实验二 寄存器堆与队列

noname

2020 年 5 月 15 日

1 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

2 逻辑设计

2.1 寄存器堆 (Register File)

逻辑较为简单: 定义 NUM 个 $WIDTH$ 位的寄存器本体, 声明 **1** 个同步写和 **2** 个异步读端口
同步写在 $always@(posedgeclk)$ 内在写使能有效时对寄存器堆内对应写地址的寄存器写入写端口数据
异步读直接通过 $assign$ 直接将对应地址寄存器内容赋给读端输出
编写 **Register File** 代码并进行功能仿真
testbench 代码在附件内与源码一同给出

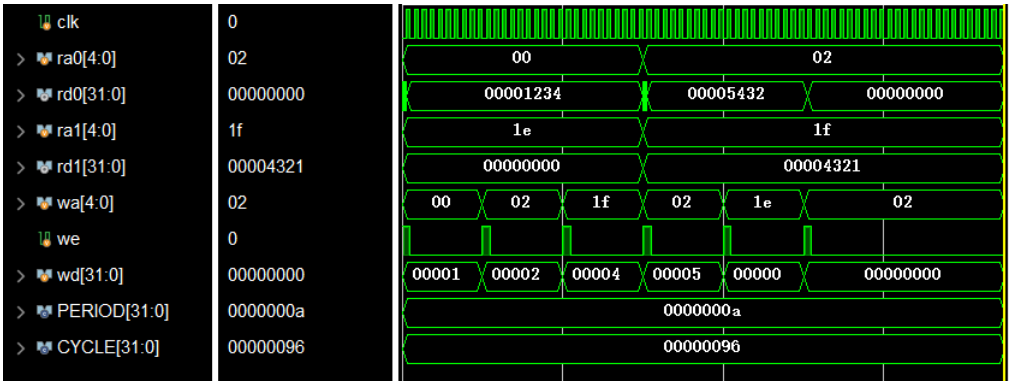
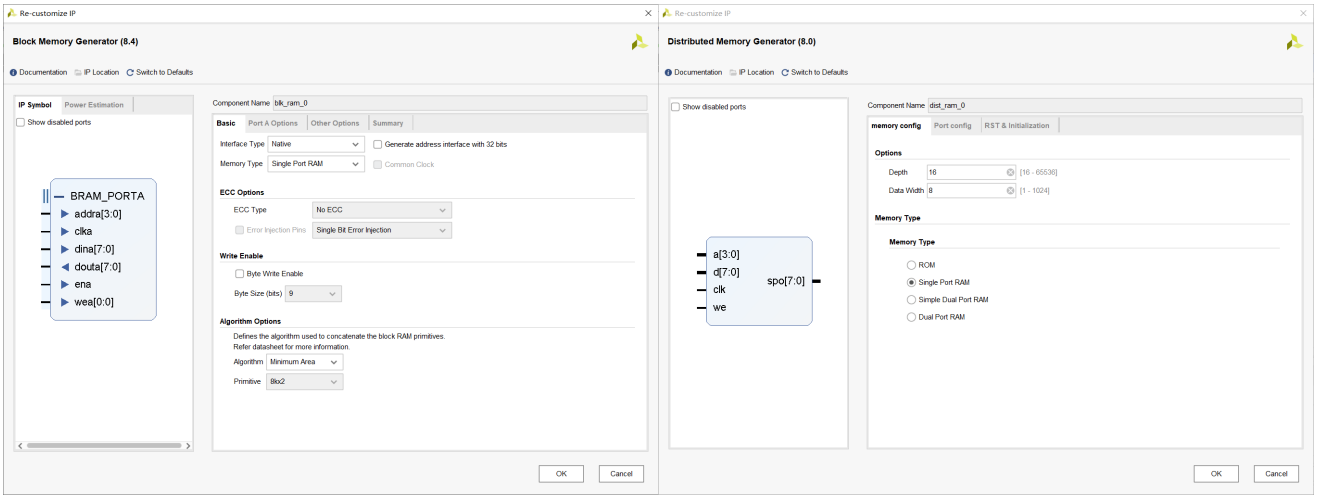


图 1: 寄存器堆仿真波形图

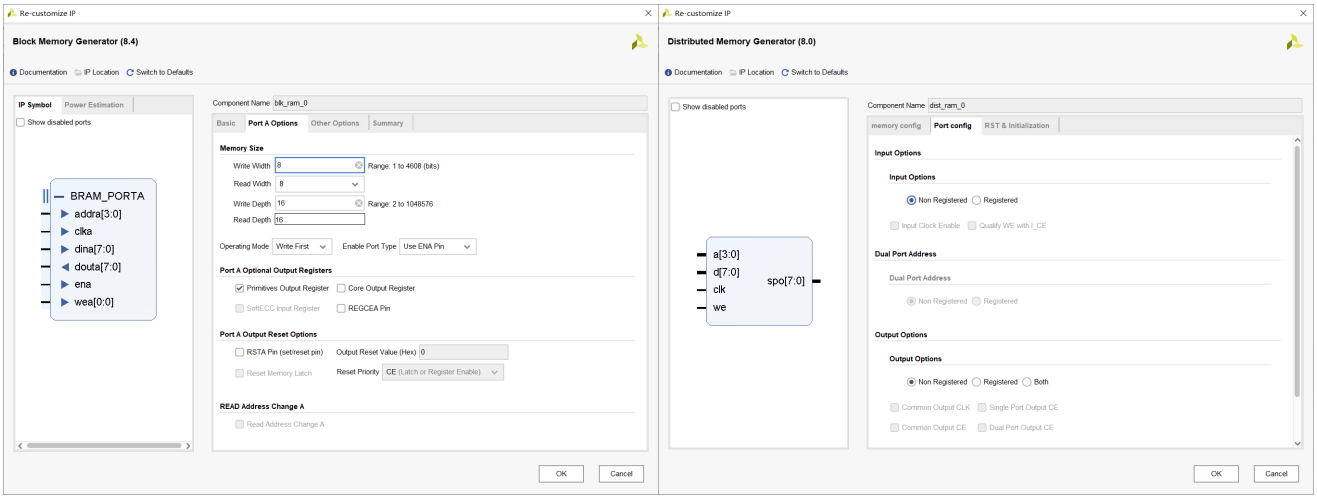
2.2 存储器

展示单端口块式 **RAM** 和分布式 **RAM** 的例化过程



(a) BRAMstep1 (b) DRAMstep1

图 2: STEP1



(a) BRAMstep2 (b) DRAMstep2

图 3: STEP2

bram 和 **dram** 的区别为 **bram** 输出需要时钟，而 **dram** 给出地址后即可输出数据
bram 的储存空间较大，而 **dram** 是由逻辑单元拼出的，浪费 **LUT** 资源
仿真波形如下，可看出 **bram** 与 **dram** 输出的不同之处

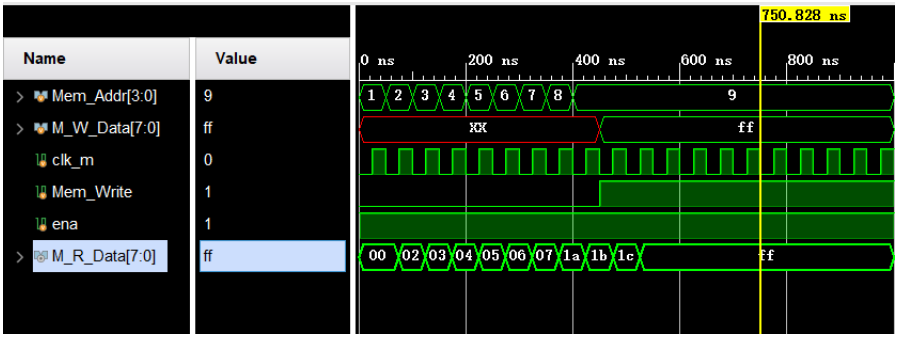


图 4: bram 仿真波形图

图 5: dram 仿真波形图

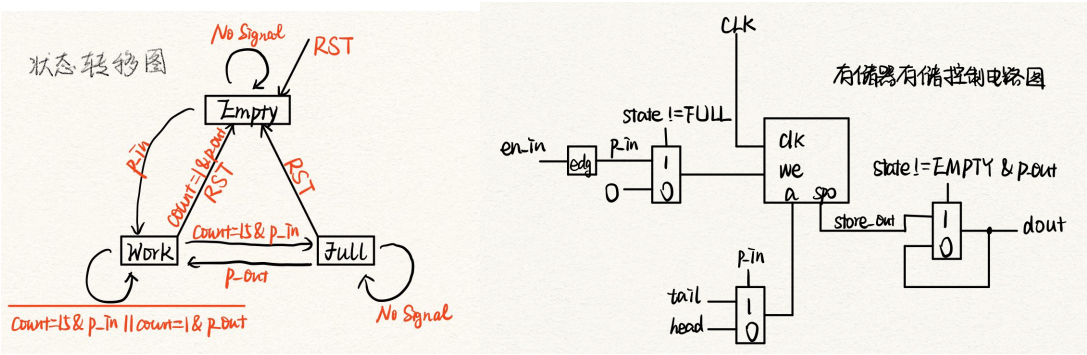
2.3 先进先出 (FIFO) 队列

利用例化的 **IP** 和适当逻辑电路, 实现 $LENGTH = 16, WIDTH = 8$ 的 **FIFO** 队列
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**
count 指示队列中数据个数, 队满时不能执行入队操作, 队空时不能进行出队操作
入队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

2.3.1 取边沿电路

为保证出入队使能的一次有效持续期间，仅允许出入队一个数据，需对 **en_in,en_out** 取边沿

2.3.2 数据通路与状态转移图



(a) 控制器状态转移图 (b) 存储器输入输出电路图

图 6: NONAME

2.3.3 仿真结果

首先正常入队至队满，再出队，检查波形图正确性

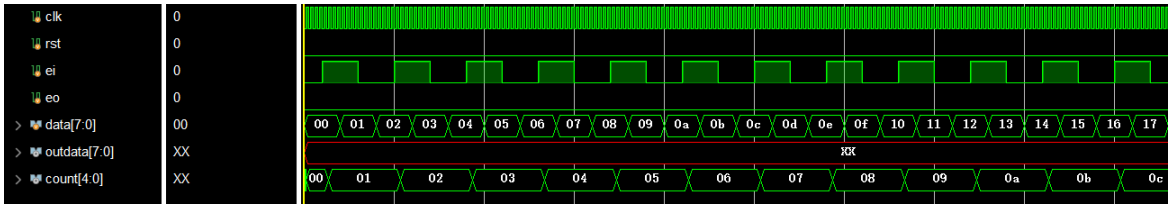


图 7: 入队数据 (队未满)

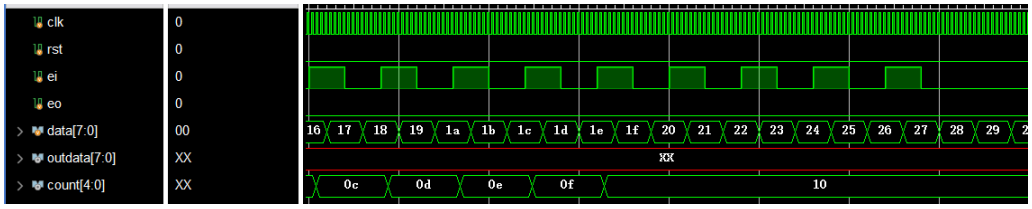


图 8: 继续入队数据 (至队满)

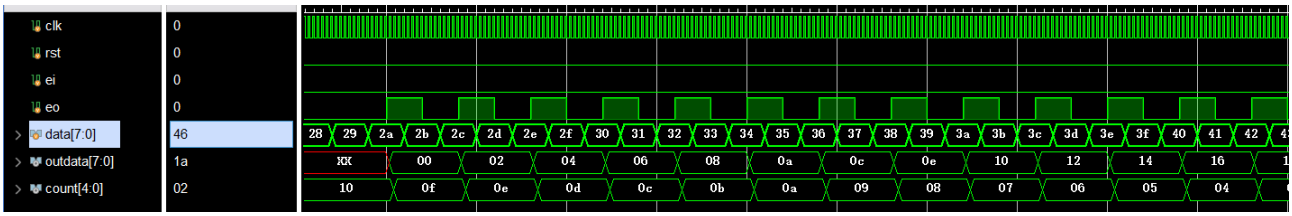


图 9: 出队, 查看出队数据, 检验与入队数据是否相同

接下来 **RST** 恢复初始状态，并入队出队交替进行数次。最后一直出队至队空，检验波形图是否正常

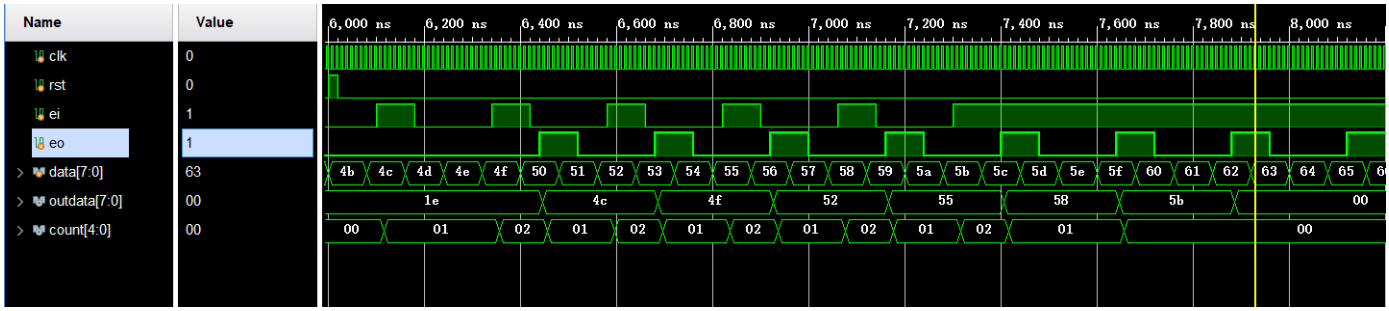


图 10: 交替入出队后出队至队空

3 实验结果

经仿真检验，寄存器堆与先进先出队列均正确实现

4 思考题

4.1 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？

按冒泡排序方法控制 **ra0** 和 **ra1** 作为读数据地址，将寄存器堆内数据依次作为 **ALU** 的两个操作数做减法
判断大小改变写入使能控制交换 (需要双端口写入使能和两个寄存器存储寄存器内原来的数据)
如果判断需要交换则反向写回，重复操作直至所有的数据均被排序 (**for loop**)

5 意见与建议

本次实验实现了 **2** 个 **CPU** 的基本单元，为 **CPU** 的设计做好了准备

6 附件

```
1  module register_file                                    //32 x WIDTH寄存器堆
2  #(  parameter WIDTH = 32,                  //数据宽度
3  parameter NUM = 32)
4  (
5
6
7  input clk,                                                    //时钟（上升沿
       有效）
8  input [4:0] ra0,                                        //读端口0地址
9  input [4:0] ra1,                                        //读端口1地址
10
11 output [WIDTH-1:0] rd0,          //读端口0数据
12 output [WIDTH-1:0] rd1,          //读端口1数据
13
14 input [4:0] wa,                                         //写端口地址
15 input we,                                               //写使能，高电平有效
16 input [WIDTH-1:0] wd             //写端口数据
17 );
18
19 reg [WIDTH-1:0]REG[0:NUM-1];     //定义寄存器堆的寄存器
20 integer i;
21
22 //初始化寄存器堆数据为0
23 initial
24 for(i = 0;i < NUM;i = i + 1)
25 REG[i] <= 0;
26 //同步写操作，需时钟控制
27 always@(posedge clk)
28 begin
29 if(we)
30 REG[wa] <= wd;
31 end
32
33 //异步读操作，不需时钟控制，组合逻辑
34 assign rd0 = REG[ra0];
35 assign rd1 = REG[ra1];
36
37 endmodule
```

```
38 module reg_file_tb;
39 reg clk;
40 reg [4:0] ra0;
41 wire [31:0] rd0;
42 reg [4:0] ra1;
43 wire [31:0] rd1;
44 reg [4:0] wa;
```

```
45                  reg we;
46                  reg [31:0] wd;
47
48                  register_file reg0(.clk(clk),.ra0(ra0),.ra1(ra1),.wa(wa),.rd0(rd0),.rd1
                        (rd1),.wd(wd),.we(we));
49
50                  parameter    PERIOD = 10, CYCLE = 150 ;
51                  initial
52                  begin
53                  clk = 0;
54                  repeat(CYCLE)
55                  #(PERIOD/2) clk = ~clk;
56                  $finish;
57                  end
58
59                  initial
60                  begin
61                  we = 1;
62                  wa = 5'b00000;
63                  wd = 32'h1234;
64                  #PERIOD we = 0;
65
66                  #(PERIOD*9)
67                  we = 1;
68                  wa = 5'b00010;
69                  wd = 32'h2345;
70                  #PERIOD we = 0;
71
72                  #(PERIOD*9)
73                  we = 1;
74                  wa = 5'b11111;
75                  wd = 32'h4321;
76                  #PERIOD we = 0;
77
78                  #(PERIOD*9)
79                  we = 1;
80                  wa = 5'b0010;
81                  wd = 32'h5432;
82                  #PERIOD we = 0;
83
84                  #(PERIOD*9)
85                  we = 1;
86                  wa = 5'b11110;
87                  wd = 32'h0015;
88                  #PERIOD we = 0;
89
90                  #(PERIOD*9)
91                  we = 1;
```

```
92          wa = 5'b00010;
93          wd = 0;
94          #PERIOD we = 0;
95          end
96
97          initial
98          begin
99          ra0 = 5'b00000;
100         ra1 = 5'b11110;
101
102         #(PERIOD*30)
103         ra0 = 5'b00010;
104         ra1 = 5'b11111;
105         end
106
107         endmodule
```

```
108         module blk_test();
109         reg [3:0]Mem_Addr;//读出和写入的地址
110         reg [7:0]M_W_Data;//写入的数据
111         reg clk_m, Mem_Write;//时钟和，写控制信号(高电平有效)，在时钟上升沿时写
                入，读也是上升沿
112         reg ena;
113         wire [7:0]M_R_Data;//读出的数据，输出
114         initial clk_m=0;
115         always #25 clk_m=~clk_m;  //每25ns，时钟翻转一次
116         initial//数据初始化
117         begin
118         Mem_Addr=4'b0000;Mem_Write=1'b0;ena=1'b1;
119         Mem_Addr=4'b0001;#50;
120         Mem_Addr=4'b0010;#50;
121         Mem_Addr=4'b0011;#50;
122         Mem_Addr=4'b0100;#50;
123         Mem_Addr=4'b0101;#50;
124         Mem_Addr=4'b0110;#50;
125         Mem_Addr=4'b0111;#50;
126         Mem_Addr=4'b1000;#50;
127         Mem_Addr=4'b1001;#50;
128         Mem_Write=1'b1;M_W_Data=8'b11111111;#50;
129         end
130
131         blk_ram_0 test (
132         .clka(clk_m),    // input wire clka
133         .wea(Mem_Write),      // input wire [0 : 0] wea
134         .addra(Mem_Addr),  // input wire [3 : 0] addra
135         .ena(ena),
136         .dina(M_W_Data),    // input wire [7 : 0] dina
137         .douta(M_R_Data)  // output wire [7 : 0] douta
```

```
138                 );
139                 endmodule
```

```
140                 module dist_test();
141                 reg [3:0]Mem_Addr;//读出和写入的地址
142                 reg [7:0]d;
143                 reg clk_m;  //时钟和，写控制信号(高电平有效)，在时钟上升沿时写入，读也
                        是上升沿
144                 reg we;     //写使能
145                 wire [7:0]M_R_Data;//读出的数据，输出
146                 initial clk_m=0;
147                 always #25 clk_m=~clk_m;  //每25ns，时钟翻转一次
148                 initial//数据初始化
149                 begin
150                 Mem_Addr=4'b0000;we=1'b0;
151                 Mem_Addr=4'b0001;#50;
152                 Mem_Addr=4'b0010;#50;
153                 Mem_Addr=4'b0011;#50;
154                 Mem_Addr=4'b0100;#50;
155                 Mem_Addr=4'b0101;#50;
156                 Mem_Addr=4'b0110;#50;
157                 Mem_Addr=4'b0111;#50;
158                 Mem_Addr=4'b1000;#50;
159                 Mem_Addr=4'b1001;#50;
160                 we=1'b1;d=8'b11111111;#50;
161                 end
162
163                 dist_ram_0 dist_ram(
164                 .a(Mem_Addr),       // input wire [3 : 0] a
165                 .d(d),      // input wire [7 : 0] d
166                 .clk(clk_m),  // input wire clk
167                 .we(we),    // input wire we
168                 .spo(M_R_Data)  // output wire [7 : 0] spo
169                 );
170                 endmodule
```

```
171                 module edg(
172                 input clk, rst, y,
173                 output p
174                 );
175
176                 reg [1:0] CUR_STATE,NEXT_STATE;
177                 parameter S0 = 2'b00;    parameter S1 = 2'b01;    parameter S2 = 2'b10;
178
179                 assign  p = (CUR_STATE == S1);
180
181                 always @(posedge clk, posedge rst)
182                 begin
```

```
183                 if (rst) CUR_STATE <= S0;
184                 else CUR_STATE <= NEXT_STATE;
185                 end
186
187                 always @(*)
188                 begin
189                 NEXT_STATE = CUR_STATE;
190                 case (CUR_STATE)
191                 S0: begin if (y)  NEXT_STATE = S1;  end
192                 S1: begin
193                 if (y) NEXT_STATE = S2;
194                 else   NEXT_STATE = S0;
195                 end
196                 S2: begin if (~y) NEXT_STATE = S0;  end
197                 default: NEXT_STATE = S0;
198                 endcase
199                 end
200
201                 endmodule
```

```
202                 module FIFO(
203                 input clk, rst,          //时钟（上升沿有效）、异步复位（高电平有效）
204                 input en_in,             //入队列使能，高电平有效
205                 input en_out,            //出队列使能，高电平有效
206                 input [7:0] din,                 //入队列数据
207                 output [7:0] dout,       //出队列数据
208                 output [4:0] count       //队列数据计数
209                 );
210
211
212                 parameter EMPTY = 2'b00,WORK = 2'b01,FULL = 2'b10;
213
214                 reg [1:0] CUR_STATE = EMPTY,NEXT_STATE;
215                 wire p_in,p_out,wea/*,ena*/;
216                 wire [3:0] addr,tail,head;
217                 wire [7:0] store_out;
218
219                 //取边沿电路
220                 edg in_edg(
221                 .clk(clk), .rst(rst), .y(en_in),.p(p_in) );
222                 edg out_edg(
223                 .clk(clk), .rst(rst), .y(en_out),.p(p_out) );
224
225                 Ctrl_Unit cu(.clk(clk), .p_in(p_in), .p_out(p_out),
226                 .CUR_STATE(CUR_STATE),.head(head), .tail(tail), .count(count));
227
228                 dist_ram_0  mem(
229                 .a(addr), .d(din),.clk(clk),.we(wea), .spo(store_out));
```

```
230
231             assign dout = p_out? ((CUR_STATE == EMPTY)? 8'b0 : store_out) : dout;
232             assign wea =  (CUR_STATE == FULL)? 1'b0 : p_in;
233             assign addr = p_in ? tail : head;
234
235             always@(posedge clk, posedge rst)
236             begin
237             if(rst)
238             CUR_STATE <= EMPTY;
239             else
240             CUR_STATE <= NEXT_STATE;
241             end
242
243             always@(*)
244             begin
245             case(CUR_STATE)
246             EMPTY:
247             if(p_in)
248             NEXT_STATE <= WORK;
249             else
250             NEXT_STATE <= EMPTY;
251             WORK:
252             begin
253             if(p_in)
254             begin
255             if(count == 15)
256             NEXT_STATE <= FULL;
257             else
258             NEXT_STATE <= WORK;
259             end
260             else if(p_out)
261             begin
262             if(count == 1)
263             NEXT_STATE <= EMPTY;
264             else
265             NEXT_STATE <= WORK;
266             end
267             else
268             NEXT_STATE <= WORK;
269             end
270             FULL:
271             begin
272             if(p_out)
273             NEXT_STATE <= WORK;
274             else
275             NEXT_STATE <= FULL;
276             end
277             default:
```

```
                NEXT_STATE <= EMPTY;
                endcase
                end
                endmodule
```

```
                module Ctrl_Unit(
                input clk,
                input p_in,p_out,
                input [1:0] CUR_STATE,
                output reg [3:0] tail,head,
                output reg [4:0] count );


                parameter EMPTY = 2'b00,WORK = 2'b01,FULL = 2'b10;

                always@(posedge clk)
                begin
                case(CUR_STATE)
                EMPTY:
                begin
                if(p_in)
                begin
                count <= 5'b1;
                head <= 0;
                tail <= 1;
                end
                else
                begin
                count <= 5'b0;
                head <= 0;
                tail <= 0;
                end
                end
                WORK:
                begin
                if(p_in)              //不允许同时出入队列，入队优先
                begin
                count <= count + 1;
                head <= head;
                tail <= tail + 1;
                end
                else if(p_out)
                begin
                count <= count - 1;
                head <= head + 1;
                tail <= tail;
                end
                end
```

```
325                 FULL:
326                 begin
327                 if(p_out)
328                 begin
329                 count <= count -1;
330                 head <= head + 1;
331                 tail <= tail;
332                 end
333                 end
334                 default:
335                 begin
336                 count <= count;
337                 head <= head;
338                 tail <= tail;
339                 end
340                 endcase
341                 end
342
343                 endmodule
```

```
344                 module FIFO_test();
345
346                 reg clk,rst;
347                 reg ei,eo;
348                 reg [7:0] data;
349                 wire [7:0] outdata;
350                 wire [4:0] count;
351
352                 FIFO FIFO_tb(.clk(clk),.rst(rst),.en_in(ei),.en_out(eo),.din(data),.
                        dout(outdata),.count(count));
353
354                 initial clk = 0;
355                 always#5 clk = ~clk;//时钟频率：100MHz
356
357                 initial data = 0;
358                 always#80 data = data + 1;  //数据至少3个周期
359
360                 initial begin
361                 ei = 0;
362                 #40 ei = 1;
363                 repeat(40)
364                 #80 ei = ~ei;
365                 ei = 0;
366                 end
367
368                 initial begin
369                 eo = 0;
370                 #3400 eo = 1;
```

```
371                 repeat(32)
372                 #80 eo = ~eo;
373                 eo = 0;
374                 end
375
376                 initial
377                 begin
378                 rst = 0;
379                 #6000 rst = 1;
380                 #20 rst = 0;
381                 end
382
383                 initial
384                 begin
385                 #6100 ei = 1;
386                 repeat(5)
387                 begin
388                 #80 ei = 0;
389                 #160 ei = 1;
390                 end
391                 end
392
393                 initial
394                 begin
395                 #6200 eo = 0;
396                 repeat(8)
397                 begin
398                 #80 eo = 0;
399                 #160 eo = 1;
400                 end
401                 end
402
403
404                 endmodule
```