



《计算机组成原理实验》

实验报告

(实验二)

学院名称 : 数据科学与计算机学院

专业(班级) : 17 软件工程 2 班

学生姓名 : 张伟焜

学号 : 17343155

时间 : 2018 年 11 月 25 日

成 绩 :

实验二：单周期CPU设计与实现

一. 实验目的

- (1) 掌握单周期 CPU 数据通路图的构成、原理及其设计方法；
- (2) 掌握单周期 CPU 的实现方法，代码实现方法；
- (3) 认识和掌握指令与 CPU 的关系；
- (4) 掌握测试单周期 CPU 的方法。

二. 实验内容

设计一个单周期 CPU，该 CPU 至少能实现以下指令功能操作。指令与格式如下：

=> 算术运算指令

- (1) add rd , rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能: $rd \leftarrow rs + rt$ 。 reserved 为预留部分，即未用，一般填“0”。

- (2) sub rd , rs , rt

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

功能: $rd \leftarrow rs - rt$ 。

- (3) addiu rt , rs , immediate

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

功能: $rt \leftarrow rs + (\text{sign-extend})\text{immediate}$; immediate 符号扩展再参加“加”运算。

=> 逻辑运算指令

- (4) andi rt , rs , immediate

010000	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能: $rt \leftarrow rs \& (\text{zero-extend})\text{immediate}$; immediate 做“0”扩展再参加“加”运算。

- (5) and rd , rs , rt

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

功能: $rd \leftarrow rs \& rt$; 逻辑与运算。

- (6) ori rt , rs , immediate

010010	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能: $rt \leftarrow rs | (\text{zero-extend})\text{immediate}$; immediate 做“0”扩展再参加“或”运算。

- (7) or rd , rs , rt

010011	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能: $rd \leftarrow rs | rt$; 逻辑或运算。

==>移位指令

(8) sll rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa(5 位)	reserved
--------	----	---------	---------	---------	----------

功能: $rd \leftarrow rt \ll (zero-extend)sa$, 左移 sa 位 , (zero-extend)sa。**==>比较指令**

(9) slti rt, rs,immediate 带符号数

011100	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) 写存储器

100110	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能: $memory[rs + (sign-extend)immediate] \leftarrow rt$; 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。

(11) lw rt , immediate(rs) 读存储器

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

功能: $rt \leftarrow memory[rs + (sign-extend)immediate]$; 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**

(12) beq rs,rt,immediate

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

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

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”。

(13) bne rs,rt,immediate

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

功能: if($rs \neq rt$) $pc \leftarrow pc + 4 + (sign-extend)immediate \ll 2$ else $pc \leftarrow pc + 4$

特别说明: 与 beq 不同点是, 不等时转移, 相等时顺序执行。

(14) bltz rs,immediate

110010	rs(5 位)	00000	immediate(16 位)
--------	---------	-------	-----------------

功能: if($rs < \$zero$) $pc \leftarrow pc + 4 + (sign-extend)immediate \ll 2$ else $pc \leftarrow pc + 4$ 。

==>跳转指令

(15) j addr

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

功能: $pc \leftarrow \{(pc+4)[31:28],addr[27:2],2'b00\}$, 无条件跳转。

说明: 由于 MIPS32 的指令代码长度占 4 个字节, 所以指令地址二进制数最低 2 位均为 0, 将指令地址放进指令代码中时, 可省掉! 这样, 除了最高 6 位操作码外, 还有 26 位可用于存放地址, 事实上, 可存放 28 位地址, 剩下最高 4 位由 $pc+4$ 最高 4 位拼接上。

==> 停机指令

(16) halt

111111	0000000000000000000000000000(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:

31	2625	2120	1615	1110	65	0
op	rs	rt	rd	sa	funct	

6 位 5 位 5 位 5 位 5 位 6 位

I 类型:

31	2625	2120	1615	immediate	0
op	rs	rt		16 位	

6 位 5 位 5 位

J 类型:

31	2625	address	0
op		26 位	

其中，

op: 为操作码；

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

rt: 可读可写。为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd: 只写。为目的操作数寄存器，寄存器地址（同上）；

sa: 为位移量 (shift amt)，移位指令用于指定移多少位；

funct: 为功能码，在寄存器类型指令中 (R 类型) 用来指定指令的功能与操作码配合使用；

immediate: 为 16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载 (Load) / 数据保存 (Store) 指令的数据地址字节偏移量和分支指令中相对程序计数器 (PC) 的有符号偏移量；

address: 为地址。

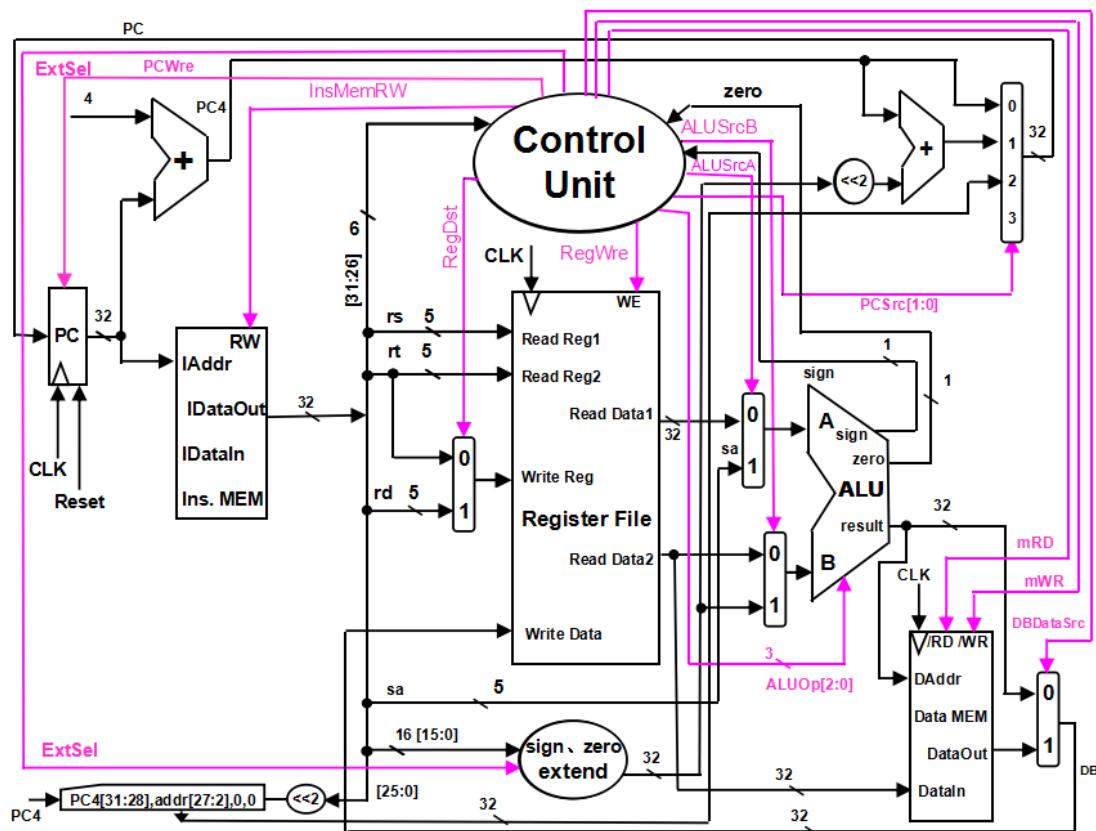


图 2 单周期 CPU 数据通路和控制线路图

图 2 是一个简单的基本上能够在单周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，先给出寄存器地址，读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。图中控制信号作用如表 1 所示，表 2 是 ALU 运算功能表。

表 1 控制信号的作用

相关部件及引脚说明：

控制信号名	状态“0”	状态“1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改，相关指令： halt	PC 更改，相关指令： 除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指令： add、sub、addiu、or、and、andi、ori、slti、beq、bne、bltz、sw、lw	来自移位数 sa，同时，进行 (zero-extend)sa，即 $\{27\{1'b0\}, sa\}$ ，相关指令： sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令： add、sub、or、and、beq、bne、bltz	来自 sign 或 zero 扩展的立即数，相关指令： addi、andi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令： add、addiu、sub、ori、or、and、andi、slti、sll	来自数据存储器 (Data MEM) 的输出，相关指令： lw
RegWre	无写寄存器组寄存器，相关指令： beq、bne、bltz、sw、halt	寄存器组写使能，相关指令： add、addiu、sub、ori、or、and、andi、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	输出高阻态	读数据存储器，相关指令： lw
mWR	无操作	写数据存储器，相关指令： sw
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令： addiu、andi、ori、slti、lw	写寄存器组寄存器的地址，来自 rd 字段，相关指令： add、sub、and、or、sll
ExtSel	(zero-extend)immediate(0 扩展)，相关指令： andi、ori	(sign-extend)immediate (符号扩展)，相关指令： addiu、slti、sw、lw、beq、bne、bltz
PCSrc[1..0]	00: pc<-pc+4，相关指令： add、addiu、sub、or、ori、and、andi、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0); 01: pc<-pc+4+(sign-extend)immediate<<2，相关指令： beq(zero=1)、bne(zero=0)、bltz(sign=1); 10: pc<-{(pc+4)[31:28],addr[27:2],2'b00}，相关指令： j; 11: 未用	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

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 1 读

/WR, 数据存储器写控制信号, 为 1 写

Register File: 寄存器组

Read Reg1, rs 寄存器地址输入端口

Read Reg2, rt 寄存器地址输入端口

Write Reg, 将数据写入的寄存器端口, 其地址来源 rt 或 rd 字段

Write Data, 写入寄存器的数据输入端口

Read Data1, rs 寄存器数据输出端口

Read Data2, rt 寄存器数据输出端口

WE, 写使能信号, 为 1 时, 在时钟边沿触发写入

ALU: 算术逻辑单元

result, ALU 运算结果

zero, 运算结果标志, 结果为 0, 则 zero=1; 否则 zero=0

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

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	$Y = A + B$	加
001	$Y = A - B$	减
010	$Y = B \ll A$	B 左移 A 位
011	$Y = A \vee B$	或
100	$Y = A \wedge B$	与
101	$Y = (A < B) ? 1 : 0$	比较 A < B 不带符号
110	$Y = (((A < B) \&\& (A[31] == B[31])) ((A[31] == 1) \&\& (B[31] == 0))) ? 1 : 0$	比较 A < B 带符号
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. 实验过程与结果

1. 单周期CPU设计的思想、方法：

采用模块化的设计方法，将单周期CPU划分为13个底层模块和1个顶层模块。单周期CPU主要分为如下器件（模块），PC，指令存储器，控制单元，ALU，寄存器组，存储器，立即数扩展单元，除此之外，还有许多数据选择模块。（具体模块如下）



由数据通路图可知，需要时钟信号触发的有三个模块，即PC变化模块，写寄存器模块，写存储器模块。其他模块没有时钟信号输入，相应的功能依靠组合逻辑电路实现。

PC指出当前执行的指令地址，在时钟上升沿到来时更新指令地址。指令存储器读取指令后将相应的操作数送至各个模块。控制单元则根据接受的操作代码和标志寄存器的值确定控制信号，使指令正确执行，并更新PC的值。如果需要进行写操作，在时钟下降沿到来时执行写操作，将数据写入寄存器或存储器。

数据选择模块根据收到的信号，从多个输入中选择需要的信息输出，作为后序模块的输入参数。例如，ALU的输入中，DataA是寄存器的值和移位数sa二选一，DataB是寄存器的值和立即数二选一；PC的更新，可以是PC+4，可以是跳转对应的指令条数，也可以是跳转地址。以上例子都需要依靠数据选择确定具体的参数。

在顶层模块中，将各个主要器件和数据模块组合，按数据通路图所示正确传递参数。

2. 单周期CPU设计流程：

根据实验内容中的指令说明及控制信号的作用，写出指令与控制信号之间的关系。

(见下表)

指令	zero	sign	PCWre	ALU srcA	ALU srcB	DB DataSrc	Reg Wre	InsMem RW	/RD	/WR	ExtSel	PCSrc	RegDst	ALUOp [2..0]
add	x	x	1	0	0	0	1	1	0	0	x	00	1	000
sub	x	x	1	0	0	0	1	1	0	0	x	00	1	001
addiu	x	x	1	0	1	0	1	1	0	0	1	00	0	000
andi	x	x	1	0	1	0	1	1	0	0	0	00	0	100
and	x	x	1	0	0	0	1	1	0	0	x	00	1	100
ori	x	x	1	0	1	0	1	1	0	0	0	00	0	011
or	x	x	1	0	0	0	1	1	0	0	x	00	1	011
sll	x	x	1	1	0	0	1	1	0	0	x	00	1	010
slti	x	x	1	0	1	0	1	1	0	0	1	00	0	110
sw	x	x	1	0	1	0	0	1	0	1	1	00	x	000
lw	x	x	1	0	1	1	1	1	1	0	1	00	0	000
beq	0	x	1	0	0	0	0	1	0	0	1	00	x	001
	1	x	1	0	0	0	0	1	0	0	1	01	x	001
bne	0	x	1	0	0	0	0	1	0	0	1	01	x	001
	1	x	1	0	0	0	0	1	0	0	1	00	x	001
bltz	0	0	1	0	0	0	0	1	0	0	1	00	x	001
	0	1	1	0	0	0	0	1	0	0	1	01	x	001
	1	x	1	0	0	0	0	1	0	0	1	00	x	001
j	x	x	1	0	x	x	0	1	x	x	x	10	x	xxx
halt	x	x	0	x	x	x	0	1	0	0	x	xx	x	xxx

由上表，写出ControlUnit模块。

ControlUnit:

通过条件判断给控制信号赋值。一位的控制信号依据当前操作码赋值，多位的控制信号 (PCsrc, ALUop) 按位分别赋值。当操作码，零标志或符号标志发生变化时（敏感信号），会对控制信号重新赋值。依据该表，控制模块代码如下。

```
module ControlUnit(opCode, zero, sign, PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, RD,
WR, ExtSel, RegDst, PCSrc, ALUOp);
    input [5:0] opCode;
    input zero, sign;
    output reg PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, RD, WR, ExtSel, RegDst;
    output reg [1:0] PCSrc;
    output reg [2:0] ALUOp;
    always@(opCode or zero or sign) begin
        PCWre = (opCode == 6'b111111)? 0 : 1;
        ALUSrcA = (opCode == 6'b011000)? 1 : 0;
        ALUSrcB = (opCode == 6'b000010 || opCode == 6'b010000)
    end
endmodule
```

```

    || opCode == 6'b010010 || opCode == 6'b011100 || opCode == 6'b100110
    || opCode == 6'b100111)? 1 : 0;
DBDataSrc = (opCode == 6'b100111)? 1 : 0;
RegWre = (opCode == 6'b000000 || opCode == 6'b000001 || opCode == 6'b000010 ||
           opCode == 6'b010000 || opCode == 6'b010001 || opCode == 6'b010010 ||
           opCode == 6'b010011 || opCode == 6'b011000 || opCode == 6'b011100 ||
           opCode == 6'b100111)? 1 : 0;
RD = (opCode == 6'b100111)? 1 : 0;
WR = (opCode == 6'b100110)? 1 : 0;
ExtSel = (opCode == 6'b010000 || opCode == 6'b010010)? 0 : 1;
PCSsrc[1] = (opCode == 6'b111000)? 1:0;
PCSsrc[0] = ((opCode == 6'b110000 && zero == 1)
              || (opCode == 6'b110001 && zero == 0) || (opCode == 6'b110010 && zero
              == 0 && sign==1))? 1 : 0;
RegDst = (opCode == 6'b000010 || opCode == 6'b010000 || opCode == 6'b010010
           || opCode == 6'b011100 || opCode == 6'b100111)? 0 : 1;
ALUOp[2] = (opCode == 6'b010001 || opCode == 6'b011100
              || opCode == 6'b010000)? 1 : 0;
ALUOp[1] = (opCode == 6'b010010 || opCode == 6'b010011 || opCode == 6'b011000
              || opCode == 6'b011100)? 1 : 0;
ALUOp[0] = (opCode == 6'b000001 || opCode == 6'b010010 || opCode == 6'b010011
              || opCode == 6'b110000 || opCode == 6'b110001 || opCode ==
              6'b110010)? 1 : 0;
end
endmodule

```

insROM:

指令存储器根据PC的值取得指令，再从指令中提取操作数，包括操作代码，rs，rd，rt，立即数，位移数，跳转地址。赋值使用按位赋值的方法。因为操作数在指令中的位宽和位置是固定的，这里没有判断具体的指令类型而决定哪些需要赋值，而是全部赋值，这样减少了判断过程，同时因为控制信号由操作码决定，也不影响指令的正常执行。

```

module insROM ( pc, RW, op, rs, rt, rd, sa, immediate, jumpimmediate);
    input [31:0] pc;
    input RW;//在单周期CPU中始终为0
    output reg [5:0] op;
    output reg [4:0] rs, rt, rd, sa;
    output reg [15:0] immediate;
    output reg [25:0] jumpimmediate;
    reg [7:0] rom [99:0];
    reg [31:0] dataOut;

```

```

initial begin // 加载数据到指令存储器insROM
    $readmemb ("D:/Sophomore/ECOP-2018/ECOP-17343155-02/
                CPU_sim/rom.txt", rom);
end

always @( RW or pc ) begin
    if (RW==1) begin // 将指令读出 大端方式
        dataOut[31:24] = rom[pc];
        dataOut[23:16] = rom[pc+1];
        dataOut[15:8] = rom[pc+2];
        dataOut[7:0] = rom[pc+3];
        op = dataOut [31:26];
        rs = dataOut [25:21];
        rt = dataOut [20:16];
        rd = dataOut [15:11];
        immediate = dataOut [15:0];
        sa = dataOut [10:6];
        jumpimmediate = dataOut [25:0];
    end
end
endmodule

```

ALU:

ALU模块，输入为3位功能码（结合ALU运算功能表）和两个需要运算的32位数，输出为零标志，符号标志和32位运算结果。功能码决定了进行的运算类型，运算结果为零时，零标志为1，其余情况为0，运算结果为正或零，符号标志为0，结果为负，符号标志为1。

```

module ALU(ALUopcode, rega, regb, zero, sign, result);
    input [2:0] ALUopcode;
    input [31:0] rega;
    input [31:0] regb;
    output reg zero;
    output reg sign;
    output reg [31:0] result;
    always @(* ALUopcode or regb or rega) 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; // 无符号比较
            3'b110 : if (rega<regb &&(rega[31] == regb[31])) // 有符号比较
                        result = 1;
        endcase
    end
endmodule

```

```

        else if ( rega[31]==1 && regb[31]==0)
            result = 1;
        else result = 0;
    3'b111 : result = rega ^ regb;
    default : begin
        result = 32'h00000000;
        $display ("ERROE");
    end
endcase
sign = result[31];
zero = (result==0) ? 1:0;
end
endmodule

```

extend:

立即数扩展模块，ExSel决定扩展类型，0为零扩展，1为符号扩展。

```

module extend(immediate, ExtSel, out);
    input [15:0] immediate;
    input ExtSel;
    output [31:0] out;
    assign out[15:0] = immediate;
    assign out[31:16] = ExtSel? (immediate[15]? 16'hffff : 16'h0000)//ExSel为1，符号位扩展，ExSel为0，零扩展
endmodule

```

ALU_dataA:

ALU输入A选择模块，根据控制信号选择寄存器的值或sa。

```

module ALU_dataA(input ALUsrcA, input [31:0] rega, input [4:0] sa, output [31:0] dataA);
    assign dataA = ALUsrcA ? {{27{0}},sa} : rega;//ALUsrcA为1，dataA为位移数零扩展，ALUsrcA为0，dataA为位rega
endmodule

```

ALU_dataB:

ALU输入B选择模块，根据输入信号选择寄存器的值或立即数。

```

module ALU_dataB(input ALUsrcB, input [31:0] regb, input [31:0] immediate, output [31:0]dataB);
    assign dataB = ALUsrcB ? immediate : regb;//ALUsrcB为1，dataA为扩展后的32位立即数，ALUsrcB为0，dataB为位regb
endmodul

```

regFile:

寄存器组模块，依据传入地址读相应寄存器的值，当时钟下降沿到来时将数据写入对应寄存器，RegWre是写使能信号，为1时可以写寄存器，为0时只可读，同时0号寄存器的值不可修改，恒为0。

```
module regFile(CLK, RST, RegWre, ReadReg1, ReadReg2, WriteReg, WriteData, ReadData1,
ReadData2);
    input CLK;
    input RST;
    input RegWre;
    input [4:0] ReadReg1,ReadReg2,WriteReg;
    input [31:0] WriteData;
    output [31:0] ReadData1,ReadData2;
    reg [31:0] regFile[0:31];
    integer i;
    assign ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1];
    assign ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];
    always @ (negedge CLK) begin // 必须用时钟边沿触发
        if (RST==0) begin
            for(i=1;i<32;i=i+1)
                regFile[i] <= 0;
        end
        else if(RegWre == 1 && WriteReg != 0) // WriteReg != 0, $0 寄存器不能修改
            regFile[WriteReg] <= WriteData; // 写寄存器
    end
endmodule
```

regWriteAddress:

寄存器写地址选择模块，选择需要写的寄存器是rd还是rt。

```
module regWriteAddress(input RegDst, input [4:0] rtreg, input [4:0] rdreg, output [4:0]
regwrite);
    assign regwrite = RegDst ? rdreg : rtreg;
endmodule
```

regWriteData:

寄存器写数据选择模块，选择需要写进寄存器的值来自ALU还是存储器。

```
module regWriteData(input DBDataSrc, input [31:0] dataALU, input [31:0] dataRAM, output
[31:0] datawrite);
    assign datawrite = DBDataSrc ? dataRAM : dataALU;
endmodule
```

dataRAM:

数据存储器模块，mRD信号为1时可读，mWR信号为1时可写，写时将数据写入地址值对应的存储单元，读时将地址对应的存储单元的值输出。

```
module dataRAM(clk, address, writeData, mRD, mWR, Dataout);
    input clk;
    input [31:0] address;
    input [31:0] writeData;
    input mRD;
    input mWR;
    output [31:0] Dataout;
    reg [7:0] ram [0:60];
    // 读
    assign Dataout[7:0] = (mRD==1)?ram[address + 3]:8'bz; // z为高阻态
    assign Dataout[15:8] = (mRD==1)?ram[address + 2]:8'bz;
    assign Dataout[23:16] = (mRD==1)?ram[address + 1]:8'bz;
    assign Dataout[31:24] = (mRD==1)?ram[address ]:8'bz;
    // 写
    always@( negedge clk ) begin // 用时钟下降沿触发写存储器
        if( mWR==1 ) 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
```

Jext:

跳转立即数扩展，将j指令中的立即数扩展为32位地址，最后两位为0，前四位为PC+4的前四位。

```
module Jext(pc, jimme, jpc);
    input [31:0] pc;
    input [25:0] jimme;
    output [31:0] jpc;
    assign jpc[27:2] = jimme[25:0];
    assign jpc[1:0] = 2'b00;
    assign jpc[31:28] = pc[31:28];
endmodule
```

nextPC:

确定PC的下一个值，由PCSrc决定，为0则为PC+4，即顺序执行，为1则为偏移对应的指令条数，为2则跳转至对应的地址，并将PC的下一个值输出。

```
module nextPC(pc, PCSrc, imme, jpc, nextpc);
    input [31:0] pc;
    input [1:0] PCSrc;
    input [31:0] imme;
    input [31:0] jpc;
    output reg [31:0] nextpc;
    always@(pc or PCSrc) begin
        case(PCSrc)
            2'b00:nextpc = pc + 4;
            2'b01:nextpc = pc + (imme << 2) +4;
            2'b10:nextpc = jpc;
            default:;
        endcase
    end
endmodule
```

PC:

PC模块，PC的值为指令在指令存储器中的地址，使用同步清零的方法，当上升沿到来时，Reset为0则清零，PCWre为1代表可更改，为0代表不可更改，即停机状态。

```
module PC(clk, Reset, PCWre, nextAddress, Address);
    input clk, Reset, PCWre;
    input [31:0] nextAddress;
    output reg [31:0] Address;
    always @(posedge clk)begin
        if (Reset == 0)
            Address <= 0;
        else if(PCWre==1)
            Address <= nextAddress;
    end
endmodule
```

CPU_top:

顶层模块，实例化其余模块，并将其组合起来

```
module CPU_top(
    input clk, Reset,
    output zero, sign, PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, RD, WR,
    ExtSel, RegDst,
    output [1:0] PCSrc,
    output [2:0] ALUOp,
    output [4:0] rs, rd, rt, sa, regwrite,
    output [5:0] Opcode,
    output [31:0] Pc, nextPC, Regdata1, Regdata2, WRegdata, Result, ExtOut, Jpc, Ramout,
    dataA, dataB );
    wire [15:0] immediate;
    wire [25:0] jimme;
    wire InsMemRW;
    assign InsMemRW=1;
    PC pc(clk, Reset, PCWre, nextPC, Pc);
    insROM ins(Pc, InsMemRW, Opcode, rs, rt, rd, sa, immediate, jimme);
    ControlUnit control(Opcode, zero, sign, PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre,
    RD, WR, ExtSel, RegDst, PCSrc, ALUOp);
    extend ext(immediate, ExtSel, ExtOut);
    Jext jext(Pc, jimme, Jpc);
    regWriteAddress regadd(RegDst, rt, rd, regwrite);
    regWriteData regdata(DBDataSrc, Result, Ramout, WRegdata);
    regFile regf(clk,Reset,RegWre, rs, rt, regwrite, WRegdata, Regdata1, Regdata2);
    ALU_dataA aluA(ALUSrcA, Regdata1, sa, dataA);
    ALU_dataB aluB(ALUSrcB, Regdata2, ExtOut, dataB);
    ALU alu(ALUOp, dataA, dataB, zero, sign, Result);
    dataRAM ram(clk, Result, Regdata2, RD, WR, Ramout);
    nextPC nextpc(Pc, PCSrc, ExtOut, Jpc, nextPC);
endmodule
```

3. 单周期CPU仿真

完善测试表格，写出测试代码：

(测试程序段)

地址	汇编程序	指令代码					16 进制数代码
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)		
0x00000000	addiu \$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	add \$3,\$2,\$1	000000	00010	00001	0001 1000 0000 0000	=	00411800
0x0000000C	sub \$5,\$3,\$2	000001	00011	00010	0010 1000 0000 0000	=	04622800
0x00000010	and \$4,\$5,\$2	010001	00101	00010	0010 0000 0000 0000	=	44A22000
0x00000014	or \$8,\$4,\$2	010011	00100	00010	0100 0000 0000 0000	=	4C824000
0x00000018	sll \$8,\$8,1	011000	00000	01000	0100 0000 0100 0000	=	60084040
0x0000001C	bne \$8,\$1,-2 (#,转 18)	110001	01000	00001	1111 1111 1111 1110	=	C501FFF E
0x00000020	slti \$6,\$2,4	011100	00010	00110	0000 0000 0000 0100	=	70460004
0x00000024	slti \$7,\$6,0	011100	00110	00111	0000 0000 0000 0000	=	70C70000
0x00000028	addiu \$7,\$7,8	000010	00111	00111	0000 0000 0000 0000	=	08E70008
0x0000002C	beq \$7,\$1,-2 (=,转 28)	110000	00111	00001	1111 1111 1111 1110	=	C0E1FFF E
0x00000030	sw \$2,4(\$1)	100110	00001	00010	0000 0000 0000 0100	=	98220004
0x00000034	lw \$9,4(\$1)	100111	00001	00010	0000 0000 0000 0100	=	9C290004
0x00000038	addiu \$10,\$0,-2	000010	00000	01010	1111 1111 1111 1110	=	080AFFFE
0x0000003C	addiu \$10,\$10,1	000010	01010	01010	0000 0000 0000 0001	=	094A0001
0x00000040	bltz \$10,-2(<0,转 3C)	110010	01010	00000	1111 1111 1111 1110	=	C940FFF E
0x00000044	andi \$11,\$2,2	010000	00010	01011	0000 0000 0000 0010	=	404B0002
0x00000048	j 0x00000050	111000	00000	00000	0000 0000 0001 0100	=	E0000014
0x0000004C	or \$8,\$4,\$2	010011	01000	00010	0100 0000 0000 0000	=	4D024000
0x00000050	halt	111111	00000	00000	0000000000000000	=	FC000000

(测试代码)

00001000	00000001	00000000	00001000
01001000	00000010	00000000	00000010
00000000	01000001	00011000	00000000
00000100	01100010	00101000	00000000
01000100	10100010	00100000	00000000
01001100	10000010	01000000	00000000
01100000	00001000	01000000	01000000
11000101	00000001	11111111	11111110
01110000	01000110	00000000	00000100
01110000	11000111	00000000	00000000
00001000	11100111	00000000	00001000
11000000	11100001	11111111	11111110
10011000	00100010	00000000	00000100
10011100	00101001	00000000	00000100
00001000	00001010	11111111	11111110
00001001	01001010	00000000	00000001
11001001	01000000	11111111	11111110
01000000	01001011	00000000	00000010
11100000	00000000	00000000	00010100
01001101	00000010	01000000	00000000
11111100	00000000	00000000	00000000

波形图中部分变量说明：

RegWre:寄存器组写使能，1为写

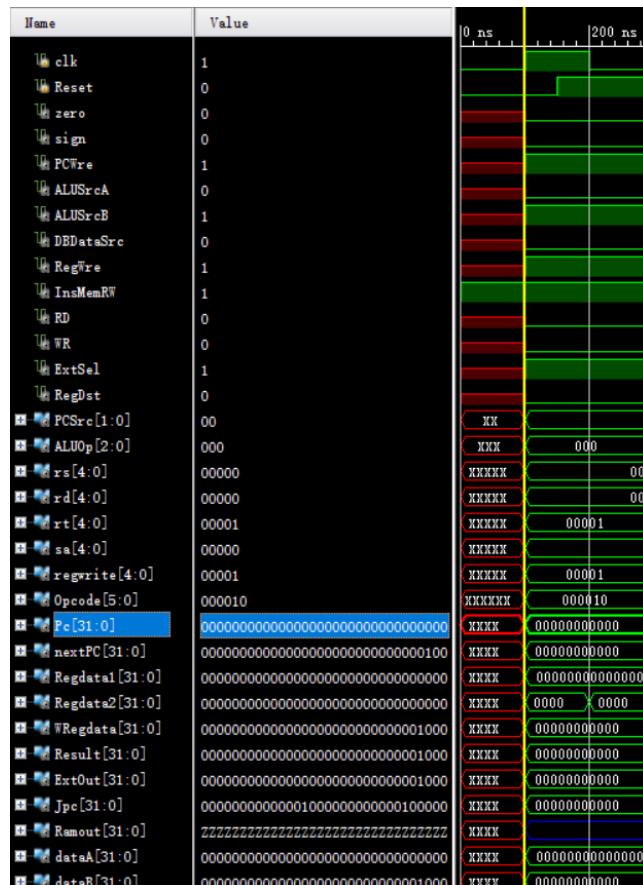
DBDataSrc:数据来源说明, 0为来自ALU运算结果

RegDst: 选择写寄存器组地址, 0写入rt, 1写入rd

Regwrite: 写寄存器组地址

WRegdata: 向寄存器组写入的数据

(1) addiu \$1,\$0,8



Pc=00,执行addiu \$1,\$0,8。操作码为000010, rt=1, rs=0, ALUOp=000。dataA来自\$0,dataB来自立即数8的符号扩展，进行加运算。结果为8，放入rt（此时\$1=8）。PCsrc=00,nextPC=04.

(2) ori \$2,\$0,2

Name	Value	
clk	1	
Reset	1	
zero	0	
sign	0	
PCWr	1	
ALUSrcA	0	
ALUSrcB	1	
DBDataSrc	0	
RegWr	1	
InsMemRW	1	
RD	0	
WR	0	
ExtSel	0	
RegDst	0	
PCSrc[1:0]	00	
ALUOp[2:0]	011	000 011
rs[4:0]	00000	00000
rd[4:0]	00000	00000
rt[4:0]	00010	00 00010
sa[4:0]	00000	
regwrite[4:0]	00010	00 00010
Opcode[5:0]	010010	00 010010
Pc[31:0]	0000000000000000000000000000100	00 0000000000000000000000000000000
nextPC[31:0]	00000000000000000000000000001000	00 0000000000000000000000000000000
Regdata1[31:0]	000000000000000000000000000000000	000000000000000000000000000000000
Regdata2[31:0]	000000000000000000000000000000000	00 0000 000000000000000000000000000
WRegdata[31:0]	000000000000000000000000000000010	00 0000000000000000000000000000000
Result[31:0]	000000000000000000000000000000010	00 0000000000000000000000000000000
ExtOut[31:0]	000000000000000000000000000000010	00 0000000000000000000000000000000
Jpc[31:0]	00000000000010000000000000000001000	00 0000000000000000000000000000000
Ramout[31:0]	ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ	
dataA[31:0]	000000000000000000000000000000000	000000000000000000000000000000000
dataB[31:0]	000000000000000000000000000000000	00 0000000000000000000000000000000

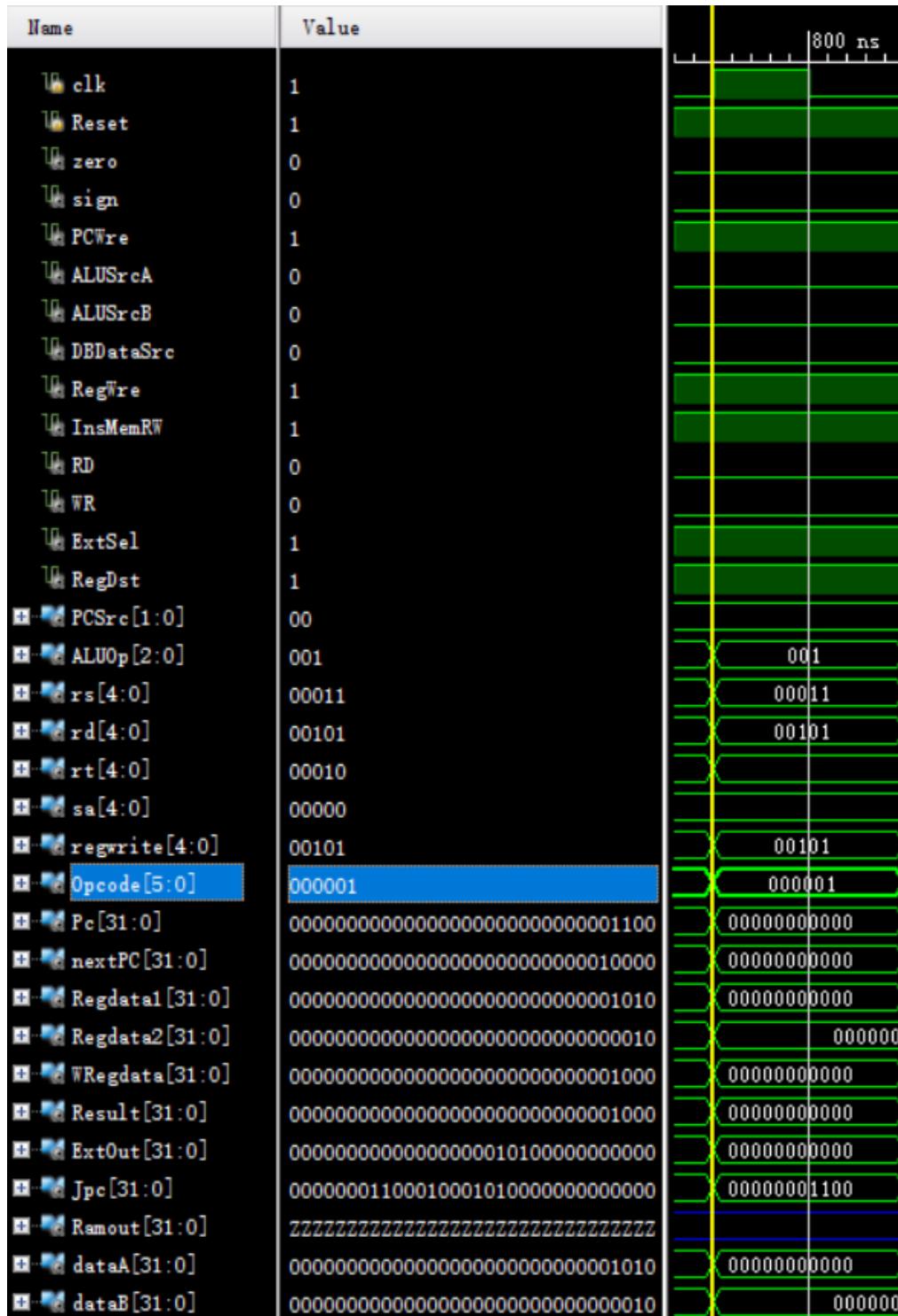
Pc=04,执行ori \$2,\$0,2。操作码为010010, rt=2, rs=0, ALUOp=011。dataA来自\$0,dataB来自立即数2的零扩展，进行或运算。结果为2，时钟下降沿到来时，写入rt(此时\$1=8,\$2=2)。PCsrc=00,nextPC=08.

(3) add \$3,\$2,\$1

Name	Value	
clk	1	
Reset	1	
zero	0	
sign	0	
PCWre	1	
ALUSrcA	0	
ALUSrcB	0	
DBDataSrc	0	
RegWre	1	
InsMemRW	1	
RD	0	
WR	0	
ExtSel	1	
RegDst	1	
PCSrc[1:0]	00	
ALUOp[2:0]	000	000
rs[4:0]	00010	00010
rd[4:0]	00011	00011
rt[4:0]	00001	00001
sa[4:0]	00000	
regwrite[4:0]	00011	00011
Opcode[5:0]	000000	000000
Pc[31:0]	00000000000000000000000000001000	000000000000
nextPC[31:0]	00000000000000000000000000001100	000000000000
Regdata1[31:0]	00000000000000000000000000001010	000000000000
Regdata2[31:0]	00000000000000000000000000001000	000000000000
WRegdata[31:0]	00000000000000000000000000001010	000000000000
Result[31:0]	00000000000000000000000000001010	000000000000
ExtOut[31:0]	00000000000000000000000011000000000000	000000000000
Jpc[31:0]	00000001000001000110000000000000	000000010000
Ramout[31:0]	ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ	
dataA[31:0]	0000000000000000000000000000000010	000000000000
dataB[31:0]	000000000000000000000000000000001000	000000000000

Pc=08,操作码为000000,rs=2,rt=1,rd=3,ALUOp=000。dataA来自rs,dataB来自rt,进行加运算。结果为10,时钟下降沿到来时,写入rd(此时\$1=8,\$2=2,\$3=10)。Pcsrc=00,nextPC=0C.

(4) sub \$5,\$3,\$2



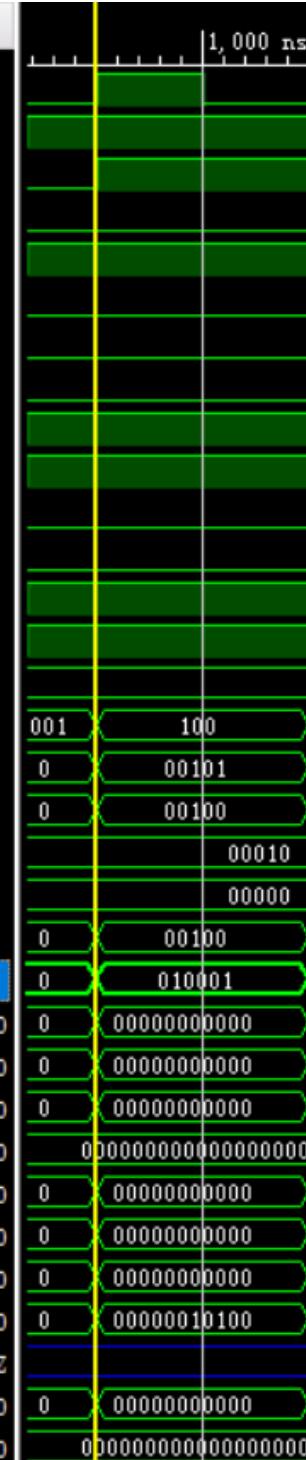
Pc=0C, 操作码为000001, rs=3, rt=2, rd=5, ALUOp=001。dataA来自rs, dataB来自rt,

进行减运算。结果为8, 时钟下降沿到来时, 写入rd(此时\$1=8,\$2=2,\$3=10,\$5=8)。

PCsrc=00,nextPC=10.

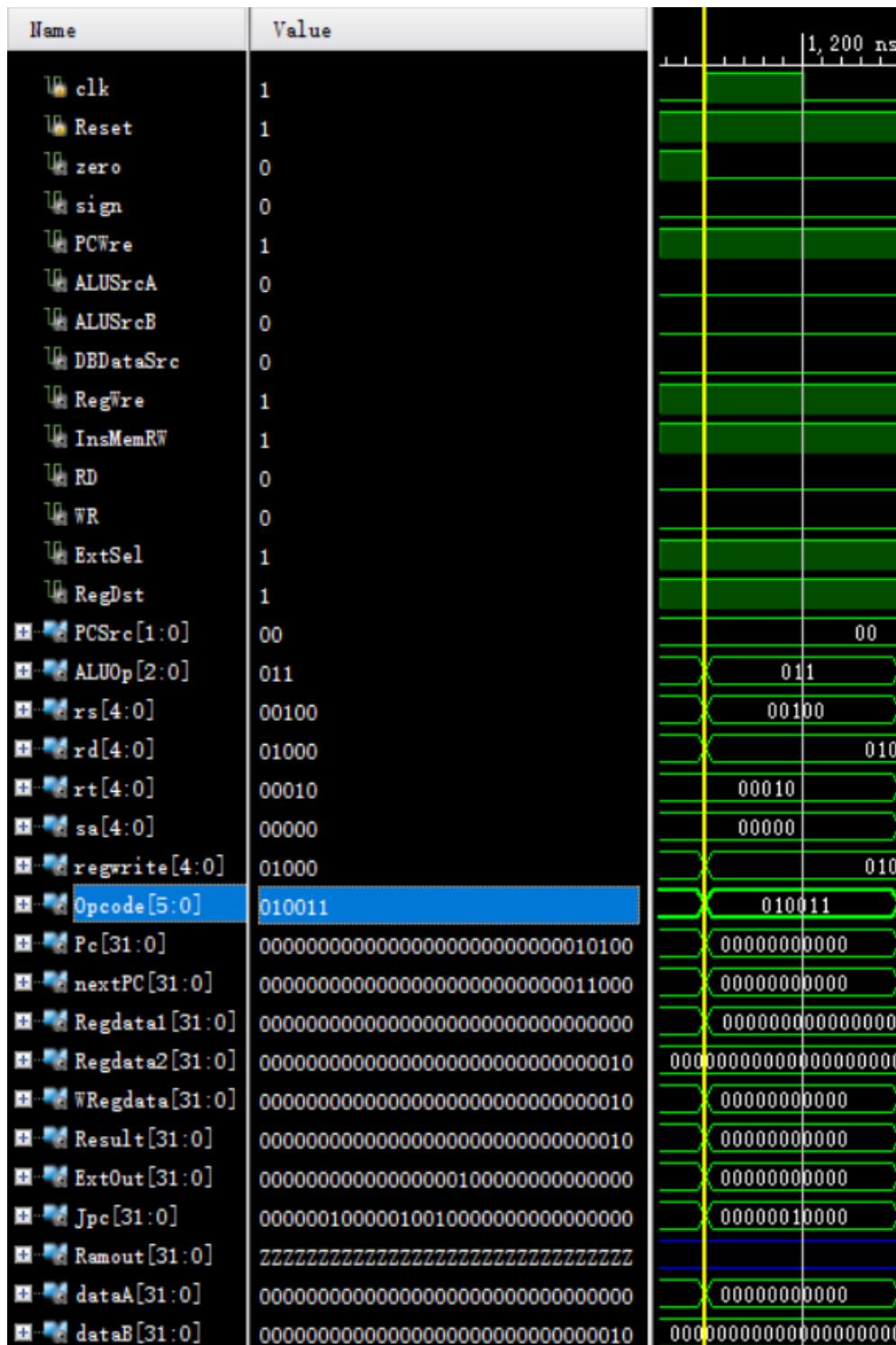
(5) and \$4,\$5,\$2

Name	Value	
clk	1	
Reset	1	
zero	1	
sign	0	
PCWre	1	
ALUSrcA	0	
ALUSrcB	0	
DBDataSrc	0	
RegWre	1	
InsMemRW	1	
RD	0	
WR	0	
ExtSel	1	
RegDst	1	
PCSsrc[1:0]	00	
ALUOp[2:0]	100	001 100
rs[4:0]	00101	0 00101
rd[4:0]	00100	0 00100
rt[4:0]	00010	00010
sa[4:0]	00000	00000
regwrite[4:0]	00100	0 00100
Opcode[5:0]	010001	0 010001
Pc[31:0]	0000000000000000000000000000000010000	0 00000000000000000000000000000000
nextPC[31:0]	0000000000000000000000000000000010100	0 00000000000000000000000000000000
Regdata1[31:0]	000000000000000000000000000000001000	0 00000000000000000000000000000000
Regdata2[31:0]	0000000000000000000000000000000000000010	0000000000000000000000000000000000000010
WRegdata[31:0]	00	0 00000000000000000000000000000000
Result[31:0]	00	0 00000000000000000000000000000000
ExtOut[31:0]	000000000000000000000000000000001000000000000000	0 00000000000000000000000000000000
Jpc[31:0]	000000101000100010000000000000000	0 000000101000
Ramout[31:0]	ZZ	
dataA[31:0]	000000000000000000000000000000001000	0 00000000000000000000000000000000
dataB[31:0]	0000000000000000000000000000000000000010	0000000000000000000000000000000000000010



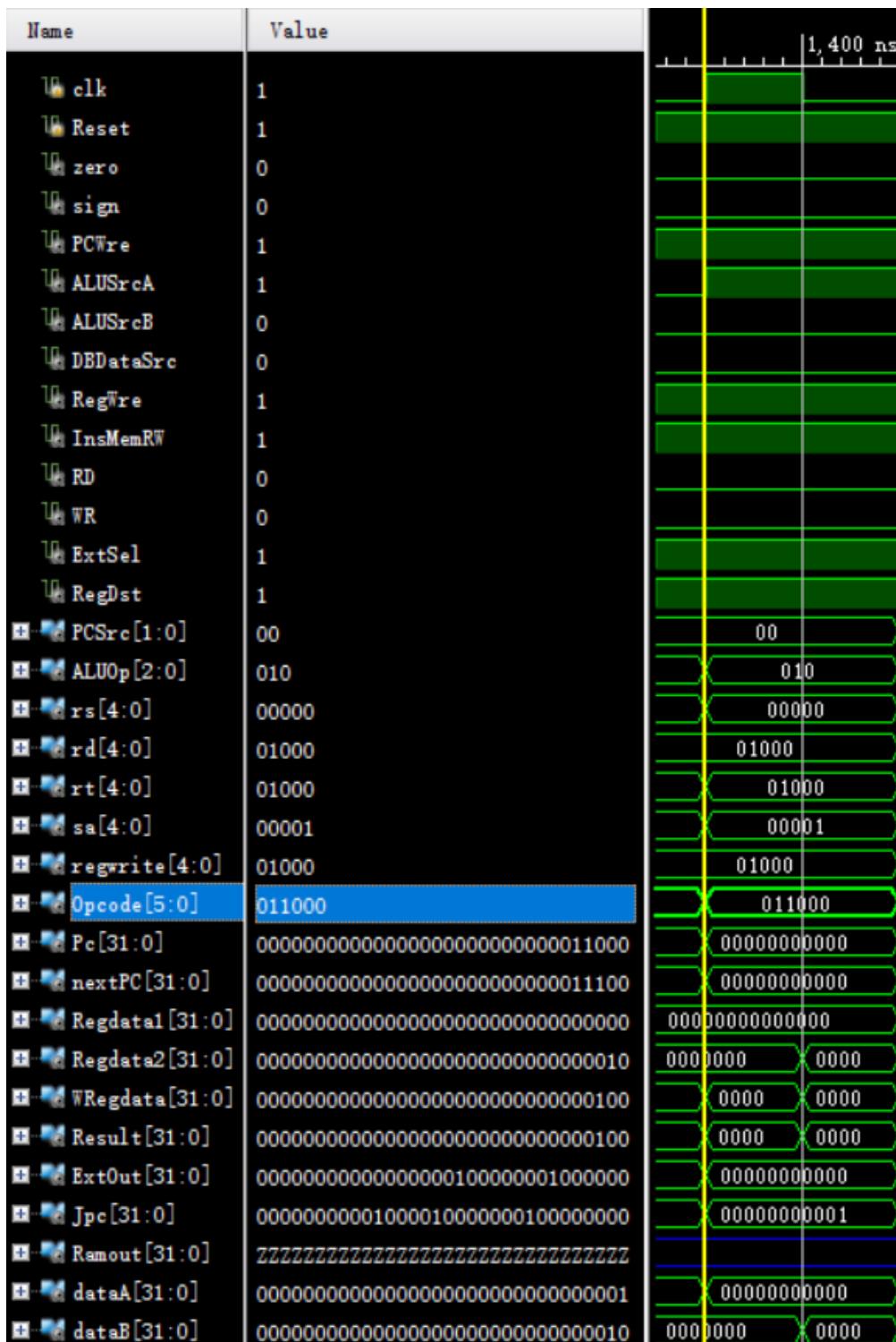
Pc=10,操作码为010001,rs=5,rt=2,rd=4,ALUOp=100。进行与运算。结果为0，时钟下降沿到来时，写入rd(此时\$1=8,\$2=2,\$3=10,\$4=0,\$5=8)。PCsrc=00,nextPC=14.

(6) or \$8,\$4,\$2



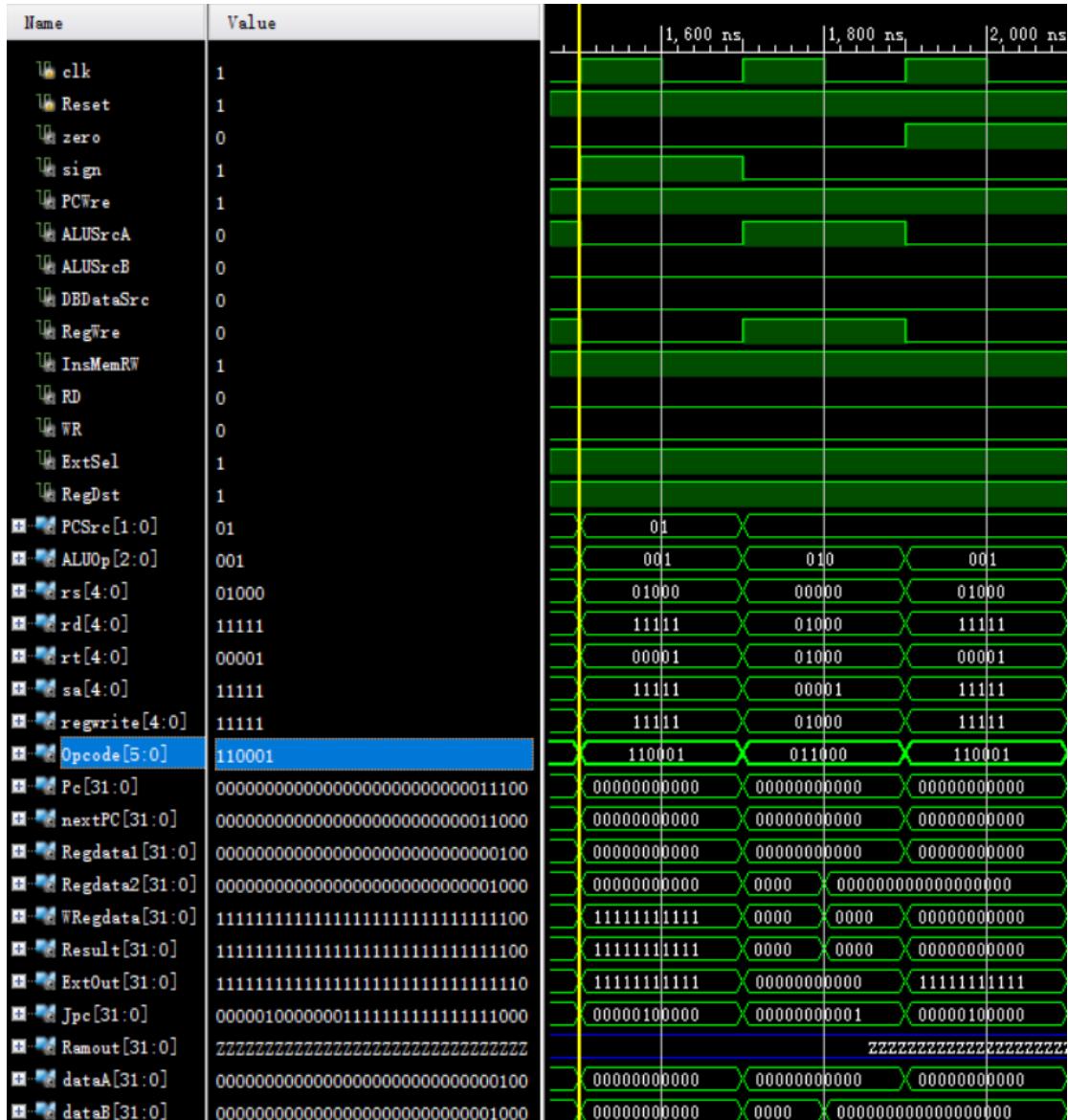
Pc=14, 操作码为010011, rs=4, rt=2, rd=8, ALUOp=011。dataA来自rs, dataB来自rt, 进行或运算。结果为2, 时钟下降沿到来时, 写入rd(此时 \$1=8, \$2=2, \$3=10, \$4=0, \$5=8, \$8=2)。PCsrc=00, nextPC=18.

(7) sll \$8,\$8,1



Pc=18, 操作码为 011000, rt=8, rd=8, ALUOp=011。dataA 来自移位数, dataB 来自 rt, 进行移位运算。结果为 4, 时钟下降沿到来时, 写入 rd(此时 \$1=8, \$2=2, \$3=10, \$4=0, \$5=8, \$8=4)。PCsrc=00, nextPC=1C.

(8) bne \$8,\$1,-2



Pc=1C, 操作码为110001, rs=8, rt=1, 立即数为-2, ALUOp=001。立即数为需要跳转的指令数, 需要进行符号扩展再左移两位, dataA来自rs, dataB来自rt, 进行减法运算。结果为-4。PCsrc=01, pc \leftarrow pc + 4 + (sign-extend)immediate <<2, nextPC=18.

(此时\$1=8,\$2=2,\$3=10,\$4=0,\$5=8,\$8=4)

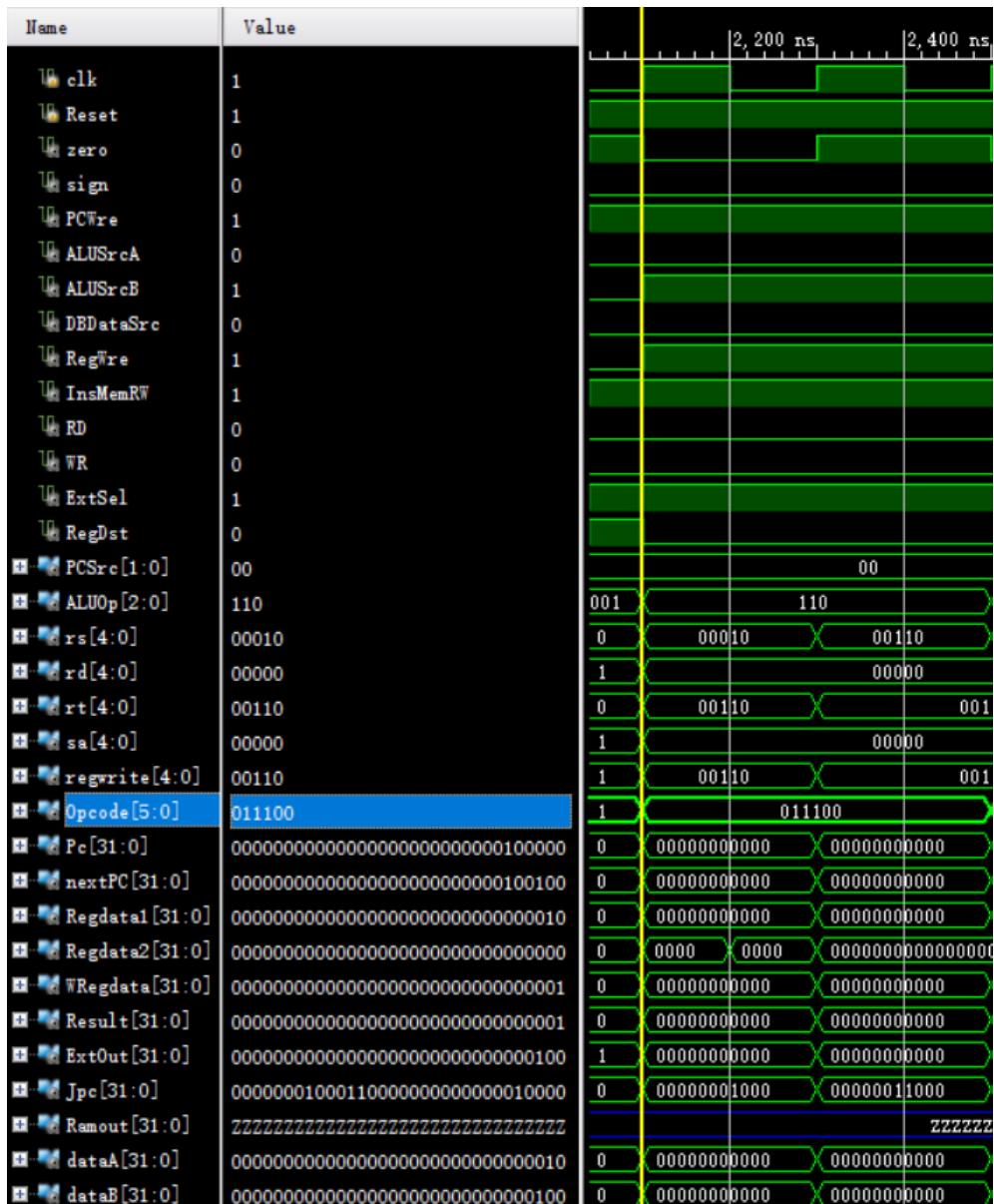
再次执行 sll \$8,\$8,1 结果为 \$1=8,\$2=2,\$3=10,\$4=0,\$5=8,\$8=8。

PCsrc=00, nextPC=1C.

第二次执行bne \$8,\$1,-2 rs=rt, PCsrc=00, nextPC=20.

(此时, \$1=8,\$2=2,\$3=10,\$4=0,\$5=8,\$8=8)

(9) slti \$6,\$2,4



Pc=20,操作码为011100, rs=2,rt=6,立即数为4,ALUOp=110。立即数进行符号扩展, rs与扩展后的符号数进行比较运算(带符号)。由于 $2 < 4$,结果为1。时钟下降沿到达时,写入rt. (此时, \$1=8,\$2=2,\$3=10,\$4=0,\$5=8,\$6=1,\$8=8) PCsrc=00,nextPC=24.

(10) slti \$7,\$6,0

(波形图在上条指令中)

Pc=24,操作码为011100, rs=6,rt=7,立即数为0,ALUOp=110。立即数进行符号扩展, rs与扩展后的符号数进行比较运算(带符号)。由于 $1 < 0$,结果为0。时钟下降沿到达时,写入rt. (此时, \$1=8,\$2=2,\$3=10,\$4=0,\$5=8,\$6=1,\$7=0,\$8=8) PCsrc=00,nextPC=28.

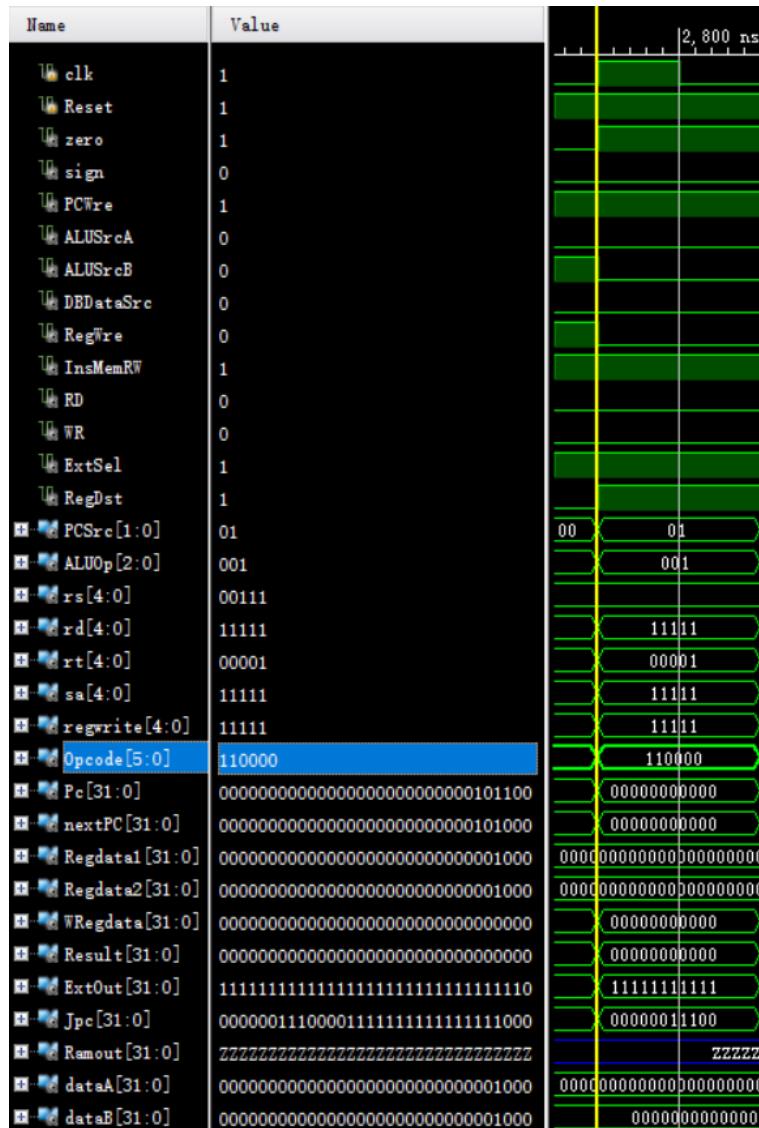
(11) addiu \$7,\$7,8

Name	Value			
clk	1			
Reset	1			
zero	0			
sign	0			
PCWre	1			
ALUSrcA	0			
ALUSrcB	1			
DEDataSrc	0			
RegWre	1			
InsMemRW	1			
RD	0			
WR	0			
ExtSel	1			
RegDst	0			
PCSsrc[1:0]	00	00		
ALUOp[2:0]	000	110	000	
rs[4:0]	00111	0		
rd[4:0]	00000		00000	
rt[4:0]	00111		00111	
sa[4:0]	00000		00000	
regwrite[4:0]	00111		00111	
Opcode[5:0]	000010	0	000010	
Pc[31:0]	0000000000000000000000000000101000	0	000000000000	
nextPC[31:0]	0000000000000000000000000000101100	0	000000000000	
Regdata1[31:0]	000000000000000000000000000000000000	0	0000	00000000
Regdata2[31:0]	000000000000000000000000000000000000	00000000		00000000
WRegdata[31:0]	000000000000000000000000000000001000	0	0000	0000
Result[31:0]	000000000000000000000000000000001000	0	0000	0000
ExtOut[31:0]	000000000000000000000000000000001000	0	00000000	0000
Jpc[31:0]	0000001110011100000000000000100000	0	00000011100	
Ramout[31:0]	ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ			
dataA[31:0]	000000000000000000000000000000000000	0	0000	00000000
dataB[31:0]	0000000000000000000000000000000000001000	0		00

Pc=28, 执行addiu \$7,\$7,8。操作码为000010, rt=7, rs=7, ALUOp=000。dataA

来自\$7,dataB来自立即数8的符号扩展，进行加运算。结果为8，放入rt（此时，
\$1=8,\$2=2,\$3=10,\$4=0,\$5=8,\$6=1,\$7=8,\$8=8).PCsrc=00,nextPC=2C.

(12) beq \$7,\$1,-2



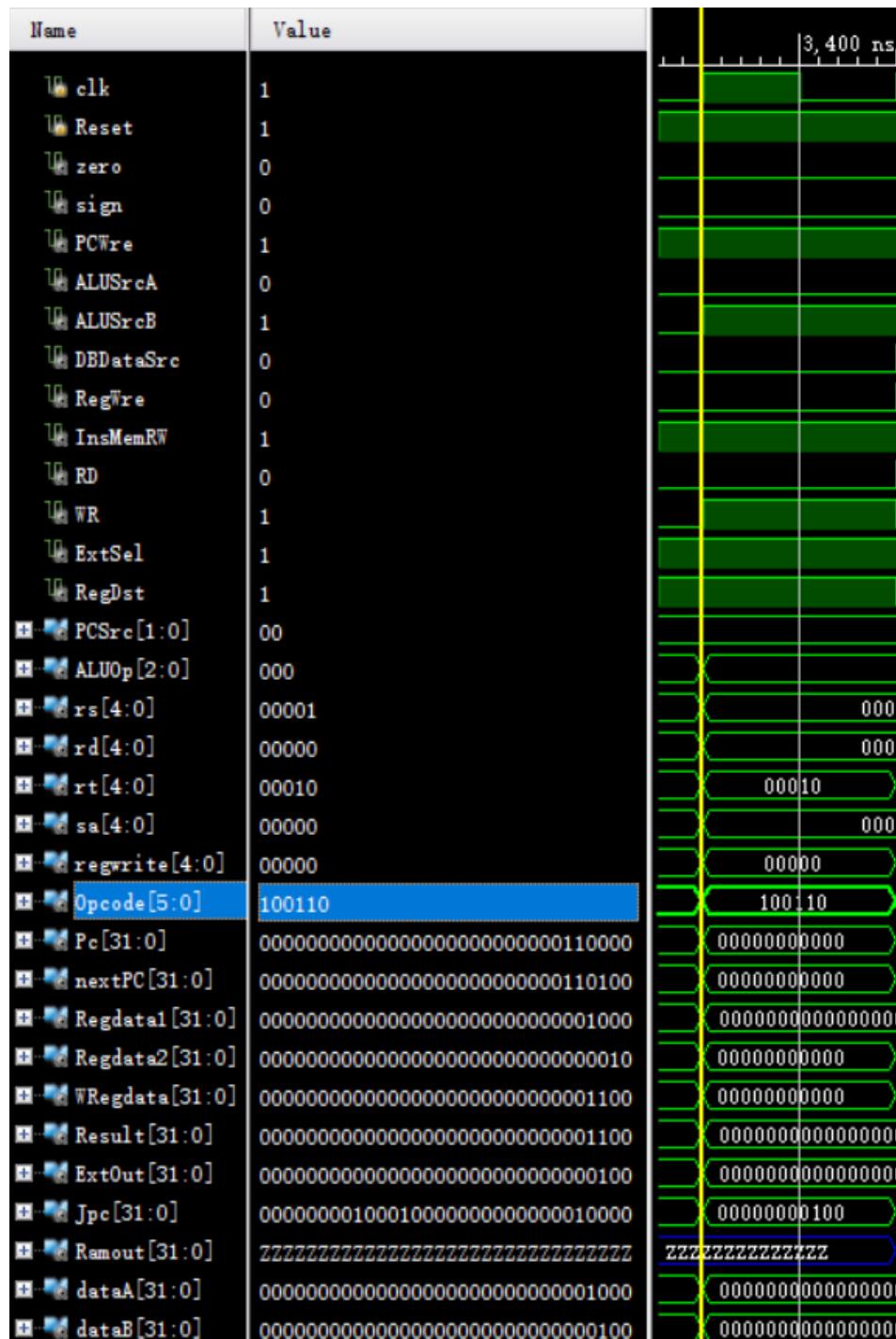
Pc=2C, 操作码为 110000, rs=7, rt=1, 立即数为 -2, ALUOp=001。立即数进行符号扩展, dataA 来自 rs, dataB 来自 rt, 进行减法运算。结果为 0。因为 rs==rt, PCsrc=01, pc←pc + 4 + (sign-extend)immediate <<2, nextPC=28.

(此时, \$1=8,\$2=2,\$3=10,\$4=0,\$5=8,\$6=1,\$7=8,\$8=8)

Pc=28, 再次执行 addiu \$7,\$7,8。操作码为 000010, rt=7, rs=7, ALUOp=000. dataA 来自 \$S7, dataB 来自 立即数 8 的符号扩展, 进行加运算。结果为 16, 放入 rt (此时, \$1=8,\$2=2,\$3=10,\$4=0,\$5=8,\$6=1,\$7=16,\$8=8). PCsrc=00, nextPC=2C.

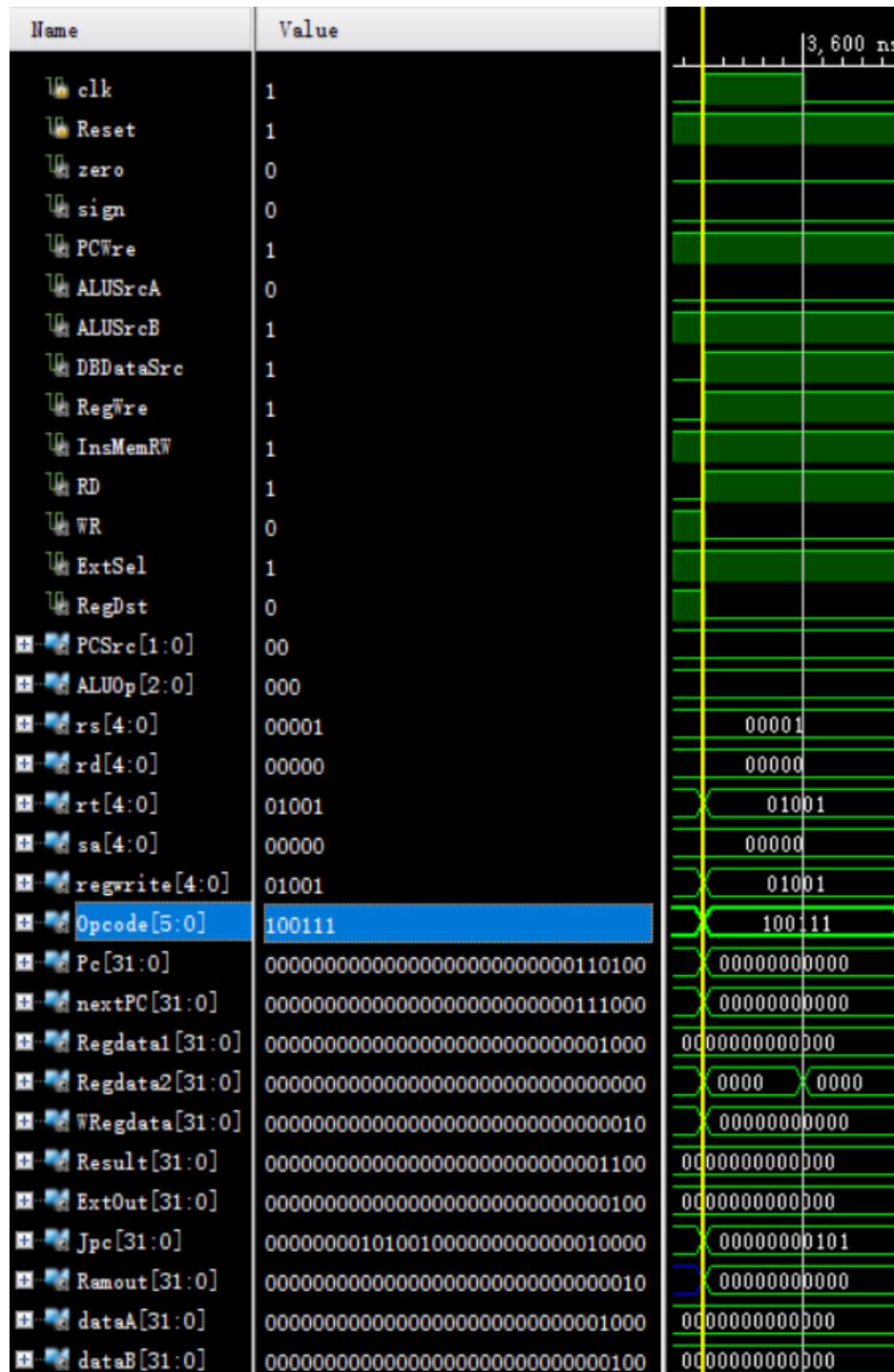
Pc=2C, 再次执行 beq \$7,\$1,-2, 操作码为 110000, rs=7, rt=1, 立即数为 -2, ALUOp=001。立即数进行符号扩展, dataA 来自 rs, dataB 来自 rt, 进行减法运算。结果为 8。因为 rs!=rt, PCsrc=00, pc←pc + 4, nextPC=30.

(13) sw \$2,4(\$1)



PC=30: 指令为sw \$2,4(\$1)，操作码为100110，rs=1，rt=2，立即数为4，存储器可写，ALU操作码为000，DataA来自\$1，DataB来自符号扩展的立即数，执行加法操作，结果为12，即需要写的存储器地址为12，值为Regdata2即\$2的值，时钟下降沿到来时，写入存储器中，PC的下一个值为PC+4。

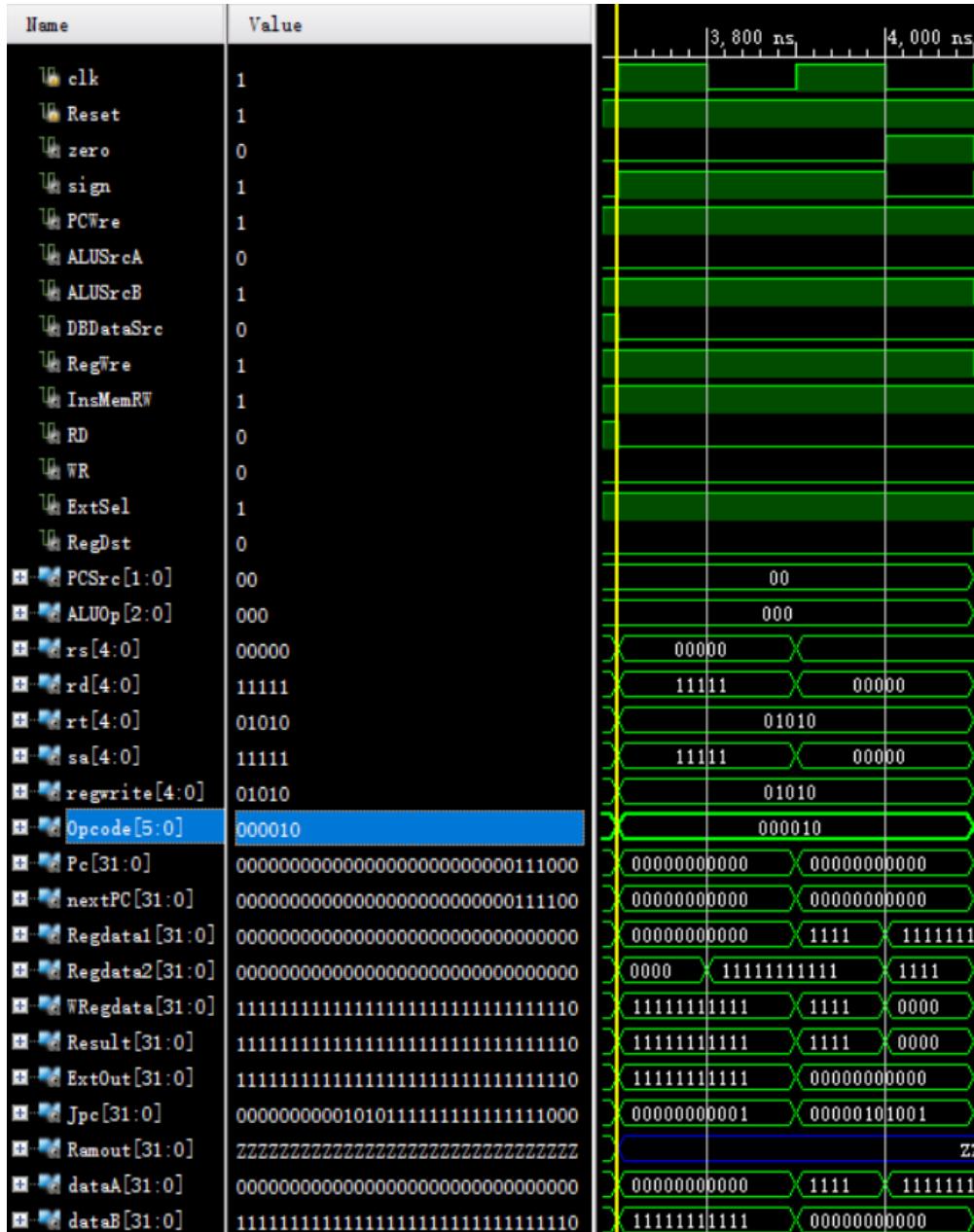
(14) lw \$9,4(\$1)



PC=34: 指令为lw \$9,4(\$1), 操作码为100111, rs=1, rt=9, 立即数为4, 存储器可读, 寄存器可写, ALU操作码为000, DataA来自\$1, DataB来自符号扩展的立即数, 执行加法操作, 结果为12, 即需要读的存储器地址为12, 时钟下降沿到来时, 将该存储单元的值即Ramout写入\$9(regwrite)中, PC的下一个值为PC+4。

(此时, \$1=8,\$2=2,\$3=10,\$4=0,\$5=8,\$6=1,\$7=8,\$8=8,\$9=2)

(15) addiu \$10,\$0,-2



Pc=38, 执行addiu \$10,\$0,-2。操作码为000010, rt=10, rs=0, ALUOp=000.

dataA来自rs,dataB来自立即数-2的符号扩展, 进行加运算。结果为-2, 放入rt (此时,
\$1=8,\$2=2,\$3=10,\$4=0,\$5=8,\$6=1,\$7=8,\$8=8,\$9=2.\$10=-2).

PCsrc=00,nextPC=3C.

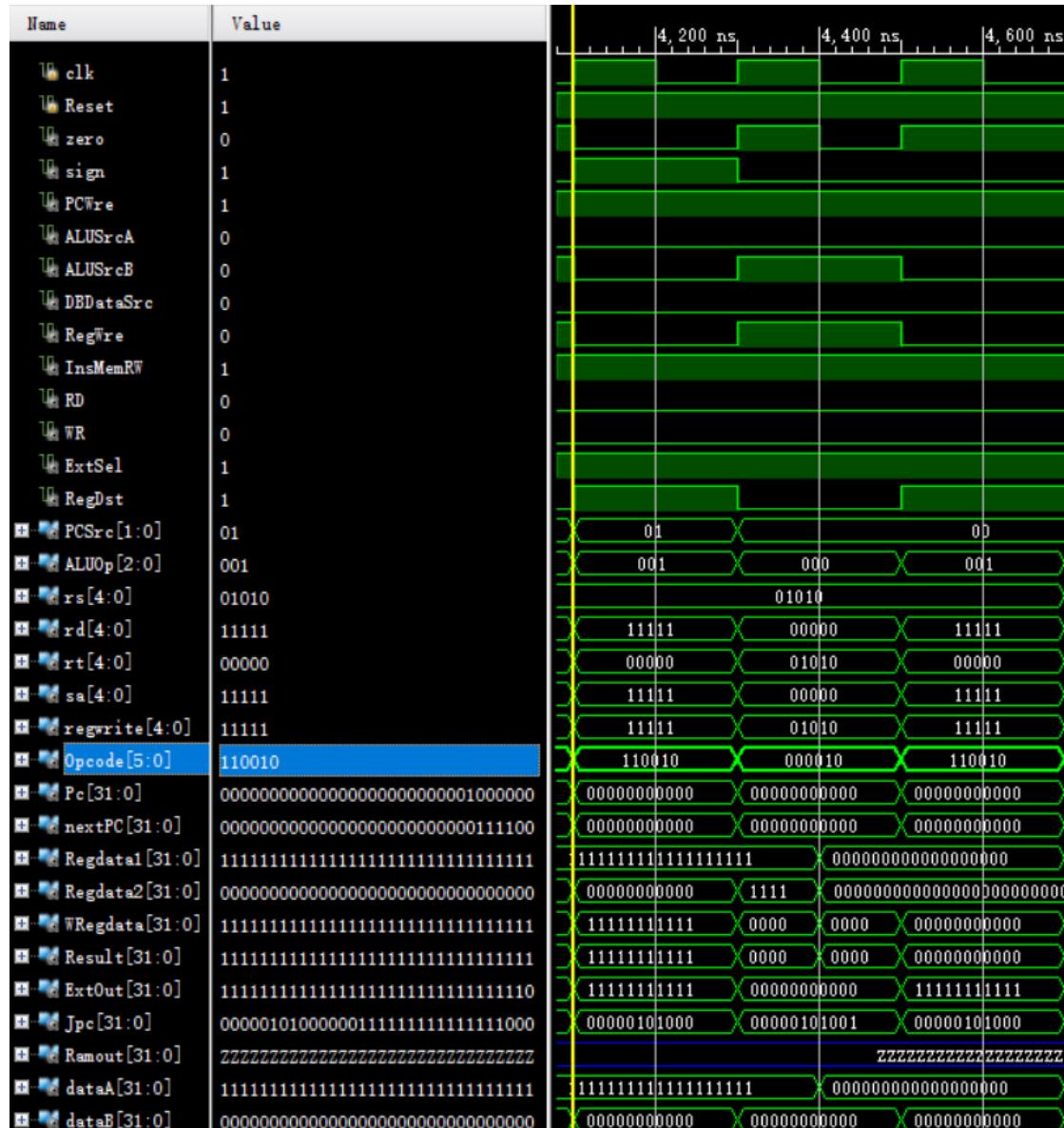
(16) addiu \$10,\$10,1

(波形图在上条指令中)

Pc=3C, 执行addiu \$10,\$10,1。操作码为000010, rt=10, rs=10, ALUOp=000.

dataA来自rs,dataB来自立即数1的符号扩展，进行加运算。结果为-1，放入rt（此时，\$1=8,\$2=2,\$3=10,\$4=0,\$5=8,\$6=1,\$7=8,\$8=8,\$9=2.\$10=-1).
 PCsrc=00,nextPC=40.

(17) bltz \$10,-2(<0,转3C)



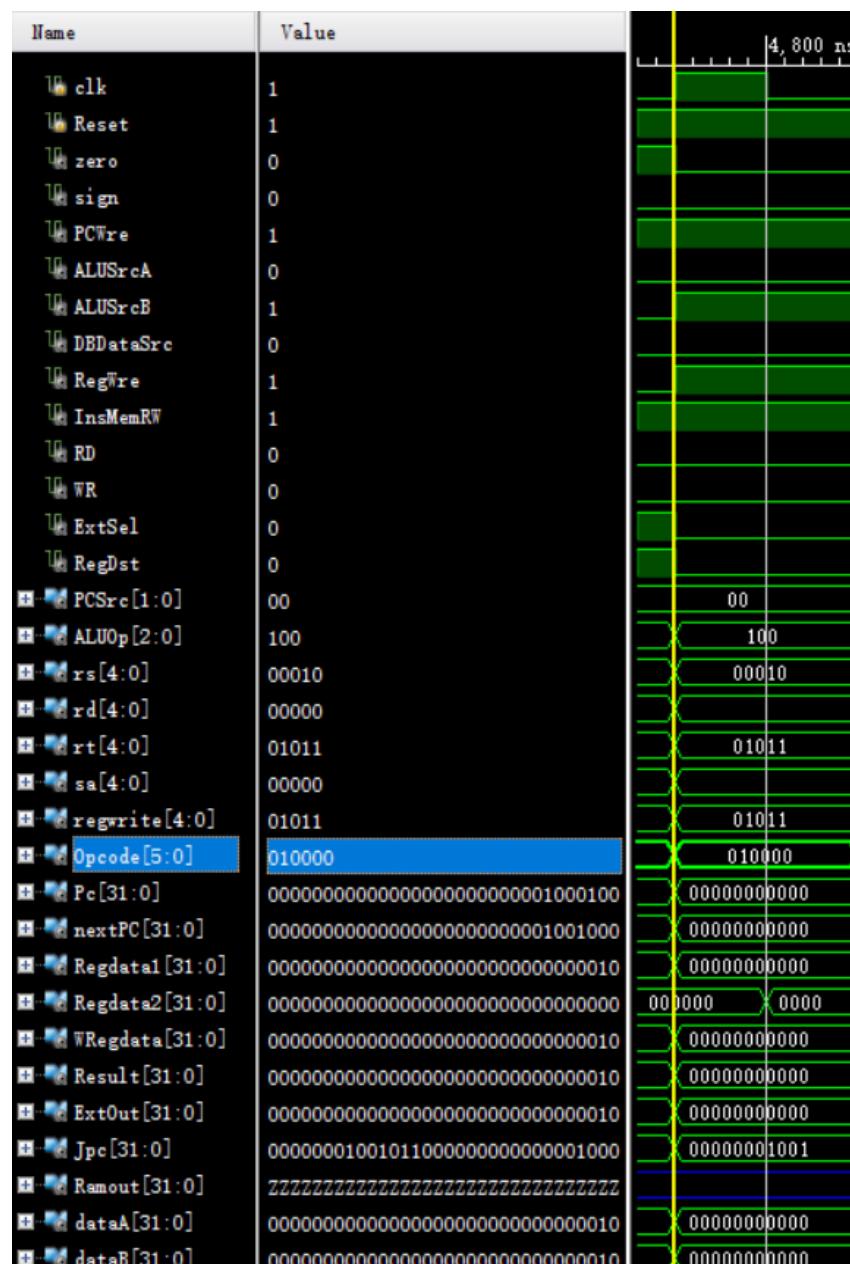
Pc=40,操作码为110010, rs=10,立即数为-2,ALUOp=001。立即数进行符号扩展，dataA来rs,dataB为0,进行减法运算。结果为-1。因为rs<0,PCsrc=01, pc←pc + 4 + (sign-extend)immediate <<2, nextPC=3C.

(此时, \$1=8,\$2=2,\$3=10,\$4=0,\$5=8,\$6=1,\$7=8,\$8=8,\$9=2.\$10=-1)

Pc=3C, 再次执行 addiu \$10,\$10,1。操作码为 000010 , rt=10 , rs=10 , ALUOp=000。dataA来自rs,dataB来自立即数1的符号扩展, 进行加运算。结果为0, 放入rt (此时, \$1=8,\$2=2,\$3=10,\$4=0,\$5=8,\$6=1,\$7=8,\$8=8,\$9=2.\$10=0).
PCsrc=00,nextPC=40.

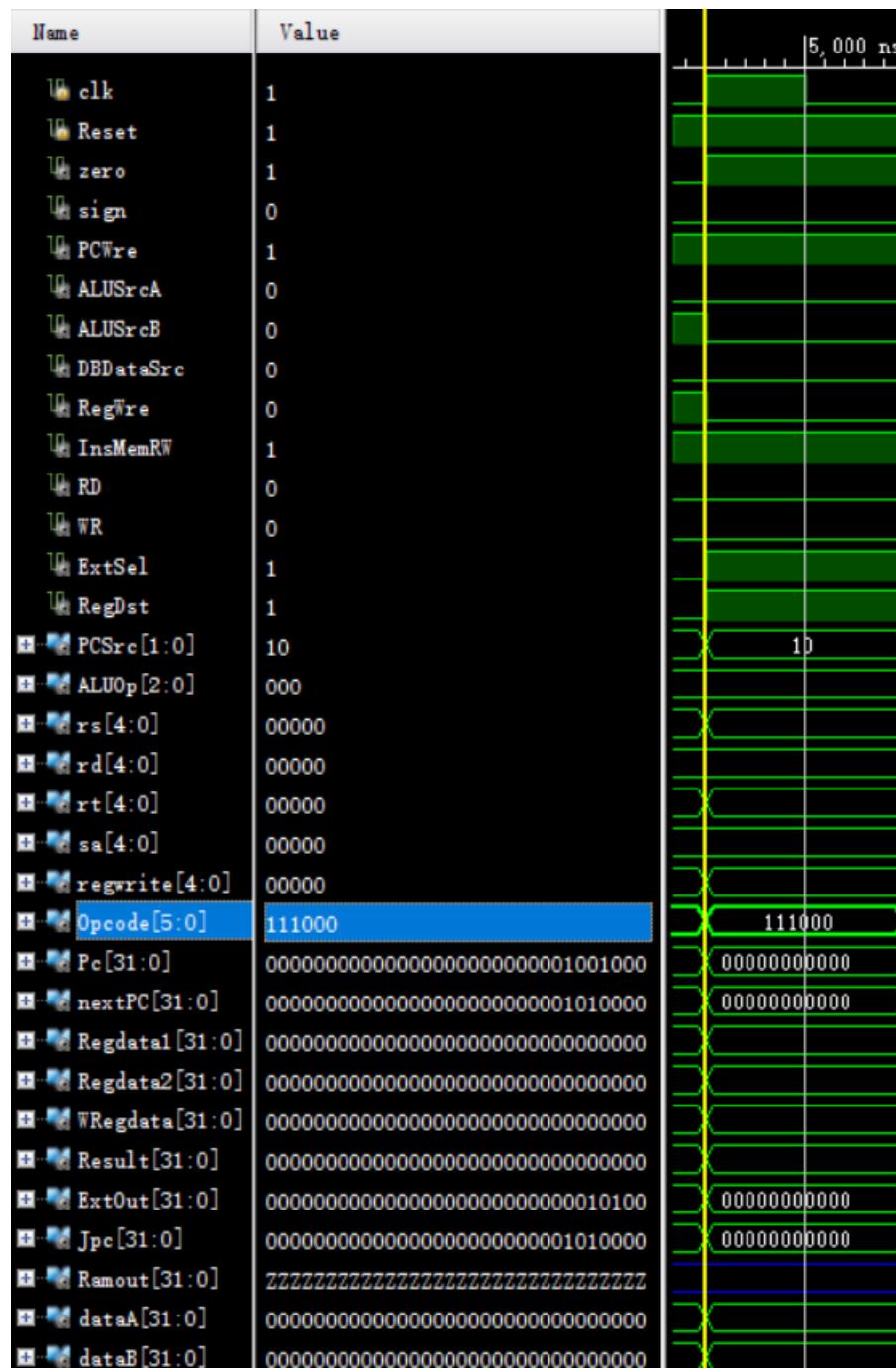
Pc=40,操作码为 110010, rs=10,立即数为-2,ALUOp=001。立即数进行符号扩展, dataA 来rs,dataB 为 0, 进行减法运算。结果为 0。因为 rs==0,PCsrc=00,pc=pc+4, nextPC=44.

(18) andi \$11,\$2,2



Pc=44, 执行andi \$11,\$2,2。操作码为010000, rt=11, rs=2, ALUOp=100。dataA 来自rs,dataB来自立即数2的零扩展，进行与运算。结果为2，放入rt（此时，\$1=8,\$2=2,\$3=10,\$4=0,\$5=8,\$6=1,\$7=8,\$8=8,\$9=2,\$10=0,\$11=2)PCsrc=00,next PC=48.

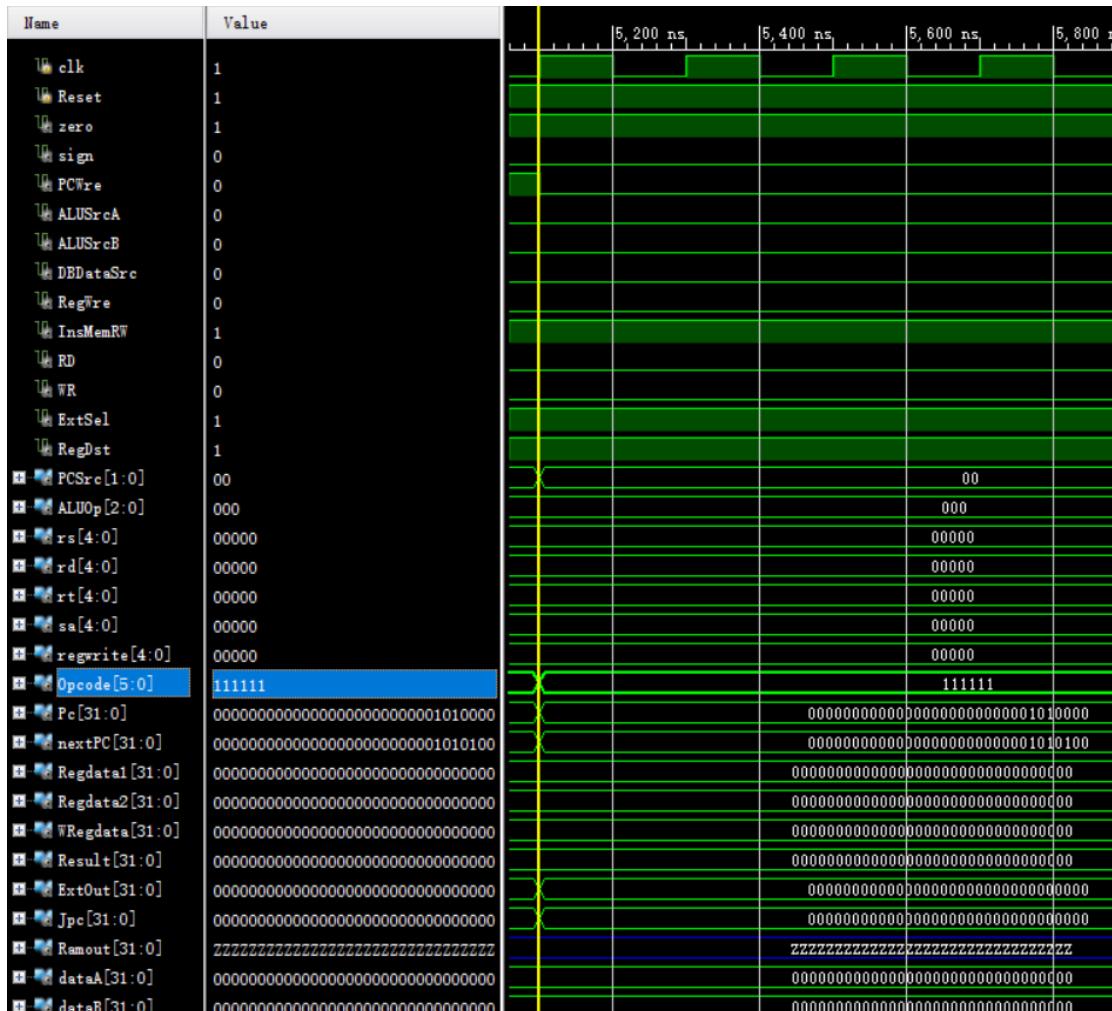
(19) j 0x00000050



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

PC=48,操作码为111000,立即数为20,将立即数左移2位,得到80,十六进制表示为50,再与PC最高四位拼接,得到跳转地址为50,PC的下一个值为50。

(20) halt



PC=50: 指令为halt, PCwre=0, PC不再变化, 停机。

4.单周期CPU实现

(1) 实现思路及代码

分频器模块, 利用计数器计数, 得到190Hz时钟信号用于驱动数码管和消抖采样。

```
module CLK_div(
    input clk,
    output clk_190hz);
    reg [17:0] q;
    always @ (posedge clk) begin
        q<=q+1;
    end
    assign clk_190hz = q[17];
endmodule
```

按键消抖模块，依据时钟信号采样，三个样点一致时，可视为信号稳定。

```
module debouncing(clk,key_in,key_out);
    input clk;
    input key_in;
    output key_out;
    reg dout1,dout2,dout3;
    assign key_out = dout1 | dout2 | dout3;
    always@(posedge clk) begin
        dout1<=key_in;
        dout2<=dout1;
        dout3<=dout2;
    end
endmodule
```

数据选择模块，选择需要显示的数据，四个数码管最多显示四个十六进制数，也就是16位，将显示数据按位赋值给data，前八位与后八位分别对应一个数据。

```
module show( Sel, PC, NextPC ,rs, ReadData1, rt, ReadData2, Result, DB, Out);
    input [1:0] Sel;
    input [4:0] rs,rt;
    input [31:0] PC, NextPC, ReadData1, ReadData2, Result, DB;
    output reg [15:0] Out;
    always @ (*)begin
        case(Sel)
            0: begin
                Out[15:8] = PC[7:0];
                Out[7:0] = NextPC[7:0];
            end
            1: begin
                Out[15:8] = {3'b000,rs[4:0]};
                Out[7:0] = ReadData1[7:0];
            end
            2: begin
                Out[15:8] = {3'b000,rt[4:0]};
                Out[7:0] = ReadData2[7:0];
            end
            3:begin
                Out[15:8] = Result[7:0];
                Out[7:0] = DB[7:0];
            end
        endcase
    end
endmodule
```

数码管显示模块，使用滚动扫描的方法，将输入的16位信号显示出来，每个时钟周期内只有一个数码管亮，时钟周期合适的话，利用视觉残留，实现同时显示的效果，每个显示的16进制数都要译码，得到驱动每一段数码管的信号。

```

module Hex7seg(
    input wire [15:0] data,
    input clk,
    output reg [6:0] a_to_g,
    output reg [3:0] dig);
    reg [1:0] sel;
    reg [3:0] num;
    initial
        sel = 0;
    always @ (posedge clk) begin
        sel = sel+1;
    end
    always @ (sel) begin //滚动显示四个数字
        case(sel)
            0: num = data[3:0];
            1: num = data[7:4];
            2: num = data[11:8];
            3: num = data[15:12];
            default: num = data[3:0];
        endcase
    end
    always @ (sel) begin //使四个数码管循环亮起，每次只亮一个
        case(sel)
            0: dig = 4'b0111;
            1: dig = 4'b1011;
            2: dig = 4'b1101;
            3: dig = 4'b1110;
        endcase
    end
    always @ (num) begin //将显示的数字译码，得到驱动每段数码管的信号
        case(num)
            0: a_to_g = 7'b0000001;
            1: a_to_g = 7'b1001111;
            2: a_to_g = 7'b0010010;
            3: a_to_g = 7'b0000110;
            4: a_to_g = 7'b1001100;
            5: a_to_g = 7'b0100100;
            6: a_to_g = 7'b0100000;
            7: a_to_g = 7'b0001111;
            8: a_to_g = 7'b0000000;
        endcase
    end
endmodule

```

```

9: a_to_g = 7'b0000100;
10:a_to_g = 7'b0001000;
11:a_to_g = 7'b1100000;
12:a_to_g = 7'b0110001;
13:a_to_g = 7'b1000010;
14:a_to_g = 7'b0110000;
15:a_to_g = 7'b0111000;
default: a_to_g = 7'b1111111;

endcase
end
endmodule

```

实现文件顶层模块，将原顶层模块和在数码管显示模块实例化并组合。

```

module imp_top(
    input clk,
    input step,
    input Reset,
    input [1:0] Sel,
    output [6:0] atog,
    output [3:0] enlight);
    wire clk190hz,key_out,zero, sign, PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre,
    InsMemRW, RD, WR, ExtSel, RegDst;
    wire [1:0] PCSrc;
    wire [2:0] ALUOp;
    wire [4:0] rs, rd, rt, sa, regwrite;
    wire [5:0] Opcode;
    wire [15:0] data;
    wire [31:0] Pc, nextPC, Regdata1, Regdata2, WRegdata, Result, ExtOut, Jpc, Ramout,
    dataA, dataB;
    CLK_div div(clk, clk190hz);
    debouncing key(clk190hz, step, key_out);
    show out(Sel, Pc, nextPC, rs, Regdata1, rt, Regdata2, Result, WRegdata, data);
    Hex7seg hex(data, clk190hz, atog, enlight);
    CPU_top cputop(key_out, Reset, zero, sign, PCWre, ALUSrcA, ALUSrcB, DBDataSrc,
    RegWre, InsMemRW, RD, WR, ExtSel, RegDst, PCSrc, ALUOp, rs, rd, rt, sa, regwrite, Opcode,
    Pc, nextPC, Regdata1, Regdata2, WRegdata, Result, ExtOut, Jpc, Ramout, dataA, dataB);
endmodule

```

(2) 结果

前六条指令结果如下：

(图片从左到右依次为：当前指令地址PC:下条指令地址PC；RS寄存器地址:RS寄存器数据；RT寄存器地址:RT寄存器数据；ALU结果输出 :DB总线数据)

(1)



Pc=00, 执行addiu \$1,\$0,8。操作码为000010, rt=1, rs=0, ALUOp=000。dataA来自\$0,dataB来自立即数8的符号扩展，进行加运算。结果为8，放入rt（此时\$1=8）。PCsrc=00,nextPC=04.

(2)



Pc=04, 执行ori \$2,\$0,2。操作码为010010, rt=2, rs=0, ALUOp=011。dataA来自\$0,dataB来自立即数2的零扩展，进行或运算。结果为2，时钟下降沿到来时，写入rt(此时\$1=8,\$2=2)。PCsrc=00,nextPC=08.

(3)



Pc=08, 操作码为000000,rs=2,rt=1,rd=3,ALUOp=000。dataA来自rs,dataB来自rt,进行加运算。结果为10，时钟下降沿到来时，写入rd(此时\$1=8,\$2=2,\$3=10)。PCsrc=00,nextPC=0C.

(4)



Pc=0C, 操作码为000001,rs=3,rt=2,rd=5,ALUOp=001。dataA来自rs,dataB来自rt,进行减运算。结果为8，时钟下降沿到来时，写入rd(此时\$1=8,\$2=2,\$3=10,\$5=8)。PCsrc=00,nextPC=10.

(5)



Pc=10, 操作码为010001,rs=5,rt=2,rd=4,ALUOp=100。进行与运算。结果为0，时钟下降沿到来时，写入rd(此时\$1=8,\$2=2,\$3=10,\$4=0,\$5=8)。PCsrc=00,nextPC=14.

(6)



Pc=14, 操作码为010011, rs=4, rt=2, rd=8, ALUOp=011。dataA来自rs, dataB来自rt, 进行或运算。结果为2, 时钟下降沿到来时, 写入rd(此时 \$1=8, \$2=2, \$3=10, \$4=0, \$5=8, \$8=2)。PCsrc=00, nextPC=18.

六. 实验心得

在实验中遇到的问题及解决方法:

在实验中遇到了大大小小的许多问题, 下面仅列举一些比较特殊的错误。

仿真时, 发现许多变量都处于不确定状态。经分析, 发现是在仿真文件中给reset=1的赋值过早(早于第一个时钟上升沿)导致无法初始化。正确做法应为在第一个时钟上升沿到来时reset=0, 进行初始化, 在第二个时钟上升沿到来前将reset置为1。

在仿真时, 遇到Regdata2在遇到部分指令时波形图会出现状态不确定(XXXX)的情况。经分析代码, 发现是在寄存器组模块中, 没有在刚开始(reset=0)为寄存器初始化所致。

仿真时, 在实例化regFile时, 由于参数传递顺序错误, 导致仿真结果错误。

在烧板时, 遇到pc=18和pc=1C两句重复执行多次(64次), 才能跳转到pc=20。在对照仿真波形图发现, 是因为没有按照波形图操作, 导致一些控制信号错误(由于是使用过程中的操作所致, 无法具体确定是哪个参数错了), 即第一个时钟上升沿过后, reset才由0变为1。正确操作, 按住step按钮(T17), 拨动reset开关(V17), 再松开step按钮。

感悟及收获:

本次实验花费了将近三周时间, 是一项难度较高的实验。一方面, 对于Verilog语言不是很熟悉, 所以打代码的进度较慢; 另一方面, 由于模块较多, 各个模块间相互关联, 调试时比较费精力。此外, CPU的构造及各个模块的关联也需要花一定的时间去理解。

在结合指令说明和控制信号填写真值表时, 由于指令和控制信号较多, 导致一直出现串行错误; 写指令时, 由于是二进制代码, 都是0、1, 难以找出错误。

在写代码时, 因为对Verilog语法不熟悉, 并且实验涉及到的变量个数较多, 模块调用时出现了各种错误, 比如忘记写输出类型, 实例化时, 变量顺序错误, 忘记声明部分变量的

位宽。此外，也有一些语法问题，比如在always语句中对wire类型变量赋值，使用assign对reg类型变量赋值，条件判断和位扩展时数的表示不正确，敏感变量没有写全等。

仿真过程中，波形出了较多错误。这让我学会了通过波形图debug的方法。即在波形图中找出与预期值不同的控制信号，返回顶层模块中，通过实例化语句，找出该控制信号对应的产生模块和作为参数使用的模块，这样可以较快地确定bug位置，节省时间。

在使用vivado烧板过程中，用到了以前数电实验课学到的知识。其中时钟分频和数字显示的模块参考了上个学期数电实验中的代码，节约了一些时间，使烧板过程相对顺利。

虽然实验中遇到的问题和困难很多，但是当自己一点一点地学习，一点一点地debug，最终完成实验后，有很大的成就感和满足感。

这次实验使我受益匪浅。在实验中，我更好地掌握了Verilog语言，并对CPU的构造及工作原理有了更深的理解。此次实验既加深了我对理论知识的理解，又提高了我的动手能力，让我学到了很多。