



# 《计算机组成原理实验》

## 实验报告

学院名称 : 计算机学院

专业(班级) : 计算机科学与计算(超算)

学生姓名 : 黄玟瑜

学号 : 19335074

时间 : 2020 年 12 月 12 日

# 成 绩 :

## 实 验 : 单周期CPU设计与实现

### 一. 实验目的

1. 理解MIPS常用的指令系统并掌握单周期CPU的工作原理与逻辑功能实现。
2. 通过对单周期CPU的运行状况进行观察和分析，进一步加深理解。

### 二. 实验内容

#### 实验要求

1、利用 HDL 语言，基于 Xilinx FPGA basys3 实验平台，用 Verilog HDL 语言或 VHDL 语言来编写，实现单周期CPU 的设计，这个单周期 CPU 能够完成 16 条MIPS 指令，至少包含以下指令：

支持基本的内存操作如 lw, sw 指令

支持基本的算术逻辑运算如 add, sub, and, ori, slt, addi 指令

支持基本的程序控制如 beq, j 指令

2、掌握各个指令的相关功能并输出仿真结果进行验证，并最后在 FGPA 上实现，将其中的 alu 运算结果在开发板数码管上显示出来。

3、可拓展添加其他指令。

4、PC和寄存器组写状态使用时钟边缘触发。

5、指令存储器和数据存储器存储单元宽度一律使用8位，（便于添加字节存取）即一个字节的存储单位。不能使用32位作为存储器存储单元宽度。深度的选择满足指令要求即可（测试代码的长度）。

6、控制器部分要学会用控制信号真值表方法分析问题并写出逻辑表达式；或者用case语句方法逐个产生各指令控制信号。

7、必须按统一测试用的汇编程序段进行测试所设计的CPU。

8、必须注意，实验报告中，对每条指令必须有指令执行的波形（截图），且图上必须包含关键信号，同时还要对关键信号以文字说明，这样才能说明该指令的正确性。

### 三. 实验原理

#### 1. 单时钟周期 CPU

单周期 CPU 的特点是每条指令的执行只需要一个时钟周期，一条指令执行完再执行下一条指令。再这一个周期中，完成更新地址，取指，解码，执行，内存操作以及寄存器操作。由于每个时钟上升沿时更新地址，因此要在上升沿到来之前完成所有运算，而这所有的运算除可以利用一个下降沿外，只能通过组合逻辑解决。这给寄存器和存储器 RAM 的制作带来了些许难度。且因为每个时钟周期的时间长短必须统一，因此在确定时钟周期的时间长度时，要依照最长延迟的指令时间来定，这也限制了它的执行效率。

单周期 CPU 在每个 CLK 上升沿时更新 PC，并读取新的指令。此指令无论执行时间长短，都必须在下一个上升沿到来之前完成。其时序示意如图 I。

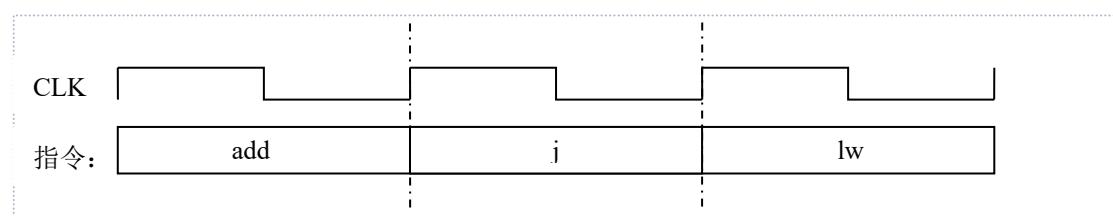
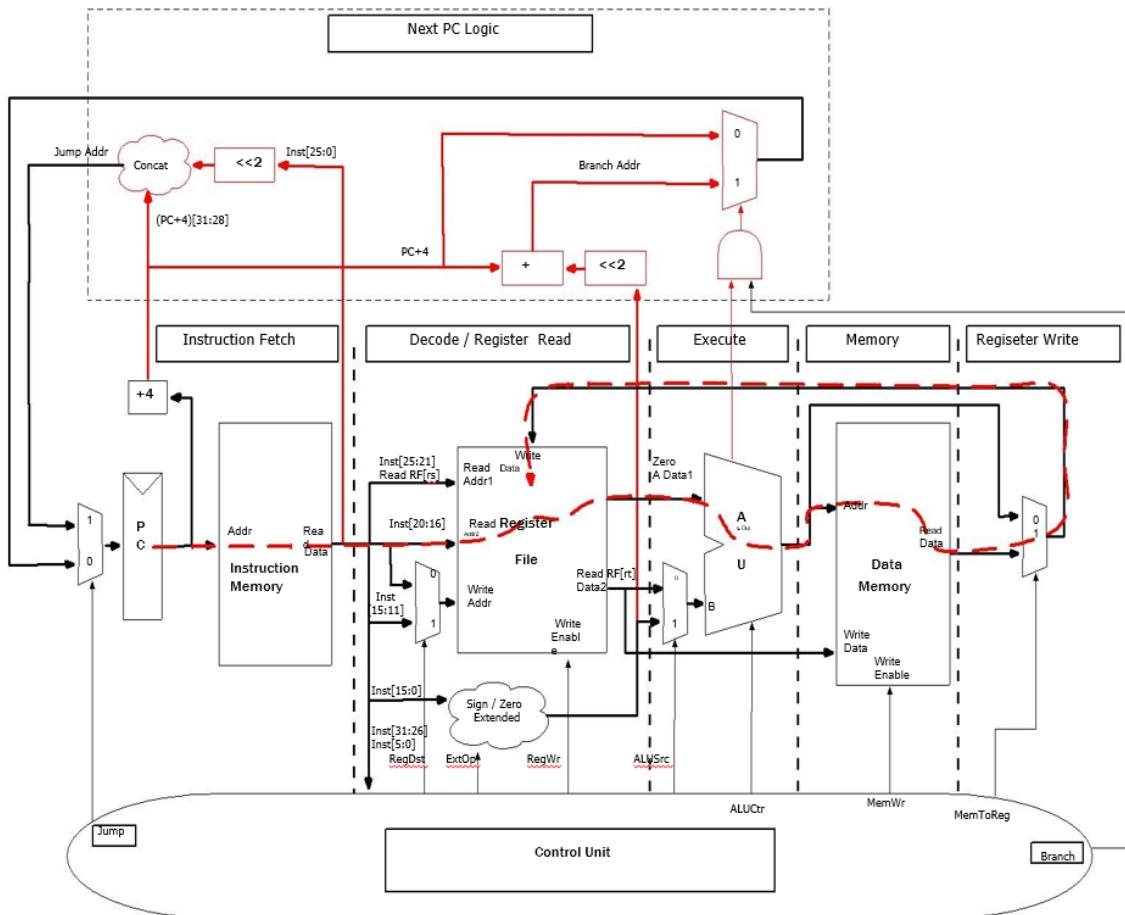


图 I 单时钟周期 CPU 时序示意图

下图是一个单周期 CPU 的顶层结构实现。主要器件有程序计数器 PC、程序存储器、寄存器堆、ALU、数据存储器和控制部件等。所有的控制信号简单地说明如下：



图II 单时钟周期 CPU 详细逻辑设计图

其中，控制单元(Control Unit)定义如下：

- (1) jmp: 为 1 时，选择跳转目标地址；为 0 时，选择由 Branch 选出的地址；
- (2) memToReg: 为 1 时，选择存储器数据；为 0 时，选择 ALU 输出的数据；
- (3) branch: 为 1 时，选择转移目标地址；为 0 时，选择 PC +4 (图中的 NextPC)；
- (4) memWrite: 为 1 时写入存储器。存储器地址由 ALU 的输出决定，写入数据为寄存器 rt 的内容；
- (5) aluop: ALU 控制码；
- (6) aluSrc: ALU 操作数 B 的选择，为 1 时，选择扩展的立即数；为 0 时，选择寄存器数据；

(7) regWrite: 为 1 时写入寄存器堆, 目的寄存器号 ra 或由 RegDst 选出的 rt 或 rd,

写入数据是 PC+4 或由 MemToReg 选出的存储器数据或 ALU 的输出结果;

(8) ExtOp: 符号扩展。为 1 时, 符号扩展; 为 0 时, 0 扩展;

(9) regDst: 目的地址, 为 1 时, 选择 rd; 为 0 时, 选择 rt;

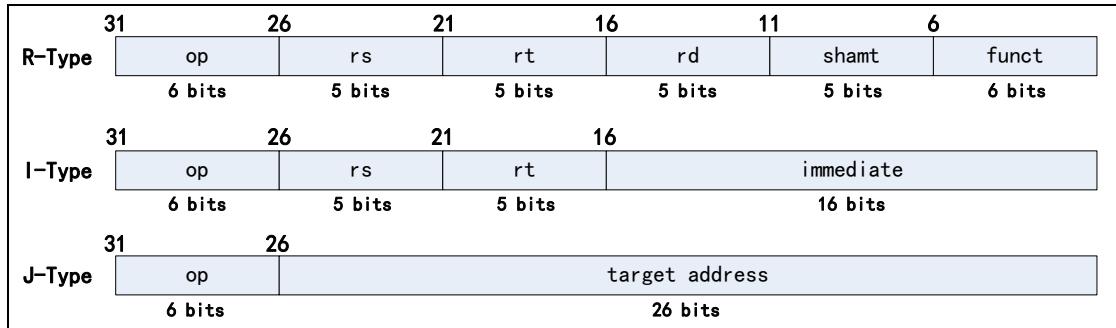
(10) ld: 装载信号, 为 1 时, 将 PC+4 装载到\$ra 中;

(11) jr: 寄存器跳转信号, 为 1 时, 跳转到 rs 输出数据的地址。

## 2. MIPS 指令集

本次实验共涉及三种类型的 MIPS 指令, 分别为 R 型、I 型和 J 型, 三种类型的 MIPS 指令格式定义如下:

- R (register) 类型的指令从寄存器堆中读取两个源操作数, 计算结果写回寄存器堆;
- I (immediate) 类型的指令使用一个 16 位的立即数作为一个源操作数;
- J(jump)类型的指令使用一个 26 位立即数作为跳转的目标地址(target address);



图III MIPS 指令集

一条指令的执行过程一般有下面的五个阶段, 指令的执行过程就是这五个状态的重复

过程：

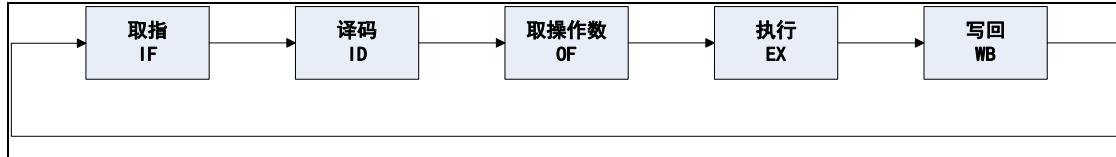


图 IV MIPS 指令集

助记符	指 令 格 式						示 例	示例含义	操作及解释
BIT #	31..26	25..21	20..16	15..11	10..6	5..0			
R-类型	op	rs	rt	rd	shamt	func			
add	000000	rs	rt	rd	00000	100000	add \$1,\$2,\$3	\$1=\$2+S3	(rd)←(rs)+(rt); rs=\$2,rt=\$3,rd=\$1
addu	000000	rs	rt	rd	00000	100001	addu \$1,\$2,\$3	\$1=\$2+S3	(rd)←(rs)+(rt); rs=\$2,rt=\$3,rd=\$1,无符号数
sub	000000	rs	rt	rd	00000	100010	sub \$1,\$2,\$3	\$1=\$2-S3	(rd)←(rs)-(rt); rs=\$2,rt=\$3,rd=\$1
subu	000000	rs	rt	rd	00000	100011	subu \$1,\$2,\$3	\$1=\$2-S3	(rd)←(rs)-(rt); rs=\$2,rt=\$3,rd=\$1,无符号数
and	000000	rs	rt	rd	00000	100100	and \$1,\$2,\$3	\$1=\$2&S3	(rd)←(rs)&(rt); rs=\$2,rt=\$3,rd=\$1
or	000000	rs	rt	rd	00000	100101	or \$1,\$2,\$3	\$1=\$2 S3	(rd)←(rs)   (rt); rs=\$2,rt=\$3,rd=\$1
xor	000000	rs	rt	rd	00000	100110	xor \$1,\$2,\$3	\$1=\$2^S3	(rd)←(rs)^ (rt); rs=\$2,rt=\$3,rd=\$1
nor	000000	rs	rt	rd	00000	100111	nor \$1,\$2,\$3	\$1= ~(\$2   S3)	(rd)←~((rs)   (rt)); rs=\$2,rt=\$3,rd=\$1
slt	000000	rs	rt	rd	00000	101010	slt \$1,\$2,\$3	if(\$2<\$3) \$1=1 else \$1=0	if (rs < rt) rd=1 else rd=0; rs=\$2, rt=\$3, rd=\$1
sltu	000000	rs	rt	rd	00000	101011	sltu \$1,\$2,\$3	if(\$2<\$3) \$1=1 else \$1=0	if (rs < rt) rd=1 else rd=0; rs=\$2, rt=\$3, rd=\$1, 无符号数
sll	000000	00000	rt	rd	shamt	000000	sll \$1,\$2,10	\$1=\$2<<10	(rd)←(rt)<<shamt, rt=\$2, rd=\$1, shamt=10
srl	000000	00000	rt	rd	shamt	000010	srl \$1,\$2,10	\$1=\$2>>10	(rd)←(rt)>>shamt, rt=\$2, rd=\$1, shamt=10, (逻辑右移)
sra	000000	00000	rt	rd	shamt	000011	sra \$1,\$2,10	\$1=\$2>>10	(rd)←(rt)>>shamt, rt=\$2, rd=\$1, shamt=10, (算术右移, 注意符号位保留)
sllv	000000	rs	rt	rd	00000	000100	sllv \$1,\$2,\$3	\$1=\$2<<\$3	(rd)←(rt)<<(rs), rs=\$3, rt=\$2, rd=\$1
srlv	000000	rs	rt	rd	00000	000110	srlv \$1,\$2,\$3	\$1=\$2>>\$3	(rd)←(rt)>>(rs), rs=\$3, rt=\$2, rd=\$1, (逻辑右移)
sraw	000000	rs	rt	rd	00000	000111	sraw \$1,\$2,\$3	\$1=\$2>>\$3	(rd)←(rt)>>(rs), rs=\$3, rt=\$2, rd=\$1, (算术右移, 注意符号位保留)
jr	000000	rs	00000	00000	00000	001000	jr \$31	goto \$31	(PC)←(rs)
I-类型	op	rs	rt	immediate					
addi	001000	rs	rt	immediate			addi \$1,\$2,10	\$1=\$2+10	(rt)←(rs)+(sign-extend)immediate, rt=\$1, rs=\$2
addiu	001001	rs	rt	immediate			addiu \$1,\$2,10	\$1=\$2+10	(rt)←(rs)+(sign-extend)immediate, rt=\$1, rs=\$2
andi	001100	rs	rt	immediate			andi \$1,\$2,10	\$1=\$2&10	(rt)←(rs)&(zero-extend)immediate, rt=\$1, rs=\$2

ori	001101	rs	rt	immediate	ori \$1,\$2,10	\$1=\$2 10	$(rt) \leftarrow (rs)   (zero-extend) immediate, rt = \$1, rs = \$2$
xori	001110	rs	rt	immediate	xori \$1,\$2,10	\$1=\$2^10	$(rt) \leftarrow (rs) ^ (zero-extend) immediate, rt = \$1, rs = \$2$
lui	001111	00000	rt	immediate	lui \$1,10	\$1=10*65536	$(rt) \leftarrow immediate << 16 \& OFFFF0000H, 将 16 位立即数放到目的寄存器高 16 位, 目的寄存器的低 16 位填 0$
lw	100011	rs	rt	offset	lw \$1,10(\$2)	\$1=Memory[\$2+10]	$(rt) \leftarrow Memory[(rs) + (sign\_extend) offset], rt = \$1, rs = \$2$
sw	101011	rs	rt	offset	sw \$1,10(\$2)	Memory[\$2+10] = \$1	$Memory[(rs) + (sign\_extend) offset] \leftarrow (rt), rt = \$1, rs = \$2$
beq	000100	rs	rt	offset	beq \$1,\$2,40	if(\$1==\$2) goto PC+4+40	if ((rt)=(rs)) then (PC)←(PC)+4+(Sign-Extend) offset<<2), rs=\$1, rt=\$2
bne	000101	rs	rt	offset	bne \$1,\$2,40	if(\$1≠\$2) goto PC+4+40	if ((rt)≠(rs)) then (PC)←(PC)+4+(Sign-Extend) offset<<2), rs=\$1, rt=\$2
slti	001010	rs	rt	immediate	slti \$1,\$2,10	if(\$2<10) \$1=1 else \$1=0	if ((rs)<(Sign-Extend)immediate) then (rt)←1; else (rt)←0, rs=\$2, rt=\$1
sltiu	001011	rs	rt	immediate	sltiu \$1,\$2,10	if(\$2<10) \$1=1 else \$1=0	if ((rs)<(Zero-Extend)immediate) then (rt)←1; else (rt)←0, rs=\$2, rt=\$1
J-类型	op	address					
j	000010	address			j 10000	goto 10000	$(PC) \leftarrow (Zero-Extend) address << 2, address = 10000/4$
jal	000011	address			jal 10000	\$31=PC+4 goto 10000	$(\$31) \leftarrow (PC) + 4;$ $(PC) \leftarrow (Zero-Extend) address << 2, address = 10000/4$

图 V 指令详细结构

以上为MIPS指令集的31条基本指令，以下为拓展的指令。

指 令 格 式						
BIT #	31..26	25..21	20..16	15..11	10..6	5..0
R-类型	op	rs	rt	rd	shamt	func
jalr	000000	rs	00000	11111	00000	001001
clo	000000	rs	00000	rd	00000	100001
clz	000000	rs	00000	rd	00000	100000
mul	000000	rs	rt	rd	00000	000010
mult	000000	rs	rt	00000	00000	011000
multu	000000	rs	rt	00000	00000	011001

div	000000	rs	rt	00000	00000	011010
divu	000000	rs	rt	00000	00000	011011
madd	011100	rs	rt	00000	00000	000000
maddu	011100	rs	rt	00000	00000	000001
msub	011100	rs	rt	00000	00000	000100
msubu	011100	rs	rt	00000	00000	000101
mfhi	000000	00000	00000	rd	00000	010000
mflo	000000	00000	00000	rd	00000	010010
mthi	000000	rs	00000	00000	00000	010001
mtlo	000000	rs	00000	00000	00000	010011
I-类型	op	rs	rt	immediate		
beqz	000100	rs	00000	immediate		
bnez	000101	rs	00000	immediate		
bltz	000001	rs	00000	immediate		
blez	000110	rs	00000	immediate		
bltzal	000001	rs	10000	immediate		
bgtz	000111	rs	00000	immediate		
bgez	000001	rs	00001	immediate		
bgezal	000001	rs	10001	immediate		
lb	100000	rs	rt	immediate		
lbu	100100	rs	rt	immediate		
lh	100001	rs	rt	immediate		
luh	100101	rs	rt	immediate		
sb	101000	rs	rt	immediate		
sh	101001	rs	rt	immediate		

共61条指令（MIPS基本指令31条+30条拓展指令）。

#### 四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

#### 五. 实验过程与结果

##### 1. CPU的设计

该MIPS主要由10个模块组成，各个子模块分别设计其特定的功能，最终利用一个总的模块（TOP模块）进行子模块间连接，使得整个CPU能连贯执行指令，在仿真结果中观察设

计结果，最终进行硬件下载，验证设计。其中各个模块简单功能如下：

- (1) 指令存储器模块：具备基本的读写功能，用于存放指令。
- (2) 寄存器堆模块：由32个32位的寄存器组成，提供较大的存储空间，用于存放暂存数据和指令。
- (3) ALU模块：执行加减法等算术运算，与非或等逻辑运算，以及比较移位传送等操作的功能部件，是该CPU的设计核心部分，存在不同的运算处理功能，是体现实验设计结果正确性的模块。
- (4) 符号扩展模块：执行I型指令时需要立即数扩展，该模块用于MIPS符号扩展，将16位数据扩展为32位数据。
- (5) 主控制模块：用于控制各个模块之间的分工运行，产生不同数据通路的控制信号，保证指令顺序执行不发生紊乱。
- (6) ALU控制模块：用于生成ALU执行各种功能的控制信号，使ALU内部运行不发生紊乱。
- (7) 数据存储器模块，用于LW/SW指令数据存取。
- (8) PC计数模块：进行指令的取出。
- (9) PC控制模块：根据控制信号或ALU运算结果计算出下一条PC的值。
- (10) 显示模块DISPLAY 数码管显示，显示16位运算结果。

以下为各模块的设计，为了防止报告过于冗长省略部分代码。

### 1、控制器模块（主控模块）

根据指令中的指令码 (op) 和功能码 (funct) 的不同组合输出相应的控制信号。

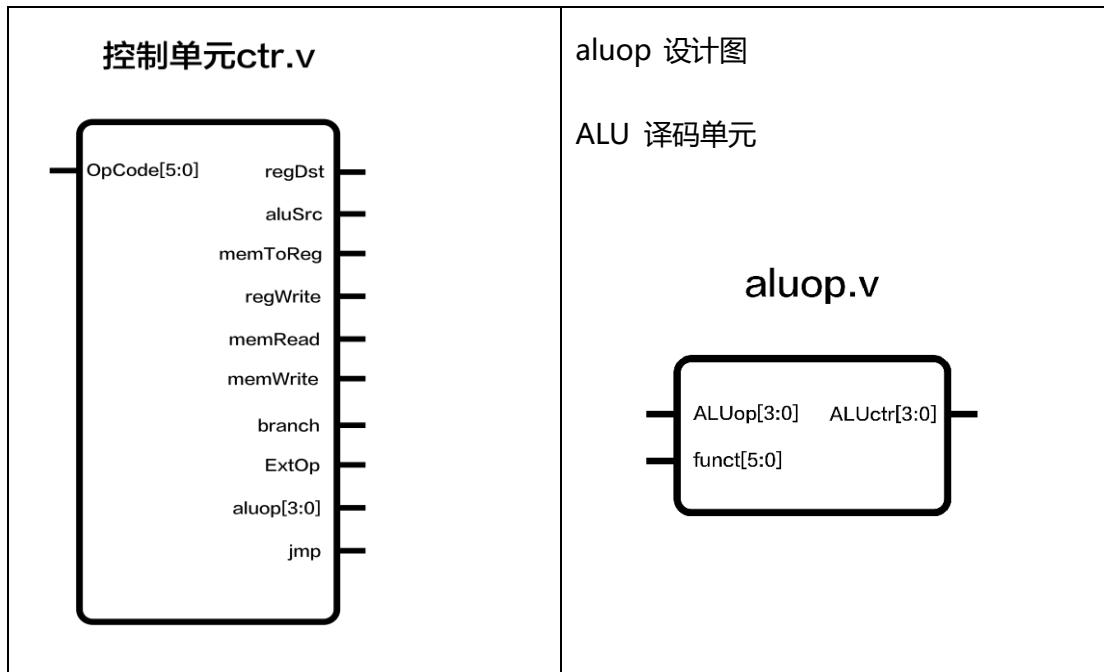
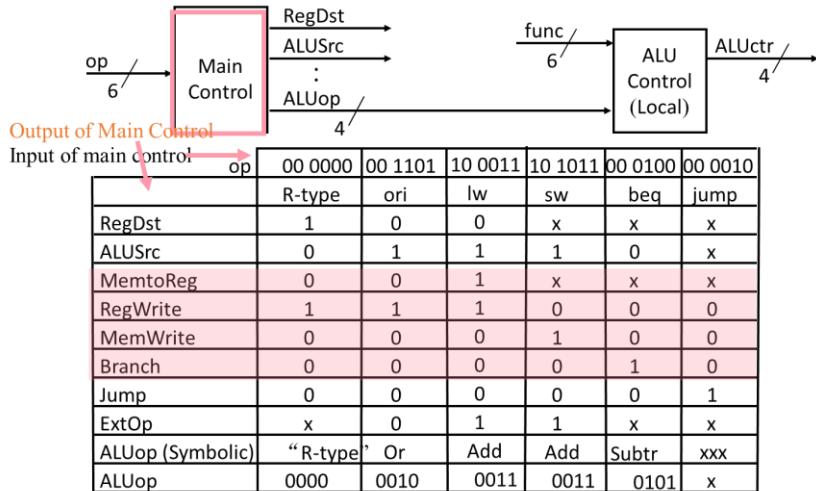


图 控制器组成原理图

图片仅供参考，在此用到的ALUop和ALUctr为5位，ALU Control的输入信号还包括rt。

设计思想：

根据6位指令码op译出**reg\_dst**, **alu\_src**, **memtoreg**, **regwrite**, **memWrite**, **branch**, **ExtOp**, **jmp**, **load**这8个控制信号以及输送给ALU控制模块的**aluop[4:0]**信号。

对R型指令（操作码为000000），根据R型指令的特点（目的寄存器为rd，要进行寄存器写操作，写入数据为ALU运算结果等）设计如下：

//R型指令

```

regDst = 1; aluSrc = 0; memToReg = 0;

regWrite = 1; memWrite = 0;

branch = 0; jmp = 0; load = 0; ExtOp=0;

aluop = 5'b00000;

```

对其他类型的指令，如I型，根据每条指令不同的需求输出控制信号，对无关的控制信号缺省值为0，防止烧板时报错。

```

// 'addi' 指令操作码: 001000

regDst = 0; aluSrc = 1; memToReg = 0;

regWrite = 1; memWrite = 0;

branch = 0; jmp = 0; load = 0; ExtOp = 1;

aluop = 5'b00011;

// 'j' 指令操作码: 000010, 无需 ALU

regDst = 0; aluSrc = 0; memToReg = 0;

regWrite = 0; memWrite = 0;

jmp = 1; branch = 0; load = 0; ExtOp=0;

aluop = 5'b00000;

```

对操作码相同的指令归结为同一类，在ALU控制译码再译出相应的操作，例如bltz、bltzal、bgez、bgezal的操作码：

```

6'b000001: begin

regDst = 1; aluSrc = 0; memToReg = 0;

regWrite = 1; memWrite = 0;

branch = 1; jmp = 0; load = 0; ExtOp = 1;

```

```
aluop = 5'b10001;  
end // bltz bgez bltzal bgezal
```

该模块代码大致如下：

```
`timescale 1ns / 1ps  
  
module ctr(  
    input [5:0] opCode,  
    output reg regDst,  
    output reg aluSrc,  
    output reg memToReg,  
    output reg regWrite,  
    output reg memWrite,  
    output reg branch,  
    output reg ExtOp, //符号扩展方式, 1 为 sign-extend, 0 为 zero-extend  
    output reg load,  
    output reg jmp,  
    output reg[4:0] aluop// 经过 ALU 控制译码决定 ALU 功能  
);
```

```
always@(opCode) begin  
    // 操作码改变时改变控制信号  
    case(opCode)  
        // 'R 型' 指令操作码: 000000
```

```
6'b000000: begin  
  
    regDst = 1;  aluSrc = 0; memToReg = 0;  
  
    regWrite = 1;memWrite = 0;  
  
    branch = 0; jmp = 0; load = 0; ExtOp=0;  
  
    aluop = 5'b00000;  
  
    end  
  
//'I型指令操作码  
  
6'b001000: begin  
  
    regDst = 0;  aluSrc = 1; memToReg = 0;  
  
    regWrite = 1; memWrite = 0;  
  
    branch = 0; jmp = 0;load = 0; ExtOp = 1;  
  
    aluop = 5'b00011;  
  
    end // 'addi' 指令操作码: 001000
```

.....

```
6'b011100: begin  
  
    regDst = 1;  aluSrc = 0; memToReg = 0;  
  
    regWrite = 0; memWrite = 0;  
  
    branch = 0; jmp = 0; load = 0; ExtOp=0;  
  
    aluop = 5'b11010;  
  
    end //madd msub maddu msubu
```

```

default: begin

    regDst = 0; aluSrc = 0; memToReg = 0;

    regWrite = 0; memWrite = 0;

    branch = 0; jmp = 0; load = 0; ExtOp = 0;

    aluop = 5'b11111;

    end // 默认设置

endcase

end

endmodule

```

## 2、ALU控制译码模块

ALU 主要执行的操作有与、或、加、减、小于设置等等。这些操作需要用五位编码来控制。指令不同，则对应的 ALU 运算不同，所以该模块需要根据指令来控制ALU 进行正确的运算。

lw, sw, addi 指令均要求 ALU 执行加操作，则可分为一类，aluop编码 00011； sub、subu、beq指令要求 ALU 执行减操作，则分为一类，编码 00111； div、divu指令要求 ALU 执行除操作，则分为一类，编码 10111； bltz、bltzal指令要求 ALU 在第一个操作数小于0发出跳转信号，则分为一类，编码10001； bgez、bgezal指令要求 ALU 在第一个操作数小于0发出跳转信号，则分为一类，编码 10100； 等等，以此类推。

最后一类是 R 型指令，可以编码为 00000；但不同的R 型指令对应不同的 ALU 运算，故需要再通过指令的功能码进一步确定 ALU 的运算。

经观察发现，存在部分I型指令（bltz、bltzal、bgez、bgezal）操作码相同，他们之间通过rt段进行区分，因此ALU控制模块需要增加一个输入rt。最终该模块的实现包括输入 5

位操作码aluop、 6 位功能码funct以及5位rt, 输出 5 位ALU 控制信号码 (此外还输出控制信号jr、load、nRW, 具体说明见代码说明)。

译码器的真值表:

ALU 功能真值表

指令	ALUop	func	rt	ALU_Operation	功能描述
add	00000	100000	x	00010	add
addu	00000	100001	x	00010	add
sub	00000	100010	x	00110	sub
subu	00000	100011	x	00110	sub
and	00000	100100	x	00000	and
or	00000	100101	x	00001	or
xor	00000	100110	x	00100	xor
nor	00000	100111	x	00101	nor
slt	00000	101010	x	00111	slt
sltu	00000	101011	x	01110	sltu
sll	00000	000000	x	01000	sll
srl	00000	000010	x	01001	srl
sra	00000	000011	x	01010	sra
sllv	00000	000100	x	01011	sllv
srlv	00000	000110	x	01100	srlv
srav	00000	000111	x	01101	srav
clo	11010	100001	00000	11010	clo
clz	11010	100000	00000	11011	clz
mul	11010	000010	x	11001	mul
mult	00000	011000	x	10100	mult
multu	00000	011001	x	10101	multu
div	00000	011010	x	10110	div
divu	00000	011011	x	10110	div
mfhi	00000	010000	00000	10111	mfhi
mflo	00000	010010	00000	11000	mflo

mthi	00000	010001	00000	11100	mthi
mtlo	00000	010011	00000	11101	mtlo
addi	00011	x	x	00010	add
addiu	00011	x	x	00010	add
andi	00001	x	x	00000	and
ori	00010	x	x	00001	or
xori	00101	x	x	00100	xor
lui	10000	x	x	01111	lui
lw	00011	x	x	00010	add
lb	00011	x	x	00010	add
lbu	00011	x	x	00010	add
lh	00011	x	x	00010	add
lhu	00011	x	x	00010	add
sw	00011	x	x	00010	add
sh	00011	x	x	00010	add
sb	00011	x	x	00010	add
beq	00111	x	x	00110	sub
beqz	00111	x	00000	00110	sub
bne	00100	x	x	00011	bne
bnez	00100	x	00000	00011	bne
slti	01000	x	x	00111	slt
sltiu	01111	x	x	01110	sltu
bltz	10001	x	00000	10000	bltz,load=0
bltzal	10001	x	10000	10000	bltz,load=1
bgez	10001	x	00001	10011	bgez, load=0
bgezal	10001	x	10001	10011	bgez, load=1
blez	10010	x	00000	10001	blez
bgtz	10011	x	00000	10010	bgtz
j	x	x	x	x	load=0,jr=0
jal	x	x	x	x	load=1,jr=0
jr	00000	001000	x	x	load=0,jr=1
jalr	00000	001001	x	x	load=1,jr=1

模块代码如下：

/\*说明：除了aluCtr，还增加了3个控制信号jr、load2、nRW。其中：

- nRW为寄存器写控制信号，nRW=1时不进行寄存器写操作，该信号用于在主控模块译码时regWrite为1，但实际上不需要进行寄存器写操作，比如jr指令，jr为R型指令，在主控译码时regWrite=1，但实际上不需要进行寄存器写，若进行寄存器写操作将会导致寄存器堆的值被错误修改，因此增设nRW信号用于防止这一情况发生；
- load2为装数信号，load2为1时需要将PC+4写入ra中；
- jr为寄存器跳转信号，为1时跳转方式为寄存器跳转。

\*/

`timescale 1ns / 1ps

module aluctr(

input [4:0] ALUOp,

input [5:0] funct,

input [4:0] rt,

output reg [4:0] ALUCtr,

output reg jr, load2, nRW

);

initial begin

ALUCtr = 5'b00000;

jr = 0;

load2 = 0;

nRW = 0;

end

```

always @(ALUOp or funct or rt) begin// 如果操作码或者功能码变化执行操作

casex({ALUOp, funct, rt}) // 拼接操作码和功能码便于下一步的判断

//I型

16'b000011xxxxxxxxxx: begin ALUCtr = 5'b00010; jr = 0; load2 =
0; nRW = 0; end// lw,sw,addi,addiu

16'b00001xxxxxxxxxxx: begin ALUCtr = 5'b00000; jr = 0; load2 =
0; nRW = 0; end// andi

......

16'b11010000x01xxxx: begin ALUCtr = 5'b11110; jr = 0; load2 =
0; nRW = 1; end//maddu msuzu

default: begin ALUCtr = 5'b11111; jr = 0; load2 = 0; nRW = 1; end

endcase

end

endmodule

```

### 3、指令存储器模块

该模块有8位地址输入和32位数据输出，首先将61种类型79条指令写入存储单元中，然后根据8位地址输入选择相应的单元指令内容，将数据写入到输出变量中。

虽然用IP核设计的指令存储器在修改指令时较为方便，但是在烧板时会出现许多意想不到的问题，所以在这里不使用IP核设计指令存储器。虽然实验前期为了调试IP核付出了非常多的时间和精力，在最后烧板时才发现IP核不好用，这个过程中浪费了大量的时间和头发，但最终我还是决定使用最基本的方法，虽然这个方法让我在复制粘贴指令时差点暴走。

模块设计如下：

```
`timescale 1ns / 1ps

module IM_unit(
    input [7:0] Addr,      //指令存储器地址编码
    output [31:0] instruction// 寄存器的值
);

reg [31:0] IM [0:255]; // 寄存器组

assign instruction=IM[Addr];

integer i;

initial begin

    IM[0]<=32'h00000000;      //0000
    IM[1]<=32'h24010008;      //0004
    IM[2]<=32'h34020002;      //0008
    .....
    IM[79]<=32'h3981ffff;    //013c

    for(i=80;i<256;i=i+1)

        IM[i]<=0;
end

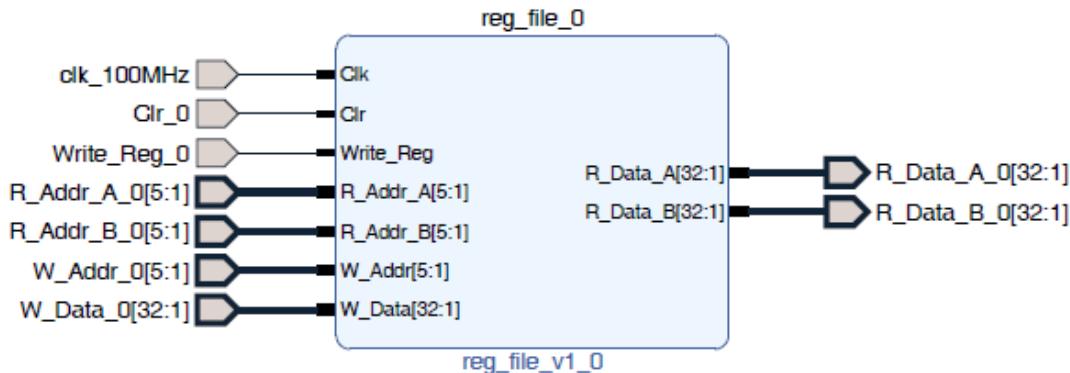
endmodule
```

为防止指令读取到最后出现数组越界的情况，把第79条之后的指令全部初始化为0。

#### 4、寄存器堆模块

寄存器组是指令操作的主要对象，MIPS 处理器里一共有 32 个 32 位的寄存器，故可

以声明一个包含 32 个 32 位的寄存器数组。读寄存器时需要 Rs, Rt 的地址，得到其数据。写寄存器 Rd 时需要所写地址，所写数据，同时需要写使能。以上所有操作需要在时钟和复位信号控制下操作。故寄存器组设计如下：



其中clk为时钟信号，clr为复位信号，当时钟上升沿到来时若写使能为1将所写数据写入写地址对应的寄存器，当复位信号的上升沿到来时所有寄存器的数据清零。

模块代码如下：

```

`timescale 1ns / 1ps//寄存器堆模块

module RegFile

    #(parameter ADDR = 5,          //寄存器编码、地址位宽
      parameter NUMB = 1<<ADDR,      //寄存器个数
      parameter SIZE = 32)           //寄存器数据位宽

    (  input Clk,                //写入时钟信号
      input Clr,                  //清零信号
      input Write_Reg,            //写控制信号
      input [ADDR-1:0] R_Addr_A,   //A端口读寄存器地址
      input [ADDR-1:0] R_Addr_B,   //B端口读寄存器地址
      input [ADDR-1:0] W_Addr,     //写寄存器地址

```

```

input [SIZE-1:0] W_Data,           //写入数据
output [SIZE-1:0] R_Data_A,       //A端口读出数据
output [SIZE-1:0] R_Data_B,       //B端口读出数据
output [SIZE-1:0] ra             //ra寄存器的值

);

reg [SIZE-1:0]REG_Files[0:NUMB-1];//寄存器堆本体
integer i;//用于遍历NUMB个寄存器

initial//初始化NUMB个寄存器，全为0
for(i=0;i<NUMB;i=i+1)
    REG_Files[i]<=0;

always@(posedge Clk or posedge Clr)begin//时钟信号或清零信号上升沿
    //清零
    if(Clr)
        for(i=0;i<NUMB;i=i+1)
            REG_Files[i]<=0;
    //不清零,检测写控制, 高电平则写入寄存器
    else if(Write_Reg)
        REG_Files[W_Addr]<=W_Data;
end      //读操作没有使能或时钟信号控制, 使用数据流建模(组合逻辑电路,读不需要时钟控制)

```

```
assign R_Data_A=REG_Files[R_Addr_A];  
assign R_Data_B=REG_Files[R_Addr_B];  
assign ra=REG_Files[SIZE-1];  
  
endmodule
```

采用参数控制地址位宽、数据位宽和寄存器个数，便于以后的拓展和修改。

## 5、符号拓展模块

根据符号拓展信号ExtOp对16位的立即数进行拓展，若ExtOp=0则进行0拓展，对高16位补0，若ExtOp=1则进行符号拓展，用立即数的最高位对高16位进行拓展。

模块代码如下：

```
`timescale 1ns / 1ps  
  
module signext(  
    input [15:0] inst, // 输入16位  
    input ExtOp,  
    output [31:0] data // 输出32位  
);  
  
// 根据符号补充符号位  
// 如果符号位为1，则补充16个1，即16'h ffff  
// 如果符号位为0，则补充16个0  
  
    assign data= inst[15:15]&ExtOp?{16'hffff,inst}:{16'h0000,inst};  
  
endmodule
```

## 6、ALU模块（包括寄存器hi、lo）

ALU的逻辑框图如图所示。

在图中各信号的功能：

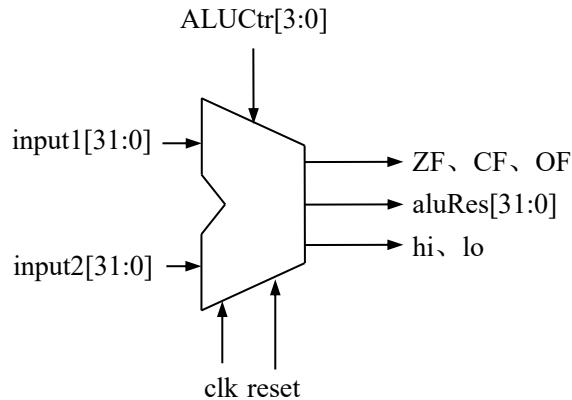


图 ALU 电路符号

- ✧ input1：操作数，32位；
- ✧ input2：操作数，32位；
- ✧ ALUCtr：5位操作码，输入；
- ✧ hi、lo：hi、lo寄存器的值，32位；
- ✧ aluRes：运算结果，32位；
- ✧ ZF：零标志和跳转信号，1位；当运算结果为0时，该位为1，否则该位为0；遇到跳转指令时（beq、bne、bltz等），该位标志着是否需要跳转，根据运算结果，ZF为1时表示需要跳转，否则不跳转；
- ✧ CF：进位标志位；
- ✧ OF：溢出标志位；
- ✧ clk：时钟信号，用于控制hi、lo寄存器的写操作；
- ✧ reset：复位信号，用于将hi、lo复位；

模块代码如下：

```

/*说明：
> sub 用于区别 madd 和 msub;
> hi、lo 是 hi 寄存器和 lo 寄存器本体，HI 和 LO 只是用于暂存计算结果，时钟上升沿到来时，若 hi、lo 的写使能 hiWE 为 1 则将 HI、LO 的值写入 hi、lo;
> 与加法、减法和乘法不同，Verilog HDL 没有自带的除法运算功能，故设计并使用除法器 IP 核 div_0;
> 对加法、减法来说无符号数和有符号数的运算在机器看来都一样，都是补码的运算，对除法也一样，但对乘法有区别（要对乘出来的项进行符号拓展，有符号数做符号拓展，）

```

- 无符号数做 0 拓展), 因此在运算时特别加以区分;  
 ➤ 若 input 和 aluctr 同时改变, 可能会出现数据冒险, 因此 ALU 的运算在被触发过后的 1ns 后, 也就是数据稳定之后再开始进行。\*/

```

`timescale 1ns / 1ps

module alu(
    input reset,
    input clk,
    input sub,
    input [4:0] shamt,
    input [31:0] input1,
    input [31:0] input2,
    input [4:0] aluCtr,
    output reg [31:0] aluRes,
    output reg ZF, CF, OF,
    output reg [31:0] hi, lo
);
    reg hlWE;
    reg [31:0] HI, LO;
    // 初始化
    initial begin
        HI <= 0;
        LO <= 0;
        hi <= 0;
        lo <= 0;
        hlWE <= 0;
        ZF <= 0;
        CF <= 0;
        OF <= 0;
        aluRes <= 0;
    end
    // 将 HI、LO 的值写入 hi、lo
    always@(posedge reset or posedge clk)begin
        if(reset)begin
            hi <= 0;

```

```

    lo <= 0;
end
else if(hlWE)begin
    hi <= HI;
    lo <= LO;
end
end
wire [31:0] yushu,shang;
//实例化除法器 IP 核
div_0 div(
    .a(input1),           //输入， 32 位
    .b(input2),           //输入， 32 位
    .yshang(shang),       //输出 a 除以 b 的商， 32 位
    .yyushu(yushu)        //输出 a 除以 b 的余数， 32 位 1
);
integer i;
always @(aluCtr or input1 or input2 or shamt ) // 运算数或控制码变化时操作
#1 //延时 1ns 以防止数据冒险
begin
    case(aluCtr)
        5'b00000: // and andi
            begin
                hlWE = 0;
                aluRes = input1 & input2;
                if(aluRes==0) ZF=1;
                else ZF=0;
            end
        5'b00001: // or ori
            begin
                hlWE = 0;
                aluRes = input1 | input2;
            end
    end
end

```

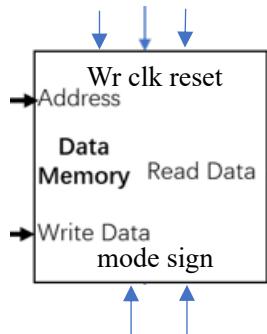
```
if(aluRes==0) ZF=1;
else ZF=0;
end

.....
5'b11110://maddu msubu
begin
    hlWE = 1;
    if(sub)
        {HI,LO} = ($unsigned({HI,LO})) - ($unsigned(input1)) *
($unsigned(input2));
    else
        {HI,LO} = ($unsigned({HI,LO})) + ($unsigned(input1)) *
($unsigned(input2));

    aluRes=LO;
    if(aluRes==0) ZF=1;
    else ZF=0;
end

default:
begin
    aluRes <= 0;
    ZF<=0;
    OF<=0;
    CF<=0;
    hlWE <= 0;
end
endcase
end
endmodule
```

## 7、数据存储器模块



- ✧ Address: 读或写的内存地址, 32 位
- ✧ WriteData: 要写入的数据, 32 位
- ✧ ReadData: 读出的数据, 32 位
- ✧ Wr: 写使能
- ✧ clk: 时钟信号, 当时钟上升沿到来时若写使能为 1 则将要写入的数据写入地址对应的内存单元中
- ✧ reset: 复位信号, reset 上升沿到来时每个内存单元的值都清零
- ✧ mode: 写入或读出模式控制信号, 为 00 时做字节操作, 01 时做半字操作, 1x 时为字操作, 2 位
- ✧ sign: 符号拓展信号, 在半字读或字节读的时候控制拓展模式, 为 1 时表示做符号拓展, 为 0 时表示做 0 拓展

模块代码如下:

```

module DM_unit(
    input clk,
    input Wr,
    input reset,
    input sign,
    input [1:0] mode,
    input [7:0] DMAadr,

```

```
input [31:0] wd,  
output reg [31:0] rd;  
  
reg [31:0] RAM[0:255];  
wire [31:0] target;  
  
//read  
assign target = RAM[DMAadr];  
  
always@(mode or target)begin  
    casex(mode)  
        2'b00: begin rd={{24{sign}}, target[7:0]}; end  
        2'b01: begin rd={{16{sign}}, target[15:0]}; end  
        2'b1x: begin rd=target; end  
        default: begin rd=target; end  
    endcase  
end  
  
//write  
integer i;  
always @(posedge clk,posedge reset)begin  
    if(reset)begin  
        for(i = 0; i < 256; i = i + 1)  
            RAM[i] = 0;  
    end  
    else if(Wr) begin  
        casex(mode)  
            2'b00: RAM[DMAadr][7:0] = wd[7:0];  
            2'b01: RAM[DMAadr][15:0] = wd[15:0];  
            2'b1x: RAM[DMAadr] = wd;  
            default: RAM[DMAadr] = wd;  
        endcase  
    end  
end
```

---

```
endmodule
```

## 8、PC 控制模块

根据控制信号和 ALU 运算结果等产生下一条 PC 的值, 若该指令为跳转指令, 则若 jmp 为 1, 则说明要发生跳转, 同时根据 jr 的值来选择进行寄存器跳转还是立即数跳转; 若 jmp 为 0 则根据 branch 来决定是否跳转, 若 branch 为 1 且 ALU 运算结果符合跳转条件 (用 zero 来标志) 则发生跳转, 否则不进行跳转, 下条指令为 PC+4。

模块代码如下:

```
'timescale 1ns / 1ps
module next_pc(
    input branch,
    input zero,
    input jmp,
    input jr,
    input [31:0] pc_Add4,
    input [31:0] expand,
    input [31:0] rs,
    input [31:0] instruction,
    output reg[31:0] next_pc
);

    wire PCSrc1, PCSrc2;
    wire [31:0] J_Addr,branch_addr;

    assign branch_addr = pc_Add4 + (expand << 2);
    assign J_Addr = jr? rs:{pc_Add4[31:28], instruction[25:0], 2'b00};

    //PC 的多选器
    assign PCSrc1 = (branch & zero)? 1'b1:1'b0;
    assign PCSrc2 = (jmp | jr)? 1'b1:1'b0;

    always@(*)begin
        casex({PCSrc2, PCSrc1})

```

```

    2'b00:next_pc<=pc_Add4;
    2'b01:next_pc<=branch_Addr;
    2'b1x:next_pc<=J_Addr;
    default:next_pc<=pc_Add4;

endcase
end
endmodule

```

## 9、PC 计数模块

用于 PC 的更新，当  
模块代码如下：

```

/*说明：每个时钟 clk 上升沿更新一次 PC，复位信号 reset 上升沿到来将 PC 复位*/
`timescale 1ns / 1ps
module pc_count(
    input [31:0] next_pc,
    input clk,
    input reset,
    output reg [31:0] pc
);
    always@(posedge clk or posedge reset)begin
        if(reset)
            pc=32'h00000000;
        else
            pc=next_pc;
    end
endmodule

```

## 10、数码管显示模块

七段译码显示的内容是16进制的，而ALU的运算结果是32位的，将运算结果分为高十六位和低十六位，分别传进七段译码模块；在顶层模块用一个开关up来选择是显示高16位还是低16位。sm\_wei 选择哪一个数码管亮， sm\_duan 选择数码管的哪一段亮， sm\_wei 变换的速度是 1 秒 1000 次，使人眼看起来数码管是同时显示数值的。

此处将display模块封装为IP核，在顶层模块中调用。为防止实验报告过于冗长，模块代

码不在实验报告里展示。

## 顶层模块 (TOP模块) 设计

顶层模块需要将前面的多个模块实例化后, 通过导线以及多路复用器将各部件链接起来, 并且在时钟的控制下修改 PC 的值, PC 是一个 32 位的寄存器, 每个时钟沿自动加 4 。

顶层模块需要输入时钟和复位信号, 然后首先读取 IM\_unit 的机器指令, 然后通过各个模块执行。

模块代码如下:

```
'timescale 1ns / 1ps

module top(
    input clk,           //clk for display
    input clkin,         //clkin for PC
    input reset,          //reset时寄存器堆、PC、hi、lo和存储器MEM都置初值
    input [3:0]SW,        //select the data
    input up,             //show upper bits
    output [6:0] seg,      //段码
    output [3:0] sm_wei,    //哪个数码管
    //以下是为了debug输出的信号
    //output [31:0] PC,
    //output [31:0] aluRes,
    //output [31:0] instruction,
    //output [31:0] next_pc,hi,lo,ra,input1, input2
```

```
);

//将要在数码管上展示的16位数

reg [15:0] data;

wire [31:0] PC;

wire [31:0] aluRes;

wire [31:0] instruction;

wire [31:0] next_pc,hi,lo,ra;

//PC+4的值

reg [31:0] pc_Add4;

always@(PC) begin

pc_Add4=PC+4;      //pc_Add4 stores the PC+4

end

// CPU 控制信号线, load在主控模块中译出, load2在ALU控制模块中译出

wire reg_dst, alu_src, memtoreg, regwrite, regwrite2, memWrite, branch, ExtOp,
jmp, jr,ld, nRW,load,load2;

//alu操作码、alu控制信号

wire [4:0] aluop, aluCtr;

//存储器读出数据

wire[31:0] memReadData;

//16位立即数经过32位符号拓展模块后的数据

wire [31:0] expand;
```

```
//寄存器写地址  
wire [4:0] regWriteAddr, regWriteAddr2;  
  
//寄存器写入数据  
wire [31:0] regWriteData, regWriteData2;  
  
//寄存器输出数据  
wire [31:0] RsData, RtData;  
  
//alu运算标志位  
wire ZF,OF,CF;  
  
//ALU的第二个操作数  
wire [31:0] operand2;  
  
//偏移量  
wire [4:0] shamt;  
  
//装数信号 (将PC+4写入寄存器ra) 在主控模块和ALU控制模块都能产生  
assign Id = (load | load2);  
  
//寄存器写信号由主控模块产生的信号和ALU控制模块产生的信号共同决定  
assign regwrite= regwrite2 & (!nRW);  
  
//写寄存器的目标寄存器来自rt或rd  
assign regWriteAddr2 = reg_dst ? instruction[15:11] : instruction[20:16];  
  
//若Id为1, 则目标寄存器为ra (第31个寄存器, 地址为11111)  
assign regWriteAddr = regWriteAddr2 | {5{Id}};  
  
//写入寄存器的数据来自ALU或数据寄存器  
assign regWriteData2 = memtoreg ? memReadData : aluRes;
```

```
//若Id为1，则写入数据为PC+4

assign regWriteData = Id? pc_Add4 : regWriteData2;

assign shamt = instruction[10:6];

//ALU的第二个操作数来自寄存器堆输出或指令低16位的符号扩展

assign operand2 = alu_src ? expand : RtData;

//以下对各个模块（共10个）分别实例化

//实例化PC计数模块

pc_count pcCount(
    .next_pc(next_pc), //input
    .clk(clkin),      //input
    .reset(reset),    //input
    .pc(PC)           //output
);

//实例化PC控制模块

next_pc nextPC(
    .reset(reset),          //input
    .branch(branch),        //input
    .zero(ZF),              //input
    .jmp(jmp),              //input
    .jr(jr),                //input
    .pc_Add4(pc_Add4),      //input
);
```

```

    .expand(expand),           //input
    .rs(RsData),              //input
    .instruction(instruction), //input
    .next_pc(next_pc)         //output
);

.....
//实例化数码管显示模块

display_0 Smg(
    .clk(clk),               //input
    .sm_wei(sm_wei),          //output
    .data(data),              //input
    .sm_duan(seg)             //output
);

endmodule

```

## 2、验证CPU正确性

CPU的核心部分是控制逻辑（包括主控模块和ALU控制模块），因此先对该模块进行仿真，确保该模块无误后再对整体进行仿真。

### 控制模块仿真文件ctrsim.v的设计

```

/*定义一些测试模块输入所用到的寄存器，用于产生对测试模块输入信号（即将测试模块
input类型信号改为reg类型信号）；定义用于观察的输出信号接到测试模块的输出（即将测
试模块output类型信号改为wire类型信号）*/

```

```
module ctrsim;

// Inputs

reg [5:0] opCode;
reg [5:0] funct;
reg [4:0] rt;

// Outputs

wire regDst;
wire aluSrc;
wire memToReg;
wire regWrite;
wire memWrite;
wire branch;
wire Extop;
wire Id,load,load2;
wire jmp;
wire [4:0] aluop;
wire [4:0] aluCtr;
wire jr;
wire nRW;

assign Id = load|load2;

// Instantiate the Unit Under Test (UUT) 例化测试模块

ctr uut (
```

```
.opCode(opCode),  
.regDst(regDst),  
.aluSrc(aluSrc),  
.memToReg(memToReg),  
.regWrite(regWrite),  
.memWrite(memWrite),  
.branch(branch),  
.aluop(aluop),  
.ExtOp(Extop),  
.load(load),  
.jmp(jmp)  
);  
  
aluctr uut1(  
.ALUOp(aluop),  
.funct(funct),  
.rt(rt),  
.ALUCtr(aluCtr),  
.jr(jr),  
.load2(load2),  
.nRW(nRW)  
);  
  
//开始测试时尽量穷举出所有可能的情况
```

```

initial begin

    // Initialize Inputs

    opCode = 6'b000000;

    .....

    #100; opCode = 6'b000001;

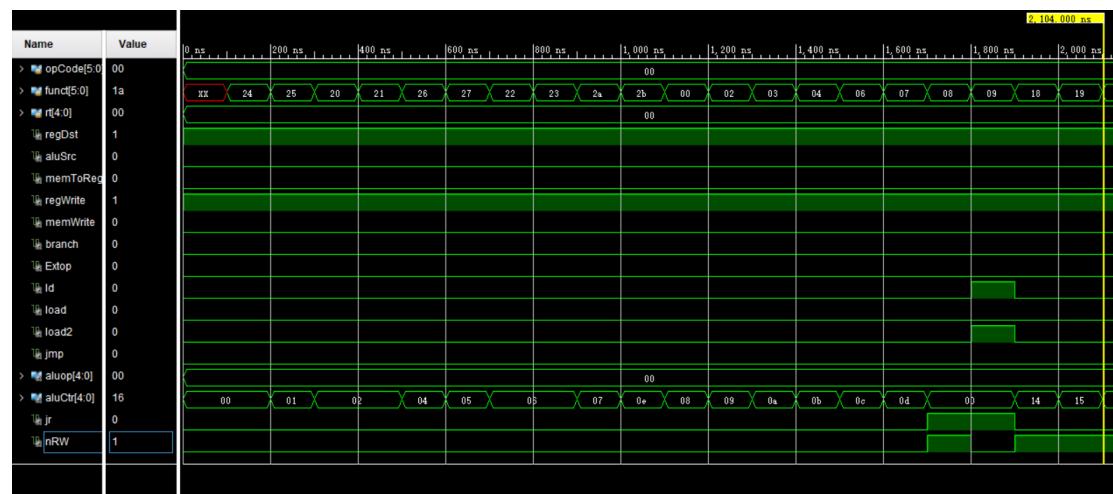
    .....

end

endmodule

```

开始仿真：



分析：

100~1700ns为R型指令的基本运算指令，根据不同的运算操作需求译出不同的aluCtr来控制ALU进行运算，并将运算结果写入rd中，因此regDst、regWrite为1；

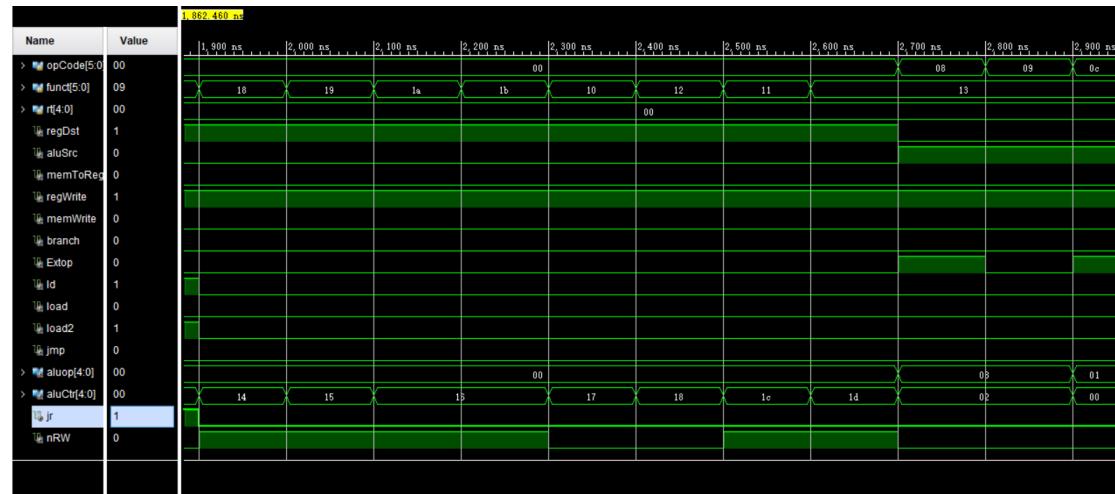
1700~1800ns为jr指令，由于jr为R型指令，在主控模块不能直接译出，因此regWrite为1，

但在ALU控制模块中识别出jr指令，因此nRW为1，阻止了寄存器堆被错误地写入数据，同时需要进行寄存器跳转，jr为1；

1800~1900ns为jalr指令，由于jr为R型指令，在主控模块不能直接译出，因此load为1，但

在ALU控制模块中识别出jalr指令，因此load2为1，故Id为1；同时需要进行寄存器跳转，jr

为1；



分析：

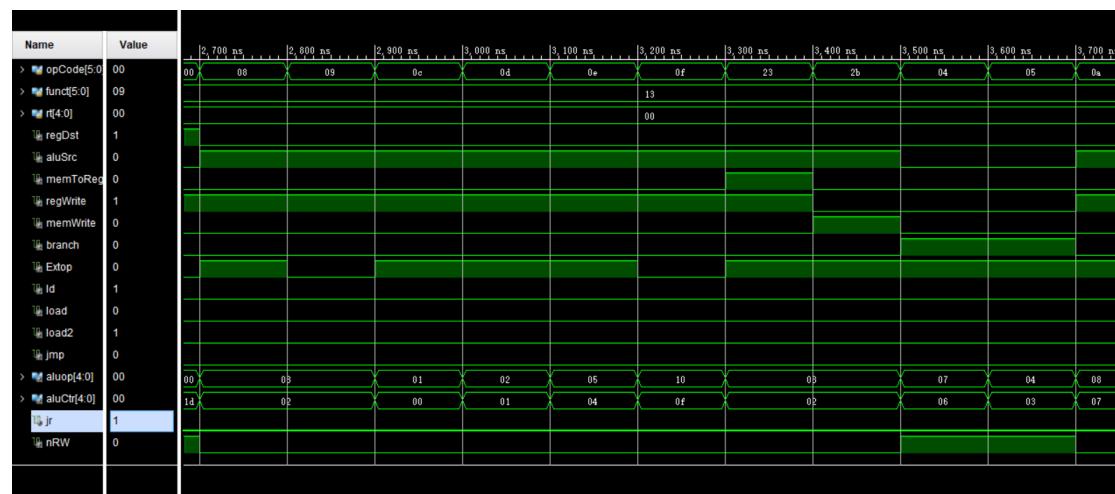
1900~2300ns分别为mult、 multu、 div、 divu指令，它们将结果放入hi、 lo中，故不需要

进行寄存器写操作，nRW=1；

2300~2400ns分别为mfhi、 mflo，需要进行寄存器写操作，因此nRW=0；

1900~2300ns分别为mthi、 mtlo指令，它们将输入数据放入hi或lo中，不需要进行寄存器

写操作，故nRW=1；



分析：

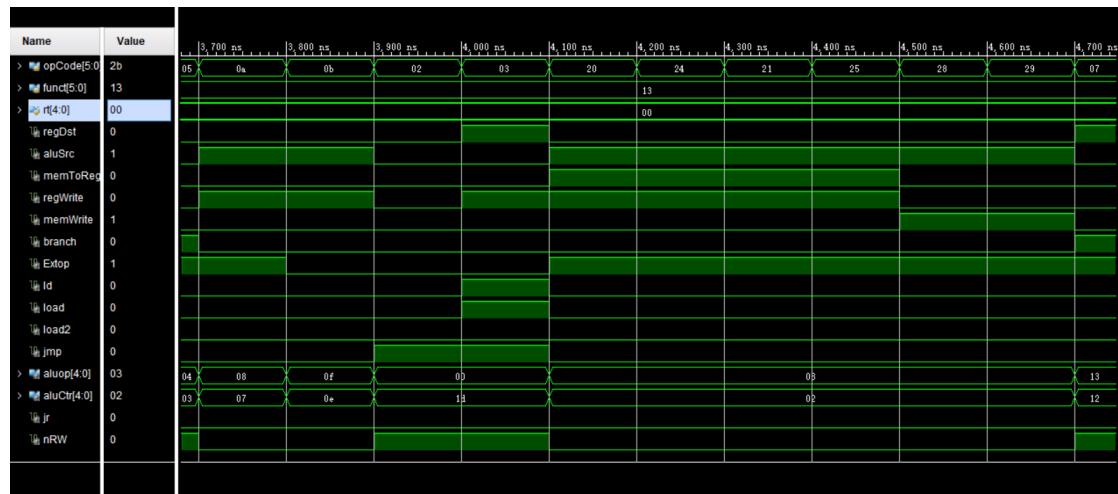
2700~3300ns分别为addi、 addiu、 andi、 ori、 xor、 lui指令，其中addiu指令不需要进行

符号拓展，lui指令不必做符号拓展，因此Extop为0；

3300~3400ns为lw指令，需要将读出的数据写入寄存器，memToReg=1；

3400~3500ns为sw指令，需要将数据写入存储器，memWrite=1；

3500~3700ns分别为beq、bne，分支指令，branch=1；



分析：

3700~3900ns分别为slti、sltiu指令，其中sltiu不需要进行符号拓展，因此Extop=0；

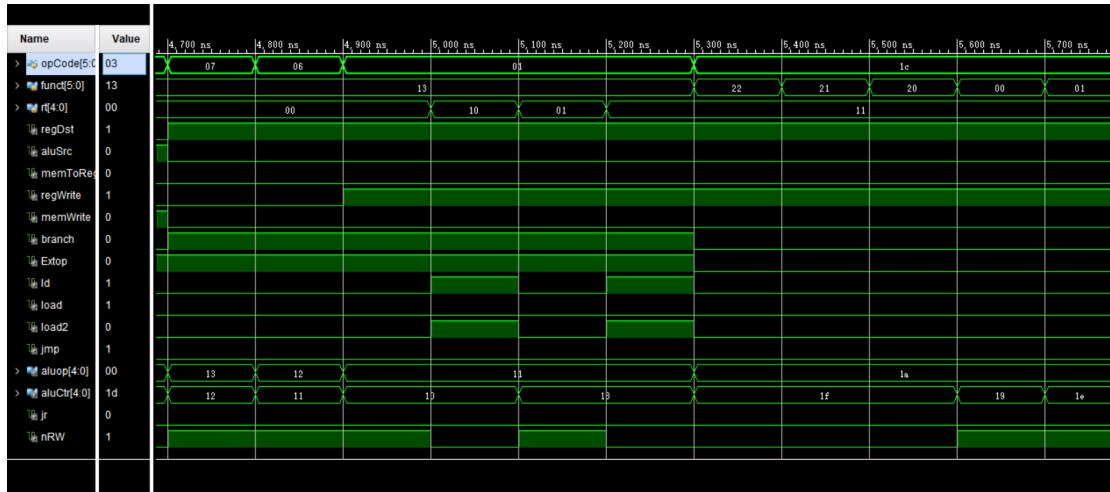
3900~4000ns为j指令，jmp=1；

4000~4100ns为jal指令，jmp=1，同时需要进行装数，在主控模块识别出jal指令故load=1，

因此Id=1；

4100~4500ns分别为lb、lbu、lh、lhu指令，需要使用存储器读出的数据，memToReg=1；

4500~4700ns分别为sb、sh指令，需要将数据写入存储器，memWrite=1；



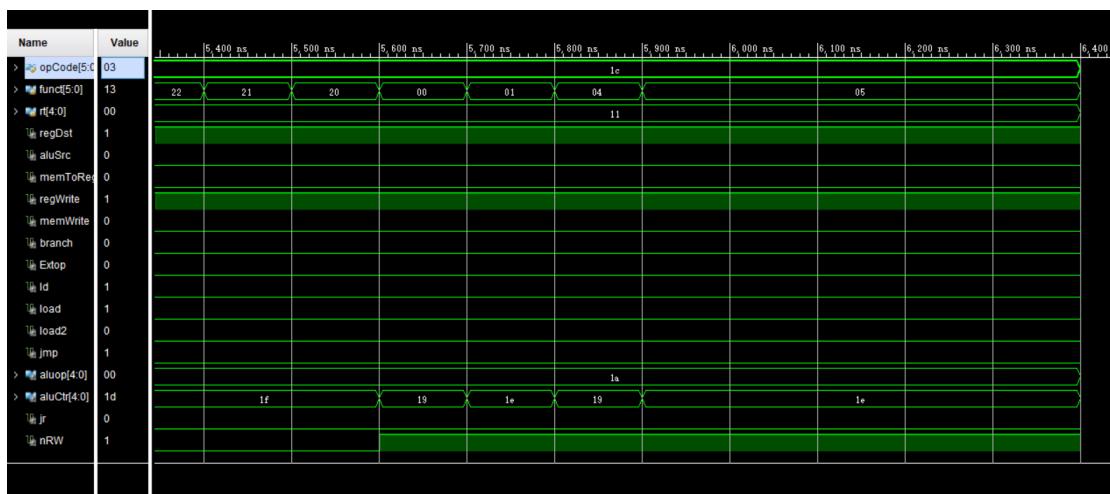
分析：

4700~4900ns分别为bgtz、blez指令，分支指令，branch=1；

4900~5300ns分别为bltz、bltzal、bgez、bgezal指令，分支指令，branch=1，它们的操作码相同，其中bltz、bgez在主控模块译码出regWrite信号，但实际上不需要写入寄存器，故nRW=1，而bltzal、bgezal需要将PC+4写入ra中，故nRW=0，Id=1；

5300~5600ns分别为mul、clo、clz指令，需要将运算结果写入rd，regDst=1，regWrite=1；

5600~5800ns分别为madd、maddu指令，不需要将结果写入寄存器，因此nRW=1；



分析：

5800~6000ns分别为msub、msubu指令，不需要将结果写入寄存器，因此nRW=1。

控制模块的仿真完成，仿真波形图符合预期，确保了控制信号正确无误，下面继续实验。

设计测试样例对各条指令进行测试，分别对各条指令进行验证。

### 对顶层模块进行仿真

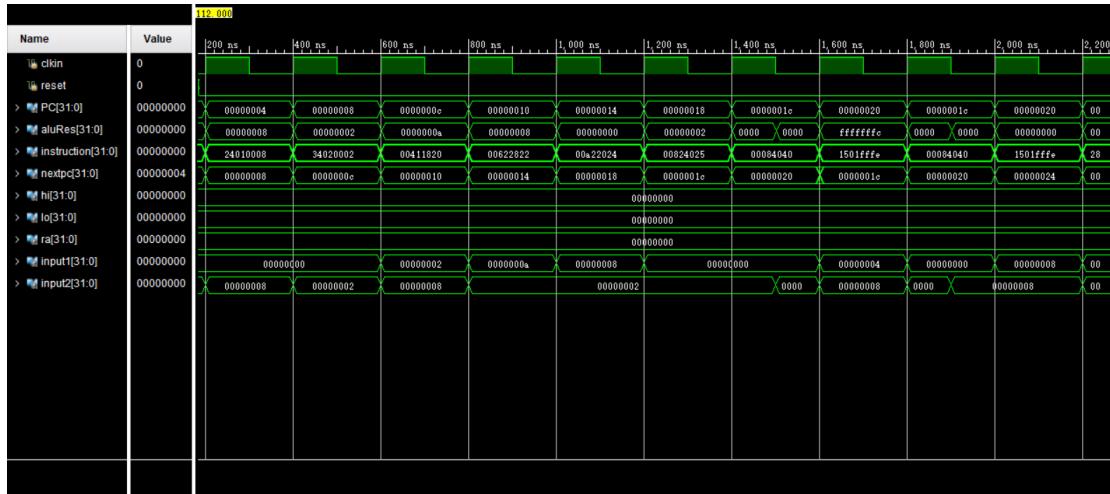
设计测试程序如下：

地址	汇编程序	指令编码	运行结果
0x00000000	nop	00000000	无
0x00000004	addiu \$1,\$0,8	24010008	\$1=8
0x00000008	ori \$2,\$0,2	34020002	\$2=2
0x0000000C	add \$3,\$2,\$1	00411820	\$3=10
0x00000010	sub \$5,\$3,\$2	00622822	\$5=8
0x00000014	and \$4,\$5,\$2	00a22024	\$4=0
0x00000018	or \$8,\$4,\$2	00824025	\$8=2
label1:0x0000001C	sll \$8,\$8,1	00084040	\$8=4
0x00000020	bne \$8,\$1,label1	1501ffff	转 1C
0x00000024	slti \$6,\$2,4	28460004	\$6=1
0x00000028	slti \$7,\$6,-1	28c7ffff	\$7=0
0x0000002C	sltiu \$6,\$6,-1	2cc6ffff	\$6=1
label2:0x00000030	addi \$7,\$7,8	20e70008	\$7=8
0x00000034	beq \$7,\$1,label2	10e1ffff	转 30
0x00000038	addi \$1,\$0,-1	2001ffff	\$1=-1
label3:0x0000003C	addiu \$10,\$10,1	254a0001	\$10=1
0x00000040	bltz \$10,label3	0540ffff	转 3C
0x00000044	andi \$11,\$2,2	304b0002	\$11=2
label4:0x00000048	jal label9	0c000038	转 e0, ra=4C
0x0000004C	xor \$8, \$2, \$11	004b4026	\$8=0
0x00000050	nor \$8, \$6, \$11	00cb4027	\$8=0xffffffff
0x00000054	addu \$8, \$8, \$11	010b4021	\$8=0xffffffff
0x00000058	subu \$8, \$8, \$6	01064023	\$8=0xffffffff
0x0000005C	srl \$1, \$1, 1	00010842	\$1=0xffffffff
0x00000060	sra \$8, \$8, 2	00084083	\$8=0xffffffff
0x00000064	lui \$9, 7	3c090007	\$9=0x00070000
0x00000068	slt \$10, \$1, \$8	0028502a	\$10=0
0x0000006C	sltu \$10, \$1, \$8	0028502b	\$10=1
0x00000070	sllv \$4, \$8, \$6	00c82004	\$4=0xffffffff
label5:0x00000074	addi \$4, \$4, 1	20840001	\$4=0xffffffff
0x00000078	blez \$4, label5	1880ffff	转 70
0x0000007C	bltzal \$8, label8	05100017	转 dc, ra=80
label6:0x00000080	addi \$2, \$2, -1	2042ffff	\$2=1
0x00000084	bgtz \$2, label6	1c40ffff	转 80
label7:0x00000088	addi \$11, \$11, -1	216bffff	\$11=\$11-1
0x0000008C	bgez \$11, label7	0561ffff	转 88

0x00000090	bgezal \$0, label8	04110012	转 dc, ra=94
0x00000094	srlv \$12, \$8, \$7	00e86006	\$12=0x0000ffff
0x00000098	sra v \$12, \$8, \$7	00e86007	\$12=0xffffffff
0x0000009C	clo \$2, \$12	71801021	\$2=32
0x000000A0	sll \$12, \$12, 4	000c6100	\$12=0xfffffffff0
0x000000A4	clo \$2, \$12	71801021	\$2=28
0x000000A8	clz \$2, \$8	71001020	\$2=0
0x000000AC	srl \$8, \$8, 7	000841c2	\$8=0x01fffff
0x000000B0	clz \$2, \$8	71001020	\$2=7
0x000000B4	mul \$1, \$2, \$3	70430802	\$1=70, hi=0, lo=70
0x000000B8	mult \$2, \$11	004b0018	hi=0xffffffff, lo=0xffffffff9
0x000000BC	multu \$2, \$11	004b0019	hi=0x00000006, lo=0xffffffff9
0x000000C0	div \$7, \$5	00e5001a	hi=0, lo=2
0x000000C4	divu \$1, \$7	0027001b	hi=6, lo=4
0x000000C8	mfhi \$4	00002010	\$4=6
0x000000CC	mflo \$6	00003012	\$6=4
0x000000D0	mthi \$5	00a00011	hi=8
0x000000D4	mtlo \$1	00200013	lo=46
0x000000D8	j end	0800003a	转 e8
label8:0x000000DC	jr \$ra	03e00008	转\$ra
label9:0x000000E0	addi \$30, \$ra, 0	23fe0000	\$30=\$ra
0x000000E4	jalr \$30	03c0f809	转\$30, ra=E8
end:0x000000E8	madd \$10, \$11	714b0000	hi=0x00000008, lo=0x00000045
0x000000EC	maddu \$10, \$11	714b0001	hi=0x00000009, lo=0x00000044
0x000000F0	msub \$5, \$6	70a60004	hi=0x00000009, lo=0x00000024
0x000000F4	msubu \$10, \$11	714b0005	hi=0x00000008, lo=0x00000025
0x000000F8	sw \$8, 0(\$5)	aca80000	将\$8 按字写入存储器
0x000000FC	sh \$8, 4(\$5)	a4a80004	将\$8 按半字写入存储器
0x00000100	sb \$8, 8(\$5)	a0a80008	将\$8 按字节写入存储器
0x00000104	lw \$1, 0(\$5)	8ca10000	\$1=0x01fffff
0x00000108	addi \$1, \$1, 0	20210000	\$1=0x01fffff
0x0000010C	lw \$1, 4(\$5)	8ca10004	\$1=0x0000ffff
0x00000110	addi \$1, \$1, 0	20210000	\$1=0x0000ffff
0x00000114	lw \$1, 8(\$5)	8ca10008	\$1=0x000000ff
0x00000118	addi \$1, \$1, 0	20210000	\$1=0x000000ff
0x0000011C	lh \$1, 0(\$5)	84a10000	\$1=0xffffffff
0x00000120	addi \$1, \$1, 0	20210000	\$1=0xffffffff
0x00000124	lhu \$1, 0(\$5)	94a10000	\$1=0x0000ffff
0x00000128	addi \$1, \$1, 0	20210000	\$1=0x0000ffff
0x0000012C	lb \$1, 0(\$5)	80a10000	\$1=0xffffffff
0x00000130	addi \$1, \$1, 0	20210000	\$1=0xffffffff
0x00000134	lbu \$1, 0(\$5)	90a10000	\$1=0x000000ff
0x00000138	addi \$1, \$1, 0	20210000	\$1=0x000000ff

0x00000013C	xori \$1, \$11, -1	3981ffff	\$1=0xffffffff0
-------------	--------------------	----------	-----------------

下面开始仿真。



分析：

第1个时钟周期 (0~200ns) : nop

地址为0x00000000, 该指令编码为0x 00000000, 不进行任何操作;

第2个时钟周期 (200~400ns) : addiu \$1,\$0,8

地址为0x00000004, 该指令编码为0x 24010008, \$0和立即数8相加结果为8, aluRes=8;

第3个时钟周期 (400~600ns) : ori \$2,\$0,2

地址为0x00000008, 该指令编码为0x 34020002, \$0和立即数2相或结果为2, aluRes=2;

第4个时钟周期 (600~800ns) : add \$3,\$2,\$1

地址为0x0000000c, 该指令编码为0x 00411820, \$1的值为8, \$2的值为2, 因此\$1和\$2

相加结果为10, aluRes=10;

第5个时钟周期 (800~1000ns) : sub \$5,\$3,\$2

地址为0x00000010, 该指令编码为0x 00622822, \$3的值为10, \$2的值为2, 因此\$3-

\$2=8, aluRes=8;

第6个时钟周期 (1000~1200ns) : and \$4,\$5,\$2

地址为0x00000014，该指令编码为0x 00a22024，\$5的值为8，\$2的值为2，因此\$5和\$2相与结果为0，aluRes=0；

第7个时钟周期 (1200~1400ns) : or \$8,\$4,\$2

地址为0x00000018，该指令编码为0x 00824025，\$4的值为0，\$2的值为2，因此\$4和\$2相或结果为2，aluRes=2；

Name	Value	1,400 ns	1,500 ns	1,600 ns	1,700 ns	1,800 ns	1,900 ns	2,000 ns	2,100 ns	2,200 ns	2,300 ns	2,400 ns
ls ckin	1											
ls reset	0											
> PC[31:0]	00000028		0000001c		00000020		0000001c		00000020		00000024	
> aluRes[31:0]	00000000		00000004		00000008		fffffff0		00000008		00000000	
> instruction[31:0]	28c7ff		00084040		1501ffff		00084040		1501ffff		28460004	
> nextpc[31:0]	0000002c		00000020		0000001c		00000020		00000024		00000028	
> hi[31:0]	00000000						00000000					
> lo[31:0]	00000000						00000000					
> ra[31:0]	00000000						00000000					
> input1[31:0]	00000001		00000000		00000004		00000000		00000008		00000002	
> input2[31:0]	fffff		00000002		00000004		00000008		00000008		00000004	

分析：

第8个时钟周期 (1400~1600ns) : sll \$8,\$8,1

地址为0x0000001c，该指令编码为0x 00084040，\$8的值为2，左移1位后为4，aluRes=4；

第9个时钟周期 (1600~1800ns) : bne \$8,\$1,-2

地址为0x00000020，该指令编码为0x 1501ffff，\$8的值为4，\$1的值为8，\$8≠\$1，故在此处进行跳转，下一条指令地址为0x0000001c；

第10个时钟周期 (1800~2000ns) : sll \$8,\$8,1

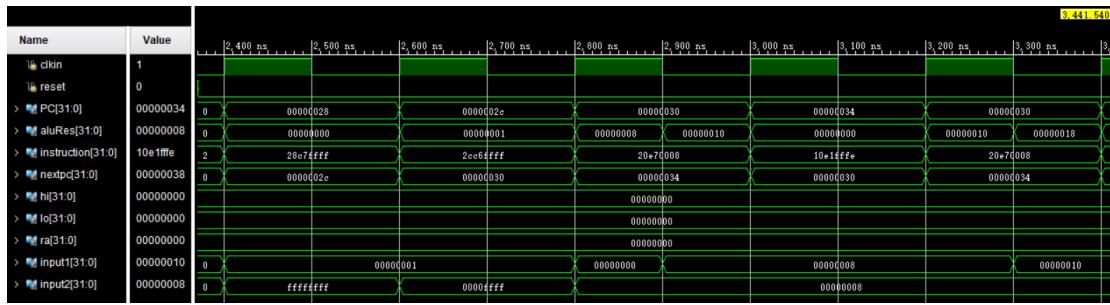
地址为0x0000001c，该指令编码为0x 00084040，\$8的值为4，左移1位后为8，aluRes=8；

第11个时钟周期 (2000~2200ns) : bne \$8,\$1,-2

地址为0x00000020，该指令编码为0x 1501ffff，\$8的值为8，\$1的值为8，\$8=\$1，故在此处不进行跳转，下一条指令地址为0x00000024；

第12个时钟周期 (2200~2400ns) : slti \$6,\$2,4

地址为0x00000020，该指令编码为0x 28460004，\$2=2<4，因此运算结果为1，aluRes=1；



分析：

第13个时钟周期 (2400~2600ns) : slti \$7,\$6,-1

地址为0x00000028, 该指令编码为0x 28c7ffff, \$6=1>-1, 因此运算结果为0, aluRes=0;

第14个时钟周期 (2600~2800ns) : sltiu \$6,\$6,-1

地址为0x0000002c, 该指令编码为0x 2cc6ffff, 对-1做无符号拓展结果为0x0000ffff,

\$6=0x00000001<0x0000ffff, 因此运算结果为1, aluRes=1;

第15个时钟周期 (2800~3000ns) : addi \$7,\$7,8

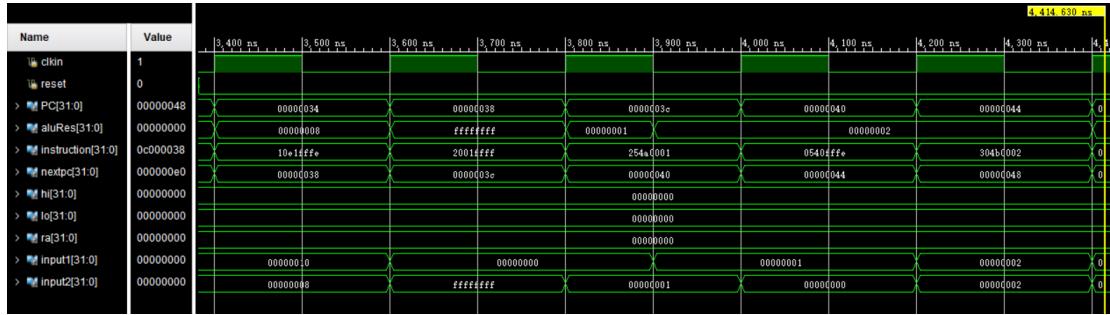
地址为0x00000030, 该指令编码为0x 20e70008, \$7的值为0, 0+8=8, 因此运算结果为8, aluRes=8;

第16个时钟周期 (3000~3200ns) : beq \$7,\$1,-2

地址为0x00000034, 该指令编码为0x 10e1ffe, \$7的值为8, \$1的值为8, \$7=\$1, 故在此处进行跳转, 下一条指令地址为0x00000030;

第17个时钟周期 (3200~3400ns) : addi \$7,\$7,8

地址为0x00000030, 该指令编码为0x 20e70008, \$7的值为8, 8+8=16, 因此运算结果为16, aluRes=16;



分析：

第18个时钟周期 (3400~3600ns) : beq \$7,\$1,-2

地址为0x00000034, 该指令编码为0x 10e1ffe, \$7的值为16, \$1的值为8, \$7≠\$1, 故在此处不进行跳转, 下一条指令地址为0x00000038;

第19个时钟周期 (3600~3800ns) : addi \$1,\$0,-1

地址为0x00000038, 该指令编码为0x 2001ffff, 运算结果为-1, aluRes=-1;

第20个时钟周期 (3800~4000ns) : addiu \$10,\$10,1

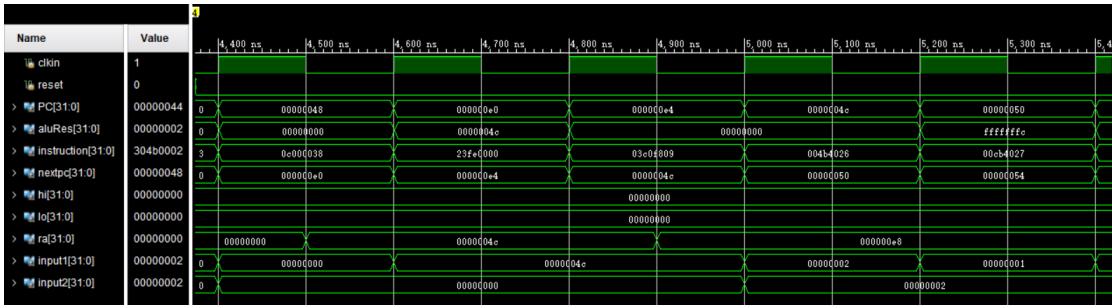
地址为0x0000003c, 该指令编码为0x 254a0001, 运算结果为1, aluRes=1;

第21个时钟周期 (4000~4200ns) : bltz \$10,-2

地址为0x00000040, 该指令编码为0x 0540ffff, \$10=1>0, 故在此处不进行跳转, 下一条指令地址为0x00000044;

第22个时钟周期 (4200~4400ns) : andi \$11,\$2,2

地址为0x00000044, 该指令编码为0x 304b0002, \$2的值为2, 2&2=2, aluRes=2;



分析：

第23个时钟周期 (4400~4600ns) : jal 0x000000e0

地址为0x00000048，该指令编码为0x 0c000038，下一条指令地址为0x000000e0；同时将当前PC+4写入ra中，ra=0x0000004c；

第24个时钟周期 (4600~4800ns) : addi \$30, \$ra, 0

地址为0x000000e0，该指令编码为0x 23fe0000，目的是将\$ra复制到\$30中，运算结果为0x0000004c，aluRes=0x0000004c；

第25个时钟周期 (4800~5000ns) : jalr \$30

地址为0x000000e4，该指令编码为0x 03c0f809，跳转到寄存器\$30中的地址，下一条指令地址为0x0000004c；同时将当前PC+4写入ra中，ra=0x000000e8；

第26个时钟周期 (5000~5200ns) : xor \$8, \$2, \$11

地址为0x0000004c，该指令编码为0x 004b4026，\$2的值为2，\$11的值为2（第22个时钟周期的运算结果），因此\$2和\$11异或结果为0，aluRes=0；

第27个时钟周期 (5200~5400ns) : nor \$8, \$6, \$11

地址为0x00000050，该指令编码为0x 00cb4027，\$6的值为1（第14个时钟周期的运算结果），\$11的值为2，因此\$2和\$11或非结果为0xfffffff0，aluRes=0xfffffff0；

Name	Value	[5,400 ns]	[5,500 ns]	[5,600 ns]	[5,700 ns]	[5,800 ns]	[5,900 ns]	[6,000 ns]	[6,100 ns]	[6,200 ns]	[6,300 ns]	[6,400 ns]
clk	1											
reset	0											
> PC[31:0]	00000044	00000054		00000058		0000005c		00000060		00000064		0
> aluRes[31:0]	00000002	fffffff4	00000000	fffffff4	fffffff0	7fffffff	3fffffff	fffffff4	fffffff4	00070000		0
> instruction[31:0]	304b0002	010b4021		01064023		00010842		00084083		3c090007		0
> nextpc[31:0]	00000048	00000058		0000005e		00000060		00000064		00000068		0
> hi[31:0]	00000000					00000000						
> lo[31:0]	00000000					00000000						
> ra[31:0]	00000000					00000008						
> input1[31:0]	00000002	fffffff4	fffffff4	fffffff4			00000000					7
> input2[31:0]	00000002	00000002		00000001	fffffff4	7fffffff	fffffff4	fffffff4	fffffff4	00000007		f

分析：

第28个时钟周期 (5400~5600ns) : addu \$8, \$8, \$11

地址为0x00000054, 该指令编码为0x 010b4021, \$8的值为0xfffffff4 (第27个时钟周期的运算结果), \$11的值为2, 因此\$8和\$11按无符号数相加结果为0xffffffff, aluRes=0xffffffff;

第29个时钟周期 (5600~5800ns) : subu, \$8, \$8, \$6

地址为0x00000058, 指令编码为0x 01064023, \$8的值为0xffffffff, \$6值为1, 此\$8-\$6=0xfffffff4, aluRes=0xfffffff4;

第30个时钟周期 (5800~6000ns) : srl \$1, \$1, 1

地址为0x0000005c, 指令编码为0x 00010842, \$1的值为0xffffffff, 右移1位结果为0x7fffffff, aluRes=0x7fffffff;

第31个时钟周期 (6000~6200ns) : sra \$8, \$8, 2

地址为0x00000060, 指令编码为0x 00084083, \$8的值为0xfffffff4, 算数右移2位结果为0xffffffff, aluRes=0xffffffff;

第32个时钟周期 (6200~6400ns) : lui \$9, 0x0007

地址为0x00000064, 指令编码为0x 3c090007, 将16位立即数放入运算结果高16位为0x00070000, aluRes=0x00070000;

Name	Value	[6,400 ns]	[6,500 ns]	[6,600 ns]	[6,700 ns]	[6,800 ns]	[6,900 ns]	[7,000 ns]	[7,100 ns]	[7,200 ns]	[7,300 ns]	[7,400 ns]
clk	1											
reset	0											
PC[31:0]	00000044	00000068		0000006c		00000070		00000074		00000078		000
aluRes[31:0]	00000002	00000000		00000001		fffffe*		fffffe*		00000000		000
instruction[31:0]	304b0002	0028502a		0028502b		00c82004		20840001		1880ffe*		208
nextpc[31:0]	00000048	0000006c		00000070		00000074		00000078		00000074		000
hi[31:0]	00000000					00000000						
lo[31:0]	00000000					00000000						
ra[31:0]	00000000					00000008						
input1[31:0]	00000002		7fffffff			00000001	fffffe*		fffffe*			
input2[31:0]	00000002		fffffe*					00000001		00000000		000

分析：

第33个时钟周期 (6400~6600ns) : slt \$10, \$1, \$8

地址为0x00000068, 指令编码为0x 0028502a, \$1=0x7fffffff, \$8=0xffffffff, \$1>\$8,

因此运算结果为0, aluRes=0;

第34个时钟周期 (6600~6800ns) : sltu \$10, \$1, \$8

地址为0x0000006c, 指令编码为0x 0028502b, \$1=0x7fffffff, \$8=0xffffffff, 把0xffffffff

看做无符号数它是32位数能表示的最大的数, 因此\$1<\$8, 因此运算结果为1, aluRes=1;

第35个时钟周期 (6800~7000ns) : sllv \$4, \$8, \$6

地址为0x00000070, 指令编码为0x 00c82004, \$8=0xffffffff, \$6=1, 将\$8左移1位结果

为0xfffffff, aluRes=0xfffffff;

第36个时钟周期 (7000~7200ns) : addi \$4, \$4, 1

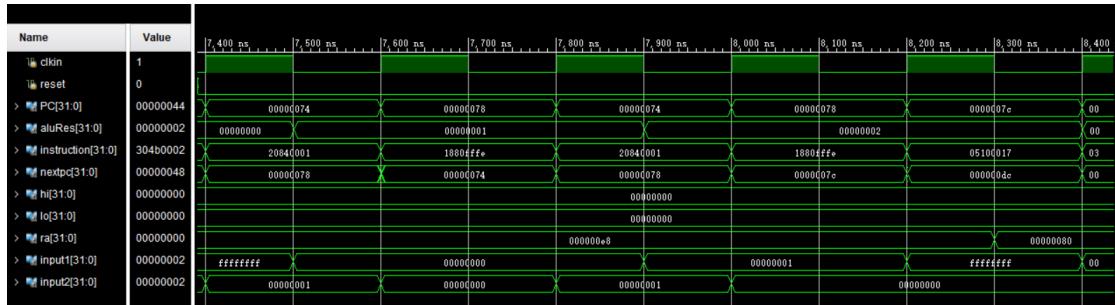
地址为0x00000074, 指令编码为0x 20840001, \$4=-2 (第35个时钟周期运算结果),

-2+1=-1, 运算结果为0xffffffff, aluRes=0xffffffff;

第37个时钟周期 (7200~7400ns) : blez \$4, -2

地址为0x00000078, 指令编码为0x 1880ffe, \$4=-1 (第36个时钟周期运算结果),

-1≤0, 故在此处进行跳转, 下一条指令地址为0x00000074;



分析：

第38个时钟周期 (7400~7600ns) : addi \$4, \$4, 1

地址为0x00000074, 指令编码为0x 20840001, \$4=-1 (第36个时钟周期运算结果),

-1+1=0, 运算结果为0, aluRes=0;

第39个时钟周期 (7600~7800ns) : blez \$4, -2

地址为0x00000078, 指令编码为0x 1880ffff, \$4=0 (第38个时钟周期运算结果),

0≤0, 故在此处进行跳转, 下一条指令地址为0x00000074;

第40个时钟周期 (7800~8000ns) : addi \$4, \$4, 1

地址为0x00000074, 指令编码为0x 20840001, \$4=0 (第38个时钟周期运算结果),

0+1=1, 运算结果为1, aluRes=1;

第41个时钟周期 (8000~8200ns) : blez \$4, -2

地址为0x00000078, 指令编码为0x 1880ffff, \$4=1 (第40个时钟周期运算结果), 1>0,

故在此处不进行跳转, 下一条指令地址为0x0000007c;

第42个时钟周期 (8200~8400ns) : bltzal \$8, 17

地址为0x0000007c, 指令编码为0x 05100017, \$8=-1 (第31个时钟周期运算结果), -

1<0, 故在此处进行跳转, 下一条指令地址为0x000000dc, 同时将PC+4写入ra中,

ra=0x00000080;

Name	Value	8,400 ns	8,500 ns	8,600 ns	8,700 ns	8,800 ns	8,900 ns	9,000 ns	9,100 ns	9,200 ns	9,300 ns	9,400 ns
ckin	1											
reset	0											
> PC[31:0]	00000044	0000004c	00000080		00000084	00000080		00000084		0		
> aluRes[31:0]	00000002	00000000	00000001		00000000			ffffffffff		0		
> instruction[31:0]	304b0002	05e00088	2042ffff		1c40ffff	2042ffff		1c40ffff		2		
> nextpc[31:0]	00000048	00000080	00000084		00000080	00000084		00000088		0		
> hi[31:0]	00000000				00000000							
> lo[31:0]	00000000				00000000							
> ra[31:0]	00000000				00000000							
> input1[31:0]	00000002	00000080	00000002		00000001			00000000		0		
> input2[31:0]	00000002	00000000	ffffffffff		00000000	ffffffffff		00000000		f		

分析：

第43个时钟周期 (8400~8600ns) : jr \$ra

地址为0x000000dc, 指令编码为0x 03e00008, 跳转到ra中的地址, 下一条指令地址为

0x00000080;

第44个时钟周期 (8600~8800ns) : addi \$2, \$2, -1

地址为0x00000080, 指令编码为0x 2042ffff, \$2的值为2, 2-1=1, 运算结果为1,

aluRes=1;

第45个时钟周期 (8800~9000ns) : bgtz \$2, -2

地址为0x00000084, 指令编码为0x 1c40ffff, \$2=1 (第44个时钟周期运算结果), 1>0,

故在此处进行跳转, 下一条指令地址为0x00000080;

第46个时钟周期 (9000~9200ns) : addi \$2, \$2, -1

地址为0x00000080, 指令编码为0x 2042ffff, \$2的值为1, 1-1=0, 运算结果为0,

aluRes=0;

第47个时钟周期 (9200~9400ns) : bgtz \$2, -2

地址为0x00000084, 指令编码为0x 1c40ffff, \$2=0 (第44个时钟周期运算结果), 故在

此处不进行跳转, 下一条指令地址为0x00000088;



分析：

第48个时钟周期 (9400~9600ns) : addi \$11, \$11, -1

地址为0x00000088, 指令编码为0x 216bffff, \$11=2 (第22个时钟周期运算结果), 2-1=1, 运算结果为1, aluRes=1;

第49个时钟周期 (9600~9800ns) : bgez \$11, -2

地址为0x0000008c, 指令编码为0x 0561fffe, \$11=1 (第48个时钟周期运算结果), 1≥0, 故在此处进行跳转, 下一条指令地址为0x00000088;

第50个时钟周期 (9800~10000ns) : addi \$11, \$11, -1

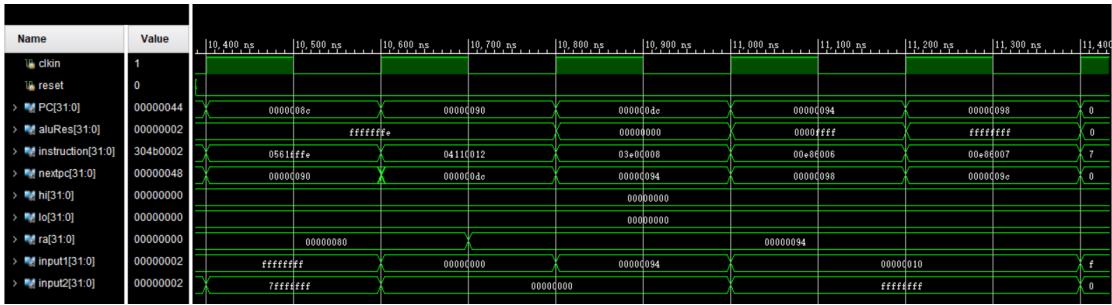
地址为0x00000088, 指令编码为0x 216bffff, \$11=1 (第48个时钟周期运算结果), 1-1=0, 运算结果为0, aluRes=0;

第51个时钟周期 (10000~10200ns) : bgez \$11, -2

地址为0x0000008c, 指令编码为0x 0561fffe, \$11=0 (第50个时钟周期运算结果), 0≥0, 故在此处进行跳转, 下一条指令地址为0x00000088;

第52个时钟周期 (10200~10400ns) : addi \$11, \$11, -1

地址为0x00000088, 指令编码为0x 216bffff, \$11=0 (第50个时钟周期运算结果), 0-1=-1, 运算结果为-1, aluRes=-1;



分析：

第53个时钟周期 (10400~10600ns) : bgez \$11, -2

地址为0x0000008c, 指令编码为0x 0561ffff, \$11=-1(第52个时钟周期运算结果), -1<0,

故在此处不跳转, 下一条指令地址为0x00000090;

第54个时钟周期 (10600~10800ns) : bgezal \$0, 12

地址为0x00000090, 指令编码为0x 04110012, \$0=0, 0≥0, 故在此处跳转, 下一条指

令地址为0x000000dc; 同时将PC+4存入ra中, ra=0x00000094;

第55个时钟周期 (10800~11000ns) : jr \$ra

地址为0x000000dc, 指令编码为0x 03e00008, 跳转到ra中的地址, 下一条指令地址为

0x00000094;

第56个时钟周期 (11000~11200ns) : srlv \$12, \$8, \$7

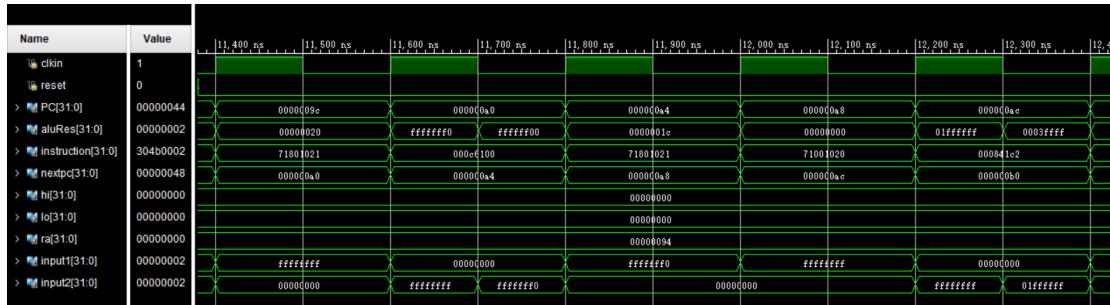
地址为0x00000094, 指令编码为0x 00e86006, \$8=0xffffffff, \$7=16, 将\$8右移16位结

果为0x0000ffff, aluRes=0x0000ffff;

第57个时钟周期 (11200~11400ns) : srav \$12, \$8, \$7

地址为0x00000098, 指令编码为0x 00e86007, \$8=0xffffffff, \$7=16, 将\$8算数右移16

位结果为0xffffffff, aluRes=0xffffffff;



分析：

第58个时钟周期 (11400~11600ns) : clo \$2, \$12

地址为0x0000009c, 指令编码为0x 71801021, \$12=0xffffffff, 从最高位开始数起共有32

个连续的1, 结果为32, aluRes=32;

第59个时钟周期 (11600~11800ns) : sll \$12, \$12, 4

地址为0x000000a0, 指令编码为0x 000c6100, \$12=0xffffffff, 左移4位后为0xfffffff0,

aluRes=0xfffffff0;

第60个时钟周期 (11800~12000ns) : clo \$2, \$12

地址为0x000000a4, 指令编码为0x 71801021, \$12=0xfffffff0, 从最高位开始数起共有

28个连续的1, 结果为28, aluRes=28;

第61个时钟周期 (12000~12200ns) : clz \$2, \$8

地址为0x000000a8, 指令编码为0x 71001020, \$8=0xffffffff, 从最高位开始数起共有0个

连续的0, 结果为0, aluRes=0;

第62个时钟周期 (12200~12400ns) : srl \$8, \$8, 7

地址为0x000000ac, 指令编码为0x 000841c2, \$8=0xffffffff, 右移7位后为0x01ffff,

aluRes=0x01ffff;

Name	Value	12,400 ns	12,500 ns	12,600 ns	12,700 ns	12,800 ns	12,900 ns	13,000 ns	13,100 ns	13,200 ns	13,300 ns	13,400 ns
clk	1											
reset	0											
> PC[31:0]	00000044		000000b0		00000054		00000088		000000bc		000000e0	00
> aluRes[31:0]	00000002		00000007		00000046		fffffff9		00000002		00	
> instruction[31:0]	304b0002		71001020		70430802		004b0018		004b0019		00e5001a	00
> nextpc[31:0]	00000048		000000b4		000000b8		000000c0		000000c0		000000c4	00
> hi[31:0]	00000000		00000000		00000046		fffffff9		00000006		00000000	
> lo[31:0]	00000000		00000000		00000046		fffffff9		00000002		00000002	
> ra[31:0]	00000002		01ffffff		00000007		fffffff9		00000010		00	
> input1[31:0]	00000002		00000000		0000000a		fffffff9		00000008		00	
> input2[31:0]	00000002		00000000		fffffff9		fffffff9		00000008		00	

分析：

第63个时钟周期 (12400~12600ns) : clz \$2, \$8

地址为0x000000b0, 指令编码为0x 71001020, \$8=0 x01ffff, 从最高位开始数起共有

7个连续的0, 结果为7, aluRes=7;

第64个时钟周期 (12600~12800ns) : mul \$1, \$2, \$3

地址为0x000000b4, 指令编码为0x 70430802, \$2=7, \$3=10,  $7 \times 10 = 70$ , 运算结果为

70, aluRes=70 (0x00000046); 同时将运算结果写入hi、lo中, hi=0x00000000, lo=0x00000046;

第65个时钟周期 (12800~13000ns) : mult \$2, \$11

地址为0x000000b8, 指令编码为0x 004b0018, \$2=7, \$11=-1,  $7 \times (-1) = -7$ , 运算结果为-7, aluRes=-7 (0xfffffff9); 同时将运算结果写入hi、lo中, hi=0xffffffff, lo=0xfffffff9;

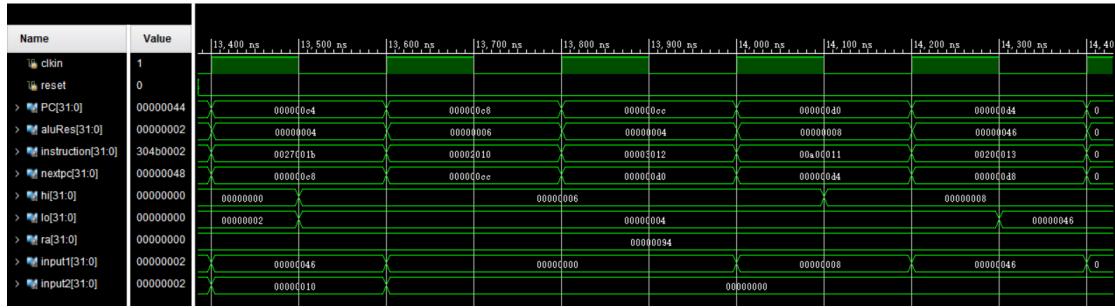
第66个时钟周期 (13000~13200ns) : multu \$2, \$11

地址为0x000000bc, 指令编码为0x 004b0019, \$2=7, \$11=0xffffffff,  $0x00000007 \times 0xffffffff = 0x00000006fffff9$ , aluRes=0x00000006fffff9; 同时将运算结果写入hi、lo中, hi=0x00000006, lo=0xfffffff9;

第67个时钟周期 (13200~13400ns) : div \$7, \$5

地址为0x000000c0, 指令编码为0x 00e5001a, \$7=16, \$5=8,  $16 \div 8 = 2$ , aluRes=2;

同时将余数和商分别写入hi、lo中, hi=0, lo=2;



分析：

第68个时钟周期 (13400~13600ns) : divu \$1, \$7

地址为0x000000c4, 指令编码为0x 0027001b, \$1=70, \$7=16,  $70 \div 16 = 4$ 余6, aluRes=6;

同时将余数和商分别写入hi、lo中, hi=6, lo=4;

第69个时钟周期 (13600~13800ns) : mfhi \$4

地址为0x000000c8, 指令编码为0x 00002010, ALU输出结果为hi的值, 并将该值写入寄存器\$4中；

第70个时钟周期 (13800~14000ns) : mflo \$6

地址为0x000000cc, 指令编码为0x 00003012, ALU输出结果为lo的值, 并将该值写入寄存器\$6中；

第71个时钟周期 (14000~14200ns) : mthi \$5

地址为0x000000d0, 指令编码为0x 00a00011, \$5的值为8, 将\$5的值写入hi中, hi=8;

第72个时钟周期 (14200~14400ns) : mtlo \$1

地址为0x000000d4, 指令编码为0x 00200013, \$1的值为70, 将\$1的值写入lo中, lo = 70;

Name	Value	14,400 ns	14,500 ns	14,600 ns	14,700 ns	14,800 ns	14,900 ns	15,000 ns	15,100 ns	15,200 ns	15,300 ns	15,400 ns
clk	1											
reset	0											
> PC[31:0]	00000044	000000d8		000000e8		000000ec		000000f0		000000f4		000
> aluRes[31:0]	00000002	00000000		00000045		00000044		00000024		00000025		000
> instruction[31:0]	304b0002	0800003a		714b0000		714b0001		70a60004		714b0005		a5a
> nextpc[31:0]	00000048	000000a8		000000ec		000000f0		000000f4		000000f8		000
> hi[31:0]	00000000	00000008						00000009			00000008	
> lo[31:0]	00000000	00000046		00000045		00000044		00000024		00000025		
> ra[31:0]	00000000					00000094						
> input1[31:0]	00000002	00000000		00000001		00000008		00000001			000	
> input2[31:0]	00000002	00000000		ffffffffff		00000004		ffffffffff			000	

分析：

第73个时钟周期 (14400~14600ns) : j 0x0000000e8

地址为0x000000d8, 指令编码为0x 0800003a, PC跳转到0x000000e8, 故下一条指令为0x000000e8;

第74个时钟周期 (14600~14800ns) : madd \$10, \$11

地址为0x000000e8, 指令编码为0x 714b0000, \$10=1, \$11=-1,  $1 \times (-1) = -1$ , 将运算结果与hi、lo相加, 结果lo的值-1, hi=0x00000008, lo=0x00000045;

第75个时钟周期 (14800~15000ns) : maddu \$10, \$11

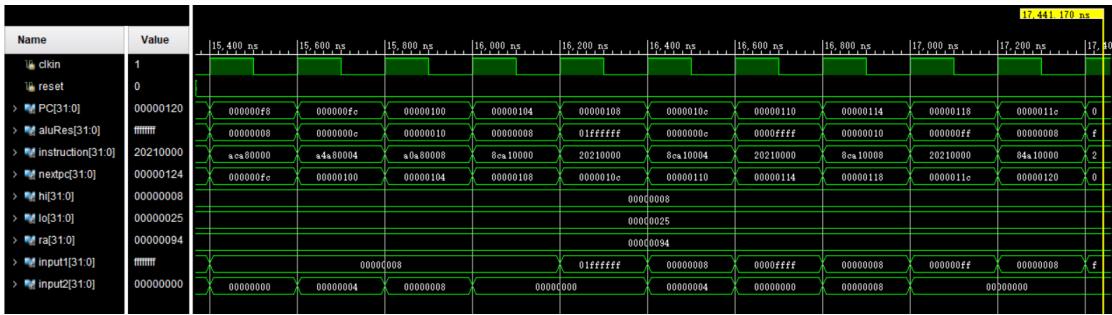
地址为0x000000ec, 指令编码为0x 714b0001, \$10=1, \$11=0xffffffff,  $0x00000001 \times 0xffffffff = 0x00000000ffffffff$ , 将运算结果与hi、lo相加, 结果hi的值为0x00000009, lo的值为0x00000044;

第76个时钟周期 (15000~15200ns) : msdu \$5, \$6

地址为0x000000f0, 指令编码为0x 70a60004, \$5=8, \$6=4,  $8 \times 4 = 32$ , hi、lo减去运算结果, 结果lo的值-32, hi=0x00000009, lo=0x00000024;

第77个时钟周期 (15200~15400ns) : msduu \$10, \$11

地址为0x000000f4, 指令编码为0x 714b0005, \$10=1, \$11=0xffffffff,  $0x00000001 \times 0xffffffff = 0x00000000ffffffff$ , hi、lo减去运算结果, 结果hi的值为0x00000008, lo的值为0x00000025;



分析：

第78个时钟周期 (15400~15600ns) : sw \$8, 0(\$5)

地址为0x000000f8, 指令编码为0x aca80000, \$8=0x01ffffff, \$5=8, 8+0=8, aluRes=8,

将0x01ffffff按字写入存储器地址为0x00000008的内存单元中；

第79个时钟周期 (15600~15800ns) : sh \$8, 4(\$5)

地址为0x000000fc, 指令编码为0x a4a80004, \$8=0x01ffffff, \$5=8, 8+4=12, aluRes=12,

将0x01ffffff按半字写入存储器地址为0x0000000c的内存单元中；

第80个时钟周期 (15800~16000ns) : sb \$8, 8(\$5)

地址为0x00000100, 指令编码为0x a0a80008, \$8=0x01ffffff, \$5=8, 8+8=16,

aluRes=16, 将0x01ffffff按字节写入存储器地址为0x00000010的内存单元中；

第81个时钟周期 (16000~16200ns) : lw \$1, 0(\$5)

地址为0x00000104, 指令编码为0x 8ca10000, \$5=8, 8+0=8, aluRes=8, 将存储器地

址为0x00000008的内存单元的值写入\$1中；

第82个时钟周期 (16200~16400ns) : addi \$1, \$1, 0

地址为0x00000108, 指令编码为0x 20210000, 该指令的ALU运算结果输出\$1的值, 目

的是为了检验前面的写操作是否成功实现, \$1=0x01ffffff;

第83个时钟周期 (16400~16600ns) : lw \$1, 4(\$5)

地址为0x0000010c, 指令编码为0x 8ca10004, \$5=8, 8+4=12, aluRes=12, 将存储器

地址为0x0000000c的内存单元的值写入\$1中；

第84个时钟周期 (16600~16800ns) : addi \$1, \$1, 0

地址为0x00000110，指令编码为0x 20210000，该指令的ALU运算结果输出\$1的值，目的是为了检验前面的写操作是否成功实现，\$1=0x0000ffff；

第85个时钟周期 (16800~17000ns) : lw \$1, 8(\$5)

地址为0x00000114，指令编码为0x 8ca10008，\$5=8，8+8=16，aluRes=16，将存储器地址为0x00000010的内存单元的值写入\$1中；

第86个时钟周期 (17000~17200ns) : addi \$1, \$1, 0

地址为0x00000118，指令编码为0x 20210000，该指令的ALU运算结果输出\$1的值，目的是为了检验前面的写操作是否成功实现，\$1=0x000000ff；

第87个时钟周期 (17200~17400ns) : lh \$1, 0(\$5)

地址为0x0000011c，指令编码为0x 84a10000，\$5=8，8+0=8，aluRes=8，将存储器地址为0x00000008的内存单元的低16位做符合拓展写入\$1中；

Name	Value	17.400 ns	17.500 ns	17.600 ns	17.700 ns	17.800 ns	17.900 ns	18.000 ns	18.100 ns	18.200 ns	18.300 ns	18.400 ns
clk	0											
reset	0											
PC[31:0]	00000104	00000120		00000124		00000128		0000012c		00000130		
aluRes[31:0]	00000008	ffffffff		00000008		0000ffff		00000008		ffffffff		
instruction[31:0]	8ca10000	20210000		94a10000		20210000		80a10000		20210000		
nextpc[31:0]	00000108	00000124		00000128		0000012c		00000130		00000134		
hi[31:0]	00000008					00000008						
lo[31:0]	00000025					00000025						
ra[31:0]	00000094					00000094						
input1[31:0]	00000008	ffffffff		00000008		0000ffff		00000008		ffffffff		
input2[31:0]	00000000					00000000						

第88个时钟周期 (17400~17600ns) : addi \$1, \$1, 0

地址为0x00000120，指令编码为0x 20210000，该指令的ALU运算结果输出\$1的值，目的是为了检验前面的写操作是否成功实现，\$1=0xffffffff；

第89个时钟周期 (17600~17800ns) : lhu \$1, 0(\$5)

地址为0x00000124，指令编码为0x 94a10000，\$5=8，8+0=8，aluRes=8，将存储器地

址为0x00000008的内存单元的低16位做0拓展写入\$1中；

第90个时钟周期 (17800~18000ns) : addi \$1, \$1, 0

地址为0x00000128，指令编码为0x 20210000，该指令的ALU运算结果输出\$1的值，目的是为了检验前面的写操作是否成功实现，\$1=0x0000ffff；

第91个时钟周期 (18000~18200ns) : lb \$1, 0(\$5)

地址为0x0000012c，指令编码为0x 80a10000，\$5=8，8+0=8，aluRes=8，将存储器地址为0x00000008的内存单元的低8位做符合拓展写入\$1中；

第92个时钟周期 (18200~18400ns) : addi \$1, \$1, 0

地址为0x00000130，指令编码为0x 20210000，该指令的ALU运算结果输出\$1的值，目的是为了检验前面的写操作是否成功实现，\$1=0xffffffff；

Name	Value	18,100 ns	18,200 ns	18,300 ns	18,400 ns	18,500 ns	18,600 ns	18,700 ns	18,800 ns	18,900 ns	19,000 ns
clk	0										
reset	0										
> PC[31:0]	00000104	0000012c	00000130	00000134		00000138	0000013c			00000140	
> aluRes[31:0]	00000008	00000008	ffffffff	00000008		000000ff	0000000f		00000000	00000000	
> instruction[31:0]	8ca10000	80a10000	20210000	90a10000		20210000	3981ffff			00000000	
> nextpc[31:0]	00000108	00000130	00000134	00000138		0000013c	00000140		00000144		
> hi[31:0]	00000008			00000008							
> lo[31:0]	00000025			00000025							
> ra[31:0]	00000094			00000094							
> input1[31:0]	00000008	00000008	ffffffff	00000008		000000ff	ffffffff		00000000	00000000	
> input2[31:0]	00000000			00000000					ffffffff	00000000	

第93个时钟周期 (18400~18600ns) : lbu \$1, 0(\$5)

地址为0x00000134，指令编码为0x 90a10000，\$5=8，8+0=8，aluRes=8，将存储器地址为0x00000008的内存单元的低8位做0拓展写入\$1中；

第94个时钟周期 (18600~18800ns) : addi \$1, \$1, 0

地址为0x00000138，指令编码为0x 20210000，该指令的ALU运算结果输出\$1的值，目的是为了检验前面的写操作是否成功实现，\$1=0x000000ff；

第95个时钟周期 (18800~19000ns) : xori \$1, \$11, -1

地址为0x0000013c，指令编码为0x 3981ffff，\$11=0xfffffff0，因此运算结果为0x0000000f，

```
aluRes=0x0000000f;
```

至此，61条指令逐条仿真验证完成，下面将工程烧录到Basys3板子上。

## 硬件实现

设计思路：

说明：指令执行采用单步（按键控制）执行方式，由开关（SW0~SW3）控制选择查看码管上的相关信息，地址和数据。地址或数据的输出经以下模块代码转换后接到数码管上。

增设按键up，默认状态下显示对应信息的低16位，按下up时显示高16位的信息，开关SW（SW0~SW3）状态情况如下：

SW=0000：显示当前PC的值

SW=0001：显示下条PC的值

SW=0010：显示ra的值

SW=0100：显示当前指令的值

SW=1000：显示rs和rt的地址

SW=1010：显示rs的值

SW=1001：显示ALU第二个操作数的值

SW=1100：显示ALU运算结果

SW=0011：显示lo的值

SW=0110：显示hi的值

1、实现CPU在板上运行需要两个时钟信号，CPU工作时钟和Basys3板系统时钟。CPU工作时钟即为按键，是CPU正常工作时钟信号；Basys3板系统时钟即为板提供的正常工作

时钟信号，即为100MHZ。Basys3板系统时钟信号引脚对应管脚W5。

2、每个按键周期，4个数码管都必须刷新一次。数码管位控信号AN3-AN0是1110、1101、1011、0111，为0时点亮该数码管，当然，还应该为数码管各位“1gfedcba”引脚输出信号，最高位为“1”。比如，“当前PC值”低8位中的高4位和低4位，必须经下页转换后送给数码管各引脚。

显示模块设计大概分为4个部分：

- (1) 对Basys3板系统时钟信号进行分频，分频的目的用于计数器；
- (2) 生成计数器，计数器用于产生4个数。这4数用于控制4个数码管；
- (3) 根据计数器产生的数生成数码管相应的位控信号（输出）和接收CPU来的相应数据；
- (4) 将从CPU接收到的相应数据转换为数码管显示信号，再送往数码管显示（输出）。

下面编写约束文件

根据顶层模块的各个端口需求连接到不同的管脚，顶层模块端口定义如下：

```
module top(
    input clk,           //clk for display
    input clkin,         //clkin for PC
    input reset,          //reset时寄存器堆、PC、存储器MEM都置初值
    input [3:0]SW,        //select the data
    input up,             //show upper bits
    output [6:0] seg,      //段码
    output [3:0] sm_wei   //哪个数码管
);
```

数码管显示器的时钟clk接系统时钟W5

```
set_property PACKAGE_PIN W5 [get_ports clk]
```

复位信号reset接开关SW15

```
set_property PACKAGE_PIN R2 [get_ports reset]
```

CPU工作时钟信号clkin接按键U18

```
set_property PACKAGE_PIN U18 [get_ports clkin]
```

高16位的显示信号up接按键T18

```
set_property PACKAGE_PIN T18 [get_ports up]
```

显示信息的选择开关SW接开关SW0~SW3

```
set_property PACKAGE_PIN V17 [get_ports {SW[0]}]
```

```
set_property PACKAGE_PIN V16 [get_ports {SW[1]}]
```

```
set_property PACKAGE_PIN W16 [get_ports {SW[2]}]
```

```
set_property PACKAGE_PIN W17 [get_ports {SW[3]}]
```

数码管显示器的段码seg、位码sm\_wei分别和Basys3版的内置特定的管脚相连

```
set_property PACKAGE_PIN U2 [get_ports {sm_wei[0]}]
```

```
set_property PACKAGE_PIN U4 [get_ports {sm_wei[1]}]
```

```
set_property PACKAGE_PIN V4 [get_ports {sm_wei[2]}]
```

```
set_property PACKAGE_PIN W4 [get_ports {sm_wei[3]}]
```

```
set_property PACKAGE_PIN W7 [get_ports {seg[0]}]
```

```
set_property PACKAGE_PIN W6 [get_ports {seg[1]}]
```

```
set_property PACKAGE_PIN U8 [get_ports {seg[2]}]
```

```
set_property PACKAGE_PIN V8 [get_ports {seg[3]}]
```

```
set_property PACKAGE_PIN U5 [get_ports {seg[4]}]
```

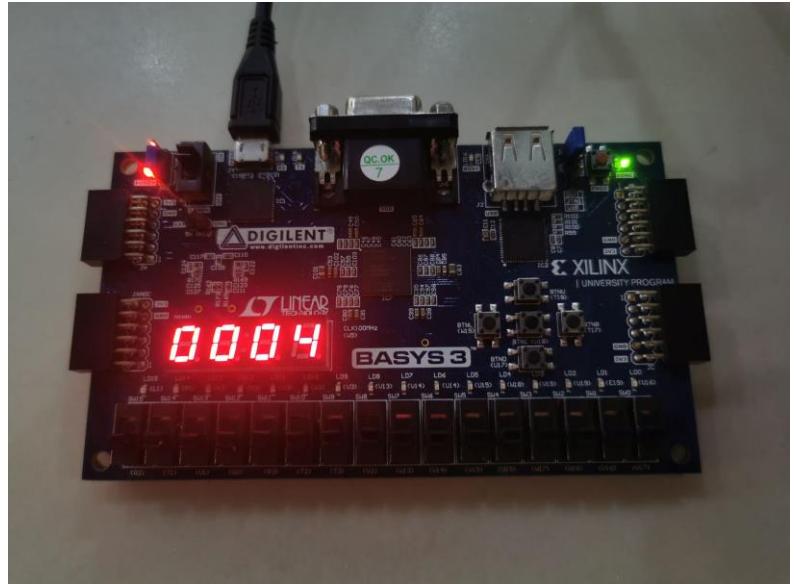
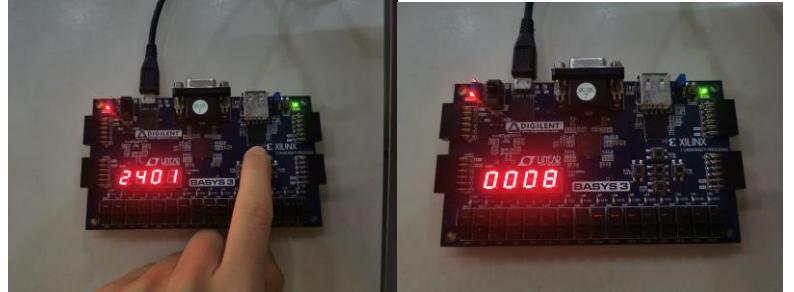
```
set_property PACKAGE_PIN V5 [get_ports {seg[5]}]
```

```
set_property PACKAGE_PIN U7 [get_ports {seg[6]}]
```

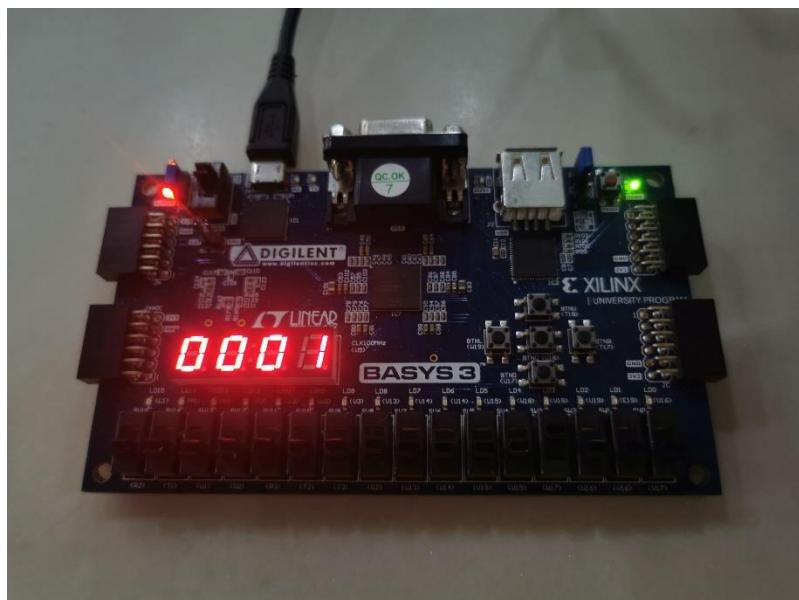
约束文件书写完毕，综合无误后生成比特流文件，将工程烧录到板子上。

为了防止报告过于冗长，在此特别说明前5条R型指令，分支指令bne，跳转指令jalr，读写指令lw，和乘除有关的指令mul、div、mtlo；

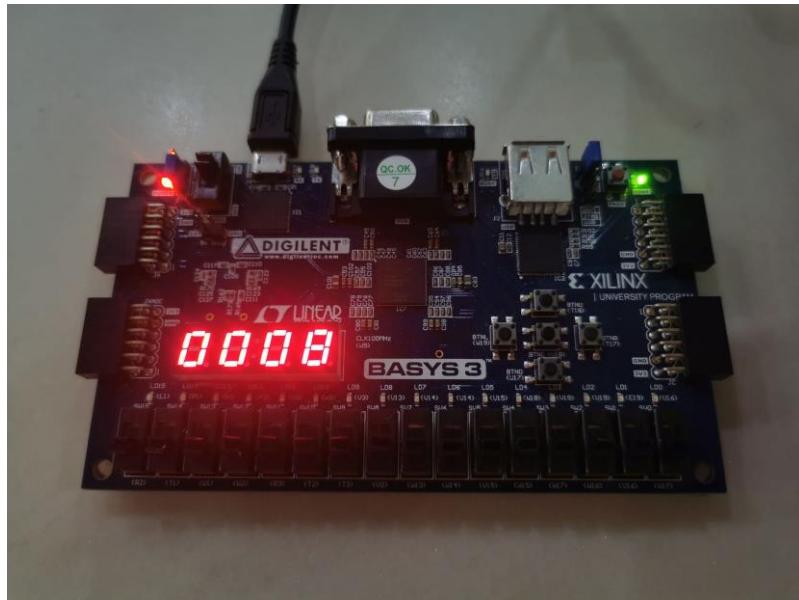
跳过第一条指令nop的说明，第二条指令为addiu \$1,\$0,8

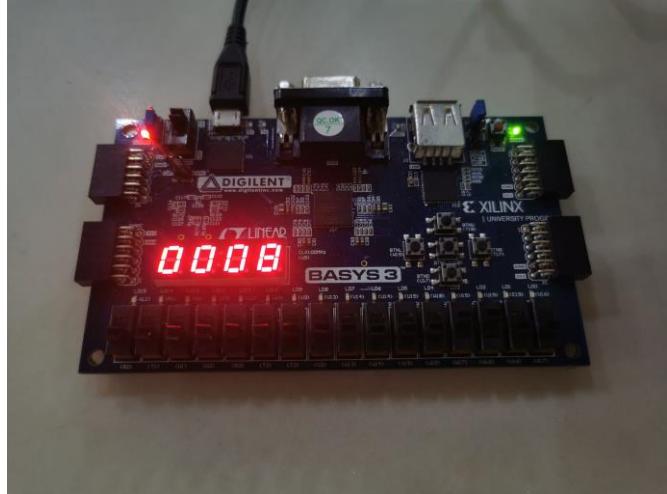
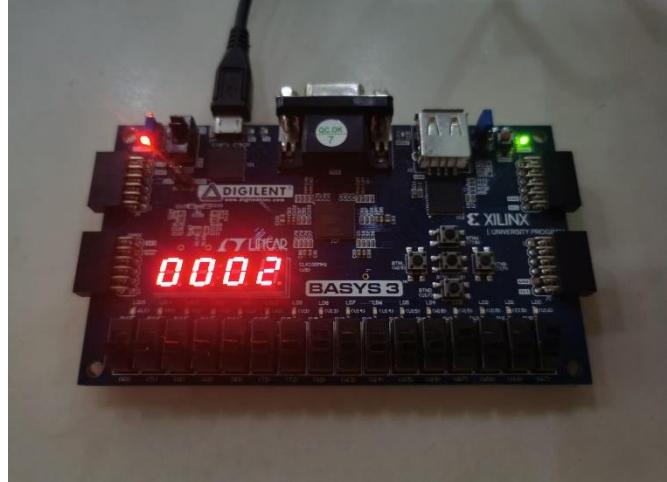
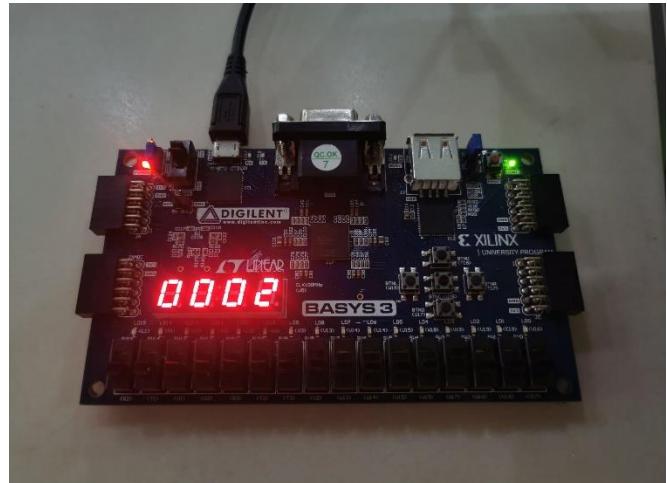
指令	addiu \$1,\$0,8
地址	
编码	

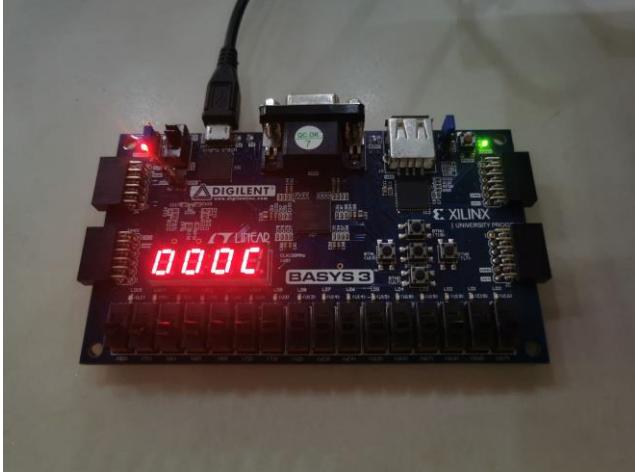
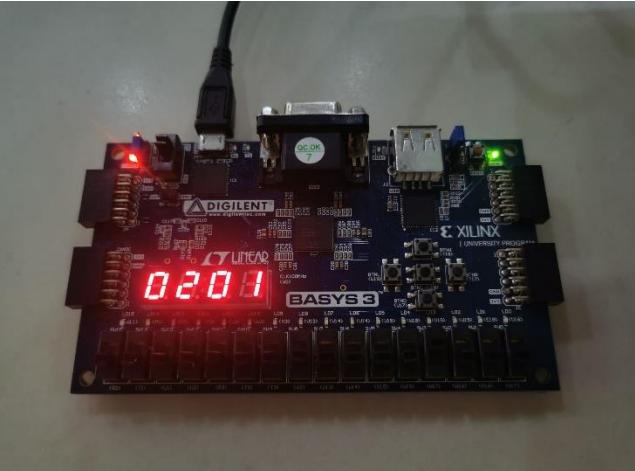
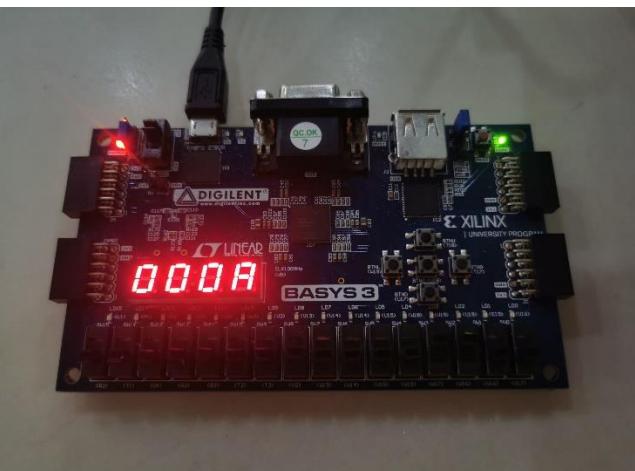
rs地址: rt地址

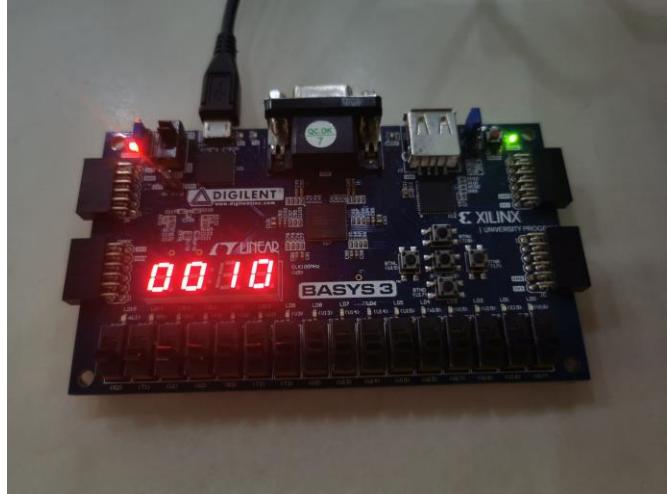
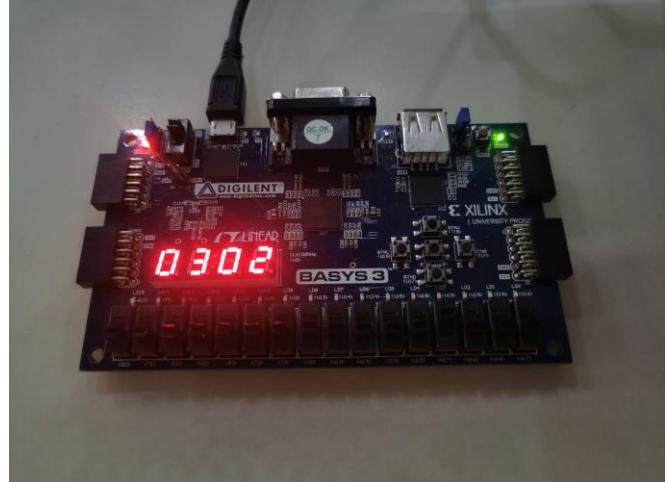
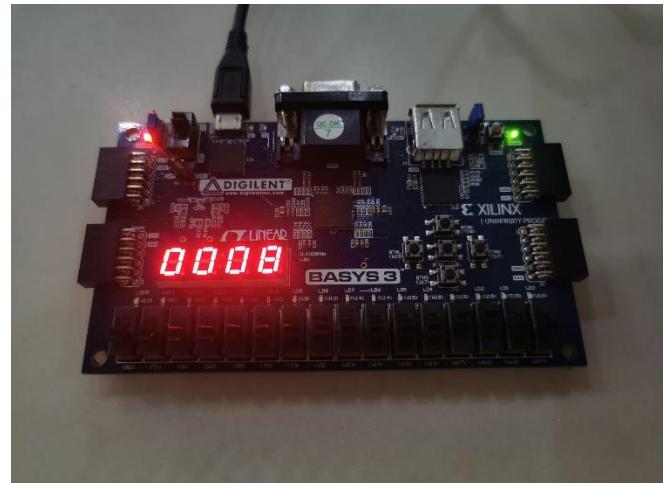


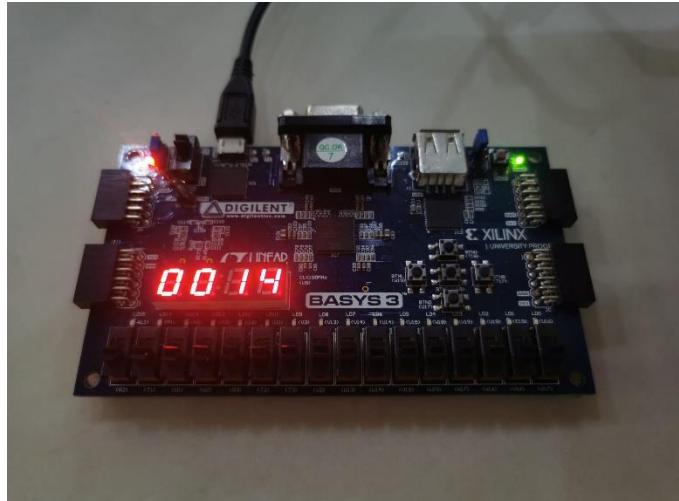
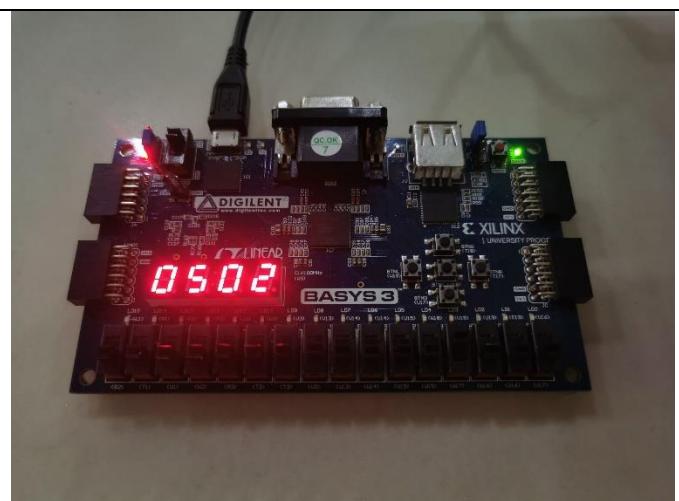
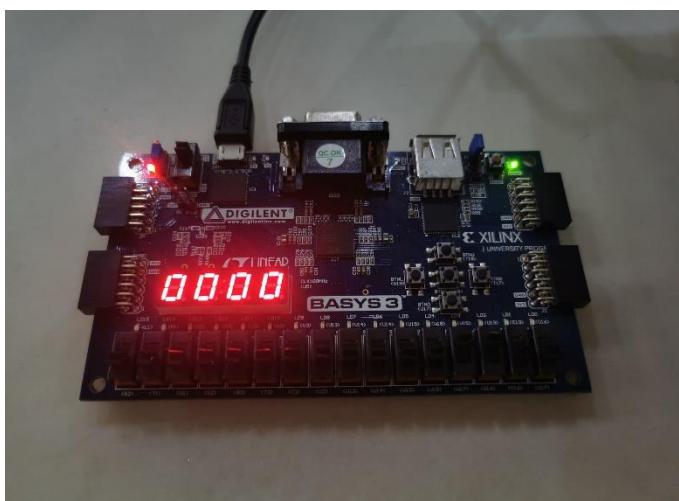
运算结果

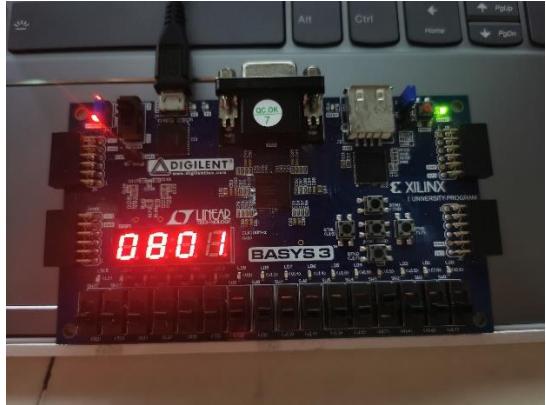
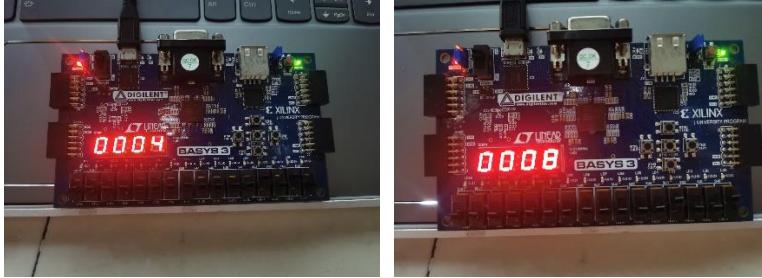
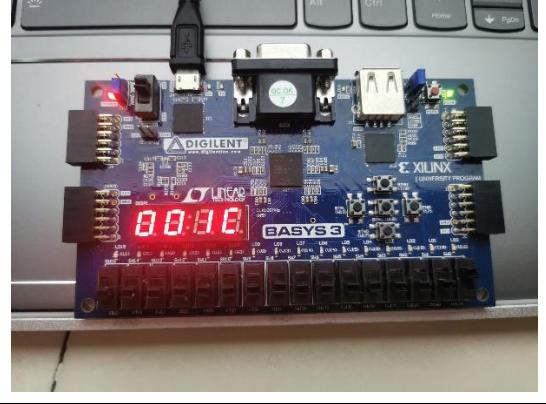


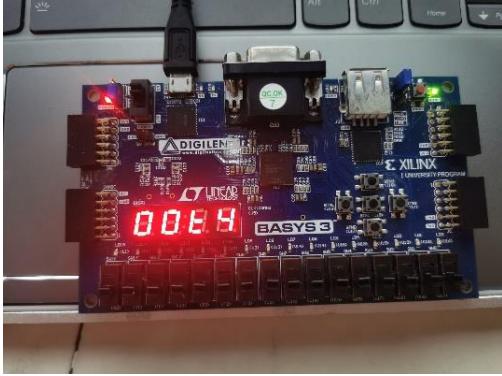
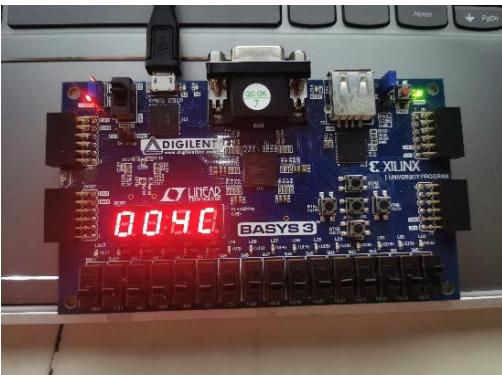
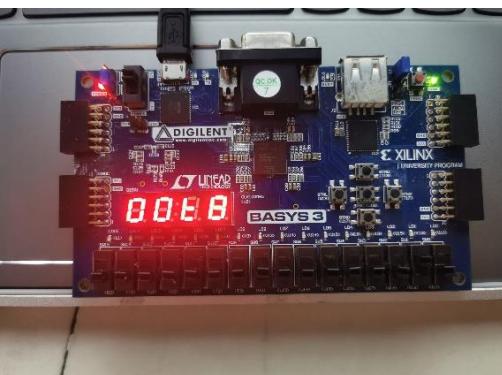
指令	ori \$2,\$0,2
地址	
rs地址: rt地址	
运算结果	

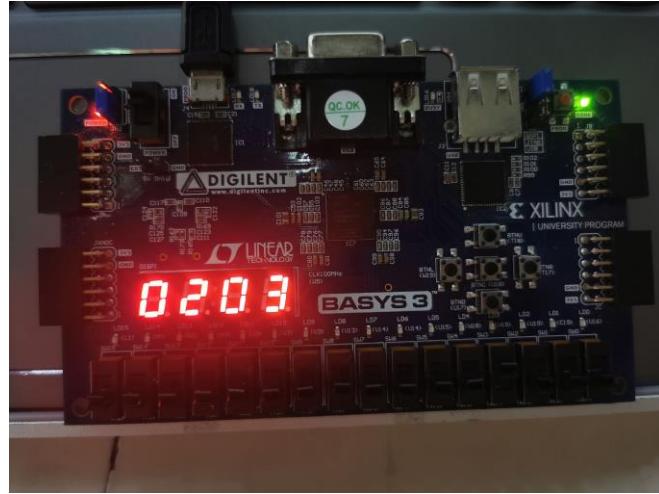
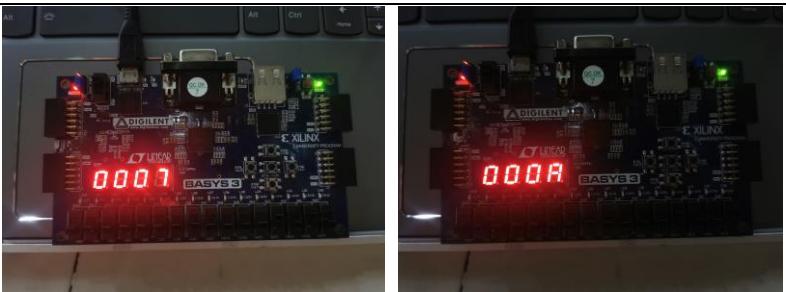
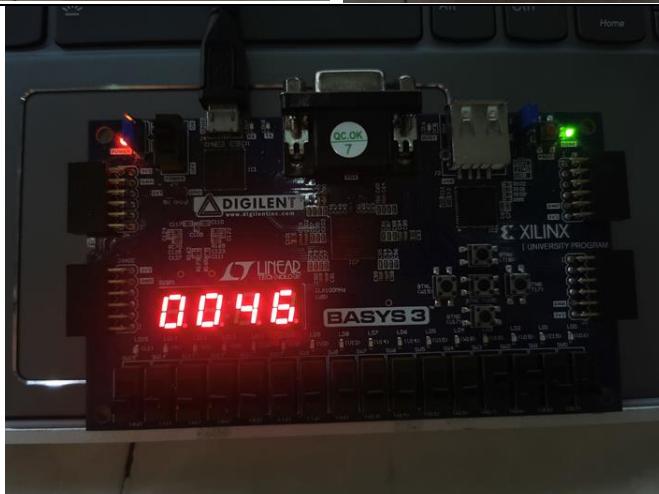
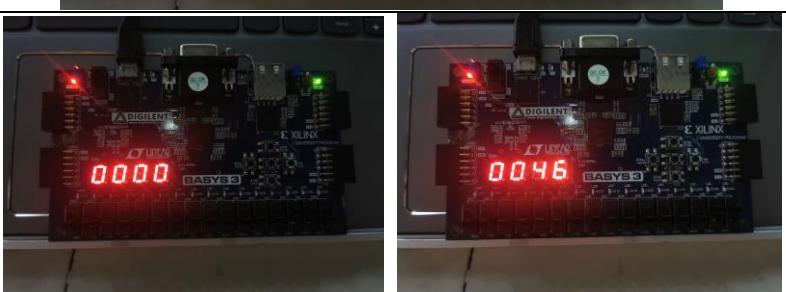
指令	add \$3,\$2,\$1
地址	
rs地址: rt地址	
运算结果	

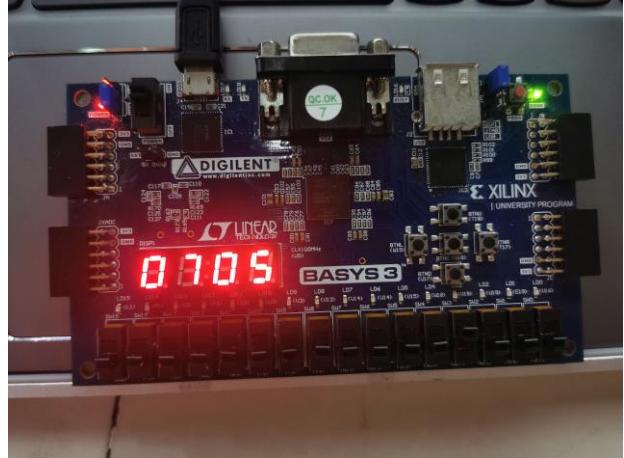
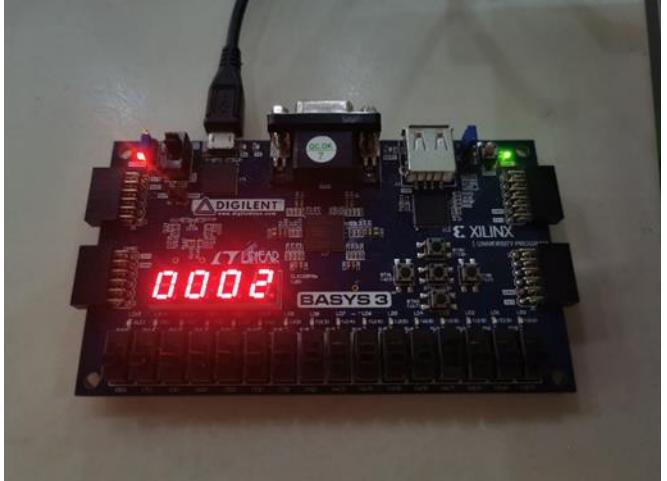
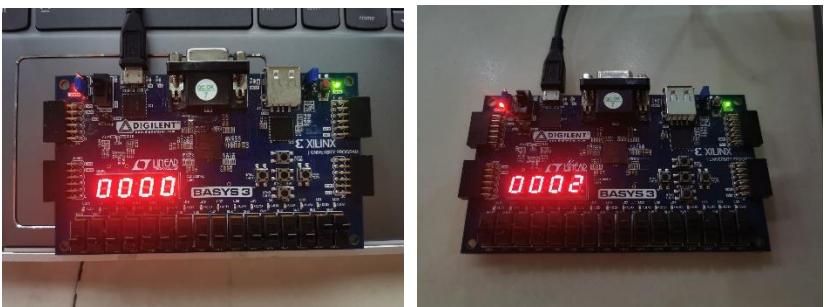
指令	sub \$5,\$3,\$2
地址	 A photograph of an Eyasys 3 development board. The board is black with various components and connectors. A red 7-segment display at the bottom left shows the binary value "0010". Other displays on the board show "LINEAR" and "EASYS 3". A red LED is illuminated on the left side.
rs地址: rt地址	 A photograph of the same Eyasys 3 development board. The red 7-segment display now shows the binary value "0302". The other displays and LEDs remain the same.
运算结果	 A photograph of the Eyasys 3 development board. The red 7-segment display now shows the binary value "0008". The other displays and LEDs remain the same.

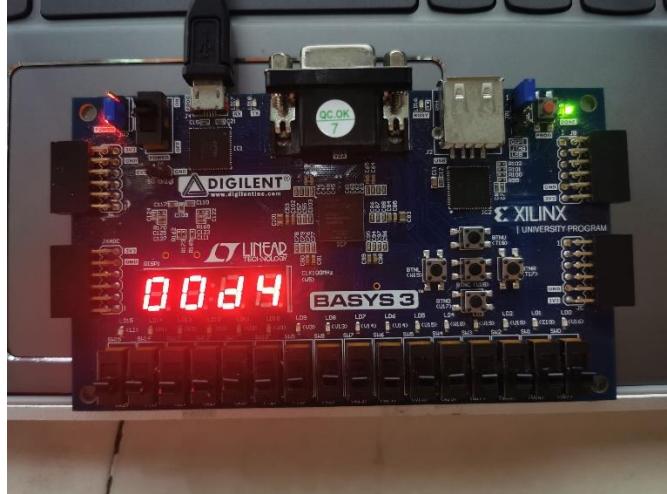
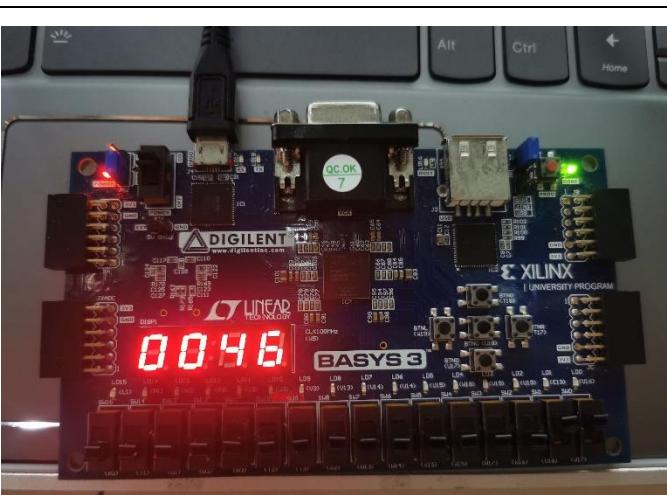
指令	and \$4,\$5,\$2
地址	 A photograph of an Eyasys 3 development board. The board is black with various components and connectors. A red 7-segment display in the center shows the binary value "0011". A red LED is illuminated above the display. A green power LED is visible on the right side.
rs地址: rt地址	 A photograph of the same Eyasys 3 development board. The red 7-segment display now shows the binary value "0502". The red LED and green power LED are also present.
运算结果	 A photograph of the Eyasys 3 development board. The red 7-segment display shows the binary value "0000". The red LED and green power LED are visible.

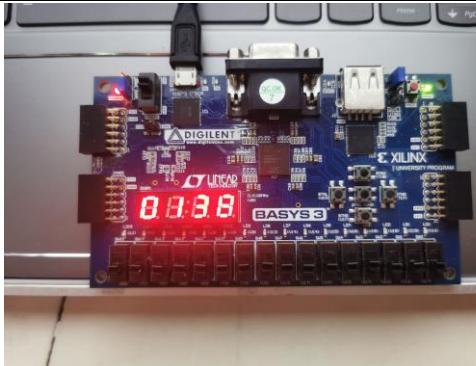
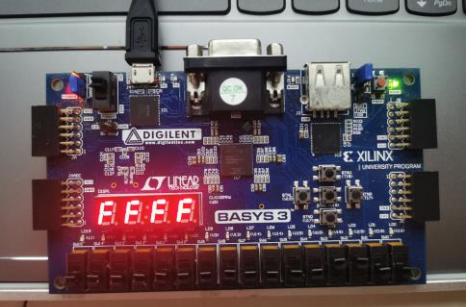
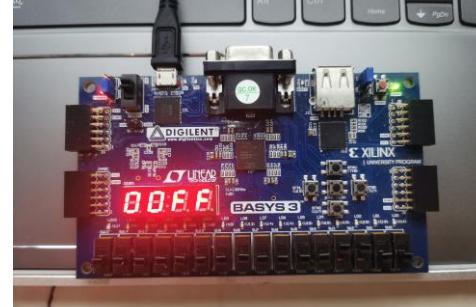
指令	bne \$8,\$1,-2
地址	
rs地址: rt地址	
rs数据: rt数据	
rs≠rt, 故发送跳转, 下条指令为 0x00000001c	

指令	jalr \$30
地址	
rs数据	
下一条指令地址	
ra被置为PC+4, 即0x000000e8	

指令	mul \$1, \$2, \$3
rs地址: rt地址	
rs数据: rt数据	
运算结果	
hi、lo	

指令	div \$7, \$5
rs地址: rt地址	
rs数据: rt数据	
运算结果	
hi、lo	

指令	mtlo \$1
地址	
rs数据	
运行完后lo的值	

指令	lb \$1, 0(\$5)后的addi \$1, \$1, 0	lbu \$1, 0(\$5)后的addi \$1, \$1, 0
地址		
运行结果		

CPU在板上运行的结果符合预期，实验结束。

## 六. 实验心得

在本次实验中，我学会了设计单周期CPU的方法，最后实验成功了，但过程是曲折的，我遇到了一些问题，但也有不少收获：

- 一开始没有对每个控制信号进行仿真，导致结果不符合预期，又找不出来错误在哪，后来单独对控制模块进行仿真，找到了问题所在。比如：jr指令在主控模块中不能识别，因为它是R型指令，regWrite=1，但不需要寄存器写操作，在ALU控制模块中才能识别它，因此添加nRW信号来阻止数据被错误写入寄存器堆）

- 把nextPC部分放在顶层模块虽然是可行的，但在调试时不利于观察，因此将该部分代码封装到一个模块里，易于调试和修改，这也教会了我模块化的设计思想，利于产品的调试和维护。
- nextPC模块中最开始的时候没有使用非阻塞赋值，导致PC一直出错。非阻塞赋值使得在结构块结束的时候才进行赋值，而不是立即赋值，后来使用非阻塞赋值后解决了这个错误。
- 若将assign用在过程块initial和always中，此时只支持reg类型，但是对于非过程块中连续赋值，assign支持net型。
- 原先不知道case、casez、casex的区别，在ALU控制模块中发现有很多信号识别不出来，根据老师给的资料理解和区别它们的功能，把case换成casex后成功识别出相应的信号。
- 学会了逻辑左移“<<”、逻辑右移“>>” 和算术左移“<<<”、算术右移“>>>”的区别，一开始不知道如何实现算术左移“<<<” 和算术右移“>>>”，通过查阅资料发现verilog HDL有直接算术左移“<<<” 和算术右移“>>>” 的实现方法和对应语句。
- 对模块仿真和测试的时候设计仿真文件的方法，如输入定义为reg类型、输出定义为wire类型，尽量穷举出所有可能的情况等等。

不足：

为了节约设计成本，将hi、lo合并到ALU模块中，但实际上在CPU的设计中hi、lo寄存器应该和ALU独立。这也导致了alu控制信号的位宽不够，需要进行拓展。

总而言之，此次实验使我受益匪浅。