计算机组成原理实验 实验报告

实验题目：Lab2 寄存器堆与队列

学生姓名：王章瀚

学生学号：PB18111697

完成日期：2020 年 5 月 13 日

计算机实验教学中心制

2019年09月

1 实验题目

Lab2 寄存器堆与队列

2 实验目的

1. 掌握寄存器堆（Register File）和存储器（Memory）的功能、时序及其应用；
2. 熟练掌握数据通路和控制器的设计和描述方法。

3 实验平台

Vivado

4 实验过程

4.1 寄存器堆

这里寄存器堆的要求是: 该寄存器堆含有32 个寄存器(r0 r31，其中r0的内容恒定为零), 寄存器的位宽由参数WIDTH指定, 具有2个异步读端口和1个同步写端口。 因此只需要维护一个这样的寄存器数组:

```
reg [WIDTH-1: 0] registers[SIZE-1: 0];
```

然后在we使能的时候对寄存器进行相应修改, 并固定0地址输出为0即可. 代码相对简单, 直接贴上.

```
module register_file
    #(
    parameter WIDTH = 32,    // 数据宽度
    parameter ADDR_WIDTH = 5, // 地址长度
    parameter SIZE = {1'b1, {(ADDR_WIDTH){1'b0}}} // 地址数量
    )
    (
    input clk,      // 时钟(上升沿有效)
    input [ADDR_WIDTH-1: 0] ra0,    // 读端口0地址
    output [WIDTH-1: 0] rd0,     //读端口0数据
    input [ADDR_WIDTH-1: 0] ra1,    // 读端口1地址
    output [WIDTH-1: 0] rd1,     //读端口1数据
    input [ADDR_WIDTH-1: 0] wa,   // 写端口位置
    input we,    // 写使能(高电平有效)
    input [WIDTH-1: 0] wd    // 写端口数据
    );

    reg [WIDTH-1: 0] registers[SIZE-1: 0];

    assign rd0 = ra0 == {ADDR_WIDTH{1'b0}} ? {WIDTH{1'b0}} : registers[ra0]; // 保证了0地址输出为0
    assign rd1 = ra1 == {ADDR_WIDTH{1'b0}} ? {WIDTH{1'b0}} : registers[ra1]; // 保证了0地址输出为0

    always @(posedge clk) begin
        if(we) begin
            if(wa != {ADDR_WIDTH{1'b0}}) begin
```

```
                registers[wa] <= wd;
            end
        end
    end

endmodule
```

4.2 存储器

4.2.1 行为方式描述存储器

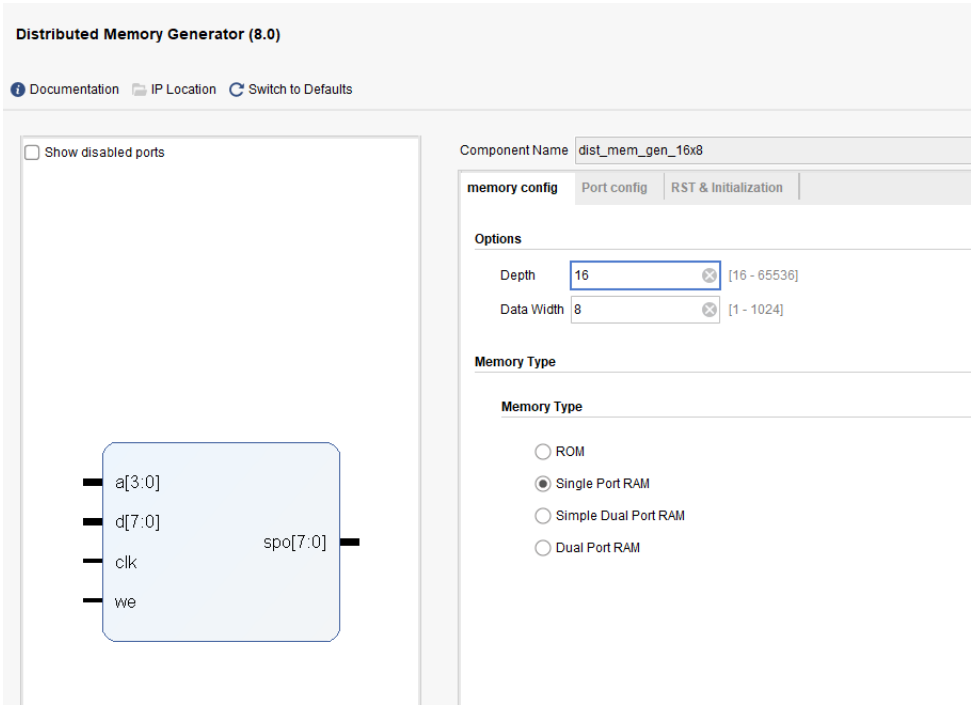
这里填补了老师留的空. 由于不是实验步骤要求的, 这里只是提一下.

```
module  ram_16x8                //16x8位单端口RAM
    (
    input   clk,                //时钟（上升沿有效）
    input en, we,                   //使能，写使能
    input [3: 0]  addr, //地址
    input [7: 0]  din,        //输入数据
    output [7: 0]  dout //输出数据
    );
    reg [3: 0] addr_reg;
    reg [7: 0] mem[15: 0];

    //初始化RAM的内容
    initial
    $readmemh("init.mem", mem);

    assign dout = mem[addr_reg];

    always@(posedge clk) begin
        if(en) begin
            addr_reg <= addr;
            if(we)
                mem[addr] <= din;
        end
    end
endmodule
```

4.2.2 IP核例化存储器

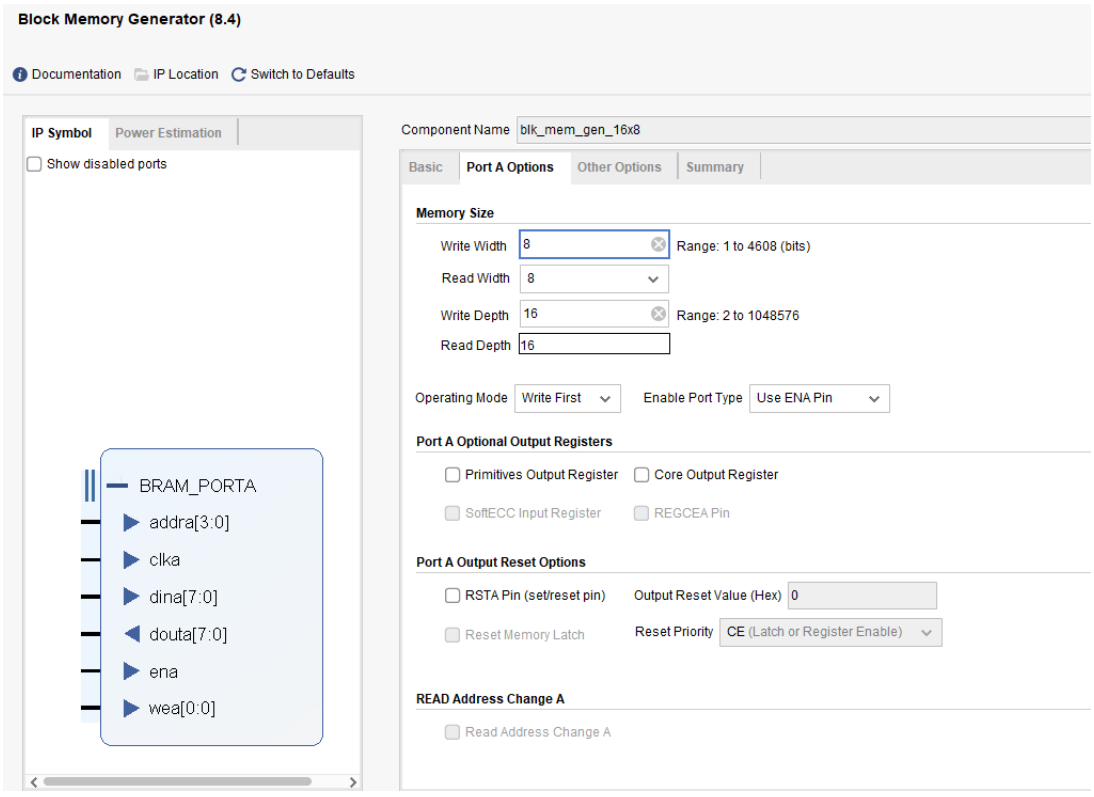
1. 分布式存储器(Distributed Memory Generator)

   修改相应 Depth 和 Data Width 等即可.



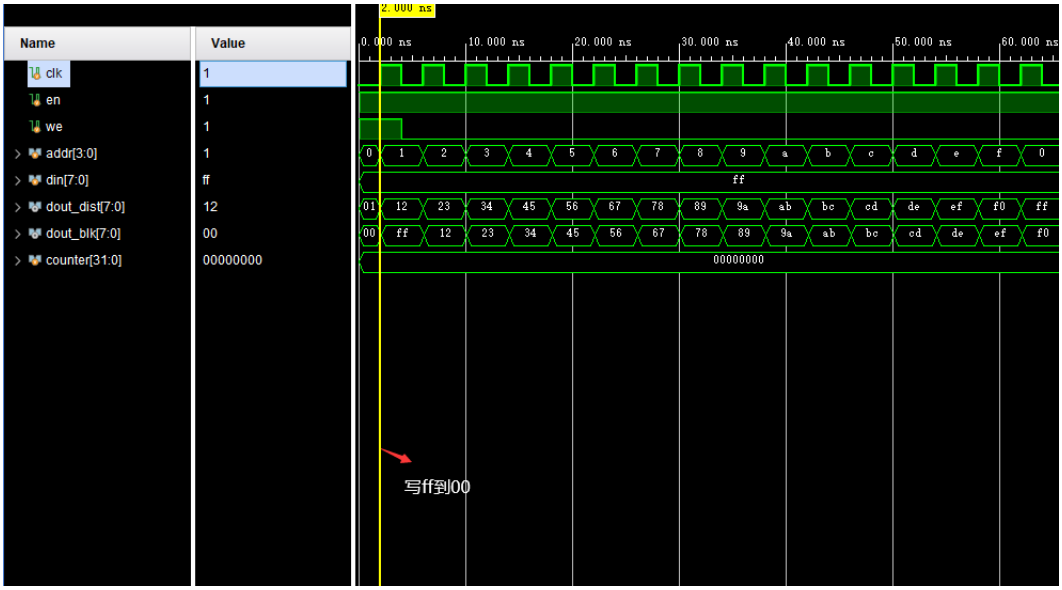
2. 块式存储器(Block Memory Generator)

   修改相应 Write Width, Write Depth 和 Read Width, Read Depth 等即可.



3. 功能仿真结果与对比

仿真结果见下图, dout_dist和dout_blk分别是分布式存储器和块式存储器的仿真输出结果. 可见写的时候, 二者都是同步写入(图中标识了写ff到地址00上的时刻). 而读的时候, 分布式存储器是异步读, 而块式的则是同步读(正是因此, 块式的输出结果总是落后一个周期).



至于其他方面的对比, 大致可见下表

	分布式存储器	块式存储器
读的方式	异步	同步
写的方式	同步	同步
物理资源使用	CLB的LUT	独立的专门存储器
适用情形	小存储器	大存储器

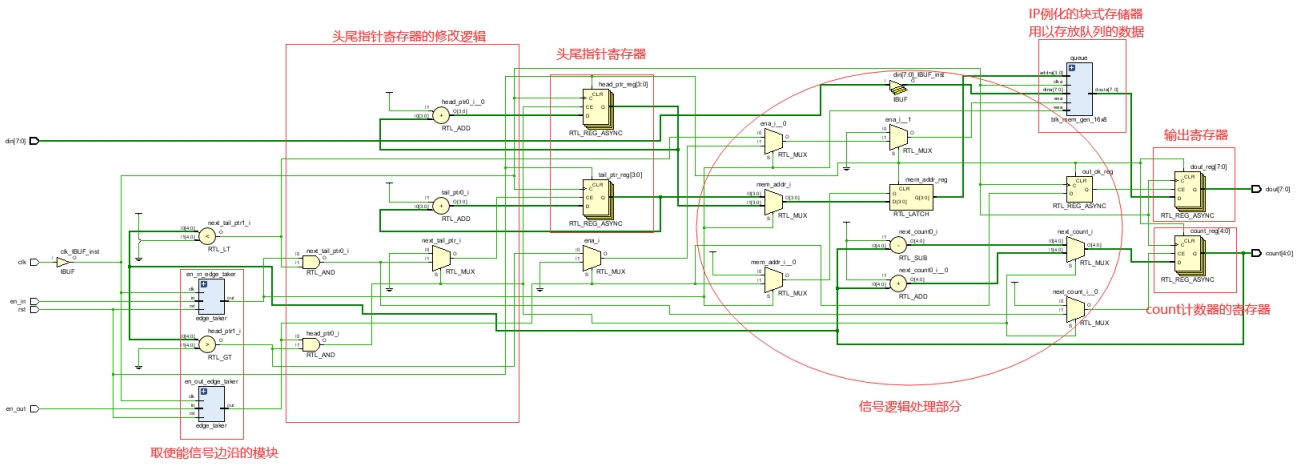
4.3 FIFO队列电路

4.3.1 基本过程

这里采用一个小状态机来实现. 状态由两部分组成: head_ptr(队列头指针)和tail_ptr(队列尾指针). 然后每当有入队或出队的使能信号, 就准备好数据送入块式存储器, 待从块式存储器中取出数据之后, 就输出出来. 这其中, 对使能信号做了取边沿处理, 以防使能周期过长.

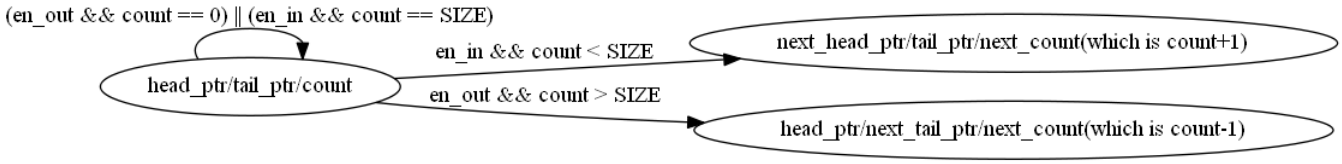
4.3.2 数据通路

这里数据通路较为复杂, 故直接用Vivado的RTL分析得到的结果来分析. 各个模块功能已经标注如下图



4.3.3 状态机控制

这里状态机若画完整, 会重复很多单元, 所以只是画一个示例:



4.3.4 代码讲解

1. 取边沿模块

老师给出的取边沿模块略有问题, 故采用上学期模拟与数字电路实验用的取边沿模块.

```
module edge_taker
    #(parameter N = 1)
    (
    input clk, rst, // 时钟(上升沿有效), 复位(异步复位, 高电平有效)
    input [N-1: 0]in,    // 输入信号
    output [N-1: 0] out    // 输出信号
    );

    reg [N-1: 0] in1 = 0;
    reg [N-1: 0] in2 = 0;
    always@(posedge clk) in1 <= in;
    always@(posedge clk) in2 <= in1;
    assign out = rst ? {N{1'b0}} : in1 & ~in2;
endmodule

```

2. FIFO队列模块

代码各个部分的功能已经注释清楚.

其他一些值得一提的有:

(a) 如果指针到了最后一个地址, 就需要换到0地址处, 这里需要稍作处理.

(b) count会在入队时+1, 出队时-1.

(c) 当且仅当有使能信号的时候, next_head_ptr和next_tail_ptr才会是+1的状态.

(d) 其中取边沿模块可以避免一个使能信号内多次出入队列.

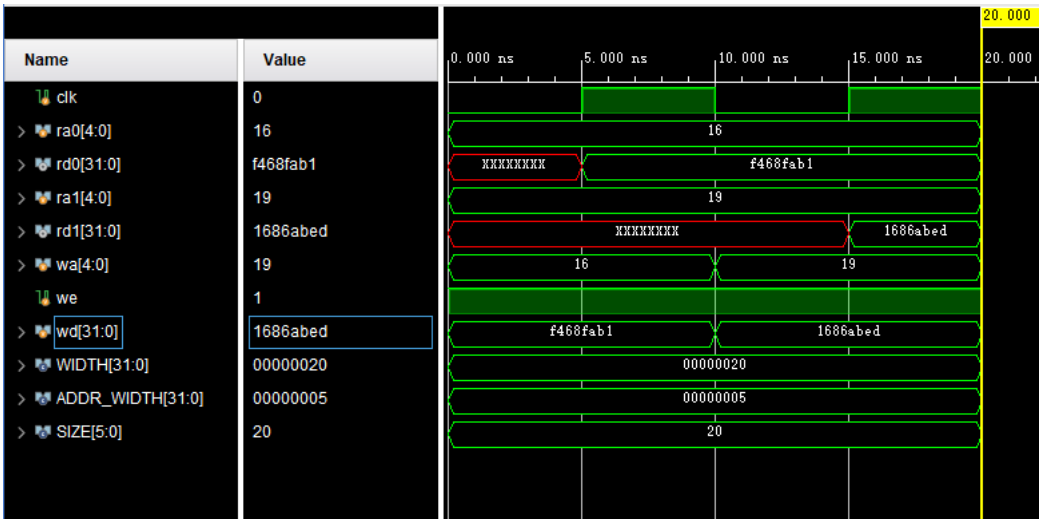
```
module fifo
    #(
    parameter WIDTH = 8,
    parameter ADDR_WIDTH = 4,
    parameter SIZE = {1'b1, {ADDR_WIDTH{1'b0}}}
    )
    (
    input clk, rst,      //时钟（上升沿有效）、异步复位（高电平有效）
    input [WIDTH-1: 0] din,      //入队列数据
    input en_in,         //入队列使能，高电平有效
    input en_out,        //出队列使能，高电平有效
    output reg [WIDTH-1: 0] dout,    //出队列数据
    output reg [ADDR_WIDTH: 0] count     //队列数据计数
    );

    reg [ADDR_WIDTH-1: 0] head_ptr;
    reg [ADDR_WIDTH-1: 0] tail_ptr;
    reg [ADDR_WIDTH-1: 0] next_head_ptr;
    reg [ADDR_WIDTH-1: 0] next_tail_ptr;
    reg [ADDR_WIDTH-1: 0] mem_addr;
    reg [ADDR_WIDTH: 0] next_count;
    reg ena;
    wire [WIDTH-1: 0] mem_out;
    wire en_in_edge, en_out_edge;
    reg out_ok;

    // 数据通路
    //// 块内存
    blk_mem_gen_16x8 queue(.addra({mem_addr}), .clka(clk), .dina(din), .douta(mem_out), .ena(
    ena), .wea(en_in_edge));
    //// 取使能信号边缘
    edge_taker #(.N(1)) en_in_edge_taker(.in(en_in), .clk(clk), .rst(rst), .out(en_in_edge));
    edge_taker #(.N(1)) en_out_edge_taker(.in(en_out), .clk(clk), .rst(rst), .out(en_out_edge
    ));

    // 状态机状态转移
    always @(posedge clk, posedge rst) begin
        if(rst) begin
            head_ptr <= {ADDR_WIDTH{1'b0}};
            tail_ptr <= {ADDR_WIDTH{1'b0}};
            next_head_ptr <= {ADDR_WIDTH{1'b0}};
            next_tail_ptr <= {ADDR_WIDTH{1'b0}};
            count <= {(ADDR_WIDTH+1){1'b0}};
            dout <= {WIDTH{1'b0}};
            out_ok <= 1'b0;
        end
        else begin
            head_ptr <= next_head_ptr;
```

```
47              tail_ptr <= next_tail_ptr;
48              count <= next_count;
49              out_ok <= en_out_edge;
50              if(out_ok) dout <= mem_out;
51          end
52      end
53
54      // 状态描述
55      always @(*) begin
56          if(rst) begin
57              mem_addr = {ADDR_WIDTH{1'b0}};
58              ena = 1'b0;
59          end
60          else begin
61              // head_ptr的状态控制
62              if(en_out_edge && (count > 0)) begin
63                  next_head_ptr = head_ptr == SIZE ? 0 : head_ptr + 1;
64                  next_tail_ptr = tail_ptr;
65                  next_count = count - 1;
66                  mem_addr = head_ptr;
67                  ena = 1'b1;
68              end
69              // tail_ptr的状态控制
70              else if(en_in_edge && (count < SIZE)) begin
71                  next_head_ptr = head_ptr;
72                  next_tail_ptr = tail_ptr == SIZE ? 0 : tail_ptr + 1;
73                  next_count = count + 1;
74                  mem_addr = tail_ptr;
75                  ena = 1'b1;
76              end
77              else begin
78                  next_head_ptr = head_ptr;
79                  next_tail_ptr = tail_ptr;
80                  next_count = count;
81                  ena = 1'b0;
82                  mem_addr = mem_addr;
83              end
84          end
85      end
86
87  endmodule
88
```

5 实验结果

5.1 寄存器堆

这是个简单的仿真测试. 这里寄存器刚开始读出来是x, 而后进行写入, 则异步读出的结果相应改变. 如下图.



因此可以认为, 该寄存器文件模块的两个异步读和一个同步写都能够正常工作.

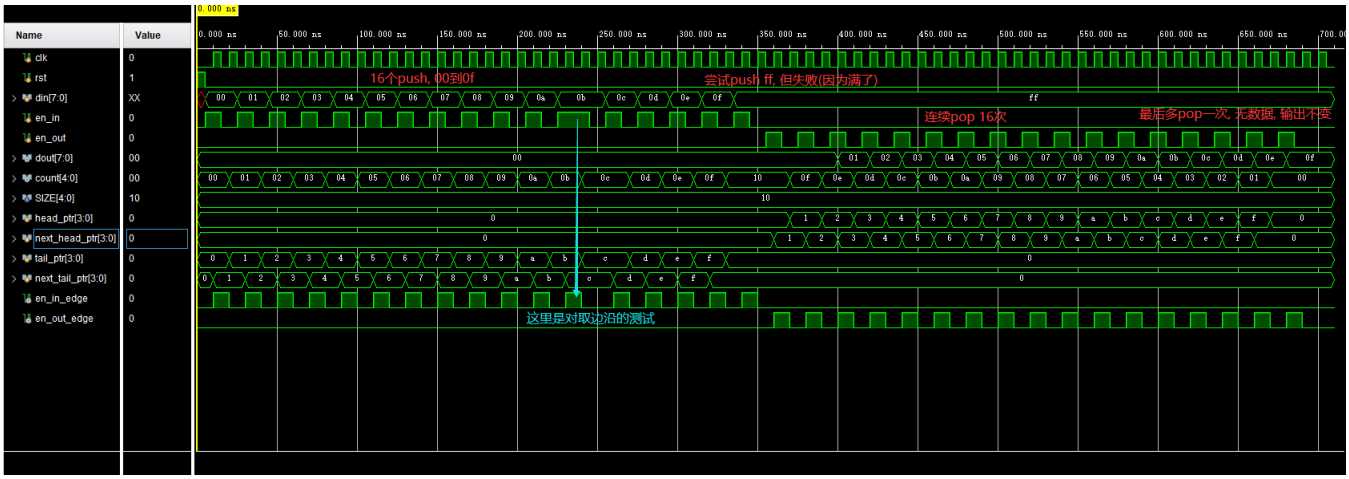
5.2 存储器

该部分的实验结果已经在前面实验过程中讲述, 这里就不在重复描述了.

5.3 FIFO队列

下图清晰地展示了仿真过程及正确性.

这个仿真先后做了这些操作: push 00到0f, push ff(但由于队列大小只有2, 故失败), pop 16次(按顺序出00到0f), pop(已经没有数据, 输出不变). 此外, 图中蓝色标出了一个取边沿的示例.



6 思考题

6.1 如何利用寄存器堆和适当电路设计实现可变个数的数据排序电路？

对于给定排序数据的起始地址和终止地址(这里假设顺序存储了待排数据), 可以采用冒泡排序, 两两比较和交换(类似于Lab1). 和Lab1所不同的是, Lab1的状态是程序开始前就知道的, 而这里的状态是可变的, 但此时只需要维护两个指针(类似于冒泡排序时, 两层循环分别需要的循环变量), 并依据这两个指针来控制状态转移, 即可完成可变个数的数据排序电路.

7 心得体会

本次实验让我更深刻地理解了 Distributed Memory 和 Block Memory 的区别, 并初次理解了怎么使用Verilog描述一个FIFO的过程.

8 意见建议

这次实验的FIFO的对应实验要求描述并不是特别清楚. 例如, 应当在输入之后几个周期内给出结果等. 这对我实现造成了一定困扰.