基本单周期 CPU 设计 - 实验报告

- 学号：
- 姓名：

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。

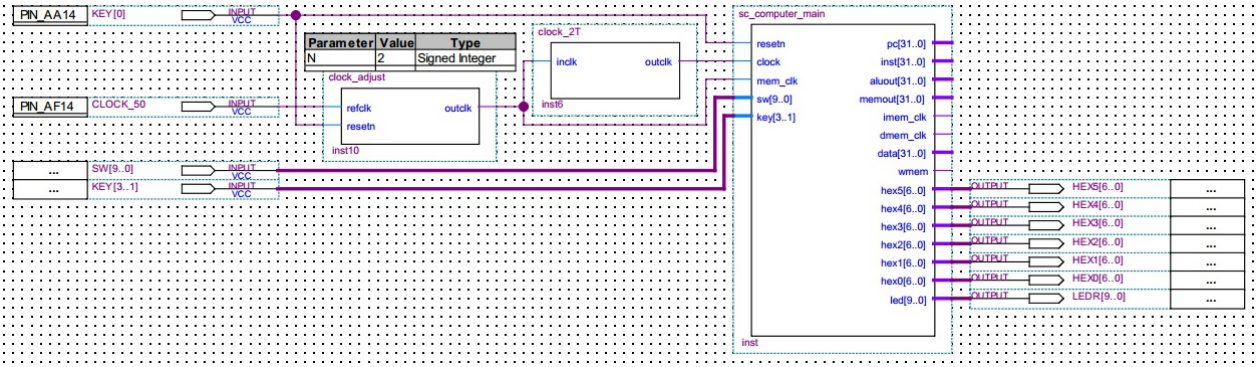
实验仪器与平台

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

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 DE1 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 DE1 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 MIPS 指令集的应用功能的程序设计代码，并提供程序主要流程图。

顶层设计



板载时钟经过 `clock_adjust` 模块调节周期（用于加速编译）之后作为 CPU 的 `mem_clk` 信号。`clock_2T` 模块产生周期二倍于 `mem_clk` 的时钟信号作为 CPU 的 `clock` 信号，以满足本 CPU 的时序控制要求。`KEY[0]` 作为 CPU 的 reset 信号。CPU 的输入端口包括 `SW[9..0]` 以及 `KEY[3..1]`，输出端口包括 `HEX5[6..0]`、`HEX4[6..0]`、`HEX3[6..0]`、`HEX2[6..0]`、`HEX1[6..0]`、`HEX0[6..0]` 以及 `LED[9..0]`。其余输出信号用于仿真验证时查看，故未分配引脚。

```
// Generate a clock whose period is N times the period of the reference clock.
module clock_adjust(refclk, resetn, outclk);
    input      refclk, resetn;
    output reg outclk;
    reg [31:0] counter;
    parameter N = 2;

    initial begin
        counter <= 0;
        outclk <= 0;
    end

    always @(posedge refclk or negedge resetn) begin
        if (!resetn) begin
            counter <= 0;
            outclk <= 0;
        end
        else begin
            if (counter >= N / 2 - 1) begin
                counter <= 0;
                outclk <= ~outclk;
            end
            else
                counter <= counter + 1;
        end
    end
endmodule
```

```
// Generate a clock whose period is twice the period of the reference clock.
module clock_2T(inclk, outclk);
    input      inclk;
    output reg outclk;

    initial begin
        outclk <= 0;
    end

    always @(posedge inclk)
        outclk <= ~outclk;
endmodule
```

主模块

```
module sc_computer_main(resetn, clock, mem_clk, pc, inst, aluout, memout, imem_clk, dmem_clk, data, wmem,
    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 [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 and IO ports.
endmodule
```

该单周期 CPU 计算机由三部分组成：CPU（`sc_cpu`）、指令存储（`sc_instmem`）以及数据存储（`sc_datamem`）。I/O 访问由数据存储模块使用统一编址的 I/O 端口扩展方法实现（见后文）。

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;

    /* inst: the fetched instruction
     * mem: result of a data memory read
     * npc: next pc
     * res: data to write into register
     * adr: branch addr
     * ra: result a from regfile
     * data: result b from regfile, may be written into memory
     * alu: result of ALU, may serve as data address
     * alu_mem: candidate of data to write into register (maybe from ALU or memory)
     * reg_dest: candidate of register number to write
     * wn: register number to write
     */
    wire   [31:0] p4, 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 sa to 32 bits for shift instruction
    wire          e = sext & inst[15]; // the bit to extend
    wire   [15:0] imm = {16{e}}; // high 16 sign bit when sign extend (otherwise 0)
    wire   [31:0] offset = {imm[13:0], inst[15:0], 2'b00}; // branch addr offset (include extend)
    wire   [31:0] immediate = {imm, inst[15:0]}; // extend immediate to high 16
    wire   [31:0] jpc = {p4[31:28], inst[25:0], 2'b00}; // j address

    assign p4 = pc + 32'h4; // pc + 4
    assign adr = p4 + offset; // branch addr
    assign wn = reg_dest | {5{jal}}; // reg_dest or 31 (jal: r31 <-- p4;)

    dff32 ip(npc, clock, resetn, pc);  // define a D-register for PC
    sc_cu cu(inst[31:26], inst[5:0], zero, wmem, wreg, regrt, m2reg, aluc, shift, aluimm, pcsource, jal, sext);
    regfile rf(inst[25:21], inst[20:16], res, wn, wreg, clock, resetn, ra, data);
    mux2x32 alu_a(ra, sa, shift, alua);
    mux2x32 alu_b(data, immediate, aluimm, alub);
    alu al_unit(alua, alub, aluc, alu, zero);
    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);
    mux4x32 nextpc(p4, adr, ra, jpc, pcsource, npc);
endmodule
```

该 CPU 由程序计数器寄存器（`ip`）、控制器（`sc_cu`）、寄存器文件（`regfile`）、算术/逻辑单元（`alu`）以及各部件信号的链路（直接连接、位扩展、多路选择器选通等）组成。相关细节与 PPT 上所提供的设计图相同。

控制器

```
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 == 6'b000000;
    wire i_add = r_type & func == 6'b100000;
    wire i_sub = r_type & func == 6'b100010;
    wire i_and = r_type & func == 6'b100100;
    wire i_or  = r_type & func == 6'b100101;
    wire i_xor = r_type & func == 6'b100110;
    wire i_sll = r_type & func == 6'b000000;
```

```
    wire i_srl = r_type & func == 6'b000010;
    wire i_sra = r_type & func == 6'b000011;
    wire i_jr  = r_type & func == 6'b001000;
    wire i_addi = op == 6'b001000;
    wire i_andi = op == 6'b001100;
    wire i_ori  = op == 6'b001101;
    wire i_xori = op == 6'b001110;
    wire i_lw   = op == 6'b100011;
    wire i_sw   = op == 6'b101011;
    wire i_beq  = op == 6'b000100;
    wire i_bne  = op == 6'b000101;
    wire i_lui  = op == 6'b001111;
    wire i_j    = op == 6'b000010;
    wire i_jal  = op == 6'b000011;

    assign pcsource[1] = i_jr | i_j | i_jal;
    assign pcsource[0] = (i_beq & z) | (i_bne & ~z) | i_j | 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 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 m2reg = i_lw;
    assign regrt = i_addi | i_andi | i_ori | i_xori | i_lw | i_lui;
    assign jal = i_jal;
endmodule
```

控制器在 CPU 中充当“指挥官”的角色。它读取指令的操作码（`op`）和功能码（`func`）以及 ALU 运算产生的条件码（`z`），依此产生一系列控制信号，以控制 CPU 的其他部件正常工作。

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, r0 always contains 0
    assign qb = (rnb == 0) ? 0 : register[rnb]; // read, r0 always contains 0

    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
```

该寄存器文件共有 32 个寄存器，其中 0 号寄存器的值永远为 0（读取到的值永远为 0，写入无效）。对寄存器的读取为异步操作，而对寄存器的写入在下一周期时钟上升沿生效。`resetn` 信号会将所有寄存器的值置为 0。

算术/逻辑单元

```
module alu(a, b, aluc, s, z);
    input      [31:0] a, b;
    input      [3:0]  aluc;
    output reg [31:0] s;
    output reg        z;

    always @ (a or b or aluc) begin
        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
        z = (s == 0) ? 1'b1 : 1'b0;
    end
endmodule
```

算术/逻辑单元按照 `aluc` 控制信号的指示对操作数执行相应的运算，并设置条件码 `z`（运算结果是否为 0）。

指令存储

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

    assign imem_clk = clock & ~mem_clk;

    rom_1port irom(addr[8:2], imem_clk, inst);
endmodule
```

指令存储使用 Altera 提供的 megafunction `ROM:1-PORT` 实现，此处产生了一个 `imem_clk` 作为读指令存储的时钟信号，用于满足 CPU 的时序控制要求。

数据存储及 I/O 访问

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

    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, mem_dataout);

    // IO ports design.
    always @(posedge dmem_clk or negedge resetn) begin
```

```
        if (!resetn) begin // reset hexs and leds
            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 // write when dmem_clk posedge comes
            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 // read when dmem_clk posedge comes
        case (addr)
            32'hffffff00: dataout <= {22'b0, sw};
            32'hffffff10: dataout <= {28'b0, key, 1'b1}; // can only read key[3..1], key0 is used for reset
            default: dataout <= mem_dataout;
        endcase
    end
endmodule
```

此处的实现有误，详见流水线实验报告中的相关说明。

数据存储使用 Altera 提供的 megafunction `RAM:1-PORT` 实现，此处产生了一个 `dmem_clk` 作为读写数据存储的时钟信号，用于满足 CPU 的时序控制要求。

I/O 端口设计：预留一部分高端地址，将其映射为 I/O 端口：

- `32'hffffff00` -> `SW[9..0]`（输入）
- `32'hffffff10` -> `KEY[3..1]`（输入）
- `32'hffffff20` -> `HEX0[6..0]`（输出）
- `32'hffffff30` -> `HEX1[6..0]`（输出）
- `32'hffffff40` -> `HEX2[6..0]`（输出）
- `32'hffffff50` -> `HEX3[6..0]`（输出）
- `32'hffffff60` -> `HEX4[6..0]`（输出）
- `32'hffffff70` -> `HEX5[6..0]`（输出）
- `32'hffffff80` -> `LED[9..0]`（输出）

为使 I/O 数据读写更加有序，采用与数据存储相同的 `dmem_clk` 作为时钟信号进行控制（同步读写）。

使用 CPU 实现计算器

```
start:       lui $7, 0               # $7 stores op (0->add (default), 1->sub, 2->xor)
             j main_loop             # enter main loop
sevenseg:    sll $30, $30, 2         # calculate sevenseg table item addr to load
             lw $29, 0($30)          # load sevenseg code of arg($30) from data memory to $29
             jr $ra                  # return
split:       add $29, $0, $0         # $29 stores tens digit
split_loop:  addi $30, $30, -10      # decrement arg($30) by 10
             sra $28, $30, 31        # extend sign digit of the result
             bne $28, $0, split_done # if $30 has become negative, goto split_done
             addi $29, $29, 1        # increment tens digit
             j split_loop            # continue loop
split_done:  addi $28, $30, 10       # get units digit and store to $28
             jr $ra                  # return
show:        add $20, $31, $0        # store return address to $20
             sll $26, $29, 5         # $26 = 32 * $29(arg2, pos)
             addi $26, $26, 0xff20   # calculate sevenseg pair base addr and store to $26
```

```
              jal split               # call split (passing $30, arg1, value)
              add $30, $29, $0        # move $29(returned tens digit) to $30
              jal sevenseg            # call split (passing $30)
              sw $29, 16($26)         # show sevenseg tens digit
              add $30, $28, $0        # move $28(returned units digit) to $30
              jal sevenseg            # call split (passing $30)
              sw $29, 0($26)          # show sevenseg units digit
              add $31, $20, $0        # restore return address
              jr $ra                  # return
  get_op:     lw $5, 65296($0)        # load state of keys(0xff10) to $5
              addi $6, $0, -1         # store 32'bffffffff to $6
              xor $5, $5, $6          # $5 = ~$5
              andi $6, $5, 0x8        # get state of key3
              bne $6, $0, add_op      # if key3 is pressed, change op to add
              andi $6, $5, 0x4        # get state of key2
              bne $6, $0, sub_op      # if key2 is pressed, change op to sub
              andi $6, $5, 0x2        # get state of key1
              bne $6, $0, xor_op      # if key1 is pressed, change op to xor
              jr $ra                  # no key pressed, no op change, return
  add_op:     addi $6, $6, -5         # calculate new opcode
  sub_op:     addi $6, $6, -3         # calculate new opcode
  xor_op:     add $7, $6, $0          # calculate new opcode and store to $7
              jr $ra                  # return
  do_op:      bne $7, $0, not_add     # check if op is add
              add $4, $2, $3          # do add
              jr $ra                  # return
  not_add:    addi $8, $7, -1         # check if op is sub
              bne $8, $0, not_sub     # check if op is sub
              sub $4, $2, $3          # do sub
              sra $5, $4, 31          # extend sign digit of the result
              beq $5, $0, sub_done    # result is positive, done
              sub $4, $0, $4          # result = -result (get abs of result)
  sub_done:   jr $ra                  # return
  not_sub:    xor $4, $2, $3          # do xor
              jr $ra                  # return
  main_loop:  lw $1, 65280($0)        # load state of switches(0xff00) to $1
              sw $1, 65408($0)        # store $1 to state of leds(0xff80)
              andi $2, $1, 0x3e0      # calculate value1 and store to $2
              srl $2, $1, 5           # calculate value1 and store to $2
              andi $3, $1, 0x1f       # calculate value2 and store to $3
              jal get_op              # call get_op
              jal do_op               # call do_op (passing $2 and $3)
              add $30, $4, $0         # move $4(result) to $30
              addi $29, $0, 0         # set pos to 0 (right pair)
              jal show                # call show (passing $30 and $29)
              add $30, $2, $0         # move $2(value1) to $30
              addi $29, $0, 2         # set pos to 2 (left pair)
              jal show                # call show (passing $30 and $29)
              add $30, $3, $0         # move $3(value2) to $30
              addi $29, $0, 1         # set pos to 1 (middle pair)
              jal show                # call show (passing $30 and $29)
              j main_loop             # loop forever
```

简要介绍

以上 MIPS 程序实现了一个多功能计算器，能对输入的两个数进行加法、减法（大减小，即差的绝对值）及异或三种操作，并把操作数和结果显示到七段数码管上。可通过按钮切换运算模式。

主循环

程序运行时将初始计算模式设定为加法（存储在 `$7` 中），然后进入主循环。主循环获取开关的状态，显示到对应的 LED 上，并分离出两个操作数（以 `SW[9..5]` 和 `SW[4..0]` 为两个操作数，是范围在 [0, 32) 的整数）。接着调用 `get_op` 函数获取当前运算模式，调用 `do_op` 函数执行运算，然后调用 3 次 `show` 函数将两个操作数和计算结果分别显示到三组七段数码管上。主循环是一个死循环，不断地执行。

获取当前运算模式

`get_op` 函数获取当前按钮状态，并作出如下响应：

- 当 key3 被按下时，当前运算模式切换为加法（`$7 <- 0`）
- 当 key2 被按下时，当前运算模式切换为减法（`$7 <- 1`）
- 当 key1 被按下时，当前运算模式切换为异或（`$7 <- 2`）
- 若无按钮按下，则当前运算模式保持原状

执行运算

`do_op` 函数按照 `$7` 中存储的当前运算模式执行相应的运算。在减法操作中，若所得结果为负数，则取相反数再返回结果。

七段数码管显示

`show` 函数调用 `split` 函数将一个两位十进制数分割成十位和个位，再分别显示到给定位置的数码管上。传入的参数 `$29` 表示要显示到哪一组数码管，2 为左侧组，1 为中间组，0 为右侧组（这样编码方便地址计算）。通过调用 `sevenseg` 函数进行七段数码管的译码。

十进制数分割

`split` 函数实现了将一个两位十进制数分割成十位和个位的操作（相当于实现了求整除 10 的商和余数）。这是通过不断减 10 直到变为负数来实现的。

七段数码管译码

通过读取数据存储中的码表实现。

```
DEPTH = 32;            % Memory depth and width are required %
WIDTH = 32;            % Enter a decimal number %
ADDRESS_RADIX = HEX;   % Address and value radixes are optional %
DATA_RADIX = HEX;      % Enter BIN, DEC, HEX, or OCT; unless %
CONTENT                % otherwise specified, radixes = HEX %
BEGIN
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' %
END ;
```

仿真验证

```
`timescale 1ps/1ps

module sc_computer_sim;
    reg         resetn_sim;
    reg         clock_50M_sim;
    reg         mem_clk_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;
    reg  [9:0]  sw;
    reg  [3:1]  key;
    wire [6:0]  hex5, hex4, hex3, hex2, hex1, hex0;
    wire [9:0]  led;

    initial begin
        sw <= 10'b1010101010;
        key <= 3'b111;
```

```
    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, data_sim, wmem_sim,
        sw, key, hex5, hex4, hex3, hex2, hex1, hex0, led);

    initial begin // Generate clock.
        clock_50M_sim = 1;
        while (1)
            #2 clock_50M_sim = ~clock_50M_sim;
    end

    initial begin // Generate mem_clk.
        mem_clk_sim = 1;
        while (1)
            #1 mem_clk_sim = ~mem_clk_sim;
    end

    initial begin // Generate a reset signal at the start.
        resetn_sim = 0;
        while (1)
            #5 resetn_sim = 1;
    end

    initial begin // Simulate switch changes.
        #1800 sw = ~sw;
    end

    initial begin // Simulate key presses.
        while (1) begin
            #600 key <= 3'b101; // key2 pressed, should change to sub mode
            #600 key <= 3'b110; // key1 pressed, should change to xor mode
            #600 key <= 3'b011; // key3 pressed, should change to add mode
        end
    end

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

该仿真测试代码模拟出 CPU 工作所需要的时钟信号和复位信号，并在适当的时刻模拟开关和按钮操作。可在仿真过程中观察各个输出信号的值是否正确，以验证 CPU 工作是否正常。

实验总结

实验结果

实验代码经过编译综合，载入到开发板后，运行自己所编写的汇编程序，能正常完成预期的多功能计算器的功能。经过仿真验证，查看各个输出信号的值，我们可以认为本次 CPU 的设计是符合预期的。

经验教训

1. 在使用 Altera 提供的 megafunction `RAM:1-PORT` 和 `ROM:1-PORT` 时，不要让向导自动生成 `*_bb.v` 文件（要手动取消勾选），否则会在这些文件中产生重名的 RAM 和 ROM 模块，造成冲突，导致 RAM/ROM 功能不正常，仿真值全为高阻态。
2. 在硬件上若发现有的灯处于半亮状态，则是信号值不稳定，在 0 和 1 之间高频地变化。这时可以把时钟周期放慢（到 0.1 s 甚至 1 s），仔细观察硬件的行为是否出现了异常。
3. 不要忽略 Quartus 在编译期间报告的警告。虽然编译可能通过了，但这些警告可能是导致一些错误和异常行为的来源。比如 "latch … has unsafe behavior"，载入到板子上出现的信号值不稳定现象就与这个警告有关，应该去修改给信号赋值的代码来消除 unsafe behavior。建议右键单击每个警告查看相应的 Help 网页，阅读里面详细的说明。

感受

在本次实验中我实现了一个结构简单的单周期 CPU，并在其上用汇编码实现出了一个简单的多功能计算器，更清晰地理解了 CPU 的工作步骤、计算机各部件之间的通信以及层次的组织关系。同时，调试硬件也需要很多耐心，想办法定位问题之所在，必要时和老师、同学沟通能更有效地解决问题。