实验报告——ALU 运算单元

一、实验设计

1. 实验的**输入、输出**及对应的**管脚**：

变量名	类型	描述	宽度	管脚
op	input	操作符	2	T8，U8
a	input	输入信号 A	4	J15，L16,M13,R15
b	input	输入信号 B	4	R17，T18，U18，R13
out	output	输出的 F	4	H17，K15，J13，N14
Cout	output	进位输出	1	R18

备注：

① value(a)= $\Sigma a[i]*2^i(i=0,1,2,3)$

② **以上表中的管脚顺序对应于相应变量的二进制表示的顺序为：从权重较小的位到权重较大的位（如： op[0]对应的是 T8，op[1]对应的是 U8）**

2. **半加器的设计**

(1)输入：a(4-bit)， b(4-bit)

(2)返回值：out(4-bit)， Cout(1-bit)

(3)分析：

① a 与 b 相加，若得到的和不足八位，高位上补零；

② 得到的和的低四位为 out；

③ 得到的和的最高位为 Cout，原因是：若 a 与 b 相加有进位，那么和的最高位上为 1，否则为 0，恰好与 Cout 的取值一致，因此可以直接将和的最高位赋给 Cout

3. **半减器的设计**

(1)输入：a(4-bit)，b(4-bit)

(2)返回值：out(4-bit)，Cout(1-bit)

(3)分析：

① a 与 b 相减，若得到的差不足八位，高位上补零；

② 得到的差的低四位为 out；

③ 得到的差的最高位为 Cout，原因是：若 a 与 b 相减有借位，那么和的最高位上为 1，否则为 0，恰好与 Cout 的取值一致，因此可以直接将差的最高位赋给 Cout

4. **取反操作的设计**

(1)输入：a(4-bit)

(2)返回值：out(4-bit)

(3)分析：直接对 a 取反赋给 out

5. **乘法器的设计**

(1)输入：a(4-bit)，b(4-bit)

(2)返回值：out(4-bit)，Cout(1-bit)

(3)分析：

a 与 b 相乘，得到的乘积最多有 8 位，低四位就是 out，如果高四位

上的值大于 0，说明有进位，Cout 为 1，否则为 0

6. **代码实现(如下图)，其中 couttmp 中是做加、减、乘法运算时高四位，outtmp 是低四位。如果运算结束后，couttmp 的值大于 0，表示有进位，Cout=1，否则为 0**

```
module ALU(
  input [1:0]op,
  input [3:0]a, b,
  output  [3:0]out,
  output Cout
  );
  reg [3:0] outtmp;
  reg [3:0] couttmp;
  initial begin
    couttmp = 0;
  end
  always @(op, a, b) begin
    case(op)
      2'b0: {couttmp, outtmp} = a + b;
      2'b01: {couttmp, outtmp} = a - b;
      2'b10: outtmp = ~a;
      default: {couttmp, outtmp} = a * b;
    endcase
  end
  assign out = outtmp;
  assign Cout = (couttmp>0);
endmodule
```

二、仿真波形

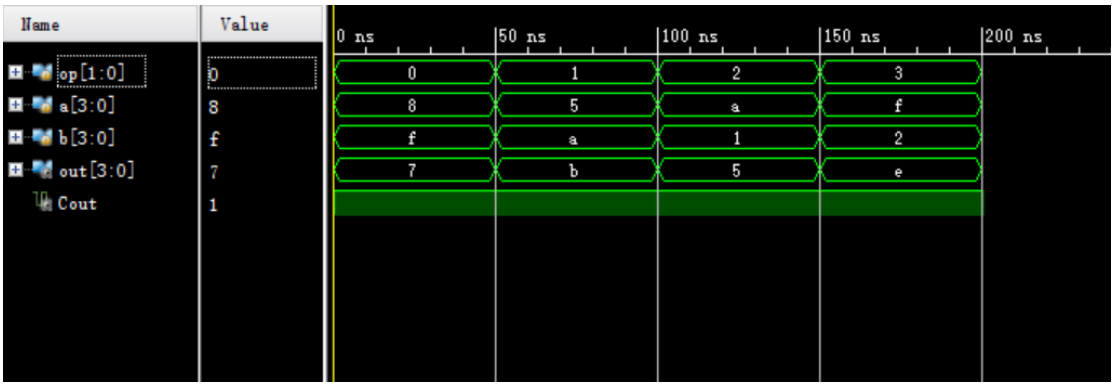
1. 测试代码：

```
module test_ALU();
   reg [1:0]op;
   reg [3:0]a, b;
   wire [3:0]out;
   wire Cout;
   ALU test(op, a, b, out, Cout);
   initial begin
      op = 2'b0;
      a = 4'b0;
      b = 4'b0;
   end
   initial begin
         a = 4'b1000; b = 4'b1111; op = 0;
         #50  a = 4'b0101; b = 4'b1010; op = 1;
         #50  a = 4'b1010; b = 1; op = 2;
         #50  a = 4'b1111; b = 4'b0010; op = 3;
    end
   initial begin
      $monitor($stime,,, "{%b %b %b} = {%b, %b}", a, op, b, out, Cout);
      #200
      $finish;
   end
endmodule
```

2. 我针对四种运算，选取了四组数据进行分别测试，测试数据在控制台的输出如下：（格式为$time {a op b}={out, Cout}）

```
  0 {1000 00 1111} = {0111, 1}
 50 {0101 01 1010} = {1011, 1}
100 {1010 10 0001} = {0101, 1}
150 {1111 11 0010} = {1110, 1}
```

3. 对应的仿真波形：



三、实验中遇到的问题和解决思路

在此次实验中我遇到了好多问题，大部分都是因为对 vivado 设计流程不熟悉导致的，下面列出来的是我认为比较有意义的两个问题，也算是经验和教训吧：

1. [Synth 8-2576] procedural assignment to a non-register led is not permitted

   这个是 verilog 的语法错误，经过查找一些教程，我对这个问题的初步的理解如下：

   (1)概念：**连续赋值、过程赋值**

   连续赋值：在连续赋值语句中，表达式右侧的计算结果可以立即更新表达式的左侧

   过程赋值：表达式右侧的计算结果在某种条件的触发（如时钟的上升沿）下放到一个变量中

   (2)概念：**wire 类型、reg 类型**

   wire：表示直通，只要输入有变化，输出马上无条件地反映

   reg：表示一定要有触发，输出才会反应输入

   (3)**连续赋值和过程赋值的区别：**

连续赋值	过程赋值
主要赋值给 nets	主要赋值给 reg 变量
有关键词 assign 标识	没有 assign 标识
不可以用于 always 和 initial 等过程块中（上面 ERROR 的原因）	可以用在 always 和 initial 等过程块中
只于右端的计算结果有关	赋值是有“过程”的，即与上下文其他语句有关

   (4)**wire 和 reg 的区别：**

Wire	reg
不保存状态，它的值可以随时改变，不受时钟信号限制	可以保存输出状态。状态改变通常在下一个时钟信号边沿翻转时进行
可以用 assign 赋值	不可以用 assign 这种过程赋值的方式赋值
input 一定是 wire 类型	output 两种类型皆可

2. **在烧制 FPGA 时，已经用数据线连接了电路板，但是始终无法打开，查了好多资料，最后才发现是电源没有打开，白白耗费了很多时间**

四、实验中的收获

1. 了解了一些 verilog 的基础语法
2. 了解了 vivado 的设计流程和 FPGA 的烧制过程
3. 对于时序电路和组合电路，从硬件方面有了更加底层的认识

五、实验的感想

这次实验中接触到的所有内容都是之前没有了解过的，所以中间过程有点艰辛，bug 百出，但同时收获很多，做完之后也很有成就感，在此过程中，很感谢助教的耐心答疑。