中山大學
SUN YAT-SEN UNIVERSITY

《计算机组成原理实验》

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）：	19 级计科 1 班
学 生 姓 名 ：	郝裕玮
学　　号 ：	18329015
时　　间 ：	2020　年　11　月　25　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。

二. 实验内容

1.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)	00000 100000

功能：GPR[rd] ← GPR[rs] + GPR[rt]。

（2）sub rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100010

功能：GPR[rd] ← GPR[rs] - GPR[rt]。

（3）addiu rt , rs ,**immediate**

001001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] + sign_extend(**immediate**)； **immediate** 做符号扩展再参加“与”运算。

==> 逻辑运算指令

（4）andi rt , rs ,**immediate**

001100	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] and zero_extend(**immediate**)；**immediate** 做 0 扩展再参加“与”运算。

（5）and rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100100

功能：GPR[rd] ← GPR[rs] and GPR[rt]。

（6）ori rt , rs ,**immediate**

001101	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] or zero_extend(**immediate**)。

（7）or rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100101

功能：GPR[rd] ← GPR[rs] or GPR[rt]。

==>移位指令

（8）sll　rd, rt,sa

000000	00000	rt(5 位)	rd(5 位)	sa(5 位)	000000

功能：GPR[rd] ← GPR[rt] << sa。

==>比较指令

（9）　slti　rt, rs,**immediate**　　带符号数

001010	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if GPR[rs] < sign_extend(**immediate**)　GPR[rt] =1 else　GPR[rt] = 0。

==> 存储器读/写指令

（10）sw　rt , **offset** (rs) 写存储器

101011	rs(5 位)	rt(5 位)	**offset**(16 位)

功能：memory[GPR[base] + sign_extend(**offset**)] ← GPR[rt]。

（11) lw　rt , **offset** (rs) 读存储器

100011	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：GPR[rt] ← memory[GPR[base] + sign_extend(**offset**)]。

==> 分支指令

（12）beq　rs,rt, **offset**

000100	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：if(GPR[rs] = GPR[rt]) pc←pc + 4 + sign_extend(**offset**)<<2

else pc ←pc + 4

特别说明：offset 是从 PC+4 地址开始和转移到的指令之间指令条数。offset 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 **offset** 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”。

（13）bne　rs,rt, **offset**

000101	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：if(GPR[rs] != GPR[rt]) pc←pc + 4 + sign_extend(**offset**) <<2

else pc ←pc + 4

（14）bltz　rs, **offset**

000001	rs(5 位)	00000	**offset** (16 位)

功能：if(GPR[rs] < 0) pc←pc + 4 + sign_extend (**offset**) <<2

else pc ←pc + 4。

==>跳转指令

（15）j　addr

000010	**addr**(26 位)

功能：PC ← {PC[31:28] , **addr** , 2’b0}，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，剩下最高 4 位由 pc+4 最高 4 位拼接上。

==> 停机指令

（16）halt

111111	00000000000000000000000000(26 位)

功能：停机；不改变 PC 的值，PC 保持不变。

2.实验要求

(1)PC 和寄存器组写状态使用时钟边缘触发。

(2)指令存储器和数据存储器存储单元宽度一律使用 8 位，即一个字节的存储单位。不能使用 32 位作为存储器存储单元宽度。

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

(4)必须按统一测试用的汇编程序段进行测试所设计的 CPU。

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

三. 实验原理

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。

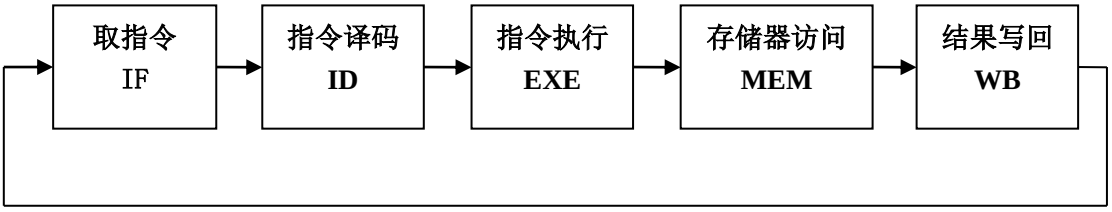
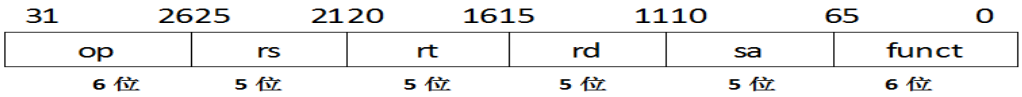


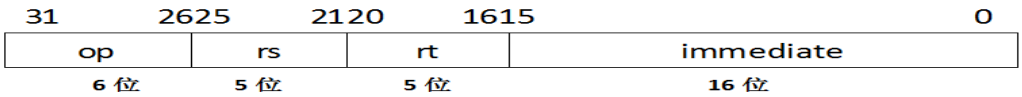
图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：

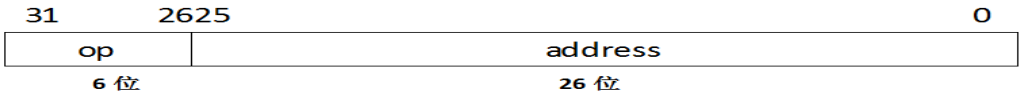
R 类型：



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（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。

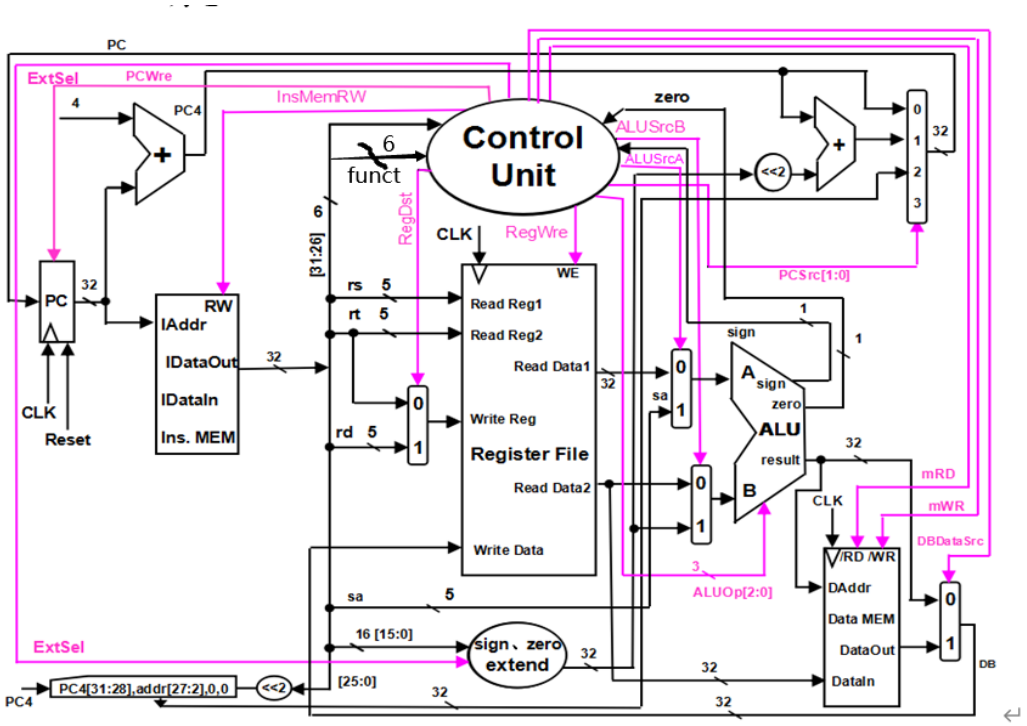


图 2　单周期 CPU 数据通路和控制线路图

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，读操作时不需要时钟信号，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、addiu、or、and、andi、ori、slti、beq、bne、bltz、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{1'b0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、beq、bne、bltz	来自 sign 或 zero 扩展的立即数，相关指令：addi、andi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addiu、sub、ori、or、and、andi、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、bltz、sw、halt	寄存器组写使能，相关指令：add、addiu、sub、ori、or、and、andi、

		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：addiu、andi、ori、slti、lw	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、sll
ExtSel	(zero-extend)**immediate(**0 扩展**)**，相关指令：addiu、andi、ori	(sign-extend)**immediate (**符号扩展**)**，相关指令：slti、sw、lw、beq、bne、bltz
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addiu、sub、or、ori、and、andi、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0)； 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bne(zero=0)、bltz(sign=1)； 10：pc<－{(pc+4)[31:28],addr[27:2],2'b00}，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

sign，运算结果标志，结果最高位为 0，则 sign=0，正数；否则，sign=1，负数

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，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。

控制信号与指令之间的关系表如下表 3 所示。

表 3 ALU 控制信号与指令关系表

指令	Reset	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	InsMemRW
add	x	1	0	0	0	1	1
sub	x	1	0	0	0	1	1
addiu	x	1	0	1	0	1	1
andi	x	1	0	1	0	1	1
and	x	1	0	0	0	1	1
ori	x	1	0	1	0	1	1
or	x	1	0	0	0	1	1
sll	x	1	1	0	0	1	1
slti	x	1	0	1	0	1	1
sw	x	1	0	1	0	0	1
lw	x	1	0	1	1	1	1
beq	x	1	0	0	0	0	1
bne	x	1	0	0	0	0	1
bltz	x	1	0	0	0	0	1
j	x	1	0	0	0	0	1
halt	x	0	0	0	0	0	1

指令	mRD	mWR	RegDst	ExtSel	PCSrc[1…0]	ALUOp[2…0]
add	0	0	1	1	00	000
sub	0	0	1	1	00	001
addiu	0	0	0	1	00	000
andi	0	0	0	0	00	100
and	0	0	1	1	00	100
ori	0	0	0	0	00	011
or	0	0	1	1	00	011
sll	0	0	1	1	00	010
slti	0	0	0	1	00	110
sw	0	1	1	1	00	000
lw	1	0	0	1	00	0000
beq	0	0	1	1	0(zero==0?0:1)	111
bne	0	0	1	1	0(zero==1?0:1)	111
bltz	0	0	1	1	0(sign==0?0:1)	000
j	0	0	1	1	10	000
halt	0	0	1	1	xx	000

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、 设计思想

在设计的过程中使用模块化设计的思想，逐个模块设计，并使某些模块可以共用，从而减少工作量。

2、 CPU设计过程

设计CPU的基本思想是自顶向下，分模块实现。首先我们根据数据通路图（图2）把数据通路的每个模块功能所需的输入，输出端口进行统一。之后分别实现各个模块，最后用顶层文件对各个模块进行连接即可。

3、代码分析

Basys3.v

最顶层文件，用于连接CPU、remove_shake、 display三个模块，并能调节显示模式。

```
`timescale 1ns / 1ps

module Basy3(
    input[1:0] display_mode,
    input CLK, Reset, Button,
```

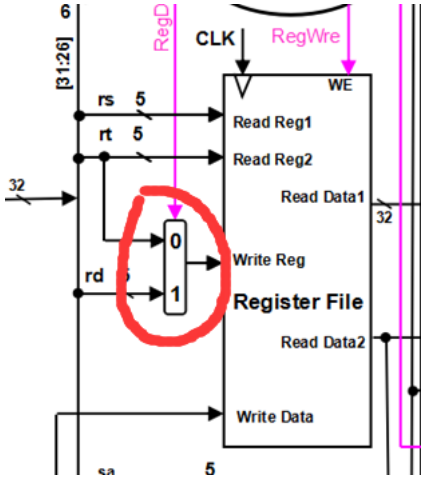
```
    output[3:0] AN,
    output[7:0] Out
);

    wire[31:0] ALUresult, curPC, WriteData,ReadData1,ReadData2,instruct
ion,nextPC;
    wire myCLK, myReset;
    reg[3:0] store;

    CPU cpu(myCLK,myReset,op,funct,ReadData1,ReadData2,curPC,nextPC,ALU
result,instruction,WriteData);
    remove_shake remove_shake_clk(CLK, Button, myCLK);
    remove_shake remove_shake_reset(CLK, Reset, myReset);
    clk_slow slowclk(CLK, myReset, AN);
    display show_in_7segment(store, myReset, Out);

    always@(myCLK)begin
       case(AN)
            4'b1110:begin
                case(display_mode)
                    2'b00: store <= nextPC[3:0];
                    2'b01: store <= ReadData1[3:0];
                    2'b10: store <= ReadData2[3:0];
                    2'b11: store <= WriteData[3:0];
                endcase
            end
            4'b1101:begin
                case(display_mode)
                    2'b00: store <= nextPC[7:4];
                    2'b01: store <= ReadData1[7:4];
                    2'b10: store <= ReadData2[7:4];
                    2'b11: store <= WriteData[7:4];
                endcase
            end
            4'b1011:begin
                case(display_mode)
                    2'b00: store <= curPC[3:0];
                    2'b01: store <= instruction[24:21];
                    2'b10: store <= instruction[19:16];
                    2'b11: store <= ALUresult[3:0];
                endcase
            end
            4'b0111:begin
                case(display_mode)
```

```
                    2'b00: store<= curPC[7:4];
                    2'b01: store <= {3'b000,instruction[25]};
                    2'b10: store <= {3'b000,instruction[20]};
                    2'b11: store <= ALUresult[7:4];
                endcase
            end
        endcase
    end
endmodule
```

CPU.v

该部分是CPU的各个功能模块的顶层文件， 用于连接各个功能模块。

同时与数据通路图中Register File相关的选择器（如下图中红圈所示）我也已将其涵盖在此部分代码中，没有单独写在另一个文件中。




```
RegisterFile RegisterFile(.CLK(CLK),
                          .RegWre(RegWre),
                          .ReadReg1(rs),
                          .ReadReg2(rt),
                          .WriteReg(RegDst ? rd : rt),
                          .WriteData(DB),
                          .ReadData1(ReadData1),
                          .ReadData2(ReadData2)
```

```
`timescale 1ns / 1ps

module CPU(
        input CLK,
        input Reset,
        output [5:0] op,
```

```
        output [5:0] funct,
        output [31:0] ReadData1,
        output [31:0] ReadData2,
        output [31:0] curPC,
        output [31:0] nextPC,
        output [31:0] ALUresult,
        output [31:0] instruction
    );

    wire [4:0] rs;
    wire [4:0] rt;
    wire [4:0] rd;
    wire [15:0] immediate;
    wire [25:0] addr;
    wire [31:0] extended;
    wire [4:0] sa;
    wire [31:0] DataOut;
    wire [31:0] DB;
    wire PCWre, InsMemRW, RegDst, RegWre, ALUSrcA, ALUSrcB, mRD, mWR, D
BDataSrc, ExtSel;
    wire [2:0] ALUop;
    wire [1:0] PCSrc;
    wire ALU_zero, ALU_sign;
    // 对各个功能模块进行初始化赋值
    Adder Adder(CLK,Reset,PCSrc,extended,addr,curPC,nextPC);
    PC PC(CLK,Reset,PCWre,PCSrc,nextPC,curPC);
    InsMem InsMem(InsMemRW,curPC,instruction);
    InsSelect InsSelect(instruction,op,funct,rs,rt,rd,sa,immediate,addr
);
    ControlUnit ControlUnit(ALU_zero,ALU_sign,op,funct,InsMemRW,RegDst,
RegWre,ALUop,PCSrc,ALUSrcA,ALUSrcB,mRD,mWR,DBDataSrc,ExtSel,PCWre);
    RegisterFile RegisterFile(.CLK(CLK),
                              .RegWre(RegWre),
                              .ReadReg1(rs),
                              .ReadReg2(rt),
                              .WriteReg(RegDst ? rd : rt),
                              .WriteData(DB),
                              .ReadData1(ReadData1),
                              .ReadData2(ReadData2)
    );
    ALU ALU(ReadData1,ReadData2,sa,extended,ALUSrcA,ALUSrcB,ALUop,ALU_s
ign,ALU_zero,ALUresult);
    DataMem DataMem(ALUresult,ReadData2,CLK,mRD,mWR,DBDataSrc,DataOut,D
B);
```

```
    sign_zero_extend sign_zero_extend(immediate,ExtSel,extended);
endmodule
```

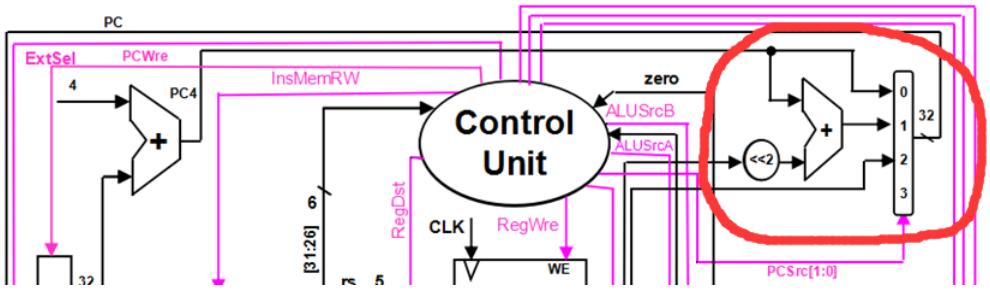
PC.v

PC模块用于存放当前指令的地址，该部分对应数据通路中的PC。当PC的值发生改变时，CPU会根据程序计数器PC中新得到的指令地址，从指令存储器中取出对应地址的指令。在单周期CPU的运行周期中，PC值的变化是最先发生的，并且最后会根据PCSrc这一控制信号的值来给curPC赋值，如PC+4或者跳转到某一地址。

```
`timescale 1ns / 1ps

module PC(
       input CLK,
       input Reset,
       input PCWre,
       input [1:0] PCSrc,
       input [31:0] nextPC,// PC 即将跳转的地址
       output reg[31:0] curPC// 当前地址
    );

    initial curPC <= 0;

    always@(posedge CLK or posedge Reset) begin
        if(Reset==0) curPC <= 0;//如果 Reset=0 则将当前地址清零
        else begin
            if(PCWre == 1) curPC <= nextPC;//若 Reset=1 且 PCWre=1.则将即将跳转的地址 nextPC 赋值给当前地址 curPC
            else curPC <= curPC;//若 Reset=1 且 PCWre=0，则当前地址保持不变
        end
    end
endmodule
```

Adder.v

该模块主要用于计算下一个 PC 值，并且数据通路图中右上角的选择器（如下图中红圈所示）我也已将其涵盖在此部分代码中（因为该选择器的其中一路输入是Adder的输出），没有单独写在另一个文件中。




```
            case(PCSrc) // 对应上图的选择器
                2'b00: nextPC = curPC + 4;
                2'b01: nextPC = curPC + 4 + immediate * 4;
                2'b10: nextPC = {newPC[31:28],addr,2'b00};
                2'b11: nextPC = nextPC;
```

```
`timescale 1ns / 1ps

module Adder(
        input CLK,
        input Reset,
        input [1:0] PCSrc,
        input [31:0] immediate,
        input [25:0] addr,
        input [31:0] curPC,
        output reg[31:0] nextPC
    );

    reg [31:0] newPC;
    initial newPC = 0 ;

    always@(negedge CLK or negedge Reset) begin
        if(Reset == 0) begin
            nextPC = 0;
        end
        else begin
            newPC = curPC + 4;
            case(PCSrc) // 对应上图的选择器
                2'b00: nextPC = curPC + 4;
                2'b01: nextPC = curPC + 4 + immediate * 4;
                2'b10: nextPC = {newPC[31:28],addr,2'b00};
                2'b11: nextPC = nextPC;
            endcase
        end
    end
endmodule
```

InsMem.v

该部分为指令寄存器，对应数据通路中的Ins Mem。该部分首先使用一个128位大小的8位寄存器数组来保存从文件中读入的所有指令，并通过输入的地址来找到相应的指令，最终将其输出到IDataOut。

该部分的功能是依次存储读入的每一条32位的指令，并根据程序计数器PC中的指令地址进行取指令操作。并且同时需要注意指令存储遵循高位指令存储在低位地址的规则（即小端存储）。

```
`timescale 1ns / 1ps

module InsMem(
        input InsMemRW,
        input [31:0] IAddr,
        output reg[31:0] IDataOut
    );

    reg [7:0] rom[0:127];
    initial $readmemb("C:\\Users\\93508\\Desktop\\instruction.txt", rom
);
    always@(IAddr or InsMemRW) begin
        if(InsMemRW) begin //进行小端存储
            IDataOut[7:0] = rom[IAddr + 3];
            IDataOut[15:8] = rom[IAddr + 2];
            IDataOut[23:16] = rom[IAddr + 1];
            IDataOut[31:24] = rom[IAddr];
        end
    end
endmodule
```

InsSelect.v

该部分是对指令类型进行分析，将读入指令的各字段进行拆分，从而确定指令类型（R型，I型和J型）和具体操作内容（add，sub等）。最后将指令的各部分传递给其他模块进行后续运算。

另外，根据指令格式和内容可知，部分指令的操作码(op)相同，需根据功能码(funct)来进行区分，所以我们实际上需要在图2中增加一条数据通路（已在图中用黑线标出）。

```
`timescale 1ns / 1ps

module InsSelect(
        input [31:0] instruction, // 指令
        output reg[5:0] op, // 操作码
```

```
        output reg[5:0] funct, // 功能码
        output reg[4:0] rs,
        output reg[4:0] rt,
        output reg[4:0] rd,
        output reg[4:0] sa,
        output reg[15:0] immediate,
        output reg[25:0] addr
    );

    initial begin // 初始化
        op = 5'b00000;
        rs = 5'b00000;
        rt = 5'b00000;
        rd = 5'b00000;
        funct = 5'b00000;
    end

    always@(instruction) begin// 对指令进行分割
        op = instruction[31:26];
        funct=instruction[5:0];
        rs = instruction[25:21];
        rt = instruction[20:16];
        rd = instruction[15:11];
        sa = instruction[10:6];
        immediate = instruction[15:0];
        addr = instruction[25:0];
    end
endmodule
```

ControlUnit.v

该部分对应数据通路中的Control Unit。CPU控制单元通过输入的zero，标志位sign，当前指令中对应的操作码op和功能码funct来确定当前CPU需要采取的运算形式。

```
`timescale 1ns / 1ps

module ControlUnit(
        input zero,
        input sign,
        input [5:0] op,
        input [5:0] funct,
        output reg InsMemRW,
        output reg RegDst,
```

```
        output reg RegWre,
        output reg [2:0] ALUop,
        output reg [1:0] PCSrc,
        output reg ALUSrcA,
        output reg ALUSrcB,
        output reg mRD,
        output reg mWR,
        output reg DBDataSrc,
        output reg ExtSel,
        output reg PCWre
    );

    always@(op or zero or sign or funct) begin
        case(op) //根据操作码来区分指令
            6'b000000: begin
                case(funct) //操作码相同时需要用功能码来区分
                    6'b100000:begin     //add
                        PCWre = 1;
                        {ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW,
mRD, mWR, RegDst, ExtSel} = 9'b000110010;
                        PCSrc[1:0] = 2'b00;
                        ALUop[2:0] = 3'b000;
                    end
                    6'b100010:begin     //sub
                        PCWre = 1;
                        {ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW,
mRD, mWR, RegDst, ExtSel} = 9'b000110011;
                        PCSrc[1:0] = 2'b00;
                        ALUop[2:0] = 3'b001;
                    end
                    6'b100100:begin     //and
                        PCWre = 1;
                        {ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW,
mRD, mWR, RegDst, ExtSel} = 9'b000110010;
                        PCSrc[1:0] = 2'b00;
                        ALUop[2:0] = 3'b100;
                    end
                    6'b100101:begin     //or
                        PCWre = 1;
                        {ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW,
mRD, mWR, RegDst, ExtSel} = 9'b000110010;
                        PCSrc[1:0] <= 2'b00;
                        ALUop[2:0] <= 3'b011;
                    end
```

```
                    6'b000000:begin     //sll
                        PCWre = 1;
                        {ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW,
 mRD, mWR, RegDst, ExtSel} = 9'b100110010;
                        PCSrc[1:0] = 2'b00;
                        ALUop[2:0] = 3'b010;
                    end
                endcase
            end
           // 以上 5 个指令(add,sub,and,or,sll)的操作码相同，需要用功能码来区
分。但剩下的 11 条指令的操作码均不相同，所以只需用操作码来进行区分即可
           // 以下只贴出 addiu 的判定内容，其他的指令如 andi,ori,slti 等指令判
定框架和 addiu 大体相同，不再贴出
            6'b001001: begin    //addiu
                PCWre = 1;
                {ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, mRD, mW
R, RegDst, ExtSel} = 9'b010110001;
                PCSrc[1:0] = 2'b00;
                ALUop[2:0] = 3'b000;
            end
```

RegisterFile.v

该部分为寄存器读写单元，储存寄存器组，并根据地址对寄存器组进行读写。对应数据通路中的Register File。该模块中，我们通过一个32位大小的32位寄存器数组来模拟寄存器，开始时全部置0。通过访问寄存器的地址，来获取寄存器里面的值，并进行操作。

寄存器组会根据操作码op与rs，rt字段相应的地址来读取数据,同时将rs，rt寄存器的地址和其中的数据输出，并在CLK的下降沿到来时将数据存放到rd或者rt字段的相应地址的寄存器内。

```
`timescale 1ns / 1ps

module RegisterFile(
        input [4:0] ReadReg1,
        input [4:0] ReadReg2,
        input [4:0] WriteReg,
        input [31:0] WriteData,
        input CLK,
        input RegWre,
        output [31:0] ReadData1,
        output [31:0] ReadData2
    );
```

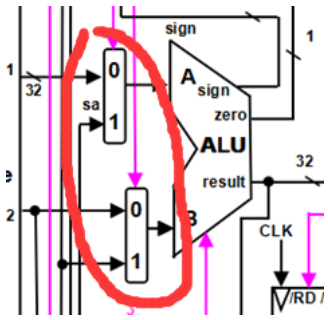
```
    reg [31:0] registers[0:31]; // 模拟寄存器
    integer i;
    initial begin // 对寄存器进行初始化
        for(i = 0; i < 32; i = i + 1) registers[i] <= 0;
    end

    assign ReadData1 = registers[ReadReg1];
    assign ReadData2 = registers[ReadReg2];

    always@(negedge CLK) begin // 每次在时钟下降沿将数据写入到寄存器中
        if(WriteReg!=0 && RegWre) registers[WriteReg] <= WriteData;
    end
endmodule
```

ALU.v

该部分为算术逻辑单元，用于逻辑指令计算，比较运算和跳转指令。对应数据通路图中的ALU。A、B为输入数，ALUop用于决定对A、B两数的操作内容，zero、sign主要用于beq、bne、bltz等指令的判断，result为运算结果。

该部分的功能是根据控制信号从输入数据中选取对应的操作数，根据操作码进行运算并输出结果与零标志位zero。

同时与这一模块相关的选择器（如下图中红圈所示）我也已将其涵盖在代码中，没有单独写在另一个文件中。




```
    always@(*) begin
        A = (ALUSrcA == 0) ? ReadData1 : sa; // 对应数据通路图中与输入 A 相连的 0-1 选择器
        B = (ALUSrcB == 0) ? ReadData2 : extend; // 对应数据通路图中与输入 B 相连的 0-1 选择器
```

```
`timescale 1ns / 1ps

module ALU(
    input [31:0] ReadData1,
```

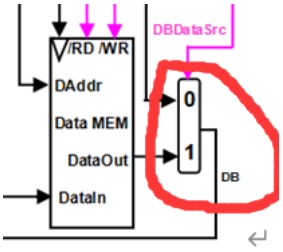
```
    input [31:0] ReadData2,
    input [4:0] sa,
    input [31:0] extend,
    input ALUSrcA,
    input ALUSrcB,
    input [2:0] ALUop,
    output reg sign,
    output reg zero,
    output reg [31:0] result
    );

    reg [31:0] A;
    reg [31:0] B;

    always@(*) begin
        A = (ALUSrcA == 0) ? ReadData1 : sa; // 对应数据通路图中与输入 A 相连的 0-1 选择器
        B = (ALUSrcB == 0) ? ReadData2 : extend; // 对应数据通路图中与输入 B 相连的 0-1 选择器

        case(ALUop)
            3'b000: result = A + B;
            3'b001: result = A - B;
            3'b010: result = B << A;
            3'b011: result = A | B;
            3'b100: result = A & B;
            3'b101: result = (A < B) ? 1 : 0;
            3'b110: result = ((A < B) && (A[31] == B[31]))||(((A[31] == 1) && (B[31] == 0)) ? 1 : 0);
            3'b111: result = A ^ B;
        endcase
        zero = (result == 0) ? 1 : 0;
        sign = result[31];
    end
endmodule
```

DataMem.v

该部分控制内存存储，用于内存的存储和读写。对应数据通路图中的Data Mem。这里我们用256位大小的8位寄存器数组来模拟内存，采用小端存储方式。

同时与这一模块相关的选择器（如下图中红圈所示）我也已将其涵盖在代码中，没有单

独写在另一个文件中。




```
        DB = DBDataSrc ? DataOut : DAddr; // 对应数据通路图中 Data Mem 右边的 0-1 选择器↵
```

```
`timescale 1ns / 1ps

module DataMem(
        input [31:0] DAddr,
        input [31:0] DataIn,
        input CLK,
        input mRD,
        input mWR,
        input DBDataSrc,
        output reg[31:0] DataOut,
        output reg[31:0] DB
    );
    reg [7:0] dataMemory[0:255];
    integer i;
    initial begin
        DB <= 16'b0;
        DataOut <= 16'b0;
        for(i = 0;i < 256; i = i + 1) dataMemory[i] <= 0;
    end

    always@(mRD or DAddr or DBDataSrc) begin
        DataOut[31:24] = mRD ? dataMemory[DAddr] : 8'bz;
       // 如果 mRD 为 0 则输出高阻抗
        DataOut[23:16] = mRD ? dataMemory[DAddr + 1] : 8'bz;
        DataOut[15:8] = mRD ? dataMemory[DAddr + 2] : 8'bz;
        DataOut[7:0] = mRD ? dataMemory[DAddr + 3] : 8'bz;
        DB = DBDataSrc ? DataOut : DAddr; // 对应数据通路图中 Data Mem 右边的 0-1 选择器
    end

    always@(negedge CLK) begin
        if(mWR == 1) begin
            dataMemory[DAddr] = DataIn[31:24];
```

```
            dataMemory[DAddr + 1] = DataIn[23:16];
            dataMemory[DAddr + 2] = DataIn[15:8];
            dataMemory[DAddr + 3] = DataIn[7:0];
        end
    end
endmodule
```

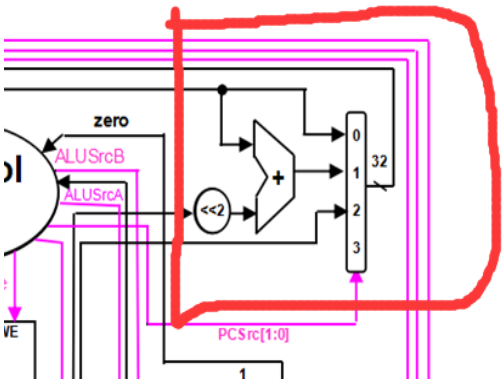
sign_zero_extend.v

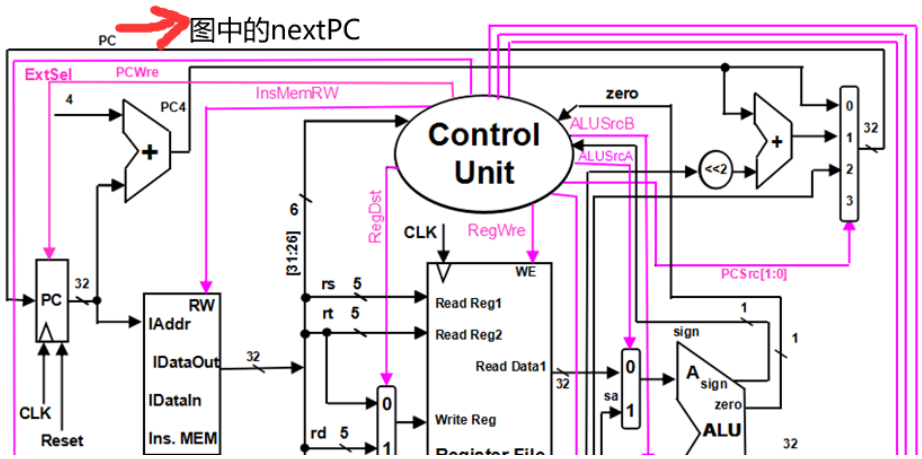
该模块用于立即数的符号位扩展，当ExtSel信号为0时做0扩展，为1时做1扩展。

```
`timescale 1ns / 1ps

module sign_zero_extend(
        input [15:0] immediate,
        input ExtSel,
        output [31:0] extendImmediate
    );

    assign extendImmediate = {ExtSel && immediate[15] ? 16'hffff : 16'h
0000, immediate};

endmodule
```

4、结果分析

补充说明：正如前面Adder部分所解释的那样，波形图中的nextPC是数据通路图中右上角的选择器的输出（已涵盖在Adder部分代码中），也即PC部分的输入PC。

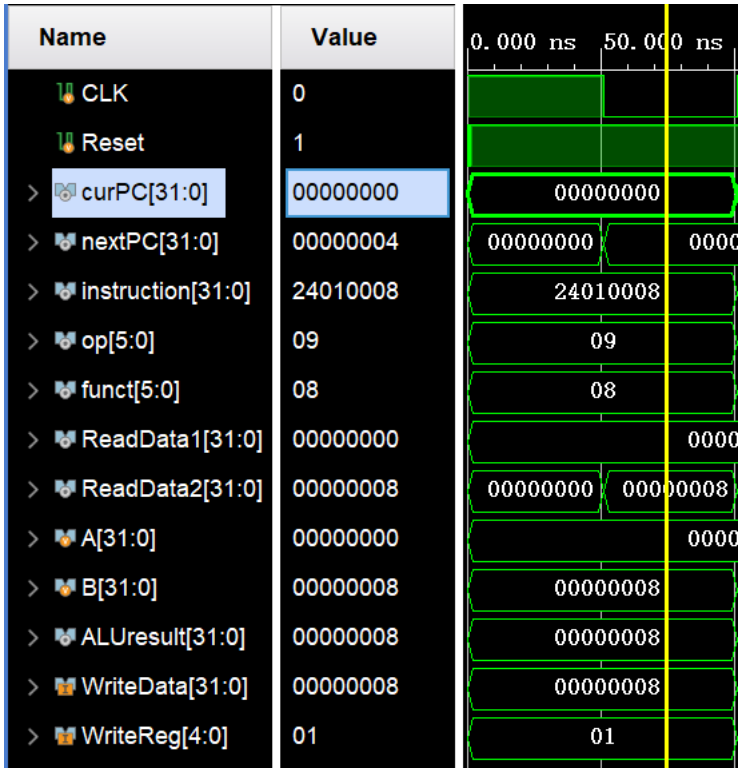




所以nextPC的波形会比curPC提前50ns出现（因为时钟电平每50ns变化1次），但这并不影响其他结果的输出（如ALUresult，WriteReg，WriteData等）。虽然这一行的波形图没有和curPC对齐，显得不太美观，但我觉得在代码模块化方面没有太大问题，所以就没有对代码进行修改，如给老师带来审阅不便还请谅解，谢谢老师！

接下来我会根据测试程序段来逐条分析波形图，以此来验证结果的正确性。

① addiu $1,$0,8



由测试程序表可知，当前地址PC为0000 0000，下条地址nextPC为0000 0004

操作码op=001001=09，功能码funct=001000=08

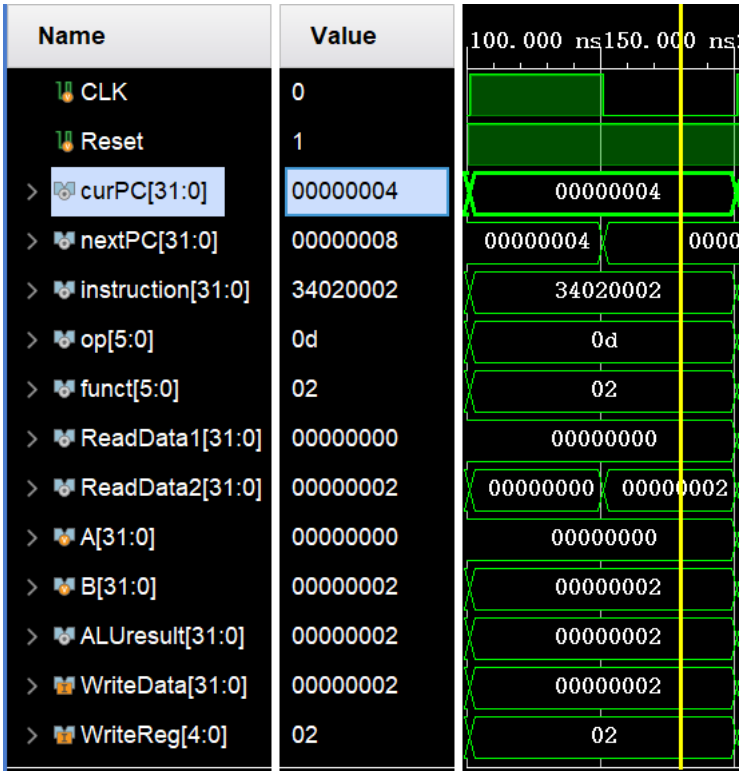
immediate=0000 0008

ReadData1=rs=$0=0000 0000,ReadData2=rt=$1,$1在执行指令前为0000 0000，执

行指令后为0000 0008（rt=rs+immediate）

所以显然A=rs=0000 0000，B=rt=0000 0008，又因为ALUop=000，所以ALUresult=A+B=0000 0008(rt=rs+immediate)，DB=0000 0008，并将结果写入寄存器01（即$1=0000 0008）

② ori $2,$0,2



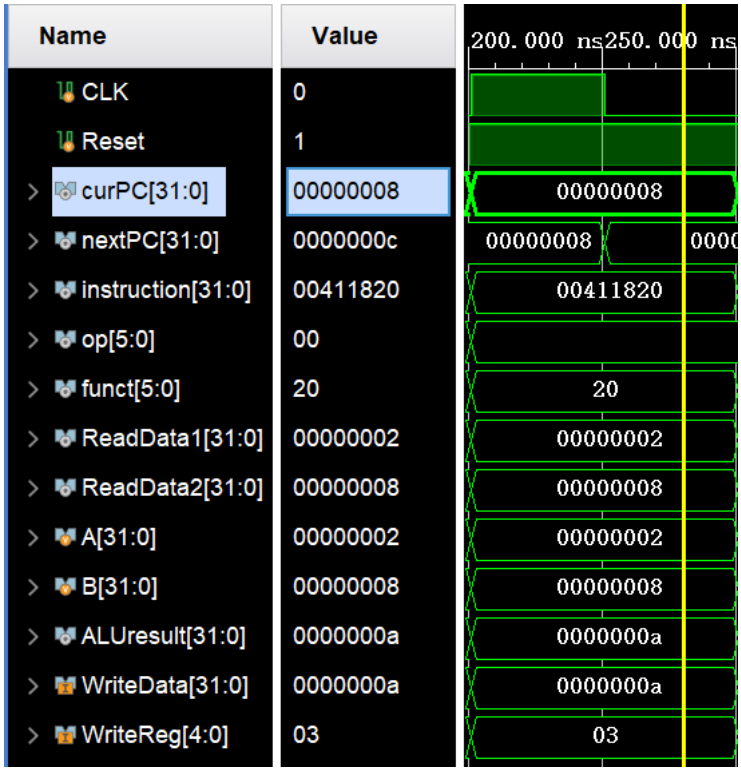
由测试程序表可知，当前地址PC为0000 0004，下条地址nextPC为0000 0008.

操作码op=001101=0d，功能码funct=000010=02

ReadData1=rs=$0=0000 0000,ReadData2=rt=$2,$2在执行指令前为0000 0000，执行指令后为0000 0002(rt=rs | immediate)

所以显然A=rs=0000 0000，B=rt=0000 0002，又因为ALUop=011，所以ALUresult = A∨B=0000 0002(rt=rs | immediate)，DB=0000 0002，并将结果写入寄存器02（即$2=0000 0002）

③ add $3,$2,$1



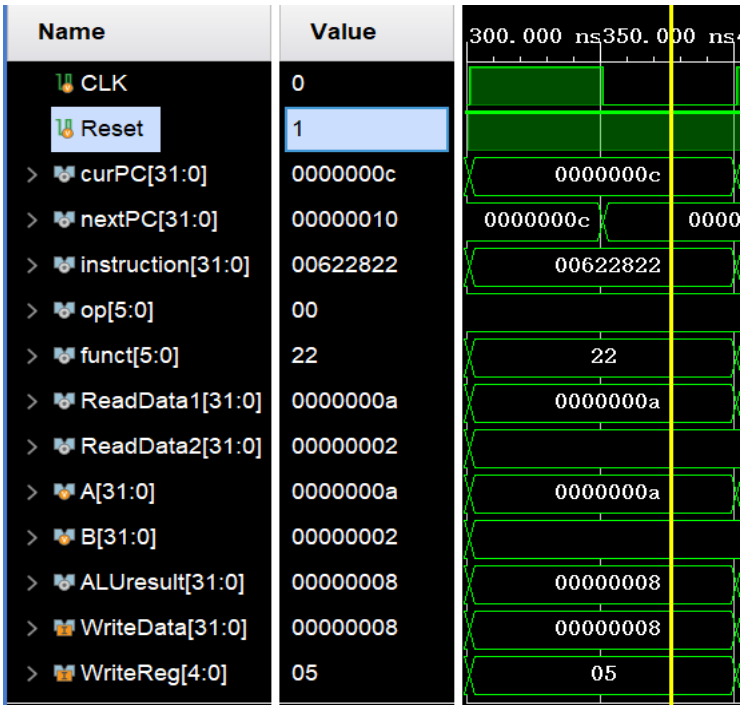
由测试程序表可知，当前地址PC为0000 0008，下条地址nextPC为0000 000c.

操作码op=000000=00，功能码funct=100000=20

ReadData1=rs=$2=0000 0002,ReadData2=rt=$1=0000 0008

所以显然A=rs=0000 0002，B=rt=0000 0008，又因为ALUop=000，所以ALUresult=A+B=0000 000a(rd=rs+rt)，DB=0000 000a，并将结果写入寄存器03（即$3=rs+rt=0000 000a）

④ sub $5,$3,$2



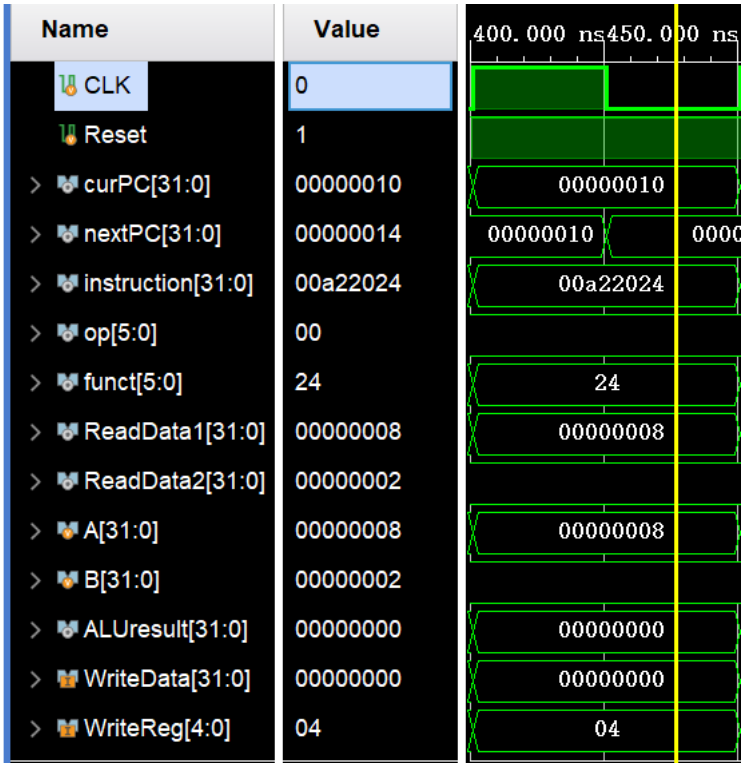
由测试程序表可知，当前地址PC为0000 000c，下条地址nextPC为0000 0010.

操作码op=000000=00，功能码funct=100010=22

ReadData1=rs=$3=0000 000a,ReadData2=rt=$2=0000 0002

所以显然A=rs=0000 000a，B=rt=0000 0002，又因为ALUop=001，所以ALUresult=A-B=0000 0008(rd=rs-rt)，DB=0000 000a，并将结果写入寄存器05（即$5=rs-rt=0000 0008）

⑤ and $4,$5,$2



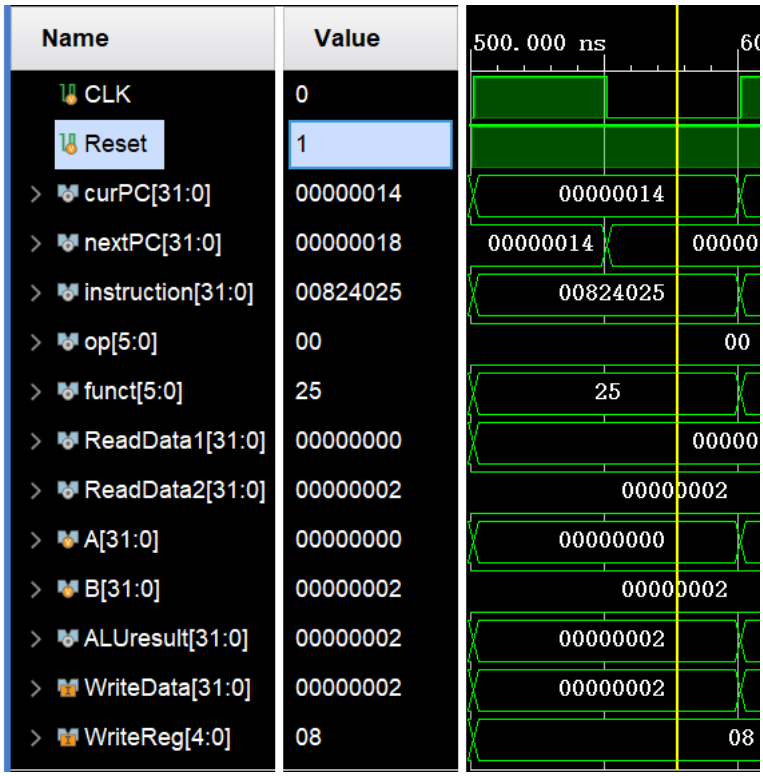
由测试程序表可知，当前地址PC为0000 0010，下条地址nextPC为0000 0014.

操作码op=000000=00，功能码funct=100100=24

ReadData1=rs=$5=0000 0008,ReadData2=rt=$2=0000 0002

所以显然A=rs=0000 0008，B=rt=0000 0002，又因为ALUop=100，所以ALUresult = A ∧ B=0000 0000(rd=rs&rt)，DB=0000 000a，并将结果写入寄存器04（即$4=rs&rt=0000 0000）

⑥ or $8,$4,$2



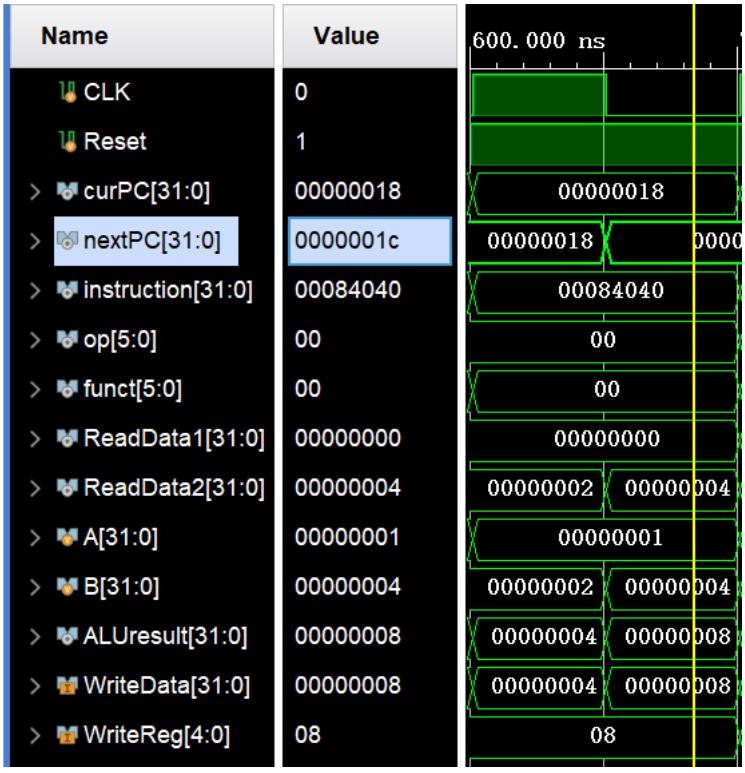
由测试程序表可知，当前地址PC为0000 0014，下条地址nextPC为0000 0018.

操作码op=000000=00，功能码funct=100101=25

ReadData1=rs=$4=0000 0000,ReadData2=rt=$2=0000 0002

所以显然A=rs=0000 0000，B=rt=0000 0002，又因为ALUop=011，所以ALUresult = A ∨ B = 0000 0002(rd=rs | rt)，DB=0000 0002,并将结果写入寄存器08（即$8=rd=rs | rt=0000 0002）

⑦ sll $8,$8,1



由测试程序表可知，当前地址PC为0000 0018，下条地址nextPC为0000 001c.

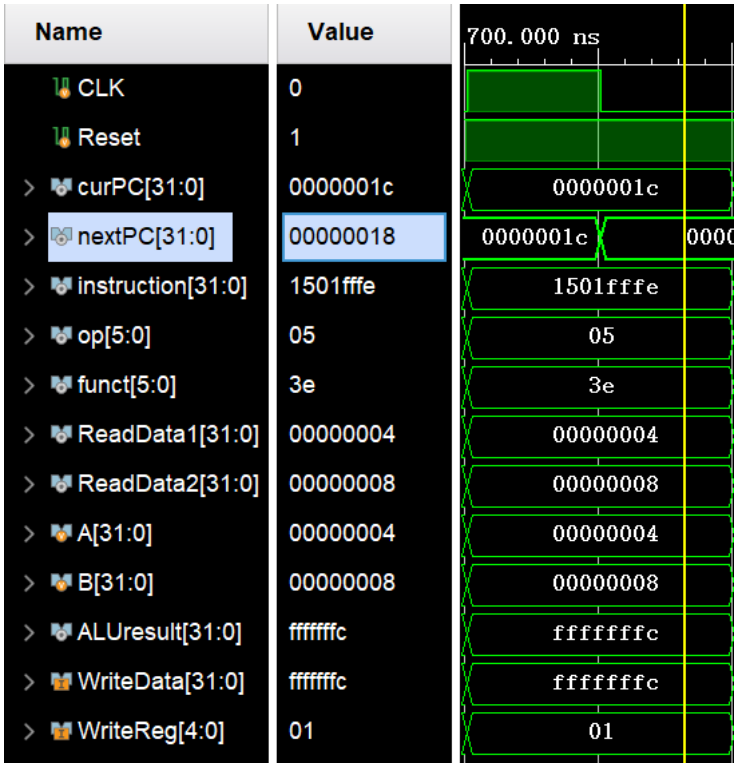
操作码op=000000=00，功能码funct=000000=00

immediate=sa=0000 0001

ReadData1=rs=0000 0000,ReadData2=rd=$8, $8在执行指令前为0000 0002，执行指令后为0000 0004(rt=rs<<sa)

所以显然A=sa=0000 0001，B=rt(移位前rt=0000 0002，移位后rt=0000 0004)，又因为ALUop=010，所以ALUresult=B<<A=rd(移位前rd=0000 0004，移位后rd=0000 0008)，DB=ALUresult,并将结果写入寄存器08

⑧ bne $8,$1,-2 (≠,转18)



由测试程序表可知，当前地址PC为0000 001c，下条地址nextPC为0000 0018.

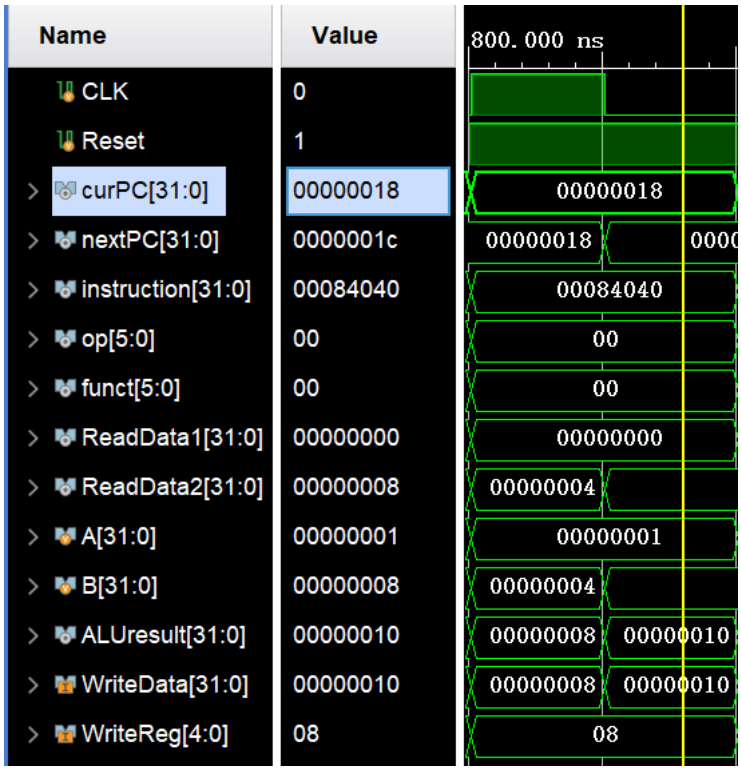
操作码op=000101=05

ReadData1=rs=$8=0000 0004,ReadData2=rt=$1=0000 0008

offset=-2

所以显然A=rs=0000 0004，B=rt=0000 0008，又因为ALUop=001，所以ALUresult=A-B=ffff fffc，DB=ffff fffc,并将结果写入寄存器01,又因为A-B≠0(rs≠rt)，所以跳转地址为nextPC+4+ sign_extend(offset)<<2=0000 0018而不是0000 0020

⑨ sll $8,$8,1



由测试程序表可知，当前地址PC为0000 0018，下条地址nextPC为0000 001c

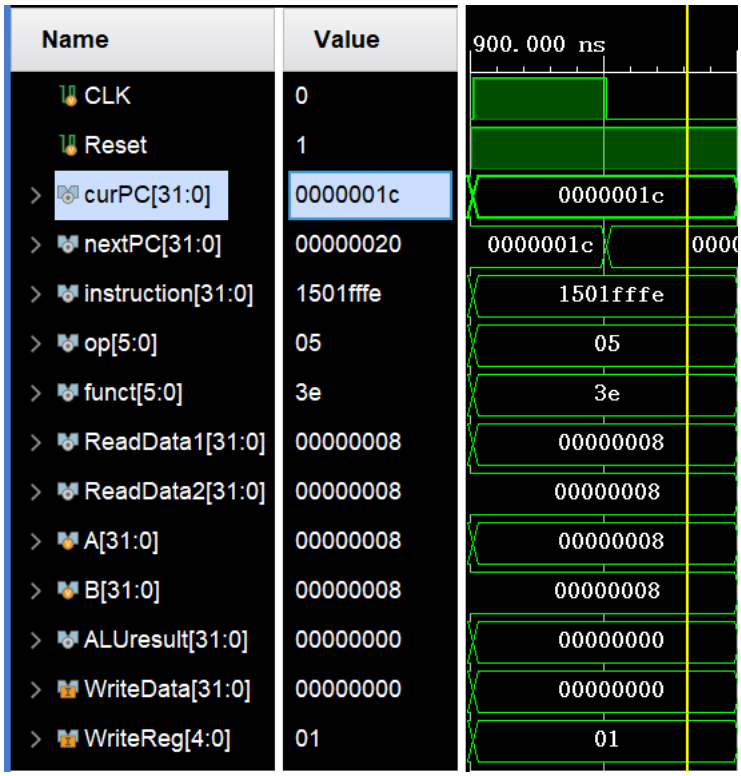
操作码op=000000=00，功能码funct=000000=00

immediate=01

ReadData1=rs=0000 0000,ReadData2=rd=$8, $8在执行指令前为0000 0004，执行指令后为0000 0008(rt=rs<<sa)

所以显然A=sa=0000 0001，B=rt(移位前rt=0000 0004，移位后rt=0000 0008)，又因为ALUop=010，所以ALUresult=B<<A=rd(移位前rd=0000 0008，移位后rd=0000 0010)，DB=ALUresult,并将结果写入寄存器08

⑩ bne $8,$1,-2 (≠,转18)



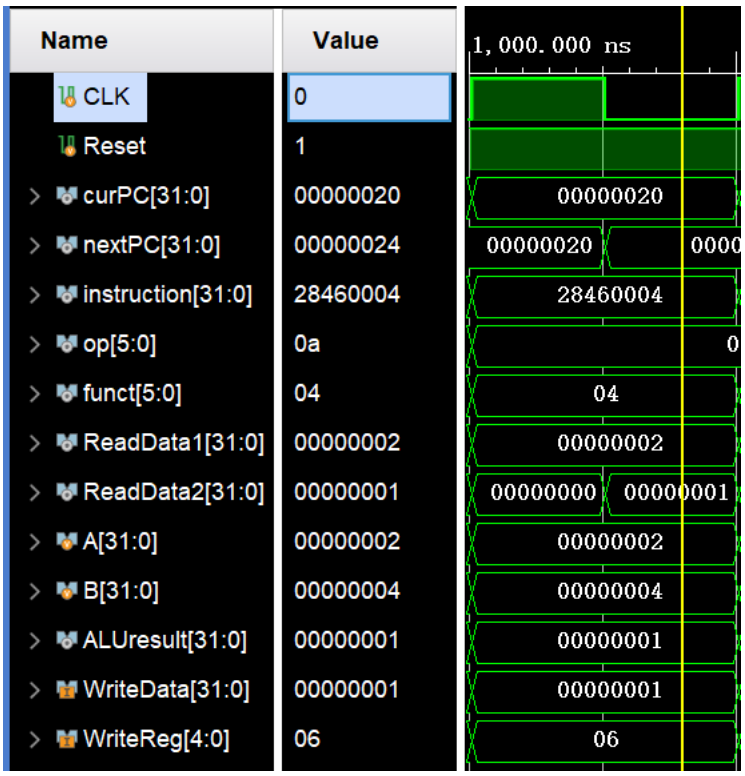
由测试程序表可知，当前地址PC为0000 001c，下条地址nextPC为0000 0020

操作码op=000101=05

ReadData1=rs=$8=0000 0008,ReadData2=rt=$1=0000 0008

所以显然A=rs=0000 0008，B=rt=0000 0008，又因为ALUop=001，ALUresult=A-B=0000 0000，DB=0000 0000,并将结果写入寄存器01,又因为A-B=0(rs=rt)，所以跳转地址为（PC+4），即0000 0020

⑪ slti $6,$2,4



由测试程序表可知，当前地址PC为0000 0020，下条地址nextPC为0000 0024.

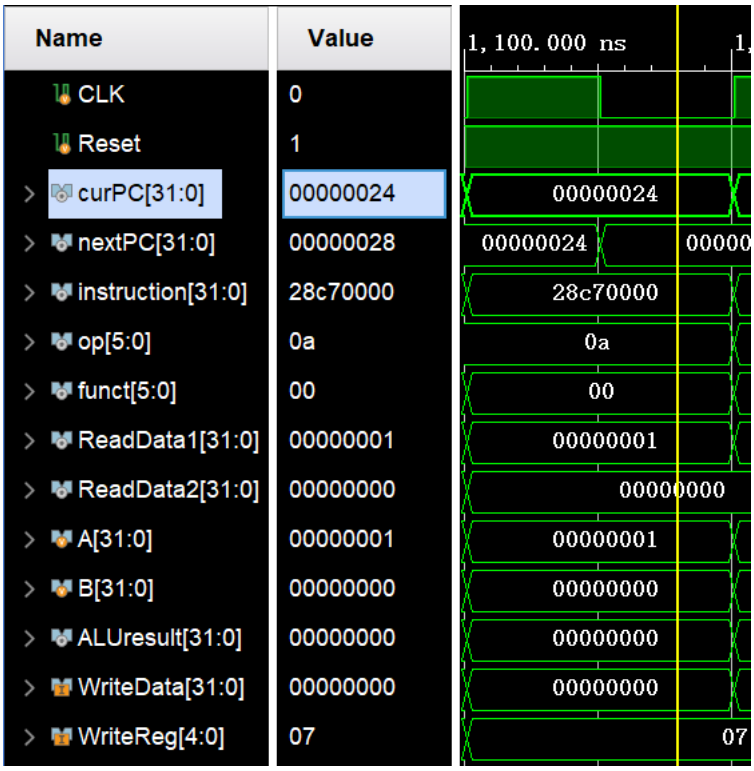
操作码op=001010=0a

immediate=0000 0100=04

ReadData1=rs=$2=0000 0002,ReadData2=rt=$6, $6在执行指令前为0000 0000，执行指令后为0000 0001

显然A=rs=0000 0002，B=immediate=0000 0004，又因为ALUop=101，所以ALUresult= (A<B) ?1: 0=0000 0001(因为rs<immediate，所以rt=1)，DB=0000 0001，并将结果写入寄存器06

⑫ slti $7,$6,0



由测试程序表可知，当前地址PC为0000 0024，下条地址nextPC为0000 0028.

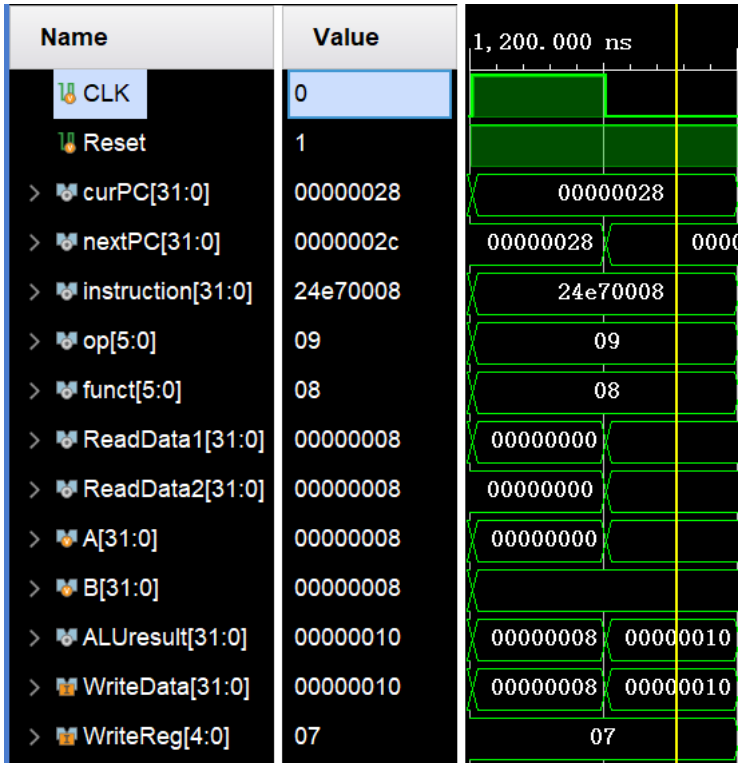
操作码op=001010=0a

immediate=0000 0000=00

ReadData1=rs=$6=0000 0001,ReadData2=rt=$7=0000 0000

显然A=rs=0000 0001，B=immediate=0000 0000，又因为ALUop=101，所以ALUresult= (A<B) ?1: 0=0000 0000(因为rs>immediate，所以rt=0)，DB=0000 0000，并将结果写入寄存器07

⑬ addiu $7,$7,8



由测试程序表可知，当前地址PC为0000 0028，下条地址nextPC为0000 002c

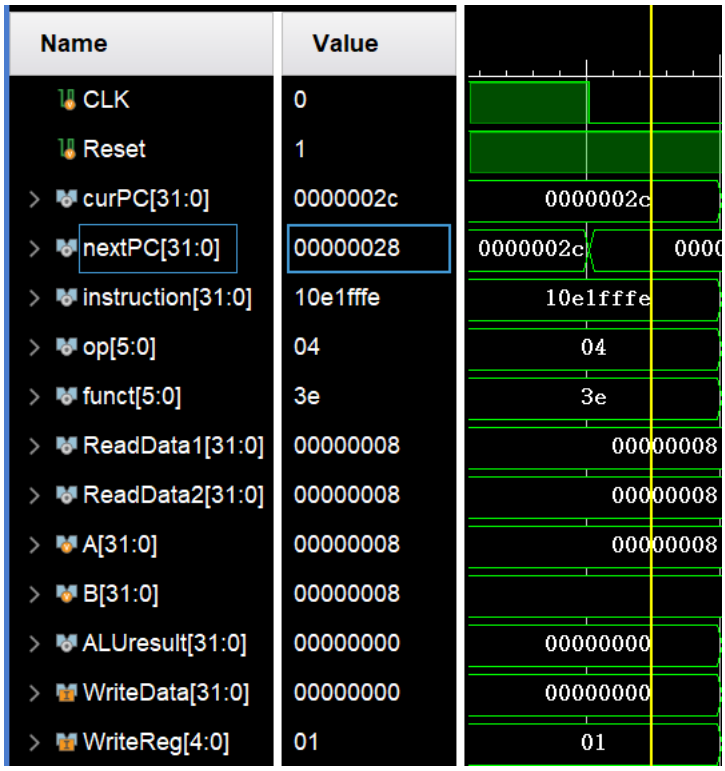
操作码op=001001=09，功能码funct=001000=08

immediate=0000 0008

ReadData1=rs=$7,$7 在执行指令前为 0000 0000，执行指令后为 0000 0008($7=$7+8)，ReadData2=rt=$7

所以显然 A=rs(执行指令前为 0000 0000，执行指令后为 0000 0008)，B=immediate=0000 0008，又因为ALUop=000，所以ALUresult=A+B，在执行指令前为0000 0008，执行指令后为0000 0010，DB=ALUresult，并将结果写入寄存器07

⑭ beq $7,$1,-2 (=,转28)



由测试程序表可知，当前地址PC为0000 002c，下条地址nextPC为0000 0028

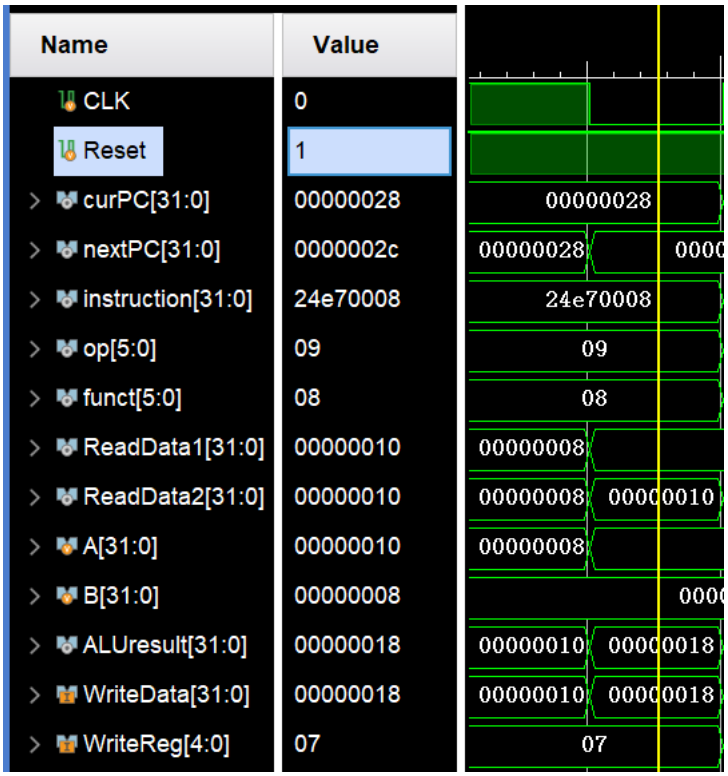
操作码op=00100=04

offset = -2

ReadData1=rs=$7=0000 0008,ReadData2=rt=$1=0000 0008

所以显然A=rs=0000 0008,B=rt=0000 0008，又因为ALUop=001，所以ALUresult=A-B=0000 0000，DB=0000 0000，并将结果写入寄存器01,且因为A-B=0(rs=rt)，所以跳转地址为curPC+4+sign_extend(offset)<<2=0000 0028

⑮ addiu $7,$7,8



由测试程序表可知，当前地址PC为0000 0028，下条地址nextPC为0000 002c

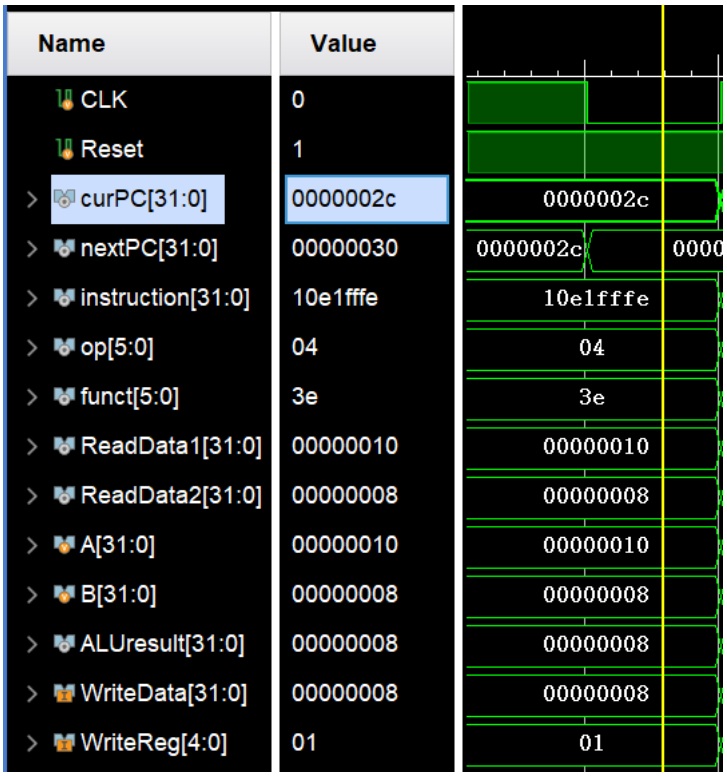
操作码op=001001=09，功能码funct=001000=08

immediate=0000 0008

ReadData1=rs=$7,$7 在执行指令前为0000 0008，执行指令后为0000 0010($7=$7+8)，ReadData2=rt=$7

所以显然A=rs(执行指令前为0000 0008，执行指令后为0000 0010)，B=immediate=0000 0008，又因为ALUop=000，所以ALUresult=A+B在执行指令前为0000 0010，执行指令后为0000 0018，DB=ALUresult，并将结果写入寄存器07

⑯ beq $7,$1,-2 (=,转28)



由测试程序表可知，当前地址PC为0000 002c，下条地址nextPC为0000 0030

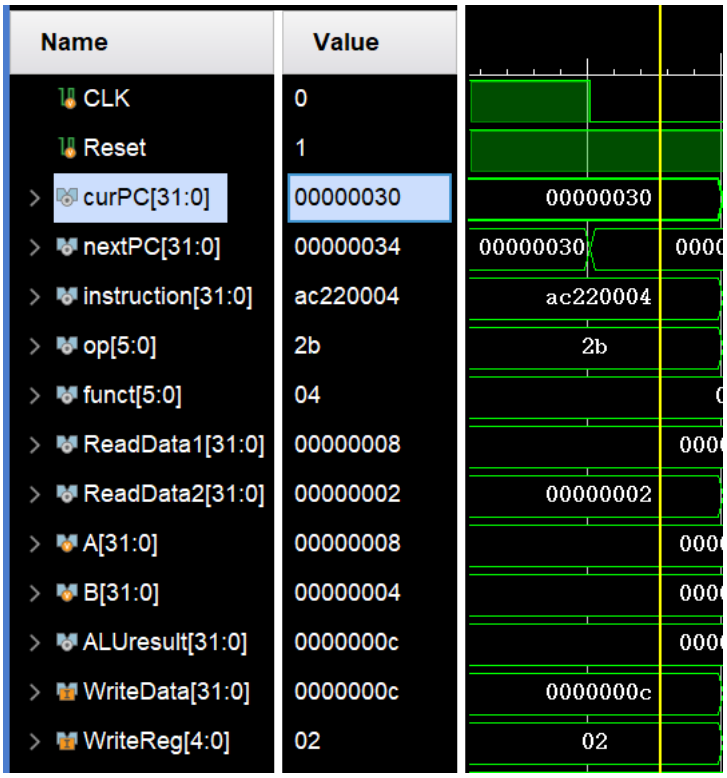
操作码op=00100=04

offset = -2

ReadData1=rs=$7=0000 0010,ReadData2=rt=$1=0000 0008

所以显然A=rs=0000 0010,B=rt=0000 0008，又因为ALUop=001，所以ALUresult=A-B=0000 0008，DB=0000 0008，并将结果写入寄存器01,且因为A-B≠0(rs≠rt)，所以跳转地址为curPC+4=0000 0030

⑰ sw $2,4($1)



由测试程序表可知，当前地址PC为0000 0030，下条地址nextPC为0000 0034

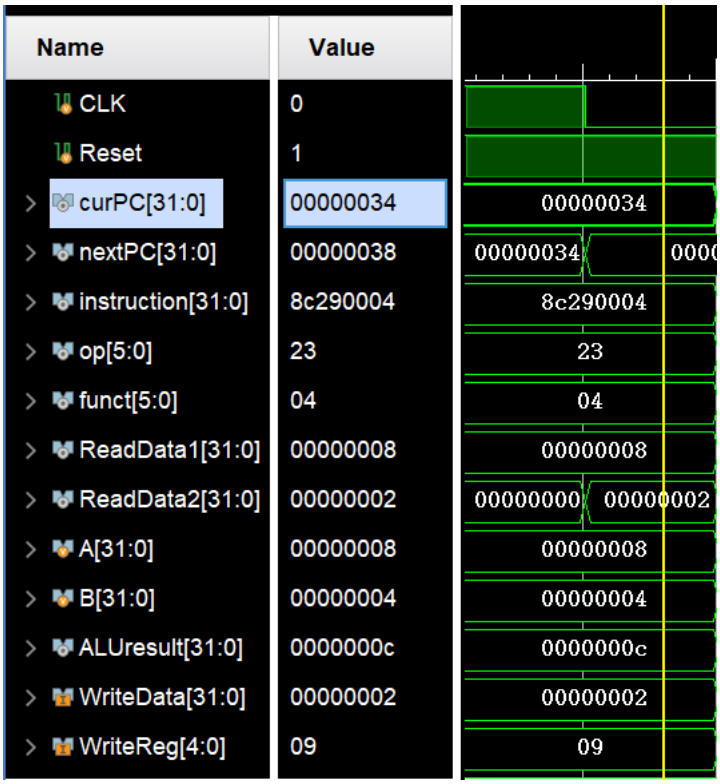
操作码op=101011=2b

immediate=0000 0004

ReadData1=rs=$1=0000 0008,ReadData2=rt=$2=0000 0002

所以显然A=rs，B=immediate=0000 0004，又因为ALUop=000，所以ALUresult=A+B=0000 000c，DB=0000 000c，并将结果写入寄存器02

⑱ lw $9,4($1)



由测试程序表可知，当前地址PC为0000 0034，下条地址nextPC为0000 0038

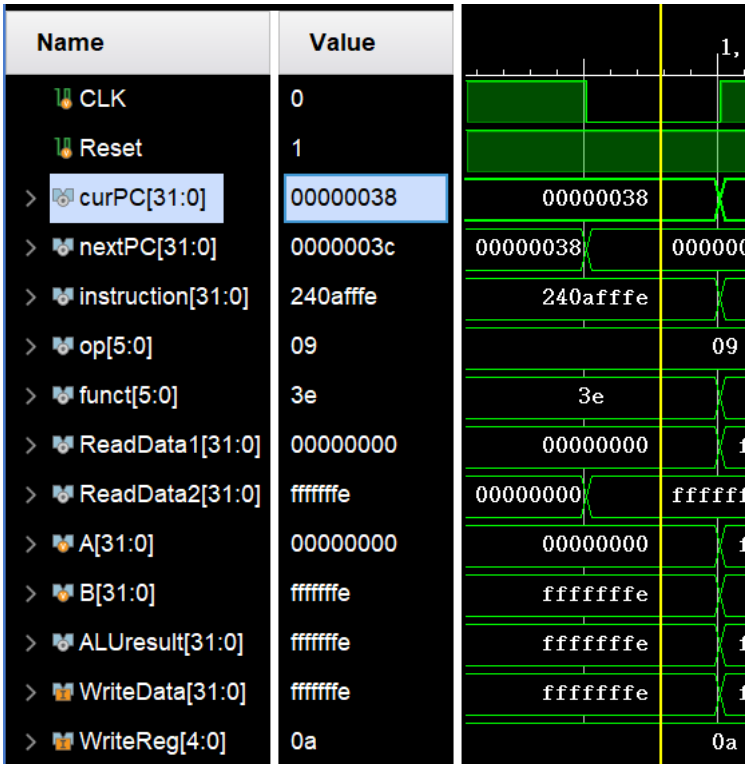
操作码op=100011=23

immediate=0000 0004

ReadData1=rs=$1=0000 0008,ReadData2=rt=$9，$9在执行指令为0000 0000,执行指令后为0000 0002

所以显然A=rs=0000 0008，B=immediate=0000 0004，又因为ALUop=000，所以ALUresult=A+B=0000 000c，DB=0000 0002，并将结果写入寄存器09

⑲ addiu $10,$0,-2



由测试程序表可知，当前地址PC为0000 0038，下条地址nextPC为0000 003c

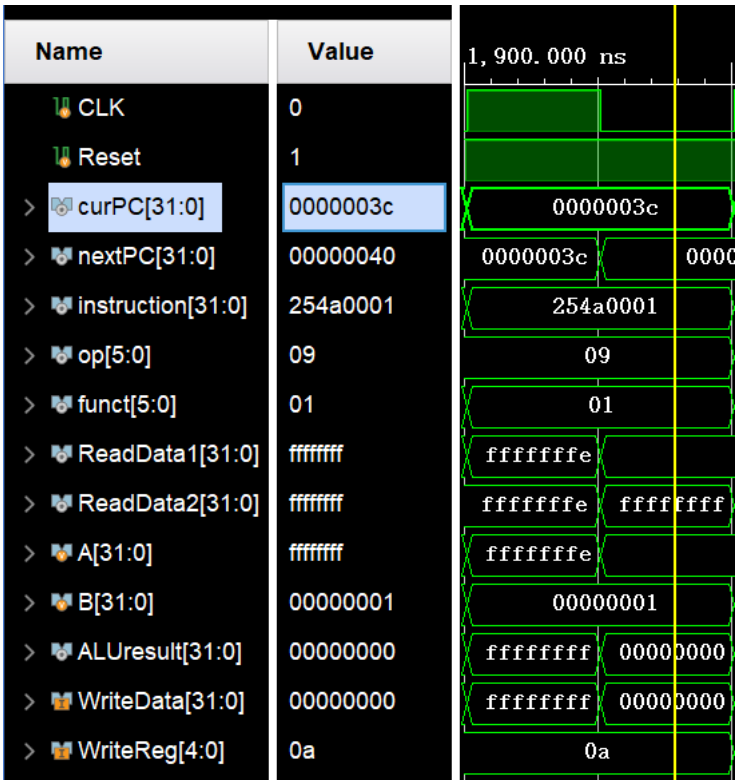
操作码op=001001=09

immediate = -2

ReadData1=rs=$0=0000 0000,ReadData2=rt=$10,$10在执行指令前为0000 0000，执行指令后为ffff fffe

所以显然A=rs=0000 0000，B=[-2]补=ffff fffe，又因为ALUop=000，所以ALUresult=A+B=ffff fffe，DB=ALUresult，并将结果写入寄存器0a

⑳ addiu $10,$10,1



由测试程序表可知，当前地址PC为0000 003c，下条地址nextPC为0000 0040

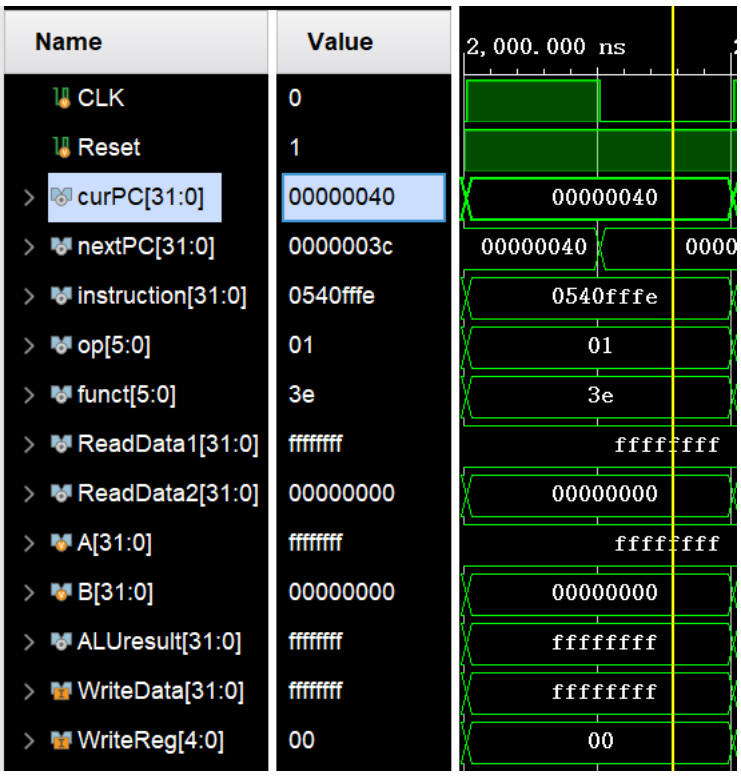
操作码op=001001=09

immediate = 1

ReadData1=rs=$10，ReadData2=rt=$10，$10在执行指令前为ffff fffe，执行指令后为ffff ffff

所以显然A=rs=$10，$10在执行指令前为ffff fffe，执行指令后为ffff ffff ,B=immediate=0000 0001，又因为ALUop=000，所以ALUresult=A+B,在执行指令前为ffff ffff,执行指令后为0000 0000，DB=ALUresult，并将结果写入寄存器0a

㉑ bltz $10,-2(<0,转3C)



由测试程序表可知，当前地址PC为0000 0040，下条地址nextPC为0000 003c

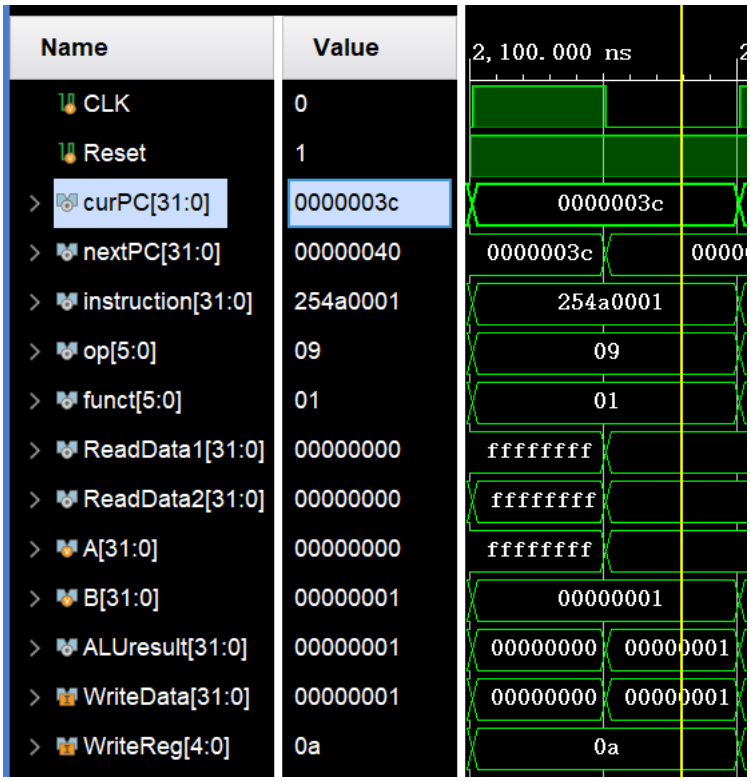
操作码op=000001=01

offset = -2

ReadData1=rs=$10=ffff ffff,ReadData2=0000 0000

所以显然A=rs=ffff ffff,B=rt=0000 0000，又因为ALUop=000，所以ALUresult=A+B=ffff ffff，DB=ffff ffff，并将结果写入寄存器00,且因为A+B<0(rs<0)，所以跳转地址为curPC+4+sign_extend (offset) <<2=0000 003c

㉒ addiu $10,$10,1



由测试程序表可知，当前地址PC为0000 003c，下条地址nextPC为0000 0040

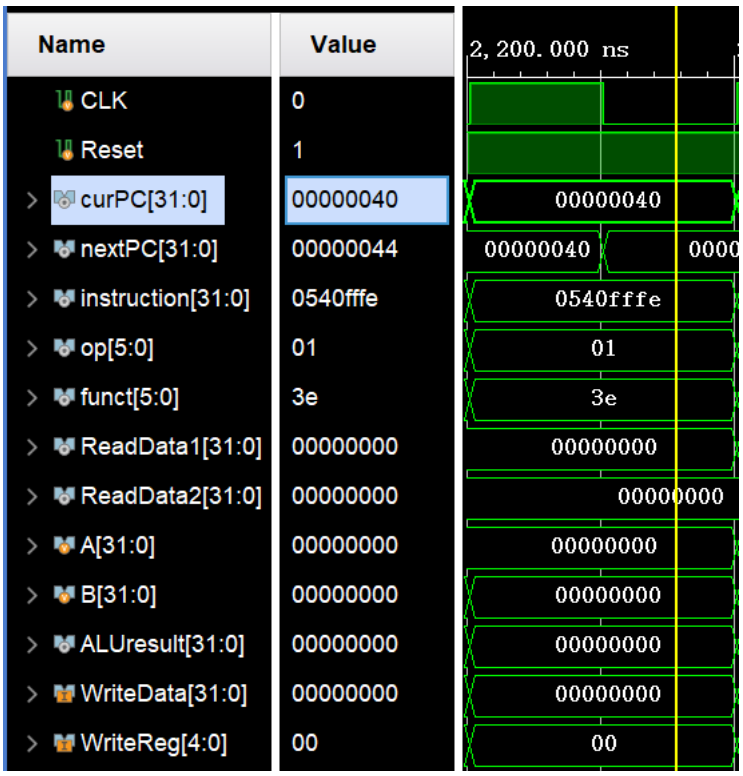
操作码op=001001=09

immediate = 1

ReadData1=rs=$10，ReadData2=rt=$10，$10在执行指令前为ffff ffff，执行指令后为0000 0000

所以显然A=rs=$10，$10在执行指令前为ffff ffff，执行指令后为0000 0000,B=immediate=0000 0001，又因为ALUop=000，所以ALUresult=A+B,在执行指令前为0000 0000,执行指令后为0000 0001，DB=ALUresult，并将结果写入寄存器0a

㉓ bltz $10,-2(<0,转3C)



由测试程序表可知，当前地址PC为0000 0040，下条地址nextPC为0000 0044

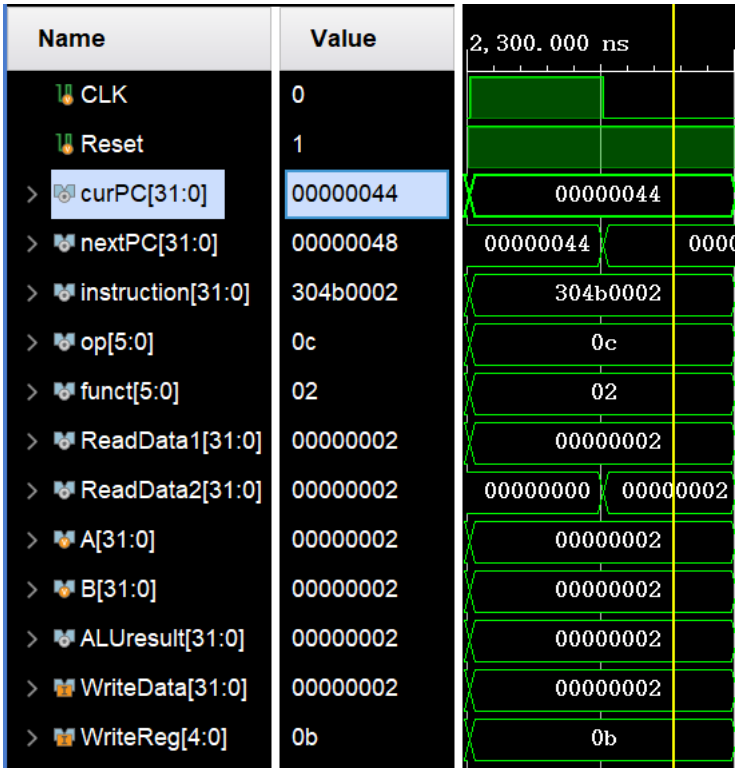
操作码op=000001=01

offset = -2

ReadData1=rs=$10=0000 0000,ReadData2=0000 0000

所以显然A=rs=0000 0000,B=rt=0000 0000，又因为ALUop=000，所以ALUresult=A+B=0000 0000，DB=0000 0000，并将结果写入寄存器00,且因为A+B=0(rs>=0)，所以跳转地址为curPC+4=0000 0044

㉔ andi $11,$2,2



由测试程序表可知，当前地址PC为0000 0044，下条地址nextPC为0000 0048.

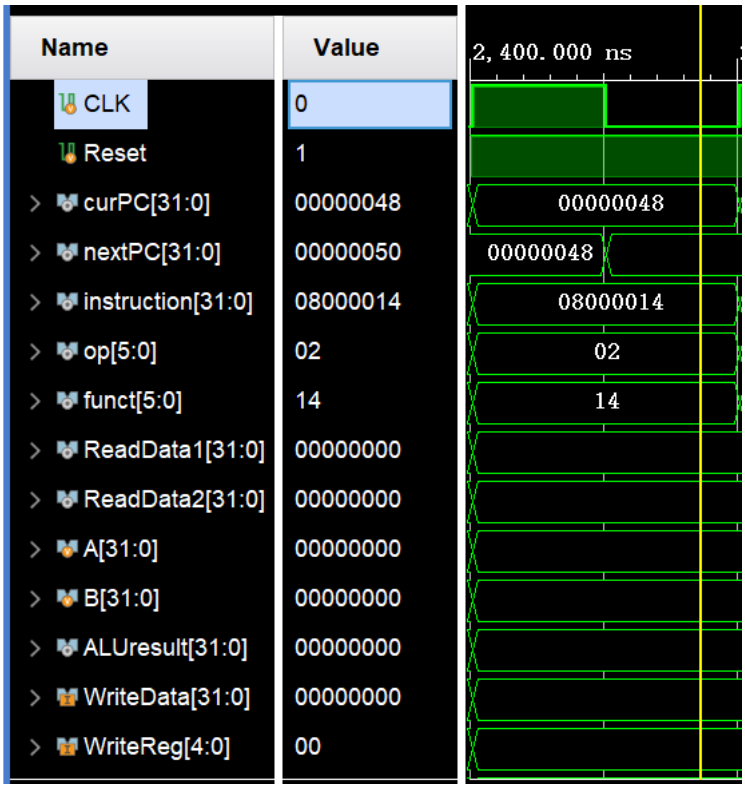
操作码op=001100=0d

immediate=0000 0002

ReadData1=rs=$2=0000 0002,ReadData2=rt=$11,$11在执行指令前为0000 0000，执行指令后为0000 0002(rt=rs&immediate)

所以显然A=rs=0000 0002，B=rt=0000 0002，又因为ALUop=100，所以ALUresult=0000 0002(rt=rs&immediate)，DB=0000 0002，并将结果写入寄存器0b

㉕ j 0x00000050



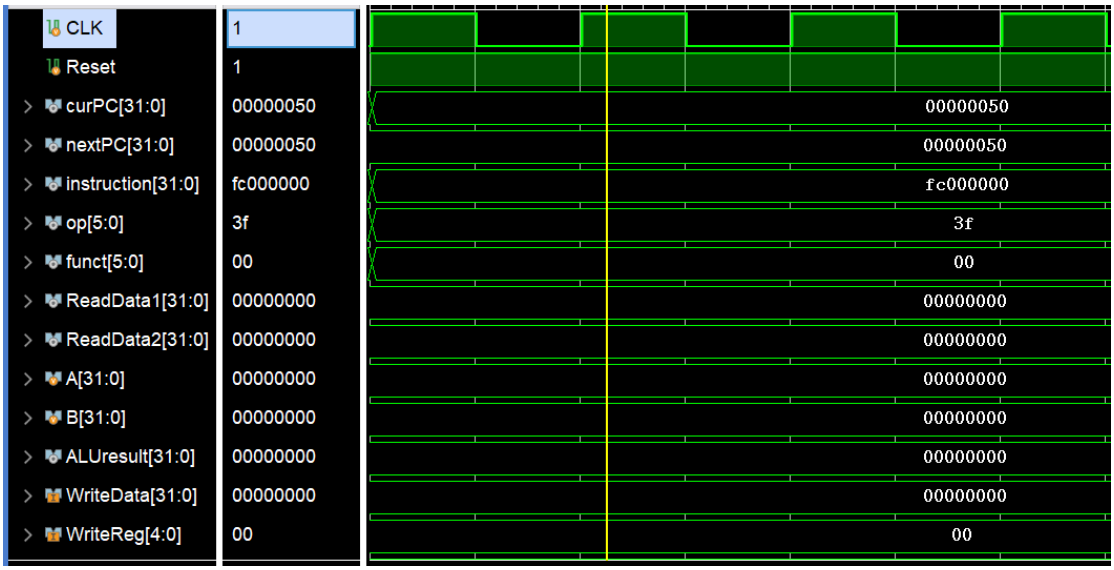
由测试程序表可知，当前地址PC为0000 0048，下条地址nextPC为0000 0050

addr=0x00000050

由j指令定义可知，curPC={curPC[31:28] , addr , 2’b0}，无条件跳转。

所以跳过了地址为0x0000004C的指令：or $8,$4,$2

㉖ halt

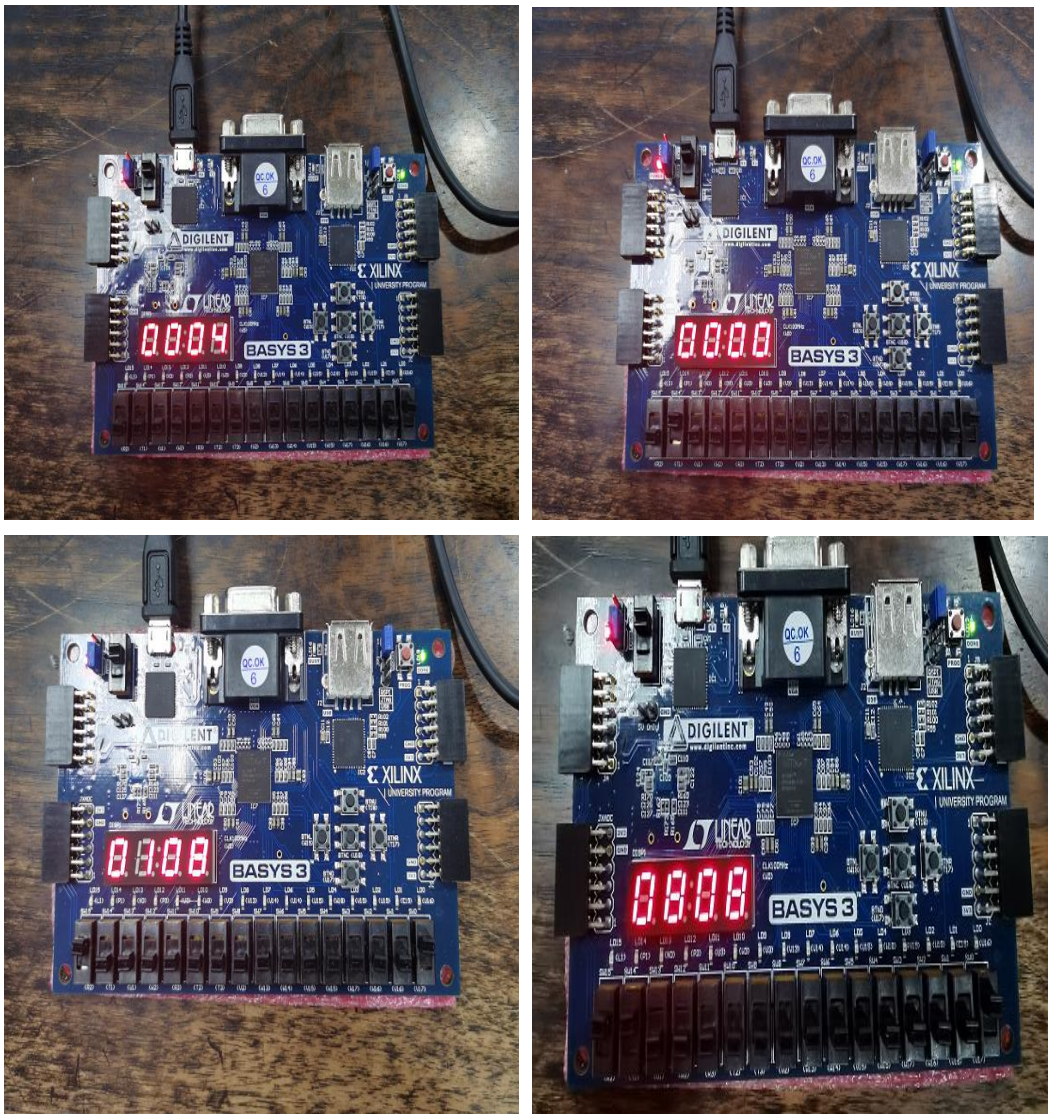


从此开始，curPC保持不变，程序运行结束。

5、实现

根据实验要求，现展示测试程序代码段前5条指令的Basys3显示图

① addiu $1,$0,8



图片从左到右，从上到下，分别对应为：

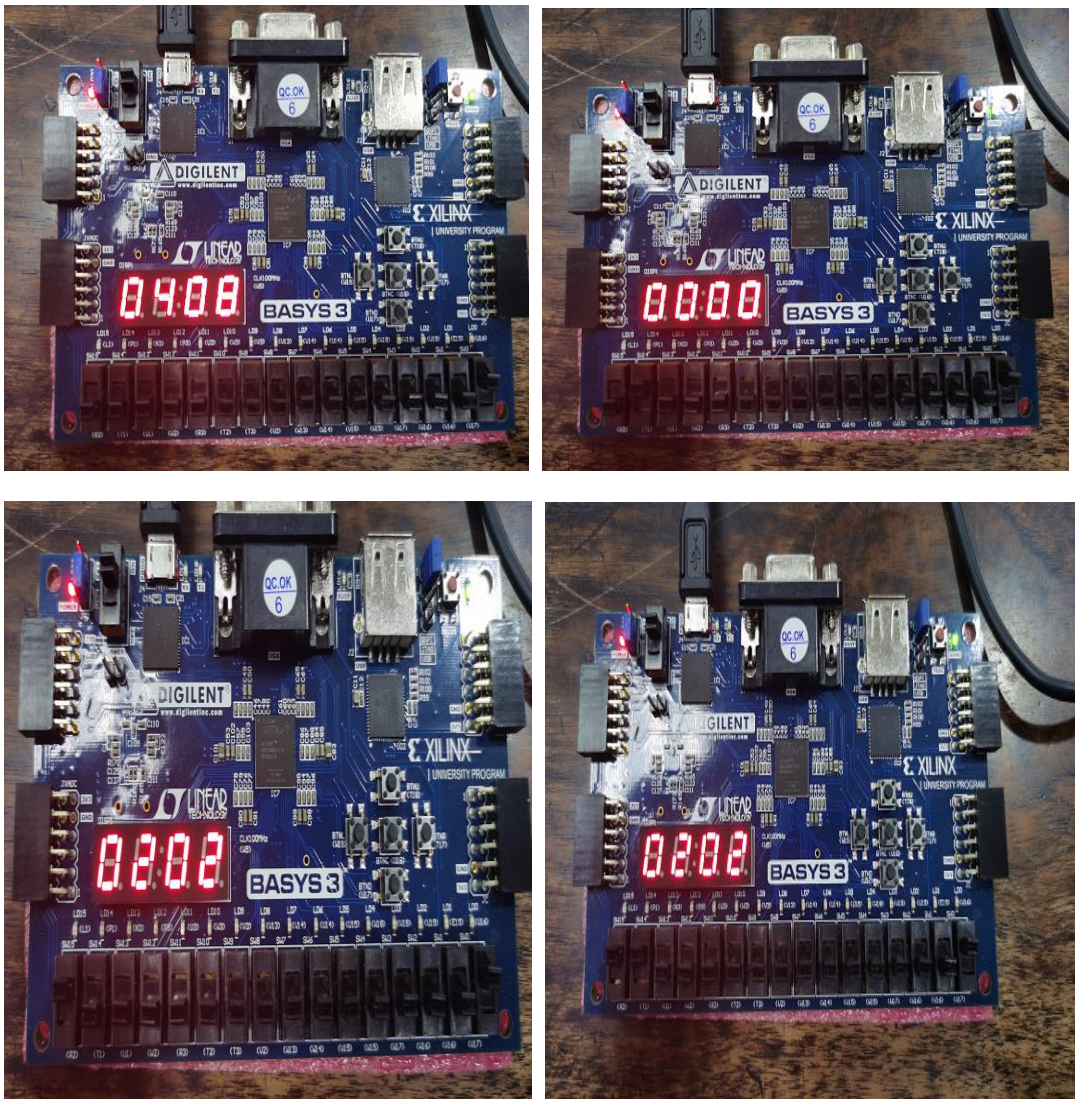
当前PC值为00，下条指令PC值为04

rs寄存器地址为00，rs寄存器数据为00

rt寄存器地址为01，rt寄存器数据为08

ALU结果输出为08，DB总线数据为08

② ori $2,$0,2



图片从左到右，从上到下，分别对应为：

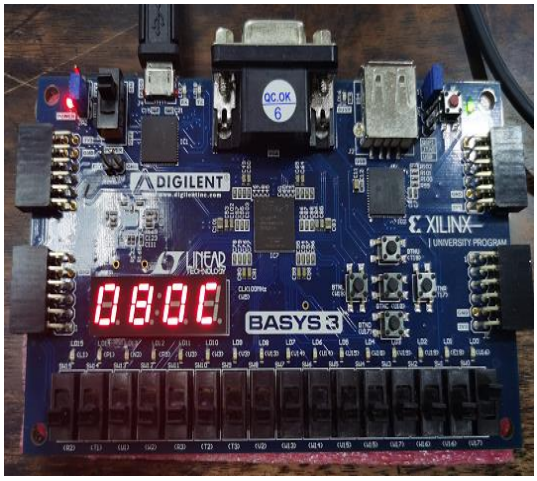
当前PC值为04，下条指令PC值为08

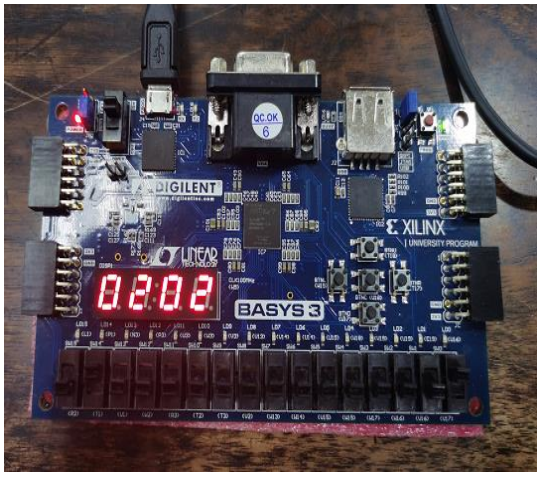
rs寄存器地址为00，rs寄存器数据为00

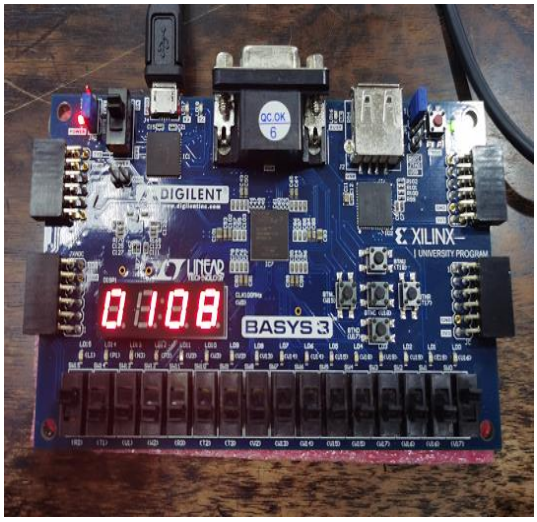
rt寄存器地址为02，rt寄存器数据为02

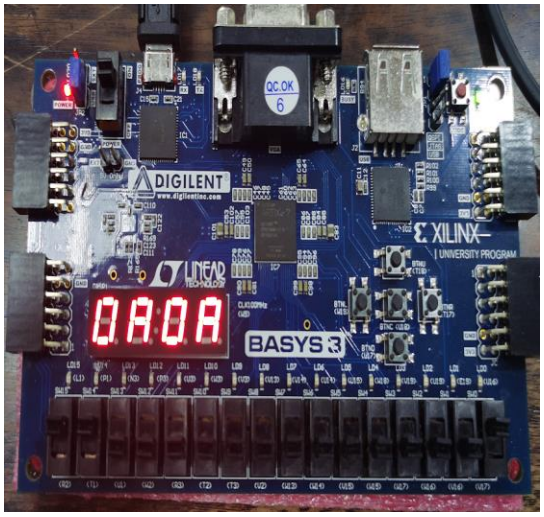
ALU结果输出为02，DB总线数据为02

③ add $3,$2,$1









图片从左到右，从上到下，分别对应为：

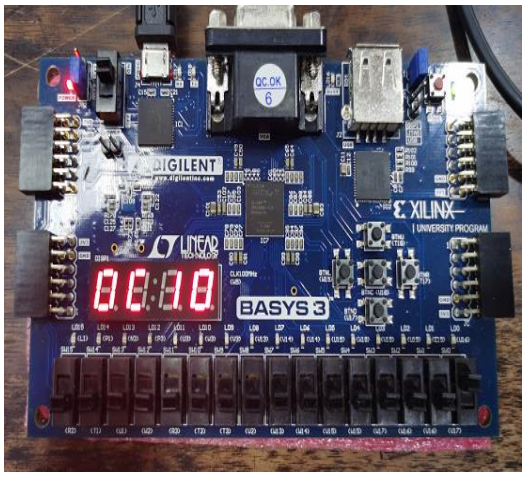
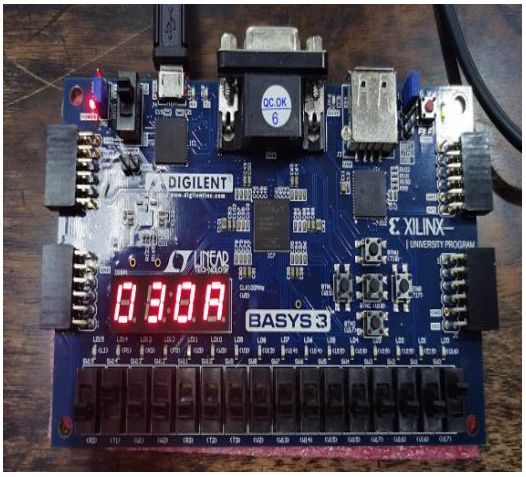
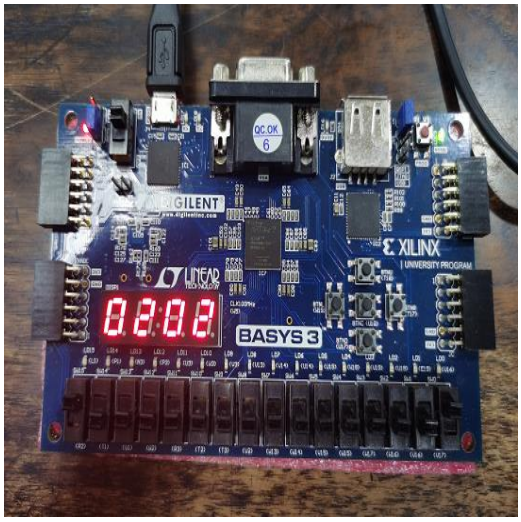
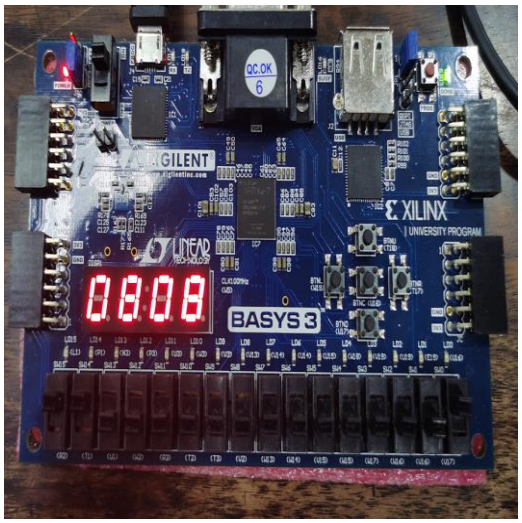
当前PC值为08，下条指令PC值为0C

rs寄存器地址为02，rs寄存器数据为02

rt寄存器地址为01，rt寄存器数据为08

ALU结果输出为0a，DB总线数据为0a

④ sub $5,$3,$2

图片从左到右，从上到下，分别对应为：

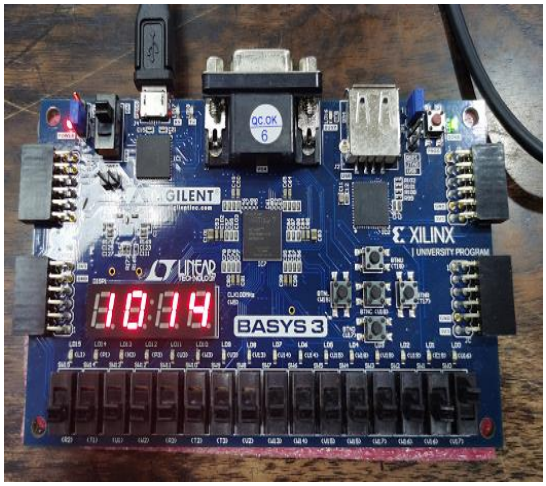
当前PC值为0C，下条指令PC值为10

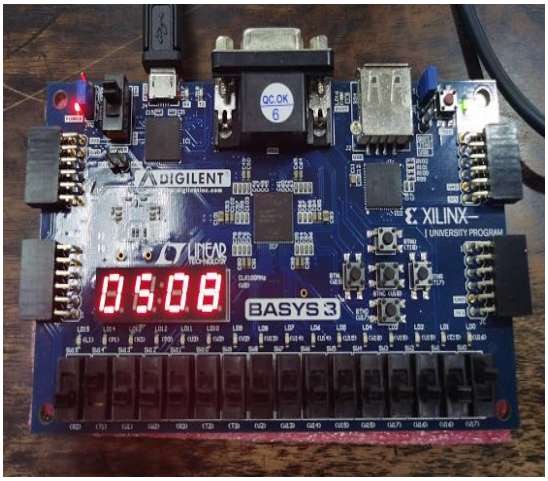
rs寄存器地址为03，rs寄存器数据为0A

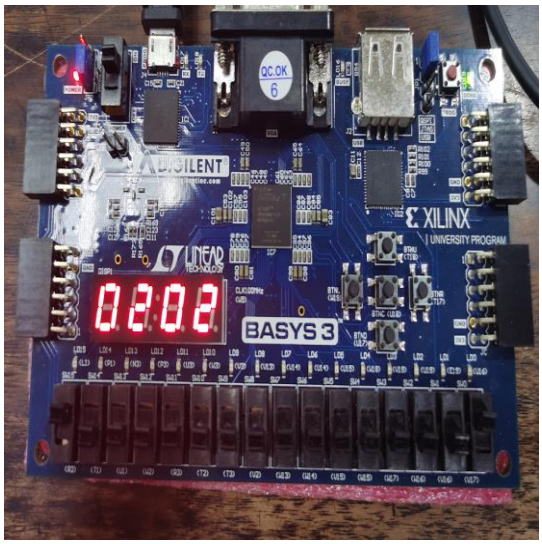
rt寄存器地址为02，rt寄存器数据为02

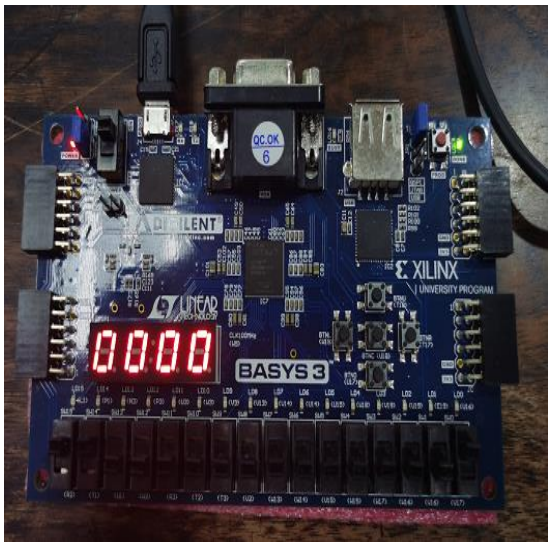
ALU结果输出为08，DB总线数据为08

⑤ and $4,$5,$2









图片从左到右，从上到下，分别对应为：

当前PC值为10，下条指令PC值为14

rs寄存器地址为05，rs寄存器数据为08

rt寄存器地址为02，rt寄存器数据为02

ALU结果输出为00，DB总线数据为00

六. 实验心得

本次实验让我真的收获到了很多。从一开始的不知所措到后来完成整个实验，写下这篇实验报告，我经历了多次心态上的变化，并花大量时间克服了实验过程中遇上的各种各样的问题。

一开始拿到这个实验的时候，我根本不知道如何下手。感觉和课上的关联程度不是

特别大。在上网查阅了一些相关资料和参考代码之后，才慢慢有了一些思路。

这个实验其实写完之后我才发现并没有我想象中的那么难，只要能够把下面这张数据通路图理解清楚，然后再用verilog语言实现各个模块即可。

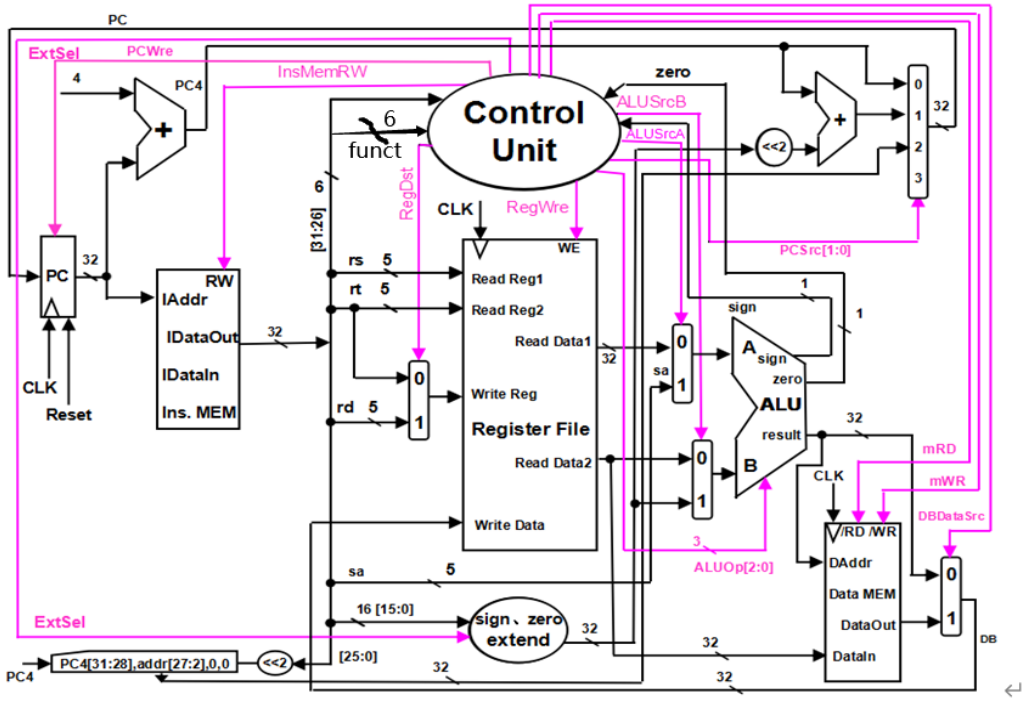


图 2　单周期 CPU 数据通路和控制线路图

其实每个模块的实现步骤都是一样的：（1）确立该模块的输入输出；（2）确定该模块的功能，根据功能写出该模块的具体内容.

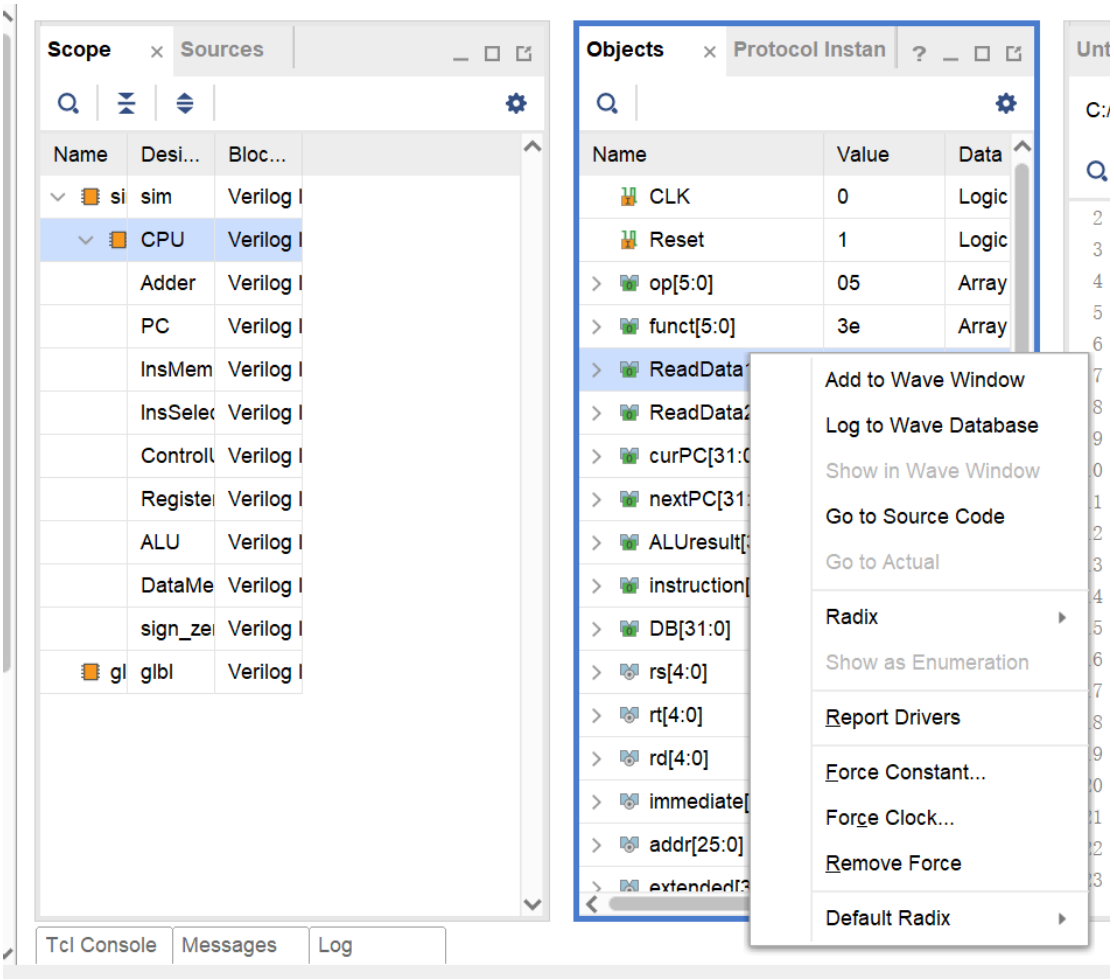
最后写出顶层文件CPU来连接各个模块即可完成CPU的整体构造。

其实在实验初期时，verilog语言的学习对我来说也是一大难题。因为之前是完全没有接触过这门语言，所以只好自己在网上下载了指导教材以及往年该实验的参考代码来逐步学习体会。好在这次实验所需要用到的知识并不算太多，大多都是verilog的基础语法，所以在后期，verilog语言本身并没有给我造成太大的阻碍。

实验过程中我曾多次遇到变量名的问题，每个模块里都有很多变量，并且大多数变量与其它模块也有联系（输入输出的关系），所以如果写错了大小写，就会导致系统报错或者波形图显示异常（如一直高阻抗或XX或一直为0）。不过如果报错的话，可以在Message窗口中寻找相关信息，查看哪里写错了变量，所以这个问题并没有给我造成太大的麻烦。

在实验运行时，我一开始不懂得如何往波形图中添加我想看的变量的波形，只能每次往CPU顶层文件中CPU的初始化列表里添加这一变量并加上新的数据通路。但后来

我发现可以在scope里面找到我想要的变量再直接添加到波形图中即可，这大大减少了我在实验过程中调试代码的时间（可以迅速找出是哪个输入或输出变量出了问题）



此外，我在参考网上相关代码的时候，还对代码进行了一定程度上的优化：即对于CPU中独立存在的那些选择器，我发现可以把这些选择器归并到与其相关的某个模块当中，这样就使得整个代码结构更加精简，并且易于理解。具体归并过程可见我的代码分析部分。

总而言之，这次实验让我对于CPU的组成有了更深层次的理解，明白了CPU各部分的功能以及各部分之间的联系。同时也对vivado软件的使用有了初步的了解，并且可以看懂并复现出大多数verilog的基础代码。