实验：基本单周期 CPU 设计实验报告

姓名：Meng Yit Koh 学号：517030990022

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。

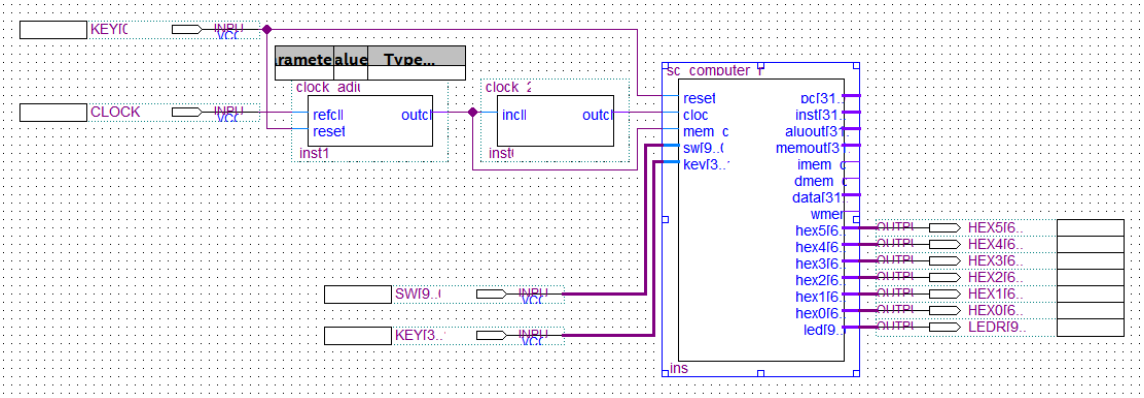
本次实验所用的仪器以及软件

- DE1-SOC 实验板
- Quartus Prime Lite 18.1.0
- ModelSim - Intel FPGA Starter Edition 10.5b

实验内容

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。
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，提供以 上两种指令集（MIPS）的应用功能的程序设计代码，并提供程序主 要流程图。

Block Diagram



clk, mem_clk

```
// 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
```

注解：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。

sc_computer 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
```

注解：主模块由三个部分组成。依照老师给的代码，在数据存储那部分实现了 I/O 端口扩展。

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]   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
```

注解：CPU 由程序计数器寄存器（ip）、控制器（sc_cu）、寄存器文件（regfile）、逻辑单元（alu）以及各部件信号的路组成。

指令控制器

```
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
```

注解：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

```
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
```

注解：共有 32 个寄存器，其中 0 号寄存器的值永远为 0，而对寄存器的写入在下一周期时钟上升沿生效。

ALU

```
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
```

注解：逻辑运算按照 aluc 控制信号的指示对操作数执行相应的运算。

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
```

注解：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。

数据存储和 I/O

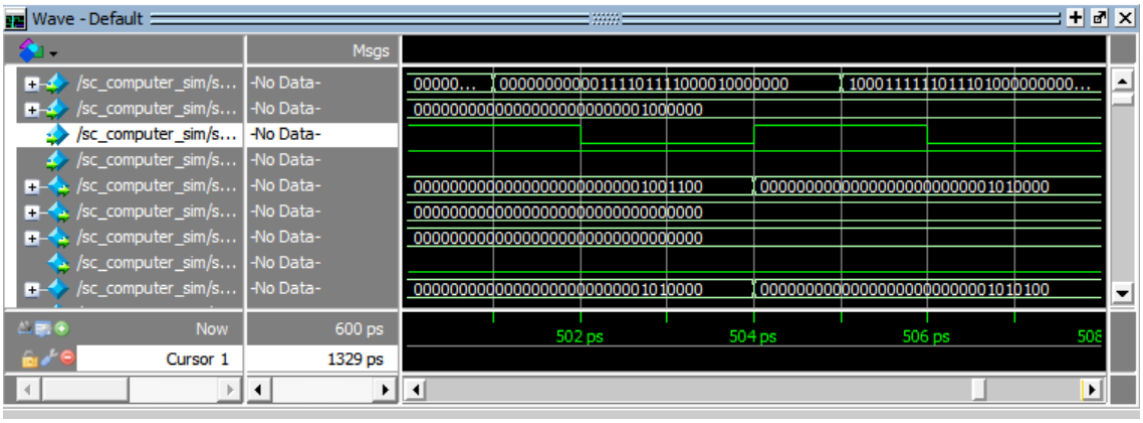
```
module sc_datamem(resetn, addr, datain, dataout, we, clock, mem_clk, dmem_clk,
    sw, key, hex5, hex4, hex3, hex2, hex1, hex0, led);
    input              resetn;
    input      [31:0]  addr, datain;
    input              we, clock, mem_clk;
    input      [9:0]   sw;
    input      [3:1]   key;
    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);
    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'hfffffff20: hex0 <= datain[6:0];
                32'hfffffff30: hex1 <= datain[6:0];
                32'hfffffff40: hex2 <= datain[6:0];
                32'hfffffff50: hex3 <= datain[6:0];
                32'hfffffff60: hex4 <= datain[6:0];
                32'hfffffff70: hex5 <= datain[6:0];
                32'hfffffff80: 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
```

注解：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。

预留一部分地址，将其映射为 I/O 端口。

仿真



注解：仿真结果符合预期，并没有无信号事件发生。ALU，CU 信号也会改变，表示信号在每个周期都在运动。

运行简单 **calculator** 程序

```
DEPTH = 4096;
WIDTH = 32;
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;
CONTENT
BEGIN
 0 : 3c070000;
 1 : 8c01ff00;
 2 : ac01ff80;
 3 : 302203e0;
 4 : 00011142;
 5 : 3023001f;
 6 : 8c05ff10;
 7 : 2006ffff;
 8 : 00a62826;
 9 : 30a60008;
 a : 20c6fffb;
 b : 00432020;
 c : 0080f020;
 d : 201d0000;
 e : 03e0a020;
 f : 001dd140;
10 : 235aff20;
11 : 0000e820;
12 : 23defff6;
13 : 001ee7c3;
14 : 141c0019;
15 : 23bd0001;
16 : 08000014;
17 : 23dc000a;
18 : 03a0f020;
19 : 001ef080;
1a : 8fdd0000;
1b : af5d0010;
```

```
1c : 0380f020;
1d : 001ef080;
1e : 8fdd0000;
1f : af5d0000;
20 : 08000001;
END;
```

注解：以上代码实现基本加法计算器。原理就是实现一个无限循环。在每个循环里把从 sw 映射的数据存到寄存器，实现加法（或减法）然后把结果存到另一个寄存器。然后 jump 到另一个 label，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。

实验里还有参考实现的加减法计算器。具体代码可以参考实验源代码文件。

实验总结

实验代码经过编译综合，载入到开发板后，能正常完成预期的 CPU 功能。I/O 处理良好，未出现奇怪显示状况。

感谢上海交通大学软件学院开设“数字电路设计这门课”，让我学习如何使用 FPGA 实现单周期 CPU 和基本 MIPS 指令。感谢王老师的教导。感谢热心的助教。