实验报告

实验目的：

1、熟悉vivado设计流程

2、掌握利用vivado创建设计的方法（以实现4位加法器为例）

3、掌握编写testbench的方法，以及行为仿真方法

实验环境：

vivado版本号为2019.2

原理说明：

1、4位超前进位加法器的原理

超前进位加法器的思想是并行计算进位以缩短关键路径。

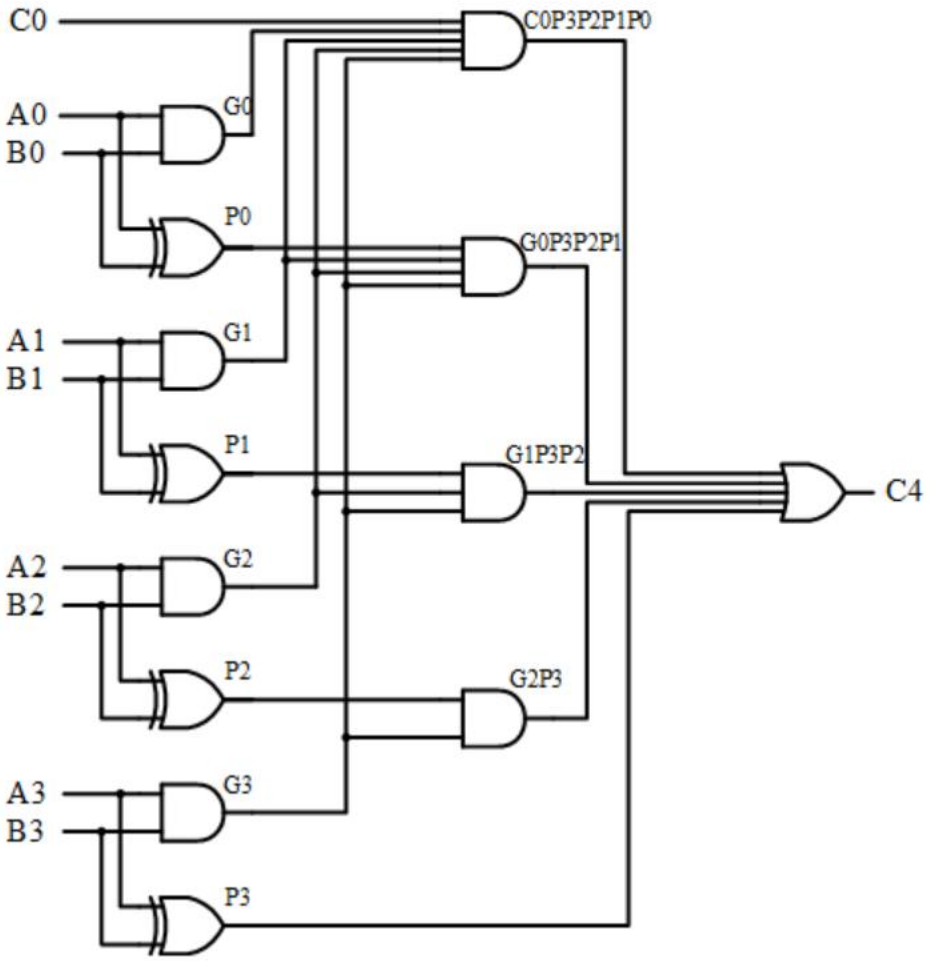
具体逻辑运算如图所示：

$$\begin{cases} P_k = A_k \oplus B_k, & \text{k=0,...,N-1} \\ G_k = A_k B_k & \text{k=0,...,N-1} \end{cases}$$

$$\begin{cases} S_k = P_k \oplus C_k, & \text{k=1,...,N} \\ C_k = G_{k-1} + C_{k-1} P_{k-1} & \text{k=1,...,N} \\ C_N = C_{out} \\ C_0 = C_{in} \end{cases}$$

$$\begin{cases} C_1 = G_0 + C_0 P_0 \\ C_2 = G_1 + C_1 P_1 = G_1 + G_0 P_1 + C_0 P_1 P_0 \\ C_3 = G_2 + C_2 P_2 = G_2 + G_1 P_2 + G_0 P_2 P_1 + C_0 P_2 P_1 P_0 \\ C_4 = G_3 + C_3 P_3 = G_3 + G_2 P_3 + G_1 P_3 P_2 + G_0 P_3 P_2 P_1 + C_0 P_3 P_2 P_1 P_0 \end{cases}$$

电路图如下所示：



2、1位全加器的原理：

1位全加器的本位值其实就是数A,B,CIN中1的个数，当1的个数为奇数时，本位值为1，反之为0。进位值则是数A,B,CIN中1的个数是否大于等于2，若大于等于2则为1，反之为0。

具体逻辑运算如下所示：

$$\begin{cases} S = A \oplus B \oplus C_i \\ C_o = AB + C_i(A \oplus B) \end{cases}$$

3、输出32位二进制数从低到高的第一个1的index（若全0输出32）的原理

使用for循环让器从第0位开始比较，当找到第一个1或者遍历后仍然找不到1时停止循环。

接口定义：

1、4位超前进位加法器接口：

名称	方向	位宽	功能描述
A	IN	4	第一个加数
B	IN	4	第二个加数
CIN	IN	1	低位进位输入
SUM	OUT	4	和
COUT	OUT	1	高位进位输出

2、1位全加器接口：

名称	方向	位宽	功能描述
A	IN	1	第一个加数
B	IN	1	第二个加数
CIN	IN	1	低位进位输入
SUM	OUT	1	和
COUT	OUT	1	高位进位输出

3、输出32位二进制数从低到高的第一个1的index（若全0输出32）的模块：

名称	方向	位宽	功能描述
A	IN	32	输入的32位二进制数
index	IN	6	从低到高第一个1的位置

调试过程及结果波形：

本次实验较为简单，写完后并未出过bug，所以无调试过程。

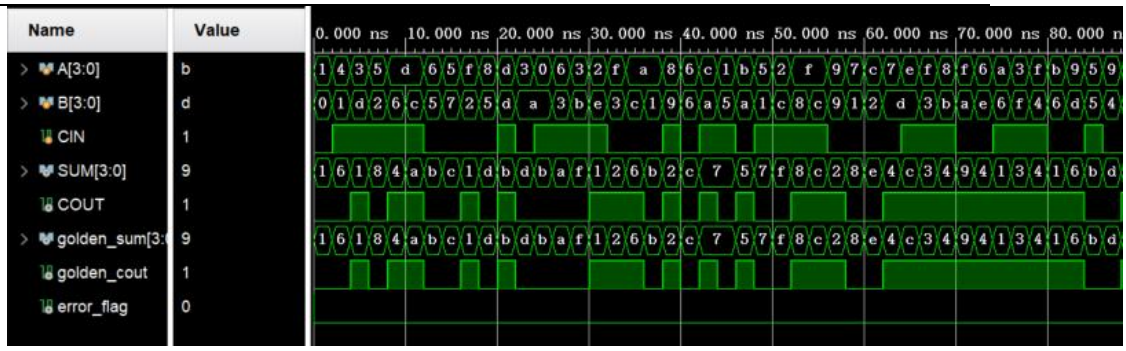
1、4位超前进位加法器：

激励文件：

```
module test_add_4();
    reg [3:0] A;
    reg [3:0] B;
    reg CIN;
    wire [3:0] SUM;
    wire COUT;
    add4 u_add4(
        .A(A),
        .B(B),
        .CIN(CIN),
        .SUM(SUM),
        .COUT(COUT)
    );
    initial begin
        A = 4'h1;
        B = 4'h0;
        CIN = 1'd0;
    end
    always begin
        #2;
        A = $random() % 16;
        B = $random() % 16;
        CIN = $random() % 2;
    end
    wire [3:0] golden_sum;
    wire golden_cout;
    wire error_flag;
    assign {golden_cout,golden_sum} = A + B + CIN;
    assign error_flag = (golden_sum != SUM) || (golden_cout != COUT);
endmodule
```

可以使用vivado中自带的加法生成金标准golden_sum与golden_cout，然后申明一个变量error_flag代表自编写的4位超前进位加法器输出与金标准是否一致，若一致则为0，否则为1。这种方法可以让验证更为方便，无需人为的一一比较输出。

波形图如下所示：



error_flag从未被拉高，所以设计的4位超前进位加法器正确。

2、1位全加器：

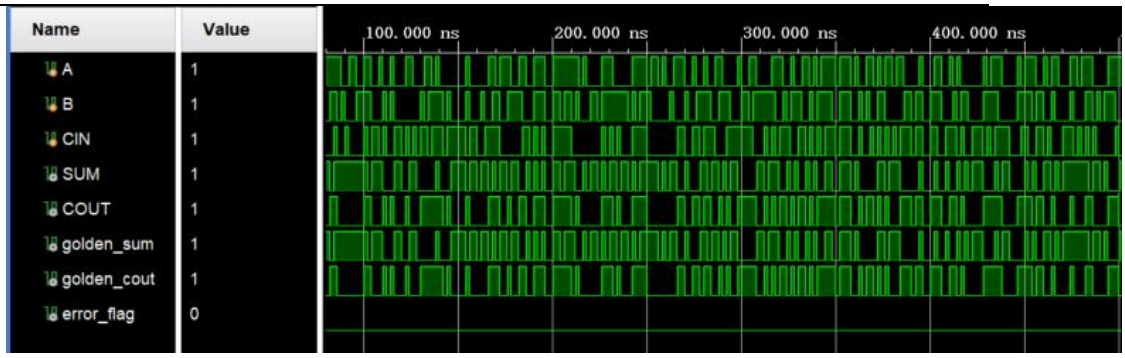
激励文件：

```
module test_add1(

    );
    reg A;
    reg B;
    reg CIN;
    wire SUM;
    wire COUT;
    add1 u_add1(
        .A(A),
        .B(B),
        .CIN(CIN),
        .SUM(SUM),
        .COUT(COUT)
    );
    initial begin
        A = 1'd1;
        B = 1'd0;
        CIN = 1'd0;
    end
    always begin
        #2;
        A = $random() % 2;
        B = $random() % 2;
        CIN = $random() % 2;
    end
    wire golden_sum;
    wire golden_cout;
    wire error_flag;
    assign {golden_cout,golden_sum} = A + B + CIN;
    assign error_flag = (SUM != golden_sum) || (COUT != golden_cout);
endmodule
```

同样地创建了金标准和error_flag信号用于判断1位全加器是否正确。这种方法可以让验证更为方便，无需人为的一一比较输出。

波形图如下所示：



error_flag从未被拉高，所以设计的1位全加器正确。

3、输出32位二进制数从低到高的第一个1的index（若全0输出32）的模块：

激励文件：

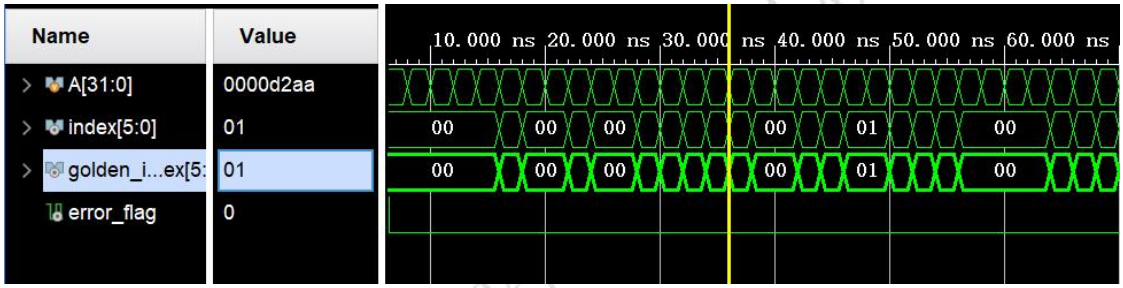
```
module test_cnt(

    );
    reg [31:0] A;
    wire [5:0] index;
    cnt u_cnt(
        .A(A),
        .index(index)
    );
    initial begin
        A  = 32'd0;
    end
    always begin
        #2;
        A = $random() & 32'hffff;
    end
    wire [5:0] golden_index;
    assign golden_index = A[0] ? 6'd0 :
                          A[1] ? 6'd1 :
                          A[2] ? 6'd2 :
                          A[3] ? 6'd3 :
                          A[4] ? 6'd4 :
                          A[5] ? 6'd5 :
                          A[6] ? 6'd6 :
                          A[7] ? 6'd7 :
                          A[8] ? 6'd8 :
                          A[9] ? 6'd9 :
                          A[10] ? 6'd10 :
                          A[11] ? 6'd11 :
                          A[12] ? 6'd12 :
                          A[13] ? 6'd13 :
                          A[14] ? 6'd14 :
                          A[15] ? 6'd15 :
                          A[16] ? 6'd16 :
                          A[17] ? 6'd17 :
                          A[18] ? 6'd18 :
                          A[19] ? 6'd19 :
                          A[20] ? 6'd20 :
                          A[21] ? 6'd21 :
                          A[22] ? 6'd22 :
                          A[23] ? 6'd23 :
```

```
                    A[24] ? 6'd24 :
                    A[25] ? 6'd25 :
                    A[26] ? 6'd26 :
                    A[27] ? 6'd27 :
                    A[28] ? 6'd28 :
                    A[29] ? 6'd29 :
                    A[30] ? 6'd30 :
                    A[31] ? 6'd31 : 6'd32;
    wire error_flag;
    assign error_flag = golden_index != index;
endmodule
```

采用三目运算，枚举法生成golden_index，然后生成error_flag信号。这种方法可以让验证更为方便，无需人为的一一比较输出。

波形图如下所示：



error_flag从未被拉高，所以设计的模块正确。

实验总结：

本次实验比较简单，大致20分钟就可以完成3个任务，即使verilog新手入门也可以很好地完成任务。

事实上题目中并未要求4位加法器采用先行进位或超前进位等方法，但是笔者认为正常的算法门延迟太高了，索性编写了超前进位的加法器。

同样题目中并未要求1位加法器是全加器，笔者看到部分同学编写代码时并未考虑低位进位CIN，个人并不建议那样的写法，还是得考虑CIN更好。

最后一题可以使用枚举法、FOR循环比较、二分法等方法。笔者采用枚举法生成金标准，用FOR循环比较编写模块。

源代码：

1、4位超前进位加法器：

```verilog
module add4(
  input [3:0] A,
  input [3:0] B,
  input CIN,
  output [3:0] SUM,
  output COUT
);

wire [3:0] G;
wire [3:0] P;
wire [3:0] C;

// Generate G and P signals
assign G[0] = A[0] & B[0];
assign P[0] = A[0] ^ B[0];
assign G[1] = A[1] & B[1];
assign P[1] = A[1] ^ B[1];
assign G[2] = A[2] & B[2];
assign P[2] = A[2] ^ B[2];
assign G[3] = A[3] & B[3];
assign P[3] = A[3] ^ B[3];

// Calculate carry values
assign C[0] = G[0] | (P[0] & CIN);
assign C[1] = G[1] | (P[1] & C[0]);
assign C[2] = G[2] | (P[2] & C[1]);
assign C[3] = G[3] | (P[3] & C[2]);

// Calculate sum values
assign SUM[0] = A[0] ^ B[0] ^ CIN;
assign SUM[1] = A[1] ^ B[1] ^ C[0];
assign SUM[2] = A[2] ^ B[2] ^ C[1];
assign SUM[3] = A[3] ^ B[3] ^ C[2];

// Output carry value
assign COUT = C[3];

endmodule
```

2、1位全加器：

```verilog
module add1(
    input A,
    input B,
    input CIN,
    output SUM,
    output COUT
    );
    assign SUM = A ^ B ^ CIN;
    assign COUT = (A & B) | ((A | B) & CIN);
endmodule
```

3、输出32位二进制数从低到高的第一个1的index（若全0输出32）的模块：

```
module cnt(
    input [31:0] A,
    output reg [5:0] index
);
integer i;
always @(*) begin
    index = 6'd32; // 默认设置为32，表示没有找到1
    for(i = 0;i < 32;i = i + 1) begin
        if(A[i] == 1'd1) begin
            index = i;
            i = 32;
        end
    end
end

endmodule
```