

# 计算机组成原理 实验报告

姓名：张艺耀

学号：PB20111630

实验日期：2022-3-22

## 实验题目

寄存器堆与存储器及其应用

## 实验目的

- 掌握寄存器堆和存储器的功能、时序及其应用
- 熟练掌握数据通路和控制器的设计和描述方法

## 实验平台

- FPGAOL
- Vivado
- Mac + VSCode-remote + SSH + VLab

## 实验过程

### Step 1: 行为方式参数化描述寄存器堆 功能仿真

本步骤要求实现一个32\*WIDTH的寄存器并进行功能仿真。

register\_file.v

```
1 module register_file #(parameter WIDTH = 32) (  
2     input clk,  
3     input [4:0] ra0,    //读端口0地址  
4     output [WIDTH - 1:0] rd0,    //读端口0数据  
5     input [4:0] ra1,    //读端口1地址  
6     output [WIDTH - 1:0] rd1,    //读端口1数据  
7     input [4:0] wa,    //写端口地址
```

```

8     input we,    //写使能 高电平有效
9     input [WIDTH - 1:0] wd //写端口数据
10 );
11
12 reg [WIDTH - 1:0] regfile [0:31];
13 assign rd0 = regfile[ra0];
14 assign rd1 = regfile[ra1];
15
16 always @(posedge clk) begin
17     if(we) regfile[wa] <= wd;
18 end
19
20 endmodule

```

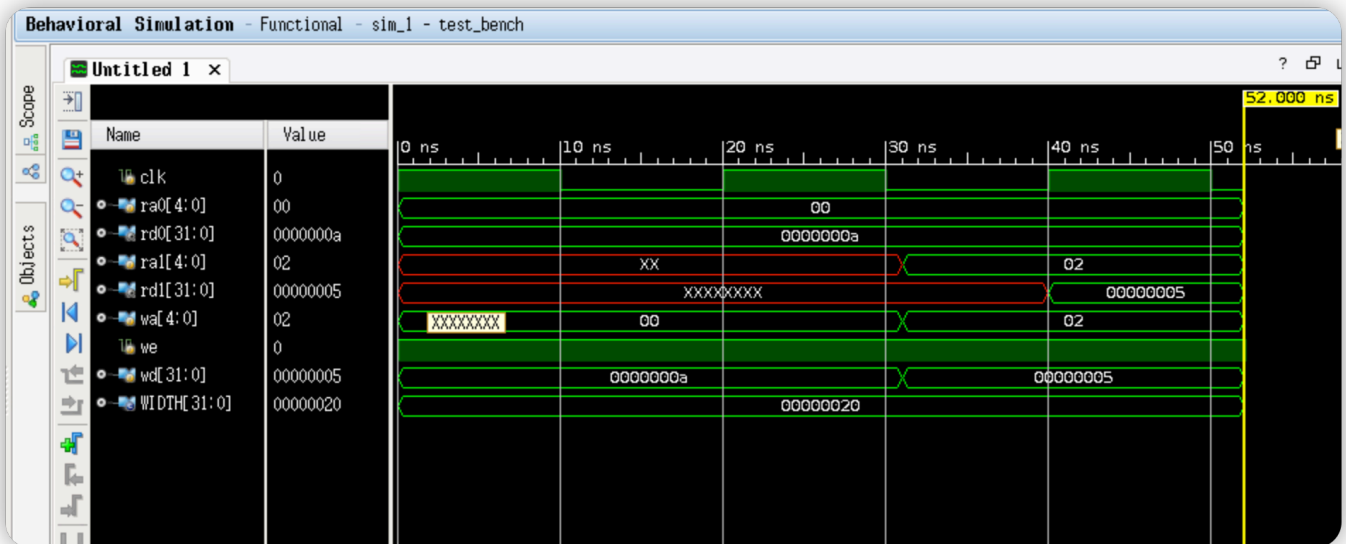
仿真文件test\_bench.v:

```

1 module test_bench #(parameter WIDTH = 32) ();
2 reg clk;
3 reg [4:0] ra0;    //读端口0地址
4 wire [WIDTH - 1:0] rd0;    //读端口0数据
5 reg [4:0] ra1;    //读端口1地址
6 wire [WIDTH - 1:0] rd1;    //读端口1数据
7 reg [4:0] wa; //写端口地址
8 reg we;    //写使能 高电平有效
9 reg [WIDTH - 1:0] wd; //写端口数据
10 register_file register_file(clk, ra0, rd0, ra1, rd1, wa, we, wd);
11
12 initial begin
13     clk = 1'b1;
14     forever #10 clk = ~clk;
15 end
16
17 initial begin
18     we = 1'b1;
19     ra0 = 5'b0;
20     wa = 5'b0;
21     wd = 31'b1010;
22     #31 wa = 5'b10; ra1 = 5'b10; wa = 5'b10; wd = 31'b0101;
23     #21 we = 1'b0; $finish;
24 end
25
26 endmodule

```

仿真结果:



可以看出初始时先向寄存器地址0中写入a（31'b1010） ra0 读出数据；31ns之后向地址2中写入5（31'b0101） ra1读出数据。符合寄存器功能要求。

## Step 2 : IP例化分布式和块式16 \* 8 位单端口RAM 功能仿真和对比

按照实验文档所给步骤操作即可。

先把总的仿真文件test\_bench.v放出来：

```

1  //module test_bench ();
2
3  // RF_8_4.v test
4  /*
5  module test_bench ();
6  reg clk, rst;
7  reg [2:0] ra0;    //读端口0地址
8  wire [3:0] rd0;   //读端口0数据
9  reg [2:0] ra1;    //读端口1地址
10 wire [3:0] rd1;   //读端口1数据
11 reg [2:0] wa; //写端口地址
12 reg we;    //写使能 高电平有效
13 reg [3:0] wd; //写端口数据
14 RF_8_4 RF_8_4(clk,rst, ra0, rd0, ra1, rd1, wa, we, wd);
15
16 initial begin
17     rst = 0;
18     clk = 1'b1;
19     forever #5 clk = ~clk;
20 end
21
22 initial begin
23     we = 1'b1;
24     ra0 = 5'b0;
25     wa = 5'b0;

```

```

26     wd = 31'b1010;
27     #31 wa = 5'b10; ra1 = 5'b10; wa = 5'b10; wd = 31'b0101;
28     #21 we = 1'b0; #20 rst = 1;
29     #20 $finish;
30 end
31
32 endmodule
33 */
34
35 //blk_mem_gen_0.v test
36 /*
37 module test_bench ();
38 reg clka;
39 reg ena;
40 reg wea;
41 reg [3:0] addra;
42 reg [7:0] dina;
43 wire [7:0] douta;
44 blk_mem_gen_0 blk_mem_gen_0(clka, ena, wea, addra, dina, douta);
45
46 initial begin
47     clka = 1'b1;
48     forever #5 clka = ~clka;
49 end
50
51 initial begin
52     ena = 1'b1;
53     wea = 1'b0;
54     addra = 1'b0;
55     while (addra < 15) begin
56         #10 addra = addra + 1'b1;
57     end
58     wea = 1'b1;
59     addra = 4'hf;
60     dina = 8'b0;
61     #10 wea = 1'b0; addra = 4'h0; #20 $finish;
62 end
63
64 endmodule
65
66 */
67
68 //dist_mem_gen_0.v test
69 /*
70 module test_bench ();
71 reg [3:0] a;    //端口地址
72 reg [7:0] d;    //端口数据
73 reg clk;
74 reg we;
75 wire [7:0] spo;
76 dist_mem_gen_0 dist_mem_gen_0(a, d, clk, we, spo);
77

```

```

78 initial begin
79     clk = 1'b1;
80     forever #10 clk = ~clk;
81 end
82
83 initial begin
84     we = 1'b0;
85     a = 1'b0;
86     while (a < 15) begin
87         #10 a = a + 1'b1;
88     end
89     we = 1'b1;
90     a = 4'hf;
91     d = 8'b0;
92     #10 we = 1'b0; a = 4'h0; #20 $finish;
93 end
94
95 endmodule
96
97 /*
98 //register_file.v test
99 /*
100 module test_bench #(parameter WIDTH = 32) ();
101 reg clk;
102 reg [4:0] ra0;    //读端口0地址
103 wire [WIDTH - 1:0] rd0;    //读端口0数据
104 reg [4:0] ra1;    //读端口1地址
105 wire [WIDTH - 1:0] rd1;    //读端口1数据
106 reg [4:0] wa; //写端口地址
107 reg we;    //写使能 高电平有效
108 reg [WIDTH - 1:0] wd; //写端口数据
109 register_file register_file(clk, ra0, rd0, ra1, rd1, wa, we, wd);
110
111 initial begin
112     clk = 1'b1;
113     forever #10 clk = ~clk;
114 end
115
116 initial begin
117     we = 1'b1;
118     ra0 = 5'b0;
119     wa = 5'b0;
120     wd = 31'b1010;
121     #31 wa = 5'b10; ra1 = 5'b10; wa = 5'b10; wd = 31'b0101;
122     #21 we = 1'b0; $finish;
123 end
124
125 endmodule
126 */

```

## 分布式：

dist\_mem\_gen\_0\_stub.v

```
1 module dist_mem_gen_0(a, d, clk, we, spo)
2 /* synthesis syn_black_box black_box_pad_pin="a[3:0],d[7:0],clk,we,spo[7:0]" */;
3   input [3:0]a;
4   input [7:0]d;
5   input clk;
6   input we;
7   output [7:0]spo;
8 endmodule
```

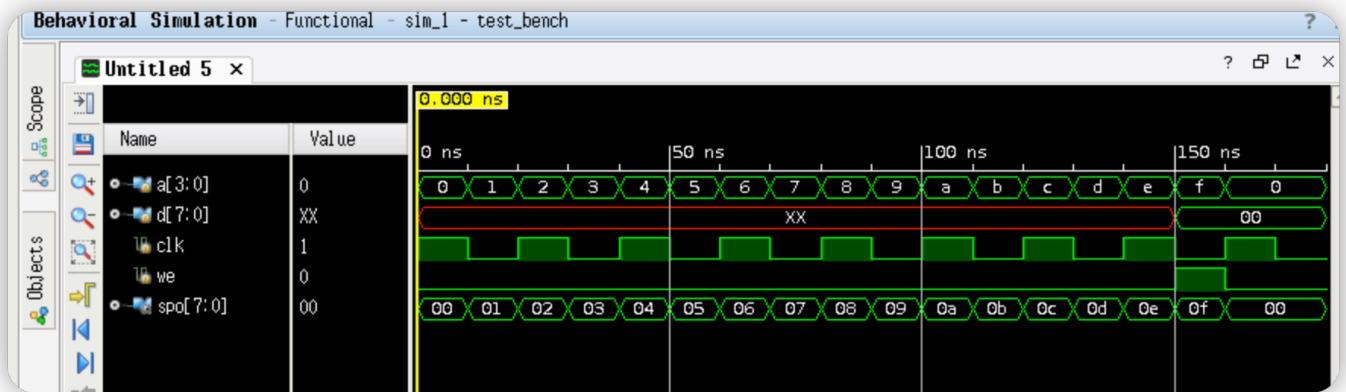
test\_bench.v

```
1 module test_bench ();
2   reg [3:0] a;    //端口地址
3   reg [7:0] d;    //端口数据
4   reg clk;
5   reg we;
6   wire [7:0] spo;
7   dist_mem_gen_0 dist_mem_gen_0(a, d, clk, we, spo);
8
9   initial begin
10     clk = 1'b1;
11     forever #10 clk = ~clk;
12   end
13
14   initial begin
15     we = 1'b0;
16     a = 1'b0;
17     while (a < 15) begin
18       #10 a = a + 1'b1;
19     end
20     we = 1'b1;
21     a = 4'hf;
22     d = 8'b0;
23     #10 we = 1'b0; a = 4'h0; #20 $finish;
24   end
25
26 endmodule
```

reg1.coe

```
1 memory_initialization_radix = 16;
2 memory_initialization_vector =
3 0 1 2 3 4 5 6 7 8 9 a b c d e f;
```

仿真结果：



块式：

blk\_mem\_gen\_0\_stub.v

```

1 module blk_mem_gen_0(clka, ena, wea, addra, dina, douta)
2   /* synthesis syn_black_box
   black_box_pad_pin="clka,ena,wea[0:0],addra[3:0],dina[7:0],douta[7:0]" */;
3   input clka;
4   input ena;
5   input [0:0]wea;
6   input [3:0]addra;
7   input [7:0]dina;
8   output [7:0]douta;
9 endmodule

```

test\_bench.v

```

1 module test_bench ();
2   reg clka;
3   reg ena;
4   reg wea;
5   reg [3:0] addra;
6   reg [7:0] dina;
7   wire [7:0] douta;
8   blk_mem_gen_0 blk_mem_gen_0(clka, ena, wea, addra, dina, douta);
9
10  initial begin
11    clka = 1'b1;
12    forever #10 clka = ~clka;
13  end
14
15  initial begin
16    ena = 1'b1;
17    wea = 1'b0;
18    addra = 1'b0;
19    while (addra < 15) begin
20      #10 addra = addra + 1'b1;
21    end
22    wea = 1'b1;

```

```

23     addra = 4'hf;
24     dina = 8'b0;
25     #10 wea = 1'b0; addra = 4'h0; #20 $finish;
26 end
27
28 endmodule

```

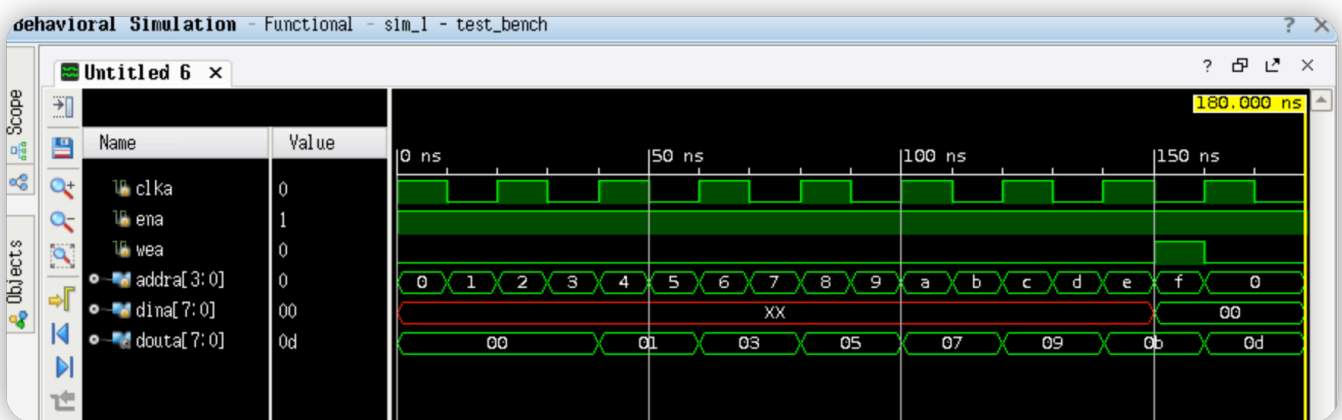
reg1.coe

```

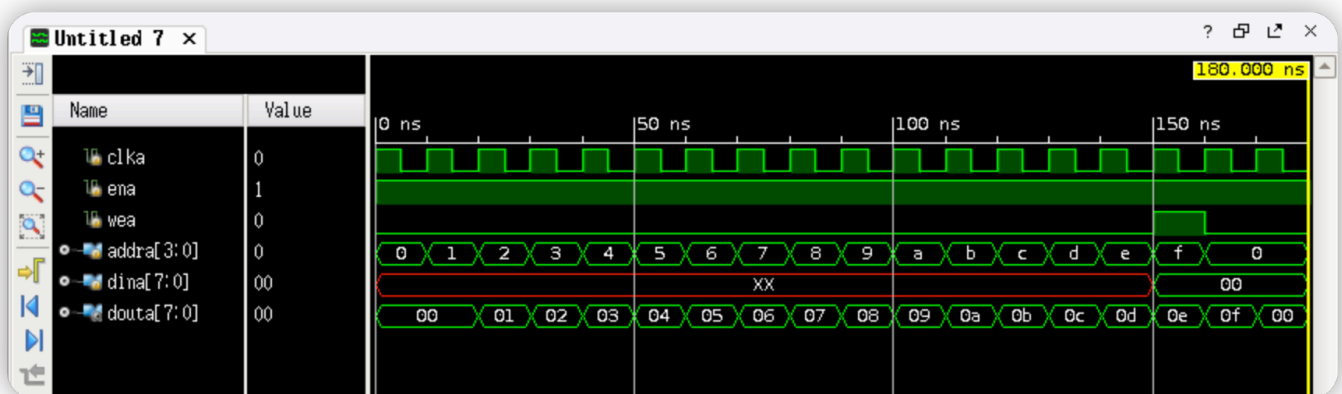
1 memory_initialization_radix = 16;
2 memory_initialization_vector =
3 0 1 2 3 4 5 6 7 8 9 a b c d e f;

```

仿真结果：



此时意识到时钟周期不合适。将时钟频率更改为原来的两倍 `#5 clka = ~clka` 得到的仿真波形图如下：可以看出douta在下一个时钟周期更新读入上一个地址里的值。



由此可以发现BRAM和DRAM的区别：**BRAM读、写均和时钟同步，而DRAM写和时钟同步，读不需要时钟。**



## Step3 : FIFO队列

fifo.v :

fifo队列主模块 将三个分模块联结起来。

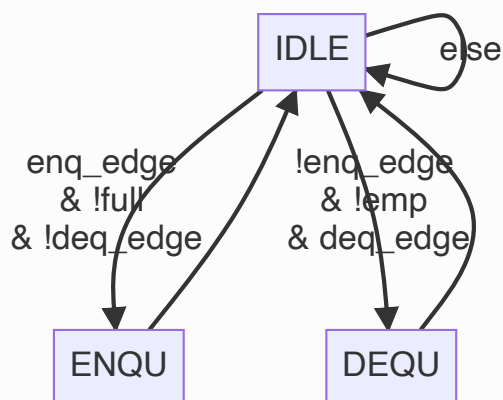
```
1  module fifo (
2      input clk, rst, enq,    //时钟 (上升沿有效)  同步复位 (高电平有效)  入队列使能 (高电平有效)
3      input [3:0] in, //入队列数据
4      input deq,    //出队列使能 (高电平有效)
5      output [3:0] out,    //出队列数据
6      output [2:0] an,    //数码管选择
7      output [3:0] hexplay_data,    //数码管数据
8      output full, emp
9  );
10
11  wire [2:0] ra0; wire [2:0] ra1;
12  wire [3:0] rd0; wire [3:0] rd1;
13  wire [2:0] wa; wire we; wire [3:0] wd; wire [7:0] valid;
14
15  lcu lcu(in, enq, deq, clk, rst, rd0, out, full, emp, ra0, wa, we, wd, valid);
16  RF_8_4 RF_8_4(clk, rst, ra0, rd0, ra1, rd1, wa, we, wd);
17  sdu sdu(rd1, valid, clk, emp, ra1, an, hexplay_data);
18
19  endmodule
```

lcu.v :

队列控制单元lcu。

设置了三个状态，分别是**闲置模式**、**入队列**和**出队列**。维护两个指针head和tail用来指示队头队尾，两个变量full和emp用来指示队列是否满/空：`full <= (tail + 1'b1 == head) ? 1'b1 : 1'b0;` 和 `emp <= (tail == head + 1'b1) ? 1'b1 : 1'b0;`。34~39行是取开关上升沿的模块，之后采用三段式FSM实现主体逻辑功能。

状态图：



```

1  module lcu (
2      input [3:0] in,
3      input enq, deq, clk, rst,
4      input [3:0] rd0,
5      output [3:0] out,
6      output reg full, emp,
7      output reg [2:0] ra0,
8      output reg [2:0] wa,
9      output reg we,
10     output reg [3:0] wd,
11     output reg [7:0] valid
12 );
13
14 reg [2:0] head;
15 reg [2:0] tail;
16
17 parameter IDLE = 2'b00;
18 parameter ENQU = 2'b01;
19 parameter DEQU = 2'b10;
20
21 reg [1:0] cs;
22 reg [1:0] ns;
23 reg tmp_1;
24 reg tmp_2;
25 reg enq_edge;
26 reg deq_edge;
27
28 initial begin
29     cs <= 2'b0;
30     ra0 <= 3'b0;
31     valid <= 8'b0;
32 end
33
34 always @(posedge clk) begin
35     tmp_1 <= enq;
36     enq_edge <= enq & ~tmp_1;
37     tmp_2 <= deq;
38     deq_edge <= deq & ~tmp_2;
39 end
40
41 always @(*) begin
42     case(cs)
43         IDLE: begin
44             if(enq_edge & !full & !deq_edge) ns = ENQU;
45             else if(!enq_edge & !emp & deq_edge) ns = DEQU;
46             else ns = IDLE;
47         end
48         ENQU: begin
49             ns = IDLE;
50         end

```

```

51     DEQU: begin
52         ns = IDLE;
53     end
54     default: ns = IDLE;
55 endcase
56 end
57
58 always @(posedge clk) begin
59     if(rst) begin
60         head <= 3'b0;
61         tail <= 3'b0;
62         valid <= 8'b0;
63         emp <= 1'b1;
64         full <= 1'b0;
65     end
66     else begin
67         case(cs)
68             ENQU: begin
69                 we <= 1'b1;
70                 wa <= tail;
71                 wd <= in;
72                 valid[tail] <= 1'b1;
73                 tail <= tail + 1'b1;
74                 full <= (tail + 1'b1 == head) ? 1'b1 : 1'b0;
75                 emp <= 1'b0;
76             end
77             DEQU: begin
78                 ra0 <= head;
79                 valid[head] <= 1'b0;
80                 head <= head + 1'b1;
81                 emp <= (tail == head + 1'b1) ? 1'b1 : 1'b0;
82                 full <= 1'b0;
83             end
84             default: begin
85                 we <= 1'b0;
86             end
87         endcase
88     end
89 end
90
91 always @(posedge clk) begin
92     if(rst) cs <= IDLE;
93     else cs <= ns;
94 end
95
96 assign out = rd0;
97
98 endmodule

```

RF\_8\_4.v:

与之前的寄存器模块相比只增加了一个rst输入。

```
1 module RF_8_4 (
2     input clk, rst,
3     input [2:0] ra0,    //读端口0地址
4     output [3:0] rd0,   //读端口0数据
5     input [2:0] ra1,    //读端口1地址
6     output [3:0] rd1,   //读端口1数据
7     input [2:0] wa, //写端口地址
8     input we,    //写使能 高电平有效
9     input [3:0] wd //写端口数据
10 );
11
12 reg [3:0] regfile [0:7];
13 assign rd0 = regfile[ra0];
14 assign rd1 = regfile[ra1];
15 integer i;
16
17 always @(posedge clk) begin
18     if(rst) begin
19         for(i = 0; i <= 7; i = i + 1) begin
20             regfile[i] <= 4'b0;
21         end
22     end
23     else if(we) regfile[wa] <= wd;
24 end
25
26 endmodule
```

sdu.v:

添加了中间变量cnt以使valid为0时对应an位不显示数字。（这里本人进行了许多次尝试 即注释的内容。如果仅仅将an设置为线网类型变量并且直接用 `assign an = ra1` 和 `assign hexplay_data = (valid[ra1]) ? rd1 : x0;` 的话会导致队列以外的位置全为0而非空，这种情况下只有可以设置高阻态输出才是可行的；而如果使用 `assign an = ra1 * valid[ra1]` 的话会导致第0位显示值在队列满之前始终为0、在队列即将满时闪烁，在队列已满时才能正常显示。这是因为当valid中有0时赋给an[0]的值是不确定的）

最后多加了一个emp变量用于判断队列空时将输出置0（否则会显示留在队列里的数字）。

```
1 /*
2 module sdu (
3     input [3:0]rd1,//read data
4     input [7:0]valid,
5     input clk,
6     output [2:0]ra1,//read address
7     output [2:0]an,//segment address
8     output [3:0]hexplay_data//segment data
9 );
10
```

```

11 reg [23:0] count;
12 wire [3:0] x0;
13 //wire [3:0] x1;
14
15 assign x0 = 4'h0;
16 //assign an = (valid[ra1]) ? ra1 : x1;
17 assign an = ra1;
18 assign ra1 = count[15:13];
19
20 always @(posedge clk) begin
21     count <= count + 1;
22 end
23
24 assign hexplay_data = (valid[ra1]) ? rd1 : x0;
25
26 endmodule
27 */
28
29 module sdu (
30     input [3:0] rd1,
31     input [7:0] valid,
32     input clk, emp,
33     output [2:0] ra1,
34     output reg [2:0] an,
35     output reg [3:0] hexplay_data
36 );
37
38 reg [32:0] hexplay_cnt;
39 reg [2:0] cnt;
40
41 initial begin
42     hexplay_cnt <= 0;
43     cnt <= 0;
44 end
45
46 always @(*) begin
47     if(emp) hexplay_data = 0;
48     else hexplay_data = rd1;
49 end
50
51 always@(posedge clk) begin
52     if (hexplay_cnt >= (2000000/8)) hexplay_cnt <= 0;
53     else hexplay_cnt <= hexplay_cnt + 1;
54 end
55
56 always@(posedge clk) begin
57     if (hexplay_cnt == 0) begin
58         if(cnt == 7) cnt <= 0;
59         else cnt <= cnt + 1;
60
61         if(emp) an <= 0;
62         else if(valid[cnt]) an <= cnt;

```

```

63     end
64 end
65
66 assign ra1 = an;
67
68 endmodule

```

## Icu改进

由于为了节省代码行数在FSM状态转换的过程中省略了ENQU状态和DEQU状态之后对自身的转换导致可能的连续执行出入队列操作不成功的问题，这样会导致在出入队列之后的一个时钟周期无法进行出入队列（但是在现实中这种问题可以忽略，因为实际操作开关的时间远大于时钟周期）

改进：

```

1  always @(*) begin
2      case(cs)
3          IDLE: begin
4              if(enq_edge & !full & !deq_edge) ns = ENQU;
5              else if(!enq_edge & !emp & deq_edge) ns = DEQU;
6              else ns = IDLE;
7          end
8          ENQU: begin
9              if(enq_edge & !full & !deq_edge) ns = ENQU;
10             else if(!enq_edge & !emp & deq_edge) ns = DEQU;
11             else ns = IDLE;
12         end
13         DEQU: begin
14             if(enq_edge & !full & !deq_edge) ns = ENQU;
15             else if(!enq_edge & !emp & deq_edge) ns = DEQU;
16             else ns = IDLE;
17         end
18         default: ns = IDLE;
19     endcase
20 end

```

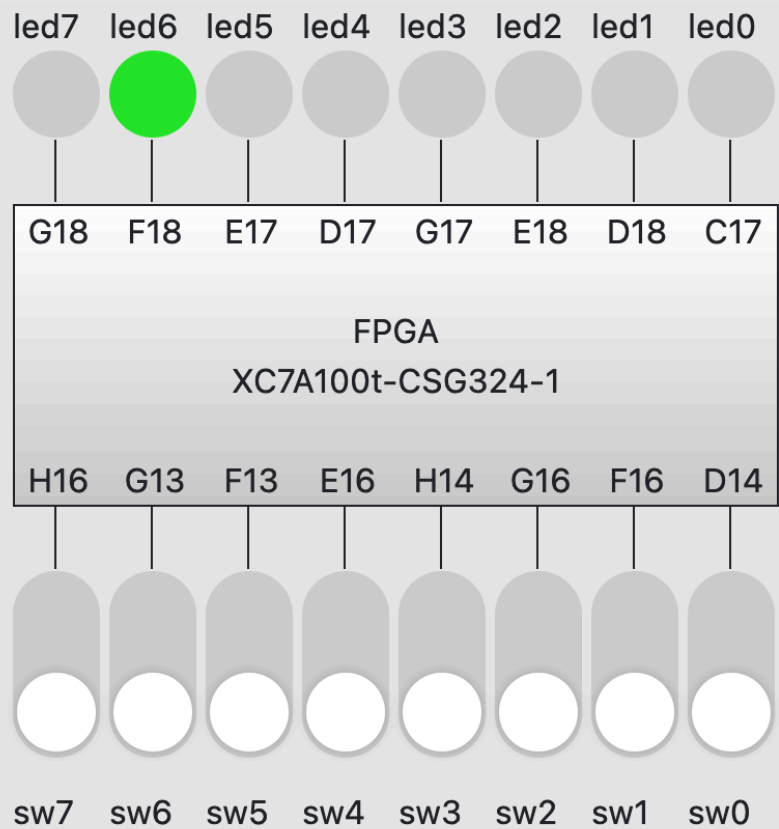
## 实验结果

仿真结果已经在实验过程中贴出。

fifo.bit烧写到FPGAOL上的结果：

初态：

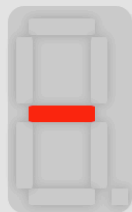
# FPGA interface



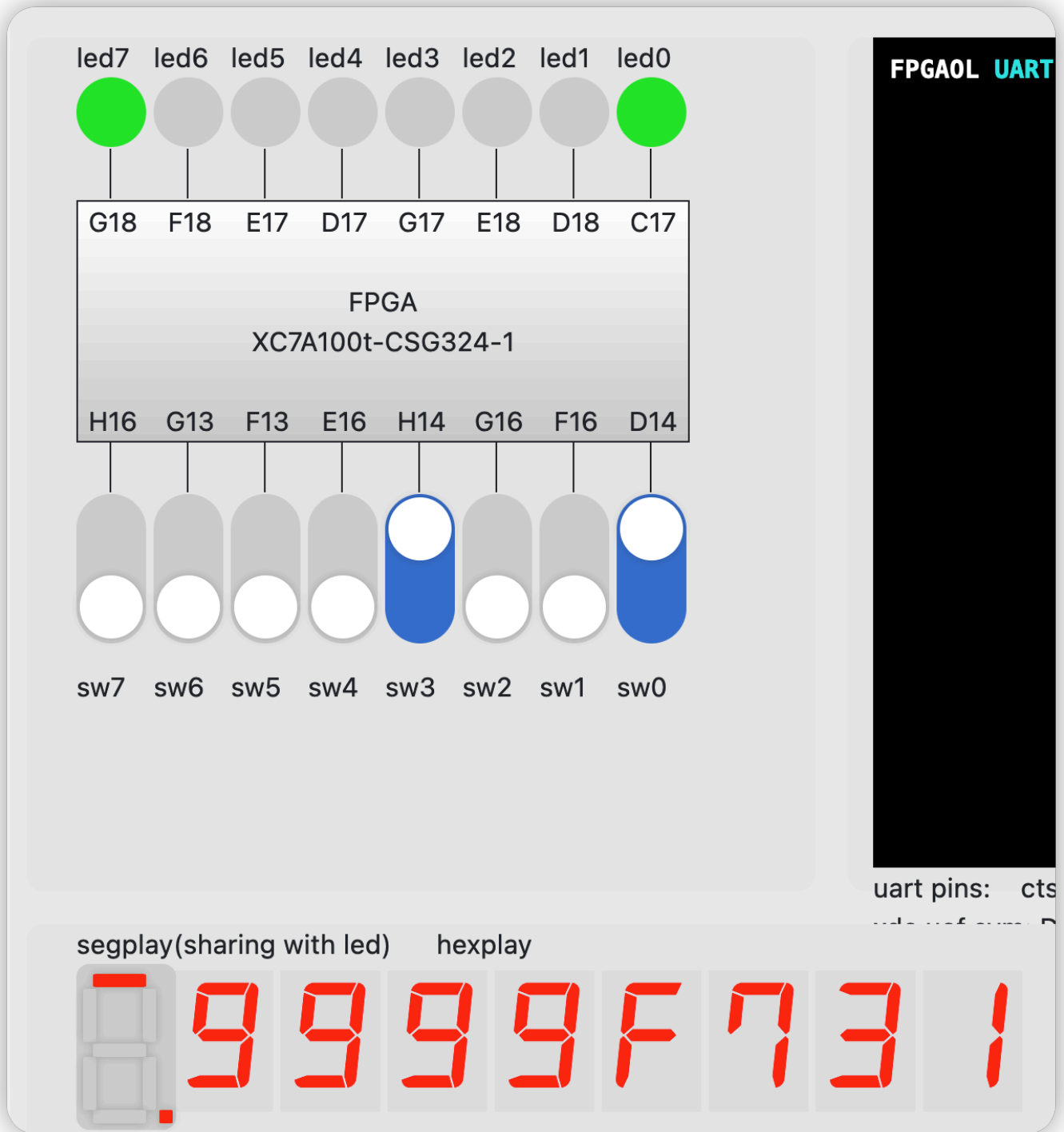
FPGA0L UART

uart pins: cts  
rx tx rts

segplay(sharing with led) hexplay

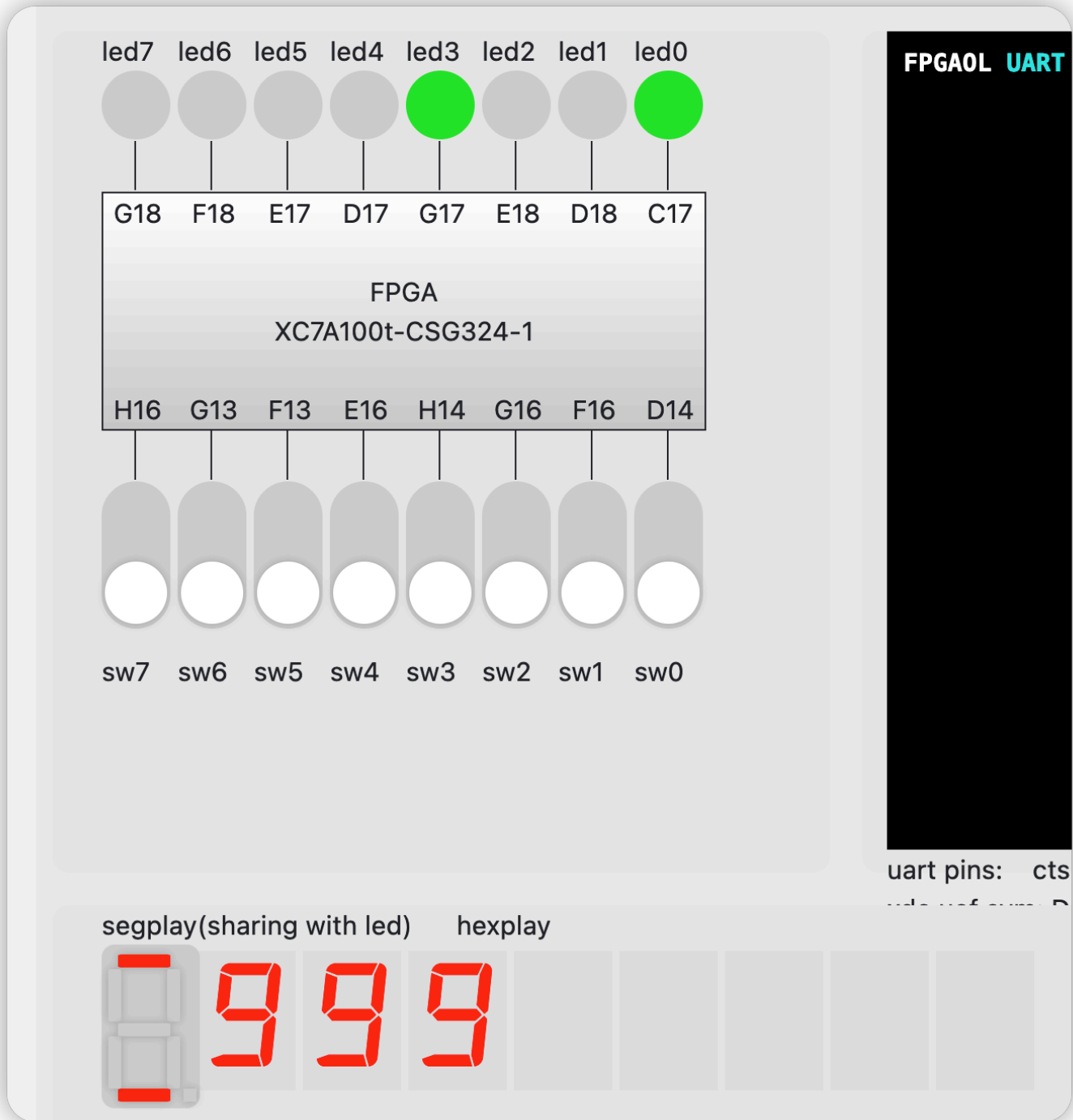


137F9999 入队列直至队列满





出队列



# 心得体会

实验的逻辑很简单，但是sdu的编写很让人头痛。

由于一开始不知道怎么样才能实现出队列时数码管上不显示值走了很多弯路。但是由于数码管总是有一个是亮的 这样还是无法完全做到队列空时8个管子完全没有值显示。

不过总地来说加深了自己对状态机的理解并提升了自己对一些显示bug的处理能力。