实验报告

曾锦鸿
2021k8009929009

实验目的：

1、熟悉 verilog 编程、调试
2、熟悉 FIFO 工作原理
3、实现功能较复杂的数字电路

实验环境：

vivado版本号为2019.2

原理说明：

1、只存在一种工作状态的FIFO，即要么只接收数据不输出数据，要么只输出数据不接收数据：

这里我使用一个wr寄存器保存FIFO的状态，其中wr为1为写状态，wr为0为读状态，只有在完成最后一次读或者最后一次写时切入另一种状态。

2、FIFO 中有数据就可以向外读出数据，FIFO 没有填满就可以向内写入数据：

增添32位宽的valid信号，表示FIFO中对应位置的数据是否被读过，被读过为0，反之为1。同时用empty和full信号控制是否可以读出和写入。

接口定义：

1、检测序列0110的状态机：

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
data_in	IN	16	写入FIFO的数据
rstn	IN	1	复位信号的非（当它为0时，复位）
write_req	IN	1	写FIFO的请求
read_req	IN	1	读FIFO的请求
wr	OUT	1	FIFO的状态，1为写状态，0为读状态
data_out	OUT	8	读出FIFO的数据
write_addr	OUT	5	写FIFO的地址
read_addr	OUT	5	读FIFO的地址

2、检测序列11011的状态机：

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
data_in	IN	16	写入FIFO的数据
rstn	IN	1	复位信号的非（当它为0时，复位）
write_req	IN	1	写FIFO的请求
read_req	IN	1	读FIFO的请求
write_rdy	OUT	1	在读写不冲突的情况下，允许FIFO进行写操作的信号
read_rdy	OUT	1	在读写不冲突的情况下，允许FIFO进行读操作的信号
data_out	OUT	8	读出FIFO的数据
write_addr	OUT	5	写FIFO的地址
read_addr	OUT	5	读FIFO的地址
valid	OUT	32	FIFO对应位置的数据是否被读过，未读过为1，反之为0
conf	OUT	1	是否发生了读写冲突

调试过程及结果波形：

本次实验较为简单，写完后并未出过bug，课上很快就可以完成，所以无调试过程。

1、只存在一种工作状态的FIFO，即要么只接收数据不输出数据，要么只输出数据不接收数据：

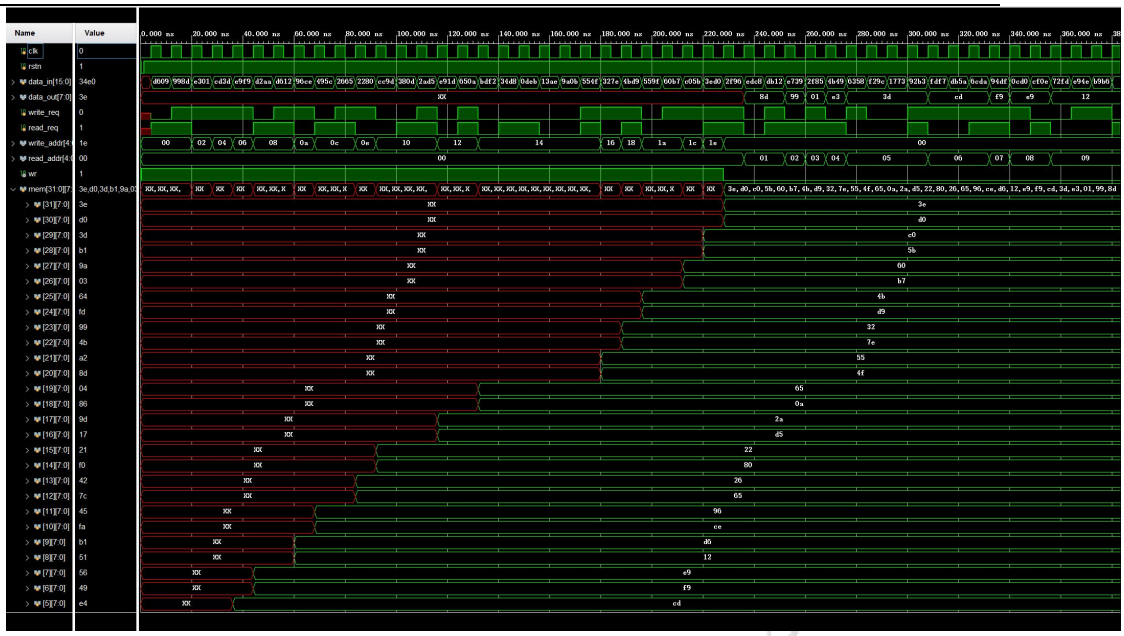
激励文件：

```
module test_fifo(

    );
    reg clk,rstn;
    reg [15:0] data_in;
    wire [7:0] data_out;
    reg write_req;
    reg read_req;
    wire [4:0] write_addr;
    wire [4:0] read_addr;
    wire wr;
    FIFO1 u_FIFO(
        .clk(clk),
        .rstn(rstn),
        .write_req(write_req),
        .read_req(read_req),
        .wr(wr),
        .data_in(data_in),
        .data_out(data_out),
        .write_addr(write_addr),
        .read_addr(read_addr)
    );
    initial begin
        clk = 0;
        rstn = 1;
        #0.1 rstn = 0;
        #1.1 rstn = 1;
    end
    always begin
        #4 clk = ~clk;
    end
    always @(posedge clk) begin
        write_req <= $random() % 2;
        read_req <= $random() % 2;
        data_in <= $random() % 17'b10000000000000000;
    end
endmodule
```

在激励文件中，写请求write_req、读请求read_req以及写入数据data_in均随机产生。

波形图如下所示：



(信号优点多，但是确实给出了正确的结果，导致波形图很小)

2、FIFO 中有数据就可以向外读出数据，FIFO 没有填满就可以向内写入数据：

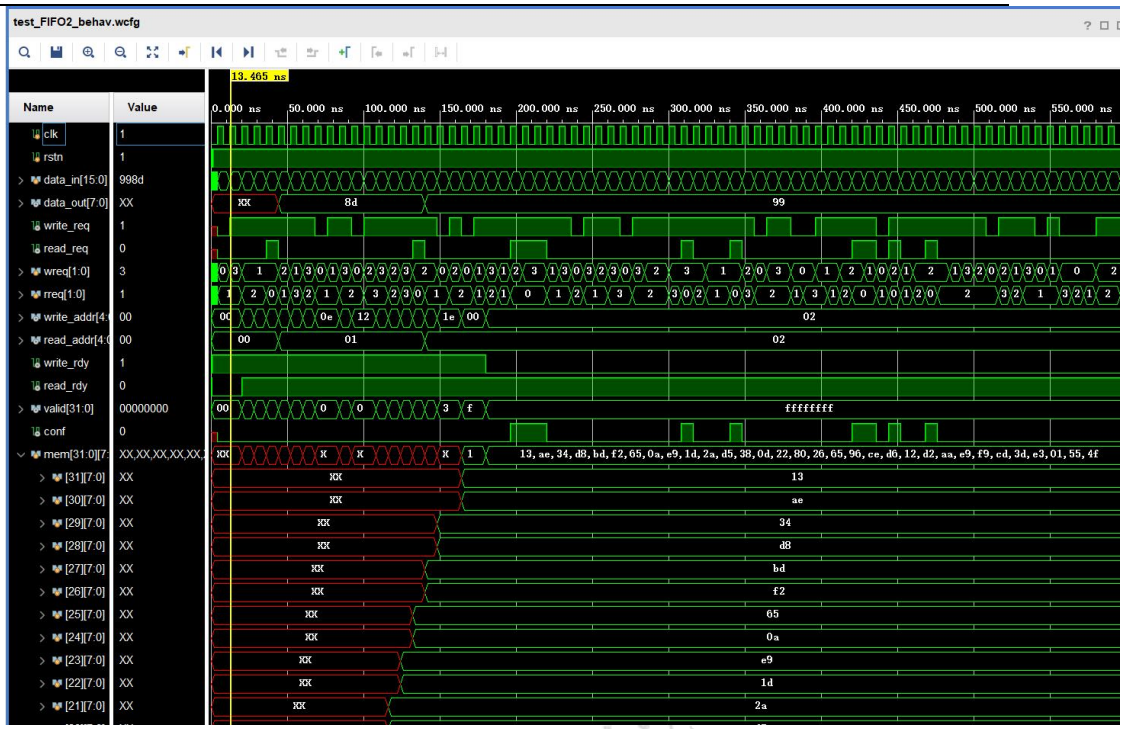
激励文件：

```
module test_FIFO2(

    );
    reg clk,rstn;
    reg [15:0] data_in;
    wire [7:0] data_out;
    wire write_req;
    wire read_req;
    reg [1:0] wreq;
    reg [1:0] rreq;
    wire [4:0] write_addr;
    wire [4:0] read_addr;
    wire write_rdy;
    wire read_rdy;
    wire [31:0] valid;
    wire conf;
    FIFO2 u_FIFO(
        .clk(clk),
        .rstn(rstn),
        .write_req(write_req),
        .read_req(read_req),
        .write_rdy(write_rdy),
        .read_rdy(read_rdy),
        .data_in(data_in),
        .data_out(data_out),
        .write_addr(write_addr),
        .read_addr(read_addr),
        .valid(valid),
        .conf(conf)
    );
    initial begin
        clk = 0;
        rstn = 1;
        #0.1 rstn = 0;
        #1.1 rstn = 1;
    end
    always begin
        #4 clk = ~clk;
    end
    always @(posedge clk) begin
        wreq <= $random() % 4;
        rreq <= $random() % 4;
        data_in <= $random() % 17'b10000000000000000;
    end
    assign write_req = wreq != 2'd0;
    assign read_req = rreq == 2'd0;
endmodule
```

题中要求编写的testbench中写请求与读请求的比例为3：1，而且写请求和读请求可以同时发出，所以我申明了两个请求生成的寄存器，它们都是2位，每个时钟周期更新为0-3的随机数，当wreq不为0时，写请求write_req置为1，当rreq为0时，读请求read_req被置为1。

波形图如下所示：



同理，信号有点多，但是功能确实正确。

实验总结：

本次实验总体来说并不困难，主要是让同学们熟悉一下握手的概念以及同步FIFO的概念。

源代码：

1、基础实验：

```
module FIFO1(
    input clk,
    input rstn,
    input write_req,
    input read_req,
    output reg wr,//1为写，0为读，其实就是两个rdy的综合
    input [15:0] data_in,
    output reg [7:0] data_out,
    output reg [4:0] write_addr,
    output reg [4:0] read_addr
    );
    reg [7:0] mem [31:0];
    always @(posedge clk or negedge rstn) begin
        if(rstn == 0) begin
            write_addr <= 5'd0;
            read_addr <= 5'd0;
            wr <= 1'd1;
        end else if(wr == 1 && write_req == 1) begin
            mem[write_addr] <= data_in[7:0];
            mem[write_addr + 1] <= data_in[15:8];
            wr <= (write_addr != 5'b11110);
            write_addr <= write_addr + 2;
        end else if(wr == 0 && read_req == 1) begin
            data_out <= mem[read_addr];
            wr <= (read_addr == 5'b11111);
            read_addr <= read_addr + 1;
        end
    end
endmodule
```

2、附加实验：

```
module FIFO2(
    input clk,
    input rstn,
    input write_req,
    input read_req,
    output write_rdy,
    output read_rdy,
    input [15:0] data_in,
    output reg [7:0] data_out,
    output reg [4:0] write_addr,
    output reg [4:0] read_addr,
    output reg [31:0] valid,
    output wire conf
    );
    reg [7:0] mem [31:0];
    wire full;
    wire empty;
    //wire conf;
    assign full = valid[write_addr] == 1 || valid[write_addr + 1] == 1;
    assign empty = valid == 32'd0;
    assign write_rdy = ~full;
    assign read_rdy = ~empty;
    assign conf = ((read_addr == write_addr) || ((write_addr + 1) == read_addr)) && write_req && read_req ;
    always @(posedge clk or negedge rstn) begin
        if(rstn == 0) begin
            write_addr <= 5'd0;
            read_addr <= 5'd0;
            valid <= 32'd0;
        end else begin
            if(write_req == 1 && write_rdy == 1 && !conf) begin
                mem[write_addr] <= data_in[7:0];
                valid[write_addr] <= 1'd1;
                mem[write_addr + 1] <= data_in[15:8];
                valid[write_addr + 1] <= 1'd1;
                write_addr <= write_addr + 2;
            end
         if(read_req == 1 && read_rdy == 1 && !conf) begin
             data_out <= mem[read_addr];
             valid[read_addr] <= 1'd0;
             read_addr <= read_addr + 1;
         end
         if(read_req == 1 && write_req == 1 && empty) begin
             mem[write_addr + 1] <= data_in[15:8];
             valid[write_addr + 1] <= 1'd1;
             write_addr <= write_addr + 2;
             data_out <= data_in[7:0];
             valid[read_addr] <= 1'd0;
             read_addr <= read_addr + 1;
         end
         if(read_req == 1 && write_req == 1 && (write_addr + 1) == read_addr && valid[read_addr] == 1 && valid[write_addr] == 0)begin
             mem[write_addr] <= data_in[7:0];
             valid[write_addr] <= 1'd1;
             data_out <= mem[read_addr];
             mem[write_addr + 1] <= data_in[15:8];
             valid[write_addr + 1] <= 1'd1;
             write_addr <= write_addr + 2;
             read_addr <= read_addr + 1;
         end
        end
    end
endmodule
```