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

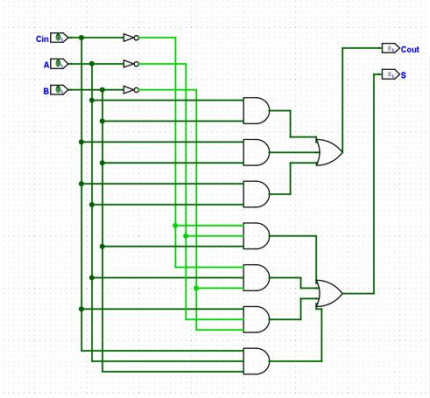
1 掌握 Vivado 或 Logisim 开发工具的使用，掌握以上电路的设计和测试方法；

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

对一位加法器模块的 logisim 实现



- 1 0 折叠重复行 显示所有行

显示8行(共8行)

Cin	A	B	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

输出表达式(双击可编辑):

Cout = A·B+Cin·B+Cin·A

$S = \overline{Cin}\cdot\overline{A}\cdot B+\overline{Cin}\cdot A\cdot\overline{B}+Cin\cdot\overline{A}\cdot\overline{B}+Cin\cdot A\cdot B$

使用 logisim 软件实现一位加法器；

全加器是一种组合逻辑电路，用于执行二进制加法操作。它能够同时处理三个输入：两个加数（通常称为 A 和 B）和一个进位输入（通常称为 Cin），并产生两个输出：和（Sum，通常称为 S）和进位（Carry Out，通常称为 Cout）。

全加器的原理如下：

1.A 和 B 是要相加的两个二进制位。Cin 是来自前一位的进位（对于一位全加器，这是初始进位）。

2. S 是 A、B 和 Cin 的异或运算（XOR），这表示和的位。

S = A ⊕ B ⊕ Cin= ~ Cin· ~ A·B+ ~ Cin·A· ~ B+Cin· ~ A· ~ B+Cin·A·B，使用与或非三种门实现即使图中所示。

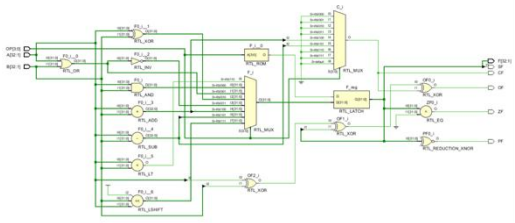
3. 进位输出（Cout）：Cout 是 A、B 和 Cin 的与运算（AND）以及 A 和 B 的或运算（OR）的组合。Cout = A · B + Cin ·A +Cin· B

全加器的工作原理是将输入的两个二进制位相加，并考虑来自前一位的进位，以计算和（S）和进位（Cout）。这使得多个全加器可以级联在一起，以执行多位二进制加法，确保正确的进位传递。全加器是计算机中的基本构建块，用于实现加法运算和其他逻辑操作。

ALU 设计：

行为描述方式：

```
22
23  module ALU(OP, A, B, F,ZF, CF, OF, SF, PF);
24      parameter SIZE = 32;
25      input [3:0] OP;
26      input [SIZE:1] A;
27      input [SIZE:1] B;
28      output [SIZE:1] F;
29      output CF, OF, SF, PF,ZF;
30      reg [SIZE:1] F;
31      reg C,ZF,CF,OF,SF,PF;
32
33      always @( *)
34      begin
35          C = 0;
36          case(OP)
37              4'b0000: begin F = A&B; end
38              4'b0001: begin F = A|B; end
39              4'b0010: begin F = A^B; end
40              4'b0011: begin F = ~(A|B); end
41              4'b0100: begin {C,F} = A + B; end
42              4'b0101: begin {C,F} = A - B; end
43              4'b0110: begin F = A < B; end
44              4'b0111: begin F=B<<A; end
45          endcase
46          ZF=F==0;
47          CF=C;
48          OF=A[SIZE]^B[SIZE]^F[SIZE]^C;
49          SF=F[SIZE];
50          PF=~^F;
51      end
52
53  endmodule
54
```



这是一个使用 Verilog 编写的 ALU（算术逻辑单元）模块，用于执行不同的操作，如与、或、异或、加法、减法、比较等。

定义了一个 Verilog 模块，名为 ALU，它包含了输入、输出和一些内部寄存器。

parameter SIZE = 32;：定义了一个名为 SIZE 的参数，它被设置为 32。这个参数用于指定数据的位数。

输入和输出端口：

input [3:0] OP;：一个 4 位宽的输入端口，用于指定要执行的操作的操作码。

input [SIZE:1] A;：一个 SIZE+1 位宽的输入端口，用于接收操作的第一个操作数。

input [SIZE:1] B;：一个 SIZE+1 位宽的输入端口，用于接收操作的第二个操作数。

output [SIZE:1] F;：一个 SIZE+1 位宽的输出端口，用于存储操作的结果。

output CF, OF, SF, PF, ZF;：五个单独的输出端口，分别表示进位标志（CF）、溢出标志（OF）、符号标志（SF）、奇偶标志（PF）和零标志（ZF）。

reg [SIZE:1] F;：定义了一个内部寄存器 F，用于存储操作的结果。

reg C, ZF, CF, OF, SF, PF;：定义了一些内部寄存器，用于存储进位标志（C）、零标志（ZF）、进位标志（CF）、溢出标志（OF）、符号标志（SF）和奇偶标志（PF）的值。

always @(*)：这是一个组合逻辑块，表示其中的操作会在输入发生变化时立即执行。

begin 和 end：这标记了 always 块的开始和结束。
C = 0;：将进位标志 C 初始化为 0。
case(OP)：基于操作码（OP）的值，使用 case 语句选择不同的操作。
在每个操作的分支中，计算操作的结果，并将结果存储在寄存器 F 中，同时更新进位标志 C。
计算各种标志位，包括零标志（ZF）、进位标志（CF）、溢出标志（OF）、符号标志（SF）和奇偶标志（PF）的值。
这个模块实际上是一个通用的 ALU，根据操作码（OP）的不同值，执行不同的算术和逻辑操作，并生成相应的标志位。最终结果存储在 F 中。这个模块可以用于执行多种运算，如与、或、异或、加法、减法和比较等。不同的操作由操作码指定。
RTL 分析如图示可见

ALU 三种描述方式：

```
module ALU_8(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;

    parameter ALU_AND = 4'b0000;
    parameter ALU_OR = 4'b0001;
    parameter ALU_XOR = 4'b0010;
    parameter ALU_NOR = 4'b0011;
    parameter ALU_ADD = 4'b0100;
    parameter ALU_SUB = 4'b0101;
    parameter ALU_SLT = 4'b0110;
    parameter ALU_SLL = 4'b0111;

    wire [7:0] EN;
    wire [SIZE-1:0] Fw, Fa;

    assign Fa = A & B;

    always@(*) begin
        case (OP)
            ALU_AND: begin F<=Fa; end
            ALU_OR:  begin F<= A|B;end
            ALU_XOR: begin F <=A^B;end
            ALU_NOR: begin F <= ~(A|B);end
            default: F = Fw;
        endcase
    end

    Decoder38 decoder38_1(OP[2:0], EN[7:0]);

    ADD add_1(Fw, CF, A, B, EN[4]);
    SUB sub_1(Fw, CF, A, B, EN[5]);
    SLT slt_1(Fw, A, B, EN[6]);
    SLL sll_1(Fw, A, B, EN[7]);

endmodule
```

```
module SLT(F,A,B,EN);
    parameter N=32;
    output reg [N-1:0]F;
    input [N-1:0]A,B;
    input EN;

    always@(A,B,EN)begin
        if((A<B)&&EN) F<=1;
        else F<=32'bz;
    end
endmodule
```

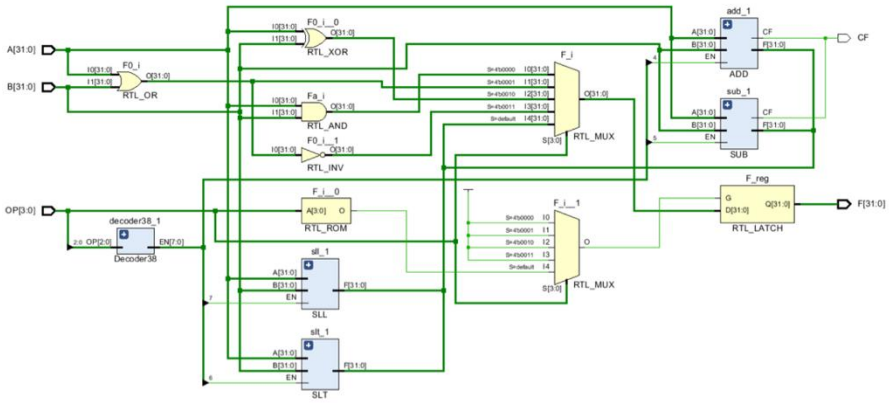
```
module SUB(F,CF,A,B,EN);
   parameter N=32;
   output reg [N-1:0]F;
   output reg CF;
   input [N-1:0]A,B;
   input EN;

   always @(*) begin:
      if (EN==1)begin
         if(A<B)begin F<=B-A; CF=1;end
         else begin{CF, F} <= A - B;end
      end
      else begin
         F = 32'bz;
         CF=1'bz;
      end
   end

endmodule
```

```
module ADD(F,CF,A,B,EN);
    parameter N=32;
    output reg [N-1:0]F;
    output reg CF;
    input [N-1:0]A,B;
    input EN;

    always @(*)begin
        if (EN) begin
            {CF, F} = A + B;
        end else begin
            F = 32'bz; CF=1'bz;
        end
    end

endmodule
```

```
module SLL(F,A,B,EN);
    parameter N=32;
    output reg [N-1:0]F;
    input [N-1:0]A,B;
    input EN;

    always@(A,B,EN)begin
        if(EN==1) F<=B<<A;
        else F<=32'bz;
    end
endmodule
```

```
module Decoder38(OP, EN);
input [2:0]OP;
output reg [7:0]EN;
    always @(*)begin
        case (OP)
            3'b000: EN <= 8'b00000001;
            3'b001: EN <= 8'b00000010;
            3'b010: EN <= 8'b00000100;
            3'b011: EN <= 8'b00001000;
            3'b100: EN <= 8'b00010000;
            3'b101: EN <= 8'b00100000;
            3'b110: EN <= 8'b01000000;
            3'b111: EN <= 8'b10000000;
            default: EN <= 8'b00000000;
        endcase
    end
endmodule
```

```
module ALU_sim;

    reg [31:0] A, B;
    reg [3:0] OP;
    wire [31:0] F;
    wire CF;

    ALU_8 alu(
        .F(F),
        .CF(CF),
        .A(A),
        .B(B),
        .OP(OP)
    );


    initial begin

        B = 32'h55555555;
        A = 32'hAAAA5555;

        OP = 4'b0000;
        #10 OP = 4'b0001;
        #10 OP = 4'b0010;
        #10 OP = 4'b0011;
        #10 OP = 4'b0100;
        #10 OP = 4'b0101;
        #10 OP = 4'b0110;
        #10 OP = 4'b0111;
    end


endmodule
```



ALU_8:

1. parameter SIZE = 32; 定义了模块的参数 SIZE，指定 ALU 的位宽为 32 位。
2. output reg [SIZE-1:0] F;定义了一个 32 位输出端口 F，用于存储 ALU 的计算结果。
3. output CF; 定义了一个进位输出端口 CF，用于处理算术操作时可能产生的进位。
4. input [SIZE-1:0] A, B;定义了两个 32 位输入端口 A 和 B，用于输入操作数。
5. input [3:0] OP;定义了一个 4 位输入端口 OP，用于选择 ALU 的操作模式。
6. 接下来，使用 parameter 定义了一系列操作码，如 ALU_AND、ALU_OR 等，用于指定不同的 ALU 操作。
7. wire [7:0] EN; 定义了一个 8 位的 wire，用于传递到 ALU 的各个子模块以控制它们的使能。
8. wire [SIZE-1:0] Fw, Fa;定义了两个 wire，Fw 用于存储 ALU 操作后的结果，Fa 用于存储 AND 操作的结果。
9. assign Fa = A & B;用于将 A 和 B 的按位 AND 结果赋给 Fa。
10. always @(*)开始一个组合逻辑块，根据 OP 的值执行不同的操作。根据 OP 的不同，选择 ALU 的操作，并将结果存储在 F 中。
11. case (OP) 语句根据 OP 的值执行不同的操作，如 AND、OR、XOR 等，对应于不同的操作码。

12. Decoder38 decoder38_1(OP[2:0], EN[7:0]);创建了一个 3-8 解码器，根据 OP 的低 3 位来选择子模块的使能信号 EN。
13. 接下来，根据 EN 中的不同位来选择执行 ADD、SUB、SLT 或 SLL 子模块，从而实现不同的操作。
这段代码定义了 ALU 模块的输入、输出和控制逻辑，它根据 OP 的值执行不同的操作，同时调用子模块来执行具体的算术和逻辑操作。在模块内部使用了 assign 语句来计算 AND 操作的结果 Fa。这是 ALU 模块的第一部分，其他子模块的定义和实现在后续代码段中。

Decoder38:

这是 ALU 模块的第二段代码，它定义了一个名为 Decoder38 的模块，用于根据输入的 3 位操作码 OP 生成一个 8 位的使能信号 EN。以下是对代码的逐个分析:
1. input [2:0] OP;定义了一个 3 位的输入端口 OP，表示 ALU 的操作码。
2. output reg [7:0] EN; 定义了一个 8 位的输出端口 EN，用于表示子模块的使能信号。
3. always @(*) 开始一个组合逻辑块，当 OP 变化时，根据不同的操作码值，为 EN 分配不同的使能信号。
4. case (OP)语句根据 OP 的值执行不同的操作。
5. 针对每个可能的 3 位操作码值，分别分配相应的 8 位 EN 信号。例如，当 OP 为 3'b000 时，EN 被设置为 8'b00000001，以此类推。
6. 在 default 语句中，如果 OP 的值不匹配任何已知的操作码，那么 EN 被设置为 8'b00000000，表示没有使能任何子模块。

这段代码的作用是根据输入的 3 位操作码 OP 生成一个 8 位的使能信号 EN，用于控制 ALU 模块内的子模块的执行。EN 的不同位分别对应不同的子模块，根据操作码值，可以选择执行特定的子模块以执行相应的操作。

SLT:

这是 ALU 模块的第三段代码，定义了一个名为 SLT 的子模块，用于执行比大小的操作。以下是对代码的逐个分析:
1. parameter N = 32; 定义了模块的参数 N，指定 ALU 的位宽为 32 位。
2.output reg [N-1:0] F; 定义了一个 32 位的输出端口 F，用于存储 SLT 操作的结果。
3.input [N-1:0] A, B;定义了两个 32 位的输入端口 A 和 B，用于比较操作。
4.input EN;定义了一个使能信号 EN，用于控制 SLT 操作的执行。
5. always @(A, B, EN) 表示组合逻辑块会在 A、B 或 EN 变化时执行。
6. 在 always 块中，使用条件语句判断，如果 A 小于 B 且 EN 为真，那么 F 被设置为 1，表示 A 小于 B。否则，F 被设置为 32 位的高阻态（32'bz）。

这段代码实现了根据输入的 A、B 以及使能信号 EN，判断 A 是否小于 B，并将结果存储在输出端口 F 中。如果条件成立，F 被设置为 1，否则 F 被设置为高阻态表示不满足条件。这是 ALU 模块的一个子模块，用于实现比较操作。

SUB:

这是 ALU 模块的第四段代码，定义了一个名为 SUB 的子模块，用于执行减法操作。以下是对代码的逐个分析:

1. parameter N = 32;定义了模块的参数 N，指定 ALU 的位宽为 32 位。
2. output reg [N-1:0] F;定义了一个 32 位的输出端口 F，用于存储减法操作的结果。
3. output reg CF; 定义了一个输出端口 CF，用于存储借位（Carry）标志。
4.input [N-1:0] A, B;定义了两个 32 位的输入端口 A 和 B，表示被减数和减数。
5.input EN;定义了一个使能信号 EN，用于控制减法操作的执行。
6.always @(*) 表示组合逻辑块会在输入信号 A、B 或 EN 变化时执行。
7. 在 always 块中，首先检查使能信号 EN 是否为 1。如果 EN 为 1，表示执行减法操作。
8. 如果 A 小于 B，则 F 被设置为 B 减去 A 的结果，并将 CF 设置为 1 表示没有借位。
9. 否则，使用非阻塞赋值 {CF, F} <= A - B; 来同时计算 CF 和 F 的结果，确保正确的借位处理。
10. 如果 EN 不为 1，表示不执行减法操作，那么 F 和 CF 都被设置为高阻态（32'bz 和 1'bz）。

这段代码实现了减法操作，根据输入的 A、B 以及使能信号 EN，计算减法结果并正确处理借位。 F 存储减法的结果，CF 存储借位标志。这是 ALU 模块的一个子模块，用于执行减法操作。

ADD:
这是 ALU 模块的第五段代码，定义了一个名为 ADD 的子模块，用于执行加法操作。以下是对代码的逐个分析:
1. parameter N = 32;定义了模块的参数 N，指定 ALU 的位宽为 32 位.
2. output reg [N-1:0] F; 定义了一个 32 位的输出端口 F，用于存储加法操作的结果。
3. output reg CF;定义了一个输出端口 CF，用于存储进位（Carry）标志。
4. input [N-1:0] A, B;定义了两个 32 位的输入端口 A 和 B，表示被加数和加数。
5. input EN;定义了一个使能信号 EN，用于控制加法操作的执行。
6. always @(*) 表示组合逻辑块会在输入信号 A、B 或 EN 变化时执行。
7. 在 always 块中，首先检查使能信号 EN 是否为真（非零）。如果 EN 为真，表示执行加法操作。
8. 使用非阻塞赋值{CF, F} = A + B; 来同时计算 CF 和 F 的结果，确保正确处理进位。
9. 如果 EN 不为真，表示不执行加法操作，那么 F 和 CF 都被设置为高阻态（32'bz 和 1'bz）。

这段代码实现了加法操作，根据输入的 A、B 以及使能信号 EN，计算加法结果并正确处理进位。 F 存储加法的结果，CF 存储进位标志。这是 ALU 模块的一个子模块，用于执行加法操作。

SLL:

这是 ALU 模块的第六段代码，定义了一个名为 SLL 的子模块，用于执行逻辑左移位操作。以下是对代码的逐个分析:
1. parameter N = 32; 定义了模块的参数 N，指定 ALU 的位宽为 32 位。
2. output reg [N-1:0] F;定义了一个 32 位的输出端口 F，用于存储逻辑左移位操作的结果。
3.input [N-1:0] A, B; 定义了两个 32 位的输入端口 A 和 B，表示要左移的数据和左移的位数。

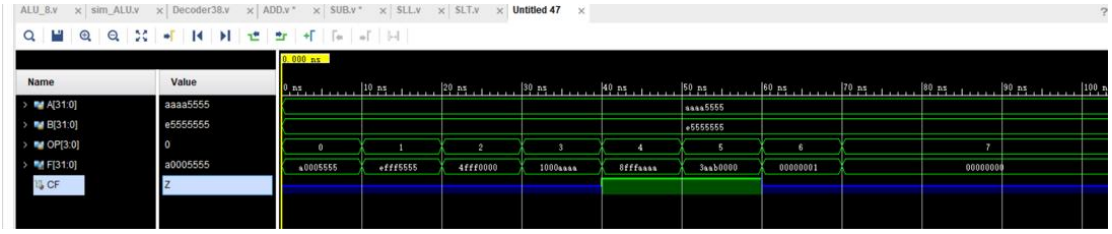
4. input EN; 定义了一个使能信号 EN，用于控制逻辑左移位操作的执行。
5. always @(A, B, EN) 表示组合逻辑块会在输入信号 A、B 或 EN 变化时执行。
6. 在 always 块中，首先检查使能信号 EN 是否为 1。如果 EN 为 1，表示执行逻辑左移位操作。
7. 使用逻辑左移位操作 F <= B << A; 来将输入数据 B 左移 A 位，并将结果存储在输出端口 F 中。
8. 如果 EN 不为 1，表示不执行逻辑左移位操作，那么 F 被设置为高阻态（32'bz）。

这段代码实现了逻辑左移位操作，根据输入的 A 和 B 以及使能信号 EN，对数据 B 进行逻辑左移位操作，并将结果存储在输出端口 F 中。如果不执行操作，则输出 F 为高阻态。这是 ALU 模块的一个子模块，用于执行逻辑左移位操作。

总体分析这六段代码，它们一起构成了一个基本的 ALU（算术逻辑单元）模块，用于执行不同的算术和逻辑操作。以下是对整个 ALU 模块的总体分析：

1. ALU 模块的整体结构由这六个子模块组成，其中包括 ALU_8（主模块）、Decoder38（操作码解码器）、SUB（减法模块）、ADD（加法模块）、SLT（小于比较模块）和 SLL（逻辑左移模块）。参数 N 被用来定义 ALU 的位宽，而其他参数如操作码常量（ALU_AND、ALU_OR 等）用于区分不同的操作。
2. ALU_8 主模块：ALU_8 主模块包含了输入端口 A 和 B，操作码端口 OP，以及输出端口 F 和 CF（进位标志）。根据操作码 OP，它选择执行不同的操作，并调用相应的子模块来执行。
3. Decoder38 模块：Decoder38 子模块负责将 3 位的操作码 OP 解码为 8 位的使能信号 EN。这些使能信号用于选择执行哪个子模块，从而实现不同的操作。
4. SUB 模块：SUB 子模块用于执行减法操作。根据输入 A、B 和 EN，它计算减法结果和进位标志 CF，并将结果存储在输出端口 F 和 CF 中。
5. ADD 模块：ADD 子模块用于执行加法操作。类似于 SUB 模块，它根据输入 A、B 和 EN，计算加法结果和进位标志 CF，并将结果存储在输出端口 F 和 CF 中。
6. SLT 模块：SLT 子模块用于执行小于比较操作。根据输入 A、B 和 EN，它判断 A 是否小于 B，并将结果存储在输出端口 F 中，同时用高阻态表示 CF。
7. SLL 模块：SLL 子模块用于执行逻辑左移位操作。根据输入 A、B 和 EN，它将数据 B 左移 A 位，并将结果存储在输出端口 F 中。
8. 总体逻辑：ALU_8 主模块根据操作码 OP 选择执行不同的操作，其余子模块则根据输入执行相应的操作，同时将结果存储在 F 和 CF 中。高阻态（32'bz）被用于表示未执行操作的情况。
这一系列子模块组成了一个完整的 ALU，能够执行多种算术和逻辑操作。

simulation:



仿真文件程序 ALU_sim，它模拟了 ALU 的操作，根据不同的操作码 OP 和输入数据 A、B 来测试 ALU 的功能。以下是对仿真文件程序的分析以及其时序输出：
1. reg [31:0] A, B; 定义了两个 32 位寄存器 A 和 B，用于存储输入数据。
2. reg [3:0] OP; 定义了一个 4 位寄存器 OP，用于存储操作码。
3. wire [31:0] F; 定义了一个 32 位的 wire F，用于接收 ALU 计算的结果。
4. wire CF; 定义了一个 wire CF，用于接收进位标志。
5. ALU 模块 ALU_8 被实例化为 alu，并通过端口连接 A、B、OP、F 以及 CF。这允许在仿真中使用 ALU 模块来执行操作。
6. 在 initial 块中，首先初始化输入数据 B 和 A，分别为 32'he5555555 和 32'hAAAA5555。
7. 然后，通过设置不同的操作码 OP 来测试 ALU 的不同操作。每个操作码 OP 以 10 个时间单位的间隔进行测试。
现在让我们来分析仿真的时序输出，以了解 ALU 在不同操作下的行为：
OP = 4'b0000：这表示 ALU 执行 AND 操作。根据提供的输入数据 A 和 B，ALU 应该执行 A & B，并将结果存储在 F 中。可以得到 F=a0005555;
OP = 4'b0001：这表示 ALU 执行 OR 操作。ALU 应该执行 A | B，并将结果存储在 F 中。得到 F=efff5555;
OP = 4'b0010：这表示 ALU 执行 XOR 操作。ALU 应该执行 A ^ B，并将结果存储在 F 中。得到 F=4fff0000;
OP = 4'b0011：这表示 ALU 执行 NOR 操作。ALU 应该执行 ~(A | B) 并将结果存储在 F 中。得到 F=1000aaaa;
OP = 4'b0100：这表示 ALU 执行 ADD 操作。ALU 应该执行 A + B 并将结果存储在 F 中。得到 F=8fffaaaa；有 CF=1 的进位标志;
OP = 4'b0101：这表示 ALU 执行 SUB 操作。ALU 应该执行 A - B 并将结果存储在 F 中，同时设置 CF 标志。得到 F=3aab0000；因为借位有 CF=1;
OP = 4'b0110：这表示 ALU 执行 SLT 操作。ALU 应该判断 A 是否小于 B 并将结果存储在 F 中。因为 A<B 得到 F=00000001;
OP = 4'b0111：这表示 ALU 执行 SLL 操作。ALU 应该执行 B 左移 A 位并将结果存储在 F 中。得到 F=00000000;

五、调试和心得体会

1. 模块化设计原则：ALU 模块采用模块化的设计方法，不同的操作被分成独立的子模块，如加法、减法、逻辑运算等。这种模块化设计使得每个操作更容易理解和维护。
2. 操作码解码：ALU 模块使用操作码来确定执行哪种操作。在设计中，操作码的解码非常关键，因为它决定了 ALU 将执行哪个操作。通过适当的操作码解码，我们可以选择正确的子模块来执行操作。
3. 组合逻辑和时序逻辑：在 ALU 模块的设计中，我学会了如何处理组合逻辑和时序逻辑。组合逻辑用于生成输出只基于当前输入，而时序逻辑考虑了信号随时间的变化。了解这两者的区别和应用对于正确的设计至关重要。
4. 仿真和调试：在编写 ALU 模块的代码时，我学会了如何使用仿真工具来验证模块的功能。这是发现和修复逻辑错误的关键步骤，可以在硬件制造前排除问题。

5. 应用能力：ALU 模块是计算机中一个核心的模块，用于执行多种算术和逻辑操作。这些操作在计算机的指令执行中发挥重要作用，因此 ALU 模块的设计和实现对于计算机体系结构的理解和应用至关重要。

总的来说，ALU 模块的设计教会了我如何将计算机硬件模块化，根据操作码选择执行不同的操作，以及如何在硬件层面实现常见的算术和逻辑运算。这些知识和经验对于深入理解计算机组成原理和数字电路设计非常有帮助。在未来的计算机工程领域，我将能够更自信地设计和理解硬件模块。