

- 一. 实验目标
- 二. 实验内容
- 三. 实验步骤
- 四. 实验检查
- 五. 实验过程
 - 5.1 设计概述
 - 5.2 第一步：丰富指令集
 - 5.3 第二步：加入 I/O 和 BUS
- 六. 实验总结与建议

一. 实验目标

1. 理解计算机系统的组成结构和工作原理；
2. 理解计算机总线和接口的结构和功能；
3. 掌握软硬件综合系统的设计和调试方法。

二. 实验内容

设计实现一个简单的计算机应用系统。CPU可以选用Lab3-5设计的或者改进设计的（例如，增加指令、中断处理等）CPU，自选存储器的类型和容量；自选外设，例如拨动/按钮开关、指示灯、数码管、定时/计数器、键盘、鼠标、VGA显示、串口通信等；自选应用，例如计算斐波拉契序列、求最大/小值、排序、流水灯、画图、串口通信等。

待设计的计算机硬件组成结构如图-1所示，除CPU和存储器外，还有输入/输出设备（简称外设），所有部分通过总线（BUS）互联，外设通过I/O接口连接到总线。总线包括数据总线（D-BUS）、地址总线（A-BUS）和控制总线（C-BUS）。

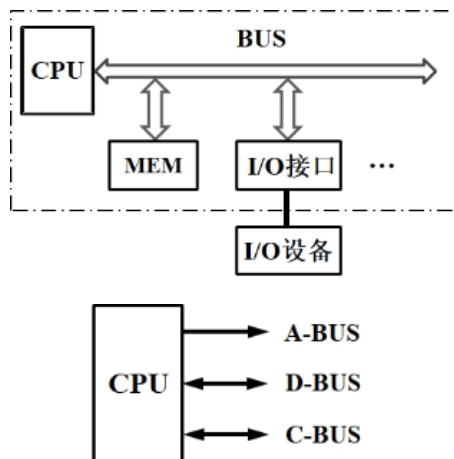


图-1 计算机硬件系统组成结构

I/O接口的主要功能是选择外设、数据缓冲和联络控制，一般组成结构如图-2所示。I/O接口中包含若干CPU可直接访问的寄存器，称之为I/O端口，其中数据输入寄存器（Data Input Register, DIR）和输出寄存器（Data Output Register, DOR）用于暂存待输入或输出的数据，状态寄存器（Status Register, SR）保存外设和接口的状态，控制寄存器（Control Register, CR）存放控制外设的命令和配置接口的工作方式。

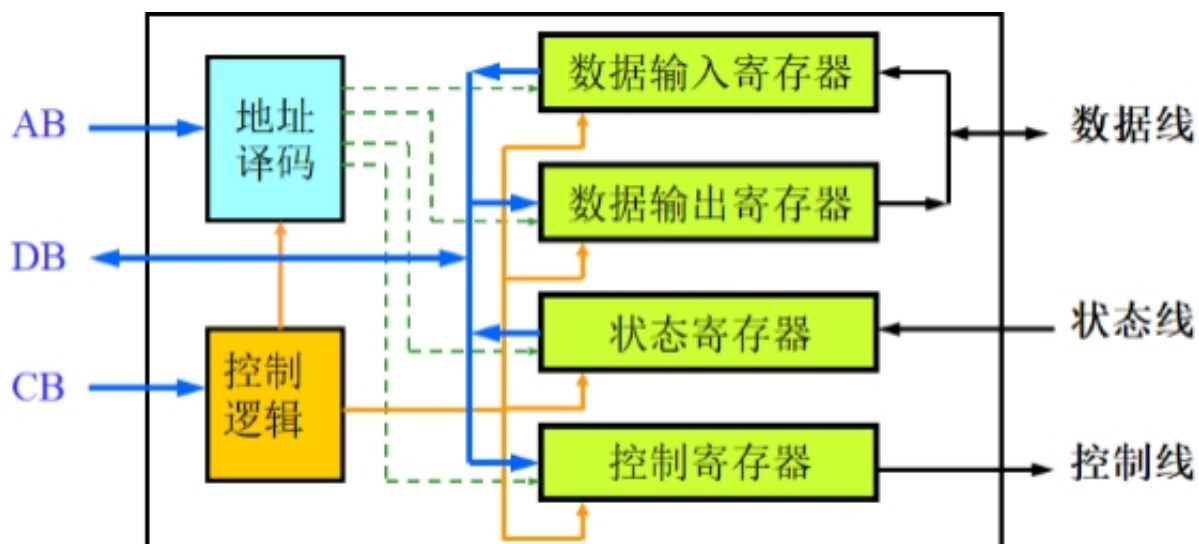


图-2 I/O接口的组成结构

I/O端口采用存储器映像 (Memory- mapped) 编址，即CPU将I/O端口与存储器单元同等对待，端口和主存单元占据同一地址空间的不同部分。这种编址的优点是系统设计简单；不需要专设I/O指令，用访存指令即可访问端口；控制简单，使用同一组地址和控制信号来访问端口和主存；访问端口的指令多、寻址方式灵活、功能强等。图-3给出了一种简单的总线实现结构。

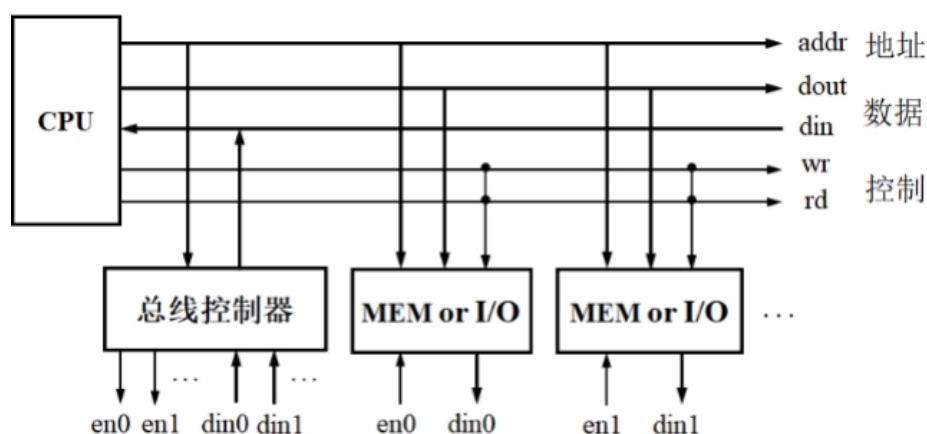
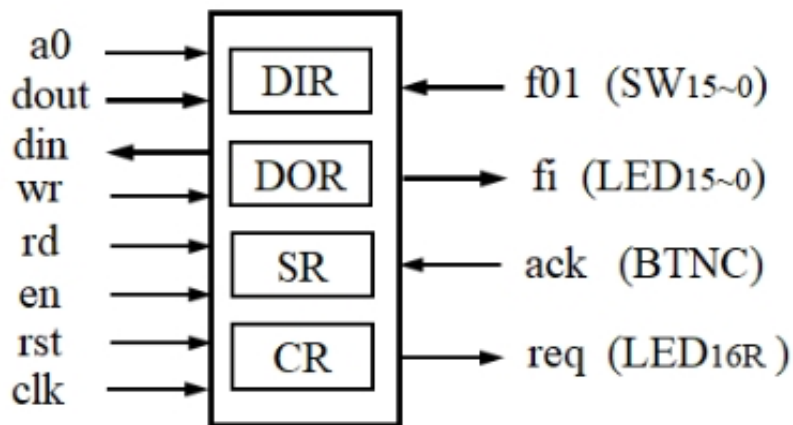


图-3 一种简单的总线结构

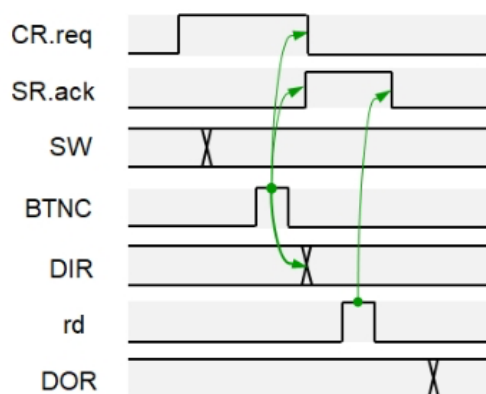
应用示例：计算斐波拉契序列： $F_{i+2} = F_i + F_{i+1}$, $i = 0, 1, 2 \dots$

- 依次从拨动开关（SW15~0）输入F0和F1
- 计算结果F2、F3...
- 依次在LED指示灯（LED15~0）上显示F0、F1...

实现该应用采用的接口结构和信号工作时序如图-4所示。



(a) 信号



(b) 时序

图-4 I/O接口信号和时序

三. 实验步骤

1. 自选外设，设计实现总线控制器和I/O接口，并进行功能仿真；
2. 自选应用，设计实现应用汇编程序，将汇编程序的机器代码存入存储器，并进行功能仿真；
3. 将完整应用系统下载至FPGA中测试。

四. 实验检查

1. 检查I/O接口的功能仿真；
2. 检查完整应用系统下载至FPGA后的运行功能。

五. 实验过程

5.1 设计概述

应用为一维细胞自动机（rule 90）：

Left	Center	Right	Center's next state
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	1
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	0

灵感来自 Coursera 的一门课 `model thinking`。

细胞自动机介绍摘自维基百科：

- 1 细胞自动机（英语：Cellular automaton），又称格状自动机、元胞自动机，是一种离散模型，在可计算性理论、数学及理论生物学都有相关研究。它是由无限个有规律、坚硬的方格组成，每格均处于一种有限状态。整个格网可以是任何有限维的。同时也是离散的。每格于 t 时的态由 $t-1$ 时的一集有限格（这集叫那格的邻域）的态决定。每一格的“邻居”都是已被固定的。（一格可以是自己的邻居。）每次演进时，每格均遵从同一规矩一齐演进。
- 2
- 3 就形式而言，细胞自动机有三个特征：
- 4
- 5 平行计算（parallel computation）：每一个细胞个体都同时同步的改变
- 6 局部的（local）：细胞的状态变化只受周遭细胞的影响。
- 7 一致性的（homogeneous）：所有细胞均受同样的规则所支配
- 8
- 9 史蒂芬·沃尔夫勒姆在《一种新科学》和几篇从80年代中期开始的论文中定义了四类细胞自动机和其他几个简单的计算模型。元胞自动机的早期研究往往试图确定具体规则的模式类型，他提出的分类是对规则本身份类的第一次尝试。按照复杂性分类的秩序：
- 10
- 11 1类：几乎所有的初始模式迅速演变成一个稳定的，均匀的状态。在初始模式的任何随机性会消失。
[5]
- 12 2类：几乎所有的初始模式迅速演化为稳定或振荡结构。一些在初始模式的随机性可能会被过滤掉，但是还有一些保留。在初始模式的局部变化倾向于继续保持局部性。[5]
- 13 3类：几乎所有的初始形态将会演变成一个伪随机或混沌的形式。任何稳定的结构很快会被周围的噪音破坏。在初始模式的局部变化有无限蔓延的倾向。[5]

14 4类：几乎所有的初始模式将会演变成相互作用的复杂和有趣的方式结构，并且局部结构的形成能够长时间存在。[6]2类的稳定或振荡的结构可能是最终的结果，但需要达到这个状态的步骤数目可能是非常大的，即使在初始模式比较简单的情況下。初始模式的局部变化可能会无限蔓延。史蒂芬·沃尔夫勒姆已推测不是所有的4类细胞自动机能够进行通用计算。这已被证明对于规则110和约翰·何顿·康威的生命游戏。

操作流程为：按下status（某btn按钮）通过SW输入16位初始细胞，每次按下step（某btn按钮）就会更新一次，由16位的led显示。

不同的初始输入可能会发现细胞可能是任意4类中的一类，可以尝试的可能有 2^{16} 。

对应到CPU的基本操作为：

```
1 cycle lw $t0, $zero, input_addr # input_addr = 0
2 srl $t1, $t0, 1
3 sll $t2, $t0, 1
4 xor $t0, $t1, $t2
5 sw $t0, $zero, input_addr
6 j cycle
```

其中xor的作用可由分析真值表得到，rule 90实际上就是对输入的错位异或。

转化成instruction的二进制文件为：

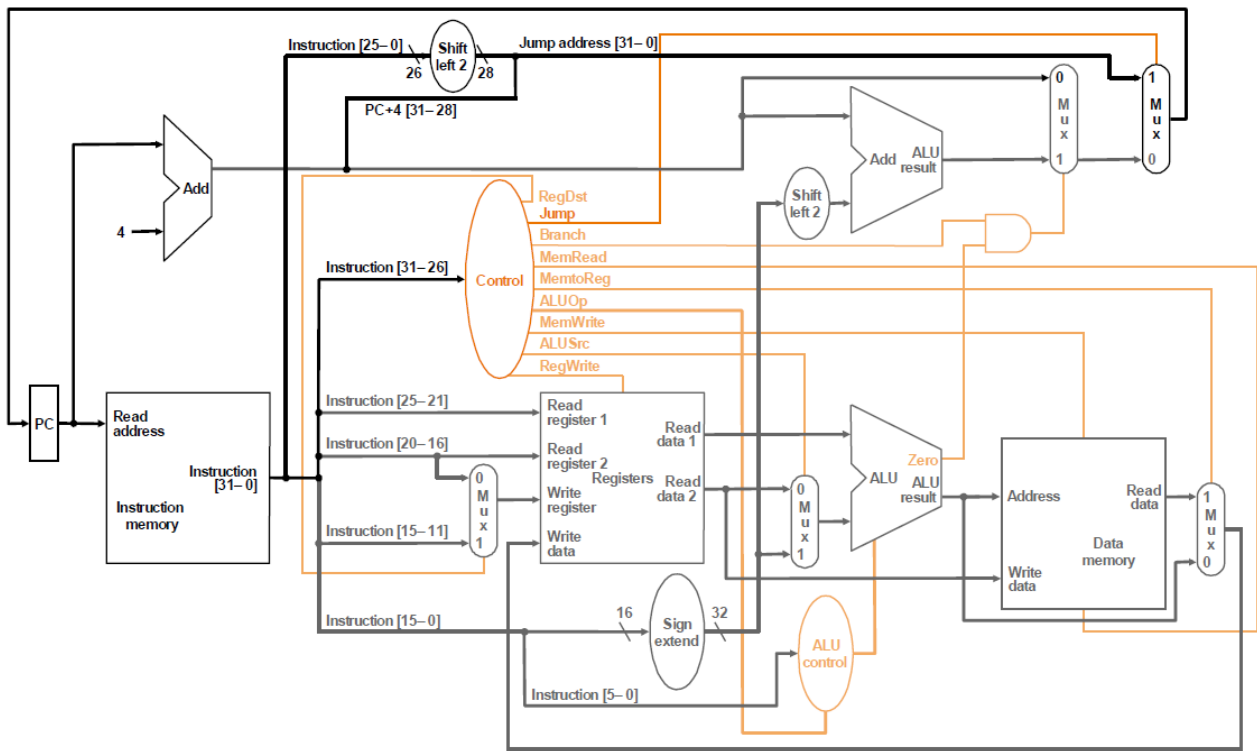
```
1 100011 00000 01000 0000000000000000
2 000000 00000 01000 01001 00001 000010
3 000000 00000 01000 01010 00001 000000
4 000000 01001 01010 01000 00000 100110
5 101011 00000 01000 0000000000000000
6 000010 0000000000000000000000000000
```

5.2 第一步：丰富指令集

Bit#	31..26	25..21	20..11	10..6	5..0	示例	示例含义	操作以及解释
xor	000000	rs	rt rd	00000	100110	xor \$1, \$2, \$3	\$1=\$2 ^ \$3	rd <- rs xor rt ; 其中 rs=\$2, rt=\$3, rd=\$1(异或)
sll	000000	00000	rt rd	shamt	000000	sll \$1, \$2, 10	\$1=\$2<<10	rd <- rt << shamt ; shamt 存放移位的位数，也就是指令中的立即数，其中rt=\$2, rd=\$1
srl	000000	00000	rt rd	shamt	000010	srl \$1, \$2, 10	\$1=\$2>>10	rd <- rt >> shamt ; shamt (logical), 其中rt=\$2, rd=\$1

对于CPU的修改大致有：

- 增加ALU的处理左右移，结合CPU设计图在ALU的A输入通过新增的MUX，接入shamt



module ALU (结合图可以理解是 $b \gg a$)

```

1      3'b101: // <<
2      begin
3          y = b << a;
4          zf = ~|y;
5          cf = 0; //todo
6          of = 0; //todo
7      end
8      3'b110: // >> 逻辑右移
9      begin
10         y = b >> a;
11         zf = ~|y;
12         cf = 0; //todo
13         of = 0; //todo
14     end

```

新增的控制信号 `ShamtSignal` 在ALUControl中处理:

module ALUControl

```

1  always@(*)
2      if(funcnt == 6'b000000 | funcnt == 6'b000010)
3          ShamtSignal <= 1'b1;
4      else
5          ShamtSignal <= 1'b0;
6  endmodule

```

也需要修改几个接口 (略)。

The timing diagram displays the following signals and their values over time:

Signal	Value
clk	0
rst	0
PERI...1	0000000a
wa[4:0]	08
wd[31:0]	000000aa
we	1
MemR...	00000000
ALUr...1	000000aa
a[31:0]	00000022
b[31:0]	00000088
ALUSrc	0
Sham...	0
exten...1	00000000

CPU功能正常。

SingleCPU.v

```

1 module cpu_one_cycle( // 单周期 CPU
2     input clk,         // 时钟（上升沿有效）
3     input rst          // 异步复位，高电平有效
4 );
5 localparam m = 3'b0;
6 // PC
7 wire [31 : 0] new_addr, cur_addr, PCadd;
8 wire [31 : 0] insturction;          // insturction
9 wire [31 : 0] ALUresult0;
10 wire [27 : 0] Shift0;
11 wire [31 : 0] JumpAddr, Mux4out, Mux6out;
12 wire [33 : 0] Shift1;
13 wire [5 : 0] funct;
14 wire [4 : 0] shamt;
15 // 信号
16 wire [5:0] op;
17 wire ShamtSignal;
18 wire RegDst, Jump, Branch, MemtoReg, MemWrite, ALUSrc, RegWrite;
19 // Registers
20 wire [4 : 0] ReadReg1, ReadReg2, ReadReg3, WriteReg;
21 wire [31 : 0] ReadData1, ReadData2, WriteData;
22 // signExtend

```

```

23 wire [15 : 0] imm;
24 wire [31 : 0] extendImm;
25 // 纯拓展
26 wire [31 : 0] extendShamt;
27 // ALU
28 wire [31 : 0] ALUin2, ALUresult1;
29 wire zero;
30 //wire [2 : 0] m;
31 // Mem
32 wire [31 : 0] MemReadData;
33 // ALUControl
34 wire Op1, Op0;
35 wire [2:0] ALUOp;
36
37 // assign
38 assign op = insturction[31 : 26];
39 assign ReadReg1 = insturction[25 : 21];
40 assign ReadReg2 = insturction[20 : 16];
41 assign ReadReg3 = insturction[15 : 11];
42 assign imm = insturction[15 : 0];
43 assign funct = insturction[5 : 0];
44 assign shamt = insturction[10 : 6];
45 assign extendShamt = {27'b0, shamt};
46 //assign m = 3'b000;
47 assign PCadd = cur_addr + 4;
48 assign Shift0 = {insturction[25 : 0], 2'b00};
49 assign JumpAddr = {PCadd[31 : 28], Shift0};
50 assign Shift1 = {extendImm, 2'b00};
51
52 Mux32 mux6(
53     .control(ShamtSignal),
54     .in0(ReadData1),
55     .in1(extendShamt),
56     .out(Mux6out)
57 );
58
59 Mux32 mux4(
60     .control(Branch&zero),
61     .in0(PCadd),
62     .in1(ALUresult0),
63     .out(Mux4out)
64 );
65
66 Mux32 mux5(
67     .control(Jump),
68     .in0(Mux4out),
69     .in1(JumpAddr),
70     .out(new_addr)
71 );

```



```

72
73     ALU alu0(
74         .y(ALUresult0),
75         .zf(),
76         .cf(),
77         .of(),
78         .a(PCadd),
79         .b(Shift1[31 : 0]),
80         .m(m)
81     );
82
83     /*
84     module ALUControl(
85         input Op1, Op0,
86         input [5 : 0] funct,
87         output reg [2 : 0] ALUOp
88     );
89     */
90     ALUControl alucontrol(
91         .Op1(Op1),
92         .Op0(Op0),
93         .funct(funct),
94         .ALUOp(ALUOp),
95         .ShamtSignal(ShamtSignal)
96     );
97
98
99     /*
100     module PC(
101         input clk,
102         input rst,
103         input [31:0] new_addr,
104         output reg [31:0] cur_addr
105     );*/
106     PC pc(
107         .clk(clk),
108         .rst(rst),
109         .new_addr(new_addr),
110         .cur_addr(cur_addr)
111     );
112
113     // InstructionMemory 256*32
114     ROM rom(
115         .a(cur_addr[9:2]),          // 读地址
116         .spo(instruction)          // 读数据
117     );
118
119     /*
120     module Control(

```

```

121     input [5:0] insturction,
122     output reg RegDst,
123     output reg ALUSrc,
124     output reg MemtoReg,
125     output reg RegWrite,
126     output reg MemRead,
127     output reg MemWrite,
128     output reg Branch,
129     output reg ALUOp1, ALUOp0
130     output reg Jump
131 );
132 */
133 Control control(
134     .instruction(op),
135     .RegDst(RegDst),
136     .ALUSrc(ALUSrc),
137     .MemtoReg(MemtoReg),
138     .RegWrite(RegWrite),
139     .MemRead(),
140     .MemWrite(MemWrite),
141     .Branch(Branch),
142     .ALUOp1(Op1),
143     .ALUOp0(Op0),
144     .Jump(Jump)
145 );
146
147 /*
148 module Registers          //32 x WIDTH寄存器堆
149 #(parameter WIDTH = 32)  //数据宽度
150 (
151     input clk,            //时钟 (上升沿有效)
152     input [4:0] ra0,       //读端口0地址
153     output reg [WIDTH-1:0] rd0,    //读端口0数据
154     input [4:0] ra1,       //读端口1地址
155     output reg [WIDTH-1:0] rd1,    //读端口1数据
156     input [4:0] wa,        //写端口地址
157     input we,             //写使能, 高电平有效
158     input [WIDTH-1:0] wd   //写端口数据
159 );
160 */
161 Registers registers(
162     .clk(clk),
163     .ra0(ReadReg1),
164     .rd0(ReadData1),
165     .ra1(ReadReg2),
166     .rd1(ReadData2),
167     .wa(WriteReg),
168     .we(RegWrite),
169     .wd(WriteData)

```

```

170 );
171
172 /*
173 module Mux5(
174     input control,
175     input [4:0] in1, in0,
176     output [4:0] out
177 );
178 */
179 Mux5 mux0(
180     .control(RegDst),
181     .in0(ReadReg2),
182     .in1(ReadReg3),
183     .out(WriteReg)
184 );
185
186 Mux32 mux1(
187     .control(ALUSrc),
188     .in0(ReadData2),
189     .in1(extendImm),
190     .out(ALUin2)
191 );
192
193 /*
194 module Sign_extend(
195     input [15:0] imm,
196     output [31:0] extendImm
197 );
198 */
199 Sign_extend signExtend(
200     .imm(imm),
201     .extendImm(extendImm)
202 );
203
204 /*
205 module ALU
206 #(parameter WIDTH = 32)    //数据宽度
207 (output reg [WIDTH-1:0] y,    //运算结果
208 output reg zf,              //零标志
209 output reg cf,              //进位/借位标志
210 output reg of,              //溢出标志
211 input [WIDTH-1:0] a, b,      //两操作数
212 input [2:0] m                //操作类型
213 );
214 */
215 ALU alu1(
216     .y(ALUresult1),
217     .zf(zero),
218     .cf(),

```

```

219     .of(),
220     .a(Mux6out),
221     .b(ALUin2),
222     .m(ALUOp)
223 );
224
225 // DataMemory 256*32
226 RAM ram (
227     .a(ALUresult1[9 : 2]), // input wire [7 : 0] 地址
228     .d(ReadData2),        // input wire [31 : 0] d 写数据
229     .clk(clk),             // input wire clk
230     .we(MemWrite),         // input wire we 写使能
231     .spo(MemReadData)      // output wire [31 : 0] spo 读数据
232 );
233
234 Mux32 mux2(
235     .control(MemtoReg),
236     .in0(ALUresult1),
237     .in1(MemReadData),
238     .out(WriteData)
239 );
240
241 endmodule

```

func.v

```

1  module ALU
2  #(parameter WIDTH = 32) //数据宽度
3  (output reg [WIDTH-1:0] y, //运算结果
4  output reg zf, //零标志
5  output reg cf, //进位/借位标志
6  output reg of, //溢出标志
7  input [WIDTH-1:0] a, b, //两操作数
8  input [2:0] m //操作类型
9  );
10 always@(*)
11     begin
12         case(m)
13             3'b000: // +
14                 begin
15                     {cf, y} = a + b;
16                     of = (~a[WIDTH-1] & ~b[WIDTH-1] & y[WIDTH-1])
17                         | (a[WIDTH-1] & b[WIDTH-1] & ~y[WIDTH-1]);
18                     zf = ~|y;
19                 end
20             3'b001: // -
21                 begin
22                     {cf, y} = a - b;
23                     of = (~a[WIDTH-1] & b[WIDTH-1] & y[WIDTH-1])

```

```

24         | (a[WIDTH-1] & ~b[WIDTH-1] & ~y[WIDTH-1]);
25         zf = ~|y;
26     end
27     3'b010: // &
28     begin
29         y = a & b;
30         zf = ~|y;
31         cf = 0;
32         of = 0;
33     end
34     3'b011: // |
35     begin
36         y = a | b;
37         zf = ~|y;
38         cf = 0;
39         of = 0;
40     end
41     3'b100: // ^
42     begin
43         y = a ^ b;
44         zf = ~|y;
45         cf = 0;
46         of = 0;
47     end
48     3'b101: // <<
49     begin
50         y = b << a;
51         zf = ~|y;
52         cf = 0; //todo
53         of = 0; //todo
54     end
55     3'b110: // >> 逻辑右移
56     begin
57         y = b >> a;
58         zf = ~|y;
59         cf = 0; //todo
60         of = 0; //todo
61     end
62     default:
63     begin
64         y = 0;
65         zf = 0;
66         cf = 0;
67         of = 0;
68     end
69 endcase
70 end
71 endmodule
72

```

```

73 module Registers          //32 x WIDTH寄存器堆
74 #(parameter WIDTH = 32)    //数据宽度
75 (
76     input clk,              //时钟（上升沿有效）
77     input [4:0] ra0,         //读端口0地址
78     output reg [WIDTH-1:0] rd0, //读端口0数据
79     input [4:0] ra1,         //读端口1地址
80     output reg [WIDTH-1:0] rd1, //读端口1数据
81     input [4:0] wa,          //写端口地址
82     input we,                //写使能，高电平有效
83     input [WIDTH-1:0] wd     //写端口数据
84 );
85 reg [WIDTH-1:0] mem [255:0];
86 // 初始化 RAM 的内容
87 initial
88 begin
89     //$readmemh("C:/Users/mi/Desktop/text.txt", mem, 0, 255);
90     $readmemh("C:/Users/mi/Desktop/lab6_cpu/initReg.vec", mem, 0, 255);
91 end
92 // 异步读
93 always@(*)
94 begin
95     rd0 = mem[ra0];
96     rd1 = mem[ra1];
97 end
98 // 同步写
99 always@(posedge clk)
100 begin
101     if(we & wa!=0)
102         mem[wa] <= wd;
103 end
104 endmodule
105
106 module Mux5(
107     input control,
108     input [4:0] in1, in0,
109     output [4:0] out
110 );
111 assign out = control? in1:in0;
112 endmodule
113
114 module Mux32(
115     input control,
116     input [31:0] in1, in0,
117     output [31:0] out
118 );
119 assign out = control? in1:in0;
120 endmodule
121

```

```

122 module Sign_extend(
123     input [15:0] imm,
124     output [31:0] extendImm
125 );
126 assign extendImm[15:0] = imm;
127 assign extendImm[31:16] = imm[15] ? 16'hffff : 16'h0000;
128 endmodule
129
130 module Control(
131     input [5:0] instruction,
132     output reg RegDst,
133     output reg ALUSrc,
134     output reg MemtoReg,
135     output reg RegWrite,
136     output reg MemRead,
137     output reg MemWrite,
138     output reg Branch,
139     output reg ALUOp1, ALUOp0,
140     output reg Jump
141 );
142 // add addi lw sw beq j srl sll xor
143 // x 都归为 0
144 always @(instruction)
145 begin
146     case(instruction)
147         6'b000000: // add func=100000
148             {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
149             MemWrite, Branch, ALUOp1, ALUOp0,
150             Jump} <= 10'b1001000100;
151         6'b100011: // lw
152             {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
153             MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'b0111100000;
154         6'b101011: // sw
155             {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
156             MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'bx1x0010000;
157         6'b000100: // beq
158             {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
159             MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'bx0x0001010;
160         6'b001000: // addi
161             {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
162             MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'b0101000000;
163         6'b000010: // j
164             {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
165             MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'bxxx000x011;
166         /*6'b000000: // xor func=100110
167             {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
168             MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'b1001000??0;
169         6'b000000: // func sll 000000 srl 000001
170         */

```

```

171     endcase
172 end
173 endmodule
174
175 module ALUControl(
176     input Op1, Op0,
177     input [5 : 0] funct,
178     output reg [2 : 0] ALUOp,
179     output reg ShamtSignal
180 );
181 always@(*)
182 begin
183     case({Op1, Op0})
184         2'b00: ALUOp <= 3'b000;
185         2'b01: ALUOp <= 3'b001;
186         2'b10: begin
187             case(funct)
188                 6'b100000: ALUOp <= 3'b000; // +
189                 6'b100010: ALUOp <= 3'b001; // -
190                 6'b100100: ALUOp <= 3'b010; // and
191                 6'b100101: ALUOp <= 3'b011; // or
192                 6'b100110: ALUOp <= 3'b100; // xor
193                 6'b000000: ALUOp <= 3'b101; // <<
194                 6'b000010: ALUOp <= 3'b110; // >>
195                 default: ALUOp <= 3'b111;
196             endcase
197         end
198     endcase
199 end
200 always@(*)
201     if(funct == 6'b000000 | funct == 6'b000010)
202         ShamtSignal <= 1'b1;
203     else
204         ShamtSignal <= 1'b0;
205 endmodule
206
207 module PC(
208     input clk,
209     input rst,
210     input [31:0] new_addr,
211     output reg [31:0] cur_addr
212 );
213 initial
214     cur_addr <= 0;
215 always@(posedge clk or posedge rst)
216 begin
217     if(rst)
218         cur_addr <= 0;
219     else

```



```

220         cur_addr <= new_addr;
221     end
222 endmodule

```

5.3 第二步：加入 I/O 和 BUS

设计概况：

对应到CPU的基本操作如下。

1-4行是轮询，当按钮按下，I_step 为高电平时，flag_place 所指的寄存器被置为特殊值，即可跳出轮询一次。

而 I_flag 高电平会存 SW 的内容到 input_addr 所指的寄存器，从 CPU 执行的代码中看不出来这个。

5-8行是细胞生成的过程。

```

1  cycle lw $t3, $zero, flag_place # flag_place = 8'hfe
2  addi $t4, $zero, flag_data # flag_data = 32'b1
3  beq $t3, $t4, calculate
4  j cycle
5  calculate lw $t0, $zero, input_addr # input_addr = 8'hff
6  srl $t1, $t0, 1
7  sll $t2, $t0, 1
8  xor $t0, $t1, $t2
9  sw $t0, $zero, input_addr
10 j cycle

```

转化成instruction的二进制表达为：

```

1  100011 00000 01011 000000 11111110 00
2  001000 00000 01100 00000000 00000001
3  000100 01100 01011 00000000 00000001
4  000010 000000000000000000000000000000
5  100011 00000 01000 000000 11111111 00
6  000000 00000 01000 01001 00001 000010
7  000000 00000 01000 01010 00001 000000
8  000000 01001 01010 01000 00000 100110
9  101011 00000 01000 000000 11111111 00
10 000010 000000000000000000000000000000

```

代码实现相比于 5.2 的改进主要在引入了 BUS，根据各自情况改寄存器：

```

1  /*
2  btn1 I_step高电平 存 flag_data 进 flag_place
3  btn2 I_flag高电平 读入SW 存16位SW进 input_addr
4  */
5  module bus(
6      input clk,
7      input I_step, I_flag, // btn

```

```

8      input [15 : 0] I_data,
9      output [15 : 0] O_led,
10     input [7 : 0] Mem_a,
11     input [31 : 0] Mem_d,
12     input Mem_we,
13     output reg [31 : 0] Mem_spo
14 );
15 localparam flag_place = 8'hfe;
16 localparam input_addr = 8'hff;
17 reg [31 : 0] flag_data=0;
18 reg [31 : 0] input_data=0;
19 wire [31 : 0] inter_spo;
20
21 assign O_led = input_data[15 : 0];
22
23 always@(posedge clk)
24 begin
25     if(I_step)
26         flag_data <= 32'b1; // 特殊值
27     else
28         flag_data <= 32'b0;
29 end
30
31 always@(*) // MEM 读
32 begin
33     if(Mem_a == flag_place)
34         Mem_spo = flag_data;
35     else if(Mem_a == input_addr)
36         Mem_spo = input_data;
37     else
38         Mem_spo = inter_spo;
39 end
40
41 always@(posedge clk) // MEM 写
42 begin
43     if(Mem_we)
44     begin
45         if(Mem_a == input_addr)
46             input_data <= Mem_d;
47         else
48             input_data <= input_data;
49     end
50     else
51     begin
52         if(I_flag)
53             input_data <= {16'b0, I_data};
54         else
55             input_data <= input_data;
56     end

```

```

57 end
58
59 // DataMemory 256*32
60 RAM ram (
61     .a(Mem_a), // input wire [7 : 0] 地址
62     .d(Mem_d), // input wire [31 : 0] d 写数据
63     .clk(clk), // input wire clk
64     .we(Mem_we), // input wire we 写使能
65     .spo(inter_spo) // output wire [31 : 0] spo 读数据
66 );
67
68 endmodule

```

仿真测试文件为：

```

1  module tb();
2  reg clk;
3  reg rst;
4  reg I_step, I_flag;
5  reg [15 : 0] I_data;
6  wire [15 : 0] O_led;
7
8  /*
9  module cpu_one_cycle( // 单周期 CPU
10     input clk, // 时钟（上升沿有效）
11     input rst, // 异步复位，高电平有效
12     input I_step, I_flag, // btn
13     input [15 : 0] I_data,
14     output [15 : 0] O_led,
15 );
16 */
17 cpu_one_cycle cpu(
18     .clk(clk),
19     .rst(rst),
20     .I_step(I_step),
21     .I_flag(I_flag),
22     .I_data(I_data),
23     .O_led(O_led)
24 );
25 parameter PERIOD = 10;
26 initial
27 begin
28     rst = 1;
29     # (PERIOD*1);
30     rst = 0;
31 end
32
33 initial
34 begin

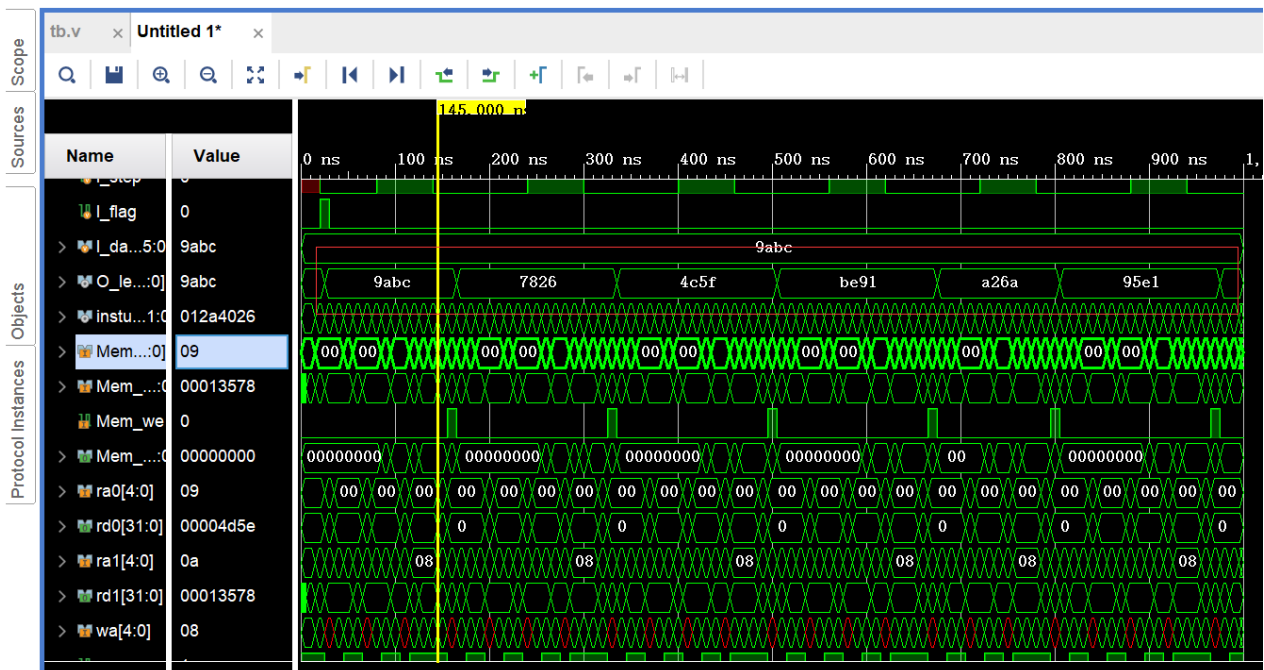
```

```

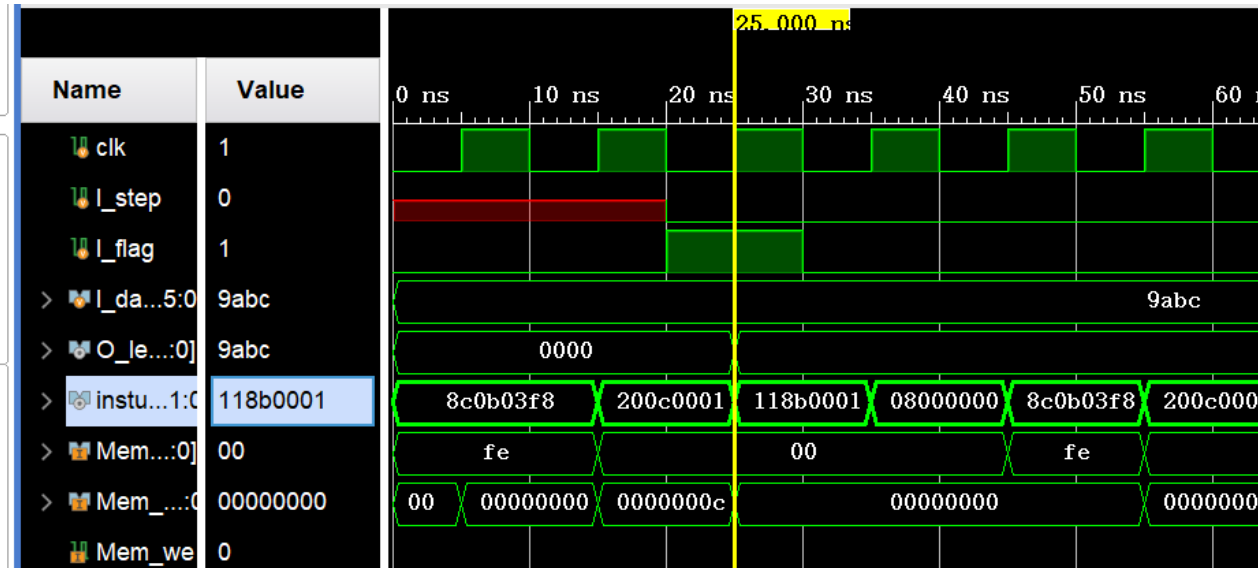
35     clk = 0;
36     repeat (400) // 待定
37         #(PERIOD/2) clk = ~clk;
38     $finish;
39 end
40
41 /*
42 btn1 I_step高电平 存 flag_data 进 flag_place
43 btn2 I_flag高电平 读入SW 存16位SW进 input_addr
44 */
45 initial
46 begin
47     I_data = 16'h9abc; // test
48     I_flag = 0;
49     # (PERIOD*2);
50     I_flag = 1; // 改变了input_data
51     I_step = 0;
52     # (PERIOD);
53     I_flag = 0;
54     # (PERIOD*5);
55     repeat(10)
56     begin
57         I_step = 1;
58         # (PERIOD*6);
59         I_step = 0;
60         # (PERIOD*10);
61     end
62 end
63 endmodule

```

先将 SW 读入一次，从 I_flag 行和 I_data 行可以看出。后面 O_led 的几次变化来自 I_step 的几次脉冲。



具体来看，O_led 即可变化为了第一个输入（25.000ns）。

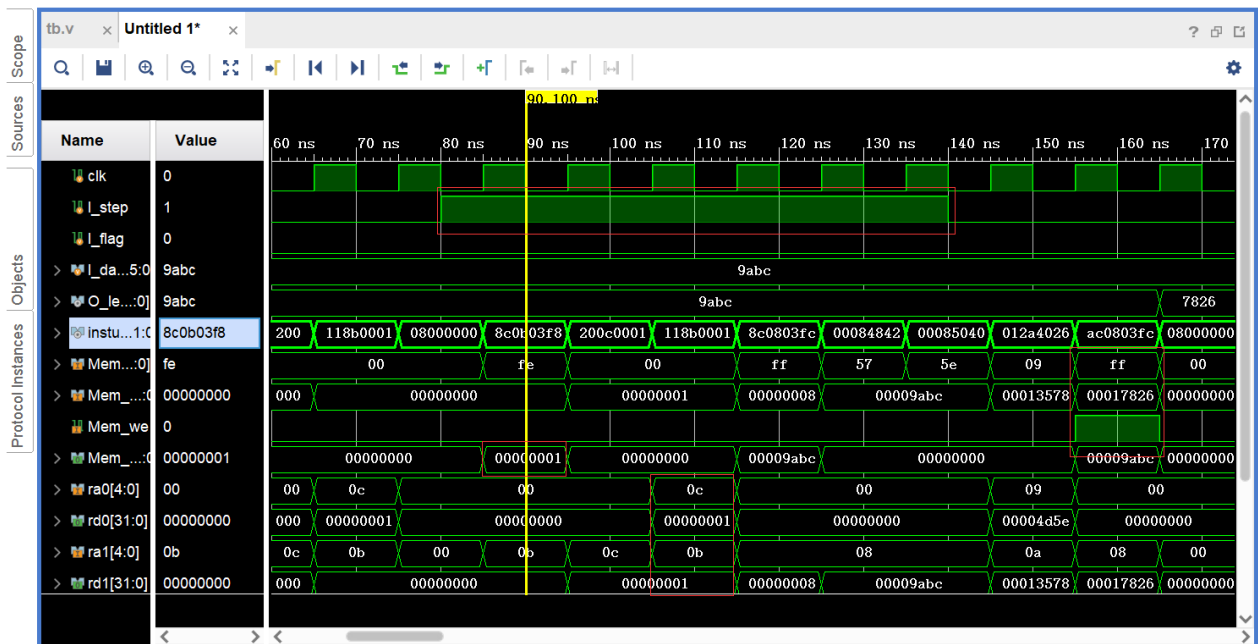


画了红框的都是重要的几处。

l_step 的长度需要覆盖轮询的所有阶段，因为默认按下按钮时并不知道轮询段跑到哪里了；而不需要覆盖所有代码，因为默认两次按按钮会相隔足够远，不需要担心计算没有结束。

发现beq的比较 t3 和 t4 相等，所以跳转成功。

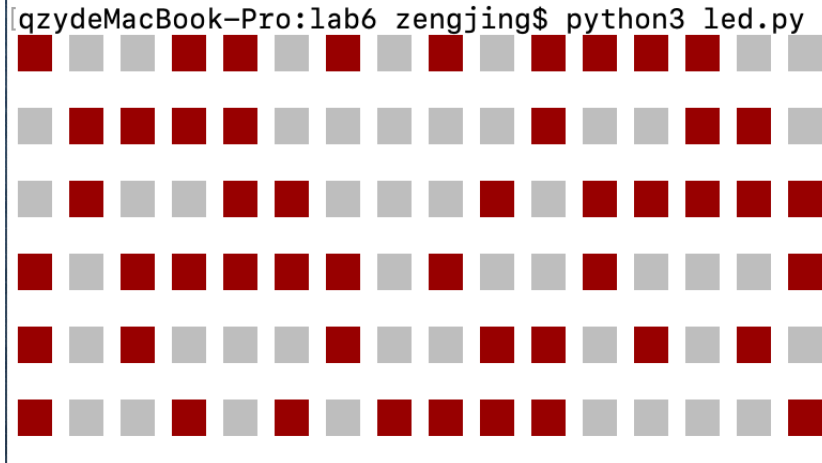
图的右边可以看到，计算结束后，Mem_we 变为高电平，写入了新的细胞值。



一切符合预期。

用 Python 写了一个显示成LED的模拟程序，红色是亮灯：

```
qzydeMacBook-Pro:lab6 zengjing$ python3 led.py
```



数据是 CPU 计算出来的，源代码如下：

```
1  import time
2
3  string = "1001101010111100 \
4  0111100000100110 010011000101111 1011111010010001 \
5  1010001001101010 1001010111100001"
6
7  strings = string.split(' ')
8  for num in strings:
9      for point in num:
10         if(point=='0'):
11             print("\033[1;30;47m  \033[0m",end="")
12         if(point=='1'):
13             print('\033[1;30;41m  \033[0m',end="")
14         print(" ",end="")
15     print("\n")
16     time.sleep(1)
```

附本版本 Verilog 代码：

SingleCPU.v

```
1  module cpu_one_cycle( // 单周期 CPU
2      input clk,          // 时钟（上升沿有效）
3      input rst,          // 异步复位，高电平有效
4      input I_step, I_flag, // btn
5      input [15 : 0] I_data,
6      output [15 : 0] O_led,
7  );
8  localparam m = 3'b0;
9  // PC
10 wire [31 : 0] new_addr, cur_addr, PCadd;
11 wire [31 : 0] insturction;          // insturction
12 wire [31 : 0] ALUresult0;
13 wire [27 : 0] Shift0;
14 wire [31 : 0] JumpAddr, Mux4out, Mux6out;
```

```

15 wire [33 : 0] Shift1;
16 wire [5 : 0] funct;
17 wire [4 : 0] shamt;
18 // 信号
19 wire [5:0] op;
20 wire ShamtSignal;
21 wire RegDst, Jump, Branch, MemtoReg, MemWrite, ALUSrc, RegWrite;
22 // Registers
23 wire [4 : 0] ReadReg1, ReadReg2, ReadReg3, WriteReg;
24 wire [31 : 0] ReadData1, ReadData2, WriteData;
25 // signExtend
26 wire [15 : 0] imm;
27 wire [31 : 0] extendImm;
28 // 纯拓展
29 wire [31 : 0] extendShamt;
30 // ALU
31 wire [31 : 0] ALUin2, ALUresult1;
32 wire zero;
33 //wire [2 : 0] m;
34 // Mem
35 wire [31 : 0] MemReadData;
36 // ALUControl
37 wire Op1, Op0;
38 wire [2:0] ALUOp;
39
40 // assign
41 assign op = insturction[31 : 26];
42 assign ReadReg1 = insturction[25 : 21];
43 assign ReadReg2 = insturction[20 : 16];
44 assign ReadReg3 = insturction[15 : 11];
45 assign imm = insturction[15 : 0];
46 assign funct = insturction[5 : 0];
47 assign shamt = insturction[10 : 6];
48 assign extendShamt = {27'b0, shamt};
49 //assign m = 3'b000;
50 assign PCadd = cur_addr + 4;
51 assign Shift0 = {insturction[25 : 0], 2'b00};
52 assign JumpAddr = {PCadd[31 : 28], Shift0};
53 assign Shift1 = {extendImm, 2'b00};
54
55 Mux32 mux6(
56     .control(ShamtSignal),
57     .in0(ReadData1),
58     .in1(extendShamt),
59     .out(Mux6out)
60 );
61
62 Mux32 mux4(
63     .control(Branch&zero),

```

```

64     .in0(PCadd),
65     .in1(ALUresult0),
66     .out(Mux4out)
67 );
68
69 Mux32 mux5(
70     .control(Jump),
71     .in0(Mux4out),
72     .in1(JumpAddr),
73     .out(new_addr)
74 );
75
76 ALU alu0(
77     .y(ALUresult0),
78     .zf(),
79     .cf(),
80     .of(),
81     .a(PCadd),
82     .b(Shiftl[31 : 0]),
83     .m(m)
84 );
85
86 /*
87 module ALUControl(
88     input Op1, Op0,
89     input [5 : 0] funct,
90     output reg [2 : 0] ALUOp
91 );
92 */
93 ALUControl alucontrol(
94     .Op1(Op1),
95     .Op0(Op0),
96     .funct(funct),
97     .ALUOp(ALUOp),
98     .ShamtSignal(ShamtSignal)
99 );
100
101
102 /*
103 module PC(
104     input clk,
105     input rst,
106     input [31:0] new_addr,
107     output reg [31:0] cur_addr
108 );*/
109 PC pc(
110     .clk(clk),
111     .rst(rst),
112     .new_addr(new_addr),

```



```

113     .cur_addr(cur_addr)
114 );
115
116 // InstructionMemory 256*32
117 ROM rom(
118     .a(cur_addr[9:2]),          // 读地址
119     .spo(insturction)          // 读数据
120 );
121
122 /*
123 module Control(
124     input [5:0] insturction,
125     output reg RegDst,
126     output reg ALUSrc,
127     output reg MemtoReg,
128     output reg RegWrite,
129     output reg MemRead,
130     output reg MemWrite,
131     output reg Branch,
132     output reg ALUOp1, ALUOp0
133     output reg Jump
134 );
135 */
136 Control control(
137     .instruction(op),
138     .RegDst(RegDst),
139     .ALUSrc(ALUSrc),
140     .MemtoReg(MemtoReg),
141     .RegWrite(RegWrite),
142     .MemRead(),
143     .MemWrite(MemWrite),
144     .Branch(Branch),
145     .ALUOp1(Op1),
146     .ALUOp0(Op0),
147     .Jump(Jump)
148 );
149
150 /*
151 module Registers          //32 x WIDTH寄存器堆
152 #(parameter WIDTH = 32)  //数据宽度
153 (
154     input clk,            //时钟 (上升沿有效)
155     input [4:0] ra0,       //读端口0地址
156     output reg [WIDTH-1:0] rd0,    //读端口0数据
157     input [4:0] ra1,       //读端口1地址
158     output reg [WIDTH-1:0] rd1,    //读端口1数据
159     input [4:0] wa,        //写端口地址
160     input we,              //写使能, 高电平有效
161     input [WIDTH-1:0] wd   //写端口数据

```

```

162 );
163 */
164 Registers registers(
165     .clk(clk),
166     .ra0(ReadReg1),
167     .rd0(ReadData1),
168     .ra1(ReadReg2),
169     .rd1(ReadData2),
170     .wa(WriteReg),
171     .we(RegWrite),
172     .wd(WriteData)
173 );
174
175 /*
176 module Mux5(
177     input control,
178     input [4:0] in1, in0,
179     output [4:0] out
180 );
181 */
182 Mux5 mux0(
183     .control(RegDst),
184     .in0(ReadReg2),
185     .in1(ReadReg3),
186     .out(WriteReg)
187 );
188
189 Mux32 mux1(
190     .control(ALUSrc),
191     .in0(ReadData2),
192     .in1(extendImm),
193     .out(ALUin2)
194 );
195
196 /*
197 module Sign_extend(
198     input [15:0] imm,
199     output [31:0] extendImm
200 );
201 */
202 Sign_extend signExtend(
203     .imm(imm),
204     .extendImm(extendImm)
205 );
206
207 /*
208 module ALU
209 #(parameter WIDTH = 32)    //数据宽度
210 (output reg [WIDTH-1:0] y,    //运算结果

```

```

211 output reg zf,          //零标志
212 output reg cf,          //进位/借位标志
213 output reg of,          //溢出标志
214 input [WIDTH-1:0] a, b,  //两操作数
215 input [2:0] m           //操作类型
216 );
217 */
218 ALU alu1(
219     .y(ALUresult1),
220     .zf(zero),
221     .cf(),
222     .of(),
223     .a(Mux6out),
224     .b(ALUin2),
225     .m(ALUOp)
226 );
227
228 /*
229 module bus(
230     input clk,
231     input I_step, I_flag, // btn
232     input [15 : 0] I_data,
233     output [15 : 0] O_led,
234     input [7 : 0] Mem_a,
235     input [31 : 0] Mem_d,
236     input Mem_we,
237     output [31 : 0] Mem_spo
238 );
239 // DataMemory 256*32
240 RAM ram (
241     .a(ALUresult1[9 : 2]), // input wire [7 : 0] 地址
242     .d(ReadData2),        // input wire [31 : 0] d 写数据
243     .clk(clk),            // input wire clk
244     .we(MemWrite),        // input wire we 写使能
245     .spo(MemReadData)     // output wire [31 : 0] spo 读数据
246 );*/
247 bus BUS(
248     .clk(clk),
249     .I_step(I_step),
250     .I_flag(I_flag),
251     .I_data(I_data),
252     .O_led(O_led),
253     .Mem_a(ALUresult1[9 : 2]), // input wire [7 : 0] 地址
254     .Mem_d(ReadData2),        // input wire [31 : 0] d 写数据
255     .Mem_we(MemWrite),        // input wire we 写使能
256     .Mem_spo(MemReadData)     // output wire [31 : 0] spo 读数据
257 );
258
259 Mux32 mux2(

```

```

260     .control(MemtoReg),
261     .in0(ALUresult1),
262     .in1(MemReadData),
263     .out(WriteData)
264 );
265
266 endmodule

```

func.v

```

1  module ALU
2  #(parameter WIDTH = 32)    //数据宽度
3  (output reg [WIDTH-1:0] y,    //运算结果
4  output reg zf,                //零标志
5  output reg cf,                //进位/借位标志
6  output reg of,                //溢出标志
7  input [WIDTH-1:0] a, b,      //两操作数
8  input [2:0] m                //操作类型
9  );
10 always@(*)
11     begin
12         case(m)
13             3'b000: // +
14             begin
15                 {cf, y} = a + b;
16                 of = (~a[WIDTH-1] & ~b[WIDTH-1] & y[WIDTH-1])
17                     | (a[WIDTH-1] & b[WIDTH-1] & ~y[WIDTH-1]);
18                 zf = ~|y;
19             end
20             3'b001: // -
21             begin
22                 {cf, y} = a - b;
23                 of = (~a[WIDTH-1] & b[WIDTH-1] & y[WIDTH-1])
24                     | (a[WIDTH-1] & ~b[WIDTH-1] & ~y[WIDTH-1]);
25                 zf = ~|y;
26             end
27             3'b010: // &
28             begin
29                 y = a & b;
30                 zf = ~|y;
31                 cf = 0;
32                 of = 0;
33             end
34             3'b011: // |
35             begin
36                 y = a | b;
37                 zf = ~|y;
38                 cf = 0;
39                 of = 0;

```

```

40         end
41         3'b100: // ^
42         begin
43             y = a ^ b;
44             zf = ~|y;
45             cf = 0;
46             of = 0;
47         end
48         3'b101: // <<
49         begin
50             y = b << a;
51             zf = ~|y;
52             cf = 0; //todo
53             of = 0; //todo
54         end
55         3'b110: // >> 逻辑右移
56         begin
57             y = b >> a;
58             zf = ~|y;
59             cf = 0; //todo
60             of = 0; //todo
61         end
62         default:
63         begin
64             y = 0;
65             zf = 0;
66             cf = 0;
67             of = 0;
68         end
69     endcase
70 end
71 endmodule
72
73 module Registers //32 x WIDTH寄存器堆
74 #(parameter WIDTH = 32) //数据宽度
75 (
76     input clk, //时钟（上升沿有效）
77     input [4:0] ra0, //读端口0地址
78     output reg [WIDTH-1:0] rd0, //读端口0数据
79     input [4:0] ra1, //读端口1地址
80     output reg [WIDTH-1:0] rd1, //读端口1数据
81     input [4:0] wa, //写端口地址
82     input we, //写使能，高电平有效
83     input [WIDTH-1:0] wd //写端口数据
84 );
85 reg [WIDTH-1:0] mem [255:0];
86 // 初始化 RAM 的内容
87 initial
88 begin

```

```

89     //$readmemh("C:/Users/mi/Desktop/text.txt", mem, 0, 255);
90     $readmemh("C:/Users/mi/Desktop/lab6_design/initReg.vec", mem, 0,
255);
91 end
92 // 异步读
93 always@(*)
94 begin
95     rd0 = mem[ra0];
96     rd1 = mem[ra1];
97 end
98 // 同步写
99 always@(posedge clk)
100 begin
101     if(we & wa!=0)
102         mem[wa] <= wd;
103 end
104 endmodule
105
106 module Mux5(
107     input control,
108     input [4:0] in1, in0,
109     output [4:0] out
110 );
111 assign out = control? in1:in0;
112 endmodule
113
114 module Mux32(
115     input control,
116     input [31:0] in1, in0,
117     output [31:0] out
118 );
119 assign out = control? in1:in0;
120 endmodule
121
122 module Sign_extend(
123     input [15:0] imm,
124     output [31:0] extendImm
125 );
126 assign extendImm[15:0] = imm;
127 assign extendImm[31:16] = imm[15] ? 16'hffff : 16'h0000;
128 endmodule
129
130 module Control(
131     input [5:0] instruction,
132     output reg RegDst,
133     output reg ALUSrc,
134     output reg MemtoReg,
135     output reg RegWrite,
136     output reg MemRead,

```

```

137     output reg MemWrite,
138     output reg Branch,
139     output reg ALUOp1, ALUOp0,
140     output reg Jump
141 );
142 // add addi lw sw beq j srl sll xor
143 // x 都归为 0
144 always @(instruction)
145 begin
146     case(instruction)
147         6'b000000: // add func=100000
148             {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
149             MemWrite, Branch, ALUOp1, ALUOp0,
150             Jump} <= 10'b1001000100;
151         6'b100011: // lw
152             {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
153             MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'b0111100000;
154         6'b101011: // sw
155             {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
156             MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'b1x1x0010000;
157         6'b000100: // beq
158             {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
159             MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'bx0x0001010;
160         6'b001000: // addi
161             {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
162             MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'b0101000000;
163         6'b000010: // j
164             {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
165             MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'bxxx000x011;
166         /*6'b000000: // xor func=100110
167             {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
168             MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'b1001000??0;
169         6'b000000: // func sll 000000 srl 000001
170         */
171     endcase
172 end
173 endmodule
174
175 module ALUControl(
176     input Op1, Op0,
177     input [5 : 0] funct,
178     output reg [2 : 0] ALUOp,
179     output reg ShamtSignal
180 );
181 always@(*)
182 begin
183     case({Op1, Op0})
184         2'b00: ALUOp <= 3'b000;
185         2'b01: ALUOp <= 3'b001;

```

```

186         2'b10: begin
187             case(funcnt)
188                 6'b100000: ALUOp <= 3'b000; // +
189                 6'b100010: ALUOp <= 3'b001; // -
190                 6'b100100: ALUOp <= 3'b010; // and
191                 6'b100101: ALUOp <= 3'b011; // or
192                 6'b100110: ALUOp <= 3'b100; // xor
193                 6'b000000: ALUOp <= 3'b101; // <<
194                 6'b000010: ALUOp <= 3'b110; // >>
195                 default: ALUOp <= 3'b111;
196             endcase
197         end
198     endcase
199 end
200 always@(*)
201     if(funcnt == 6'b000000 | funcnt == 6'b000010)
202         ShamSignal <= 1'b1;
203     else
204         ShamSignal <= 1'b0;
205 endmodule
206
207 module PC(
208     input clk,
209     input rst,
210     input [31:0] new_addr,
211     output reg [31:0] cur_addr
212 );
213 initial
214     cur_addr <= 0;
215 always@(posedge clk or posedge rst)
216 begin
217     if(rst)
218         cur_addr <= 0;
219     else
220         cur_addr <= new_addr;
221 end
222 endmodule

```

六、实验总结与建议

从最初对于总线的茫然，到后期结合了上学期计算机系统概论（ICS）所学，我受益匪浅。

找到应用的过程也很有意思，我起初调研到的有趣的应用大多是跟 VGA 有关的。后来无意间想到与其想着用 VGA 做二维细胞自动机，不如用 LED 来显示一维细胞自动机，虽然简单了一点，但是核心不变。

最后想到用Python将细胞自动机的结果可视化，算是给本学期的组原实验画上可爱的句号吧。