数电实验EXP3

刘永鹏 191220070 计科 693901492@qq.com 2020.9.14

1. 实验目的

根据硬件资源，完成一个**进行补码加减运算的 4 位加减运算器**，此加减 运算器的核心部件是一个 4 位加法器，能够根据控制端完成加、减运算，并能 判断结果是否为 0，是否溢出，是否有进位等。这里，输入的操作数 A 和 B 都 已经是补码。

设计一个能实现如下功能的简单 ALU

功能选择	功能	操作
000	加法	A+B
001	减法	A-B
010	取反	Not A
011	与	A and B
100	或	A or B
101	异或	A xor B
110	比较大小	If A>B then out=1; else out=0;
111	判断相等	If A==B then out=1; else out=0;

由于**ALU已经把加法器的功能囊括在内**，我们下面不再单独讨论加法器，即task 3-1。

2. 实验原理

1. 加减法

   原理：即基于补码的运算。利用补码表示下**模运算加减一致**的特性完成设计。

   利用3.2中，八位加法器的实现，可以轻易完成加法功能的实现。

   首先，采用如下的算式计算溢出与进位（零置位较简单）：

$$Overflow = (in_x n-1 == in_y n-1)\&\&(out_s n-1 != in_x n-1) \tag{1}$$
$$out_c, out_s = in_x + in_y$$

   溢出为正常的溢出位，但进位和正常进位有所不同：它表示**在加法运算过程中，操作数的最高位向外是否有进位。**

   对于减法，只需将减数**按位取反加一**，即可转化为加法逻辑进行处理。

2. 取反，与，或，异或，判断相等：

   这些操作都可以用现有的Verilog操作符直接完成，较为简单，不再赘述。

3. 判断大小：

   根据符号位进行分类讨论，按照比较规则为LEDR[0]赋值。

```
if(SW[7]&&!SW[3]) //符号位
    LEDR[0] = 1;
else if(SW[3]&&!SW[7]) //符号位
    LEDR[0] = 0;
else if(SW[3]&&SW[7])
begin
    LEDR[0] = (SW[6]&&!SW[2]) || ((SW[6]==SW[2])&&SW[5]&&!SW[1])
    || ((SW[6]==SW[2])&&(SW[5]==SW[1])&&SW[4]&&!SW[0]);
end
else
begin
    LEDR[0] = !((SW[6]&&!SW[2]) || ((SW[6]==SW[2])&&SW[5]&&!SW[1])
    || ((SW[6]==SW[2])&&(SW[5]==SW[1])&&SW[4]&&!SW[0]) || (SW[3:0] ==
SW[7:4]));
end
```

3. 实验环境

1. Quartuz + systembuilder, 仍然使用systembuilder自动生成项目。
2. SW[3:0]与SW[7:4]分别作为两个操作数，在减法时，后者作为减数。
3. **LEDR[3:0]作为result, LEDR[4]为进位，LEDR[5]为溢出位，LEDR[6]则为零置位。**（在加减法时）
4. 其他操作与上述基本相同。、
5. key（按钮）作为控制ALU功能的控制器。

4. 程序代码/流程图

下面是case部分的代码：

```
always @ (*)
begin
    case(KEY[2:0])
    3'b000:begin //加法
        {LEDR[4],LEDR[3:0]} = SW[3:0] + later[3:0];
        LEDR[5] = (SW[3] == later[3])&&(LEDR[3] != SW[3]);
        LEDR[6] = (LEDR[3:0] == 4'b0000)?1:0;
    end
    3'b001:begin //减法
    later[3:0] = ~SW[7:4] + 1;
        {LEDR[4],LEDR[3:0]} = SW[3:0] + later[3:0];
        LEDR[5] = (SW[3] == later[3])&&(LEDR[3] != SW[3]);
        LEDR[6] = (LEDR[3:0] == 4'b0000)?1:0;
    end
    3'b010:begin //取反
        LEDR[3:0] = ~SW[3:0];
    end
    3'b011:begin //与
        LEDR[3:0] = SW[3:0] & SW[7:4];
    end
    3'b100:begin //或
        LEDR[3:0] = SW[3:0] | SW[7:4];
    end
    3'b101:begin //异或
        LEDR[3:0] = SW[3:0] ^ SW[7:4];
    end
    3'b110:begin //比较大小
```

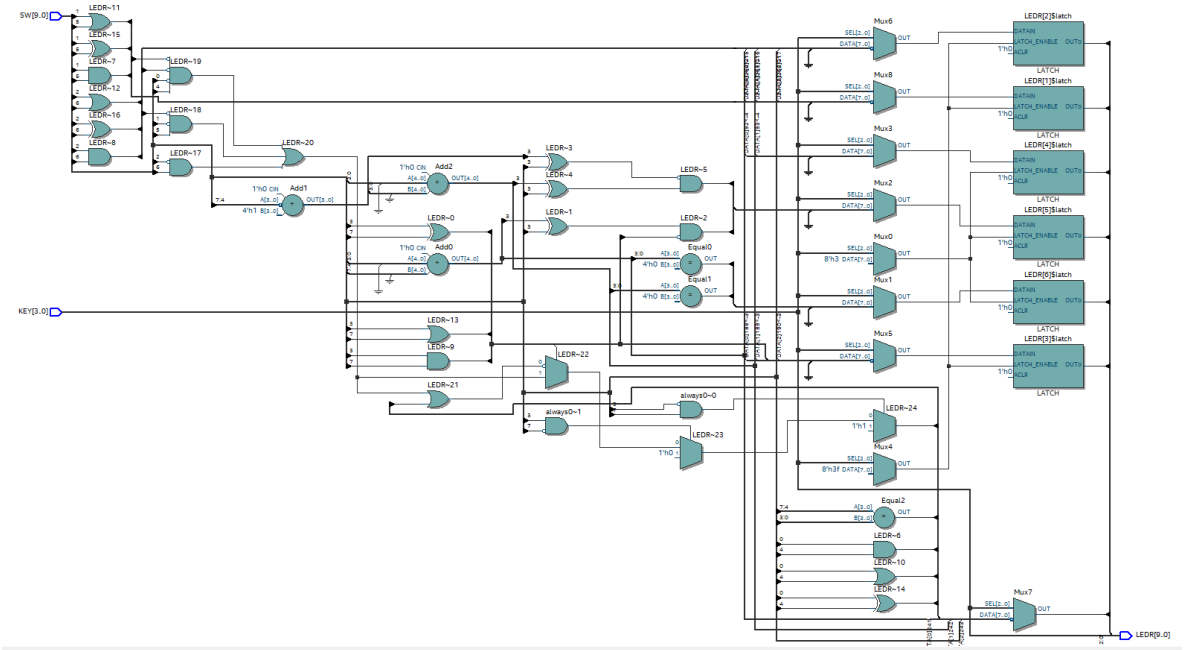
```
        if(SW[7]&&!SW[3])
            LEDR[0] = 1;
        else if(SW[3]&&!SW[7])
            LEDR[0] = 0;
        else if(SW[3]&&SW[7])
        begin
            LEDR[0] = (SW[6]&&!SW[2]) || ((SW[6]==SW[2])&&SW[5]&&!SW[1])
            || ((SW[6]==SW[2])&&(SW[5]==SW[1])&&SW[4]&&!SW[0]);
        end
        else
        begin
            LEDR[0] = !((SW[6]&&!SW[2]) || ((SW[6]==SW[2])&&SW[5]&&!SW[1])
            || ((SW[6]==SW[2])&&(SW[5]==SW[1])&&SW[4]&&!SW[0]) || (SW[3:0]
== SW[7:4]));
        end
    end
    3'b111:begin //比较相等
        LEDR[0] = (SW[3:0] == SW[7:4])?1:0;
    end
    endcase
```

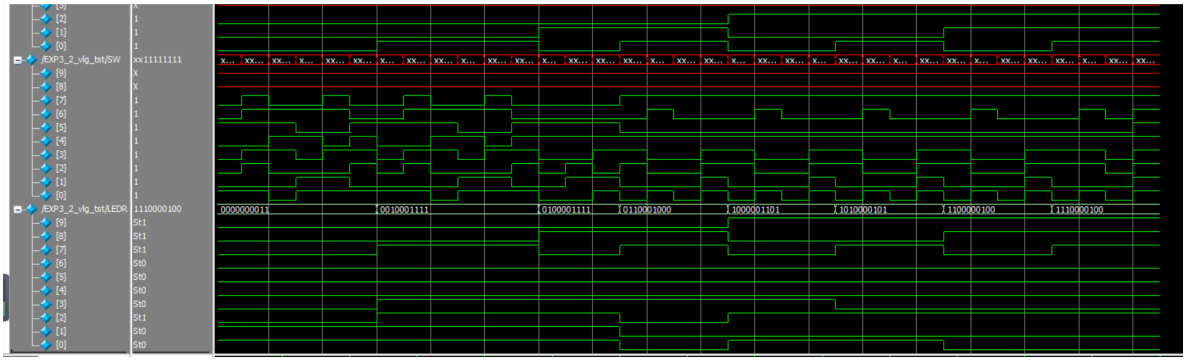
接下来是testbench部分的项目代码：

```
initial
begin
// code that executes only once
// insert code here --> begin
KEY[2:0] = 000; SW[3:0] = 1;    SW[7:4] = 2; #10; // 加
SW[3:0] = -3;   SW[7:4] = -2; #10;
SW[3:0] = 8;    SW[7:4] = 7; #10;
SW[3:0] = 3;    SW[7:4] = 5; #10;
SW[3:0] = -5;   SW[7:4] = -4; #10;
SW[3:0] = -3;   SW[7:4] = 3; #10;
KEY[2:0] = 001; SW[3:0] = 1;    SW[7:4] = 2; #10; //减
SW[3:0] = -3;   SW[7:4] = -2; #10;
SW[3:0] = 8;    SW[7:4] = 7; #10;
SW[3:0] = 3;    SW[7:4] = 5; #10;
SW[3:0] = -5;   SW[7:4] = -4; #10;
SW[3:0] = -3;   SW[7:4] = 3; #10;
KEY[2:0] = 010; SW[3:0] = 4'b0000; #10; //取反
SW[3:0] = 4'b0110; #10;
SW[3:0] = 4'b1011; #10;
KEY[2:0] = 011; SW[3:0] = 4'b1100;  SW[7:4] = 4'b1001; #10; //与
SW[3:0] = 4'b0001;  SW[7:4] = 4'b1101; #10;
SW[3:0] = 4'b0000;  SW[7:4] = 4'b1001; #10;
SW[3:0] = 4'b1111;  SW[7:4] = 4'b1001; #10;
KEY[2:0] = 100; SW[3:0] = 4'b1100;  SW[7:4] = 4'b1001; #10;//或
SW[3:0] = 4'b0001;  SW[7:4] = 4'b1101; #10;
SW[3:0] = 4'b0000;  SW[7:4] = 4'b1001; #10;
SW[3:0] = 4'b1111;  SW[7:4] = 4'b1001; #10;
KEY[2:0] = 101; SW[3:0] = 4'b1100;  SW[7:4] = 4'b1001; #10; //异或
SW[3:0] = 4'b0001;  SW[7:4] = 4'b1101; #10;
SW[3:0] = 4'b0000;  SW[7:4] = 4'b1001; #10;
SW[3:0] = 4'b1111;  SW[7:4] = 4'b1001; #10;
KEY[2:0] = 110; SW[3:0] = 4'b1100;  SW[7:4] = 4'b1001; #10; //比较大小
SW[3:0] = 4'b0001;  SW[7:4] = 4'b1101; #10;
SW[3:0] = 4'b0000;  SW[7:4] = 4'b1001; #10;
```

```
35  SW[3:0] = 4'b1111;  SW[7:4] = 4'b1001; #10;
36  KEY[2:0] = 111; SW[3:0] = 4'b1100;  SW[7:4] = 4'b1001; #10; //相等
37  SW[3:0] = 4'b1101;  SW[7:4] = 4'b1101; #10;
38  SW[3:0] = 4'b0000;  SW[7:4] = 4'b1001; #10;
39  SW[3:0] = 4'b1111;  SW[7:4] = 4'b1111; #10;
40  // --> end
41  $display("Running testbench");
42  end
```

接下来是仿真结果和RTL视图（应该看不出什么名堂来，过于复杂(＃°Д°)）。





5. 实验过程

1. 主要难点在加法器上，事实上，**根据所给exp03.pdf中的公式**即可以顺利完成该部分。
2. 实验流程仍然是：
    1. Systembuilder自动建立项目。
    2. 编写程序代码。
    3. 编译通过后自动生成Testbench。
    4. 连接Testbench, RTL仿真。
    5. 仿真结果无误后烧制代码在板上再次验证。

6. 测试方法

直接看波形图的方法几乎不可行：加减法的功能太过复杂。

因此，基于task调用的testbench使得测试更加方便。不过，由于如果使用TASK编写testbench，必然要用一段代码作为验证标准，而这段代码正是实验中用到的，这导致**循环证明**。

因此，我还是采用笨方法手写了testbench。

烧板验证确认无误。

7. 实验结果

确认无误。

8. 问题与解决方案

1. 逻辑上没有遇到太大困难，由于本次实验采用的**进位**概念与正常情况下有符号的进位不太相同，导致起初题目理解上有些问题。
2. 由于我们的开发板**坏了两个按钮！！！！！！**，导致我在烧板验证的时候还以为自己写错了。

9. 启示

应当进一步深刻理解task和function的使用，已更加便捷地进行仿真。

10. 意见与建议

暂无

11. 思考题

1. *在此加减运算的运算器中，如果用判断参与运算的加数和运算结果符号位 是否相同的方法来判断是否溢出，那么此时判断溢出位的时候，应该是比较操 作数 A，B 和运算结果的符号位，还是比较 A1、B1 和运算结果的符号位?*

应当比较**A1,B1,与运算结果的符号位。**

由于在做减法运算时，是将原来的第二个加数**取反加一**变为**减数**之后进行加法运算，因此使用加法器的溢出进位法则来进行判断的。

因此，不应当使用原数，而应当使用A1,B1。

2. ☞ **思考题 2**:

方法一:

```
assign t_no_Cin = {n{ Cin }}^B ;
assign {Carry,Result} = A + t_no_Cin + Cin;
assign Overflow = (A[n-1] == t_no_Cin[n-1]) && (Result [n-1] != A[n-1]);
```

方法二:

```
assign t_add_Cin =( {n{Cin}}^B )+ Cin ;  //  在这里请注意^运算和+运算的顺序
assign { Carry, Result } = A + t_add_Cin;
assign Overflow = (A[n-1] == t_add_Cin[n-1]) && (Result [n-1] != A[n-1]);
```

以上两种方法的产生的运算结果、进位位和溢出位值都是完全是一样的吗? 如果不一样，为什么结果会不一样呢? 在哪一步产生了差别? 哪一个正确?

不完全一样，第一种方法正确。

使用这种方法，可以将加减法统一进行判断。在**cin == 1**时，直接对减数取反+1。

第二种做法的错误在于，它**在取反+1的时候就进行加一运算**，可能会导致溢出，这样，虽然后续的运算结果正确，但可能会导致OF的判断不正确。

3.

|*result* **是逐位或。**也就是说，一个n位的二进制串逐一和比自己低的位进行“或”运算。

那么，

```
assign zero = ~(| Result);
```

的结果就是zero为0**当且仅当result每一位都为0。**

在程序中，让result和0直接比较比较方便，但使用上面的语句可以减少门电路的数量。