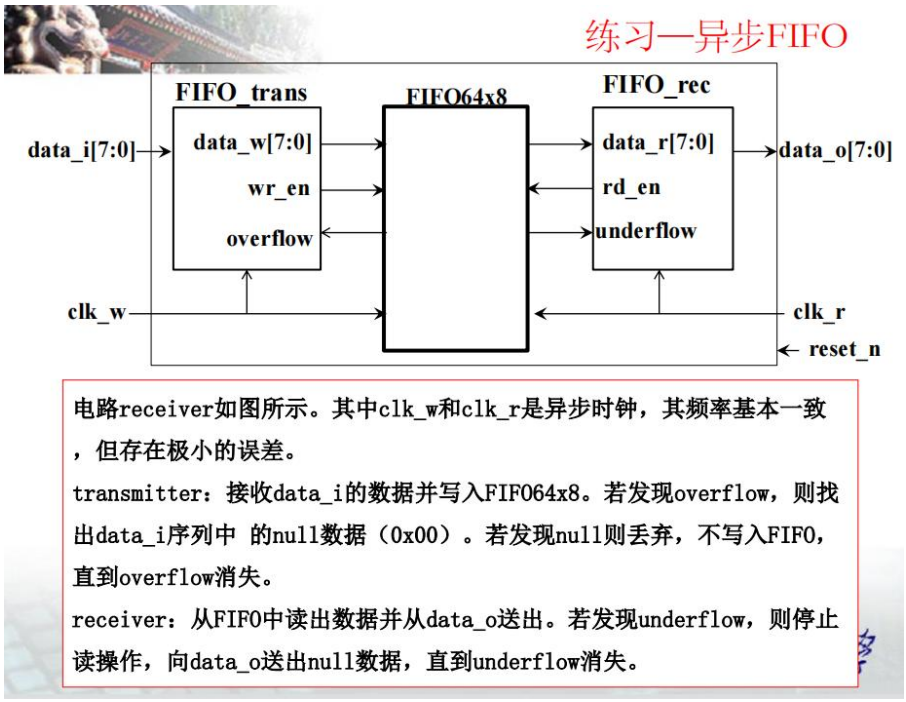
一. 异步 FIFO



1. 模块功能代码：

```
//异步FIFO，未考虑时钟同步，用NULL处理
module FIFO_asyn_64x8_NULL #(
    //param
        parameter DEPTH = 6,                    //数据深度（2^6）
        parameter WIDTH = 8,                    //每个数据的宽度（8位）
        parameter MAX_COUNT = (1 << DEPTH), //存储器中的数据个数（2^6）
        parameter NULL = 8'b0
    )(
    //input
        input wr_clk,                       //写时钟
        input rd_clk,                       //读时钟
        input wr_rstn,                      //写复位（低电平有效）
        input rd_rstn,                      //读复位（低电平有效）
        input wr_en,                        //写使能
        input rd_en,                        //读使能
        input [WIDTH-1:0] wr_data,          //写入的数据

    //output
        output reg wr_overflow,             //FIFO将满，75%（影响写入，在写时钟端判断）
        output reg rd_underflow,            //FIFO将空，25%（影响读出，在读时钟端判断）
        output reg [WIDTH-1:0] rd_data,   //读出的数据
    //output_for_test
        output reg [DEPTH-1:0] wr_ptr,    //写指针，指向下一个要写的位置
        output reg [DEPTH-1:0] rd_ptr,    //读指针，指向下一个要读的位置
        output reg [DEPTH-1:0] count      //计数FIFO中的数据个数
    );

    /*
    //定义读写指针
    reg [DEPTH-1:0] wr_ptr; //写指针，指向下一个要写的位置
    reg [DEPTH-1:0] rd_ptr; //读指针，指向下一个要读的位置
    */

    //定义FIFO存储器
    reg [WIDTH-1:0] fifo_mem[MAX_COUNT-1:0]; //共有64个8bits数据的存储器
```

```
    //写
    always @(posedge wr_clk or negedge wr_rstn)  //写控制端异步复位
    begin
        if(!wr_rstn) wr_ptr <= 0;
        else if(wr_en && !wr_overflow) //写使能有效且FIFO未满时可写
        begin
            fifo_mem[wr_ptr] <= wr_data;
            wr_ptr <= wr_ptr + 1'b1;
        end
        else if(wr_en && wr_overflow) //当overflow时找出输入数据中的NULL进行丢弃，直到overflow消失
        begin
            if(wr_data == NULL)
            begin
                wr_ptr <= wr_ptr;
            end
            else
            begin
                fifo_mem[wr_ptr] <= wr_data;
                wr_ptr <= wr_ptr + 1'b1;
            end
        end
        else wr_ptr <= wr_ptr;
    end

    //读
    always @(posedge rd_clk or negedge rd_rstn)  //读控制端异步复位
    begin
        if(!rd_rstn) rd_ptr <= 0;
        else if(rd_en && !rd_underflow) //读使能有效且FIFO未空时可读
        begin
            rd_data <= fifo_mem[rd_ptr];
            rd_ptr  <= rd_ptr + 1'b1;
        end
        else if(rd_en && rd_underflow) //当underflow时停止读数据，直接送出NULL，直到underflow消失
        begin
            rd_data <= NULL;
            rd_ptr  <= rd_ptr;
        end
        else rd_ptr <= rd_ptr;
    end
```

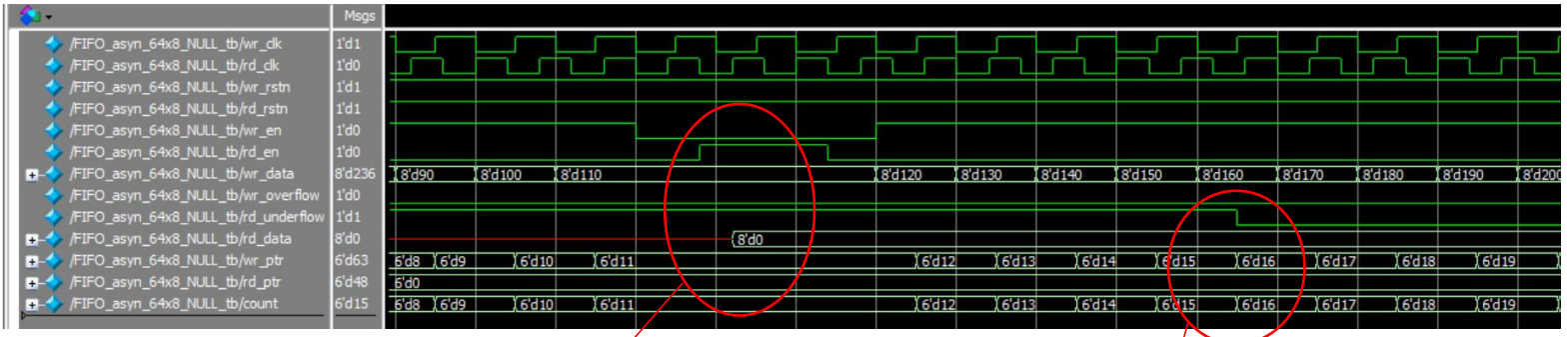
```
    //数据个数计数
    always @(*)
    begin
        if(!wr_rstn | !rd_rstn) count = 0;
        else
            count = wr_ptr - rd_ptr;
    end

    //更新标志位overflow和underflow
    always @(*)
    begin
        wr_overflow  = count[(DEPTH-1)-:2] == 2'b11; //count高两位是11时，说明数据个数>=48（75%）
        rd_underflow = count[(DEPTH-1)-:2] == 2'b00; //count高两位是11时，说明数据个数<16（25%）
    end

endmodule
```

2. 模块测试代码：先从 0 写 FIFO，直到写指针递增为 63，再将 FIFO 读空到 underflow 状态；设读时钟的频率高于写时钟；将读/写指针以及计数器输出，方便调试。

```v
//FIFO_asyn_64x8_NULL testbench
`timescale 1ns/1ns
module FIFO_asyn_64x8_NULL_tb;
    reg wr_clk;
    reg rd_clk;
    reg wr_rstn;
    reg rd_rstn;
    reg wr_en;
    reg rd_en;
    reg [7:0] wr_data;

    wire wr_overflow;
    wire rd_underflow;
    wire [7:0] rd_data;

    wire [5:0] wr_ptr;
    wire [5:0] rd_ptr;
    wire [5:0] count;


    FIFO_asyn_64x8_NULL FIFO_asyn_64x8_NULL_test(
    //input
        .wr_clk(wr_clk),                    //写时钟
        .rd_clk(rd_clk),                    //读时钟
        .wr_rstn(wr_rstn),                  //写复位（低电平有效）
        .rd_rstn(rd_rstn),                  //读复位（低电平有效）
        .wr_en(wr_en),                      //写使能
        .rd_en(rd_en),                      //读使能
        .wr_data(wr_data),                  //写入的数据

    //output
        .wr_overflow(wr_overflow),          //FIFO将满，75%（影响写入，在写时钟端判断）
        .rd_underflow(rd_underflow),        //FIFO将空，25%（影响读出，在读时钟端判断）
        .rd_data(rd_data),                  //读出的数据
    //output_for_test
        .wr_ptr(wr_ptr),  //写指针，指向下一个要写的位置
        .rd_ptr(rd_ptr),  //读指针，指向下一个要读的位置
        .count(count)     //计数FIFO中的数据个数
    );
```

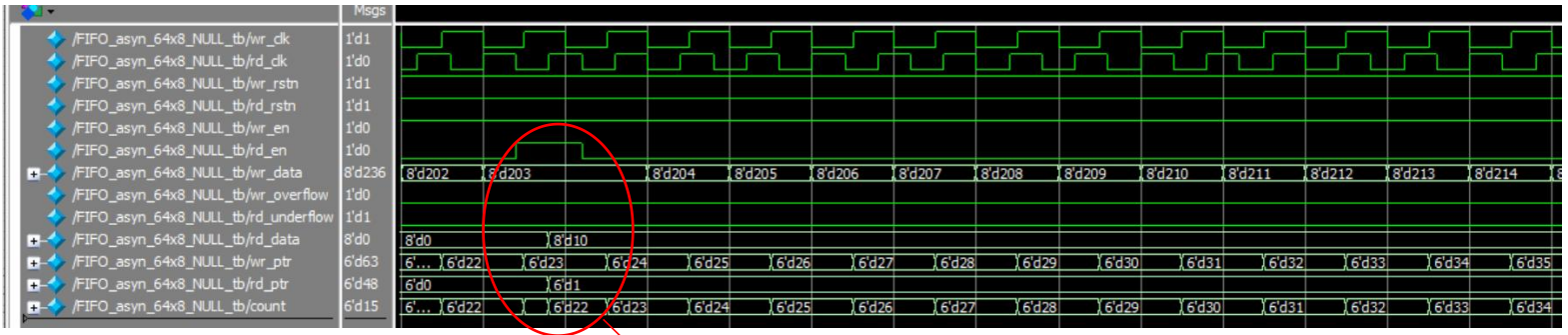
```
    always #25 wr_clk = ~wr_clk;
    always #20 rd_clk = ~rd_clk;

    initial begin
        wr_en = 0; rd_en = 0; wr_rstn = 1; rd_rstn = 1; wr_data = 8'b0; wr_clk = 0; rd_clk = 0;
    #10 wr_rstn = 0; rd_rstn = 0;
    #20 wr_rstn = 1; rd_rstn = 1;

    @ (negedge wr_clk);
    wr_data = 10;
    wr_en = 1;

    @ (negedge wr_clk);
    wr_data = 20;
    @ (negedge wr_clk);
    wr_data = 30;
    @ (negedge wr_clk);
    wr_data = 40;
    @ (negedge wr_clk);
    wr_data = 50;
    @ (negedge wr_clk);
    wr_data = 60;
    @ (negedge wr_clk);
    wr_data = 70;
    @ (negedge wr_clk);
    wr_data = 80;
    @ (negedge wr_clk);
    wr_data = 90;
    @ (negedge wr_clk);
    wr_data = 100;
    @ (negedge wr_clk);
    wr_data = 110;

    @ (negedge wr_clk);
    wr_en = 0;

    @ (negedge rd_clk);
    rd_en = 1;
    #80
    @ (negedge rd_clk);
    rd_en = 0;

    @(negedge wr_clk);
    wr_data = 120;
    wr_en = 1;
    @(negedge wr_clk);
    wr_data = 130;
    @(negedge wr_clk);
    wr_data = 140;
    @(negedge wr_clk);
    wr_data = 150;
    @(negedge wr_clk);
    wr_data = 160;
    @(negedge wr_clk);
    wr_data = 170;
    @(negedge wr_clk);
    wr_data = 180;
    @(negedge wr_clk);
    wr_data = 190;
    @(negedge wr_clk);
    wr_data = 200;
    @(negedge wr_clk);
    wr_data = 201;
    @(negedge wr_clk);
    wr_data = 202;
    @(negedge wr_clk);
    wr_data = 203;

    @ (negedge rd_clk);
    rd_en = 1;
    #40
    @ (negedge rd_clk);
    rd_en = 0;

    @(negedge wr_clk);
    wr_data = 204;
    @(negedge wr_clk);
    wr_data = 205;
    @(negedge wr_clk);
    wr_data = 206;
    @(negedge wr_clk);
    wr_data = 207;
    @(negedge wr_clk);
    wr_data = 208;
    @(negedge wr_clk);
    wr_data = 209;
    @(negedge wr_clk);
    wr_data = 210;
    @(negedge wr_clk);
    wr_data = 211;
    @(negedge wr_clk);
    wr_data = 212;
    @(negedge wr_clk);
    wr_data = 213;
    @(negedge wr_clk);
    wr_data = 214;
    @(negedge wr_clk);
    wr_data = 215;

    @(negedge wr_clk);
    wr_data = 0;
    @(negedge wr_clk);
    wr_data = 0;

    @(negedge wr_clk);
    wr_data = 216;
    @(negedge wr_clk);
    wr_data = 217;
    @(negedge wr_clk);
    wr_data = 218;
    @(negedge wr_clk);
    wr_data = 219;
    @(negedge wr_clk);
    wr_data = 220;
    @(negedge wr_clk);
    wr_data = 221;
    @(negedge wr_clk);
    wr_data = 222;
    @(negedge wr_clk);
    wr_data = 223;
    @(negedge wr_clk);
    wr_data = 224;
    @(negedge wr_clk);
    wr_data = 225;
    @(negedge wr_clk);
    wr_data = 226;

    @(negedge wr_clk);
    wr_data = 0;
    @(negedge wr_clk);
    wr_data = 0;

    @(negedge wr_clk);
    wr_data = 227;
    @(negedge wr_clk);
    wr_data = 228;
    @(negedge wr_clk);
    wr_data = 229;
    @(negedge wr_clk);
    wr_data = 230;
    @(negedge wr_clk);
    wr_data = 231;
    @(negedge wr_clk);
    wr_data = 232;
    @(negedge wr_clk);
    wr_data = 233;
    @(negedge wr_clk);
    wr_data = 234;
    @(negedge wr_clk);
    wr_data = 235;
    @(negedge wr_clk);
    wr_data = 0;

    @(negedge rd_clk);
    rd_en = 1;
    #600;

    @(negedge rd_clk);
    rd_en = 0;

    @ (negedge wr_clk);
    wr_en = 0;

    @(negedge wr_clk);
    wr_data = 236;

    @ (negedge rd_clk);
    rd_en = 1;
    #1920
    @ (negedge rd_clk);
    rd_en = 0;

    #50 $finish;
    end

endmodule
```

3. ModelSim 仿真结果及分析：

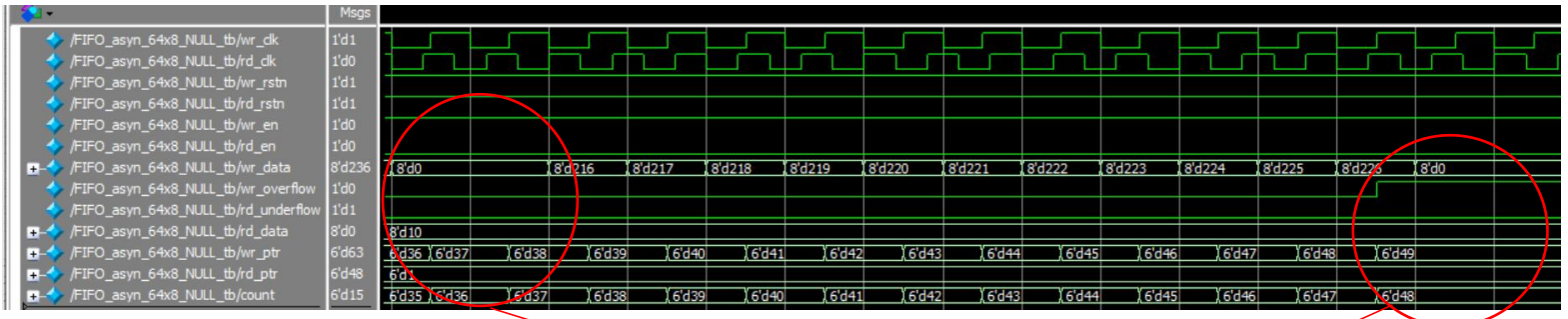


在 FIFO 中数据量为 11 时，underflow 为 1，此时读数据读到的不是 FIFO 中的数据，而是 NULL（0），此时读指针也不+1

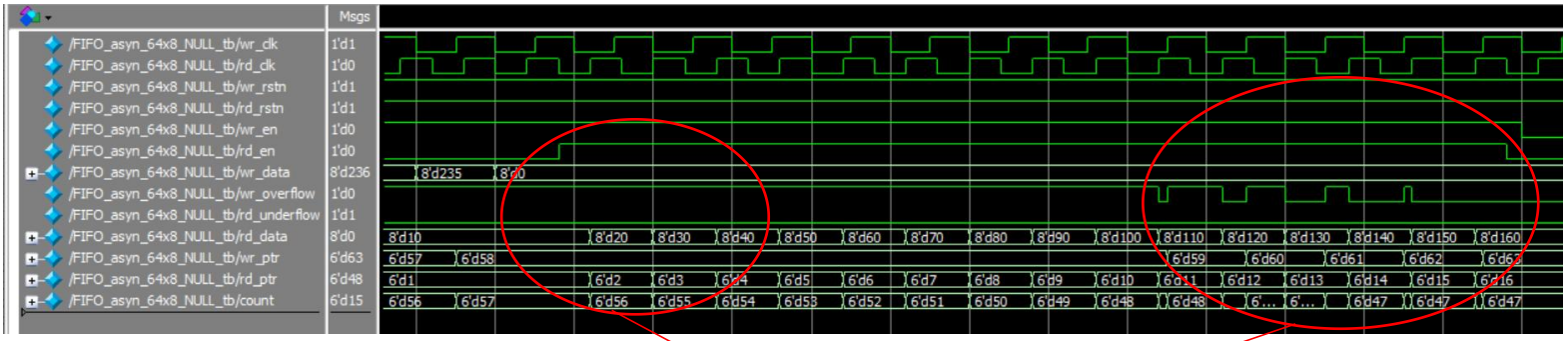
在 FIFO 中数据量为 16 时，underflow 变成 0



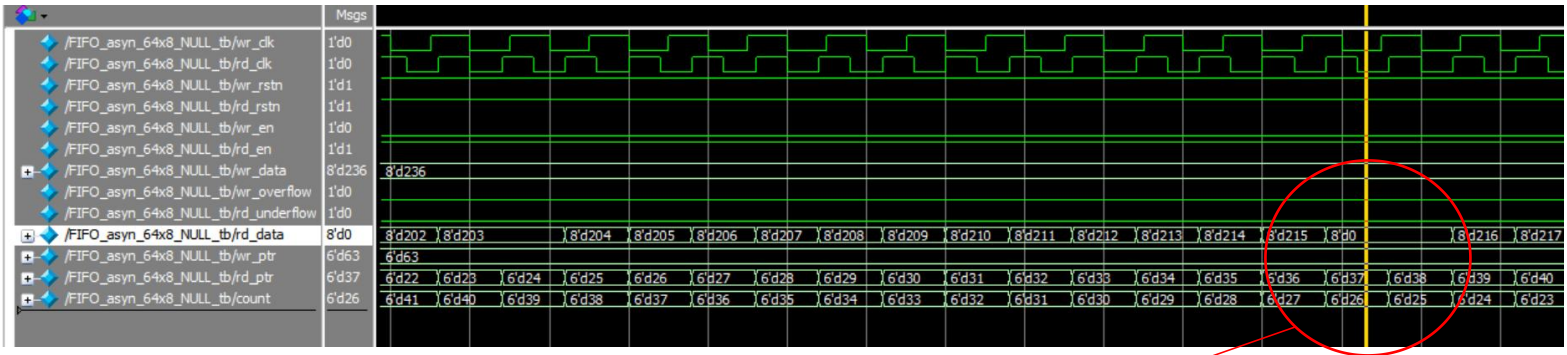
在 FIFO 中数据量为 22 时，underflow 为 0，此时 rd_data 读取了 FIFO 中的第一个数据 10，读指针也+1。之后继续往 FIFO 中写入数据



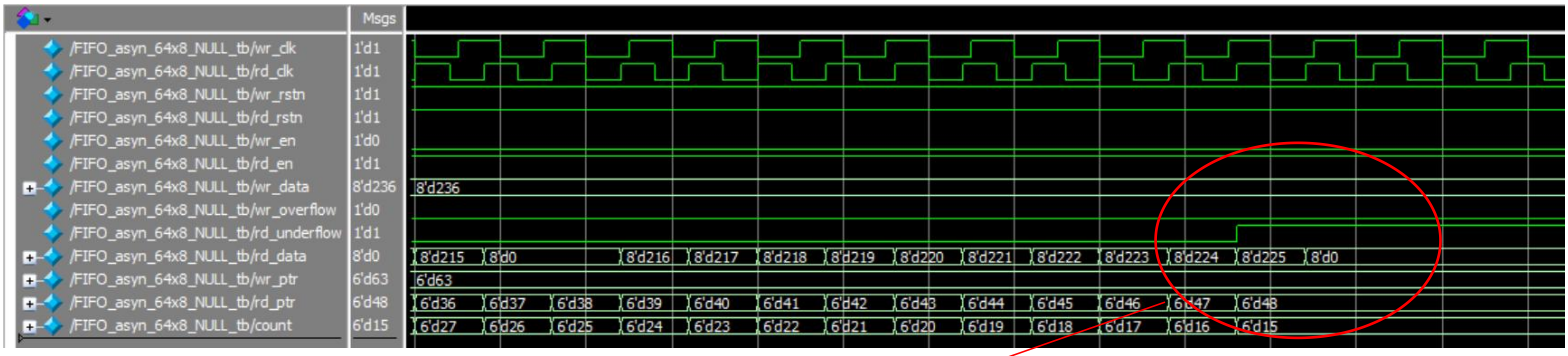
在 FIFO 中数据量为 48 时，overflow 为 1，此时 wr_data 中的 NULL 信号不能被写入 FIFO，写指针不变。对比 overflow 为 0 时，wr_data 中的 NULL 可以写入 FIFO，写指针+1



在 overflow 状态下开始连续读取数据，不写入（因为 wr_data 为 NULL），FIFO 中的数据量递减（count），最先存入 FIFO 的数据依次读到 rd_data。当数据量递减到 47 时，overflow 为 0，wr_data 中的 NULL 可以写入 FIFO，由于时钟不同步会产生毛刺，稳定后读写同时进行，数据量为 47 不变。



不断读出数据，不写入（写指针不变，读指针递增，计数器递减）。可以看出，读到了在 overflow 为 0 时写入 FIFO 的 NULL 数据。



当 FIFO 中的数据量为 15 时，underflow 为 1，读出的数据 rd_data 不再是 FIFO 中的数据，而是变成 NULL，读指针不变，FIFO 中的数据量也不减少。

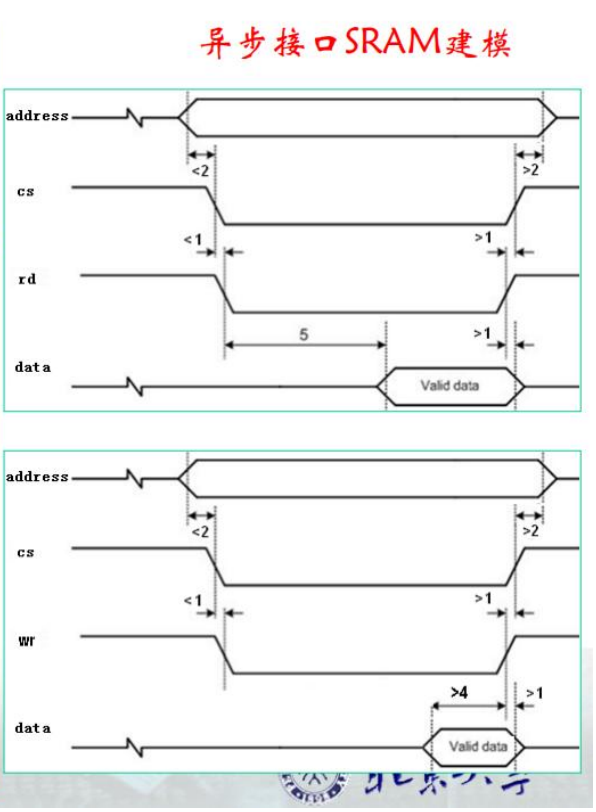
4. 存在的问题：

标志位的产生会有不稳定状态出现，是由于时钟不同步造成的。改进时可以利用两级同步延迟的方法，进行跨时钟域同步。

二．异步接口 SRAM 建模

试描述一个1Kx8的异步SRAM，并对其端口信号进行如右图所示的时序检查，确定信号符合时序要求。若不满足时序要求，则将输出数据置为X值。同时给出测试程序验证时序检查的正确性。

```
module SRAMSP_1Kx8 #(
   parameter width = 8,
             addr  = 10,
             depth = 1 << addr
   ) (
   inout wire [width-1:0] data    ,
   input wire [addr-1 :0] address ,
   input wire             cs      ,
                          rd      ,
                          wr
   );

   reg [width-1:0] mem [depth-1:0]; // 1Kx8

   assign data = (!cs && !rd) ? mem[address]
                              : 'bz;
   always @( posedge wr)
         if( !cs )
            mem[address] = data;
endmodule
```



1. 功能模块代码：对多个信号之间分别进行时序分析，利用 notifier。

```
1  //异步接口SRAM建模，时序检查
2  `timescale 1ns/1ns
3  module SRAMSP_1Kx8 #(
4      parameter width = 8,
5      parameter addr = 10,
6      parameter depth = (1 << addr)
7  )(
8      inout wire [width-1:0] data,
9      input wire [addr-1:0] address,
10     input wire cs,
11     input wire rd,
12     input wire wr,
13
14     output reg re_c2a
15     //output reg re_w2c,
16     //output reg re_r2c,
17     //output reg re_w2d,
18     //output reg sk_a2c,
19     //output reg sk_c2w,
20     //output reg sk_c2r,
21     //output reg wid_data,
22     //output reg sk_w2d
23
24 );
25     reg [width-1:0] mem[depth-1:0];
26     assign data = (!cs && !rd)? mem[address] : 'bz;
27     reg [width-1:0] data_reg;
28
29     always @(negedge wr)
30         if(!cs) mem[address] <= data;
```

```
    specify
        $setup( posedge cs, negedge address, 2, se_c2a);
        //$recovery(posedge wr, posedge cs, 1, re_w2c);
        //$recovery(posedge rd, posedge cs, 1, re_r2c);
        //$recovery(posedge wr, negedge data, 1, re_w2d);
        //$skew(posedge address, negedge cs, 2, sk_a2c);
        //$skew(negedge cs, negedge wr, 1, sk_c2w);
        //$skew(negedge cs, negedge rd, 1, sk_c2r);
        //$width(posedge data, 4, 0, wid_data);
        //$skew(negedge rd, posedge data, 5, sk_w2d);
    endspecify

    always @(se_c2a)
    data_reg = 8'bx;

    assign data = data_reg;

endmodule
```

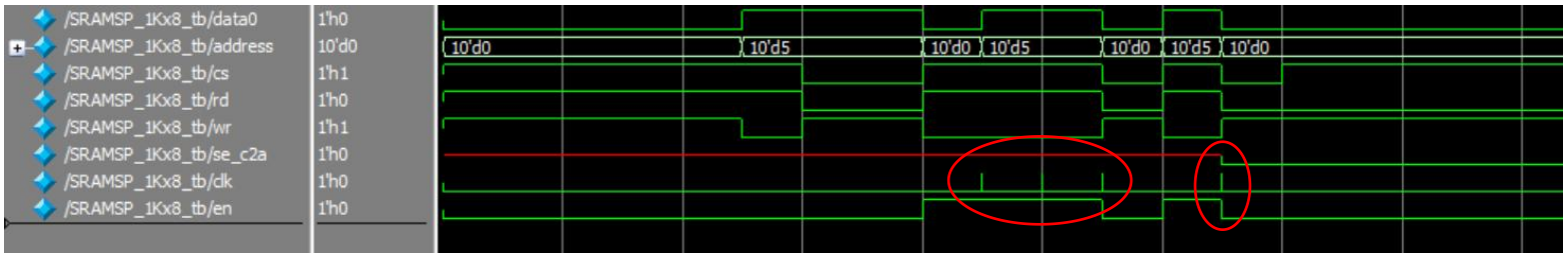
2. 测试模块代码：

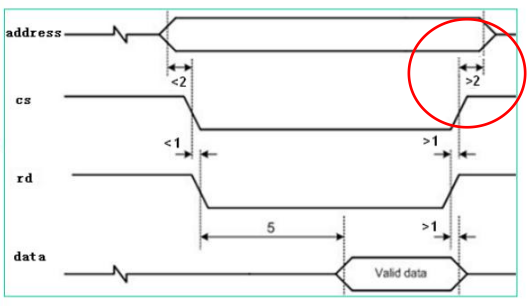
```
`timescale 1ns/1ns
`include"SRAMSP_1Kx8.v"
module SRAMSP_1Kx8_tb;
    wire [7:0] data;
    reg [9:0] address;
    reg cs;
    reg rd;
    reg wr;

    wire re_c2a;
    //wire re_w2c;
    //wire re_w2d;
    //wire sk_a2c;
    //wire sk_c2w;
    //wire wid_data;
    //wire sk_w2d;
    //wire re_r2c;
    //wire sk_c2r;

    wire clk;
    reg en;
    reg data0;

SRAMSP_1Kx8 SRAMSP_1Kx8_test(
    .data(data),
    .address(address),
    .cs(cs),
    .rd(rd),
    .wr(wr),
    .re_c2a(re_c2a)
```

```
    //.re_w2c(re_w2c),
    //.re_w2d(re_w2d),
    //.sk_a2c(sk_a2c),
    //.sk_c2w(sk_c2w),
    //.wid_data(wid_data),
    //.sk_w2d(sk_w2d),
    //.re_r2c(re_r2c),
    //.sk_c2r(sk_c2r)
    );

    assign data = !wr? data0 : 4'bz;

    and #(0.8, 0.2) (clk, en, !clk);

    initial begin
        data0 = 0; en = 0; address = 0; cs = 1; rd = 1; wr = 1;
    #5  data0 = 1; en = 1; address = 5; cs = 1; rd = 1; wr = 0;
    #1  data0 = 1; en = 0; address = 5; cs = 0; rd = 0; wr = 1;
    #3  data0 = 0; en = 1; address = 0; cs = 1; rd = 1; wr = 0;
    #4  data0 = 1; en = 0; address = 0; cs = 1; rd = 1; wr = 0;
    #6  data0 = 0; en = 1; address = 5; cs = 0; rd = 0; wr = 1;
    #2  data0 = 1; en = 0; address = 0; cs = 1; rd = 1; wr = 0;
    #3  data0 = 0; en = 1; address = 0; cs = 1; rd = 0; wr = 1;
    #20 $finish;
    end

endmodule
```

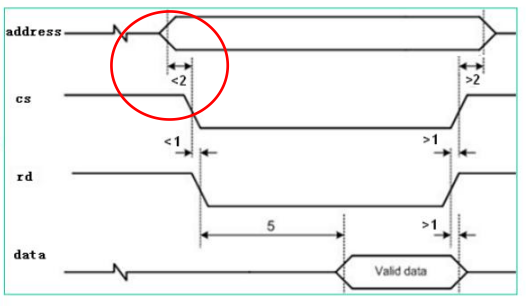
3. ModelSim 仿真结果及分析：

对 setup、skew、width 分别选取一例进行分析。

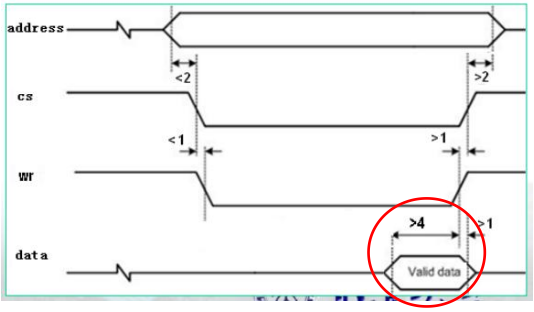




时差小于等于 2 时，notifier（re_c2a）进行翻转，大于 2 时不翻转。



时差大于 2 时，notifier（sk_a2c）进行翻转，小于 2 时不翻转。



data1 的宽度等于 2 或 3 时，均小于 4，notifier（wid_data）进行翻转；data1 宽度大于 4 时不翻转。