中山大學
SUN YAT-SEN UNIVERSITY

《计算机组成原理实验》

实验报告

（实验二）

学 院 名 称：	数据科学与计算机学院
专业（班级）：	17级计科教务一班
学 生 姓 名：	陈鸿峥
学 号：	17341015
时 间：	2018 年 11 月 12 日

成绩：

实验二：单周期CPU设计与实现

一、 实验目的

1. 掌握单周期CPU数据通路图的构成、原理及其设计方法
2. 掌握单周期CPU的实现方法，代码实现方法
3. 认识和掌握指令与CPU的关系
4. 掌握测试单周期CPU的方法

二、 实验内容

设计一个单周期CPU，使其至少能实现以下指令功能操作。指令与格式如下：

表 1: 基本MIPS指令及其格式

	指令	功能	op	rs	rt	rd	sham/func
算术运算	add rd, rs, rt	rd ← rs + rt	000000	rs(5位)	rt(5位)	rd(5位)	reserved
	sub rd, rs, rt	rd ← rs - rt	000001	rs(5位)	rt(5位)	rd(5位)	reserved
	addiu rt, rs, imm	rt ← rs + SignExt(imm)	000010	rs(5位)	rt(5位)	imm16	
逻辑运算	andi rt, rs, imm	rt ← rs & ZeroExt(imm)	010000	rs(5位)	rt(5位)	imm16	
	and rd, rs, rt	rd ← rs & rt	010001	rs(5位)	rt(5位)	rd(5位)	reserved
	ori rt, rs, imm	rt ← rs | ZeroExt(imm)	010010	rs(5位)	rt(5位)	imm16	
	or rd, rs, rt	rd ← rs | rt	010011	rs(5位)	rt(5位)	rd(5位)	reserved
移位	sll rd, rt, sa	rd ← rt<<ZeroExt(sa)	011000	reserved	rt(5位)	rd(5位)	sa(5位)
比较	slti rt, rs, imm	rs<SignExt(imm) ? rt=1 : rt=0	011100	rs(5位)	rt(5位)	imm16	
访存	sw rt, imm(rs)	mem[rs+SignExt(imm)] ← rt	100110	rs(5位)	rt(5位)	imm16	
	lw rt, imm(rs)	rt ← mem[rs+\text{SignExt}(imm)]	100111	rs(5位)	rt(5位)	imm16	
分支	beq rs, rt, imm	rs==rt ? pc+4+SignExt(imm)<<2 : pc+4	110000	rs(5位)	rt(5位)	imm16	
	bne rs, rt, imm	rs!=rt ? pc+4+SignExt(imm)<<2 : pc+4	110001	rs(5位)	rt(5位)	imm16	
	bltz rs, imm	rs<$0 ? pc+4+SignExt(imm)<<2 : pc+4	110010	rs(5位)	00000	imm16	
跳转	j addr	pc ← {(pc+4)[31:28],addr[27:2],2'b00}	111000	addr[27:2]			
停机	halt	halt	111111	00000000000000000000000000(26位)			

注意：reserved为预留部分，一般用0填充

三、 实验器材

电脑一台、Xilinx Vivado软件一套、Basys3板一块

四、 实验原理

I. 基本概念

i. 单周期CPU

单周期CPU是指CPU的每条指令都在一个时钟周期内完成，通常在每个时钟周期的上升沿触发。

CPU在处理指令时一般要经过以下五个阶段：

1. 取指(IF)：根据程序计数器PC中的指令地址，从存储器中取出一条指令，同时，PC根据指令字长度自动递增产生下一条指令所需要的指令地址；但遇到“地址转移”指令时，则需要对“转移地址”进行处理后送入PC。
2. 译码(ID)：对取指操作中得到的指令进行译码，确定该指令需要完成的操作，从而产生相应的操作控制信号，用于下一步的执行。
3. 执行(EXE)：根据指令译码得到的操作控制信号，执行指令动作。
4. 访存(MEM)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单，或者从存储器中得到数据地址单元中的数据。
5. 写回(WB)：将指令执行的结果或者访问存储器得到的数据写回相应的目标寄存器。

ii. 数据通路及控制信号

表 2: 控制信号

控制信号名	状态0	状态1
Reset	初始化PC为0	PC接收新地址
PCWre	PC不更改，halt	PC更改
RegDst	写入寄存器地址来自rt字段	写入寄存器地址来自rd字段
RegWrite	寄存器不可写	寄存器可写
ALUSrcA	寄存器rs内容	立即数sa
ALUSrcB	寄存器rt内容	立即数imm
ExtOp	零扩展	符号扩展
ALUOp	见表4	
MemToReg	ALU输出	内存读取
MemWrite	内存不可写	内存可写
Branch	非分支	分支
Jump	非跳转	跳转

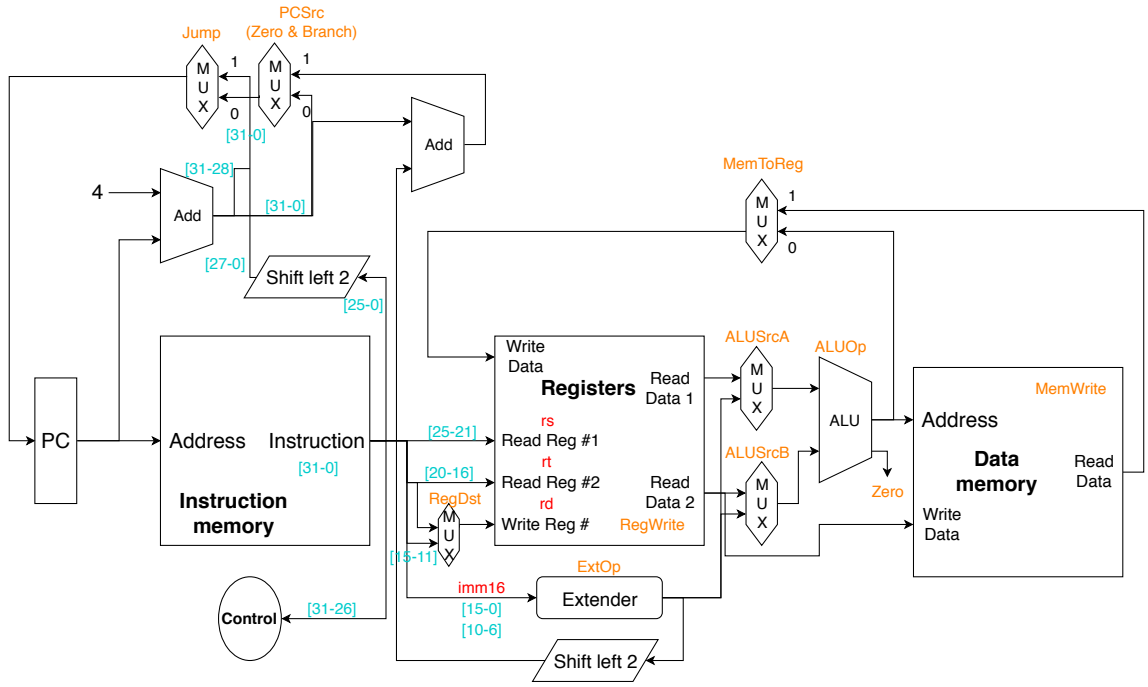

图 1: 基本数据通路

iii. MIPS指令格式

MIPS指令可分为以下三种格式： 其中各字段缩写含义如表3所示。

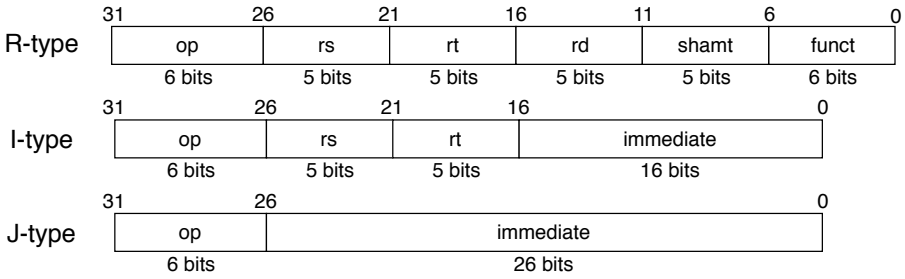

图 2: MIPS指令类型

II. 各指令数据通路分析

公共的数据通路为取指和PC+4，见图3最左红色通路，下文将不再提及。

i. 算术逻辑运算

1. add/sub/and/or指令均为R类型，数据通路相同，唯一不同为ALU操作码的选择。见图3，执行阶段从指令中读入rs、rt、rd三个寄存器的地址，然后从寄存器堆中读出rs和rt寄存器的内容，送至ALU进行运算，最后写回rd寄存器。
2. addiu/andi/ori/slti指令均为I类型，数据通路相同，同样是ALU操作码不同。见图4，执行阶段从指令中读入rs、rt寄存器的地址，但rt是作为写入寄存器(RegDst=0)；指令低16位进行扩充，将rs的内容和立即数送入ALU运算，最后写回rt寄存器。注意addiu/slti进行

表 3: MIPS指令各字段缩写含义

op	操作码
rs	只读，第一个源操作数寄存器地址/编号，范围为0x00~0x1F
rt	可读可写，第二个源操作数地址或目标操作数寄存器地址
rd	只写，目的操作数寄存器地址
sham(shift amt)	位移量，在移位指令中用于指定移多少位
funct	功能码，在R类型指令中配合op一起使用
immediate	16位立即数
address	目标转移地址

符号扩展，andi/ori进行零扩展。

3. sll指令为R类型，但是指令格式比较特殊。见图5，执行阶段从指令中读入rt寄存器的地址并取出，取指令的[10:6]位读出立即数sa并进行零扩展(ALUSrcA=1)，利用ALU对rt的内容及sa进行移位操作，结果写回rd寄存器。

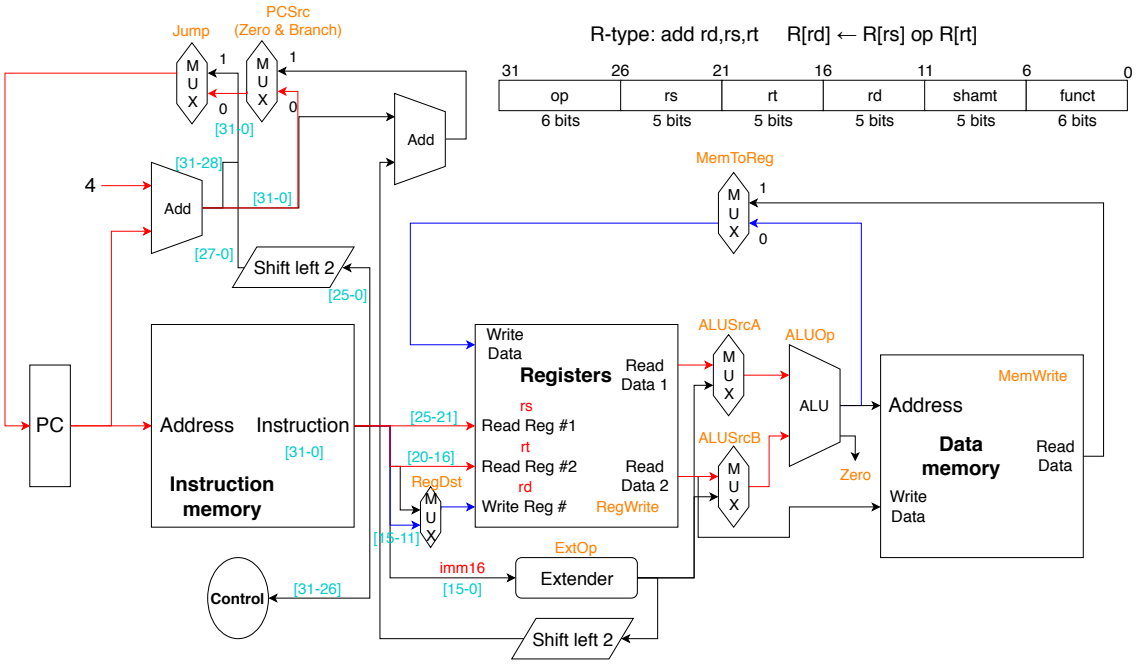

图 3: Add/sub/and/or通路

ii. 访存

1. sw为I类型。见图6，执行阶段从指令中读入rs、rt寄存器的地址，对偏移量(imm)进行**符号**扩展，与rs寄存器的内容相加得到内存地址，将rt寄存器的内容写入内存。
2. lw为I类型。见图7，执行阶段从指令中读入rs、rt寄存器的地址，rt作为写入寄存器(RegDst=0)，对偏移量(imm)进行**符号**扩展，与rs寄存器的内容相加得到内存地址，将内存的内容写

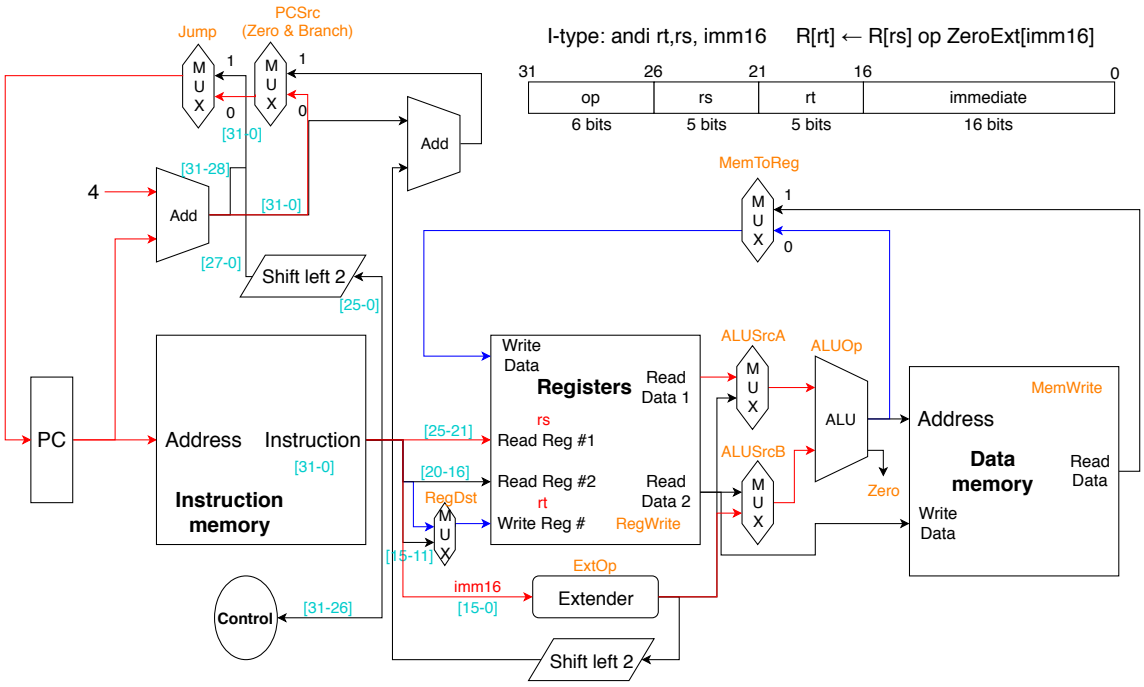

图 4: Addiu/andi/ori/slti通路

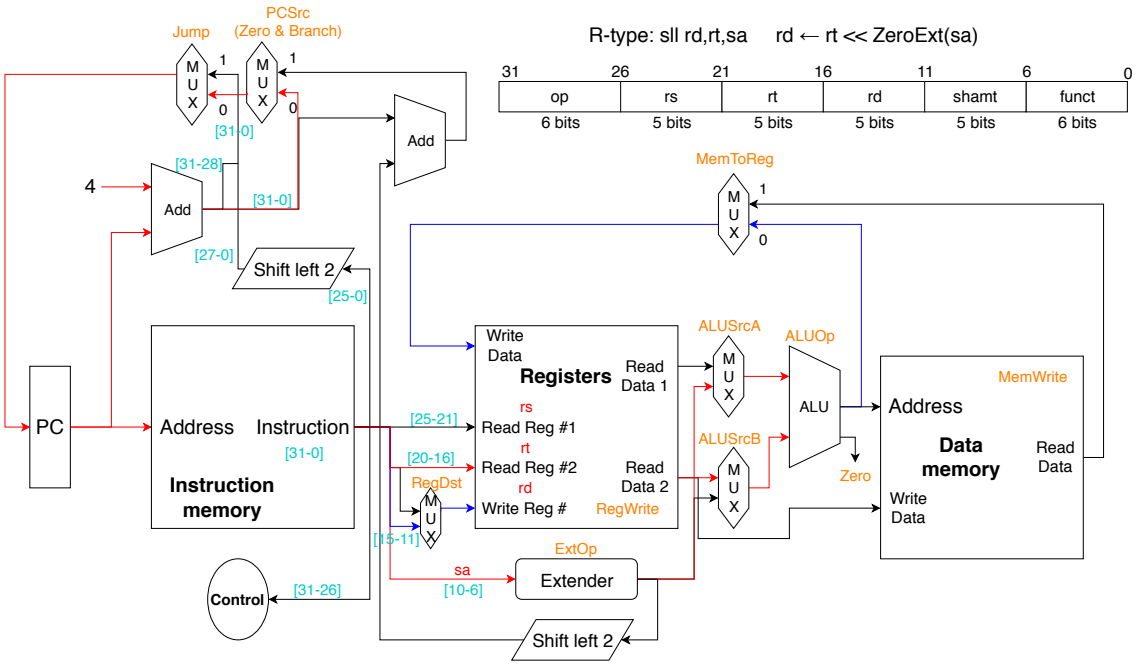

图 5: sll通路

入rt寄存器。

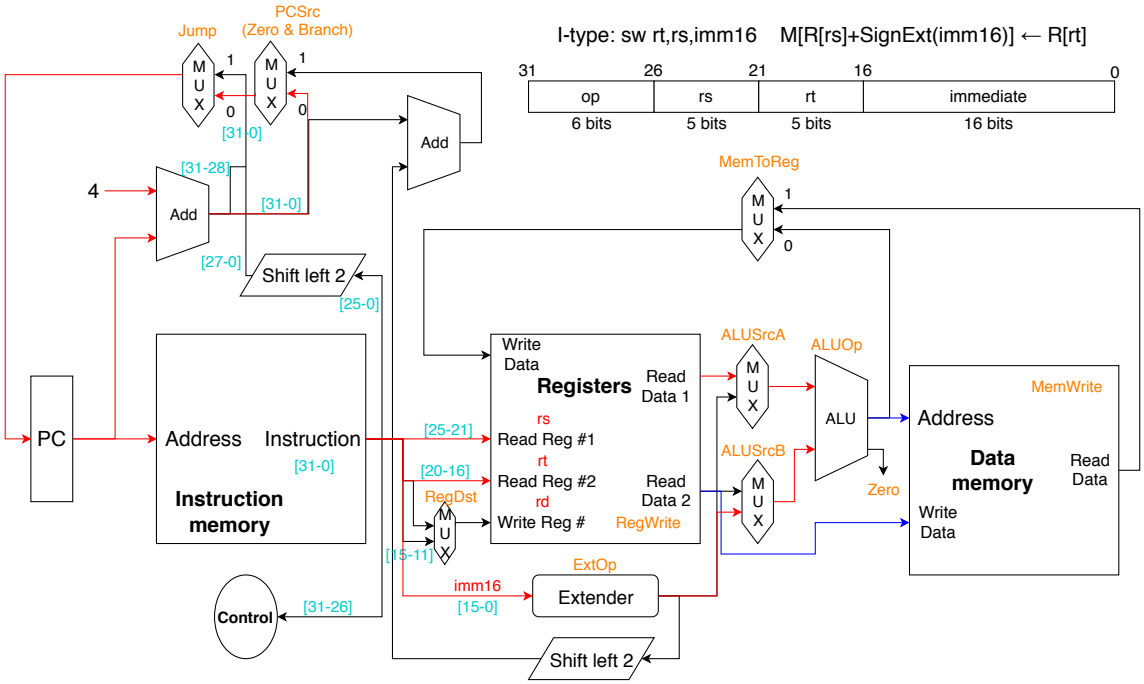

图 6: sw通路

iii. 分支跳转

由于MIPS指令为32位(4字节)，一般情况下PC每次自增都加4，即PC末两位一定为0，放在指令中（PC偏移/转移地址）即可省略末两位。指令读出时则要左移2位，将末尾的0补齐。为最大程度利用指令的每一位，跳转指令也是将高4位和低2位都省略掉了，读出时要补回。

1. beq/bne/bltz为I类型。见图8，执行阶段从指令中读入rs、rt寄存器的地址，利用ALU对rs、rt寄存器的内容进行相减(beq/bne)或有符号比较(bltz)，若结果为0，则Zero标志置1，否则置0；同时，立即数imm进行符号扩展，并左移两位，与原有的PC+4再相加，得到是否执行分支跳转指令(beq: PCSrc=Zero, bne/bltz: PCSrc=~Zero)。
2. jump为j类型。见图9，执行阶段直接对指令的低26位左移2位，补上PC+4的高4位，得到跳转地址，更新PC。

iv. 停机指令

PCWrite置为0，不改变PC的值，PC不再变化

III. 各组件实现

各组件实现详情见第8章节，均有详细注释。在这里仅仅提一些需要注意的点。

1. 指令存储器和数据存储器均采用小端(little endian)存储，并且位宽为8位，故生成的指令文件需要进行预处理，最先读入的应该是指令的低8位。这里用`Python`进行预处理并生成指令二进制代码文件，读入时直接将其初始化到指令存储器中。

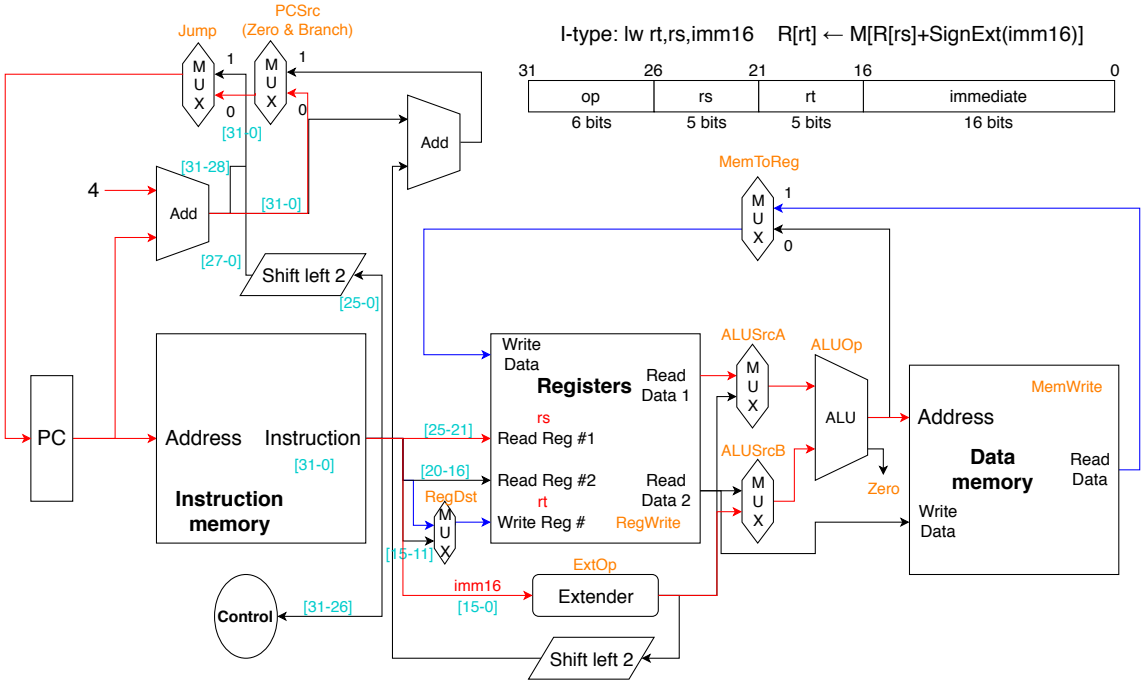

图 7: lw通路

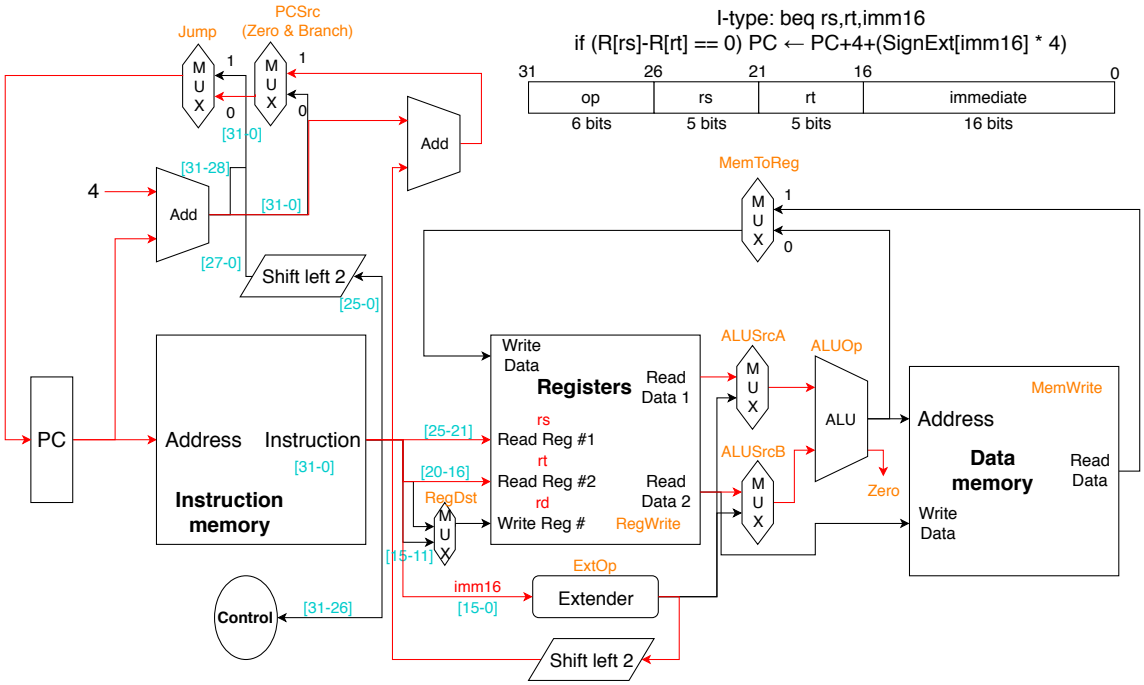

图 8: beq/bne/bltz通路

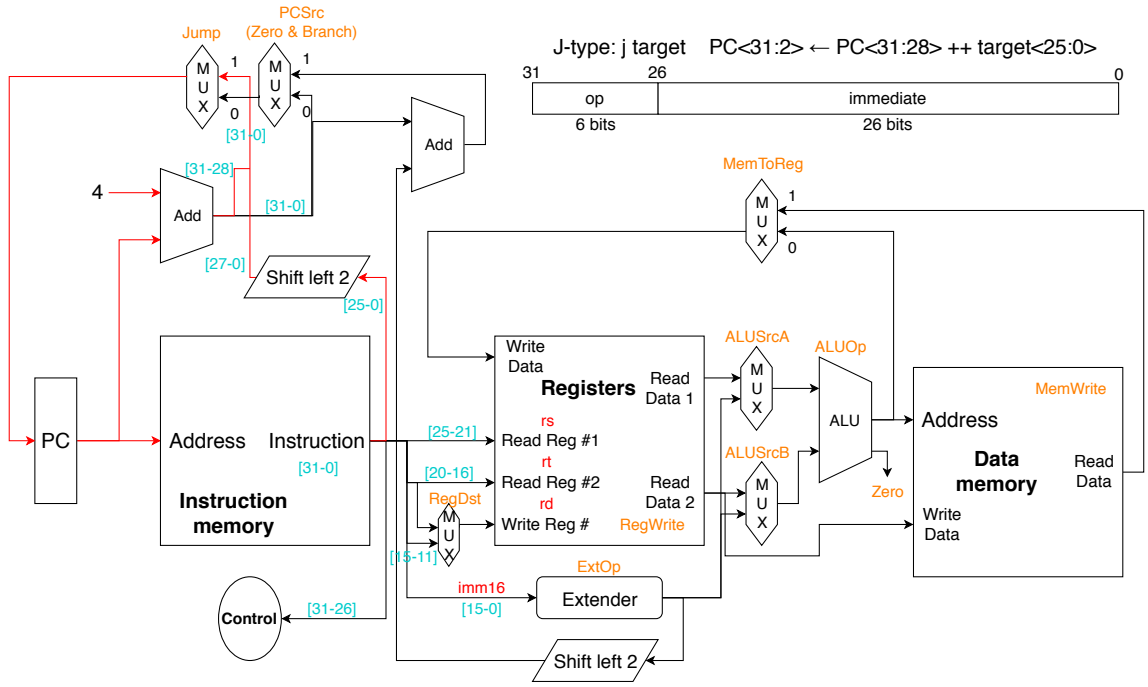

图 9: jump通路

2. 程序计数器(PC)需要初始化置零(0x000000)，遇到reset信号时同样置零，而且只有在PCWrite=1时才更新PC。
3. 多路选择器(MUX)需要实现一个32位和一个5位的，32位可以复用（创建多个实例）。
4. ALU的功能见表4。
5. 控制单元的设计见第4.4节。

IV. 控制单元

由前面的分析，可以得到控制单元的编码表，见表5。

五、 实验步骤

I. 仿真模拟

采用表6给出的汇编程序代码段进行测试。注意指令的二进制编码应该对应到MIPS的每一个字段上，特别注意sll的书写，负数用补码表示等。生成二进制指令文件后，CPU将其读入指令存储器并执行，看波形结果。

II. 上板

使用Basys3板运行所设计的CPU时，需要通过4个七段数码管来查看当前CPU的执行情况。

只考虑指令和数据的低8位，即指令存储器中的指令地址范围和数据存储器中的数据地址范围均为 0x00～ 0xFF。

通过Basys3板上的开关SW15、SW14选择七段数码管显示的内容，具体显示的功能码如表7所示。

表 4: ALU功能码

ALUOp[2:0]	功能	描述
000	$Y = A + B$	加
001	$Y = A - B$	减
010	$Y = B << A$	左移
011	$Y = A \vee B$	逻辑或
100	$Y = A \wedge B$	逻辑与
101	$Y = (A < B)\ ?\ 1\ :\ 0$	比较无符号数
110	$Y = (((A < B)\&\&(A[31] == B[31]))$ $\|\|\ \ ((A[31] == 1\&\&B[31] == 0)))$ $?\ 1\ :\ 0$	比较有符号数
111	$Y = A \oplus B$	逻辑异或

III. 七段数码管显示电路

基本实现步骤如下：

1. 对Basys3板系统时钟信号(100MHz)进行分频，使得数码管内容能够正常显示。频率不宜过快，过快会导致全部数码管显示的都是8；也不宜过慢，否则数码管会出现暂留现象，无法连续显示。在本次实现中采用10kHz的频率进行显示。
2. 用上面得到的频率生成4进制计数器，用于产生4个数位选信号AN3-AN0。这4个数可控制哪个数码管亮，一共四组编码(1110、1101、1011、0111)，每组编码中只有一位为0（亮），其余都为1（灭），在每一个位选信号到来之时更新数码管显示。其实前两步可以一并完成，见第8章分频计数器一节。
3. 将从CPU接收到的相应数据转换为数码管显示信号，与位选信号一起送往数码管显示输出。

IV. 其他注意事项

1. 共阳极数码管

   Basys3板的数码管均为共阳极，故设置七段数码管和位选信号时都要考虑0为亮，1为灭。

2. 两个时钟信号

   CPU工作时钟和Basys3板系统时钟是两个不同的时钟，但是FPGA只支持单一全局时钟，否则在routing阶段会报错

   ```
   Poor placement for routing between an IO pin and BUFG
   ```

   故需要采取其他方法。只设置全局时钟为`clk`，并将CPU工作时钟`clk_cpu`与其同步，即在每一个`clk`上升沿时，赋值`in<=clk_cpu`，通过下面消抖处理后，将`in`作为真正的CPU工作时钟。

表 5: 指令控制器编码

	指令	op	RegDst	ExtSel	RegWrite	ALUSrcA/B		ALUOp	MemToReg	MemWrite	Branch	Jump
算术运算	add rd, rs, rt	000000	1	x	1	0	0	000	0	0	0	0
	sub rd, rs, rt	000001	1	x	1	0	0	001	0	0	0	0
	addiu rt, rs, imm	000010	0	**1**	1	0	1	000	0	0	0	0
逻辑运算	andi rt, rs, imm	010000	0	0	1	0	1	100	0	0	0	0
	and rd, rs, rt	010001	1	x	1	0	0	100	0	0	0	0
	ori rt, rs, imm	010010	0	0	1	0	1	011	0	0	0	0
	or rd, rs, rt	010011	1	x	1	0	0	011	0	0	0	0
移位	sll rd, rt, sa	011000	1	x	1	1	0	010	0	0	0	0
比较	slti rt, rs, imm	011100	0	1	1	0	1	110	0	0	0	0
访存	sw rt, imm(rs)	100110	x	1	0	0	1	000	x	1	0	0
	lw rt, imm(rs)	100111	0	1	1	0	1	000	1	0	0	0
分支	beq rs, rt, imm	110000	x	1	0	0	0	001	x	0	1	0
	bne rs, rt, imm	110001	x	1	0	0	0	001	x	0	1	0
	bltz rs, imm	110010	x	1	0	0	0	110	x	0	1	0
跳转	j addr	111000	x	x	0	x	x	x	x	0	0	1
停机	halt	111111	x	x	x	x	x	x	x	x	x	x

表 6: 测试代码段指令

address	instruction	op	rs	rt	rd/imm	hex
0x00000000	addiu $1,$0,8	000010	00000	00001	0000 0000 0000 1000	0x08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	0x48020002
0x00000008	add $3,$2,$1	000000	00010	00001	0001 1000 0000 0000	0x00411800
0x0000000C	sub $5,$3,$2	000001	00011	00010	0010 1000 0000 0000	0x04622800
0x00000010	and $4,$5,$2	010001	00101	00010	0010 0000 0000 0000	0x44A22000
0x00000014	or $8,$4,$2	010011	00100	00010	0100 0000 0000 0000	0x4C824000
0x00000018	sll $8,$8,1	011000	00000	01000	0100 0000 0100 0000	0x60084040
0x0000001C	bne $8,$1,-2 (≠,转18)	110001	01000	00001	1111 1111 1111 1110	0xC501FFFE
0x00000020	slti $6,$2,4	011100	00010	00110	0000 0000 0000 0100	0x70460004
0x00000024	slti $7,$6,0	011100	00110	00111	0000 0000 0000 0000	0x70c70000
0x00000028	addiu $7,$7,8	000010	00111	00111	0000 0000 0000 1000	0x08e70008
0x0000002C	beq $7,$1,-2 (=,转28)	110000	00111	00001	1111 1111 1111 1110	0xC0E1FFFE
0x00000030	sw $2,4($1)	100110	00001	00010	0000 0000 0000 0100	0x98220004
0x00000034	lw $9,4($1)	100111	00001	01001	0000 0000 0000 0100	0x9C290004
0x00000038	addiu $10,$0,-2	000010	00000	01010	1111 1111 1111 1110	0x080AFFFE
0x0000003C	addiu $10,$10,1	000010	01010	01010	0000 0000 0000 0001	0x094A0001
0x00000040	bltz $10,-2 (< 0,转3C)	110010	01010	00000	1111 1111 1111 1110	0xC940FFFE
0x00000044	andi $11,$2,2	010000	00010	01011	0000 0000 0000 0010	0x404B0002
0x00000048	j 0x00000050	111000	00000	00000	0000 0000 0001 0100	0xE0000014
0x0000004C	or $8,$4,$2	010011	00100	00010	0100 0000 0000 0000	0x4C824000
0x00000050	halt	111111	00000	00000	0000 0000 0000 0000	0xFC000000

表 7: 数码管显示功能码

SW15	SW14	左边	右边
0	0	当前PC	下条PC
0	1	rs寄存器地址	rs寄存器数据
1	0	rt寄存器地址	rt寄存器数据
1	1	ALU结果输出	DB总线数据

3. 消抖处理

   如果一次触发持续一段时间不改变状态（如原状态是0，变更为1且保持100ns），则该触发是有效的人为触发，否则则视为抖动。故可以设置一个按键状态的历史记录(16位)，如果连续14位都为1，则将`clk_cpu`设为高电平，否则为无效触发。具体实施细节见第8章写板电路一节。

V. 引脚分配

见表8。

表 8: 引脚分配表

引脚	名称	作用
W5	clk	全局时钟
T17	clk_cpu	CPU时钟（单脉冲信号）
V17	reset	复位信号
R2/T1	SW_in	数码管显示内容选择
W4-U2	AN3-AN0	数码管位选信号
W7-U7	seg6-seg0	七段数码管内容

六、 结果与分析

I. 仿真模拟

II. 上板

七、 实验心得

1.

八、 程序清单

I. CPU主模块

```
timescale 1ns / 1ps

module CPU (
    input clk, reset,
    output RegDst,
    output ExtSel,
    output RegWrite, MemWrite,
    output ALUSrcA, ALUSrcB,
    output [2:0] ALUOp,
    output MemToReg,
    output Branch, Jump, Zero,
    output PCWrite,
    output [31:0] currPC, nextPC, instruction, alu_res,
    output wire [31:0] d1, d2, rsData, rtData,
    output wire [4:0] rs, rt, rd, sa, dbData
    );

    wire [5:0] opcode;
    wire [15:0] imm;
    wire [31:0] pc4;

    assign opcode = instruction[31:26];
    assign rs = instruction[25:21];
    assign rt = instruction[20:16];
    assign rd = instruction[15:11];
    assign sa = instruction[10:6];
    assign imm = instruction[15:0];

    PC pc(
        .clk(clk),
        .reset(reset),
        .PCWrite(PCWrite),
        .currPC(currPC),
        .nextPC(nextPC)
        );

    Adder add_pc4(
        .A(currPC),
        .B({{29{1'b0}},3'b100}),
        .res(pc4)
        );

    // instruction fetch (IF)
    InstructionMemory im(
        .address(currPC),
        .dataOut(instruction)
        );

    // instruction decode (ID)
    ControlUnit control(
        // input
        .opcode(opcode),
        .Zero(Zero),
        // output
        .RegDst(RegDst),
        .ExtSel(ExtSel),
        .RegWrite(RegWrite),
        .ALUSrcA(ALUSrcA),
        .ALUSrcB(ALUSrcB),
        .ALUOp(ALUOp),
        .MemToReg(MemToReg),
        .MemWrite(MemWrite),
        .Branch(Branch),
        .Jump(Jump),
        .PCWrite(PCWrite)
        );

    // execution (EXE)
    Registers reg_file(
        .clk(clk),
        .r1(rs),
        .r2(rt),
        .wr(mux_regdst.res),
        .RegWrite(RegWrite),
        .wd(mux_memToReg.res)
        // d1 -> mux_srcA.A
        // d2 -> mux_srcB.A / dm.dataIn
        );

    assign d1 = mux_srcA.res;
    assign d2 = mux_srcB.res;
    assign rsData = reg_file.d1;
    assign rtData = reg_file.d2;

    MUX5 mux_regdst(
        .Sel(RegDst),
        .A(rt),
        .B(rd)
        // res -> reg_file.wr
        );

    MUX mux_srcA(
        .Sel(ALUSrcA),
        .A(reg_file.d1),
        .B({{27{1'b0}},sa})
        // res -> alu.A
        );

    Extender extender(
        .Sel(ExtSel),
        .dataIn(imm)
        // dataOut -> mux_srcB.B / sl_16
        );

    MUX mux_srcB(
        .Sel(ALUSrcB),
        .A(reg_file.d2),
        .B(extender.dataOut)
        // res -> alu.B
        );

    ALU alu(
        .op(ALUOp),
        .A(mux_srcA.res),
        .B(mux_srcB.res),
        // res -> dm.address / mux_memToReg.A
        .res(alu_res),
        .zero(Zero)
        );

    // access memory (MEM)
    DataMemory dm(
        .clk(clk),
        .address(alu_res),
        .MemWrite(MemWrite),
        .dataIn(reg_file.d2)
        // dataOut -> mux_memToReg
        );

    // write back (WB)
    MUX mux_memToReg(
        .Sel(MemToReg),
        .A(alu_res),
        .B(dm.dataOut)
        // res -> reg_file.wd
        );

    assign dbData = mux_memToReg.res;

    // jump & branch
    ShiftLeft sl(
        .dataIn(extender.dataOut)
        // dataOut -> add_target.B
        );

    Adder add_target(
        .A(pc4),
        .B(sl.dataOut)
        // res -> mux_branch.B
        );

    MUX mux_branch(
        .Sel(Branch),
        .A(pc4),
        .B(add_target.res)
        // res -> mux_jump.A
        );

    MUX mux_jump(
        .Sel(Jump),
        .A(mux_branch.res),
```

```
        .B({pc4[31:28],instruction[25:0],2'b00}),
        .res(nextPC)
        );

endmodule
```

II. 指令存储器

```
module InstructionMemory (
    input [31:0] address,
    output [31:0] dataOut
    );

    reg [7:0] memory [0:255];

    // initialization
    initial $readmemb("D:/instruction.data",memory);
    // $display("Read in data successfully!");

    // output data (little endian)
    assign dataOut = {memory[address + 3],
                      memory[address + 2],
                      memory[address + 1],
                      memory[address]};

endmodule
```

III. 程序计数器(PC)

```
module PC (
    input clk,
    input reset,
    input PCWrite,
    input [31:0] nextPC,
    output reg [31:0] currPC
    );

    initial currPC = 0;

    always @(posedge clk or negedge reset) begin
        if (reset == 0)
            currPC <= 0;
        else if (PCWrite == 1)
            currPC <= nextPC;
    end

endmodule
```

IV. 寄存器堆

```
module Registers (
    input clk,
    input [4:0] r1, // read reg #1 address
    input [4:0] r2, // read reg #2 address
    input [4:0] wr, // write reg address
    input RegWrite,
    input [31:0] wd, // write data
    output [31:0] d1, // read data 1
    output [31:0] d2 // read data 2
    );

    reg [31:0] register [0:31];

    // initialization
    integer i;
    initial begin
        for (i = 0; i < 32; i = i + 1)
            register[i] = 0;
    end

    // read data
    assign d1 = (r1 == 0) ? 0 : register[r1];
    assign d2 = (r2 == 0) ? 0 : register[r2];

    // write data
    always @(posedge clk) begin
        if (wr != 0 && RegWrite == 1)
            register[wr] <= wd;
    end

endmodule
```

V. 数据存储器

```
module DataMemory (
    input clk,
    input [31:0] address,
    input MemWrite,
    input [31:0] dataIn, // write data
    output [31:0] dataOut // read data
    );

    reg [7:0] memory [0:255];

    // initialization
    integer i;
    initial begin
        for (i = 0; i < 7; i = i + 1)
            memory[i] = 0;
    end

    // read data
    assign dataOut = {memory[address + 3], memory[
         address + 2], memory[address + 1], memory[
         address]};

    // write data
    always @(posedge clk) begin
        if (MemWrite == 1 && address >= 1 && address <=
              255) begin
            // little endian
            memory[address + 3] <= dataIn[31:24];
            memory[address + 2] <= dataIn[23:16];
            memory[address + 1] <= dataIn[15:8];
            memory[address] <= dataIn[7:0];
        end
    end

endmodule
```

VI. 控制单元

```
module ControlUnit (
    input [5:0] opcode,
    input Zero,
    output reg RegDst,
    output reg ExtSel,
    output reg RegWrite,
    output reg ALUSrcA,
    output reg ALUSrcB,
    output reg [2:0] ALUOp,
    output reg MemToReg,
    output reg MemWrite,
    output reg Branch,
    output reg Jump,
    output reg PCWrite
    );

    always @ (opcode or Zero) begin // Zero!
        RegDst   <= 0;
        ExtSel   <= 0;
        RegWrite <= 1;
        ALUSrcA  <= 0;
        ALUSrcB  <= 0;
        ALUOp    <= 3'b000;
        MemToReg <= 0;
        MemWrite <= 0;
        Branch   <= 0;
        Jump     <= 0;
        PCWrite  <= 1;
        case (opcode)
            6'b000000: begin // add rd, rs, rt
                    RegDst <= 1;
                    end
            6'b000001: begin // sub rd, rs, rt
                    RegDst <= 1;
                    ALUOp <= 3'b001;
                    end
            6'b000010: begin // addiu rt, rs, imm
                    ExtSel <= 1; // ???
```

```
                    ALUSrcB <= 1;
                    end
            6'b010000: begin // andi rt, rs, imm
                    ALUSrcB <= 1;
                    ALUOp <= 3'b100;
                    end
            6'b010001: begin // and rd, rs, rt
                    RegDst <= 1;
                    ALUOp <= 3'b100;
                    end
            6'b010010: begin // ori rt, rs, imm
                    ALUSrcB <= 1;
                    ALUOp <= 3'b011;
                    end
            6'b010011: begin // or rd, rs, rt
                    RegDst <= 1;
                    ALUOp <= 3'b011;
                    end
            6'b011000: begin // sll rd, rt, sa
                    RegDst <= 1;
                    ALUSrcA <= 1;
                    ALUOp <= 3'b010;
                    end
            6'b011100: begin // slti rt, rs, imm
                    ExtSel <= 1;
                    ALUSrcB <= 1; // remember!
                    ALUOp <= 3'b110;
                    end
            6'b100110: begin // sw rt, imm(rs)
                    ExtSel <= 1;
                    RegWrite <= 0;
                    ALUSrcB <= 1;
                    MemWrite <= 1;
                    end
            6'b100111: begin // lw rt, imm(rs)
                    ExtSel <= 1;
                    ALUSrcB <= 1;
                    MemToReg <= 1;
                    end
            6'b110000: begin // beq rs, rt, imm
                    ExtSel <= 1;
                    RegWrite <= 0;
                    ALUOp <= 3'b001;
                    Branch <= Zero;
                    end
            6'b110001: begin // bne rs, rt, imm
                    ExtSel <= 1;
                    RegWrite <= 0;
                    ALUOp <= 3'b001;
                    Branch <= ~Zero; // (rs - rt == 0)
                         ? 1 : 0 Not equal!
                    end
            6'b110010: begin // bltz rs, imm
                    ExtSel <= 1;
                    RegWrite <= 0;
                    ALUOp <= 3'b110; // compare sign
                    Branch <= ~Zero; // a < 0 ? 1 : 0
                    end
            6'b111000: begin // j addr
                    RegWrite <= 0;
                    Jump <= 1;
                    end
            6'b111111: begin // halt
                    PCWrite <= 0;
                    end
        endcase
    end

endmodule
```

VII. 算术逻辑单元(ALU)

```
module ALU (
    input [2:0] op,
    input [31:0] A,
    input [31:0] B,
    output reg [31:0] res,
    output reg zero
    );

    initial begin
        res = 0;
    end

    always @(op or A or B) begin
        case (op)
            3'b000: res = A + B;
            3'b001: res = A - B;
            3'b010: res = B << A;
            3'b011: res = A | B;
            3'b100: res = A & B;
            3'b101: res = (A < B) ? 8'h00000001 : 0;
            3'b110: res = ((A < B && A[31] == B[31]) //
                   both pos/neg num
                         || (A[31] == 1 && B[31] == 0))
                              // A neg B pos
                        ? 8'h00000001 : 0;
            3'b111: res = A ^ B;
        endcase
        zero <= (res == 0) ? 1 : 0;
    end

endmodule

module Adder (
    input [31:0] A,
    input [31:0] B,
    output [31:0] res
    );

    assign res = A + B;

endmodule

module ShiftLeft (
    input [31:0] dataIn,
    output [31:0] dataOut
    );

    assign dataOut = dataIn << 2;

endmodule
```

VIII. 多路选择器(MUX)

```
module MUX (
    input Sel,
    input [31:0] A,
    input [31:0] B,
    output reg [31:0] res
    );

    always @(Sel or A or B) begin
        res <= (Sel == 0) ? A : B;
    end

endmodule

module MUX5 (
    input Sel,
    input [4:0] A,
    input [4:0] B,
    output reg [4:0] res
    );

    always @(Sel or A or B) begin
        res <= (Sel == 0) ? A : B;
    end

endmodule
```

IX. 数据扩展器

```
module Extender (
    input Sel,
    input [15:0] dataIn,
    output reg [31:0] dataOut
    );

    initial dataOut = 0;

    always @(Sel or dataIn) begin // dataIn!!!
        if (Sel == 0) // ZeroExt
```

```
            dataOut = {{16{1'b0}},dataIn[15:0]};
        else // SignExt
            dataOut = {{16{dataIn[15]}},dataIn[15:0]};
    end

endmodule
```

X. 仿真代码

```
module CPU_sim (
    output RegDst,
    output ExtSel,
    output RegWrite, MemWrite,
    output ALUSrcA, ALUSrcB,
    output [2:0] ALUOp,
    output MemToReg,
    output Branch, Jump, Zero,
    output PCWrite,
    output [31:0] currPC, nextPC, instruction, alu_res,
    output wire [31:0] d1, d2
    );

    reg clk;
    reg reset;

    CPU cpu(
        .clk(clk),
        .reset(reset),
        .RegDst(RegDst),
        .ExtSel(ExtSel),
        .RegWrite(RegWrite),
        .MemWrite(MemWrite),
        .ALUSrcA(ALUSrcA),
        .ALUSrcB(ALUSrcB),
        .ALUOp(ALUOp),
        .MemToReg(MemToReg),
        .Branch(Branch),
        .Jump(Jump),
        .Zero(Zero),
        .PCWrite(PCWrite),
        .currPC(currPC),
        .nextPC(nextPC),
        .instruction(instruction),
        .alu_res(alu_res),
        .d1(d1),
        .d2(d2)
        );

    initial begin
        clk = 1;
        reset = 0;
        // wait for initialization
        #80;
        reset = 1;
    end

    always #50 clk = ~clk;

endmodule
```

XI. 分频计数器

```
module Counter(
    input clr, // clear, say reset
    input clk, // original clock
    output reg [1:0] count_4,
    output reg clk_seg
    );

    // display 10kHz
    reg [16:0] count_dis; // 26 bits to store count:
         2^17 > 10^5
    always @ (posedge clk or posedge clr)
    begin
        if (clr == 1) // reset
        begin
            clk_seg <= 0;
            count_dis <= 0;
        end
        else if (count_dis == 50_000 - 1) // return 0
        begin
            clk_seg <= ~clk_seg;
            count_dis <= 0;
        end
        else
        begin
            clk_seg <= clk_seg;
            count_dis <= count_dis + 1;
        end
    end

    always @ (posedge clk_seg or posedge clr)
    begin
        if (clr == 1 || count_4 == 4)
            count_4 <= 0;
        else
            count_4 <= count_4 + 1;
    end

endmodule
```

XII. 七段数码管

```
module SegDisplay (
    input [3:0] data,
    output reg [6:0] dispcode
    );

    always @(data)
        case(data)
            // 0: on    1: off
            1: dispcode = 7'b100_1111;
            2: dispcode = 7'b001_0010;
            3: dispcode = 7'b000_0110;
            4: dispcode = 7'b100_1100;
            5: dispcode = 7'b010_0100;
            6: dispcode = 7'b010_0000;
            7: dispcode = 7'b000_1111;
            8: dispcode = 7'b000_0000;
            9: dispcode = 7'b000_0100;
            10: dispcode = 7'b000_1000; // A
            11: dispcode = 7'b110_0000; // b
            12: dispcode = 7'b011_0001; // C
            13: dispcode = 7'b100_0010; // d
            14: dispcode = 7'b001_0000; // e
            15: dispcode = 7'b011_1000; // F
        endcase

endmodule
```

XIII. 写板电路

```
module Show(
    input clk,
    input clk_cpu,
    input reset,
    input [1:0] SW_in,
    output reg [6:0] dispcode,
    output reg [3:0] out
    );

    // synchronize
    reg in=1'b0;
    always @(posedge clk) begin
        in <= clk_cpu;
    end
    wire in_syn = in;

    // reduce jitter
    reg [15:0] inhistory = 16'h0000;
    reg in_detected = 1'b0;
    always @(posedge clk) begin
        inhistory <= { inhistory[14:0] , in_syn };
        if (inhistory == 16'b0011111111111111)
          in_detected <= 1'b1;
        else
          in_detected <= 1'b0;
    end

    wire seg_num; // not reg!

    Counter counter(
```

```
        .clk(clk),
        // output clock/counter
        .count_4(seg_num)
        );

    reg [31:0] firstNum;
    reg [31:0] secondNum;

    initial firstNum = 0;
    initial secondNum = 0;

    wire [31:0] currPC, nextPC, rsData, rtData, dbData,
          alu_res;
    wire [4:0] rs, rt;

    CPU cpu(
        // input
        .clk(in_detected),
        .reset(reset),
        // output
        .currPC(currPC),
        .nextPC(nextPC),
        .rs(rs),
        .rt(rt),
        .rsData(rsData),
        .rtData(rtData),
        .dbData(dbData),
        .alu_res(alu_res)
        );

    always @(SW_in) begin
        case (SW_in)
            2'b00: begin
                firstNum <= currPC;
                secondNum <= nextPC;
                end
            2'b01: begin
                firstNum <= {{27{1'b0}},rs};
                secondNum <= rsData;
                end
            2'b10: begin
                firstNum <= {{27{1'b0}},rt};
                secondNum <= rtData;
                end
            2'b11: begin
                firstNum <= alu_res;
                secondNum <= dbData;
                end
        endcase
    end

    SegDisplay seg1(
        .data(firstNum[7:4])
        // .dispcode
        );

    SegDisplay seg2(
        .data(firstNum[3:0])
        // .dispcode
        );

    SegDisplay seg3(
        .data(secondNum[7:4])
        // .dispcode
        );

    SegDisplay seg4(
        .data(secondNum[3:0])
        // .dispcode
        );

    always @ (seg_num or firstNum or secondNum)
        case (seg_num)
            0: begin
                out = 4'b1110;
                dispcode = seg4.dispcode;
                end
            1: begin
                out = 4'b1101;
                dispcode = seg3.dispcode;
                end
            2: begin
                out = 4'b1011;
                dispcode = seg2.dispcode;
                end
            3: begin
                out = 4'b0111;
                dispcode = seg1.dispcode;
                end
        endcase

endmodule
```

XIV. 限制文件(constraints.xdc)

```
set_property PACKAGE_PIN W5 [get_ports clk]
set_property PACKAGE_PIN T17 [get_ports clk_cpu]
set_property PACKAGE_PIN V17 [get_ports reset]
set_property PACKAGE_PIN R2 [get_ports {SW_in[1]}]
set_property PACKAGE_PIN T1 [get_ports {SW_in[0]}]
set_property PACKAGE_PIN W4 [get_ports {out[3]}]
set_property PACKAGE_PIN V4 [get_ports {out[2]}]
set_property PACKAGE_PIN U4 [get_ports {out[1]}]
set_property PACKAGE_PIN U2 [get_ports {out[0]}]
set_property PACKAGE_PIN W7 [get_ports {dispcode[6]}]
set_property PACKAGE_PIN W6 [get_ports {dispcode[5]}]
set_property PACKAGE_PIN U8 [get_ports {dispcode[4]}]
set_property PACKAGE_PIN V8 [get_ports {dispcode[3]}]
set_property PACKAGE_PIN U5 [get_ports {dispcode[2]}]
set_property PACKAGE_PIN V5 [get_ports {dispcode[1]}]
set_property PACKAGE_PIN U7 [get_ports {dispcode[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk_cpu]
set_property IOSTANDARD LVCMOS33 [get_ports reset]
set_property IOSTANDARD LVCMOS33 [get_ports {SW_in[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW_in[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {out[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {out[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {out[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {out[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode
      [6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode
      [5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode
      [4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode
      [3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode
      [2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode
      [1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode
      [0]}]
```