Lab2

1.实验目的与内容

- 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

2.逻辑设计

2.1寄存器堆设计

□ 寄存器堆介绍

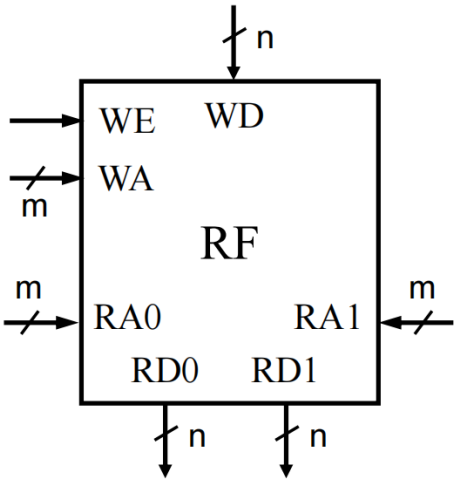
✓ **处理器的32个通用寄存器位于一个叫做寄存器堆（register file）的结构中。**

✓ **1个写端口**

- WA：写地址
- WD：写入数据
- WE：写使能

✓ **2个读端口**

- RA0、RA1：读地址
- RD0、RD1：读出数据



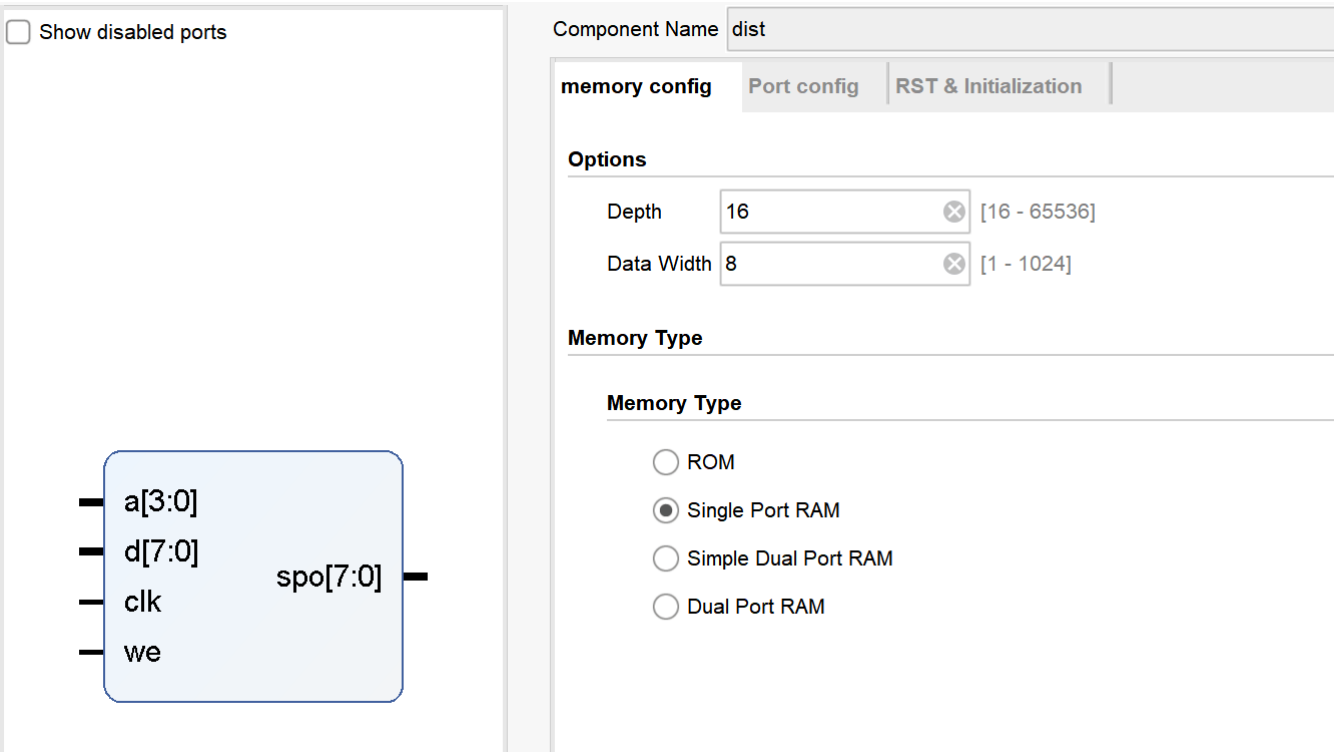
三端口的$2^m \times n$位寄存器堆外形图

```
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 = (ra0 == 0) ? 0 : regfile[ra0];
  assign rd1 = (ra1 == 0) ? 0 : regfile[ra1];

  always @(posedge clk) begin
    if (we && wa != 0)
      regfile[wa] <= wd;
  end

endmodule
```

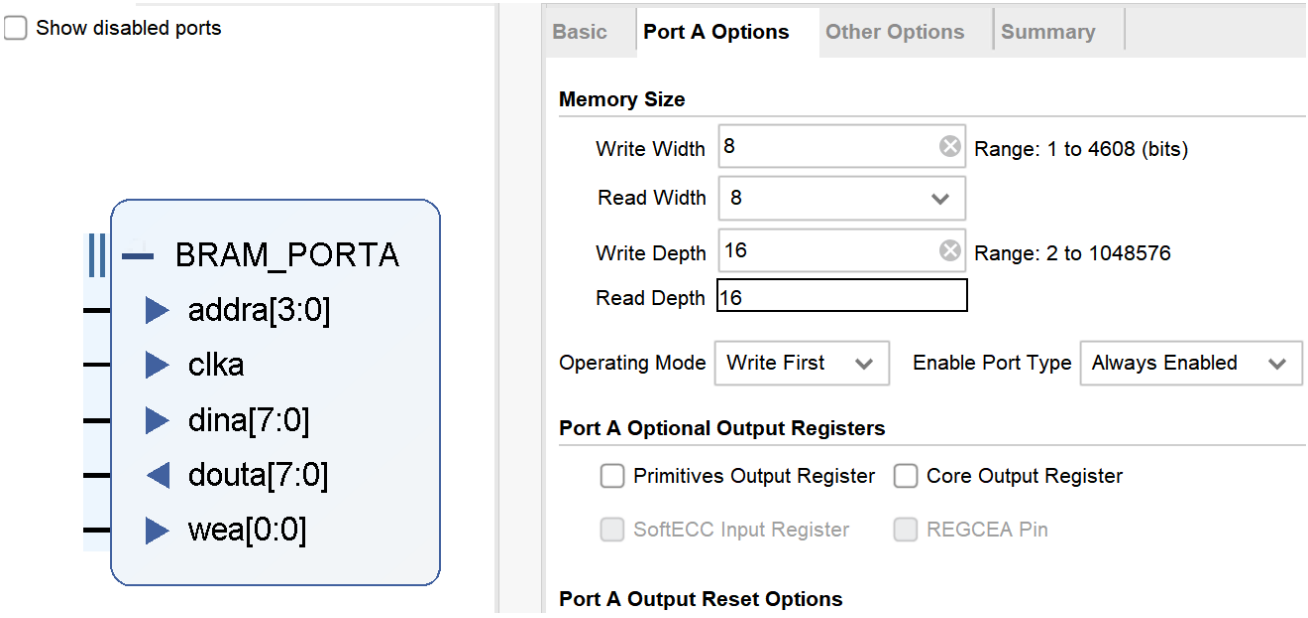
2.2 IP：RAM存储器设计

2.2.1分布式16 × 8位单端口RAM

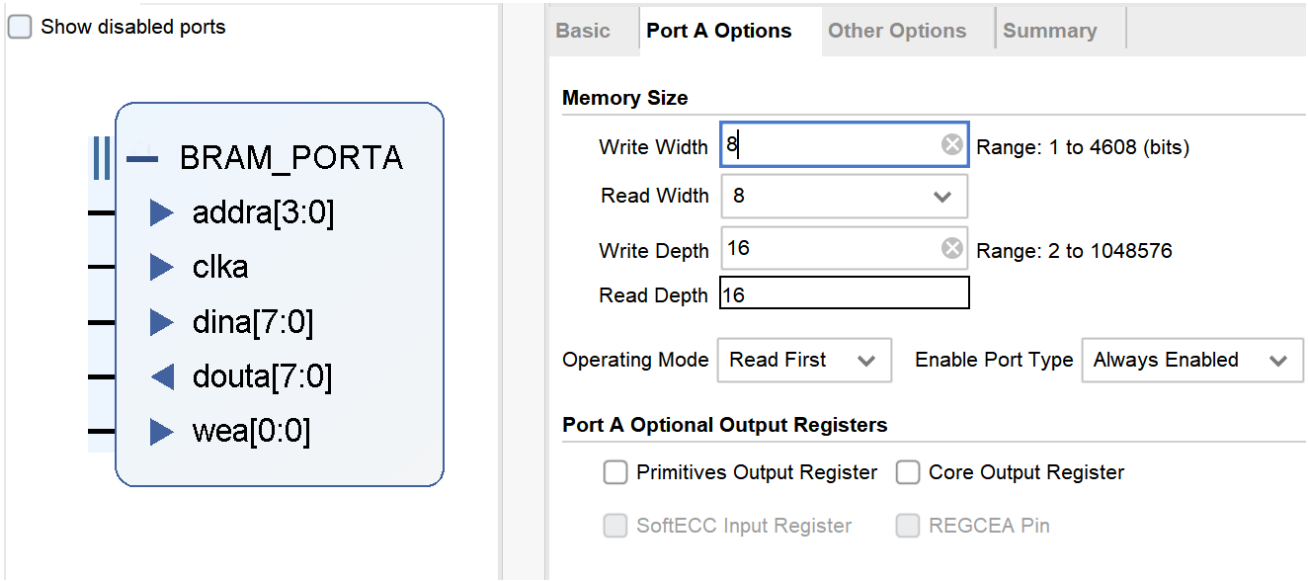


2.2.2块式16 × 8位单端口RAM

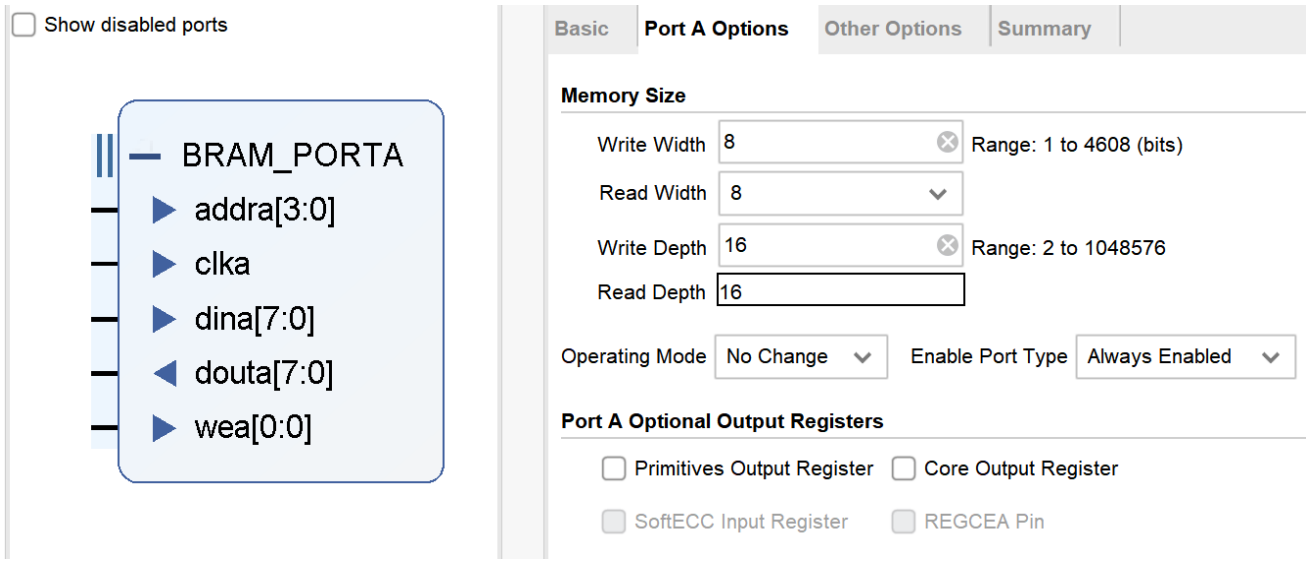
- write first)



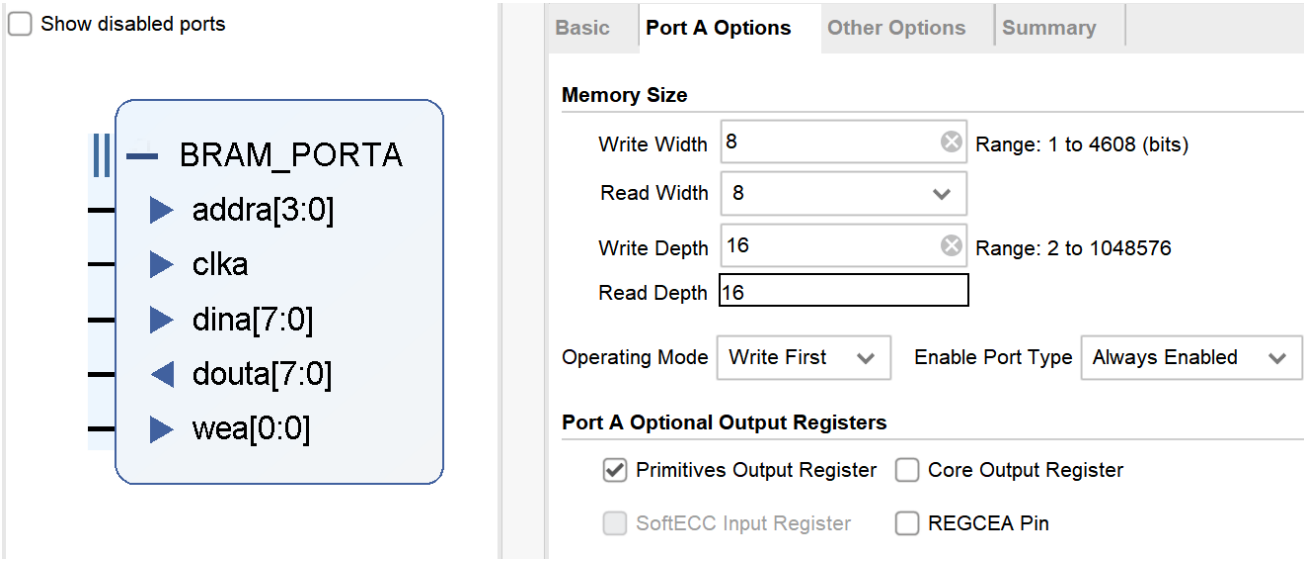
- read first)



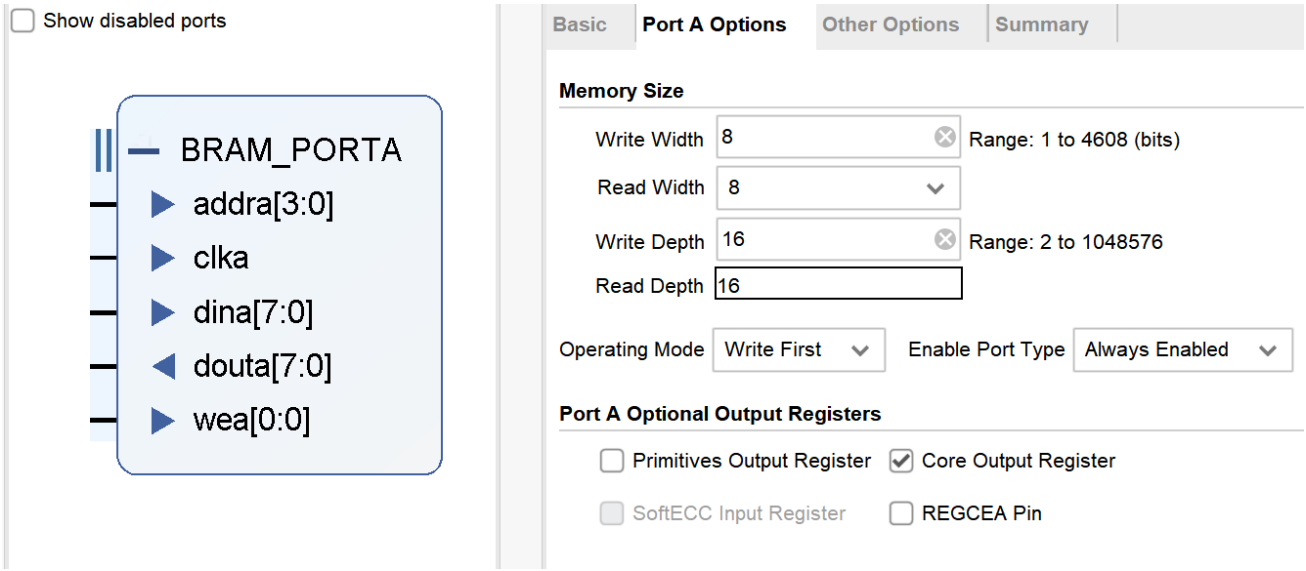
- no change)



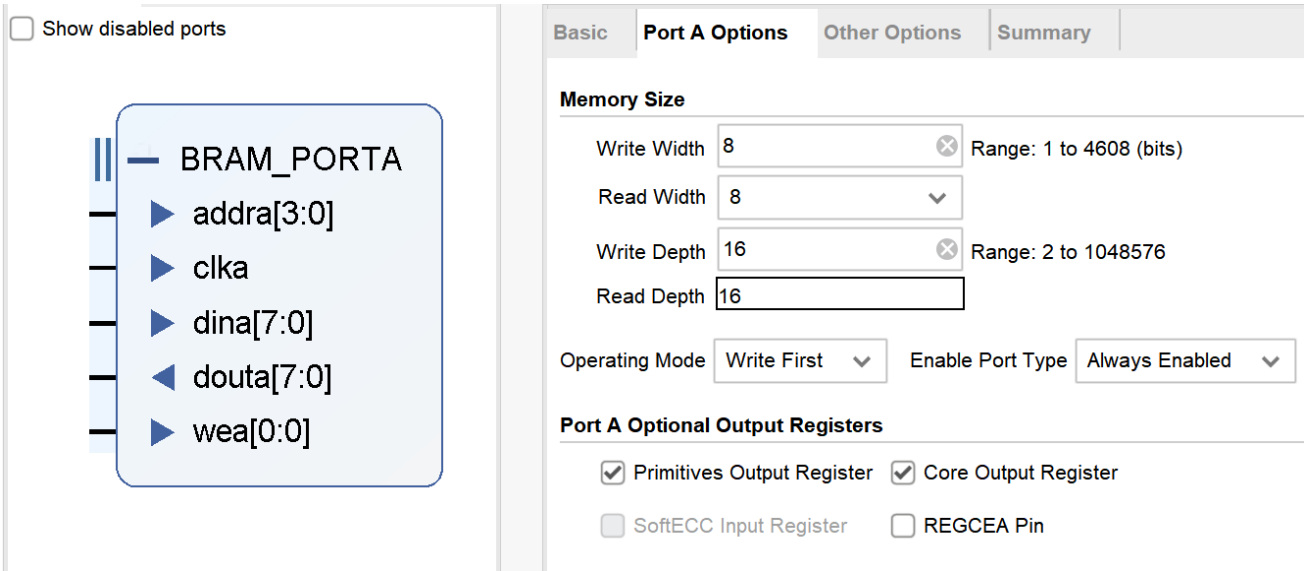
- primitive)



- core)



- both)



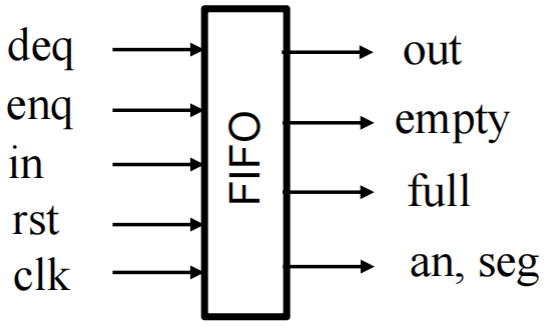
- coe文件

```
memory_initialization_radix = 16
memory_initialization_vector = 23 f4 07 21 11 ff AB e1 00 01 00 01 00 0A 00
00
```

2.3利用寄存器堆实现 FIFO 队列

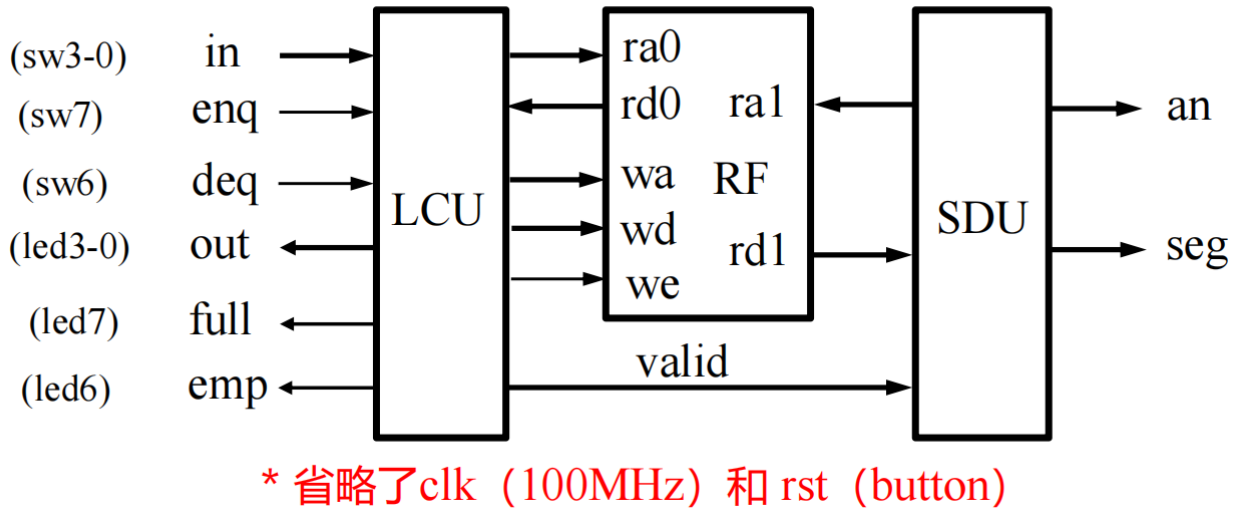
2.3.1

FIFO的输入输出



它的数据通路如下,

- 显示队列数据和出入状态



* 省略了clk（100MHz）和 rst（button）

模块层次图

```
fifo (FIFO.v):
 RF: reg_file
 SEDG_enq: signal_edge
 SEDG_deq: signal_edge
 LCU: list_control_unit
 SDU: seg_unit
```

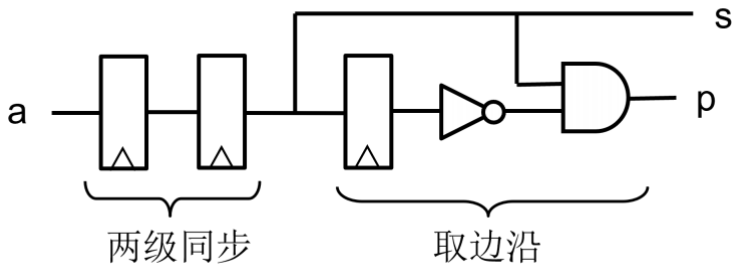
2.3.2 各模块代码

寄存器堆

```
module reg_file (
  input clk,
  input [2:0] ra0,      // read address 0
  input [2:0] ra1,      // read address 1
  input we,             // write enable
  input [2:0] wa,       // write address
  input [3:0] wd,       // write data
  output [3:0] rd0,     // read data 0
  output [3:0] rd1      // read data 1
);
  reg [3:0] regfile [0:7];
  assign rd0 = (ra0 == 0) ? 0 : regfile[ra0];
  assign rd1 = (ra1 == 0) ? 0 : regfile[ra1];

  always @(posedge clk) begin
    if (we && wa != 0)
      regfile[wa] <= wd;
  end
endmodule //reg_file
```

取边沿模块

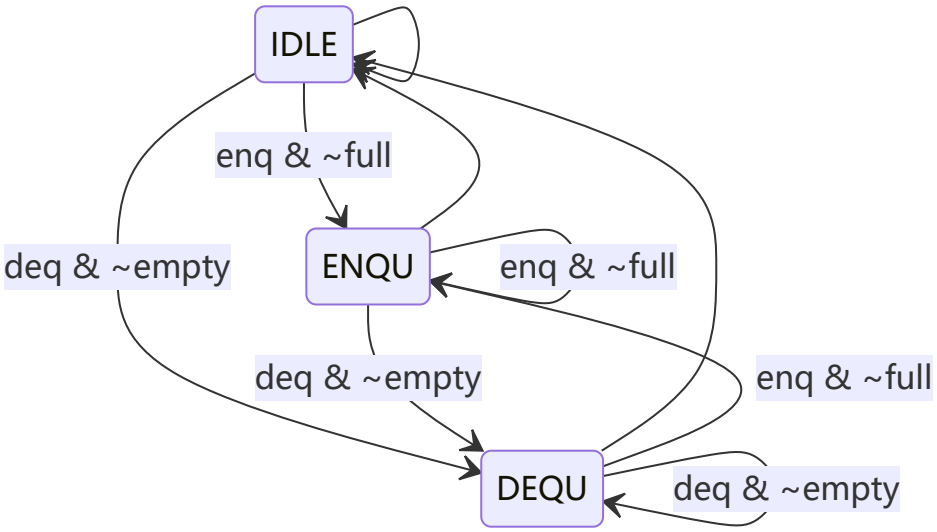
（两级同步取边沿是为了防止亚稳态干扰）




```
module signal_edge(
  input clk,
  input a,        // 输入信号
  output s,       // 同步后的信号
  output p        // 信号的边沿信号
);
  reg st;           // 输入信号的状态
  reg temp_s;       // 同步后的信号的状态
  reg pt;           // 上一个同步后的信号的状态
```

```
10
11    always @(posedge clk) begin
12      st <= a;                // 记录当前输入信号的状态
13      temp_s <= st;           // 将输入信号同步到时钟上升沿，得到同步后的信号
14      pt <= temp_s;           // 记录上一个同步后的信号的状态
15    end
16
17    assign s = temp_s;        // 输出同步后的信号
18    assign p = temp_s & ~pt; // 计算边沿信号
19
20  endmodule
21
```

状态机模块




```
1   module FSM(
2     input clk,              //时钟信号
3     input enq,              //入队标志
4     input deq,              //出队标志
5     input rst,              //同步复位(高电平有效)
6     input full,             //队满标志
7     input empty,              //队空标志
8     output reg [1:0] state     //状态
9   );
10
11  //定义状态
12  parameter IDLE = 2'b00;//空状态
13  parameter ENQU = 2'b01;//入队状态
14  parameter DEQU = 2'b10;//出队状态
15  //parameter DONE = 2'b11;
```

```
16  //定义现态和次态
17  reg [1:0] curr_state;
18  reg [1:0] next_state;
19
20  //FSM第一段，描述状态切换
21  always @(posedge clk or posedge rst) begin
22    if(rst) begin
23      curr_state <= IDLE;
24    end
25    else begin
26      curr_state <= next_state;
27    end
28    end
29
30  //FSM第二段，判断状态转移条件，描述状态转移规律
31  always @(*) begin
32    case (curr_state)
33      IDLE:
34      if(enq & ~full) begin
35        next_state <= ENQU;
36      end
37      else if(deq & ~empty) begin
38        next_state <= DEQU;
39      end
40      else begin
41        next_state <= IDLE;
42      end
43
44      ENQU:
45       if(enq & ~full) begin
46        next_state <= ENQU;
47      end
48      else if(deq & ~empty) begin
49        next_state <= DEQU;
50      end
51      else begin
52        next_state <= IDLE;
53      end
54
55      DEQU:
56      if(enq & ~full) begin
57        next_state <= ENQU;
58      end
59      else if(deq & ~empty) begin
60        next_state <= DEQU;
61      end
62      else begin
```

```
      next_state <= IDLE;
    end

    default:
      next_state <= IDLE;
  endcase
  end

//FSM第三段，描述状态输出，Moore型输出
always @(posedge clk) begin
  state <= curr_state;
end

endmodule
```

LCU模块

```
//LCU
module list_control_unit (
    input clk,                // clock (100 MHz, 高电平有效)
    input rst,                // 同步复位 (高电平有效)
    input [3:0] in,           // 入队数据
    input enq,                // 入队边缘
    input deq,                // 出队边缘
    input [3:0] rd,           // 写端口数据
    output full,
    output empty,               // 队列空标志
    output reg [3:0] out,     // 出队数据
    output [2:0] ra,          // 读端口地址
    output we,                // 写使能
    output [2:0] wa,          // 写端口地址
    output [3:0] wd,          // 写端口数据
    output reg [7:0] valid    // 数据有效标志
);

    reg [2:0] head;           // 头指针
    reg [2:0] tail;           // 尾指针
    wire [1:0] state;         // 状态

    assign full = &valid;     // 当标志的每一位都为1时,说明队列已满
    assign empty = ~(|valid);   // 当标志的每一位都为0时,说明队列为空

    assign ra = head;         // 从寄存器文件中读数据(出队),读端口地址等于队头
    assign we = enq & ~full & ~rst; // 允许向寄存器中写数据即允许入队且复位信号无效
    assign wa = tail;         // 向寄存器文件中写数据(入队)，写端口地址等于队尾
    assign wd = in;           // 读入寄存器文件的数据等于输入数据
```

```
30
31      // 状态产生
32      FSM fsm(
33          .clk(clk),
34          .enq(enq),
35          .deq(deq),
36          .rst(rst),
37          .full(full),
38          .empty(empty),
39          .state(state)
40      );
41
42      always @(posedge clk or posedge rst) begin
43          if (rst) begin
44              valid <= 8'h0;
45              head <= 3'h0;
46              tail <= 3'h0;
47              out <= 3'h0;
48          end
49          else if (state == 2'b00) begin
50              valid <= valid;
51              head <= head;
52              tail <= tail;
53              out <= out;
54          end
55          else if (state == 2'b01) begin
56              valid[tail] <= 1'b1;//赋有效位
57              tail <= tail + 1;//队伍前进
58          end
59          else if (state == 2'b10) begin
60              valid[head] <= 1'b0;
61              head <= head + 1;
62              out <= rd;
63          end
64      end
65
66  endmodule
67
```

SDU

```
1  module seg_unit(
2    input clk,
3    input [3:0] data,
4    input [7:0] valid,
5    output reg [2:0] addr,
```

```
  output [2:0] san,
  output [3:0] sdata
);
  parameter COUNTER_MAX = 250000;
  // Counter: slow the clock down to 400Hz
  wire clk_400hz;
  reg [17:0] counter;  // 计数器，用于降低时钟频率
  assign clk_400hz = ~(|counter); // clk_400hz = (counter == 0)，降低时钟频率到400Hz

  reg [2:0] san_reg;  // 段选信号寄存器
  reg [3:0] sdata_reg;  // 段码数据寄存器

  always @(posedge clk) begin
    if (counter >= COUNTER_MAX - 1) begin
      counter <= 0;
      addr <= addr + 1; //当前数码管后的以为用来显示
    end
    else
      counter <= counter + 1;
  end

  // 段选
  always @(posedge clk) begin
    if (clk_400hz && valid[addr]) begin  // 400Hz且对应地址的有效信号为1
      san_reg <= addr;  // 将地址存储到段选信号寄存器中
      sdata_reg <= data;  // 将数据存储到段码数据寄存器中
    end
  end

  assign sdata = (|valid) ? sdata_reg : 4'h0000;  // 如果有效信号全为0，则输出全0，否则输出段码数据
  assign san = (|valid) ? san_reg : 3'h000;  // 如果有效信号全为0，则输出全0，否则输出段选信号

endmodule
```

fifo

```
module fifo(
  input clk,
  input rst,
  input enq,
  input [3:0] in, // enqueue data
  input deq,
  output [3:0] out, // dequeue data
```

```
  output full,
  output empty,
  output [2:0] an, // segment display selection
  output [3:0] seg // segment display data
);

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

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

  signal_edge SEDG_enq(
    .clk(clk),
    .a (enq),
    .p (enq_edge)
  );

  signal_edge SEDG_deq(
    .clk(clk),
    .a (deq),
    .p (deq_edge)
  );

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

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

  seg_unit SDU(
    .clk (clk),
    .data (rd1),
    .valid (valid),
    .addr (ra1),
    .san (an),
    .sdata(seg)
    );
endmodule
```

3.仿真结果与分析

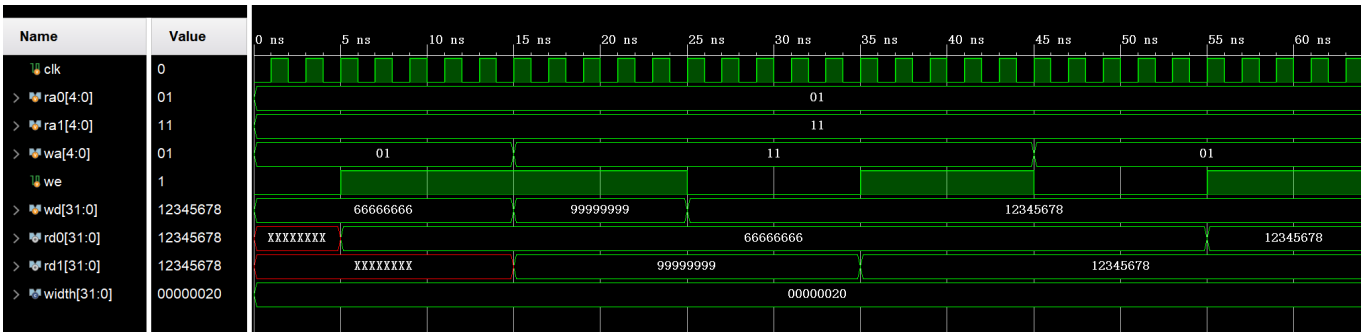
3.1寄存器堆

仿真文件

```
`timescale 1ns / 1ps

module testbench();
  parameter width = 32;
  reg clk;
  reg [4:0] ra0;
  reg [4:0] ra1;
  reg [4:0] wa;
  reg we;
  reg [width-1:0] wd;
  wire [width-1:0] rd0;
  wire [width-1:0] rd1;

  register_file regfile(
    .clk(clk),
    .ra0(ra0),
    .ra1(ra1),
    .wa(wa),
    .we(we),
    .wd(wd),
```

```
21      .rd0(rd0),
22      .rd1(rd1)
23    );
24
25    initial begin
26      clk = 0;
27      ra0 = 5'h01;
28      ra1 = 5'h11;
29      forever #1 clk = ~clk;
30    end
31
32    initial begin
33      we = 1'b0;
34      wa = 5'h01;
35      wd = 32'h66666666;
36      #5
37      we = 1'b1;
38      #10
39      wa = 5'h11;
40      wd = 32'h99999999;
41      #10
42      we = 1'b0;
43      wd = 32'h12345678;
44      #10
45      we = 1'b1;
46      #10
47      we = 1'b0;
48      wa = 5'h01;
49      #10
50      we = 1'b1;
51      #10
52      $finish;
53    end
54  endmodule
55
```

仿真波形



3.2IP_RAM

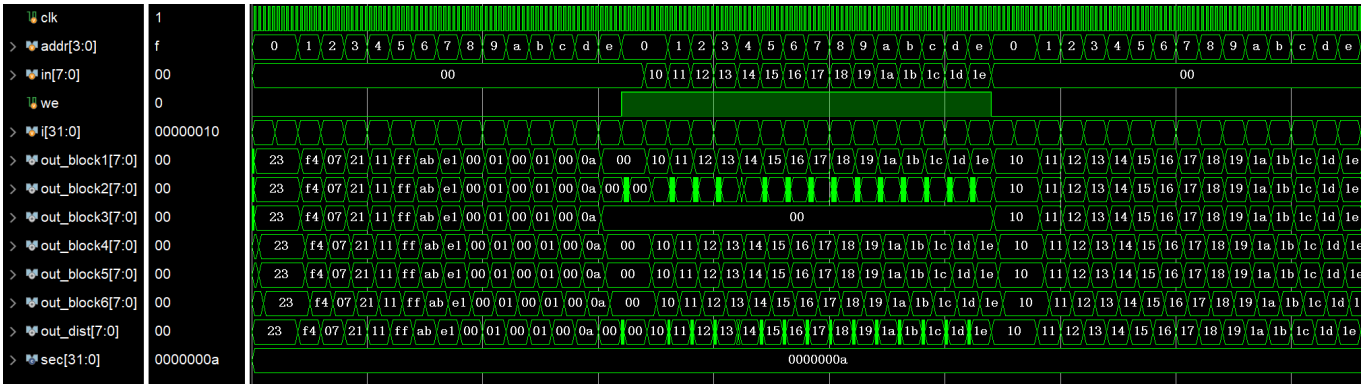
仿真文件

```
1
2  `timescale 1ns / 1ps
3
4  module ip_sim();
5    // shared input signals
6    reg clk;
7    reg [3:0] addr;
8    reg [7:0] in;
9    reg we;
10   parameter sec = 10;
11
12   initial begin
13     clk = 1'b0;
14     forever
15     #1 clk <= ~clk;
16   end
17
18   integer i;
19   initial begin
20
21   addr <= 4'h0;
22   in <= 8'h00;
23   we <= 1'b0;
24   for (i = 0; i < 16; i = i + 1) begin
25     #sec addr <= i;
26   end
27
28   addr <= 4'h0;
29   in <= 8'h00;
30   we <= 1'b1;
31   for (i = 0; i < 16; i = i + 1) begin
32     #sec addr <= i;
33     in <= i + 16'h10;
34   end
35
36   addr <= 4'h0;
37   in <= 8'h00;
38   we <= 1'b0;
39   for (i = 0; i < 16; i = i + 1) begin
40     #sec addr <= i;
41   end
42   #sec $finish;
43 end
```

```
44
45
46  wire [7:0] out_block1;
47  wire [7:0] out_block2;
48  wire [7:0] out_block3;
49  wire [7:0] out_block4;
50  wire [7:0] out_block5;
51  wire [7:0] out_block6;
52
53
54  // Instantiate memory block 1
55  blk1 test_block1(
56      .clka(clk),
57      .addra(addr),
58      .dina(in),
59      .douta(out_block1),
60      .wea(we)
61  );
62
63  // Instantiate memory block 2
64  blk2 test_block2(
65      .clka(clk),
66      .addra(addr),
67      .dina(in),
68      .douta(out_block2),
69      .wea(we)
70  );
71
72  // Instantiate memory block 3
73  blk3 test_block3(
74      .clka(clk),
75      .addra(addr),
76      .dina(in),
77      .douta(out_block3),
78      .wea(we)
79  );
80
81  blk4 test_block4(
82      .clka(clk),
83      .addra(addr),
84      .dina(in),
85      .douta(out_block4),
86      .wea(we)
87  );
88
89  blk5 test_block5(
90      .clka(clk),
```

```
    .addra(addr),
    .dina(in),
    .douta(out_block5),
    .wea(we)
);

blk6 test_block6(
    .clka(clk),
    .addra(addr),
    .dina(in),
    .douta(out_block6),
    .wea(we)
);

// Instantiate distributed memory block
wire [7:0] out_dist;
dist test_dist(
        .clk(clk),
        .a(addr),
        .d(in),
        .we(we),
        .spo(out_dist)
);

endmodul
```

仿真波形



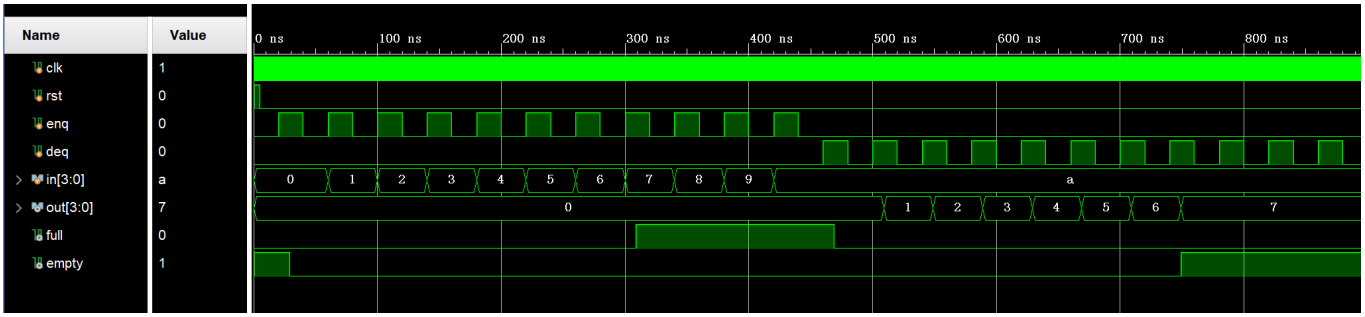
3.3FIFO

仿真文件

```
module fifo_sim();
  reg clk;
  reg rst;
  reg enq;
  reg deq;
  reg [3:0] in;
  wire [3:0] out;
  wire full;
  wire empty;
  fifo test(
    .clk(clk),
    .rst(rst),
    .enq(enq),
    .deq(deq),
    .in(in),
    .out(out),
    .full(full),
    .empty(empty)
  );
  initial begin
    clk <= 1'b0;
    forever
      #1 clk <= ~clk;
  end
  initial begin
    rst <= 1'b1;
    #5 rst <= 1'b0;
  end
  initial begin
    enq <= 1'b0;
    deq <= 1'b0;
    in <= 4'h0;
    for (integer i = 0; i < 9; i = i + 1) begin
      #20 enq <= 1'b1; // enqueue
      in <= i;
      #20 enq <= 1'b0;
    end
    #20 enq <= 1'b1; // invalid enqueue
    in <= 4'h9;
    #20 enq <= 1'b0;
    #20 enq <= 1'b1; // invalid enqueue
    in <= 4'hA;
    #20 enq <= 1'b0;
    for (integer i = 0; i < 9; i = i + 1) begin
      #20 deq <= 1'b1; // dequeue
```

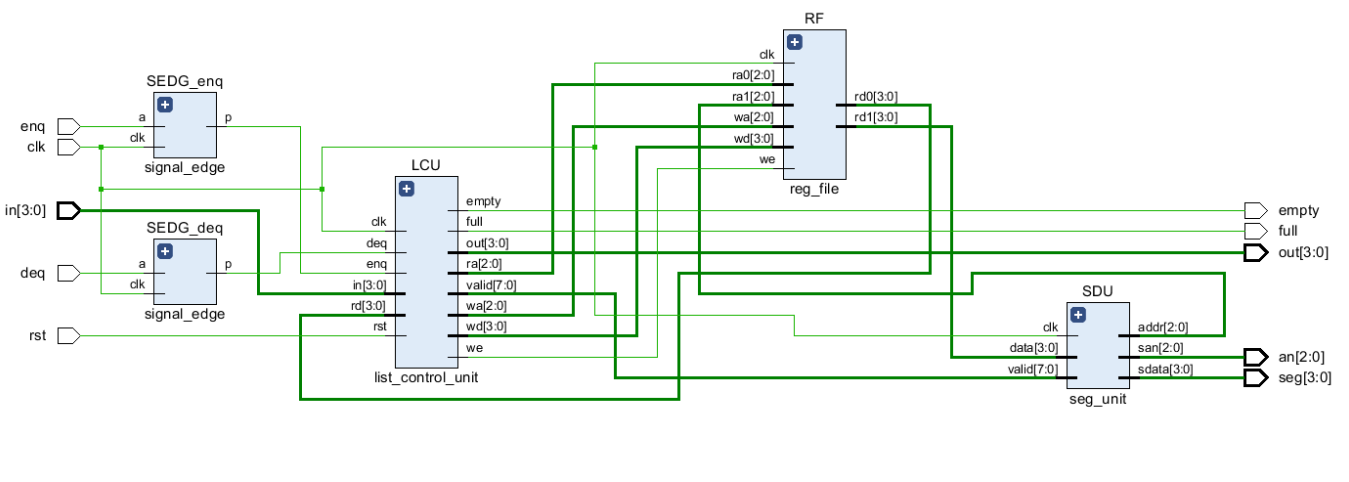
```
46        #20 deq <= 1'b0;
47      end
48      #20 deq <= 1'b1; // invalid dequeue
49      #20 deq <= 1'b0;
50      #20 deq <= 1'b1; // invalid dequeue
51      #20 deq <= 1'b0;
52      #20 $finish;
53    end
54  endmodule
55
```

仿真波形



4.电路设计与分析

FIFO的RTL分析电路



5.IP部分的比较

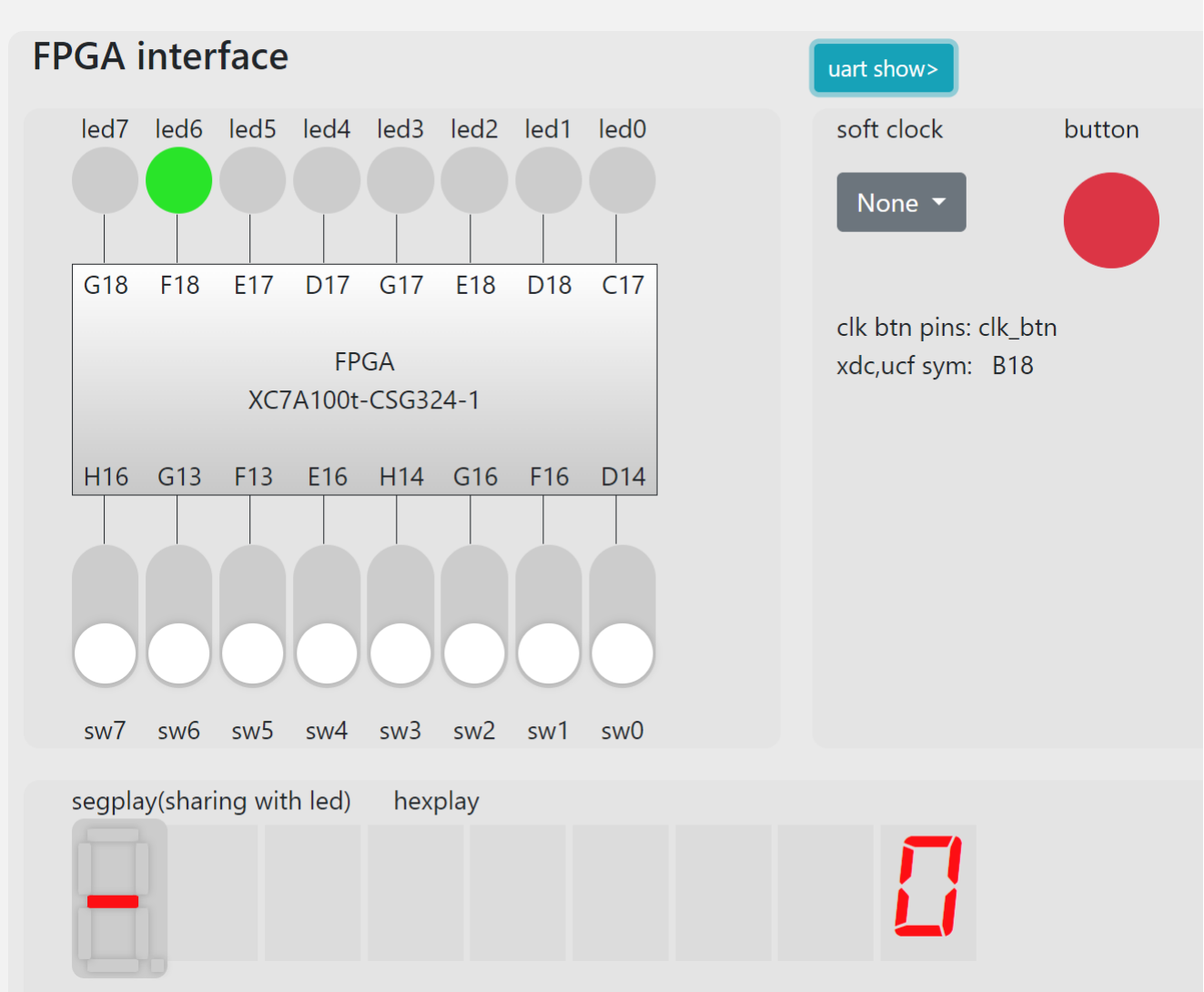
- 分布式RAM读数据的延迟时间小于块式RAM

- 块式RAM采用write first、read first和no change，呈现出的write first优先写入接着读取数据，read first优先读取接着写入数据，显示的块式RAM后者慢于前者，且有关节。no change在写使能信号为1时，不读、写入数据。
- primitive register模式和core register模式时序几乎无差异，但是core register从原理上延迟要小于前者。

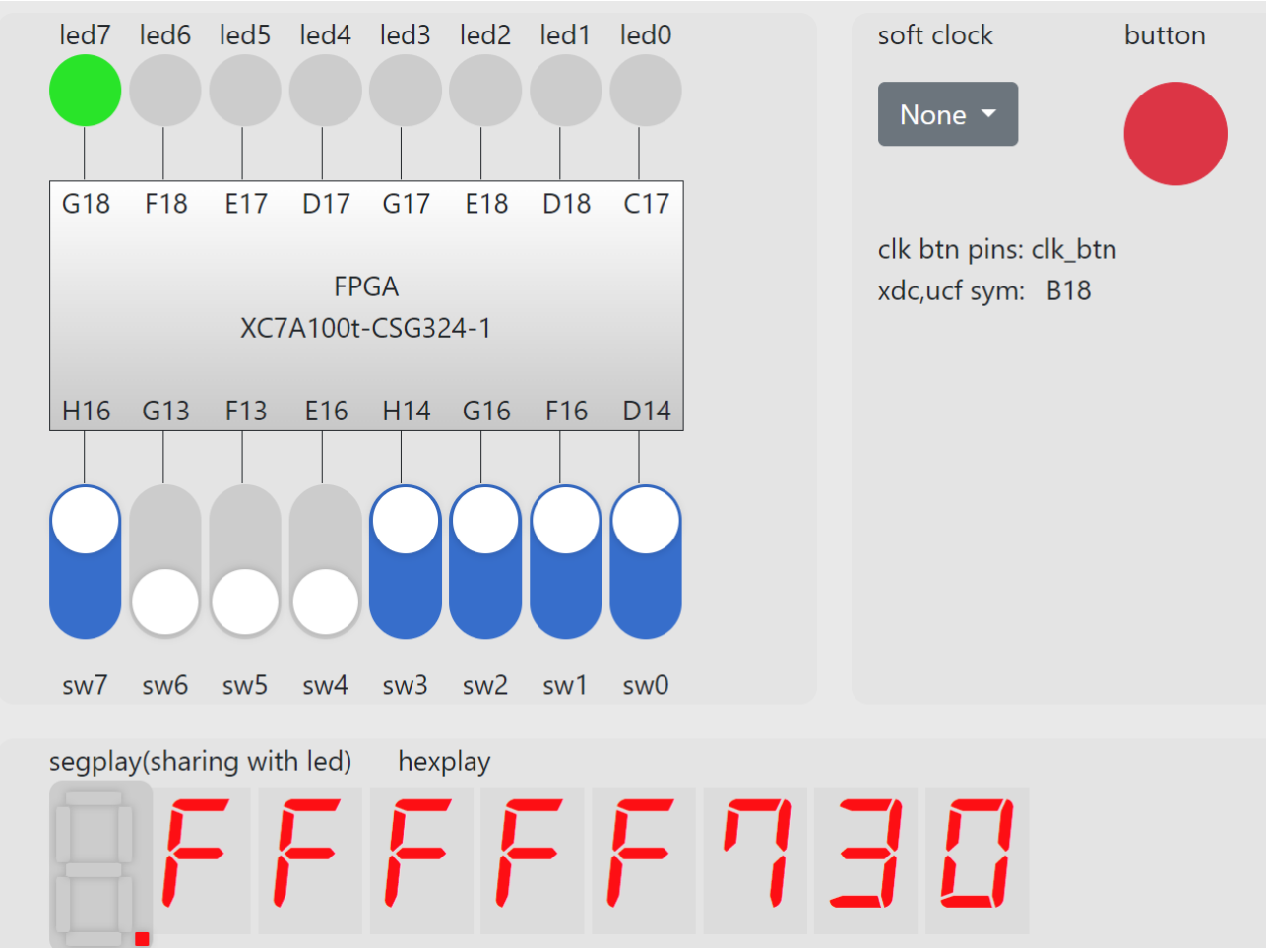
  二者并用则会使得延迟更高。

6.下载测试

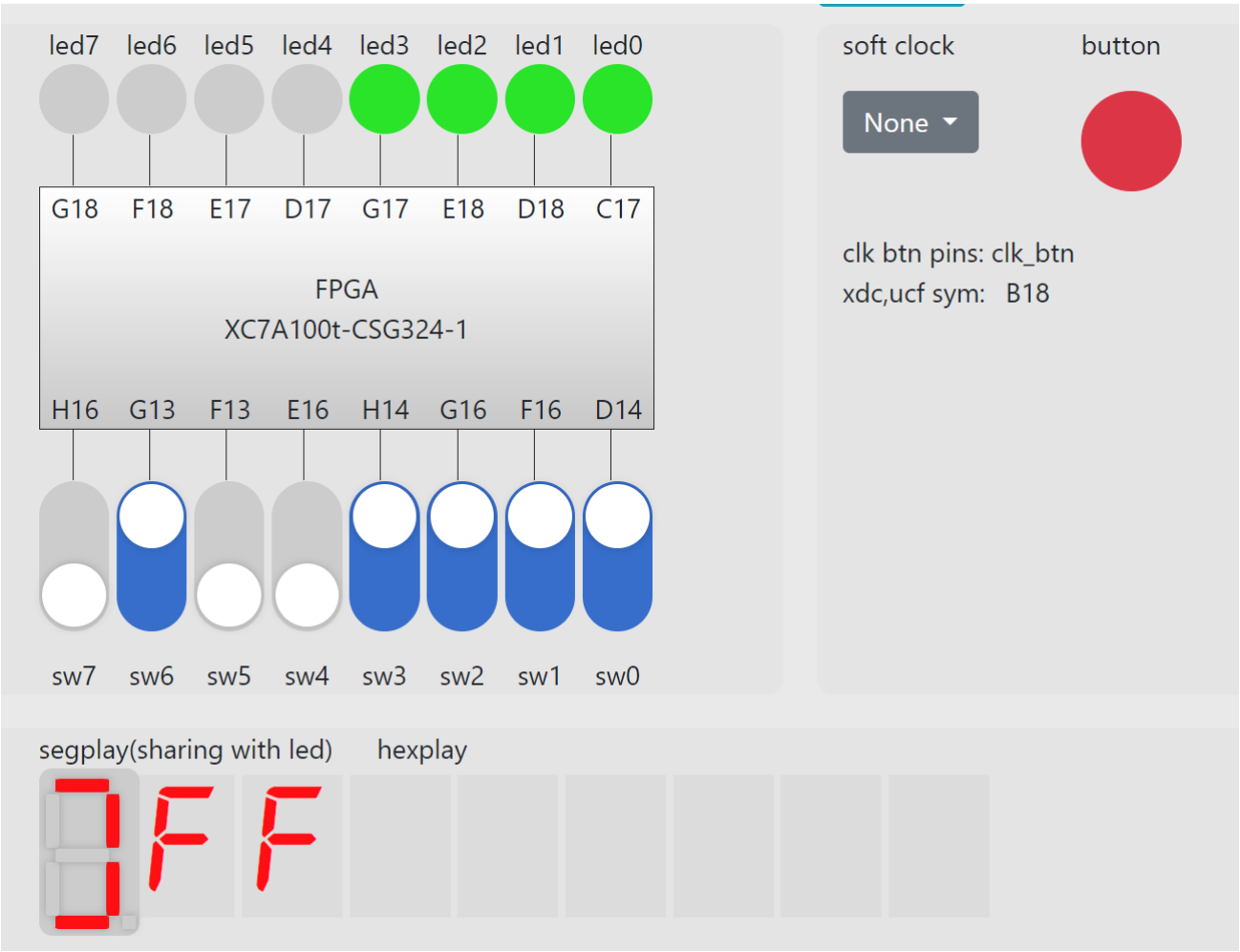
初始：



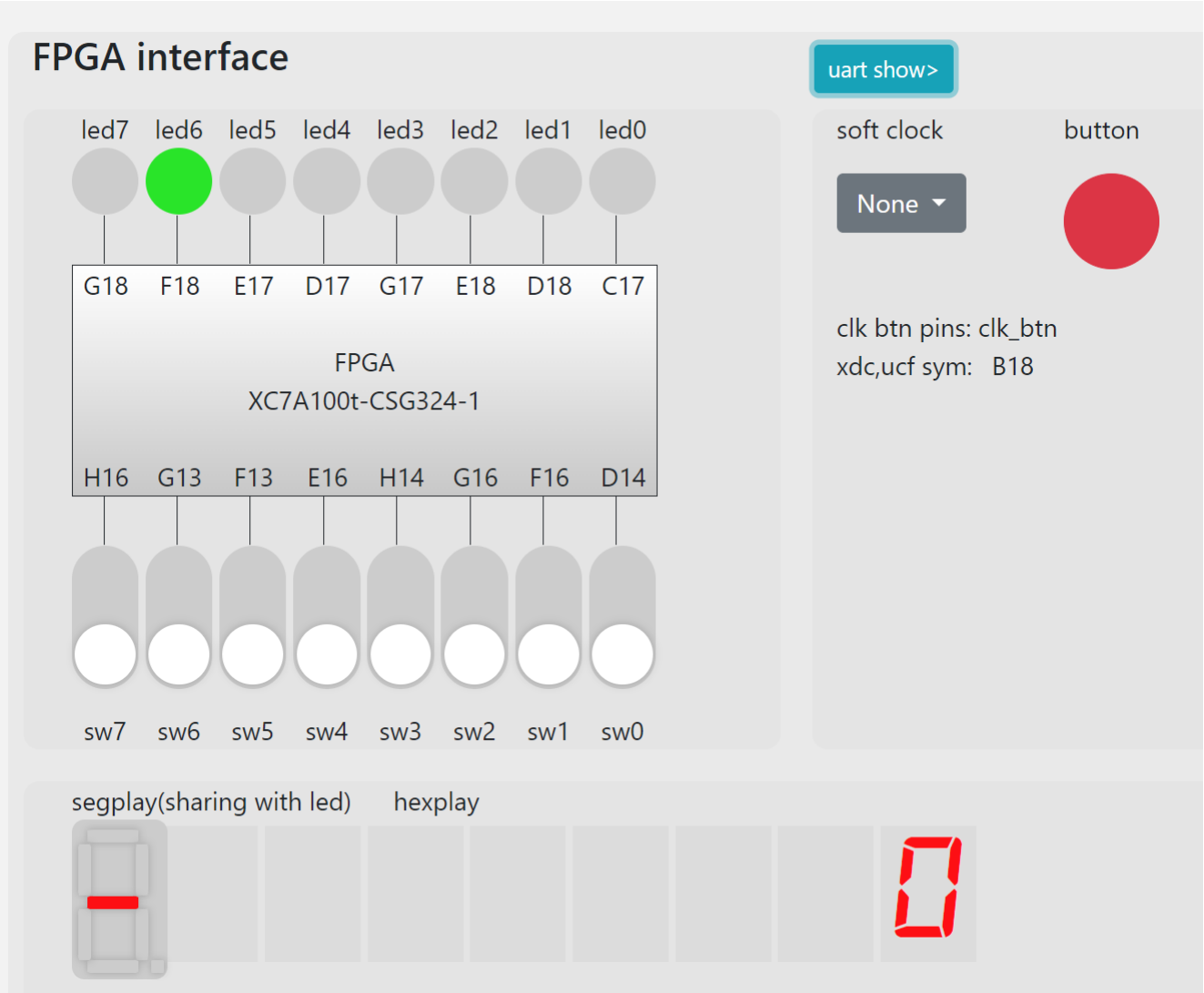
进队


led7 led6 led5 led4 led3 led2 led1 led0
G18 F18 E17 D17 G17 E18 D18 C17
FPGA
XC7A100t-CSG324-1
H16 G13 F13 E16 H14 G16 F16 D14
sw7 sw6 sw5 sw4 sw3 sw2 sw1 sw0
soft clock
button
None
clk btn pins: clk_btn
xdc,ucf sym: B18
segplay(sharing with led)
hexplay
FFFFF730

出队：


led7 led6 led5 led4 led3 led2 led1 led0
G18 F18 E17 D17 G17 E18 D18 C17
FPGA
XC7A100t-CSG324-1
H16 G13 F13 E16 H14 G16 F16 D14
sw7 sw6 sw5 sw4 sw3 sw2 sw1 sw0
soft clock
button
None
clk btn pins: clk_btn
xdc,ucf sym: B18
segplay(sharing with led)
hexplay

置位:



成功!

7.总结收获

收获:

- 对数码管的使用有了清晰的认识
- 学会了寄存器堆的使用和设计
- 了解IP的使用知识，RAM

建议:

- 实验PPT真是极不合理!