《计算机组成原理与接口技术实验》

实验报告

（实验三）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	16 软件工程三（5）班
学 生 姓 名 ：	邱奕浩
学　　号 ：	16340186
时　　间 ：	2018　年　6　月　22　日

成 绩 ：

实验三 ： 多周期CPU设计与实现

一、 实验目的

(1) 认识和掌握多周期数据通路原理及其设计方法；
(2) 掌握多周期 CPU 的实现方法，代码实现方法；
(3) 编写一个编译器，将 MIPS 汇编程序编译为二进制机器码；
(4) 掌握多周期 CPU 的测试方法;
(5) 掌握多周期 CPU 的实现方法。

二、 实验内容

设计一个多周期 CPU，该 CPU 至少能实现以下指令功能操作。需设计的指令与格式如下：（**说明：操作码按照以下规定使用，都给每类指令预留扩展空间，后续实验相同。**）

==>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

（2）sub rd, rs, rt

000001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

完成功能：rd<－rs － rt

（3）addi rt, rs, **immediate**

000010	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt<－rs + (sign-extend)**immediate**

==>逻辑运算指令

（4）or rd, rs, rt

010000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd<－rs | rt

（5）and rd, rs, rt

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

功能：rd<－rs & rt

（6）ori rt, rs, **immediate**

010010	rs(5 位)	rt(5 位)	**immediate**

功能：rt<－rs | (zero-extend)**immediate**

==>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） slt rd, rs, rt 带符号数

100110	rs(5 位)	rt(5 位)	rd(5 位)	reserved

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

（9） sltiu rt, rs,immediate 不带符号

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

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

==>存储器读写指令

（10）sw rt, **immediate**(rs)

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

功能：memory[rs+ (sign-extend)**immediate**]<－rt。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。

（11）lw rt, **immediate**(rs)

110001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt <－ memory[rs + (sign-extend)**immediate**]。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

（12) beq rs,rt, **immediate** (说明：**immediate 从 pc+4 开始和转移到的指令之间间隔条数**)

110100	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

（13）bltz rs,**immediate**

110110	rs(5 位)	00000	**immediate**

功能：if(rs<0) pc←pc + 4 + (sign-extend)**immediate** <<2 else pc ←pc + 4

==>跳转指令

（14）j addr

111000	addr[27:2]

功能：pc <－{(pc+4)[31:28],addr[27:2],2'b00}，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。

（15）jr rs

111001	rs(5 位)	未用	未用	reserved

功能：pc <－ rs，跳转。

==>调用子程序指令

（16）jal　addr

111010	addr[27:2]

功能：调用子程序，pc <－ {(pc+4)[31:28],addr[27:2],2'b00}；$31<－pc+4，返回地址设置；子程序返回，需用指令 jr $31。跳转地址的形成同 j addr 指令。

==>停机指令

（17）halt (停机指令)

111111	00000000000000000000000000(26 位)

不改变 pc 的值，pc 保持不变。

在本文档中，提供的相关内容对于设计可能不足或甚至有错误，希望同学们在设计过程中如发现有问题，请你们自行改正，进一步补充、完善。谢谢！

三、 实验原理

多周期 CPU 指的是将整个 CPU 的执行过程分成几个阶段，每个阶段用一个时钟去完成，然后开始下一条指令的执行，而每种指令执行时所用的时钟数不尽相同，这就是所谓的多周期 CPU。CPU 在处理指令时，一般需要经过以下几个阶段：

(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。

实验中就按照这五个阶段进行设计，这样一条指令的执行最长需要五个(小)时钟周期才能完成，但具体情况怎样？要根据该条指令的情况而定，有些指令不需要五个时钟周期的，这就是多周期的 CPU。

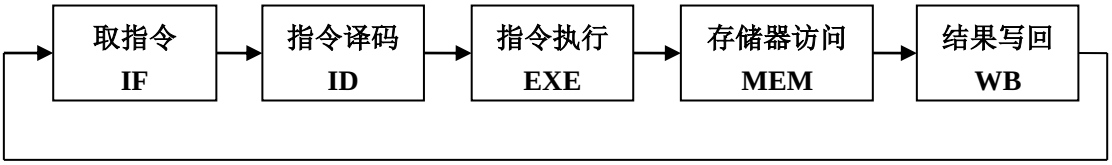
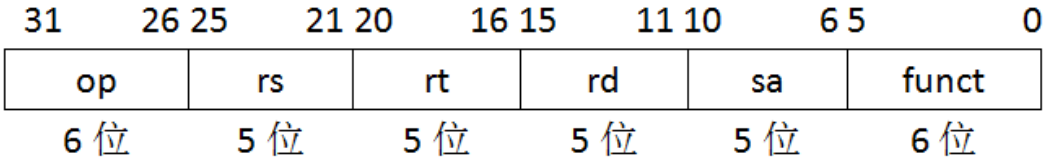


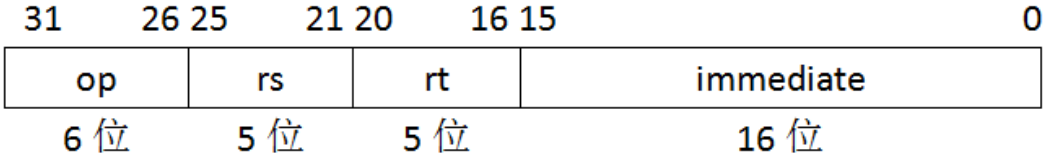
图 1　多周期 CPU 指令处理过程

MIPS 指令的三种格式:

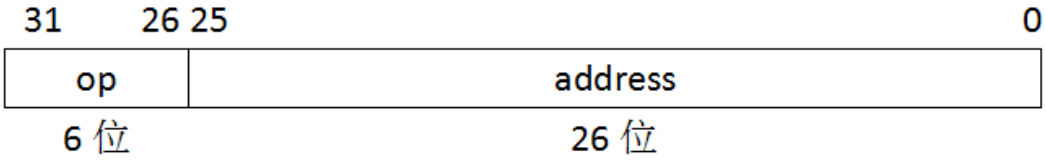
R 类型：



I 类型：



J 类型：



其中，

op：为操作码；

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

rt：为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd：为目的操作数寄存器，寄存器地址（同上）；

sa：为位移量（shift amt），移位指令用于指定移多少位；

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；

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。

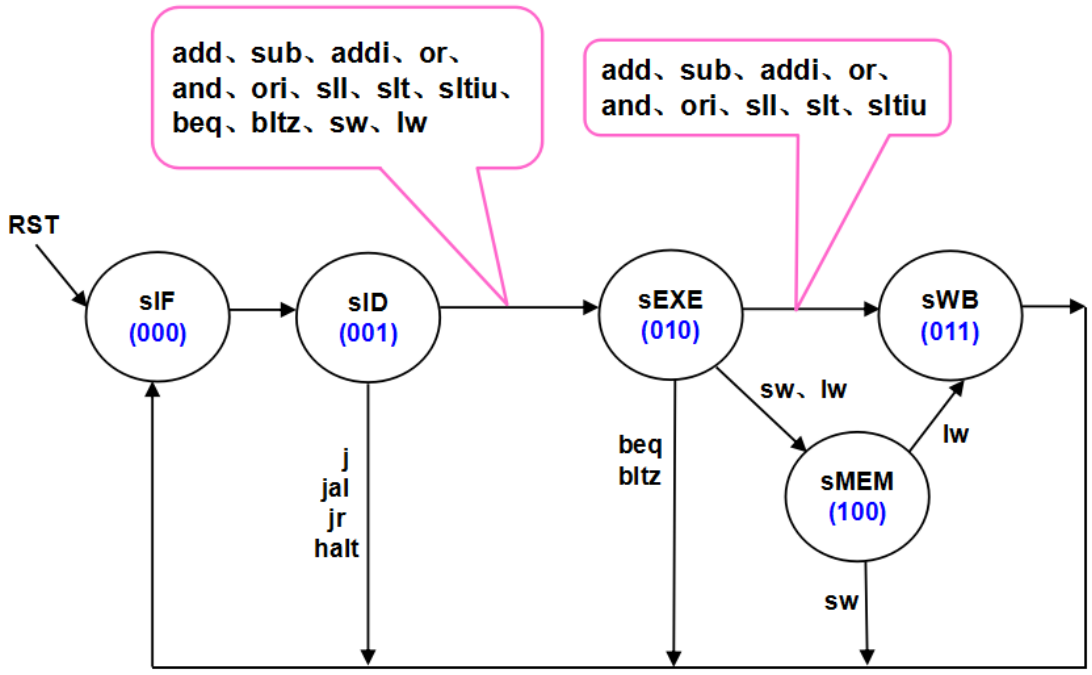


图 2 多周期 CPU 状态转移图

状态的转移有的是无条件的，例如从 sIF 状态转移到 sID 就是无条件的；有些是有条件的，例如 sEXE 状态之后不止一个状态，到底转向哪个状态由该指令功能，即指令操作码决定。每个状态代表一个时钟周期。

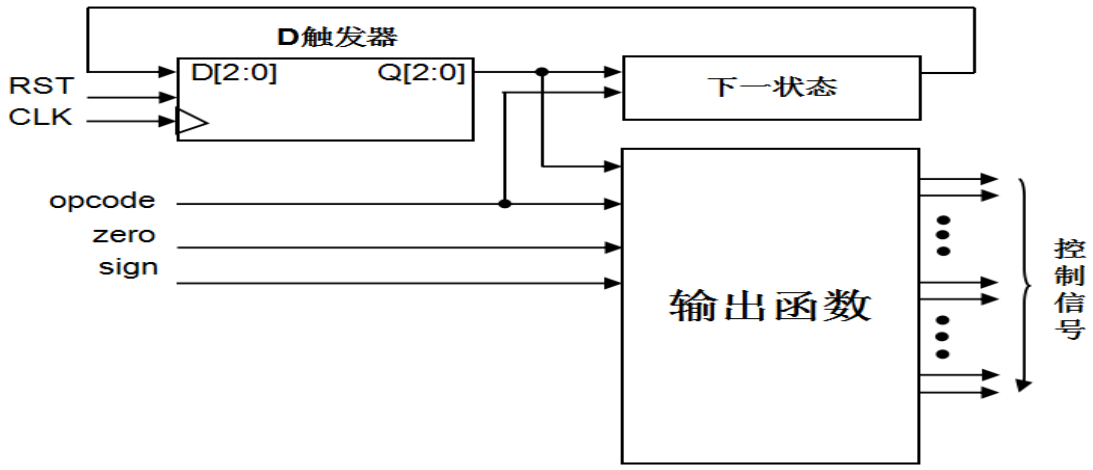


图 3 多周期 CPU 控制部件的原理结构图

图 3 是多周期 CPU 控制部件的电路结构，三个 D 触发器用于保存当前状态，是时序逻辑电路，RST 用于初始化状态“000“，另外两个部分都是组合逻辑电路，一个用于产生下一个阶段的状态，另一个用于产生每个阶段的控制信号。从图上可看出，下个状态取决于指令操作码和当前状态；而每个阶段的控制信号取决于指令操作码、当前状态和反映运算结果的状态 zero 标志和符号 sign 标志。

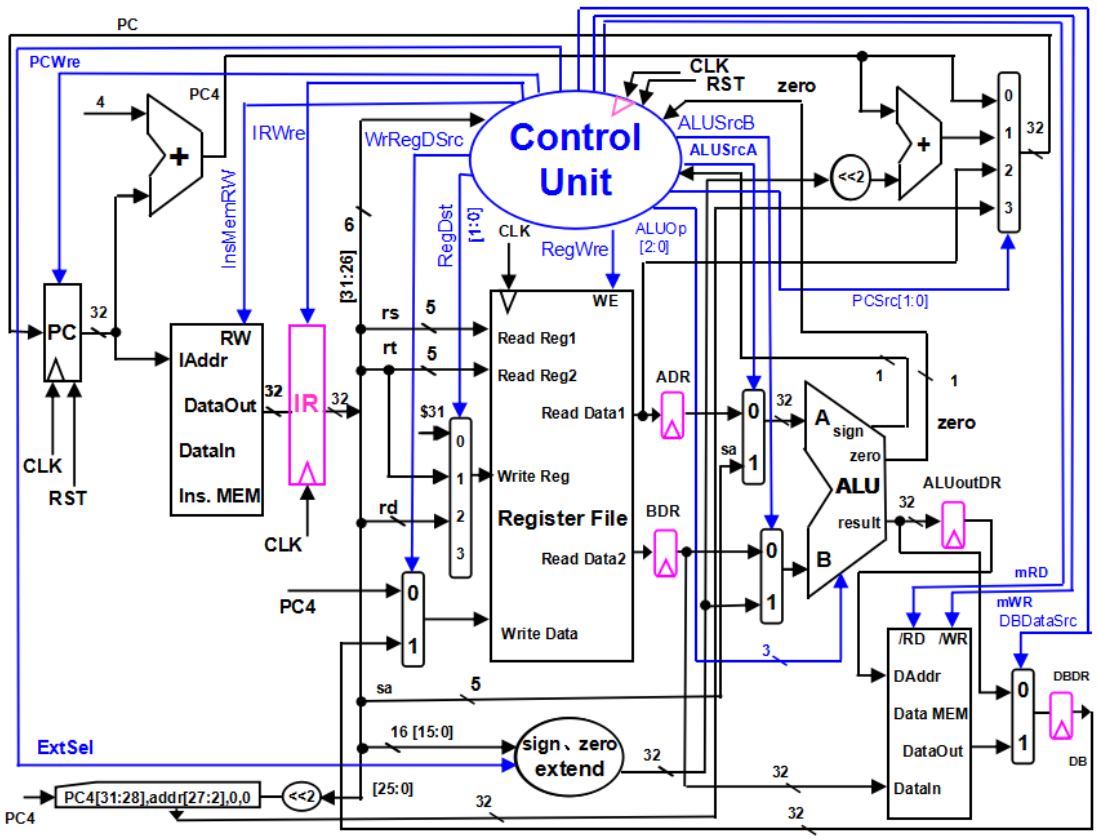


图 4　多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在多周期 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，给出寄存器地址（编号），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。图中控制信号功能如表 1 所示，表 2 是 ALU 运算功能表。

特别提示，图上增加 IR 指令寄存器，目的是使指令代码保持稳定，pc 写使能控制信号 PCWre，是确保 pc 适时修改，原因都是和多周期工作的 CPU 有关。ADR、BDR、ALUoutDR、DBDR 四个寄存器**不需要写使能信号，**其作用是切分数据通路，将大组合逻辑切分为若干个小组合逻辑，大延迟变为多个分段小延迟。

表 1　控制信号作用

控制信号名	状态“0”	状态“1”
RST	对于 PC，初始化 PC 为程序首地址	对于 PC，PC 接收下一条指令地址
PCWre	PC 不更改，相关指令：halt，另外，除‘000’状态之外，其余状态慎改 PC 的值。	PC 更改，相关指令：除指令 halt 外，另外，在‘000’状态时，修改 PC 的值合适。
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addi、or、and、ori、beq、bltz、slt、sltiu、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	来自 sign 或 zero 扩展的立即数，相关

	令：add、sub、or、and、beq、bltz、slt、sll	指令：addi、ori、sltiu、lw、sw
DBDataSrc	来自 ALU 运算结果的输出,相关指令：add、sub、addi、or、and、ori、slt、sltiu、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、bltz、j、sw、jr、halt	寄存器组寄存器写使能，相关指令：add、sub、addi、or、and、ori、slt、sltiu、sll、lw、jal
WrRegDSrc	写入寄存器组寄存器的数据来自 pc+4(pc4），相关指令：jal，写 $31	写入寄存器组寄存器的数据来自 ALU 运算结果或存储器读出的数据，相关指令：add、addi、sub、or、and、ori、slt、sltiu、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	存储器输出高阻态	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
IRWre	IR(指令寄存器)不更改	IR 寄存器写使能。向指令存储器发出读指令代码后，这个信号也接着发出，在时钟上升沿，IR 接收从指令存储器送来的指令代码。与每条指令都相关。
ExtSel	(zero-extend)**immediate**，相关指令：ori、sltiu；	(sign-extend)**immediate**，相关指令：addi、lw、sw、beq、bltz；
PCSrc[1:0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slt、sltiu、sll、sw、lw、beq(zero=0)、bltz(sign=0，或 zero=1)； 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bltz(sign=1，zero=0)； 10：pc<－rs，相关指令：jr； 11：pc<－{(pc+4)[31:28],addr[27:2],2'b00}，相关指令：j、jal；	
RegDst[1:0]	写寄存器组寄存器的地址，来自： 00：0x1F($31)，相关指令：jal，用于保存返回地址（$31<-pc+4）； 01：rt 字段，相关指令：addi、ori、sltiu、lw； 10：rd 字段，相关指令：add、sub、or、and、slt、sll； 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，指令地址输入端口

DataIn，存储器数据输入端口

DataOut，存储器数据输出端口

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，数据地址输入端口

DataIn，存储器数据输入端口

DataOut，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，其地址输入端口（rt、rd）

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

IR：　指令寄存器，用于存放正在执行的指令代码

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= (A<B) ?1: 0	比较 A 与 B 不带符号
011	Y=(((rega<regb) && (rega[31] == regb[31])) ||((rega[31] ==1 && regb[31] == 0))) ? 1:0	比较 A 与 B 带符号
100	Y = B<<A	B 左移 A 位
101	Y = A ∨ B	或
110	Y = A ∧ B	与
111	Y = A ⊕ B	异或

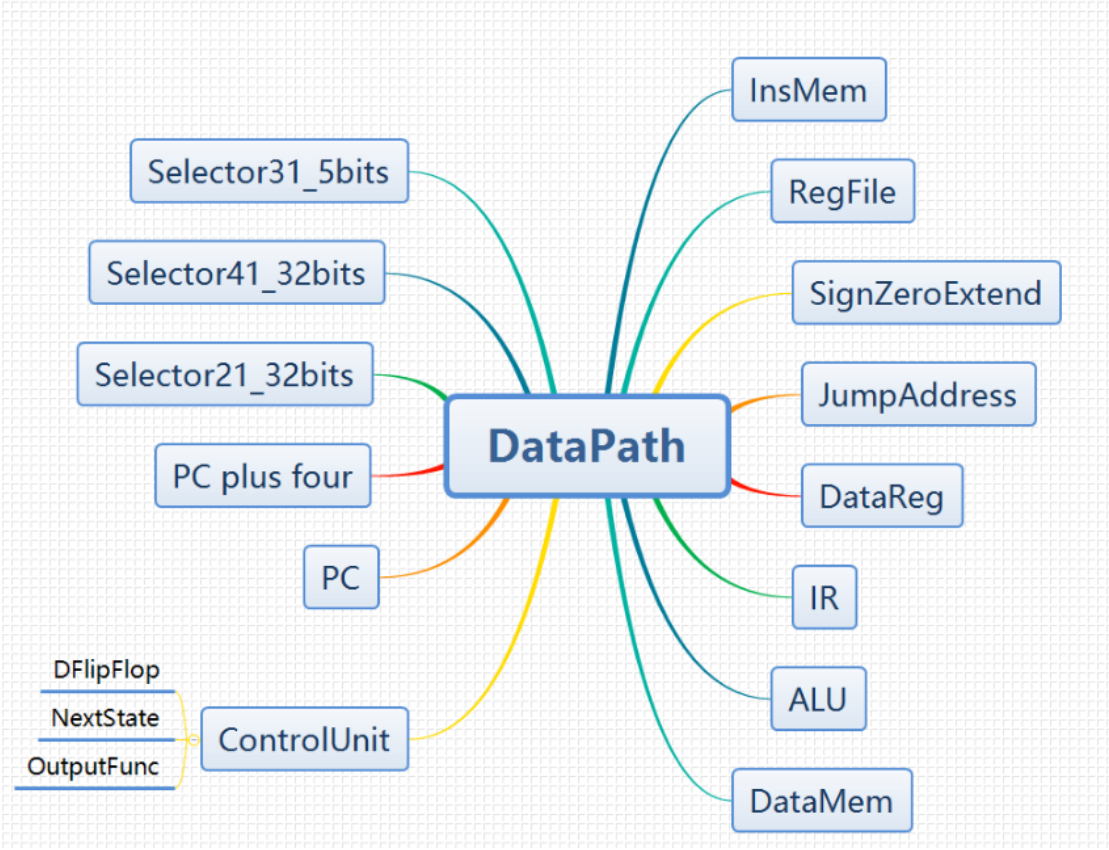
四、 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五、 实验过程与结果

● 多周期CPU设计和实验过程

本实验，根据模块划分和数据通路图，将代码划分为 20 个模块，其中包括一个顶层模块 DATa Path，14 个主模块，其中的 ControlUnit 主模块内部又有 3 个小模块。如下图。



每个模块根据需要又会实例化为多个模块来调用，以下说明根据代码说明各个模块的作用和功能。

_DataPath	DataPath	Verilog Module
_PC	PC	Verilog Module
_PCplus4	PCPlusFour	Verilog Module
_InsMem	InsMem	Verilog Module
_IR	IR	Verilog Module
controlunit	ControlUnit	Verilog Module
selector...	Selector31_5bits	Verilog Module
_RegFile	RegFile	Verilog Module
_SignZeroE...	SignZeroExtend	Verilog Module
_JumpAddre...	JumpAddressExtend	Verilog Module
Selector41...	Selector41_32bits	Verilog Module
ADR	DataReg	Verilog Module
BDR	DataReg	Verilog Module
selector32_A	Selector21_32bits	Verilog Module
selector32_B	Selector21_32bits	Verilog Module
_ALU	ALU	Verilog Module
ALUOutDR	DataReg	Verilog Module
_DataMem	DataMem	Verilog Module
ToDBDR	Selector21_32bits	Verilog Module
DBDR	DataReg	Verilog Module
ToRegFile	Selector21_32bits	Verilog Module

1. PC时钟模块：该模块用于接收下一条指令地址，输出当前指令地址，受控制信号PCWre，为1时继续执行，为0时停机。

```
module PC(
PCWre, NextAddress, CLK, RST, CurrentAddress
);

    input  PCWre;
    input wire [31:0]NextAddress;
    input  CLK;
    input  RST;
    output reg [31:0] CurrentAddress;

     //等待时钟上升沿
       always@(posedge CLK)begin
           if(RST == 0)begin  //初始化
               CurrentAddress = 0;
           end
           else if(!PCWre)begin  //停机
               CurrentAddress = CurrentAddress;
           end
           else if(PCWre)begin   //不停机，下一条指令地址成为当前指令地址
               CurrentAddress = NextAddress;
           end
       end
endmodule
```

2. PCPlus地址加4模块：相邻指令地址相差4，计算下一条指令地址，进入寻址模块。

```
module PCPlusFour(
CurrentAddress,PCPlusFourAddress
);
     input wire [31:0] CurrentAddress;
     output wire [31:0] PCPlusFourAddress;
     assign PCPlusFourAddress [31:0] = CurrentAddress[31:0] + 3'b100;
endmodule
```

3. InsMem指令存储器：指令存储器在初始化时存储指令代码，根据PC时钟的输出确定当前指令的地址，然后输出当前指令的代码。

```
module InsMem(
 InsMemRW,  CurrentAddress,  IDataOut
);
     input InsMemRW;
     input wire [31:0] CurrentAddress;
     output reg [31:0] IDataOut;
     reg [7:0] mem[127:0];  // 指令存储器单元的定义，128 个单元每个单元长度为 8
      initial begin
             {mem[0], mem[1], mem[2], mem[3]} =
32'b00001000000000010000000000001000;//此处加入指令代码
     end

     //当前指令地址进入，和读取信号
      always @( InsMemRW or CurrentAddress ) begin
          if (InsMemRW==1) begin // 为 1，读存储器。大端数据存储模式
              IDataOut[31:24] = mem[CurrentAddress];
              IDataOut[23:16] = mem[CurrentAddress+1];
              IDataOut[15:8] = mem[CurrentAddress+2];
              IDataOut[7:0] = mem[CurrentAddress+3];
          end
      end
endmodule
```

4. IR寄存器：受IRWre控制信号影响，在取指周期的下降沿触发（IRWre = 1），输出当前指令，以备下一步译码。

```
module IR(
    IRWre, IDataOut, CLK, IRDataOut
);
    input  IRWre;          // IR 寄存器写使能信号
    input [31:0] IDataOut;  // 接受指令存储器的指令值
    input CLK;             //时钟信号
    output reg [31:0] IRDataOut; // 指令寄存器的输出指令值

   always @(negedge CLK )begin
        if(IRWre == 1)
            IRDataOut = IDataOut;   //  更新指令寄存器的输出值
```

```
        else
            IRDataOut = IRDataOut;  // 保持现在输出值不变
    end
endmodule
```

5. ControlUnit控制单元：

   在不同的指令周期中，具有不同的状态，包括取指IF，译码ID，执行EXE，存储MEM，写回WB5个状态，根据指令操作码的差异，释放不同的控制信号。

   在其内部又细分为3个小模块：

   1) DFlipFlop D触发器模块。用于接收下一个周期的状态，输出当前周期的的状态。

```
module DFlipFlop(
RST, CLK, InputState, OutputState
);
    input RST, CLK;
    input [2:0] InputState;
    output reg[2:0] OutputState;
    initial begin
        OutputState = 3'b000;
    end

    always @(negedge CLK)begin
        if(RST == 0) OutputState = 3'b000;   // 重置为0
        else OutputState = InputState;  // 输出等于输入
    end

endmodule
```

   2) NextState 生成状态模块。结果当前状态和操作码，确定指令并产生下一阶段的状态。

```
module NextState(
InputState, opcode, OutputState
);
    input [5:0] opcode;
```

```
    output reg [2:0] OutputState;
    input [2:0] InputState;

   //     状态码
    parameter [2:0] StateIF = 3'b000;
    parameter [2:0] StateID = 3'b001 ;
    parameter [2:0] StateEXE = 3'b010;
    parameter [2:0] StateWB = 3'b011;
    parameter [2:0] StateMEM = 3'b100;

    always @ (InputState or opcode)begin
        case (InputState)
            StateIF: OutputState = StateID;  //取指之后直接译码状态
            StateID: begin
                case (opcode[5:3])
                    3'b111: OutputState = StateIF; //  跳转指令译码之后直接
跳转进入 IF
                    default: OutputState = StateEXE;    // 其他指令进入执行
阶段
                endcase
            end
            StateEXE:
                case(opcode[5:2])
                    4'b1101: OutputState = StateIF;   // beq bltz  to IF
                    4'b1100: OutputState = StateMEM;  // sw lw to MEM
                    default: OutputState = StateWB;   // to WB
                endcase
            StateMEM:
                    if(opcode == 6'b110000) OutputState = StateIF;     // sw
                    else OutputState = StateWB;     // lw
            StateWB: OutputState = StateIF;   // to IF
        endcase
    end

endmodule
```

3) OutputFunc 输出信号模块。根据zero sign CLK，当前状态以及操作码来确定正在执行的指令以及该指令正处于哪一个周期，由此产生相应的控制信号。事先定义好状态码和每条指令的操作码

```
//     状态码
      parameter [2:0] IF = 3'b000;
```

```
    parameter [2:0] ID = 3'b001 ;
    parameter [2:0] EXE = 3'b010;
    parameter [2:0] WB = 3'b011;

    parameter [2:0] MEM = 3'b100;
    // opcode
    parameter [5:0] _add = 6'b000000;
    parameter [5:0] _sub = 6'b000001;
    parameter [5:0] _addi = 6'b000010;
    parameter [5:0] _or = 6'b010000;
    parameter [5:0] _and = 6'b010001;
   // more…
```

这一模块是整个CPU的核心“大脑”，如何产生的控制信号是比较核心的一部分。以下注释每一个控制信号的确定方法。

```
        //PCWre,
        if(OutputState == IF && opcode != _halt) PCWre = 1;
        else PCWre = 0;  // 取指阶段除非遇到停机指令否则 为1

        //InsMemRW
        InsMemRW = 1;   //一直为1 保持取指状态

        //IRWre
        if(OutputState == IF) IRWre = 1;
        else IRWre = 0;   //除非取指状态，否则为0


         if(OutputState == EXE) begin
          //ALUSrcA
            if(opcode == _sll) ALUSrcA = 1;
            else ALUSrcA = 0;  //执行阶段 遇到sll 输入sa 移位码

                  //ALUSrcB
           if(opcode == _addi || opcode == _ori || opcode == _sltiu || opcode
== _lw || opcode == _sw)
            ALUSrcB = 1; //需要立即数的指令声明为1
            else ALUSrcB = 0;

          //ALUOp，根据opcode 和运算表确定运算操作码ALUOp
              case(opcode)
```

```
                    _add: ALUOp = 3'b000;
                    _sub: ALUOp = 3'b001;
                    _addi: ALUOp = 3'b000;

                    _or: ALUOp = 3'b101;
                    _and: ALUOp = 3'b110;
                    _ori: ALUOp = 3'b101;
                    _sll: ALUOp = 3'b100;
                    _slt: ALUOp = 3'b010;
                    _sltiu: ALUOp = 3'b010;
                    _sw: ALUOp = 3'b000;
                    _lw: ALUOp = 3'b000;
                    _beq: ALUOp = 3'b001;
                    _bltz: ALUOp = 3'b001;
                endcase
            end
            //译码阶段
            if(OutputState == ID)begin
           //ExtSel   需要0扩展的指令声明为0，需要符号为扩展时为1
                  if(opcode == _addi || opcode == _sll || opcode == _sltiu
|| opcode == _sw || opcode == _lw || opcode == _beq || opcode == _bltz)
                   ExtSel = 1;
                  else ExtSel = 0;
            end

            if(OutputState == IF)begin
                 //PCSrc
                    case(opcode)
                        _j: PCSrc = 2'b11;
                        _jal: PCSrc = 2'b11;
                        _jr: PCSrc = 2'b10;
                        _beq: begin
                              if(zero)
                                  PCSrc = 2'b01;
                              else PCSrc = 2'b00;
                        end
                        _bltz:begin
                             if(sign)
                                 PCSrc = 2'b01;
                             else PCSrc = 2'b00;
                        end
                        default: PCSrc = 2'b00;
                    endcase
            end
```

```
            //mWR  存储阶段，根据 opcode 确定读取或者存储存储器的内容

            if(OutputState == MEM)begin
                   if(opcode == _sw) mWR = 1;
                     else mWR = 0;
                     //mRD
                     if(opcode == _lw) mRD = 1;
                     else mRD = 0;
            end

            ////////////////////WB
            //DBDataSrc 筛选 lw 指令读取存储器的内容 和 ALU 运算结果
            if(OutputState == WB && opcode == _lw) DBDataSrc = 1'b1;
            else if(OutputState == WB) DBDataSrc = 1'b0;

           //WrRegDSrc 写回阶段确定写入寄存器的内容
           if(OutputState == ID && opcode == _jal) WrRegDSrc = 1'b0;
           else if(OutputState == WB) WrRegDSrc = 1'b1;


             //RegWre 写回阶段，寄存器组可写，其他阶段寄存器可读
           if(OutputState == WB || opcode == _jal) RegWre = 1'b1;
           else RegWre = 1'b0;   //写回的时候需要等于 1

             //RegDst ，写回阶段，确定写入的位置。根据操作码的不同来确定
            if(opcode == _jal && OutputState == ID) RegDst = 2'b00;  // jal
指令 需要 31 号寄存器

            if(OutputState == WB)begin
                 if(opcode == _lw || opcode == _addi || opcode == _ori || opcode
== _sltiu) RegDst = 2'b01;  // lw 指令 指定 rt 寄存器
                 else RegDst = 2'b10;              // add 等指定 rd 寄存器
            end

            //取值阶段防止写入读取存储器
            if(OutputState == IF)begin
                 RegWre = 0;
                 mRD = 0;
                 mWR = 0;
            end
```

在ControlUnit中连接3个模块。

```
// 触发器
    DFlipFlop dFlipFlop(RST, CLK, InputState, OutputState);
    // 下一状态
    NextState nextState(OutputState, opcode, InputState);
    // 输出函数
   OutputFunc outputFunc(
        opcode, zero, sign, OutputState,DBDataSrc, mWR, mRD,ExtSel, PCWre,
        IRWre, InsMemRW, WrRegDSrc, RegDst, RegWre, ALUOp, PCSrc, ALUSrcA,
ALUSrcB
    );
```

6. Selector31_5bits，5位3选1选择器：根据控制信号RegDst来确定写入寄存器组的位置。

```
module Selector31_5bits(
    rt, rd, RegDst, WriteReg
);
    input [4:0] rt;
    input [4:0] rd;
    input [1:0] RegDst;
    output reg[4:0] WriteReg;

    always @(rt or rd or RegDst)begin
        case(RegDst)
                2'b00: WriteReg = 5'b11111;   //31 号寄存器
                2'b01: WriteReg = rt;         // rt 寄存器
                2'b10: WriteReg = rd;         // rd 寄存器
                default: WriteReg = WriteReg;
        endcase
    end
endmodule
```

7. RegFile寄存器组：寄存器组根据控制信号RegWre来确定写或读功能。进行写操作时需要指定写入位置WriteReg，和写入的数据WriteData。进行读操作时，输出rs，rt寄存器的内容。

```
module RegFile(
RegWre, CLK,  rs,  rt, WriteReg ,
WriteData, ReadData1, ReadData2
);

    input  RegWre;
    input  CLK;
    input wire [4:0]rs;
    input wire [4:0]rt;
  //  input wire [1:0] RegDst;
  //  input wire [4:0] rd;
    input wire [4:0] WriteReg;
    input wire [31:0] WriteData;
    output wire [31:0] ReadData1;
    output wire [31:0] ReadData2;

    reg [31:0] regFile[0:31]; // 寄存器组值的定义，同样大端存储
    integer i;
    initial begin
        for( i = 0; i < 32; i = i+1)
           regFile[i] = 0;
    end
    //0 号寄存器的值为 0
    assign ReadData1 = (rs == 0 ? 0:regFile[rs]);  //寄存器输出值 1
    assign ReadData2 = (rt == 0 ? 0:regFile[rt]);  //寄存器输出值 2

         always @ (negedge CLK ) begin //

         if(RegWre == 1 && WriteReg != 0)begin //
            regFile[WriteReg] <= WriteData; //  写入寄存器
         end
    end

endmodule
```

8. SignZeroExtend 0符号位扩展：根据控制信号ExtSel确定，ExtSel = 0时进行0位扩展，ExtSel = 1进行符号位扩展。

```
module SignZeroExtend(
    ImmediateToExtend, ExtSel, ExtendedImmediate
```

```
);
    input wire[15:0] ImmediateToExtend;
    input ExtSel;
    output reg [31:0] ExtendedImmediate;

    initial begin
        ExtendedImmediate = 0;  //初始化
    end

    always @ (ExtSel or  ImmediateToExtend)begin
        if(ExtSel)begin  //符号位扩展
            ExtendedImmediate = {{16{ImmediateToExtend[15]}},
ImmediateToExtend[15:0]};
        end
        else begin     // 零位扩展
          ExtendedImmediate = {{16{1'b0}},ImmediateToExtend[15:0]};
        end
    end
endmodule
```

9. JumpAddressExtend 跳转地址扩展：针对跳转指令j 和jal 来对 指令中的address进行扩展。

```
module JumpAddressExtend(
    PCPlusFourAddress, AddressToExtend, ExtendedAddress
);
    input wire [31:0] PCPlusFourAddress;
    input wire [25:0] AddressToExtend;
    output wire [31:0] ExtendedAddress;

    assign ExtendedAddress = {PCPlusFourAddress[31:28],
AddressToExtend[25:0],{2{1'b0}}};

endmodule
```

10. Selector41_32bits 4选1寻址模块，根据控制信号PCSrc来确定下一条指令地址Addr。PCSrc = 0，Addr = PC+4； PCSrc = 01，Addr = PC+4 +(immediate << 2); PCSrc = 10, Addr = ReadData1 (rs); PCSrc = 11, Addr =

ExtendAddress。

```
module Selector41_32bits(
  PCPlusFourAddress, ExtendedImmediate, ReadData1, ExtendedAddress, PCSrc,
NextAddress
);
    input wire[31:0] PCPlusFourAddress;  // PC + 4  地址
    input wire[31:0] ExtendedImmediate;  // 立即数扩展
    input wire[31:0] ReadData1;          // 寄存器 rs 输出值
    input wire[31:0] ExtendedAddress;    //跳转指令扩展地址
    input [1:0] PCSrc;              // 控制信号
    output reg[31:0] NextAddress;        // 产生下一条指令地址

    always @( PCPlusFourAddress or ExtendedImmediate or ReadData1 or
ExtendedAddress or PCSrc)begin
            case (PCSrc)
                2'b00: NextAddress = PCPlusFourAddress;     // 顺序执行
                2'b01: NextAddress = PCPlusFourAddress + (ExtendedImmediate
<< 2);   // 分支跳转
                2'b10: NextAddress = ReadData1;  // 寄存器寻址
                2'b11: NextAddress = ExtendedAddress; // 跳转地址
                default:begin
                     NextAddress = NextAddress;  // 保留原地址不变
                end
            endcase
        end
endmodule
```

11. DataReg数据寄存器：将数据暂时存起来，等待下一个时钟周期输出数据。

```
module DataReg(
DataIn, DataOut, CLK
);
   input wire [31:0] DataIn;
   output reg [31:0] DataOut;
   input  CLK;

   always @(posedge CLK)begin
          DataOut = DataIn;
   end
```

```
endmodule
```

数据寄存器的作用是切分数据通路，使指令执行划分为多个周期。

划分译码和执行周期的数据寄存器ADR和BDR

划分执行和存储周期的数据寄存器ALUOutDR

划分存储（or执行）和写回周期的数据寄存器DBDR

```
DataReg ADR(ReadData1, ADROut, CLK);
DataReg BDR(ReadData2, BDROut, CLK);
DataReg ALUOutDR(ALUResult,ALUOutDROut,CLK);
DataReg DBDR(DBDRIn, DBDROut, CLK);
```

12. Selector21_32bits 32位二选一数据选择器：

```
module Selector21_32bits(
    Data1, Data2, Control, Result
);
    input wire [31:0] Data1;
    input wire [31:0] Data2;
    input  Control;
    output wire [31:0] Result;
    assign Result = (Control == 0) ? Data1: Data2;

endmodule
```

本实验需要用到4个32位二选一数据选择器

1) Selector32_A：控制信号为ALUSrcA，用于选择移位码和rs寄存器值。
2) Selector32_B：控制信号为ALUSrcB，用于选择rt寄存器值和立即数扩展。
3) ToDBDR：控制信号为DBDataSrc，用于选择ALU运算结果和数据存储器的输出
4) ToRegfile：控制信号为WrRegSrc，用于选择PC+4地址或者写回数据存入寄存器。

```
//32 位数据选择器，输入 ALU
     Selector21_32bits selector32_A(ADROut, sa, ALUSrcA, dataA);

     Selector21_32bits selector32_B(BDROut, ExtendResult, ALUSrcB,dataB);
   //二选一选择器
    Selector21_32bits ToDBDR(ALUResult, DataOut, DBDataSrc, DBDRIn);
    Selector21_32bits ToRegFile(PCPlusFourAddress, DBDROut, WrRegDSrc,
WriteData);
```

13. ALU运算模块：根据ALUOp信号和运算值A，B，执行相应的运算操作。

```
module ALU(
  DataA, DataB,  ALUOp,  ALUResult, zero,  sign
);
      input wire [31:0] DataA;
      input wire [31:0] DataB;
      input  [2:0] ALUOp;
      output reg [31:0] ALUResult;
      output  zero;
      output  sign;

      initial begin
         ALUResult = 0;
      end
      assign sign = ( ALUResult[31] == 0)?0:1; // 结果负数 为1
      assign zero = ( ALUResult == 0 )?1:0;   //    结果为0 为1

      always @( ALUOp or DataA or DataB ) begin
          case (ALUOp)  //根据操作码来判断执行的操作
              3'b000 : ALUResult = DataA + DataB;
              3'b001 : ALUResult = DataA - DataB;
              3'b010 : ALUResult = (DataA < DataB) ? 1 : 0;  //不带符号比较
              3'b011 : ALUResult = (((DataA<DataB) && (DataA[31] ==
DataB[31] )) ||( ( DataA[31] ==1 && DataB[31] == 0))) ? 1:0;    //带符号比
较
              3'b100 : ALUResult = DataB << DataA;
              3'b101 : ALUResult = DataA | DataB;
              3'b110 : ALUResult = DataA & DataB;
              3'b111: if(DataA[31] == DataB[31]) ALUResult = 0;
```

```
                    else ALUResult = 1;        //异或
            default : begin
                ALUResult = 0;                //$display (" no match");
            end
        endcase
    end
endmodule
```

14. DataMem数据存储器模块：根据控制信号mRD，mWR来确定读取或者写入存储器。当mWR = 1时，将寄存器rt的值按地址ALUResult写入存储器。当mRD = 1时，根据地址ALUResult从存储器中输出数据DataOut，将数据存入寄存器。

```
module DataMem(
 CLK, mWR,  mRD,  DataAddress,  DataIn, DataOut
);

    input mWR;
    input mRD;
    input CLK;
    input wire [31:0] DataAddress;
    input wire [31:0] DataIn;
    output reg [31:0] DataOut;
    reg [7:0] ram[0:63]; // 大端存储

     initial begin
        DataOut = 0;
     end
       // 必须使用下降沿触发，不然会发生数据冲突，数据地址是上一条指令的结果。
       always@( negedge CLK ) begin
           if( mWR == 1 ) begin   //写
               ram[DataAddress] <= DataIn[31:24];
               ram[DataAddress+1] <= DataIn[23:16];
               ram[DataAddress+2] <= DataIn[15:8];
               ram[DataAddress+3] <= DataIn[7:0];
           end
           else begin
            if(mRD == 1)begin    // 读
                   DataOut[31:24] = ram[DataAddress];
                   DataOut[23:16] = ram[DataAddress+1];
                   DataOut[15:8] = ram[DataAddress+2];
                   DataOut[7:0] = ram[DataAddress+3];
```

```
            end
            else if(mRD == 0)begin //高阻态
                DataOut[31:24] = 8'bz;
                DataOut[23:16] = 8'bz;
                DataOut[15:8] = 8'bz;
                DataOut[7:0] = 8'bz;
            end
        end
    end
endmodule
```

最终使用一个顶层模块DataPath将上述所有模块连接起来。

```
//PC 时钟运行
    PC _PC(PCWre, NextAddress, CLK, RST, CurrentAddress);
  //PC + 4 模块产生地址
   PCPlusFour _PCplus4(CurrentAddress, PCPlusFourAddress);
        // 指令存储器模块
    //  mRD = 1;
       InsMem _InsMem(InsMemRW, CurrentAddress, IDataOut);

   //指令寄存器 IR
   IR _IR(IRWre, IDataOut, CLK, IRDataOut);
   //控制单元
    ControlUnit controlunit(
   zero,  RST,  CLK, mRD,  mWR, DBDataSrc,
   ExtSel, PCWre, IRWre, InsMemRW, opcode, WrRegDSrc,
   RegDst,  RegWre,  ALUOp,  PCSrc, sign,  ALUSrcA, ALUSrcB
   );

   Selector31_5bits selectorWriteReg(rt, rd, RegDst, WriteReg);



    // 寄存器组模块化
    RegFile  _RegFile(RegWre, CLK, rs, rt, WriteReg, WriteData, ReadData1,
ReadData2);
   //0 符号扩展位
   SignZeroExtend _SignZeroExtend(immediate, ExtSel , ExtendResult);

   //产生跳转地址
   JumpAddressExtend
_JumpAddressExtend(PCPlusFourAddress,toExtendAddress,JumpAddress);
```

```
    // 四选一选择器确定下一条指令地址
     Selector41_32bits Selector41_32bits(PCPlusFourAddress, ExtendResult,
ReadData1, JumpAddress, PCSrc, NextAddress);
    // 数据寄存器
    wire [31:0] ADROut,BDROut,ALUOutDROut, DBDROut;
    DataReg ADR(ReadData1, ADROut, CLK);
    DataReg BDR(ReadData2, BDROut, CLK);
    //32 位数据选择器，输入 ALU
     Selector21_32bits selector32_A(ADROut, sa, ALUSrcA, dataA);
     Selector21_32bits selector32_B(BDROut, ExtendResult, ALUSrcB,dataB);
   // wire [31:0] ALUOutDROut;
    ALU _ALU(dataA, dataB,ALUOp, ALUResult, zero, sign);
    DataReg ALUOutDR(ALUResult,ALUOutDROut,CLK);
    //实例化数据存储器，命名一律用 _name 格式
    DataMem _DataMem(CLK, mWR, mRD, ALUOutDROut, BDROut, DataOut);


    wire [31:0] DBDRIn;
    //二选一选择器
    Selector21_32bits ToDBDR(ALUResult, DataOut, DBDataSrc, DBDRIn);
    DataReg DBDR(DBDRIn, DBDROut, CLK);
    Selector21_32bits ToRegFile(PCPlusFourAddress, DBDROut, WrRegDSrc,
WriteData);
```

- **实验测试**

编写一个测试文件TestCPU，通过实例化顶层模块DataPath以及产生时钟周期的方式来运行测试程序。

```
DataPath _DataPath(
            .CLK(_CLK),             // 时钟
            .RST(_RST),          // 复位信号

            .zero(_zero),           //零信号
```

```
        .sign(_sign),
        .CurrentAddress(_CurrentAddress),   // 当前指令地址
        .NextAddress(_NextAddress),     // 下一条指令地址
        .ALUOp(_ALUOp),           //alu 操作码
        .ReadData1(_ReadData1),       //寄存器 堆 1 输出来自 rs
        .ReadData2(_ReadData2),        // 寄存器 堆 2 输出来自 rt
        .dataA(_dataA),           //ALU 操作数 1
        .dataB(_dataB),           //ALU 操作数 2
        .ALUResult(_ALUResult),       //ALU 运算结果
        .IDataOut(_IDataOut),         //指令存储器输出指令
        .DataOut(_DataOut),          //数据存储器输出
        .WriteData(_WriteData),            //通过二选一写数据进入寄存器
        .rs(_rs),
        .rt(_rt),
        .rd(_rd),
        .sa(_sa),
        .opcode(_opcode),          // 控制单元指令单元操作码
        .PCSrc(_PCSrc),           // 确定产生下个地址的控制信号
        .PCPlusFourAddress(_PCPlusFourAddress),    //pc+4 地址
        .immediate(_immediate),        //立即数符号数
        .ExtendResult(_ExtendResult),       // 0 / 符号扩展位输出
        .toExtendAddress(_toExtendAddress),   //j 指令中待扩展地址
        .JumpAddress(_JumpAddress),       //扩展后跳转地址
        .WriteReg(_WriteReg),           //  写入寄存器的位置
        .IRDataOut(_IRDataOut)           // 指令寄存器的输出值
        );

          initial begin
                  _CLK = 0;
                  _RST = 0; //初始化输入

                  #50;    //50 ns 后开始 重置
                      _CLK = 1;
                  #50; //不再重置 pc 0
                    // _RST = 1;
                  forever #50 begin   //产生时钟信号
                      _RST = 1;
                      _CLK = !_CLK;
                  end
              end
```

通过以下的指令代码表来测试本实验的正确性。

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	000010	00000	00001	0000 0000 0000 1000	=	0x08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	=	0x48020002
0x00000008	or $3,**$2**,$1	010000	00010	00001	0001 1000 0000 0000	=	0x40411800
0x0000000C	sub $4,**$3**,$1	000001	00011	00001	0010 0000 0000 0000	=	0x04612000
0x00000010	and $5,**$4**,$2	010001	00100	00010	0010 1000 0000 0000	=	0x44822800
0x00000014	**sll $5,$5,2**	011000	00000	00101	0010 1000 1000 0000	=	0x60052880
0x00000018	**beq $5,$1,-2(=,转 14)**	110100	00101	00001	1111 1111 1111 1110	=	0xD0A1FFFE
0x0000001C	**jal 0x0000040**	111010	00000	00000	0000 0000 0001 0000	= =	0xE8000010
0x00000020	**slt $8,$12,$1**	100110	01100	00001	0100 0000 0000 0000	=	0x99814000
0x00000024	**addi $13,$0,-2**	000010	00000	01101	1111 1111 1111 1110	=	0x080DFFFE
0x00000028	**slt $9,$8,$13**	100110	01000	01101	0100 1000 0000 0000	=	0x990D4800
0x0000002C	**sltiu $10,$9,2**	100111	01001	01010	0000 0000 000 00010	=	0x9D2A0002
0x00000030	**sltiu $11,$10,0**	100111	01010	01011	0000 0000 0000 0000	=	0x9D4B0000
0x00000034	addi $13,$13,1	000010	01101	01101	0000 0000 0000 0001	=	0x09AD0001
0x00000038	**bltz $13,-2 (<0, 转 34)**	110110	01101	00000	1111 1111 1111 1110	=	0xD9A0FFFE
0x0000003C	**j 0x000004C**	111000	00000	00000	0000 0000 0001 0011	=	0xE0000013
0x00000040	sw $2,4($1)	110000	00001	00010	0000 0000 0000 0100	=	0xC0220004
0x00000044	lw $12,4($1)	110001	00001	01100	0000 0000 0000 0100	=	0xC42C0004
0x00000048	**jr $31**	111001	11111	00000	0000 0000 0000 0000	=	0xE7E00000
0x000000C	**halt**	111111	00000	00000	0000 0000 0000 0000	=	0xFC000000

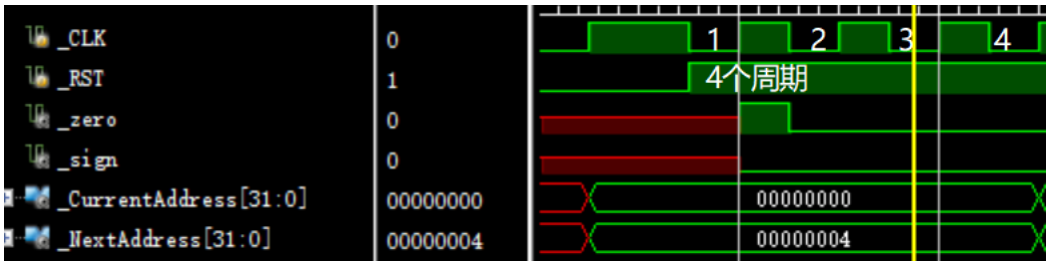
● 实验结果

以下分别测试每一条指令。

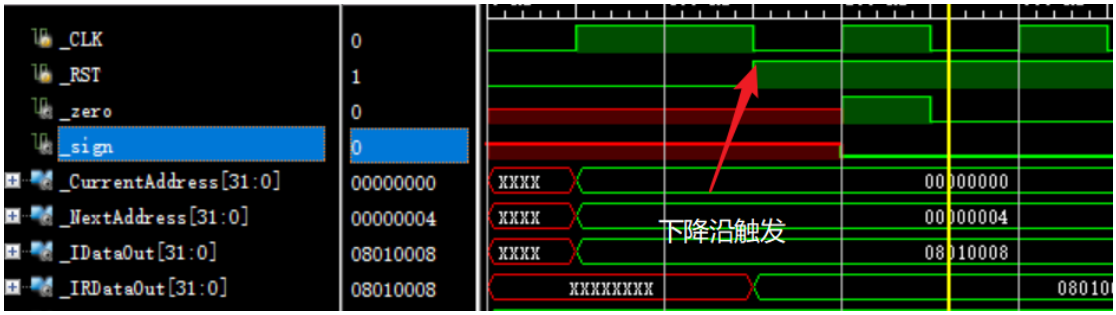
地址	汇编程序	指令代码					
		op（6）	**rs(5)**	**rt(5)**	**rd(5)/immediate(16)**	**16 进制数代码**	
0x00000000	addi $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	0x08010008

该指令经过取指，译码，执行，写回四个阶段，需要四个周期。

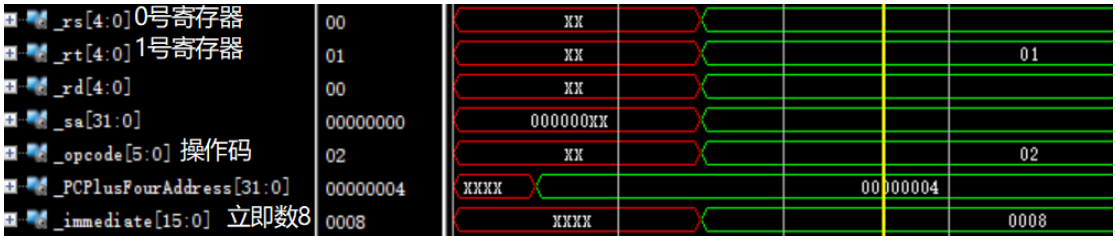
当前地址为0000000，下一条指令地址为00000004。



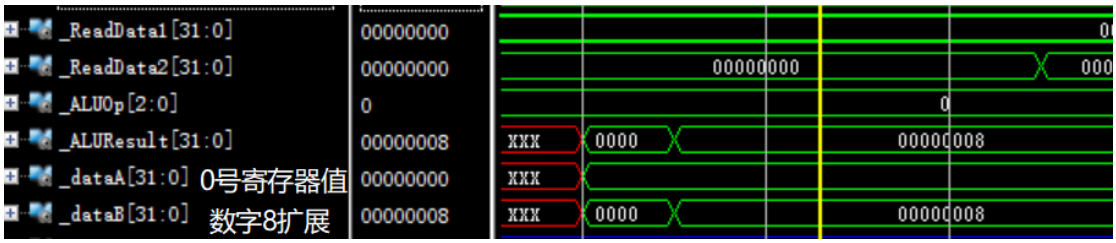
取指阶段：从指令存储器中取出数据，在下降沿触发的时候将指令传给指令寄存器。



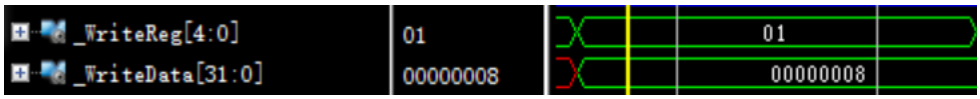
译码阶段：译码得到0号1号寄存器，以及立即数8。



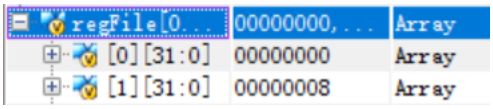
执行阶段：对0号寄存器值和数字8，进行加法操作，得到结果8



写回阶段：写入寄存器为1号，写入数据为0000008

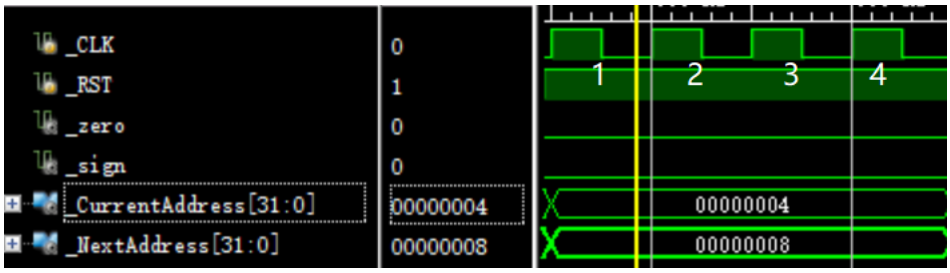


结果写入1号寄存器

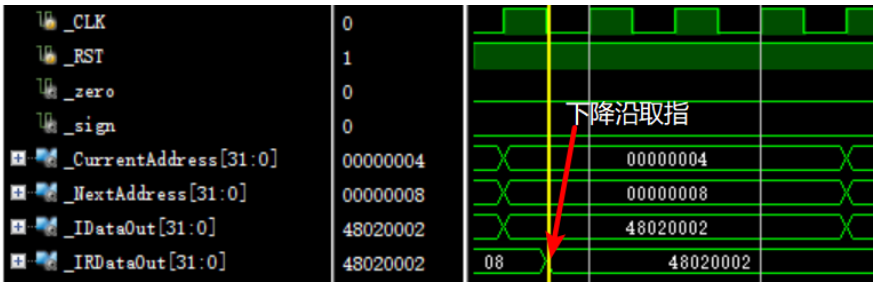


地址	汇编程序	指令代码				
		op（6）	rs(5)	rt(5)	rd(5)/immediate(16)	16 进制数代码
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	= 0x48020002

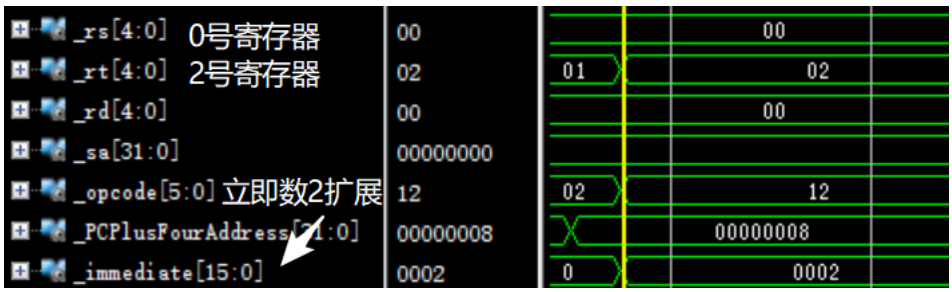
该指令经过取指，译码，执行，写回四个阶段，需要四个周期。



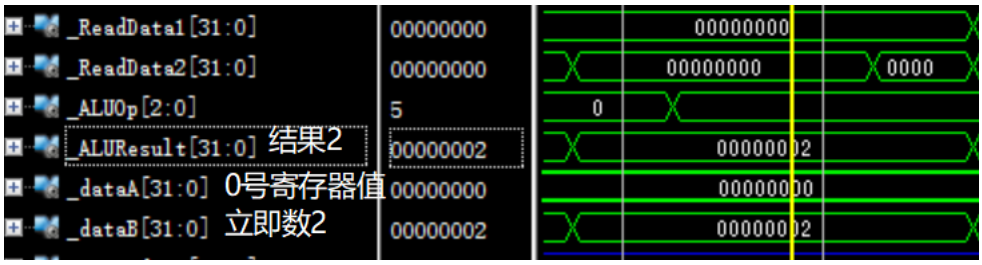
取指阶段：在下降沿读取指令4802002



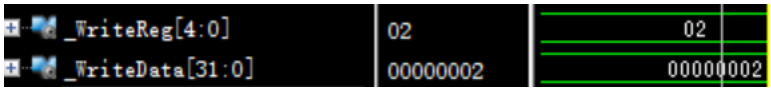
译码阶段：解析得到0号寄存器，2号寄存器以及立即数2。



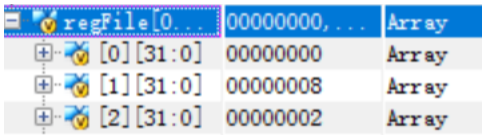
执行阶段：0号寄存器值为0，和立即数2进行或操作，运算结果为00000002。



写回阶段：将运算结果00000002写回2号寄存器。

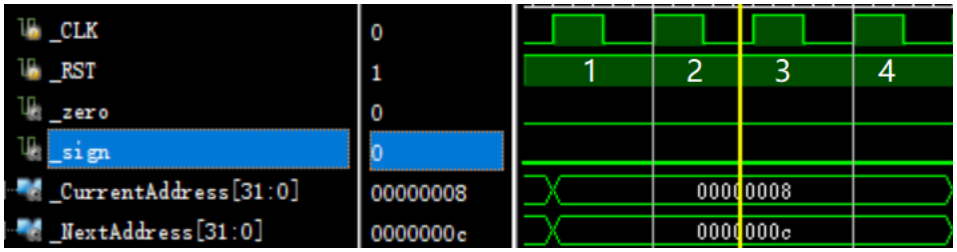


寄存器组内容，可以看到已经写入成功。



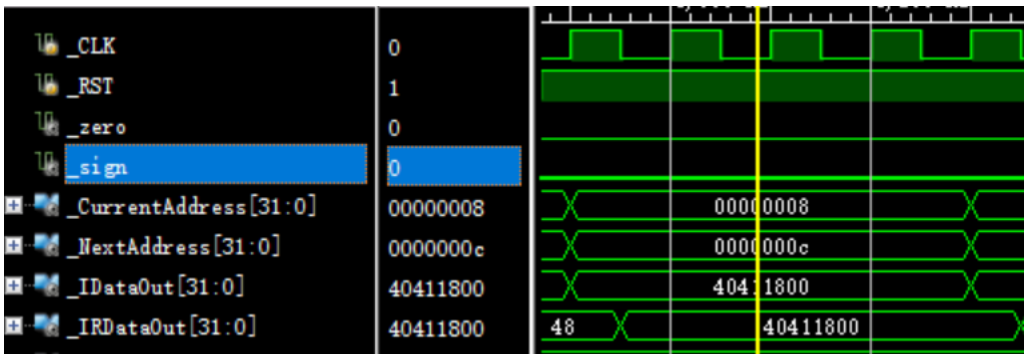
地址	汇编程序	指令代码				
		op（6）	rs(5)	rt(5)	rd(5)/immediate(16)	16 进制数代码
0x00000008	or $3,**$2**,$1	010000	00010	00001	0001 1000 0000 0000	= 0x40411800

该指令经过取指，译码，执行，写回四个阶段，需要四个周期。

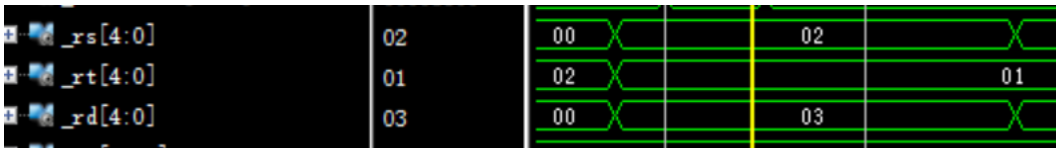


当前指令地址0000008，下一条指令000000c。

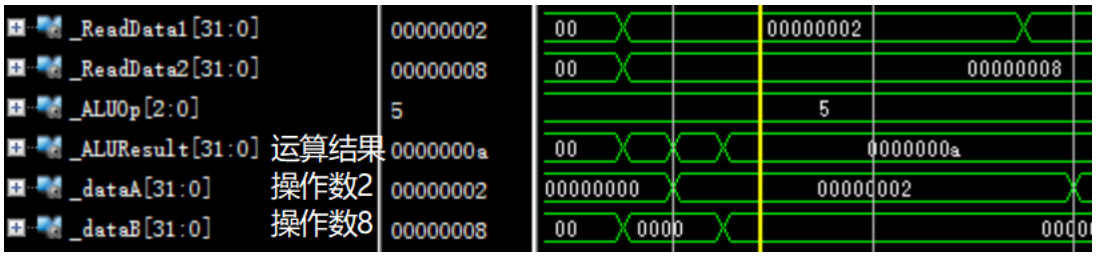
取指阶段：当前指令为40411800。下降沿触发，指令寄存器输出指令。



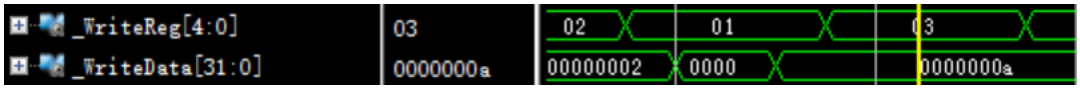
译码阶段：得到寄存器2，1，3。



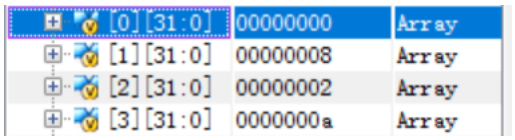
执行阶段：操作数00000002和00000008执行或运算，得到000000a



写回阶段：将数据00000a写入3号寄存器。



3号寄存器已写入。

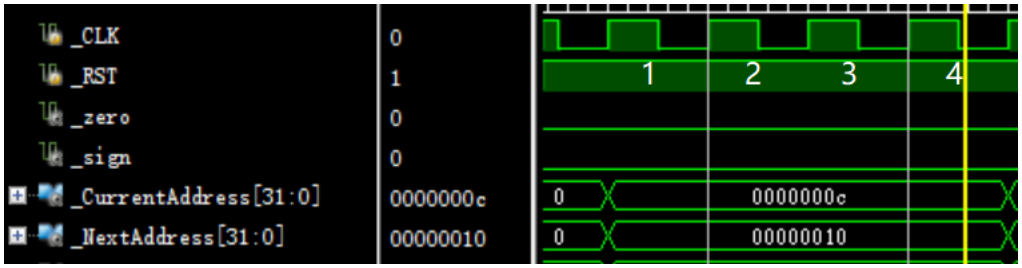


地址	汇编程序	指令代码

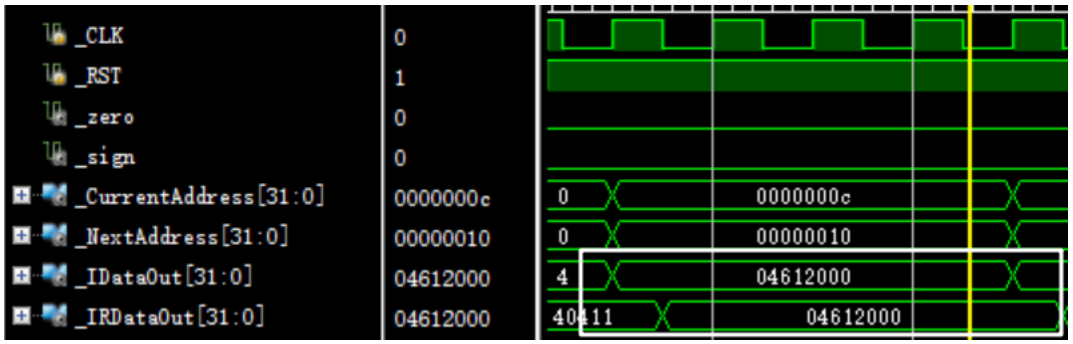
		op（6）	rs(5)	rt(5)	rd(5)/immediate(16)	16 进制数代码	
0x0000000C	sub $4,**$3**,$1	000001	00011	00001	0010 0000 0000 0000	=	0x04612000

该指令经过取指，译码，执行，写回四个阶段，需要四个周期。

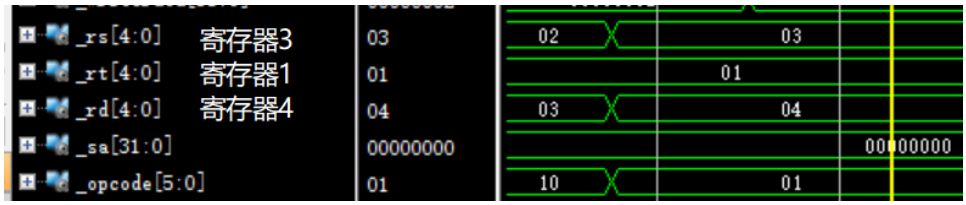
当前指令地址为0000000c，下一条指令地址为00000010



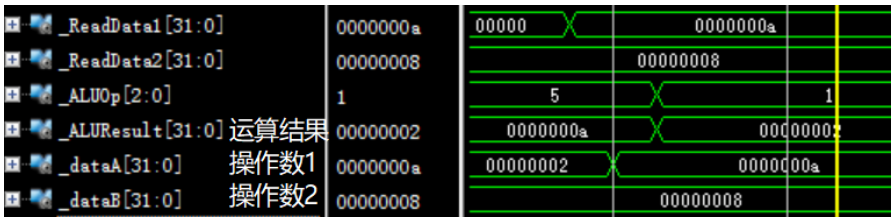
取值阶段：在第一个周期的下降沿触发，指令寄存器输出指令值。



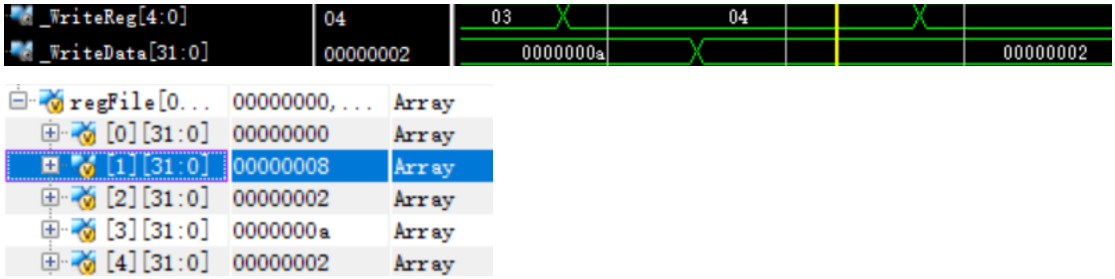
译码阶段：获取寄存器1，3， 4，操作码为000001



执行阶段：获取1号和3号寄存器的值，进行减法运算，得到结果2。



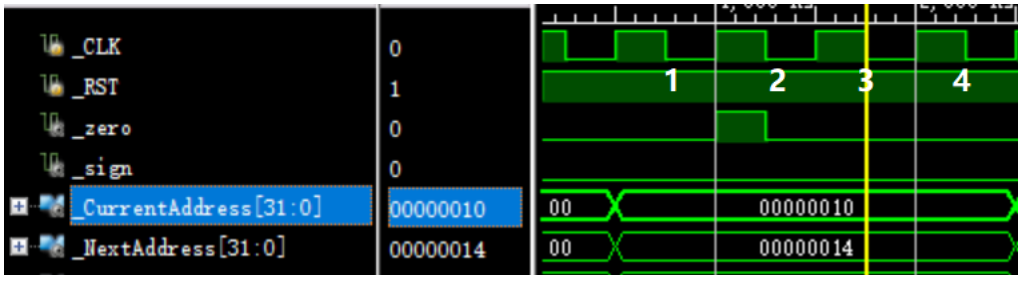
写回阶段：将结果2写回 4号寄存器。



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
0x00000010	and $5,**$4**,$2	010001	00100	00010	0010 1000 0000 0000	= 0x44822800

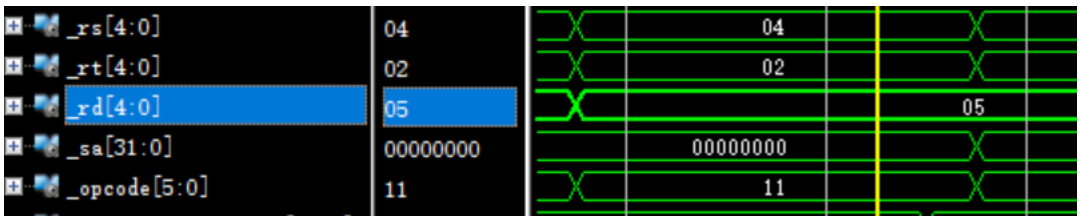
该指令经过取指，译码，执行，写回四个阶段，需要四个周期。

当前指令地址为00000010，下一条指令地址为0000014

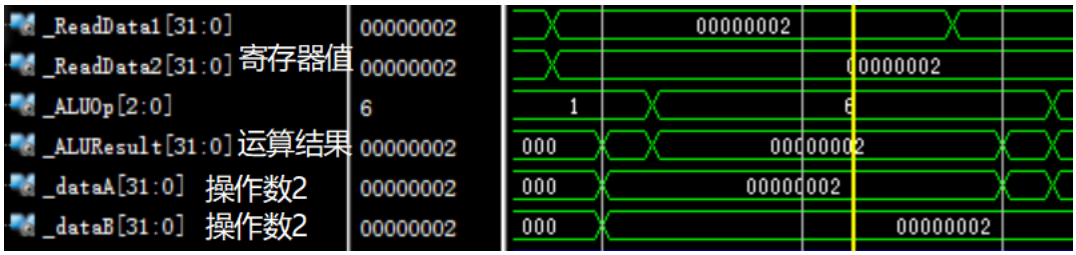


取指阶段：下降沿触发，指令寄存器输出44822800.

译码阶段：得到寄存器5，4，2，操作码为010001。



执行阶段：寄存器2，4的值都为2，作为操作数进行与运算得到运算结果0000002.

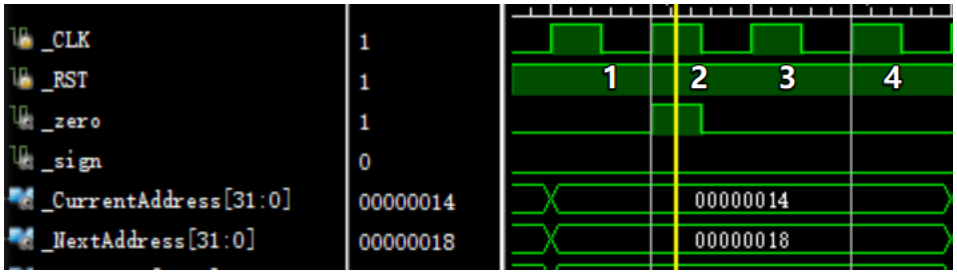


写回阶段：将运算结果0000002写入寄存器5号。

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
0x00000014	sll $5,$5,2	011000	00000	00101	0010 1000 1000 0000	= 0x60052880

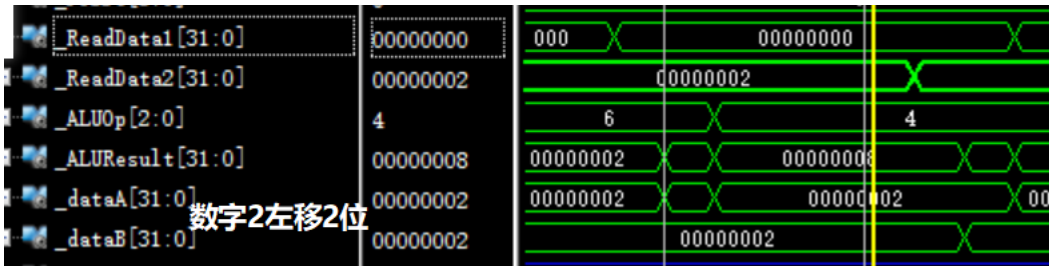
该指令经过取指，译码，执行，写回四个阶段，需要四个周期。

当前指令为00000014，下一条指令地址为00000018。



取指阶段：下降沿，指令寄存器输出指令60052880。

执行阶段：5号寄存器内容为0000002，左移2位，得到结果00000008.

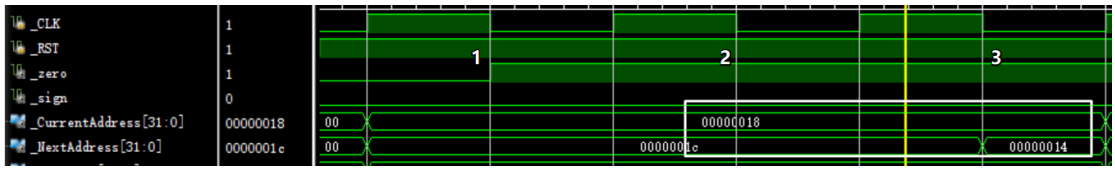


写回阶段：将000008写入5寄存器。

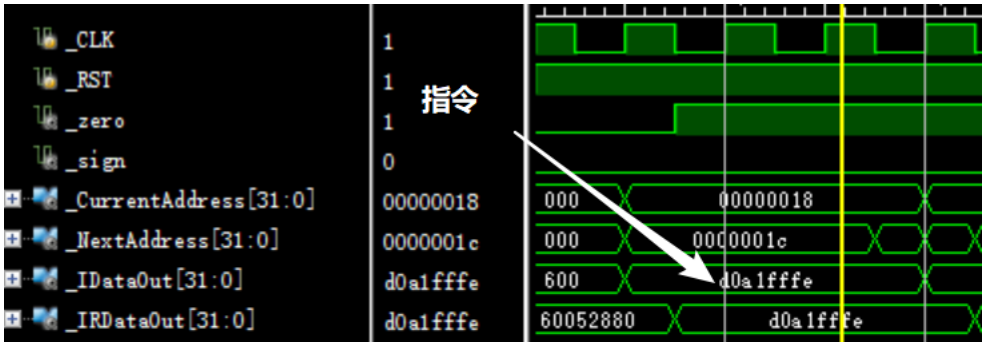
地址	汇编程序	指令代码					
		op（6）	rs(5)	rt(5)	rd(5)/immediate(16)	16 进制数代码	
0x00000018	**beq $5,$1,-2(=,转 14)**	**110100**	**00101**	**00001**	**1111 1111 1111 1110**	=	0xD0A1FFFE

该指令经过取指，译码，执行三个阶段，需要三个周期。

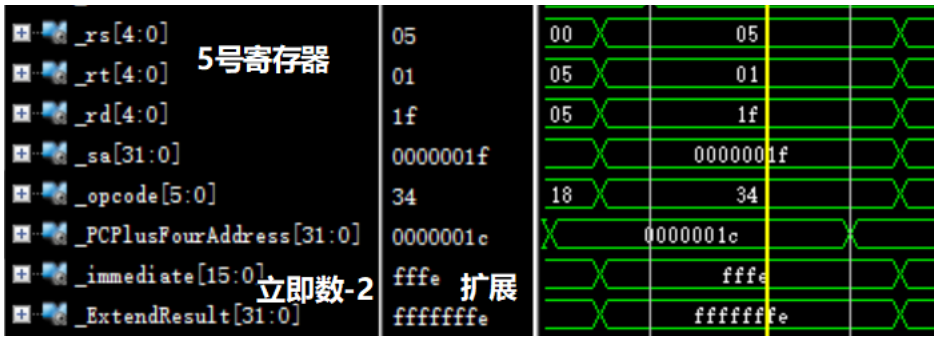
当前地址为00000018，下一条指令在执行阶段得到 $5 = $1，此时zero为0，可以看到在第三个周期，地址变为00000014，即重复上一条指令。



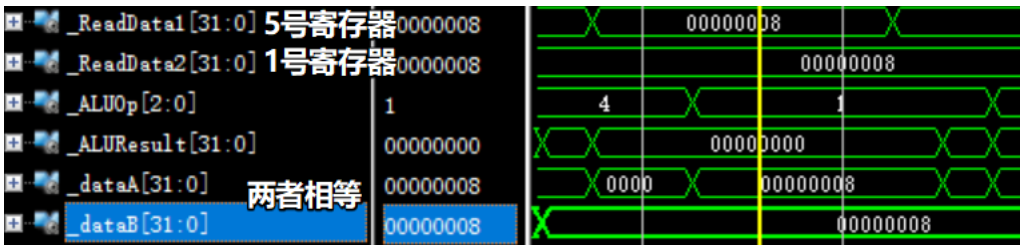
取指阶段：在取指周期，下降沿触发，指令寄存器输出指令d0a1fffe。



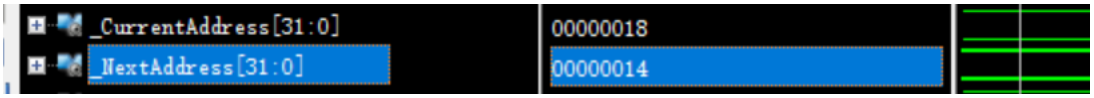
译码阶段：得到5号寄存器和立即数-2，fffe，可以看出立即数扩展的结果是ffffffe。



执行阶段：5号寄存器和1号寄存器内容相等，运算结果为0，产生zero = 1的信号，进行地址跳转。

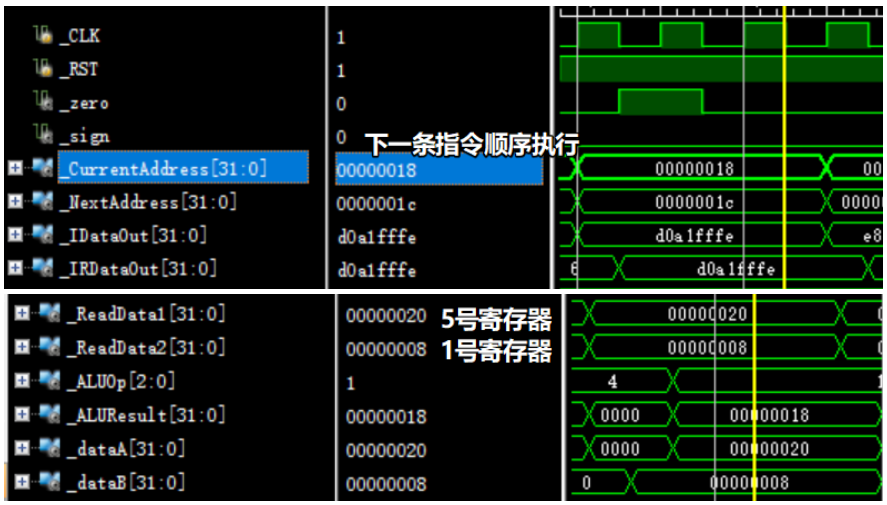


跳转地址的形成是：PC+4地址00000018，和立即数扩展-2（ffffffe）相加得到结果00000014，即上一条指令地址。



重复执行上一条指令，5号寄存器值左移两位，得到运算结果00000020.

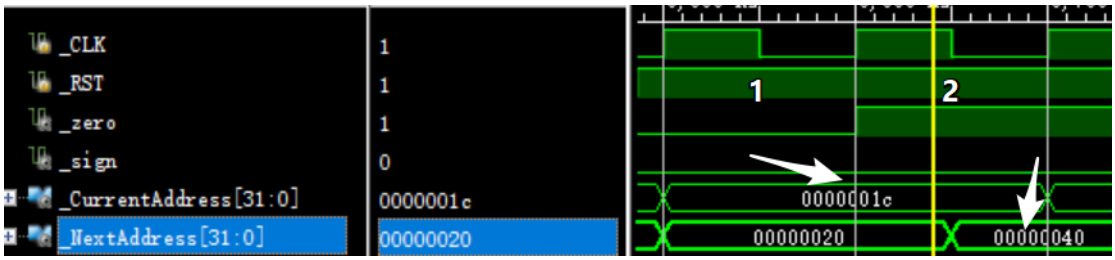
继续进行beq指令。5号和1号寄存器值不同，顺序执行。



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	
0x0000001C	jal 0x0000040	111010	00000	00000	0000 0000 0001 0000	=	0xE8000010

该指令经过取值，译码两个阶段，需要两个周期。

当前地址为0000001c，下一条指令地址为00000040。



PC+4地址写入31号寄存器。

地址扩展得到下一条指令地址为00000040。

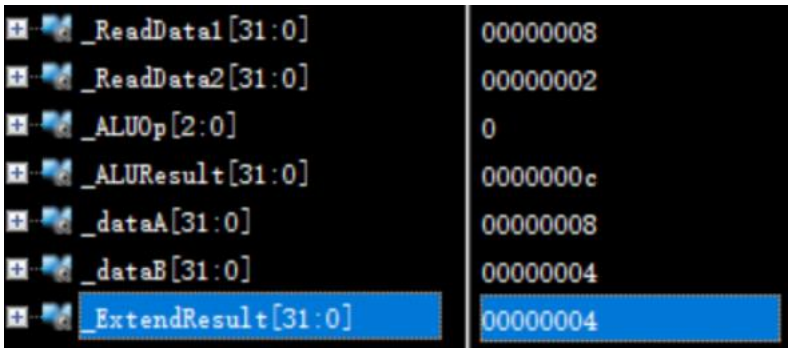
执行指令 sw $2 4($1)，地址为00000040，指令顺序执行。

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
0x00000040	sw $2,4($1)	110000	00001	00010	0000 0000 0000 0100	= 0xC0220004

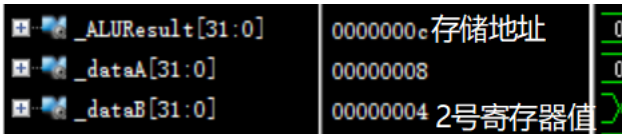
当前指令为00000040，下一条指令地址为00000044。

译码阶段：得到1号，2号寄存器，和立即数4

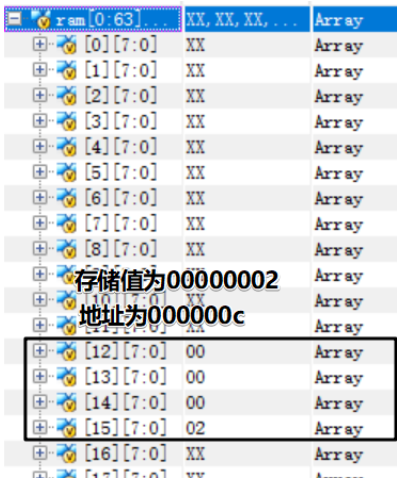
执行阶段：可以得到1号寄存器内容为000008，2号寄存器内容为00000002。立即数扩展为00000004。通过ALU加法运算得到结果000000c，该结果用于定位1号寄存器值在数据存储器中的位置。



存储阶段：将ALU运算结果存入ALUoutDR寄存器，然后进入存储器的DataAddress，同时2号寄存器的值为2，需要进入存储器的DataIn接口。



进入存储器DataMem中，可以看到存储数组，从0000000C开始，有数据00000002存储。

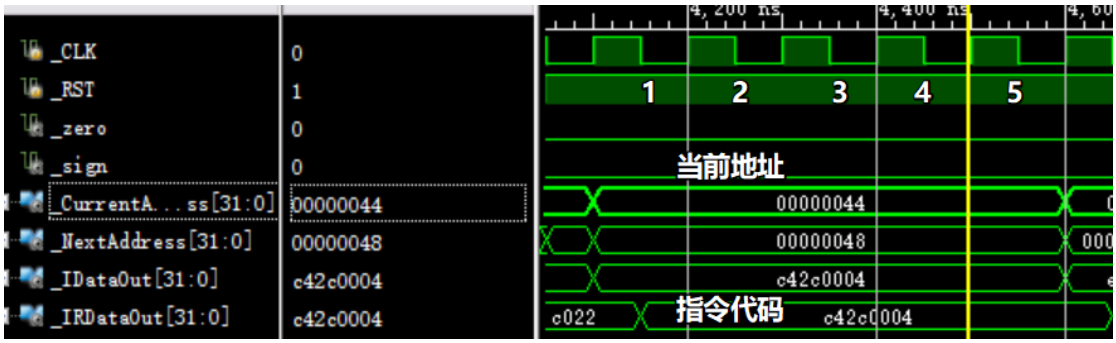


然后继续执行lw指令。

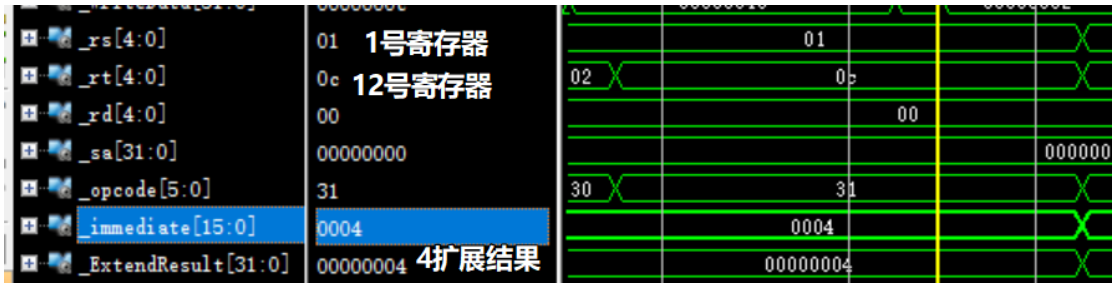
地址	汇编程序	指令代码				
		op（6）	rs(5)	rt(5)	rd(5)/immediate(16)	16 进制数代码
0x00000044	lw $12,4($1)	110001	00001	01100	0000 0000 0000 0100	= 0xC42C0004

可以看到该指令经过5个周期，取指，译码，执行，存储，写回操作。

当前地址为00000044，下一条指令为00000048.在取指周期的下降沿，指令寄存器输出指令代码c42c004.

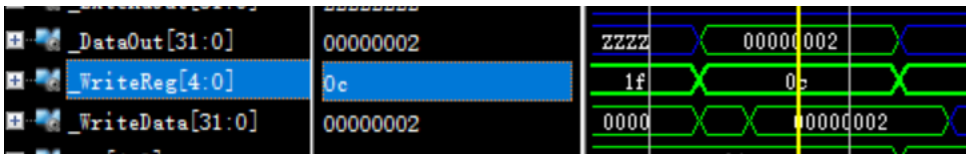


译码阶段：译码得到1号寄存器和12号寄存器，然后对立即数4，做0位扩展。



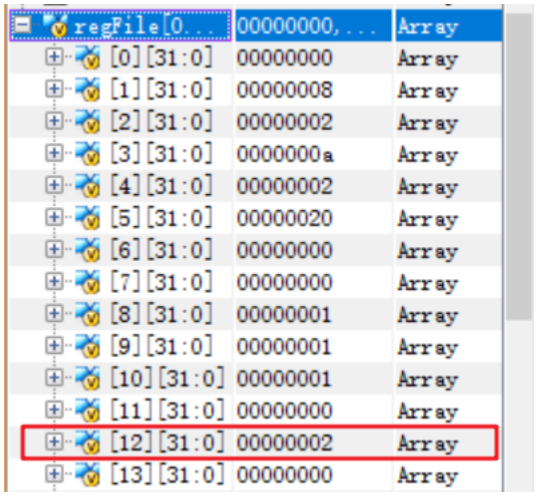
执行阶段：对4号寄存器值和立即数4进行加法操作，得到存储地址0000000c。

由上一条指令可知，存储器中地址位0000000c的值为00000002，当存储阶段开始时，存储器输出Dataout = 00000002。



写回阶段：将数据 00000002写回寄存器12号。

进入寄存器组RegFile，可以看到12号寄存器值为00000002.

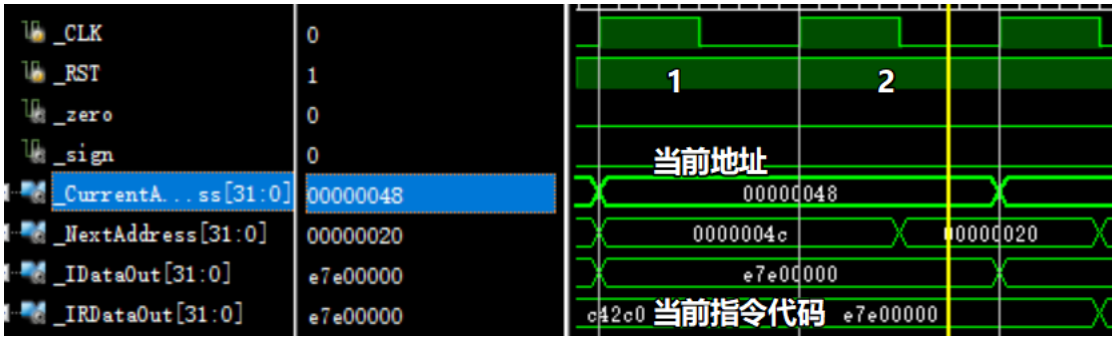


然后继续顺序执行指令jr。

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	
0x00000048	jr $31	111001	11111	00000	0000 0000 0000 0000	=	0xE7E00000

当前指令需要取指，译码两个阶段，需要两个周期的时间来完成。

从下图可知，当前地址为000000048，下一条指令地址为00000020。为31号寄存器的值。从上面的指令jal可知31号寄存器的值为00000020。当指令执行结束之后，跳转回到地址为00000020的指令。

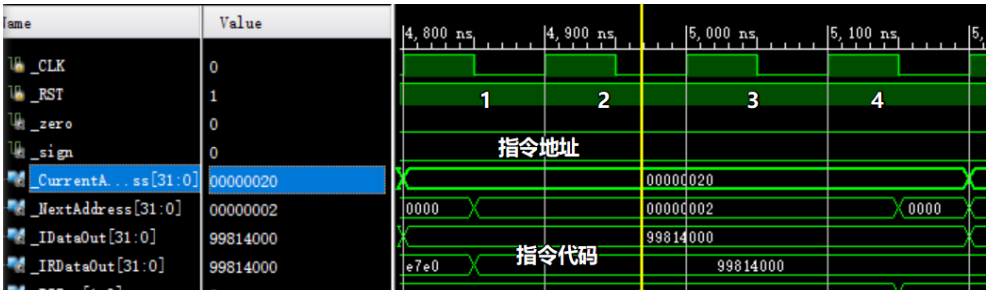


译码得到，31号寄存器的值00000020，然后进入四选一寻址模块，产生下一条指令地址00000020。

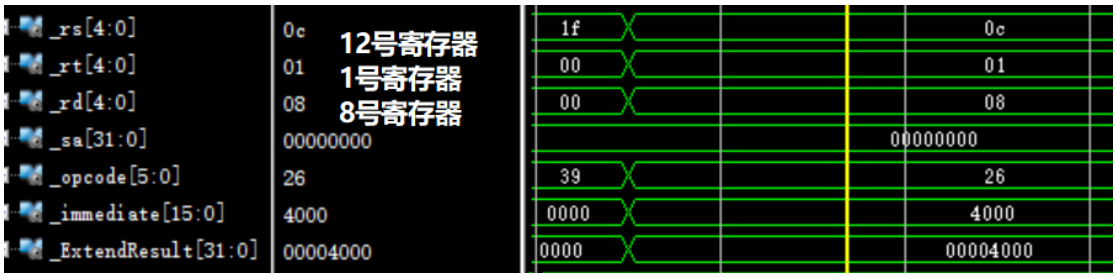
然后跳转回到地址00000020，执行slt指令。

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	
0x00000020	slt $8,$12,$1	100110	01100	00001	0100 0000 0000 0000		0x99814000

当前指令需要4个周期完成，取值，译码，执行，写回。取指周期下降沿时，指令寄存器输出指令代码。

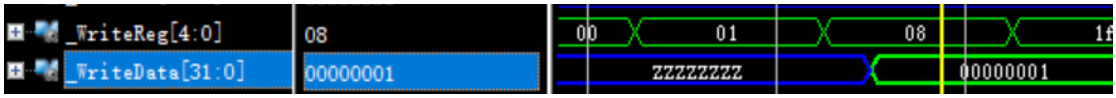


译码阶段；得到12号寄存器，1号寄存器，8号寄存器。

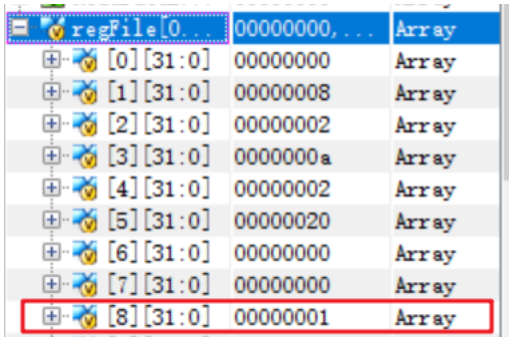


执行阶段：12号寄存器值小于1号寄存器值，运算结果为1.

写回阶段：对8号寄存器，写入运算结果00000001.



从寄存器组中的存储数组中可以看到8号寄存器值为00000001。



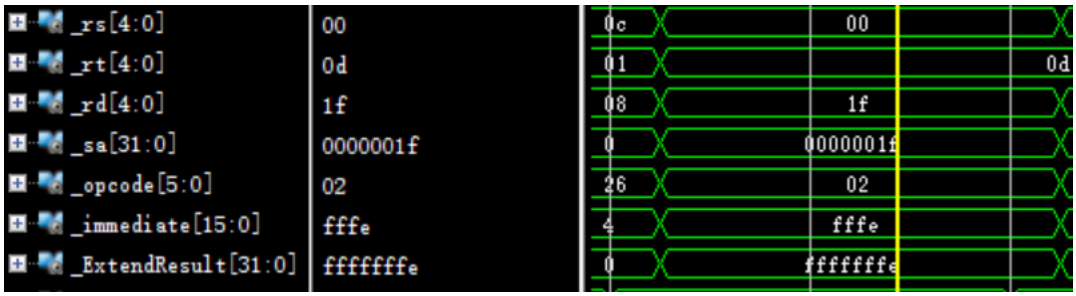
地址	汇编程序	指令代码

		op（6）	rs(5)	rt(5)	rd(5)/immediate(16)	16 进制数代码	
0x00000024	**addi $13,$0,-2**	000010	00000	01101	1111 1111 1111 1110	=	0x080DFFFE

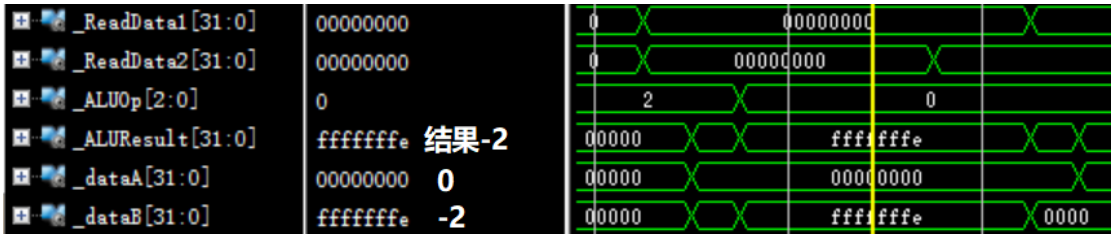
该指令需要4个周期，分别用与取指，译码，执行，写回。

当前指令地址为00000024，下一条指令地址00000028，顺序执行。

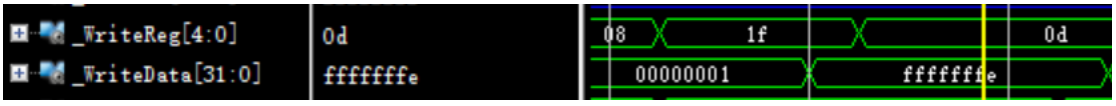
译码阶段：得到0，13号寄存器，以及立即数-2扩展，得到fffffffe。



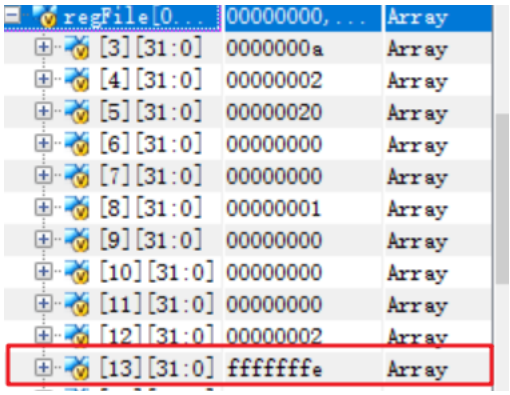
执行阶段：0号寄存器0和立即数-2相加得到结果-2.



写回阶段：将运算结果-2（fffffffe）写回13号寄存器。



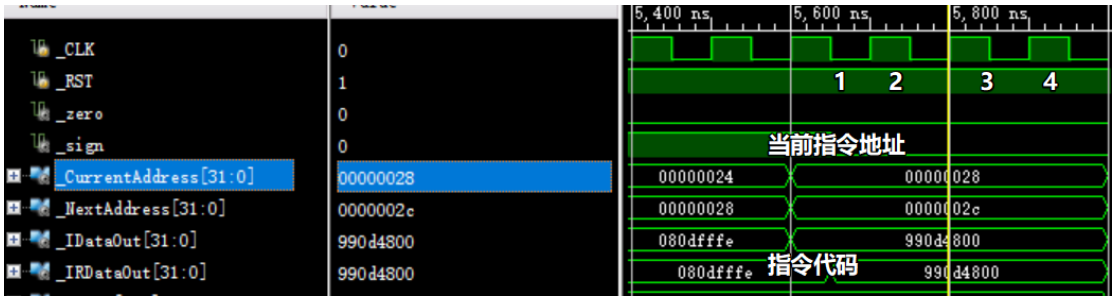
进入寄存器组可以看到13号寄存器值为fffffffe。



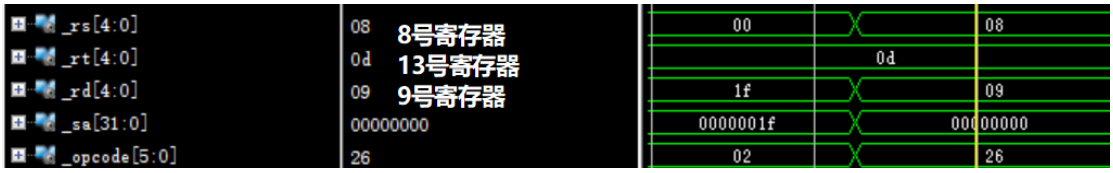
由上图可以13号寄存器值为-2，8号寄存器值为1.

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
0x00000028	slt $9,$8,$13	100110	01000	01101	0100 1000 0000 0000	= 0x990D4800

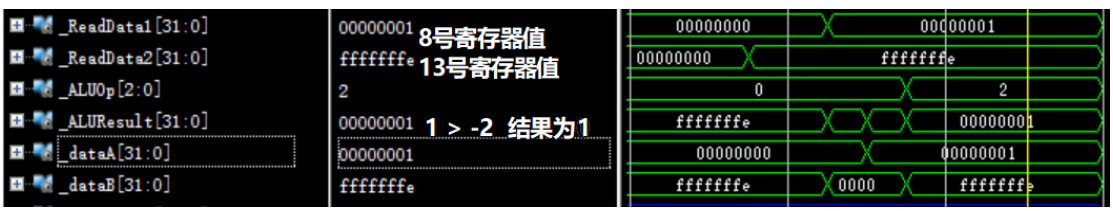
该指令需要4个周期，用于取指，译码，执行，写回。



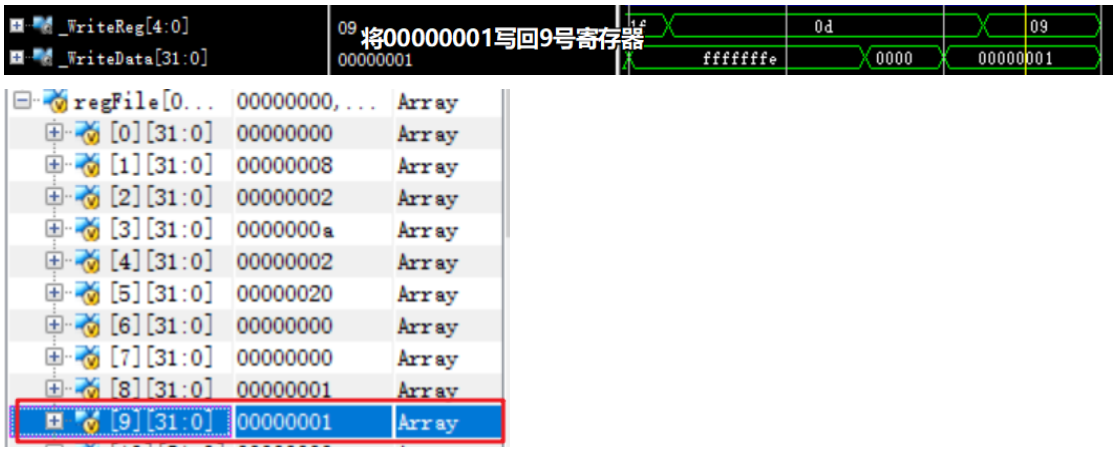
译码阶段：得到8，9，13号寄存器。



执行阶段：8号寄存器值为1，大于13号寄存器值-2，得到结果1，则rt = 1。



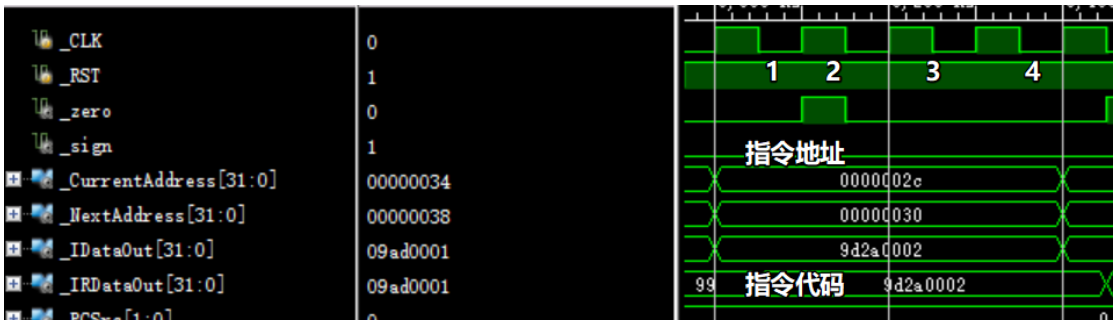
写回：将00000001写回9号寄存器。



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
0x0000002C	sltiu $10,$9,2	100111	01001	01010	0000 0000 000 00010	= 0x9D2A0002

由上可知9号寄存器内容为1.

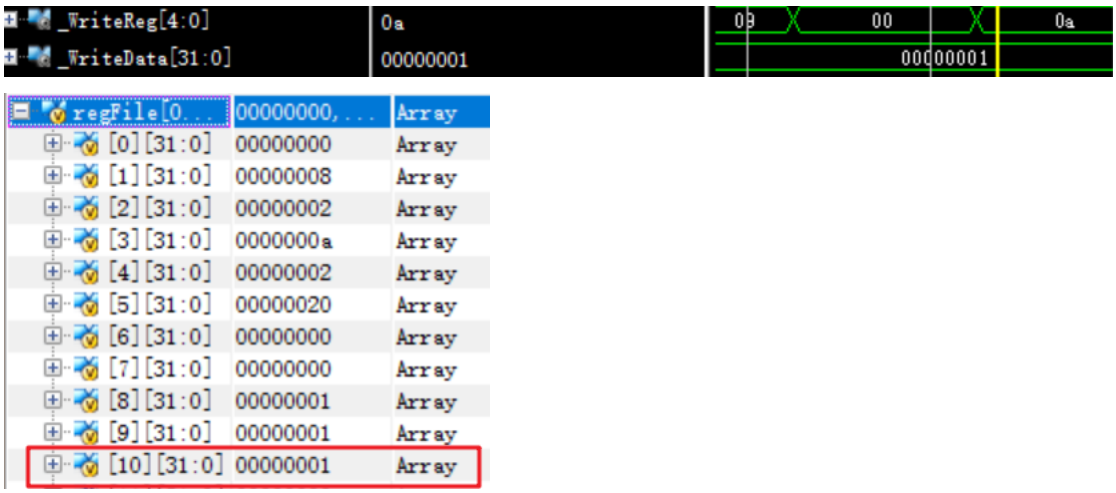
当前指令地址为0000002c，指令代码为9d2a0002，需要4个周期，用于取指，译码，执行，写回。



取指阶段：9号和10号寄存器，2数字0位扩展。

执行阶段：9号寄存器值1和数字2做比较，1<2,得到运算结果1，则rt = 1

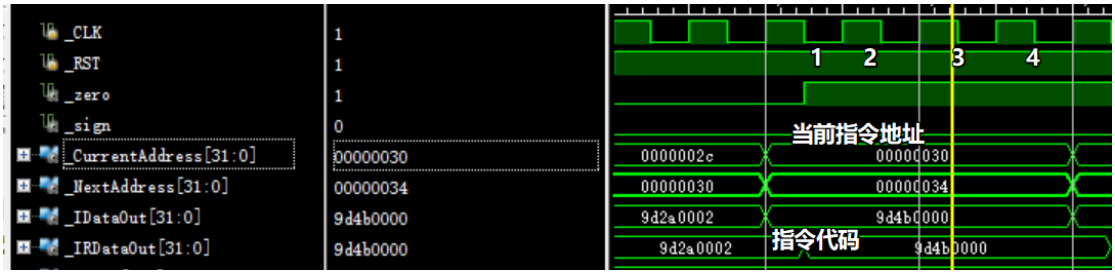
写回阶段：将运算结果00000001写回寄存器值10.



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
0x00000030	sltiu $11,$10,0	100111	01010	01011	0000 0000 0000 0000	= 0x9D4B0000

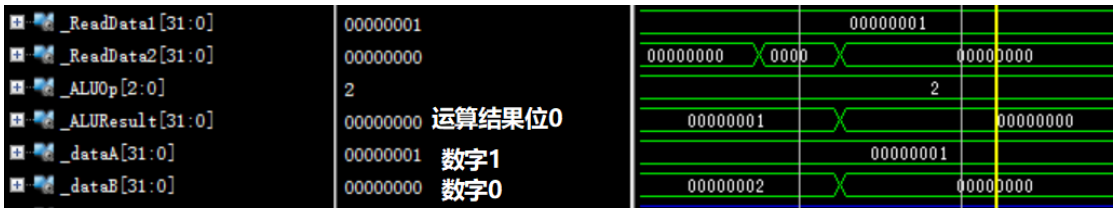
当前指令需要4个周期，用于取指，译码，执行，写回。

可以看到当前指令位000000030，下一条指令位00000034。



译码阶段：得到10和11号寄存器，以及数字0的扩展。

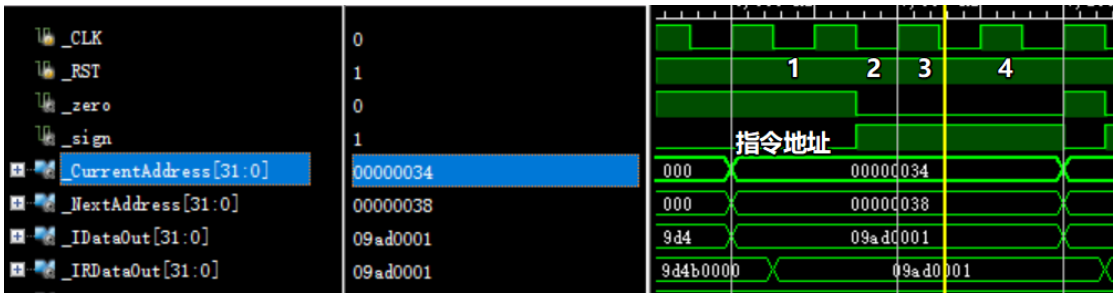
执行阶段：对10号寄存器值1和数字0，进行比较，得到结果0，



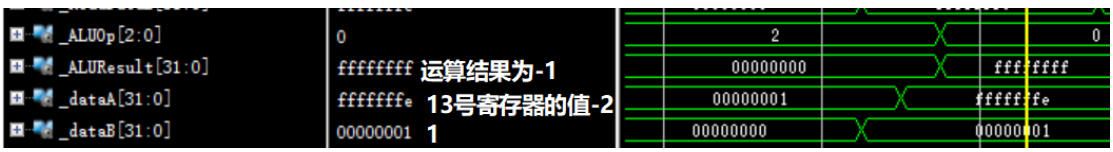
写回阶段：将运算结果0，写入11号寄存器。

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	
0x00000034	addi $13,$13,1	000010	01101	01101	0000 0000 0000 0001	=	0x09AD0001

当前指令需要4个周期，取指，译码，执行，写回。



执行阶段：-2 + 1得到运算结果为-1。

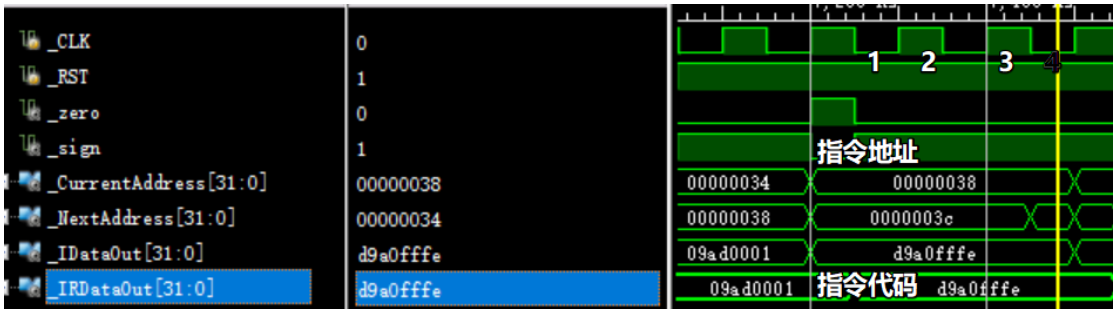


写回阶段：将运算结果-1写入13号寄存器。

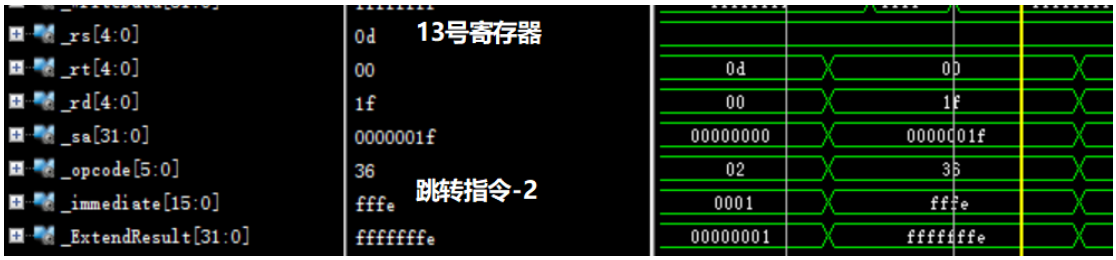
地址	汇编程序	指令代码					
		op（6）	rs(5)	rt(5)	rd(5)/immediate(16)	16 进制数代码	
0x00000038	bltz $13,-2 (<0, 转 34)	110110	01101	00000	1111 1111 1111 1110	=	0xD9A0FFFE

当前指令需要3个周期，取指，译码，执行。

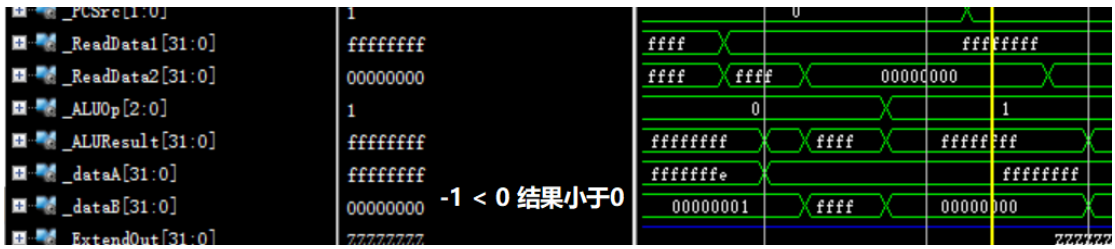
当前指令地址为00000038，下一条指令为00000034，需要跳回上一条指令，执行13号寄存器累加1。



译码阶段；得到13号寄存器和相邻跳转指令数-2.

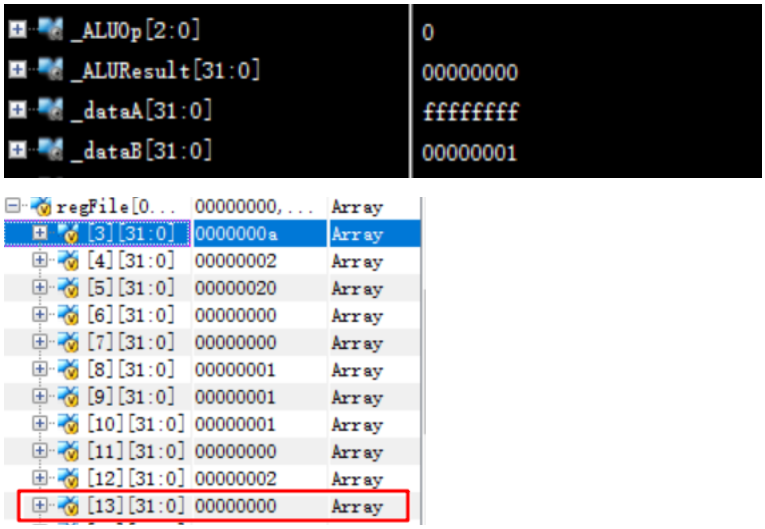


执行阶段：13号寄存器值为-1，小于0，得到运算结果为负数，符号为sign = 1



产生跳转地址：需要PC+4 + (立即数<<2),然后得到地址00000034.

重新执行00000034所在指令，13号寄存器数值由-1 变为0。



然后继续进入bltz指令。13号寄存器值等于0，顺序执行下一条指令

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	
0x0000003C	j 0x000004C	111000	00000	00000	0000 0000 0001 0011	=	0xE0000013

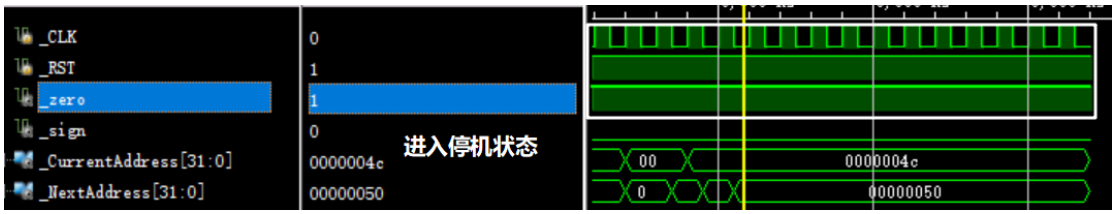
该指令需要执行两个周期，取指和译码周期，然后完成跳转。

当前指令地址为0000003c，下一条指令地址为0000004c。

产生跳转地址

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	
0x0000004C	halt	111111	00000	00000	0000 0000 0000 0000	=	0xFC000000

该指令执行后进入停机状态，地址一直为0000004c.



六、 实验心得

上次单周期CPU实验的经历，让我掌握了基本的Verilog语言的用法，以及如何使用vivado来协助程序的更好编写，在理解了多周期和单周期CPU实验的异同之后，实验难度不是想象中的大。

本次多周期CPU实验和上一次的单周期CPU实验的原理大致相似，主要模块差别不大，大体结构都是通过控制单元根据指令操作码，输出控制信号控制每个模块。但是增加了许多新的指令，难度稍微加大了，实验模块由13个变为20个。而且基于其多周期的特性，控制单元的内部结构变得更为复杂。还需要加入数据寄存器或指令寄存器来切分数据通路，将指令执行过程切分为多个周期来完成。

整个实验中，最花费时间和脑力的模块当属于控制单元模块。该模块可谓整个CPU的核心“首脑”，对其他模块进行“发号施令”。整个多周期实验的主要概念就是将每一条指令的每一个阶段用一个周期来完成，在本次实验中，根据阶段的划分，有取指周期，译码周期，执行周期，存储周期，以及写回周期，每一个周期对应一个状态。根据老师的提示，将控制单元内部结构提炼为3个小模块：D触发器模块（用于输出当前状态，接收下一步状态）；状态转换模块（确定下一个周期是什么状态）；以及输出信号模块（辨别指令和状态输出信号）。

一开始比较难以理解的是内部的指令状态转换问题，怎么配合不同的指令来产生不同的状态，后来结合状态转换图和指令操作码解决了这个问题。据我们所知，相同类型的指令基本具有相同的状态转换关系，不同指令需要经历的状态转换周期不同，jal等跳转指令需要取指，译码2个周期，add需要取指，译码，执行，写回4个周期，搞清楚每一条指令的状态变换关系是实验成功的关键，而且同一类型指令的操作码具有局部相同的特性，根据操作码来识别当前执行指令，结合上一周期的状态产生下一阶段的指令状态。

解决了状态转换的问题，来到了本实验最难的部分——根据指令操作码和状态来确定每一状态相应的控制信号。这需要对每一条指令的过程相当熟悉才不会出现太多bug一开始对某些指令比较陌生（如beq，sltiu，slt），所以对这些指令的控制信号不是很明白，需要通过运行程序，观察波形以及模块内部数据才能逐步排查。

该CPU之所以能够多周期执行，是因为在每个指令阶段之间加入了寄存器，切分了数据通路。在指令存储器之后加入一个指令寄存器，在下降沿触发输出指令代码，这里划分了取指和译码阶段；在寄存器组的输出之后加入两个数据寄存器，用于缓存当前数据，等待下一周期下降沿输出数据，这里划分了译码和执行阶段；在ALU运算结果后加入了数据寄存器，这里划分了执行阶段和存储阶段；在写回ALU运算结果或者存储器输出的时候，加入了数据寄存器，划分了执行（或存储）阶段和写回阶段。正是基于这些寄存器的存在，才让一条指令有多个周期执行的条件。

至此，多周期CPU的概念和特点也差不多清晰了，但多周期CPU优点在哪呢？一开始认为单周期CPU，一个周期完成一条指令效率不是更好计算不是更好吗，但后来想想，多周期CPU的好处在于保持数据的稳定，同时提高了CPU的工作频率，还为组成指令流水线提供了基础。

回头看，单周期CPU，多周期CPU，流水线CPU的设计就是一个迭代优化的过程。相信有了多周期CPU设计的基础，对流水线的CPU的设计也有了一定的认识。多周期CPU的设计，也让我对计算机组成原理有了更为深刻、清醒的认识。