

嵌入式系统结构与设计课程实验

实验报告 3

Lab 7、Lab 8、Lab 8 及 Lab 10 四次实验的实验报告（Version 2）

邱迪聪 / 11331262

2013/10/23

目录

一、实验目的	1
二、Lab7 (优先编码器的设计与实现)	1
1、使用 Verilog 语句表达优先编码器的逻辑	1
2、优先编码器的模拟运行	2
3、综合、实现及程序生成	3
4、真机测试	4
三、Lab8 (4 位 Bin-BCD 码转换器的设计与实现)	5
1、使用 Verilog 语句表达 BCD 码转换器的逻辑	5
2、4 位 Bin-BCD 码转换器的模拟运行	6
3、综合、实现及程序生成	7
4、真机测试	8
四、Lab9 (4 位加法器的设计与实现)	10
1、使用 Verilog 语言设计 4 位加法器	10
2、两模块的模拟运行	10
3、综合、实现及程序生成	11
4、真机测试	13
五、Lab9 (超前进位 4 位加法器的设计与实现)	15
1、使用 Verilog 语言设计超前进位 4 位加法器	15
2、两模块的模拟运行	15
3、综合、实现及程序生成	16
4、真机测试	16
六、Lab10 (4 位移位器的设计与实现)	17
1、使用 Verilog 语句表达 4 位移位器的逻辑	17
2、4 位移位器的模拟运行	17
3、综合、实现与程序生成	19
4、真机测试	20
七、实验感想	22

一、实验目的

- 熟悉使用 ISE 软件设计并仿真；
- 学会程序下载；
- 更深入地学习与使用 Verilog 语言来设计硬件电路。

二、Lab7 (优先编码器的设计与实现)

1、使用 Verilog 语句表达优先编码器的逻辑

实验题目已经给出了优先编码器的真值表，如下：

x0	x1	x2	x3	x4	x5	x6	x7	y2	y1	y0
1	0	0	0	0	0	0	0	0	0	0
x	1	0	0	0	0	0	0	0	0	1
x	x	1	0	0	0	0	0	0	1	0
x	x	x	1	0	0	0	0	0	1	1
x	x	x	x	1	0	0	0	1	0	0
x	x	x	x	x	1	0	0	1	0	1
x	x	x	x	x	x	1	0	1	1	0
x	x	x	x	x	x	x	1	1	1	1

我们可以直接使用这个真值表，通过组合逻辑语句来描述整个优先编码器的逻辑。可是这样的做法非常耗费精力，而且浪费时间。经过简单思考，其实可以观察出，一种优先级关系，正如要实现的功能一样。当 $x_4 - x_7$ 为 1 的时候， y_2 必然为 1；当 $x_6 - x_7$ 为 1 的时候， y_1 必然为 1 等。

故我把该逻辑设计为如下 Verilog 语句：

```
assign y[2] = x[7] | x[6] | x[5] | x[4];  
  
assign y[1] = x[7] | x[6]  
           | (~x[5] & ~x[4] & (x[3] | x[2]));  
  
assign y[0] = x[7]  
           | (~x[6] & x[5])  
           | (~x[6] & ~x[4] & x[3])  
           | (~x[6] & ~x[4] & ~x[2] & x[1]);
```

由该程序段里面也可以很容易地看出优先级的顺序关系。

可是这段代码在综合的时候是会出现警告的，因为里面根本就没有 `x[0]` 出现必要，故最后无论如何也会把它约掉，所以 ISE 就会报出有变量没有被使用的警告。

可是在仔细观察实验要求，在提示中我发现了还有一个输出内容，就是 `Valid` 变量。这个变量用于表示输出是否合法，当且仅当输入全部为 0 的时候是非法的。为了显示的美观和贴近实际生活，我把 `Valid` 作为一个低电平表示合法的变量来设计。这样做的好处是，当把它接到电路的 LED 中时，它可以作为一个警告灯使用，当输入非法的时候，对应的 LED 灯会亮起；而输入正常的情况下，这个警告灯会熄灭，以不影响对正常结果的阅读。

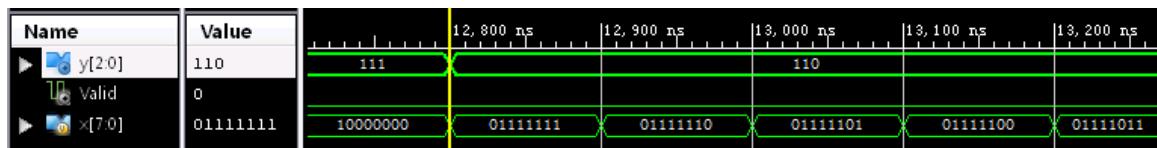
2、优先编码器的模拟运行

本实验中，我使用了全范围模拟，能够对每个可能的输入都模拟运行一次。由于模拟周期非常长，有 256 个变化情况，就不一一列出了，只截取部分关键的图片以作说明。

以下为 `x[7]` 优先级被触发，可见只要 `x[7]` 为 1，后面怎么变都使输出为 111，该过程持续了 128 个变化周期。

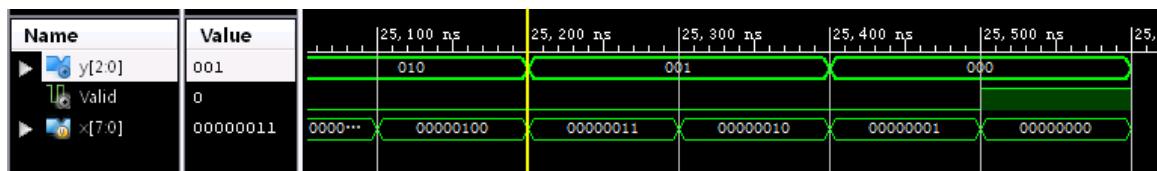


以下为 `x[6]` 优先级被触发的情况，同上面的情况类似，而该周期持续了 64 个变化周期。



中间几个的情况类似，则可跳过，并观察最后几个周期的一些差别。

以下为最后的 5 个输入变化周期。从 11111111 到最后倒数第二个周期，都有输出值相同的时间呈 2 次方指数下降的性质，且这所有 255 个情况中，`Valid` 的值都为 0，符合预期设想。而最后一个变化周期中，可以见到所有的输入都为 0，输入不合法，故 `Valid` 输出 1 表示警告状态，也符合预期设定。



模拟结果全部与设想符合，测试通过。

3、综合、实现及程序生成

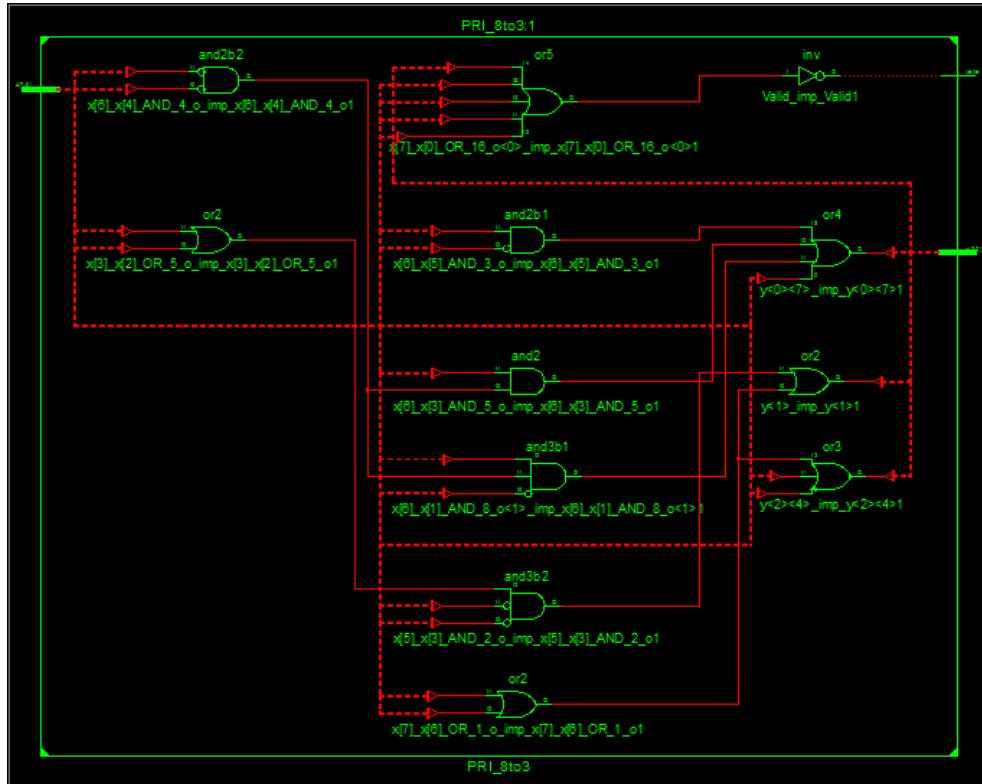
编写管脚约束，如下：

```
// Inputs
NET "x[7]" LOC = "T5"; // left most button
NET "x[6]" LOC = "V8";
NET "x[5]" LOC = "U8";
NET "x[4]" LOC = "N8";
NET "x[3]" LOC = "M8";
NET "x[2]" LOC = "V9";
NET "x[1]" LOC = "T9";
NET "x[0]" LOC = "T10"; // right most button

// Outputs
NET "y[2]" LOC = "T11"; // left most LED
NET "y[1]" LOC = "R11";
NET "y[0]" LOC = "N11";
NET "Valid" LOC = "U16"; // right most LED
```

该管脚约束将开发板上的八个拨动按钮左右输入，从左到右分别从 MSB 到 LSB；而左数起三个 LED 灯作为优先级触发状态输出，最右边的一个 LED 灯作为警报，其余保留不用。

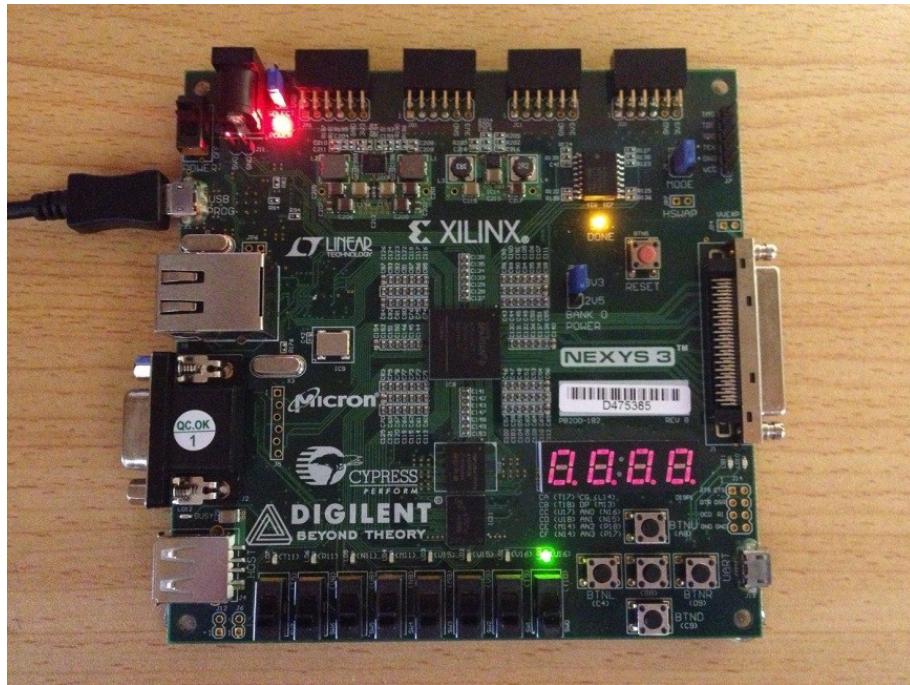
在经过综合后得到如下 RTL 图：



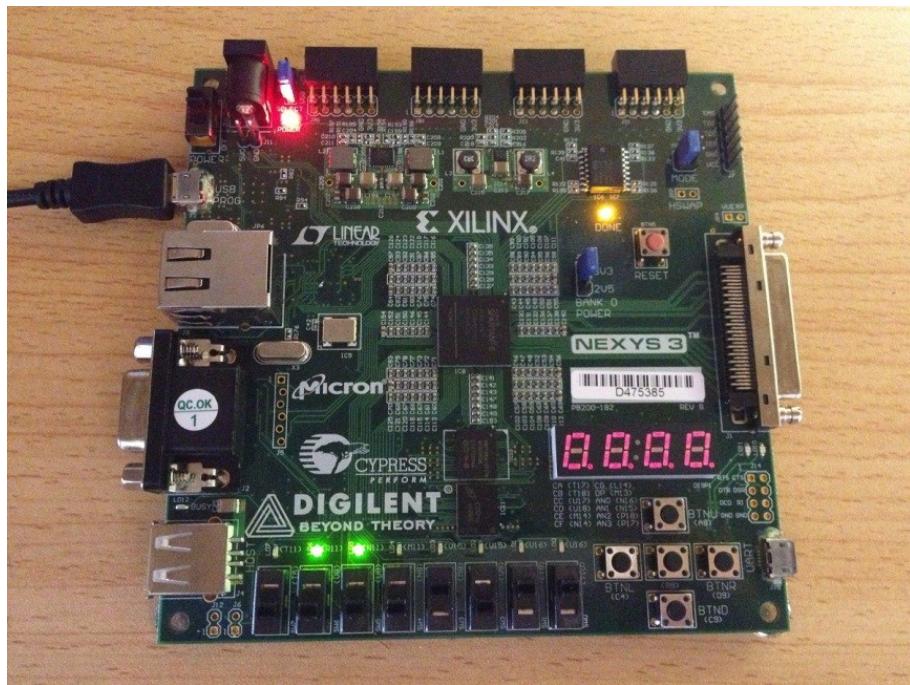
之后进行实现与生成程序步骤均通过。

4、真机测试

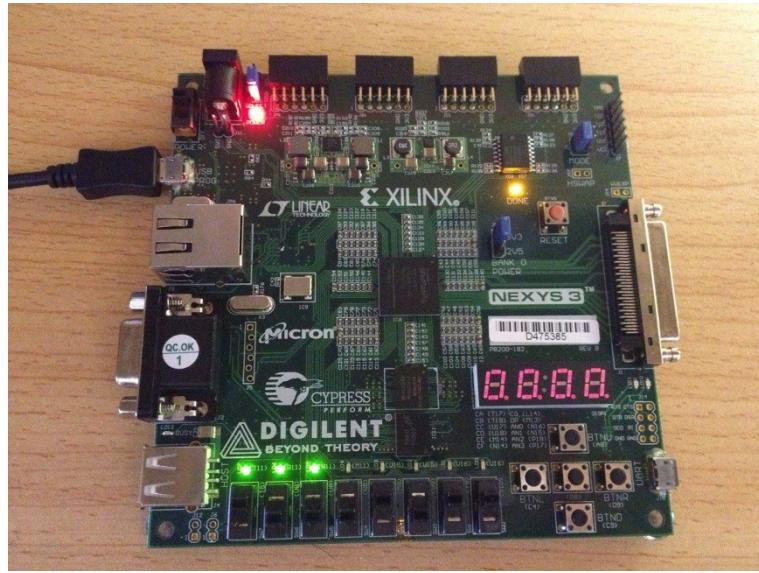
输入为 00000000 时，警告灯亮起：



输入为 00001011 时，激活优先级 3 (011)：



输入为 10001011 时，激活优先级 7 (111)：



在实际测试中，还对其他情况进行了相关测试，发现只要某个具有更高优先级的按钮被设置为 1，则低优先级的按钮无论如何拨动，LED 灯显示的都是对应的最高优先级。

故真机测试通过。

三、Lab8 (4 位 Bin-BCD 码转换器的设计与实现)

1、使用 Verilog 语句表达 BCD 码转换器的逻辑

根据要求实现的 BDC 码转换器的电路图及真值表，如下：

二进制				二进制编码十进制数 (BCD)						
HEX	b3	b2	b1	b0	p4	p3	p2	p1	p0	BCD
0	0	0	0	0	0	0	0	0	0	00
1	0	0	0	1	0	0	0	0	1	01
2	0	0	1	0	0	0	0	1	0	02
3	0	0	1	1	0	0	0	1	1	03
4	0	1	0	0	0	0	1	0	0	04
5	0	1	0	1	0	0	1	0	1	05
6	0	1	1	0	0	0	1	1	0	06
7	0	1	1	1	0	0	1	1	1	07
8	1	0	0	0	0	1	0	0	0	08
9	1	0	0	1	0	1	0	0	1	09
A	1	0	1	0	1	0	0	0	0	10
B	1	0	1	1	1	0	0	0	1	11
C	1	1	0	0	1	0	0	1	0	12
D	1	1	0	1	1	0	0	1	1	13
E	1	1	1	0	1	0	1	0	0	14
F	1	1	1	1	1	0	1	0	1	15

从图的真值表可见，由 4 位的二进制码转换为不完全的两位 BCD 码是非常复杂的过程。前面 0 – 9 的转换是比较轻松的，即在输入没有超过 10 的时候，可以直接把输入输出到结果中；可是一旦加入了后面的 10 – 15 这 6 个情况，如果使用普通的电路逻辑来设计，复杂程度与难度就会远超原来 10 以内的情况，而且 10 以内的那种设计方案也需要做非常大的更改。

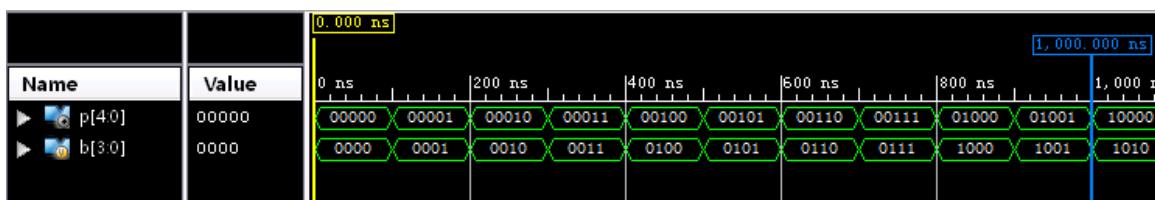
为了设计方便，而且为了更好学习 Verilog 语言，我决定尝试使用判断语句。可是在我查找相关资料后发现，判断语句的大部分情况虽然能够用组合逻辑实现，可是它好像并不能直接应用于组合逻辑语句的编写。因此我这次使用到的寄存器 reg 作为中间变量。

具体的逻辑是，在一个对任意内部内容敏感的 always 语句段里面，每次都先把输入 b 赋值给一个中间寄存器 rb，然后对 rb 进行判断。如果 rb 小于 10，则把 rb 内容直接赋值给另外一个中间寄存器 rp 的低 4 位，而 rp 的高 1 为置 0；否则先把 rb 减去 10，然后再把其赋值给 rp 的低 4 位，rp 的高 1 位置 1。并在 always 语句段外面，将 rp 用 assign 语句连续性赋值给输出 p。（具体的程序编写见提交的代码）

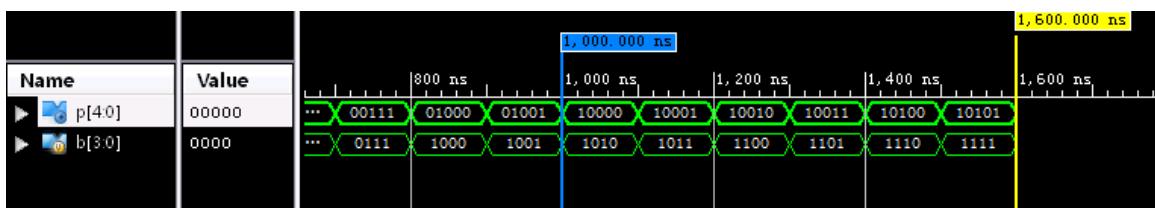
2、4 位 Bin-BCD 码转换器的模拟运行

本次实验，我进行了完整性测试，对每个输入的可能性都进行了一次模拟测试。

下图显示为在输入没有超过 10，即 0000 – 1001 这 10 中情况。在设计中，当输入没有超过 10 的时候，只需要用到 BCD 码的低 4 位，即只有个位的 BCD 码被使用，而高位的 BCD 码应该被置 0，以表示在十进制中，只有个位的情况。而在模拟结果中，可见输出 p 的最高位一直为 0，低 4 位与输入完全相同，符合程序设计预期。



下图显示为超过或等于 10 的情况，即 1010 – 1111 这 6 种情况。可见在 BCD 码中高位一直为 1，表示十进制中的十位一直为 1，而个位则依次由 0 变化为 5，符合 10 – 15 的输入变化，符合程序设计预期。



有以上模拟结果看出，模拟测试符合设计预期，通过测试。

3、综合、实现及程序生成

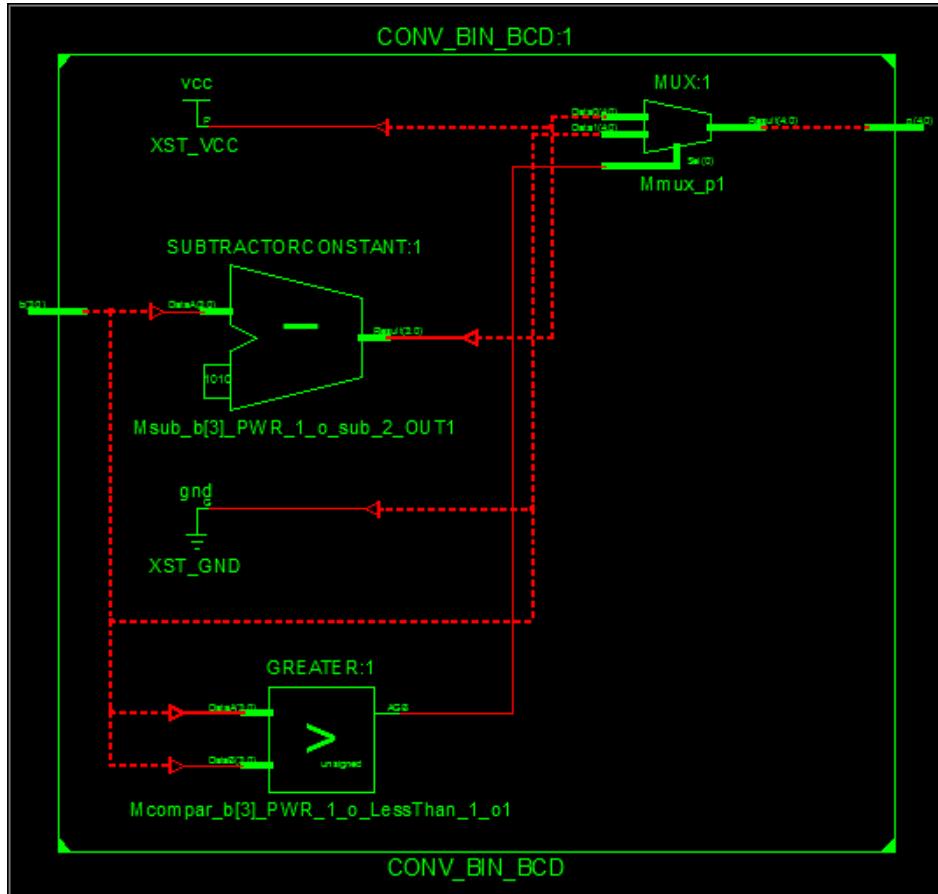
管脚约束如下：

```
// Inputs
NET "b[3]" LOC = "T5"; // left most button
NET "b[2]" LOC = "V8";
NET "b[1]" LOC = "U8";
NET "b[0]" LOC = "N8";

// Outputs
NET "p[4]" LOC = "T11"; // left most LED
NET "p[3]" LOC = "R11";
NET "p[2]" LOC = "N11";
NET "p[1]" LOC = "M11";
NET "p[0]" LOC = "V15";
```

以上管脚约束，将最左边的四个拨动按钮作为输入，由左到右为 MSB 到 LSB。而使用了五个 LED 灯作为输出，最左边的 LED 灯作为 BCD 码的高 1 位，另外四个作为 BCD 码的低 4 位。

以下为综合后生成的 RTL 图：

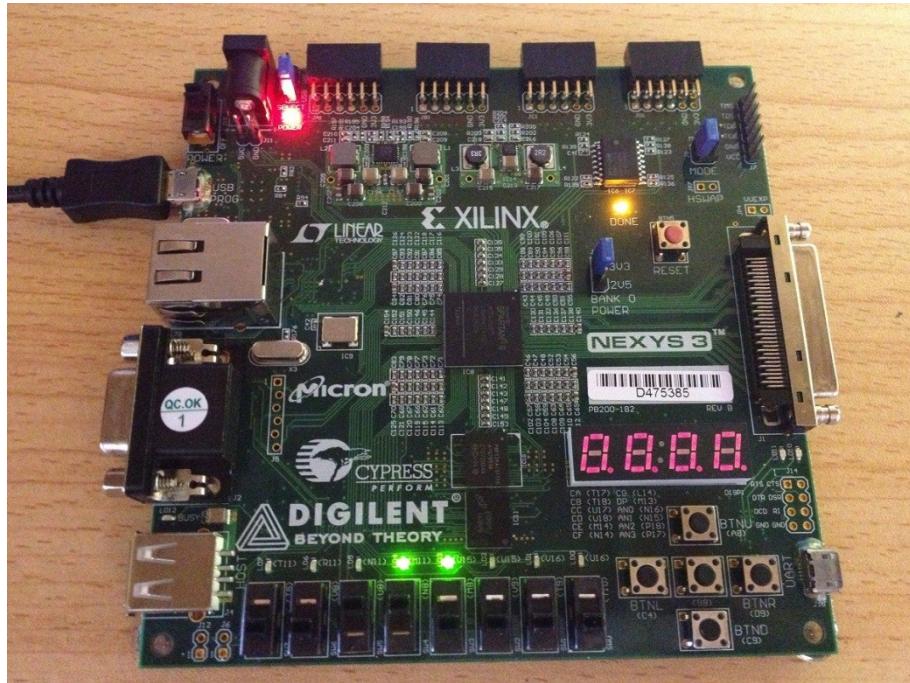


之后进行实现与程序生成均成功。

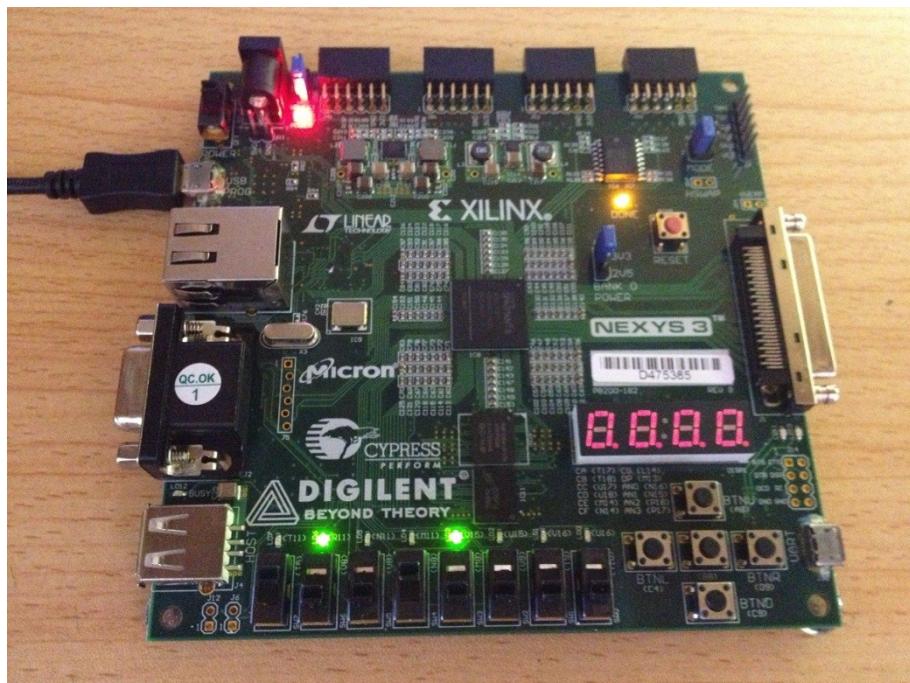
4、真机测试

将.bit文件上传到开发板上，并进行真机测试。由于情况比较多，故只挑选几个情况作为展示使用。

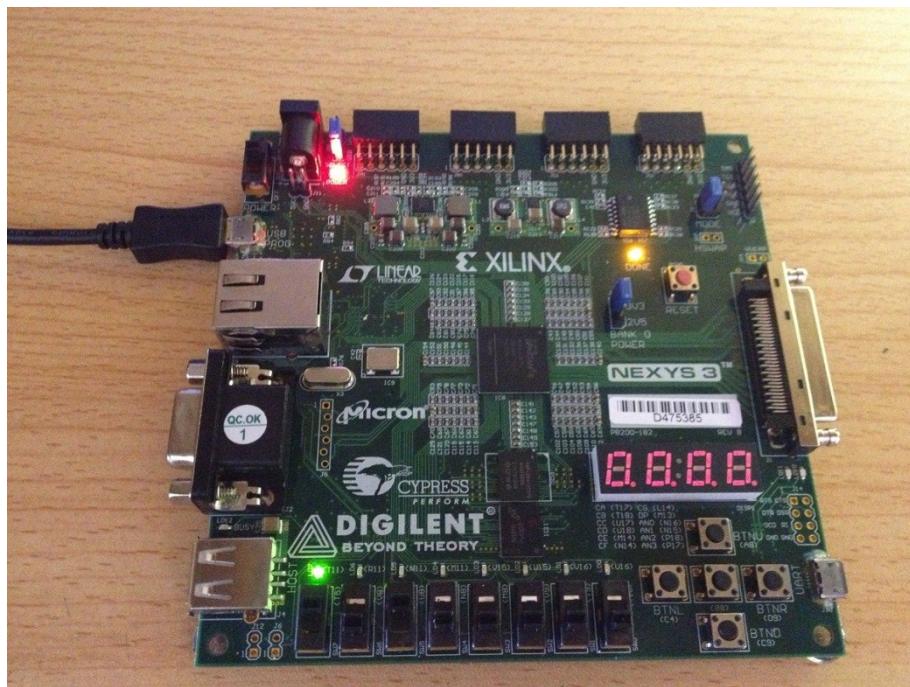
输入为 0011，输出为 0 0011 (03):



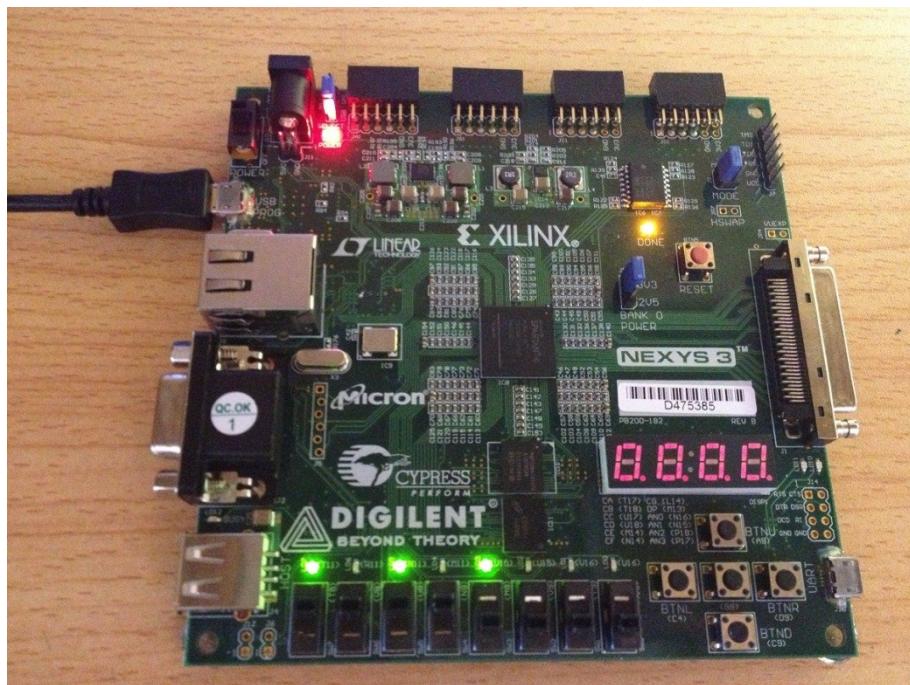
输入为 1001，输出为 0 1001 (9):



输入为 1010，输出为 1 0000 (10):



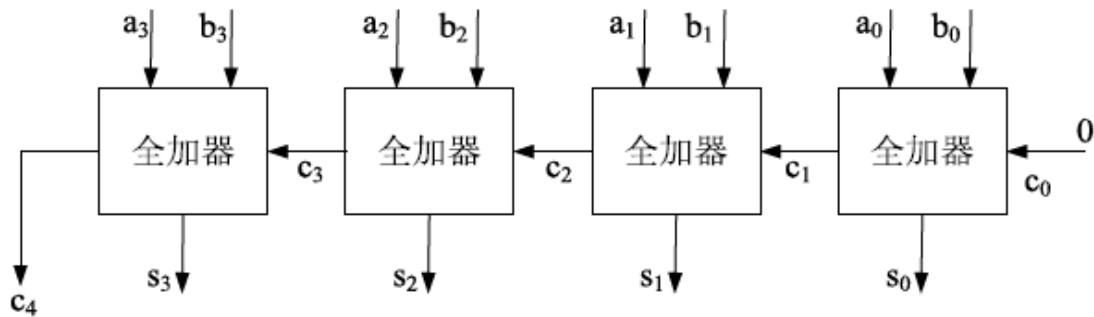
输入为 1111，输出为 1 0101 (15):



四、Lab9 (4位加法器的设计与实现)

1、使用 Verilog 语言设计 4 位加法器

4 位加法器的逻辑图如下：



由上图可见，4位加法器由4个全加器组成，故第一步先要设计出一个全加器，然后在进行对他们的组合。在这个实验中，我第一次在一个项目中应用到了两个非测试模块。

其中一个模块用于设计全加器，其核心语句如下：

```
module FullAdder(input wire A, B, Cin, output wire S, Cout);
    assign S = (A^B)^Cin;
    assign Cout = A&B | B&Cin | A&Cin;
endmodule
```

在全加器中，有三个输入，两个输出，其逻辑非常简单。

然后就是要设计一个4位加法器模块了。4位加法器中间需要用到3条连接线，作为进位的输入与输出，如下：

```
wire C1, C2, C3;
```

而在调用全加器模块的时候，我了解到，需要进行实例化，然后语句语法如下：

```
FullAdder fa1(.A(A[0]), .B(B[0]), .Cin(Cin), .S(S[0]), .Cout(C1));
```

以上是其中一个全加器的调用，剩下的3个就不一一列出了。经过综合，发现无错通过，故本次编写成功通过语法检测。

2、两模块的模拟运行

本次模拟运行有两个部分，一个是模拟全加器的工作情况，另外一部分是模拟4位加法器的工作情况。

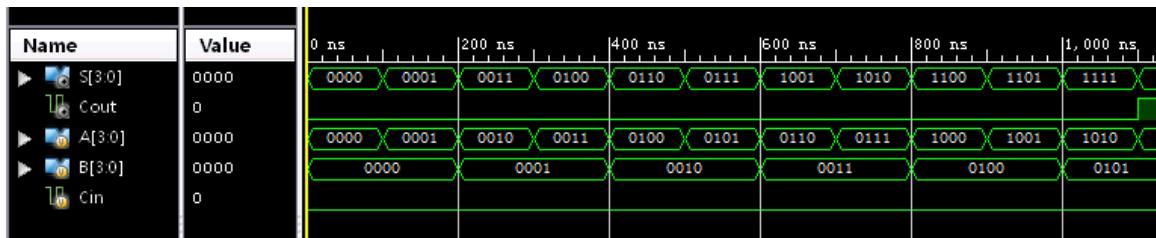
以下是全加器的模拟运行结果：



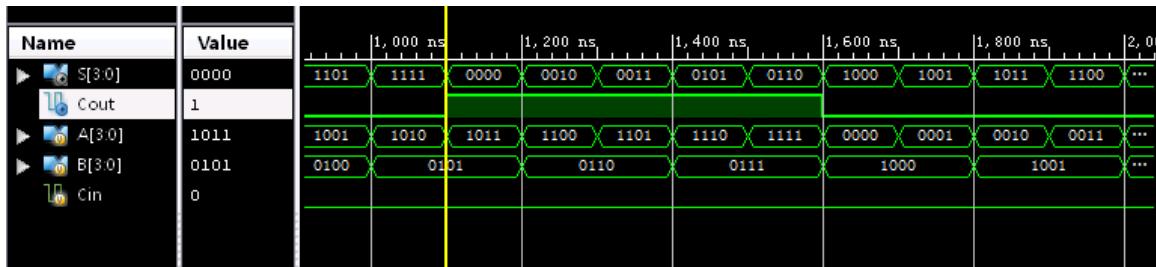
结果符合全加器的设计要求。

而 4 位加法器的输入就有 9 个，但是还是进行了完整性测试，而结果太多，就不完全展示出来了，仅挑选部分比较关键的部分进行展示。而为了展现有更多的随机性，故以下模拟结果非完整性测试结果，而是有比较大随机性的结果，以方便从中挑选出比较有代表性的关键部分。

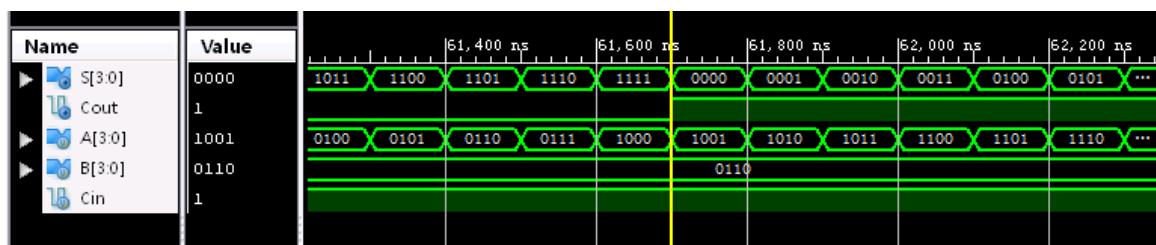
如下为部分没有进位，且 Cin 为 0 的情况：



以下为 Cin 为 0，但有进位的情况：



以下为有 Cin 的情况：



模拟测试结果均符合预期，通过测试。

3、综合、实现及程序生成

管脚约束如下：

```

// Input: A
NET "A[3]" LOC = "T5"; // left most button
NET "A[2]" LOC = "V8";
NET "A[1]" LOC = "U8";
NET "A[0]" LOC = "N8";

// Input: B
NET "B[3]" LOC = "M8";
NET "B[2]" LOC = "V9";
NET "B[1]" LOC = "T9";
NET "B[0]" LOC = "T10"; // right most button

// Input: Carry in
NET "Cin" LOC = "B8"; // central button

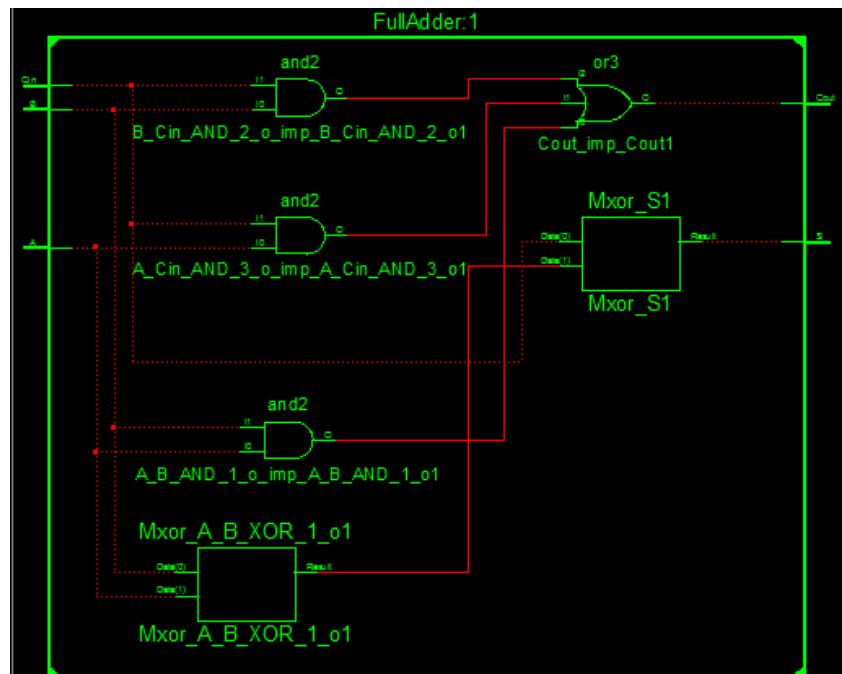
// Output: Carry out
NET "Cout" LOC = "T11"; // left most LED

// Output: S
NET "S[3]" LOC = "N11";
NET "S[2]" LOC = "M11";
NET "S[1]" LOC = "V15";
NET "S[0]" LOC = "U15";

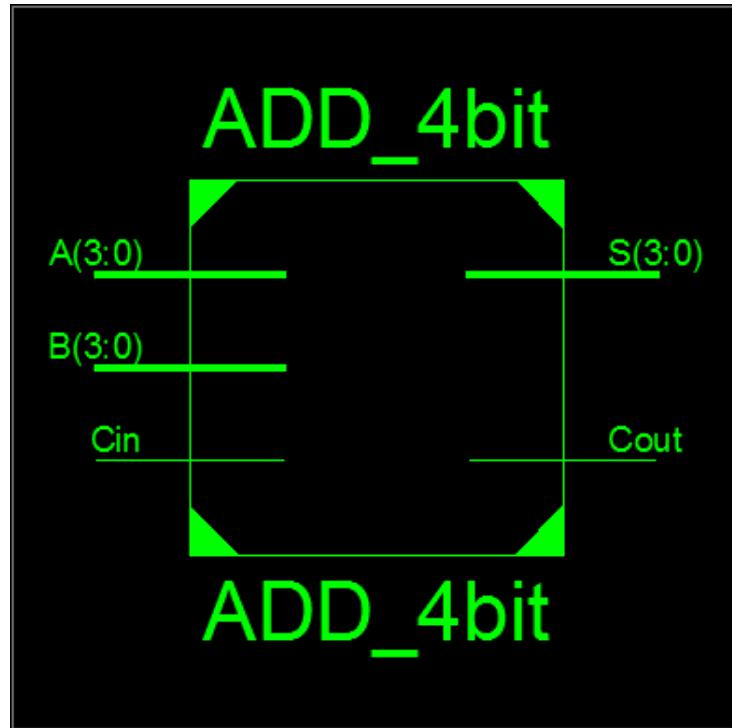
```

以上管脚约束将 8 个拨动按钮的左 4 位作为 A 输入，右 4 位作为 B 输入，而右边的十字按钮中间的按钮作为低位进位输入。最左边的 LED 灯作为进位输出，而隔开一个 LED 灯后，依次 4 个作为和输出。

以下为全加器综合后生成的 RTL 图：



以下为 4 位加法器综合后生成的 RTL 图:

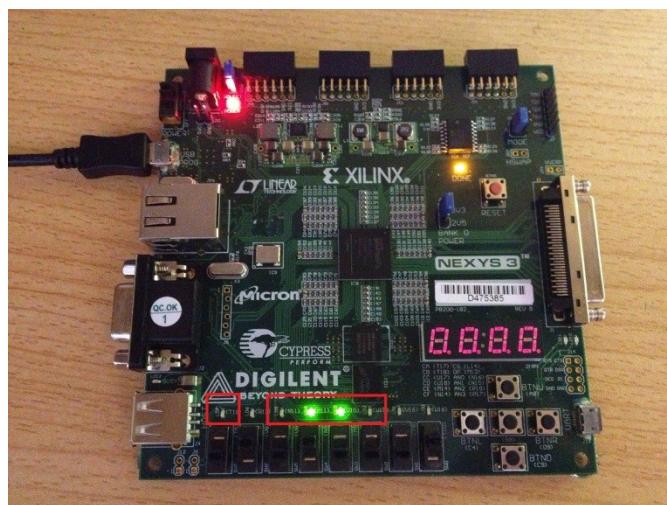


在经过实现和程序生成后，就可以得到相应的.bit 文件了。

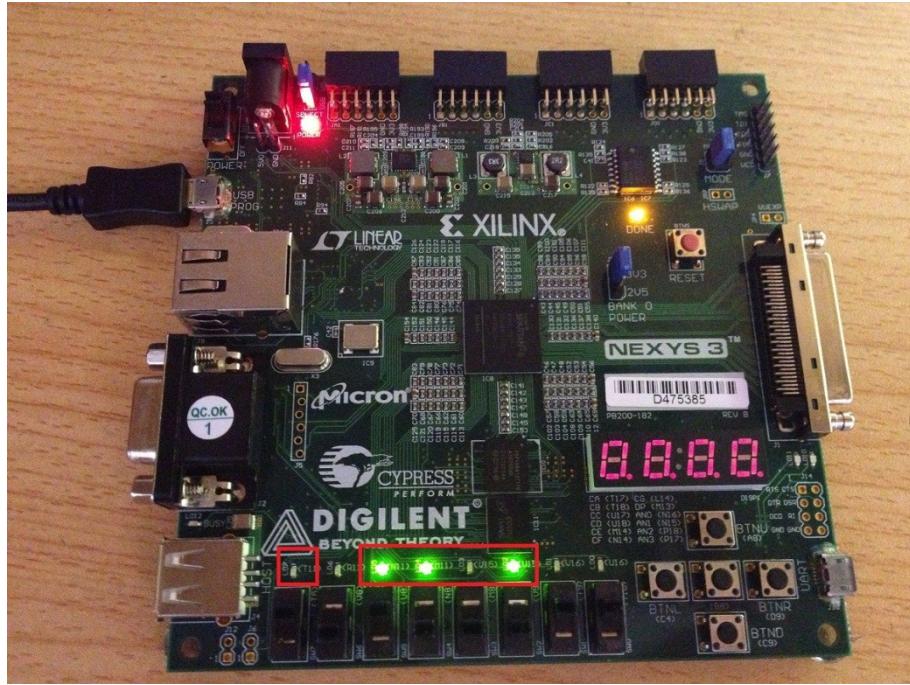
生成过程成功。

4、真机测试

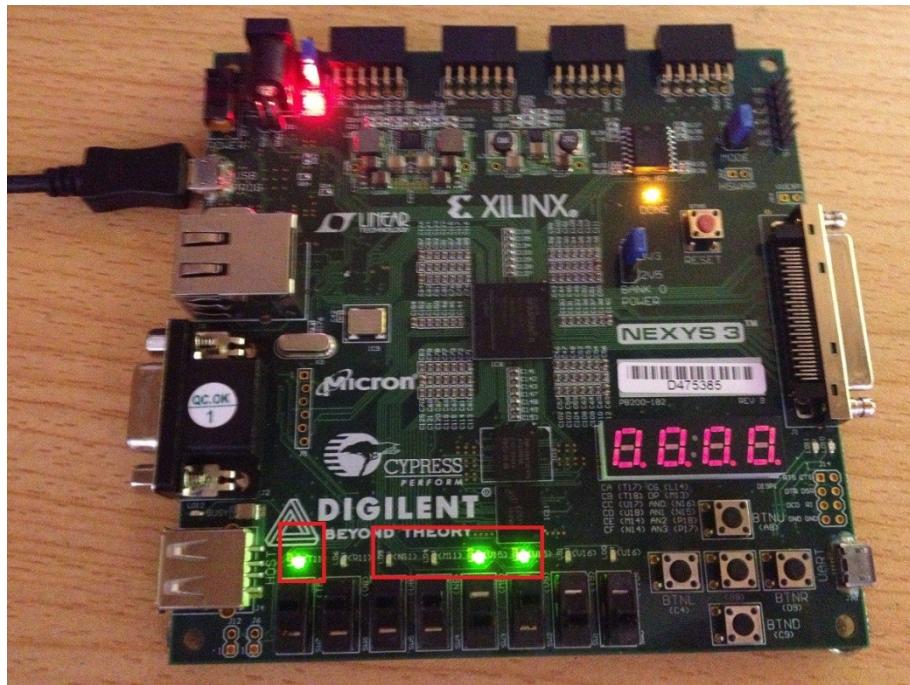
5+1=6:



10+3=13:



15+4=19 (3 进 1):



五、Lab9 (超前进位 4 位加法器的设计与实现)

1、使用 Verilog 语言设计超前进位 4 位加法器

与一般的 4 位加法器一样，需要先定义一个全加器模块，然后再将全加器模块进行相应的组合。

在全加器模块中，不同的是没有了 Cout 输出，因为进位都使用另外的逻辑进行实现了。其中一个模块用于设计全加器，其核心语句如下：

```
module FullAdder(input wire A, B, Cin, output wire S);
    assign S = (A^B)^Cin;
endmodule
```

在全加器中，有三个输入，一个输出，其逻辑非常简单。

然后就是要设计一个 4 位加法器模块了。按照超前进位加法器的设计思路，先把对应的设计公式写出来：

$$C_{in} = C_{in}$$

$$C_1 = A_0 B_0 + C_{in}(A_0 + B_0)$$

$$C_2 = A_1 B_1 + C_1(A_1 + B_1)$$

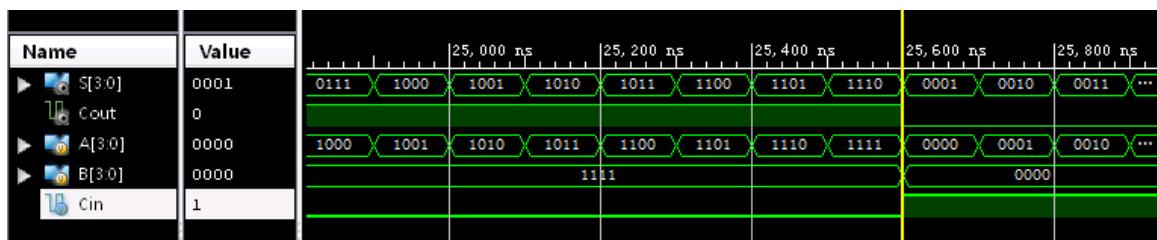
$$C_3 = A_2 B_2 + C_2(A_2 + B_2)$$

$$C_{out} = A_3 B_3 + C_3(A_3 + B_3)$$

按照如上公式，在主模块中使用 `assign` 语句将对应的超前进位作为全加器的进位输入，或者作为 4 位加法器的高位进位输出即可。在写成相应语句之前，需要将以上公式迭代至最基本的输入。(具体编写见提交的程序)

2、两模块的模拟运行

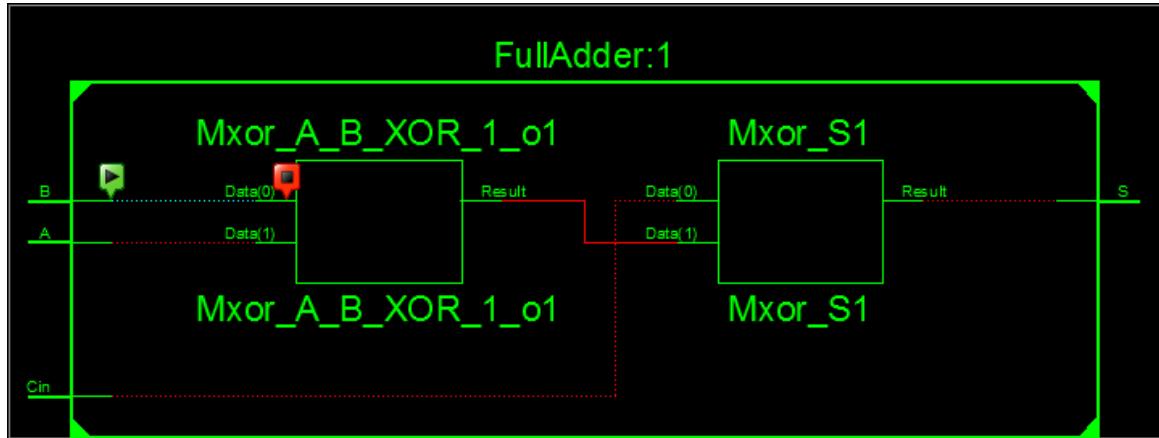
超前进位 4 位加法器的模拟测试结果和一般 4 位加法器的模拟运行结果相同，均符合 4 位加法器的设计预期，通过模拟测试。



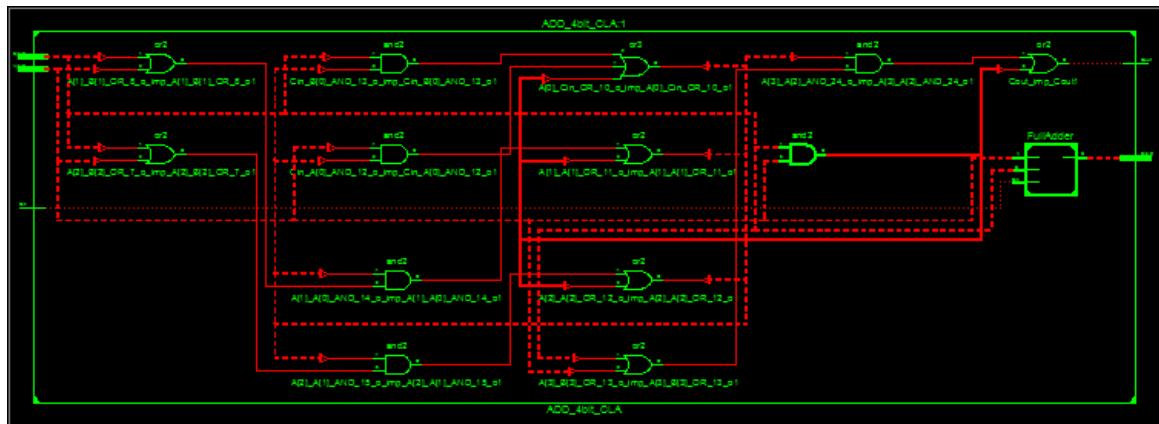
3、综合、实现及程序生成

管脚约束与一般 4 位加法器的管脚约束相同。

在综合后，得到如下全加器的 RTL 图：



而 4 位加法器的 RTL 图如下：



之后再进行实现与程序生成，并得到对应的 .bit 文件。

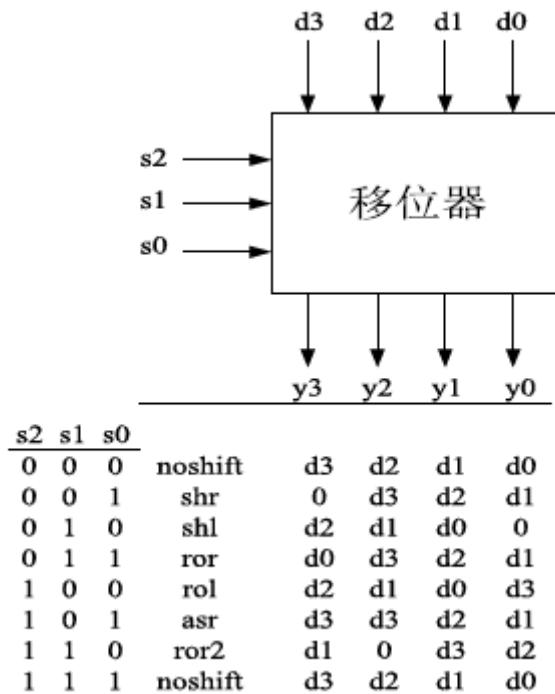
4、真机测试

超前进位 4 位加法器的真机测试结果与一般 4 位加法器的测试结果相同，在此处就不重复列出了。在真机测试中，程序逻辑得到验证，符合设计预期，故通过测试。

六、Lab10 (4位移位器的设计与实现)

1、使用 Verilog 语句表达 4 位移位器的逻辑

4 位移位器的逻辑图与真值表如下：



由于控制位并无太大规律，所以该移位器只能通过每个情况都进行一个特化处理的方式来设计了。而在该设计过程中，由于用到了判断语句，故在程序中有相应的寄存器作为中间变量。

在自动敏感的 `always` 语句中，每次运行都把输入 `d` 和 `s` 分别赋值到对应的寄存器 `rd` 和 `rs` 中，然后进行 8 判断，看看是输入哪种控制要求，再把对应的结果输出到 `ry` 寄存器中。在 `always` 语句段外面，用 `assign` 语句将 `ry` 的结果值绑定到 `y` 作为输出即可。（具体程序见提交电子文件的内容）

2、4 位移位器的模拟运行

本次实验对输入进行了完整性测试，而由于情况太多，这里只显示部分关键内容。

S=001, SHR:

Name	Value	1,600 ns	1,800 ns	2,000 ns	2,200 ns	2,400 ns
► y[3:0]	0000	1110	1111	0000	0001	0010
► d[3:0]	0000	1110	1111	0000	0001	0100
► s[2:0]	001	000	000	001	011	1000

S=010, SHL:

Name	Value	3,200 ns	3,400 ns	3,600 ns	3,800 ns	4,000 ns
► y[3:0]	0000	0111	0000	0010	0100	0110
► d[3:0]	0000	1110	1111	0000	0001	0010
► s[2:0]	010	001			010	

S=011, ROR:

Name	Value	4,800 ns	5,000 ns	5,200 ns	5,400 ns	5,600 ns
► y[3:0]	0000	1100	1110	0000	1000	0001
► d[3:0]	0000	1110	1111	0000	0001	0010
► s[2:0]	011	010			011	

S=100, ROL:

Name	Value	5,400 ns	5,600 ns	5,800 ns	7,000 ns	7,200 ns
► y[3:0]	0000	0111	1111	0000	0010	0100
► d[3:0]	0000	1110	1111	0000	0001	0010
► s[2:0]	100	011			100	

S=101, ASR:

Name	Value	8,000 ns	8,200 ns	8,400 ns	8,600 ns	8,800 ns
► y[3:0]	0000	1101	1111	0000	0001	0010
► d[3:0]	0000	1110	1111	0000	0001	0010
► s[2:0]	101	100			101	

S=110, ROR2:

Name	Value	9,600 ns	9,800 ns	10,000 ns	10,200 ns	10,400 ns
► y[3:0]	0000	1111	0000	0100	1000	1100
► d[3:0]	0000	1110	1111	0000	0001	0010
► s[2:0]	110	101			110	

其他没有显示出来的情况也符合设计预期，而另外两个没有操作的控制情况由于比较简单，就不进行显示了。

3、综合、实现与程序生成

管脚约束如下：

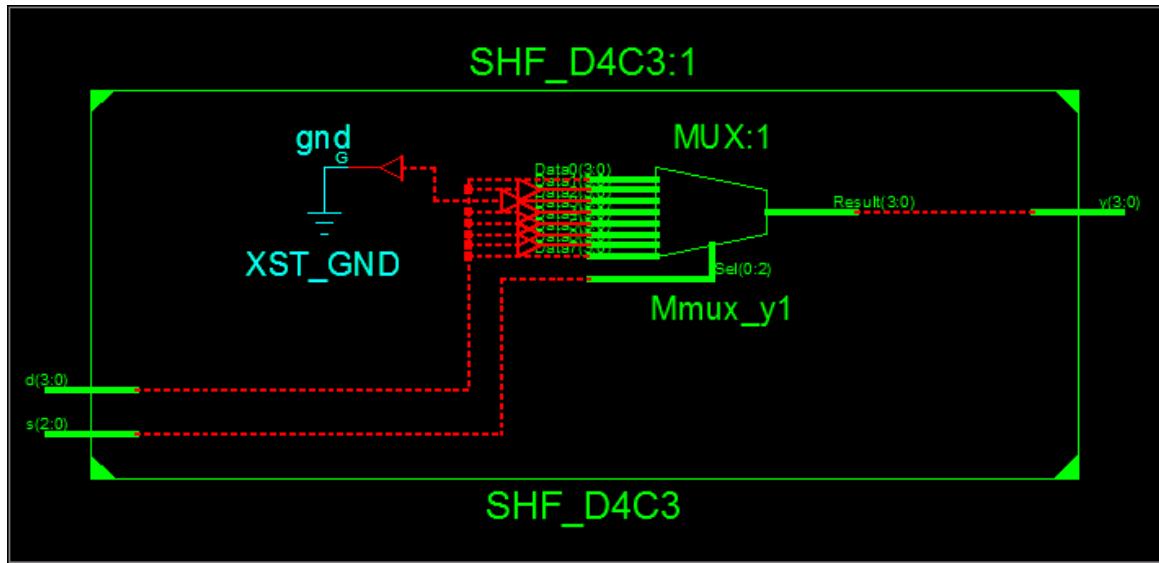
```
// Input: data bits
NET "d[3]" LOC = "T5"; // left most button
NET "d[2]" LOC = "V8";
NET "d[1]" LOC = "U8";
NET "d[0]" LOC = "N8";

// Input: control bits
NET "s[2]" LOC = "V9";
NET "s[1]" LOC = "T9";
NET "s[0]" LOC = "T10"; // right most button

// Output
NET "y[3]" LOC = "T11"; // left most LED
NET "y[2]" LOC = "R11";
NET "y[1]" LOC = "N11";
NET "y[0]" LOC = "M11";
```

该管脚约束将左 4 个拨动开关作为数据输入，右 3 个拨动开关作为控制输入，第 5 个保留不使用。左 4 个 LED 灯作为输出结果显示。

综合后生成的 RTL 图如下：

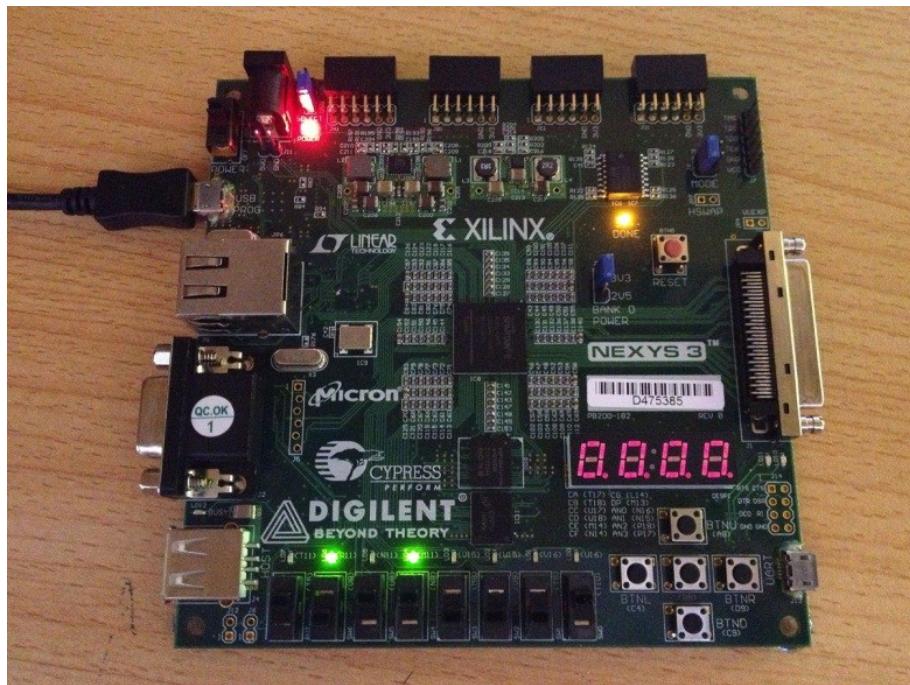


上图可见，虽然在程序中使用了较多的 if...else...语句，可是 ISE 是能够自动把它们转换成 case 语句的。在综合后，再进行实现与程序生成，即可得到对应的 .bit 文件。

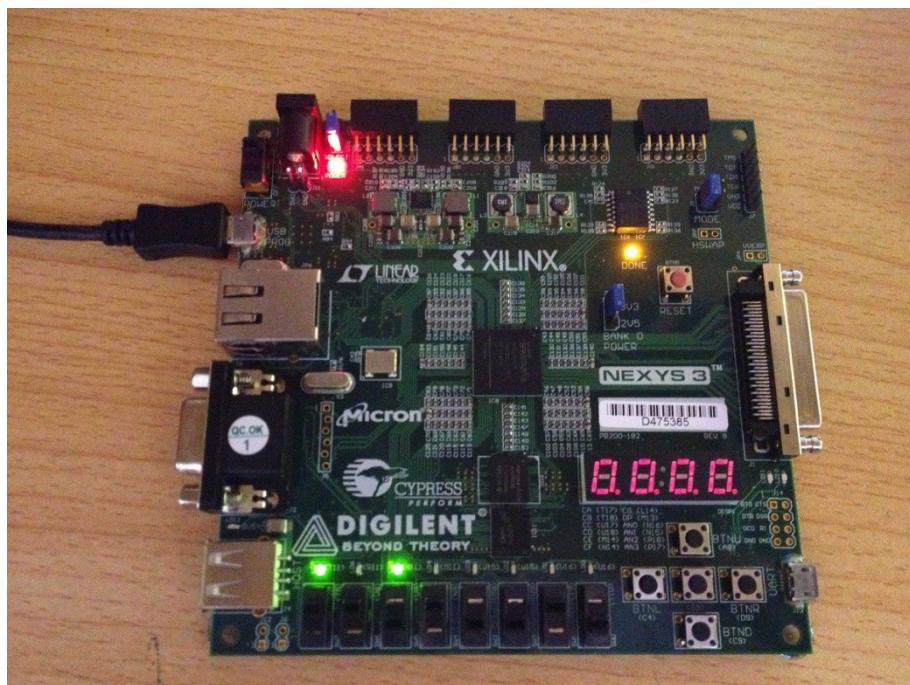
4、真机测试

由于情况比较多，故只在这里展示部分真机测试结果。

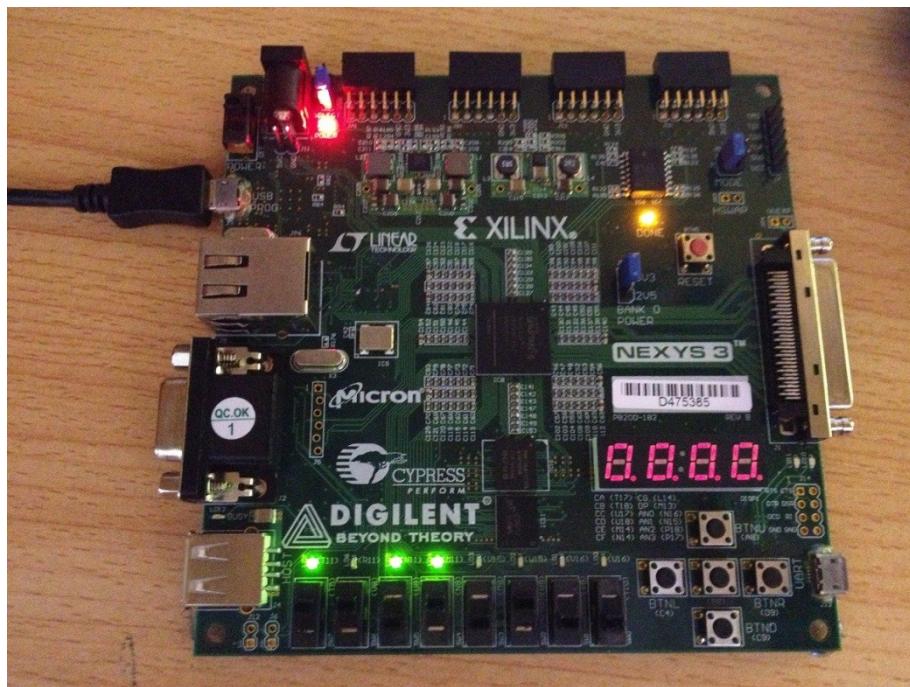
D=1011, S=001, SHR:



D=1101, S=010, SHL:



D=1101, S=100, ROL:



七、实验感想

这一次的实验，也像上次一样，一共做了 4 个，的确耗费了好多时间去做。

上次实验时已经说过了，虽然这些实验在数字电路的实验课上都做过了，可是那个是利用逻辑电路来做的，每做一个功能，要么就要有现成的芯片，要么就要通过现成的芯片去搭出这样一个功能模块，非常地不方便、折腾人。而在这个课程上我知道了居然能用语言来把这些电路逻辑描述出来，而且 Verilog 这个语言非常完善而且方便。

这次我的实验中尝试了更多的 Verilog 语言的特性。

在上一次的实验中，我只在模拟部分，为了方便进行完整性测试，而用了 `always` 语句。然后上次的实验中，其实我也想尝试一下使用判断语句了，可是发现判断语句必须要在 `always` 中使用，而且不能直接调用 `wire`，所以觉得比较冒险就没有使用了。

可是这次的实验，特别是移位器的实验，如果没有使用判断语句的话，可能还真的非常折腾人，如果说要写真值表，那就更加不是一个现代大学生应该做的事情了。所以我还是找了相关的资料，然后从小处尝试，尝试看看判断语句最基本的情况如何写。在尝试到它正确的语法之后，我再把这个语法应用到实际中，用判断语句来实现了移位器。这是这次实验中，我学到的一个比较好的内容。

经过这次高强度的实验，我更加发现 Verilog 实在强大了，可是我也发现了其中的一些不足。例如我们写判断语句的时候不能够直接使用 `wire` 来编写，必须使用 `reg` 类型。其实在理论上，很多的判断语句都是可以通过组合逻辑电路来实现的，可是我就不太明白这个语言里面为什么会有这一种规定。所以虽然这个语言非常强大，可是还是有些地方是有可改进之处的，而且在看相关的资料的时候，我也看到了，有人说这个语言会有些陷阱，所以在使用的时候还是得比较小心。

邱迪聪

2013 年 10 月 16 日

附录 1：附件列表

Lab7 PRI_8to3

\ PRI_8to3.v

\ PRI_8to3_ctr.ucf

\ PRI_8to3_test.v

Lab8 CONV_BIN_BCD

\ CONV_BIN_BCD.v

\ CONV_BIN_BCD_ctr.ucf

\ CONV_BIN_BCD_test.v

Lab9 ADD_4bit

\ ADD_4bit.v

\ ADD_4bit_ctr.ucf

\ ADD_4bit_test.v

Lab9 ADD_4bit_CLA

\ ADD_4bit_CLA.v

\ ADD_4bit_CLA_ctr.ucf

\ ADD_4bit_CLA_test.v

Lab10 SHF_D4C3

\ SHF_D4C3.v

\ SHF_D4C3_ctr.ucf

\ SHF_D4C3_test.v

其中 X.v 文件为程序模块文件，X_ctr.ucf 为管脚约束文件，X_test.v 为模拟测试代码文件。