计算机组成原理lab6实验报告

PB18000227艾语晨

综合实验

设计了一个带有简单总线系统和单周期CPU的系统，完成计算斐波那契数列的任务

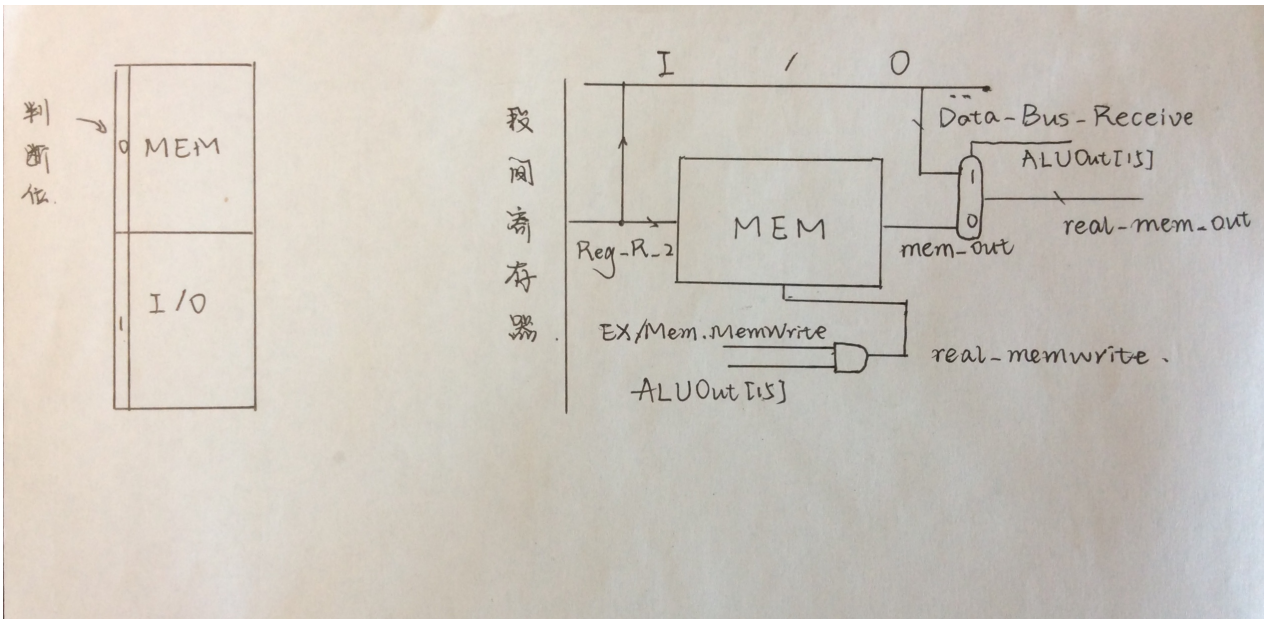
任务描述

用开关输入两个数字 (按钮来确认输入)，作为数列的第0项和第1项。求出数列的第9项并在LED上显示出来

> 附计算结果 (输入为1和2)
>
> 1，2，3，5，8，13，21，34，55，89

实现思路

采用单周期CPU，增加CPU向I/O总线的接口。内存依然采用直接模块调用的方式 (立即响应)，略去了二者的接口；开关和LED均为单向数据传输，故轮询计数和设备选择线亦省去。外设与访存一样采用LW和SW指令，只是在立即数的最高位有区分 (调用I/O为1，访问主存为0)



mem段的数据选择通路如上图所示

实验代码

- 顶层设计

```
module top
    (
        input clk, rst,
        input button_real,
        input [15:0] SW_real
    );

    wire button;                                        // put to SW
Interface
    wire [15:0] SW_data;                                // put to SW
Interface
    wire [31:0] Data_to_CPU, Data_from_CPU;
    wire [1:0] status;
    wire launch, launch_sw, launch_led;
    wire Device_Choose;
    wire catch;
    wire [15:0] LED_data;

    assign launch_sw = ~Device_Choose & launch;
    assign launch_led = Device_Choose & launch;

    sin_CPU CPU (
        .clk(clk),
        .rst(rst),
        .Data_Bus_Receive(Data_to_CPU),
        .Status_Bus_Receive(status),
        .launch(launch),
        .Device_Choose(Device_Choose),
        .catch(catch),
        .Data_Bus_Send(Data_from_CPU)
    );

    SW SW (
        .clk(clk),
        .rst(rst),
        .button_real(button_real),
        .SW_real(SW_real),
        .button(button),
        .SW_out(SW_data)
    );

    IO_Interface_SW SWITCH (
        .clk(clk),
        .rst(rst),
        .Data_IO_In(SW_data),
        .button(button),
        .launch(launch_sw),
        .catch(catch),
        .Data_Bus_Out(Data_to_CPU),
```

```
        .Status_Bus(status)
    );

    IO_Interface_LED LED (
        .clk(clk),
        .rst(rst),
        .Data_Bus_In(Data_from_CPU),
        .launch(launch_led),
        .LED_return(LED_data)
    );
endmodule
```

CPU

```
module sin_CPU
    #(parameter WIDTH = 32)
    (
        input clk, rst,
        input [31:0] Data_Bus_Receive,
        input [1:0] Status_Bus_Receive,
        output launch,
        output Device_Choose,                   // 0 for SWITCH, 1 for LED
        output catch,
        output [31:0] Data_Bus_Send
    );

    localparam [5:0] R_TYPE = 6'b000000;
    localparam [5:0] LW = 6'b100011;
    localparam [5:0] SW = 6'b101011;
    localparam [5:0] BEQ = 6'b000100;
    localparam [5:0] BNE = 6'b000101;
    localparam [5:0] J_TYPE = 6'b000010;
    localparam [5:0] ADDI = 6'b001000;
    localparam LED = 1'b1;
    localparam SWITCH = 1'b0;

    reg [WIDTH-1:0] PC;                 // program counter
    // reg [WIDTH-1:0] NPC;             // new PC
    reg [2:0] alu_ctrl;                 // ALU_ctrl
    reg regdst, jump, branch, memread, memtoreg, memwrite, alusrc, regwrite;
    reg [1:0] aluop;

    // comments are on the circuit
    wire [7:0] pc_addr;
    wire [WIDTH-1:0] npc, fin_npc;
    wire [WIDTH-1:0] ins;
    wire [5:0] ins_ctrl;
    wire [4:0] ins_reg_1, ins_reg_2, ins_reg_3, ins_reg_write;
```

```
    wire [15:0] imme_addr;
    wire [WIDTH-1:0] read_data_1, read_data_2, write_data;
    wire [WIDTH-1:0] extend_addr;
    wire [WIDTH-1:0] jump_addr;
    wire RegDst, RegWrite, ALUSrc, MemRead, MemWrite, MemtoReg, Branch,
PCSrc, Jump;
    wire [1:0] ALUOp;
    wire [WIDTH-1:0] ALU_a, ALU_b, ALU_result;
    wire ALU_Zero;
    wire [2:0] ALU_ctrl;
    wire cf, of, sf;
    wire [WIDTH-1:0] beq_result, not_jump;
    wire [WIDTH-1:0] Mem_Out;
    wire [WIDTH-1:0] real_mem_out;              // choose from mem
result and I/O result, MEM stage
    wire real_memwrite;                         // if MemWrite_EM (is
SW) and not to I/O, than it's real, MEM stage
    wire IO_Stall;                              // if need to stall in
case of waiting I/O, MEM stage

    assign pc_addr = PC[9:2];
    // assign npc = NPC;
    assign RegDst = regdst;
    assign RegWrite = regwrite;
    assign ALUSrc = alusrc;
    assign MemRead = memread;
    assign MemWrite = memwrite;
    assign MemtoReg = memtoreg;
    assign Branch = branch;
    assign Jump = jump;
    assign ALUOp = aluop;

    // inst of a program
    dist_inst_rom instruction(
        .a(pc_addr),
        .spo(ins)
    );

    // for the MUX at the entrance of reg_pile
    mux #(5) mux_reg (
        .m(RegDst),
        .in_1(ins_reg_2),
        .in_2(ins_reg_3),
        .out(ins_reg_write)
    );

    // for the MUX at the entrance of ALU
    mux #(32) mux_alu (
        .m(ALUSrc),
        .in_1(read_data_2),
        .in_2(extend_addr),
        .out(ALU_b)
```

```
    );

    mux #(32) mux_beq (
        .m(PCSrc),
        .in_1(npc),
        .in_2(beq_result),
        .out(not_jump)
    );

    mux #(32) mux_jump (
        .m(Jump),
        .in_1(not_jump),
        .in_2(jump_addr),
        .out(fin_npc)
    );

    mux #(32) mux_wb (
        .m(MemtoReg),
        .in_1(ALU_result),
        .in_2(real_mem_out),
        .out(write_data)
    );

    assign npc = PC + 32'd4;
    // always @(posedge clk) begin          // pre-load
    //  NPC = PC + 32'd4;
    // end

    // inst of a reg_pile, ID
    reg_file register_file (
        .clk(clk),
        .ra0(ins_reg_1),
        .ra1(ins_reg_2),
        .wa(ins_reg_write),
        .rd0(read_data_1),
        .rd1(read_data_2),
        .wd(write_data),
        .we(RegWrite)
    );

    // ID
    assign ins_ctrl = ins[WIDTH-1:26];
    assign ins_reg_1 = ins[25:21];
    assign ins_reg_2 = ins[20:16];
    assign ins_reg_3 = ins[15:11];
    assign imme_addr = ins[15:0];
    // sign extend
    assign extend_addr = {((imme_addr[15]) ? 16'hffff : 16'h0000),
imme_addr};
    // jump addr
    assign jump_addr = {npc[31:28], ins[25:0], 2'b00};

```

```
134     // control unit
135     always @(*) begin
136         {regdst, jump, branch, memread, memtoreg, alusrc, regwrite,
    memwrite, aluop} = 10'b0;
137         case (ins[31:26])
138             6'b000000: begin            // R-type
139                 regdst = 1'b1;
140                 regwrite = 1'b1;
141                 aluop = 2'b10;
142             end
143             6'b100011: begin            // lw
144                 alusrc = 1'b1;
145                 memtoreg = 1'b1;
146                 regwrite = 1'b1;
147                 memread = 1'b1;
148             end
149             6'b101011: begin            // sw
150                 alusrc = 1'b1;
151                 memwrite = 1'b1;
152             end
153             6'b000100: begin            // beq
154                 branch = 1'b1;
155                 aluop = 2'b01;
156             end
157             6'b000101: begin            // bne
158                 branch = 1'b1;
159                 aluop = 2'b01;
160             end
161             6'b001000: begin            // addi
162                 alusrc = 1'b1;
163                 regwrite = 1'b1;
164             end
165             6'b000010: begin            // jump
166                 jump = 1'b1;
167             end
168             default: {regdst, jump, branch, memread, memtoreg, alusrc,
    regwrite, memwrite, aluop} = 'dz;
169         endcase
170     end
171
172     // ALU control
173     assign ALU_ctrl = alu_ctrl;
174     always @(*) begin
175         case (ALUOp)
176             2'b00: begin                // LW & SW
177                 alu_ctrl = 3'b010;
178             end
179             2'b01: begin                // BEQ & BNE
180                 alu_ctrl = 3'b110;
181             end
182             2'b10: begin                // R-type
183                 case (ins[5:0])
```

```
                    6'b100000: begin    // add
                        alu_ctrl = 3'b010;
                    end
                    6'b100010: begin    // sub
                        alu_ctrl = 3'b110;
                    end
                    6'b100100: begin    // and
                        alu_ctrl = 3'b000;
                    end
                    6'b100101: begin    // or
                        alu_ctrl = 3'b001;
                    end
                    6'b101010: begin    // slt
                        alu_ctrl = 3'b111;
                    end
                    default: alu_ctrl = 'dz;
                endcase
            end
            default: alu_ctrl = 'dz;
        endcase
    end

    // EX
    assign ALU_a = read_data_1;
    ALU alu (
        .y(ALU_result),
        .zf(ALU_Zero),
        .cf(cf),
        .of(of),
        .sf(sf),
        .a(ALU_a),
        .b(ALU_b),
        .m(ALU_ctrl)
    );

    assign PCSrc = (ins_ctrl == BNE) ? (~ALU_Zero & Branch) : (ALU_Zero & Branch);
    assign beq_result = npc + {extend_addr[29:0], 2'b00};

    // MEM
    assign real_memwrite = MemWrite & ~ALU_result[15];  // if ALU_result[15] or Imme[15] is 1, then go to I/O
    assign real_mem_out = ALU_result[15] ? Data_Bus_Receive : Mem_Out;
    assign launch = ALU_result[15];
    assign Device_Choose = MemWrite ? LED : SWITCH;
    // CAUTION: this stall should be done on ALL FOUR stage-registers and PC
    assign IO_Stall = (launch & ~(Status_Bus_Receive == 2'b10)) & (ins_ctrl == LW);
    assign catch = launch & (Status_Bus_Receive == 2'b10);
    assign Data_Bus_Send = read_data_2;

```

```
    wire [8:0] read_mem_addr;
    assign read_mem_addr = ALU_result[10:2];
    dist_data_ram memory (
        .a(read_mem_addr),
        .d(read_data_2),
        .clk(clk),
        .we(real_memwrite),
        .spo(Mem_Out)
    );

    // change PC
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            PC = 32'b0;
        end
        else if (~IO_Stall) begin
            PC = fin_npc;
        end
        else begin
            PC = PC;
        end
    end

endmodule
```

I/O接口

```
module IO_Interface_SW
    (
        input clk, rst,
        // input [31:0] Instruction_Bus,    // from bus(CPU)
        input [15:0] Data_IO_In,        // from I/O devices
        input button,                   // if pressed (1), read-data from SW is valid
        input launch,                   // lasts for 1 cycle, if 1, means CPU is waking up I/O devices
        input catch,                    // lasts for 1 cycle, if 1, means CPU is ending transmission
        output [31:0] Data_Bus_Out,     // to bus(CPU)
        output [1:0] Status_Bus         // to bus(CPU), D&B
    );

    reg [31:0] Ins_Reg;                 // save ins from CPU
    reg [15:0] DBR;                     // data buffer in Interface, FIFO
    // reg buffer_count_curr;               // count for items in buffer
    // reg buffer_count_next;               // count for items in buffer

    // wake up, saving instruction into Ins_Reg
    // always @(posedge clk) begin
```

```
    //  if (rst || catch) begin
    //      Ins_Reg <= 0;
    //  end
    //  else if (launch) begin
    //      Ins_Reg <= Instruction_Bus;
    //  end
    //  else begin
    //      Ins_Reg <= Ins_Reg;
    //  end
    // end

    // start to prepare
    reg [1:0] curr_state, next_state;
    localparam IDLE = 2'b00;
    localparam DONE = 2'b10;
    localparam BUSY = 2'b01;

    initial begin
        next_state = IDLE;
    end

    always @(posedge clk) begin
        if (rst) begin
            curr_state <= IDLE;
        end
        else begin
            curr_state <= next_state;
        end
    end

    assign Status_Bus = curr_state;
    assign Data_Bus_Out = {((DBR[15]) ? 16'hffff : 16'h0000), DBR};

    always @(*) begin
        if (curr_state == IDLE && launch) begin
            next_state = BUSY;
        end
        else if (curr_state == BUSY && button) begin
            next_state = DONE;
        end
        else if (curr_state == DONE && catch) begin
            next_state = IDLE;
        end
        else begin
            next_state = next_state;
        end
    end

    always @(posedge button) begin
        if (rst) begin
            DBR <= 0;
        end
```

```
        if (curr_state == BUSY) begin
            DBR <= Data_IO_In;
        end
        else begin
            DBR <= DBR;
        end
    end
endmodule

module IO_Interface_LED
    (
        input clk, rst,
        input [31:0] Data_Bus_In,       // from bus(CPU)
        // input [31:0] Instruction_Bus,    // from bus(CPU)
        input launch,                   // lasts for 1 cycle, if 1, means CPU is waking up I/O devices
        output [15:0] LED_return        // back to top file, assuming that there ARE LEDs
    );

    reg [15:0] Data_IO_Out;             // to I/O devices
    reg [31:0] Ins_Reg;                 // save ins from CPU
    reg [15:0] DBR [0:1];               // data buffer in Interface, FIFO

    // wake up, saving instruction into Ins_Reg
    // always @(posedge clk) begin
    //  if (launch) begin
    //      Ins_Reg <= Instruction_Bus;
    //  end
    //  else begin
    //      Ins_Reg <= Ins_Reg;
    //  end
    // end

    // start to prepare
    always @(posedge clk) begin
        if (rst) begin
            Data_IO_Out <= 0;
        end
        else if (launch) begin
            Data_IO_Out <= Data_Bus_In[15:0];
        end
        else begin
            Data_IO_Out <= Data_IO_Out;
        end
    end

    LED LED (
        .clk(clk),
        .LED_data(Data_IO_Out),
        .LED_out(LED_return)
```

```
    );
endmodule
```

SW和LED (I/O devices)

SW

```
module SW
    (
        input clk, rst,
        input button_real,
        input [15:0] SW_real,
        output button,
        output [15:0] SW_out
    );

    wire button_clr, button_edge;

    jitter_clr clr_button (
        .clk(clk),
        .button(button_real),
        .button_clean(button_clr)
    );

    signal_edge edge_button (
        .clk(clk),
        .button(button_clr),
        .button_edge(button_edge)
    );

    assign button = button_edge;
    assign SW_out = SW_real;
endmodule

module jitter_clr(
    input clk,
    input button,
    output button_clean
    );

    reg [3:0] cnt;

    always @(posedge clk) begin
        if (button == 1'b0) begin
            cnt <= 4'h0;
        end
        else if (cnt < 4'h8) begin
            cnt <= cnt + 1'b1;
        end
    end
```

```
44
45      assign button_clean = cnt[3];
46  endmodule
47
48  module signal_edge(
49      input clk,
50      input button,
51      output button_edge
52      );
53
54      reg button_r1, button_r2;
55
56      always @(posedge clk) begin
57          button_r1 <= button;
58      end
59
60      always @(posedge clk) begin
61          button_r2 <= button_r1;
62      end
63
64      assign button_edge = button_r1 & ~button_r2;
65  endmodule
```

LED

```
1   module LED
2       (
3           input clk,
4           input [15:0] LED_data,
5           output reg [15:0] LED_out
6       );
7
8       always @(posedge clk) begin
9           if (LED_data == 'dz) begin
10              LED_out <= 16'b0;
11          end
12          else begin
13              LED_out <= LED_data;
14          end
15      end
16  endmodule
17
```

MUX 和 reg_file

MUX

```
module mux
#(parameter WIDTH = 32)
(
    input m, // control signal
    input [WIDTH-1:0] in_1,in_2,
    output [WIDTH-1:0] out
);
    assign out=(m == 1'b0 ? in_1 : in_2);

endmodule // mux
```

Reg_file

```
module reg_file
    #(parameter WIDTH = 32)
    (
        input clk,
        input [4:0] ra0,            // read port 0 addr
        output [WIDTH-1:0] rd0,     // read port 0 data
        input [4:0] ra1,            // read port 1 addr
        output [WIDTH-1:0] rd1,     // read port 1 data
        input [4:0] wa,             // write port addr
        input we,                   // write enable, valid at '1'
        input [WIDTH-1:0] wd        // write port data
    );

    reg [WIDTH-1:0] reg_file [0:31];

    assign rd0 = reg_file[ra0];
    assign rd1 = reg_file[ra1];

    integer i;                      // loop varible
    initial begin
        for (i = 0; i < 32; i = i + 1) begin
            reg_file [i] = 0;
        end
    end

    always @(posedge clk) begin
        if (we && wa != 4'b0) begin
            reg_file[wa] = wd;
        end
    end
endmodule
```

测试文件

Test.s

```
# 采用类似于斐波那契数列的计算方式
# 用开关和按键输入两个数，作为数列的 a0 和 a1，求 a9
# s0, s1 用来存储两个计算数据，s0 < s1
# t1 是循环变量，记录当前最大的数是第几个，最后取出来 s0 就行

_Input:
        lw      $s0, -32768($0)     # read from I/O                 0
        lw      $s1, -32768($0)     # read from I/O again           4
        addi    $sp, $0, 0x7fc      # $sp = $t1 + 0x7fc             8
        addi    $t1, $0, 1          # $t1 = $0 + 1                  12
        addi    $t2, $0, 1          # $t2 = $0 + 1                  16
        addi    $t3, $0, 9          # $t3 = $0 + 9                  20
_Stack:
        sw      $s0, 0($sp)         # store to stack top            24
        sw      $s1, -4($sp)        # store to stack                28
        j       _Sort               # jump to _Sort                 32
_Sort:
        slt     $t0, $s0, $s1       # if $s0 < $s1, $t0 = 1         36
        beq     $t0, $t2, _Cal      # if $t0 == $t1 then _Cal       40
                # SWAP
                lw      $s0, -4($sp)# save primary $s1 to $s0       44
                lw      $s1, 0($sp) # save primary $s0 to $s1       48
_Cal:
        add     $s0, $s1, $s0       # $s0 = $s1 + $s0               52
        addi    $t1, $t1, 1         # $t1 = $t1 + 1                 56
        bne     $t1, $t3, _Stack    # if $t1 != $t3 then _Stack     60
_Output:
        sw      $s0, -32768($0)     # to LED                        64
        j       _Success            # jump to _Success              68
_Success:
        j       _Success            # jump to _Success              72
```

test.coe

```
memory_initialization_radix  = 16;
memory_initialization_vector =
8c108000
8c118000
201d07fc
20090001
200a0001
200b0009
afb00000
afb1fffc
08000009
0211402a
110a0002
8fb0fffc
8fb10000
02308020
```

```
21290001
152bfff6
ac108000
08000012
08000012
```

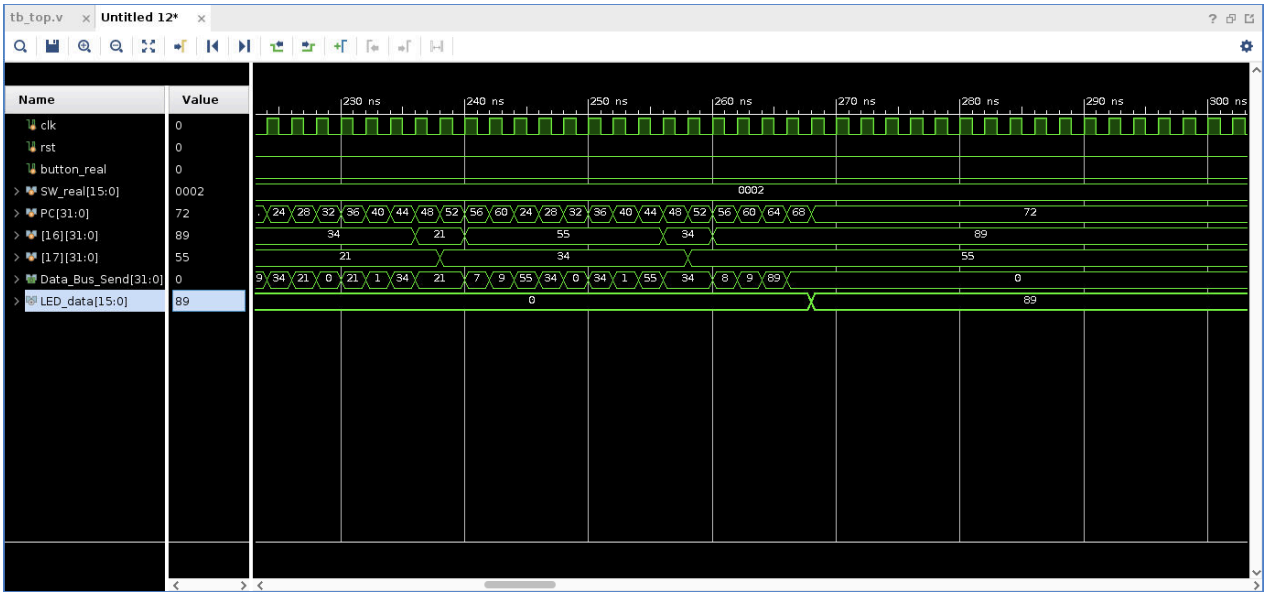
仿真

仿真文件

```
module tb_top();
    reg clk, rst;
    reg button_real;
    reg [15:0] SW_real;

    top fake_computer (
        .clk(clk),
        .rst(rst),
        .button_real(button_real),
        .SW_real(SW_real)
    );

    initial
    begin
        clk = 1;
        rst = 1;
        # 2 rst = 0;
        # 998 $finish;
    end

    initial
    begin
        button_real = 0;
        # 20 button_real = 1;
        # 20 button_real = 0;
        # 40 button_real = 1;
        # 20 button_real = 0;
        # 10 button_real = 0;
        # 20 button_real = 0;
        # 870 $finish;
    end

    initial
    begin
        SW_real = 0;
        # 10 SW_real = 2'd1;
        # 50 SW_real = 2'd2;
        # 940 $finish;
    end
```

```
40
41      always
42      # 1 clk = ~clk;
43  endmodule
```

仿真结果



可以看到，最终LED的输出结果为正确的89