**北京科技大学数字逻辑实验报告**

学院： 计通 专业： 班级：

姓名： 学号： 实验日期： 2024年 10 月 14 日

**1、实验名称：**实验二 组合逻辑实验

**2、实验目的：**

(1) 学习用verilog设计较复杂的组合逻辑电路

(2) 进一步熟悉vivado工具

**3、实验内容：**

1. **实验2.1——CPU译码逻辑模块**
   1. 了解译码逻辑模块的功能；
   2. 编写代码，实现指令解码功能，即将原始指令拆解为不同字段，如操作码，源寄存器，目标寄存器，立即数等；
   3. 在以上代码的基础上，实现识别操作码功能，即确定这条指令要执行什么操作；
   4. 在以上代码的基础上，实现访问寄存器功能，即根据指令要求，从指定的寄存器中读取源寄存器的值；
   5. 在以上代码的基础上，实现生成控制信号功能，即根据操作码和其他字段生成控制信号，而这些信号将在后续的执行、访存和写回阶段发挥作用。
2. **实验2.2——ALU设计**

根据实验手册，完成能满足手册需求的ALU的设计。

1. **实验2.3——加法器设计**

实现32位逐位进位加法器、32位选择进位加法器。

* 1. 学习课件中逐位进位加法器、选择进位加法器的原理；
  2. 新建工程，完成两种加法器的Verilog描述，其中**一位全加器**已经提供（**module add1**在add32\_tb.v文件中已定义）直接使用即可，两种加法器请都使用add1作为基本模块开始搭建；模块名称及端口定义如下：

module csadd32 (a,b,cin,s,cout); //选择进位加法器

module rcadd32 (a,b,cin,s,cout); //逐位进位加法器

* 1. 编写测试激励：可更改add32\_tb.v文件中激励产生代码，按照自己的思路产生测试数据，通过仿真验证加法器功能。

4、**实验2.1（CPU译码逻辑模块）的实现及仿真验证**

（1）自己实现的CPU译码逻辑模块代码（由于我们对该模块的具体实现方式并未做出约束，故该代码不应出现雷同）。

`timescale 1ns / 1ps

module id(

    input wire [15:0] fs\_to\_ds\_bus,

    output wire [27:0] ds\_to\_es\_bus,

    output wire [1:0] rx,

    output wire [1:0] ry,

    input wire [7:0] rx\_value,

    input wire [7:0] ry\_value

);

    wire [3:0] onehot\_output;

    binary\_to\_onehot decoder (

        .binary\_in(fs\_to\_ds\_bus[7:4]),

        .onehot\_out(onehot\_output)

    );

    assign rx = fs\_to\_ds\_bus[1:0];

    assign ry = fs\_to\_ds\_bus[3:2];

    assign ds\_to\_es\_bus[27:24] = onehot\_output;

    assign ds\_to\_es\_bus[23:16] = ry\_value[7:0];

    assign ds\_to\_es\_bus[15:8]  = rx\_value[7:0];

    assign ds\_to\_es\_bus[7:0]   = fs\_to\_ds\_bus[15:8];

endmodule

module binary\_to\_onehot (

    input wire [3:0] binary\_in,

    output reg [3:0] onehot\_out

);

    always @(\*) begin

        case (binary\_in)

            4'b0001: onehot\_out = 4'b1000;

            4'b0010: onehot\_out = 4'b0100;

            4'b0011: onehot\_out = 4'b0010;

            4'b0100: onehot\_out = 4'b0001;

            default: onehot\_out = 4'b0000;

        endcase

    end

endmodule

（2）该模块的仿真验证波形及说明（简要说明testbench设计思路，以及为什么这个结果是正确的）

使用随机数产生不同的信号作为验证，以下是testbench代码。

`timescale 1ns / 1ps

module sim\_1;

reg [15:0] fs\_to\_ds\_bus;

reg [7:0] rx\_value;

reg [7:0] ry\_value;

wire [27:0] ds\_to\_es\_bus;

wire [1:0] rx;

wire [1:0] ry;

id ID(

.fs\_to\_ds\_bus(fs\_to\_ds\_bus),

.ds\_to\_es\_bus(ds\_to\_es\_bus),

.rx(rx),

.ry(ry),

.rx\_value(rx\_value),

.ry\_value(ry\_value)

);

initial begin

forever begin

#20

fs\_to\_ds\_bus[15:8] = $random % 8'b11111111;

fs\_to\_ds\_bus[7:4] = $random % 4'b0100 + 1;

fs\_to\_ds\_bus[3:0] = $random % 4'b1111;

rx\_value[7:0] = $random % 8'b11111111;

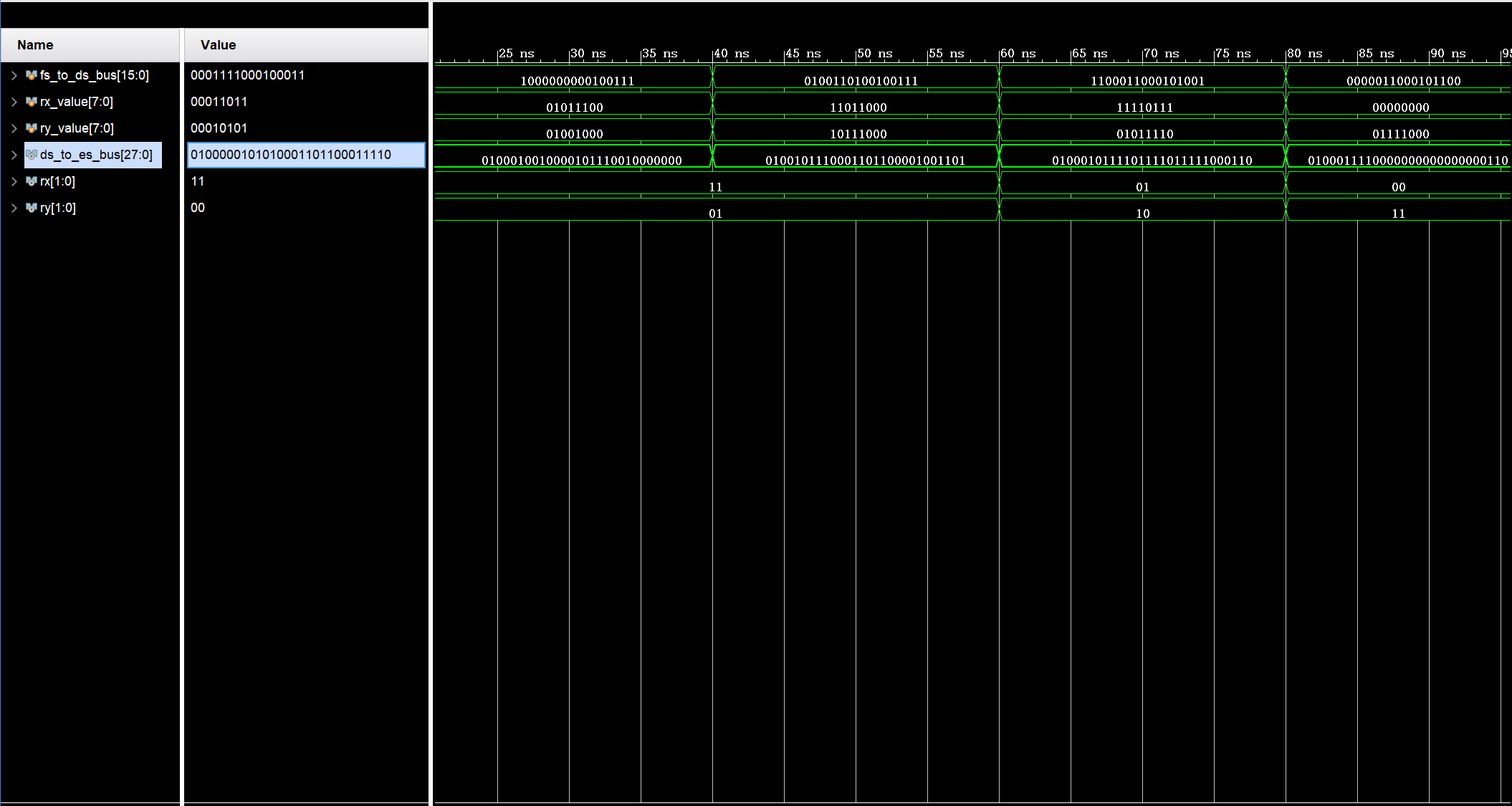
ry\_value[7:0] = $random % 8'b11111111;

end

$stop;

end

endmodule



以第二组随机数据为例，可以看到 fs\_to\_ds\_bus的前八位 01001101 PC码复制到了 ds\_to\_es\_bus 的最后八位；中间4位0010被翻译成独热码0100作为ds\_to\_es\_bus的最高4位；读入的 ry\_value, rx\_value 则作为ds\_to\_es\_bus中间的16位；输出的 ry, rx 则为fs\_to\_ds\_bus 的最后4位。

5、**实验2.2（ALU设计）的实现及验证**

（1）自己实现的CPU译码逻辑模块代码（由于我们对该模块的具体实现方式并未做出约束，故该代码不应出现雷同）。

`timescale 1ns / 1ps

module ALU (

    input wire [7:0] alu\_src1,

    input wire [7:0] alu\_src2,

    input wire [11:0] alu\_op,

    output wire [7:0] alu\_result

);

    reg [7:0] tmp\_result;

    reg [3:0] tmp\_op;

    reg [8:0] sum;

    integer counter, i;

    always @(\*) begin

        case (alu\_op)

            12'h001: tmp\_result = alu\_src1 + alu\_src2;

            12'h002: tmp\_result = alu\_src1 - alu\_src2;

            12'h004: tmp\_result = alu\_src1 & alu\_src2;

            12'h008: tmp\_result = (alu\_src1 || alu\_src2) ? 8'h01 : 8'h00;

            12'h010: tmp\_result = alu\_src1 << alu\_src2[1:0];

            12'h020: tmp\_result = alu\_src1 >>> alu\_src2[1:0];

            12'h040: begin

                tmp\_result = alu\_src1;

                counter = alu\_src2[1:0];

                for (i = 0; i < counter ; i = i + 1) begin

                    tmp\_result = {tmp\_result[0], tmp\_result[7:1]};

                end

            end

            12'h080: tmp\_result = ($signed(alu\_src1) < $signed(alu\_src2)) ? 8'h01 : 8'h00;

            12'h100: tmp\_result = alu\_src1 < alu\_src2 ? 8'h01 : 8'h00;

            12'h200: begin

                sum = alu\_src1 + alu\_src2;

                if(sum[8])

                    tmp\_result = sum[8:1];

                else

                    tmp\_result = sum[7:0];

            end

            12'h400: tmp\_result = alu\_src1 ^ alu\_src2;

            12'h800: begin

                if(alu\_src2[4]) begin

                    tmp\_result[3:0] = {alu\_src1[1:0], alu\_src1[7:6]};

                    tmp\_op = alu\_src1[5:2];

                end else if(alu\_src2[5]) begin

                    tmp\_result[3:0] = {alu\_src1[5:4], alu\_src1[3:2]};

                    tmp\_op = {alu\_src1[7:6], alu\_src1[1:0]};

                end else if(alu\_src2[6]) begin

                    tmp\_result[3:0] = {alu\_src1[7:6], alu\_src1[3:2]};

                    tmp\_op = {alu\_src1[5:4], alu\_src1[1:0]};

                end else if(alu\_src2[7]) begin

                    tmp\_result[3:0] = {alu\_src1[5:4], alu\_src1[1:0]};

                    tmp\_op = {alu\_src1[7:6], alu\_src1[3:2]};

                end

                tmp\_result[7:4] = alu\_src2[3:0];

                if(tmp\_op[0]) begin

                    counter = tmp\_op[3:1];

                    for (i = 0; i < counter ; i = i + 1) begin

                        tmp\_result = {tmp\_result[6:0], tmp\_result[7]};

                    end

                end else begin

                    counter = tmp\_op[3:1];

                    for (i = 0; i < counter ; i = i + 1) begin

                        tmp\_result = {tmp\_result[0], tmp\_result[7:1]};

                    end

                end

            end

            default: tmp\_result = 8'h00;

        endcase

    end

    assign alu\_result = tmp\_result;

endmodule

（2）该模块的仿真验证波形及说明（简要说明testbench设计思路，以及为什么这个结果是正确的）

使用随机数产生不同的两个操作数和一个操作符作为验证，因为 alu\_op是独热码的形式，因此初始将其设置为1，每次测试左移一位即可，以下是testbench代码。

`timescale 1ns / 1ps

module sim\_1;

  reg [7:0] alu\_src1;

  reg [7:0] alu\_src2;

  reg [11:0] alu\_op;

  wire [7:0] alu\_result;

  ALU alu(

    .alu\_src1(alu\_src1),

    .alu\_src2(alu\_src2),

    .alu\_op(alu\_op),

    .alu\_result(alu\_result)

  );

  initial begin

    forever begin

      alu\_src1 = $random % 8'hFF;

      alu\_src2 = {$random % 4'hF + 1, $random % 4'hF};

      alu\_op = 12'b0000\_0000\_0001;

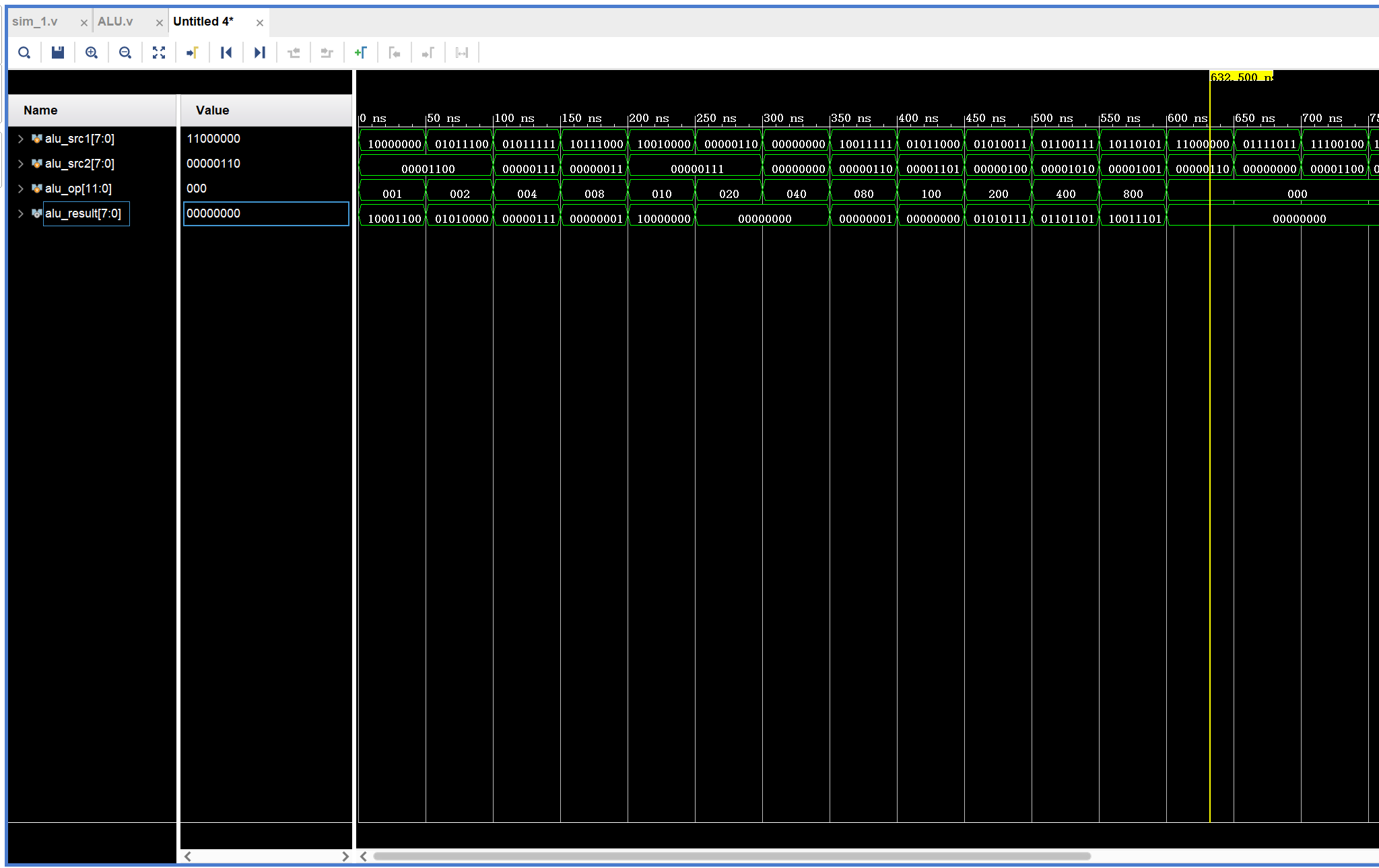
      #50

      alu\_op = alu\_op << 1;

    end

  end

endmodule



分12种情况实现了ALU的实验要求。

（3）板级测试验证（非强制）

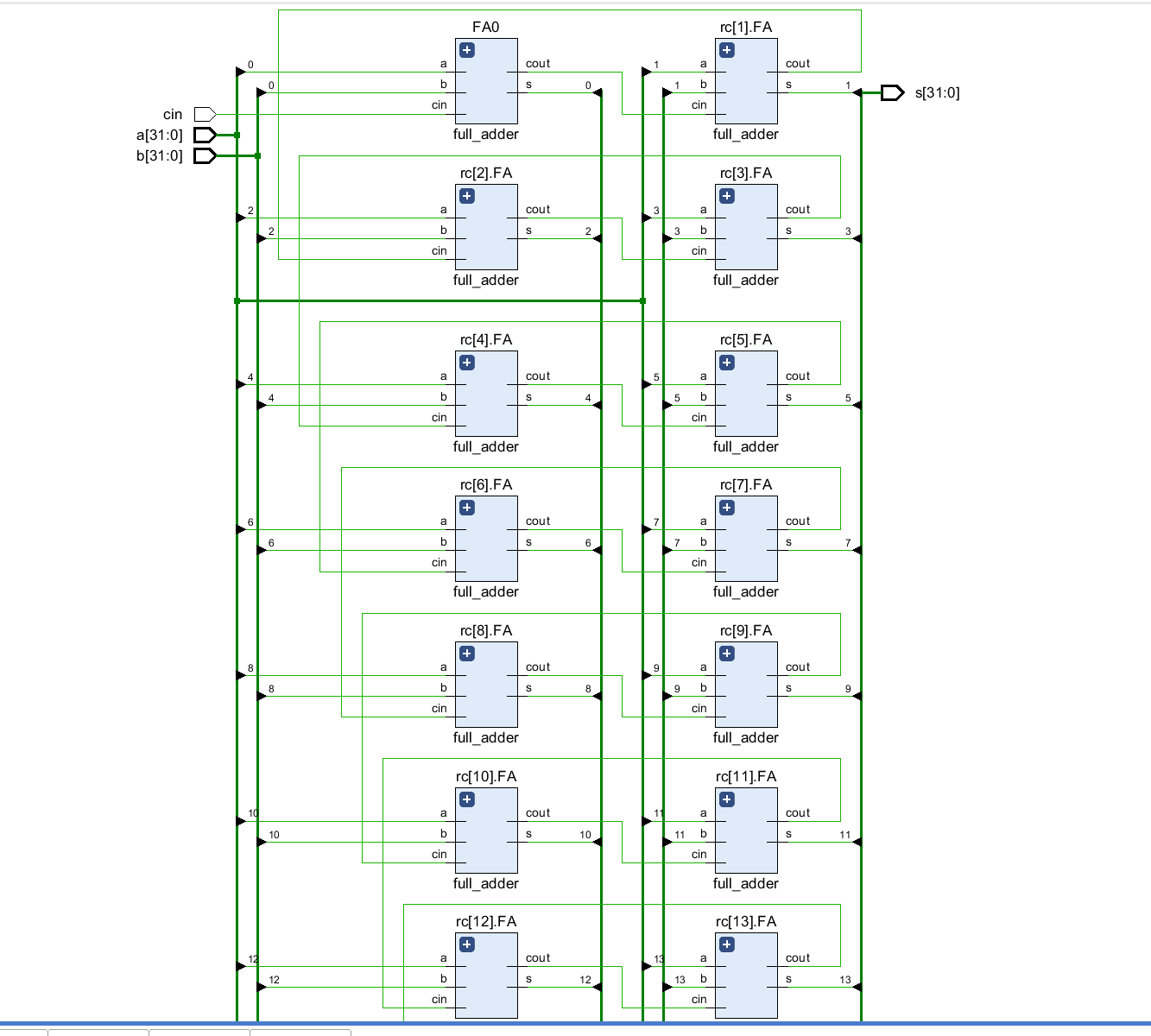
（给出下载到实验板后的现象照片。注意照片不要占篇幅太大，以能看清主要信息为好，并有相应文字对图进行解释说明）

6、**实验2.3（加法器设计）的实现及仿真验证**

（1）32位逐位进位加法器模块RTL分析结构图及说明

先实现了一个全加器，之后每一位都使用上一位的进位结果和两个数这一位的值作为全加器的输入，全加器的输出继续输入给下一位，如此循环32次就完成了一个32位的逐位进位加法器。

RTL分析如下图（太长了所以只展示部分，剩下的部分几乎完全一样）：

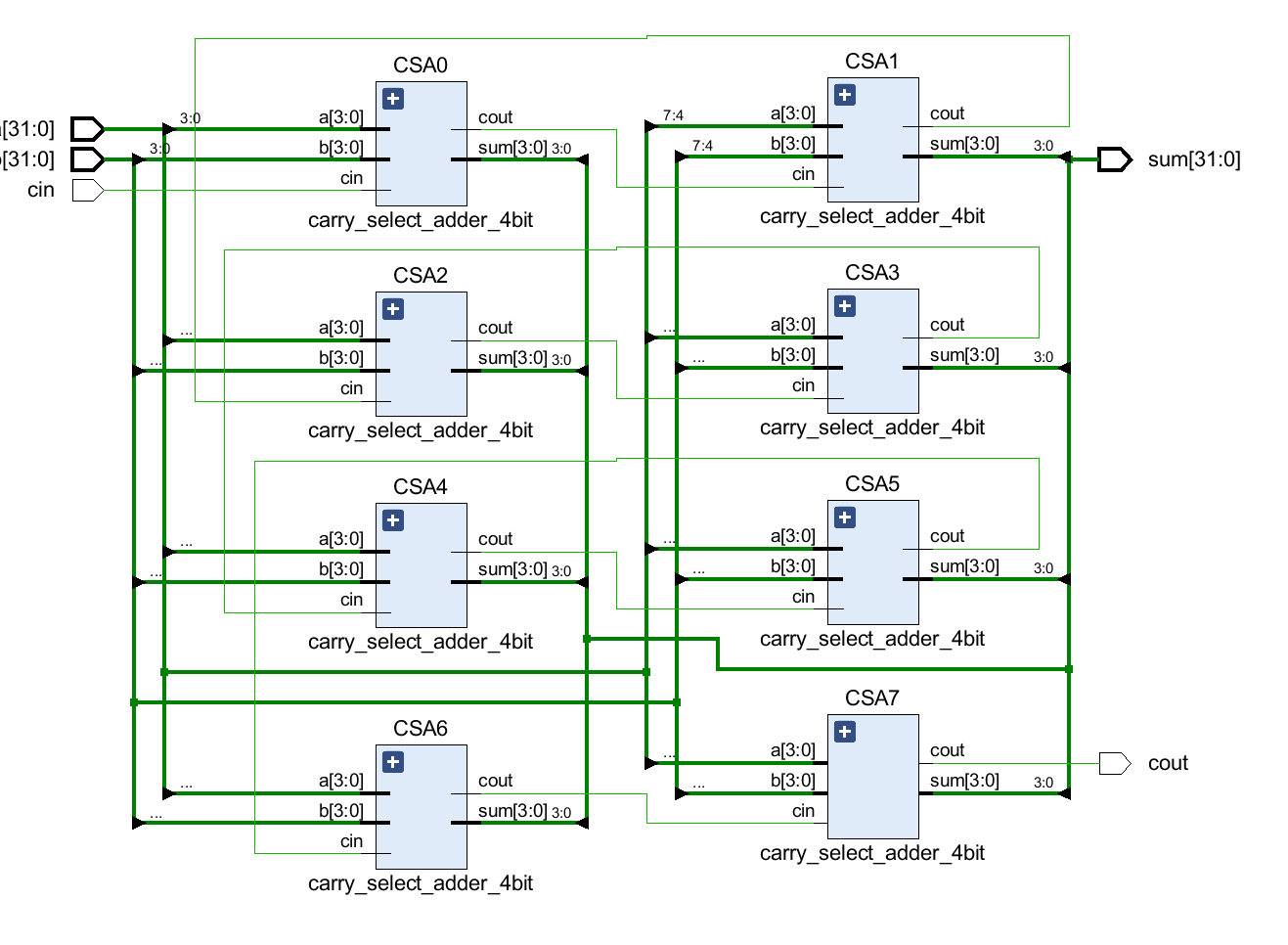


（2）32位选择进位加法器模块RTL分析结构图及说明（可以和（1）写在一起，则跳过本步骤）

我把32位的加法器拆分成8个4位的加法器，每一个4位加法器都分别计算假设上一个4位加法器进位是1和进位是0的两种结果，根据实际的进位选择正确的结果。

每个4位加法器由4个1位全加器拼起来。

RTL 分析如下图（分别展示了8个4位加法器组合整体电路；如何用两个4位加法器分别计算进位1和进位0的两个不同结果并选择正确结果；4位加法器内部电路）：



图示

描述已自动生成

图示, 示意图

描述已自动生成

（3）测试激励的设计（说明：测试方案，包括数据如何产生、测试哪些情况、如何判定结果是否正确、如何对比性能等等）

使用随机数产生 a, b 和初始进位 cin 信号作为输入，同时运行两个加法器，输出分别为 s0, cout0和s1, cout1。另外使用verilog中的加法运算设计一个标准加法器作为正确与否的评判，输出为 s2, cout2。

通过观测输入数据后两个加法器的输出答案和标准加法器答案是否相同作为对错标准，输出答案的快慢作为评判性能的标准。

Testbench如下：

module sim\_1;

    reg [31:0] a;

    reg [31:0] b;

    reg cin;

    reg clk;

    wire [31:0] s0, s1, s2;

    wire cout0, cout1, cout2;

    initial begin

        a = 4'bxxxx;

        b = 4'bxxxx;

        cin = 1'bx;

        clk = 0;

    end

    always #100 clk = ~clk;

    always@ (posedge clk) begin

        a = $random % (2 \*\* 30);

        b = $random % (2 \*\* 30);

        cin = $random % 2;

        #150

        cin = 1'bx;

    end

    csadd32 A(a, b, cin, s0, cout0);//选择进位加法器

    rcadd32 B(a, b, cin, s1, cout1);//逐位进位加法器

    vadd32  C(a, b, cin, s2, cout2);//标准加法器

endmodule

（4）仿真波形及说明（功能验证+性能对比）

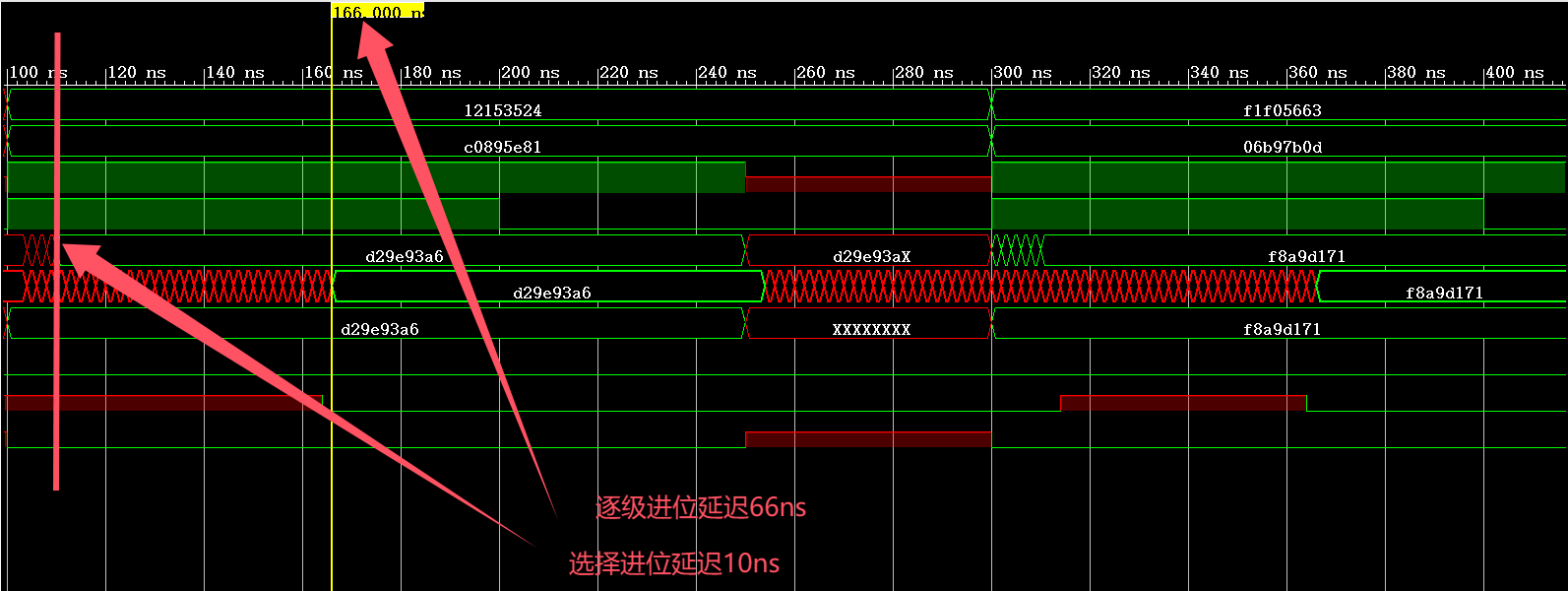
可以看到两个加法器的输出和标准加法器的输出均相同，这说明两个加法器都是正确的。

逐级进位加法器比选择进位加法器明显速度要慢。

延迟分析：

因为使用的加法器进位有 2ns 的延迟，计算有4ns的延迟，逐级进位加法器进位要传递31次，延迟为31\*2=62ns，最后一位计算需要4ns，所以总延迟为66ns。选择进位加法器先花了3次进位和1次计算的时间算好了每4位的答案，延迟为3\*2+4=10ns，最后根据进位情况直接选择每4位的答案，这个过程不耗时间（其实按道理应该消耗2ns左右？但是我的代码里没有体现出来选择耗时这一点），所以总延迟为10ns。

这两个计算出来的延迟速度也符合实际的仿真波形（如下图）。



**7、实验中遇到的问题、现象及解决方法（如没遇到可不写，帮助别人解决的也可以写）**

问题1：（截屏或拍照或文字描述）

现象：

问题原因：

解决方法：

问题2：

**9、本次实验心得体会**

（学到了什么，或理解了什么，或加深了什么的认识等，无需长篇大论）

**10、关于本次实验课程的改进建议**

（没有可以不写）

**11、附录：你所实现的各种32位加法器Verilog代码 + 测试激励代码**

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 2024/10/14 17:07:03

// Design Name:

// Module Name: csadd32

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module csadd32 (

input wire [31:0] a,

input wire [31:0] b,

input wire cin,

output wire [31:0] s,

output wire cout

);

wire [31:0] s0, s1;

wire [7:0] carry;

carry\_select\_adder\_4bit CSA0 (

.a(a[3:0]),

.b(b[3:0]),

.cin(cin),

.s(s[3:0]),

.cout(carry[0])

);

carry\_select\_adder\_4bit CSA1 (

.a(a[7:4]),

.b(b[7:4]),

.cin(carry[0]),

.s(s[7:4]),

.cout(carry[1])

);

carry\_select\_adder\_4bit CSA2 (

.a(a[11:8]),

.b(b[11:8]),

.cin(carry[1]),

.s(s[11:8]),

.cout(carry[2])

);

carry\_select\_adder\_4bit CSA3 (

.a(a[15:12]),

.b(b[15:12]),

.cin(carry[2]),

.s(s[15:12]),

.cout(carry[3])

);

carry\_select\_adder\_4bit CSA4 (

.a(a[19:16]),

.b(b[19:16]),

.cin(carry[3]),

.s(s[19:16]),

.cout(carry[4])

);

carry\_select\_adder\_4bit CSA5 (

.a(a[23:20]),

.b(b[23:20]),

.cin(carry[4]),

.s(s[23:20]),

.cout(carry[5])

);

carry\_select\_adder\_4bit CSA6 (

.a(a[27:24]),

.b(b[27:24]),

.cin(carry[5]),

.s(s[27:24]),

.cout(carry[6])

);

carry\_select\_adder\_4bit CSA7 (

.a(a[31:28]),

.b(b[31:28]),

.cin(carry[6]),

.s(s[31:28]),

.cout(cout)

);

endmodule

module carry\_select\_adder\_4bit (

input wire [3:0] a,

input wire [3:0] b,

input wire cin,

output wire [3:0] s,

output wire cout

);

wire [3:0] s0, s1;

wire cout0, cout1;

ripple\_carry\_adder\_4bit RCA0 (

.a(a),

.b(b),

.cin(1'b0),

.s(s0),

.cout(cout0)

);

ripple\_carry\_adder\_4bit RCA1 (

.a(a),

.b(b),

.cin(1'b1),

.s(s1),

.cout(cout1)

);

assign s = (cin) ? s1 : s0;

assign cout = (cin) ? cout1 : cout0;

endmodule

module ripple\_carry\_adder\_4bit (

input wire [3:0] a,

input wire [3:0] b,

input wire cin,

output wire [3:0] s,

output wire cout

);

wire [3:0] carry;

full\_adder FA0 (

.a(a[0]),

.b(b[0]),

.cin(cin),

.s(s[0]),

.cout(carry[0])

);

genvar i;

generate

for (i = 1; i < 4; i = i + 1) begin : ripple\_carry

full\_adder FA (

.a(a[i]),

.b(b[i]),

.cin(carry[i-1]),

.s(s[i]),

.cout(carry[i])

);

end

endgenerate

assign cout = carry[3];

endmodule

module rcadd32 (

input wire [31:0] a,

input wire [31:0] b,

input wire cin,

output wire [31:0] s,

output wire cout

);

wire [31:0] carry;

full\_adder FA0 (

.a(a[0]),

.b(b[0]),

.cin(cin),

.s(s[0]),

.cout(carry[0])

);

genvar i;

generate

for (i = 1; i < 32; i = i + 1) begin : rc

full\_adder FA (

.a(a[i]),

.b(b[i]),

.cin(carry[i-1]),

.s(s[i]),

.cout(carry[i])

);

end

endgenerate

assign cout = carry[31];

endmodule

module full\_adder (

input wire a,

input wire b,

input wire cin,

output wire s,

output wire cout

);

assign #4 s = a ^ b ^ cin;

assign #2 cout = (cin == 1) | (cin == 0) ? (a & cin) | (b & cin) | (a & b) : 1'bx;

endmodule

module vadd32(

input wire [31:0] a,

input wire [31:0] b,

input wire cin,

output wire [32:0] s,

output wire cout

);

wire [32:0] sum;

assign sum = a + b + cin;

assign cout = sum[32];

assign s = sum[31:0];

endmodule

测试激励代码上面已经给出，这里不再重复。