实验三 算术逻辑单元设计

一、实验目的

1 掌握 Verilog 语言和 Vivado、Logisim 开发平台的使用；

2 掌握算术逻辑单元的设计和测试方法。

二、实验内容

1 运算模块的设计与测试

2 算术逻辑单元设计与测试

三、实验要求

1 掌握 Vivado 或 Logisim 开发工具的使用，掌握以上电路的设计和测试方法；

2 记录设计和调试过程（Verilog 代码/电路图/表达式/真值表，Vivado 仿真结果，Logisim 验证结果等）；

3 分析 Vivado 仿真波形/Logism 验证结果，注重输入输出之间的对应关系。

四、实验过程及分析

1.一位全加器

设计代码

```
module fulladder_1(
    input A,B,Ci,
    output S,Co
    );
    assign Co=(A&B)|(A&Ci)|(B&Ci);
    assign S=(~A&~Ci&B)|(A&~Ci&~B)|(~A&Ci&~B)|(A&Ci&B);
endmodule
```

分析：输入端口包括 A、B 和 Ci，它们分别是 1 位输入。

输出端口包括 S（1 位的和）和 Co（1 位的进位输出）。

使用 assign 语句计算 Co 和 S 的值。Co 的值等于 A 和 B 的与操作结果、A 和 Ci 的与操作结果以及 B 和 Ci 的与操作结果的逻辑或。S 的值由四个条件的逻辑或操作计算，分别对应全加器的四个输入组合。

仿真代码

```
module sim_fulladder_1;
    reg A,B,ci;
    wire s,co;
    initial begin
        A= 1'b0;
        B= 1'b0;
        ci = 1'b0;
    end
    always begin
        #50 B <= 1'b1;
```

```
        #50 A <= 1'b1;
        #50 B <= 1'b0;
        #50
        ci <= 1'b1;
        A  <= 1'b0;
        B  <= 1'b0;
        #50 B <= 1'b1;
        #50 A <= 1'b1;
        #50 B <= 1'b0;
        #10
        $finish;
    end
    fulladder_1 FA1(
        .A      (A),
        .B      (B),
        .Ci     (ci),
        .S      (s),
        .Co     (co)
        );
Endmodule
```

分析：输入端口包括 A、B 和 Ci，它们分别是 1 位的寄存器。

输出端口包括 s 和 co，它们分别是 1 位的线（wire）。

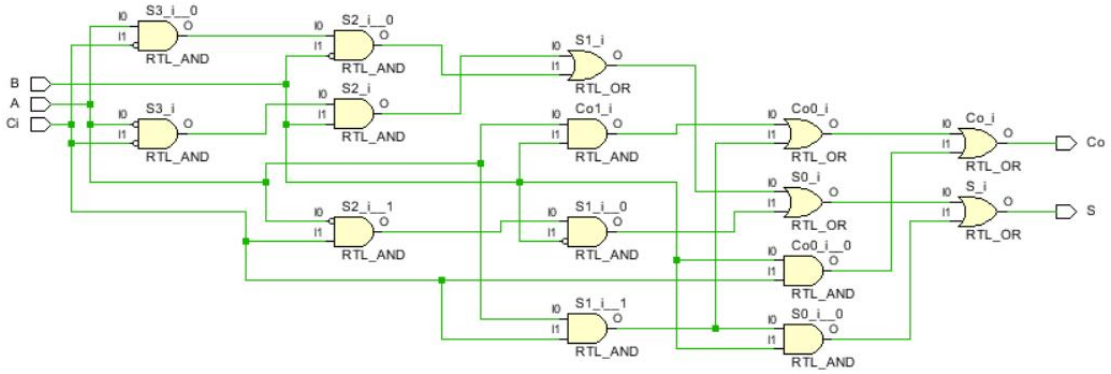
在 initial 语句块中，将 A、B 和 Ci 初始化为特定的值。

在 always 块中，定义了一系列操作，每个操作之间都有一定的延迟 #，以模拟时钟周期。在每个时钟周期内，A 和 B 的值被更改，模拟输入信号的变化。

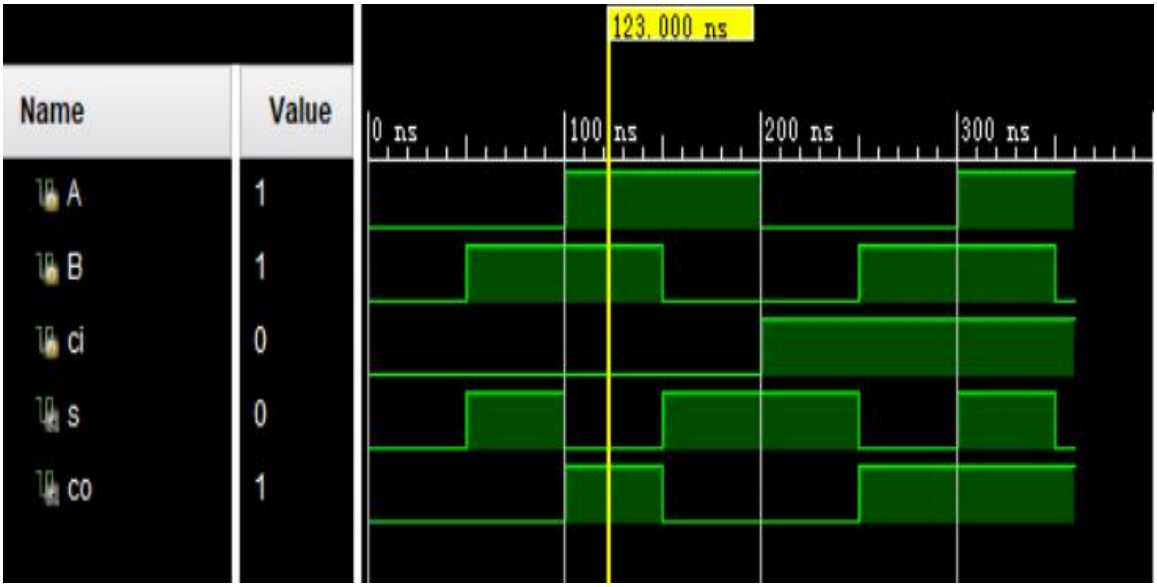
最后，在 sim_fulladder_1 模块中实例化了 fulladder_1 模块，并将输入和输出信号连接到它。

最后，在仿真的最后，使用 $finish 终止仿真，表示仿真已完成。

电路图



波形图



波形图中正确显示了 1 位全加器的运算结果，包括有无输入进位、输出进位。当 A=1，B=1，Ci=0 时，输出为 0，进位为 1，符合要求。当 A=0，B=1，Ci=1 时，输出为 0，进位为 1，符合要求。

2. 四位全加器

设计代码

```
module fulladder_4(
    input[3:0] A,B,
    input   Ci,
    output Co,
    output[3:0] S
    );
    wire C0,C1,C2;
    fulladder_1 FA0(
        .A  (A[0]),
```

```
        .B  (B[0]),
        .Ci (Ci),
        .S (S[0]),
        .Co (C0)
    );
    fulladder_1 FA1(
            .A  (A[1]),
            .B  (B[1]),
            .Ci (C0),
            .S (S[1]),
            .Co (C1)
        );
    fulladder_1 FA2(
                .A  (A[2]),
                .B  (B[2]),
                .Ci (C1),
                .S (S[2]),
                .Co (C2)
            );
    fulladder_1 FA3(
                    .A  (A[3]),
                    .B  (B[3]),
                    .Ci (C2),
                    .S (S[3]),
                    .Co (Co)
                );

endmodule
```

分析：输入端口包括 A 和 B，每个都是 4 位输入，以及 Ci（进位输入）。

输出端口包括 Co（输出进位）和 S（4 位的输出）。

在 fulladder_4 模块内部，有 4 个 1 位全加器 fulladder_1（FA0 到 FA3）。每个 1 位全加器都有类似的接口，将 A 和 B 的相应 1 位切片与进位输入相加，生成 1 位输出和一个输出进位。

C0 到 C2 是用来传递进位的中间信号，分别连接每个 1 位全加器的输出进位。

最后一个 1 位全加器的输出进位 C3 连接到 Co，而每个 1 位全加器的 1 位输出通过连接到 S 的相应切片中，形成 4 位输出。

仿真代码

```
module sim_fulladder_4;
```

```
    reg[3:0] A,B;
    reg Ci; //最低位进位
    wire Co;
    wire[3:0] S;
    integer i,j;
    initial begin
        A = 4'b0000;
        B = 4'b0000;
        Ci = 1;
    end
    always begin

        for(i = 0; i <15;i=i+1) begin
            #10 ;
            B <= 4'b0000;
            A <= A + 4'b0001;
            for(j=0;j<15;j=j+1) begin
                #10 ;
                B <= B + 4'b0001;
            end
        end
    end
     fulladder_4 ADD4(
        .A (A),
        .B (B),
        .Ci (Ci),
        .Co (Co),
        .S (S)
     );
Endmodule
```

分析：输入端口包括 A 和 B，每个都是 4 位的寄存器，并且有一个 Ci 寄存器作为进位输入。

输出端口包括 Co 和 S，分别是 1 位和 4 位的线（wire）。

使用整数 i 和 j 来控制仿真中的循环。

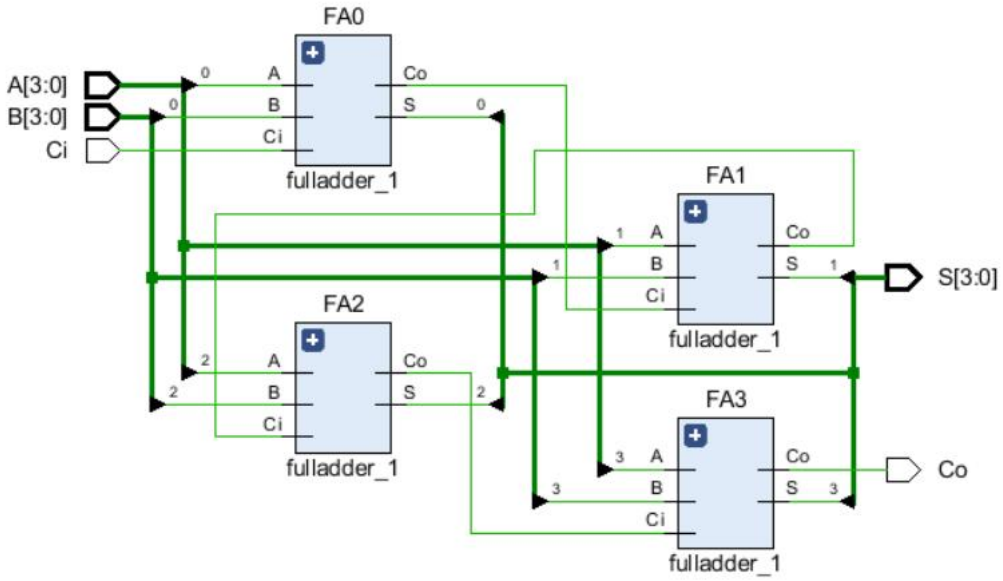
在 initial 语句块中，将 A、B 和 Ci 初始化为特定的值。A 和 B 初始化为全零，而 Ci 初始化为 1，表示最低位进位。

在 always 块中，使用嵌套的 for 循环来生成一系列输入，以模拟输入信号的变化。在每个时钟周期内，A 和 B 的值被更改，以模拟输入信号的变化。

最后，在 sim_fulladder_4 模块中实例化了 fulladder_4 模块，并将输入和输出信号连接到它。

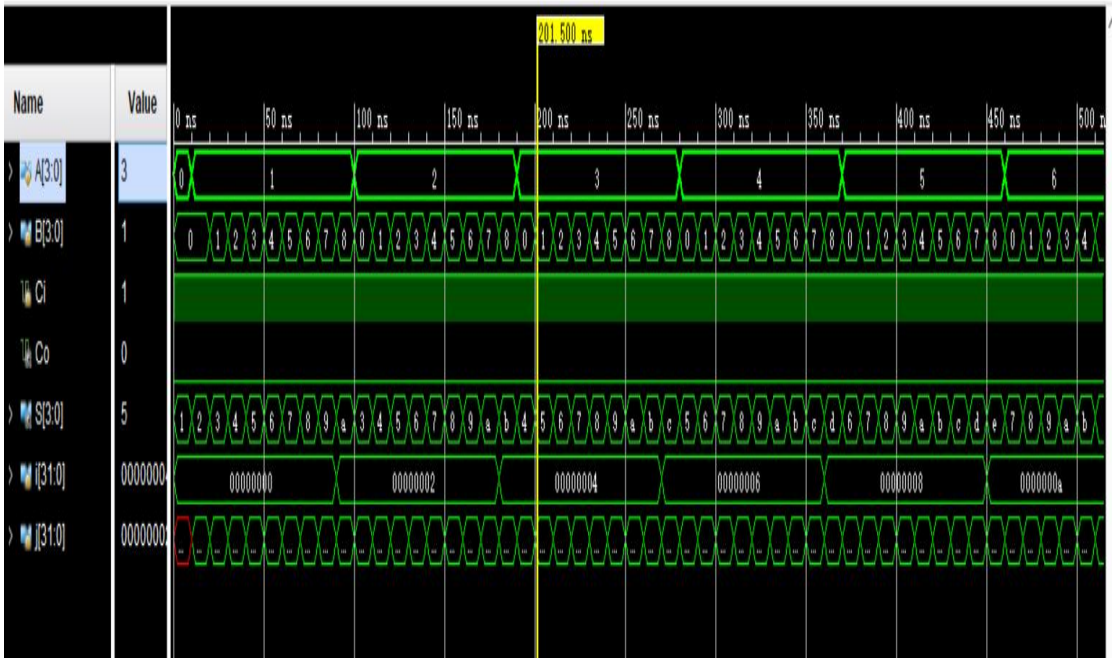
在仿真的最后，仿真会使用 $finish 终止，表示仿真已完成

电路图



波形图

波形图中正确显示了 4 位全加器的运算结果，包括有无输入进位、输出进位。当输入 A=3，B=1，Ci=1，输出为 5，没有进位。符号要求。



3. 16 位全加器

设计代码

```
module fulladder_16(
    input [15:0] A,B,
    input Ci,
    output Co,
    output [15:0] S
    );
    wire C0,C1,C2;
    fulladder_4 A0(
    .A (A[3:0]),
    .B (B[3:0]),
    .S (S[3:0]),
    .Ci (Ci),
    .Co (C0)
    );
    fulladder_4 A1(
    .A (A[7:4]),
    .B (B[7:4]),
    .S (S[7:4]),
    .Ci (C0),
    .Co (C1)
    );
    fulladder_4 A2(
    .A (A[11:8]),
    .B (B[11:8]),
    .S (S[11:8]),
    .Ci (C1),
    .Co (C2)
    );
    fulladder_4 A3(
    .A (A[15:12]),
    .B (B[15:12]),
    .S (S[15:12]),
    .Ci (C2),
    .Co (Co)
    );
Endmodule
```

分析：输入端口包括 A 和 B，每个都是 16 位的输入，以及 Ci（进位输入）。

输出端口包括 Co（输出进位）和 S（16 位的输出）。

在 fulladder_16 模块内部，有 4 个 4 位全加器 fulladder_4（A0 到 A3）。每个 4 位全加器都有类似的接口，将 A 和 B 的相应 4 位切片与进位输入相加，生成 4 位输出和一个输出进位。

C0 到 C2 是用来传递进位的中间信号，分别连接每个 4 位全加器的输出进位。

最后一个 4 位全加器的输出进位 C3 连接到 Co，而每个 4 位全加器的 4 位输出通过连接到 S 的相应切片中，形成 16 位输出。

仿真代码

```
module sim_fulladder_16;
    reg[15:0] A,B;
    reg Ci;
    wire Co;
    wire[15:0] S;
    initial begin
        A = 16'd0;
        B = 16'd0;
        Ci = 0;
    end
    always begin

        #10
        A <= {$random};
        B <= {$random};
        #50
        A <= {$random};
        B <= {$random};

        #50
        Ci<= 1;
        A <= {$random};
        B <= {$random};

        #50
        A <= {$random};
        B <= {$random};
        end

        fulladder_16 ADD16(
```

```
.A (A),
.B (B),
.Ci (Ci),
.Co (Co),
.S (S)
);
Endmodule
```

分析：输入端口包括 A 和 B，每个都是 16 位的寄存器，并且有一个 Ci 寄存器作为进位输入。

输出端口包括 Co 和 S，分别是 1 位和 16 位的线（wire）。

在 initial 语句块中，将 A、B 和 Ci 初始化为特定的值。A 和 B 初始化为全零，而 Ci 初始化为 0。

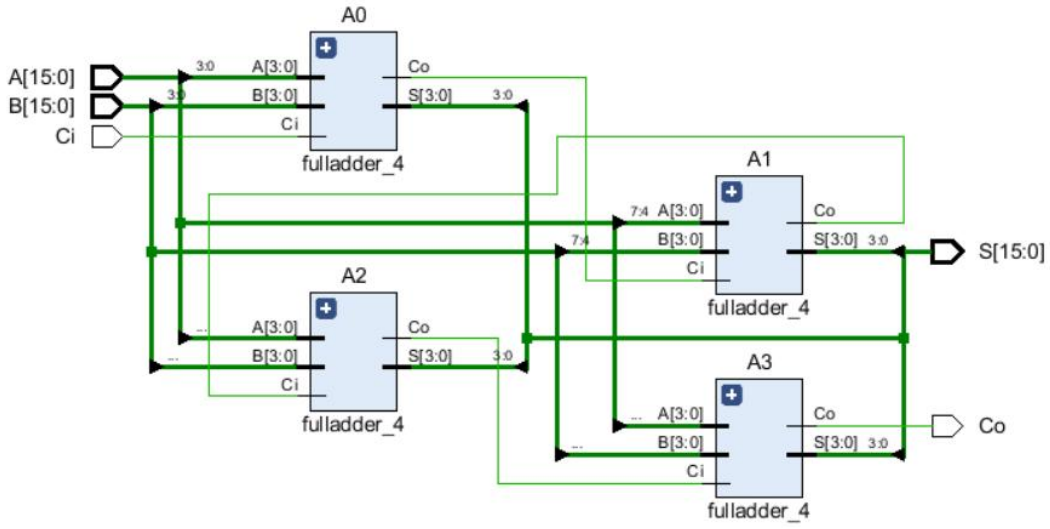
在 always 块中，使用 $random 系统函数生成随机的 16 位值，并在不同的时钟周期内更新 A 和 B 的值，模拟输入信号的变化。

在每个时钟周期内， A 和 B 的值会发生变化，模拟输入信号的变化。

最后，在 sim_fulladder_16 模块中实例化了 fulladder_16 模块，并将输入和输出信号连接到它。

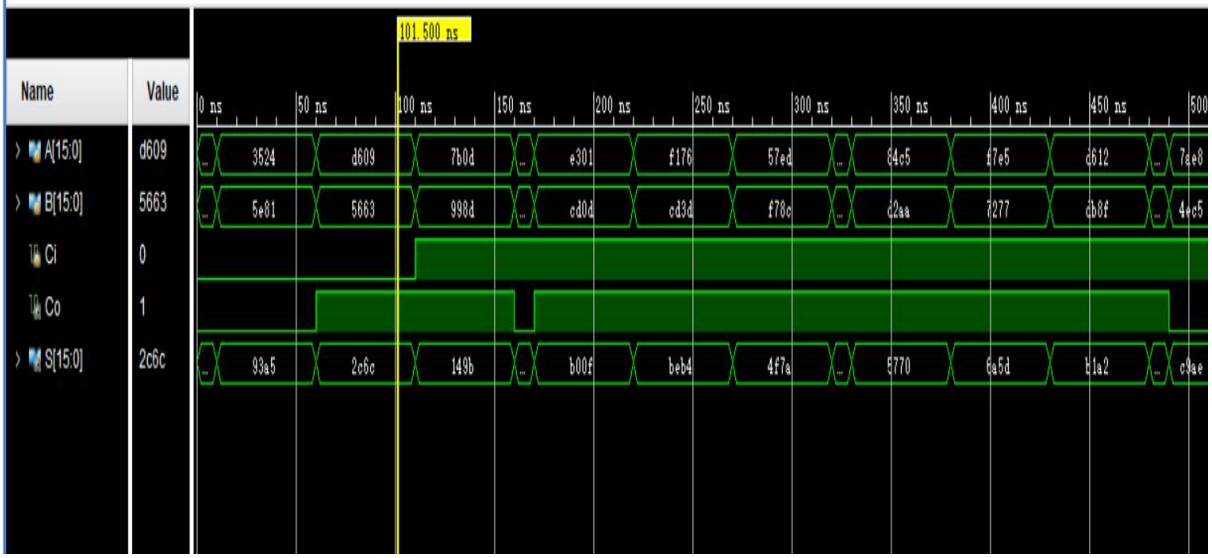
在仿真的最后，使用 $finish 终止仿真，表示仿真已完成。

电路图



波形图

波形图中正确显示了 16 位全加器的运算结果，包括有无输入进位、输出进位。当输入为 A=0xd509,B=0x5663，Ci=0，输出进位为 1，S=0x2b6c，符合要求。当输入为 A=e301，B=cd3d,Ci=1 时，输出为 0xb03f,进位为 1，符合要求。



e

4.32 位全加器

设计代码

```
module fulladder_32(
    input [31:0] A,B,
    input Ci,
    output Co,
    output [31:0] S
    );
    wire C0,C1,C2,C3,C4,C5,C6;
    fulladder_4 A0(
    .A (A[3:0]),
    .B (B[3:0]),
    .S (S[3:0]),
    .Ci (Ci),
    .Co (C0)
    );
    fulladder_4 A1(
    .A (A[7:4]),
    .B (B[7:4]),
    .S (S[7:4]),
    .Ci (C0),
    .Co (C1)
    );
    fulladder_4 A2(
    .A (A[11:8]),
    .B (B[11:8]),
    .S (S[11:8]),
    .Ci (C1),
    .Co (C2)
```

```
    );
    fulladder_4 A3(
    .A (A[15:12]),
    .B (B[15:12]),
    .S (S[15:12]),
    .Ci (C2),
    .Co (C3)
    );
    fulladder_4 A4(
    .A (A[19:16]),
    .B (B[19:16]),
    .S (S[19:16]),
    .Ci (C3),
    .Co (C4)
    );
    fulladder_4 A5(
    .A (A[23:20]),
    .B (B[23:20]),
    .S (S[23:20]),
    .Ci (C4),
    .Co (C5)
    );
    fulladder_4 A6(
    .A (A[27:24]),
    .B (B[27:24]),
    .S (S[27:24]),
    .Ci (C5),
    .Co (C6)
    );
    fulladder_4 A7(
    .A (A[31:28]),
    .B (B[31:28]),
    .S (S[31:28]),
    .Ci (C6),
    .Co (Co)
    );
Endmodule
```

分析：输入端口包括 A 和 B，每个都是 32 位的输入，以及 Ci（进位输入）。

输出端口包括 Co（输出进位）和 S（32 位的输出）。

在 fulladder_32 模块内部，有 8 个 4 位全加器 fulladder_4（A0 到 A7）。每个 4 位全加器都有类似的接口，将 A 和 B 的相应 4 位切片与进位输入相加，生成 4 位输出和一个输出进位。

C0 到 C6 是用来传递进位的中间信号，分别连接每个 4 位全加器的输出进位。

最后一个 4 位全加器的输出进位 C7 连接到 Co，而每个 4 位全加器的 4 位输出通过连接到 S 的相应切片中，形成 32 位输出。

仿真代码

```
module sim_fulladder_32;
    reg[31:0] A,B;
    reg Ci;
    wire Co;
    wire[31:0] S;
    initial begin
        A = 32'd0;
        B = 32'd0;
        Ci = 0;
    end
    always begin

        #10
        A <= {$random};
        B <= {$random};
        #50
        A <= {$random};
        B <= {$random};

        #50
        Ci<= 1;
        A <= {$random};
        B <= {$random};

        #50
        A <= {$random};
        B <= {$random};
        end

        fulladder_16 ADD16(
```

```
.A (A),
.B (B),
.Ci (Ci),
.Co (Co),
.S (S)
);
Endmodule
```

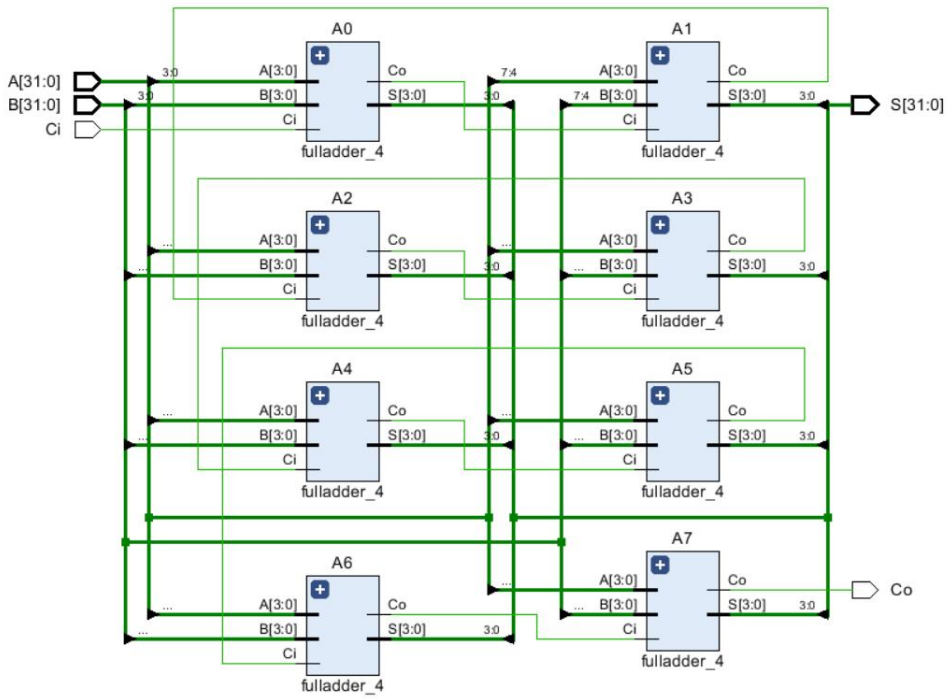
分析：输入端口包括 A 和 B，每个都是 32 位的寄存器，并且有一个 Ci 寄存器作为进位输入。

输出端口包括 Co 和 S，分别为输出进位和输出信号。

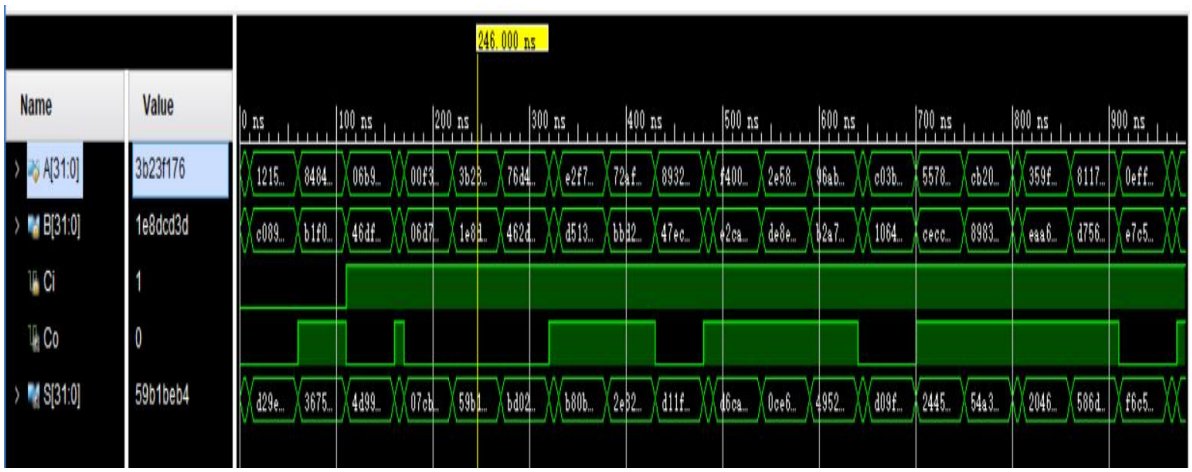
在 initial 语句块中，将 A、B 和 Ci 初始化为零。

always 块用于生成输入，并且包含了一系列的延迟 # 操作，以便在仿真中模拟时钟周期。在不同的时钟周期内，生成了不同的随机 A 和 B 输入，以及对 Ci 的设置。

电路图



波形图



波形图中正确显示了 32 位全加器的运算结果，包括有无输入进位、输出进位。当输入为 A=0x3b23f176,B=0x1e8dcd3d,Ci=1 时，输出为 0x59b1beb4,进位为 0，符合要求。

5.8 功能 32 位 ALU

设计代码

1ALU8

```
module ALU8(F, CF, A,B, OP);
    parameter SIZE = 32;
    output reg [SIZE-1:0] F;
    output CF;
    input[SIZE-1: 0] A, B;
    input[3:0] OP;
    //功能定义
    parameter ALU_AND= 4'b0000;//按位与 0
    parameter ALU_OR=4'b0001;//按位或 1
    parameter ALU_XOR = 4'b0010;//按位异或 2
    parameter ALU_NOR = 4'b0011;//按位或非 3
    parameter ALU_ADD= 4'b0100;//算数加法 4
    parameter ALU_SUB=4'b0101;//算数减法 5
    parameter ALU_SLT = 4'b0110;//比较，若 A<B，则输出 1;否则输出 06
    parameter ALU_SLL = 4'b0111;//逻辑左移，E 逻辑左移 A 所指定的位数 7
    //中间变器
    wire [7:0] EN;  //运算使能信号
    wire [SIZE-1:0] Fw, Fa;//线网输出 F
    //数据流描述
    assign Fa =A & B;//按位与
    //行为描述
    always@(*) begin
```

```
        case(OP)
            ALU_AND: begin F<=Fa; end//按位与
            ALU_OR: begin F <=A|B; end//按位或
            ALU_XOR: begin F <= A^B; end//按位异或
            ALU_NOR: begin F <=~(A|B); end//按位或非
            default: F = Fw;
        endcase
    end
    //结构化描述
    //3-8 编码,生成片选 EI 信号
    Decoder38 decoder38_1(OP[2:0],EN);
    //算术加法
    ADD add_1(Fw,CF,A,B, EN[4]);
    //算术减法
    SUB sub_1(Fw, CF, A,B,EN[5]);
    //比较,若 A<B，则输出 1:否则输出 0
    SLT slt_1(Fw,A,B, EN[6]);
    //逻辑左移,B 逻辑左移 A 所指定的位数
    SLL sll_1(Fw,A,B, EN[7]);
Endmodule
```

分析：ALU8 模块是主要的 ALU 模块，它接收两个 32 位输入 A 和 B，一个 4 位操作码 OP，以及产生一个 32 位输出 F 和一个进位输出 CF。

ALU8 模块包括一个 Decoder38 模块，用于将 4 位的操作码 OP 转换为 8 位的使能信号 EN，然后根据不同的操作码执行相应的操作。

操作码 OP 包括按位与、按位或、按位异或、按位或非、算数加法、算数减法、比较和逻辑左移。

操作码 OP 与使能信号 EN 之间的关系由 Decoder38 模块实现。

2Decoder38

```
module Decoder38 (
    input [2:0] A,  // 3 位输入信号
    output wire [7:0] Y  // 8 位输出信号
);

assign Y = (A == 3'b000) ? 8'b00000001 :
           (A == 3'b001) ? 8'b00000010 :
           (A == 3'b010) ? 8'b00000100 :
           (A == 3'b011) ? 8'b00001000 :
           (A == 3'b100) ? 8'b00010000 :
```

```
        (A == 3'b101) ? 8'b00100000 :
        (A == 3'b110) ? 8'b01000000 :
                        8'b10000000;
Endmodule
```

分析：将 3 位的操作码 OP 转化为 8 位的使能信号 EN，根据操作码的不同，EN 的不同位会被设置为 1。

3ADD

```
module ADD (
  output [31:0] Fw, // 输出
  output CF,        // 进位
  input [31:0] A,   // 加数 A
  input [31:0] B,   // 加数 B
  input EN          // 使能信号
);

  wire [31:0] sum; // 中间结果的和
  wire carry_out;  // 中间结果的进位

  // 调用 32 位全加器模块 fulladder_32
  fulladder_32 fulladder (
     .A(A),           // 加数 A
    .B(B),           // 加数 B
    .Ci(0),          // 使能信号作为进位输入
    .Co(carry_out),  // 中间结果的进位
    .S(sum)          // 中间结果的和
  );

  // 根据使能信号 EN 决定输出的值
  assign Fw = (EN == 1'b1) ? sum : 32'bz;
  assign CF = (EN == 1'b1) ? carry_out : 1'bz;

Endmodule
```

分析：计算两个输入 A 和 B 的和，产生一个进位 CF 和一个 32 位的输出 F。

4SUB

```
module SUB (
  output [31:0] Fw,   // 输出
  output CF,          // 借位输出
  input [31:0] A,     // 操作数 A
```

```
  input [31:0] B,      // 操作数 B
  input EN             // 使能信号
);

  wire [31:0] NotB;   // B 的按位取反
  wire [31:0] AddB;   // B 的二进制补码
  wire [31:0] TempSum;// 用于存储加法结果的临时变量

  // 计算 B 的按位取反
  assign NotB = ~B;

  // 计算 B 的二进制补码
  assign AddB = NotB + 1'b1;

  // 如果 EN[5] 为 1，执行正常运算
  // 否则，将输出置为高阻态
  // 当 EN[5] 为 0 时，Fw 和 CF 都被设置为高阻态
  assign Fw = (EN == 1'b1) ? (A + AddB) : 32'bzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz;
  assign CF = (EN == 1'b1) ? (A < B) : 1'bz;

Endmodule
```

分析：计算两个输入 A 和 B 的差，产生一个借位 CF 和一个 32 位的输出 F。

5SLT

```
module SLT (
  output reg [31:0] Fw,      // 输出
input [31:0] A, // 操作数 A
input [31:0] B, // 操作数 B
input EN        // 使能信号
);


  always @(A,B,EN)begin
        if(EN==1)Fw<= (A<B)?32'b1:32'b0;
        else Fw<=32'bz;
 end

Endmodule
```

分析：执行比较操作，如果 A < B，则输出 1，否则输出 0。

6SLL

```
module SLL (F,A,B,EN);
  output reg [31:0] F; // 输出
  input [31:0] A;   // 操作数 A
  input [31:0] B;   // 操作数 B
  input EN;          // 使能信号


  // 逻辑左移操作，将操作数 B 逻辑左移 A 所指定的位数
  always @(A,B,EN)begin
        if(EN==1)F<=B<<A;
        else F<=32'bz;
 end

Endmodule
```

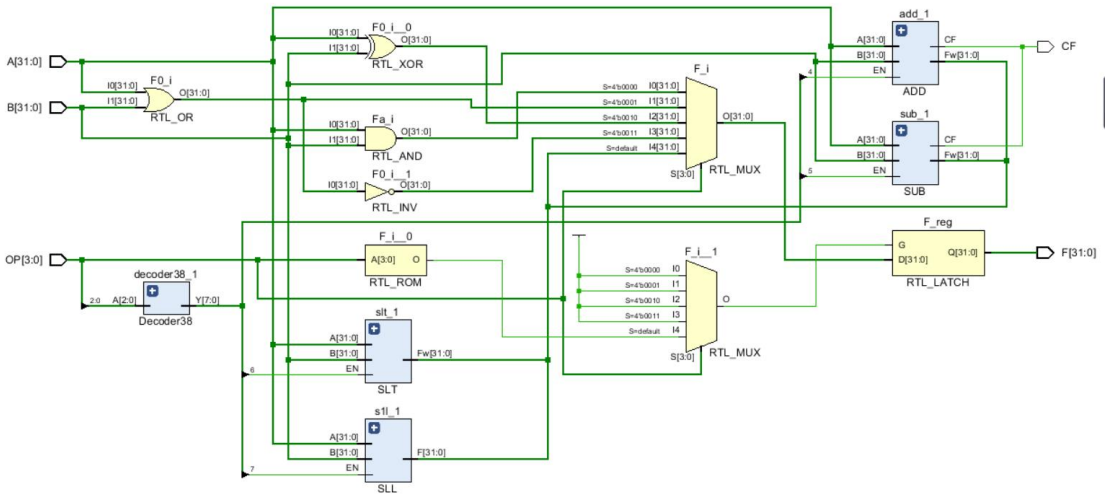
分析：执行逻辑左移操作，将输入 B 按照 A 指定的位数向左移动。

仿真代码

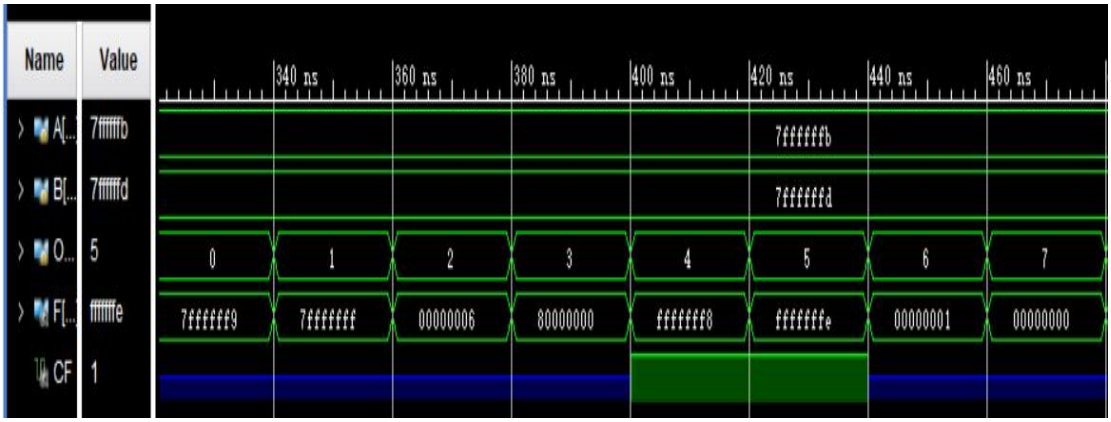
```
module sim_ALU;
    reg [31:0] A, B;
    reg [3:0] OP;
    wire [31:0] F;
    wire CF;
    initial begin
        A = 2147483643 ;
        B = 2147483645;
        OP = 0;
    end
    always begin
        repeat(8) #20 OP = OP + 1;
        OP=4'b0000;
    end
    ALU8 ALU(F, CF, A, B, OP);
Endmodule
```

分析：模拟了 ALU 执行不同操作的功能。通过初始化 A、B 和 OP，以及定期更改 OP 的值，可以观察 ALU 的不同操作，如按位与、按位或、按位异或等。在每个操作之后，ALU 的输出结果将存储在 F 中，进位输出存储在 CF 中。验证 ALU 模块的正确性和功能。

电路图



波形图



A 的值为 7ffffffb0x,B 的值为 7ffffffd0x,当 OP=4 时，为加，A、B 相加会出现溢出，此时 CF 输出高电平，满足要求；当 OP=5 时，为减，A、B 相减会出现借位，此时 CF 输出高电平，满足要求。

6. 先行进位加法器

设计代码

1.一位全加器

```
module adder(X,Y,Ci,F,Co);
  input X,Y,Ci;
  output F,Co;
  assign F = X ^ Y ^ Ci;
  assign Co = (X ^ Y) & Ci | X & Y;
endmodule
```

分析： 1 位全加器，接受两个输入 X 和 Y，以及进位输入 Ci，并产生和输出 F 和进位输出 Co。

2.4 位 CLA 部件

```
module CLA(c0,c1,c2,c3,c4,a1,a2,a3,a4,b1,b2,b3,b4);
    input c0,b1,b2,b3,b4,a1,a2,a3,a4;
    output c1,c2,c3,c4;
    assign c1 = b1 ^ (a1 & c0),
           c2 = b2 ^ (a2 & b1) ^ (a2 & a1 & c0),
           c3 = b3 ^ (a3 & b2) ^ (a3 & a2 & b1) ^ (a3 & a2 & a1 & c0),
           c4 = b4^(a4&b3)^(a4&a3&b2)^(a4&a3&a2&b1)^(a4&a3&a2&a1&c0);
endmodule
```

分析：4 位 CLA 部件，接受 4 位输入 a1 到 a4 和 b1 到 b4，以及进位输入 c0，并产生 4 位进位输出 c1 到 c4

3.四位并行进位加法器

```
module adder_4(x,y,c0,c4,F,Bm,Am);
     input [4:1] x;
     input [4:1] y;
     input c0;
     output c4,Bm,Am;
     output [4:1] F;
     wire a1,a2,a3,a4,b1,b2,b3,b4;
     wire c1,c2,c3;
     adder adder1(
               .X(x[1]),
                   .Y(y[1]),
                   .Ci(c0),
                   .F(F[1]),
                   .Co()
              );
     adder adder2(
               .X(x[2]),
                   .Y(y[2]),
                   .Ci(c1),
                   .F(F[2]),
                   .Co()
              );
     adder adder3(
               .X(x[3]),
                   .Y(y[3]),
                   .Ci(c2),
                   .F(F[3]),
```

```
                .Co()
            );
    adder adder4(
             .X(x[4]),
                .Y(y[4]),
                .Ci(c3),
                .F(F[4]),
                .Co()
            );
      CLA CLA(
          .c0(c0),
          .c1(c1),
          .c2(c2),
          .c3(c3),
          .c4(c4),
          .a1(a1),
          .a2(a2),
          .a3(a3),
          .a4(a4),
          .b1(b1),
          .b2(b2),
          .b3(b3),
          .b4(b4)
      );
  assign   a1 = x[1] ^ y[1],
           a2 = x[2] ^ y[2],
           a3 = x[3] ^ y[3],
           a4 = x[4] ^ y[4];
  assign   b1 = x[1] & y[1],
           b2 = x[2] & y[2],
           b3 = x[3] & y[3],
           b4 = x[4] & y[4];
  assign Am = a1 & a2 & a3 & a4,
         Bm = b4 ^ (a4 & b3) ^ (a4 & a3 & b2) ^ (a4 & a3 & a2 & b1);
endmodule
```

分析：四位并行进位加法器模块，接受两个 4 位输入 x 和 y，进位输入 c0，并产生 4 位和输出 F、4 位进位输出 Bm 和 1 位进位输出 Am。

4.16 位 CLA 部件

```
module CLA_16(A,B,c0,S,ax,bx);
    input [16:1] A;
    input [16:1] B;
    input c0;
    output bx,ax;
    output [16:1] S;
    wire c4,c8,c12;
    wire Am1,Bm1,Am2,Bm2,Am3,Bm3,Am4,Bm4;
    adder_4 adder1(
         .x(A[4:1]),
          .y(B[4:1]),
          .c0(c0),
          .c4(),
          .F(S[4:1]),
          .Bm(Bm1),
          .Am(Am1)
    );
    adder_4 adder2(
         .x(A[8:5]),
          .y(B[8:5]),
          .c0(c4),
          .c4(),
          .F(S[8:5]),
          .Bm(Bm2),
          .Am(Am2)
    );
    adder_4 adder3(
         .x(A[12:9]),
          .y(B[12:9]),
          .c0(c8),
          .c4(),
          .F(S[12:9]),
          .Bm(Bm3),
          .Am(Am3)
    );
    adder_4 adder4(
         .x(A[16:13]),
          .y(B[16:13]),
```

```
            .c0(c12),
            .c4(),
            .F(S[16:13]),
            .Bm(Bm4),
            .Am(Am4)
        );
        assign   c4 = Bm1 ^ (Am1 & c0),
                 c8 = Bm2 ^ (Am2 & Bm1) ^ (Am2 & Am1 & c0),
                 c12 = Bm3 ^ (Am3 & Bm2) ^ (Am3 & Am2 & Bm1) ^ (Am3 & Am2 &
Am1 & c0);
        assign  ax = Am1 & Am2 & Am3 & Am4,
                bx = Bm4 ^ (Am4 & Bm3) ^ (Am4 & Am3 & Bm2) ^ (Am4 & Am3 & Am2
& Bm1);
    endmodule
```

分析： 16 位 CLA 部件，接受 16 位输入 A 和 B，以及进位输入 c0，并产生 16 位进位输出 bx 和 1 位进位输出 ax。

5.32 位并行进位加法器顶层模块

```
    module Adder_32(A,B,S,C32);
         input [32:1] A;
         input [32:1] B;
         output [32:1] S;
         output C32;
         wire ax1,bx1,ax2,bx2;
         wire c16;
      CLA_16 CLA1(
          .A(A[16:1]),
            .B(B[16:1]),
            .c0(0),
            .S(S[16:1]),
            .ax(ax1),
            .bx(bx1)
        );
      CLA_16 CLA2(
            .A(A[32:17]),
              .B(B[32:17]),
              .c0(c16),
              .S(S[32:17]),
              .ax(ax2),
```

```
        .bx(bx2)
    );
  assign c16 = bx1 ^ (ax1 && 0), //c0 = 0
         C32 = bx2 ^ (ax2 && c16);
endmodule
```

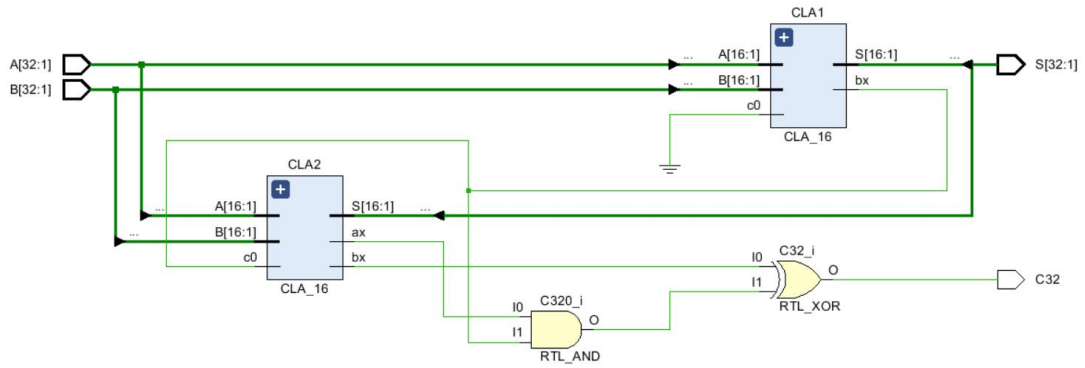
分析：32 位并行进位加法器的顶层模块，接受两个 32 位输入 A 和 B，并产生 32 位和输出 S 和 1 位进位输出 C32。它使用两个 16 位 CLA 部件（CLA1 和 CLA2）连接，其中 CLA2 的进位输入由 CLA1 的 c16 输出控制。

仿真代码

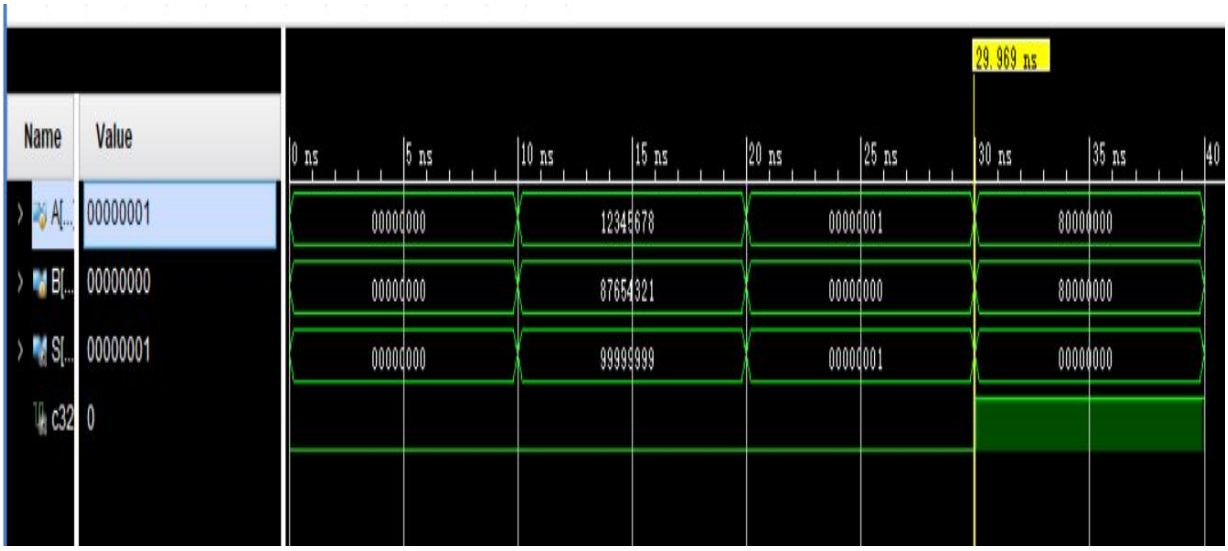
```
module sim_Adder_32;
  reg [32:1] A;
  reg [32:1] B;
  wire [32:1] S;
  wire c32;
  Adder_32 Adder_32(
       .A(A),
        .B(B),
        .S(S),
        .C32(c32)
       );
    initial begin
            // 初始化输入
            A = 32'h0;
            B = 32'h0;
            // 仿真测试用例
            #10 A = 32'h1234_5678;
            B = 32'h8765_4321;
            #10 A = 32'h1;
            B = 32'h0;
            #10 A = 32'h8000_0000;
            B = 32'h8000_0000;
            #10 $finish;
        end
Endmodule
```

分析：模拟了先行进位加法器的功能，包括输入 A 和 B 的初始化，以及不同的测试用例。在仿真中，输入 A 和 B 会随时间变化，以验证先行进位加法器的正确性。

电路图



波形图



波形图中正确显示了 32 位全加器的运算结果，包括有无输出进位。加数为 0x000000000x 和 0x00000000 时输出为 0x00000000，加数为 0x12345678 和 0x87654321 时输出为 0x99999999，加数为 0x00000001 和 0x00000000 时输出为 0x00000001。前三种情况均无进位。当加数为 0x80000000 和 0x80000000 时，输出为 0x00000000，且在此种情况有有进位，c32 为进位输出为 1.满足要求。

五、调试和心得体会

本次实验的主要目的是利用 case 语句、 always 语句编写模块，从而实现全加器、 4 位全加器、16 位全加器、 32 位全加器、 ALU 等模块。通过完成实验，我掌握了 always、assign、 case 等语句的详细用法， 并且熟练掌握了 Verilog 模块化设计的一般思路，并且用 Verilog 实现了数字电路中所学的全加器、 4 位全加器、16 位全加器、 32 位全加器、ALU 等内容，对数电所学知识得到了复习回顾。除了在设计电路方面， 在设计行为仿真的过程中我也有所收获，通过对各个模块的仿真，我熟练掌握了对模块输入输出信号的处理，尤其是统一时钟信号 clk 的仿真，此外我学会了利用$random 功能实现随机仿

真。同时，在实验过程中学习的基础上，结合模块化设计思想，设计了先行进位加法器，并用仿真对其结果正确性进行了验证，满足了要求。总体来说，受益良多。