

《数字电路与数字系统实验》实验报告

第 5 次实验： 触发器与锁存器

姓名： 张逸凯

学号： 171840708

院系： 物理 学院

邮箱： 645064582@qq.com

电话： 18051988316

实验时间： 2019 年 4 月 3 日

○. 预习部分

基本定义

锁存器和触发器是时序电路的基本构件。锁存器和触发器都是由独立的逻辑门电路和反馈电路构成的，**锁存器在时钟信号为有效电平的整个时间段，不断监测其所有的输入端，此段时间内的任何满足输出改变条件的输入，都会改变输出；触发器只有在时钟信号变化的瞬间才改变输出值。**

D 锁存器

锁存器(Latch)是一种对脉冲电平敏感的存储单元电路，它们可以在特定输入脉冲电平作用下改变状态。锁存，就是把信号暂存以维持某种电平状态。

RS 锁存器中有一个 Q 和 Q 非同时为 1 的无效状态，这是 R 和 S 同时为 1 的缘故，如果强制 R 和 S 总是相反的逻辑，就可以避免这一现象产生。如图 4-3 所示，这个电路就是 D 锁存器电路，当时钟触发信号为 0 时，输出保持不变，当时钟触发信号为 1 时，Q 输出 D 的值，即 Q 随着 D 值的改变而改变。

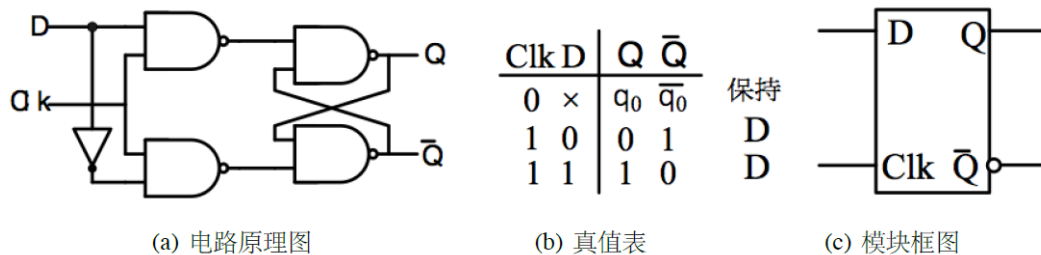


图 4-3: D 锁存器

我们可以很容易理解 D 锁存器的代码和测试代码，并实现波形仿真：

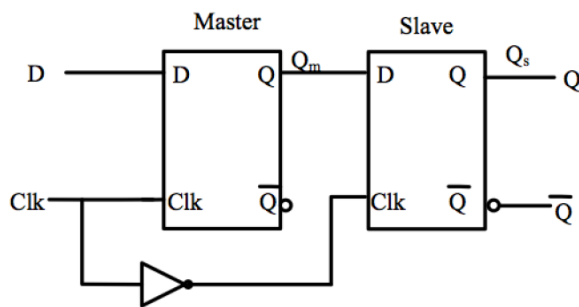
```
1 module D_latch(clk,in_d,en,out_q1);
2     input clk;
3     input in_d;
4     input en;
5     output reg out_q1;
6
7     always @ (*)
8         if ( en )
9             begin
10                 if (clk) out_q1 <= in_d;
11             end
12         else
13             out_q1 <= out_q1;
14 endmodule
```

D 触发器:

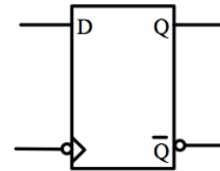
D 触发器是一个具有记忆功能的,具有两个稳定状态的信息存储器件,是构成多种时序电路的最基本逻辑单元,也是数字逻辑电路中一种重要的单元电路。触发器的次态取决于 CP 的脉冲上升沿到来之前 D 端的状态,即次态=D。因此,它具有置 0、置 1 两种功能。

我们希望锁存器只在时钟的特定时刻(如上升沿或者下降沿)触发,锁存此时刻 D 的值,这样的锁存器通常称为时钟边沿触发的触发器。

用两个锁存器可以构成触发器,如图 4-5(a)所示。图中的两个 D 锁存器,前者为主锁存器,后者为从锁存器,当 Clk 信号为 1 时,主锁存器的 Q_m 随着 D 的变化而变化,从锁存器的状态保持不变。当时钟信号变为 0 后,主锁存器的状态不再变化,从锁存器 Q_s 的状态则跟随 Q_m 状态的变化而变化,由于当 Clk=0 时, Q_m 不会发生变化,因此对于外部的观察者而言,在一个时钟周期内 Q 只在 Clk 从 1 变为 0 (即时钟的负跳变沿或下降沿)的时候发生一次变化。因此,我们也可以说输出信号 Q 是在时钟下降沿采集到的输入信号 D 的瞬间值。



(a) 电路原理图



(b) 模块框图

D 触发器的代码也非常好理解:

```
1 module D_trigger(clk,in_d,en,out_qt);
2     input clk;
3     input in_d;
4     input en;
5     output reg out_qt;
6
7     always @ (posedge clk )
8         if(en)
9             out_qt <= in_d;
10        else
11            out_qt <= out_qt;
12    endmodule
```

这里提到了测试代码中的 **\$stop** 语句:

`$stop` 语句类型:

```
$stop;  
$stop(n);
```

`$stop` 任务的作用是将 EDA 工具设置为暂停模式，在仿真环境下给出一个交互式的命令，将控制权交给用户。参数值越大，输出信息越多。默认的是一直仿真下去，如果在测试代码中加入 `$stop` 指令的话，那么编译器仿真到 `$stop` 指令时就在此处停止仿真了。

触发器设计中的非阻塞赋值语句

Verilog 语言中有两种赋值语句，之前我们使用的赋值语句采用赋值符号“=”，这种赋值被称为阻塞赋值语句；在设计触发器的时候我们使用另一种赋值语句，采用赋值符号“<=”，此赋值语句被称为非阻塞赋值语句。

阻塞赋值语句是立即赋值语句，其形式和作用都类似于其他任何过程语言（如 C 语言）的赋值语句。阻塞赋值语句在语句执行时，首先计算赋值语句右边的表达式的值，得到结果后立即将值赋给赋值语句左边的变量。

而非阻塞赋值语句却不同，非阻塞语句一般出现在 `always` 语句块中，非阻塞语句在执行时，虽然也是立即计算赋值语句右边的表达式的值，但却不将结果立即赋值给表达式左边，要等到整个 `always` 块执行完毕后，经过一个无穷小的延时才完成赋值。

老师还很贴心地给出了一个总结：

Verilog 编码指导方针

1. 当为时序逻辑电路建模时，使用“非阻塞赋值 '`<=`'”。
2. 当为锁存器（latch）建模时，使用“非阻塞赋值 '`<=`'”。
3. 当用 `always` 块为组合逻辑建模时，使用“阻塞赋值 '`=`'”。
4. 当一个 `always` 块里既有组合逻辑又有时序逻辑建模时，用“非阻塞赋值 '`<=`'”。
5. 不要在同一 `always` 块里面混合使用“阻塞赋值 '`=`'”和“非阻塞赋值 '`<=`'”。
6. 不要在两个或两个以上 `always` 块里面对同一个变量进行赋值。
7. `assign` 语句使用“阻塞赋值 '`=`'”即可。

开始实验部分

一. 实验目的

1. 复习锁存器和触发器的工作原理.
2. 复习时序电路中电路时序图的分析 and 阅读.
3. 学习如何对时序电路进行仿真, 了解Verilog语言中阻塞赋值语句和非阻塞赋值语句的区别.
4. 请在工程中设计两个触发器, 一个是带有异步清零端的D触发器, 而另一个是带有同步清零端的D触发器。
5. 用阻塞赋值语句设计一个JK触发器

二. 实验原理 (知识背景, 结合理论课总结)

➤ 基本定义:

异步清零, 是指与时钟不同步, 即清零信号有效时, 无视触发脉冲, 立即清零; 同步是时钟触发条件满足时检测清零信号是否有效, 有效则在下一个时间周期的触发条件下, 执行清零.

同步清零就是把清零信号和时钟信号与或者与非处理后输入到清零端, 同步清零可以保证状态在时钟的有效期内不会改变.

注意 PDF 里的代码提示:



大家在设计异步清零的触发器的时候, 如果触发器的清零信号 (假设命名为 `clr_n`) 是“0”有效, 那么在敏感列表中应该是检测 `clr_n` 的下降沿, 即代码应该这样:

```
1 always @(posedge clk or negedge clr_n)
2 if(!clr_n)
3     begin    ...    end
4 else ...
```

这样的代码是错误的

```
1 always @(posedge clk or posedge clr_n)
2 if(!clr_n)
3     begin    ...    end
4 else ...
```

注意实例化要求:

传递参数的方法: 传递的参数是子模块中定义的 parameter。

1. module_name #(parameter1, parameter2) inst_name(port_map);
2. module_name #(parameter_name(para_value), parameter_name(para_value))
inst_name (port map);

本实验还要求同学们学习 Verilog HDL 模块实例化的方法。

我们之前所有的实验功能均相对单一、简单，都是在一个模块内完成所有的设计。但是，在完成相对大一点的工程的时候，一个人或者一个模块不能完成所有设计，这个时候就需要利用多个单元模块来进行工程设计，每个单元模块完成一个特定的功能，最后在顶层实体模块中将所有的单元连接在一起，完成工程的整体设计。请查阅资料学习 Verilog HDL 模块实例化的方法，并在设计中应用。

在利用多个模块进行设计的过程中，我们可以将所有模块放在一个文件中，也可以为每一个模块新建一个文件。Verilog HDL 建议为每一个模块新建一个文件，文件名和模块名相同，这样在一个工程中设计好的模块可以直接在另一个工程中被实例化。不仅提高了工程的可读性也提高了模块的可重用性。

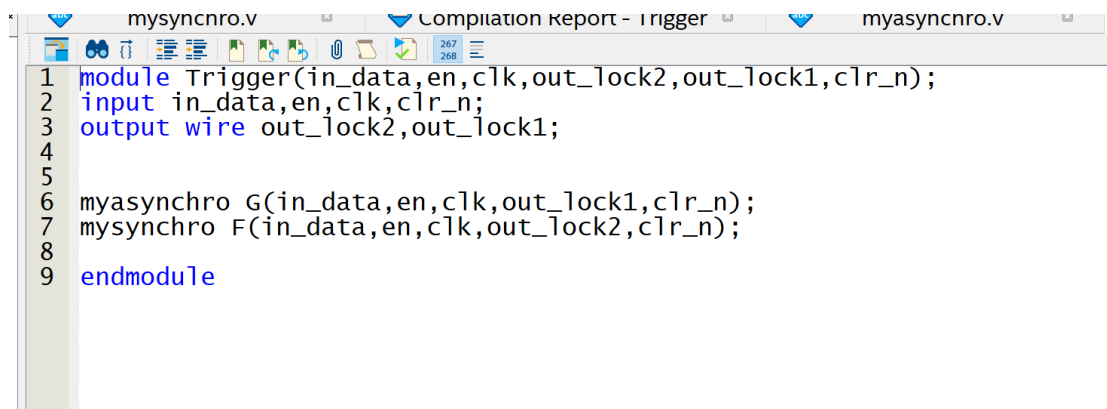
三. 实验设备环境

硬件器材: FPGA 开发板.

软件平台: Qaurtus 开发平台.

四. 实验步骤 / 过程 (设计思路、设计代码、测试代码、仿真结果和硬件实现等的截图代码等)

➤ 设计思路:



```
mysyncnro.v  Compilation Report - Trigger  myasyncnro.v
1 module Trigger(in_data,en,clk,out_lock2,out_lock1,clr_n);
2   input in_data,en,clk,clr_n;
3   output wire out_lock2,out_lock1;
4
5
6   myasynchro G(in_data,en,clk,out_lock1,clr_n);
7   mysynchro F(in_data,en,clk,out_lock2,clr_n);
8
9   endmodule
```

基于PPT例子 (和查找资料)

```
mysynchro.v
1 module mysynchro(in_data,en,clk,out_lock2,clr_n);
2
3 input in_data,en,clk,clr_n;
4 output reg out_lock2;
5
6 initial out_lock2 = 0;
7
8 always @ (posedge clk or negedge clr_n) begin
9     if(!clr_n)begin
10         if(en)
11             out_lock2 <= 0;
12         else
13             out_lock2 <= out_lock2;
14     end
15     else begin
16         if(en)out_lock2 <= in_data;
17         else out_lock2 <= out_lock2;
18     end
19 end
20
21 endmodule
```

```
268
1 module myasynchro(in_data,en,clk,out_lock1,clr_n);
2 input in_data,en,clk,clr_n;
3 output reg out_lock1;
4
5 initial out_lock1 = 0;
6
7 always @ (posedge clk) begin
8     if(en && !clr_n)
9         out_lock1 <= 0;
10    else begin
11        if(en)out_lock1 <= in_data;
12    else out_lock1 <= out_lock1;
13    end
14 end
15
16 endmodule
```

设计思路:

首先设计一个同步清零的触发器（过程简单，不赘述），然后通过仿真测试直到得到正确的结果。接下来再写一个异步清零的触发器，这个触发器和同步清零有所不同，只要清零端为0，并且使能端为1，就可以直接清零。接下来为了完成模块实例化，再添加一个顶层文件Trigger.v。然后再一起仿真测试，直到正确为止。

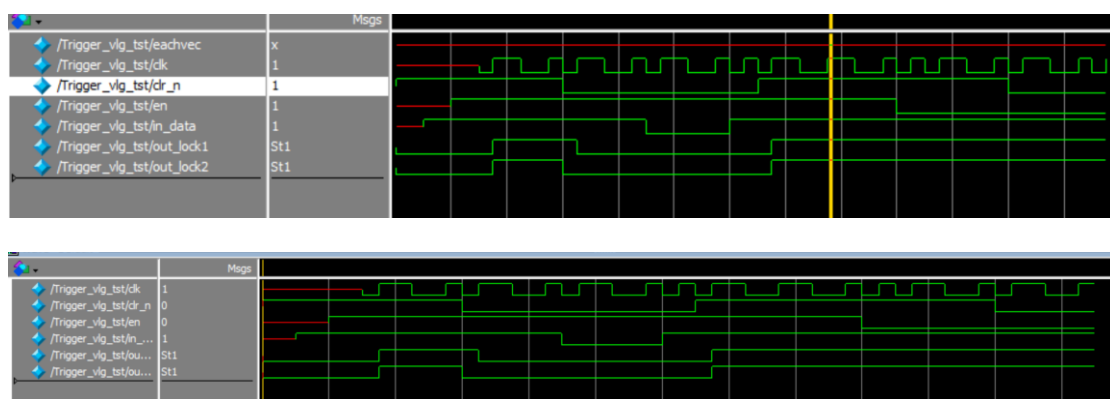
✚ 激励代码:

```

50 );
51 initial
52 begin
53     clr_n = 1'b1;      #10;
54     in_data = 1'b1;    #10;
55     en = 1'b1;         #10;
56     clk = 1'b1;        #10;
57     clk = 1'b0;        #10;
58     clk = 1'b1;        #10;
59
60     clr_n = 1'b0;      #10;
61     clk = 1'b0;        #10;
62     clk = 1'b1;        #10;
63     in_data = 0'b0;    #10;
64     clk = 1'b0;        #10;
65     clk = 1'b1;        #10;
66     in_data = 1'b1;    #10;
67     clr_n = 1'b1;      #10;
68     clk = 1'b0;        #10;
69     clk = 1'b1;        #10;
70     clk = 1'b0;        #10;
71     clk = 1'b1;        #10;
72     en = 1'b0;         #10;
73     in_data = 1'b1;    #10;
74     clk = 1'b0;        #10;
75     clk = 1'b1;        #10;
76     clr_n = 1'b0;      #10;
77     clk = 1'b0;        #10;
78     clk = 1'b1;        #10;
79     $stop;
80 end
81
82 always

```

✚ ModelSim 仿真波形:



➤ 硬件实现:

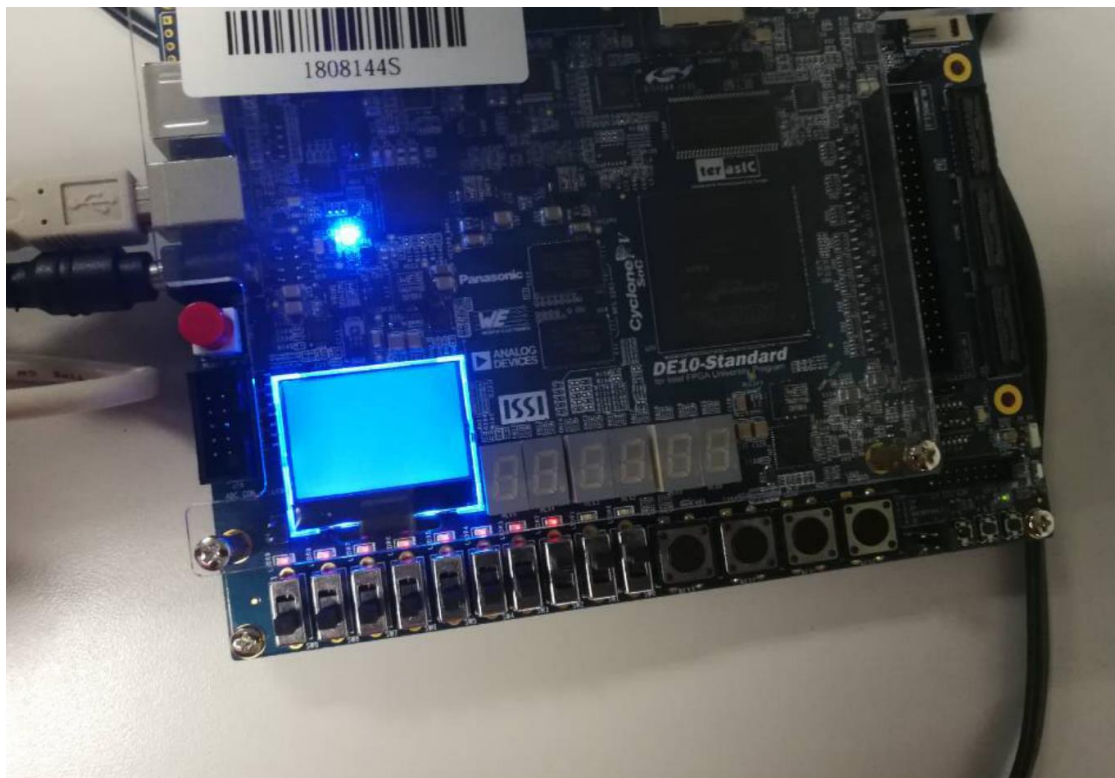
🔗 引脚分配:

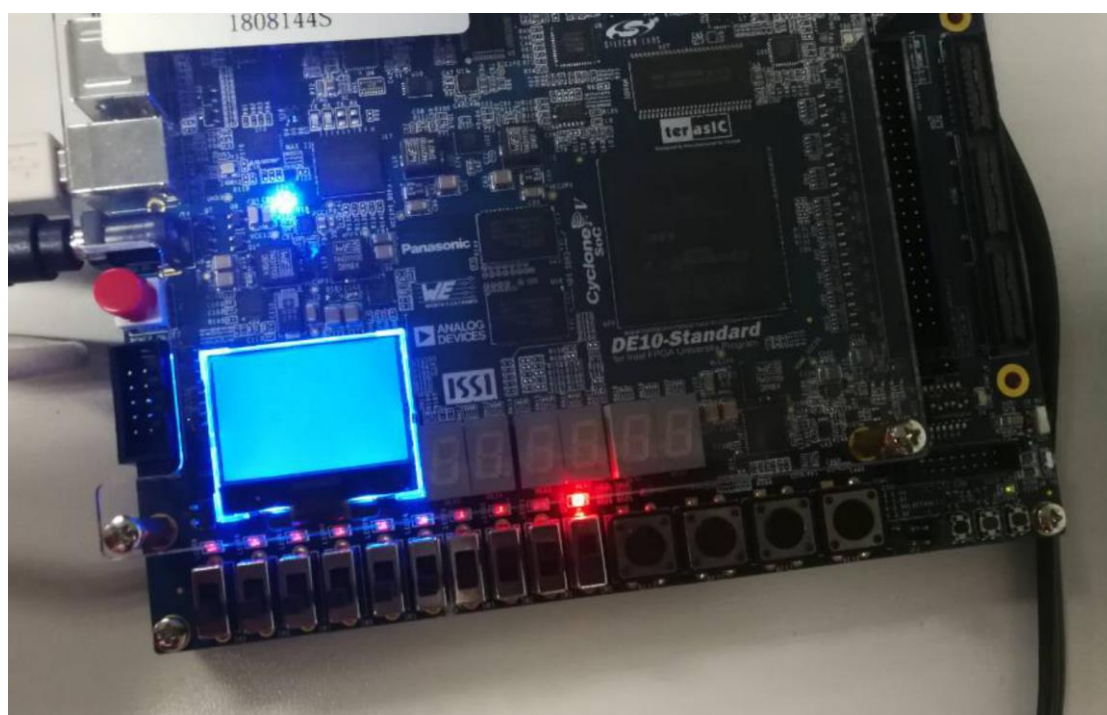
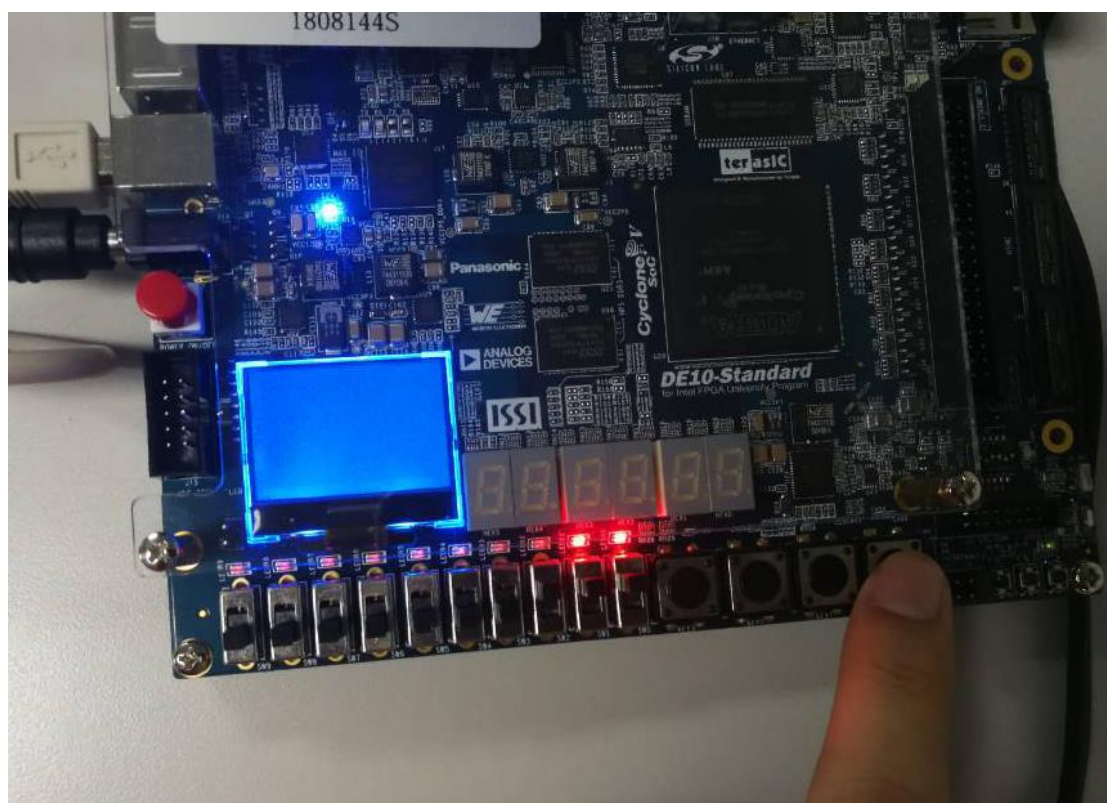
Node Name	Direction	Location	O Ban	EF Grc	Stand
clk	Input	PIN_AJ4	3B	B...0	2.5...lt)
clr_n	Input	PIN_AB30	5B	B...0	2.5...lt)
en	Input	PIN_Y27	5B	B...0	2.5...lt)
in_data	Input	PIN_AB28	5B	B...0	2.5...lt)
out_lock1	Output	PIN_AA24	5A	B...0	2.5...lt)
out_lock2	Output	PIN_AB23	5A	B...0	2.5...lt)
<<new node>>					

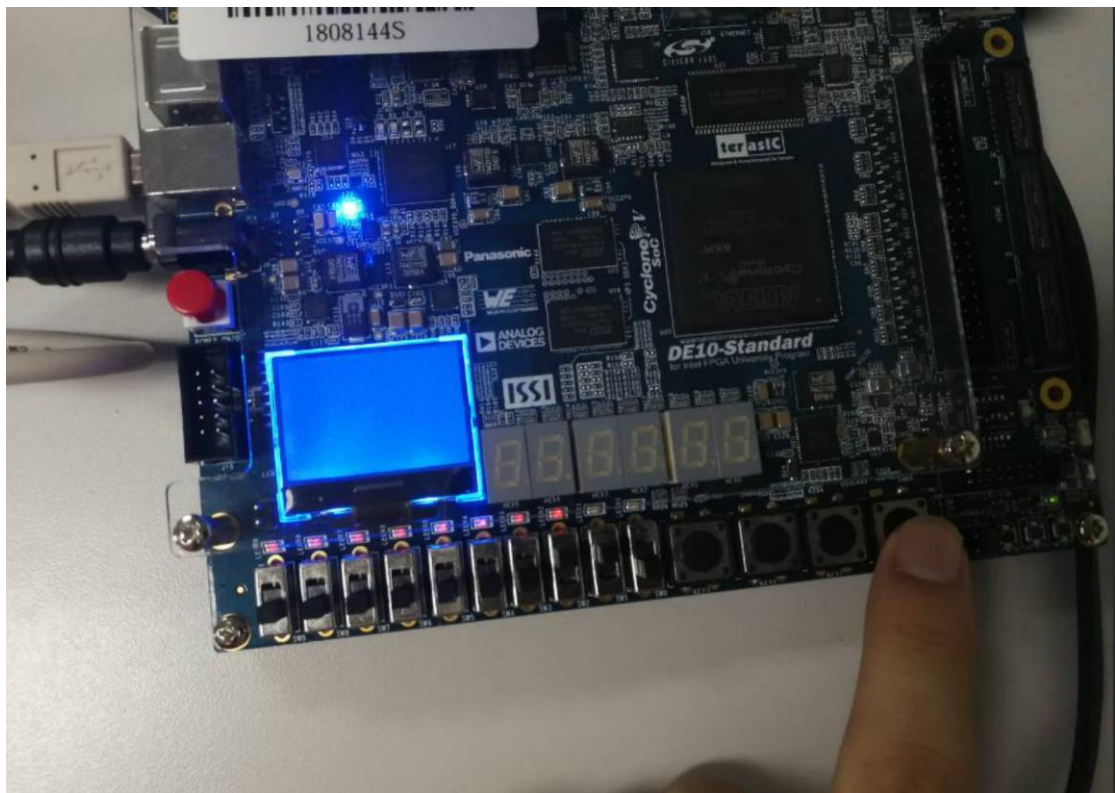
➤ 开发板实现

Led1 是使能端, Led0 是清零端, Led1 是 in_data 端, key0 是时钟端。

开始, en=1, in_data=1, clr_n=1, 然后时钟变化, 两个灯都亮起来, 接下来清零端置0,
异步清零的触发器的输出变为零, 时钟变化, 此时同步清零的触发器的输出也变为零。







五. 实验中遇到的问题及解决方案（请具体的描述问题和解决方法）

- a) 连接模块端口的中间变量使用了 `reg` 类型，使编译出错，解决方案：改用 `wire` 类型，而不应该是 `reg` 类型。
- b) 第一次使用模块实例化，出现了一些技术性的 `bug`，解决方案：查找相关资料，一步一步解决问题。
- c) `always @ (posedge clk or negedge clr_n)` 报错，解决方案：在 `always` 里面首先 `if(!clr_n)`，如此，便不会报这个错误了。

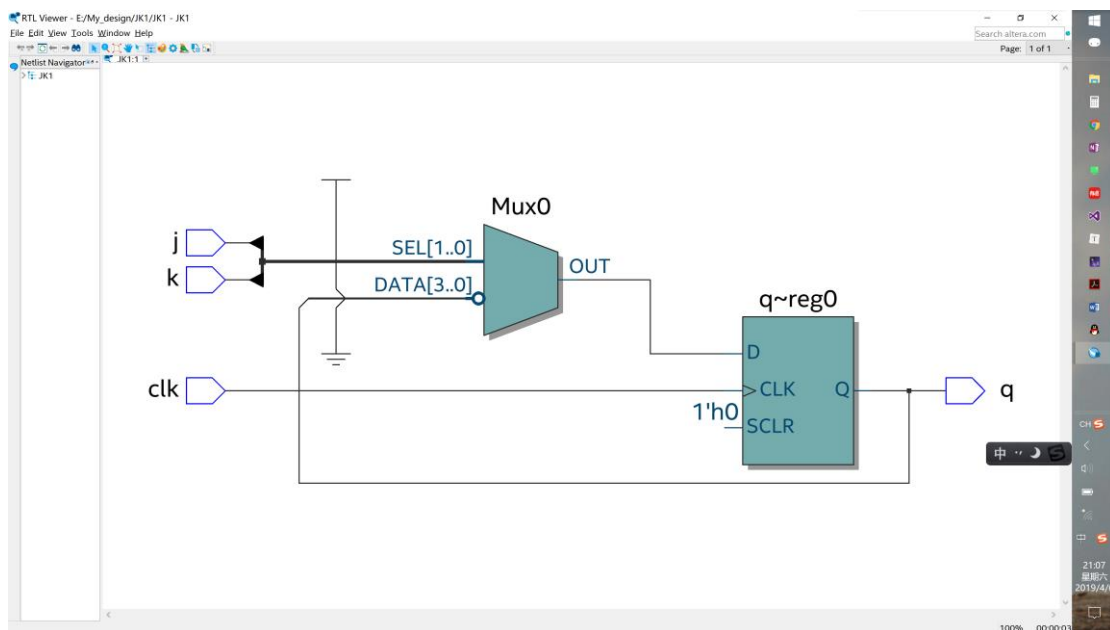
六. 实验得到的启示（积极思考）

面对自己陌生的知识以及实验过程，不要急于动手操作，着急只会使实验变得更加复杂，应该好好查阅资料，了解大概的框架之后，再动手去实验。

预习部分补：用阻塞赋值语句设计一个 JK 触发器

(因为 word 不像 LaTeX 一样好排版, 所以加在后面)

```
2 module JK1(clk, j, k, q);
3
4 input clk, j, k;
5 output q;
6 reg q;
7 wire qb;
8 always@(posedge clk)           //时钟上升沿到来时, 判断jk的值
9 begin
10     case({j,k})
11     2'b00: q <= q;             //如果{j,k}=00, 则触发器处于保持状态
12     2'b01: q <= 1'b0;         //如果{j,k}=01, 则触发器置1
13     2'b10: q <= 1'b1;         //同理10, 清零
14     2'b11: q <= ~q;          //11, 翻转
15     default: q <= q;
16     endcase
17 end
18 assign qb = ~q;
19 endmodule
```



JK 触发器是数字电路触发器中的一种基本电路单元。JK 触发器具有置 0、置 1、保持和翻转功能。

JK 触发器和触发器中最基本的 RS 触发器结构相似, 其区别在于, RS 触发器不允许 R 与 S 同时为 1, 而 JK 触发器允许 J 与 K 同时为 1。当 J 与 K 同时变为 1 的同时, 输出的值状态会反转。也就是说, 原来是 0 的话, 变成 1; 原来是 1 的话, 变成 0。

七. 通过这次实验附加的学习内容:

Verilog 中典型的 counter 逻辑是这样的:

```
always@(posedge clk or negedge reset) begin
    if(reset == 1'b0)
        reg_inst1 <= 8'd0;
    else if(clk == 1'b1)
        reg_inst1 <= reg_inst1 + 1'd1;
    else
        reg_inst1 <= reg_inst1;
end
```

clk 为什么要用 posedge, 而不用 negedge 呢?

一般情况下, 系统中统一用 posedge, 避免用 negedge, 降低设计的复杂度, 可以减少出错。

在 ModelSim 仿真中, 时钟是很严格的, 但是在真实的晶振所产生的 clock 却是不严格的, 比如高电平和低电平的时间跨度不一样, 甚至非周期性的微小波动。如果只使用 posedge, 则整个系统的节拍都按照 clock 上升延对齐, 如果用到了 negedge, 则系统的节拍没有统一到一个点上。上升延到上升延肯定是一个时钟周期, 但是上升延到下降延却很可能不是半个周期。这都会出现问题。

仿真过程中出现了未知的 x:

是因为如果把 @eachvec; 那一行注释掉的话你仿真才能得到一段很长的波形, 不然仿真时间就非常短, 如果在它之前有在这个 always 过程块里规定时钟信号的翻转的话, 这个时钟信号也不会翻转。

网上解答 (当测试文件中有时钟信号, 并且有 @eachvec 时, 仿真时间很短, 如果在它之前有在 always 过程块里规定时钟信号的翻转的话, 这个时钟信号也不会翻转, 那一行注释掉的话仿真才能得到一段很长的波形。但是当测试文件中没有时钟信号, 去掉这一行, 仿真就没有波形。推断 eachvec 是类似时钟信号一样的驱动信号)。

在没有 clk 的程序中, 保留 eachvec, 有 clk 的程序中, 屏蔽 eachvec

八. 意见和建议等

这里的模块实例化的部分，pdf 里面叫我们去找相关资料，但是并不容易找到(也可能是菜鸡太菜了)，建议老师或者助教能够指路（给出相关的链接之类?），谢谢老师和助教!