《计算机组成原理与接口技术实验》

实验报告

（实验二）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	16 软件工程二（4）班
学 生 姓 名 ：	刘亚辉
学　　号 ：	16340157
时　　间 ：	2018　年　5　月　22　日

成 绩 ：

实验二 ： 单周期CPU设计与实现

一. 实验目的

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

二. 实验内容

设计一个单周期 CPU，该 CPU 至少能实现以下指令功能操作。指令与格式如下：

==> 算术运算指令

（1）**add rd , rs, rt** （**说明：以助记符表示，是汇编指令；以代码表示，是机器指令**）

000000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs + rt。**reserved 为预留部分，即未用，一般填“0”。**

（2）addi rt , rs ,**immediate**

000001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs + (sign-extend)**immediate；immediate** 符号扩展再参加“加”运算。

（3）sub rd , rs , rt

000010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs - rt

==> 逻辑运算指令

（4）ori rt , rs ,**immediate**

010000	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs | (zero-extend)**immediate；immediate** 做“0”扩展再参加“或”运算。

（5）and rd , rs , rt

010001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs & rt；逻辑与运算。

（6）or rd , rs , rt

010010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

==>移位指令

（7）sll　rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved

功能：rd<－rt<<(zero-extend)sa，左移 sa 位 ，(zero-extend)sa

==>比较指令

（8）slti　rt, rs,immediate　　带符号

011011	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if (rs <(sign-extend)**immediate**)　rt =1 else　rt=0，　具体请看表 2 ALU 运算功能表，带符号

==> 存储器读/写指令

（9）sw　rt ,**immediate**(rs)　写存储器

100110	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：memory[rs+ (sign-extend)**immediate**]←rt；**immediate** 符号扩展再相加。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

（10) lw　rt , **immediate**(rs)　读存储器

100111	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt ← memory[rs + (sign-extend)**immediate**]；**immediate** 符号扩展再相加。

即读取 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数，然后保存到 rt 寄存器中。

==> 分支指令

（11）beq　rs,rt,**immediate**

110000	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if(rs=rt) pc←pc + 4 + (sign-extend)**immediate** <<2　else pc ←pc + 4

特别说明：**immediate 是从 PC+4 地址开始和转移到的指令之间指令条数。immediate** 符号扩展之后左移 2 位再相加。为什么要左移 2 位？由于跳转到的指令地址肯定是 4 的倍数（每条指令占 4 个字节），最低两位是“00”，因此将 **immediate** 放进指令码中的时候，是右移了 2 位的，也就是以上说的“指令之间指令条数”。

（12）bne　rs,rt,**immediate**

110001	rs(5 位)	rt(5 位)	**immediate**

功能：if(rs!=rt) pc←pc + 4 + (sign-extend)immediate <<2　else pc ←pc + 4

特别说明：与 beq 不同点是，不等时转移，相等时顺序执行。

==>跳转指令

（13）j　addr

111000	addr[27..2]

功能：pc <－{(pc+4)[31..28],addr[27..2],2{0}}，无条件跳转。

说明：由于 MIPS32 的指令代码长度占 4 个字节，所以指令地址二进制数最低 2 位均为 0，将指令地址放进指令代码中时，可省掉！这样，除了最高 6 位操作码外，还有 26 位

可用于存放地址，事实上，可存放 28 位地址了，剩下最高 4 位由 pc+4 最高 4 位拼接上。

==> 停机指令

（14）halt

111111	00000000000000000000000000(26 位)

功能：停机；不改变 PC 的值，PC 保持不变。

三. 实验原理

单周期CPU指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期

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。

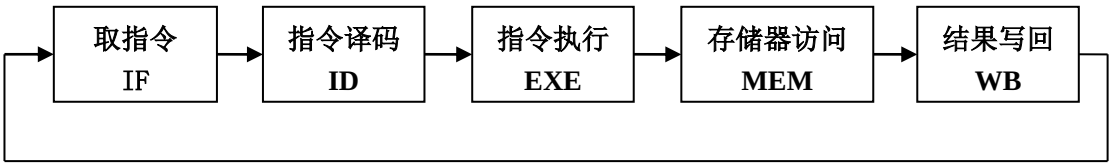
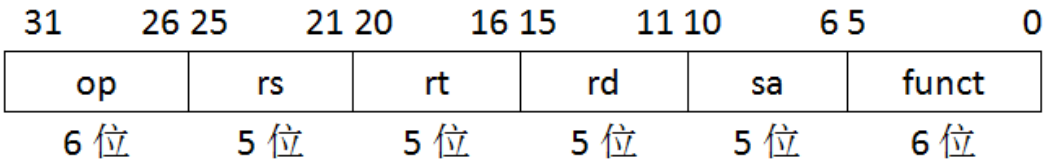


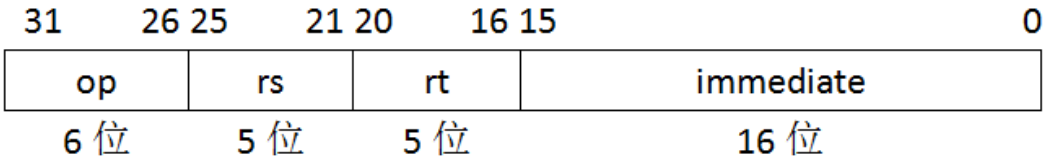
图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：

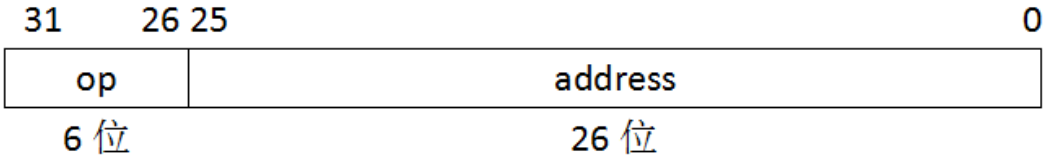
R 类型：



I 类型：



J 类型：



其中，

op：为操作码；

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

rt：可读可写。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：只写。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；

funct：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与操作码配合使用；

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（Load）/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。

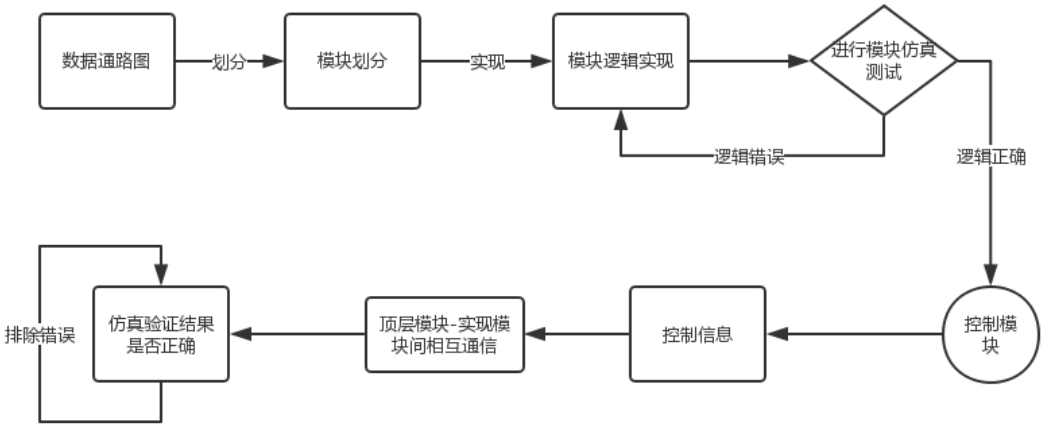
指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC的改变是在时钟上升沿进行的，这样稳定性较好。

四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

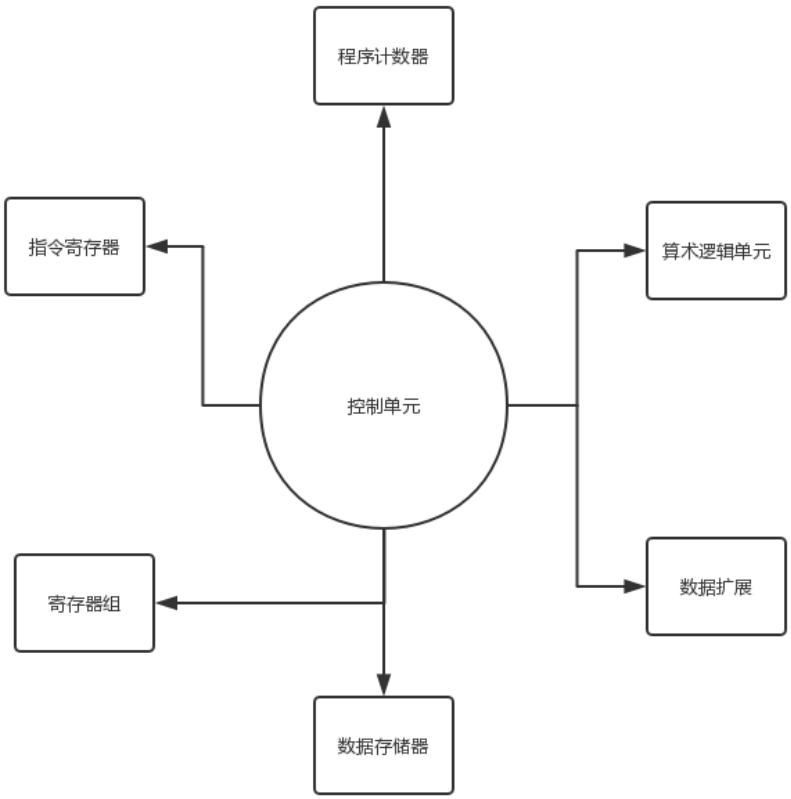
五. 实验过程与结果

1.CPU设计流程图：



2.CPU模块划分

在CPU的设计中，采用模块化设计的思想，将CPU设计分为多个主要模块，如下图所示：程序计数器，指令寄存器，数据存储器，算术逻辑单元，寄存器组，数据扩展，控制单元。其中，控制单元的设计最为重要，它像每个模块传递控制信号，使得各个控制模块可以相互作用从而正常工作。



3. 数据通路图

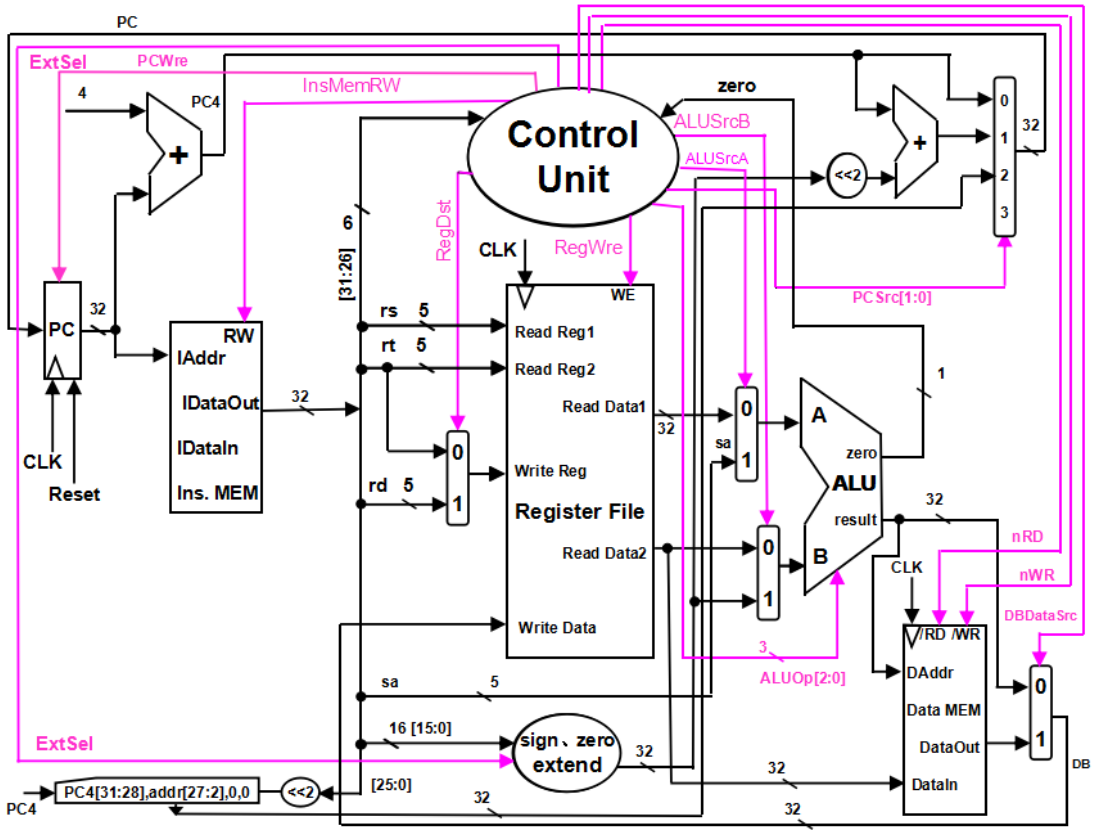


图 2　单周期 CPU 数据通路和控制线路图

4. 控制信号表

图 2 是一个简单的基本上能够在单周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，先给出寄存器地址，读操作时，输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。图中控制信号作用如表 1 所示，表 2 是 ALU 运算功能表。

表 1 控制信号的作用

控制信号名	状态“0”	状态“1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addi、or、and、ori、beq、bne、slti、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、sll、beq、bne	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、	来自数据存储器（Data MEM）的输出，相关指令：lw

	slti、sll	
RegWre	无写寄存器组寄存器，相关指令：beq、bne、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	输出高阻态	**读数据存储器，相关指令：lw**
mWR	无操作	**写数据存储器，相关指令：sw**
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addi、ori、lw、slti	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、sll
ExtSel	(zero-extend)**immediate(0 扩展)，**相关指令：ori	(sign-extend)**immediate (符号扩展)**，相关指令：addi、slti、sw、lw、beq、bne
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)； 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bne(zero=0)； 10：pc<－{(pc+4)[31:28],addr[27:2],2{0}}，相关指令：j； 11：未用	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

相关部件及引脚说明：

Instruction Memory：指令存储器，

Iaddr，指令存储器地址输入端口

IDataIn，指令存储器数据输入端口（指令代码输入端口）

IDataOut，指令存储器数据输出端口（指令代码输出端口）

RW，指令存储器读写控制信号，为 0 写，为 1 读

Data Memory：数据存储器，

Daddr，数据存储器地址输入端口

DataIn，数据存储器数据输入端口

DataOut，数据存储器数据输出端口

mRD，数据存储器读控制信号，为 0 读

mWR，数据存储器写控制信号，为 0 写

Register File：寄存器组

Read Reg1，rs 寄存器地址输入端口

Read Reg2，rt 寄存器地址输入端口

Write Reg，将数据写入的寄存器端口，其地址来源 rt 或 rd 字段

Write Data，写入寄存器的数据输入端口

Read Data1，rs 寄存器数据输出端口

Read Data2，rt 寄存器数据输出端口

WE，写使能信号，为 1 时，在时钟边沿触发写入

ALU：　算术逻辑单元

result，ALU 运算结果
zero，运算结果标志，结果为 0，则 zero=1；否则 zero=0

通过利用控制信号表，我们可以容易的获得一套完整的贯穿于各个模块之前的标准。控制信号表就相当于各个模块所需要遵守的约定，在某个信息到来之后做出某种响应。即通过控制单元产生信号，其他模块捕获信号，然后根据控制信号表的准则做出相应的响应。比如：现在将要进行一个加法操作，那么控制单元应该赋值：PCWre=1，表示PC需要进行写操作来读取下一条指令；ALUOp=3’b000，表示ALU逻辑控制单元需要进行对输入的数字进行加法运算……PC模块在获得PCWre的信号值之后，就将PC进行写操作，获取下一条指令的地址，ALU在获得ALUOp信号之后根据判断，当输入的数字进行加操作，然后输出结果……所以，通过在控制单元对各个信号值进行相应的赋值即可完成指令的执行操作。

5．CPU模块实现

在具体实现的过程中，为了在仿真阶段更加清晰的理解数据的传递，并且更加容易的进行排错，我将上方的模块划分为更小的子模块：

· 程序计数器模块：（包括PCAdd4模块，PCAddOffset模块，PC_Join模块，Multiplexer4模块，PC模块）

PCAdd4模块：（此模块执行+4操作）

```
always @(pc) begin
    pc4<=pc+4;
end
```

PCAddOffset模块：（pc4加上偏移量offset，用于beq和bne等分支跳转语句）

```
always @(pc4 or offset) begin
    pc4Offset <= pc4+(offset << 2);
end
```

PC_Join模块：（用于J指令，将pc4与左移两位的跳转地址相加）

```
always @(addr) begin
    PCJoined <= {pc4[31:28], addr, 2'b00};
end
```

Multiplexer4模块：（通过判断PCSrc的值，给pc的下一条指令pcIn赋值，其中pc1，

pc2，pc3分别代表上面三个模块的输出值）

```
always @(PCSrc or pc3 or pc2 or pc1) begin
    case(PCSrc)
        2'b00: pcIn <= pc1;
        2'b01: begin
            pcIn <= pc2;
        end
        2'b10: pcIn <= pc3;
        default:
            $display("PCSrc Wrong!");
    endcase
    //$display("PCSrc---*****---%h", PCSrc);
end
```

PC模块：（此模块主要对获得下一条指令地址，存于pcNext中）

```
always @(posedge CLK or negedge Reset) begin
    if (Reset == 1) begin
        if (PCWre == 1) begin
            pcNext <= pcIn;
           end
    end
    else
        pcNext <= 0;
end
```

· 指令寄存器模块

IR模块：（利用$readmemb函数来从文件中获取所有指令，并保存于IDataMem8位宽的reg数组中，然后通过对IDataOut赋值即可完成指令的读取）

```
reg [7:0]IDataMem[0:80];
initial begin
    $readmemb("D:/Vivado_projects/Single_CPU/Instructions.txt",IDataMem);
end
always@(*)
begin
    if(InsMemRW == 1)
    begin
        IDataOut[31:24] <= IDataMem[IAddr];
        IDataOut[23:16] <= IDataMem[IAddr+1];
        IDataOut[15:8] <= IDataMem[IAddr+2];
```

```
        IDataOut[7:0] <=IDataMem[IAddr+3];
    end
end
```

·寄存器组模块

rd_rt模块：（此模块中，主要进行WB写回操作时寄存器的选择，通过判断信号RegDst，当其为0时，写回到rt所指向的寄存器中；当RegDst为1时，写回到rd所指向的寄存器中）

```
always@(rd or rt or RegDst)begin
    if(RegDst==0)
        result<=rt;
    else
        result<=rd;
end
```

RegFile模块：（指令位宽为32为，寄存器有31个（去除0号寄存器），所以定义一个宽为32的寄存器数组，然后可以利用assign语句来对ReadData1，ReadData2输出进行持续型赋值，在always块中，判断控制信号，对寄存器进行写回操作）

```
reg [31:0] regFile[1:31];
assign ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1];
assign ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];

always@(negedge CLK)
begin
    if(RegWre==1 && WriteReg != 0)
        regFile[WriteReg]<=WriteData;
end
```

·数据扩展模块：

DataExtend模块：（通过判断控制信号ExtSel的输入，当其为0时，进行无符号扩展，为1时，进行有符号扩展，判断imData数据最高位即可）

```
always @(ExtSel or imData) begin
    if(ExtSel == 0)
        dataExtend <= {16'h0000,imData};
    else begin
        if(imData[15] == 0)
              dataExtend <= {16'h0000,imData};
          else begin
              dataExtend <= {16'hffff,imData};
```

```
                $display("%h",dataExtend);
            end
        end
end
```

·逻辑运算单元模块：

ALUSrcASelect模块：（此模块处理ALU模块的输入A，判断控制信号ALUSrcA的值，当其为0时，选择寄存器组的ReadData1；当其为1时，选择sa，即偏移量）

```
always@(Data1 or sa or ALUSrcA) begin
    if(ALUSrcA==0)
        result<=Data1;
    else
        result<={16'h000000,2'b000,sa};
end
```

ALUSrcBSelect模块：（此模块处理ALU模块的输入B，判断控制信号ALUSrcB的值，当其为0时，选择寄存器组的ReadData2；当其为1时，选择immediate，即立即数）

```
always@(Data2 or immediate or ALUSrcB) begin
    if(ALUSrcB==0)
        result<=Data2;
    else
        result<=immediate;
end
```

ALU模块：（从模块即为逻辑运算单元的核心部分，根据ALUopcode的输入，来执行不同的逻辑运算。其中zero的输出由最终的运算结果决定，当result=0时，zero输出为1，反之输出为0）

```
assign zero = (result == 0) ? 1 : 0;
always @(ALUopcode or rega or regb)
begin
    case(ALUopcode)
        3'b000:result <= rega + regb;
        3'b001:result <= rega - regb;
        3'b010:result <= regb << rega;
        3'b011:result <= rega | regb;
        3'b100:result <= rega & regb;
        3'b101:result <= (rega < regb) ? 1 : 0;
        3'b110:
```

```
            begin
                if(rega<regb && rega[31]==regb[31])
                    result<=1;
                else if(rega[31]==1 && regb[31]==0)
                    result<=1;
                else
                    result<=0;
            end
        3'b111:result <= rega ^ regb;
        default:
            result<=32'h00000000;
    endcase
end
```

逻辑运算表如下：

ALUOp[2..0]	功能	描述
000	Y = A + B	加
001	Y = A − B	减
010	Y = B<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y= (A<B) ?1: 0	比较 A 与 B 不带符号
110	Y=(((rega<regb) && (rega[31] == regb[31])) ||((rega[31] ==1 && regb[31] == 0))) ? 1:0	比较 A 与 B 带符号
111	Y = A ⊕ B	异或

· 数据存储器模块

RAM模块：（在时钟脉冲上升沿到来时进行读操作，当mRD为1时，即可进行读操作，将存储器内部的值存储到输出信号DataOut中；在时钟脉冲下降沿打来时进行写操作，当mWR为1时，即可进行写操作，将writeData中的数据存储到存储器内。这样的处理方式更加稳定。）

```
reg [7:0] ram[0:60];
//Read
assign DataOut[7:0] = (mRD==1) ? ram[address+3] : 8'bz;
assign DataOut[15:8] = (mRD==1) ? ram[address+2] : 8'bz;
assign DataOut[23:16] = (mRD==1) ? ram[address+1] : 8'bz;
```

```
assign DataOut[31:24] = (mRD==1) ? ram[address] : 8'bz;
//Write
always@(negedge clk)
begin
    if(mWR==1)
    begin
        ram[address] <= writeData[31:24];
        ram[address+1] <= writeData[23:16];
        ram[address+2] <= writeData[15:8];
        ram[address+3] <= writeData[7:0];
    end
end
```

·控制单元模块：

ControlUnit模块：（此模块即为CPU的核心部分，上述的控制信号都是由此模块发出，通过判断opcode的值，来发出不同的信号，下属代码以add指令为例，其中PCWre=0，mRD=0,mWR=0,InsMemRW=0,PCSrc=00,RegDst=1,RegWre=1,ALUSrcA=0,ALUSrcB=0,ExtSel=z,ALUOp=000,DNDataSrc=0）

```
assign PCWre = (opcode == 6'b111111) ? 0 : 1;
assign mRD = (opcode == 6'b100111) ? 1 : 0;
assign mWR = (opcode == 6'b100110) ? 1 : 0;
assign InsMemRW = 1;
always @(opcode or zero) begin
    //add rd,rs,rt
    if(opcode == 6'b000000) begin
        PCSrc[1:0]<=2'b00;
        RegDst <= 1;
        RegWre <= 1;
        ALUSrcA <= 0;
        ALUSrcB <= 0;
        ExtSel <= 1'bz;
        ALUOp[2:0] <= 3'b000;
        DBDataSrc <= 0;
    end
```

其他指令的代码与上述代码类似。

所有指令的真值表如下：

指令	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	InsMemRW	nRD	nWR	RegDst	ExtSel	PCSrc[1:0]	ALUOp[2:0]
add (opcode=="000000")	1	0	0	0	1	1	0	0	1	z	00	000
addi (opcode=="000001")	1	0	1	0	1	1	0	0	0	1	00	000
sub (opcode=="000010")	1	0	0	0	1	1	0	0	1	z	00	001
ori (opcode=="010000")	1	0	0	0	1	1	0	0	0	1	00	011
add (opcode=="010001")	1	0	0	0	1	1	0	0	1	z	00	100
or (opcode=="010010")	1	0	0	0	1	1	0	0	1	z	00	011
sll (opcode=="011000")	1	1	0	0	1	1	0	0	0	z	00	010
slti (opcode=="011011")	1	0	1	0	1	1	0	0	0	1	00	110
sw (opcode=="100110")	1	0	1	0	0	1	0	1	z	1	00	000
lw (opcode=="100111")	1	0	1	1	1	1	1	0	0	1	00	000
beq (opcode=="110000")	1	0	0	z	0	1	0	0	z	1	zero==1?01:00	001
bne (opcode=="110001")	1	0	0	z	0	1	0	0	z	1	zero==1?00:01	001
j (opcode=="111000")	1	z	z	z	0	1	0	0	z	z	10	zzz
halt (opcode=="111111")	0	z	z	z	0	1	0	0	z	z	zz	zzz

· 其他模块：

DBDataSrcSelect模块：（判断控制单元的输出信号DBDataSrc，当其为0时，数据总线的上的数值，为ALU模块的输出结果；当其为1时，数据总线上的数值为RAM的输出结果）

```
always@(DBDataSrc or result or DataOut) begin
    if(DBDataSrc==0)
        DB<=result;
    else
        DB<=DataOut;
end
```

· 顶层模块：

Single_CPU模块：（将模块进行实例化，并将其用线连接起来，完成整个数据通路，下面是PC模块和RAM模块的例子）

```
PC pc(
    .CLK(clock),
    .Reset(reset),
    .PCWre(PCWre),
    .pcIn(pcIn),
    .pcNext(pcNext)
);
RAM ram(
    .clk(clock),
    .address(result),
    .writeData(Data2),
    .mRD(mRD),
    .mWR(mWR),
    .DataOut(DataOut)
```

```
15. );
```

其他模块与此类似。

6. 验证CPU设计的正确性：

・仿真文件

```
1.  reg CLK;
2.  reg reset;
3.  initial begin
4.      CLK = 0;
5.      reset = 0;
6.      #20 reset = 1;
7.  end
8.  always #10 CLK = ~CLK;
```

定义时钟信号，周期为20ns，reset变量延迟20ns之后置为1。定义的wire类型的变量省略，生成Single_CPU实例如下：

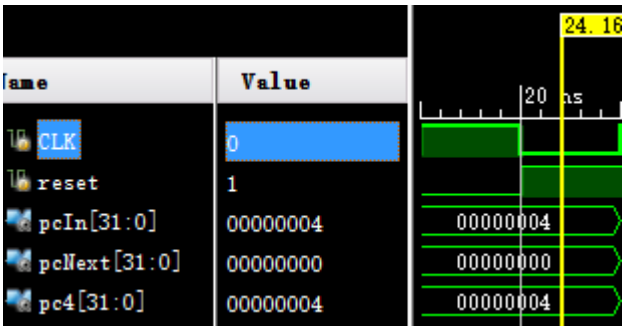
```
1.  Single_CPU single_cpu(CLK,reset,pcIn,pcNext,pc4,IDataOut,opcode,rs, rt,rd,
2.
3.      sa,immediate,address,pc_Join_result,rd_rtSelected,rega,regb,RegDst,
4.
5.      InsMemRW, PCWre,ExtSel,DBDataSrc,mWR,mRD,ALUSrcB,ALUSrcA, PCSrc,ALUOp,
6.
7.      RegWre, immediateExtend,zero,Data1, Data2,result,DB,pc4Offset,DataOut);
```

其中变量较多，用于输出波形查错。

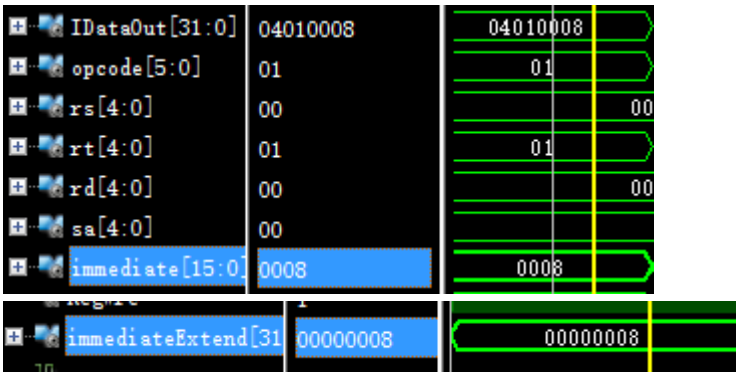
说明：变量pcNext表示当前pc所输出的值（当前pc的指令），pcIn表示pc模块输入的pc指令（即下一条pc的值）

・指令1：addi $1,$0,8

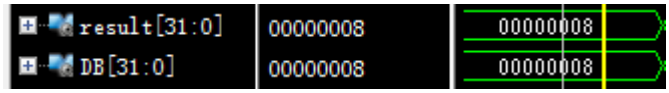
此时$0=0x0，所以运行结束后$1=0x8



上图可知，指令寄存器正确显示，当前指令pcNext地址为0x00000000，下一条指令pcIn为0x00000004；



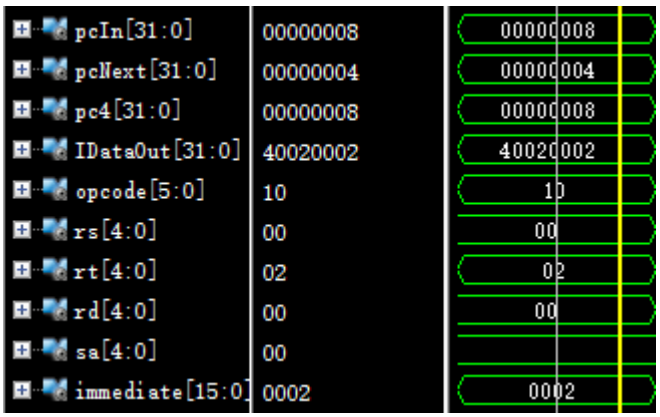
上图可知IDataOut输出的指令为0x04010008，即为addi $1,$0,8的十六进制表示，同时rs为0x 0，rt为0x 1，立即数扩展后为0x 8，所以指令读取正确，opcode为0x 00，即为addi运算

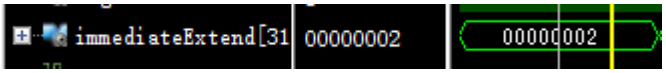


此时ALU计算结果result为0x 8，DB总线上的数据也为0x 8，所以该指令运行正确

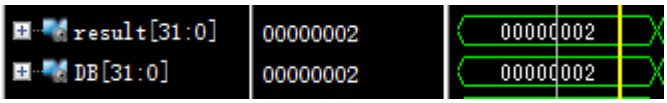
·指令2：ori $2,$0,2

运行结束后，$2=0x2





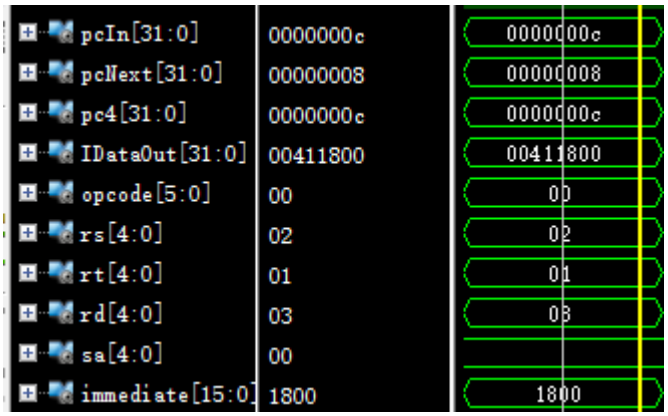
上图可知，当前指令pcNext的值为0x00000004，pcIn的值为0x00000008，正确；opcode为0x 01，即为ori运算，IdataOut的输出为0x40020002，正确，rs为0x 0，rt为0x 2，immediate扩展后为0x 2，数据正确；



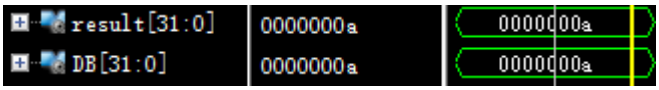
此时ALU计算结果result为0x 2，DB总线上的数据也为0x 2，所以该指令运行正确

·指令3：add $3,$2,$1

此时$2=0x2，$1=0x8，所以$3=0xa



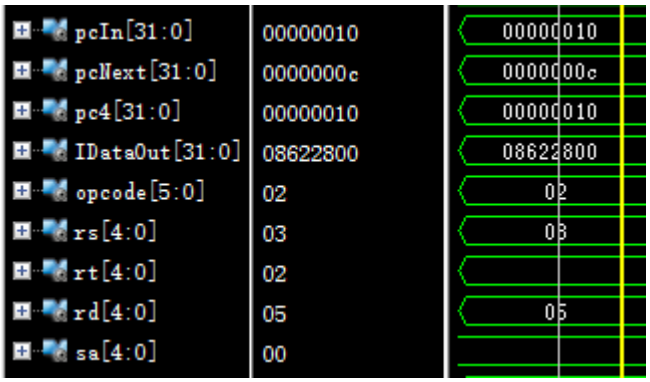
上图可知，当前指令pcNext的值为0x00000008，pcIn的值为0x0000000c，正确；opcode为0x 00，即为add运算，IdataOut的输出为0x00411800，正确，rs为0x 2，rt为0x 1，rd为0x 3，正确；



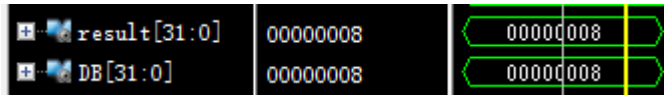
此时ALU计算结果result为0x 10，DB总线上的数据也为0x 10，所以该指令运行正确

·指令4：sub $5,$3,$2

此时$3=0xa，$2=0x2，所以$5=0x8



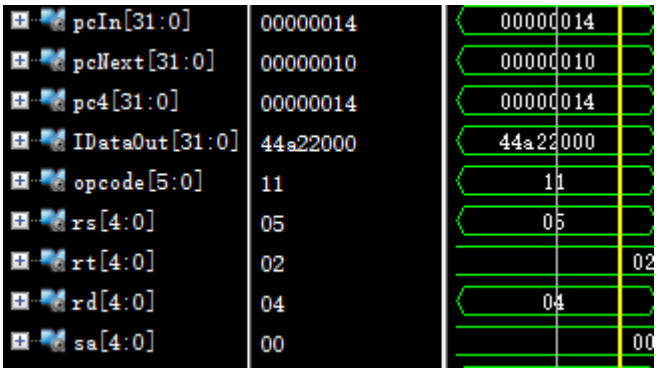
上图可知，当前指令pcNext的值为0x0000000c，pcIn的值为0x00000010，正确；opcode为0x02，即为sub运算，IdataOut的输出为0x08622800，正确；rs为0x 3，rt为0x 3，rd为0x 5，正确；



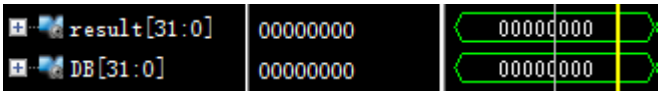
此时ALU计算结果result为0x 8，DB总线上的数据也为0x 8，所以该指令运行正确

·指令5：and $4,$5,$2

此时$5=0x8，$2=0x2，所以$4=0x0



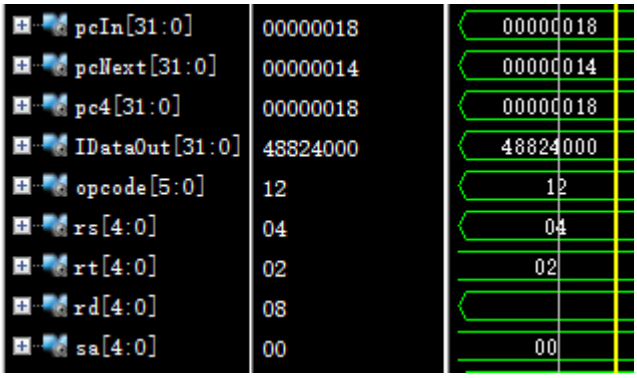
上图可知，当前指令pcNext的值为0x00000010，pcIn的值为0x00000014，正确；opcode为0x11，即为and运算，IdataOut的输出为0x44122000，正确；rs为0x 5，rt为0x 2，rd为0x 4，正确；



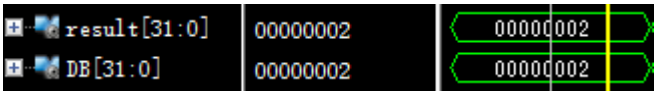
此时ALU计算结果result为0x 0，DB总线上的数据也为0x 0，所以该指令运行正确

·指令6：or $8,$4,$2

此时$4=0x0，$2=0x2，所以$8=0x2



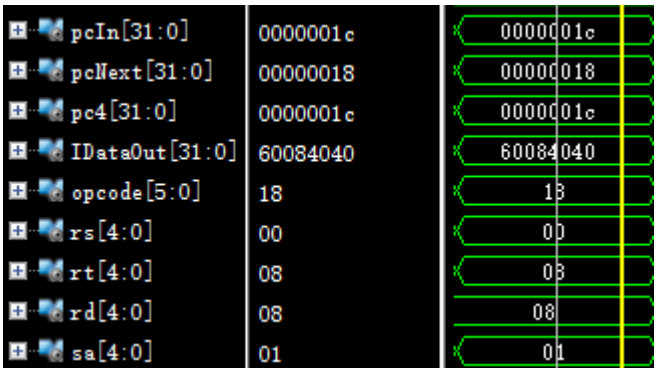
上图可知，当前指令pcNext的值为0x00000014，pcIn的值为0x00000018，正确；opcode为0x12，即为or运算，IdataOut的输出为0x48824000，正确；rs为0x 4，rt为0x 2，rd为0x 8，正确；



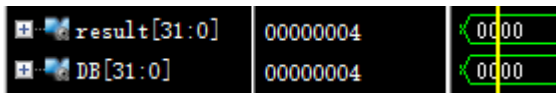
此时ALU计算结果result为0x 2，DB总线上的数据也为0x 2，所以该指令运行正确

·指令7：sll $8,$8,1

此时$8=0x2，所以运行结束后$8=0x4



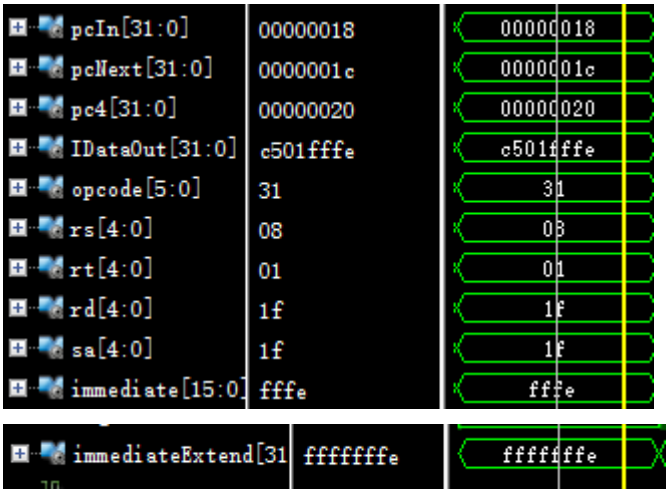
上图可知，当前指令pcNext的值为0x00000018，pcIn的值为0x0000001c，正确；opcode为0x18，即为sll运算，IdataOut的输出为0x60084040，正确；rt为0x8，rd为0x8，sa为0x1，正确；



此时ALU计算结果result为0x 4，DB总线上的数据也为0x 4，所以该指令运行正确

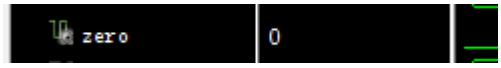
·指令8：bne $8,$1,-2

此时$8=4，$1=8，二者不相等，所以向前跳转一条指令



由于$8此时等于4，$1等于8，所以该指令会跳转到之前的一条指令，即sll $8,$8,1，上图可知，当前指令pcNext的值为0x0000001c，pcIn的值为0x00000018，正确；opcode为0x31，即为bne运算，IdataOut的输出为0xc501fffe，正确；rs为0x8，rd为0x1，immediate扩展后为0xfffffffe，正确；

在ALU中，我执行的是减法语句，所以zero输出为1，正确

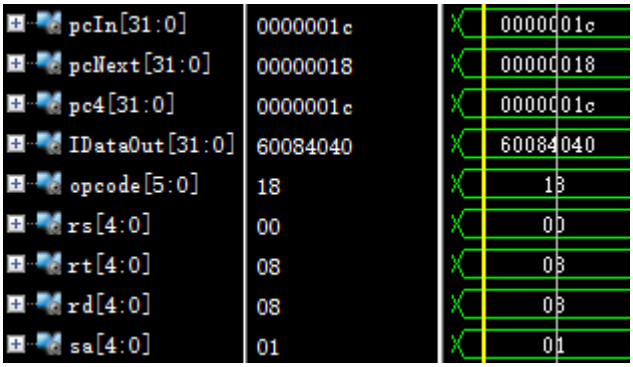


zero信号输出正确，计算结果不为0，所以输出为0

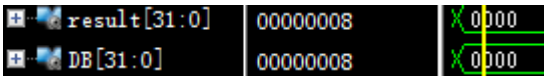
·指令9：sll $8,$8,1

再一次执行该语句：

此时$8=0x2，所以运行结束后$8=0x4



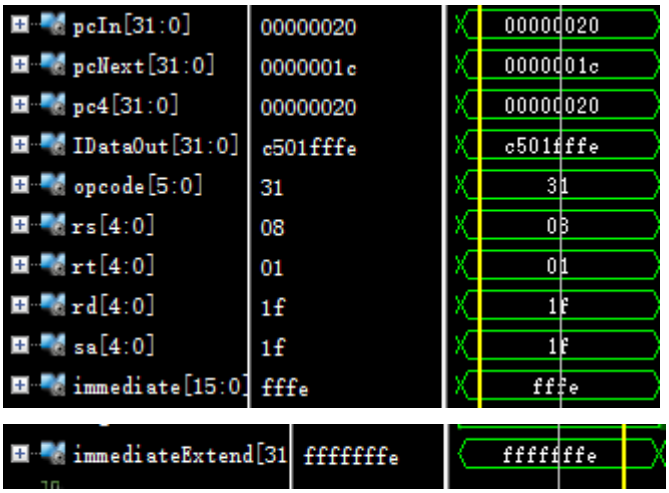
上图可知，当前指令pcNext的值为0x00000018，pcIn的值为0x0000001c，正确；opcode为0x18，即为sll运算，IdataOut的输出为0x60084040，正确；rt为0x8，rd为0x8，sa为0x1，正确；



$8内的值再次左移1为，ALU输出结果result=8，DB为8正确；

·指令10：bne $8,$1,-2

此时$8等于8，与$1相等，所以该不执行跳转



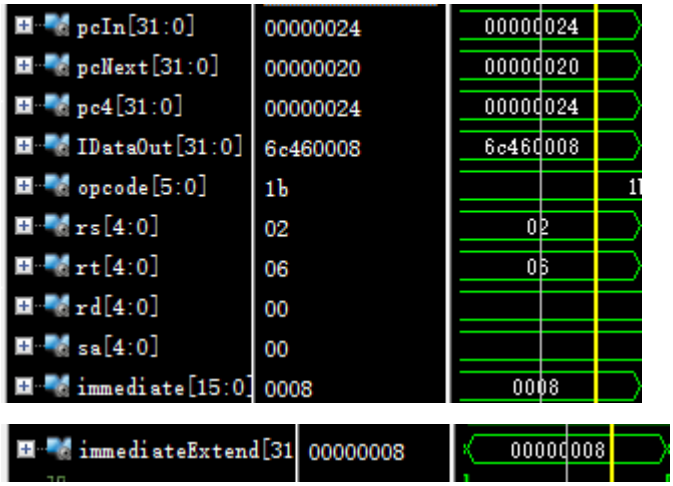
上图可知，当前指令pcNext的值为0x0000001c，pcIn的值为0x00000020，正确；opcode为0x31，即为bne运算，IdataOut的输出为0xc501fffe，正确；rs为0x8，rd为0x1，immediate扩展后为0xfffffffe，正确；

在ALU中，我执行的是减法语句，所以zero输出为1，正确

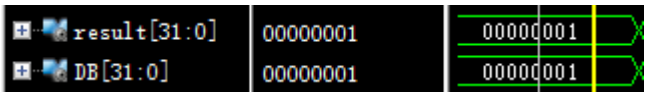
ALU的zero信号输出为1，正确

·指令11：slti $6,$2,8

此时$2的值为2，小于8，所以$6应为1



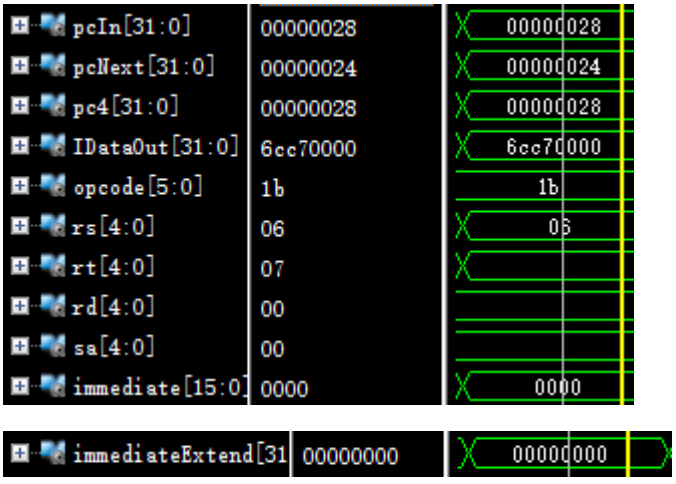
上图可知，当前指令pcNext的值为0x00000020，pcIn的值为0x00000024，正确；opcode为0x1b，即为slti运算，IdataOut的输出为0x6c460008，正确；rs为0x2，rt为0x6，immediate扩展后为0x8，正确；



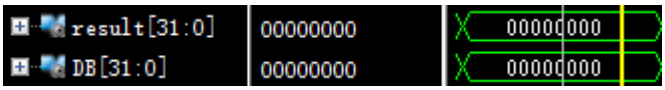
此时ALU计算结果result为0x 1，DB总线上的数据也为0x 1，所以该指令运行正确

·指令12：slti $7,$6,0

此时$6为1，大于0，所以$7为0



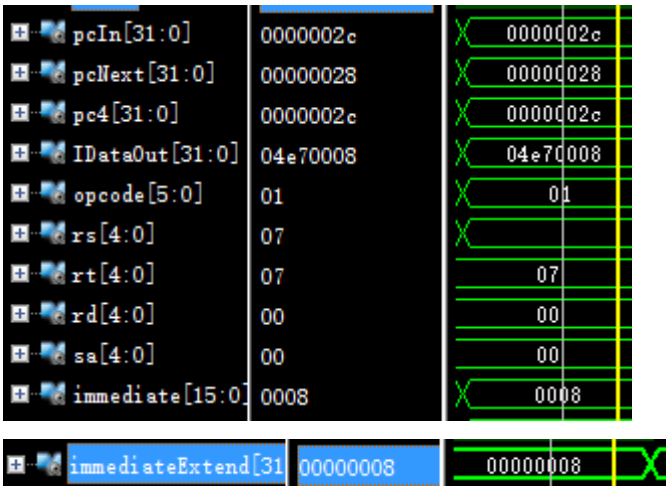
上图可知，当前指令pcNext的值为0x00000024，pcIn的值为0x00000028，正确；opcode为0x1b，即为slti运算，IdataOut的输出为0x6cc70000，正确；rs为0x6，rt为0x7，immediate扩展后为0x0，正确；



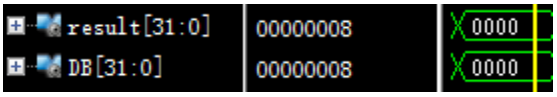
此时ALU计算结果result为0x 0，DB总线上的数据也为0x 0，所以该指令运行正确

·指令13：addi $7,$7,8

此时$7值为0，所以加上8之后结果为8



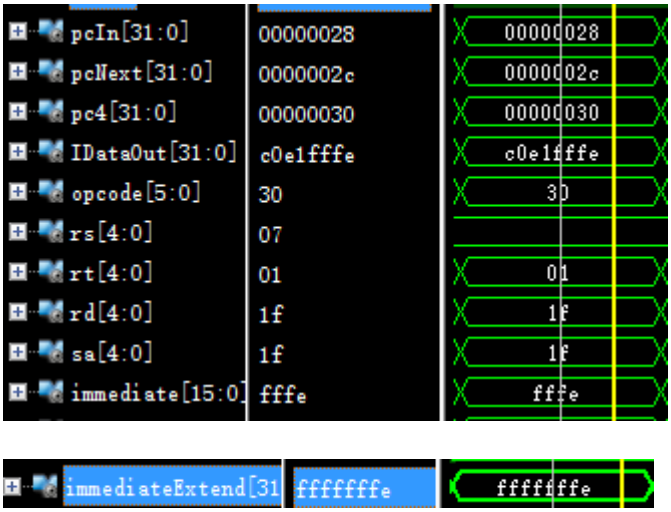
上图可知，当前指令pcNext的值为0x00000028，pcIn的值为0x0000002c，正确；opcode为0x01，即为addi运算，IdataOut的输出为0x04e70008，正确；rs为0x7，rt为0x7，immediate扩展为0x8，正确；



此时ALU计算结果result为0x 8，DB总线上的数据也为0x 8，所以该指令运行正确

·指令14：beq $7,$1,-2

此时$7=8,$1=8,所以该指令结束之后，会向前跳转1条指令；

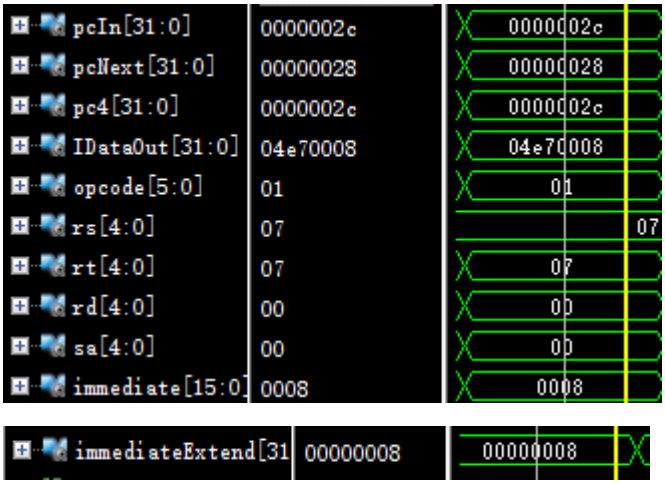


上图可知，当前指令pcNext的值为0x0000002c，pcIn的值为0x00000028，正确；opcode为0x30，即为beq运算，IdataOut的输出为0xc0e1fffe，正确；rs为0x7，rt为0x1，immediate扩展后为0xfffffffe，正确；

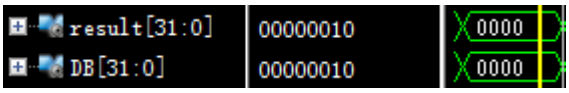
在ALU中，我执行的是减法语句，所以zero输出为1，正确

·指令15：addi $7,$7,8

此时$7为8，加上8之后，$7应为16



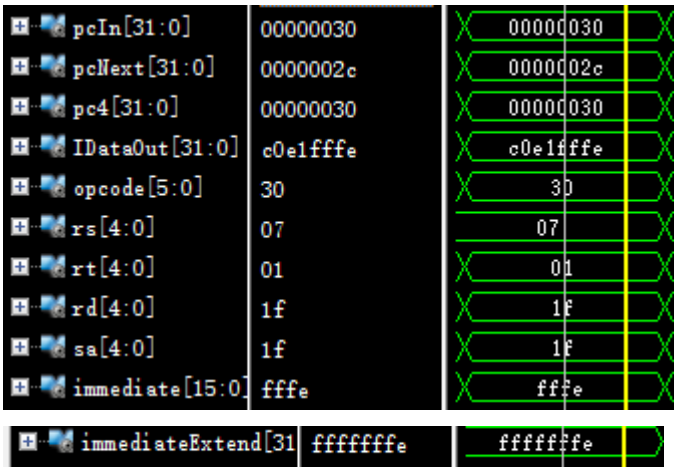
上图可知，当前指令pcNext的值为0x00000028，pcIn的值为0x0000002c，正确；opcode为0x01，即为addi运算，IdataOut的输出为0x04e70008，正确；rs为0x7，rt为0x7，immediate扩展后为0x8，正确；



此时ALU计算结果result为0x 10，DB总线上的数据也为0x 10，所以该指令运行正确

·指令16：beq $7,$1,-2

此时$7为16，$1为8，所以二者不相等，部进行跳转

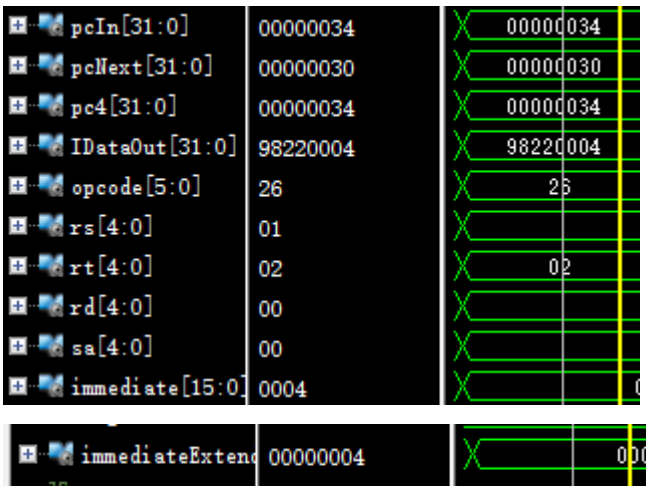


上图可知，当前指令pcNext的值为0x0000002c，pcIn的值为0x00000028，正确；opcode为0x30，即为beq运算，IdataOut的输出为0xc0e1fffe，正确；rs为0x7，rt为0x1，immediate扩展后为0xfffffffe，正确；

在ALU中，我执行的是减法语句，所以zero输出为0，正确

·指令17：sw $2,4($1)

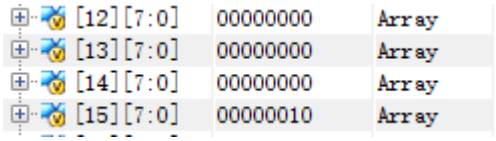
此时$2为2，$1为8，所以最终结果为，在RAM第12到15字节存储的数据即为0x00000002



上图可知，当前指令pcNext的值为0x00000030，pcIn的值为0x00000034，正确；opcode为0x26，即为sw运算，IdataOut的输出为0x98220004，正确；rs为0x1，rt为0x2，immediate为0x4，正确；

Data2（寄存器组的输出）为0x2，此输入接入到RAM的DataIn输入端口

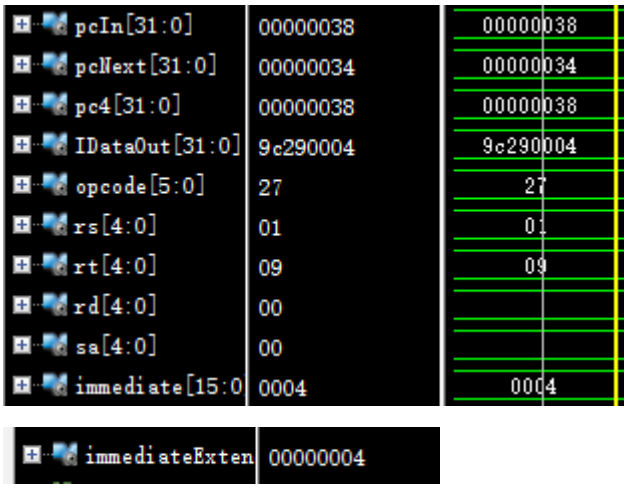
此时mWR为1，表示写操作，所以该指令正确将数据写入到RAM中



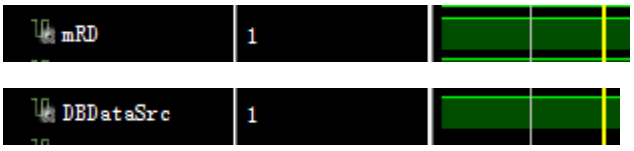
由上图可知，在RAM的12-15字节正确存储数字2.

·指令18：lw $9,4($1)

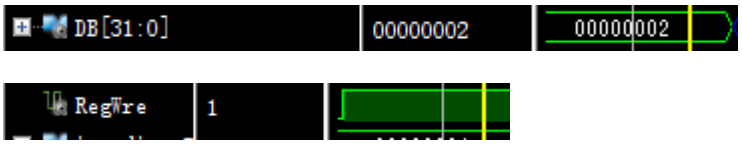
此时$1为8，这条指令即将RAM中第12-15字节的数字存储到$9中，最终$9为2



上图可知，当前指令pcNext的值为0x00000034，pcIn的值为0x00000038，正确；opcode为0x27，即为lw运算，IdataOut的输出为0x9c290004，正确；rs为0x1，rt为0x9，immediate为0x4，扩展之后为0x4，正确；



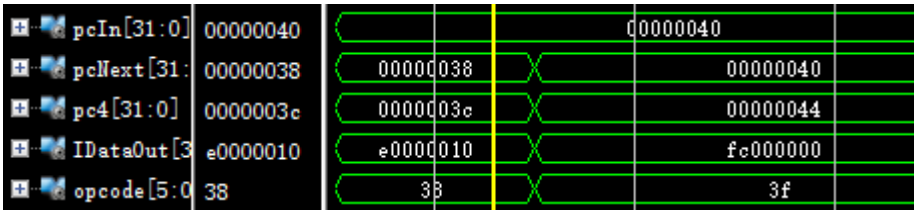
mRD为1，表示RAM处于读状态，DBDataSrc为1，表示DB总线中的数据为RAM读取得到的

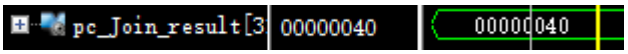


由上图可知，DB中的数据为0x2，寄存器组的RegWre控制信号为1，所以该指令执行

正确

·指令19：j 0x00000040

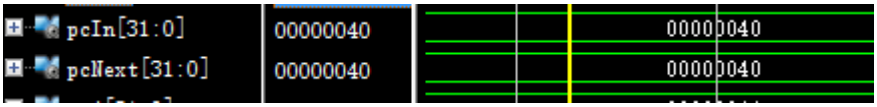




上图可知，当前指令pcNext的值为0x0000003c，pcIn的值为0x00000040，变量pc跳转合并结果值pc_Join_result为0x00000040正确；opcode为0x38，即为J跳转指令，IdataOut的输出为0xe0000010，正确；

·指令20：addi $10,$10,10（未执行）

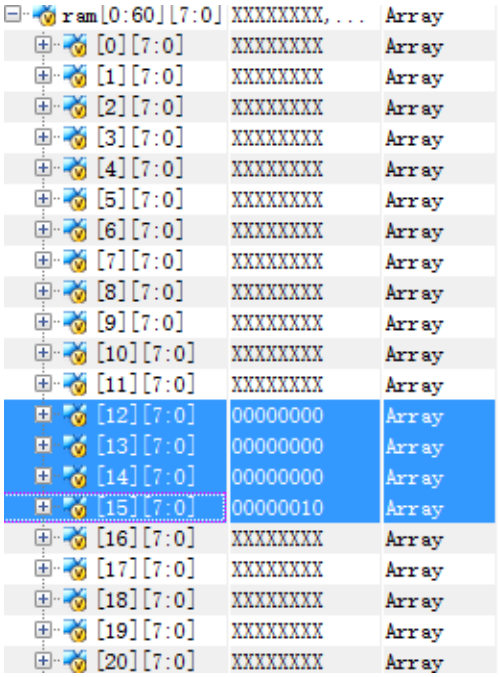
·指令21：halt



pc保持不变。

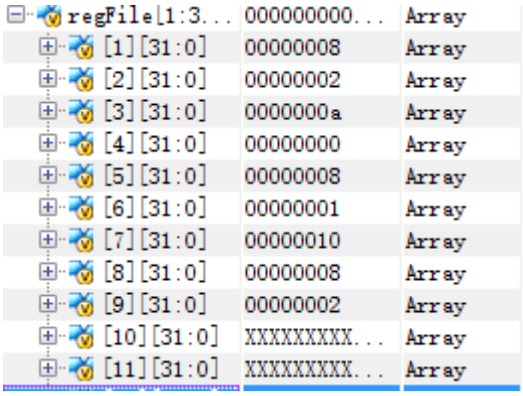
7. 最终执行结果如下：

·RAM（数据存储器）中最终存储的结果为：



在12-15字节存储的数据为0x00000002

·RegFile（寄存器组）最终存储的结果为：



regFile[1:3...	000000000...	Array
[1][31:0]	00000008	Array
[2][31:0]	00000002	Array
[3][31:0]	0000000a	Array
[4][31:0]	00000000	Array
[5][31:0]	00000008	Array
[6][31:0]	00000001	Array
[7][31:0]	00000010	Array
[8][31:0]	00000008	Array
[9][31:0]	00000002	Array
[10][31:0]	XXXXXXXXX...	Array
[11][31:0]	XXXXXXXXX...	Array

$1=0x8; $2=0x2; $3=0xa; $4=0x0; $5=0x8; $6=0x1; $7=0x10; $8=0x8; $9=0x2

8. 实现：将代码写到Basys3板上：

在原有代码的基础之上，主要添加显示display模块，和按钮模拟时钟输入的消抖EliminationBuffet模块.

·EliminationBuffet模块：

```
1.  reg delay1,delay2,delay3;
2.  assign out_key = delay1&delay2&delay3;
3.  always@(posedge clk)//CLK 100M
4.  begin
5.      delay1  <= in_key;
6.      delay2  <= delay1;
7.      delay3  <= delay2;
8.  end
```

有上述代码可知，只有当delay1，delay2和delay3都为高电平时，按键才会产生一个高电平的脉冲。通过将输入，进行延迟，延迟三个标准的时钟周期，即可简单的实现消抖处理，避免了按键不稳定的情况导致时钟信号输入错误。

·display模块：

变量如下：

```
1.  input clk,
2.  input switch1,
3.  input switch2,
```

```
input [31:0]pcNext,
input [31:0]pcIn,
input [4:0]rs,
input [31:0]rsData,
input [4:0]rt,
input [31:0]rtData,
input [31:0]result,
input [31:0]DB,
output reg [3:0]Anode_Activate,
output reg [6:0]LED_out

reg [25:0] div_counter = 0;
reg [3:0] LED_BCD;
```

input和output输入输出信号与外部其他模块通过wire变量相连；Anode_Activate表示4个七段数码管位控信号，4位，分别用0111，1011，1101，1110表示点亮不同的七段数码管；LED_out表示显示数字的数码管信号，7位，分别表示a，b，c，d，e，f，g七段数码管；div_counter为分频计数器；LED_BCD为数码管显示的数字的二进制表示。

分频部分：（每个时钟脉冲到来都使得分频计数器加1）

```
 always @(posedge clk) begin
    div_counter <= div_counter + 1;
end
```

数值部分：（每个时钟脉冲到来，判断当前分频计数器的20和19位的数值，分别显示不同的七段数码管，同时判断switch1和switch2来控制显示信号）

```
always@(posedge clk)
begin
    case(div_counter[20:19])
    2'b00:
        begin
            Anode_Activate=4'b0111;
            if(switch1==0 && switch2==0)
                LED_BCD=pcNext[7:4];
            else if(switch1==0 && switch2==1)
                LED_BCD={3'b000,rs[4:4]};
            else if(switch1==1 && switch2==0)
                LED_BCD={3'b000,rt[4:4]};
            else
```

```
                LED_BCD=result[7:4];
        end
    2'b01:
        begin
            Anode_Activate=4'b1011;
            if(switch1==0 && switch2==0)
                LED_BCD=pcNext[3:0];
            else if(switch1==0 && switch2==1)
                LED_BCD=rs[3:0];
            else if(switch1==1 && switch2==0)
                LED_BCD=rt[3:0];
            else
                LED_BCD=result[3:0];
        end
    2'b10:
        begin
            Anode_Activate=4'b1101;
            if(switch1==0 && switch2==0)
                LED_BCD=pcIn[7:4];
            else if(switch1==0 && switch2==1)
                LED_BCD=rsData[7:4];
            else if(switch1==1 && switch2==0)
                LED_BCD=rtData[7:4];
            else
                LED_BCD=DB[7:4];
        end
    2'b11:
        begin
            Anode_Activate=4'b1110;
            if(switch1==0 && switch2==0)
                LED_BCD=pcIn[3:0];
            else if(switch1==0 && switch2==1)
                LED_BCD=rsData[3:0];
            else if(switch1==1 && switch2==0)
                LED_BCD=rtData[3:0];
            else
                LED_BCD=DB[3:0];
        end
    endcase
end
```

显示部分：（在每个始终脉冲到来时，根据LED_BCD显示数字的二进制表示来得到LED_out的七段数码管显示信号）

```
always@(posedge clk)
begin
    case(LED_BCD)
        4'b0000: LED_out=7'b0000001;
        4'b0001: LED_out=7'b1001111;
        4'b0010: LED_out=7'b0010010;
        4'b0011: LED_out=7'b0000110;
        4'b0100: LED_out=7'b1001100;
        4'b0101: LED_out=7'b0100100;
        4'b0110: LED_out=7'b0100000;
        4'b0111: LED_out=7'b0001111;
        4'b1000: LED_out=7'b0000000;
        4'b1001: LED_out=7'b0000100;
        4'b1010: LED_out=7'b0001000;
        4'b1011: LED_out=7'b1100000;
        4'b1100: LED_out=7'b0110001;
        4'b1101: LED_out=7'b1000010;
        4'b1110: LED_out=7'b0110000;
        4'b1111: LED_out=7'b0111000;
        default: LED_out=7'b1111111;
    endcase
end
```

之后，在顶层模块中，加入对上述两个模块的实例化，并将PC模块、RegFile模块和RAM模块的时钟信号更改为work_clk即可

```
EliminationBuffet eliminationBuffet(
    .in_key(button_clk),
    .clk(clock),
    .out_key(work_clk)
);

Display display(
    .clk(clock),
    .switch1(switch1),
    .switch2(switch2),
    .pcNext(pcNext),
    .pcIn(pcIn),
    .rs(rs),
    .rsData(Data1),
    .rt(rt),
    .rtData(Data2),
    .result(result),
    .DB(DB),
```

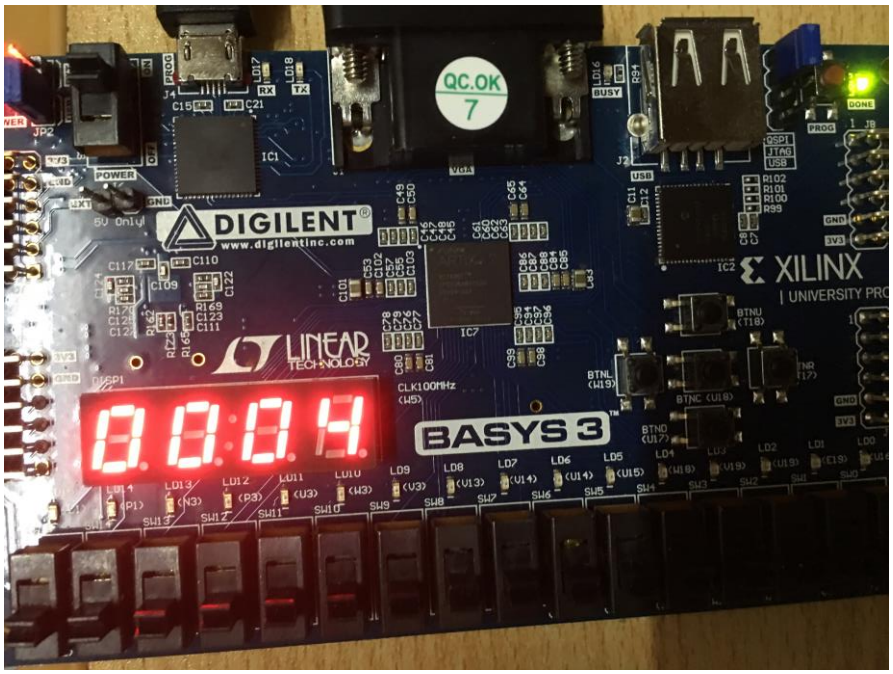
```
19.     .Anode_Activate(Anode_Activate),
20.     .LED_out(LED_out)
21. );
```

最终，添加约束文件，进行分配引脚即可。（引脚分配与下面代码类似）

```
1.  set_property PACKAGE_PIN W5 [get_ports clock]
2.      set_property IOSTANDARD LVCMOS33 [get_ports clock]
```

9．Basys3板显示效果如下：

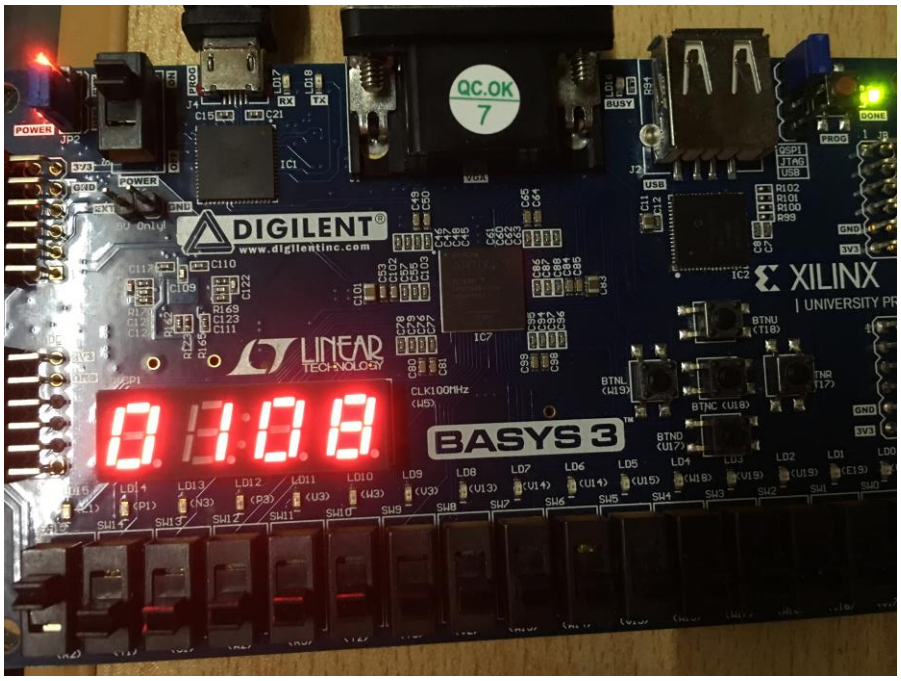
初始：



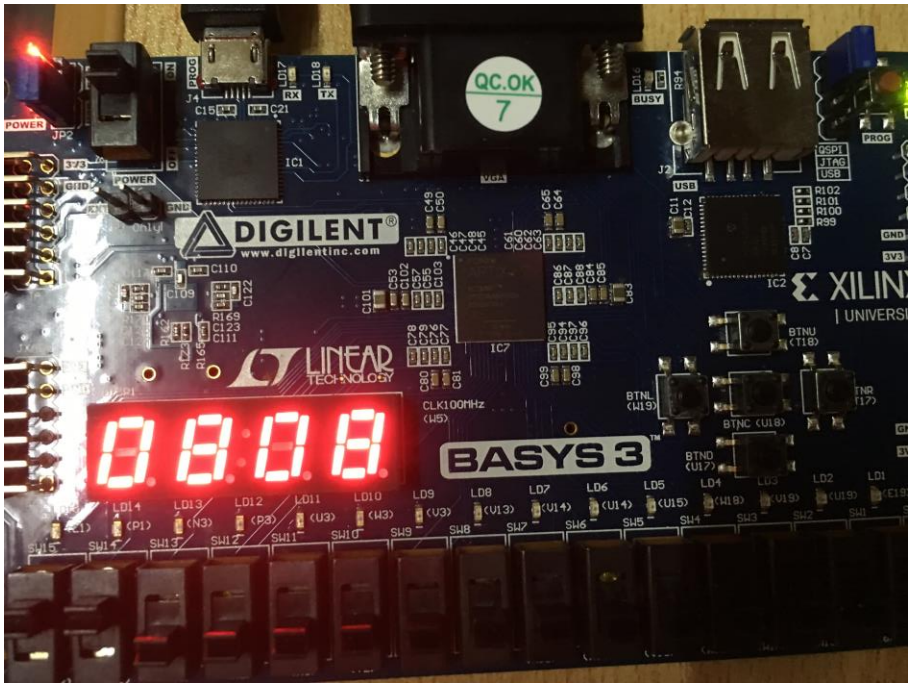
pc为0x00，下一条pc指令地址0x04

执行：

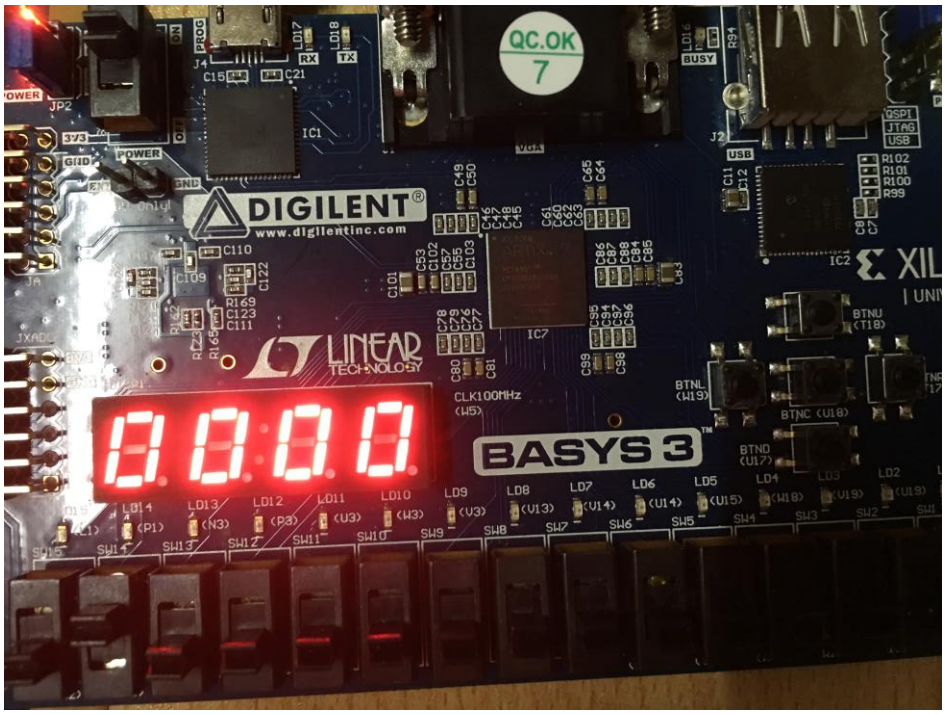
控制开关switch1和switch2，即最左侧两个开关，当switch1=1，switch2=0时，下图显示的是rt和rtData，以为第一条指令为addi $1,$0,8，所以rt即为$1,其数值为8，显示正确；



当switch1=1，switch2=1时，显示数值为ALUresult和DB：由于add计算结果为8，并且DBDataSrc在此时为0，所以输出正确；

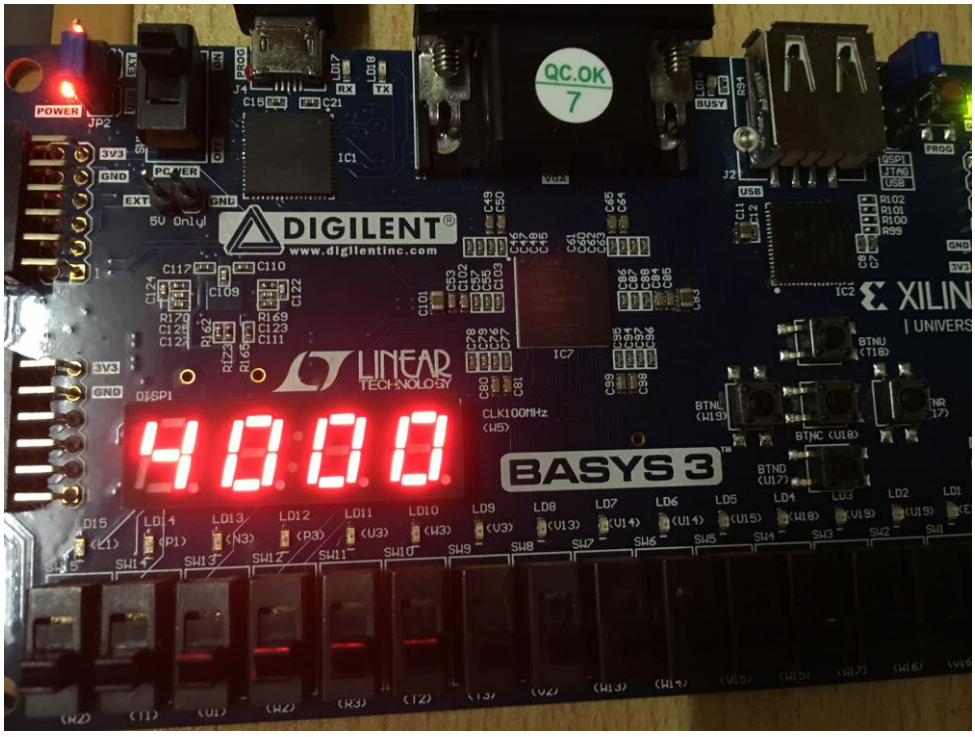


当switch1=0，switch2=1时，显示数值为rs和rsData：由于rs=$0，所以显示数值为0；



其他指令执行结果类似。

结束：pc为40，由于halt时我未对下一条指令赋值，所以默认显示为0x00



六. 实验心得

在这次实验中，通过自行设计单周期CPU，我对CPU的执行有了更加深刻的了解。在开始设计CPU之前，我只知道CPU由控制单元，逻辑运算单元，存储器和寄存器等组成，

并知道通过控制器的控制信号使得各个部件有机地结合在一起，但对其真正的内部构造，还有数据通路并没有清楚的认识。

在设计整个单周期的CPU的过程中，主要进行了以下阶段：1.对CPU指令解读；2.数据通路的设计；3.模块划分；4.设计模块；5.整合模块；6.进行仿真检验CPU设计的正确性。在课堂上，我们对CPU指令已经熟悉，并通过第一次实验中汇编代码的书写对其有了整体上的把握，对不同类型的指令的数据段划分也更加清楚，这让我们对CPU指令的判断更加准确。数据通路的设计是整个CPU设计的核心，设计过程中也需要进行模块划分，所以在设计之初，模块的划分就显得至关重要，在此次实验中，老师提供了数据通路图，所以我们掌握其数据传递方式和流程即可。

在模块划分中，我采用层次化、模块化的思想，先整体划分为控制模块，程序计数器模块，寄存器组模块，数据存储器模块，ALU模块等，然后在对其进行进一步划分和整合，细化每一个细节，逐步实现每个模块，然后一步一步整合即可得到整个CPU设计模块。

在实现的过程中，主要遇到了，逻辑混乱、触发信号的选择等问题，在Basys3板上实现时，还遇到了消抖的问题。 1. 逻辑混乱，主要是因为设计思路不够清晰，对数据通路图的理解不够到位,需偶要对数据通路图中每个细节都要了如指掌,才能成功执行每一条指令。数据通路图中，控制单元分发各种信号，每条指令在执行的每个阶段都在相应的模块内受不同的控制信号的约束，然后最终执行产生正确的结果，并进行存储等操作。理清楚逻辑之后问题就迎刃而解。 2. 触发信号的选择，关于时钟上升沿还是下降沿的选择，原则如下：指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC的改变是在时钟上升沿进行的，这样稳定性较好。 3. 消抖的处理，消抖是因为我们再点击按钮时，存在接触不良的情况，会产生一系列断开闭合的方波，为了得到稳定的方波，我采用延迟处理的方式，在多个时钟脉冲都是高电平时，才输出一个高电平信号。方法可能不太准确，但却是减小的抖动的影响。

未解决的问题：再写入Basys3板子上时，PC模块有以下两种写法：（只是判断顺序不同，并无本质上的区别）

```
always @(posedge CLK or negedge Reset) begin
    if (Reset == 1) begin
        if (PCWre == 1) begin
            pcNext <= pcIn;
        end
    end
    else
        pcNext <= 0;
    $display("##################");
end
```

此方法在仿真中可以正常运行，但写入到板子上是，就出现了 pc 值混乱的情况，出现莫名其妙的错误，无法正常执行。

```
always @(posedge CLK or negedge Reset) begin
    if (Reset == 0)
        pcNext <= 0;
    else if (PCWre == 1) begin
            pcNext <= pcIn;
        end
    $display("##################");
end
```

此方法可以正常执行。