



《计算机组成原理实验》 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 计算机类 4 班

学 生 姓 名 : 郑育聪

学 号 : 16337329

时 间 : 2017 年 12 月 15 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(5 位)	reserved
--------	----	---------	---------	---------	----------

功能: $rd \leftarrow rt \ll (\text{zero-extend})sa$, 左移 sa 位, $(\text{zero-extend})sa$

==>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) slti rt, rs, immediate 带符号

100111	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

(10) sw rt, immediate(rs)

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

功能: $\text{memory}[rs + (\text{sign-extend})immediate] \leftarrow rt$ 。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: $rt \leftarrow \text{memory}[rs + (\text{sign-extend})immediate]$ 。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) $pc \leftarrow pc + 4 + (\text{sign-extend})immediate \ll 2$ else $pc \leftarrow pc + 4$

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

110101	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能: if(rs!=rt) $pc \leftarrow pc + 4 + (\text{sign-extend})immediate \ll 2$ else $pc \leftarrow pc + 4$

(14) bgtz rs, immediate

110110	rs(5 位)	00000	Immediate	
--------	---------	-------	-----------	--

功能: if(rs>0) $pc \leftarrow pc + 4 + (\text{sign-extend})immediate \ll 2$ else $pc \leftarrow pc + 4$

==>跳转指令

(15) j addr

111000	addr[27:2]			
--------	------------	--	--	--

功能: $pc \leftarrow \{(pc+4)[31:28], \text{addr}[27:2], 0, 0\}$, 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。

(16) jr rs

111001	rs(5 位)	未用	未用	Reserved
--------	---------	----	----	----------

功能: $pc \leftarrow rs$, 跳转。

==>调用子程序指令

(17) jal addr

111010	addr[27..2]
--------	-------------

功能：调用子程序， $pc \leftarrow \{(pc+4)[31:28], addr[27:2], 0, 0\}$ ； $\$31 \leftarrow pc+4$ ，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

(18) halt (停机指令)

111111	00000000000000000000000000000000(26 位)
--------	----------------------------------------

不改变 pc 的值，pc 保持不变。

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。

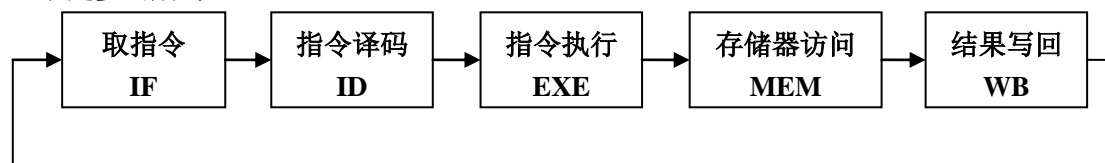


图 1 多周期 CPU 指令处理过程

MIPS 指令的三种格式：

R 类型:

31	26 25	21 20	16 15	11 10	6 5	0
op	rs	rt	rd	sa	funct	
6 位	5 位	5 位	5 位	5 位	6 位	

I 类型:

31	26 25	21 20	16 15	0
op	rs	rt	immediate	
6 位	5 位	5 位	16 位	

J 类型:

31	26 25	0
op	address	
6 位	26 位	

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。

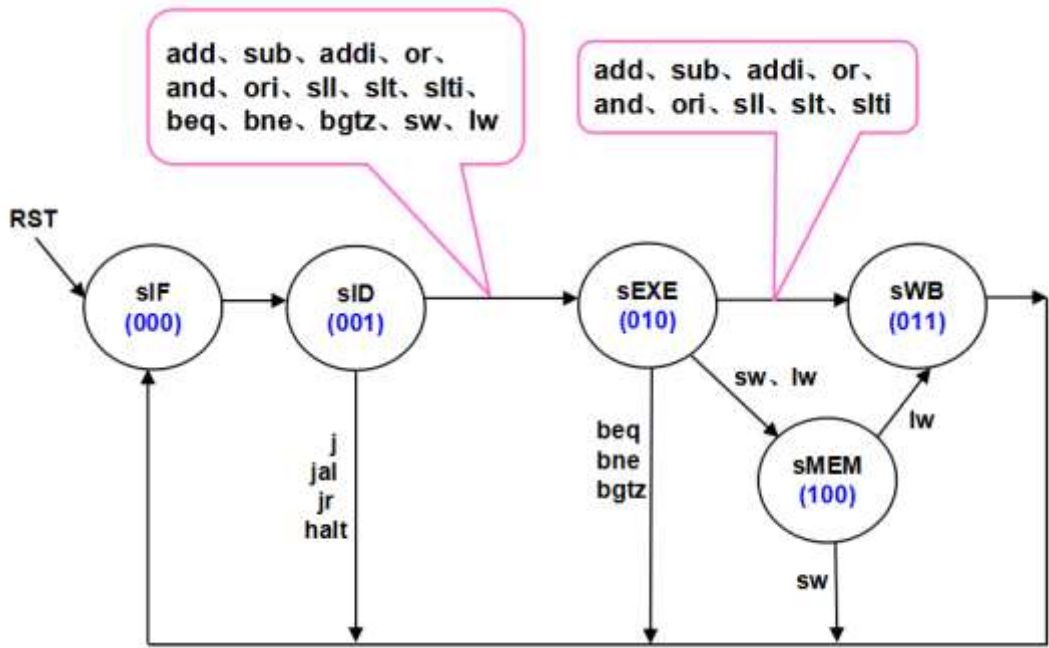


图 2 多周期 CPU 状态转移图

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。

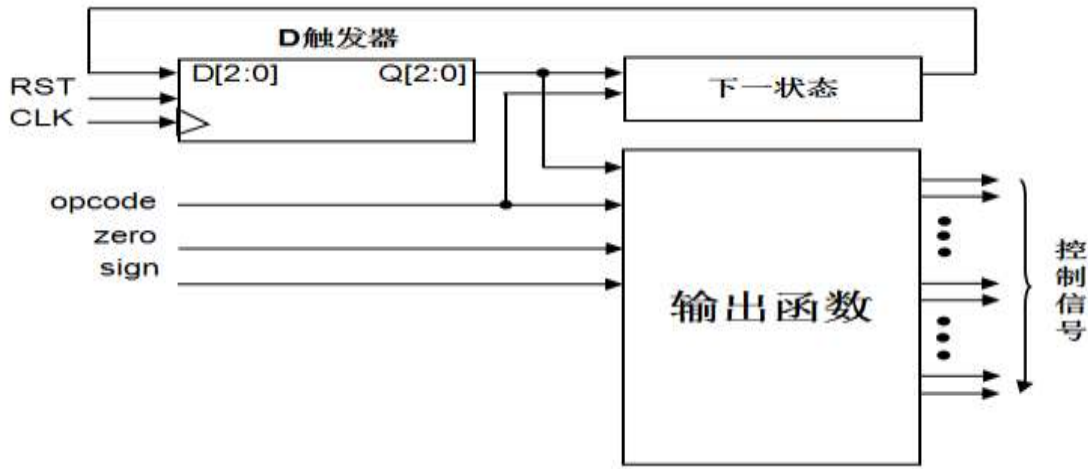


图 3 多周期 CPU 控制部件的原理结构图

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。

	令: add、sub、or、and、beq、bne、bgtz、slt、sll	指令: addi、ori、slti、lw、sw
DBDataSrc	来自 ALU 运算结果的输出,相关指令: add、sub、addi、or、and、ori、slt、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、bgtz、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、slti、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、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
/RD	读数据存储器, 相关指令: lw	存储器输出高阻态
/WR	写数据存储器, 相关指令: sw	无操作
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;	(sign-extend) immediate , 相关指令: addi、lw、sw、beq、bne、bgtz;
PCSrc[1..0]	00: $pc \leftarrow pc+4$, 相关指令: add、addi、sub、or、ori、and、slt、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bgtz(sign=1, 或 zero=1); 01: $pc \leftarrow pc+4+(sign-extend)immediate$, 相关指令: beq(zero=1)、bne(zero=0)、bgtz(sign=0, 且 zero=0); 10: $pc \leftarrow rs$, 相关指令: jr; 11: $pc \leftarrow \{pc[31:28], addr[27:2], 0, 0\}$, 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 ($\$31 \leftarrow pc+4$) ; 01: rt 字段, 相关指令: addi、ori、slti、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	if (A < B && (A[31] == B[31])) Y = 1; else if (A[31] == 1 && B[31] == 0) Y = 1; else Y = 0;	比较 A 与 B 带符号
100	$Y = B \ll A$	B 左移 A 位
101	$Y = A \vee B$	或
110	$Y = A \wedge B$	与
111	$Y = A \oplus B$	异或

四. 实验器材

电脑一台, Xilinx Vivado 软件一套, Basys3板一块。

五. 仿真多周期CPU实验过程与结果

A.设计思路

多周期 CPU 分为五个阶段（每个阶段为一个时钟周期），分别为取指令(IF)，指令译码(ID)，指令执行(EXE)，存储器访问(MEM)，结果写回(WB)，每个阶段进行的操作参见上

述实验原理，不同的指令进行的周期数不同。

相比单周期 CPU，多周期 CPU 存在数据延迟问题。为使数据稳定，新增加了 IR、ADR、BDR、ALUoutDR、DBDR 五个数据延迟单元，分别对 32 位指令、根据指令在寄存器组读出的第一个数据、读出的第一个数据、ALU 单元运算结果、将要写入寄存器组的数据进行数据延迟。

实验基本思路为，在每个时钟周期的上升沿，控制单元根据输入信号输出对不同单元的相应控制信号，并在该时钟周期的下降沿，不同单元执行对应的操作。即控制单元和 PC 单元由上升沿触发，而 IR、ADR、BDR、ALUoutDR、DBDR 均由下降沿触发。但此时会产生数据冲突——在 IF 取指令阶段的上一阶段，PCWre 为 0，则在下一个周期（即 IF）的上升沿时 PC 不能及时改变导致错误。

为解决上述问题，即在 IF 取指令阶段上升沿到来前，PCWre 必须置 1 使能。做出改动——在每条指令的最后一个周期，令 PCWre 置 1 使能，使得在下一个周期的上升沿时 PC 可以及时改变。

综上，相对单周期 CPU 所做调整为：控制单元与 PC 单元由上升沿触发，而 IR、ADR、BDR、ALUoutDR、DBDR 均由下降沿触发。在每条指令的最后一个周期，令 PCWre 置 1 使能。

此时，各类指令执行情况如下（数字为周期数）：

算术、逻辑、移位、比较运算指令（4 个周期）

1. 上升沿时，PCWre 为 1，得到当前 PC，并得到对应指令存入 IR；下降沿时，IR 输出指令。
2. 上升沿时，控制单元根据指令和状态给出各单元控制信号；下降沿时，ADR、BDR 输出寄存器组中读出的两个数据，并在 ALU 单元进行计算。
3. 上升沿时，控制单元根据指令和状态给出各单元控制信号；下降沿时，ALUoutDR 和 DBDR 输出 ALU 运算结果。
4. 上升沿时，控制单元根据指令和状态给出各单元控制信号，RegWre 为 1 使能；下降沿时，将数据写入寄存器。

存储器写指令（sw，4 个周期）

1. 上升沿时，PCWre 为 1，得到当前 PC，并得到对应指令存入 IR；下降沿时，IR 输出指令。
2. 上升沿时，控制单元根据指令和状态给出各单元控制信号；下降沿时，ADR、BDR 输出寄存器组中读出的两个数据，并在 ALU 单元进行计算地址。
3. 上升沿时，控制单元根据指令和状态给出各单元控制信号；下降沿时，ALUoutDR 和输出 ALU 运算结果，即操作数地址。
4. 上升沿时，控制单元给出数据存储器控制信号，WR 为 0 使能，将数据写入对应存储器单元。

存储器读指令（lw，5 个周期）

1. 上升沿时，PCWre 为 1，得到当前 PC，并得到对应指令存入 IR；下降沿时，IR 输出指令。
2. 上升沿时，控制单元根据指令和状态给出各单元控制信号；下降沿时，ADR、BDR 输出寄存器组中读出的两个数据，并在 ALU 单元进行计算地址。
3. 上升沿时，控制单元根据指令和状态给出各单元控制信号；下降沿时，ALUoutDR

和输出 ALU 运算结果，即操作数地址。

4. 上升沿时，控制单元给出数据存储器控制信号，RD 为 0 使能，输出数据；下降沿时，DBDR 输出将要写入寄存器的数据。
5. 上升沿时，控制单元根据指令和状态给出各单元控制信号，RegWre 为 1 使能；下降沿时，将数据写入寄存器。

jump 指令（包括调用子程序指令，2 个周期）

1. 上升沿时，PCWre 为 1，得到当前 PC，并得到对应指令存入 IR；下降沿时，IR 输出指令，读出第一个数据（jr 指令时为将要跳转的指令地址）。
2. 上升沿时，控制单元根据指令和状态给出对应 PCSrc 信号，调整下一条指令的地址。

branch 指令（假设需分支跳转，3 个周期）

1. 上升沿时，PCWre 为 1，得到当前 PC，并得到对应指令存入 IR；下降沿时，IR 输出指令，读出待比较的两个数据。
2. 上升沿时，控制单元根据指令和状态给出各单元控制信号；下降沿时，ADR、BDR 输出寄存器组中读出的两个数据，并在 ALU 单元进行比较。
3. 上升沿时，控制单元根据 sign 和 zero 输出 PCSrc 信号，调整下一条指令的地址。

B.代码实现

根据上述实验原理及实验分析，各模块代码如下：

1.控制单元模块 ControlUnit

```
module ControlUnit(input clk, Reset, zero, sign, input [5:0] op, output reg PCWre, ExtSel,
InsMemRW, RegWre, ALUSrcA, ALUSrcB, RD, WR, DBDataSrc, WrRegDSrc, IRWre, output reg
[1:0] PCSrc, RegDst, output reg [2:0] ALUOp);
    reg [2:0] state, nextstate;
    initial begin
        state = 0;
        nextstate = 0;
        InsMemRW = 0;
    end
    //上升沿时状态跳转
    always@ (posedge clk or negedge Reset)begin
        if (Reset == 0) state = 0;
        else begin
            state = nextstate;
            case(state)
                3'b000:begin//IF
                    nextstate = 3'b001; //下一状态为 ID
                end
                3'b001:begin//ID
                    if (op[5:3] == 3'b111) nextstate = 3'b000; //J 指令或 halt, 下一状态为 IF
                    else nextstate = 3'b010; //其他, 下一状态为 EXE
                end
            endcase
        end
    end
```

```

        end
        3'b010:begin//EXE
            if (op[5:2] == 4'b1100) nextstate = 3'b100; //sw,lw,下一状态为 MEM
            else if (op[5:2] == 4'b1101) nextstate = 3'b000; //branch,下一状态为 IF
            else nextstate = 3'b011; //其他, 下一状态为 WB
        end
        3'b011:begin//WB
            nextstate = 3'b000; //下一状态为 IF
        end
        3'b100:begin//MEM
            if (op == 6'b110000) nextstate = 3'b000; //sw,下一状态为 IF
            else nextstate = 3'b011; //lw,下一状态为 WB
        end
    endcase
end

//根据当前状态和 op 设置各控制信号
always@ (*)begin
    PCWre = (op != 6'b111111 && Reset == 1 && nextstate == 3'b000);
    ExtSel = !(op == 6'b010010 || op == 6'b011000);
    InsMemRW = 1;
    RegWre = (op == 6'b111010 && state == 3'b001) || (state == 3'b011);
    ALUSrcA = (op == 6'b011000); //sll
    ALUSrcB = (op == 6'b010010 || op == 6'b000010 || op == 6'b100111 || op == 6'b110000 ||
op == 6'b110001); //ori addi slti, sw lw
    RD = (op != 6'b110001 || state != 3'b100); //lw 时为 0
    WR = (op != 6'b110000 || state != 3'b100); //sw 时为 0
    DBDataSrc = (op == 6'b110001); //lw
    WrRegDSrc = !(op == 6'b111010); //jal
    IRWre = (state == 3'b000); //IF 可改
    //PCSrc 信号
    if (op == 6'b111001 && state == 3'b001) PCSrc = 2'b10;
    else if (op == 6'b110100 && zero == 1) PCSrc = 2'b01;
    else if (op == 6'b110101 && zero == 0) PCSrc = 2'b01;
    else if ((op == 6'b111000 || op == 6'b111010) && state = 3'b001) PCSrc = 2'b11;
    else PCSrc = 2'b00;

    if (op == 6'b111010) RegDst = 2'b00;
    else if (op == 6'b010010 || op == 6'b000010 || op == 6'b100111 || op == 6'b110001) RegDst
= 2'b01; //ori addi slti, lw
    else RegDst = 2'b10;

    if (op == 6'b000000 || op == 6'b000010 || op[5:1] == 5'b11000) ALUOp = 3'b000; //add addi
sw lw

```

```

        else if (op[5:2] == 4'b1101 || op == 6'b000001) ALUOp = 3'b001; //branch sub
        else if (op == 6'b010010 || op == 6'b010000) ALUOp = 3'b011; //ori or
        else if (op == 6'b010001) ALUOp = 3'b100; //and
        else if (op == 6'b011000) ALUOp = 3'b010; //sll
        else if (op == 6'b100110 || op == 6'b100111) ALUOp = 3'b110; //slt,slti
        else ALUOp = 3'b111;
    end
endmodule

```

2. PC单元 PCUnit

//1.PCWre 判断是否停机 2.curPC 为当前指令地址, $PC4 = PC + 4$ 3.上升沿触发

```

module PCUnit(input clk, Reset, PCWre, input [31:0] nextPC, output reg [31:0] curPC, PC4);
    initial begin
        PC4 = 32'H00000000;
        curPC = 32'H00000000;
    end
    always@(posedge clk or negedge Reset) begin
        //always@ (PCWre or Reset) begin
            if (Reset == 1'b0) begin
                curPC = 32'h00000000;
                PC4 = 32'h00000000;
            end
            else if (PCWre == 1'b1) begin
                curPC = nextPC;
                PC4 = curPC + 4;
            end
        end
    end
endmodule

```

3. 计算跳转地址单元JumpAdd

```

module JumpAdd(input [31:0] PC4, input [25:0] Instruction, output [31:0] JumpAdd);
    assign JumpAdd = {PC4[31:28], Instruction, 2'b00};
endmodule

```

4. PC选择单元PcChoose

//根据控制信号 PCSrc 选择下一指令的地址

```

module PcChoose(input [1:0] PCSrc, input [31:0] PC4, Extend_Imme, input [31:0] JumpAdd,
    JumpRegister, output reg [31:0] PCNext);
    initial begin
        PCNext = 0;
    end
    always @(*)begin

```

```

    case(PCSrc)
        2'b00: PCnext <= PC4;
        2'b01: PCnext <= PC4 + Extend_Imme * 4;
        2'b10: PCnext <= JumpRegister;
        2'b11: PCnext <= JumpAdd;
        default: PCnext <= 32'h00000000;
    endcase
end
endmodule

```

5. 指令存储器单元InstructMem

//根据信号及地址读出指令

```

module InstructMem (input clk, InsMemRW, IRWre, input [31:0] InsAddr, output reg [31:0]
Instruction);
    reg [7:0] ROM [0:99];
    reg [31:0] TemplInstruct;
    initial begin
        $readmemb("F:/rom.txt", ROM);
    end
    always@(InsMemRW or InsAddr)begin
        if (InsMemRW == 1 && IRWre == 1) TemplInstruct = {{ROM[InsAddr]}, {ROM[InsAddr +
1]}, {ROM[InsAddr + 2]}, {ROM[InsAddr + 3]}};
    end
    //IR 寄存器
    always@(negedge clk)begin
        Instruction = TemplInstruct;
    end
endmodule

```

6. 寄存器组RegisterFile

//下降沿触发，RegWre 为 1 时写使能

```

module RegisterFile(input clk, RST, RegWre, input [4:0] read1, read2, write, input [31:0] writeData,
output [31:0] readData1, readData2);
    reg [31:0] REGS[1:31];
    assign readData1 = (read1 == 0)? 0: REGS[read1];
    assign readData2 = (read2 == 0)? 0: REGS[read2];
    integer i;
    always@(negedge clk or negedge RST)begin
        if (RST == 0)begin
            for (i = 1; i < 32; i = i + 1)
                REGS[i] <= 0;
        end
        else if (RegWre == 1 && write != 0)

```

```

        REGS[write] <= writeData;
    end
endmodule

```

7.写寄存器选择单元chooseWr

//选择写寄存器

```

module chooseWr(input [1:0]RegDst, input [4:0] rt, rd, output reg [4:0] write);
    always@(*)begin
        if (RegDst == 2'b00) write = 31;
        else if (RegDst == 2'b01) write = rt;
        else write = rd;
    end
endmodule

```

8.ALU单元

//根据 ALUOp 运算

```

module ALU(input [2:0] ALUOp, input [31:0] A, B, output sign, zero, output reg [31:0] result);
    assign zero = (result == 0);
    assign sign = result[31];
    always@(ALUOp or A or B)begin
        case(ALUOp)
            3'b000: result = A + B;
            3'b001: result = A - B;
            3'b010: result = B << A;
            3'b011: result = A | B;
            3'b100: result = A & B;
            3'b101: result = A < B;
            3'b110: begin //小于为 1
                if (A < B && A[31] == B[31])result = 1;
                else if (A[31] && !B[31]) result = 1;
                else result = 0;
            end
            3'b111: result = A ^ B;
        endcase
    end
endmodule

```

9.二选一单元choose32

//选择 ALU 操作数等

```

module choose32(input signal, input [31:0] A, B, output[31:0] C);
    assign C = (signal == 1)? A: B;
endmodule

```

10. 数据存储器单元DataMem

//电平触发, RD,WR 低电平有效

```
module DataMem(input RD, WR, input [31:0] dataAddr, dataIn, output [31:0] dataOut);
    reg[7:0] RAM[0: 60];
    assign dataOut = (RD == 0)? {RAM[dataAddr],RAM[dataAddr + 1],RAM[dataAddr +
2],RAM[dataAddr + 3]} : 32'h00000000;
    always@(*)begin
        if (WR == 0) {RAM[dataAddr],RAM[dataAddr + 1],RAM[dataAddr + 2],RAM[dataAddr + 3]} =
dataIn;
    end
endmodule
```

11. 扩展单元 ExtendUnit

//零扩展或符号扩展

```
module ExtendUnit(input ExtSel, input [15:0] immediate, output [31:0] Extend_Imme);
    assign Extend_Imme = (ExtSel == 0)?{16'h0000,immediate}:{16{immediate[15]}},immediate);
endmodule
```

12. 数据延迟单元DR

//数据延迟单元, 下降沿触发, 包括 ADR,BDR,ALUoutDR,DBDR

```
module DR(input clk, input [31:0] dataIn, output reg [31:0] dataOut);
    always @(negedge clk)begin
        dataOut = dataIn;
    end
endmodule
```


C.实验结果

根据上述代码，编写顶层模块TOP.

编写仿真程序进行仿真，测试程序如下：

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	=	08010008
0x00000004	ori \$2,\$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002
0x00000008	or \$3,\$2,\$1	010000	00010	00001	00011 000000000000	=	40411800
0x0000000C	sub \$4,\$3,\$1	000001	00011	00001	00100 000000000000	=	04612000
0x00000010	and \$5,\$4,\$2	010001	00100	00010	00101 000000000000	=	44822800
0x00000014	sll \$5,\$5,2	011000	00000	00101	00101 00010 000000	=	60052880
0x00000018	beq \$5,\$1,-2(=,转 14)	110100	00101	00001	1111 1111 1111 1110	=	D0A1FFFE
0x0000001C	jal 0x0000048	111010	00000	00000	0000 0000 0001 0010	=	E8000012
0x00000020	slt \$8,\$12,\$1	100110	01100	00001	01000 000000000000	=	99814000
0x00000024	addi \$14,\$0,-2	000010	00000	01110	1111 1111 1111 1110	=	080EFFFF
0x00000028	slt \$9,\$8,\$14	100110	01000	01110	01001 000000000000	=	990E4800
0x0000002C	slti \$10,\$9,2	100111	01001	01010	0000 0000 0000 0010	=	9D2A0002
0x00000030	slti \$11,\$10,0	100111	01010	01011	0000 0000 0000 0000	=	9D4B0000
0x00000034	add \$11,\$11,\$8	000000	01011	01000	01011 000000000000	=	01685800
0x00000038	bne \$11,\$2,-2 (≠,转 34)	110101	01011	00010	1111 1111 1111 1110	=	D562FFFE
0x0000003C	addi \$2,\$2,-1	000010	00010	00010	1111 1111 1111 1111	=	0842FFFF
0x00000040	bgtz \$2,-2 (>0,转 3C)	110110	00010	00000	1111 1111 1111 1110	=	D840FFFE
0x00000044	j 0x0000054	111000	00000	00000	0000 0000 0001 0101	=	E0000015
0x00000048	sw \$2,4(\$1)	110000	00001	00010	0000 0000 0000 0100	=	C0220004
0x0000004C	lw \$12,4(\$1)	110001	00001	01100	0000 0000 0000 0100	=	C42C0004
0x00000050	jr \$31	111001	11111	00000	0000 0000 0000 0000	=	E7E00000
0x00000054	halt	111111	00000	00000	0000 0000 0000 0000	=	FC000000

(curPC 为当前指令地址, nextPC 是下一指令地址,state 为当前状态)

addi \$1,\$0,8

state = 0 时, 上升沿时 PC 为第一条指令地址, 下降沿 IR 传出指令, 此时 PC 由 Reset 初始化, 不改变

state = 1 时, 寄存器组读出数据输出 (\$0 = 0, 立即数为 8)

state = 2 时, DBDR 输出写入寄存器的数据 (Dbout = result = 8)

state = 3 时, 可以看到 REG[1]在下降沿被写入, 结果为 8 (\$1 = 8)

nextPC[31:0]	00000004	X			00000004				X
curPC[31:0]	00000000				00000000				X
Instruction[31:0]	08010008				08010008				
write[4:0]	01				01				
ALUOp[2:0]	0				0				
state[2:0]	3	0	X	1	X	2	X	3	X
readData[31:0]	00000000								00000000
readData2[31:0]	00000008				00000000			X	00000008
readDataout1[31:0]	00000000								00000000
readDataout2[31:0]	00000000				00000000				
Extend_Imme[31:0]	00000008				00000008				
A[31:0]	00000000								00000000
B[31:0]	00000008				00000008				
result[31:0]	00000008				00000008				
ALUout[31:0]	00000008					00000008			
dataOut[31:0]	00000000								00000000
DBdata[31:0]	00000008				00000008				
DBout[31:0]	00000008					00000008			
writeData[31:0]	00000008					00000008			
REGS[1][31:0]	00000008				00000000			X	

ori \$2,\$0,2

state = 0 时, 上升沿时 PC 修改, 下降沿时 IR 传出指令, 读出立即数 2

state = 1 时, 下降沿时, 寄存器组读出数据输出 (\$0 = 0 参与运算, result = 2)

state = 2 时, DBDR 输出写入寄存器的数据 (2)

state = 3 时, 可以看到 REG[2]在下降沿被写入 (\$2 = 2)

nextPC[31:0]	00000008	00	X		00000008		X				
curPC[31:0]	00000004	00	X		00000004		X				
Instruction[31:0]	48020002	08010008	X		48020002						
write[4:0]	02	01	X		02						
ALUOp[2:0]	3	0	X				3				
state[2:0]	3	3	X	0	X	1	X	2	X	3	X
readData1[31:0]	00000000				00000000						
readData2[31:0]	00000002	00000008	X		00000000				00000002		
readDataout1[31:0]	00000000				00000000						
readDataout2[31:0]	00000000	00000000	X	00000008	X		00000000				
Extend_Imme[31:0]	00000002	00000008	X		00000002						
A[31:0]	00000000				00000000						
B[31:0]	00000002	00000008	X				00000002				
result[31:0]	00000002	00000008	X				00000002				
ALUout[31:0]	00000002	00000008	X				00000002				
dataOut[31:0]	00000000	00000008	X						00000000		
DBdata[31:0]	00000002	00000008	X				00000002				
DBout[31:0]	00000002	00000008	X				00000002				
writeData[31:0]	00000002	00000008	X				00000002				
REGS[2][31:0]	00000002				00000000						

or \$3,\$2,\$1

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组读出数据输出 (\$2 = 2, \$1 = 8)

state = 2 时，下降沿时，DBDR 输出写入寄存器的数据 (DBout = result = 10)

state = 3 时，可以看到 REG[3]在下降沿被写入 (\$3 = 10)

nextPC[31:0]	0000000c	00000		0000000c		
curPC[31:0]	00000008	00000		00000008		
Instruction[31:0]	40411800	48020002		40411800		
write[4:0]	03	02		03		
ALUOp[2:0]	3			3		
state[2:0]	3	3	0	1	2	3
readData1[31:0]	00000002	00000000		00000002		
readData2[31:0]	00000008	00000002				00000000
readDataout1[31:0]	00000002	00000000		00000002		
readDataout2[31:0]	00000008	00000000	00000002			
Extend_Imme[31:0]	00001800	00000002		00001800		
A[31:0]	00000002	00000000		00000002		
B[31:0]	00000008	00000002				
result[31:0]	0000000a	00000002		0000000a		
ALUout[31:0]	0000000a		00000002			0000000a
dataOut[31:0]	00000000				00000000	
DBdata[31:0]	0000000a	00000002		0000000a		
DBout[31:0]	0000000a		00000002			0000000a
writeData[31:0]	0000000a		00000002			0000000a
REGS[3][31:0]	0000000a		00000000			

sub \$4,\$3,\$1

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组读出数据输出 (\$3 = 10, \$1 = 8)

state = 2 时，下降沿时，DBDR 输出写入寄存器的数据 (DBout = result = 2)

state = 3 时，可以看到 REG[4]在下降沿被写入，结果为 (\$4 = 10)

nextPC[31:0]	00000010	0		00000010		
curPC[31:0]	0000000c	0		0000000c		
Instruction[31:0]	04612000	40411800		04612000		
write[4:0]	04	03		04		
ALUOp[2:0]	1	3		1		
state[2:0]	3	3	0	1	2	3
readData1[31:0]	0000000a	00000002		0000000a		
readData2[31:0]	00000008			00000008		
readDataout1[31:0]	0000000a	00000002		0000000a		
readDataout2[31:0]	00000008			00000008		
Extend_Imme[31:0]	00002000	00001800		00002000		
A[31:0]	0000000a	00000002		0000000a		
B[31:0]	00000008			00000008		
result[31:0]	00000002	0000000a	fffffffa	00000002		
ALUout[31:0]	00000002	0000000a	fffffffa			00000002
dataOut[31:0]	00000000					00000000
DBdata[31:0]	00000002	0000000a	fffffffa	00000002		
DBout[31:0]	00000002	0000000a	fffffffa			00000002
writeData[31:0]	00000002	0000000a	fffffffa			00000002
REGS[4][31:0]	00000002		00000000			

and \$5,\$4,\$2

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组读出数据输出 (\$4 = 2, \$2 = 2)

state = 2 时，下降沿时，DBDR 输出写入寄存器的数据 (DBout = result = 2)

state = 3 时，可以看到 REG[3]在下降沿被写入 (\$5 = 2)

nextPC[31:0]	00000014	0		00000014		
curPC[31:0]	00000010	0		00000010		
Instruction[31:0]	44822800	04612000		44822800		
write[4:0]	05	04				05
ALUOp[2:0]	4	1		4		
state[2:0]	3	3	0	1	2	3
readData1[31:0]	00000002	0000000a		00000002		
readData2[31:0]	00000002	00000008				00000002
readDataout1[31:0]	00000002	0000000a				00000002
readDataout2[31:0]	00000002	00000008				
Extend_Imme[31:0]	00002800	00002000		00002800		
A[31:0]	00000002	0000000a				
B[31:0]	00000002	00000008				
result[31:0]	00000002	00000002	00000008	00000002		
ALUout[31:0]	00000002	00000002	00000008			00000002
dataOut[31:0]	00000000					00000000
DBdata[31:0]	00000002	00000002	00000008	00000002		
DBout[31:0]	00000002	00000002	00000008			00000002
writeData[31:0]	00000002	00000002	00000008			00000002
REGS[5][31:0]	00000002		00000000			

sll \$5,\$5,2

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组和扩展单元输出运算数 (\$5 = 2, sa = 2)

state = 2 时，下降沿时，DBDR 输出写入寄存器的数据 (DBout = result = 8)

state = 3 时，可以看到 REG[5]在下降沿被写入，结果为 (\$5 = 8)

nextPC[31:0]	00000018	00		00000018		
curPC[31:0]	00000014	00		00000014		
Instruction[31:0]	60052880	44822800		60052880		
write[4:0]	05			05		
ALUOp[2:0]	2	4		2		
state[2:0]	3	3	0	1	2	3
readData1[31:0]	00000000	00000002		00000000		
readData2[31:0]	00000008			00000002		
readDataout1[31:0]	00000000	00000002				00000000
readDataout2[31:0]	00000002			00000002		
Extend_Imme[31:0]	00002880	00002800		00002880		
A[31:0]	00000002			00000002		
B[31:0]	00000002			00000002		
result[31:0]	00000008	00000002		00000008		
ALUout[31:0]	00000008	00000002				00000008
dataOut[31:0]	00000000					00000000
DBdata[31:0]	00000008	00000002		00000008		
DBout[31:0]	00000008	00000002				00000008
writeData[31:0]	00000008	00000002				00000008
REGS[5][31:0]	00000008		00000002			

beq \$5,\$1,-2(=,转 14)

state = 0 时, 上升沿时 PC 修改, 此时 nextPC 为 $PC + 4 = 1CH$, 下降沿时 IR 传出指令, 计算得 zero = 1, nextPC 为 14H

state = 1 时, 下降沿时, 寄存器组读出数据输出 ($\$5 = 8, \$1 = 8$)

state = 2 时, 下降沿时, ALUout 输出 ALU 结果 (ALUout = result = 0)

此时, zero = 1, PCSrc = 1, 分支跳转, 可以看到 nextPC = 14H, 下一状态为 0

nextPC[31:0]	00000014	0000001c	00000014	0000
curPC[31:0]	00000018	00000018		
PCSrc[1:0]	1	0	1	
zero	1			
Instruction[31:0]	d0a1ffffe	60052880	d0a1ffffe	
write[4:0]	1f	05	1f	
ALUOp[2:0]	1	2	1	
state[2:0]	2	3	2	0
readData1[31:0]	00000008	00000000	00000008	
readData2[31:0]	00000008		00000008	
readDataout1[31:0]	00000008	00000000	00000008	
readDataout2[31:0]	00000008	00000002		00
Extend_Imme[31:0]	fffffffe	00002880	fffffffe	
A[31:0]	00000008	00000002	00000008	
B[31:0]	00000008	00000002		00
result[31:0]	00000000	00000008	ffffff8	00000000
ALUout[31:0]	00000000	00000008	ffffff8	00000000

sll \$5,\$5,2

state = 0 时, 上升沿时 PC 修改, 下降沿时 IR 传出指令

state = 1 时, 下降沿时, 寄存器组和扩展单元输出运算数 ($\$5 = 8, sa = 2$)

state = 2 时, 下降沿时, DBDR 输出写入寄存器的数据 (DBout = result = 20H)

state = 3 时, 可以看到 REG[5]在下降沿被写入, 结果为 ($\$5 = 20H$)

nextPC[31:0]	00000018	000	0000	00000018	
curPC[31:0]	00000014	000		00000014	
Instruction[31:0]	60052880	d0a1ffffe		60052880	
write[4:0]	05	1f		05	
ALUOp[2:0]	2	1		2	
state[2:0]	3	2	0	1	2
readData1[31:0]	00000000	00000008		00000000	
readData2[31:0]	00000020		00000008		00000020
readDataout1[31:0]	00000000	00000008		00000000	
readDataout2[31:0]	00000008		00000008		
Extend_Imme[31:0]	00002880	fffffffe		00002880	
A[31:0]	00000002	00000008		00000002	
B[31:0]	00000008		00000008		
result[31:0]	00000020	00000000		00000020	
ALUout[31:0]	00000020	00000000		00000020	
dataOut[31:0]	00000000			00000000	
DBdata[31:0]	00000020	00000000		00000020	
DBout[31:0]	00000020	00000000		00000020	
writeData[31:0]	00000020	00000000		00000020	
REGS[5][31:0]	00000020		00000008		

beq \$5,\$1,-2(=,转 14)

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组读出数据输出 (\$5 = 20H, \$1 = 8)

state = 2 时，下降沿时，ALUout 输出 ALU 结果 (ALUout = result = 18H)

此时，zero = 0, PCSrc = 0, 不分支，可以看到 nextPC = 1CH, 下一状态为 0

nextPC[31:0]	0000001c	0000001c	0000
curPC[31:0]	00000018	00000018	00
Instruction[31:0]	d0a1fffe	60052880	d0a1fffe
write[4:0]	1f	05	1f
PCSrc[1:0]	0	0	
zero	0		
ALUOp[2:0]	1	2	1
state[2:0]	2	3	0
readData1[31:0]	00000020	00000000	00000020
readData2[31:0]	00000008	00000020	00000008
readDataout1[31:0]	00000020	00000000	00000020
readDataout2[31:0]	00000008	00000020	00000008
Extend_Imme[31:0]	fffffffe	00002880	fffffffe
A[31:0]	00000020	00000002	00000000
B[31:0]	00000008	00000008	00000020
result[31:0]	00000018	00000020	ffffffe0
ALUout[31:0]	00000018	00000020	ffffffe0
dataOut[31:0]	00000000		00000018

jal 0x0000048

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令，计算 JumpAdd = 48H

state = 1 时，上升沿给出 PCSrc = 3, nextPC 修改为 48H

可以看到下一状态为 0

nextPC[31:0]	00000048	00000020	00000048
curPC[31:0]	0000001c	0000001c	
Instruction[31:0]	e8000012	d0a1fffe	e8000012
PCSrc[1:0]	3	0	3
state[2:0]	1	2	0
readData1[31:0]	00000000	00000020	00000000
readData2[31:0]	00000000	00000008	00000000
readDataout1[31:0]	00000000	00000020	00000000
readDataout2[31:0]	00000000	00000008	00000000
JumpAdd[31:0]	00000048	0287fff8	00000048
Extend_Imme[31:0]	00000012	fffffffe	00000012
A[31:0]	00000000	00000020	00000000
B[31:0]	00000000	00000008	00000000
result[31:0]	00000000	00000018	00000028
ALUout[31:0]	00000028	00000018	00000028

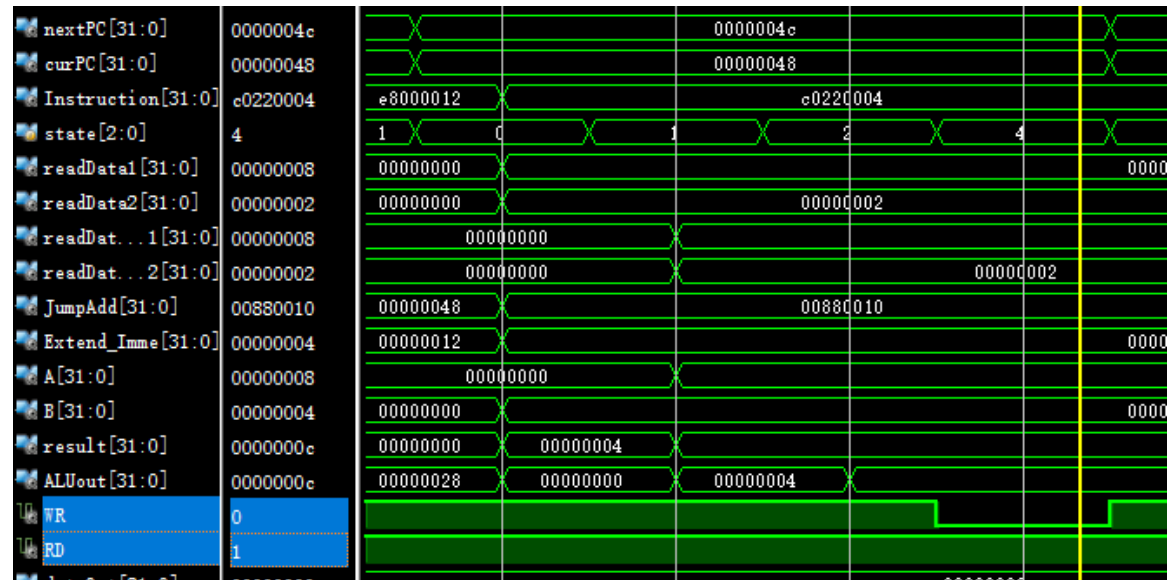
sw \$2,4(\$1)

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组读出数据输出 (\$1 = 8, \$2 = 2)

state = 2 时，下降沿时，ALUout 输出计算结果即存储器地址 (ALUout = result = 12)

state = 4 时，上升沿给出电平信号 WR/RD，存储器写使能，写入存储器



lw \$12,4(\$1)

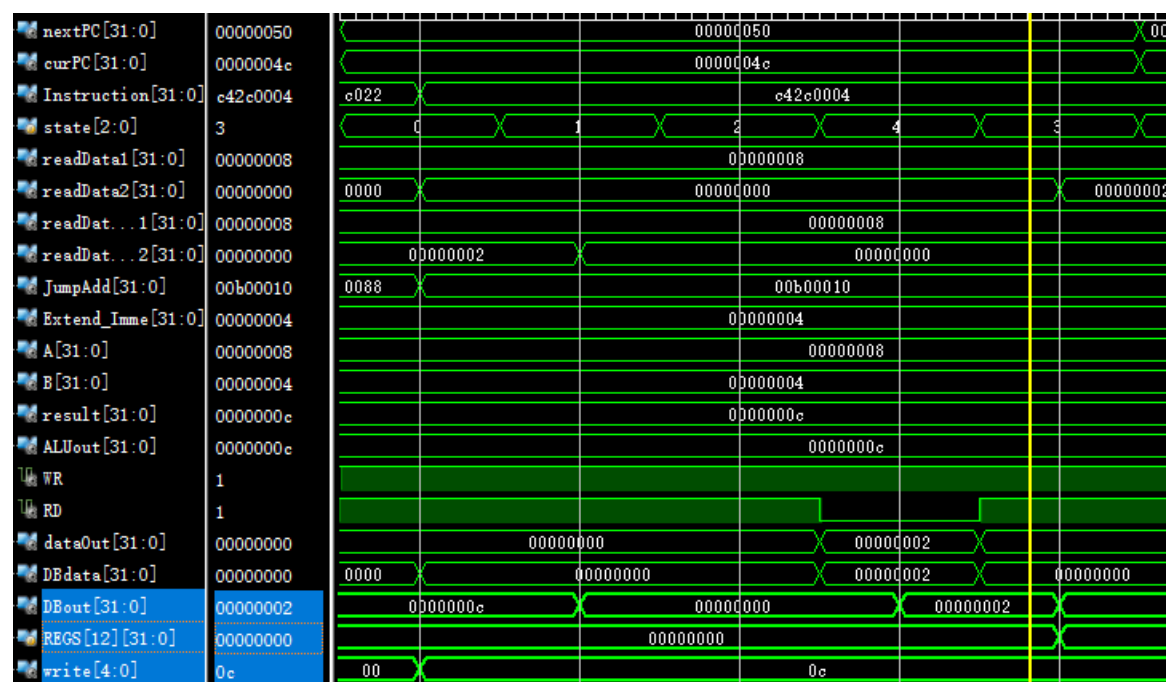
state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组和扩展单元输出运算数 (\$1 = 8, \$12 = 0)

state = 2 时，下降沿时，ALUout 输出计算结果即存储器地址 (ALUout = result = 12)

state = 4 时，上升沿给出 WR/RD 控制信号，下降沿 DBDR 修改为所读出存储器的值

state = 3 时，可以看到 REG[12]在下降沿被写入，结果为 (write = 12, \$12 = 2)




```
jr $31
```

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，上升沿给出 PCSrc = 2, nextPC = readData1 = 20H

可以看到下一状态为 0

nextPC[31:0]	00000020	000	00000054	00000020
curPC[31:0]	00000050	000	00000050	
Instruction[31:0]	e7e00000	c42c0004	e7e00000	
state[2:0]	1	3	0	1
PCSrc[1:0]	2	0	2	
readData1[31:0]	00000020	00000008	00000020	
readData2[31:0]	00000000	00000002	00000000	
readDataout1[31:0]	00000020	00000008		00000000
readDataout2[31:0]	00000000	00000002		00000000
JumpAdd[31:0]	0f800000	00b00010	0f800000	
Extend_Imme[31:0]	00000000	00000004	00000000	

```
slt $8,$12,$1
```

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组读出数据输出 (\$12 = 2, \$1 = 8)

state = 2 时，下降沿时，DBDR 输出写入寄存器的数据 (DBout = result = 1)

state = 3 时，可以看到 REG[8]在下降沿被写入 (\$8 = 1)

nextPC[31:0]	00000024	X	00000024	
curPC[31:0]	00000020	X	00000020	
Instruction[31:0]	99814000	e7e0	99814000	
state[2:0]	3	X	0	1
readData1[31:0]	00000002	00000	00000002	
readData2[31:0]	00000008	00000	00000008	
readDataout1[31:0]	00000002	00000020	00000002	
readDataout2[31:0]	00000008	00000000	00000008	
JumpAdd[31:0]	06050000	0f800	06050000	
Extend_Imme[31:0]	00004000	00000	00004000	
A[31:0]	00000002	00000020	00000002	
B[31:0]	00000008	00000000	00000008	
result[31:0]	00000001	00000	00000001	
ALUout[31:0]	00000001	00000	00000020	00000000
dataOut[31:0]	00000000			00000000
DBdata[31:0]	00000001	00000	00000000	00000001
DBout[31:0]	00000001	00000	00000020	00000000
writeData[31:0]	00000001	00000	00000020	00000000
REGS[8][31:0]	00000001		00000000	

addi \$14,\$0,-2

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令，读出立即数 2

state = 1 时，下降沿时，寄存器组读出数据输出 (\$0 = 0 参与运算，立即数为-2)

state = 2 时，DBDR 输出写入寄存器的数据 (Dbout = ALUout = result = -2)

state = 3 时，可以看到 REG[2]在下降沿被写入 (\$14 = -2)

nextPC[31:0]	00000028	00		00000028		
curPC[31:0]	00000024	00		00000024		
Instruction[31:0]	080efffe	99814000		080efffe		
state[2:0]	3	3	0	1	2	3
readData1[31:0]	00000000	00000002		00000000		
readData2[31:0]	fffffffe	00000008		00000000		
readDataout1[31:0]	00000000	00000002		00000000		
readDataout2[31:0]	00000000	00000008		00000000		
JumpAdd[31:0]	003bfff8	06050000		003bfff8		
Extend_Imme[31:0]	fffffffe	00004000		fffffffe		
A[31:0]	00000000	00000002		00000000		
B[31:0]	fffffffe	00000008				fffffffe
result[31:0]	fffffffe	00000001	00000000	fffffffe		
ALUout[31:0]	fffffffe	00000001	00000000			fffffffe
dataOut[31:0]	00000000					00000000
DBdata[31:0]	fffffffe	00000001	00000000	fffffffe		
DBout[31:0]	fffffffe	00000001	00000000			fffffffe
writeData[31:0]	fffffffe	00000001	00000000			fffffffe
REGS[14][31:0]	fffffffe		00000000			

slt \$9,\$8,\$14

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组读出数据输出 (\$8 = 1, \$14 = -2)

state = 2 时，下降沿时，DBDR 输出写入寄存器的数据 (Dbout = result = 0)

state = 3 时，可以看到 REG[9]不改变 (\$9 = 0)

nextPC[31:0]	0000002c	0		0000002c		
curPC[31:0]	00000028	0		00000028		
Instruction[31:0]	990e4800	080efffe		990e4800		
state[2:0]	3	3	0	1	2	3
readData1[31:0]	00000001	00000000		00000001		
readData2[31:0]	fffffffe			fffffffe		
readDataout1[31:0]	00000001	00000000		00000001		
readDataout2[31:0]	fffffffe	00000000		fffffffe		
JumpAdd[31:0]	04392000	003bfff8		04392000		
Extend_Imme[31:0]	00004800	fffffffe		00004800		
A[31:0]	00000001	00000000		00000001		
B[31:0]	fffffffe			fffffffe		
result[31:0]	00000000	fffffffe		00000000		
ALUout[31:0]	00000000	fffffffe		00000000		
dataOut[31:0]	00000000					00000000
DBdata[31:0]	00000000	fffffffe		00000000		
DBout[31:0]	00000000	fffffffe		00000000		
writeData[31:0]	00000000	fffffffe		00000000		
REGS[9][31:0]	00000000					00000000

slti \$10,\$9,2

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组读出数据输出 (\$9 = 0, 立即数为 2, result = 1)

state = 2 时，下降沿时，DBDR 输出写入寄存器的数据 (DBout = result = 1)

state = 3 时，可以看到 REG[10]在下下降沿被写入 (\$10 = 1)

nextPC[31:0]	00000030	0		00000030		
curPC[31:0]	0000002c	0		0000002c		
Instruction[31:0]	9d2a0002	990e4800		9d2a0002		
state[2:0]	3	3	0	1	2	3
readData1[31:0]	00000000	00000001		00000000		
readData2[31:0]	00000001	fffffffe		00000000		00000001
readDataout1[31:0]	00000000	00000001		00000000		
readDataout2[31:0]	00000000	fffffffe		00000000		
JumpAdd[31:0]	04a80008	04392000		04a80008		
Extend_Imme[31:0]	00000002	00004800		00000002		
A[31:0]	00000000	00000001		00000000		
B[31:0]	00000002	fffffffe		00000002		
result[31:0]	00000001	00000000		00000001		
ALUout[31:0]	00000001	00000000		00000001		
dataOut[31:0]	00000000					00000000
DBdata[31:0]	00000001	00000000		00000001		
DBout[31:0]	00000001	00000000		00000001		
writeData[31:0]	00000001	00000000		00000001		
REGS[10][31:0]	00000001		00000000			

slti \$11,\$10,0

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组读出数据输出 (\$10 = 1, 立即数为 0)

state = 2 时，下降沿时，DBDR 输出写入寄存器的数据 (DBout = result = 0)

state = 3 时，可以看到 REG[11]不改变 (\$11 = 0)

nextPC[31:0]	00000034	00		00000034		
curPC[31:0]	00000030	00		00000030		
Instruction[31:0]	9d4b0000	9d2a0002		9d4b0000		
state[2:0]	3	3	0	1	2	3
readData1[31:0]	00000001	00000000		00000001		
readData2[31:0]	00000000	00000001		00000000		
readDataout1[31:0]	00000001	00000000		00000001		
readDataout2[31:0]	00000000	00000000	00000001		00000000	
JumpAdd[31:0]	052c0000	04a80008		052c0000		
Extend_Imme[31:0]	00000000	00000002		00000000		
A[31:0]	00000001	00000000		00000001		
B[31:0]	00000000	00000002		00000000		
result[31:0]	00000000	00000001		00000000		
ALUout[31:0]	00000000	00000001		00000000		
dataOut[31:0]	00000000					00000000
DBdata[31:0]	00000000	00000001		00000000		
DBout[31:0]	00000000	00000001		00000000		
writeData[31:0]	00000000	00000001		00000000		
REGS[11][31:0]	00000000					00000000

add \$11,\$11,\$8

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组读出数据输出 (\$11 = 0, \$8 = 1)

state = 2 时，下降沿时，DBDR 输出写入寄存器的数据 (DBout = result = 1)

state = 3 时，可以看到 REG[11]在下降沿被写入，结果为 (\$11 = 1)

nextPC[31:0]	00000038	X		00000038		X
curPC[31:0]	00000034	X		00000034		X
Instruction[31:0]	01685800	9d4b0	X	01685800		X
state[2:0]	3	X	0	1	2	3
readData1[31:0]	00000001	00000	X	00000000		X
readData2[31:0]	00000001	00000	X	00000001		X
readDataout1[31:0]	00000000	00000001	X	00000000		X
readDataout2[31:0]	00000001	00000000	X	00000001		X
JumpAdd[31:0]	05a16000	052c0	X	05a16000		X
Extend_Imme[31:0]	00005800	00000	X	00005800		X
A[31:0]	00000000	00000001	X	00000000		X
B[31:0]	00000001	00000000	X	00000001		X
result[31:0]	00000001	00000	X	00000001		X
ALUout[31:0]	00000001	00000000	X	00000001		X
dataOut[31:0]	00000000		X		00000000	X
DBdata[31:0]	00000001	00000	X	00000001		X
DBout[31:0]	00000001	00000000	X	00000001		X
writeData[31:0]	00000001	00000000	X	00000001		X
REGS[11][31:0]	00000001		X	00000000		X

bne \$11,\$2,-2 (≠,转 34)

state = 0 时，上升沿时 PC 修改，此时 nextPC 为 PC + 4 = 3CH，下降沿时 IR 传出指令，计算得 zero = 0， nextPC 为 34H

state = 1 时，下降沿时，寄存器组读出数据输出 (\$11 = 1, \$2 = 2)

state = 2 时，下降沿时，ALUout 输出 ALU 结果 (ALUout = result = -1)

此时，zero = 0，PCSrc = 1，分支跳转，可以看到 nextPC = 34H，下一状态为 0

nextPC[31:0]	00000034	X	0000003c	X	00000034	X	0000	X
curPC[31:0]	00000038	X		X	00000038		X	
Instruction[31:0]	d562fffe	0168	X		d562fffe		X	
state[2:0]	2	X	0	X	1	X	2	X
PCSrc[1:0]	1		0	X	1		X	
zero	0			X			X	
readData1[31:0]	00000001			X			00000001	X
readData2[31:0]	00000002	0000	X		00000002		X	
readDataout1[31:0]	00000001	0000	X				X	
readDataout2[31:0]	00000002	00000001	X		00000002		X	
JumpAdd[31:0]	058bfff8	05a1	X		058bfff8		X	
Extend_Imme[31:0]	fffffffe	0000	X		fffffffe		X	
A[31:0]	00000001	0000	X				X	
B[31:0]	00000002	00000001	X		00000002		X	
result[31:0]	ffffffff	0000	X	00000000	X	ffffffff	X	

add \$11,\$11,\$8

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组读出数据输出 (\$11 = 1, \$8 = 1)

state = 2 时，下降沿时，DBDR 输出写入寄存器的数据 (DBout = result = 2)

state = 3 时，可以看到 REG[11]在下降沿被写入，结果为 (\$11 = 2)

nextPC[31:0]	00000038	0	0000	00000038	0000
curPC[31:0]	00000034	0		00000034	
Instruction[31:0]	01685800	d562fffe	01685800		
state[2:0]	3	2	0	1	2
readData1[31:0]	00000002		00000001		
readData2[31:0]	00000001	00000002	00000001		
readDataout1[31:0]	00000001		00000001		
readDataout2[31:0]	00000001	00000002	00000001		
JumpAdd[31:0]	05a16000	058bfff8	05a16000		
Extend_Imme[31:0]	00005800	fffffffe	00005800		
A[31:0]	00000001		00000001		
B[31:0]	00000001	00000002	00000001		
result[31:0]	00000002	ffffff	00000003	00000002	
ALUout[31:0]	00000002	ffffff	00000003	00000002	
dataOut[31:0]	00000000			00000000	
DBdata[31:0]	00000002	ffffff	00000003	00000002	
DBout[31:0]	00000002	ffffff	00000003	00000002	
writeData[31:0]	00000002	ffffff	00000003	00000002	
REGS[11][31:0]	00000002		00000001		

bne \$11,\$2,-2 (≠,转 34)

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组读出数据输出 (\$11 = 2, \$2 = 2)，判断不分支，nextPC 改变

state = 2 时，ALU 结果 (ALUout = result = 0)

此时，zero = 1，PCSrc = 0，不分支，可以看到 nextPC = 3CH，下一状态为 0

nextPC[31:0]	0000003c	0000	00000034	0000003c	
curPC[31:0]	00000038		00000038		
Instruction[31:0]	d562fffe	01685800	d562fffe		
state[2:0]	2	3	0	1	2
zero	1				
PCSrc[1:0]	0	0	1		
readData1[31:0]	00000002			00000002	
readData2[31:0]	00000002	00000001		00000002	
readDataout1[31:0]	00000002	00000001			
readDataout2[31:0]	00000002	00000001			
JumpAdd[31:0]	058bfff8	05a16000	058bfff8		
Extend_Imme[31:0]	fffffffe	00005800	fffffffe		
A[31:0]	00000002	00000001			
B[31:0]	00000002	00000001	00000002		
result[31:0]	00000000	00000002	00000001	00000000	

addi \$2,\$2,-1

state = 0 时，上升沿时 PC 修改，下降沿 IR 传出指令

state = 1 时，寄存器组读出数据输出 (\$2 = 2，立即数为-1)

state = 2 时，DBDR 输出写入寄存器的数据 (DBout = result = 1)

state = 3 时，可以看到 REG[2]在下降沿被写入，结果为 1 (\$2 = 1)

nextPC[31:0]	00000040	X		00000040			X
curPC[31:0]	0000003c	X		0000003c			X
Instruction[31:0]	0842ffff	d562f		0842ffff			
state[2:0]	3	X	0	X	2	X	3
readData1[31:0]	00000001			00000002			X
readData2[31:0]	00000001			00000002			X
readDataout1[31:0]	00000002			00000002			
readDataout2[31:0]	00000002			00000002			
JumpAdd[31:0]	010bffff	058bf		010bffff			
Extend_Imme[31:0]	ffffff	fffff		ffffff			
A[31:0]	00000002			00000002			
B[31:0]	ffffff	00000		ffffff			
result[31:0]	00000001	00000		00000001			
ALUout[31:0]	00000001	00000000		00000001			
dataOut[31:0]	00000000						00000000
DBdata[31:0]	00000001	00000		00000001			
DBout[31:0]	00000001	00000000		00000001			
writeData[31:0]	00000001	00000000		00000001			
REGS[2][31:0]	00000001			00000002			

bgtz \$2,-2 (>0,转 3C)

state = 0 时，上升沿时 PC 修改，此时 nextPC 为 PC + 4 = 44H，下降沿时 IR 传出指令，计算得 sign = 0，zero = 0，nextPC 为 3CH

state = 1 时，下降沿时，寄存器组读出数据输出 (\$2 = 1)

state = 2 时，下降沿时，ALUout 输出 ALU 结果 (ALUout = result = 1)

此时，zero = 0，PCSrc = 1，分支跳转，可以看到 nextPC = 34H，下一状态为 0

nextPC[31:0]	0000003c	00	00000044	0000003c		0000
curPC[31:0]	00000040	00	00000040			
Instruction[31:0]	d840ffff	0842ffff		d840ffff		
state[2:0]	2	3	0	1	2	
zero	0					
sign	0					
PCSrc[1:0]	1					
readData1[31:0]	00000001					00000001
readData2[31:0]	00000000	00000001		00000000		
readDataout1[31:0]	00000001	00000002				
readDataout2[31:0]	00000000	00000002	00000001			00000000
JumpAdd[31:0]	0103ffff	010bffff		0103ffff		
Extend_Imme[31:0]	fffffffe	ffffff		fffffffe		
A[31:0]	00000001	00000002				
B[31:0]	00000000	ffffff	00000001		00000000	
result[31:0]	00000001	00000001	00000000		00000001	
ALUout[31:0]	00000001	00000001	00000000		00000000	000

addi \$2,\$2,-1

state = 0 时，上升沿时 PC 修改，下降沿 IR 传出指令

state = 1 时，寄存器组读出数据输出 (\$2 = 1，立即数为-1)

state = 2 时，DBDR 输出写入寄存器的数据 (DBout = result = 0)

state = 3 时，可以看到 REG[2]在下降沿被写入，结果为 1 (\$2 = 0)

nextPC[31:0]	00000040	0000003c	0000	00000040
curPC[31:0]	0000003c	00000040		0000003c
Instruction[31:0]	0842ffff	d840fffe		0842ffff
state[2:0]	3	1	2	0
readData1[31:0]	00000000		00000001	
readData2[31:0]	00000000	00000000		00000001
readDataout1[31:0]	00000001		00000001	
readDataout2[31:0]	00000001	00000000		00000001
JumpAdd[31:0]	010bffff	0103fff8		010bffff
Extend_Imme[31:0]	ffffffff	fffffffe		ffffffff
A[31:0]	00000001		00000001	
B[31:0]	ffffffff	00000000		ffffffff
result[31:0]	00000000	00000001		00000000
ALUout[31:0]	00000000	00000000	00000001	00000000
dataOut[31:0]	00000000			00000000
DBdata[31:0]	00000000	00000001		00000000
DBout[31:0]	00000000	00000000	00000001	00000000
writeData[31:0]	00000000	00000000	00000001	00000000
REGS[2][31:0]	00000000		00000001	

bgtz \$2,-2 (>0,转 3C)

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令

state = 1 时，下降沿时，寄存器组读出数据输出 (\$2 = 0)，判断不分支，nextPC 不变

state = 2 时，ALU 结果 (ALUout = result = 0)

此时，zero = 1，sign = 0，PCSrc = 0，不分支，可以看到 nextPC = 44H，下一状态为 0

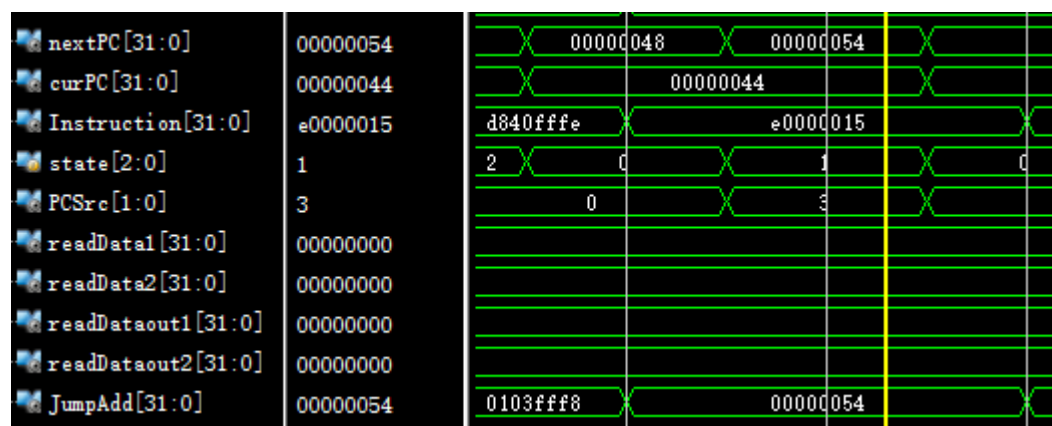
nextPC[31:0]	00000044	00000044		00000044
curPC[31:0]	00000040	00000040		00000040
Instruction[31:0]	d840fffe	0842ffff		d840fffe
state[2:0]	2	3	0	1
PCSrc[1:0]	0			0
sign	0			
zero	1			
readData1[31:0]	00000000			
readData2[31:0]	00000000			
readDataout1[31:0]	00000000	00000001		
readDataout2[31:0]	00000000	00000001		
JumpAdd[31:0]	0103fff8	010bffff		0103fff8
Extend_Imme[31:0]	fffffffe	ffffffff		fffffffe
A[31:0]	00000000	00000001		
B[31:0]	00000000	ffffffff		
result[31:0]	00000000			
ALUout[31:0]	00000000			

j 0x0000054

state = 0 时，上升沿时 PC 修改，下降沿时 IR 传出指令，计算 JumpAdd = 54H

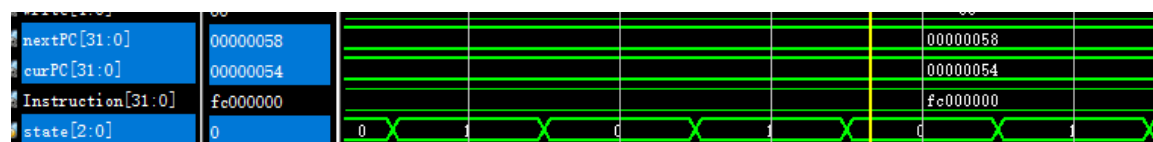
state = 1 时，上升沿给出 PCSrc = 3，nextPC 修改为 54H

可以看到下一状态为 0



halt

PC 不再修改，state 在 0 和 1 之间往复



综上，所有指令正确执行，与上述实验分析中预测的执行情况相同。

六. basys多周期CPU烧录实验过程与结果

A. 设计思路及代码

(a.改动部分) CPU 实现方法同仿真时相同，但由于 basys3 板子内部的接口有限，不能容纳上述实现。对上述实现在不影响指令功能的情况下，对部分接口及代码实现做以下调整：

1. 指令地址，即 PC 的宽度由 32 位变为 8 位
2. 寄存器宽度由 32 位变为 8 位，相应地，运算数、运算结果等做同样处理
3. 立即数仍为 16 位，但经过扩展单元后变为 8 位，即不进行 0 扩展或符号扩展，而直接取低 8 位
4. 跳转指令的目标地址变为立即数左移 2 位所得结果的低 8 位，即不再与 PC 拼接

(b.补充部分) 为在板子上显示指令执行结果，新增 SW_in 信号选择显示的信息另外新增模块如下：

divide: 分频单元，对板子上自带时钟进行分频

```
module divide(clk, n, clkout);
    input clk;
    input [30: 0] n;//分频倍数
    output clkout;
    reg clkout;
```

```

    reg [39: 0] temp;
    initial begin
        clkout = 1;
        temp = 0;
    end
    always@(posedge clk)
        begin
            temp = temp + 1;
            if (temp == n / 2)begin
                clkout = 0;//一半时间时翻转
            end
            if (temp == n)begin
                clkout = 1;
                temp = 0;//完整脉冲，再次翻转
            end
        end
    end
endmodule

```

choose_show: 接收选择信号，输出在数码管上要显示的两个数据

module choose_show(input [1:0] SW_in, input [7:0] curPC, nextPC, readData1, readData2, result, resultDB, input [4:0] read1, read2, output reg[7:0] show1, show2);

```

    always@(*)begin
        if (SW_in == 2'b00) {show1, show2} = {curPC, nextPC};
        else if (SW_in == 2'b01) {show1, show2} = {3'b000, read1, readData1};
        else if (SW_in == 2'b10) {show1, show2} = {3'b000, read2, readData2};
        else {show1, show2} = {result, resultDB};
    end
endmodule

```

count4: 4 进制计数器，根据上述分频时钟实现四进制计数

module count4(input CLK, output reg[1:0] count);

```

    initial begin
        count = 2'b00;
    end
    always @(negedge CLK) count = count + 1;
endmodule

```

chooseNUM: 根据上述计数器计数结果及显示的数据，选择当前时刻显示的数字（数码管一个时刻只能显示一个数字）

module chooseNUM(input [1:0] count, input [7:0] show1, show2, output reg[3:0]x);

```

    always@(*) begin
        if (count == 2'b00) x = show1[7:4];
        else if (count == 2'b01) x = show1[3:0];
        else if (count == 2'b10) x = show2[7:4];
    end
endmodule

```



```

        else if (count == 2'b11) x = show2[3:0];
    end
endmodule

```

NUMPOS: 根据上述计数器计数结果, 选择当前时刻显示的数字在数码管上的位置

```

module NUMPOS(input [1:0]num,output reg [3:0] position);
    always@(*)begin
        case(num)
            2'b00:position = 4'b0111;
            2'b01:position = 4'b1011;
            2'b10:position = 4'b1101;
            2'b11:position = 4'b1110;
        endcase
    end
endmodule

```

seg_7: 七段码译码器

```

module seg_7(input [3:0] num, output reg[7:0] dispcode);
    always@(num)begin
        case(num)
            4'b0000 : dispcode = 8'b1100_0000; //0; '0'-亮灯, '1'-熄灯
            4'b0001 : dispcode = 8'b1111_1001; //1
            4'b0010 : dispcode = 8'b1010_0100; //2
            4'b0011 : dispcode = 8'b1011_0000; //3
            4'b0100 : dispcode = 8'b1001_1001; //4
            4'b0101 : dispcode = 8'b1001_0010; //5
            4'b0110 : dispcode = 8'b1000_0010; //6
            4'b0111 : dispcode = 8'b1101_1000; //7
            4'b1000 : dispcode = 8'b1000_0000; //8
            4'b1001 : dispcode = 8'b1001_0000; //9
            4'b1010 : dispcode = 8'b1000_1000; //A
            4'b1011 : dispcode = 8'b1000_0011; //b
            4'b1100 : dispcode = 8'b1100_0110; //C
            4'b1101 : dispcode = 8'b1010_0001; //d
            4'b1110 : dispcode = 8'b1000_0110; //E
            4'b1111 : dispcode = 8'b1000_1110; //F
            default : dispcode = 8'b0000_0000; //不亮
        endcase
    end
endmodule

```

RM_Shake: 按键消抖模块, 参考博客文章《按键消抖的原理和基于 verilog 的消抖设计》

```

module RM_Shake (

```

```

input BJ_CLK, //采集时钟, 40Hz
input BUTTON_IN, //按键输入信号
output wire BUTTON_OUT //消抖后的输出信号
);
reg BUTTON_IN_Q, BUTTON_IN_2Q, BUTTON_IN_3Q;
always @(posedge BJ_CLK)
begin
    BUTTON_IN_Q <= BUTTON_IN;
    BUTTON_IN_2Q <= BUTTON_IN_Q;
    BUTTON_IN_3Q <= BUTTON_IN_2Q;
end
assign BUTTON_OUT = BUTTON_IN_2Q | BUTTON_IN_3Q;
endmodule

```

B. 实验结果

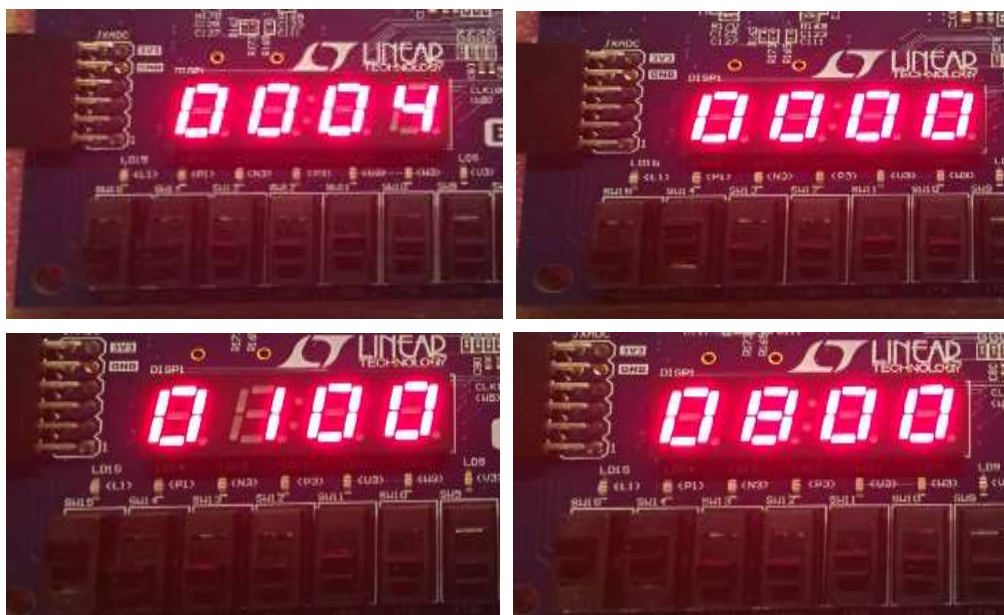
(以前两条指令为例)

addi \$1,\$0,8

state = 0

SW_in = 00, PC = 0, nextPC = 4; SW_in = 01, rs = 0, (rs) = 0;

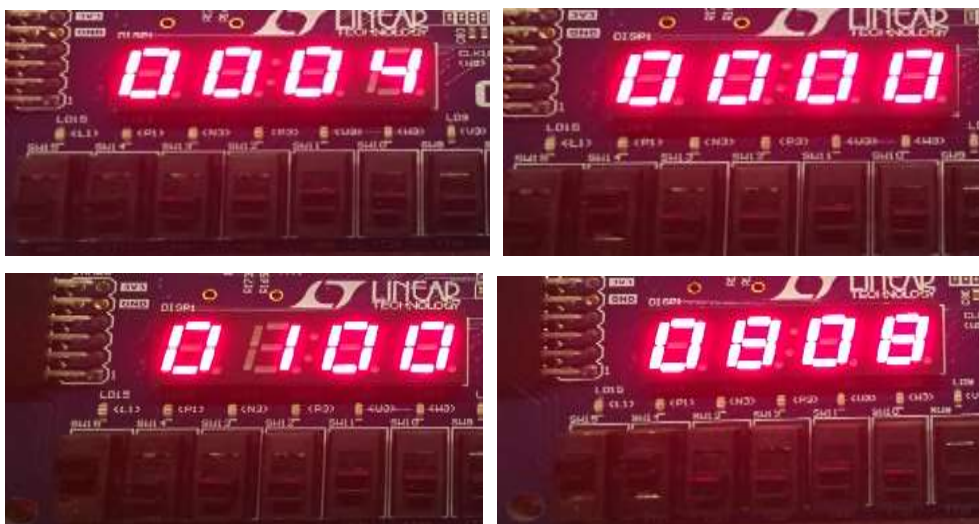
SW_in = 10, rt = 1, (rt) = 0; SW_in = 11, result = 8, DBDR = 0



state = 1 时, 寄存器组读出数据输出 (\$0 = 0,立即数为 8)

SW_in = 00, PC = 0, nextPC = 4; SW_in = 01, rs = 0, (rs) = 0;

SW_in = 10, rt = 1, (rt) = 0; SW_in = 11, result = 8, DBDR = 8

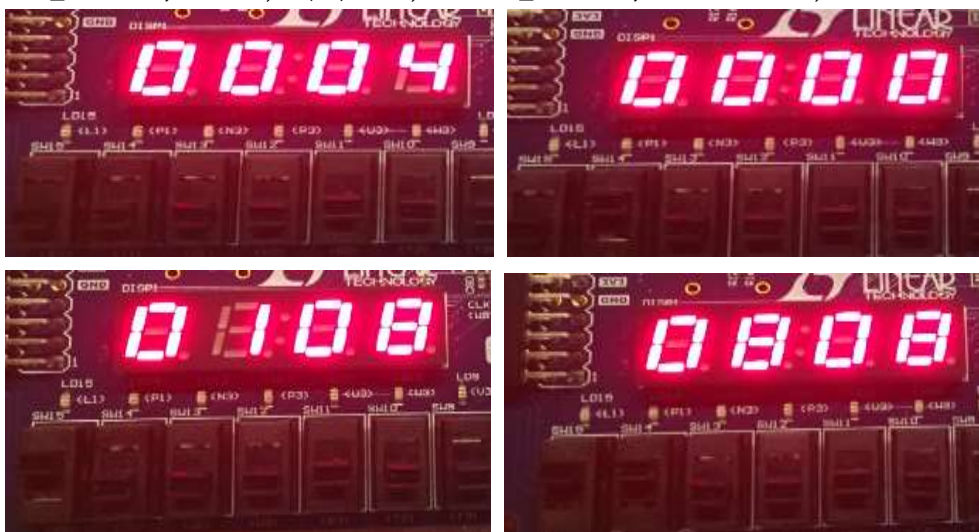


state = 2 时, DBDR 输出写入寄存器的数据 (Dbout = result = 8)
同 state = 1 的情况

state = 3 时, REG[1]在下降沿被写入, 结果为 8 (\$1 = 8)

SW_in = 00, PC = 0, nextPC = 4; SW_in = 01, rs = 0, (rs) = 0;

SW_in = 10, rt = 1, (rt) = 0; SW_in = 11, ALUout = 0, DBDR = 0

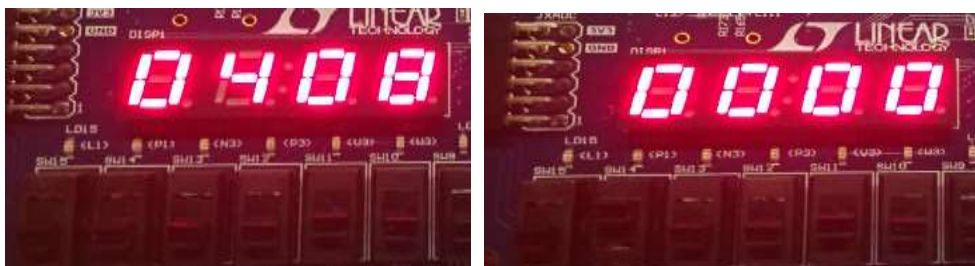


ori \$2,\$0,2

state = 0 (下降沿到达时 DBDR 还未更新, 仍为之前的 8)

SW_in = 00, PC = 4, nextPC = 8; SW_in = 01, rs = 0, (rs) = 0;

SW_in = 10, rt = 2, (rt) = 0; SW_in = 11, result = 2, DBDR = 8





state = 1 (下降沿时 DBDR 更新为 2)

SW_in = 00, PC = 4, nextPC = 8; SW_in = 01, rs = 0, (rs) = 0;

SW_in = 10, rt = 2, (rt) = 0; SW_in = 11, result = 2, DBDR = 2



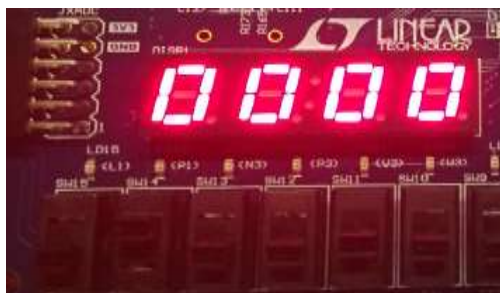
state = 2

同 state = 1 的情况

state = 3 REG[2]在下降沿被写入, 结果为 2 (\$2 = 2)

SW_in = 00, PC = 4, nextPC = 8; SW_in = 01, rs = 0, (rs) = 0;

SW_in = 10, rt = 2, (rt) = 2; SW_in = 11, result = 2, DBDR = 2



以上两条指令均按照预期进行，之后的指令经检验，也正确执行，此处不再赘述。

综上，basy3 多周期 CPU 烧录实验完成。

七. 实验心得与课程总结

从星期一到星期五，花了挺长的时间做这个实验。

多周期CPU的很多模块跟单周期是一样的，很多代码可以直接用，所以写代码所花的时候不是很多，但是比起单周期来说，耗费的精力更多。在把代码大致完成后，剩下的是调整各个模块是由时钟的上升沿或是下降沿触发的时候很烦恼。

一开始的想法是控制单元由上升沿触发，IF阶段给出PCWre使能信号，那为了使PC修改之前PCWre已经为使能，PC单元应该在下降沿触发，而IR模块又必须在PC单元之后触发，同时在控制单元之前触发，觉得出现了一个矛盾。

无从下手，试了几种情况也都不行，于是到网上查找其他人是如何设计的。看了几份，他们的PC都是由电平触发，来解决这个冲突问题，或者是没有考虑这个冲突问题。我也尝试了以下，仿真的波形没有问题，但是在烧录到板子的时候就会出现问題。

想了很久，最后决定到教室问老师。在听了我对时钟的安排时，老师开始很仔细地给我用每一条指令分析我的安排是否可行，然后告诉我应该如何解决——原来PC单元和控制单元都是上升沿触发，为了让PC可以及时改变，PCWre应该在每条指令的最后一个周期置1使能。这样在下一个周期的上升沿到达时，PC就可以正确改变。在听了老师讲解大概半个小时，终于知道该怎么解决这一问题。

回到宿舍马上对代码进行了修改，终于波形图与板子都可以正确执行，对照着测试程序，一条一条，每个周期该做什么事情都如预期进行，很有成就感。

之后着手开始写实验报告，尽量在实验分析中把思路讲得详细一些，各种不同周期数的指令应该如何执行也详细地写在里面，通过写报告这个过程也把多周期CPU的理解加深了一些。

这次已经是本学期的最后一个实验了，感觉过得很快，但是在这个学期的整个学习和实验过程里面真的学到了很多。结合理论所学，跟实际情况又不尽相同，在这个过程中有遇到很多问题，但是很高兴的是我都能独立思考，思考如何去解决，再不懂时查阅相关资料信息，跟同学讨论以及请教老师。我相信这对我来说是一个很有意义的锻炼。在遇到问题的时候也是让我进步和看到自己不足的时候，印象最深的应该是在问老师多周期CPU如何解决冲突问题的时候，老师让我说说我是怎么安排的。我还不能很熟练地说出来，但是老师在听完之

后，以非常快的速度对一些指令的每个阶段进行了分析，而我甚至不能跟上老师的速度。这才知道原来我的掌握是这么的浅，跟其他人的差距有多么大。说真的是挺有触动的。算是一种动力吧，今后也会更加努力，周围也有很多很强的人，希望我也能变得很厉害。

这学期的实验都顺利完成了，但是还有很多不足的地方，在以后的其他课程中，希望自己能做得更好。然后这学期学得很值！！

八. 编写一个编译器，将 MIPS 汇编程序编译为二进制机器码

代码实现如下：

```
#include <iostream>
#include <string>
#include <fstream>
#include <bitset>
using namespace std;
enum instype{rtype, jump, jmp, itype, sll, ls, beq, bne, bgtz};
string get_code(const string & s){
    if (s == "halt") return "111111_00000000000000000000000000000000";
    string result = "";
    int first_space = s.find(' ');
    string op = s.substr(0,first_space);
    instype flag = rtype;
    if (op == "add") result = "000000_";
    else if (op == "sub") result = "000001_";
    else if (op == "and") result = "010001_";
    else if (op == "or") result = "010000_";
    else if (op == "slt") result = "100110_";
    else if (op == "j") { result = "111000_"; flag = jump; }
    else if (op == "jal") { result = "111010_"; flag = jump; }
    else if (op == "jr") { result = "111001_"; flag = jmp; }
    else if (op == "sll") { result = "011000_"; flag = sll; }
    else if (op == "lw") { result = "110001_"; flag = ls; }
    else if (op == "sw") { result = "110000_"; flag = ls; }
    else if (op == "beq") { result = "110100_"; flag = beq; }
    else if (op == "bne") { result = "110101_"; flag = bne; }
    else if (op == "bgtz") { result = "110110_"; flag = bgtz; }
    else {
        flag = itype;
        if (op == "addi") result = "000010_";
        else if (op == "ori") result = "010010_";
        else if (op == "sli") result = "100111_";
    }
}
string leave = s.substr(first_space);
//rtype
if (flag == rtype){
    int i = 0, times = 3;
    string rd = "", rt = "", rs = "";
    while(times){
        if (leave[i] == '$'){
            if (times == 3) rd = bitset<5>(atoi(leave.substr(i + 1).c_str())).to_string();
            else if (times == 2) rs = bitset<5>(atoi(leave.substr(i + 1).c_str())).to_string();
            else if (times == 1) rt = bitset<5>(atoi(leave.substr(i + 1).c_str())).to_string();
        }
    }
}
```

```

        times--;
    }
    i++;
}
result = result + rs + '_' + rt + '_' + rd + "_000000000000";
}
//itype
else if (flag == itype || flag == slt || flag == beqz){
    unsigned int i = 0, times = 2;
    string rt = "", rs = "", imme = "";
    while(times){
        if (leave[i] == '$'){
            if (times == 2) rt = bitset<5>(atoi(leave.substr(i + 1).c_str())).to_string();
            else if (times == 1) rs = bitset<5>(atoi(leave.substr(i + 1).c_str())).to_string();
            times--;
        }
        i++;
    }
    while(i < leave.size()){
        if (leave[i] == ','){
            if (flag == itype || flag == beqz) imme = bitset<16>(atoi(leave.substr(i +
1).c_str())).to_string();
            else imme = bitset<5>(atoi(leave.substr(i + 1).c_str())).to_string();
            break;
        }
        i++;
    }
    if (flag == itype) result = result + rs + '_' + rt + '_' + imme;
    else if (flag == beqz) result = result + rt + '_' + rs + '_' + imme;
    else result = result + "00000_" + rs + '_' + rt + '_' + imme + "_000000";
}
//bgtz
else if (flag == bgtz){
    unsigned int i = 0;
    string rs = "", imme = "";
    while(i < leave.size()){
        if (leave[i] == '$'){
            rs = bitset<5>(atoi(leave.substr(i + 1).c_str())).to_string();
            break;
        }
        i++;
    }
    while(i < leave.size()){
        if (leave[i] == ','){

```



```

        imme = bitset<16>(atoi(leave.substr(i + 1).c_str())).to_string();
        break;
    }
    i++;
}
result = result + rs + "_00000_" + imme;
}
//ls
else if (flag == ls){
    unsigned int i = 0;
    string rt = "", rs = "", imme = "";
    while(i < leave.size()){
        if (leave[i] == '$'){
            rt= bitset<5>(atoi(leave.substr(i + 1).c_str())).to_string();
            break;
        }
        i++;
    }
    while(i < leave.size()){
        if (leave[i] == ','){
            imme = bitset<16>(atoi(leave.substr(i + 1).c_str())).to_string();
            break;
        }
        i++;
    }
    while(i < leave.size()){
        if (leave[i] == '$'){
            rs= bitset<5>(atoi(leave.substr(i + 1).c_str())).to_string();
            break;
        }
        i++;
    }
    result = result + rs + '_' + rt + '_' + imme;
}
//sll
if (flag == jump){
    string temp = "";
    for (unsigned int i = 0; i < leave.size(); i++){
        if (leave[i] == 'x' || leave[i] == 'X'){
            string hexi[16]={"0000","0001","0010","0011","0100","0101",
"0110","0111","1000","1001","1010","1011","1100","1101","1110","1111"};
            for (unsigned int j = i + 1; j < leave.size(); j++){
                if ('0' <= leave[j] && '9' >= leave[j]) temp += hexi[leave[j] - '0'];
                else if ('a' <= leave[j] && 'z' >= leave[j]) temp += hexi[leave[j] - 'a'];
            }
        }
    }
}

```

```

        else temp += hexi[leave[j]] - 'A';
    }
    break;
}

int len = 28 - temp.size();
for (int i = 0; i < len; i++) result += '0';
result += temp.substr(0, temp.size() - 2);
}

if (flag == jmpr){
    unsigned int i = 0;
    while(i < leave.size()){
        if (leave[i] == '$'){
            result += bitset<5>(atoi(leave.substr(i + 1).c_str())).to_string();
            result += bitset<21>(0).to_string();
            break;
        }
        i++;
    }
}

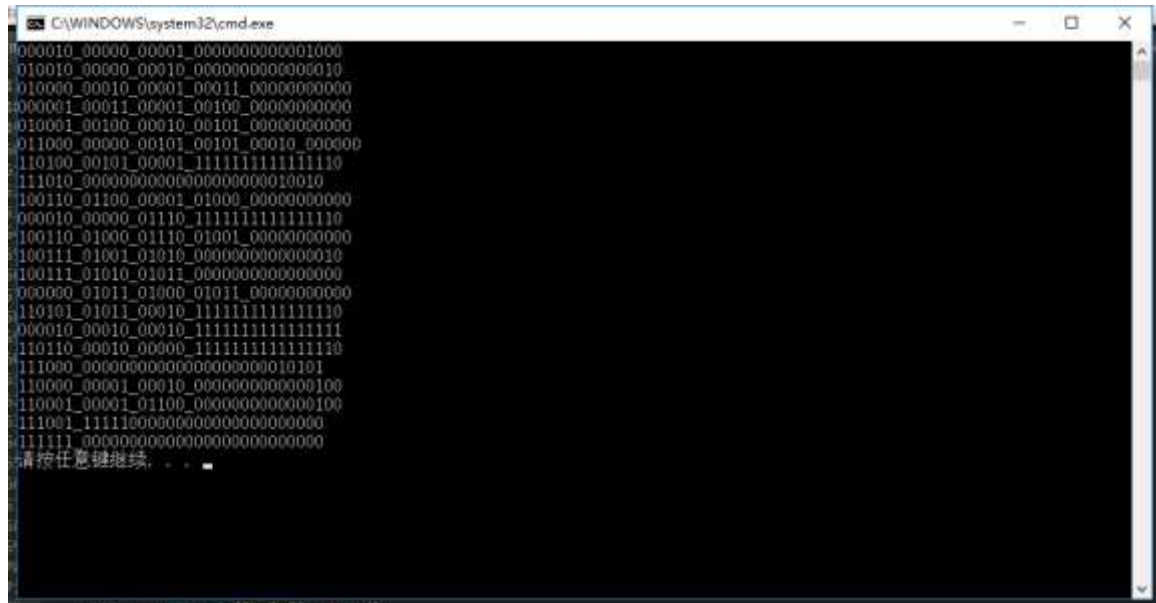
return result;
}

int main(){
    cout << get_code("addi  $1,$0,8") << endl;
    cout << get_code("ori   $2,$0,2") << endl;
    cout << get_code("or    $3,$2,$1") << endl;
    cout << get_code("sub   $4,$3,$1") << endl;
    cout << get_code("and   $5,$4,$2") << endl;
    cout << get_code("sll   $5,$5,2") << endl;
    cout << get_code("beq   $5,$1,-2") << endl;
    cout << get_code("jal   0x0000048") << endl;
    cout << get_code("slt   $8,$12,$1") << endl;
    cout << get_code("addi  $14,$0,-2") << endl;
    cout << get_code("slt   $9,$8,$14") << endl;
    cout << get_code("slti  $10,$9,2") << endl;
    cout << get_code("slti  $11,$10,0") << endl;
    cout << get_code("add   $11,$11,$8") << endl;
    cout << get_code("bne   $11,$2,-2") << endl;
    cout << get_code("addi  $2,$2,-1") << endl;
    cout << get_code("bgtz $2,-2") << endl;
    cout << get_code("j     0x0000054") << endl;
    cout << get_code("sw    $2,4($1)") << endl;
    cout << get_code("lw    $12,4($1)") << endl;
    cout << get_code("jr    $31") << endl;
    cout << get_code("halt") << endl;
}

```

}

程序执行情况如下：



```
C:\WINDOWS\system32\cmd.exe
000010_00000_00001_0000000000001000
010010_00000_00010_0000000000000010
010000_00010_00001_00011_000000000000
000001_00011_00001_00100_000000000000
010001_00100_00010_00101_000000000000
011000_00000_00101_00101_00010_000000
110100_00101_00001_1111111111111110
111010_000000000000000000000010010
100110_01100_00001_01000_000000000000
000010_00000_01110_1111111111111110
100110_01000_01110_01001_000000000000
100111_01001_01010_0000000000000010
100111_01010_01011_0000000000000000
000000_01011_01000_01011_000000000000
110101_01011_00010_1111111111111110
000010_00010_00010_1111111111111111
110110_00010_00000_1111111111111110
111000_000000000000000000000010101
110000_00001_00010_000000000000100
110001_00001_01100_000000000000100
111001_11110000000000000000000000
111111_0000000000000000000000000000
请按任意键继续...
```