《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 软件工程 2 班
时　　间 ：	2019 年 12 月 25 日
组　　员 ：	贺恩泽 17364025 陈志远 17338020

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。

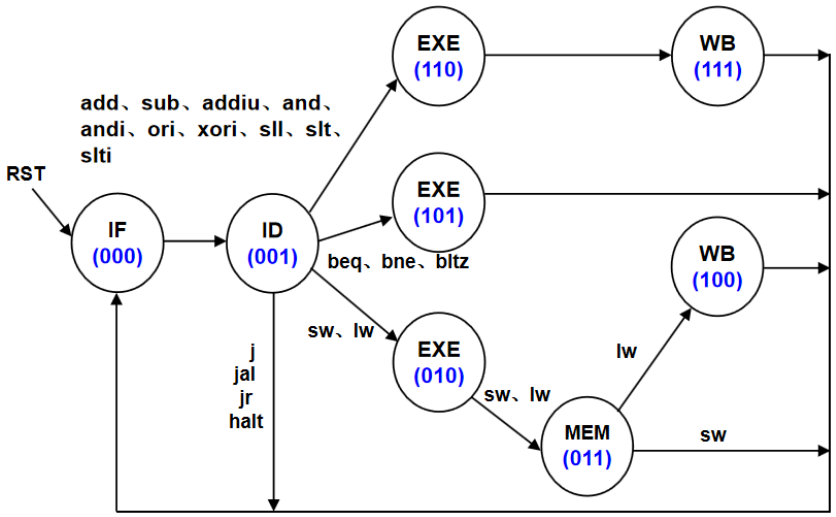


图 2 多周期 CPU 状态转移图

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。

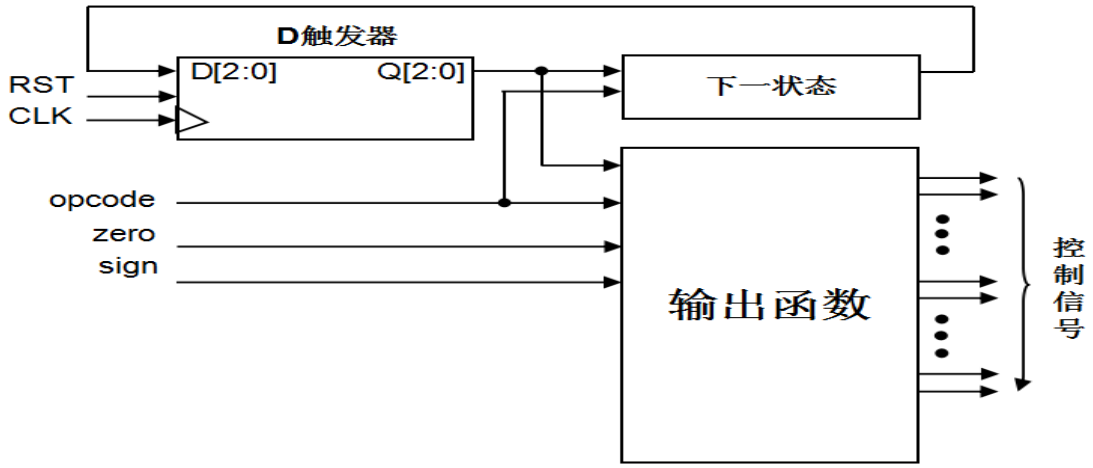


图 3 多周期 CPU 控制部件的原理结构图

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。

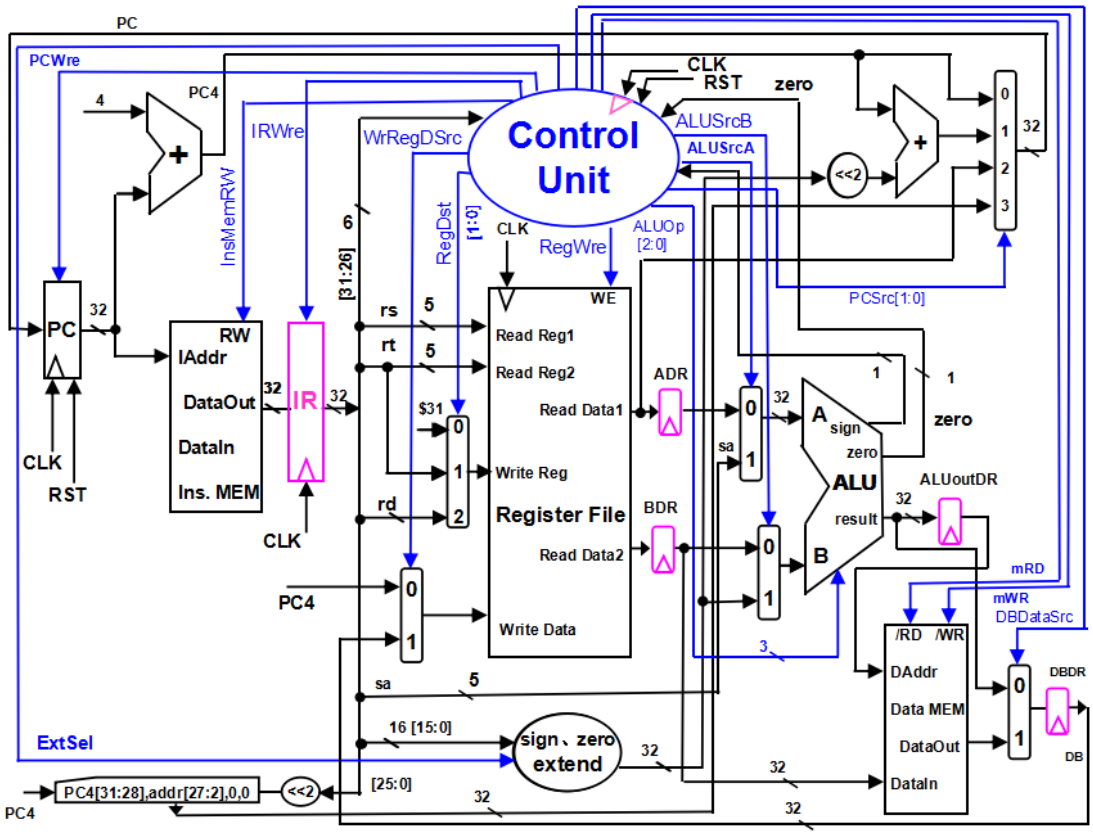


图 4　多周期 CPU 数据通路和控制线路图

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

一. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

二. 实验过程与结果

1、设计思路

多周期CPU的设计主要着重与“多周期”，就是说每个时钟周期只执行指令的一个阶段，所以不同的指令有可能所需要的时钟周期数不一样。而且跳转指令等也会使得数据通路更加复杂。因此多周期CPU的设计需要明确每个状态的转化关系，如上面的图所示，在设计的时候在 InstructionMemory 之外额外增加控制组件；其次还需要处理数据延迟的问题，在外部增加了多个处理 RegisterFile、ALU、DataMemory 和 InstructionMemory 存在的延迟问题。

2、模块设计

(1) 符号定义

为了更加明晰程序代码，并避免因二进制代码书写错误导致的问题，对状态码、操作码等做出如下定义：

a) ALU操作码

```
`define ALU_OP_ADD 3'b000
`define ALU_OP_SUB 3'b001
`define ALU_OP_SLL 3'b010
```

```
`define ALU_OP_OR 3'b011
`define ALU_OP_AND 3'b100
`define ALU_OP_LT 3'b101
`define ALU_OP_SLT 3'b110
`define ALU_OP_XOR 3'b111
```

b) 指令操作码

```
`define OP_ADD 6'b000000
`define OP_SUB 6'b000001
`define OP_ADDIU 6'b000010
`define OP_AND 6'b010000
`define OP_ANDI 6'b010001
`define OP_ORI 6'b010010
`define OP_XORI 6'b010011
`define OP_SLL 6'b011000
`define OP_SLTI 6'b100110
`define OP_SLT 6'b100111
`define OP_SW 6'b110000
`define OP_LW 6'b110001
`define OP_BEQ 6'b110100
`define OP_BNE 6'b110101
`define OP_BLTZ 6'b110110
`define OP_J 6'b111000
`define OP_JR 6'b111001
`define OP_JAL 6'b111010
`define OP_HALT 6'b111111
```

c) PC 跳转类型

```
`define PC_NEXT 2'b00
`define PC_REL_JUMP 2'b01
`define PC_REG_JUMP 2'b10
`define PC_ABS_JUMP 2'b11
```

d) 状态

```
`define STATE_IF 3'b000
`define STATE_ID 3'b001
`define STATE_EXE_AL 3'b110
`define STATE_EXE_BR 3'b101
`define STATE_EXE_LS 3'b010
`define STATE_MEM 3'b011
`define STATE_WB_AL 3'b111
`define STATE_WB_LD 3'b100
```

(2) 控制单元(ControlUnit)

与单周期 CPU 在输入输出基本一致，但是内容有很大的区别：指令状态转化的实

现以及控制信号的赋值。

a) 状态转移的实现

根据上文中状态转移图所示，由当前状态和指令操作码决定下一状态，具体实现如下：

```
always @(posedge CLK or negedge RST) begin
    if (!RST) State <= `STATE_IF;
    else begin
        case (State)
            `STATE_IF: State <= `STATE_ID;
            `STATE_ID: begin
                case (OpCode)
                    `OP_ADD, `OP_SUB, `OP_ADDIU, `OP_AND, `OP
_ANDI, `OP_ORI,
                    `OP_XORI, `OP_SLL, `OP_SLTI, `OP_SLT: Sta
te <= `STATE_EXE_AL;
                    `OP_BNE, `OP_BEQ, `OP_BLTZ: State <= `STA
TE_EXE_BR;
                    `OP_SW, `OP_LW: State <= `STATE_EXE_LS;
                    `OP_J, `OP_JAL, `OP_JR, `OP_HALT: State <
= `STATE_IF;
                    default: State <= `STATE_EXE_AL;
                endcase
            end
            `STATE_EXE_AL: State <= `STATE_WB_AL;
            `STATE_EXE_BR: State <= `STATE_IF;
            `STATE_EXE_LS: State <= `STATE_MEM;
            `STATE_WB_AL: State <= `STATE_IF;
            `STATE_MEM: begin
                case (OpCode)
                    `OP_SW: State <= `STATE_IF;
                    `OP_LW: State <= `STATE_WB_LD;
                endcase
            end
            `STATE_WB_LD: State <= `STATE_IF;
            default: State <= `STATE_IF;
        endcase
    end
end
```

b) 控制信号的实现

不同控制信号根据不同的操作码得到，因此可以列出对于不同操作码的各控制信号的真值表：

Op	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	WrRegDSrc	InsMemRW
add	0	0	0	0	1	1	1
sub	0	0	0	0	1	1	1
addiu	0	0	1	0	1	1	1
and	0	0	0	0	1	1	1
andi	0	0	1	0	1	1	1
ori	0	0	1	0	1	1	1
xori	0	0	1	0	1	1	1
sll	0	1	0	0	1	1	1
slti	0	0	1	0	1	1	1
slt	0	0	0	0	1	1	1
sw	0	0	1	X	0	X	1
lw	0	0	1	1	1	1	1
beq	0	0	0	X	0	X	1
bne	0	0	0	X	0	X	1
bltz	0	0	0	X	0	X	1
j	0	X	X	X	0	X	1
jr	0	X	X	X	0	X	1
jal	0	X	X	X	1	0	1
halt	1	X	X	X	0	X	1
Op	**mRD**	**mWR**	**IRWre**	**ExtSel**	**PCSrc**	**RegDst**	**ALUOp**
add	X	X	1	X	00	10	000
sub	X	X	1	X	00	10	001
addiu	X	X	1	1	00	01	000
and	X	X	1	X	00	10	100
andi	X	X	1	0	00	01	100
ori	X	X	1	0	00	01	011
xori	X	X	1	0	00	01	111
sll	X	X	1	X	00	10	010
slti	X	X	1	1	00	01	110
slt	X	X	1	X	00	10	110
sw	X	1	1	1	00	XX	000
lw	1	X	1	1	00	01	000
beq	X	X	1	1	00(Zero=0) 01(Zero=1)	XX	001
bne	X	X	1	1	00(Zero=1) 01(Zero=0)	XX	001
bltz	X	X	1	1	00(Sign=0) 01(Sign=1)	XX	001
j	X	X	1	X	11	XX	XXX
jr	X	X	1	X	10	XX	XXX
jal	X	X	1	X	11	00	XXX
halt	X	X	1	X	XX	XX	XXX

与单周期 CPU 不同的地方在于，控制信号不仅仅取决于操作码，还取决于当前的状态。各控制信号实现如下：

i. ALUSrcA：EXE 阶段 LS、SLL

```
ALUSrcA = ((State == `STATE_EXE_AL || State == `STATE_EXE
_BR || State == `STATE_EXE_LS) && OpCode == `OP_SLL) ? 1
: 0;
```

ii. ALUSrcB：EXE 阶段 ADDIU、ANDI、ORI、XORI、SLTI、LW、SW

```
ALUSrcB = ((State == `STATE_EXE_AL || State == `STATE_EXE
_BR || State == `STATE_EXE_LS) &&
            (OpCode == `OP_ADDIU || OpCode == `OP_ANDI ||

            OpCode == `OP_ORI || OpCode == `OP_XORI ||
            OpCode == `OP_SLTI || OpCode == `OP_LW ||
            OpCode == `OP_SW)) ? 1 : 0;
```

iii. RegWre：ID 阶段 JAL，或 WB 阶段 LD

```
RegWre = ((State == `STATE_ID && OpCode == `OP_JAL) || (S
tate == `STATE_WB_AL || State == `STATE_WB_LD)) ? 1 : 0;
```

iv. WrRegDSrc：ID 阶段 JAL

```
WrRegDSrc = (State == `STATE_ID && OpCode == `OP_JAL) ? 0
: 1;
```

v. mRD：MEM 或 WB 阶段 LW

```
mRD = ((State == `STATE_MEM || State == `STATE_WB_LD) &&
OpCode == `OP_LW) ? 1 : 0;
```

vi. mWR：MEM 阶段 SW

```
mWR = (State == `STATE_MEM && OpCode == `OP_SW) ? 1 : 0;
```

vii. IRWre：IF 阶段

```
IRWre = (State == `STATE_IF) ? 1 : 0;
```

viii. ExtSel：EXE 阶段 ANDI、ORI、XORI

```
ExtSel = ((State == `STATE_EXE_AL || State == `STATE_EXE_
BR || State == `STATE_EXE_LS) && (OpCode == `OP_ANDI || O
pCode == `OP_ORI || OpCode == `OP_XORI)) ? 0 : 1;
```

ix. PCSrc：IF 或 ID 阶段 JR 为 PC_REG_JUMP，IF 或 ID 阶段 J、JAL 为 PC_ABS_JUMP，EXE 阶段 BEQ、BNE、BLTZ 为 PC_REL_JUMP，否则均为 PC_NEXT

```
if ((State == `STATE_IF || State == `STATE_ID) && OpCode
== `OP_JR) PCSrc = `PC_REG_JUMP;
else if ((State == `STATE_IF || State == `STATE_ID) && (O
pCode == `OP_J || OpCode == `OP_JAL)) PCSrc = `PC_ABS_JUM
P;
else if ((State == `STATE_EXE_AL || State == `STATE_EXE_B
R || State == `STATE_EXE_LS) &&
        (OpCode == `OP_BEQ && Zero) || (OpCode == `OP_BNE
 && !Zero) || (OpCode == `OP_BLTZ && Sign)) PCSrc = `PC_R
EL_JUMP;
else PCSrc = `PC_NEXT;
```

x. RegDst：ID 阶段 JAL 为 b00，WB 阶段 ADDIU、ANDI、ORI、XORI、SLTI、LW 为 b01，否则均为 b10

```
if (State == `STATE_ID && OpCode == `OP_JAL) RegDst = 2'b
00;
else if ((State == `STATE_WB_AL || State == `STATE_WB_LD)
 && (OpCode == `OP_ADDIU || OpCode == `OP_ANDI || OpCode
== `OP_ORI || OpCode == `OP_XORI || OpCode == `OP_SLTI ||
 OpCode == `OP_LW)) RegDst = 2'b01;
else RegDst = 2'b10;
```

xi. ALUOp：根据真值表即可得出

```
case (OpCode)
    `OP_ADD, `OP_ADDIU, `OP_SW, `OP_LW: ALUOp = `ALU_OP_A
DD;
    `OP_SUB, `OP_BEQ, `OP_BNE, `OP_BLTZ: ALUOp = `ALU_OP_
SUB;
    `OP_SLL: ALUOp = `ALU_OP_SLL;
    `OP_ORI: ALUOp = `ALU_OP_OR;
    `OP_AND, `OP_ANDI: ALUOp = `ALU_OP_AND;
    `OP_SLTI, `OP_SLT: ALUOp = `ALU_OP_SLT;
    `OP_XORI: ALUOp = `ALU_OP_XOR;
Endcase
```

xii. PCWre：ID 阶段 J、JAL、JR，或 EXE 阶段 BEQ、BNE、BLTZ，或 MEM 阶段 SW，或 WB 阶段。另外，为保证在每条指令最初阶段的

时钟上升沿 PC 发生改变，需要在上一条指令的最后一个下降沿将 PCWre 设置为 1，这样才能保证 PC 在每条指令最开始的时钟上升沿改变。

```
always @(negedge CLK) begin
    case (State)
        `STATE_ID: begin
            if (OpCode == `OP_J || OpCode == `OP_JAL || O
pCode == `OP_JR) PCWre <= 1;
        end
        `STATE_EXE_AL, `STATE_EXE_BR, `STATE_EXE_LS: begi
n
            if (OpCode == `OP_BEQ || OpCode == `OP_BNE ||
 OpCode == `OP_BLTZ) PCWre <= 1;
        end
        `STATE_MEM: begin
            if (OpCode == `OP_SW) PCWre <= 1;
        end
        `STATE_WB_AL, `STATE_WB_LD: PCWre <= 1;
        default: PCWre <= 0;
    endcase
end
```

(3) 算术运算单元(ALU)

接受寄存器的数据和控制信号，计算后将结果、零和符号标志输出，根据 ALU 功能表即可写出实现，与单周期 CPU 无区别。

(4) 数据存储单元(DataMemory)

数据存储单元负责存取数据，且由时钟下降沿出发写操作。实现为1字节8位的大端方式存储，与单周期 CPU 无区别。

(5) 指令存储器(InstructionMemory)

把指令集以二进制的形式写成一个文件，然后在指令存储器中读进来，以读文件的方式把指令存储到内存中，实现指令的读取。

汇编程序程序转化为二进制代码，如下：

地址	汇编程序	指令代码					
		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码	
0x00000000	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	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 00010 0000 00	=	40822800
0x00000014	**sll $5,$5,2**	011000	00000	00101	00101 00010 0000 00	=	60052880
0x00000018	**beq $5,$1,-2(=,转 14)**	110100	00101	00001	1111 1111 1111 1110	=	D0A1FFFE
0x0000001C	**jal 0x0000050**	111010	00000	00000	0000 0000 0001 0100	=	E8000014
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	01011	00010	1111 1111 1111 1110	=	D562FFFE
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 0001 0111	=	E0000017
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

根据上表的二进制代码，可以得到二进制指令代码文件，其中二进制代码八位一组，然后将文件并放置于 MEMORY_FILE_PATH 处，便可以读文件的方式直接将该文件内容读入。除二进制指令不同之外，与单周期 CPU 实现无区别。

(6) 寄存器组(RegisterFile)

寄存器组接受 InstructionMemory 的输入，输出对应寄存器的数据，从而实现读取寄存器里的数据的功能，在实现上与单周期 CPU 无区别。

(7) 符号扩展单元(SignZeroExtend)

该组件有两个功能：符号扩展和零扩展，输入的扩展方法和待扩展的数据，输出扩展后的数据，在实现上与单周期 CPU 无区别。

(8) 程序计数器(PC)

与单周期 CPU 的 PC 单元大多相同，但是还需增加根据寄存器组输出的跳转地址进行跳转的分支 PC_REG_JUMP，该处代码如下：

```
module NextPCHelper(
    input RST,
    input [1:0] PCSrc,
    input [31:0] PC,
    input [31:0] Immediate,
    input [31:0] RegPC,
    input [31:0] JumpPC,
    output reg [31:0] NextPC);
```

```
    always @(RST or PCSrc or PC or Immediate or RegPC or JumpPC)
begin
        if (!RST) NextPC = PC + 4;
        else begin
            case (PCSrc)
                `PC_NEXT: NextPC = PC + 4;
                `PC_REL_JUMP: NextPC = PC + 4 + (Immediate << 2);
                `PC_REG_JUMP: NextPC = RegPC;
                `PC_ABS_JUMP: NextPC = JumpPC;
                default: NextPC = PC + 4;
            endcase
        end
    end
endmodule
```

(9) Display模块

用于在 Basys3 上的数位管显示数据，需增加状态阶段的展示。

```
case(Mode)
    3'b000: displayContent <= {CurrentPC, NextPC};
    3'b001: displayContent <= {RsAddr, RsData};
    3'b010: displayContent <= {RtAddr, RtData};
    3'b011: displayContent <= {ALUResult, DBData};
    3'b100: displayContent <= {8'h00, State};
endcase
```

(10) 选择器(Selector)

数据选择，用于数据存储单元之后的选择，除单周期所需的二选一数据选择器，还需增加三选一数据选择器。

```
module Selector1In3#(
    parameter WIDTH = 5
)(
    input [1:0] Sel,
    input [WIDTH-1:0] A,
    input [WIDTH-1:0] B,
    input [WIDTH-1:0] C,
    output reg [WIDTH-1:0] Y);

    always @(Sel or A or B or C) begin
        case (Sel)
            2'b00: Y <= A;
            2'b01: Y <= B;
            2'b10: Y <= C;
            default: Y <= 0;
```

```
            endcase
        end
    endmodule
```

(11) 指令寄存器(IR)

用时钟信号 CLK 驱动，采用边缘触发写入指令二进制码。代码如下：

```
module IR(
    input CLK,
    input IRWre,
    input [31:0] DataIn,
    output reg [31:0] DataOut
    );
    always @(posedge CLK) begin
        if (IRWre) begin
            DataOut <= DataIn;
        end
    end
endmodule
```

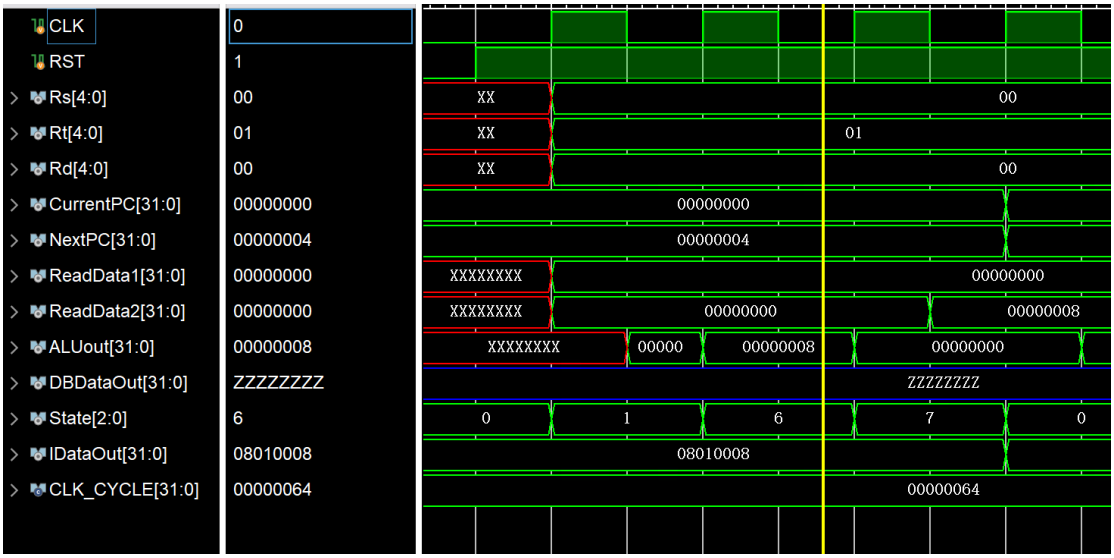
(12) 数据延迟处理（ADR、BDR、ALUoutDR、DBDR）

这部分模块用于切割数据通路，代码相同。

```
module XDR(
    input CLK,
    input [31:0] DataIn,
    output reg [31:0] DataOut
    );
    always @(negedge CLK) DataOut <= DataIn;
endmodule
```

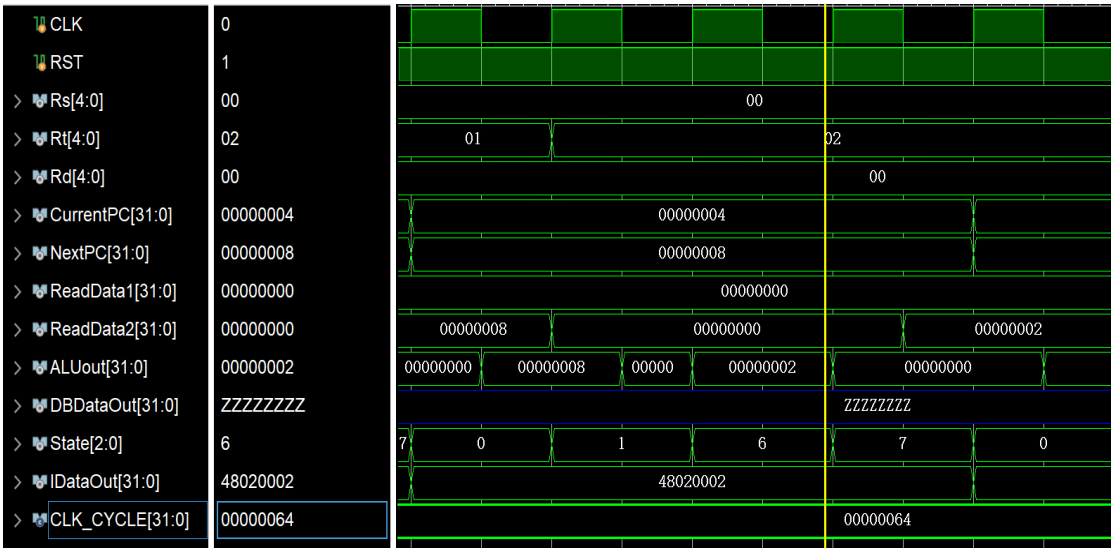
3、仿真测试

(1) addiu $1,$0,8



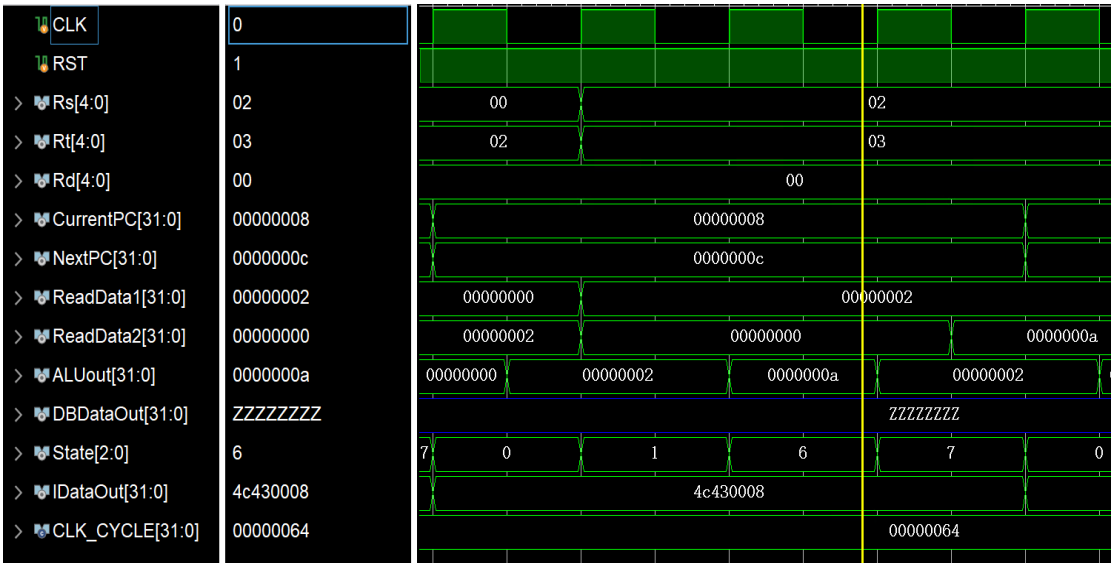
寄存器rs为$0，rt为$1，ALU运算结果为8，$1 = $0 + 8 = 8，结果正确

(2) ori $2,$0,2



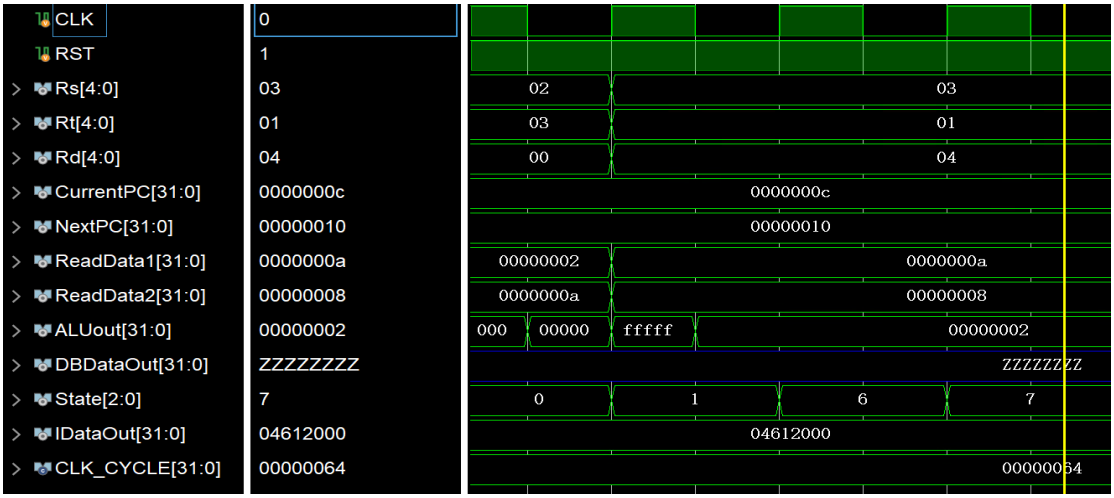
寄存器rs为$0，rt为$2，ALU运算结果为2，$2 = $0 | 2 = 2，结果正确

(3) xori $3,$2,8



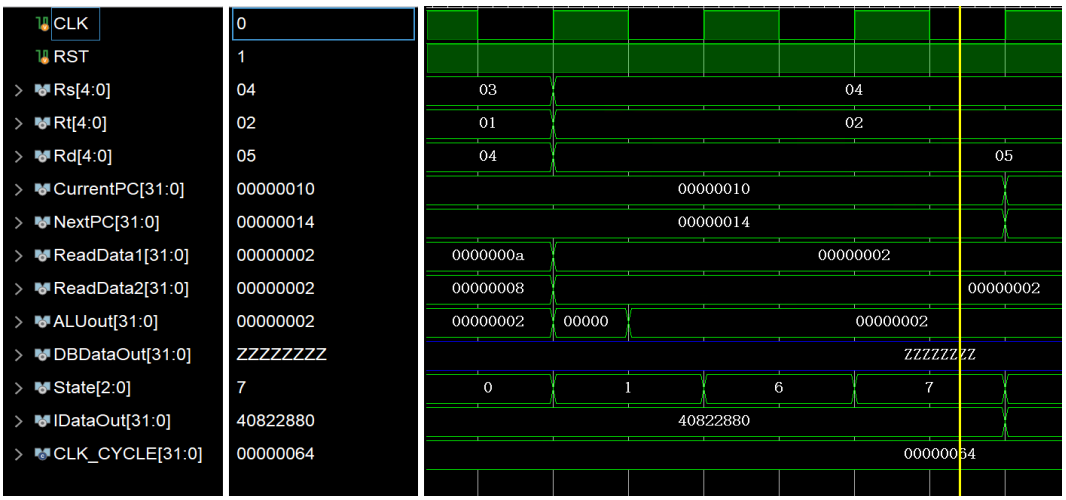
寄存器rs为$2，rt为$3，ALU运算结果为10，$3 = $2 XOR 8 = 10，结果正确

(4) sub $4,$3,$1



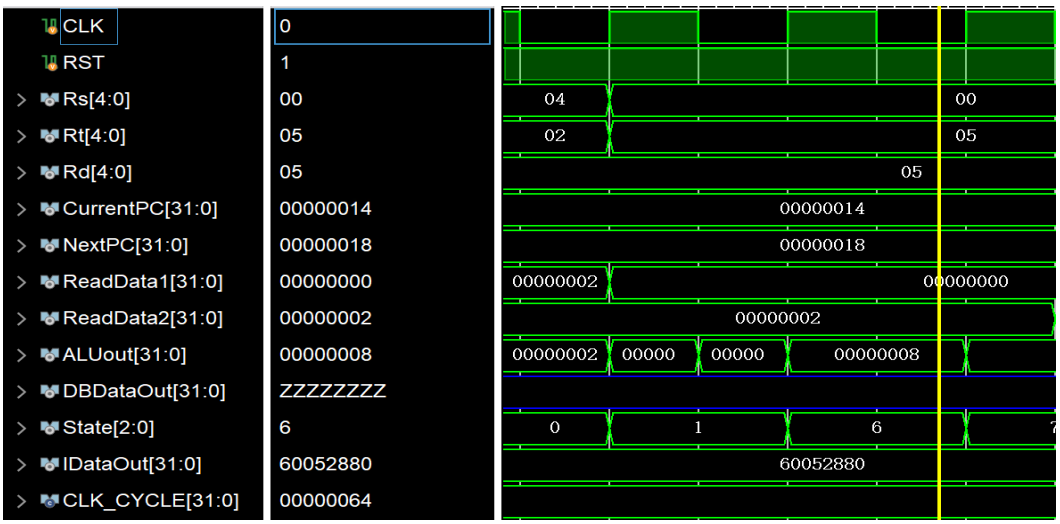
寄存器rs为$3，rt为$1，rd为$4，ALU运算结果为2，$4 = $3 - $1= 2，结果正确

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



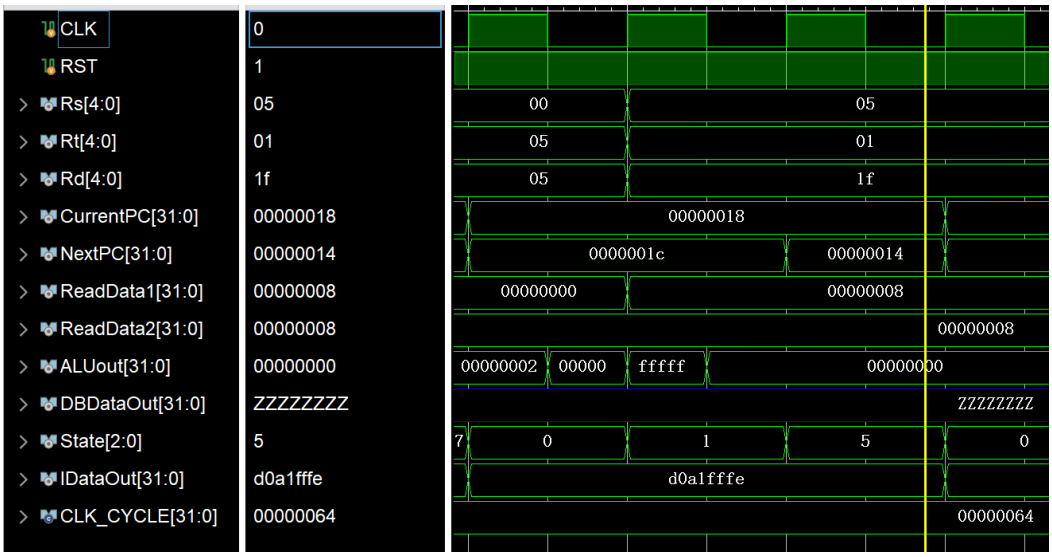
寄存器rs为$4，rt为$2，rd为$5，ALU运算结果为2，$5=$4 & $2=2，结果正确

(6) sll $5,$5,2



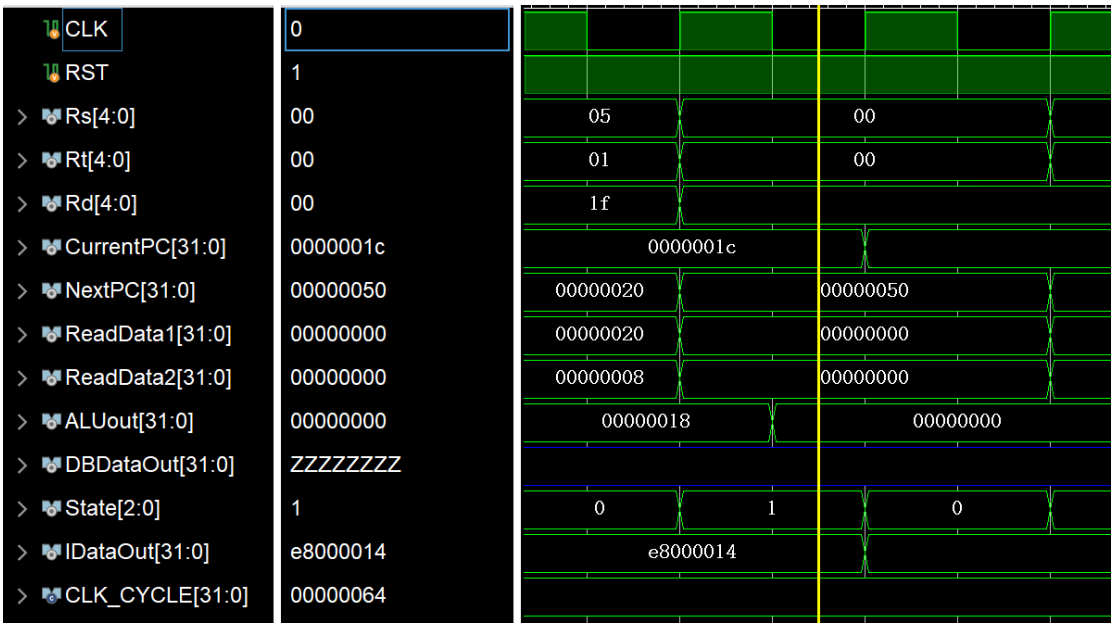
寄存器rt为$5，rd为$5，ALU运算结果为8，$5 = $5 << 2 = 8，结果正确

(7) beq $5,$1,-2



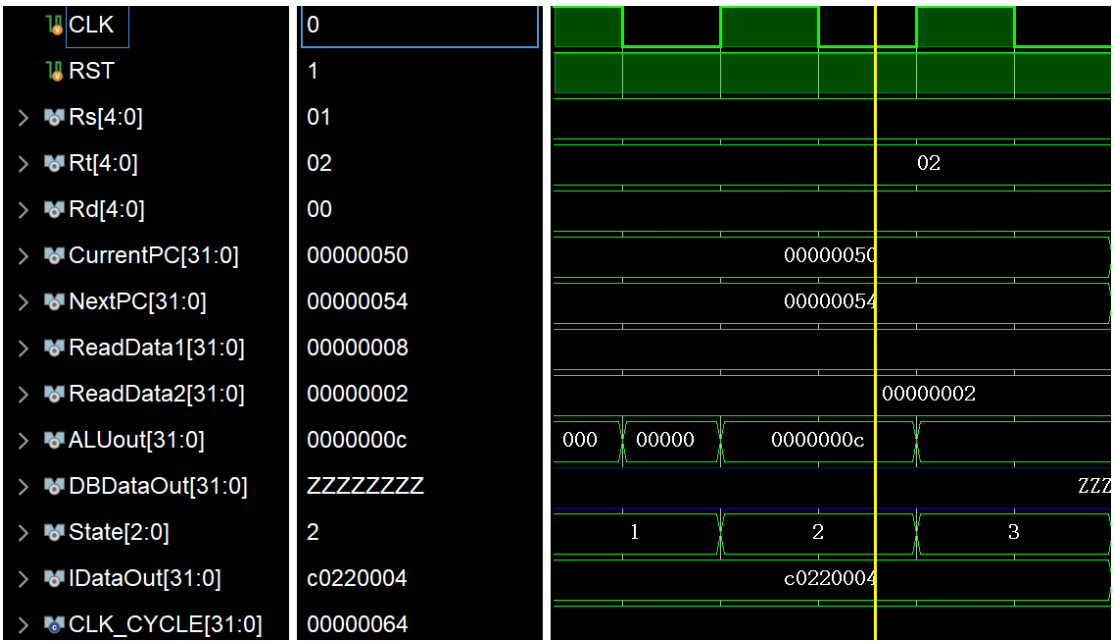
寄存器rs为$5，rt为$1，ALU运算结果为0，即$5-$1=0，当前PC为0x00000018，NextPC变为0x00000014，结果正确

(8) jal 0x0000050



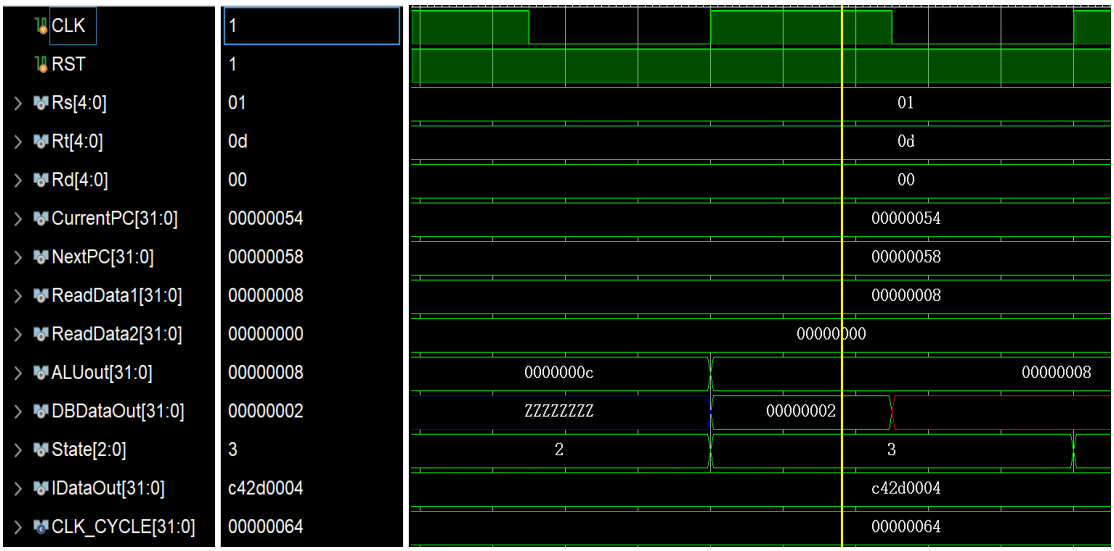
子程序调用，CurrentPC为0x0000001C，将CurrentPC+4写入寄存器rd $31（$1f），NextPC变为0x00000050，结果正确

(9) sw $2,4($1)



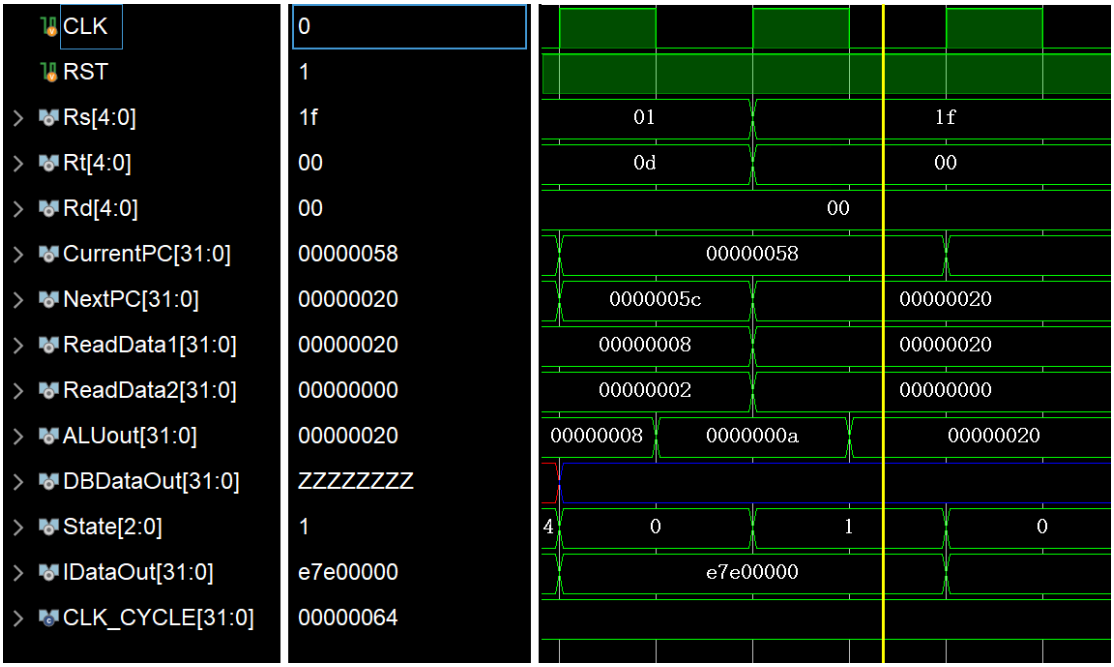
寄存器rs为$1，rt为$2，ALU运算结果为12（0000000c），写入数据存储器memory[12] = $2 = 2，正确性通过 (10) 的 lw 进行验证。

(10) lw $12,4($1)



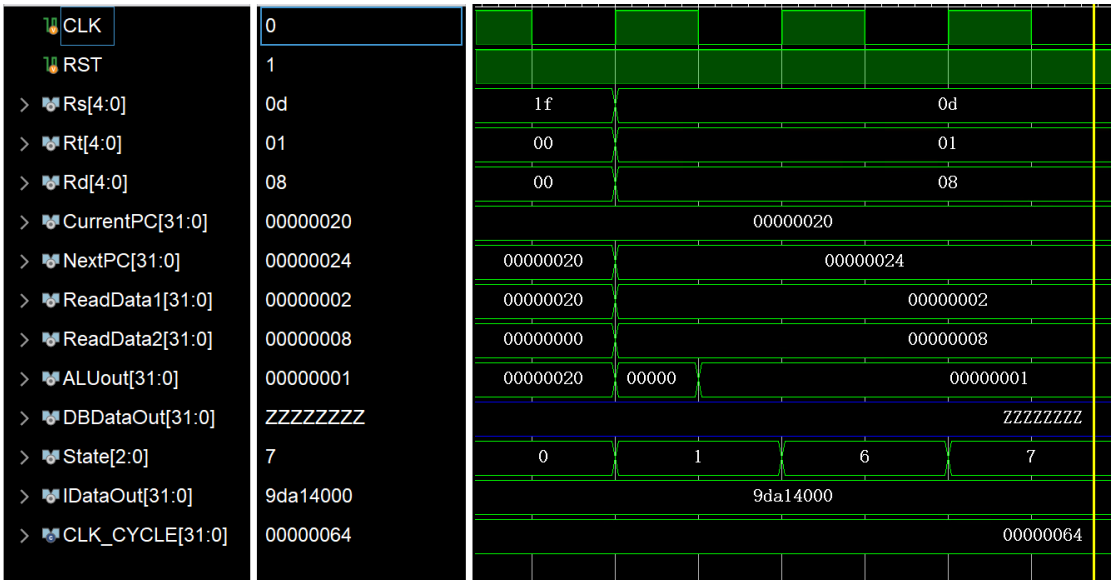
寄存器rs为$1，rt为$13（$0d），ALU运算结果为12（0000000c），从数据存储器 memory[12] 读入数据到 $13，数据为2，该数据由步骤 (9) 写入到数据存储器 memory[12]，因此 (9)、(10) 两步执行均正确

(11) jr $31



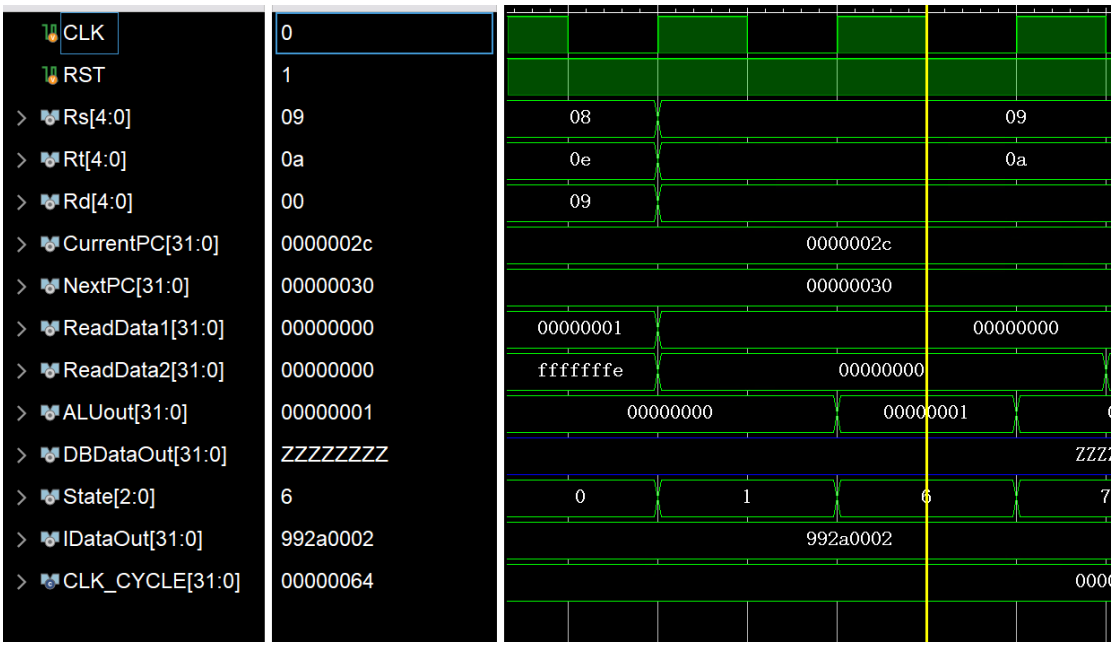
寄存器rs为$31，读取出数据 ReadData1 即0x00000020，CurrentPC为0x00000058，NextPC变为0x00000020，结果正确

(12) slt $8,$13,$1



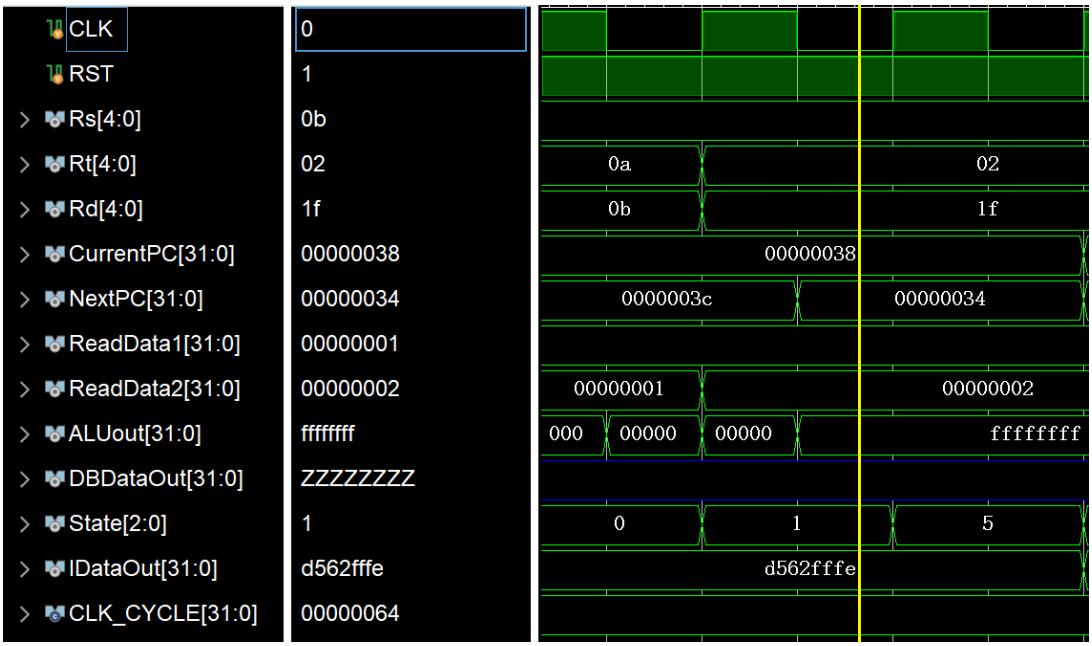
寄存器rs为$13（$0d），rt为$1，rd为$8，ALU运算结果为1，$8 = $13 < $1?1:0 = 1，结果正确

(13) slti $10,$9,2



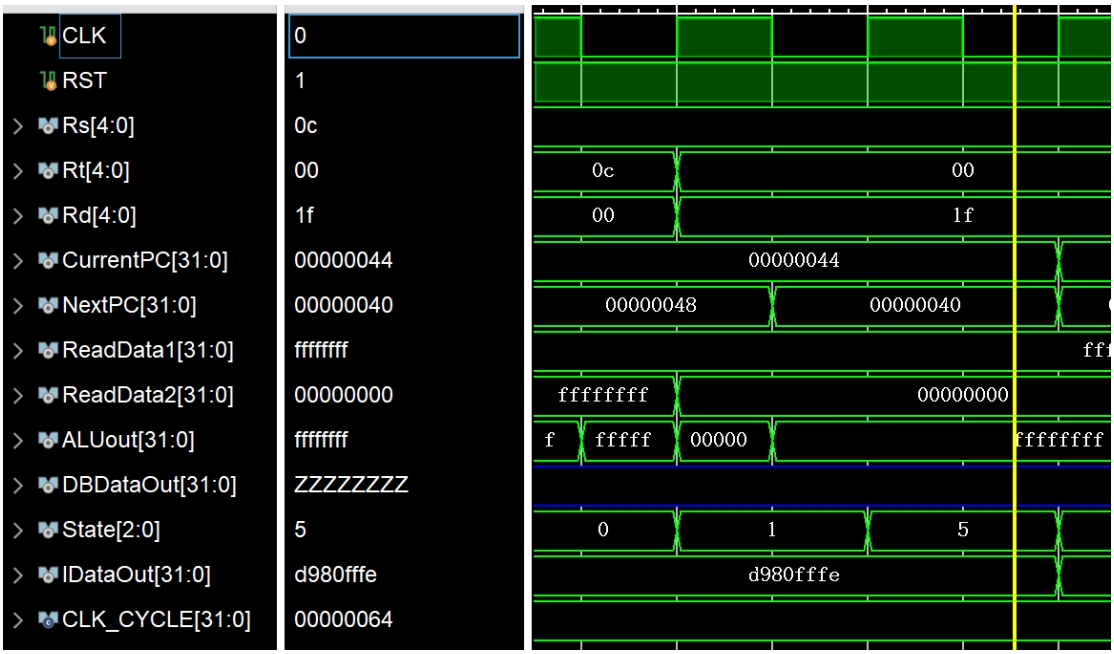
寄存器rs为$9，rt为$10（$0a），ALU运算结果为1，$10=$9 < 2?1:0=1，结果正确

(14) bne $11,$2,-2



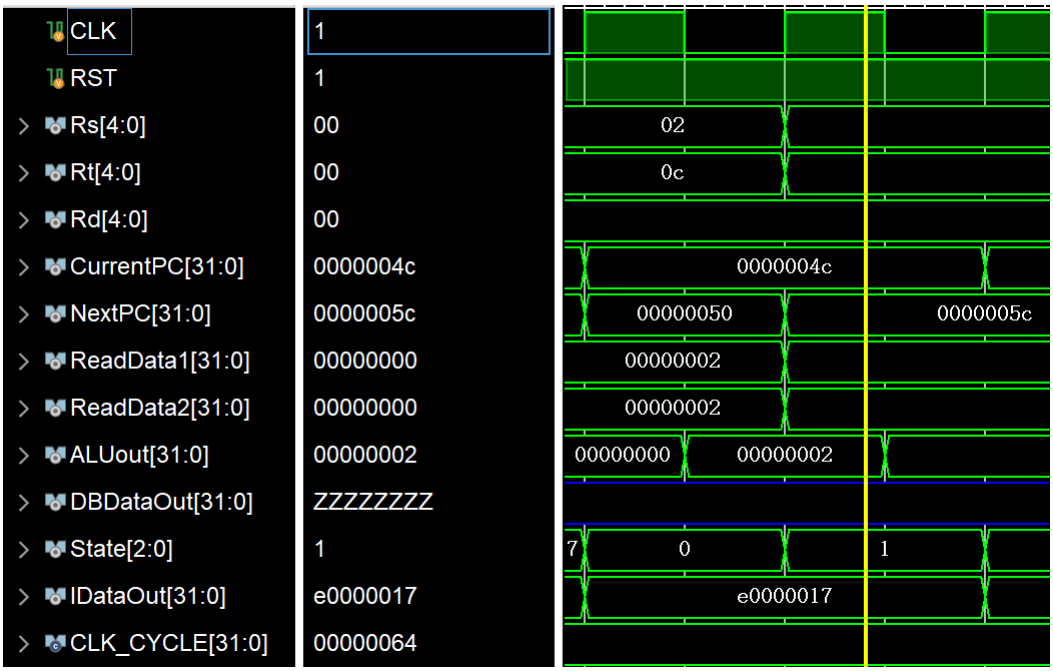
寄存器rs为$11（$0b），rt为$2，ALU运算结果为-1（ffffffff），$11=$2 != -2？1：0，CurrentPC为0x00000038，NextPC变为0x00000034，结果正确

(15) bltz $12,-2



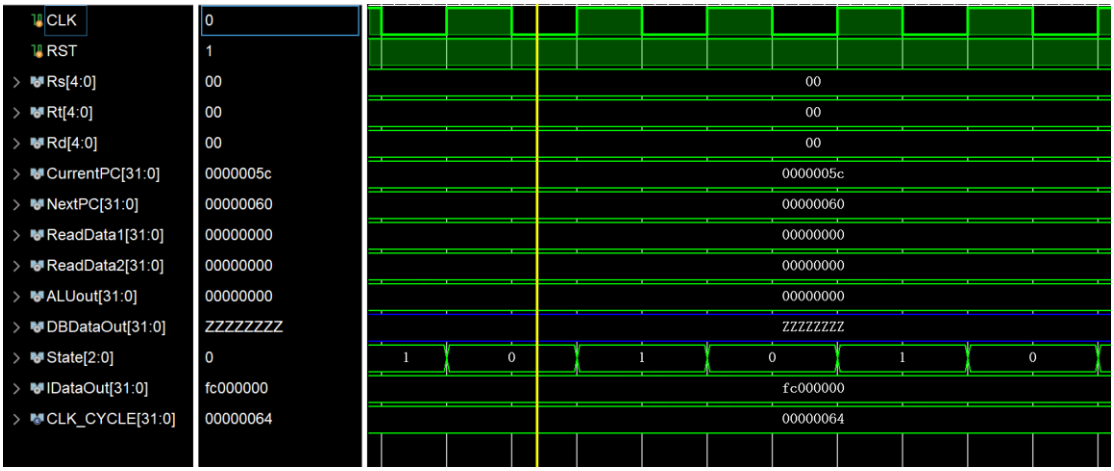
寄存器rs为$12($1c)，ALU运算结果为-1(ffffffff)，CurrentPC为0x00000044，由于$12<$0，NextPC变为0x00000040，结果正确

(16) j 0x000005C



CurrentPC为0x0000004C，NextPC变为0x0000005C，执行正确

(17) halt



停机，PC不再跳转，执行正确

4、烧板运行

(1) addiu $1,$0,8

State，CurrentPC，NextPC

State，Rs，RsData

State，Rt，RtData

State，ALUout，DBDataOut

(2) beq $5,$1,-2

State（IF 阶段），CurrentPC，NextPC

State（EXE 阶段），CurrentPC，NextPC

State（EXE 阶段），Rs，RsData

State（EXE 阶段），Rt，RtData

State（EXE 阶段），ALUout，DBDataOut

(3) jal 0x0000050

State（IF 阶段），CurrentPC，NextPC

State（ID 阶段），CurrentPC，NextPC

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

State（IF 阶段），CurrentPC，NextPC

State（ID 阶段），Rs，RsData

State（ID 阶段），Rt，RtData

State（EXE 阶段），ALUout，DBDataOut

(5) lw $13, 4($1)

State（IF 阶段），CurrentPC，NextPC

State（ID 阶段），Rs，RsData

State（ID 阶段），Rt，RtData

State（EXE 阶段），ALUout，DBDataOut

State（MEM 阶段），ALUout，DBDataOut

(6) jr $31

State（IF 阶段），CurrentPC，NextPC

State（ID 阶段），CurrentPC，NextPC

State（ID 阶段），Rs，RsData

(7) j 0x000005C

State（IF 阶段），CurrentPC，NextPC

State（ID 阶段），CurrentPC，NextPC

(8) halt

State（IF 阶段），CurrentPC，NextPC

State（ID 阶段），CurrentPC，NextPC

综上，CPU 执行正确

三. 实验心得

1、 在完成了之前单周期CPU的实验的基础上，本来任务多周期CPU只是比单周期CPU的设计增加了状态控制模块，但在真正写的时候发现其实需要更多的其他模块的支持，以此支持状态转移等的实现。

2、 JAL指令的实现理解一开始存在问题，JAL的跳转地址没意识到需要补全成32位地址，需要在高位补充PC+4的最高四位，末位补两个0，因此导致一开始JAL命令跳转出错。

3、 因为有了单周期CPU的设计经验，分模块进行设计，分工合作，CPU 各模块的实现都变了清晰了很多。

4、 整个多周期CPU的设计最难的是ControlUnit的设计上，一开始对状态的变化等无从下手，然后通过慢慢查找资料，明白了通过状态信号进行控制，并且对操作码设计不同的选择case，实现状态转移。

5、 在第一次仿真的时候发现波形不太正确，但是组件的代码找不到问题，接着调试了很久才发现原来是二进制代码有些许错误。因此在构建指令代码表时应当谨慎并核查是否正确。

6、 重新编写了 Display 和 Key 模块，但是忘记为按键添加防抖，导致一开始在板上调试时，一次按键跑了多条指令，后来重新添加了按键防抖机制便解决了此问题。

7、 在 Basys3 板上进行调试时，发现 NextPC、ALU 结果不正确，后来发现应该在对应的状态阶段读取对应的值，因此为数位管添加了状态的显示，在正确的状态读取不同组件的数值。

8、 经过这次多周期 CPU 试验后，我们对 CPU 的原理有了更加详细的理解，了解到了单周期 CPU 和多周期 CPU 的区别，更加熟悉 Vivado 开发套件的使用和 Verilog 语言的运用，并动手解决了很多在真正实现时遇到的问题，这些都是只上理论课程得不到的东西，进一步提升了我们的思考能力。