中山大學
SUN YAT-SEN UNIVERSITY

《计算机组成原理实验》

实验报告

（实验二）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	16 计算机类 7 班
学 生 姓 名 ：	王锡淮
学 号 ：	16337236
时 间 ：	2017 年 11 月 19 日

成 绩 ：

实验二 ： 单周期CPU设计与实现

一. 实验目的

(1) 掌握单周期CPU数据通路图的构成、原理及其设计方法；

(2) 掌握单周期CPU的实现方法，代码实现方法；

(3) 认识和掌握指令与CPU的关系；

(4) 掌握测试单周期CPU的方法；

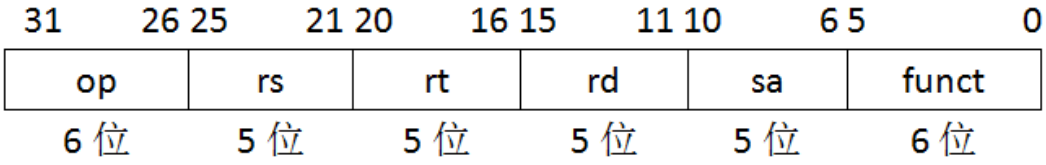
(5) 掌握单周期CPU的实现方法。

二. 实验内容

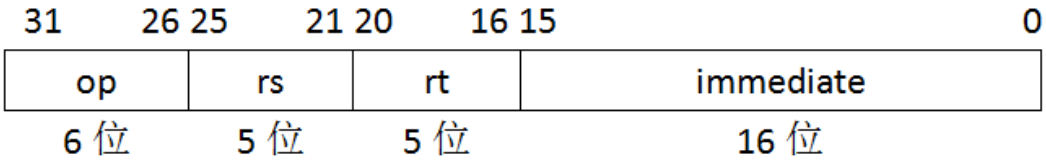
实验的具体内容与要求。

MIPS 指令的三种格式：

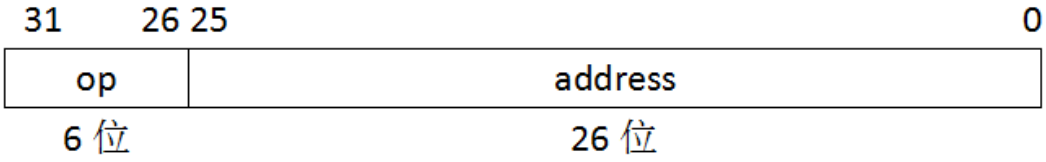
R 类型：



I 类型：



J 类型：



其中，

op：为操作码；

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

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

rd：只写。为目的操作数寄存器，寄存器地址（同上）；

sa：为位移量（shift amt），移位指令用于指定移多少位；

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

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

address：为地址。

设计一个单周期 CPU，该 CPU 能实现以下 16 条功能操作指令。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。**reserved为预留部分，即未用，一般填“0”。**

（2）addi rt , rs ,**immediate**

000001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs + (sign-extend)**immediate；immediate** 符号扩展再参加“加”运算。

（3）sub rd , rs , rt

000010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs - rt

==> 逻辑运算指令

（4）ori rt , rs ,**immediate**

010000	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。

（5）and rd , rs , rt

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

功能：rd←rs & rt；逻辑与运算。

（6）or rd , rs , rt

010010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

==>移位指令

（7）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

（8） slt rd, rs, rt 带符号数

011100	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

（9）sw rt ,**immediate**(rs) 写存储器

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

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

（10) lw　rt , **immediate**(rs)　读存储器

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

功能：rt ← memory[rs + (sign-extend)**immediate**]；**immediate** 符号扩展再相加。

即读取 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数，然后保存到 rt 寄存器中。

==> 分支指令

（11）beq　rs,rt,**immediate**

110000	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

特别说明：**immediate 是从 PC+4 地址开始和转移到的指令之间指令条数。**immediate 符号扩展之后左移 2 位再相加。为什么要左移 2 位？由于跳转到的指令地址肯定是 4 的倍数（每条指令占 4 个字节），最低两位是“00”，因此将 **immediate** 放进指令码中的时候，是右移了 2 位的，也就是以上说的“指令之间指令条数”。

（12）bne　rs,rt,**immediate**

110001	rs(5 位)	rt(5 位)	**immediate**

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

特别说明：与 beq 不同点是，不等时转移，相等时顺序执行。

（13）bgtz　rs,**immediate**

110010	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

（14）j　addr

111000	addr[27..2]

功能：pc <—{(pc+4)[31..28],addr[27..2],0,0}，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 位地址了，剩下最高 4 位由 pc+4 最高 4 位拼接上。

（15）jr rs

111001	rs(5 位)	00000	0000000000000000

功能：pc <—{rs}

说明：将 pc 赋值为 rs 寄存器的值。

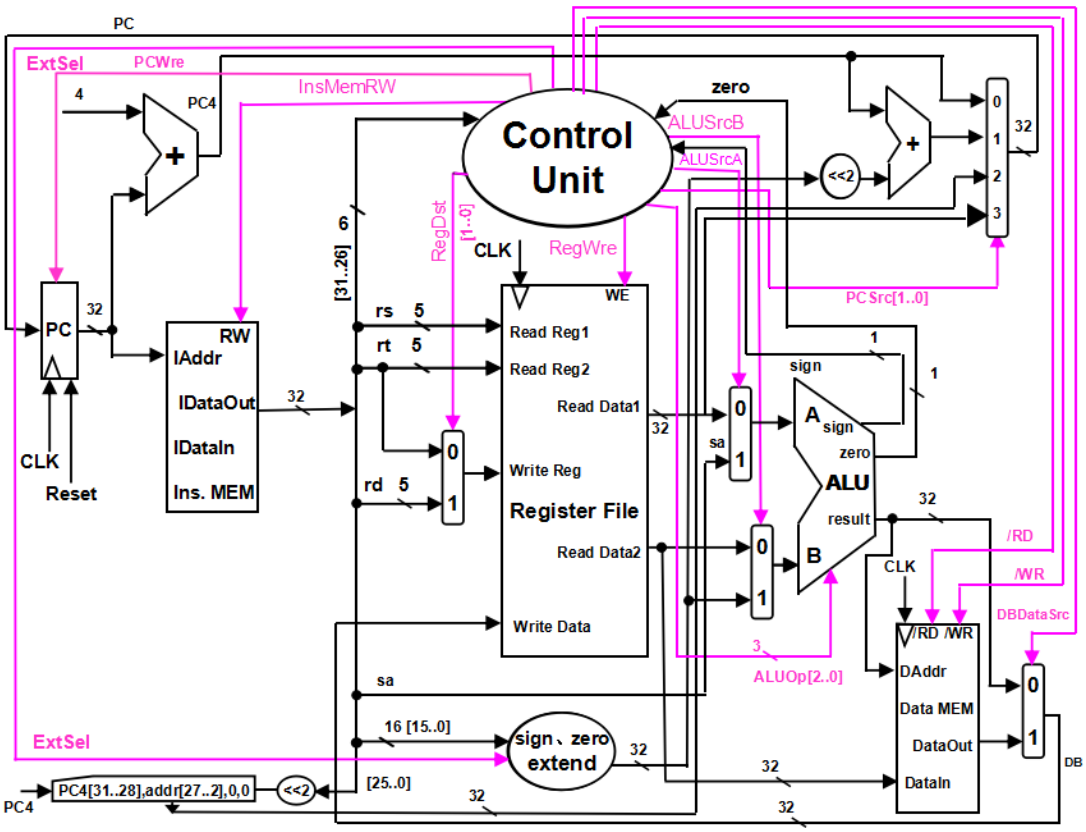
==> 停机指令

（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



上图是一个简单的基本上能够在单周期CPU上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，先给出寄存器地址，读操作时，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。

数据通路图中所使用的控制信号如下表所示：

控制信号的作用

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、addi、or、and、ori、beq、bne、bgtz、slt、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、sll、slt、	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、sw、lw

	beq、bne、bgtz	
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、slt、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、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、slt、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	输出高阻态
/WR	写数据存储器，相关指令：sw	无操作
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addi、ori、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、slt、sll
ExtSel	(zero-extend)**immediate(0 扩展)**，相关指令：ori	(sign-extend)**immediate (符号扩展)**，相关指令：addi、sw、lw、beq、bne、bgtz
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slt、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，zero=0); 10：pc<－{(pc+4)[31:28],addr[27:2],0,0}，相关指令：j; 11：pc<—rs	
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，指令存储器地址输入端口

IDataIn，指令存储器数据输入端口（指令代码输入端口）

IDataOut，指令存储器数据输出端口（指令代码输出端口）

RW，指令存储器读写控制信号，为 0 写，为 1 读

Data Memory：数据存储器，

Daddr，数据存储器地址输入端口

DataIn，数据存储器数据输入端口

DataOut，数据存储器数据输出端口

/RD，数据存储器读控制信号，为 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端口，其地址来源 rt 或 rd 字段

Write Data，写入寄存器的数据输入端口

Read Data1，rs 寄存器数据输出端口

Read Data2，rt 寄存器数据输出端口

WE，写使能信号，为 1 时，在时钟边沿触发写入

ALU： 算术**逻辑单元**

result，ALU 运算结果

zero，运算结果标志，结果为 0，则 zero=1；否则 zero=0

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

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 与 B 不带符号

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

111	Y = A ⊕ B	异或

在本次实验中指令执行的结果将在时钟**下降沿**存储在寄存器或存储器中，而PC在时钟**上升沿**变化，这样可以取得稳定效果。

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

CPU的设计：

CPU在处理指令时，一般需要经过以下几个步骤：

⑴ 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。

⑵ 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，同时还需要读取寄存器堆。

(3) 指令执行(EXE)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

(4) 存储器访问(MEM)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

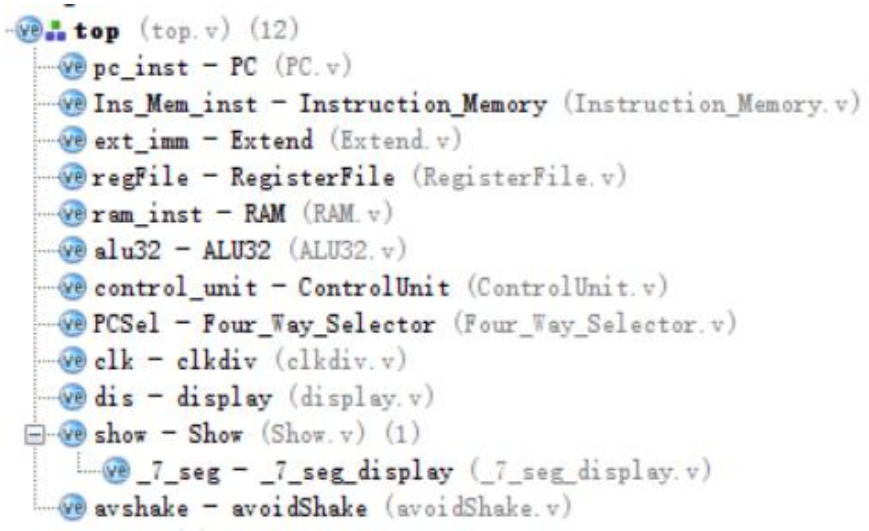
(5) 结果写回(WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

单周期CPU，是在一个时钟周期内完成这五个阶段的处理。

于是按照上图的五个阶段，逐个阶段地设计所需模块，并在最后将所有模块整合起来，得到一个完整的单周期CPU设计。

模块设计：

本次实验中设计全部模块如下：



下面在各个阶段对每个模块以及每个阶段的细节进行解释：

需要提到的一点是，由于除了PC模块、寄存器堆模块的写功能和存储器模块的写功能是同步（依靠时钟边沿）的之外，其余模块都是异步模块，即当PC模块输出要执行的PC地址之后，这些模块可以立即（相对于时钟频率）计算出这条要执行的指令，该指令用到的寄存器和存储器数据（包括位扩展），以及下一条PC的四种候选，和ALU运算结果和标志位。则当这些模块的输入变化时，该模块所需的各种输入已经是执行该条

指令所需的输入。

1. **取指令（IF）阶段：**

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：

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

always @(IAddr or IDataIn or RW) begin

  if(1 == RW) 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，下面来逐一分析：

a. 在指令存储器中当前执行指令的下一条指令的地址，由于MIPS指令定长，为32位4字节，又由于存储单元以字节为单位，则下一条指令地址即为当前指令地址加4。

   代码如下：

```
assign PCAddedFour = IAddrOut + 4;
```

b. 分支指令中的immediate所决定的地址。这里的immediate作为偏移量，则所决定的地址为a部分所言的当前执行指令的下一条指令的地址与经符号扩展的immediate再左移两位（由指令的设计决定）相加的结果。

   代码如下：

```
assign PCFromBranch = (Ext_immediate << 2) + PCAddedFour;
```

c. J型指令格式中的跳转指令（本实验中只有J指令），该种指令所决定的地址为a部分所言的当前执行指令的下一条指令的地址的高四位与指令中的26位立即数与低两位补零得到。

主要代码如下：

```
assign PCFromJIns = {PCAddedFour[31:28], address, 1'b0, 1'b0};
```

d. R型指令格式中的跳转指令（本实验中只有Jr指令），该种指令所决定的地址为rs寄存器中的内容，即寄存器堆模块（在第二阶段介绍）的输出之一。

得到了这四种下一PC地址的候选之后，需要一个PC选址信号来选择实际上PC的下一地址，为此我设计了一个四路选择器模块。

Four_Way_Selector模块

输入：四个输入数据，一个选择信号。

输出：被选择的输入数据。

功能说明：选择信号选择四个输入数据之一。

设计：主要代码如下：

```
module Four_Way_Selector(
  input [31:0] In0,
  input [31:0] In1,
  input [31:0] In2,
  input [31:0] In3,
  input [1:0] Selector,
  output reg [31:0] Out
  );

always @(Selector or In0 or In1 or In2 or In3) begin
  case(Selector)
  2'b00: Out <= In0;
  2'b01: Out <= In1;
  2'b10: Out <= In2;
  2'b11: Out <= In3;
  default: Out <= 32'b0;
  endcase
end

endmodule
```

在使用时的实例化代码如下：

```
Four_Way_Selector PCSel(
  .In0(PCAddedFour),
  .In1(PCFromBranch),
  .In2(PCFromJIns),
  .In3(Read_data1),
  .Selector(PCSrc),
  .Out(IAddrIn)
  );
```

2. **指令译码（ID）阶段：**

该阶段主要有三个任务，其一为将指令拆解成多个传递信息；其二为根据指令类型输出各种控制信号，其三为读取寄存器堆。下面对这三个任务进行分析：

任务一：

功能说明：如“实验内容”部分提到的三种MIPS指令格式，每一种格式都含有不一样的内容，要在将指令分解的同时将指令格式区分出来是相当困难而且对后续阶段的继续是相当不利的，于是采用数据冗余的办法，先不区分指令的格式，无论输入的是哪一种指令格式都拆分出三种指令格式所需的所有内容，再结合后面的任务二通过控制信号在各个模块将指令格式区分出来，并获取该指令所需要的内容来执行后续步骤。

设计：主要代码如下：

```
assign op = IDataOut[31:26];
assign rs = IDataOut[25:21];
assign rt = IDataOut[20:16];
assign rd = IDataOut[15:11];
assign sa = IDataOut[10:6];
assign Ext_sa = {27'b0000_0000_0000_0000_0000_0000_000,sa};
assign immediate = IDataOut[15:0];
assign address = IDataOut[25:0];
```

由于MIPS指令格式的一致性，每一个同名的内容都存放在指令的同一位置，可以不用区分指令类型便使用统一的方式获取以下内容：op, rs, rt, rd, sa, immediate, address。同时因为sa在sll指令中将被用作ALU的输入，而ALU被设计为32位，于是sa在此处将被零扩展为32位。同理，immediate也将被进行位扩展，但是不同的地方在于immediate需要根据指令的不同选择进行的是零扩展还是符号扩展。于是设计了一个模块完成此功能。

Extend模块

输入：扩展类型的选择信号ExtSel，原始数据。

输出：经扩展的数据。

功能说明：根据选择信号经行位扩展。

设计：主要代码如下：

```
always @(ExtSel or OriData) begin
  if(0 == ExtSel)
    ExtData <= {16'b0, OriData};
  else if(1 == OriData[15])
    ExtData <= {16'hffff, OriData};
  else ExtData <= {16'h0000, OriData};
end
```

如果选择的是零扩展，则直接在原始数据前补上16个0得到经扩展的数据；如果选择的是符号扩展，则在原始数据前补上16个原始数据的符号位得到经扩展的数据。至此，任务一从指令中提取出了将要用到的信息，接下来是将各个信息传递到各个模块，则任务一完成。

任务二：

功能说明：通过任务一得到的op信息并根据指令的设计生成控制信号，以便在后面的步骤中区分各种指令格式，在这里设计一个模块ControlUnit来完成该功能。

ControlUnit模块

该模块需要结合指令的设计来决定，于是列出指令与控制信号的对应关系如下图：

（注：本表中将不定态都设置为低电平，即0）

指令与控制信号的对应

Ins\Signal	Reset	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	InsMemRW	RD	WR	RegDst	ExtSel	PCSrc	ALUOp
000000(add)		1	0	0	0	1	1	1	1	1	0	00	000
000001(addi)		1	0	1	0	1	1	1	1	0	1	00	000
000010(sub)		1	0	0	0	1	1	1	1	1	0	00	001
010000(ori)		1	0	1	0	1	1	1	1	0	0	00	011
010001(a		1	0	0	0	1	1	1	1	1	0	00	100

nd)													
010010(or)		1	0	0	0	1	1	1	1	1	0	00	011
011000(sll)		1	1	0	0	1	1	1	1	1	0	00	010
011100(slt)		1	0	0	0	1	1	1	1	1	0	00	110
100110(sw)		1	0	1	0	0	1	1	0	1	1	00	000
100111(lw)		1	0	1	1	1	1	0	1	0	1	00	000
110000(beq)		1	0	0	0	0	1	1	1	1	1	01/00	001
110001(bne)		1	0	0	0	0	1	1	1	1	1	01/00	001
110010(bgtz)		1	0	0	0	0	1	1	1	1	1	01/00	001
111000(j)		1	0	0	0	0	1	1	1	1	0	10	000
111001(jr)		1	0	0	0	0	1	1	1	1	0	11	000
111111(halt)		0	0	0	0	0	0	0	0	0	0	00	000

出于可扩展性的考虑，本模块将采用case语句作为主体。

输入：指令操作码op，ALU的输出标志zero（zero为1代表运算结果为0，反之不为0），sign（sign为1代表运算结果为负数，反之代表为非负数）。

输出：PCWre，ALUSrcA，ALUSrcB，DBDataSrc，RegWre，InsMemRW，RD，WR，RegDst，ExtSel，PCSrc，ALUOp等控制信号，其功能在“实验原理”部分有所介绍。

功能说明：根据指令操作码和ALU的标志位输出控制信号。

设计：该模块相似的部分较多，故下面只选取其中典型的和相对较复杂的部分进行

说明。

在Control模块中，要实现的16条指令在其中的框架相似，下面用add指令进行说明。

```
6'b000000: //add
begin
  PCWre = 1;
  ALUSrcA = 0;
  ALUSrcB = 0;
  DBDataSrc = 0;
  RegWre = 1;
  InsMemRW = 1;
  RD = 1;
  WR = 1;
  RegDst = 1;
  ExtSel = 0;
  PCSrc = 2'b00;
  ALUOp = 3'b000;
end
```

即每一个指令的操作码对应一个case语句中的一种情况，而输出取决于上表“指令与控制信号的对应”。

下面举几个相对较复杂的指令，即beq，bne，bgtz指令进行介绍：

1）beq

```
6'b110000: //beq
begin
  PCWre = 1;
  ALUSrcA = 0;
  ALUSrcB = 0;
  DBDataSrc = 0;
  RegWre = 0;
  InsMemRW = 1;
  RD = 1;
  WR = 1;
  RegDst = 1;
  ExtSel = 1;
  if(1 == zero)
    PCSrc = 2'b01;
  else PCSrc = 2'b00;
  ALUOp = 3'b001;
  end
```

指令beq需要判断寄存器rs和rt的内容是否相同，所以使用ALU的减法功能，并用zero标志来判断是否满足条件，zero为1时PCSrc为2’b01，即下一个PC选择由beq指令的偏移量决定的值；反之则选择下一条指令的地址。

2）bne

```
6'b110001: //bne
begin
  PCWre = 1;
  ALUSrcA = 0;
  ALUSrcB = 0;
  DBDataSrc = 0;
  RegWre = 0;
  InsMemRW = 1;
  RD = 1;
  WR = 1;
  RegDst = 1;
  ExtSel = 1;
  if(0 == zero)
    PCSrc = 2'b01;
  else PCSrc = 2'b00;
  ALUOp = 3'b001;
end
```

指令bne与beq相似，只是将zero为不为零要输出的内容交换。

3）bgtz

```
6'b110010: //bgtz
begin
  PCWre = 1;
  ALUSrcA = 0;
  ALUSrcB = 0;
  DBDataSrc = 0;
  RegWre = 0;
  InsMemRW = 1;
  RD = 1;
  WR = 1;
  RegDst = 1;
  ExtSel = 1;
  if(0 == zero && 0 == sign)
    PCSrc = 2'b01;
  else PCSrc = 2'b00;
  ALUOp = 3'b001;
end
```

指令bgtz也是使用ALU的减法功能以及其zero和sign标志来判断条件是否满足。即当zero为0和sign为0，就是说结果不为零且是非负数，即正数时，满足条件，下一个PC取有这条指令的偏移量以及当前PC地址所决定的值；反之则选择下一条指令的地址。

至此，任务二完成。

在这一阶段的任务一和任务二中运用的数据冗余策略以及ControlUnit模块能够很好地统筹各个模块并对指令进行正确的解码。

任务三：

设计一个寄存器堆模块一管理寄存器堆的写入和读取。

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。

除此之外，还要根据指令的不同选择要写入的寄存器编号，这里只需要一个控制信号RegDst，这一部分代码如下：

```
assign Write_reg = (0 == RegDst)? rt:rd;
```

这样就完成了寄存器堆的读写操作。

3. **指令执行(EXE)阶段：**

在这一阶段主要使用的便是对在第二阶段以及得到的输入数据进行运算，于是需要设计一个32位的ALU（算术逻辑单元）模块。

ALU32模块

输入：ALU所需的两个操作数rega和regb，ALU功能选择码ALUopcode。

输出：零标志zero，符号标志sign，以及运算结果。

功能说明：根据ALU功能选择码，对两个操作数进行相应的运算，即执行该条指令所需的操作，并得到一个运算结果以及两个标志。

设计：主要代码如下：

```
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 = regb << rega;
    3'b011: result = rega | regb;
    3'b100: result = rega & regb;
    3'b101: result = (rega < regb)? 1:0;
    3'b110: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'b111: result = (~rega & regb) | (rega & ~regb);
    default: begin
      result = 0;
    end
    endcase
end
```

ALU功能选择码与其要实现的功能的关系见实验原理部分“ALU运算功能表”。

ALU中单个功能的实现较为简单，不予赘述。

在这一阶段中还涉及到的是ALU的输入选择问题，即不同的指令会有不同来源的数据输入到ALU中，这里需要用到两个控制信号：ALUSrcA，ALUSrcB。

代码实现如下：

```
assign InALU_A = (0 == ALUSrcA)? Read_data1:Ext_sa;
assign InALU_B = (0 == ALUSrcB)? Read_data2:Ext_immediate;
```

至此，阶段三的任务完成。

4. 存储器访问（MEM）阶段：

由于MIPS指令集的设计使整个指令集中涉及到存储器访问的只有两类指令：数据加载和数据存储指令，而在本实验中只有lw和sw指令。这两条指令都经过如下过程：计算数据地址 => 访问存储器，不同的只是lw指令取出数据，sw指令存入数据，于是需要；两个控制信号RD和WR来进行区分。

设计的RAM模块，该模块的数据读取为异步操作，而数据写入在时钟下降沿完成。

RAM模块

输入：RD、WR控制信号，时钟信号CLK，数据地址Daddr，写入的数据DataIn。

输出：输出的数据DataOut。

功能说明：控制数据存储部分。

设计：主要代码如下：

```
reg [7:0] memory [0:63];

assign DataOut[31:24] = (RD==1)? 8'bz: memory[Daddr];
assign DataOut[23:16] = (RD==1)? 8'bz:memory[Daddr+1];
assign DataOut[15:8] = (RD==1)? 8'bz:memory[Daddr+2];
assign DataOut[7:0] = (RD==1)? 8'bz:memory[Daddr+3];

always @(negedge CLK) begin
  if(0 == WR) begin
    memory[Daddr] <= DataIn[31:24];
    memory[Daddr+1] <= DataIn[23:16];
    memory[Daddr+2] <= DataIn[15:8];
    memory[Daddr+3] <= DataIn[7:0];
  end
end
```

代码原理相当简单，唯一值得一提的是采用大端方式，其余含义不言自明。

5. 结果写回（WB）阶段：

该阶段只需完成一个相对简单的任务，即将运算结果或者从存储器中获得的数据写回到目标寄存器中，即根据DBDataSrc控制信号选择要写回的结果然后将结果传递给寄存器堆模块。

代码如下：

```
assign DBData = (0 == DBDataSrc)? result:DataOutFromRAM;
```

接下来的工作是将各模块按照数据通路图相连。

至此，CPU设计完成。

验证设计的正确性

为了验证CPU设计的正确性，我编写了一段汇编程序，该汇编程序使用了全部要求设计以及我自己加入的指令。

如下图所示：（PC从0开始）

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	**000001**	**00000**	**00001**	**0000 0000 0000 1000**	=	04010008
0x00000004	ori $2,$0,2	010000	00000	00010	0000 0000 0000 0010	=	40020002
0x00000008	add $3,$2,$1	000000	00010	00001	0001 1000 0000 0000	=	00411800
0x0000000C	sub $5,$3,$2	000010	00011	00010	0010 1000 0000 0000	=	08622800
0x00000010	and $4,$5,$2	010001	00101	00010	0010 0000 0000 0000	=	44A22000
0x00000014	or $8,$4,$2	010010	00100	00010	0100 0000 0000 0000	=	48824000
0x00000018	sll $8,$8,1	011000	00000	01000	0100 0000 0100 0000	=	60084040
0x0000001C	**bne $8,$1,-2 (≠,转 18)**	110001	01000	00001	**1111 1111 1111 1110**	=	C501FFFE
0x00000020	slt $6,$2,$1	011100	00010	00001	0011 0000 0000 0000	=	70413000
0x00000024	slt $7,$6,$0	011100	00110	00000	0011 1000 0000 0000	=	70C03800
0x00000028	addi $7,$7,8	000001	00111	00111	0000 0000 0000 1000	=	04E70008
0x0000002C	**beq $7,$1,-2 (=,转 28)**	110000	00111	00001	1111 1111 1111 1110	=	C0E1FFFE
0x00000030	sw $2,4($1)	100110	00001	00010	0000 0000 0000 0100	=	98220004
0x00000034	lw $9,4($1)	100111	00001	01001	0000 0000 0000 0100	=	9C290004
0x00000038	**bgtz $9,1 (>0,转 40)**	110010	01001	00000	0000 0000 0000 0001	=	C9200001
0x0000003C	**halt**	111111	00000	00000	0000 0000 0000 0000	=	FC000000
0x00000040	**addi $9,$0,-1**	000001	00000	01001	1111 1111 1111 1111	=	0409FFFF
0x00000044	**Addi $10,$0,0x50**	000001	00000	01010	0000 0000 0101 0000	=	040A0050
0x00000048	**jr $10**	111001	01010	00000	0000 0000 0000 0000	=	E5400000
0x0000004C	**Addi $9,$9,3**	000001	01001	01001	0000 0000 0000 0011	=	05290003
0x00000050	**j 0x0000003C**	111000	00000	00000	0000 0000 0000 1111	=	E000000F

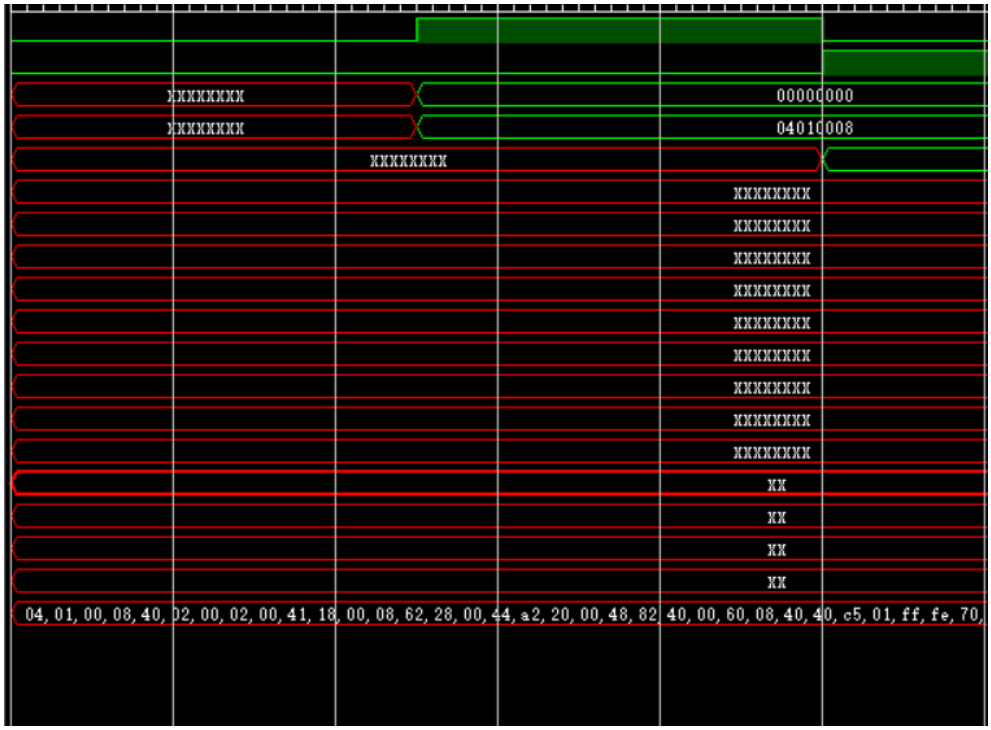
以下以波形图的形式展示每一条指令的运行情况，注意下面的波形图的波形的名称从上到下是：

CLK =» Reset =» PC =» Ins（当前指令）=» 1~10号寄存器的内容 =» 地址为12~15的存储器内容 =» 指令存储器中的内容。

Add $1,$0,8

指令执行前：$1 = 32’hX

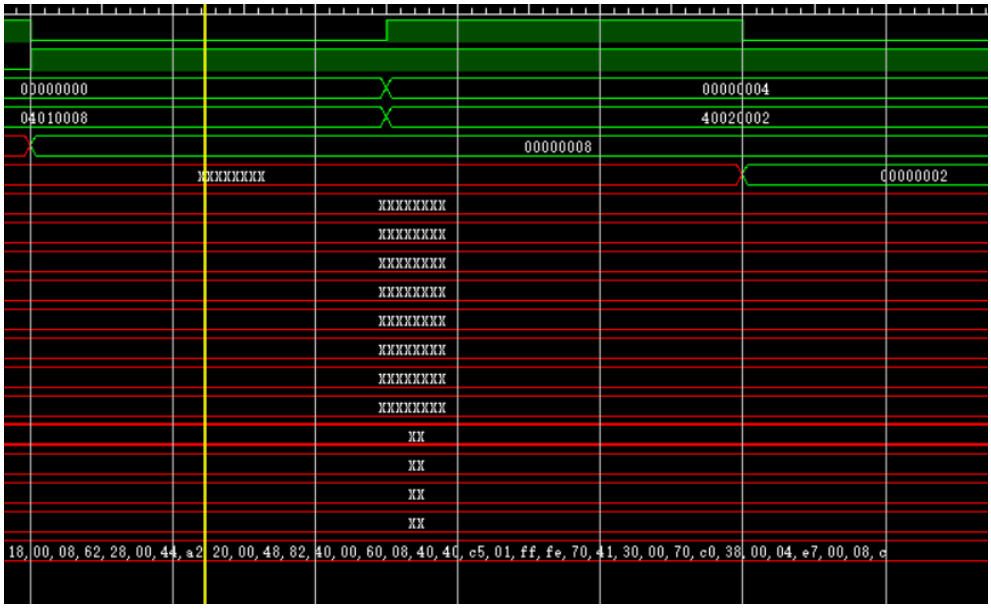
指令执行后：$1 = 32’h8



Ori $2,$0,2

指令执行前：$2 = 32’hX

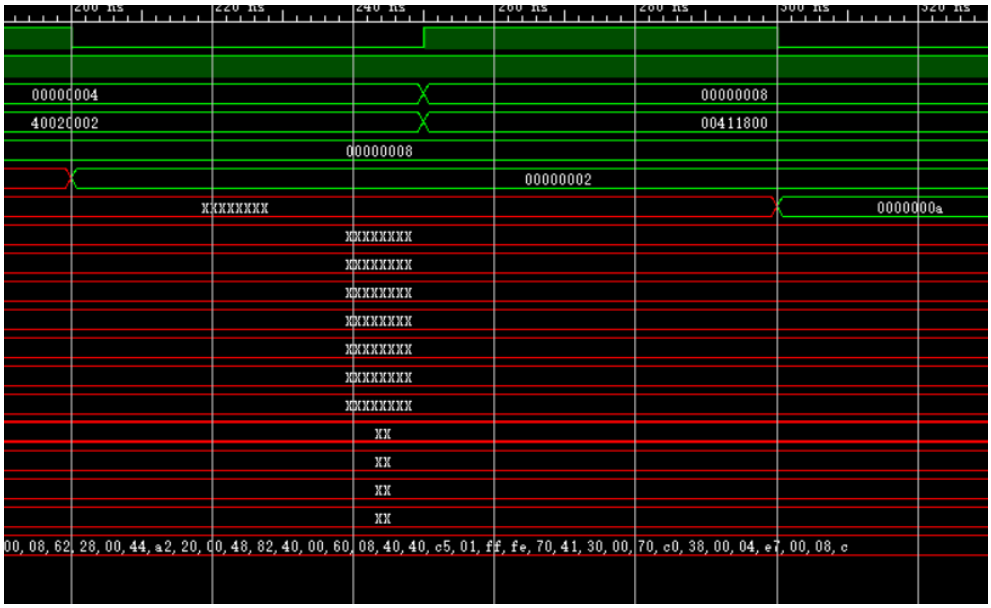
指令执行后：$2 = 32’h2



Add $3,$2,$1

指令执行前：$3 = 32’hX, $2 = 32’h2

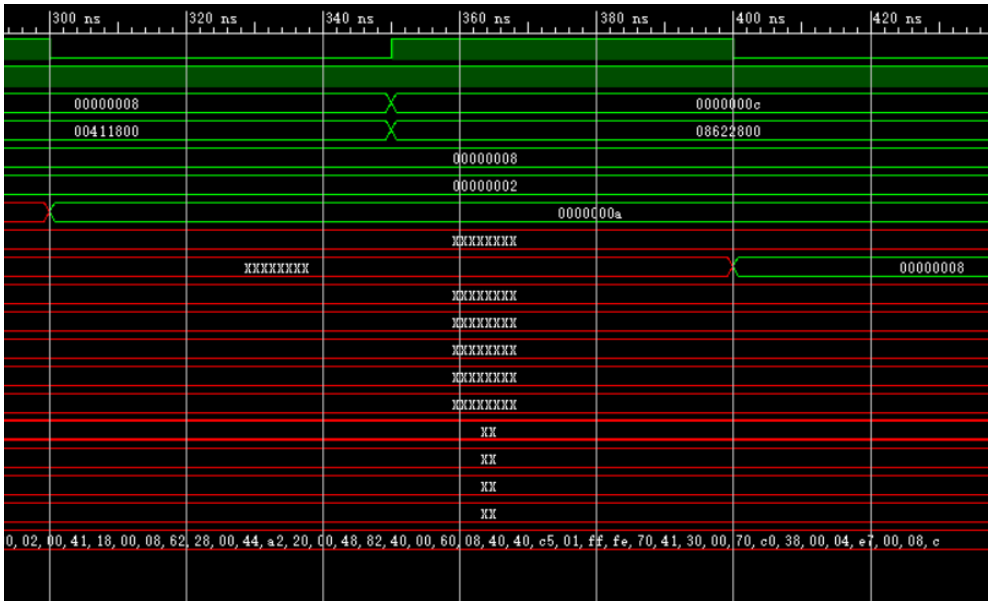
指令执行后：$3 = 32’ha



Sub $5,$3,$2

指令执行前：$5 = 32’hX, $3 = 32’ha, $2 = 32’h2

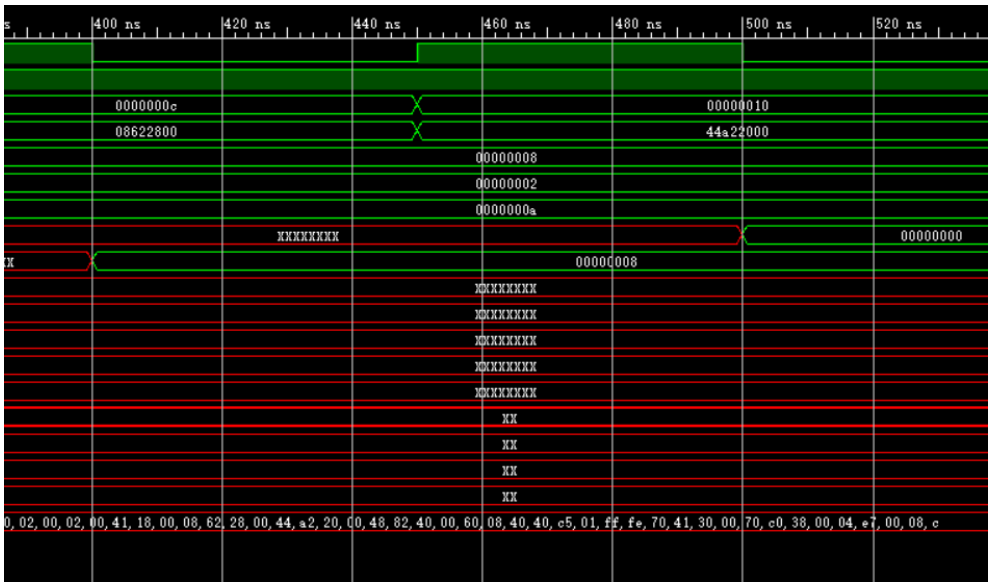
指令执行后：$5 = 32’h8



And $4,$5,$2

指令执行前：$4 = 32’hX, $5 = 32’h8, $2 = 32’h2

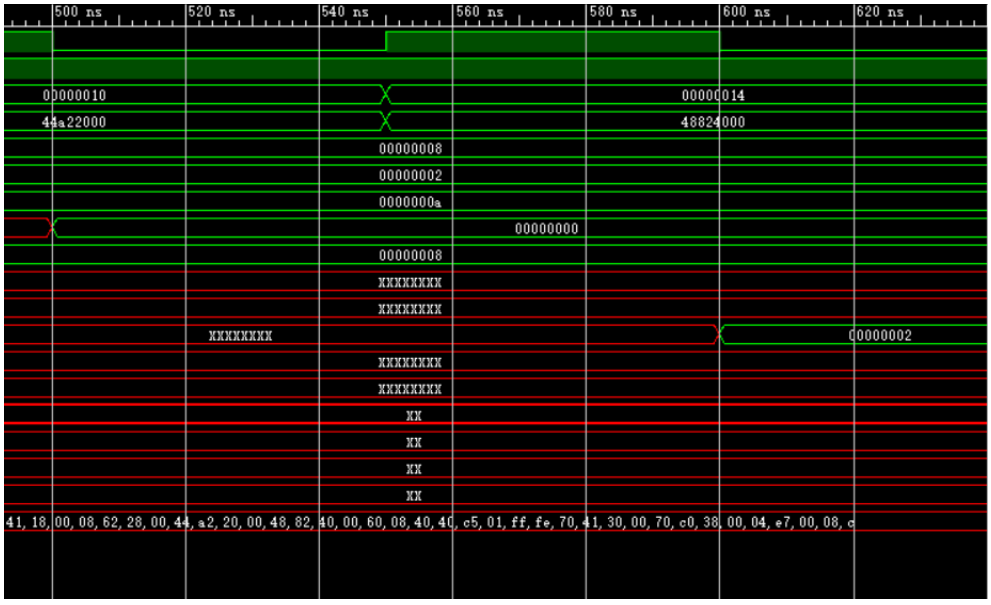
指令执行后：$4 = 32’h0



Or $8, $4, $2

指令执行前：$8 = 32’hX, $4 = 32’h0, $2 = 32’h2

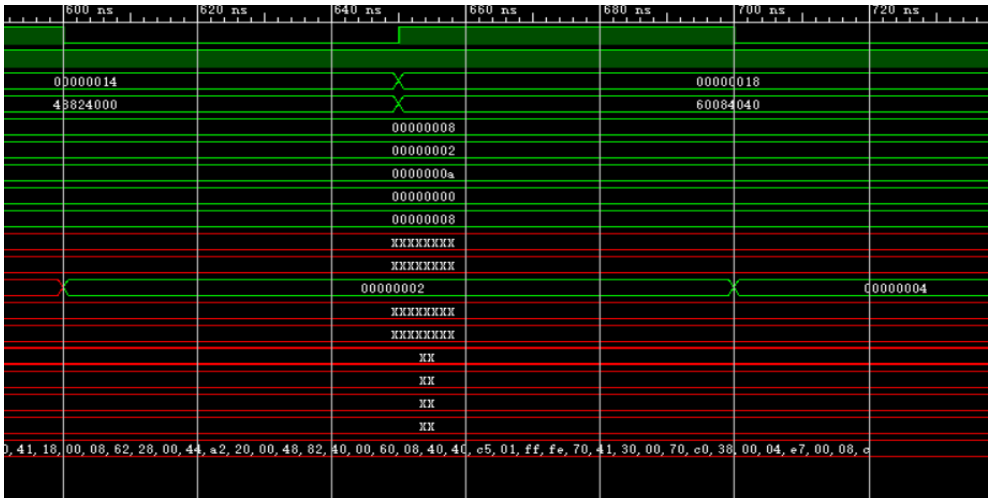
指令执行后：$8 = 32’h2



Sll $8,$8,1

指令执行前：$8 = 32’h2

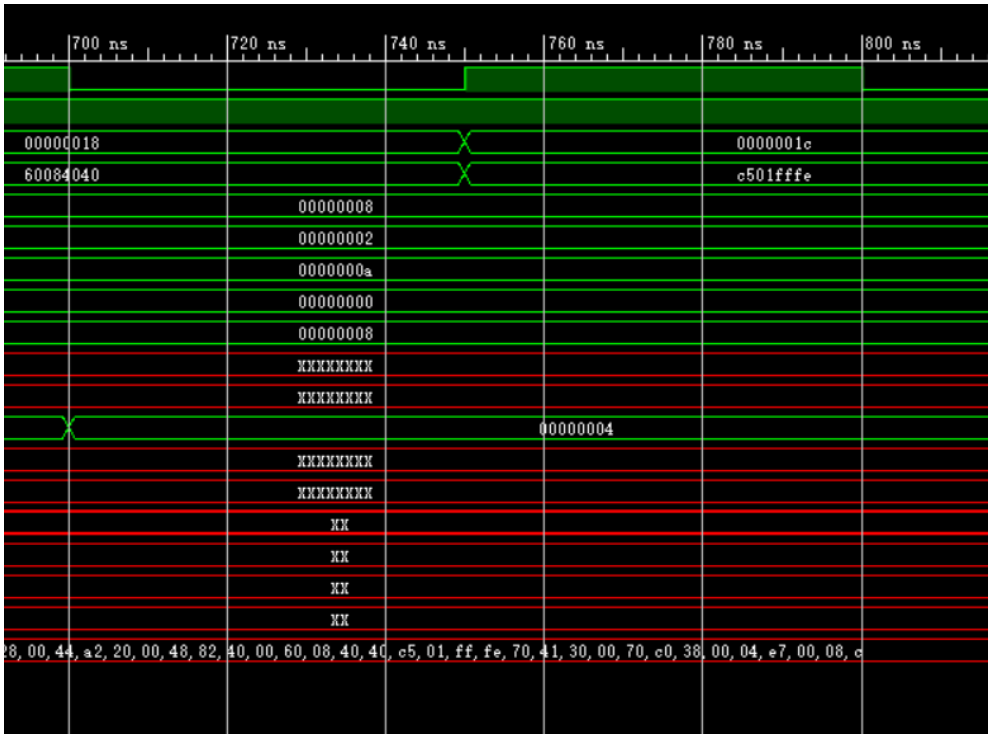
指令执行后：$8 = 32’h4



Bne $8,$1,-2

指令执行前：$8 = 32’h4, $1 = 32’h8, PC = 32’h1C

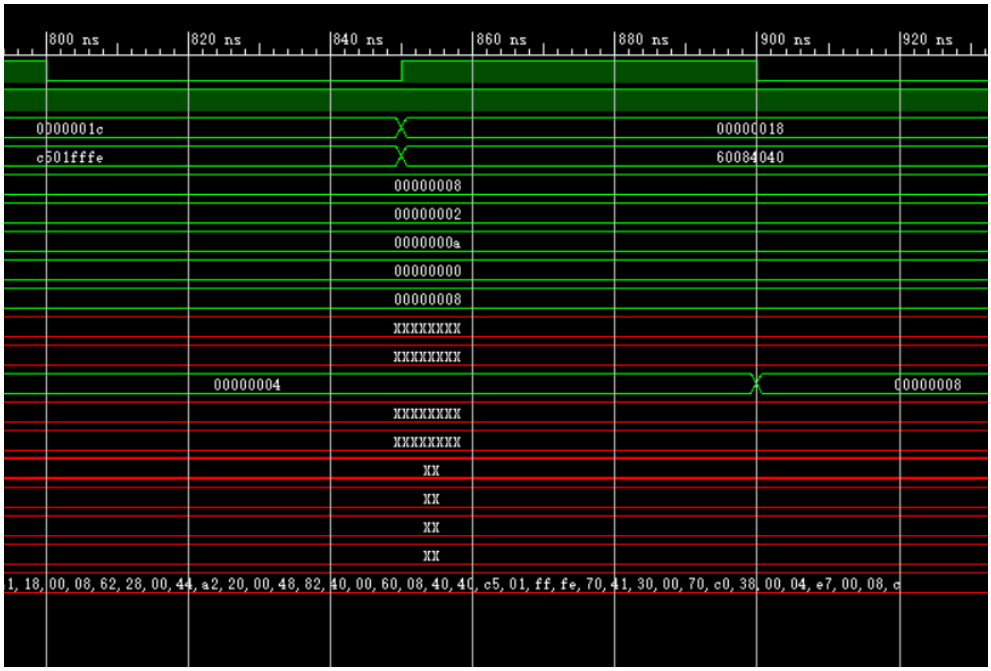
指令执行后：PC = 32’h18



Sll $8,$8,1

指令执行前：$8 = 32’h4

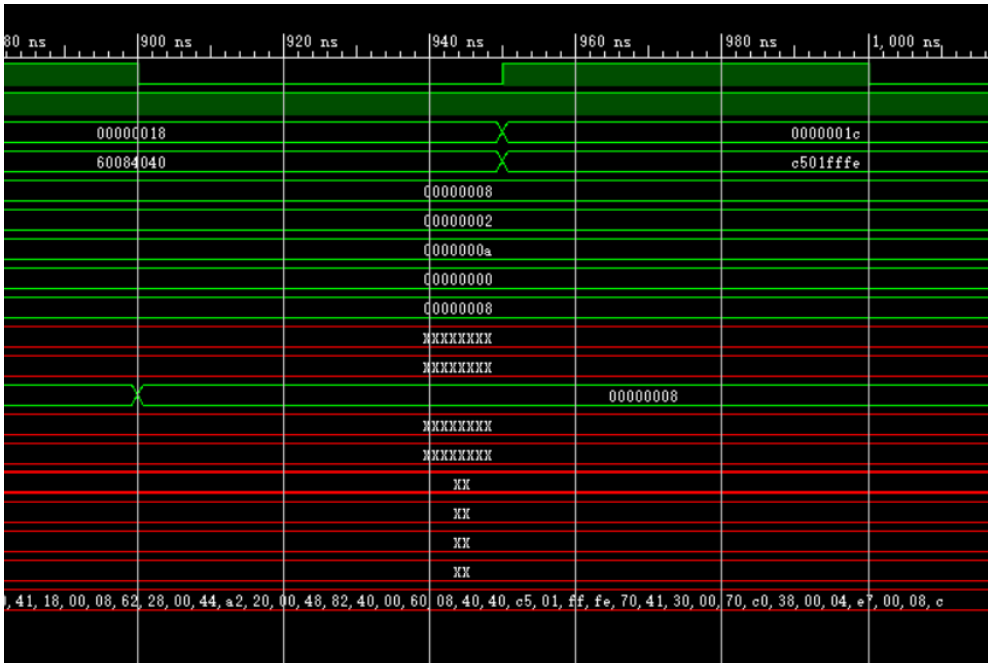
指令执行后：$8 = 32’h8



Bne $8,$1,-2

指令执行前：$8 = 32’h8, $1 = 32’h8, PC = 32’h1C

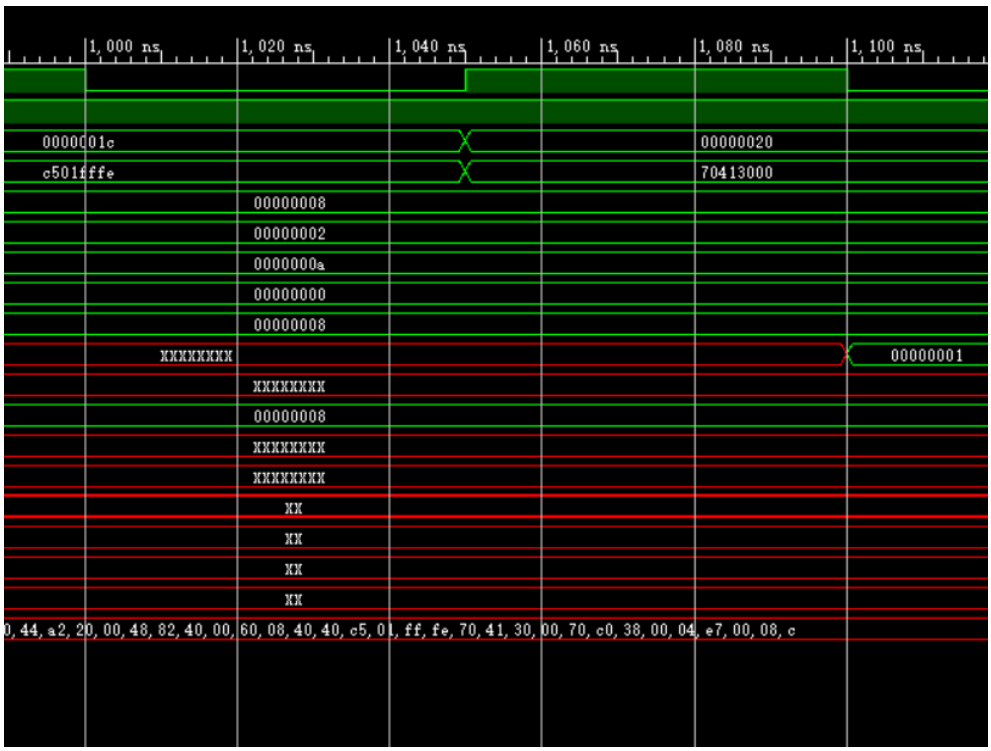
指令执行后：PC = 32’h18



Slt $6,$2,$1

指令执行前：$6 = 32’hX, $2 = 32’h2, $1 = 32’h8

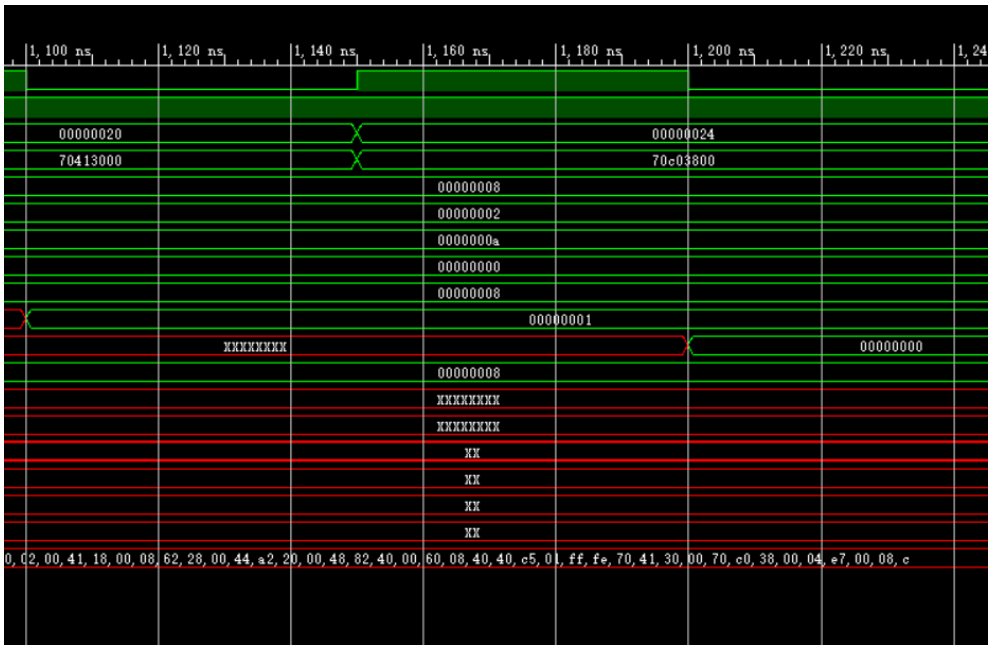
指令执行后：$6 = 32’h1



Slt $7,$6,$0

指令执行前：$7 = 32’h X, $6 = 32’h1

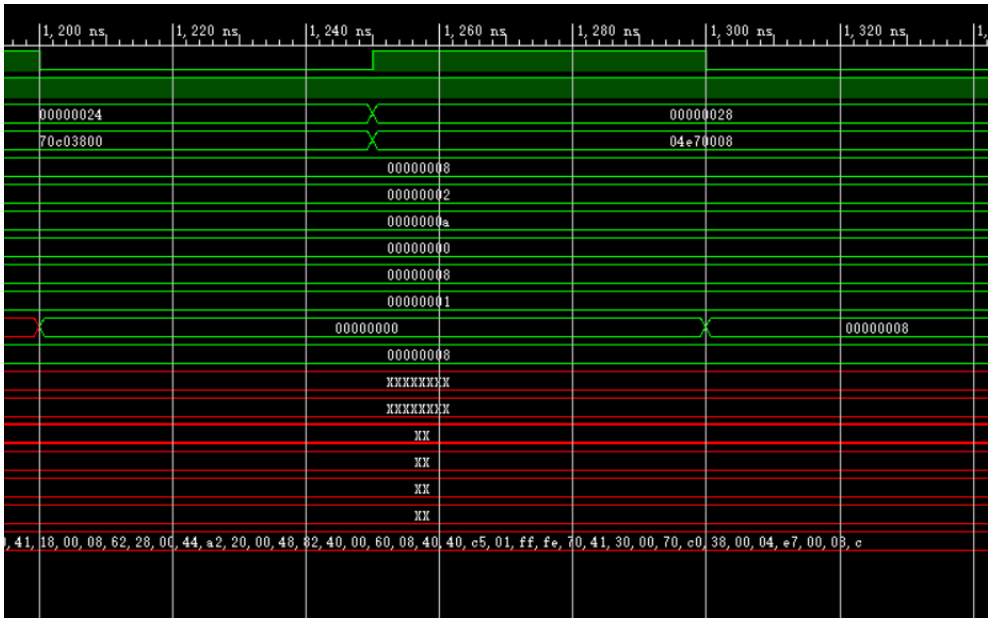
指令执行后：$7 = 32’h0



Addi $7,$7,8

指令执行前：$7 = 32’h0

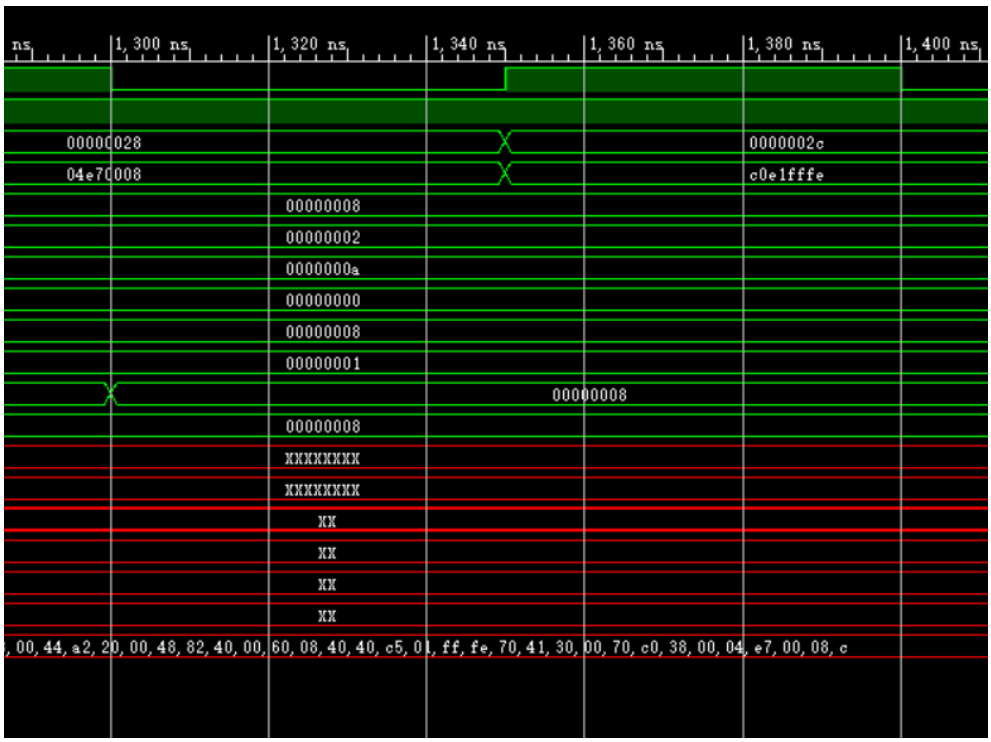
指令执行后：$7 = 32’h8



Beq $7,$1,-2

指令执行前：$7 = 32’h8, $1 = = 32’h8, PC = 32’h2C

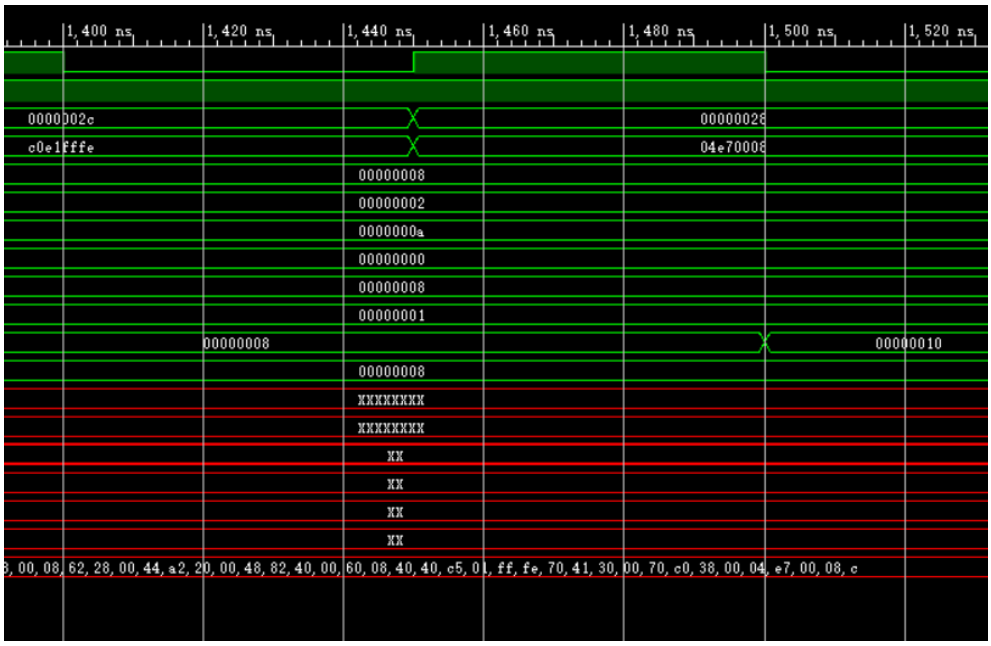
指令执行后：PC = 32’h28



Addi $7,$7,8

指令执行前：$7 = 32’h8

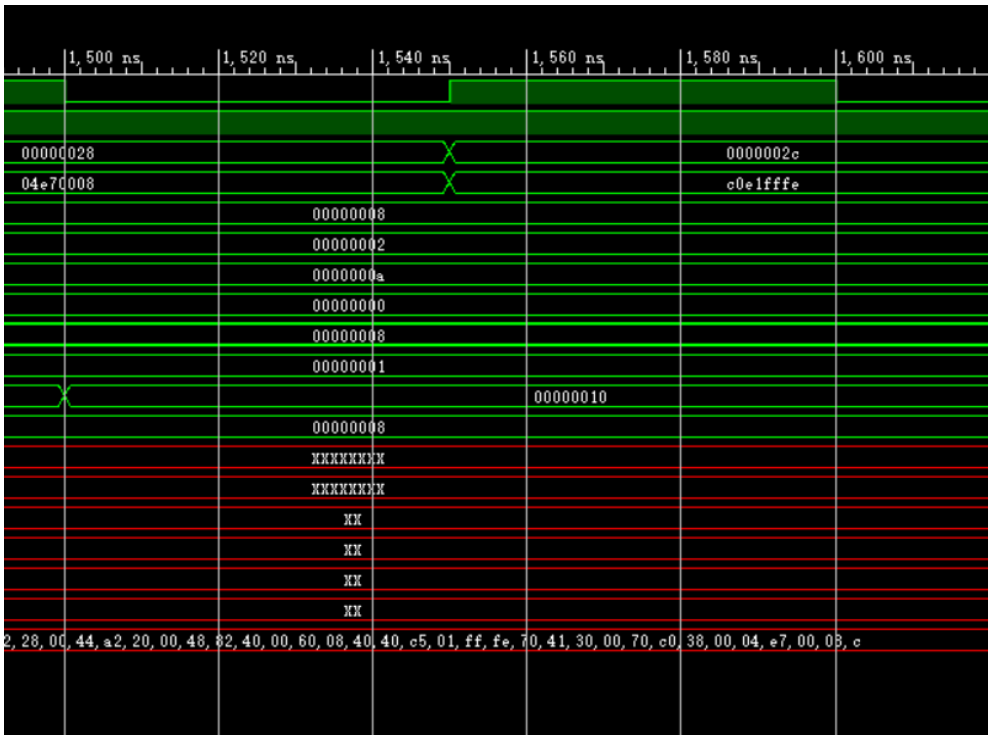
指令执行后：$7 = 32’h10



Beq $7,$1,-2

指令执行前：$7 = 32’h10, $1 = 32’h8, PC = 32’h2C

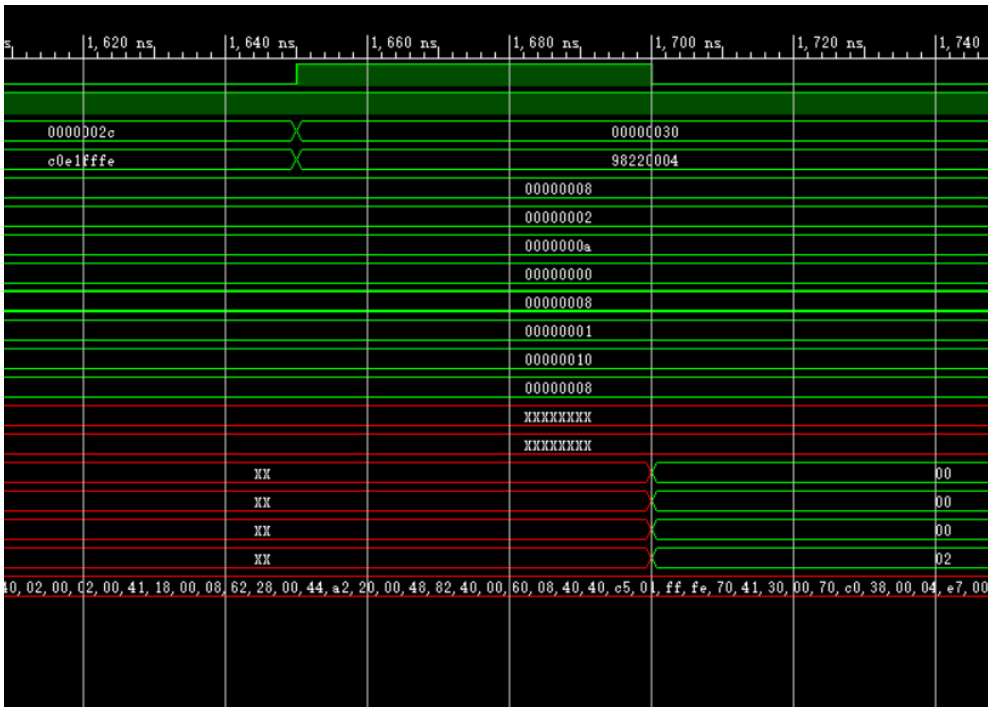
指令执行后：PC = 32’h30



Sw $2,4($1)

指令执行前：$2 = 32’h2, 4($1) = 32’hX

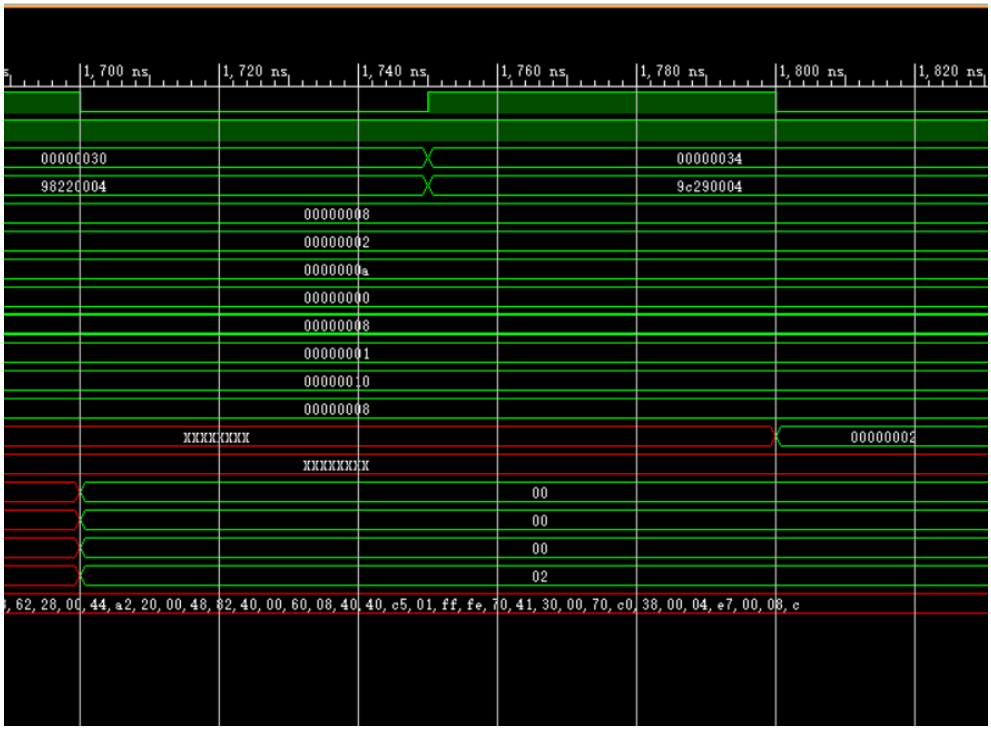
指令执行后：4($1) = 32’h2(大端方式)



Lw $9, 4($1)

指令执行前：4($1) = 32’h2(大端方式), $9 = 32’hX

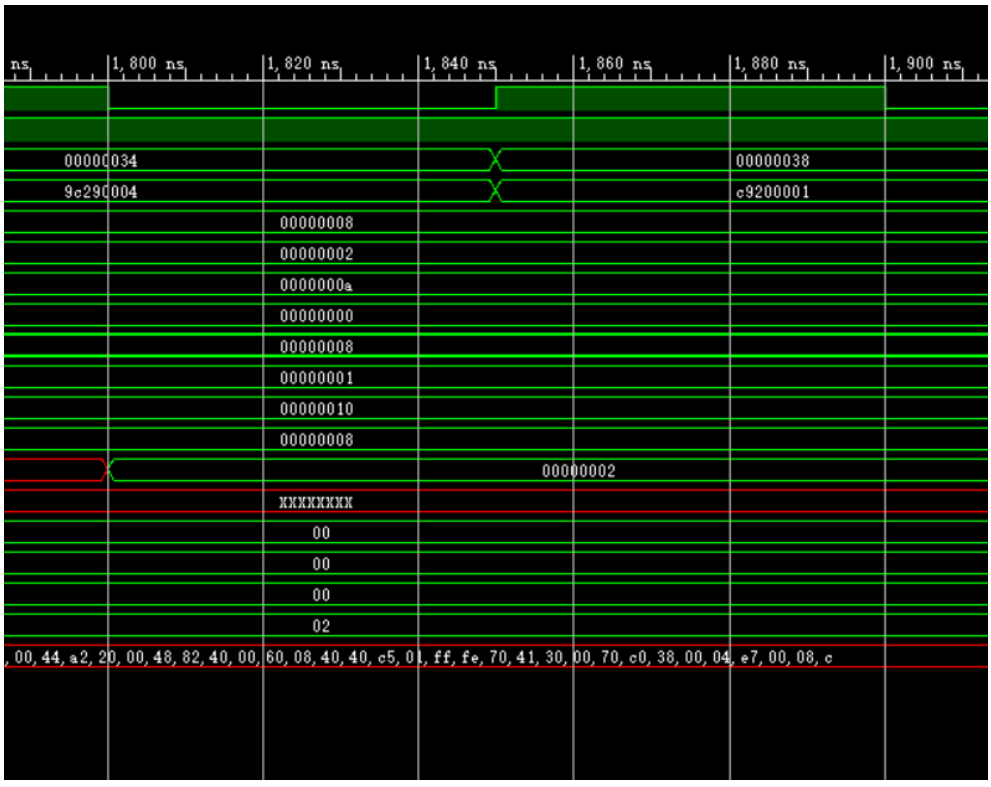
指令执行后：$9 = 32’h2



Bgtz $9,1

指令执行前：$9 = 32’h2, PC = 32’h38

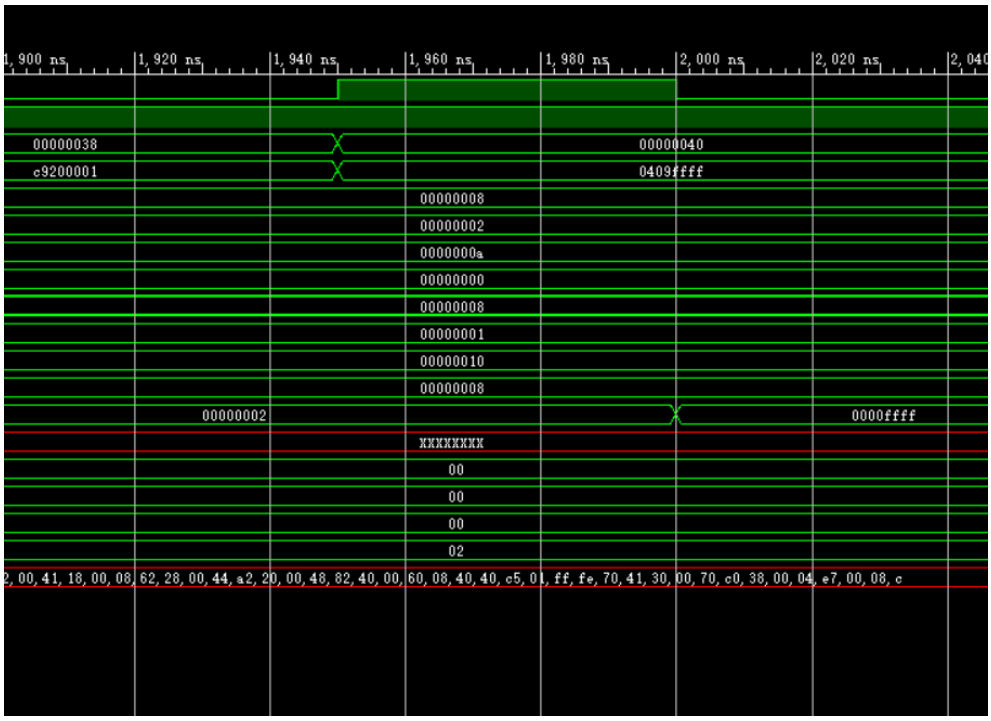
指令执行后：PC = 32’h40



Addi $9,$0,-1

指令执行前：$9 = 32’h2

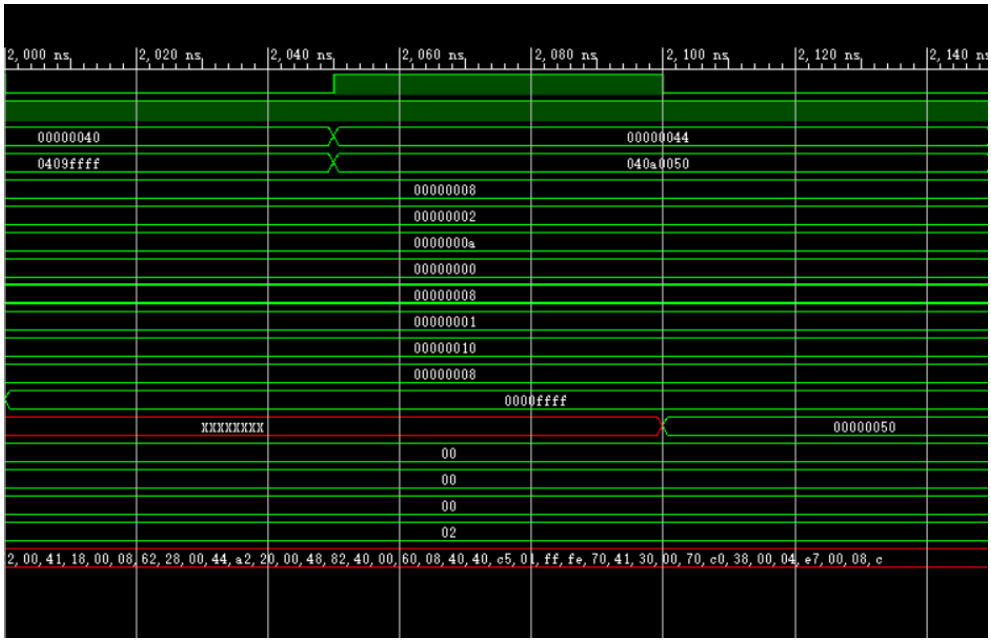
指令执行后：$9 = 32’hffff_ffff



Addi $10,$0,0x50

指令执行前：$10 = 32’h0

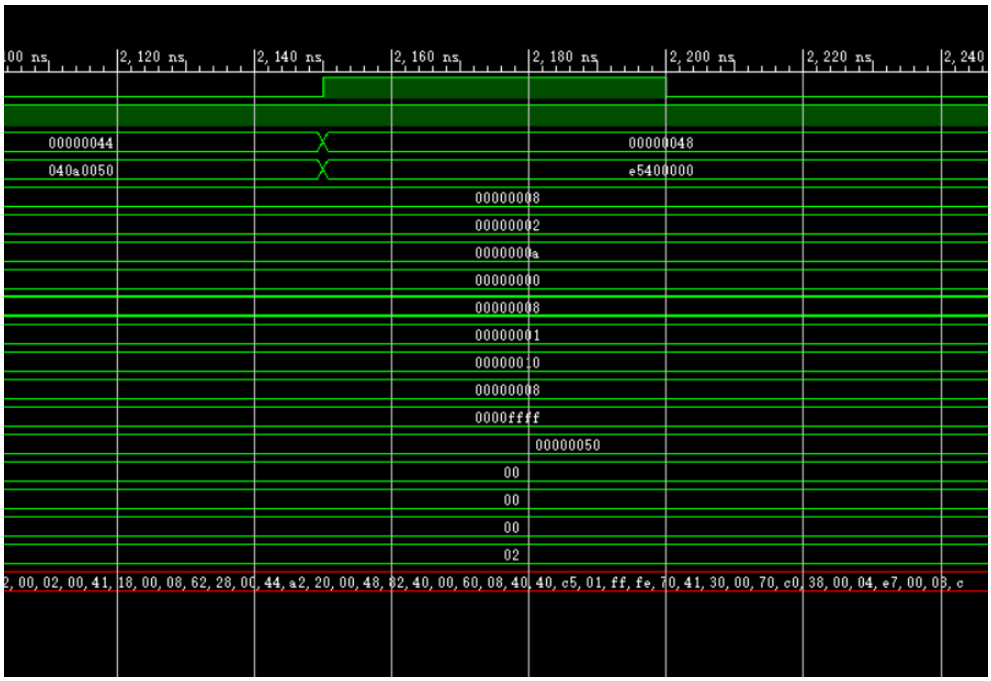
指令执行后：$10 = 32’h50



Jr $10

指令执行前：$10 = 32’h50,PC = 32’h48

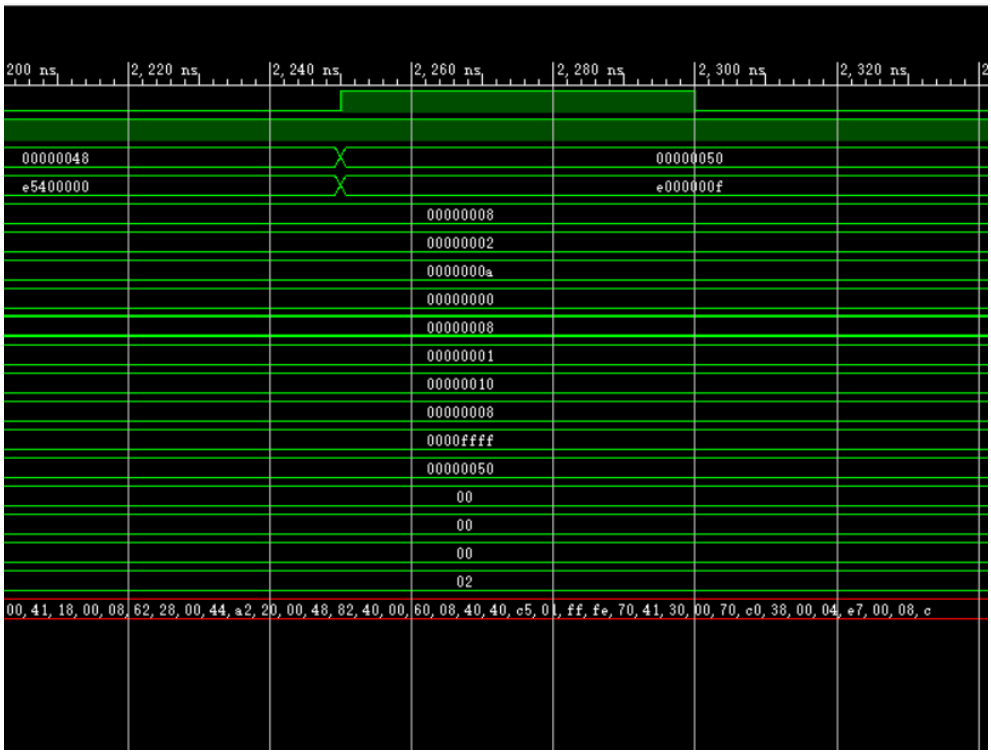
指令执行后：PC = 32’h50



J 0x0000003C

指令执行前：PC = 32’h50

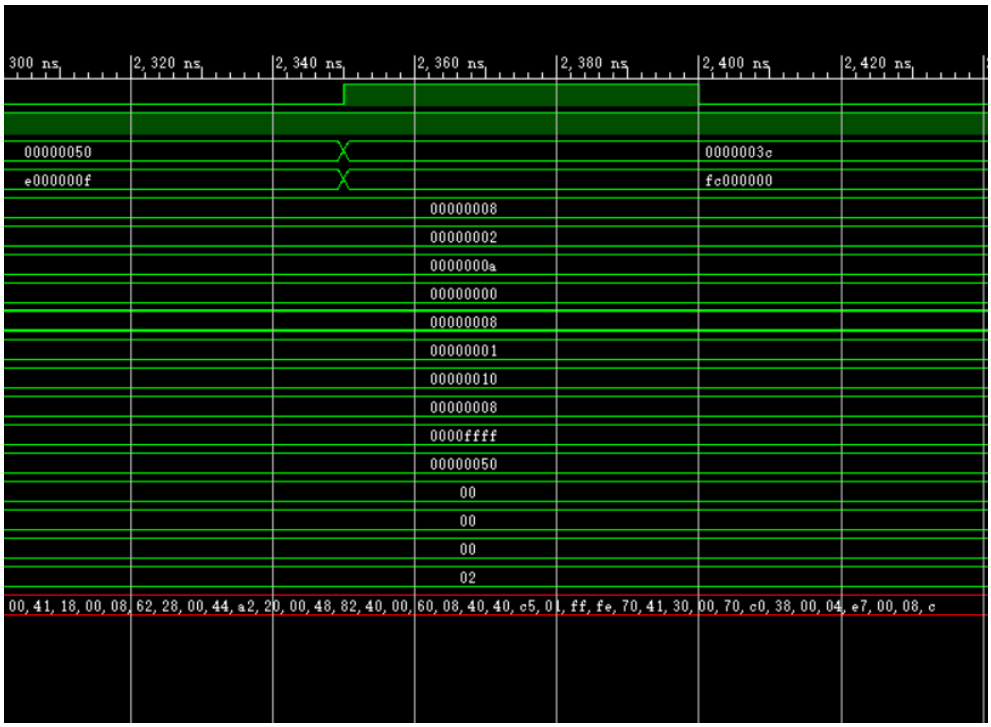
指令执行后：PC = 32’h3C



halt

指令执行前：PC = 32’h3C

指令执行后：PC = 32’h3C



此后PC的值一直维持在0x0000003C。

至此验证了CPU设计的准确性。

实现：

a. 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）

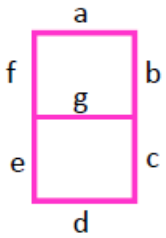
Clkdiv模块

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

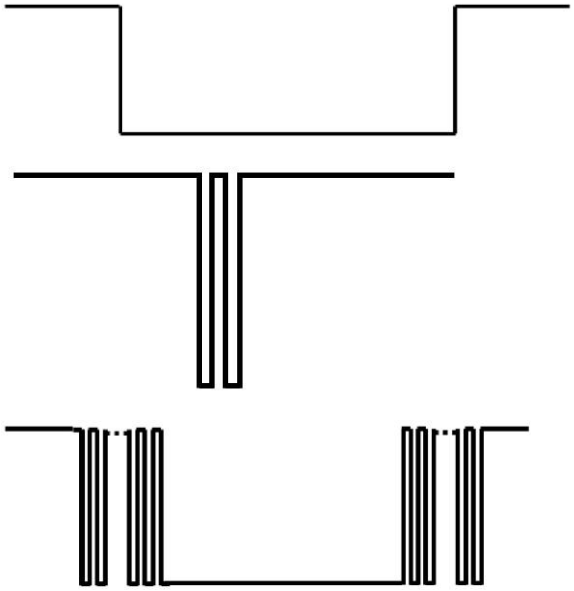
输入：扫描时钟，Reset控制信号，要显示的两个数

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：

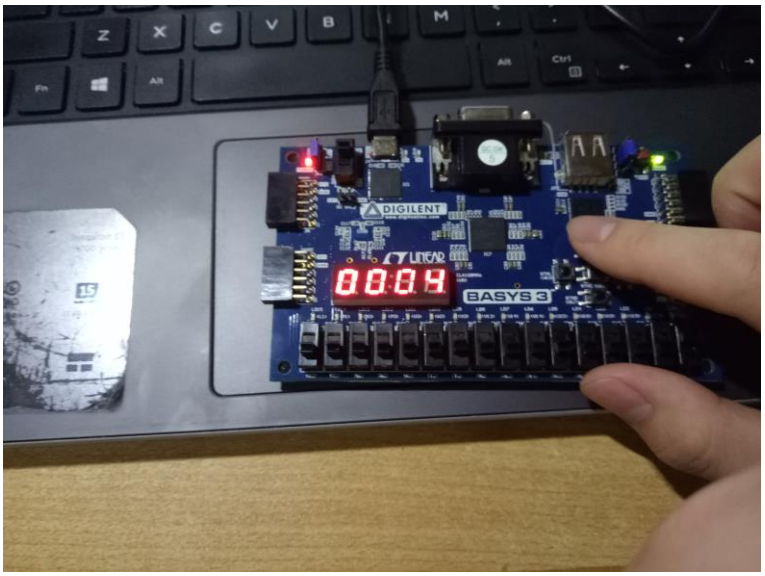
```
reg [19:0] fifo;
reg [1:0] key_in_r;
wire change;

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
```

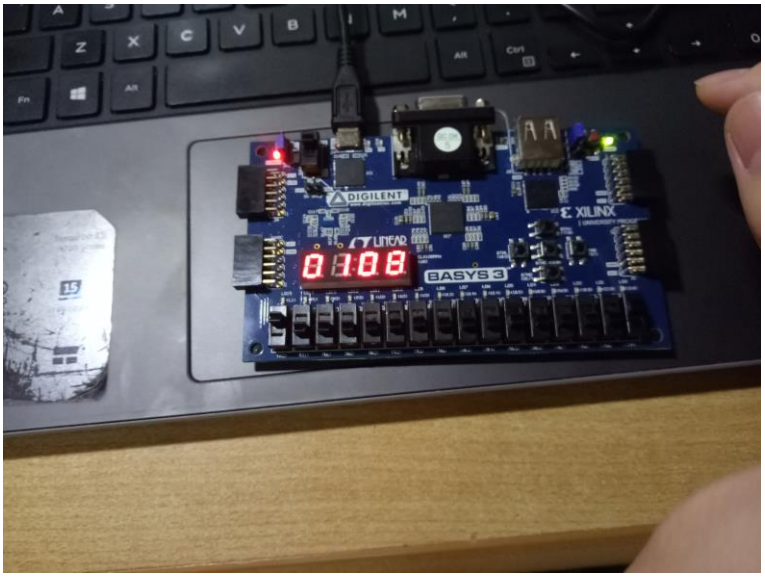
b. 结果展示

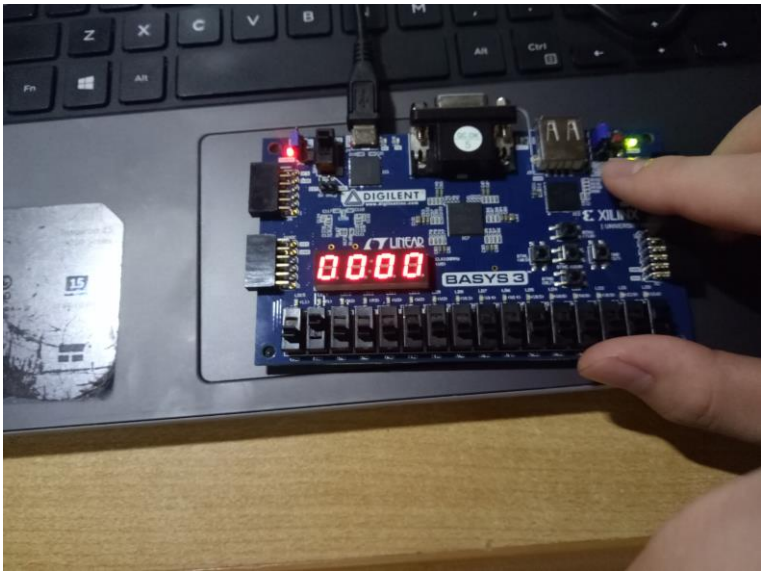
由于每一条指令有四种输出显示结果，又因为仿真步骤已经确认了内部逻辑的正确性，于是在只展示少许指令，其中大部分指令只展示每一条 PC 和下一个 PC 的输出，少数指令展示全部输出显示以证实显示模块的正确性。

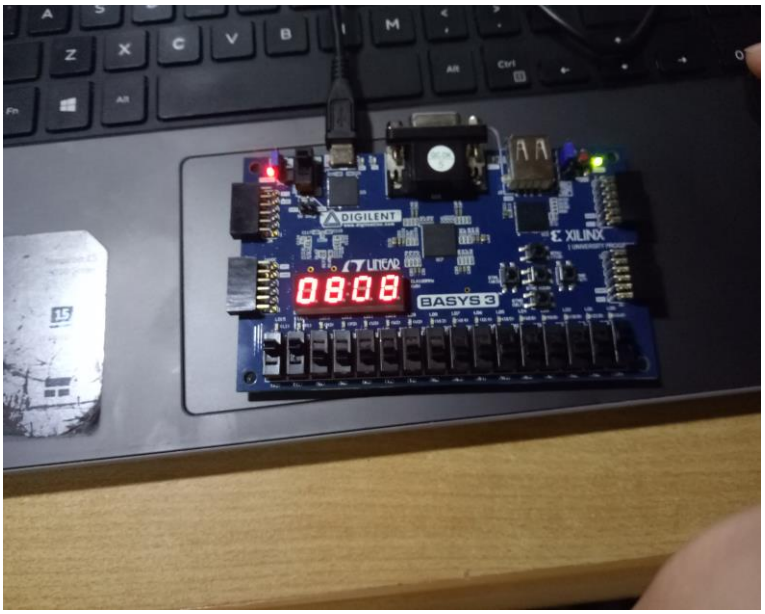
从上往下为
PC:下一条 PC
RS 编号:RS 内容
RT 编号:RT 内容
ALU 输出:写回的数据

第一条指令：Addi $1,$0,8

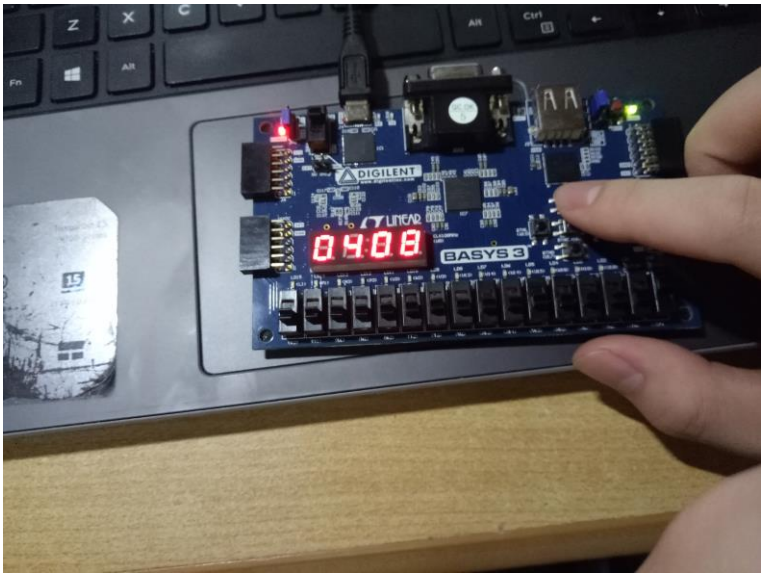

DIGILENT
0004
BASYS 3

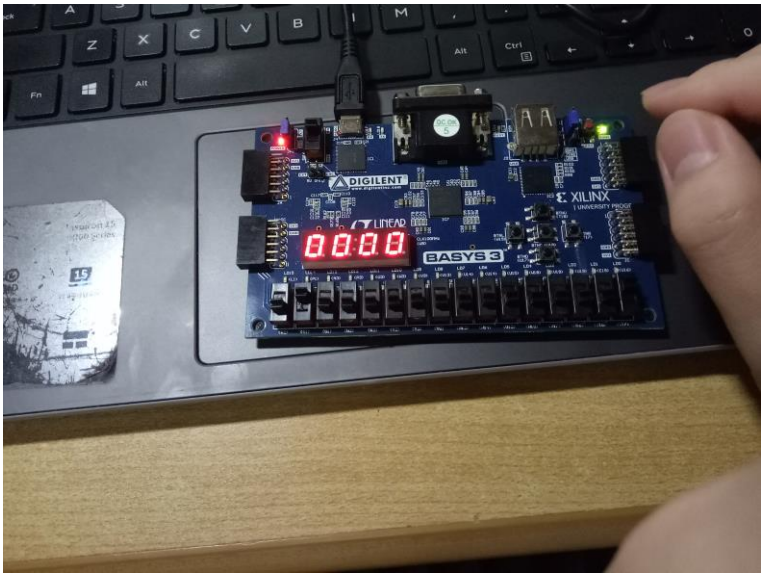

DIGILENT
0108
BASYS 3

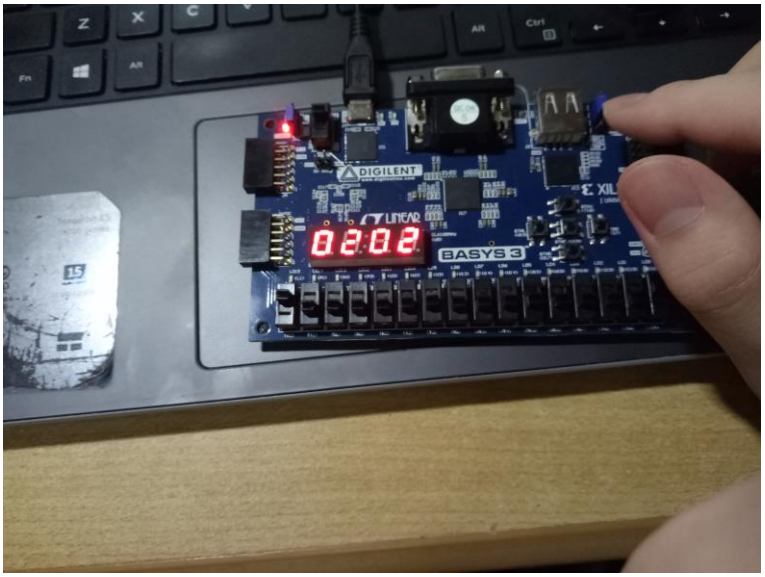

DIGILENT
0000
BASYS 3



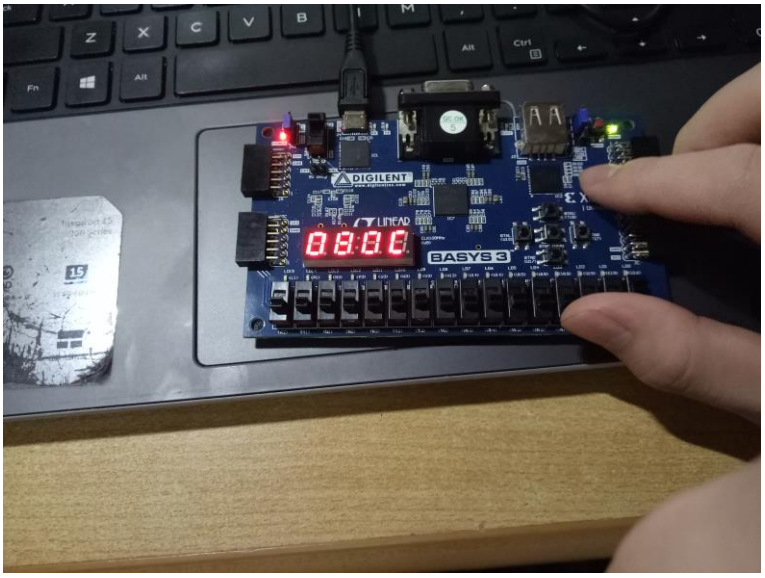
第二条指令：ori　$2,$0,2

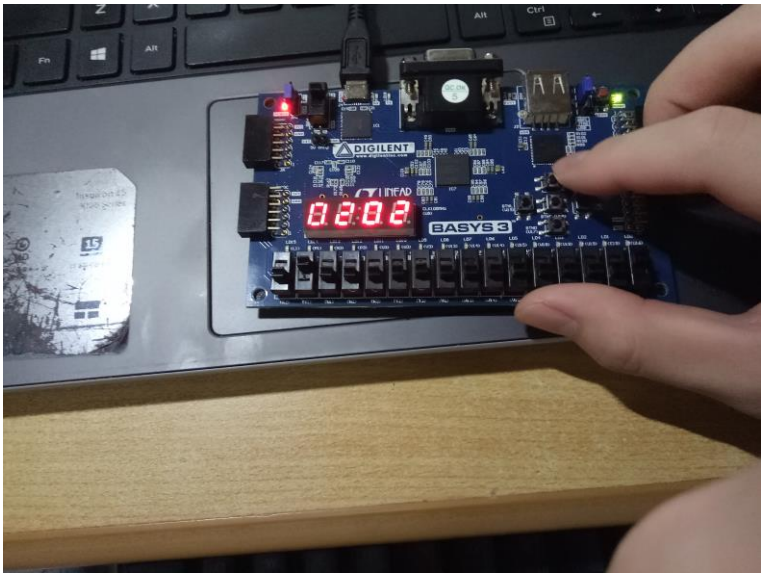


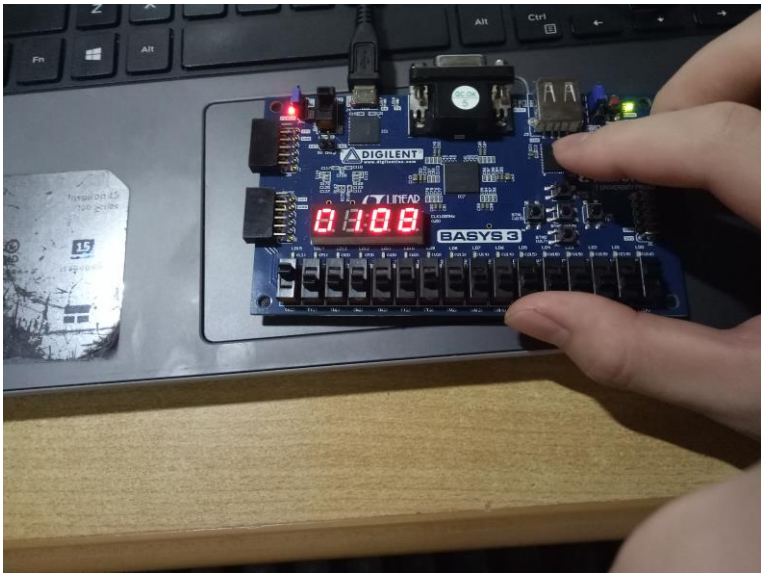


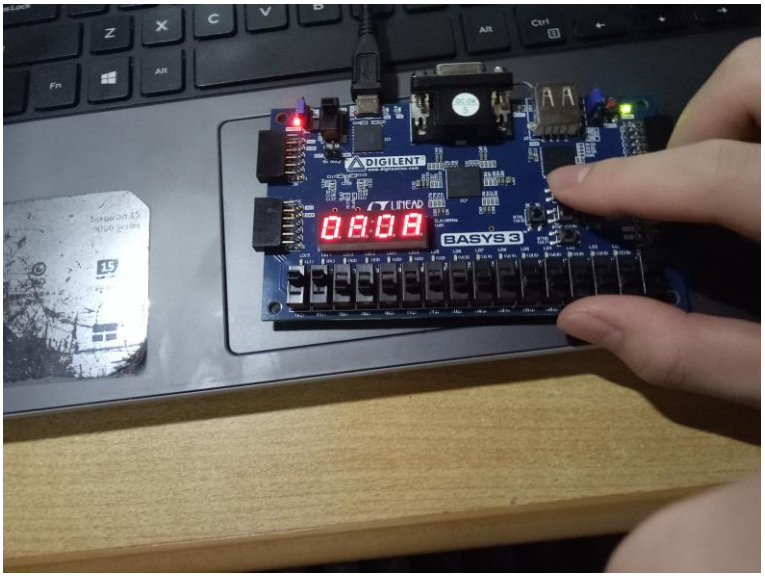


第三条指令：add　$3,$2,$1

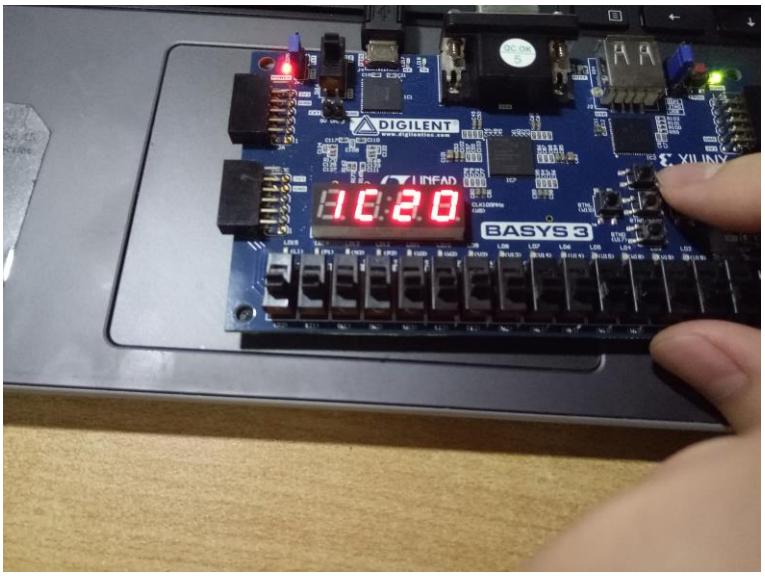

DIGILENT
BASYS 3


DIGILENT
BASYS 3

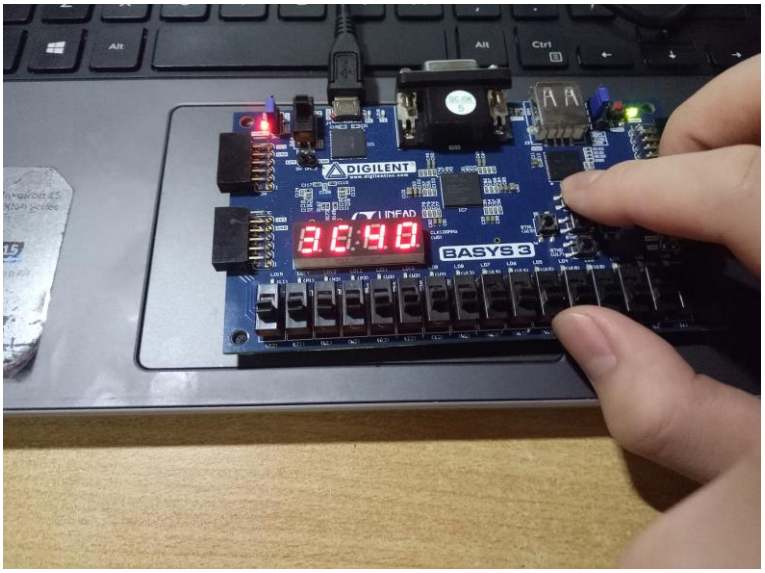

DIGILENT
BASYS 3



PC = 32’h1C时的**bne $8,$1,-2**



停机指令halt：



六. 实验心得

实验未开始时的思考：

刚刚开始进行实验的时候我考虑的更多是如何将上面“实验原理”部分的数据通路实现出来，即如何将每个模块的输入输出相连。但是没有去了解 MIPS 架构的 CPU 的内部原理，导致在实现 CPU 的过程中写出了许多错误，而且这些错误是来自于概念的混淆，比如时序逻辑与组合逻辑的矛盾。导致了大量时间的浪费。这个教训又强调了一遍先思考、先理清楚原理再动手的重要性。

实验进行时得到的收获：

1) **对于 CPU 本身和指令集：**

a) 寄存器和存储器的读取是异步的，不需要时钟信号。一开始我设计的寄存器和存储器的数据读取和数据写入都是在时钟下降沿触发的，这样导致的问题就是当时钟下降沿到来时 ControlUnit 模块已经读取了下一条指令的操作码，导致数据与指令的不匹配。

b) 没有注意到时序逻辑和组合逻辑的矛盾，比如在某个模块内使用可能在模块外部被更改的输入变量却没有使用锁存器将它的值锁定。这个问题被 vivado 软件自动修复了，在本次实验中并没有造成多大的问题，甚至没有造成问题，只是导致了一个警告，但是在日后更复杂的设计中是一定要注意到的。

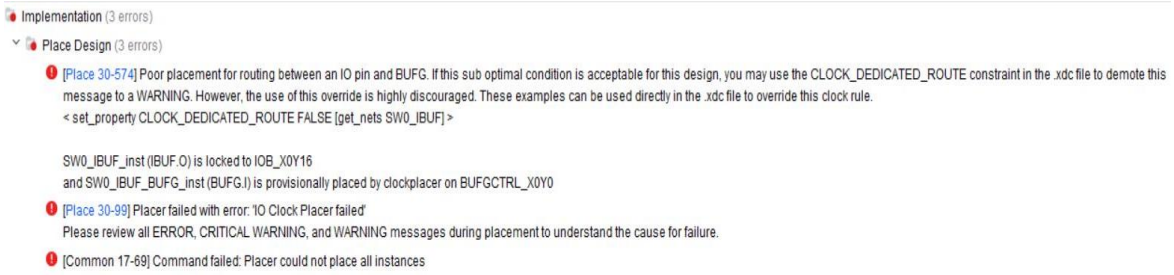
c) J 指令和分支指令（比如 beq，bne，bgtz 等等）涉及到的地址计算是不一样的，J 指令的 address 信号指的是地址，而分支指令的 immediate 指的是相对当前指令的下一条指令的偏移量。

除此之外，J 指令要跳转到的地址还需要向右移两位再写入到该条指令代码里面。

d) 对于移位指令，我忘记了要将 sa 信号写入指令代码中。

2) **对于实现、工具：**

a) 实现消抖这一过程对我而言是有巨大收获的。在这个实验中我初步掌握了延时消抖这个方法的原理和实现。但是这个收获还不是最大的，更大的收获是知道了以下这个常常出现的错误的原因：



（也就是 Place Design Error）

使用上面建议解决这个错误（在约束条件中加入< set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets SW0_IBUF] >一句）其实并不代表了解了这一错误发生的内在原因，该错误发生的内在原因是我们试图使用一个未经消抖的时钟信号来充当时钟信号，而 vivado 试图给该时钟信号添加一个消抖的过程但是失败了。

b) Readmemb 系统任务的调用学习。该任务并非像表面那么简单。下面为了叙述方便将要读取的文件记为 file，数据存储所在的 reg 类型变量记为 mem。如果在将 file 读取到 mem 之后对 mem 进行了修改，那么 vivado 会将这个 readmemb 指令忽略，导致没有指令在指令存储器中。

还有就是如何用相对地址获取指令所在的文件。老师所给的 ppt 中要求文件的地址要用绝对地址，然而我感觉用相对地址会更加方便，所以使用了相对地址，举个例子，如果指令文件为 rom.txt，且放在了项目文件夹下，那么由于该指令文件相对于项目源文件的位置为../../../rom.txt。

c) 还有感想的就是如何处理大量的变量名。我的解决方法就是在要用到的时候再声明这样一个变量，但是我和同学讨论之后发现有一个较好的方法，就是声明一个长度为各个变量总和的变量，然后对这个变量的每一个位记为一个单独的变量，以减少代码长度和难于理解。

d) 在这个实验中我多次忘记要将某些变量写入 always 过程的敏感变量列表中。

e) 还有一个问题就是会忘记重新仿真，然后纠结于一个已经修改过的地方。

f) 还有就是在这个实验中消耗了我最长时间的错误。

当我完成了仿真，进入将 CPU 实现到 basys 3 实验板的过程中时，发现第一条指令 addi $1,$0,8 在 PC 刚刚初始化时就以及运行到了写回阶段，等待时钟下降沿的到达，但是当我按下按键生成时钟信号时，又更换了一条指令，所以$1 的值并没有被成功写入。

在经过了一天检测了各个模块终于发现了这个问题之后，我想到了一个解决方案，初始化向各个模块传递的时钟信号为 1，这样就能在 PC 初始化只有产生一个时钟下降沿。

g) 学会了使用 xilinx 的官方手册。

h) 还有比较迷惑的一点是在将程序烧写到实验板的时候，不知道处于什么原因，需要先将电源关闭在开启然后重置实验板才能成功启动实验板。

3) 对实验的改进：

a) Jr 指令是没有包含在本实验本身的要求之中的，但是这条指令产生 PC 的方式是和

其他指令都不一样的，于是我将其加入了要实现的指令中并更新了数据通路图，且对这条指令进行了测试。