

**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**

- a) 了解译码逻辑模块的功能;
- b) 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;
- c) 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;
- d) 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;
- e) 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。

**(3) 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。

- a) 学习课件中逐位进位加法器、选择进位加法器的原理;
- b) 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
```

- c) 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。

### 实验步骤：

#### 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)。

代码如下：

```
cpuDecode.v
`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
);
reg [3:0] op;
always @(*) begin
    case (fs_to_ds_bus[7:4])
        4'b0001: op = 4'b1000; // MOV
        4'b0010: op = 4'b0100; // ADD
        4'b0011: op = 4'b0010; // SUB
        4'b0100: op = 4'b0001; // MUL
        default: op = 4'b0000; // Default
    endcase
end
assign {ds_to_es_bus,ry,rx} = {op, ry_value, rx_value,
    fs_to_ds_bus[15:8],fs_to_ds_bus[3:2],fs_to_ds_bus[1:0]};
endmodule
```

以上代码实现了译码的功能，但是接口过多，为了在板子上能运行，设计了如下代码（省略了部分代码）。

```
cpuDecodeTest.v
module display(
    input wire [5:0] rx,
    input wire [5:0] ry,
    input wire [3:0] op,
    output wire [7:0] data
);
wire [1:0] rx_addr = 2'b00;
wire [1:0] ry_addr = 2'b01;
```

```

wire [10:0] tmp_data;
wire [7:0] rx_value;
wire [7:0] ry_value;
assign rx_value = {2'b0, rx};
assign ry_value = {2'b0, ry};
reg [7:0] pc = 8'b00000000;
id id_instance(
    .fs_to_ds_bus({pc, op, ry_addr, rx_addr}),
    .ds_to_es_bus(),
    .rx(rx_addr),
    .ry(ry_addr),
    .rx_value(rx_value),
    .ry_value(ry_value),
    .rf_bus(tmp_data)
);
assign data = tmp_data[7:0];
endmodule
module id(
    .....
    output wire [10:0] rf_bus
);
reg [7:0] write_data; reg write_enable;
reg [3:0] op;
always @(*) begin
    write_enable = 0; write_data = 8'b00000000;
    case (fs_to_ds_bus[7:4])
        //ALU
        4'b0001: begin
            write_enable = 1; write_data = rx_value; .....
        end
        4'b0010: begin
            write_enable = 1; write_data = rx_value + ry_value; .....
        end
        4'b0011: begin
            write_enable = 1; write_data = rx_value - ry_value; .....
        end
        4'b0100: begin
            write_enable = 1; write_data = rx_value * ry_value; .....
        end
        default: .....
    endcase
end
.....
assign rf_bus = {write_enable, rx, write_data};

```

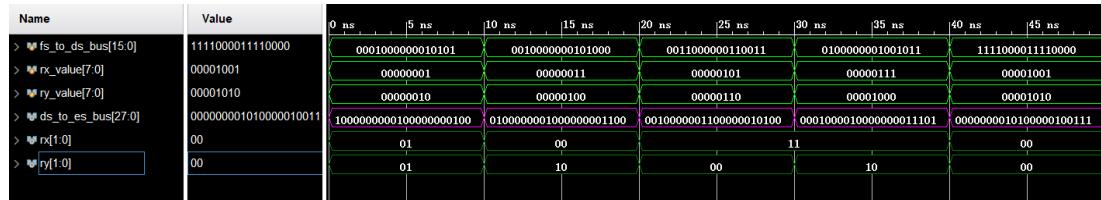
```
endmodule
```

(2) 该模块的仿真验证波形及说明

(下列为部分仿真代码, 针对于 `cpuDecode.v`)

```
fs_to_ds_bus = 16'b00010000_0001_01_01;
rx_value = 8'b00000001; ry_value = 8'b00000010;
#10;
fs_to_ds_bus = 16'b00100000_0010_10_00;
rx_value = 8'b00000011; ry_value = 8'b00000100;
#10;
fs_to_ds_bus = 16'b00110000_0011_00_11;
rx_value = 8'b00000101; ry_value = 8'b00000110;
#10;
fs_to_ds_bus = 16'b01000000_0100_10_11;
rx_value = 8'b00000111; ry_value = 8'b00001000;
#10;
fs_to_ds_bus = 16'b11110000_1111_00_00;
rx_value = 8'b00001001; ry_value = 8'b00001010;
#10;
```

仿真验证波形如下 (二进制数位太多了有一部分没显示出来)

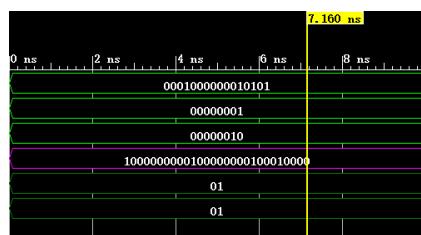


CpuDecode.v 完整的仿真验证波形

以 0-10ns 为例,  $fs\_to\_ds\_bus = 16'b00010000_0001_01_01$ ;  $rx\_value = 8'b00000001$ ;  $ry\_value = 8'b00000010$ ; (如下图)

则  $pc = 00010000$ ;  $op = 1000$  (独热码);  $rx = 01$ ;  $ry = 01$ ;

从而  $ds\_to\_es\_bus$  为  $1000_00000010_00000001_00010000$



CpuDecode.v 测试用例 1 的仿真验证波形

## 5、实验 2.2 (ALU 设计) 的实现及验证

(1) 自己实现的 ALU 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)。

代码如下:

### ○ alu.v

```
`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
);
wire signed [7:0] src1,src2;
assign src1 = alu_src1;
assign src2 = alu_src2;
reg [7:0] answer;
reg [8:0] tmp;
reg [3:0] shift;
always @(*) begin
    case (alu_op)
        12'h001: answer = src1 + src2;
        12'h002: answer = src1 - src2;
        12'h004: answer = alu_src1 & alu_src2;
        12'h008: answer = alu_src1 || alu_src2;
        12'h010: answer = alu_src1 << alu_src2[1:0];
        12'h020: answer = src1 >>> src2[1:0];
        12'h040: answer = {src1, src1} >> src2[1:0];
        12'h080: answer = src1 < src2;
        12'h100: answer = alu_src1 < alu_src2;
        12'h400: answer = src1 ^ src2;
        12'h200: begin
            tmp = {1'b0, alu_src1} + {1'b0, alu_src2};
            answer = tmp[8] ? tmp[8:1] : tmp[7:0];
        end
        12'h800: begin
            if(src2[4]) {tmp, shift} = {{alu_src2[3:0]}, alu_src1[1:0],
                alu_src1[7:6]}, alu_src1[5:2];
            else if(src2[5]) {tmp, shift} = {{alu_src2[3:0]}, alu_src1[5:4],
                alu_src1[3:2]}, {alu_src1[7:6], alu_src1[1:0]};
            else if(src2[6]) {tmp, shift} = {{alu_src2[3:0]}, alu_src1[7:6],
                alu_src1[3:2]}, {alu_src1[5:4], alu_src1[1:0]};
            else if(src2[7]) {tmp, shift} = {{alu_src2[3:0]}, alu_src1[5:4],
                alu_src1[1:0]}, {alu_src1[7:6], alu_src1[3:2]};
            else {tmp, shift} = {8'b0, 4'b0};
            answer = shift[0] ? ({tmp[7:0]},tmp[7:0]) >> (8 - shift[3:1])) :
                ({tmp[7:0]},tmp[7:0]) >> shift[3:1]);
        end
    endcase
```

```

end
assign alu_result = answer;
endmodule

```

## (2) 该模块的仿真验证波形及说明



alu.v 完整的仿真验证波形

以最后的两个测试用例为例:

`src1=FF,src2=01,op=200` 则转化为二进制为  $11111111+00000001=100000000$ , 按照要求应该舍弃低位得到  $10000000$ , 即 80

`src1=B6,src2=CE,op=800` 则转化为二进制按要求计算为  $10110110 \oplus 11001110 = 11010011$ , 即 D3

其他的用例经验证均符合结论。

(下列为部分仿真代码, 针对于 alu.v)

```

src1 = 8'h05; src2 = 8'h03; op = 12'h001; #10;//8'h08
src1 = 8'h05; src2 = 8'h02; op = 12'h002; #10;//8'h03
src1 = 8'h0F; src2 = 8'hF0; op = 12'h004; #10;//8'h00
src1 = 8'h0F; src2 = 8'h00; op = 12'h008; #10;//8'h01
src1 = 8'h01; src2 = 8'h02; op = 12'h010; #10;//8'h04
src1 = 8'h80; src2 = 8'h01; op = 12'h020; #10;//8'h40
src1 = 8'b10000001; src2 = 8'h01; op = 12'h040; #10;//8'b11000000
src1 = 8'h03; src2 = 8'h04; op = 12'h080; #10;//8'h01
src1 = 8'b11111111; src2 = 8'h01; op = 12'h100; #10;//8'h00
src1 = 8'h0F; src2 = 8'hF0; op = 12'h400; #10;//8'hFF
src1 = 8'hFF; src2 = 8'h01; op = 12'h200; #10;//8'b10000000
src1 = 8'b10110110; src2 = 8'b11001110; op = 12'h800; #10;// 8'b11010011

```

## (3) 板级测试验证（非强制）

由于板子上的输入端不足以支撑 8 位的计算, 故这里将相应的计算缩减到四位数上, 并删除了针对 8 位数的 itoa 运算。并用译码模块压缩输入 op 的位数, 具体的实现如下。

(下列为译码模块代码)

```

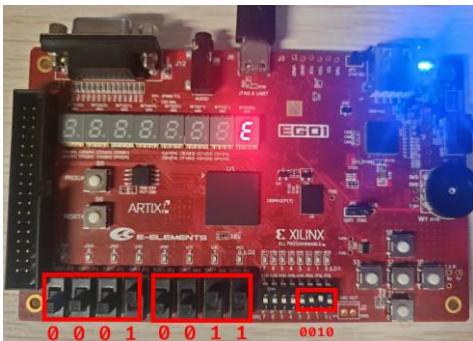
always @(*) begin
    case (op)
        4'b0001: alu_op = 12'h001; //plus
        4'b0010: alu_op = 12'h002; //minus
        4'b0011: alu_op = 12'h004; //digit AND
        4'b0100: alu_op = 12'h008; //logic OR

```

```

4'b0101: alu_op = 12'h010; //logic left shift
4'b0110: alu_op = 12'h020; //arithmetic right shift
4'b0111: alu_op = 12'h040; //loop right shift
4'b1000: alu_op = 12'h080; //less than (signed)
4'b1001: alu_op = 12'h100; //less than (unsigned)
4'b1010: alu_op = 12'h200; //special ADD
4'b1011: alu_op = 12'h400; //XOR
//4'b1100: alu_op = 12'h800; //complex op `i`
default: alu_op = 12'h000; // default case
endcase
end

```



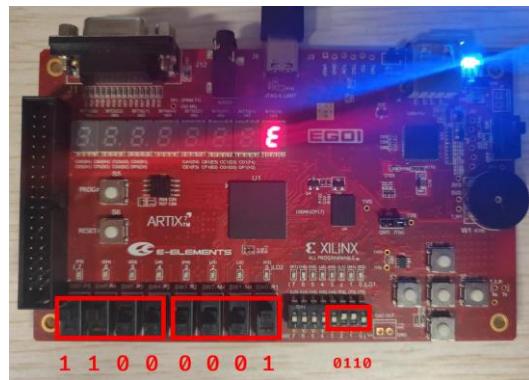
### 验证 1: 有符号减法

说明: 左侧的两个四位数代表两个输入值, 右侧的四位数代表运算。 $1 - 3 = -2$ , 补码即 1110, 故显示 E。



### 验证 2: 保首位加法

说明: 左侧的两个四位数代表两个输入值, 右侧的四位数代表运算。 $F(1111) + 1(0001) = 10000$ , 保留首位即 1000, 故显示 8。



验证 3: 算术右移

说明: 有符号右移会根据数的正负补首位, 1100 是负数故右移时补 1, 即 1110, 故显示 E。

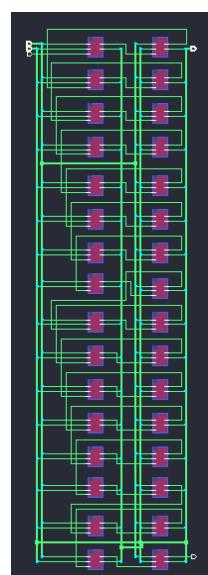


验证 4: 循环右移

说明: 循环右移, 0011 右移 1 位得到 1001, 故显示 9。

## 6、实验 2.3（加法器设计）的实现及仿真验证

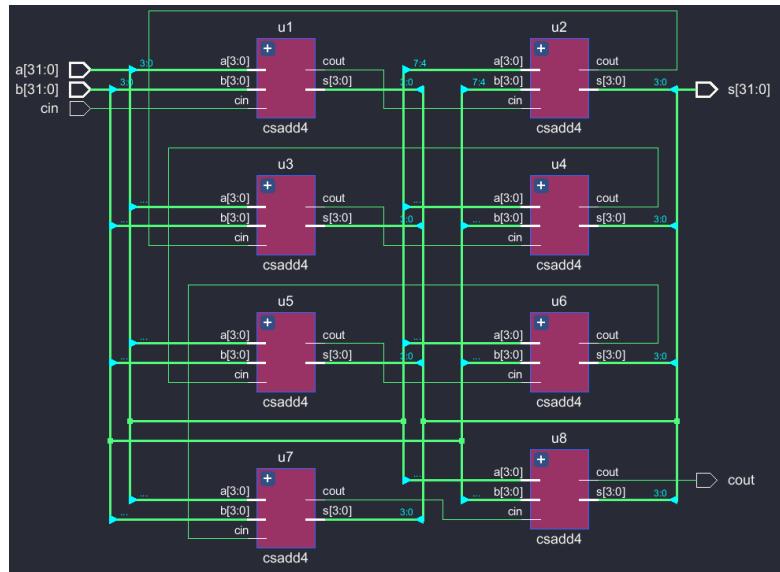
### （1）32 位逐位进位加法器模块 RTL 分析结构图及说明



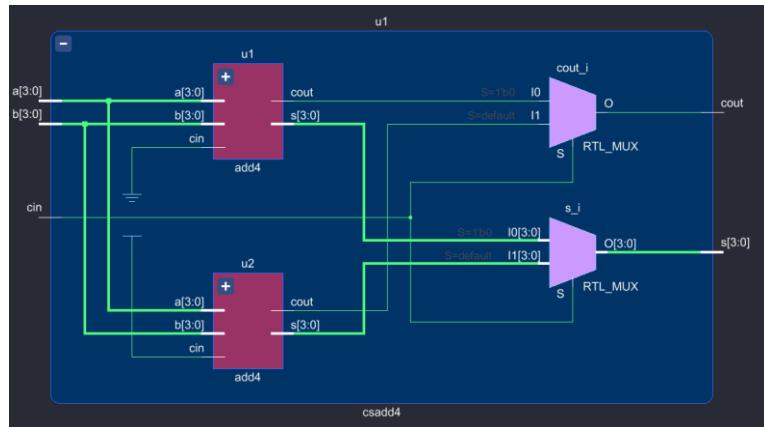
逐位进位加法器 RTL 分析图

逐位进位加法器即拼接 32 个全加器从而逐位计算结果。

## (2) 32 位选择进位加法器模块 RTL 分析结构图及说明



选择进位加法器 RTL 分析图



选择进位的具体实现

如上，由于 32 位的选择加法器设计过于复杂，故考虑拼接 4 位的选择进位加法器来实现。使用两个四位逐位进位加法器计算  $\text{cin}$  为 0 或 1 的两种情况，再根据多路选择器来具体选择。

## (3) 测试激励的设计

为了方便，手动生成测试数据，测试的情况有：

- 全 1 输入( $a = 32'hFFFFFFF; b = 32'hFFFFFFF; \text{cin} = 1'b1;$ )
- 进位链测试( $a = 32'hFFFFFFF; b = 32'h00000001; \text{cin} = 1'b0;$ )
- 随机测试( $a = 32'hA3F5C9D7; b = 32'h4B6E89A2; \text{cin} = 1'b1;$ )

利用计算出相应的计算结果，结果手动对比即可。

期待的结果分别为

- (1)FFFFFFFFFF
- (1)00000000

➤ (0)EF64537a

观察仿真波形图，计算执行加法运算所花费的时间来对比两类加法的性能差距，为此需要一定的延迟保证输出稳定下来，这个时间可以通过测量电路从未知状态恢复为稳定态的时间（即初始化所需时间）得知。

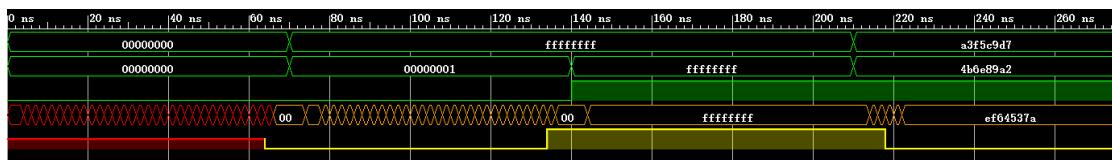
(4) 仿真波形及说明

为了实现以上测试激励，设计了如下所示的仿真代码。

```
module test_add32;
    reg [31:0] a;
    reg [31:0] b;
    reg cin;
    wire [31:0] s;
    wire cout;
    add32 uut (
        .a(a),
        .b(b),
        .cin(cin),
        .s(s),
        .cout(cout)
    );
    initial begin
        a = 32'h00000000; b = 32'h00000000; cin = 1'b0;
        //#delay;
        a = 32'hFFFFFF; b = 32'h00000001; cin = 1'b0;
        //#delay;
        a = 32'hFFFFFF; b = 32'hFFFFFF; cin = 1'b1;
        //#delay;
        a = 32'hA3F5C9D7; b = 32'h4B6E89A2; cin = 1'b1;
        //#delay;
        $finish;
    end
endmodule
```

经测试，rcadd32 的 delay 设置为 70，csadd32 的 delay 设置为 20 可以使得输出稳定。

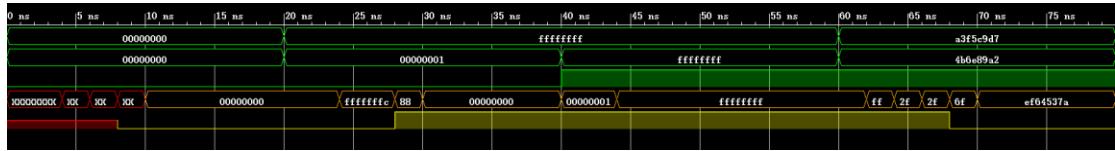
以下为仿真波形图。



逐位进位加法器的仿真波形图

不难看出输出结果是正确的，从初始化测试(0-70ns)的输出可以看到延迟为 66ns。详细分析

波形图，可以得到加法所需时间分别为 66ns(0-66ns), 66ns(70-136ns), 4ns(140-144ns), 12ns(210-222ns)



选择进位加法器的仿真波形图

不难看出输出结果是正确的，从初始化测试(0-20ns)的输出可以看到延迟为 10ns。详细分析波形图，可以得到加法所需时间分别为 10ns(0-10ns), 10ns(20-30ns), 4ns(40-44ns), 10ns(60-70ns)

如使用实验文档中的测试激励代码，则可以清晰地看到逐位进位加法器（s0, cout0）面对未知干扰后会花费更长的时间调整（红色段更长），这代表其鲁棒性更差，性能更低。



由上述结果可以看出，选择进位加法器的性能高于逐位进位加法器（尤其是有多次进位的加法）。

## 7、实验中遇到的问题、现象及解决方法

(Q1-Q4 为被同学问到的问题，Q5 是自己遇到的暂未解决的问题)

Q1：关于 CPU 译码，有没有什么方便的拼接二进制串的手段？

A：有的！使用大括号{}可以非常方便的拼接字符串，使用方法如下：

```
assign {tmp1,tmp2} = {tmpa, tmpb, tmpc, tmpd[15:8]};
```

Q2：关于 ALU 中的循环右移，有没有什么方便的办法解决？

A：有的！化环为链，把两个二进制串接在一起再右移即可。

```
12'h040: answer = {src1, src1} >> src2[1:0];
```

Q3：关于 ALU 中的有无符号比较，有没有什么方便的办法解决？

A：有的！verilog 中的数默认是无符号的，只要使用\$signed(number)将其转化为有符号数即可直接用小于号<比较。

Q4：关于 ALU 中的 itoa 运算，有没有什么方便的办法解决？

A：没有！不过熟练使用{}拼接和三目运算符?:可以省去不必要的赋值，让代码看着很简洁。另外，循环左移可以用循环右移来实现。详见上文代码

**Q5：在加法器仿真时，遇到了需要判断“信号是否稳定”的问题，有什么方法可以检测信号的稳定性？**

Try：考虑连续数次判断值是否发生变化，但无法衡量判断的标准而失败。

A: 暂时没有找到比较好的答案, 不过实验文档上的方案算是一种绕开这个问题的手段。

## 9、本次实验心得体会

进一步理解了模块化的方便之处, 了解了 verilog 的更多语法知识, 理解了选择进位加法器的性能优势。

## 10、关于本次实验课程的改进建议

1. 希望理论课或者实验文档上能有更多关于 verilog 语法知识的讲解, 据个人观察, 理论课上的内容难以提供解决问题的知识储备, 比如“没有加 `signed` 修饰, 默认就是无符号数”这一点不给说明很难意识到。

2. 实验一(CPU 译码)的实用性不大, 如果只是考察拼接字符串和译码器完全不需要这样的题目背景, 如果真的要实现 CPU 译码模块可以把这个实验放在 ALU 后面, 译码后调用 ALU 计算出结果返回。

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

逐位进位加法器 rcadd32:

```
`timescale 1ns / 1ps
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] c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13,
c14, c15, c16, c17, c18, c19, c20, c21, c22, c23, c24, c25, c26, c27,
c28, c29, c30, c31, c32;
    add1 add1_0 (a[0], b[0], cin, s[0], c1[0]);
    add1 add1_1 (a[1], b[1], c1[0], s[1], c2[1]);
    add1 add1_2 (a[2], b[2], c2[1], s[2], c3[2]);
    //后略.....
    add1 add1_31 (a[31], b[31], c31[30], s[31], c32[31]);
    assign cout = c32[31];
endmodule
module add1(
    input a,
    input b,
    input cin,
    output sum,
    output cout
);
```

```

assign #4 sum = a ^ b ^ cin;
assign #2 cout = (cin==1) | (cin==0) ? ((a & cin) | (b & cin)| (a & b)) :
1'bx;
endmodule

```

选择进位加法器 csadd32:

```

module csadd32 (
    input wire [31: 0] a,
    input wire [31: 0] b,
    input wire cin,
    output wire [31: 0] s,
    output wire cout
);
    wire c1, c2, c3, c4, c5, c6, c7, c8;
    csadd4 u1(a[3:0], b[3:0], cin, s[3:0], c1);
    csadd4 u2(a[7:4], b[7:4], c1, s[7:4], c2);
    csadd4 u3(a[11:8], b[11:8], c2, s[11:8], c3);
    csadd4 u4(a[15:12], b[15:12], c3, s[15:12], c4);
    csadd4 u5(a[19:16], b[19:16], c4, s[19:16], c5);
    csadd4 u6(a[23:20], b[23:20], c5, s[23:20], c6);
    csadd4 u7(a[27:24], b[27:24], c6, s[27:24], c7);
    csadd4 u8(a[31:28], b[31:28], c7, s[31:28], c8);
    assign cout = c8;
endmodule
module csadd4(
    input wire [3: 0] a,
    input wire [3: 0] b,
    input wire cin,
    output wire [3: 0] s,
    output wire cout
);
    wire [3: 0] s1, s2;
    wire tmp1,tmp2;
    add4 u1(a, b, 1'b0, s1, tmp1);
    add4 u2(a, b, 1'b1, s2, tmp2);
    assign s = (cin == 1'b0) ? s1 : s2;
    assign cout = (cin == 1'b0) ? tmp1 : tmp2;
endmodule
module add4 (
    input wire [3: 0] a,
    input wire [3: 0] b,
    input wire cin,
    output wire [3: 0] s,
    output wire cout
);

```

```

wire c1, c2, c3, c4;
add1 u1(a[0], b[0], cin, s[0], c1);
add1 u2(a[1], b[1], c1, s[1], c2);
add1 u3(a[2], b[2], c2, s[2], c3);
add1 u4(a[3], b[3], c3, s[3], c4);
assign cout = c4;
endmodule

module add1(
    input a,
    input b,
    input cin,
    output sum,
    output cout
);
assign #4 sum = a ^ b ^ cin;
assign #2 cout = (cin==1) | (cin==0) ? ((a & cin) | (b & cin)| (a & b)) :
1'b0;
endmodule

```

### 超前进位选择加法器 csa32:

```

module carry_select_adder32(
    input [31:0] a,
    input [31:0] b,
    input cin,
    output [31:0] sum,
    output cout
);
    wire [15:0] sum0, sum1;
    wire cout0, cout1;
    carry_select_adder16 U1 (
        .a(a[15:0]),
        .b(b[15:0]),
        .cin(cin),
        .sum(sum[15:0]),
        .cout(cout0)
    );
    carry_select_adder16 U2 (
        .a(a[31:16]),
        .b(b[31:16]),
        .cin(1'b0),
        .sum(sum0),
        .cout(cout1)
    );

```

```

carry_select_adder16 U3 (
    .a(a[31:16]),
    .b(b[31:16]),
    .cin(1'b1),
    .sum(sum1),
    .cout(cout)
);
assign sum[31:16] = (cout0 == 1'b0) ? sum0 : sum1;

endmodule

module carry_select_adder16(
    input [15:0] a,
    input [15:0] b,
    input cin,
    output [15:0] sum,
    output cout
);
    wire [7:0] sum0, sum1;
    wire cout0, cout1;

    ripple_carry_adder8 U1 (
        .a(a[7:0]),
        .b(b[7:0]),
        .cin(cin),
        .sum(sum[7:0]),
        .cout(cout0)
    );
    ripple_carry_adder8 U2 (
        .a(a[15:8]),
        .b(b[15:8]),
        .cin(1'b0),
        .sum(sum0),
        .cout(cout1)
    );
    ripple_carry_adder8 U3 (
        .a(a[15:8]),
        .b(b[15:8]),
        .cin(1'b1),
        .sum(sum1),
        .cout(cout)
    );
    assign sum[15:8] = (cout0 == 1'b0) ? sum0 : sum1;

```

```
endmodule
module ripple_carry_adder8 (
    input [7:0] a,
    input [7:0] b,
    input cin,
    output [7:0] sum,
    output cout
);
    wire [7:0] p, g, c;

    assign p = a ^ b;
    assign g = a & b;

    assign c[0] = cin;
    assign c[1] = g[0] | (p[0] & c[0]);
    assign c[2] = g[1] | (p[1] & c[1]);
    assign c[3] = g[2] | (p[2] & c[2]);
    assign c[4] = g[3] | (p[3] & c[3]);
    assign c[5] = g[4] | (p[4] & c[4]);
    assign c[6] = g[5] | (p[5] & c[5]);
    assign c[7] = g[6] | (p[6] & c[6]);

    assign sum = p ^ c;
    assign cout = g[7] | (p[7] & c[7]);
endmodule
```