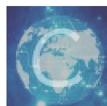
计算机组成与设计实验二：单周期CPU设计

原创

wu-kan 于 2018-11-23 18:40:49 发布 11284 收藏 108

分类专栏： 计算机组成与原理 文章标签： verilog cpu 单周期 中山大学 计算机组成与设计

本文及后续更新已转到https://wu-kan.cn 本文为博主原创文章，转载需要注明来源。
本文链接：https://blog.csdn.net/w_weilan/article/details/84349845
版权


计算机组成与原理 专栏收录该内容

3 篇文章 0 订阅
订阅专栏

> 此为中山大学17级计算机组成与设计实验课题。为不影响老师的教学，本文已经删去所有实现代码，完整含代码版本将在本学期结束时发布在我的个人博客https://wu-kan.cn/_posts/2018-11-23-单周期CPU设计/

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。

实验内容

设计一个单周期CPU，该CPU至少能实现以下指令功能操作，指令与格式如下。

算术运算指令

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”。

sub rd rs rt

000001	rs(5位)	rt(5位)	rd(5位)	reserved

功能：rd←rs - rt。

addiu rt rs immediate

000010	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。

逻辑运算指令

andi 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“与”运算。

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。

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。

or rd rs rt

010011	rs(5位)	rt(5位)	rd(5位)	reserved

功能：rd←rs | rt；逻辑或运算。

移位指令

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

slti rt rs immediate

011100	rs(5位)	rt(5位)	immediate(16位)

功能：if (rs< (sign-extend)immediate) rt =1 else rt=0, 带符号比较，详见ALU运算功能表。

存储器读/写指令

sw rt immediate(rs)

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。

lw rt immediate(rs)

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

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”。

bne rs rt immediate

110001	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：与beq不同点是，不等时转移，相等时顺序执行。

bltz rs immediate

110010	rs(5位)	00000	immediate(16位)

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

跳转指令

j addr

111000	addr[27:2]

功能：pc <一{(pc+4)[31:28],addr[27:2],2’b00}，无条件跳转。
说明：由于MIPS32的指令代码长度占4个字节，所以指令地址二进制数最低2位均为0，将指令地址放进指令代码中时，可省掉！这样，除了最高6位操作码外，还有26位可用于存放地址，事实上，可存放28位地址，剩下最高4位由pc+4最高4位拼接上。

停机指令

halt

111111	00000000000000000000000000(26位)

功能：停机；不改变PC的值，PC保持不变。

实验原理

单周期CPU指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期（如果晶振的输出没有经过分频就直接作为CPU的工作时钟，则时钟周期就等于振荡周期。若振荡周期经二分频后形成时钟脉冲信号作为CPU的工作时钟，这样，时钟周期就是振荡周期的两倍）。

CPU在处理指令的几个步骤

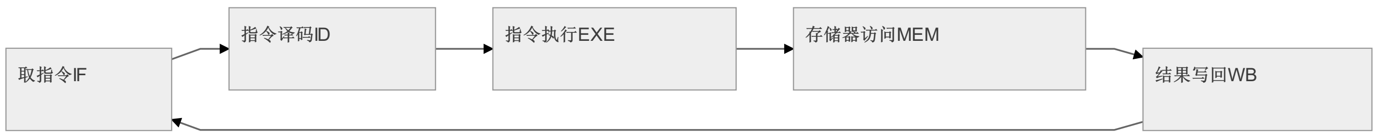


图1 CPU指令处理过程

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。

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。

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。

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。

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。

MIPS指令的三种格式

缩写	说明
op	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	位移量（shift amt），移位指令用于指定移多少位
funct	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	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（Laod）/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；

缩写	说明
address	地址

R类型

31-26	25-21	20-16	15-11	10-6	5-0
op	rs	rt	rd	sa	func
6位	5位	5位	5位	5位	6位

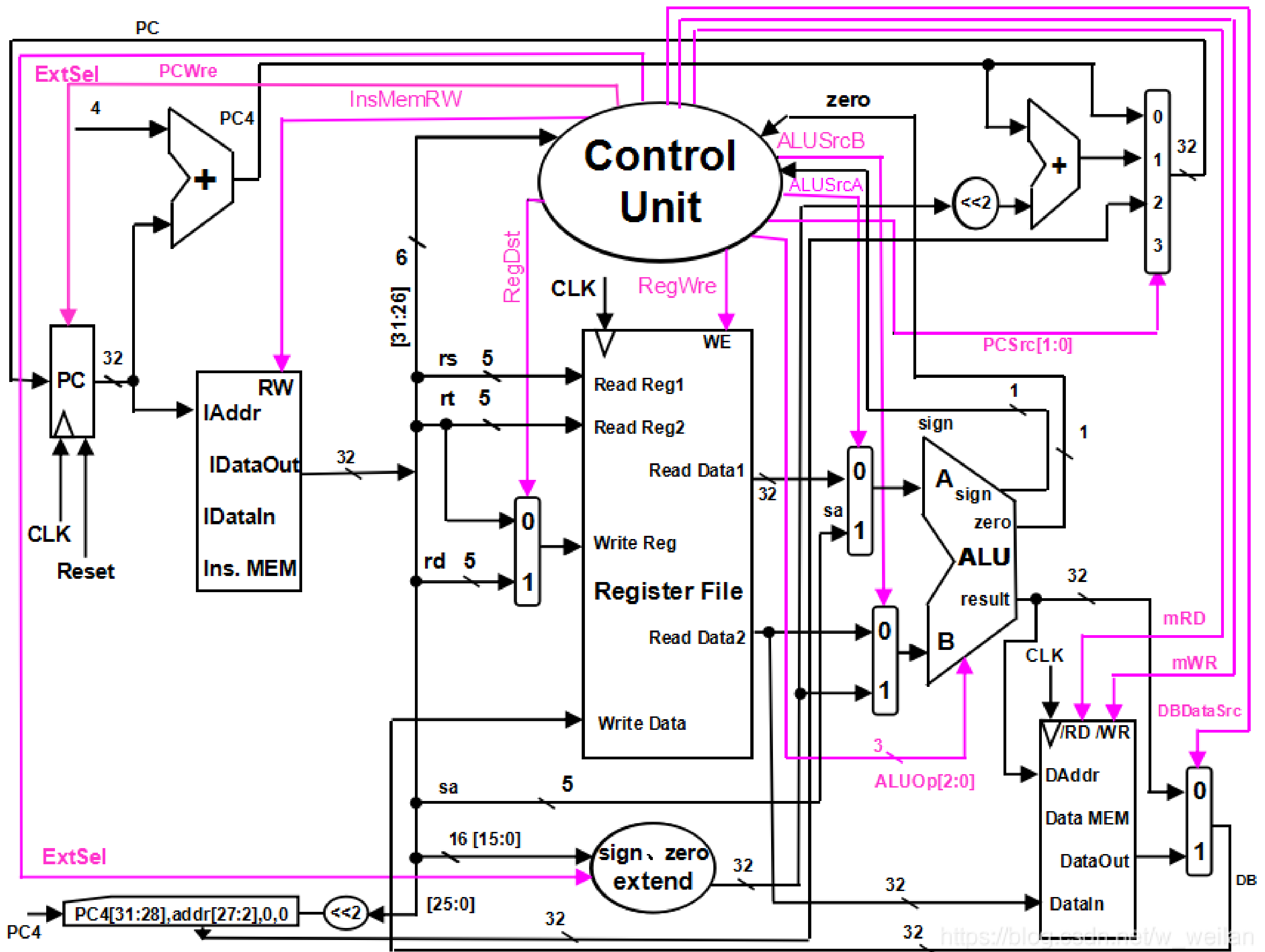
I类型

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

单周期CPU数据通路和控制线路图



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。

指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC的改变是在时钟上升沿进行的，这样稳定性较好。另外，值得注意的问题，设计时，用模块化的思想方法设计，关于ALU设计、存储器设计、寄存器组设计等等，也是必须认真考虑的问题。

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。

控制信号的作用表

以上数据通路图是根据要实现的指令功能的要求画出来的，同时，还必须确定ALU的运算功能(当然，以上指令没有完全用到提供的ALU所有功能，但至少必须能实现以上指令功能操作)。从数据通路图上可以看出控制单元部分需要产生各种控制信号，当然，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出以上表1，这样，从表1可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系（见下面表中的“相关指令”），从而写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

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

控制信号名	状态“0”	状态“1”
ALUSrcA	来自寄存器堆data1输出，相关指令：add、sub、addiu、or、and、andi、ori、slti、beq、bne、bltz、sw、lw	来自移位数sa，同时，进行(zero-extend)sa，即，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、bltz	来自sign或zero扩展的立即数，相关指令：addiu、andi、ori、slti、sw、lw
DBDataSrc	来自ALU运算结果的输出，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、bltz、sw、halt	寄存器组写使能，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	输出高阻态	读数据存储器，相关指令：lw
mWR	无操作	写数据存储器，相关指令：sw
RegDst	写寄存器组寄存器的地址，来自rt字段，相关指令：addiu、andi、ori、slti、lw	写寄存器组寄存器的地址，来自rd字段，相关指令：add、sub、and、or、sll
ExtSel	(zero-extend)immediate（0扩展），相关指令：andi、ori	(sign-extend)immediate（符号扩展），相关指令：addiu、slti、sw、lw、beq、bne、bltz

ALUOp的功能表

ALUOp[2…0]	功能	描述	相关指令
000	Y=A+B	加	add、addiu、sw、lw、j、halt
001	Y=A–B	减	sub、beq、bne、bltz
010	Y=B<<A	B左移A位	sll
011	Y=A∨B	或	ori、or
100	Y=A∧B	与	andi、and
101	Y=A<B	不带符号比较A<B	
110	Y=A[31]!=B[31]?A[31]>B[31]:A<B	带符号比较A<B	slti
111	Y=A^B	异或	

PCSrc的功能表

PCSrc[1…0]	功能	相关指令
00	pc<—pc+4	add、addiu、sub、or、ori、and、andi、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0)
01	pc<—pc+4+(sign-extend)immediate<<2	beq(zero=1)、bne(zero=0)、bltz(sign=1)
10	pc<—{(pc+4)[31:28],addr[27:2],2’b00}	j

PCSrc[1...0]	功能	相关指令
11	未用	

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

实验器材

电脑一台，Xilinx Vivado 2017.4 软件一套，Basys3实验板一块。

实验过程与结果

代码实现

SingleCPU.v

单周期CPU的顶层连接文件，主要是调用下层模块并将它们输入输出连在一起，并计算下一个指令的地址（正常+4或跳转）。

ControlUnit.v

控制信号模块，通过解析op得到该指令的各种控制信号。定义了很多用到的常量，可读性还是比较高的。
控制单元通过输入的zero零标志位与当前指令中对应的指令部分来确定当前整个CPU程序中各模块的工作和协作情况，根据CPU运行逻辑，事先对整个CPU中控制信号的控制，以此来达到指挥各个模块协同工作的目的。

ALU.v

该部分为算术逻辑单元，用于逻辑指令计算和跳转指令比较。ALUOp用于控制算数的类型，A、B为输入数，result为运算结果，zero、sign主要用于beq、bne、bltz等指令的判断。
ALU算术逻辑单元的功能是根据控制信号从输入的数据中选取对应的操作数，根据操作码进行运算并输出结果与零标志位。

DataMemory.v

该部分控制内存存储，用于内存存储、读写。用255大小的8位寄存器数组模拟内存，采用小端模式。DataMenRW控制内存读写。由于指令为真实地址，所以不需要`<<2`。

InstructionMemory.v

该部分为指令寄存器，通过一个256大小的8位寄存器数组来保存从文件输入的全部指令。然后通过输入的地址，找到相应的指令，输出到IDataOut。
指令存储器的功能是存储读入的所有32-bit位宽的指令，根据程序计数器PC中的指令地址进行取指令操作并对指令类型进行分析，通过指令类型对所取指令的各字段进行区分识别，最后将对应部分传递给其他模块进行后续处理。
指令存储器中每个单元的位宽为8-bit，也就是存储每条32-bit位宽的指令都需要占据4个单元，所以第n（n大于或等于0）条指令所对应的起始地址为4n，且占据第4n，4n+1，4n+2，4n+3这四个单元。取出指令就是将这四个单元分别取出，因为指令的存储服从高位指令存储在低位地址的规则，所以4n单元中的字段是该条指令的最高8位，后面以此类推，并通过左移操作将指令的四个单元部分移动到相对应的位置，以此来得到所存指令。

Multiplexer32.v

三十二线双路选择器。

Multiplexer5.v

五线双路选择器。

PC.v

CLK上升沿触发，更改指令地址。由于指令地址存储在寄存器里，一开始需要赋currentAddress为0。Reset是重置信号，当为1时，指令寄存器地址重置。PCWre的作用为保留现场，如果PCWre为0，指令地址不变。
PC程序计数器用于存放当前指令的地址，当PC的值发生改变的时候，CPU会根据程序计数器PC中新得到的指令地址，从指令存储器中取出对应地址的指令。在单周期CPU的运行周期中，PC值的变化是最先的，而且是根据PCSrc控制信号的值选择指令地址是要进行PC+4或者跳转等操作。若PC程序计数器检测到Reset输入信号为0时，则对程序计数器存储的当前指令地址进行清零处理。

RegisterFile.v

该部分为寄存器读写单元，储存寄存器组，并根据地址对寄存器组进行读写。WE的作用是控制寄存器是否写入。同上，通过一个32大小的32位寄存器数组来模拟寄存器，开始时全部置0。通过访问寄存器的地址，来获取寄存器里面的值，并进行操作。（由于$0恒为0，所以写入寄存器的地址不能为0）
寄存器组中的每个寄存器位宽32-bit,是存放ALU计算所需要的临时数据的,与数据存储器不同，可能会在程序执行的过程中被多次覆盖，而数据存储器内的数据一般只有sw指令才能进行修改覆盖。寄存器组会根据操作码opCode与rs，rt字段相应的地址读取数据,同时将rs，rt寄存器的地址和其中的数据输出，在CLK的下降沿到来时将数据存放到rd或者rt字段的相应地址的寄存器内。

SignZeroExtend.v

比较简单的一个模块，用于立即数的扩展。ExtSel为控制补位信号。判断后，将extendImmediate的前16位全补1或0即可。

仿真检验

将指令转换成二进制代码，如下表：

地址	汇编程序	op（6）	rs(5)	rt(5)	rd(5)/immediate(16)	16进制数代码
0x00000000	`addiu $1,$0,8`	000010	00000	00001	00000000 00001000	08010008
0x00000004	`ori $2,$0,2`	010010	00000	00010	00000000 00000010	48020002
0x00000008	`add $3,$2,$1`	000000	00010	00001	00011000 00000000	00411800
0x0000000C	`sub $5,$3,$2`	000001	00011	00010	00101000 00000000	04622800
0x00000010	`and $4,$5,$2`	010001	00101	00010	00100000 00000000	44a22000
0x00000014	`or $8,$4,$2`	010011	00100	00010	01000000 00000000	4c824000
0x00000018	`sll $8,$8,1`	011000	00000	01000	01000000 01000000	60084040
0x0000001C	`bne $8,$1,-2`	110001	01000	00001	11111111 11111110	c501fffe
0x00000020	`slti $6,$2,4`	011100	00010	00110	00000000 00000100	70460004

地址	汇编程序	op（6）	rs(5)	rt(5)	rd(5)/immediate(16)	16进制数代码
0x00000024	`slti $7,$6,0`	011100	00110	00111	00000000 00000000	70c70000
0x00000028	`addiu $7,$7,8`	000010	00111	00111	00000000 00001000	08e70008
0x0000002C	`beq $7,$1,-2`	110000	00111	00001	11111111 11111110	c0e1fffe
0x00000030	`sw $2,4($1)`	100110	00001	00010	00000000 00000100	98220004
0x00000034	`lw $9,4($1)`	100111	00001	01001	00000000 00000100	9c290004
0x00000038	`addiu $10,$0,-2`	000010	00000	01010	11111111 11111110	080afffe
0x0000003C	`addiu $10,$10,1`	000010	01010	01010	00000000 00000001	094a0001
0x00000040	`bltz $10,-2`	110010	01010	00000	11111111 11111110	c940fffe
0x00000044	`andi $11,$2,2`	010000	00010	01011	00000000 00000010	404b0002
0x00000048	`j 0x00000050`	111000	00000	00000	00000000 00010100	e0000014
0x0000004C	`or $8,$4,$2`	010011	00100	00010	01000000 00000000	4c824000
0x00000050	`halt`	111111	00000	00000	00000000 00000000	fc000000

input.txt

```
00001000 00000001 00000000 00001000
01001000 00000010 00000000 00000010
00000000 01000001 00011000 00000000
00000100 01100010 00101000 00000000
01000100 10100010 00100000 00000000
01001100 10000010 01000000 00000000
01100000 00001000 01000000 01000000
11000101 00000001 11111111 11111110
01110000 01000110 00000000 00000100
01110000 11000111 00000000 00000000
00001000 11100111 00000000 00001000
11000000 11100001 11111111 11111110
10011000 00100010 00000000 00000100
10011100 00101001 00000000 00000100
00001000 00001010 11111111 11111110
00001001 01001010 00000000 00000001
11001001 01000000 11111111 11111110
01000000 01001011 00000000 00000010
11100000 00000000 00000000 00010100
01001100 10000010 01000000 00000000
11111100 00000000 00000000 00000000
```

Sim.v

仿真模块。

```
```

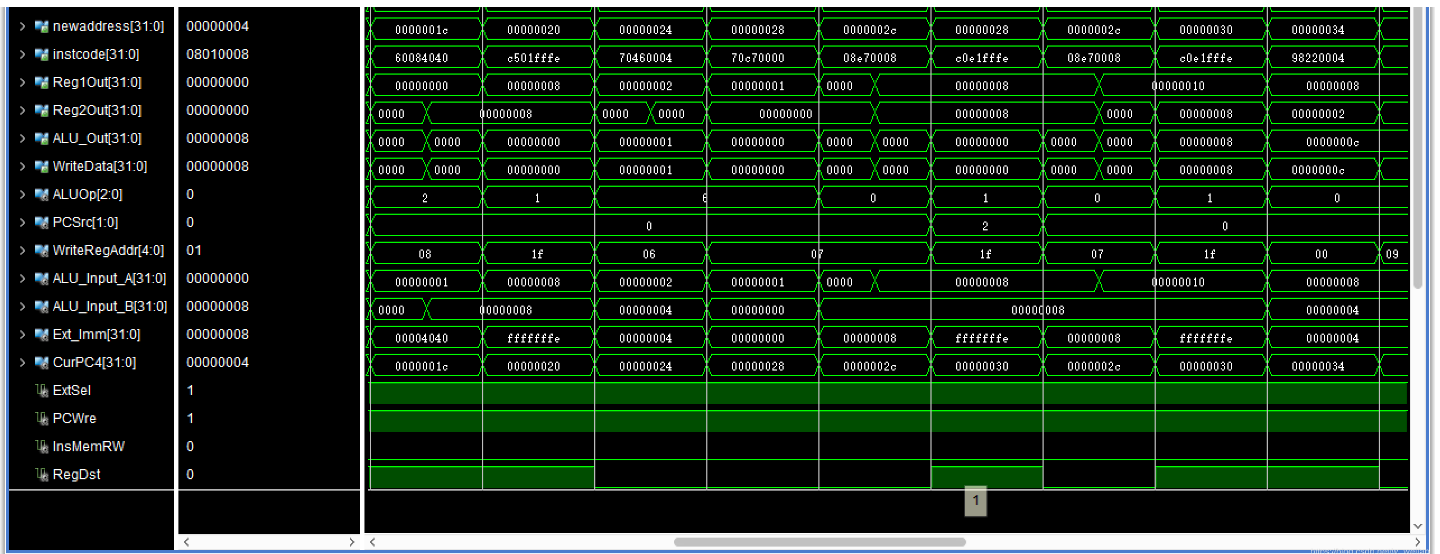
仿真波形

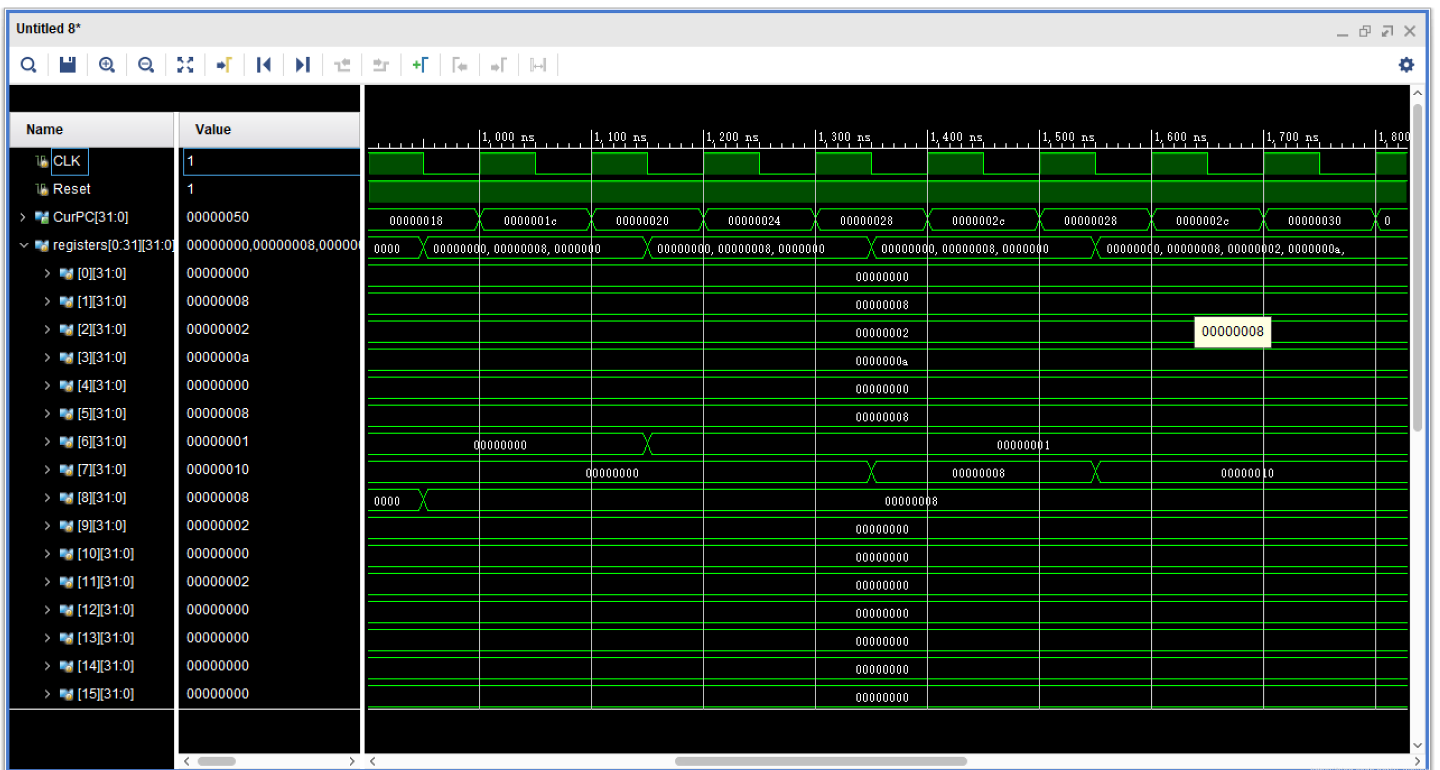
波形比较长，分成三部分逐一分析。

前100ps为初始化，各寄存器的值被初始化为0。

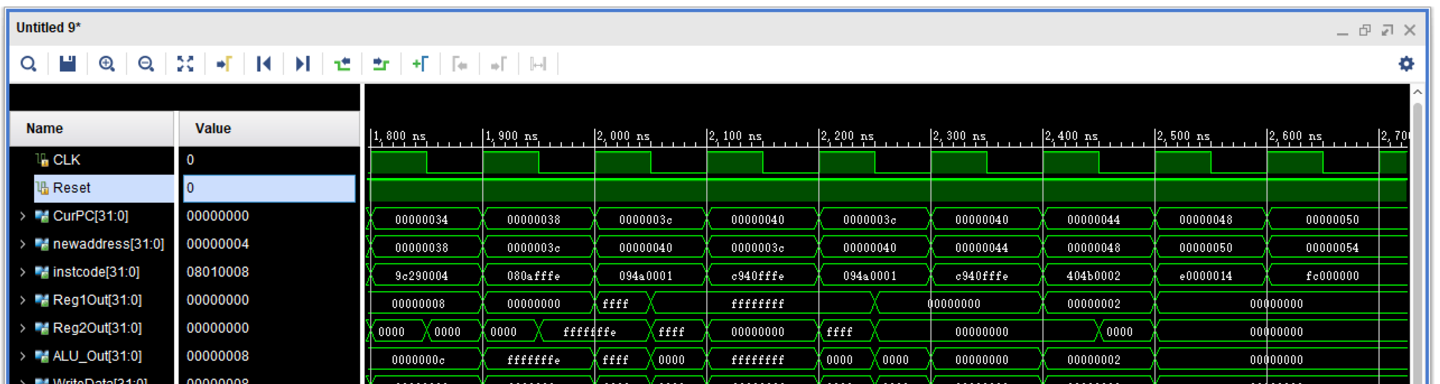
第800ps时执行了 `bne $8,$1,-2` ，此时寄存器 `$8` 的值是4，寄存器 `$1` 的值是8，两者不等，发生了一步跳转，于是第900ps的地址跳转到 `00000018` 。

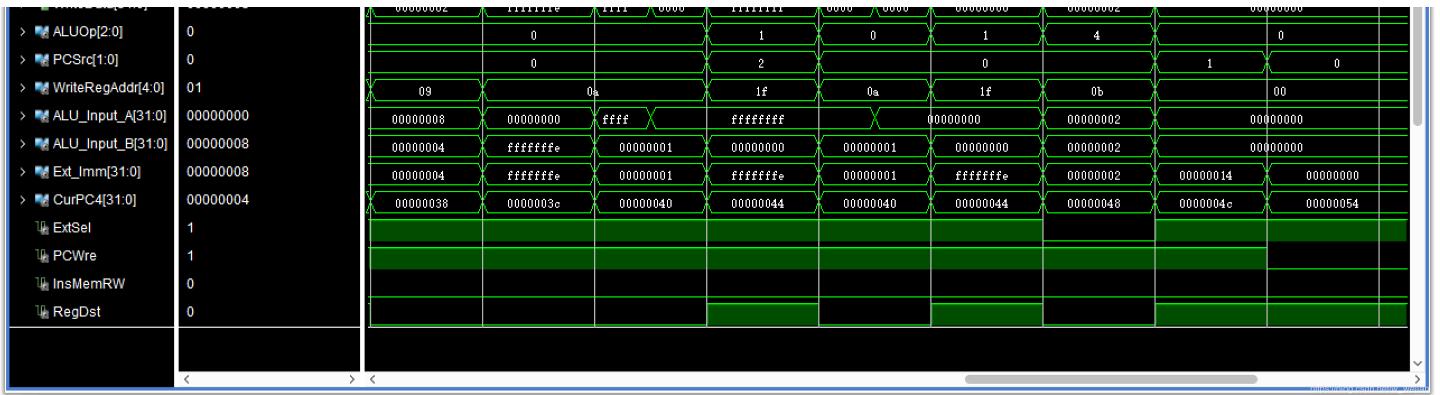
在第900ps前寄存器1~11的值依次为（16进制） `8,2,a,0,8,0,0,4,0,0,0` 。

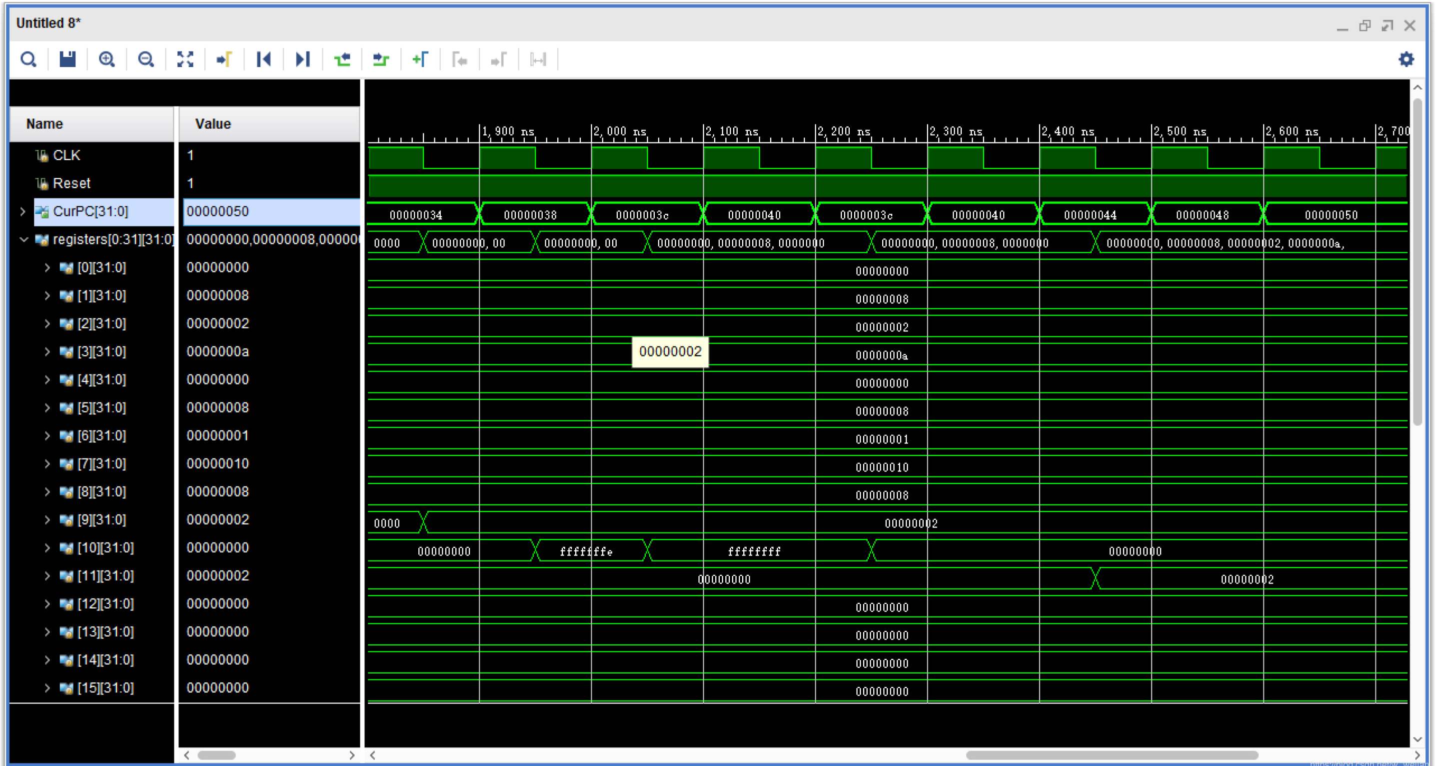




第1000ps时再次执行了`bne $8,$1,-2`，此时寄存器`$8`的值是`8`，寄存器`$1`的值是`8`，两者相等，pc+4执行下一条指令。
第1400ps时执行了`beq $7,$1,-2`，此时寄存器`$7`的值是`8`，寄存器`$1`的值是`8`，两者相等，发生了一步跳转，于是第1500ps的地址跳转到`00000028`。
第1600ps时再次执行了`beq $7,$1,-2`，此时寄存器`$7`的值是`10`，寄存器`$1`的值是8，两者不等，pc+4执行下一条指令。
在第1800ps前寄存器1~11的值依次为（16进制）`8,2,a,0,8,1,10,8,0,0,0`。







第2100ps时执行了`bltz $10,-2`，此时寄存器`$10`的值是`ffffffff`，小于`0`，发生了一步跳转，于是第2200ps的地址跳转到`0000003c`。
第2300ps时再次执行了`bltz $10,-2`，此时寄存器`$10`的值是`0`，不小于`0`，pc+4执行下一条指令。
第2500ps时执行了`j 0x00000050`，于是第2600ps的地址跳转到`00000050`。
第2600ps时执行了`halt`，程序终止，pc不再跳转。
在第2700ps前寄存器1~11的值依次为（16进制）`8,2,a,0,8,1,10,8,2,0,2`。

烧写到Basys3实验板

Basys3.v

顶层模块。

```
```

Debounce.v

按键消抖模块。Basys3板采用的是机械按键，在按下按键时按键会出现人眼无法观测但是系统会检测到的抖动变化，这可能会使短时间内电平频繁变化，导致程序接收到许多错误的触发信号而出现许多不可知的错误。消抖操作是每当检测到CLK上升沿到来时检测一次当前电平信号并记录，同计数器开始计数，若在计数器达到5000之前电平发生变化，则将计数器清零，若达到5000，则将该记录电平取反输出。
因为程序开始时已经运行第一条指令，为避免跳过第一条指令计算值的写入，我们的输入需要从下降沿开始，因此我们给按键信号取反后再输入。

SegLED.v

数码管译码模块。译码模块将CPU运算的结果转换成7段数码管中各个数码管显示所需的高低电平信号,该单元的输入为4-bit位宽的二进制数。其中，七段数码管的八个电平控制输出中最低位是小数点的显示信号，但小数点在CPU运行时没有用到，恰好用于标记Reset状态。

运行结果

端口映射

I/O Ports

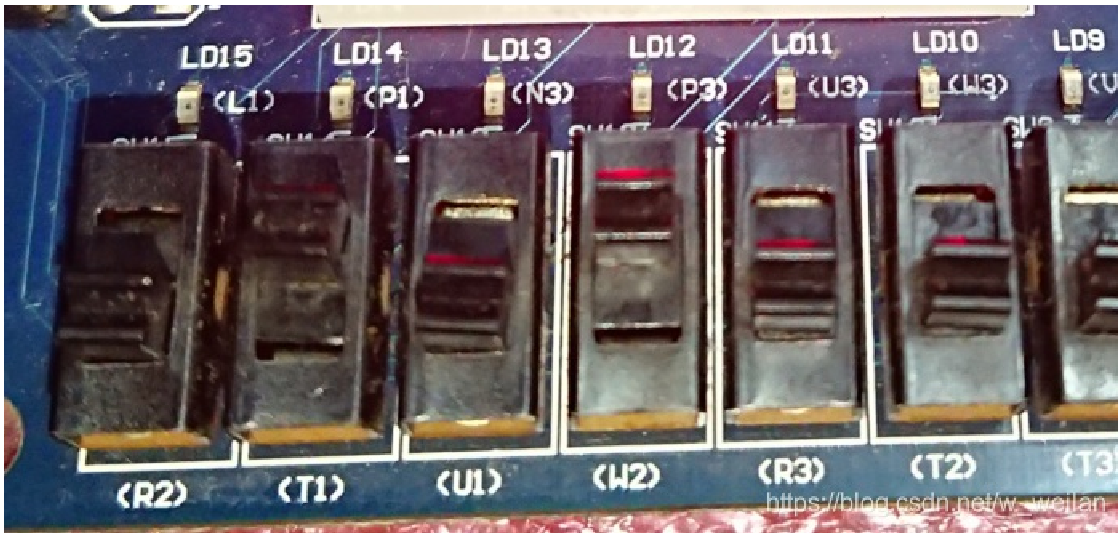
Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type
All ports (17)											
AN (4)	OUT			✓	34	LVCMOS33*	3.300		12	SLOW	NONE
AN[3]	OUT		U2	✓	34	LVCMOS33*	3.300		12	SLOW	NONE
AN[2]	OUT		U4	✓	34	LVCMOS33*	3.300		12	SLOW	NONE
AN[1]	OUT		V4	✓	34	LVCMOS33*	3.300		12	SLOW	NONE
AN[0]	OUT		W4	✓	34	LVCMOS33*	3.300		12	SLOW	NONE
Out (8)	OUT			✓	34	LVCMOS33*	3.300		12	SLOW	NONE
Out[7]	OUT		W7	✓	34	LVCMOS33*	3.300		12	SLOW	NONE
Out[6]	OUT		W6	✓	34	LVCMOS33*	3.300		12	SLOW	NONE
Out[5]	OUT		U8	✓	34	LVCMOS33*	3.300		12	SLOW	NONE
Out[4]	OUT		V8	✓	34	LVCMOS33*	3.300		12	SLOW	NONE
Out[3]	OUT		U5	✓	34	LVCMOS33*	3.300		12	SLOW	NONE
Out[2]	OUT		V5	✓	34	LVCMOS33*	3.300		12	SLOW	NONE
Out[1]	OUT		U7	✓	34	LVCMOS33*	3.300		12	SLOW	NONE
Out[0]	OUT		V7	✓	34	LVCMOS33*	3.300		12	SLOW	NONE
SW (2)	IN			✓	34	LVCMOS33*	3.300				NONE
SW[1]	IN		U1	✓	34	LVCMOS33*	3.300				NONE
SW[0]	IN		W2	✓	34	LVCMOS33*	3.300				NONE
Scalar ports (3)											
Button	IN		R2	✓	34	LVCMOS33*	3.300				NONE
CLK	IN		W5	✓	34	LVCMOS33*	3.300				NONE
Reset	IN		T1	✓	34	LVCMOS33*	3.300				NONE

初始化

所有寄存器被初始化为0。

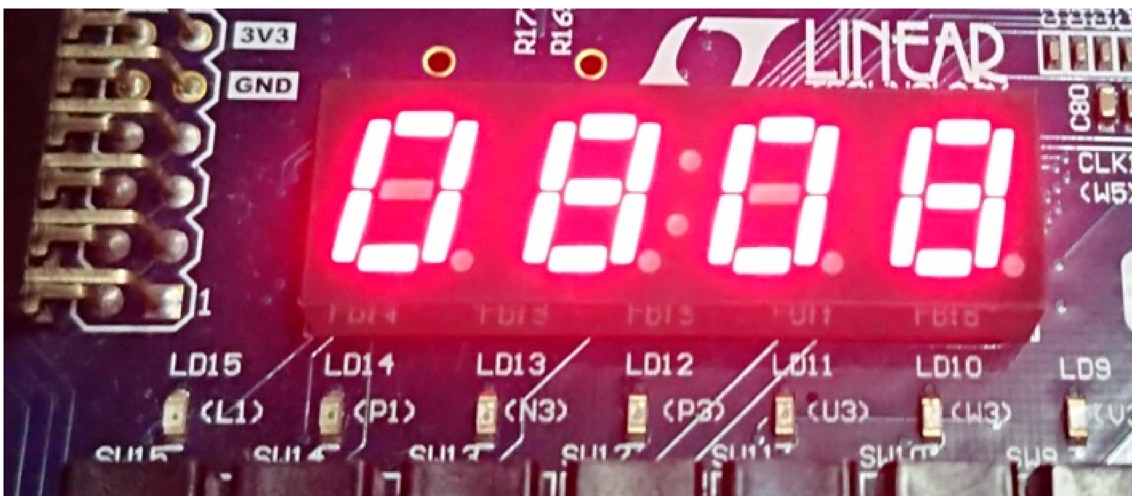
第1条指令 `addiu $1,$0,8`

当前地址00，下一地址04。



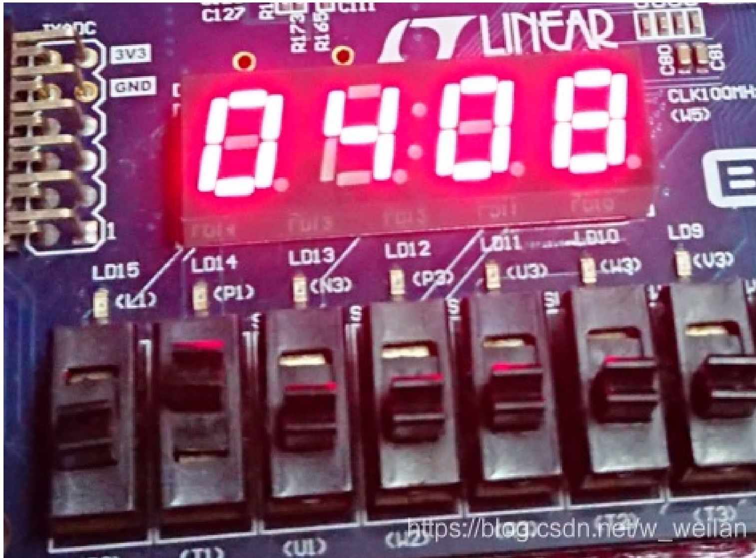
0号寄存器，值为0。

1号寄存器，值为0。

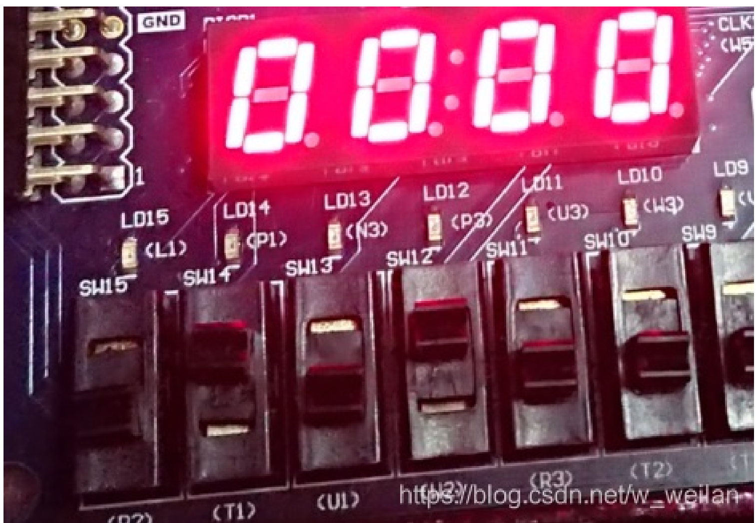


ALU结果为8。

第2条指令`ori $2,$0,2`



当前地址04，下一地址08。

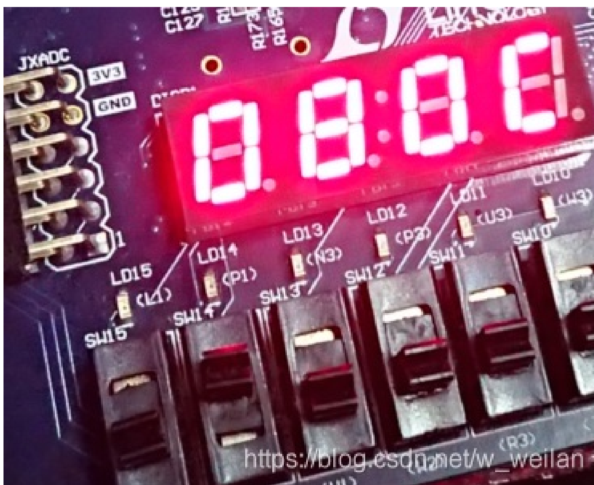


0号寄存器，值为0。

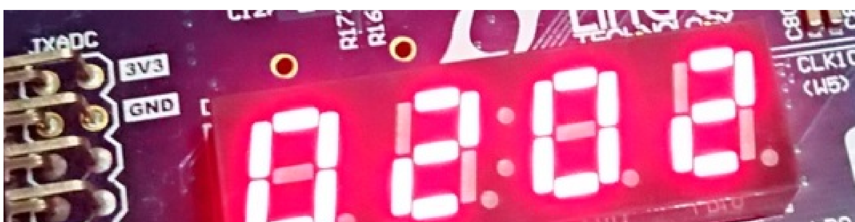
2号寄存器，值为0。

ALU结果为2。

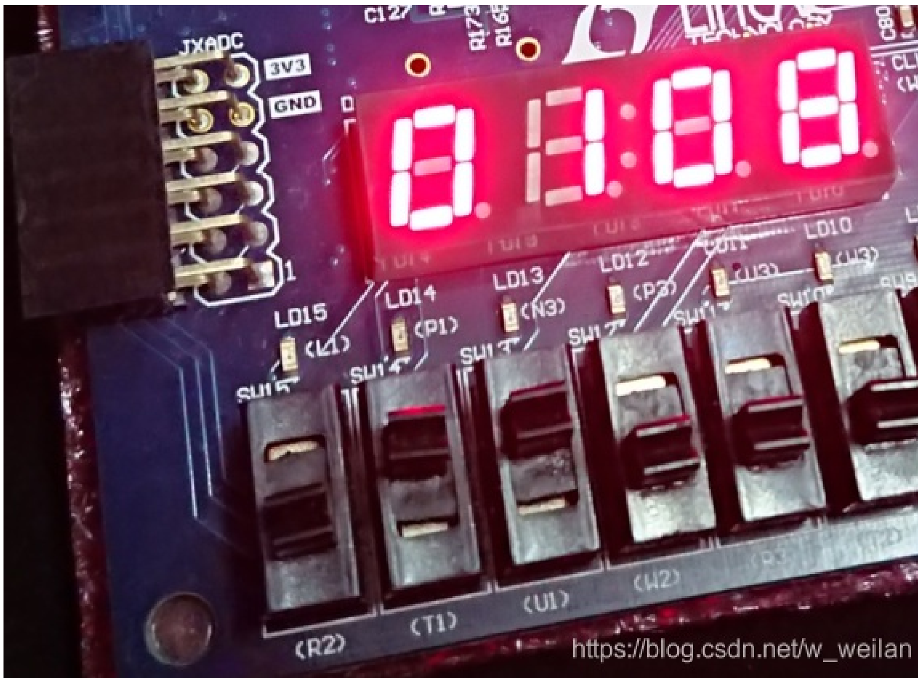
第3条指令`add $3,$2,$1`



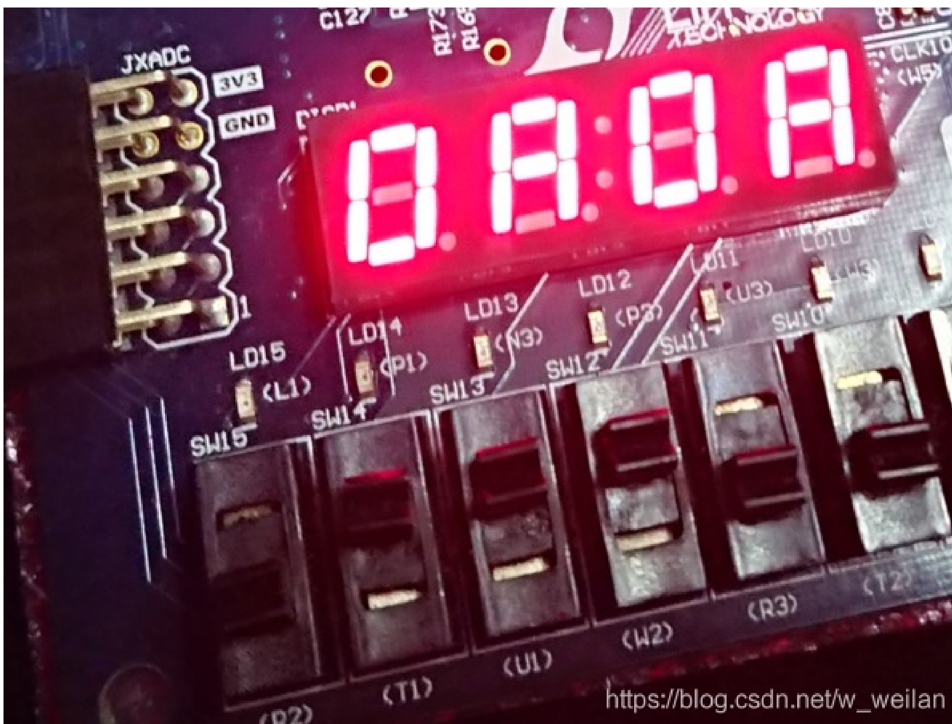
当前地址08，下一地址0c。



2号寄存器，值为2。

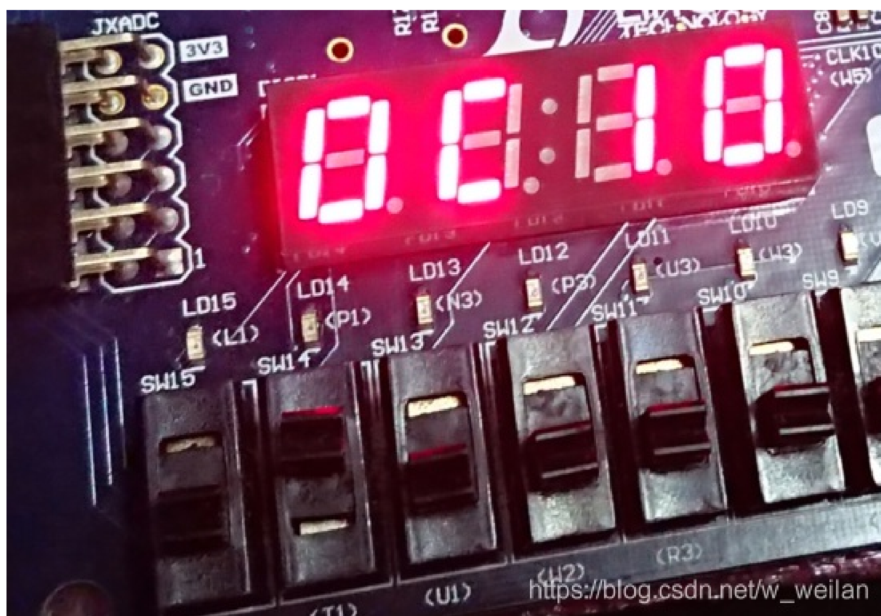


1号寄存器，3号寄存器。



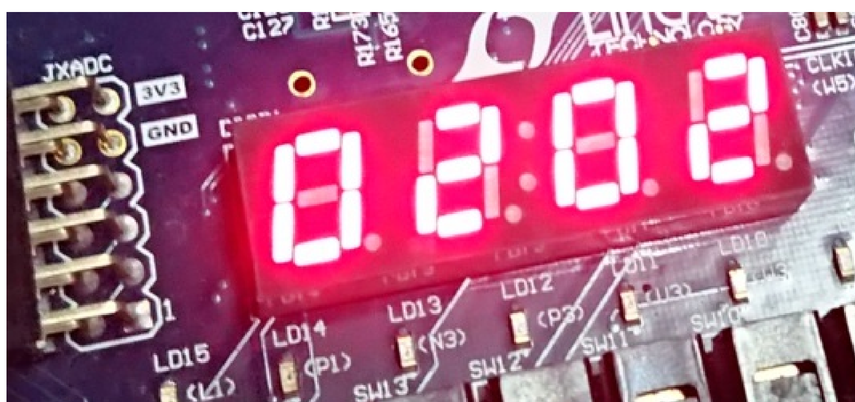
输出结果为0a。

第4条指令`sub $5,$3,$2`

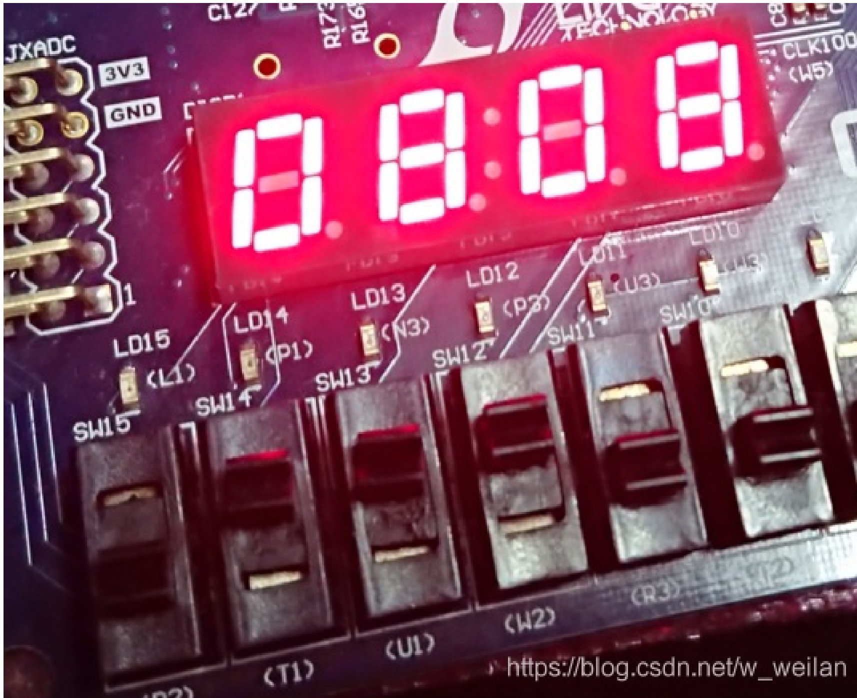


当前地址0c，下一地址10。

3号寄存器，值为0a。



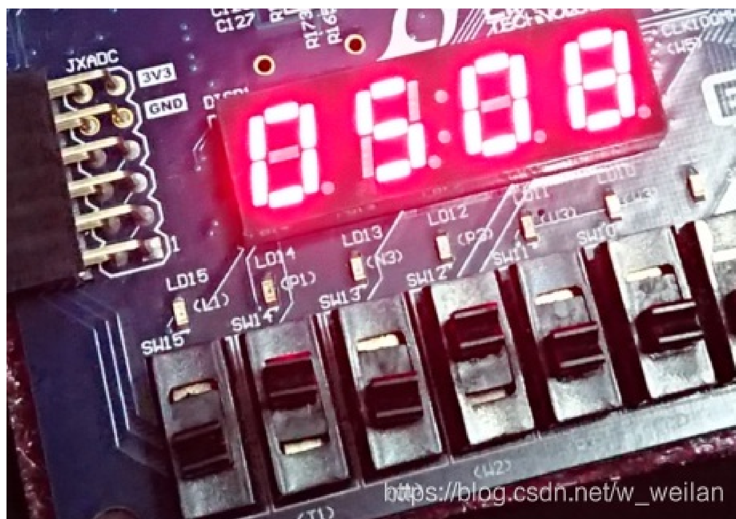
2号寄存器，值为2。



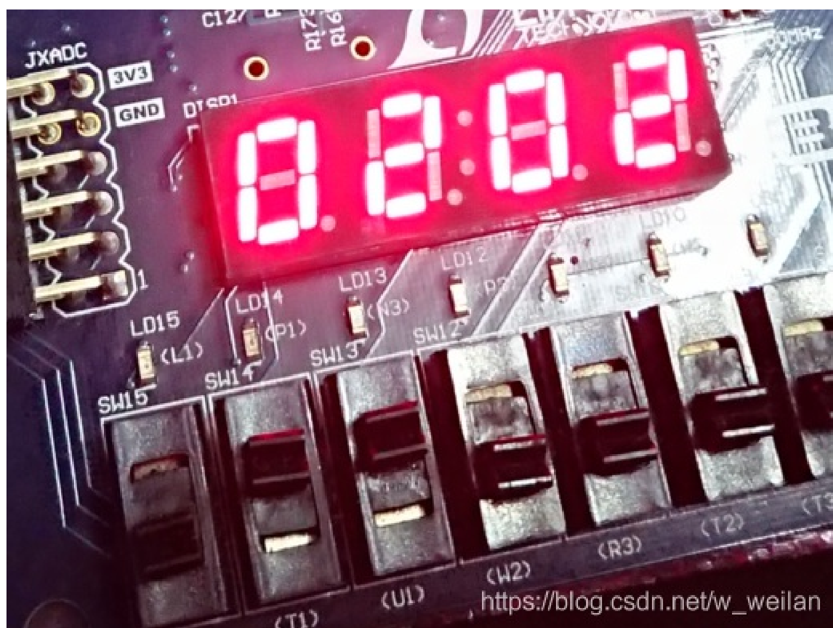
ALU结果为8。

第5条指令 `and $4,$5,$2`

当前地址10，下一地址14。



5号寄存器，值为3。



2号寄存器，值为2。

ALU结果为0。

实验心得

对于第一次使用verilog语言的我来说，设计单周期CPU是一个不小的挑战。总的来说，从开始构思到真正写板完成大约用了三整天的时间，期间遇到了很多有困难的地方，也翻了很多网上的博客。不得不说，网上能找到的很多博客给出的代码都是有些问题的，确实在一定程度上误导了自己。但是这样一个比较复杂的系统又确实很难什么都不去参考而直接写出来。

因此，还是希望老师能够多做一些有关verilog语言的讲学吧，遇到的很多问题其实都是关于语法方面的，因为vivado并不会对语法错误进行提示（我用的2017.4版会在可能错误的地方画一道波浪线但是并不会提示错在哪里），而我遇到的问题很多都是语法方面的（例如，在readmemb的时候地址的斜杠和操作系统的是反的，这花费了我很多时间检查才发现）。