lab 2 寄存器堆与存储器及其应用

PB21081601 张芷苒

1. 实验目的

- 掌握寄存器堆（Register File）功能、时序及其应用
- 掌握存储器的功能、时序熟练
- 掌握数据通路和控制器的设计和描述方法

实验要求：

1. 寄存器堆仿真及设计
2. 存储器 IP 核例化及仿真
3. 寄存器堆应用：FIFO 队列

2. 逻辑设计

2.1 register file 的设计

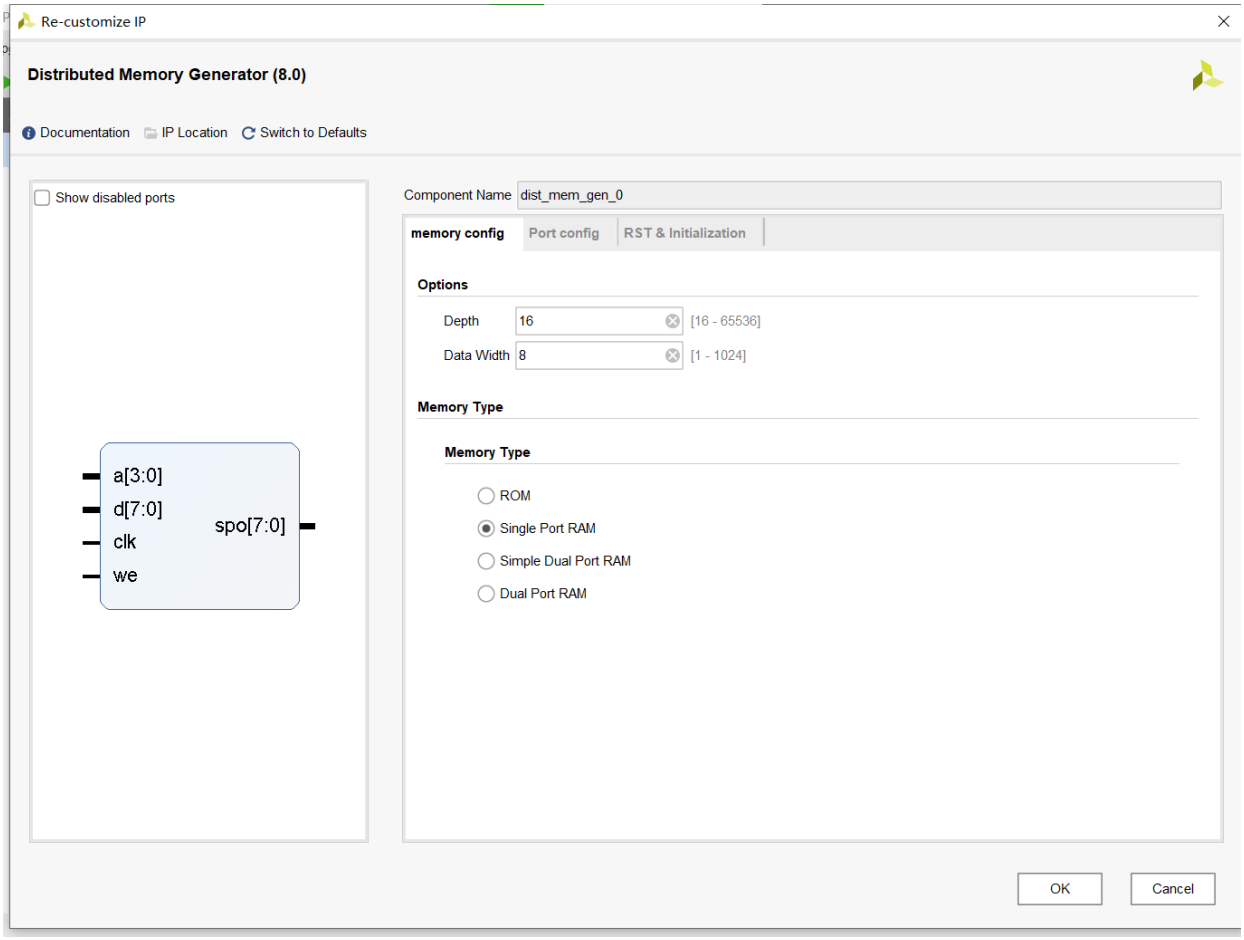
依据 ppt 要求，修改代码使得寄存器堆的0号地址始终为0.

```
module register_file #(
    parameter WIDTH = 32)
    (
    input  clk,
    input  [4:0] ra0,
    output [WIDTH-1:0] rd0,
    input  [4:0] ra1,
    output [WIDTH-1:0] rd1,
    input  [4:0] wa,
    input  we,
    input  [WIDTH-1:0] wd
);
    reg [WIDTH-1:0] regfile [0:31];
    assign rd0 = regfile[ra0];
    assign rd1 = regfile[ra1];
    always @(posedge clk)
    begin
        if (we)
        begin
            if (wa == 0) regfile[0] <= 0;
            else regfile[wa] <= wd;
```

```
        end
    end
endmodule
```

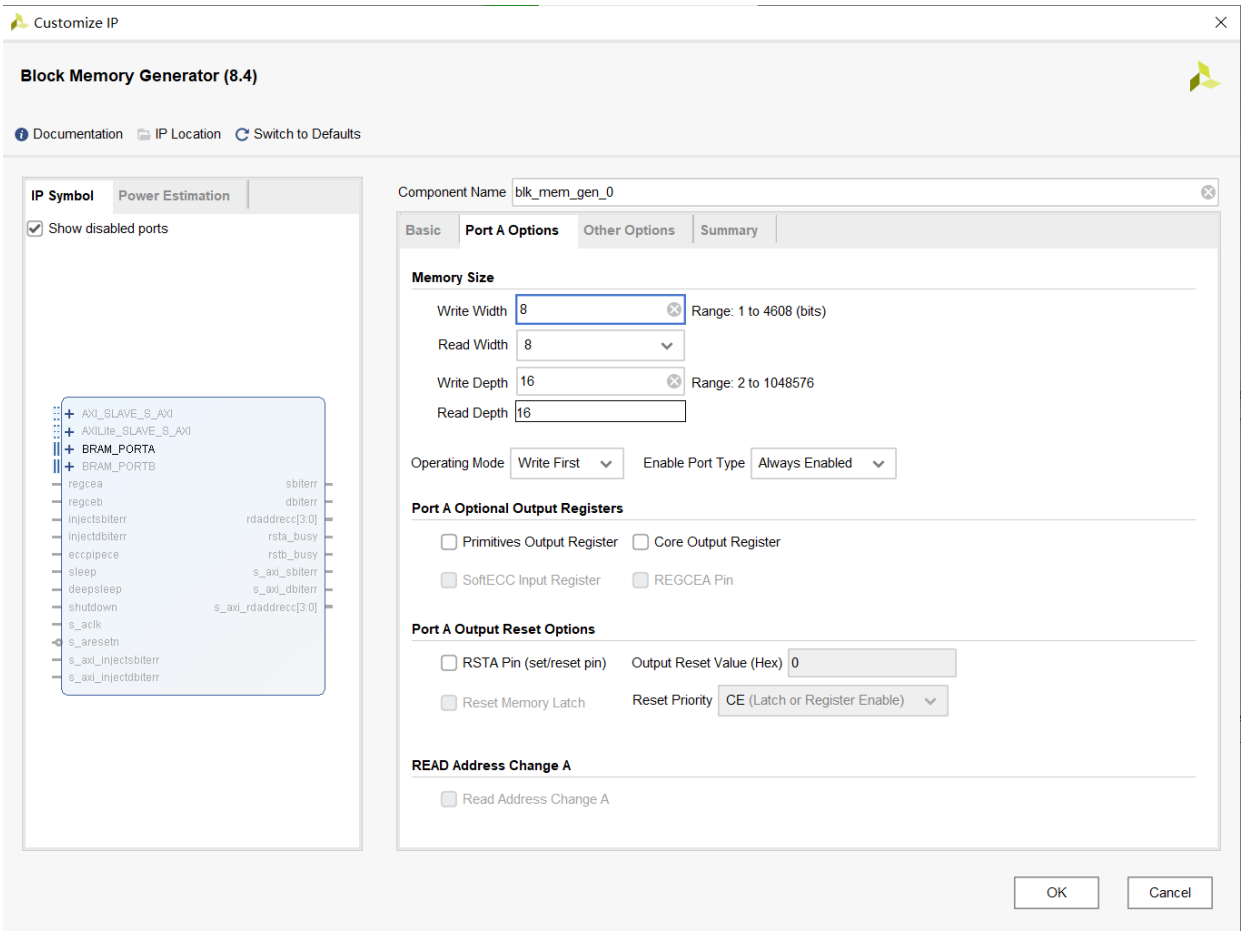
2.2 IP core 的设计

2.2.1 distributed

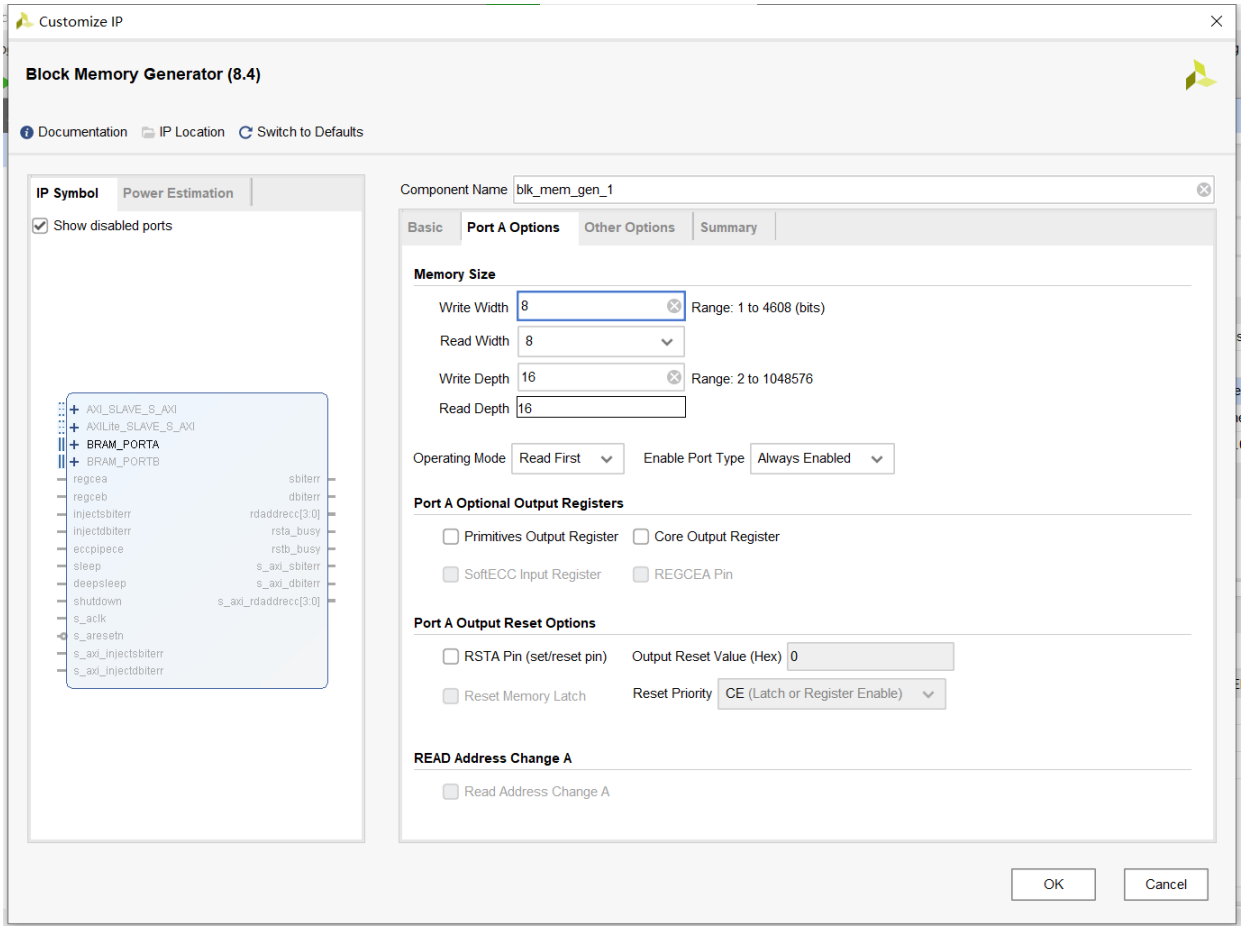


2.2.2 block

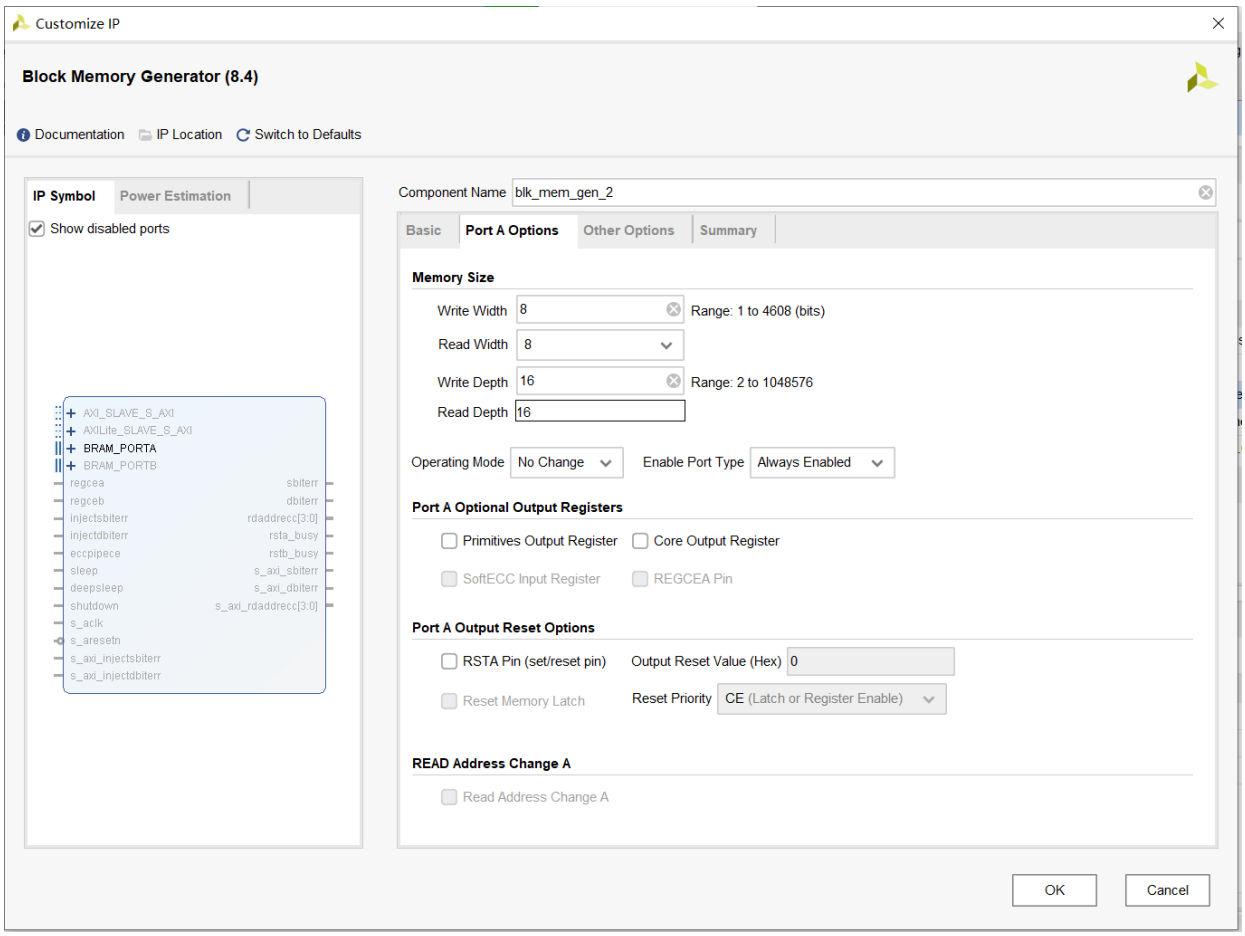
write first:


Customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
AXI_SLAVE_S_AXI
AXILite_SLAVE_S_AXI
BRAM_PORTA
BRAM_PORTB
regcea
regceb
injectsbiterr
injectdbiterr
eccpipece
sleep
deepsleep
shutdown
s_aclk
s_aresetn
s_axi_injectsbiterr
s_axi_injectdbiterr
sbiterr
dbiterr
rdaddrecc[3:0]
rsta_busy
rstb_busy
s_axi_sbiterr
s_axi_dbiterr
s_axi_rdaddrecc[3:0]
Component Name
blk_mem_gen_0
Basic
Port A Options
Other Options
Summary
Memory Size
Write Width
8
Range: 1 to 4608 (bits)
Read Width
8
Write Depth
16
Range: 2 to 1048576
Read Depth
16
Operating Mode
Write First
Enable Port Type
Always Enabled
Port A Optional Output Registers
Primitives Output Register
Core Output Register
SoftECC Input Register
REGCEA Pin
Port A Output Reset Options
RSTA Pin (set/reset pin)
Output Reset Value (Hex)
0
Reset Memory Latch
Reset Priority
CE (Latch or Register Enable)
READ Address Change A
Read Address Change A
OK
Cancel

read first:

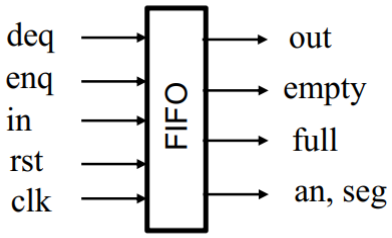

Customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
AXI_SLAVE_S_AXI
AXILite_SLAVE_S_AXI
BRAM_PORTA
BRAM_PORTB
regcea
regceb
injectsbiterr
injectdbiterr
eccpipece
sleep
deepsleep
shutdown
s_aclk
s_aresetn
s_axi_injectsbiterr
s_axi_injectdbiterr
sbiterr
dbiterr
rdaddrecc[3:0]
rsta_busy
rstb_busy
s_axi_sbiterr
s_axi_dbiterr
s_axi_rdaddrecc[3:0]
Component Name
blk_mem_gen_1
Basic
Port A Options
Other Options
Summary
Memory Size
Write Width
8
Range: 1 to 4608 (bits)
Read Width
8
Write Depth
16
Range: 2 to 1048576
Read Depth
16
Operating Mode
Read First
Enable Port Type
Always Enabled
Port A Optional Output Registers
Primitives Output Register
Core Output Register
SoftECC Input Register
REGCEA Pin
Port A Output Reset Options
RSTA Pin (set/reset pin)
Output Reset Value (Hex)
0
Reset Memory Latch
Reset Priority
CE (Latch or Register Enable)
READ Address Change A
Read Address Change A
OK
Cancel

no change:

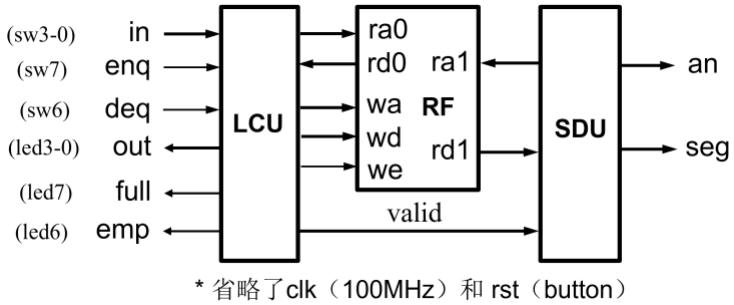


2.3 fifo 队列的设计

fifo队列的输入输出：



其数据通路：



* 省略了clk（100MHz）和 rst（button）

下面介绍设计中的几个关键模块：

2.3.1 寄存器堆模块

与第一题相同，x0恒为0.

```
module reg_file #(
    parameter WIDTH = 32)
    (
    input  clk,
    input  [4:0] ra0,
    output [WIDTH-1:0] rd0,
    input  [4:0] ra1,
    output [WIDTH-1:0] rd1,
    input  [4:0] wa,
    input  we,
    input  [WIDTH-1:0] wd
);
    reg [WIDTH-1:0] regfile [0:31];
    assign rd0 = regfile[ra0];
    assign rd1 = regfile[ra1];
    always @(posedge clk)
    begin
        if (we) regfile[0] <= wd;
        regfile[0] <= 32'b0;
    end
endmodule
```

2.3.2 List Control Unit（LCU）模块

该模块处理输入信号，对寄存器堆进行写入，并**向显示单元发送 valid 信号**。由于 enq 与 deq 在经过取边沿后每次有效的时间都是一个周期，并且进入三个状态的条件与当前状态一样，所以没有用到状态机。

在该模块中，定义了两个寄存器变量： 头指针 head 与尾指针 tail，两者都指向下一次入/出队的数据下标，即初始值均为 0。 在入队后，尾指针 tail 自增，在出队后，头指针 head 自增。对于判断队列是否为空 (满)，只需要判断 valid 的每一位是否都是 0/1。

```
module list_control_unit(
    input clk,
    input rst,  //同步复位
    input [3:0] in,
    input enq,  //入队
    input deq,  //出队
    input [3:0]  rd,
    output full,
    output emp,
    output reg [3:0] out,  //出队数据
    output [2:0] ra,
    output we,
    output [2:0] wa,
    output [3:0]  wd,
    output reg [7:0] valid
);
```

```
    reg [2:0] head;  //头指针
    reg [2:0] tail;  //尾指针

    assign full = &valid;
    assign emp = ~(|valid);

    assign ra = head;
    assign we = enq & ~full & ~rst;
    assign wa = tail;
    assign wd = in;

    always @(posedge clk) begin
        if (rst) begin
            valid <= 8'h00;
            head <= 3'h0;
            tail <= 3'h0;
            out <= 3'h0;
        end
        else if(enq & ~full) begin
            valid[tail] <= 1'b1;
            tail <= tail + 3'h1;
        end
        else if(deq & ~emp) begin
            valid[head] <= 1'b0;
            head <= head + 3'h1;
            out <= rd;
        end
    end
endmodule
```

2.3.3 Segment Display Unit (SDU) 模块

在显示单元中，输入的时钟信号是 100MHz 的，对于数码管来说频率过快，因此对其降频到 400Hz。

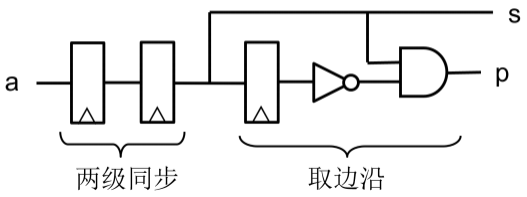
这里使用了一个 18 位的模 250000 计数器，在每个 100MHz 的时钟上升沿计数，并在进位(计数器值大于等于 249999) 时对输出的地址 (输出到寄存器堆的 ra0) 加一，实现对寄存器堆 400Hz 的扫描。同时，在计数器值为 1 时会输出一个脉冲，用于控制数码管信号的输出。**若队列为空**，则显示单元会输出信号使得最低位数码管显示 0，否则会使得有效位的数码管上显示该位的值。

```
module segplay_unit(
    input clk_100mhz,
    input [3:0] data,
    input [7:0] valid,
    output reg [2:0] addr,
    output [2:0] segplay_an,
    output [3:0] segplay_data
);
    //给时钟降速
    wire clk_400hz;
    reg [17:0] clk_cnt;
    assign clk_400hz = ~(|clk_cnt); //clk_400hz = (clk_cnt == 0)
```

```
    always @(posedge clk_100mhz) begin
        if (clk_cnt >= 18'h3D08F) begin //clk_cnt >= 249999
            clk_cnt <= 18'h00000;
            addr <= addr + 3'b001;
        end else
            clk_cnt <= clk_cnt + 18'h00001;
    end
    //显示输出
    reg [2:0] segplay_an_reg;
    reg [3:0] segplay_data_reg;
    always @(posedge clk_100mhz) begin
        if (clk_400hz && valid[addr]) begin
            segplay_an_reg <= addr;
            segplay_data_reg <= data;
        end
    end
    assign segplay_data = (|valid) ? segplay_data_reg : 4'h0;
    assign segplay_an = (|valid) ? segplay_an_reg : 3'h0;
endmodule
```

2.3.4 取边沿模块

按照如下电路对信号两级同步并取边沿。




```
module sig_edge(
    input clk,
    input insig,  //输入信号
    output sync,  //同步信号
    output outedge  //信号边沿
);
    reg in_reg_0;
    reg in_reg_1;
    reg in_reg_2;
    always @(posedge clk) begin
        in_reg_0 <= insig;
        in_reg_1 <= in_reg_0;
        in_reg_2 <= in_reg_1;
    end
    assign sync = in_reg_1;
    assign outedge = in_reg_1 & ~in_reg_2;
endmodule
```

2.3.5 顶层 fifo 模块

```
module fifo(
    input clk,
    input rst,
    input enq,
    input [3:0] in,
    input deq,
    output [3:0] out,
    output full,
    output emp,
    output [2:0] an,
    output [3:0] seg
);

    wire enq_edge;
    wire deq_edge;
    wire we;
    wire [2:0] ra0, ra1, wa;
    wire [3:0] rd0, rd1, wd;
    wire [7:0] valid;

    reg_file RF(
        .clk(clk),
        .ra0(ra0),
        .ra1(ra1),
        .we(we),
        .wa(wa),
        .wd(wd),
        .rd0(rd0),
        .rd1(rd1)
    );

    sig_edge SEDG_enq(
        .clk(clk),
        .insig(enq),
        .outedge(enq_edge)
    );
    sig_edge SEDG_deq(
        .clk(clk),
        .insig(deq),
        .outedge(deq_edge)
    );

    list_control_unit LCU(
        .clk(clk),
        .rst(rst),
        .in(in),
        .enq(enq_edge),
        .deq(deq_edge),
        .rd(rd0),
        .full(full),
```

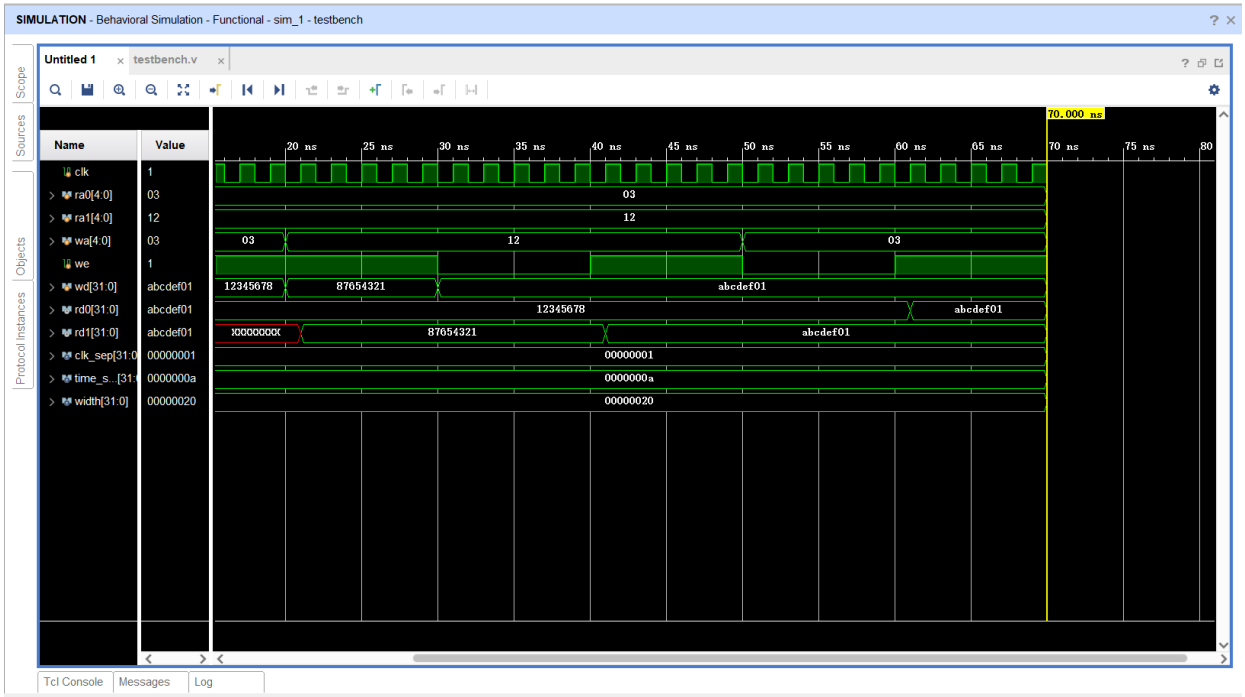
```
        .emp(emp),
        .out(out),
        .ra(ra0),
        .we(we),
        .wa(wa),
        .wd(wd),
        .valid(valid)
    );

    segplay_unit SDU(
        .clk_100mhz(clk),
        .data(rd1),
        .valid(valid),
        .addr(ra1),
        .segplay_an(an),
        .segplay_data(seg)
    );
endmodule
```

3. 仿真结果与分析

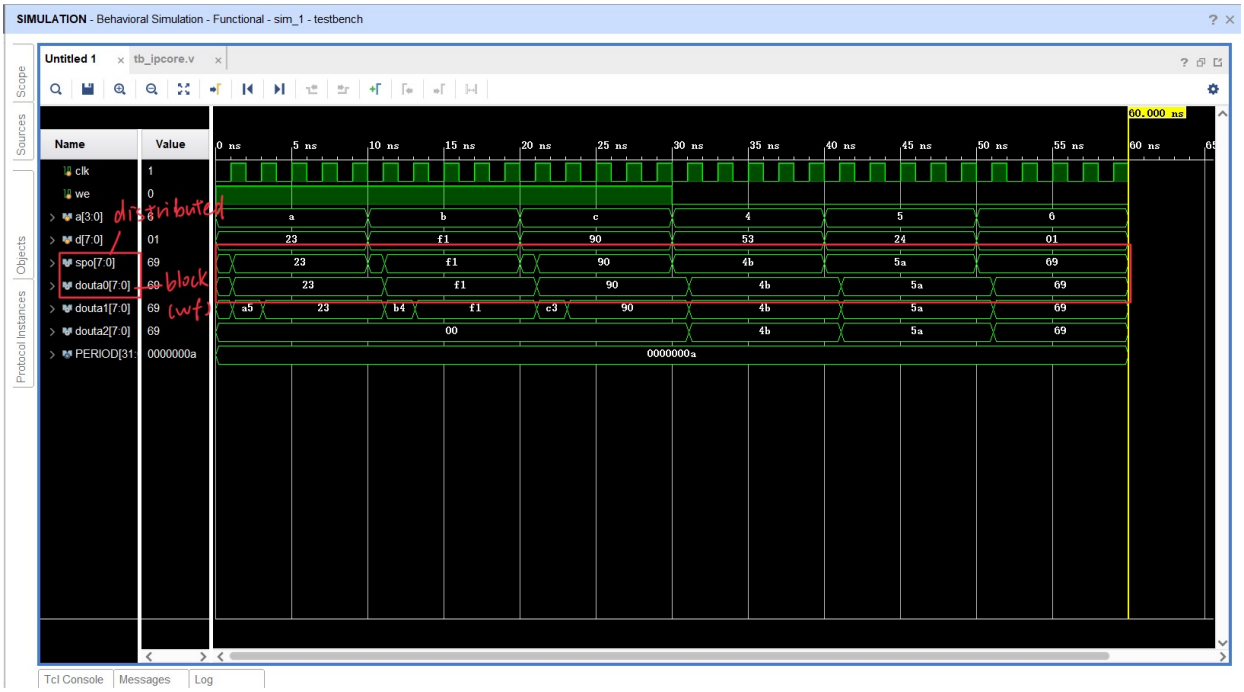
3.1 register file 仿真结果

仿真如下：



3.2 IP core 仿真结果

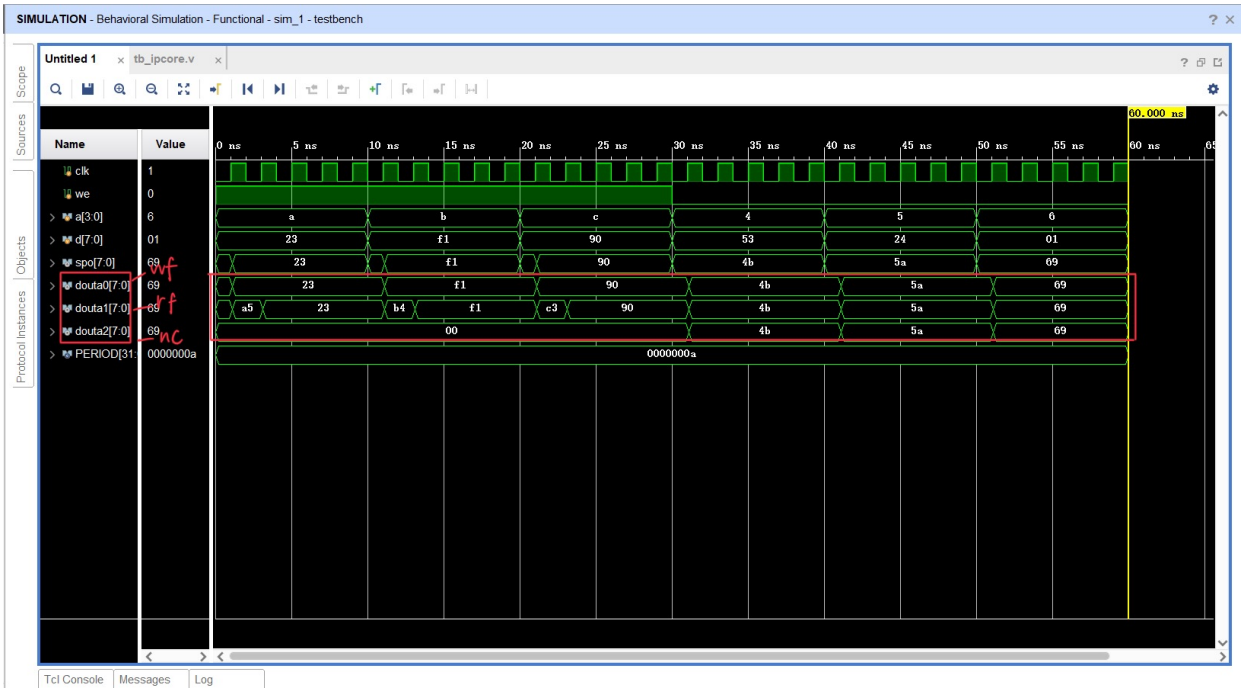
3.2.1 对比分布式和块式16 x 8位单端口RAM之间的时序差异



该图展示了块式与分布式 RAM 之间的时序差异。通过对比，可以发现它们有如下区别：

- 分布式RAM的读写延迟时间更短，因此在时钟上升沿后，输出信号几乎立即出现。相比之下，块RAM的输出信号需要较长的时间来响应。
- 在分布式RAM中，读和写操作的时间几乎相同，因为分布式RAM的读写延迟时间相似。而在块RAM中，读和写操作的延迟时间可能存在较大差异。
- 块RAM的数据输入和输出通常需要占用更多的时间（为了处理较大的数据块）。相比之下，分布式RAM的数据输入和输出通常需要占用更少的时间。

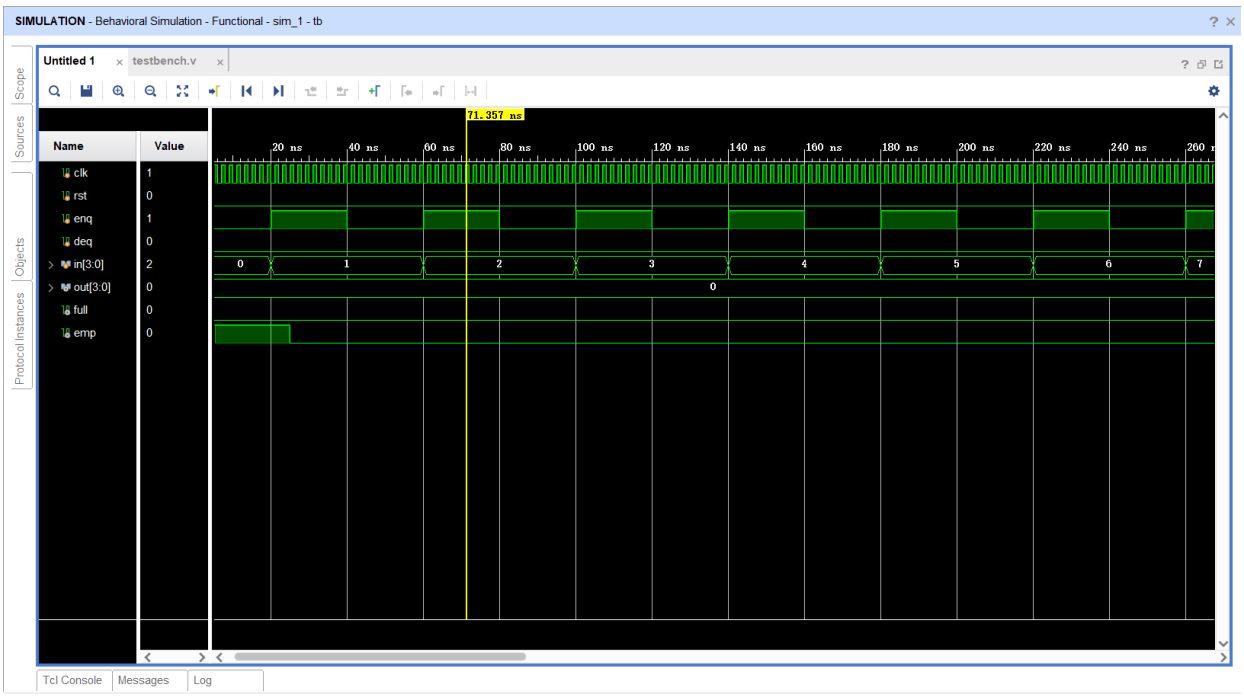
3.2.2 块式RAM三种不同模式的对比



设置 ip core 时分别采用 write first, read first 和 no change 的方式，经过仿真得到如上的图像。通过对比，可以得知：

1. write first模式：当同一个地址同时进行读写操作时，输出数据是写入的数据。写操作优先于读操作。
2. read first模式：当同一个地址同时进行读写操作时，输出数据是原来存储在该地址的数据，读操作优先于写操作。
3. no change模式：当同一个地址同时进行读写操作时，输出数据不变。读写操作都不影响输出数据。

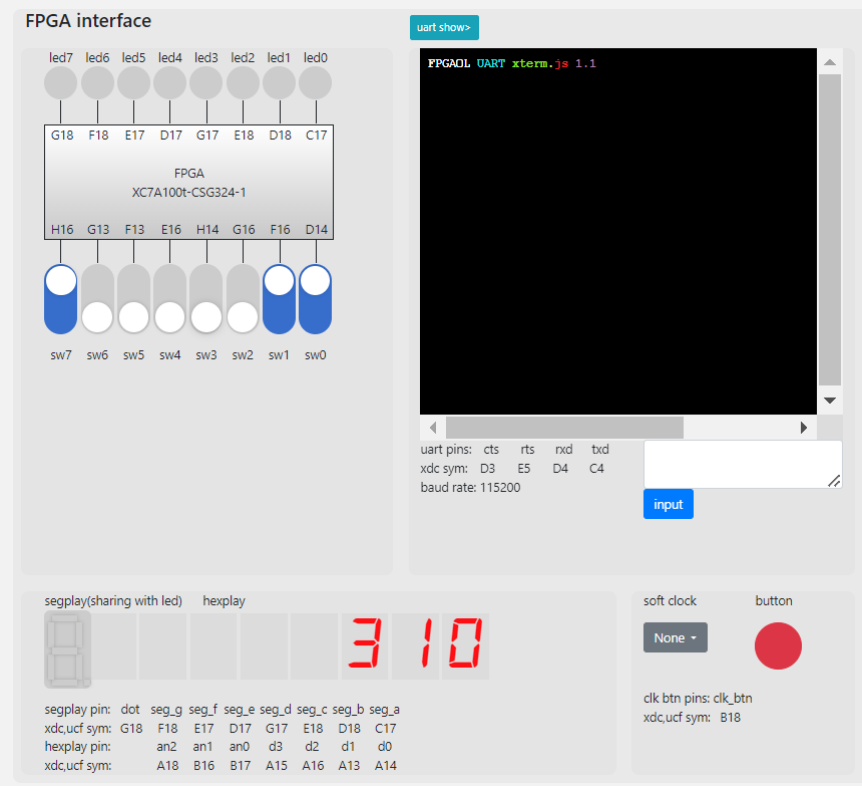
3.3 fifo 队列的仿真



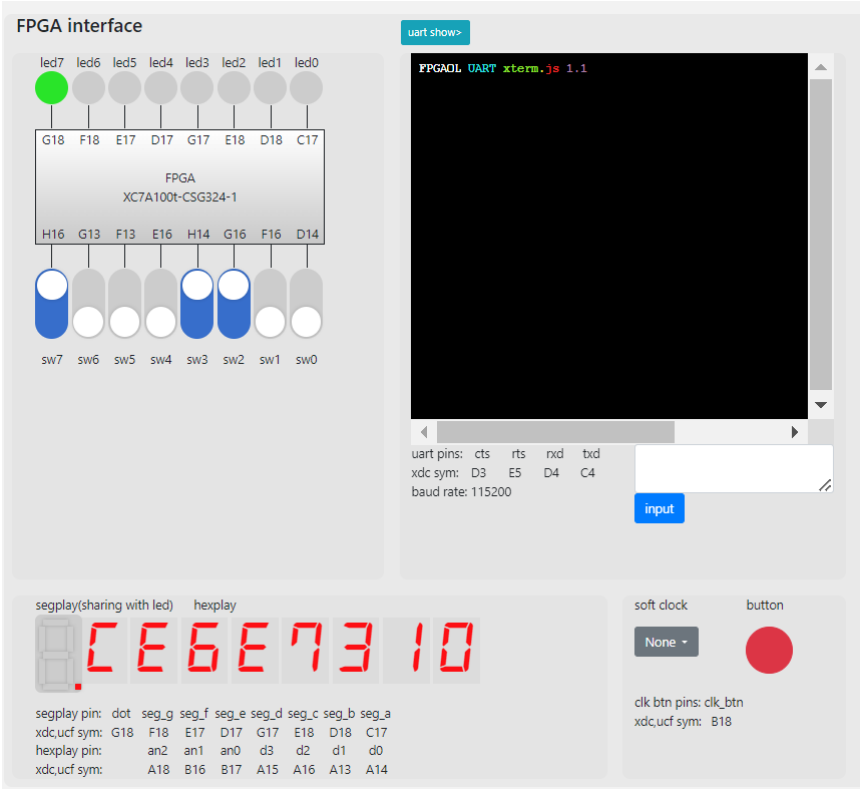
4. 测试结果与分析

主要展示fifo队列的上板结果。

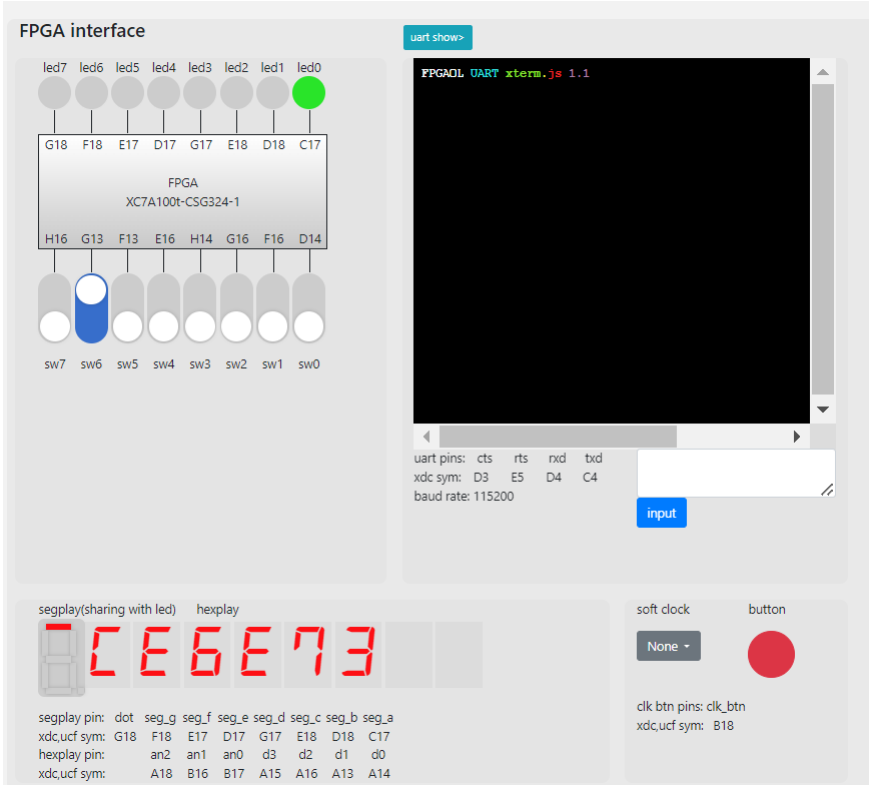
第一个数据入队时，入队数据写入x0寄存器。因为x0寄存器恒0，这时队列中含有一个0，数码管显示一个0，表现出与复位时一样的显示，但这与复位时显示一个0是两种不同的状态。后续数据正常入队。在出队时，若出队的数据从x0寄存器中读出，此时出队的数据为0。



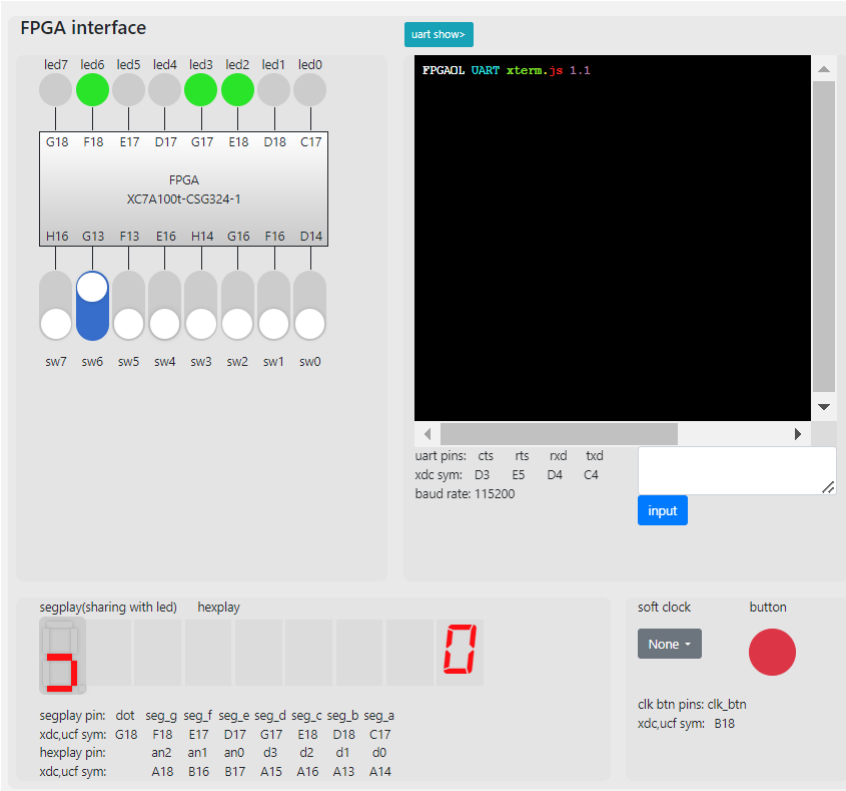
队满，led7 亮起。



出队，led 3-0 显示上一个出队的数。



全部出队，下方数码管显示0，同时 led 6 亮起，代表队空。



5. 总结

本次实验任务有些琐碎，ppt 依然有许多不够详尽的地方，但是相信以后有了实验手册之后会更加完善。