《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）：17 计教学 2 班

学生姓名 ：王程钥

学号 ：17341146

时间 ：2018 年 12 月 12 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图的构成、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：

==>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）addiu 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）and rd , rs , rt

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

功能：rd←rs & rt；逻辑与运算。

（5）andi rt , rs ,**immediate**

010001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs & (zero-extend)**immediate**；**immediate** 做“0”扩展再参加“与”运算。

（6）ori rt , rs ,**immediate**

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

功能：rt←rs | (zero-extend)**immediate**；**immediate** 做“0”扩展再参加“或”运算。

（7）xori rt , rs , **immediate**

010011	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs ⊕ (zero-extend)**immediate**；**immediate** 做“0”扩展再参加“异或”运算。

==>移位指令

（8）sll　rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved

功能：rd<－rt<<(zero-extend)sa，左移 sa 位 ，(zero-extend)sa。

==>比较指令

（9）slti　rt, rs,immediate　　带符号

100110	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。

（10）slt　rd, rs, rt　　带符号

100111	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。

==>存储器读写指令

（11）sw　rt, **immediate**(rs)

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

功能：memory[rs+ (sign-extend)**immediate**]<－rt。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

（12）lw　rt, **immediate**(rs)

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

功能：rt <－ memory[rs + (sign-extend)**immediate**]。即读取 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数，然后保存到 rt 寄存器中。

==>分支指令

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

110100	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if(rs=rt) pc <－pc + 4 + ((sign-extend)**immediate** <<2) else pc <－pc + 4。

（14) 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 <－pc + 4 + ((sign-extend)**immediate** <<2) else pc <－pc + 4。

（15）bltz　rs,**immediate**

110110	rs(5 位)	00000	**immediate**

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

==>跳转指令

（16）j　addr

111000	addr[27:2]

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

说明：由于 MIPS32 的指令代码长度占 4 个字节，所以指令地址二进制数最低 2 位均

为 0，将指令地址放进指令代码中时，可省掉！这样，除了最高 6 位操作码外，还有 26 位可用于存放地址，事实上，可存放 28 位地址，剩下最高 4 位由 pc+4 最高 4 位拼接上。

（17）jr　rs

111001	rs(5 位)	未用	未用	reserved

功能：pc <－ rs，跳转。

==>调用子程序指令

（18）jal　addr

111010	addr[27:2]

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

==>停机指令

（19）halt (停机指令)

111111	00000000000000000000000000(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。

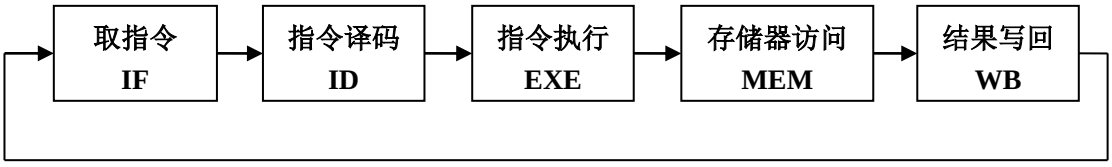


图 1　多周期 CPU 指令处理过程

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。

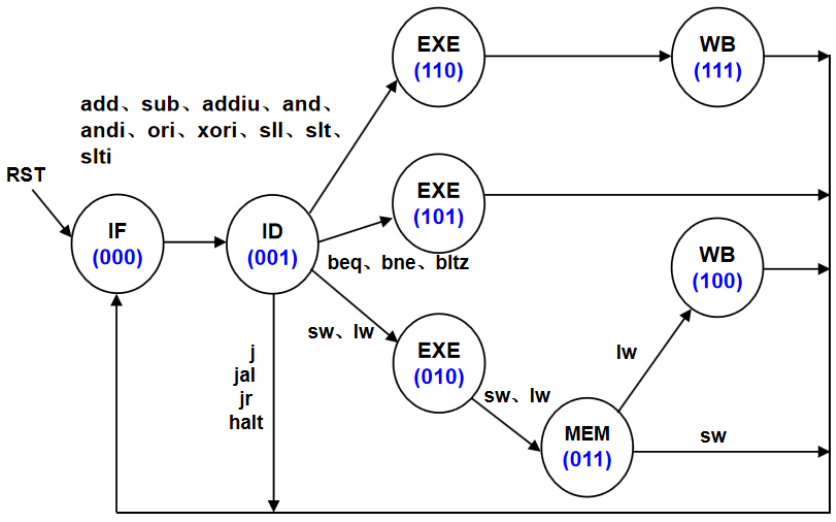


图 2 多周期 CPU 状态转移图

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。

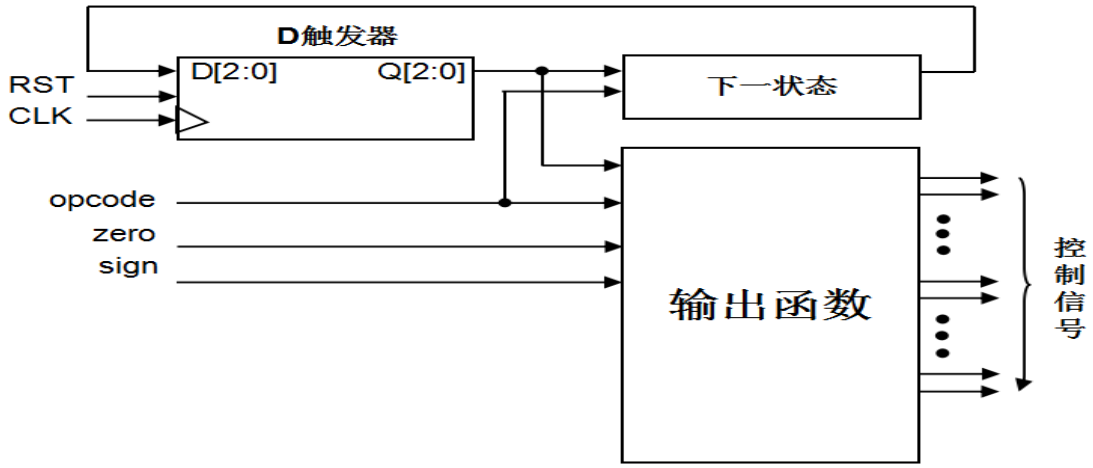


图 3 多周期 CPU 控制部件的原理结构图

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。

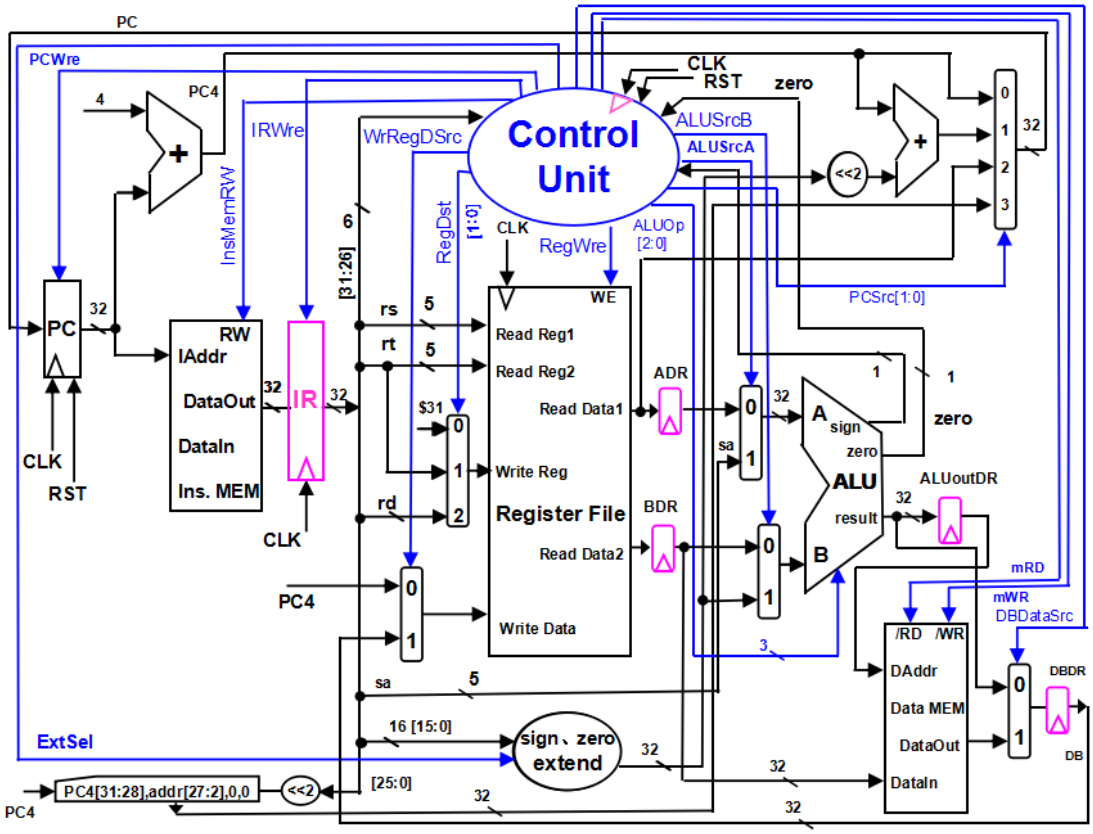


图 4　多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在多周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，给出寄存器地址（编号），读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，

在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。图中控制信号功能如表 1 所示，表 2 是 ALU 运算功能表。

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

表 1 控制信号作用

控制信号名	状态“0”	状态“1”
RST	对于 PC，初始化 PC 为程序首地址	对于 PC，PC 接收下一条指令地址
PCWre	PC 不更改，相关指令：halt，另外，除‘000’状态之外，其余状态慎改 PC 的值。	PC 更改，相关指令：除指令 halt 外，另外，在‘000’状态时，修改 PC 的值合适。
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addiu、and、andi、ori、xori、slt、slti、sw、lw、beq、bne、bltz	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、and、slt、sll、beq、bne、bltz	来自 sign 或 zero 扩展的立即数，相关指令：addiu、andi、ori、xori、slti、lw、sw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、sub、addiu、and、andi、ori、xori、sll、slt、slti	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、j、sw、jr、halt	寄存器组寄存器写使能，相关指令：add、sub、addiu、and、andi、ori、xori、sll、slt、slti、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、addiu、sub、and、andi、ori、xori、sll、slt、slti、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	存储器输出高阻态	**读数据存储器，相关指令：lw**
mWR	无操作	**写数据存储器，相关指令：sw**
IRWre	IR(指令寄存器)不更改	IR 寄存器写使能。向指令存储器发出读指令代码后，这个信号也接着发出，在时钟上升沿，IR 接收从指令存储器送来的指令代码。与每条指令都相关。
ExtSel	(zero-extend)**immediate**，相关指令：andi、xori、ori;	(sign-extend)**immediate**，相关指令：addiu、slti、lw、sw、beq、bne、bltz;
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addiu、sub、and、andi、ori、xori、slt、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0); 01：pc<－pc+4+(sign-extend)**immediate ×4**，相关指令：beq(zero=1)、	

	bne(zero=0)、bltz(sign=1)； 10：pc<－rs，相关指令：jr； 11：pc<－{pc[31:28],addr[27:2],2'b00}，相关指令：j、jal；
RegDst[1..0]	写寄存器组寄存器的地址，来自： 00：0x1F($31)，相关指令：jal，用于保存返回地址（$31<-pc+4）； 01：rt 字段，相关指令：addiu、andi、ori、xori、slti、lw； 10：rd 字段，相关指令：add、sub、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 = B<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与

101	Y= (A<B) ?1: 0	比较 A<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 ⊕ B	异或

四．实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五．实验过程与结果

多周期CPU主要需要实现CPU设计部分，汇编语言编译器部分，仿真部分和写板部分。

1、CPU设计部分。

（1）时钟边沿分配

多周期CPU与单周期CPU不同，单周期CPU一条指令在一个周期完成，而多周期CPU一条指令在多个周期内完成。因此，PC，IR，D触发器，状态寄存器，寄存器组寄存器必须要使用时序电路，而其他部分则可以使用组合电路。

多周期CPU可以分为5个阶段：IF（指令获取），ID（指令译码），EXE（计算），MEM（内存读写），WB（寄存器写）。每个阶段的触发时钟分别为PC（地址跳转），IR（指令寄存器），ADR/BDR（ALU输入寄存器），ALUoutDR（ALU输出寄存器），DBDR（数据总线寄存器）。

因此，以上的五个阶段的出发点，以及状态获取部分（Get_state）都采用上升沿触发，而寄存器写则采用下降沿触发。每个状态在一个时钟周期执行

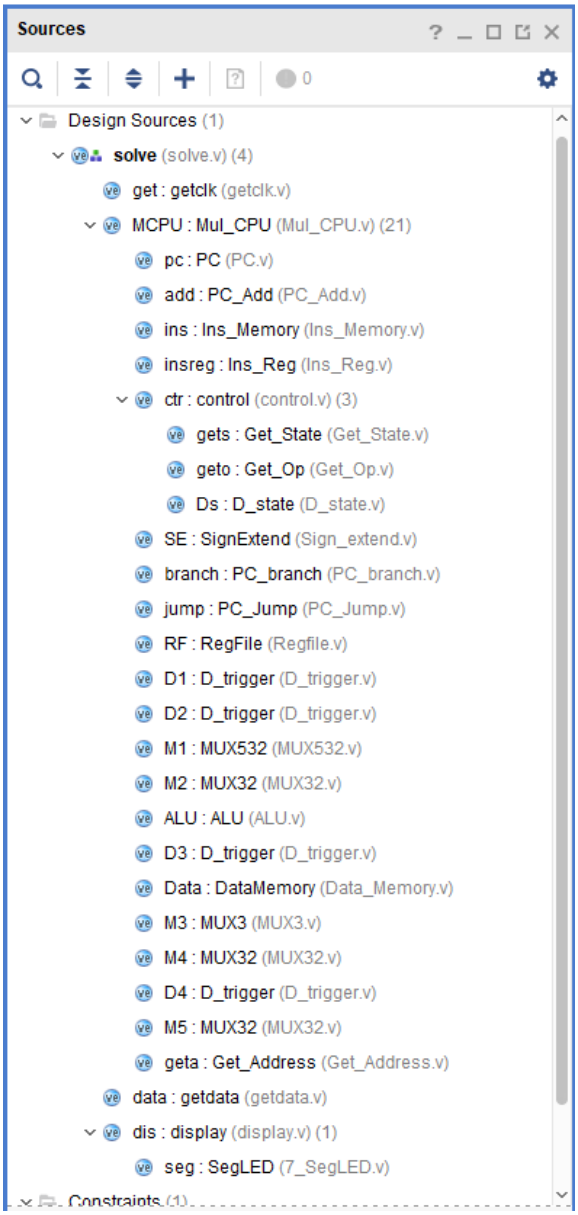
（2）各部分的实现

CPU的设计需要实现地址跳转（PC），指令内存（Instruction Memory）,控制信号译码（control），符号拓展（Sign Extend），寄存器（Regfile），计算器（ALU），选择器（MUX），数组内存（Data Memory），地址获取（Get Address），状态（stste）获取与跳转，D触发器（D flip-flop），指令寄存器（Instruction Register）。

声明：出于代码书写方便与可读性，我将状态和指令类型的二进制编码存在了若干参数里，在下面的代码中直接调用相关的二进制编码。以下为参数与二进制编码的对应。

```
parameter [2:0]
  IF=3'b000,ID=3'b001,aEXE=3'b010,bEXE=3'b101,
  cEXE=3'b110,MEM=3'b011,aWB=3'b111,cWB=3'b100;
parameter [5:0]
  add=6'b000000,sub=6'b000001,addiu=6'b000010,And=6'b010000,
  andi=6'b010001,ori=6'b010010,xori=6'b010011,sll=6'b011000,
  slti=6'b100110,slt=6'b100111,sw=6'b110000,lw=6'b110001,
  beq=6'b110100,bne=6'b110101,bltz=6'b110110,
  j=6'b111000,jr=6'b111001,jal=6'b111010,halt=6'b111111;
```

以下为CPU设计所写的所有顶层与底层模块



1）地址跳转（PC）与地址获取（Get Address）

地址跳转部分，在上升沿触发，输入当前的地址，输出下一步要跳转的地址。根据Reset信号决定是否要清空地址，根据PCWre信号决定是否跳转。

输入：clk,Reset,PCWre,address

输出：NowAddress

核心代码：

```
always @(posedge clk or negedge Reset)
begin
    if (Reset == 0) NowAddress<=0;
    else begin
       if (PCWre) NowAddress<=address;
       else NowAddress<=NowAddress;
    end
end
```

地址获取部分，本质为一个四选一选择器。输入当前地址和四种可能的跳转位置，根据PCSrc信号决定下一个地址的位置。

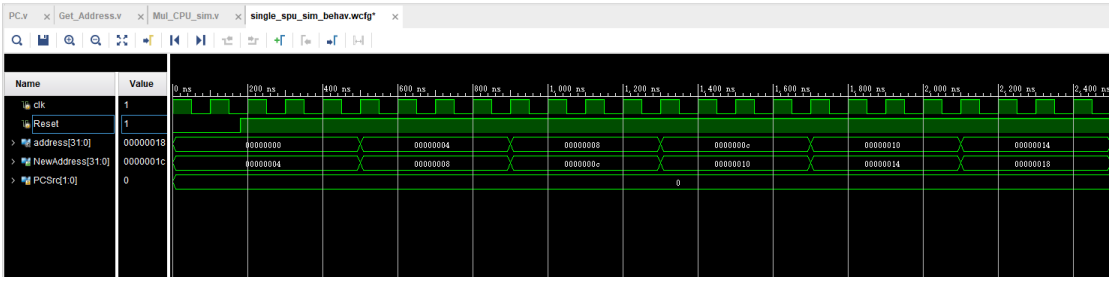
输入：PCSrc,A,B,C,D

输出：NewAddress

核心代码：

```
always @(*)begin
    case (PCSrc)
       2'b00: NewAddress=A;
       2'b01: NewAddress=B;
       2'b10: NewAddress=C;
       2'b11: NewAddress=D;
    endcase
end
```

仿真结果：



2)状态（state）获取与跳转

状态获取部分，根据状态转移通路图，根据当前状态获和指令取下一个状态。

状态跳转部分，是一个上升沿触发的D触发器，下面会介绍D触发器的实现。

输入：reset，state，OP

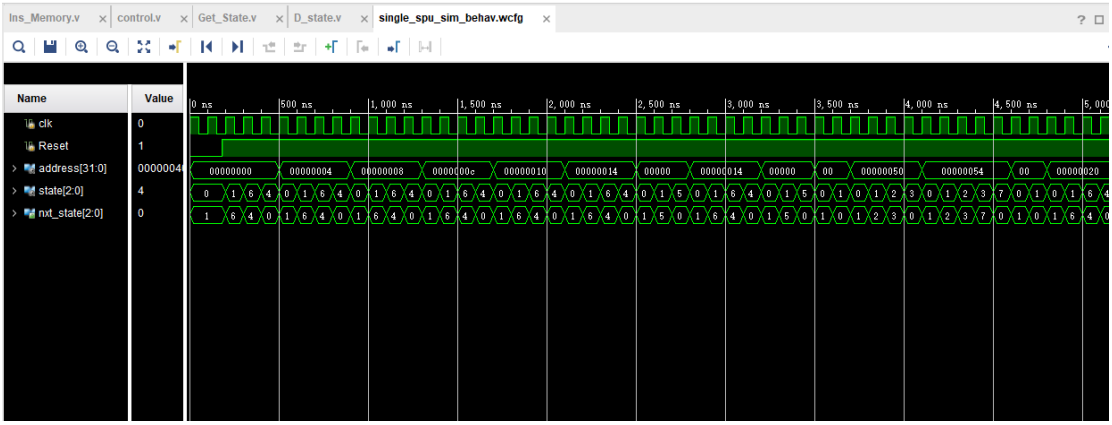
输出：nxt_state

核心代码：

```
always@( state or OP )begin
    begin
        case (state)
            IF: nxt_state<=ID;
            ID: begin
                    if( OP==j || OP==jal || OP==jr || OP==halt )nxt_state<=IF;
                    else if( OP==sw || OP==lw )nxt_state<=aEXE;
                    else if( OP==beq || OP==bne || OP==bltz )nxt_state<=bEXE;
                    else nxt_state<=cEXE;
                end
            aEXE: nxt_state=MEM;
            bEXE: nxt_state=IF;
            cEXE: nxt_state=cWB;
            MEM: begin
                    if( OP==lw )nxt_state=aWB;
                    else nxt_state=IF;
                 end
            aWB: nxt_state=IF;
```

```
            cWB: nxt_state=IF;
        endcase
    end
end
```

仿真结果：



第4行为当前状态，第5行位下一个状态

3)指令内存（Instruction Memory）

指令从文件“input.txt”读入，开一个数组模拟内存存储所有指令。每次调用的时候根据地址从指令内存数组中获取当前的指令。

输入：InsMenRW,address

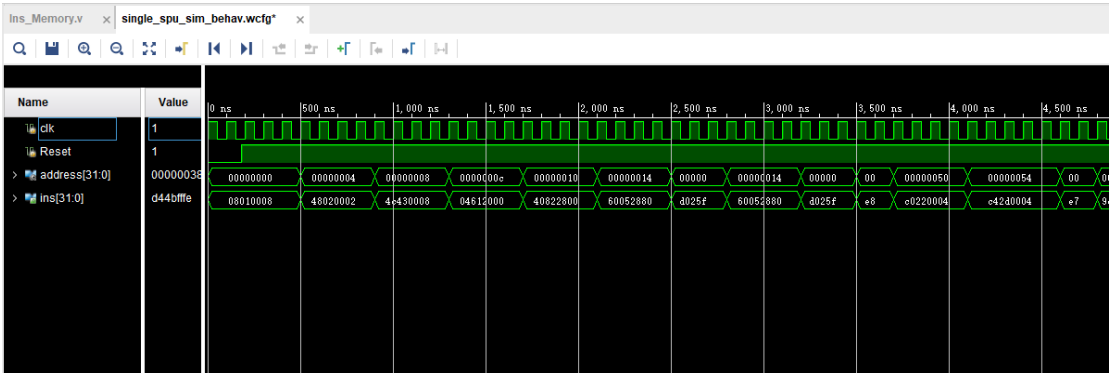
输出：ins

核心代码：

```
reg [7:0] Memory [255:0];
initial begin
    $readmemb("D:/vivado/MUL_CPU/input.txt", Memory);
end

always@(address or InsMenRW)begin
    if(InsMenRW)begin
        ins={ Memory[address] , Memory[address+1] , Memory[address+2] ,
Memory[address+3] };
```

```
    end
end
```

仿真结果：



4）指令寄存器（Instruction Register）

指令寄存器是ID阶段的触发点，在上升沿传输执行。为了代码编写方便，在指令寄存器的输出的时候顺便把指令分割成OP,rs,rt,rd,sa,immediate。

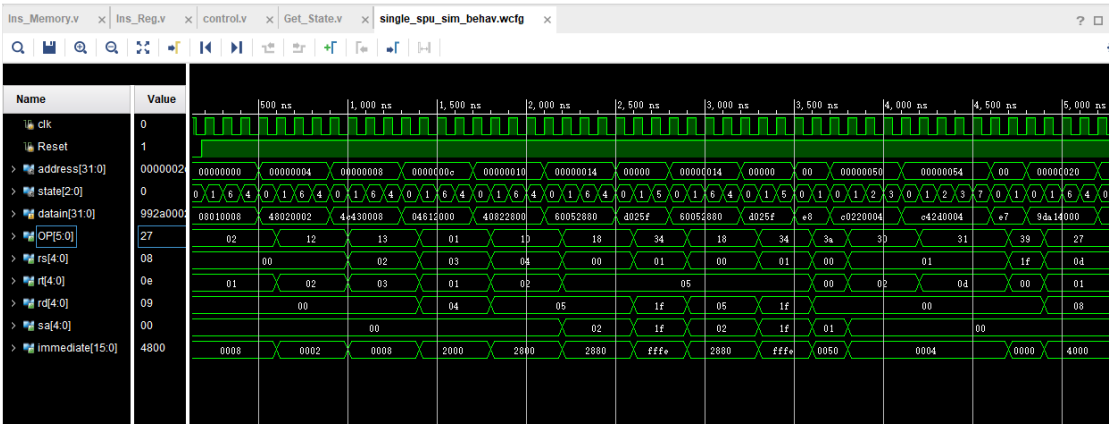
输入：clk,datain

输出：OP,rs,rt,rd,sa,immediate

核心代码：

```
always @(posedge clk)begin
    OP= datain [31:26];
    rs= datain [25:21];
    rt= datain [20:16];
    rd= datain [15:11];
    sa= datain [10:6];
    immediate= datain [15:0];
end
```

仿真结果：



如图，在每条指令的第二个阶段（ID），指令的各个部分才会被译码出来

5）译码部分（control）

根据指令的类型与当前状态，确定各种控制信号的值，完成对控制信号的译码。

输入：OP,zero

输出:PCWre,IRWre,ALUSrcA,ALUSrcB,WrRegDSrc,DBDataSrc,RegWre, mRD, mWR,InsMenRW,RegDst,ExtSel,PCSrc,ALUOp

真值表:

状态	指令	zero	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	WrRegDSrc	InsMemRW	mRD	mWR	IRWre	Extsel	PCSrc	RegDst	ALUOp
IF	not halt	x	0	x	x	x	0	x	1	x	x	1	x	x	x	x
ID	j	x	1	x	x	x	0	x	x	x	0	0	x	11	x	x
	jal	x	1	x	x	x	1	0	x	x	0	0	x	11	0	x
	jr	x	1	x	x	x	0	x	x	x	0	0	x	10	x	x
	halt	x	1	x	x	x	0	x	x	x	0	0	x	x	x	x
aEXE	lw	x	0	0	1	x	0	x	x	x	0	0	1	x	x	0
	sw	x	0	0	1	x	0	x	x	x	0	0	1	x	x	0
MEM	lw	x	0	x	x	x	0	x	x	1	0	0	x	x	x	x
	sw	x	1	x	x	x	0	x	x	x	1	0	x	0	x	x
aWB	lw	x	1	x	x	1	1	1	x	x	0	0	x	0	1	x
bEXE	beq	0	1	0	0	x	0	x	x	x	0	0	1	0	x	1
		1	1	0	0	x	0	x	x	x	0	0	1	1	x	1
	bne	0	1	0	0	x	0	x	x	x	0	0	1	1	x	1
		1	1	0	0	x	0	x	x	x	0	0	1	0	x	1
	bltz	0	1	0	0	x	0	x	x	x	0	0	1	1	x	110
		1	1	0	0	x	0	x	x	x	0	0	1	0	x	110
cEXE	add	x	0	0	0	x	0	x	x	x	0	0	x	x	x	0
	sub	x	0	0	0	x	0	x	x	x	0	0	x	x	x	1
	and	x	0	0	0	x	0	x	x	x	0	0	x	x	x	100
	addiu	x	0	0	1	x	0	x	x	x	0	0	1	x	x	0
	andi	x	0	0	1	x	0	x	x	x	0	0	0	x	x	100
	ori	x	0	0	1	x	0	x	x	x	0	0	0	x	x	11
	xori	x	0	0	1	x	0	x	x	x	0	0	0	x	x	111
	sll	x	0	1	0	x	0	x	x	x	0	0	x	x	x	10
	slt	x	0	0	0	x	0	x	x	x	0	0	x	x	x	110
	slti	x	0	0	1	x	0	x	x	x	0	0	1	x	x	110
cWB	add	x	1	x	x	0	1	1	x	x	0	0	x	0	10	x
	sub	x	1	x	x	0	1	1	x	x	0	0	x	0	10	x
	and	x	1	x	x	0	1	1	x	x	0	0	x	0	10	x
	addiu	x	1	x	x	0	1	1	x	x	0	0	x	0	1	x
	andi	x	1	x	x	0	1	1	x	x	0	0	x	0	1	x
	ori	x	1	x	x	0	1	1	x	x	0	0	x	0	1	x
	xori	x	1	x	x	0	1	1	x	x	0	0	x	0	1	x
	sll	x	1	x	x	0	1	1	x	x	0	0	x	0	10	x
	slt	x	1	x	x	0	1	1	x	x	0	0	x	0	10	x
	slti	x	1	x	x	0	1	1	x	x	0	0	x	0	1	x

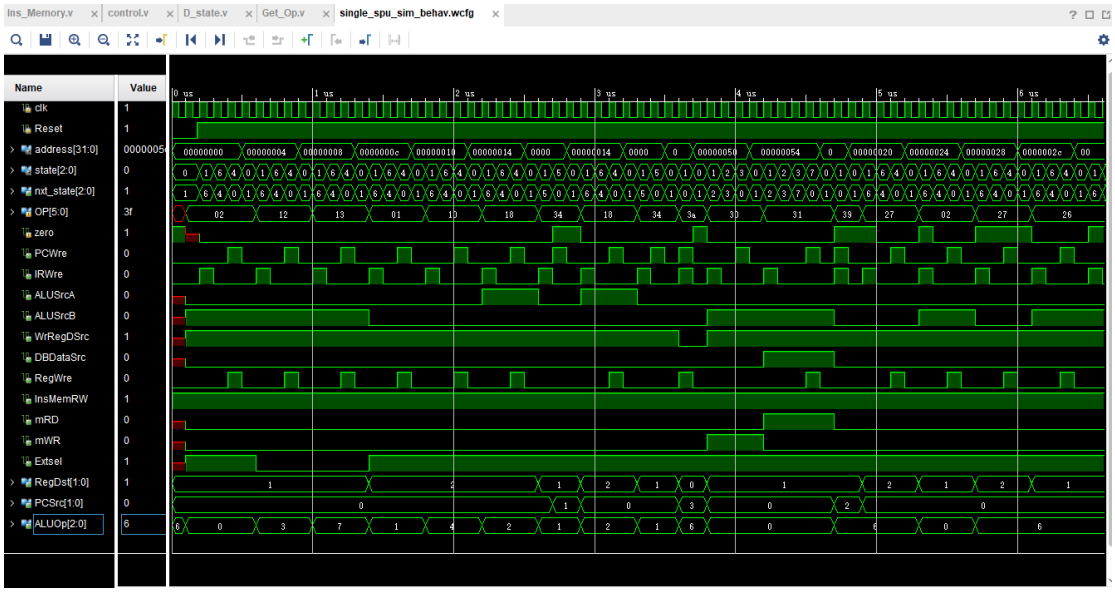
核心代码:

```
always@( state or nxt_state or OP or zero )begin
```

```
    PCWre=( nxt_state==IF && OP!=halt )?1:0;//
    ALUSrcA=( OP==sll )?1:0;
    ALUSrcB=( OP==lw || OP==sw || OP==addiu || OP==andi || OP==ori || OP==xori
|| OP==slti )?1:0;
    DBDataSrc=( OP==lw )?1:0;
    //RegWre=( OP==bne || OP==beq || OP==bltz || OP==halt || OP==sw || OP==j
|| OP==jr )?0:1;
    RegWre=( state==aWB || state==cWB || ( OP==jal && state==ID ) )?1:0;
    WrRegDSrc=( OP==jal )?0:1;
    InsMemRW=1;
    mRD=( OP==lw )?1:0;
    mWR=( OP==sw )?1:0;
    IRWre=( state==ID )?1:0;
    Extsel=( OP==andi || OP==ori || OP==xori )?0:1;
    if( OP==j || OP==jal )PCSrc=2'b11;
    else if( OP==jr )PCSrc=2'b10;
    else if( (OP==beq && zero==1) || (OP==bne && zero==0) || (OP==bltz &&
zero==0) )PCSrc=2'b01;
    else PCSrc=2'b00;
    //PCSrc
    if( OP==add || OP==sub || OP==And || OP==sll || OP==slt )RegDst=2'b10;
    else if( OP==jal )RegDst=2'b00;
    else RegDst=2'b01;
    //RegDst
    if( OP==lw || OP==sw || OP==add || OP==addiu )ALUOp=3'b000;
    else if( OP==beq || OP==bne || OP==sub )ALUOp=3'b001;
    else if( OP==sll )ALUOp=3'b010;
    else if( OP==ori )ALUOp=3'b011;
    else if( OP==And || OP==andi )ALUOp=3'b100;
```

```
    else if( OP==xori )ALUOp=3'b111;
    else ALUOp=3'b110;
    //ALUOp
end
```

仿真结果：



6)符号拓展（Sign Extend）

根据控制信号ExtSel决定拓展的类型，将16位立即数拓展为32位立即数

根据控制信号ExtSel决定拓展的类型，将16位立即数拓展为32位立即数

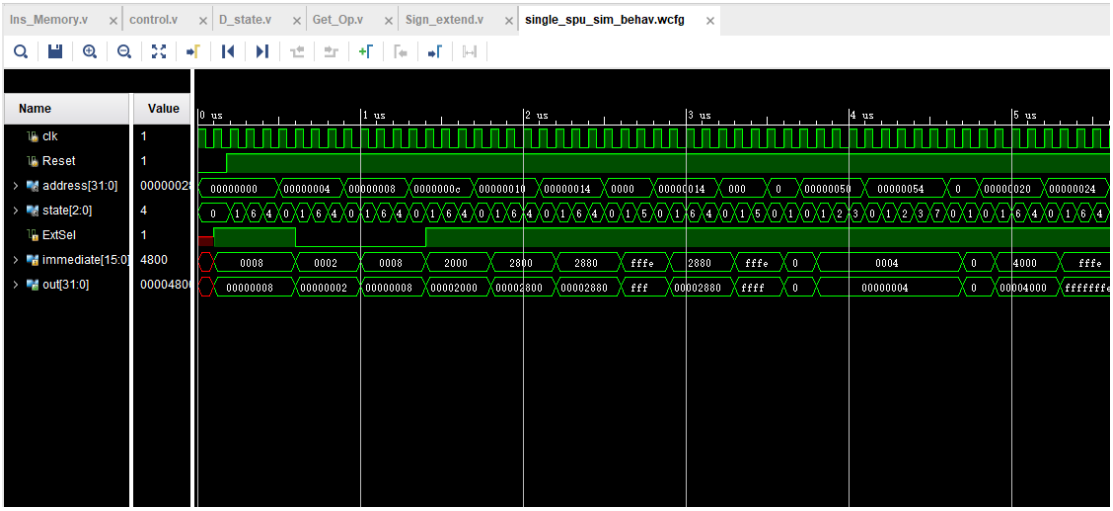
输入：immediate，ExtSel

输出：out

核心代码：

```
assign out[15:0] = immediate;
assign out[31:16] = ExtSel? (immediate[15]? 16'hffff : 16'h0000) : 16'h0000;
```

仿真结果：



7)寄存器（Regfile）

寄存器分为读写数据两个部分。

读操作从寄存器中读取对应位置的数据，两个地址为ReadReg1和ReadReg2。

写操作只有在时钟下降沿才会进行，在对应的内存写入数据，写入地址为WriteReg，写入的数据为WriteData。

Regfile数组模拟寄存器，数据存储在该数组里。

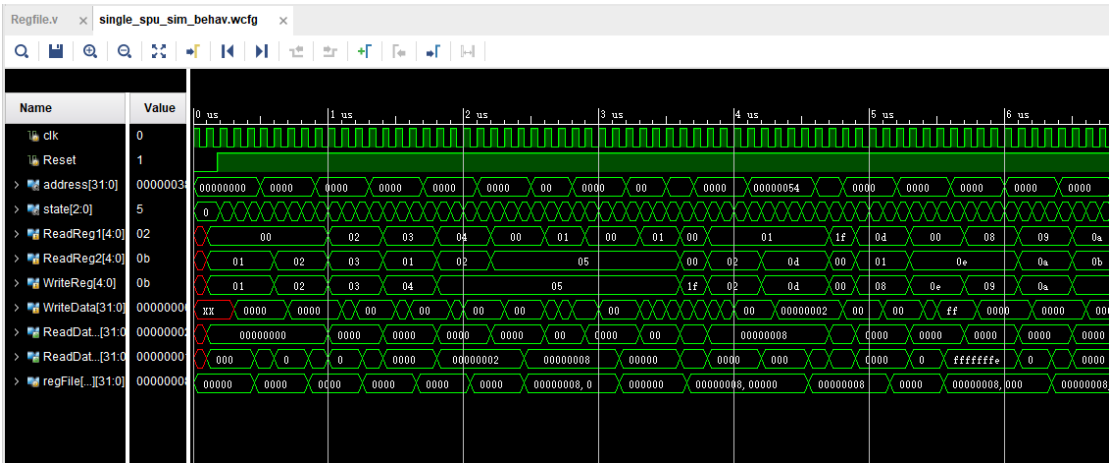
输入：CLK,RST,RegWre,ReadReg1,ReadReg2,WriteReg, WriteData,

输出：ReadData1,ReadData2

核心代码：

```
reg [31:0] regFile[1:31];
integer i;
assign ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1];
assign ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];
always @ (negedge CLK or negedge RST) begin
    if (RST==0) begin
        for(i=1;i<32;i=i+1) regFile[i] <= 0;
    end
    else if(RegWre == 1 && WriteReg != 0)
        regFile[WriteReg] <= WriteData;
end
```

仿真结果：



可以看到，寄存器写都是在下降沿进行。

8）计算部分（ALU）

根据输入的ALUopcode信号决定计算的类型，并将rega和regb两个数据进行计算，结果输出到result里。Zero位用于判断结果是否为0，用于条件跳转指令。

输入：ALUopcode,rega,regb

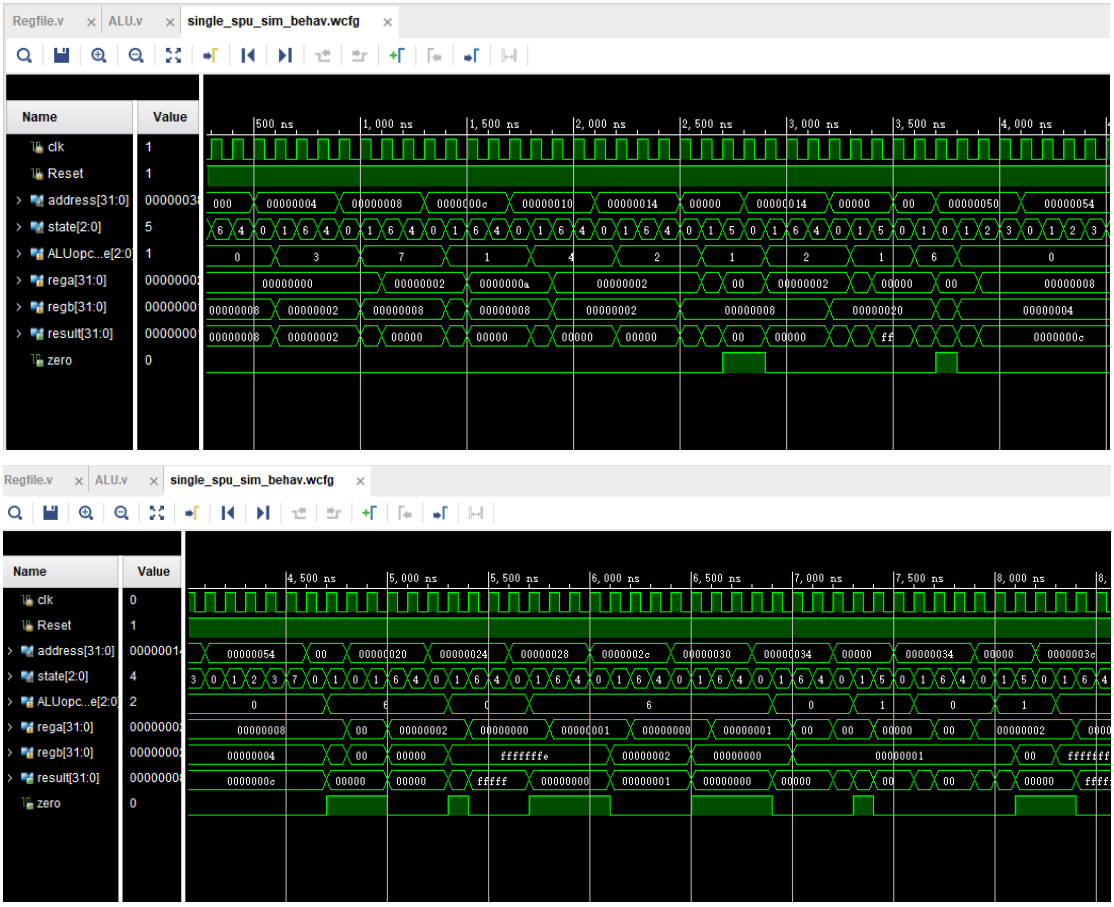
输出：result,zero

核心代码：

```
assign zero = (result==0)?1:0;

    always @( ALUopcode or rega or regb ) begin

        case (ALUopcode)

            3'b000 : result = rega + regb;

            3'b001 : result = rega - regb;

            3'b010 : result = regb << rega;

            3'b011 : result = rega | regb;

            3'b100 : result = rega & regb;

            3'b101 : result = (rega < regb)?1:0; // 不带符号比较

            3'b110 : begin // 带符号比较

                        if(rega < regb && (rega[31] == regb[31]))result = 1;

                        else if (rega[31] == 1 && regb[31] == 0) result = 1;
```

```
                    else result = 0;
                end
        3'b111 : result = rega ^ regb;
        default : begin
                result = 32'h00000000;
                $display (" no match");
        end
    endcase
end
```

仿真结果



如图，在EXE周期才会确定稳定的ALU的输入数据，并得到结果。

9）数据内存（Data Memory）

数据内存分为数据读写两个部分。读数据部分，将内存中的数据写到Dataout数组中。写数据只在下降沿写，将writeData的数据写入内存。Ram数组模拟数据内存。

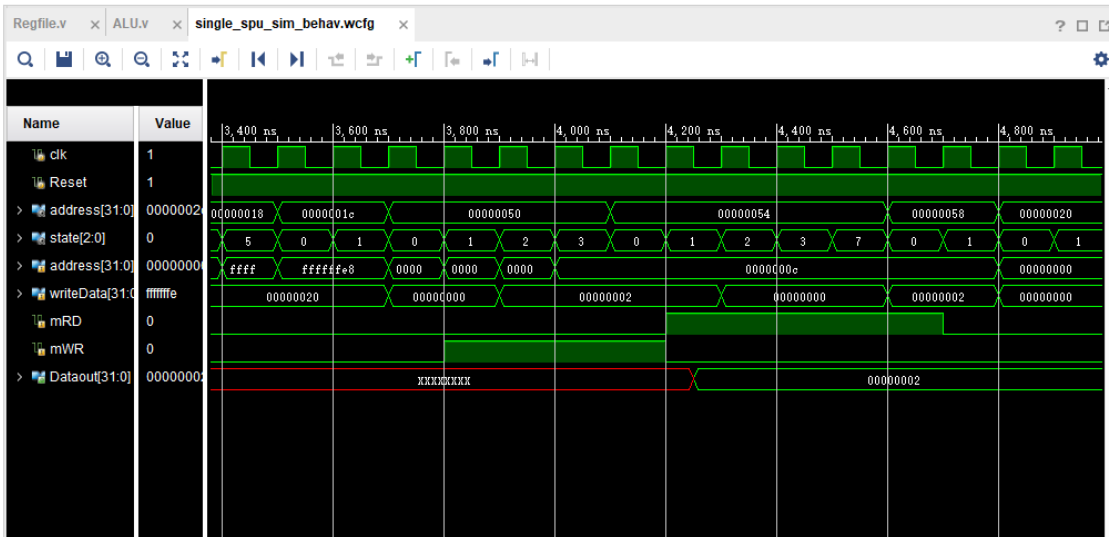
输入：clk,address,writeData,mRD,mWR,

输出：Dataout

核心代码：

```
reg [7:0] ram [0:60];
//读
always @(address or mRD) begin
    Dataout[7:0] = (mRD==1)?ram[address + 3]:8'bz;
    Dataout[15:8] = (mRD==1)?ram[address + 2]:8'bz;
    Dataout[23:16] = (mRD==1)?ram[address + 1]:8'bz;
    Dataout[31:24] = (mRD==1)?ram[address ]:8'bz;
end
// 写
always@( negedge clk ) begin
    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
```

仿真结果：



10）选择器（MUX）

根据需求需要建立三种选择器，分别为两个五进制（用于选择寄存器写入地址），两个32进制（用于选择ALU中第二个数据和寄存器写入数据），一个5进制一个32进制（用于选择ALU中第一个数据）。

输入两个数据(data0,data1)和选择控制信号(op)，输出其中一个数据。

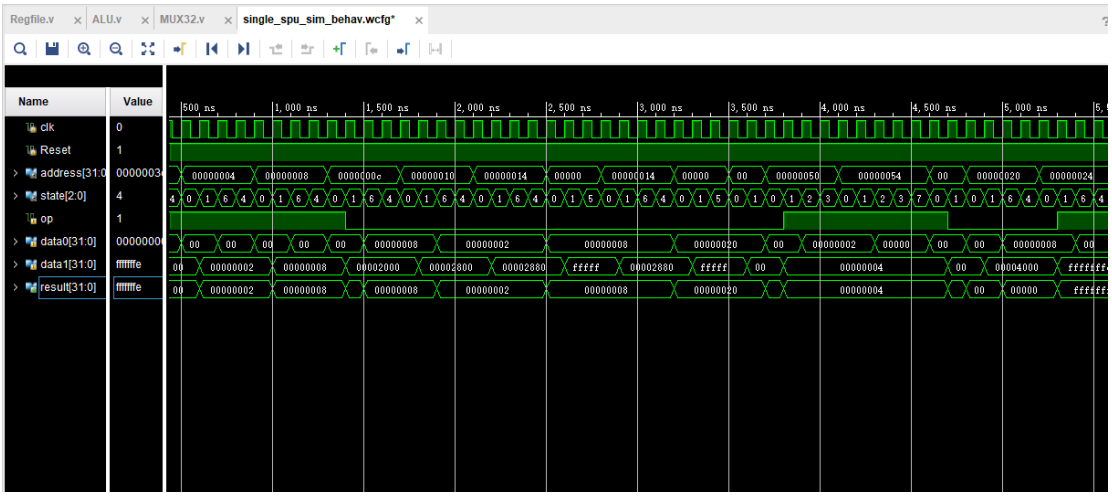
输入：data0,data1,op

输出：result

核心代码：

```
always@(op or data0 or data1)begin
    result=(op==0)?data0:data1;
end
```

仿真结果：



11）D触发器（D flip-flop）

D触发器有两种类型，上升沿触发与下降沿触发。用type决定D触发器的类型。

输入：clk,datain,type

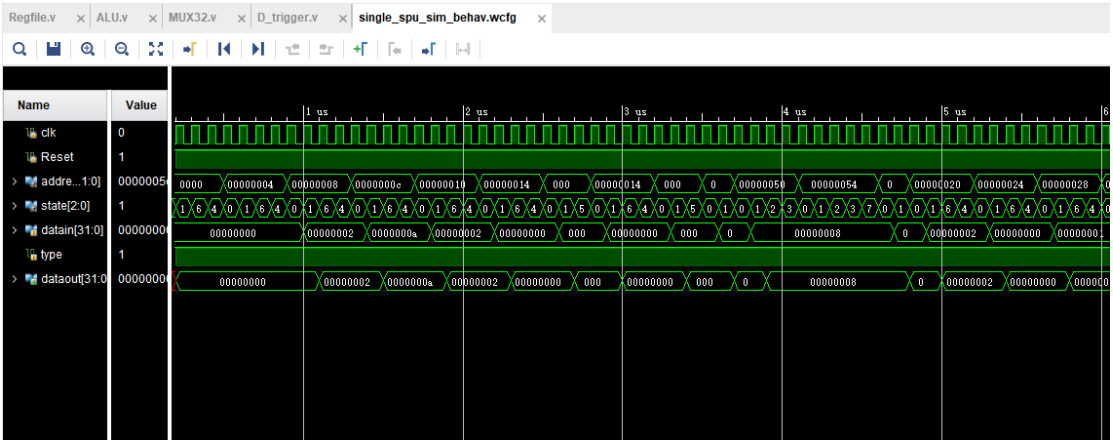
输出：dataout

核心代码：

```
always @(posedge clk)begin
    if(type)dataout<=datain;
```

```
end

always @(negedge clk)begin

    if(!type)dataout<=datain;

end
```

仿真结果：



如图，数据输出与数据输入相差一个周期。

12）顶层模块

顶层文件将上述底层文件串联在一起，构成多周期CPU。

因为verilog支持并行，所以调用底层文件的顺序并没有什么影响。

将各种控制信号和数据存储在顶层文件中。

输入：clk,reset

输出：address

核心代码：

```
PC pc(clk,Reset,PCWre,NewAddress,address);

//get_PC

PC_Add add(address,PC4);

//PC+4

Ins_Memory ins(InsMenRW,address,inst);

//取指

Ins_Reg insreg(clk,inst,OP,rs,rt,rd,sa,immediate);

//指令寄存器

control
```

```
ctr(clk,Reset,OP,zero,PCWre,IRWre,ALUSrcA,ALUSrcB,WrRegDSrc,DBDataSr
c,RegWre,InsMenRW,mRD,mWR,ExtSel,RegDst,PCSrc,ALUOp);
//译码
SignExtend SE(immediate,ExtSel,Extend);
//符号拓展
PC_branch branch(PC4,Extend,PCbranch);
PC_Jump jump(PC4,Extend,PCjump);
//PC_jump
RegFile
RF(clk,Reset,RegWre,rs,rt,RegWrite,writeData,Regout1,Regout2);
//读写寄存器
D_trigger D1(clk,Regout1,1'b1,data3);
D_trigger D2(clk,Regout2,1'b1,data4);
//ALU输入触发器
MUX532 M1(data3,sa,ALUSrcA,data1);
MUX32 M2(data4,Extend,ALUSrcB,data2);
//ALU输入选择器
ALU ALU(ALUOp,data1,data2,result,zero);
//计算
D_trigger D3(clk,result,1'b1,DAddr);
//内存地址触发器
DataMemory Data(clk,Reset,DAddr,data4,mRD,mWR,Dataout);
//数据内存读写
MUX3 M3(RegDst,endReg,rt,rd,RegWrite);
//选择数据寄存器的写入地址
MUX32 M4(result,Dataout,DBDataSrc,wData1);
D_trigger D4(clk,wData1,1'b1,wData);
//寄存器写入触发器
MUX32 M5(PC4,wData,WrRegDSrc,writeData);
```

```
//选择写入寄存器数字
Get_Address geta(PCSrc,PC4,PCbranch,Regout1,PCjump,NewAddress);
//算新地址
```

2、仿真部分

（1）仿真模块设计

设立虚拟时钟clk初值为1，Reset初值为1，每50ps刷新一次，即clk=!clk。

核心代码：

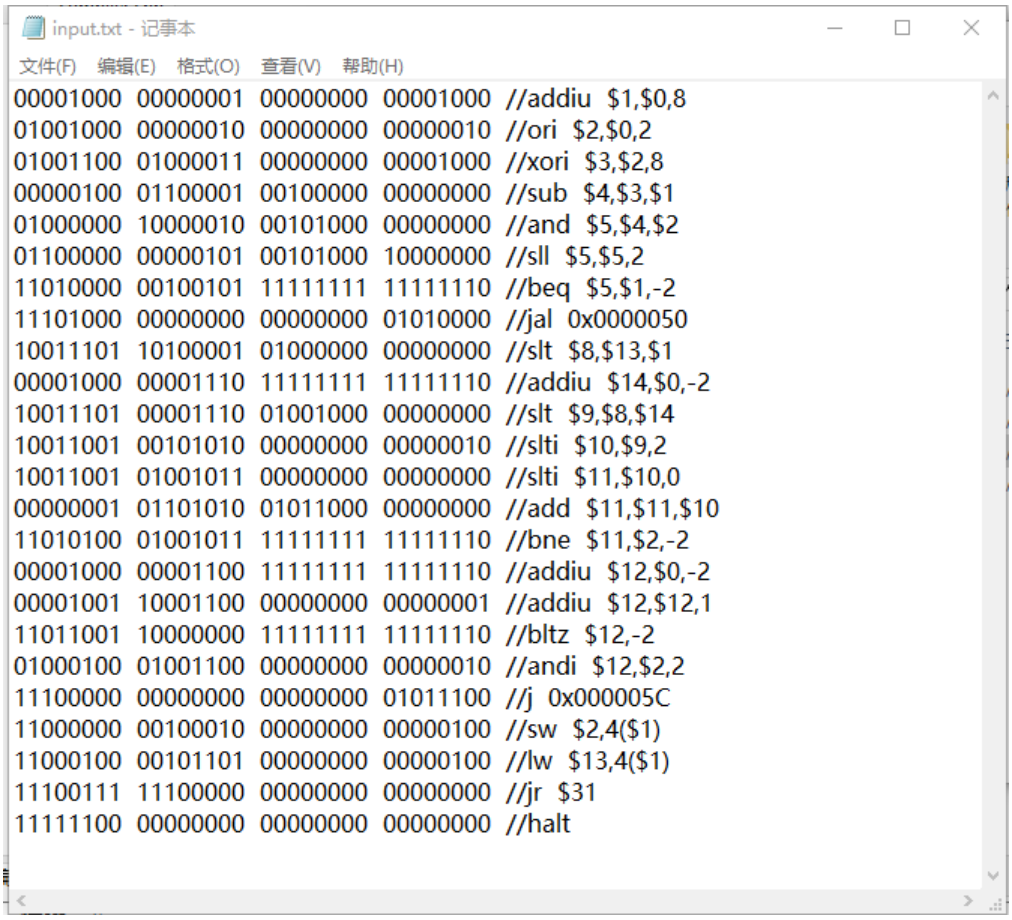
```
module mul_spu_sim();
reg clk,Reset;
wire [31:0]address;
Mul_CPU cpu(
    .clk(clk),
    .Reset(Reset),
    .address(address)
);
always #50 clk=!clk;
initial begin
    Reset=0;clk=0;clk=1;
    #180;Reset=1;
end
endmodule
```

（2）仿真结果与分析

1）指令集

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	addiu $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	=	44020002
0x00000008	xori $3,$2,8	010011	00010	00011	0000 0000 0000 1000	=	4C430008
0x0000000C	sub $4,$3,$1	000001	00011	00001	00100 000 0000 0000	=	04612000
0x00000010	and $5,$4,$2	010000	00100	00010	00101 000 0000 0000	=	40822800
0x00000014	sll $5,$5,2	011000	00000	00101	00101 00010 00 0000	=	60052880
0x00000018	beq $5,$1,-2(=,转 14)	110100	00001	00101	1111 1111 1111 1110	=	D025FFFE
0x0000001C	jal 0x0000050	111010	00000	00000	0000 0000 0101 0000	=	E8000050
0x00000020	slt $8,$13,$1	100111	01101	00001	01000 000 0000 0000	=	9DA14000
0x00000024	addiu $14,$0,-2	000010	00000	01110	1111 1111 1111 1110	=	080EFFFE
0x00000028	slt $9,$8,$14	100111	01000	01110	01001 000 0000 0000	=	9D0E4800
0x0000002C	slti $10,$9,2	100110	01001	01010	0000 0000 0000 0010	=	992A0002
0x00000030	slti $11,$10,0	100110	01010	01011	0000 0000 0000 0000	=	994B0000
0x00000034	add $11,$11,$10	000000	01011	01010	01011 000 0000 0000	=	016A5800
0x00000038	bne $11,$2,-2 (≠,转 34)	110101	00010	01011	1111 1111 1111 1110	=	D44BFFFE
0x0000003C	addiu $12,$0,-2	000010	00000	01100	1111 1111 1111 1110	=	080CFFFE
0x00000040	addiu $12,$12,1	000010	01100	01100	0000 0000 0000 0001	=	098C0001
0x00000044	bltz $12,-2 (<0,转 40)	110110	01100	00000	1111 1111 1111 1110	=	D980FFFE
0x00000048	andi $12,$2,2	010001	00010	01100	0000 0000 0000 0010	=	444C0002
0x0000004C	j 0x000005C	111000	00000	00000	0000 0000 0101 1100	=	E000005C
0x00000050	sw $2,4($1)	110000	00001	00010	0000 0000 0000 0100	=	C0220004
0x00000054	lw $13,4($1)	110001	00001	01101	0000 0000 0000 0100	=	C42D0004
0x00000058	jr $31	111001	11111	00000	0000 0000 0000 0000	=	E7E00000
0x0000005C	halt	111111	00000	00000	0000 0000 0000 0000	=	FC000000

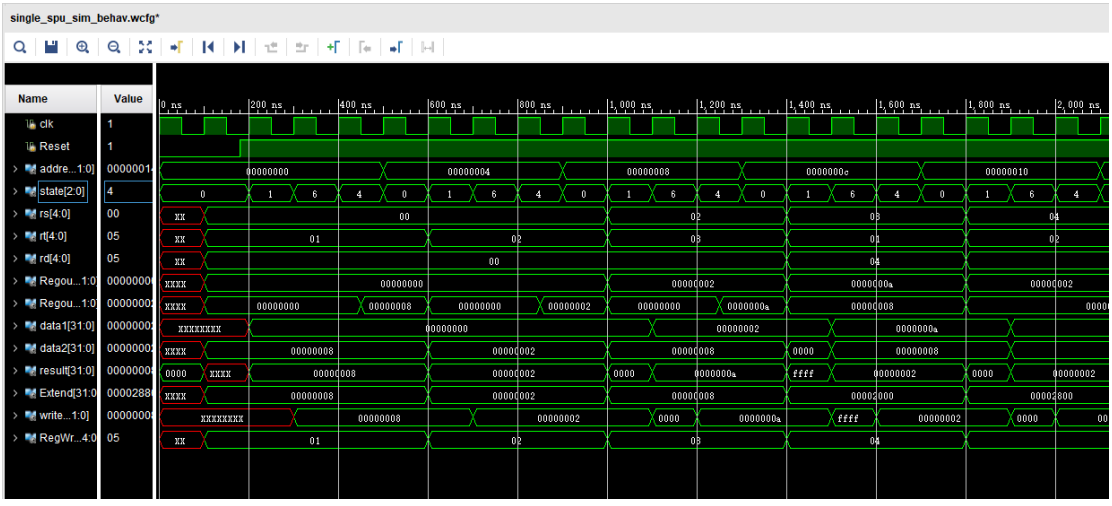
如图为多周期CPU的指令集



如图为多周期CPU的测试文件

2）仿真结果与分析

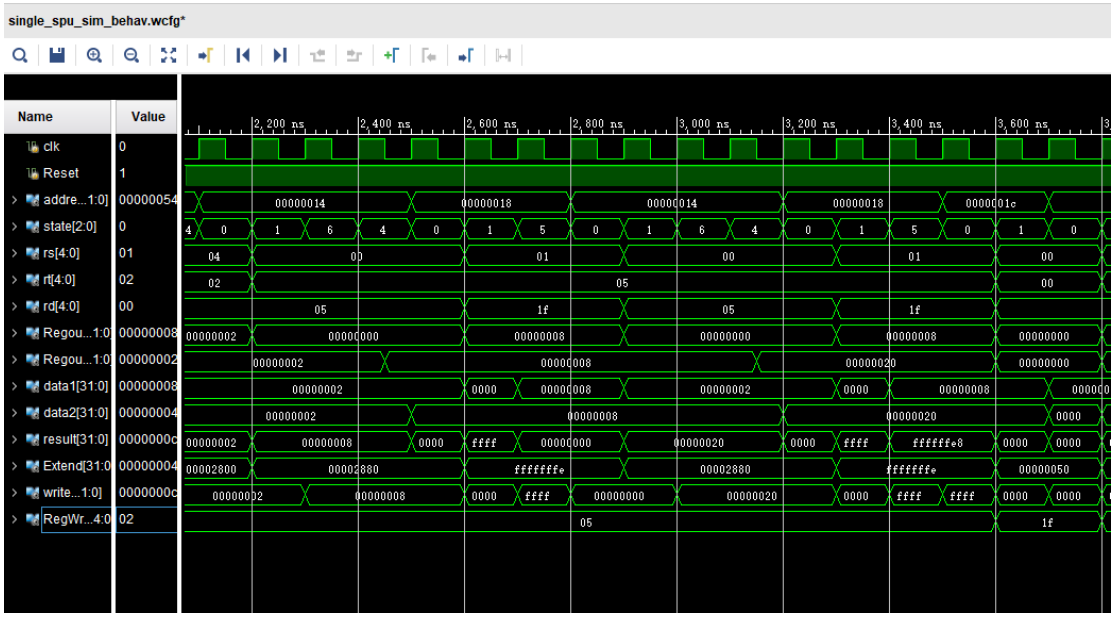
以下为仿真的结果波形图



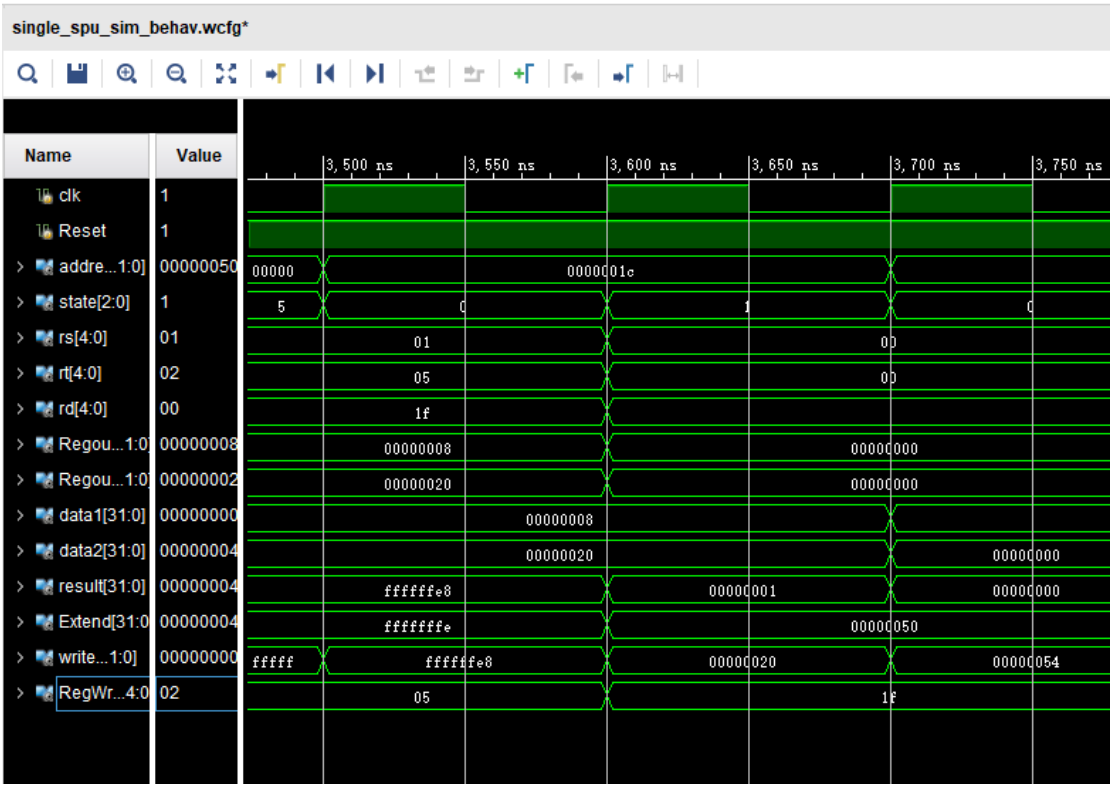
1~5条指令

```
addiu  $1,$0,8     $1=$0+8=0+8=8

ori  $2,$0,2       $2=$0+2=0+2=2
```

```
xori  $3,$2,8      $3=$2^8=2^8=10
sub  $4,$3,$1      $4=$3-$1=10-8=2
and  $5,$4,$2      $5=$4&$2=2&2=2
```

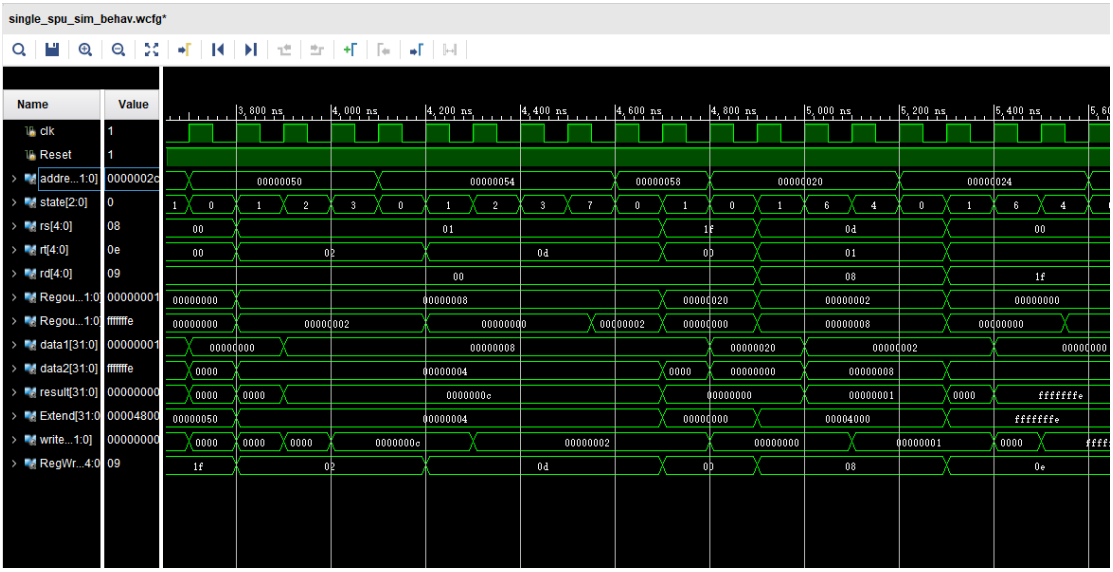



```
sll   $5,$5,2      $5=$5<<2=2<<2=8
beq  $5,$1,-2      $5=8,$1=8,8==8,跳转到0x14
sll   $5,$5,2      $5=$5<<2=8<<2=32
beq  $5,$1,-2      $5=32,$1=8,32!=8,不跳转
jal  0x0000050     跳转到0x50，并将PC+1=0x20存入$31
```

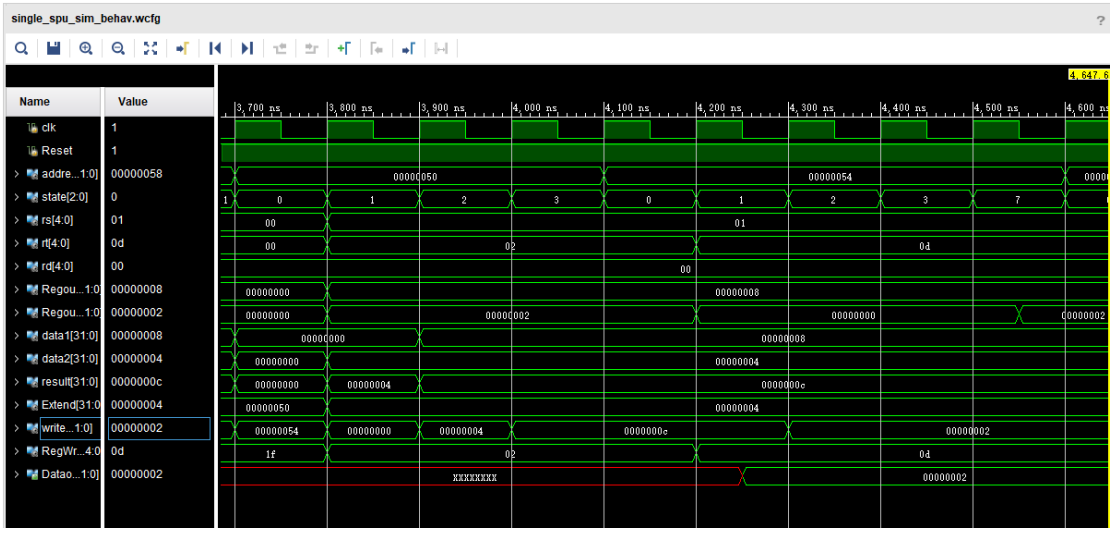


如图为jal指令的具体波形图，寄存器写的地址为0x1f，即$31。

寄存器写的数值为0x20,即PC+4。寄存器写的时间为3650ns。



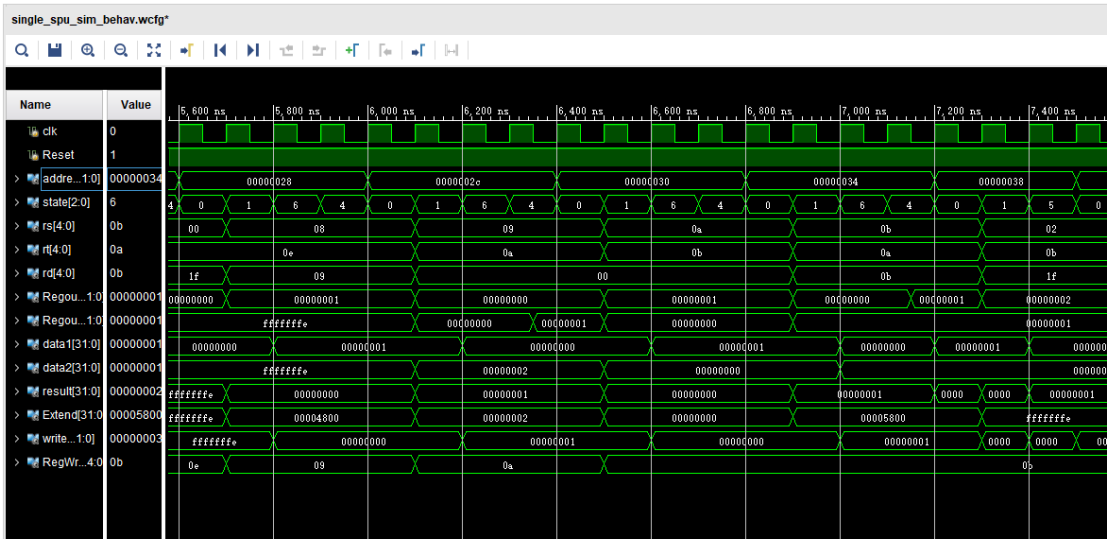

```
sw  $2,4($1)        $1+4=8+4=12,内存$12=$2=2

lw  $13,4($1)       $1+4=8+4=12,$13=内存$12=2

jr  $31             跳转到$31存的地址,即$20

slt  $8,$13,$1      $8=($13<$1)=(2<8)=1

addiu  $14,$0,-2    $14=$0-2=0-2=-2
```



如图为sw指令和lw指令的具体波形图。

rs=$1,$1=8,8+4=12,读写访问的内存为地址为$12,即0xc。

sw指令不读取内存，lw指令内存读取的数值为2。



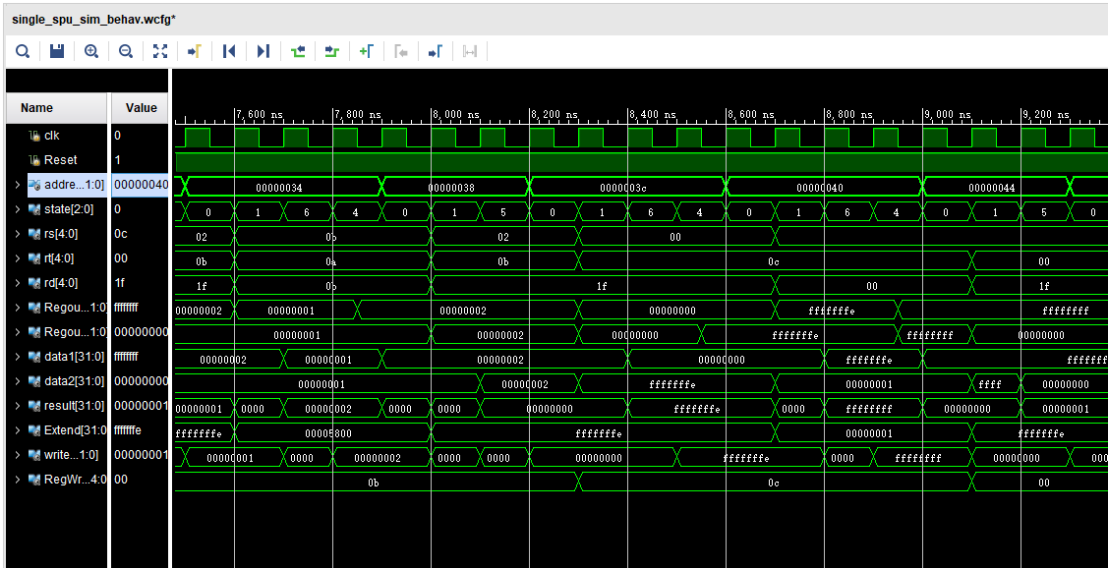
slt $9,$8,$14 $9=($8<$14)=(1<-2)=0

slti $10,$9,2 $10=($9<2)=(0<2)=1

slti $11,$10,0 $11=($10<0)=(1<0)=0

add $11,$11,$10 $11=$11+$10=0+1=1

bne $11,$2,-2 $11=1,$2=2,1!=2,跳转到0x34



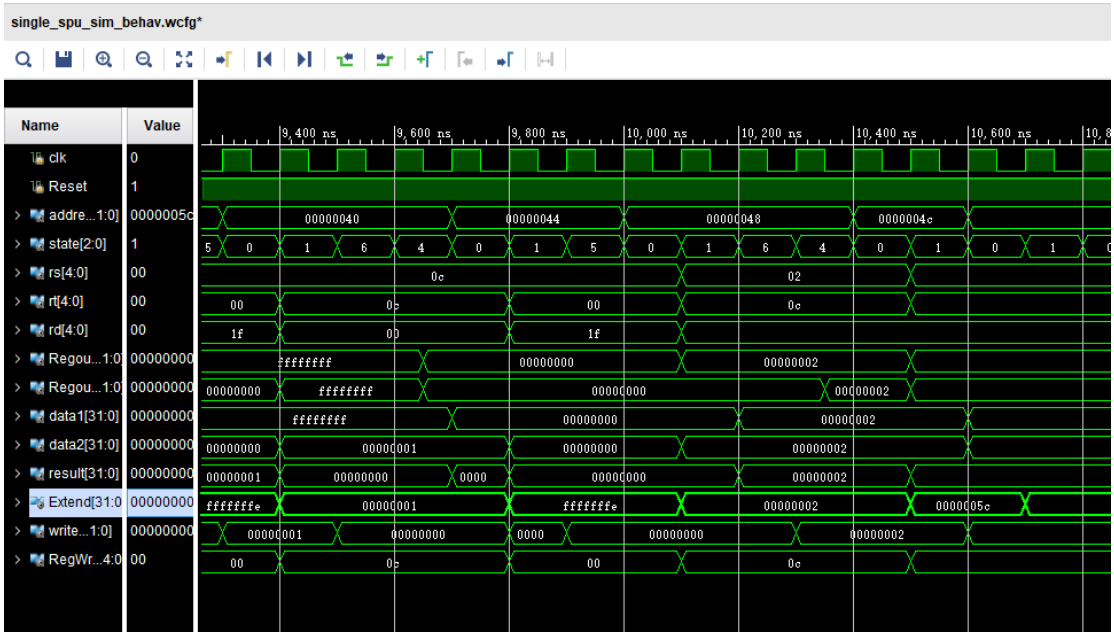
add $11,$11,$10 $11=$11+$10=1+1=2

bne $11,$2,-2 $11=2,$2=2,2==2,不跳转

addiu $12,$0,-2 $12=$0-2=0-2=-2

addiu $12,$12,1 $12=$12+1=-2+1=-1

bltz $12,-2 $12=-1,-1<-2,跳转到0x40

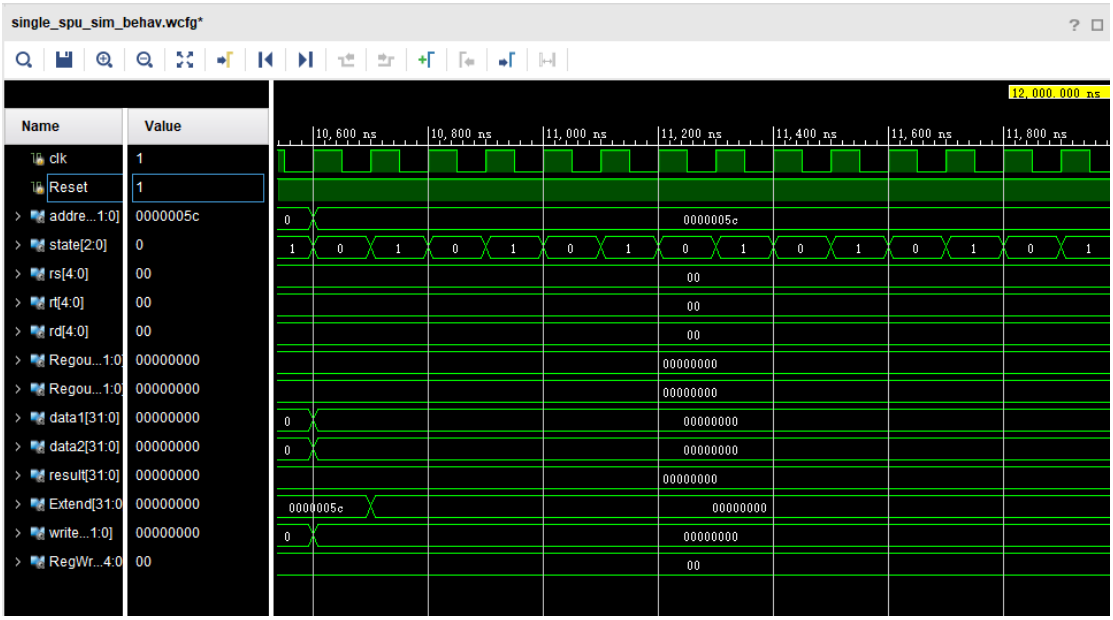


addiu $12,$12,1 $12=$12+1=-1+1=0

bltz $12,-2 $12=0,0==0,不跳转

andi $12,$2,2 $12=$2&2=2&2=2

j 0x000005C 跳转到0x50



halt　　停机指令

3、基于MIPS的简单编译器

多周期CPU实验要求实现一个编译器，输入MIPS指令，输出对应的二进制机器语言编码。编译器的实现思路其实很简单，主要就是完成一些字符串处理操作。编译器的实现主要分为两部分，一是对指令类型的判定，二是根据判定的指令类型和输入的指令信息，生成其对应的二进制机器语言编码。

（1）指令类型的判定

将实验要求中出现的所有指令和对应的编码进行对应，即可得到指令的op域。

同时，根据指令二进制机器码的类型的不同，将指令分为8类进行译码。

1)add,sub,and,slt　2)sll　3)jr　4)lw,sw　5)bltz

6)addiu,andi,ori,xori,slti,beq,bne　7)j,jal　8)halt

（2）生成二进制机器语言编码

根据不同指令类型，从输入的字符串中得到输出的机器码。

其中ins是指指令名称

1)add,sub,and,slt　　指令格式：ins　rd,rs,rt

op	rs	rt	rd	reserved

2)sll　　指令格式：ins　rd,rt,sa

op	00000	rt	rd	sa	reserved

3)jr　　指令格式：ins　rs

op	rs	00000	00000	reserved

4)lw,sw　　指令格式：ins　rt,immediate(rs)

op	rs	rt	immediate

5)bltz　　指令格式：ins　rs,immediate

op	rs	00000	immediate

6)addiu,andi,ori,xori,slti,beq,bne　指令格式：ins　rs,rt,immediate

op	rs	rt	immediate

7)j,jal　　指令格式：ins　addr

op	addr

8)halt　　指令格式：ins

op	00 0000 0000 0000 0000 0000 0000

（3）代码实现

代码见complier.cpp。

4、写板部分

写板部分将CPU的运行结果写到basy3板上。根据要求，需要根据输入输出当前地址，下一步地址，rs，$rs，rt，$rt，ALUout，datadb等数据。写板的代码可以分为4个部分，分别为CPU部分，写板数据获取部分，时钟处理部分和数码管显示部分。

CPU部分上面已经介绍，接下来介绍其他部分。

（1）算法设计

1） 写板数据获取

根据CPU运行的数据，和输入的输出类型信号(type)，生成四个数码管对应的输出数据A,B,C,D。$rs和$rt的数据在下降沿后可能会被修改，所以需要提前写入并保存。

输入：clk,type,NowAddress,NxtAddress,result,datas,datat,rs,rt,writeData。

输出：A,B,C,D。

核心代码：

```
always @(negedge clk)begin
    DS=datas;
    DT=datat;
end
always @( type or NowAddress or NxtAddress or rs or rt or result ) begin
    case (type)
        2'b00: begin
           A=NowAddress[7:4];B=NowAddress[3:0];
           C=NxtAddress[7:4];D=NxtAddress[3:0];
        end
        2'b01: begin
            A={3'b000,rs[4:4]};B=rs[3:0];
            C=DS[7:4];D=DS[3:0];
        end
        2'b10: begin
            A={3'b000,rt[4:4]};B=rt[3:0];
            C=DT[7:4];D=DT[3:0];
        end
        2'b11: begin
            A=result[7:4];B=result[3:0];
            C=writeData[7:4];D=writeData[3:0];
        end
    endcase
end
```

2）时钟处理

根据要求，需要维护两个时钟，CPU时钟和写板扫描时钟。本部分需要根据系统时钟和按键输入得到上述两个时钟clk1，clk2。针对按键输入需要进行按键消抖处理。当连续4

个时钟周期按键输入tag为1时，认为出现上升沿，clk=1。当连续100个时钟周期按键输入tag为0时，认为出现下降沿，clk=0。

输入：clk,tag,Reset,

输出：clk1,clk2

```
always @(posedge clk) begin
    cnt=cnt+1;
    if(cnt>50)begin
        clk2=!clk2;cnt=0;
    end
    if(tag==0)sum0=sum0+1;
    else sum0=0;
    if(tag==1)sum1=sum1+1;
    else sum1=0;
    if(sum1>4)now=1;
    if(sum0>100)now=0;
end
assign clk1=now;
```

3）数码管显示

根据数码管显示时钟确定数码管显示的数值和扫描电路选通的数码管编号。后调用SegLED得到8端数码管输出值。

输入：clk,A,B,C,D

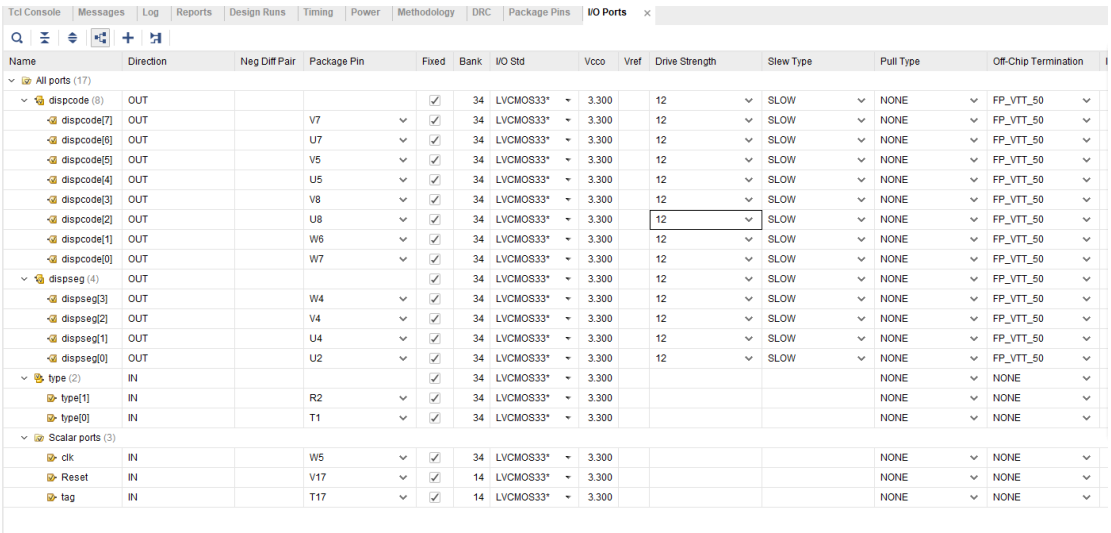
输出：dispcode,dispseg

核心代码：

```
always @(posedge clk) begin
    i=i+1;
    if(i>3)i=0;
    case (i)
        0: begin
            tmp=A; dispseg=4'b0111;
```

```
        end
        1: begin
            tmp=B; dispseg=4'b1011;
        end
        2: begin
            tmp=C; dispseg=4'b1101;
        end
        3: begin
            tmp=D; dispseg=4'b1110;
        end
    endcase
end
SegLED seg(tmp,dispcode);
Endmodule
```

（2）引脚分配



Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termination
All ports (17)												
dispcode (8)	OUT			☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
dispcode[7]	OUT		V7	☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
dispcode[6]	OUT		U7	☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
dispcode[5]	OUT		V5	☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
dispcode[4]	OUT		U5	☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
dispcode[3]	OUT		V8	☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
dispcode[2]	OUT		U8	☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
dispcode[1]	OUT		W6	☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
dispcode[0]	OUT		W7	☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
dispseg (4)	OUT			☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
dispseg[3]	OUT		W4	☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
dispseg[2]	OUT		V4	☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
dispseg[1]	OUT		U4	☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
dispseg[0]	OUT		U2	☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
type (2)	IN			☑	34	LVCMOS33*	3.300				NONE	NONE
type[1]	IN		R2	☑	34	LVCMOS33*	3.300				NONE	NONE
type[0]	IN		T1	☑	34	LVCMOS33*	3.300				NONE	NONE
Scalar ports (3)												
clk	IN		W5	☑	34	LVCMOS33*	3.300				NONE	NONE
Reset	IN		V17	☑	14	LVCMOS33*	3.300				NONE	NONE
tag	IN		T17	☑	14	LVCMOS33*	3.300				NONE	NONE

如图为引脚分配的情况。

dispcode[0]~dispcode[7]对应8跟数码管对应的接口。

dispseg[0]~dispseg[3]对应4个选择段接口。

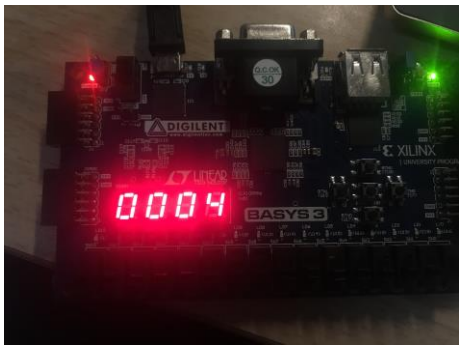
Type为显示输入选择，对应R2和T1两个开关。

时钟接W5，清空接V17，时钟触发按键接T17。

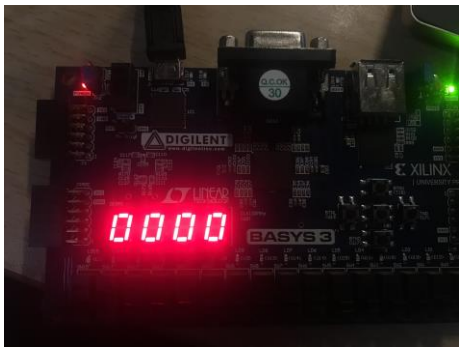
（3）实验结果与分析

声明：以下的截图中，地址部分为IF状态时的地址，rs和rt寄存器的位置和值为ID状态时的值，运算结果与数据总线部分为每条指令最后一个状态的值。

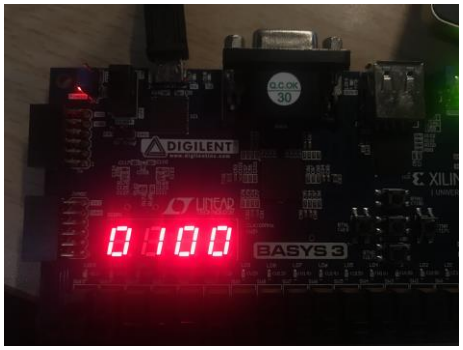
1、 addiu $1,$0,8



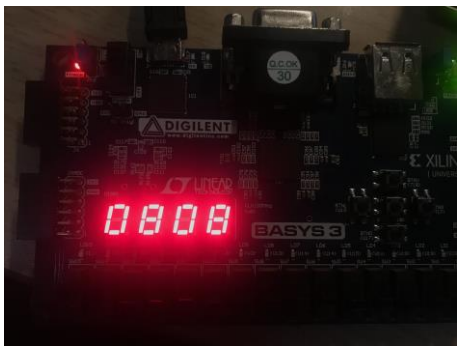
当前指令PC=0，下调指令PC=4



rs=0,$rs=0

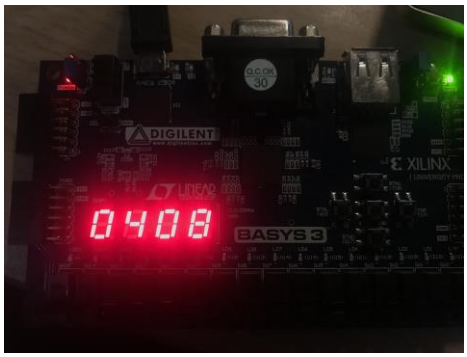


rt=1,$rt=0

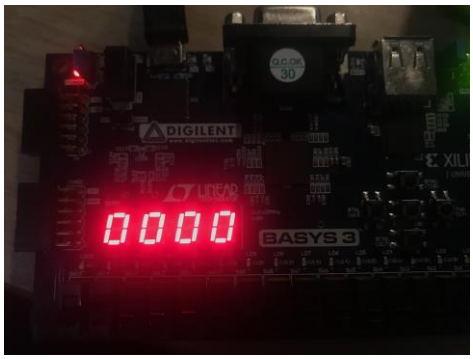


ALUout=8，DB=8

2、ori $2,$0,2

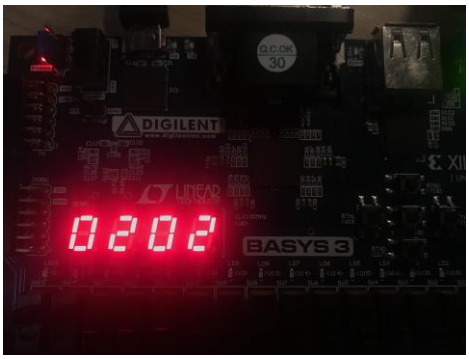


当前指令PC=4，下调指令PC=8



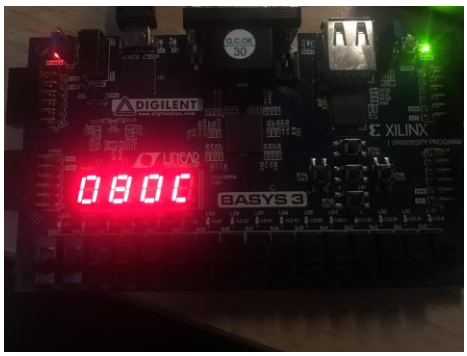
rs=0,$rs=0

rt=2,$rt=0

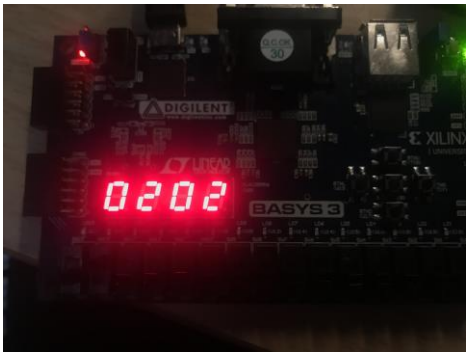


ALUout=2, DB=2

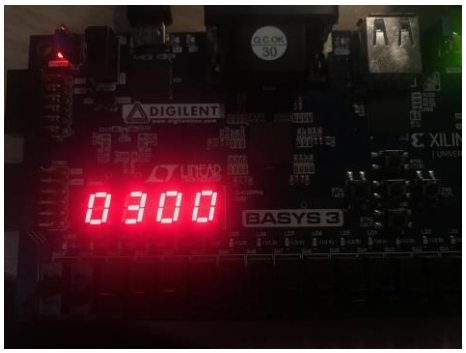
3、xori $3,$2,8



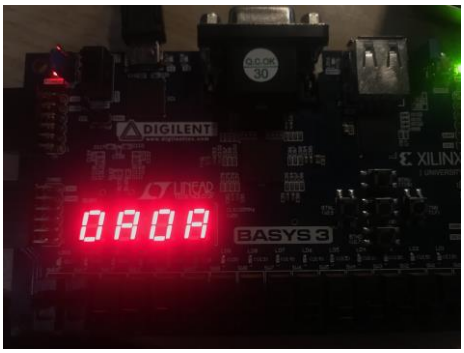
当前指令PC=8，下调指令PC=12



rs=2,$rs=2

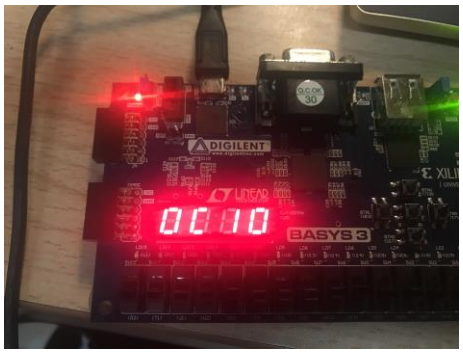


rt=3,$rt=0

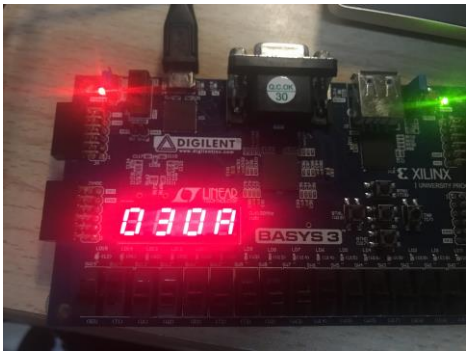


ALUout=10, DB=10

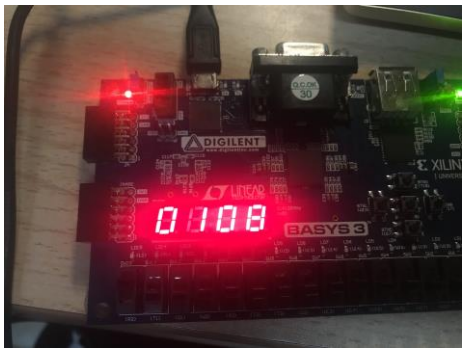
4、sub $4,$3,$1



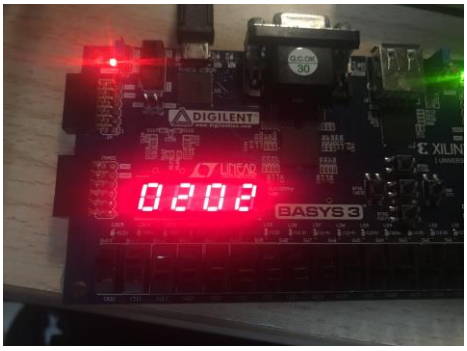
当前指令PC=12，下调指令PC=16



rs=3,$rs=10

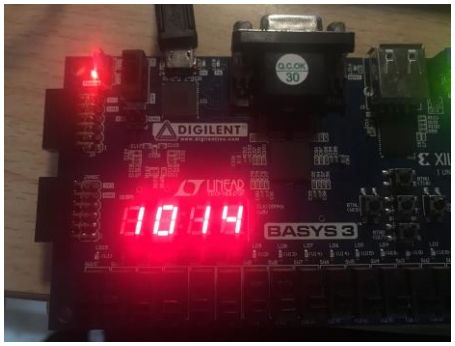


rt=1,$rt=8

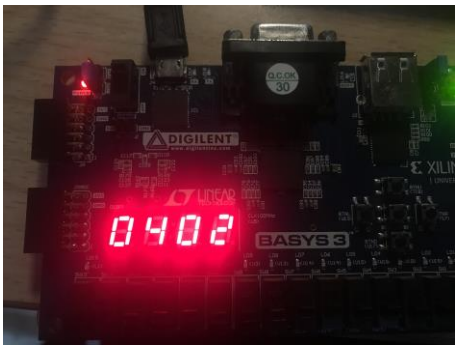


ALUout=2，DB=2

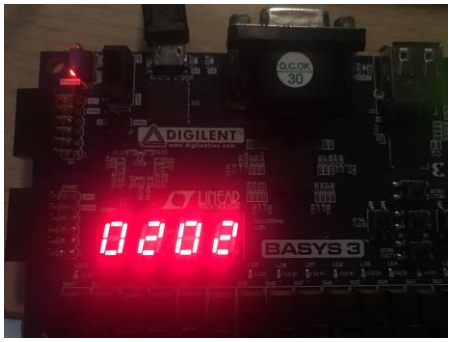
5、and $5,$4,$2



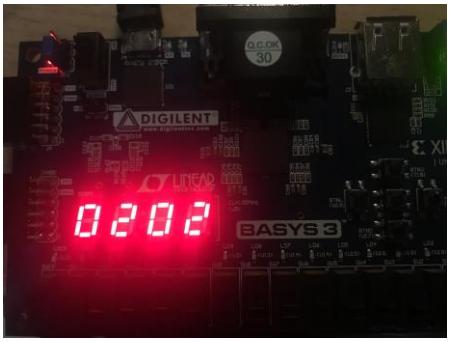
当前指令PC=16，下调指令PC=20



rs=4,$rs=2



rt=2,$rt=2



ALUout=2，DB=2

六. 实验心得

本次实验需要使用vivado软件和basy3板实现多周期CPU。

上一次实验完成过单周期CPU，本次实验中的很多代码可以直接从单周期CPU中借鉴，比如寄存器组，内存，ALU，符号拓展，选择器等部分，还有上次实验中出现很多bug，调试了很久的写板部分。因此，本次实验在代码编写与调试中少花了很多时间。

单周期CPU和多周期CPU最大的区别就是，单周期CPU的一条指令在一个时钟周期内执行，而多周期CPU的一条指令则在多个时钟周期内执行。多周期CPU一条指令有多个阶段，包括IF,ID,EXE,MEM,WB。多周期CPU最大的设计难点就是时钟周期的分配，最开始我调试了很久，后来发现每个周期都有一个特定的出发端，把这些触发端都设计成上升沿触发，剩下的设计成下降沿触发，就可以解决时钟周期分配的问题。

本次实验还增加了编译器的编写，但因为指令种类，不需要用到词法分析，语法分析这些编译原理的知识。直接对字符串进行简单处理即可，编写起来也较为方便。

总体来说，本次实验还是收获颇丰的。通过本次实验，我对多周期CPU的数据通路更加熟悉，对时序电路与逻辑电路的设计更加熟悉，也更熟悉了vivado软件的使用。

七、本学期心得

计组实验是一门很有意思的课程，在这门课上我们需要学习使用汇编语言模拟器和vivado 软件，并需要学习 verilog 语言。这门课需要完成三次大作业，分别是基于汇编模拟器的汇编语言代码编写，基于 verilog 的单周期 CPU 编写和基于 berilog 的多周期 CPU 编写。

汇编语言的实验比较简单，了解 mips 指令，对着一份 c++代码编写就好了。

上学期数字电路实验的时候曾经用过 vivado 软件，但是主要是使用 block design 部分，并没有学习 verilog 语言。学习一门新的语言还是需要花费一些时间的，verilog 语言在一些设定和编码规定上和 C 语言不同，有一些很烦人的限制规定，而且 verilog 语言是并行执行的语言，所以在设计上也有一定的难度。

而且编写 Verilog 语言需要使用 vivado 软件，vivado 软件以其编译时间长而闻名，在烧板的时候每改一次代码就需要花费十几分钟跑综合跑实现，因此也浪费了很多时间。在单周期 CPU 的实验中我花费了比较多的时间来适应 verilog 语言，尤其是在烧板的时候。但在适应了之后实验也就比较简单了。

在单周期 CPU 实验的基础上，多周期 CPU 其实也就很好实现了。多周期八成的代码可以直接借鉴单周期部分，只需要增加状态部分即可。在单周期实验中花费了很多时间的写板部分在多周期中直接照抄就好了，省去了很多时间。因此在多周期 CPU 的实现过程相比单周期就少花了很多时间，大概一两天就写完了。

通过本学期的实验，我对单周期 CPU 和多周期 CPU 的数据通路和运行方式更加了解，这对计组理论课的学习也有了很大的帮助。我还学会了 verilog 语言的编写和 vivado 软件的使用。