上海交通大學

《计算机组成原理》课程

学生实验报告

实验名称：课程大作业 CPU 设计

姓　　名：王 玫

学　　号：5120309691

班　　级：F1203026

手　　机：18818212442

邮　　箱：594189534@qq.com

同组同学：None

任课老师：方向忠教授

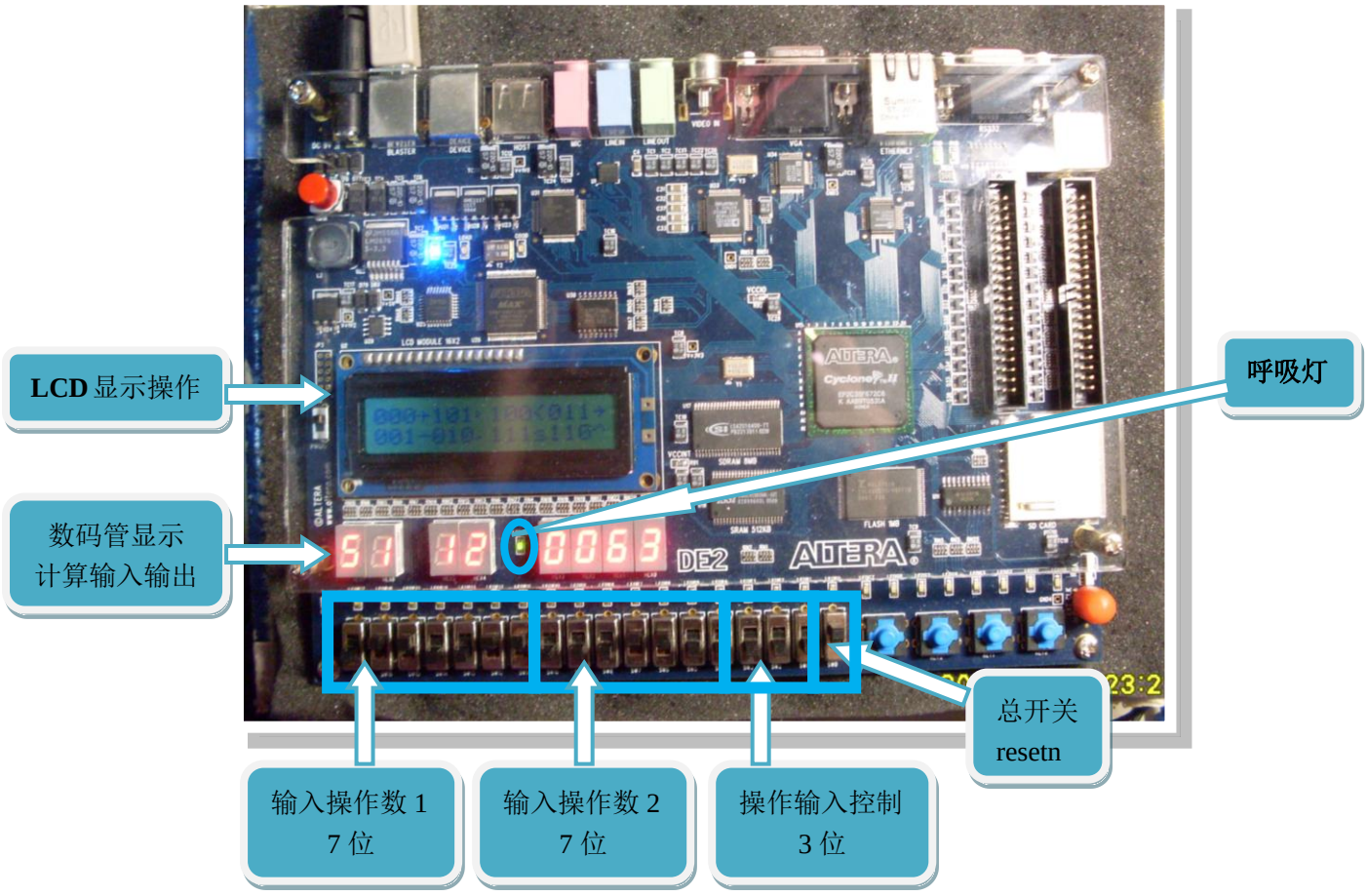
2014 年 7 月 2 日

摘要

本次实验作为计算机组成原理课程的最后大作业，我在前面完成的流水线 cpu 上增加了很多其他的 **MIPS 指令**，然后通过 mif 文件编程完成了一个具有加、减、乘、除、移位、比较、异或和或非 **8 个功能简单计算器**。其中，14 个开关输入两个 7 位运算操作符，一个 3 位计算操作码控制计算器运算输出结果，**七段数码管**显示十进制输入输出，同时在 **LCD** 液晶屏上显示的八个计算操作的操作编码，将计算器的功能与输入显示出来。最后，还有一盏 **LED 呼吸灯**的设计，给开发板增添了生命力。

关键词：指令补充 ；8 功能计算器；数码管；LCD 显示屏； LED 呼吸灯

PS：代码中阴影部分为新加的或重点实现部分



目录

1 实验目的.....4
2 实验原理.....4
2.1 硬件原理图设计.....4
2.2 软件算法分析.....6
Expended_Pipelined_CPU.....6
五段流水线.....6
流水线寄存器.....6
show 数码管.....6
2.2.1 顶层文件分析：.....6
2.2.2 部分 MIPS 指令补充.....7
2.2.3 输入输出数码管十进制显示.....9
2.2.4 指令 instmem_mif 文件.....10
2.2.5 控制输入信号的添加.....11
2.2.6 LCD 显示屏.....12
2.2.7 LED 呼吸灯.....14
2.2.8 仿真 sc_computer_sim.v.....15
3 实验步骤.....16
3.1 步骤一：理解流水线 CPU 工作原理，IO 扩展知识，熟练运用.....16
3.2 步骤二：MIPS 指令的添加补充.....17
3.3 步骤三：MIPS 指令编译 MIF 文件导入 imem.....17
3.4 步骤四：数码管显示及增加输入内存地址.....18
3.5 步骤五：补充实现扩展，LCD 显示操作提示，LED 呼吸灯。.....18
3.5 步骤六：编译及 debug.....19
3.7 步骤七：仿真.....19
4. 实验结果.....20
5. 感想和建议.....21

注：

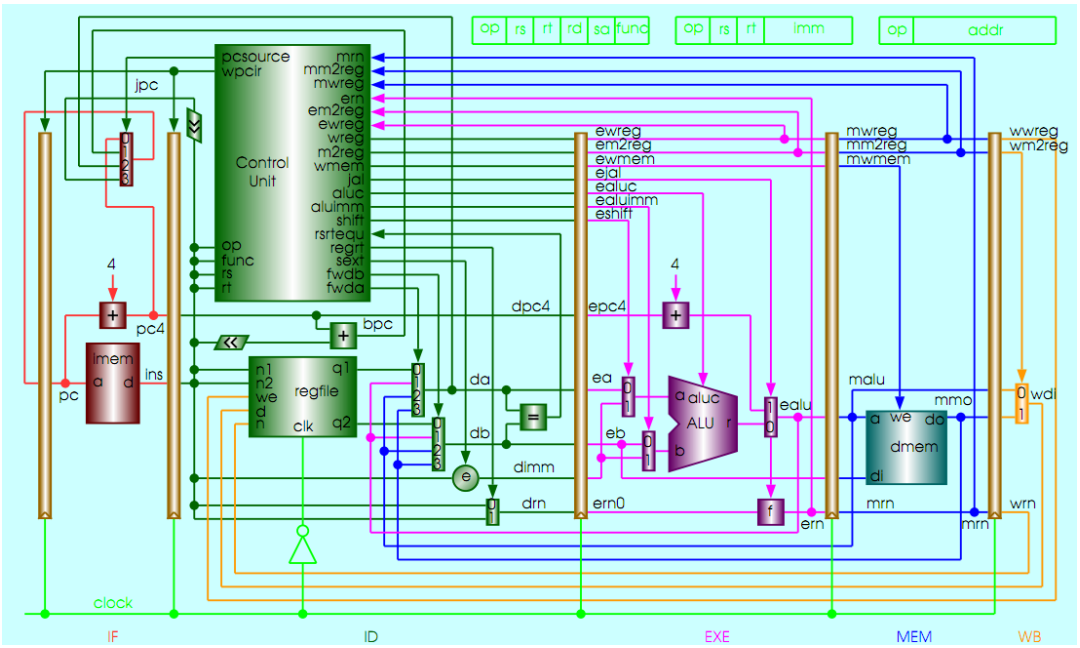
1. 硬件可以加入原理图，必要的话可做标记。
2. 软件设计需要画出流程图，并附加相应的源代码；
3. 软件源代码要求加入注解，便于阅读。
4. 实验结果分析可以结合实验的体会做深入的讨论。

1 实验目的

1. 在前几次实验基础上丰富 cpu 指令，完善其功能，并提高 cpu 性能。
2. 利用自己设计的 cpu，设计完成一项扩展功能，即八操作简易计算器。
3. 尝试开发 DE2 开发板上其他 I/O 的利用，LCD 显示及呼吸灯。
4. 综合表现出研发 cpu 单片机的能力，掌握计算机组成的原理。

2 实验原理

2.1 硬件原理图设计



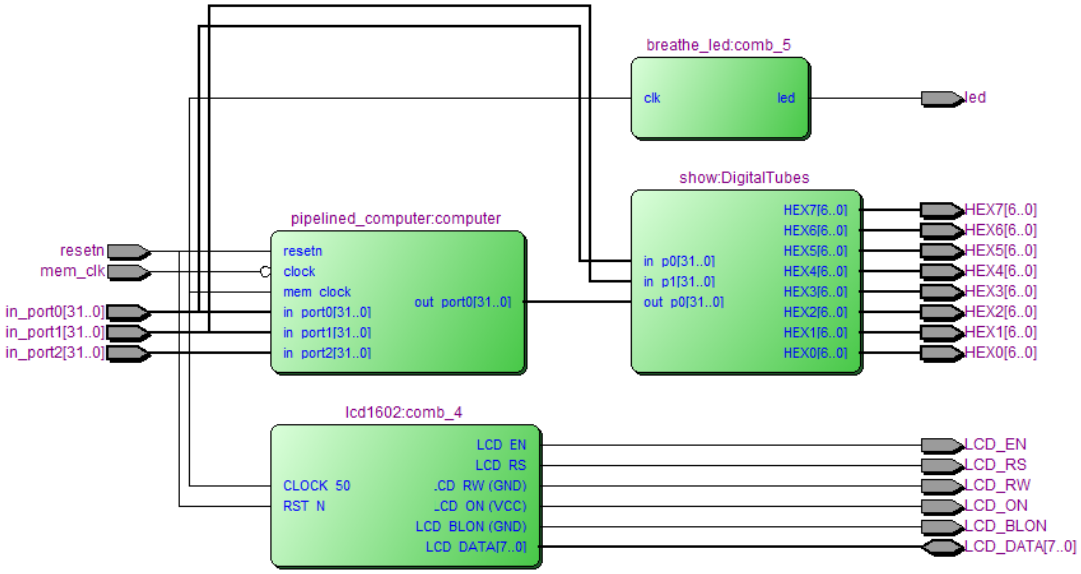
实验硬件器材：Altera-DE2 实验板套件

硬件结构设计：流水线 pipelined_computer 由 IF / ID / EXE / MEM / WB 五个阶段模块+各段之间的流水线寄存器模块组成。

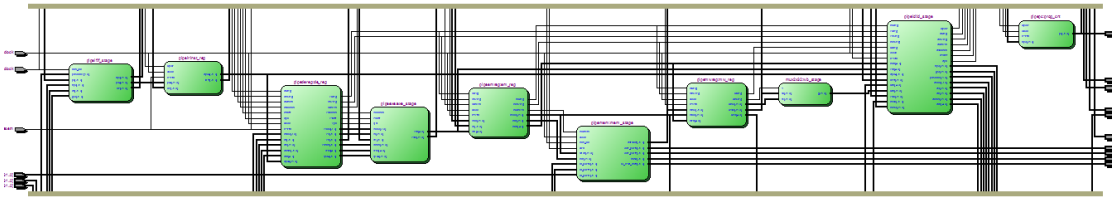
应用扩展设计：

1. 具有加、减、乘、除、移位、比较、异或和或非 8 个功能的简易计算器；
2. LCD 字符显示；
3. 呼吸灯控制。

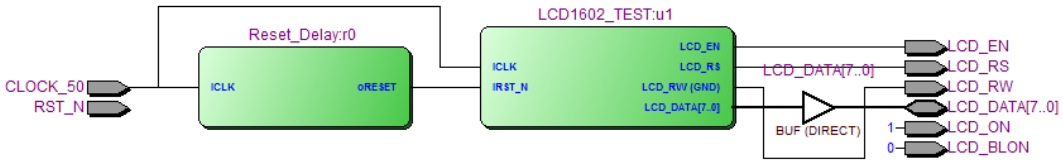
pipelinedcpu_top RTL Viewer:



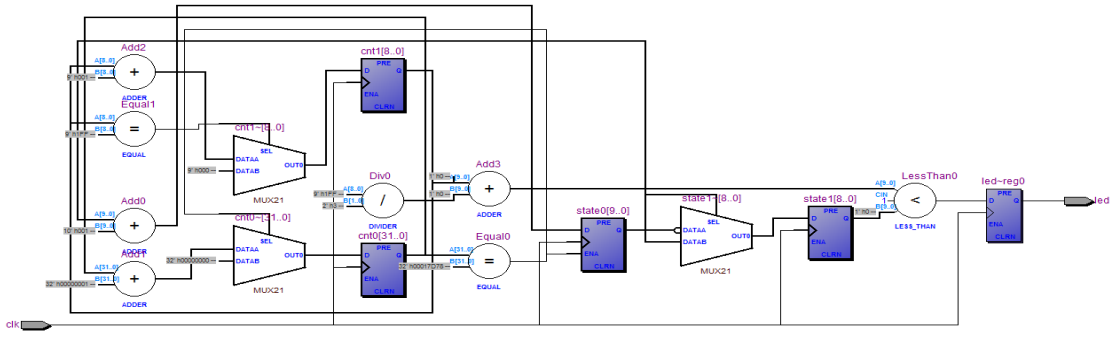
Pipelined CPU RTL Viewer：



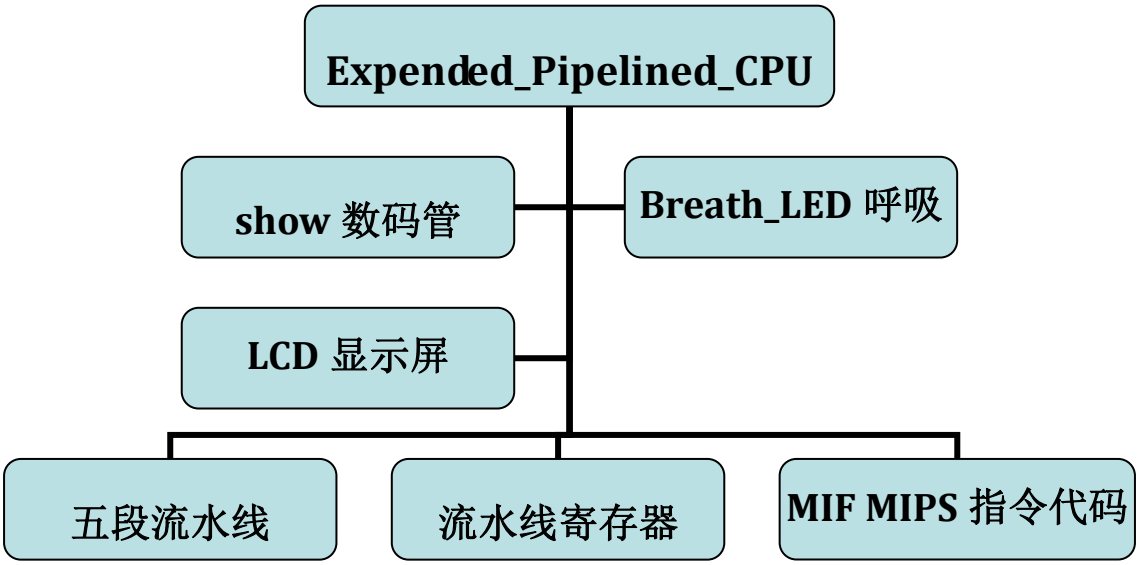
LCD Model RTL Viewer:



Breath-LED Model RTL Viewer



2.2 软件算法分析



2.2.1 顶层文件分析：

```
module pipelinecpu_top (HEX7,HEX6,HEX5,HEX4,HEX3,HEX2,HEX1,HEX0,
                        resetn,mem_clk,in_port0,in_port1,in_port2,
            LCD_DATA,LCD_EN,LCD_RS,LCD_RW,LCD_ON,LCD_BLON,led);

    input resetn,mem_clk;
    input   [31:0]   in_port0,in_port1,in_port2;
    output [6:0] HEX7,HEX6,HEX5,HEX4,HEX3,HEX2,HEX1,HEX0;

    inout[7:0] LCD_DATA;
    output LCD_EN, LCD_RS, LCD_RW, LCD_ON, LCD_BLON;
    output led;

    wire clock,wmem;
    wire [31:0] pc,ins,inst,ealu,malu,walu,memout,data,mem_dataout,io_read_data;
    wire [31:0] out_port0,out_port1;

    assign clock = ~mem_clk;

    pipelined_computer computer(resetn,clock,mem_clk,pc,ins,inst,ealu,malu,walu,
                out_port0,out_port1,in_port0,in_port1,in_port2,mem_dataout,io_read_data);

    show
DigitalTubes(in_port0,in_port1,out_port0,HEX7,HEX6,HEX5,HEX4,HEX3,HEX2,HEX1,H
EX0);

    lcd1602(mem_clk,resetn,LCD_DATA,LCD_EN,LCD_RS,LCD_RW,LCD_ON,LCD_BL
ON);

    breathe_led(mem_clk,led);

endmodule
```

2.2.2 部分 MIPS 指令补充

MIPS 精简指令架构中共 31 条基本指令，之前的流水线 CPU 实现了其中的最基本的 20 条，我补充了基本指令 slt sllv 以及伪指令 mul div nor nop，还有无符号的实现 addu subu addiu，这里先不赘述。由此，我的 CPU 设计基本实现了 MIPS 的所有精简指令集。

在下图中为指令格式及控制信号真值表的填充实现，根据它完成对 Project 的 pipe_cu.v 及 alu.v 的填充完善，即可顺利实现。此外，为实现无符号数的操作，需要在模块间增加一个变量 duext，euext 传递参数。

输入

指令	指令格式 R	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	0 0 0 0	0	0	x	0	1	0	0	0
addu	addu $1,$2,$3	0	rs	rt	rd	0	100001	x	0 0		0	0	x	0	1	0	0	0
sub	sub rd, rs, rt	000000	rs	rt	rd	00000	100010	x	0 0	0 1 0 0	0	0	x	0	1	0	0	0
subu	subu $1,$2,$3	0	rs	rt	rd	0	100011	x	0 0		0	0	x	0	1	0	0	0
mul	mul rd,rs,rt	000000	rs	rt	rd	0	110001	x	0 0	1 1 0 1	0	0	x	0	1	0	0	0
div	div rd,rs,rt	000000	rs	rt	rd	0	110011	x	0 0	1 1 1 0	0	0	x	0	1	0	0	0
and	and rd, rs, rt	000000	rs	rt	rd	00000	100100	x	0 0	0 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	0 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	0 0 1 0	0	0	x	0	1	0	0	0
nor	nor $1,$2,$3	0	rs	rt	rd	0	100111	x	0 0	1 0 0 1	0	0	x	0	1	0	0	0
slt	slt $1,$2,$3	0	rs	rt	rd	0	101010	x	0 0	1 0 1 0	0	0	x	0	1	0	0	0
sltu	sltu $1,$2,$3	0	rs	rt	rd	0	101011	x	0 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	0	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	0	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	0	0	1	0	0	0
sllv	sllv $1,$2,$3	0	rs	rt	rd	0	000100	x	0 0	0 0 1 1	0	0	0	0	1	0	0	0
srlv	srlv $1,$2,$3	0	rs	rt	rd	0	000110	x	0 0	0 1 1 1	0	0	0	0	1	0	0	0
srav	srav $1,$2,$3	0	rs	rt	rd	0	000111	x	0 0	1 1 1 1	0	0	0	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 I	op	rs	rt	imm	pcsource [1..0]	aluc [3..0]	shift	aluimm	sext	wmem	wreg	m2reg	regrt	call jal
addi	addi rt, rs, imm	001000	rs	rt	imm	x	0 0	0 0 0 0	0	1	1	0	1	0	1	0
addiu	addiu $1,$2,100	001001	rs	rt	imm	x	0 0		0	1	1					
andi	andi rt, rs, imm	001100	rs	rt	imm	x	0 0	0 0 0 1	0	1	0	0	1	0	1	0
ori	ori rt, rs, imm	001101	rs	rt	imm	x	0 0	0 1 0 1	0	1	0	0	1	0	1	0
xori	xori rt, rs, imm	001110	rs	rt	imm	x	0 0	0 0 1 0	0	1	0	0	1	0	1	0
lw	lw rt, imm(rs)	100011	rs	rt	imm	x	0 0	0 0 0 0	0	1	1	0	1	1	1	0
sw	sw rt, imm(rs)	101011	rs	rt	imm	x	0 0	0 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 x x x	0	0	1	0	0	x	x	x
bne	bne rs, rt, imm	000101	rs	rt	imm	0 1	0 1 0 0	x x x x	0	0	1	0	0	x	x	x
lui	lui rt, imm	001111	00000	rt	imm	x	0 0	0 1 1 0	0	1	1	0	1	0	1	0
slti	slti $1,$2,10	001010	rs	rt	mediate	x	0 0	1 0 1 0	0	1	1	0	1	0	1	0
sltiu	sltiu $1,$2,10	001011	rs	rt	mediate	x	0 0									

指令	指令格式 J	op	addr	pcsource [1..0]	aluc [3..0]	shift	aluimm	sext	wmem	wreg	m2reg	regrt	call jal
j	j addr	000010	addr	x	1 1	x x x x	x	x	x	0	0	x	x	x
jal	jal addr	000011	addr	x	1 1	x x x x	x	x	x	0	1	x	x	1

【pipe_cu.v】

```
module pipe_cu ( op,func,rs,rt,rsrtequ,dwreg,dm2reg,dwmem,daluc,daluimm,dshift,djal,
regrt,sext, fwda,fwdb,mrn,mm2reg,mwreg,ern,em2reg,ewreg,pcsource,wpcir,clock );
.....
.....
        wire i_mul = r_type &func[5] & func[4] & ~func[3] & ~func[2] & ~func[1] & func[0];
//110001 mul
        wire i_div = r_type & func[5] & func[4] & ~func[3] & ~func[2] & func[1] & func[0];
//110011 div
        wire i_nor = r_type & func[5] & ~func[4] & ~func[3] & func[2] & func[1] & func[0];
//100111 nor
        wire i_slt = r_type & func[5] & ~func[4] & func[3] & ~func[2] & func[1] & ~func[0];
//101010 slt
        wire i_nop = r_type & ~func[5] & ~func[4] &    func[3] & func[2] & func[1] & func[0];
//001111 nop
wire i_addu = r_type & func[5] & ~func[4] & ~func[3] & ~func[2] & ~func[1] &    func[0];
//100001 addu
wire i_subu = r_type & func[5] & ~func[4] & ~func[3] & ~func[2] &    func[1] &    func[0];
//100011 subu

    wire i_rs    = i_add | i_sub | i_and | i_or | i_xor | i_jr | i_addi | i_andi |
                   i_ori | i_xori | i_lw | i_sw | i_beq | i_bne | i_mul | i_div | i_nor | i_slt | i_sllv
                   | i_addu | i_subu | i_addiu;
    wire i_rt    = i_add | i_sub | i_and | i_or | i_xor | i_sll | i_srl | i_sra |
            i_sw | i_beq | i_bne | i_mul | i_div | i_nor | i_slt | i_sll | i_addu | i_subu | i_addiuv;

    assign daluc[3] =    i_sra | i_mul | i_div | i_nor | i_slt;
    assign daluc[2] =    i_sub | i_or | i_srl | i_sra | i_ori | i_lui | i_mul | i_div |i_subu;
    assign daluc[1] =    i_xor | i_sll | i_srl | i_sra | i_xori | i_lui | i_div | i_slt | i_sllv;
    assign daluc[0] =    i_and | i_or | i_sll | i_srl | i_sra | i_andi | i_ori | i_lui | i_mul | i_nor |
i_sllv;
    assign dshift      = i_sll | i_srl | i_sra ;
    assign daluimm     = i_addi | i_andi | i_ori | i_xori |i_lw | i_sw | i_lui | i_addiu;
    assign sext        = i_addi | i_lw | i_sw | i_beq | i_bne ;
    assign duext       =    i_addu | i_subu | i_addiu ;
    assign dm2reg      = i_lw ;
    assign regrt       = i_addi |i_andi | i_ori | i_xori |i_lw | i_lui | i_addiu;
    assign djal        = i_jal;
    assign dwreg = (i_add | i_sub | i_and | i_or | i_xor | i_sll | i_addu | i_subu | i_addiu |
                    i_srl | i_sra | i_addi | i_andi |    i_ori | i_xori |
                    i_lw | i_lui | i_jal | i_mul | i_div | i_nor | i_slt | i_sllv) & wpcir;
```

【alu.v】

```
module alu (a,b,aluc4,s,z);

    casex (aluc4)
                4'b0000: s = a + b;                    //0000 ADD
                4'b0100: s = a - b;                   //0100 SUB
                4'b0001: s = a & b;                    //0001 AND
                4'b0101: s = a | b;                   //0101 OR
                4'b0010: s = a ^ b;                    //0010 XOR
                4'b0110: s = b << 32'h0F;              //0110 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)
                4'b1101: s = a * b;                   //1101 MUL
                4'b1110: s = a / b;                   //1110 DIV
                4'b1001: s = ~ (a | b);               //1001 NOR
                4'b1010: s = (a < b)? 32'b1:32'b0; //1010 SLT
                 default: s = 0;
            endcase
            if (s == 0 )    z = 1;
                else z = 0;
```

2.2.3 输入输出数码管十进制显示

前面的实验主要重点放在单周期 CPU 及流水线的实现上，数码管的表示采用的十六进制，本次完善为十进制表示，每个输入 7 位开关控制，由两个数码管表示，输出有四个数码管表示。

由阴影部分可以看到，直接采用除以十模十的方法即可快速得到十进制每一位数，从而转化为十进制数表示输入操作数及输出结果。

【present.v】

```
Dmodule show(
        input [31:0] in_p0,
        input [31:0] in_p1,
        input [31:0] out_p0,
        output [6:0] HEX7,HEX6,HEX5,HEX4,HEX3,HEX2,HEX1,HEX0
);

        present Tube7(.in((in_p0[6:0] / 'd10) % 'd10), .out_wire(HEX7));
        present Tube6(.in(in_p0[6:0] % 'd10), .out_wire(HEX6));
        present Tube5(.in((in_p1[6:0] / 'd10) % 'd10), .out_wire(HEX5));
        present Tube4(.in(in_p1[6:0] % 'd10), .out_wire(HEX4));
        present Tube3(.in((out_p0 / 'd1000) % 'd10), .out_wire(HEX3));
        present Tube2(.in((out_p0 / 'd100) % 'd10), .out_wire(HEX2));
        present Tube1(.in((out_p0 / 'd10) % 'd10), .out_wire(HEX1));
        present Tube0(.in(out_p0 % 'd10), .out_wire(HEX0));

endmodule


module present(
        input      [3:0] in,
        output    [6:0] out_wire
);
reg [6:0]out;
assign out_wire = out;

always @(*)
begin
  case(in)
    'h0: out = 7'b1000000;
    'h1: out = 7'b1111001;
    'h2: out = 7'b0100100;
    'h3: out = 7'b0110000;
    'h4: out = 7'b0011001;
    'h5: out = 7'b0010010;
    'h6: out = 7'b0000010;
    'h7: out = 7'b1111000;
    'h8: out = 7'b0000000;
    'h9: out = 7'b0010000;
    default: out = 7'b0;
  endcase
end

endmodule
```

2.2.4 指令 instmem_mif 文件

为实现八个运算的控制，汇编指令用 bne 实现 switch 功能，根据控制进行运算和输出。

```
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 : 20020080;   % addi $2, $0, 10000000b          # address 80h %
1 : 200300c0;   % addi $3, $0, 11000000b          # address c0h %
2 : 200400c4;   % addi $4, $0, 11000100b          # address c4h %
3 : 200500c8;   % addi $5, $0, 11001000b          # address c8h %

4 : 8c660000;   % lw $6, 0($3)                    # input data from [c0h] %
5 : 8c870000;   % lw $7, 0($4)                    # input data from [c4h] %
6 : 8ca80000;   % lw $8, 0($5)                    # input data from [c8h] %

7 : 15000003;   % bne $8, $0,OP0-+                # in_port2 beq lw + %
8 : 00c74820;   % add $9, $6, $7                  # add input data %
9 : ac490000;   % sw $9, 0($2)                    # output to [80h] %
a : 08000004;   % j loop                          # loop %

b : 20110001;   % addi $17, $0, 1                  # $17=1 %
c : 15110003;   % beq $8, $17,OP0--               # in_port2 beq lw - %
d : 00c75022;   % sub $10, $6, $7                 # sub input data %
e : ac4a0000;   % sw $10, 0($2)                   # output to [80h] %
f : 08000004;   % j loop                          # loop %

10: 22310001;  % addi $17, $17, 1                 # $17=2 %
11: 15110003;  % beq $8, $17,OP0-*                # in_port2 beq lw * %
12: 00c75831;  % mul $11, $6, $7                  # mul input data %
13: ac4b0000;  % sw $11, 0($2)                    # output to [80h] %
14: 08000004; % j loop                           # loop %

15: 22310001;  % addi $17, $17, 1                 # $17=3 %
16: 15110003;  % beq $8, $17,OP0-/                # in_port2 beq lw / %
17: 00c76033;  % div $12, $6, $7                  # div input data %
18: ac4c0000;  % sw $12, 0($2)                    # output to [80h] %
19: 08000004; % j loop                           # loop %

1a: 22310001;  % addi $17, $17, 1                 # $17=4 %
1b: 15110003;  % beq $8, $17,OP0-xor              # in_port2 beq lw xor %
1c: 00c76826;  % xor $13, $6, $7                  # xor input data %
1d: ac4d0000;  % sw $13, 0($2)                    # output to [80h] %
1e: 08000004;  % j loop                          # loop %

1f: 22310001;  % addi $17, $17, 1                 # $17=5 %
20: 15110003;  % beq $8, $17,OP0-~                # in_port2 beq lw ~ %
21: 00c77027;  % nor $14, $6, $7                  # nor input data %
22: ac4e0000;  % sw $14, 0($2)                    # output to [80h] %
23: 08000004; % j loop                           # loop %

24: 22310001;  % addi $17, $17, 1                 # $17=6 %
25: 15110003;  % beq $8, $17,OP0-<                # in_port2 beq lw < %
26: 00c7782a;  % slt $15, $6, $7                  # slt input data %
27: ac4f0000;  % sw $15, 0($2)                    # output to [80h] %
28: 08000004; % j loop                           # loop %

29: 22310001;  % addi $17, $17, 1                 # $17=7 %
2a: 15110002;  % beq $8, $17,OP0-<<               # in_port2 beq lw << %
2b: 00c78004;  % sllv $16, $6, $7                 # sllv input data %
2c: ac500000;  % sw $16, 0($2)                    # output to [80h] %
2d: 08000004; % j loop                           # loop %
END ;
```

2.2.5 控制输入信号的添加

我们采用的是统一编址方式实现 I/O 端口扩展，现在需要三个输入（两个输入值，一个控制计算值），所以需要对内存编码进行补充。

【io_input_reg.v】

```
module io_input_reg (addr,io_clk,io_read_data,in_port0,in_port1,in_port2);
input   [31:0]     addr;
input              io_clk;
input   [31:0]     in_port0,in_port1,in_port2;
output [31:0]     io_read_data;
reg   [31:0]     in_reg0;          // input port0
reg   [31:0]     in_reg1;          // input port1
reg   [31:0]       in_reg2;
io_input_mux io_imput_mux2x32(in_reg0,in_reg1,in_reg2,addr[7:2],io_read_data);
always @(posedge io_clk)
begin
in_reg0 <= {25'b0,in_port0[6:0]};   //   输入端口在 io_clk 上升沿时进行数据锁存
in_reg1 <= {25'b0,in_port1[6:0]};     //   输入端口在 io_clk 上升沿时进行数据锁存
in_reg2 <= {29'b0,in_port2[2:0]};
// more ports，可根据需要设计更多的输入端口。
end
endmodule
module io_input_mux(a0,a1,a2,sel_addr,y);
input     [31:0]     a0,a1,a2;
input     [ 5:0]     sel_addr;
output   [31:0]     y;
reg       [31:0]     y;
always @ *
case (sel_addr)
6'b110000: y = a0;//c0h
6'b110001: y = a1;//c4h
6'b110010: y = a2;//c8h
// more ports，可根据需要设计更多的端口。
endcase
endmodule
```

此外，关于初值的影响问题。由于板子开关只有 18 个，而每个 in_port 都是 32 位，这样存在没有 pin 的位容易给输入带上初值而导致错误和影响，为避免这一问题，我在赋值时将前面高位设为零，则不再出现初值问题的影响。如下所示：

【io_input_reg.v】

```
module io_input_reg (addr,io_clk,io_read_data,in_port0,in_port1,in_port2);

always @(posedge io_clk)
begin
in_reg0 <= {25'b0,in_port0[6:0]};   //   输入端口在 io_clk 上升沿时进行数据锁存
in_reg1 <= {25'b0,in_port1[6:0]};     //   输入端口在 io_clk 上升沿时进行数据锁存
in_reg2 <= {29'b0,in_port2[2:0]};
```

2.2.6 LCD 显示屏

LCD 显示由顶层模块 LED1602，初始延迟部分，和底层控制、底层设计代码实现。

```
module lcd1602(
                    input CLOCK_50,
                    input RST_N,
                    inout[7:0] LCD_DATA,
                    output LCD_EN,
                    output LCD_RS,
                    output LCD_RW,
                    output LCD_ON,
                    output LCD_BLON
                    );
                  //initial//
                    wire[7:0] LCD_D_1;
                    wire LCD_RS_1;
                    wire LCD_RW_1;
                    wire LCD_EN_1;
                    wire DLY_RST;

        assign LCD_DATA=LCD_D_1;
        assign LCD_RS=LCD_RS_1;
        assign LCD_RW=LCD_RW_1;
        assign LCD_EN=LCD_EN_1;
        assign LCD_ON=1'b1;
        assign LCD_BLON=1'b0;


        Reset_Delay r0(.iCLK(CLOCK_50),
.               oRESET(DLY_RST));

        LCD1602_TEST u1(//Host Side
          .iCLK(CLOCK_50),
          .iRST_N(DLY_RST),

//LCD Side
          .LCD_DATA(LCD_D_1),
          .LCD_RW(LCD_RW_1),
          .LCD_EN(LCD_EN_1),
          .LCD_RS(LCD_RS_1)
          );
endmodule



//RSTN 初始延迟代码
module Reset_Delay(iCLK,oRESET);
input iCLK;
output reg oRESET;
reg[19:0] Cont;

always@(posedge iCLK)
begin
        if(Cont!=20'hfffff)
                begin
                Cont<=Cont+1'b1;
                oRESET<=1'b0;
                end
        else
        oRESET<=1'b1;
end
endmodule
```

```
//底层代码 LCD1602_TEST：
module LCD1602_TEST(
input iCLK,iRST_N,//Host Side
output[7:0] LCD_DATA, //LCD1602 Side
output LCD_RS,LCD_RW,LCD_EN);
//Internal Wires/Registers
reg[5:0] LUT_INDEX;
reg[8:0] LUT_DATA;
reg[5:0] mLCD_ST;
reg[17:0] mDLY;
reg[7:0] mLCD_DATA;
reg mLCD_Start;
reg mLCD_RS;
wire mLCD_Done;
parameter LCD_INITIAL=0;
parameter LCD_LINE1    =5;
                        //Just have 5 control command
parameter CD_CH_LINE=LCD_LINE1+16;
                                        //Change Line
parameter LCD_LINE2=LCD_LINE1+16+1;
parameter LUT_SIZE=LCD_LINE1+32+1;

always@(posedge iCLK or negedge iRST_N)
begin
        if(!iRST_N)
                begin
                        LUT_INDEX<=0;
                        mLCD_ST<=0;
                        mDLY<=0;
                        mLCD_Start<=0;
                        mLCD_DATA<=0;
                        mLCD_RS<=0;
                end
        else
                begin
                if(LUT_INDEX<LUT_SIZE)
                begin
case(mLCD_ST)
        0:    begin
        mLCD_DATA<=LUT_DATA[7:0];
        mLCD_RS<=LUT_DATA[8];
        mLCD_Start<=1;
        mLCD_ST<=1;
        end
        1:    begin
        if(mLCD_Done)
                begin
                        mLCD_Start<=0;
                        mLCD_ST<=2;
                end  end
        2:   begin
        if(mDLY<18'h3ffff)
                        mDLY<=mDLY+18'b1;
        else
                begin
                mDLY<=0;
                mLCD_ST<=3;
                end  end
        3:   begin
        LUT_INDEX<=LUT_INDEX+6'b1;
        mLCD_ST<=0;
                end
endcase
```

```
//底层代码 LCD1602_Controller：
module LCD1602_Controller(
	//Host Side
	input iCLK,iRST_N,
	input iRS,iStart,
	input[7:0] iDATA,
	output reg oDone,
	//LCD1602 Interface
	output[7:0] LCD_DATA,
	output LCD_RS,
	output LCD_RW,
	output reg LCD_EN
	);
//Internal Register
reg[4:0] Cont;
reg[1:0] ST;
reg preStart,mStart;
//Only write to LCD,bypass iRS to LCD_RS
assign LCD_DATA=iDATA;
assign LCD_RS=iRS;
assign LCD_RW=1'b0;

parameter CLK_Divide=16;

always@(posedge iCLK or negedge iRST_N)
begin
	if(!iRST_N)
	 begin
			oDone<=1'b0;
			LCD_EN<=1'b0;
			preStart<=1'b0;
			mStart<=1'b0;
			Cont<=0;
			ST<=0;
	 end
	else
		begin
		//Input Start Detect
		preStart <= iStart;
		if({preStart,iStart}==2'b01)
		begin
			mStart<=1'b1;
			oDone<=1'b0;
		end
		if(mStart)
		begin
			case(ST)
				0:ST<=1;//Wait Setup
	1:begin
		LCD_EN<=1'b1;
		ST<=2;
		end
	2:begin
		if(Cont<CLK_Divide)
			Cont <= Cont+5'b1;
		else
			ST<=3;
		end
	3:begin
		LCD_EN<=1'b0;
		mStart<=1'b0;
		oDone<=1'b1;
		Cont<=0;
		ST<=0;
	end
endcase
	end  end   end
endmodule
```

```
end end end

always
begin
	case(LUT_INDEX)
		//Inital

	LCD_INITIAL+0:LUT_DATA<=9'h038
;//设置 16x2 显示，5x7 点阵，8 位数据接口
	LCD_INITIAL+1:LUT_DATA<=9'h00
C;//设置开显示，不显示光标
	LCD_INITIAL+2:LUT_DATA<=9'h001
;//显示清零，数据指针清零
	LCD_INITIAL+3:LUT_DATA<=9'h006
;//写一个字符后地址指针加一
	LCD_INITIAL+4:LUT_DATA<=9'h080
;//Line1 First Address
//   Line1
LCD_LINE1+0:LUT_DATA<=9'h130; //0
LCD_LINE1+1:LUT_DATA<=9'h130; //0
LCD_LINE1+2:LUT_DATA<=9'h130; //0
LCD_LINE1+3:LUT_DATA<=9'h12b; //+
LCD_LINE1+4:LUT_DATA<=9'h131; //1
LCD_LINE1+5:LUT_DATA<=9'h130; //0
LCD_LINE1+6:LUT_DATA<=9'h131; //1
LCD_LINE1+7:LUT_DATA<=9'h12A; //*
LCD_LINE1+8:LUT_DATA<=9'h131; //1
LCD_LINE1+9:LUT_DATA<=9'h130; //0
LCD_LINE1+10:LUT_DATA<=9'h130; //0
LCD_LINE1+11:LUT_DATA<=9'h13C; //<
LCD_LINE1+12:LUT_DATA<=9'h130; //0
LCD_LINE1+13:LUT_DATA<=9'h131; //1
LCD_LINE1+14:LUT_DATA<=9'h131; //1
LCD_LINE1+15:LUT_DATA<=9'h17E; //~
//Change Line
LCD_CH_LINE:LUT_DATA<=9'h0C0;
//Line2
LCD_LINE2+0:LUT_DATA<=9'h130; //0
LCD_LINE2+1:LUT_DATA<=9'h130; //0
LCD_LINE2+2:LUT_DATA<=9'h131; //1
LCD_LINE2+3:LUT_DATA<=9'h12D; //-
LCD_LINE2+4:LUT_DATA<=9'h130; //0
LCD_LINE2+5:LUT_DATA<=9'h131; //1
LCD_LINE2+6:LUT_DATA<=9'h130; //0
LCD_LINE2+7:LUT_DATA<=9'h12F; ///
LCD_LINE2+8:LUT_DATA<=9'h131; //1
LCD_LINE2+9:LUT_DATA<=9'h131; //1
LCD_LINE2+10:LUT_DATA<=9'h131; //1
LCD_LINE2+11:LUT_DATA<=9'h173; //s
LCD_LINE2+12:LUT_DATA<=9'h131; //1
LCD_LINE2+13:LUT_DATA<=9'h131; //1
LCD_LINE2+14:LUT_DATA<=9'h130; //0
LCD_LINE2+15:LUT_DATA<=9'h15E; //^
default:LUT_DATA<=9'h120;
endcase
end
LCD1602_Controller u0 (//Host Side
		.iDATA(mLCD_DATA),
		.iRS(mLCD_RS),
		.iStart(mLCD_Start),
		.oDone(mLCD_Done),
		.iCLK(iCLK),
		.iRST_N(iRST_N),
		//LCD1602 Interface
		.LCD_DATA(LCD_DATA),
		.LCD_RW(LCD_RW),
		.LCD_EN(LCD_EN),
		.LCD_RS(LCD_RS)	);
endmodule
```

2.2.7 LED 呼吸灯

LED 灯光在 CPU 控制之下完成由亮到暗的周期性逐渐变化，详细注释写在代码中。

```
module breathe_led(
        input clk,
        output reg led);
        parameter FREQUENCE=50_000_000;
        parameter    WIDTH=9;
        reg [WIDTH:0] state0;
        reg [WIDTH-1:0] state1;

        //=============================================
        //控制每个占空比的持续时间
        //=============================================
        reg [31:0] cnt0;
        always @ (posedge clk)
        begin
                if(cnt0==(FREQUENCE/(2**WIDTH)))
                        begin       cnt0<=0;
                                    state0<=state0+1'b1;        end
                else
                        begin       cnt0<=cnt0+1'b1;            end
        end

        //=============================================
        //控制占空比增大与减小
        //=============================================
        always @ (posedge clk)
                begin
                        if(state0[WIDTH])
                                state1<=state0[WIDTH-1:0];
                        else
                                state1<=~state0[WIDTH-1:0];
                        end

        //=============================================
        //生成与 state1 进行大小比较的计数器 cnt1
        //=============================================
        wire [WIDTH-1:0] time_over;
        assign time_over={WIDTH{1'b1}};
        reg [WIDTH-1:0] cnt1;

        always @ (posedge clk)
        begin
                if(cnt1==time_over)
                        begin       cnt1<=0;          end
                else
                        begin       cnt1<=cnt1+1'b1;     end
        end

        //=============================================
        //计数器 cnt1 与 state1 进行大小比较,以使 led 脉冲的占空比实现渐变
        //=============================================
        always @ (posedge clk)
        begin
                 if((cnt1+time_over/3)<=state1)       //在此增加了 time_over/3 这个量,是为了使
                                                      led 呼吸一次之后保持 1/3 时间的熄灭状态
                        led<=1;    //led 亮;
                else
                        led<=0;    //led 灭;
        end
endmodule
```

2.2.8 仿真 sc_computer_sim.v

仿真是必备而又有力的实验工具，而仿真文件匹配好参数即可，与原来无大不同。

```
`timescale 1ps/1ps                    // 仿真时间单位/时间精度

module pipelined_computer_sim;

      reg                 resetn_sim,mem_clk_sim,clock_sim;
      wire     [31:0]     pc_sim,inst_sim,ins_sim;
      wire     [31:0]     ealu_sim,malu_sim,walu_sim;

      reg      [31:0] in_port0_sim,in_port1_sim,in_port2_sim;
      wire     [31:0]     out_port0_sim,out_port1_sim,mem_dataout_sim,io_read_data_sim;

      pipelined_computer
pipeline_cpu_instance(resetn_sim,clock_sim,mem_clk_sim,pc_sim,ins_sim,inst_sim,ealu_si
m,malu_sim,walu_sim,

      out_port0_sim,out_port1_sim,in_port0_sim,in_port1_sim,in_port2_sim,mem_dataout_si
m,io_read_data_sim);

      initial
            begin
                  in_port0_sim = 7;
            end


       initial
            begin
                  in_port1_sim = 6;
            end


      initial
            begin
                  in_port2_sim = 4;
            end

      initial
      begin
            mem_clk_sim = 0;
            while (1)
                  #1    mem_clk_sim = ~ mem_clk_sim;
      end

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

      initial
      begin
            resetn_sim = 0;
            while (1)
                  #10 resetn_sim = 1;
      end

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

endmodule
```

3 实验步骤

整个实验过程可简单分为以下步骤：

1. 首先要熟练掌握流水线 CPU 基本原理和 IO 扩展方法，能用 Verilog 语言运用它。
2. 丰富 MIPS 指令集，补充代码，完善提高 CPU 的性能。
3. 要在 instmem 里加入需要运行 MIPS 程序的 mif 文件，可用 Qtspim 实现转换。
4. 做好输入输出在硬件上的控制与显示，加输入内存地址，并定义管脚。
5. 补充实现的扩展部分，LCD 显示操作提示，LED 呼吸灯。
6. 接下来则是编译，编译出现的问题逐一进行 debug。
7. 编译通过后写好 pipelined_coputer_sim.v 仿真文件，在 ModelSim Altara 进行仿真。
8. 细节的完善：十进制输出 未定义管脚的初值问题
9. 最后烧到板子上完成实验。

3.1 步骤一：理解流水线 CPU 工作原理，IO 扩展知识，熟练运用

- 理解流水线 CPU 实现的架构——5 个阶段及他们之间的流水线寄存器：取指令，译码，执行，访问数据 MEM，回写寄存器。
- 关于时钟，为实现流水线寄存器在读写操作的相互兼容，我们需要设计一个反相时钟 clock 和 mem_clock。因为 Quartus 只能识别上升沿触发，所以通过两个相反的时钟实现。
- Verilog 语言本质上是硬件描述语言，当 CPU 整体电路图成竹在胸时，其代码就像连电路一般机械描述即可。
- 流水线关键在于三大冒险的处理
  结构相关——增加硬件资源；编译器调度；
  控制相关——延迟转移；分支预测；
  数据相关——内部前推 forwarding；编译器调度。
- 统一编址方式实现 I/O 端口扩展
  在 pipemem.v 模块文件中，通过对输入的 lw 或 sw 指令的地址（试验中是 data[7]）的判断，实现对数据 RAM 和 IO 控制寄存器组的区分、和分别控制。这时需要将 io_input_reg.v、io_output_reg.v 放入 Project 中，同时完成主函数参数的对应。

3.2 步骤二：MIPS 指令的添加补充

MIPS 精简指令架构中共 31 条基本指令，之前的流水线 CPU 实现了其中的最基本的 20 条，我补充了基本指令 slt sllv 以及伪指令 mul div nor nop，还有无符号的实现 addu subu addiu，由此，我的 CPU 设计基本实现了 MIPS 的所有精简指令集。

通过对指令格式及控制信号真值表的填充实现，根据它完成对 Project 的 pipe_cu.v 及 alu.v 的填充完善，即可顺利实现其他指令。

关于乘除法器的实现，有三种方法，一是在 alu 中直接利用 Verilog 乘法语言，二是在 alu 中用左移加的循环实现，三是在 mif 文件的 MIPS 指令集中分支循环左移加实现乘法。具体如下：

```
4'b1101: s = a * b;      //1101 MUL;
```

```
4'b1101:                  //1101 MUL;
      begin
        s = 64'b0;
        exa=a; exb=b;
        for (i=1; i<33; i=i+1)
            begin
            if (exa[0] == 1'b1) s = s + exb;
            else s = s;
            exb = exb << 1;
            exa = exa >> 1;
      end
```

```
            add   $t0, $a0, $zero
            add   $t1, $a1, $zero
            add   $s0, $zero, $zero
Loop:       mod $t3, $t0, 2
            beq   $t3, $zero, Goon
            add   $s0, $s0, $t0
Goon:       sll     $t0, $t0, 1
            sra     $t1, $t1, 1
            j       Loop
```

3.3 步骤三：MIPS 指令编译 MIF 文件导入 imem

本次试验比前几次都要复杂，mif 文件指令也会长很多。我从 18 个开关中空出三个控制操作输入，意味着可以控制八种不同的计算，所以我挑出了八种最有代表性和新加的计算，即加、减、乘、除、移位、比较、异或和或非。

而这样一种简单的 switch 方法在汇编语言中却要用几十行实现，不断分支跳转和循环。值得注意的是，由于控制冒险是流水线 CPU 中代价最高的冒险，而实现八种不同的控制恰恰需要很多跳转，所以我实现时尽量将跳转语句之间的距离离远一些，以减少代价。

3.4 步骤四：数码管显示及增加输入内存地址

我们采用的是统一编址方式实现 I/O 端口扩展，现在需要三个输入（两个输入值，一个控制计算值），所以需要对内存编码进行补充。即在 io_input_reg.v 文件中加一个内存地址表示控制操作的输入。

此外，关于初值的影响问题。由于板子开关只有 18 个，而每个 in_port 都是 32 位，这样存在没有 pin 的位容易给输入带上初值而导致错误和影响，为避免这一问题，我在 io_input_reg.v 里赋值传递时将前面高位设为零，则不再出现初值问题的影响。

数码管显示是老生常谈了，不过这次实现十六进制转成十进制表示的方法比较简便，直接数以十再模十，即可依次得到十进制的每一位，再用七段数码管编译即可。

3.5 步骤五：补充实现扩展，LCD 显示操作提示，LED 呼吸灯。

关于 LCD 的显示，由于输入输出的动态显示已由数码管完成，而八种操作的说明需要显示出来，出于这个需要，我只采用了 LCD 的静态显示字符的功能。

LCD 显示由顶层模块 LED1602，初始延迟部分，和底层控制、底层设计代码实现。LCD 在时钟的控制下，由,LCD_DATA,LCD_EN,LCD_RS,LCD_RW,LCD_ON,LCD_BLON 这几个参量确定输出。module Reset_Delay(iCLK,oRESET) 为初始延迟代码，底层代码 LCD1602_Controller 实现了字符的输出，底层代码 LCD1602_TEST(iCLK,iRST_N, LCD_DATA, output LCD_RS,LCD_RW,LCD_EN)完成了定义输出内容和格式。

而 LED 呼吸灯则是 LED 灯光在微电脑控制之下完成由亮到暗的逐渐变化，这个效果既可以由硬件电路实现，也可以编程实现。我采用后者，通过定义控制每个占空比的持续时间，控制占空比增大与减小，以及生成与 state1 进行大小比较的计数器 cnt1，和最后计数器 cnt1 与 state1 进行大小比较,以使 led 脉冲的占空比实现渐变。

3.5 步骤六：编译及 debug

在编译过程中，总会出现各种各样的错误。有的时候 debug 导致没有治聋反而治哑了的悲剧。所以在 debug 时一定做好备份，避免不必要的弯路。

在本次试验中，印象较深的 bugs 分别是：

- 输入高位未定义管脚，有初值干扰。此外，关于初值的影响问题。由于板子开关只有 18 个，而每个 in_port 都是 32 位，这样存在没有 pin 的位容易给输入带上初值而导致错误和影响，为避免这一问题，我在赋值时将前面高位设为零，则不再出现初值问题的影响。如下所示：

  【io_input_reg.v】

  ```
  in_reg0 <= {25'b0,in_port0[6:0]};
  ```

- 关于本实验中的时钟，需要两个反相的 clock 和 memmem_clk。由于 clock 在 pipelined_computer 中既有对它和 mem_clk 取反的代码，又有在后续模块中的调用，导致编译时频频报错。我的解决办法是，将 clock 设为 output，再定义 wire 变量进行 assign，就可避免这一错误。

- 关于流水线实现与多跳转指令之间的矛盾问题，一开始我将所有操作都运算出来保存到不同的寄存器中，最后再看输入的操作决定 sw 输出那一个结果，这样流水线效率很低而且容易出错。所以后来改变指令顺序，先判断是何种输入，再运行计算存储，既提高效率，也提高正确率。

3.7 步骤七：仿真

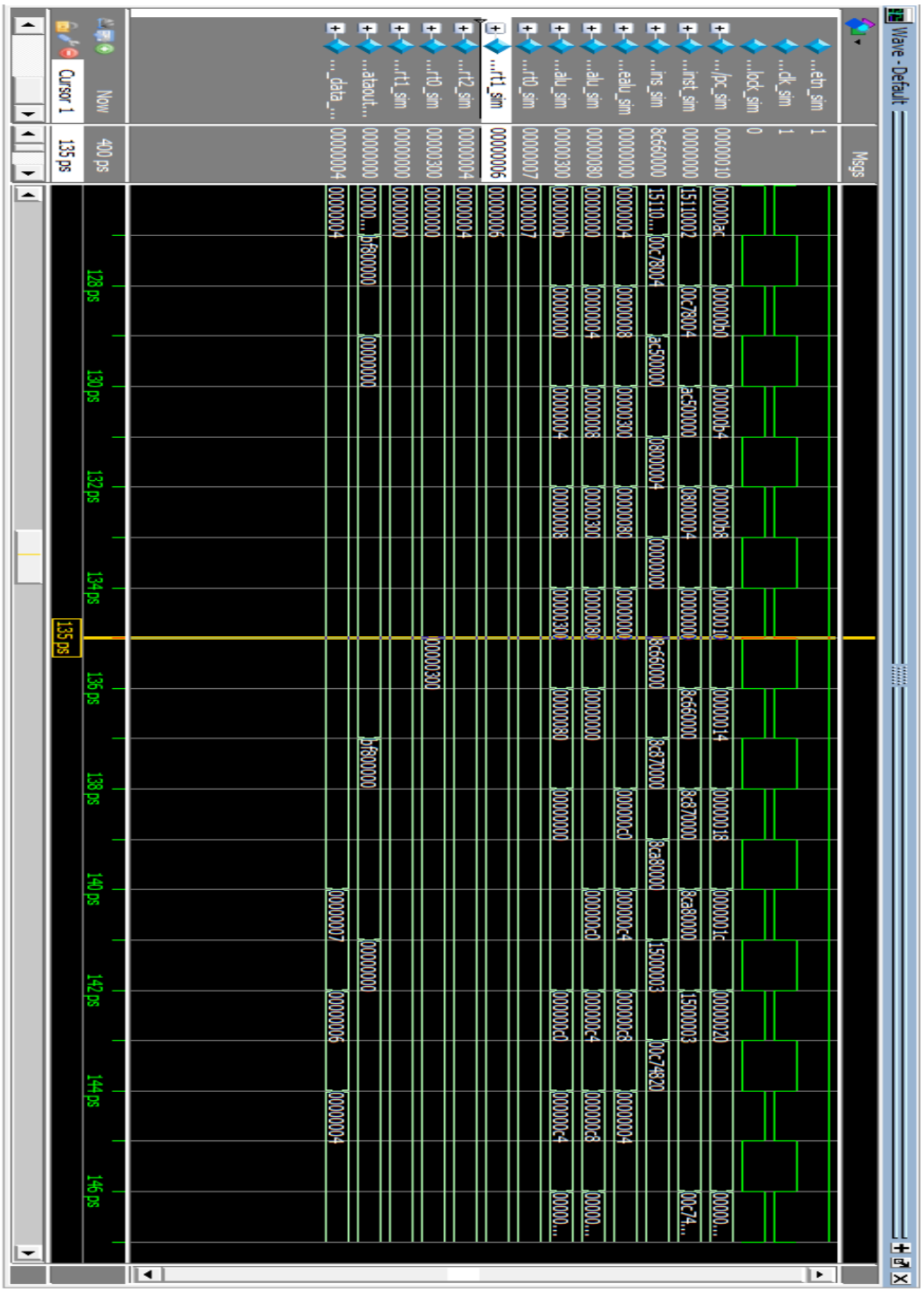
在 I/O 扩展后可先对输入输出进行仿真。直接将测试输入文件加入 Project 即可。

在对仿真 sc_computer_sim.v 文件完成编写后，将所有 verilog 文件导入 ModelSim Altara，进行 Compile All，然后 Add to -> wave > all items in region，运行即可得到仿真波形图。【仿真结果见实验结果】

4. 实验结果

实验成果是，有 2*7 个开关控制两个两位的数码管输入，3 个开关控制运算操作，另外四个数码管显示其输入的和。输入开关为二进制，数码管表示为十进制。

最终仿真截图如下：



5. 感想和建议

这次最终的实验是在我们自己定义的 CPU 上实现扩展功能，这不仅以增强对流水线 cpu 的理解和运用能力，还增加了我们对单片机的兴趣和爱好，甚至于自豪感，还有自主探索能力的提高。希望这次作业能向老师交上一份满意的答卷，以回报方老师和助教一个学期以来的指导与付出。

关于 CPU 的性能，采用流水线设计的最大时钟频率为 180.83MHz，与周围同学差不多，看来整体架构大家都相似，主要区别于实现的功能上。

其次，通过本次 CPU 设计大实验，我充分感受到了实验的意义并不是一项老师给学生的任务，而是在实验全过程后无论是对知识还是技术都有了更深一层次的理解，这种理解不是学习课本知识能达到的。同时实验这种摸索、探索、钻研、纠错的循环迭代中，领悟到科研的魅力。

还有仿真也是实验中不可或缺的步骤和方法，debug 时运用仿真，看每个时钟周期每个变量的状态，可以很快找到问题所在及解决办法，不至于像以前那样一头雾水，浪费时间与精力。

本次实验 debug 过程中还学会利用系统弹出的错误、警告、其他能等所有信息来挖掘对应的错误，从而纠正它的时候可以对症下药，快捷很多。

最后，实验的完成永远都没有止境，在我们搭的 CPU 上每个同学都有自己的想法、创意和亮点，课余有时间我们可以相互沟通交流改进。

总之，在这一学期的实验中学到了很多，特别是这最后一次。知识巩固与方法总结，兴趣培养与成就感提升，收获颇丰，这都要感谢老师和助教的指导与付出，谢谢！

王玫 5120309691

F1203026

phone 18818212442

wangmei1994515@qq.com