电子科技大学

University of Electronic Science and Technology of China

# 本科生实验报告

**实验课程：** 复杂数字集成电路设计（挑战性课程）

**实验名称：** 实验 5 固定优先级与轮换优先级仲裁器

**实验地点：** 无

**学生姓名：** 周岳恒

**学　　号：** 2021340105016

**指导教师：** 廖永波

**实验时间：** 2023 年 10 月 1 日

# 一、 实验目的

理解仲裁器的使用背景，了解固定优先级仲裁器（Fixed Priority Arbiter）和轮换优先级仲裁器（Round Robin Arbiter）。学会设计固定优先级仲裁器（组合逻辑）和轮换优先级仲裁器（包含时序逻辑），且尝试用不同的方法简化设计实现轮换优先级仲裁器。

# 二、 实验任务

1. 了解仲裁器的使用背景；
2. 根据需求设计固定优先级仲裁器；
3. 根据需求设计轮换优先级仲裁器；
4. 尝试用不同于状态机的方法实现轮换优先级仲裁器。

# 三、 实验原理

在通信电路中存在主设备和从设备。只有主设备有权限主动和各个从设备联络通信，从设备只能相应主设备的请求而不能发起通信请求。在片上总线结构中，可能存在一个主设备或多个主设备的情况。如果有多个主设备同时请求发起通信，而针对一个主线只允许在同一时间内允许一个主设备发起通信，所以此时需要有仲裁器仲裁各个主设备的通信权限。

固定优先级仲裁表示各个主设备的优先级是固定的，当有更高优先级的主设备请求时，低优先级的主设备只有等到更高优先级的主设备完成通信后才能使用主线进行通信。

轮换优先级仲裁表示在每一个时钟边沿优先级按照一个方向轮换改变，当某设备根据优先级得到通信权限后，下一个时钟这个设备优先级变成最低，相邻的设备优先级依次变成最大，次大，等等。从而保证总线上各个主设备的优先级都可能最高，从而避免总线资源不能平均分给各个设备的情况。

固定优先级仲裁器的输入为各个设备请求，高为发起请求，低为空闲。假定信号线地位优先级最高，依次递减。输出为独热码，对应各个设备的应答信号，最多只有一个位为高。

轮换优先级仲裁器的优先级依次递减，且最高优先级对应位循环左移，输入输出同固定优先级仲裁器。

## 四、 实验过程

**固定优先级仲裁器：**

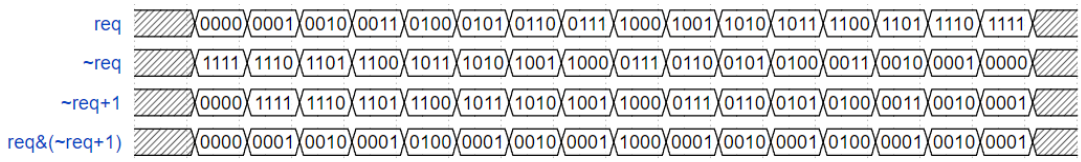
verilog 代码：

```
module fixed_prior_arb #(
    parameter N = 4
)(
    input [N-1:0] req,
    input enable,
    output [N-1:0] gnt,
    output valid
);
    /* use gate level description */
    assign gnt = req & (~req + 1'b1); // lower bit has higher priority
    assign valid = enable & (|gnt);

endmodule

module fixed_prior_arb_behav #(
    parameter N = 4
)(
    input [N-1:0] req,
    input enable,
    output [N-1:0] gnt,
    output valid
);
    /* use behavior level description */
    integer i;
    reg [N-1:0] result;
    reg find;

    always @(req) begin
        result = 0;
        find = 0;
        for(i = 0; i < N; i = i +1) begin
            if(req[i] & ~find) begin
                result[i] = 1'b1;
                find = 1'b1;
            end
        end
    end
    assign gnt = result;
    assign valid = enable & (|gnt);
```

```
endmodule

module fixed_prior_arb_cons #(
    parameter N = 4
)(
    input [N-1:0] req,
    input enable,
    output [N-1:0] gnt,
    output valid
);
    /*use structure level description */
    wire [N-1:0] find;
    assign find = {find[N-2:0] | req[N-2:0], 1'b0};
    assign gnt = req & ~find;
    assign valid = enable & (|gnt);

endmodule
```

我设计了三种方式描述：

**门级描述：**对于 gnt：假设输入 req 低位优先级最高，输出 gnt 的任务是找到 req 从低位开始的第一个 1 并给对应位置 1，其他位置 0。这里我们选择一个取巧的办法：一个数和它的补码相与结果是独热码，且此独热码正好是我们所需求的独热码。因为对应的输入中我们应该应答的那一位是 1，它之前的所有低位必须都是 0，保证那一位会被响应。把 req 取反后那一位应答位位 0，比它低的所有位都是 1，比它高的所有位 0 和 1 转换。则~req+1 的结果是：比应答位低的所有位加 1 后全为 0，应答位因为之前是 1 所以现在是 0，而高位则不变。req&(~req+1)的结果就是：应答位 1&1=1，低位 0&1=0，高位 a&~a=0 实现了目标。




而 valid 信号的输出取决于 enable 为 1 和 gnt 有效（产生独热码而不能是全 0，全 0 表示输入无效，输入没有请求发生）。

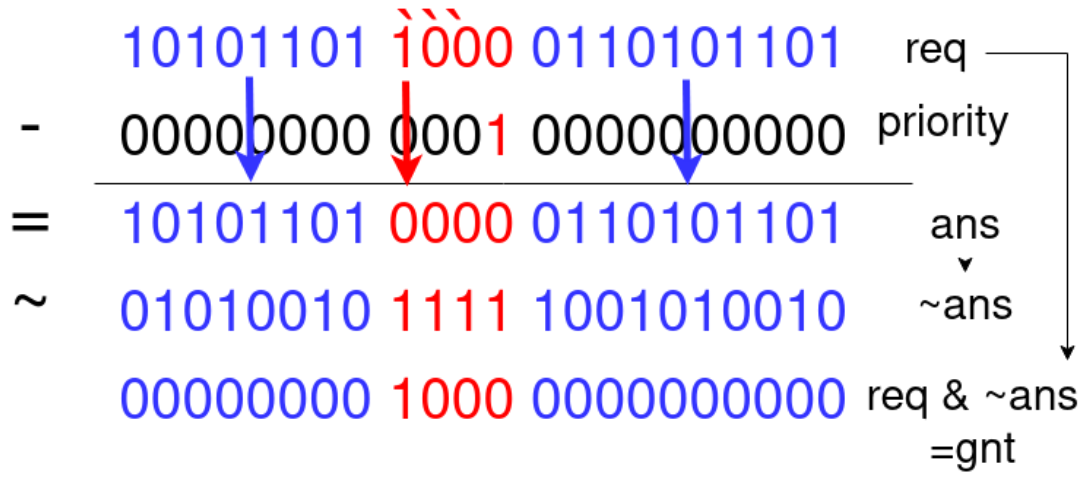



图 1 固定优先级计算原理

让我们对这一算法进行扩展：本质上，上述的~req+1 是取补码的过程，这一过程等价于~(req-1)，把顺序颠倒，先做减法再取反和先取反再做加法的结果一样，因为取反相当于把一个数变换到这个数到 0 的距离等于新变换的那个数到模的距离，减 1 拉近与 0（模）的距离，加 1 拉远与 0（模）的距离，所以结果都是与 req 到 0 的距离-1 就是新数到模的距离。所以，我们都改成~(req-1)方便理解。

```
module fixed_prior_arb_give #(
parameter N = 4
)(
input [N-1:0] req,
input enable,
input [N-1:0] priority,
output [N-1:0] gnt,
output valid
);
/* use gate level description */
wire [2*N-1:0] req_double;
wire [2*N-1:0] gnt_double;
assign req_double = {req, req};
assign gnt_double = req_double & ~(req_double - priority);

assign gnt = gnt_double[N-1:0] | gnt_double[2*N-1:N];
assign valid = enable & (|gnt);
endmodule
```

减 1 是最低位优先级最高，在一般的情况下可以指定减数控制优先级：哪一位为 1，它的优先级最高，向左依次循环优先级递减，到达末尾进行轮换递减优

先级，直到为 1 的那一位的左侧，如图 1 所示。

验证此算法的正确性：保证 priority 的第 n 位为 1，其他为均为 0。则在做减法 req-priority 时，req 低于 n 的位减去的是 0，所以结果保持不变；从第 n 位开始，req 的相应位相减后如果一直遇到 0，则一直向前借位，且结果为 0，直到遇到 1 停止向前借位，假设这一位为 m。这些位做减法的结果都是 0，借位的原来是 0 结果是 0，最后不借位的位 m 原来是 1 结果为 0，相当于仅仅对这一位 m 取反。而更高位因为不借位，相减结果不变。总结：req 只有从第 n 位开始向上第一个 1（对应第 m 位）结果取反，其他位不变。那么~(req-priority)&req 就是只有第 m 位为 1，其他位为 0，因为 a&(~a)=0。

考虑特殊情况：当借位到最高位时还没有遇到 1，相当于此时优先级要轮回从低位开始递减查找，此时出现溢出，结果不正确。为了避免这种情况，这里我用 double_req={req, req}使高位连接 req 的低位，这样可以确保在 req 有效（|req!=0）的情况下得到正确的结果。最后得到 gnt 为 double_gnt 的 N-1~0 位和 2N-1~N 位相或，因为 double_gnt 是独热码，相应位 m 结果为 1|0=1，得到的结果也是独热码，和结果一致。而如果没有借更高高位导致轮回，那么 2N-1~N 位也都是 0，由于 a|0=a 也不影响结果，可以验证算法的正确性。

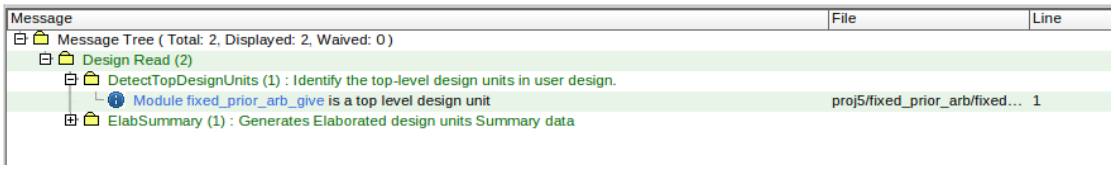
**行为级描述：**和软件的写法一样，spyglass 会产生 warning，不建议这样写。

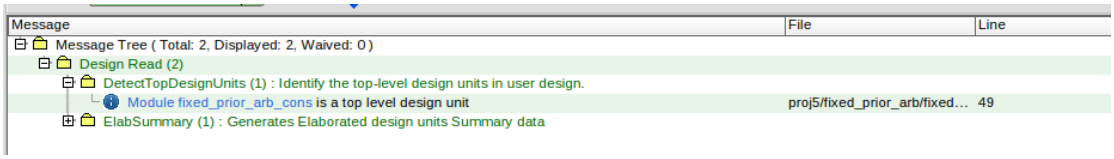
**结构级描述：**设置 find 信号，使 find 每一位都是 find 的上一位和 req 这一位的低一位相或，find 最低位置 0。这样形成菊式链，保证在减少重复使用逻辑资源的情况下实现 find[i]＝|req[i-1:0]的功能。find 反映对应位之前更低位有没有出现 1，所以 find 从应答位高一位开始高位都是 1（不包括应答位，它是 0），更低位都是 0。最后得到的 gnt 为~find&req，~find 保证 req 的应答位和更低位为 1，其他位为 0。和 req 相与则只留下 req 应答位和低位部分，其他都置 0。同样可以实现目标效果。

valid 信号和上述一样，不再赘述。

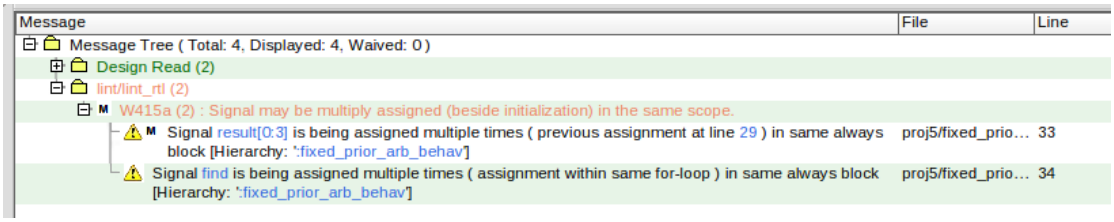
spyglass：

fixed_prior_arb_gate(give), fixed_prior_arb_cons：PASS.






fixed_prior_arb_behav：*reg is being assigned multiple times.*




**轮换优先级仲裁器：**

verilog 代码：

```
module round_robin_arbiter_fsm(
    input clk,
    input rst_n,
    input [3:0] req,
    input enable,
    output [3:0] gnt,
    output valid
);
    reg [3:0] cur_state,nxt_state;
    localparam IDLE = 4'b0000,
               S1   = 4'b0001,
               S2   = 4'b0010,
               S3   = 4'b0100,
               S4   = 4'b1000;

    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            cur_state <= IDLE;
        else
            cur_state <= nxt_state;
    end
    always @(*) begin
        if(enable) begin
            case (cur_state)
                S1: nxt_state =     (req[1]) ? S2 : (
                                   (req[2]) ? S3 : (
                                   (req[3]) ? S4 : (
                                   (req[0]) ? S1 : IDLE )));
```

```
                S2: nxt_state =     (req[2]) ? S3 : (
                                    (req[3]) ? S4 : (
                                    (req[0]) ? S1 : (
                                    (req[1]) ? S2 : IDLE )));

                S3: nxt_state =     (req[3]) ? S4 : (
                                    (req[0]) ? S1 : (
                                    (req[1]) ? S2 : (
                                    (req[2]) ? S3 : IDLE )));

                default: nxt_state =(req[0]) ? S1 : (
                                    (req[1]) ? S2 : (
                                    (req[2]) ? S3 : (
                                    (req[3]) ? S4 : IDLE )));
            endcase
        end
        else
            nxt_state = IDLE;
    end

    assign gnt = cur_state;
    assign valid = enable & (|gnt);

endmodule

module round_robin_arbiter #(
    parameter N = 4
)(
    input clk,
    input rst_n,
    input [N-1:0] req,
    input enable,
    output [N-1:0] gnt,
    output valid
);
    reg [N-1:0] gnt_reg;
    reg [N-1:0] mask_reg;
    wire [N-1:0] mask;
    wire [N-1:0] mask_out;
    wire routine;
    assign mask = ~gnt_reg & ~(gnt_reg - 1) & mask_reg;
    assign mask_out = req & mask;
    assign routine = ~(|mask_out);
```

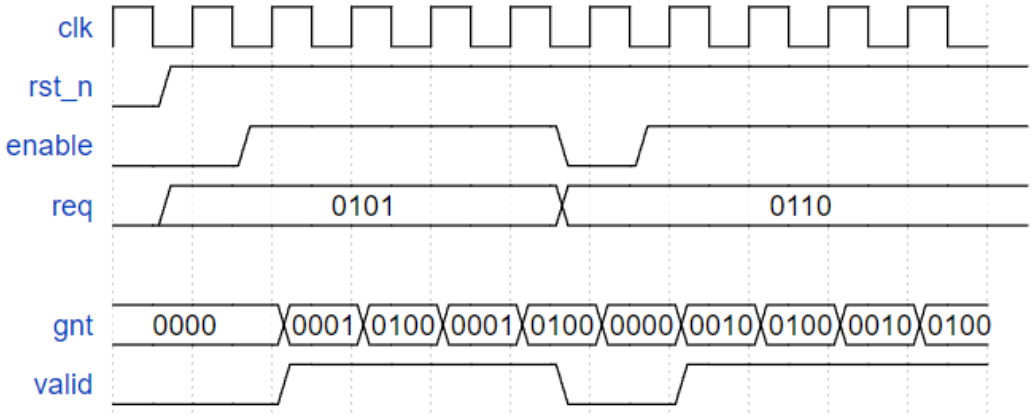
```
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            mask_reg <= {N{1'b1}};
        else if(enable)
            mask_reg <= routine ? {N{1'b1}} : mask;
        else
            mask_reg <= {N{1'b1}};
    end

    wire [N-1:0] req_sel;
    assign req_sel = routine ? req : mask_out;
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            gnt_reg <= {N{1'b0}};
        else if(enable)
            gnt_reg <= req_sel & (~req_sel + 1'b1);
        else
            gnt_reg <= {N{1'b0}};
    end
    assign gnt = gnt_reg;

    reg valid_reg;
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            valid_reg <= 1'b0;
        else if(enable & |req)
            valid_reg <= 1'b1;
        else
            valid_reg <= 1'b0;
    end
    assign valid = valid_reg & enable;

endmodule
```

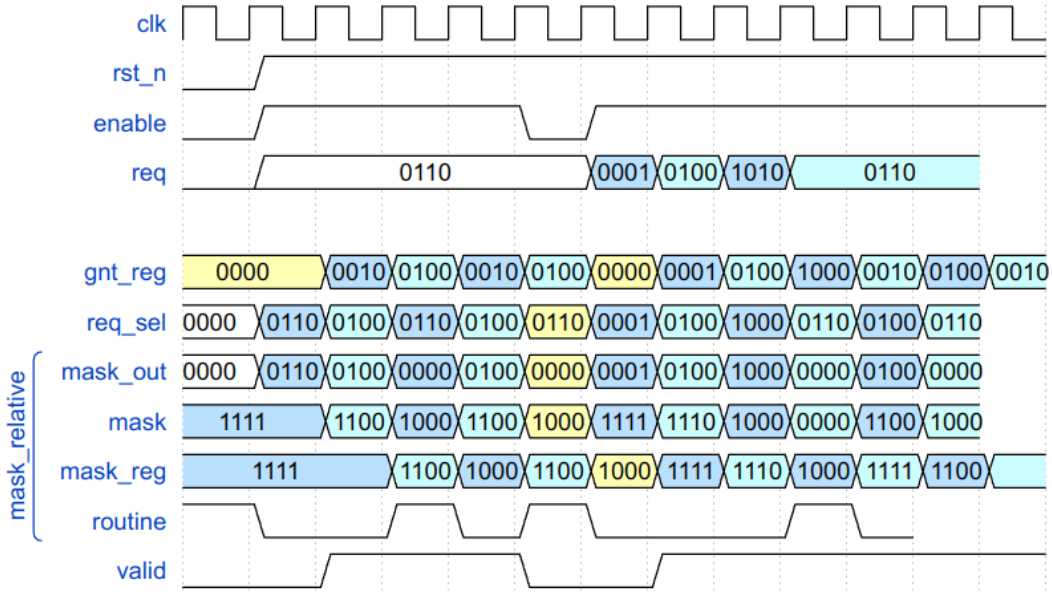
一个比较简单的思路是设计**状态机**，不过缺点是占用资源过多，且代码很长，如果申请位数过多则给代码编写和电路规模控制带来很大的难度，逻辑资源比较浪费，算作简单方案，但不够好。对于 n 路请求，状态机设计 n+1 个状态，其中 n 个状态就是对应位应答，另有一个状态是 IDLE 空闲状态。不使能时在 IDLE 状态，使能后状态使用级联多路复用器的方式，包含优先级循环查找下一个有效的位并跳转置所在位对应状态。这样的查找允许跳过多个位，也允许轮回过所有位（高位都是 0）就从比自身更低的位开始查找有效位。如果请求信号全为 0 则

跳转到 IDLE 状态使输出无效。gnt 直接输出各个状态寄存器的置，在设置状态编码时就设置成独热码了。

另一个思路是使用**掩码**的方式对输入队列信号做一次处理后改变优先级，再连接之前的固定优先级仲裁器实现仲裁。基本思想是：在每一个时钟周期内进行依次仲裁，之后记录本次仲裁对应位到寄存器中，形成掩码，下一次仲裁就把这一位开始到低位掩掉从而改变优先级，从更高的位依次寻找，实现改变优先级的功能。有一个特殊情况是掩码后所有的位都是 0，此时应加入多路选择器取消掩码，从低位开始判断优先级，可以实现对应的功能。

波形：




在这张图中，gnt_reg 表示输出结果（用寄存器延迟一拍使结果在时钟边沿有效），mask 表示掩码，req_sel 表示 gnt_reg 的来源。同一颜色表示使用同一相

位的数据，如果同一颜色有的数据延迟一个时钟周期，说明它是寄存器保持的值，其他都是组合逻辑产生的信号。在这里，gnt=req_sel & ~(req_sel-1)即 req_sel 位固定优先级仲裁器的输入。那么 req_sel 又是怎么来的呢？根据上面的表述，req_sel 应该是掩码后的结果或由于高位都是 0 重新多路复用到 req 上，判断的基准就是在掩码掩掉后临时产生的 gnt_reg 输出会不会都是 0，如果都是 0 就输入 req 且把 mask_reg 初始化，不然输出掩码结果 mask&req。而掩码的产生来源于每次 gnt_reg 输出和上一次掩码的记录 mask_reg，因为每次不仅要掩掉之前的一位，还要掩掉更早的低位，所以要它们两作为 mask 的产生来源。把 gnt_reg 输出取非后和 mask_reg 相与就是 mask。

在轮换优先级仲裁器中 valid 信号由组合逻辑和时序逻辑两者决定，组合逻辑观察 enable 有效 valid 才有效，时序逻辑观察在时钟边沿才保证 mask 掩码后的结果输出到 gnt_reg 上，需要同步打拍。

使用寄存器保存 gnt_reg 结果控制 **priority 实现轮换优先级**比较：

```
module round_robin_arbiter_give #(
parameter N = 4
)(
input clk,
input rst_n,
input [N-1:0] req,
input enable,
output [N-1:0] gnt,
output valid
);

reg [N-1:0] priority_reg;
wire [N-1:0] gnt_wire;
always @(posedge clk, negedge rst_n) begin
if(~rst_n)
priority_reg <= 1;
else if(enable & |req)
priority_reg <= {gnt_wire[N-2:0], gnt_wire[N-1]};
end
```

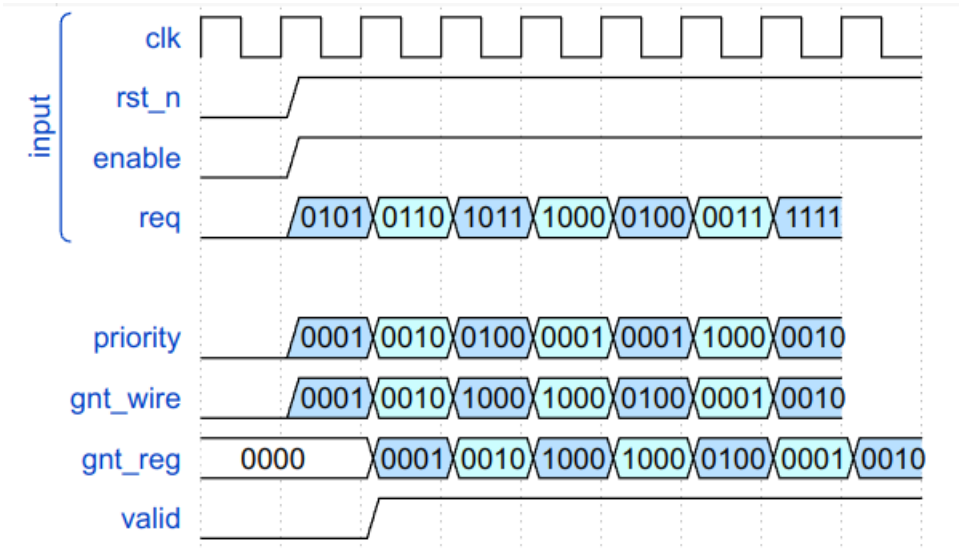
```
fixed_prior_arb_give #(
.N(N)
) u_fixed_sel_priority (
.req(req),
.enable(),
```

```
.priority(priority_reg),
.gnt(gnt_wire),
.valid()
);
```

```
reg [N-1:0] gnt_reg;
always @(posedge clk, negedge rst_n) begin
if(~rst_n)
gnt_reg <= 0;
else if(enable)
gnt_reg <= gnt_wire;
end
assign gnt = gnt_reg;
```

```
reg valid_reg;
always @(posedge clk, negedge rst_n) begin
if(~rst_n)
valid_reg <= 1'b0;
else if(enable & |req)
valid_reg <= 1'b1;
else
valid_reg <= 1'b0;
end
assign valid = valid_reg & enable;
```

```
endmodule
```



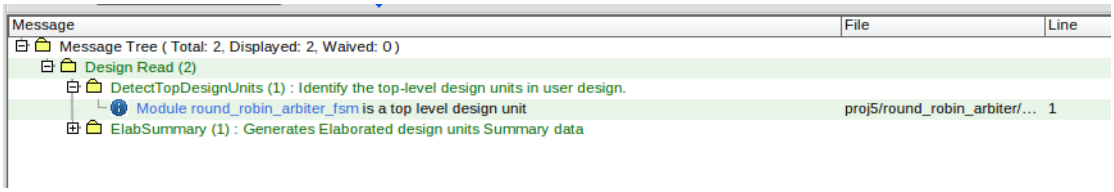

在这里，同样同一颜色表示数据在一个相位上，相互存在依赖关系。这里 gnt_reg 在同一颜色上延迟一个相位，说明它是寄存器在打拍记录数据。valid 信

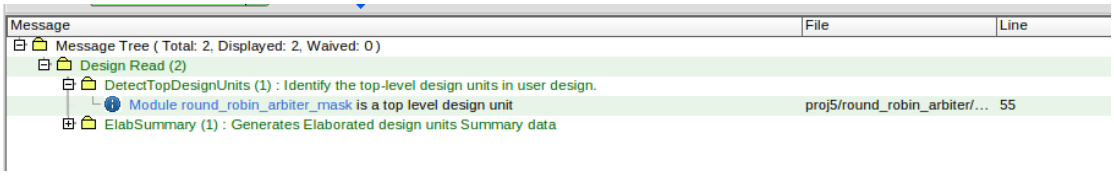
号在 enable 有效的下一个时钟周期有效，所以 gnt_reg 结果要用寄存器延时一个节拍。

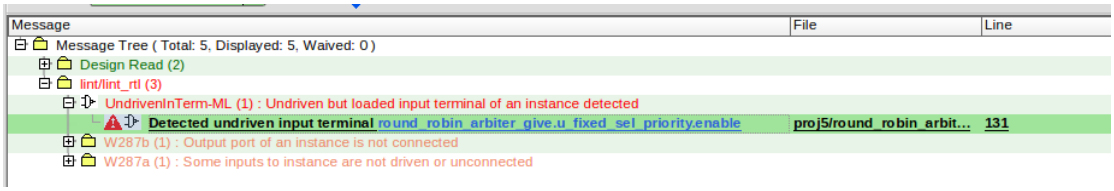
priority 也是寄存器，只是它的值取决于上一个节拍 gnt_wire 组合逻辑的值。每次它的输入数据是 gnt_wire 的循环左移一位的信号。gnt_wire 就是 req 根据 priority 用固定优先级仲裁器仲裁出的结果。根据轮换优先级仲裁器的定义，每次被仲裁的值循环左移下一位的值拥有最高优先级，而 priority 独热码的 1 表示的就是最高优先级的位，所以每次左移一位表示顺移优先级。这样就实现了轮换仲裁的功能。只需要在原有固定优先级仲裁器的基础上加 valid，priority，gnt 的寄存器就可以实现功能。

设计时需要注意时序，特别是 gnt 需要寄存器保持，但是 priority 需要 gnt_wire 的组合逻辑信号，且 priority 本身也要寄存器寄存。

spyglass 的 lint 检查：



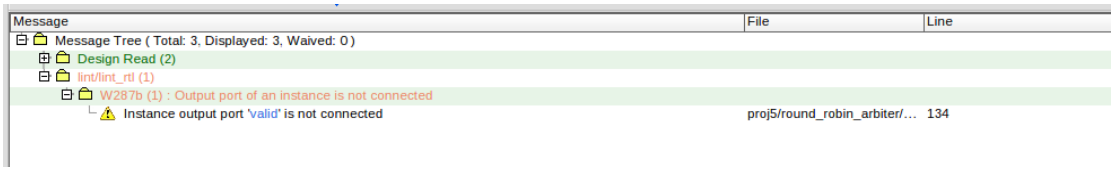




有端口输入悬空需要给与输入，哪怕不需要对应的输出：

```
) u_fixed_sel_priority (
.enable(),
```

改为

```
) u_fixed_sel_priority (
.req(req),
.enable(1'b1),
```




修改后输出端口悬空是可以接受的。

## 五、 仿真结果

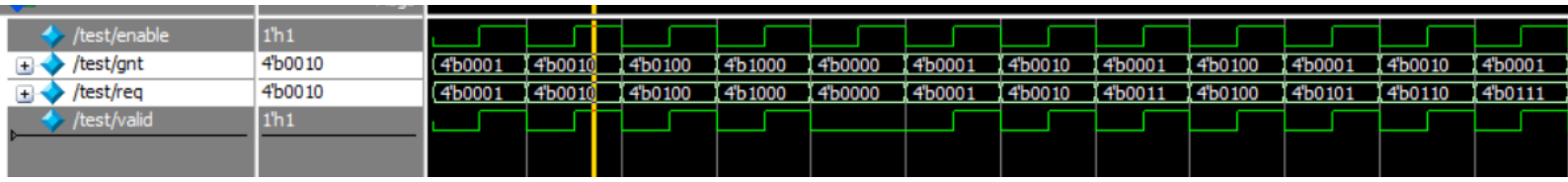
**固定优先级仲裁器：**

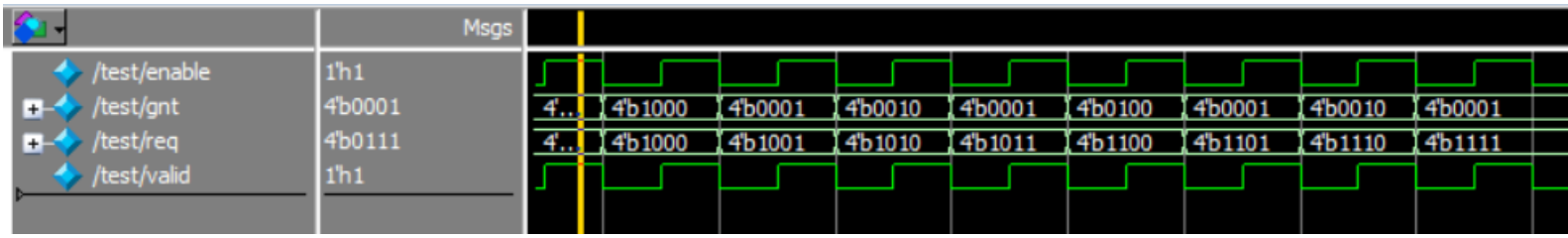
testbench：

```
`timescale 1ps/1ps
module test();
    parameter N = 4;
    reg [N-1:0] req;
    reg enable;
    wire [N-1:0] gnt;
    wire valid;

    fixed_prior_arb #(
        .N(N)
    ) u1 (
        .req(req),
        .enable(enable),
        .gnt(gnt),
        .valid(valid)
    );

    integer i;
    initial begin
        enable = 1'b0;
        // req = 0;
        // #5 enable = 1'b1;
        // #5 enable = 1'b0;
        for(i = 0; i < N; i = i + 1) begin
            req = 1 << i;
            #5 enable = 1'b1;
            #5 enable = 1'b0;
        end
        for(i = 0; i < 2**N; i = i + 1) begin
            req = i;
            #5 enable = 1'b1;
            #5 enable = 1'b0;
        end
    end
```

```
endmodule
```

仿真波形：




波形和预期一致。对于给定的 req，gnt 的反馈符合需求。

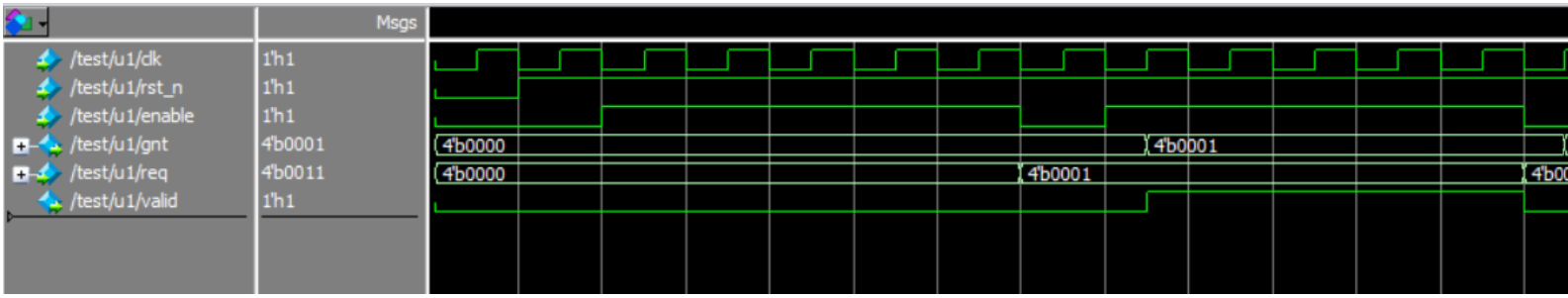
**轮换优先级仲裁器：**

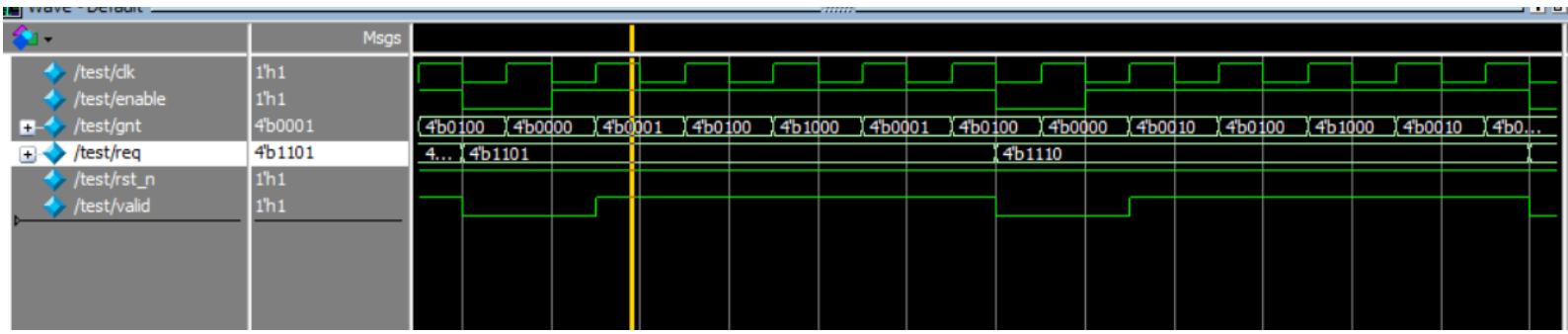
testbench：

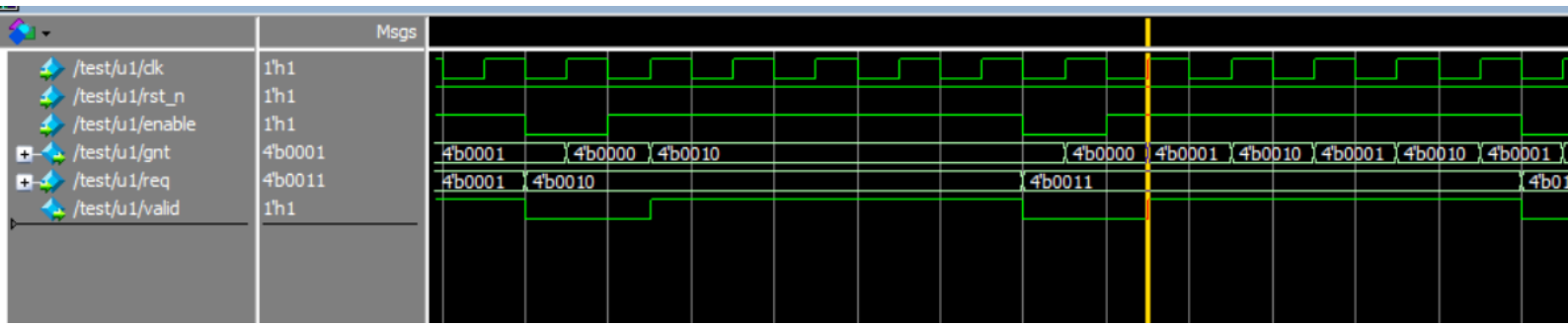
```
`timescale 1ps/1ps
module test();
    reg clk;
    reg rst_n;
    reg [3:0] req;
    reg enable;
    wire [3:0] gnt;
    wire valid;

    // round_robin_arbiter #(
    //     .N(4)
    // ) u1 (
    //     .clk(clk),
    //     .rst_n(rst_n),
    //     .req(req),
    //     .enable(enable),
    //     .gnt(gnt),
    //     .valid(valid)
    // );
    round_robin_arbiter_fsm u1 (
        .clk(clk),
        .rst_n(rst_n),
        .req(req),
        .enable(enable),
```

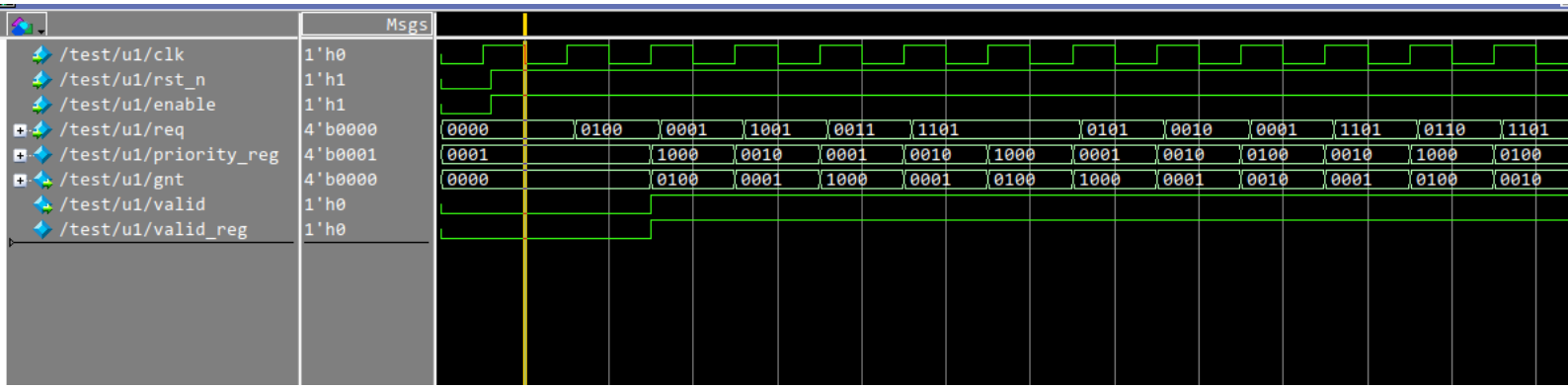
```
        .gnt(gnt),
        .valid(valid)
    );

    initial begin
        clk = 1'b0;
        forever #5 clk = ~clk;
    end

    integer i;
    initial begin
        rst_n = 1'b0;
        enable = 1'b0;
        req = 4'b0000;
        #10 rst_n = 1'b1;
        for(i = 0; i < 16; i = i + 1) begin
            req = i;
            #10 enable = 1'b1;
            #50 enable = 1'b0;
        end
    end

endmodule
```

仿真波形：






可以看到不管是怎样的输入结果都能正确显示。gnt 在时钟边沿跳变，valid 也保持和 enable 和时钟同步。

使用随机化输入的方式在使能时就改变 req 的值，可以看到结果也是正确的。

## 六、 实验总结

通过比较固定优先级仲裁器和轮换优先级仲裁器，理解了在不同总线通信场景下不同的仲裁需求会产生不同的仲裁规则，根据相应需求设计功能不同的电路。固定优先级仲裁器可以使用补码相与或菊式链级联的方法实现，轮换优先级仲裁器除了使用有限状态机外还可以使用掩码寄存器的方式实现。当然还有别的设计思路，都很值得学习。

在设计轮换优先级仲裁器的掩码方式实现中我遇到了困难，花了非常多的时间尝试写 verilog 代码，但是仿真总是发现时序不正确，掩码寄存器的反应效果和预期不符，且信号过多导致电路信号连接很复杂，使我一时不能判断如何设计，哪些信号要用组合逻辑实现，哪些信号要用时序逻辑实现。最后我尝试先画波形

图，把需要解决的信号通过穷举的方式列举出来，信号要一个一个分析逻辑，结果思路清晰很多，也知道了掩码 mask 要用组合逻辑实现，配合 mask_reg 寄存器使用。本次设计我收获颇多，我的电路设计分析能力有很大的提升，解决的之前很久我都没有解决的问题。