# 实验报告：基本单周期 CPU 设计

姓名：江玓璠

学号：518021910550

## 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。

## 实验内容和任务

1. 采用 Verilog HDL 在 Quartus Ⅱ 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. 在实现 MIPS 基本 20 条指令的基础上，实现 Y86 相应的基本指令。
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，提供以上两种指令集（MIPS 和 Y86）的应用功能的程序设计代码，并提供程序主要流程图。

## 实验仪器

- 硬件：DE1-SoC 实验板
- 软件：Altera Quartus II 13.1、 Altera ModelSim 10.1d

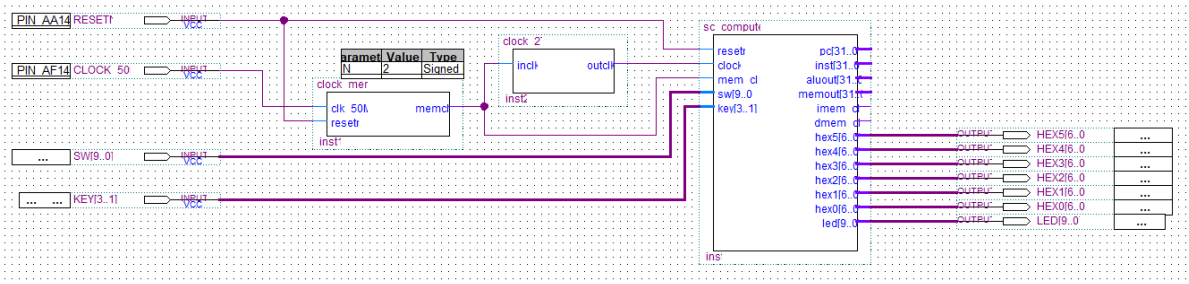
## 实验详情

### 一、设计思路

本实验的大部分代码已经给出，按照实验指导书的要求补全代码完成了单周期 CPU 的设计和仿真，主要完成的部分是 ALU 模块 CU 模块和 Datamem 模块，各自详细的内容将在后文给出。

采用的 MIPS 汇编代码是一段简单的计算器程序，读取两个输入的加数以及运算符，求出对应的和并将结果和加数都显示在 LED 数码管上，具体的代码也会在后文给出。

## 二、顶层设计




输入为板载 50 MHz 的时钟、KEY 0 作为复位按钮，KEY 1-3 作为计算器运算符的输入，SW 0-9 作为计算器两个运算数的输入；

clock_mem 模块将板载时钟转换为主模块的 memclk 时钟，clcok_2T 模块将 memclk 转换成周期翻倍的时钟作为主模块的 clock，两个 clock 共同作为主模块的时钟；

主模块 sc_computer 进行指令的读取、解码、运算、访存等操作；

输出为 6 个七段 LED 数码管和 10 个 LED 发光二极管，其他的输出接口为代码自带的，可以用于仿真的时候进行调试。

## 三、部分源代码设计

1. `clock_mem` 模块

```
// 将板载 50MHz 的时钟转换成需要的 memclk 时钟
module clock_mem(clk_50M, resetn, memclk);
    ...
    parameter N = 2;
    ...
    always @(posedge clk_50M or negedge resetn) begin
        // 当 resetn 被按下，重置时钟
        if (!resetn) begin
            counter <= 0;
            memclk <= 0;
        end
        else begin
            if (counter >= N / 2 - 1) begin
                counter <= 0;
                memclk <= ~memclk;
            end
            else
                counter <= counter + 1;
        end
    end
endmodule
```

因为实验对 CPU 的时钟周期没有硬性的要求，因此这里直接采用板载时钟的 2 倍周期作为 CPU 的 `mem_clk` 的周期。

2. `clock_2T` 模块

```
// 把输入的 mem_clk 转换为两倍周期的 cpu clock
module clock_2T(inclk, outclk);
    ...
    always @(posedge inclk)
        outclk <= ~outclk;
    ...
endmodule
```

这个模块用于将 `mem_clk` 的周期翻倍后作为 CPU 的 `clock` 时钟，两路时钟信号通过组合逻辑生成 4 个节拍信号，提供 `imem_clk`、`dmem_clk` 两个节拍作为 ROM 和 RAM 的同步时钟，这样 CPU 的工作就能在一个时钟周期之内顺序进行。

3. `sc_computer` 主模块

```
module sc_computer
(resetn,clock,mem_clk,pc,inst,aluout,memout,imem_clk,dmem_clk,
    sw,key,hex5,hex4,hex3,hex2,hex1,hex0,led);
   input resetn,clock,mem_clk;
   input [9:0] sw;
   input [3:1] key;
   output [31:0] pc,inst,aluout,memout;
   output        imem_clk,dmem_clk;
   output [6:0]  hex5,hex4,hex3,hex2,hex1,hex0;
   output [9:0]  led;
   wire   [31:0] data;
   wire          wmem; // all these "wire"s are used to connect or interface
the cpu,dmem,imem and so on.

   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
```

主模块没有逻辑，仅仅是定义了输入输出，以及实例化三个子模块：`CPU`、指令内存 ROM `imem`，数据内存 RAM `dmem`；

4. `sc_cpu` 模块

```
module sc_cpu (clock,resetn,inst,mem,pc,wmem,alu,data);
    ...
    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]   offset = {imm[13:0],inst[15:0],1'b0,1'b0};
//offset(include sign extend)
```

```
    wire [31:0]   immediate = {imm,inst[15:0]}; // sign extend to high 16

    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
```

这部分代码实验指导书已经给出，主要是声明了 CPU 中的一些信号量如 `p4`，`imm`，`adr` 等，以及实例化了一些子模块如控制模块 `cu`，运算模块 `alu`，寄存器堆 `regfile`，多路选择器 `mux2x32` 等，主要的逻辑是和 CPU 的设计思路相同的，这里就不再复述实验指导书的内容了。其他大部分子组件的代码都是给出的，且逻辑也不复杂，后文也不再展开讲述了。

5. `sc_cu` 控制模块

```
module sc_cu (op, func, z, wmem, wreg, regrt, m2reg, aluc, shift,
              aluimm, pcsource, jal, sext);
   ...
   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

     // PC 值来源
    assign pcsource[1] = i_jr | i_j | i_jal;
    assign pcsource[0] = ( i_beq & z ) | (i_bne & ~z) | i_j | i_jal ;
    // 写寄存器信号
    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;
    // 运算器控制信号
    assign aluc[3] = i_sra;
    assign aluc[2] = i_sub | i_or | i_srl | i_sra | i_ori | 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;
     // 位移信号
    assign shift   = i_sll | i_srl | i_sra ;
     // 操作数 b 来源
    assign aluimm  = i_addi | i_andi | i_ori | i_xori | i_lw | i_sw | i_lui;
     // 符号扩展/0扩展
    assign sext    = i_addi | i_lw | i_sw | i_beq | i_bne;
     // 写内存
    assign wmem    = i_sw;
     // 读内存到寄存器
    assign m2reg   = i_lw;
     // 写寄存器 rd/rt
    assign regrt   = i_addi | i_andi | i_ori | i_xori | i_lw | i_lui;
     // 是否跳转
    assign jal     = i_jal;
endmodule
```

`sc_cu` 模块根据指令的类型，读取每个指令的 `op`、`func` 以及 `ALU` 计算模块传回的 `z` 信号，产生对应的信号量输出到各个其他模块，每个指令对应的信号量的数值已在 Excel 表格中给出。

6. `sc_alu` 计算模块

```
module alu (a,b,aluc,s,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
            // 设置条件码 z
         if (s == 0 )  z = 1;
            else z = 0;
      end
endmodule
```

计算模块根据操作数 `a` 、 `b` ，操作符 `aluc` 计算产生结果 `s` 并将条件码 `z` 置为合适的值。每个运算符对应的二进制码也在实验指导书中有给出。

7. `sc_instmem` 指令内存模块

```
module sc_instmem (addr,inst,clock,mem_clk,imem_clk);
   ...
   wire          imem_clk;
   assign  imem_clk = clock & ( ~ mem_clk );
   rom_1port irom(addr[8:2],imem_clk,inst);
endmodule
```

指令内存模块根据两路时钟生成 `imem_clk` 作为模块的时钟，在上升沿的时候读取一条指令进行处理，`irom` 为实例化的 Altera 库自带的 `ROM_1PORT` 的 MegaFunction，可以读取对于内存地址的指令。

8. `sc_datamem` 数据内存模块

```
module sc_datamem (resetn,addr,datain,dataout,we,clock,mem_clk,dmem_clk,
    sw,key,hex5,hex4,hex3,hex2,hex1,hex0,led);
    ...
    wire        dmem_clk;
    wire        write_enable;
    wire [31:0] memdata;
    assign write_enable = we & ~clock & (addr[31:8] != 24'hffffff);
    assign dmem_clk = mem_clk & ( ~ clock) ;
    ram_1port  dram(addr[6:2],dmem_clk,datain,write_enable,memdata);

    // I/O
    always @(posedge dmem_clk or negedge resetn) begin
        if (!resetn) begin // 重置所有 I/O 的值
            hex0 <= 7'b1111111;
```

```
            hex1 <= 7'b1111111;
            hex2 <= 7'b1111111;
            hex3 <= 7'b1111111;
            hex4 <= 7'b1111111;
            hex5 <= 7'b1111111;
            led <= 10'b0000000000;
        end else if (we) begin // 当 we 为 1 时触发写内存的操作，采用高地址映射 I/O
，此处为输出
            case (addr)
                32'hffffff20: hex0 <= datain[6:0];
                32'hffffff30: hex1 <= datain[6:0];
                32'hffffff40: hex2 <= datain[6:0];
                32'hffffff50: hex3 <= datain[6:0];
                32'hffffff60: hex4 <= datain[6:0];
                32'hffffff70: hex5 <= datain[6:0];
                32'hffffff80: led <= datain[9:0];
            endcase
        end
    end

    always @(posedge dmem_clk) begin // 读取内存的操作，采用高地址映射 I/O ，此处
为输入
        case (addr)
            32'hffffff00: dataout <= {22'b0, sw};
            32'hffffff10: dataout <= {28'b0, key, 1'b1};
            default: dataout <= memdata;
        endcase
    end
endmodule
```

数据内存模块进行内存的读写操作，通过两路时钟产生的 `dmem_clk` 作为模块的时钟，在上升沿的时候读写数据。根据指令的类型和操作的目标地址判断是读还是写。`dram` 是实例化的 Altera 库的 `RAM_1PORT` 的 MegaFunction，可以读写对应地址的数据。为了将 I/O 的部分加入，采用的高端地址映射的方式，当读取指定高端地址时，从 `SW` 或者 `RESETN` 的输入读取数据；当写指定的高端地址时，转化为对 6 个七段 LED 数码管（表示计算器的操作数和结果）和 LED 发光二极管（表示开关的状态）的输出，具体的映射关系可以参考源代码。

9. `calculator.mif` 计算器程序

   **由于提供的汇编器有一定的 bug，本程序代码参考自网络上的资料**

```
 0 : 3c070000;        % (000) start:      lui $7, 0               # $7
stores op (0->add (default), 1->sub, 2->xor)        %
 1 : 08000033;        % (004)             j main_loop             # enter
main loop                                       %
 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 $ra                  # return
                                            %
 5 : 0000e820;        % (014) split:      add $29, $0, $0         # $29
stores tens digit                                %
 6 : 23defff6;        % (018) split_loop: addi $30, $30, -10      #
decrement arg($30) by 10                                %
 7 : 001ee7c3;        % (01c)             sra $28, $30, 31        # extend
sign digit of the result                        %
```

```
 8 : 17800002;        % (020)             bne $28, $0, split_done # if $30
has become negative, goto split_done           %
 9 : 23bd0001;        % (024)             addi $29, $29, 1        #
increment tens digit                                 %
 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 $ra                  # return
                                          %
 D : 03e0a020;        % (034) show:       add $20, $31, $0        # store
return address to $20                              %
 E : 001dd140;        % (038)             sll $26, $29, 5         # $26 = 32
* $29(arg2, pos)                           %
 F : 235aff20;        % (03c)             addi $26, $26, 0xff20   #
calculate sevenseg pair base addr and store to $26     %
10 : 0c000005;        % (040)             jal split               # call
split (passing $30, arg1, value)                    %
11 : 03a0f020;        % (044)             add $30, $29, $0        # move
$29(returned tens digit) to $30                     %
12 : 0c000002;        % (048)             jal sevenseg            # call
split (passing $30)                                 %
13 : af5d0010;        % (04c)             sw $29, 16($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 (passing $30)                                 %
16 : af5d0000;        % (058)             sw $29, 0($26)          # show
sevenseg units digit                                %
17 : 0280f820;        % (05c)             add $31, $20, $0        # restore
return address                                %
18 : 03e00008;        % (060)             jr $ra                  # return
                                          %
19 : 8c05ff10;        % (064) get_op:     lw $5, 65296($0)        # load
state of keys to $5                                 %
1A : 2006ffff;        % (068)             addi $6, $0, -1         # store
32'bffffffff to $6                                 %
1B : 00a62826;        % (06c)             xor $5, $5, $6          # $5 = ~$5
                                          %
1C : 30a60008;        % (070)             andi $6, $5, 0x8        # get
state of key3                                       %
1D : 14c00005;        % (074)             bne $6, $0, add_op      # if key3
is pressed, change op to add                  %
1E : 30a60004;        % (078)             andi $6, $5, 0x4        # get
state of key2                                       %
1F : 14c00004;        % (07c)             bne $6, $0, sub_op      # if key2
is pressed, change op to sub                  %
20 : 30a60002;        % (080)             andi $6, $5, 0x2        # get
state of key1                                       %
21 : 14c00003;        % (084)             bne $6, $0, xor_op      # if key1
is pressed, change op to xor                  %
22 : 03e00008;        % (088)             jr $ra                  # no key
pressed, no op change, return                 %
23 : 20c6fffb;        % (08c) add_op:     addi $6, $6, -5         #
calculate new opcode                                %
24 : 20c6fffd;        % (090) sub_op:     addi $6, $6, -3         #
calculate new opcode                                %
```

```
25 : 00c03820;        % (094) xor_op:     add $7, $6, $0          #
calculate new opcode and store to $7                  %
26 : 03e00008;        % (098)             jr $ra                  # return
                                                      %
27 : 14e00002;        % (09c) do_op:      bne $7, $0, not_add     # check if
op is add                                             %
28 : 00432020;        % (0a0)             add $4, $2, $3          # do add
                                                      %
29 : 03e00008;        % (0a4)             jr $ra                  # return
                                                      %
2A : 20e8ffff;        % (0a8) not_add:    addi $8, $7, -1         # check if
op is sub                                             %
2B : 15000005;        % (0ac)             bne $8, $0, not_sub     # check if
op is sub                                             %
2C : 00432022;        % (0b0)             sub $4, $2, $3          # do sub
                                                      %
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          # result =
-result (get abs of result)                           %
30 : 03e00008;        % (0c0) sub_done:   jr $ra                  # return
                                                      %
31 : 00432026;        % (0c4) not_sub:    xor $4, $2, $3          # do xor
                                                      %
32 : 03e00008;        % (0c8)             jr $ra                  # return
                                                      %
33 : 8c01ff00;        % (0cc) main_loop:  lw $1, 65280($0)        # load
state of switches to $1                                 %
34 : ac01ff80;        % (0d0)             sw $1, 65408($0)        # store $1
to state of leds                                      %
35 : 302203e0;        % (0d4)             andi $2, $1, 0x3e0      #
calculate value1 and store to $2                        %
36 : 00011142;        % (0d8)             srl $2, $1, 5           #
calculate value1 and store to $2                        %
37 : 3023001f;        % (0dc)             andi $3, $1, 0x1f       #
calculate value2 and store to $3                        %
38 : 0c000019;        % (0e0)             jal get_op              # call
get_op                                                  %
39 : 0c000027;        % (0e4)             jal do_op               # call
do_op (passing $2 and $3)                               %
3A : 0080f020;        % (0e8)             add $30, $4, $0         # move
$4(result) to $30                                       %
3B : 201d0000;        % (0ec)             addi $29, $0, 0         # set pos
to 0 (right pair)                                     %
3C : 0c00000d;        % (0f0)             jal show                # call
show (passing $30 and $29)                              %
3D : 0040f020;        % (0f4)             add $30, $2, $0         # move
$2(value1) to $30                                       %
3E : 201d0002;        % (0f8)             addi $29, $0, 2         # set pos
to 2 (left pair)                                      %
3F : 0c00000d;        % (0fc)             jal show                # call
show (passing $30 and $29)                              %
40 : 0060f020;        % (100)             add $30, $3, $0         # move
$3(value2) to $30                                       %
41 : 201d0001;        % (104)             addi $29, $0, 1         # set pos
to 1 (middle pair)                                    %
```

```
42 : 0c00000d;        % (108)            jal show                # call
show (passing $30 and $29)                      %
43 : 08000033;        % (10c)            j main_loop             # loop
forever                                         %
```

`start` 是程序的入口函数，主要是将运算符置为默认的值，表示加法；

`main_loop` 是主循环函数，每一个循环读取开关和按钮的输入，进行一次计算器的计算，并进行值的显示；十个开关分为两组（0-4，5-9），以二进制的形式表示两个操作数，例如：如果开关（从左到右）的状态是 0011000011 则表示两个操作数分别为 6 和 3；

`get_op` 函数根据按钮的状态获取计算器的运算符，key 3 按下对应加法，key 2 按下对应减法，key 1 按下对应异或操作；

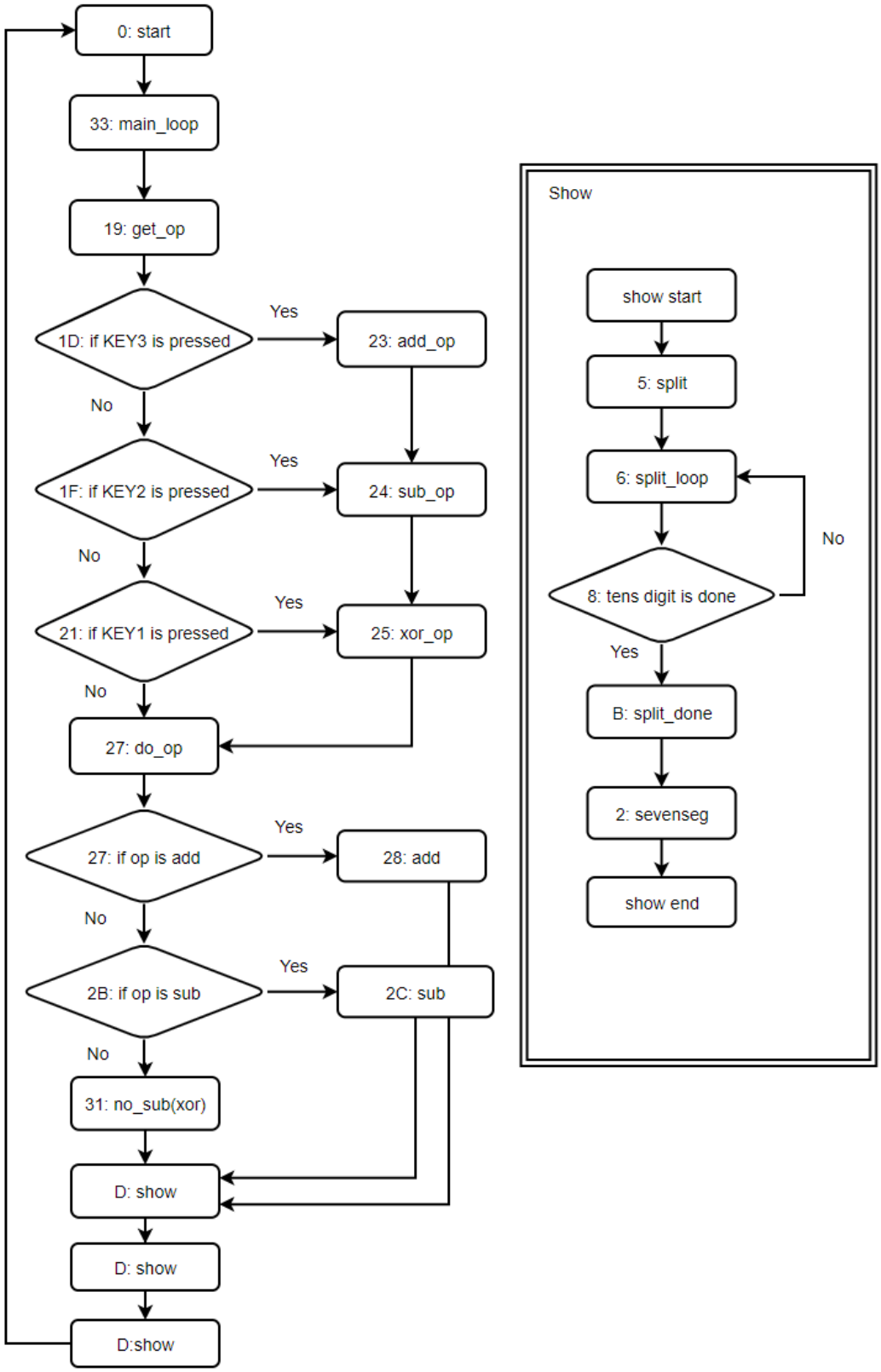
`do_op` `not_add` `not_sub` `add_op` `sub_op` `xor_op` `not_sub` 函数共同起作用，进行相应的计算并得出结果；

`show` 函数根据实现存储在内存的 7 段 LED 数码管编码，将操作数和结果的数值根据不同的位置显示在 LED 数码管上，在调用 `show` 函数的时候利用了 `$29` 寄存器储存数码管的位置信息，0 对应右边两个数码管，显示运算的结果，1 对应中间的两个数码管，显示一个操作数（减数），2 对应左边的两个数码管，显示另一个操作数（被减数），如果减法运算的结果为负值，将只显示其绝对值；

`split` `split_loop` `split_done` 三个函数共同完成数字的十位和个位的分割；

`sevenseg` 根据每一位的值转换成对应的数字需要的 7 段 LED 数码管编码；

具体的流程图如下：




这个文件用作初始化 `rom_1port` 模块的文件，当项目导入板子后会自动取指令运行在设计的 CPU 上；

10. `sevenseg.mif` 内存布局

```
0: 00000040;                % sevenseg code for '0' %
1: 00000079;                % sevenseg code for '1' %
2: 00000024;                % sevenseg code for '2' %
3: 00000030;                % sevenseg code for '3' %
4: 00000019;                % sevenseg code for '4' %
5: 00000012;                % sevenseg code for '5' %
6: 00000002;                % sevenseg code for '6' %
7: 00000078;                % sevenseg code for '7' %
8: 00000000;                % sevenseg code for '8' %
9: 00000010;                % sevenseg code for '9' %
```

用于预存 7 段 LED 数码管对应数字的编码，用作初始化 `ram_1port` 模块的文件，用作初始化的内存 content；

11. `sc_computer_sim` 仿真 Testbench 文件

```
module sc_computer_sim;
    ...
    sc_computer    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,sw_sim,key_sim,hex5_sim,hex4_sim,

 hex3_sim,hex2_sim,hex1_sim,hex0_sim,led_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;
                while (1)
                #5 resetn_sim = 1;
        end

    initial
        begin
            sw_sim = 10'b0001100110;
            key_sim = 4'b0;
        end

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

```
            # 12500 $display($time," out_port0 = %b  out_port1 = %b
out_port2 = %b  out_port3 = %b  out_port4 = %b  out_port5 = %b ",
hex5_sim,hex4_sim,hex3_sim,hex2_sim,hex1_sim,hex0_sim );
        end
endmodule
```

Testbench 实例化了一个 `sc_computer` 模块，模拟对其的输入信号并打印输出的结果，这里将 `resetn_sim` 置为 1，`sw_sim` 置为 0001100110，`key_sim` 置为 0，表示计算 3 + 6 的值，最后打印 7 段数码管的输出。利用 ModelSim 进行仿真测试可以观察不同信号量的波形图，进而判断 CPU 的设计和汇编代码是否正确。

# 实验总结

## 实验结果

设计的单周期 CPU 可以正常通过编译，在 ModelSim 中运行 Testbench 可以产生正确的波形图，且输出符合预期。将代码载入开发板后可以正常运行编译后的计算器程序，7 段 LED 数码管和 LED 发光二极管可以正常显示，10 个开关和 4 个按钮都可以正常输入，计算器运行逻辑符合预期。

## 经验总结与反思

1. 本实验中大部分的代码已经给出，主要完成的部分包括 `ALU` 模块，`CU` 模块，I/O 映射，计算器程序和仿真测试；
2. `CU` 模块根据指令的类型，产生多个信号量，涉及到的逻辑不复杂但是内容较多，需要按照实验指导书一一实现，一个 bit 的值出错就可能导致整个程序无法得到预期的结果；
3. 实验给出的汇编器本身有一些 bug 不能正确的生成所需的 `.mif` 文件，因为实际上运行的程序对 CPU 本身的设计是无关的，为了简便起见，参考了网络上的 MIPS 汇编代码的计算器程序；
4. ModelSim 的仿真需要重新编译一遍文件，仅仅是通过了 Quartus 的编译仍然可能报错，在仿真时因为 CPU 的时钟周期很小，测试程序其实很快就跑完了，而波形图的显示可能由于重叠等原因无法看清，需要放大到合适的倍数才能正常观察程序的波形图；
5. 提供的 RAM 和 ROM 模块对版本有要求，因此在助教的帮助下换成了新的 MegaFunction，实际上对原来的代码进行一定的修改也是可以实现的，但由于对 Altera 库和 Quartus 的操作不熟悉就没有采用原有的代码；
6. 实验中采用的汇编代码比较复杂，涉及了读取输入、计算逻辑、位数的分割、7 段数码管编码的转换、显示信号的输出等等，调试起来也更加复杂，其实也可以将其中一些部分提取出来，单独做成项目的 module 模块，这样可以简化汇编代码的逻辑，也更方便对代码的调试，但会一定程度上增加不同模块的耦合度。