《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
学生姓名	：	朱多煜
学号	：	16340314
专业（班级）	：	16软件工程四（8）班
时间	：	2018 年 5 月 27 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：

(1) 取指令(**IF**)：根据程序计数器PC中的指令地址，从存储器中取出一条指令，同时，PC根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入PC，当然得到的“地址”需要做些变换才送入PC。

(2) 指令译码(**ID**)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

(3) 指令执行(**EXE**)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

(4) 存储器访问(**MEM**)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

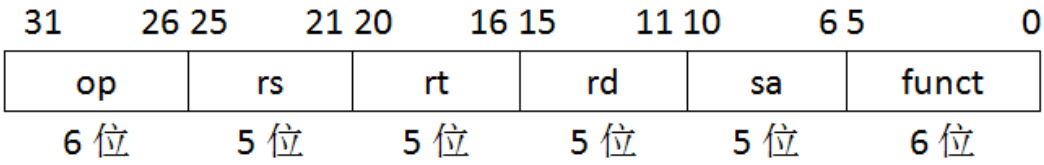
(5) 结果写回(**WB**)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

单周期CPU，是在一个时钟周期内完成这五个阶段的处理。

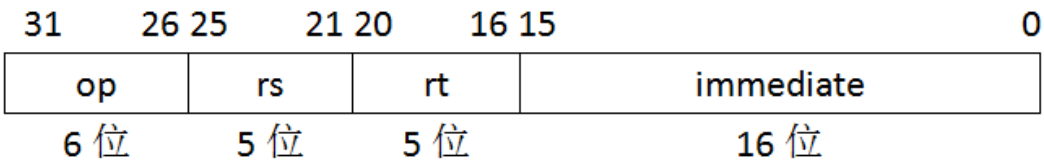
图1 单周期CPU指令处理过程

MIPS指令的三种格式：

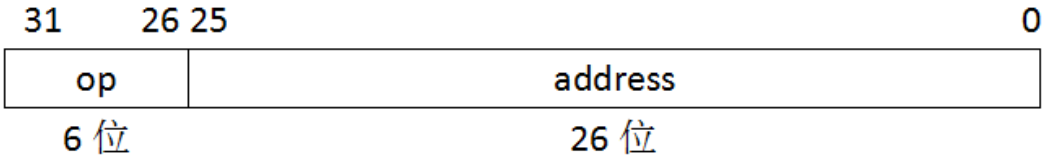
R 类型：



I 类型：



J 类型：



其中，

op：为操作码；

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

r t：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。

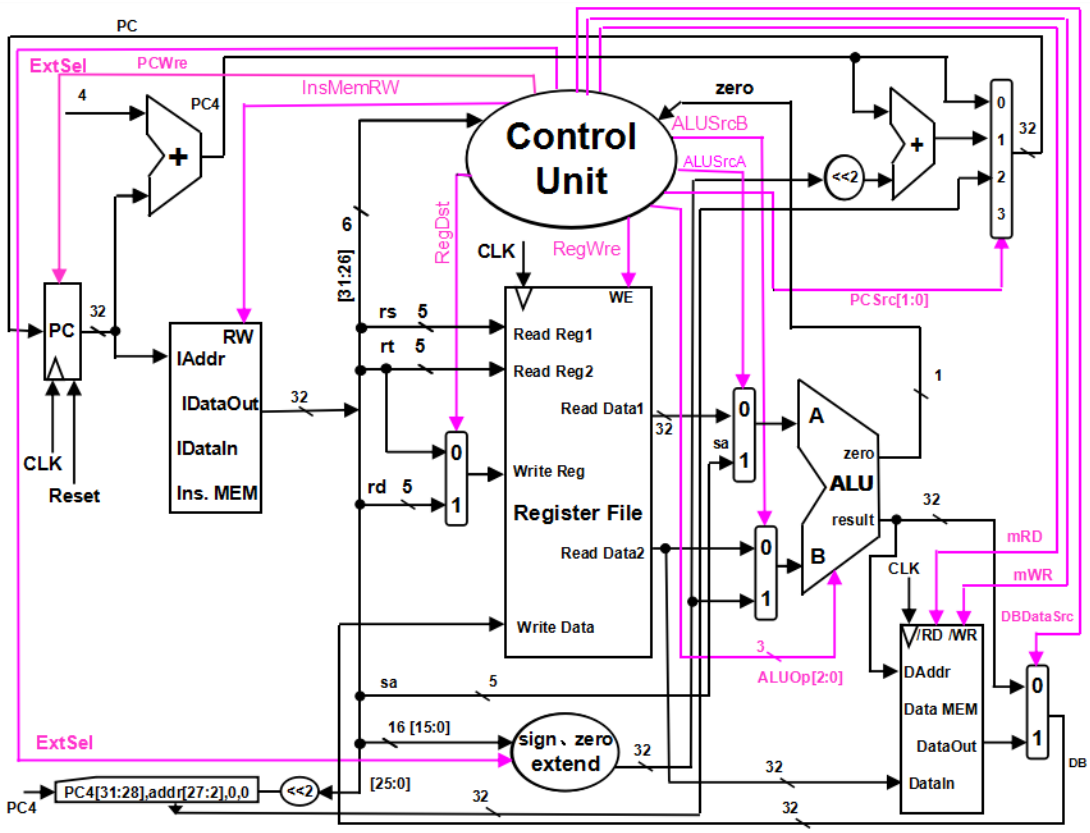


图2 单周期CPU数据通路和控制线路图

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。

四．实验器材

PC机一台，BASYS 3 实验板一块，Xilinx Vivado 开发软件一套。

五．实验分析与设计

1. 各模块的代码实现以及设计思路

（1）PC模块——PC模块主要功能是获取下一条要执行的指令，并输出该指令。由PCWre来控制指令改变的格式，代码如下：

```
`timescale 1ns / 1ps
module PC(
    input CLK,
    input Reset,
    input [31:0] PC,
    input PCWre,
    output reg [31:0] IAddr
);
    initial IAddr<=0;
    always @(posedge CLK or negedge Reset)begin
        if(Reset==0) IAddr<=0;
        else if(PCWre) IAddr<=PC;
        else IAddr<=IAddr;
    end
endmodule
```

（2）指令改变模块——在数据通路图中，获取下一个指令的方式有三种，第一种方式为在原地址的基础上加4，第二种为将传入的立即数左移两位，再加上原指令加4，第三种为获取到所需要跳转的指令。各改变方式的代码如下

<1> 直接加4

```
`timescale 1ns / 1ps
module AddPC1(
    input [31:0] PC,
    output reg [31:0] PC4
);
    initial PC4<=0;
    always @(PC)begin
        PC4=PC+4;
    end
endmodule
```

<2> 左移立即数后加原指令，再加4

```
`timescale 1ns / 1ps
module AddPC2(
    input [31:0] pc4,
    input [31:0] extend_result,
    output reg [31:0] newpc
);
    //reg [31:0] shift;
    initial begin
        newpc<=0;
        //shift<=0;
    end
    always@(pc4 or extend_result)begin
        //shift[1:0]=2'b00;
        //shift[31:2]=extend_result[29:0];
        newpc=(extend_result<<2)+pc4;
    end
endmodule
```

<3> 指令跳转

```
`timescale 1ns / 1ps
module AddPC2(
    input [31:0] pc4,
    input [31:0] extend_result,
    output reg [31:0] newpc
);
    //reg [31:0] shift;
    initial begin
        newpc<=0;
        //shift<=0;
    end
    always@(pc4 or extend_result)begin
        //shift[1:0]=2'b00;
        //shift[31:2]=extend_result[29:0];
        newpc=(extend_result<<2)+pc4;
    end
endmodule
```

（3）指令选择模块——指令选择模块通过PCSrc输入去决定，下一条指令来自于以上三条指令中的哪一个，模块代码和真值表如下

PCSrc	nextPC
00	PC+4
01	PC4+(immediate<<2)
10	JumpPC
11	未用

```
`timescale 1ns / 1ps

module NextPC(
    input [31:0] outPC0,
    input [31:0] outPC1,
    input [31:0] outPC2,
    input [1:0] PCSrc,
    output reg [31:0] nextPC
);
    initial nextPC <= 0;
    always@(PCSrc or outPC0 or outPC1 or outPC2)
    begin
        case(PCSrc)
            2'b00: nextPC = outPC0;
            2'b01: nextPC = outPC1;
            2'b10: nextPC = outPC2;
            //default nextPC = 0;
        endcase
    end
endmodule
```

（4）指令存储模块——将我们需要测试的指令从文件中读入，并储存。模块等我代码实现如下

```
`timescale 1ns / 1ps
module RW(
    input InsMemRW,
    input [31:0] IAddr,
    output reg [31:0] IDataOut
    );

    reg [7:0] mem[0:99];

    initial begin
       $readmemb ("C:/Users/zhuduoyu/SingleCPU/rom.txt", mem);
       IDataOut<=0;
     end
    always @( IAddr or InsMemRW)
        if(InsMemRW==0) begin
            IDataOut[7:0] = mem[IAddr + 3];
            IDataOut[15:8] = mem[IAddr + 2];
            IDataOut[23:16] = mem[IAddr + 1];
            IDataOut[31:24] = mem[IAddr];
        end
endmodule
```

（5）寄存器模块——寄存器模块的实现包括两部分，一部分是WriteReg的输入选择，另一部分就是寄存器的主体部分。寄存器主体部分的代码老师已经给出，并没有做任何的修改。输入选择部分则是由RegDst输入来选择rt或者rd的输入。这两部分的代码如下：

<1> 主体部分

```
`timescale 1ns / 1ps

module RegFile(
        input CLK,
        input RST,
        input RegWre,
        input [4:0] ReadReg1,
        input [4:0] ReadReg2,
        input [4:0] WriteReg,
        input [31:0] WriteData,
        output [31:0] ReadData1,
        output [31:0] ReadData2
        );
        reg [31:0] regFile[1:31]; //  ¼Ä´æ·¶¨Òå±ØÐëÓÃregÀàÐÍ
        integer i;

        assign  ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1];   // ¶Á¼Ä´æ·ÊÝ¼Ý
        assign  ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];

        always @ (negedge CLK or negedge RST) begin   // ±ØÐëÓÃÊ±ÖÓ±ßÑØ´¥·¢
            if (RST==0) begin
                  for(i=1;i<32;i=i+1)
                  regFile[i] <= 0;
            end
            else if(RegWre == 1 && WriteReg != 0)      // WriteReg != 0£¬$0¼Ä´æ·²»ÄÜÐ´Ä
                  regFile[WriteReg] <= WriteData;            // Ð´¼Ä´æ·
        end
endmodule
```

<2> 输入选择部分

```
`timescale 1ns / 1ps
module ChooseReg(
    input RegDst,
    input [4:0] rt,
    input [4:0] rd,
    output reg [4:0] wr
);
    initial wr<=0;
    always@(RegDst or rt or rd) begin
        wr=(RegDst==0)?rt:rd;
    end
endmodule
```

(6)立即数扩展模块——立即数扩展模块实现了将16位的立即数扩展为32位，以便于其参加之后模块的运算。该模块由ExtSel输入控制是有符号扩展还是无符号扩展。真值表和代码如下：

ExSel	补全
0	进行无符号扩展
1	进行有符号扩展

```
`timescale 1ns / 1ps
module extend(
    input [15:0] datain,
    input ExtSel,
    output reg [31:0] result
    );
    always@(datain or ExtSel) begin
        result[15:0]<=datain[15:0];
        if(ExtSel==1 && datain[15]==1)
            result[31:16]<=16'b1111_1111_1111_1111;
        else
            result[31:16]=0;
    end
endmodule
```

（7）ALU模块——ALU模块分为三个部分，包括A端的输入选择模块，B端的输入选择模块和ALU主体模块。两个输入模块分别由ALUSrcA，ALUSrcB控制输出信息的选择。模块A在ReadData1和sa中选择输出，模块B在ReadData2 和扩展后的立即数中选择输出。ALU主体模块的真值表和各模块的代码如下。

<1> ALU主体模块真值表

表1 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 Å B	异或

<2>ALU主体模块代码

```
`timescale 1ns / 1ps
module ALU32(
    input [2:0] ALUopcode,
    input [31:0] rega,
    input [31:0] regb,
    output reg [31:0] result,
    output zero
    );
    initial result = 0;
    always @(ALUopcode or rega or regb) begin
        case(ALUopcode)
        3'b000: result <= rega + regb;
        3'b001: result <= rega - regb;
        3'b010: result <= regb << rega;
        3'b011: result <= rega | regb;
        3'b100: result <= rega & regb;
        3'b101: result <= (rega < regb)?1:0;
        3'b110: result <= (((rega<regb) && (rega[31] == regb[31] ))||((rega[31] == 1&&regb[31]==0)))?1:0;
        3'b111: result <= rega ^ regb;
        endcase
   end
       assign zero = (result==0)?1:0;
endmodule
```

<3>A端的输入选择模块

```
`timescale 1ns / 1ps
module ChooseInputA(
    input ALUSrcA,
    input [31:0] ReadData1,
    input [4:0] sa,
    output reg [31:0] A
);
    initial A = 0;
    always@(ALUSrcA or sa or ReadData1) begin
        if(ALUSrcA==0)
            A<=ReadData1;
        else
        begin
            A[4:0]<=sa[4:0];
            A[31:5]<=0;
        end
    end
endmodule
```

<4>B端的输入选择模块

```
`timescale 1ns / 1ps
module ChooseInputB(
    input ALUSrcB,
    input [31:0] ReadData2,
    input [31:0] extend_result,
    output reg [31:0] B
);
    always@(ALUSrcB or ReadData2 or extend_result) begin
        if(ALUSrcB==0)
            B<=ReadData2;
        else
            B<=extend_result;
    end
endmodule
```

（8）数据储存和输出模块——该模块包括数据的储存模块和输出模块两个部分.其中数据储存的部分的代码老师也已经给出，代码如下：

<1> 数据储存模块

```
`timescale 1ns / 1ps
module RAM(
    input clk,
    input [31:0] address,
    input [31:0] writeData,
    input nRD,
    input nWR,
    output [31:0] Dataout
    );


    reg [7:0] RAM [0:31];
    integer i;
    initial
    begin
        for(i = 0; i < 32; i = i+1) RAM[i] <= 0;
    end

    assign Dataout[7:0]   = (nRD==0)?RAM[address + 3]:8'bz;
    assign Dataout[15:8]  = (nRD==0)?RAM[address + 2]:8'bz;
    assign Dataout[23:16] = (nRD==0)?RAM[address + 1]:8'bz;
    assign Dataout[31:24] = (nRD==0)?RAM[address ]:8'bz;


    always@( negedge clk ) begin
       if( nWR==0 ) begin
           RAM[address]   <= writeData[31:24];
           RAM[address+1] <= writeData[23:16];
           RAM[address+2] <= writeData[15:8];
           RAM[address+3] <= writeData[7:0];
       end
    end
endmodule
```

<2>数据输出模块——数据输出模块由DBDataSrc控制，选择直接输出计算结果还是从数据储存模块中输出结果

```
`timescale 1ns / 1ps
module DBoutput(
    input [31:0] result,
    input [31:0] DataOut,
    input DBDataSrc,
    output reg [31:0] DB
);
    initial DB<=0;
    always @(result or DataOut or DBDataSrc)begin
        if(DBDataSrc==0)
            DB<=result;
        else
            DB<=DataOut;
    end
endmodule
```

（9）transTocut模块——并不是属于CPU的必要模块，是在查找相关资料时发现的一个简化输出的模块，便于我们仿真时核对指令的正确性。其主要原理就是在多位的输出中截取我们需要的那几位，具体实现的代码如下：

```
`timescale 1ns / 1ps
module transTocut(
    input [31:0] address,new_address,data1,data2,result,DB,
    input [4:0] rs,rt,
    output [7:0] addr_4,new_addr_4,rs_4,data1_4,rt_4,data2_4,result_4,
    output [7:0] DB_4
);

    assign      addr_4[7:0] = address[7:0];
    assign      new_addr_4[7:0] = new_address[7:0];
    assign      rs_4[4:0] = rs[7:0];
    assign      rs_4[7:5] = 0;
    assign      data1_4[7:0] = data1[7:0];
    assign      rt_4[4:0] = rt[7:0];
    assign      rt_4[7:5] = 0;
    assign      data2_4[7:0] = data2[7:0];
    assign      result_4[7:0] = result[7:0];
    assign      DB_4[7:0] = DB[7:0];
endmodule
```

（10）ControlUnit模块——该模块可以说是整个单周期CPU的控制中心，通过op的输入来确定指令的格式，并通过各命令的输出，去控制其他模块的运行。其中个控制信号的作用以及相关指令如下：

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

相关部件及引脚说明：

Instruction Memory：**指令存储器**，

Iaddr，指令存储器地址输入端口

IDataIn，指令存储器数据输入端口（指令代码输入端口）

IDataOut，指令存储器数据输出端口（指令代码输出端口）

RW，指令存储器读写控制信号，为0写，为1读

Data Memory：**数据存储器**，

Daddr，数据存储器地址输入端口

DataIn，数据存储器数据输入端口

DataOut，数据存储器数据输出端口
/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

其代码如下：

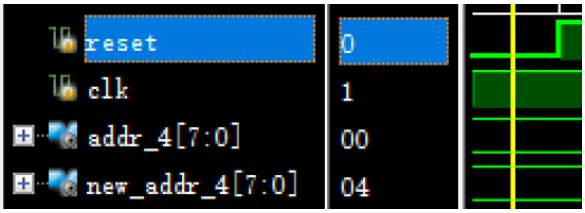
```
`timescale 1ns / 1ps
module ControlUnit(
    input [5:0] Op,
    input Zero,
    output RegDst,
    output InsMemRW,
    output PCWre,
    output ExtSel,
    output DBDataSrc,
    output mWR,
    output mRD,
    output ALUSrcB,
    output ALUSrcA,
    output [1:0] PCSrc,
    output [2:0] ALUOp,
    output RegWre
);
    assign PCWre=(Op==6'b111111)?0:1;
    assign ALUSrcA=(Op==6'b011000)?1:0;
    assign ALUSrcB=((Op==6'b000001)|||(Op==6'b010000)|||(Op==6'b011011)|||(Op==6'b100110)|||(Op==6'b100111))?1:0;
    assign DBDataSrc=(Op==6'b100111)?1:0;
    assign RegWre=((Op==6'b110000)|||(Op==6'b110001)|||(Op==6'b100110)|||(Op==6'b111000)|||(Op==6'b111111))?0:1;
    assign InsMemRW=0;
    assign mRD=(Op==6'b100111)?0:1;
    assign mWR=(Op==6'b100110)?0:1;
    assign RegDst=((Op==6'b000001)|||(Op==6'b010000)|||(Op==6'b011011)|||(Op==6'b100111))?0:1;
    assign ExtSel=(Op==6'b010000)?0:1;
    assign PCSrc[1]=(Op==6'b111000)?1:0;
    assign PCSrc[0]=((Op==6'b110000&&Zero==1)|||(Op==6'b110001&&Zero==0))?1:0;
    assign ALUOp[2]=((Op==6'b011011) || (Op==6'b010001))?1:0;
    assign ALUOp[1]=((Op==6'b111111) || (Op==6'b011011) || (Op==6'b011000) || (Op==6'b010010) || (Op==6'b010000))?1:0;
    assign ALUOp[0]=((Op==6'b000010) || (Op==6'b010000) || (Op==6'b010010) || (Op==6'b110000) || (Op==6'b110001))?1:0;


endmodule
```

(11)顶层文件——将各模块连接起来，实现代码如下

```
`timescale 1ns / 1ps
module SingleCPU(
    input clk,
    input reset,
    output [7:0] addr_4,
    output [7:0] new_addr_4,
    output [7:0] rs_4,
    output [7:0] data1_4,
    output [7:0] rt_4,
    output [7:0] data2_4,
    output [7:0] result_4,
    output [7:0] DB_4
);
    wire zero;
    wire [2:0] ALUOp;
    wire mRD,mWR,RegDst,InsMemRW,PCWre,ExtSel,RegWre,DBDataSrc,ALUSrcA,ALUSrcB;
    wire [31:0] A,B,result, data1,data2,DataOut,command,extended,address,new_address,pc4,jump_addr,branch_addr,new_addr,DB;
    wire [2:0] ALUOp;
    wire [1:0] PCSrc;
    wire [4:0] WriteReg;
    AddPC2 add2(.pc4(pc4),.extend_result(extended),.newpc(branch_addr));

    ALU32 alu(.ALUopcode(ALUOp),.rega(A),.regb(B),.zero(zero),.result(result));

    ControlUnit controlunit(.Op(command[31:26]),.Zero(zero),.RegDst(RegDst),.InsMemRW(InsMemRW),.PCWre(PCWre),.ExtSel(ExtSel),.DBDataSrc(DBDataSrc),
                            .mWR(mWR),.mRD(mRD),.ALUSrcB(ALUSrcB),.ALUSrcA(ALUSrcA),.PCSrc(PCSrc),.ALUOp(ALUOp),.RegWre(RegWre));

    extend ex(.datain(command[15:0]),.ExtSel(ExtSel),.result(extended));

    JumpPC jump(.pc4(pc4),.addr(command[25:0]),.newaddr(jump_addr));

    ChooseInputA inputA(.ALUSrcA(ALUSrcA),.ReadData1(data1),.sa(command[10:6]),.A(A));

    ChooseInputB inputB(.ALUSrcB(ALUSrcB),.ReadData2(data2),.extend_result(extended),.B(B));

    NextPC nextPC(.PCSrc(PCSrc),.outPC0(pc4),.outPC1(branch_addr),.outPC2(jump_addr),.nextPC(new_address));

    AddPC1 add1(.PC(address),.PC4(pc4));

    PC pc(.CLK(clk),.Reset(reset),.PC(new_address),.PCWre(PCWre),.IAddr(address));

    RAM ram(.clk(clk),.address(result),.writeData(data2),.nRD(mRD),.nWR(mWR),.Dataout(DataOut));

    ChooseReg choosereg(.RegDst(RegDst),.rt(command[20:16]),.rd(command[15:11]),.wr(WriteReg));

    RegFile regfile(.CLK(clk),.RST(reset),.RegWre(RegWre),.ReadReg1(command[25:21]),.ReadReg2(command[20:16]),.WriteReg(WriteReg),
                    .WriteData(DB),.ReadData1(data1),.ReadData2(data2));

    DBoutput db(.result(result),.DataOut(DataOut),.DBDataSrc(DBDataSrc),.DB(DB));

    RW rw(.InsMemRW(InsMemRW),.IAddr(address),.IDataOut(command));

    transTocut trsc(.address(address),.new_address(new_address),.rs(command[25:21]),.data1(data1),.rt(command[20:16]),.data2(data2),.result(result),
                    .DB(DB),.addr_4(addr_4),.new_addr_4(new_addr_4),.rs_4(rs_4),.data1_4(data1_4),.rt_4(rt_4),.data2_4(data2_4),.result_4(result_4),.DB_4(DB_4));

endmodule
```

2. 指令执行情况

(1)初始时——初始情况时reset为0，CPU未开始工作，PC的输出地址不变，故当前地址为0，下一条地址为4。



(2)立即数相加——addi $1, $0, 8

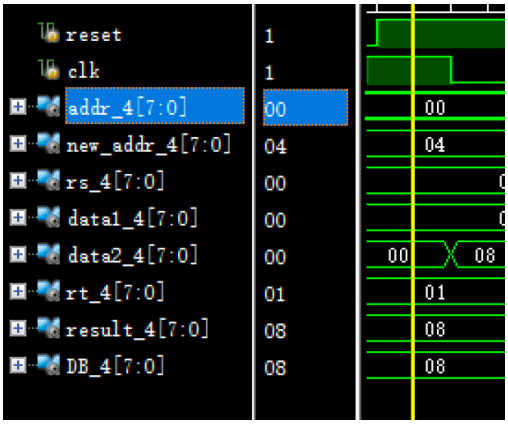
立即数相加时当前地址为0，下一条地址为4。

Rs为0号寄存器，其对应的data1为0

Rt为1号寄存器，其对应的data2在时钟下降沿时变为8

ALU加法器输出8

不经过数据储存器，数据输出DB为8



(3)立即数的或运算——ori $2,$0,2

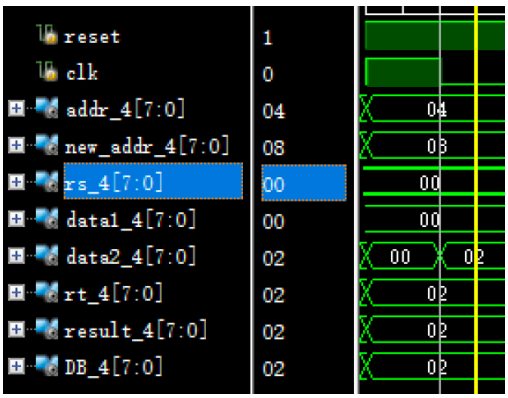
或运算时当前地址为4，下一条地址为8。

Rs为0号寄存器，其对应的data1为0

Rt为1号寄存器，其对应的data2在时钟下降沿时变为2

ALU加法器输出2

不经过数据储存器，数据输出DB为2



(4)寄存器的加法运算—— add $3,$2,$1

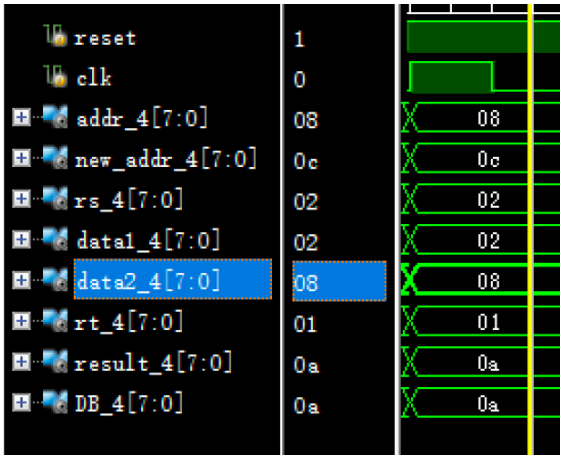
当前地址为8，下一条地址为c。

Rs为2号寄存器，其对应的data1为2

Rt为1号寄存器，其对应的data2位8,因为不需要对Rt写入数据，所以无时钟下降沿的变化

ALU加法器输出a

不经过数据储存器，数据输出DB为a



(5)寄存器的减法运算——sub $5,$3,$2

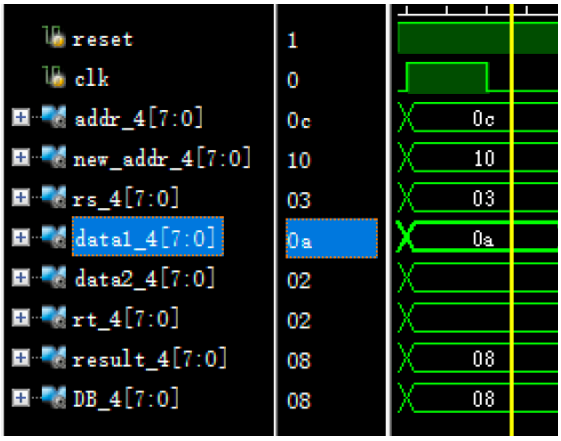
当前地址为c，下一条地址为10。

Rs为3号寄存器，其对应的data1为a

Rt为2号寄存器，其对应的data2为2,因为不需要对Rt写入数据，所以无时钟下降沿的变化

ALU加法器输出8

不经过数据储存器，数据输出DB为8



(6)与运算——and $4,$5,$2

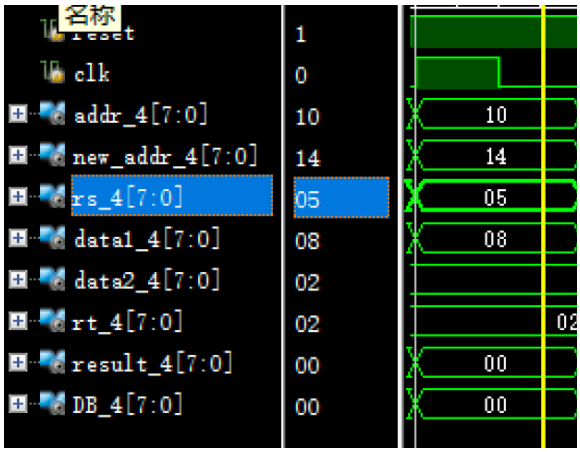
当前地址为10，下一条地址为14。

Rs为5号寄存器，其对应的data1为8

Rt为2号寄存器，其对应的data2为2,因为不需要对Rt写入数据，所以无时钟下降沿的变化

ALU加法器输出0

不经过数据储存器，数据输出DB为0



(7)或运算——or $8,$4,$2

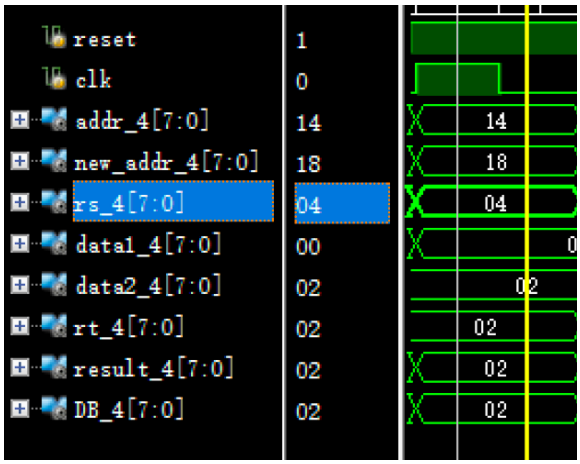
当前地址为14，下一条地址为18。

Rs为4号寄存器，其对应的data1为0

Rt为2号寄存器，其对应的data2为2,因为不需要对Rt写入数据，所以无时钟下降沿的变化

ALU加法器输出2

不经过数据储存器，数据输出DB为2



(8)位移运算——sll $8,$8,1

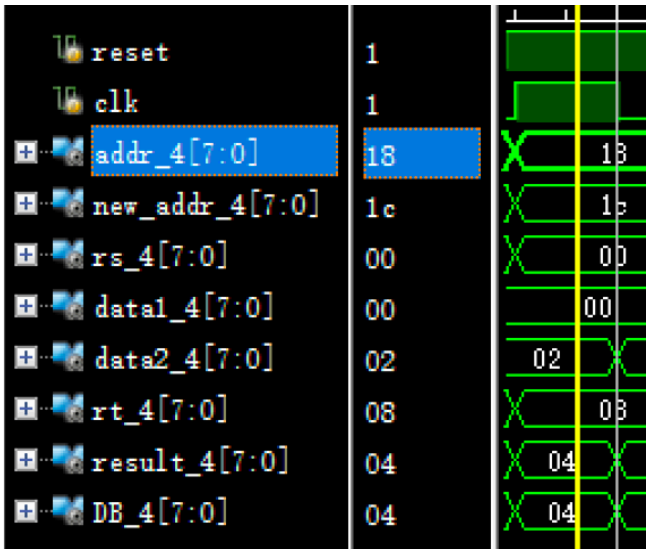
当前地址为18，下一条地址为1c。

Rs为0号寄存器，其对应的data1为0

Rt为8号寄存器，其对应的data2由2,在下一个时钟下降沿时变为4

ALU加法器输出4

不经过数据储存器，数据输出DB为4



(9)不相等时跳转的指令——bne $8,$1,-2

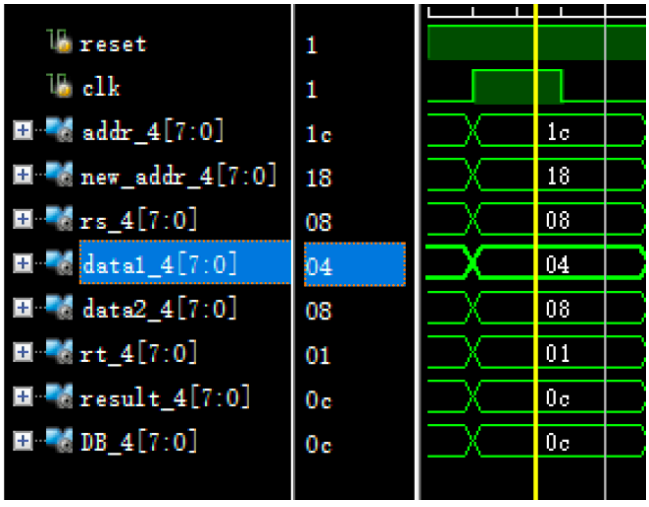
当前地址为1c，下一条地址为18。

Rs为8号寄存器，其对应的data1为4

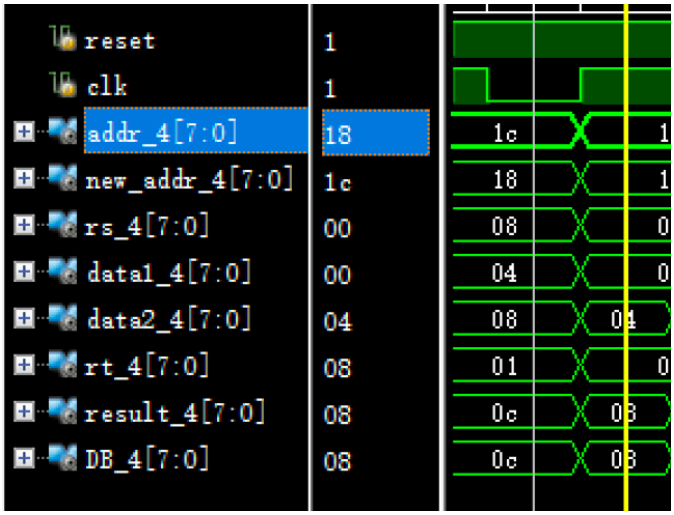
Rt为1号寄存器，其对应的data2由8,因为不需要对Rt写入数据，所以无时钟下降沿的变化

ALU加法器输出c(取异或)

不经过数据储存器，数据输出DB为c

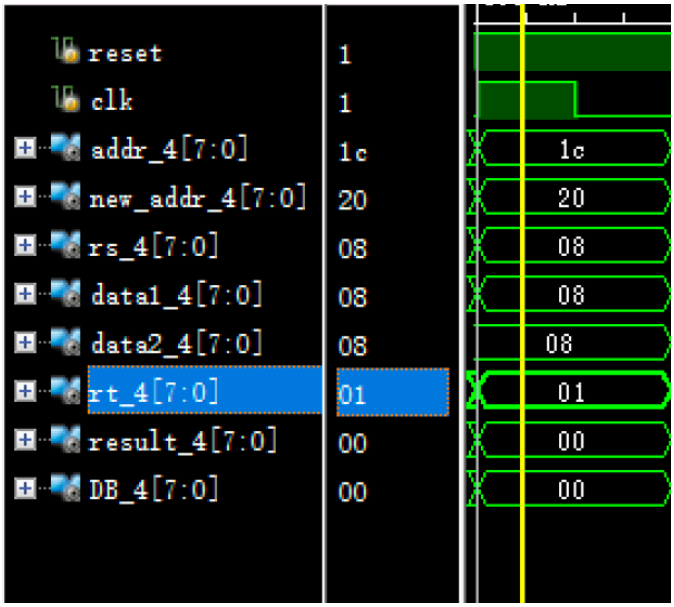


跳转到上一条指令后重新执行左移运算将8号寄存器的值变为8



这是再进行不相等时跳转，并将ALU输出为0(取异或)

不经过数据储存器，数据输出DB为0



(10)立即数比较指令——slti $6,$2, 8

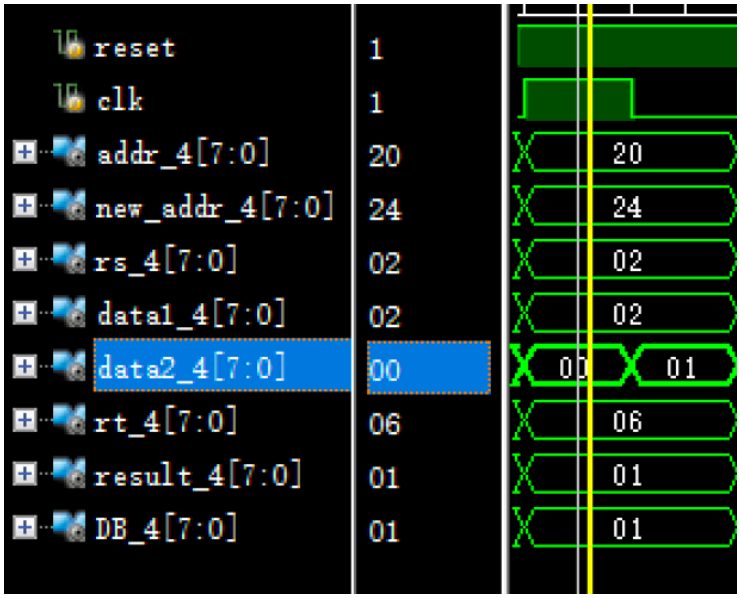
当前地址为20，下一条地址为24。

Rs为2号寄存器，其对应的data1为2

Rt为6号寄存器，其对应的data2由于立即数大于2号寄存器中的值，会在时钟下降沿变为1

ALU加法器输出1

不经过数据储存器，数据输出DB为1



slti $7,$6,0

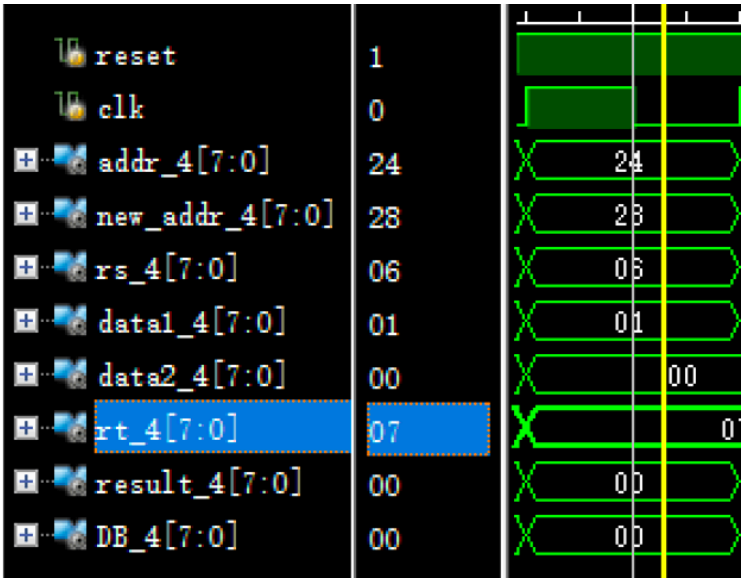
当前地址为24，下一条地址为28。

Rs为6号寄存器，其对应的data1为1

Rt为7号寄存器，其对应的data2由于立即数小于6号寄存器中的值，7号寄存器的值保持为0

ALU加法器输出0

不经过数据储存器，数据输出DB为0



addi $7,$7,8

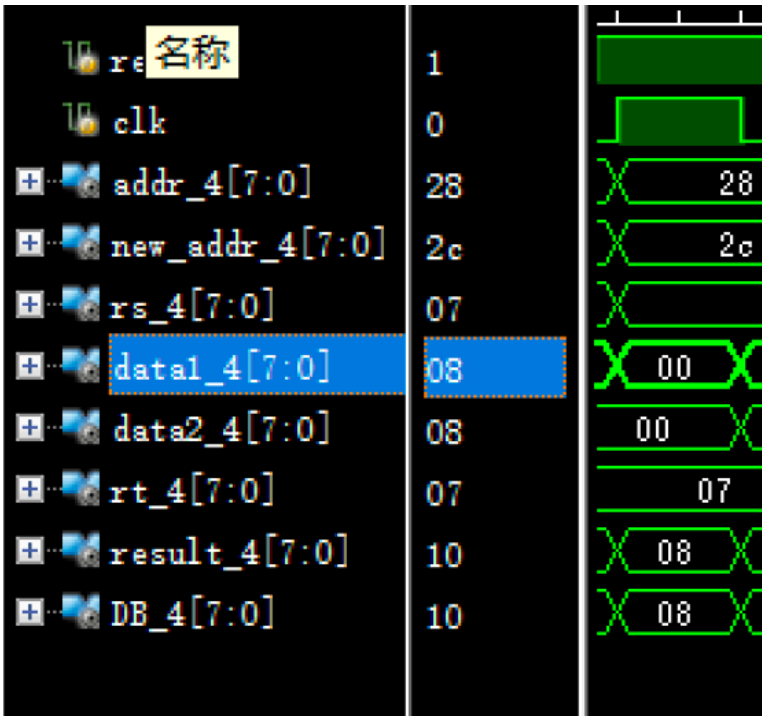
当前地址为28，下一条地址为2c。

Rs为7号寄存器，其对应的data1由0,在时钟下降沿变为8

Rt为7号寄存器，其对应的data2由0,在时钟下降沿变为8

ALU加法器输出8

不经过数据储存器，数据输出DB为8



(11)相等跳转指令——beq $7,$1,-2

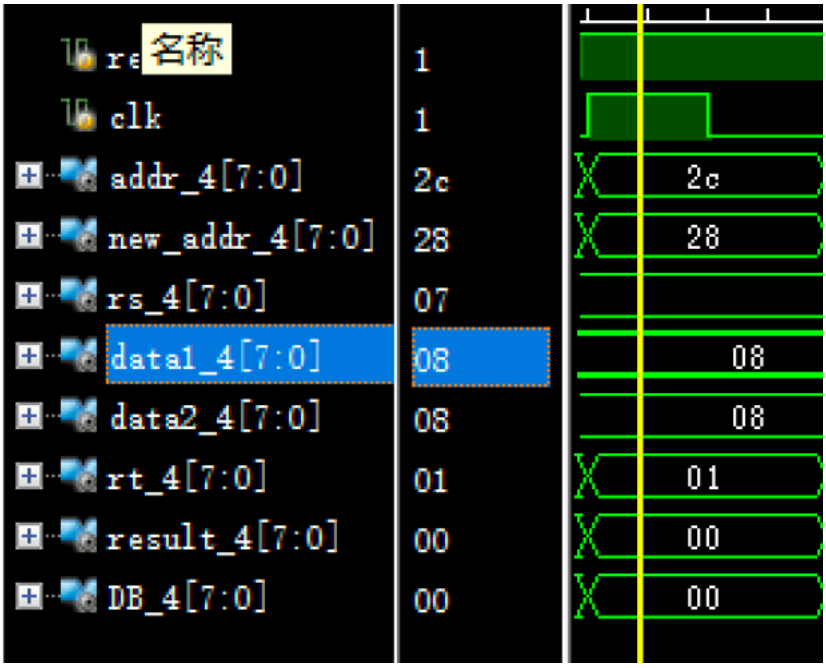
当前地址为2c，下一条地址为28。

Rs为7号寄存器，其对应的data1为8

Rt为1号寄存器，其对应的data2为8

ALU加法器输出0(取异或)

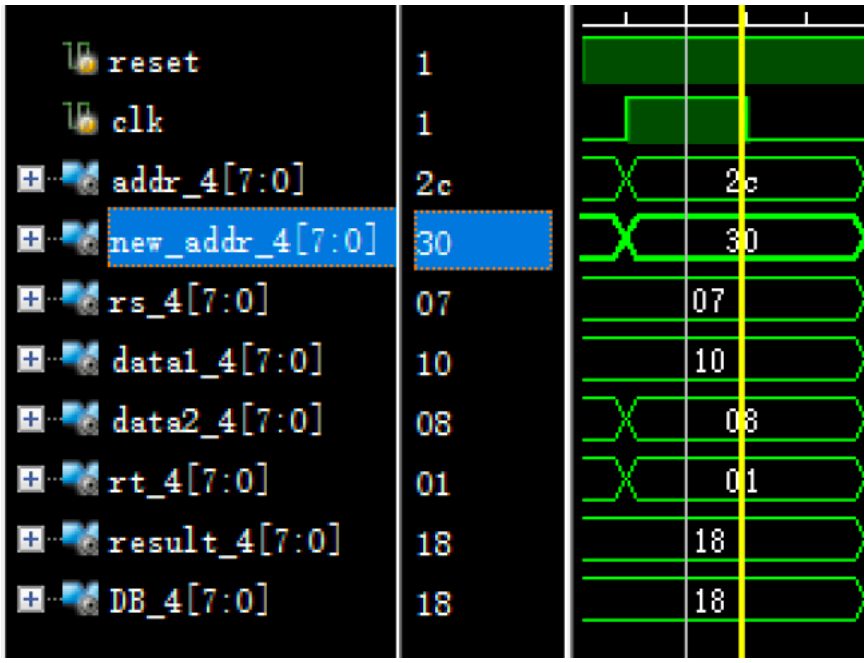
不经过数据储存器，数据输出DB为0



跳转到上一条立即数的加法指令后7号寄存器和1号寄存器的值将不相等

ALU加法器输出为18(取异或)

不经过数据储存器，数据输出DB为18



(12)写存储器——sw $2, 4($1)

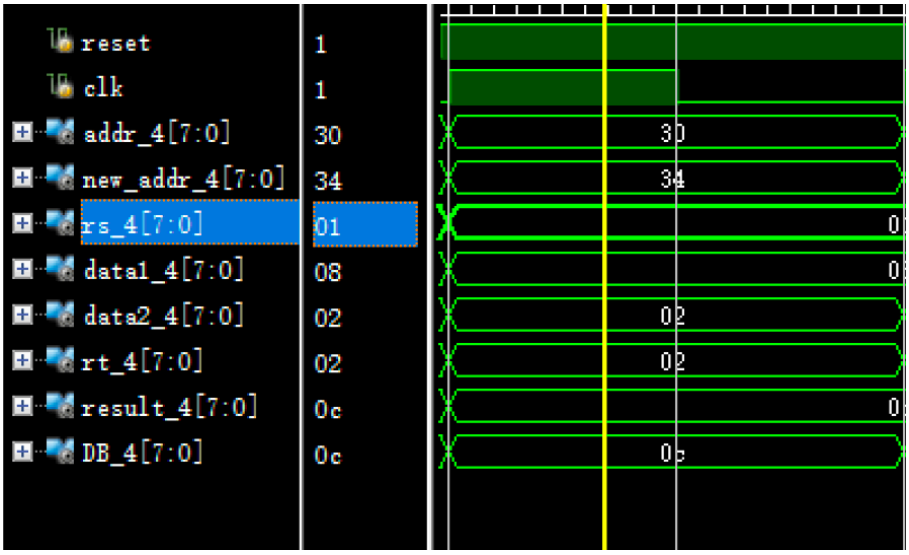
当前地址为30，下一条地址为34。

Rs为1号寄存器，其对应的data1为8

Rt为2号寄存器，其对应的data2为2

ALU加法器将8与立即数4相加输出c

不经过数据储存器，数据输出DB为c



(13)读储存器——lw $9,4($1)

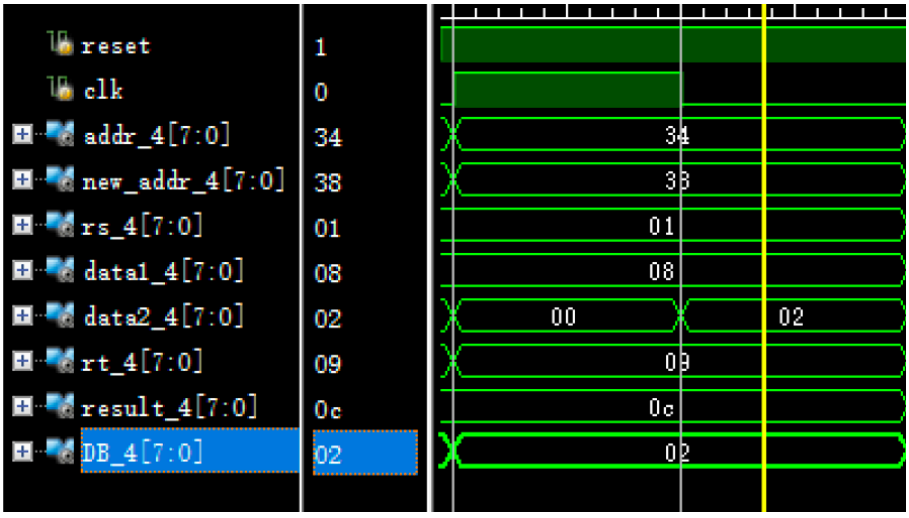
当前地址为34，下一条地址为38。

Rs为1号寄存器，其对应的data1为8

Rt为9号寄存器，其对应的data2在时钟下降沿变为2

ALU加法器将8与立即数4相加输出c

经过数据储存器，数据输出DB为2



(14)跳转指令——j 0x00000040 (跳过立即数下加法指令)

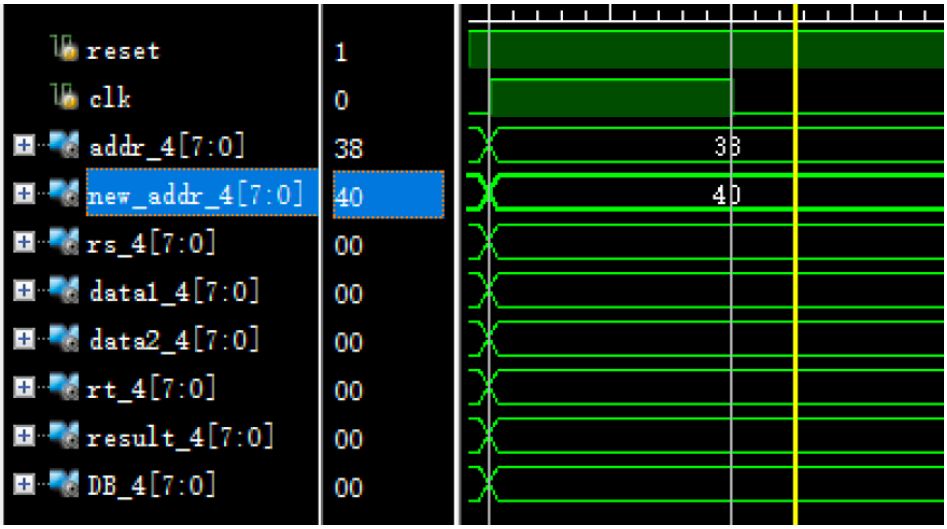
当前地址为38，下一条地址为40。

Rs为0号寄存器，其对应的data1为0

Rt为0号寄存器，其对应的data2为0

ALU加法器输出0

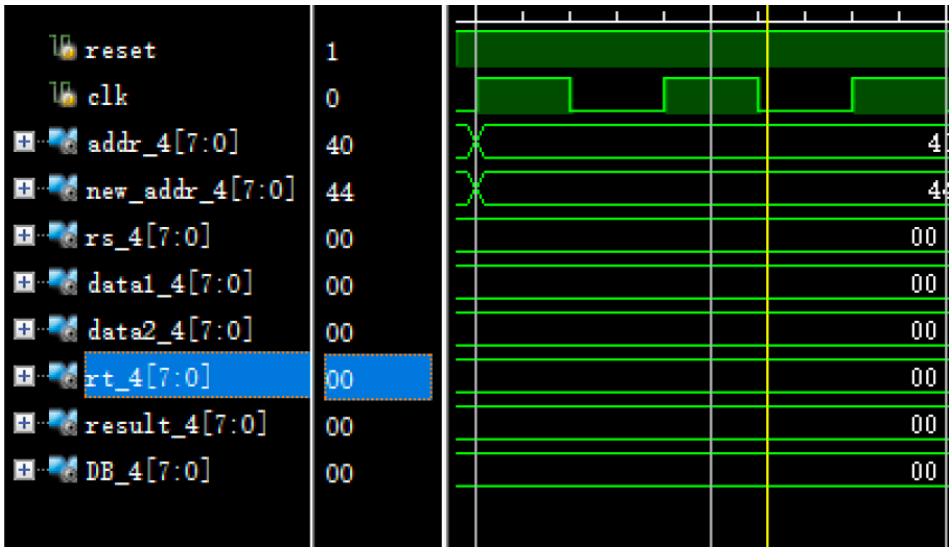
不经过数据储存器，数据输出DB为0



(15)停机指令——halt

当前地址为40，下一条地址为44。

地址不再更改



六. 实验心得

1. 在实现CPU的过程中，对数据通路的理解是至关重要的，它概括了CPU运作的各个核心模块，和实现CPU所需要解决的所有问题。我们只要真正读懂了数据通路，就会发现其实每个模块实现起来还是很简单的。

2. 虽然CPU设计看起来很难，而且还需要接触我们之前没有学过的新语言。但是当我们真的着手去一个模块一个模块慢慢实现的时候，就会发现，实现模块真的很简单。真正难的地方在于在每个模块实现后的debug过程。

3. 在仿真过程中，遇到最大的问题就是如何解决reg格式和wire格式的赋值问题。reg格式不能用于assign的赋值，assign只能对Wire类型的变量使用，而在initial段和always段进行赋值操作时又不允许使用assign，只能通过reg格式才能赋值。所以要想仿真时通过编译，就需要注意区分<=和assign哪种赋值方式具有异步的特征。

4. 在各模块的实现过程中，我们要注意对于reg类型的中间变量最好进行初始化，否则会容易出现不确定输出的问题。

5. 在刚开始编程时，对begin和end的使用情况又很多的不理解，因为在网上的很多实例代码中，似乎都没有一个很规范的使用方式。后来自己的看过Verilog的语法之后，才发现我们可以将begin和end当作时我们C++语言中的括号，用于划分程序块而已。

6. 最好在每个模块实现的过程中顺便检测一下每个模块是否能正确的运行输出。否则当实现了全部模块，然后去仿真运行的时候，很容易发现输出基本为XXXXXX或者ZZZZZ的情况。而这时我们再去一个模块一个模块，一条线一条线的检查就会很麻烦，会浪费我们的很多时间。

7. 如果使用文件读入的方式去得到我们编写的程序的指令时，一定要注意路径的问题。因为我是在虚拟机中进行仿真的，刚开始将指令文件放在桌面上就一直读取不了，不管用哪种格式的路径。最后只能选择将文件直接放进vivado的项目文件夹中，利用项目文件的目录才成功读入了文件。