# 实验二 寄存器堆与队列

## 一. 实验目标

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。

## 二. 实验内容

1. 寄存器堆

设计参数化的寄存器堆，其逻辑符号如图-1所示。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。

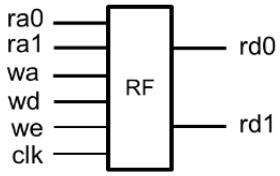



图-1 寄存器堆逻辑符号

参数化的寄存器堆端口声明如下：

```
module register_file        //32 x WIDTH寄存器堆

\#(parameter WIDTH = 32)  //数据宽度

(clk,           //时钟（上升沿有效）

input [4:0] ra0,         //读端口0地址

output [WIDTH-1:0] rd0, //读端口0数据

input [4:0] ra1,         //读端口1地址

output [WIDTH-1:0] rd1, //读端口1数据

input [4:0] wa,          //写端口地址

input we,          //写使能，高电平有效

input [WIDTH-1:0] wd     //写端口数据

);

……
```

```
endmodule
```

2. 存储器

存储器与寄存器堆的功能类似，都是用于存储信息，只是存储器的容量更大，配置方式更多，例如ROM/RAM、单端口/简单双端口/真正双端口、分布式/块式等方式。设计存储器可以通过行为方式描述，也可以通过IP例化方式实现。

例如，设计一容量为16 x 8位（即深度DEPTH：16，宽度WIDTH：8）的单端口RAM，其逻辑符号如图-2所示。用行为方式描述的Verilog代码如下：（请补充代码中空缺的参数）

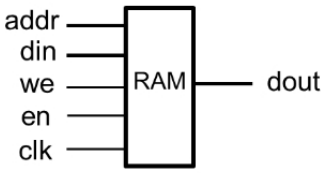



图-2 单端口RAM逻辑符号

```
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(“初始化数据文件名”, mem);
assign dout = mem[addr_reg];
always@(posedge clk) begin
 if(en) begin
  addr_reg <= addr;
  if(we)
   mem[addr] <= din;
 end
end
endmodule
```

Verilog HDL程序可以利用两个系统任务$readmemb和$readmemh，从文件中读取数据来初始化存储器。其语法格式（以$readmemb为例，$readmemh类似）如下：

$readmemb("<数据文件名>",<存储器名>,<起始地址>,<终止地址>);

数据文件是文本格式，只能包含空白（空格、换行、制表格tab）、注释和二进制（对于$readmemb）或十六进制数据（$readmemh），数据中可以有不定值x或X、高阻值z或Z、或者下画线，但不能包含位宽书名和格式说明。起始地址和结束地址是可选的。当地址出现在数据文件中时，其格式是字符“@”后跟上十六进制数据，例如：@hhhh。当读取中遇到地址说明符，会将地址后的数据存

放到相应的地址中。

例如，从文件“rx.vec”中读取的第一个数字被存储在地址15中，下一个存储在地址16，并以此类推直到地址30。

$readmemb("rx.vec", MemA, 15, 30);

例如，文件init.vec内容如下：

```
@002
11111111  01010101
00000000 10101010
@006
1111zzzz  00001111
```

Verilog程序如下：

```
reg [7:0] mem[0:7];

initial $readmemb("init.vec", mem);
```

则存储器中的内容如下：

```
mem[0]=xxxxxxxx;

mem[1]=xxxxxxxx;

mem[2]=11111111;

mem[3]=01010101;

mem[4]=00000000;

mem[5]=10101010;

mem[6]=1111zzzz;

mem[7]=00001111;
```

通过Vivado例化存储器IP的配置界面如图-3~8所示。配置时可以指定COE文件对存储器的内容初始化。COE文件包含两个分号结束的参数：memory_initialization_radix和memory_initialization_vector。前者说明数据进制，可以是2、10、16；后者列举出由空格或逗号分隔数据序列。

例如，初始化一个32x16 ROM的COE文件如下：

```
memory_initialization_radix = 16;
memory_initialization_vector =
23f4, 0721, 11ff, ABe1, 0001, 1, 0A, 0,
23f4, 0721, 11ff, ABe1, 0001, 1, 0A, 0,
23f4, 721, 11ff, ABe1, 0001, 1, A, 0,
23f4, 721, 11ff, ABe1, 0001, 1, A, 0;
```

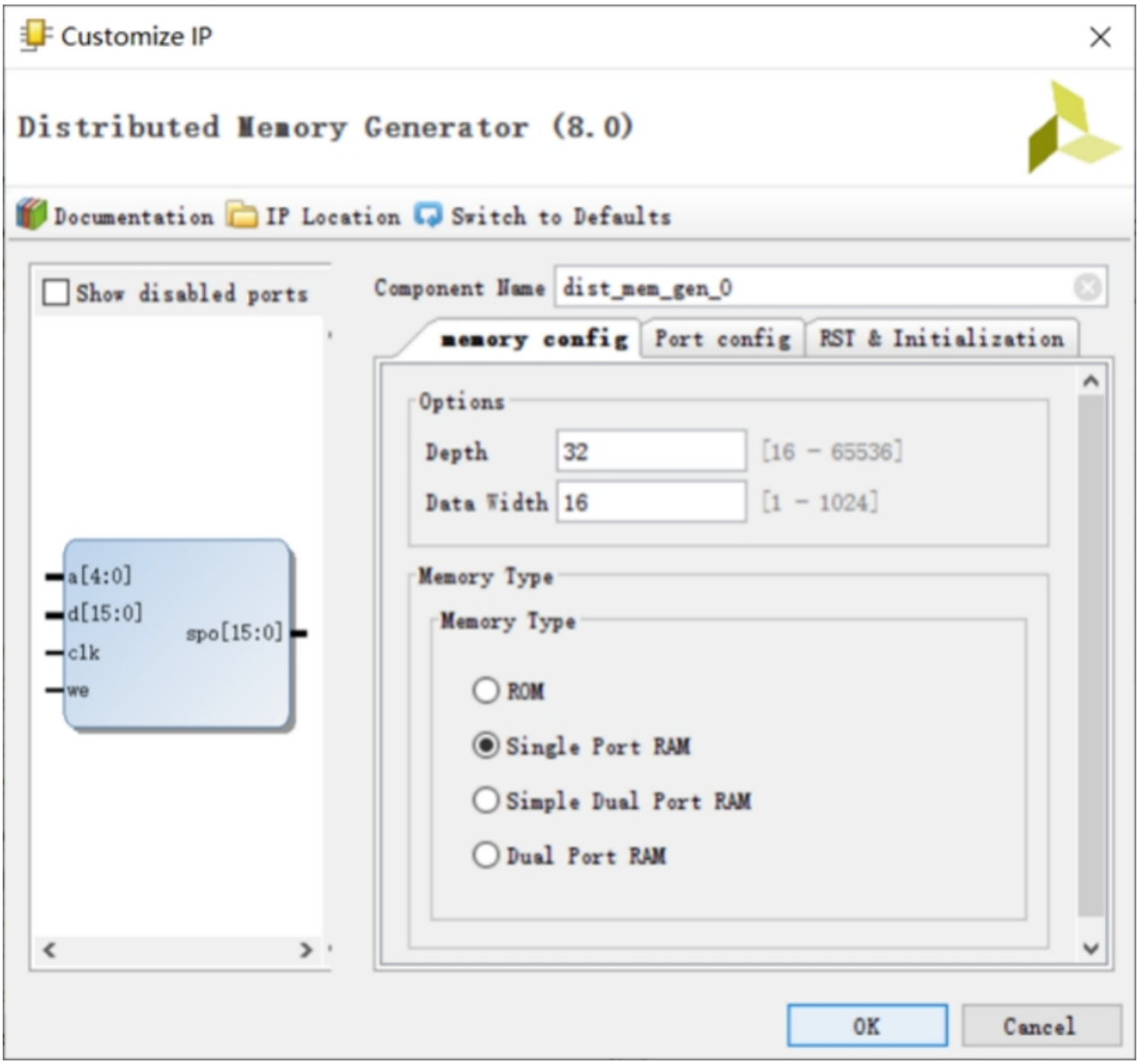



图-3 分布式存储器例化界面—存储器配置

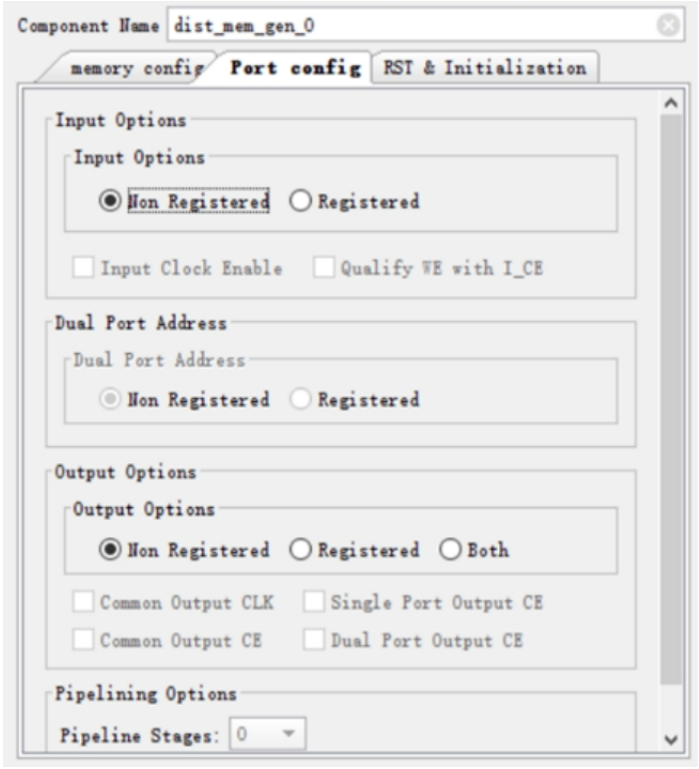



图-4 分布式存储器例化界面—端口配置

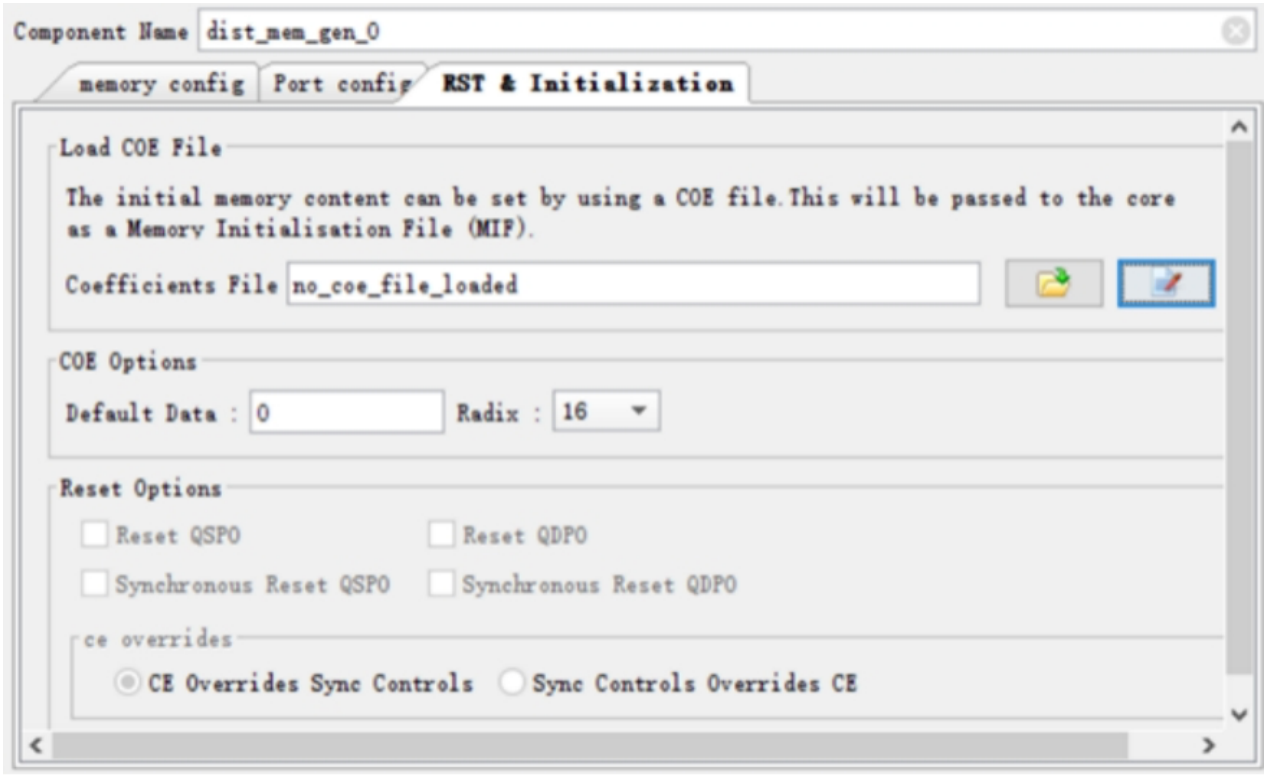



图-5 分布式存储器例化界面—复位和初始化

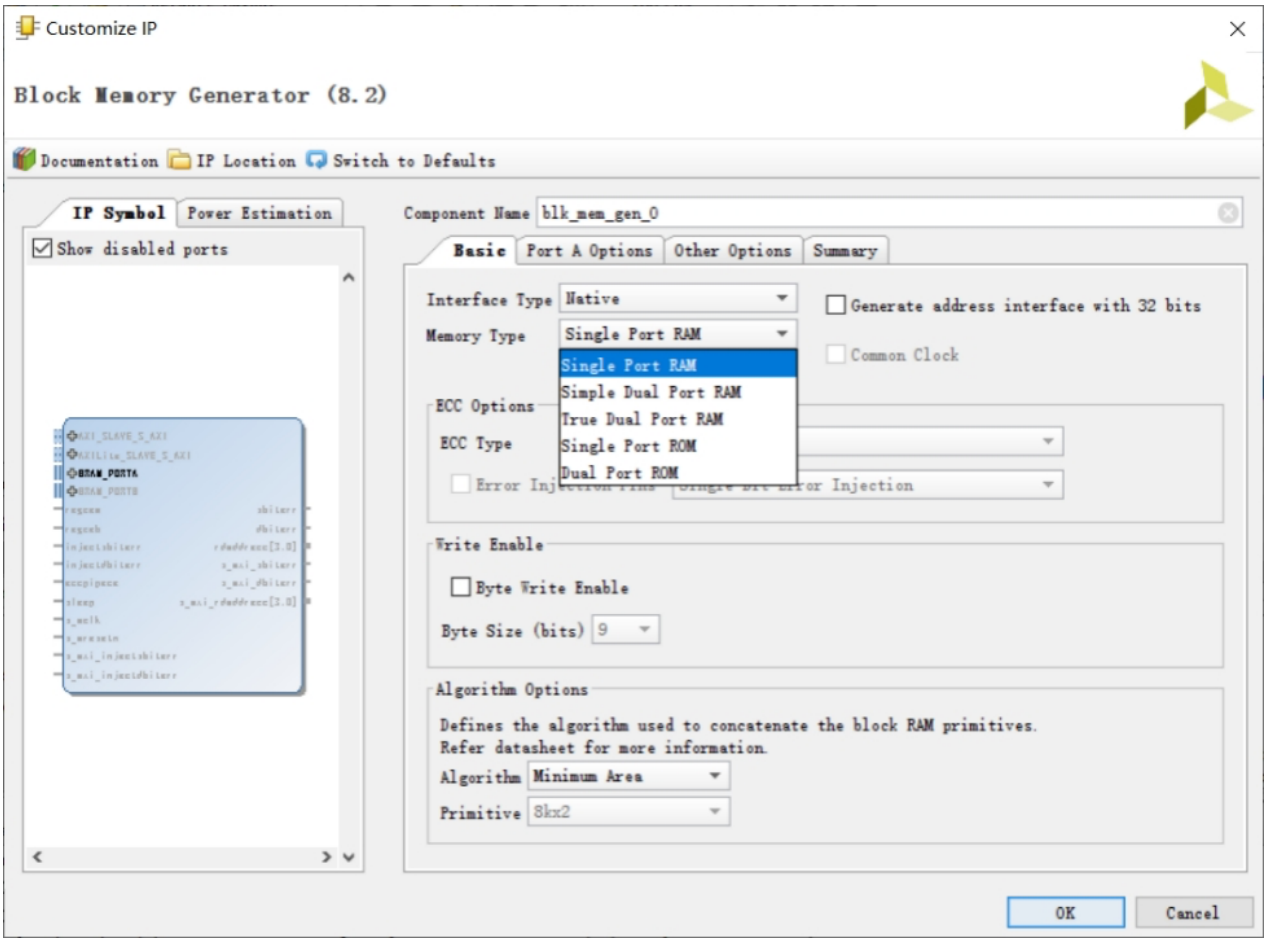



图-6 块式存储器例化界面—基本

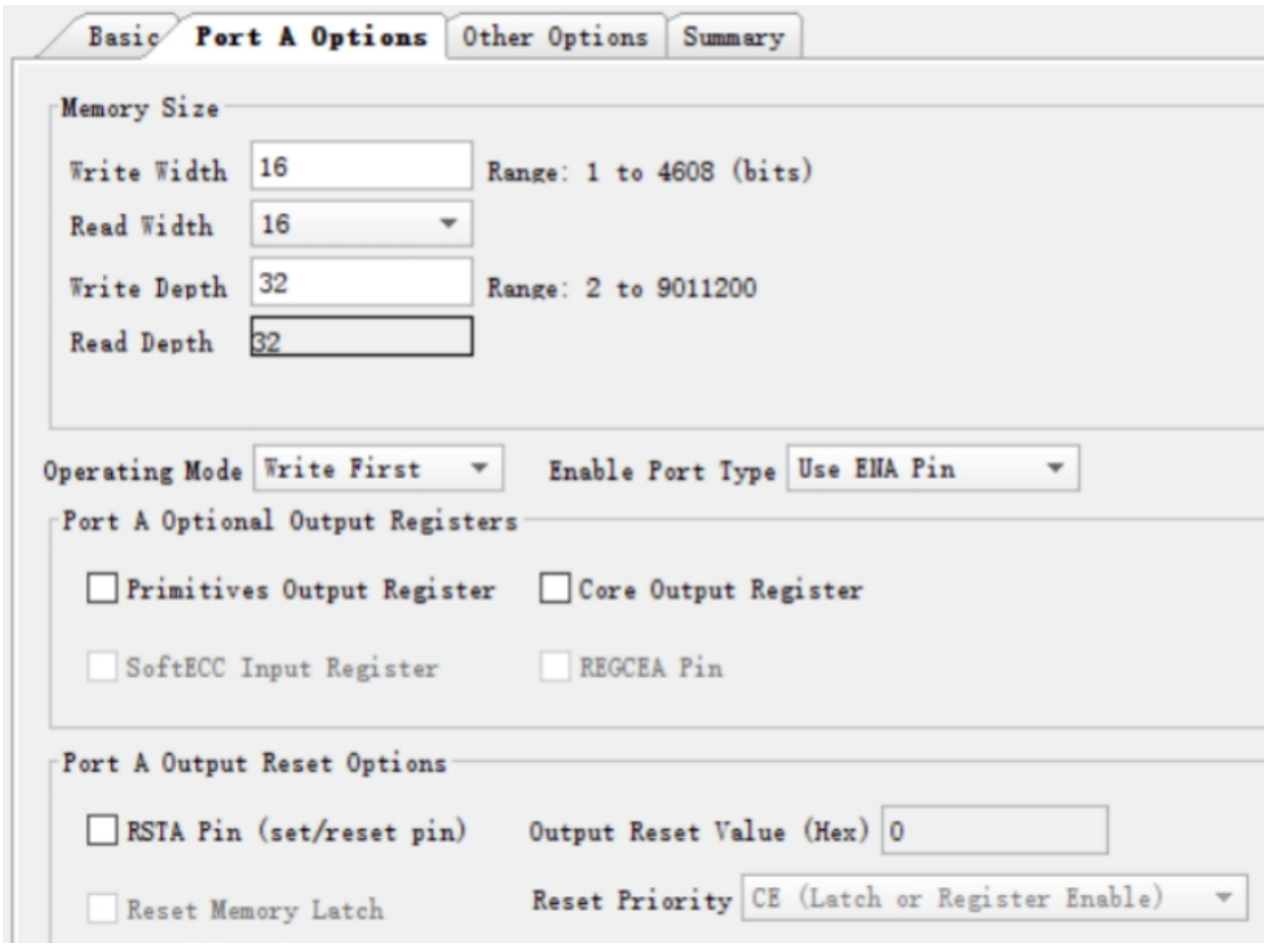


图-7 块式存储器例化界面—端口选项

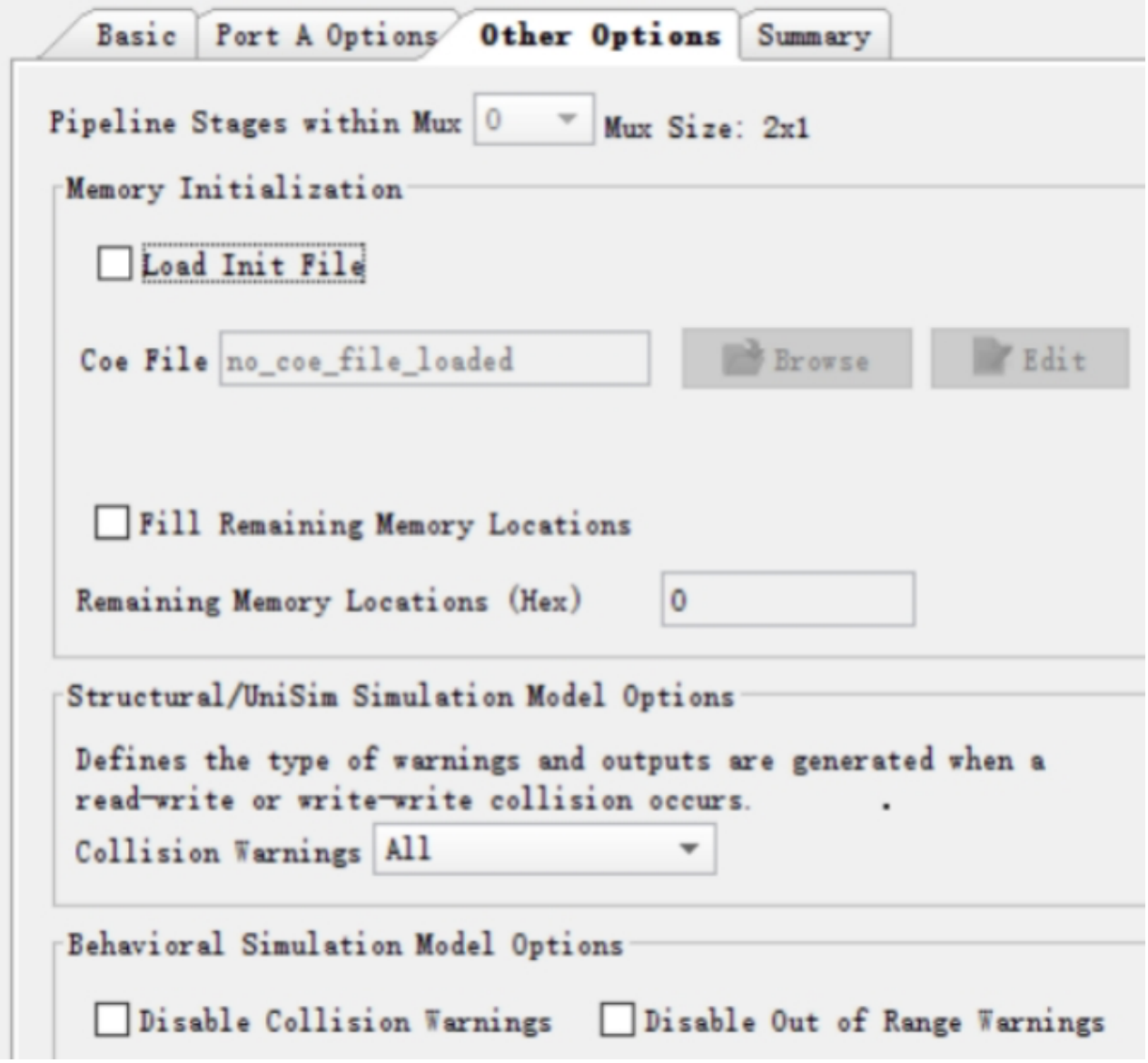



图-8 块式存储器例化界面—其他选项

3. 先进先出（FIFO）队列

利用例化的存储器IP（16 x 8位块式的单端口RAM）和适当的逻辑电路，设计实现数据宽度为8位、最大长度为16的FIFO队列，其逻辑符号如图-9所示。入队列使能（en_in）有效时，将输入数据（din）加入队尾；出队列使能（en_out）有效时，将队列头数据输出（dout）。队列数据计数（count）指示队列中有效数据个数。当队列满（count = 16）时不能执行入队操作，队列空（count = 0）时不能进行出队操作。在入对使能信号的一次有效持续期间，仅允许最多入队一个数据，出队操作类似。

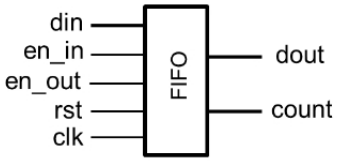



图-9 FIFO队列逻辑符号

该FIFO队列模块端口声明如下：

module fifo

(input clk, rst, //时钟（上升沿有效）、异步复位（高电平有效）

input [7:0] 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 [7:0] dout, //出队列数据

output [4:0] count //队列数据计数

);

……

endmodule

## 三. 实验步骤

1. 采用行为方式描述参数化的寄存器堆，并进行功能仿真；
2. 采用IP例化方式分别实现分布式和块式的16 x 8位单端口RAM，并进行功能仿真和对比；
3. 设计FIFO队列电路的数据通路和控制器，采用结构化方式描述数据通路，两段式FSM描述控制器，并进行功能仿真；
4. 将FIFO队列电路下载至FPGA中测试：din对应SW7 ~ SW0，dout对应LED7 ~ LED0，count对应LED15~LED11，en_in对应BTNU，en_out对应BTND，clk对应100 MHz时钟（clk100mhz），rst对应BTNL。

## 四. 实验检查

1. 检查寄存器堆和RAM的功能仿真；
2. 检查FIFO队列电路的功能仿真；
3. 检查FIFO队列电路下载到FPGA后的运行功能。

## 五、实验过程

1. 寄存器堆

遇到的bug：

- 设置mem数组，应该设为[31:0] 而不是 [4:0]
- `register_file #(4) Register` 而不是 `register_file Register #(4)`
- `$readmemb` 的路径与Windows的斜杠不同，应该使用'/'
- txt文件的初始化格式我可能写的有问题，使用讲义里教的.vec格式就没问题了

核心代码：其中初始化文件是0~15 / 16~31 分别赋值为0~F / 0~F

```
/*module signal_edge(
    input clk,
    input button,
    output button_edge);
reg button_r1,button_r2;
always@(posedge clk)
```

```
    button_r1 <= button;
always@(posedge clk)
    button_r2 <= button_r1;
assign button_redge = button_r1 & (~button_r2);
endmodule*/

module register_file        //32 x WIDTH寄存器堆
#(parameter WIDTH = 4)          //数据宽度
(
    input clk,             //时钟（上升沿有效）
    input [4:0] ra0,         //读端口0地址
    output reg [WIDTH-1:0] rd0,          //读端口0数据
    input [4:0] ra1,         //读端口1地址
    output reg [WIDTH-1:0] rd1,        //读端口1数据
    input [4:0] wa,          //写端口地址
    input we,              //写使能，高电平有效
    input [WIDTH-1:0] wd         //写端口数据
);

reg [WIDTH-1:0] mem [31:0];
// 初始化 RAM 的内容
initial
begin
    //$readmemh("C:/Users/mi/Desktop/text.txt", mem, 0, 31);
    $readmemb("C:/Users/mi/Desktop/init.vec", mem, 0, 31);
end

// 异步读
always@(*)
begin
    rd0 = mem[ra0];
    rd1 = mem[ra1];
end

// 取信号边沿
//wire we_edge;
//signal_edge CLK(.clk(clk), .button(we), .button_edge(we_edge));

// 同步写
always@(posedge clk)
begin
    if(we)
        mem[wa] <= wd;
end
endmodule
```

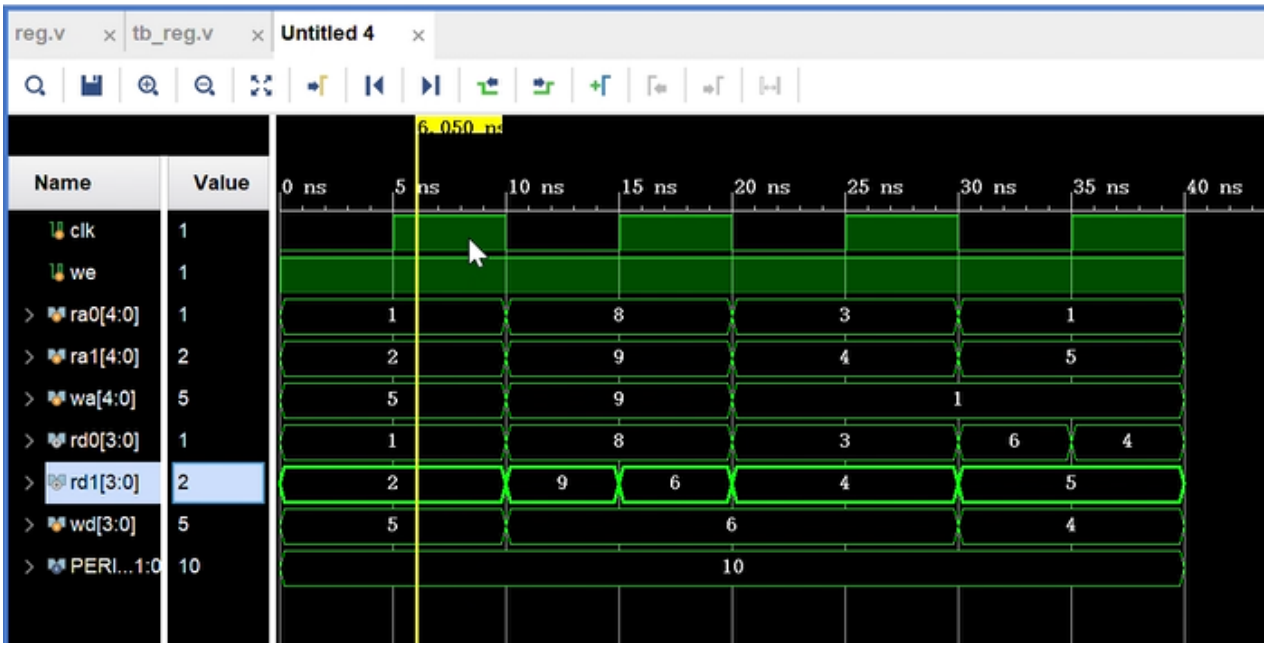
仿真代码：

```
module tb();
```

```
reg clk, we;
reg [4:0] ra0, ra1, wa;
wire [3:0] rd0, rd1;
reg [3:0] wd;

register_file #(4) Register
    (.clk(clk), .ra0(ra0), .ra1(ra1),
    .rd0(rd0), .rd1(rd1), .wa(wa), .we(we), .wd(wd));

parameter PERIOD = 10;

initial
begin
    clk = 0;
    repeat (8)
        #(PERIOD/2) clk = ~clk;
    $finish;
end

initial
begin

    ra0 = 1;
    ra1 = 2;
    we = 1;
    wa = 5;
    wd = 5;

    # (PERIOD);
    ra0 = 8;
    ra1 = 9;
    we = 1;
    wa = 9;
    wd = 6;

    # (PERIOD);
    ra0 = 3;
    ra1 = 4;
    we = 1;
    wa = 1;
    wd = 6;

    # (PERIOD);
    ra0 = 1;
    ra1 = 5;
    we = 1;
    wa = 1;
    wd = 4;
end
```

```
endmodule
```

结果如图：




结果全部符合预期。

2. IP例化分布式和块式的16 x 8位单端口RAM

代码：

```
module IP(
input [4:0] a,
input [15:0] wd,
input clk,
input we,en,
output [15:0] rdD, rdB
);

dist_mem_gen_0 dist0 (
  .a(a),      // input wire [4 : 0]  地址
  .d(wd),      // input wire [15 : 0] d 写数据
  .clk(clk),  // input wire clk
  .we(we),    // input wire we 写使能
  .spo(rdD)  // output wire [15 : 0] spo 读数据
);

blk_mem_gen_0 dist1 (
  .clka(clk),    // input wire clka
  .ena(en),      // input wire ena 总使能
  .wea(we),      // input wire [0 : 0] wea 写使能
  .addra(a),  // input wire [4 : 0] addra 地址
  .dina(wd),    // input wire [15 : 0] dina 写数据
  .douta(rdB)  // output wire [15 : 0] douta 读数据
);
```

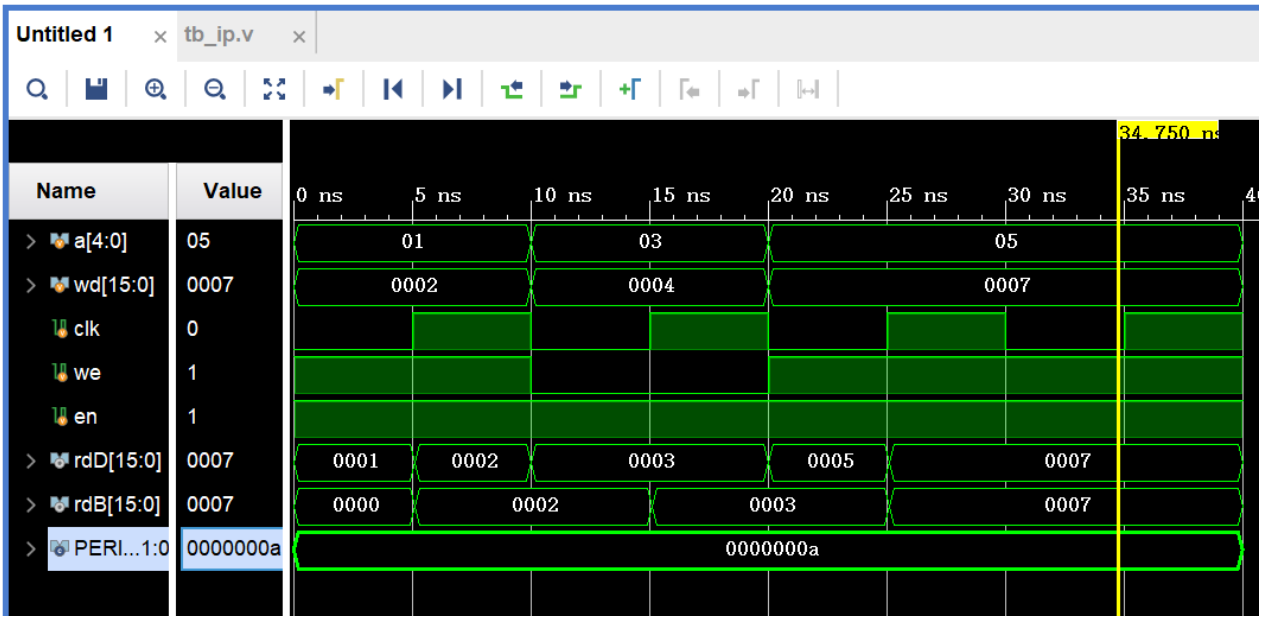
```
endmodule
```

仿真代码：

```
module tb();

reg [4:0] a;
reg [15:0] wd;
reg clk;
reg we, en;
wire [15:0] rdD, rdB;

IP test(
    .a(a),
    .wd(wd),
    .clk(clk),
    .we(we),
    .en(en),
    .rdD(rdD),
    .rdB(rdB)
);

parameter PERIOD = 10;

initial
begin
    clk = 0;
    repeat (8)
        #(PERIOD/2) clk = ~clk;
    $finish;
end

initial
begin
    a = 1;
    wd = 2;
    we = 1;
    en = 1;

    # (PERIOD);
    a = 3;
    wd = 4;
    we = 0;
    en = 1;

    # (PERIOD);
    a = 5;
    wd = 7;
```

```
    we = 1;
    en = 1;

end
endmodule
```

结果如图：




分析得，

DRAM：异步读（10ns），同步写（25ns）

BRAM：同步读（10ns），同步写（25ns）

3. FIFO队列电路

考虑了取边沿导致的延迟以及上板子的真实情况。

核心代码：

```
module signal_edge(
    input y,rst,clk,
    output p
);

    localparam S0 = 0;
    localparam S1 = 1;
    localparam S2 = 2;

    reg [1:0]state,next_state;
    //output logic
    assign  p = (state==S1);
    //state logic
    always @(posedge clk, posedge rst)
        if (rst) state <= S0;
```

```
        else state <= next_state;
    //next state logic
    always @(*)
    begin
        next_state = state;
        case (state)
            S0: if (y) next_state = S1;
            S1:begin if (y) next_state = S2;
                     else next_state = S0;
                end
            S2: if (!y) next_state = S0;
            default: next_state = S0;
        endcase
    end
endmodule

module fifo
(
    input clk, rst,        //时钟（上升沿有效）、异步复位（高电平有效）
    input [7:0] 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 [7:0] dout,  //出队列数据
    output reg [4:0] count  //队列数据计数
);
    reg [1:0] state,next_state;
    reg [3:0] add_r, add_w, add; //读地址，写地址，最终用到地址
    wire p1,p2;
    reg en_r,en_w; //读使能和写使能
    wire [7:0]out;

    localparam NULL = 0;
    localparam MIDDLE = 1;
    localparam FULL = 2;

    //DATA Path
    signal_edge EDG1(.y(en_in), .rst(rst), .clk(clk), .p(p1));
    signal_edge EDG2(.y(en_out), .rst(rst), .clk(clk), .p(p2));
    dist_mem_gen_0 FIFO(.a(add), .d(din), .clk(clk), .we(en_w), .spo(out));

    //Control Unit
    always @(posedge clk , posedge rst)
    begin
        if(rst) state <= NULL;
        else state <= next_state;
    end
    always @(*)
    begin
        if(rst)
```

```
            {count, add_r, add_w} = 0;
        en_r=0;
        en_w=0;
        next_state = state;
        case(state)
            NULL:
                if(p1)
                begin
                    en_w = 1;
                    next_state = MIDDLE;
                    add = add_w;
                    add_w = (add_w == 31)?0:add_w + 1;
                    count = count + 1;
                end
            MIDDLE:
                begin
                    if(p1)
                    begin
                        en_w = 1;
                        add = add_w;
                        add_w = (add_w == 31)?0 : add_w + 1;
                        count = count + 1;
                        if(count == 16) next_state = FULL;
                    end
                    else if(p2)
                    begin
                        en_r = 1;
                        add = add_r;
                        add_r = (add_r == 31)?0 : add_r + 1;
                        count = count - 1;
                        if(count == 0) next_state = NULL;
                    end
                end
            FULL:
                if(p2)
                begin
                    en_r = 1;
                    next_state = MIDDLE;
                    add = add_r;
                    add_r = add_r + 1;
                    count = count - 1;
                end
            default: next_state = NULL;
        endcase
    end
    always @(*)
    begin
        if(rst) dout = 0;
        else if(en_r) dout = out;
```

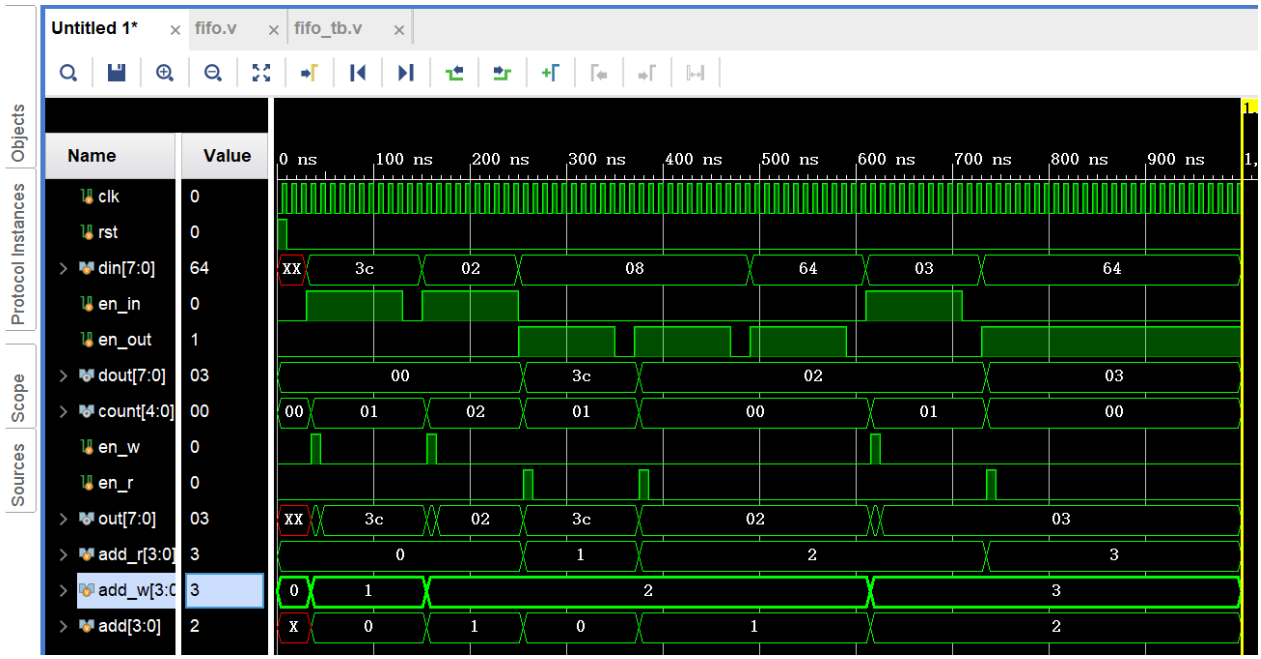
```
    end
endmodule
```

仿真代码：模拟了上板子的时候，按键时长>>周期（足以让延续了多个周期的处理顺利完成）

```
module fifo_tb();
reg clk, rst;
reg [7:0] din;
reg en_in, en_out;
wire [7:0] dout;
wire [4:0] count;

fifo F(.clk(clk), .rst(rst), .din(din), .en_in(en_in), .en_out(en_out),
.dout(dout), .count(count));

parameter PERIOD = 10;

initial
begin
    clk = 0;
    repeat (300) // 待定
        #(PERIOD/2) clk = ~clk;
    $finish;
end

initial
begin
    rst = 1;
    #PERIOD rst = 0;
end

initial
begin
    en_in = 0;
    en_out = 0;
    # (PERIOD*3);

    en_in = 1;
    en_out = 0;
    din = 60;
    # (PERIOD*10);

    en_in = 0;
    en_out = 0;
    # (PERIOD*2);

    en_in = 1;
    en_out = 0;
    din = 2;
```

```
    # (PERIOD*10);

    en_out = 1;
    en_in = 0;
    din = 8;
    # (PERIOD*10);

    en_in = 0;
    en_out = 0;
    # (PERIOD*2);

    en_out = 1;
    en_in = 0;
    din = 8;
    # (PERIOD*10);

    en_in = 0;
    en_out = 0;
    # (PERIOD*2);

    en_out = 1;
    en_in = 0;
    din = 100;
    # (PERIOD*10);

    en_in = 0;
    en_out = 0;
    # (PERIOD*2);

    en_in = 1;
    en_out = 0;
    din = 3;
    # (PERIOD*10);

    en_in = 0;
    en_out = 0;
    # (PERIOD*2);

    en_out = 1;
    en_in = 0;
    din = 100;
    # (PERIOD*10);
end
endmodule
```

结果如图：



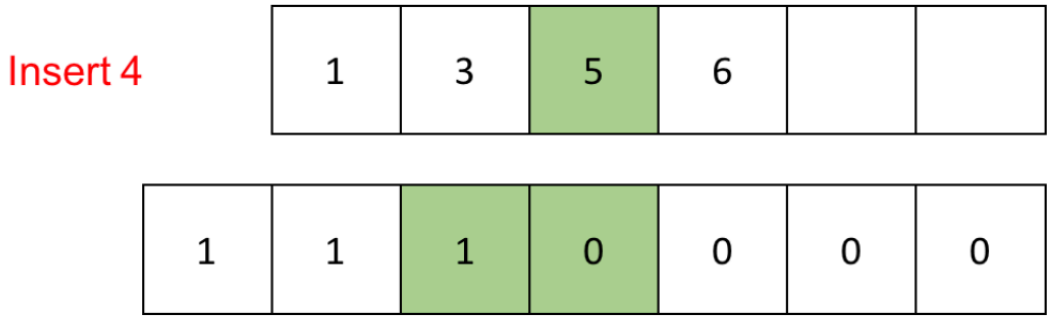

结果完全符合预期。

## 六. 思考题

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?

可以使用插入排序法，但对于Verilog电路的并发性利用得并不好。具体思路如下：

创建较大的一组 bit_array 和一组 register_file。bit_array 只为了记录插入在哪一位，




代码举例如下，其中位宽和个数都可调整，data_in 可以通过 sw 和 btn 想方法输入。

```
reg [6:0]bit_array;
reg [2:0]insert_array[5:0];

always@(*)begin
  if(rst)begin
  for(i=0;i<6;i=i+1)
    bit_array[i] = 0;
  bit_array[6] = 1;
end
else if(in_valid)begin
  for(i=0;i<6;i=i+1)
    bit_array[i] = (data_in > insert_array[i]) ? 1 : 0;
end
```

```
else begin
  for(i=0;i<6;i=i+1)
    bit_array[i] = 0;
  bit_array[6] = 1;
end
end

always@(posedge clk)begin
  if(in_valid)begin
  for(i=6;i>1;i=i-1)begin
    insert_array[i-1] <= (bit_array[i-:2]==2'b10) ? data_in :
                (bit_array[i-:2]==2'b11) ? insert_array[i-1] :
insert_array[i];
  end
end
end

always@(posedge clk)begin
  if(in_valid)
  out_valid <= 1;
else
  out_valid <= 0;
end
```

## 七. 实验总结与建议

本实验帮助我对寄存器堆和存储器的理解更加深入，为我写单周期CPU的过程省了不少力气。同时，本实验对状态机的复习也很到位，学到了新的取边沿的办法以及加深了状态机的印象。

思考题也很有意思。一开始看到我也确实头脑空空，后来查了些资料，了解了更多算法之后，也对 Verilog 有了不一样的认识。