《数字电路》实验报告

实验名称： FIFO 实验 指导教师：王珧，范志华

姓名：韩初晓 学号：2023K8009908002 专业：计算机科学与技术 班级：2306

实验日期：2024.12.12 实验地点：教学楼 224 是否调课/补课：否 成绩：

目录

第1部分 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

1.1 实验环境

- Vivado 2017.4 开发工具
- FPGA 开发平台（根据手册中的默认设置进行选择）

第2部分 实验一：实现同步 FIFO

2.1 实验原理

2.1.1 FIFO 原理

本次实验实现一个深度为 32，存储单元宽度为 16 bit 的 FIFO (First-In, First-Out) 缓冲器。FIFO 是一种先进先出的数据结构，数据按照写入的顺序被读取，常用于数据缓冲和速率匹配等场景。

2.1.2 实现细节

- **数据存储：**使用二维寄存器数组 `mem[31:0][1:0]` 来存储 FIFO 中的数据，每个存储位置可以存储两个 8 bit 数据，从而实现存储 16 bit 数据。
- **写入控制：**通过 `input_valid` 信号指示是否有新数据写入。`input_enable` 信号控制是否允许写入，当 FIFO 为满时，`input_enable` 置低，禁止写入。`write_addr` 作为写入地址指针，同时使用 `write_state` 状态机来控制 16 bit 数据的写入。
- **读取控制：**通过 `output_enable` 信号控制是否允许读取。`output_valid` 信号指示 FIFO 是否有数据可读。当 FIFO 为空时，`output_valid` 置低，禁止读取。`read_addr` 作为读取地址指针。
- **数据宽度：** FIFO 存储的数据宽度是 16 位，由两个 8 位数据构成，读取时一次性读取 16 位数据。
- **满/空判断：**使用 `fifo_empty` 和 `fifo_full` 标志来判断 FIFO 的状态。当 `read_addr` 等于 31 时，FIFO 被认为是空，禁止读取；当 `write_addr` 等于 31 并且 `write_state` 等于 `2'b01` 时，FIFO 被认为满，禁止写入。
- **读写状态机：**使用 `write_state` 来控制写入数据到 `mem` 的顺序。

2.2 接口定义

- 输入信号：
  - `clk`：时钟信号。
  - `rstn`：复位信号，低电平有效。
  - `input_valid`：输入数据有效信号。
  - `output_enable`：输出使能信号。
  - `data[7:0]`：8 位输入数据。
- 输出信号：
  - `write_addr[4:0]`：写入地址指针。

- `read_addr[4:0]`：读取地址指针。
- `input_enable`：输入使能信号，指示 FIFO 是否可以写入数据。
- `output_valid`：输出数据有效信号，指示 FIFO 是否有数据可读。
- `out[15:0]`：16 位输出数据。

2.3 调试过程及结果

2.3.1 代码分析

模块 `task1_module` 实现了深度为 32 的 FIFO 缓存器。核心逻辑包括使用 `mem` 数组存储数据，使用 `write_addr` 和 `read_addr` 指针管理数据的写入和读取。使用 `input_enable` 和 `output_valid` 信号来控制数据的写入和读取。数据以 8bit 写入的，以 16bit 读取。并且通过 `write_state` 状态机控制写入数据到 `mem` 数组的顺序。

2.3.2 仿真验证

在仿真测试中,我们验证了 FIFO 在不同输入条件下的正确性。我们通过控制 `input_valid`, `output_enable`, 和 `data` 输入，观察 `write_addr`, `read_addr`, `input_enable`, `output_valid` 和 `out` 输出，验证了 FIFO 的数据写入和读取功能。仿真波形显示了 FIFO 的满/空标志以及写入和读取地址的正确变化，以及输出数据的有效性。仿真结果如下：

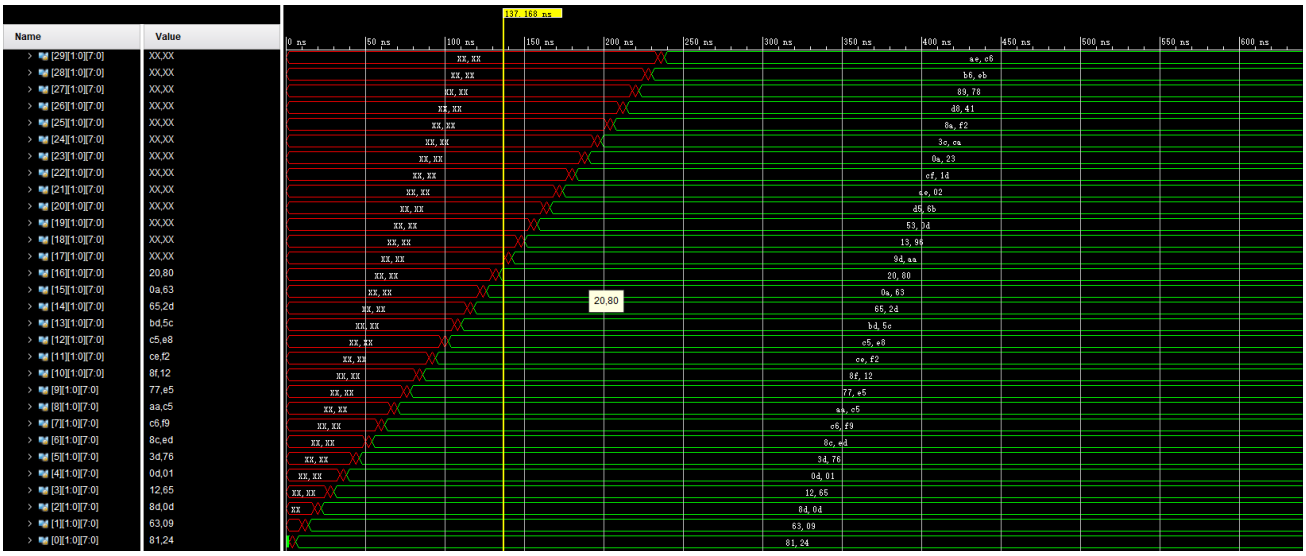

图 1: FIFO 模块的测试结果

第 3 部分 实验二：实现循环读写 FIFO

3.1 实验原理

3.1.1 FIFO 原理

本次实验实现一个深度为 32，存储单元宽度为 16 bit 的 FIFO (First-In, First-Out) 缓冲器。FIFO 是一种先进先出的数据结构，数据按照写入的顺序被读取，常用于数据缓冲和速率匹配等场景。

3.1.2 实现细节

- **数据存储：**使用二维寄存器数组 `mem[31:0][1:0]` 来存储 FIFO 中的数据，每个存储位置可以存储两个 8 bit 数据，从而实现存储 16 bit 数据。
- **写入控制：**通过 `input_valid` 信号指示是否有新数据写入。`input_enable` 信号控制是否允许写入，当 FIFO 接近满时，`input_enable` 置低，禁止写入。`write_addr` 作为写入地址指针。
- **读取控制：**通过 `output_enable` 信号控制是否允许读取。`output_valid` 信号指示 FIFO 是否有数据可读。当 FIFO 为空时，`output_valid` 置低，禁止读取。`read_addr` 作为读取地址指针。
- **数据宽度：** FIFO 存储的数据宽度是 16 位，但是读取时以 8 位方式读取，需要进行两次读取操作才能读取到全部 16 位数据。
- **满/空判断：**当写入地址追上读取地址时，FIFO 被认为满，禁止写入；当读取地址追上写入地址时，FIFO 被认为空，禁止读取。
- **读取操作状态机：**通过 `read_state` 状态机控制读取操作，先读取低 8 位，再读取高 8 位。

3.2 接口定义

- 输入信号：
  - `clk`：时钟信号。
  - `rstn`：复位信号，低电平有效。
  - `input_valid`：输入数据有效信号。
  - `output_enable`：输出使能信号。
  - `data[15:0]`：16 位输入数据。
- 输出信号：
  - `write_addr[4:0]`：写入地址指针。
  - `read_addr[4:0]`：读取地址指针。
  - `input_enable`：输入使能信号，指示 FIFO 是否可以写入数据。
  - `output_valid`：输出数据有效信号，指示 FIFO 是否有数据可读。
  - `out[7:0]`：8 位输出数据。

3.3 调试过程及结果

3.3.1 代码分析

模块 `task2_module` 实现了深度为 32 的 FIFO 缓存器。核心逻辑包括使用 `mem` 数组存储数据，使用 `write_addr` 和 `read_addr` 指针管理数据的写入和读取。使用 `input_enable` 和 `output_valid` 信号来控制数据的写入和读取。数据按 16bit 写入的，按 8bit 读取的，并且使用 `read_state` 状态机控制读取顺序。

3.3.2 仿真验证

在仿真测试中,我们验证了 FIFO 在不同输入条件下的正确性。我们通过控制 `input_valid`, `output_enable`, 和 `data` 输入，观察 `write_addr`, `read_addr`, `input_enable`, `output_valid` 和 `out` 输出，验证了 FIFO 的数据写入和读取功能。仿真波形显示了 FIFO 的满/空标志以及写入和读取地址的正确变化，以及输出数据的有效性。仿真结果如下：

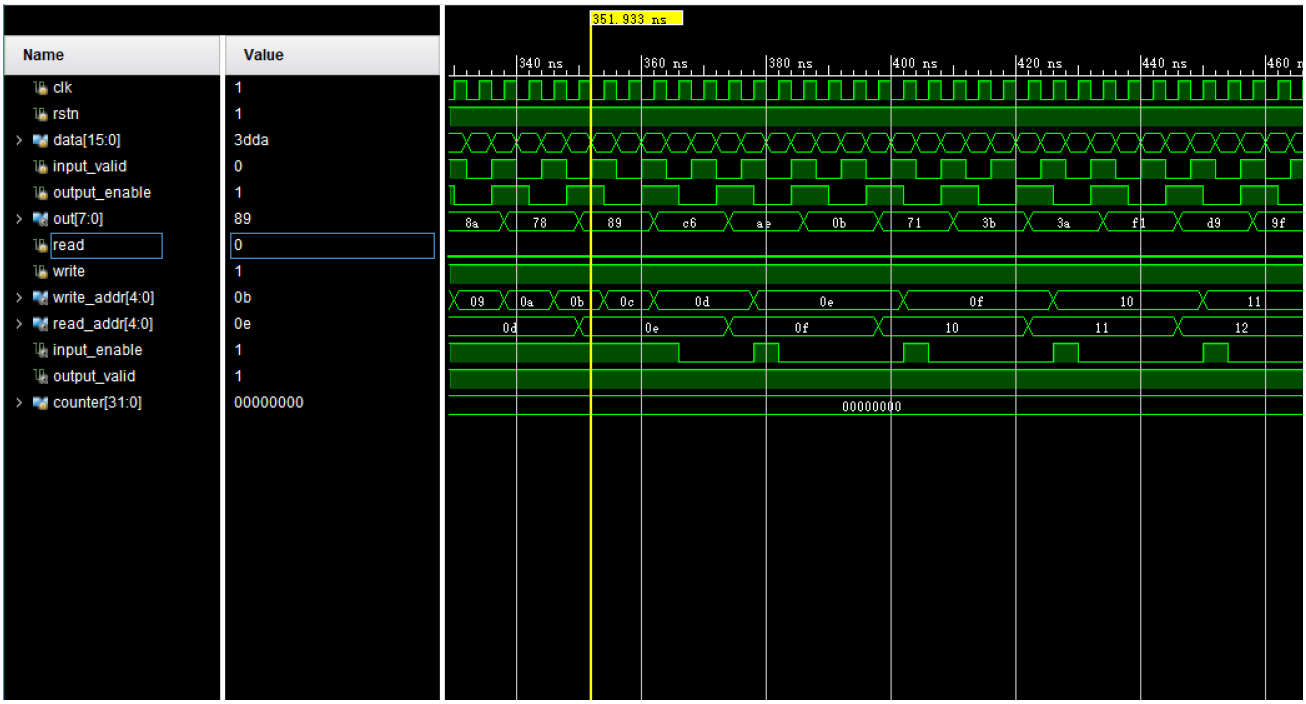

图 2: FIFO 模块的测试结果

根据上述波形图可以看出，当 FIFO 被写满时，只有向外读出一个数据时 FIFO 才能继续写入数据。可见这个 FIFO 模块的功能是正确的。

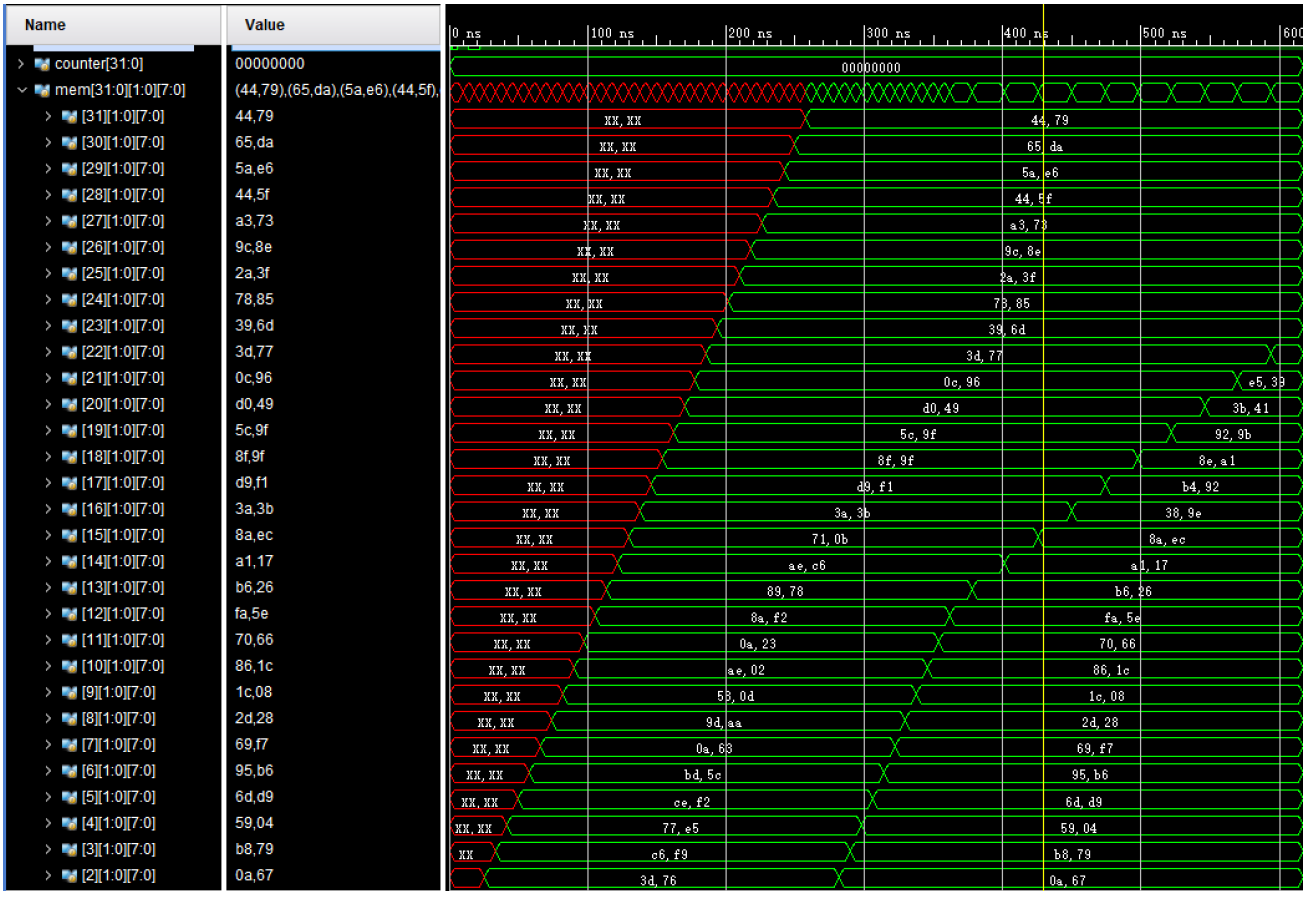

图 3: FIFO 模块的循环读写功能

根据这个波形图可以看出，FIFO 模块可以实现循环读写，在后边写入速度变慢的原因是因为由于在

testbench 中读出的频率比写入要慢，因此在后期 FIFO 处于写满状态，必须等待读出一个数据后方能写入下一个数据。

第 4 部分 实验三：实现 FIFO 的读写控制

由于在实验二中实现的模块的读写位宽已经满足要求，因此在实验三中只需要实现读写控制即可。

4.1 task2_module 接口定义

本次实验的模块仍然为为 task2_module，其接口定义如下：

- **输入信号：**
  - clk：时钟信号。
  - rstn：复位信号，低电平有效。
  - input_valid：输入数据有效信号，指示输入数据 data 有效。
  - output_enable：输出使能信号，指示可以读取输出数据。
  - data [15:0]：16 位输入数据。
- **输出信号：**
  - out [7:0]：8 位输出数据。
  - write_addr [4:0]：FIFO 写地址。
  - read_addr [4:0]：FIFO 读地址。
  - input_enable: FIFO 可以写入信号。
  - output_valid：输出数据有效信号，指示输出数据 out 有效。

4.2 Testbench 描述

Testbench 代码的主要功能如下：

- **时钟和复位：** clk 生成一个周期为 4 个时间单位的时钟信号，rstn 生成一个复位信号，低电平有效。
- **输入信号：**
  - input_valid 信号以 8 个时间单位的周期翻转，模拟输入数据的有效性。
  - output_enable 信号以 12 个时间单位的周期翻转，模拟输出使能信号，由此，保证了读写频率之比为 3 : 2。
  - data 信号随机生成 16 位的数据。
- **仿真结束：**仿真在 600 个时间单位后结束。

4.3 仿真结果分析

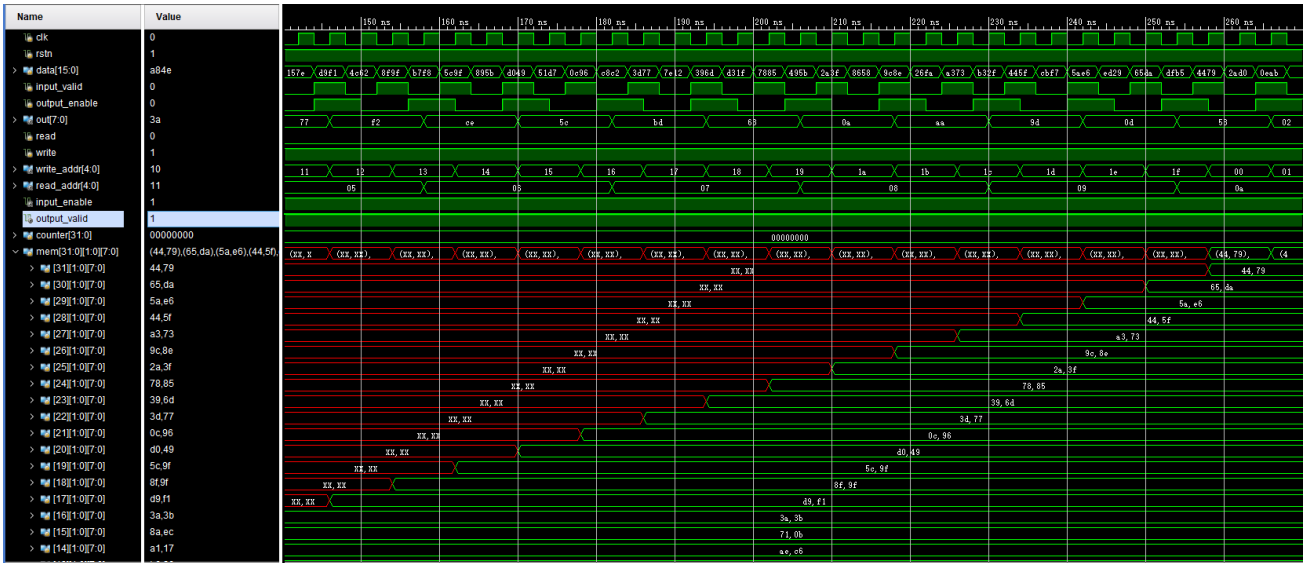

图 4: `task2_module` 模块的测试结果

根据上述波形图，`write_addr` 和 `read_addr` 按照所希望的频率比变化。

第 5 部分 实验总结

在本实验中，我通过 Verilog 实现了一个普通的同步 FIFO 模块，以及一个循环读写的 FIFO 模块。在实验过程中，我熟悉了 FIFO 的工作原理，以及如何通过状态机等方法实现 FIFO 的读写控制。在实验过程中，我通过仿真验证了 FIFO 的正确性。通过本次实验，我对 FIFO 的工作原理有了更深入的理解。

除此之外，在实现循环读写的 FIFO 模块的过程中，我实际上意识到了这是一个以双指针维护的环形队列。在实现过程中，我通过两个指针，一个指向写入位置，一个指向读取位置，来实现了循环读写的功能。这个过程中，我将程序设计课程中所学到的知识应用到了数字电路实验中，体会到了 FIFO 缓冲器与队列之间的联系。

第 6 部分 源代码

6.1 实验一：实现同步 FIFO

```
1
2  module task1_module (
3      input clk,
4      input rstn,
5      input input_valid,
6      input output_enable,
7      input [7:0] data,
8      output reg [4:0] write_addr,
9      output reg [4:0] read_addr,
10     output reg input_enable,
```

```
    output reg output_valid,
    output reg [15:0] out
);

  reg fifo_empty;
  reg fifo_full;
  reg input_enable_next = 1'b1;

  reg [7:0] mem[31:0][1:0];

  reg [1:0] write_state;
  reg [1:0] read_state;

  always @(posedge clk) begin
    if (read_addr == 31) begin
      fifo_empty   = 1'b1;
      input_enable = 1'b1;
      output_valid = 1'b0;
    end else begin
      fifo_empty   = 1'b0;
      input_enable = 1'b0;
      output_valid = 1'b1;
    end
    if (write_addr == 31 && write_state == 2'b01) begin
      fifo_full = 1'b1;
      input_enable_next <= 1'b0;
      output_valid <= 1'b1;
    end else begin
      fifo_full = 1'b0;
      input_enable = 1'b1;
      output_valid = 1'b0;
    end
  end

  always begin
    #8;
    input_enable <= input_enable_next;
  end

  always @(posedge clk or negedge rstn) begin
    if (rstn == 0) begin
      write_addr <= 5'b0;
      read_addr <= 5'b0;
      write_state <= 2'b00;
      read_state <= 2'b00;
      fifo_empty <= 1'b1;
      fifo_full <= 1'b0;
      output_valid <= 1'b0;
      input_enable <= 1'b0;
    end else begin

      if (input_valid && input_enable) begin
```

```
        if (write_state == 2'b00) begin
          mem[write_addr][0] <= data;
          write_state <= 2'b01;
        end
        if (write_state == 2'b01) begin
          mem[write_addr][1] <= data;
          write_state <= 2'b00;
          write_addr <= write_addr + 1;
        end
      end

      if (output_valid && output_enable) begin
        out <= {mem[read_addr][1], mem[read_addr][0]};
        read_addr <= read_addr + 1;
      end
    end
  end
endmodule
```

实验一 testbench 代码如下：

```

module task1_testbench ();

  reg clk, rstn;
  reg [7:0] data;
  reg input_valid, output_enable;
  wire [15:0] out;
  wire [4:0] write_addr, read_addr;
  wire input_enable, output_valid;

  task1_module task1_module_inst (
      .clk(clk),
      .rstn(rstn),
      .input_valid(input_valid),
      .output_enable(output_enable),
      .data(data),
      .out(out),
      .write_addr(write_addr),
      .read_addr(read_addr),
      .input_enable(input_enable),
      .output_valid(output_valid)
  );


  always #2 begin
    clk = ~clk;
  end

  initial begin
    clk  = 1'b0;
    rstn = 1'b1;
    #0.1 rstn = 1'b0;
```

```
    #0.2 rstn = 1'b1;
    input_valid = 1'b1;
  end

  always begin
    data = $random() % 9'b1_0000_0000;
    input_valid = 1'b1;
    output_enable = 1'b1;
    #4;
  end

  always begin
    #5;
    #6;
    #6;
    #360;
    rstn = 0;
    #4;
    rstn = 0;
    #370;
    $finish;
  end
endmodule
```

6.2 实验二：实现循环读写 FIFO

```

module task2_module (
    input clk,
    input rstn,
    input input_valid,
    input output_enable,
    input [15:0] data,
    output reg [4:0] write_addr,
    output reg [4:0] read_addr,
    output reg input_enable,
    output reg output_valid,
    output reg [7:0] out
);

  integer counter = 0;
  reg [7:0] mem[31:0][1:0];

  reg [1:0] write_state;
  reg [1:0] read_state;

  always @(posedge clk) begin
    if ((write_addr + 1) % 32 == read_addr) begin
      input_enable = 1'b0;
    end else begin
      input_enable = 1'b1;
```

```
    end
    if (write_addr == (read_addr + 1) % 32) begin
      output_valid = 1'b0;
    end else begin
      output_valid = 1'b1;
    end
  end

  always @(posedge clk or negedge rstn) begin
    if (rstn == 0) begin
      write_addr <= 5'b0;
      read_addr <= 5'b0;
      write_state <= 2'b00;
      read_state <= 2'b00;
      output_valid <= 1'b0;
      input_enable <= 1'b0;
    end else begin

      if (input_valid && input_enable) begin
        mem[write_addr][1] <= data[15:8];
        mem[write_addr][0] <= data[7:0];
        write_addr <= (write_addr + 1) % 32;
      end

      if (output_valid && output_enable) begin
        if (read_state == 2'b00) begin
          out <= mem[read_addr][0];
          read_state <= 2'b01;
        end else if (read_state == 2'b01) begin
          out <= mem[read_addr][1];
          read_state <= 2'b00;
          read_addr <= (read_addr + 1) % 32;
        end
      end
    end
  end
endmodule
```

6.3 实验三：控制 FIFO 读写频率

```

module task2_testbench ();

  reg clk, rstn;
  reg [15:0] data;
  reg input_valid, output_enable;
  wire [7:0] out;
  reg read, write;
  wire [4:0] write_addr, read_addr;
  wire input_enable, output_valid;

```

```
task2_module task2_module_inst (
    .clk(clk),
    .rstn(rstn),
    .input_valid(input_valid),
    .output_enable(output_enable),
    .data(data),
    .out(out),
    .write_addr(write_addr),
    .read_addr(read_addr),
    .input_enable(input_enable),
    .output_valid(output_valid)
);


always #2 begin
  clk = ~clk;
end

integer counter = 0;
initial begin
  clk = 1'b0;
  rstn = 1'b1;
  write = 1'b0;
  read = 1'b0;
  input_valid = 1'b0;
  output_enable = 1'b0;
  #1 rstn = 1'b0;
  #2 rstn = 1'b1;
  #5;
end

always begin
  input_valid = ~input_valid;
  #4;
end

always begin
  output_enable = ~output_enable;
  #6;
end

always begin
  #4;
  data[7:0]  = $random() % 9'b100000000;
  data[15:8] = $random() % 9'b100000000;
end

always begin
  #5;
  write = 1'b1;
  #6;
  write = 1'b0;
```

```
64      read  = 1'b1;
65      #6;
66      write = 1'b1;
67      read  = 1'b0;
68      #600;
69      $finish;
70    end
71  endmodule
```