

成绩：

实验二：单周期CPU设计与实现

一. 实验目的

1. 掌握单周期CPU的设计方法。
2. 通过实践制作CPU加深对CPU内部各个模块的原理、数据通路的构造、整体的工作原理的理解。
3. 通过在FPGA开发板实现CPU的设计，验证设计的正确性和可行性。

二. 实验内容

使用Verilog硬件描述语言，设计实现一个支持精简的MIPS指令集的单周期CPU，将其写入Basys3开发板。将一段MIPS汇编代码转化为机器代码后读入作为程序rom，通过Basys3开发板上的按键单步执行该程序，观察7段数码显示管的输出验证试验结果的正确性。

要求实现的指令包括：

- (1) **add rd , rs, rt**
- (2) **addi rt , rs ,immediate**
- (3) **sub rd , rs , rt**
- (4) **ori rt , rs ,immediate**
- (5) **and rd , rs , rt**
- (6) **or rd , rs , rt**
- (7) **sll rd, rt,sa**
- (8) **slt rd, rs, rt**
- (9) **sw rt ,immediate(rs)**
- (10) **lw rt , immediate(rs)**
- (11) **beq rs,rt,immediate**
- (12) **bne rs,rt,immediate**
- (13) **bgtz rs,immediate**
- (14) **j addr**
- (15) **halt**

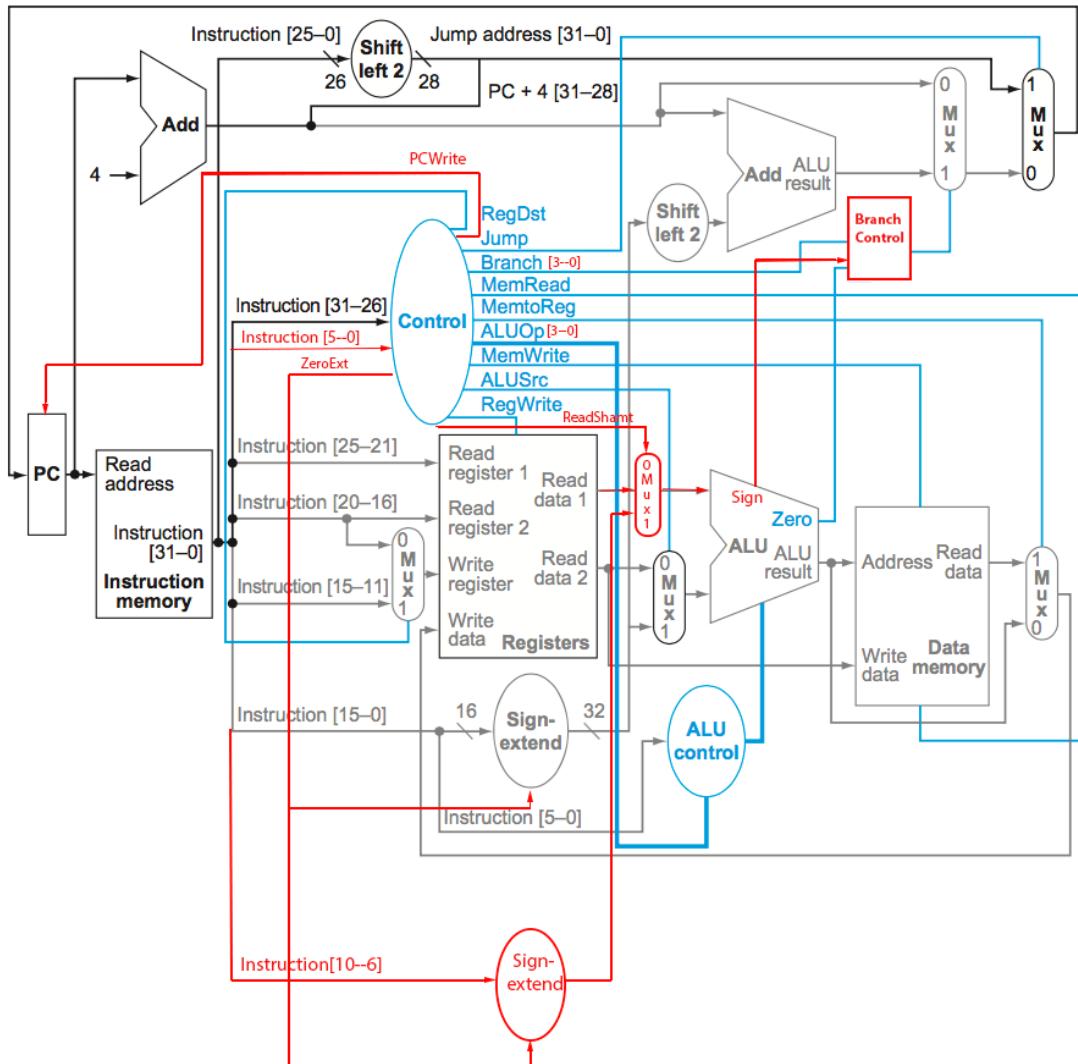
三. 实验原理

CPU本质执行的是一个取“指令->操作”循环，根据读取的指令值，发出控制信号，进行逻辑和算数运算，影响储存单元的内容和下一条指令的地址。设计CPU首先要设计操作和运算的规则，包括指令集、控制信号、ALU功能表等内容。这些表格将在实验过程部分给出。

之后，要设计具有特定功能，能够进行逻辑、算数运算和储存功能的PC、指令/数据存储器、控制单元、ALU等原件，并通过良好设计的数据通路整合为一个协同工作的系统。

本实验采用的数据通路是结合理论课课本的设计和我的改进和创新形成的。之所以没有采用老师给的数据通路设计，是因为在老师给出那个设计之前的几天我的空闲时间较多，又知道接下来会有设计CPU的作业，就按照课本上的CPU设计已经写好了顶层设计。之后拿到实验要求后为实现课本上没有要求的几条指令进行了改进。

数据通路图如下：图中蓝色和黑色部分是从理论课课本上截取下的未改动的部分。红色是我自己改动添加的部分。



本设计的创新之处在于将控制分支语句是否执行的模块（Branch Control）从Control Unit中分离出来。Branch Control输入Control单元根据指令类型发出Branch类型信号、ALU发出的Sign和Zero信号，根据这些决定是否执行分支语句。

这样的设计避免了ALU与Control Unit相互送数的情况，降低了系统的耦合性。同时保证CPU执行的IF, ID, EX, MEM这几个阶段数据基本从图中左侧原件向右侧原件移动，使得系统的工作流程更加清晰。Branch类型信号设计为4位，预留了扩展更多分支指令的空间。

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

(一)、设计思想

本次CPU设计的过程中，我严格采用了自顶向下、模块化的设计思想，我首先为每一个CPU的小模块编写一个Verilog文件，声明接口，但不写具体的实现。然后在顶层文件中进行连接，通过vivado的线路图功能和仔细阅读warning信息进行检查。发现错误进行修改，确认顶层设计无误的情况下再编写每一个模块的代码。

(二)、设计方法

具体步骤如下：

1. 设计指令集，控制信号，ALU功能表
2. 设计数据通路
3. 在Vivado中写各个模块的接口设计，并编写顶层文件连接各个端口
4. 具体实现各个模块
5. 编写仿真文件进行仿真
6. 编写输入输出模块，进行测试，测试行为正确后与CPU顶层文件接口连接
7. 烧写到Basys3开发板，验证结果

(三)、指令集与控制信号表

1. 指令集

本CPU支持R型，I型，J型三种指令，指令格式如下：

BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct	
	31 26 25	21 20	16 15	11 10	6 5		0
I	opcode	rs	rt		immediate		
	31 26 25	21 20	16 15				0
J	opcode			address			
	31 26 25						0

指令集如下：共包含15条指令，其中6条R型指令、7条I型指令、2条J型指令。表格描述了它们的名称、类型、OPCODE（指令高6位），FUNCT（指令低6位）以及它们的功能。除halt指令外该表格中OPCODE和FUNCT部分完全依据真实的MIPS指令集编写。

指令名	类型	OPCODE	FUNCT	功能
add	R	000000	100000	$rd = rs + rt$
sub	R	000000	100010	$rd = rs - rt$
and	R	000000	100100	$rd = rs \& rt$
or	R	000000	100101	$rd = rs rt$
sll	R	000000	000000	$rd = rt << (\text{zero-extend}) shamt$
slt	R	000000	101010	$rd = rs < rt ? 1 : 0$ (signed comp)
beq	I	000100		if(rs=rt) pc \leftarrow pc + 4 + (sign-extend)immediate <<2 else pc \leftarrow pc + 4
bne	I	000101		if(rs!=rt) pc \leftarrow pc + 4 + (sign-extend)immediate <<2 else pc \leftarrow pc + 4
bgtz	I	000111		if(rs>0) pc \leftarrow pc + 4 + (sign-extend)immediate <<2 else pc \leftarrow pc + 4
addi	I	001000		$rt \leftarrow rs + (\text{sign-extend})\text{immediate}$
ori	I	001101		$rt \leftarrow rs (\text{zero-extend})\text{immediate}$
lw	I	100011		$rt \leftarrow \text{memory}[rs + (\text{sign-extend})\text{immediate}]$
sw	I	101011		$\text{memory}[rs + (\text{sign-extend})\text{immediate}] \leftarrow rt$
j	J	000010		jump to address {PC[31:28], Instruction[25:0], 2b'00}
halt	J	111111		PC no more work

2. Control Unit信号

控制信号共有12个，它们的含义如下：

控制信号名	状态“0”	状态“1”
RegDst	写寄存器组寄存器的地址，来自rt字段，相关指令：addi、ori、lw	写寄存器组寄存器的地址，来自rd字段，相关指令：add、sub、and、or、slt、sll
ALUSrc	无需读取常数的其他指令	来自sign或zero扩展的立即数，相关指令：addi、ori、sw、lw
MemtoReg	数据存储器读出的值不写寄存器	数据存储器读出的值送寄存器相关指令：lw
RegWrite	无写寄存器组寄存器，相关指令：beq、bne、bgtz、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、slt、sll、lw
MemRead	输出高阻态	读数据存储器，相关指令：lw
MemWrite	输出高阻态	读数据存储器，相关指令：lw
Jump	不执行跳转	执行跳转
PCWrite	PC不更改，相关指令：halt	PC更改，相关指令：除指令halt外
ZeroExt	执行符号扩展（其他指令）	(zero-extend)immediate (0扩展)，相关指令：ori, sll
ReadShamt	其他指令，ALU输入1读取寄存器文件输出1	ALU输入1读取R型指令中，0扩展后的指令[10:6]位移量数据。相关指令：sll

续表

控制信号名	
Branch[3-0]	标识三种分支语句：0001：beq, 0010：bne, 0011：bgtz
ALUOp[3-0]	ALU8种运算功能选择,见下文功能表

这些信号实际控制各个原件的工作，下表完整描述了各个指令决定的Control Unit信号。

指令名	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch
add	1	0	0	1	0	0	0000
sub	1	0	0	1	0	0	0000
and	1	0	0	1	0	0	0000
or	1	0	0	1	0	0	0000
sll	1	0	0	1	0	0	0000
slt	1	0	0	1	0	0	0000
beq	0	0	0	0	0	0	0001
bne	0	0	0	0	0	0	0010
bgtz	0	0	0	0	0	0	0011
addi	0	1	0	1	0	0	0000
ori	0	1	0	1	0	0	0000
lw	0	1	1	1	1	0	0000
sw	0	1	0	0	0	1	0000
j	0	0	0	0	0	0	0000
halt	0	0	0	0	0	0	0000

续表

指令名	ALUOp	Jump	PCWrite	ZeroExt	ReadShamt
add	0010	0	1	0	0
sub	0010	0	1	0	0
and	0010	0	1	0	0
or	0010	0	1	0	0
sll	0010	0	1	1	1
slt	0010	0	1	0	0
beq	0001	0	1	0	0
bne	0001	0	1	0	0
bgtz	0011	0	1	0	0
addi	0100	0	1	0	0
ori	0101	0	1	1	0
lw	0000	0	1	0	0
sw	0000	0	1	0	0
j	0000	1	1	0	0
halt	0000	0	0	0	0

3.AUL功能表

本设计中ALU共支持以下8中功能：分别是与、或、加、减、无符号比较、有符号比较、左移、异或。

ALU Control	功能
0000	AND
0001	OR
0010	add
0110	sub
0111	slt_u
1000	slt_sign
1001	shift left
1100	NOR

ALU执行哪种功能是由ALUOp 和 FUNCT决定的（算数和移位运算的ALUOp均为0010）

ALUOp	FUNCT	ALU Control
0000	No(sw/lw)	0010
0001	No(beq/bne)	0110
0010	100000(add)	0010
0010	100010(sub)	0110
0010	100100(and)	0000
0010	100101(or)	0001
0010	101010(slt)	1000
0010	000000(sll)	1001
0011	No(bgtz)	1000
0100	No(addi)	0010
0101	No(ori)	0001

(四)、设计数据通路

数据通路的设计在实验原理部分已经提到。通过这种设计，各种指令均可正确执行，下面将说明它们的执行过程：

1. R型指令：

CPU读取PC处的指令，寄存器收到地址中的rs和rt地址，送出两个寄存器的值。

Control Unit根据FUNCT值发ReadShamt（仅当sll指令时为1，且此时ZeroExt也为1），决定ALU的输入1是rs寄存器的值还是0扩展后的位移量(shamt)。

Control Unit发ALUSrc = 0，决定ALU的输入2是rt寄存器的值

Control Unit发ALUOp，ALU Control根据ALUOp和FUNCT决定ALU Control信号值，决定ALU进行的运算

ALU将运算结果送到右下角Mux，Control Unit发MemtoReg = 0，使得ALU运算结果送到Regfile的Write data

寄存器Control Unit发RegDst = 1，选择ALU运算结果要写入rt寄存器。

Branch Control发信号0，Control Unit发Jump信号0，因此PC+4写入PC。

2. I型指令

CPU读取PC处的指令，寄存器收到地址中的rs和rt地址，送出两个寄存器的值。

Control Unit根据OPCODE判断指令类型

若是跳转类指令，Control Unit发对应的Branch信号，与此同时发对应的ALUOp，选择相应的ALU计算功能（beq和bne指令使用减法功能，bgtz指令使用有符号比较功能）。Control发ALUSrc = 0，ReadShamt = 0，故ALU的输入来自rs和rt。rs和rt的值经过相应的计算后，ALU根据计算结果发出Zero（结果是否为0）。Sign信号直接给出rs最高位的值。Branch Control根据Branch信号，Sign信号和Zero信号决定是否跳转。决定的代码如下：

```
4'b0001: BranchControl_Output_Exec_Branch = Zero_in; //beq  
4'b0010: BranchControl_Output_Exec_Branch = ~Zero_in; //bne  
4'b0011: BranchControl_Output_Exec_Branch = (~Sign_in && ~Zero_in); //bgtz
```

若决定跳转，发Exec_Branch = 1，跳转的地址 $pc + 4 + (\text{sign-extend})\text{immediate}$ 写入PC。

若是addi指令，ALUSrc为1，符号扩展后的指令低16位作为ALU输入2。rs作为ALU输入1。ALU执行加法功能。最后结果送回rt寄存器。PC+4写入PC。

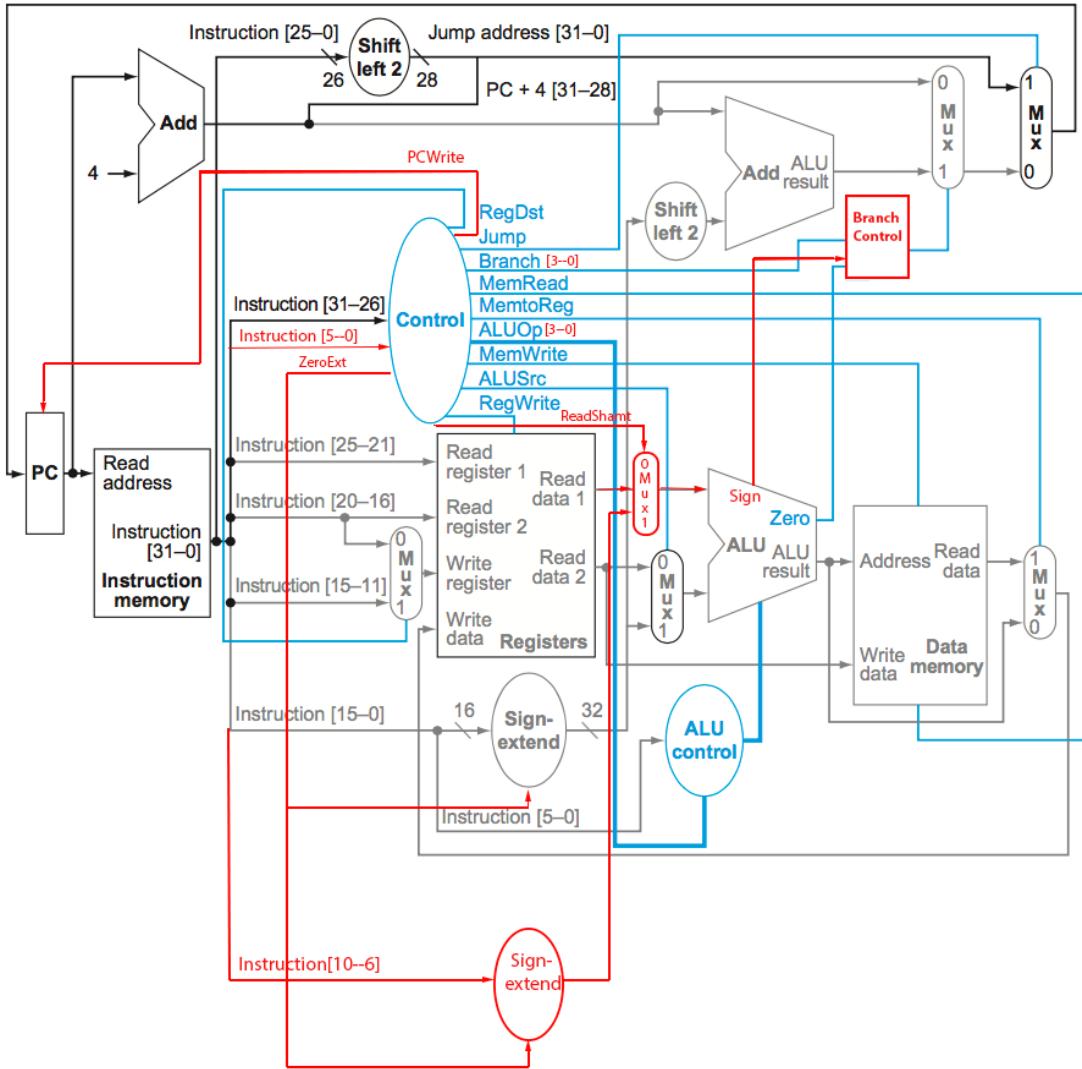
ori指令与addi指令除ALU执行的是或功能的区别外，Control Unit会发ZeroExt = 1，进行0扩展。

若是lw指令，CPU读取rs和符号扩展后的指令低16位作为数据储存器地址，Control Unit发MemtoReg = 1，使得数据寄存器送出该地址的值，到Write data。Control Unit发RegDst = 0，以rt为写入目标寄存器。PC+4写入PC。

若是sw指令，CPU同样读取rs和符号扩展后的指令低16位作为要写入写入数据存储器地址。Control Unit发MemWrite = 1，使得Regfile的Read data2，即rt的值，写入指定的数据存储器地址。PC+4写入PC。

3. J型指令

CPU读取PC处的指令，若发现是J指令，Control Unit发Jump信号，PC写入{PC[31:28], Instruction[25:0], 2b'00}。若是Halt指令，Control Unit发PCWrite = 1，PC停止增加，CPU停机。



(五)、设计模块接口和顶层文件，然后实现每个模块

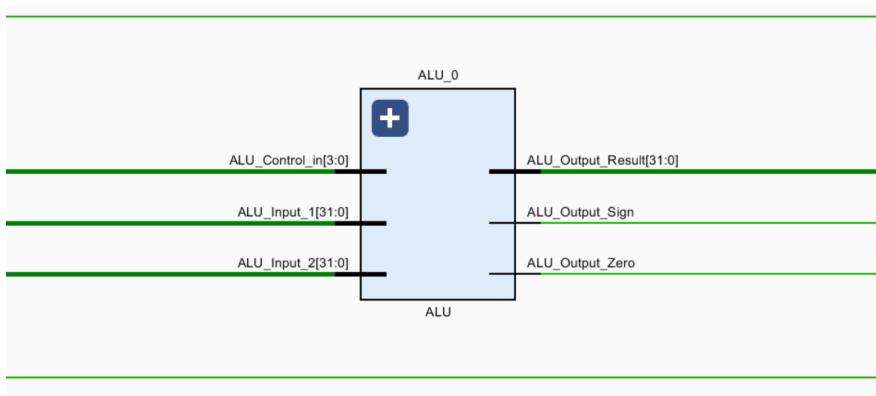
1. ALU

ALU Control信号决定ALU功能，ALU Input 1 和ALU Input 2 是两个操作数的值。ALU_Output_Result是计算结果。ALU_Output_Sign输出操作数1最高位的值，ALU_Output_Zero运算操作结果==0，这两者用于Branch Control判断是否执行跳转指令。功能相关代码如下：

```

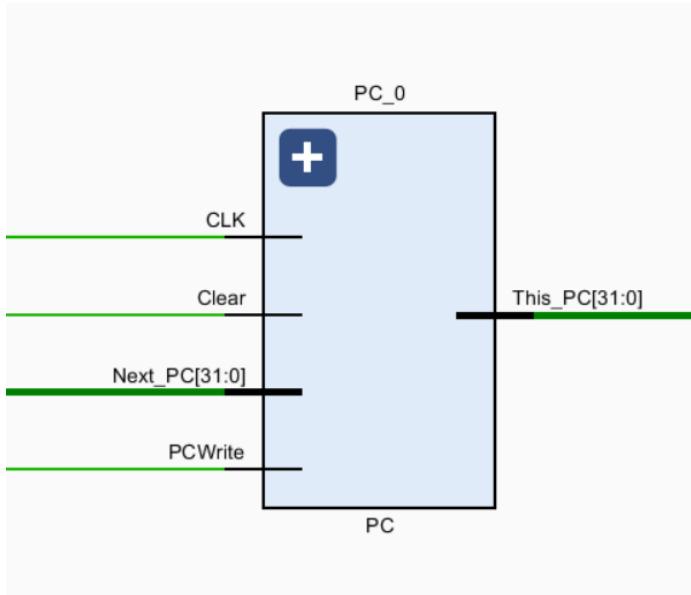
assign ALU_Output_Zero = ALU_Input_1 == ALU_Input_2 ? 1 : 0; //输出运算结果是否为0
assign ALU_Output_Sign = ALU_Input_1[31] ? 1 : 0; //输出操作数1是否为负数(用于bgtz)
always @(ALU_Control_in or ALU_Input_1 or ALU_Input_2)
begin
    case(ALU_Control_in)
        4'b0000: ALU_Output_Result = ALU_Input_1 & ALU_Input_2; //And功能
        4'b0001: ALU_Output_Result = ALU_Input_1 | ALU_Input_2; //Or功能
        4'b0010: ALU_Output_Result = ALU_Input_1 + ALU_Input_2; //
        4'b0110: ALU_Output_Result = ALU_Input_1 - ALU_Input_2; //
        4'b0111: ALU_Output_Result = ALU_Input_1 < ALU_Input_2 ? 1 : 0; //
        4'b1000: //有符号比较小于
begin
    if(ALU_Input_1[31] == 1 && ALU_Input_2[31] == 0)
        ALU_Output_Result = 1; // 前者为负后者为正
    else if ((ALU_Input_1 < ALU_Input_2) && (ALU_Input_1[31] ==
ALU_Input_2[31]))
        ALU_Output_Result = 1; //符号相同, 直接比较, 前者小于后者
    else
        ALU_Output_Result = 0; //前者大于后者
end
4'b1001: ALU_Output_Result = ALU_Input_2 << ALU_Input_1; //左移运算
4'b1100: ALU_Output_Result = ALU_Input_1 ^ ALU_Input_2; //异或运算

```



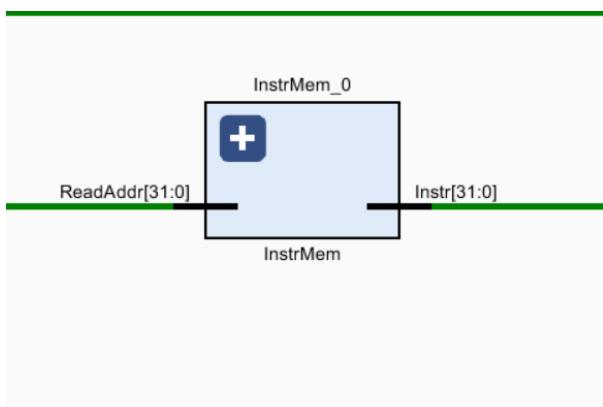
2.PC

CLK是时钟信号，时钟信号上升沿到来且PCWrite = 1时更新This_PC = Next_PC。Clear是清零信号，高电平有效。



3. Instruction Memory

根据PC指出的指令地址，读出指令



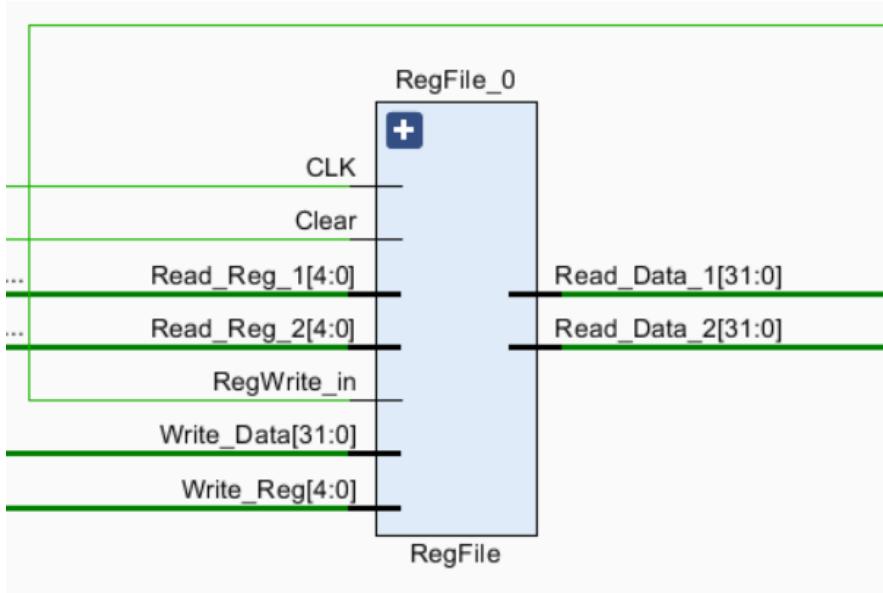
4. Register File

CLK是时钟信号，在下降沿触发尝试写寄存器的操作。Clear的上升沿和下降沿会触发写寄存器操作，这是为了在清零后的PC = 0，CPU执行第一条指令，但却没有CLK信号到来时，正确地送数给寄存器。

```

always @(negedge CLK or negedge Clear)
begin
    if(Clear)
        begin
            for(i = 1; i < 32; i = i + 1) impl_reg[i] <= 0;
            if(RegWrite_in == 1 && Write_Reg != 0)
                impl_reg[Write_Reg] <= Write_Data;
        end
    else if(RegWrite_in == 1 && Write_Reg != 0)
        impl_reg[Write_Reg] <= Write_Data;
end

```



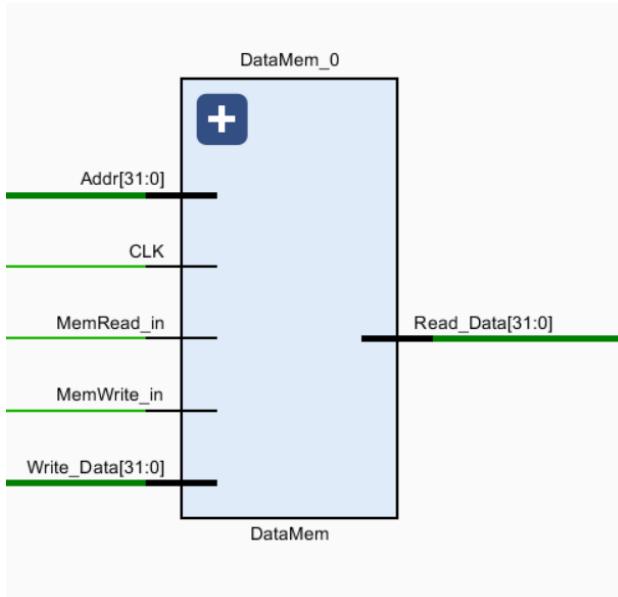
5.Data Memory

Addr是数据存储器要进行读取或写入操作的数据单元地址。CLK是时钟信号，当时钟下降沿到来时，如果 MemWrite = 1，将Write_Data写入指定地址。若MemRead = 1 输出Addr地址的值，否则输出高阻态。

```

reg[ 7:0] ram[ 127:0];
assign Read_Data[ 7:0]      = (MemRead_in==1)?ram[Addr + 3] :8'bz;
assign Read_Data[ 15:8]     = (MemRead_in==1)?ram[Addr + 2] :8'bz;
assign Read_Data[ 23:16]    = (MemRead_in==1)?ram[Addr + 1] :8'bz;
assign Read_Data[ 31:24]    = (MemRead_in==1)?ram[Addr ]      :8'bz;
always @ (negedge CLK)
begin
  if(MemWrite_in)
  begin
    ram[Addr]      <= Write_Data[ 31:24];
    ram[Addr+1]    <= Write_Data[ 23:16];
    ram[Addr+2]    <= Write_Data[ 15:8];
    ram[Addr+3]    <= Write_Data[ 7:0];
  end
end

```



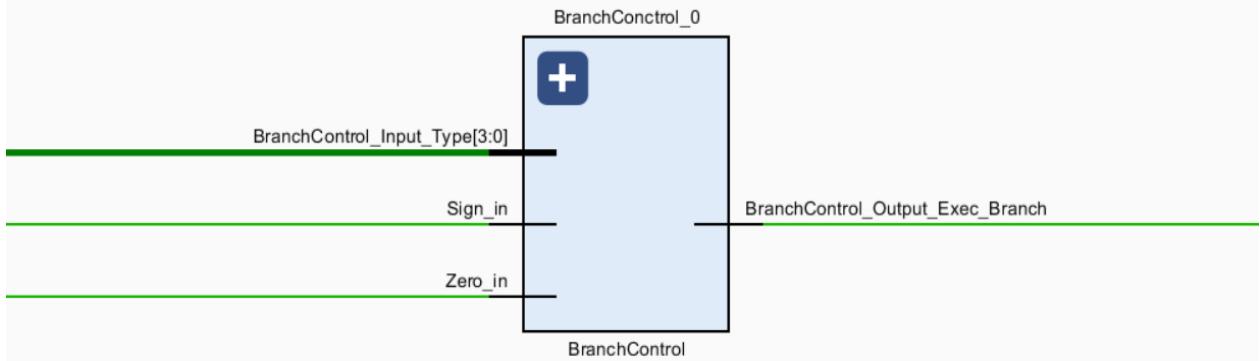
6. Branch Control

Branch Control输入Control单元根据指令类型发出Branch类型信号、ALU发出的Sign和Zero信号，根据这些决定是否执行分支语句。

```

always@(BranchControl_Input_Type or Sign_in or Zero_in)
begin
    case (BranchControl_Input_Type)
        4'b0000: BranchControl_Output_Exec_Branch = 1'b0; //非分支语句
        4'b0001: BranchControl_Output_Exec_Branch = Zero_in; //beq语句
        4'b0010: BranchControl_Output_Exec_Branch = ~Zero_in; //bne语句
        4'b0011:
            begin
                BranchControl_Output_Exec_Branch = (~Sign_in && ~Zero_in); //bget语句
            end
        default:
            begin
                BranchControl_Output_Exec_Branch = 1'b0;
            end
    endcase
end

```



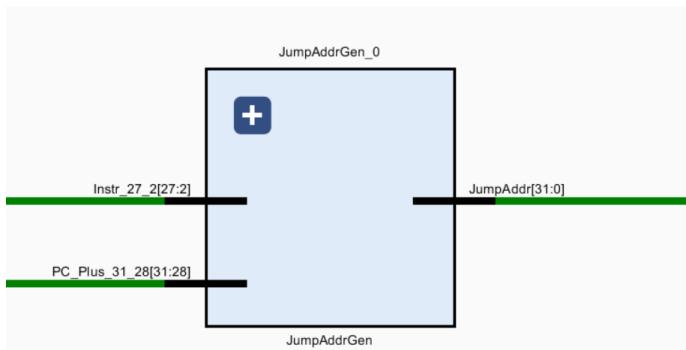
7. Jump Address Generator

生成Jump跳转地址。

```

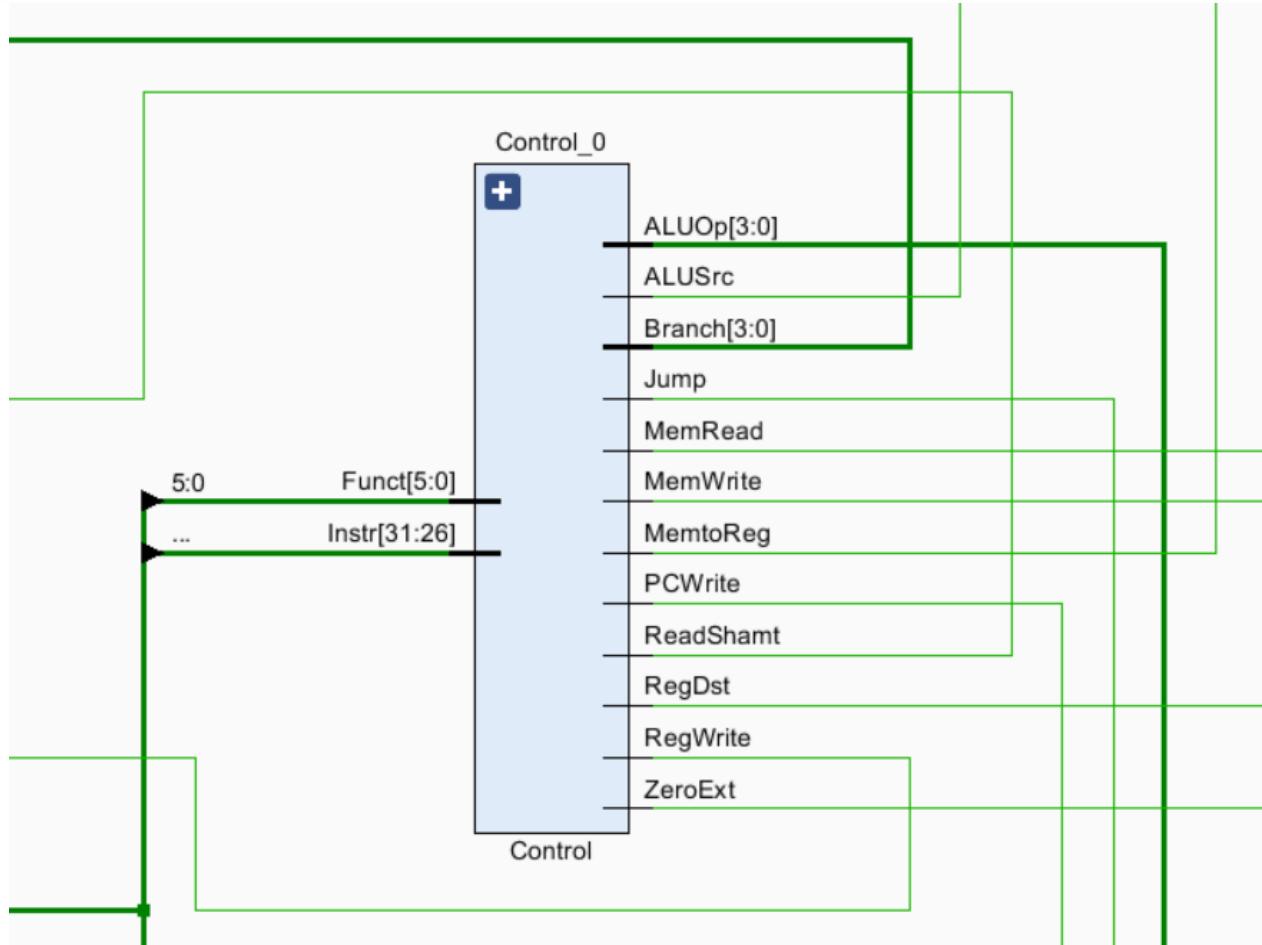
module JumpAddrGen(
    input [27:2] Instr_27_2,
    input [31:28] PC_Plus_31_28,
    output [31:0] JumpAddr
);
    assign JumpAddr = {PC_Plus_31_28, Instr_27_2, 2'b00};
endmodule

```



8. Control Unit

Control Unit的功能，各个输入输出信号的意义，见上文“（三）2.Control Unit信号”段落的描述。



9. Add, Number Extend, Mux

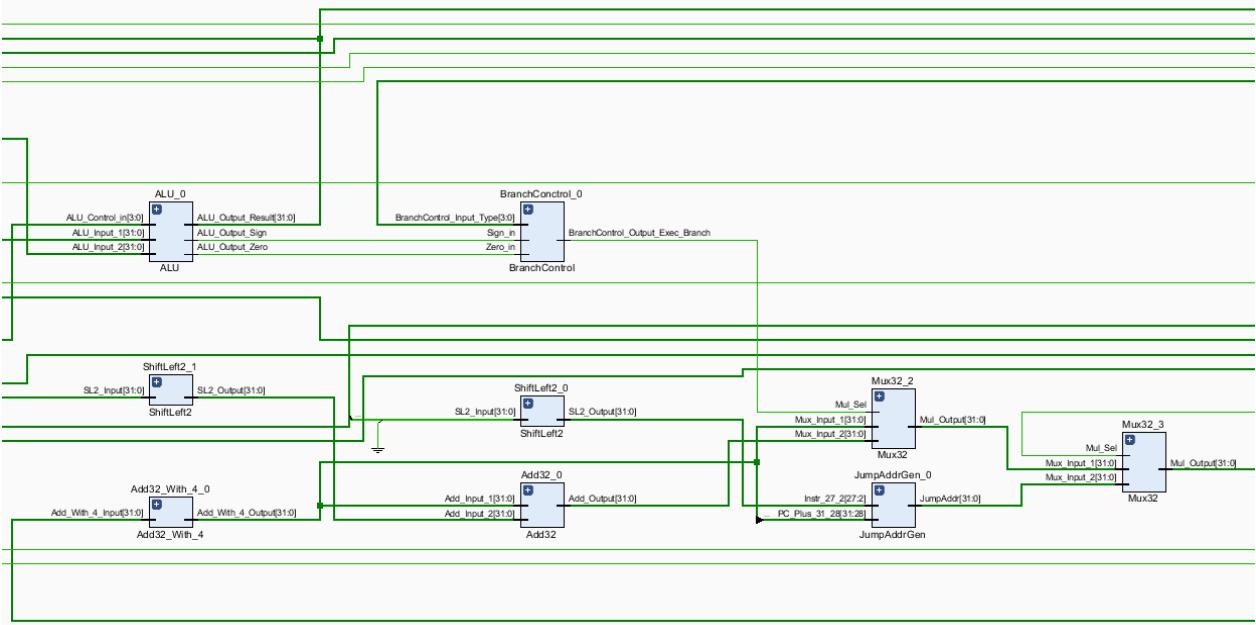
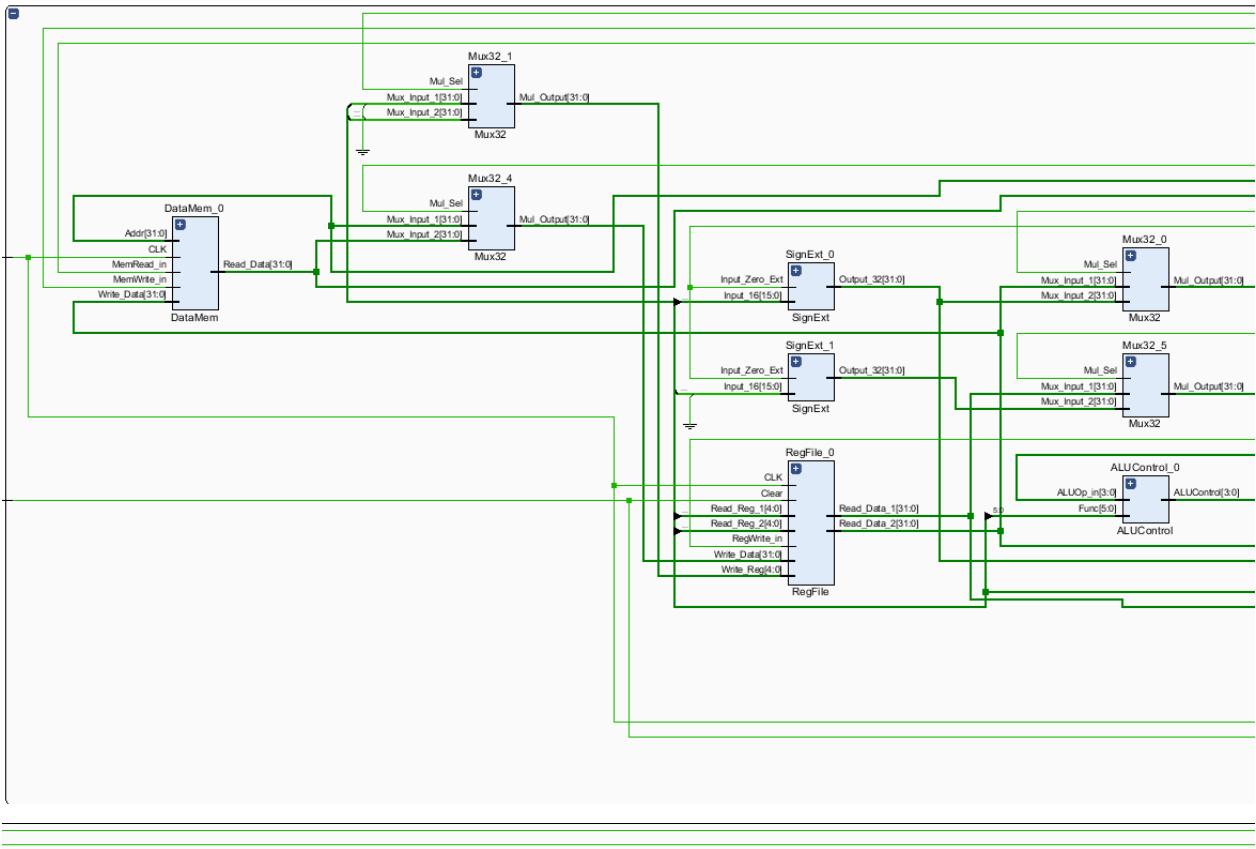
系统还用到了多个加法器（这里直接指定+4用于生成PC+4），数位扩展，多路复用器模块，这些模块较为基本，这里直接展示它们的代码

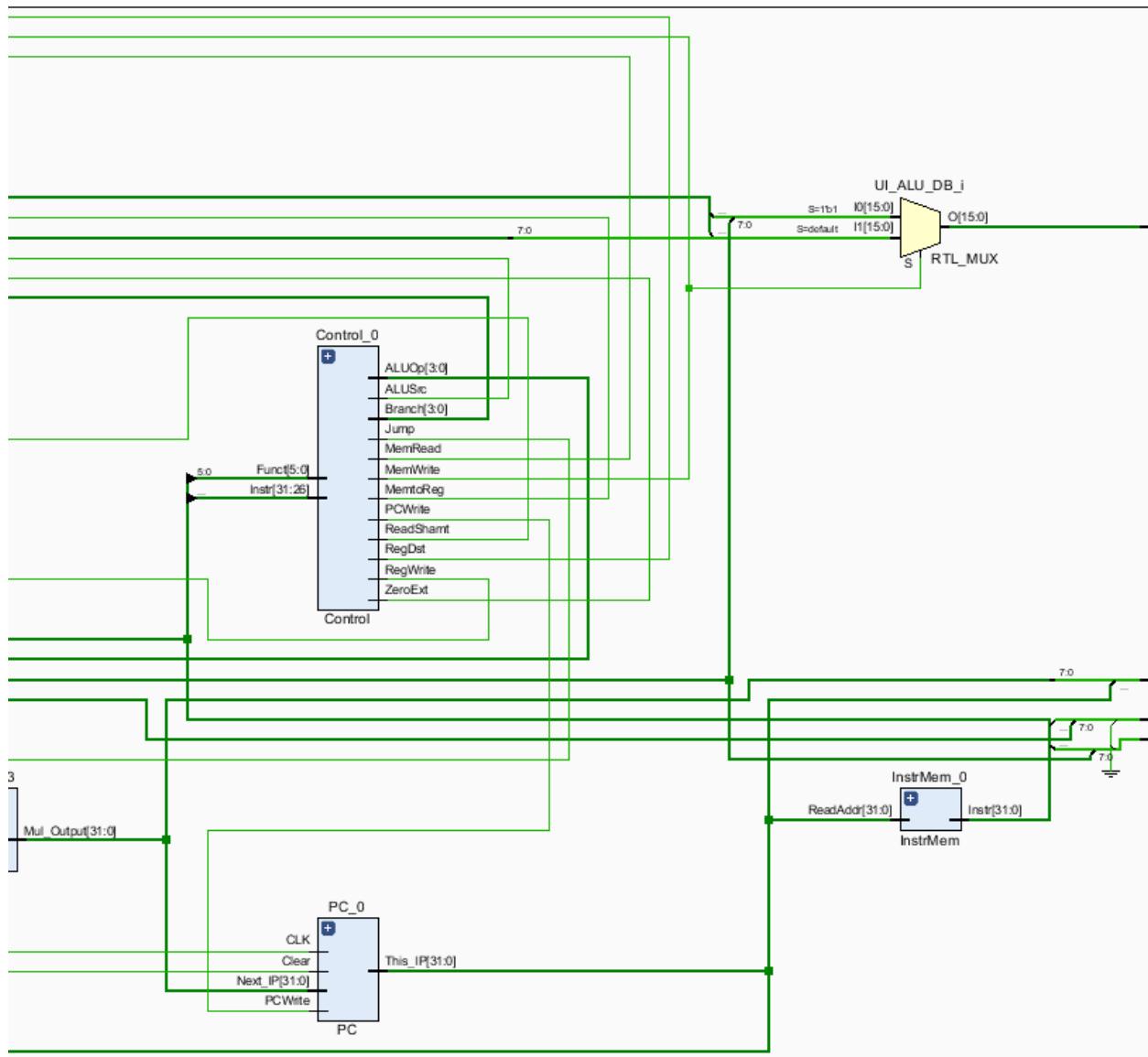
```
//Add32_With_4
assign Add_With4Output = Add_With4Input + 32'h00000004;
//SignExt
assign Output_32 = Input_Zero_Ext ? {16'b0, Input_16[15:0]} :
{{16{Input_16[15]}}, Input_16[15:0]}; //Input_Zero_Ext = 1 时进行0扩展，否则进行符号扩展
//Mux
assign Mul_Output = Mul_Sel? Mux_Input_2 : Mux_Input_1;
```

10. 最终连接出的顶层设计

通过在Top.v中实例化各个模块并通过wire类型变量进行“接线”，最终完成了CPU的顶层设计。如下图所示：

（由于CPU系统复杂，生成的图片过长，故从左到右切分为了3张图片以便查看）





(七)、进行仿真测试

1. 编写测试所用的rom代码

为了对编写出的CPU功能进行验证，我使用了老师提供的汇编代码进行测试。（保存为mips1.asm）

由于我的指令集设计是按照真实的MIPS 指令集进行，故整个过程使用可以工具自动化完成，大大节省了手动转换的时间并保证了正确性。

(1) 使用以下shell代码，直接调用MIPS版本的gcc交叉编译出了机器代码，使用objdump结合awk工具，直接得到了对应的16进制机器代码。为精简结果并与老师的代码保持一致，我删除了gcc插入的几条nop指令（00000000）并因而调整了部分跳转指令的跳转地址，最后保存为rom_16.txt。

```
cat mips1.asm
mips-netbsd-elf-gcc -S mips1.asm
mips-netbsd-elf-objdump -d a.out | awk '{print $2}' | grep '[0-9a-f]{8}'
```

```
lixinrui@lixinruis-MBP ~/Downloads>
→ Downloads cat mips1.asm
addi $1,$0,8
ori $2,$0,2
add $3,$2,$1
sub $5,$3,$2
and $4,$5,$2
or $8,$4,$2
L1: sll $8,$8,1
bne $8,$1,L1
slt $6,$2,$1
slt $7,$6,$0
L2: addi $7,$7,8
beq $7,$1,L2
sw $2,4($1)
lw $9,4($1)
L4: bgtz $9,L3
nop
L3: addi $9,$0,-1
j L4
→ Downloads mips-netbsd-elf-gcc -S mips1.asm
mips-netbsd-elf-gcc: warning: mips1.asm: linker input file unused because linking not done
→ Downloads mips-netbsd-elf-objdump -d a.out | awk '{print $2}' | grep '[0-9a-f]\{8\}'
20010008
34020002
00411820
006222822
00a22024
00824025
00084040
1501ffffe
00000000
0041302a
00c0382a
20e70008
10e1ffffe
00000000
ac220004
8c290004
00000000
1d200002
00000000
00000000
08000011
2009ffff
→ Downloads
```

(2) 使用以下Python3脚本转换为2进制

```
with open('rom_16.txt') as src:
    for row in src.readlines():
        print("{0:032b}".format(int(row, 16)))
```

```
In [13]: with open('rom_16.txt') as src:
    ...:     for row in src.readlines():
    ...:         print("{:0:032b}".format(int(row, 16)))
00100000000000100000000000001000
00110100000000100000000000000010
00000000100001000110000100000
0000000011000100010100000100010
00000000101000100010000000100100
00000000100000100100000000100101
00000000000010000100000000100000
00010101000000111111111111110
00000000010000010011000000101010
00000000110000000011100000101010
00100000111001110000000000001000
00010000111000011111111111110
101011000010001000000000000000100
100011000010100100000000000000100
0001110100100000000000000000000001
1111110000000000000000000000000000
00100000000010011111111111111111
000010000001000000000000000000001110
```

最终得到得到下表。

地址	汇编程序	指令代码						
		op (6)	rs(5)	rt(5)	rd(5)/immediate (16)/shamt(5)	Funct(5)	16进制数代码	
0x000000000	addi \$1,\$0,8	001000	00000	00001	00000000 00001000	/	20010008	
0x000000004	ori \$2,\$0,2	001101	00000	00010	00000000 00000010	/	34020002	
0x000000008	add \$3,\$2,\$1	000000	00010	00001	00011	00000	00411820	
0x00000000C	sub \$5,\$3,\$2	000000	00011	00010	00101	00010	00622822	
0x000000010	and \$4,\$5,\$2	000000	00101	00010	00100	00100	00a22024	
0x000000014	or \$8,\$4,\$2	000000	00100	00010	01000	00101	00824025	
0x000000018	sll \$8,\$8,1	000000	00000	01000	000 01	00000	00084040	
0x00000001C	bne \$8,\$1,-2 (=, 转18)	000101	01000	00001	11111111 11111110	/	1501ffff	

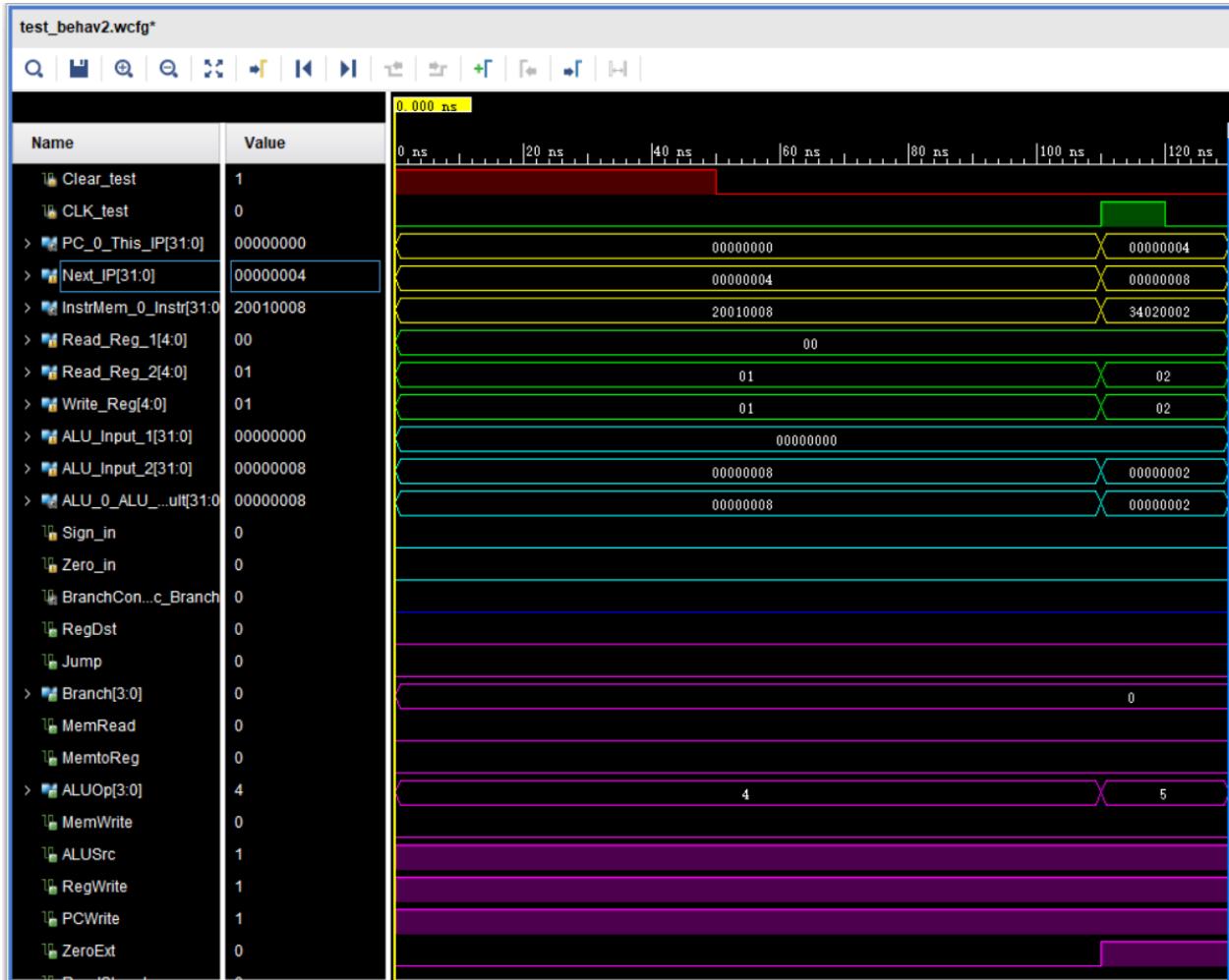
0x000000020	slt \$6,\$2,\$1	000000	00010	00001	00110	01010	0041302a
0x000000024	slt \$7,\$6,\$0	000000	00110	00000	00111	01010	00c0382a
0x000000028	addi \$7,\$7,8	001000	00111	00111	00000000 00001000	/	20e70008
0x00000002C	beq \$7,\$1,-2 (=, 转28)	000100	00111	00001	11111111 11111110	/	10e1ffffe
0x000000030	sw \$2,4(\$1)	101011	00001	00010	00000000 00000100	/	ac220004
0x000000034	lw \$9,4(\$1)	100011	00001	01001	00000000 00000100	/	8c290004
0x000000038	bgtz \$9,1 (>0,转40)	000111	01001	00000	00000000 00000001	/	1d200001
0x00000003C	halt	111111	00000	00000	00000000 00000000	/	fc000000
0x000000040	addi \$9,\$0,-1	001000	00000	01001	11111111 11111111	/	2009ffff
0x000000044	j 0x00000038	000010	00000	00000	00000000 00001110	/	0810000e
0x000000048							
0x00000004C							

2. 执行仿真测试

如下图所示，仿真窗口中右侧从上到下由红到紫的一系列变量，分别是清零信号和时钟信号（红）、PC和指令（黄）、rs/rt/rd的地址（绿）、ALU的输入和输出（青）、分支跳转信号（蓝），Control Unit的各个控制信号。通过查看这些信号的值，足以验证程序正确执行了所有算数、逻辑、分支语句，并最终正确进入了停机状态。

图一

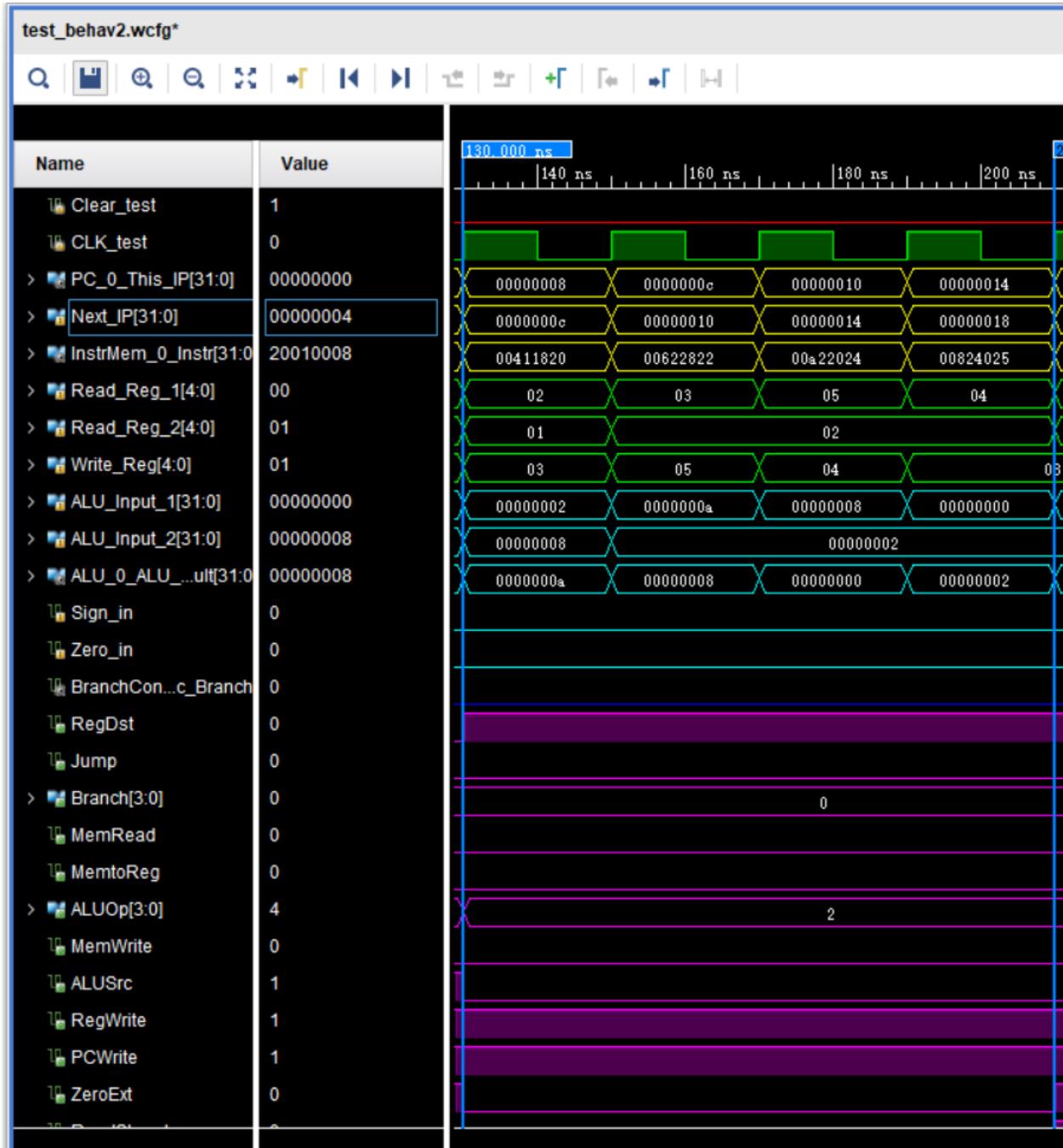
PC	执行的代码
0x000000000	addi \$1,\$0,8
0x000000004	ori \$2,\$0,2



注：在Clear信号变为低，清零结束后一段时间，仿真程序才发出第一个时钟，模拟了手动单步执行时的场景。

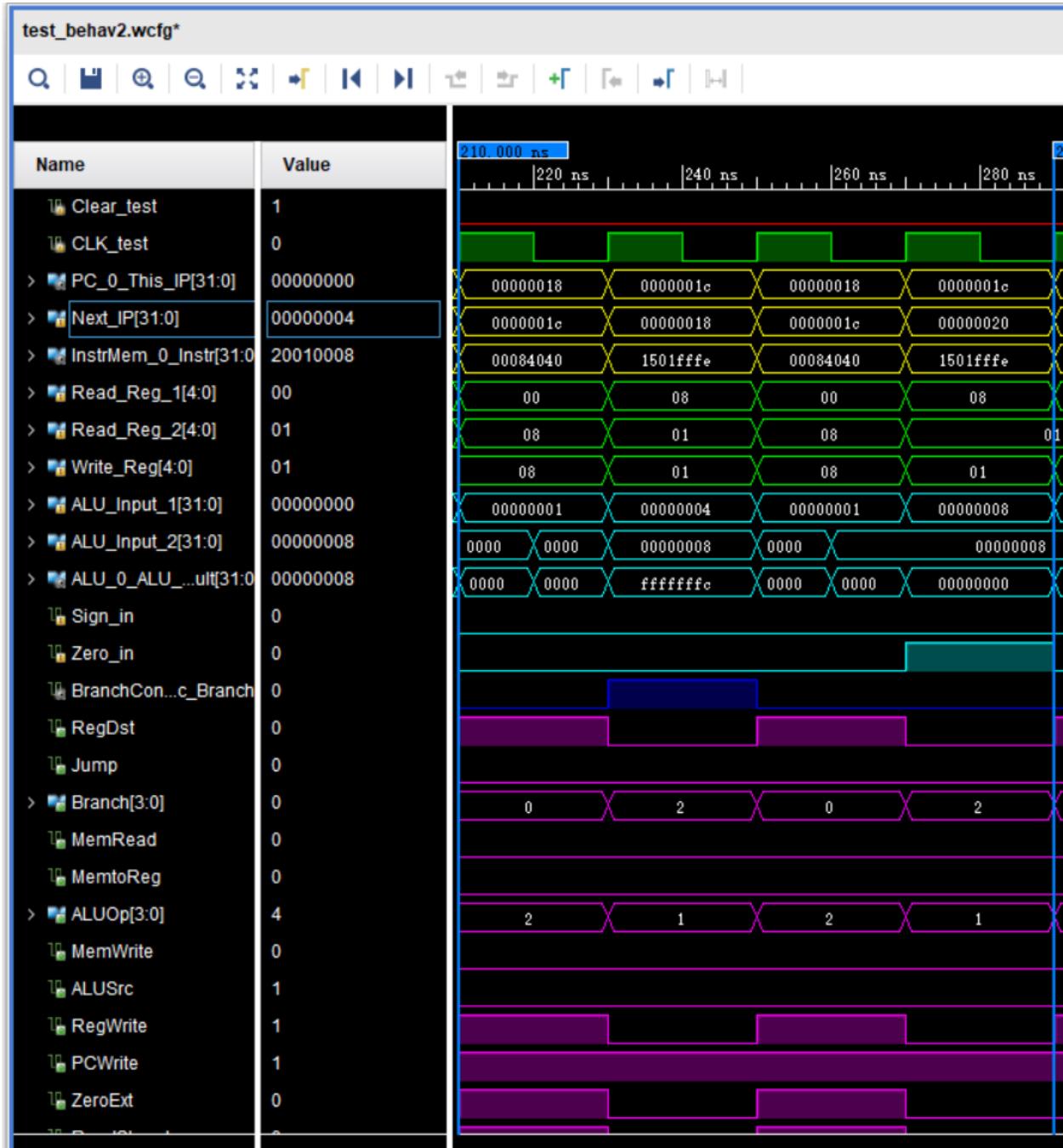
图二

PC	执行的代码
0x00000008	add \$3,\$2,\$1
0x0000000C	sub \$5,\$3,\$2
0x00000010	and \$4,\$5,\$2
0x00000014	or \$8,\$4,\$2



图三

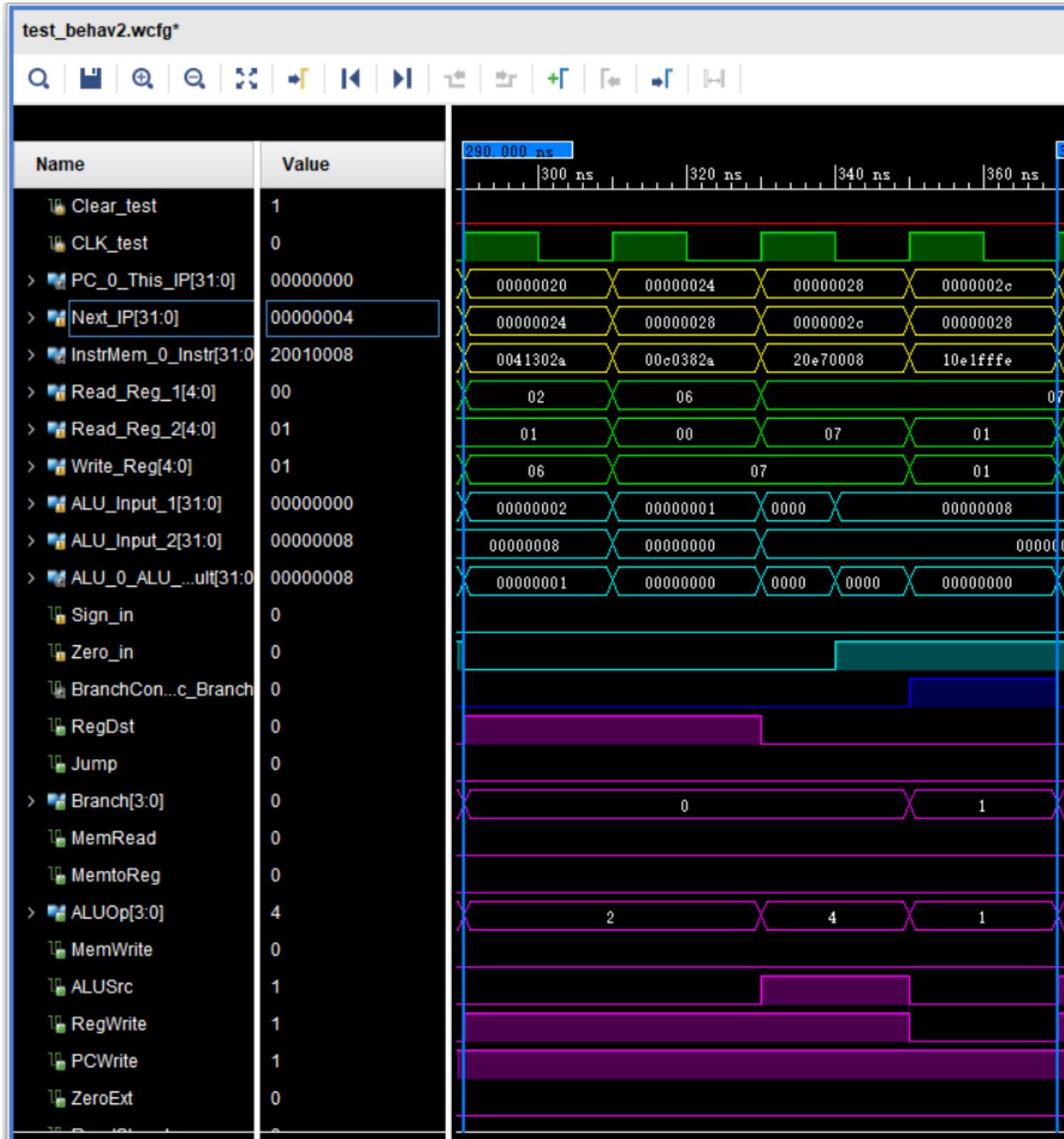
PC	执行的代码
0x00000018	sll \$8,\$8,1
0x0000001C	bne \$8,\$1,-2 (#,转18)
0x00000018	sll \$8,\$8,1
0x0000001C	bne \$8,\$1,-2 (#,转18)



这里正确执行了两轮循环后继续执行0x00000020处的指令

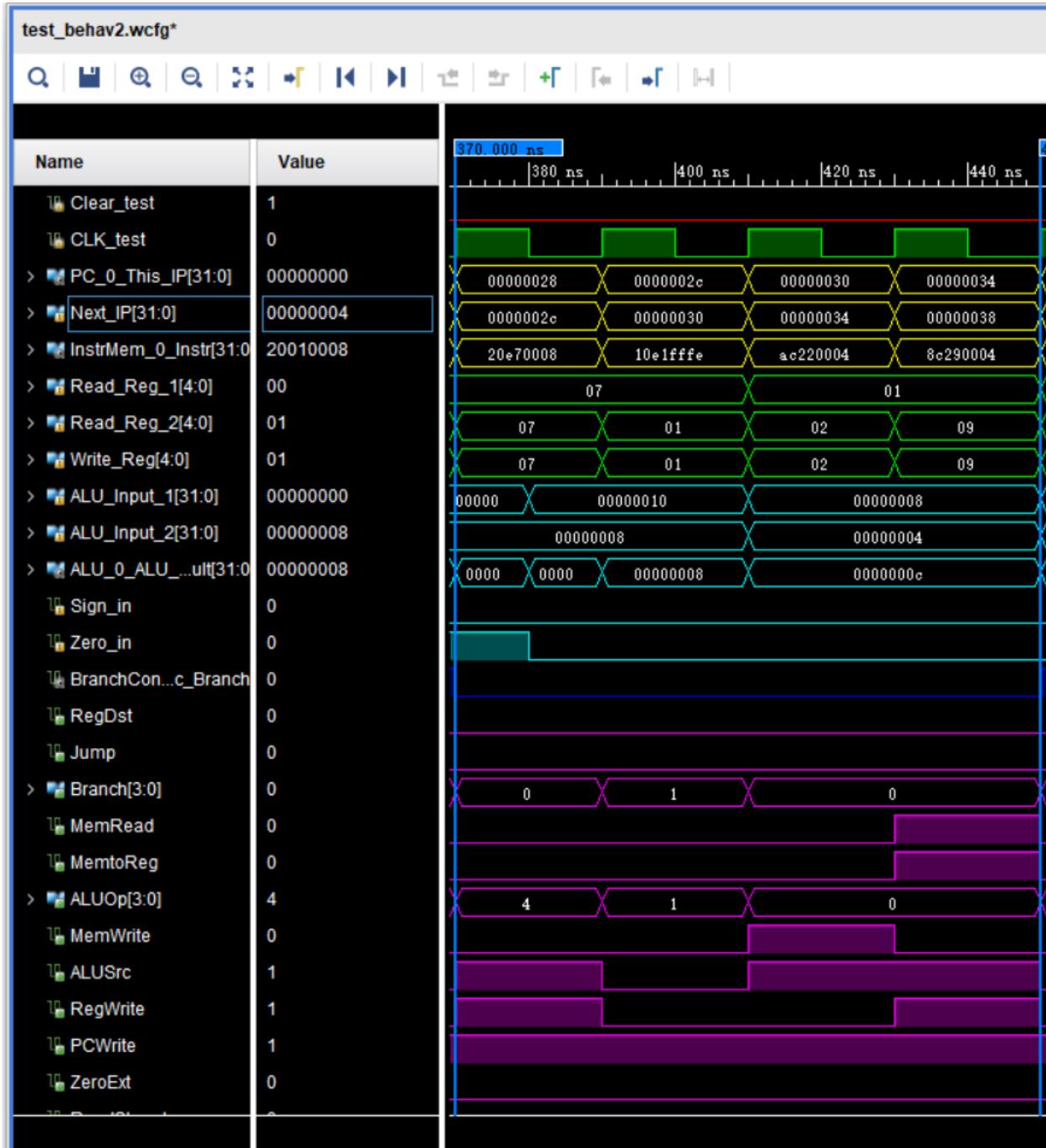
图四

PC	执行的代码
0x00000020	slt \$6,\$2,\$1
0x00000024	slt \$7,\$6,\$0
0x00000028	addi \$7,\$7,8
0x0000002C	beq \$7,\$1,-2 (=,转28)



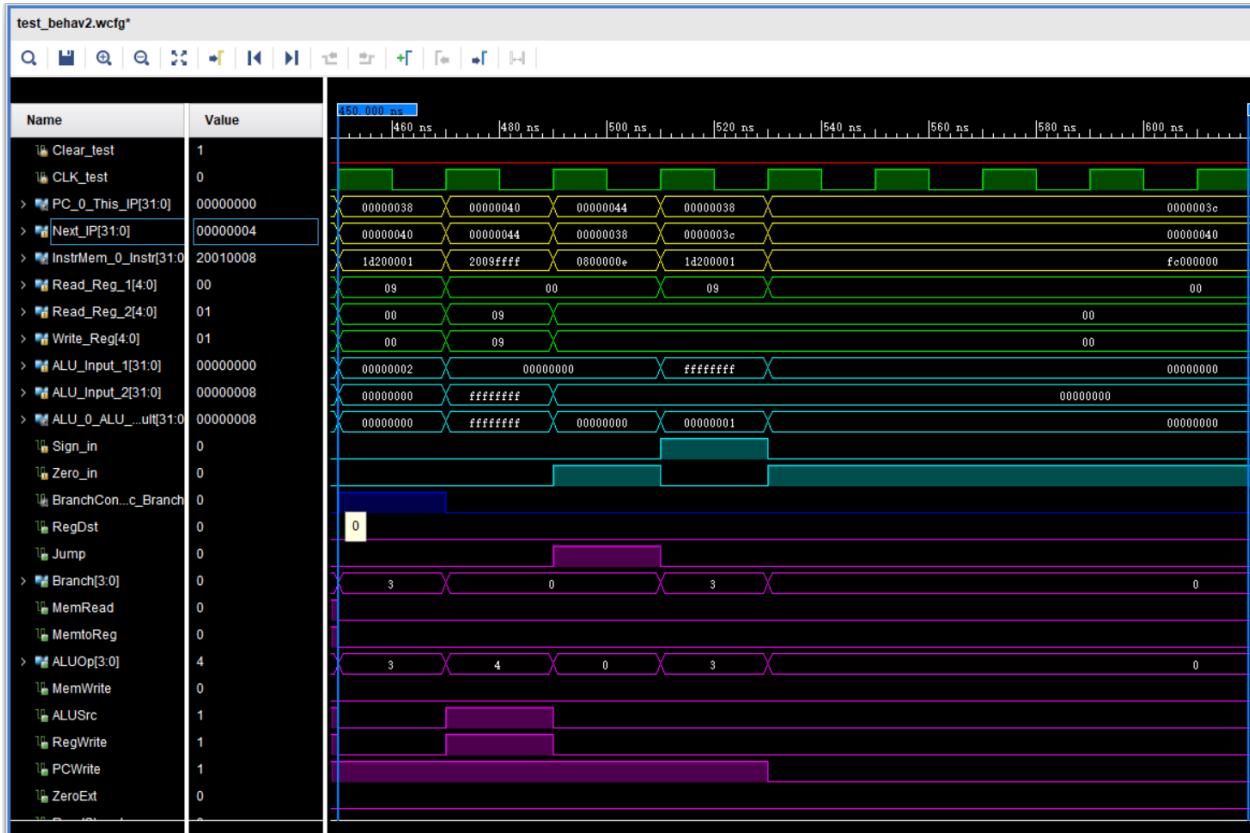
图五

PC	执行的代码
0x00000028	addi \$7,\$7,8
0x0000002C	beq \$7,\$1,-2 (=,转28)
0x00000030	sw \$2,4(\$1)
0x00000034	lw \$9,4(\$1)



图六

PC	执行的代码
0x00000038	bgtz \$9,1 (>0,转40)
0x00000040	addi \$9,\$0,-1
0x00000044	j 0x00000038
0x00000038	bgtz \$9,1 (>0,转40)
0x0000003C	halt



(八) 编写、测试、连接输入输出模块

1. 编写输入输出模块

输入模块主要由时钟分频、输入消抖、7段显示管驱动三个部分组成。

时钟分频的原理是定义一个计数器cnt，每当原始时钟CLK到来时加一。当计数器计满 $2^N - 1$ 后，再将分频时钟CLK_Slow翻转一次，这样就将CLK进行了 2^N 分频，代码如下

```
module clk_slow(
    input CLK_100mhz, //100mhz的原始时钟信号
    output reg CLK_slow //输出的分频时钟信号
);
reg [31:0] count = 0; //计数器
reg [31:0] N = 524288/2-1; //分频2^19，得到一个周期大约5ms的信号
initial CLK_slow = 0;
always @(posedge CLK_100mhz) begin
    if(count >= N)
        begin
            count <= 0;
            CLK_slow <= ~CLK_slow;
        end
    else begin
        count <= count + 1;
    end
end
endmodule
```

输入消抖同样使用了N位一个计数器cnt，当时钟信号到来时，如果按键处于按下状态就将计数器+1，而如果发现按键未按下，则清空计数器。将模块输出的防抖按键信号连接到cnt第N位，这样就保证持续按下 2^N 个时钟周期的时间后才会输出按键信号，从而屏蔽了短暂的抖动信号。

7段显示管驱动的原理是：7段数码管每一段的点亮与否由一个信号控制，给出一个segment[7]的信号，同时控制给出这7段的点亮和熄灭情况，就可以绘制出所期望的数字。（这里segment信号其实是8位，因为Basys3的数码管上还有一个小数点DP，要用一位来控制）。多个相连的7段数码管同一时刻实际上只能显示一个数字，同时显示多个数字的原理是：给出一个周期性1110->1101->1011->0111的4位select扫描信号，每个时间点点亮一个显示管，显示一个数字，在超过人眼可以识别的频率下造成4个显示管同时点亮并显示不同值的效果。实现上述操作的代码如下：

```

always @ (posedge CLK_slow)
begin
    begin
        case (cnt) //扫描显示4个数码管
            2'b00: begin Select = 4'b0111; Segdata <= Data[3:0];end
            2'b01: begin Select = 4'b1011; Segdata <= Data[7:4];end
            2'b10: begin Select = 4'b1101; Segdata <= Data[11:8];end
            2'b11: begin Select = 4'b1110; Segdata <= Data[15:12];end
            default: Segdata <= 0;
        endcase
    end
end
always @ (*)
begin
    if(Clear)
    begin
        Segment <= 8'b11111111;
    end
    else
    begin
        case(Segdata) //各个数字对应的段选信号
            4'h0: Segment = 8'b10000001;
            4'h1: Segment = 8'b11001111;
            4'h2: Segment = 8'b10010010;
            4'h3: Segment = 8'b10000110;
            4'h4: Segment = 8'b11001100;
            4'h5: Segment = 8'b10100100;
            4'h6: Segment = 8'b10100000;
            4'h7: Segment = 8'b10001111;
            4'h8: Segment = 8'b10000000;
            4'h9: Segment = 8'b10000100;
            4'hA: Segment = 8'b10001000;
            4'hB: Segment = 8'b11100000;
            4'hC: Segment = 8'b10110001;
            4'hD: Segment = 8'b11000010;
            4'hE: Segment = 8'b10110000;
            4'hF: Segment = 8'b10111000;
            default: Segment <= 8'b11111111;
        endcase
    end
end

```

2. 测试输入输出模块，和CPU模块连接形成最终程序

实验项目中输入输出有关的模块是和实际硬件密切相关，容易出错的部分。因此我将输入输出模块和CPU设计为两个不耦合的子系统，编写了一个TestIO.v，烧写到板上单独测试输入输出模块的功能。编写test.v通过仿真单独测试CPU的工作。在两者测试均正常工作后将输入输出模块和CPU的顶层文件连接起来。最终形成UserInterface.v，作为烧入到开发板上的最终程序。

```

module UserInterface(
    input Btn_Press, //按键输入

```

```

input Func_Choose1, //输出选项信号高位
input Func_ChOOSE0, //输出选择信号低位
input CLK, //100mhz时钟信号
input ClearCPU, //CPU状态重置按键, 消抖后作为清零信号, 高电平有效
input ClearUI, //显示管和防抖模块重置信号, 高电平有效
output [3:0] Select, //数码管扫描信号
output [7:0] Segment //数码管段选信号
);

wire Btn_State; //消抖后的运行按钮信号, 作为单步时钟
wire Clr_State; //消抖后的清零信号
wire CLK_slow; //5ms周期慢时钟
reg [15:0] Data; //数码管显示的数据
wire [1:0] Func_ChOOSE; //选择显示什么
assign Func_ChOOSE = {Func_ChOOSE1, Func_ChOOSE0};
wire [15:0] UI_PC_NextPC; //PC和下一条指令PC
wire [15:0] UI_RsAddr_RsValue; //Rs地址和值
wire [15:0] UI_RtAddr_RtValue; //Rt地址和值
wire [15:0] UI_ALU_DB; //ALU输出和DataBus的值
//CPU顶层
top top_0 (
    .Clear_top(Clr_State),
    .CLK_top(Btn_State),
    .UI_PC_NextPC(UI_PC_NextPC),
    .UI_RsAddr_RsValue(UI_RsAddr_RsValue),
    .UI_RtAddr_RtValue(UI_RtAddr_RtValue),
    .UI_ALU_DB(UI_ALU_DB)
);

```

```

always @ (posedge CLK)
begin
    case(Func_ChOOSE) //选择数码管显示什么
        2'b00: Data <= UI_PC_NextPC;
        2'b01: Data <= UI_RsAddr_RsValue;
        2'b10: Data <= UI_RtAddr_RtValue;
        2'b11: Data <= UI_ALU_DB;
    endcase
end
//分频模块
clk_slow clk_slow_0 (
    .CLK_100mhz(CLK),
    .CLK_slow(CLK_slow)
);
//消抖模块1(输出作为单步时钟)
Input_OBtn Input_OBtn_0(
    .CLK_slow(CLK_slow),
    .Clear(ClearUI),
    .Btn_Input(Btn_Press),
    .Btn_State(Btn_State)
);
//消抖模块2(输出作为清零信号)
Input_OBtn Input_OBtn_1(
    .CLK_slow(CLK_slow),

```

```

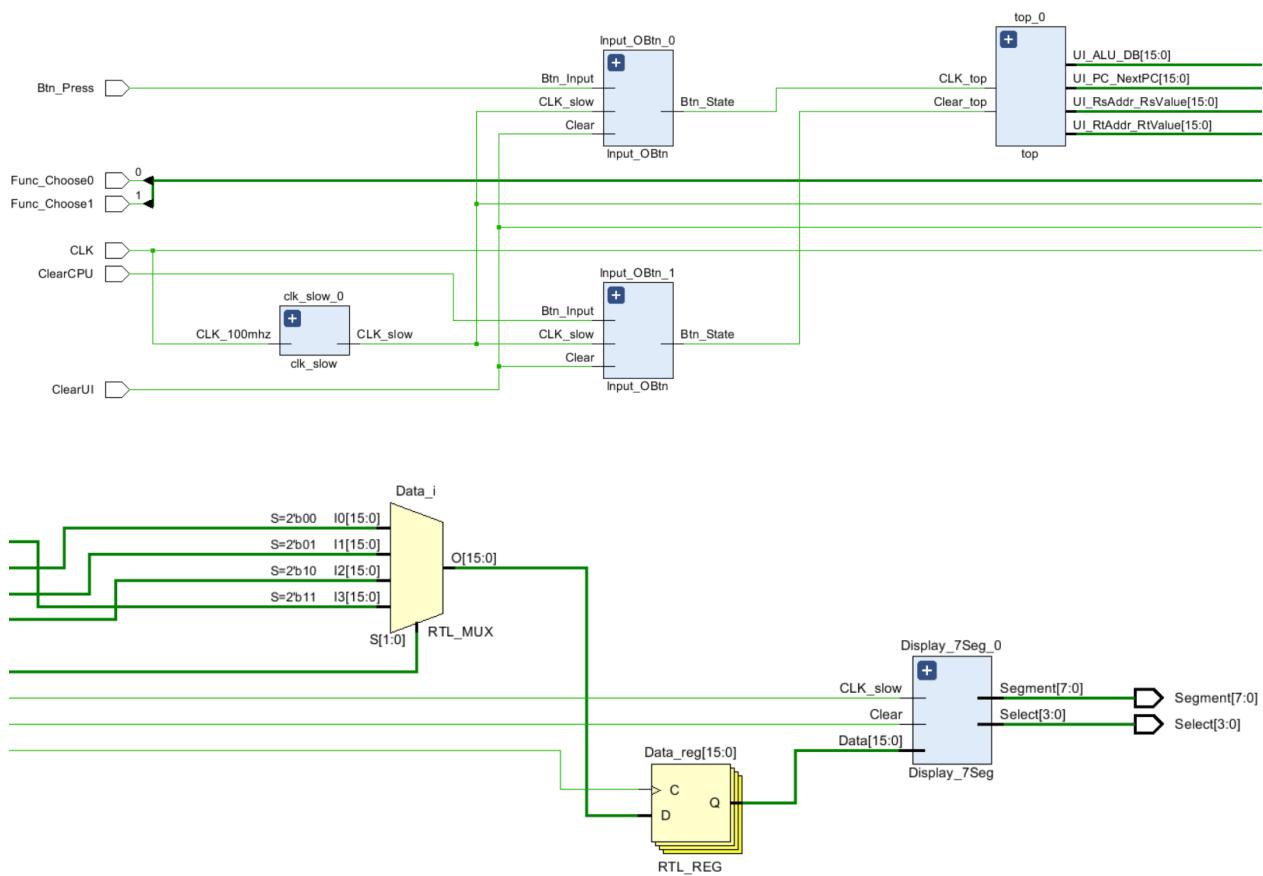
    .Clear(ClearUI),
    .Btn_Input(ClearCPU),
    .Btn_State(Clr_State)
);

//7段显示管
Display_7Seg Display_7Seg_0(
    .Data(Data),
    .Clear(ClearUI),
    .CLK_slow(CLK_slow),
    .Select(Select),
    .Segment(Segment)
);

```

endmodule

线路图如下：（从左到右切分为两张图）



(九)、烧写CPU到开发板，验证最终结果正确性

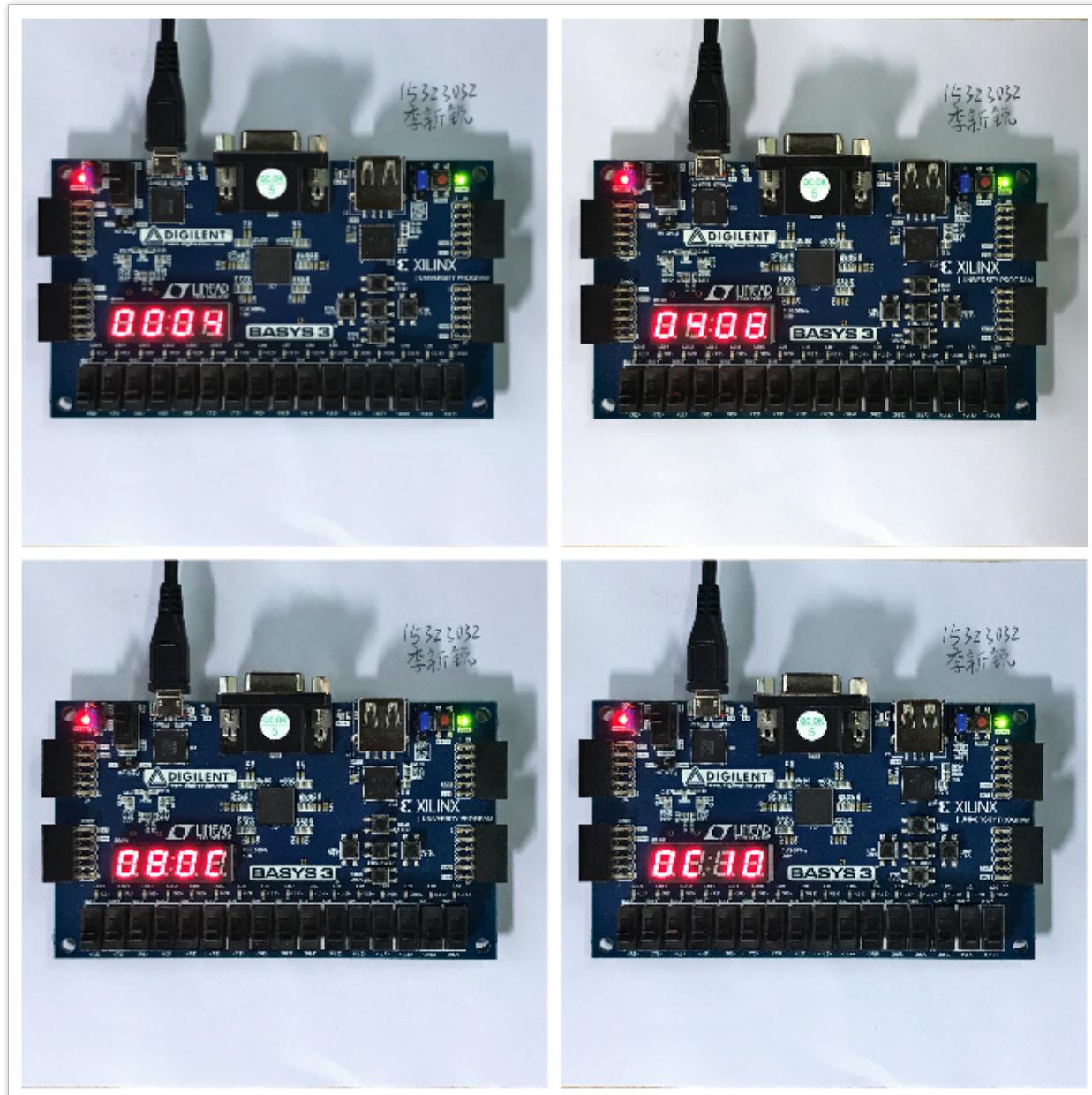
程序烧写到开发板后，首先按下按键W19进行清理（相当于初始化），然后不断按下按键T17即可不断单步执行。

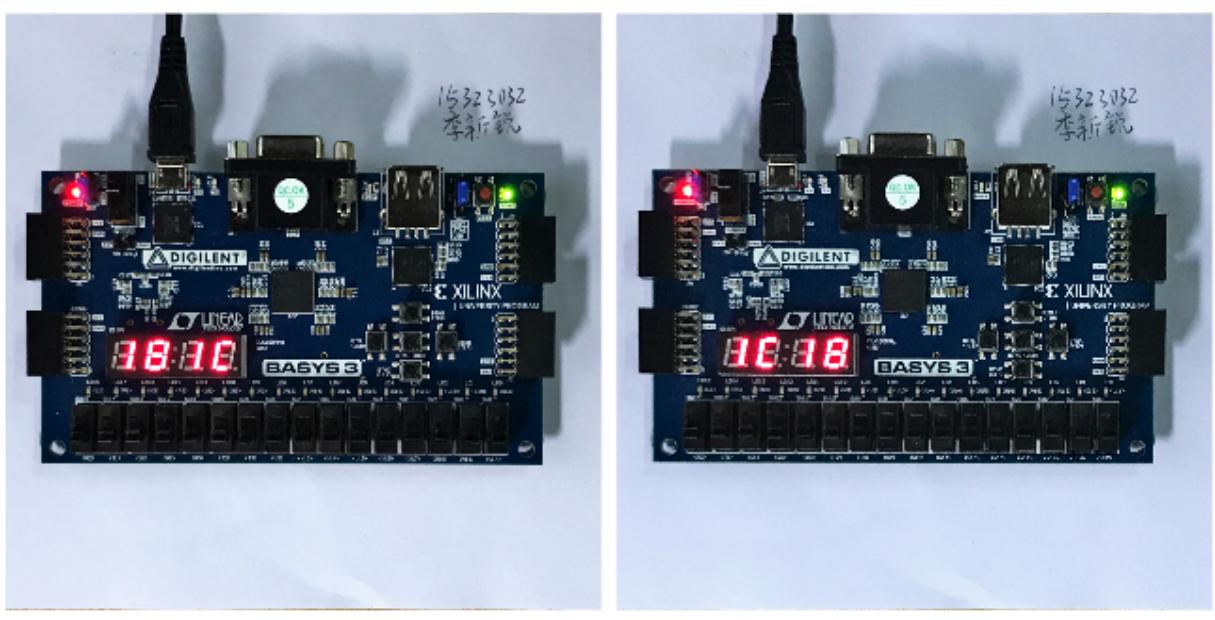
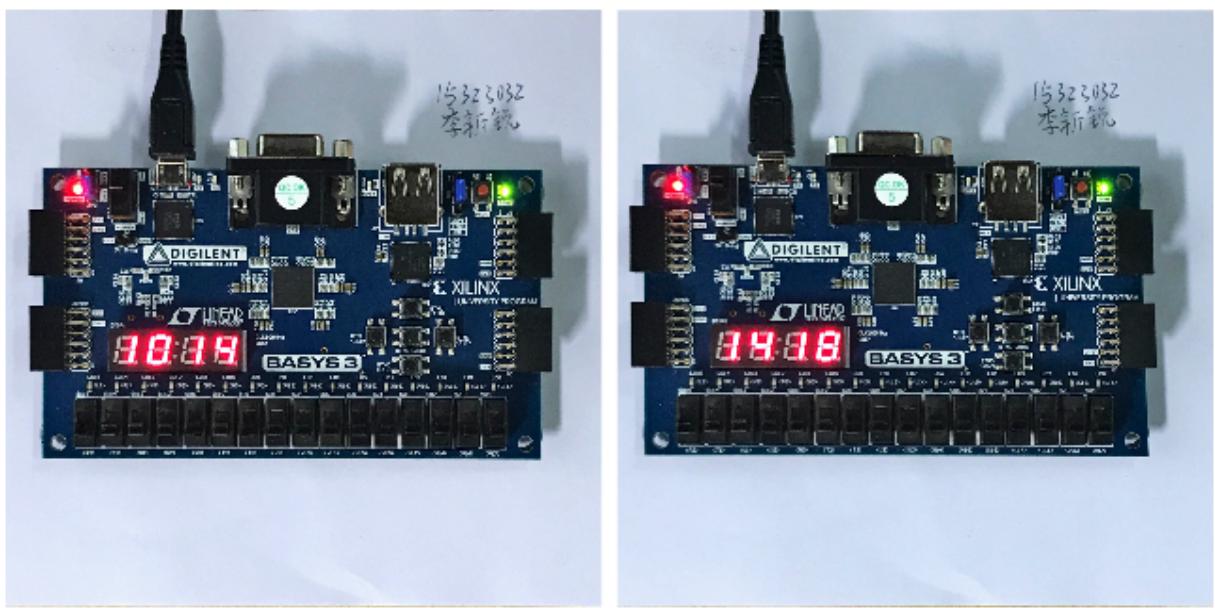
将开关{R2,T1}拨到00, 01, 10, 11四个状态，即可分别验证PC——NextPC、Rs——Rs Value、Rt——Rt Value、ALU——Data Bus数据的正确性，从而说明CPU设计的正确性。

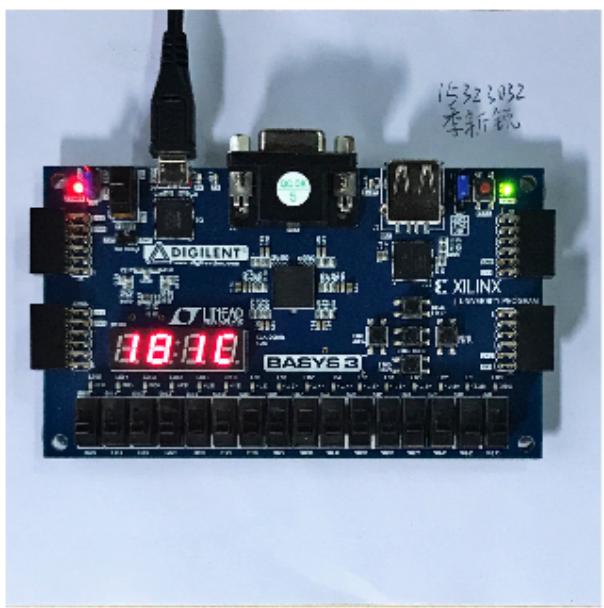
1.验证PC——NextPC正确性

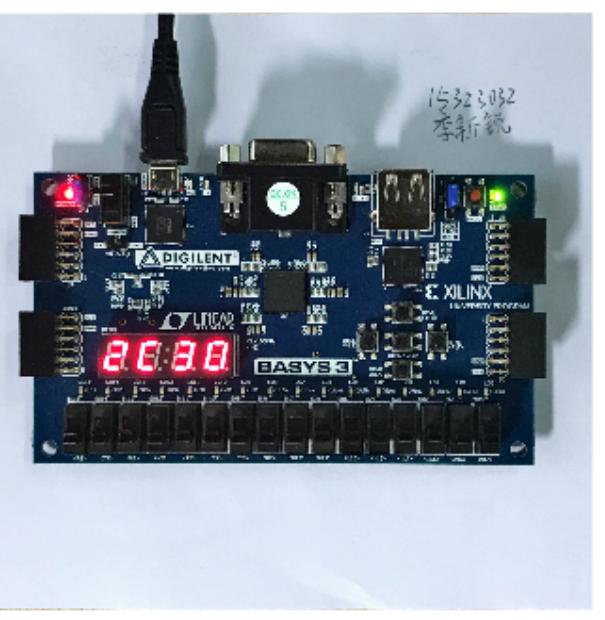
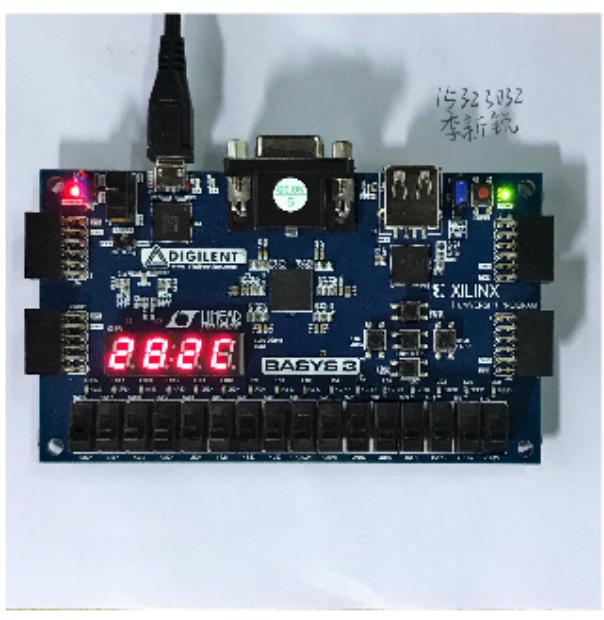
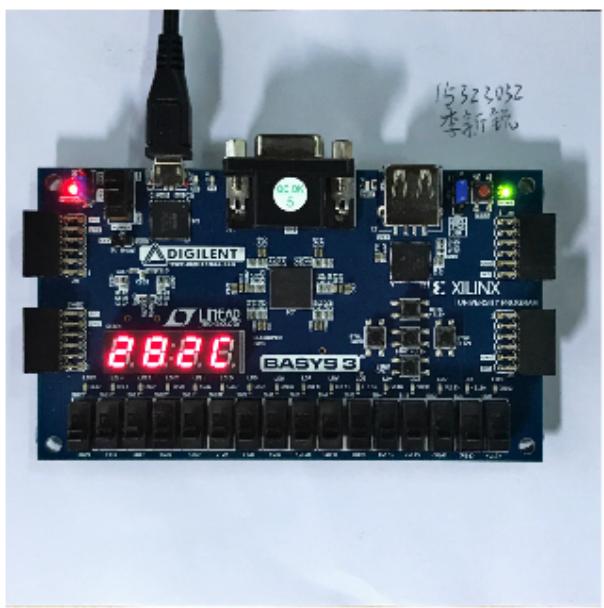
下面前23张截图显示了CPU单步执行PC和Next PC的变化，与预期结果相同。

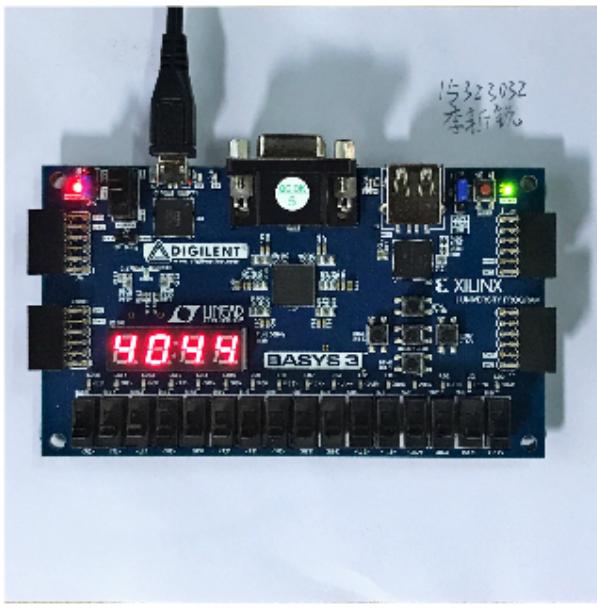
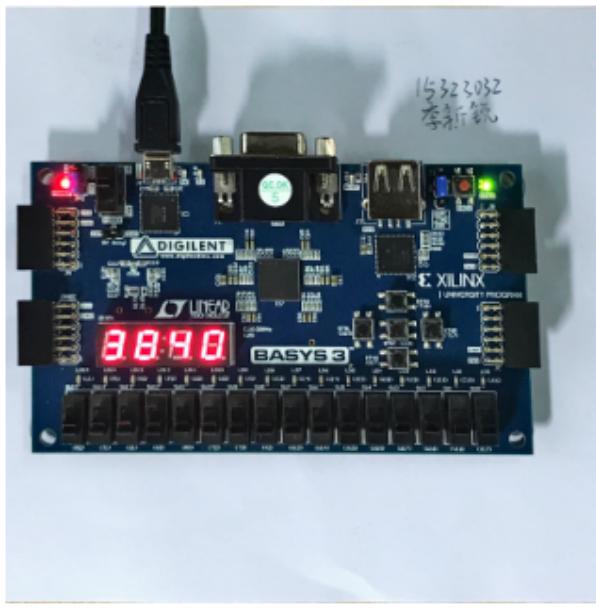
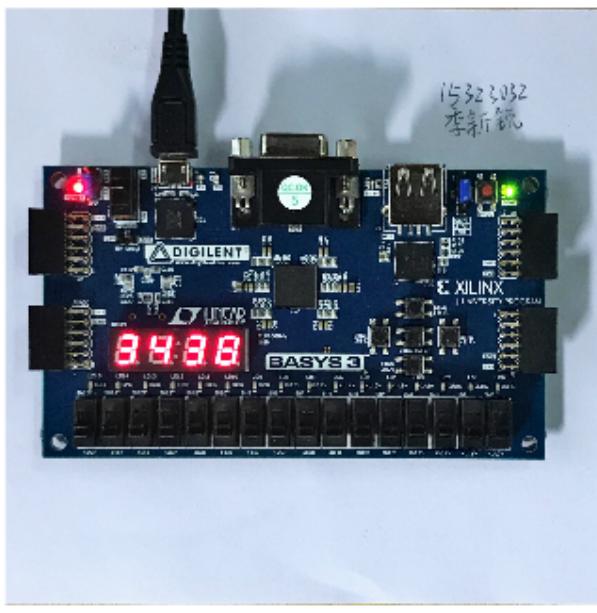
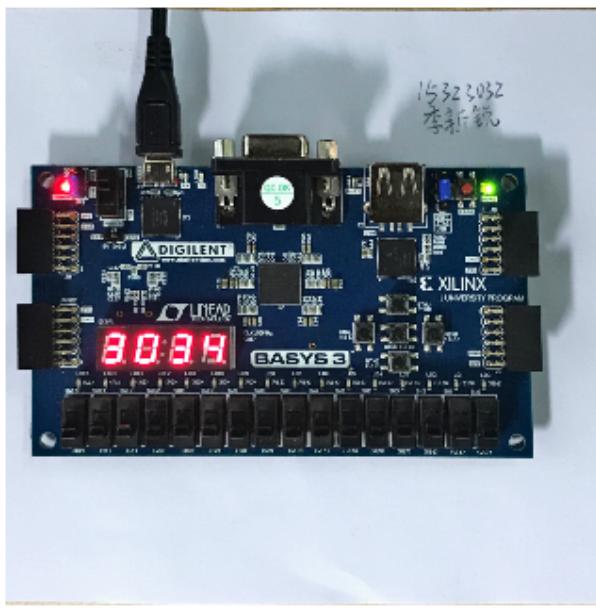
最后一张图片是按下清零后的效果，验证了将CPU状态重置功能的有效性。

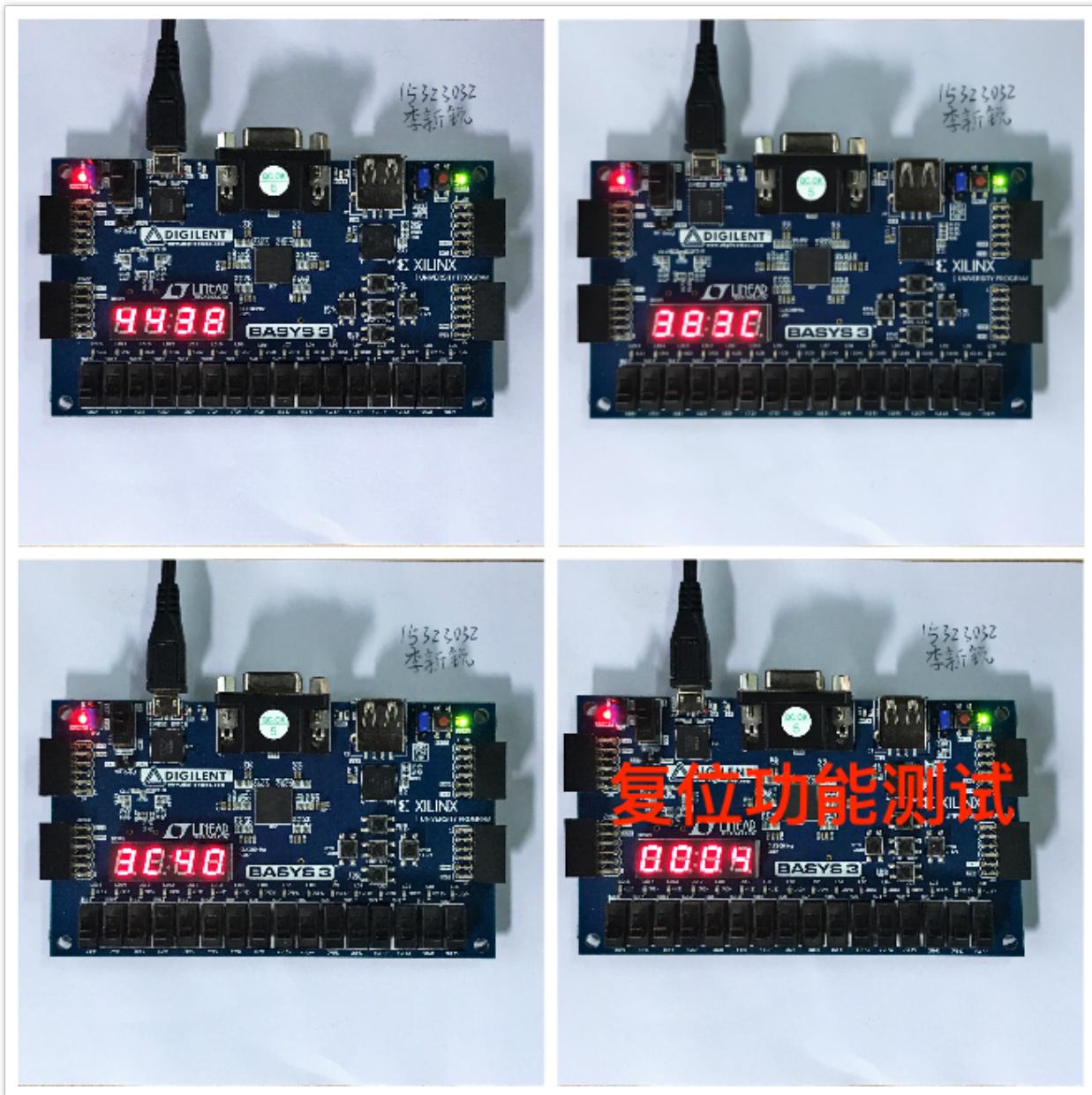




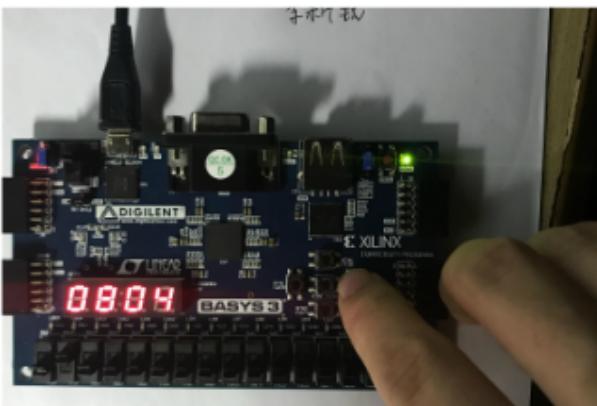
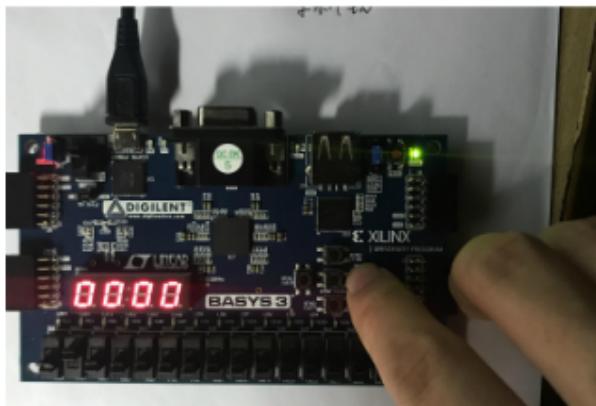
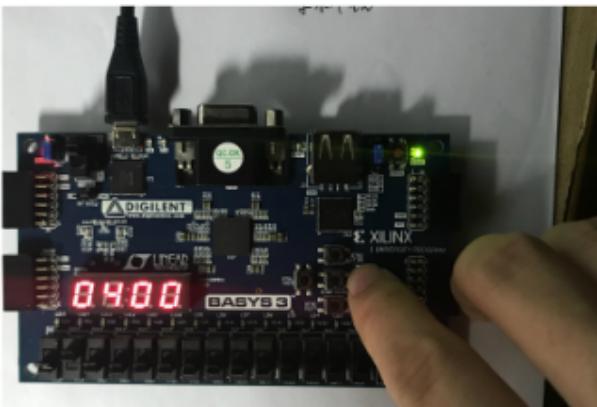
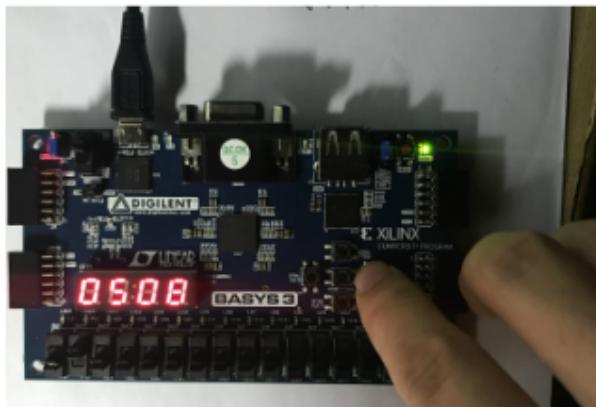
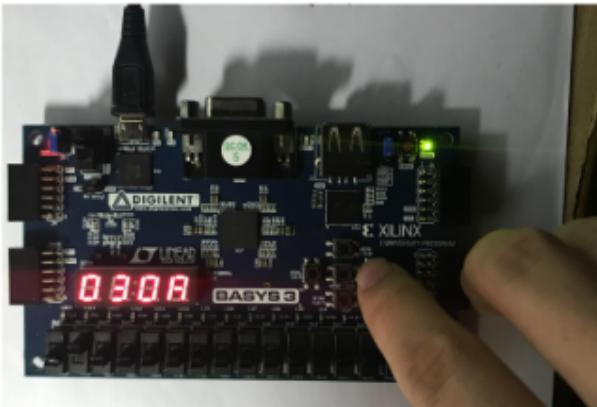
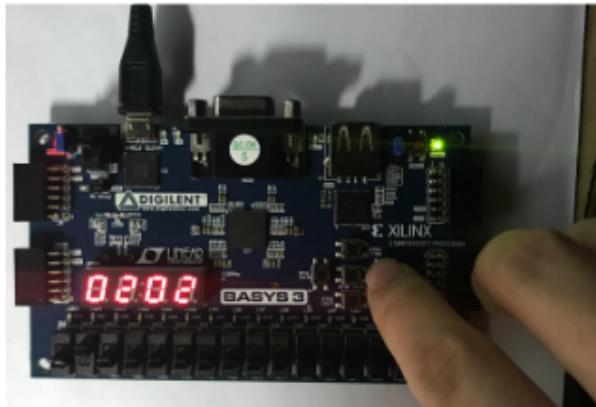
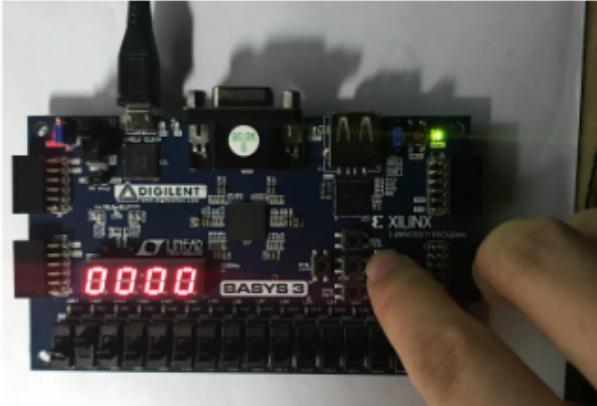
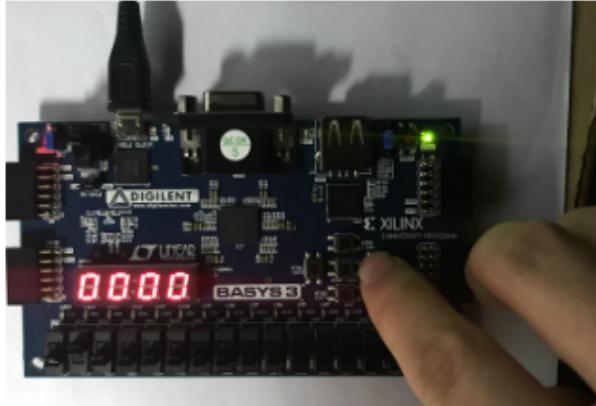


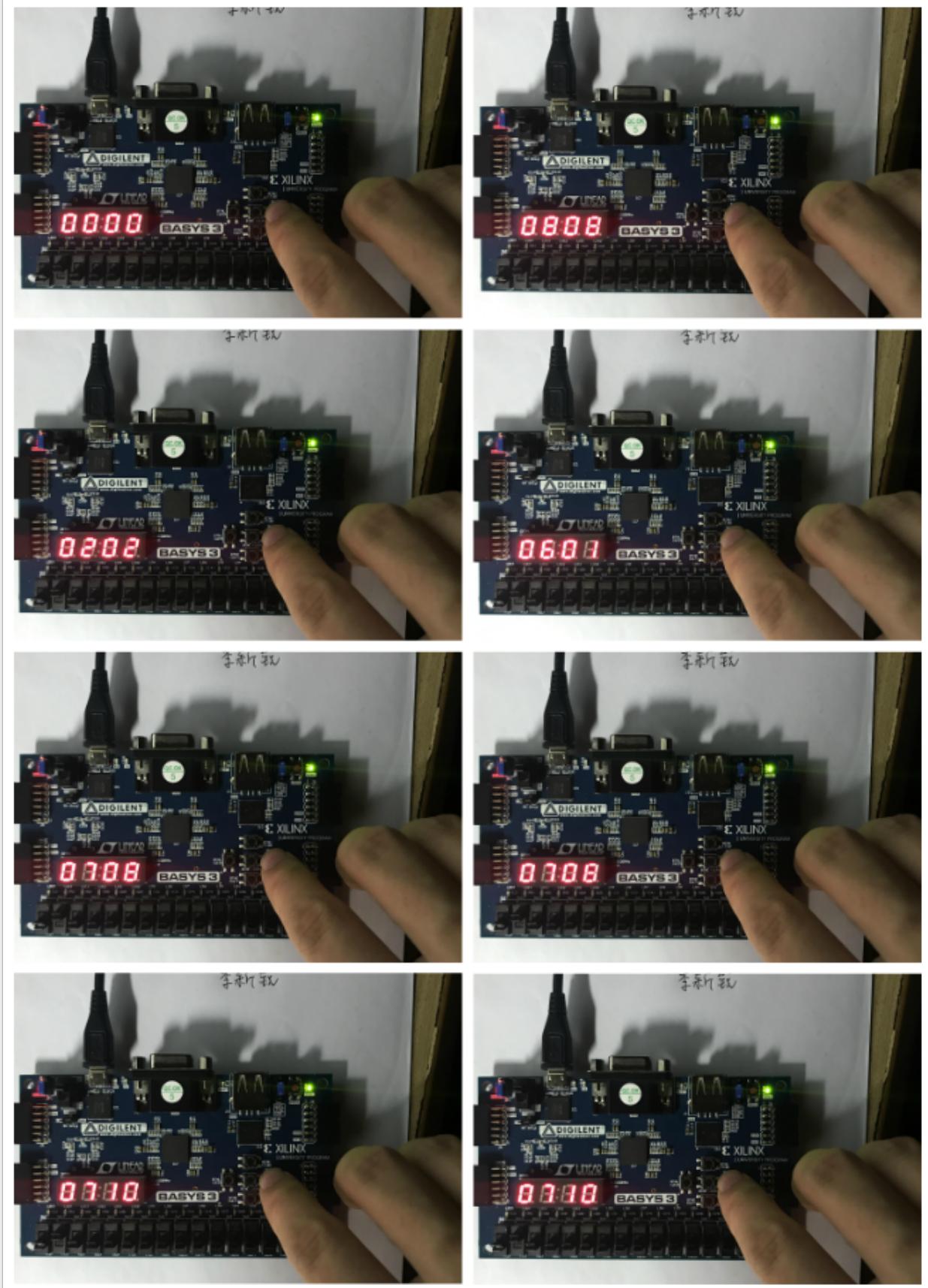


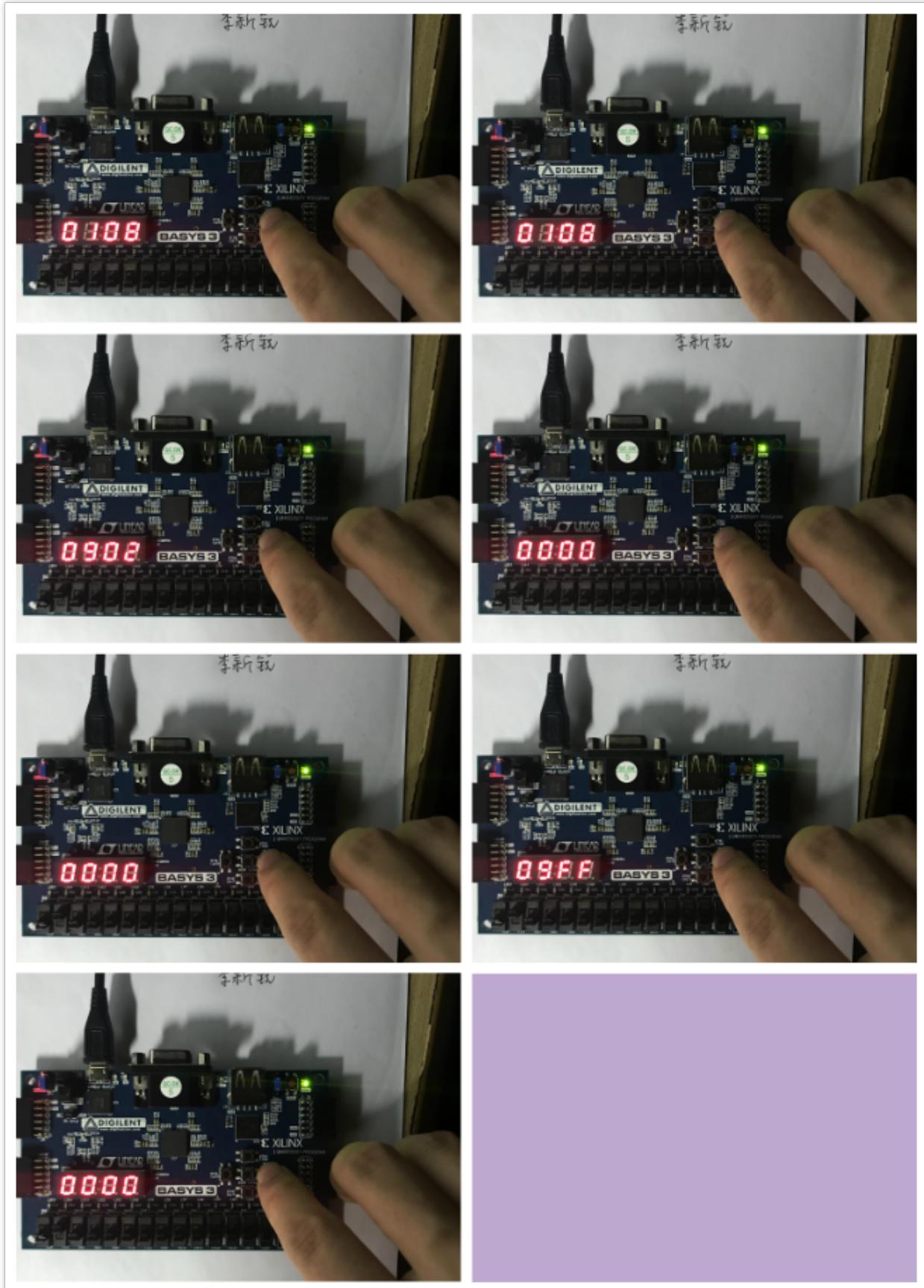




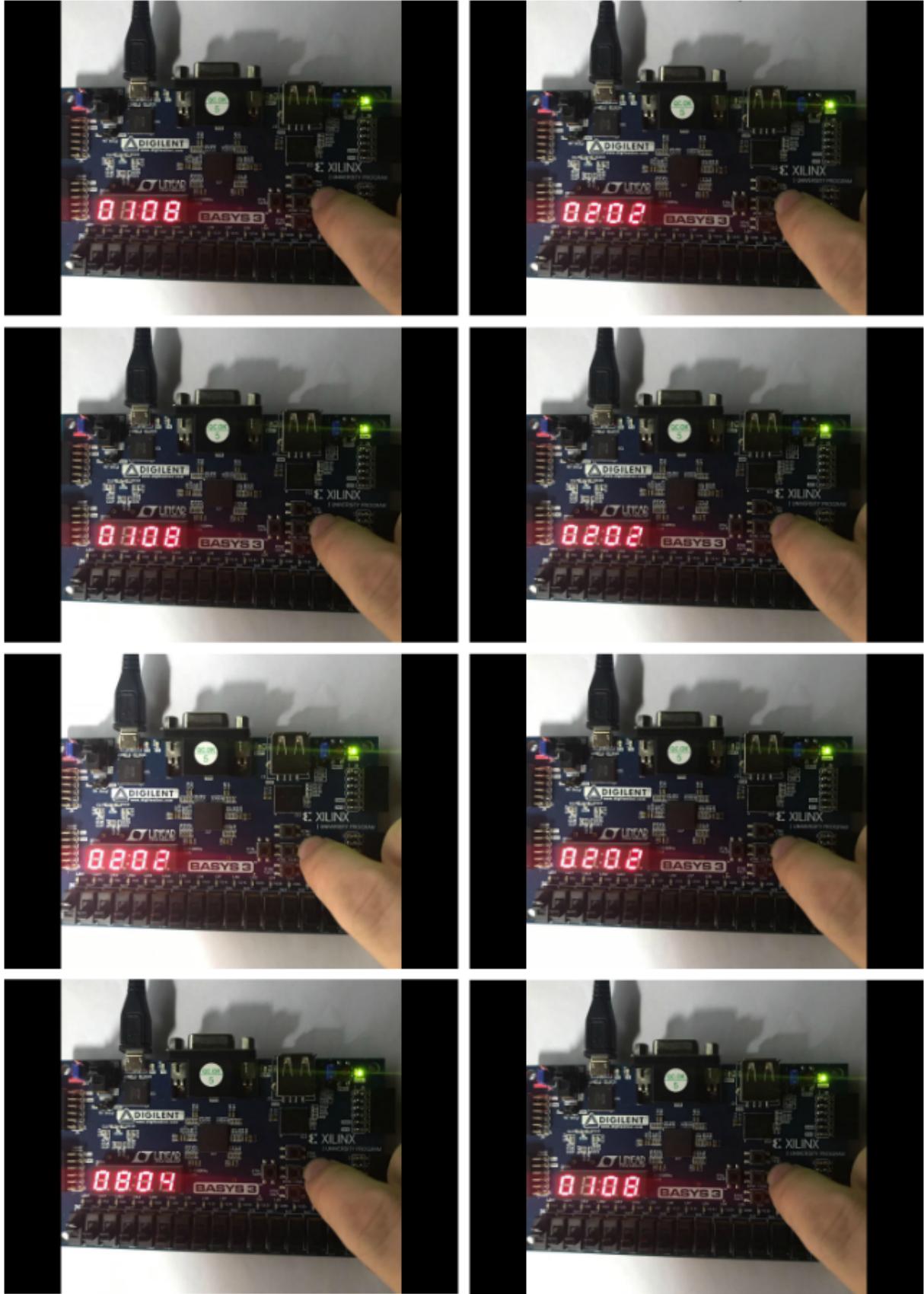
2.验证Rs——Rs Value正确性

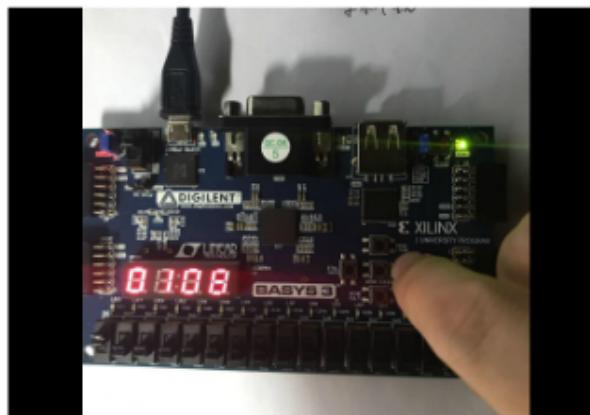
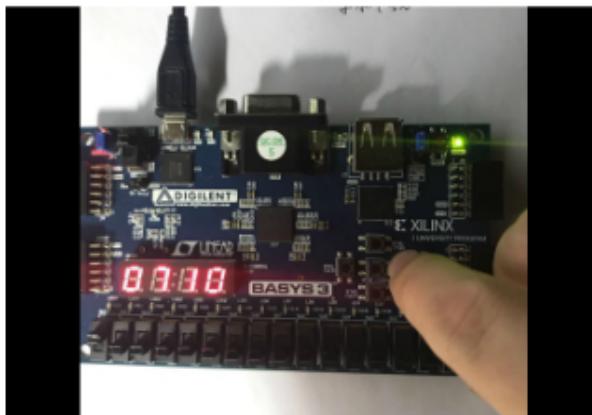
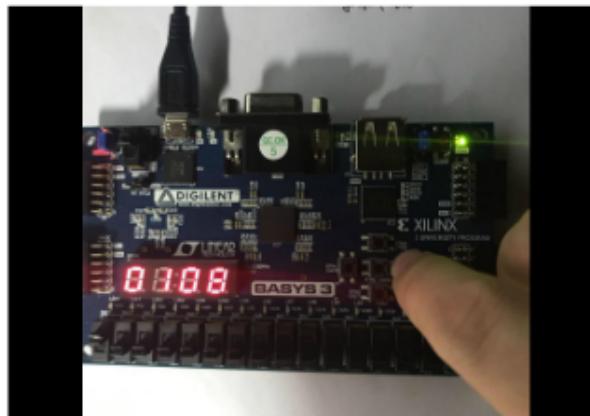
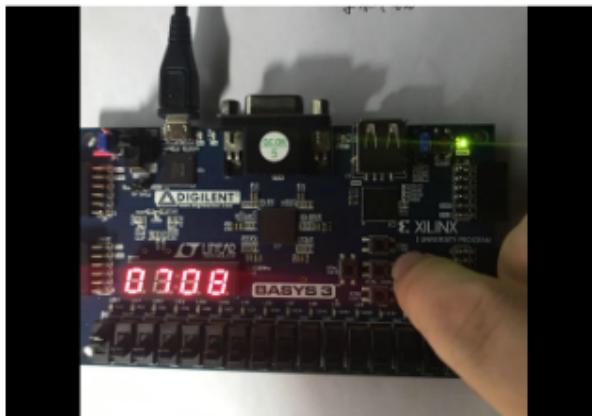
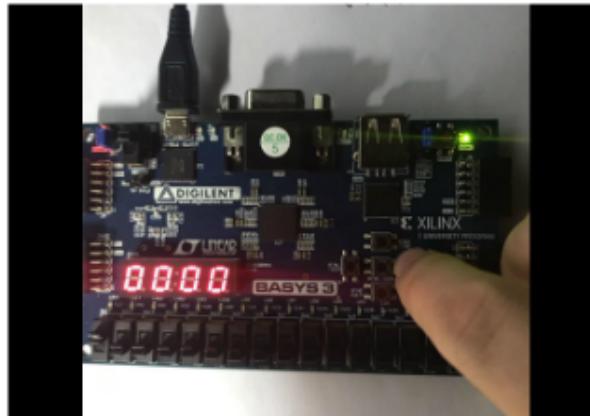
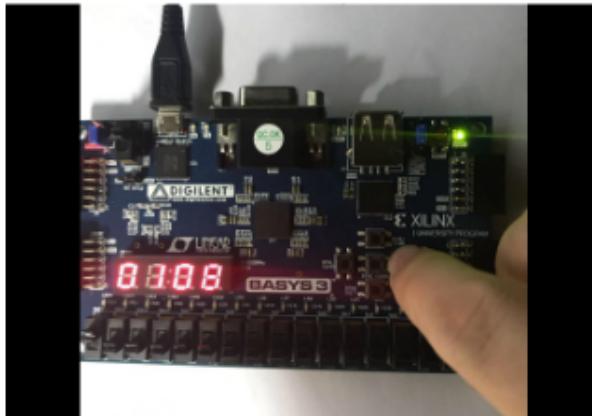
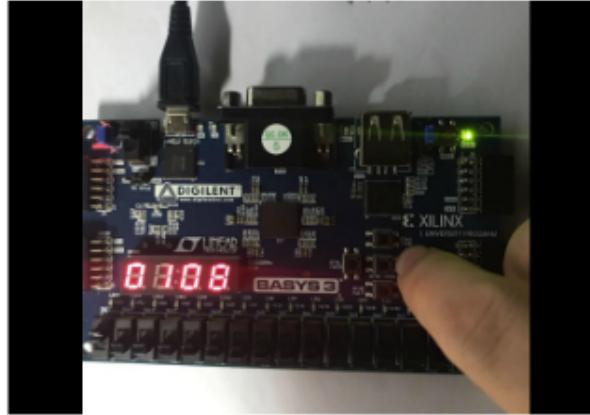
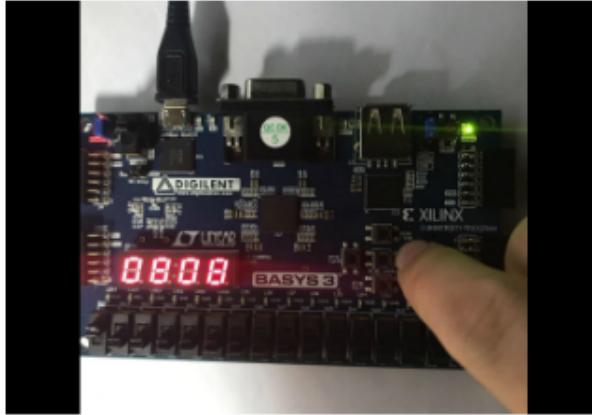


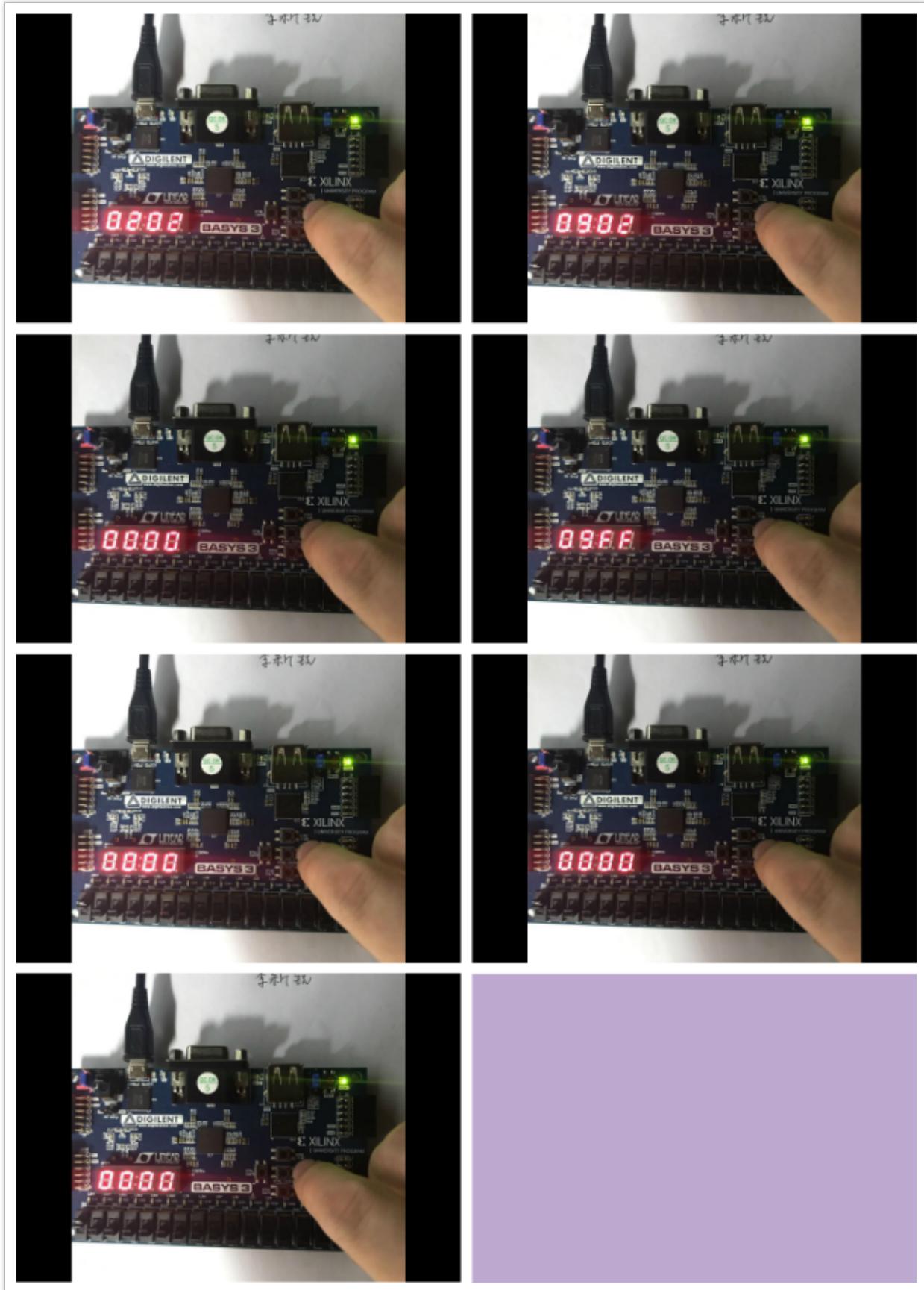




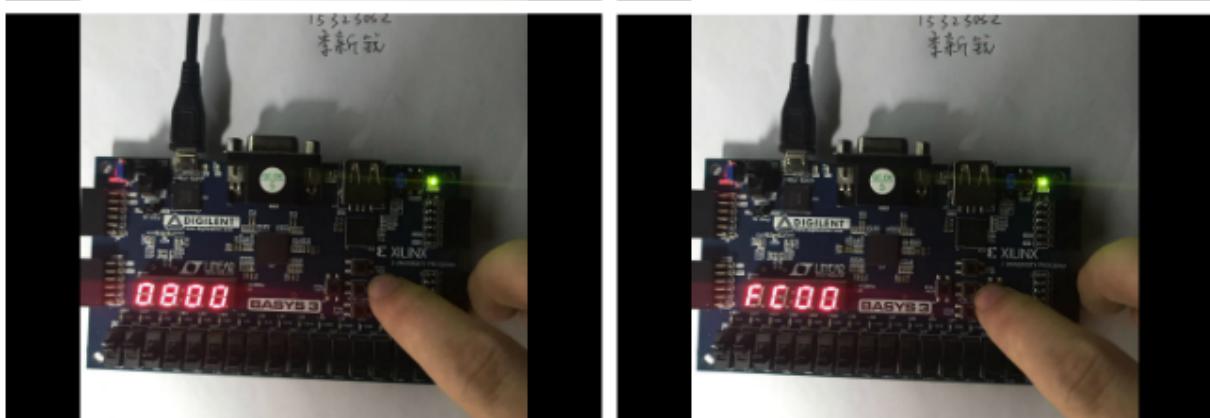
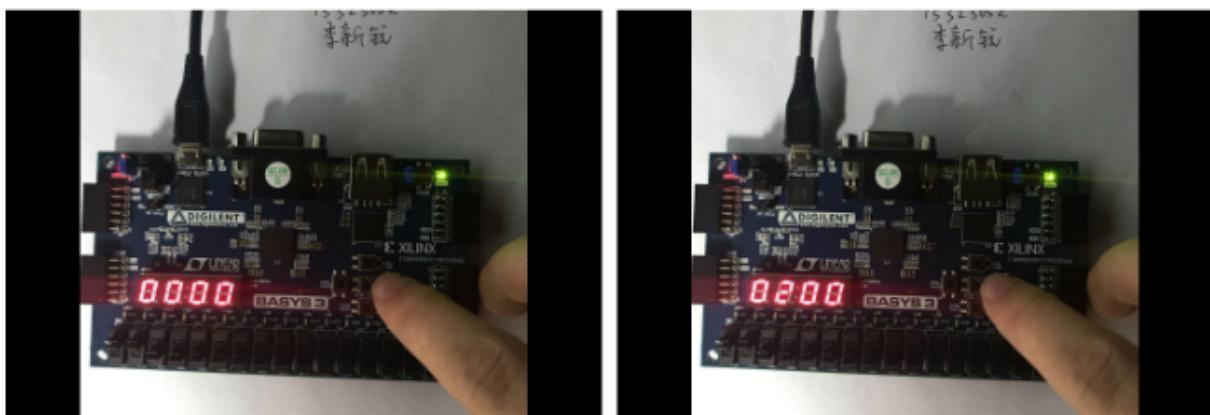
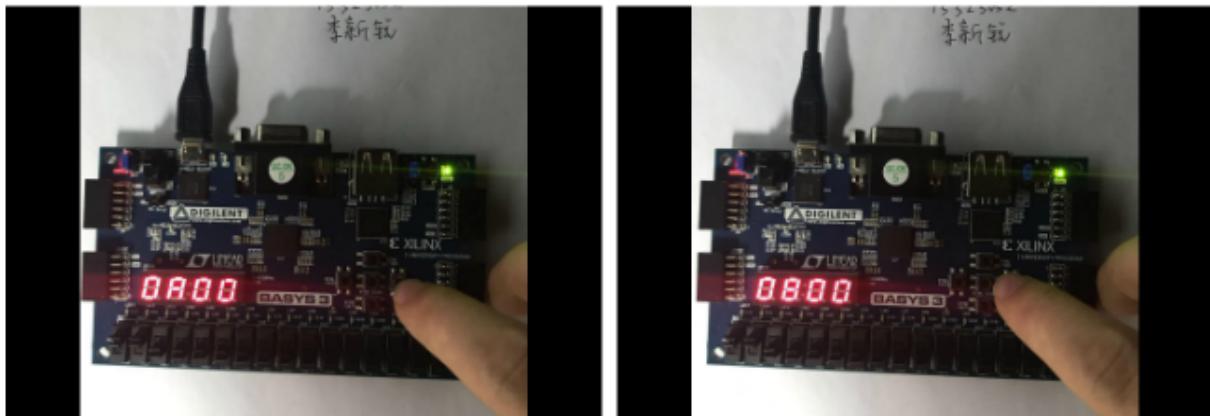
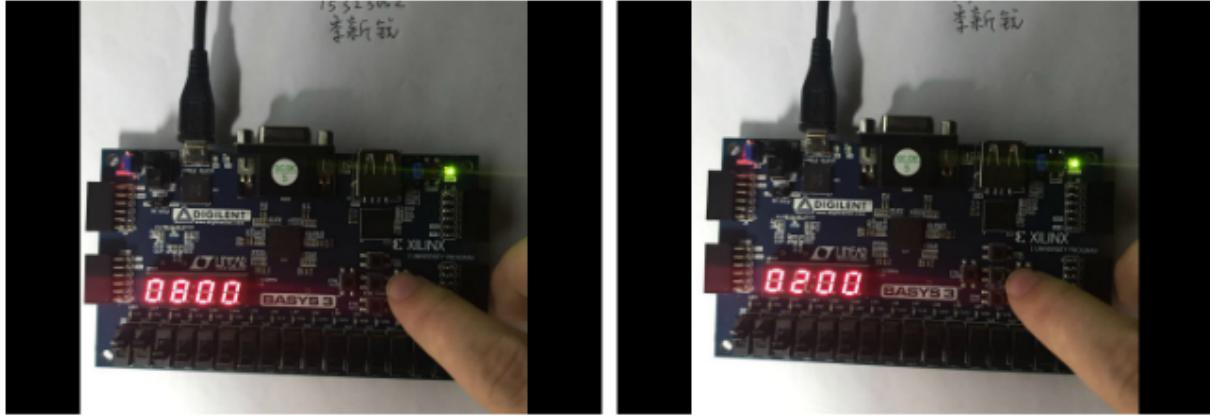
3.验证Rt——Rt Value正确性

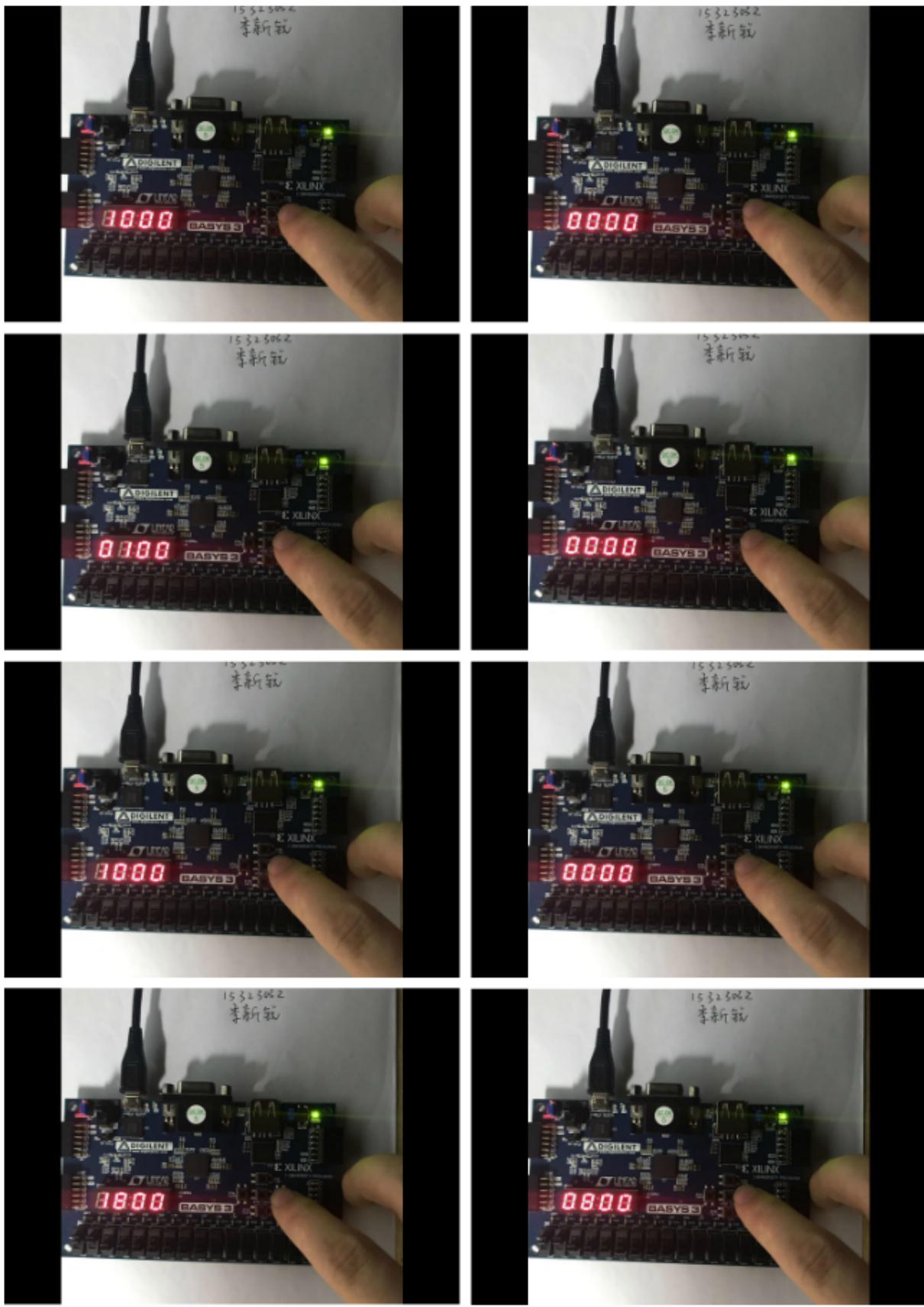






4. 验证ALU——Data Bus正确性







六. 心得体会

本次CPU设计实验，我的感想颇多：

一：自顶向下的设计思想极大地节省了我完成实验的时间，以后在其他系统的设计中我也会尽可能使用

本次实验我共花了4天的空闲时间，大概30个小时的时间完成。其中后六个小时是在解决一个硬件“玄学”问题（后面将讲述），再减去配置环境花的时间，大概用了20小时左右完成CPU的设计。其中写具体实现的时间大概只有一个晚上，4小时左右。而在上学期数字电路逻辑设计的课程中，我的课程设计是一个计算器程序，相对于CPU来说实在太简单了，然而我花的时间却是几倍于这次CPU设计的，一来是缺乏经验，二来是没有采用自顶向下的设计模式，在没有确定整个系统的结构的情况下忙着写某个原件的具体实现，之后写下一个原件时又发现上一个原件的功能或输出不符合要求，于是又要回去重新修改。同时因为写了实现，所以是进入了implementation步骤了，该步骤往往产生大量和底层逻辑原件有关的警告信息，和那些对我们意义更大的设计上的警告信息混在一起，让前者无法便是出来，失去了提醒设计上的失误的意义。

二：vivado的警告信息很有用

如第一点中所讲，在上学期的设计中我基本忽略了警告信息，而在本次设计中，我在顶层设计的阶段认真查看了每一条warning信息，发现了很多自己设计上的错误，比如接口之间数据位数不一致，连漏了线等，并在Synthesis阶段就加以改正，而不至于烧到板子上出现了错误结果才加以改正，也节省了我大量的时间。

三：在硬件设计中，一切信号的上升下降并不是理想的，务必谨慎。在15号晚上，我成功将做好的CPU烧入了开发板，然而遇到了0x00000018和0x0000001c之间陷入死循环的问题，一番调试后发现是由于在清零信号Clear变为0，CPU开始执行第一条指令后，ALU的结果8并没有送到rt（1号寄存器），导致后面一系列运算结果错误。然而在仿真中这并没有问题，因为如下所示，在Clear变为0时，它的下降沿信号会触发Regfile中的always块（参考以下代码），而此时Clear为0，就会进行送数。

```
//Regfile, 无法正常工作的原始版本
always @ (negedge CLK or negedge Clear)
begin
    if(Clear)
        begin
            for(i = 1; i < 32; i = i + 1) impl_reg[i] <= 0;
        end
    else if(RegWrite_in == 1 && Write_Reg != 0)
        impl_reg[Write_Reg] <= Write_Data;
end
```

会不会是RegWrite信号或者Rt地址出错了？废了一番功夫把这两个信号接到显示管上，然而显示出来都是对的。于是我百思不得其解。这时我想起来了之前的一条错误信息，含义大概是：按键W19（清零按键）不是时钟敏感的，如果要将它作为敏感信号，要在引脚约束文件里面手动加入一条语句……来将这个错误降级为警告。我恍然大悟，那如果我对它进行消抖处理，让清零信号实际由和时钟相关的模块参数不就好了。满心欣喜地这样改了，然后结果还是错的，无论怎么按清零，ALU都不会在松开手后送数。然后时间过去了很久，我终于意识到，会不会是Clear下降沿触发always块时，执行的并不是送数的那条指令？原因是下降沿发生时Clear的值还不稳定，if(Clear)判断为true，又执行了清零。同时我又想，在一次Clear信号到来之后，PC的值为0，是一个有效的指令，那么Clear作为它的触发者应该引发送数操作。也就是说无论是清零之后也要送数。因此我写出了这样的代码：

```
always @ (negedge CLK or negedge Clear)
begin
    if(Clear)
        begin
            for(i = 1; i < 32; i = i + 1) impl_reg[i] <= 0;
        end
    if(RegWrite_in == 1 && Write_Reg != 0) //将else if改为了if
        impl_reg[Write_Reg] <= Write_Data;
end
```

然而这样的代码在Verilog中却是会引发错误的，因此最终代码写为。

```
always @ (negedge CLK or negedge Clear)
begin
    if(Clear)
        begin
            for(i = 1; i < 32; i = i + 1) impl_reg[i] <= 0;
            if(RegWrite_in == 1 && Write_Reg != 0)
                impl_reg[Write_Reg] <= Write_Data;
        end
    else if(RegWrite_in == 1 && Write_Reg != 0)
        impl_reg[Write_Reg] <= Write_Data;
end
```

最终问题成功解决。