基于FPGA的时序逻辑电路设计和实现
EDA实验二

赵晓燕
电工电子实验中心

主要内容

一. 基本语法

二. always模块

三. 时序电路的HDL描述

四. 阻塞式与非阻塞式赋值

五. 有限状态机的HDL描述

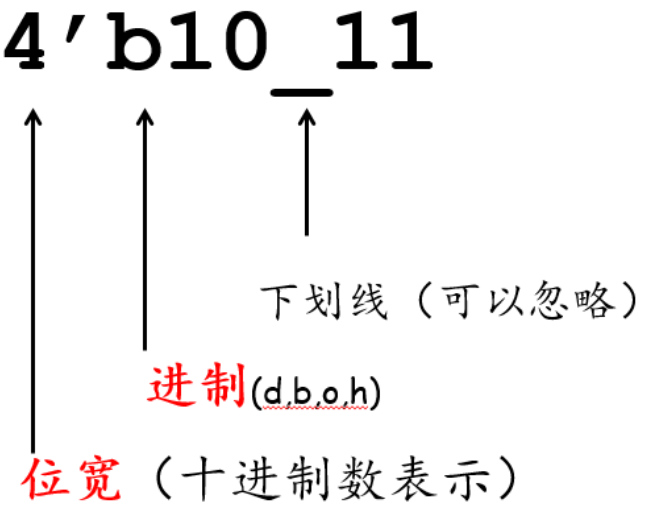
六. EDA实验二内容

基本语法-----数据类型

常量

- 数字

```
8'b0000_0100
6'h1f
128
```



- 参数

```
parameter WIDTH = 8;

wire [WIDTH-1:0] data;
```

变量

- 线型 （wire）　　assign赋值 =
- 寄存器型(reg)　　always赋值<=

基本语法-----运算符

关系运算符(>、<、>=、<=、==、!=)

- (4'b1011 < 4'b0111) = 0
- (4'b1011 != 4'b0111) = 1

条件运算符(?:)

```
<conditional_expression> ? <expression1> : <expression2>
```

- assign out=(sel)?in1:in0;

级联运算符{}

- 级联多个运算数
  {5’b10110, 2’b10, 1’b0}=8’b1011_0100
- 同一个运算数级联多次
  {3{3’b101}}=9’b101_101_101

基本语法----运算符

位运算符~、 &、|、^ 、~^（异或）

- 运算数是矢量，运算逐位进行

```
4'b0100 | 4'b1001 = 4'b1101
~8'b0110_1100 = 8'b1001_0011
```

~a	NOT
a & b	AND
a | b	OR
a ^ b	XOR
a ~^ b	XNOR

逻辑运算符 ！、&&、||

- 产生一个逻辑值

```
4'b0000 || 4'b0111 = 1
4'b0000 && 4'b0111 = 0
!4'b0000 = 1
```

!a	NOT
a && b	AND
a || b	OR

基本语法-----运算符

算数运算符（+、−、*、/、%）

- /、%（模运算）运算符当右边运算数是2的幂次方时可综合

移位运算符（>>、<<）

```
8'b0011_1100 >> 2 = 8'b0000_1111
8'b0011_1100 << 2 = 8'b1111_0000
```

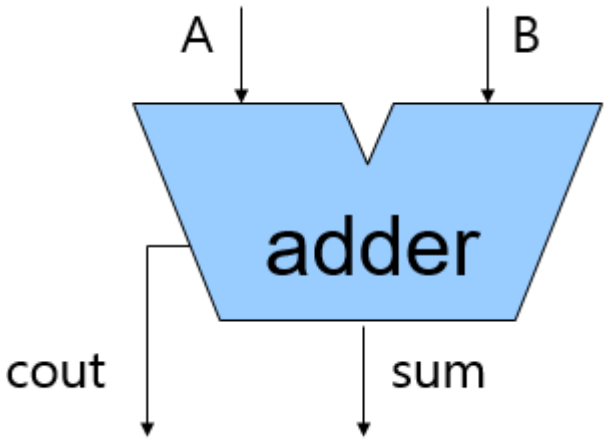
位缩减运算符 & ~& | ~| ^ ^~

- 只有一个运算数
- 对每个向量完成逐位运算
- 产生1比特运算结果

```
&4'b0101 = 0 & 1 & 0 & 1 = 0
|4'b0101 = 0 | 1 | 0 | 1 = 1
```

&a	AND
~&	NAND
|	OR
~|	NOR
^	XOR

模块定义



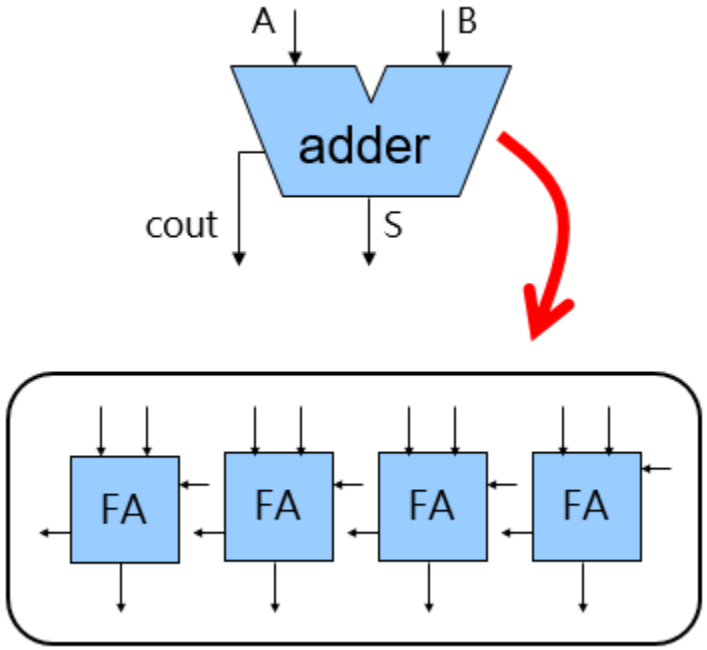

```
module adder( A, B, cout, sum );
  input  [3:0] A, B;
  output       cout;
  output [3:0] sum;

  // HDL modeling of
  // adder functionality

endmodule
```

1995版

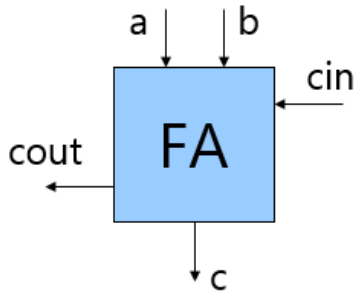
```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] sum );

  // HDL modeling of 4 bit
  // adder functionality

endmodule
```

2001版

模块实例化

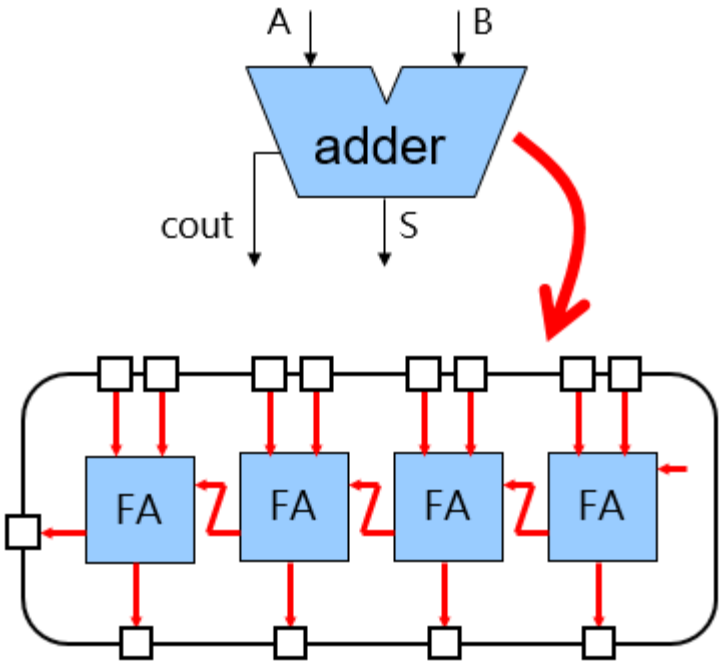



```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );

  FA fa0( ... );
  FA fa1( ... );
  FA fa2( ... );
  FA fa3( ... );

endmodule
```




```
module FA( input  a, b, cin
           output cout, sum );

  // HDL modeling of 1 bit
  // adder functionality

endmodule
```

模块连接




```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );

  wire c0, c1, c2;

  FA fa0( A[0], B[0], 0,  c0, S[0] );
  FA fa1( A[1], B[1], c0, c1, S[1] );
  FA fa2( A[2], B[2], c1, c2, S[2] );
  FA fa3( A[3], B[3], c2, cout, S[3] );

endmodule
```

```
FA fa0( .a(A[0]), .b(B[0]),
        .cin(0), .cout(c0),
        .sum(S[0] );

  FA fa1( .a(A[1]), .b(B[1]),
          ...
```

- 定义模块间连线 wire
- 端口连接可通过变量位置或名字

always 模块表达形式

always@(敏感列表)

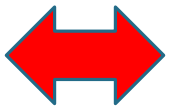
赋值语句

- always@(a,b,c)
- always@(*)
- always@(posedge clk)
- always@(negedge clk,posedge rst)

always 模块使用

- 时序always模块：always@(posedge clock)用<=赋值；
- 组合always模块：always@（*）用=赋值；
- always模块中的变量必须声明为reg型；
- 一个always模块中不能混用<=，=赋值；
- 等价语句

```
reg z
always@ (*)
z=x&&y
```



```
assign z=x&&y
```

组合always模块

```
module mux4( input  a, b, c, d,
             input [1:0] sel,
             output reg out  );

  always @( * )
  begin
    case ( sel )
      0 : out = a;
      1 : out = b;
      2 : out = c;
      3 : out = d;
    endcase
  end

endmodule
```

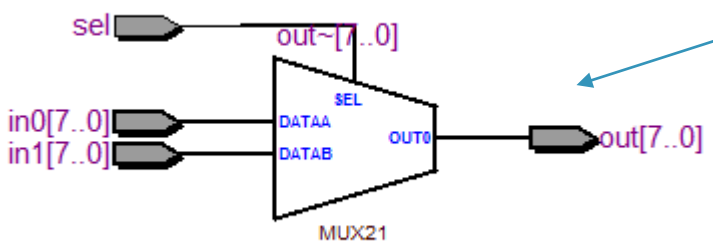
- always语句块可以包含:
  if else语句、case语句、for、wile循环语句
- 一般每个if 语句要与else匹配
- case语句分支要完整，否则要包括default

8位二选一数据选择器

```
module Mux_2to1_8bit(
  input  wire[7:0] in0,
  input  wire[7:0] in1,
  input  wire      sel,
output  wire [7:0]  out
);

assign out=(sel)?in1:in0;

endmodule
```

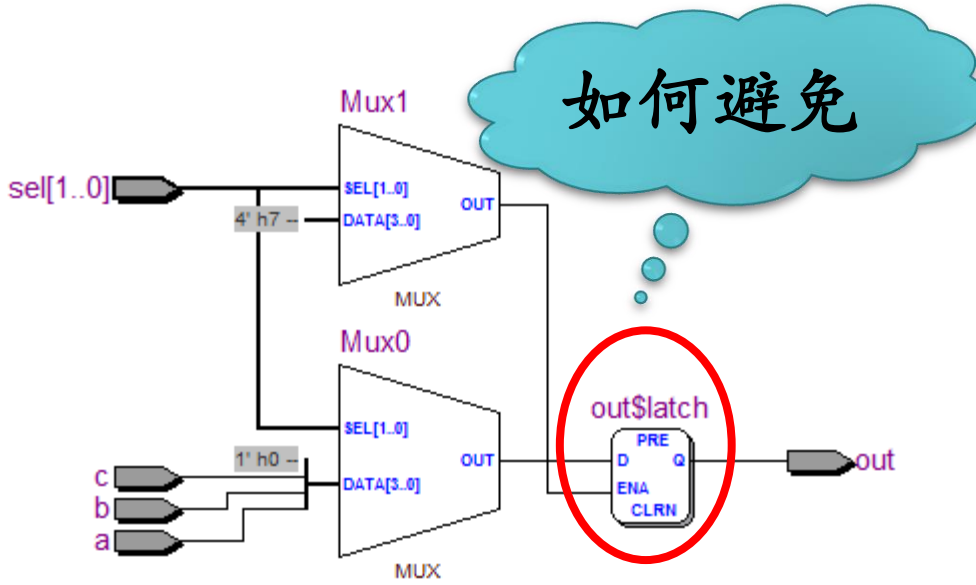
```
output  reg [7:0]  out
always@(*)
        case (sel)
            1'b0:  out=in0;
            1'b1:  out=in1;
         endcase
```

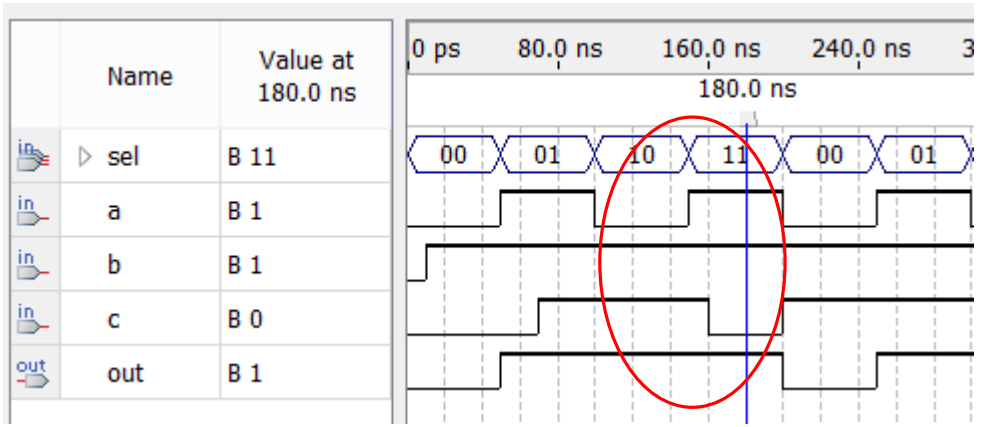
```
output  reg [7:0]  out
always@(*)
        if (sel==0)
            out=in0;
         else
            out=in1;
```



分支不完全的case语句

```
module mux4(input  a, b, c,
            input [1:0] sel,
            output reg out );


  always @( a or b or c or sel )
  begin
    case ( sel )
      2'b00 : out = a;
      2'b01 : out = b;
      2'b10 : out = c;
    endcase
  end

endmodule
```

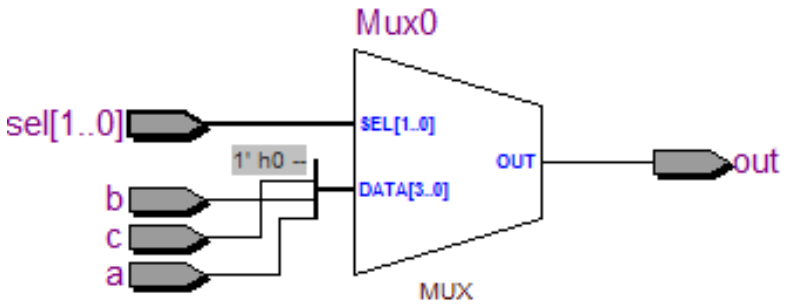




分支不完全的case语句

```
always @( a or b or c or sel
)
  begin
    case ( sel )
      2'b00 : out = a;
      2'b01 : out = b;
      2'b10 : out = c;
      default: out=1'bx;
    endcase
  end
```

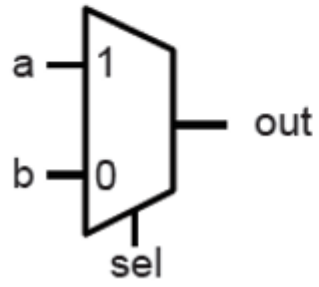
```
always @( a or b or c or sel
)
  begin
    out=1'bx;
    case ( sel )
      2'b00 : out = a;
      2'b01 : out = b;
      2'b10 : out = c;
    endcase
  end
```



组合与时序always 模块

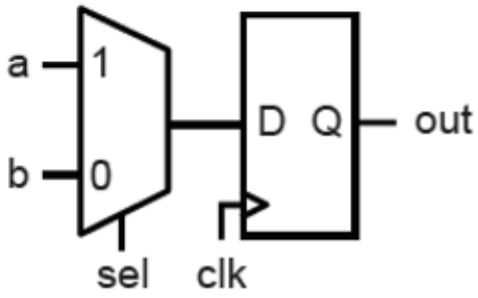
Combinational

```
module combinational(a, b, sel,
                             out);
  input a, b;
  input sel;
  output out;
  reg out;
  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end
endmodule
```



Sequential

```
module sequential(a, b, sel,
                          clk, out);
  input a, b;
  input sel, clk;
  output out;
  reg out;
  always @ (posedge clk)
  begin
    if (sel) out <= a;
    else out <= b;
  end
endmodule
```



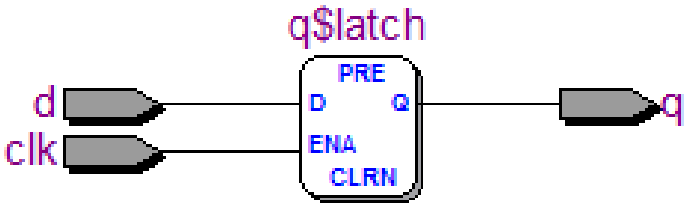
时序电路-----D触发器

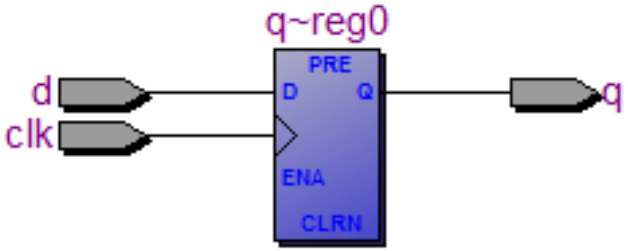
```
module latc1
(input clk,
 input d,
 output reg q);
always @(clk or d )
begin
    if(clk)
     q<=d;
    end
endmodule
```

电平触发

```
module FF0
(input clk,
 input d,
 output reg q);
always @(posedge clk )
begin
    q<=d;
    end
endmodule
```

边沿触发

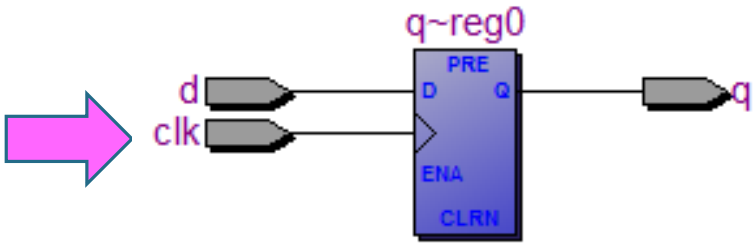




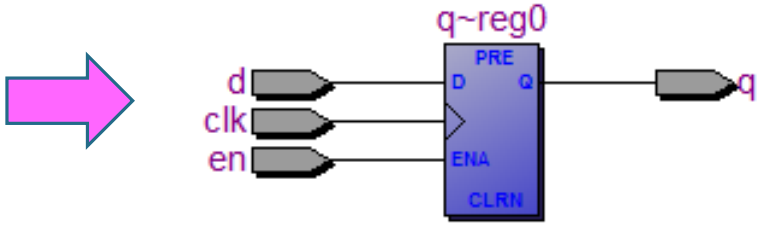
D型锁存器

时序电路------D触发器

```
module FF0(input clk,input
d,output reg q);
always @( posedge clk )
begin
    q<=d;
end
endmodule
```




```
module FF0(input clk,input
d,output reg q);
always @( posedge clk )
begin
    if  (en)
    q<=d;
end
endmodule
```



时序电路------D触发器

```
module FF1#(parameter  WIDTH=1)
(input clk,
 input [WIDTH-1:0]d,
 input en,
 output reg [WIDTH-1:0] q);
always @(posedge clk)
begin
    if(en)
     q<=d;
    end
endmodule
```

时序电路------D触发器

```
always @( posedge clk )
begin
   if (~resetN)
          Q<=0;
   else if (enble)
          Q<=D;//同步置数
end
```

同步复位

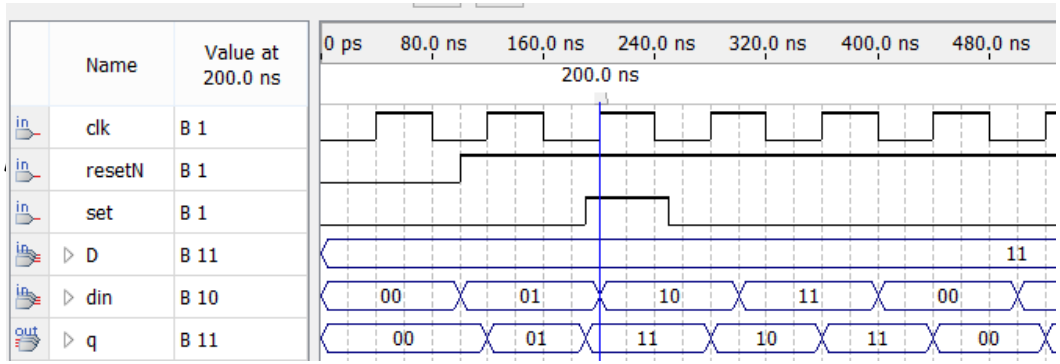
```
always @( posedge clk or negedgde resetN )
begin
   if (~resetN)
          Q<=0;
   else if (enble)
          Q<=D;//同步置数
end
```

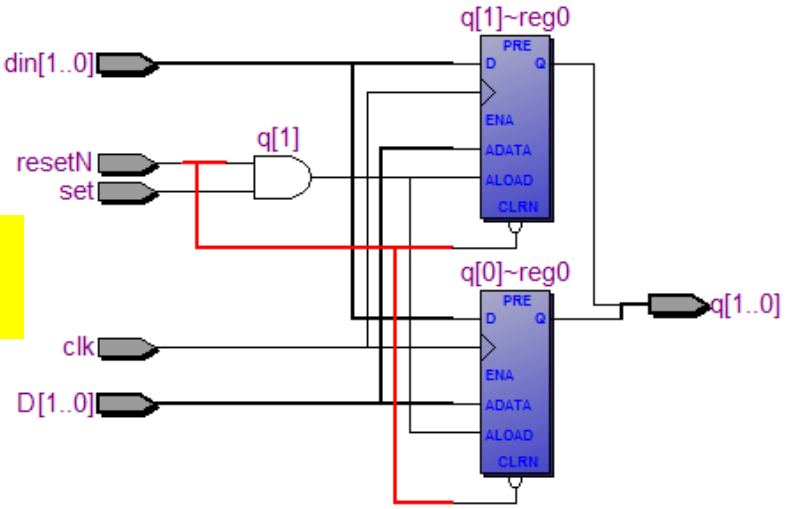
异步复位

时序电路------D触发器

```
module flipflop
(input clk,resetN,set,
 input [1:0]D,din,
 output reg [1:0]q);

always @(posedge clk or negedge resetN or posedge set)
begin
  if (~resetN)
        q<=0;

  else if (set)
        q<=D;//

  else q<=din;
end
endmodule
```

异步置数





时序电路------计数器

```
module dff_asyn3(clk,d,rst,en,q);
input clk,rst,en;
input [1:0]d;
output [1:0]q;
reg [1:0]q;

always@(posedge clk or negedge rst)
 begin
  if(!rst)
      q<=0;//   异步清零
  else  if(!en)

        q<=d;  // 同步置数
   else
        q<=q+1; //

  end
endmodule
```

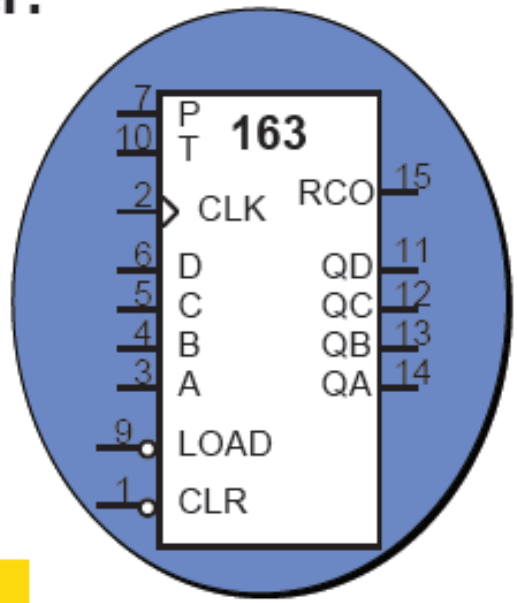
计数

- **Behavioral description of the ‘163 counter:**

```
module counter(LDbar, CLRbar, P, T, CLK, D,
               count, RCO);
  input LDbar, CLRbar, P, T, CLK;
  input [3:0] D;
  output [3:0] count;
  output RCO;
  reg [3:0] Q;

  always @ (posedge CLK) begin
    if (!CLRbar) Q <= 4'b0000;
    else if (!LDbar) Q <= D;
    else if (P && T) Q <= Q + 1;
  end

  assign count = Q;
  assign RCO = Q[3] & Q[2] & Q[1] & Q[0] & T;
endmodule
```

priority logic for control signals

RCO gated by T input



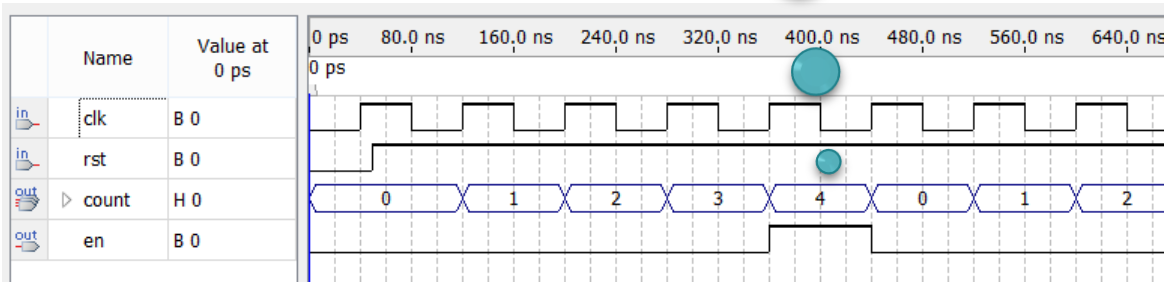
时序电路------分频器

```
module fenp(clki,rst,clk1,clk2);
input clki,rst;
output clk1,clk2;

reg [30:0]q;

always@(posedge clki, negedge rst)

   if(!rst)  q<=0;
   else  q=q+1;

assign clk1=q[16];//输出250hz
assign clk2=q[24];//输出1HZ

endmodule
```

时序电路------分频器

```
module fenp(clk,rst,en,count);
input clk,rst;
output en;
output [3:0] count;
reg [3:0] count;

always@(posedge clk,negedge rst)
begin
    if (!rst) count<=0;
        else count<=(count==4)?0:count+1;
end

assign en=(count==4);

endmodule
```

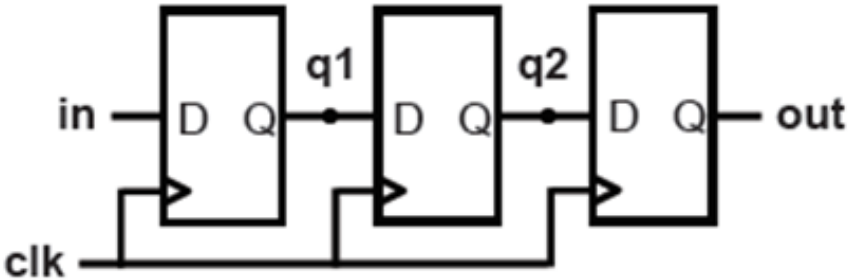
产生使能控制信号



非阻塞（<=）和阻塞赋值（=）

```
always @ (posedge clk)
begin
   q1 <= in;
   q2 <= q1;
   out <= q2;
end
```

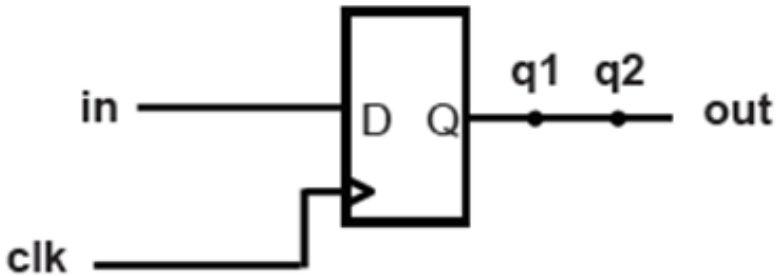
"At each rising clock edge, *q1*, *q2*, and *out* simultaneously receive the old values of *in*, *q1*, and *q2*."



移位寄存器

```
always @ (posedge clk)
begin
   q1 = in;
   q2 = q1;
   out = q2;
end
```

"At each rising clock edge, *q1* = *in*.
After that, *q2* = *q1* = *in*.
After that, *out* = *q2* = *q1* = *in*.
Therefore *out* = *in*."



非阻塞（<=）和阻塞赋值（=）

- 阻塞式赋值

```
always @(a,b) begin
  x=a;
  y=b&x;
  z=x|y;
end
```

a	b	x	y	z
0	0	0	0	0
1	1	1	1	1

- 非阻塞式赋值

```
always @(a,b) begin
  x<=a;
  y<=b&x;
  z<=x|y;
end
```

a	b	x	y	z
0	0	0	0	0
1	1	1	0	0

- 阻塞赋值（=），数据计算与更新同时进行
- 非阻塞赋值（<=），先进行数据计算，所有赋值语句执行完，再进行数据更新。

Coding Guidelines

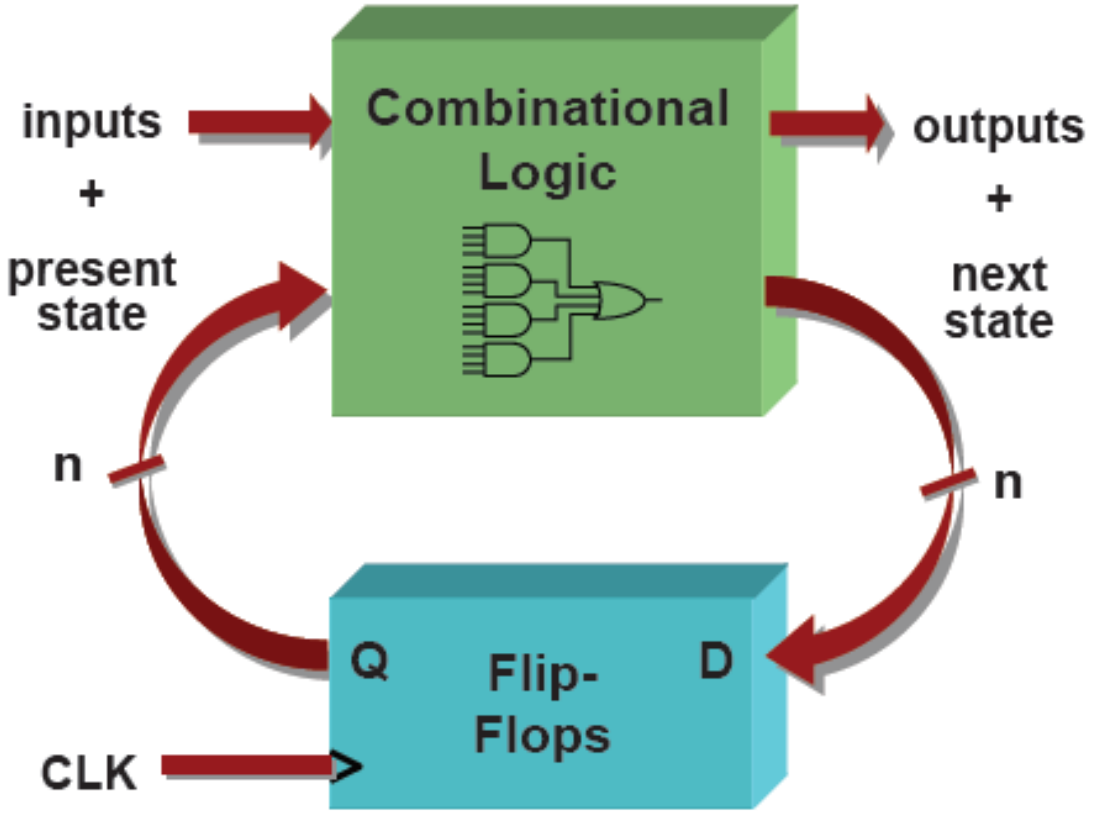
The following helpful guidelines are from the Cummings paper. If followed, they ensure your simulation results will match what they synthesized hardware will do:

1. When modeling sequential logic, use nonblocking assignments.
2. When modeling latches, use nonblocking assignments.
3. When modeling combinational logic with an always block, use blocking assignments.
4. When modeling both sequential and "combinational" logic within the same always block, use nonblocking assignments.
5. Do not mix blocking and nonblocking assignments in the same always block.
6. Do not make assignments to the same variable from more than one always block.
7. Use $strobe to display values that have been assigned using nonblocking assignments.
8. Do not make assignments using #0 delays.

For more info see: http://www.sunburst-design.com/papers/CummingsSNUG2002Boston_NBAwithDelays.pdf

时序电路------有限状态机FSM

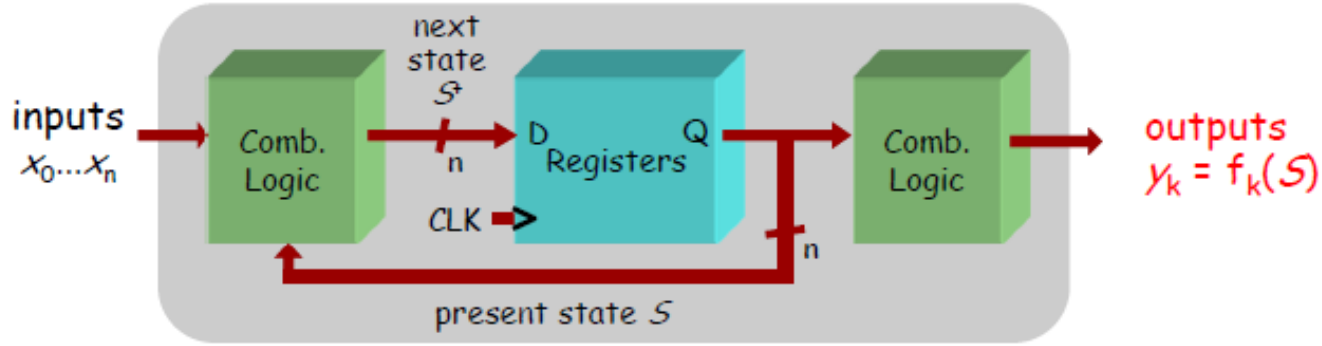
- **Finite State Machines (FSMs) are a useful abstraction for sequential circuits with centralized "states" of operation**
- **At each clock edge, combinational logic computes *outputs* and *next state* as a function of *inputs* and *present state***



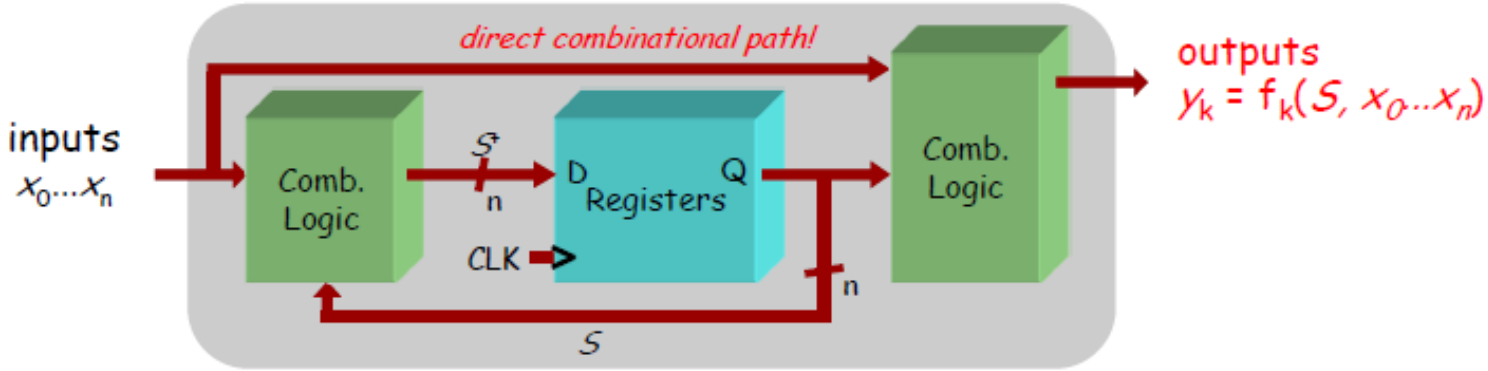
Two Types of FSMs

Moore and Mealy FSMs : different output generation

- Moore FSM:



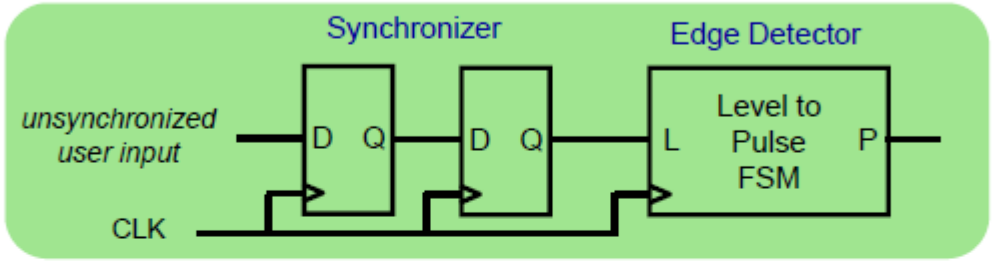
- Mealy FSM:



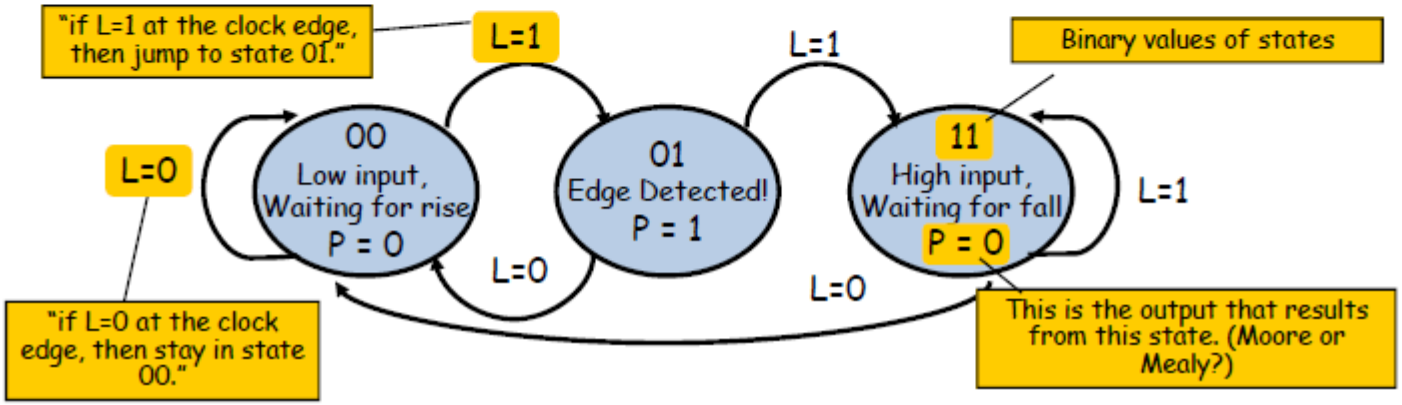
有限状态机FSM设计

Step 1: State Transition Diagram

- Block diagram of desired system:

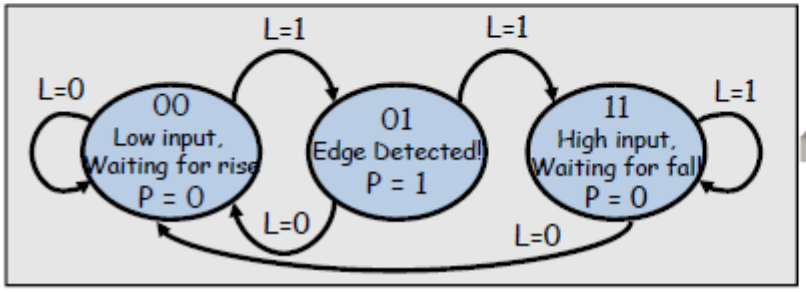


- State transition diagram is a useful FSM representation and design aid:



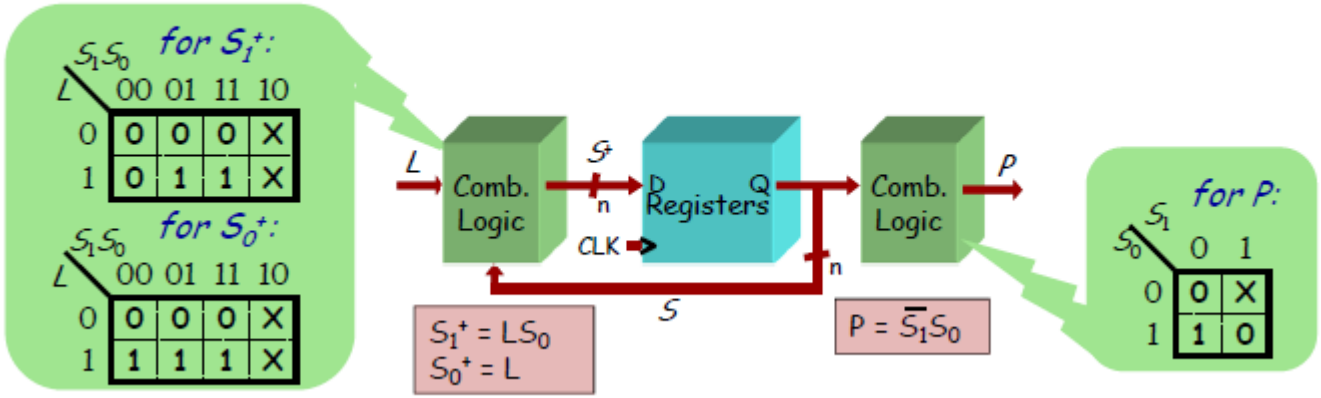
Step 2: Logic Derivation

Transition diagram is readily converted to a state transition table (just a truth table)

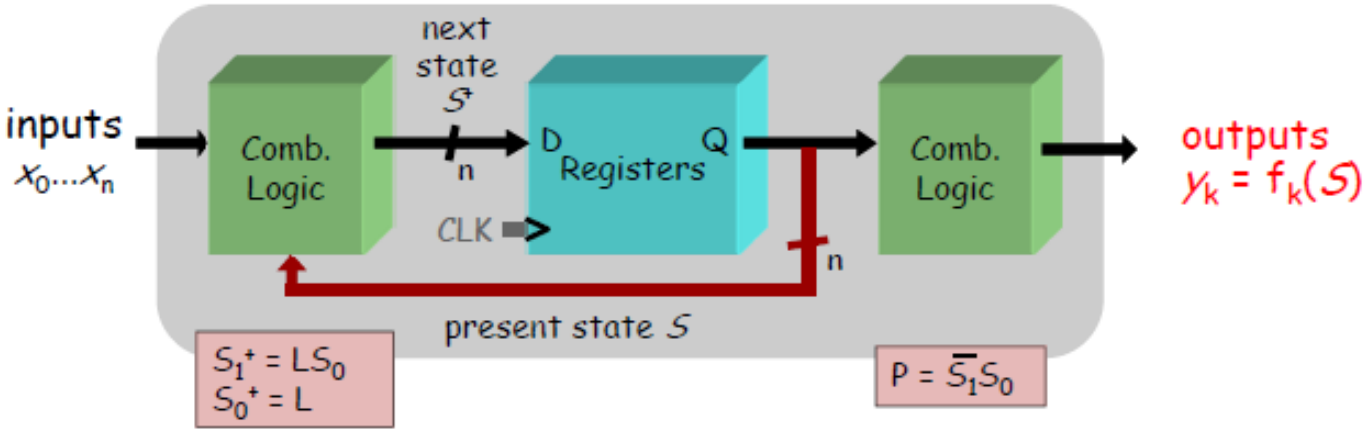


Current State		In	Next State		Out
S_1	S_0	L	S_1^+	S_0^+	P
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	1
1	1	0	0	0	0
1	1	1	1	1	0

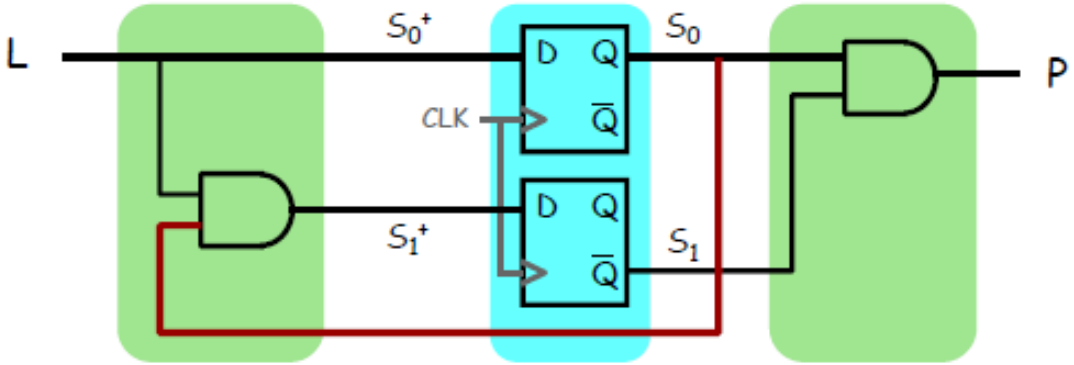
- Combinational logic may be derived using Karnaugh maps



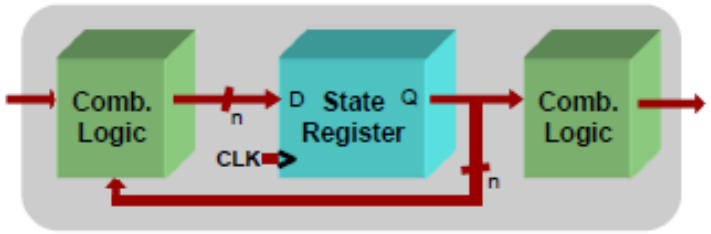
Moore Level-to-Pulse Converter



Moore FSM circuit implementation of level-to-pulse converter:



两段式FSM（输出组合电路）



FSMs are easy in Verilog.
Simply write one of each:

- State register
  (sequential always block)
- Next-state
  combinational logic
  (comb. always block with case)
- Output combinational
  logic block
  (comb. always block *or* assign statements)

```
module mooreVender (
  input N, D, Q, clk, reset,
  output DC, DN, DD,
  output reg [3:0] state);

  reg  next;
```

States defined with parameter keyword

```
parameter IDLE = 0;
parameter GOT_5c = 1;
parameter GOT_10c = 2;
parameter GOT_15c = 3;
parameter GOT_20c = 4;
parameter GOT_25c = 5;
parameter GOT_30c = 6;
parameter GOT_35c = 7;
parameter GOT_40c = 8;
parameter GOT_45c = 9;
parameter GOT_50c = 10;
parameter RETURN_20c = 11;
parameter RETURN_15c = 12;
parameter RETURN_10c = 13;
parameter RETURN_5c = 14;
```

State register defined with sequential always block

```
always @ (posedge clk or negedge reset)
  if (!reset)    state <= IDLE;
  else           state <= next;
```

两段式FSM（输出组合电路）

Next-state logic within a combinational always block

```
always @ (state or N or D or Q) begin

   case (state)
     IDLE:      if (Q) next = GOT_25c;
           else if (D) next = GOT_10c;
             else if (N) next = GOT_5c;
             else next = IDLE;

     GOT_5c:    if (Q) next = GOT_30c;
           else if (D) next = GOT_15c;
             else if (N) next = GOT_10c;
             else next = GOT_5c;

     GOT_10c:   if (Q) next = GOT_35c;
           else if (D) next = GOT_20c;
             else if (N) next = GOT_15c;
             else next = GOT_10c;

     GOT_15c:   if (Q) next = GOT_40c;
           else if (D) next = GOT_25c;
             else if (N) next = GOT_20c;
             else next = GOT_15c;

     GOT_20c:   if (Q) next = GOT_45c;
           else if (D) next = GOT_30c;
             else if (N) next = GOT_25c;
             else next = GOT_20c;

     GOT_25c:   if (Q) next = GOT_50c;
           else if (D) next = GOT_35c;
             else if (N) next = GOT_30c;
             else next = GOT_25c;

       GOT_30c:  next = IDLE;
       GOT_35c:  next = RETURN_5c;
       GOT_40c:  next = RETURN_10c;
       GOT_45c:  next = RETURN_15c;
       GOT_50c:  next = RETURN_20c;

       RETURN_20c:  next = RETURN_10c;
       RETURN_15c:  next = RETURN_5c;
       RETURN_10c:  next = IDLE;
       RETURN_5c:   next = IDLE;

       default: next = IDLE;
     endcase
   end
```

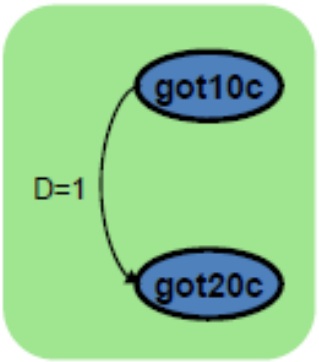
Combinational output assignment

```
  assign DC = (state == GOT_30c || state == GOT_35c ||
               state == GOT_40c || state == GOT_45c ||
               state == GOT_50c);
  assign DN = (state == RETURN_5c);
  assign DD = (state == RETURN_20c || state == RETURN_15c ||
               state == RETURN_10c);
endmodule
```

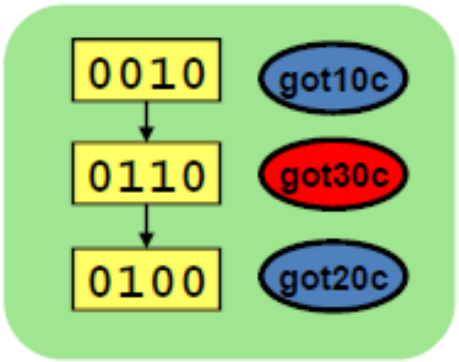
FSM Output Glitching

- FSM state bits may not transition at precisely the same time
- Combinational logic for outputs may contain hazards
- Result: your FSM outputs may glitch!

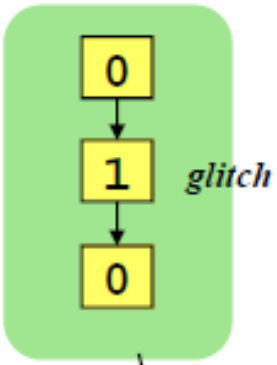
during this state transition...



...the state registers may transtion like this...

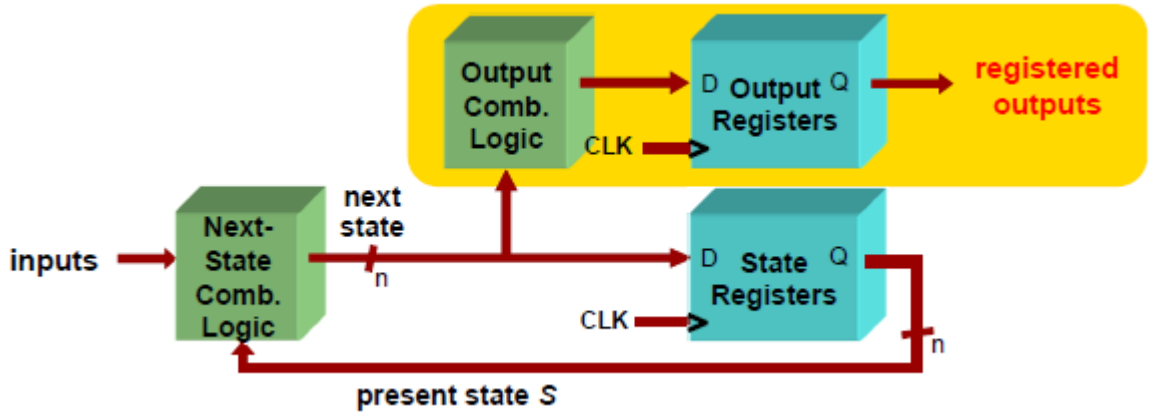


*...causing the DC output to **glitch** like this!*




```
assign DC = (state == GOT_30c || state == GOT_35c ||
             state == GOT_40c || state == GOT_45c ||
             state == GOT_50c);
```

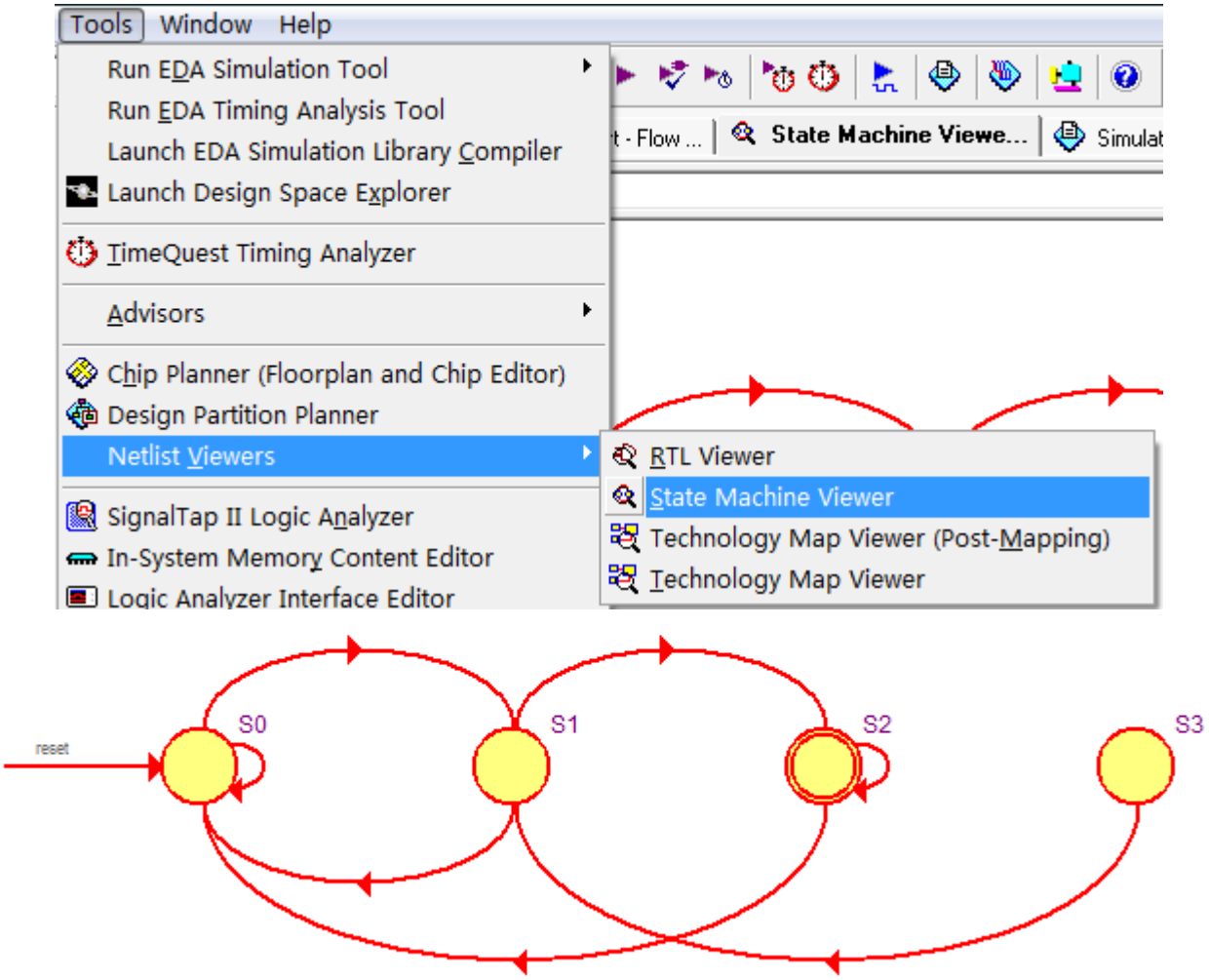
三段式FSM（输出寄存器电路）



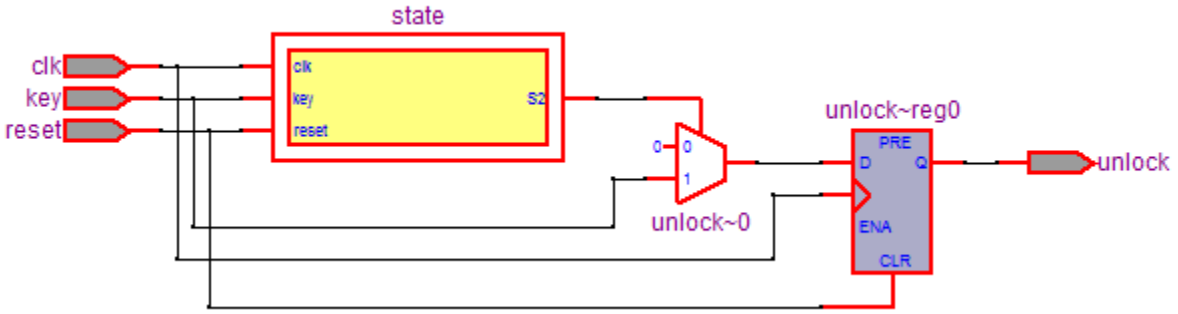
- Move output generation into the sequential always block
- Calculate outputs based on next state
- Delays outputs by one clock cycle. Problematic in some application.

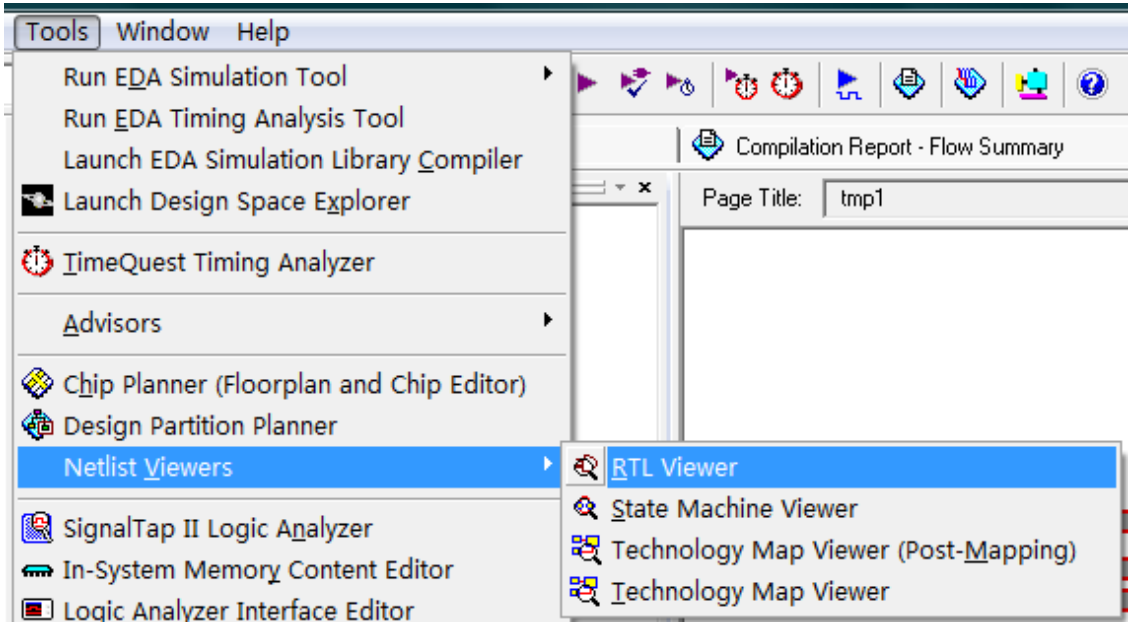
```
reg DC,DN,DD;

 // Sequential always block for state assignment
 always @ (posedge clk or negedge reset) begin
   if (!reset)   state <= IDLE;
   else if (clk) state <= next;

   DC <= (next == GOT_30c || next == GOT_35c ||
          next == GOT_40c || next == GOT_45c ||
          next == GOT_50c);
   DN <= (next == RETURN_5c);
   DD <= (next == RETURN_20c || next == RETURN_15c ||
          next == RETURN_10c);
 end
```

状态图


Tools
Window
Help
Run EDA Simulation Tool
Run EDA Timing Analysis Tool
Launch EDA Simulation Library Compiler
Launch Design Space Explorer
TimeQuest Timing Analyzer
Advisors
Chip Planner (Floorplan and Chip Editor)
Design Partition Planner
Netlist Viewers
SignalTap II Logic Analyzer
In-System Memory Content Editor
Logic Analyzer Interface Editor
RTL Viewer
State Machine Viewer
Technology Map Viewer (Post-Mapping)
Technology Map Viewer
State Machine Viewe...
reset
S0
S1
S2
S3

电路结构图


state
clk
key
reset
clk
key
reset
S2
0
1
unlock~0
unlock~reg0
PRE
D
Q
ENA
CLR
unlock


Tools
Window
Help
Run EDA Simulation Tool
Run EDA Timing Analysis Tool
Launch EDA Simulation Library Compiler
Launch Design Space Explorer
TimeQuest Timing Analyzer
Advisors
Chip Planner (Floorplan and Chip Editor)
Design Partition Planner
Netlist Viewers
SignalTap II Logic Analyzer
In-System Memory Content Editor
Logic Analyzer Interface Editor
RTL Viewer
State Machine Viewer
Technology Map Viewer (Post-Mapping)
Technology Map Viewer
Compilation Report - Flow Summary
Page Title:
tmp1

EDA实验二内容

Vending machine

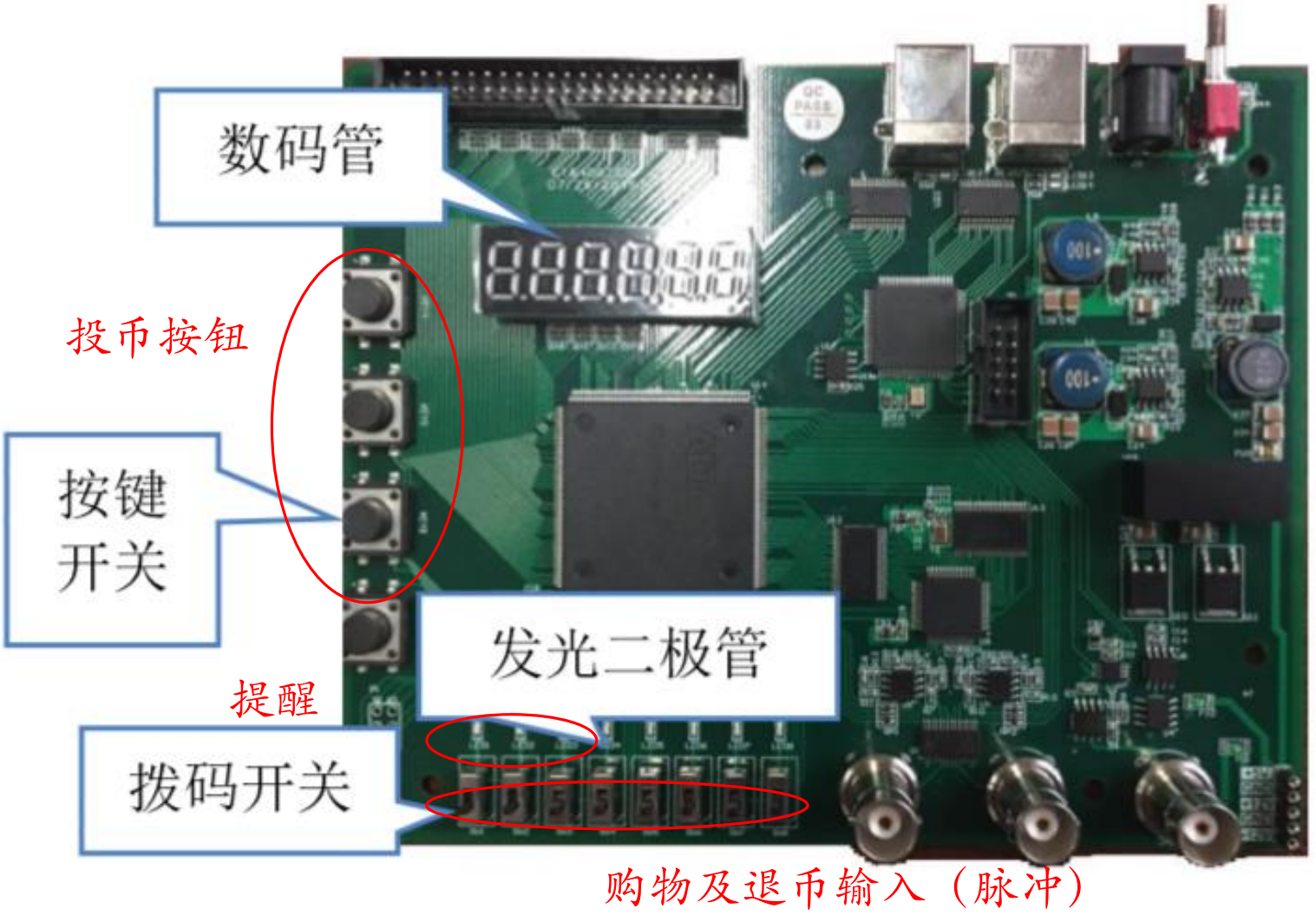
利用实验板上的拨码开关和按键开关模拟投币、购物和退币输入，用发光二极管模拟各种提示信息，用数码管显示余额，实现一个自动售货机内部控制电路。要求满足如下规格:

1）可接受5角、1元和5元的投币，每次购买允许投入多种不同币值的钱币；用3只数码管显示当前投币金额，如055表示已投币5.5元;

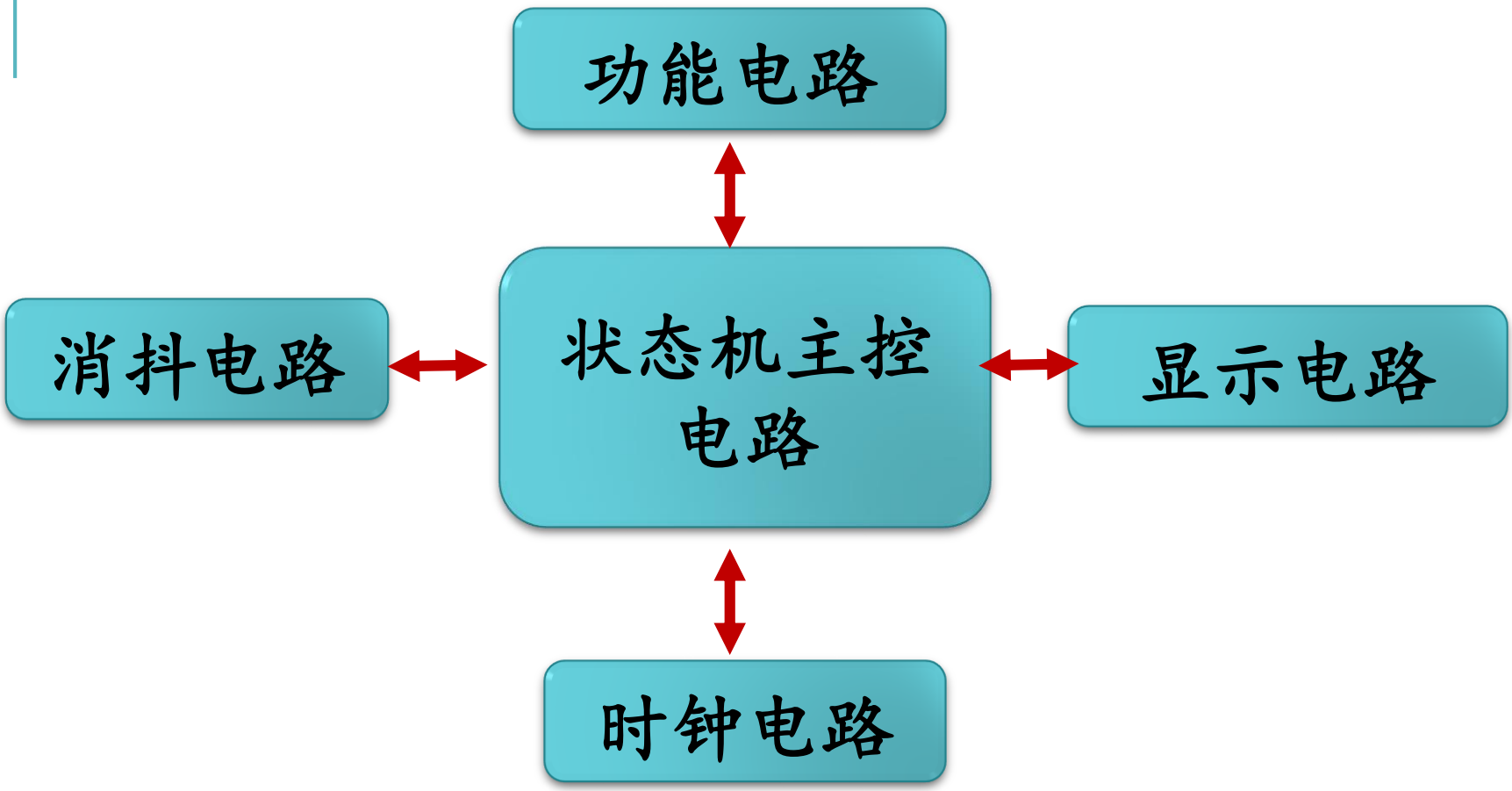
2）可售出价格分别为1.5元和2.5元的商品，假设用户每次购买时只选择单件、一种商品；允许用户多次购买商品，每次购买后，可以进行补充投币；

3）选择购买商品后，如果投币金额不足，则提醒；否则，售出相应的商品，并提醒用户取走商品;

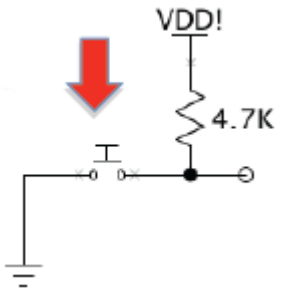
4）若用户选择退币，则退回余下的钱，并提醒用户取钱。

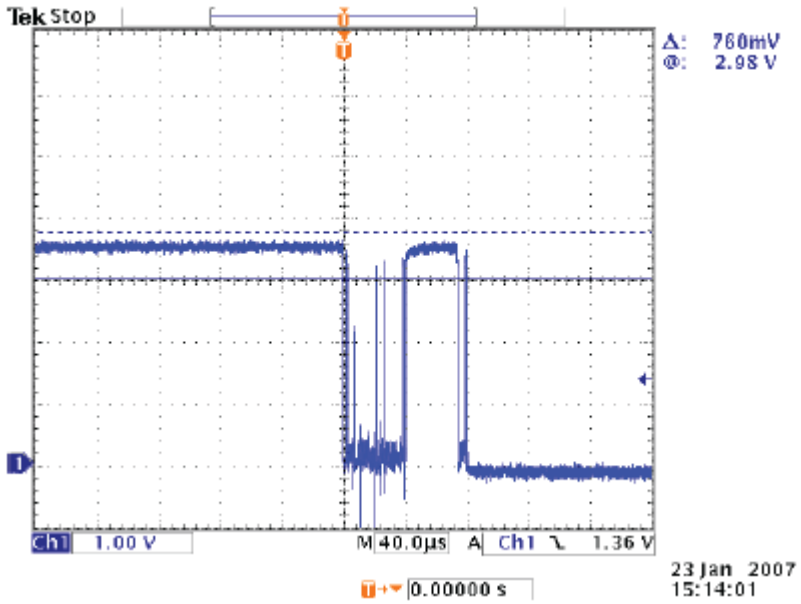


实验板上有40MHz的时钟信号，对应FPGA引脚号为PIN_152，自动售货机的工作时钟及数码管循环扫描显示的时钟可由该40MHz分频得到。


功能电路
消抖电路
状态机主控电路
显示电路
时钟电路

消抖电路






```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
// DELAY = .01 sec with a 27Mhz clock
module debounce #(parameter DELAY=270000-1)
                 (input reset, clock, bouncey,
                  output reg steady);

   reg [18:0] count;
   reg old;

   always @(posedge clock)
     if (reset)    // return to known state
       begin
         count <= 0;
         old <= bouncey;
         steady <= bouncey;
       end
     else if (bouncey != old)  // input changed
       begin
         old <= bouncey;
         count <= 0;
       end
     else if (count == DELAY) // stable!
         steady <= old;
     else                     // waiting…
       count <= count+1;

endmodule
```