目录

1 组合逻辑

1.1 组合逻辑一般问题

给出一个组合逻辑电路，要求分析逻辑功能。所谓组合逻辑电路的分析，就是找出给定逻辑电路输出和输入之间的关系，并指出电路的逻辑功能。
分析过程一般按下列步骤进行：
1：根据给定的逻辑电路，从输入端开始，逐级推导出输出端的逻辑函数表达式。
2：根据输出函数表达式列出真值表；
3：用文字概括处电路的逻辑功能；

1.2 设计一位加器

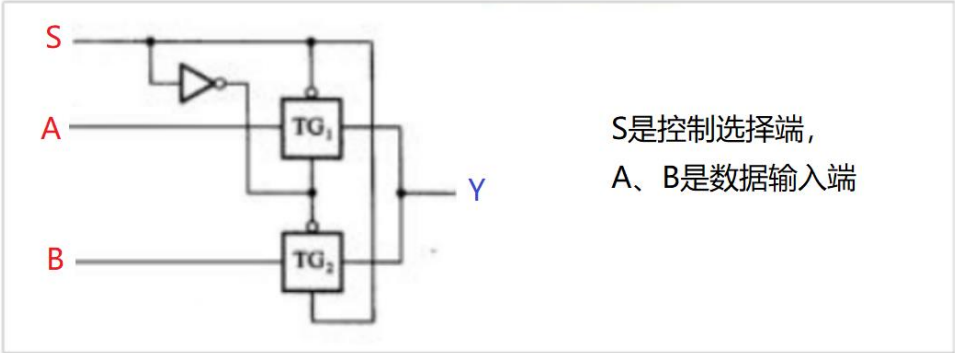
```
module( input clk,
    input a,
    input b,
    input cin,
    output s,
    output cout
    );
always@(posedge clk)begin
    {cout,s} <= a + b + cin;
end
endmodule
```

1.3 画出 NOT、NAND、NOR 的符号

真值表 ， 还有 transistor level （晶体管级）的电路；《数字电子技术基础（第五版）》117 页-134 页

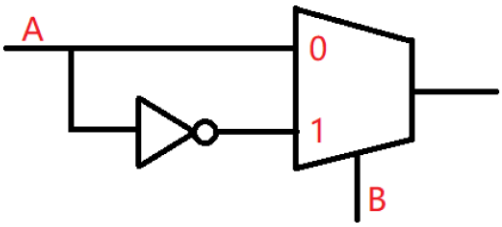
1.4 画出二选一选择器的 CMOS 图

可以用传输管和反相器搭建



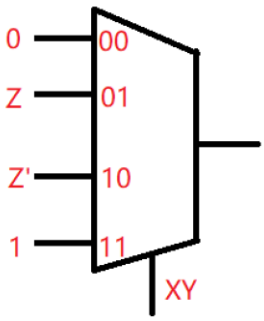
Y=SA+S’ B 利用与非门和反相器，进行变换后 Y=((SA)’ *(S’ B))’，三个与非门，一个反相器。

1.5 用一个二选一 mux 和一个 inv 实现异或



1.6 利用 4 选 1 实现 F(x,y,z) = xz+yz’

F(x,y,z) = x yz+xy’ z+xyz’ +x’ yz’ = x’ y’ 0+x’ yz’ +xy’ z+xy1
Y=A’ B’ D0+A’ BD1+AB’ D2+ABD3
所以 D0=0 ，D1=z ’，D2=z ，D3=1



1.7 与非门和反相器搭建电路

AB+CD+E = ((AB)′ (CD)′ E′)′ :三个两输入与非门，一个三输入与非门
Y=AB+C = ((AB)′ C′)′ :一个反相器，两个两输入与非门
Y=AB+CD=((AB)′ (CD)′)′ :三个两输入与非门

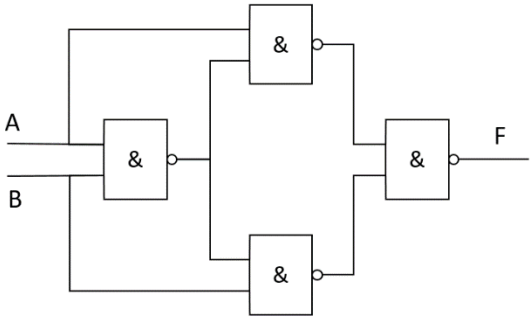
1.8 与非门设计全加器

考察与非门做异或门

$A \oplus B = AB' + A'B = AB' + AA' + BB' + A'B$

$= A(A' + B') + B(A' + B') = A(AB)' + B(AB)' = ((A(AB)')' * (B(AB)')')'$

即：4 个与非门即可实现异或



一位全加器的表达式如下：

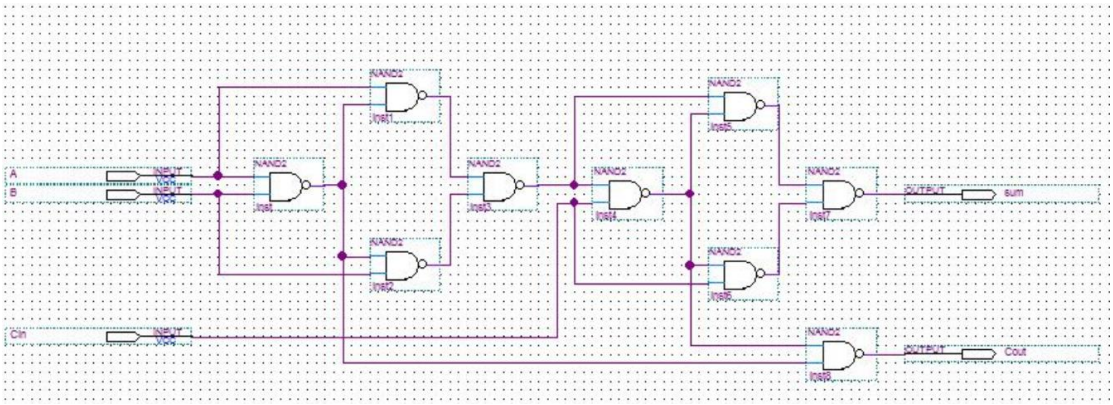
$$S_i = A_i \oplus B_i \oplus C_{i-1}$$

$$C_i = A_i B_i + C_{i-1}(A_i + B_i)$$

第二个表达式也可用一个异或门来代替或门对其中两个输入信号进行求和：因为修改成异或后，因为 AB 的存在，化简结果仍是上文第二个表达式

$$C_i = A_i B_i + C_{i-1}(A_i \oplus B_i)$$

即可使用与非门搭建异或门，需要 9 个与非门搭建一位全加器：



1.9 少数服从多数

A,B,C,D,E 进行投票,多数服从少数,输出是 F 也就是如果 A,B,C,D,E 中 1 的个数比 0 多,那么 F 输出为 1,否则 F 为 0, 用与非门实现,输入数目没有限制？（与非-与非形式）先画出卡诺图来化简，化成与或形式，再两次取反便可。

Y = ABC + ABD + ABE + ACD + ACE + ADE + BCD + BCE + BDE + CDE

使用最小项将多数大于少数的表达式或起来

1.10 选择门实现逻辑

为了实现逻辑 Y=A’B+AB’+CD，请选用以下逻辑中的一种，并说明为什么？

1）INV 2 ）AND 3 ）OR 4 ）NAND 5 ）NOR 6 ）XOR

答案： NAND （未知）

电路性能稳定、抗干扰能力比较强、产品类型比较多。
可能是因为 CMOS 逻辑都是非结构，与结构相对于或结构速度快，因为与门电容串联小，充放电速度快

1.11 乘法次数计算

x^4 + ax^3 + x^2 + cx + d 最少需要做几次乘法？
(((x+ax)+1)x+c)x+d
最少需要三次乘法即可,这是一种有效的加速表达式求解的方法

1.12 搭建计数器

用常用的逻辑门搭一个 3 位宽的计数器；
使用 3 个 D 触发器级联即可，下一级触发器的时钟为上一级触发器的 Q’,触发器自身的 D 与 Q’相连接；

1.13 最小式化简

化简 F(A,B,C,D)= m(1,3,4,5,10,11,12,13,14,15)的和。
将最小式带入卡卡诺图当中，画圈化简

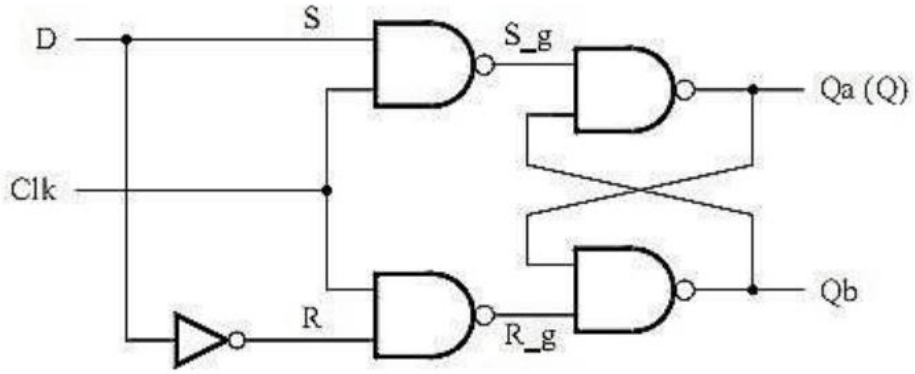
	00	01	11	10
00	0	1	1	0
01	1	1	1	0
11	1	0	1	1
10	0	0	1	1

F(A,B,C,D) = AC + BC’ +A’ B’ D

2 时序逻辑

2.1 画出一种 CMOS 的 D 锁存器的电路图和版图

D 锁存器（可以用与非门搭建，或者用与门和或非门搭建）



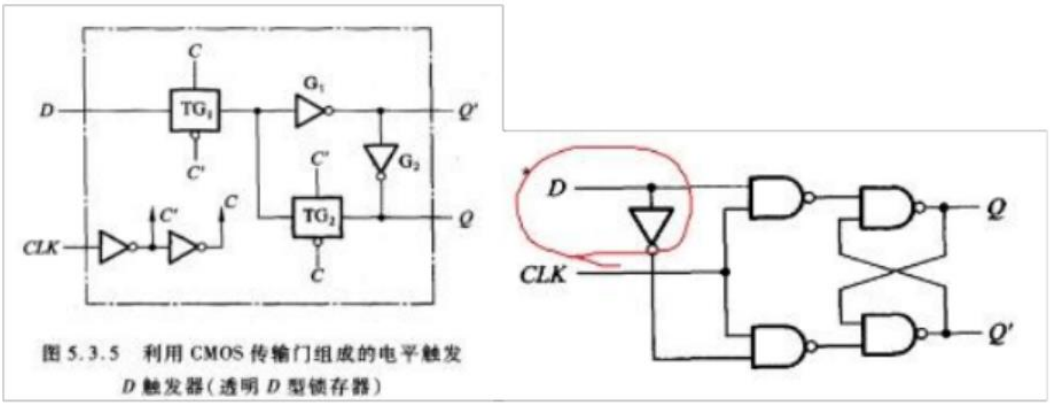


图 5.3.5 利用 CMOS 传输门组成的电平触发 D 触发器（透明 D 型锁存器）

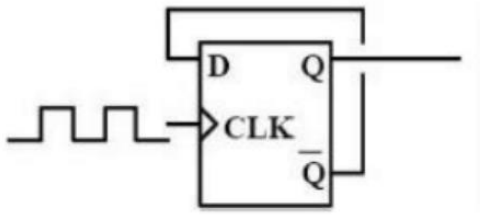
也可以将右图中的与非门和反相器用 CMOS 电路画出来。

2.2 用 D 触发器做个二分频的电路并画出逻辑电路

```
module div2 (clk,rst,clk_out);
    input clk,rst;
    output reg clk_out;
always@(posedge clk) begin
    if(!rst)
        clk_out <=0;
    else
        clk_out <=~ clk_out;
end
endmodule
```

现实 工程设计中一般不采用这样的方式来设计，二分频一般通过 DCM 来实现。

通过 DCM 得到的分频信号没有相位差。或者是从 Q 端引出加一个反相器，相当于过一拍取反一次。如果是四分频，就用两个触发器，过两拍取反一次，以此类推



2.3 可预置初值的 7 进制循环计数器

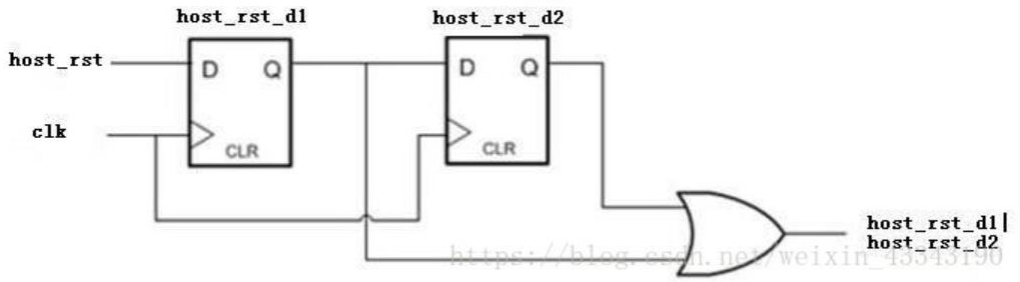
```
module counter7(clk,rst,load,data,cout);//同步复位，同步置数
 input clk,rst,load;
 input [2:0] data;
 output reg [2:0] cout;
 always@(posedge clk) begin
    if(!rst)
       cout<=3 'd0;
    else if(load)
       cout<=data;
    else if(cout>=3 'd6)
       cout<=3 'd0;
    else
       cout<=cout+3 'd1;
 end
endmodule
```

```
module counter7(clk,rst,load,data,cout);//异步复位，同步置数
 input clk,rst,load;
 input [2:0] data;
 output reg [2:0] cout;
 always@(posedge clk or negedge rst)begin
    if(!rst)
       cout<=3 'd0;
    else if(load)
       cout<=data;
    else if(cout>=3 'd6)
       cout<=3 'd0;
    else
       cout<=cout+3 'd1;
    end
endmodule
```

2.4 消除一个 glitch 毛刺

用 Verilog 或 VHDL 写一段代码 ， 实现消除一个 glitch （毛刺）？
将传输过来的信号经过两级触发器就可以消除毛刺。 （这种方式消除毛刺是需要满足一定条件的，并不能保证一定可以消除，毛刺宽度需要小于一个时钟周期）
该电路针对负边沿的毛刺，对于双边沿的毛刺怎么解决？

滤掉小于1个周期glitch的原理图如下：




```
//使用两级 D 触发器实现
//消除大于一个周期，小于两个周期的毛刺，需要 3 级 D 触发器实现
module digital_filter_(clk_in,rst,host_rst,host_rst_filter);
 input clk_in;
 input rst;
 input host_rst;
 output host_rst_filter;
 reg host_rst_d1;
 reg host_rst_d2;
 always@(posedge clk_in or negedge rst) begin
    if(~rst) begin
        host_rst_d1 <= 1'b1;
        host_rst_d2 <= 1'b1;
    end
    else begin
        host_rst_d1 <= host_rst;
        host_rst_d2 <= host_rst_d1;
    end
 end
 assign host_rst_filter = host_rst_d1 | host_rst_d2;
endmodule
```

2.5 实现分频电路

实现三分频电路， 3/2 分频电路等（偶数倍分频 奇数倍分频）

图 2 是 3 分频电路，用 JK-FF 实现 3 分频很方便，不需要附加任何逻辑电路就能实现同步计数分频。 但用 D-FF 实现 3 分频时， 必须附加译码反馈电路， 如图 2 所示的译码复位电路，强制计数状态返回到初始全零状态，就是用 NOR 门电路把 Q2，Q1=－11B‖的状态译码产生－H‖ 电平复位脉冲，强迫 FF1 和 FF2 同时瞬间(在下一时钟输入 Fi 的脉冲到来之前)复零，于是 Q2，Q1=－11B‖状态仅瞬间作为 －毛刺 ‖存在而不影响分频的周期，这种 －毛刺 ‖仅在 Q1 中存在， 实用中可能会造成错误， 应当附加时钟同步电路或阻容低通滤波电路来滤除， 或者仅使用 Q2 作为输出。 D-FF 的 3 分频，还可以用 AND 门对 Q2，Q1 译码来实现返回复零。

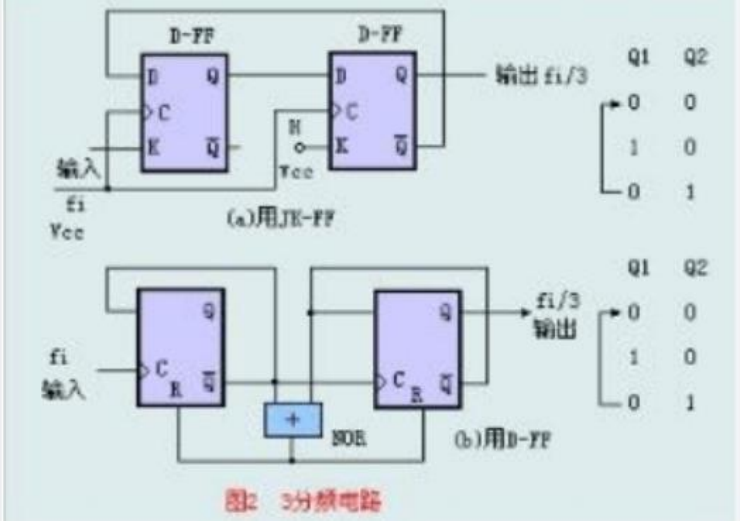
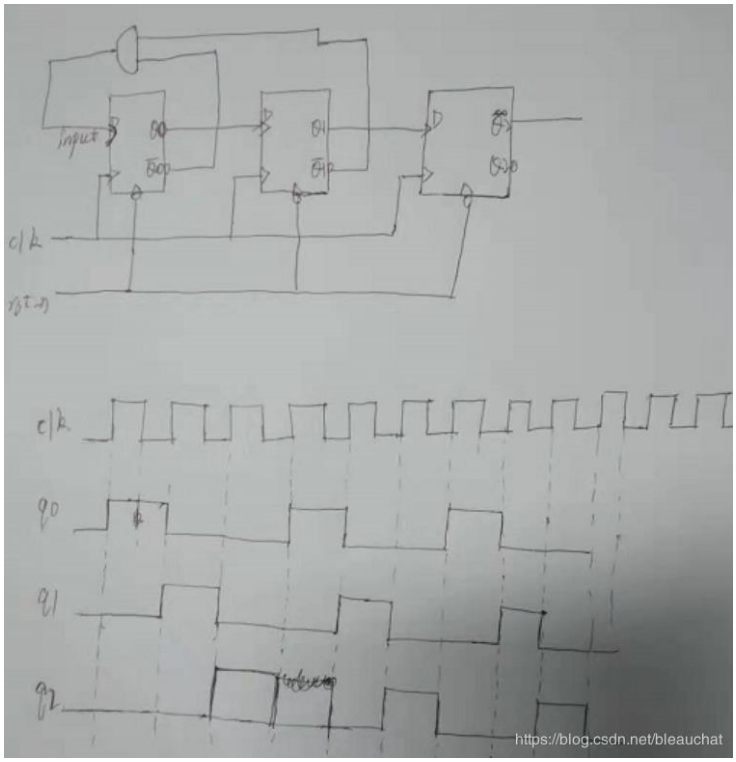
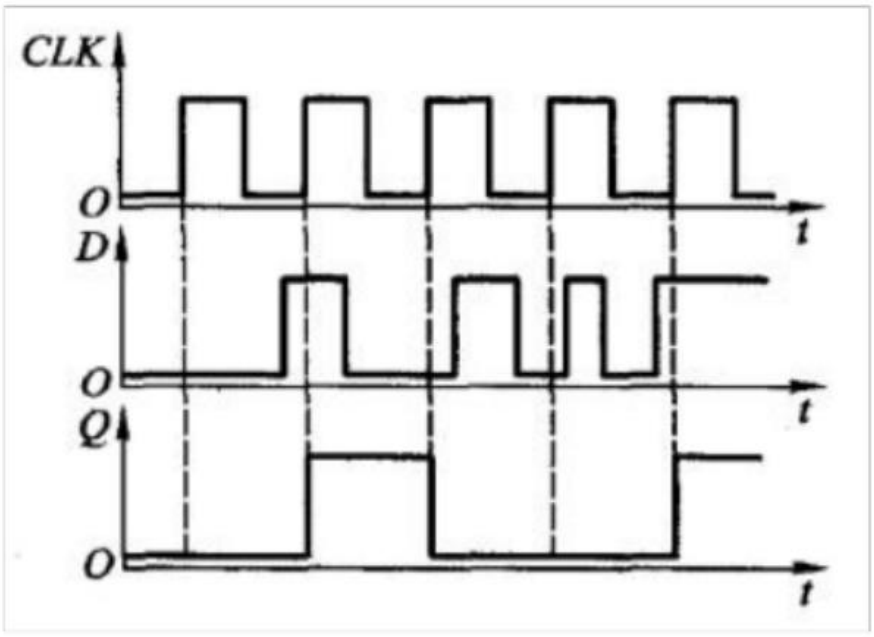


图2 3分频电路



若需要实现 50%占空比，需要一个下降沿触发器，实现将 3 分频输出延时一拍，再与原输出相与即可

2.6 用波形描述触发器的功能



2.7 传输门和反相器搭建边沿触发器 DFF

－通过级联两个 D 锁存器组成
时钟加反相器 buff，增强时钟驱动能力

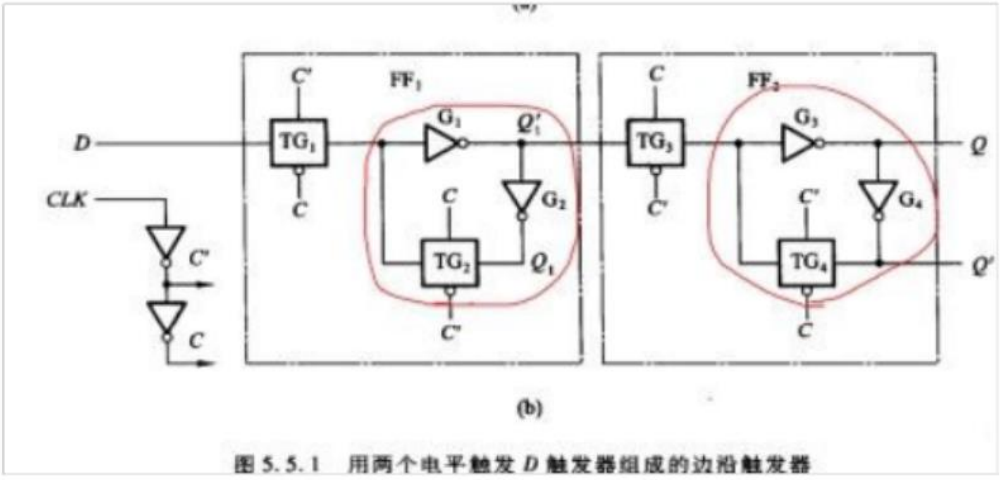
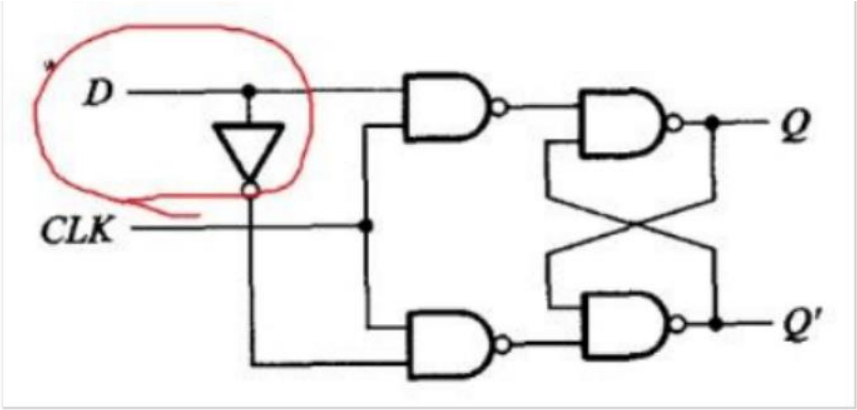


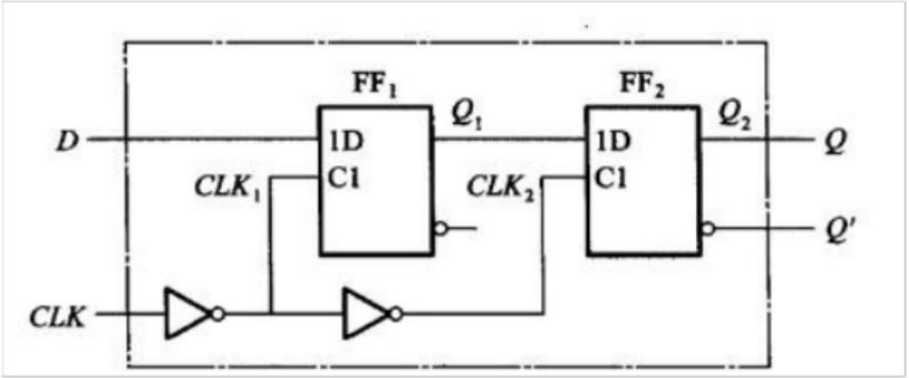
图 5.5.1 用两个电平触发 D 触发器组成的边沿触发器

通过级联两个 D 锁存器组成

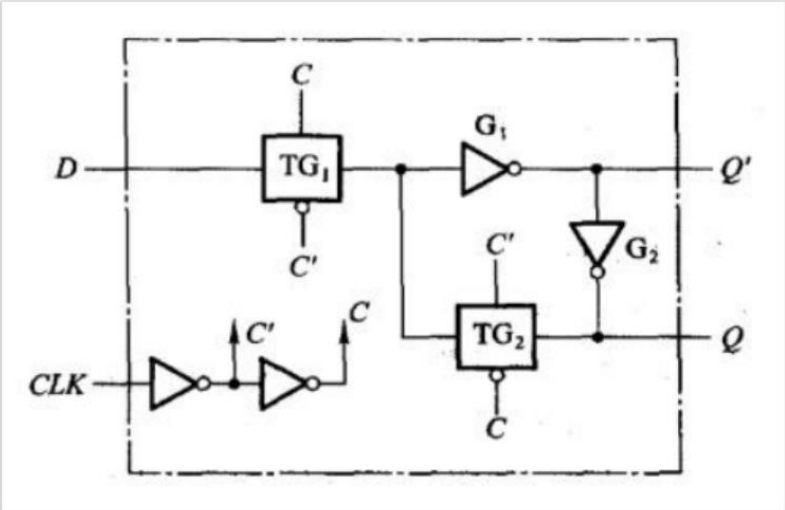
2.8 逻辑门搭建触发器



电平触发的 D 触发器（ D 锁存器）牢记！



2.9 画出锁存器的电路图



或者是利用前面与非门搭的 D 锁存器实现

2.10 D 触发器搭建计数器

用 D 触发器做个 4 进制的计数

使用两个 D 触发器，每个触发器的 Q’ 与 D 相连接，第一个 D 触发器的 Q’ 接第二个触发器的 CLK ，两个 D 触发器的 Q’ 即为计数器的值

也可以按照时序逻辑电路的设计步骤来：
写出状态转换表
寄存器的个数确定
状态编码
卡诺图化简
状态方程，驱动方程等

2.11 实现 N 位扭环计数器

约翰逊 Johnson 计数器又称扭环计数器,是一种用 n 位触发器来表示 2n 个状态的计数器。它与环形计数器不同,后者用 n 位触发器仅可表示 n 个状态。2n 进制计数器 n 为触发器的个数有 2n 个状态。若以四位二进制计数器为例,它可表示 16 个状态。但由于 8421 码每组代码之间可能有二位度或二位以上的二进制代码发生改变,这在计数器中特别回是异步计数器中就有可能产生错误的译码信号,从而造成永久性的错误。而约翰逊计数答器的状态表中,相邻两组代码只可能有一位二进制代码不同,故在计数过程中不会产生错误的译码信号。鉴于上述优点,约翰逊计数器在同步计数器中应用比较广泛。

扭环计数器有 2N 个状态，其中有无效状态，

```
module johnson#(
 parameter N = 4
)(
 input clk,
 input rst_n,
 output reg [N-1:0] q
);
 always@(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)
        q <= {N{1'b0}};
    else
        q <= {~q[0],q[N-1,1]}
 end
endmodule
```

2.12 设计 5 分频逻辑即 7 分频 50%占空比电路

```
//5 分频，占空比不是 50%
module div_5(clk,rst_n,clk_out);
 input clk,rst_n;
 output reg clk_out;
 reg [3:0] count;
 always@(posedge clk or negedge rst_n) begin
    if(!rst_n)begin
       count <= 0;
       clk_out <= 0;
    end
    else if(count == 3'd4) begin
       count <= 0;
       clk_out <= 1'b1;
    end
    else begin
       count<=count+1;
       clk_out <= 1'b0;
    end
 end
endmodule
```

```
//实现 7 倍分频且占空比为 50% 的情况
module div_7_55(
 //system signals
 input clk ,
 input rst_n ,
 output clk_out
);
reg clk_pose;
reg clk_nege;
reg [3:0] count_pose ;
reg [3:0] count_nege ;
//上升沿计数
always @ (posedge clk or negedge rst_n) begin
 if(rst_n == 1'b0)
    count_pose <= 4'd0;
 else if(count_pose == 4'd6)
    count_pose <= 4'd0;
 else
    count_pose <= count_pose + 1'b1;
end
//下降沿计数
```

```
always @ (negedge clk or negedge rst_n) begin
 if(rst_n == 1'b0)
    count_nege <= 4'd0;
 else if(count_nege == 4'd6)
    count_nege <= 4'd0;
 else
    count_nege <= count_nege + 1'b1;
end
//上升沿分频
always @ (posedge clk or negedge rst_n) begin
 if(rst_n == 1'b0)
    clk_pose <= 1'b0;
 else if(count_pose < 4'd3)
    clk_pose <= 1'b0;
 else
    clk_pose <= 1'b1;
end
//下降沿分频
always @ (negedge clk or negedge rst_n) begin
 if(rst_n == 1'b0)
    clk_nege <= 1'b0;
 else if(count_nege < 4'd3)
    clk_nege <= 1'b0;
 else
    clk_nege <= 1'b1;
end
//上升沿分频与下降沿分频相与，得 50%占空比
assign clk_out = clk_nege & clk_pose;
endmodule
```

2.13 设计一个计算连续 Leading Zeros 个数的电路

计算前导零的个数，输入 8-bit，输出 4-bit。
00001000 0100
00100010 0010
10001000 0000
可以 parameterize 你的设计吗？其 hardware 是什么样子的？
https://zhuanlan.zhihu.com/p/84589627

```
module method1(
    input[7:0] din ,
    input clk ,
    input rst_n ,
    output reg [4:0] dout
);
```

```
always@(posedgeclk or negedge rst_n) begin
    if(!rst_n)begin
        dout<= 0 ;
    end
    else begin
         if(din[7])dout <= 0;
         else if(din[6]) dout <= 1;
         else if(din[5]) dout <= 2;
         else if(din[4]) dout <= 3;
         else if(din[3]) dout <= 4;
         else if(din[2]) dout <= 5;
         else if(din[1]) dout <= 6;
         else if(din[0]) dout <= 7;
         else           dout <= 8;
    end
end
endmodule
```

可以发现上述代码使用了 7 级 MUX 级联，可以在单周期内完成运算。但是，不易参数化。

```
module method1#(
   parameter N = 8,
   parameter M = 4
)(
    input[N-1:0] din ,
    inputclk ,
    inputrst_n ,
    outputreg [M-1:0] dout
    );
   reg [N-1:0] dout_1;
   reg [M-1:0] dout_2 [N-1:0];
   genvar i ;
   generate
      for(i=0;i<N;i=i+1) begin
          always@(*) begin
              dout_1[i]=(din[N-1:N-1-i]==1'b0)?1'b1:1'b0;
              dout_2[i] =  (i==0)?dout_1[i]:dout_1[i]+ dout_2[i-
1];
          end
      end
   endgenerate
    always@(posedgeclk or negedge rst_n) begin
       if(!rst_n)begin
               dout<= 0 ;
       end
```

```
   else begin
      dout <= dout_2[N-1];
   end
   end
endmodule
```

该代码实现了参数化的需求。使用了 7 级异或级联，可以在单周期内完成运算。

2.14 设计地址生成器

设计地址生成器。要求依次输出以下序列：
0, 8, 2, 10, 4, 12, 6, 14, 1, 9, 3, 11, 5, 13, 7, 15,
16, 24, 18, 26, ……, 31,
32, 40, 34, 42, ……, 47,
48, 56, 50, 58, ……, 63,
64, 72, 66, 76, ……, 79

//将数据每 16 个数分为一组，再将一族中相同规律的数字总结起来

```
module addr_counter(
 //system signals
 input clk ,
 input rst_n ,
 output reg [7:0] counter
);
 reg [3:0] count_0;
 reg [7:0] count_1;
 always @ (posedge clk or negedge rst_n) begin
    if(rst_n == 1'b0)begin
       count_0 <= 4'd0;
       count_1 <= 8'd0;
    end
    else if(count_1 < 8'd80)begin
       count_0 <= count_0 + 1'b1;
       count_1 <= count_1 + 1'b1;
    end
 end
 always @ (posedge clk or negedge rst_n) begin
    if(rst_n == 1'b0)
       counter <= 8'd0;
    else begin
        case (count_0)
           0,2,4,6,9,11,13,15: counter <= count_1;
           1,3,5,7: counter <= count_1 + 8'd7;
```

```
            8,10,12,14: counter <= count_1 - 8'd7;
            default: counter <= 8'd0;
        endcase
    end
 end
endmodule
```

2.15 用 Verilog 实现串并转换

lsb 优先
msb 优先

```
module Deserialize(
 input clk,
 input rst_n,
 input data_i,
 output reg [7:0] data_o
);
 //先补充高位
 /*always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)
       data_o <= 8'b0;
    else
       data_o <= {data_o[6:0], data_i};
 end*/
 //先补充低位
 always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)
       data_o <= 8'b0;
    else
       data_o <= {data_i,data_o[7:1]};
 end
endmodule
```

并行转串行数据输出：采用计数方法，将并行的数据的总数先表示出来，然后发送一位数据减一，后面的接收的这样表示：data_out <= data[cnt];//cnt 表示计数器

2.16 异步双端口 RAM

用 Verilog 实现一个异步双端口 ram，深度 16，位宽 8bit。A 口读出，B 口写入。支持片选，读写请求，要求代码可综合。

RAM 读写冲突解决方法
https://blog.csdn.net/weixin_45985643/article/details/104652439

```
module Dual_Port_Sram
 #(
```

```
 parameter ADDR_WIDTH = 4,
 parameter DATA_WIDTH = 8,
 parameter DATA_DEPTH = 1 << ADDR_WIDTH
 )
 (
 input clka,
 input clkb,
 input rst_n,
 input csen_n,
 //Port A Signal
 input [ADDR_WIDTH-1:0] addra,
 output reg [DATA_WIDTH-1:0] data_a,
 input rdena_n,
 //Port B Signal
 input [ADDR_WIDTH-1:0] addrb,
 input wrenb_n,
 input [DATA_WIDTH-1:0] data_b
 );
 integer i;
 reg [DATA_WIDTH-1:0] register[DATA_DEPTH-1:0];
 always @(posedge clkb or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        for(i = 0; i < DATA_DEPTH; i = i + 1)
        register[i] <= 'b0000_1111;
    end
    else if(wrenb_n == 1'b0 && csen_n == 1'b0)
       register[addrb] <= data_b;
 end
 always @(posedge clka or negedge rst_n)begin
    if(rst_n == 1'b0)begin
       data_a <= 0;
    end
    else if(rdena_n == 1'b0 && csen_n == 1'b0 && wrenb_n == 1'b0
&& addra
       data_a <= data_b;
    else if(rdena_n == 1'b0 && csen_n == 1'b0 && wrenb_n == 1'b1)
       data_a <= register[addra];
    else
       data_a <= data_a;
 end
endmodule
```

2.17 实现异步复位同步释放电路。

```
module Sys_Rst(
 input clk,
 input rst_n,
 output sys_rst
);
 reg rst_r0;
 reg rst_r1;
 //异步复位将复位状态存储起来，再经过同步状态输出
 always @(posedge clk or negedge rst_n)begin
    if(rst)begin
        rst_r0 <= 1'b0;
        rst_r1 <= 1'b0;
    end
    else begin
        rst_r0 <= 1'b1;
        rst_r1 <= rst_r0;
    end
 end
 assign sys_rst = rst_r1;
endmodule
```

2.18 按键消抖

用 Verilog 实现按键抖动消除电路，抖动小于 15ms，输入时钟 12MHz。

http://www.yasheng.fun/posts/3675679110/

```
module debounce(
  input      clk,//12Mhz
  input      rst_n,
  input      key_in,
  output      key_flag
);
parameter  JITTER  =  240;//12Mhz / (1/20ms)
reg  [1:0]   key_r;
wire       change;
reg  [15:0]   delay_cnt;
always @(posedge clk or negedge rst_n)begin
  if(rst_n == 1'b0)begin
    key_r <= 0;
  end
  else begin
    key_r <= {key_r[0],key_in};
```

```
  end
end
assign  change = (~key_r[1] & key_r[0]) | (key_r[1] & ~key_r[0]);
always @(posedge clk or negedge rst_n)begin
  if(rst_n == 1'b0)begin
    delay_cnt <= 0;
  end
  else if(change == 1'b1)
    delay_cnt <= 0;
  else if(delay_cnt == JITTER)
    delay_cnt <= delay_cnt;
  else
    delay_cnt <= delay_cnt + 1;
end
assign  key_flag = ((delay_cnt == JITTER - 1) && (key_in ==
1'b1))? 1'b1:1'b0;
endmodule
```

2.19 实现一个同步 FIFO

用 Verilog 实现一个同步 FIFO，深度 16，数据位宽 8bit。

```
module fifo_syn #(
 parameter WIDTH = 8 ,
 parameter DEPTH = 256 //深度
)
 ( //system signals
 input clk ,
 input rst_n ,
 input fifo_wr ,
 input fifo_rd ,
 input [WIDTH - 1:0] data_in ,
 output reg [WIDTH - 1:0] data_out ,
 output full , output empty );
 reg [7 : 0] ram [DEPTH - 1'b1 : 0]; //开辟存储区
 reg [DEPTH-1 : 0] count; //计数
 reg [DEPTH-1 : 0] p_wr; //写指针
 reg [DEPTH-1 : 0] p_rd; //读指针 //简单状态机
 always @ (posedge clk or negedge rst_n) begin
    if(rst_n == 1'b0)begin
        count <= 0;
        p_wr <= 0;
        p_rd <= 0;
        data_out<= 8'd0;
    end
```

```
    else begin
        case ({fifo_wr,fifo_rd})
        2'b10 :begin
            if(~full)begin
                ram[p_wr] <= data_in;
                count <= count + 1'b1;
                p_wr <= p_wr + 1'b1;
            end
       end
        2'b01 :begin
            if(~empty)begin
                data_out <= ram[p_rd];
                count <= count - 1'b1;
                p_rd <= p_rd + 1'b1;
            end
        end
        2'b11 :begin
            if(empty)begin
                ram[p_wr] <= data_in;
                count <= count + 1'b1;
                p_wr <= p_wr + 1'b1;
        end
        else begin
            ram[p_wr] <= data_in;
            p_wr <= p_wr + 1'b1;
            data_out <= ram[p_rd];
            p_rd <= p_rd + 1'b1;
            count <= count;
        end
       end
       endcase
   end
 end
 //判断空满状态
 assign full = (count == 8'hff) ? 1 : 0;
 assign empty = (count == 8'd0) ? 1 : 0;
endmodule
```

2.20 时钟切换设计

方法 1（缺点：会有毛刺）

```
module Change_Clk_Source(
  input     clk1,
```

```
  input      clk0,
  input      select,
  input      rst_n,
  output      outclk
);
//-------------------------------------------------------

assign outclk = (clk1 & select) | (~select & clk0);

endmodule
```

方法 2（缺点：当 sel 信号出现再 clk 的下降沿，或者两个寄存器的下降沿重合的地方可能会产生亚稳态）

```
module clk_select2(
    input                                clk1,
    input                                clk2,
    input                                rst_n,
    input                                sel,
    output                               clk_out
);

reg ff1;
reg ff2;
always @(negedge clk1 or negedge rst_n) begin
    if(!rst_n) ff1 <= 1'b0;
    else begin
        ff1 <= ~ff2 & sel;
      end
    end
always @(negedge clk2 or negedge rst_n) begin
    if(!rst_n) ff2 <= 1'b0;
    else begin
        ff2 <= ~ff1 & ~sel;
      end
    end
assign clk_out = (ff1 & clk1) | (ff2 & clk2);
endmodule
```

方法 3（方法 2 的基础上添加的）

```
module clk_select3(

input        clk1,
input        clk2,
input        rst_n,
input        sel,
output       clk_out
```

```
);
reg ff1,ff1_d;
reg ff2,ff2_d;
always @(negedge clk1 or negedge rst_n) begin
    if(!rst_n)  {ff1,ff1_d} <= 2'b00;
    else begin
        ff1_d <= ~ff2 & sel;
        ff1 <= ff1_d;
      end
    end
always @(negedge clk2 or negedge rst_n) begin
    if(!rst_n) {ff2,ff2_d} <= 2'b00;
    else begin
        ff2_d <= ~ff1 & ~sel;
        ff2 <= ff2_d;
      end
    end
assign clk_out = (ff1 & clk1) | (ff2 & clk2);
endmodule
```

2.21 跨时钟域（慢到快）

```
reg     [1:0]   signal_r;
//----------------------------------------------------------

always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        signal_r <= 2'b00;
    end
    else begin
        signal_r <= {signal_r[0], signal_in};
    end
end
assign  signal_out = signal_r[1];
```

2.22 跨时钟域（快到慢）

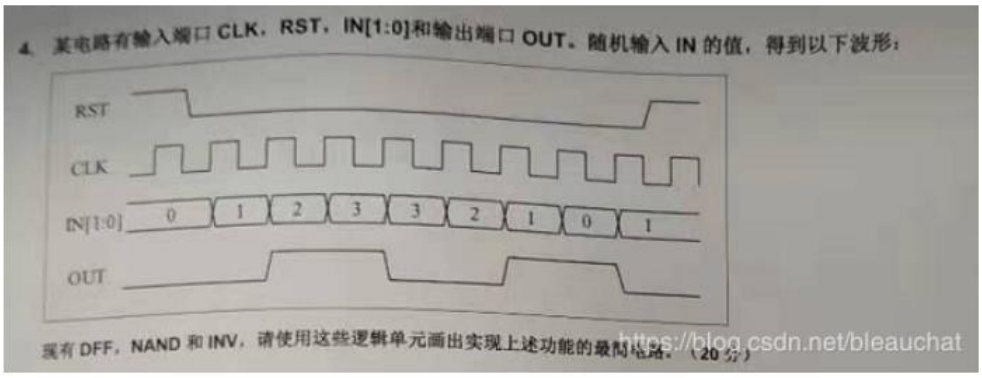
```
//Synchronous
module Sync_Pulse(
    input           clka,
    input           clkb,
    input           rst_n,
    input           pulse_ina,
    output          pulse_outb,
    output          signal_outb
```

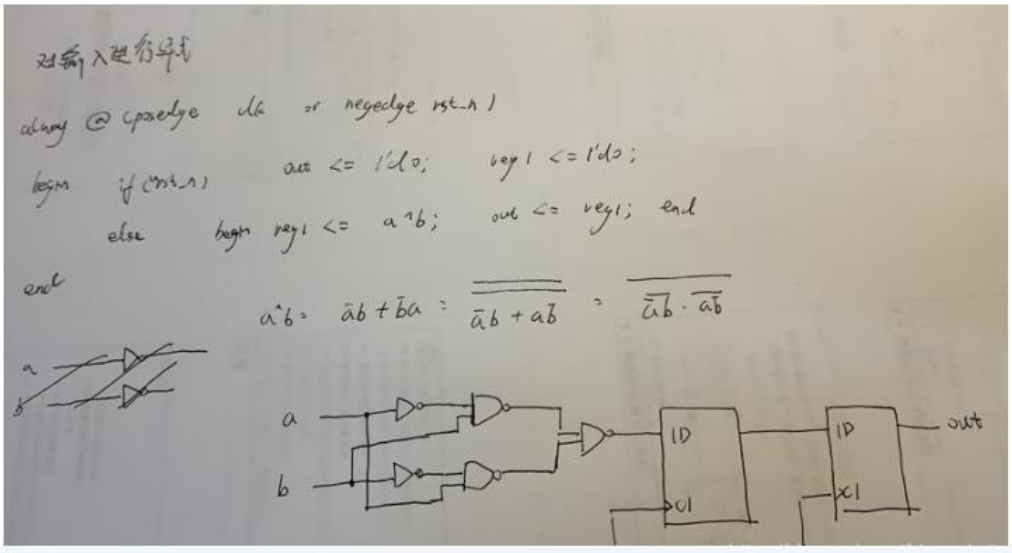
```
);

//-------------------------------------------------------
reg           signal_a;
reg           signal_b;
reg    [1:0]   signal_b_r;
reg    [1:0]   signal_a_r;

//-------------------------------------------------------
//在 clka 下，生成展宽信号 signal_a
always @(posedge clka or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        signal_a <= 1'b0;
    end
    else if(pulse_ina == 1'b1)begin
        signal_a <= 1'b1;
    end
    else if(signal_a_r[1] == 1'b1)
        signal_a <= 1'b0;
    else
        signal_a <= signal_a;
end

//-------------------------------------------------------
//在 clkb 下同步 signal_a
always @(posedge clkb or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        signal_b <= 1'b0;
    end
    else begin
        signal_b <= signal_a;
    end
end

//-------------------------------------------------------
//在 clkb 下生成脉冲信号和输出信号
always @(posedge clkb or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        signal_b_r <= 2'b00;
    end
    else begin
        signal_b_r <= {signal_b_r[0], signal_b};
    end
end
```

```
assign    pulse_outb = ~signal_b_r[1] & signal_b_r[0];
assign    signal_outb = signal_b_r[1];

//-----------------------------------------------------------
//在 clka 下采集 signal_b[1]，生成 signal_a_r[1]用于反馈拉低 signal_a
always @(posedge clka or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        signal_a_r <= 2'b00;
    end
    else begin
        signal_a_r <= {signal_a_r[0], signal_b_r[1]};
    end
end

endmodule
```

2.23 时序逻辑波形设计电路

Q = a’b+ab’,异或，加 D 触发器，注意时钟沿前一个时刻的输入值



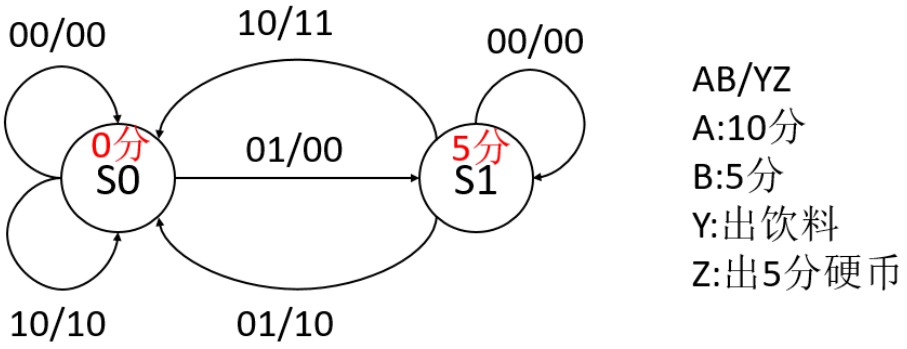


3 状态机

moore 状态机和 mealy 状态机可以相互转化，moore 状态多，mealy 状态少
moore 状态机的输出，最好使用时序逻辑，避免组合逻辑形成的毛刺
https://blog.csdn.net/Reborn_Lee/article/details/88918615

3.1 自动饮料售卖机

设计一个自动饮料售卖机，饮料 10 分钱，硬币有 5 分和 10 分两种，并考虑找零；
画出 fsm（有限状态机）
用 verilog 编程，语法要符合 FPGA 设计的要求
设计工程中可使用的工具及设计大致过程？
设计过程：
1、首先确定输入输出， A=1 表示投入 10 分，B=1 表示投入 5 分，Y=1 表示弹出饮料， Z=1
表示找零。
2、确定电路的状态， S0 表示没有进行投币， S1 表示已经有 5 分硬币。
3、画出状态转移图。




```
module sell(
 input clk,
 input rst_n,
 input a,
 input b,
 output y,
 output z
);
 parameter S0 = 1'b0,S1 = 1'b1;
 reg stage_current,stage_next;
 always@(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)
       state_current <= S0;
    else
       state_current <= stage_next;
```

```
end
always@(*)begin
    case(stage_current)
    S0:begin
        if(a == 1'b1 && b == 1'b0) begin
            stage_next = S0;
            y = 1;
            z = 0;
       end
        else if(a == 1'b0 && b == 1'b1)begin
            stage_next = S1;
            y = 0;
            z = 0;
        end
        else begin
           stage_next = S0;
            y = 0;
            z = 0;
        end
    end
    S1:begin
        if(a == 1'b1 && b == 1'b0) begin
           stage_next = S0;y = 1;z = 1;
        end
        else if(a == 1'b0 && b == 1'b1) begin
           stage_next = S0;y = 1 ;z= 0;
        end
        else begin
           stage_next = S0;y = 0 ;z= 0;
        end
    end
    default:stage_next = S0;
    endcase
end
endmodule
```

扩展：设计一个自动售饮料机的逻辑电路。它的投币口每次只能投入一枚五角或一元的硬币。投入一元五角硬币后给出饮料；投入两元硬币时给出饮料并找回五角。

确定输入输出，投入一元硬币 A=1 ，投入五角硬币 B=1，给出饮料 Y=1，找回五角 Z=1；

确定电路的状态数，投币前初始状态为 S0，投入五角硬币为 S1，投入一元硬币为 S2。画出转该转移图，根据状态转移图可以写成 Verilog 代码。

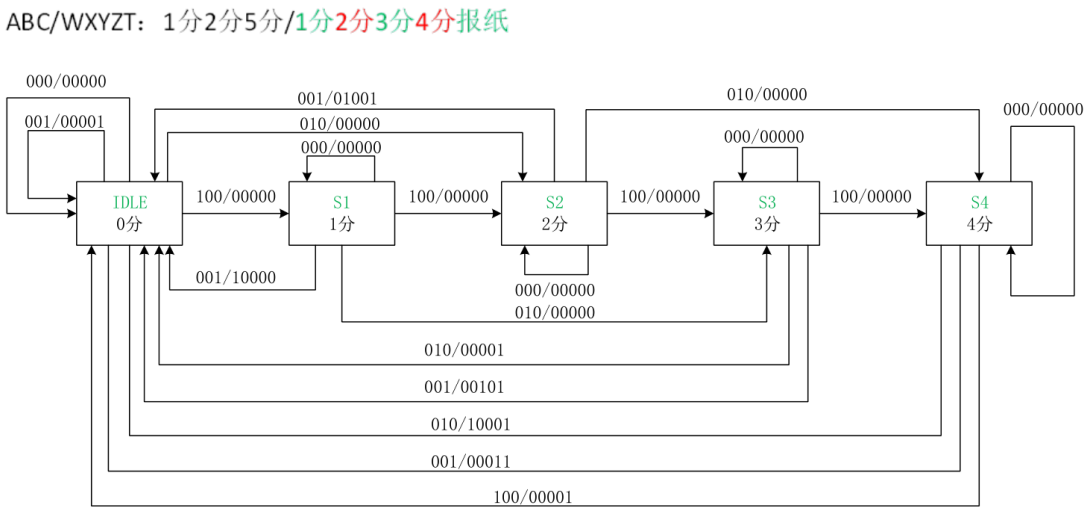
```
module drink_state (
 //system signals
 input clk ,
 input rst_n ,
 input a ,
 input b ,
 output reg y,
 output reg z
);
 parameter [1:0] S0 = 2'b00;
 parameter [1:0] S1 = 2'b01;
 parameter [1:0] S2 = 2'b10;
 reg [1:0] state_current;
 reg [1:0] state_next;
 always@(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)
       state_current <= S0;
    else
       state_current <= state_next;
 end
 always@(*)begin
    case(state_current)
    S0: if(a == 1'b0 && b == 1'b1)
          state_next <= S1;
        else if(a == 1'b1 && b == 1'b0)
          state_next <= S2;
        else
          state_next <= S0;
    S1: if(a == 1'b0 && b == 1'b1)
          state_next <= S2;
       else if(a == 1'b1 && b== 1'b0)
          state_next <= S0;
       else
          state_next <= S1;
    S2: if(a == 1'b0 && b == 1'b1)
          state_next <= S0;
       else if(a == 1'b1 && b== 1'b0)
          state_next <= S0;
       else
          state_next <= S2;
 endcase
 end
 always @ (posedge clk or negedge rst_n) begin
    if(rst_n == 1'b0)begin
```

```
        y <= 1'b0;
        z <= 1'b0;
    end
    else if(state_current == S1)begin
        if(a == 1'b1 && b== 1'b0)begin
            y <= 1'b1;
            z <= 1'b0;
        end
        else begin
            y <= 1'b0;
            z <= 1'b0;
        end
    end
    else if(state_current == S2)begin
        if(a == 1'b0 && b == 1'b1)begin
            y <= 1'b1;
            z <= 1'b0;
        end
        else if(a == 1'b1 && b== 1'b0)begin
            y <= 1'b1;
            z <= 1'b1;
        end
        else begin
            y <= 1'b0;
            z <= 1'b0;
        end
    end
    else begin
        y <= 1'b0;
        z <= 1'b0;
    end
 end
endmodule
```

3.2 卖报纸

画状态机，接受 1，2，5 分钱的卖报机，每份报纸 5 分钱
1、确定输入输出，投 1 分钱 A=1 ，投 2 分钱 B=1 ，投 5 分钱 C=1 ，给出报纸 Y=1
2、确定状态数画出状态转移图，没有投币之前的初始状态 S0，投入了 1 分硬币 S1，投入
了 2 分硬币 S2，投入了 3 分硬币 S3 ，投入了 4 分硬币 S4。
3、画卡诺图或者是利用 verilog 编码

链接解释没有给出，当输入 4 分钱后，给出 5 分钱的情况，不知是否需要处理。
https://blog.csdn.net/dongdongnihao_/article/details/79954335 待续




```
module fsm_01 (
    //system signals
    input                clk          ,
    input                rst_n        ,
    input          [2:0]  d_in          ,
    output     reg [3:0]  d_out         ,
    output     reg        y

);
//三段式状态机
//状态存储器
reg            [2  :0] state_c;
reg            [2  :0] state_n;
//状态定义
localparam         IDLE    =   3'd0       ;
localparam         S1      =   3'd1       ;
localparam         S2      =   3'd2       ;
localparam         S3      =   3'd3       ;
localparam         S4      =   3'd4       ;
//第一段：同步时序 always 模块，格式化描述次态寄存器迁移到现态寄存器
always@(posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        state_c <= IDLE;
    end
    else begin
        state_c <= state_n;
    end
end
```

```
//第二段：组合逻辑 always 模块，描述状态转移条件判断
always@(*)begin
    case(state_c)
        IDLE:begin
            if(d_in == 3'b100)begin
                state_n = S1;
            end
            if(d_in == 3'b010)begin
                state_n = S2;
            end
            if(d_in == 3'b001)begin
                state_n = IDLE;
            end
            else begin
                state_n = state_c;
            end
        end
        S1:begin
            if(d_in == 3'b100)begin
                state_n = S2;
            end
            if(d_in == 3'b010)begin
                state_n = S3;
            end
            if(d_in == 3'b001)begin
                state_n = IDLE;
            end
            else begin
                state_n = state_c;
            end
        end
        S2:begin
            if(d_in == 3'b100)begin
                state_n = S3;
            end
            if(d_in == 3'b010)begin
                state_n = S4;
            end
            if(d_in == 3'b001)begin
                state_n = IDLE;
            end
            else begin
                state_n = state_c;
            end
```

```
        end
        S3:begin
            if(d_in == 3'b100)begin
                state_n = S4;
            end
            if(d_in == 3'b010)begin
                state_n = IDLE;
            end
            if(d_in == 3'b001)begin
                state_n = IDLE;
            end
            else begin
                state_n = state_c;
            end
        end
        S4:begin
            if(d_in == 3'b100 | d_in == 3'b010 | d_in ==
3'b001)begin
                state_n = IDLE;
            end
            else begin
                state_n = state_c;
            end
        end
        default:begin
            state_n = IDLE;
        end
    endcase
end
//第三段：同步时序 always 模块，格式化描述寄存器输出（可有多个输出）
always  @(posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        d_out   <= 4'd0;
        y       <= 1'b1;
    end
    else begin
        case (state_c)
            IDLE    :begin
                case (d_in)
                    3'b100 :begin d_out <= 4'b0000; y <= 1'b0;  end
                    3'b010 :begin d_out <= 4'b0000; y <= 1'b0;  end
                    3'b001 :begin d_out <= 4'b0000; y <= 1'b1;  end
                    default:begin d_out <= d_out  ; y <= y;     end
                endcase
```

```
            end
            S1  :begin
                case (d_in)
                    3'b100 :begin d_out <= 4'b0000; y <= 1'b0;  end
                    3'b010 :begin d_out <= 4'b0000; y <= 1'b0;  end
                    3'b001 :begin d_out <= 4'b1000; y <= 1'b1;  end
                    default:begin d_out <= d_out  ; y <= y;     end
                endcase
            end
            S2  :begin
                case (d_in)
                    3'b100 :begin d_out <= 4'b0000; y <= 1'b0;  end
                    3'b010 :begin d_out <= 4'b0000; y <= 1'b0;  end
                    3'b001 :begin d_out <= 4'b0100; y <= 1'b1;  end
                    default:begin d_out <= d_out  ; y <= y;     end
                endcase
            end
            S3  :begin
                case (d_in)
                    3'b100 :begin d_out <= 4'b0000; y <= 1'b0;  end
                    3'b010 :begin d_out <= 4'b0000; y <= 1'b1;  end
                    3'b001 :begin d_out <= 4'b0010; y <= 1'b1;  end
                    default:begin d_out <= d_out  ; y <= y;     end
                endcase
            end
            S4  :begin
                case (d_in)
                    3'b100 :begin d_out <= 4'b0000; y <= 1'b1;  end
                    3'b010 :begin d_out <= 4'b1000; y <= 1'b1;  end
                    3'b001 :begin d_out <= 4'b0001; y <= 1'b1;  end
                    default:begin d_out <= d_out  ; y <= y;     end
                endcase
            end
            default:begin d_out <= 4'b0000; y <= 1'b0;  end
            endcase
    end
end
endmodule
```

3.3 序列检测

画出可以检测 10010 串的状态图 ， 并 verilog 实现
1、输入 data，1 和 0 两种情况，输出 Y=1 表示连续输入了 10010
2、确定状态数没输入之前 S0，输入一个 0 到了 S1,10 为 S2,010 为 S3,0010 为 S4

3.4 十字路口红绿灯信号

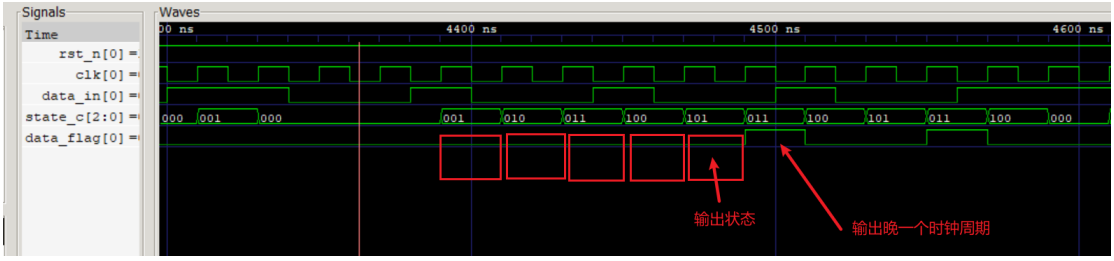
具体实现看链接：https://blog.csdn.net/weixin_43417303/article/details/104382822

3.5 帧头检测

连续两次收到 ‘0x23’ ，输出一个脉冲
未整

3.99 摩尔状态机

输出信号，采用 always 时序块，而且使用 state_current，输出延迟一个时钟周期，使用 state_next 解决输出晚一拍的问题




```
//检测序列 10010
`timescale 1ns/1ps
module fsm_moore (
 //system signals
 input clk ,
 input rst_n ,
 input data_in ,
 output reg data_flag
);
 //三段式状态机
 //状态存储器
 reg [ 2:0] state_c;
 reg [ 2:0] state_n;
 //状态定义
 localparam IDLE = 3'b000 ;
 localparam S1 = 3'b001 ;
 localparam S2 = 3'b010 ;
```

```
 localparam S3 = 3'b011 ;
 localparam S4 = 3'b100 ;
 localparam S5 = 3'b101 ;
 //第一段：同步时序 always 模块，格式化描述次态寄存器迁移到现态寄存器(不需更改)
 always@(posedge clk or negedge rst_n)begin
 if(!rst_n)begin
 state_c <= IDLE;
 end
 else begin
 state_c <= state_n;
 end
 end
 //第二段：组合逻辑 always 模块，描述状态转移条件判断
 always@(*)begin
 case(state_c)
 IDLE:begin state_n = data_in == 1'b1 ? S1 : IDLE;end
 S1 :begin state_n = data_in == 1'b0 ? S2 : IDLE;end
 S2 :begin state_n = data_in == 1'b0 ? S3 : IDLE;end
 S3 :begin state_n = data_in == 1'b1 ? S4 : IDLE;end
 S4 :begin state_n = data_in == 1'b0 ? S5 : IDLE;end
 S5 :begin state_n = data_in == 1'b0 ? S3 : IDLE;end
 default:begin state_n = IDLE;end
 endcase
 end
 //第三段：同步时序 always 模块，格式化描述寄存器输出（可有多个输出）
 //可以使用 state_next 解决输出晚一拍的问题
 always @(posedge clk or negedge rst_n)begin
 if(!rst_n)
 data_flag <= 1'b0;
 else
 case(state_n)
 S5 : data_flag <= 1'b1;
 default:data_flag <= 1'b0;
 endcase
 end
endmodule
```

3.99 米利状态机

输出信号，采用 always 时序块，而且使用 state_current，根据输入决定输出结果，输出没有晚一拍的问题

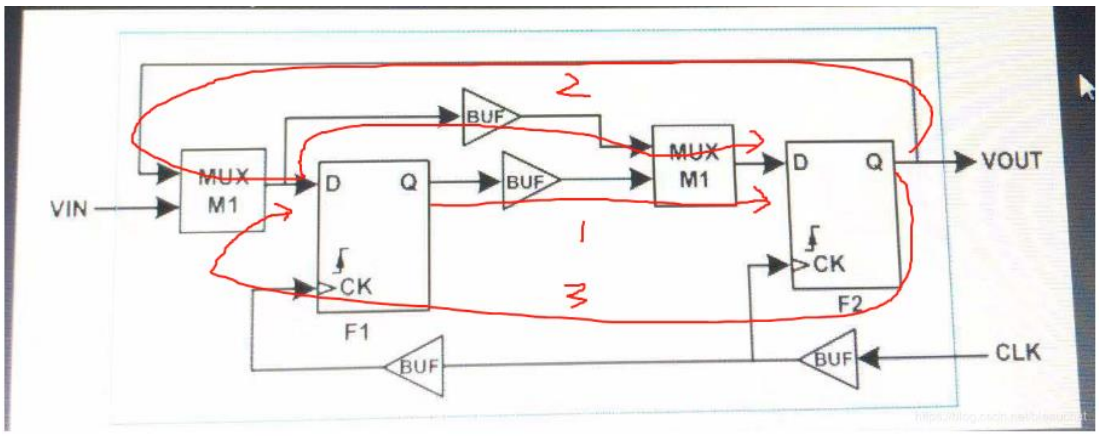
```
//序列检测 10010
`timescale 1ns/1ps
```

```
module fsm_mealy (
 //system signals
 input clk ,
 input rst_n ,
 input data_in ,
 output reg data_flag
);
//三段式状态机
//状态存储器
reg [ 2:0] state_c;
reg [ 2:0] state_n;
//状态定义
localparam IDLE = 3'b000 ;
localparam S1 = 3'b001 ;
localparam S2 = 3'b010 ;
localparam S3 = 3'b011 ;
localparam S4 = 3'b100 ;
localparam S5 = 3'b101 ;
//第一段：同步时序 always 模块，格式化描述次态寄存器迁移到现态寄存器(不需更
改）
always@(posedge clk or negedge rst_n)begin
 if(!rst_n)begin
 state_c <= IDLE;
 end
 else begin
 state_c <= state_n;
 end
end
//第二段：组合逻辑 always 模块，描述状态转移条件判断
always@(*)begin
 case(state_c)
 IDLE:begin state_n = data_in == 1'b1 ? S1 : IDLE;end
 S1 :begin state_n = data_in == 1'b0 ? S2 : IDLE;end
 S2 :begin state_n = data_in == 1'b0 ? S3 : IDLE;end
 S3 :begin state_n = data_in == 1'b1 ? S4 : IDLE;end
 S4 :begin state_n = data_in == 1'b0 ? S5 : IDLE;end
 S5 :begin state_n = data_in == 1'b0 ? S3 : IDLE;end
 default:begin state_n = IDLE;end
 endcase
end
//第三段：同步时序 always 模块，格式化描述寄存器输出（可有多个输出）
//使用 state_current 和当前输入的信号决定当前输出，信号无延迟
always @(posedge clk or negedge rst_n)begin
 if(!rst_n)begin
```

```
 data_flag <= 1'b0;
 end
 else
 case(state_c)
 S4 : data_flag <= (data_in == 1'b0) ? 1'b1 : 1'b0;
 default:data_flag <= 1'b0;
 endcase
end
endmodule
//只有第二段的状态转移，和最后一段的逻辑输出，需要根据状态图进行调整
//（该序列检测，moore 和 mealy 的第二段状态转移是一样的）
```

4 时序分析

4.1 建立时间、保持时间、最小时钟频率



分析建立时间和保持时间必须找到传输路径，如图所示存在 3 条传输路径：

1、F1_Q --> F2_D

2、F2_Q --> F2_D

3、F2_Q --> F1_D

问题 1，电路是否存在保持时间违例，保持时间需要满足，

Tco + Tdelay >= Thold + Tskew

时钟歪斜：

路径 1：Tskew_1 = Tbuf - 2*Tbuf = -Tbuf

路径 2：Tskew_2 = 0

路径 3：Tskew_3 = 2*Tbuf - Tbuf = Tbuf

分析保持时间

路径 1：Tco_min + Tbuf_min + Tmux_min >= Thold + Tskew_1_max

1 + 1 + 1 >= 1.5 - 1 ：满足条件

路径 2：Tco_min + Tmux_min + Tbuf_min + Tmux_min >= Thold + Tskew_2_max

1 + 1 + 1 + 1 >= 1.5 + 0 ：满足条件

路径 3：Tco_min + Tmux_min >= Thold + Tskew_3_max

1 + 1 >= 1.5 + 1：不满足条件

存在保持时间违例的情况，只需要将 clk 的两个 buf 去掉即可

问题 2：关键路径为 2

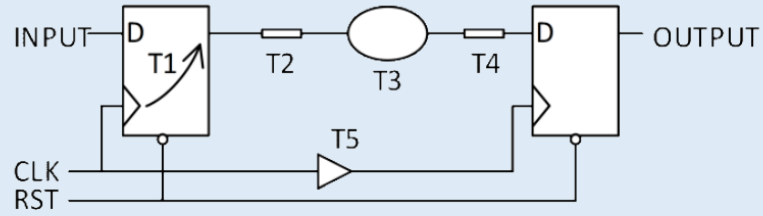
Tco_max + Tmux_max + Tbuf_max + Tmux_max + Tsetup <= T + Tskew_2_min

4 + 3 + 2 + 3 + 4 <= T + 0

T >= 16ns 即 62.5MHz

4.2 建立时间、保持时间分析

4.在同步电路设计中，电路时序模型如下：T1为触发器时钟到输出延时，T2和T4为连线延时，T3为组合路径延时，T5为时钟网络延时，假设时钟周期为Tcycle，Tsetup，Thold分别为触发器建立保持时间，为保证？？正确采样（改路径为multi-cycle路径），下列必须满足的是：



① T1+T2+T3+T4<Tcycle-Tsetup+T5，T1+T2+T3+T4>Thold

② T1+T2+T3+T4<Tcycle-Tsetup，T1+T2+T3+T4+T5>Thold

③ T1+T2+T3+T4+T5<Tcycle-Tsetup，T1+T2+T3+T4>Thold

④ T1+T2+T3+T4<Tcycle-Tsetup+T5，T1+T2+T3+T4> Thold+T5

这个题目堪称经典中的经典，因为从这个题目是标准的建立时间和保持时间考题，并从中可以总结出系统最大时钟频率以及建立时间和保持时间需要满足的公式。

建立时间分析

我们知道系统周期需要满足的条件是：

Tcycle + Tskew > Tco + Tgate + Tsu;

代入上面的条件，得知

Tcycle + T5 > T1 + T2 + T3 + T4 + Tsetup

保持时间分析

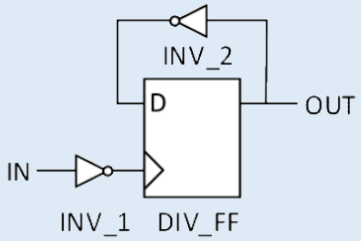
上面的时钟路径是存在偏斜的，而且是正时钟偏斜，则对于保持时间的满足更加的困难，需要满足：

Thold + Tskew < Tc0 + Tgate

代入上面的条件，得： Thold + T5 < T1 + T2 + T3 + T4

4.3 求系统最高频率

1.如下分频电路，触发器DIV_F的建立时间2ns，保持时间2ns，逻辑延时6ns，反相器INV_1，INV_2的逻辑延时为2ns，连线延时为0，该电路正常工作的最高频率？

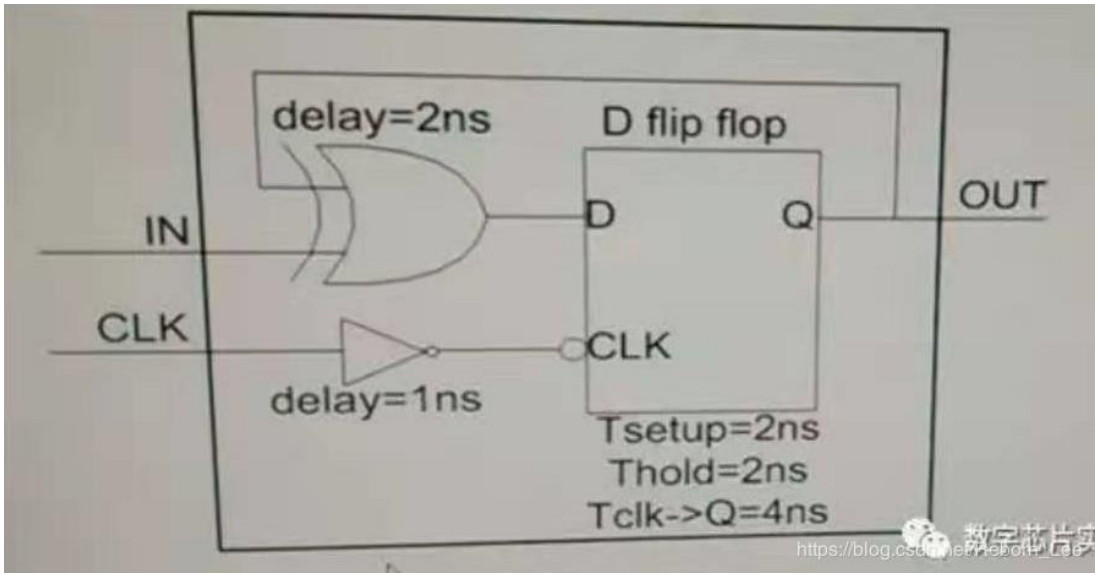


仔细看来，这是一个触发器到自身的反馈，可以看做两个触发器之间进行数据传

输，其实也就是如此。
需要明白的是，由于时钟到达这个触发器的时间一致，所以不存在时钟偏斜。
给出系统周期满足的关系：
Tcycle > Tco + Tgate + Tsu = 6 + 2 + 2= 10ns
所以时钟最大频率可以为 100MHz。

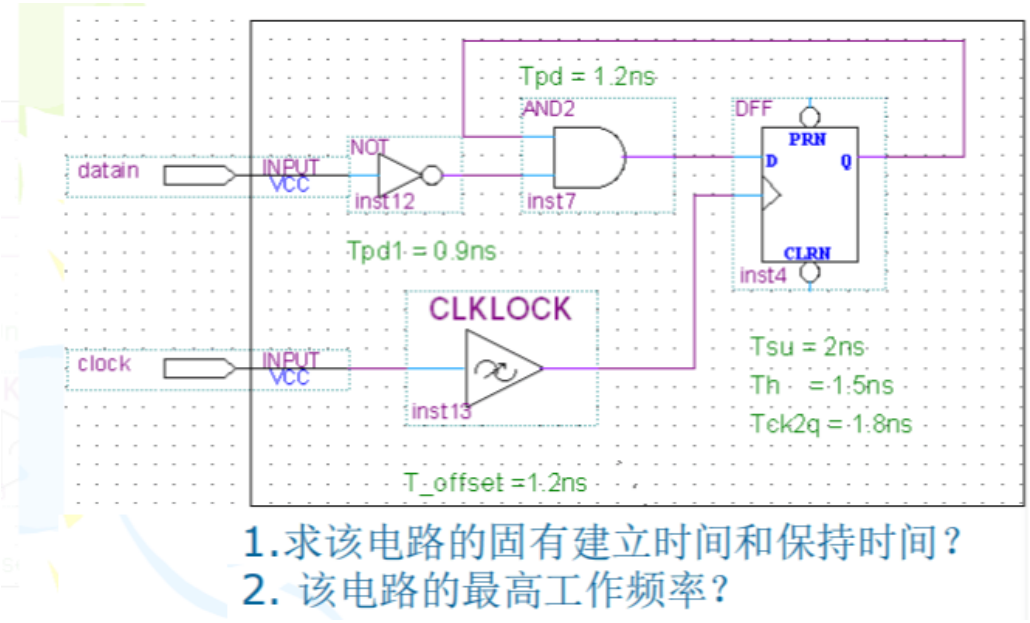
4.4 有效建立时间保持时间

如下图，将框内的电路作为一个寄存器，那么其有效的建立时间和保持时间是多少



有效建立时间分析
假设电路的有效 Setup 为 Tsetup_valid：
对于 D 触发器而言，其本身的建立时间是 2ns，也就是说数据必须在时钟有效沿到达之前 2ns 保持稳定，这样到达 D 端后就一定是稳定的数据了。
这个电路的数据来自于 IN，时钟来自于 CLK；
考虑时钟路径延迟影响：
时钟 CLK 要早于触发器的时钟 1ns 到达，因此对于 D 触发器建立时间的满足是有害的，电路有效建立时间
Tsetup_valid = Tsetup - 1ns = 1ns（因为数据需要提前 1ns 稳定下来）
考虑数据路径延迟影响：
Tsetup_valid = Tsetup - 1ns + 2ns = 3ns；（经过组合逻辑后的数据需要在时钟有效沿之前 Tsetup 时间稳定下来）
有效保持时间分析
和建立时间分析套路一致，对于 D 触发器而言，数据需要在时钟有效沿到来之后保持 Thold 时间。
考虑时钟延迟的影响：
考虑到电路时钟对于触发器时钟早到 1ns，所以电路有效保持时间 Thold_valid = Thold + 1ns = 3ns;
考虑路径延迟影响：
数据需要经过一段组合逻辑之后才能保持稳定，因此电路的有效保持时间为：
Thold_valid = Thold + 1ns - 2ns = 1ns。

4.5 有效建立时间保持时间



因此固有建立时间为 Tsu_valid = Tsu - T_offset + Tpd + Tpd1 = 2 - 1.2 + 1.2 + 0.9 = 2.9ns

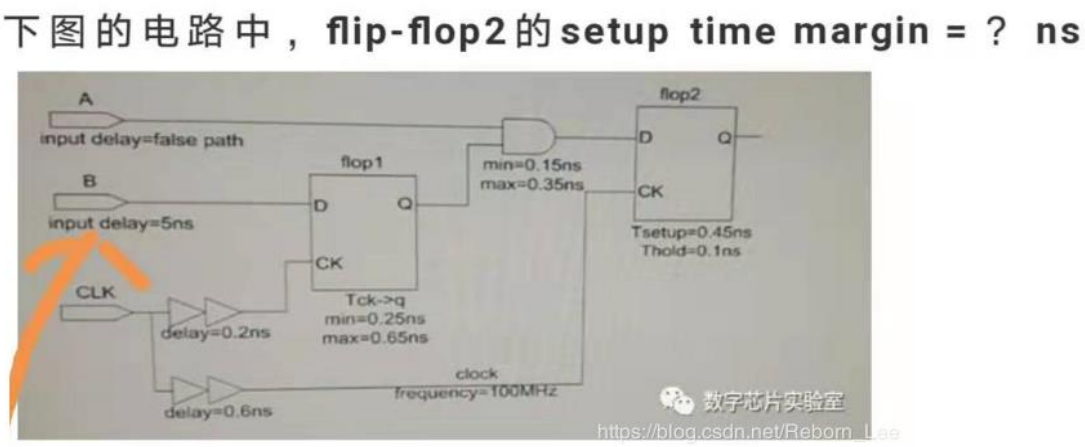
固有保持时间为：Th_valid = Th + T_offset - Tpd - Tpd1 = 1.5 + 1.2 - 1.2 - 0.9 = 0.6ns.

而系统的最高频率呢？

先求系统的最小周期，考虑两个触发器之间的路径：

Tmin = Tco + Tpd + Tsu = 1.8 + 1.2 + 2 = 5ns,那么系统最高频率为 200MHz。

4.6 建立时间裕量



这个题目让求 setup time margin，意思大概就是建立时间裕量，就是系统周期减去 Tco，Tgate 以及 Tsu 之后还可以有多少裕量，那，Tco，Tgate 以及 Tsu 当然要用最大的来代入，因为要保证系统在最恶劣的情况下，能有多少裕量。

因此：

Tmargin = 10ns + 0.6ns - 0.2ns - 0.65ns - 0.35ns - 0.45ns = 8.95ns

其中 0.6ns - 0.2ns 表示的是时钟偏斜量，可见是时钟整偏斜，有利于时钟裕量。

4.7 小结

总结，最难的部分应该是求电路的固有建立时间和保持时间了吧，建立时间和保持时间是一对冤家，利你不利它，如果你分析了建立时间，那么保持时间相反就可以了；例如逻辑门延迟不利于系统建立时间（+），那么利于系统保持时间（-），时钟延迟有利于系统建立时间（-），则不利于系统保持时间（+）。

5 版图

5.1 用 mos 管搭出一个二输入与非门

与非门：上并下串 或非门：上串下并

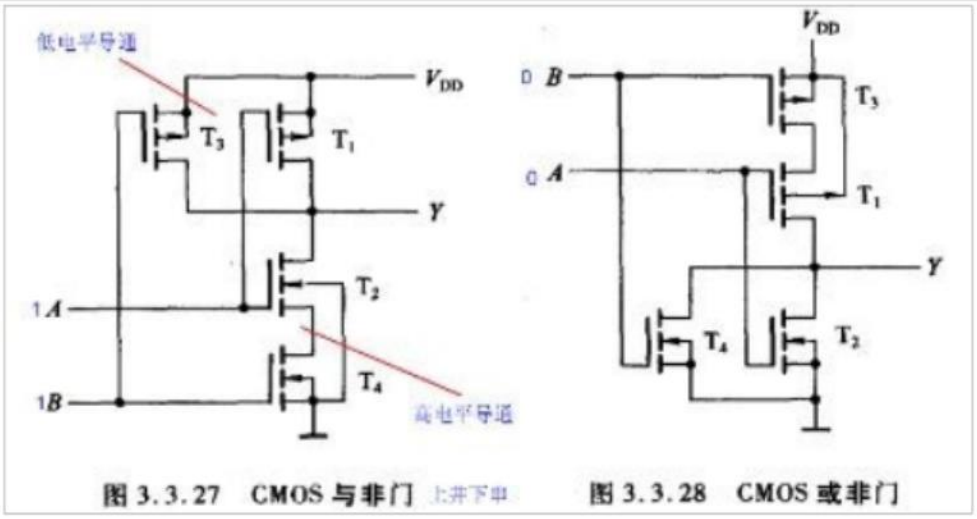


图 3.3.27 CMOS 与非门 上并下串　　图 3.3.28 CMOS 或非门

5.2 CMOS 电路设计

根据逻辑，分析电路结构，使用 cmos 进行整理，上下互补对称