《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 软件工程一（2）班
学 生 姓 名 ：	陈亚楠
学　　号 ：	16340041
时　　间 ：	2018　年　5　月　26　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**

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。时钟周期一般也称振荡周期（**如果晶振的输出没有经过分频就直接作为 CPU 的工作时钟，则时钟周期就等于振荡周期。若振荡周期经二分频后形成时钟脉冲信号作为 CPU 的工作时钟，这样，时钟周期就是振荡周期的两倍。**）

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。

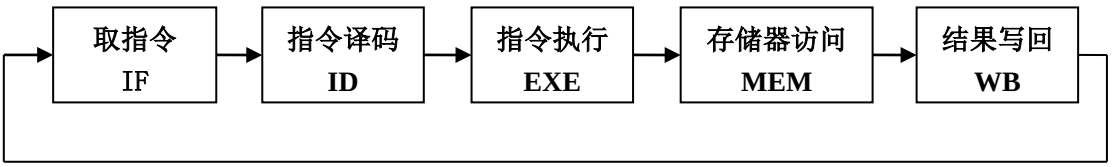
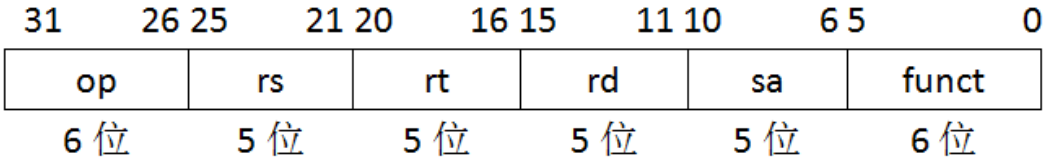


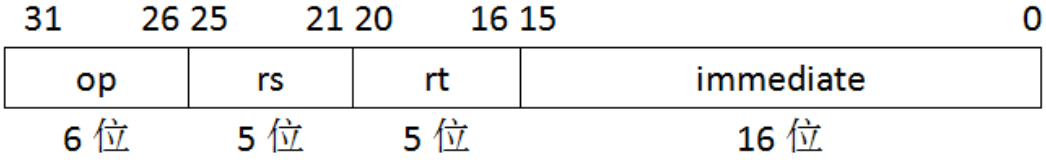
图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：

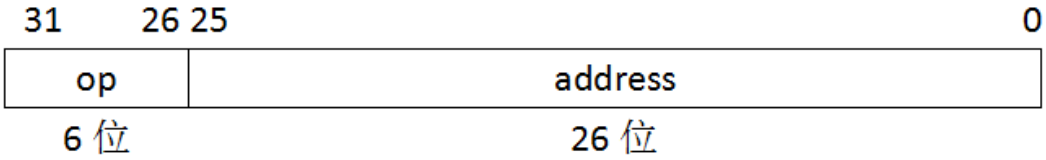
R 类型：



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。

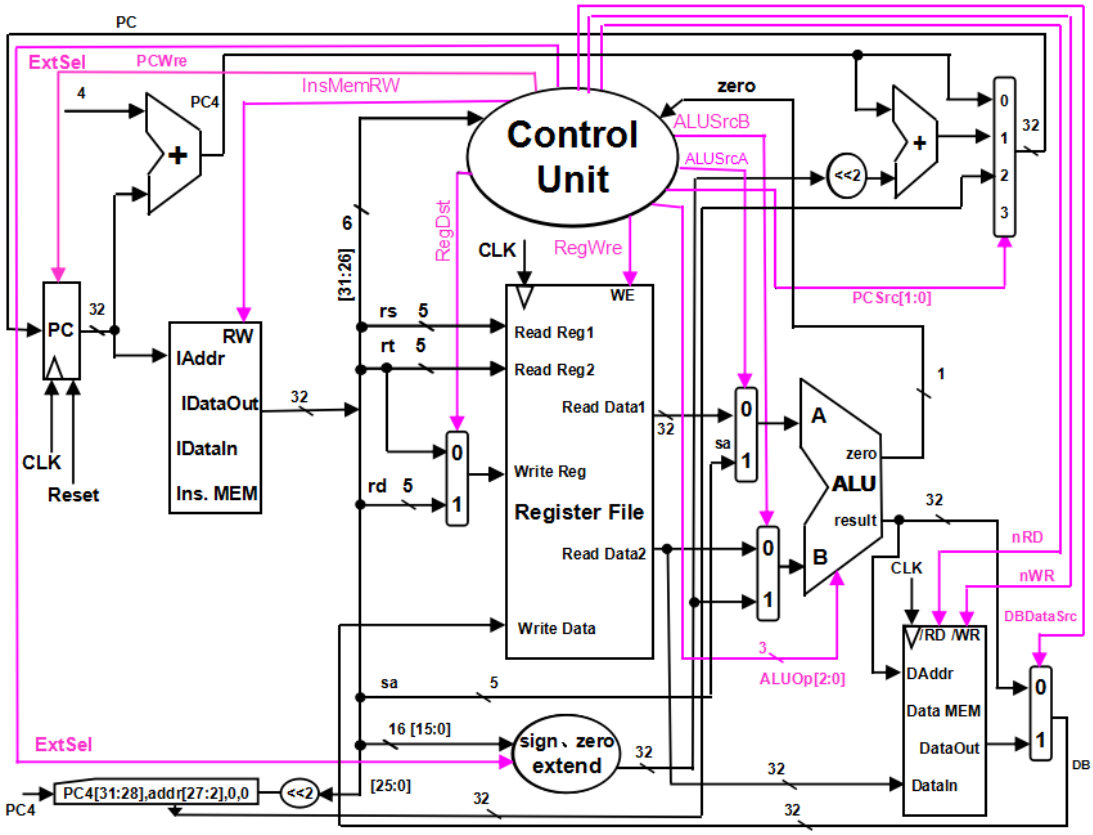


图 2　单周期 CPU 数据通路和控制线路图

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、slti、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
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)
nRD	输出高阻态	**读数据存储器，相关指令：lw**
nWR	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

/RD，数据存储器读控制信号，为 0 读

/WR，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

表 2 ALU 运算功能表

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

需要说明的是以上数据通路图是根据要实现的指令功能的要求画出来的，同时，还必须确定 ALU 的运算功能(当然，以上指令没有完全用到提供的 ALU 所有功能，但至少必须能实现以上指令功能操作)。从数据通路图上可以看出控制单元部分需要产生各种控制信号，当然，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出以上表 1，这样，从表 1 可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系表（留给学生完成），再根据关系表可以写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

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。另外，值得注意的问题，设计时，用模块化、层次化的思想方法设计，关于如何划分模块、如何整合成一个系统等等，是必须认真考虑的问题。

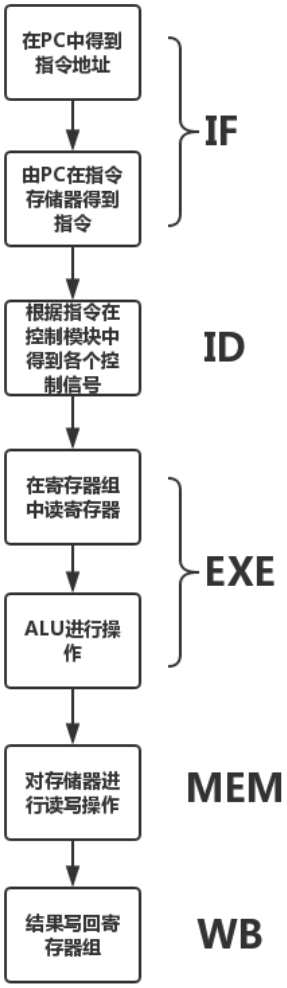
四. 实验器材

电脑一台，Xilinx Vivado 2018.1 软件一套，Basys3板一块。

五. 实验过程与结果

1、CPU设计的思想、方法：

（1）CPU的工作流程图：



（2）建立控制信号真值表：

控制信号表给出了本次实验中的每个控制信号的作用，我们可以由该表看出各个控制信号与指令之间的关系，并根据这种关系建立控制信号与指令之间的真值表：

表3

	op	ExtSel	PCWre	InsMemRW	ALUSrcA	ALUSrcB	PCSrc[1:0]
add	000000	1	1	0	0	0	00
addi	000001	1	1	0	0	1	00
sub	000010	1	1	0	0	0	00
ori	010000	0	1	0	0	1	00
and	01000	1	1	0	0	0	00

	1						
or	010010	1	1	0	0	0	00
sll	011000	1	1	0	1	0	00
slti	011011	1	1	0	0	1	00
sw	100110	1	1	0	0	1	00
lw	100111	1	1	0	0	1	00
beq(zero=0)	110000	1	1	0	0	0	00
beq(zero=1)	110000	1	1	0	0	0	01
bne(zero=0)	110001	1	1	0	0	0	01
bne(zero=1)	110001	1	1	0	0	0	00
j	111000	1	1	0	0	0	10
halt	111111	1	0	0	0	0	00

表3（续表）

	op	RegDst	RegWre	ALUOp[2:0]	nRD	nWR	DBDataSrc
add	000000	1	1	000	0	0	0
addi	000001	0	1	000	0	0	0
sub	000010	1	1	001	0	0	0
ori	010000	0	1	011	0	0	0
and	010001	1	1	100	0	0	0
or	010010	1	1	011	0	0	0
sll	011000	1	1	010	0	0	0
slti	011011	0	1	110	0	0	0
sw	100110	1	0	000	0	1	0
lw	100111	0	1	000	1	0	1
beq(zero=0)	110000	1	0	001	0	0	0
beq(zero=1)	110000	1	0	001	0	0	0
bne(zero=0)	110001	1	0	001	0	0	0

bne(zero=1)	110001	1	0	001	0	0	0
j	111000	1	0	000	0	0	0
halt	111111	1	0	000	0	0	0

（3）设计模块说明及其相关代码实现：

在本次实验中，根据数据通路图将CPU分为了总计12个模块，其中包括六个大模块和六个小模块，六个大模块分别为PC（程序计数器）、InstructionRegister（指令寄存器）、RegFile（寄存器组）、ALU（算术逻辑运算单元）、DataRegister（数据寄存器）、ControlUnit（控制单元）；六个小模块分别为Extend（立即数扩展）、PCSelect（计算下一个PC）、SelectALUDA（选择ALU的数据A输入）、SelectALUDB（选择ALU的数据B输入）、SelectDB（选择写回寄存器组的数据）、SelectWriReg（选择寄存组中的写寄存器）。

下面就各个模块的代码实现加以阐述：

① PC（程序计数器）

PC模块的输入有CLK（时钟）、Reset（初始化）、PCWre（PC变化的使能）、nextPC（计算得到的下一时刻的PC），该模块为上升沿触发，当Reset==0时初始化当前PC为0，否则PC=nextPC。

```
always @(posedge clk or negedge reset) begin
        if(reset == 1'b0)
            curPC <= 0; // 初始化 PC
        else if(PCWre)
            curPC <= nextPC; //下条指令的地址
end
```

② InstructionRegister（指令寄存器）

首先初始化指令存储器，将保存在.txt文件中的指令数据读入存储器中；当读使能信号有效时，则将存储器中的指令数据输出。

```
module InstructionRegister(
    input rd, input [31:0] iaddr, output reg [31:0] dataOut
    ); // 指令存储器模块，只读，用来存储指令
    // rd 读使能信号
    // iaddr 指令存储器指令地址输入端口
    // dataOut 指令存储器指令输出端口
    reg [31:0] dataOut;
    reg [7:0] rom [99:0]; // 存储器定义必须用 reg 类型，存储器存储单元 8 位长度，
共 100 个存储单元
```

```
    initial begin // 加载数据到存储器 rom，注意：必须使用绝对路径
        $readmemb ("D:/SingleCycleCPU/rom_data.txt", rom);
    end

    always @(rd or iaddr) begin
        if (rd == 0) begin // 为0，读存储器写入 dataOut。大端数据存储模式
        // 数据的高字节保存在存储器的低地址中
            dataOut[31:24] = rom[iaddr];
            dataOut[23:16] = rom[iaddr+1];
            dataOut[15:8] = rom[iaddr+2];
            dataOut[7:0] = rom[iaddr+3];
        end
    end
endmodule
```

③ ControlUnit（控制单元）

该模块的主要功能是根据操作码得到相应的各个控制信号的值，详情见表3。

下面代码仅展示一条指令作为说明（使用case语句方法逐个产生各指令控制信号）：

```
case (curOP)
     6'b000000 : begin //add
         RegWre = 1;
         RegDst = 1;
         ALUOp = 3'b000;
     End
endcase
```

④ RegFile（寄存器组）

在这一模块中，两个读数的寄存器ReadReg1和ReadReg2在时钟周期的前半个周期将rs和rt寄存器中的数读出；WriteReg在时钟周期的后半个周期经过ALU等一系列操作后将得到的数据写回寄存器。当控制信号RegWre==1表示有需要写回的数据，由SelectWriReg模块确定将需要写回的数据写回WriteReg（rt或rd）。

```
integer i;
    assign ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1]; // 读寄存器数据
    assign ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];
    always @ (negedge CLK or negedge RST)
    begin // 必须用时钟边沿触发
        if (RST==0) begin
            for(i=1;i<32;i=i+1)
                regFile[i] <= 0;
        end
        else if(RegWre == 1 && WriteReg != 0) // WriteReg != 0，$0 寄存器不能修改
             regFile[WriteReg] <= WriteData; // 写寄存器
```

```
end
```

⑤ DataRegister（数据寄存器）

该模块用于sw和lw这两个指令。定义ram用于储存数据，一个单位有8个寄存器。输入address为在ram进行操作的地址，nRD==1时为读，将ram中的数据写入Dataout中。nWR==1时为写，将writeData中的数据写入ram，时钟下降沿触发。

```
// 读
assign Dataout[7:0] = (nRD==1)?ram[address + 3]:8'bz;
assign Dataout[15:8] = (nRD==1)?ram[address + 2]:8'bz;
assign Dataout[23:16] = (nRD==1)?ram[address + 1]:8'bz;
assign Dataout[31:24] = (nRD==1)?ram[address ]:8'bz;
// 写
always@( negedge clk ) begin // 用电平信号触发写存储器，个例
    if( nWR==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
```

⑥ ALU（算术逻辑运算单元）

ALU总计八种操作，详见表2。操作数为rega和regb，sign为结果的符号位，当result为0时zero为1。

```
assign zero = (result==0)?1:0;
assign sign = result[31];
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; // 无符号比较
         3'b110 : begin // 带符号比较
         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;
     endcase
end
```

⑦ Extend（立即数扩展）

该模块为符号扩展或零扩展，根据输入的控制信号 ExtSel 判断要进行扩展类型；

零扩展的时候，立即数的高 16 为均为 0，符号扩展时，正数高 16 为均为 0，反之即为 1。

```
module Extend(
    input ExtSel,
    input [15:0] imme,
    output reg [31:0] extend_num
    );
    always@(ExtSel or imme)
    begin
        extend_num[15:0] = imme;
        if(imme[15]==0|ExtSel==0)
            extend_num[31:16]=16'h0000;
        else
            extend_num[31:16]=16'hffff;
    end
endmodule
```

⑧ PCSelect（计算下一个PC）

该模块根据输入的控制信号 PCSrc 的值计算下一条指令的地址：

一般情况下，大多数指令没有跳转，直接执行下一条指令，由于每一条指令占用4个指令寄存器，下一条指令的地址就等于当前指令的地址加4；当指令为beq、bne时，要跳转到的指令地址是4的倍数，最低两位是“00”，因此在将扩展的立即数放进指令码中的时候，要右移2位；当指令为j时，除了j指令的6位操作码外，addr就只有26位，因为PC一定能被4整除，后两位就默认为0，然后前四位则为下一指令的前四位，这样就导致了指令的跳转有一定的范围。

```
always@(PCSrc or curPC or extend_imme or addr) begin
        PC0 = curPC + 4;
        PC1 = (extend_imme << 2)+ PC0;
        PC2[31:28] = PC0[31:28];
        PC2[27:2] = addr[25:0];
        PC2[1:0] = 2'b00;
        case(PCSrc)
            2'b00 :  nextPC = PC0;
            2'b01 :  nextPC = PC1;
            2'b10 :  nextPC = PC2;
            2'b11 : nextPC = 32'bz;
        endcase
end
```

⑨ SelectALUDA（选择ALU的数据A输入）

根据控制信号 ALUSrcA 选择从寄存器组中读到的 Read_data1 和 sa 中的一个为 ALU 的操作数 A。

```
always@(sa or Read_data1 or ALUSrcA)
     if(ALUSrcA)begin
          rega[4:0] = sa;
```

```
            rega[31:5] = 27'b0;
        end
        else
            rega = Read_data1;
```

⑩ SelectALUDB（选择ALU的数据B输入）

根据控制信号ALUSrcB选择从寄存器组中读到的Read_data2和imme_num中的一个为ALU的操作数B。

```
always@(Read_data2 or imme_num or ALUSrcB)
        if(ALUSrcB)
            regb = imme_num;
        else
            regb = Read_data2;
```

⑪ SelectDB（选择写回寄存器组的数据）

根据不同的指令得到 DBDataSrc，再决定写回寄存器组的数据是来自于 ALU 还是内存。

```
always@(DBDataSrc or ALU_result or Mem_dataout)
if(DBDataSrc)
    DB = Mem_dataout;
else
    DB = ALU_result;
```

⑫ SelectWriReg（选择寄存组中的写寄存器）

该模块根据不同指令得到的RegDst选择写回的数据写入rt还是rd。

```
always@(RegDst or rt or rd)begin
    if(RegDst)
        WriReg = rd;
    else
        WriReg = rt;
end
```

2、验证CPU的正确性：

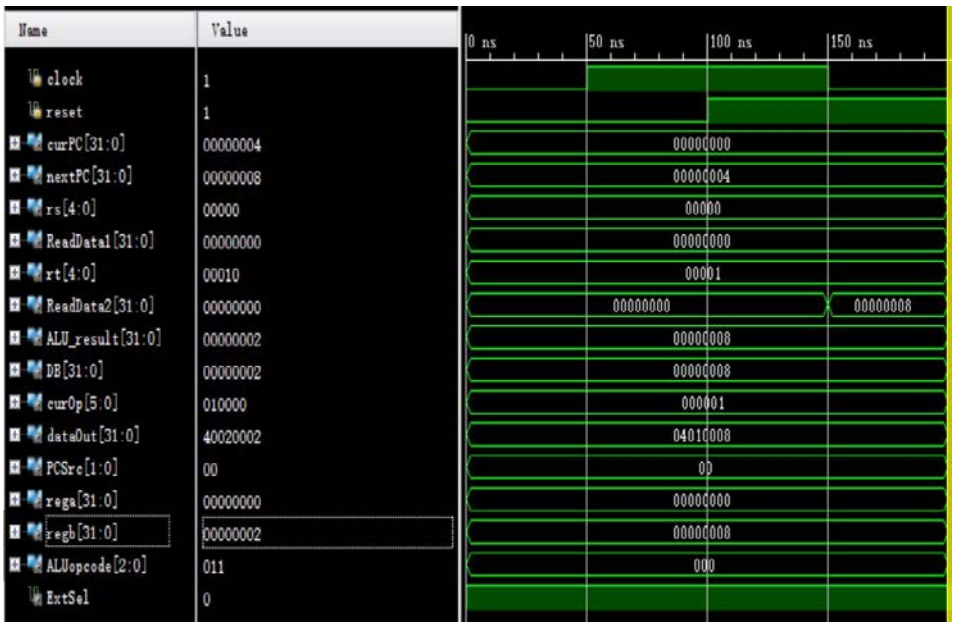
（1）测试程序段：

地址	汇编程序	指令代码					
		op (6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码	
0x00000000	addi $1,$0,8	000001	00000	00001	0000 0000 0000 1000	=	04010008
0x00000004	ori $2,$0,2	010000	00000	00010	0000 0000 0000 0010	=	40020002
0x00000008	add $3,$2,$1	000000	00010	00001	0001 1000 0000 0000	=	00410C00
0x0000000C	sub $5,$3,$2	000010	00011	00010	0010 1000 0000 0000	=	08622800

0x00000010	and $4,$5,$2	010001	00101	00010	0010 0000 0000 0000	=	44A22000
0x00000014	or $8,$4,$2	010010	00100	00010	0100 0000 0000 0000	=	48824000
0x00000018	sll $8,$8,1	011000	00000	01000	0100 0000 0100 0000	=	60084040
0x0000001C	bne $8,$1,-2 (≠, 转 18)	110001	01000	00001	1111 1111 1111 1110	=	C501FFFE
0x00000020	slti $6,$2,8	011011	00010	00110	0000 0000 0000 1000	=	6C460008
0x00000024	slti $7,$6,0	011011	00110	00111	0000 0000 0000 0000	=	6CC70000
0x00000028	addi $7,$7,8	000001	00111	00111	0000 0000 0000 1000	=	04E70008
0x0000002C	beq $7,$1,-2 (=, 转 28)	110000	00111	00001	1111 1111 1111 1110	=	C0E1FFFE
0x00000030	sw $2,4($1)	100110	00001	00010	0000 0000 0000 0100	=	98220004
0x00000034	lw $9,4($1)	100111	00001	01001	0000 0000 0000 0100	=	9C290004
0x00000038	**j 0x00000040**	111000	00000	00000	0000 0000 0001 0000	=	E0000040
0x0000003C	addi $10,$0,10	000001	00000	01010	0000 0000 0000 1010	=	040A000A
0x00000040	**halt**	111111	00000	00000	0000000000000000	=	FC000000

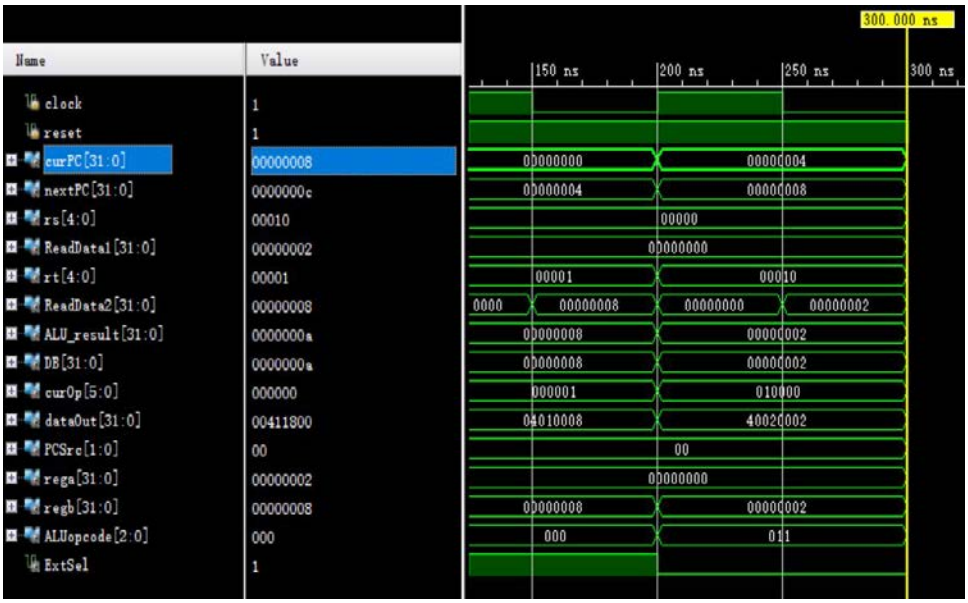
（2）指令检查：

① addi $1,$0,8:



当前的PC=00000000，PCSrc=00，下一时期的PC即nextPC=00000004，rs为$0，rt为$1，ReadData1为$0中读的数0，ReadData2为立即数8，ALU的两个操作数rega和regb分别为0和8，ALU的计算结果为8，写回寄存器的值DB也为8，将结果写回$1中，所以在下降沿时$1由0变成了8。此时的立即数扩展为sign_extend，故ExtSel=1，测试第一步正确。

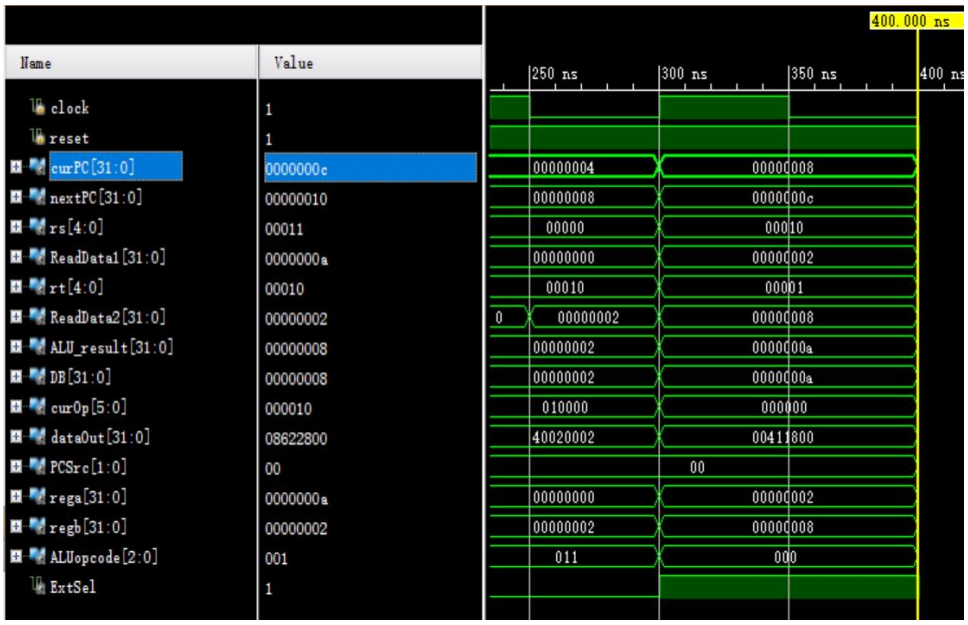
② ori $2,$0,2:



当前的PC=00000004，PCSrc=00，下一时期的PC即nextPC=00000008，rs为$0，rt为$2，ReadData1为$0中读的数0，ReadData2为立即数2，ALU的两个操作数rega和regb分别为0和2，ALU的计算结果为2，写回寄存器的值DB也

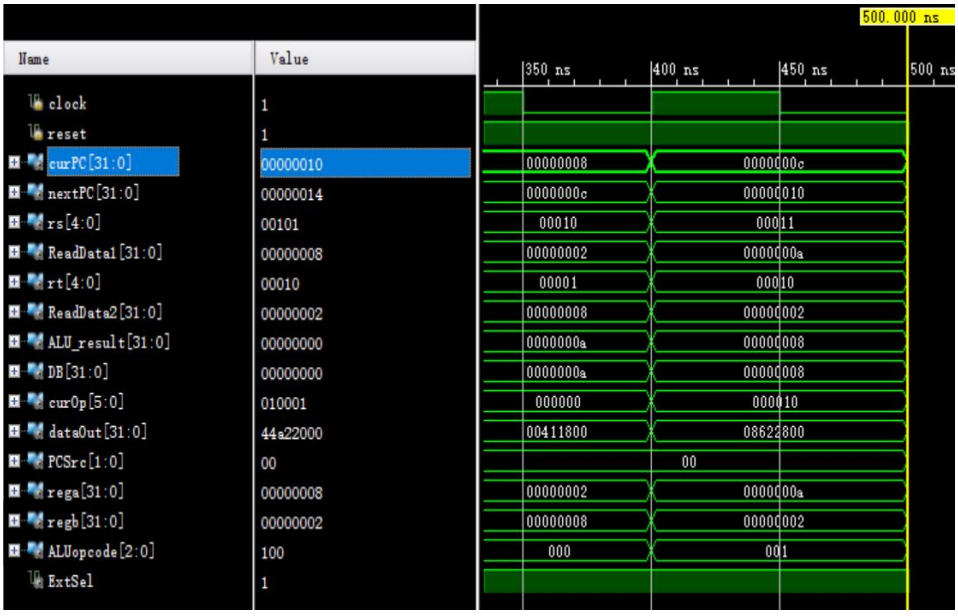
为2，将结果写回$2中，所以在下降沿时$2由0变成了2。值得注意的是，在这一步有立即数扩展，且ori为zero_extend，此时ExtSel=0。测试第二步正确。

③ add $3,$2,$1:



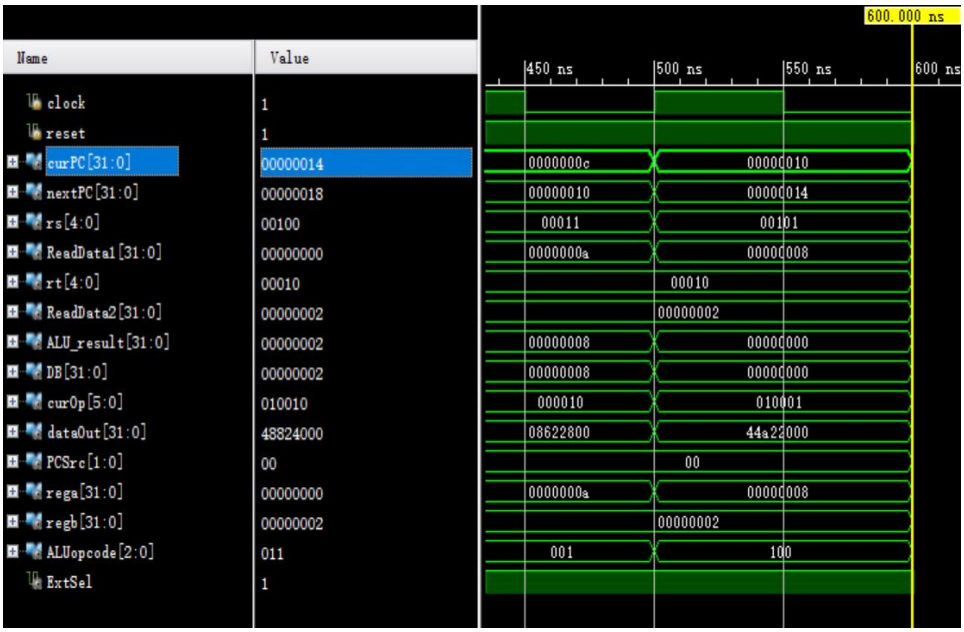
当前的PC=00000008，PCSrc=00，下一时期的PC即nextPC=0000000c，rs为$2，rt为$1，ReadData1为$2中读的数2，ReadData2为为$1中读的数8，ALU的两个操作数rega和regb分别为2和8，ALU的计算结果为10，写回寄存器的值DB也为10，此时写回的寄存器是rd而不是rt，故rd并没有在下降沿变化。测试第三步正确。

④ sub $5,$3,$2:



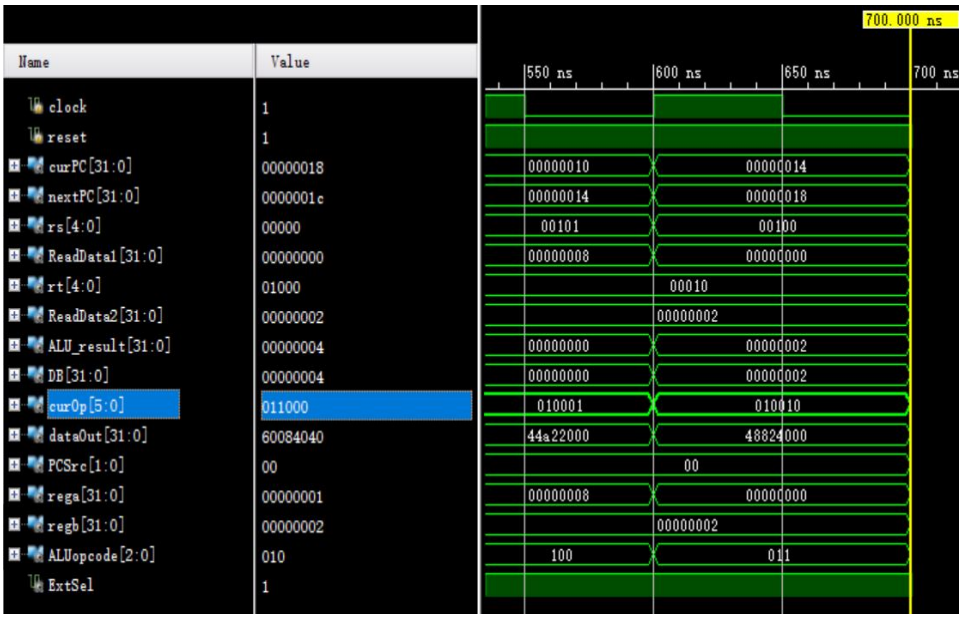
当前的PC=0000000c，PCSrc=00，下一时期的PC即nextPC=00000010，rs为$3，rt为$2，ReadData1为$3中读的数10，ReadData2为$2中读的数2，ALU的两个操作数rega和regb分别为10和2，ALU的计算结果为8，写回寄存器的值DB也为8，测试第四步正确。

⑤ and $4,$5,$2:



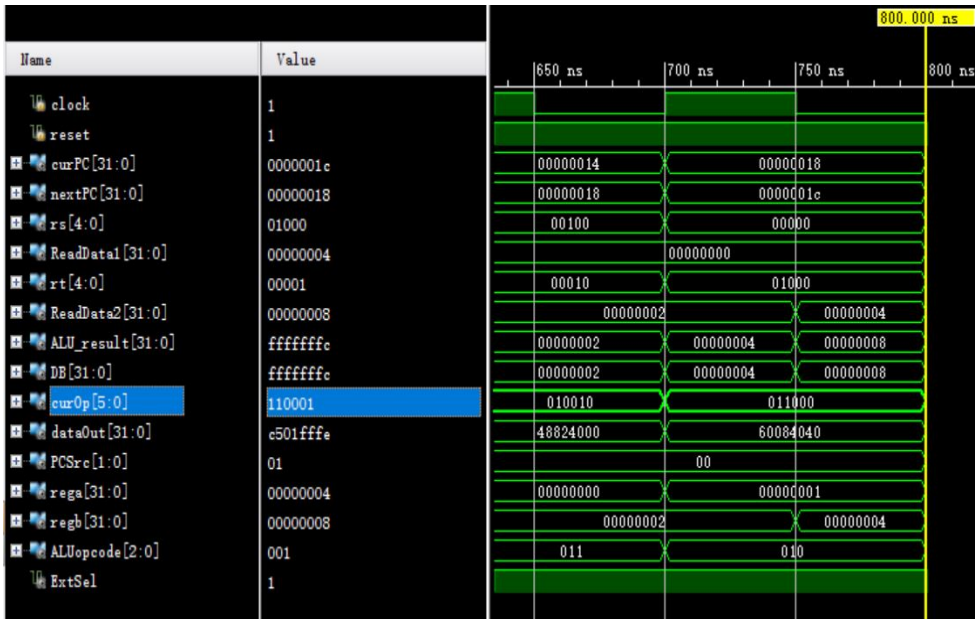
当前的PC=00000010，PCSrc=00，下一时期的PC即nextPC=00000014，rs为$5，rt为$2，ReadData1为$5中读的数8，ReadData2为$2中读的数2，ALU的两个操作数rega和regb分别为8和2，ALU的计算结果为0，写回寄存器的值DB也为0，测试第五步正确。

⑥ or $8,$4,$2:



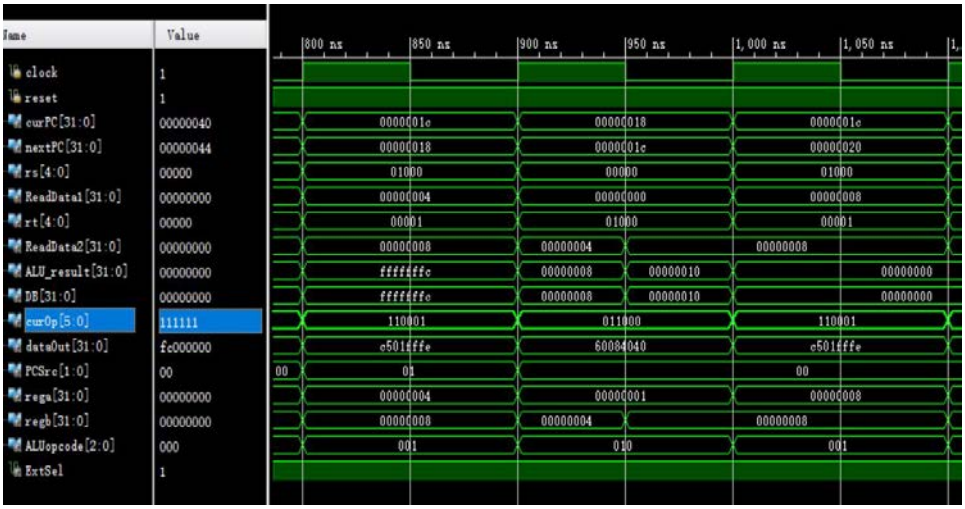
当前的PC=00000014，PCSrc=00，下一时期的PC即nextPC=00000018，rs为$4，rt为$2，ReadData1为$5中读的数0，ReadData2为$2中读的数2，ALU的两个操作数rega和regb分别为0和2，ALU的计算结果为2，写回寄存器的值DB也为2，测试第六步正确。

⑦ sll $8,$8,1:



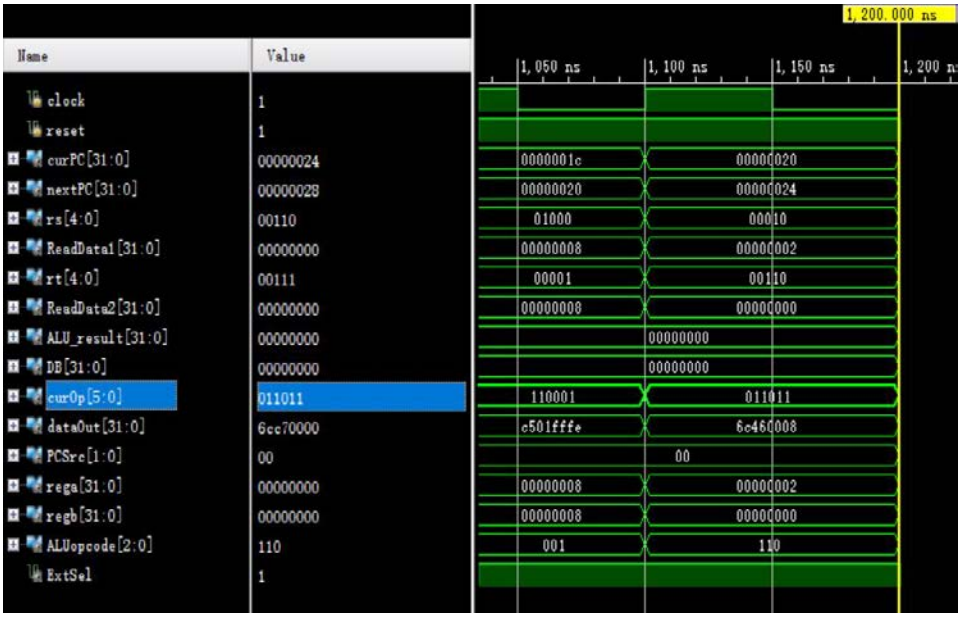
当前的PC=00000018，PCSrc=00，下一时期的PC即nextPC=0000001c，rd为$8，rt为$8，ReadData1为未用的数0，ReadData2为$8中读到的数2，ALU的两个操作数rega和regb分别为1和2，ALU的操作为左移，计算结果为4，写回寄存器的值DB也为4，测试第七步正确。

⑧ bne $8,$1,-2 (≠,转18):



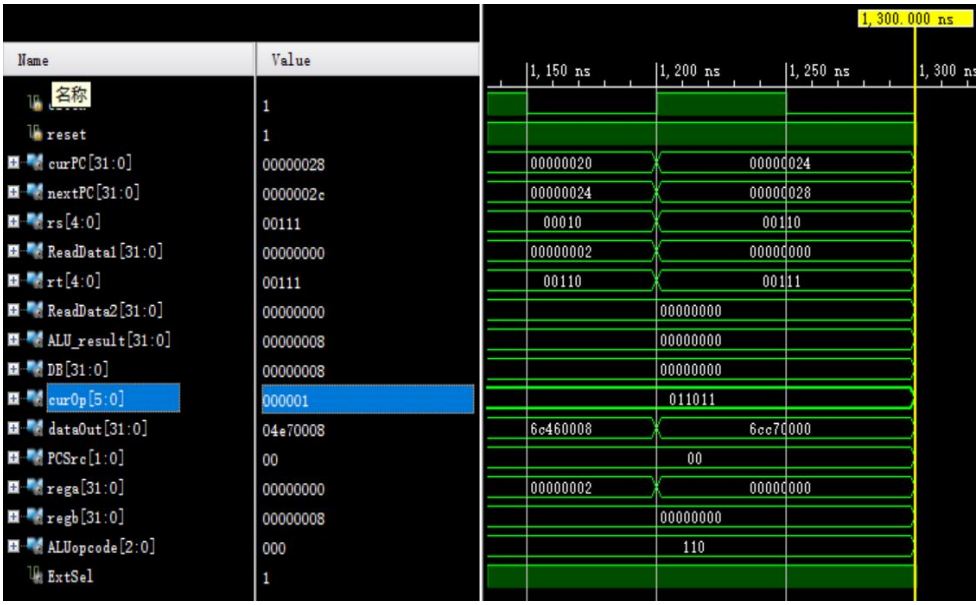
当前的PC=0000001c，PCSrc=01，下一时期的PC即nextPC=00000018，rs为$8，rt为$1，ReadData1为$8用的数4，ReadData2为$1中读到的数8，ALU的两个操作数rega和regb分别为4和8，此时两个数不相等，PC跳转至PC=00000018处，执行第七步，此时ALU的两个操作数rega和regb分别为8和8，两数相等，测试第八步正确。

⑨ slti $6,$2,8:



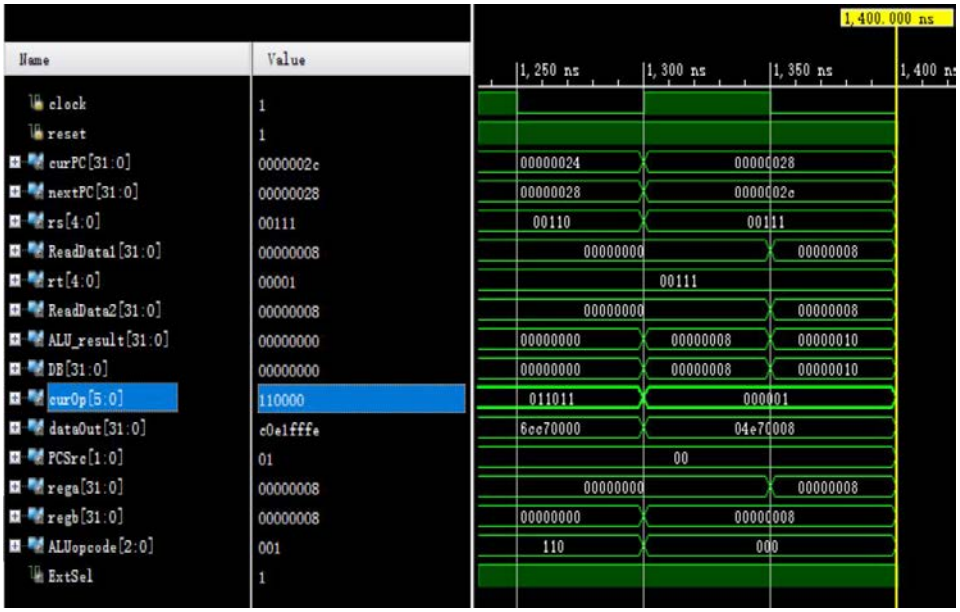
当前的PC=00000020，PCSrc=00，下一时期的PC即nextPC=00000024，rs为$2，rt为$6，写入$6的值为1，测试第九步正确。

⑩ slti $7,$6,0:



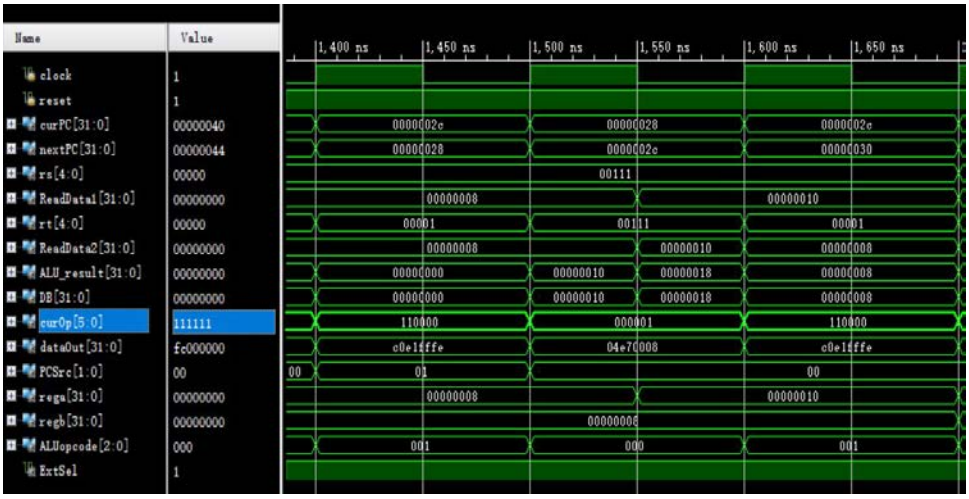
当前的PC=00000024，PCSrc=00，下一时期的PC即nextPC=00000028，rs为$6，rt为$7，写入$7的值为0，测试第十步正确。

⑪ addi $7,$7,8:



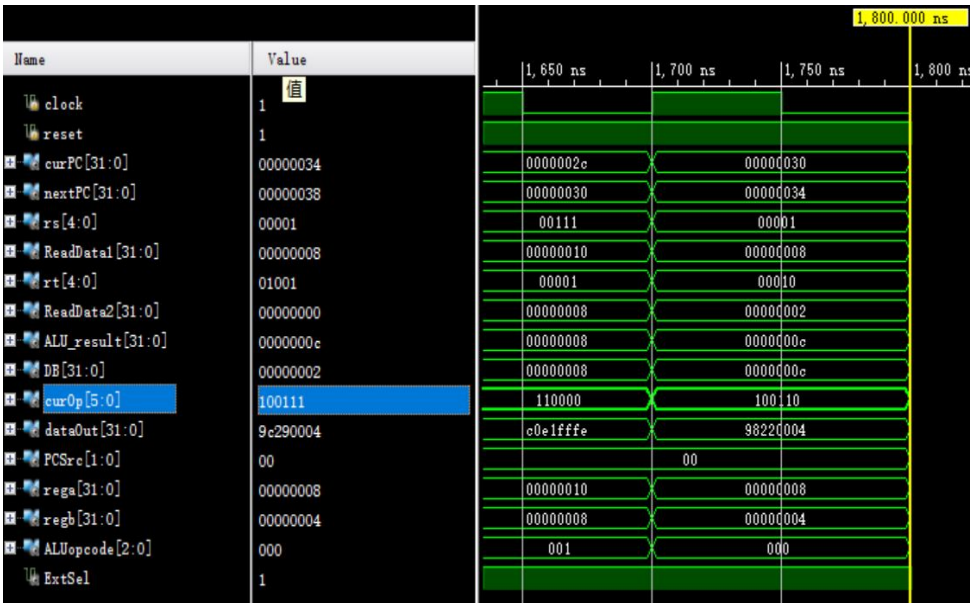
当前的PC=00000028，PCSrc=00，下一时期的PC即nextPC=0000002e，rs为$7，rt为$7，ReadData1为$7中读的数8，ReadData2为立即数8，ALU的两个操作数rega和regb分别为8和8，ALU的计算结果为16，写回寄存器的值DB也为16，将结果写回$7中，所以在下降沿时$1由0变成了16。此时的立即数扩展为sign_extend，故ExtSel=1，测试第十一步正确。

⑫ beq $7,$1,-2 (=,转28):



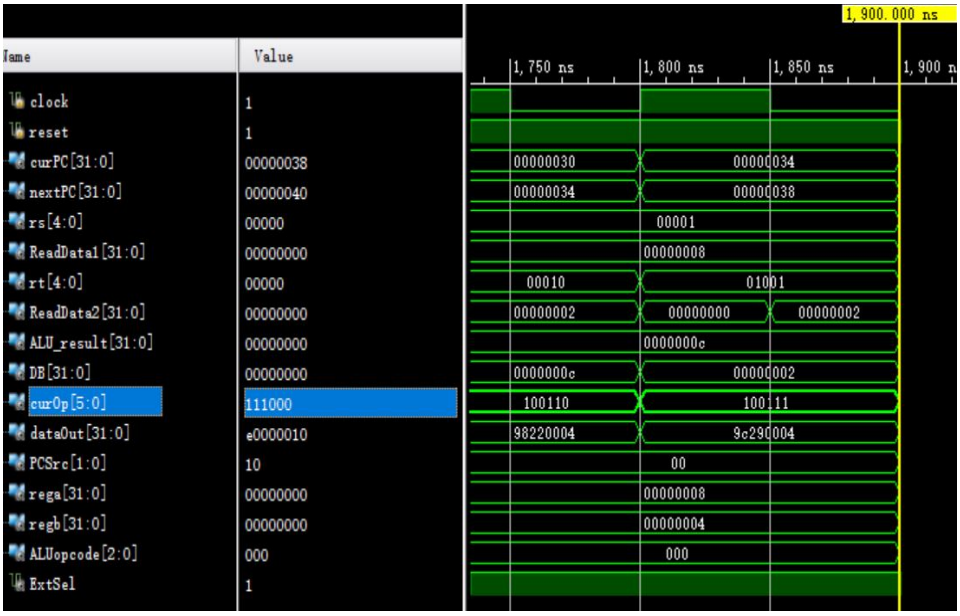
当前的PC=0000002c，PCSrc=01，下一时期的PC即nextPC=00000028，rs为$7，rt为$1，ReadData1为$7中读出的数8，ReadData2为$1中读出的数8，ALU的两个操作数rega和regb为8和8，8==8，此时ALU的运算结果为1，调转到00000028，测试第十二步正确。

⑬ sw $2,4($1):



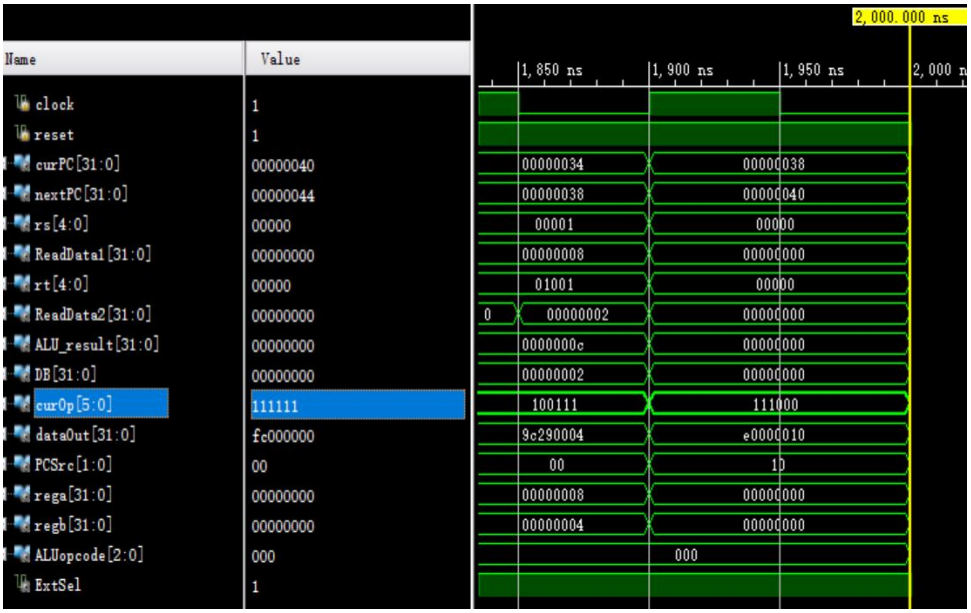
当前的PC=00000030，PCSrc=00，下一时期的PC即nextPC=00000034，rs为$1，rt为$2，立即数imme为4，且为sign_extend，ReadData1为$1中读出的数8，ReadData2为$2中读出的数2，ALU的两个操作数rega和regb为$1中的数8和立即数4，ALU计算结果为12，即在内存地址为12的地方存入$2中的2，测试第十三步正确。

⑭ lw $9,4($1):



当前的PC=00000034，PCSrc=0，下一时期的PC即nextPC=00000038，rs为$1，rt为$9，立即数imme为4，且为sign_extend，ReadData1为$1中读出的数8，ReadData2为$9中读出的数0，ALU的两个操作数rega和regb为$1中的数8和立即数4，ALU计算结果为12，即在存储器中的位置为12的地方的数存入$9中，在时钟周期后半个周期2写入$9中，测试第十四步正确。

⑮ j 0x00000040:

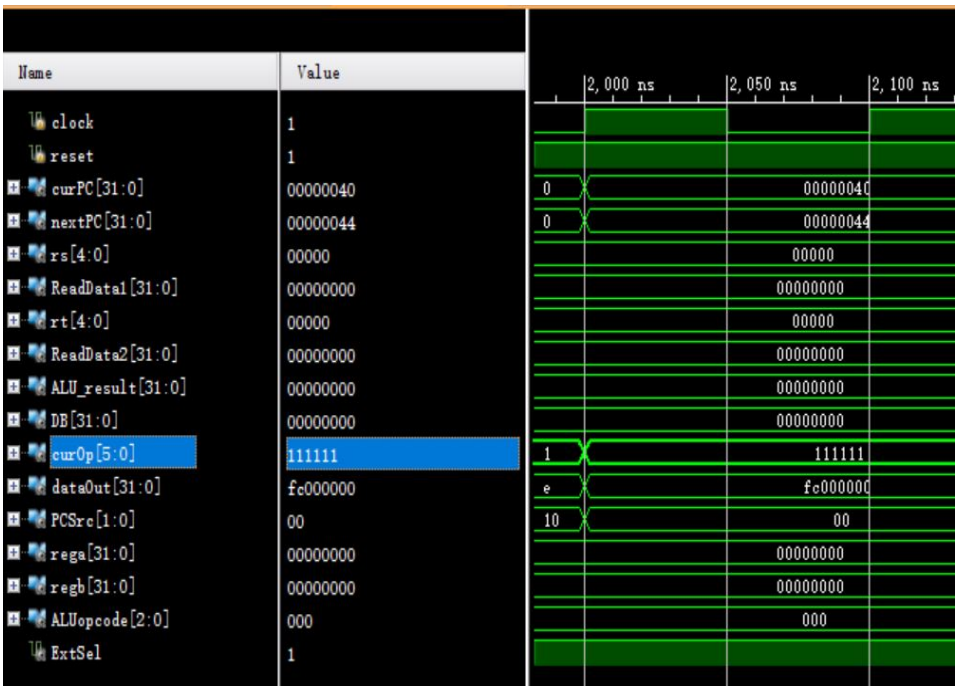


当前的PC=00000038，下一时期的PC即nextPC=00000040，PCSrc=10，测试第十五步正确。

⑯ addi $10,$0,10:

无。

⑰ halt:



当前的PC=00000040，操作码是111111，此时PC停止在00000040，其他所有单元全部暂停。

六. 实验心得

1．总体了解了单周期cpu的整个工作过程，理解了CPU的IF、ID、EXE、MEM、WB这五个过程在CPU的哪些位置完成，如何完成。IF是取指令，根据当前的PC在指令存储器得到指令。ID是指令译码，从指令中得到相应的信息，比如一些控制信号，比如PCSrc、ExtSel等。EXE是执行，主要是在读寄存器和ALU中完成一些操作，对于不同的指令操作对应着ALU中 8 种不同的操作。MEM是储存器访问，用于lw和sw，在数据存储器中完成。WB是写回数，写回的数有的是ALU计算的结果，有的是在内存中读出的数，写的寄存器根据不同的指令可能是rs或rt寄存器。

2．熟悉掌握了MIPS三种指令的三种格式，并且大致了解了数据存储器、指令存储器和PC它们的之间分配和工作，根据数据通路和控制线路图，熟悉PC、指令存储器、ALU、数据存储器等的输入和输出，在完成这些大模块之后，还有一些简单的选择模块，用来减轻这些模块的负担，最后就是控制模块。要完成控制模块首先就是要弄清楚控制模块的各种控制信号，控制单元除了sign和zero就只有操作码，根据操作码确定是哪个功能，再依次确定各

个操作的控制信号，这需要对数据通路和控制线路图有足够的熟悉。

3. 对于vivado软件的使用也更加熟练，掌握了debug的比较有效的方法以及有效观察波形的办法，对于仿真文件和顶层文件也有了属于自己的理解。

4．对于时钟触发，根据资料的查找和老师上课的讲述，PC是时钟上升沿触发，寄存器组的写操作和内存的写操作都是时钟下降沿触发，在上半个周期完成IF、ID、EXE和MEM中的读，MEM的写和WB阶段在下半个周期完成，这样就能保证一条指令在一个周期完成，不然可能存在一条指令重复执行。

5．在实验中，首先遇到的问题就是reg、wire、assign以及always之间错综复杂的关系。经过总结，可以得到wire用于assign，reg用于initial和always的结论。还有在写顶层文件的时候，我的很多变量都是reg，会报错，百度之后才知道，在模块外面变量的申明都应该为wire，在模块外面，变量都被封装起来，就只留下输入输出作为端口，连接端口的只能是wire，不可以reg。

6．再次体会到了模块化、层次化设计思想的重要性，以及简单源于规整的设计原则。