单周期 CPU 设计
实验报告

姓名：张镓伟
学号：15352408
学院：数据科学与计算机学院
专业：软件工程(移动信息工程)
班级：15 级 18 班
Email：70907544@qq.com
电话：13531810182
指导老师：郭雪梅
助教：李声涛、王绍菊
完成时间：2017 年 6 月 16 日

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. 实验任务

利用 verilog 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 10-16 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

这次我设计的 CPU 支持 22 条指令，具体见第四大点实验设计。

MIPS 指令集的指令格式

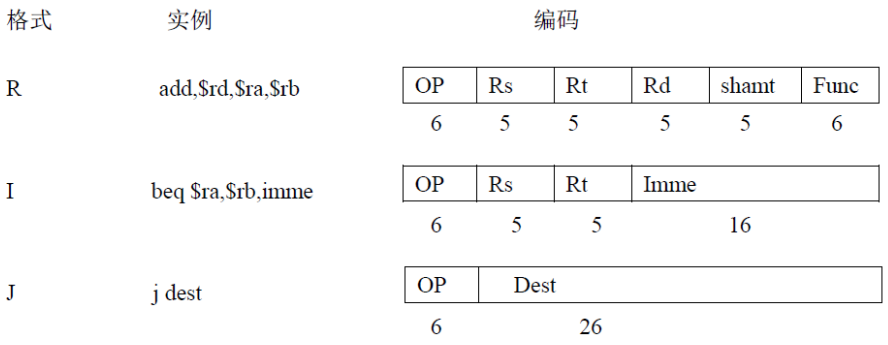


图 1 MIPS32™ 指令格式

其中Rs和Rt为两个源操作数寄存器，Rd为目的操作数寄存器。shamt为移位操作时的移位运算值，是一个立即数。Func为R型指令的功能码。imme为I型指令的立即数。Dest为J型指令的跳转地址。

设计所需要支持的指令集如下所示：表 1 为指令的格式编码。

表 1 指令的编码格式

Addi rt,rs,imme	Rt<-Rs+imme	I	001000	n/a
Addiu rt,rs,imme	Rt<-Rs+imme(无溢出判断)	I	001001	n/a
SLTI rt,rs,imme	Rt<-(Rs<imme)	I	001010	n/a
Andi rt,rs,imme	Rt<-Rs and imme	I	001100	n/a
ORI rt,rs,imme	Rt<-Rs OR imme	I	001101	n/a
XORI rt,rs,imme	Rt<-Rs XOR imme	I	001110	n/a
LUI rt,imme	Rt<-(imme<<16)&0xffff0000	I	001111	n/a
LW rt,offset(rs)	Rt<-Mem(offset+Rs)	I	100011	n/a
SW rt,offset(rs)	Mem(offset+rs) <-Rt	I	101011	n/a

指令	功能	编码	OP	Func
SLL rd,rt,shamt	Rd<-Rt<<shamt	R	000000	000000
SRL rd,rt,shamt	Rd<-rt>>shamt	R	000000	000010
SRA rd,rt,rs	Rd<-rt>>shamt(arithmetic)	R	000000	000011
SLLV rd,rt,rd	Rd<-rt<<rs	R	000000	000100
SRAV rd,rt,rs	Rd<-rt>>rs(arithmetic)	R	000000	000111
Jr rs	PC<-rs	R	000000	001000
Add rd,rs,rt	Rd<-Rs+Rt	R	000000	100000
Addu rd,rs,rt	Rd<-Rs+Rt(无溢出判断)	R	000000	100001
SUB rd,rs,rt	Rd<-Rs-Rt	R	000000	100010
SUBU rd,rs,rt	Rd<-Rs-Rt(无溢出判断)	R	000000	100011
XOR rd,rs,rt	Rd<-Rs XOR Rt	R	000000	100110
And rd,rs,rt	Rd<-Rs and Rt	R	000000	100100
OR rd,rs,rt	Rd<-Rs OR Rt	R	000000	100101
NOR rd,rs,rt	Rd<-Rs NOR Rt	R	000000	100111
SLT rd,rs,rt	Rd<-(Rs<Rt)	R	000000	101010
Bltz rs,offset	If Rs<0 then branch	I	000001	n/a
J target	Branch to target	J	000010	n/a
Beq rs,rt,offset	If Rs=Rt then branch	I	000100	n/a
Bgtz rs,offset	If Rs>0 then branch	I	000111	n/a

该指令系统中有4种寻址方式：
立即数寻址方式：比如I型指令使用16位立即数，即直接将16位二进制数作为操作数。
相对寻址：操作数是下一条指令的PC值加上一个32位偏移量，主要用于条件转移指令。
寄存器寻址：操作数是存放在寄存器中，指令里放的是寄存器号。
寄存器相对寻址：操作数存放在存储器中，其有效地址有两部分组成，基地址存放在一个寄存器中，偏移地址部分为一个16位的立即数。

CPU的设计、仿真与测试
该部分是处理器的总体设计，确定处理器由哪些部分组成：运算器(ALU)，寄存器(Reg)，控制单元(CU)；定义各个期间的控制信号以及控制方法。处理器的设计要结合存储器(Mem)单元进行设计，确定控制单元对存储器的控制信号及控制方法。
当处理器设计完成后进行测试仿真。

三. 实验原理

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。在 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，还可以通过组合逻辑解决。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 1。

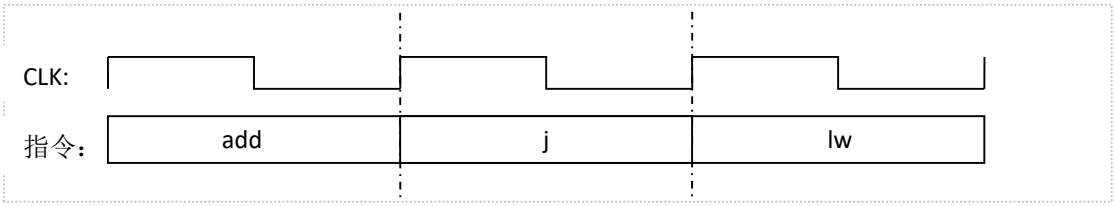


图 1 单时钟周期 CPU 时序示意图

CPU 在处理指令时，一般需要经过以下几个步骤：

(1) 取指令(IF)：根据程序计数器 PC 中的指令地址，从存储器中取出一条指令，同时，PC 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 PC，当然得到的“地址”需要做些变换才送入 PC。

(2) 指令译码(ID)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

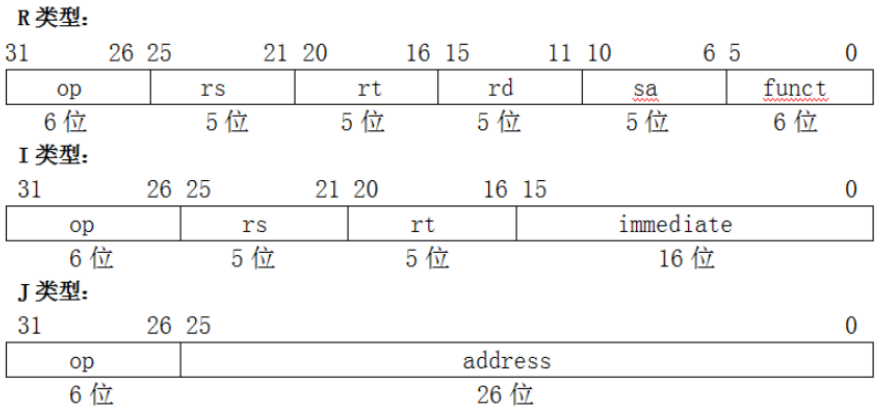
(3) 指令执行(EXE)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

(4) 存储器访问(MEM)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得
到数据地址单元中的数据。

(5) 结果写回(WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

MIPS 指令的三种格式：



其中，

op： 为操作码；

rs： 为第 1 个源操作数寄存器，寄存器地址（编号）是 00000~11111， 00~1F；

rt： 为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd： 为目的操作数寄存器，寄存器地址（同上）；

sa： 为位移量（shift amt），移位指令用于指定移多少位；

func： 为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能（reserved 为预留部分，即未用，一般填“0”）；

immediate： 为 16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载（Laod） /数据保存（Store）指令的数据地址字节偏移量和分支指令中相对程序计数

器（PC）的有符号偏移量；

address： 为地址。

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：

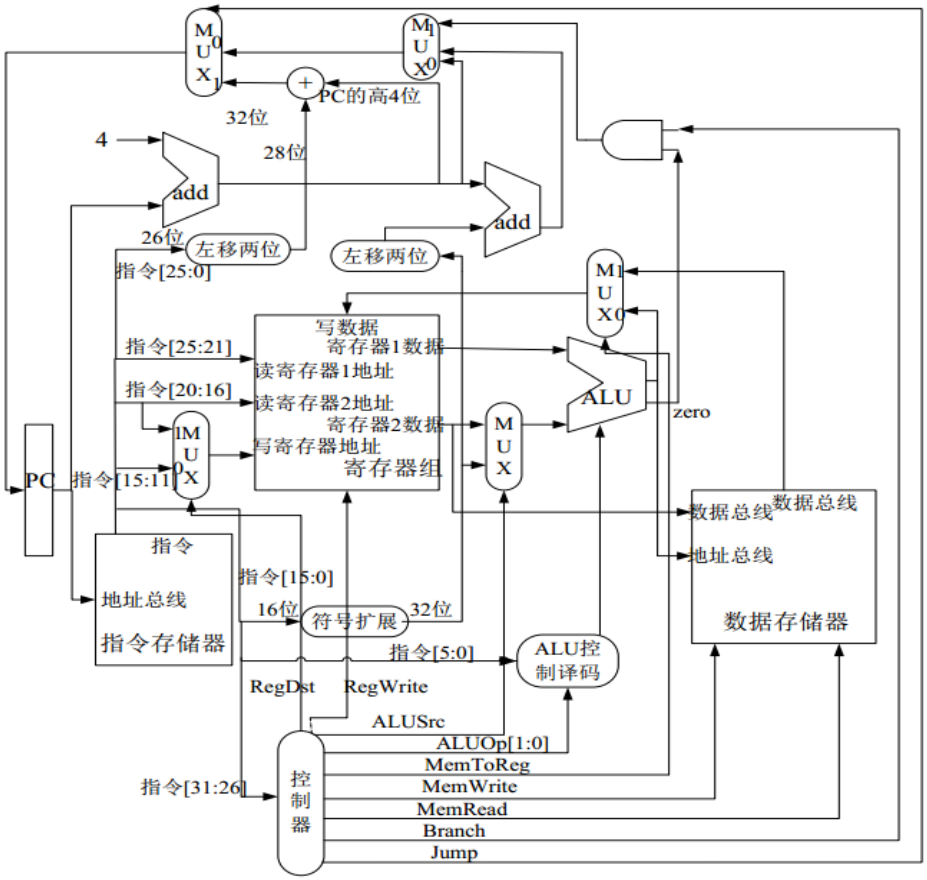


图 2 单时钟周期 CPU 详细逻辑设计图

四. 实验设计

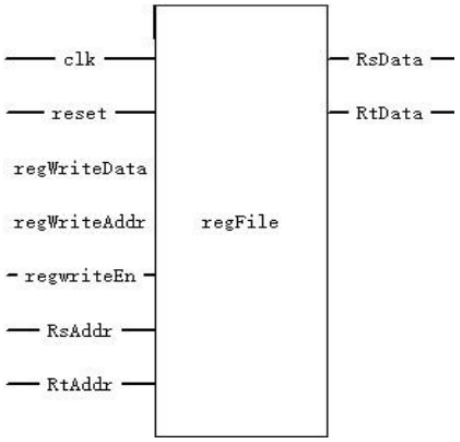
1.Fetch（取指单元）

A.定义指令 ROM 存储器
B.到程序 ROM 中取指令
C.对 PC 值进行＋4 处理
D.完成各种跳转指令的 PC 修改功能
E.在有中断的情况下处理中断到来时的 PC 修改

2. 寄存器组

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， Rd 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：

input	clk
input	reset
input	RsAddr[4:0]
input	RtAddr[4:0]
input	regWriteData[31:0]
input	regWriteAddr[4:0]
input	regWriteEn
output	RsData[31:0]
output	RtData[31:0]



这里 32 个寄存器需要注意的一点是，0 号寄存器根据其使用定义需一直保持值为 0，为了保证这一特点，我们在代码中对 0 号寄存器进行特判，如果是 0 号寄存器就输出 0，否则就输出对应寄存器储存的数据。代码如下：

```
module regFile(
    input clk,
    input reset,
    input [31:0] regWriteData,
    input [4:0] regWriteAddr,
    input regWriteEn,
    output [31:0] RsData,
    output [31:0] RtData,
    input [4:0] RsAddr,
    input [4:0] RtAddr );
    reg[31:0] regs[0:31];
    //地址是 0 号寄存器则输出 0，否则输出对应数据
    assign RsData=(RsAddr==5'b0)?32'b0:regs[RsAddr];
    assign RtData=(RtAddr==5'b0)?32'b0:regs[RtAddr];
```

```
    integer i;
    always@(posedge clk)
    begin
        if(!reset) begin
            if(regWriteEn==1)begin//在时钟上升沿写数据
                regs[regWriteAddr]=regWriteData;
            end
        end
        else begin
            for(i=0;i<32;i=i+1)
                regs[i]=0;
        end
    end
endmodule
```

3. 符号扩展模块

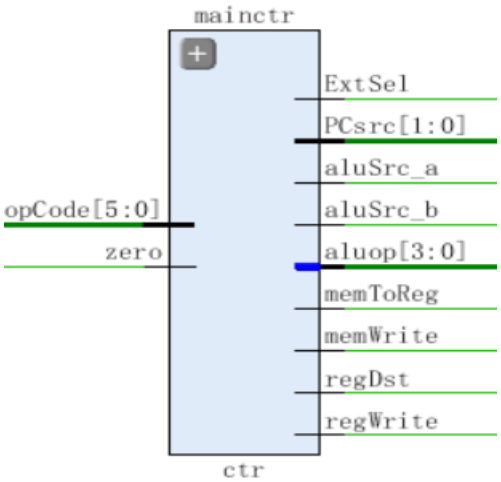
符号扩展模块是将 16 位二进制数扩展为 32 位的二进制数，分为有符号扩展和无符号扩展，由 ExtSel 控制，若 Extsel=1 则是有符号扩展，需要在其前面补足 16 位符号；若 Extsel=0 则是无符号扩展，需要在前面补足 16 位 0。代码如下：

```
module signext(
    input ExtSel,
    input [15:0] inst,
    output [31:0] data
);
    assign
data=ExtSel==1?(inst[15:15]==1?{16'hffff,inst}:{16'h0000,inst}):({16'h0000,inst});

endmodule
```

4. 控制模块

控制器模块输入为指令的操作码 opCode 段，输出各个复用器、存储器读写等的信号，控制数据通路的正常进行。根据指令的不同，输出不同的信号即可。具体如下表：



各管脚功能如下表：

Input：

OpCode[5:0]	由实验原理可知其用来决定指令类型(R、I、J)
zero	此信号是 alu 的计算结果，由于 beq 和 bne 这两个条件跳转指令是根据 rs 和 rt 两个寄存器中的值相减是否为 0 来决定是否跳转的，所以使用 zero 信号去确定 PC 的跳转方式。

Output：

ExtSel:	符号位的扩展信号，为 0 表示无符号扩展，为 1 表示有符号扩展。
PCsrc	用来选择接下来的 PC 的跳转方向,共有 4 种情况： PCsrc=00: 跳转到 PC+4； PCsrc=01: 跳转到相对地址，需要符号扩展； PCsrc=10: 跳转到绝对地址，跳转的地址是由 PC[31:28]+immediate+00;(j 指令) PCsrc=11: 以寄存器里的值进行跳转；
aluSrc_a	表示 alu 运算传入的第一个参数是来自 rs 寄存器还是立即数，为 0 表示来自 rs 寄存器，为 1 表示来自立即数（无符号扩展后的，在 sll，srl 两条指令会用到）
aluSrc_b	表示 alu 运算传入的第二个参数是来自 rt 寄存器还是立即数，为 0 表示来自 rt 寄存器，为 1 表示来自立即数（无符号扩展后的，在 j 指令中需要用到）
aluop	根据 OpCode 来确定对应的 alu 操作。 1.对于 R 型指令还需要在 alu 控制模块中根据 funct 功能码一起来决定具体的 alu 操作，所以这里对 R 型指令统一赋值 aluop= 0010 2.对于不同的 I 型或 J 型指令就赋一个值(这个值自己随便设定就好)，之后由 alu 控制模块会根据这个值确定 alu 的具体操作
memToReg	选择写入寄存器的值来自数据存储器 RAM 还是来自 ALU 运算结果。为 0 表示来自 ALU 运算结果，为 1 表示来自 RAM
memWrite	选择是否写入数据存储器 RAM 中，为 0 表示不写，为 1 表示写。
regDst	选择写进寄存器的时候写的是 rs 寄存器还是 rd 寄存器。为 0 表示 rs 寄存器，为 1 表示 rd 寄存器；RegRt 与 RegDst 相反，为 1 表示 rs 寄存器，为 0 表示 rd 寄存器。
memRead	是否读数据存取器 RAM 里的内容，为 1 表示需要读取内容(lw)，为 0 表示不需要读。
regWrite	控制是否需要写入寄存器，为 1 表示需要写入寄存器，为 0 表示不需要写入寄存器

22 条指令对应的控制信号值：

信号 指令	ExtSel	PCsrc	aluSrc_a	aluSrc_b	aluop	memToReg	memWrite	regDst	memRead	regWrite
R 型指令 9 条： add sub	0	00	0	0	0010	0	0	1	0	1

and or xor nor slt sll srl										
I型指令 12条	ExtSel	PCsrc	aluSrc_a	aluSrc_b	aluop	memToReg	memWrite	regDst	memRead	regWrite
addi	1	00	0	1	0000	0	0	0	0	1
addiu	0	00	0	1	0000	0	0	0	0	1
andi	0	00	0	1	0100	0	0	0	0	1
ori	0	00	0	1	0101	0	0	0	0	1
xori	0	00	0	1	0110	0	0	0	0	1
lui	0	00	0	1	0011	0	0	0	0	1
lw	1	00	0	1	0000	1	0	0	1	1
sw	1	00	0	1	0000	0	1	0	0	0
slti	1	00	0	1	0111	0	0	0	0	1
sltiu	0	00	0	1	1000	0	0	0	0	1
beq	1	zero=1:PCsrc=01; zero=0:PCsrc=00	0	0	0001	0	0	0	0	0
bne	1	zero=1:PCsrc=00; zero=0:PCsrc=01	0	0	0001	0	0	0	0	0
J型指令	ExtSel	PCsrc	aluSrc_a	aluSrc_b	aluop	memToReg	memWrite	regDst	memRead	regWrite
j	0	10	0	0	0000	0	0	0	0	0

代码如下：

```
module ctr(
    input [5:0] opCode,
    input zero,
    output reg ExtSel,
    output reg regDst,
    output reg regRt,
    output reg aluSrc_a,
    output reg aluSrc_b,
    output reg memToReg,
                aluSrc_a=0;
                aluSrc_b=0;
                memToReg=0;
                regWrite=0;
                memRead=0;
                memWrite=0;
                aluop=4'b0001;
                ExtSel=1;
                if(zero==1)
                    aluop=4'b0100;
                    ExtSel=0;
                    PCsrc=2'b00;
                end

            6'b001101://ori
                begin
                    regDst=0;
                    regRt=1;
```

```
    output reg regWrite,
    output reg memRead,
    output reg memWrite,
    output reg[3:0] aluop,
    output reg[1:0] PCsrc
);
    always@(opCode)
    begin
        case(opCode)
        6'b000010: //J型
            begin
                regDst=0;
                regRt=0;
                aluSrc_a=0;
                aluSrc_b=0;
                memToReg=0;
                regWrite=0;
                memRead=0;
                memWrite=0;
                aluop=4'b0000;
                ExtSel=0;
                PCsrc=2'b10;
            end
        6'b000000://R型
            begin
                regDst=1;
                regRt=0;
                aluSrc_a=0;
                aluSrc_b=0;
                memToReg=0;
                regWrite=1;
                memRead=0;
                memWrite=0;
                aluop=4'b0010;
                ExtSel=0;
                PCsrc=2'b00;
            end
        6'b100011://lw
            begin
                regDst=0;
                regRt=1;
                aluSrc_a=0;
                aluSrc_b=1;
                memToReg=1;
PCsrc=2'b01;
                else
PCsrc=2'b00;
            end
        6'b000101://bne
                begin
                    regDst=0;
                    regRt=0;
                    aluSrc_a=0;
                    aluSrc_b=0;
                    memToReg=0;
                    regWrite=0;
                    memRead=0;
                    memWrite=0;

aluop=4'b0001;
                    ExtSel=1;
                    if(zero==1)
PCsrc=2'b00;
                    else
PCsrc=2'b01;
                end
        6'b001000://addi
            begin
                regDst=0;
                regRt=1;
                aluSrc_a=0;
                aluSrc_b=1;
                memToReg=0;
                regWrite=1;
                memRead=0;
                memWrite=0;
                aluop=4'b0000;
                ExtSel=1;
                PCsrc=2'b00;
            end
        6'b001001://addiu
                begin
                    regDst=0;
                    regRt=1;
                    aluSrc_a=0;
                    aluSrc_b=1;
                    memToReg=0;
                    regWrite=1;
                    aluSrc_a=0;
                    aluSrc_b=1;
                    memToReg=0;
                    regWrite=1;
                    memRead=0;
                    memWrite=0;
                    aluop=4'b0101;
                    ExtSel=0;
                    PCsrc=2'b00;
            end
        6'b001110: //xori
                begin
                    regDst=0;
                    regRt=1;
                    aluSrc_a=0;
                    aluSrc_b=1;
                    memToReg=0;
                    regWrite=1;
                    memRead=0;
                    memWrite=0;

aluop=4'b0110;
                    ExtSel=0;
                    PCsrc=2'b00;
                end
            6'b001010://slti
                begin
                    regDst=0;
                    regRt=1;
                    aluSrc_a=0;
                    aluSrc_b=1;
                    memToReg=0;
                    regWrite=1;
                    memRead=0;
                    memWrite=0;

aluop=4'b0111;
                    ExtSel=1;
                    PCsrc=2'b00;
                end
            6'b001011: //sltiu
                begin
                    regDst=0;
                    regRt=1;
```

```
            regWrite=1;
            memRead=1;
            memWrite=0;
            aluop=4'b0000;
            ExtSel=1;
            PCsrc=2'b00;
        end
    6'b101011://sw
        begin
            regDst=0;
            regRt=1;
            aluSrc_a=0;
            aluSrc_b=1;
            memToReg=0;
            regWrite=0;
            memRead=0;
            memWrite=1;
            aluop=4'b0000;
            ExtSel=1;
            PCsrc=2'b00;
        end
    6'b000100://beq
        begin
            regDst=0;
            regRt=0;
```

```
                    memRead=0;
                    memWrite=0;
aluop=4'b0000;
                    ExtSel=0;
                    PCsrc=2'b00;
                end
            6'b001111://lui
                begin
                    regDst=0;
                    regRt=1;
                    aluSrc_a=0;
                    aluSrc_b=1;
                    memToReg=0;
                    regWrite=1;
                    memRead=0;
                    memWrite=0;
                    aluop=4'b0011;
                    ExtSel=0;
                    PCsrc=2'b00;
                end
            6'b001100: //andi
                begin
                    regDst=0;
                    regRt=1;
                    aluSrc_a=0;
                    aluSrc_b=1;
                    memToReg=0;
                    regWrite=1;
                    memRead=0;
                    memWrite=0;
```

```
                    aluSrc_a=0;
                    aluSrc_b=1;
                    memToReg=0;
                    regWrite=1;
                    memRead=0;
                    memWrite=0;
aluop=4'b1000;
                    ExtSel=0;
PCsrc=2'b00;
                end
            default:
                begin
                    regDst=0;
                    regRt=0;
                    aluSrc_a=0;
                    aluSrc_b=0;
                    memToReg=0;
                    regWrite=0;
                    memRead=0;
                    memWrite=0;
                    aluop=4'b0000;
                    ExtSel=1;
                    PCsrc=2'b00;
                end
            endcase
        end
endmodule
```

5. ALU 控制译码模块 aluctr

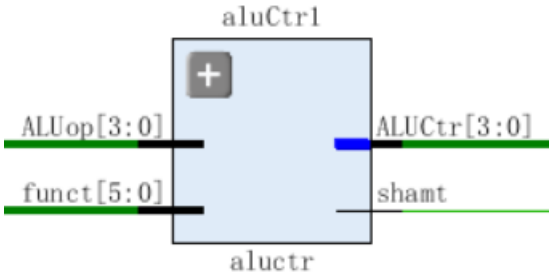
ALU 主要执行 10 种操作：与，或，加，减，小于设置，异或，或非，左移，逻辑右移，lui 左移。这 10 种操作可以使用四位的编码表示：0000，0001，0010，01110，0111，1001,1100,0011,0100，1000。指令不同，则对应的 ALU 运算不同，所以该模块需要根据指令来控制 ALU 进行正确的运算。

由于 lw， sw， addi 指令均要求 ALU 执行加操作，则可分为一类，将 aluop 编码 0000；

由于 beq、bne 指令要求 ALU 执行减操作，则分为一类，编码 0001；

最后一类是 R 型指令，可以编码为 10；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 funct 进一步确定 ALU 的运算。此外，对于 R 型指令中的 sll,srl,因为我的左移右移都是在 alu 中进行的,则还要增加一个控制信号 shamt 来选择立即数作为 alu 里的操作数， 不然的话默认所有 R 型指令的操作数都取自寄存器。

最终该模块即实现 4 位操作码以及 6 位功能码输出 4 位 ALU 控制信号码。



各指令功能及对应编码如下表：

R 型指令	aluop	aluctr	功能
add	0010	0010	rd <- rs + rt
sub	0010	0110	rd <- rs - rt
and	0010	0000	rd <- rs & rt
or	0010	0001	rd <- rs | rt
xor	0010	1001	rd <- rs xor rt
nor	0010	1100	rd <- not(rs | rt)
slt	0010	0111	if (rs < rt) rd=1 else rd=0 ；
sll	0010	0011	rd <- rt << 立即数
srl	0010	0100	rd <- rt >> 立即数 逻辑右移
I 型指令	**aluop**	**aluctr**	**功能**
andi	0100	0000	rt <- rs & (zero-extend)immediate
ori	0101	0001	rt <- rs | (zero-extend)immediate
xori	0110	1001	rt <- rs xor (zero-extend)immediate
slti	0111	0111	if (rs <(sign-extend)immediate) rt=1 else rt=0 ；
sltiu	1000	0111	if (rs <(zero-extend)immediate) rt=1 else rt=0 ； 其中 rs＝$2，rt=$1
lw, sw, addi，addiu	0000	0010	加
beq, bne	0001	0110	减
lui	0011	1000	rt<immediate*65536；左移 16 位

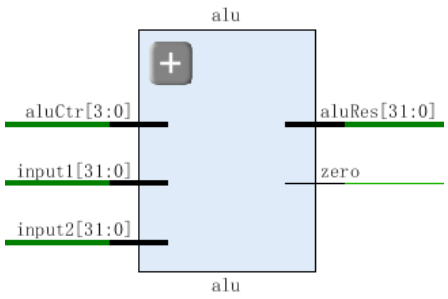
代码如下：

```
module aluctr(
    input[3:0] ALUop,
    input[5:0] funct,
```

```
    output reg[3:0] ALUCtr,
    output reg shamt
);
    always@(ALUop or funct)
    case(ALUop)
    4'b0000:begin ALUCtr=4'b0010;shamt=0;end//addi, lw,sw
    4'b0001:begin ALUCtr=4'b0110;shamt=0;end//beq,bne
    4'b0010:
    begin
        case(funct)
        6'b100000:begin ALUCtr=4'b0010;shamt=0;end//add
        6'b100010:begin ALUCtr=4'b0110;shamt=0;end//sub
        6'b100100:begin ALUCtr=4'b0000;shamt=0;end//and
        6'b100101:begin ALUCtr=4'b0001;shamt=0;end//or
        6'b101010:begin ALUCtr=4'b0111;shamt=0;end//slt
        6'b000000:begin ALUCtr=4'b0011;shamt=1;end//sll
        6'b000010:begin ALUCtr=4'b0100;shamt=1;end//srl
        6'b100111:begin ALUCtr=4'b1100;shamt=0;end//nor
        6'b100110:begin ALUCtr=4'b1001;shamt=0;end//xor
        default:begin ALUCtr=4'b0000;shamt=0;end
        endcase
    end
    4'b0011:begin ALUCtr=4'b1000;shamt=0; end //lui
    4'b0100:begin ALUCtr=4'b0000;shamt=0; end //andi
    4'b0101:begin ALUCtr=4'b0001;shamt=0; end //ori
    4'b0110:begin ALUCtr=4'b1001;shamt=0; end //xori
    4'b0111:begin ALUCtr=4'b0111;shamt=0; end //slti
    4'b1000:begin ALUCtr=4'b0111;shamt=0; end //sltiu
    default:begin ALUCtr=4'b0000;shamt=0; end
    endcase
endmodule
```

6. alu 计算模块

ALU 模块主要接受两个操作数，完成各类运算操作。 包括与，或，加，减，小于设置，异或，或非，左移，逻辑右移，lui。需要输入 ALU 控制信号判断 ALU 进行的操作，再进一步对两个操作数进行操作，操作完输出结果(aluRes)。



ALU 控制信号与对应操作表：

aluCtr	功能
0010	加
0110	减
0000	与
0001	或
0111	小于置 1
0011	左移
0100	逻辑右移
1000	lui
1100	或非
1001	异或

代码如下：

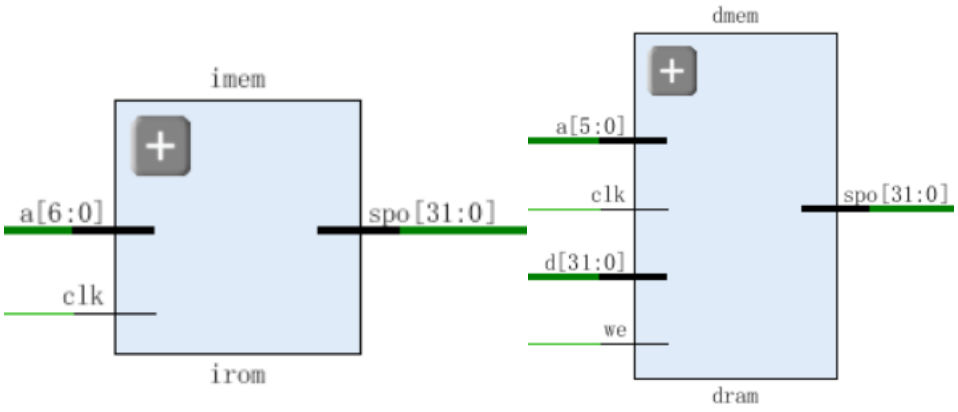
```
module alu(
    input [31:0] input1,
    input [31:0] input2,
    input [3:0]  aluCtr,
    output reg[31:0] aluRes,
    output reg zero
);
    always@(input1 or input2 or aluCtr)
    begin
        case(aluCtr)
        4'b0110: //sub,beq,bne
        begin
            aluRes=input1-input2;
            if(aluRes==0) zero=1;
            else zero=0;
        end
        4'b0010:  //add
            begin
                aluRes=input1+input2;
                if(aluRes==0) zero=1;
                else zero=0;
            end
        4'b0000:aluRes=input1&input2;//and,andi
        4'b0001:aluRes=input1|input2;//or,ori
        4'b1100:aluRes=~(input1|input2);//nor
        4'b1001:aluRes=(input1^input2);//xor,xori
        4'b0111://slt,slti,sltiu
            begin
                if(input1<input2) aluRes=1;
                else aluRes=0;
```

```
            end
        4'b0011://sll
            begin
                aluRes=input2<<input1;
            end
        4'b0100:aluRes=input2>>input1;//srl
        4'b1000: aluRes=input2<<16; // lui
        default:
            aluRes=0;
        endcase
        if(aluRes==0) zero=1;
        else zero=0;
    end
endmodule
```

7. ROM 和 RAM

该模块由 xilinx 提供的 IP 核进行设计， 其中 ROM 模块由地址线、数据线和时钟信号线组成。 RAM 模块由时钟线，地址线，读写使能线，数据线构成。

在每个时钟的上升沿，ROM 模块（指令存储器）根据 PC[8:2]从内存中取得一条指令。RAM 模块(数据存储器)则在每个时钟的上升沿根据地址进行读写数据操作。其设计如下：



相应管脚说明：

Rom		RAM	
a[6:0]	PC[8:2]，地址输入端口	a[5:0]	数据存储器地址输入端口
clk	时钟	d[31:0]	数据存储器数据输入端口
spo[31:0]	存储数据输出端口，在这里是输出一条指令	clk	时钟
		we	读写控制信号，1 为写，0 为读
		spo[31:0]	数据存储器数据输出端口

设计方式：

在 Flow Navigator 选项卡选择 IP Catalog，在右侧的“IP Catalog”界面上 Memories & Storage Element —— RAM & ROMS —— Distributed Memory Generator 双击，进入 ip 核的设置界面，设置好数据宽度和深度，是否时钟触发，然后选择 COE 文件去例化(RAM 用与 ROM 一样的 COE 文件也可)，coe 文件内容是指令的机器码，可以先写好汇编文件再翻译，我写的汇编代码及 coe 文件内容如下：

汇编代码及对应机器码(最左侧的 8 位 16 进制数是机器码)：

```
main:
20111111        addi    $17,$0,0x1111
24112222        addiu $17,$0,0x2222
32310022        andi    $17,$17,0x22
36322200        ori     $18,$17,0x2200
3a512200        xori    $17,$18,0x2200
3c120022        lui     $18,0x0022
22510022        addi    $17,$18,0x22
ae110000        sw       $17, 0($16)
8e120000        lw       $18, 0($16)
16320010        bne $17,$18,exit
2a330001        slti $19,$17,1
2e710001        sltiu $17,$19,1
8e130000        lw    $19, 0($16)
02538820        add $17,$18,$19
02329822        sub $19,$17,$18
20120002        addi $18,$0,2
02728824        and $17,$19,$18
02329827        nor $19,$17,$18
02729825        or    $19,$19,$18
02728826        xor $17,$19,$18
8e110000        lw    $17, 0($16)
00119c02        srl    $19,$17,16
00138c00        sll    $17,$19,16
1233000e        beq $17,$19,exit
0232982a        slt    $19,$17,$18
08100000        j main
                        exit:
20110000        addi    $17,$0,0
08100000        j main
```

coe 文件内容：

```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
20111111
24112222
32310022
```

```
36322200
3a512200
3c120022
22510022
ae110000
8e120000
16320010
2a330001
2e710001
8e130000
02538820
02329822
20120002
02728824
02329827
02729825
02728826
8e110000
00119c02
00138c00
1233000e
0232982a
08100000
20110000
08100000
```

文件中前两行：

```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
```

第一行是进制的说明，第二行是机器码开始的标志，即接下去就是机器码的内容。

8. 数码管显示模块

本次实验要求我们在 Basys3 开发板上显示 alu 的计算结果 aluRes。

动态数码管显示的原理是： 每次选通其中一位， 送出这位要显示的内容， 然后一段时间后选通下一位送出对应数据，4 个数码管这样依次选通并送出相应的数据，结束后再重复进行。这样只要选通时间选取的合适，由于人眼的视觉暂留，数码管看起来就是连续显示的。在这里我们使用 sm_wei[3:0]来表示选通 4 个数码管的哪一个去显示，sm_duan[7:0]控制显示 7 段码哪些亮。代码中 sm_wei 和 wei_ctr 是同一个东西。由于我们要显示的 aluRes 是 32 位的，所以我再新增了一个 switch，控制显示的是高 16 位还是低 16 位。

代码如下：

```
module smg_ip_model(
        input clkin_,
        input [31:0]data,
```

```
        input switch,
        input reset,
        output [3:0] sm_wei,
        output [7:0] sm_duan
    );

    integer clk_cnt;
    reg clk_400Hz;
    always@(posedge clkin_)
    if(reset==0)
    begin
        if(clk_cnt==32'd100000)
        begin
            clk_cnt<=1'b0;clk_400Hz<=~clk_400Hz;
        end
        else
        begin
            clk_cnt<=clk_cnt+1'b1;
        end
    end
    else
    begin
        clk_cnt<=1'b0;clk_400Hz<=1'b0;
    end
    reg[3:0] wei_ctr=4'b1110;
    always@(posedge clk_400Hz)
        wei_ctr<={wei_ctr[2:0],wei_ctr[3]};

    reg [3:0]duan_ctr;
    always@(wei_ctr)
    begin
        if(switch==0)
        begin
            case(wei_ctr)
            4'b1110:duan_ctr<=data[3:0];
            4'b1101:duan_ctr=data[7:4];
            4'b1011:duan_ctr=data[11:8];
            4'b0111:duan_ctr=data[15:12];
            endcase
        end
        else
        begin
            case(wei_ctr)
            4'b1110:duan_ctr=data[19:16];
```

```
                4'b1101:duan_ctr=data[23:20];
                4'b1011:duan_ctr=data[27:24];
                4'b0111:duan_ctr=data[31:28];
                endcase
            end
        end
        reg[7:0]duan;
        always@(duan_ctr)
        case(duan_ctr)
        4'h0:duan=8'b1100_0000;//0
        4'h1:duan=8'b1111_1001;//1
        4'h2:duan=8'b1010_0100;//2
        4'h3:duan=8'b1011_0000;//3
        4'h4:duan=8'b1001_1001;//4
        4'h5:duan=8'b1001_0010;//5
        4'h6:duan=8'b1000_0010;//6
        4'h7:duan=8'b1111_1000;//7
        4'h8:duan=8'b1000_0000;//8
        4'h9:duan=8'b1001_0000;//9
        4'ha:duan=8'b1000_1000;//a
        4'hb:duan=8'b1000_0011;//b
        4'hc:duan=8'b1100_0110;//c
        4'hd:duan=8'b1010_0001;//d
        4'he:duan=8'b1000_0110;//e
        4'hf:duan=8'b1000_1110;//f
        4'hf:duan=8'b1111_1111;//不显示
        default : duan = 8'b1100_0000;//0
        endcase
        assign sm_wei=wei_ctr;
        assign sm_duan=duan;
endmodule
```

9. 顶层模块

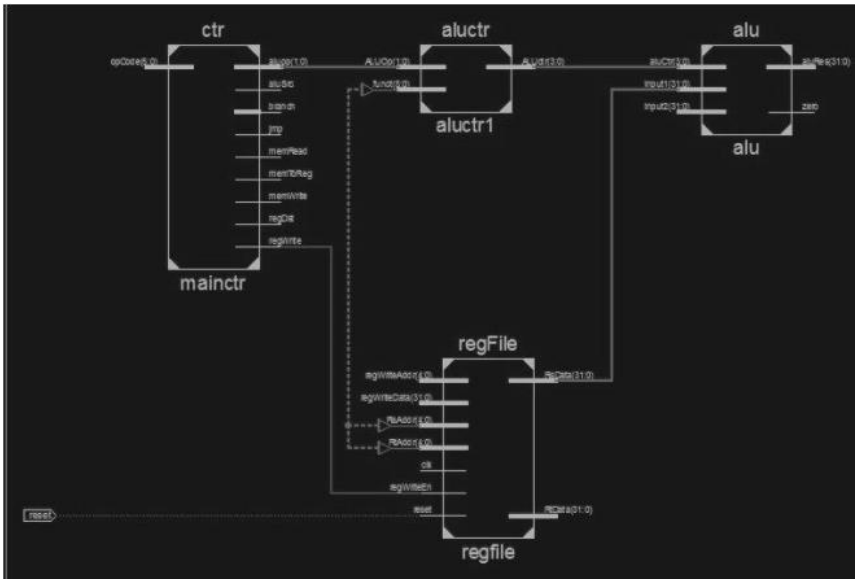
顶层模块需要将前面的多个模块实例化后，通过导线以及多路复用器将各个部件链接起来，并且在时钟的控制下修改 PC 的值， PC 是一个 32 位的寄存器，每个时钟沿自动增加 4。

多路复用器 MUX 直接通过三目运算符实现：

```
Assign  OUT=SEL? INPUT1：INPUT2；
```

其中，OUT，SEL，INPUT1，INPUT2 都是预先定义的信号

顶层模块的基本运行路线是：首先通过时钟沿的变换去改变 PC 的值，接着根据 PC 的值去取得 rom 的机器指令，然后通过各个模块的执行完成整个指令。示意图如下：



PC 的改变：每个是时钟的上升沿，PC 的值改变成 next_pc，但是 next_pc 会根据有有无跳转指令、跳转的方式而有 4 种选择，根据 PCsource[1:0]决定：

1. Pcsource=00：跳转到 PC+4，即选择 add4
2. PCsource=01：跳转到相对地址，需要符号扩展，即选择 address
3. PCsource=10：跳转到绝对地址，跳转的地址是由 PC[31:28]+immediate+00，即选择 jmpaddr
4. PCsource=11：以寄存器里的值进行跳转，即选择 RsData

```
assign expand2=expand<<2;
assign next_PC=PCsource[1]?(PCsource[0]?RsData:jmpaddr):(PCsource[0]?address:add4);
assign jmpaddr={add4[31:28],inst[25:0],2'b00};
assign address=add4+expand2;
```

其他详见代码及注释：

```
module top(
    input clkin,
    input reset,
    input switch,
    output [3:0] sm_wei,
    output [7:0] sm_duan
    );
    reg[31:0] pc;
    reg[31:0] add4;
    //复用器信号线
    wire[31:0] expand2,mux_memToReg,next_PC,address,jmpaddr,inst;
    wire[4:0] mux_regDst;
    //CPU 控制信号
    wire regDst,regRt,jmp,memRead,memWrite,memToReg;
    wire[3:0] aluop;
    wire alu_a,alu_b,regWrite;
    //alu 信号线
```

```
wire zero;
wire[31:0]aluRes;
//alu 控制信号线
wire[3:0]aluCtr;
wire shamt;
//内存信号线
wire[31:0]memReadData;
//寄存器信号线
wire[31:0]RsData,RtData;
//扩展信号线
wire[31:0] expand;

wire[31:0] mux_alu_a,mux_alu_b;
wire[1:0] PCsource;
wire [5:0] sa;

wire alu_a_a;
wire [31:0] sa_ext;
wire ExtSel;

integer clk_cnt;
reg clk_div;
always@(negedge clkin)
begin
    if(reset==0)
      begin
          if(clk_cnt==32'd100000000-1)
          //if(clk_cnt==32'd1000-1)
          begin clk_cnt<=1'b0;
                clk_div=~clk_div;
          end
          else
          clk_cnt<=clk_cnt+1'b1;
      end
      else
           begin clk_div<=0;clk_cnt<=0;end
end

always@(negedge clk_div)
begin
      if(!reset)
      begin
            pc=next_PC;
            add4=pc+4;
```

```
        end
        else
        begin
            pc=32'b0;
            add4=32'h4;
        end
    end
    //实例化控制模块
    ctr mainctr(
        .opCode(inst[31:26]),
        .zero(zero),
        .ExtSel(ExtSel),
        .regDst(regDst),
        .regRt(regRt),
        .aluSrc_a(alu_a),
        .aluSrc_b(alu_b),
        .memToReg(memToReg),
        .regWrite(regWrite),
        .memRead(memRead),
        .memWrite(memWrite),
        .aluop(aluop),
        .PCsrc(PCsource)
     );

    //实例化 alu 模块
    alu alu(
        .input1(mux_alu_a),
        .input2(mux_alu_b),
        .aluCtr(aluCtr),
        .zero(zero),
        .aluRes(aluRes)
    );
    //实例化 alu 控制模块
    aluctr aluCtr1(
        .ALUop(aluop),
        .funct(inst[5:0]),
        .ALUCtr(aluCtr),
        .shamt(shamt)
    );
    //实例化数据存储器 RAM
    dram dmem(
        .a(aluRes[7:2]),
        .d(RtData),
        .clk(!clk_div),
```

```
    .we(memWrite),
    .spo(memReadData)
);

wire[6:0] pc8_2;
assign pc8_2=pc[8:2];
//实例化指令存储器 ROM
irom imem(
  .a(pc8_2),
  .clk(clk_div),
  .spo(inst));
//实例化寄存器堆
regFile regfile(
  .RsAddr(inst[25:21]),
  .RtAddr(inst[20:16]),
  .clk(!clk_div),
  .reset(reset),
  .regWriteAddr(mux_regDst),
  .regWriteData(mux_memToReg),
  .regWriteEn(regWrite),
  .RsData(RsData),
  .RtData(RtData)
);
//实例化符号扩展模块
signext signext(
      .ExtSel(ExtSel),
      .inst(inst[15:0]),
      .data(expand)
);
//实例化数码管显示模块
smg_ip_model smg_(
    .clkin_(clkin),
    .data(aluRes),
    .switch(switch),
    .reset(reset),
    .sm_wei(sm_wei),
    .sm_duan(sm_duan)
);

// 移位指令中立即数的扩展：
assign sa=inst[10:6];
assign sa_ext={26'h000000,1'b0,sa};
//写寄存器时地址的选择，即 RS 和 RD 寄存器的选择：
assign mux_regDst=regDst?inst[15:11]:inst[20:16];
```

```
    //表示 ALU 第一个操作数来自寄存器 rs 还是来及立即数，为 0 来自寄存器，为 1 来自立即数
    assign alu_a_a=shamt?1'b1:alu_a;
    //ALU 第一个操作数的选择：
    assign mux_alu_a=alu_a_a?sa_ext:RsData;
    //ALU 第二个操作数的选择：
    assign mux_alu_b=alu_b?expand:RtData;
    //写寄存器时内容的选择，即选择来自 ALU 计算结果还是储存器：
    assign mux_memToReg=memToReg?memReadData:aluRes;
    //PC 的改变
    assign expand2=expand<<2;
    assign next_PC=PCsource[1]?(PCsource[0]?RsData:jmpaddr):(PCsource[0]?address:add4);
    assign jmpaddr={add4[31:28],inst[25:0],2'b00};
    assign address=add4+expand2;

endmodule
```

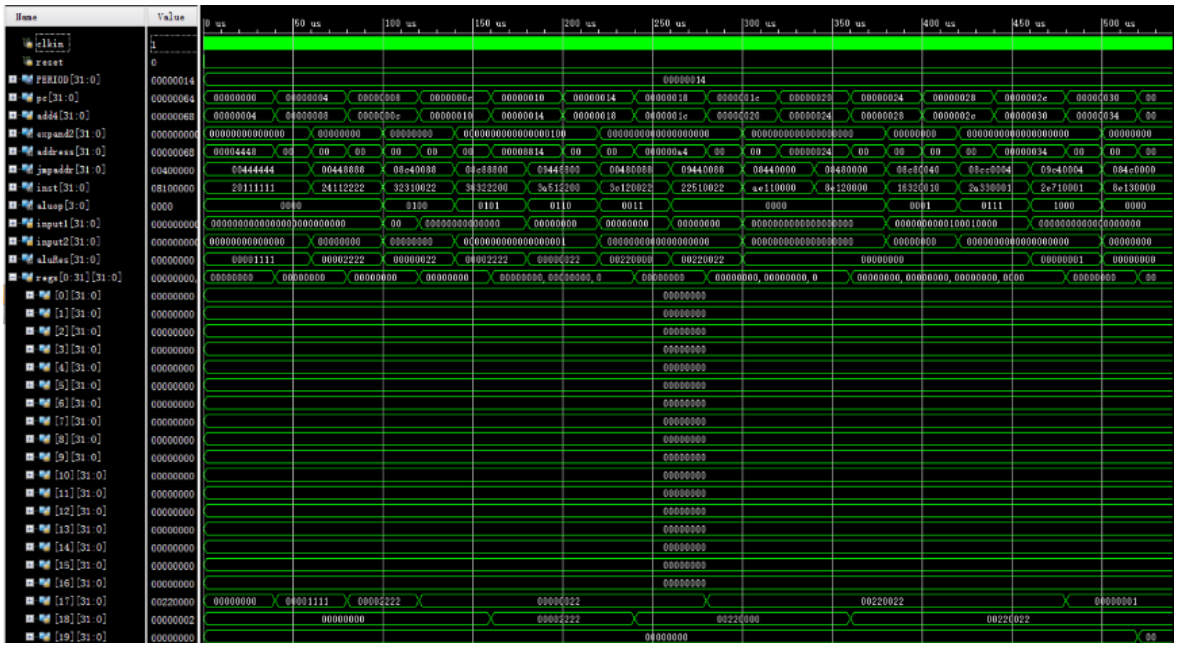
五. 仿真及实验结果

仿真代码：

```
module topsim();
// Inputs
    reg clkin;
    reg reset;
    reg switch;
    wire[3:0]sm_wei;
    wire[7:0]sm_duan;
    // Instantiate the Unit Under Test (UUT)
    top uut (
        .clkin(clkin),
        .reset(reset),
        .switch(switch),
        .sm_wei(sm_wei),
        .sm_duan(sm_duan)
    );
    initial begin
    // Initialize Inputs
    clkin = 0;
    reset = 1;
    switch=1;
```

```
        // Wait 100 ns for global reset to finish
        #100;
        reset = 0;
        end
        parameter PERIOD = 20;
        always begin
                clkin = 1'b0;
                #(PERIOD / 2) clkin = 1'b1;
                #(PERIOD / 2) ;
        end
endmodule
```

根据前面所述的 coe 文件中的指令进行仿真，仿真结果如下图：



根据波形记录下运算结果如下表：

机器码指令	汇编指令	运算结果
	main:	
20111111	addi $17,$0,0x1111	#$17=aluRes=0x1111
24112222	addiu $17,$0,0x2222	#$17=aluRes=0x2222
32310022	andi $17,$17,0x22	#$17=aluRes=0x22
36322200	ori $18,$17,0x2200	#$18=aluRes=0x2222
3a512200	xori $17,$18,0x2200	#$17=aluRes=0x22
3c120022	lui $18,0x0022	#$18=aluRes=0x220000
22510022	addi $17,$18,0x22	#$17=aluRes=0x220022
ae110000	sw $17, 0($16)	0000 000c+0 位置（16）的单元值为 0x220022，aluRes=0
8e120000	lw $18, 0($16)	#$18=0x220022，aluRes=0
16320010	bne $17,$18,exit	不跳转，aluRes=0
2a330001	slti $19,$17,1	#$19=aluRes=0;

2e710001	sltiu $17,$19,1	#$17=aluRes=1;
8e130000	lw $19, 0($16)	#$19=0x220022，aluRes=0
02538820	add $17,$18,$19	#$17=aluRes=0x440044
02329822	sub $19,$17,$18	#$19=aluRes=0x220022;
20120002	addi $18,$0,2	#$18=aluRes=0x2;
02728824	and $17,$19,$18	#$17=aluRes=0x2;
02329827	nor $19,$17,$18	#$19=aluRes=0xfffffffd;
02729825	or $19,$19,$18	#$19=aluRes=0xffffffff;
02728826	xor $17,$19,$18	#$17=aluRes=0xfffffffd;
8e110000	lw $17, 0($16)	#$17=0x220022，aluRes=0
00119c02	srl $19,$17,16	#$19=aluRes=0x0022;
00138c00	sll $17,$19,16	#$17=aluRes=0x220000;
1233000e	beq $17,$19,exit	不跳转，#aluRes=0x21ffde
0232982a	slt $19,$17,$18	#$s3= aluRes=0
08100000	j main	aluRes=0
	exit:	
20110000	addi $17,$0,0	不会运行到这里
08100000	j main	不会运行到这里

以上运算结果经检验均正确。

Basys3 数码管显示结果：

由于指令太多，且碍于文件大小的限制，这里贴两条指令的结果的后 16 位，完整的显示结果已经由 TA 检验过。

xor $17,$19,$18：

slt $19,$17,$18：

六. 实验感想

这次 CPU 设计的实验的过程是曲折的，但是结果是快乐的。就我而言，我觉得最大的困难时一开始先把老师给出的代码理清逻辑。在理论课上，我们学习过 CPU 的基本原理，不过缺乏实际演练的话，其实还是会对这个工作流程的认识模糊不清。为了清晰地认识整个流程，我对照着原理图，从 PC 开始，一步一步顺着线路去理解代码，终于在最后理清了整个代码的逻辑关系。

搞清楚代码逻辑关系和基本原理之后，接下来我要做的就是增加指令。我们可以发现 R 型指令中，仅仅依靠功能码去区分不同的 R 型指令，所以我首先选择增加 R 型指令。这个过

程还是比较顺利，只需要修改一下 alu 计算模块和 alu 控制模块即可。J 型指令我没有去增加，而 I 型指令由于从操作码开始就不同了，所以我们要先修改控制单元，在控制单元中根据操作码增加 I 型指令，然后再对应到 ALU 的操作中去。然后由于我们增加的指令，导致 PC 有 4 种改变方式，这里我们可以通过增加复用器去根据不同情况来选择，其它地方出现类型情况也采用同样的解决方法。

这次实验我觉得自己还是有不少收获的，重要的是对 CPU 的工作流程和基本原理的认识又更进了一步。