实验二：基本单周期 CPU 设计实验报告

杨健邦 515030910223

一、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。

二、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。（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。

三、预习内容：

1. 实验前仔细阅读 Altera-DE1-SOC User Manual 及相关用户应用数据手册，学习并掌握其板载相关资源的工作原理、连接方式、和应用注意事项。
2. 根据课程所讲单周期 CPU 设计原理，提前设计并仿真实现相关设计代码。

四、实验器材：

1. Altera-DE1-SOC 实验板套件 1 套
2. 万用表 1 台
3. 示波器 1 台

五、设计思想和方法：

1. Control Unit 的实现，先填写完成真值表，通过真值表来实现 control unit 模块，真值表见图一。

输入																			
指令	指令格式	op	rs	rt	rd	sa	func	z	pcsource [1..0]	aluc [3..0]	shift	aluimm	sext	wmem	wreg	m2reg	regrt	call	jal
add	add rd, rs, rt	000000	rs	rt	rd	00000	100000	x	0 0	x 0 0 0	0	0	x	0	1	0	0	0	
sub	sub rd, rs, rt	000000	rs	rt	rd	00000	100010	x	0 0	x 1 0 0	0	0	x	0	1	0	0	0	
and	and rd, rs, rt	000000	rs	rt	rd	00000	100100		0 0	x 0 0 1	0	0	x	0	1	0	0	0	
or	or rd, rs, rt	000000	rs	rt	rd	00000	100101		0 0	x 1 0 1	0	0	x	0	1	0	0	0	
xor	xor rd, rs, rt	000000	rs	rt	rd	00000	100110		0 0	x 0 1 0	0	0	x	0	1	0	0	0	
sll	sll rd, rt, sa	000000	00000	rt	rd	sa	000000	x	0 0	0 0 1 1	1	0	x	0	1	0	0	0	
srl	srl rd, rt, sa	000000	00000	rt	rd	sa	000010		0 0	0 1 1 1	1	0	x	0	1	0	0	0	
sra	sra rd, rt, sa	000000	00000	rt	rd	sa	000011		0 0	1 1 1 1	1	0	x	0	1	0	0	0	
jr	jr rs	000000	rs	00000	00000	00000	001000	x	1 0	x x x x	x	x	x	0	0	x	x	x	
指令	指令格式	op	rs	rt	rd	sa	func		pcsource [1..0]	aluc [3..0]	shift	aluimm	sext	wmem	wreg	m2reg	regrt	call	jal
addi	addi rt, rs, imm	001000	rs	rt	imm				0 0	0 0 0 0	0	1	1	0	1	0	1	0	
andi	andi rt, rs, imm	001100	rs	rt	imm				0 0	x 0 0 1	0	1	0	0	1	0	1	0	
ori	ori rt, rs, imm	001101	rs	rt	imm				0 0	x 1 0 1	0	1	0	0	1	0	1	0	
xori	xori rt, rs, imm	001110	rs	rt	imm				0 0	x 0 1 0	0	1	0	0	1	0	1	0	
lw	lw rt, imm(rs)	100011	rs	rt	imm				0 0	x 0 0 0	0	1	1	0	1	1	1	0	
sw	sw rt, imm(rs)	101011	rs	rt	imm				0 0	x 0 0 0	0	1	1	1	0	x	x	x	
beq	beq rs, rt, imm	000100	rs	rt	imm			0	0 0	x 1 0 0	0	0	1	0	0	x	x	x	
								1	0 1										
bne	bne rs, rt, imm	000101	rs	rt	imm			1	0 0	x 1 0 0	0	0	1	0	0	x	x	x	
								0	0 1										
lui	lui rt, imm	001111	00000	rt	imm				0 0	x 1 1 0	x	1	x	0	1	0	1	0	
j	j addr	000010	addr						1 1	x x x x	x	x	x	0	0	x	x	x	
jal	jal addr	000011	addr						1 1	x x x x	x	x	x	0	1	x	x	1	

图一：真值表

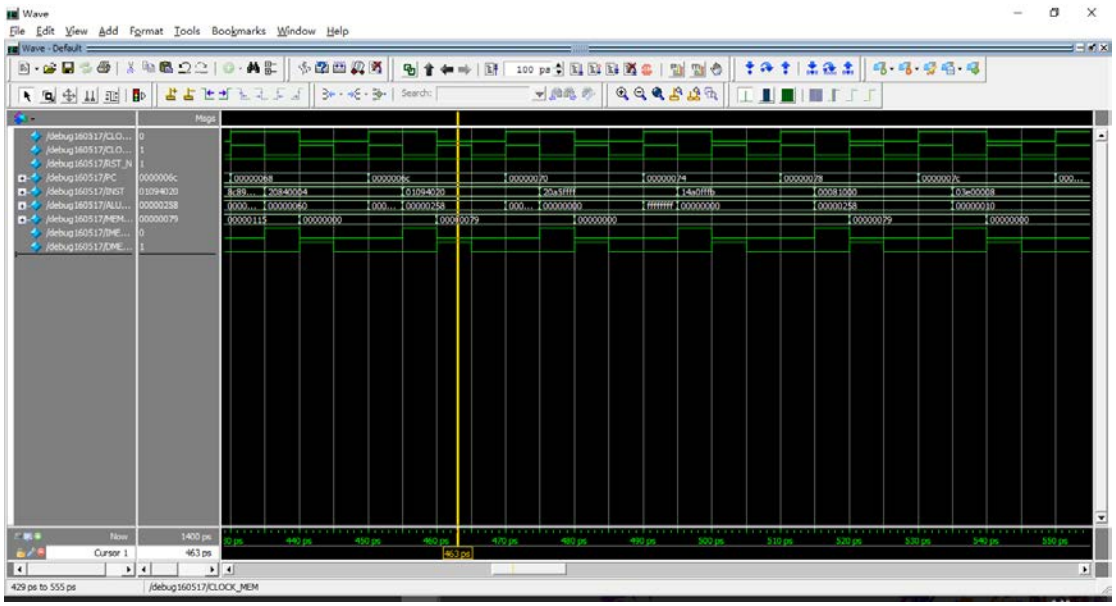
2. 填写 alu 模块：根据注释的提示填写即可。
3. 时钟的生成：需要提供同步的两个时钟，其中有一个时钟的频率是另一个时钟频率的两倍。基本的解决方法是，传入一个由 FPGA 板子 AF14 引脚生成的时钟，通过这个时钟再生成一个频率为其一半的时钟（通过一个寄存器保存，每次在传入时钟上升沿的时候取反），这样就可以保证两个时钟是同步的了。

```
module clock_gen(origin,cpu_clk,mem_clk);
        input origin;
        output reg cpu_clk;
        output mem_clk;

        assign mem_clk = origin;

        always @ (posedge origin)
                cpu_clk <= ~cpu_clk;

        initial cpu_clk <= 0;

endmodule
```

4. IO 设计：板上的 10 个开关分成 2 组，每组都作为一个输入端口。6 个数码管分成 3 组，每组都作为一个输出端口，显示的是十进制的数字，由于数量的限制，只能显示十位和个位的数字，通过一个 num2sevseg 模块来转换。
5. 汇编代码的设计：有一个永真循环，从两个输入端口读入数据，再把两个输入加起来，三个输出端口分别显示两个加数和结果。最高只能显示 31+31=62 的数据。

```
0 : 20020080; % (00) addi $2, $0, 10000000b # address 80h %
1 : 20030084; % (04) addi $3, $0, 10000100b # address 84h %
2 : 20010088; % (08) addi $1, $0, 10001000h # address 88h %
3 : 200400c0; % (0c) addi $4, $0, 11000000b # address c0h %
4 : 200500c4; % (10) addi $5, $0, 11000100b # address c4h %
5 : 8c860000; % (14) loop: lw $6, 0($4) # input data from [c0h] %
6 : 8ca70000; % (18)       lw $7, 0($5) # input data from [c4h] %
7 : 00c74020; % (1c) add $6, $7, $8 # $8 = $6 + $7 %
8 : ac460000; % (20) sw $6, 0($2) # output [C0h] to [80h] %
9 : ac670000; % (24) sw $7, 0($3) # output [c4h] to [84h] %
10 : ac280000; % (28) sw $8, 0($1) # output [c0h] + [c4h] to [88h] %
11 : 08000005; % (2c) j loop # loop %
```

6. 仿真：需要将 sc_computer 替换成 sc_computer_sim，并且还要添 altera_mf.v 以及 220model.v 两个库才能仿真成功。



六、心得体会：

1. 如果发现烧不进去板子，可以尝试添加一个顶层 bdf 文件，然后把所有引脚分配完，就能烧进板子。
2. 需要读取 ram 的程序的仿真需要添加 altera_mf.v 以及 220model.v 两个库。
3. 变量写错不会导致编译失败，需要很认真地检查。

```
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
```

```
module clock_gen(origin,cpu_clk,mem_clk);
   input origin;
   output reg cpu_clk;
   output mem_clk;

   assign mem_clk = origin;

   always @ (posedge origin)
      cpu_clk <= ~cpu_clk;

   initial cpu_clk <= 0;

endmodule
```

```
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
```

```
module sc_datamem (addr,datain,dataout,we,clock,mem_clk,dmem_clk,
out_port0,out_port1,out_port2,in_port0,in_port1,mem_dataout,io_read_data);

   input [31:0] addr;
   input [31:0] datain;
   input we, clock,mem_clk;
   input [4:0] in_port0,in_port1;

   output [31:0] dataout;
   output dmem_clk;
   output [31:0] out_port0,out_port1,out_port2;
   output [31:0] mem_dataout;
   output [31:0] io_read_data;

   wire [31:0] dataout;
   wire dmem_clk;

   wire write_enable;
   wire wrire_datamem_enable;
   wire [31:0] mem_dataout;

   assign dmem_clk = mem_clk & ( ~ clock) ; //注意
   assign write_enable = we & ~clock; //注意
   assign write_datamem_enable= write_enable & ( ~ addr[7]); //注意
   assign write_io_output_reg_enable= write_enable & ( addr[7]); //注意

   mux2x32 mem_io_dataout_mux(mem_dataout,io_read_data,addr[7],dataout);
   // module mux2x32 (a0,a1,s,y);
   // when address[7]=0, means the access is to the datamem.
   // that is, the address space of datamem is from 000000 to 011111 word(4
bytes)

   lpm_ram_dq_dram dram(addr[6:2],dmem_clk,datain,write_datamem_enable,
mem_dataout );
   // when address[7]=1, means the access is to the I/O space.
   // that is, the address space of I/O is from 100000 to 111111 word(4 byte

   io_output_reg io_output_regx2 (addr,datain,write_io_output_reg_enable,
dmem_clk,out_port0,out_port1,out_port2);

   io_input_reg io_input_regx2(addr,dmem_clk,io_read_data,in_port0,in_port1);

endmodule
```

```
module io_input_reg (addr,io_clk,io_read_data,in_port0,in_port1);
   input [31:0] addr;
   input io_clk;
   input [4:0] in_port0,in_port1;
   output [31:0] io_read_data;

   reg [31:0] in_reg0; // input port0
   reg [31:0] in_reg1; // input port1

   io_input_mux io_imput_mux2x32(in_reg0,in_reg1,addr[7:2],io_read_data);

   always @(posedge io_clk)
      begin
         in_reg0[4:0] <= in_port0; // 输入端口在io_clk上升沿时进行数据锁存
         in_reg1[4:0] <= in_port1; // 输入端口在io_clk上升沿时进行数据锁存
         // more ports 可根据需要设计更多的输入端口。
      end
endmodule

module io_input_mux(a0,a1,sel_addr,y);
   input [31:0] a0,a1;
   input [ 5:0] sel_addr;
   output [31:0] y;
   reg [31:0] y;

   always @ (*)
      case (sel_addr)
         6'b110000: y = a0;
         6'b110001: y = a1;
         // more ports 可根据需要设计更多的端口。
      endcase
endmodule
```

```
module io_output_reg (addr,datain,write_io_enable,io_clk,out_port0,out_port1,
out_port2);
   input [31:0] addr,datain;
   input write_io_enable,io_clk;

   output [31:0] out_port0,out_port1,out_port2;

   reg [31:0] out_port0; // output port0
   reg [31:0] out_port1; // output port1
   reg [31:0] out_port2; // output port1

   always @(posedge io_clk)
      begin
         if (write_io_enable == 1)
            case (addr[7:2])
               6'b100000: out_port0 <= datain; // 80h
               6'b100001: out_port1 <= datain; // 84h
               6'b100010: out_port2 <= datain; // 84h
               // more ports 可根据需要设计更多的输出端口。
            endcase
      end
endmodule
```

```
module num2sevenseg(num, sevenseg0, sevenseg1);
   input  [31:0] num;
   output [6:0]  sevenseg0,sevenseg1;

   reg [3:0] units,tens;

   sevenseg high(tens, sevenseg0);
   sevenseg low(units, sevenseg1);

   always @(*)
      begin
         units <= num % 10;
         tens  <= (num % 100) / 10;
      end

endmodule
```