中山大學
SUN YAT-SEN UNIVERSITY

《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）	：	18 计教学 3 班
学 生 姓 名	：	谢俊杰
学　　号	：	18340181
时　　间	：	2019　年　11　月　17　日

成 绩 ：

实验三 ： 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)	00000 100000

功能：GPR[rd] ← GPR[rs] + GPR[rt]。

（2）sub rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100010

功能：GPR[rd] ← GPR[rs] - GPR[rt]。

（3）addiu rt , rs ,**immediate**

001001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] + sign_extend(**immediate**)； **immediate** 做符号扩展再参加“与”运算。

==> 逻辑运算指令

（4）andi rt , rs ,**immediate**

001100	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] and zero_extend(**immediate**)；**immediate** 做 0 扩展再参加“与”运算。

（5）and rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100100

功能：GPR[rd] ← GPR[rs] and GPR[rt]。

（6）ori rt , rs ,**immediate**

001101	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] or zero_extend(**immediate**)。

（7）or rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100101

功能：GPR[rd] ← GPR[rs] or GPR[rt]。

==>移位指令

（8）sll　rd, rt,sa

000000	00000	rt(5 位)	rd(5 位)	sa(5 位)	000000

功能：GPR[rd] ← GPR[rt] << sa。

==>比较指令

（9）　slti　rt, rs,**immediate**　　带符号数

001010	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if GPR[rs] < sign_extend(**immediate**)　GPR[rt] =1 else　GPR[rt] = 0。

==> 存储器读/写指令

（10）sw　rt , **offset** (rs) 写存储器

101011	rs(5 位)	rt(5 位)	**offset**(16 位)

功能：memory[GPR[base] + sign_extend(**offset**)] ← GPR[rt]。

（11) lw　rt , **offset** (rs) 读存储器

100011	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：GPR[rt] ← memory[GPR[base] + sign_extend(**offset**)]。

==> 分支指令

（12）beq　rs,rt, **offset**

000100	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：if(GPR[rs] = GPR[rt]) pc←pc + 4 + sign_extend(**offset**)<<2

else pc ←pc + 4

特别说明：**offset 是从 PC+4 地址开始和转移到的指令之间指令条数。**offset 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 **offset** 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, **offset**

000101	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：if(GPR[rs] != GPR[rt]) pc←pc + 4 + sign_extend(**offset**) <<2

else pc ←pc + 4

（14）bltz　rs, **offset**

000001	rs(5 位)	00000	**offset** (16 位)

功能：if(GPR[rs] < 0) pc←pc + 4 + sign_extend (**offset**) <<2

else pc ←pc + 4。

==>跳转指令

（15）j　addr

000010	**addr**(26 位)

功能：PC ← {PC[31:28] , **addr** , 2’b0}，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

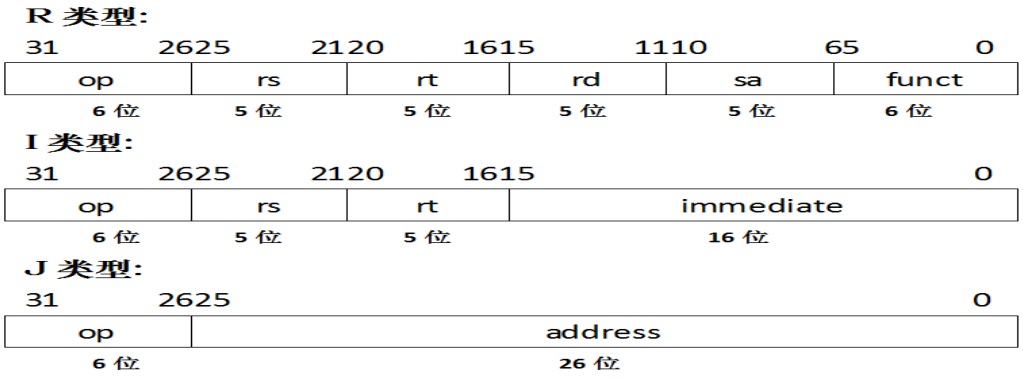
（16）halt

111111	00000000000000000000000000(26 位)

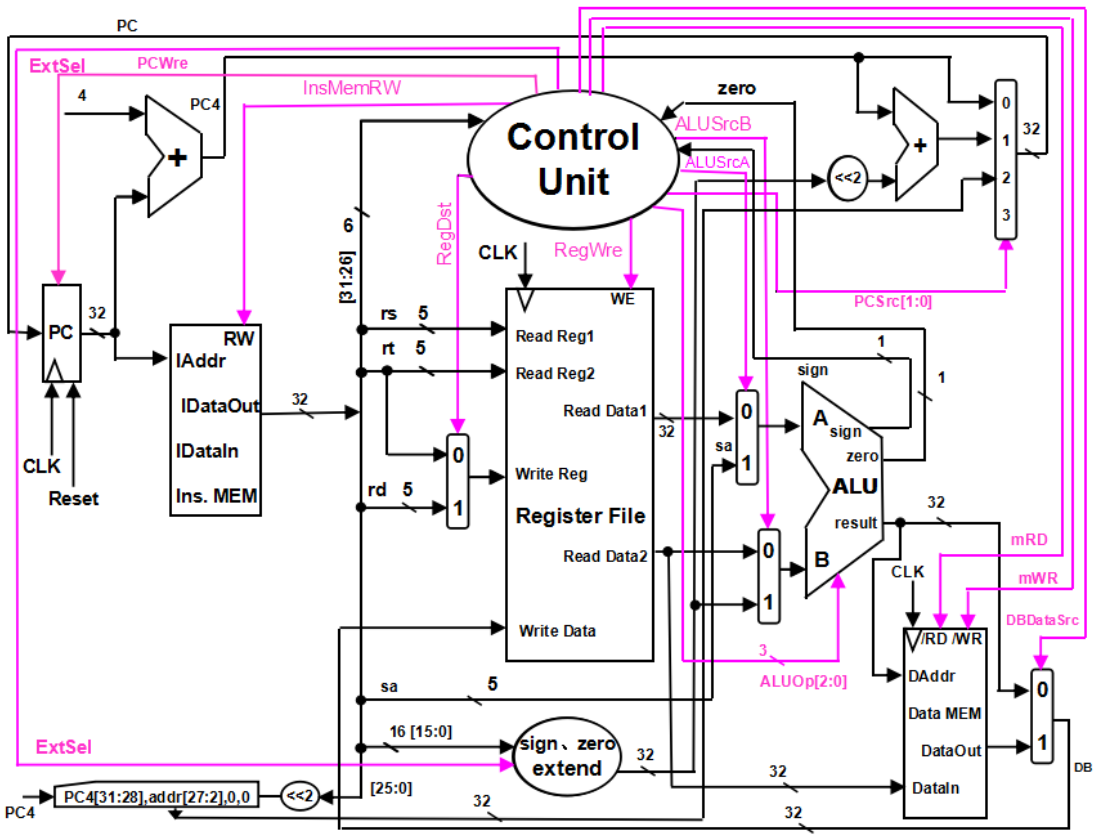
功能：停机；不改变PC的值，PC保持不变。

三. 实验原理

单周期CPU是指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。而对于该实验，CPU解释指令采取MIPS的指令的三种格式即R型、I型和J型，如图：



一个简单的能完成上述指令的单周期CPU需要以下的数据通路和控制线路图：



单周期CPU数据通路和控制线路图

其中指令存储在指令存储器中，数据存在数据存储器中。访问这些存储器时，先给出内存地址，由读或写使能信号控制操作，以避免误操作；对于寄存器组，也是先给出地址，读取数据操作不要时钟信号，但写操作则需要写使能信号和时钟下降沿的到来才能将数据写入相应寄存器。

控制单元部分需要产生各种控制信号，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出控制信号与指令间的关系表（下表），这样，从表可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系表，即真值表，再根据关系表可以将CPU的各部件间联系起来，从而实现统一。

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	来自 sign 或 zero 扩展的立即数，相关

	令：add、sub、or、and、beq、bne、bltz	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**，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：pc<—pc	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)	

指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC的改变是在时钟上升沿进行的，这样稳定性较好；否则会出现指令的写操作部分不能顺利进行。另外，值得注意的问题，设计时，用模块化的思想方法设计，

四．实验器材

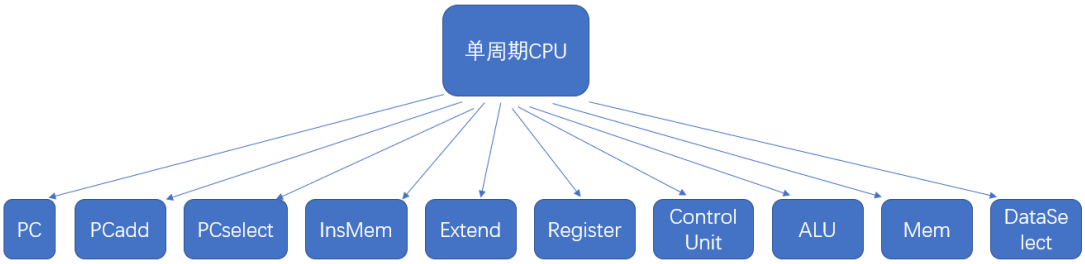
电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五．实验过程与结果

1、CPU设计的思想、方法

（1）在这里，我采用的是自顶向下的设计方法，而在要求文档中的单周期CPU数据通路和控制线路图和TA教导的用IP核连接方法的强烈暗示下，我将整个CPU的设计划分为10个具体的功能模块，分别为PC、PCadd（PC+4）、PCselect（选择PC）、InsMem（指令存储器）、Extend（符号或零扩展）、Register（寄存器组）、ControlUnit（控制单

元）、ALU（运算器）、Mem（数据存储器）、DataSelect（2选1数据选择器），即如图所示：



在实现过程中，每个功能模块的实现只简单关心其输出和输入即其内部的逻辑关系（具体的我参考数据通路图），而暂时忽略各模块间的联系。

然后将各功能模块的功能具体实现后封装成IP核（每个模块就是一个IP核），再根据数据通路的图，设计一个顶层文件，在该文件中调用上述IP核并将相应接口相连，但是要注意的是需要进行单元测试，如果在顶层设计的仿真测试出现问题时，可能会耗费大量时间去debug，所以我对每个功能模块在实现后都会进行仿真测试，以保证其独立时的正确性。

（2）其次，从数据通路的图上可以看出，各功能模块都需要控制单元产生的控制信号来进行相应指令的相应操作，同时控制单元也会接受信号如操作码、功能码从而产生相应的控制信号。所以，这里需要将指令、操作码（、功能码）、操作、控制信号对应建立一个真值表（在控制单元模块说明处示出）。

（3）各模块的具体实现和说明

1）PC模块

功能：更新当前PC值

输入：CPU时钟信号clk、PC写使能信号PCWre、复位信号Reset、下个PC的地址nexPC

输出：当前PC的地址curPC

由于PC的改变是在时钟的上升沿或Reset信号的下降沿进行，故使用边沿敏感always，并使用非阻塞赋值的方式。而PC的值的改变受PCWre和Reset两者影响：当Reset为0时，当前PC值就会被初始化为0；当Reset不为0时，PCWre为1时，改变当前PC值，即curPC等于nexPC；PCWre为0时，当前PC不能被改变。具体代码为：

```
always @(posedge clk or negedge Reset)
    begin
        if (Reset == 0)
         curPC <= 0;
        else
         if (PCWre == 1)
             curPC <= nexPC;
         else
             curPC <= curPC;
    end
```

2）PCadd模块

功能：将输入的PC值+4

输入：输入的当前PC值

输出：输出的PC+4的值

由于该模块仅是组合逻辑电路，将输出定义为wire类型，使用assign语句并使用阻塞赋值。具体代码如下：

```
assign o_PC1 = i_PC1 + 4;
```

3）PCselect模块

功能：根据对输入的数据进行运算和选择，输出下一条PC地址

输入：输入的已+4的PC、PCSrc、偏移量立即数immediate、地址address

输出：选择得到的下一条PC地址

该模块的主要逻辑实则是4选1数据选择器，根据PCSrc的值去选择下一条PC的地址，具体的选择如下表：

PCSrc[1:0]	nexPC
00	PC<-PC+4
01	PC<-PC+4+immediate*4
10	PC<-{(PC+4)[31:28], address[25:0], 2’b00}
11	PC<-PC

由于该表中PCSrc为11时的nexPC如何赋值未定义，故在此我定义为PC，用以停机指令，而注意到该表中右边的（PC+4）就是输入的PC地址。同时，输入的immediate是从

PC+4 地址开始和转移到的指令之间指令条数，在计算地址值时需要左移2位即乘以4。具体代码如下：

```
always @(*)
    begin
    case (PCSrc)
        2'b00: o_PC2 = i_PC2;
        2'b01: o_PC2 = i_PC2 + immediate * 4;
        2'b10:
         begin
            PC_high4 = i_PC2;///
            o_PC2 = {PC_high4[31:28], address[25:0], 2'b00};
         end
        default: o_PC2 = i_PC2 - 4;
    endcase
    end
```

4）InsMem模块

功能：根据输入的当前PC地址，在预先存储指令的指令存储器中寻找到相应指令的二进制码并根据R型、I型、J型指令完成译码功能，即将当前指令的各部分分别取出并输出。

输入：指令寄存器的读使能信号InsMemRW、当前PC地址i_addr

输出：指令各部分划分的数据集合（如op、rs、rt、rd等）

由于需要存储的指令条数为17条，而每条指令为32位，根据要求，指令存储器存储单元宽度为8位，所以指令存储器中至少需要68个存储单元，而在此我使用了128个8位存储单元的指令存储器memory：

```
reg [7:0]memory[0:127];
```

在模块开始运行时，从txt文件读入程序指令到指令存储器memory中，注意到MIPS的大端存储方式，指令高位放在低地址上：

```
initial
begin
    $readmemb("D:/code/freshman/digital/InsMem/ins.txt", memory);
    currrent_instrution = 0;
end
```

再根据输入的PC地址i_addr在指令存储器进行定位，取出当前相应的指令，由于MIPS是大端方式存储，高位存放在低地址，故低地址的数据拼接在高位：

```
always @(*)
begin
    currrent_instrution = {memory[i_addr][7:0], memory[i_addr+1][7:0],
    memory[i_addr+2][7:0], memory[i_addr+3][7:0]};
end
```

然后再根据R、I、J型指令的不同格式，分别取出各段数据输出：

```
assign op = currrent_instrution[31:26];
assign rs = currrent_instrution[25:21];
assign rt = currrent_instrution[20:16];
assign rd = currrent_instrution[15:11];
assign sa = currrent_instrution[10:6];
assign funct = currrent_instrution[5:0];
assign immediate = currrent_instrution[15:0];
assign address = currrent_instrution[25:0];
```

5）Extend模块

功能：根据符号/零选择信号对输入数据进行对应的符号/零扩展

输入：16位立即数i_immediate、选择信号ExtSel

输出：32位扩展后的立即数o_imme

该模块主要逻辑也是一个2选1数据选择器，当ExtSel为0时，高位补0，进行零扩展；当ExtSel为1时，高位补符号位，进行符号扩展。

```
always @(*)
begin
    if (ExtSel == 0)
        o_imme = {16'h0000, i_immediate[15:0]};
    else
    begin
        if (i_immediate[15] == 0)
            o_imme = {16'h0000, i_immediate[15:0]};
        else
            o_imme = {16'hffff, i_immediate[15:0]};
    end
end
```

6）Register模块

功能：给出两个寄存器的地址，对这两个寄存器进行读取数据的操作，同时给出要写入寄存器的地址和要写入的数据，在写使能信号为1且在时钟下降沿处，将数据写入相应寄存器中。

输入：时钟clk、写使能信号RegWre、读取的寄存器地址ReadReg1、ReadReg2、写入的寄存器地址WriteReg、写入的数据WriteData

输出：读取到的寄存器中的数据ReadData1、ReadData2

首先，在该模块开始运行时，需要对开辟的寄存器组（32个32位存储单元）进行初始

化为0：

```
reg [31:0]Reg[0:31];
integer i;

initial
begin
    for (i = 0; i < 32; i = i + 1)
        Reg[i] <= 0;
end
```

其次，由于读操作时不需要时钟信号，输出端可以直接输出相应数据；而对于写操作，由于0号寄存器不能被更改，故当写使能信号为1且要写入的寄存器地址不为0时，则可以在时钟下降沿处写入相应数据，否则不能写入。

```
assign ReadData1 = Reg[ReadReg1];
assign ReadData2 = Reg[ReadReg2];

always @(negedge clk)
begin
    if (WriteReg != 0 && RegWre == 1)
        Reg[WriteReg] <= WriteData;
end
```

7）ControlUnit模块

功能：根据6位操作码、6位功能码、零信号、符号信号去选择生成各控制信号的值，使其他模块能够按照指令正常地运行。

输入：零信号zero、符号信号sign、6位操作码op、6位功能码funct

输出：真值表中除输入外的信号

该模块的主要逻辑也是数据选择器，由于数据项数比较多，故在代码中使用case语句对各控制信号根据下真值表进行逐项赋值，部分代码如下：

```
6'b001001: begin//addiu
    PCWre = 1;
    ALUSrcA = 0;
    ALUSrcB = 1;
    DBDataSrc = 0;
    RegWre = 1;
    InsMemRW = 1;
    mRD = 0;
    mWR = 0;
    RegDst = 0;
    ExtSel = 1;
    PCSrc = 2'b00;
    ALUOp = 3'b000;
end
```

ins	000000	000000	001001	001100	000000	001101	000000	000000	001010	101011	100011	000100	000101	000001	000010	111111
funct	100000	100010			101000		100101	000000								
Op	add	Sub	Addiu	Andi	And	Ori	Or	Sll	Slti	Sw	Lw	Beq	Bne	Bltz	J	Halt
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
PCWre	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
ALUsrcA	0	0	0	0	0	0	0	1	0	0	0	0	0	0	X	X
ALUsrcB	0	0	1	1	0	1	0	0	1	1	1	0	0	0	X	X
DBDataSrc	0	0	0	0	0	0	0	0	0	X	1	X	X	X	X	X
RegWre	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0
InsMemRW	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
mRD	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
mWR	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
RegDst	1	1	0	0	1	0	1	1	0	X	0	X	X	X	X	X
ExtSel	X	X	1	0	X	0	X	0	1	1	1	1	1	1	X	X
PCSrc[1..0]	00	00	00	00	00	00	00	00	00	00	00	00/01	00/01	00/01	10	11
ALUOp[2..0]	000	001	000	100	100	011	011	010	110	000	000	001	001	000	xxx	xxx

8）ALU模块

功能：根据控制单元模块生成的ALUOp信号，选择相应的算术逻辑运算，同时根据运算后得到的结果对zero和sign输出信号赋值。

输入：操作数A、操作数B和算术逻辑选择信号ALUOp

输出：结果result、零信号zero、符号信号sign

算数逻辑选择信号和相应的算数逻辑是一一对应的关系，具体如下表所示（该表来源于要求文档）。

ALUOp[2..0]	功能	描述
000	Y = A + B	加
001	Y = A － B	减
010	Y = B<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y= (A<B) ?1: 0	比较 A 与 B 不带符号
110	Y=(((rega<regb) && (rega[31] == regb[31])) ‖((rega[31] ==1 && regb[31] == 0))) ? 1:0	比较 A 与 B 带符号
111	Y = A ⊕ B	异或

而对于生成的信号，当result为0时，zero信号为1，反之为0；而当result<0时，即result的最高位为1时，sign信号为1，反之为0，具体代码如下：

```
always @(*)
begin
    case(ALUOp)
        3'b000: result = input_A + input_B;
        3'b001: result = input_A - input_B;
        3'b010: result = input_B << input_A;
        3'b011: result = input_A | input_B;
        3'b100: result = input_A & input_B;
        3'b101: result = (input_A < input_B) ? 1 : 0;
        3'b110: result = (((input_A < input_B) &&
            (input_A[31] == input_B[31])) ||
            ((input_A[31] == 1) && (input_B[31] == 0))) ? 1 : 0;
        3'b111: result = input_A ^ input_B;
        default: result = 0;
    endcase
end

assign zero = (result == 0) ? 1 : 0;
assign sign = (result[31] == 0) ? 0 : 1;//////////////////
```

9）Mem模块

功能：根据相应控制信号，读取对应地址的数据或由对应地址对存储器进行写入数据操

作。

输入：时钟clk、读使能信号mRD、写使能信号mWR、数据地址DataAddr、要写入的数据DataIn

输出：读取出的相应的数据DataOut

在该模块中，首先申请128个8位存储单元的空间作为存储器的空间，然后初始化为0。

```
reg [7:0]memory[0:127];
integer i;
initial
begin
    for (i = 0; i < 128; i = i + 1)//初始化
        memory[i] <= 0;
end
```

在时钟下降沿来临时且写使能信号为1时，将输入的32位数据拆分为4个8位的数据，由大端方式存储到存储器中，高位存放在低地址；在读使能信号为1时，将相应地址的数据拼接成为一个32位的数据输出。具体代码如下：

```
always @(negedge clk)
begin
    if (mWR == 1)
    begin
        memory[DataAddr * 4] <= DataIn[31:24];
        memory[DataAddr * 4 + 1] <= DataIn[23:16];
        memory[DataAddr * 4 + 2] <= DataIn[15:8];
        memory[DataAddr * 4 + 3] <= DataIn[7:0];
    end
end

always @(*)
begin
    if (mRD == 1)
    begin
        DataOut <= {memory[DataAddr * 4][7:0],
            memory[DataAddr * 4 + 1][7:0],
            memory[DataAddr * 4 + 2][7:0],
            memory[DataAddr * 4 + 3][7:0]};
    end
    else
        DataOut <= 32'hzzzzzzzz;
end
```

10）DataSelect模块

功能：数据选择器，根据信号译码选择数据

输入：信号signal、两个数据input_A、input_B

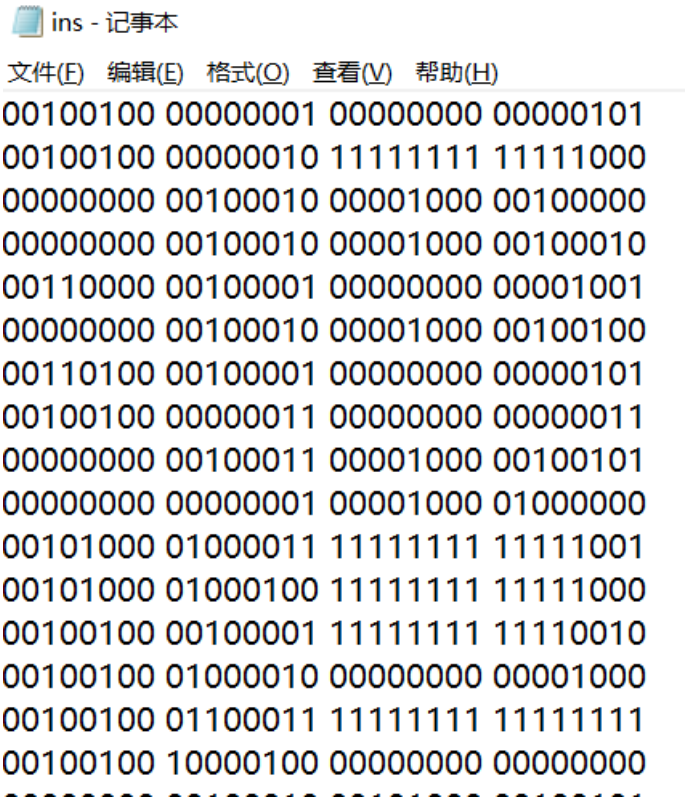
输出：选择的其中一个输入数据

该模块主要是数据选择器，将需要多个选择输入数据的模块连接起来。具体代码如下：

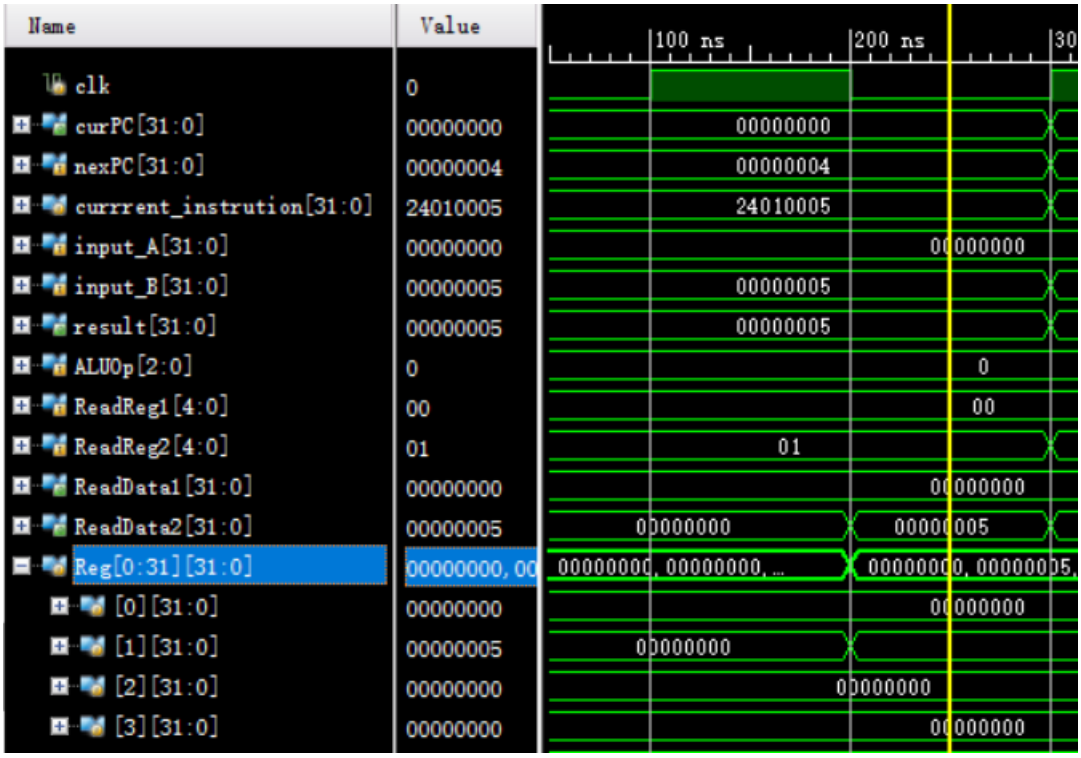
```
assign output_C2 = (signal2 == 0) ? input_A2 : input_B2;
```

2、验证CPU正确性

对CPU的正确性的检测中，使用给出的实验检查时的测试程序段进行测试，由于需要检测的是每一指令的正确性，便能得出CPU的正确性，故对于一些重复出现的同类型指令，说明部分侧重在第一次出现的地方。首先将给出的程序段转换成32位二进制指令，每8位用空格隔开，并存放指令寄存器部分要读取的文件中，部分指令如下图：



指令1：addiu $1,$0,5

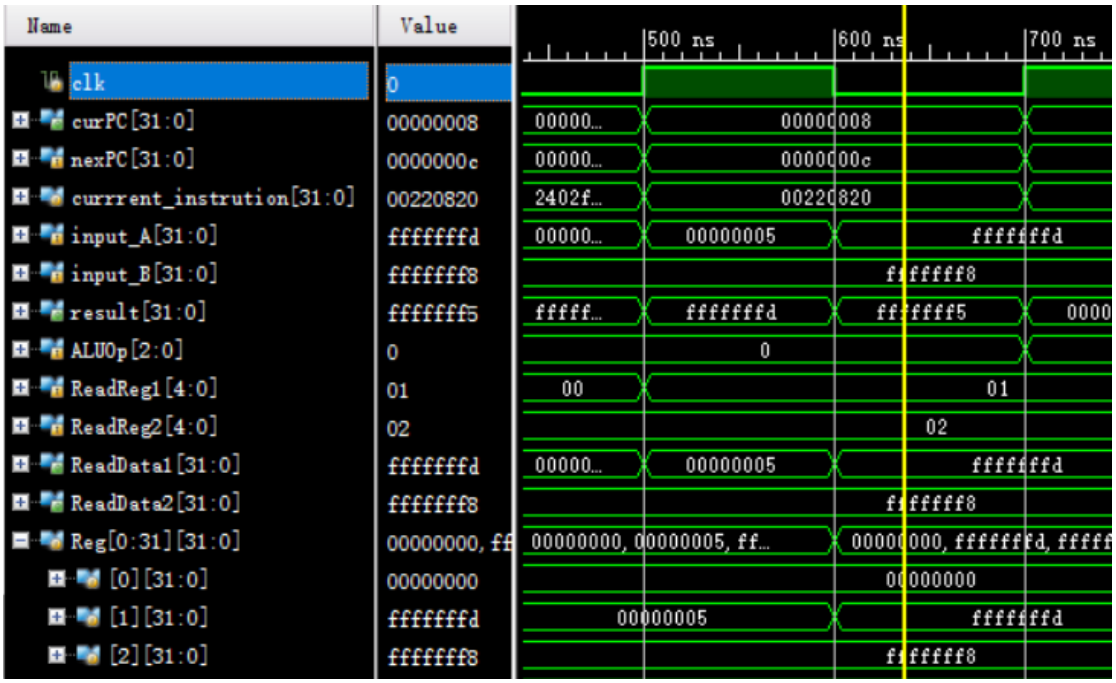


（1）在波形图中，curPC表示当前指令的地址为0x00000000，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）addiu指令为I型指令，ALU的输入分别为rs（0号寄存器）、立即数，而$0为0，立即数作符号扩展操作，故input_A、input_B分别为0、5，而ALU的操作应为两数相加，故ALUOp为000，也与上图符合，运算后的结果为5即result。

（3）addiu指令在运算后时钟下降沿处结果需要写入rt（1号寄存器）中，故可在寄存器组的1号寄存器看见已被写为result——5。

指令2：add $1,$1,$2

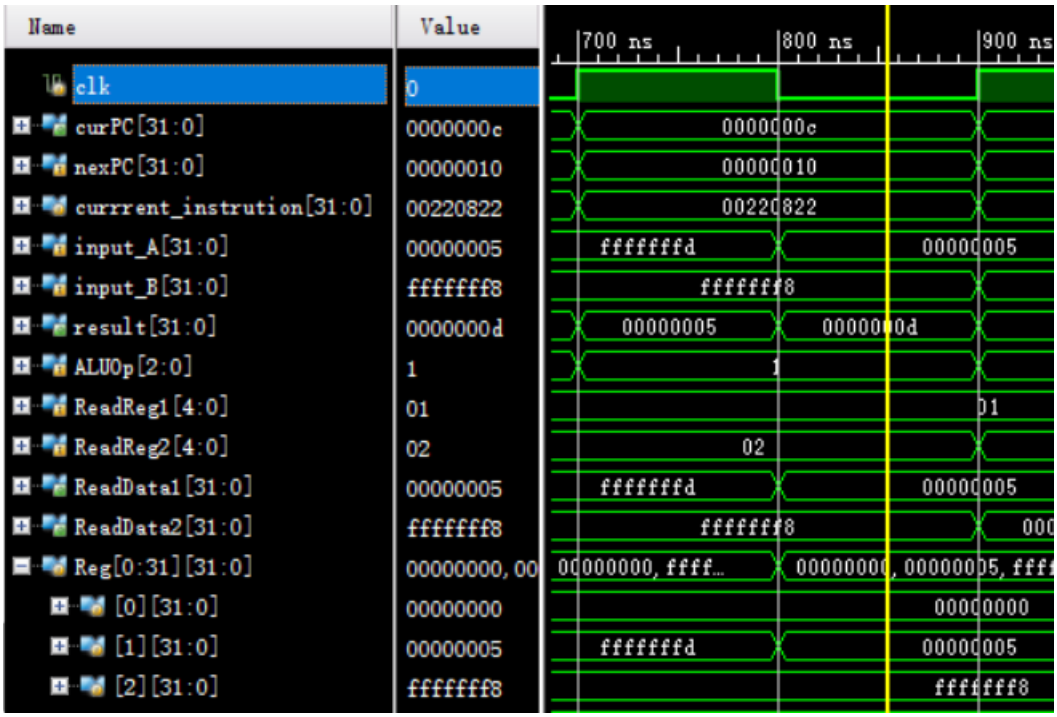


（1）在波形图中，curPC表示当前指令的地址为0x00000008，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）add指令为R型指令，ALU的输入分别为rs（1号寄存器）、rt（2号寄存器），而在时钟下降沿前$1为5，$2为-8，故input_A、input_B分别为5、-8，在时钟下降沿后此处的input发生变化是因为$1被改写发生变化，导致读取了变化后的值，但是并没有被写入对后续指令无影响，而ALU的操作应为两数相加，故ALUOp为000，也与上图符合，运算后的结果为-3即result。

（3）add指令在运算后时钟下降沿处结果需要写入rd（1号寄存器）中，故可在寄存器组的1号寄存器看见已被写为result——-3。

指令3：sub $1,$1,$2

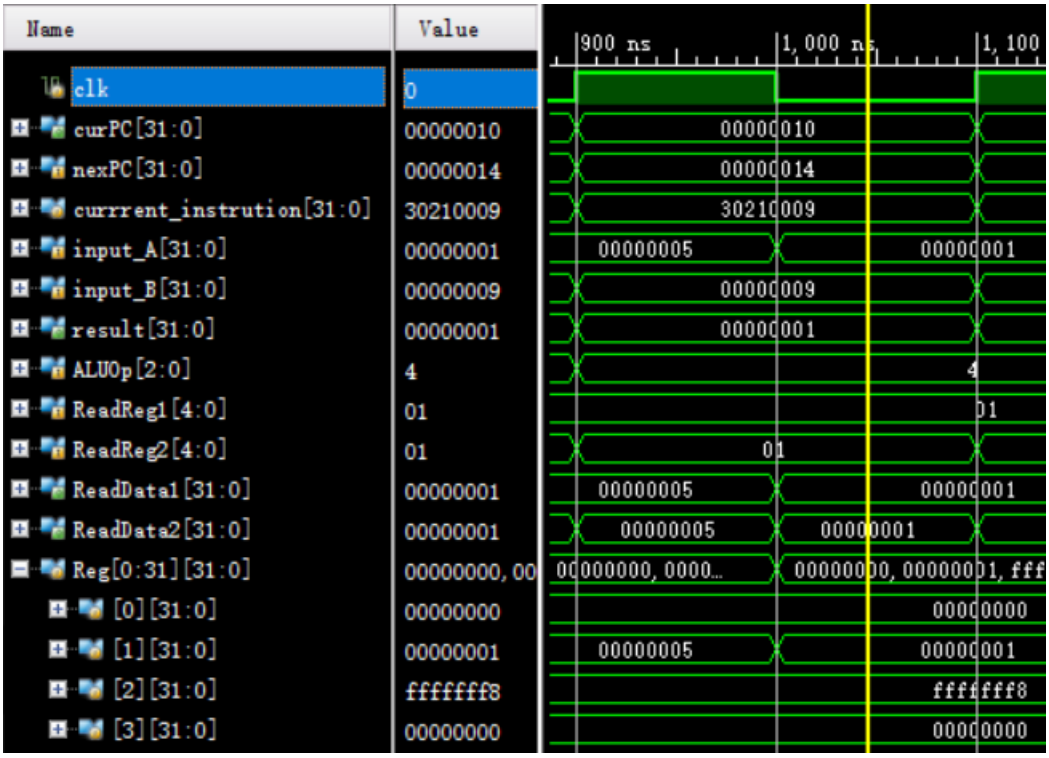


（1）在波形图中，curPC表示当前指令的地址为0x0000000c，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）sub指令为R型指令，ALU的输入分别为rs（1号寄存器）、rt（2号寄存器），而在时钟下降沿前$1为-3，$2为-8，故input_A、input_B分别为-3、-8，在时钟下降沿后此处的input发生变化是因为$1被改写发生变化，导致读取了变化后的值，但是并没有被写入对后续指令无影响，而ALU的操作应为两数相减，故ALUOp为001，也与上图符合，运算后的结果为5即result。

（3）sub指令在运算后时钟下降沿处结果需要写入rd（1号寄存器）中，故可在寄存器组的1号寄存器看见已被写为result——5。

指令4：andi $1,$1,9

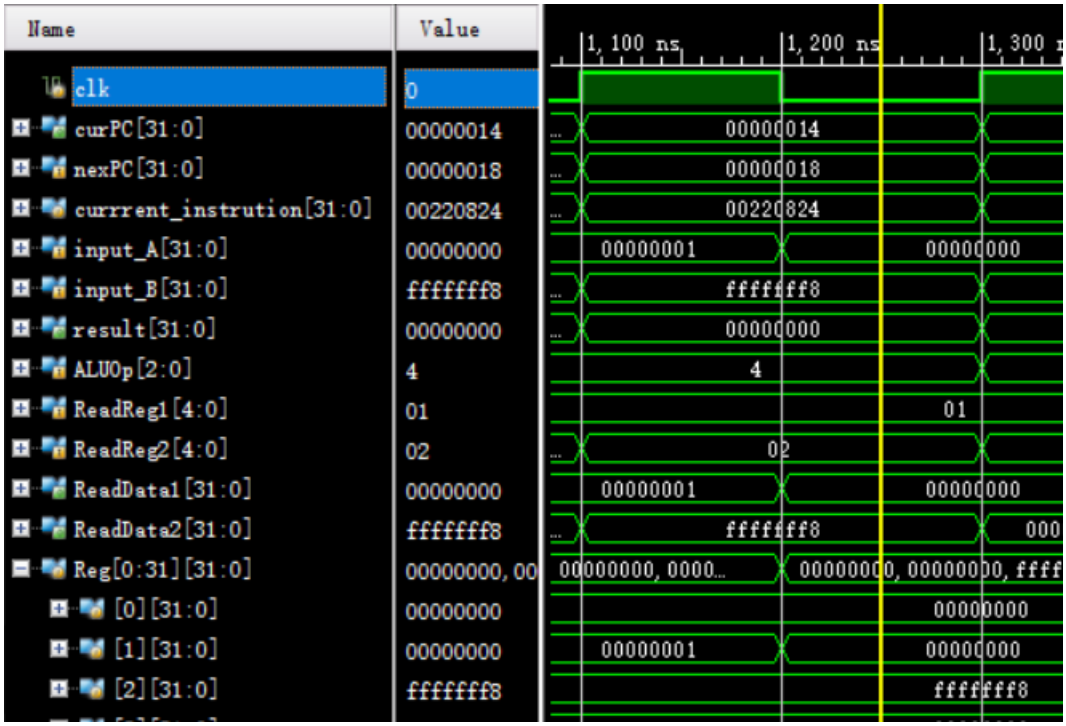


（1）在波形图中，curPC表示当前指令的地址为0x00000010，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）andi指令为I型指令，ALU的输入分别为rs（1号寄存器）、立即数，而$1为5，立即数作符号扩展操作，故input_A、input_B分别为5、9，而ALU的操作应为两数相与，故ALUOp为100，也与上图符合，运算后的结果为1即result。

（3）andi指令在运算后时钟下降沿处结果需要写入rt（1号寄存器）中，故可在寄存器组的1号寄存器看见已被写为result——1。

指令5：and $1,$1,$2

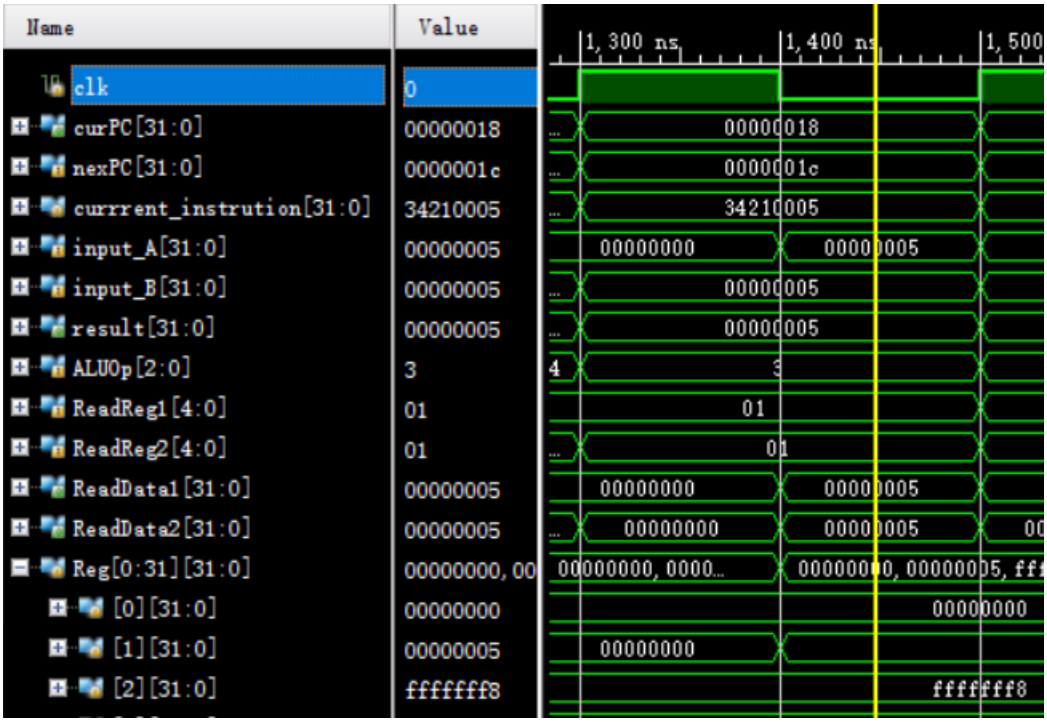


（1）在波形图中，curPC表示当前指令的地址为0x00000014，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）and指令为R型指令，ALU的输入分别为rs（1号寄存器）、rt（2号寄存器），而在时钟下降沿前$1为1，$2为-8，故input_A、input_B分别为1、-8，在时钟下降沿后此处的input发生变化是因为$1被改写发生变化，导致读取了变化后的值，但是并没有被写入对后续指令无影响，而ALU的操作应为两数相与，故ALUOp为100，也与上图符合，运算后的结果为0即result。

（3）and指令在运算后时钟下降沿处结果需要写入rd（1号寄存器）中，故可在寄存器组的1号寄存器看见已被写为result——0。

指令6：ori $1,$1,5

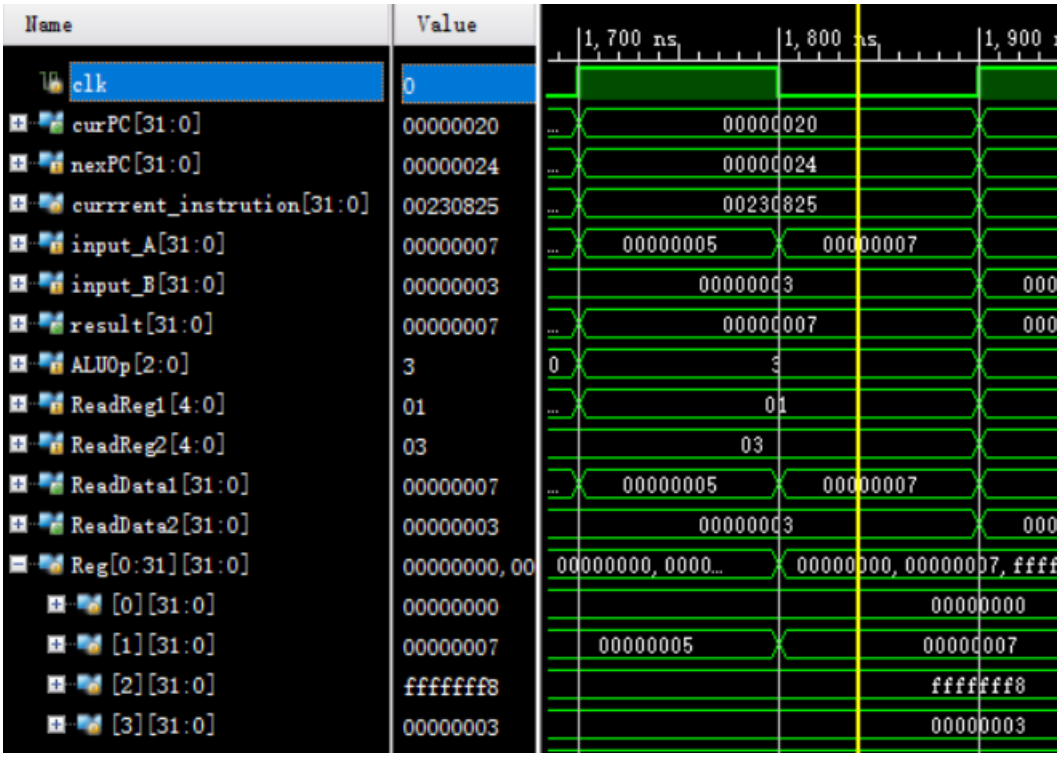


（1）在波形图中，curPC表示当前指令的地址为0x00000018，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）ori指令为I型指令，ALU的输入分别为rs（1号寄存器）、立即数，而$1的值原为0，立即数作符号扩展操作，故input_A、input_B分别为0、5，而ALU的操作应为两数相或，故ALUOp为011，也与上图符合，运算后的结果为5即result。

（3）ori指令在运算后时钟下降沿处结果需要写入rt（1号寄存器）中，故可在寄存器组的1号寄存器看见已被写为result——5。

指令7：or $1,$1,$3

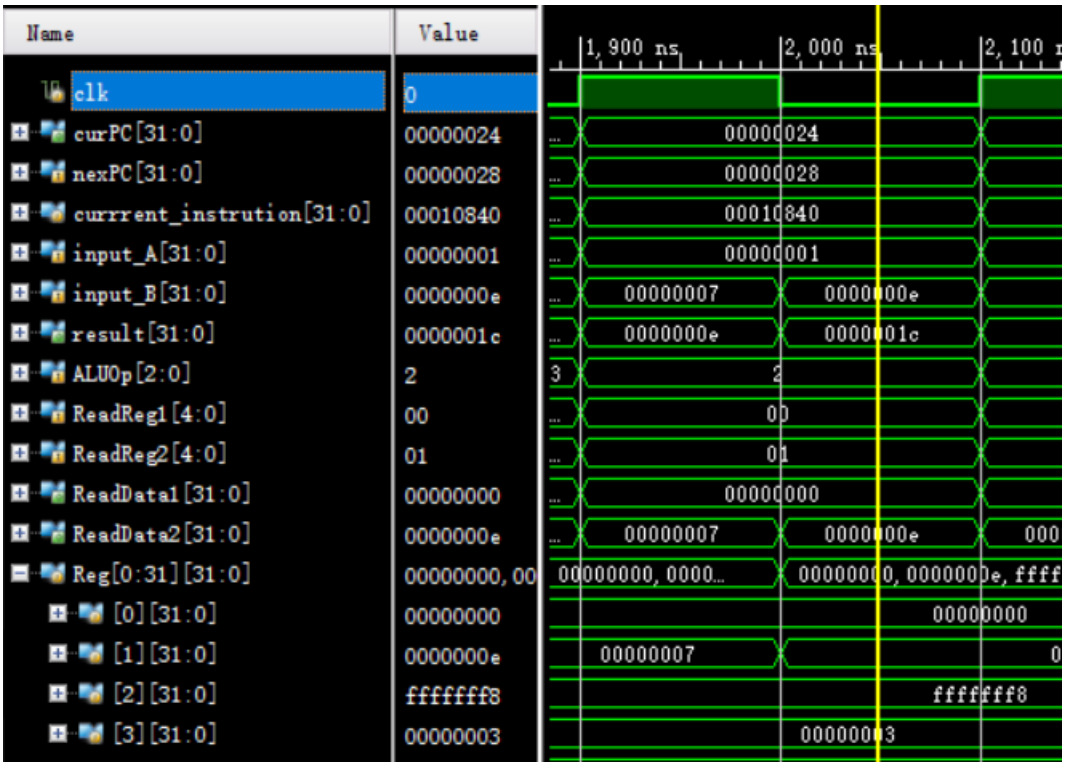


（1）在波形图中，curPC表示当前指令的地址为0x00000020，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）or指令为R型指令，ALU的输入分别为rs（1号寄存器）、rt（3号寄存器），而在时钟下降沿前$1为5，$3为3，故input_A、input_B分别为5、3，在时钟下降沿后此处的input发生变化是因为$1被改写发生变化，导致读取了变化后的值，但是并没有被写入对后续指令无影响，而ALU的操作应为两数相或，故ALUOp为011，也与上图符合，运算后的结果为7即result。

（3）or指令在运算后时钟下降沿处结果需要写入rd（1号寄存器）中，故可在寄存器组的1号寄存器看见已被写为result——7。

指令8：sll $1,$1,1

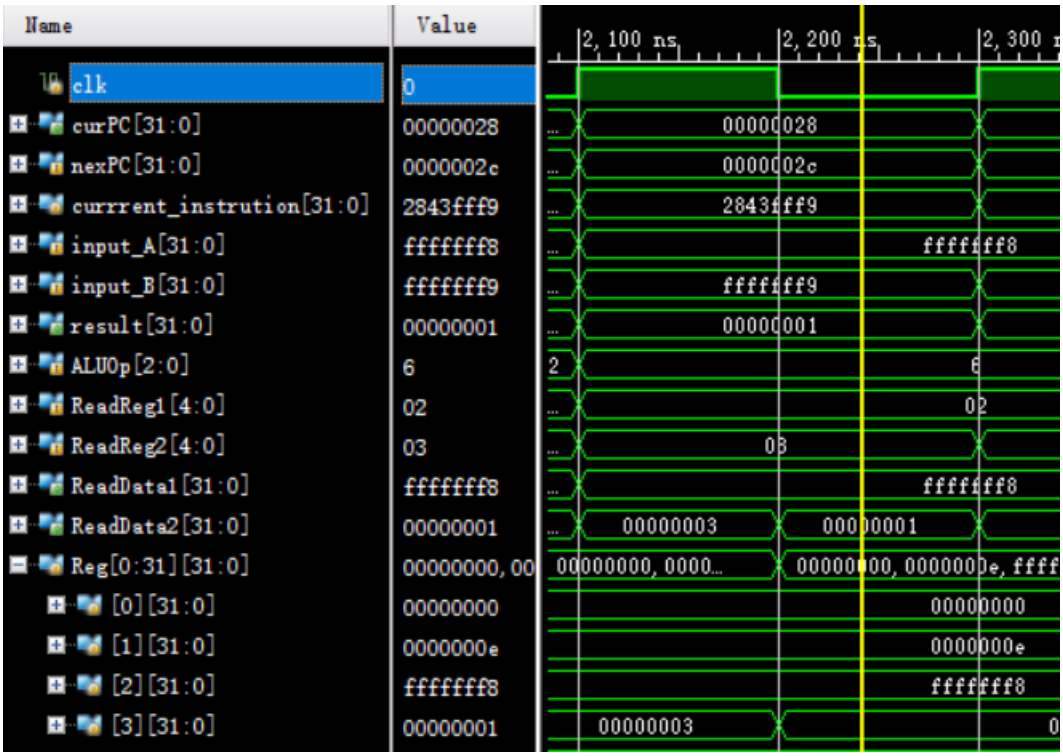


（1）在波形图中，curPC表示当前指令的地址为0x00000024，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）sll指令为I型指令，ALU的输入分别为偏移量sa、rt（1号寄存器），而$1的值原为7，sa作零扩展操作，故input_A、input_B分别为1、7，而ALU的操作应为左移操作，故ALUOp为010，也与上图符合，运算后的结果为e即result。

（3）sll指令在运算后时钟下降沿处结果需要写入rd（1号寄存器）中，故可在寄存器组的1号寄存器看见已被写为result——e。

指令9：slti $3,$2,-7

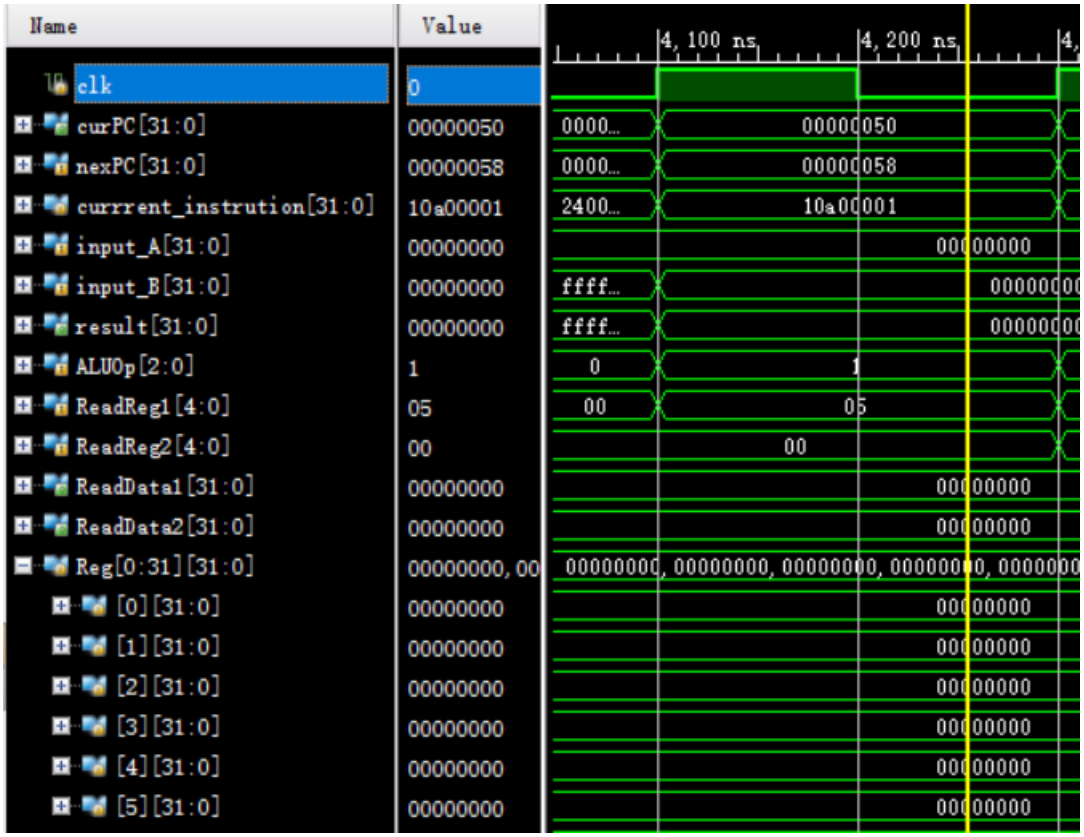


（1）在波形图中，curPC表示当前指令的地址为0x00000028，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）slti指令为I型指令，ALU的输入分别为rs（2号寄存器）、立即数，而$2的值原为-8，立即数作符号扩展操作，故input_A、input_B分别为-8、-7，而ALU的操作应为带符号的比较操作，故ALUOp为110，也与上图符合，运算后的结果为1即result。

（3）slti指令在运算后时钟下降沿处结果需要写入rt（3号寄存器）中，故可在寄存器组的1号寄存器看见已被写为result——1。

指令10：beq $5,$0,1

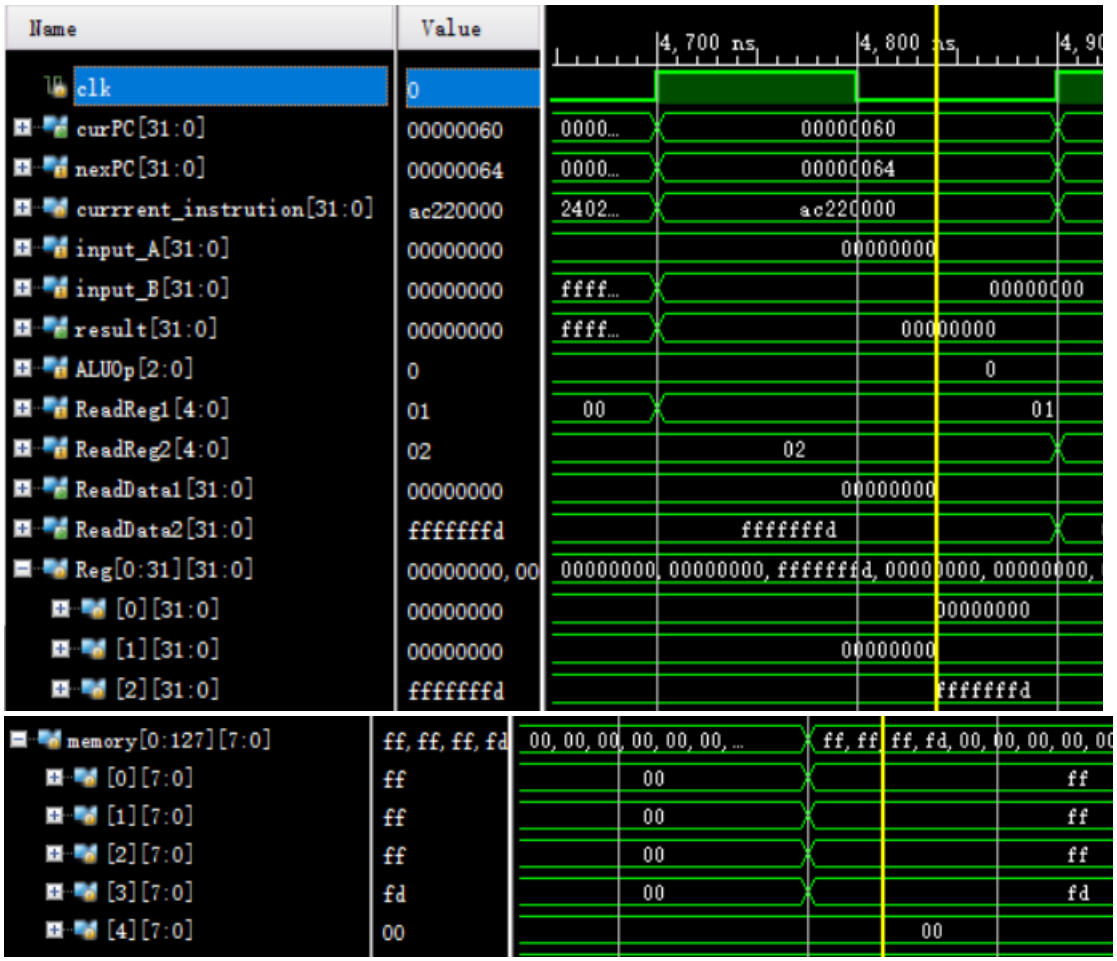


（1）在波形图中，curPC表示当前指令的地址为0x00000050，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）beq指令为I型指令，ALU的输入分别为rs（5号寄存器）、rt（0号寄存器），而$5的值为0，$0的值为0，故input_A、input_B分别为0、0，而ALU的操作应为两数相减操作，故ALUOp为001，也与上图符合，运算后的结果为0即result。

（3）beq指令根据ALU的运算结果为0，从而进行跳转，而跳转的地址为当前PC+4+immediate*4即PC+8，与图中nexPC相符。

指令11：sw $2,0($1)

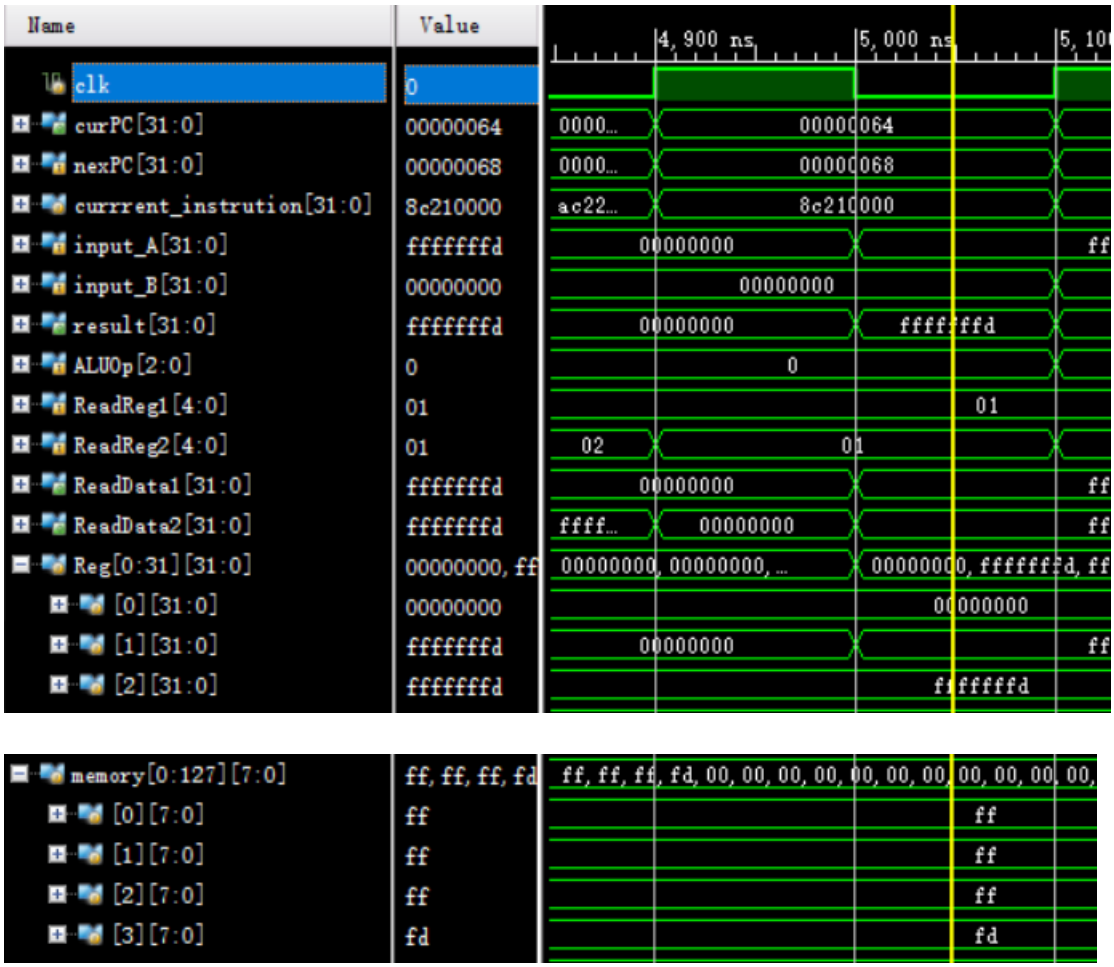


（1）在波形图中，curPC表示当前指令的地址为0x00000060，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）sw指令为I型指令，ALU的输入分别为rs（1号寄存器）、立即数，而$1的值原为0，立即数作符号扩展操作，故input_A、input_B分别为0、0，而ALU的操作应为两数相加操作，故ALUOp为000，也与上图符合，运算后的结果为0即result。

（3）sw指令在运算后时钟下降沿处需要将$2的数据写入存储器中，地址为result——0，所以查看存储器的值时，可发现是以大端方式存在了下标0-3的4个8位存储单元中，且数值与$2的一致。

指令12：lw $1,0($1)

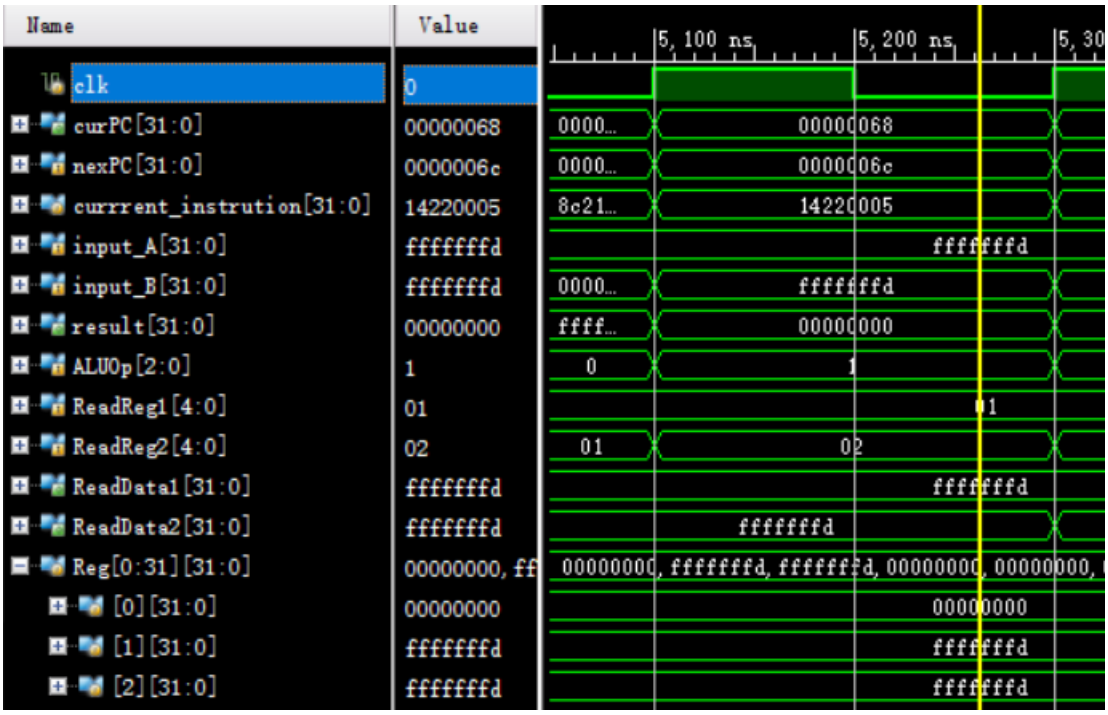


（1）在波形图中，curPC表示当前指令的地址为0x00000064，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）lw指令为I型指令，ALU的输入分别为rs（1号寄存器）、立即数，而$1的值原为0，立即数作符号扩展操作，故input_A、input_B分别为0、0，而ALU的操作应为两数相加操作，故ALUOp为000，也与上图符合，运算后的结果为0即result。

（3）lw指令在ALU运算中得到的结果为要在存储器中取数的首地址，即0-3这4个8位存储单元拼接起来的数值，并存放到$1中，故可查看$1发现其存放的数值也确与存储器中的值一致，即成功取数。

指令13：bne $1,$2,5

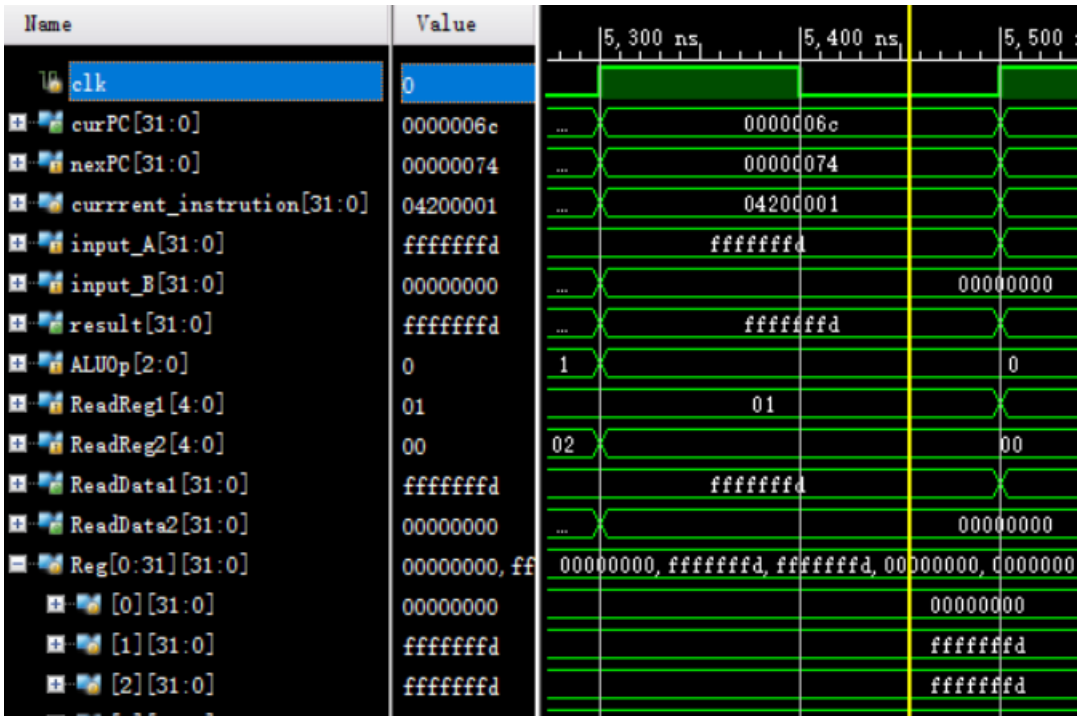


（1）在波形图中，curPC表示当前指令的地址为0x00000068，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）bne指令为I型指令，ALU的输入分别为rs（1号寄存器）、rt（2号寄存器），而$1的值为-3，$2的值为-3，故input_A、input_B分别为-3、-3，而ALU的操作应为两数相减操作，故ALUOp为001，也与上图符合，运算后的结果为0即result。

（3）bne指令根据ALU的运算结果为0，从而不进行跳转，即顺序进行，下一条地址为当前PC+4，与图中nexPC相符。

指令14：bltz $1,1

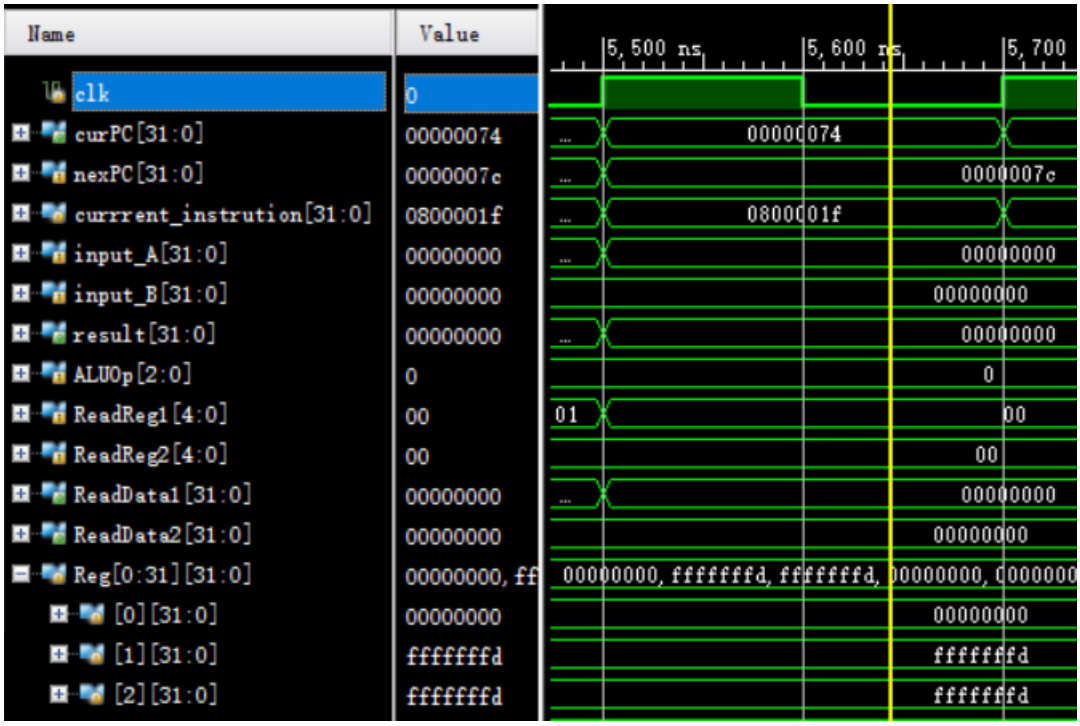


（1）在波形图中，curPC表示当前指令的地址为0x0000006c，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）bltz指令中ALU的输入分别为rs（1号寄存器）、0（0号寄存器），而$1的值为-3，$0的值为0，故input_A、input_B分别为-3、0，而ALU的操作应为两数相减操作，故ALUOp为001，也与上图符合，运算后的结果为-3即result。

（3）bltz指令根据ALU的运算结果为-3，符号位为1，则sign控制信号为1，从而进行跳转，即跳转到的地址为当前PC+4+immediate*4即PC+8，与图中nexPC相符。

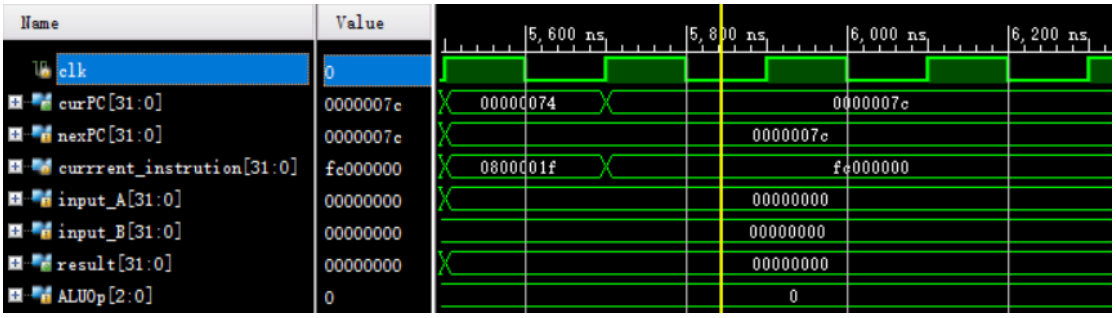
指令15：j 0x7C



（1）在波形图中，curPC表示当前指令的地址为0x00000074，current_instrution为当前指令的十六进制代码值，经比对，取出的指令确为该指令的代码。

（2）该指令为J型指令，直接跳转到0x7C的地址，查看nexPC，也确相符。

指令16：halt



由于该指令是停机指令，不改变PC的值，随着时钟继续进行，PC仍保持不变。

3、在Basys3板上的实现

（1）在板上运行CPU时，需要通过4个数码管显示器来观察CPU的相应指令的执行情况，由于在该CPU上指令地址、数据地址的范围都在0-255内，寄存器地址范围为0-31，也就是只使用了低8位，每2位数码管便可以显示出上述数据，根据烧板文档的说明，配合两个开关SW_in（SW15，SW14），有以下显示内容：

SW_in=00，当前指令地址PC：下条指令地址PC

SW_in=01，RS寄存器地址：RS寄存器数据

SW_in=10，RT寄存器地址：RT寄存器数据

SW_in=11，ALU结果输出：DB总线输出

而同时Reset信号也接在开关上（SW0），CPU的执行时钟依靠按键（单脉冲）即BTNR单步执行，以便于查看不同信息。

（2）在板上显示数码管的内容时，需要对时钟信号进行分频，分频的目的是计数，采用扫描式显示，利用数码管的余晖效应和人眼的视觉暂留效应以达到多个数码管“同时”被点亮的效果。而结合上学期数电的知识，此处的分频具体代码如下：

```
parameter num = 49999;
integer i = 0;

always @(posedge clk)
begin
    if (i >= num)
    begin
        i <= 0;
        clk_div <= ~clk_div;
    end
    else
        i <= i + 1;
end
```

（3）由于相应数码管被点亮时，位控信号AN0-AN3分别为0111、1011、1101、1110，所以使用上述分频后的时钟计数4次循环以点亮数码管。具体代码如下：

```
initial
    count <= 2'b00;
always @(posedge clk)
begin
    if (count == 2'b11)
        count <= 2'b00;
    else
        count <= count + 1;
end
```

（4）由上述计数器生成的数对应生成数码管位控信号和根据SW_in的闭合情况生成CPU执行时需要显示的信息段，然后再将信息段按照每个数码管（8位）划分进对应的数码

管：

```
always @(*)
begin
    case (SW_in)
        2'b00: total_data = {curPC[7:0], nexPC[7:0]};
        2'b01: total_data = {2'b00, rs[4:0], rs_data[7:0]};
        2'b10: total_data = {2'b00, rt[4:0], rt_data[7:0]};
        2'b11: total_data = {ALU_result[7:0], DB_data[7:0]};
        default: total_data = 16'h0000;
    endcase
end

always @(*)
begin
    case (count)
        2'b00: begin
            AN_count = 4'b1110;
            display_data = total_data[3:0];
        end
        2'b01: begin
            AN_count = 4'b1101;
            display_data = total_data[7:4];
        end
        2'b10: begin
            AN_count = 4'b1011;
            display_data = total_data[11:8];
        end
        2'b11: begin
            AN_count = 4'b0111;
            display_data = total_data[15:12];
        end
        default: begin
            AN_count = 4'bxxxx;
            display_data = 4'bxxxx;
        end
    endcase
end
```

（5）将上述生成的要显示的数据送到数码管显示信号的转换器中，即烧板文档中共阳极的代码段，其低7位即可送往数码管分别连接显示。

（6）消抖处理

由于CPU时钟是由按键（单脉冲）产生的，而人在按下去时会产生中间的波动脉冲，使得被误认为高/低电平，而对于这个消抖处理我并不是很明白，所以我是在百度百科上找

到的“在用基于Verilog语言的时序逻辑电路设计按键消抖电路时，通常认为机械抖动的最大周期是20ms，对每一个时钟脉冲信号对按键状态进行取样，以便进行按键消抖处理。在程序中设置一个计数器，来采集按键的值，若按键的值在20ms内都是低电平或者高电平，则可确定这次是人为按键。”这一段话，并结合其下代码，写下了以下消抖处理代码，而在Reset=0时将CPU时钟置为1是为了确保在第一条指令的执行中必须有时钟的下降沿，从而顺利使CPU正常执行。（注：消抖处理是在完成仿真、烧板后加上的，而加上前并没有出现问题，但为防止在某次在板子上操作时翻车而加上了，加上后也没有出现问题。然而在加上后，在对于仿真时时钟周期的初始化设计时出现了问题，即不知道怎样去仿真按键的CPU时间周期，也有可能是消抖处理的写法不对。所以在烧板时将CPU封装成IP核再加上了该模块，但在仿真时则使用去除消抖处理的工程进行仿真）

```
reg check1, check2, check3;
reg clk_div = 0;//20ms时钟检测抖动

integer i = 0;
always @(posedge clk)
begin
    if (i >= 999999)
    begin
        clk_div = ~clk_div;
        i <= 0;
    end
    else
        i = i + 1;
end
always @(posedge clk_div or negedge Reset)
begin
    if (Reset == 0)
    begin
        check1 <= 1;
        check2 <= 1;
        check3 <= 1;
    end
    else
    begin
        check1 <= i_clk;
        check2 <= check1;
        check3 <= check2;
    end
end
assign o_clk = check1 & check2 & check3;
```

（7）操作方法：先将Reset（SW0）置零初始化PC值为0，再将Reset置为1后，每按下一次BTNR，即执行一条指令，置SW_in为不同的值可查看不同信息。前6条指令在板上运行的贴图如下，顺序为：

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总线输出

指令1 addiu $1,$0,5

指令2 addiu $2,$0,-8

指令3 add $1,$1,$2

指令4 sub $1,$1,$2

指令5 andi $1,$1,9




指令6 and $1,$1,$2




六. 实验心得

首先，对于CPU的设计，在刚开始的第一周确实是摸不着头脑，只知道需要用vivado去实现文档中那张看起来很复杂的数据通路图，而仅做了老师提及的建立控制信号与指令关系的真值表。直至第二周，我重新翻看ppt时记下Verilog语言的各种语法、规则，然后渐渐明白Verilog的变量类型和结构说明语句与持续赋值语句之间的关系，如initial语句的赋值对象必须是reg，这在初始化和仿真时用得比较多的；always语句可分为电平敏感和边沿敏感，赋值对象也必须是reg型，而在边沿敏感的语句中，需要使用非阻塞赋值；assign语句赋值对象则需要是wire类型等等。

其次，CPU设计的分模块思想主要是从数据通路图中、上学期数电实验指导书中和与同学的讨论中了解到的，也就是将CPU的实现划分为一个个小的模块，就像数据通路图中的小器件，这个分模块的思想使得我在这个复杂的CPU实验中找到了突破口，使得我将精力专注于每个模块的编写之中，也就是明白个模块功能、输入和输出，像写高级语言的函数一样去写。然后再使用单元测试的方法，每个部件在写好后都写了一个仿真文件去测试，以减少在顶层连接时的bug。

对于遇到的bug，有些细小之处就使我花费了很多时间，比如，指令存储器中读取txt文件中的二进制代码时，存在读不了的bug，在一顿乱改无果后想起程序设计课上有说过斜杠和反斜杠的区别，然后才发现windows的文件路径不能直接复制到要读取的路径上，需要把反斜杠换成斜杠才能正确读取。

然后，在CPU的顶层文件连接好后准备上板的时候却发现上学期很多关于数码管的知识都遗忘了,只好重新翻看以往的项目和实验指导。其实在这里就是需要自己写时钟分频器、计数器、数据选择器，而相应数码管显示数字和数据的转换采用共阳极的代码段。这次的知识遗忘也让我意识到很多时候都需要温故而知新，在当时学习的时候就需要学好，使得印象深刻，在看回自己笔记的时候就能够很好地回忆起相应知识点。

还有就是对于按键时钟仿真和消抖处理间的协调问题，不能在有消抖处理的CPU上进行仿真，不懂怎去写此处的时钟周期，导致最后交上去的文件中有会有2个工程，即最终加上消抖处理的用来烧板的工程(basys)和没有加消抖处理的用来仿真的工程(CPU)。

最后，从整个单周期CPU实验来看，知识点的掌握主要是Veilog语言的掌握和理论课上对单周期CPU数据通路的知识，而我认为背后的思想是最重要的——分模块，自顶向下地逐渐划分，将大的问题划分为小问题，对于小问题完成后，再做单元测试，这种思想不仅仅是在这个实验中，对于其他问题亦可应用其中。