计算机组成实验报告

实验 2：基本单周期 CPU 设计

姓 名: 郭 嘉 宋
学 号: 517030910374
班 号: F1703015

上海交通大学
IEEE 试点班

2019 年 6 月 8 日

1 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。

2 实验所用的仪器以及元器件

Intel/DE1-SOC 实验板 1 套
示波器 1 台
数字万用表 1 台

3 实验内容和任务

1. 采用 Verilog HDL 在 quartus 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，掌握新指令的扩展方法。
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。

4 实验原理和结果展示

4.1 CPU 设计原理

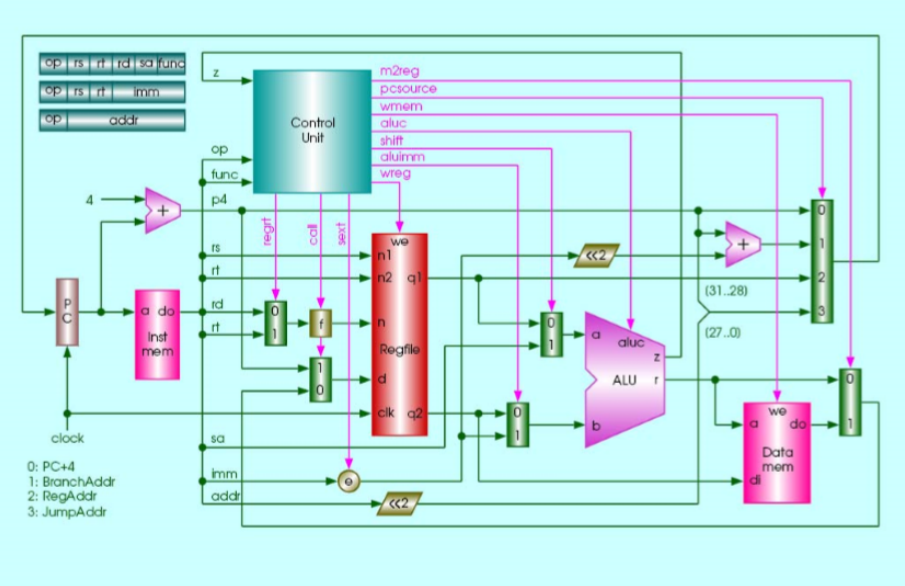

图 1: 单周期 CPU 实现图

其中需要修改设计 ALU 和 CU 模块即可。

4.2 ALU 模块实现

```
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;
   reg [31:0] tmp;
   always @ (a or b or aluc)
      begin                                    // event
```

```
11          casex (aluc)
12              4'b1011:                            //1011 HAMM
13              begin
14                tmp = a ^ b;
15                s = tmp[0] + tmp[1] + tmp[2] + tmp[3];
16              end
17              4'bx000: s = a + b;                 //x000 ADD
18              4'bx100: s = a - b;                 //x100 SUB
19              4'bx001: s = a & b;                 //x001 AND
20              4'bx101: s = a | b;                 //x101 OR
21              4'bx010: s = a ^ b;                 //x010 XOR
22              4'bx110: s = b << 16;           //x110 LUI: imm << 16bit
23              4'b0011: s = b << a;              //0011 SLL: rd <- (rt << sa)
24              4'b0111: s = b >> a;              //0111 SRL: rd <- (rt >> sa) (
                    logical)
25              4'b1111: s = $signed(b) >>> a;    //1111 SRA: rd <- (rt >> sa) (
                    arithmetic)
26              default: s = 0;
27          endcase
28          if (s == 0)  z = 1;
29             else z = 0;
30      end
31  endmodule
```

需要注意的是，Verilog 默认对寄存器变量采用无符号数保存，因此在 SRA 指令编写时，一定要添加 signed 函数限制成符号数，否则将永远补 0，不能达到算术移动的目的。

4.3 CU 模块实现

```
1  module sc_cu (op, func, z, wmem, wreg, regrt, m2reg, aluc, shift,
2                aluimm, pcsource, jal, sext);
3    input  [5:0] op,func;
4    input        z;
5    output       wreg,regrt,jal,m2reg,shift,aluimm,sext,wmem;
6    output [3:0] aluc;                                    //see alu.v
7    output [1:0] pcsource; //mux4x32 nextpc(p4,adr,ra,jpc,pcsource,npc);
8    wire r_type = ~|op;
9    wire i_add = r_type & func[5] & ~func[4] & ~func[3] &
10                ~func[2] & ~func[1] & ~func[0];          //100000
11   wire i_sub = r_type & func[5] & ~func[4] & ~func[3] &
12                ~func[2] &  func[1] & ~func[0];          //100010
13
14   //  please complete the deleted code.
15
16   wire i_and =  r_type & func[5] & ~func[4] & ~func[3] & func[2] &  ~func
         [1] & ~func[0];
17   wire i_or  =  r_type & func[5] & ~func[4] & ~func[3] & func[2] &  ~func
         [1] & func[0];
```

```
18
19  wire i_xor = r_type & func[5] & ~func[4] & ~func[3] & func[2] &  func[1]
        & ~func[0];
20  wire i_sll = r_type & ~func[5] & ~func[4] & ~func[3] & ~func[2] &  ~func
        [1] & ~func[0];
21  wire i_srl = r_type & ~func[5] & ~func[4] & ~func[3] & ~func[2] &  func
        [1] & ~func[0];
22  wire i_sra = r_type & ~func[5] & ~func[4] & ~func[3] & ~func[2] &  func
        [1] & func[0];
23  wire i_jr  = r_type & ~func[5] & ~func[4] & func[3] & ~func[2] &  ~func
        [1] & ~func[0];
24
25  wire i_addi = ~op[5] & ~op[4] &  op[3] & ~op[2] & ~op[1] & ~op[0]; //
        001000
26  wire i_andi = ~op[5] & ~op[4] &  op[3] &  op[2] & ~op[1] & ~op[0]; //
        001100
27
28  wire i_ori  =   ~op[5] & ~op[4] &  op[3] & op[2] & ~op[1] & op[0];   //
        complete by yourself.
29  wire i_xori =   ~op[5] & ~op[4] &  op[3] & op[2] & op[1] & ~op[0];
30  wire i_lw   =    op[5] & ~op[4] &  ~op[3] & ~op[2] & op[1] & op[0];
31  wire i_sw   = op[5] & ~op[4] &  op[3] & ~op[2] & op[1] & op[0];
32  wire i_beq  = ~op[5] & ~op[4] &  ~op[3] & op[2] & ~op[1] & ~op[0];
33  wire i_bne  = ~op[5] & ~op[4] &  ~op[3] & op[2] & ~op[1] & op[0];
34  wire i_lui  = ~op[5] &   ~op[4] & op[3] & op[2] & op[1] &   op[0];
35  wire i_j    = ~op[5] & ~op[4] &  ~op[3] & ~op[2] & op[1] & ~op[0];
36  wire i_jal  =  ~op[5] & ~op[4] &  ~op[3] & ~op[2] & op[1] & op[0];
37
38  wire I_HAMM = r_type & func[5] & func[4] & ~func[3] & ~func[2] & ~func[1]
         & ~func[0];
39                                                                  //I_HAMM
                                                                        FUNC
                                                                        =110000


40
41  assign pcsource[1] = i_jr | i_j | i_jal;
42  assign pcsource[0] = ( i_beq & z ) | (i_bne & ~z) | i_j | i_jal ;
43
44  assign wreg = i_add | i_sub | i_and | i_or   | i_xor  |
45                i_sll | i_srl | i_sra | i_addi | i_andi |
46                i_ori | i_xori | i_lw | i_lui  | i_jal | I_HAMM;//WRITE TO
                      REGISTER
47                                                                  //I_HAMM
                                                                        ALUC
                                                                        =1011
48  assign aluc[3] =  i_sra | i_sw | I_HAMM;  // complete by yourself.
49  assign aluc[2] =  i_sub | i_or | i_srl | i_sra | i_ori |i_bne |i_beq|
        i_lui;
50  assign aluc[1] =  i_xor |i_sll | i_srl | i_sra | i_xori|i_lui | I_HAMM;
51  assign aluc[0] =  i_and | i_or | i_andi| i_ori |i_sll | i_srl | i_sra |
        I_HAMM;
```

```
52    assign shift   =  i_sll | i_srl | i_sra ;
53
54    assign aluimm  =  i_addi| i_ori| i_andi|i_xori| i_lw| i_sw |i_lui;    //
          complete by yourself.
55    assign sext    =  i_addi| i_lw |i_sw |i_beq |i_bne;
56    assign wmem    =  i_sw;
57    assign m2reg   =  i_lw;
58    assign regrt   = i_addi| i_ori| i_andi|i_xori| i_lw| i_sw| i_lui;
59    assign jal     = i_jal;
60
61 endmodule
```

CU 模块完全按照真值表填写即可，控制信号对应写成 Verilog 代码如上图所示。

4.4 完整真值表

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	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	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	x 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	1	0
beq	beq rs, rt, imm	000100	rs	rt	imm			0 1	0 0 0 1	x 1 0 0	0	0	1	0	0	0	0	0
bne	bne rs, rt, imm	000101	rs	rt	imm			0 1	0 1 0 0	x 1 0 0	0	0	1	0	0	0	0	0
lui	lui rt, imm	001111	00000	rt	imm				0 0	x 1 1 0	0	1	0	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
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

图 2: 完整真值表

4.5 I/O 模块原理

I/O 设备与内存并列，通过地址的第 7 位进行判断数据属于 I/O 还是内存。

4.6 I/O 模块实现

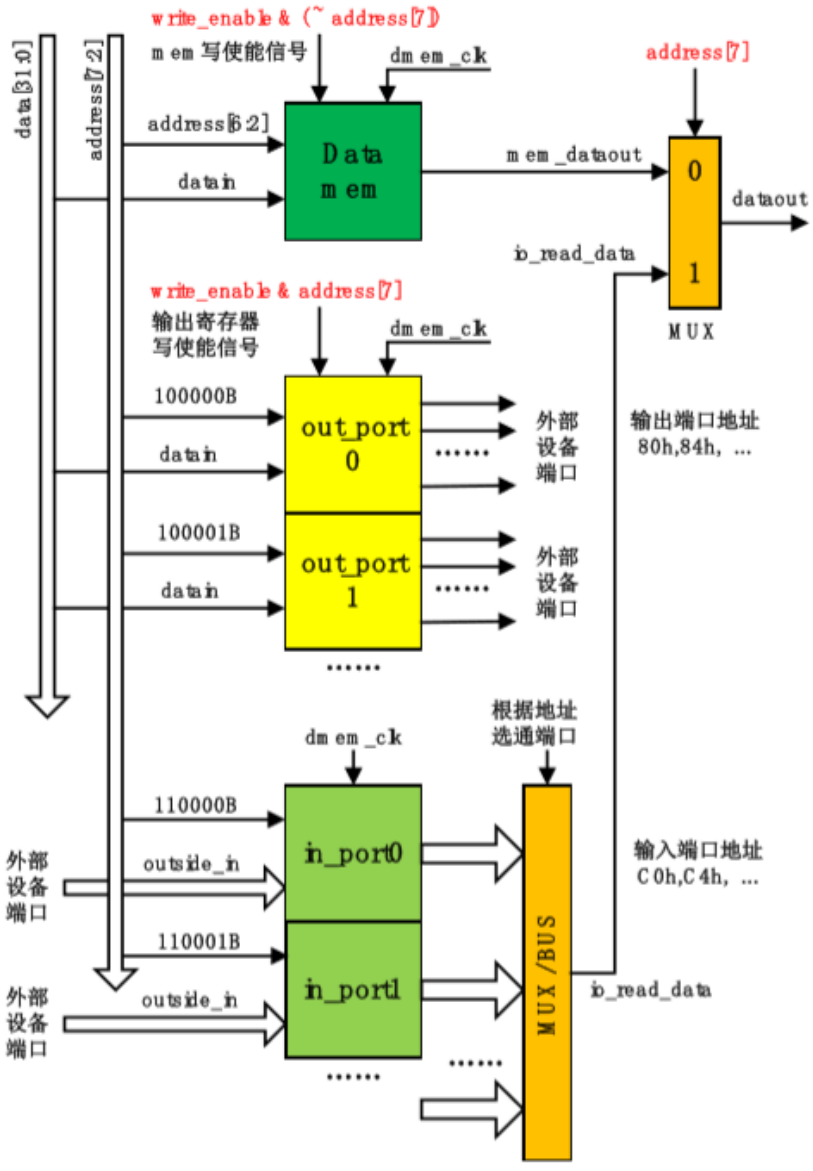
write_enable & (~ address[7])
mem 写使能信号
dmem_ck
address[7]
data[31:0]
address[7:2]
address[6:2]
Data mem
datain
mem_dataout
0
1
dataout
io_read_data
MUX
write_enable & address[7]
输出寄存器
写使能信号
dmem_ck
100000B
datain
out_port 0
外部设备端口
输出端口地址
80h,84h, ...
100001B
datain
out_port 1
外部设备端口
dmem_ck
根据地址选通端口
110000B
外部设备端口
outside_in
in_port0
输入端口地址
C0h,C4h, ...
110001B
外部设备端口
outside_in
in_port1
MUX/BUS
io_read_data

图 3: I/O 接口原理图

```
使用io\_input\_reg和io\_output\_reg可对两个I/O寄存器进行控制。
module io_input_reg (addr,io_clk,io_read_data,in_port0,in_port1);
//inport:  外部直接输入进入ioreg
    input  [31:0]  addr;
    input          io_clk;
    input  [31:0]  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 <= in_port0;   // 输入端口在 io_clk 上升沿时进行数据锁存
        in_reg1 <= 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,clrn,out_port0,
    out_port1,out_port2);

  input [31:0] addr, datain;
  input write_io_enable, io_clk;
  input clrn;
  //reset signal. if necessary,can use this signal to reset the output to 0.
  output [31:0] out_port0, out_port1, out_port2;
  reg [31:0] out_port0, out_port1, out_port2;

  always @(posedge io_clk or negedge clrn)
  begin
```

```
    if(clrn == 0)
    begin
      out_port0 <= 0;
      out_port1 <= 0;
      out_port2 <= 0;
    end
    else
    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;//88h
      endcase
    end
  end
endmodule
```

其中输出端口即最后需要烧在 FPGA 上的输出 HEX 端口。输入端口这里只指定了两个寄存器，随后会在指令中调用定义的两个寄存器。

4.7 测试指令

指令从 sc_instmem.mif 文件中调用，因为 quartus 存在类似缓存的机制，因此不同的指令文件需要使用不同的名字，文件夹里我写了 sc_instmemADD,sc_instmenSUB 等等。

这里仅拿 sc_instmenSUB 作为展示。

```
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;        % (00)       addi $1,$0,192 #  %
1 : 20020080;        % (04)       addi $2,$0,128 #  %
2 : 8c230000;        % (08) loop: lw $3,0($1)    #  %
3 : 8c240004;        % (0c)       lw $4,4($1)    #  %
4 : ac430000;        % (10)       sw $3,0($2)    #  %
5 : ac440004;        % (14)       sw $4,4($2)    #  %
6 : 00642822;        % (18)       sub $5,$3,$4   #  %
7 : ac450008;        % (1c)       sw $5,8($2)    #  %
8 : 08000000;        % (20)       j loop         #  %
END ;
```

其中 192 和 128 号寄存器即是之前 input_reg 指定的输入端口。

5 实验总结

这次实验是完成了我小时候的愿望之一，就是彻底了解了为什么计算机可以自动工作，同时过程中也遇到了很多困难。

首先，更改指令无效非常头疼。更改了 mif 文件按理说重新编译会读入新的指令，可是后来发现 quartus 有类似 cache 的机制，即同一个文件读取一次之后，第二次 quartus 会自动判断，如果同名那么就不再读取一次。因此每次读取时都要更改 mif 的文件名，非常恶心。

其次，很多指令的调用关系较为复杂，需要慢慢耐下心来研究，比如 pc 调用指令的时候明明已经 PC+4 了为何还要从倒数第二位开始读取，因为 PC 最后两位永远是 0，读取浪费了时间降低了速度。再比如为什么读数据是直接从 addr 的倒数第 3 位开始取，因为数据宽度为 32，因此后 2 个 bit 仅表示每一位里的偏移量，不做考虑。

最后，CPU 真的非常神奇，制作单周期 CPU 的过程也相对困难，希望之后继续努力，对 Verilog 有更深的了解！

517030910374 郭嘉宋