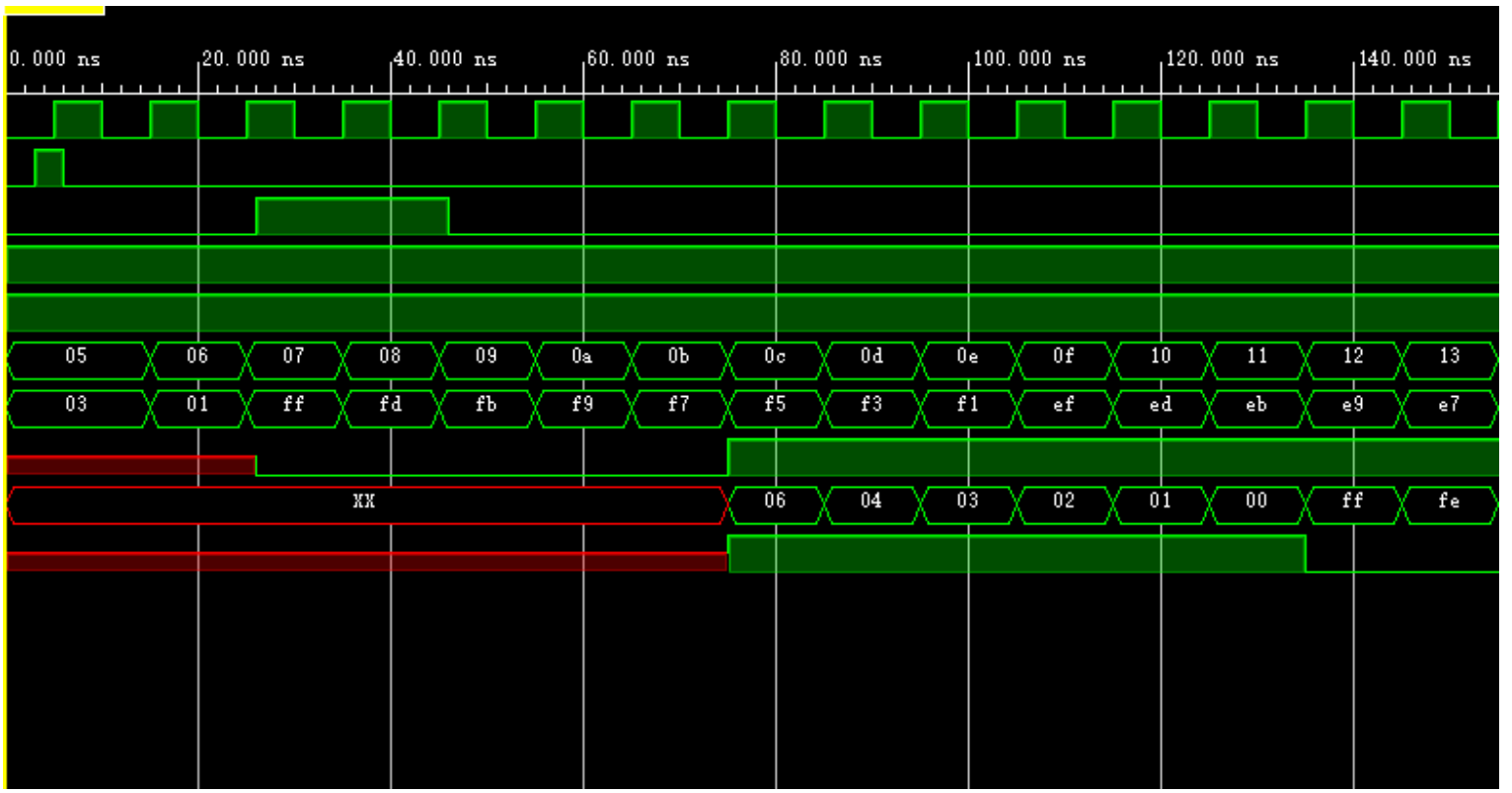
- 测试代码

```
`timescale 1ns / 1ps
```

```
module lab1_fullAdder_clog_tb;

    // Parameters

    //Ports
    reg  clk;
    reg  rst;
    reg  stop;
    reg  validin;
    reg  allowout;
    reg [7:0] A;
    reg [7:0] B;
    wire  validout;
    wire [7:0] res;
    wire  C;

    initial begin
        clk=0;
        rst=0;

        A  =  5;
    B  =  3;

        validin    =  1'b1;  // 可以读入
    allowout  =  1'b1;  // 可以流出
    stop    =  1'b0;    // 没有暂停

        #3 rst=1;
        #3 rst=0;

        #20 stop=1;
        #20 stop=0;
```

```
end
always @(posedge clk) begin
    #10;
    A=A+1;
    B=B-2;
end

lab1_fullAdder_clog  lab1_fullAdder_clog_inst (
    .clk(clk),
    .rst(rst),
    .stop(stop),
    .validin(validin),
    .allowout(allowout),
    .A(A),
    .B(B),
    .validout(validout),
    .res(res),
    .C(C)
);

    always #5  clk = ! clk ;

endmodule
```

Verilog



存储器IP设计

https://blog.csdn.net/u010594449/article/details/106245700书签：手撸MIPS32——5、利用Vivado IP设计指令存储器和数据存储器_vivado添加数据存储器_迷路的小黑的博客-CSDN博客

简单的存储器IP实例化

实验要求

本次实验使用 Vivado 的 Block Memory Generator 模拟数据在存储器中的存取过程。实验使用**单端口 ROM**。初始化 ROM 存储器中的内容，通过开关选择相应的地址，将对应的存储器中内容读出来，并通过七段数码管显示。实验原理如图 1.3 所示：

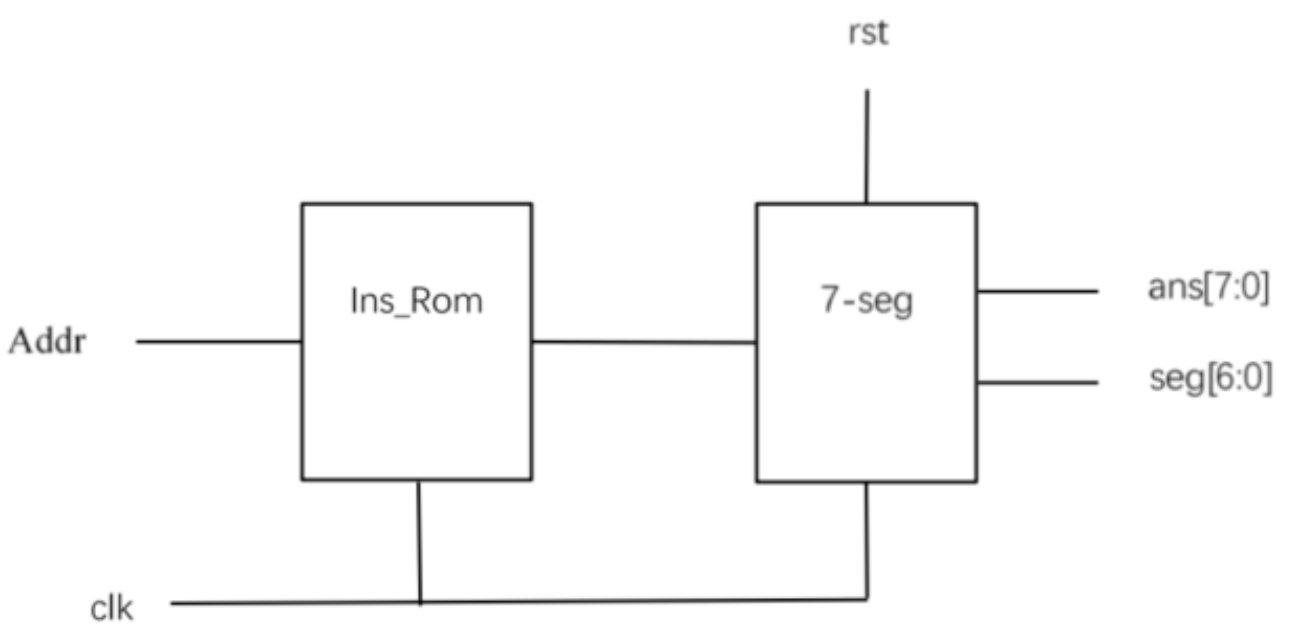


图 1.3

顶层模块输入addr和clk、rst，输出数码管的段选码和位选码

insRom模块输入addr和clk，输出读出的数据

数码管显示模块同之前的ALU实验

新建存储器IP的步骤

- 新建存储器IP的步骤

1. 新建IP

   查找路径是在导航栏的**Window>IP Catalog的搜索栏中搜索Block Memory Generator**

2. 设置RAM参数

Basic 需要设置存储器类型，**Interface Type 需选择 Native，选中 Generate address interface with 32bits**，将地址长度设置为 32 位，Memory Type 根据实验要求——“**单端口**”选择，其他选项无需设置。

端口设置需要设置数据字宽度及阵列深度，根据实验要求，**字宽均为 32 位， 阵列深度需根据需求自定义，但不可超过 155520 字**。

读写端口均默认使能，可根据自己需求选择 Enable Port Type。

其他设置主要用于加载 coe 文件，需要**勾选“Load Init File”**，并 选中需要装载的初始化文件(.coe 文件)。 .coe 文件为 Vivado 中存储器初始化 文件，其格式如下：

```
memory_initialization_radix = 16;
memory_initialization_vector =
24010001,
00011100,
.....
```

Verilog

还**需要选中Fill Remaining Memory Locations 以防读数据操作时**，地址超过 coe 文件已有数据范围，导致异常

3. 之后就可以将其当作一个模块来使用，如下

   clka、ena、addra、douta

```
blk_mem_gen_0 Ins_Rom(
    .clka(clk),          // input wire clka
    .ena(ena),          //en使能
    .addra(addr),      // input wire addra
    .douta(data[31:0])          // output wire [31 : 0] douta
);
```

设计代码

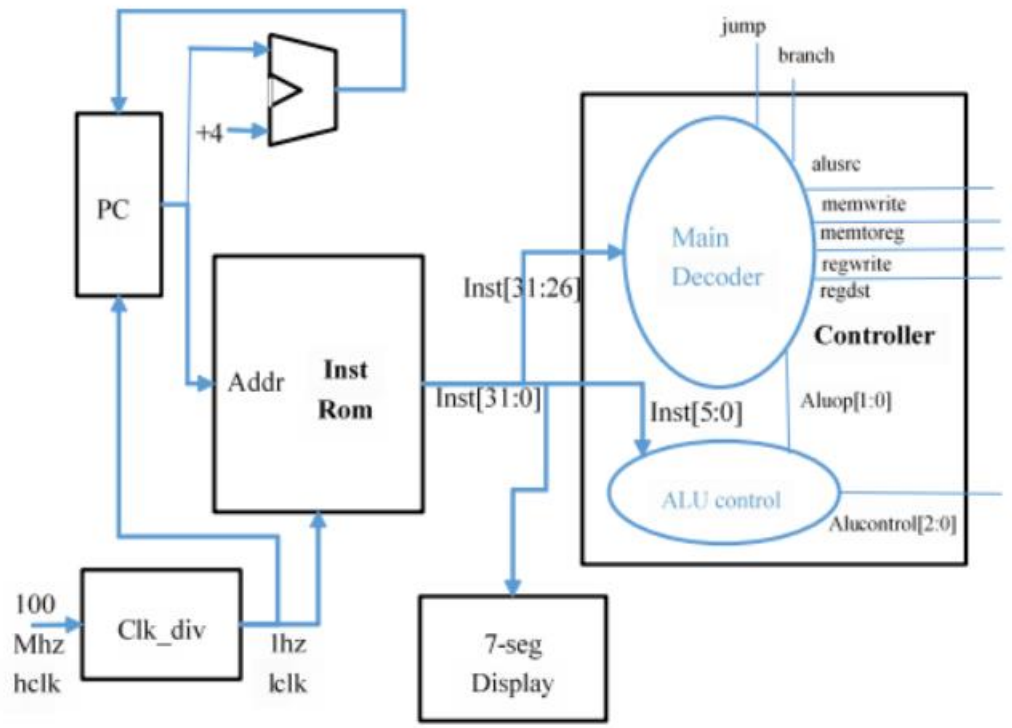
```
`timescale 1ns / 1ps

module memRead(
    input [31:0] addr,
    input clk,
    input reset,
```

```
    output [6:0] seg,
    output [7:0] ans
);
    wire [31:0] data;
    wire ena=1;
    blk_mem_gen_0 Ins_Rom(
        .clka(clk),         // input wire clka
        .ena(ena),         //en使能
        .addra(addr),     // input wire addra
        .douta(data)         // output wire [31 : 0] douta
    );
    digitshow  digitshow_inst (
        .clk(clk),
        .rst(reset),
        .res(data),
        .loc_sel(ans),
        .seg_sel(seg)
  );
endmodule
```

Verilog

取指译码模块设计

实验要求



本次实验包括以下模块：

- 顶层模块：按照上图的连接关系，链接各个模块
- PC：D触发器结构，三个输入clk、rst、指令地址，两个输出addr和ena，addr位数由coe中的指令数决定（这里先统一为32位）
- 加法器：计算下一条指令的地址，两个输入PC和32'h4，输出是输入的和
- 指令存储器IP，输入是clk、ena、addr，然后32位的输出

  与之前存储器IP设计不同的地方

  注意： Basic 中 Generate address interface with 32 bits 选项不选中

  PortA Options 中 Enable Port Type 选择为 Use ENA Pin
- 时钟分频器，将100Mhz降低为1hz，输出给PC和指令存储器——采用循环次数降频
- 7段数码管——显示读出的指令➡这里直接输出指令值吧
- 控制器
  - MainDecoder：根据指令的高6位，输出多个控制位和两位的aluop
  - alu_control：输入是指令的低5位和aluop，输出是3位的alu操作码

实现

1. PC模块

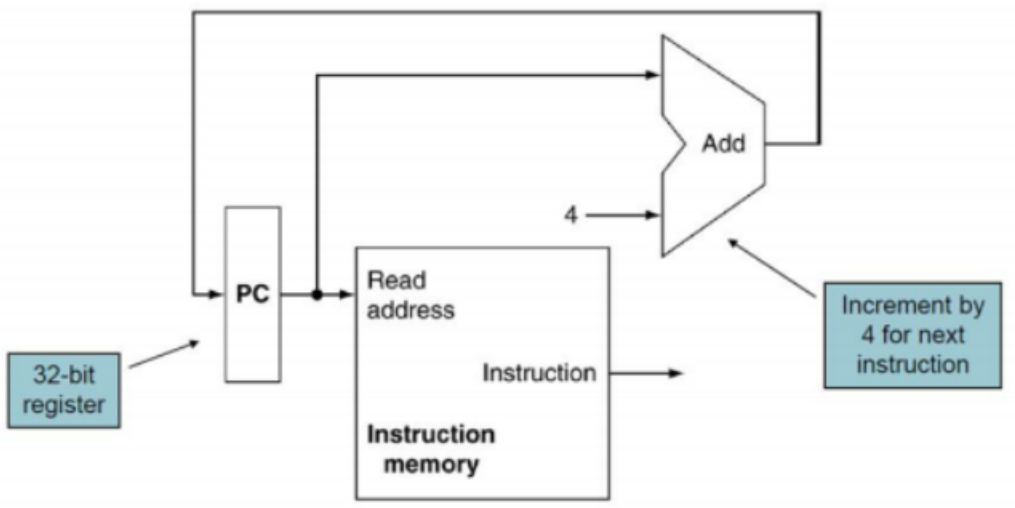


图 2.2

三个输入clk、rst和i_addr，两个输出o_addr、ena

```
`timescale 1ns / 1ps

module pc(
    input clk,
    input rst,
    input [31:0] i_addr,

    output reg[31:0]o_addr,
    output reg ena
);
    always @(posedge clk) begin
        if (~rst) begin
            o_addr=0;
            ena=0;
        end
        else begin
            o_addr=i_addr;
            ena=1;
        end
```

```
    end
endmodule
```

Verilog

2. PC_ADD模块

接收当前PC，加4生成下一个PC

```
`timescale 1ns / 1ps

module pc_adder(
    input [31:0]i_addr,
    output [31:0]o_addr
);
    assign o_addr=i_addr+32'h4;
endmodule
```

Verilog

3. 时钟分频器

新时钟频率1HZ是原来100MHZ的差不多2^28倍，则将时钟延长

每到2^28再赋新时钟

```
`timescale 1ns / 1ps
/*
一个clk周期100MHz的时间是1ns，为此，需要将clk信号分频。
此处，将时钟信号分频至1Hz左右，即延长2^28倍的时间，大概在2.68s左右，在人眼识别范围内
*/
module clk_div(
    input hclk,
    input rst,
    output reg lclk
);
    reg [27:0]count;
```

```verilog
    always @(posedge hclk) begin
        if (~rst) begin
            lclk=0;
            count=0;
        end
        else begin
            if (count[27]==1) begin
                lclk=~lclk;
                count=0;
            end
            else begin
                count=count+1;
            end
        end
    end
endmodule
```

Verilog

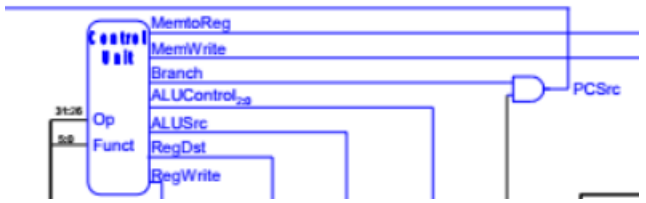
4. 存储器IP

构建步骤同之前的“简单的存储器IP实例化”，只不过这里不再设置读使能信号

coe文件内容如下：

```
memory_initialization_radix = 16;
memory_initialization_vector =
00000000,
8d410000,
ad410000,
112a004c,
21490002,
0810004f,
014b4822,
3549000c,
00000000
```

5. controler模块



controler模块是mainDecoder和alu_control的顶层模块，代码如下：

```
`timescale 1ns / 1ps

module control_top(
    input [5:0]op,alu_funct,
    input isZero,

    output  memtoreg,//写寄存器的结果来自mem还是alu
    output  memwrite,//是否写存储
    output  pcsrc,//下一个PC的值是PC+4还是跳转后的地址0表示src，1表示跳转——由branch
    output  alusrc,//aluB的源操作数是立即数还是寄存器读出的0表示寄存器1表示立即数
    output  regdst,//写入寄存器堆的地址是rt还是rd,0是rt，1是rd
    output  regwrite,//是否写寄存器
    output  branch,//是否是分支指令，且满足分支条件
    output  jump,//是否是无跳转指令
    output  [2:0]control
);
    wire [1:0]aluop;
    main_decoder  main_decoder_inst (
        .ins_func(op),
        .isZero(isZero),
        .memtoreg(memtoreg),
        .memwrite(memwrite),
        .pcsrc(pcsrc),
        .alusrc(alusrc),
```

```
        .regdst(regdst),
        .regwrite(regwrite),
        .branch(branch),
        .jump(jump),
        .aluop(aluop)
    );

    alu_control  alu_control_inst (
        .aluop(aluop),
        .alu_func(alu_funct),
        .control(control)
  );
endmodule
```

Verilog

● mainDecoder模块

mainDecoder这里接受指令的高6位，然后生成多个控制位以及alu的控制字段

首先介绍控制位的具体意义：

```
memtoreg,//写寄存器的结果来自mem还是alu

memwrite,//是否写存储

pcsrc,//下一个PC的值是PC+4还是跳转后的地址pcBranch

alusrc,//aluB的源操作数是立即数还是寄存器读出的

regdst,//写入寄存器堆的地址是rt还是rd,0是rt，1是rd

regwrite,//是否写寄存器

branch,//是否是分支指令，且满足分支条件

jump,//是否是无跳转指令

[1:0] aluop//alu控制
```

Verilog

这里按照下表来生成各个控制位

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000	1	0	1	0	0	0	00
j	000100 000010	0	X	X	X	0	X	XX

```
`timescale 1ns / 1ps

module main_decoder(
    input [5:0]ins_func,
    input isZero,//alu比较操作数输入到controler, 结合branch生成pcsrc

    output  memtoreg,//写寄存器的结果来自mem还是alu
    output  memwrite,//是否写存储
    output  pcsrc,//下一个PC的值是PC+4还是跳转后的地址0表示src, 1表示跳转——由branch
    output  alusrc,//aluB的源操作数是立即数还是寄存器读出的0表示寄存器1表示立即数
    output  regdst,//写入寄存器堆的地址是rt还是rd,0是rt, 1是rd
    output  regwrite,//是否写寄存器
    output  branch,//是否是分支指令, 且满足分支条件
    output  jump,//是否是无跳转指令
    output  [1:0] aluop//alu控制

    //其中无影响的设为0
);
```

```
    //按照上述控制位的顺序生成等位宽的变量
    wire [8:0]control;
    assign control=(ins_func==6'b000000)?9'b000110010:
                   (ins_func==6'b100011)?9'b101010000:
                   (ins_func==6'b101011)?9'bx11x00000:
                   (ins_func==6'b000100)?9'bx00x01001:
                   (ins_func==6'b001000)?9'b001010000:
                   (ins_func==6'b000010)?9'bx0xx0x1xx:9'bxxxxxxxxx;
    assign {memtoreg,memwrite,alusrc,regdst,regwrite,branch,jump}=control[8:2];
    assign aluop=control[1:0];
    assign pcsrc=branch&isZero;

endmodule
```

Verilog

● aluDecoder模块

aluDecoder模块输入是“指令的低六位”以及mainDecoder模块的输出aluop

按照下图输出alucontrol

Opcode	AluOp	Operation	Funct	Alu function	Alu control
Lw	00	Load word	XXXXXX	Add	010
Sw	00	Store word	XXXXXX	Add	010
Beq	01	Branch equal	XXXXXX	Subtact	110
R-type	10	Add	100000	Add	010
		Subtract	100010	Subtract	110
		And	100100	And	000
		Or	100101	Or	001
		Set-on-less-than	101010	SLT	111

```
`timescale 1ns / 1ps

module alu_control(
    input [1:0]aluop,
```

```
    input [5:0]alu_func,

    output reg[2:0]control
);

    always @(*) begin
        case (aluop)
            2'b00:control=3'b010;
            2'b01:control=3'b110;
            2'b10:begin
                case (alu_func)
                    6'b100000: control=3'b010;//ADD
                    6'b100010: control=3'b110;//SUB
                    6'b100100: control=3'b000;//AND
                    6'b100101: control=3'b001;//OR
                    6'b101010: control=3'b111;//SLT
                    default: control=3'b000;
                endcase
            end
            default:  control=3'b000;
        endcase
    end
endmodule
```

Verilog

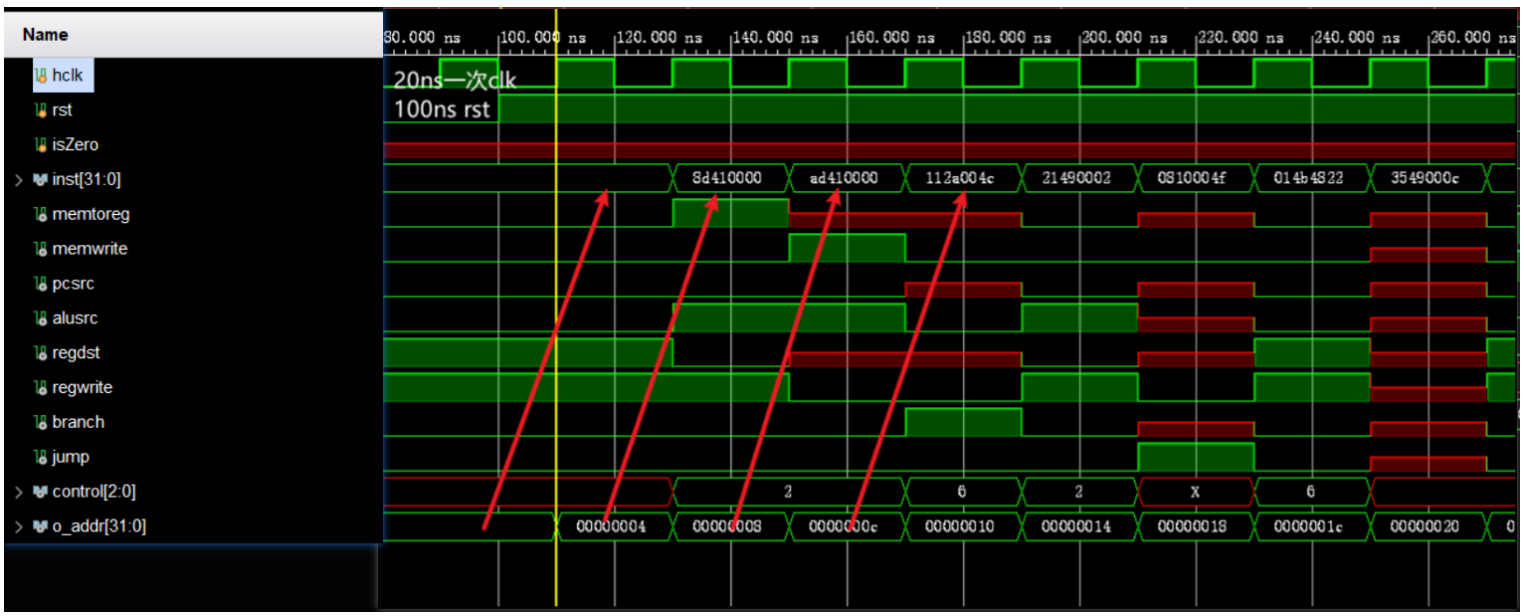
6. 系统顶层模块

```
`timescale 1ns / 1ps
module top(
    input hclk,
    input rst,
    input isZero,
```

```
    output [31:0]inst,
    output  memtoreg,//写寄存器的结果来自mem还是alu
    output  memwrite,//是否写存储
    output  pcsrc,//下一个PC的值是PC+4还是跳转后的地址0表示src，1表示跳转——由branch
    output  alusrc,//aluB的源操作数是立即数还是寄存器读出的0表示寄存器1表示立即数
    output  regdst,//写入寄存器堆的地址是rt还是rd,0是rt，1是rd
    output  regwrite,//是否写寄存器
    output  branch,//是否是分支指令，且满足分支条件
    output  jump,//是否是无跳转指令
    output  [2:0]control
);
    wire clk;
    wire [31:0]i_addr,o_addr;
    wire ena;

    // clk_div  clk_div_inst (
    //     .hclk(hclk),
    //     .rst(rst),
    //     .lclk(clk)
    // );
    pc  pc_inst (
        .clk(hclk),
        .rst(rst),
        .i_addr(i_addr),
        .o_addr(o_addr),
        .ena(ena)
    );
    pc_adder  pc_adder_inst (
        .i_addr(o_addr),
        .o_addr(i_addr)
```

```
    );
    blk_mem_gen_0 blk_mem_gen_1(
        .clka(hclk),
        .ena(ena),
        .addra(o_addr),
        .douta(inst)
    );
    control_top  control_top_inst (
        .op(inst[31:26]),
        .alu_funct(inst[5:0]),
        .isZero(isZero),
        .memtoreg(memtoreg),
        .memwrite(memwrite),
        .pcsrc(pcsrc),
        .alusrc(alusrc),
        .regdst(regdst),
        .regwrite(regwrite),
        .branch(branch),
        .jump(jump),
        .control(control)
    );
endmodule
```

Verilog


Name
hclk
rst
isZero
inst[31:0]
memtoreg
memwrite
pcsrc
alusrc
regdst
regwrite
branch
jump
control[2:0]
o_addr[31:0]
80.000 ns
100.000 ns
120.000 ns
140.000 ns
160.000 ns
180.000 ns
200.000 ns
220.000 ns
240.000 ns
260.000 ns
20ns一次clk
100ns rst
8d410000
ad410000
112a004c
21490002
0810004f
014b4822
3549000c
2
6
2
X
6
00000004
00000008
0000000c
00000010
00000014
00000018
0000001c
00000020