数字电路与数字系统实验

实验三

加法器和 ALU

计算机科学与技术系

191220008 陈南曈

924690736@qq.com

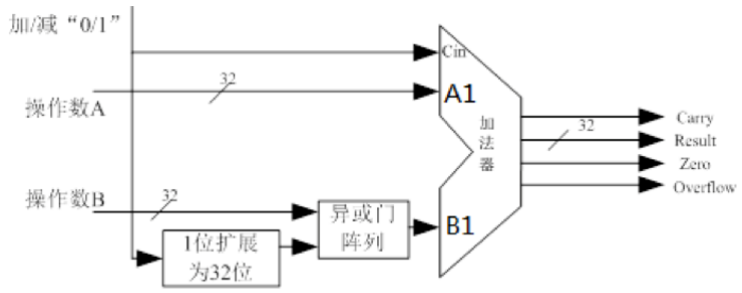
2020.9.15

一、实验目的

加法是数字系统中最常执行的运算，加法器是 ALU 的核心部件。减法可 以看作是被减数与取负后的减数进行加法。即用加法器同时实现加法和减法两 种运算。乘法也可以利用移位相加的算法来实现。因此，加法器可以说是计算 机中最“繁忙”的部件了。 本实验的目的是复习一位全加器的原理。

（一）简单加减法运算器的设计

请根据硬件资源，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。



（二）实现一个带有逻辑运算的简单 ALU

设计一个能实现如下功能的简单 ALU

在实现此 ALU 的时候，请考虑各种运算的进位位 C 和溢出位 overflow 位的输出。（一般情况下，涉及加减运算的，可以按照加减运算器来考虑进位位和溢出位；涉及逻辑运算的，可以直接设置进位位和溢出位为“0”。比较大小时需要考虑符号位。）

请实现对 4 位有符号数操作的 ALU，由于开发板上输入有限，可以使用 SW 作为数据输入，button 作为选择端。

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;

二、实验原理（知识背景）

1、加法器：

加法器是产生数的和的装置。加数和被加数为输入，和数与进位为输出的装置为半加器。若加数、被加数与低位的进位数为输入，而和数与进位为输出则为全加器。常用作计算机算术逻辑部件，执行逻辑操作、移位与指令调用。在电子学中，加法器是一种数位电路，其可进行数字的加法计算。三码，主要的加法器是以二进制作运算。由于负数可用二的补数来表示，所以加减器也就不那么必要。

2、ALU：

算术逻辑单元是能实现多组算术运算和逻辑运算的组合逻辑电路，简称 ALU。能够对 2 个 n 位的操作数进行若干不同的算术和逻辑操作，由一组功能选择输入来指定要执行的操作。

三、实验环境/器材等

1) 软件环境：

Quartus (Quartus Prime 17.1) Lite Edition

2) 硬件环境：

DE10-Standard 开发板

FPGA 部分：

Intel Cyclone V SE 5CSXFC6D6 F31C6N

- 110K 逻辑单元
- 5,761Kbit RAM

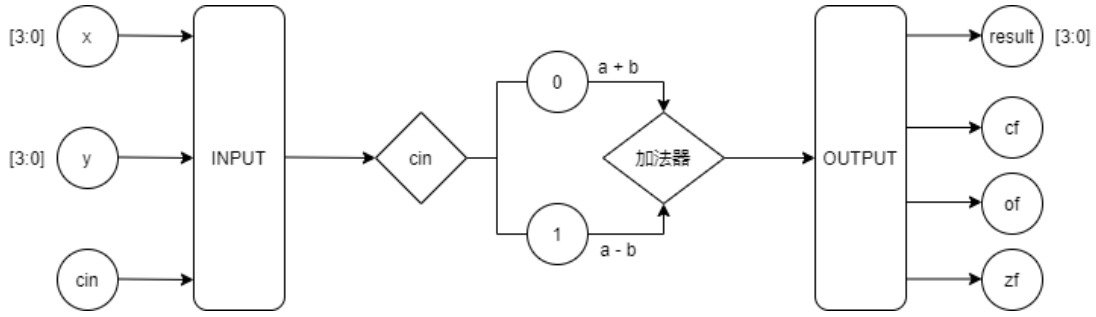
HPS 部分：

Dual-core ARM Cortex A9

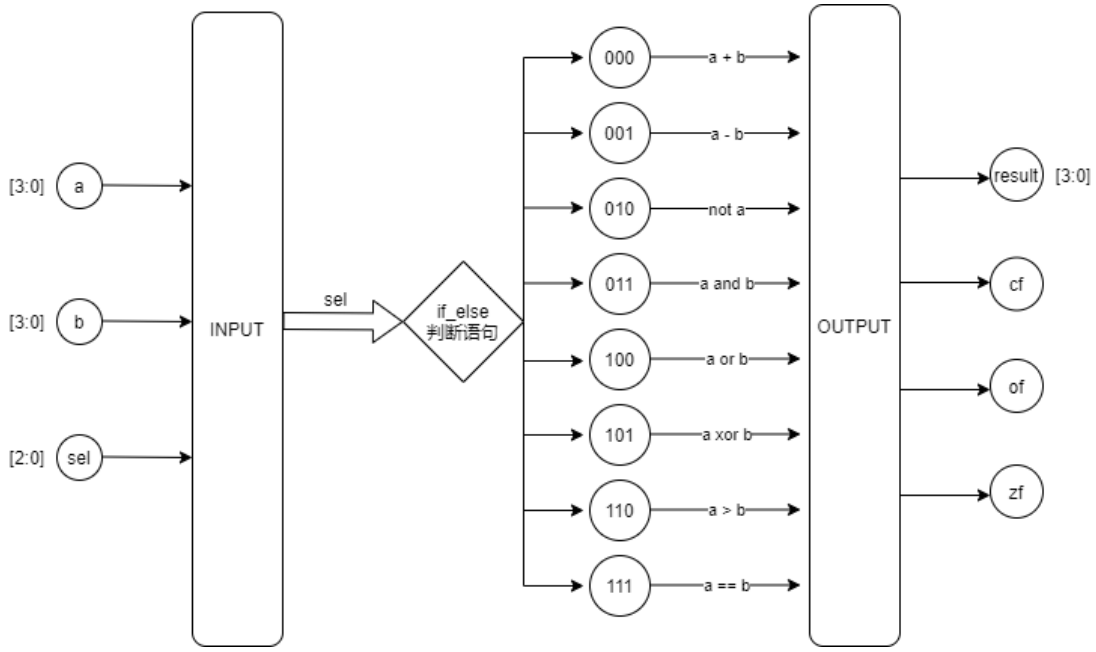
- 925MHz
- 1GB DDR

四、程序代码或流程图

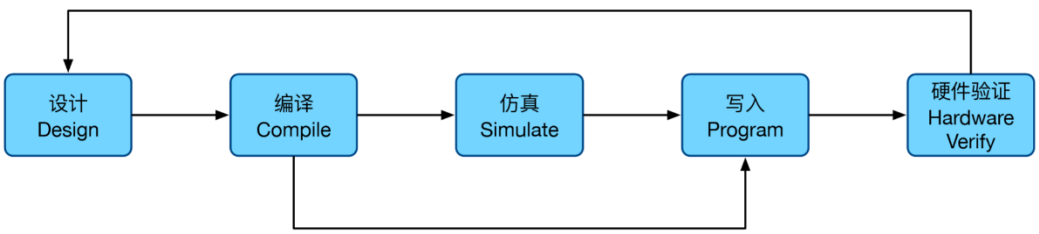
1、加减运算器：



2、ALU：



五、实验步骤/过程



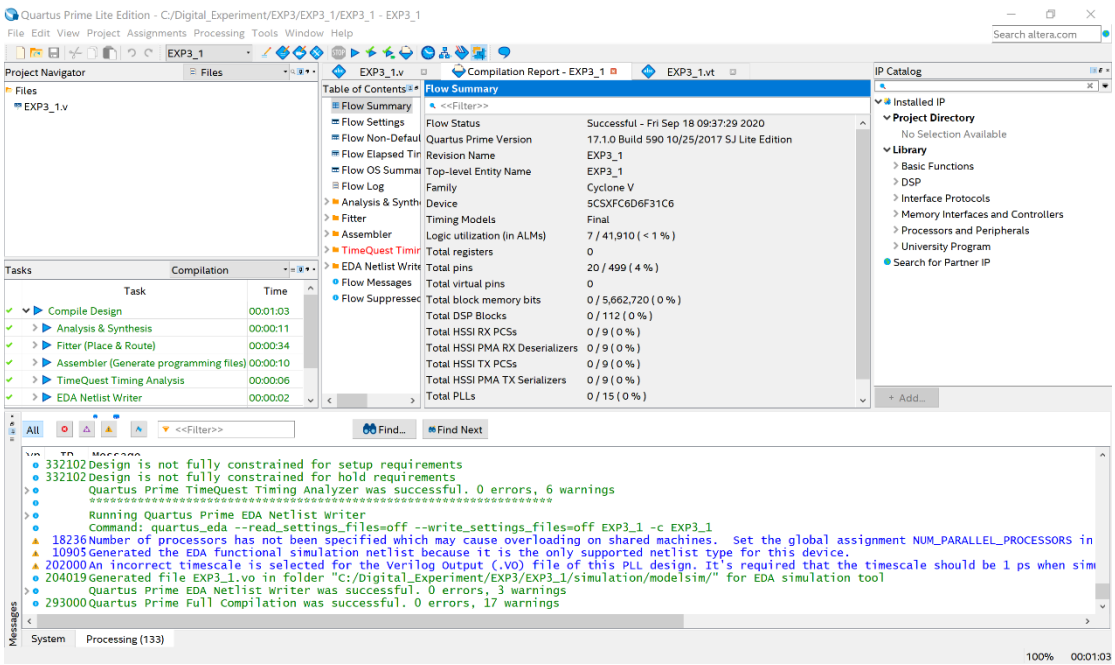
设计：

```
module EXP3_1(x, y, my, cin, result, cf, of, zf);
  input [3:0] x;
  input [3:0] y;
  input cin;

  output reg [3:0] my;
  output reg [3:0] result;
  output reg cf;
  output reg of;
  output reg zf;

  always @ (*)
  begin

    my = {4{cin}} ^ y;
    {cf, result} = x + my + cin;
    cf = cf ^ cin;
    of = (x[3] == my[3]) && (result[3] != x[3]);
    zf = ~(| result);
  end

endmodule
```

测试：

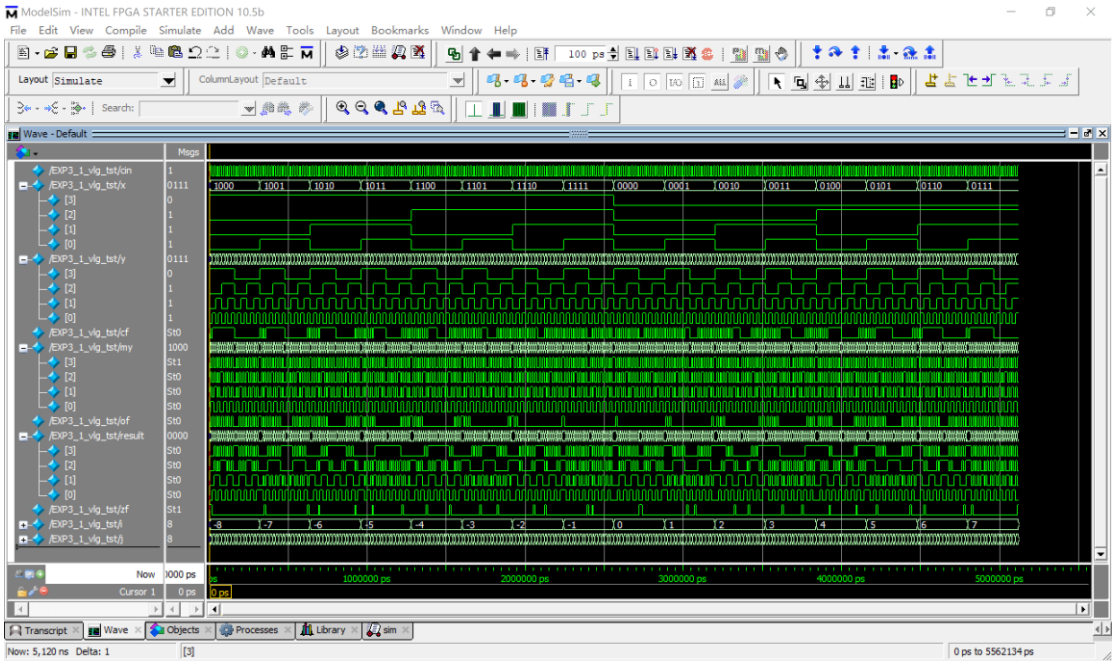
```
initial
begin
// code that executes only once
// insert code here --> begin
  for(i = -8; i <= 7; i = i + 1)
    for(j = -8; j <= 7; j = j + 1)
    begin
      cin = 0; x = i; y = j; #10;
      cin = 1; x = i; y = j; #10;
    end
// --> end
//$display("Running testbench");
end
```

编译：



引脚分配：无

仿真：



写入：无

硬件验证：无

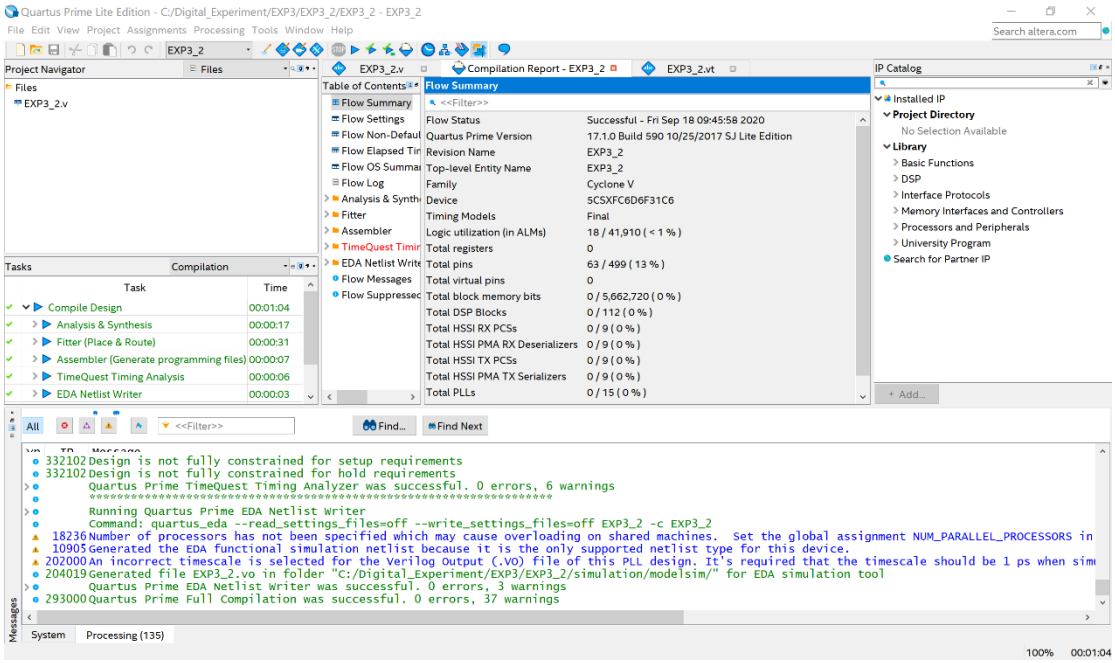
2、ALU：

设计：详见代码文件

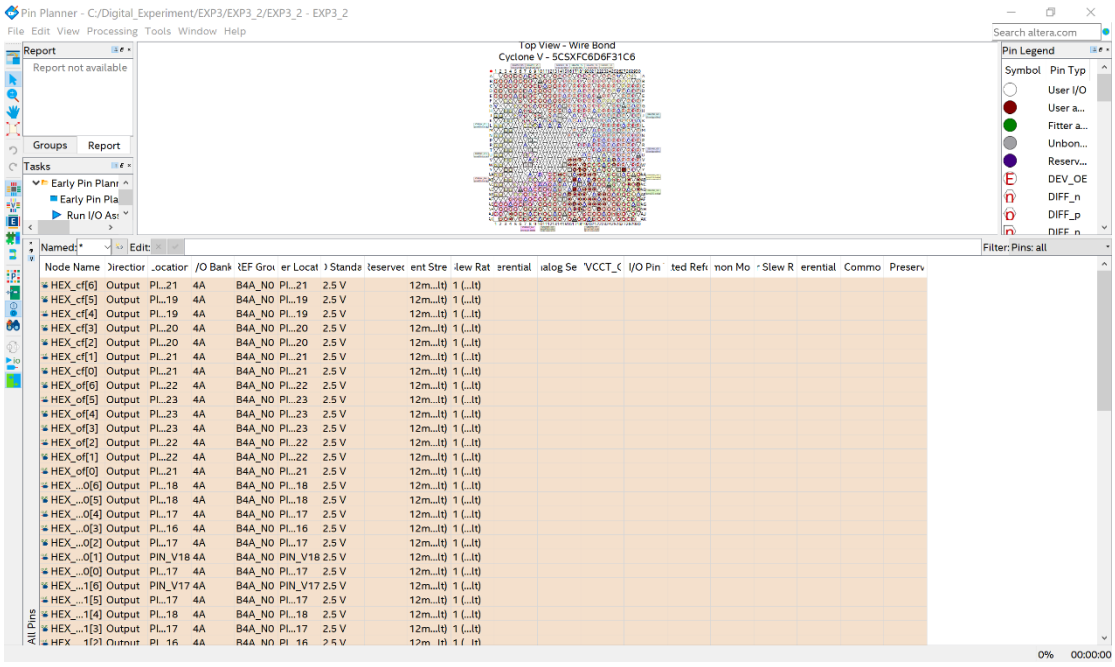
测试：

```
initial
begin
// code that executes only once
// insert code here --> begin
  for(i = 0; i <= 7; i = i + 1)
    for(j = 0; j <= 15; j = j + 1)
      for(k = 0; k <= 15; k = k + 1)
      begin
        sel = i; a = j; b = k; #10;
      end
// --> end
//$display("Running testbench");
end
```

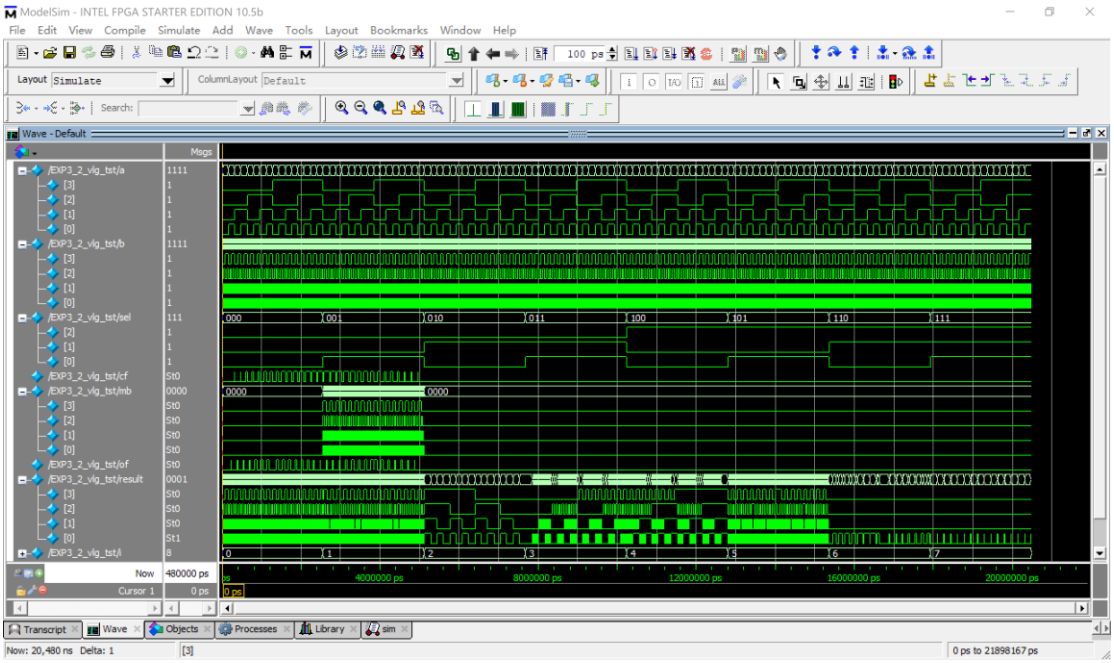
编译：



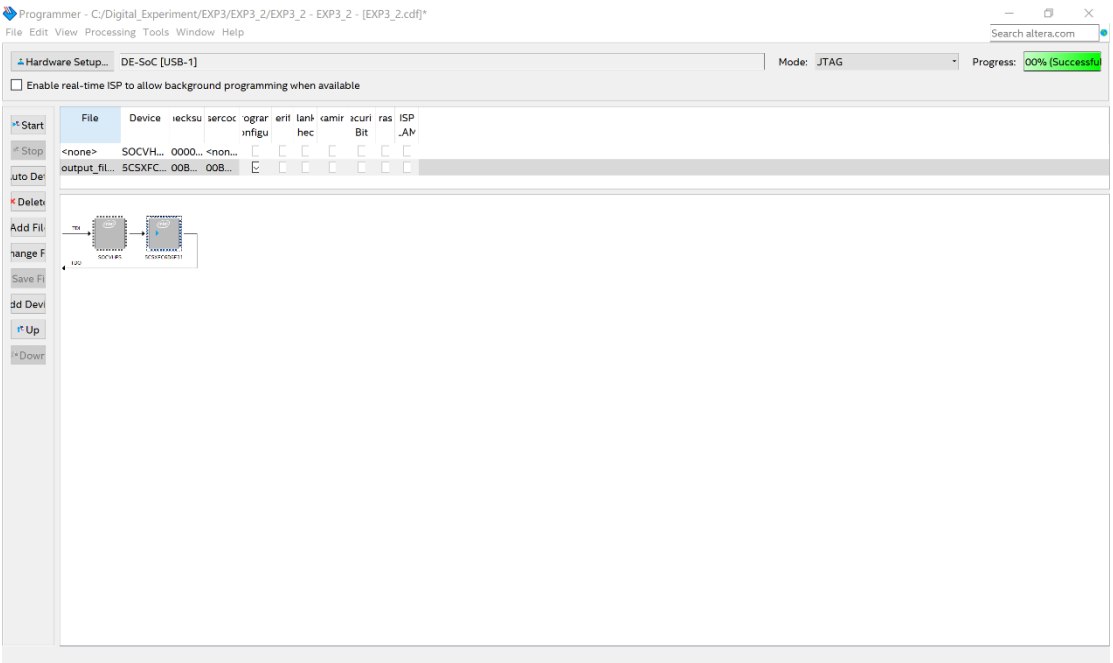
引脚分配：



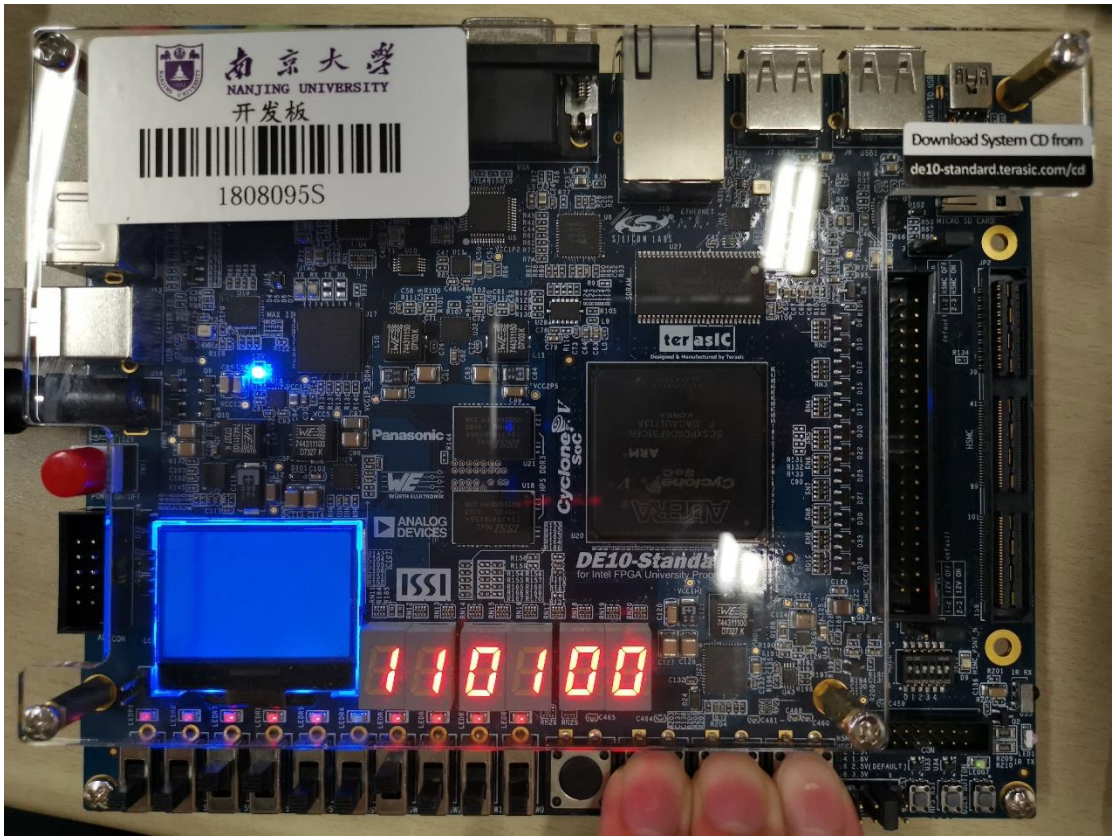
仿真：


ModelSim - INTEL FPGA STARTER EDITION 10.5b
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Wave - Default
/EXP3_2_vlg_tst/a
/EXP3_2_vlg_tst/b
/EXP3_2_vlg_tst/sel
/EXP3_2_vlg_tst/cf
/EXP3_2_vlg_tst/mb
/EXP3_2_vlg_tst/of
/EXP3_2_vlg_tst/result
/EXP3_2_vlg_tst/i
Now
Cursor 1
Transcript
Wave
Objects
Processes
Library
sim

写入：


Programmer - C:/Digital_Experiment/EXP3/EXP3_2/EXP3_2 - EXP3_2 - [EXP3_2.cdf]*
File Edit View Processing Tools Window Help
Hardware Setup...
DE-SoC [USB-1]
Mode: JTAG
Progress:
Enable real-time ISP to allow background programming when available
Start
Stop
Up
File
Device

硬件验证：



六、测试方法

利用 for 循环语句，将每种功能的每种输入情况依次遍历呈现

1、加减运算器

```
initial
begin
// code that executes only once
// insert code here --> begin
  for(i = -8; i <= 7; i = i + 1)
    for(j = -8; j <= 7; j = j + 1)
    begin
      cin = 0; x = i; y = j; #10;
      cin = 1; x = i; y = j; #10;
    end
// --> end
//$display("Running testbench");
end
```

2、ALU

```
initial
begin
// code that executes only once
// insert code here --> begin
  for(i = 0; i <= 7; i = i + 1)
    for(j = 0; j <= 15; j = j + 1)
      for(k = 0; k <= 15; k = k + 1)
      begin
        sel = i; a = j; b = k; #10;
      end
// --> end
//$display("Running testbench");
end
```

七、实验结果

仿真结果和实际硬件结果一致，也和真值表完全相符

八、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?

判断 A1、B1 和运算结果的符号位

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？

运算结果一样，进位位和溢出位不一样

进位位：当 B == 4‘b0000 时，方法一{Carry, Result} = A + 4’b1111 + 1 = A + 5’b10000，方法二{Carry, Result} = A + 4’b0000。故 Carry 位相反。

溢出位：当 A 为负数，B == 4‘b1000 时，方法一 Overflow = 0，方法二 Overflow = 1

在第一步产生了差别

方法一正确

3、在判断输出结果是否为零的时候也有两种判断方式，一种是用 if 语句， 将 Result 和“0”相比较，这样在硬件上会产生一个比较器。还可以使用如下语句：

assign zero = ~ (| Result) ;

“| Result”操作称为一元约简运算，这个运算在硬件上几个逻辑门就可以实现了，请查阅 Verilog 相关语法资料，了解此运算的操作过程。

在该公式中，“|”是缩减运算符。

缩减运算符是单目运算符，也有与或非运算。其与或非运算规则类似于位运算符的与或非运算规则，但其运算过程不同。位运算是对操作数的相应位进行与或非运算，操作数是几位数则运算结果也是几位数。而缩减运算则不同，缩减运算是对单个操作数进行与或非递推运算，最后的运算结果是一位的二进制数。缩减运算的具体运算过程是这样的：第一步先将操作数的第一位与第二位进行与或非运算，第二步将运算结果与第三位进行与或非运算，依次类推，直到最后一位。例如：

```
reg[3:0]B;
reg C;
C=&B;
```

相当于：C=((B[0]&B[1])&B[2])&B[3] ;

所以，| Result =(((Result[0] | Result[1]) | Result[2]) | Result[3] ;

八、实验中遇到的问题及解决办法

1、对补码运算的相关概念仍混淆不清，花了几天时间都没弄明白题目中的方法的原理，曾一度以为两种方法都是错的。

解决办法：将以前讲补码运算的课件重新复习了几遍，并找了一些大佬多次询问，才在最后拎清了相关概念。

2、数据输入的情况太多，难以通过仿真图判断结果的正确性。

解决办法：不需要每种情况都验证，只需要进行归类加以讨论，去验证个别代表性的数据和一些边界特殊值，即可判断其正确性。

九、实验得到的启示

1、在没有弄清楚实验中的相关概念前，不要着急着盲目做实验，要先搞明白相关原理和知识后，再进行实验，不然如果方向错误，所有实验可能都白做了。

2、对于一些新鲜的概念或方法，可以在网上查阅相关资料，可以辅助自己理解

十、意见和建议

第二题的输入没说清是原码还是补码，我按着原码做了两天没做出来，还把自己绕晕了，最后才知道输入的都是补码。建议在题目中像第一题一样补充说明一下。