2021-2022学年秋季学期

第五部分-B
高级时序逻辑设计

授课团队：宋威
助　　教：薛子涵

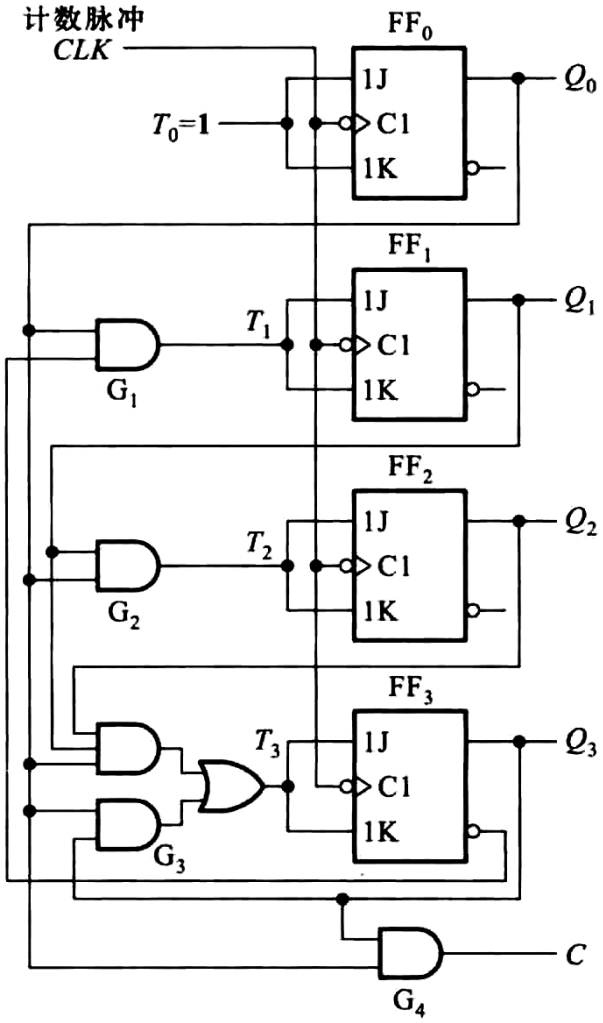
高级时序逻辑设计

- 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
  - 排序器
- 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

常用时序逻辑—计数器（十进制）

```
module counter10(
  input clk,
  output [3:0] q,
  output c);

  reg [3:0] q;
  always @(negedge clk)
    if(q < 4'd9) q <= q + 1;
    else         q <= 0;

  assign c = q == 4'd9;
endmodule
```

如果我们在设计时还不确定计数器的计数数值怎么办？



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

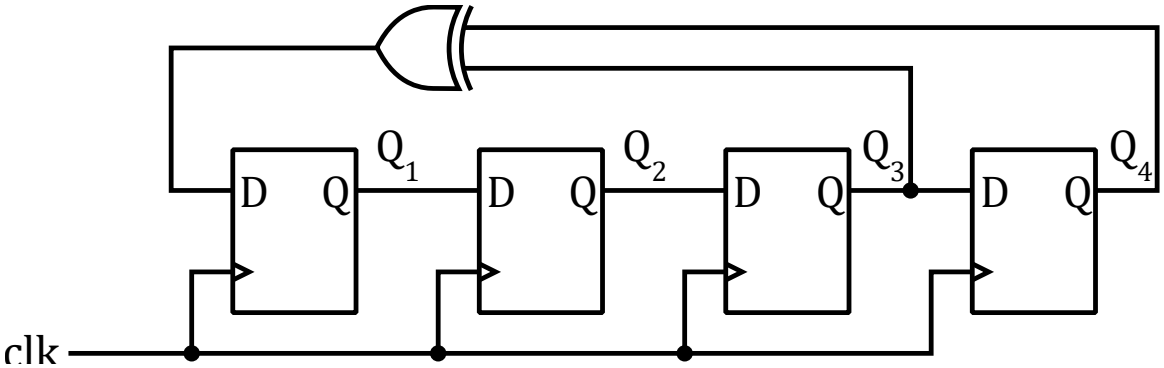
伪随机数发生器

- 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）



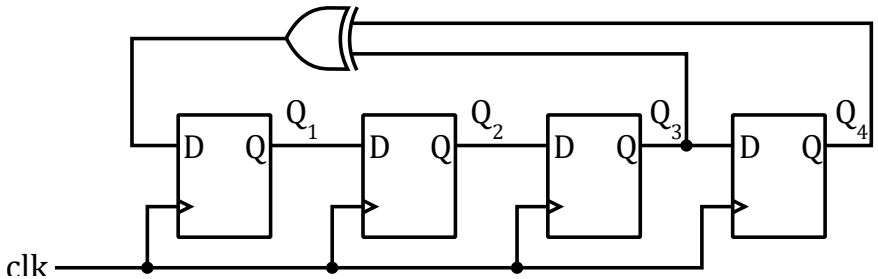
$$D_1 = Q_3 \oplus Q_4$$

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？

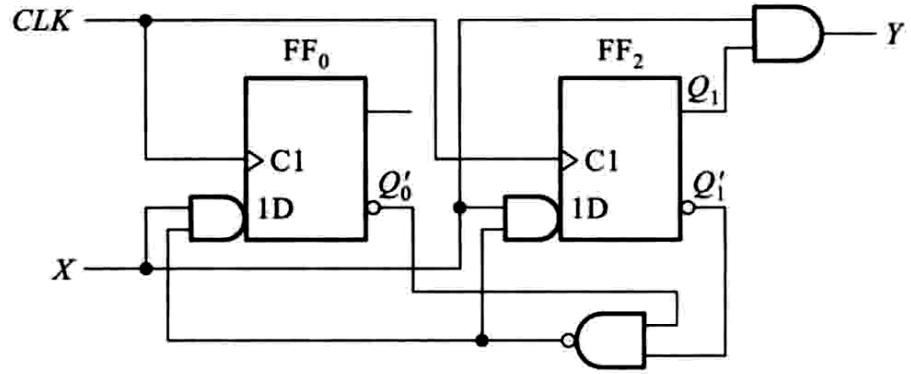
n位伪随机数发生器

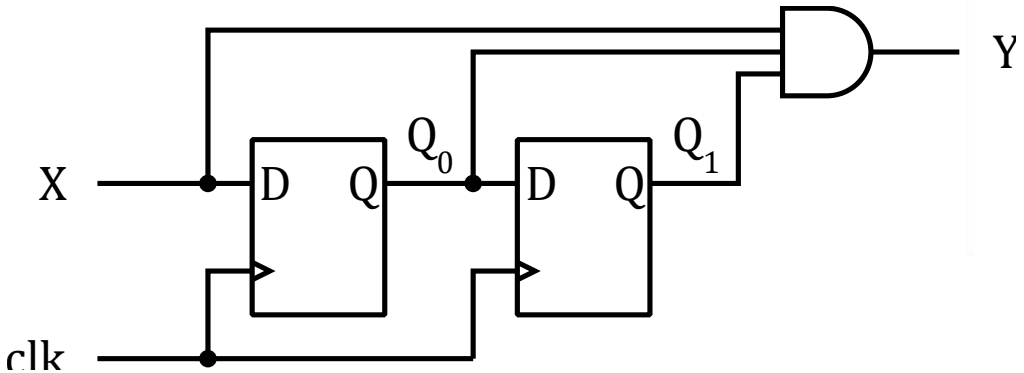
```
module lfsrN #(parameter N=4) (
  input clk,
  output [N-1:0] q);

  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
```

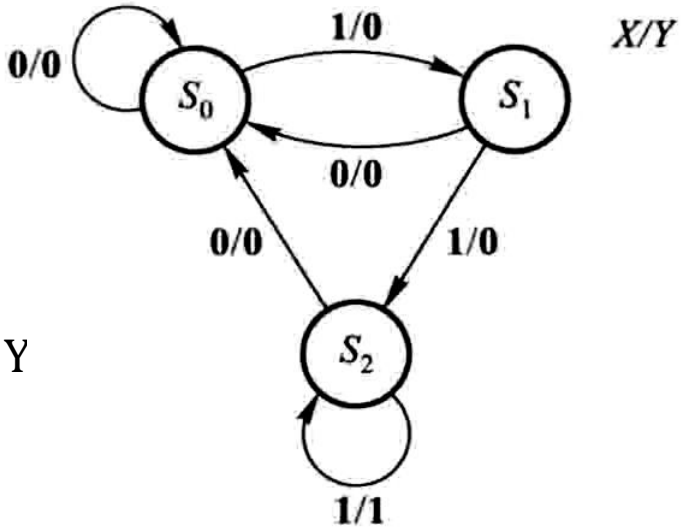
N	F()	N	F()
3	$Q_3 \oplus Q_2$	4	$Q_4 \oplus Q_3$
5	$Q_5 \oplus Q_3$	6	$Q_6 \oplus Q_5$
7	$Q_7 \oplus Q_6$	8	$Q_8 \oplus Q_6 \oplus Q_5 \oplus Q_4$

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



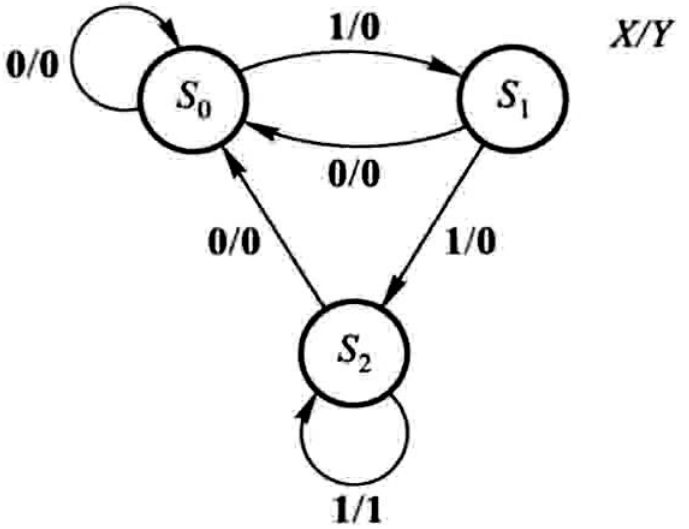




时序逻辑电路设计：簇发检测器

- 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
```

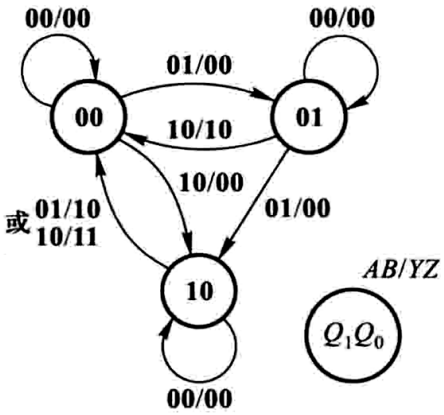
时序逻辑电路设计：贩卖机

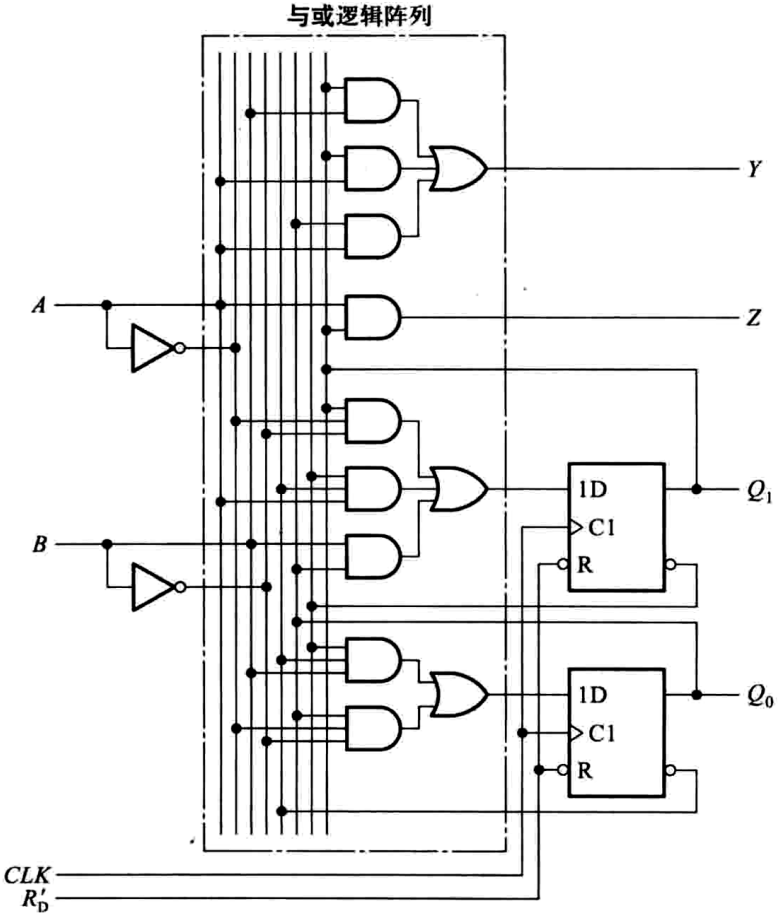
- 设计一个饮料贩卖机，售价1.5元，每次接受一个硬币（1元/05毛）。如果投入1.5元，给一杯饮料。如果投入2元，给一杯饮料，吐出5毛。
  - A：投入1元硬币
  - B：投入5毛硬币
  - Y：给饮料
  - Z：退还5毛硬币

$$\begin{cases} Q_1^* = Q_1A'B' + Q_1'Q_0'A + Q_0B \\ Q_0^* = Q_1'Q_0'B + Q_0A'B' \end{cases}$$

$$\begin{cases} D_1 = Q_1A'B' + Q_1'Q_0'A + Q_0B \\ D_0 = Q_1'Q_0'B + Q_0A'B' \end{cases}$$

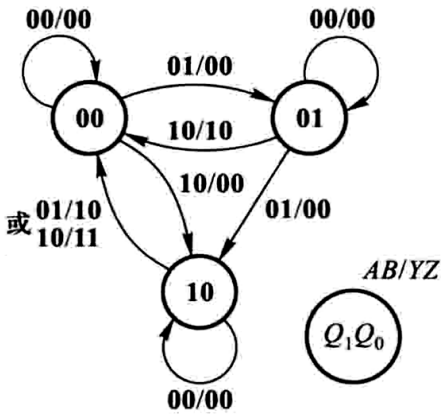
$$\begin{cases} Y = Q_1B + Q_1A + Q_0A \\ Z = Q_1A \end{cases}$$





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

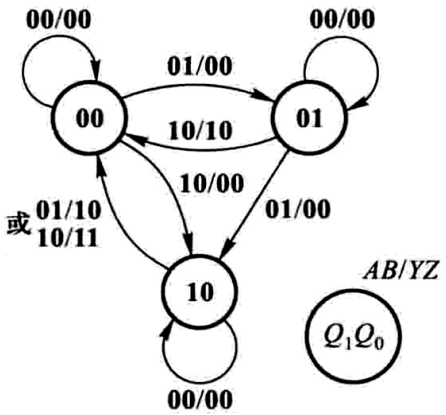


状态机描述形式

```
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) 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
```

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



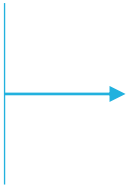

```
module vending_machine (
  input clk, rstn,
  input a, b,
  output y, z);

  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) 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

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

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



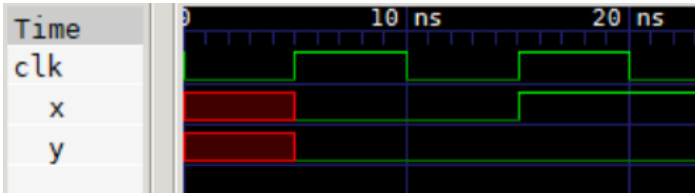
时钟驱动

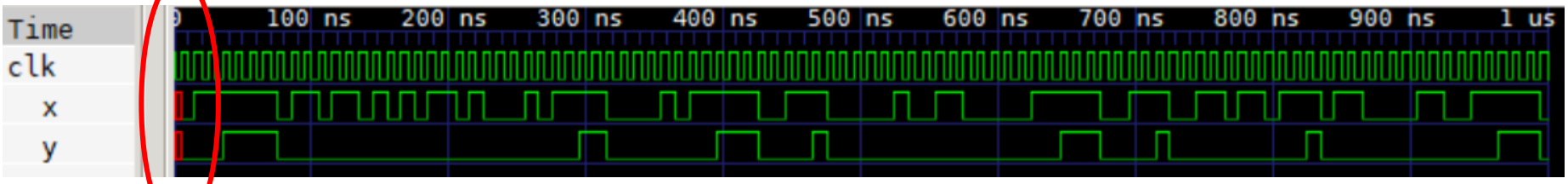
设置结束时间

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

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

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

时钟驱动



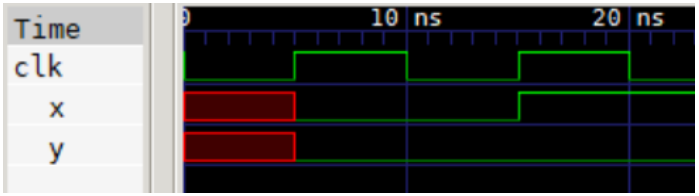


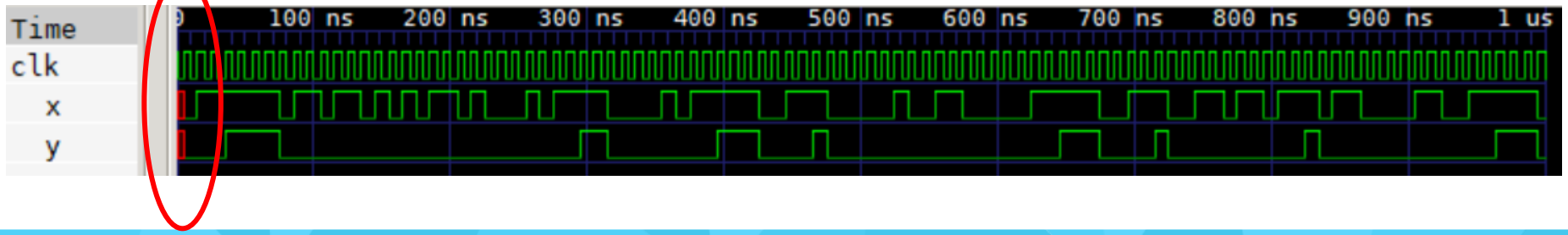
测试N位簇发检测器：无需复位信号（仿真自启）

```
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

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

endmodule
```

cnt也无需复位

当x=0时，y被强制为0



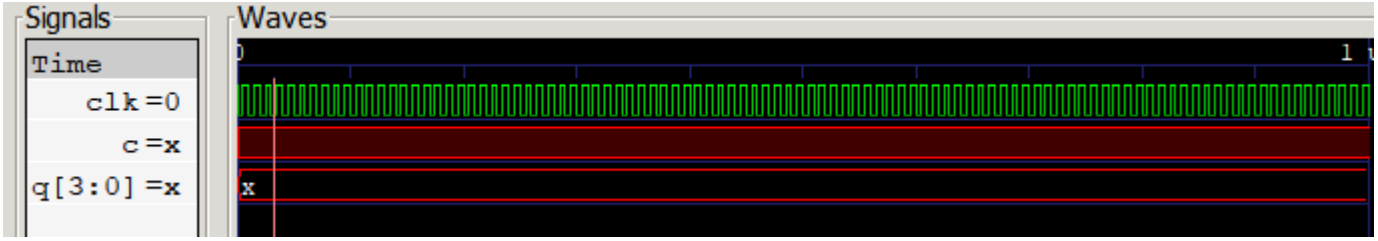


测试任意计数器：需要复位信号

```
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
```

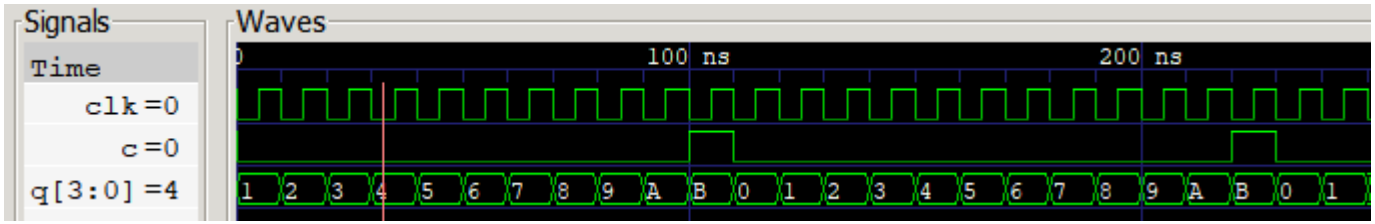
c的初始值是x



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
```

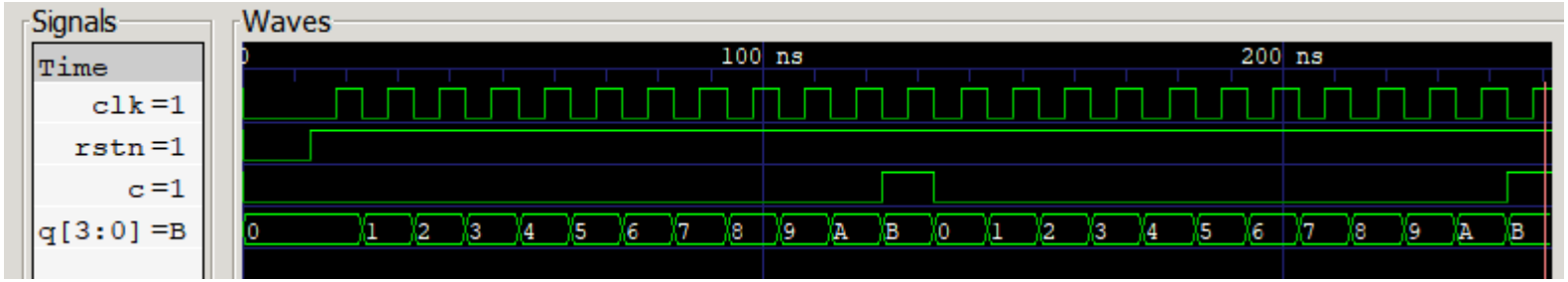
```
module counterN #(parameter num=10) (
  input clk,
  output [3:0] q,
  output c);

  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
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
```



电路复位比较

- $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。

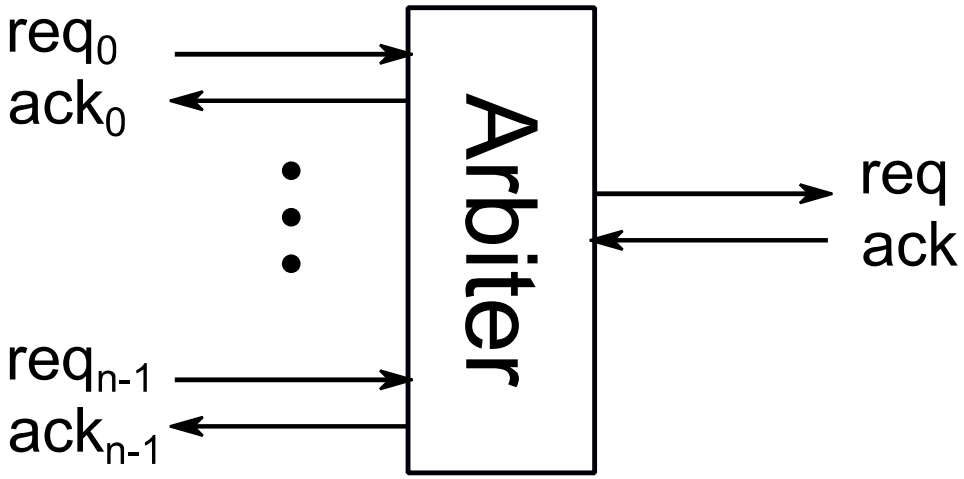
时序逻辑电路的测试设计总结

- 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

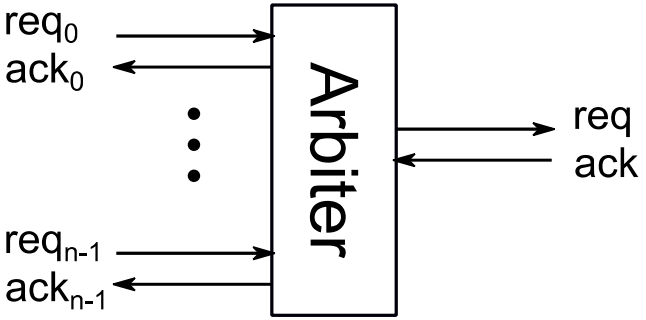
- 仲裁器（Arbiter）
  - 当一个共享的资源被多个电路争用时，我们需要仲裁器在可能同时出现的多个请求中选出一路请求，让其使用资源。

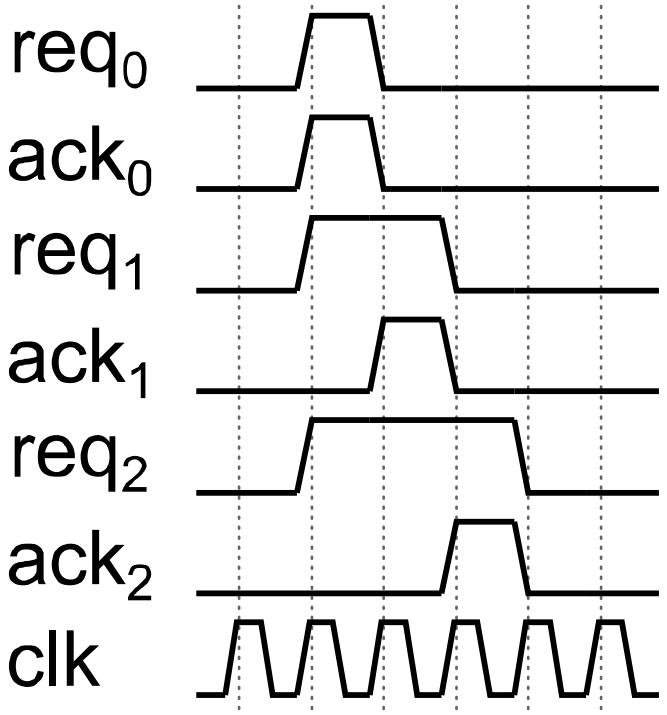


  - 如何知道哪一路信号正在请求资源?
  - 如何通知被选中的那一路信号?
  - （请求，响应）： (request, acknowledge) 或者 (req, ack)

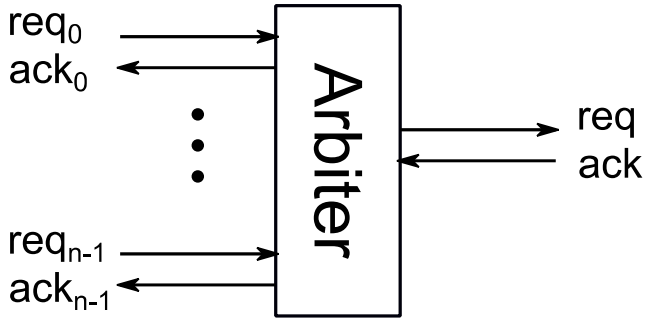
仲裁器

- 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

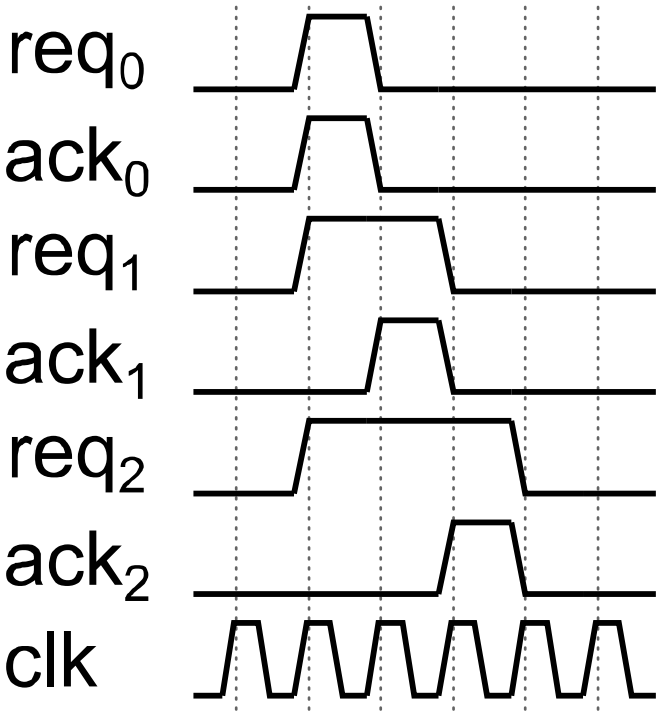


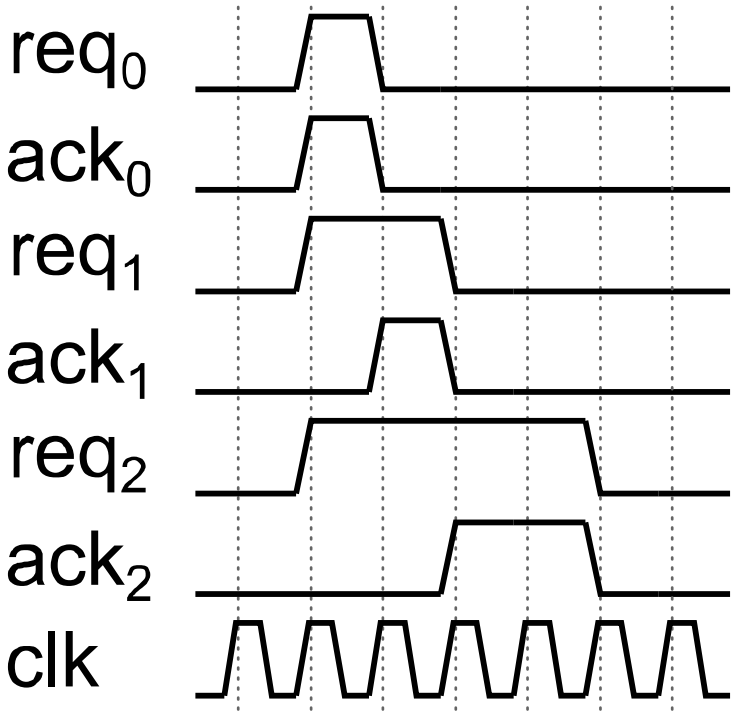


仲裁器输入

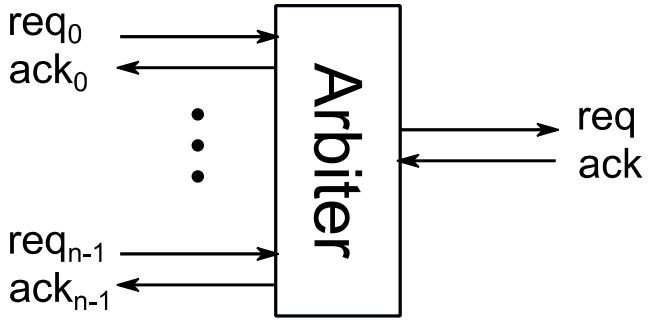

req0
ack0
reqn-1
ackn-1
Arbiter
req
ack

对吗？

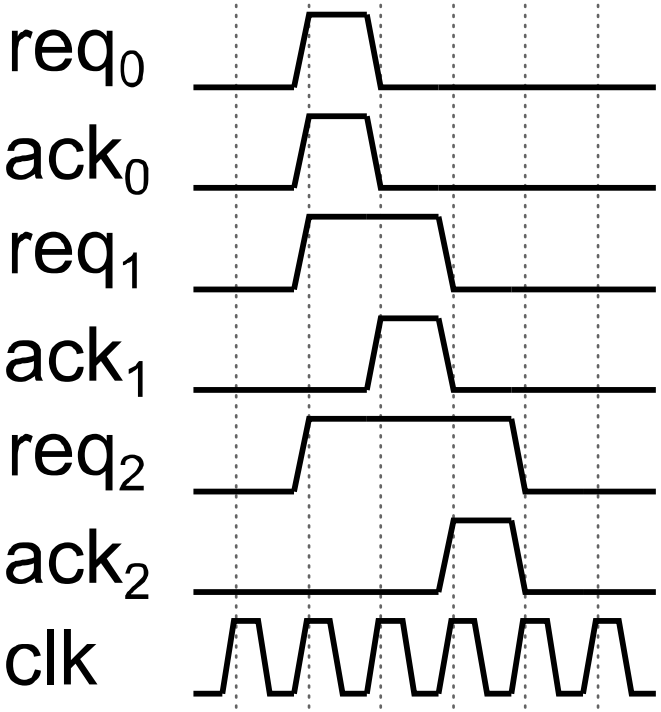

req0
ack0
req1
ack1
req2
ack2
clk

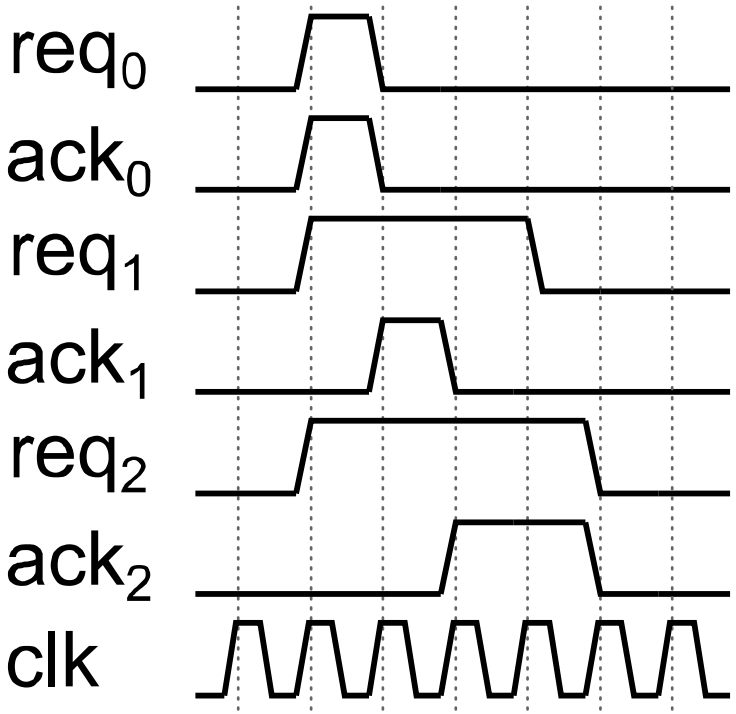

req0
ack0
req1
ack1
req2
ack2
clk

仲裁器输入

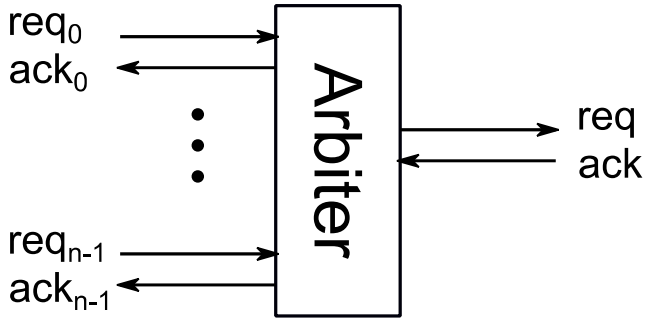

req0
ack0
reqn-1
ackn-1
Arbiter
req
ack

对吗？

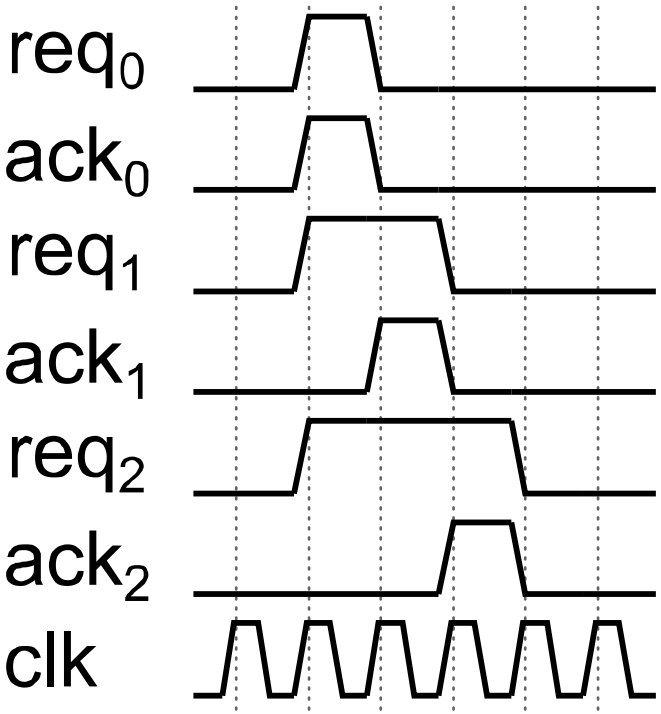

req0
ack0
req1
ack1
req2
ack2
clk

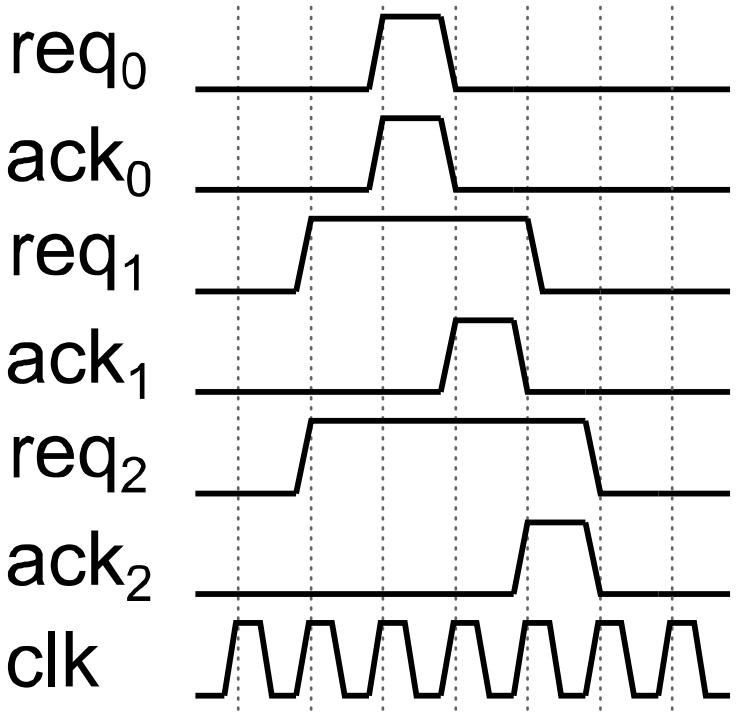

req0
ack0
req1
ack1
req2
ack2
clk

仲裁器输入

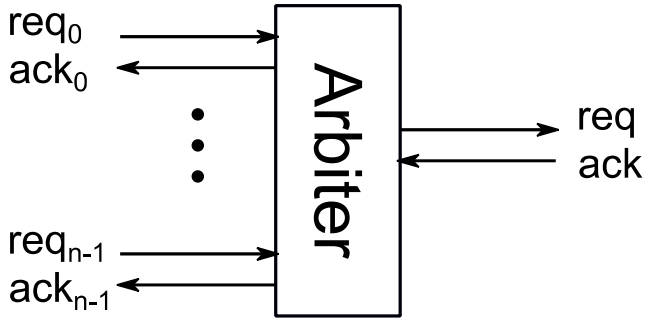

req0
ack0
reqn-1
ackn-1
Arbiter
req
ack

对吗？

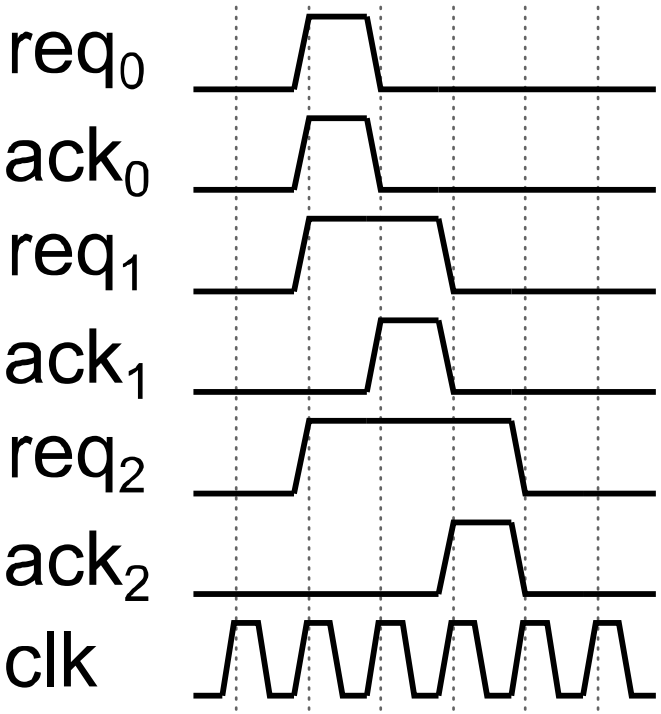

req0
ack0
req1
ack1
req2
ack2
clk

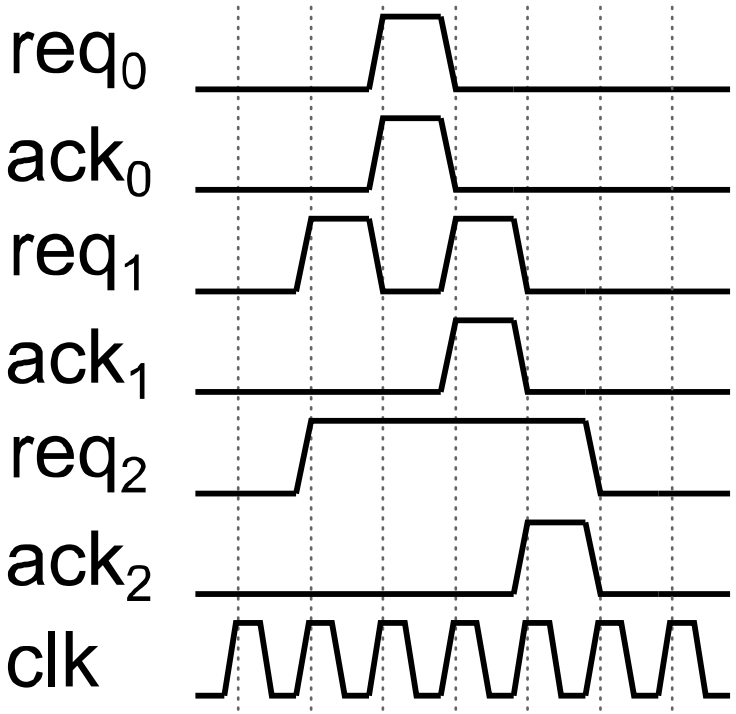

req0
ack0
req1
ack1
req2
ack2
clk

仲裁器输入


req0
ack0
reqn-1
ackn-1
Arbiter
req
ack

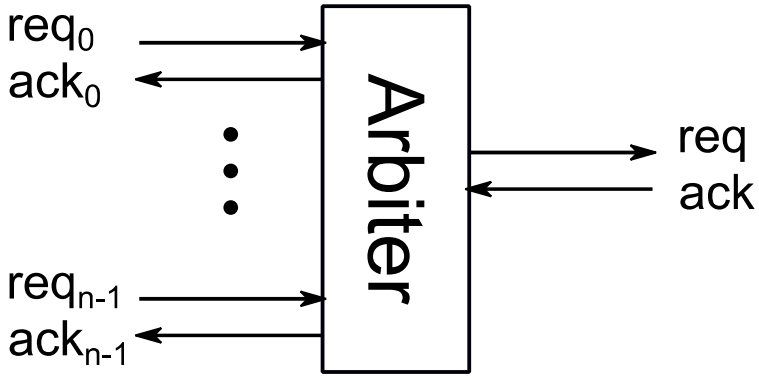
对吗？


req0
ack0
req1
ack1
req2
ack2
clk


req0
ack0
req1
ack1
req2
ack2
clk

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

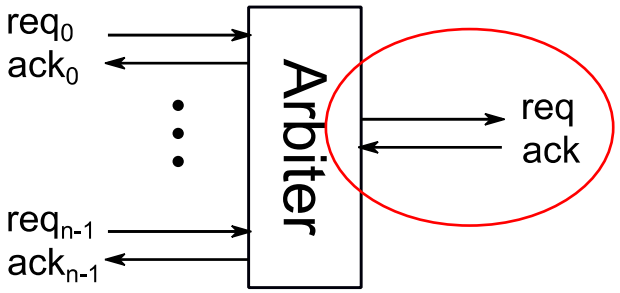
```
module ArbStatic3 (input clk, rstn,
                   input [2:0] req,
                   output [2:0] ack);
  reg [2:0] ack;

  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)

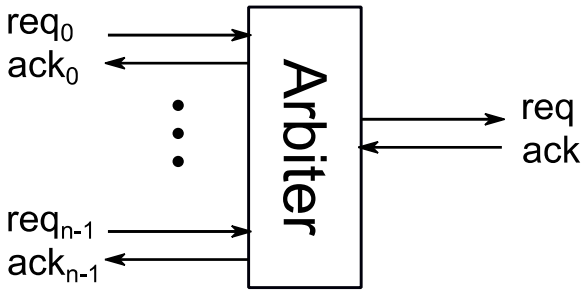
- 如果考虑输出端可能暂时不能响应呢?

```
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

endmodule
```



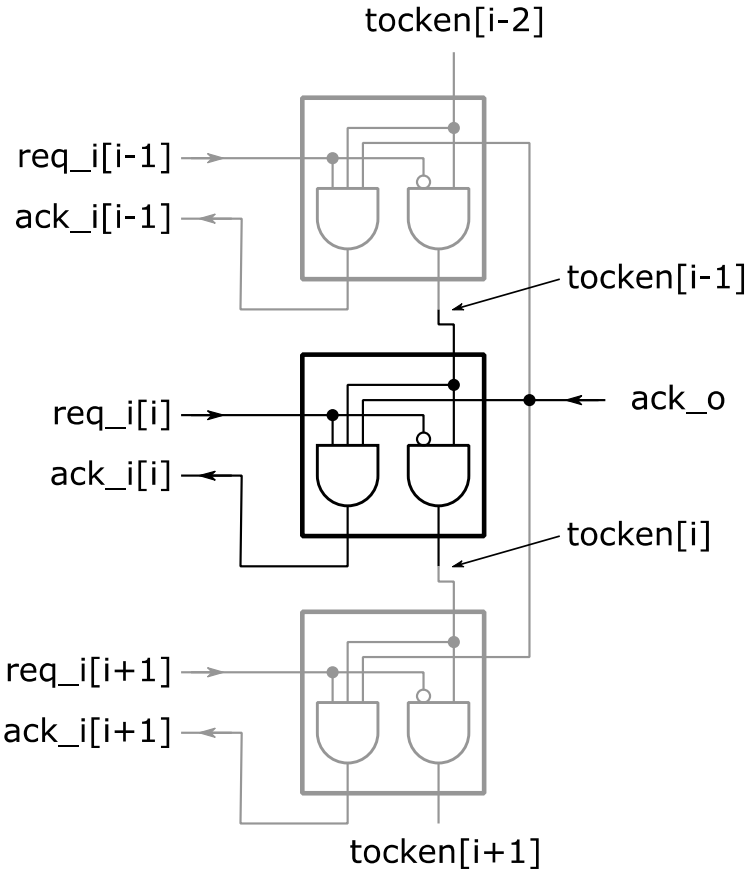
N路静态优先级仲裁器

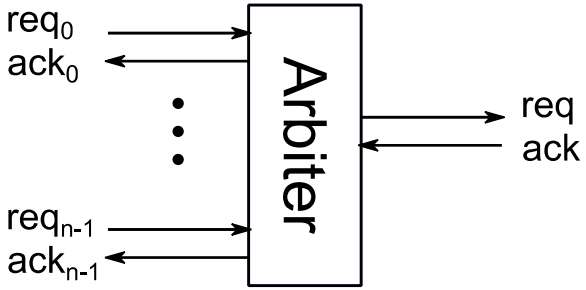
- 我们如何从3路扩展到N路?
  - 是否可以将各路拆开，推导每一路的响应信号的表达式?



N路静态优先级仲裁器

- 我们如何从3路扩展到N路?

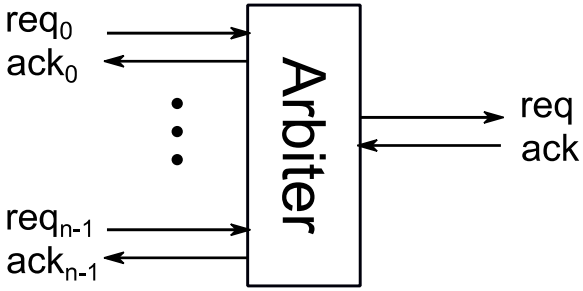




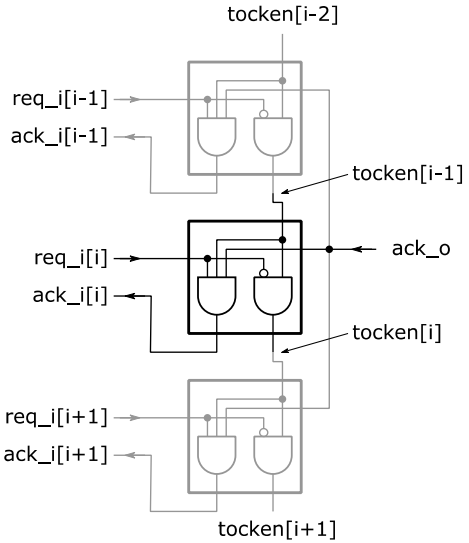

```
ack_i[i] = token[i-1] & req_i[i] & ack_o;
token[i] = token[i-1] & ~req_i[i];
```

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
```

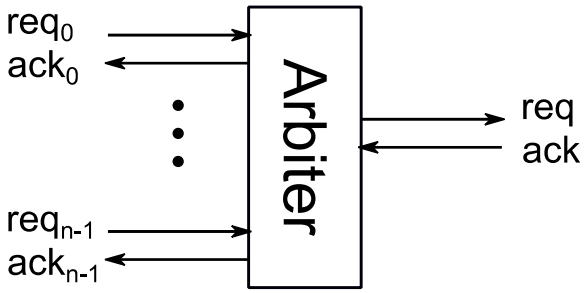


N路动态优先级仲裁器

- 我们如何动态设定优先级?

原有优先级: $7 \leftarrow 6 \leftarrow 5 \leftarrow 4 \leftarrow 3 \leftarrow 2 \leftarrow 1 \leftarrow 0$

$7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1 \quad 0$



- 添加一个优先级的输入端

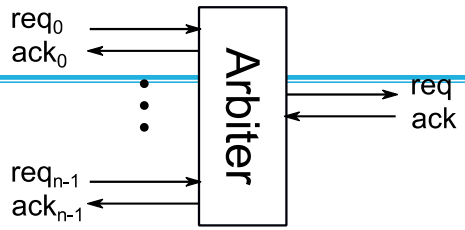
```
prio[7:0] = 8'b0001_0000;
```

req[4] 有最高优先级

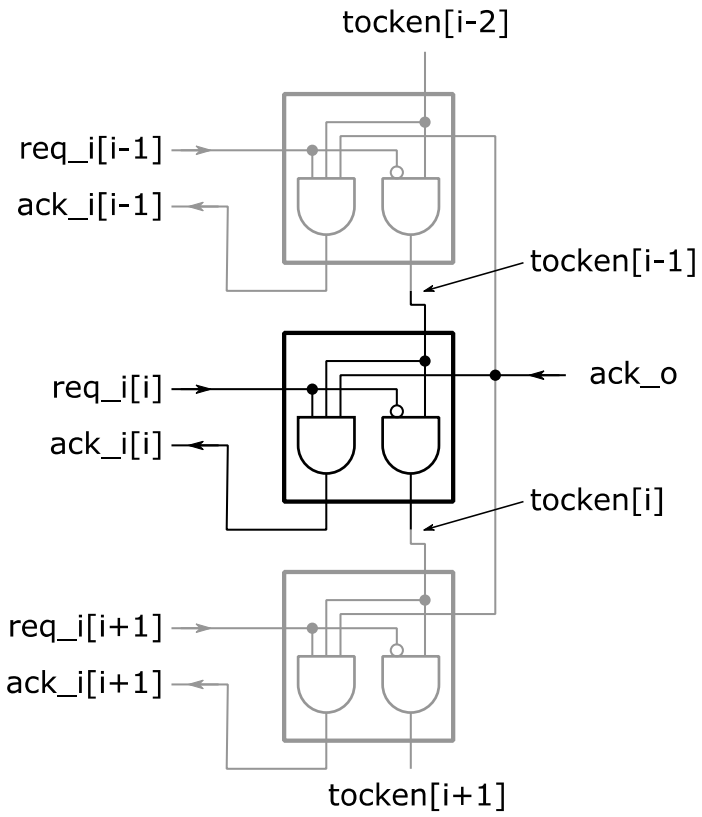
优先级排序: $3 \leftarrow 2 \leftarrow 1 \leftarrow 0 \leftarrow 7 \leftarrow 6 \leftarrow 5 \leftarrow 4$

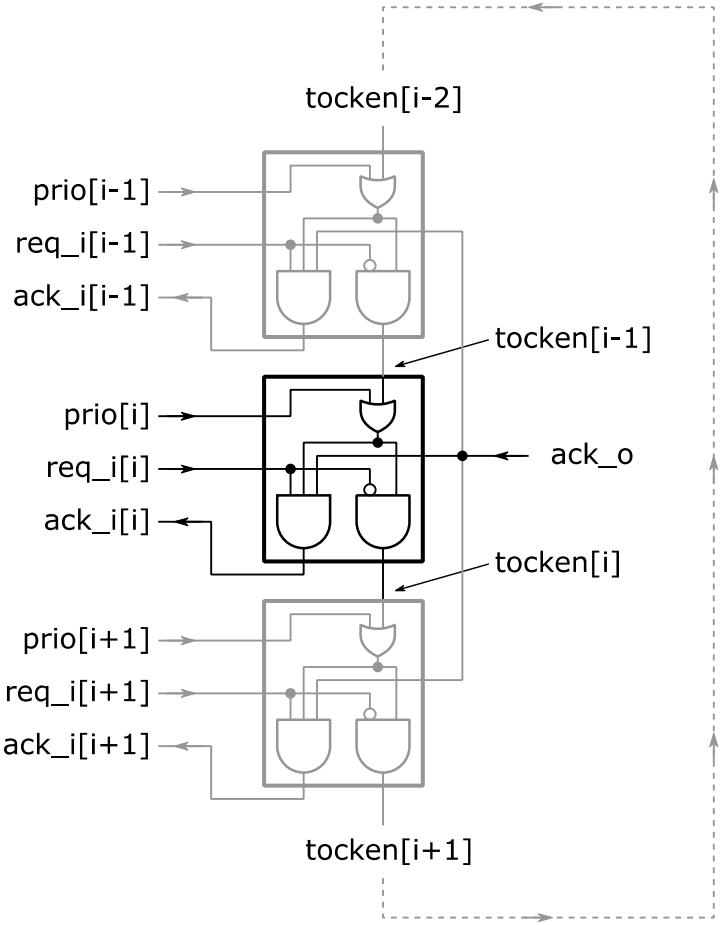
$7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1 \quad 0$

N路动态优先级仲裁器



- 我们如何动态设定优先级?

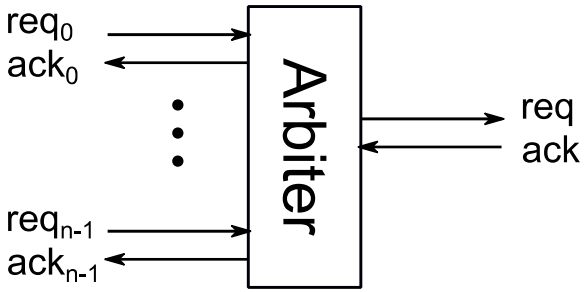


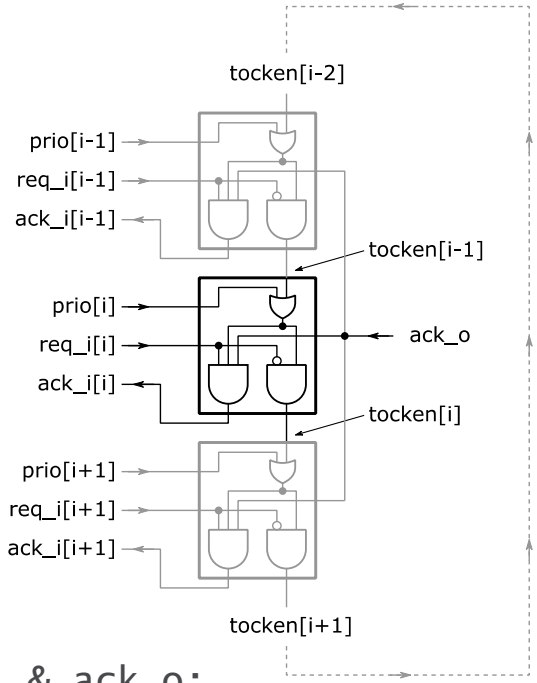



```
ack_i[i] = (token[i-1] | prio[i]) & req_i[i] & ack_o;
token[i] = (token[i-1] | prio[i]) & ~req_i[i]
```

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
```

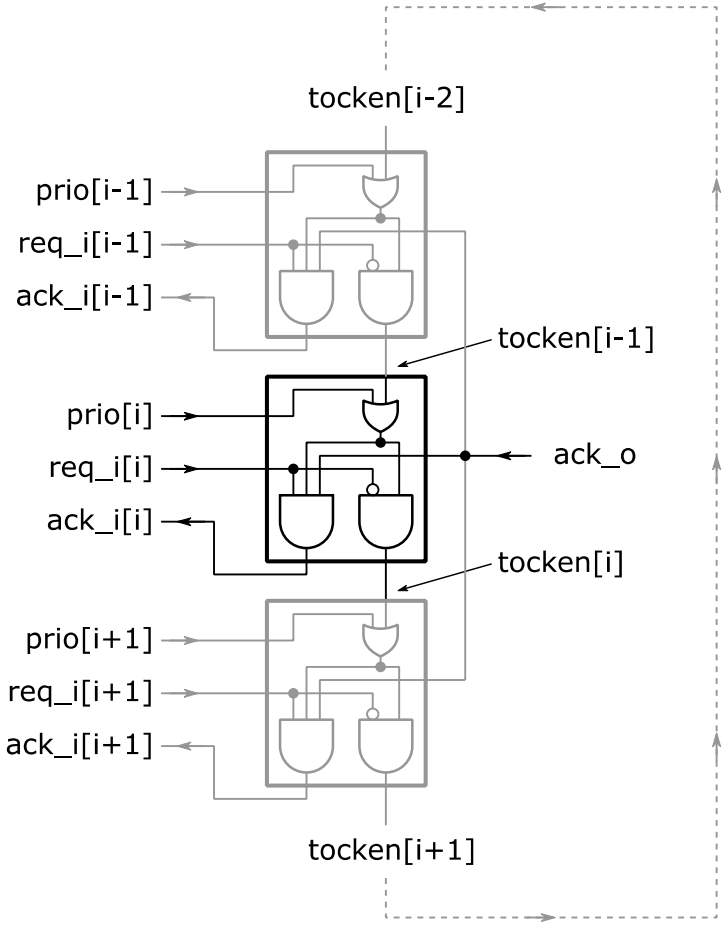


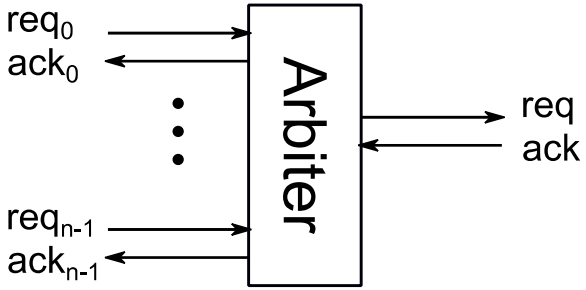


组合逻辑环

N路动态优先级仲裁器

- 我们如何动态设定优先级?



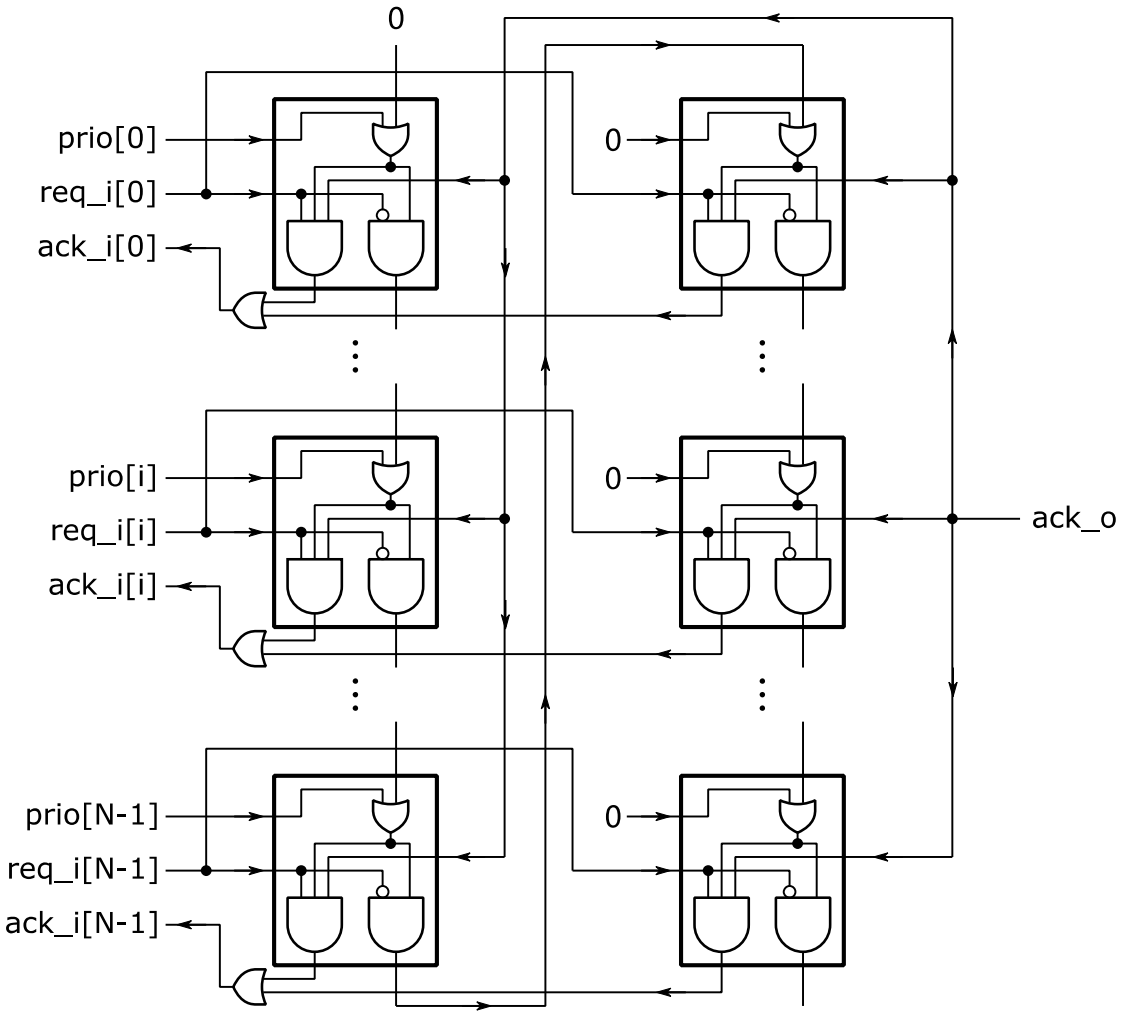


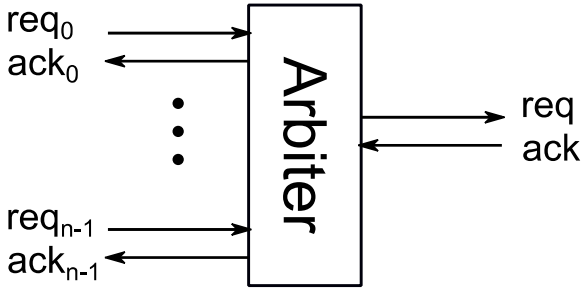
如何解决环的问题?

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

N路动态优先级仲裁器

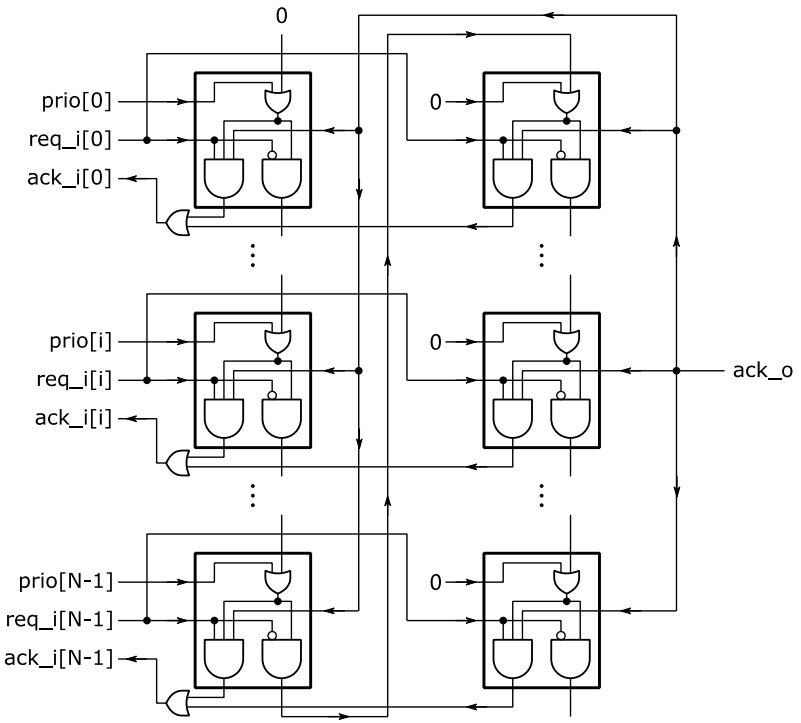
- 我们如何动态设定优先级?





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 endgenerate
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：$req = 8'b01001010$

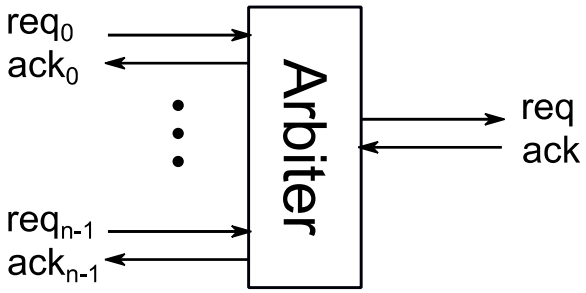
$ack = 8'b01000000$

第五周期：$req = 8'b01001010$

$ack = 8'b00000010$

第六周期：$req = 8'b01001001$

$ack = 8'b00001000$

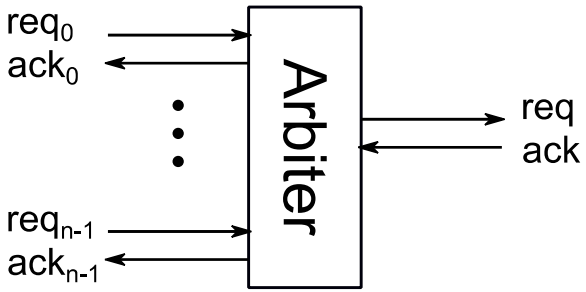


当多个请求同时有效时，循环响应。

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
```



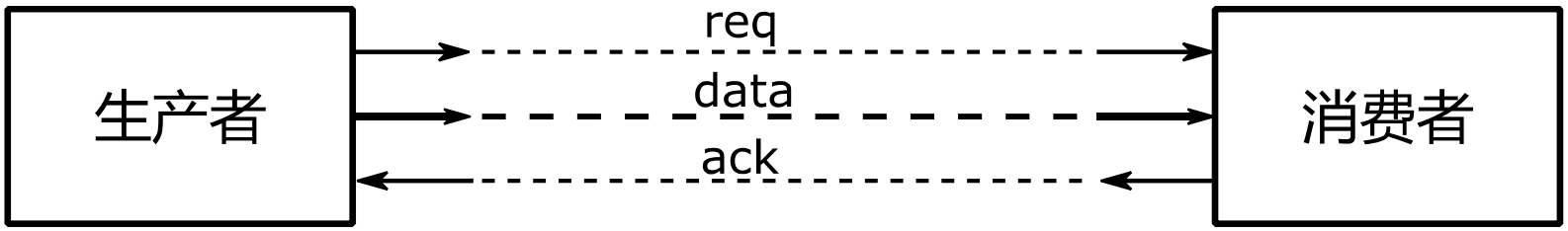
仲裁器总结

- 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

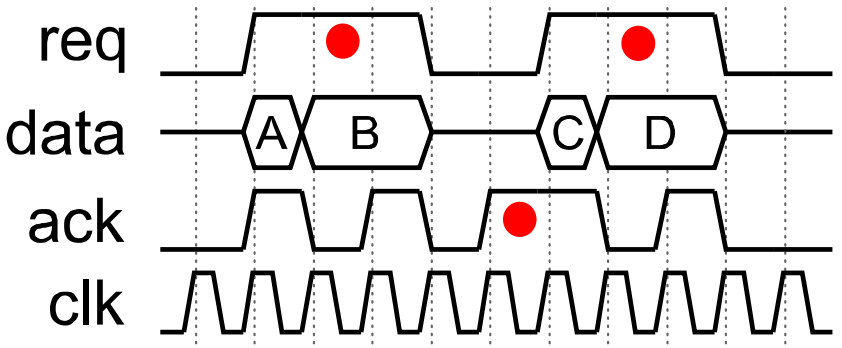
- FIFO缓冲（Buffer）
  - **当一个数据流的产生者和消费者的步调不统一，直接连接会导致数据吞吐率下降**
    - 生产者产生了数据消费者不能立即接收
    - 消费者可以消费数据却没有



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

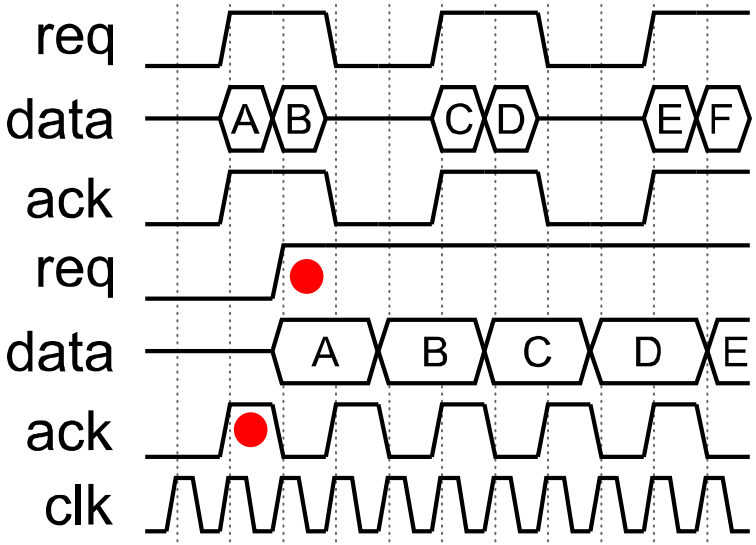
- FIFO缓冲（Buffer）
  - **在这种情况下，插入一个FIFO缓冲可以协调两边的步调**
    - 生产者产生数据消费者不能接受时，先保存在缓冲器内。
    - 当消费者可以消费数据而生产者没有数据时，可以从缓冲区拿数。

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满时随时接受数据

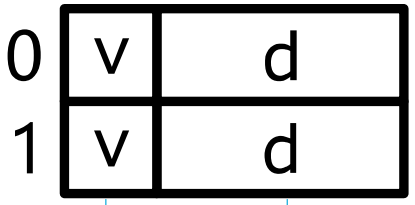
缓冲区可以在有数据时一直输出数据

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

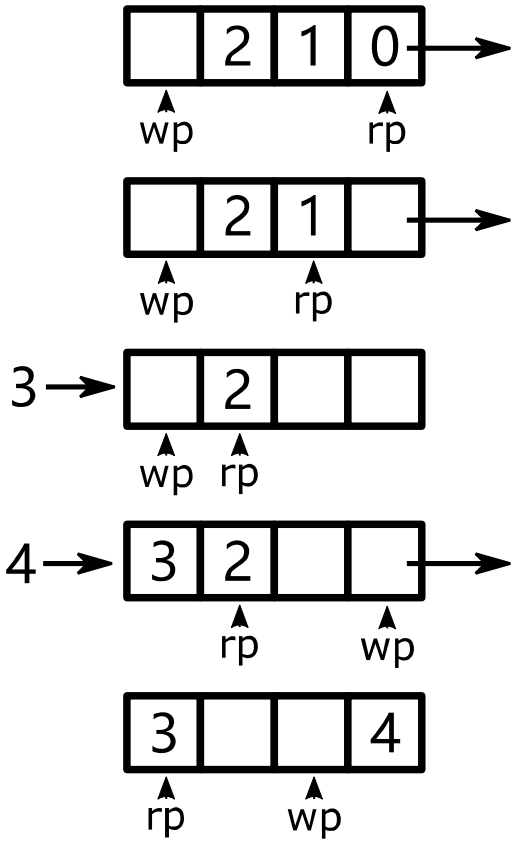


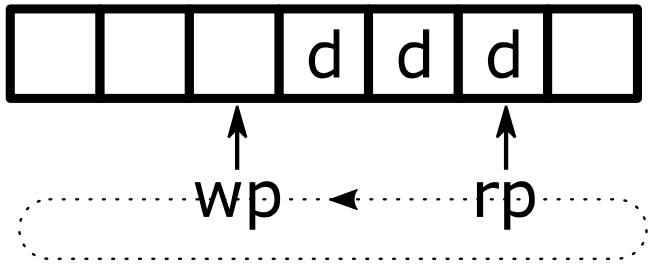

```
module fifo2 #(parameter dw=8) ( input clk, rstn,
  input [dw-1:0] d_in, input req_in, output ack_in,
  output [dw-1:0] d_out, output req_out, input ack_out);

  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
```

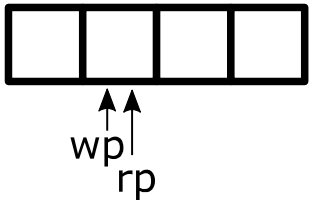
仅依赖于缓冲区数据的状态位还不足以完成控制。

深度为L的FIFO缓冲

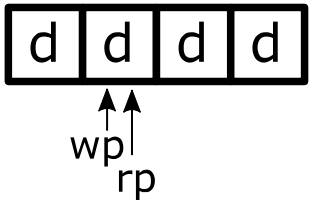
我们需要引入指针！







空



满

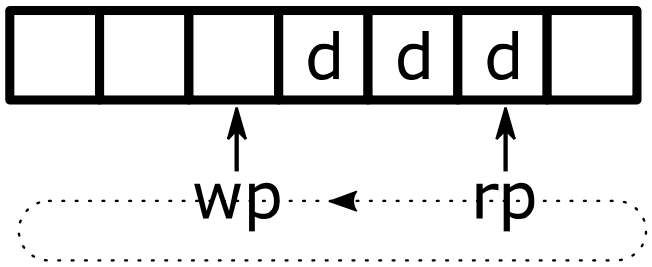
数据存入：wp自增，数据保存并使能。
数据读出：rp自增，数据读出并作废。
缓冲区满：wp等于rp并且valid[wp]有效。
缓冲区空：wp等于rp并且valid[rp]无效。

FIFO最小延迟1周期。

深度为L的FIFO缓冲

我们需要引入指针！

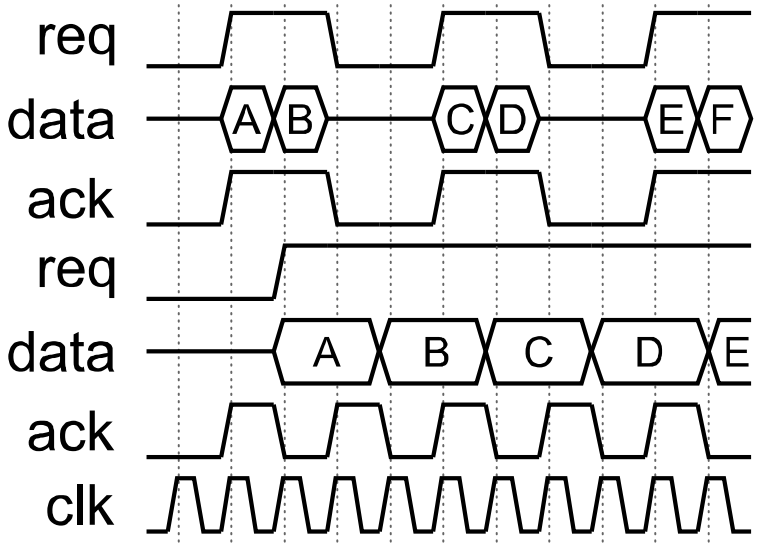
```
module fifo2 #(parameter dw=8, L=7)
( input clk, rstn,
  input [dw-1:0] d_in, input req_in, output ack_in,
  output [dw-1:0] d_out, output req_out, input ack_out);

  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
endmodule
```



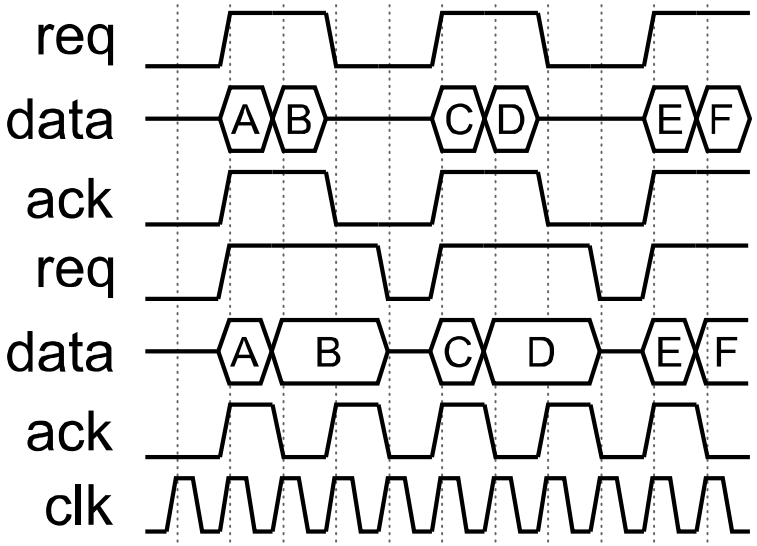
数据存入：wp自增，数据保存并使能。
数据读出：rp自增，数据读出并作废。
缓冲区满：wp等于rp并且valid[wp]有效。
缓冲区空：wp等于rp并且valid[rp]无效。

FIFO最小延迟1周期。

是否可以将最小延迟将为0周期?



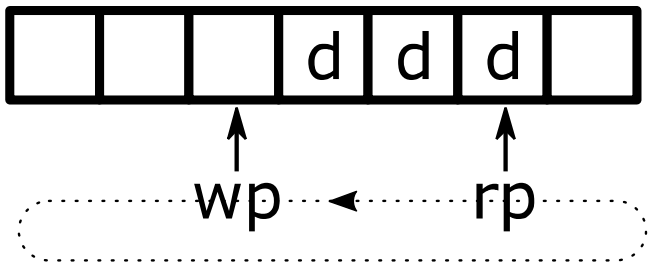
1周期传输延时
缓冲深度至少为2



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
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）

复杂时序逻辑电路设计：排序器

- 假设有N个整数，如何能在一个周期内给出关于这N个数的有序序列?

  输入序列：23, 16, 71, 52, 45, 38

  输出序列：16, 23, 38, 45, 52, 71

- 应用用途
  - 科学计算加速
  - 用于优先级比较
  - 表的合并操作
  - 网络报文归序

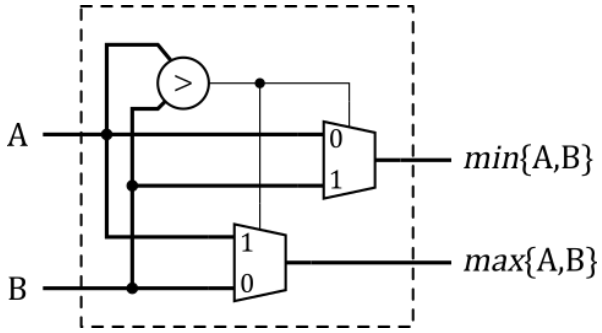
排序器的naïve实现

```
module sorter #(parameter N=8, dw=8)
( input [N-1:0][dw-1:0] d_in,
  output [N-1:0][dw-1:0] d_out); // SystemVerilog

  reg [N-1:0][dw-1:0] data;
  reg [dw-1:0] m;
  integer i, j;

  always @(d_in) begin
    data = d_in;
    for(i=N-1; i>0; i=i-1)
      for(j=0; j<i; j=j+1)
        if(data[j] > data[j+1]) begin
          m = data[j];
          data[j] = data[j+1];
          data[j+1] = m;
        end
  end

  assign d_out = data;
endmodule
```

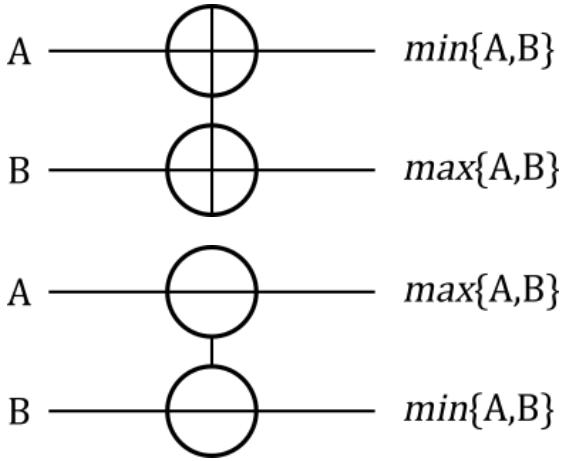
提问：这个电路长什么样子？
电路的复杂度如何？

CAE算子

- Compare-And-Exchange (CAE) 算子
  - 排序器的基本硬件比较单元
  - 硬件实现



  - 符号

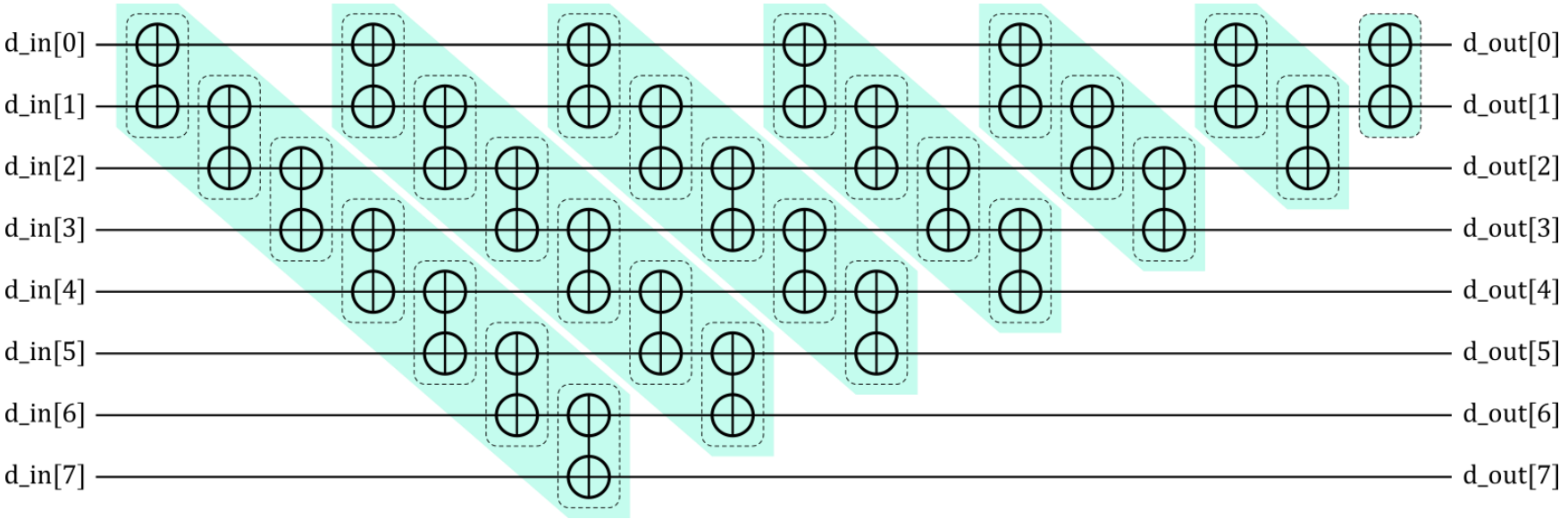


排序器的naïve实现

```
module sorter #(parameter N=8, dw=8)
( input [N-1:0][dw-1:0] d_in,
  output [N-1:0][dw-1:0] d_out); // SystemVerilog

  reg [N-1:0][dw-1:0] data;
  reg [dw-1:0] m;
  integer i, j;

  always @(d_in) begin
    data = d_in;
    for(i=N-1; i>0; i=i-1)
      for(j=0; j<i; j=j+1)
        if(data[j] > data[j+1]) begin
          m = data[j];
          data[j] = data[j+1];
          data[j+1] = m;
        end
  end

  assign d_out = data;
endmodule
```

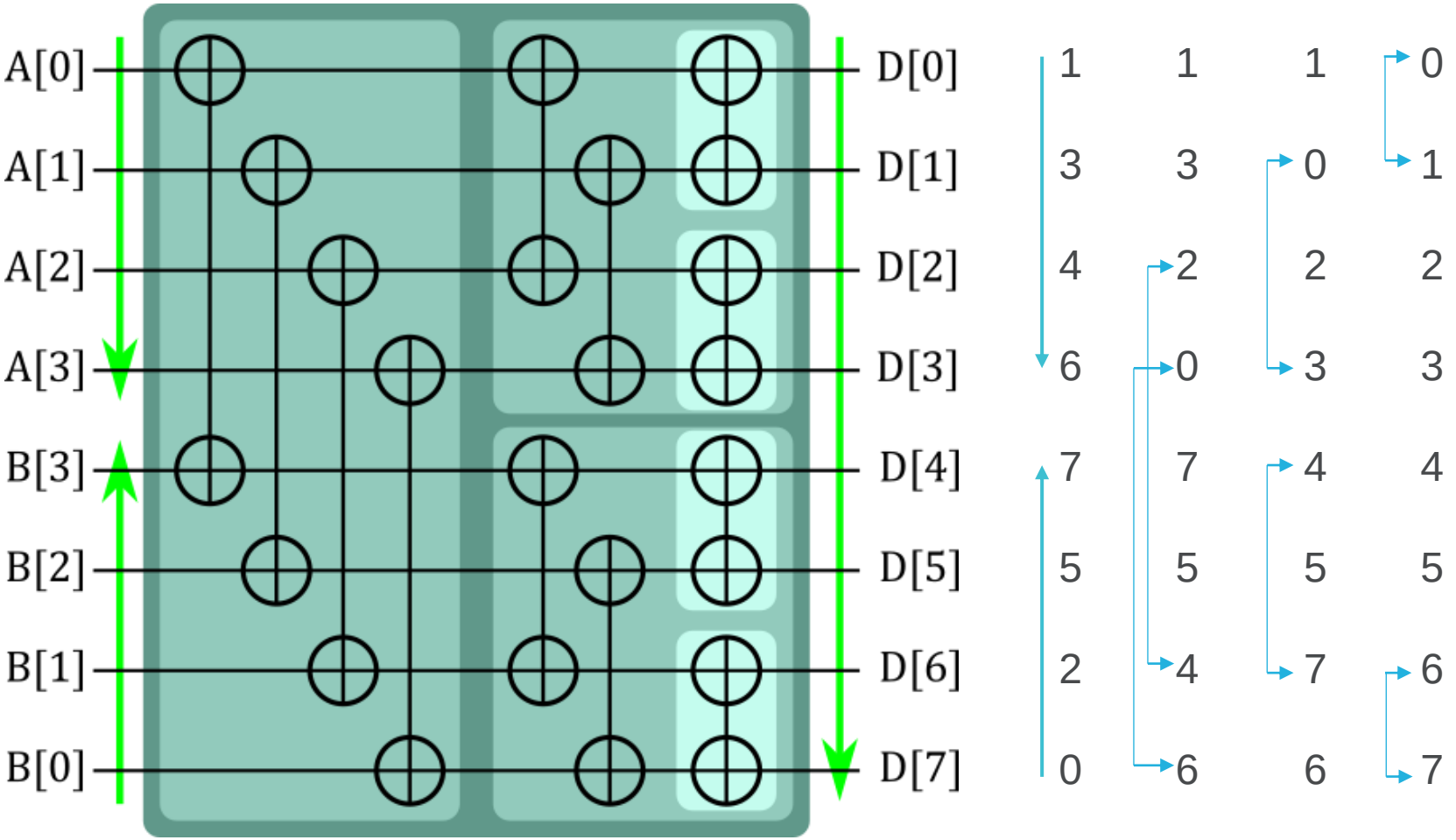
电路面积：$A \sim \frac{N(N-1)}{2} \sim O(N^2)$

电路速度：$T \sim 2N-3 \sim O(N)$



Bitonic 排序网络：8输入归并器

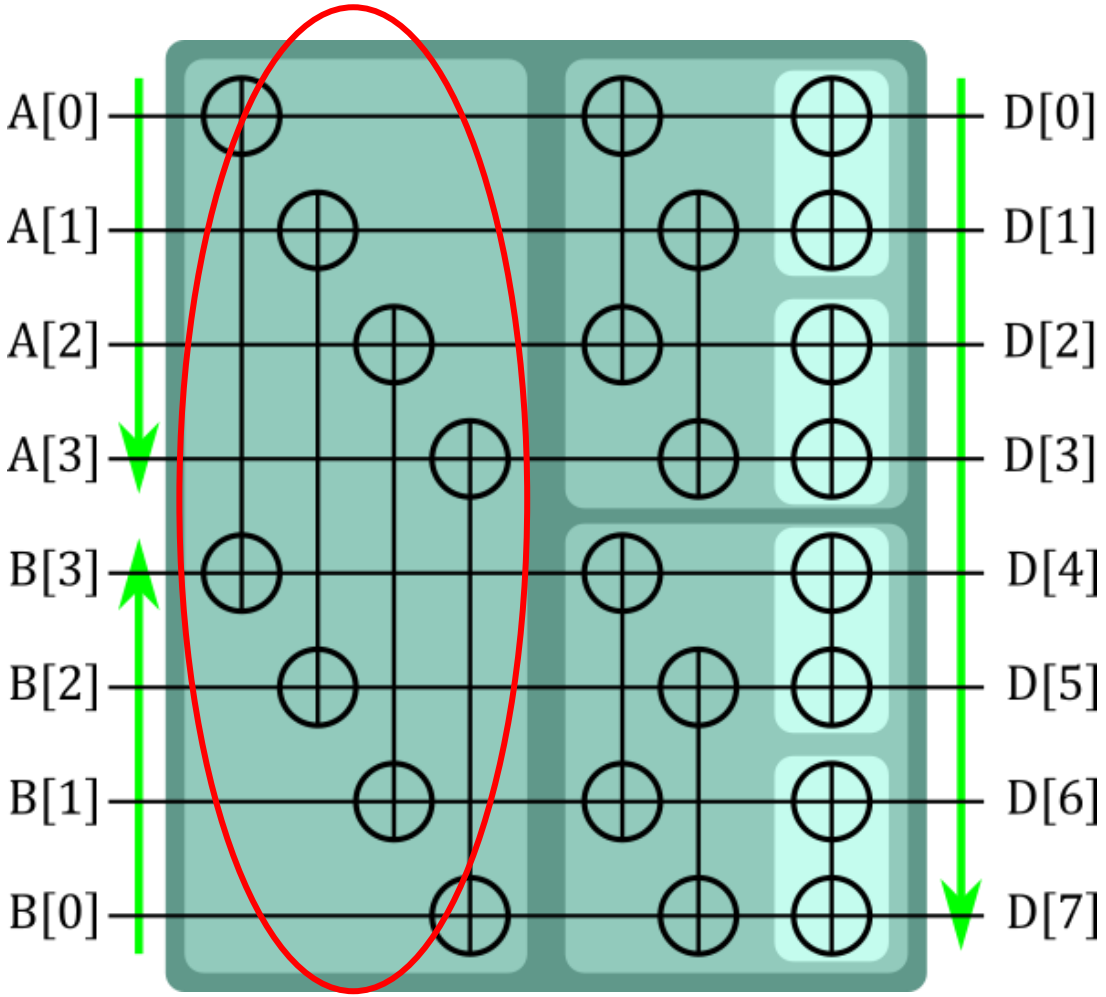
- 8-input Bitonic Merger



K.E. Batcher, Sorting Networks and Their Applications. AFIPS Proceedings of the Spring Joint Computer Conference, pp. 307-314, 1968.

Bitonic 排序网络：8输入归并器

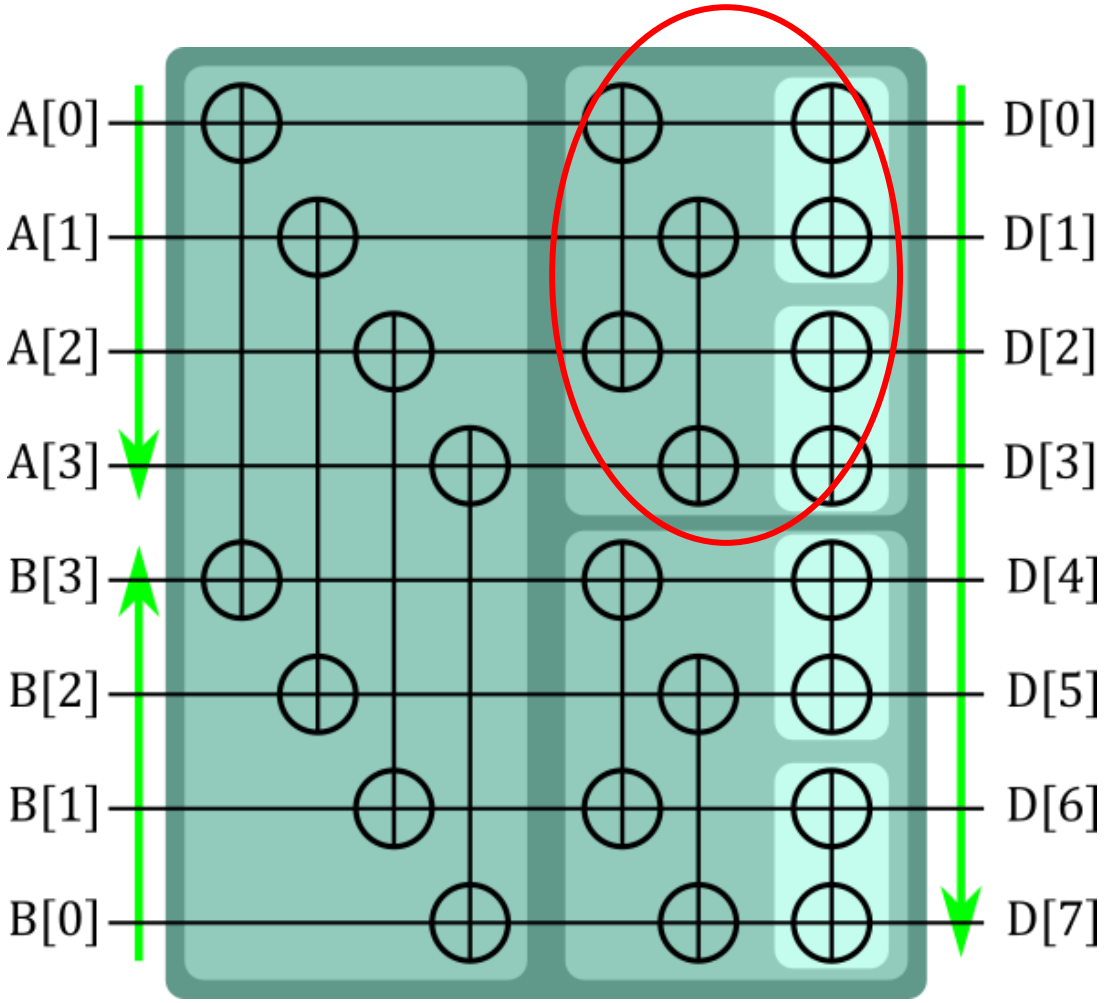
- 8-input Bitonic Merger



第一级，将4个小数放到上半边，4个大数放到下半边。

Bitonic 排序网络：8输入归并器

- 8-input Bitonic Merger

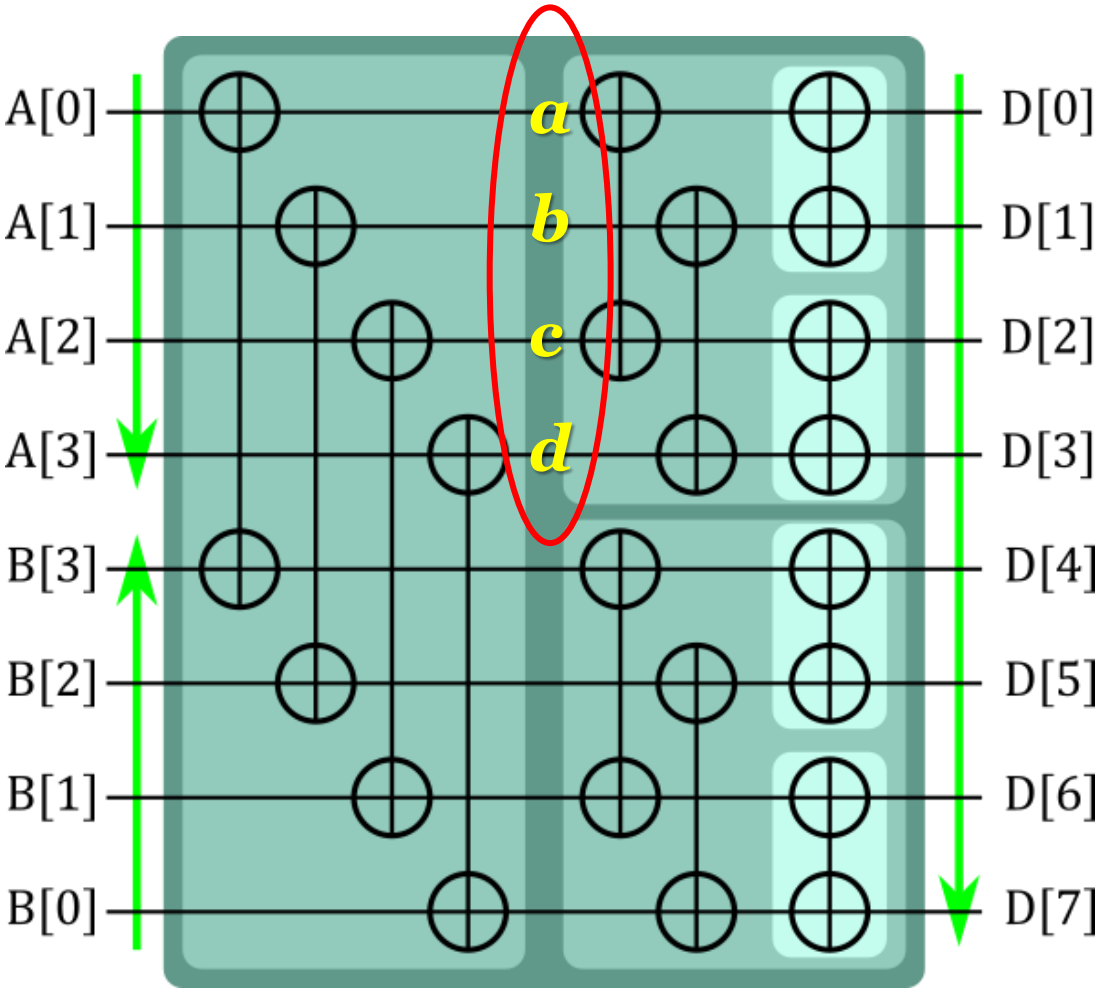


D[0]总是最小的，D[3]总是D[3]到D[0]中最大的。

为什么D[1]一定小于D[2]?

Bitonic 排序网络：8输入归并器

- 8-input Bitonic Merger



那我们假设D[1] > D[2]

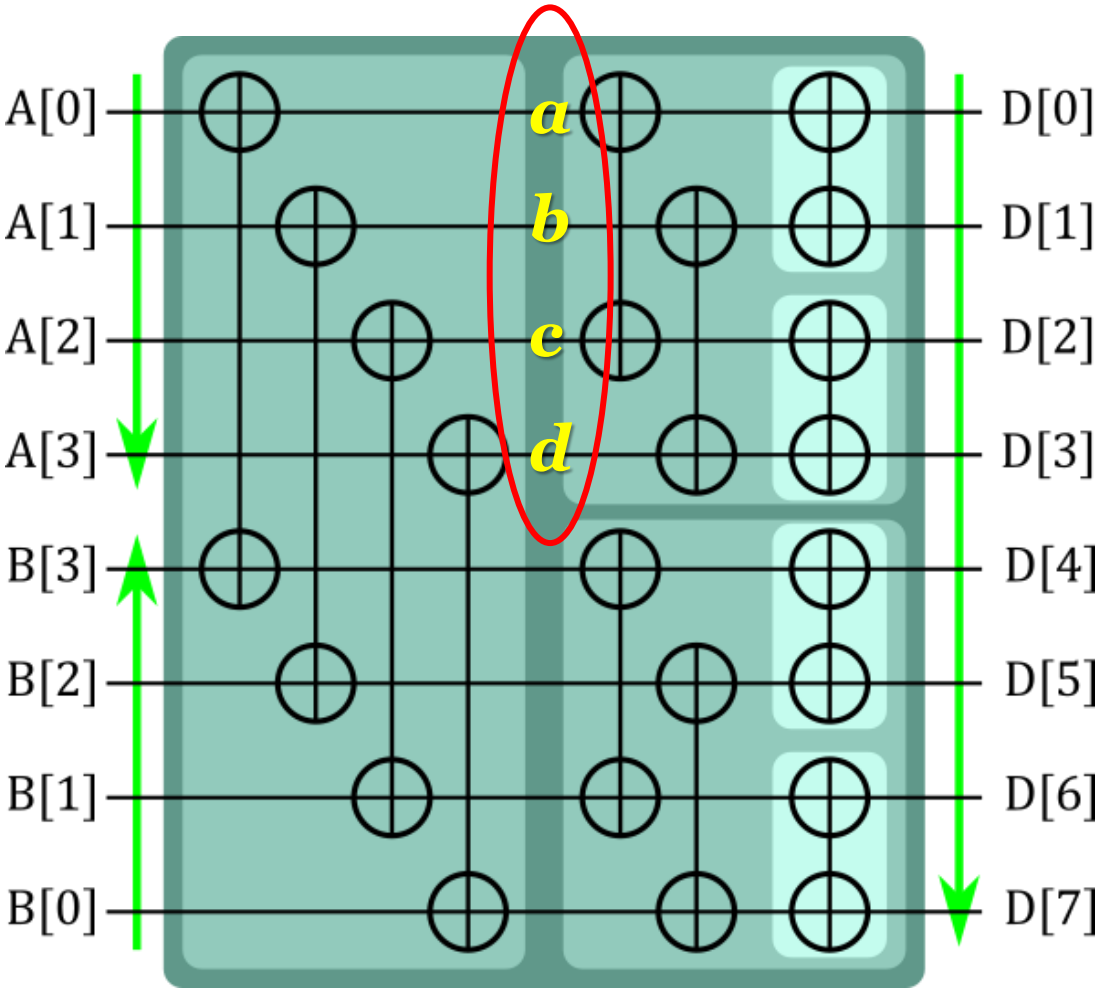
则推论：

$$\max\{\min\{a,c\},\min\{b,d\}\} > \min\{\max\{a,c\},\max\{b,d\}\}$$

进一步我们假设
D[1]=min{*a*,*c*}
我们则有
min{*a*,*c*} > max{*b*,*d*}

Bitonic 排序网络：8输入归并器

- 8-input Bitonic Merger



min{*a*,*c*} > max{*b*,*d*}

我们知道所有数中最小的是A[0]或B[0]，

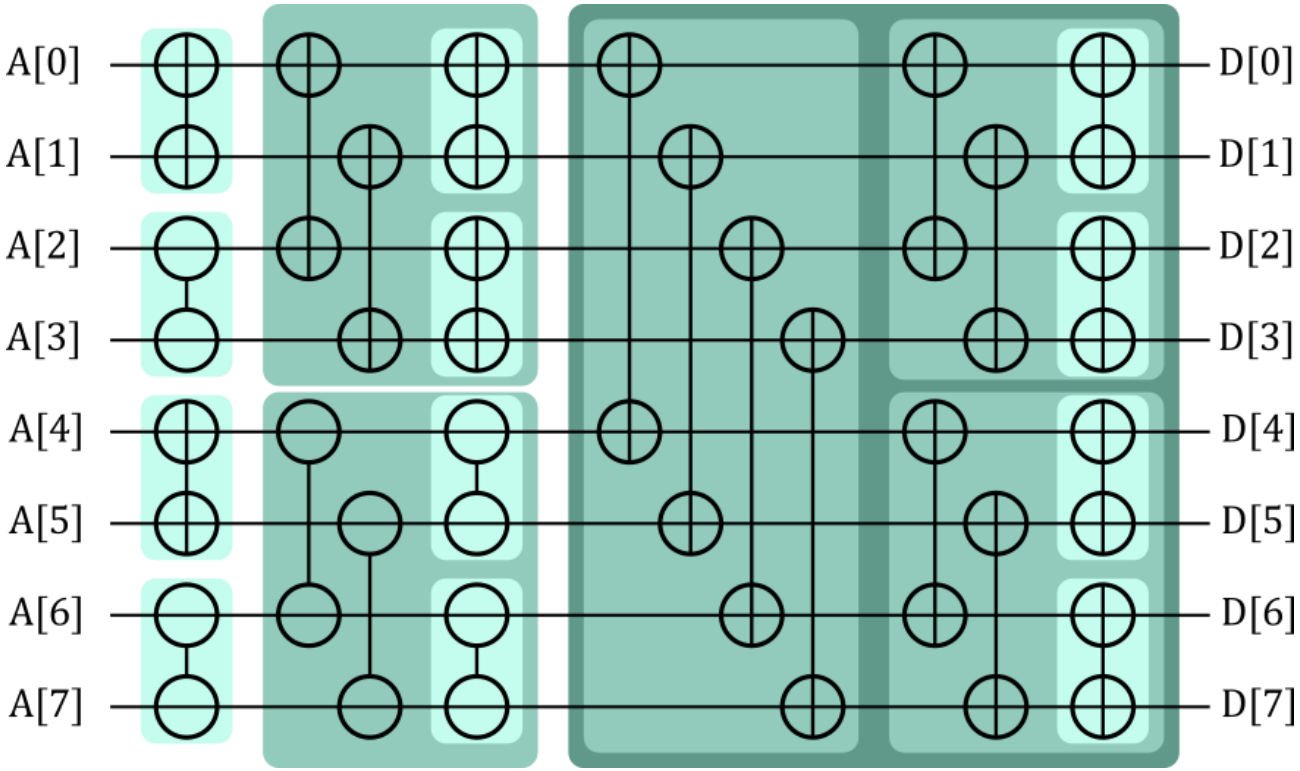
假设A[0]<=B[0]，
那么 min{a,c} = A[0]
A[0]不可能大于max{b,d}

假设A[0]>=B[0]，
那么max{*b*,*d*} = min{A[1],B[2]}
min{*a*,*c*} = min{A[0],B[1]}

然而min{A[0],B[1]}不可能大于min{A[1],B[2]}。 ■

Bitonic 排序网络

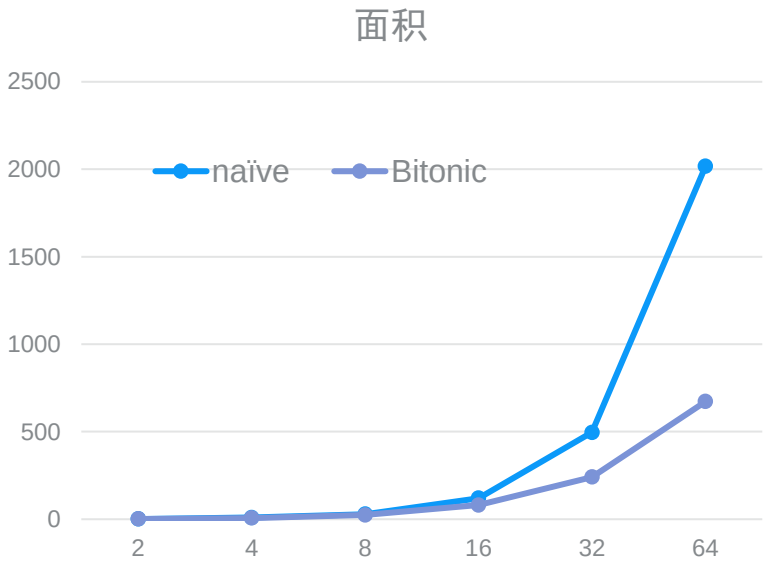
- 8-input Bitonic Sorting Network

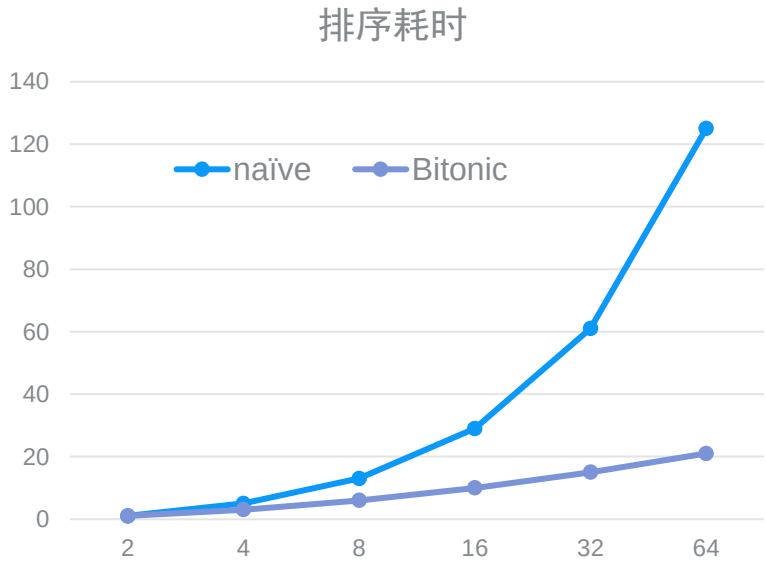


电路面积：$A \sim \frac{N(\log_2 N+1)\cdot\log_2 N}{4} \sim O(N \cdot \log_2^2 N)$

电路速度：$T \sim \frac{(\log_2 N+1)\cdot\log_2 N}{2} \sim O(\log_2^2 N)$

排序器比较





Naïve

电路面积：$A \sim \frac{N(N-1)}{2} \sim O(N^2)$

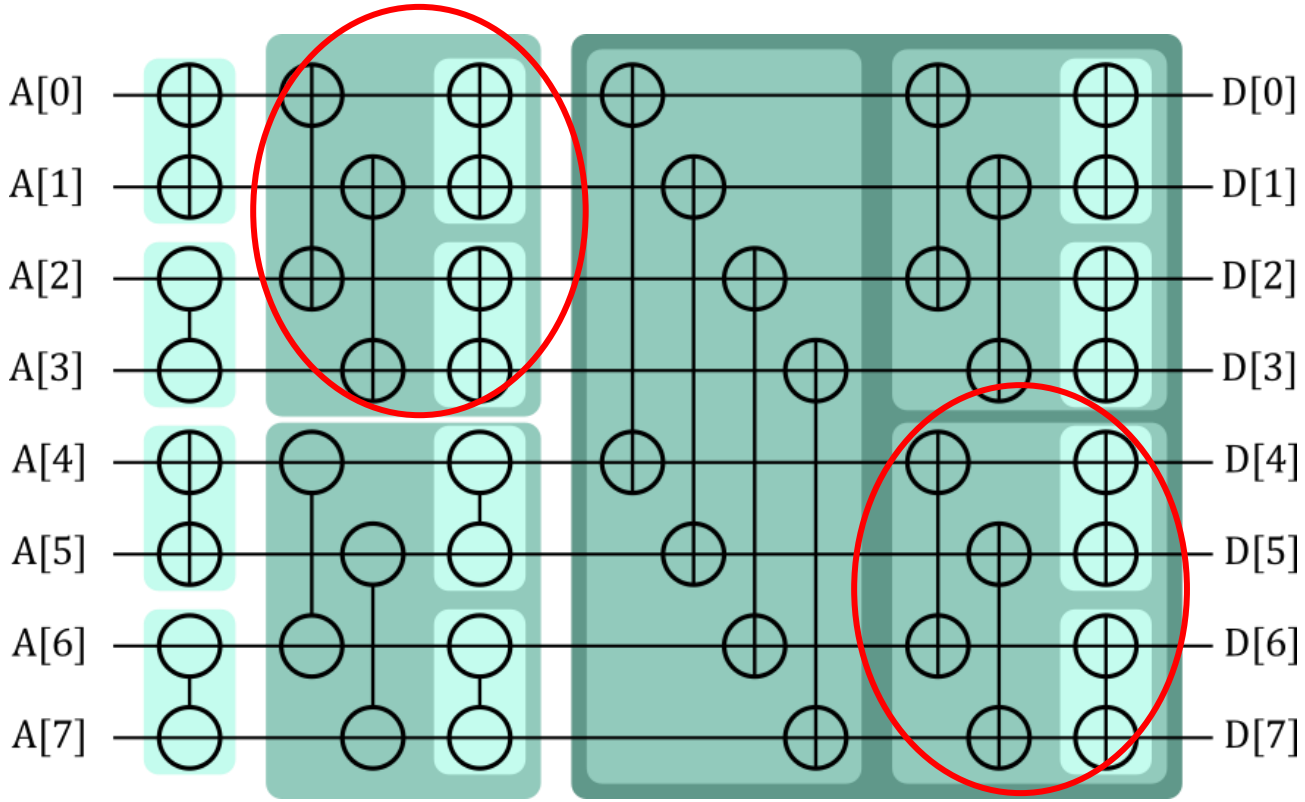
电路速度：$T \sim 2N - 3 \sim O(N)$

Bitonic

电路面积：$A \sim \frac{N(\log_2 N+1)\cdot \log_2 N}{4} \sim O(N \cdot \log_2^2 N)$

电路速度：$T \sim \frac{(\log_2 N+1)\cdot \log_2 N}{2} \sim O(\log_2^2 N)$

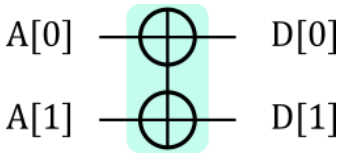
Bitonic 排序网络：如何实现?



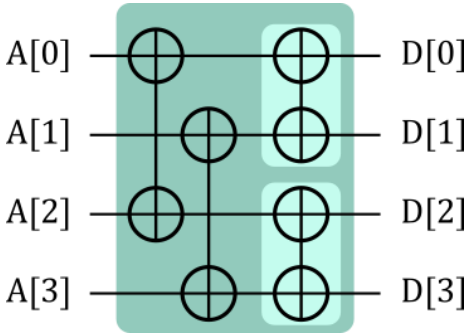
我们怎么用Verilog来实现一个Bitonic排序网络?

- 大的排序网络是由小的排序网络构成的
- 能实现小的排序网络，我们就能实现大的。

BM2和BM4

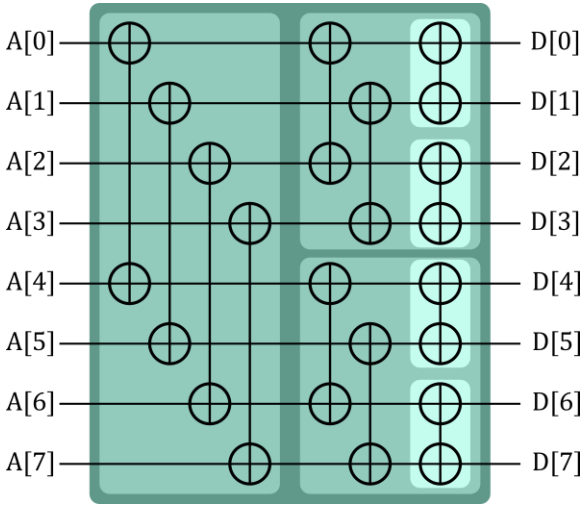



```
module BM2 #(parameter dw=8, dir = 0) (
  input [1:0][dw-1:0] d_in,
  output [1:0][dw-1:0] d_out
);
  wire swap;
  assign swap  = dir ^ (d_in[0] > d_in[1]);
  assign d_out = swap ? {d_in[0],d_in[1]} :d_in;
endmodule
```




```
module BM4 #(parameter dw=8, dir = 0) (
  input [3:0][dw-1:0] d_in,
  output [3:0][dw-1:0] d_out
);

  wire [3:0][dw-1:0] data;
  wire [1:0] swap;

  genvar i;

  generate for (i=0; i<2; i=i+1) begin
    assign swap[i]   = dir ^ (d_in[i] > d_in[2+i]);
    assign data[i]   = swap[i] ? d_in[2+i] : d_in[i];
    assign data[2+i] = swap[i] ? d_in[i]   : d_in[2+i];
  end endgenerate

  BM2 #(dw, dir) M0 (data[1:0], d_out[1:0]);
  BM2 #(dw, dir) M1 (data[3:2], d_out[3:2]);

endmodule
```

BM8



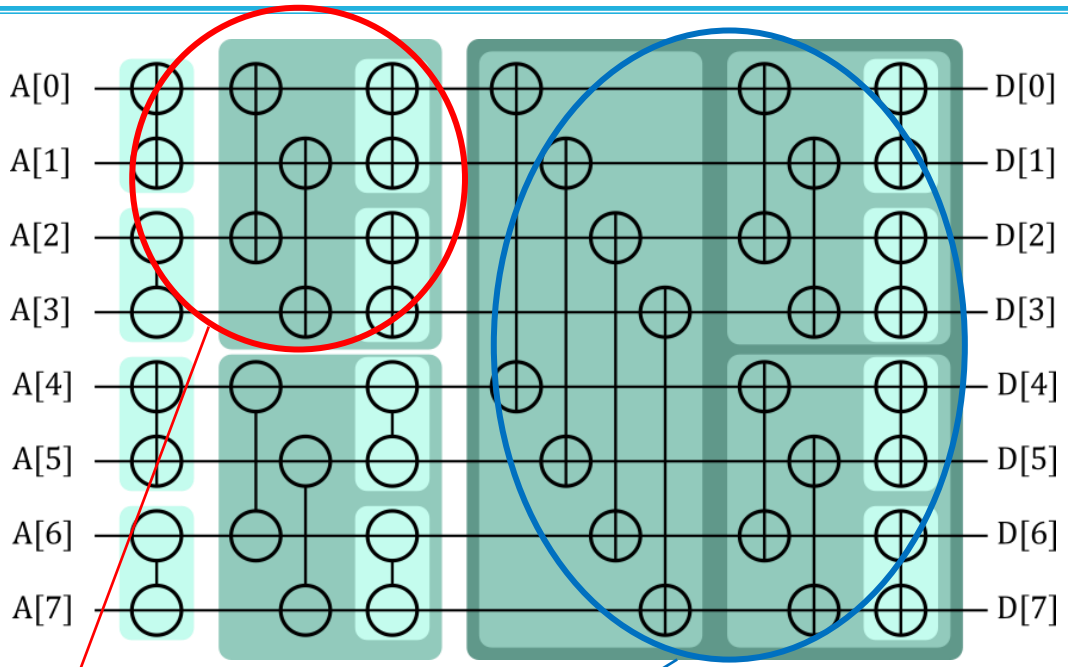

```
module BM8 #(parameter dw=8, dir = 0) (
  input [7:0][dw-1:0] d_in,
  output [7:0][dw-1:0] d_out
);

  wire [7:0][dw-1:0] data;
  wire [3:0] swap;

  genvar i;

  generate for (i=0; i<4; i=i+1) begin
    assign swap[i]   = dir ^ (d_in[i] > d_in[4+i]);
    assign data[i]   = swap ? d_in[4+i] : d_in[i];
    assign data[4+i] = swap ? d_in[i]   : d_in[4+i];
  end endgenerate

  BM4 #(dw, dir) M0 (data[3:0], d_out[3:0]);
  BM4 #(dw, dir) M1 (data[7:4], d_out[7:4]);

endmodule
```

N-port Bitonic Merger

```
module BM #(parameter LP=3, dw=8, dir = 0) (
  input [2**LP-1:0][dw-1:0] d_in,
  output [2**LP-1:0][dw-1:0] d_out
);
  localparam PN = 2**LP;
  localparam HPN = PN/2;
  wire [PN-1:0][dw-1:0] data;
  wire [HPN-1:0] swap;

  genvar i;

  generate
    if(LP==1)
      BM2 #(dw, dir) B(d_in, d_out);
    else begin
      for (i=0; i<HPN; i=i+1) begin
        assign swap[i]     = dir ^ (d_in[i] > d_in[HPN+i]);
        assign data[i]     = swap[i] ? d_in[HPN+i] : d_in[i];
        assign data[HPN+i] = swap[i] ? d_in[i]     : d_in[HPN+i];
      end
      BM #(LP-1, dw, dir) M0 (data[HPN-1:0],  d_out[HPN-1:0]);
      BM #(LP-1, dw, dir) M1 (data[PN-1:HPN], d_out[PN-1:HPN]);
    end
  endgenerate

endmodule
```

递归实例化

Bitonic 排序网络




```
module BN
#(parameter LP=3, dw=8, dir = 0)
(
  input [2**LP-1:0][dw-1:0] d_in,
  output [2**LP-1:0][dw-1:0] d_out
);
  localparam PN = 2**LP;
  localparam HPN = PN/2;
  wire [PN-1:0][dw-1:0] data;

  genvar i;

  generate
    if(LP==1)
      BM2 #(dw, dir) B(d_in, d_out);
    else begin
      BN #(LP-1, dw, 0) BN0 (d_in[HPN-1:0],  data[HPN-1:0]);
      BN #(LP-1, dw, 1) BN1 (d_in[PN-1:HPN], data[PN-1:HPN]);
      BM #(LP, dw, dir) BMOut (data, d_out);
    end
  endgenerate

endmodule
```

一个利用排序的仲裁器

2019年期末考试B卷最后一题：

七、请使用Verilog HDL设计一个三路（三个请求端）仲裁器，并满足以下特性（25分）：

（1）假设三个请求端分别是A、B和C：在每8个时钟周期的大周期内，A随机产生4次请求，B随机产生2次请求，C随机产生1次请求；

（2）仲裁器的输出端一直都处于空闲状态（总是可以接受请求）；

（3）仲裁器将尽快响应来自A、B和C的所有请求(三个端口的平均响应延迟大致相等并最小)。

题解：如果仲裁器的输出一直可以响应请求，8个周期可以响应8次请求。三个请求端的总平均请求数量为7，小于8，所以不会发生饿死。为了降低平均等待延时，让各端口的权值和频率呈正比，然后等待的越久优先级越高。

假设每个端口的请求等待时间为t_P，那么，各端口的权重为

$$W_A = t_A \cdot 4$$
$$W_B = t_B \cdot 2$$
$$W_C = t_C \cdot 1$$

然后，仲裁器优先响应权重值最高的端口。

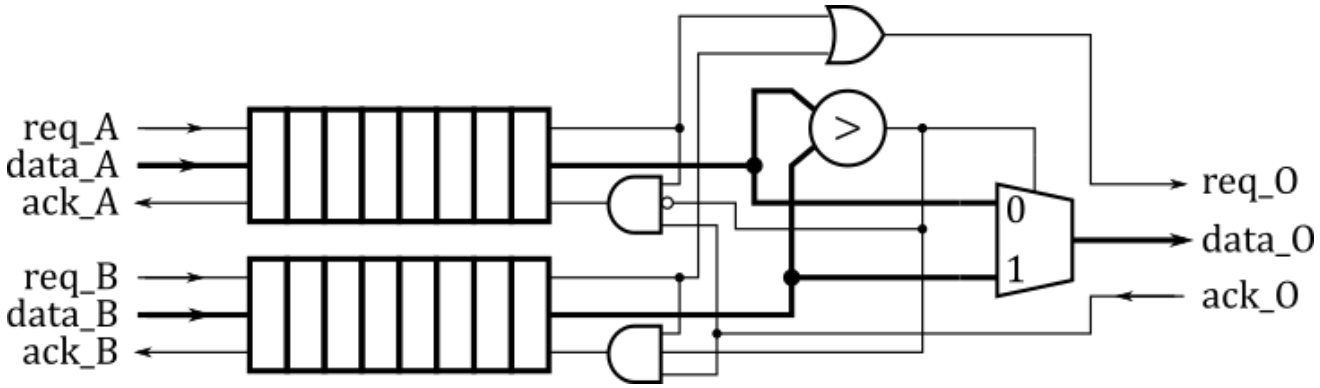
一个利用排序的仲裁器

```
module arbiter(input  clk, rstn,
               input  [2:0] req_in,  output [2:0] ack_in,
               output       req_out, input        ack_out);

  wire [2:0] fire;
  assign fire = ack_out ? (req_in & ack_in) : 3'b0;

  reg [2:0][5:0] weight_pre;
  wire [2:0][5:0] weight;

  reg [1:0] sel;

  assign weight[0] = req_in[0] ? weight_pre[0] + 4;
  assign weight[1] = req_in[1] ? weight_pre[1] + 2;
  assign weight[2] = req_in[2] ? weight_pre[2] + 1;

  always @(posedge clk or negedge rstn)
    if(!rsnt) weight <= 0;
    else begin
      weight_pre[0] <= fire[0] ? 0 : weight[0];
      weight_pre[1] <= fire[1] ? 0 : weight[1];
      weight_pre[2] <= fire[2] ? 0 : weight[2];
    end

  always @(weight) begin
    sel = 0;
    if(weight[1] > weight[sel]) sel = 1;
    if(weight[2] > weight[sel]) sel = 2;
  end

  assign ack_in = (req_out && ack_out) ? (3'b001 << sel) : 3'd0;
  assign req_out = |req_in;

endmodule
```

任意长度序列的排序器?

- 利用Bitonic 排序网络可以排序一个固定长度的序列
- 如何利用硬件来排序一个任意长度的序列?
  - 首先，单周期排序肯定是不现实的了
  - 引入多周期的排序
  - 常见的硬件排序算法：归并排序
  - 利用两个先入先出队列和一个CAE算子构造最简单的无限长度归并排序器。

无限长度的归并排序器




```
module seq_merger #(parameter dw=8, L=16) (
  input  clk, rstn,
  input  [dw-1:0] data_a, input  req_a, output ack_a,
  input  [dw-1:0] data_b, input  req_b, output ack_b,
  output [dw-1:0] data_o, output req_o, input  ack_o
  );

  wire [dw-1:0] A, B;
  wire Areq, Aack, Breq, Back, sel;

  fifo #(.L(16), .dw(dw)) fifoA (clk, rstn, data_a, req_a, ack_a, A, Areq, Aack);
  fifo #(.L(16), .dw(dw)) fifoB (clk, rstn, data_b, req_b, ack_b, B, Breq, Back);

  assign sel = A > B;
  assign data_o = sel ? B : A;
  assign req_o = Areq | Breq;
  assign Aack = Areq & ack_o & !sel;
  assign Back = Breq & ack_o & sel;

endmodule
```

其实这里还有一个问题没解决，
能看出问题吗？

排序器总结

- 单周期的组合逻辑排序器
  - 直接将冒泡排序硬件化（速度慢、面积大）
  - 利用Bitonic 排序网络
  - Bitonic排序网络的推导
- 任意长度的归并排序器
  - Bitonic排序网络的端口数固定
  - 利用时序逻辑
  - 双路归并排序器

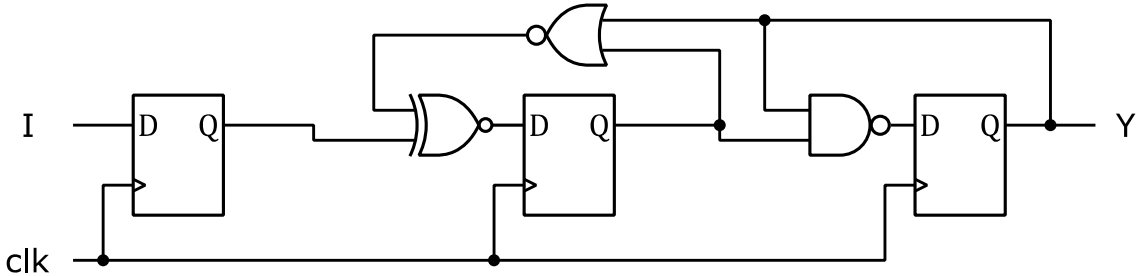
时序逻辑电路的时序分析

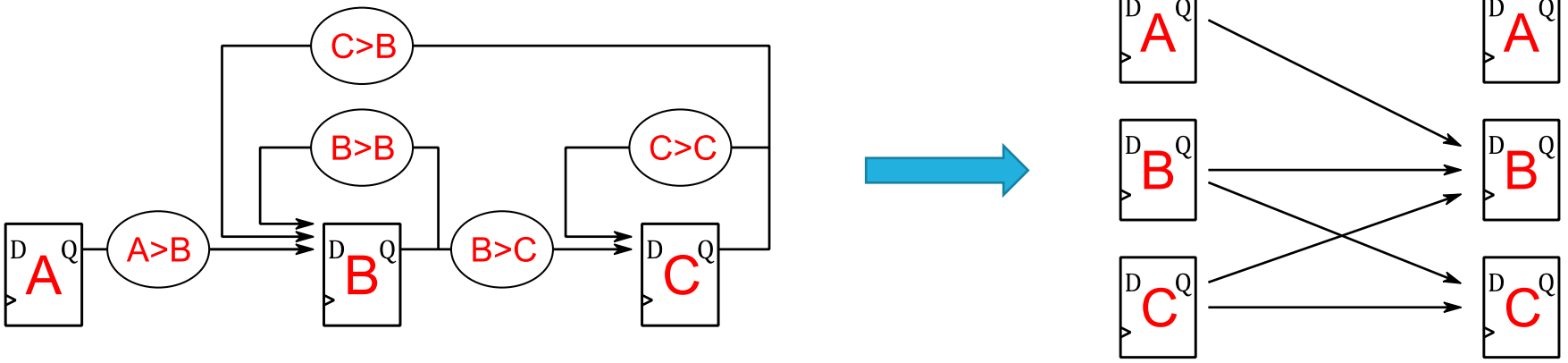
- 已有知识
  - **组合逻辑电路的时序分析**
    - 电路的关键路径
    - 电路的延时计算
  - **寄存器的时序分析**
    - 寄存器的保持和建立时间
    - 寄存器的传输延时

- 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

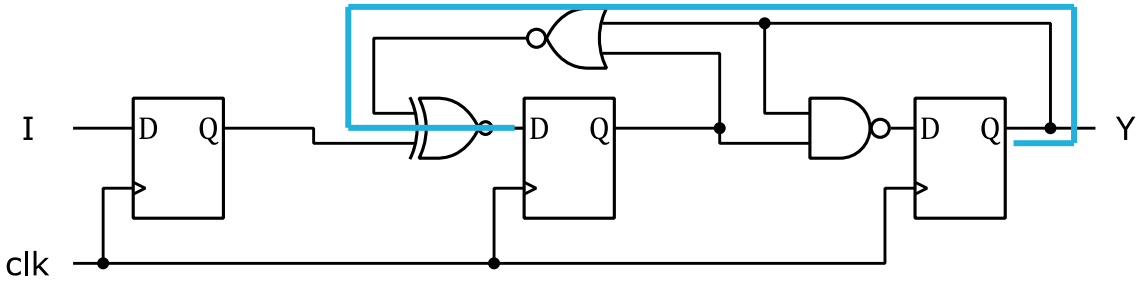
- 我们如何能知道最高频率?





X>Y: 从X到Y的路径群。

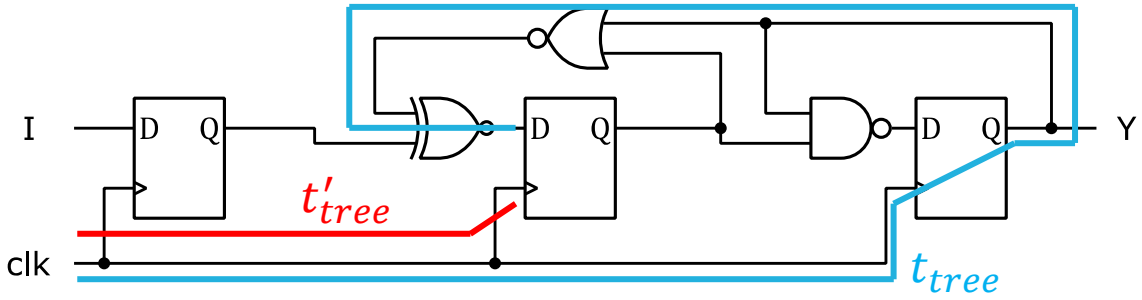
分析一条路径的延时



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

- 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

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

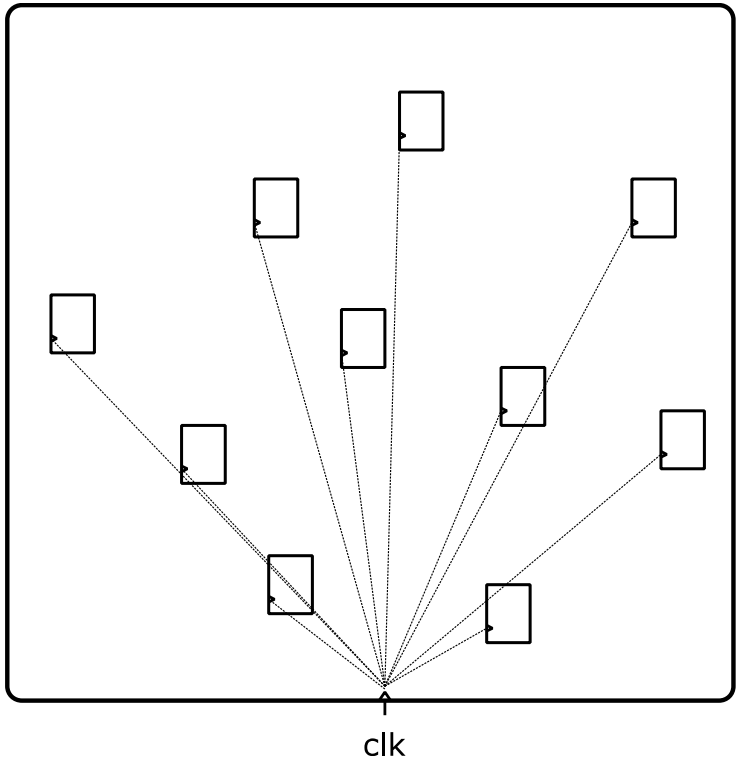
所以

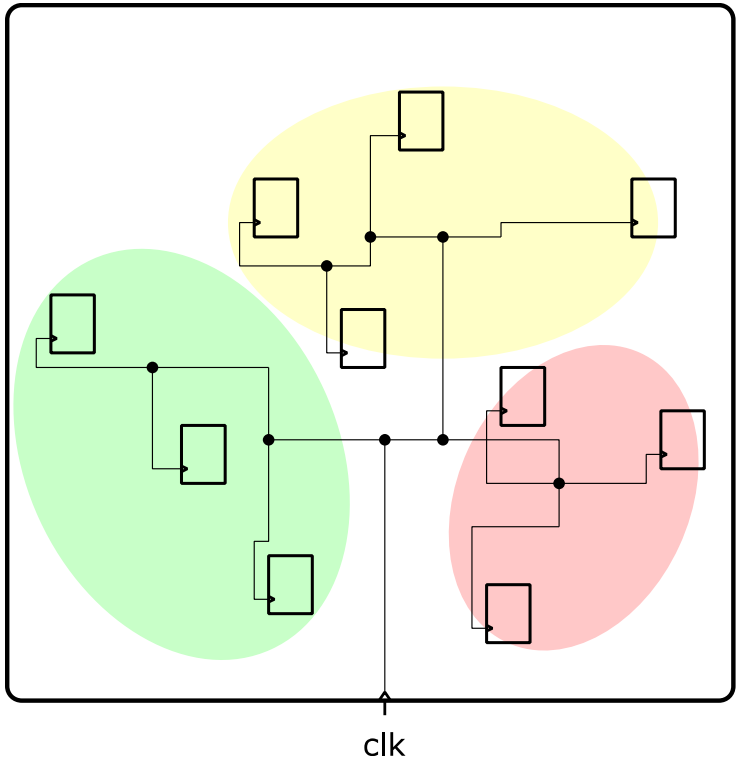
$$T_{min} > \underbrace{(t_{tree} - t'_{tree})}_{\text{时钟树偏差}} + \underbrace{t_{su}}_{\text{建立时间}} + \underbrace{t_{pd}(R)}_{\text{传输时间}} + \underbrace{t_p(NOR) + t_p(NXOR)}_{\text{路径延时}}$$

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

时钟树的构造与平衡

- 时钟树的平衡，影响电路的频率



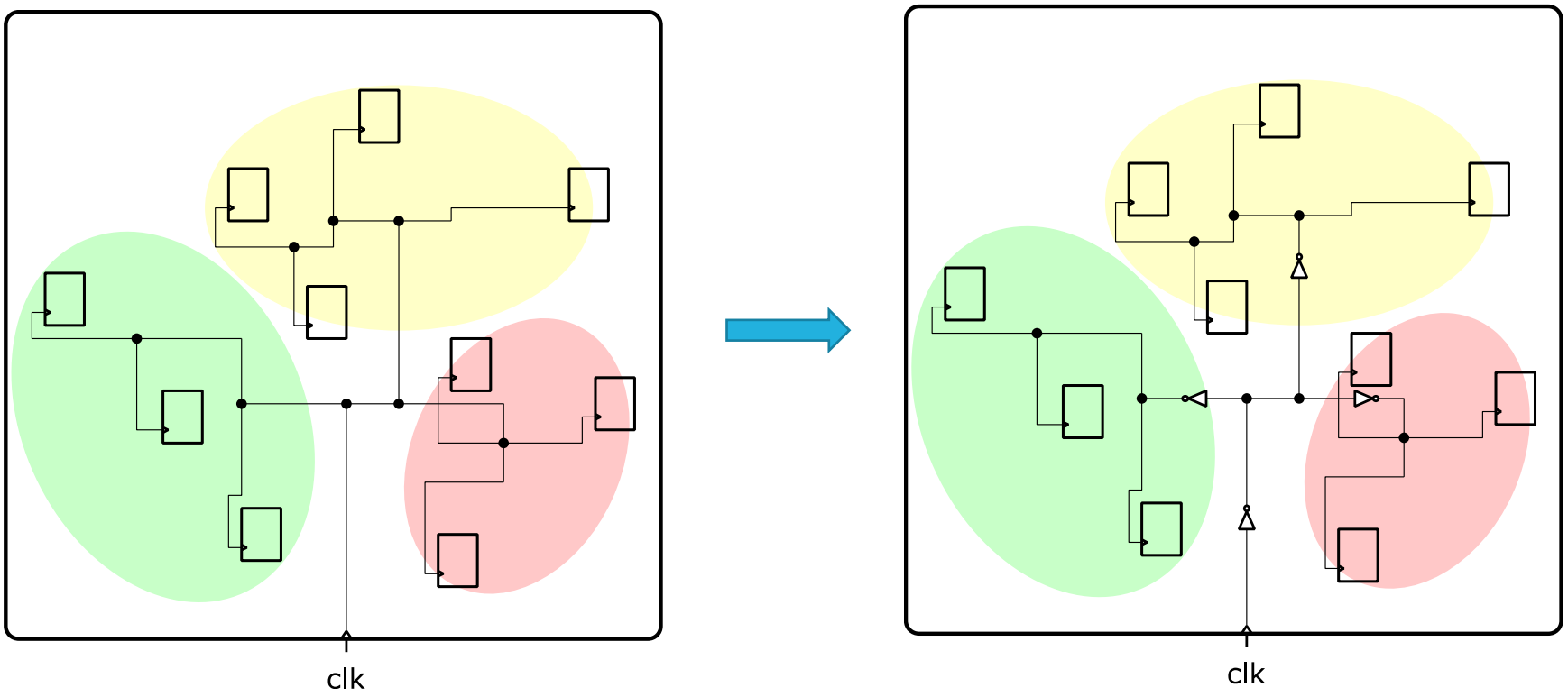


- 将时钟引到芯片中间
- 寄存器分片，片内时钟继续均匀分布
- 形成一个树形结构

时钟信号的负载很大，
导致翻转时间长！

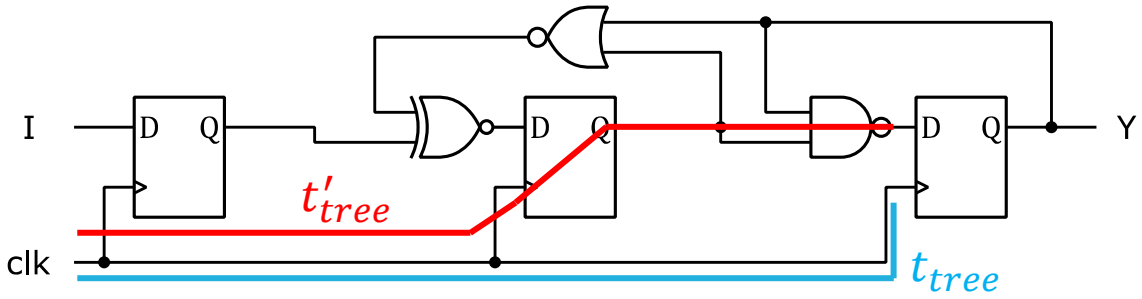
时钟树的构造与平衡

- 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

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

$t_p(NAND)$：路径延时；$(t_{tree} - t'_{tree})$：时钟树偏差；$t_h(R)$：保持时间；$t_{pd}(R)$：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

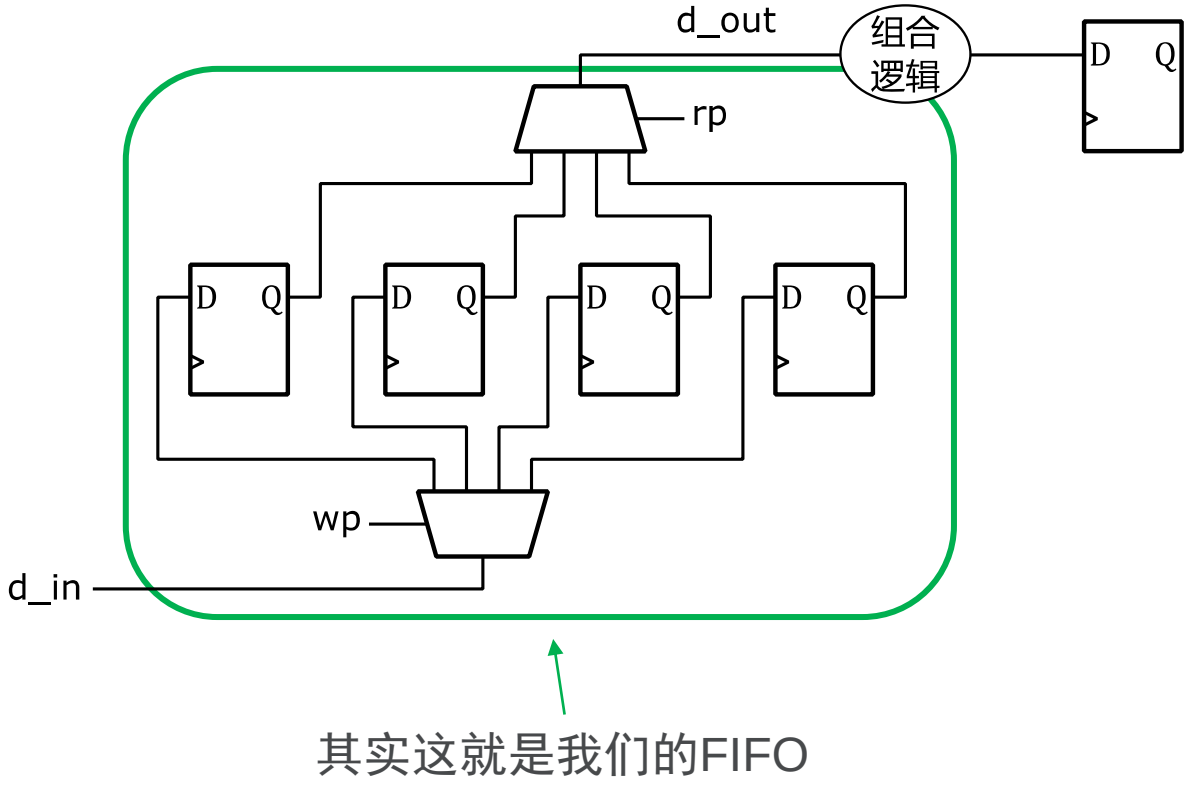
$$T_{min} > (t_{tree} - t'_{tree}) + t_{su} + t_{pd}(R) + t_{path}$$

- 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

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

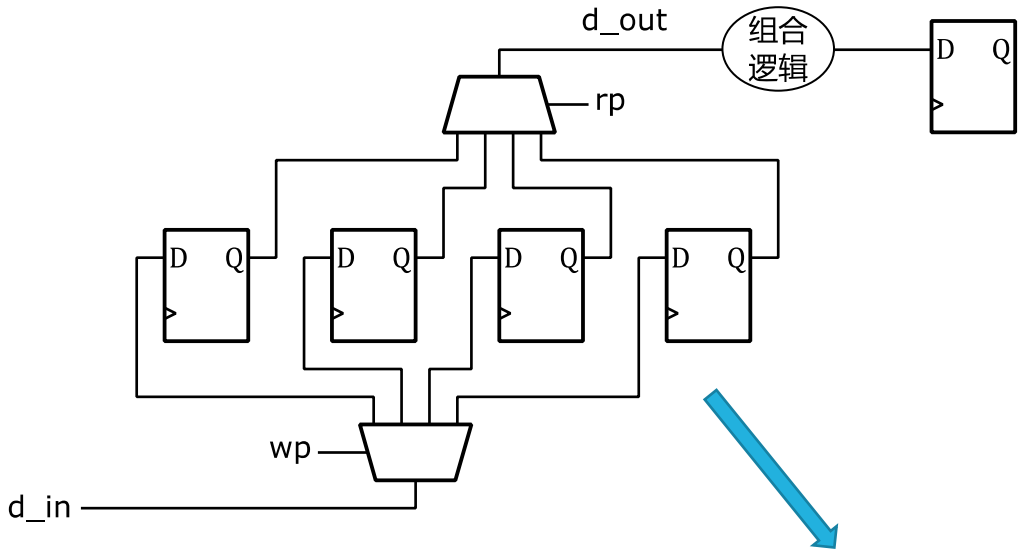
FIFO缓冲器的时序问题

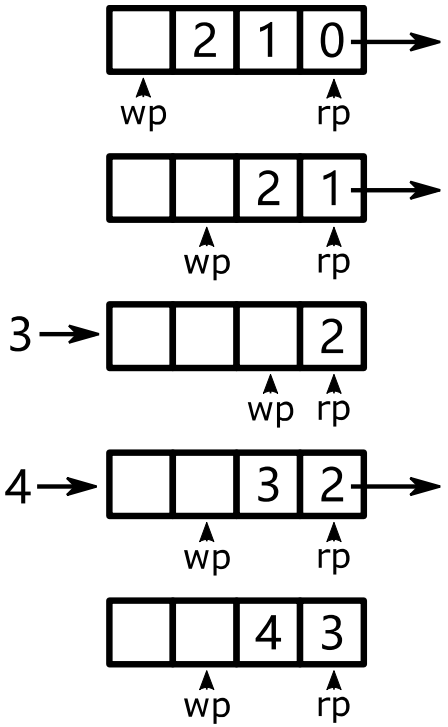
- 如果FIFO的输出级时序紧张怎么办？

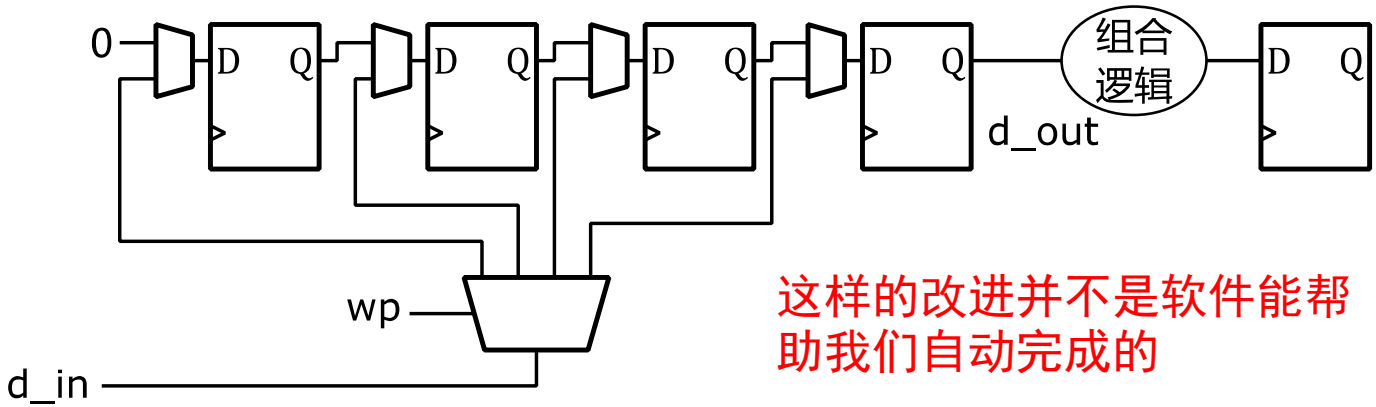


FIFO缓冲器的时序问题

- 如果FIFO的输出级时序紧张怎么办？



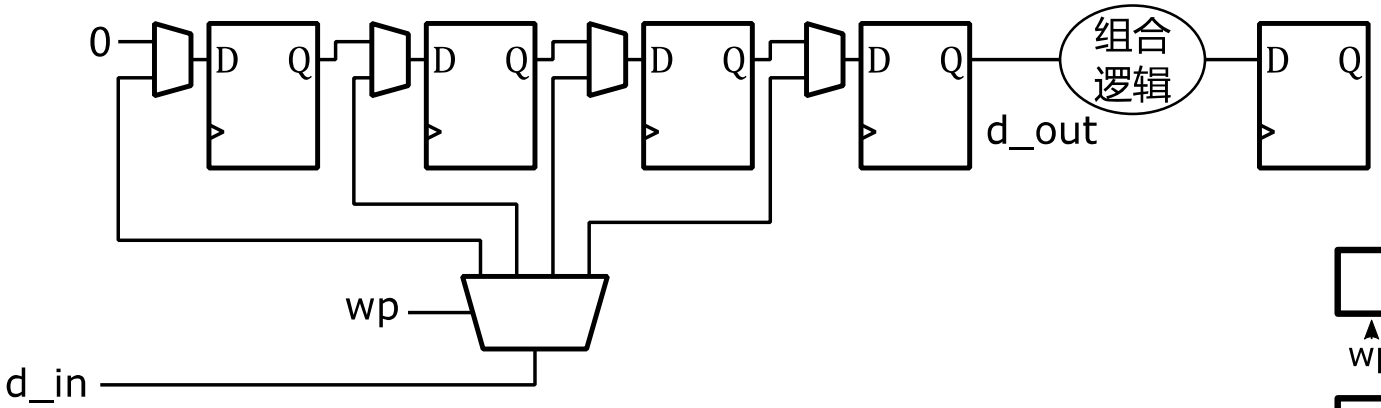


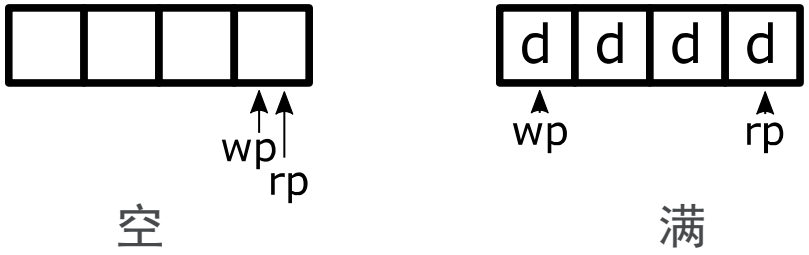


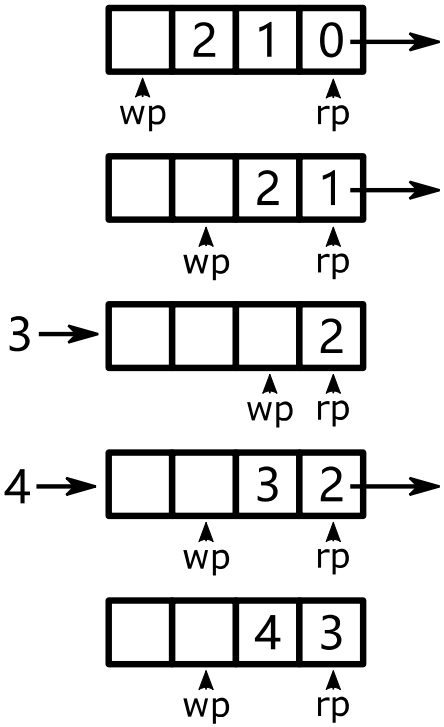
这样的改进并不是软件能帮助我们自动完成的

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
```

电路的综合

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

考试范围

- 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
  - 排序器
  - 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?

作业

编程题 1：

用Verilog HDL编写一个4路随机仲裁器。

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。

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。

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。