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 8：同步/异步 FIFO 的设计

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 28 日

一、 实验目的

通过设计和仿真了解同步 FIFO 和异步 FIFO 的细节和作用。

二、 实验任务

1. 设计同步 FIFO，了解地址指针的作用，并通过配置指针实现 full，empty 和对应 almost 信号的实现。
2. 了解 CDC 跨时钟域设计的方法，了解建立时间和保持时间的概念，学会使用寄存器链延迟减少亚稳态的破坏。
3. 根据 CDC 设计思想在同步 FIFO 的基础上设计异步 FIFO，对地址指针的管理优化，使异步 FIFO 在最大程度上减少 CDC 的消极影响。

三、 实验原理

FIFO 是一种先进先出数据缓存器，它与普通存储器的区别是没有外部读写地址线，使用起来非常简单，缺点是只能顺序读写，而不能随机读写。

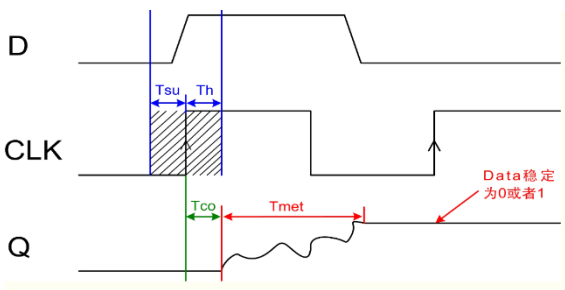
FIFO 的经典使用场景：保序：保持数据顺序；缓冲：应用于瞬时写入和读出速度不匹配的场景。（对于长期速度不匹配的场景是不可以的）

FIFO 设计的重要原则：不要向满 FIFO 中写入数据（写溢出）；不要从空 FIFO 中读出数据（读溢出）。

FIFO 本质上由同步伪双端口 RAM 和输入输出控制器组成，控制器负责产生 RAM 的读写使能信号和地址指针，保证数据读写的正确。

和同步 FIFO 不同的是，异步 FIFO 读写采用不同的时钟，它主要有两个作用，一个是实现数据在不同时钟域进行传递，另一个作用就是实现不同数据宽度的数据接口互联。

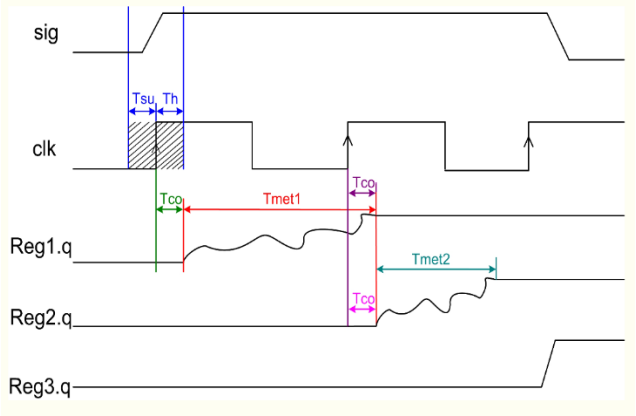
建立时间和保持时间：



如果数据传输中不满足触发器的 Tsu 和 Th 不满足，或者复位过程中复位信

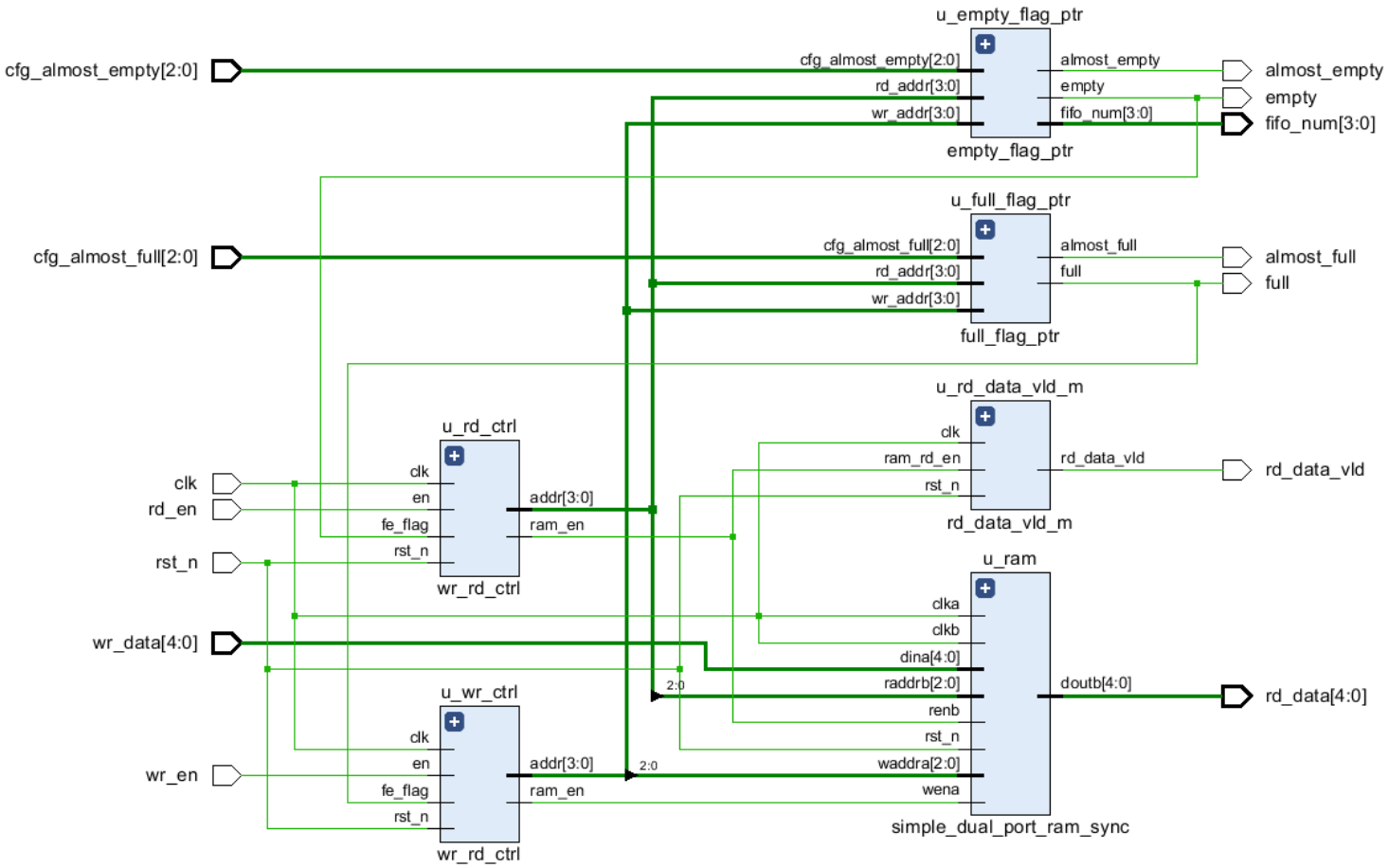
号的释放相对于有效时钟沿的恢复时间（recovery time）不满足，就可能产生亚稳态，此时触发器输出端 Q 在有效时钟沿之后比较长的一段时间处于不确定的状态，在这段时间里 Q 端在 0 和 1 之间处于振荡状态，而不是等于数据输入端 D 的值。

解决亚稳态的方法：使用寄存器链延迟打拍，同步到需要的时钟域。



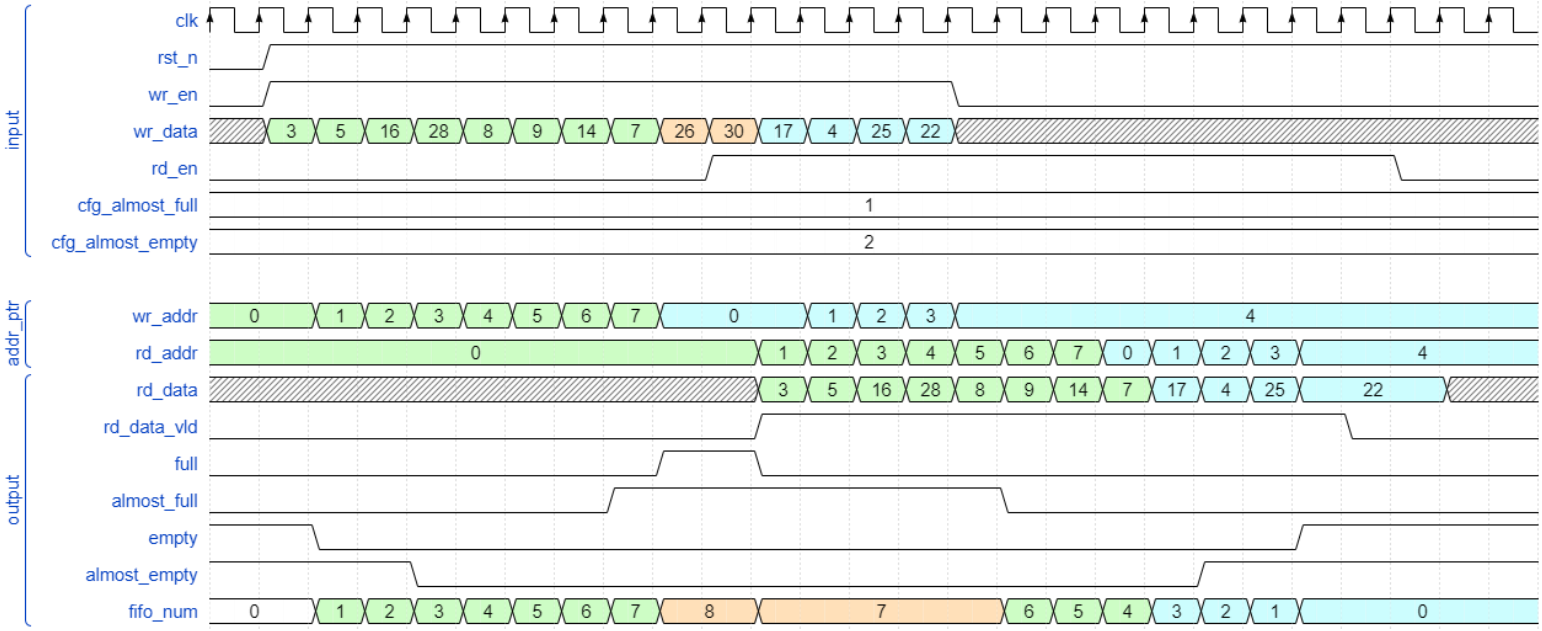
四、 实验过程

同步 FIFO：模块框架：



模块包括同步伪双端口 RAM，读写控制模块，full 和 empty 标志位产生模块，以及读数据有效信号。

波形解释：



本次示例配置 almost_full 为 1，almost_empty 为 2，RAM 深度为 8。对于写数据，在 wr_en 写使能信号有效后数据一直被读入，此时 empty 信号在写入第一个数据之后置 0，almost_empty 在第 3 个数据写入之后置 0。在 fifo_num 为 7 时 almost_full 置 1 提示将满，再写入一个数据后为 8，full 信号置 1。此时再写入一个数据 26，在后面也不会输出，因为不会被 FIFO 存入到 RAM 中。在输入第 9 个数据的同时开启读使能信号，在写入第 10 个数据时才能读到有效的数据，且第 10 个数据不会被写入 RAM，因为 full 信号美置 0。之后同时进行读写操作，FIFO 的数量保持在 7，读写指针时钟差 1。

不久写使能关闭，只有读使能有效。可以看到读出的数据不包括写入的第 9 个和第 10 个数据。当 fifo_num 的数量变为 6 后 almost_full 置 0，继续读。随后 fifo_num 为 2 后 almost_empty 置 1，fifo_num 为 0 后 empty 置 1。empty 有效后计时读使能也不会读出数据。

verilog 代码具体实现：

同步伪双端口 RAM 实现见实验 6。

读写控制端口：

```
module wr_rd_ctrl #(
    parameter DEEPWID = 3
)(
    input clk,
    input rst_n,
    input en,
    input fe_flag,  // full or empty flag
    output ram_en,
    output [DEEPWID:0] addr   // need extra bit for judging full/empty
);
    assign ram_en = en & ~fe_flag;

    reg [DEEPWID:0] addr_reg;
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            addr_reg <= 0;
        else if(ram_en)
            addr_reg <= addr_reg + 1;
    end
    assign addr = addr_reg;

endmodule
```

根据 full/empty 信号指示，在时钟边沿到来时检查外界输入的读写使能信号，如果符合条件则输出 RAM 的读写使能信号实现数据写入或读出 RAM，并且用计数器控制 RAM 地址。读地址和写地址寄存器不同，需要两次例化，是两个相同的电路模块。

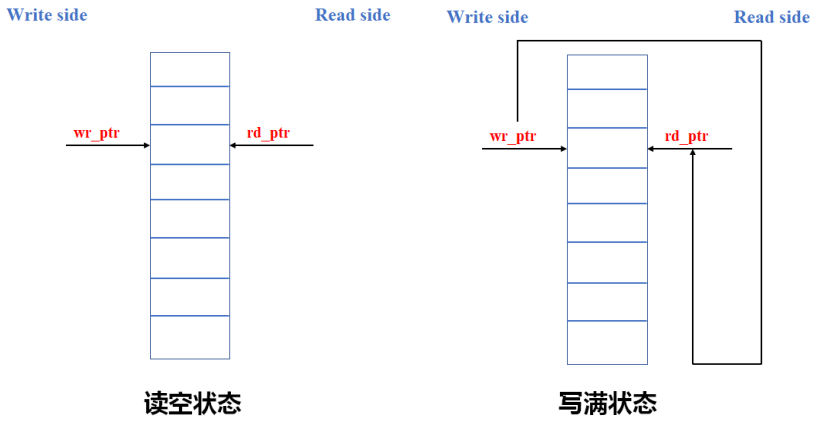
地址寄存器需要比 RAM 深度多一位，用于辅助判断空满信号。

full/empty 标志产生模块：

```
module full_flag_ptr #(
    parameter DEEPWID = 3
)(
    input [DEEPWID:0] wr_addr,
    input [DEEPWID:0] rd_addr,
    input [DEEPWID-1:0] cfg_almost_full,
    output full,
    output almost_full
```

```
);
    // assign full = (wr_addr[DEEPWID] ^ rd_addr[DEEPWID]) &&
(wr_addr[DEEPWID-1:0] == rd_addr[DEEPWID-1:0]);

    wire diff_round;
    assign diff_round = (wr_addr[DEEPWID] ^ rd_addr[DEEPWID]);

    wire [DEEPWID:0] full_gap;
    assign full_gap = diff_round ?
        (rd_addr[DEEPWID-1:0] - wr_addr[DEEPWID-1:0]) :
        (rd_addr[DEEPWID-1:0] + 2**DEEPWID - wr_addr[DEEPWID-1:0]);
    assign almost_full = (full_gap <= {1'b0, cfg_almost_full});
    assign full = (full_gap == 0);

endmodule

module empty_flag_ptr #(
    parameter DEEPWID = 3
)(
    input [DEEPWID:0] wr_addr,
    input [DEEPWID:0] rd_addr,
    input [DEEPWID-1:0] cfg_almost_empty,
    output empty,
    output almost_empty,
    output [DEEPWID:0] fifo_num
);

    // assign empty = ~(wr_addr[DEEPWID] ^ rd_addr[DEEPWID]) &&
(wr_addr[DEEPWID-1:0] == rd_addr[DEEPWID-1:0]);

    wire diff_round;
    assign diff_round = (wr_addr[DEEPWID] ^ rd_addr[DEEPWID]);

    wire [DEEPWID:0] empty_gap;
    assign empty_gap = diff_round ?
        (wr_addr[DEEPWID-1:0] + 2**DEEPWID - rd_addr[DEEPWID-1:0]) :
        (wr_addr[DEEPWID-1:0] - rd_addr[DEEPWID-1:0]);
    assign almost_empty = (empty_gap <= {1'b0, cfg_almost_empty});
    assign empty = (empty_gap == 0);

    assign fifo_num = empty_gap;

endmodule
```

空满信号的产生：



读写指针寄存器的最高位表示它们超前彼此的情况。如果最高位相同，表示二者轮回次数相同，若此时指针除最高位以外的所有位相同则表示空。

若最高位相反，则说明彼此不在同一个轮回中。由于读指针不可能超前写指针，所以写指针肯定超前读指针一个轮回。如果此时除最高位以外的其他位都相等，那么写指针超前读指针一个轮回，说明 RAM 被写满。

almost 信号的产生和此类似。根据最高位判断写指针是否超过读指针一个轮回，基于此判断读写指针之间的间隔是直接作差（不超前）还是用模减去读指针加写指针（超前轮回）。算出间隔就可以添加比较器和输入的 config 的数进行比较，判断 almost_full/empty。间隔分为 full 的间隔和 empty 的间隔，为 0 时 full 或 empty 信号有效。

并且 empty 的 gap 间隔判断可以直接输出 RAM 已经存入数据的数量 fifo_num。

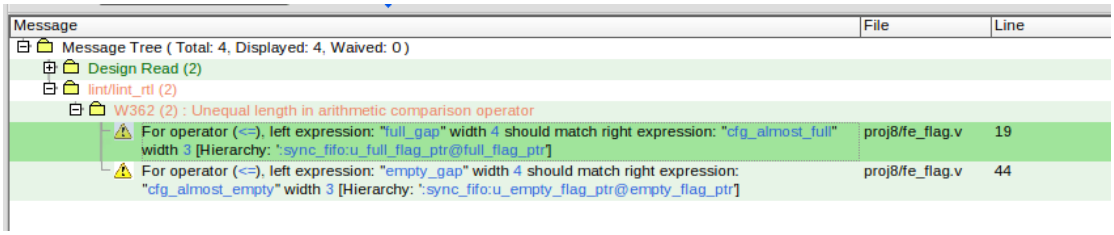
读有效信号：

```
module rd_data_vld_m (
    input clk,
    input rst_n,
    input ram_rd_en,
    output rd_data_vld
);
    reg rd_data_vld_reg;
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            rd_data_vld_reg <= 1'b0;
        else if(ram_rd_en)
```

```
            rd_data_vld_reg <= 1'b1;
        else
            rd_data_vld_reg <= 1'b0;
    end
    assign rd_data_vld = rd_data_vld_reg;

endmodule
```

将读写控制产生的读 RAM 使能信号进行寄存器延迟打拍，即可满足时序要求。

spyglass 报告：

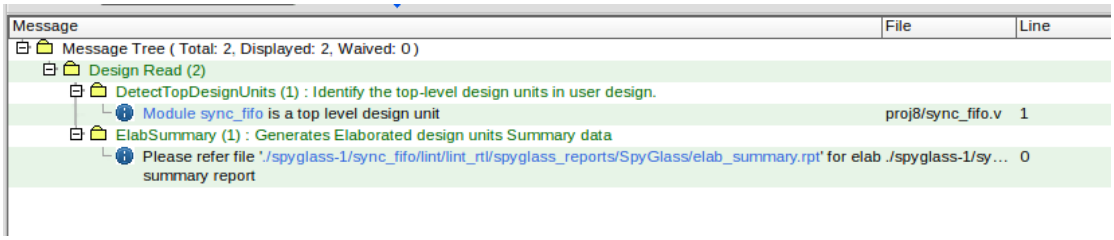



```
    assign almost_full = (full_gap <= cfg_almost_full);
```

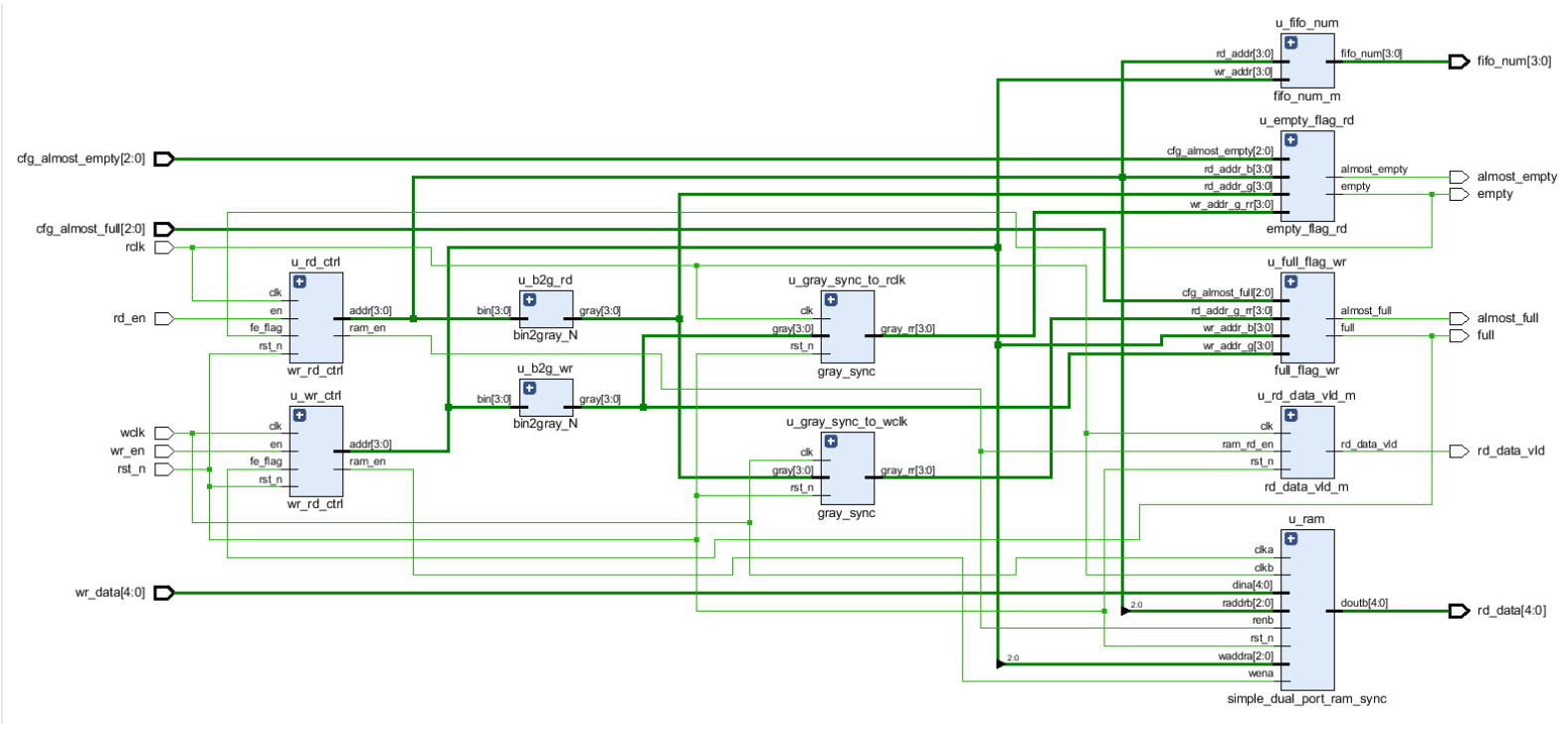
警告在判断 almost 信号时位宽不一致，手动加 0 使位宽一致：

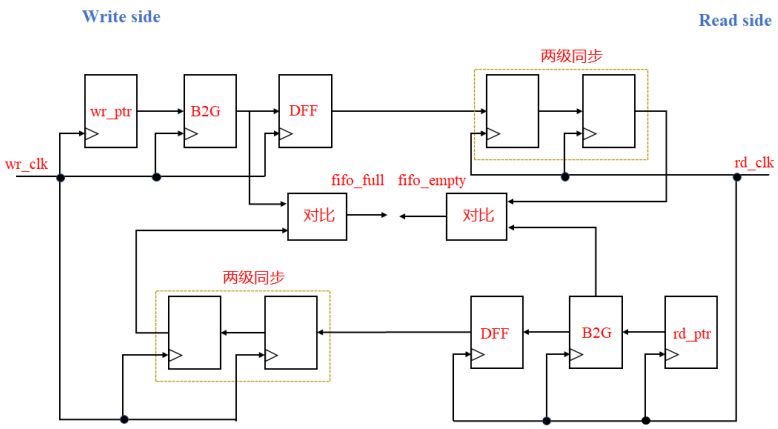
```
    assign almost_full = (full_gap <= {1'b0, cfg_almost_full});
```

修改后解决问题。



异步 FIFO：模块框架：





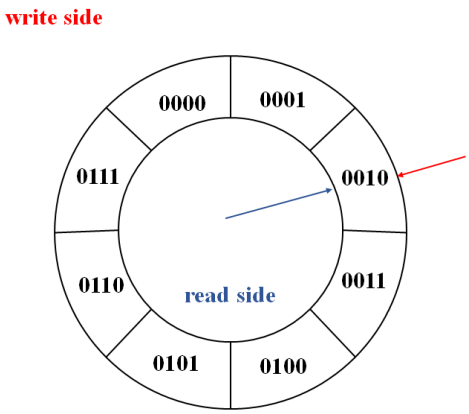
异步 FIFO 的读写时钟不同，需要 CDC 设计。在同步 FIFO 的基础上变化主要体现在 full 和 empty 标志信号的产生。对于跨时钟域的设计，需要用寄存器链延迟采样信号，尽可能减少亚稳态造成的破坏性。使用格雷码同步地址信号的优势在于每次变化只变一位，所以产生亚稳态的位限制到一位，不至于多位变化都是亚稳态造成更大的破坏。亚稳态只会在时钟边沿采样时被采样信号在跳变时发生。

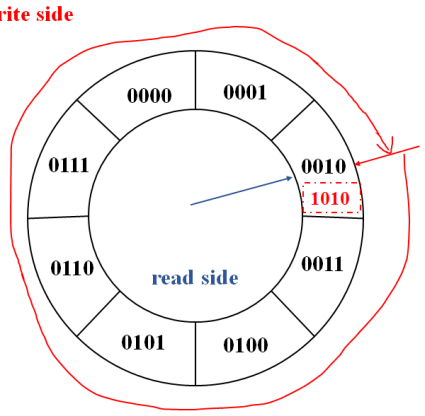
但是后续对于地址的判断还是要用减法器，所以在二进制码转换成格雷码后还要转换成二进制码，格雷码只在跨时钟域传递时起作用。

举例：读地址要被送到写满信号标志位产生器中，而满信号是用来控制 RAM

写使能是否有效，所以读时钟域的地址信号要同步到写时钟域用来判断 full 信号是否产生。反之同理，写时钟域的地址信号要同步到读时钟域判断 empty 信号是否产生，用来判断 RAM 的读使能是否产生。

虽然 almost 信号需要把格雷码转换到二进制码判断，但仅仅 empty 和 full 信号是可以直接根据格雷码判断。





十进制数	4位自然二进制码	4位典型格雷码
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

当两个指针在一个轮回里时最高位相同，其他位都相同，当然格雷码也相同，判断此时为空。当两个指针在不同轮回时（写指针超前读指针一个轮回）且二进制的其他位相同时，格雷码的高两位相反，低两位相同，判断此时为满。

One data pushed to fifo:							
RP	**WP**						
5 more data pushed to fifo							
RP						**WP**	
3 data popped from fifo							
			RP			**WP**	
3 more data pushed to fifo							
	WP		**RP**				
2 more data pushed to fifo: **fifo full**							
			WP, RP				
6 data popped from fifo							
	RP		**WP**				

延迟的合理性：假设在读时钟域观测到的写指针滞后，那么在读时钟域判断的是 empty 信号。当读指针赶上滞后的写指针时产生 empty 信号，但是写指针超前，此时 RAM 没有空，但不准读，不会造成数据错误，只是有数据不能及时读出来而已，不会读出错误数据。

假设在写时钟域观测到的读指针滞后，那么在写时钟域判断的是 full 信号。当写指针赶上滞后的读指针时产生 full 信号，但是读指针超前，此时 RAM 没有满，但不准写，不会造成数据溢出的错误，只是有空间浪费而已。

verilog 代码：

二进制和格雷码的转换：

```
module bin2gray_N
#(
    parameter N = 4
)(
    input [N-1:0] bin,
    output [N-1:0] gray
);
    assign gray = bin ^ (bin >> 1);
endmodule


module gray2bin_N
#(
    parameter N = 4
)(
    input [N-1:0] gray,
    output [N-1:0] bin
);
    assign bin = {gray[N-1], bin[N-1:1] ^ gray[N-2:0]};
```

```
endmodule
```

详细见实验 2。

空满标志位的产生：

```
module full_flag_wr #(
    parameter DEEPWID = 3
)(
    input [DEEPWID:0] wr_addr_g,
    input [DEEPWID:0] rd_addr_g_rr,
    input [DEEPWID:0] wr_addr_b,
    input [DEEPWID-1:0] cfg_almost_full,
    output full,
    output almost_full
);
    assign full = (wr_addr_g[DEEPWID:(DEEPWID-1)] ==
~rd_addr_g_rr[DEEPWID:(DEEPWID-1)]) &&
        (wr_addr_g[DEEPWID-2:0] == rd_addr_g_rr[DEEPWID-2:0]);

    wire [DEEPWID:0] rd_addr_b_rr;
    gray2bin_N #(
        .N(DEEPWID + 1)
    ) u_g2b_full_flag_rd (
        .gray(rd_addr_g_rr),
        .bin(rd_addr_b_rr)
    );

    full_flag_ptr_sync_fifo #(
        .DEEPWID(DEEPWID)
    ) u_full_flag_ptr_sync_fifo (
        .wr_addr(wr_addr_b),
        .rd_addr(rd_addr_b_rr),
        .cfg_almost_full(cfg_almost_full),
        .full(),
        .almost_full(almost_full)
    );

endmodule


module empty_flag_rd #(
    parameter DEEPWID = 3
)(
```

```
    input [DEEPWID:0] wr_addr_g_rr,
    input [DEEPWID:0] rd_addr_g,
    input [DEEPWID:0] rd_addr_b,
    input [DEEPWID-1:0] cfg_almost_empty,
    output empty,
    output almost_empty
);
    assign empty = (wr_addr_g_rr == rd_addr_g);

    wire [DEEPWID:0] wr_addr_b_rr;
    gray2bin_N #(
        .N(DEEPWID + 1)
    ) u_g2b_full_flag_wr (
        .gray(wr_addr_g_rr),
        .bin(wr_addr_b_rr)
    );

    empty_flag_ptr_sync_fifo #(
        .DEEPWID(DEEPWID)
    ) u_empty_flag_ptr_sync_fifo (
        .wr_addr(wr_addr_b_rr),
        .rd_addr(rd_addr_b),
        .cfg_almost_empty(cfg_almost_empty),
        .empty(),
        .almost_empty(almost_empty),
        .fifo_num() // not proper clock domain
    );

endmodule
```

借用同步 FIFO 的部分代码复用，专指二进制码计算的部分。

fifo_num 的产生：

```
module fifo_num_m #(
    parameter DEEPWID = 3
)(
    input [DEEPWID:0] wr_addr,  // different clock domain
    input [DEEPWID:0] rd_addr,  // different clock domain
    output [DEEPWID:0] fifo_num
);
    empty_flag_ptr_sync_fifo #(
        .DEEPWID(DEEPWID)
    ) u_fifo_num_sync_fifo (
```

```
        .wr_addr(wr_addr),
        .rd_addr(rd_addr),
        .cfg_almost_empty({DEEPWID{1'b0}}),
        .empty(),
        .almost_empty(),
        .fifo_num(fifo_num)
    );

endmodule
```

同样借用同步 fifo 的空标志位的模块代码复用。

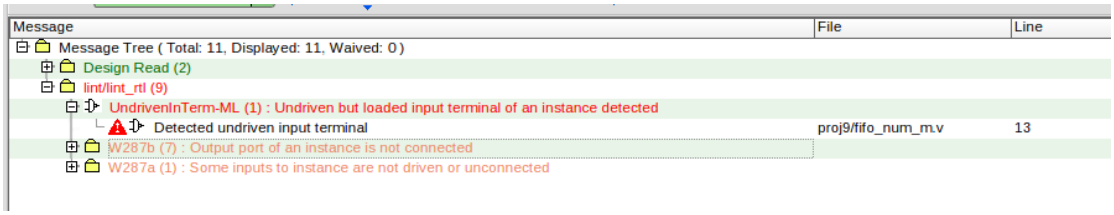
格雷码跨时钟域同步模块：

```
module gray_sync #(
    parameter N = 4
)(
    input clk,
    input rst_n,
    input [N-1:0] gray,
    output [N-1:0] gray_r,
    output [N-1:0] gray_rr
);
    reg [N-1:0] gray_r_reg;
    reg [N-1:0] gray_rr_reg;
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n) begin
            gray_r_reg <= 0;
            gray_rr_reg <= 0;
        end
        else begin
            gray_r_reg <= gray;
            gray_rr_reg <= gray_r_reg;
        end
    end
    assign gray_r = gray_r_reg;
    assign gray_rr = gray_rr_reg;

endmodule
```

用寄存器链延时到指定时钟。

其他模块和同步 FIFO 的内容一致。

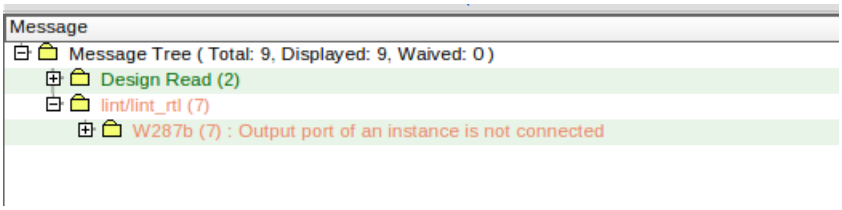
spyglass 报告：



在例化复用同步 FIFO 模块的时候有的输入端口由于用不上就悬空了，经检查把输入的悬空全部连接上信号，哪怕时没有意义的值（复用同步 fifo 模块，为了得到 fifo_num）：

```
        .cfg_almost_empty({DEEPWID{1'b0}}),
```

问题解决，剩下有的模块的输出端口没有连接悬空，确认是不需要的。



五、 仿真结果

同步 FIFO：

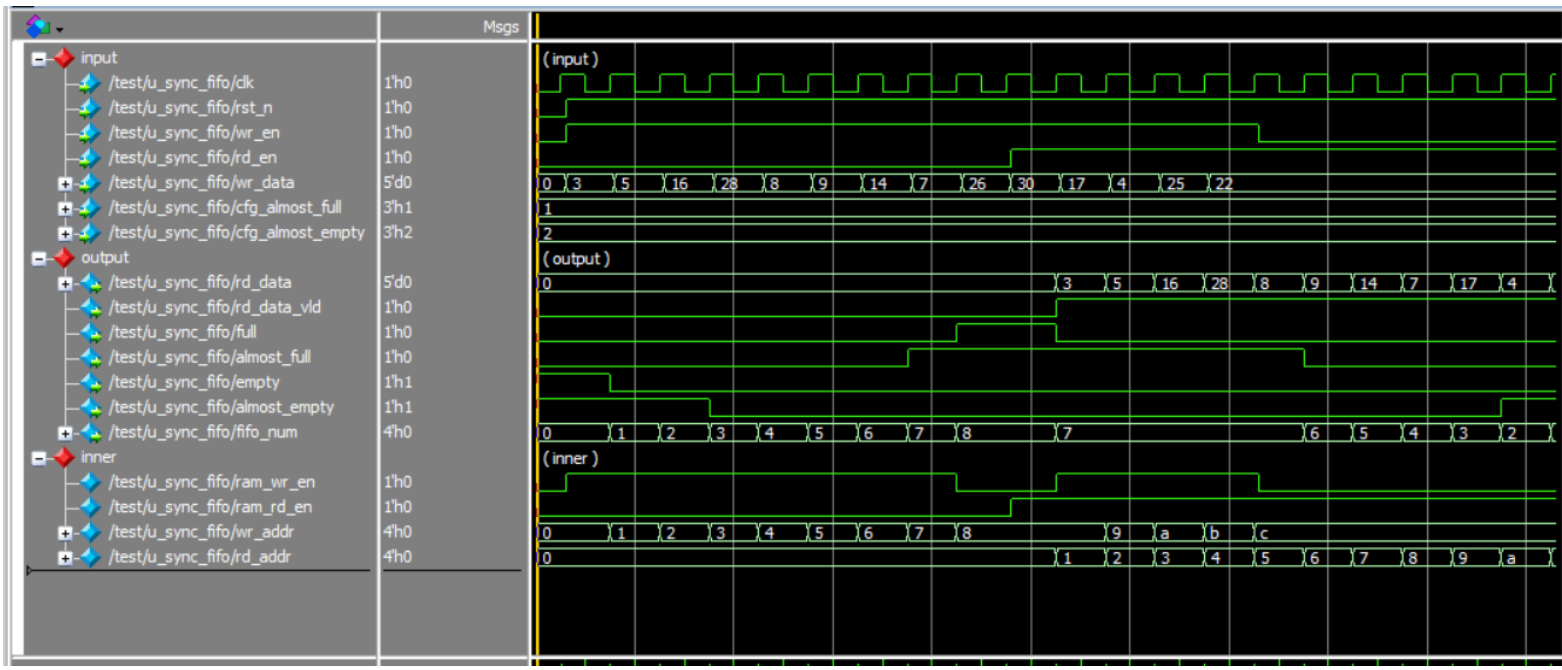
verilog 代码：

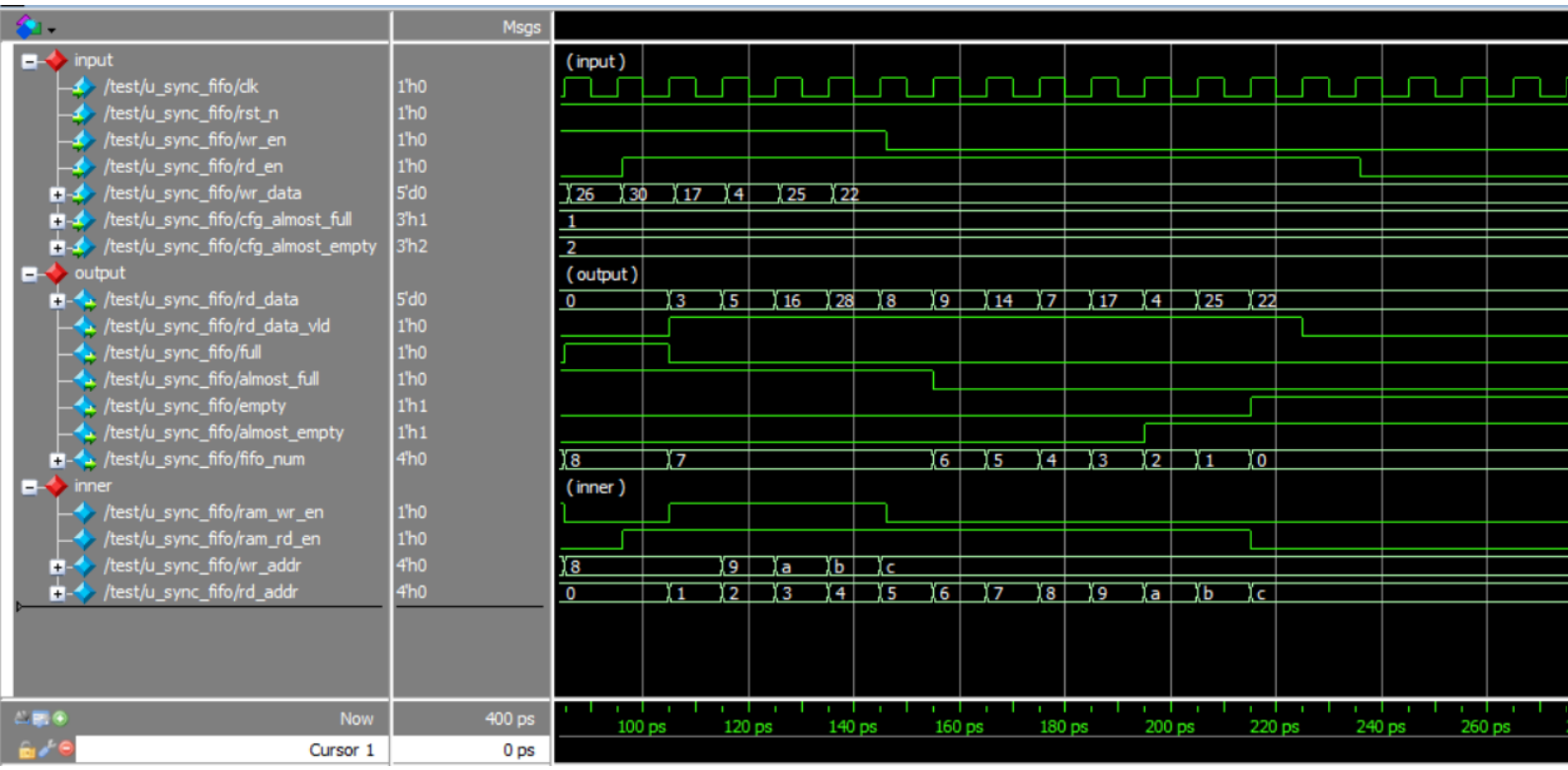
```
`timescale 1ps/1ps
module test ();
    parameter DEEPWID = 3;
    parameter BITWID = 5;

    reg clk,rst_n;
    reg wr_en;
    reg rd_en;
    reg [BITWID-1:0] wr_data;
    wire [BITWID-1:0] rd_data;
    reg [DEEPWID-1:0] cfg_almost_full;
    reg [DEEPWID-1:0] cfg_almost_empty;

    wire rd_data_vld;
    wire full;
    wire empty;
    wire almost_full;
    wire almost_empty;
    wire [DEEPWID:0] fifo_num;

    sync_fifo #(
```

```
        .DEEPWID(DEEPWID),
        .BITWID(BITWID)
    ) u_sync_fifo (
        .clk(clk),
        .rst_n(rst_n),
        .wr_en(wr_en),
        .rd_en(rd_en),
        .wr_data(wr_data),
        .rd_data(rd_data),
        .cfg_almost_full(cfg_almost_full),
        .cfg_almost_empty(cfg_almost_empty),

        .rd_data_vld(rd_data_vld),
        .full(full),
        .empty(empty),
        .almost_full(almost_full),
        .almost_empty(almost_empty),
        .fifo_num(fifo_num)
    );

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

    initial begin
        rst_n = 1'b0;
        wr_en = 1'b0;
        rd_en = 1'b0;
        wr_data = 0;
        cfg_almost_full = 1;
        cfg_almost_empty = 2;
        #6 rst_n = 1'b1;
        wr_en = 1'b1;
        wr_data = 3;
        #10 wr_data = 5;
        #10 wr_data = 16;
        #10 wr_data = 28;
        #10 wr_data = 8;
        #10 wr_data = 9;
        #10 wr_data = 14;
        #10 wr_data = 7;
        #10 wr_data = 26;
        #10 wr_data = 30;
```

```
        rd_en = 1'b1;
        #10 wr_data = 17;
        #10 wr_data = 4;
        #10 wr_data = 25;
        #10 wr_data = 22;
        #10;
        wr_en = 1'b0;
        #90 rd_en = 1'b0;
    end

endmodule
```

仿真波形：





输入和输出和上述 wavedrom 的设计一致。输入特定的序列后保持输入使 FIFO 溢出，并按顺序读取。full 和 empty 信号根据 fifo_num 的更新合理产生。

异步 FIFO：

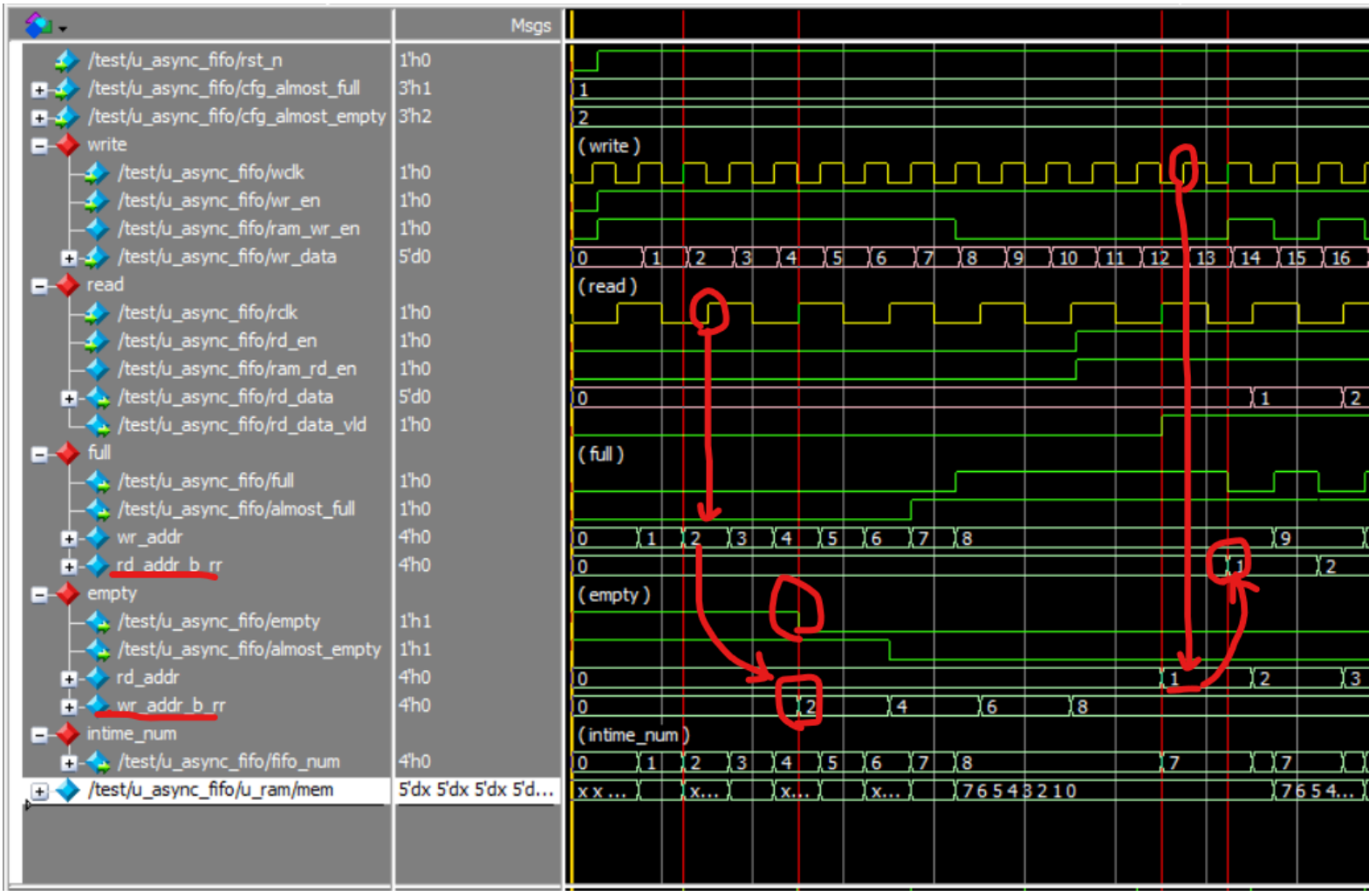
testbench：

```
`timescale 1ps/1fs
module test ();
    parameter DEEPWID = 3;
    parameter BITWID = 5;

    reg wclk,rclk;
    reg rst_n;
    reg wr_en;
    reg rd_en;
    reg [BITWID-1:0] wr_data;
    wire [BITWID-1:0] rd_data;
    reg [DEEPWID-1:0] cfg_almost_full;
    reg [DEEPWID-1:0] cfg_almost_empty;

    wire rd_data_vld;
    wire full;
    wire empty;
```

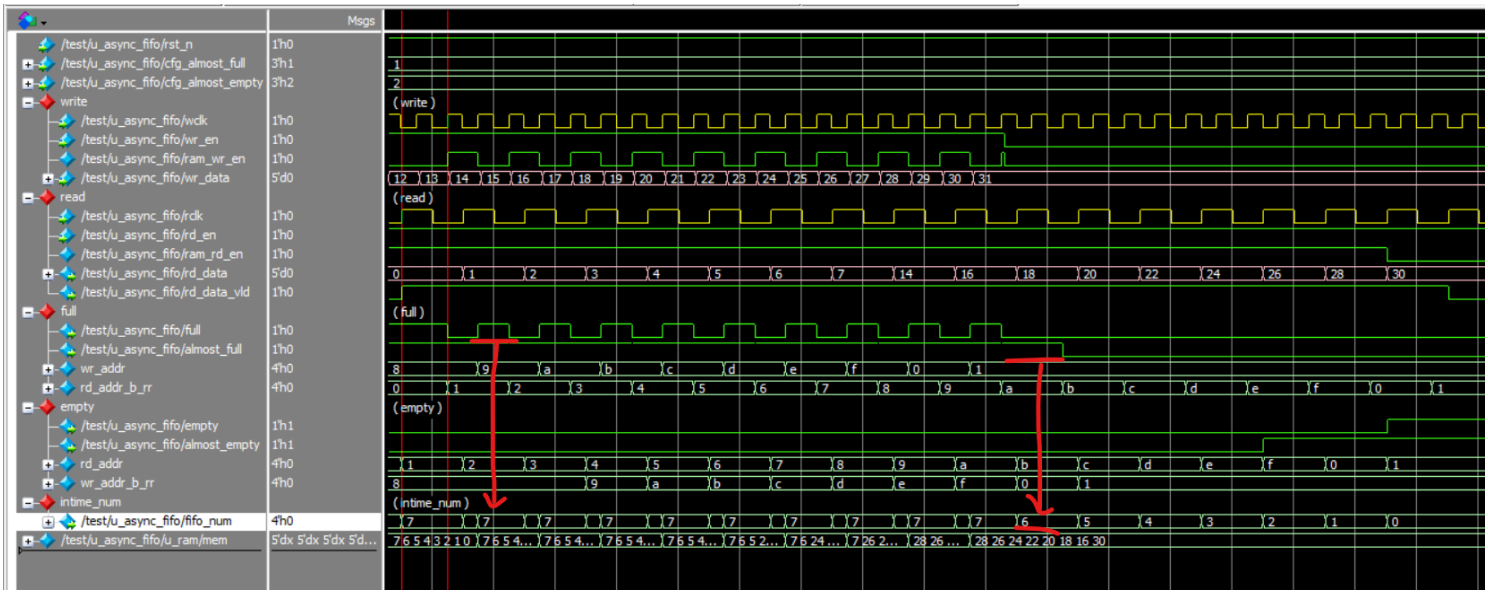
```
    wire almost_full;
    wire almost_empty;
    wire [DEEPWID:0] fifo_num;

    async_fifo #(
        .DEEPWID(DEEPWID),
        .BITWID(BITWID)
    ) u_async_fifo (
        .wclk(wclk),
        .rclk(rclk),
        .rst_n(rst_n),
        .wr_en(wr_en),
        .rd_en(rd_en),
        .wr_data(wr_data),
        .rd_data(rd_data),
        .cfg_almost_full(cfg_almost_full),
        .cfg_almost_empty(cfg_almost_empty),

        .rd_data_vld(rd_data_vld),
        .full(full),
        .empty(empty),
        .almost_full(almost_full),
        .almost_empty(almost_empty),
        .fifo_num(fifo_num)
    );

    initial begin
        wclk = 1'b0;
        forever #5 wclk = ~wclk;
    end
    initial begin
        rclk = 1'b0;
        #0.3;
        forever #10 rclk = ~rclk;
    end

    integer i,j;
    initial begin
        rst_n = 1'b0;
        wr_en = 1'b0;
        wr_data = 0;
        cfg_almost_full = 1;
        cfg_almost_empty = 2;
        #6 rst_n = 1'b1;
```

```
        wr_en = 1'b1;
        for(i = 0; i < 32; i = i + 1) begin
            wr_data = i;
            #10;
        end
        wr_en = 1'b0;
    end
    initial begin
        rd_en = 1'b0;
        #0.3;
        #11;
        #100 rd_en = 1'b1;
        #1000 rd_en = 1'b0;
    end

endmodule
```

仿真波形：



可以看到在采样延迟 2 个时钟节拍后才更新地址，不过这样的延迟不会造成数据错误，所以可以接收。empty 信号在读时钟域更新，full 信号在写时钟域更新。fifo_num 由组合逻辑产生，所以更新及时，不需要时钟也不需要时钟域同步。empty 信号滞后才消去。



可以看到 full 信号提前产生，滞后撤去，保证不会写溢出。

六、 实验总结

本次实验实现了同步 FIFO 和异步 FIFO，了解读写指针的产生方法，特别是 full 和 empty 信号的产生最为关键，可以使用拓展最高位的方式方便判断两个指针超越轮回的情况，而且方便判断之间的差值，用于用户配置 cfg_almost 的 full/empty 情况。学习设计异步 FIFO 特别是 CDC 的设计，使用寄存器链打拍延迟，且用格雷码编码减少亚稳态的发生，减轻亚稳态对后续数据通路的破坏性。仿真的 FIFO 结果满足预期，可以用到其他的模块中使用，实现模块化电路设计。