《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：刘硕

学　　号：16340154

时　　间：2018 年 5 月 30 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。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：

⑴ 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。

⑵ 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。

⑶ 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。

⑷ 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。

⑸ 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。

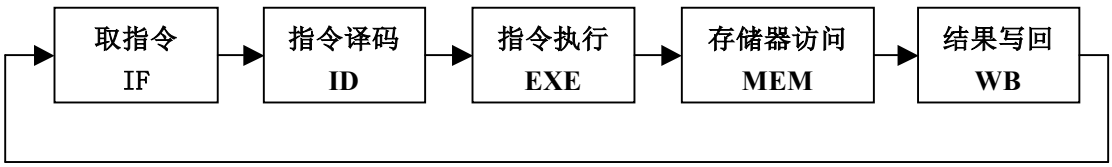
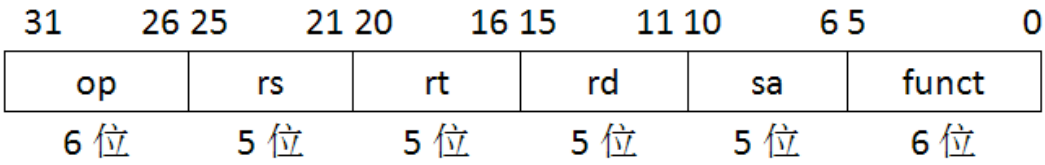


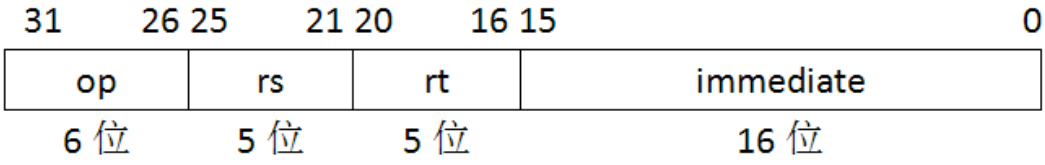
图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：

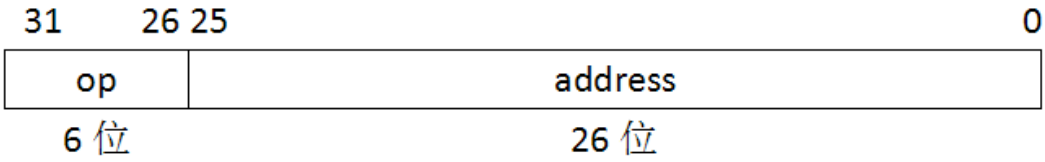
R 类型：



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。

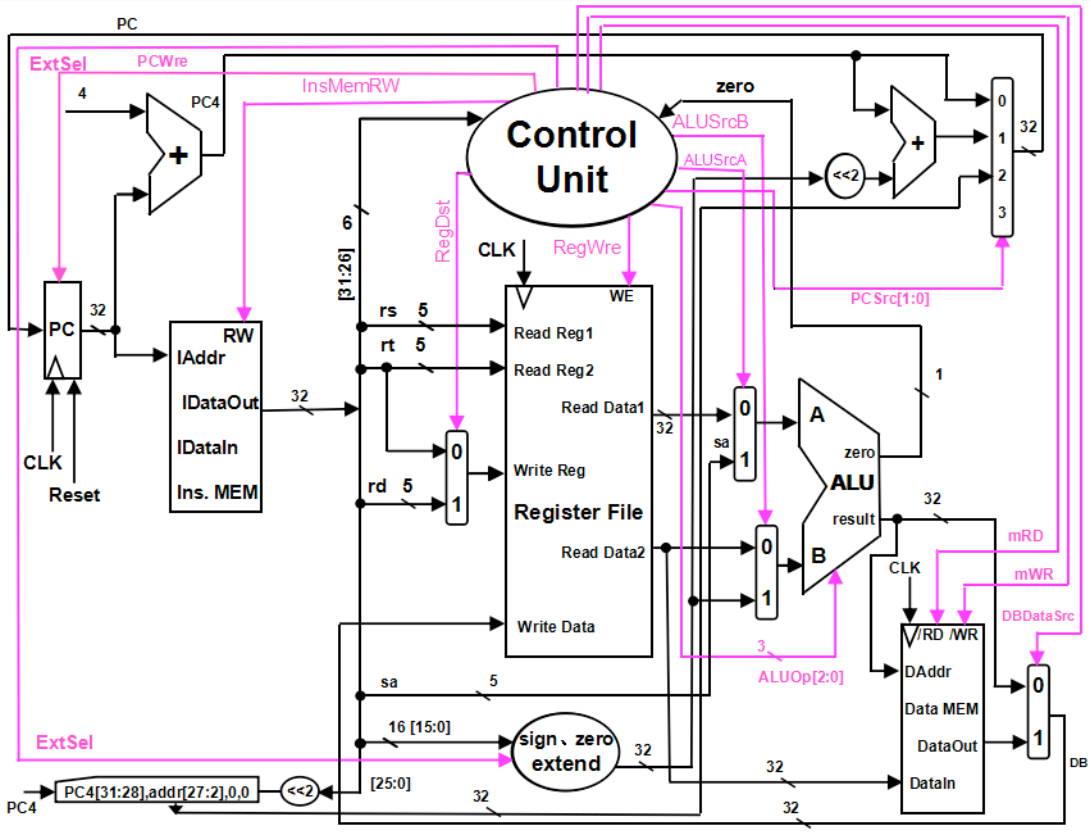


图 2　单周期 CPU 数据通路和控制线路图

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、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)
mRD(RD)	输出高阻态	**读数据存储器，相关指令：lw**
mWR(WR)	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

四．实验器材

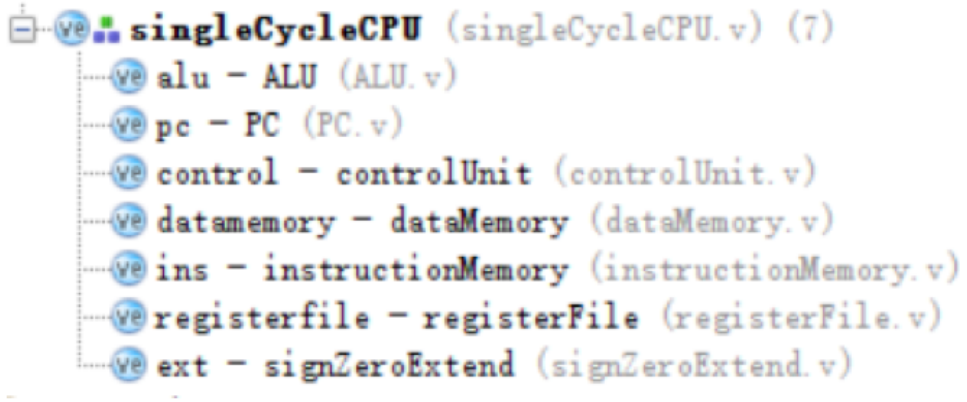
电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五．实验过程与结果

1．设计思路

此次实验实现一个单周期CPU（一条指令在一个时间周期内完成），并实现一些基本的指令操作。实现思想便是对CPU划分模块细化实现，并在顶层构建控制文件。

根据上文中的数据通路和控制线路图，发现单周期CPU数据通路和控制线路图可以分为七个单元,它们本别是:逻辑运算单元(ALU)、程序计数器(PC)、控制单元(controlUnit)、数据存储器单元（dataMemory）、指令存储器单元（instructionMemory）、寄存器单元（registerFile）、符号扩展单元（signZeroExtend）。这七个单元都受控于单周期CPU文件（sigleCycleCPU），它们的依赖关系如图所示。



而控制信号表则决定了每一个指令所对应的控制信号,控制单元部分需要产生各种控制信号，也有些信号必须要传送给控制单元。然后加上一个顶层文件（singleCycleCPU）以及仿真数据（led_sim1），便可以对单周期CPU进行仿真实验。控制信号表如下图所示。

控制信号与指控关系表

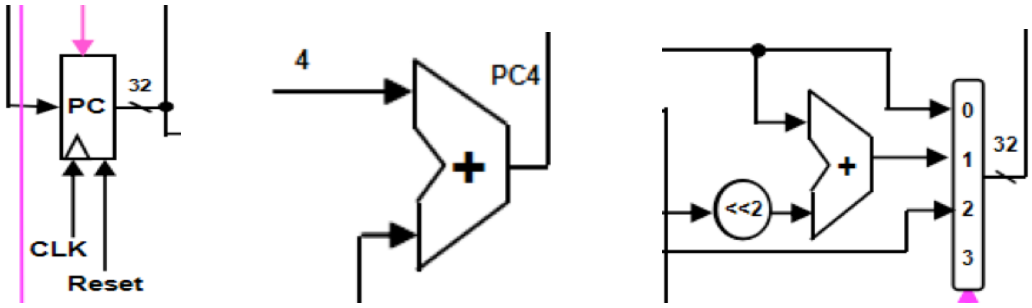
	z	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	InsMemRW	mRD	mWR	RegDst	ExtSel	PCSrc[1..0]	ALUOp[2..0]
add	x	1	0	0	0	1	1	0	0	1	0	00	000
addi	x	1	0	1	0	1	1	0	0	0	1	00	000
sub	x	1	0	0	0	1	1	0	0	1	0	00	001
ori	x	1	0	1	0	1	1	0	0	0	0	00	011
and	x	1	0	0	0	1	1	0	0	1	0	00	100
or	x	1	0	0	0	1	1	0	0	1	0	00	011
sll	x	1	0	0	0	1	1	0	0	1	0	00	010
slti	x	1	1	1	0	1	1	0	0	0	1	00	110
sw	x	1	0	1	0	0	1	0	1	0	1	00	000
lw	x	1	0	1	1	1	1	1	0	0	1	00	000
beq	0	1	0	0	0	0	1	0	0	0	1	00	111
	1	1	0	0	0	0	1	0	0	0	1	01	111
bne	0	1	0	0	0	0	1	0	0	0	1	01	111
	1	1	0	0	0	0	1	0	0	0	1	00	111
j	x	1	0	0	0	0	1	0	0	0	0	10	000
halt	x	0	0	0	0	0	0	0	0	0	0	xx	Xxx

2. 具体实现

根据数据通路和控制线路图，可以得到每个单元的信号传送情况。对各个单元模块的作用进行分析，并给出相应信号的变化：

（1）PC 单元：

单元功能：PC单元负责给指令记数，根据指令向前或者向后跳转；跳转的方式有3种——值加4跳转到下一条指令（由PC+4单元实现）、不满足判断指令后跳转新地址（PC+4单元和跳转立即数共同决定）、跳转到给定地址（新地址）；如下三图分别表示PC单元的实现、PC+4单元、选择PC跳转的功能单元。



单元信号：PCWre信号用于控制PC是否更改，当停机的时候PC值不再更改，PCWre为0。PCSrc信号控制为输入，用于控制不同的指令信号——00时执行R型指令、01时执行I型指令、10执行J指令、11未定义操作，输出当前PC 地址。

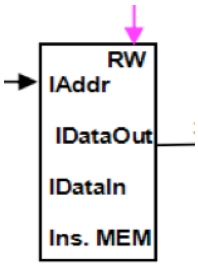
单元线路：PC单元以时钟信号clk触发，CPU是单周期说明每个时钟周期内完成一条指令的读写。有着重置标志Reset，当重置标志为1时接受新的地址，否则比如停机时初始化PC值0。IDataOut符号扩展和零扩展后的立即数ExtOut经过移位和PC+4共同决定跳转指令执行后PC的新地址。

```
module PC (clk, Reset, PCWre, PCSrc, ExtOut, curPC, IDataOut);

    input clk, Reset, PCWre;
    input [1:0] PCSrc;
    input [31:0] ExtOut, IDataOut;
    output [31:0] curPC;
    reg [31:0] curPC;

    // 等待clk时钟上升沿或reset的下降沿
    always @(posedge clk or negedge Reset)
    begin
        // Reset PC 地址
        if (Reset == 0)
        begin
            curPC = 0;
        end
        // 接受新地址
        else if (PCWre)
        begin
```

```
            if (PCSrc == 2'b01)
            // 条件语句跳转
            begin
                curPC = curPC + 4 + ExtOut * 4;
            end
            // 无条件直接跳转——j型指令
            else if (PCSrc == 2'b10)
            begin
                curPC[31:28] = curPC[31:28];
                curPC[27:2] = IDataOut[25:0];
                curPC[1:0] = 2'b00;
            end
            else
            // 正常向后跳转4位
            begin
                curPC = curPC + 4;
            end
        end
    end
endmodule
```

（2）instructionMemory 指令存储器单元：

单元功能：instructionMemory 是指令存储器单元，用来存储 32 位指令。下图表示为指令存储器单元。



单元信号：控制信号 InsMemRW 是控制读写的信号，0 为写操作、1 为读操作。

单元线路：输入当前的 32 位 curPC。输出对应 32 位指令代码 IDataOut。

```
module instructionMemory (pc, InsMemRW, IDataOut);

    input pc[31:0];
    input InsMemRW;
    output [31:0] IDataOut;
    // 测试指令需要长度为 68 数组储存
    wire [8:0] mem[0:67];

    // addi $1, $0, 8
    assign mem[0] = 8'h04;
    assign mem[1] = 8'h01;
    assign mem[2] = 8'h00;
    assign mem[3] = 8'h08;
    // ori $2, $0, 2
    assign mem[4] = 8'h40;
    assign mem[5] = 8'h02;
    assign mem[6] = 8'h00;
    assign mem[7] = 8'h02;
    // add $3, $2, $1
    assign mem[8] = 8'h00;
    assign mem[9] = 8'h41;
    assign mem[10] = 8'h18;
    assign mem[11] = 8'h00;
    // sub $5, $3, $2
    assign mem[12] = 8'h08;
    assign mem[13] = 8'h62;
    assign mem[14] = 8'h28;
    assign mem[15] = 8'h00;
    // and $4, $5, $2
    assign mem[16] = 8'h44;
    assign mem[17] = 8'hA2;
    assign mem[18] = 8'h20;
    assign mem[19] = 8'h00;
    // or $8, $4, $2
    assign mem[20] = 8'h48;
    assign mem[21] = 8'h82;
    assign mem[22] = 8'h40;
    assign mem[23] = 8'h00;
    // sll $8, $8, 1
    assign mem[24] = 8'h60;
    assign mem[25] = 8'h08;
    assign mem[26] = 8'h40;
    assign mem[27] = 8'h40;
    // bne $8, $1, -2
```

```
assign mem[28] = 8'hC5;
assign mem[29] = 8'h01;
assign mem[30] = 8'hFF;
assign mem[31] = 8'hFE;
// slti $6, $2, 8
assign mem[32] = 8'h6C;
assign mem[33] = 8'h46;
assign mem[34] = 8'h00;
assign mem[35] = 8'h08;
// slti $7, $6, 0
assign mem[36] = 8'h6C;
assign mem[37] = 8'hC7;
assign mem[38] = 8'h00;
assign mem[39] = 8'h00;
// addi $7, $7, 8
assign mem[40] = 8'h04;
assign mem[41] = 8'hE7;
assign mem[42] = 8'h00;
assign mem[43] = 8'h08;
// beq $7, $1, -2
assign mem[44] = 8'hC0;
assign mem[45] = 8'hE1;
assign mem[46] = 8'hFF;
assign mem[47] = 8'hFE;
// sw $2, 4($1)
assign mem[48] = 8'h98;
assign mem[49] = 8'h22;
assign mem[50] = 8'h00;
assign mem[51] = 8'h04;
// lw $9, 4($1)
assign mem[52] = 8'h9C;
assign mem[53] = 8'h29;
assign mem[54] = 8'h00;
assign mem[55] = 8'h04;
// j 0x00000040
assign mem[56] = 8'hE0;
assign mem[57] = 8'h00;
assign mem[58] = 8'h00;
assign mem[59] = 8'h10;
// addi  $10,$0,10
assign mem[60] = 8'h04;
assign mem[61] = 8'h0A;
assign mem[62] = 8'h00;
assign mem[63] = 8'h0A;
```

```
    // halt
    assign mem[64] = 8'hFC;
    assign mem[65] = 8'h00;
    assign mem[66] = 8'h00;
    assign mem[67] = 8'h00;

    // 输出 IDataOut
    assign IDataOut[31:24] = mem[pc][7:0];
    assign IDataOut[23:16] = mem[pc + 1][7:0];
    assign IDataOut[15:8] = mem[pc + 2][7:0];
    assign IDataOut[7:0] = mem[pc + 3][7:0];
endmodule
```

（3）signZeroExtend 单元：

单元功能：对于 I 型指令，由于指令功能需要，指令中的立即数可能需要符号扩展或者零扩展——比如用作无符号的逻辑操作数、有符号的算数操作数、数据加载和数据保存的地址字节偏移量、分支指令的由符号偏移量。IDataOut 线路经过指令单元细分，得到的立即数用 signZeroExtend 单元进行零扩展和符号扩展。扩展单元如图。

单元信号：控制信号 Extsel 用来判定进行扩展的方式——1 为符号扩展、0 为零扩展。

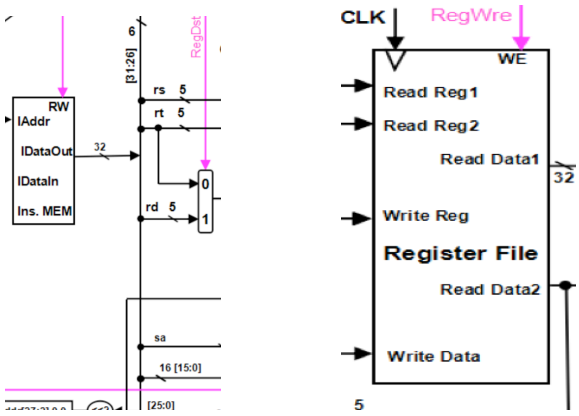
单元线路：输入 I 指令中的后 16 位立即数 immediate。输出拓展后的 32 位结果 ExtOut。

```
module signZeroExtend(immediate, ExtSel, ExtOut);
    input [15:0] immediate;
    input ExtSel;
    output [31:0] ExtOut;
```

```
        assign ExtOut[15:0] = immediate;
        // 单元扩展方式
        assign ExtOut[31:16] = ExtSel ? (immediate[15] ? 16'hffff : 16'h0000) :
16'h0000;
    endmodule
```

（4）registerFile 寄存器单元：

单元功能：registerFile 是寄存器单元，寄存器是有限储存容量的高速存储部件。指令分类经过，找到了读寄存器读数据和写寄存器写数据。如下两个图分别表示指令按功能细分部分和寄存器读写部分。



单元信号： RegWre 表示寄存器组写使能，如果有新的数值要被写入寄存器，那么该信号为 1。控制信号 RegDst 的 0 和 1 分表示指令目标寄存器是来自 rt 还是 rd（是来自 R 型指令还是 I 型指令）。DBDataSrc 信号区分来自 ALU 运算结果进行输出还是来自数据存储器的结果进行输出。

单元线路：registerFile 以时钟信号 clk 触发。IdataOut 输出的指令按功能被细分。对于 R 型指令，指令被分成 op、funct、rs、rt、rd、sa：op 被传入 Control Unit 控制各个信号；rs、rt 分别被传入 Read Reg1 和 Read Reg2，找到读寄存器的序号；rd 被传到 Write Reg，找到写寄存器的序号；根据读寄存器的序号找到 Read Data1 和 Read

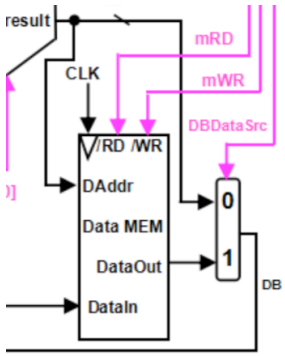
Data2；根据写寄存器的序号，把由 ALU 单元运算结果的输出或者由数据存储器的输出 Write Data 写入写寄存器。对于 I 型指令，指令被分成 rs、rt、immediate：op 被传入 Control Unit 控制各个信号；rs 被传入 Read Reg1，找到读寄存器的序号；rt 被传到 Write Reg，找到写寄存器的序号；immediate 经过扩展单元的扩展，被直接传到 ALU 进行各类运算（与寄存器单元无关，这里列举只是为了体系化）；根据读寄存器的序号找到 Read Data1；根据写寄存器的序号，把由 ALU 单元运算结果的输出或者由数据存储器的输出 Write Data 写入写寄存器。。对于 J 型指令，指令被细分成 op、addr：op 被传入 Control Unit 控制各个信号；addr 直接传给 PC+4 部件进行跳转。

```
module registerFile(clk, RegWre, RegDst, rs, rt, rd, DBDataSrc, Result, DMOut, Out1, Out2);

    input clk, RegDst, RegWre, DBDataSrc;
    input [4:0] rs, rt, rd;
    input [31:0] Result,DMOut;
    output [31:0] Out1, Out2;
    wire [4:0] writeReg;
    wire [31:0] writeData;
    reg [31:0] register[0:31];

    assign writeReg = RegDst ? rd : rt;
    assign writeData = DBDataSrc ? DMOut : Result;

    // 写数据
    integer i;
    initial
    begin
        for (i = 0; i < 32; i = i + 1)
            register[i] <= 0;
    end

    assign Out1 = register[rs];
    assign Out2 = register[rt];

    // 读数据
```

```
    always @(posedge clk)
    if (RegWre)
    begin
        register[writeReg] = writeData;
    end
endmodule
```

（5）dataMemory 数据存储器单元：

单元功能：dataMemory 是数据存储器单元，用来存储数据，在进行 lw、sw 指令的时候，会用到数据存储器。数据存储器为 8 位，需要 4 个存储器存储一条数据。数据存储器如图所示。



单元信号：信号 mRD、mWR 为 1 时分别表示这是读数据存储器或写数据存储器。

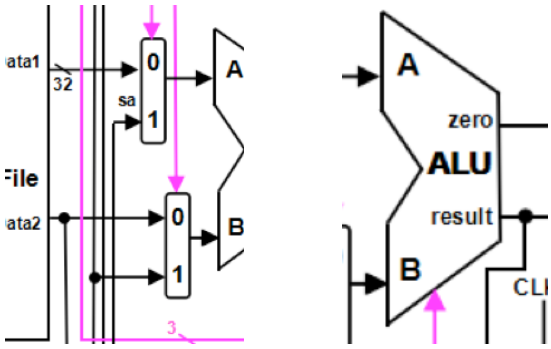
单元线路：dataMemory 用 CLK 时钟信号触发。 rs 寄存器中的数与立即数在 ALU 单元进行算数运算后的结果用作储存器的高位地址传入 DAddr 数据地址输入端口。执行读数据存储器操作的时候，输入来自从寄存器中读到的 Read Data2；执行写数据存储器操作的时候，输出从数据存储器中的值到 DataOut。

```
module dataMemory (Result,Out2, RD, WR, DMOut);

    input [31:0] Result, Out2;
    input RD, WR;
```

```
    output reg [31:0] DMOut;
    reg [7:0] memory[0:31];

    always @(RD or Result)
    begin
        // 读数据
        if (RD == 1)
        begin
            DMOut[7:0] <= memory[Result + 3];
            DMOut[15:8] <= memory[Result + 2];
            DMOut[23:16] <= memory[Result + 1];
            DMOut[31:24] <= memory[Result];
        end
    end

    integer i;
    initial
    begin
        for (i = 0; i < 32; i = i + 1)
            memory[i] <= 0;
    end

    always @(WR or Result or Out2)
    begin
        // 写数据
        if (WR == 1)
        begin
            memory[Result] = Out2[31:24];
            memory[Result + 1] = Out2[23:16];
            memory[Result + 2] = Out2[15:8];
            memory[Result + 3] = Out2[7:0];
        end
    end
endmodule
```

（6）ALU 逻辑运算单元：

单元功能：ALU 单元负责算数逻辑运算，有时运算是两个寄存器中的存数进行算数逻辑运算，有时候是一个寄存器中的存数和一个立即数（扩展后）进行运算，还有可能进行移位操作。如下两个图分别表示 ALU 操作数输入选择和 ALU 运算单元。



单元信号：ALUSrcA 是 ALU 单元一个重要的控制信号，当 ALUSrcA 信号为 0 的时候，ALU 的第一个操作数来自从寄存器中读出数据 Read Data1，当 ALUSrcA 信号为 1 的时候，ALU 的第一个操作数为移位数 sa。ALUSrcB 是 ALU 单元另一个重要的控制信号，当 ALUSrcB 信号为 0 的时候，ALU 的第二个操作数来自从寄存器中读出数据 Read Data2，当 ALUSrcB 信号为 1 的时候，ALU 的第二个操作数为扩展后的立即数。ALUOp 信号控制 ALU8 种功能选择。

单元线路：按指令功能选择的两个操作数被输入 ALU 单元。输出 ALU 运算结果 result。zero 运算结果标志——判断结果是否为 0。

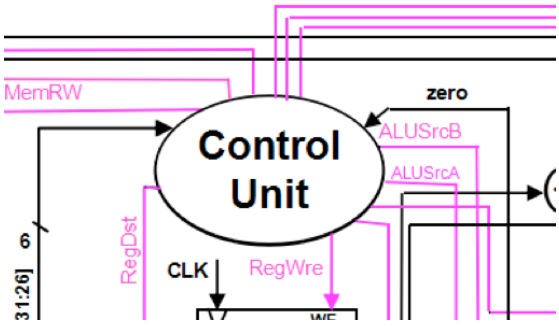
```
    module ALU (Out1 ,Out2, sa, ExtOut, ALUSrcA, ALUSrcB, ALUOp, zero, sign,
Result);
        input [31:0] Out1, Out2, ExtOut;
        input [4:0] sa;
        input ALUSrcB, ALUSrcA;
        input [2:0] ALUOp;
        output zero;
        output sign;
        output [31:0] Result;
        reg sign;
        reg zero;
        reg [31:0] Result;
        wire [31:0] B;
        wire [31:0] A;
        assign A = ALUSrcA ? {{27{0}},sa} : Out1;
        assign B = ALUSrcB ? ExtOut : Out2;
```

```
        always @(Out1 or Out2 or ExtOut or ALUSrcA or ALUSrcB or ALUOp or B or A
or sa)
        // 8 种功能
        begin
            case(ALUOp)
                    // A 加 B
                3'b000:
                begin
                    Result = A + B;
                    zero = (Result == 0) ? 1 : 0;
                    sign = (Result[31] == 0) ? 0 : 1;
                end
                    // A 减 B
                3'b001:
                begin
                    Result = A - B;
                    zero = (Result == 0)? 1 : 0;
                    sign = (Result[31] == 0) ? 0 : 1;
                end
                    // B 左移 A 位
                3'b010:
                begin
                    Result = B << A;
                    zero = (Result == 0)? 1 : 0;
                    sign = (Result[31] == 0) ? 0 : 1;
                end
                    // A 与 B
                3'b011:
                begin
                    Result = A | B;
                    zero = (Result == 0) ? 1 : 0;
                    sign = (Result[31] == 0) ? 0 : 1;
                end
                    // A 或 B
                3'b100:
                begin
                    Result = A & B;
                    zero = (Result == 0) ? 1 : 0;
                    sign = (Result[31] == 0) ? 0 : 1;
                end
                    // 比较 A 与 B 不带符号
                3'b101:
                begin
                    if(A < B) Result = 1;
```

```
                else Result = 0;
                zero = (Result == 0) ? 1 : 0;
                sign = (Result[31] == 0) ? 0 : 1;
            end
                // 比较 A 与 B 带符号
            3'b110:
            begin
                if(A < B && (A[31] == B[31])) Result = 1;
                else if(A[31]== 1 && B[31] == 0) Result = 1;
                else Result = 0;
                zero = (Result == 0) ? 1 : 0;
                sign = (Result[31] == 0)?0 : 1;
            end
                // A 异或 B
            3'b111:
            begin
                Result = A ^ B;
                zero = (Result == 0) ? 1 : 0;
                sign = (Result[31] == 0) ? 0 : 1;
            end
        endcase
    end
endmodule
```

（7）controlUnit 控制单元：

单元功能：controlUnit 控制单元接受据 IdataOut 分化出的指令的操作码，将不同的指令进行划分并根据划分出的功能参照信号与功能表把各个信号发送给子单元，控制整个指令执行的流程。controlUnit 控制单元功能图如下。



单元信号： PCWre 信号用来确定 PC 是否更改。输出 ALUSrcA、ALUSrcB 信号，明确 ALU 操作单元两个操作数的来源；输出 ALUope 操作码信号来确定 ALU 所需要执行的操作;输出 DBDataSrc 信号区分来自 ALU 运算结果的输出还是来自数据存储器的输出；输出 RegWre 信号判断寄存器是否需要将结果写到一个新的目标寄存器;输出 InsMemRW 信号是写指令存数器还是要读指令存储器；输出 mRD、mWR 信号分别用于确定数据存储器的功能是读或写数据存储器；输出 RegDst 信号确定目标寄存器的地址来自 R 型指令还是 I 型指令；输出 ExtSel 信号进行零扩展还是符号扩展；输出 PCSrc 信号控制 PC 指令地址跳转的方式。

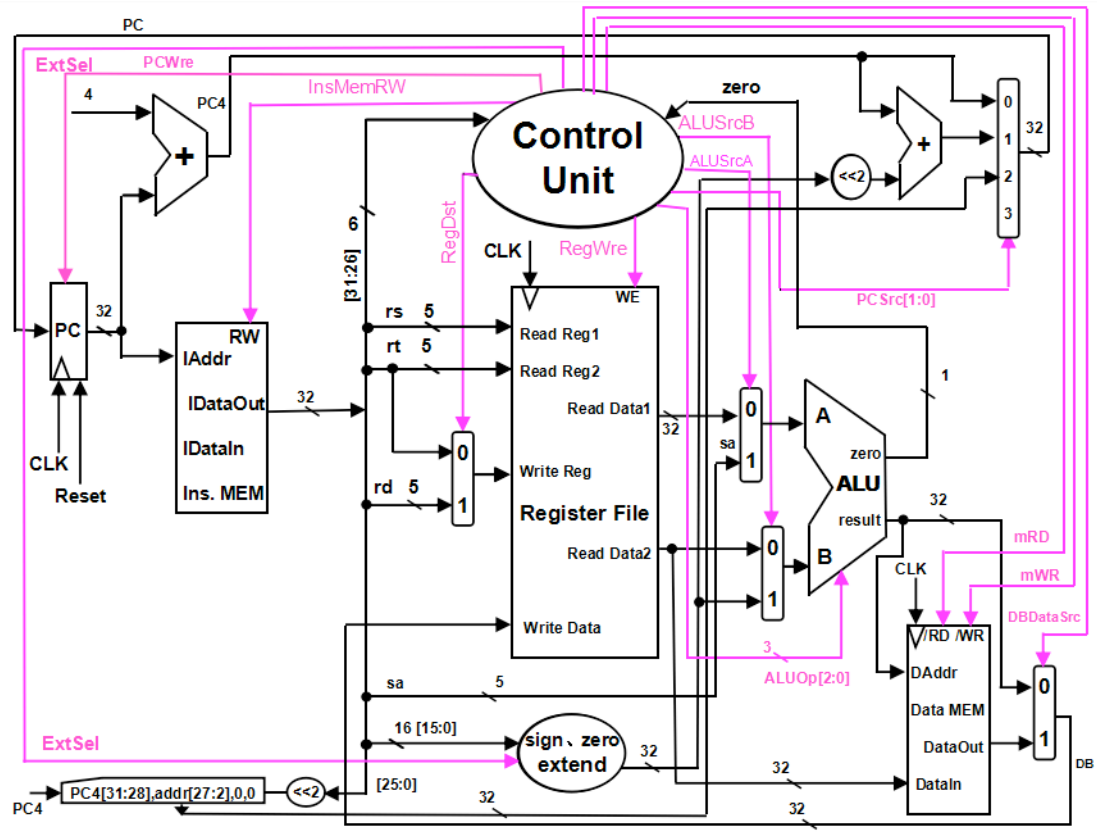
单元线路：指令寄存器中 IdataOut 未扩展的 32 位指令输入；ALU 单元中计算结果 zero 为输入。

```
    module controlUnit (opCode, sign, zero, PCWre, ALUSrcA, ALUSrcB,DBDataSrc,
RegWre, InsMemRW, RD, WR, RegDst, ExtSel,PCSrc, ALUOp);
        input[5:0]opCode;
        input sign;
        input zero;
       output PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, RD, WR, RegDst,
ExtSel;
        output[2:0] ALUOp;
        output [1:0] PCSrc;

        // 控制信号赋值
        assign PCWre = (opCode == 6'b111111) ? 0 : 1;
        assign ALUSrcA = ((opCode == 6'b000000)||(opCode == 6'b000001)||(opCode ==
        6'b000010)||(opCode == 6'b010010)||(opCode == 6'b010001)||(opCode ==
        6'b010000)||(opCode == 6'b110000)||(opCode == 6'b110001)||(opCode ==
        6'b011011)||(opCode == 6'b100110)||(opCode ==6'b100111)) ? 0 : 1;
        assign ALUSrcB = ((opCode == 6'b000001)||(opCode == 6'b010000)||(opCode ==
        6'b011011)||(opCode == 6'b100110)||(opCode == 6'b100111)) ? 1 : 0;
        assign DBDataSrc = (opCode == 6'b100111) ? 1 : 0;
        assign RegWre = ((opCode == 6'b110000)||(opCode == 6'b110001)||(opCode ==
        6'b100110)||(opCode == 6'b111111)||(opCode == 6'b111000)) ? 0 : 1;
        assign InsMemRW = 1;
```

```
        assign RD = (opCode == 6'b100111) ? 1 : 0;
        assign WR = (opCode == 6'b100110) ? 1 : 0;
        assign RegDst = ((opCode == 6'b000001)||(opCode == 6'b010000)||(opCode ==
        6'b100111)||(opCode == 6'b011011)) ? 0 : 1;
        assign ExtSel = (opCode == 6'b010000) ? 0 : 1;
        assign PCSrc[0] = ((opCode == 6'b110000 && zero == 1)||(opCode ==
        6'b110001 && zero == 0)) ? 1 : 0;
        assign PCSrc[1] = (opCode == 6'b111000) ? 1 : 0;
        assign ALUOp[0] = ((opCode == 6'b000010)||(opCode == 6'b010000)||(opCode
== 6'b010010)||(opCode == 6'b110000)||(opCode == 6'b110001)) ? 1 : 0;
        assign ALUOp[1] = ((opCode == 6'b010000)||(opCode == 6'b010010)||(opCode
== 6'b011000) ||(opCode == 6'b011011)) ? 1:0;
        assign ALUOp[2] = ((opCode == 6'b010001)||(opCode == 6'b011011))? 1 : 0;
    endmodule
```

（8）顶层 CPU：

单周期CPU共有七个子模块，每个模块分别负责一个部分的功能。之后按照连接线路图通过一个顶层文件相互连接起来。为了方便，线路图再一次给出（线路图即上文图二）。



```
module CPUmain (clk, Reset, opCode, rs, rt, rd, Out1, Out2, curPC, Result, ExtOut);

            input clk, Reset;
            output [5:0] opCode;
            output [4:0] rs, rt, rd;
            output [31:0] Out1, Out2, curPC, Result, ExtOut
            wire [1:0] PCSrc;
            wire [2:0] ALUOp;
            wire [31:0] IDataOut, DMOut;
            wire [5:0] sa;
            wire [15:0] immediate;
            wire zero, PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, sign, InsMemRW, RD, WR, RegDst, ExtSel;

            // 按功能对指令地址进行分配
            assign opCode = IDataOut[31:26];
            assign rs = IDataOut[25:21];
            assign rt = IDataOut[20:16];
            assign rd = IDataOut[15:11];
            assign sa = IDataOut[10:6];
            assign immediate = IDataOut[15:0];

            // 子模块
            ALU alu(Out1, Out2, sa, ExtOut, ALUSrcA, ALUSrcB, ALUOp, zero, sign, Result);
            PC pc(clk, Reset, PCWre, PCSrc, ExtOut, curPC, IDataOut);
           ontrolUnit control(opCode, sign, zero, PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, RD, WR, RegDst, ExtSel, PCSrc, ALUOp);
            dataMemory datamemory(Result, Out2, RD, WR, DMOut);
            instructionMemory ins(curPC, InsMemRW, IDataOut);
            registerFile registerfile(clk, RegWre, RegDst, rs, rt, rd, DBDataSrc, Result, DMOut, Out1, Out2);
            signZeroExtend ext(immediate, ExtSel, ExtOut);
    endmodule
```

3. 测试阶段

当我们完成了 CPU 的代码设计之后，用一个简单的想法便是设计一个测试程序段，根据指令执行过程的分析来进行检测。因此，我们设计测试程序段如下:

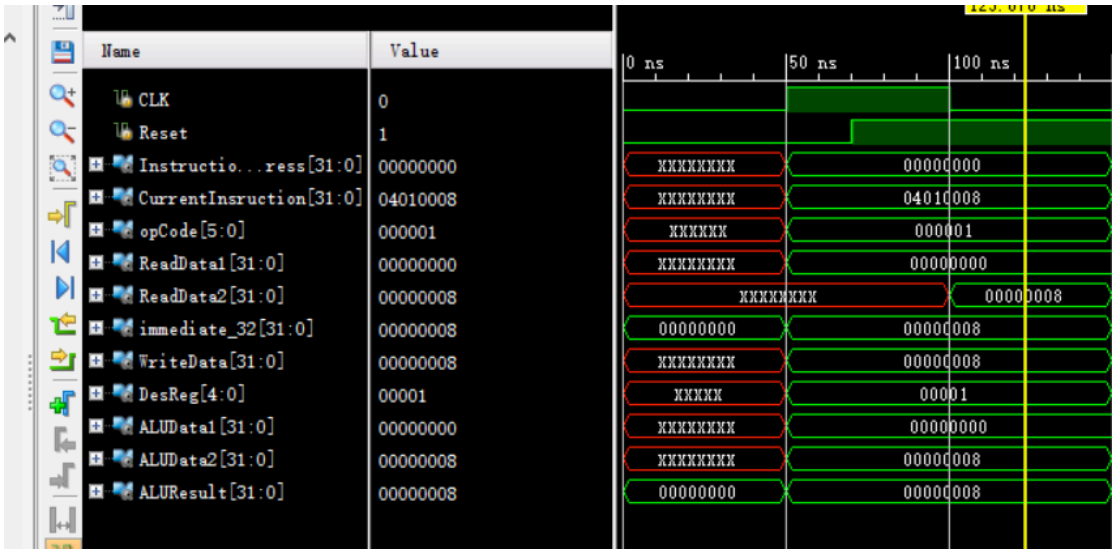
测试程序段

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	=	00411800
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 0100 0000	=	E0000010
0x0000003C	addi $10,$0,10	000001	00000	01010	0000 0000 0000 1010	=	040A000A
0x00000040	halt	111111	00000	00000	0000000000000000	=	FC000000

下面我们对每条测试语句进行详细解释：

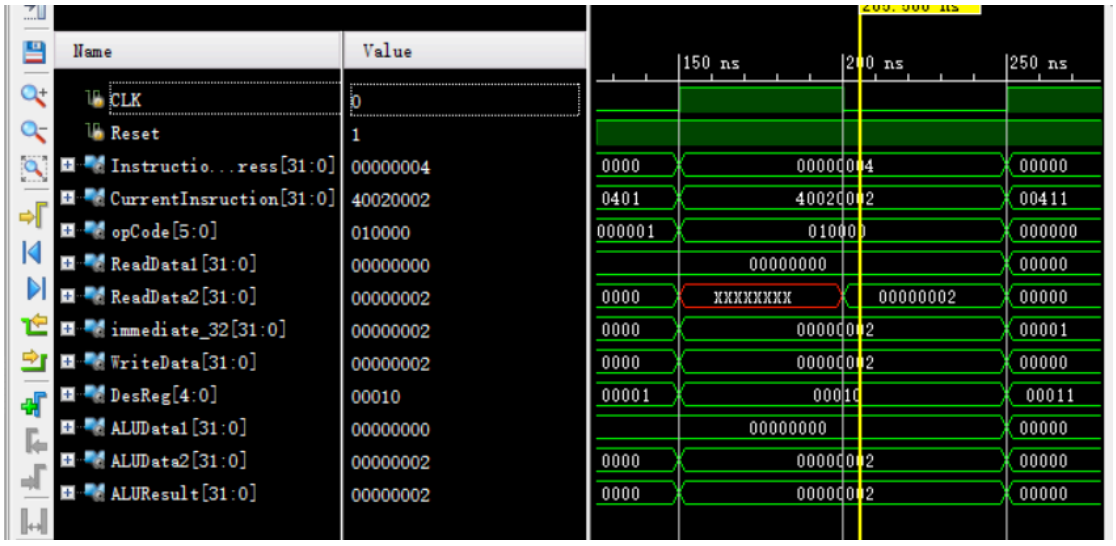
CLK是时钟信号；Reset是重置，1时继续，0时触发；InstructionAddress是指指令的地址；CurrentInstruction是当前指令的十六进制表示；opCode是操作码；ReadData是读取的数据（有ReadData1、ReadData2）、immediate是扩展后立即数的值（对于非I型指令 ，这个值是一个随机数）；WriteData是要写进寄存器的值；DesReg是目标寄存器序号的二进制表示；ALUData和ALUResult表示ALU运算的过程。

（1）addi $1, $0, 8：指令起始地址为0x00000000，该操作执行立即数的加法。将立即数“0x00000008”和0号寄存器中数据“0x00000000”相加得到结果，结果存到目标1号寄存器中。得到1号寄存器的数据为0x00000008，结果如图：



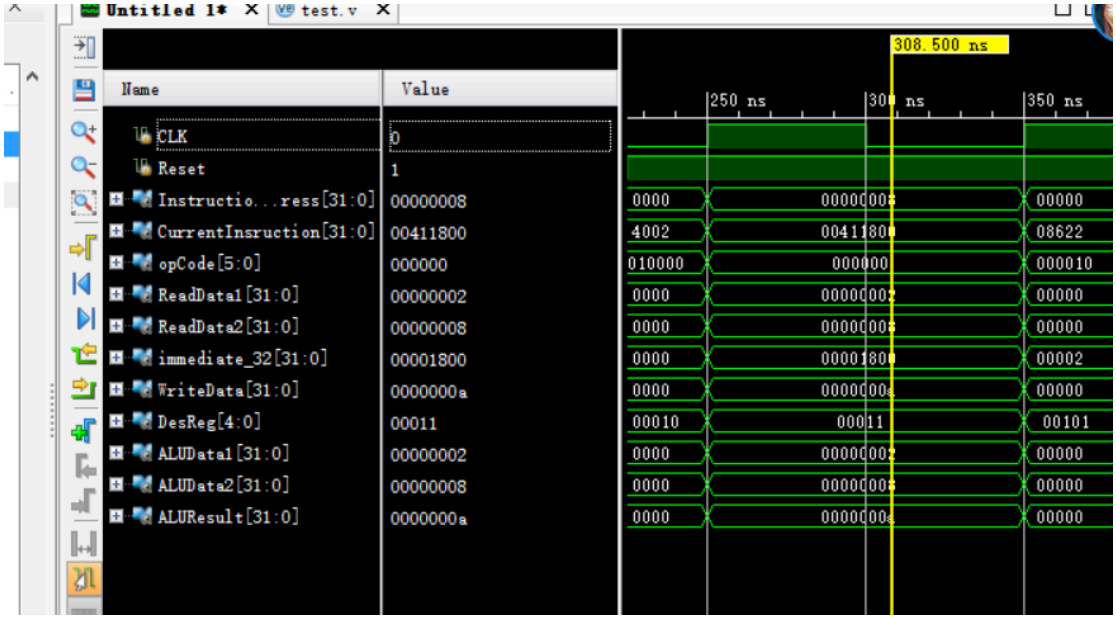
该条指令的结果为 0x00000008，结果正确。

(2) ori $2, $0, 2：指令计数器经过上步操作后加4，则本条指令地址为0x00000004。该条指令是将0号寄存器中的数据“0x00000000”和立即数“0x00000002”进行或运算，可以得到结果0x00000002，将其存入目标寄存器2号寄存器中，结果如图：



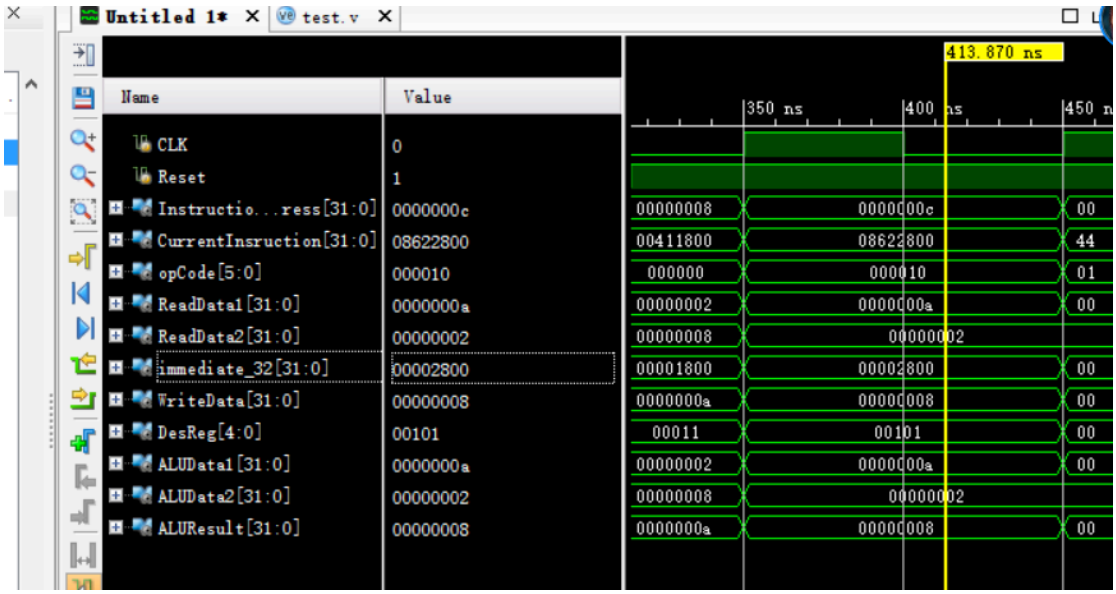
该条指令的结果为0x00000002，结果正确。

（3）add $3，$2，$1：指令计数器经过上步操作后加4，则本条指令地址为0x00000008。该条指令是将1号寄存器中已有的数“0x00000008”和2号寄存器中已有的数“0x00000002”的进行相加，将结果0x0000000a存入目标寄存器3号寄存器中，结果如图：



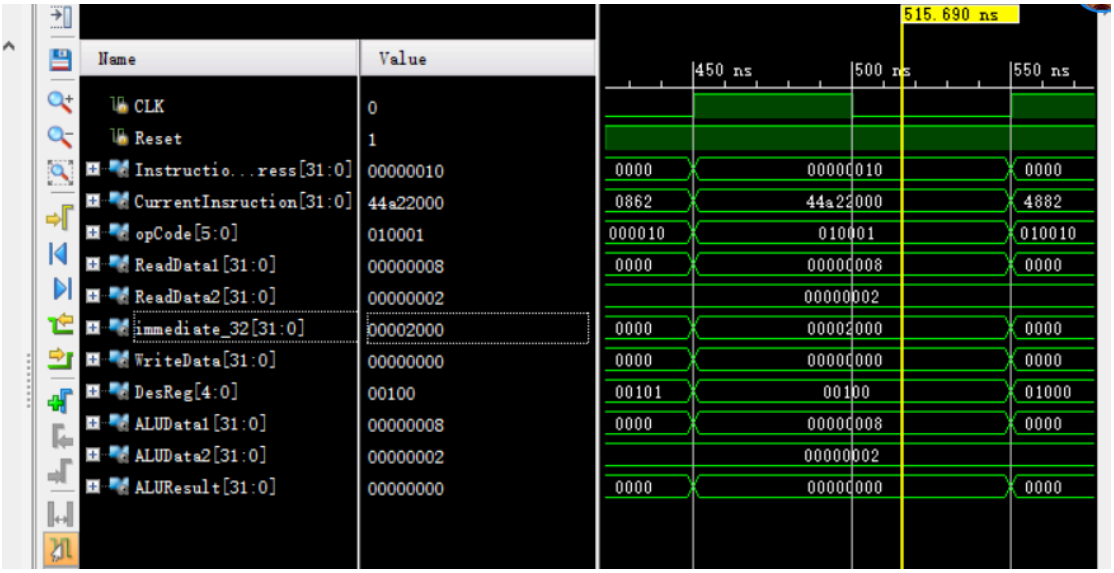
该条指令的结果为0x0000000a，结果正确。

（4）sub $5，$3，$2：指令计数器经过上步操作后加4，则本条指令地址为0x0000000c。该条指令是将3号寄存器中已有的数“0x00000010”和2号寄存器中已有的数“0x00000002”的进行相减，将结果0x00000008存入目标寄存器3号寄存器中，结果如图：



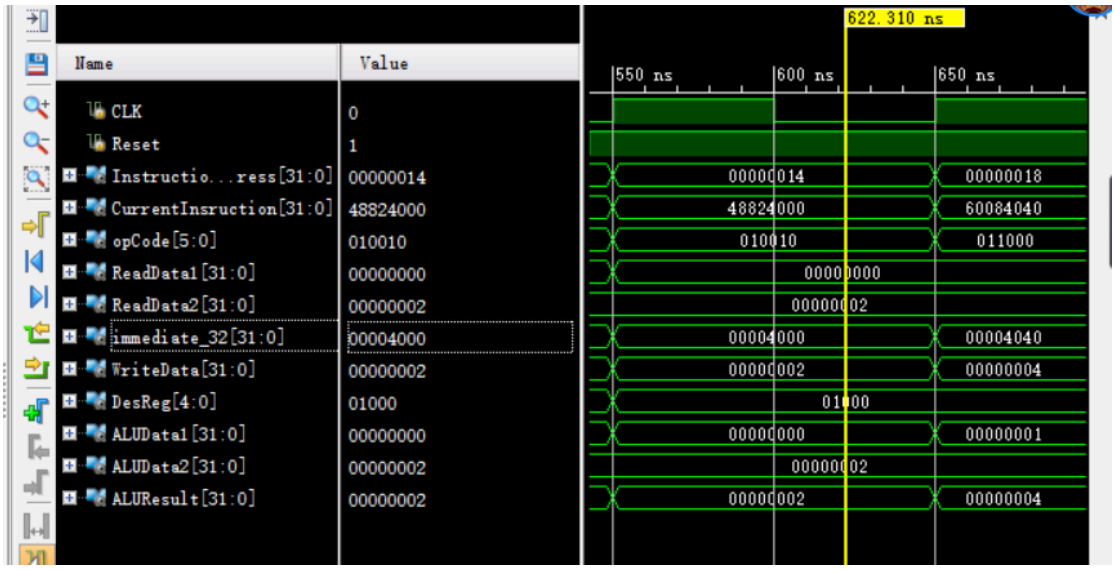
该条指令的结果为0x00000008，结果正确。

（5）and $4，$5，$2：指令计数器经过上步操作后加4，则本条指令地址为0x00000010。该条指令是将5号寄存器中已有的数“0x00000008”和2号寄存器中已有的数“0x00000002”的进行与运算，将结果0x00000000存入目标寄存器4号寄存器中，结果如图：



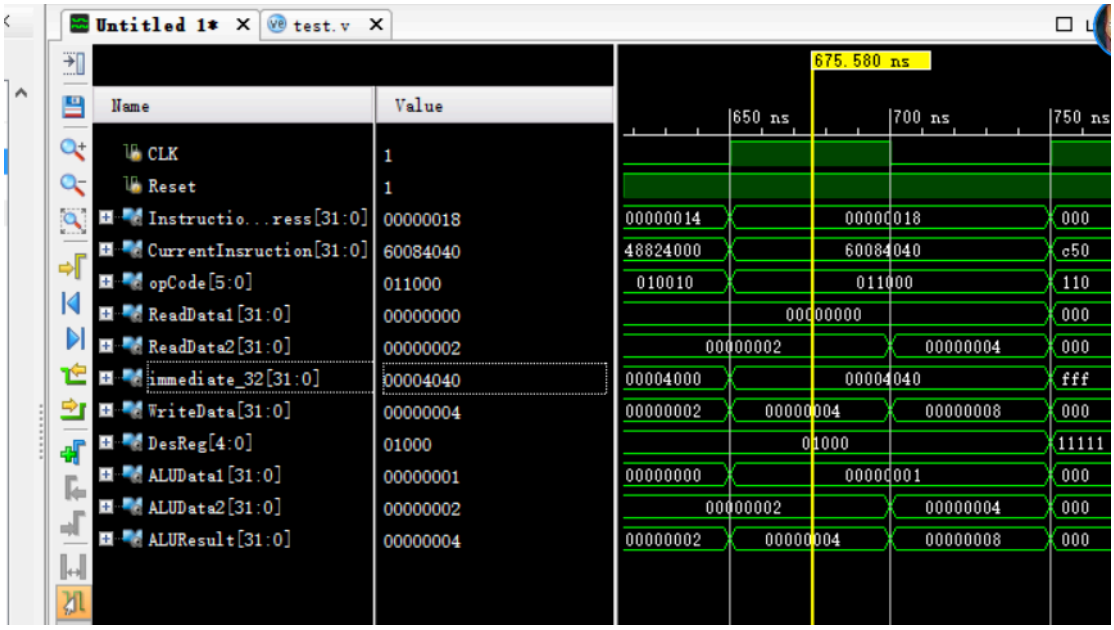
该条指令的结果为0x00000000，可知结果正确。

（6）or $8，$4，$2：指令计数器经过上步操作后加4，则本条指令地址为0x00000014。该条指令是将4号寄存器中已有的数“0x00000000”和2号寄存器中已有的数“0x00000002”的进行或运算。将结果0x00000002存入目标寄存器8号寄存器中，结果如图：



该条指令的结果为0x00000002，结果正确。

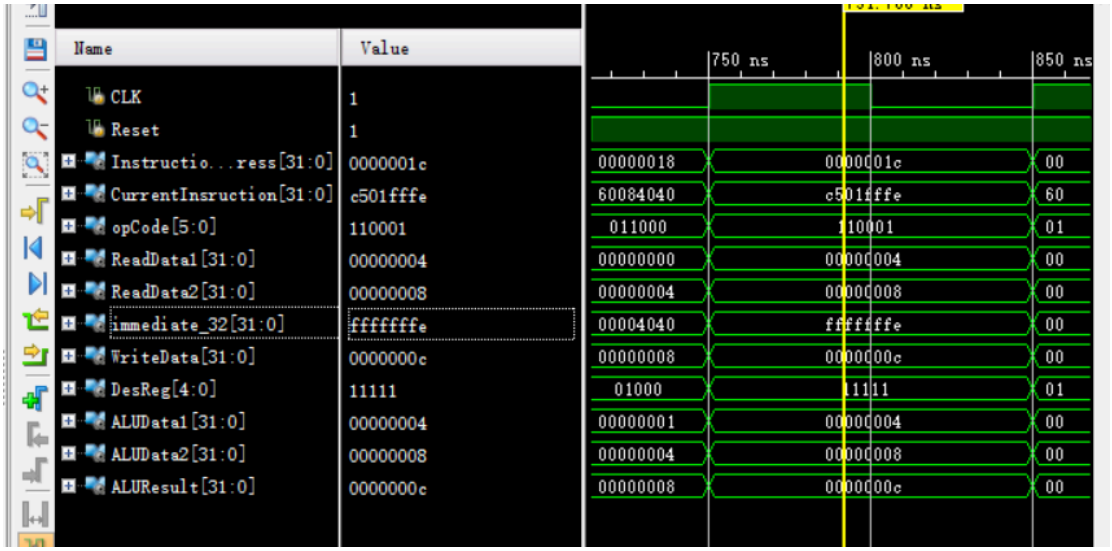
（7）sll $8, $8, 1:指令计数器经过上步操作后加4,则本条指令地址为0x00000018。该条指令是将8号寄存器中已有的数“0x00000002”进行左移立即数“0x00000001”位（也就是乘以2的一次方）的操作，将结果0x00000004存入目标寄存器8号寄存器中，结果如图:



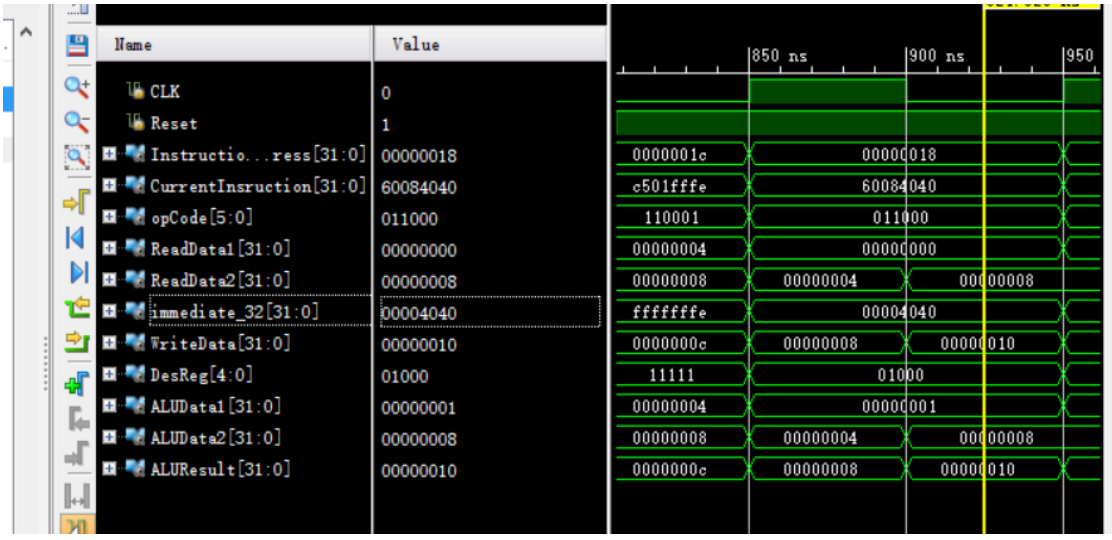
该条指令的结果为0x00000004，结果正确。

（8）bne $8, $1, -2：指令计数器经过上步操作后加4，则本条指令地址为0x0000001c。该条指令是将8号寄存器中已有的数“0x00000004”和1号寄存器中已有的数“0x00000008”进行比较，由于两者不等，则向前跳转2个指令。在执行操作后指令计数器 PC + 4，所以实际上向前跳转了1个指令，继续执

行的指令为上一条指令（sll $8, $8, 1）结果如下图所示：

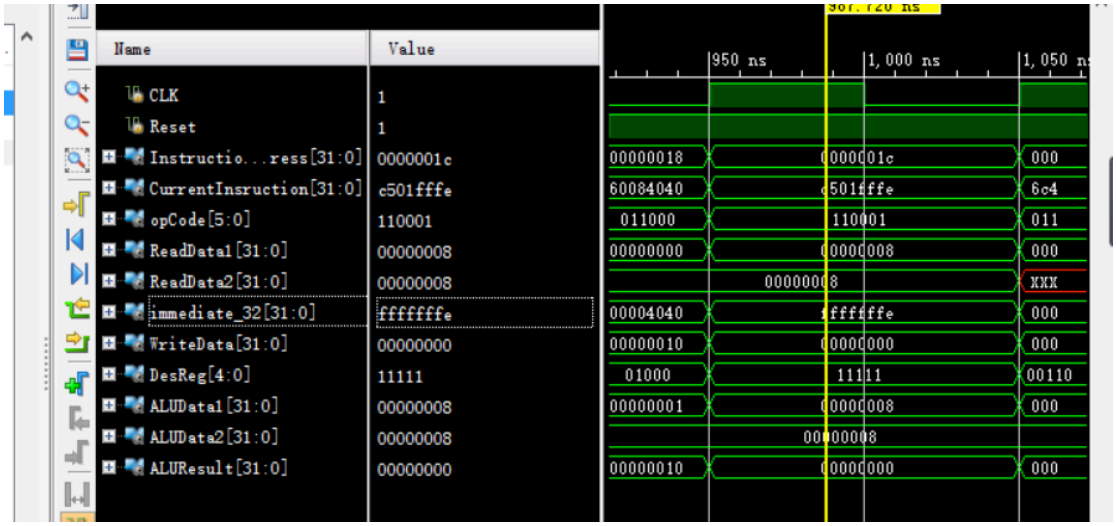


执行完这条指令的地址为0x00000018。将8号寄存器中的数“0x00000004”向左移一位，得到结果0x00000008，存入目标寄存器8号寄存器中，结果如图所示：



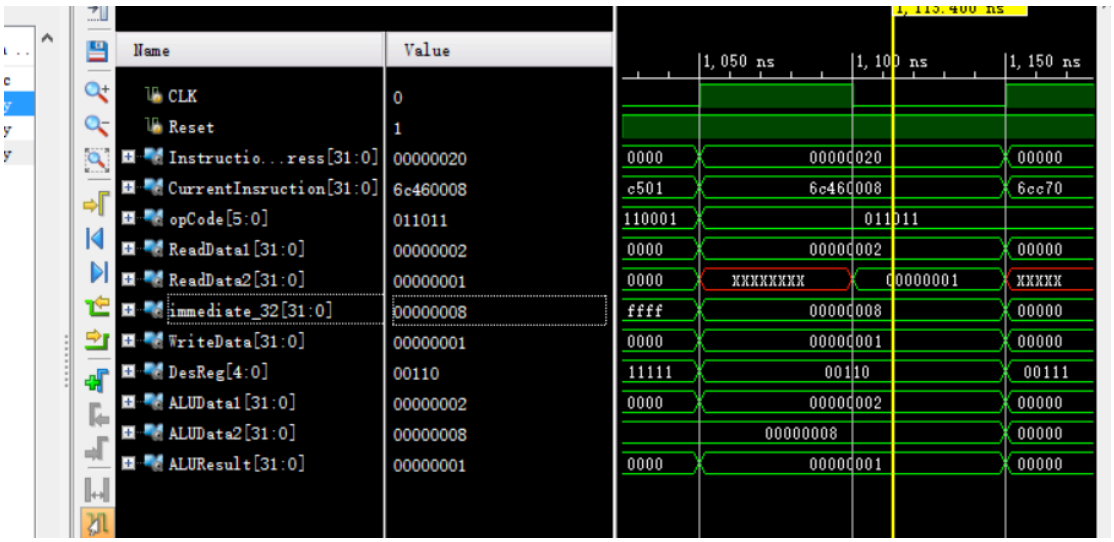
继续运行下一条指令，其地址应该为地址应该为0x0000001c。此时8号寄存器中的值为“0x00000008”，将其和1号寄存器中已有的数“0x00000008”进行比较，相等，不

进行跳转，继续执行下一条指令，结果如图：



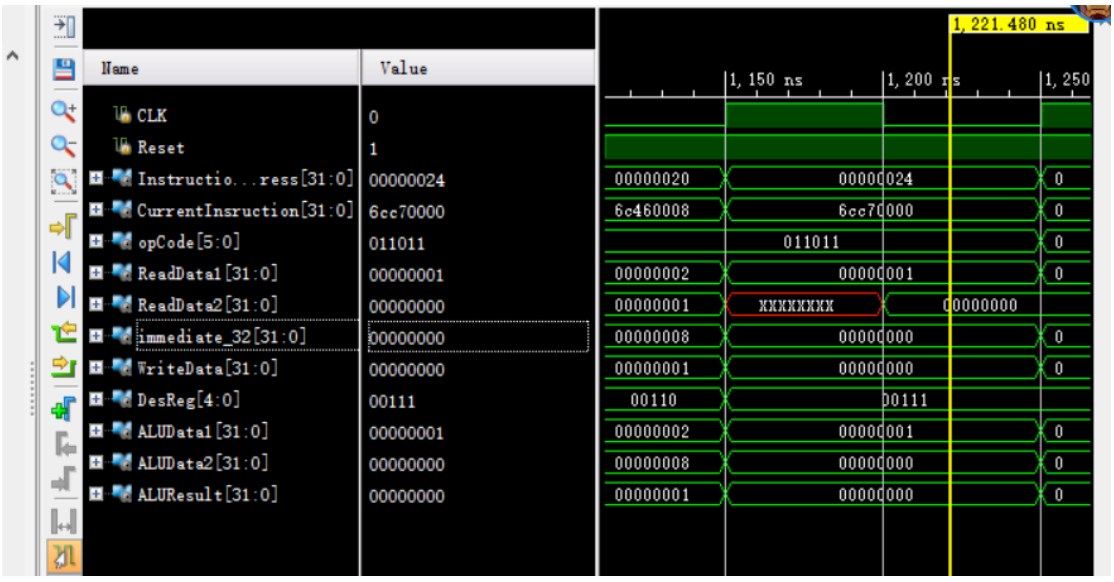
该指令的结果是两者的和，但是这个数不会存到寄存器中，所以没有实际意义。

（9）slti $6，$2，8：指令计数器经过上步操作后加4，则本条指令地址为0x00000020。该条指令是将2号寄存器中已有的数“0x00000002”和立即数“0x00000008”的进行比较，由于0x00000002比0x00000008小，将目标寄存器6号寄存器中的值设为0x00000001，结果如图：



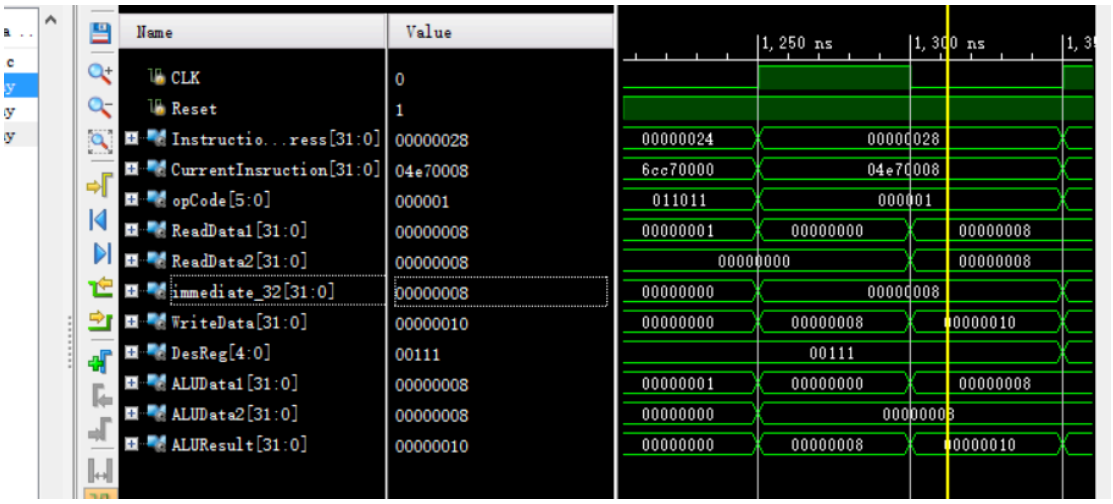
该条指令的结果为0x00000001，结果正确。

（10） slti $7，$6，0：指令计数器经过上步操作后加4，则本条指令地址为0x00000024。该条指令是将6号寄存器中已有的数“0x00000001”和立即数“0x00000000”的进行比较，由于0x00000001比0x00000000大，将目标寄存器7号寄存器中值设为0x00000000，结果如图：



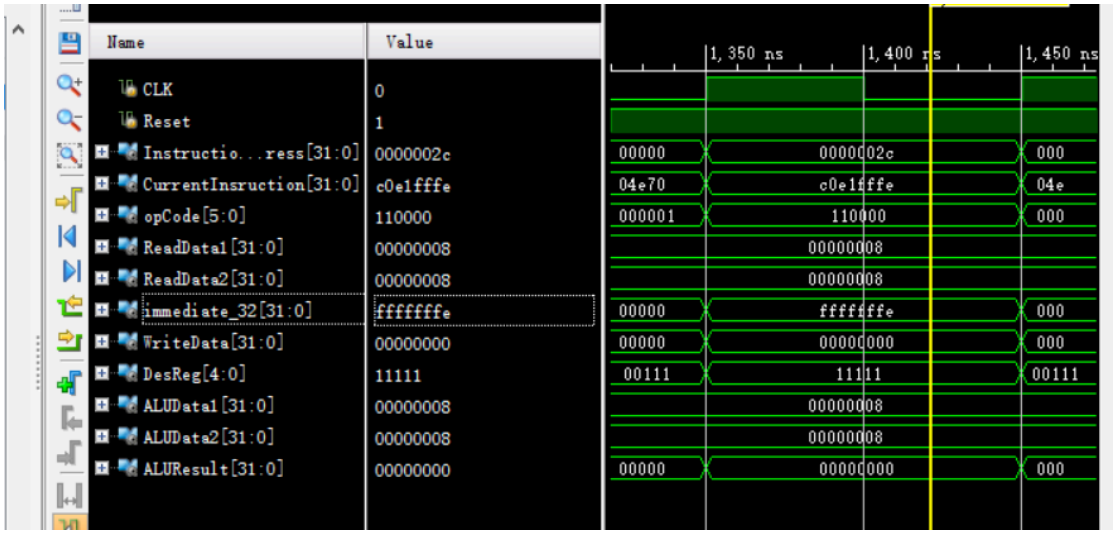
该条指令的结果为0x00000000，结果正确。

（11） addi $7, $7, 8：指令计数器经过上步操作后加4，则本条指令地址为0x00000028。该条指令是将7号寄存器中已有的数“0x00000000”和立即数“0x00000008”的进行相加，将结果0x00000008存入目标寄存器7号寄存器中，结果如下图所示：

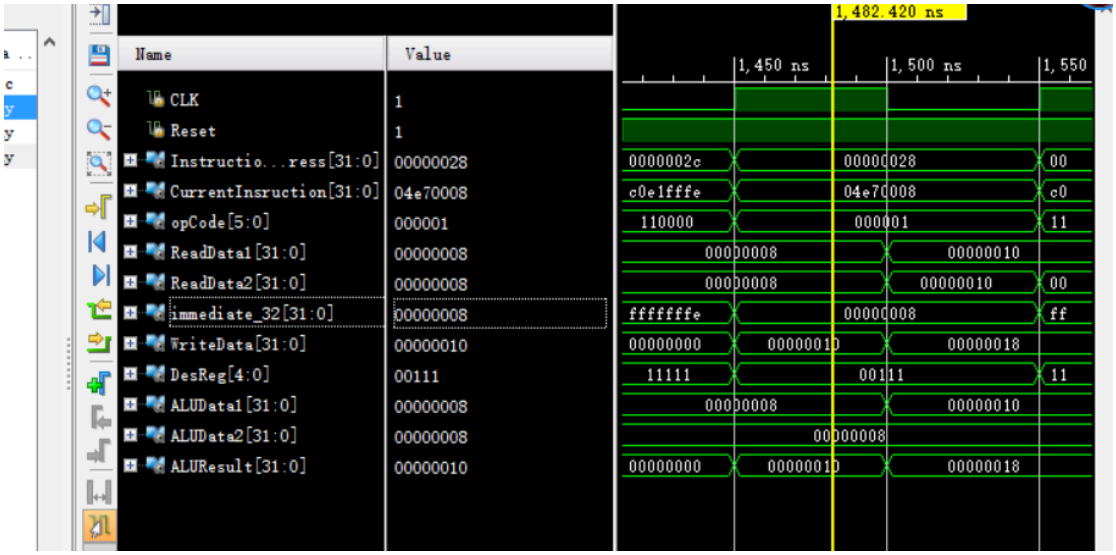


该条指令的结果为0x00000008，结果正确。

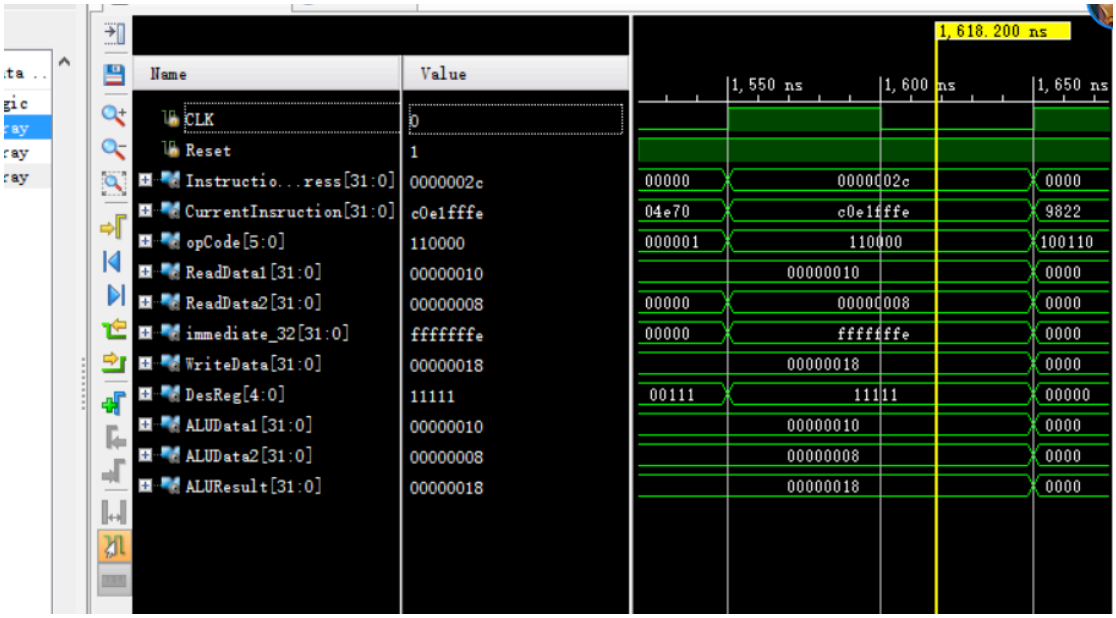
（12） beq $7, $1, -2：指令计数器经过上步操作后加4，则本条指令地址为0x0000002c。该条指令是将7号寄存器中已有的数“0x00000008”和1号寄存器中已有的数“0x00000008”进行比较，相等，向前跳转2个指令，在执行操作后指令计数器 PC+4，所以实际上向前跳转了1个指令，继续执行的指令为上一条指令（addi $7, $7, 1）结果如图：



此条指令的地址应该为0x00000028。将7号寄存器中的数“0x00000008”和立即数“0x00000008”相加，得到结果为0x00000010，将其存入目标寄存器7号寄存器中，结果如图所示：



继续运行下一条指令，其地址应该为地址应该为0x0000002c。此时7号寄存器中的值为“0x00000010”，将其和1号寄存器中已有的数“0x00000008”进行比较，不等，不进行跳转，继续下一条指令，如图所示：

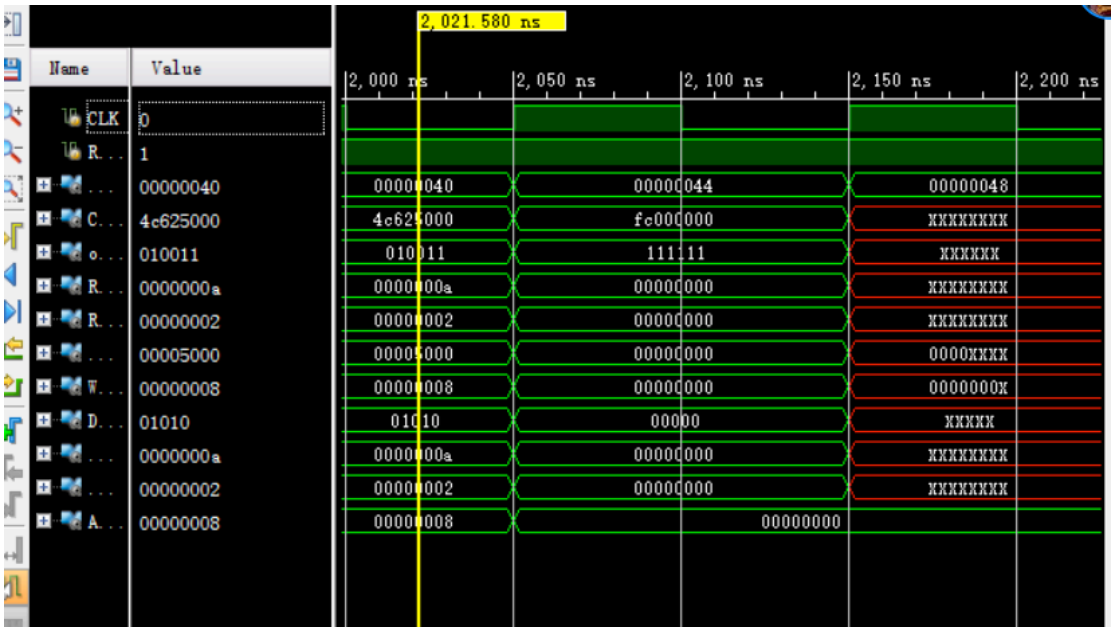


因为该指令的结果是两者的和没有被存到任何一个寄存器，所以没有实际意义。

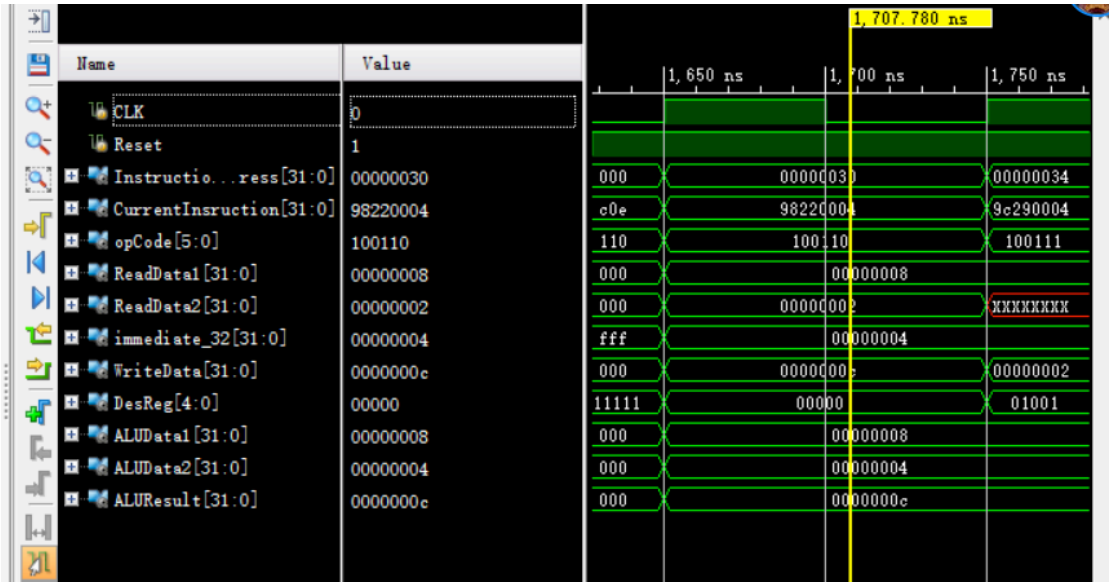
（13）sw $2，4($1)：指令计数器经过上步操作后加4，则本条指令地址为0x00000030。该条指令将1号寄存器中数“0x00000008”与立即数“0x00000004”进行相加，得到结果为0x0000000c，然后需要将2号寄存器中的数据写入第12个数据存储器。

因为一个存储单元由四个存储器组成，第十二个存储器在第三个存储单元，从大端开始。所以把0X00000002存储到第三个存储单元。

存储效果如图：



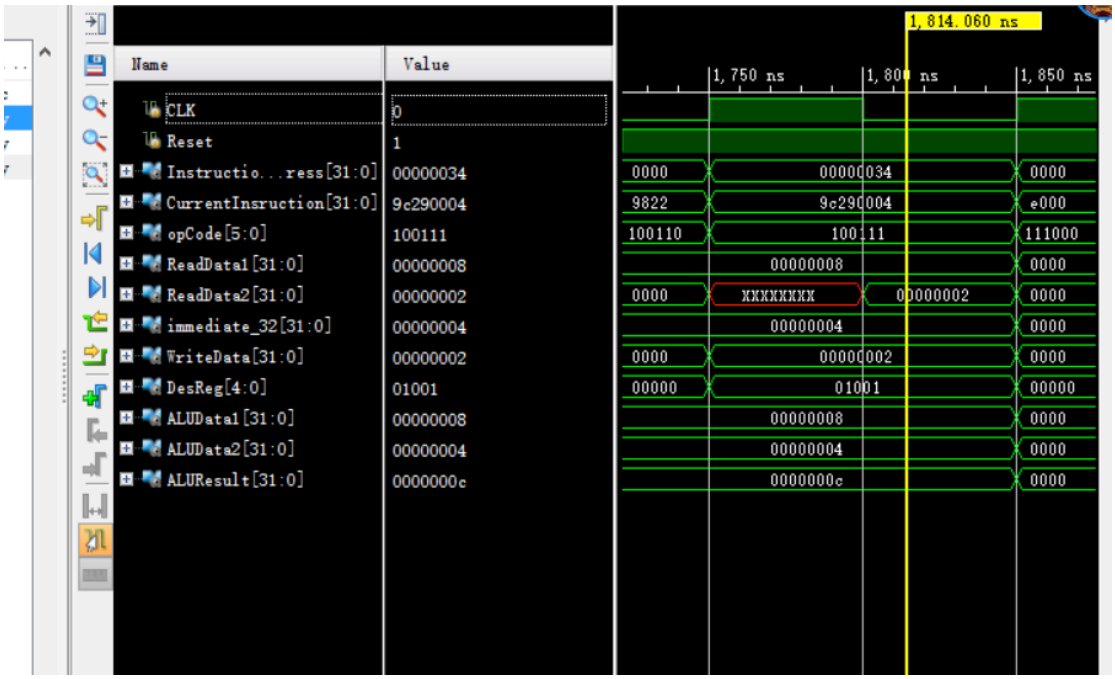
结果如图：



该条指令的结果为0x0000000c，相应的数据存储器得到值0x00000002，结果正确。

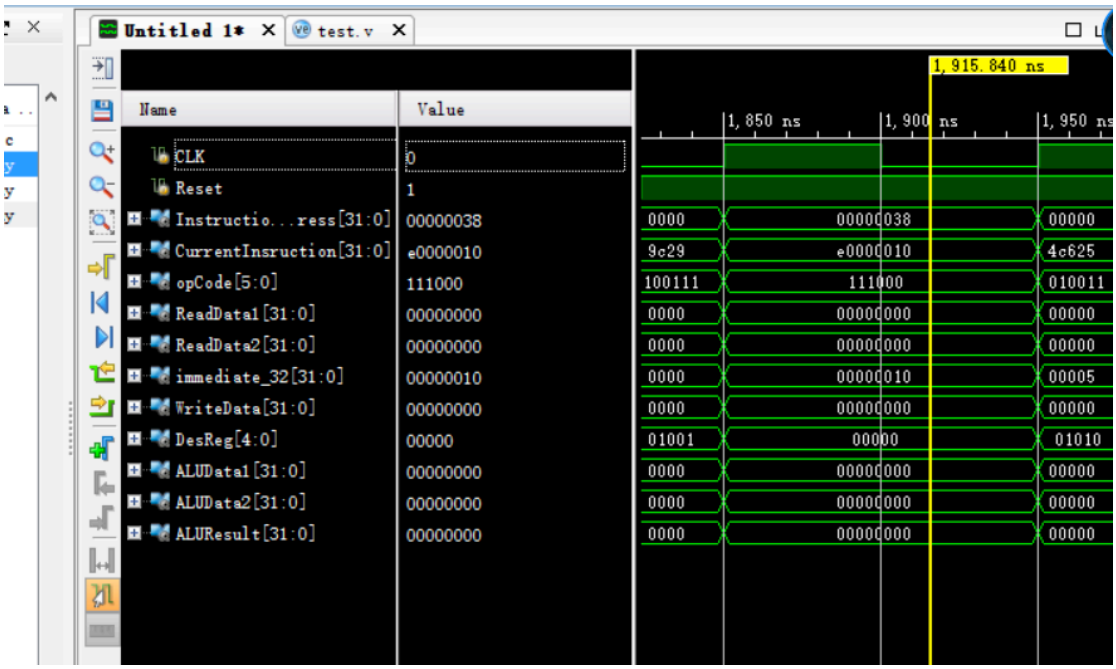
(13) lw $9, 4($1)：指令计数器经过上步操作后加4，则本条指令地址为0x00000034。该条指令将1号寄存器中数“0x00000008”与立即数

“0x00000004”进行相加，得到结果为0x0000000c，然后需要将第12个数据存储器中的数据读到9号寄存器中，同理，数据存储器的地址为0x0000000f。结果如图：



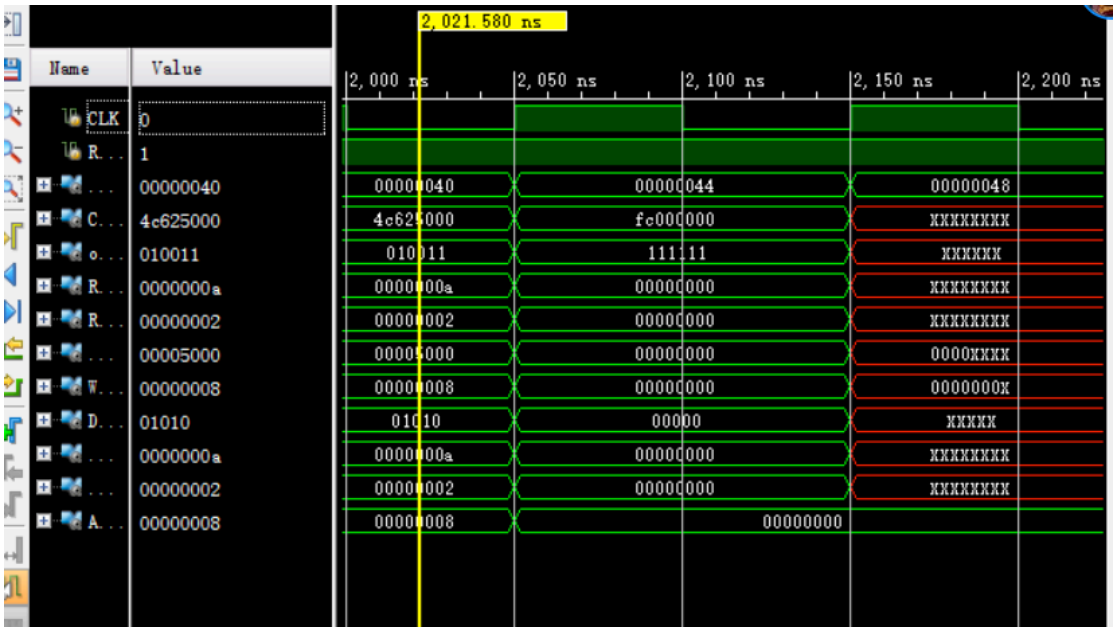
该条指令的结果为0x0000000c，相应的9号寄存器值为0x00000002，结果正确。

（15）j 0x00000040：指令计数器经过上步操作后加4，则本条指令地址为0x00000038。该条指令是PC直接切换到立即数所指的指令地址，即0x00000040，其指令为停机指令。结果如下图所示：



（16）addi $10，$0，10：指令计数器经过上步操作后加4，则本条指令地址为0x0000003c。该条指令因为上一条指令被跳过而直接执行它的下一条指令，所以没有数据发生改变。

（17）halt：指令计数器经过上步操作后加4，则本条指令地址为0x00000040。该条指令是由地址为0x00000038的指令无条件跳转过来的，结果如下图所示：



本指令将会使 CPU 进入停机状态，无论时钟信号怎么改变，各数据均不再发生改变，结果正确。

六. 实验心得

这次实验要求设计实现一个单周期 CPU，让我们对计算机组成原理这门课有了更深刻的理解。实验过程中，首先感喟老师和 TA 们精妙的设计思维，整个 CPU 基本的运行流程用一张图和几个表就表示清楚了，而且对计算机指令等基本介绍思路清晰，让我们这些入门者很受用。实验过程中收获很多，主要以下几方面。

语言层面：上学期学习数字电路的时候，我们就用到了 vivado 软件，但是对 Verilog 语言没有做过多要求，大多数实验是通过 vivado 的 block design 功能实现的。这次实验，我利用这个契机熟悉了 Verilog 语言的使用，理解了它的编程思想。Verilog 语言跟高级语言有很多不同，其特点主要体现在以下几方面：不使用初始化语句；不使用延时语句；不使用循环次数不确定的语句，如：forever，while 等；尽量采用同步方式设计电路；尽量采

用行为语句完成设计；always 过程块描述组合逻辑，应在敏感信号表中列出所有的输入信号；所有的内部寄存器都应该可以被复位；用户自定义原件（UDP 元件）是不能被综合的。

设计思路层面：自顶向下，从抽象到具体，模块化设计思路。模块化设计思路是 Verilog 语言的一大特点。一开始进行实验的时候，在巨大的项目面前无从着手，直到发现可以用模块化的设计思路庖丁解牛、化繁为简，CPU 工作的流程在心中开始明朗，Verilog 的编码也变得胸有成竹。

技术理解层面：通过这次实验，增强了我对单周期 CPU 运行流程的理解。正所谓“纸上得来终觉，绝知此事要躬行”，实验课将理论知识实例化，实现项目的过程本身就是一个纠错的过程。有些时候理论知识出现了错误，自己可能感觉不到，实验过程波形的错位可以察觉自己对理论知识掌握的错误，因此做这种大项目对于我们学生来说是很有好处的。学以致用才是根本!

心态层面：这次实验很复杂，系统很庞大，不可避免遇到困难和挑战。成功的道路并非一帆风顺，实验中出现了很多细节错误，导致整个系统都错掉了。这时候很沮丧，感觉“剪不断，理还乱”。不过通过自己耐心查找，分模块纠错——有点类似于“断点法 debug”，终于成功实现。可见，程序员是十分需要耐心的，这种碎片化知识的整合能力是通过一次次的纠错查错、与 bug 作斗争的过程中培养起来的，这种不骄不躁的品格对我们的人生发展是有好处的。