基本单周期CPU设计 - 实验报告

- 学号：517021910653
- 姓名：王祖来

实验目的

1. 理解计算机5大组成部分的协调工作原理，理解存储程序自动执行的原理。
2. 掌握运算器、存储器、控制器的设计和实现原理。重点掌握控制器设计原理和实现方法。
3. 掌握I/O端口的设计方法，理解I/O地址空间的设计方法。
4. 会通过设计I/O端口与外部设备进行信息交互。

实验仪器和平台

- DE1-SoC实验板
- QuartusII13.1
- ModelSim 10.1

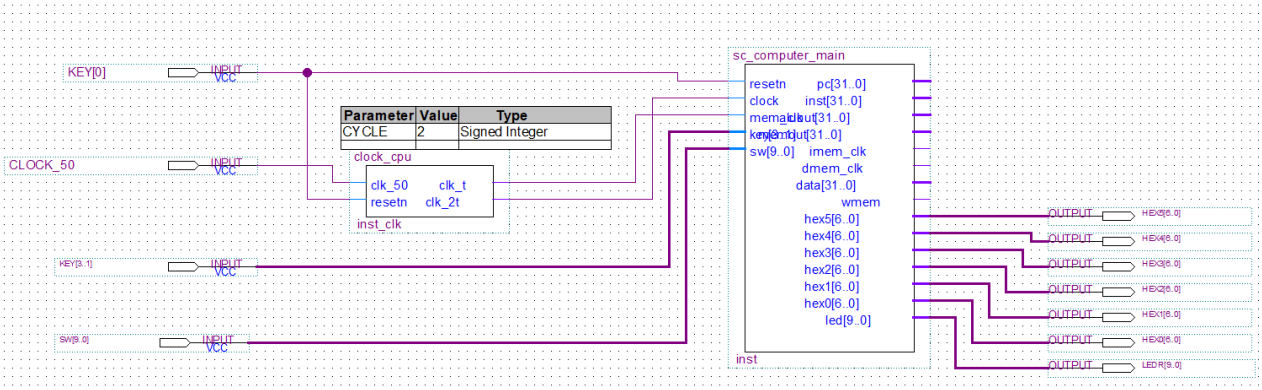
实验内容和任务

1. 采用Verilog HDL在quartusII中实现基本的具有20条MIPS指令的单周期CPU设计。
2. 利用实验提供的标准测试程序代码，完成仿真测试。
3. 采用I/O统一编址方式，即将输入输出的I/O地址空间，作为数据存取空间的一部分，实现CPU与外部设备的输入输出端口设计。实验中可采用高端地址。
4. 利用设计的I/O端口，通过lw指令，输入DE2实验板上的按键等输入设备信息。即将外部设备状态，读到CPU内部寄存器。
5. 利用设计的I/O端口，通过sw指令，输出对DE2实验板上的LED灯等输出设备的控制信号（或数据信息）。即将对外部设备的控制数据，从CPU内部的寄存器，写入到外部设备的相应控制寄存器（或可直接连接至外部设备的控制输入信号）。
6. 利用自己编写的程序代码，在自己设计的CPU上，实现对板载输入开关或按键的状态输入，并将判别或处理结果，利用板载LED灯或7段LED数码管显示出来。
7. 例如，将一路4bit二进制输入与另一路4bit二进制输入相加，利用两组分别2个LED数码管以10进制形式显示“被加数”和“加数”，另外一组LED数码管以10进制形式显示“和”等。（具体任务形式不做严格规定，同学可自由创意）。
8. 实现MIPS基本20条指令。

9. 在实验报告中，汇报自己的设计思想和方法；并以汇编语言的形式，提供以上指令集（MIPS）的应用功能的程序设计代码，并提供程序主要流程图。

设计过程

顶层设计



板载50MHz时钟接入时钟模块，生成1倍周期和2倍周期的时钟供单周期CPU主模块使用。3个按键和10个开关接入单周期CPU主模块。剩下的1个按键作为单周期CPU主模块和时钟模块的复位按键。单周期CPU主模块输出到6个七段数码管和9个发光二极管。主模块的其他端口不接到输出的引脚上，只用于仿真验证时查看波形。

主模块

```
module sc_computer_main
(resetn,clock,mem_clk,pc,inst,aluout,memout,imem_clk,dmem_clk,
  key, data, wmem, sw, hex5, hex4, hex3, hex2, hex1, hex0, led);

   input resetn,clock,mem_clk;
  input [3:1] key;
  input [9:0] sw;
   output [31:0] pc,inst,aluout,memout;
   output        imem_clk,dmem_clk;
   output   [31:0] data;
   output          wmem;
  output [6:0] hex5, hex4, hex3, hex2, hex1, hex0;
  output [9:0] led;

   sc_cpu cpu (clock,resetn,inst,memout,pc,wmem,aluout,data);           // CPU
module.
   sc_instmem  imem (pc,inst,clock,mem_clk,imem_clk);                   //
instruction memory.
   sc_datamem  dmem (resetn,aluout,data,memout,wmem,clock,mem_clk,dmem_clk,
    sw,key,hex5,hex4,hex3,hex2,hex1,hex0,led); // data memory.
```

```
endmodule
```

主模块由CPU、指令存储和数据存储三部分组成

时钟模块

```
module clock_cpu(clk_50, resetn, clk_t, clk_2t);
  input   clk_50, resetn;
  output  clk_t;
  output  clk_2t;
  reg   clk_t;
  reg   clk_2t;
  reg [31:0] cnt;
  parameter CYCLE = 2;  // the number of cycles of 1 clk_t on board clock

  initial begin
    clk_t <= 0;
    clk_2t <= 0;
    cnt <= 0;
  end

  always @ (posedge clk_50 or negedge resetn)
    begin
      if (!resetn) begin
        clk_t <= 0;
        clk_2t <= 0;
        cnt <= 0;
      end
      else begin
        if (cnt >= CYCLE/2 - 1) begin
          cnt <= 0;
          clk_t <= ~clk_t;
          if (!clk_t) begin
            clk_2t <= ~clk_2t;
          end
        end
        else begin
          cnt <= cnt + 1;
        end
      end
    end
endmodule
```

该模块利用计数器生成一倍周期的时钟信号作为CPU的mem_clk信号，生成两倍周期的时钟信号作为CPU的clk信号。一倍周期为CYCLE个板载时钟周期。

CPU

```
module sc_cpu (clock,resetn,inst,mem,pc,wmem,alu,data);
   input [31:0] inst,mem;
   input clock,resetn;
   output [31:0] pc,alu,data;
   output wmem;

   wire [31:0]   p4,bpc,npc,adr,ra,alua,alub,res,alu_mem;
   wire [3:0]    aluc;
   wire [4:0]    reg_dest,wn;
   wire [1:0]    pcsource;
   wire          zero,wmem,wreg,regrt,m2reg,shift,aluimm,jal,sext;
   wire [31:0]   sa = { 27'b0, inst[10:6] }; // extend to 32 bits from sa for
shift instruction
   wire          e = sext & inst[15];          // positive or negative sign at
sext signal
   wire [15:0]   imm = {16{e}};                // high 16 sign bit
   wire [31:0]   immediate = {imm,inst[15:0]}; // sign extend to high 16
  wire [31:0]   offset = {imm[13:0],inst[15:0],1'b0,1'b0};   //offset(include
sign extend)

   dff32 ip (npc,clock,resetn,pc);  // define a D-register for PC


   assign p4 = pc + 32'h4;       // modified
   assign adr = p4 + offset;     // modified

   wire [31:0] jpc = {p4[31:28],inst[25:0],1'b0,1'b0}; // j address

   sc_cu cu (inst[31:26],inst[5:0],zero,wmem,wreg,regrt,m2reg,
                        aluc,shift,aluimm,pcsource,jal,sext);


   mux2x32 alu_b (data,immediate,aluimm,alub);
   mux2x32 alu_a (ra,sa,shift,alua);
   mux2x32 result(alu,mem,m2reg,alu_mem);
   mux2x32 link (alu_mem,p4,jal,res);
   mux2x5 reg_wn (inst[15:11],inst[20:16],regrt,reg_dest);
   assign wn = reg_dest | {5{jal}}; // jal: r31 <-- p4;      // 31 or reg_dest
   mux4x32 nextpc(p4,adr,ra,jpc,pcsource,npc);
   regfile rf (inst[25:21],inst[20:16],res,wn,wreg,clock,resetn,ra,data);
   alu al_unit (alua,alub,aluc,alu,zero);
endmodule
```

CPU模块由程序计数器寄存器（ip）、控制单元（sc_cu），寄存器文件（regfile）和算术逻辑单元（alu）与其他线路和多路选择器连接而成。

控制单元

```
module sc_cu (op, func, z, wmem, wreg, regrt, m2reg, aluc, shift,
              aluimm, pcsource, jal, sext);
   input  [5:0] op,func;
   input        z;
   output       wreg,regrt,jal,m2reg,shift,aluimm,sext,wmem;
   output [3:0] aluc;
   output [1:0] pcsource;
   wire r_type = ~|op;
   wire i_add = r_type & func[5] & ~func[4] & ~func[3] &
                ~func[2] & ~func[1] & ~func[0];          //100000
   wire i_sub = r_type & func[5] & ~func[4] & ~func[3] &
                ~func[2] &  func[1] & ~func[0];          //100010

   //  please complete the deleted code.

   wire i_and = r_type & func[5] & ~func[4] & ~func[3] &
                func[2] & ~func[1] & ~func[0];            //100100
   wire i_or  = r_type & func[5] & ~func[4] & ~func[3] &
                func[2] & ~func[1] & func[0];           //100101

   wire i_xor = r_type & func[5] & ~func[4] & ~func[3] &
                func[2] & func[1] & ~func[0];           //100110
   wire i_sll = r_type & ~func[5] & ~func[4] & ~func[3] &
                ~func[2] & ~func[1] & ~func[0];          //000000
   wire i_srl = r_type & ~func[5] & ~func[4] & ~func[3] &
                ~func[2] & func[1] & ~func[0];            //000010
   wire i_sra = r_type & ~func[5] & ~func[4] & ~func[3] &
                ~func[2] & func[1] & func[0];           //000011
   wire i_jr  = r_type & ~func[5] & ~func[4] & func[3] &
                ~func[2] & ~func[1] & ~func[0];          //001000

   wire i_addi = ~op[5] & ~op[4] &  op[3] & ~op[2] & ~op[1] & ~op[0]; //001000
   wire i_andi = ~op[5] & ~op[4] &  op[3] &  op[2] & ~op[1] & ~op[0]; //001100

   wire i_ori  = ~op[5] & ~op[4] &  op[3] &  op[2] & ~op[1] &  op[0]; //001101
   wire i_xori = ~op[5] & ~op[4] &  op[3] &  op[2] &  op[1] & ~op[0]; //001110
   wire i_lw   =  op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] &  op[0]; //100011
   wire i_sw   =  op[5] & ~op[4] &  op[3] & ~op[2] &  op[1] &  op[0]; //101011
   wire i_beq  = ~op[5] & ~op[4] & ~op[3] &  op[2] & ~op[1] & ~op[0]; //000100
   wire i_bne  = ~op[5] & ~op[4] & ~op[3] &  op[2] & ~op[1] &  op[0]; //000101
   wire i_lui  = ~op[5] & ~op[4] &  op[3] &  op[2] &  op[1] &  op[0]; //001111
   wire i_j    = ~op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] & ~op[0]; //000010
   wire i_jal  = ~op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] &  op[0]; //000011
```

```
    assign pcsource[1] = i_jr | i_j | i_jal;
    assign pcsource[0] = ( i_beq & z ) | (i_bne & ~z) | i_j | i_jal ;

    assign wreg = i_add | i_sub | i_and | i_or   | i_xor  |
                  i_sll | i_srl | i_sra | i_addi | i_andi |
                  i_ori | i_xori | i_lw | i_lui  | i_jal;

    assign aluc[3] = i_sra;
    assign aluc[2] = i_sub | i_or  | i_srl | i_sra | i_ori |
              i_beq | i_bne | i_lui;
    assign aluc[1] = i_xor | i_sll | i_srl | i_sra | i_xori |
              i_lui;
    assign aluc[0] = i_and | i_or  | i_sll | i_srl | i_sra |
              i_andi | i_ori;
    assign shift   = i_sll | i_srl | i_sra ;

    assign aluimm  = i_addi | i_andi | i_ori | i_xori | i_lw |
              i_sw  | i_lui;
    assign sext    = i_addi | i_lw  | i_sw  | i_beq | i_bne;
    assign wmem    = i_sw;
    assign m2reg   = i_lw;
    assign regrt   = i_addi | i_andi | i_ori | i_xori | i_lw |
              i_lui;
    assign jal     = i_jal;

endmodule
```

控制单元读取指令，对指令的op部分和func部分进行分析得出当前CPU所处理的指令，并产生和输出控制信号到CPU的各个部分。

指令存储单元

```
module sc_instmem (addr,inst,clock,mem_clk,imem_clk);
    input  [31:0] addr;
    input         clock;
    input         mem_clk;
    output [31:0] inst;
    output        imem_clk;

    wire          imem_clk;

    assign  imem_clk = clock & ( ~ mem_clk );

    lpm_rom_irom irom (addr[8:2],imem_clk,inst);

endmodule
```

指令存储单元使用了quartus的宏lpm_rom_irom实现，rom的存储空间大小为7位。输入地址（addr），输出指令（inst）和用于通知CPU取指的时钟信号（imem_clk）。

数据存储单元

```
module sc_datamem (resetn,addr,datain,dataout,we,clock,mem_clk,dmem_clk,
  sw,key,hex5,hex4,hex3,hex2,hex1,hex0,led);

  input       resetn;
   input  [31:0]  addr;
   input  [31:0]  datain;

   input          we, clock,mem_clk;
  input  [9:0]  sw;
  input  [3:1]  key;
   output [31:0]  dataout;
   output         dmem_clk;
  output [6:0]  hex5, hex4, hex3, hex2, hex1, hex0;
  output [9:0]  led;
  reg  [31:0] dataout;
  reg  [6:0]  hex5, hex4, hex3, hex2, hex1, hex0;
  reg  [9:0]  led;

  wire   [31:0] mem_dataout;
   wire           dmem_clk;
   wire           write_enable;
   assign         write_enable = we & ~clock & (addr[31:8] != 24'hffffff);  //
leave 8 bit address for I/O port

   assign         dmem_clk = mem_clk & ( ~ clock) ;

  // synchronously input data to dram
   lpm_ram_dq_dram  dram(addr[6:2],dmem_clk,datain,write_enable,mem_dataout );

  // synchronously input data to I/O device(hex and led) or reset data on I/O
device
  always @ (posedge dmem_clk or negedge resetn)
    begin
      if (!resetn) begin
        hex5 <= 7'b1111111;
        hex4 <= 7'b1111111;
        hex3 <= 7'b1111111;
        hex2 <= 7'b1111111;
        hex1 <= 7'b1111111;
        hex0 <= 7'b1111111;
        led <= 10'b0;
      end
      else if (we) begin
```

```
        case (addr)
          32'hffffff00: hex0 <= datain[6:0];
          32'hffffff10: hex1 <= datain[6:0];
          32'hffffff20: hex2 <= datain[6:0];
          32'hffffff30: hex3 <= datain[6:0];
          32'hffffff40: hex4 <= datain[6:0];
          32'hffffff50: hex5 <= datain[6:0];
          32'hffffff60: led <= datain[9:0];
        endcase
      end
    end

  // asynchronously output data from I/O device(switch or key) or dram
  always @ (*)
    begin
      case (addr)
        32'hffffff70:  dataout <= {29'b0, key};
        32'hffffff80: dataout <= {22'b0, sw};
        default: dataout <= mem_dataout;
      endcase
    end
endmodule
```

数据存储单元使用了quartus的宏lpm_ram_dq_dram实现，rom的存储空间大小为5位。输入地址（addr）、数据（datain）和写信号（we），输出数据（dataout）和用于通知CPU取数据的时钟信号（dmem_clk）。

I/O设计：预留地址（0xffffff00 - 0xffffffff）作为I/O端口

- 地址0xffffff00 - 0xffffff50的写入信号会分别输入到6个七段数码管上
- 地址0xffffff60 - 0xffffff69的写入信号会输入到10个发光二极管上
- 对地址0xffffff70的读信号会将key[3:1]的按键状态作为低3位与高29位的0合并输出
- 对地址0xffffff80的读信号会将sw[9:0]的开关状态作为低10位与高22位的0合并输出
- 对非预留地址的读或写时，write_enable=1，会在lpm_ram_dq_dram中进行读写

寄存器文件

```
module regfile (rna,rnb,d,wn,we,clk,clrn,qa,qb);
   input [4:0] rna,rnb,wn;
   input [31:0] d;
   input we,clk,clrn;

   output [31:0] qa,qb;

   reg [31:0] register [1:31]; // r1 - r31
```

```
   assign qa = (rna == 0)? 0 : register[rna]; // read
   assign qb = (rnb == 0)? 0 : register[rnb]; // read

   always @(posedge clk or negedge clrn) begin
      if (clrn == 0) begin: reset // reset
      integer i;
         for (i=1; i<32; i=i+1)
            register[i] <= 0;
      end else begin
         if ((wn != 0) && (we == 1))          // write
            register[wn] <= d;
      end
   end
endmodule
```

一共有32个寄存器，其中0号寄存器永远为0。读寄存器为异步操作，qa和qb会输出rna和rnb所对应寄存器的内容。写寄存器为同步操作，CPU时钟上升沿时，若写信号为1，则将数据（d）写入wn所对应寄存器。

算术逻辑单元

```
module alu (a,b,aluc,s,z);
   input [31:0] a,b;
   input [3:0] aluc;
   output [31:0] s;
   output        z;
   reg [31:0] s;
   reg        z;
   always @ (a or b or aluc)
      begin                                  // event
         casex (aluc)
             4'bx000: s = a + b;             //x000 ADD
             4'bx100: s = a - b;             //x100 SUB
             4'bx001: s = a & b;             //x001 AND
             4'bx101: s = a | b;             //x101 OR
             4'bx010: s = a ^ b;             //x010 XOR
             4'bx110: s = b << 16;           //x110 LUI: imm << 16bit

             4'b0011: s = b << a;        //0011 SLL: rd <- (rt << sa)
             4'b0111: s = b >> a;        //0111 SRL: rd <- (rt >> sa)
(logical)
             4'b1111: s = $signed(b) >>> a;   //1111 SRA: rd <- (rt >> sa)
(arithmetic)
             default: s = 0;
         endcase
         if (s == 0 )  z = 1;
```

```
            else z = 0;
        end
endmodule
```

输入操作数a，操作数b以及运算控制信号aluc，分析运算控制信号并对a和b进行对应运算输出结果（s）和条件码（z）

MIPS计算器程序设计

指令设计

```
WIDTH=32;
DEPTH=128;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
 0 : 3c070000;  % (000) start:  lui $7, 0   # reg7 stores op  add=0 sub=1
xor=2   %
 1 : 08000033;  % (004)   j main    # start main program        %
 2 : 001ef080;  % (008) sevenseg:   sll $30, $30, 2 # calculate sevenseg table
item addr to load    %
 3 : 8fdd0000;  % (00c)   lw $29, 0($30)  # load sevenseg code of arg($30)
from data memory to $29 %
 4 : 03e00008;  % (010)   jr $31    # return            %
 5 : 0000e820;  % (014) split:  add $29, $0, $0 # set reg29 to 0          %
 6 : 23defff6;  % (018) split_loop: addi $30, $30, -10  # reg30 = result - 10
%
 7 : 001ee7c3;  % (01c)   sra $28, $30, 31  # extend sign digit of the result
%
 8 : 17800002;  % (020)   bne $28, $0, split_done # if reg30 is negative, goto
split_done   %
 9 : 23bd0001;  % (024)   addi $29, $29, 1  # reg29++         %
 A : 08000006;  % (028)   j split_loop  # continue loop         %
 B : 23dc000a;  % (02c) split_done: addi $28, $30, 10 # get units digit and
store to $28      %
 C : 03e00008;  % (030)   jr $31    # return            %
 D : 03e0a020;  % (034) display:  add $20, $31, $0  # store return address to
reg20     %
 E : 001dd140;  % (038)   sll $26, $29, 5 # $26 = 32 * $29          %
 F : 235aff00;  % (03c)   addi $26, $26, 0xff00 # store sevenseg base addr to
reg26     %
```

```
10 : 0c000005;  % (040)   jal split   # call split          %
11 : 03a0f020;  % (044)   add $30, $29, $0  # move $29(returned tens digit) to
$30      %
12 : 0c000002;  % (048)   jal sevenseg  # call split          %
13 : af5d0010;  % (04c)   sw $29, 0x10($26) # show sevenseg tens digit
%
14 : 0380f020;  % (050)   add $30, $28, $0  # move $28(returned units digit)
to $30     %
15 : 0c000002;  % (054)   jal sevenseg  # call split          %
16 : af5d0000;  % (058)   sw $29, 0x0($26)  # show sevenseg units digit
%
17 : 0280f820;  % (05c)   add $31, $20, $0  # restore return address        %
18 : 03e00008;  % (060)   jr $31    # return            %
19 : 8c05ff70;  % (064) get_op: lw $5, 0xff70($0) # load keys to reg5
%
1A : 2006ffff;  % (068)   addi $6, $0, -1 # store 0xffffffff to reg6        %
1B : 00a62826;  % (06c)   xor $5, $5, $6  # reg5 = ~reg5          %
1C : 30a60004;  % (070)   andi $6, $5, 0x4  # store key3 to reg6        %
1D : 14c00005;  % (074)   bne $6, $0, add_op  # key3 is pressed, change op to
add     %
1E : 30a60002;  % (078)   andi $6, $5, 0x2  # store key2 to reg6        %
1F : 14c00004;  % (07c)   bne $6, $0, sub_op  # key2 is pressed, change op to
sub     %
20 : 30a60001;  % (080)   andi $6, $5, 0x1  # store key1 to reg6        %
21 : 14c00003;  % (084)   bne $6, $0, xor_op  # key1 is pressed, change op to
xor     %
22 : 03e00008;  % (088)   jr $31    # no key pressed, return        %
23 : 20c6fffd;  % (08c) add_op: addi $6, $6, -3 # calculate new opcode(4-3-
2+1=0)     %
24 : 20c6fffe;  % (090) sub_op: addi $6, $6, -2 # calculate new opcode(2-
2+1=1)     %
25 : 20c70001;  % (094) xor_op: addi $7, $6, 1  # calculate new opcode and
store to $7(1+1=2)   %
26 : 03e00008;  % (098)   jr $31    # return            %
27 : 14e00002;  % (09c) calc: bne $7, $0, not_add # check if op is add
%
28 : 00432020;  % (0a0)   add $4, $2, $3  # add a and b, store result to reg4
%
29 : 03e00008;  % (0a4)   jr $31    # return            %
2A : 20e8ffff;  % (0a8) not_add:  addi $8, $7, -1 # op--, for checking if op
is sub     %
2B : 15000005;  % (0ac)   bne $8, $0, not_sub # check if op is sub          %
2C : 00432022;  % (0b0)   sub $4, $2, $3  # sub a and b, store result to reg4
%
2D : 00042fc3;  % (0b4)   sra $5, $4, 31  # extend sign digit of the result
%
2E : 10a00001;  % (0b8)   beq $5, $0, sub_done  # result is positive, done
%
2F : 00042022;  % (0bc)   sub $4, $0, $4  # get abs of result         %
```

```
30 : 03e00008;  % (0c0) sub_done: jr $31    # return            %
31 : 00432026;  % (0c4) not_sub:  xor $4, $2, $3  # xor a and b, store result
to reg4     %
32 : 03e00008;  % (0c8)   jr $31    # return            %
33 : 8c01ff80;  % (0cc) main: lw $1, 0xff80($0) # load switches to reg1
%
34 : ac01ff60;  % (0d0)   sw $1, 0xff60($0) # display reg1 to leds        %
35 : 302203e0;  % (0d4)   andi $2, $1, 0x3e0  # get high bits for value a and
store to reg2   %
36 : 00011142;  % (0d8)   srl $2, $1, 5 # calculate value a in reg2       %
37 : 3023001f;  % (0dc)   andi $3, $1, 0x1f # get low bits as value b and
store to reg3   %
38 : 0c000019;  % (0e0)   jal get_op    # call get_op         %
39 : 0c000027;  % (0e4)   jal calc    # call calc           %
3A : 0080f020;  % (0e8)   add $30, $4, $0 # move result to reg30        %
3B : 201d0000;  % (0ec)   addi $29, $0, 0 # set pos to 0          %
3C : 0c00000d;  % (0f0)   jal display # call display          %
3D : 0040f020;  % (0f4)   add $30, $2, $0 # store value a to reg30        %
3E : 201d0002;  % (0f8)   addi $29, $0, 2 # set pos to 2          %
3F : 0c00000d;  % (0fc)   jal display # call display          %
40 : 0060f020;  % (100)   add $30, $3, $0 # move value b to reg30       %
41 : 201d0001;  % (104)   addi $29, $0, 1 # set pos to 1          %
42 : 0c00000d;  % (108)   jal display # call display          %
43 : 08000033;  % (10c)   j main    # continue main           %
END ;
```

内存与I/O设计

```
WIDTH=32;
DEPTH=32;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
  00  :   00000040;
  01  :   00000079;
  02  :   00000024;
  03  :   00000030;
  04  :   00000019;
  05  :   00000012;
  06  :   00000002;
  07  :   00000078;
  08  :   00000000;
  09  :   00000010;
END;
```

功能

主函数中循环调用get_op、calc、display，达到持续读取按键和开关的输入、进行and/or/xor运算并在七段数码管上输出运算结果的效果。其中按键指定运算符，开关高5位和低5位作为两个运算数。发光二极管显示开关输入，左侧一对七段数码管显示运算数a的十进制，中间一对七段数码管显示运算数b的十进制，右边一对数码管显示运算结果的十进制。

- main: 从0xff80端口读入开关状态，并输入到0xff60端口，达到led灯显示对应按键状态的效果；将开关高五位的状态作为操作数a，存于2号寄存器中；将开关低五位的状态作为操作数b，存于3号寄存器中
- get_op：从按键的I/O端口0xff70中读入按键，并逐位判断按键是否按下，分别跳转到add_op/sub_op/xor_op，并在7号寄存器中记录此运算符
  - key[3]: add_op $7=0
  - key[2]: sub_op $7=1
  - key[1]: xor_op $7=2
- calc: 判断$7中的运算符并跳转到对应指令段进行运算，将结果存于4号寄存器中
- display：split将输入的数字分为十位和个位，结合main函数中存入29号寄存器的数据所指定显示的位置，调用两次sevenseg显示到对应位置的七段数码管中

仿真验证

TestBench

```
`timescale 1ps/1ps               // 仿真时间单位/时间精度

//
//  (1) 仿真时间单位/时间精度：数字必须为1、10、100
//  (2) 仿真时间单位：模块仿真时间和延时的基准单位
//  (3) 仿真时间精度：模块仿真时间和延时的精确程度，必须小于或等于仿真单位时间
//
//       时间单位：s/秒、ms/毫秒、us/微秒、ns/纳秒、ps/皮秒、fs/飞秒（10负15次方）。


module sc_computer_sim;

    reg            resetn_sim;
    reg            clock_50M_sim;
   reg            mem_clk_sim;
   reg    [9:0] sw_sim;
```

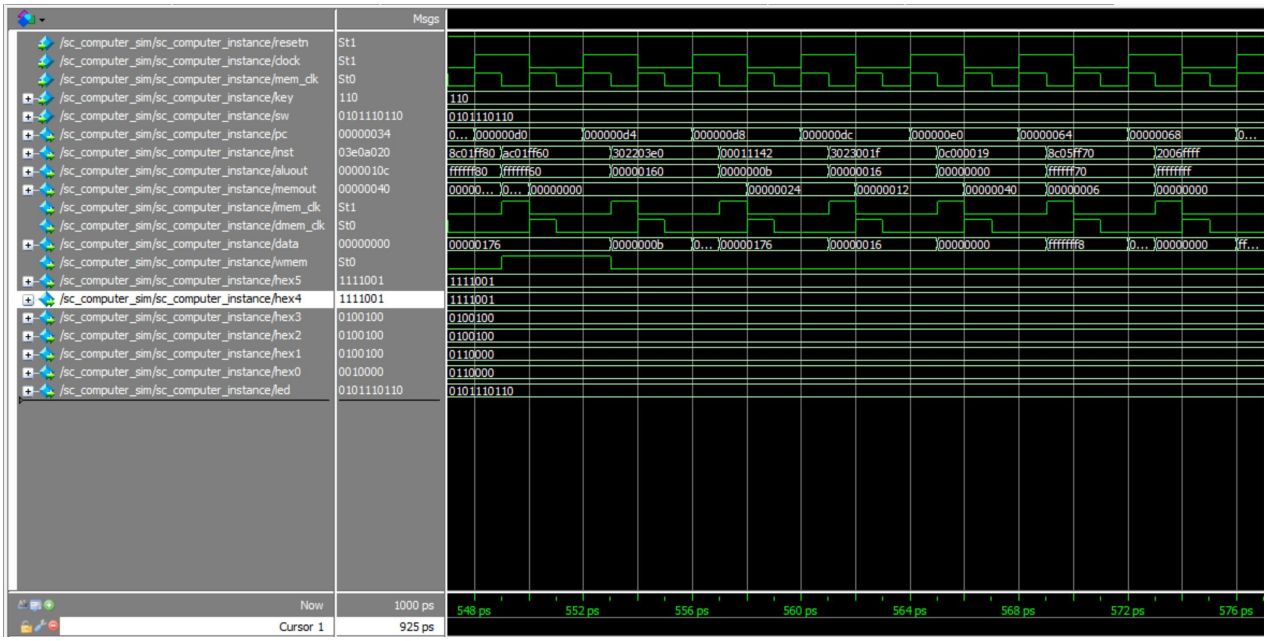
```
    reg    [3:1] key_sim;

     wire   [6:0]  hex0_sim,hex1_sim,hex2_sim,hex3_sim,hex4_sim,hex5_sim;
    wire   [9:0] led_sim;

    wire   [31:0]  pc_sim,inst_sim,aluout_sim,memout_sim;
     wire           imem_clk_sim,dmem_clk_sim;
     wire   [31:0]  data_sim;
     wire           wmem_sim;   // connect the cpu and dmem.

    wire        clk_t_sim, clk_2t_sim;


    initial
     begin
       key_sim <= 3'b111;
       sw_sim <= 10'b0101110110;
     end

     sc_computer_main    sc_computer_instance
(resetn_sim,clock_50M_sim,mem_clk_sim,pc_sim,inst_sim,
                            aluout_sim,memout_sim,imem_clk_sim,dmem_clk_sim,

key_sim,data_sim,wmem_sim,sw_sim,hex5_sim,hex4_sim,
                            hex3_sim,hex2_sim,hex1_sim,hex0_sim,led_sim);
  clock_cpu         clock_cpu_instance(clock_50M_sim, resetn_sim, clk_t_sim,
clk_2t_sim);

    initial
         begin
             clock_50M_sim = 1;
             while (1)
                 #2  clock_50M_sim = ~clock_50M_sim;
         end


    initial
         begin
             mem_clk_sim = 1;
             while (1)
                 #1  mem_clk_sim = ~ mem_clk_sim;
         end


    initial
         begin
             resetn_sim = 0;             // 低电平持续10个时间单位，后一直为1。
             while (1)
                 #5 resetn_sim = 1;
```

```
        end


    // change all switch state every 1000 time unit
    initial
        begin
         while (1)
           #1500 sw_sim = ~sw_sim;
       end


    // press key in turn every 500 time unit
    initial
     begin
       while (1) begin
         #500 key_sim <= 3'b110;
         #500 key_sim <= 3'b101;
         #500 key_sim <= 3'b011;
       end
     end


     initial
         begin
           $display($time,"resetn=%b clock_50M=%b  mem_clk =%b", resetn_sim,
clock_50M_sim, mem_clk_sim);
         end

endmodule
```

生成信号

- 周期为2个时间单位的clock_50M_sim信号作为CPU的clock信号
- 周期为1个时间单位的mem_clk_sim信号作为CPU的mem_clk信号
- resetn信号输入CPU，持续5个时间单位的低电平，然后改变为高电平并保持
- sw信号在0101110110和1010001001之间做周期为1500个时间单位的变换，作为单周期CPU的开关输入
- key信号在110、101、011之间做周期为1500个时间单位的变换，作为单周期CPU的按键输入

验证结果



在ModelSim中进行仿真验证，观察可得一个CPU的clk中mem_clk、imem_clk、dmem_clk依次出现上升沿，能够在一个时钟周期内完成取指令定制、取指令、读内存并完成其他组合逻辑运算，达到了预期的效果。对发光二极管和七段数码管的波形数据分析可得，运行于单周期CPU上的MIPS计算器程序能够正确的输出。

实验总结

实验结果

标准测试程序代码在ModelSim中与其他实验代码经过编译，能正常完成仿真验证，能正确输出预期的波形。实验代码经过编译，载入到开发板后，能正常完成预期的单周期CPU功能。运行在单周期CPU上的MIPS计算器能正常完成预期的功能。

经验教训

1. 在实验所提供的参考代码中，lpm_rom_irom的地址空间定为8位（addr[7:0]），即可以存储256(0x100)字节的指令。在MIPS计算器程序设计过程中，我所使用的汇编指令超过了256字节，导致PC超过0x100时会重新开始取0x00的指令。经过调试后我发现了问题在于rom的地址空间不够，无法取到0x100以上的指令。将lpm_rom_irom的地址空间增加到9位(addr[8:0])后，程序在0x100以上的地址空间也能够正常取指，达到预期运行效果。
2. Quartus即使编译成功也要查看编译过程中所汇报的警告。通过对警告的查看，我发现了非阻塞赋值的语句错误的写成了阻塞赋值等细节错误。Quartus编译期间发出的警告会对代码语义上的纠错很有帮助。
3. TestBench的信号设置要与其中所调用的模块的接口一一对应。在仿真验证过程中，由于我将某个接口名错写，导致仿真验证中有一个波形无法正常输出。经过调试后解决了信号名书写错误的问题，达到了预期的波形输出效果。
4. 载入到开发版之前对各个模块进行仿真验证，输出用于调试的波形，会对发现代码逻辑中的问题并

解决问题很有帮助。

感受

作为一名软件工程专业大三的学生，在本次试验中我亲身体会到设计单周期CPU、指令集，以及将汇编语言所写的程序运行在CPU上的过程。对单周期CPU的结构，各个部件的信号输入和输出，CPU各个模块的功能和设计，作为硬件与软件的接口的指令集的作用有了更加清晰的了解。同时在实验过程中查阅用户手册、verilog语法、使用仿真验证调试硬件需要很多耐心，在此过程中我定位问题、解决问题的能力得到了锻炼和提升。必要时和同学、老师的交流也带来了很多宝贵的经验和有效的思路。