

实验四 触发器和锁存器

2018 年秋季学期

这是来自她的一条信息，这条信息早已被有意地储存在书写器的存储器中，只要我写下这些文字，便会被触发。

— 《安迪密恩的觉醒》，丹·西蒙斯

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。

常见的有 RS 锁存器、D 锁存器、D 触发器、JK 触发器和 T 触发器等，其中最常用的是 D 触发器，在 FPGA 器件中，时序逻辑电路多由 D 触发器实现。但是理解 RS 锁存器、D 锁存器的工作原理有利于理解时序电路的基本原理。

本次实验的目的是复习锁存器和触发器的工作原理，复习时序电路中电路时序图的分析 and 阅读。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。

4.1 锁存器和触发器

4.1.1 RS 锁存器

图 4-1 是一个使用与非门构成的 RS 锁存器及其真值表，根据电路原理图和真值表可以看出，RS 锁存器有四种不同的状态。当 R 和 S 同时为 0 时 Q 和 \bar{Q} 均为 1，这种情况是不允许的，所以是无效状态。当仅有 S 为 0 时，输出 Q 值为 1，当仅有 R 为 0 时，输出 Q 值为 0，当 R 和 S 全为 1 时，Q 值保持原来的值不变。由电路原理图可以分析出，只有当 R 和 S 全为 0 时，此时的锁存器值无效，其他三种状态都是稳定的，只要输入的值不改变，输出的值也保持不变。

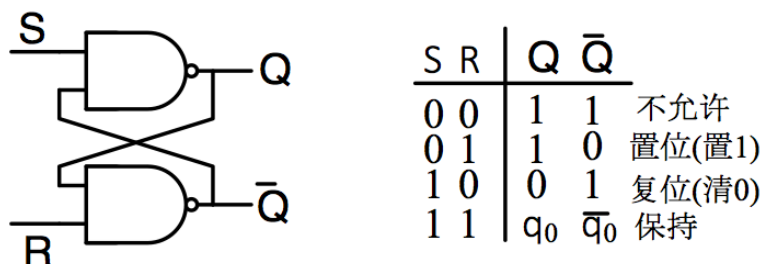


图 4-1: RS 锁存器

4.1.2 时钟触发的 RS 锁存器

在 RS 锁存器的基础上增加两个与非门，就可以构成由时钟信号触发的锁存器。图 2 中在 RS 锁存器的基础上增加了两个与非门，可以看出，当 Clk 为 0 时，这两个新加的两个与非门工作在“关门”状态输出恒为 1，此时对于 RS 锁存器而言处于保持状态，当 Clk 为 1 时，新加的两个与非门工作在“开门”状态，此时 RS 锁存器的工作状态同不带时钟使触发端的 RS 锁存器相同。

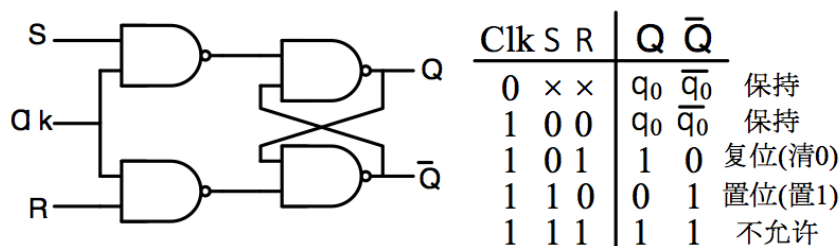


图 4-2: 时钟触发的 RS 锁存器

4.1.3 D 锁存器

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。

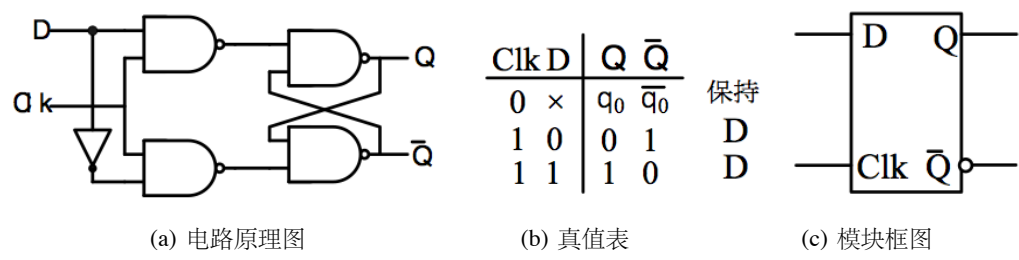


图 4-3: D 锁存器

D 锁存器的代码如表 4-1所示：

表 4-1: D 锁存器代码

```
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 锁存器的测试代码如表 4-2 所示：

表 4-2: D 锁存器测试代码

```
1  `timescale 10 ns/ 1 ps
2
3  module D_latch_vlg_tst();
4
5      reg clk;
6      reg in_d;
7      reg en;
8
9      // wires
10     wire out_ql;
11
12     D_latch i1 (
13         // port map - connection between master ports and signals/registers
14         .clk(clk),
15         .in_d(in_d),
16         .en(en),
17         .out_ql(out_ql)
18     );
19
20
21     initial
22     begin
23         // code that executes only once
24
25         clk = 0; in_d = 0; en = 0; #7;
26             in_d = 0; #7;
27             in_d = 1; #7;
28             in_d = 0; #7;
29             in_d = 1; #7;
30         en = 1; #7;
31             in_d = 0; #7;
32             in_d = 1; #7;
33             in_d = 0; #7;
34             in_d = 1; #7;
35     end
36
37     always
38     begin
39         // code executes for every event on sensitivity list
40         #5 clk = ~clk;
41     end
42 endmodule
```

对 D 锁存器进行功能仿真，如图 4-4 所示，由图中可以看出，当 **clk** 信号有效（为 1）时，输出端 Q 的值随着输入端 D 的改变而改变。当 **clk** 无效（为 0）时，无论 D 的值如何改变，输出端 Q 的值都不改变。

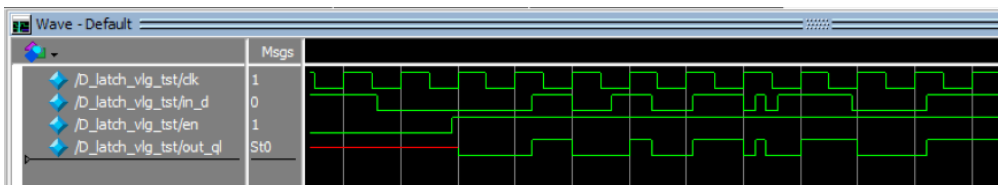


图 4-4: D 锁存器仿真结果

4.1.4 时钟边沿触发的 D 触发器

前面讨论的锁存器都是在时钟为高电平（也可以设计为低电平）时触发的，如果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。

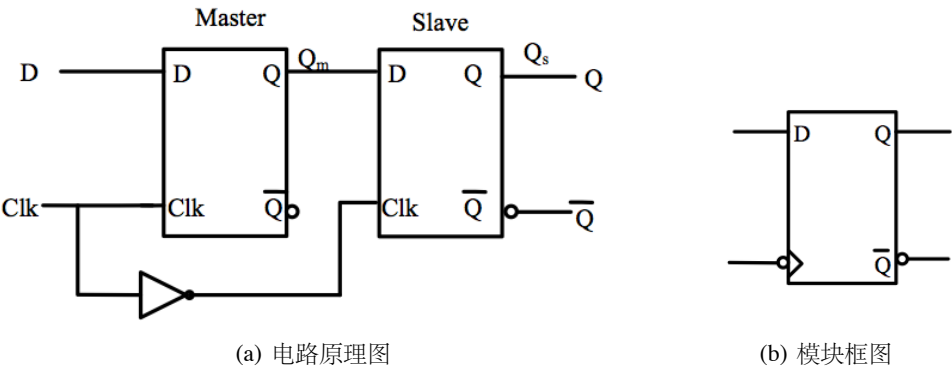


图 4-5: D 触发器

用 Verilog 语言实现 D 触发器非常方便，但也要满足其特定的代码风格。下面以一个上升沿触发的 D 触发器为例，说明触发器的设计方法。常见的触发器的 Verilog HDL 代码及其测试代码如表 4-3 和 4-4 所示。

表 4-3: D 触发器代码

```
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
```

表 4-4: D 触发器测试代码

```
1  `timescale 10 ns/ 1 ps
2  module D_latch_vlg_tst();
3      reg clk;
4      reg in_d;
5      reg en;
6
7      // wires
8      wire out_qt;
9
10     D_trigger i1 (
11         // port map - connection between master ports and signals/registers
12         .clk(clk),
13         .in_d(in_d),
14         .en(en),
15         .out_qt(out_qt)
16     );
17
18     initial
19     begin
20         // code that executes only once
21
22         clk = 0; in_d = 0; en = 0; #7;
23         in_d = 0; #7;
24         in_d = 1; #7;
25         in_d = 0; #7;
26         en = 1; #7;
27         in_d = 0; #7;
28         in_d = 1; #7;
29         in_d = 0; #7;
30         in_d = 1; #7;
31         en = 0; #7;
32         in_d = 0; #7;
33         in_d = 1; #7;
34         in_d = 0; #7;
35         in_d = 1; #7;
36
37         $stop;
38     end
39
40     always
41     begin
42         // code executes for every event on sensitivity list
43         #5 clk = ~clk;
44     end
45 endmodule
```

表 4-4 中的测试代码中有一条 `$stop` 指令，这是 Verilog HDL 的一个系统任务。在仿真的时候，默认的是一直仿真下去，如果在测试代码中加入 `$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。

对实现的 D 触发器进行功能仿真，如图 4-6 所示。从图中可以看出，在每个时钟信号的上升沿，输出 `out_qt` 读取此时刻 `in_d` 的值输出，一直保持到下一个时钟上升沿，下一个时钟上升沿到来时，`out_qt` 再重新读取 `in_d` 的值。在图中我们还可以看出，当输出 `out_qt` 没有被赋值时，此时 `out_qt` 被当成是任意值，在仿真图中以红线表示。

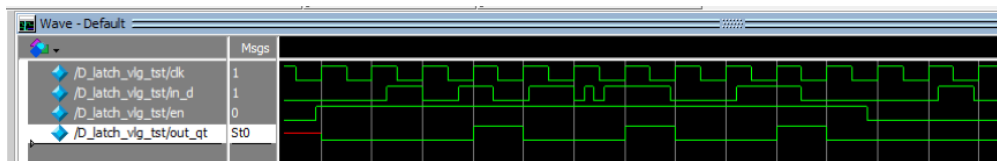


图 4-6: D 触发器仿真结果

4.2 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。

从下面的两个例子中可以看出这两种赋值语言在综合时会有何不同。

例 1 用阻塞赋值语句来设计两个触发器，其 Verilog 语言代码如图 4-7(a) 所示。分析与综合后，点击 `Tools→Netlist Viewers→RTL Viewer` 查看其寄存器传输级视图，如图 4-7(b) 所示，从 RTL 图中可以看出，程序综合出了两个并列的触发器，`out_lock1` 和 `out_lock2` 的值同时跟着输入信号 `in_data` 值的改变而改变。


```
always @(posedge clk)
```

```
  if(in_en)
  begin
```

```
    out_lock1 = in_data;
```

```
    out_lock2 = out_lock1;
```

```
  end
```

```
  else
```

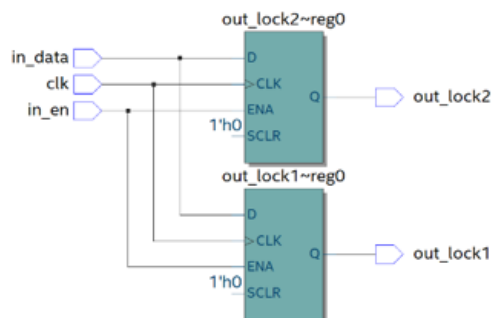
```
  begin
```

```
    out_lock1 = out_lock1;
```

```
    out_lock2 = out_lock2;
```

```
  end
```

(a) 代码



(b) RTL 视图

图 4-7: 阻塞赋值语句设计的触发器

再对设计好的电路进行功能仿真，如图 4-8 所示。

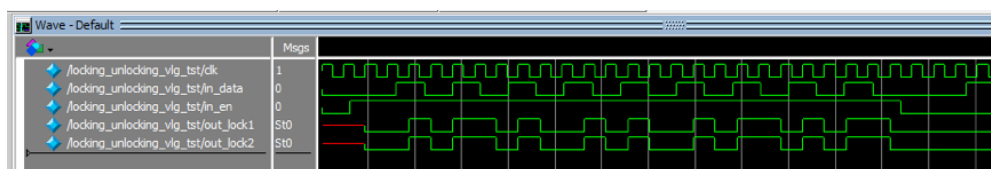


图 4-8: 阻塞赋值语句设计的触发器的仿真图

例 2 用非阻塞赋值语句来设计两个触发器，其 Verilog HDL 语言代码如图 4-9(a)所示。分析与综合后查看其寄存器传输级视图，如图 4-9(b)所示，从图中可以看出，程序综合出了两个级联的触发器。输出为 out_unlock1 的触发器的输入端是 in_data，而输出为 out_unlock2 的触发器输入信号是 out_unlock1。

```
always @(posedge clk)
```

```
  if(in_en)
  begin
```

```
    out_unlock1 <= in_data;
```

```
    out_unlock2 <= out_unlock1;
```

```
  end
```

```
  else
```

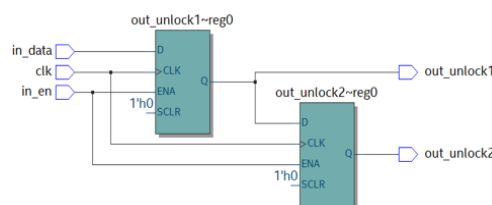
```
  begin
```

```
    out_unlock1 <= out_unlock1;
```

```
    out_unlock2 <= out_unlock2;
```

```
  end
```

(a) 代码



(b) RTL 视图

图 4-9: 非阻塞赋值语句设计的触发器

再对设计好的电路进行功能仿真，如图 4-10 所示。

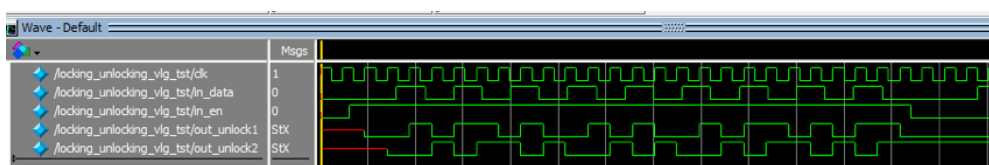


图 4-10: 非阻塞赋值语句设计的触发器的仿真图

从上述的图中可以看出，在每个时钟上升沿到来时，进入 `always` 程序块执行其中的语句，这时已经采样到 `in_data` 的值，但是并没有立即赋值给变量 `out_unlock1`，所以 `out_unlock1` 保持原来的值不变。在 `always` 块执行完毕后，`in_data` 的值被立即赋给 `out_unlock1`，在 `always` 块中保持的 `out_unlock1` 的原来的值被赋给了 `out_unlock2`。到下一个时钟上升沿到来时，程序又进入 `always` 块中执行，`always` 块执行完毕时，`in_data` 的新的值被赋给了 `out_unlock1`，原先的 `out_unlock1` 被赋给变量 `out_unlock2`。

在编写 Verilog DHL 代码时一定要注意阻塞和非阻塞赋值的使用，我们总结了一些 Verilog 代码的编码指导方针，请同学们在编写代码时注意应用。

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. 不要在同一个 `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。

4.3 实验内容

4.3.1 分析阻塞和非阻塞 RTL 视图和仿真结果

请你建立两个工程，分别研究阻塞赋值和非阻塞赋值的 RTL 级视图和仿真结果，步骤如下：

1. 新建工程，用阻塞赋值语句设计两个触发器；保存 Verilog 语言文件。
2. 在 Tools 栏，点击 Netlist Viewers 栏下的 RTL Viewer 查看生成的 RTL Schematic，看看在用阻塞赋值语句生成两个触发器的实际电路原理。
3. 新建另一个工程，用非阻塞赋值语句实现两个触发器，重复上述步骤，比较两种触发器实现方式在硬件电路实现上的异同。

4.3.2 设计一个同步清零和一个异步清零的 D 触发器

查阅资料，分析同步清零和异步清零的不同，并请在一个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。



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 ...
```

如果异步清零端（假设命名为 `clr`）是“1”有效，那么在敏感列表中应该是检测 `clr` 的上升沿即代码应该这样：

```
1 always @(posedge clk or posedge clr)
2   if(clr)
```

```
3     begin    ...    end
4 else    ...
```

这样的代码也是错误的

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

4.3.3 实践模块实例化方法

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。

建议的实验过程

1. 新建工程，比如，取工程名为“Trigger”，然后在此工程中重新添加一个文件，如“synchro.v”，完成同步清零上升沿触发的触发器的设计。编写测试代码，进行仿真设置，对其进行仿真，查看仿真结果。
2. 还是在此工程中，再新建一个文件，如“asynchro.v”，完成异步清零的下降沿触发的触发器的设计。完成设计后，保存文件。点击项目导航栏的 Hierarchy→Files

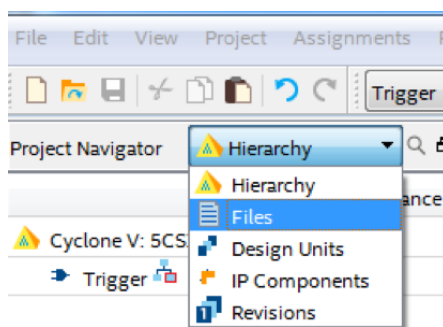


图 4-11: 选择文件视图

本项目的所有文件会出现在项目导航栏，把鼠标放在文件名“asynchro.v”上，右击，选择“Set as Top-Level Entity”，将“asynchro.v”设为顶层实体。

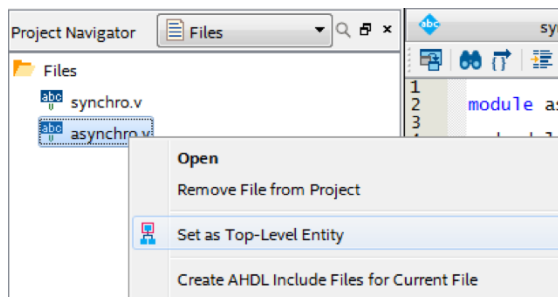




图 4-12: 设置顶层文件

点击  对“asynchro.v”进行分析与综合。

分析与综合完成之后，可以在原来为同步触发器“synchro.v”编写的测试代码的基础之上进行修改，不改变原来的仿真设置，对“asynchro.v”进行功能仿真。也可以重新为“asynchro.v”编写一个新的测试文件，修改文件，重新进行仿真设置，对“asynchro.v”进行仿真。

3. 仍然在此工程中，再新建一个文件，如“Trigger.v”，完成顶层实体的设计。在这个顶层实体中对上述两个同步清零和异步清零的触发器进行实例化，在这一个模块中，完成两个触发器的设计。

完成设计后，保存文件，此时“Trigger.v”会自动出现在 File 栏下。将鼠标放在“Trigger.v”文件上，右击，选择“Set as Top-Level Entity”，将“Trigger.v”设为顶层实体。点击  对“Trigger.v”进行分析与综合。

4. 分析与综合完成之后，可以在原来为同步触发器“synchro.v”编写的测试代码的基础之上进行修改，不改变原来的仿真设置，对“Trigger.v”进行功能仿真。也可以重新为“Trigger.v”编写一个新的测试文件，修改文件，重新进行仿真设置，对“Trigger.v”进行仿真。

分析仿真图，详述清零端和时钟端对触发器工作的控制情况。

5. 检查电路器的功能正确后，对工程进行分析与布局；然后进行引脚配置；再进行编辑，产生二进制文件，最后将编辑好的文件下载到 FPGA 开发平台上进行验证。这里要求用 FPGA 开发板上的按钮作为时钟输入端，即按钮按下，时钟由高电平变为低电平，产生一个下降沿；按钮松开，时钟由低电平变为高电平，产生一个上升沿。检查触发器的工作状态是否符合要求。