



# 《计算机组成原理实验》

# 实验报告

## (实验三)

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 12 月 10 日

# 成绩：

---

## 实验三：多周期CPU设计与实现

### 一、 实验目的

1. 认识和掌握多周期数据通路图的构成、原理及其设计方法
2. 掌握多周期CPU的实现方法，代码实现方法
3. 编写一个编译器，将MIPS汇编程序编译为二进制机器码
4. 掌握多周期CPU的测试方法
5. 掌握多周期CPU的实现方法

### 二、 实验内容

设计一个多周期CPU，使其至少能实现以下指令功能操作。指令与格式如表1所示：

表 1: 基本MIPS指令及其格式

	指令	op	rs	rt	rd	sham/func
算术运算	add rd, rs, rt	000000	rs(5位)	rt(5位)	rd(5位)	reserved
	sub rd, rs, rt	000001	rs(5位)	rt(5位)	rd(5位)	reserved
	addiu rt, rs, imm	000010	rs(5位)	rt(5位)	imm16	
逻辑运算	and rd, rs, rt	010000	rs(5位)	rt(5位)	rd(5位)	reserved
	andi rt, rs, imm	010001	rs(5位)	rt(5位)	imm16	
	ori rt, rs, imm	010010	rs(5位)	rt(5位)	imm16	
	xori rt, rs, imm	010011	rs(5位)	rt(5位)	imm16	
移位	sll rd, rt, sa	011000	reserved	rt(5位)	rd(5位)	sa(5位)
比较	slti rt, rs, imm	100110	rs(5位)	rt(5位)	imm16	
	slt rd, rs, rt	100111	rs(5位)	rt(5位)	rd(5位)	reserved
访存	sw rt, imm(rs)	110000	rs(5位)	rt(5位)	imm16	
	lw rt, imm(rs)	110001	rs(5位)	rt(5位)	imm16	
分支	beq rs, rt, imm	110100	rs(5位)	rt(5位)	imm16	
	bne rs, rt, imm	110101	rs(5位)	rt(5位)	imm16	
	bltz rs, imm	110110	rs(5位)	00000	imm16	
跳转	j addr	111000	addr[27:2]			
	jr rs	111001	rs(5位)	reserved	reserved	reserved
调用子程序	jal addr	111010	addr[27:2]			
停机	halt	111111	0000000000000000000000000000(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. 多周期CPU

多周期CPU指的是将整个CPU的执行过程分成几个阶段，每个阶段用一个时钟去完成，然后开始下一条指令的执行，而每种指令执行时所用的时钟数不尽相同。多周期CPU在处理指令时，需要经过以下五个阶段（见图1）：

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，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。
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。

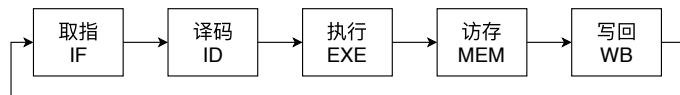


图 1: 多周期CPU的多个阶段

##### ii. 数据通路及控制信号

基本数据通路见图2，控制信号见表2。

注意，图2上增加IR指令寄存器，目的是使指令代码保持稳定；PC写使能控制信号PCWrite则确保了PC可以适时修改。ADR、BDR、ALUDR、DBDR四个寄存器都不需要写使能信号，其作用是切分数据通路，将大组合逻辑切分为若干个小组合逻辑，大延迟变为多个分段小延迟。

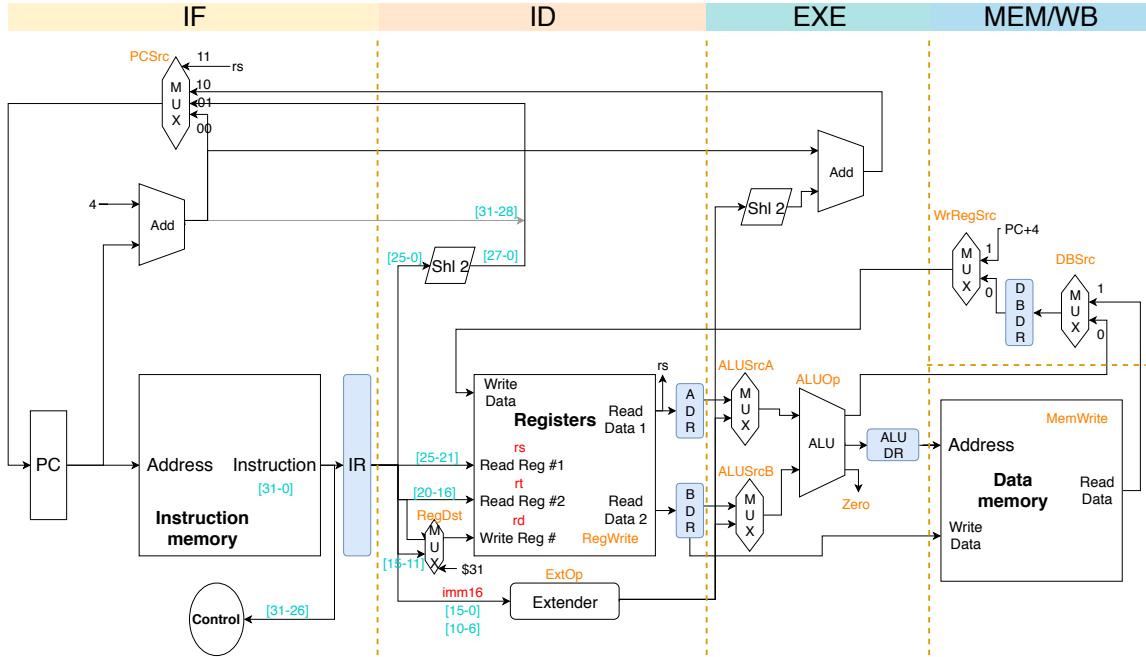


图 2: 基本数据通路

表 2: 控制信号

控制信号名	状态0	状态1	控制信号名	状态	操作
Reset	初始化PC为0	PC接收新地址	RegDst[1:0]	00	写入寄存器地址来自rt字段
PCWrite	PC不更改, halt	PC更改		01	写入寄存器地址来自rd字段
RegWrite	寄存器不可写	寄存器可写		10	\$31
ALUSrcA	寄存器rs内容	立即数sa	PCSrc[1:0]	00	PC+4
ALUSrcB	寄存器rt内容	立即数imm		01	跳转
ExtOp	零扩展	符号扩展		10	分支
DBSrc	ALU输出	内存读取		11	rs寄存器的内容
WrRegSrc	DataBus	PC+4	ALUOp[2:0]	见表	
MemWrite	内存不可写	内存可写			

### iii. MIPS指令格式

MIPS指令可分为以下三种格式，见图3。

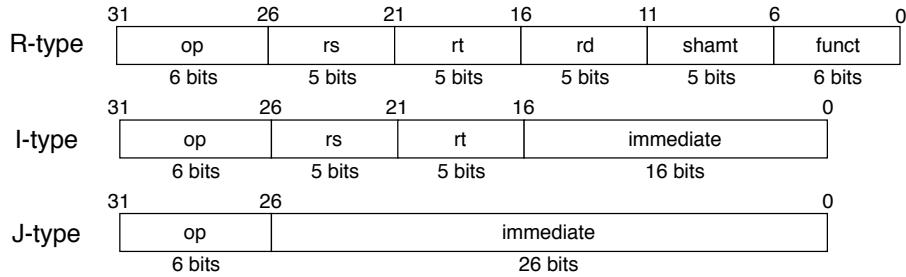


图 3: MIPS指令类型

其中各字段缩写含义如表3所示。

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

## 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4最左红色通路, 下文将不再提及。

指令通路均用红色或蓝色折线标注出来, 见图4到图12。

### i. 算术逻辑运算

- add/sub/and/or/sl<sub>t</sub>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4, 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。
- addiu/andi/ori/xori/sl<sub>t</sub>i指令均为I类型, 数据通路相同, 同样是ALU操作码不同。见图5, 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/sl<sub>t</sub>i进行符号扩展, andi/ori进行零扩展。
- sll指令为R类型, 但是指令格式比较特殊。见图6, 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。

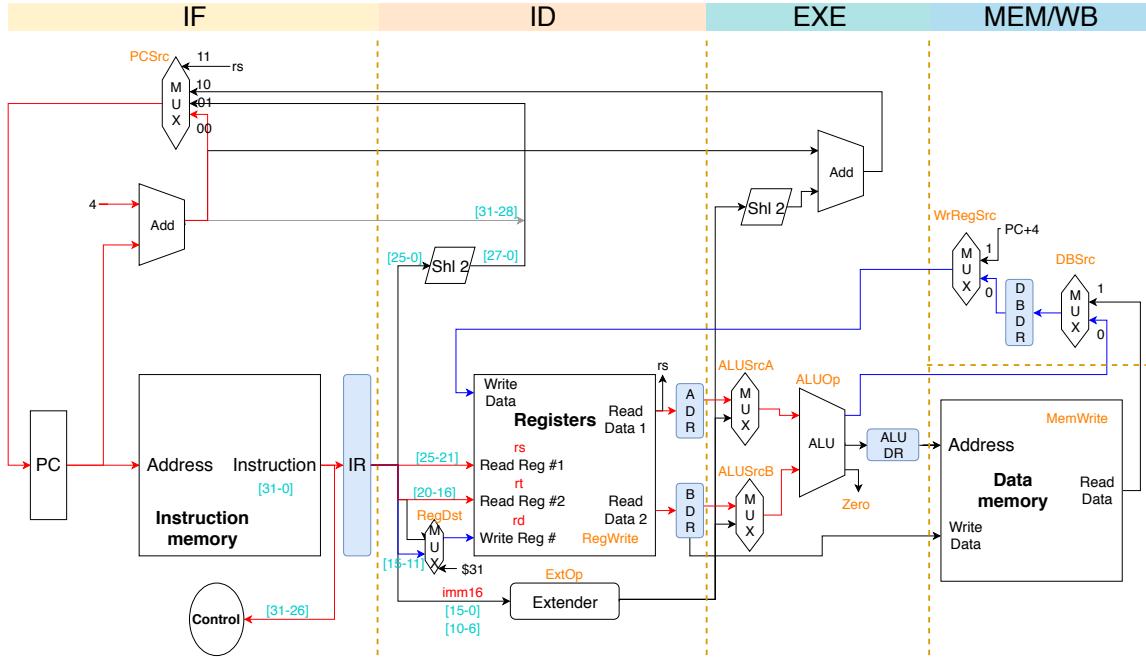


图 4: Add/sub/and/or/slt 通路

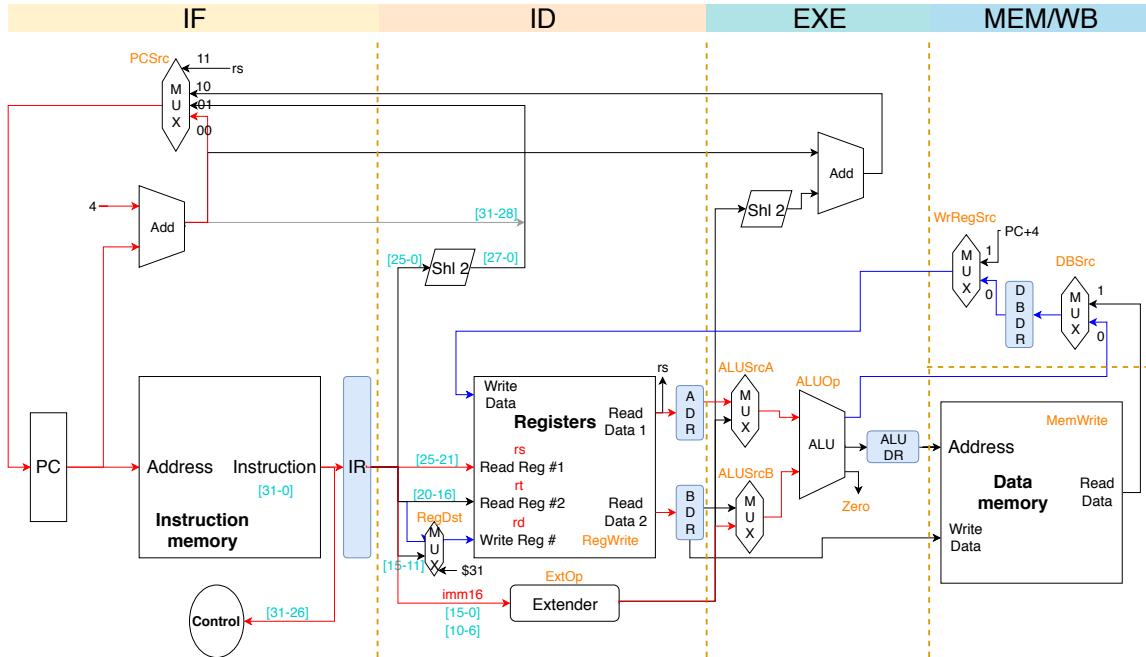


图 5: Addiu/andi/ori/xori/slti 通路

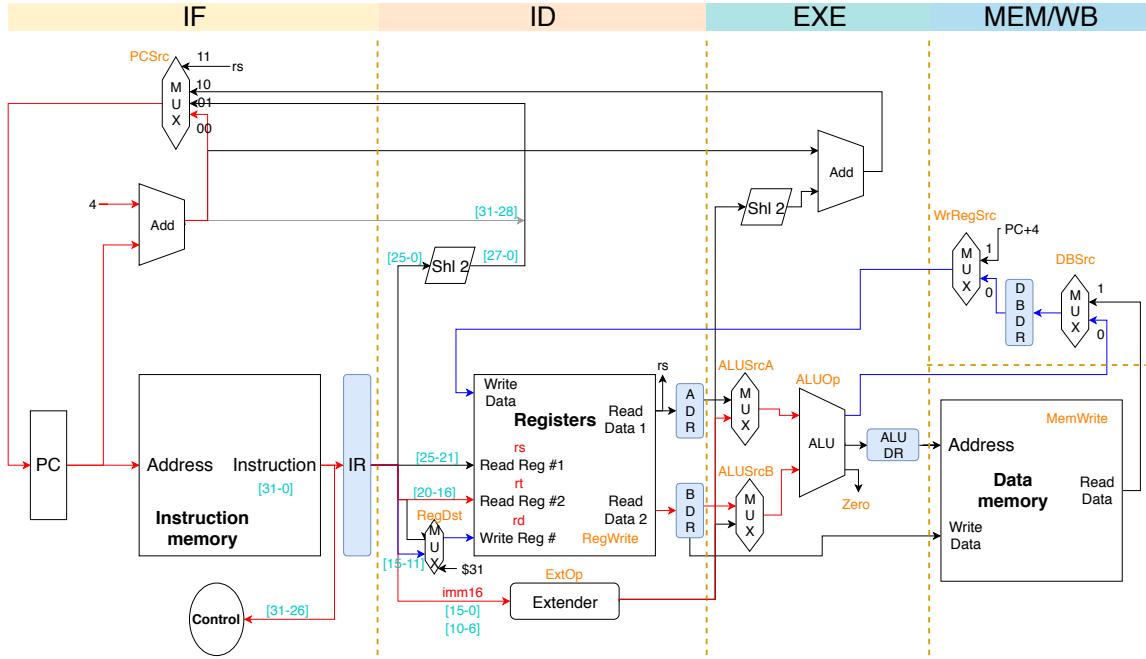


图 6: sll通路

## ii. 访存

1. sw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, 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8, 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, 将内存的内容写入rt寄存器。

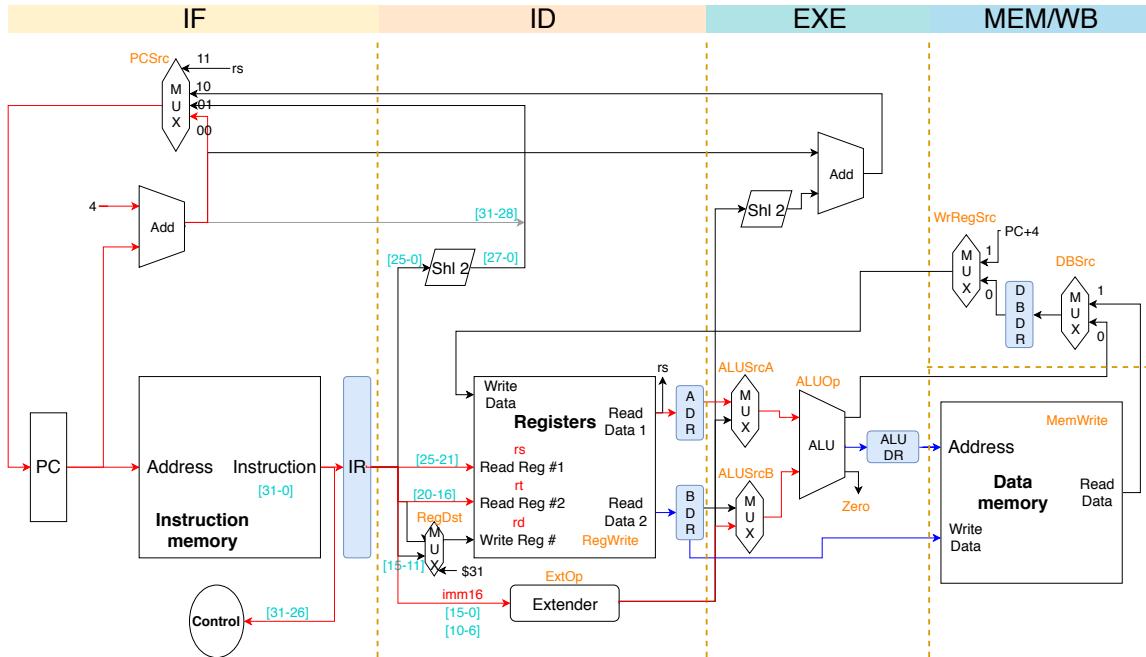


图 7: sw通路

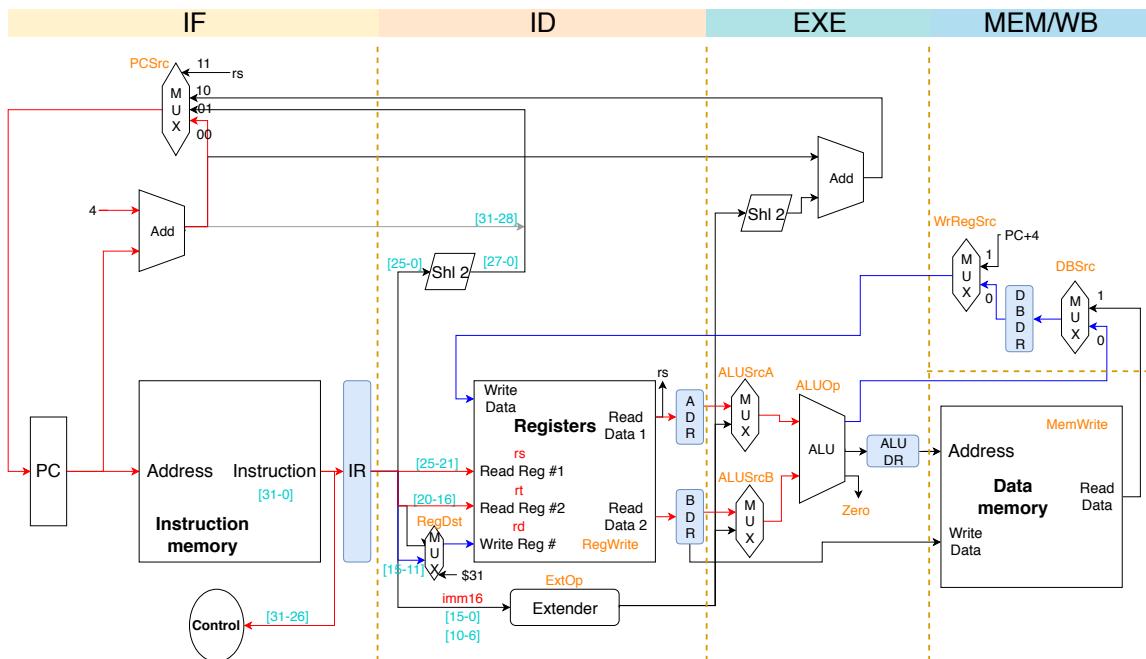


图 8: lw通路

### 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9，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，结合Zero的值得到是否执行分支跳转指令。
2. j/jal为J类型。j指令见图10，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。jal的情况类似，见图11，但注意需要将原来的PC+4写入31号寄存器，以便函数调用后返回。
3. jr为R类型，常用于函数调用后返回，见图12，需要读出rs寄存器的内容，并作为PC的跳转值

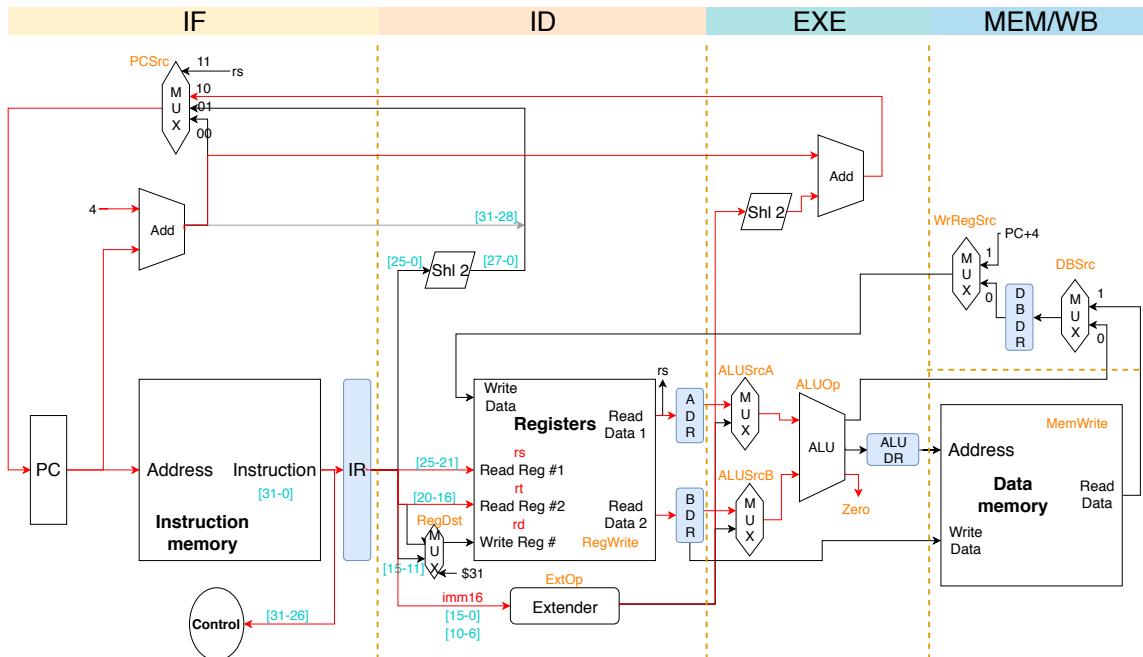


图 9: beq/bne/bltz通路

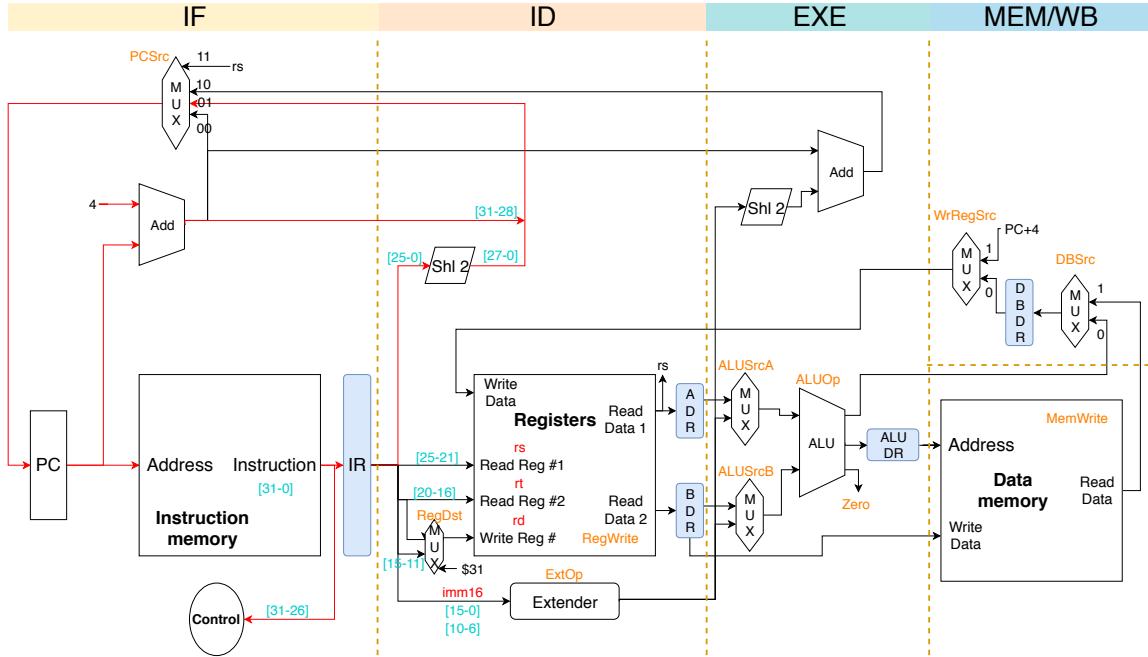


图 10: j通路

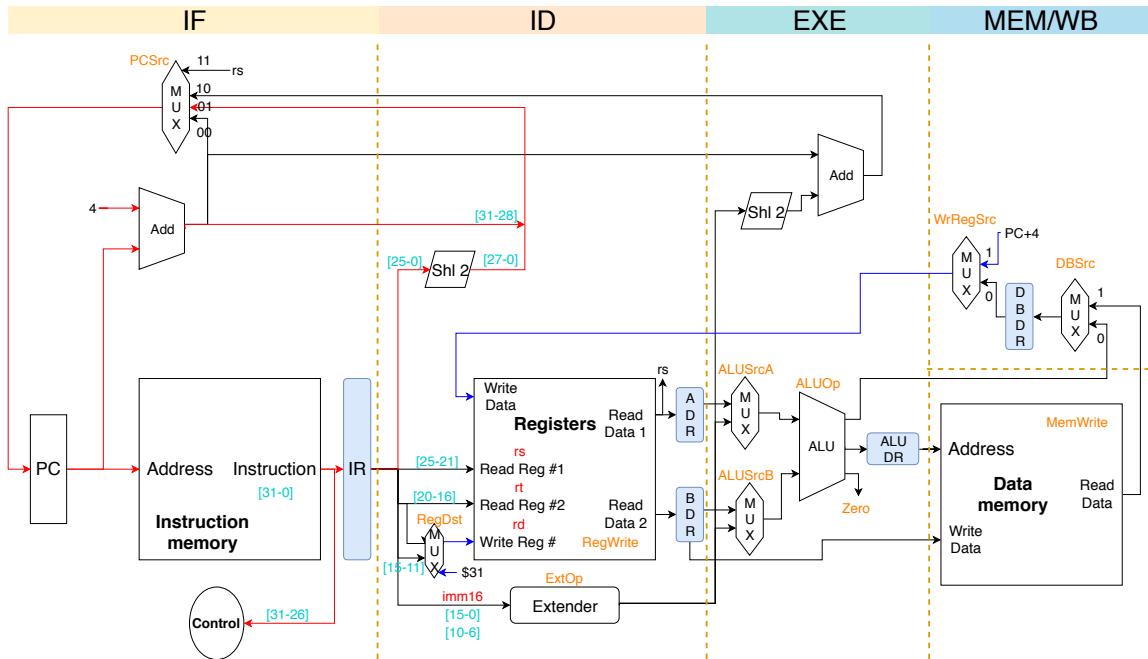


图 11: jal通路

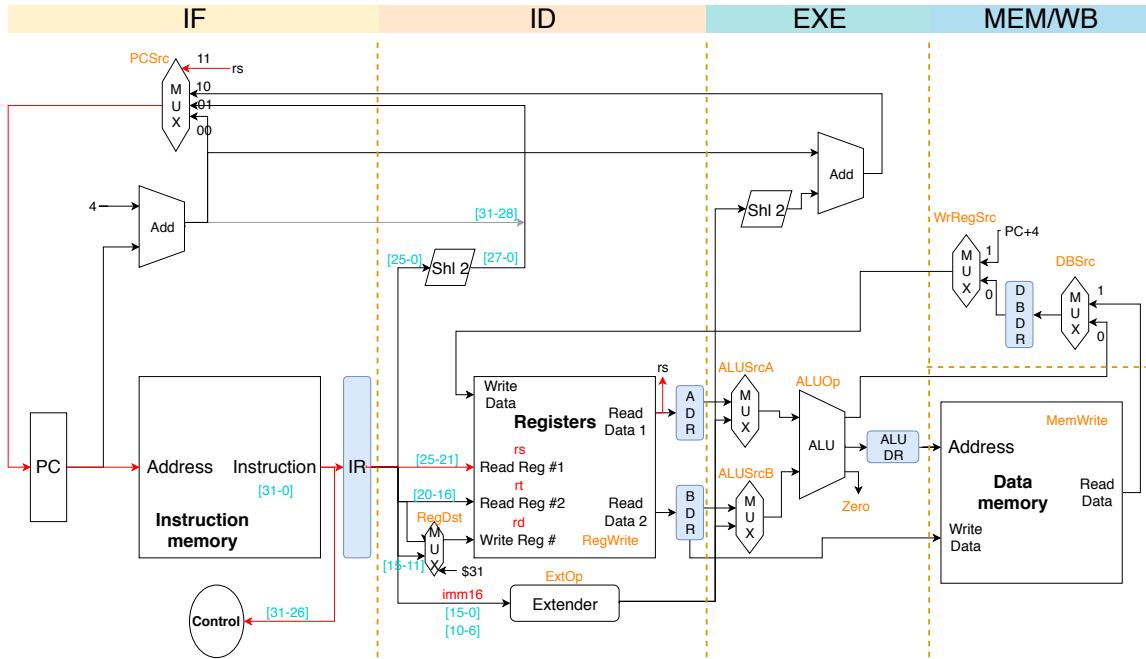


图 12: jr通路

#### 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9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MIPS程序的编译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。
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。(实际操作时是置为 $-4 = 0xFFFFFFF$ C，为使第一条指令能够正常执行完全部状态)
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）；且需要构造2输入、3输入、4输入三种不同的MUX。
4. ALU的功能见表4。
5. 控制单元的设计见第4.4节。

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	$Y = (((A < B) \&\& (A[31] == B[31]))$ $\quad \quad \quad \mid ((A[31] == 1 \&\& B[31] == 0)))$ $\quad \quad \quad ? 1 : 0$	比较有符号数
111	$Y = A \oplus B$	逻辑异或

#### IV. 控制单元

多周期CPU最重要的一点在于状态转移的分析，如图13所示。

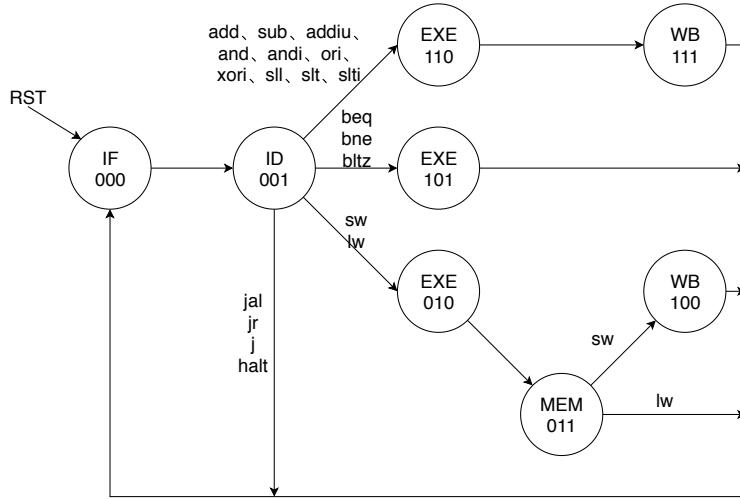


图 13: 多周期CPU状态转移图

由前面的分析，可以得到控制单元的编码表，见表5。注意表5仅仅指明了一条指令的执行过程中全部控制信号的取值，具体到各个阶段，则需结合当前状态判断控制信号是否有效。如只在IF阶段允许PC的修改，即PCWrite为1；而只在WB阶段可以进行寄存器的写操作，即RegWr为1。具体实现参见第9章控制单元一节。

表 5: 指令控制器编码

	指令	op	RegDst	ExtSel	RegWrite	ALUSrcA/B	ALUOp	DBSrc	WrRegSrc	MemWrite	PCSrc	PCWrite
算术运算	add rd, rs, rt	000000	1	x	1	00	000	0	0	0	00	1
	sub rd, rs, rt	000001	1	x	1	00	001	0	0	0	00	1
	addiu rt, rs, imm	000010	0	1	1	01	000	0	0	0	00	1
	and rd, rs, rt	010000	1	x	1	00	100	0	0	0	00	1
逻辑运算	andi rt, rs, imm	010001	0	0	1	01	100	0	0	0	00	1
	ori rt, rs, imm	010010	0	0	1	01	011	0	0	0	00	1
	xori rt, rs, imm	010011	0	0	1	01	111	0	0	0	00	1
移位	sll rd, rt, sa	011000	1	x	1	10	010	0	0	0	00	1
	slti rt, rs, imm	100110	0	1	1	01	110	0	0	0	00	1
	slt rd, rs, rt	100111	1	x	1	00	110	0	0	0	00	1
比较	sw rt, imm(rs)	110000	x	1	0	01	000	x	0	1	00	1
	lw rt, imm(rs)	110001	0	1	1	01	000	1	0	0	00	1
分支	beq rs, rt, imm	110100	x	1	0	00	001	x	0	0	10	1
	bne rs, rt, imm	110101	x	1	0	00	001	x	0	0	10	1
	bltz rs, imm	110110	x	1	0	00	110	x	0	0	10	1
跳转	j addr	111000	x	x	0	xx	x	x	0	0	01	1
	jr rs	111001	x	x	0	0x	x	x	0	0	11	1
调用子程序	jal addr	111010	2	x	1	xx	x	x	1	0	01	1
停机	halt	111111	x	x	0	xx	x	x	x	0	xx	0

## 五、 实验步骤

### I. 简易编译器

本次实验用Python写了一个简单的编译器来实现MIPS代码到二进制代码的转换。主要采用正则表达式库(re)对输入的字符串进行处理。基本实现思路如下：

1. 逐行读入汇编源文件，遇到非指令行（如注释）则直接丢弃，其余行送入parse函数。
2. 在parse函数内判断指令名称，并依据不同的指令格式分别进行字符串的分割和处理，这里采用了正则表达式进行分割。
3. 将操作、寄存器编号、立即数等全部转化为二进制。不够位数的用0补足，负数要用补码表示。
4. 将这些二进制数连接起来，得到32位的二进制字符串，返回
5. 逐行输出二进制编码和十六进制编码，并写入文件，注意这里采用小端存储（每8位作为一个字节/单位）

完整程序附在第9章编译器一节。

### II. 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。

### III. 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。同时添加了一个State\_in信号，用板上的开关SW13(U1)进行控制，显示当前的状态编码，便于上板调试。

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

### IV. 七段数码管显示电路

基本实现步骤如下：

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	xori \$3,\$2,8	010011	00010	00011	0000	0000	0000	1000	0x4c430008
0x0000000C	sub \$4,\$3,\$1	000001	00011	00001	0010	0000	0000	0000	0x04612000
0x00000010	and \$5,\$4,\$2	010000	00100	00010	0010	1000	0000	0000	0x40822800
0x00000014	sll \$5,\$5,2	011000	00000	00101	0010	1000	1000	0000	0x60052880
0x00000018	beq \$5,\$1,-2(=,转14)	110100	00101	00001	1111	1111	1111	1110	0xd0a1fffe
0x0000001C	jal 0x0000050	111010	00000	00000	0000	0000	0001	0100	0xe8000014
0x00000020	slt \$8,\$13,\$1	100111	01101	00001	0100	0000	0000	0000	0x9da14000
0x00000024	addiu \$14,\$0,-2	000010	00000	01110	1111	1111	1111	1110	0x080efffe
0x00000028	slt \$9,\$8,\$14	100111	01000	01110	0100	1000	0000	0000	0x9d0e4800
0x0000002C	slti \$10,\$9,2	100110	01001	01010	0000	0000	0000	0010	0x992a0002
0x00000030	slti \$11,\$10,0	100110	01010	01011	0000	0000	0000	0000	0x994b0000
0x00000034	add \$11,\$11,\$10	000000	01011	01010	0101	1000	0000	0000	0x016a5800
0x00000038	bne \$11,\$2,-2 ( $\neq$ ,转34)	110101	01011	00010	1111	1111	1111	1110	0xd562fffe
0x0000003C	addiu \$12,\$0,-2	000010	00000	01100	1111	1111	1111	1110	0x080cfffe
0x00000040	addiu \$12,\$12,1	000010	01100	01100	0000	0000	0000	0001	0x098c0001
0x00000044	bltz \$12,-2 (j0,转40)	110110	01100	00000	1111	1111	1111	1110	0xd980fffe
0x00000048	andi \$12,\$2,2	010001	00010	01100	0000	0000	0000	0010	0x444c0002
0x0000004C	j 0x000005C	111000	00000	00000	0000	0000	0001	0111	0xe0000017
0x00000050	sw \$2,4(\$1)	110000	00001	00010	0000	0000	0000	0100	0xc0220004
0x00000054	lw \$13,4(\$1)	110001	00001	01101	0000	0000	0000	0100	0xc42d0004
0x00000058	jr \$31	111001	11111	00000	0000	0000	0000	0000	0xe7e00000
0x0000005C	halt	111111	00000	00000	0000	0000	0000	0000	0xfc000000

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9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。

## V. 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。

### 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9章写板电路一节。

## VI. 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

## VII. 代码层次结构

见图14，其中Register用于生成五个阶段寄存器的实例。

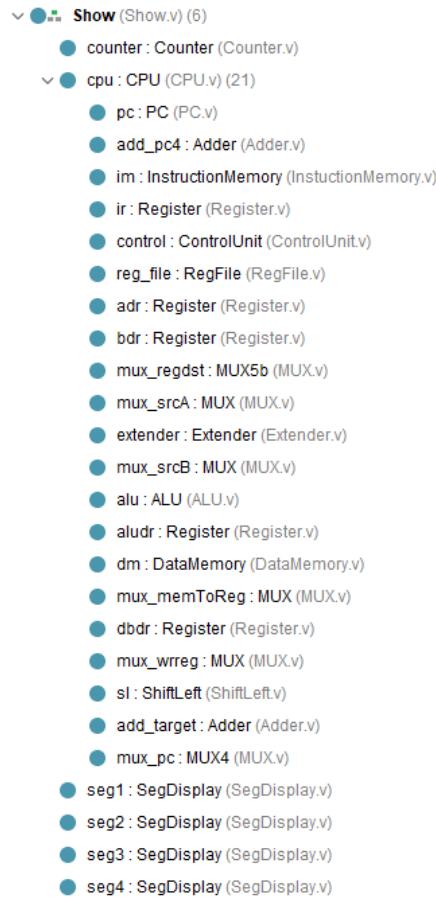


图 14: 代码层次结构

## 六、结果与分析

### I. 仿真模拟

设置时钟周期为100ns，初始30ns为准备时间，然后将Reset设为1使其开始工作。仿真结果见图15到图26，每条指令的说明均已附在波形图之下。

1. 0x00 addiu \$1,\$0,8

IF(0)状态的后半周期读入指令，并写入指令寄存器IR；WB(7)状态的后半周期将结果8写入Reg[1]

2. 0x04 ori \$2,\$0,2

WB(7)状态后半周期将结果2写入Reg[2]

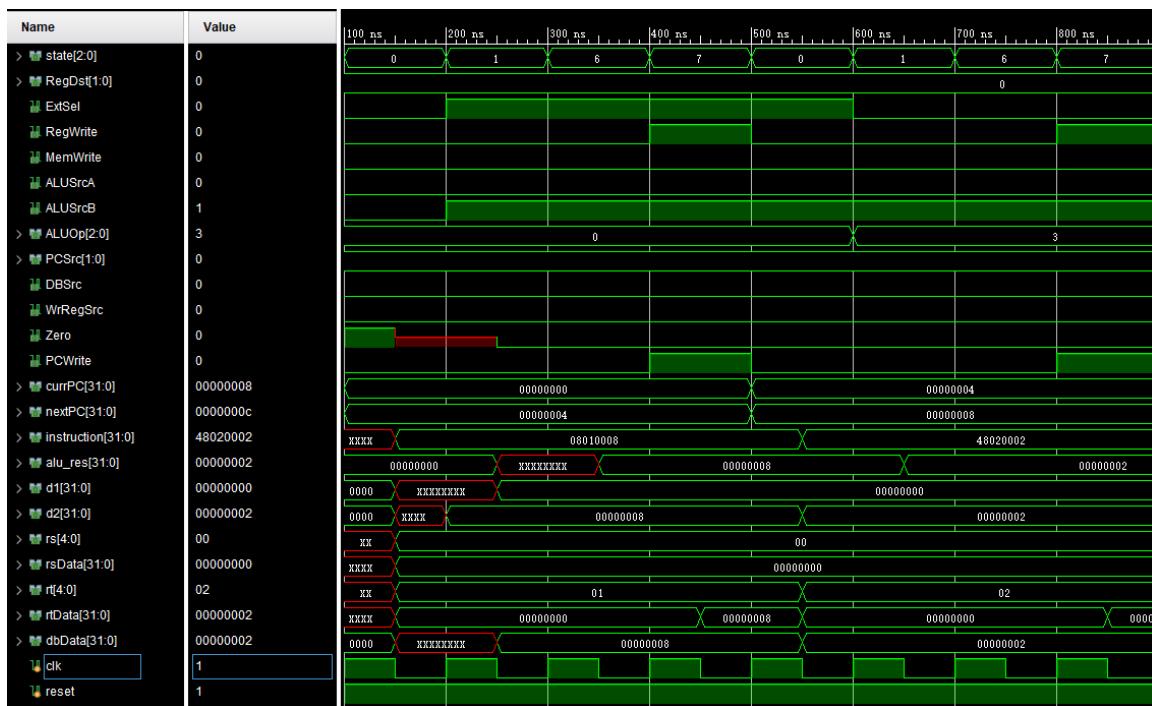


图 15: 波形图1

3. 0x08 xori \$3,\$2,8

ID(1)状态读入Reg[2]=2，与立即数8异或，即 $1000 \text{ xor } 0010 = 1010 = (10)_{10}$ ；WB(7)状态将结果(A)<sub>16</sub>写入Reg[3]

4. 0x0C sub \$4,\$3,\$1

WB(7)状态将结果 $\text{Reg}[3] - \text{Reg}[1] = 10 - 8 = 2$ 写入Reg[4]

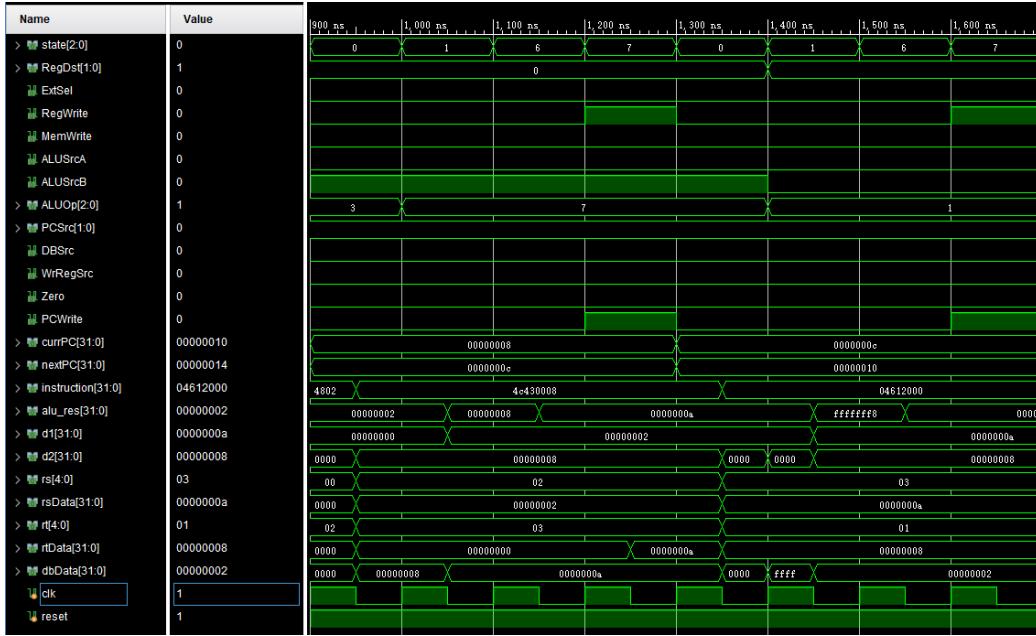


图 16: 波形图2

## 5. 0x10 and \$5,\$4,\$2

WB(7)状态将结果Reg[4]&Reg[2] = 2&2 = 2写入Reg[5]

## 6. 0x14 sll \$5,\$5,2

WB(7)状态将结果Reg[5] << 2 = 2 << 2 = 8写入Reg[5]

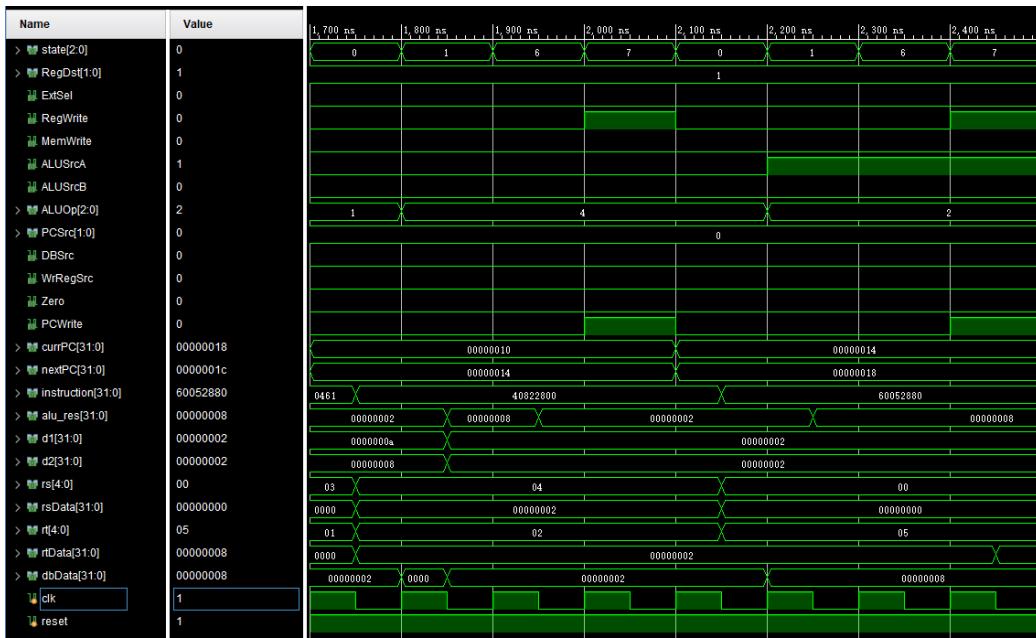


图 17: 波形图3

## 7. 0x18 beq \$5,\$1,-2

EXE(5)状态ALU结果为 $\text{Reg}[5] - \text{Reg}[1] = 8 - 8 = 0 == 0$ , 跳转回 $0x14$

8.  $0x14 \text{ sll } \$5, \$5, 2$

WB(7)状态将结果 $\text{Reg}[5] << 2 = 8 << 2 = 32 = (20)_{16}$ 写入 $\text{Reg}[5]$

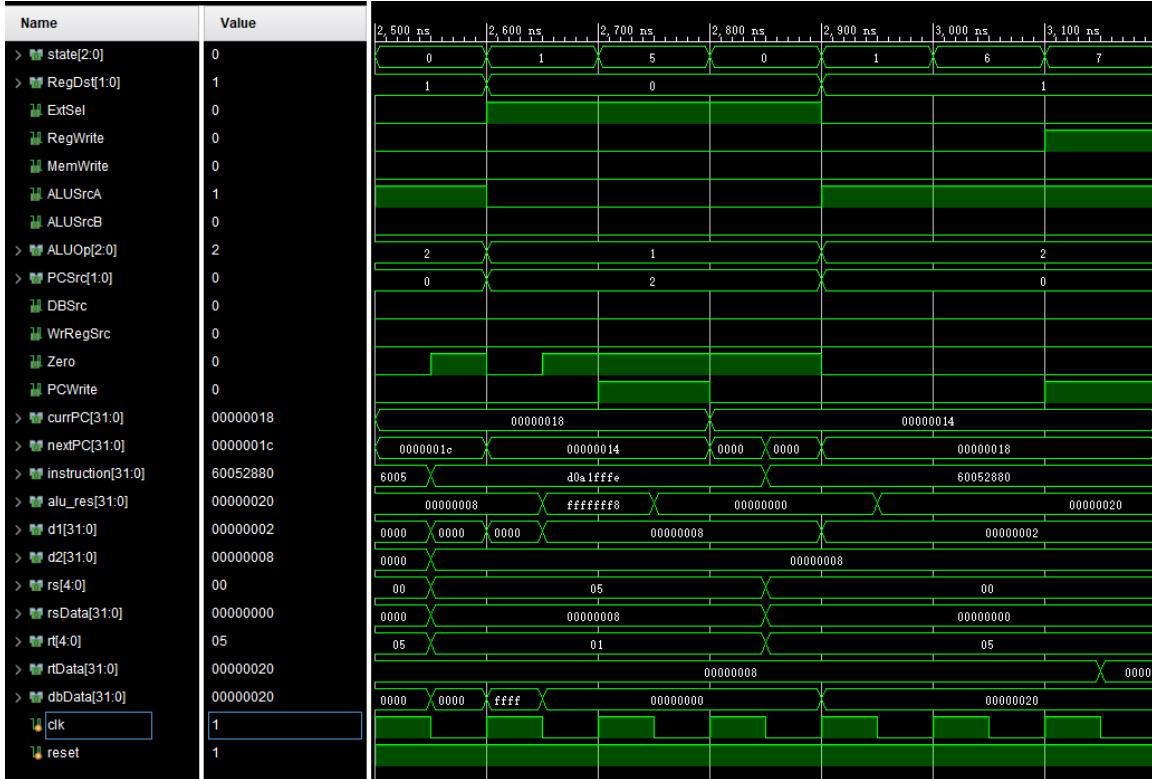


图 18: 波形图4

9.  $0x18 \text{ beq } \$5, \$1, -2$

EXE(5)状态ALU结果为 $\text{Reg}[5] - \text{Reg}[1] = 32 - 8 = 24 = (18)_{16} \neq 0$ , 不跳转, 执行下一指令

10.  $0x1C \text{ jal } 0x00000050$

调用子程序, IF(0)状态更新下一PC值为 $0x50$ ; ID(1)状态RegDst为2, 对Reg[31]写入原来的PC+4, 即 $0x20$ , 故图中ALU在ID(1)状态的后半周期值会改变, 因为RegFile写入了新的值

11.  $0x50 \text{ sw } \$2, 4(\$1)$

将 $\text{Reg}[2]=2$ 存入 $\text{Mem}[\text{Reg}[1] + 4] = \text{Reg}[12] = \text{Mem}[(C)_{16}]$

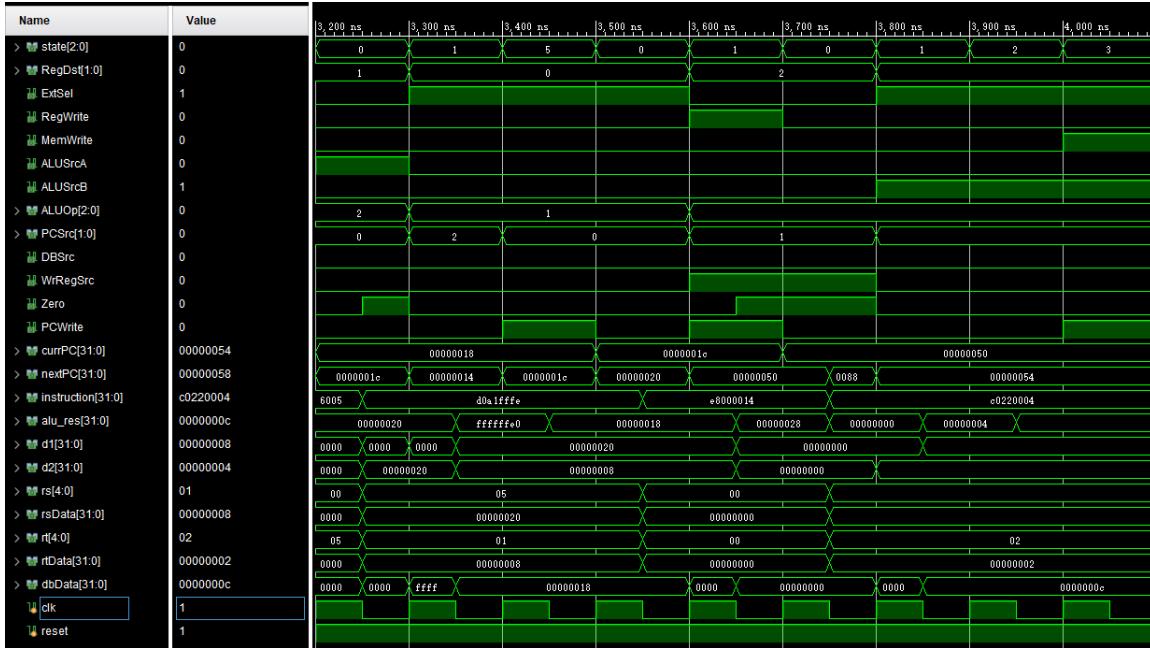


图 19: 波形图5

12. 0x54 lw \$13,4(\$1)

将Mem[12] = 2取出，在WB(4)状态存入Reg[13]

13. 0x58 jr \$31

IF(0)状态译码后即可取出Reg[31]=0x20，进而更新下一PC值为0x20

14. 0x20 slt \$8,\$13,\$1

在EXE(6)阶段算得Reg[13] < Reg[1] = 2 < 8 = 1，在WB(7)存入Reg[8]

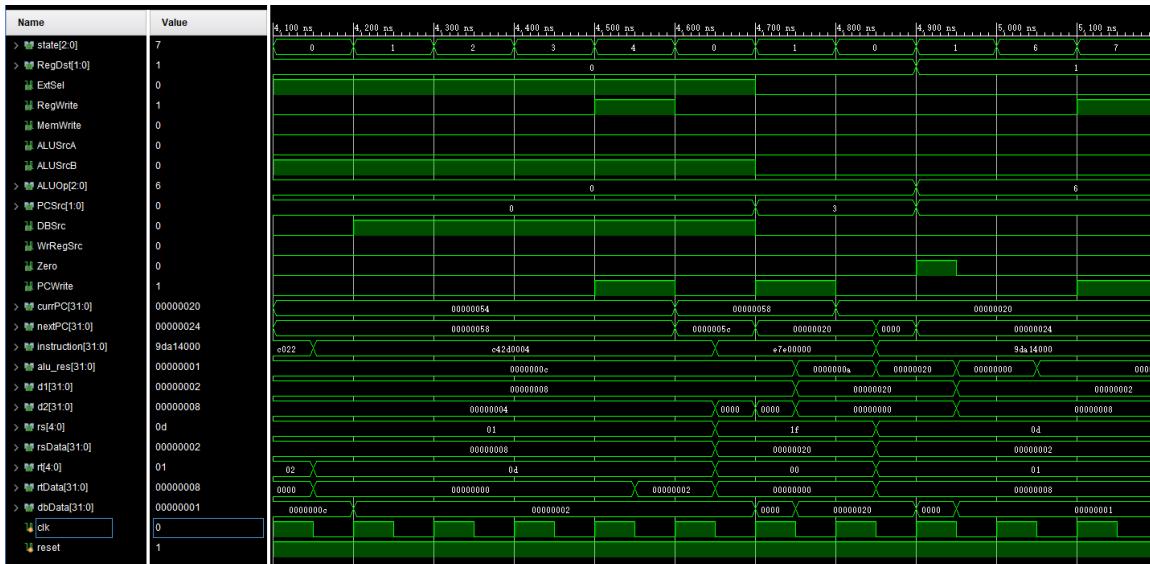


图 20: 波形图6

15. 0x24 addiu \$14,\$0,-2

在EXE(6)阶段算得Reg[0]+(-2)=0xFFFFFFF, 并在WB(7)存入Reg[14]

16. 0x28 slt \$9,\$8,\$14

在EXE(6)阶段算得Reg[8] < Reg[14] = 1 < -2 = 0, 并在WB(7)存入Reg[9]

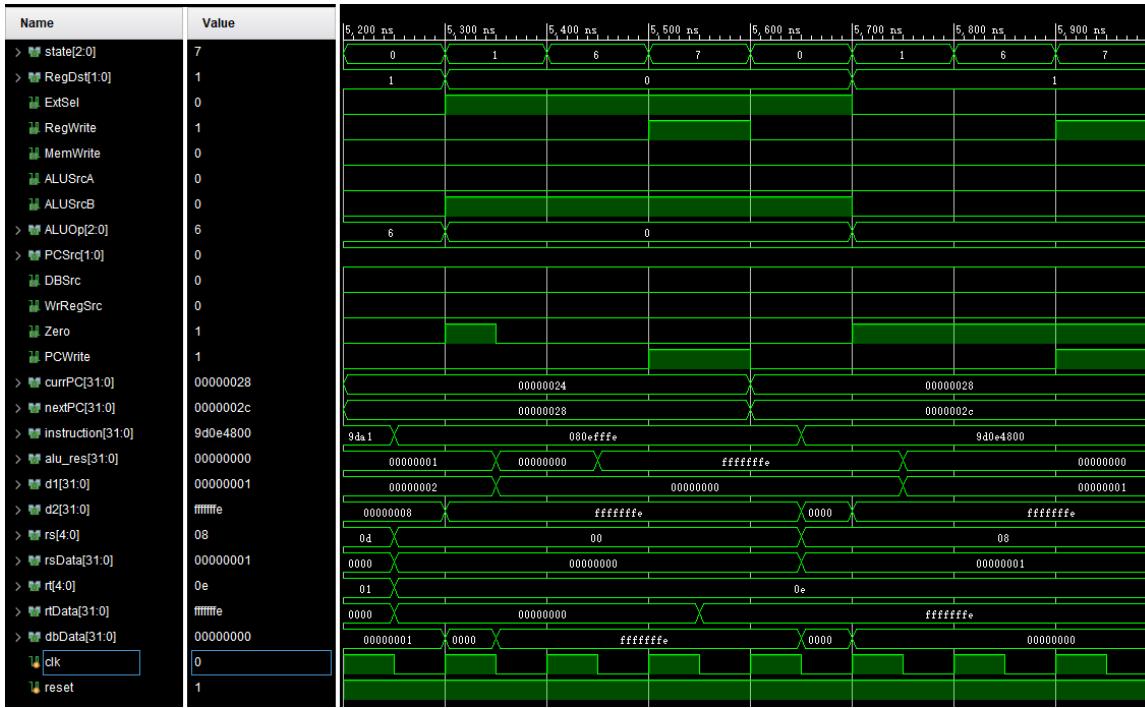


图 21: 波形图7

17. 0x2C slti \$10,\$9,2

在EXE(6)阶段算得Reg[9] < 2 = 0 < 2 = 1, 并在WB(7)存入Reg[10]

18. 0x30 slti \$11,\$10,0

在EXE(6)阶段算得Reg[10] <= 0 = 1 < 0 = 0, 并在WB(7)存入Reg[11]



图 22: 波形图8

19. 0x34 add \$11,\$11,\$10

在EXE(6)阶段算得Reg[11]+Reg[10]=0+1=1，并在WB(7)存入Reg[11]

20. 0x38 bne \$11,\$2,-2

在EXE(5)阶段算得Reg[11]-Reg[2]=1-2=-1=0xFFFFFFFF，不等于，跳转回0x34

21. 0x34 add \$11,\$11,\$10

在EXE(6)阶段算得Reg[11]+Reg[10]=1+1=2，并在WB(7)存入Reg[11]

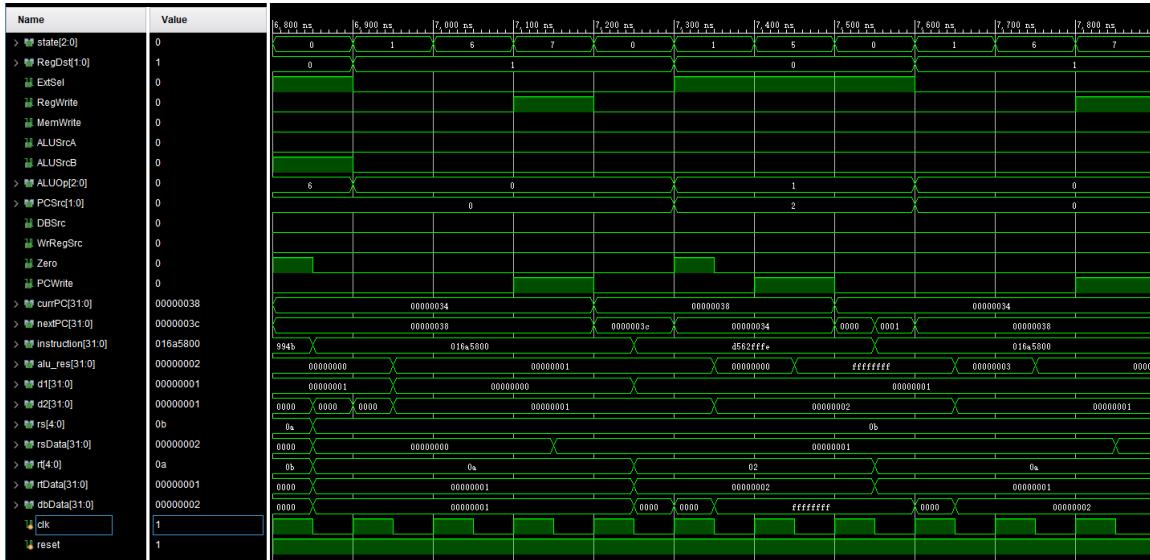


图 23: 波形图9

22. 0x38 bne \$11,\$2,-2

在EXE(5)阶段算得Reg[11]-Reg[2]=2-2=0, 相等, 不跳转, 执行下条指令

23. 0x3C addiu \$12,\$0,-2

在EXE(6)阶段算得Reg[0]+(-2)=0xFFFFFFFF, 并在WB(7)存入Reg[12]

24. 0x40 addiu \$12,\$12,1

在EXE(6)阶段算得Reg[12]+1=0xFFFFFFFF, 并在WB(7)存入Reg[12]

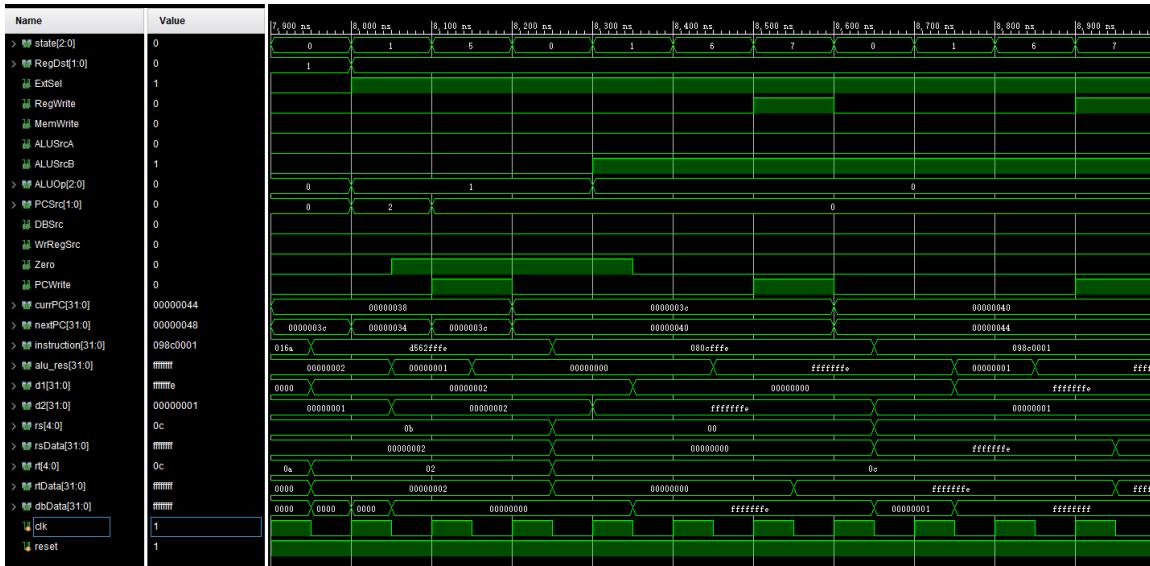


图 24: 波形图10

25. 0x44 bltz \$12,-2

在EXE(5)阶段算得Reg[12] < 0 = -1 < 0 = 1, 故跳转回0x40

26. 0x40 addiu \$12,\$12,1

在EXE(6)阶段算得Reg[12]+1=0, 并在WB(7)存入Reg[12]

27. 0x44 bltz \$12,-2

在EXE(5)阶段算得Reg[12] < 0 = 0 < 0 = 0, 故不跳转, 继续执行下一指令

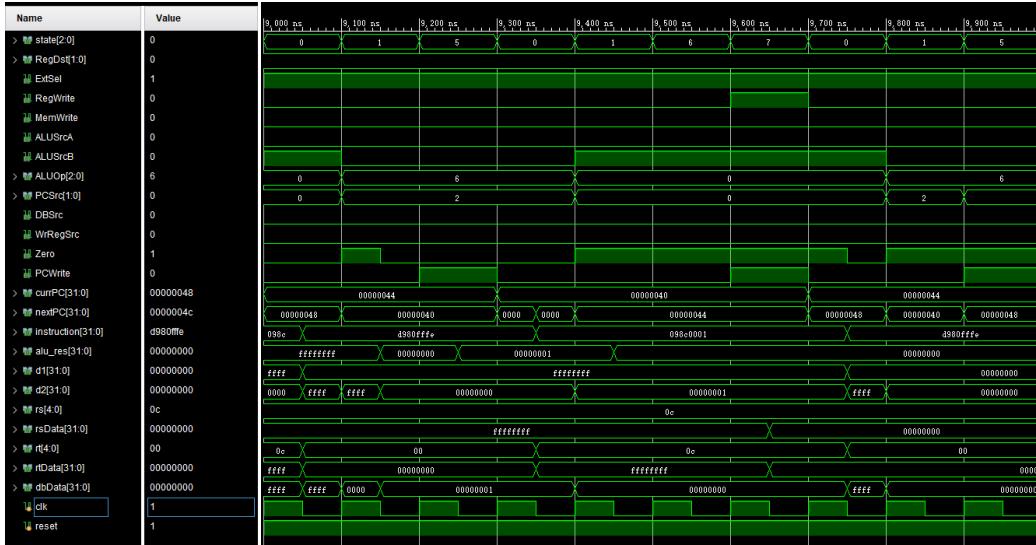


图 25: 波形图11

28. 0x48 andi \$12,\$2,2

在EXE(6)阶段算得Reg[2]&2=2&2=2，并在WB(7)存入Reg[12]

29. 0x50 j 0x0000005C

在IF(0)后半阶段写入指令到IR，译码得出其为跳转指令，更新下一PC值为0x5C

30. halt

在ID(1)阶段求得指令为停机，不再更新PC

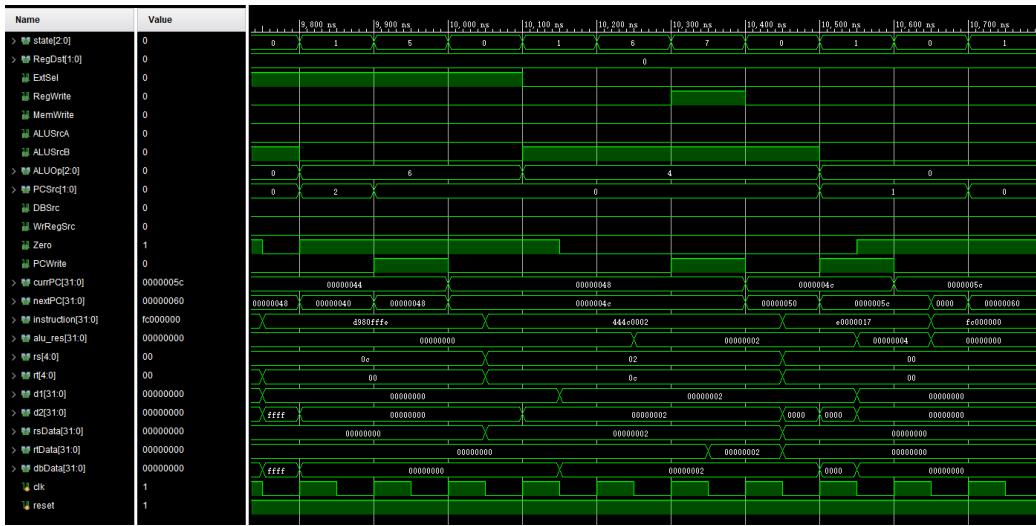


图 26: 波形图12

## II. 上板

几条代表性指令的Basys3板结果如下，其他指令结果见具体实现。从左到右从上到下四幅图依次是：当前/下条PC、rs寄存器地址/值、rt寄存器地址/值、ALU结果输出/DB总线数据。

注意这里显示的都是最后一个状态的值。又由于这些信号都采用wire类型，故写入寄存器后可能导致alu\_res的相应改变。但可以看见上板结果与前面仿真分析的结果相同，这成功证明了程序的正确性。

1. 0x00: addiu \$1, \$0, 8



图 27: 0x00结果

2. 0x14: sll \$5, \$5, 2



图 28: 0x14结果

3. 0x18: beq \$5, \$1, -2



图 29: 0x18结果

4. 0x1C: jal 0x00000050



图 30: 0x1C结果

5. 0x54: lw \$13, 4(\$1)



图 31: 0x54结果

6. 0x58: jr \$31



图 32: 0x58结果

## 七、 实验心得

本次实验在单周期CPU的基础上继续搭建，重用了代码模块，只需分析清除控制信号和状态转移即可。虽然看似比较简单，具体实施时还是遇到了不少问题。

Verilog代码编写阶段：

1. 当调整输入输出端口时，一定要记得将所有相关模块全部修改，如在控制单元中添加一个控制信号，则在CPU、CPU\_sim等模块中均需添加这个控制信号。
2. 在模块输入参数最后常常多了一个逗号，，会导致报错(ISE port connections mix)。
3. 要判断好always @内写的内容，是时序逻辑还是组合逻辑，不能混写，不能写多，也不能写少。
4. 要在每个模块合适的位置对寄存器进行初始化initial，否则在仿真中会输出xxxx。

5. 多位的控制信号一定要记得写位宽[x:y]，这个经常会漏，且不会报错
6. 应在时钟下降沿才进行写入操作

Vivado模拟仿真上板阶段：

1. 延迟设置（如#10）具体上板实施时是会被忽略的
2. 要做好门闩延迟(clock-to-Q)的设置，否则模块还没初始化完就开始操作，会出现很多问题，即对应的状态对应不上，会导致跳转指令无法正常跳转。
3. 竞争与冒险的问题，在原来设计时，状态机只保存了当前状态。由于PC的更新和状态的更新是同步进行的，而由于具体电路不同路径延时不同的问题，导致PC更新时读取到的状态还是旧的，进而PC会被延迟到下一个周期才进行更新。全程的状态都会被延后一个周期，导致结果错误。因而这里很关键的一点在于状态机要设计成存储当前状态(currState)和下一状态(nextState)，当下一状态为000时就要考虑置PCWrite为1，并在当前状态为000时对PC进行更新。这样既解决了设置延时的问题，又解决了竞争与冒险的问题。

总的来说，多周期CPU的设计实验让我更加熟悉了Verilog的语法，熟悉Vivado的使用，同时充分了解硬件设计与软件设计的巨大区别。硬件代码难以debug使得编写程序的效率极其低下，但这时一定不能心急，要冷静下来分析问题所在，并寻求方法解决它。同时，编写硬件程序时要拥有硬件思维，要考虑得更加周密，这些都是与编写软件程序很大的不同。

## 八、期末总结

上了一整个学期的计算机组成原理的实验课，收获了比以往课程多更多的东西，也充分体会到了计算机硬件的乐趣与奥秘。总的来说，收获有以下几点：

1. 明白指令集系统的原理，并学会用MIPS指令集编写简单的汇编程序
2. 深刻了解CPU的工作原理，明白各指令的流向，理解并实现了各条数据通路
3. 充分了解单周期CPU和多周期CPU的工作原理，并用Verilog具体实现了两种CPU，不仅进行了仿真，也进行了具体的上板实施
4. 实现了简单的编译器，明白如何进行解析语义，完成汇编指令到机器二进制指令的转换
5. 更加熟练Verilog代码的编写，掌握Vivado的使用，了解FPGA仿真综合跑实施的整个流程
6. 学会通过观察波形在仿真阶段进行硬件程序的调试，学会通过观察数码管显示的内容在具体实施阶段进行硬件程序的调试
7. 学会组织语言，涵盖所有要点，完成几十页的长实验报告

虽然这学期的计组实验课只让我们实现了三个项目（MIPS汇编语言程序设计、单周期CPU的设计与实现、多周期CPU的设计与实现），但是学到的东西真的非常多，下面结合我这三次项目的经历以及我个人学习计算机组成原理的感受，谈谈我自己的看法。

首先，我充分感受到为什么我国至今没有开发芯片的核心技术，对于芯片设计仍处于起步阶段。这很大程度上是因为芯片设计的投入与回报的不对等性。硬件投入的人力财力都非常巨

大，硬件程序不同于软件，开发周期时间长，debug的难度大。这学期的实验就充分体现了这几点。硬件开发要采用硬件思维，而不能用软件思维，要对问题充分考虑周全，要不然出现了bug会相当麻烦。尽管仿真没有出现问题，但是具体上板实施时却会出现各种各样的问题。这一方面是由于仿真工具无法全真模拟真实的硬件（特别是通路之间微小延时的差距），另一方面则是由于Vivado编译器对代码的检测非常不严格，对于可能出现的问题都选择忽略；更有甚者，连warning也不提醒，典型的例子即写少位宽的问题。尽管等号赋值右侧是3位的数据，但左侧没有声明[2:0]，这种情况下vivado没有明显的报错，直接编译通过，给debug带来了极大的挑战。

其次，我也了解到为什么MIPS指令集往往只用作学校的教学用，而很少有真实的硬件采用MIPS进行搭建，这很大一部分原因在于MIPS本身的局限性。对MIPS指令具体实现时会发现有些指令并不那么自然，要添加很多不必要的部件，设计一条新的数据通路。最明显的例子就是MIPS指令集中sll指令的实现，由于指令格式的不一致（5位的sa却存在最后的11位），CPU内就要添加多余的部件，并专门针对这一条指令进行数据通路的设计。这也在计算机发展了几十年以后，UCB仍要重新开发新的指令集RISC-V，并将其开源。就是因为传统的指令集肩负了太多历史包袱，越来越多新指令的出现，使得指令集变得十分笨重，也使得硬件的实施非常麻烦。

第三点则从编程语言角度说的。单周期和多周期CPU的实现，我分别用Verilog各写了1000+行代码，然后就会发现Verilog的很多问题，比如它没有现代高级语言所拥有的很多设施，如递归的支持、面向对象(OOP)、高阶函数等等，换言之Verilog对于硬件操作的抽象做得比较弱，这直接导致了硬件工程师的开发效率极其低下。举个例子，由于没有参数化(parameterize)的设施，导致我在控制单元(Control Unit, CU)中添加一个新的信号，需要在CPU中的CU实例和输出端口添加该信号，还要在顶层模块的Show和CPU\_sim中添加输出端口，弄得十分麻烦。这还仅仅是两层封装，若遇到多层模块，稍微改动一下底层的模块，上层的建筑全部都得修改，可扩展性十分的差。同样因为这种低效率，UCB才开发了一门新的硬件语言Chisel（封装在Scala语言上），并已经在他们的EECS广泛使用。正是有这门相对高级的语言，才使得他们可以实现敏捷(Agile)开发，并成功制成了Rocket和BOOM等多款CPU核，还成功流片。这种前瞻性和实践性，是国内高校尚无法达到的。

不仅仅是编程语言，编译器也是同样。由于手写RTL层级的硬件描述语言效率太低，所以才要有高层次综合(High Level Synthesis, HLS)，直接将高级语言的程序直接编译为硬件描述语言的程序，但是限于现在的技术，HLS的编译效果不是特别好。至于下层的综合，尽管Vivado已经做了很大的努力去优化它，但其运行的时间依然非常长。像对我们这样一个简单实验中的CPU进行综合操作，一次都要花上5~10分钟，这个开销是相当巨大的。这也反映了硬件开发反馈的效率极低，导致硬件难以设计难以debug、开发周期长。

综上所述，硬件开发流程中依然有很多地方是做得不够完善的，还有极大的改进空间，值得我们继续去研究探索。最后真的非常感谢学院能够开设这门课，现在都倡导软硬件协同设计，不会硬件的计算机工程师就像是缺一条腿，只有清楚地知道CPU底层的硬件实现，才有可能写出更高质量的代码，共同促进软硬件的不断发展。

## 九、 程序清单

## I. MIPS编译器(Python)

## II. CPU主模块

```

39      );
40
41     Adder add_pc4(
42       .A(currPC),
43       .B({{29{1'b0}},3'b100}),
44       .res(pc4)
45     );
46
47     // instruction fetch (IF)
48     InstructionMemory im(
49       .address(currPC),
50       .dataOut(inst)
51     );
52
53     Register ir(
54       .clk(clk),
55       .reset(reset),
56       .in(inst),
57       .out(instruction)
58     );
59
60     // instruction decode (ID)
61     ControlUnit control(
62       .clk(clk),
63       .reset(reset),
64       // input
65       .opcode(opcode),
66       .Zero(Zero),
67       // output
68       .state(state),
69       .RegDst(RegDst),
70       .ExtSel(ExtSel),
71       .RegWrite(RegWrite),
72       .ALUSrcA(ALUSrcA),
73       .ALUSrcB(ALUSrcB),
74       .ALUOp(ALUOp),
75       .DBSrc(DBSrc),
76       .WrRegSrc(WrRegSrc),
77       .MemWrite(MemWrite),
78       .PCSrc(PCSsrc),
79       .PCWrite(PCWrite)
80     );
81
82     RegFile reg_file(
83       .clk(clk),
84       .reset(reset),
85       .r1(rs),
86       .r2(rt),
87       .wr(mux_regdst.res),
88       .RegWrite(RegWrite),
89       .wd(mux_wrreg.res)
90       // d1 -> adr / mux_pc.D
91       // d2 -> bdr
92     );
93
94     Register adr(
95       .clk(clk),
96       .reset(reset),
97       .in(reg_file.d1)
98       // out -> mux_srcA.A
99     );
100
101    Register bdr(
102      .clk(clk),
103      .reset(reset),
104      .in(reg_file.d2)
105      // out -> mux_srcB.A / dm.dataIn
106    );
107
108    assign d1 = mux_srcA.res;
109    assign d2 = mux_srcB.res;
110    assign rsData = reg_file.d1;
111    assign rtData = reg_file.d2;
112
113    // execution (EXE)
114    MUX5b mux_regdst(
115      .Sel(RegDst),
116      .A(rt),
117      .B(rd),
118      .C(5'b11111)
119      // res -> reg_file.wr
120    );
121
122     MUX mux_srcA(
123       .Sel(ALUSrcA),
124       .A(adr.out),
125       .B({{27{1'b0}},sa})
126       // res -> alu.A
127     );
128
129     Extender extender(
130       .Sel(ExtSel),
131       .dataIn(imm)
132       // dataOut -> mux_srcB.B / sl_16
133     );
134
135     MUX mux_srcB(
136       .Sel(ALUSrcB),
137       .A(bdr.out),
138       .B(extender.dataOut)
139       // res -> alu.B
140     );
141
142     ALU alu(
143       .op(ALUOp),
144       .A(mux_srcA.res),
145       .B(mux_srcB.res),
146       // res -> aludr / mux_memToReg
147       .zero(Zero)
148     );
149
150     Register aludr(
151       .clk(clk),
152       .reset(reset),
153       .in(alu.res),
154       // res -> dm.address / mux_memToReg.A
155       .out(alu_res)
156     );
157
158     // access memory (MEM)
159     DataMemory dm(
160       .clk(clk),
161       .reset(reset),
162       .address(alu_res),
163       .MemWrite(MemWrite),
164       .dataIn(bdr.out)
165       // dataOut -> mux_memToReg
166     );
167
168     // write back (WB)
169     MUX mux_memToReg(
170       .Sel(DBSrc),
171       .A(alu.res), // not aludr
172       .B(dm.dataOut)
173       // res -> dbdr
174     );
175
176     Register dbdr(
177       .clk(clk),
178       .reset(reset),
179       .in(mux_memToReg.res)
180       // out -> mux_wrreg.B
181     );
182
183     MUX mux_wrreg(
184       .Sel(WrRegSrc),
185       .A(dbdr.out),
186       .B(pc4)
187       // res -> reg_file.wd
188     );
189
190     assign dbData = mux_memToReg.res;
191
192     // jump & branch
193     ShiftLeft sl(
194       .dataIn(extender.dataOut)
195       // dataOut -> add_target.B
196     );
197
198     Adder add_target(
199       .A(pc4),
200       .B(sl.dataOut)
201       // res -> mux_branch.B
202     );
203
204     MUX4 mux_pc(

```

```

205     .Sel(PCSrc),
206     .A(pc4),
207     .B({pc4[31:28], instruction[25:0], 2'b00}),
208     .C(add_target.res),
209     .D(reg_file.d1),
210     .res(nextPC)
211   );
212
213 endmodule

```

### III. 指令存储器

```

23     assign d1 = (r1 == 0) ? 0 : register[r1];
24     assign d2 = (r2 == 0) ? 0 : register[r2];
25
26 // write data
27 always @(negedge clk) begin
28   if (reset == 0)
29     register[0] <= 0;
30   else if (wr != 0 && RegWrite == 1)
31     register[wr] <= wd;
32 end
33
34 endmodule

```

```

1 module InstructionMemory (
2   input [31:0] address,
3   output [31:0] dataOut
4 );
5
6   reg [7:0] memory [0:255]; // 8 bits (bandwidth) *
7   #256 (address)
8
9   // initialization
10  initial $readmem("D:/instruction.data",memory);
11
12  // output data (little endian)
13  assign dataOut = {memory[address + 3],
14    memory[address + 2],
15    memory[address + 1],
16    memory[address]};
17
18 endmodule

```

### IV. 程序计数器(PC)

```

1 module PC (
2   input clk,
3   input reset,
4   input PCWrite,
5   input [31:0] nextPC,
6   output reg [31:0] currPC
7 );
8
9   initial currPC = 0;
10
11  always @(posedge clk or negedge reset) begin
12    // #20 // important!
13    if (reset == 0)
14      // currPC <= 0;
15    currPC <= 32'hFFFFFFFC;
16    else if (PCWrite == 1)
17      currPC <= nextPC;
18  end
19
20 endmodule

```

### V. 寄存器堆

```

1 module RegFile (
2   input clk,
3   input reset,
4   input [4:0] r1, // read reg #1 address
5   input [4:0] r2, // read reg #2 address
6   input [4:0] wr, // write reg address
7   input RegWrite,
8   input [31:0] wd, // write data
9   output [31:0] d1, // read data 1
10  output [31:0] d2 // read data 2
11 );
12
13  reg [31:0] register [0:31]; // 32 bits (bandwidth)
14  * #32 (address)
15
16 // initialization
17 integer i;
18 initial begin
19   for (i = 0; i < 32; i = i + 1)
20     register[i] = 0;
21
22 // read data

```

### VI. 单个寄存器

```

1 module Register (
2   input clk,
3   input reset,
4   input [31:0] in,
5   output [31:0] out
6 );
7
8 reg [31:0] data;
9
10 // initialization
11 initial data = 0;
12
13 // read data
14 assign out = data;
15
16 // write data
17 always @(negedge clk) begin
18   if (reset == 0)
19     data <= 0;
20   else
21     data <= in;
22 end
23
24 endmodule

```

### VII. 数据存储器

```

1 module DataMemory (
2   input clk,
3   input reset,
4   input [31:0] address,
5   input MemWrite,
6   input [31:0] dataIn, // write data
7   output [31:0] dataOut // read data
8 );
9
10 reg [7:0] memory [0:255]; // 8 bits (bandwidth) *
11 #256 (address)
12
13 // initialization
14 integer i;
15 initial begin
16   for (i = 0; i < 256; i = i + 1)
17     memory[i] = 0;
18
19 // read data
20 assign dataOut = (address == 0) ? 0 : {memory[
21   address + 3],
22   memory[
23     address
24     + 2],
25   memory[
26     address
27     + 1],
28   memory[
29     address
30     ]};
31
32 // write data
33 always @(negedge clk) begin
34   if (reset == 0)
35     memory[0] <= 0;
36   else if (MemWrite == 1 && address != 0) begin
37     // do not use <=255!!!
38     // little endian
39   end
40 end
41
42 endmodule

```

```

31     memory[address + 3] <= dataIn[31:24];      66
32     memory[address + 2] <= dataIn[23:16];      67
33     memory[address + 1] <= dataIn[15:8];       68
34     memory[address] <= dataIn[7:0];           69
35   end
36 end
37
38 endmodule

```

## VIII. 控制单元

```

1 module ControlUnit (
2   input clk,
3   input reset,
4   input [5:0] opcode,
5   input Zero,
6   output reg [2:0] state,
7   output reg [1:0] RegDst,
8   output reg ExtSel,
9   output reg ALUSrcA,
10  output reg ALUSrcB,
11  output reg [2:0] ALUOp,
12  output reg [1:0] PCSrc,
13  output reg DBSrc,
14  output reg WrRegSrc,
15  output reg RegWrite,
16  output reg MemWrite,
17  output reg PCWrite
18 );
19
20 initial state = 3'b101;
21 reg [2:0] next_state = 3'b000;
22
23 // Finite State Machine (FSM)
24 always @ (posedge clk or negedge reset) begin
25   if (reset == 0)
26     state <= 3'b101;
27   else
28     state <= next_state;
29 end
30
31 always @ (state) begin
32   case (state)
33     3'b000: begin // IF
34       next_state = 3'b001;
35     end
36     3'b001: begin // ID
37       if (opcode == 6'b110100 || opcode == 6'b110101) #if
38         beg bne blitz
39         next_state = 3'b101;
40       else if (opcode == 6'b110000 || opcode == 6'b110001) // sw & lw
41         next_state = 3'b010;
42       else if (opcode == 6'b111000 || opcode == 6'b111001 || opcode == 6'b111111) #if
43         j jr jal halt
44         next_state = 3'b000;
45       else
46         next_state = 3'b110;
47     end
48     3'b110: begin // EXE
49       next_state = 3'b111;
50     end
51     3'b111: begin // WB
52       next_state = 3'b000;
53     end
54     3'b010: begin // EXE
55       next_state = 3'b011;
56     end
57     3'b011: begin // MEM
58       if (opcode == 6'b110001) // lw
59         next_state = 3'b100;
60       else // sw
61         next_state = 3'b000;
62     end
63     3'b100: begin // WB
64       next_state = 3'b000;
65     end
66   endcase
67 end
68
69 always @ (state) begin
70   RegDst <= 2'b00;
71   ExtSel <= 0;
72   if (state == 3'b111 || state == 3'b100)
73     RegWrite <= 1;
74   else
75     RegWrite <= 0;
76   ALUSrcA <= 0;
77   ALUSrcB <= 0;
78   ALUOp <= 3'b000;
79   DBSrc <= 0;
80   MemWrite <= 0;
81   PCSrc <= 2'b00;
82   WrRegSrc <= 0;
83   if (next_state == 3'b000)
84     PCWrite <= 1;
85   else
86     PCWrite <= 0; // do not update PC if the
87     state is not IF!
88   case (opcode)
89     6'b000000: begin // add rd, rs, rt
90       RegDst <= 2'b01;
91     end
92     6'b000001: begin // sub rd, rs, rt
93       RegDst <= 2'b01;
94       ALUOp <= 3'b001;
95     end
96     6'b000010: begin // addiu rt, rs, imm
97       ExtSel <= 1; // ???
98       ALUSrcB <= 1;
99     end
100    6'b010000: begin // and rd, rs, rt
101      RegDst <= 2'b01;
102      ALUOp <= 3'b100;
103    end
104    6'b010001: begin // andi rt, rs, imm
105      ALUSrcB <= 1;
106      ALUOp <= 3'b100;
107    end
108    6'b010010: begin // ori rt, rs, imm
109      ALUSrcB <= 1;
110      ALUOp <= 3'b011;
111    end
112    6'b010011: begin // xori rt, rs, imm
113      ALUSrcB <= 1;
114      ALUOp <= 3'b111;
115    end
116    6'b011000: begin // sll rd, rt, sa
117      RegDst <= 2'b01;
118      ALUSrcA <= 1;
119      ALUOp <= 3'b010;
120    end
121    6'b100110: begin // slti rt, rs, imm
122      ExtSel <= 1;
123      ALUSrcB <= 1; // remember!
124      ALUOp <= 3'b110;
125    end
126    6'b100111: begin // slt rd, rs, rt
127      RegDst <= 2'b01;
128      ALUOp <= 3'b110;
129    end
130    6'b110000: begin // sw rt, imm(rs)
131      ExtSel <= 1;
132      RegWrite <= 0;
133      ALUSrcB <= 1;
134      if (state == 3'b011)
135        MemWrite <= 1;
136      else
137        MemWrite <= 0;
138    end
139    6'b110001: begin // lw rt, imm(rs)
140      ExtSel <= 1;
141      ALUSrcB <= 1;
142      DBSrc <= 1;
143    end
144    6'b110100: begin // beq rs, rt, imm
145      ExtSel <= 1;
146      RegWrite <= 0;
147      ALUOp <= 3'b001;

```

```

148         if (Zero == 1)
149             PCSrc <= 2'b10;
150         else
151             PCSrc <= 2'b00;
152         end
153     6'b110101: begin // bne rs, rt, imm
154         ExtSel <= 1;
155         RegWrite <= 0;
156         ALUOp <= 3'b001;
157         if (Zero == 0) // (rs - rt == 0) ?
158             1 : 0 Not equal!
159             PCSrc <= 2'b10;
160         else
161             PCSrc <= 2'b00;
162     end
163     6'b110110: begin // bltz rs, imm
164         ExtSel <= 1;
165         RegWrite <= 0;
166         ALUOp <= 3'b110; // compare sign
167         if (Zero == 0) // a < 0 ? 1 : 0
168             PCSrc <= 2'b10;
169         else
170             PCSrc <= 2'b00;
171     end
172     6'b111000: begin // j addr
173         RegWrite <= 0;
174         PCSrc <= 2'b01;
175     end
176     6'b111001: begin // jr rs
177         RegWrite <= 0;
178         PCSrc <= 2'b11;
179     end
180     6'b111010: begin // jal addr
181         if (state == 3'b001)
182             RegWrite <= 1;
183             RegDst <= 2'b10;
184             WrRegSrc <= 1;
185             PCSrc <= 2'b01;
186     end
187     6'b111111: begin // halt
188         PCWrite <= 0;
189     end
190 endcase
191 end
192 endmodule

```

## IX. 算术逻辑单元(ALU)

```

1 module ALU (
2     input [2:0] op,
3     input [31:0] A,
4     input [31:0] B,
5     output reg [31:0] res,
6     output zero
7 );
8
9 initial begin
10     res = 0;
11 end
12
13 always @ (op or A or B) begin
14     case (op)
15         3'b000: res = A + B;
16         3'b001: res = A - B;
17         3'b010: res = B << A; // B first!
18         3'b011: res = A | B;
19         3'b100: res = A & B;
20         3'b101: res = (A < B) ? 1 : 0;
21         3'b110: res = ((A < B && A[31] == B[31]) ||
22                         both pos/neg num
23                         || (A[31] == 1 && B[31] == 0))
24                         // A neg B pos
25                         ? 1 : 0; // not 3'h0000001 !!
26         3'b111: res = A ^ B;
27     endcase
28
29 assign zero = (res == 0) ? 1 : 0;
30 endmodule

```

## X. 多路选择器(MUX)

```

1 module MUX (
2     input Sel,
3     input [31:0] A,
4     input [31:0] B,
5     output reg [31:0] res
6 );
7
8 always @ (Sel or A or B) begin
9     res <= (Sel == 0) ? A : B;
10 end
11
12 endmodule
13
14 module MUX5b (
15     input [1:0] Sel,
16     input [4:0] A,
17     input [4:0] B,
18     input [4:0] C,
19     output reg [4:0] res
20 );
21
22 always @ (Sel or A or B or C) begin
23     case (Sel)
24         2'b00: res <= A;
25         2'b01: res <= B;
26         2'b10: res <= C;
27     endcase
28 end
29
30 endmodule
31
32 module MUX4 (
33     input [1:0] Sel,
34     input [31:0] A,
35     input [31:0] B,
36     input [31:0] C,
37     input [31:0] D,
38     output reg [31:0] res
39 );
40
41 always @ (Sel or A or B or C or D) begin
42     case (Sel)
43         2'b00: res <= A;
44         2'b01: res <= B;
45         2'b10: res <= C;
46         2'b11: res <= D;
47     endcase
48 end
49
50 endmodule

```

## XI. 数据扩展器

```

1 module Extender (
2     input Sel,
3     input [15:0] dataIn,
4     output reg [31:0] dataOut
5 );
6
7 initial dataOut = 0;
8
9 always @ (Sel or dataIn) begin // dataIn!!!
10     if (Sel == 0) // ZeroExt
11         dataOut = {{16{1'b0}}, dataIn[15:0]};
12     else // SignExt
13         dataOut = {{16{dataIn[15]}}, dataIn[15:0]};
14     end
15
16 endmodule

```

## XII. 仿真代码

```

1 module CPU_sim (
2     output [2:0] state,
3     output [1:0] RegDst,
4     output ExtSel,
5     output RegWrite, MemWrite,
6     output ALUSrcA, ALUSrcB,
7     output [2:0] ALUOp,

```

### XIII. 分频计数器

XIV. 七段数码管

```

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

always @ (data)
    case(data)
        // 0: on      1: off
        0: dispcode = 7'b000_0001; // remember!
        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

```

XV. 写板电路

```

1 module Show(
2     input clk,
3     input clk_cpu, // button
4     input reset,
5     input [1:0] SW_in,
6     input State_in,
7     output reg [6:0] dispcode,
8     output reg [3:0] out
9 );
10
11 // synchronize and reduce jitter
12 reg in_detected = 1'b0;
13 reg [15:0] inhistory = 16'h0000;
14 always @(posedge clk) begin
15     inhistory = {inhistory[15:0], clk_cpu};
16     if (inhistory == 16'b0011111111111111)
17         in_detected <= 1'b1;
18     else
19         in_detected <= 1'b0;
20 end
21
22 wire [1:0] seg_num; // not reg!
23
24 Counter counter(
25     .clk(clk),
26     // output clock/counter
27     .count_4(seg_num)
28 );
29
30 reg [31:0] firstNum;
31 reg [31:0] secondNum;
32
33 initial firstNum = 0;
34 initial secondNum = 0;
35

```

```

36      wire [31:0] currPC, nextPC, rsData, rtData, dbData193
37          alu_res;                                104
38      wire [4:0] rs, rt;                          105
39      wire [2:0] state;                           106
40
41      CPU cpu(
42          // input
43          .clk(in_detected),
44          .reset(reset),
45          // output
46          .currPC(currPC),
47          .nextPC(nextPC),
48          .rs(rs),
49          .rt(rt),
50          .rsData(rsData),
51          .rtData(rtData),
52          .dbData(dbData),
53          .alu_res(alu_res),
54          .state(state)
55      );
56
57      always @ (SW_in or State_in) begin
58          if (State_in == 0)
59          case (SW_in)
60              2'b00: begin
61                  firstNum <= currPC;
62                  secondNum <= nextPC;
63                  end
64              2'b01: begin
65                  firstNum <= {{27{1'b0}},rs};
66                  secondNum <= rsData;
67                  end
68              2'b10: begin
69                  firstNum <= {{27{1'b0}},rt};
70                  secondNum <= rtData;
71                  end
72              2'b11: begin
73                  firstNum <= alu_res;
74                  secondNum <= dbData;
75                  end
76          endcase
77      else begin
78          firstNum <= 0;
79          secondNum <= {{29{1'b0}},state[2:0]};
80      end
81
82      SegDisplay seg1(
83          .data(firstNum[7:4])
84          // .dispcode
85      );
86
87      SegDisplay seg2(
88          .data(firstNum[3:0])
89          // .dispcode
90      );
91
92      SegDisplay seg3(
93          .data(secondNum[7:4])
94          // .dispcode
95      );
96
97      SegDisplay seg4(
98          .data(secondNum[3:0])
99          // .dispcode
100         );
101
102     always @ (seg_num or firstNum or secondNum)
103
104         alu_res;                                104
105     wire [4:0] rs, rt;                          105
106     wire [2:0] state;                           106
107
108     CPU cpu(
109         // input
110         .clk(in_detected),
111         .reset(reset),
112         // output
113         .currPC(currPC),
114         .nextPC(nextPC),
115         .rs(rs),
116         .rt(rt),
117         .rsData(rsData),
118         .rtData(rtData),
119         .dbData(dbData),
120         .alu_res(alu_res),
121         .state(state)
122     );
123
124     case (seg_num)
125         0: begin
126             out = 4'b1110;
127             dispcode = seg4.dispcode;
128         end
129         1: begin
130             out = 4'b1101;
131             dispcode = seg3.dispcode;
132         end
133         2: begin
134             out = 4'b1011;
135             dispcode = seg2.dispcode;
136         end
137         3: begin
138             out = 4'b0111;
139             dispcode = seg1.dispcode;
140         end
141     endcase
142
143 endmodule

```

## XVI. 限制文件(constraints.xdc)

```

1 set_property PACKAGE_PIN W5 [get_ports clk]
2 set_property PACKAGE_PIN T17 [get_ports clk_cpu]
3 set_property PACKAGE_PIN V17 [get_ports reset]
4 set_property PACKAGE_PIN R2 [get_ports {SW_in[1]}]
5 set_property PACKAGE_PIN T1 [get_ports {SW_in[0]}]
6 set_property PACKAGE_PIN U1 [get_ports {State_in}]
7 set_property PACKAGE_PIN W4 [get_ports {out[3]}]
8 set_property PACKAGE_PIN V4 [get_ports {out[2]}]
9 set_property PACKAGE_PIN U4 [get_ports {out[1]}]
10 set_property PACKAGE_PIN U2 [get_ports {out[0]}]
11 set_property PACKAGE_PIN W7 [get_ports {dispcode[6]}]
12 set_property PACKAGE_PIN W6 [get_ports {dispcode[5]}]
13 set_property PACKAGE_PIN U8 [get_ports {dispcode[4]}]
14 set_property PACKAGE_PIN V8 [get_ports {dispcode[3]}]
15 set_property PACKAGE_PIN U5 [get_ports {dispcode[2]}]
16 set_property PACKAGE_PIN V5 [get_ports {dispcode[1]}]
17 set_property PACKAGE_PIN U7 [get_ports {dispcode[0]}]
18
19 set_property IOSTANDARD LVCMOS33 [get_ports clk]
20 set_property IOSTANDARD LVCMOS33 [get_ports clk_cpu]
21 set_property IOSTANDARD LVCMOS33 [get_ports reset]
22 set_property IOSTANDARD LVCMOS33 [get_ports {SW_in[1]}]
23 set_property IOSTANDARD LVCMOS33 [get_ports {SW_in[0]}]
24 set_property IOSTANDARD LVCMOS33 [get_ports {State_in}]
25 set_property IOSTANDARD LVCMOS33 [get_ports {out[3]}]
26 set_property IOSTANDARD LVCMOS33 [get_ports {out[2]}]
27 set_property IOSTANDARD LVCMOS33 [get_ports {out[1]}]
28 set_property IOSTANDARD LVCMOS33 [get_ports {out[0]}]
29 set_property IOSTANDARD LVCMOS33 [get_ports {dispcode
   [6]}]
30 set_property IOSTANDARD LVCMOS33 [get_ports {dispcode
   [5]}]
31 set_property IOSTANDARD LVCMOS33 [get_ports {dispcode
   [4]}]
32 set_property IOSTANDARD LVCMOS33 [get_ports {dispcode
   [3]}]
33 set_property IOSTANDARD LVCMOS33 [get_ports {dispcode
   [2]}]
34 set_property IOSTANDARD LVCMOS33 [get_ports {dispcode
   [1]}]
35 set_property IOSTANDARD LVCMOS33 [get_ports {dispcode
   [0]}]

```