基本单周期CPU设计

杨子超 517030910330 F1703013

1. 实验目的

- 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。
- 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。
- 掌握I/O端口的设计方法，理解I/O地址空间的设计方法。
- 会通过设计I/O端口与外部设备进行信息交互。

2. 实验所用仪器及元器件

DE1-SOC实验板 1套

3. 实验任务

3.1 实验内容和任务

- 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。
- 利用实验提供的标准测试程序代码，完成仿真测试。
- 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。
- 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。
- 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）。
- 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。
- 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）。

- 在实现MIPS基本20条指令的基础上，**掌握新指令的扩展方法。**
- 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，提供以上指令集全覆盖的测试应用功能的程序设计代码，并提供程序主要流程图。

3.2 设计过程

3.2.1 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

王赓老师已经给出了顶层设计和大部分的文件，需要我做的主要有三部分工作，首先填写完善20条指令的真值表，其次，根据真值表完善CU.v文件，生成相应的控制指令，最后完善ALU.v文件，执行相应的计算。特别地，在实际分配管脚的过程中，我发现不能把DE1-SOC上25HMz的时钟连接到CPU上，所以我更改了相关设计，mem_clk连接的是50MHz的时钟，而clock的则由mem_clk分频产生，这样clock就不是sc_computer的输入信号，而是由一个分频模块half_frequency产生。

3.2.1.1 填写完善真值表

输入									z	新pc值来源	alu操作	alu第一操作数是rs还是sa即shift amount	alu第二操作数是rt还是立即数	立即数0扩展还是符号扩展	是否写数据RAM	是否写寄存器，下一个时钟周期的起作用	要写到寄存器的数据是0alu结果还是读数据RAM	要写入哪个数据寄存器，rd指向的还是rt指向的	jal指令
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
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		x	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		x	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		x	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		x	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		x	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						pcsource [1..0]	aluc [3..0]	shift	aluimm	sext	wmem	wreg	m2reg	regrt	call jal
addi	addi rt, rs, imm	001000	rs	rt						0 0	x 0 0 0	0	1	1	0	1	0	1	0
andi	andi rt, rs, imm	001100	rs	rt						0 0	x 0 0 1	0	1	0	0	1	0	1	0
ori	ori rt, rs, imm	001101	rs	rt						0 0	x 1 0 1	0	1	0	0	1	0	1	0
xori	xori rt, rs, imm	001110	rs	rt						0 0	x 0 1 0	0	1	0	0	1	0	1	0
lw	lw rt, imm(rs)	100011	rs	rt						0 0	x 0 0 0	0	1	1	0	1	1	1	0
sw	sw rt, imm(rs)	101011	rs	rt						0 0	1 0 0 0	0	1	1	1	0	x	1	x
beq	beq rs, rt, imm	000100	rs	rt					0	0 0	x 1 0 0	0	0	1	0	0	0	0	0
									1	0 1									
bne	bne rs, rt, imm	000101	rs	rt					0	0 1	x 1 0 0	0	0	1	0	0	0	0	0
									1	0 0									
lui	lui rt, imm	001111	00000	rt						0 0	x 1 1 0	0	1	x	0	1	0	1	0
j	j addr	000010	address 26位							1 1	x x x x	x	x	x	0	0	x	x	x
jal	jal addr	000011								1 1	x x x x	x	x	x	0	1	x	x	1

3.2.1.2 根据真值表完善CU.v文件

```
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


    // R型指令
    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];   // 1000110
    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


    // I型指令
    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
```

```
34    wire i_beq  = ~op[5] & ~op[4] &  ~op[3] & op[2] & ~op[1] & ~op[0];
 //000100
35    wire i_bne  = ~op[5] & ~op[4] &  ~op[3] & op[2] & ~op[1] & op[0];
//000101
36    wire i_lui  = ~op[5] &   ~op[4] & op[3] & op[2] & op[1] &   op[0];
 //001111
37
38
39    // J型指令
40    wire i_j    = ~op[5] & ~op[4] &  ~op[3] & ~op[2] & op[1] & ~op[0];
 //000010
41    wire i_jal  =  ~op[5] & ~op[4] &  ~op[3] & ~op[2] & op[1] & op[0];
 //000011
42
43
44    assign pcsource[1] = i_jr | i_j | i_jal;
45    assign pcsource[0] = ( i_beq & z ) | (i_bne & ~z) | i_j | i_jal ;
46
47    assign wreg = i_add | i_sub | i_and | i_or   | i_xor  |
48                  i_sll | i_srl | i_sra | i_addi | i_andi |
49                  i_ori | i_xori | i_lw | i_lui  | i_jal;
50
51    assign aluc[3] =  i_sra | i_sw;
52    assign aluc[2] =  i_sub | i_or | i_srl | i_sra | i_ori |i_bne
|i_beq|i_lui;
53    assign aluc[1] =  i_xor |i_sll | i_srl | i_sra | i_xori|i_lui;
54    assign aluc[0] =  i_and | i_or | i_andi| i_ori |i_sll | i_srl |
i_sra ;
55
56
57    assign shift   = i_sll | i_srl | i_sra ;
58    assign aluimm  =  i_addi| i_ori| i_andi|i_xori| i_lw| i_sw |i_lui;

59    assign sext    =  i_addi| i_lw |i_sw |i_beq |i_bne;
60    assign wmem    =  i_sw;
61    assign m2reg   =  i_lw;
62    assign regrt   = i_addi| i_ori| i_andi|i_xori| i_lw| i_sw| i_lui;
63    assign jal     = i_jal;
64
65
66 endmodule
```

3.2.1.3 完善ALU.v文件

```
   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
```

3.2.1.3 分频模块half_frequency

```
module half_frequency(resetn,mem_clk,clock);
    input resetn,mem_clk;
    output clock;
    reg clock;
```

```
    initial
    begin
        clock = 0;
    end
    always @(posedge mem_clk)
    begin
        if(~resetn)
            clock <= 0;
        clock <= ~clock;
    end
endmodule
```

3.2.2 实现CPU与外部设备的输入输出端口设计

首先要阅读实验指导书上实验三：外部I/O及接口扩展实验部分的相关内容。对于展示设计，我的想法是完成一个加法器和减法器，当开关置于加法器时，通过八个开关输入两个4位二进制数，计算结果，将被加数，加数和和用LED显示出来；当开关置于减法器时，通过八个开关输入两个4位二进制数，计算结果，将被减数，减数和差用LED显示出来。如果差为负数，那么显示绝对值，同时负数指示灯亮起。

将王赓老师给出的io_output_reg.v 和io_input_reg.v两个文件加入进来，同时对sc_computer.v中的sc_computer模块进行修改，加入端口输入（三个输入即第一操作数，第二操作数，加减控制信号)和输出的信号（六个LED的控制信号和负数指示灯信号)，设置LED的输出值，同时添加数显的sevenseg模块；由于有三个输入，同时作差结果为负数时，需要相应的处理，所以io_output_reg.v 也需要相应的修改；相应地，io_input_reg.v文件也需要相应的修改；最后对sc_datamem.v文件进行修改。

最后，要给出对应的程序，实现上述功能。思想是读入第一操作数，读入第二操作数，读入加减控制信号，根据加减控制信号，决定执行加法指令还是减法指令，将结果输出，重复此循环。写出汇编代码，同时使用shawn233学长编写的mif.py程序即可生成相应的.mif文件。

3.2.2.1 修改sc_computer.v中sc_computer模块

```
module sc_computer ( resetn, mem_clk,  pc,  inst,aluout,
in_port0,in_port1,in_port_sub,out_port0,out_port1,out_port2,HEX0,HEX1,
HEX2,HEX3,HEX4,HEX5,LEDR4);

input [3:0]  in_port0, in_port1;
```

```
input in_port_sub; // 加减控制信号
output LEDR4; // 负数指示灯
output [31:0]   out_port0, out_port1, out_port2,aluout;
output wire [6:0] HEX0,HEX1,HEX2,HEX3,HEX4,HEX5; //六个数显
input resetn,mem_clk; //复位信号
output [31:0] pc, inst;// pc值和指令
wire [31:0] data,aluout,memout; // 模块间互联传递数据或控制信息的信号线。
wire wmem,clock; // 模块间互联传递数据或控制信息的信号线。

half_frequency hf(resetn,mem_clk,clock); //分频信号
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.

wire [3:0] op_0_ge,op_1_ge,op_2_ge,op_0_shi,op_1_shi,op_2_shi; //六个数显的输出
assign op_0_ge=out_port0%10;
assign op_0_shi=out_port0/10;
assign op_1_ge=out_port1%10;
assign op_1_shi=out_port1/10;
assign op_2_ge=out_port2%10;
assign op_2_shi=out_port2/10;

sc_datamem  dmem (aluout,data,memout,wmem,clock,mem_clk,dmem_clk,resetn,in_port0,in_port1,in_port_sub,out_port0,out_port1,out_port2,io_read_data,LEDR4); // data memory.

sevenseg s0(op_2_ge,HEX4);  //实例化六个数显
sevenseg s1(op_2_shi,HEX5);

sevenseg s2(op_1_ge,HEX2);
sevenseg s3(op_1_shi,HEX3);

sevenseg s4(op_0_ge,HEX0);
sevenseg s5(op_0_shi,HEX1);

endmodule
```

3.2.2.2 sc_computer.v中添加数显的sevenseg模块

```
module sevenseg ( data, ledsegments);
input [3:0] data;
output ledsegments;
reg [6:0] ledsegments;
always @ (*)
case(data)
// gfe_dcba // 7段LED数码管的位段编号
// 654_3210 // DE1-SOC板上的信号位编号
0: ledsegments = 7'b100_0000; // DE1-SOC板上的数码管为共阳极接法。
1: ledsegments = 7'b111_1001;
2: ledsegments = 7'b010_0100;
3: ledsegments = 7'b011_0000;
4: ledsegments = 7'b001_1001;
5: ledsegments = 7'b001_0010;
6: ledsegments = 7'b000_0010;
7: ledsegments = 7'b111_1000;
8: ledsegments = 7'b000_0000;
9: ledsegments = 7'b001_0000;
default: ledsegments = 7'b111_1111; // 其它值时全灭。
endcase
endmodule
```

3.2.2.3 修改io_output_reg.v文件

```
module
io_output_reg(addr,datain,write_io_enable,io_clk,clrn,out_port0,out_po
rt1,out_port2,LEDR4);

    input [31:0] addr, datain;
    input write_io_enable, io_clk;
    input clrn; // 输出清0
    output [31:0] out_port0, out_port1, out_port2; //第一操作数 第二操作
数 结果

    reg [31:0] out_port0, out_port1, out_port2;
    output LEDR4;  // 负数指示灯
    reg LEDR4;
    always @(posedge io_clk or negedge clrn)
```

```
    begin
        if(clrn == 0)   // 输出清0
        begin
            out_port0 <= 0;
            out_port1 <= 0;
            out_port2 <= 0;
        end
        else
        begin
            if(write_io_enable == 1)  // 写到输出端口
            case(addr[7:2])
                6'b100000: out_port0 <= datain;//80h
                6'b100001: out_port1 <= datain;//84h
                6'b100010:   //88h  增加的输出端口
                    begin
                        if(datain[31]==1)  // 作差结果为负
                        begin
                            LEDR4<=1;  // 负数指示灯亮
                            out_port2 <= ~datain+1;  //取相反数
                        end
                    else
                        begin
                            LEDR4<=0;
                            out_port2 <= datain;
                        end
                    end
            endcase
        end
    end
endmodule
```

3.2.2.4 修改io_input_reg.v文件

```
module io_input_reg
(addr,io_clk,io_read_data,in_port0,in_port1,in_port_sub);
//inport:  外部直接输入进入ioreg
    input  [31:0]  addr;
    input          io_clk;
    input  [31:0]  in_port0,in_port1,in_port_sub;
    output [31:0]  io_read_data;
```

```
7
8       reg     [31:0]  in_reg0;      // input port0
9       reg     [31:0]  in_reg1;      // input port1
10      reg     [31:0]  in_reg2;      // input port2   加减控制信号
11
12      io_input_mux
    io_imput_mux2x32(in_reg0,in_reg1,in_reg2,addr[7:2],io_read_data);
13
14      always @(posedge io_clk)
15      begin
16          in_reg0 <= in_port0;    // 输入端口在 io_clk 上升沿时进行数据锁存
17          in_reg1 <= in_port1;    // 输入端口在 io_clk 上升沿时进行数据锁存
18          in_reg2 <= in_port_sub; // 输入端口在 io_clk 上升沿时进行数据锁存
19      end
20  endmodule
21
22
23  module io_input_mux(a0,a1,a2,sel_addr,y);
24      input   [31:0]  a0,a1,a2;
25      input   [ 5:0]  sel_addr;
26      output  [31:0]  y;
27      reg     [31:0]  y;
28      always @ *
29      case (sel_addr)
30
31         6'b110000: y = a0;
32         6'b110001: y = a1;
33         6'b110010: y = a2;
34         endcase
35  endmodule
```

3.2.2.5 修改sc_datamem.v文件

```
1   module sc_datamem
    (addr,datain,dataout,we,clock,mem_clk,dmem_clk,resetn,in_port0_tmp,in_
    port1_tmp,in_portsub_tmp,out_port0,out_port1,out_port2,io_read_data,LE
    DR4);
2
3       input   [31:0]  addr;  //地址
```

```
    input  [31:0]  datain; // 输入数据
   input          we, clock, mem_clk,in_portsub_tmp;
    input [3:0] in_port0_tmp, in_port1_tmp; // I/O输入
    output [31:0]   out_port0, out_port1, out_port2,io_read_data;//输出
    output [31:0]  dataout; // 输出数据给CPU
   output           dmem_clk,LEDR4;
    input               resetn;
   wire [31:0]      io_read_data;
    wire [31:0]     mem_dataout;
   wire           dmem_clk;
   wire           write_enable, write_io_enable, write_datamem_enable;//写谁
    wire [31:0] in_port0,in_port1,in_port_sub;//加减控制信号
    assign in_port0={28'b0,in_port0_tmp}; // 位数扩展
   assign in_port1={28'b0,in_port1_tmp};
    assign in_port_sub={31'b0,in_portsub_tmp};

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

    mux2x32 mem_io_dataout_mux(mem_dataout,io_read_data,addr[7],dataout); //选择向CPU输出的数据
    lpm_ram_dq_dram dram(addr[6:2],dmem_clk,datain,write_datamem_enable,mem_dataout);

   io_output_reg io_output_regx2(addr,datain,write_io_enable,dmem_clk,resetn,out_port0, out_port1,out_port2,LEDR4);  // I/O端口输出

    io_input_reg io_input_regx2(addr,dmem_clk,io_read_data,in_port0,in_port1,in_port_sub);//I/O端口输入

endmodule
```

3.2.2.6 汇编指令及对应的.mif文件

```
          addi $1,$0,192
          addi $2,$0,128
          addi $6,$0,1
    loop: lw $3,0($1)
          lw $4,4($1)
          lw $5,8($1)
          sw $3,0($2)
          sw $4,4($2)
          beq  $5,$6,subsub
          add $7,$3,$4
          sw $7,8($2)
          j loop
  subsub: sub $7,$3,$4
          sw $7,8($2)
          j loop
```

```
DEPTH = 64;           % 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 %
% otherwise specified, radixes = HEX %
CONTENT
BEGIN
0 : 200100c0;  %  addi $1,$0,192        |
00100000000000010000000011000000 %
1 : 20020080;  %  addi $2,$0,128        |
00100000000000010000000010000000 %
2 : 20060001;  %  addi $6,$0,1          |
00100000000001100000000000000001 %
3 : 8c230000;  % loop: lw $3,0($1)      |
10001100001000110000000000000000 %
4 : 8c240004;  %    lw $4,4($1)         |
10001100001001000000000000000100 %
5 : 8c250008;  %    lw $5,8($1)         |
10001100001001010000000000001000 %
6 : ac430000;  %    sw $3,0($2)         |
10101100001000110000000000000000 %
7 : ac440004;  %    sw $4,4($2)         |
10101100001000100000000000000100 %
```

```
8 : 10a60003;  % beq  $5,$6,subsub      |
00010000101001100000000000000011 %
9 : 00643820;  % add $7,$3,$4           |
00000000011001000011100000100000 %
a : ac470008;  %  sw $7,8($2)            |
10101100010001110000000000001000 %
b : 08000003;  % j loop                  |
00001000000000000000000000000011 %
c : 00643822;  %  subsub:sub $7,$3,$4    |
00000000011001000011100000100010 %
d : ac470008;  %  sw $7,8($2)            |
10101100010001110000000000001000 %
e : 08000003;  % j loop                  |
00001000000000000000000000000011 %
END ;
```

3.3实验步骤

3.3.1 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

填写完成20条MIPS指令的真值表，完善CU.v和ALU.v文件。

3.3.2 利用实验提供的标准测试程序代码，完成仿真测试。

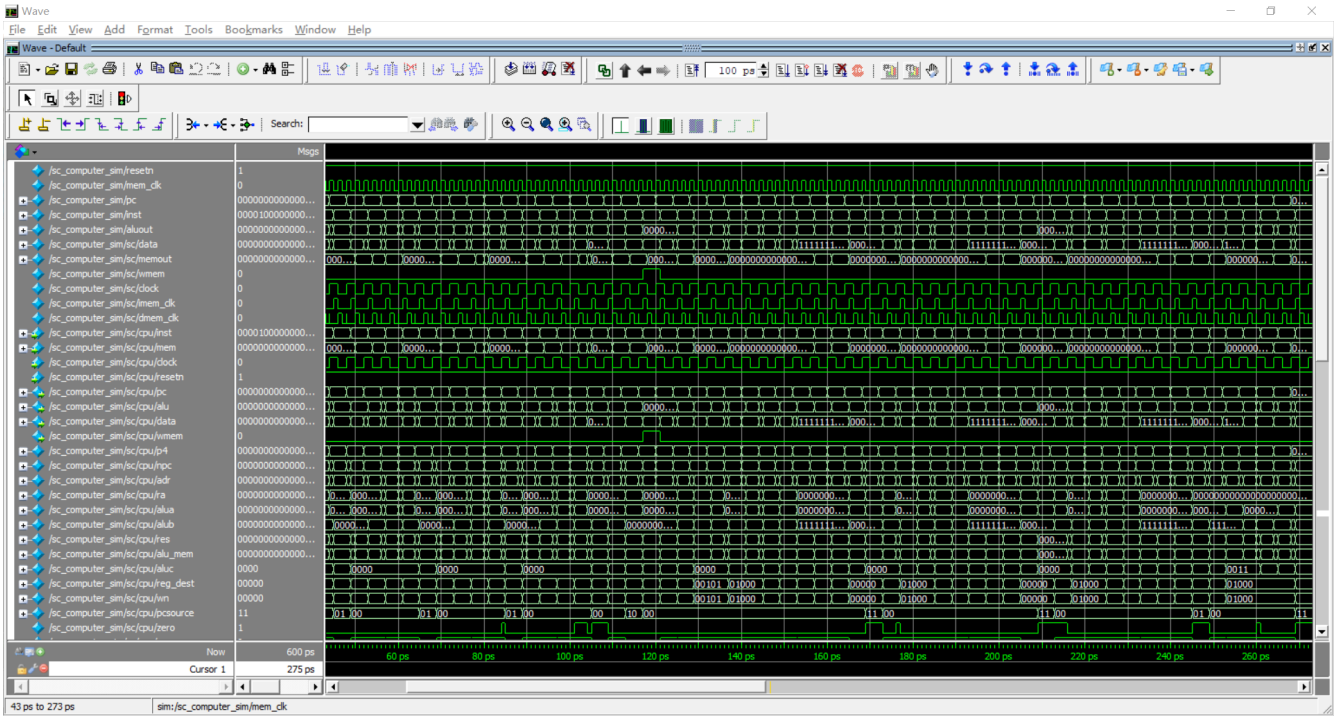
在王赓老师给出的激励文件基础上，根据实际设计情况，编写激励文件，在ModelSim中完成调试；由于已经加入了I/O扩展，所以激励文件中针对的加入I/O扩展的单周期CPU。

激励文件如下：

```
`timescale 1ps/1ps              // 仿真时间单位/时间精度
module sc_computer_sim ;
    reg in_port_sub;
    reg           resetn ;
    reg           mem_clk ;
    reg    [3:0] in_port0 ;
    reg   [3:0] in_port1 ;
    wire   [6:0] HEX0,HEX1,HEX2,HEX3,HEX4,HEX5;
```

```
    wire   [31:0]  out_port0 ,out_port1 ,out_port2;
    wire   [31:0]  pc ,inst ,aluout;

    sc_computer  sc ( resetn,  mem_clk,  pc,  inst,aluout,
in_port0,in_port1,in_port_sub,out_port0,out_port1,out_port2,HEX0,HEX1,
HEX2,HEX3,HEX4,HEX5);


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


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


    initial
        begin

          $display($time,"resetn=%b   mem_clk =%b", resetn ,  mem_clk
);

        end
endmodule
```

仿真结果如下:



3.3.3 实现CPU与外部设备的输入输出端口设计

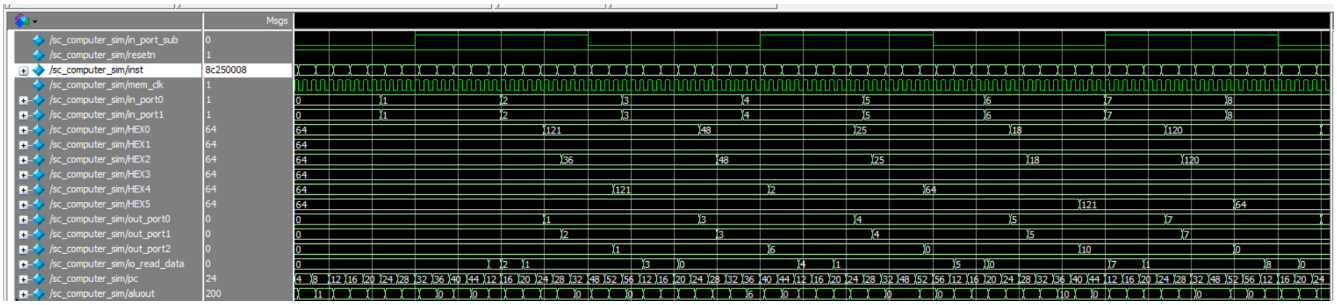
首先要阅读实验指导书上实验三：外部I/O及接口扩展实验部分的相关内容。对于展示设计，我的想法是完成一个加法器和减法器，当开关置于加法器时，通过八个开关输入两个4位二进制数，计算结果，将被加数，加数和和用LED显示出来；当开关置于减法器时，通过八个开关输入两个4位二进制数，计算结果，将被减数，减数和差用LED显示出来。如果差为负数，那么显示绝对值，同时负数指示灯亮起。

将王赓老师给出的io_output_reg.v 和io_input_reg.v两个文件加入进来，同时对sc_computer.v中的sc_computer模块进行修改，加入端口输入（三个输入即第一操作数，第二操作数，加减控制信号)和输出的信号（六个LED的控制信号和负数指示灯信号)，设置LED的输出值，同时添加数显的sevenseg模块；由于有三个输入，同时作差结果为负数时，需要相应的处理，所以io_output_reg.v 也需要相应的修改；相应地，io_input_reg.v文件也需要相应的修改；最后对sc_datamem.v文件进行修改。

最后，要给出对应的程序，实现上述功能。思想是读入第一操作数，读入第二操作数，读入加减控制信号，根据加减控制信号，决定执行加法指令还是减法指令，将结果输出，重复此循环。写出汇编代码，同时使用shawn233学长编写的mif.py程序即可生成相应的.mif文件。

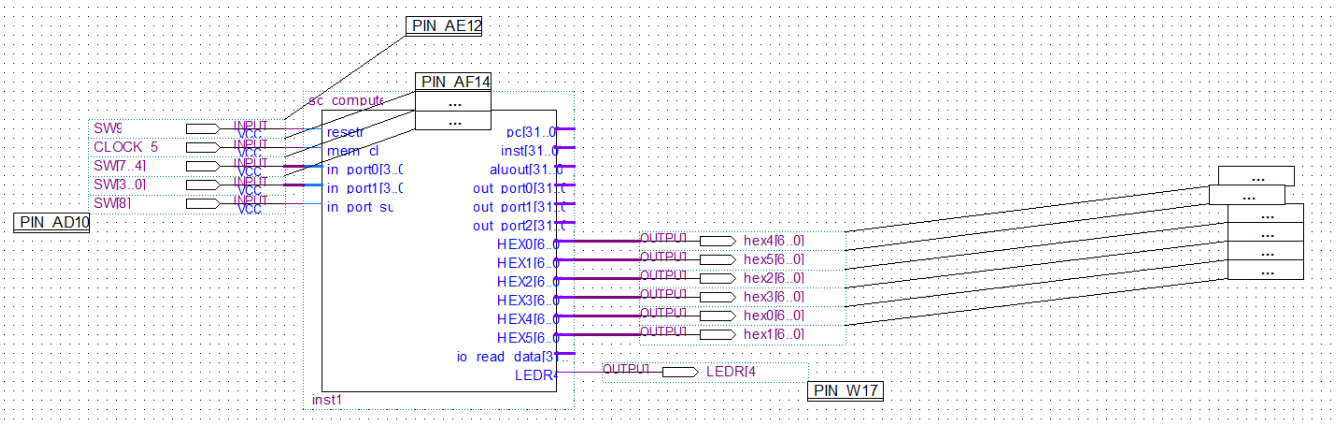
3.3.4 仿真验证设计

编写了激励文件，激励文件和前面的激励大同小异，但是增加了I/O端口的输入；同时生成了对应的.mif文件，即上面给出的.mif文件。仿真结果如下：



3.3.5 在quartusII中，进行系列操作

在quartus II中，生成symbol，添加到.bdf文件中，连接输入和输出，为管脚命名，分配管脚，结果如下：



3.3.6 编译，烧录至DE1-SOC

编译成功后，将.sof文件烧录至DE1-SOC上，实际验证功能。

3.4 Verilog代码

前面已经给出了被修改的全部Verilog代码，其余代码和王赓老师给出的代码一致，不再重复粘贴，在我的jbox上可以下载整个project，网址为https://jbox.sjtu.edu.cn/l/L04dtl。

实验总结

本次实验基本上顺利实现了实验的任务，达到了实验的目的。初步掌握了利用Verilog硬件描述语言和大规模可编程逻辑器件进行逻辑功能设计的原理和方法。在一个相对较高的层次上，利用软件和硬件的相辅相成和优势互补，设计实现了一个单周期CPU，并进行了I/O扩展。

本次实验，虽然王赓老师给出了大部分的代码，但是完成起来还是比较困难的，遇到的困难首先就是如何正确地填写真值表，因为真值表是CU.v的基础，而CU.v又是整个CPU的基础，所以对于真值表中的每一项，都要反复确认以保证正确。随后水到渠成地完成了CU.v和ALU.v的填写。

接下来，我把所有的代码都仔仔细细的看一遍，反复和王赓老师给出的单周期CPU的图进行对比。对这20条MIPS指令，我跟踪我设计的单周期CPU是怎么执行它们的。随后使用MdelSim进行仿真，这里出现的问题更多，经过了一番艰苦探索，解决了几个问题，如何编写激励文件；.mif文件的作用即初始化memory；如何设置lpm_rom_irom.v文件读取哪个.mif文件；如何把汇编语句转为.mif文件；在ModelSim的"Start Simulation" 窗体中的"Libraries"中添加元件需要的库；如何将想要查看的信号添加到wave中。

而在I/O扩展部分，在研读完实验指导书后，在明白原理的基础上，进行了相关的修改和设计，仿真成功后，编译，最终烧录到板子上。

本次实验，我有三个主要的体验和收获，首先是，一定要仔细研读实验指导书和王赓老师给出的代码，这是完成每一个实验的基础；其次，辩证地参考前人的成果，我参考了毛咏学长的代码，使用了shawn233学长编写的mif.py把汇编转为.mif文件，很多我疑惑的地方，都迎刃而解；最后一定要学会使用ModelSim，通过ModelSim直观地查看波形，可以方便地验证自己的结果，同时如果某个波形是高阻态(Hiz)，那么一定要仔细检查看是哪里出现了问题，ModelSim仿真成功后，再进行编译，烧录到板子上。不然，直接编译，再烧录到板子上，往往无法实现预想的功能，白白的浪费时间。