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 6　RAM 的设计和读写控制

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 10 日

一、 实验目的

了解 RAM 的基本原理，学会使用 verilog 代码模拟 RAM 的逻辑，了解 RAM 的读写时序，区分单端口 RAM，伪双端口 RAM，真双端口 RAM，并据此设计 RAM 的外围电路完成任务。

二、 实验任务

1. 了解单端口 RAM、伪双端口 RAM 和真双端口 RAM 的区别。理解 RAM 读写时序上的差异。
2. 设计伪双端口 RAM。完成基本的功能仿真。
3. 在自己设计的伪双端口 RAM 的基础上，完成后面的练习题。

三、 实验原理

RAM 全称为随机存取存储器（Random Access Memory），和只读存储器 ROM（Read Only Memory）不同之处在于，RAM 可写可读，可以随意指定地址进行读写，且速度较快，断电后数据不保存，适合做临时存储介质。

单端口 RAM（Single Port RAM）指的是只有一个地址线和读、写端口，即同一时间只能选择读或写一个地址，只有一个时钟。读出的数据为当前地址的旧值。

伪双端口 RAM（Simple Dual Port RAM）指的是有两个地址线和一个读、一个写端口，同一时间读地址和写地址可以不相同，两个端口时钟可以不相同，但是两个端口只能同时读写或只用写端口或只用读端口，不能两个端口同时写或同时读。

真双端口 RAM（True Dual Port RAM）指的是有两个地址线和两个读写线，同一时间读地址和写地址可以不相同，两个端口时钟可以不相同，两个端口的读写相互独立，即可以同时读或同时写不同的地址数据，也可以一个端口读一个端口写。

一般 RAM 用寄存器组实现，写入数据需要时钟和使能信号，在时钟边沿写入数据。但是读数据可以是异步（没有时钟即使读取）也可以是同步（在时钟边沿读取数据）。

四、 实验过程

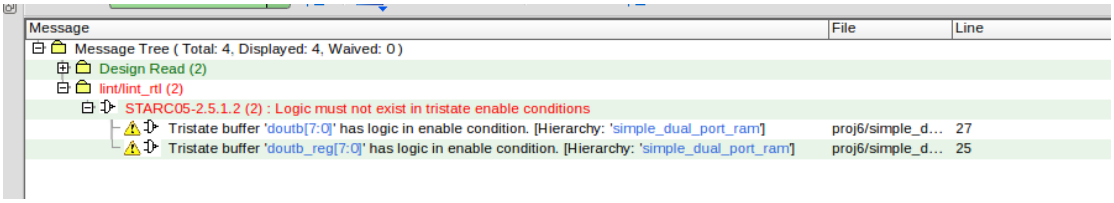
伪双端口 RAM：

verilog 代码：

```
module simple_dual_port_ram #(
    parameter DATA_WIDTH = 8,
    parameter ADDR_WIDTH = 7    // depth = 2**7 = 128
) (
    input clka,
    input clkb,
    input rst_n,
    input wena,
    input renb,
    input [ADDR_WIDTH-1:0] waddra,
    input [ADDR_WIDTH-1:0] raddrb,
    input [DATA_WIDTH-1:0] dina,
    output [DATA_WIDTH-1:0] doutb
);
    reg [DATA_WIDTH-1:0] mem [2**ADDR_WIDTH-1:0];
    always @(posedge clka) begin
        if(wena)
            mem[waddra] <= dina;
    end

    reg [DATA_WIDTH-1:0] doutb_reg;
    always @(posedge clkb, negedge rst_n) begin
        if(~rst_n)
            doutb_reg <= 0;
        else if(renb)
            doutb_reg <= mem[raddrb];
    end
    assign doutb = renb ? doutb_reg : {DATA_WIDTH{1'bz}};

endmodule
```

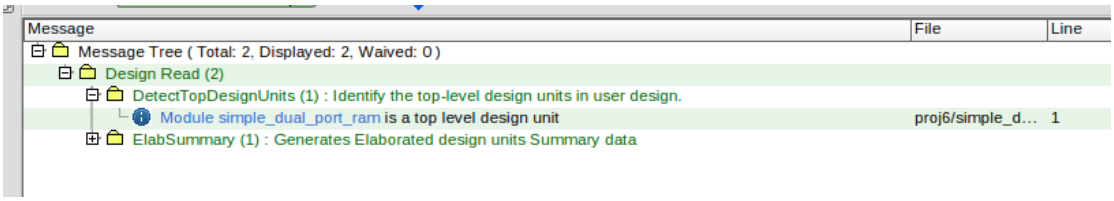
模块 IO 分为 a、b 两个端口，各有独立的时钟，独立的地址线，独立的使能信号，独立的数据端口。其中 a 端口负责写数据，b 端口负责读数据，读写都是时钟同步的。写无效时寄存器输出高阻态 z，表示这是无效的数据。使用了复位信号控制寄存器的复位状态。

spyglass 报告：




```
    always @(posedge clkb) begin
        if(renb)
            doutb_reg <= mem[raddrb];
        else
            doutb_reg <= {DATA_WIDTH{1'bz}};
    end
    assign doutb = doutb_reg;
```

改进后：把三态缓冲器改为组合逻辑：输出直连 dout，输入连接寄存器输出而不是让寄存器有三态输出功能！




```
    always @(posedge clkb, negedge rst_n) begin
        if(~rst_n)
            doutb_reg <= 0;
        else if(renb)
            doutb_reg <= mem[raddrb];
    end
    assign doutb = renb ? doutb_reg : {DATA_WIDTH{1'bz}};
```

包传输练习：

verilog 代码：

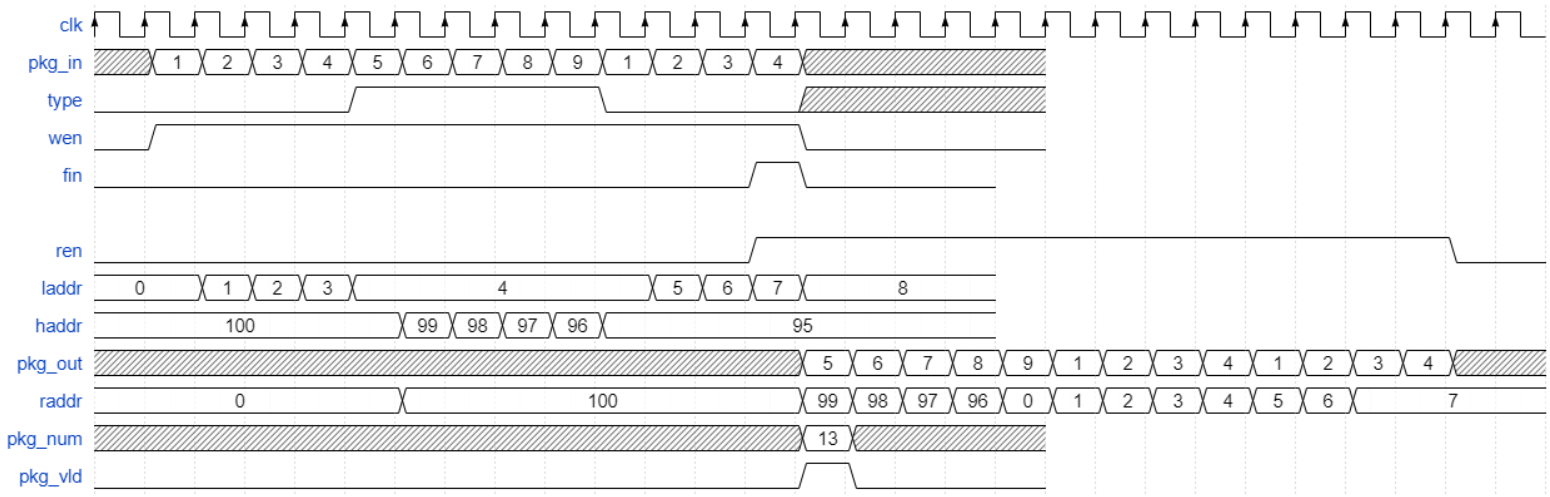
```
module data_package #(
    parameter DATA_WIDTH = 8,
    parameter ADDR_WIDTH = 7
) (
    input clk,
    input rst_n,
    input wen,
    input fin,
    input [DATA_WIDTH-1:0] pkg_in,
    input type,
    output [DATA_WIDTH-1:0] pkg_out,
    output [DATA_WIDTH-1:0] pkg_num,
    output pkg_num_vld
```

```
);
    reg [ADDR_WIDTH-1:0] haddr,laddr;
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            haddr <= {ADDR_WIDTH{1'b1}};
        else if(wen & type)
            haddr <= haddr - 1'b1;
    end
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            laddr <= 0;
        else if(wen & ~type)
            laddr <= laddr + 1'b1;
    end

    wire ren;
    reg fin_reg;
    reg haddr_fin;
    reg laddr_fin;
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            fin_reg <= 1'b0;
        else if(fin)
            fin_reg <= 1'b1;
    end
    assign ren = (fin_reg | fin) & ~laddr_fin;

    wire [ADDR_WIDTH-1:0] waddr;
    reg [ADDR_WIDTH-1:0] raddr;
    assign waddr = (wen & type) ? haddr : laddr;
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n) begin
            raddr <= 0;
            haddr_fin <= 1'b0;
            laddr_fin <= 1'b0;
        end
        else if(wen & type) // preparation for high address
            raddr <= {ADDR_WIDTH{1'b1}};
        else if(ren) begin    // read mode
            if(haddr != {ADDR_WIDTH{1'b1}} && ~haddr_fin) begin    //
high address
                if(raddr == haddr + 1'b1) begin
                    raddr <= 0;
                    haddr_fin <= 1'b1;
```

```
            end
            else
                raddr <= raddr - 1'b1;
        end
        else begin   // low address
            if(raddr == laddr - 1'b1) begin
                raddr <= raddr;
                laddr_fin <= 1'b1;
            end
            else
                raddr <= raddr + 1'b1;
        end
    end
end

simple_dual_port_ram #(
    .DATA_WIDTH(DATA_WIDTH),
    .ADDR_WIDTH(ADDR_WIDTH)
) u1 (
    .clka(clk),
    .clkb(clk),
    .rst_n(rst_n),
    .wena(wen),
    .renb(ren),
    .waddra(waddr),
    .raddrb(raddr),
    .dina(pkg_in),
    .doutb(pkg_out)
);

reg vld_reg;
always @(posedge clk, negedge rst_n) begin
    if(~rst_n)
        vld_reg <= 1'b0;
    else
        vld_reg <= fin;
end
assign pkg_num_vld = vld_reg;

reg [DATA_WIDTH-1:0] pkg_num_reg;
always @(posedge clk, negedge rst_n) begin
    if(~rst_n)
        pkg_num_reg <= 0;
    else if(wen)
```

```
            pkg_num_reg <= pkg_num_reg + 1'b1;
    end
    assign pkg_num = pkg_num_vld ? pkg_num_reg : {DATA_WIDTH{1'bz}};
    // assign pkg_num = pkg_num_vld ? ({ADDR_WIDTH{1'b1}} - haddr +
laddr - 2) : {DATA_WIDTH{1'bz}};

endmodule
```

时序图：



在原有题目的基础上，对于数据包的传输，我增加了 wen 写使能信号控制 RAM 的写数据和 fin 信号作脉冲信号表示数据包完成传输。

可以看到，在 fin 信号发出后的下一个时钟节拍立刻读取数据，所以模块内读使能需要有寄存器保存 fin 的脉冲（脉冲信号转换为电平信号）且有组合逻辑记录 fin 信号，即 ren=fin|ren_reg，保证在最后 fin 有效时 ren 也有效。整个数据传输过程也只有这个时刻 RAM 同时读写。

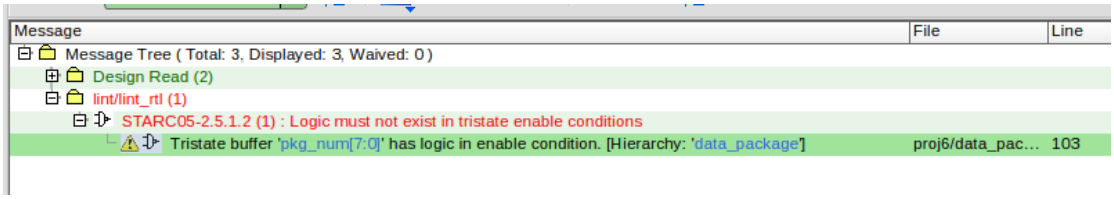
对于地址指针 laddr 和 haddr：它们都用寄存器保存，当 type 为 0 时存入低位地址，地址使用 laddr；当 type 为 1 时存入高位地址，地址使用 haddr。且在对应自身的地址时这两个地址指针递减或递增移动，不在自身范围内时使能信号关闭，寄存器保持原有值。这样最终 fin 信号到来时 haddr 和 laddr 都可作为存储有效数据的指针，指示低位地址和高位地址数据存储到了哪里。这两个指针的复位值为 0,127（地址线总宽度为 7，2^7=128），在存储最多数据的情况下有 10 个数据包，每个数据包 10 个数据，总共只要 100 个寄存器组，每个寄存器组宽度为 8 即可。在最坏情况下这些数据包都存在高位或低位，但是相应另一侧指针不会变化，所以两个指针不会发生重叠，也就没有冲突问题。

记录数据包的个数可以用计数器记录在数据写入阶段时钟个数，也可以用组合逻辑直接根据两个指针的数据作加减法计算得到。

包个数有效信号为脉冲，直接把 fin 信号延迟一个节拍就可以得到。

在数据读出时先读高位信号再读低位信号。由于写完数据后就要立即读出，且不知道数据包有没有存入高地址数据，所以在数据输入阶段就要时刻检测是否有高位地址数据：没检测到从 0 开始读，检测到就从最高位开始读，读到 haddr 指针后从 0 开始读，读到 laddr 后关闭读使能信号，读过程结束。

spyglass 报告：

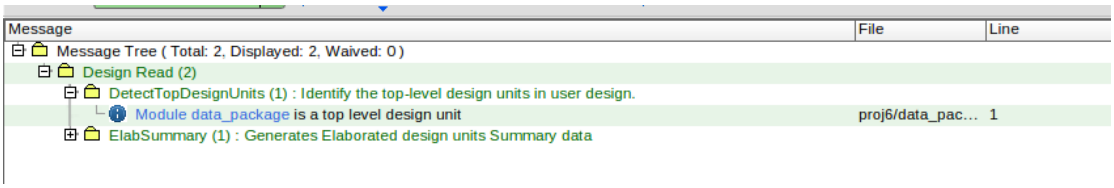


The STARC05-2.5.1.2 rule reports violation for logic in tristate enable conditions. The tristate buffer enable (select) condition must have only a signal name described without using more complex logic. Complex logic like any combo logic, sequential cell, etc., except inverters and buffers between tristate enable and module input ports, if used in tristate enable condition is reported.

```
assign pkg_num = pkg_num_vld ? pkg_num_reg : {DATA_WIDTH{1'bz}};
```

三态门的输入不能是复杂逻辑，只能是一个信号。

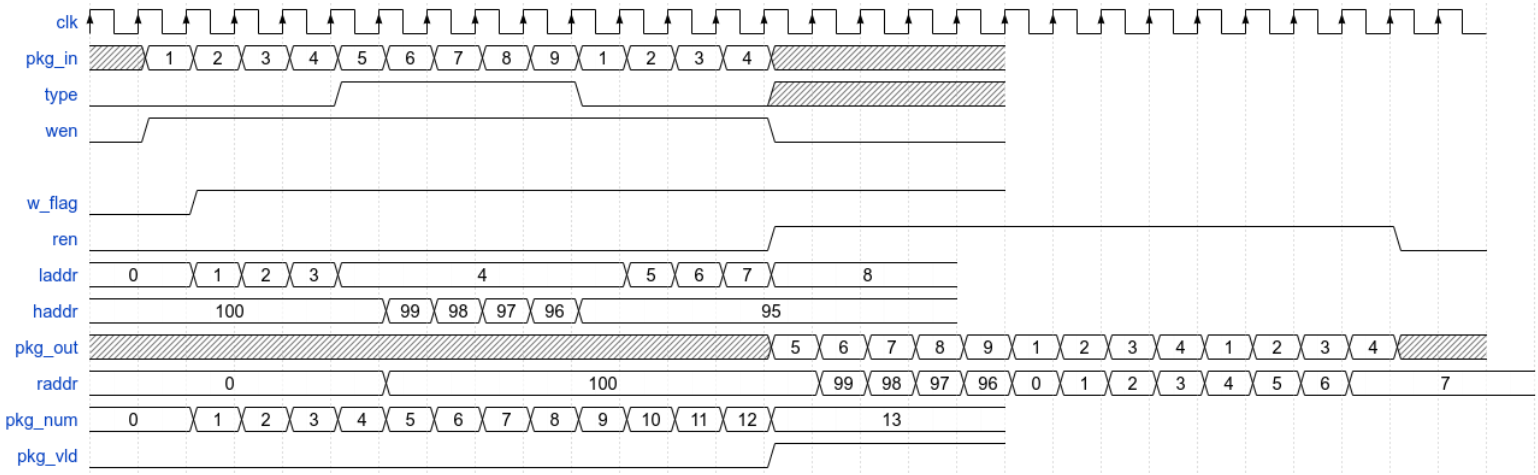
移除不必要的三态们后 warning 解决，因为有 valid 的脉冲信号告诉其他模块此信号的有效情况。




```
assign pkg_num = pkg_num_reg;
```

更好的设计：

使用异步读的 RAM，即输出寄存器改为多路选择器直接连接，ren 使能信号到来时直接读出寄存器的值，不存在读取延时。这样做的好处在于不需要再外界给 fin 信号标志完成也可以在写信号结束后立即读出数据，不需要延时一个节拍。



在 wen 信号下降的同时 ren 信号拉高，这里不需要寄存器延时，直接组合逻辑对 wen 取反就可以。w_flag 表征写过数据，保证在产生 ren 信号时，写入数据之前~wen=1，w_flag=0，不至于仅依靠~wen 导致在写入数据之前 ren 就有效。

另外，raddr 在 ren 有效之后再递减或递增，由于 ren 比之前提前，所以 raddr 的设计不需要改动。其他信号的连接和之前一样。

verilog 代码：

```
module simple_dual_port_ram_async #(
parameter DATA_WIDTH = 8,
parameter ADDR_WIDTH = 7
) (
input clka,
input clkb,
input wena,
input renb,
input [ADDR_WIDTH-1:0] waddra,
input [ADDR_WIDTH-1:0] raddrb,
input [DATA_WIDTH-1:0] dina,
output [DATA_WIDTH-1:0] doutb
);
reg [DATA_WIDTH-1:0] mem [2**ADDR_WIDTH-1:0];
always @(posedge clka) begin
if(wena)
mem[waddra] <= dina;
end

assign doutb = renb ? mem[raddrb] : 0;
```

```
endmodule
```

```
module data_package_async #(
parameter DATA_WIDTH = 8,
parameter ADDR_WIDTH = 7
) (
input clk,
input rst_n,
input wen,
input [DATA_WIDTH-1:0] pkg_in,
input type,
output [DATA_WIDTH-1:0] pkg_out,
output [ADDR_WIDTH-1:0] pkg_num,
output pkg_num_vld
);
reg [ADDR_WIDTH-1:0] haddr,laddr;
always @(posedge clk, negedge rst_n) begin
if(~rst_n)
haddr <= {ADDR_WIDTH{1'b1}};
else if(wen & type)
haddr <= haddr - 1'b1;
end
always @(posedge clk, negedge rst_n) begin
if(~rst_n)
laddr <= 0;
else if(wen & ~type)
laddr <= laddr + 1'b1;
end
```

```
wire ren;
reg haddr_fin;
reg laddr_fin;
reg w_flag;
always @(posedge clk, negedge rst_n) begin
if(~rst_n)
w_flag <= 1'b0;
else if(wen)
w_flag <= 1'b1;
end
assign ren = ~wen & ~laddr_fin & w_flag;
reg r_flag;
always @(posedge clk, negedge rst_n) begin
```

```
if(~rst_n)
r_flag <= 1'b0;
else if(ren)
r_flag <= 1'b1;
end
assign pkg_num_vld = ren | r_flag;
```

```
wire [ADDR_WIDTH-1:0] waddr;
reg [ADDR_WIDTH-1:0] raddr;
assign waddr = type ? haddr : laddr;
always @(posedge clk, negedge rst_n) begin
if(~rst_n) begin
raddr <= 0;
haddr_fin <= 1'b0;
laddr_fin <= 1'b0;
end
else if(ren) begin // read mode
if(haddr != {ADDR_WIDTH{1'b1}} && ~haddr_fin) begin // high address
if(raddr == haddr + 1'b1) begin
raddr <= 0;
haddr_fin <= 1'b1;
end
else
raddr <= raddr - 1'b1;
end
else begin // low address
if(raddr == laddr - 1'b1) begin
raddr <= raddr;
laddr_fin <= 1'b1;
end
else
raddr <= raddr + 1'b1;
end
end
else if(wen & type) // preparation for high address
raddr <= {ADDR_WIDTH{1'b1}};
end
```

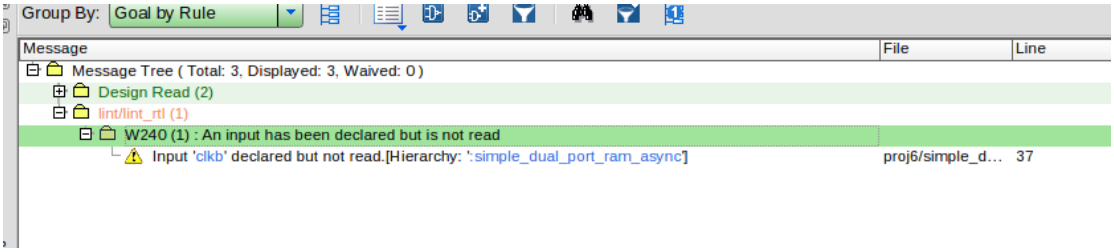
```
simple_dual_port_ram_async #(
.DATA_WIDTH(DATA_WIDTH),
.ADDR_WIDTH(ADDR_WIDTH)
) u1 (
.clka(clk),
.clkb(clk),
```

```
.wena(wen),
.renb(ren),
.waddra(waddr),
.raddrb(raddr),
.dina(pkg_in),
.doutb(pkg_out)
);
```

```
reg [DATA_WIDTH-1:0] pkg_num_reg;
always @(posedge clk, negedge rst_n) begin
if(~rst_n)
pkg_num_reg <= 0;
else if(wen)
pkg_num_reg <= pkg_num_reg + 1'b1;
end
assign pkg_num = pkg_num_reg;
```

```
endmodule
```

spyglass 报告：clkb 未使用，建议去除



把 clkb 端口去除就 OK。

五、 仿真结果

伪双端口 RAM：

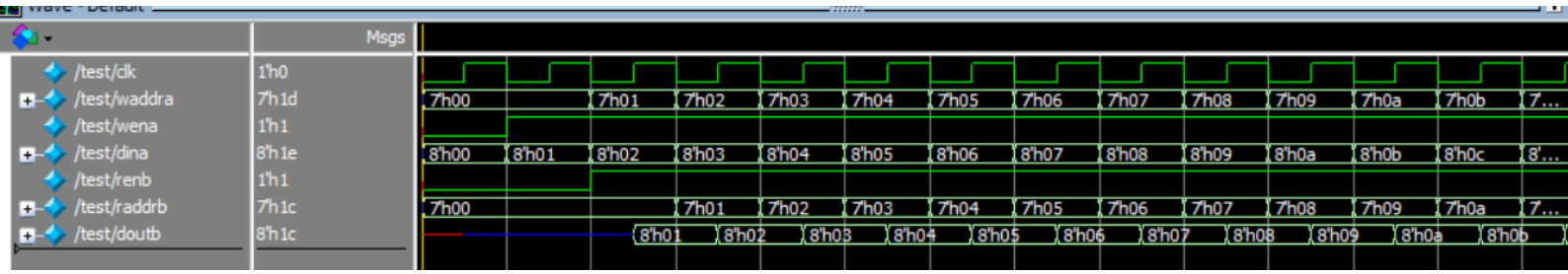
testbench：

```
`timescale 1ps/1ps
module test ();
    parameter DATA_WIDTH = 8;
    parameter ADDR_WIDTH = 7;    // depth = 2**7 = 128
    reg clk;
    reg rst_n;
    reg wena;
    reg renb;
    reg [ADDR_WIDTH-1:0] waddra;
    reg [ADDR_WIDTH-1:0] raddrb;
    reg [DATA_WIDTH-1:0] dina;
```

```
    wire [DATA_WIDTH-1:0] doutb;

    simple_dual_port_ram #(
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(ADDR_WIDTH)
    ) u1 (
        .clka(clk),
        .clkb(clk),
        .rst_n(rst_n),
        .wena(wena),
        .renb(renb),
        .waddra(waddra),
        .raddrb(raddrb),
        .dina(dina),
        .doutb(doutb)
    );

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

    integer i;
    initial begin
        wena = 1'b0;
        waddra = 0;
        dina = 0;
        rst_n = 1'b0;
        #10;
        wena = 1'b1;
        rst_n = 1'b1;
        for(i = 0; i < 2**ADDR_WIDTH; i = i + 1) begin
            waddra = i;
            dina = dina + 1'b1;
            #10;
        end
    end

    integer j;
    initial begin
        renb = 1'b0;
        raddrb = 0;
        #20;
        renb = 1'b1;
```

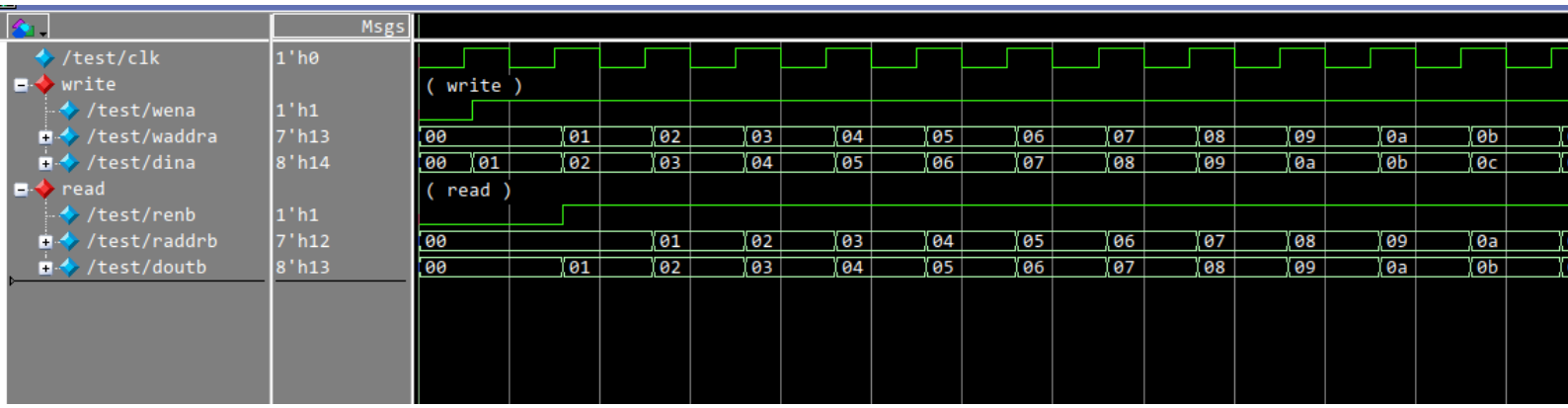
```
        for(j = 0; j < 2**ADDR_WIDTH; j = j + 1) begin
            raddrb = j;
            #10;
        end
    end

endmodule
```

仿真波形：



波形和预期相一致：读写相对独立，可以正确读出相应的数据。上升边沿给定地址和数据，读过程慢一拍，可以看到读出的地址的顺序和写入地址的顺序一致，读到的数据的顺序和写入数据的顺序一致。

异步 RAM：



可以看到在给定读地址后立即得到 RAM 的数据。

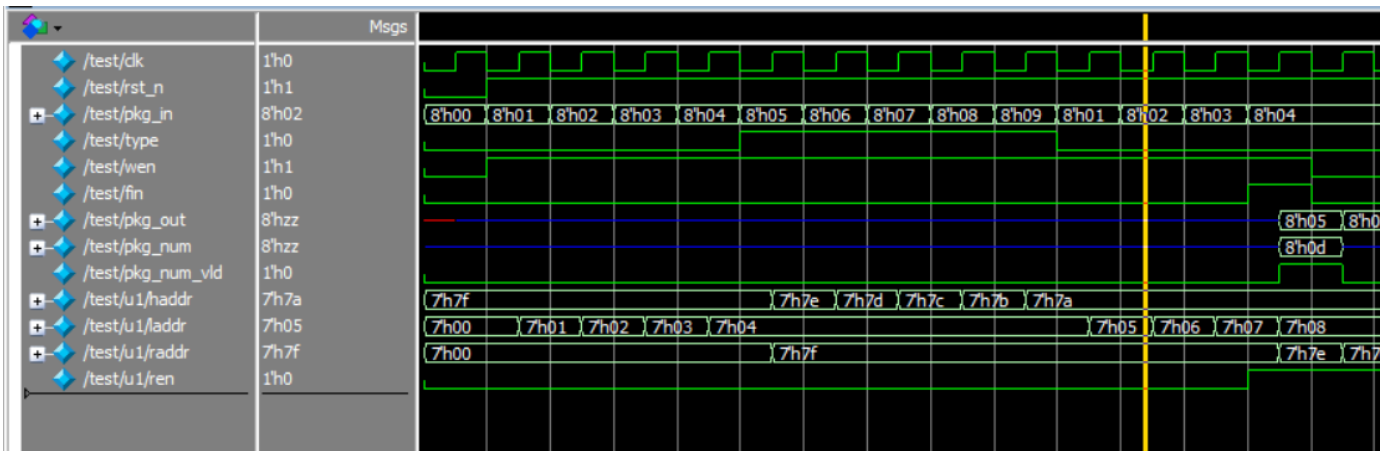
数据包传输：

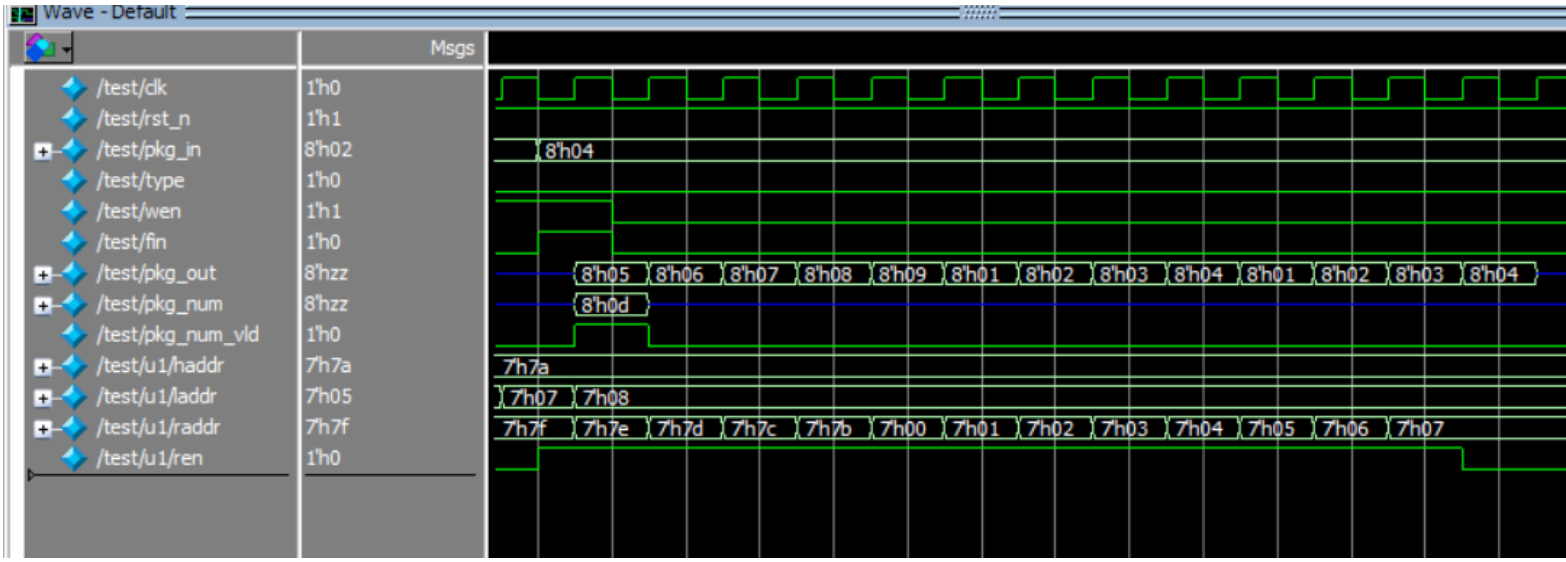
testbench：

```
`timescale 1ps/1ps
module test ();
    parameter DATA_WIDTH = 8;
    parameter ADDR_WIDTH = 7;
    reg clk;
    reg rst_n;
    reg wen;
    reg fin;
    reg [DATA_WIDTH-1:0] pkg_in;
    reg type;
    wire [DATA_WIDTH-1:0] pkg_out;
    wire [DATA_WIDTH-1:0] pkg_num;
    wire pkg_num_vld;

    data_package #(
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(ADDR_WIDTH)
    ) u1 (
        .clk(clk),
        .rst_n(rst_n),
        .wen(wen),
        .fin(fin),
        .pkg_in(pkg_in),
        .type(type),
        .pkg_out(pkg_out),
        .pkg_num(pkg_num),
        .pkg_num_vld(pkg_num_vld)
    );

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

    initial begin
        rst_n = 1'b0;
        wen = 1'b0;
        fin = 1'b0;
        pkg_in = 0;
        type = 1'b0;
        #10 rst_n = 1'b1;
        wen = 1'b1;
        pkg_in = 1;
        #10 pkg_in = 2;
```

```
        #10 pkg_in = 3;
        #10 pkg_in = 4;
        #10 pkg_in = 5;
        type = 1'b1;
        #10 pkg_in = 6;
        #10 pkg_in = 7;
        #10 pkg_in = 8;
        #10 pkg_in = 9;
        #10 pkg_in = 1;
        type = 1'b0;
        #10 pkg_in = 2;
        #10 pkg_in = 3;
        #10 pkg_in = 4;
        fin = 1'b1;
        #10 fin = 1'b0;
        wen = 1'b0;
    end


endmodule
```

仿真波形：





波形结果和示例一致。

使用更实用、更可靠的 testbench：使用随机化输入，使用验证手段：

testbench 代码：

```
`timescale 1ps/1ps
module test (
    output reg check_num,
    output reg check_data
);
    parameter DATA_WIDTH = 8;
    parameter ADDR_WIDTH = 7;
    reg clk;
    reg rst_n;
    reg wen;
    reg fin;
    reg [DATA_WIDTH-1:0] pkg_in;
    reg type;
    wire [DATA_WIDTH-1:0] pkg_out;
    wire [ADDR_WIDTH-1:0] pkg_num;
    wire pkg_num_vld;

    data_package #(
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(ADDR_WIDTH)
    ) u1 (
        .clk(clk),
        .rst_n(rst_n),
        .wen(wen),
```

```
        .fin(fin),
        .pkg_in(pkg_in),
        .type(type),
        .pkg_out(pkg_out),
        .pkg_num(pkg_num),
        .pkg_num_vld(pkg_num_vld)
    );
    // check module
    reg [ADDR_WIDTH-1:0] check_pkg_num_low;
    reg [ADDR_WIDTH-1:0] check_pkg_num_high;
    reg [DATA_WIDTH-1:0] check_pkg_out_low [2**ADDR_WIDTH-1:0];
    reg [DATA_WIDTH-1:0] check_pkg_out_high [2**ADDR_WIDTH-1:0];
    reg [ADDR_WIDTH-1:0] check_read_addr;
    task check_data_put;
        input type;
        input [DATA_WIDTH-1:0] data;
        if(type) begin
            check_pkg_out_high[check_pkg_num_high] = data;
            check_pkg_num_high = check_pkg_num_high + 1;
        end
        else begin
            check_pkg_out_low[check_pkg_num_low] = data;
            check_pkg_num_low = check_pkg_num_low + 1;
        end
    endtask

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

    reg [3:0] rand_pkg_len;
    integer i,j,k;
    initial begin
        for(k = 0; k < 5; k = k + 1) begin
            check_pkg_num_high = 0;
            check_pkg_num_low = 0;
            rst_n = 1'b0;
            wen = 1'b0;
            fin = 1'b0;
            pkg_in = 0;
            type = 1'b0;
            rand_pkg_len = 0;
```

```
            #11 rst_n = 1'b1;
            wen = 1'b1;
            for(i = 0; i < 10; i = i + 1) begin
                rand_pkg_len = 4 + {$random} % (10 - 4 + 1);    // 4~10
                type = $random;
                for(j = 0; j < rand_pkg_len; j = j + 1) begin
                    pkg_in = $random;  // auto trunk
                    check_data_put(type,pkg_in);
                    if(i >= 9 && j >= rand_pkg_len - 1)
                        fin = 1'b1;
                    @(negedge clk);
                end
            end
            fin = 1'b0;
            wen = 1'b0;
            // OK for read check
            if(pkg_num_vld && pkg_num == check_pkg_num_high +
check_pkg_num_low)
                check_num = 1'b1;
            else begin
                check_num = 1'b0;
                $display("wrong pkg num!");
                $finish;
            end
            if(check_pkg_num_high != 0) begin
                check_read_addr = 0;
                while (check_read_addr < check_pkg_num_high) begin
                    if(pkg_out == check_pkg_out_high[check_read_addr])
                        check_data = 1'b1;
                    else begin
                        check_data = 1'b0;
                        $display("wrong data!");
                        $finish;
                    end
                    @(negedge clk);
                    check_read_addr = check_read_addr + 1;
                end
            end
            if(check_pkg_num_low != 0) begin
                check_read_addr = 0;
                while (check_read_addr < check_pkg_num_low) begin
                    if(pkg_out == check_pkg_out_low[check_read_addr])
                        check_data = 1'b1;
                    else begin
```

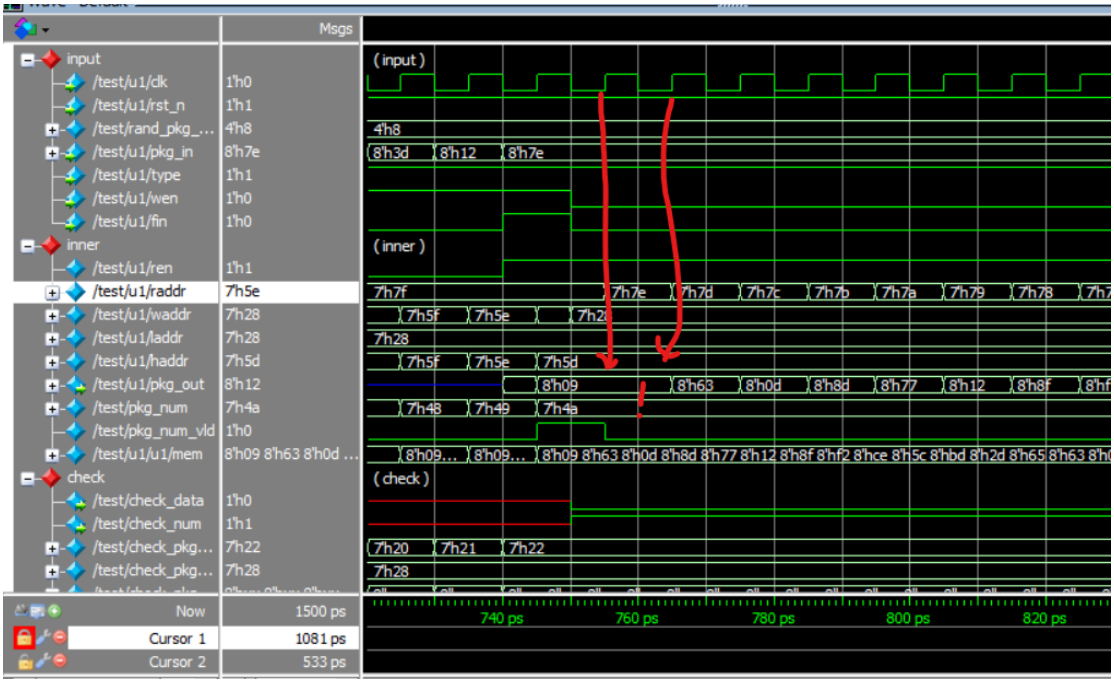
```
                        check_data = 1'b0;
                        $display("wrong data!");
                        $finish;
                    end
                    @(negedge clk);
                    check_read_addr = check_read_addr + 1;
                end
            end
            // @(u1.ren == 1'b0);
            #101;
            @(negedge clk);
        end
    end

endmodule
```

仿真波形：

通过随机化$random 产生 32 位的随机数，{$random}将其变为正数（变成无符号数），作%求余运算得到一定范围的数，付给寄存器作为随机数。可以使每

个数据包的个数、每个数据包类型、每个数据内容进行随机化。在一次程序中每次调用$random 函数，种子都会变化。在一次 testbench 使用中多进行复位重新开启功能（本次示例进行了 5 次，可以看到 rst_n 信号有 5 个脉冲），确保每次的随机数都不同，从而更好地验证模块功能。

但是在更复杂带验证功能的 testbench 中发现了问题：

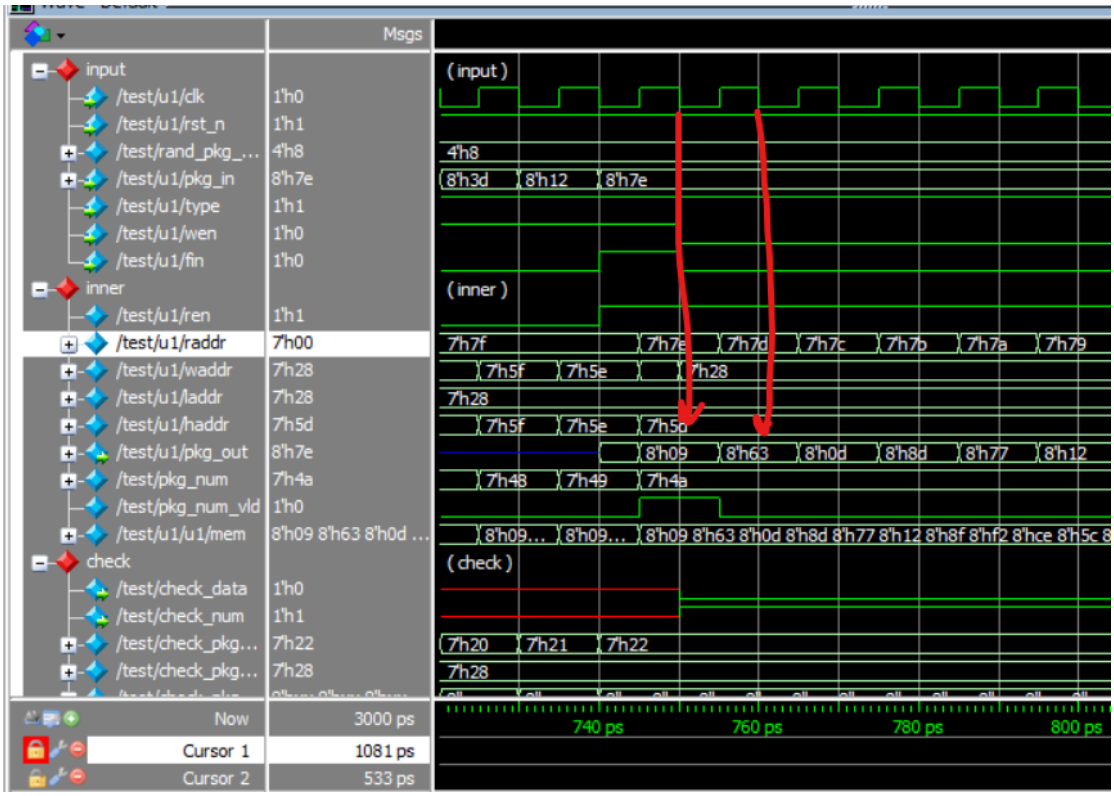


回到 verilog 代码：

```
        else if(wen & type) // preparation for high address
            raddr <= {ADDR_WIDTH{1'b1}};
        else if(ren) begin    // read mode
            ......
```

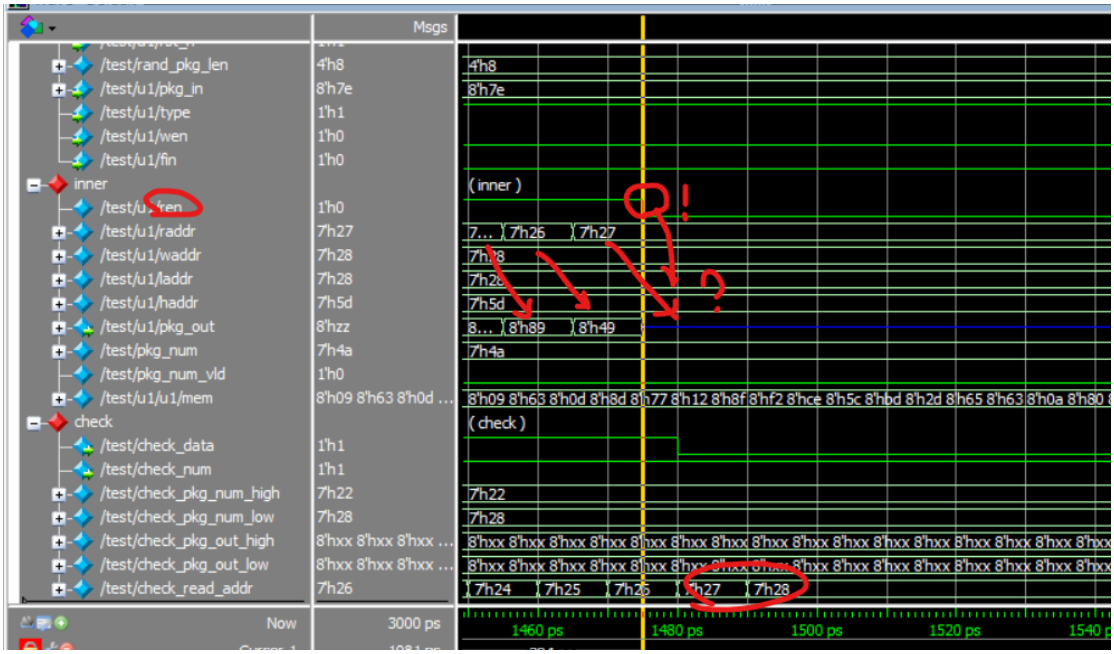
对 raddr 的寄存器配置输入时优先级错误，导致在同时读写阶段（最后一个数据输入，即 fin 信号有效时）有限选择复位 raddr 位准备阶段，最终导致整个读周期延长一拍，数据完全出错。改进：

```
        else if(ren) begin    // read mode
            ......
        else if(wen & type) // preparation for high address
            raddr <= {ADDR_WIDTH{1'b1}};
```



问题得到解决。

另外一个问题：在输出结束时读使能信号期望时同步信号，但是由于之前迁就三态门的设计导致读使能信号既是同步信号又是异步信号，导致混乱。



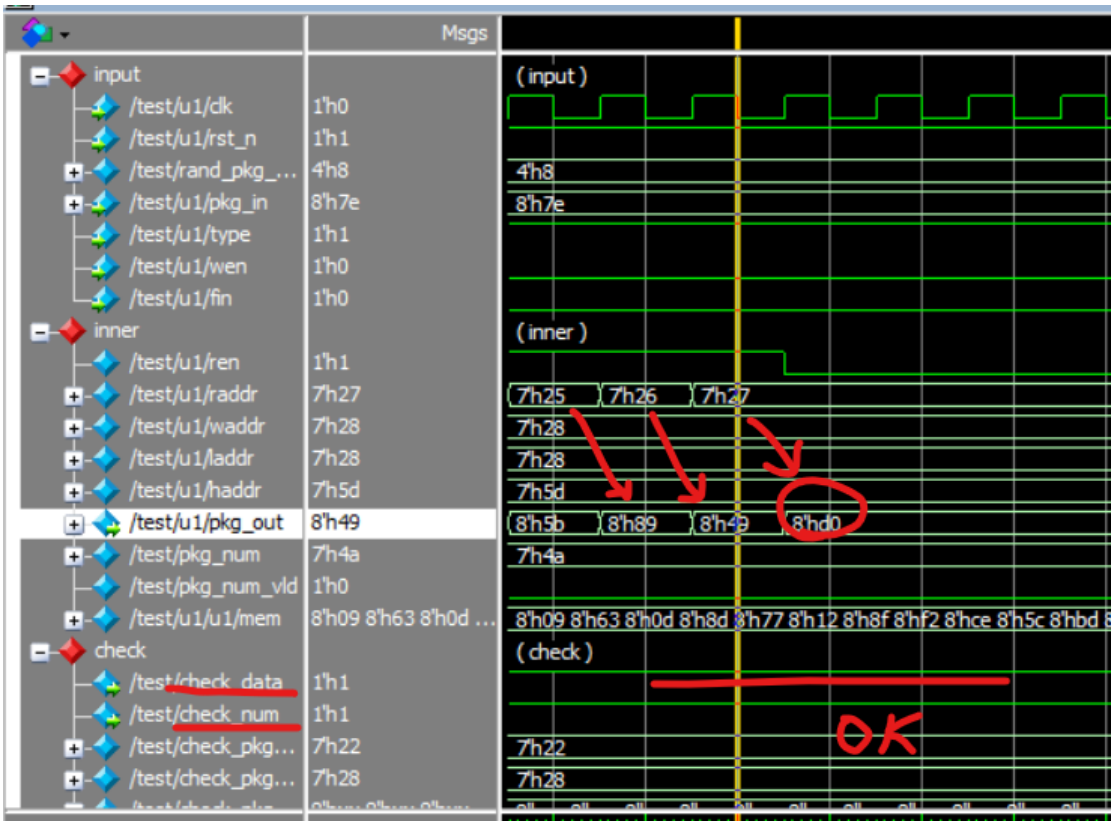

```
always @(posedge clkb, negedge rst_n) begin
    if(~rst_n)
        doutb_reg <= 0;
    else if(renb)
        doutb_reg <= mem[raddrb];
```

```
end
assign doutb = renb ? doutb_reg : {DATA_WIDTH{1'bz}};
```

把异步的部分删掉，不要三态们了：

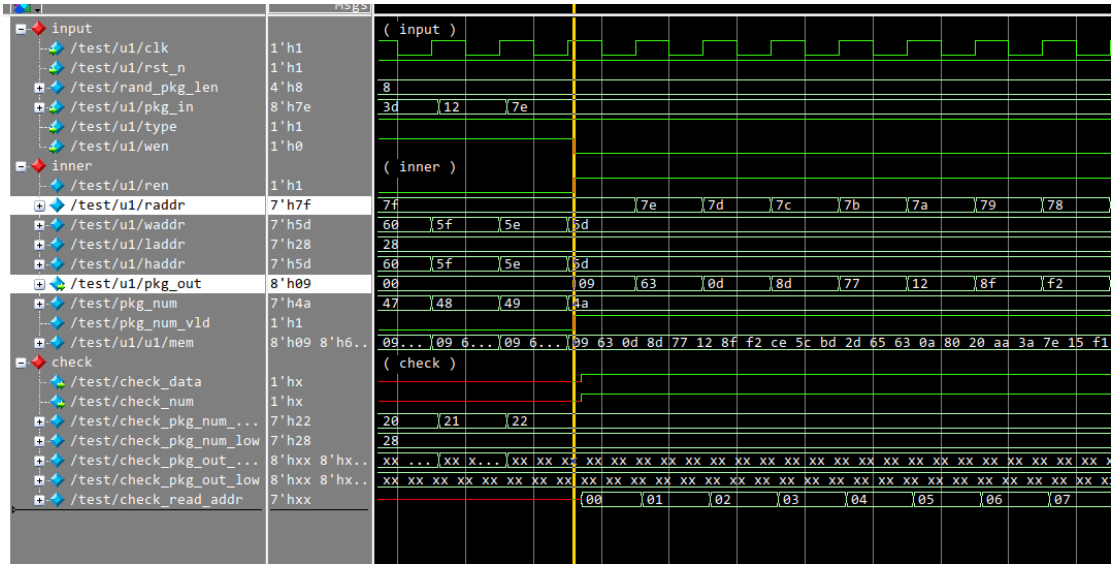
```
assign doutb = doutb_reg;
```

结果：



完美解决问题。

异步 RAM：



可以看到在读出数据的时候不需要 fin 信号，ren 信号在 wen 无效时立即生效，读出数据在读地址有效时就可以输出出来。其他部分和上述一致。

六、 实验总结

本次实验了解了 RAM 的端口种类，分为单端口，伪双端口，真双端口，并据此使用寄存器利用 verilog 代码实现类似于 RAM 的功能，编写 testbench 检验 RAM 的功能。并在实际应用中（本次练习：实现数据包的传输）将此伪双端口例化使用，检验 RAM 的功能正确，为以后学习设计同步和异步 FIFO 做好准备。

数据包传输的电路设计中通过画波形图发现有很多控制信号需要自己设计，但是光看题目需求并不知道需要这么多信号要设计，低估了设计难度。通过画波形图的方式方便了自己的设计，减少了很多设计时间，大大提高了效率，且对电路结构的认识更加深入，获益良多。

使用 testbench 学会了随机化输入，学会用 testbench 自由的写法实验验证电路功能，且真正检查到了设计中的问题，感悟很大，感到收获很多，切实感受到电路设计的重要和困难：虽然软件也很容易写逻辑错误，有很小的输入覆盖不到，但是软件易改硬件电路流片后有错误，成本太高承受不起。所以我今后设计电路时一定要把波形图，各个信号的功能弄清楚再写代码，写完后考虑到尽可能全面的情况，不要出现部分输入逻辑错误的情况。细节决定成败！