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）：	16 计算机类 3 班
学 生 姓 名 ：	王锡淮
学　　号 ：	16337236
时　　间 ：	2017　年　12　月　10　日

成 绩 ：

实验三 ： 多周期CPU设计与实现

一. 实验目的

(1) 认识和掌握多周期数据通路原理及其设计方法；
(2) 掌握多周期 CPU 的实现方法，代码实现方法；
(3) 编写一个编译器，将 MIPS 汇编程序编译为二进制机器码；
(4) 掌握多周期 CPU 的测试方法;
(5) 掌握多周期 CPU 的实现方法。

二. 实验内容

实验的具体内容与要求。

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）addi rt, rs, **immediate**

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

功能：rt<－rs + (sign-extend)**immediate**

==>逻辑运算指令

（4）or rd, rs, rt

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

功能：rd<－rs | rt

（5）and rd, rs, rt

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

功能：rd<－rs & rt

（6）ori rt, rs, **immediate**

010010	rs(5 位)	rt(5 位)	**immediate**

功能：rt<－rs | (zero-extend)**immediate**

==>移位指令

（7）sll rd, rt,sa

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

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

==>比较指令

（8） slt rd, rs, rt 带符号数

100110	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：if (rs<rt) rd =1 else rd=0, 具体请看表 2 ALU 运算功能表，带符号

（9） slti rt, rs,immediate 带符号

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

功能：if (rs <(sign-extend)**immediate**) rt =1 else rt=0， 具体请看表 2 ALU 运算功能表，带符号

==>存储器读写指令

（10）sw rt, **immediate**(rs)

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

功能：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。

（11）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

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

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

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

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

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

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

（14）bgtz 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

（15）j addr

111000	addr[27:2]

功能：pc <－{(pc+4)[31:28],addr[27:2],0,0}，跳转。

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

（16）jr rs

111001	rs(5 位)	未用	未用	reserved

功能：pc <－ rs，跳转。

==>调用子程序指令

（17）jal addr

111010	addr[27..2]

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

（18）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。

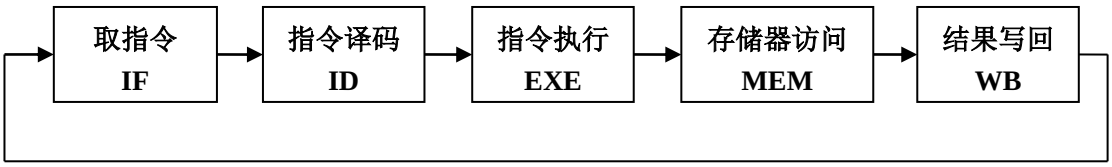


图 1　多周期 CPU 指令处理过程

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。

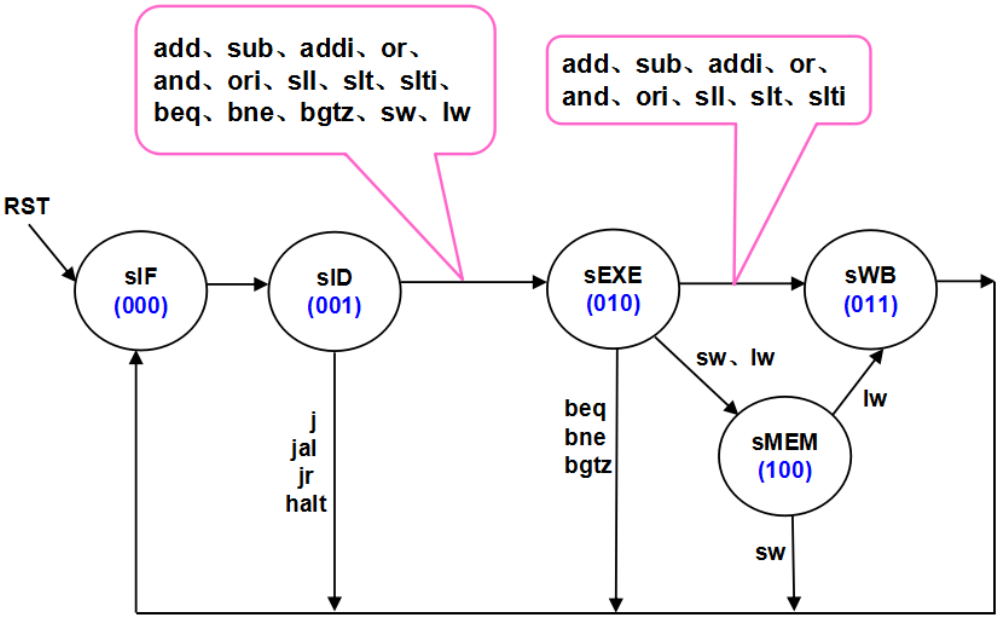


图 2 多周期 CPU 状态转移图

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。

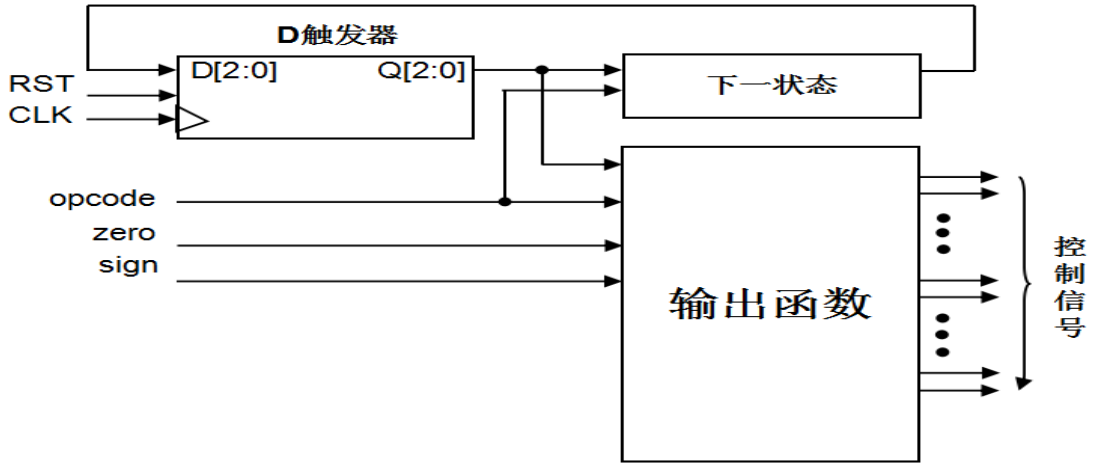


图 3 多周期 CPU 控制部件的原理结构图

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。

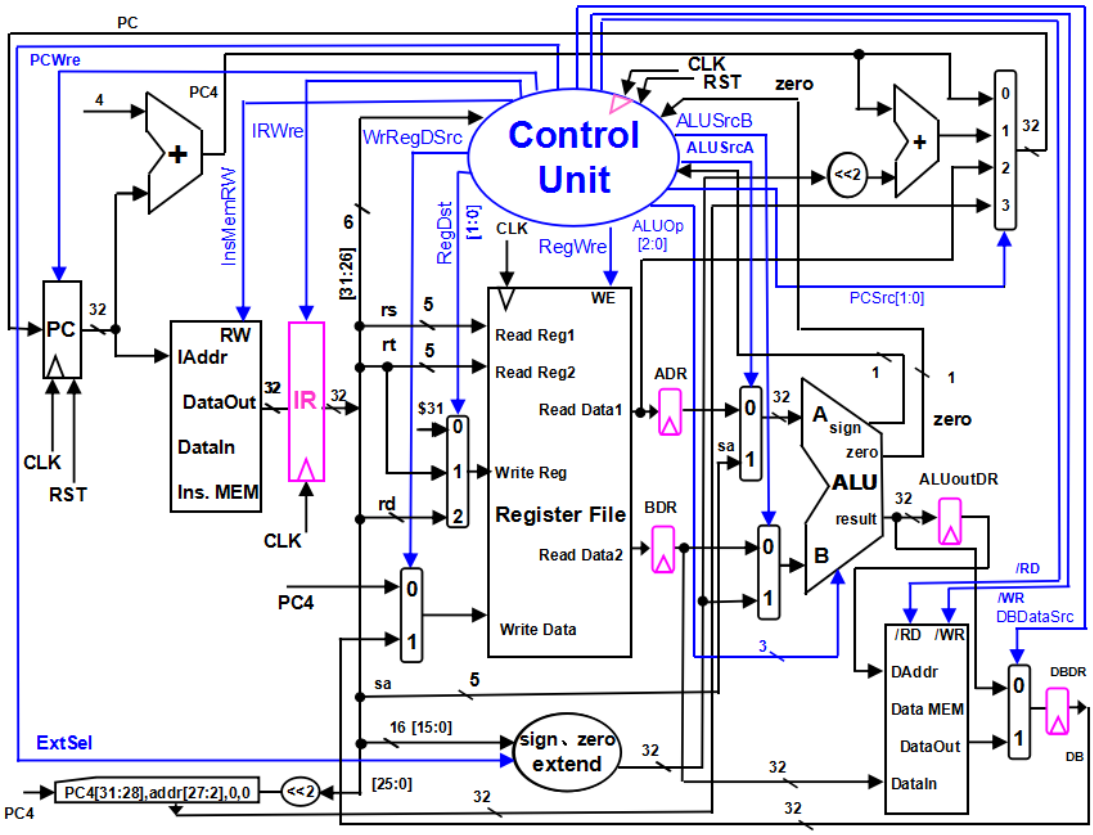


图 4　多周期 CPU 数据通路和控制线路图

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 o 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，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、addi、or、and、ori、beq、bne、bgtz、slt、slti、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、beq、bne、bgtz、slt、sll	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、slti、lw、sw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、sub、addi、or、and、ori、slt、slti、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、bgtz、j、sw、jr、halt	寄存器组寄存器写使能，相关指令：add、sub、addi、or、and、ori、slt、slti、sll、lw、jal
WrRegDSrc	写入寄存器组寄存器的数据来自 pc+4(pc4)，相关指令：jal，写$31	写入寄存器组寄存器的数据来自 ALU 运算结果或存储器读出的数据，相关指令：add、addi、sub、or、and、ori、slt、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
/RD	读数据存储器，相关指令：lw	存储器输出高阻态
/WR	写数据存储器，相关指令：sw	无操作
IRWre	IR(指令寄存器)不更改	IR 寄存器写使能。向指令存储器发出读指令代码后，这个信号也接着发出，在时钟上升沿，IR 接收从指令存储器送来的指令代码。与每条指令都相关。
ExtSel	(zero-extend)**immediate**，相关指	(sign-extend)**immediate**，相关指令：

	令：ori；	addi、lw、sw、beq、bne、bgtz；
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slt、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bgtz(sign=1，或 zero=1)； 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bne(zero=0)、bgtz(sign=0，且 zero=0)； 10：pc<－rs，相关指令：jr； 11：pc<－{pc[31:28],addr[27:2],0,0}，相关指令：j、jal；	
RegDst[1..0]	写寄存器组寄存器的地址，来自： 00：0x1F($31)，相关指令：jal，用于保存返回地址（$31<-pc+4）； 01：rt 字段，相关指令：addi、ori、slti、lw； 10：rd 字段，相关指令：add、sub、or、and、slt、sll； 11：未用；	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

相关部件及引脚说明：

Instruction Memory：指令存储器

Iaddr，指令地址输入端口

DataIn，存储器数据输入端口

DataOut，存储器数据输出端口

RW，指令存储器读写控制信号，为 0 写，为 1 读

Data Memory：数据存储器

Daddr，数据地址输入端口

DataIn，存储器数据输入端口

DataOut，存储器数据输出端口

/RD，数据存储器读控制信号，为 0 读

/WR，数据存储器写控制信号，为 0 写

Register File：寄存器组

Read Reg1，rs 寄存器地址输入端口

Read Reg2，rt 寄存器地址输入端口

Write Reg，将数据写入的寄存器，其地址输入端口（rt、rd）

Write Data，写入寄存器的数据输入端口

Read Data1，rs 寄存器数据输出端口

Read Data2，rt 寄存器数据输出端口

WE，写使能信号，为 1 时，在时钟边沿触发写入

IR：　指令寄存器，用于存放正在执行的指令代码

ALU： 算术**逻辑单元**

result，ALU 运算结果

zero，运算结果标志，结果为 0，则 zero=1；否则 zero=0

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

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	Y = A + B	加
001	Y = A - B	减
010	Y= (A<B) ?1: 0	比较 A 与 B 不带符号

011	if (A<B && (A[31] == B[31])) Y = 1;
	else if (A[31] ==1 && B[31] == 0) Y = 1;
	else Y = 0;
		比较 A 与 B
		带符号

100	Y = B<<A	B 左移 A 位
101	Y = A ∨ B	或
110	Y = A ∧ B	与
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

a. 概述

多周期CPU在处理指令时需要经过5个阶段，每个阶段都占一个时钟周期：IF，ID，EXE，MEM，WB，如图1所示。但是每条指令所需经过的指令阶段即时钟周期数不尽相同，本实验中需要实现的18条指令所经过的指令阶段如表3所示。

表3 指令对应指令阶段表

指令	IF	ID	EXE	MEM	WB
Add	√	√	√		√
Addi	√	√	√		√
Sub	√	√	√		√
Or	√	√	√		√
Ori	√	√	√		√
And	√	√	√		√
Sll	√	√	√		√
Slt	√	√	√		√
Slti	√	√	√		√
Beq	√	√	√		
Bne	√	√	√		
Bgtz	√	√	√		
Lw	√	√	√	√	√
Sw	√	√	√	√	
J	√	√			
Jal	√	√			
Jr	√	√			
Halt	√	√			

除此之外，由于各种指令所经历的指令阶段不尽相同，各种指令在不同阶段要进行的任务也不尽相同。每个阶段对于不同指令所应执行的操作如表4所示。

表4 每个阶段对应动作

阶段	R类型运算指令	分支指令	LW	SW	J Jr	Jal	Halt
IF	根据PC读取指令，并在之后将PC加4。						
ID	指令译码，读取寄存器内容，并对某些内容进行扩展。				译码，计算目标地址，对于Jal还要保存当前PC（已经加4）		译码
EXE	根据Op运算		计算内存地址				
MEM			读取数据	写入数据			
WB	将结果写回		将结果写回				

此外，每一个阶段都要受到控制单元的调控。

在多周期CPU中，要重点解决两个大的问题。

第一、控制部件CU在同一条指令的不同周期需要发出不同的控制信号。如上所述，CPU在同一条指令的不同指令阶段需要完成不同的对应任务，故此需要不同的控制信号。而这些控制信号的变化是与历史状态有关的，这种逻辑可以使用有限状态机的方式来表现，如图5所示，在此处我采用了与实验原理部分不同的有限状态机，该状态机的优点在于能够将不同指令在不同阶段的任务分开到不一样的状态中，让每一个状态的任务更加明确，状态的转移也更加明确。各个状态的转移条件会在每个阶段的表述中详细描述。控制部件CU发出的信号与历史状态以及当前输入的关系如图5所示。

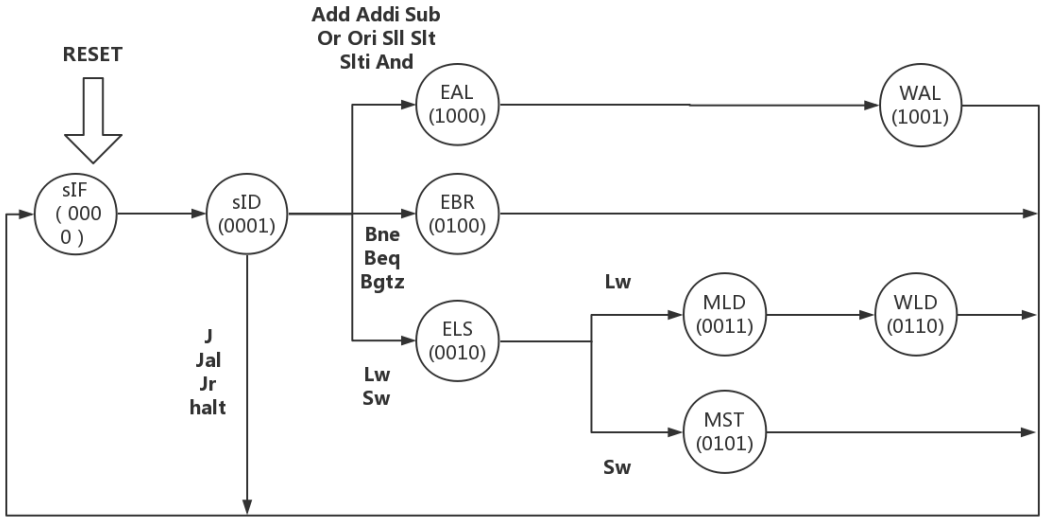


图 5 多周期 CPU 状态转移图

第二、即使是在处理同一条指令的过程中，随着周期的不同，控制部件CU以及其他部件产生的控制信号也不相同，这样在不同周期中发挥作用的部件之间传递数据时，就需要在中间加一级缓存，否则，数据在传输之中有可能被丢失掉。各寄存器的叙述也会在下文各阶段的描述中被详细表达。

表5 控制信号与历史状态以及当前输入的关系

当前状态	指令\控制信号	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	WrRegDSrc	InsMemRW	RD	WR	IRWre	ExtSel	PCSrc	RegDst	ALUOp
sIF	X	0	X	X	X	0	X	1	X	1	1	X	X	X	X
sID	J	1	X	X	X	0	X	X	X	1	0	X	11	X	X
	Jal	1	X	X	X	1	0	X	X	1	0	X	11	00	X
	Jr	1	X	X	X	0	X	X	X	1	0	X	10	X	X
	halt	0	X	X	X	0	X	X	X	1	0	X	X	X	X
EAL	Add	0	0	0	0	0	X	X	X	1	0	X	X	X	000
	sub	0	0	0	0	0	X	X	X	1	0	X	X	X	001
	Add	0	0	1	0	0	X	X	X	1	0	1	X	X	0

	i														00
	Or	0	0	0	0	0	X	X	X	1	0	X	X	X	101
	And	0	0	0	0	0	X	X	X	1	0	X	X	X	110
	Ori	0	0	1	0	0	X	X	X	1	0	0	X	X	101
	Sll	0	1	0	0	0	X	X	X	1	0	X	X	X	100
	Slt	0	0	0	0	0	X	X	X	1	0	X	X	X	011
	Slti	0	0	1	0	0	X	X	X	1	0	1	X	X	011
ELS	Sw	0	0	1	X	0	X	X	X	1	0	1	X	X	000
	Lw	0	0	1	X	0	X	X	X	1	0	1	X	X	000
EBR	Beq（zero=1）	1	0	0	X	0	X	X	X	1	0	1	01	X	001
	(zero=0)												00		
	Bne(zero=0)	1	0	0	X	0	X	X	X	1	0	1	01	X	001
	(zero=1)												00		
	Bgtz(zero=0	1	0	0	X	0	X	X	X	1	0	1	01	X	001

	&& sign=0)														
	(zero=1 || sign=1)												00		
MLD	Lw	0	X	X	1	0	X	X	0	1	0	X	X	X	X
MST	sw	1	X	X	X	0	X	X	X	0	0	X	00	X	X
WLD	Lw	1	X	X	X	1	1	X	X	1	0	X	00	01	X
WAL	Addi,ori,slti	1	X	X	X	1	1	X	X	1	0	X	00	01	X
	Add,sub,or,and,slt,sll													10	

b. **各个状态的解释**

sIF状态：

该状态的任务是在时钟边沿到来时更新PC，同时计算出PC+4的值，并将指令取出放置到IR寄存器中，然后IR寄存器在相应的时钟边沿将指令输出。

在IF状态，因为此时控制模块CU还不能知道取到的是哪条指令，即此时CU模块看到的指令是同质的，对于每一条指令，IF状态完成的任务是相同的，给出的控制信号也是相同的，是一个共有的状态。

此外，还应注意到在该状态要监测到的两个时钟边沿不能相同，否则会引起竞争与

冒险现象，**本实验设计的PC模块选择时钟上升沿，IR寄存器选择时钟下降沿。**

模块设计：

PC模块

输入：时钟信号，重置信号，PC写使能信号，下一个PC地址。下一个PC的地址要根据当前PC地址和执行的指令是否会改变PC地址来确定。

输出：下一个要执行的PC地址。

功能说明：负责PC值的改变，当时钟上升沿到来时即改变PC。同时需要重置PC的功能和区分停机指令和其他不阻碍PC变化的指令，即需要两个控制信号Reset和PCWre。

设计：关键部分代码如下：

```
initial
    IAddrOut_reg = 32'b0;

assign IAddrOut = IAddrOut_reg;

always @(posedge CLK) begin
    if(Reset == 1)begin
        if(1 == PCWre) begin
            IAddrOut_reg <= IAddrIn;
        end
        else begin
            IAddrOut_reg <= IAddrOut_reg;
        end
  end
  else IAddrOut_reg <= 32'b0;
end
```

即PC的值由IAddrOut_reg决定，并且同时需要将IAddrOut_reg初始化为零。

而当时钟上升沿到来时，要判断Reset和PCWre信号，当Reset为1时，若PCWre为1时才改变IAddrOut_reg的值，若PCWre为0时不改变IAddrOut_reg的值；当Reset为0时，IAddrOut_reg置为0。

Instructioin_Memory模块

输入：当前PC，指令输入，指令存储器写使能。

输出：指令输出。

功能说明：该模块负责指令的获取，也根据指令存储器写使能读取或写入指令（在本实验中不涉及指令的写入）。

设计：关键代码如下：

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

initial begin
    $readmemb("../../../test.txt",Ins_mem);
  end

always @(IAddr or IDataIn or InsMemRW) begin

  if(1 == InsMemRW) begin
    IDataOut[31:24] <= Ins_mem[IAddr];
    IDataOut[23:16] <= Ins_mem[IAddr+1];
    IDataOut[15:8] <= Ins_mem[IAddr+2];
    IDataOut[7:0] <= Ins_mem[IAddr+3];
  end
end
```

值得注意的是调用系统任务从本地文件获取指令文件来初始化指令的存储部分，此处直接将该文件放在工程目录下，即可用**相对地址**来读取该文件，而不必改变代码的内容。然后该模块的核心便是将指令输出，注意是按照**大端方式**输出。

如何选择下一个PC？

在本次实验中，包含了**四**种PC地址的选择，我在单周期的实验中便已经实现了这四种地址的选择，以下只简要表述：

第一种地址是PC+4的结果，也就是执行除分支、停机、跳转指令之外的选择。

第二种地址是分支指令条件满足后的跳转地址。

第三种地址是jr指令的跳转地址。

第四种地址是j、jal指令的跳转地址。

以上四种地址可以使用一个四路选择器来进行选择。

IR寄存器模块：

该模块根据IRWre信号判断是否在时钟下降沿到来时输出新的指令，其实现简单，此处略去。以及此后的ADR，BDR，ALUoutDR，DBDR寄存器模块，这些模块都是在时钟边沿到来时改变输出，本实验中采用下降沿，原因在之后都分析中会提到，这些寄存器可以用统一的模块实例化来实现，其代码较简单，此处略去。

IF状态随后**无条件地**跳转到ID状态。

sID状态：

该状态要完成的任务是译码，同时进行位扩展等数据处理，并读取寄存器的数据，将数据放入寄存器中，从这一状态起控制单元发送特定控制信号。还有j，jal，jr，halt指令会在这一阶段结束，要做出相应的动作。

由于在这一状态CPU知道了指令的内容，也就在这一阶段控制信号开始分化。下面给出控制模块ControlUnit的设计。

ControlUnit模块

输入：指令操作码opcode，重置指令Reset，ALU信号zero和sign，时钟CLK

输出：14个控制信号

功能说明：从该模块的设计上可以看出该模块包含组合逻辑和时序逻辑两部分，如图3所示，组合逻辑体现在输出函数部分和下一状态的计算部分，输出函数部分根据当前状态和输入的操作码以及ALU信号得到控制信号，下一状态的计算部分体根据当前状态和当前指令决定下一状态，**这一点的实现得益于我的状态的划分使得下一状态的确定十分明显**；时序逻辑体现在状态的更新，此处应注意的是时钟边沿的选择问题，**CU的时钟边沿选择应该能使需要该模块输出的控制信号才能正常工作并且要用到时钟触发的模块在对应的时钟边沿到来前就输出了正确的控制信号，**所以**本实验设计的CU模块选择上升沿。**

设计：

首先需要判断当前指令的操作码，采用以下方法：

```
wire addop, subop, addiop, orop, andop, oriop, sllop,
  sltop, sltiop, swop, lwop, beqop, bneop, bgtzop, jop, jrop,
  jalop, haltop;
assign addop = (opcode == 6'b000000);
assign subop = (opcode == 6'b000001);
assign addiop = (opcode == 6'b000010);
assign orop = (opcode == 6'b010000);
assign andop = (opcode == 6'b010001);
assign oriop = (opcode == 6'b010010);
assign sllop = (opcode == 6'b011000);
assign sltop = (opcode == 6'b100110);
assign sltiop = (opcode == 6'b100111);
assign swop = (opcode == 6'b110000);
assign lwop = (opcode == 6'b110001);
assign beqop = (opcode == 6'b110100);
assign bneop = (opcode == 6'b110101);
assign bgtzop = (opcode == 6'b110110);
assign jop = (opcode == 6'b111000);
assign jrop = (opcode == 6'b111001);
assign jalop = (opcode == 6'b111010);
assign haltop = (opcode == 6'b111111);
```

更新状态的部分是平凡的，此处略去。

接下来是选择状态，只需根据当前状态以及当前操作码选择即可，列举一部分代码如下：

```
always @(state_now or opcode) begin
  case(state_now)
    sIF: state_next = sID;
    sID: begin
      if (jop || jalop || jrop || haltop) begin
        state_next = sIF;
      end
      else if (addop || addiop || orop || oriop || subop ||
        sltop || sllop || sltiop || andop) begin
        state_next = EAL;
      end
      else if (beqop || bneop || bgtzop) begin
        state_next = EBR;
      end
      else begin
        state_next = ELS;
      end
    end
```

最后是控制信号的确定，本模块采用的方法是逐个信号确定，列举部分代码如下：

```
always @(state_now or opcode or zero or sign) begin
//choose each signal respectively
  //PCWre
  if ((state_now == sID &&  (jalop || jop || jrop)) || state_now == EBR ||
    state_now == MST || state_now == WLD ||
    state_now == WAL)
    PCWre = 1;
  else
    PCWre = 0;
  //ALUSrcA
  if (state_now == EAL && sllop)
    ALUSrcA = 1;
  else if ((state_now == EAL && !sllop) || state_now == ELS || state_now == EBR)
    ALUSrcA = 0;
  else
      ALUSrcA = ALUSrcA;
```

为了完成位扩展功能，本次实验中设计了一个Extend模块，该模块与单周期实验中一模一样，且实现较为简单，就不在赘述。

对于读取寄存器数据的任务，需要一个寄存器模块来完成功能。

RegisterFile模块

输入：两个寄存器rs，rt的编号，写寄存器使能WE，时钟信号CLK，写入的寄存器编号Write_reg，写入的数据Write_data。

输出：两个寄存器的内容Read_data1，Read_data2。

功能说明：负责管理寄存器的读取和写入，寄存器的读取是异步的；而寄存器的写入是同步的，发生在时钟的下降沿。

设计：主要代码如下：

```
reg [31:0] regFile[1:31];

assign Read_data1 = (Read_reg1 == 5'b0)? 0:regFile[Read_reg1];
assign Read_data2 = (Read_reg2 == 5'b0)? 0:regFile[Read_reg2];

always @(negedge CLK) begin
  if (1 == WE && Write_reg != 0) begin
    regFile[Write_reg] <= Write_data;
  end
end
```

a. 寄存器堆regFile的声明技巧，由于注意到0号寄存器始终是0，同时不可写，所以没有必要保存零号寄存器的内容，只声明1到31号寄存器。

b. 寄存器内容的读取，也是由于注意到0号寄存器始终是零，当rs或rt为0是，其内容直接输出为零。

c. 同样是因为零号寄存器不可写，当时写使能WE为1，即允许写入时，才能向非零号寄存器写入数据。

为了让j，jal，jr，halt指令在sID状态后能够进入下一条指令，在这一状态中，检测到对应操作码后，**便会输出相应的控制信号，选择正确的下一个PC地址以及选择PC写使能。**特别的，对于jal指令，在这一状态下还要将当前PC（已经经过加4）保存在第31号寄存器（$ra）中，这一动作通过选择正确的写寄存器地址信号以及写寄存器内容信号完成。

除了上述情况，其它指令都会进入到指令执行阶段。EXE阶段又根据指令的不同，划分为三种状态，R类型运算指令对应EAL状态，分支指令对于EBR状态，存储器储存或载入指令对应ELS阶段，以下展开描述：

首先以下的三个状态都需要用到ALU模块，而这一模块与单周期实验中的模块只是略有不同，此处直接放出代码。

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

always @(ALUopcode or rega or regb) begin
  case(ALUopcode)
    3'b000: result = rega + regb;
    3'b001: result = rega - regb;
    3'b010: result = (rega < regb)? 1:0;
        3'b011:begin
        if(rega < regb && (rega[31] == regb[31]))
            result = 1;
        else if(rega[31] == 1 && regb[31] == 0) result =  1;
        else result = 0;
        end
        3'b100: result = regb << rega;
    3'b101: result = rega | regb;
    3'b110: result = rega & regb;

    3'b111: result = (~rega & regb) | (rega & ~regb);
    default: begin
      result = 0;
    end
    endcase
end
```

EAL状态：

在这一状态，任务是输出正确信号，并根据ALUSrcA、ALUSrcB、ALUOp信号进行运算，将ALU运算的结果直接输入到DBDR前面的二路选择器中，并通过选择信号DBDataSrc选择运算结果作为输出输入到DBDR寄存器当中，DBDr寄存器在对应的时钟边沿到来时输出结果。

此状态的后续状态是WAL。

EBR状态：

在这一状态，任务是输出信号，然后判断分支条件是否满足，并据此分析与下一个PC地址相关的信号，决定下一条指令。

下一状态是sIF。

ELS状态：

在这一状态，任务是计算出LW或SW的目标地址，并将该地址输入到ALUoutDR中。

这一状态下一状态是MLD或MST状态，分别对应LW和SW指令。ALUoutDR在相应的时钟边沿输出。

首先在MEM阶段需要一个存储器模块，该模块在本次实验中与单周期实验中的区别

是不需要时钟，其余都一样。代码实现在此省略。

MLD状态：

这一状态的任务是从存储器中读取所需的数据，并通过相应的选择信号输出数据到DBDR中，DBDR寄存器在对应的时钟下降沿输出数据。

下一状态是WLD。

MST状态：

这一状态的任务是将数据写入存储器中。

下一状态是sIF。

WAL状态：

本状态的任务是将运算结果写入到目标寄存器中。

下一状态是sIF。

WLD状态：

本状态的任务是将从存储器中读取的数据写入到目标寄存器中。

下一状态是sIF。

最后在设计各个状态时要注意的还有两点：

（1） **各个缓冲寄存器是在哪一个阶段输出数据。由于多周期CPU中一条指令要经历多个时钟周期，分不清楚各个寄存器在哪个周期输出数据很容易会造成错误。**对应关系如6表所示：

表6 寄存器对于的周期

寄存器	IR	ADR	BDR	ALUoutDR	DBDR
阶段	IF	ID	ID	EXE	EXE/MEM

还要注意的是ALUoutDR的输出只有在MEM阶段，也就是只有LW和SW指令才会用得到，对于其它指令，ALU的运算结果直接写入到DBDR中，这也就是为什么DBDR有时需要在EXE阶段输出的原因。

（2） **还有就是PC选择时钟上升沿触发和CU模块选择上升沿触发引起的竞争与冒险的解决方案。**这样的选择会造成在控制信号到来前PC就要更新完毕的危险，但是PC模块和CU模块选择相同的时钟边沿是唯一的选择，所以只能另

寻它法。**我的解决方案是将PCWre和PCSrc信号的给出时间提前，提前到当前指令的最后一状态，在这一状态到来时便发出信号，而不是等下一个指令的sIF状态到来时再发出指令。**

c. **数据通路的设计**

首先，在这里我要完整地回答上面提到的时钟边沿的选择的问题，出于：

a. 避免电路之间的竞争与冒险。

b. 控制信号要在其它动作发生前给出。

c. 时间顺序在前的周期要给时间顺序在后的周期发生动作留下足够的空间。

三个考虑，在本次实验中使用的时钟边沿如表 7 所示：

表 7 时钟边沿的使用

部件	PC	IR	RegFile	ADR	BDR	ALUoutDR	DBDR	ControlUnit
时钟边沿	↑	↓	↓	↓	↓	↓	↓	↑

而数据通路的详细设计完全体现在上面的数据通路图中，简明易懂。与单周期实验的区别主要在以下几点：

(1) 各个寄存器需要输入输出连线。

(2) 寄存器模块的写入目标寄存器和写入数据添加了数据选择部件。

(3) 对于 ALU 模块的输出，要连接到 ALUoutDR 寄存器和存储器模块后的数据选择器，原因是 R 型运算指令的 ALU 运算结果不需要经过 ALUoutDR 寄存器。

d. **CPU 的测试**

(1) 过程

为了测试 CPU 设计的正确性，编写了一段汇编程序，该端程序使用的全部要求的设计指令。如表 8 所示：

表 8 测试用汇编程序

地址	汇编程序	指令代码					
		op（6）	**rs(5)**	**rt(5)**	**rd(5)/immediate (16)**	**16 进制数代码**	
0x00000000	addi $1,$0,8	**000010**	**00000**	**00001**	**0000 0000 0000 1000**	**=**	08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010		48020002
0x00000008	or $3,**$2**,$1	010000	00010	00001	0001 1000 0000 0000		40411800
0x0000000C	sub $4,**$3**,$1	000001	00011	00001	0010 0000 0000 0000		04612000
0x00000010	and $5,**$4**,$2	010001	00100	00010	0010 1000 0000 0000		44822800
0x00000014	**sll $5,$5,2**	011000	00000	00101	0010 1000 1000 0000		60052880
0x00000018	**beq $5,$1,-2(=,转 14)**	110100	00101	00001	**1111 1111 1111 1110**		d0a1fffe
0x0000001C	**jal 0x0000048**	111010	00000	00000	0000 0000 0001 0010		e8000012
0x00000020	**slt $8,$12,$1**	100110	01100	00001	0100 0000 0000 0000		99814000
0x00000024	**addi $14,$0,-2**	000010	00000	01110	1111 1111 1111 1110		080efffe
0x00000028	**slt $9,$8,$14**	100110	01000	01110	0100 1000 0000 0000		990e4800
0x0000002C	**slti $10,$9,2**	100111	01001	01010	0000 0000 0000 0010		9d2a0002
0x00000030	**slti $11,$10,0**	100111	01010	01011	**0000 0000 0000 0000**		9d4b0000
0x00000034	add $11,**$11**,$8	000000	01011	01000	**0101 1000 0000 0000**		01685800
0x00000038	**bne $11,$2,-2 (≠,转 34)**	110101	01011	00010	**1111 1111 1111 1110**		d562fffe
0x0000003C	addi $2,$2,-1	000010	00010	00010	**1111 1111 1111 1111**		0842ffff
0x00000040	**bgtz $2,-2 (>0,转 3C)**	110110	00010	00000	**1111 1111 1111 1110**		d840fffe
0x00000044	**j 0x0000054**	111000	00000	00000	**0000 0000 0001 0101**		e0000015
0x00000048	sw $2,4($1)	110000	00001	00010	**0000 0000 0000 0100**		c0220004
0x0000004C	lw $12,4($1)	110001	00001	01100	**0000 0000 0000 0100**		c42c0004
0x00000050	**jr $31**	111001	11111	00000	0000 0000 0000 0000		e7e00000
0x00000054	**halt**	111111	00000	00000	0000 0000 0000 0000	**=**	fc000000

(2) 结果展示

以下以波形图的形式展示每一条指令的运行情况，注意下面的波形图的波形的名称，由于每一条指令都有多个周期，对于每一周期的各个信号都展示显得累赘，于是在结果展示部分展示PC地址，状态，使用的寄存器内容以及使用的存储器内容。寄存器编号依次为1，2，3，4，5，8，9，10，11，12，14，存储器为12，13，14，15。

每个指令结束后的结果都应如图6所示。

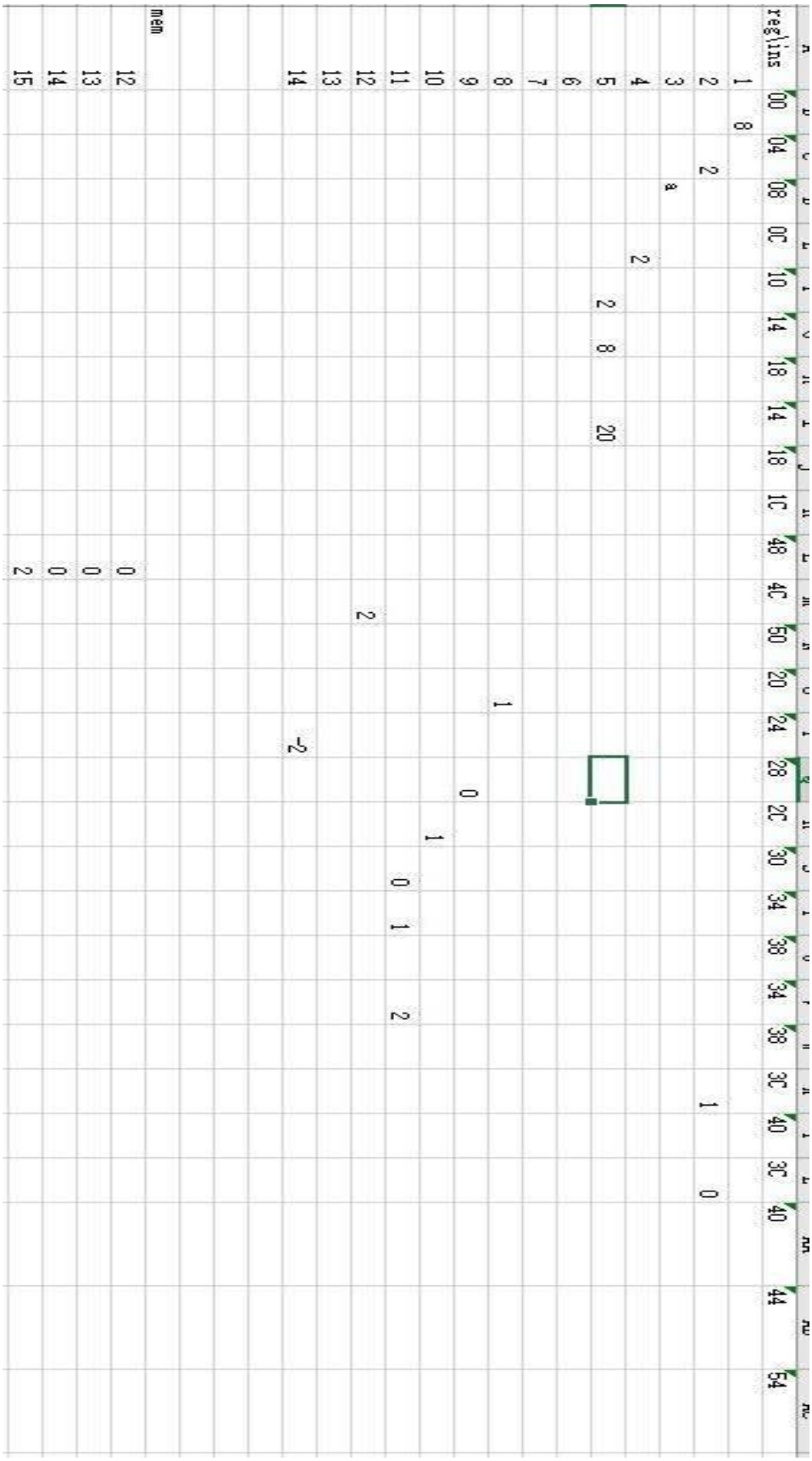
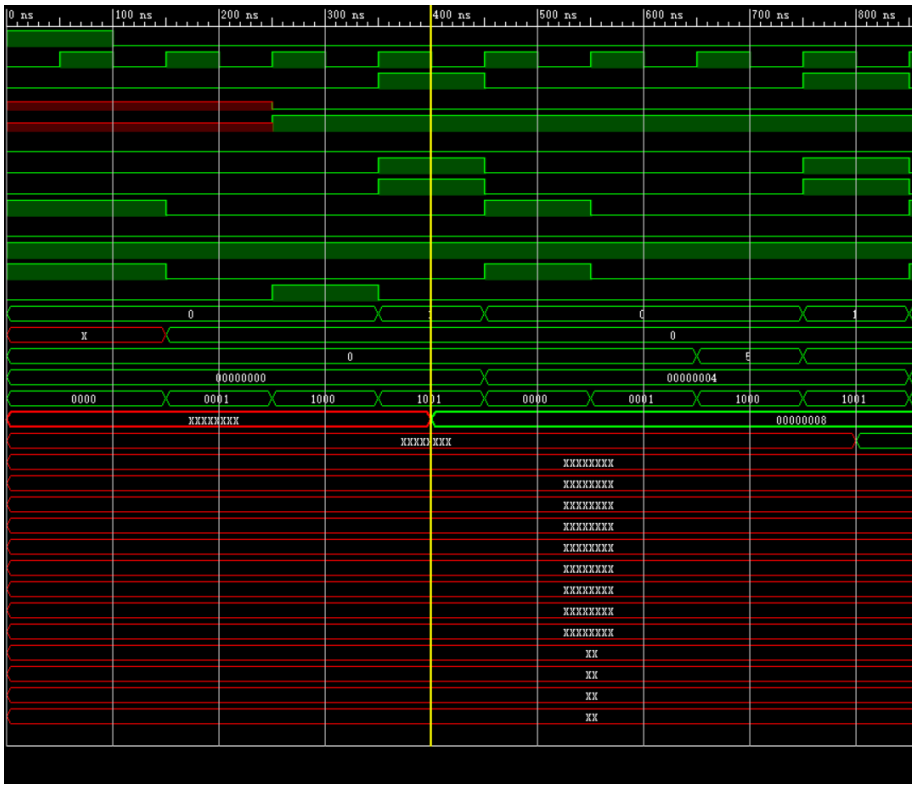


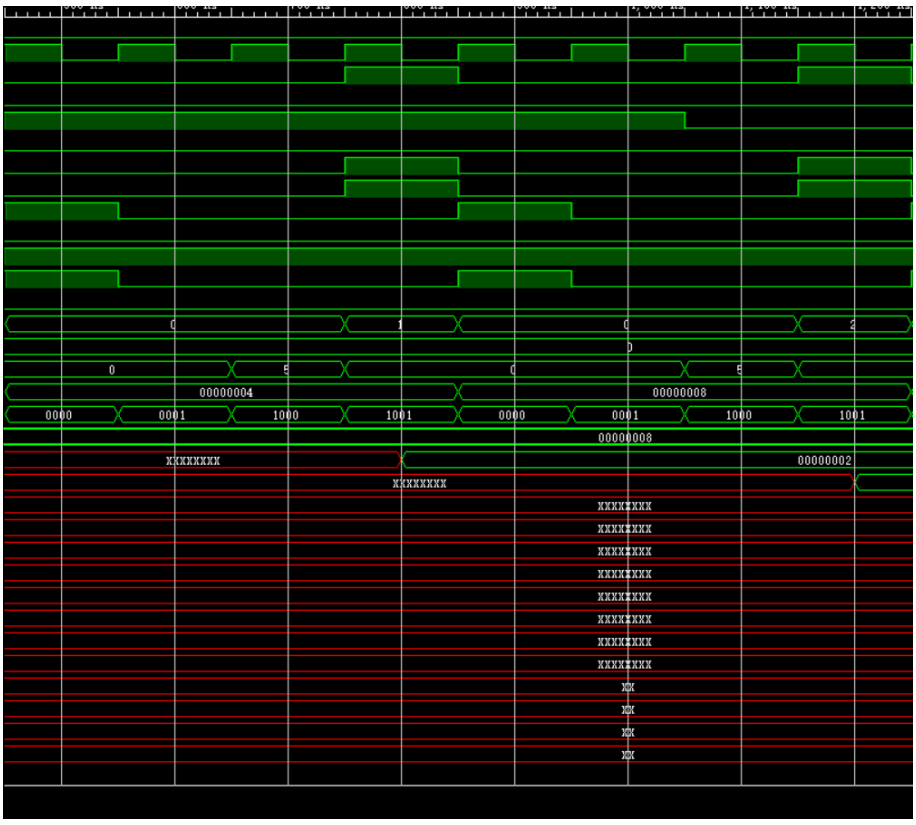
图6 测试代码的理想结果

波形图

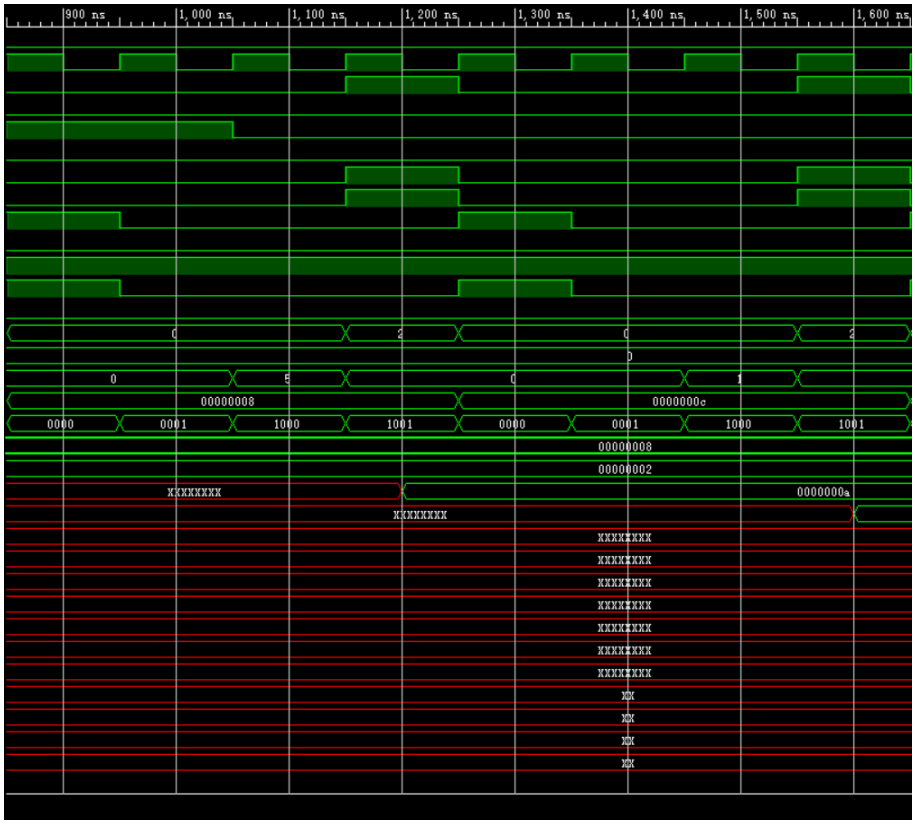
PC=0x4, addi $1,$0,8



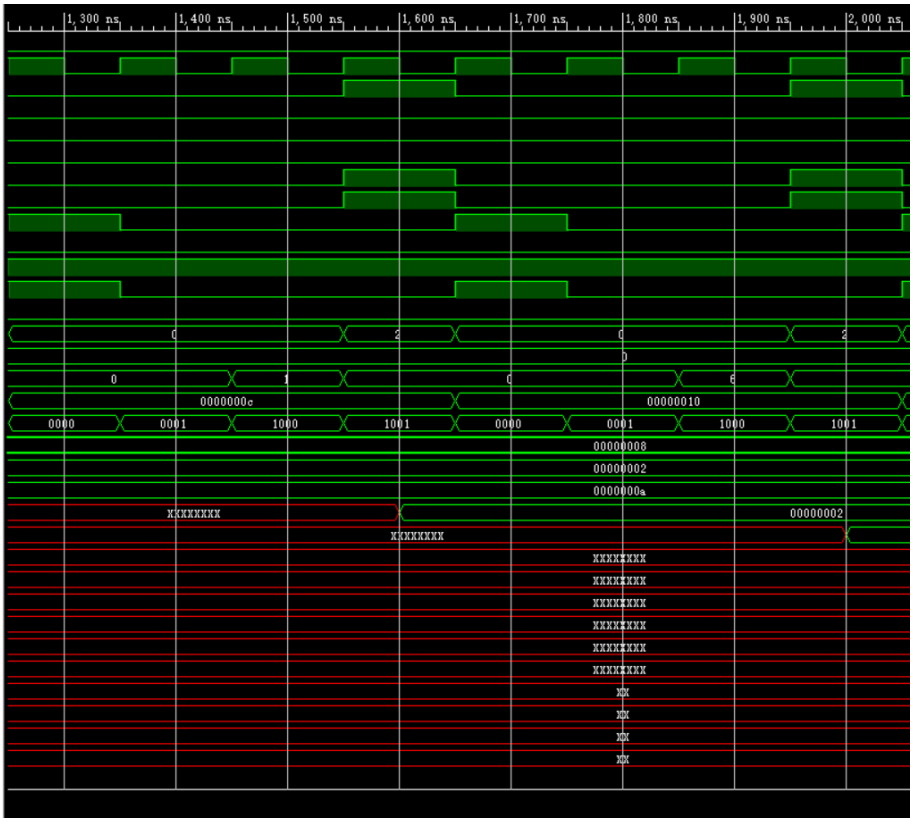
PC=0x04, ori $2,$0,2



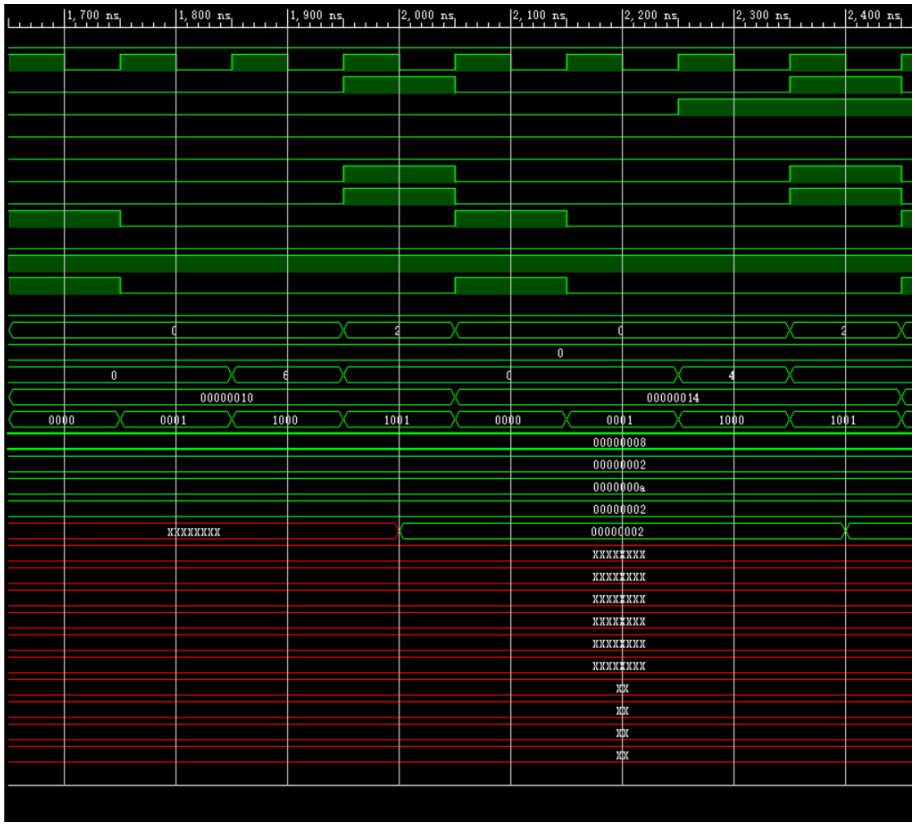
PC=0x08, or $3, $2, $1



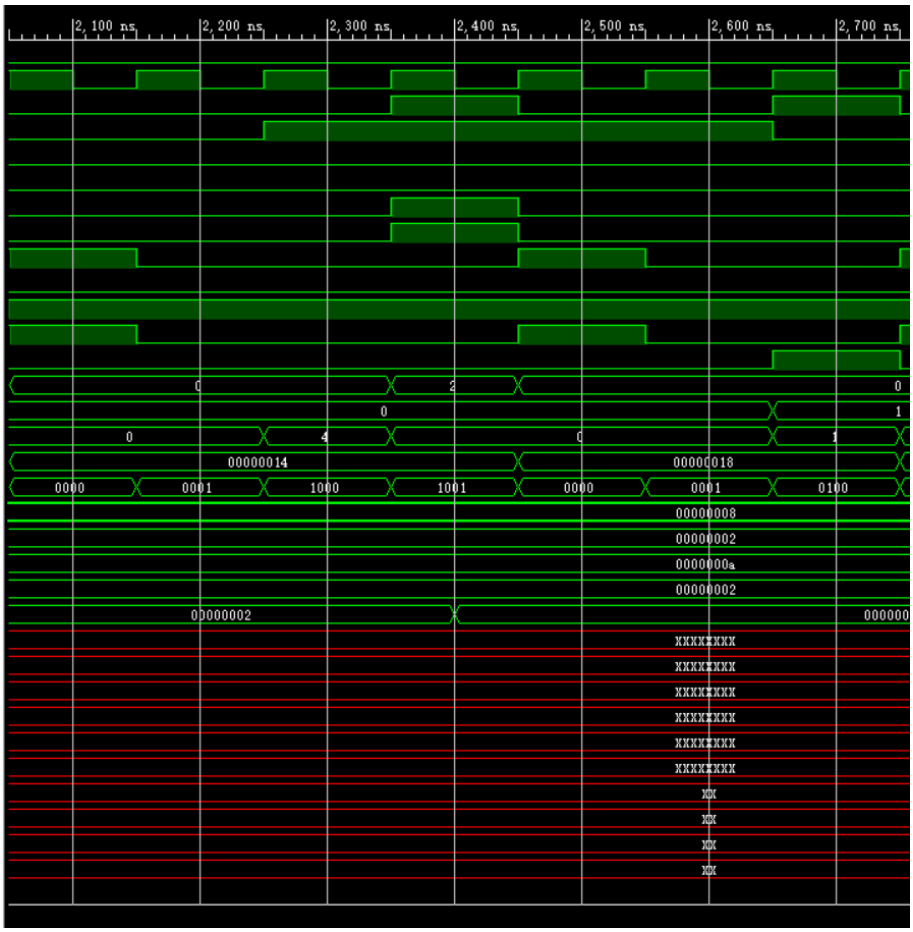
PC=0x0C, sub $4, $3, $1



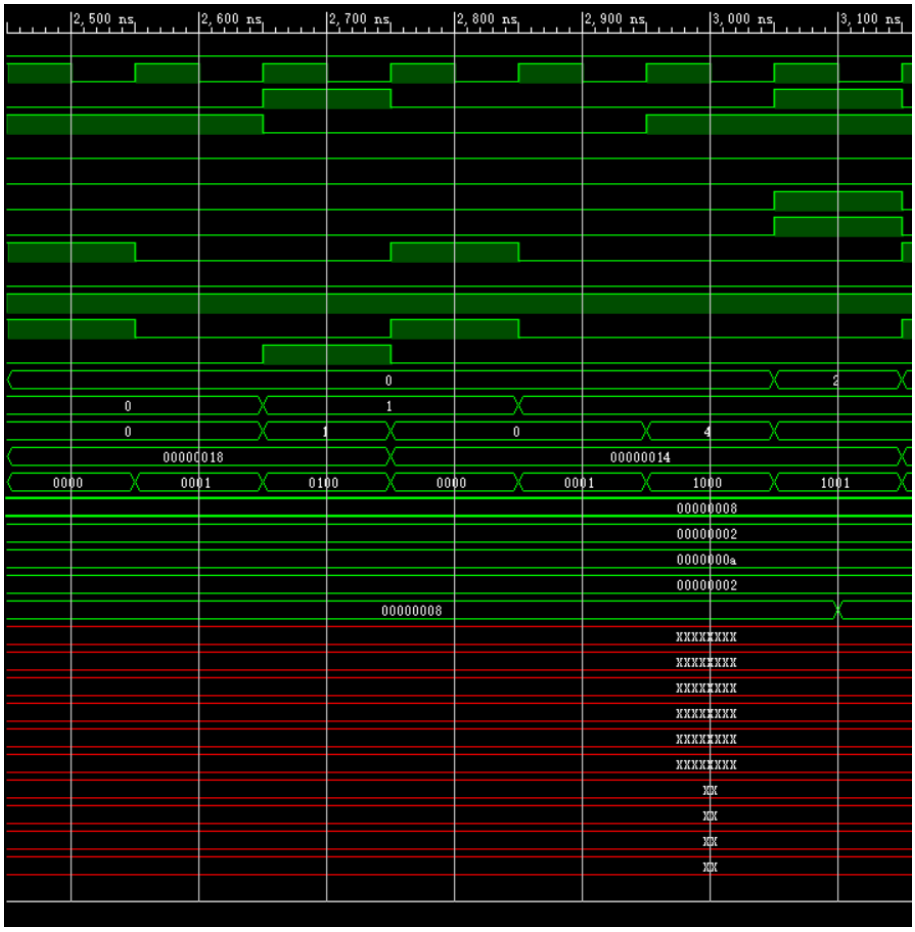
PC=0x10, and $5, $4, $2



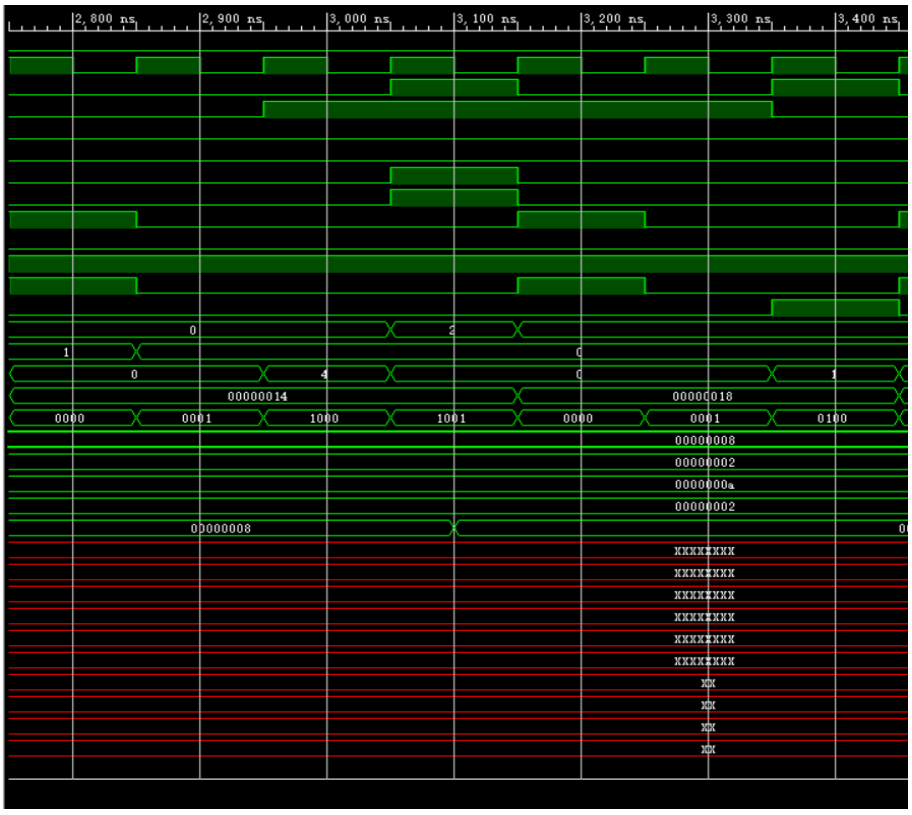
PC=0x14, sll $5, $5, 2



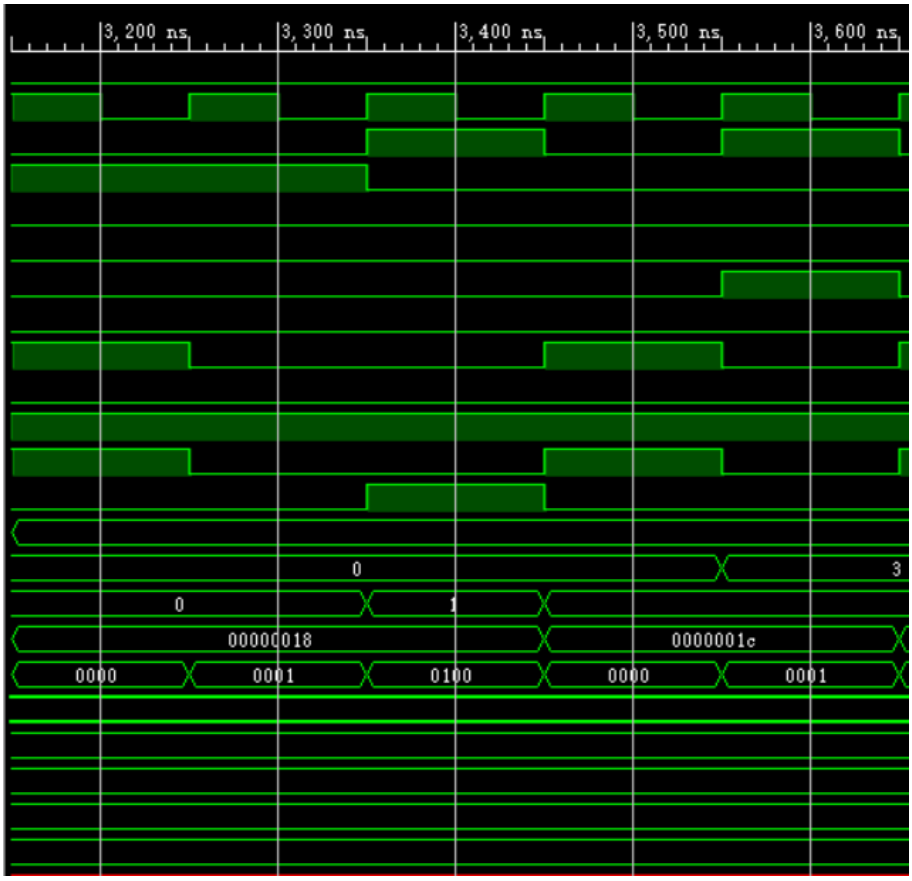
PC=0x18, beq $5,$1,-2(跳转)



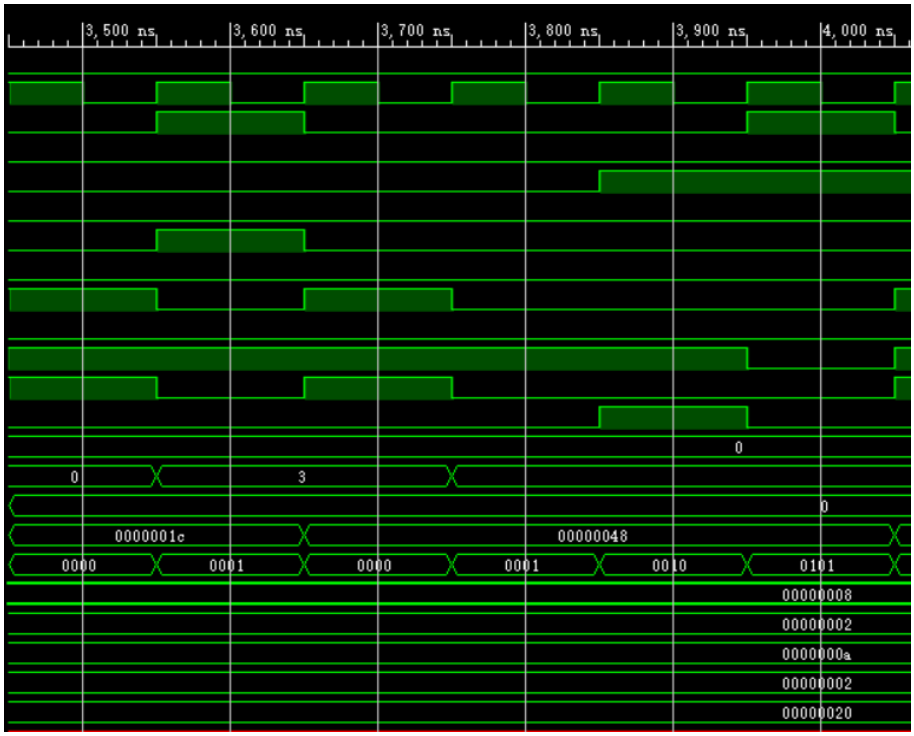
PC=0x14, sll $5,$5,2



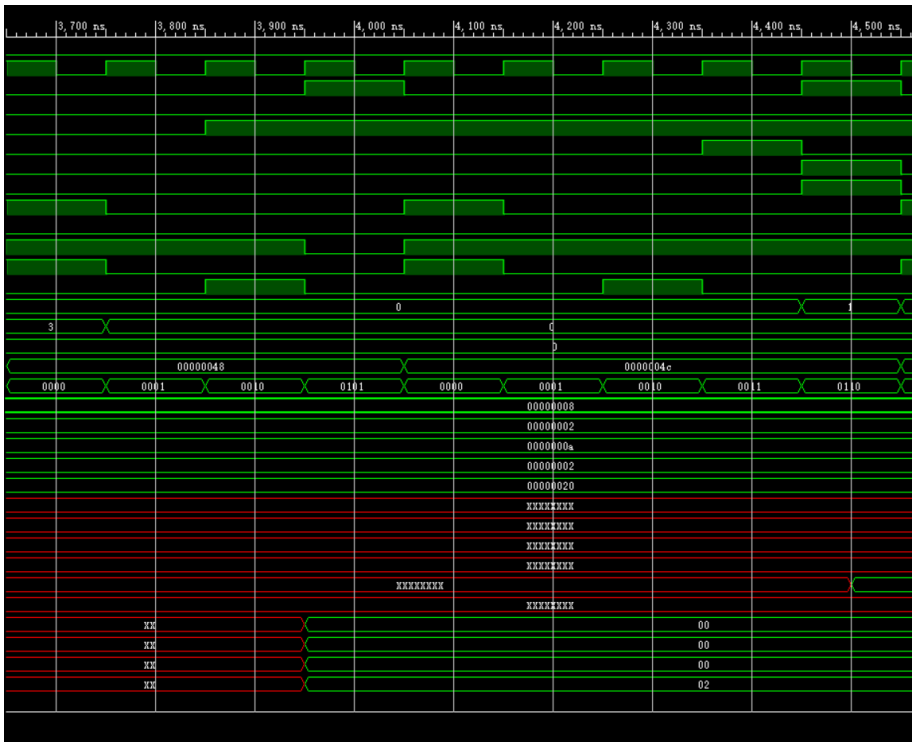
PC=0x18, beq $5,$1,-2（不跳转）



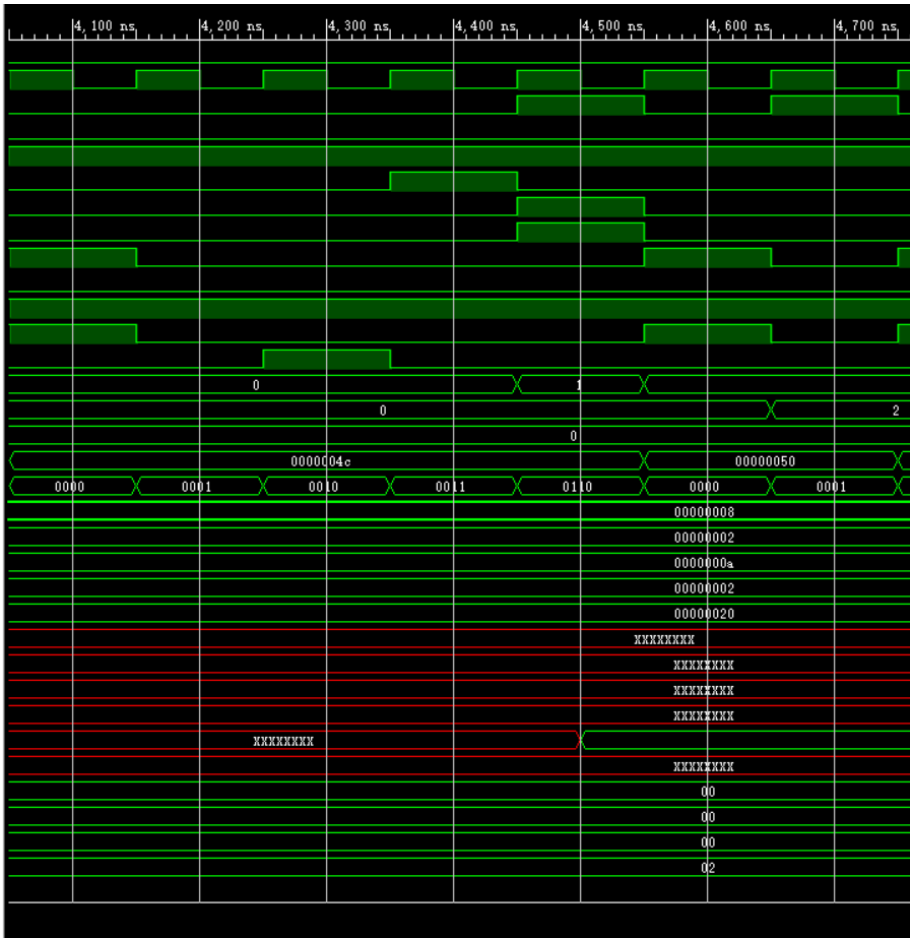
PC=0x1C, jal 0x0000048



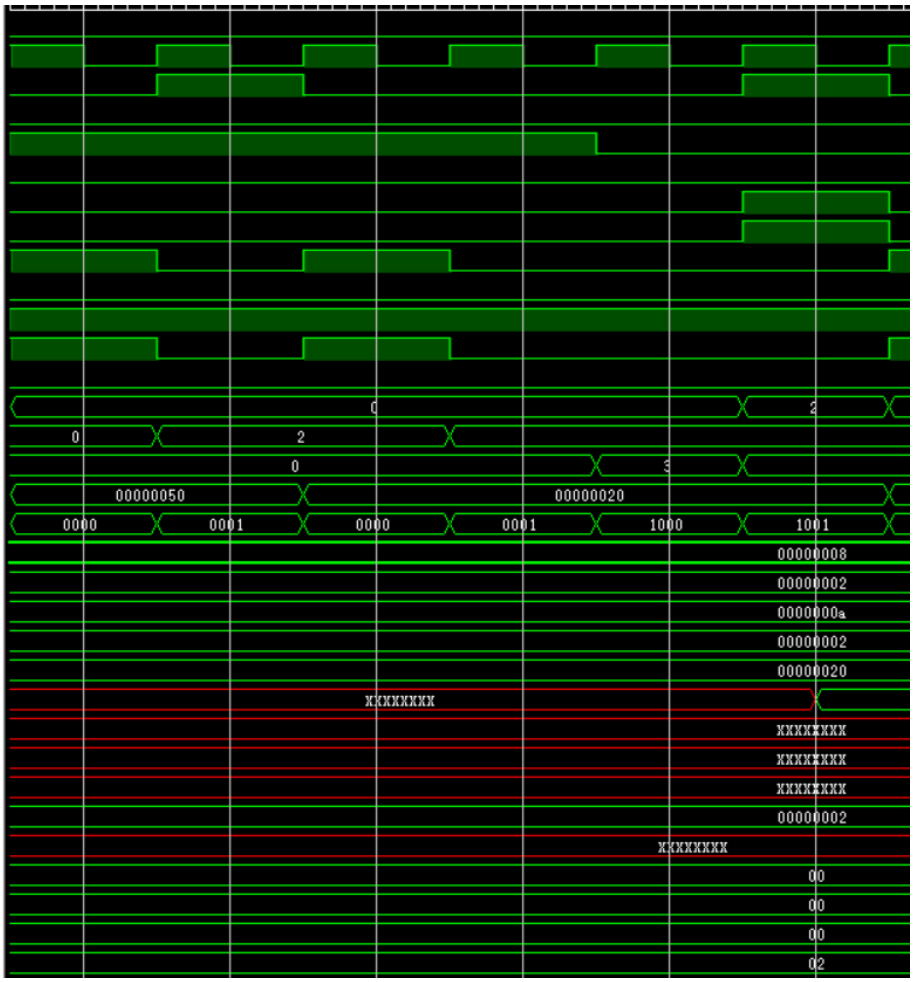
PC=0x48， sw $2,4($1)



PC=0x4C， lw $12,4($1)

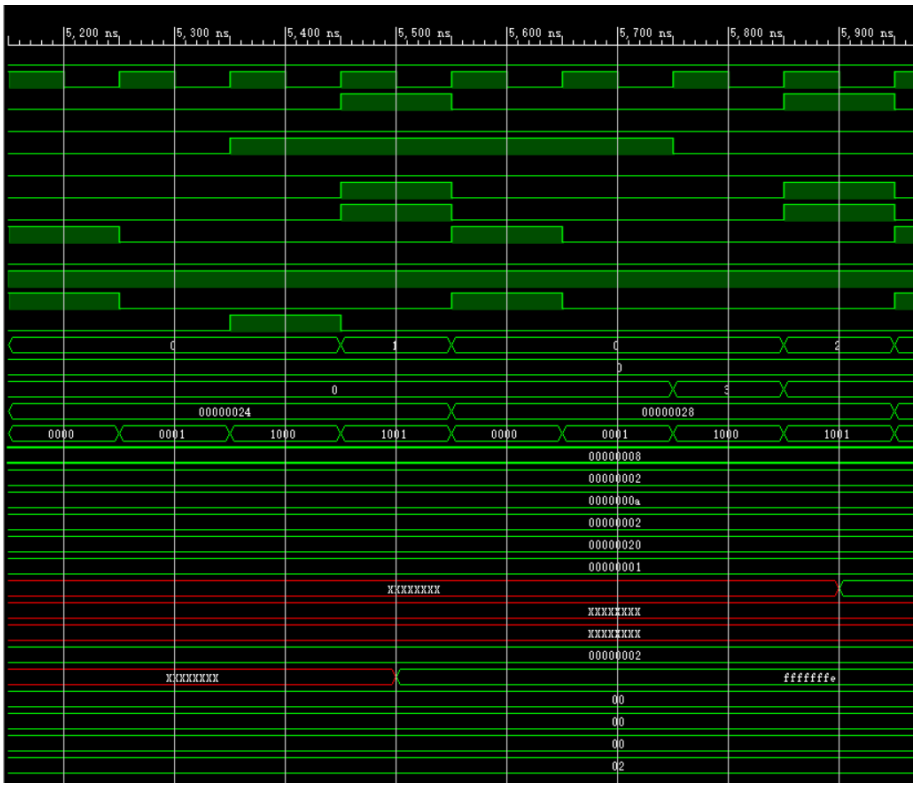


PC=0x50, jr $31

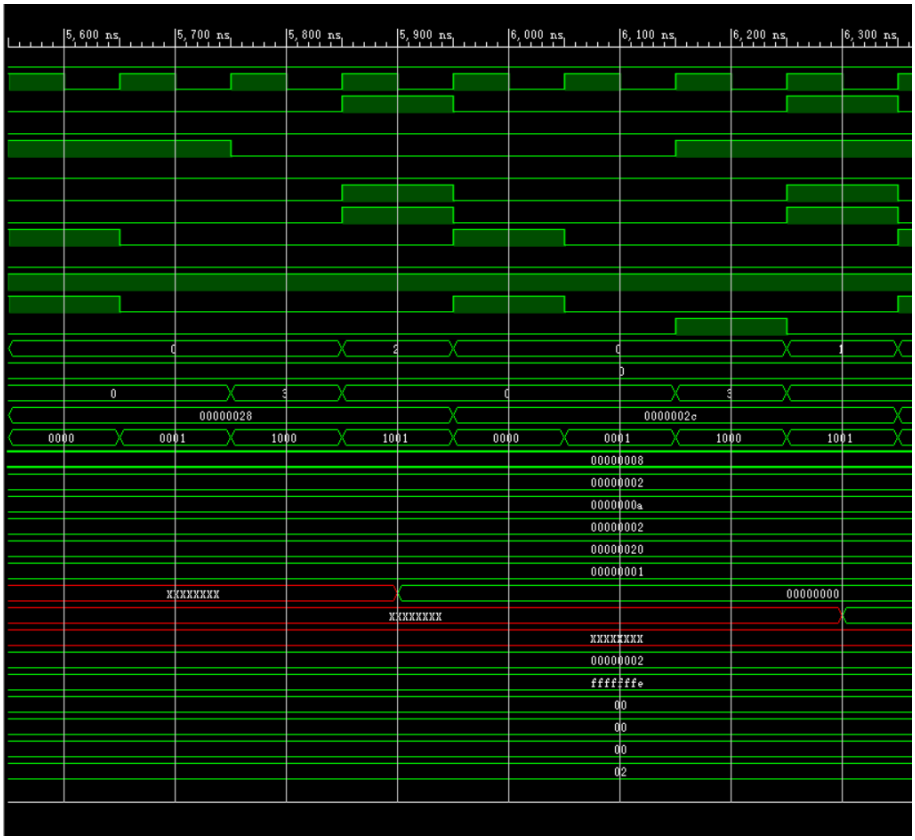


PC=0x20, slt $8, $12, $1

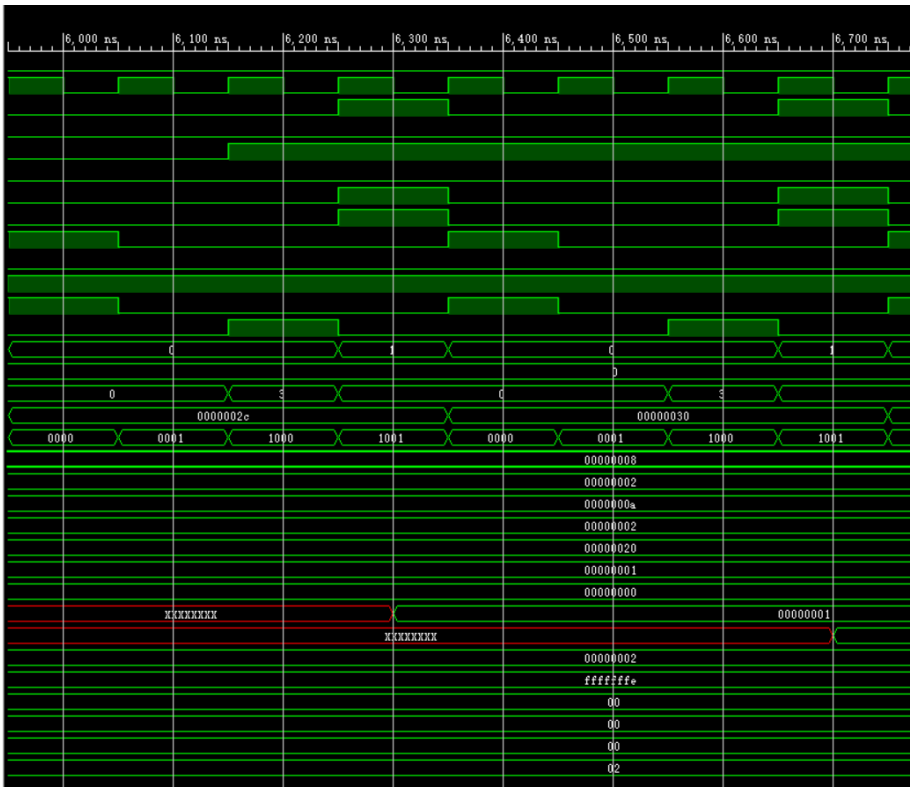
PC=0x24, addi $14,$0,-2



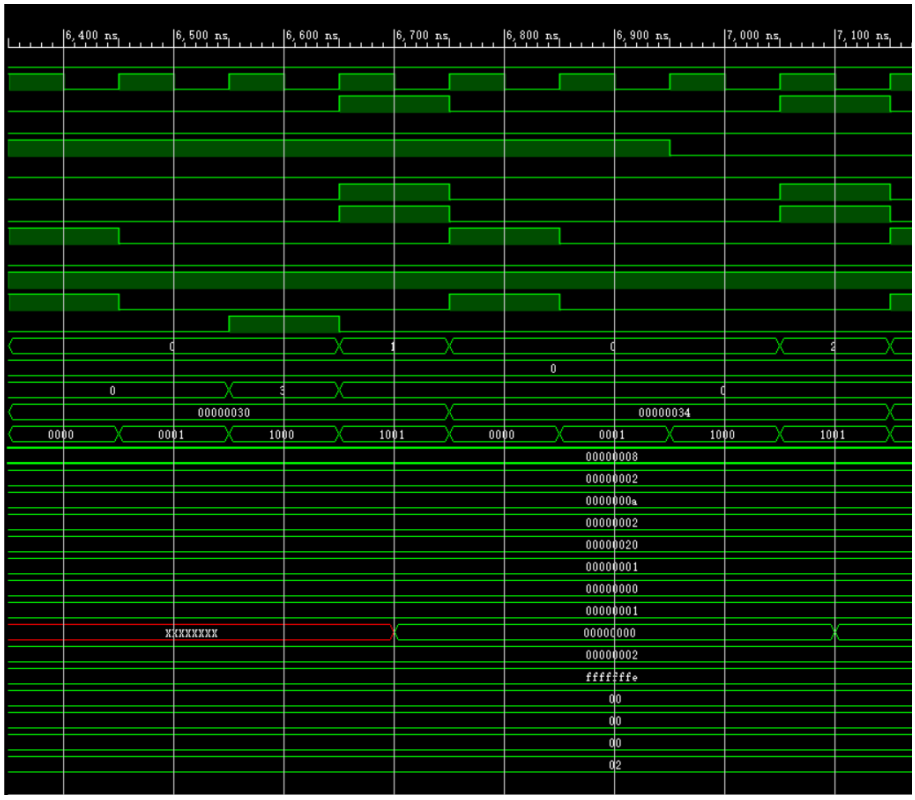
PC=0x28, slt $9,$8,$14



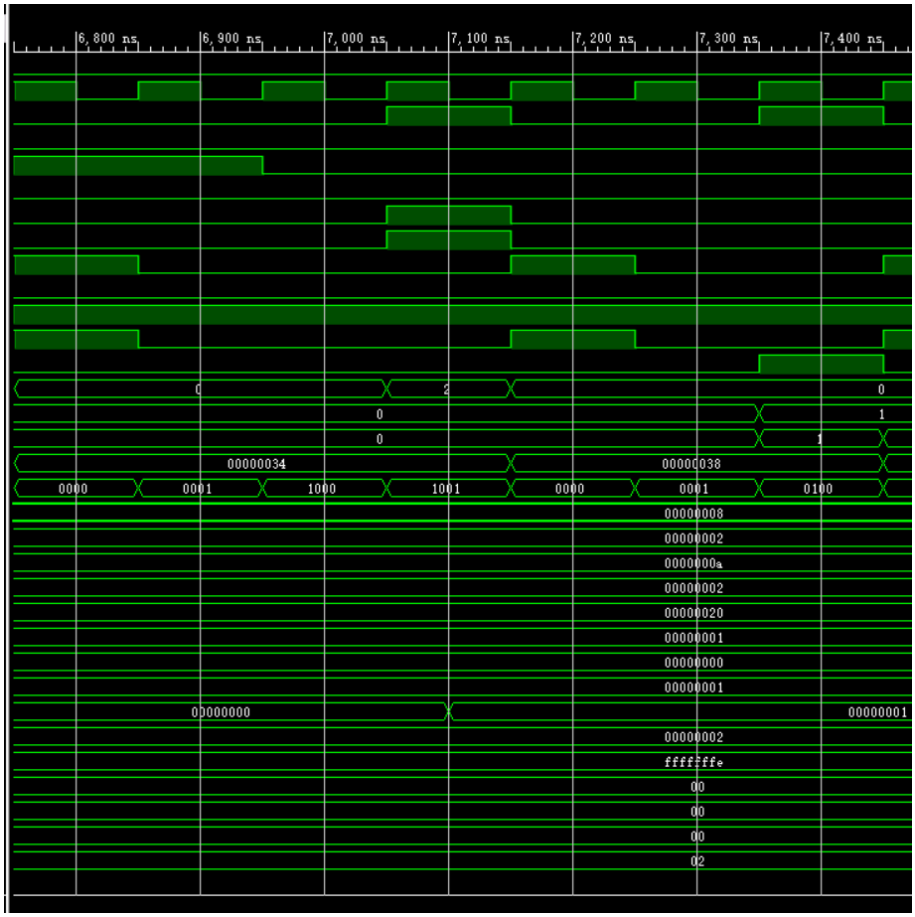
PC=0x2C, slti $10,$9,2



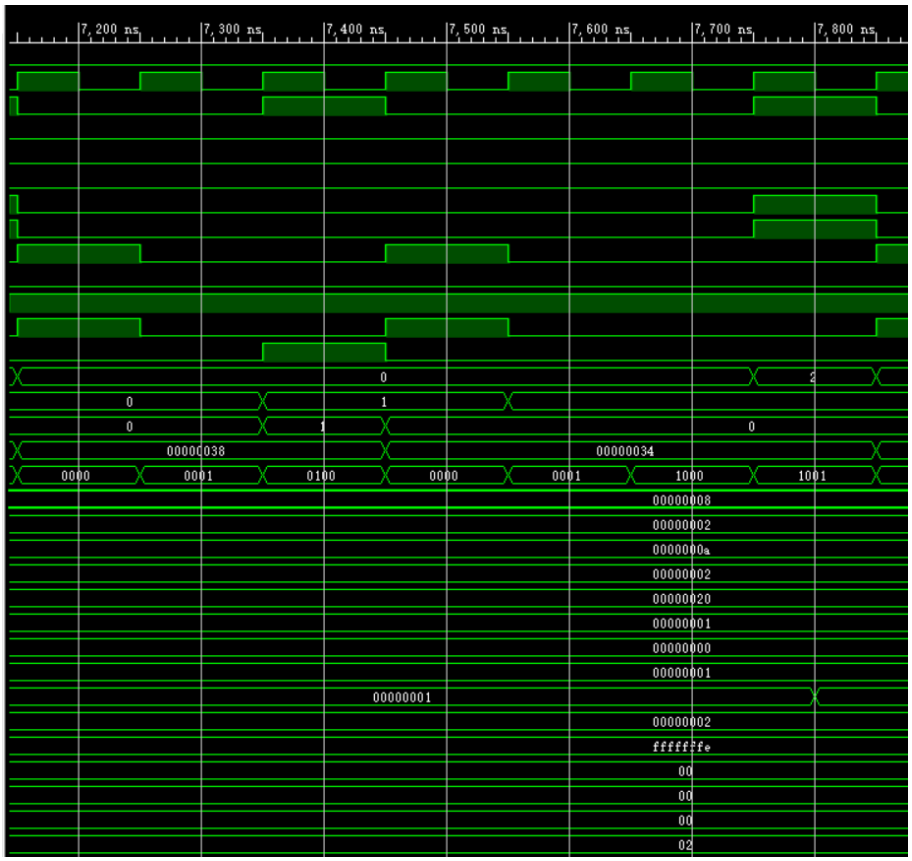
PC=0x30, slti $11,$10,0



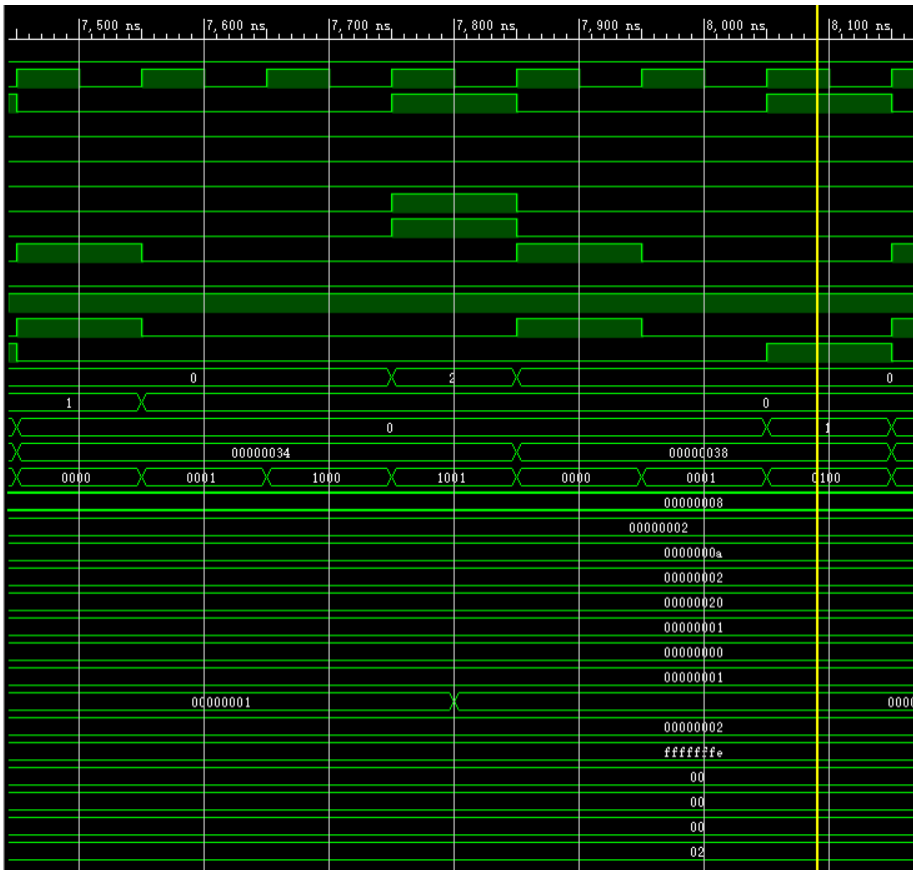
PC=0x34, add $11,$11,$8



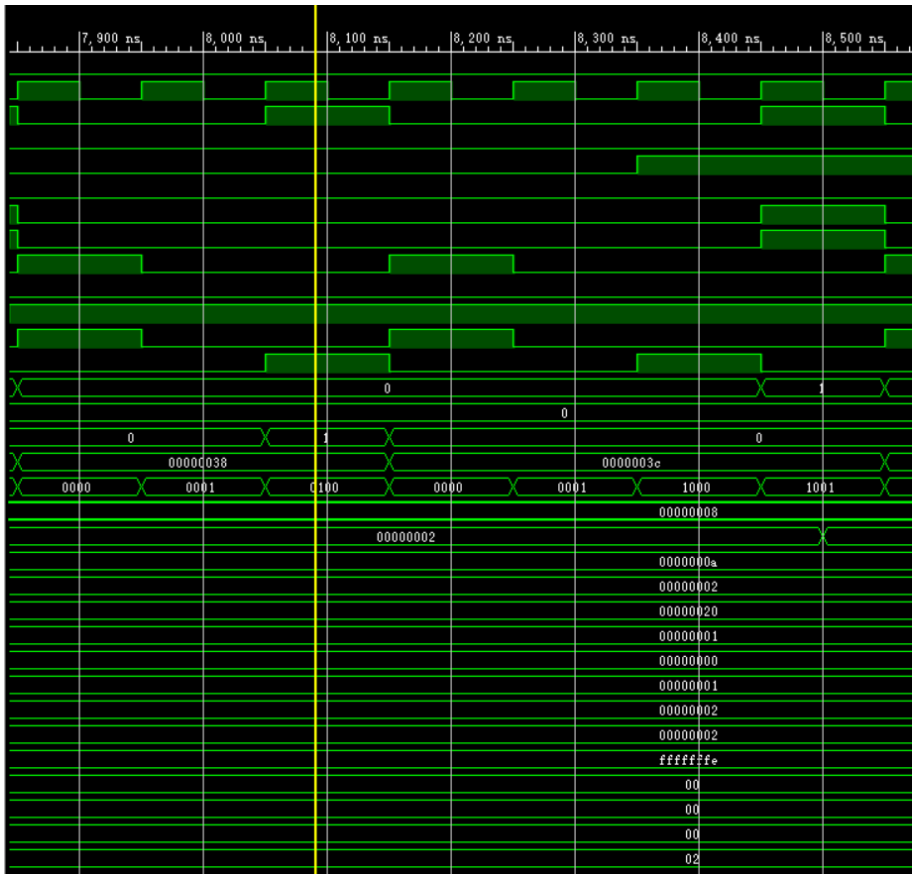
PC=0x38，bne $11,$2,-2(跳转)



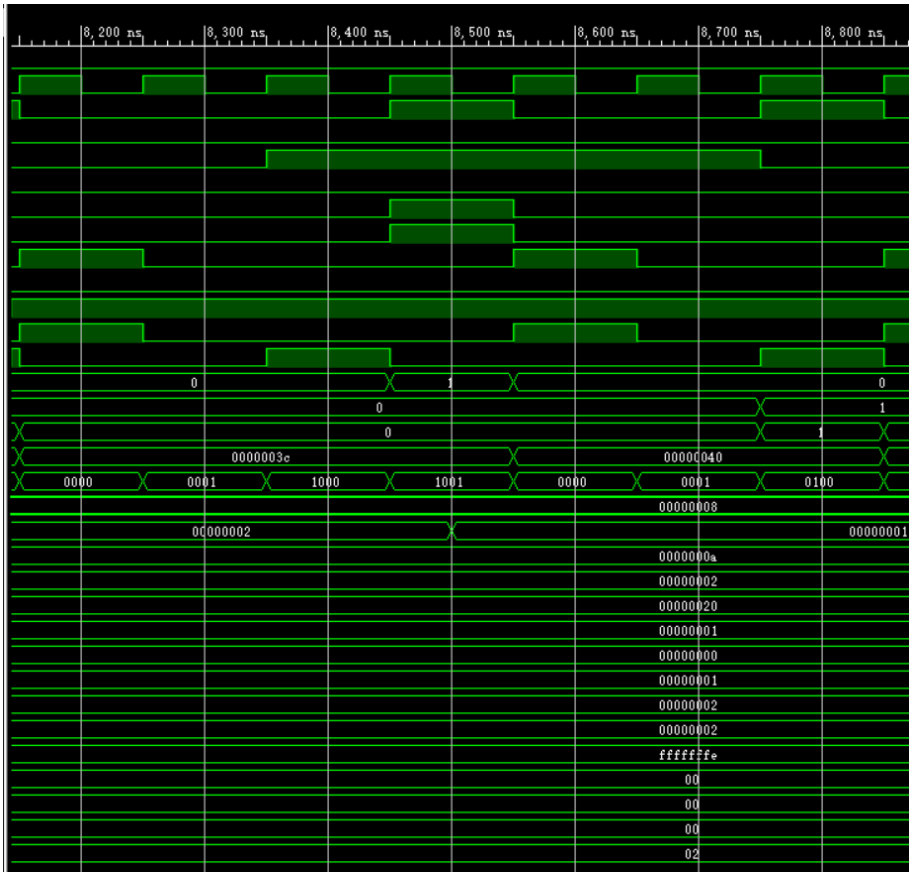
PC=0x34， add $11,$11,$8



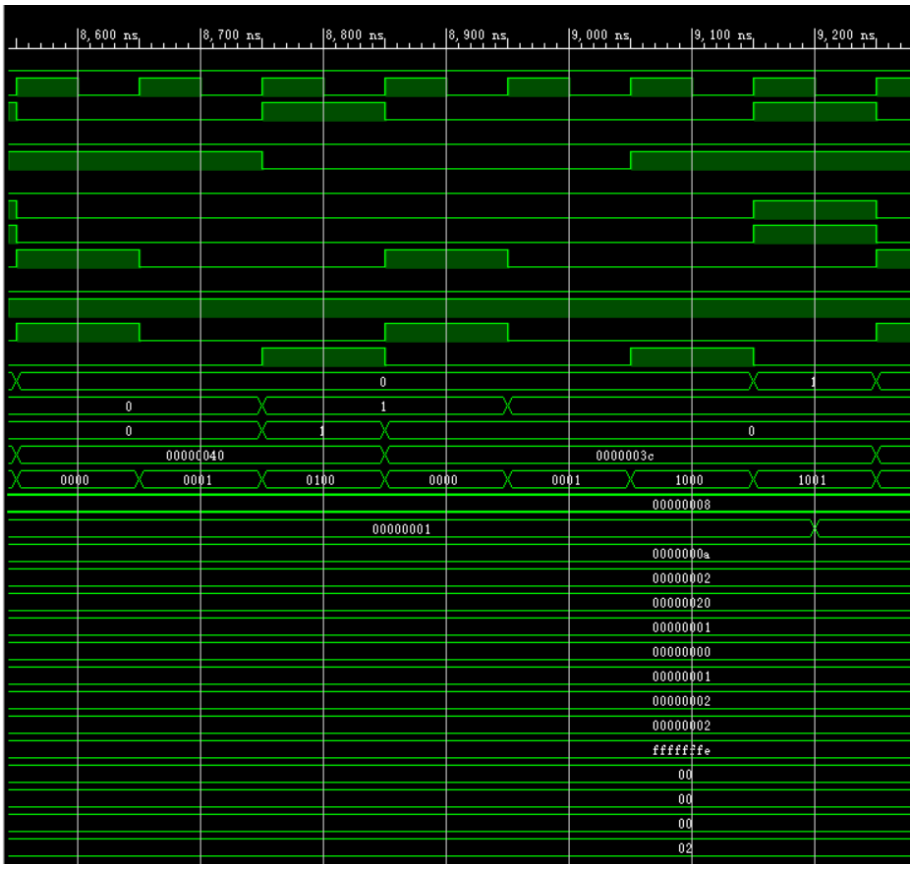
PC=0x38，bne $11,$2,-2(不跳转)



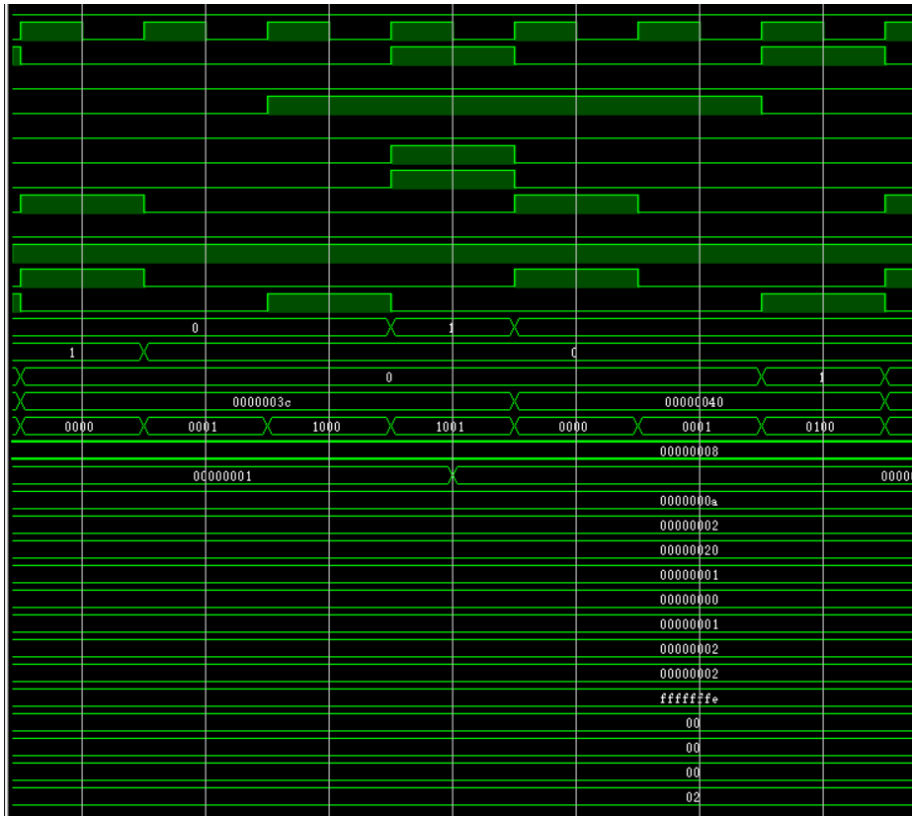
PC=0x3C, addi $2,$2,-1



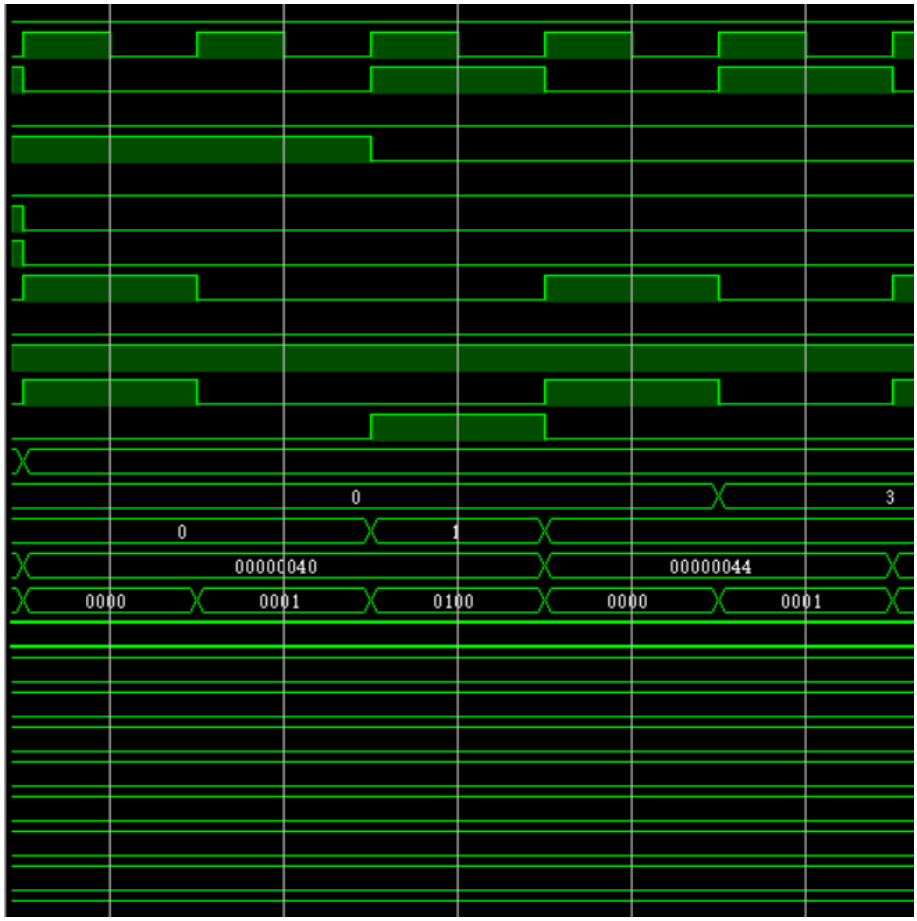
PC=0x40，bgtz $2，-2(跳转)



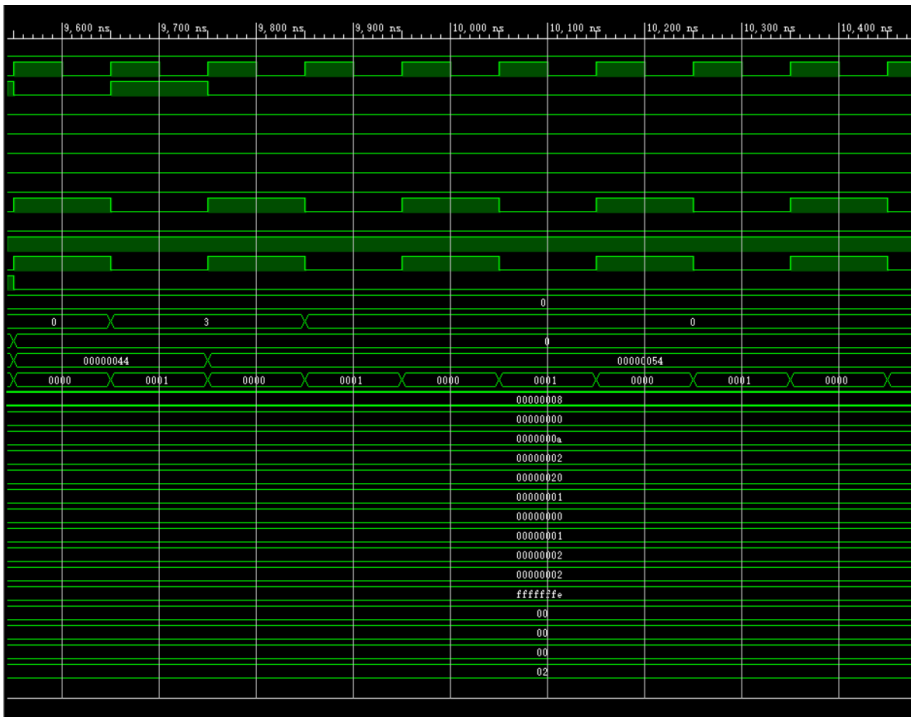
PC=0x3C, addi $2,$2,-1



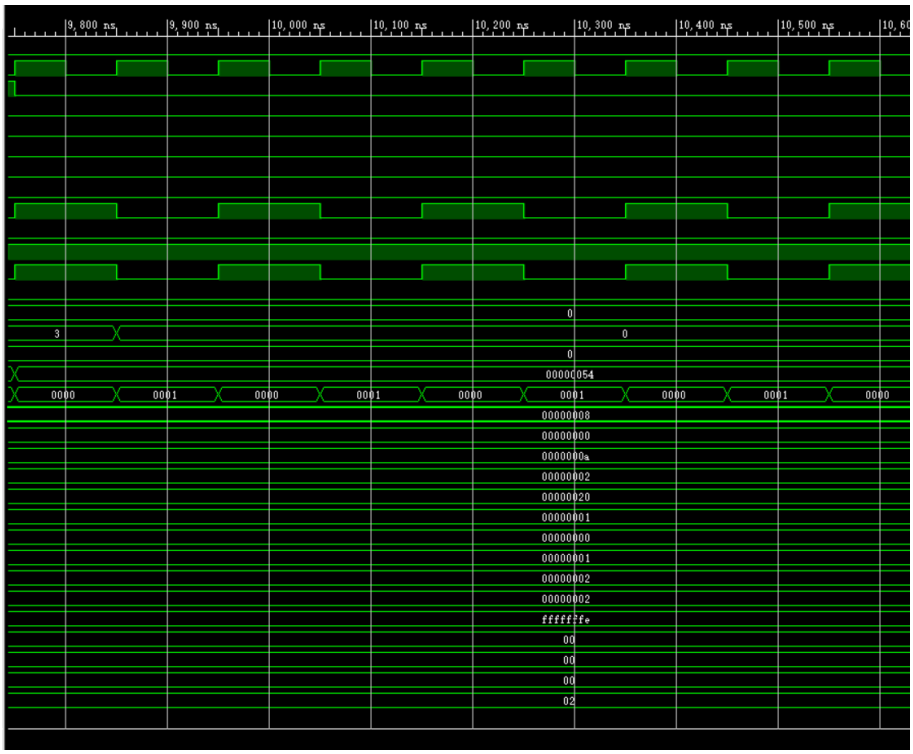
PC=0x40，bgtz $2，-2(不跳转)



PC=0x44, j 0x0000054



PC=0x54, halt



此后PC的值一直维持在0x00000054。

还有Reset功能，限于篇幅限制，结果不在这里展示了。

至此验证了CPU设计的准确性。

e. CPU设计的综合与实现

(1) 过程

实现过程

由于要将CPU实现到Basys 3实验版上，要确认CPU设计是否正确，需要一些输出来确认，在本实验中选择使用7段数码管显示来输出运行结果，要提的一点是只有四个7段数码管，而同时要显示两个信号，只能每一个信号显示两位数字（即二进制表示的后8位）。下面简述显示方法：

使用SW15和SW14，记为SW_in来选择显示内容。

SW_in = 00：显示 当前 PC值:下条指令PC值；

SW_in = 01：显示 RS寄存器地址:RS寄存器数据；

SW_in = 10：显示 RT寄存器地址:RT寄存器数据；

SW_in = 11：显示 ALU结果输出:DB总线数据。

代码实现如下：

Display模块

输入为要显示的8个32位数字，SW_in选择信号，输出为要显示的两个数。

```
always @(SW_in)begin
    if(0 == reset) begin
        out1 = 8'b1111_1111;
        out2 = 8'b1111_1111;
    end
    else begin
        case(SW_in)
        2'b00: {out1,out2} = {PC,PCNext};
        2'b01: {out1,out2} = {RSAddr,RSData};
        2'b10: {out1,out2} = {RTAddr,RTData};
        2'b11: {out1,out2} = {ALUResult,DB};
        default: {out1,out2} = 16'b0000_0000_0000_0000;
        endcase
    end
end
```

要实现扫描显示，首先要有一定的扫描频率，而人眼的视觉频率极限是60Hz，故选择用190Hz的时钟频率来刷新数码管。

代码如下：（clk1000为一个频率为1000Hz的时钟，在按键消抖时使用）

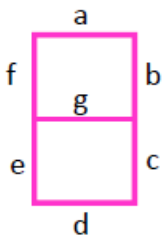
Clkdiv模块

输入：最高为450MHz的系统时钟mclk

输出：频率分别为190Hz和1000Hz的时钟频率

```
reg [26:0] q;

always @(posedge mclk) begin
begin
  if (reset == 0)
    q <= 0;
  else
    q <= q + 1;
  end
end
assign clk190 = q[14];
assign clk1000 = q[18];
```

其次还需要一个将BCD码译为8数码管点亮信号（共阳极）的模块，设计有下图的对应位置决定。



代码如下：

_7_seg_display模块

输入：要显示的BCD码，Reset控制信号

输出：8位数码管激活信号

```
always @(BCD)
begin
if(0 == Reset)
    code = 8'b1111_1111;
else begin
case(BCD)
    4'b0000: code = 8'b1100_0000;
    4'b0001: code = 8'b1111_1001;
    4'b0010: code = 8'b1010_0100;
    4'b0011: code = 8'b1011_0000;
    4'b0100: code = 8'b1001_1001;
    4'b0101: code = 8'b1001_0010;
    4'b0110: code = 8'b1000_0010;
    4'b0111: code = 8'b1101_1000;
    4'b1000: code = 8'b1000_0000;
    4'b1001: code = 8'b1001_0000;
    4'b1010: code = 8'b1000_1000;
    4'b1011: code = 8'b1000_0011;
    4'b1100: code = 8'b1100_0110;
    4'b1101: code = 8'b1010_0001;
    4'b1110: code = 8'b1000_0110;
    4'b1111: code = 8'b1000_1110;
    default: code = 8'b1111_1111;
    endcase
    end
end
```

最后需要一个扫描模块，该模块按照时钟频率扫描显示4个数字。

代码实现如下：

Show模块

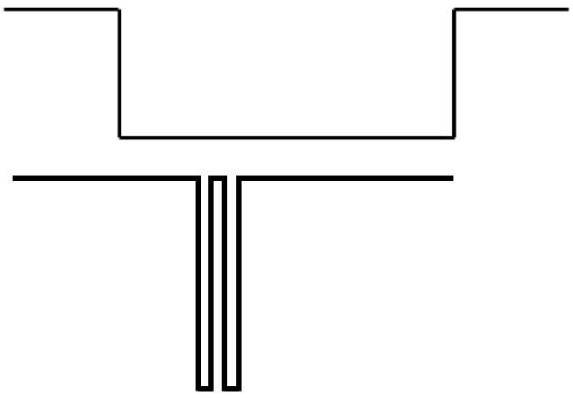
输入：扫描时钟，Reset控制信号，要显示的两个数

输出：数码管位选信号，数码管激活信号

```
reg [3:0] BCD;

initial place = 4'b1110;


always @(posedge CLK) begin
    case(place)
    4'b1110:begin
        place = 4'b1101;
        BCD = in2[7:4];
    end
    4'b1101:begin
        place = 4'b1011;
        BCD = in1[3:0];
    end
    4'b1011:begin
        place = 4'b0111;
        BCD = in1[7:4];
    end
    4'b0111:begin
        place = 4'b1110;
        BCD = in2[3:0];
    end
    default: begin
        place = 4'b1110;
        BCD = 4'b1111;
    end
    endcase
end

_7_seg_display _7_seg(
    .Reset(Reset),
    .BCD(BCD),
    .code(code)
);
```

除了显示之外，还要用按键的频率作为时钟改变PC，为了避免按键抖动导致PC的改变过快而出现问题，现在要实现按键消抖。

原理：



从上到下分别是理想的按键，受到干扰的按键，普通的按键情况。

为了使普通按键变成理想按键，同时避免发生干扰，可以采用延时按键消抖。

即将持续20ms稳定不变的按键输入当作输入，舍弃其余输入，为此设计了avoidShake模块。

avoidShake模块

输入：1000Hz的时钟，按键输入key_in

输出：消抖后的按键输入

功能说明：将持续20ms稳定不变的按键输入当作输入，舍弃其余输入，为此设计了avoidShake模块

设计：为了实现该功能，因为时钟周期为1ms，所以使用一个标志标记20ms内按键是否变化，所以也要保存连续两次的按键输入，使用一个变量记录从上一次按键开始未变化的时间。

主要代码如下：

```
reg [19:0] fifo;
reg [1:0] key_in_r;
wire change;

initial
    key_out = 1'b1;

always @(posedge clk1000) begin
    key_in_r <= {key_in_r[0],key_in};
end

assign change = key_in_r[0] ^ key_in_r[1];

always @(posedge clk1000) begin
    if(1 == change)
        fifo <= 20'b0;
    else fifo <= {fifo[18:0],1'b1};
end

always @(posedge clk1000)
begin
    if(20'hf_ffff == fifo)
        key_out <= key_in_r[0];
end
```

(2) 结果展示

以下展示个别几条指令在不同时钟周期的显示内容，从上到下依次为：

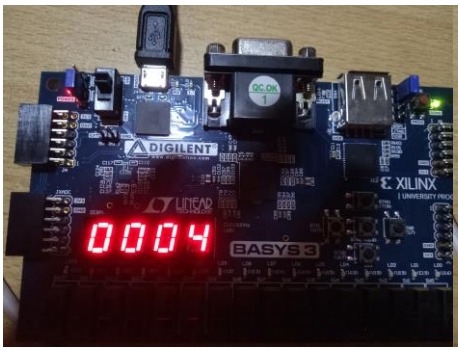
PC:下一条PC

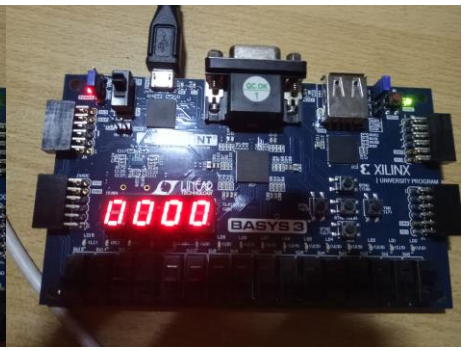
RS编号:RS内容

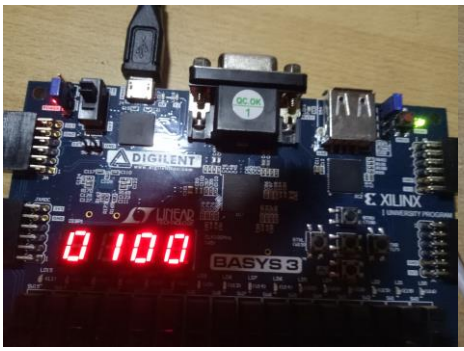
RT编号:RT内容

ALU输出:写回的数据

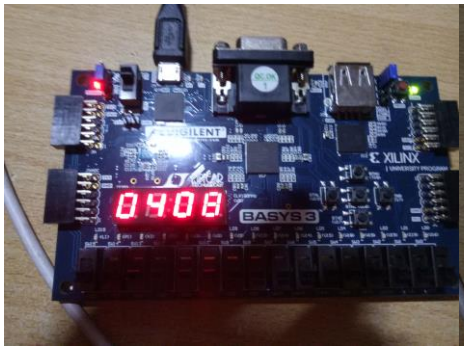
PC=0x04 addi $1,$0,8

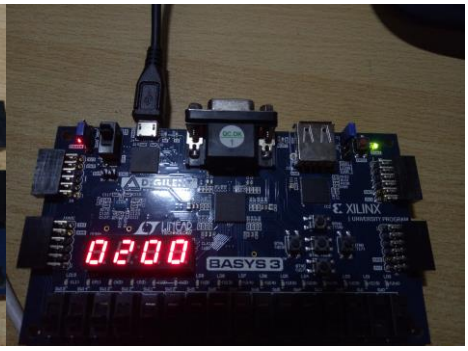


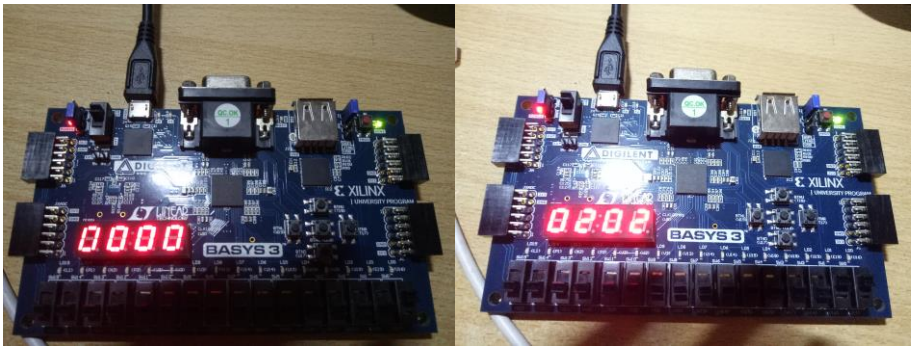




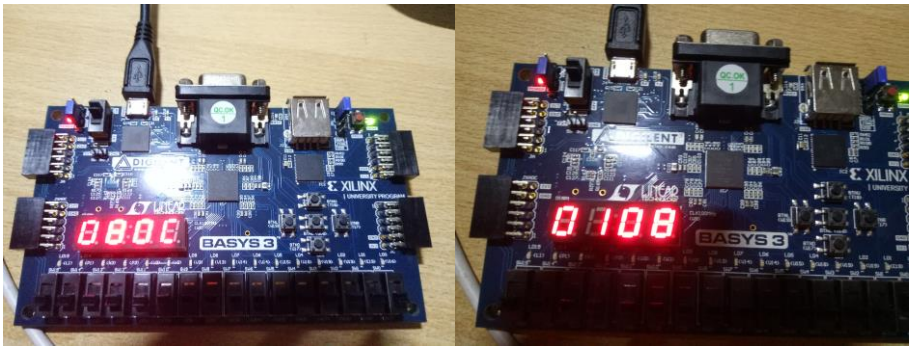
PC=0x08 ori $2,$0,2

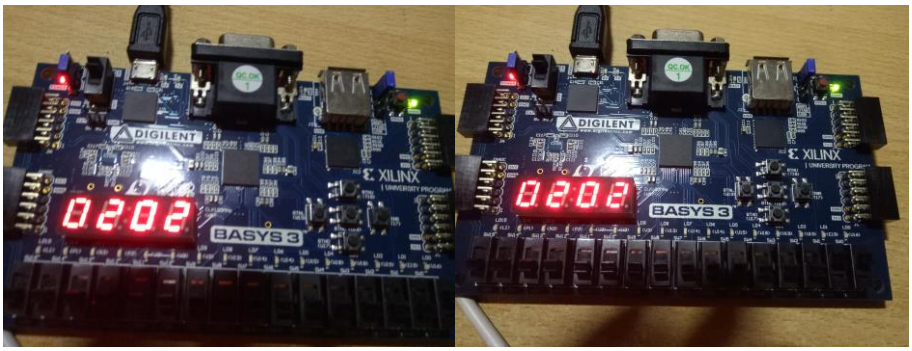




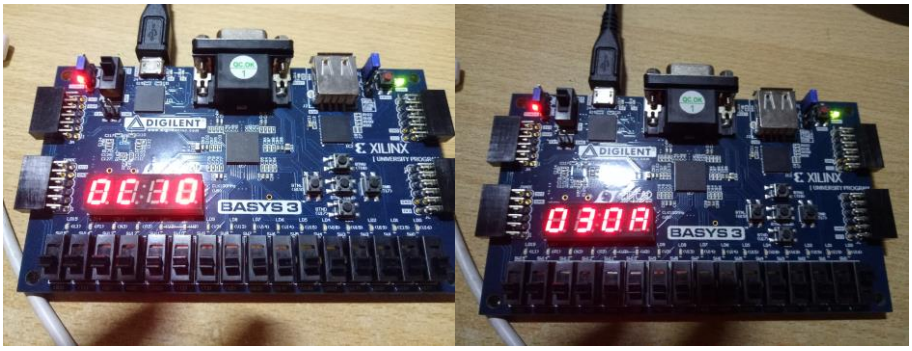


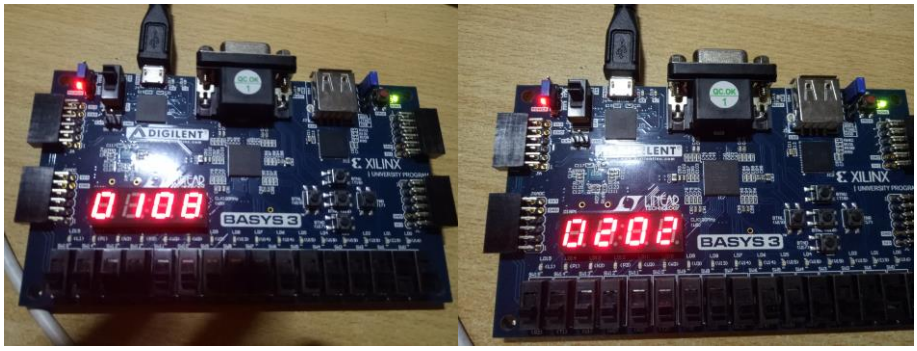
PC=0x08 or $3, $2, $1





PC=0x0C sub $4, $3, $1





六. 实验心得

体会和建议。

体会：

对于多周期CPU设计本身的体会：

（1） 我思考的第一个问题是时钟边沿的选择问题，当然这也是老师上课的时候提出的第一个问题。如何让这几个时序逻辑的部件相互配合而不发生竞争和冒险？我一开始的朴素想法便是将上升沿和下降沿错开放置，但是发现控制单元CU的时钟边沿选择如果和PC模块相同会导致PC模块的PC地址更新在相关的控制信号到来前发生，如果和其余缓冲寄存器或者寄存器模块相同就会导致部分寄存器写入或读出错误的内容，原因也是控制信号没有及时更新。所以必须另寻他法。最终通过一种一种假设的推翻，找到了**使用由时钟边沿触发的PC模块**的方法，而不是像许多身边的同学一样，使用不带时钟边沿触发的PC模块。这个方法便是实验报告正文中也提到过的**提前一个阶段将PCWre和PCSrc的信号发出**，使得PC模块被触发时所需信号已经是想要的信号，从而PC的更新是正确的。比如在WAL状态到来时就应该更改PCWre和PCSrc，使得下一条指令的IF阶段到来时能过正确更新PC。

这样虽然花多了一点功夫，但是也锻炼了我在一条思路行不通的时候改变思路的能力，分析问题也多了一点经验。

（2） 我思考的第二个问题仅仅依靠周期能否完全体现指令处理过程不同的阶段。问题的关键在于需不需要只在特定的阶段让某些特定部件运行，以体现指令的执行正在不同的阶段，这也我否决了。我的考虑首先是有一些部件是在多个阶段都被使用的，比如DBDR，其次是这样做没有意义，因为只要控制得当，在其它不需要运行的阶段运行该部件不影响结果。这里的**控制得当我认为是对于那些**

控制能否写入的使能信号，比如IRWre，RegWre不能任意赋值，必须明确他们在每一个阶段对于每一条指令的值。我在单周期的实验中也曾有过这个问题，这是该问题在我这里的又一次思考，我对**资源重复**思想的体会又深了一些。

（3） 还有的问题也在实验报告正文里面提到过，就是**寄存器是在哪一个或者哪几个阶段更改输出**。我犯过的错误是误以为IR寄存器在ID阶段才更改，误以为DBDR寄存器只有在MEM阶段的更改才有作用。后来才确定了所有的几个寄存器更改输出的阶段，以及DBDR寄存器在EXE阶段的输出也有作业。

（4） 伴随着对DBDR寄存器的误解，我对运算单元ALU的输出也有一定误解，没有如老师所给的数据通路图所示将其输出一条线连在ALUoutDR，另一条线连在DBDR，导致ALU的运算结果不能正确写入到寄存器。所以在EAL状态下需要改变DBSrc信号。

（5） ControlUnit和RegFile两个模块中同时存在组合逻辑和时逻辑的理解。正如在数字电路课程中学习的，组合逻辑是这样一种逻辑：电路的输出只取决于电路的输入，而时序逻辑是不一样的：电路的输出取决于电路的输入和电路当前所储存的状态。在这两个模块里面同时采用这两种逻辑的目的是解决某个模块的输出依赖于模块内保存的值，但是却不依赖于时钟信号的问题。具体来讲，在CU模块中，控制信号的变化都不依赖于时钟信号，而只取决于当前状态和操作码，将状态的变化与控制信号的变化分开来处理能够使信号的变化更加稳定。

（6） 即使对于同一条指令，在不同的时钟周期中，所需要的控制信号也会不相同，这一点是与单周期CPU较大的区别之一，也体现在上面的表5中。

关于实验过程的体会：

（1） 我大多数的问题，都是在填表5的过程中发现并且解决的，这给我的提示是要在实践中发现问题，而不能空想。虽然我的表5填了两三遍，看似浪费时间，但是在填完这一张表之后我对于本次实验的CPU设计了然于胸，有了对于数据通路和时钟周期的把握，在后面代码的编写过程中也避免了许多错误。

（2） 对于软件vivado的使用，我已经找到了根据软件报错信息debug的基本方法，即对于编程错误，先看message提示，再去log里面逐行查找warnings与errors；对于逻辑错误，先输出一段波形，然后再查找集中问题所在，这才将所有相关的信号波形输出，避免一开始就迷失在大量的信号波形中。

（3） 如何避免或减少初始化语句的使用。在if-else语句中使用else来处理高阻态和

不定态和case语句中使用default来处理高阻态和不定态，可以减少初始化语句的使用。而减少初始化语句的使用一是可以避免因为没有初始化带来的问题，二是可以处理各种未知情况，增强程序的稳定性。

不足：

（1） 关于Reset功能，我还有一个问题没有解决，问题是这样的：Reset的时候我重置了全部寄存器内容为零，全部存储器内容为零。在仿真中Reset功能完全正常，在Reset之后的指令运行也完全正确；然而在实现到Basys 3实验板后，Reset功能出现异常，不仅没有重置寄存器和存储器，而且接下来的一条指令还出现了错误。让我百思不得其解。