

2020-2021学年秋季学期

第六部分-B 高级时序逻辑设计

授课团队：宋威

助 教：薛子涵

- 使用Verilog HDL设计时序逻辑电路
 - 任意数制计数器、随机数发生器、簇发数据检测器、自动贩卖机
 - 参数 (parameter) 的使用
- 时序逻辑电路的测试设计
 - 时钟和复位的产生、断言检查、自启
- 复杂时序逻辑电路设计
 - 循环仲裁器 (round-robin arbiter)
 - 指针型的FIFO缓冲
- 时序逻辑电路的时序分析
 - 路径时间检查
 - 时钟树
- 附加内容 (演示)
 - 带自动结果检验的测试程序
 - 电路的综合

任意计数数值计数器-端口

```
module counterN(
    input clk,
    input [3:0] num, // 计数数值
    output [3:0] q,
    output c);

    reg [3:0] q;
    always @(negedge clk)
        if(c) q <= 0;
        else q <= q + 1;

    assign c = q == num - 1;
endmodule

wire [3:0] q;
wire c;

counter cnt(.clk(clk), .num(10), .q(q), .c(c)); // 使用计数器
```

将计数数值定为端口上的一个输入。

任意计数数值计数器-参数(parameter)

```
module counterN #(parameter num=10) (  
    input clk,  
    output [3:0] q,  
    output c);  
  
    reg [3:0] q;  
    always @(negedge clk)  
        if(c)    q <= 0;  
        else    q <= q + 1;  
  
    assign c = q == num - 1;  
endmodule  
  
wire [3:0] q;  
wire c;  
  
counter #(.num(10)) cnt(.clk(clk), .q(q), .c(c)); // 使用计数器
```

将计数数值定为一个参数。

端口和参数的比较

○端口

- 端口是真实的硬件实现
- 需要手动连接一个信号作为输入（不过可以接常量）
- 允许在运行时改变

○参数

- 参数是综合时优化技术，在实例化时必须设定为一个常数
- 方便了综合器对电路优化
- 不允许在运行时改变
- 有默认值

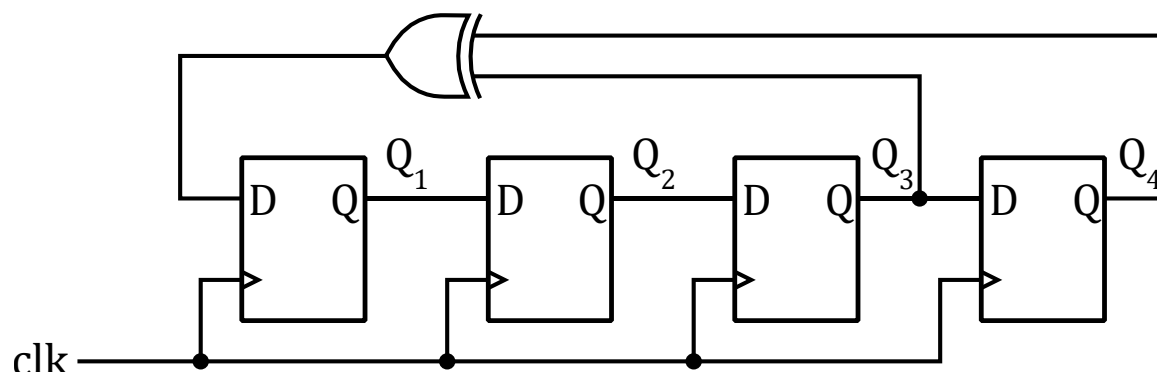
伪随机数发生器

- 使用特定的反馈函数，对于N位的移位寄存器，我们总是能生成长度为 $2^N - 1$ 的伪随机数序列。

N	F()	N	F()
3	$Q_3 \oplus Q_2$	4	$Q_4 \oplus Q_3$
5	$Q_5 \oplus Q_3$	6	$Q_6 \oplus Q_5$
7	$Q_7 \oplus Q_6$	8	$Q_8 \oplus Q_6 \oplus Q_5 \oplus Q_4$
9	$Q_9 \oplus Q_5$	10	$Q_{10} \oplus Q_7$
11	$Q_{11} \oplus Q_9$	12	$Q_{12} \oplus Q_6 \oplus Q_4 \oplus Q_1$
13	$Q_{13} \oplus Q_4 \oplus Q_3 \oplus Q_1$	14	$Q_{14} \oplus Q_5 \oplus Q_3 \oplus Q_1$
15	$Q_{15} \oplus Q_{14}$	16	$Q_{16} \oplus Q_{15} \oplus Q_{13} \oplus Q_4$

https://www.xilinx.com/support/documentation/application_notes/xapp210.pdf

4位移位寄存器：伪随机数发生器（异或）



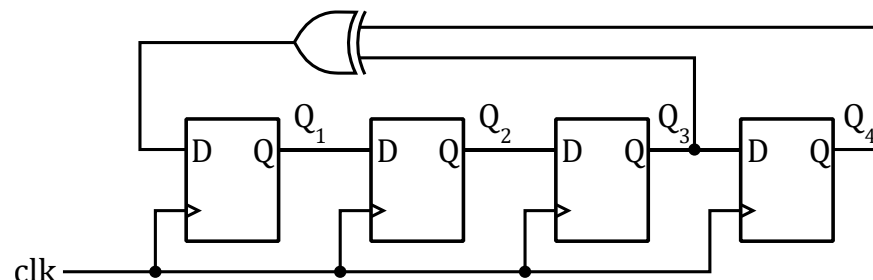
$$D_1 = Q_3 \oplus Q_4$$

1	2	3	4	5	6	7	8
0001,	1000,	0100,	0010,	1001,	1100,	0110,	1011
0101,	1010,	1101,	1110,	1111,	0111,	0011,	0001
1	8	4	2	9	12	6	11
5	10	13	14	15	7	3	1

0000为非法状态

4位伪随机数发生器

```
module lfsr4 (  
    input clk,  
    output [3:0] q);  
  
    reg [3:0] q;  
    always @(posedge clk)  
        q <= {q[2:0], q[3]^q[2]};  
  
endmodule
```



如何做一个通用模块呢？

n位伪随机数发生器

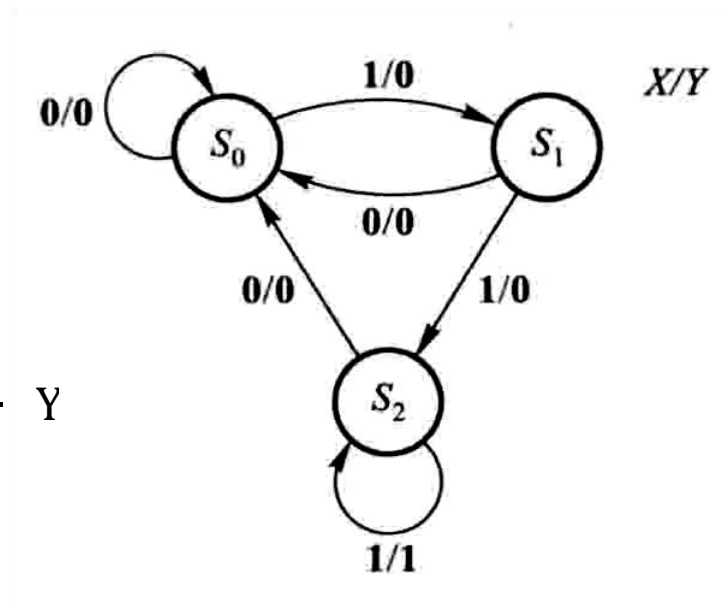
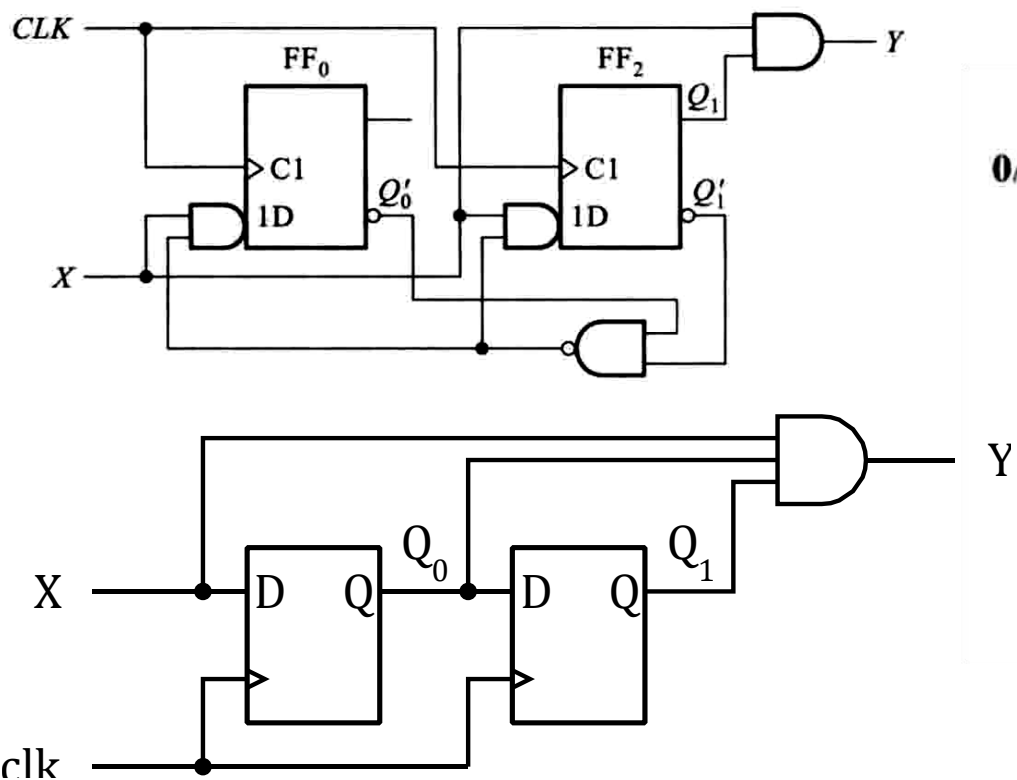
```
module lfsrN #(parameter N=4) (  
    input clk,  
    output [N-1:0] q);  
  
    reg [N-1:0] q;  
    always @(posedge clk) begin  
        q[N-1:1] <= q[N-2:0];  
        if(N==3) q[0] <= q[2]^q[1];  
        if(N==4) q[0] <= q[3]^q[2];  
        if(N==5) q[0] <= q[4]^q[2];  
        if(N==6) q[0] <= q[5]^q[4];  
        if(N==7) q[0] <= q[6]^q[5];  
        if(N==8) q[0] <= q[7]^q[5]^q[4]^q[3];  
    end  
endmodule
```

N	F()	N	F()
3	$Q_3 \oplus Q_2$	4	$Q_4 \oplus Q_3$
5	$Q_5 \oplus Q_3$	6	$Q_6 \oplus Q_5$
7	$Q_7 \oplus Q_6$	8	$Q_8 \oplus Q_6 \oplus Q_5 \oplus Q_4$

时序逻辑电路设计：簇发检测器

○设计一个串行数据检测器，当连续输入3个或以上的1时输出1，其他为0。

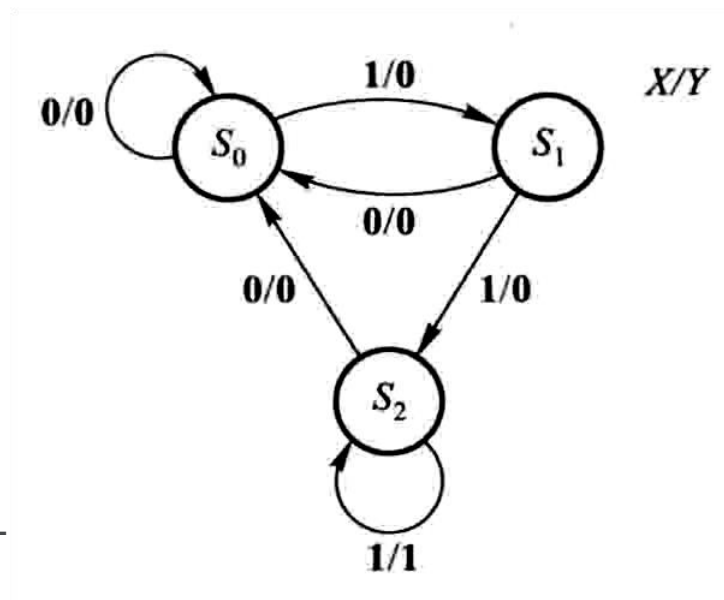
○定义状态： S_n 表示连续检测到 n 个输入1周期



时序逻辑电路设计：簇发检测器

- 设计一个串行数据检测器，当连续输入3个或以上的1时输出1，其他为0。

```
module burst_detect3 (  
    input clk,  
    input x,  
    output y);  
  
    reg [1:0] cnt; // 计数器  
    always @(posedge clk) begin  
        if(x==0) cnt <= 0;  
        else if(cnt != 2) cnt <= cnt + 1  
    end  
  
    assign y = (cnt == 2) & x;  
  
endmodule
```



推广到N位

时序逻辑电路设计：N位簇发检测器

```
module burst_detect #(parameter N=3) (  
    input clk,  
    input x,  
    output y);  
  
    reg [N-1:0] cnt; // 计数器  
    always @(posedge clk) begin  
        if(x==0) cnt <= 0;  
        else if(cnt != N-1) cnt <= cnt + 1;  
    end  
  
    assign y = (cnt == N-1) & x;  
  
endmodule
```

时序逻辑电路设计：贩卖机

- 设计一个饮料贩卖机，售价1.5元，每次接受一个硬币（1元/05毛）。如果投入1.5元，给一杯饮料。如果投入2元，给一杯饮料，吐出5毛。

- A: 投入1元硬币
- B: 投入5毛硬币
- Y: 给饮料
- Z: 退还5毛硬币

$$Q_1^* = Q_1 A' B' + Q_1' Q_0' A + Q_0 B$$

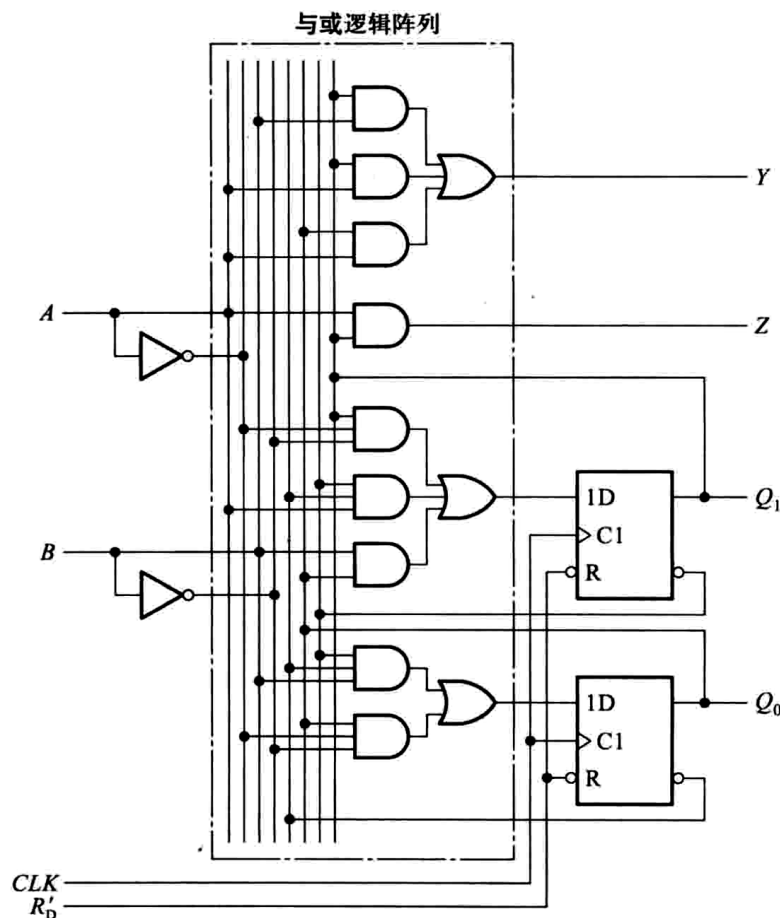
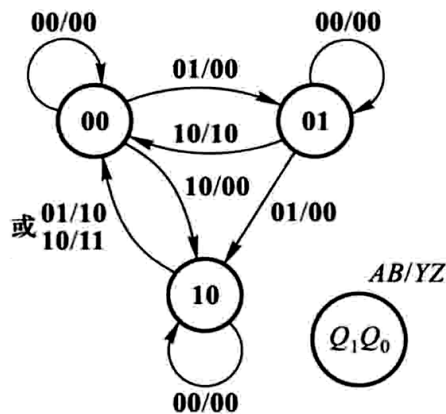
$$Q_0^* = Q_1' Q_0' B + Q_0 A' B'$$

$$D_1 = Q_1 A' B' + Q_1' Q_0' A + Q_0 B$$

$$D_0 = Q_1' Q_0' B + Q_0 A' B'$$

$$Y = Q_1 B + Q_1 A + Q_0 A$$

$$Z = Q_1 A$$



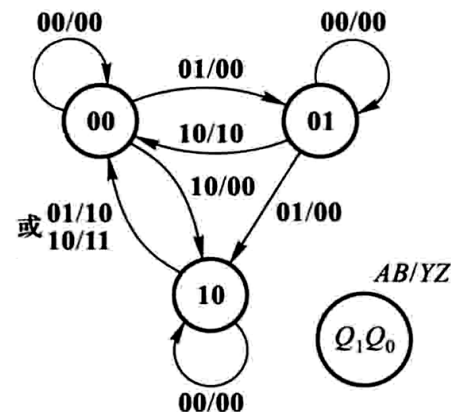
时序逻辑电路设计：贩卖机

- 设计一个饮料贩卖机，售价1.5元，每次接受一个硬币（1元/05毛）。如果投入1.5元，给一杯饮料。如果投入2元，给一杯饮料，吐出5毛。

A: 投入1元硬币、B: 投入5毛硬币、Y: 给饮料

Z: 退还5毛硬币

```
module vending_machine (  
    input clk, rstn,  
    input a, b,  
    output y, z);  
  
    reg [1:0] state; // 状态  
    always @(posedge clk or negedge rstn) begin  
        if(rstn == 0) state <= 0;  
        else case(state)  
            0: state <= a ? 2 : (b ? 1 : 0);  
            1: state <= a ? 0 : (b ? 2 : 1);  
            2: state <= a ? 0 : (b ? 0 : 2);  
            default: state <= 0;  
        endcase  
    end  
    assign y = ((state == 1) & a) | ((state == 2) & (a|b));  
    assign z = (state == 2) & a;  
endmodule
```



状态机描述形式

时序逻辑电路设计：贩卖机

- 设计一个饮料贩卖机，售价1.5元，每次接受一个硬币（1元/05毛）。如果投入1.5元，给一杯饮料。如果投入2元，给一杯饮料，吐出5毛。

A: 投入1元硬币、B: 投入5毛硬币、Y: 给饮料

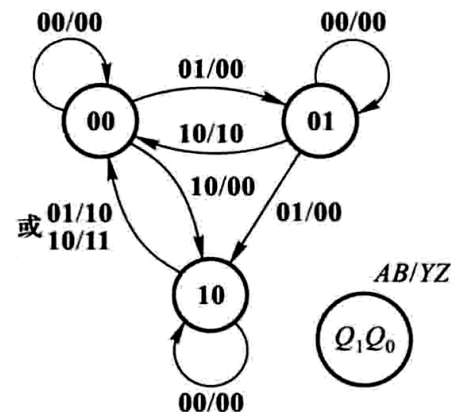
Z: 退还5毛硬币

```
module vending_machine (  
    input clk, rstn,  
    input a, b,  
    output y, z);
```

```
    reg [1:0] paid_pre; // 之前已投钱数*2  
    wire [2:0] paid; // 当前已投钱数*2
```

```
    assign paid = {1'b0, paid_pre} + {1'b0, a, 1'b0} + {2'b00, b};  
    assign y = paid >= 3;  
    assign z = paid == 4;
```

```
    always @(posedge clk or negedge rstn) begin  
        if(rstn == 0) paid_pre <= 0;  
        else if(y==0) paid_pre <= paid;  
        else  
            paid_pre <= 0;  
    end  
endmodule
```



使用Verilog HDL设计时序逻辑电路总结

○时序逻辑电路

- 沿触发的always块
- 时钟、复位信号（逻辑）

○扩展

- 在RTL级描述电路具有一定的可扩展性
- 利用端口或参数来扩展电路功能

○状态机的描述

- 单独一个always块专门描述状态机
- 其他的电路根据状态机的当前状态进行取值
- 可以不严格按照状态机实现（各有利弊）

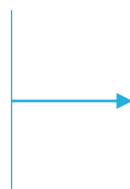
○新引入的Verilog关键字

- parameter
- 条件三元运算符？：

测试N位簇发检测器：时钟驱动

```
module test;  
  
    reg clk, x;  
    wire y;  
  
    burst_detect #(3) dut(clk, x, y);
```

```
initial begin  
    clk = 0;  
    forever #5 clk = ~clk;  
end
```



时钟驱动

```
always @(posedge clk)  
    x <= $random;
```

```
initial begin  
    $dumpfile("test.vcd");  
    $dumpvars(0);  
    #1000 $finish();  
end  
endmodule
```

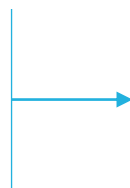


设置结束时间

测试N位簇发检测器：时钟驱动

```
module test;  
  
    reg clk, x;  
    wire y;  
  
    burst_detect #(3) dut(clk, x, y);
```

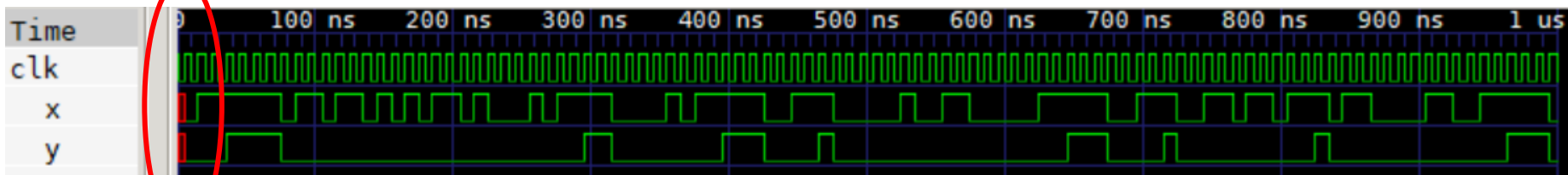
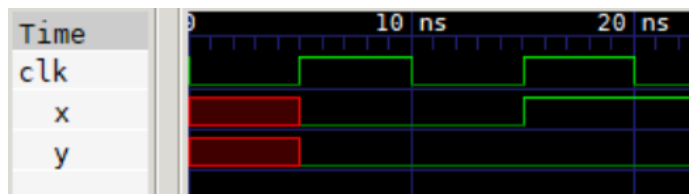
```
initial begin  
    clk = 0;  
    forever #5 clk = ~clk;  
end
```



时钟驱动

```
always @(posedge clk)  
    x <= $random;
```

```
initial begin  
    $dumpfile("test.vcd");  
    $dumpvars(0);  
    #1000 $finish();  
end  
endmodule
```



测试N位簇发检测器：无需复位信号（仿真自启）

```
module burst_detect #(parameter N=3) (  
    input clk,  
    input x,  
    output y);
```

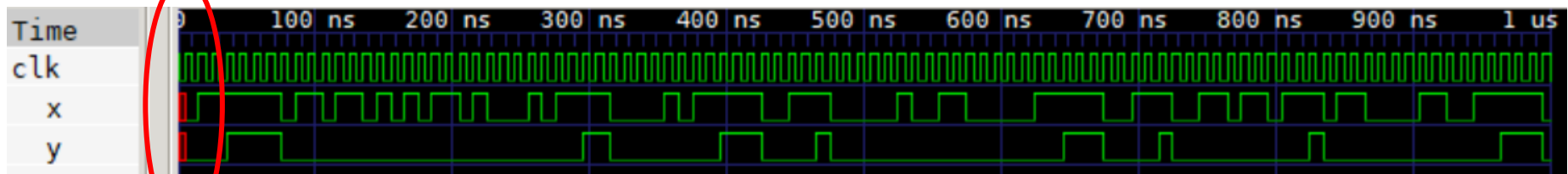
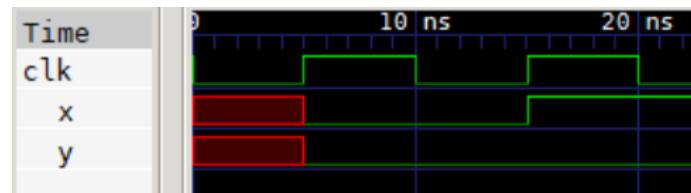
```
    reg [N-1:0] cnt; // 计数器  
    always @(posedge clk) begin  
        if(x==0) cnt <= 0; ←  
        else if(cnt != N-1) cnt <= cnt + 1;  
    end
```

cnt也无需复位

```
    assign y = (cnt == N-1) & x;
```

```
endmodule
```

当x=0时，y被强制为0



测试任意计数器：需要复位信号

```
module test;
  reg clk;
  wire [3:0] q;
  wire c;

  counterN #(12) dut(clk, q, c);

  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end

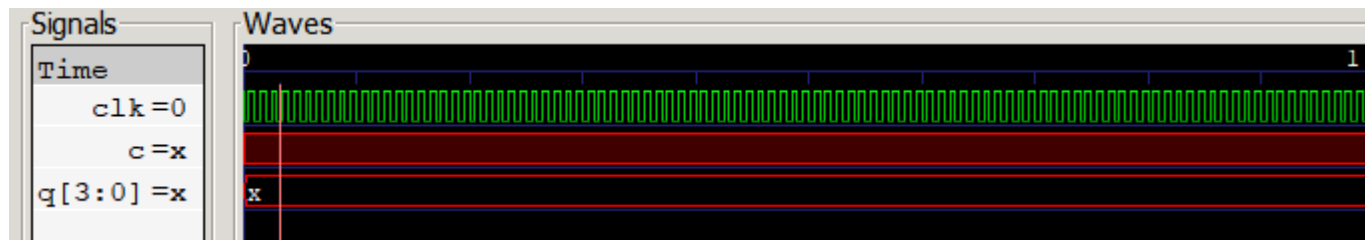
  initial begin
    $dumpfile("test.vcd");
    $dumpvars(0);
    #1000 $finish();
  end
endmodule
```

```
module counterN #(parameter num=10) (
  input clk,
  output [3:0] q,
  output c);

  reg [3:0] q;
  always @(negedge clk)
    if(c) q <= 0;
    else q <= q + 1;

  assign c = q == num - 1;
endmodule
```

c的初始值是x



复位方法一：\$deposit()

```
module test;
  reg clk;
  wire [3:0] q;
  wire c;

  counterN #(12) dut(clk, q, c);

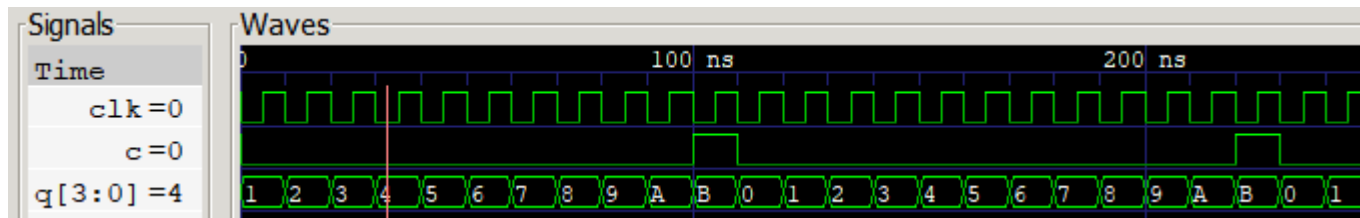
  initial begin
    $deposit(dut.q, 0);
    clk = 0;
    forever #5 clk = ~clk;
  end

  initial begin
    $dumpfile("test.vcd");
    $dumpvars(0);
    #1000 $finish();
  end
endmodule
```

```
module counterN #(parameter num=10) (
  input clk,
  output [3:0] q,
  output c);

  reg [3:0] q;
  always @(negedge clk)
    if(c) q <= 0;
    else q <= q + 1;

  assign c = q == num - 1;
endmodule
```



复位方法二：硬件复位

```
module test;
  reg clk, rstn;
  wire [3:0] q;
  wire c;

  counterN #(12) dut(clk, rstn, q, c);

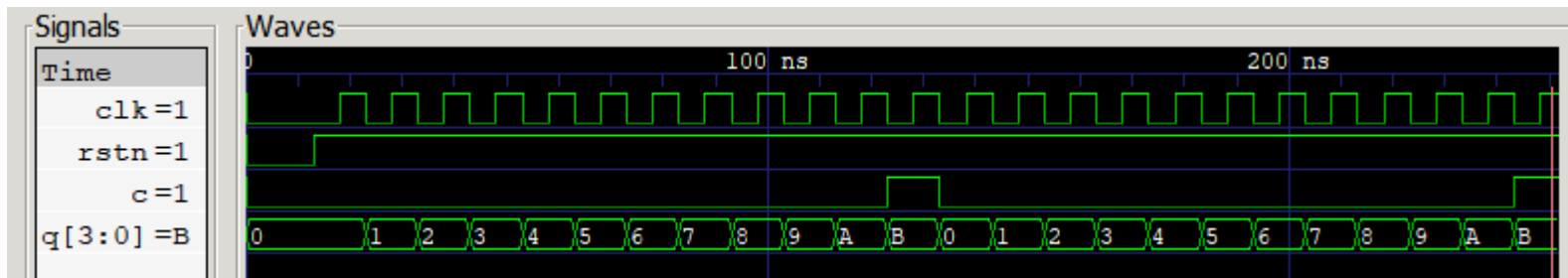
  initial begin
    clk = 0; rstn = 0;
    #13 rstn = 1;
    forever #5 clk = ~clk;
  end

  initial begin
    $dumpfile("test.vcd");
    $dumpvars(0);
    #1000 $finish();
  end
end
endmodule
```

```
module counterN #(parameter num=10) (
  input clk, rstn,
  output [3:0] q,
  output c);

  reg [3:0] q;
  always @(negedge clk or negedge rstn)
    if(~rstn) q <= 0;
    else if(c) q <= 0;
    else      q <= q + 1;

  assign c = q == num - 1;
endmodule
```



○ \$deposit() 系统函数

- Verilog HDL的运行时函数，将特定信号赋值
- 不生成任何硬件（不可综合）
- 往往用于设置初始值和仿真时注入错误
- 实际硬件应当能够自启

○ 使用硬件复位信号（推荐）

- 一般使用寄存器异步复位设置初始状态
- 也可以使用同步复位（同步/异步之间的争端）
- 生成带复位的电路
- 确保仿真和实际硬件行为一致

运行时电路功能检测（断言）

```
module test;
    reg clk, rstn;
    wire [3:0] q;
    wire c;

    counterN #(12) dut(clk, rstn, q, c);

    initial begin
        clk = 0; rstn = 0;
        #13 rstn = 1;
        forever #5 clk = ~clk;
    end

    always @(posedge clk)
        if(c & (q != 11)) begin // 断言判断
            $display("error!\n");
            $finish();
        end

    initial begin
        $dumpfile("test.vcd");
        $dumpvars(0);
        #1000 $finish();
    end
endmodule
```

断言：

在合适的时间，对电路的输出/状态警醒检测，如果检测失败，报错。

优势：

- 在测试程序中，不生成实际硬件。
- 可以直接报告出现错误的时间、信号、状态。
- 可以作为VIP复用/打包/出售。

在SystemVerilog中，被更强大的\$assert()函数代替。

时序逻辑电路的测试设计总结

○时钟和复位的产生

- 时钟信号由 `forever` 语句产生，注意时钟的初始信号。
- 复位信号在 `initial` 块中设置，注意复位信号不要和时钟信号同时变化。

○自启

- 要区分硬件自启和仿真能够自启的区别。
- 硬件自启：状态空间中的无效状态可以在有限时间内回到有效状态。
- 仿真自启：信号能够在有限时间能摆脱初始 `x` 状态。
- 当硬件可自启但是仿真不能自启的时候，可以用 `$deposit()` 复位。
- 当硬件不能自启时，必须使用硬件复位。

○断言检查

- 运行时利用测试代码直接检测电路状态的方法。

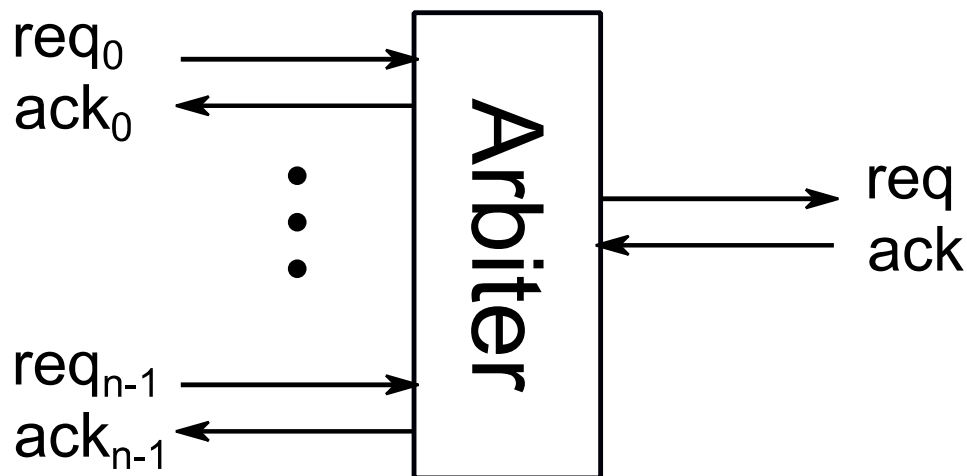
○增加的Verilog HDL关键字

`forever`, `$deposit()`, `$display`

复杂时序逻辑电路设计：仲裁器

○仲裁器 (Arbiter)

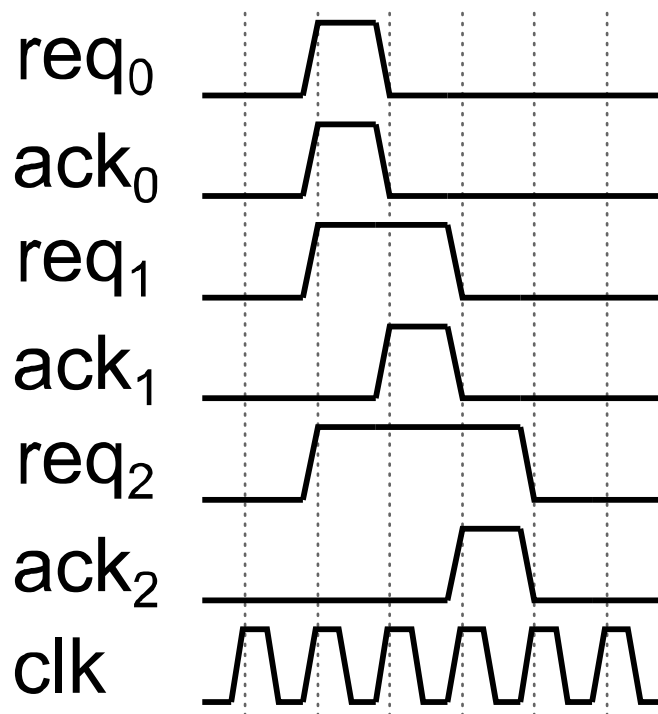
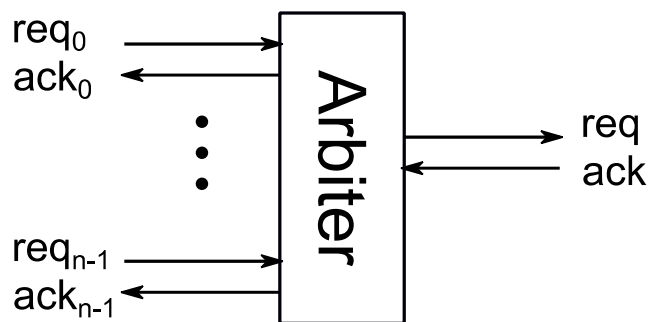
- 当一个共享的资源被多个电路争用时，我们需要仲裁器在可能同时出现的多个请求中选出一路请求，让其使用资源。



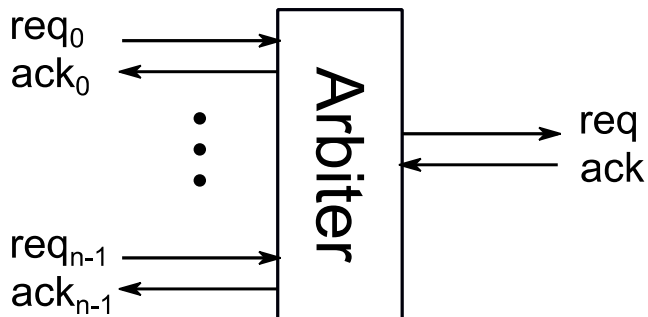
- 如何知道哪一路信号正在请求资源?
- 如何通知被选中的那一路信号?
- (请求, 响应) : (request, acknowledge) 或者 (req, ack)

○假设

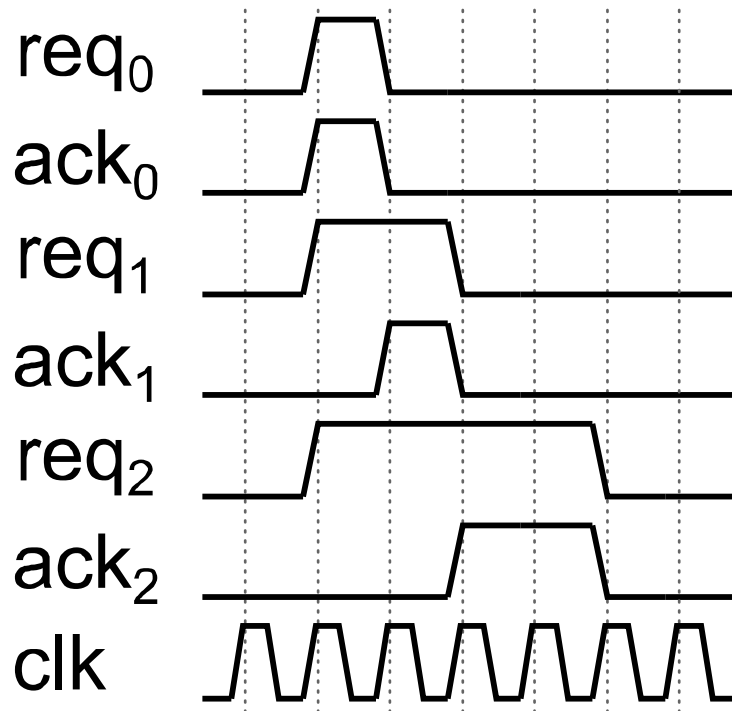
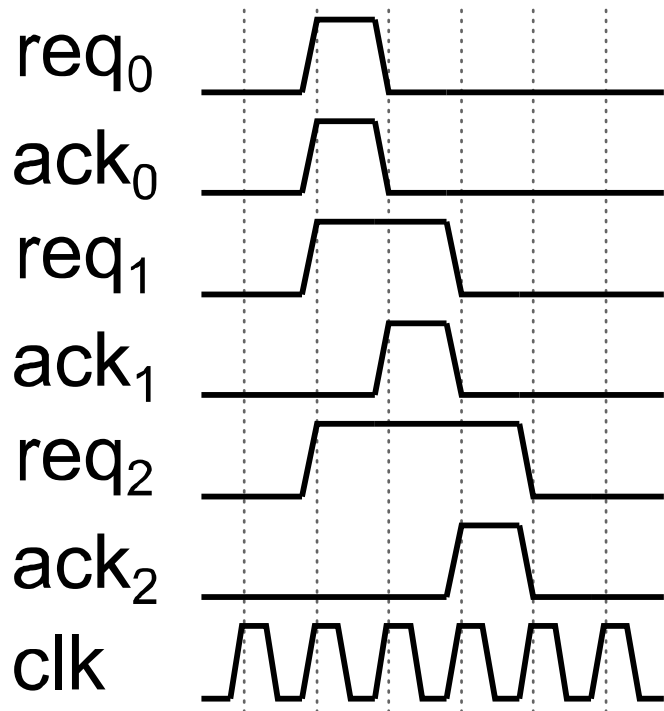
- 每个周期, $req_0 \sim req_{n-1}$ 都可能发出请求 (将其信号置为1)
- 当 req_i 变为1之后, 在 ack_i 为1之前, req_i 不能擅自归位为0
- 一次请求的持续时间为1周期
- 响应信号的高电平只维持1周期



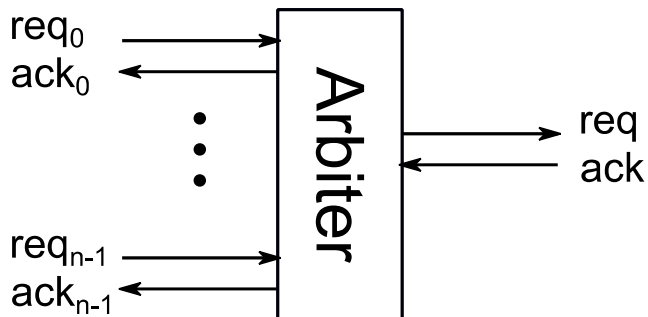
仲裁器输入



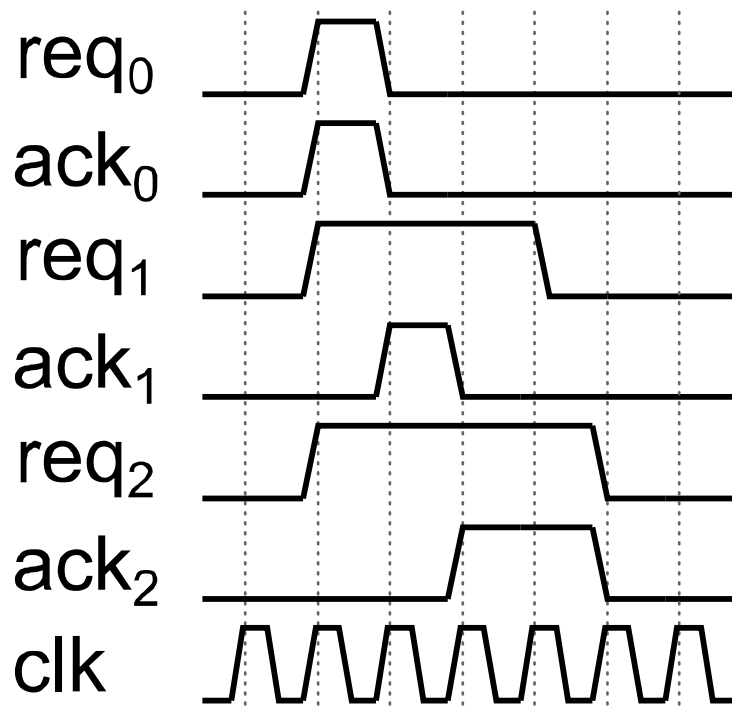
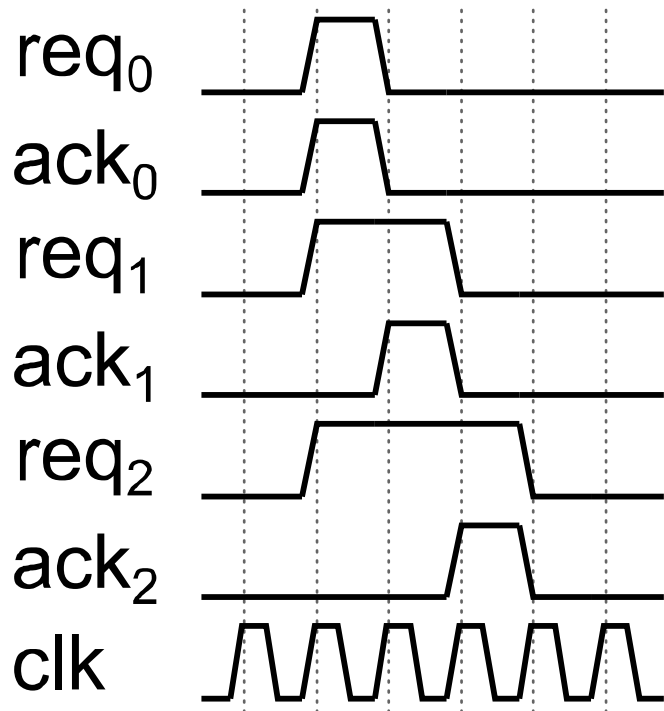
对吗?



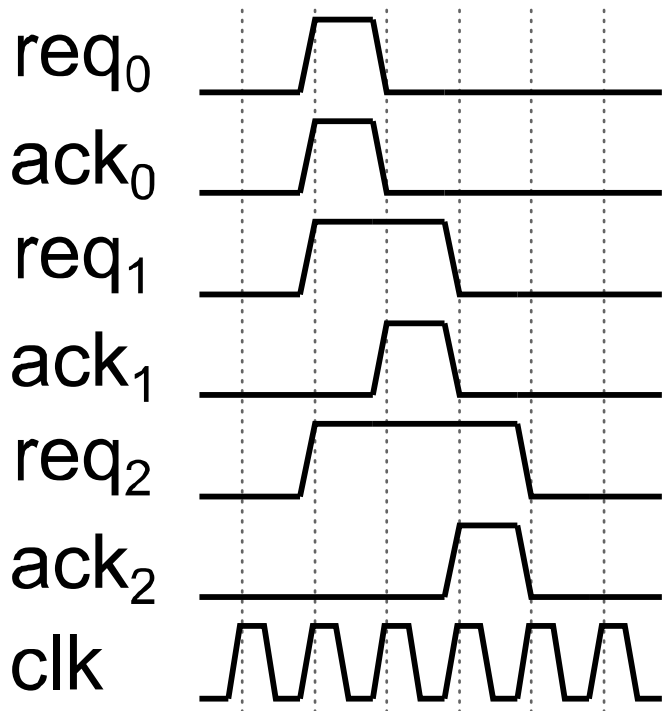
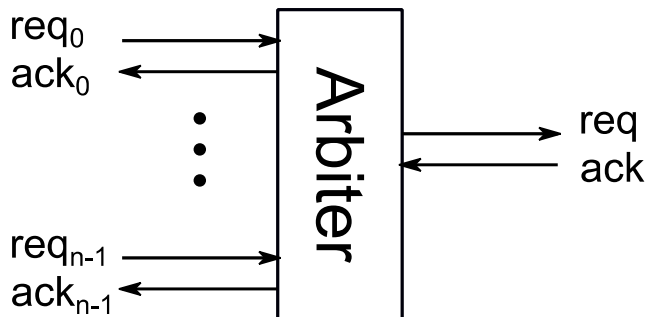
仲裁器输入



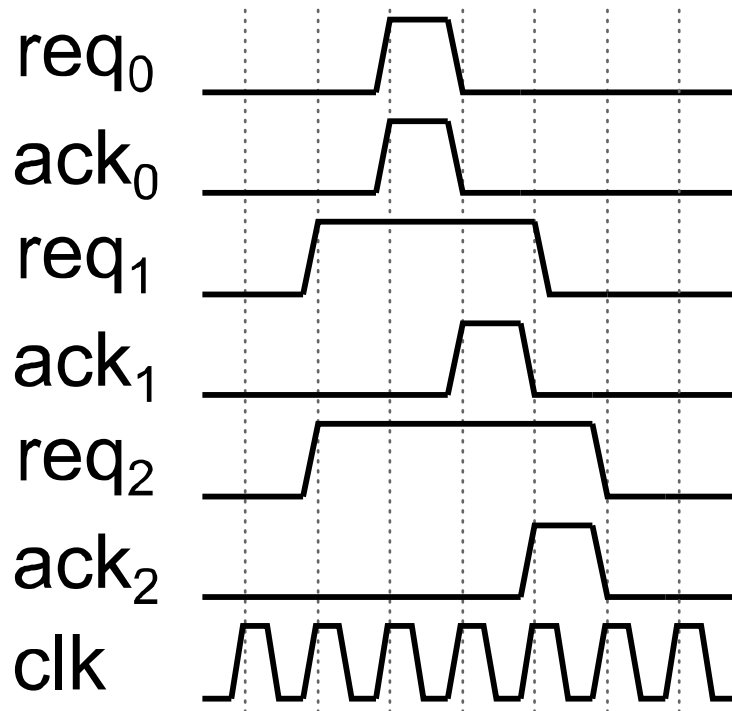
对吗?



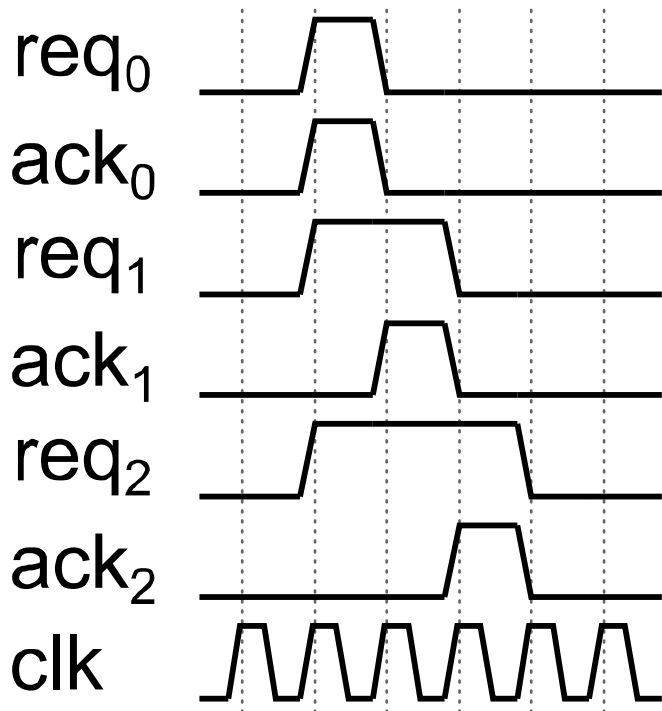
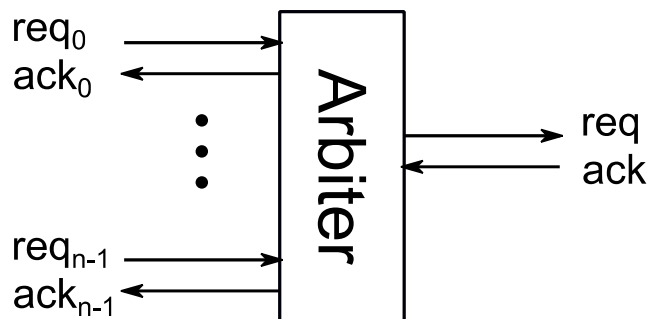
仲裁器输入



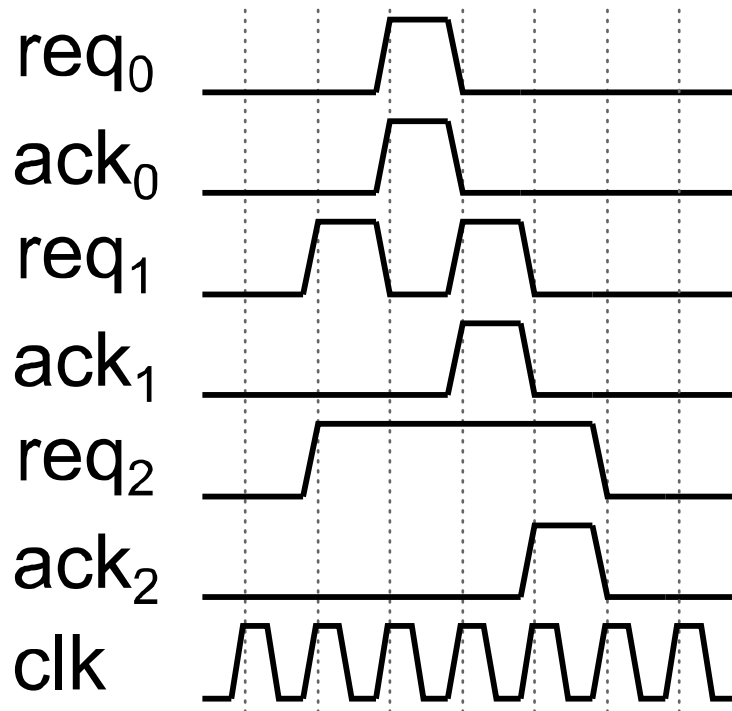
对吗?



仲裁器输入

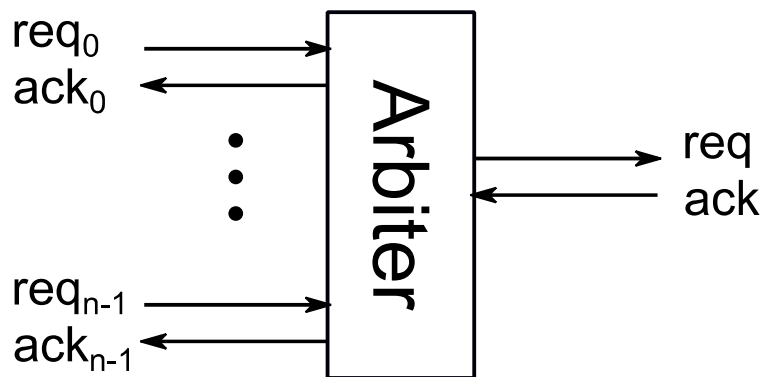


对吗?



仲裁器：仲裁规则

- 当多路请求同时有效时，如何选择哪一路请求来响应？
 - 随机选择
 - 静态优先级
 - 动态优先级
 - 循环仲裁器 (round-robin arbiter)
 - 伪随机仲裁器？



3路静态优先级仲裁器

○三路输入 (0~2) , 0路优先级最高, 2最低

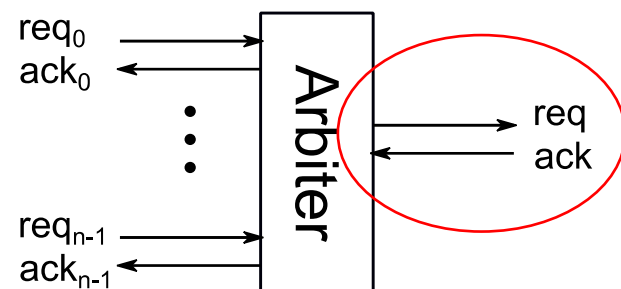
```
module ArbStatic3 (input clk, rstn,  
                  input [2:0] req,  
                  output [2:0] ack);  
    reg [2:0] ack;  
  
    always @(req) begin  
        casez(req)  
            3'b??1:  ack = 3'b001;  
            3'b?10:  ack = 3'b010;  
            3'b100:  ack = 3'b100;  
            default: ack = 3'b000;  
        endcase  
    end  
  
endmodule
```

任何情况下, 先响应req[0], 然后req[1], 最后req[2]。

3路静态优先级仲裁器(考虑输出端状态)

○如果考虑输出端可能暂时不能响应呢?

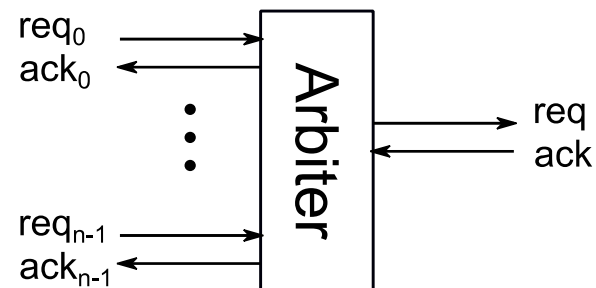
```
module ArbStatic3 (input clk, rstn,  
                  input [2:0] req_i,  
                  output [2:0] ack_i,  
                  output req_o,  
                  input ack_o);  
  
    reg [2:0] ack_i;  
  
    assign req_o = |req_i;  
  
    always @(req_i, ack_o) begin  
        if(ack_o)  
            casez(req_i)  
                3'b??1:  ack_i = 3'b001;  
                3'b?10:  ack_i = 3'b010;  
                3'b100:  ack_i = 3'b100;  
                default: ack_i = 3'b000;  
            endcase  
        else  
            ack_i = 3'b000;  
        end  
    end  
  
endmodule
```



N路静态优先级仲裁器

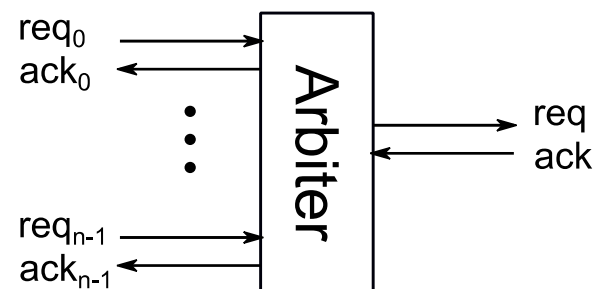
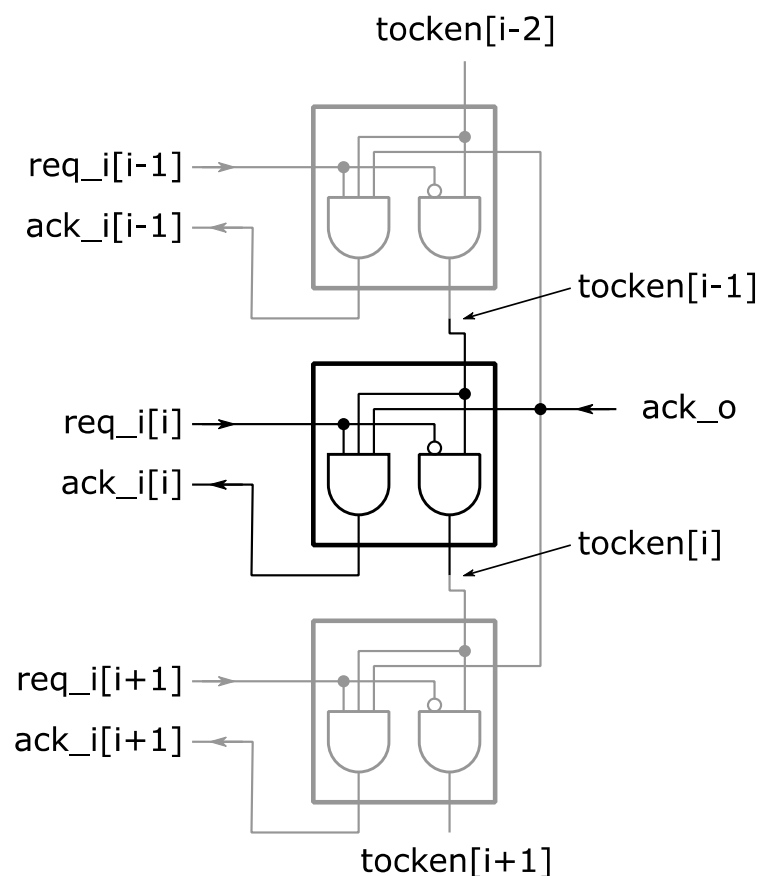
○ 我们如何从3路扩展到N路？

○ 是否可以将各路拆开，推导每一路的响应信号的表达式？



N路静态优先级仲裁器

○ 我们如何从3路扩展到N路?



$$ack_i[i] = token[i-1] \& req_i[i] \& ack_o;$$
$$token[i] = token[i-1] \& \sim req_i[i];$$

N路静态优先级仲裁器

○我们如何从3路扩展到N路?

```
module ArbStatic
#(parameter N=8)
  (input clk, rstn,
   input [N-1:0] req_i,
   output [N-1:0] ack_i,
   output req_o,
   input ack_o);
```

```
  genvar i;
```

```
  wire token[N-1:0];
```

```
  assign req_o = |req_i;
```

```
  generate for(i=0; i<N; i=i+1) begin
```

```
    if(i==0) begin
```

```
      assign ack_i[0] = req_i[0] & ack_o;
```

```
      assign token[0] = ~req_i[0];
```

```
    end else begin
```

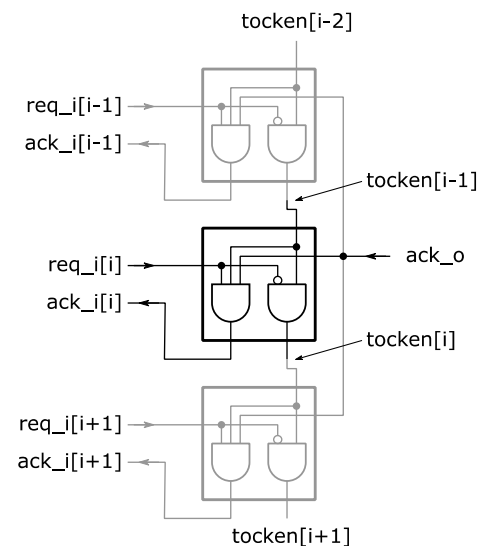
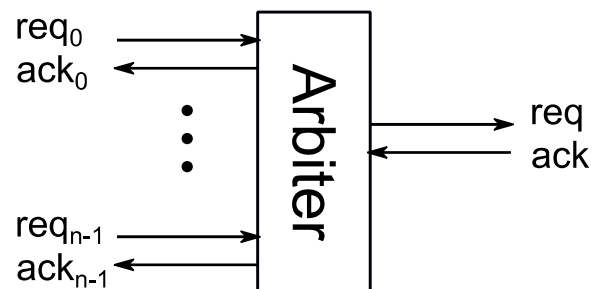
```
      assign ack_i[i] = token[i-1] & req_i[i] & ack_o;
```

```
      assign token[i] = token[i-1] & ~req_i[i];
```

```
    end
```

```
  end endgenerate
```

```
endmodule
```



N路动态优先级仲裁器

○我们如何动态设定优先级？

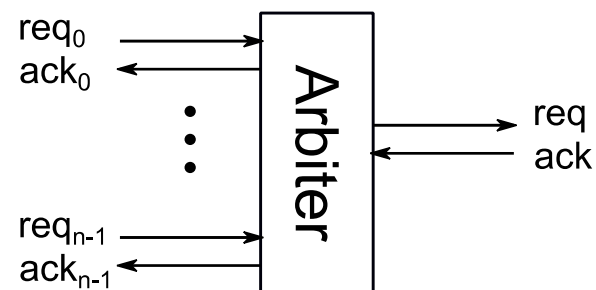
原有优先级: 7 ← 6 ← 5 ← 4 ← 3 ← 2 ← 1 ← 0
7 6 5 4 3 2 1 0

○添加一个优先级的输入端

```
prio[7:0] = 8'b0001_0000;
```

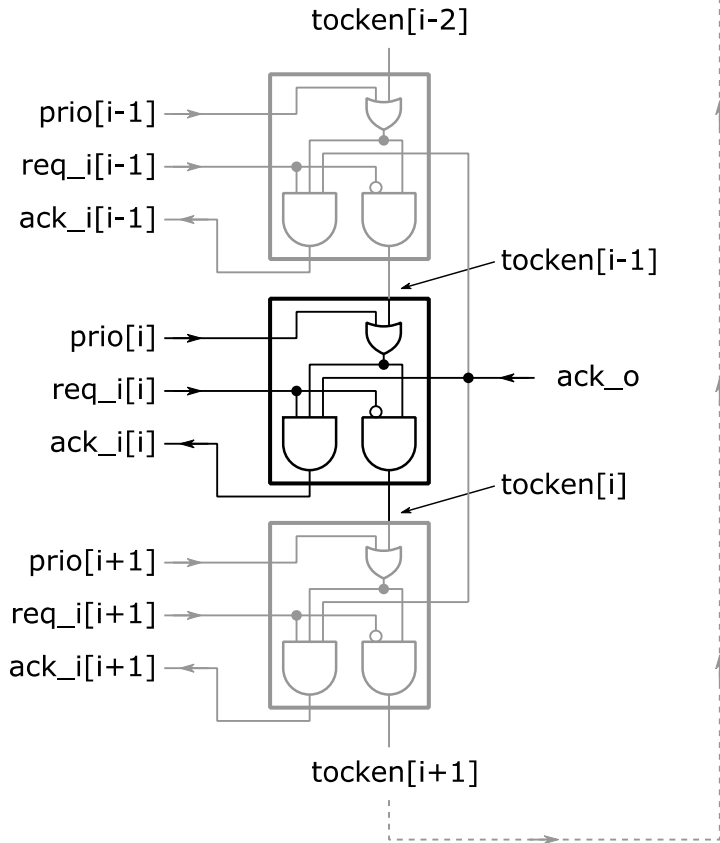
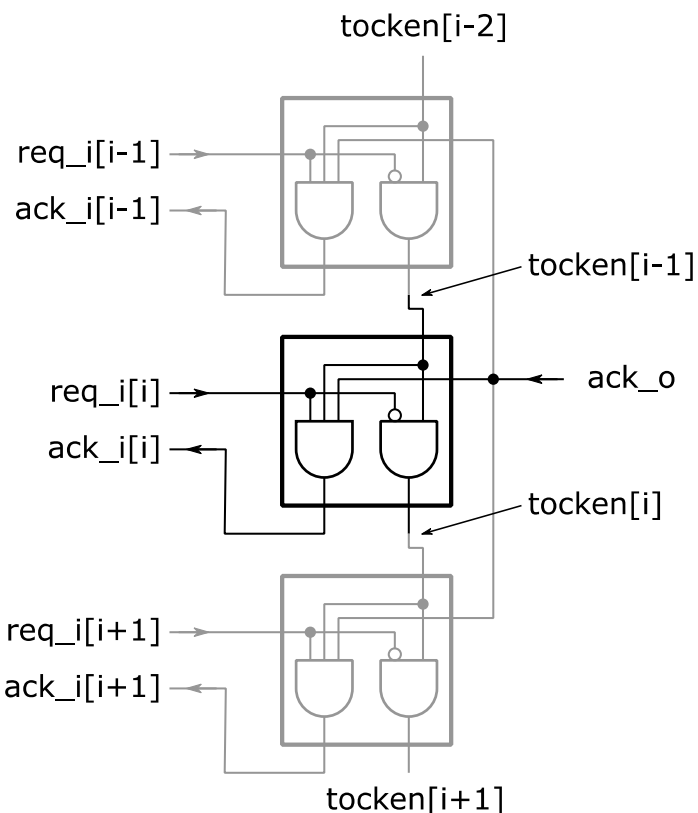
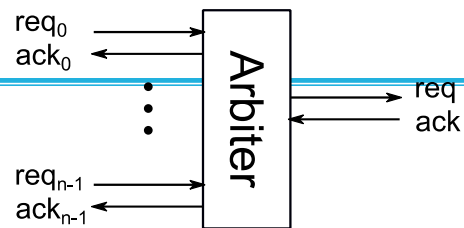
req[4] 有最高优先级

优先级排序: 3 ← 2 ← 1 ← 0 ← 7 ← 6 ← 5 ← 4
7 6 5 4 3 2 1 0



N路动态优先级仲裁器

我们如何动态设定优先级？



$$\begin{aligned} \text{ack}_i[i] &= (\text{token}[i-1] \mid \text{prio}[i]) \& \text{req}_i[i] \& \text{ack}_o; \\ \text{token}[i] &= (\text{token}[i-1] \mid \text{prio}[i]) \& \sim \text{req}_i[i] \end{aligned}$$

N路动态优先级仲裁器

```
module ArbPriority
  #(parameter N=8)
  (input clk, rstn,
   input [N-1:0] req_i,
   input [N-1:0] prio,
   output [N-1:0] ack_i,
   output req_o,
   input ack_o);
```

```
  genvar i;
  wire token[N-1:0];
```

```
  assign req_o = |req_i;
  generate for(i=0; i<N; i=i+1) begin
```

```
    if(i==0) begin
```

```
      assign ack_i[0] = (token[N-1] | prio[0]) & req_i[0] & ack_o;
```

```
      assign token[0] = (token[N-1] | prio[0]) & ~req_i[0];
```

```
    end else begin
```

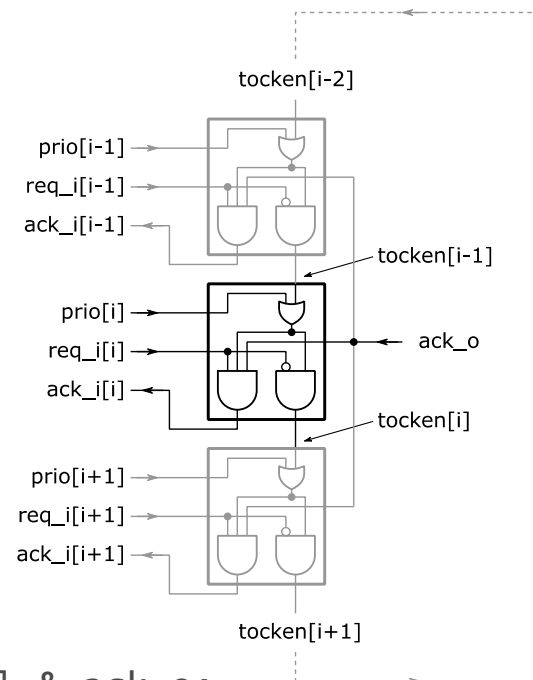
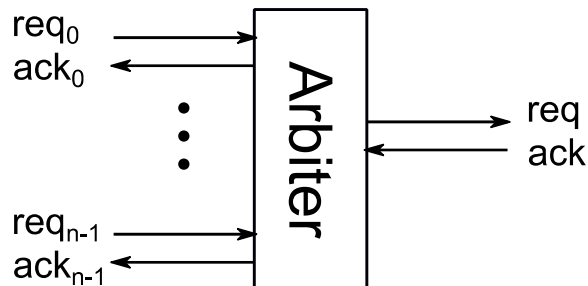
```
      assign ack_i[i] = (token[i-1] | prio[i]) & req_i[i] & ack_o;
```

```
      assign token[i] = (token[i-1] | prio[i]) & ~req_i[i];
```

```
    end
```

```
  end endgenerate
```

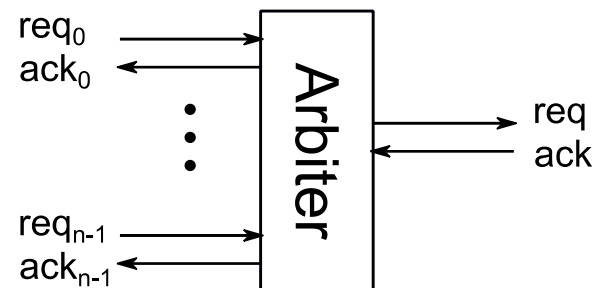
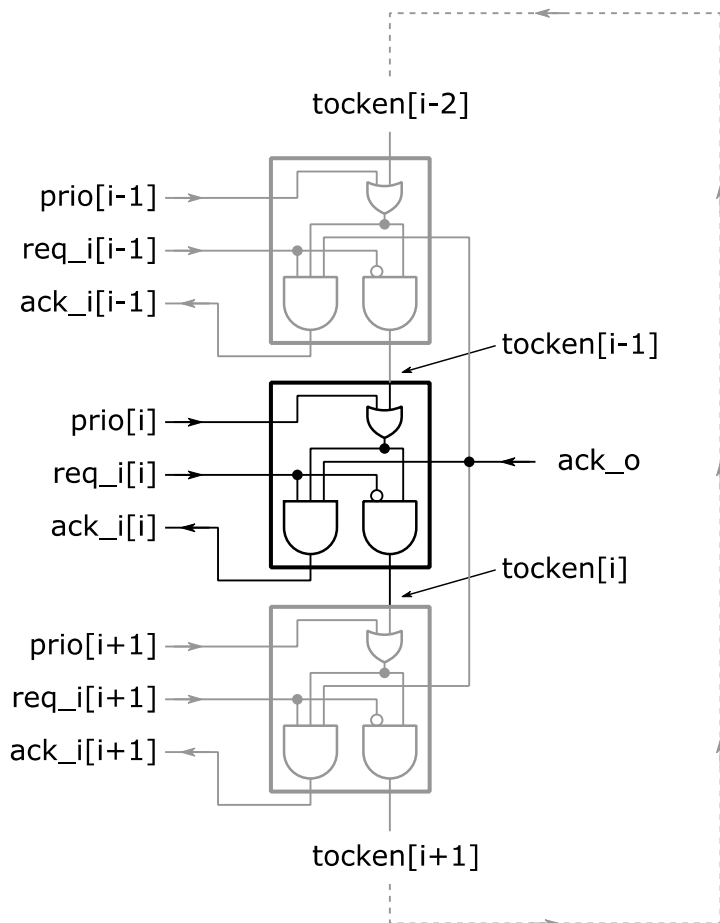
```
endmodule
```



组合逻辑环

N路动态优先级仲裁器

○ 我们如何动态设定优先级？

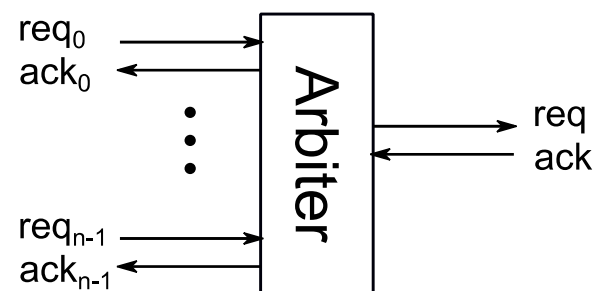
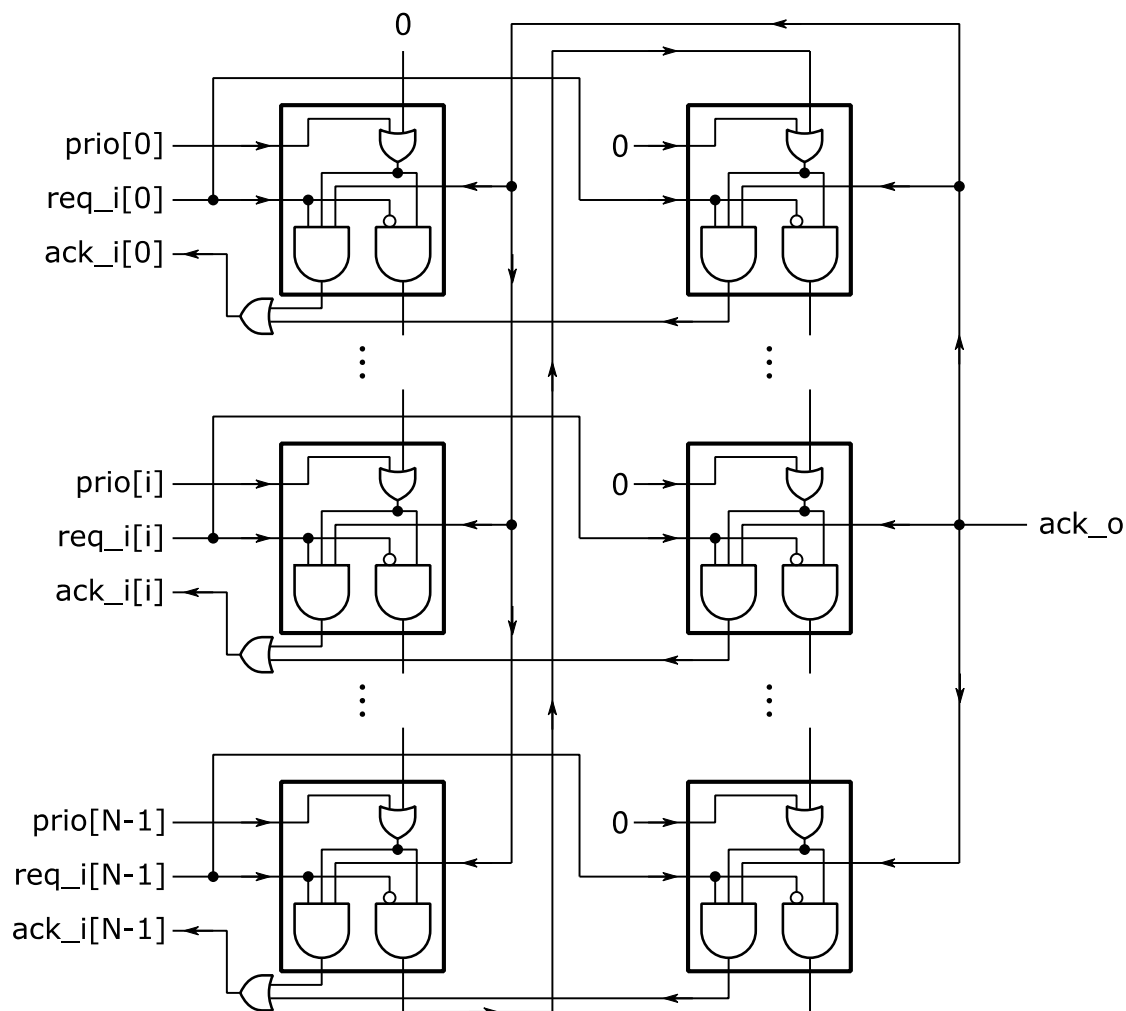


如何解决环的问题？

$$ack_i[i] = (token[i-1] \mid prio[i]) \& req_i[i] \& ack_o;$$
$$token[i] = (token[i-1] \mid prio[i]) \& \sim req_i[i]$$

N路动态优先级仲裁器

○ 我们如何动态设定优先级？

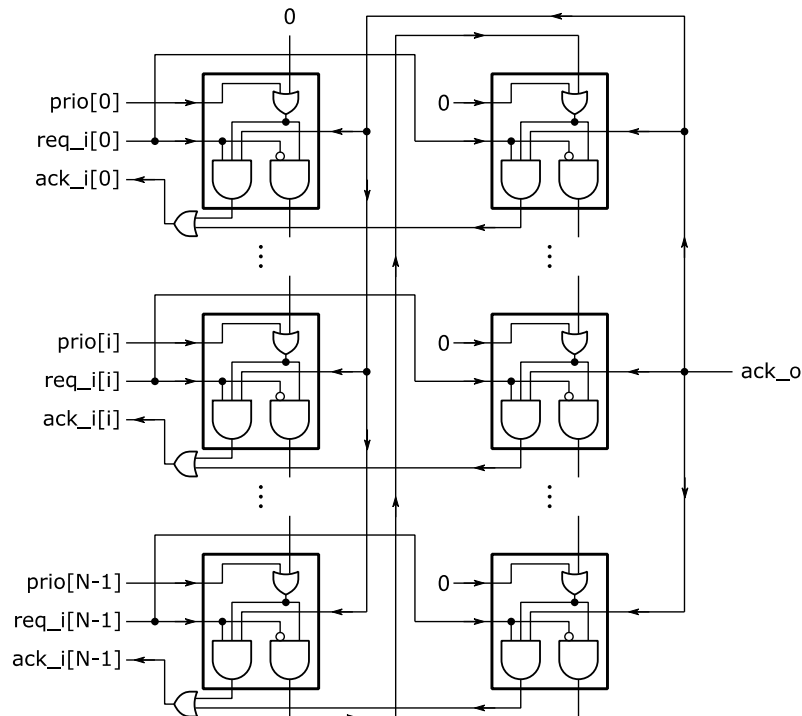


N路静态优先级仲裁器

```
module ArbPriority
  #(parameter N=8)
  (input clk, rstn,
   input [N-1:0] req_i,
   input [N-1:0] prio,
   output [N-1:0] ack_i,
   output req_o,
   input ack_o);

  genvar i;
  wire token[2*N-1:0];

  assign req_o = |req_i;
  generate for(i=0; i<N; i=i+1) begin
    if(i==0) begin
      assign ack_i[0] = (prio[0] | token[N-1]) & req_i[0] & ack_o;
      assign token[0] = prio[0] & ~req_i[0];
      assign token[N] = token[N-1] & ~req_i[0];
    end else begin
      assign ack_i[i] = (token[i-1] | prio[i] | token[N+i-1]) &
        req_i[i] & ack_o;
      assign token[i] = (token[i-1] | prio[i]) & ~req_i[i];
      assign token[N+i] = token[N+i-1] & ~req_i[i];
    end
  end
end generate
endmodule
```



N路循环优先级仲裁器

○ 循环优先级仲裁器 (round-robin arbiter)

○ 一个8路循环优先级仲裁器

第一周期: $req = 8'b01100101$

$ack = 8'b00000001$

第二周期: $req = 8'b01100100$

$ack = 8'b00000100$

第三周期: $req = 8'b01100010$

$ack = 8'b00100000$

第四周期: $req = 8'b01001010$

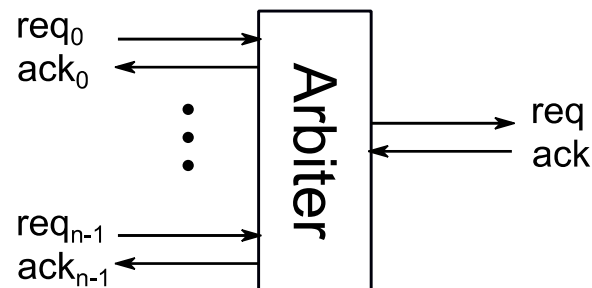
$ack = 8'b01000000$

第五周期: $req = 8'b01001010$

$ack = 8'b00000010$

第六周期: $req = 8'b01001001$

$ack = 8'b00001000$



当多个请求同时有效时，循环响应。

N路循环优先级仲裁器

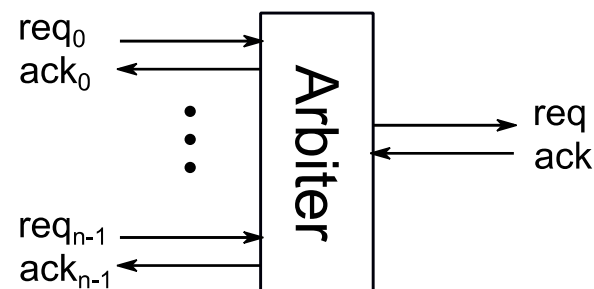
○循环优先级仲裁器 (round-robin arbiter)

```
module ArbRR #(parameter N=8)
  (input clk, rstn,
   input [N-1:0] req_i,
   output [N-1:0] ack_i,
   output req_o,
   input ack_o);
```

```
  genvar i;
  wire token[2*N-1:0];
  reg [N-1:0] prio;
```

```
  always @(posedge clk or negedge rstn)
    if(~rstn) prio <= 1;
    else if(!ack_i) prio <= {ack_i[N-2:0], ack_i[N-1]};
```

```
  assign req_o = |req_i;
  generate for(i=0; i<N; i=i+1) begin
    if(i==0) begin
      assign ack_i[0] = (prio[0] | token[N-1]) & req_i[0] & ack_o;
      assign token[0] = prio[0] & ~req_i[0];
      assign token[N] = token[N-1] & ~req_i[0];
    end else begin
      assign ack_i[i] = (token[i-1] | prio[i] | token[N+i-1]) & req_i[i] & ack_o;
      assign token[i] = (token[i-1] | prio[i]) & ~req_i[i];
      assign token[N+i] = token[N+i-1] & ~req_i[i];
    end
  end endgenerate
endmodule
```



○仲裁器

○仲裁器的概念

○仲裁器的类型

- 静态优先级

- 循环优先级

○仲裁器的设计

- 拆分，按位推逻辑表达式

- 拆解组合逻辑环

○问题

- 循环优先级仲裁器是否一定是公平的？

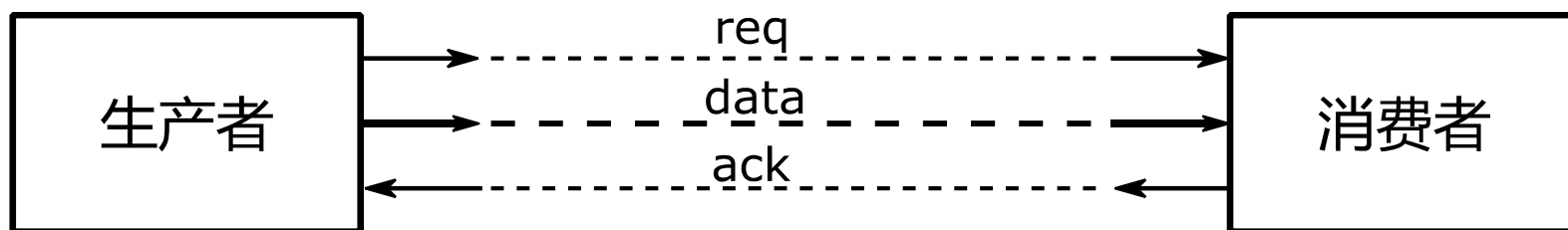
- 如何实现一个随机仲裁器？

复杂时序逻辑电路设计：FIFO缓冲

○FIFO缓冲 (Buffer)

○当一个数据流的产生者和消费者的步调不统一，直接连接会导致数据吞吐率下降

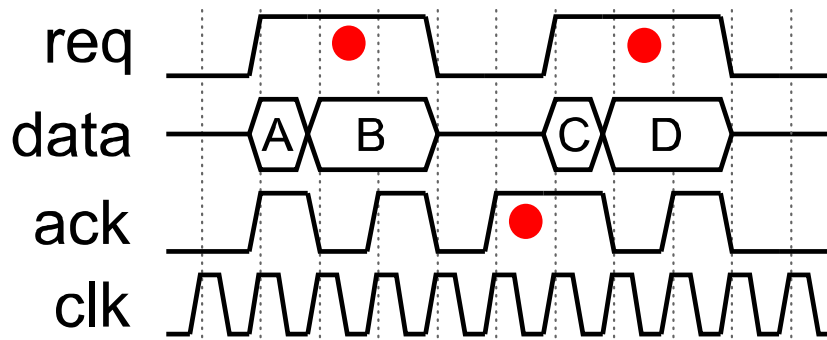
- 生产者产生了数据消费者不能立即接收
- 消费者可以消费数据却没有



生产者空闲2个周期产生2个数据。

消费者每空闲1个周期接受1个数据。

直接连接：6个周期传输2个数。



复杂时序逻辑电路设计：FIFO缓冲

○FIFO缓冲 (Buffer)

○在这种情况下，插入一个FIFO缓冲可以协调两边的步调

- 生产者产生数据消费者不能接受时，先保存在缓冲器内。
- 当消费者可以消费数据而生产者没有数据时，可以从缓冲区拿数。

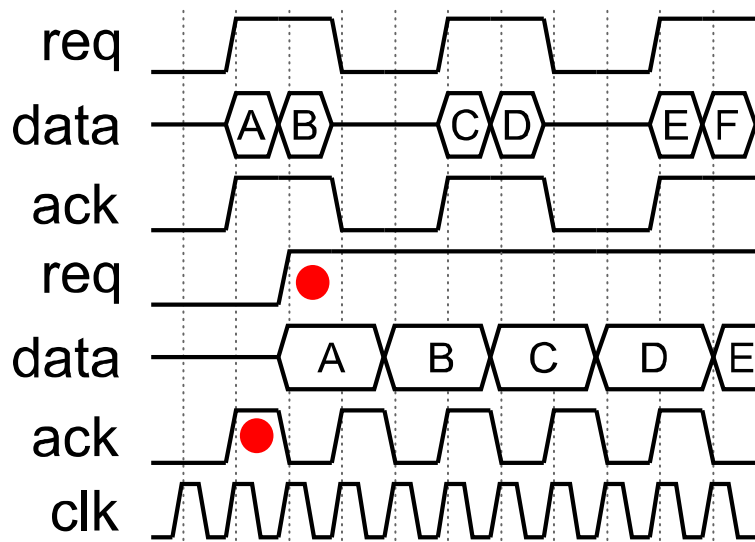


假设缓冲区有至少2个数据的空间

缓冲区可以在缓冲区未滿时随时接受数据

缓冲区可以在有数据时一直输出数据

数据传输：4个周期传输2个数



深度为2的FIFO缓冲

让我们暂时只考虑输入端和数据输出选择

```
module fifo2 #(parameter dw=8) ( input clk, rstn,  
    input [dw-1:0] d_in, input req_in, output ack_in,  
    output [dw-1:0] d_out, output req_out, input ack_out);
```

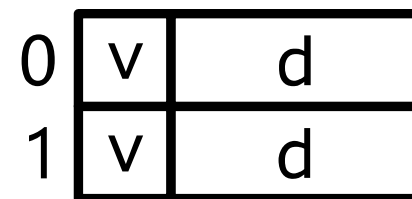
```
    reg [dw-1:0] data [1:0];  
    reg [1:0] valid;
```

```
    assign req_out = |valid;  
    assign ack_in = ~&valid;
```

```
    always @(posedge clk or negedge rstn)  
        if(~rstn) valid <= 2'b00;  
        else if(req_in)  
            case(valid)  
                2'b00: begin valid <= 2'b01; data[0] <= d_in; end  
                2'b01: begin valid <= 2'b11; data[1] <= d_in; end  
                2'b10: begin valid <= 2'b11; data[0] <= d_in; end  
                default: // do nothing  
            endcase
```

```
    reg [dw-1:0] d_out;  
    always @(data, valid)  
        case(valid)  
            2'b00, 2'b01: d_out = data[0];  
            2'b10: d_out = data[1];  
            2'b11: ??? // 无法区分了!  
        endcase
```

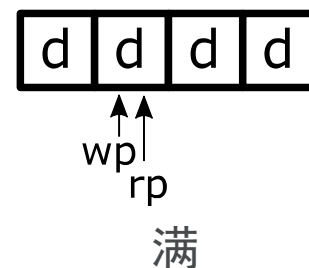
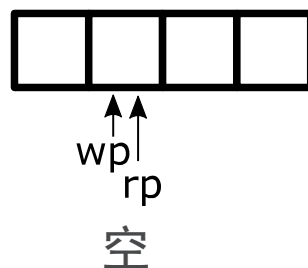
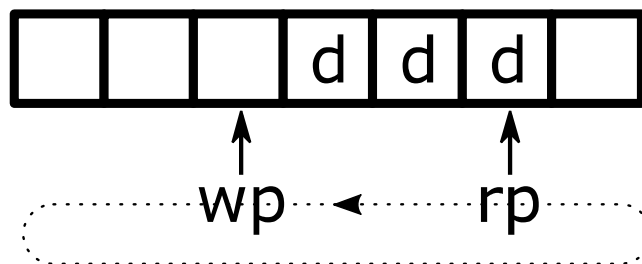
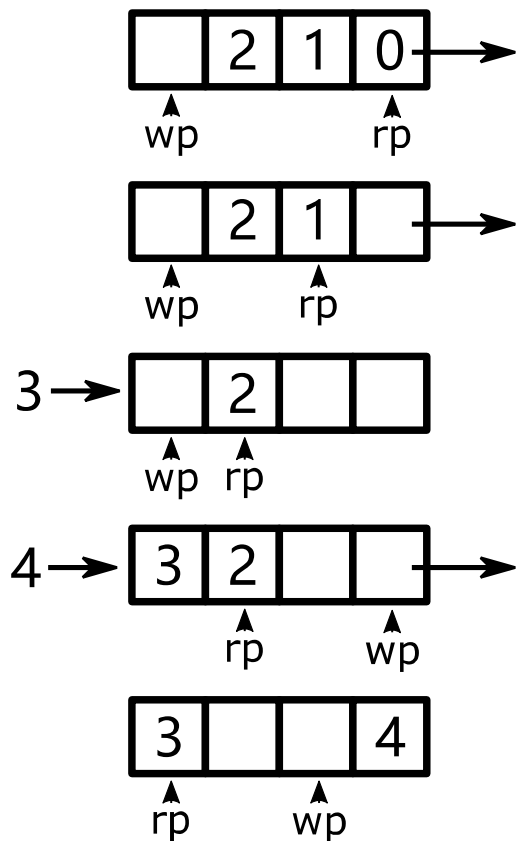
```
endmodule
```



仅依赖于缓冲区数据的状态位还不足以完成控制。

深度为L的FIFO缓冲

我们需要引入指针!



数据存入: wp 自增, 数据保存并使能。
数据读出: rp 自增, 数据读出并作废。
缓冲区满: wp 等于 rp 并且 $valid[wp]$ 有效。
缓冲区空: wp 等于 rp 并且 $valid[rp]$ 无效。

FIFO最小延迟1周期。

深度为L的FIFO缓冲

我们需要引入指针!

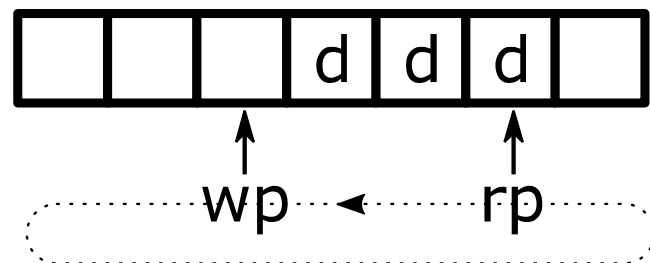
```
module fifo2 #(parameter dw=8, L=7)
  ( input clk, rstn,
    input [dw-1:0] d_in, input req_in, output ack_in,
    output [dw-1:0] d_out, output req_out, input ack_out);

  reg [dw-1:0] data [L-1:0];
  reg [L-1:0] valid;
  reg [L-1:0] wp, rp;

  assign d_out = data[rp];
  assign req_out = valid[rp];
  assign ack_in = ~valid[wp];

  always @(posedge clk or negedge rstn)
    if(~rstn) begin
      wp <= 0; rp <= 0; valid <= 0;
    end else begin
      if(req_in & ack_in) begin
        wp <= wp == L-1 ? 0 : wp + 1;
        valid[wp] <= 1'b1;
        data[wp] <= d_in;
      end

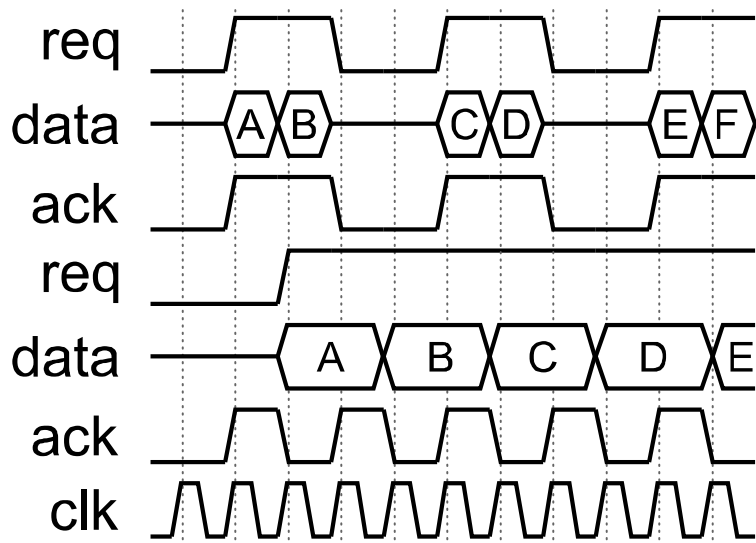
      if(req_out & ack_out) begin
        rp <= rp == L-1 ? 0 : rp + 1;
        valid[rp] <= 1'b0;
      end
    end
end
endmodule
```



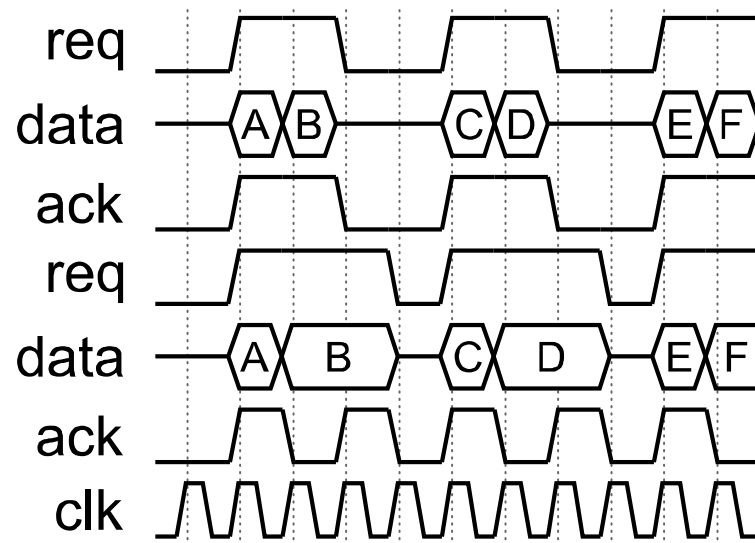
数据存入: wp自增, 数据保存并使能。
数据读出: rp自增, 数据读出并作废。
缓冲区满: wp等于rp并且valid[wp]有效。
缓冲区空: wp等于rp并且valid[rp]无效。

FIFO最小延迟1周期。

是否可以将最小延迟将为0周期?



1周期传输延时
缓冲深度至少为2



0周期传输延时
缓冲深度至少为1

深度为L的FIFO缓冲：0最小延时

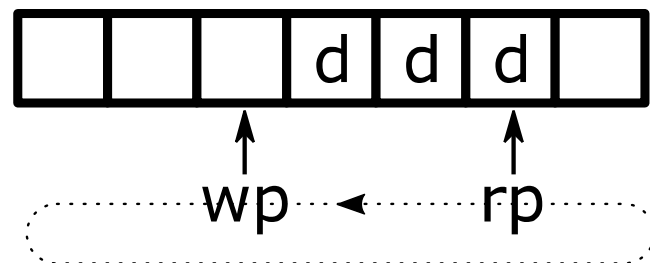
```
module fifo0 #(parameter dw=8, L=7)
( input clk, rstn,
  input [dw-1:0] d_in, input req_in, output ack_in,
  output [dw-1:0] d_out, output req_out, input ack_out);

  reg [dw-1:0] data [L-1:0];
  reg [L-1:0] valid;
  reg [L-1:0] wp, rp;
  wire empty;

  assign empty = (wp == rp) & ~valid[rp];
  assign d_out = empty ? d_in : data[rp];
  assign req_out = empty ? req_in : valid[rp];
  assign ack_in = ~valid[wp];

  always @(posedge clk or negedge rstn)
    if(~rstn) begin
      wp <= 0; rp <= 0; valid <= 0;
    end else begin
      if(req_in & ack_in & (~empty | ~ack_out))
        begin
          wp <= wp == L-1 ? 0 : wp + 1;
          valid[wp] <= 1'b1;
          data[wp] <= d_in;
        end

      if(req_out & ack_out & ~empty) begin
        rp <= rp == L-1 ? 0 : rp + 1;
        valid[rp] <= 1'b0;
      end
    end
end
endmodule
```



数据穿越缓冲：
生产者和消费者同时活跃，
同时缓冲为空。

FIFO缓冲器总结

○ FIFO缓冲器

○ 缓冲器的作用

○ 指针型缓冲器的实现

- 如何确定缓冲器的深度
- 空/满的条件
- 如何降低最小延迟

○ 问题

○ 缓冲器的时序问题

- (等我们讲完了时序逻辑电路的时序分析)

时序逻辑电路的时序分析

○ 已有知识

○ 组合逻辑电路的时序分析

- 电路的关键路径
- 电路的延时计算

○ 寄存器的时序分析

- 寄存器的保持和建立时间
- 寄存器的传输延时

○ 时序逻辑电路的时序分析

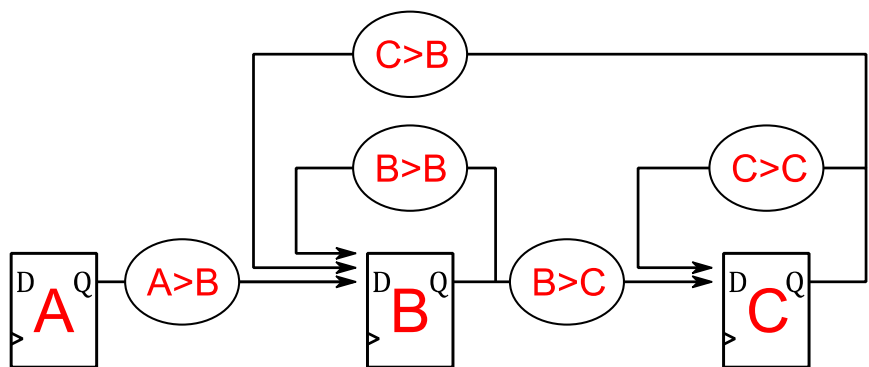
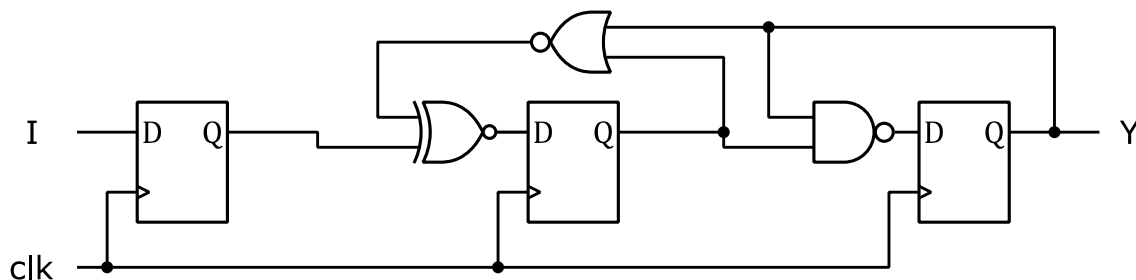
○ 分析带寄存器的电路的时间特性

○ 目的:

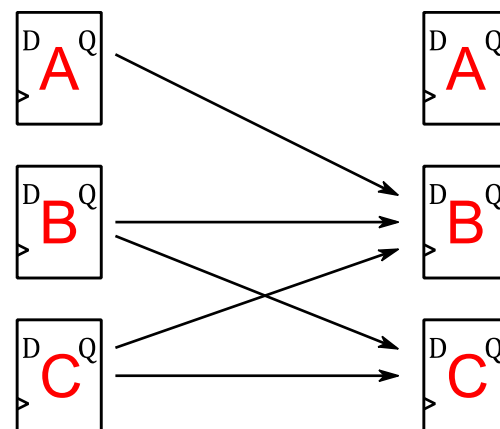
- 电路的最高频率
- 电路是否能正常工作

时序逻辑电路的简化电路模型

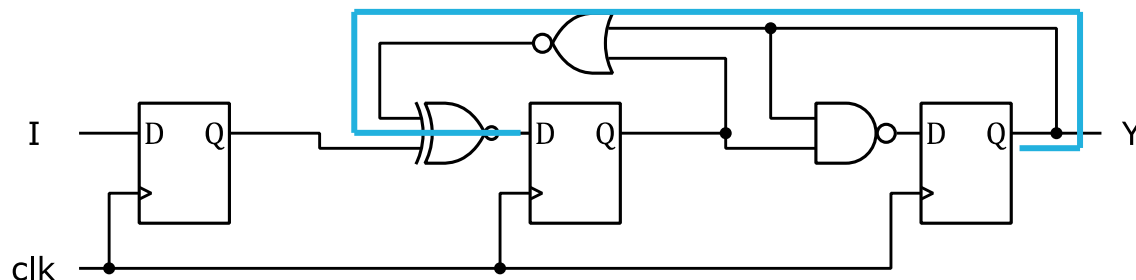
○ 我们如何能知道最高频率?



$X > Y$: 从X到Y的路径群。



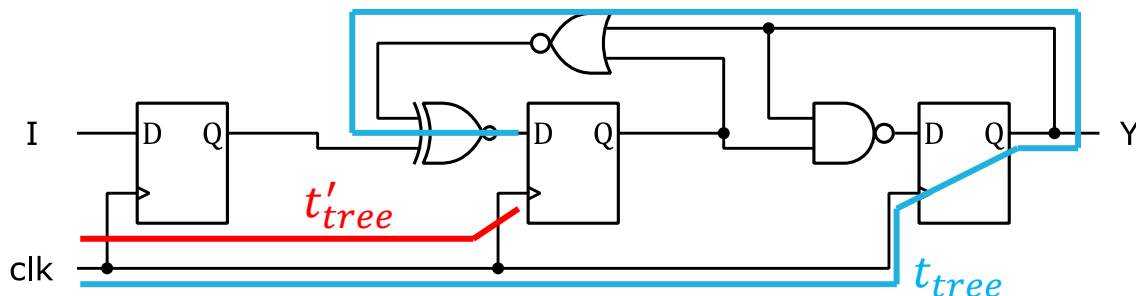
分析一条路径的延时



- $t = t_p(NOR) + t_p(NXOR)$

- 但是该时间还不能直接和时钟频率结合起来

通过路径延时分析时钟频率



○我们知道寄存器建立时间 t_{su} 和传输时间 t_{pd} 的概念

$$t_{tree} + t_{pd}(R) + t_p(NOR) + t_p(NXOR) + t_{su} < t'_{tree} + T$$

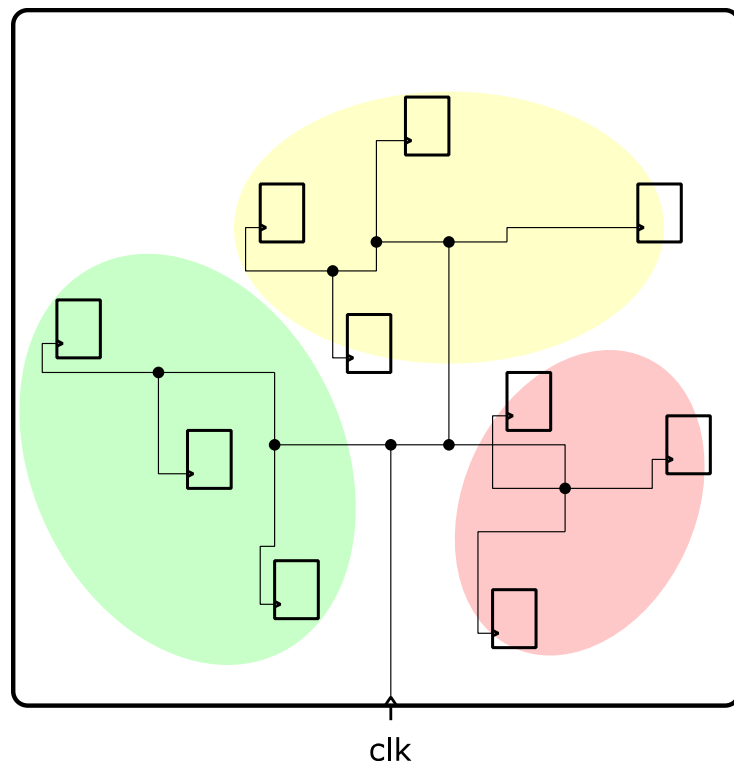
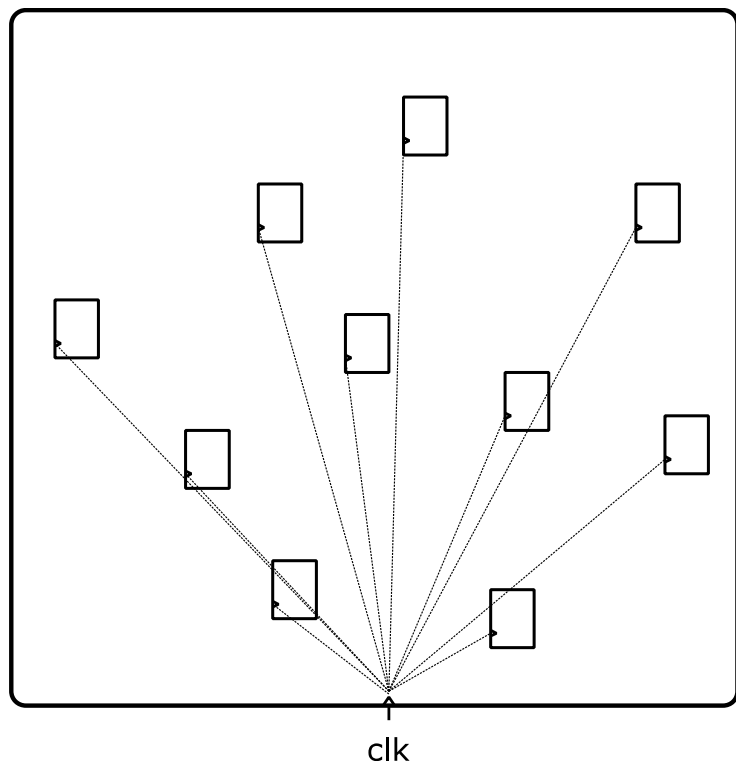
所以

$$T_{min} > \underbrace{(t_{tree} - t'_{tree})}_{\text{时钟树偏差}} + \underbrace{t_{su}}_{\text{建立时间}} + \underbrace{t_{pd}(R)}_{\text{传输时间}} + \underbrace{t_p(NOR) + t_p(NXOR)}_{\text{路径延时}}$$

$$f_{max} = \frac{1}{T_{min}}$$

时钟树的构造与平衡

○ 时钟树的平衡，影响电路的频率

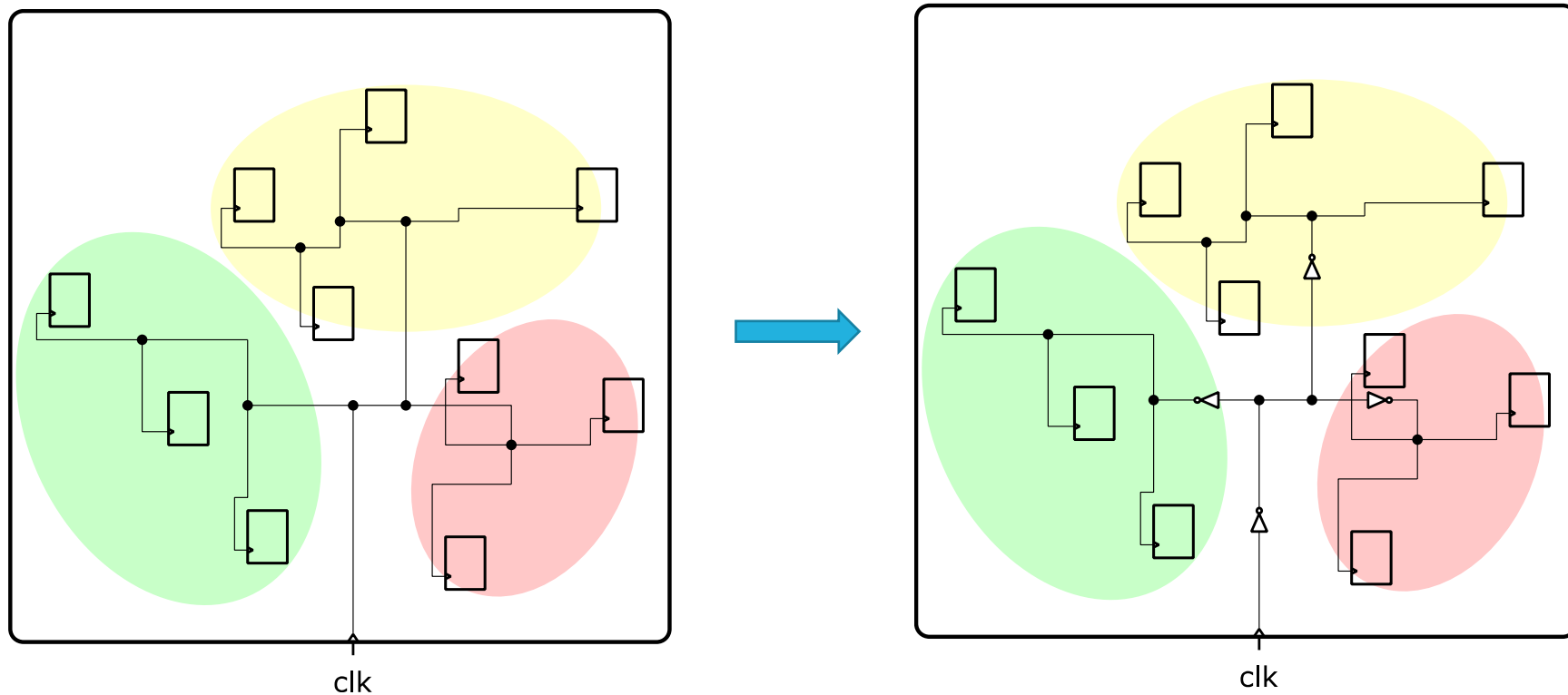


- 将时钟引到芯片中间
- 寄存器分片，片内时钟继续均匀分布
- 形成一个树形结构

时钟信号的负载很大，
导致翻转时间长！

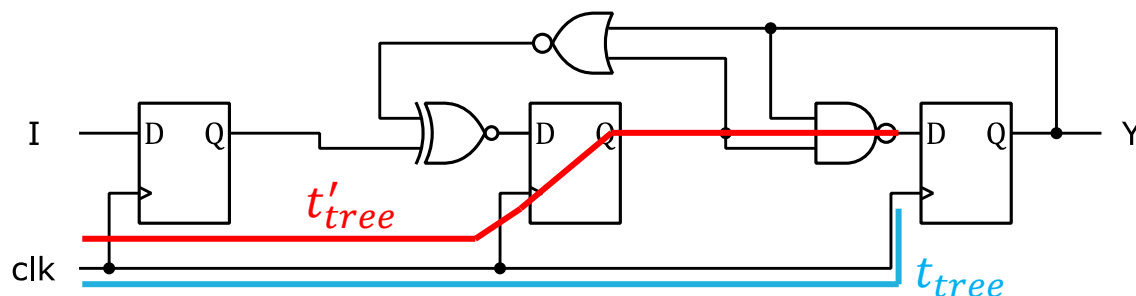
时钟树的构造与平衡

○ 时钟树的平衡，影响电路的频率



- 在时钟树上添加缓冲器（反门）
- 减小时钟信号的翻转时间，增大其驱动能力

寄存器正常工作的条件：保持时间



○我们知道寄存器保持时间 t_h 的概念

$$t_{tree} + t_h < t'_{tree} + t_{pd}(R) + t_p(NAND)$$

所以

$$\underbrace{t_p(NAND)}_{\text{路径延时}} > \underbrace{(t_{tree} - t'_{tree})}_{\text{时钟树偏差}} + \underbrace{t_h(R)}_{\text{保持时间}} - \underbrace{t_{pd}(R)}_{\text{传输时间}}$$

路径延时也不能太短，否则寄存器不能正常工作。

时序逻辑电路时序分析总结

○ 如何分析时序逻辑电路

- 将电路化简为时序路径

- 最长的时序路径决定了电路的速度

○ 分析最长时序路径

$$T_{min} > (t_{tree} - t'_{tree}) + t_{su} + t_{pd}(R) + t_{path}$$

○ 时钟树

- 时钟树的平衡是为了降低时钟偏差($t_{tree} - t'_{tree}$)

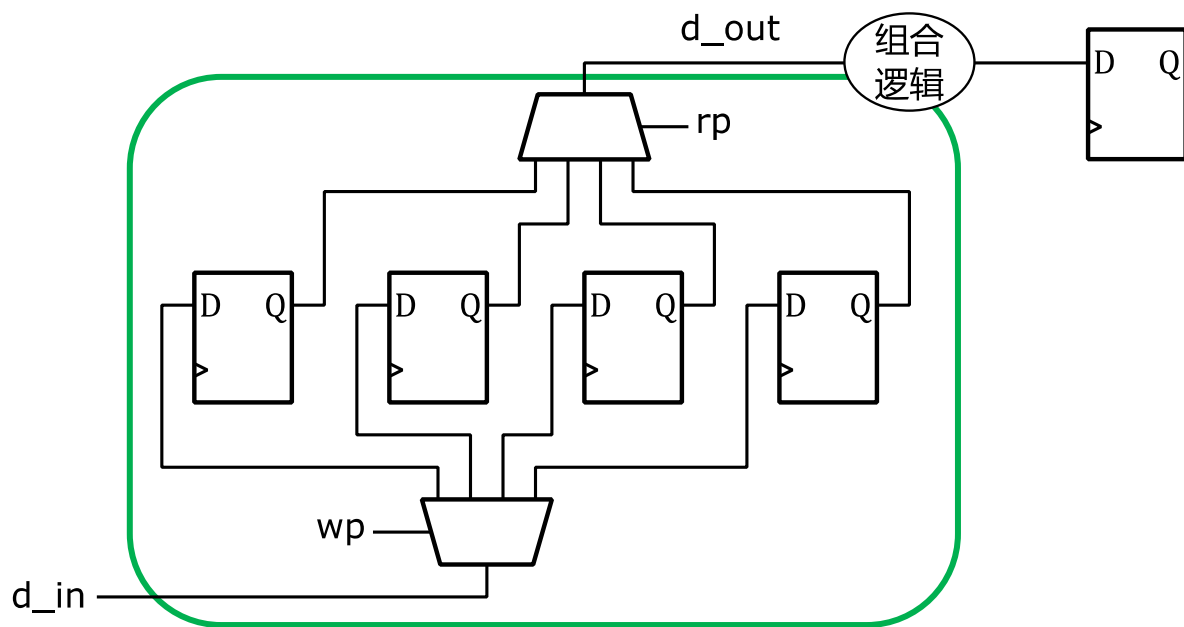
- 时钟树上使用缓冲器（反门）提高时钟的驱动能力

○ 时序电路正常工作的条件

$$t_{path} > (t_{tree} - t'_{tree}) + t_h(R) - t_{pd}(R)$$

FIFO缓冲器的时序问题

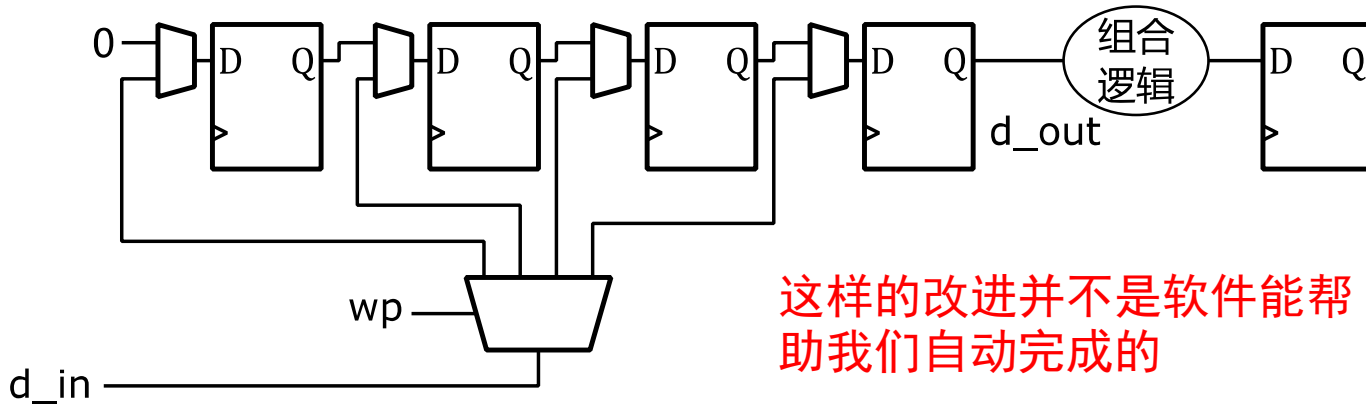
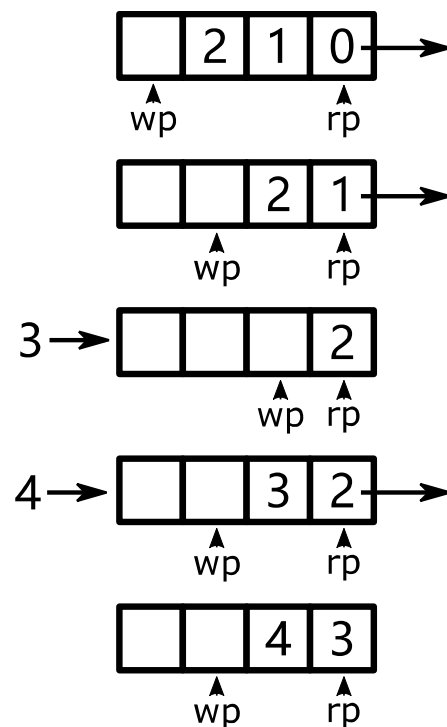
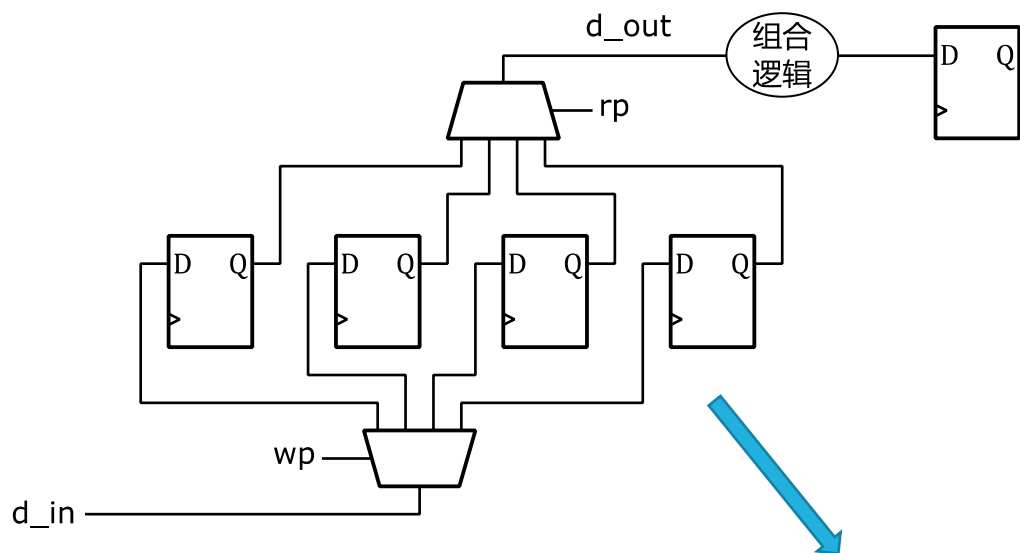
○如果FIFO的输出级时序紧张怎么办？



其实这就是我们的FIFO

FIFO缓冲器的时序问题

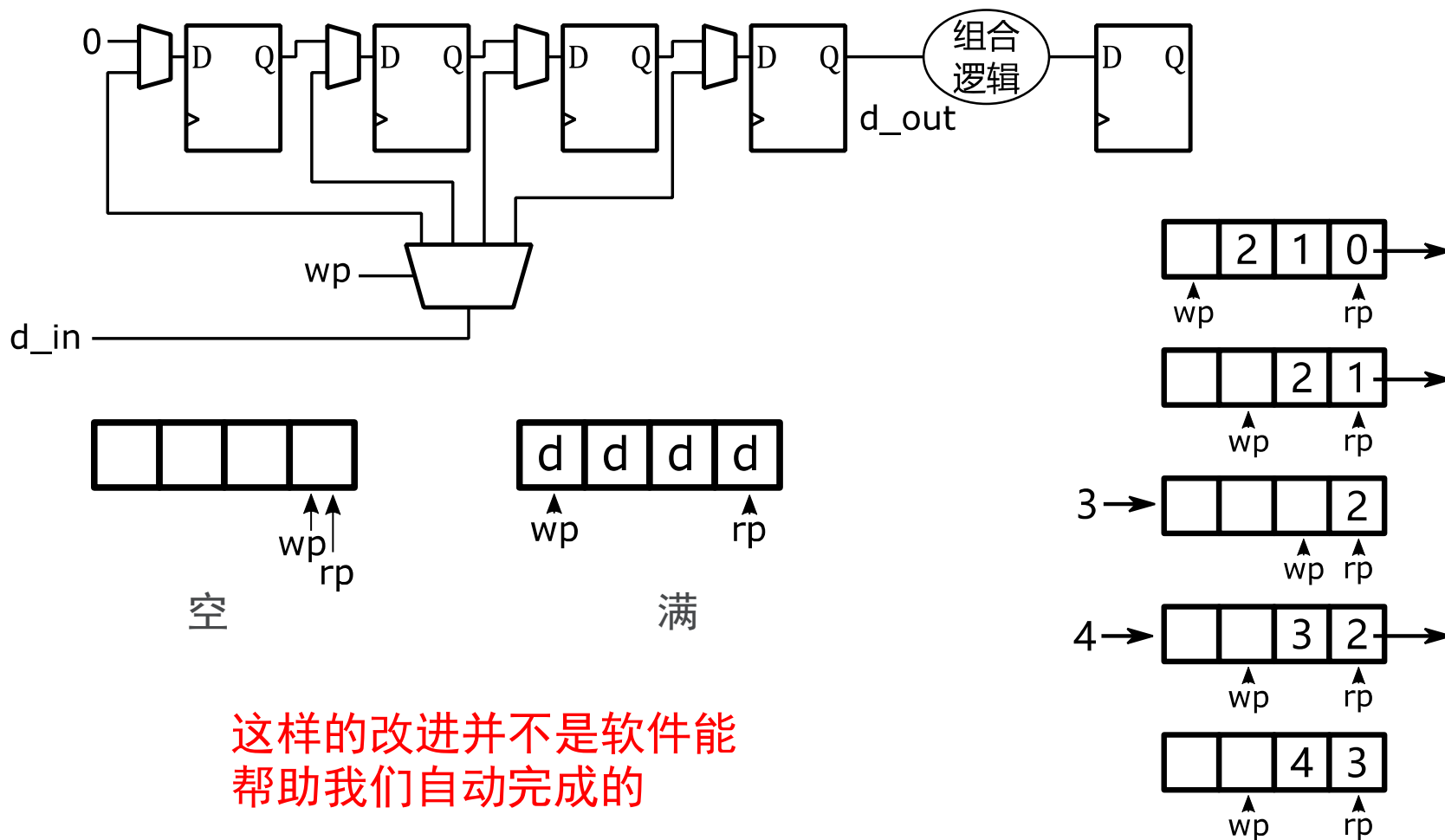
○ 如果FIFO的输出级时序紧张怎么办？



这样的改进并不是软件能帮助我们自动完成的

FIFO缓冲器的时序问题

○如果FIFO的输出级时序紧张怎么办？



这样的改进并不是软件能帮助我们自动完成的

带自动结果检验的测试程序

○ 测试输出级寄存器的FIFO

```
`timescale 1ns/1ps
module test;
  parameter dw = 16, L = 8;
  reg clk, rstn;
  reg [dw-1:0] local_fifo [L:0];
  reg [L:0] local_valid = 0;
  integer rp = 0, wp = 0;
  integer data_num = 0;
  integer full_num = 0;
  integer empty_num = 0;
  reg [dw-1:0] d_in;
  reg req_in = 1'b0;
  wire ack_in, req_out;
  wire [dw-1:0] d_out;
  reg ack_out = 1'b1;

  // DUT 实例化
  fifoRO_norm #(dw, L) dut(clk, rstn, d_in, req_in, ack_in, d_out, req_out, ack_out);

  // 时钟驱动

  // 激励生成

  // 结果自动检测/报错

  initial begin
    $dumpfile("test.vcd");
    $dumpvars(0);
    #1000000 $finish();
  end
endmodule
```

用于检测的内部FIFO

测试类型统计（功能覆盖）

带自动结果检验的测试程序

○ 测试输出级寄存器的FIFO

```
// DUT 实例化
fifoRO_norm #(dw, L) dut(clk, rstn, d_in, req_in, ack_in, d_out, req_out, ack_out);

// 时钟驱动
initial begin
    clk = 0; rstn = 0;
    #13 rstn = 1;
    forever #5 clk = ~clk;
end

// 激励生成
always @(posedge clk) begin // 随机激励
    req_in <= $random;
    d_in <= $random;
    ack_out <= $random;
end
```

带自动结果检验的测试程序

○测试输出级寄存器的FIFO

// 结果自动检测/报错

```
always @(negedge clk) begin // checks
    if(req_in & ack_in) begin
        local_fifo[wp] = d_in;
        local_valid[wp] = 1'b1;
        wp = wp == L ? 0 : wp + 1;
        data_num = data_num + 1;
    end

    if(req_in & ~ack_in) begin
        full_num = full_num + 1;
        $display($stime, ": Recorded a full FIFO.");
    end

    if(req_out & ack_out) begin
        if(local_fifo[rp] != d_out) begin
            $display($stime, ": FIFO produce a WRONG number!\n");
            #1 $finish();
        end else begin
            local_valid[rp] = 1'b0;
            rp = rp == L ? 0 : rp + 1;
        end
    end

    if(~req_out & ack_out) begin
        empty_num = empty_num + 1;
        $display($stime, ": Recorded an empty FIFO.\n");
    end

    if(data_num > 50 & empty_num > 5 & full_num > 5) begin
        $display($stime, ": Seems enough tests have been done, finish.\n");
        $finish();
    end
end
```

带自动结果检验的测试程序

○ 测试输出级寄存器的FIFO

```
23: Recorded an empty FIFO.  
33: Recorded an empty FIFO.  
233: Recorded an empty FIFO.  
753: Recorded an empty FIFO.  
763: Recorded an empty FIFO.  
773: Recorded an empty FIFO.  
803: Recorded an empty FIFO.  
833: Recorded an empty FIFO.  
843: Recorded an empty FIFO.  
1163: Recorded a full FIFO.  
1303: Recorded a full FIFO.  
1333: Recorded a full FIFO.  
1343: Recorded a full FIFO.  
1673: Recorded an empty FIFO.  
2023: Recorded an empty FIFO.  
2123: Recorded an empty FIFO.  
2133: Recorded an empty FIFO.  
2143: Recorded an empty FIFO.  
  
. . . . .  
  
5063: Recorded a full FIFO.  
5113: Recorded a full FIFO.  
5113: Seems enough tests have been done, finish.
```

带自动结果检验的测试程序

○使用SystemVerilog的验证特性

```
`timescale 1ns/1ps
module test;
  parameter dw = 16, L = 8;
  reg clk, rstn;
  reg [dw-1:0] local_fifo [$];
  integer data_num = 0;
  integer full_num = 0;
  integer empty_num = 0;
  reg [dw-1:0] d_in;
  reg req_in = 1'b0;
  wire ack_in, req_out;
  wire [dw-1:0] d_out;
  reg ack_out = 1'b1;

  // DUT 实例化
  fifoRO_norm #(dw, L) dut(clk, rstn, d_in, req_in, ack_in, d_out, req_out, ack_out);

  // 时钟驱动

  // 激励生成

  // 结果自动检测/报错

  initial begin
    $dumpfile("test.vcd");
    $dumpvars(0);
    #1000000 $finish();
  end
endmodule
```

用于检测的内部FIFO

测试类型统计（功能覆盖）

带自动结果检验的测试程序

○使用SystemVerilog的验证特性

// 结果自动检测/报错

```
always @(negedge clk) begin // checks
  if(req_in & ack_in) begin
    local_fifo.push_back(d_in);
    data_num = data_num + 1;
  end

  if(req_in & ~ack_in) begin
    full_num = full_num + 1;
    $display($stime, ": Recorded a full FIFO.");
  end

  if(req_out & ack_out) begin
    if(local_fifo.pop_front() != d_out) begin
      $display($stime, ": FIFO produce a WRONG number!\n");
      #1;
      $finish();
    end
  end

  if(~req_out & ack_out) begin
    empty_num = empty_num + 1;
    $display($stime, ": Recorded an empty FIFO.\n");
  end

  if(data_num > 50 & empty_num > 5 & full_num > 5) begin
    $display($stime, ": Seems enough tests have been done, finish.\n");
    $finish();
  end
end
```


○综合的概念

综合： synthesis

逻辑综合： Logic synthesis, 将RTL级别或更高级别的电路描述按照逻辑等效和优化的方式, 转化为等效的门级电路实现（网表）的过程。

物理综合： Physical synthesis, 在逻辑综合的基础上, 利用布局布线信息更进一步优化电路实现的高级综合过程。

○需要的信息

逻辑综合： 设计文件、（单元电路库文件）、约束文件

物理综合： 金属层的电容描述文件（cap table）

电路综合的过程

○读入单元电路库文件

目标单元门电路、门电路的时间 / 面积 / 功耗信息

○读入设计文件

○分析设计文件

确定top模块

建立模块调用关系，设计的层次结构

设计已经被转化为generic gate描述

○确定设计约束

时钟信息、设计目标（面积、功耗、速度）

○设计优化

唯一化、层次打散、路径分析和路径优化、单元库映射

○电路输出

网表文件、约束文件

FIFO的综合和后仿演示

○综合的步骤

```
yosys -s fifoRO.y
```

○后仿的步骤

```
iverilog -o fifo_post_syn_test -s test -g2012 \  
NangateOpenCellLibrary.v fifoRO_syn.v test/fifo_post_syn_test
```

```
vvp fifo_post_syn_test
```

○后仿的步骤

Reset被误认为inout端口

Specify的时间约束似乎不起作用

- 使用Verilog HDL设计时序逻辑电路
 - 会使用Verilog HDL设计基本的时序逻辑电路
- 时序逻辑电路的测试设计
 - 能够设计测试来验证时序逻辑电路
- 复杂时序逻辑电路设计
 - 循环仲裁器 (round-robin arbiter)
 - 指针型的FIFO缓冲
 - 只是介绍, 了解基本概念
- 时序逻辑电路的时序分析
 - 路径时间检查
 - 时钟树
 - 只是介绍, 了解基本概念

任何问题?

编程题 1:

用Verilog HDL编写一个4路随机仲裁器。
只提交设计电路，测试模块不提交。

编程题 2:

用Verilog HDL编写一个输出为寄存器直接输出的FIFO缓冲，缓冲深度为8，数据宽度为4位。只提交设计电路，测试模块不提交。

编程题 3:

用Verilog HDL编写一个交通灯，按照红灯10个时钟周期，黄灯1个时钟周期，绿灯9个时钟周期，黄灯2个时钟周期的顺序循环。只提交设计电路，测试模块不提交。

课堂习题 (课堂提问方式)

对于下面的时序逻辑电路：

- (1) 寻找该电路的关键路径
- (2) 分析可能优化该关键路径时序的方法，以及其优缺点。

