第 1 章 绪论

1.1 32 位模型机简介

MIPS 架构（英语：MIPS architecture，为 Microprocessor without Interlocked Pipeline Stages 的缩写，亦为 Millions of Instructions Per Second 的头字语），是一种采取精简指令集（RISC）的处理器架构，1981 年出现，由 MIPS 科技公司开发并授权，广泛被使用在许多电子产品、网络设备、个人娱乐设备与商业设备上。最早的 MIPS 架构是 32 位，最新的版本已经变成 64 位。

1.2 设计主要内容

1.2.1 简单的多周期 MIPS 处理器

在 MIPS 架构中，指令被分为三种类型：R 型、I 型和 J 型。三种类型的指令的最高 6 位均为 6 位的 opcode 码。从 25 位往下：

1.R 型指令用连续三个 5 位二进制码来表示三个寄存器的地址，然后用一个 5 位二进制码来表示移位的位数（如果未使用移位操作，则全为 0），最后为 6 位的 function 码（它与 opcode 码共同决定 R 型指令的具体操作方式）。

2.I 型指令则用连续两个 5 位二进制码来表示两个寄存器的地址，然后是一个 16 位二进制码来表示的一个立即数二进制码。

3.J 型指令用 26 位二进制码来表示跳转目标的指令地址（实际的指令地址应为 32 位，其中最低两位为 00，高四位由 PC 当前地址决定）。

类型	格式（位）					
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

多周期微结构（multicycle microarchitecture)用多个较短的周期实现 1 条指令。简单指令的执行周期数较少。而且多周期微结构叫以通过对加法器和存储器等复杂硬件部件的复用减少硬件成本。例如，同一个加法器可以在 1 条指令的不同时钟周期中用于不同的目的。多周期撖结构需要增加一些非体系结构寄存器以保存中间结果。多周期处埋器在任意时刻仅执行一条指令，但是每条指令需要多个周期。

一个简单的 32 位多周期 MIPS 处理器，可以执行 11 种指令，其中 R 指令 6 种，分别为 add(加法),sub(减法),and(位与),or(位或),xor(位异或),slt(小于)，I 指令 4 条，分别为

lw(从内存中读取一个 word),sw(向内存写入一个 word),beq(相同则跳转),addi(加立即数)，J 指令 1 条，为 j(无条件跳转)。CPU 频率为 50MHz(CLK 由开发板提供)。

第 2 章 系统设计

2.1 系统组成

在设计整体结构时，依据的是各指令的数据通路。然后采用自顶向下，逐步分解细化的方法进行设计。先整体模块，后局部模块。

表 2.1 简易微处理器的指令系统

汇编语句	操作符	功能
add $d, $s, $t	0000 00ss ssst tttt dddd d000 0010 0000	$d = $s + $t; pc+=4;
sub $d, $s, $t	0000 00ss ssst tttt dddd d000 0010 0010	$d = $s - $t; pc+=4;
and $d, $s, $t	0000 00ss ssst tttt dddd d000 0010 0100	$d = $s & $t; pc+=4;
or $d, $s, $t	0000 00ss ssst tttt dddd d000 0010 0101	$d = $s | $t; pc+=4;
xor $d, $s, $t	0000 00ss ssst tttt dddd d000 0010 0110	$d = $s ^ $t; pc+=4;
slt $d, $s, $t	0000 00ss ssst tttt dddd d000 0010 1010	$d=(bool)$s<$t;pc+=4;
addi $t, $s, imm	0010 00ss ssst tttt iiii iiii iiii iiii	$t = $s + imm; pc+=4;
beq $s, $t, offset	0001 00ss ssst tttt iiii iiii iiii iiii	if $s == $t pc+=(offset*4); else pc+=4;
lw $t, offset($s)	1000 11ss ssst tttt iiii iiii iiii iiii	$t=MEM[$s+offset]; pc+=4;
sw $t, offset($s)	1010 11ss ssst tttt iiii iiii iiii iiii	MEM[$s+offset]=$t; pc+=4;
j target	0000 10ii iiii iiii iiii iiii iiii iiii	pc=nPC;nPC=(pc& 0xf0000000) | (target << 2);

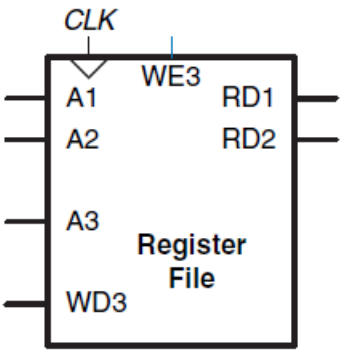
2.2 CPU 模块

可以将 CPU 分为两个互相关联的部分：数据路径（datapath) 和控制（control) 。数据路径对数据以字为单位进行操作，它包含了存储器、寄存器、ALU 和多路开关等结构。控制单元从数据路径接收当前指令，并控制数据路径如何执行这条指令。控制单元往往通过产生多路开关选择、寄存器使能、存储器写入等信号来控制数据路径的操作。

此实体里还包括了串口输入模块，但这不是 CPU 的一部分，所以没有一起展示在图中。

2.2.1 寄存器文件(regfile)

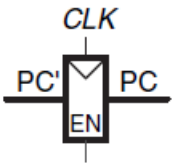
32 单元的 32 位寄存器文件（register file/regfile)有两个读出端口和一个写入端口，两个读出端口分别具有 5 位的地址输入 A1 和 A2,用于指定 2^5＝32 个寄存器中的一个作为操作数，可以将 32 位寄存器的值分别输出到 RD1 和 RD2 上,写入端口具有 5 位地址输入 A3. 32 位数据输入 WD, 写入使能 WE3 和时钟信号 CLK。如果写入使能为 1,则寄存器文件将在时钟上升沿对特定寄存器写入数据。



值得一说的是因为 0 这个值比较常用，当输入 RA 为 0 时，输出 RD 衡为 0，这就实现了$zero 寄存器。

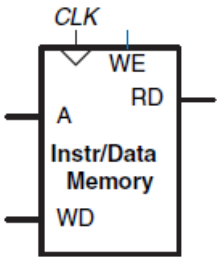
2.2.2 程序计数器(PC)

程序计数器（program counter)是一个普通的 32 位寄存器。其输出 PC 指向当前地址，输入 PC’表示下一条指令地址，若 PCEn 为 1 则将 PC’写入。



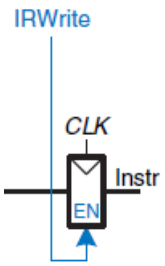
2.2.3 指令/数据存储器(imem)

指令/数据存储器（instr/data memory)有一个读写端，包括 32 位指令地址输入 A,能从当前地址读出 32 位数据（指令）到数据输出端口 RD，或将写入端口 WD 的值写入到地址 A 指定的单元中。如果写使能 WE 为 1，则数据 WD 在时钟上升沿写人到地址 A 指定的单元中。如果写使能为,则从地址 A 渎出数据到 RD。



2.2.4 指令寄存器(IR)

指令寄存器(instruction register)是一个普通的 32 位寄存器。指令从指令/数据存储器读入后将存入指令寄存器，它有一个使能控制端 IRWrite，当此信号有效时指令寄存器将更新为一条新的指令。



2.2.5 算术逻辑单元(alu)

在一个算术逻辑单元（Arithmetic/Logic Unit，ALU)内组合多种算术和逻辑的操作。

本次实现的 CPU 中算术逻辑单元可以执行加法、减法、数量比较、AND、OR、XOR 操作。

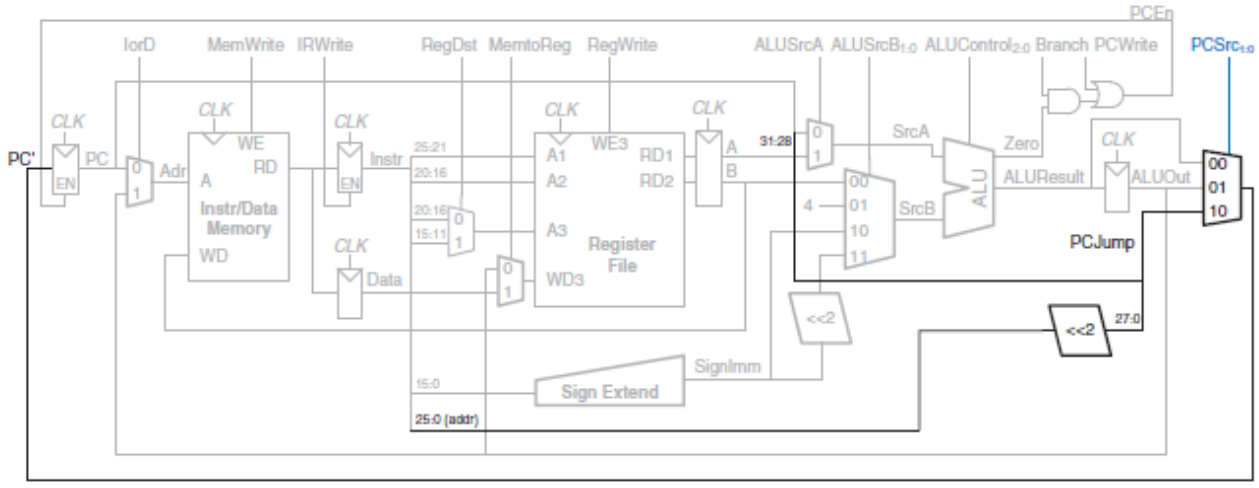
下面给出 ALU 的功能表

F[2..0]	功能
000	a and b
001	a or b
010	a + b
011	a xor b
100	未使用
101	未使用
110	a - b
111	a less than b

2.2.6 数据路径(datapath)

数据路径实现了连接各个模块，并根据控制器的输出选择操作的功能。详细的分析在控制器的说明下。

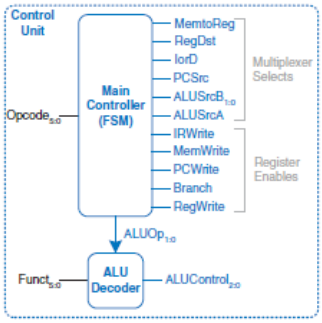
原理图



由于存储组件仅在时钟上升沿发生变化，因此是同步时序电路。微处理器由时钟驱动的存储组件和组合逻辑构成，因此也是同步时序电路.而且处理器可以看做一个有限状

态自动机，或者是若干简单而相互交互的有限状态自动机的组合。

2.2.7 控制器(contorl)

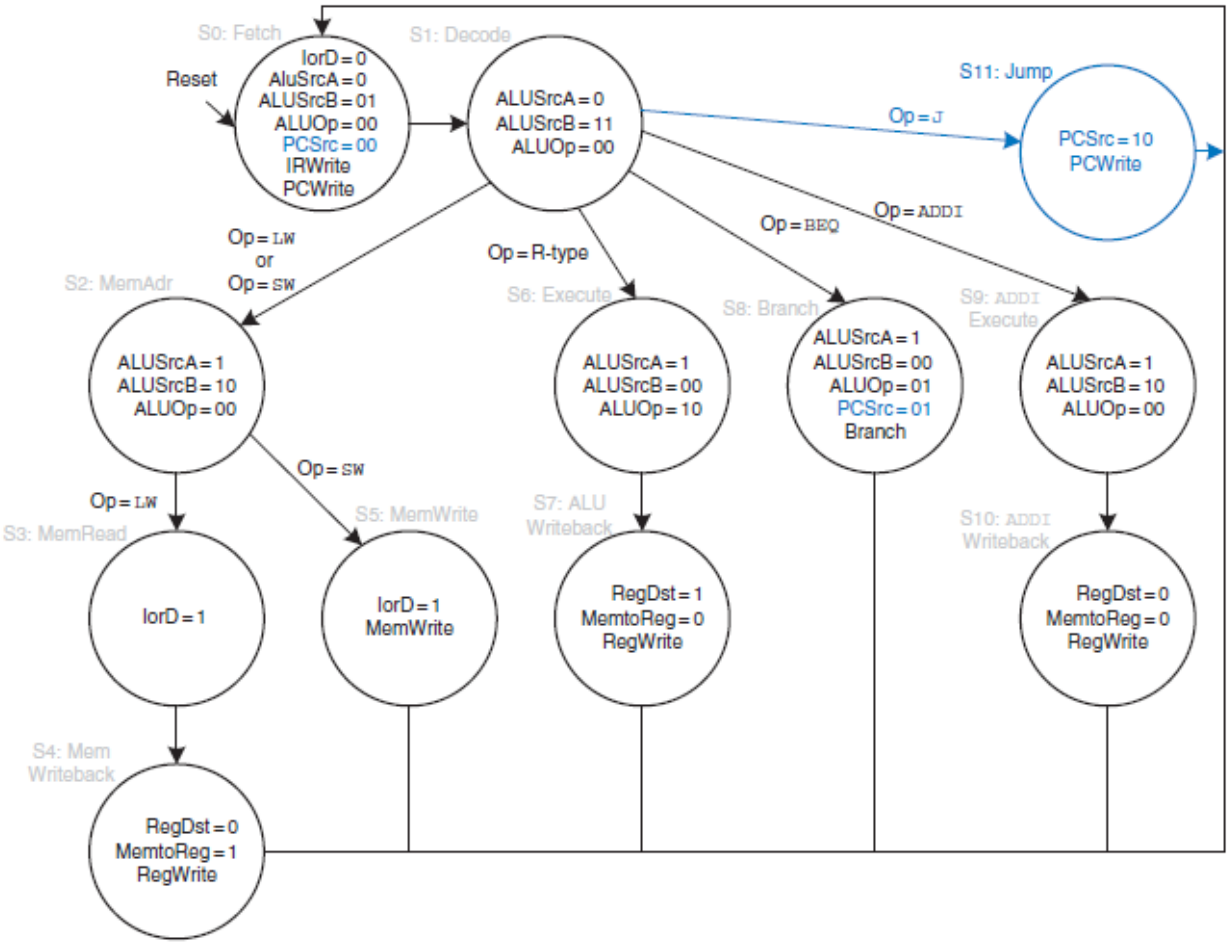


在处理器中，控制单元分为主控制器和 ALU 译码器，如图所示。

主控制器采用 FSM 实现以在合适的周期中产生合适的控制信号。控制信号序列依赖于当前正在执行的指令。

状态机(fsm)

状态机示意图如下。



所有指令的第 1 个周期都是根据 PC 指向的地址从存储器中读取指令。在复位后将进入这个状态。为了读取存储器，IorD = 0,从 PC 获得地址。IRWrite 将有效以将指令写入到指令寄存器 IR 中。同时 pc 应该+4 以指向下一条指令。由于 ALU 在此时设有被作为它用，因此处理器可以在取出指令的同时计算 PC+4。 ALUSrcA=0,使得 SrcA 来源于 PC, ALUSrcB = 01，使 得 SrcB 来源于常数 4，ALUOP=00，使得 ALU 译码器产生控制信号 ALUControl=010 完成加法。为了在此周期更新 PC, PCSrc=0，并设置 PCWrite 为有效。

下个周期将读取寄存器文件，并对指令译码。寄存器文件始终由指令中的 rs 和 rt 字段读取。同时，立即数进行符号扩展。译码操作为检查指令中 Opcode 字段以决定后续操作。译码阶段不需要控制信号，但是需要等待一个周期以完成读取和译码。现在 FSM 将根据 opcode 产生多种可能后续状态之一。如果指令时存储器读取或存储指令（lw 或 sw),处理器将基地址和符号扩展的立即数相加以计算地址，这需要 ALUSrcA =1 以选择寄存器 A, ALUSrcB=10 以选择 Signlmm, ALUOp = 00 使得 ALU 执行加法。有效地址将存储在 ALUOut 寄存器中为下一个周期做准备。

如果指令为 lw,处理器必须紧接着从存储器中读取数据并写入到寄存器件中,为了读取存储器,IorD=1 以将 ALUOut 中刚刚计算得到的值作为存储器地址，存储器将读取这个地址， 并将结果在 S3 周期存放到 Data 寄存器中。在下一个周期 S4 中，Data 将写入到寄存器文件， MemtoReg= 1 以选择 Data, RegDst = 0 以从指令中的 rt 字段获得目的寄存器地址。RcgWritc 将为 1 以执行写入操作，从而完成 lw 指令。最后，FSM 将返回到初始状态 S0 以取下一条指令。

如果 opcode 为 R 类型指令，处理器使用 ALU 计算结果并将结果写入到寄存器文件中。在 S6 状态中，通过选择寄存器 A 和 B (ALUSrcA=l, ALUSrcB=00)并根据指令中的 funct 字段执行 ALU 操作以完成指令。ALUResult 存储器在 ALUOut 中。在状态 S7, ALUOut 将写入寄存器文件。RegDst=l,这是因为目的寄存器由指令中的 rd 字段指定。MemtoRcg=0,这是因为要写入数据 WD3 来自于 ALUOut。RegWrite 设置为有效以写入寄存器文件。

对于 beq 指令，处理器必须计算目的地址，并比较两个源寄存器以确定是否发生转移。这需要两次使用 ALU，可以注意到在 S1 状态时完成寄存器读出操作，而没有使用 ALU，处理器可以在此时利用 ALU 将递增后的 PC 和 Signlmm * 4 相加以计算目的地址。ALUSrcA=0，以选择递增后的 PC，ALUSrcB=11 以选择 Signlmm*4,而且 ALUOp=00

以完成加法。目的地址将存储器在 ALUOut 中。在 S8 状态， 处理器通过减法并判断结果是否为 0 来比较两个寄存器的值，如果结果为 0，则处理器将转移到刚刚计算得到的地址。ALUSrcA=11 以选择寄存器 A，ALUSrcB=00，以选择寄存器 B, ALUOp=01,以完成减法，PCSrc=l，以从 ALUOut 中得到目的地址，当 ALU 结果为 0 时 Branch=l,以更新 PC。

对于 addi 指令，数据路径已经提供了将寄存器和立即数相加的能力，因此需要做的就是在主控制器 FSM 中加入新的状态.这个状态类似于 R 类型指令。在状态 S9 中寄存器 A 将加上 Signlmm (ALUSrcA=l， ALUSrcB =10, ALUOp=00)。其结果 ALUResult 将存储在 ALUOut.在 SI0 状态.ALUOut 将写入到指令 rt 字段指定的寄存器中（RegDst=0, MemtoReg=0. RegWrite 为有效）。

对于 j 指令，跳转目的地址由指令中的 26 位 addr 字段左移 2 位得到，接着要保持递增后 PC 的高四位，应扩展 PCSrc 多路开关将此地址作为第三个输入，S11 选择 PCJump 值作为 PC’(PCSrc = 10)并写入到 PC,转移 PCSrc。

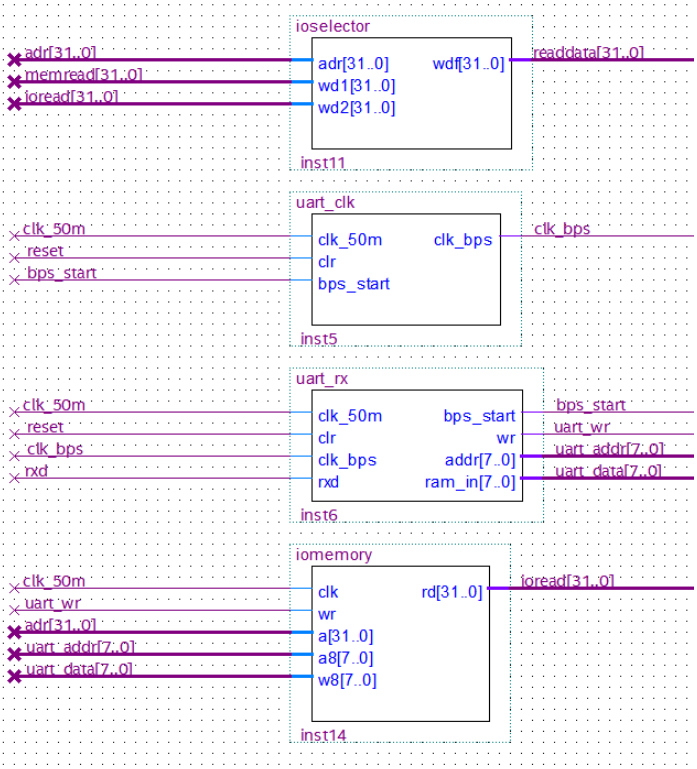
ALU 译码器(aludec)

ALU 译码器满足下方的真值表。

ALUOp	Funct	ALUControl	说明
00	X	010	(add for lw/sw/addi)
X1	X	110	(subtract for beq)
1X	100000(add)	010	(add)
1X	100010(sub)	110	(subtract)
1X	100100(and)	000	(and)
1X	100101(or)	001	(or)
1X	101010(slt)	111	(set less than)
1X	10 0110(xor)	011	(xor)

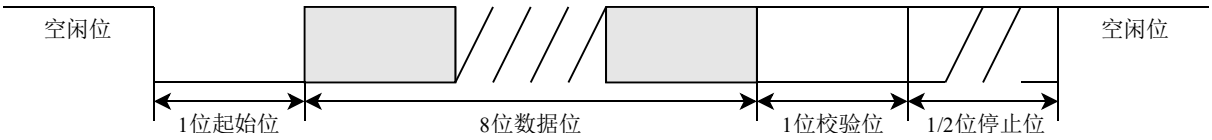
2.3 I/O 设备

2.3.1 UART 串口通信



开发板包含了 Silicon Labs CP2102GM 的 USB-UAR 芯片, USB 接口采用 MINI USB 接口，这个 USB 接口即实现了供电功能，有可以实现 USB 转串口功能，可以用一根 USB 线将它连接到上 PC 的 USB 口进行串口数据通信 。

程序需要 2 个时钟，一个是默认的 50MHz 时钟，一个是用来通信的时钟(clk_bps)，这个时钟保持 9600Hz 的频率，接收 USB 端口上 9600 波特率的信号，程序持续读取 USB 口传来的信息，直到遇到连续的 1100 脉冲，代表传输开始。发送的数据帧为 1 位起始位，8 位数据位，无校验位，1 位停止位。所以当接收模块收到下降沿信号时，表示读到起始位，发送 bps_start 信号使波特率控制模块开始计数。当读到 9 次 bps_clk 信号的高电平时，表示一次数据接收完毕，bps_start 置零，等待下一次接收。



该接收模块中设置了接收两次数据后，第一次接收到的数据放到 addr 输出端口，第二次接收到的数据放到 ram_in 输出端口，再将 wr 置 1，等待存储器从该接收模块读取数据。由于存储器在 wr 为 且时钟信号为上升沿时读入数据，所以输入模块的等待时间需要大于 CPU 模块时钟信号源所产生的一个信号周期，才能确保输入的数据被存储器成功读入。

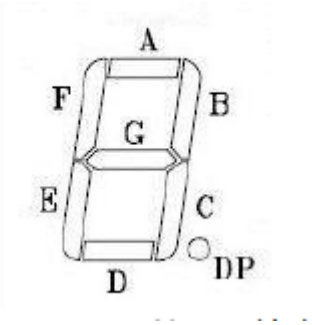
因为接收模块接收两次数据要等待存储器读入，所以 PC 上向串口发送数据时每发送两次需要等待接收模块就绪之后才能再次发送，以确保 PC 发送的信息能够被接收模

块完整接收。数据以 2 个 8 位数（8 位地址，8 位数据）的格式发给输入缓存，在下一个时钟周期写入。

CPU 可以通过访问 0xFFFF0000 到 0xFFFFFFFF 之间的地址(保留段，这里接 IO 设备)访问输入缓存。

2.3.2 数码管显示 16 进制数

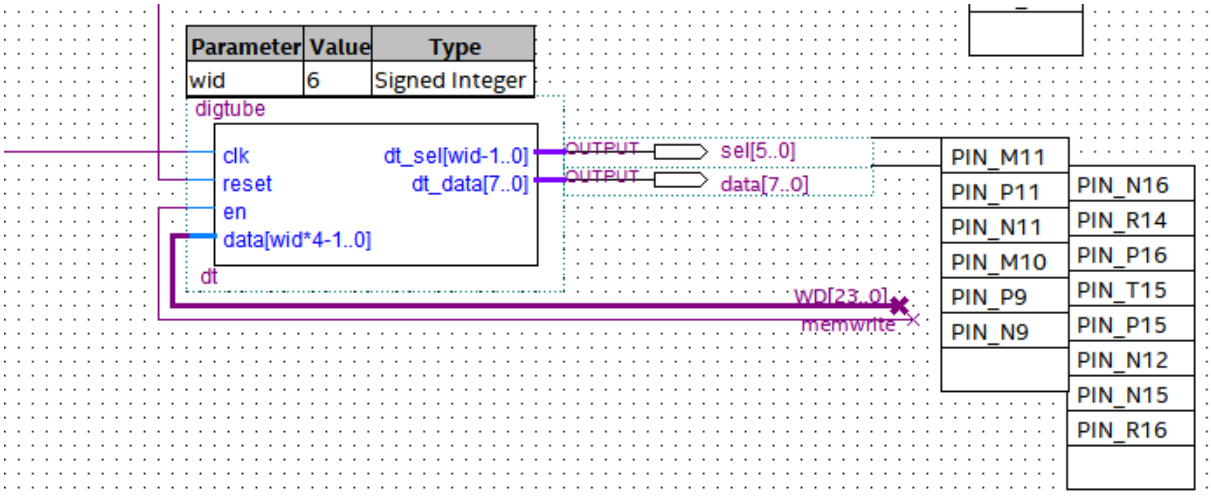
数码管是很常见的一种显示设备，一般分为七段数码管和八段数码管，两者区别就在于八段数码管比七段数码管多了一个“点”。开发板是 6 位一体的八段数码管，如图。



AX301 使用的是共阳极数码管，当某一字段对应的引脚为低电平时，相应字段就点亮，当某一字段对应的引脚为高电平时，相应字段就不亮。

六位一体数码管是属于动态显示，由于人的视觉暂留现象及发光二极管的余辉效应，尽管实际上各位数码管并非同时点亮，但只要扫描的速度足够快，给人的印象就是一组稳定的显示数据，不会有闪烁感。

六位一体数码管的相同的段都接在了一起，一共是 8 个引脚，然后加上 6 个控制信号引脚，一共是 14 个引脚，其中 DIG[0..7]是对应数码管的 A,B,C,D,E,F,G,H(即点 DP)；SEL[0..5]是六个数码管的六个控制引脚，也是低电平有效，当控制引脚为低电平时，对应的数码管有了供电电压，这样数码管才能点亮，否则无论数码管的段如何变化，也不能点亮对应的数码管。



数码管译码

数码管没有提供直接显示 16 进制数的功能，而是需要将 16 进制数译码，将译码后的结果显示到数码管上。

译码表

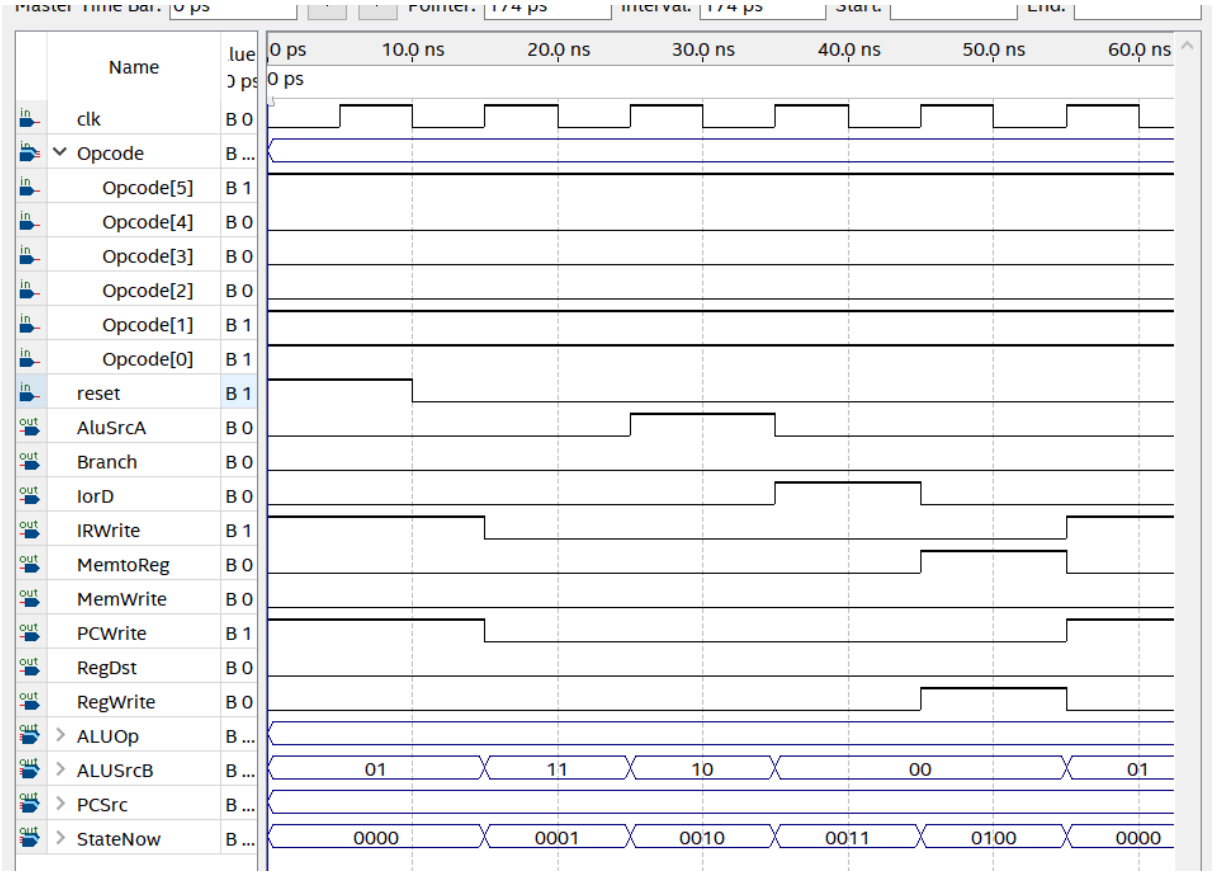
16 进制数	2 进制	译码后
0	0000	00000011
1	0001	10011111
2	0010	00100101
3	0011	00001101
4	0100	10011001
5	0101	01001001
6	0110	01000001
7	0111	00011111
8	1000	00000001
9	1001	00011001
A	1010	00010001
B	1011	11000001
C	1100	01100011
D	1101	10000101
E	1110	01100001
F	1111	01110001

输出时，根据 CLK 计数，选择当前数的[4n,4n+4)位，译码后输出到 DIG 引脚上，并使 SEL[n]为 1，其他为 0，这样只要足够快，就可以看起来正常显示数字。

第 3 章 系统仿真

3.1 控制器仿真

展示的是控制器在输入 Op 为 100011(LW)时控制器的输出。



3.2 CPU 仿真

整体仿真时在内存中事先写入了一段程序（为方便可能和标准有些差距）：

```
mem(0):="00001000";
mem(1):=x"00";
mem(2):=x"00";
mem(3):=x"03"; --j 3

mem(12):="00000000";
mem(13):="00000000";
mem(14):="00001000";
mem(15):="00100110";      --xor $1,$0,$0
mem(16):="10001100";
mem(17):="00000001";
mem(18):=x"00";
mem(19):=x"2A";          --lw $1,2Ah($0)
```

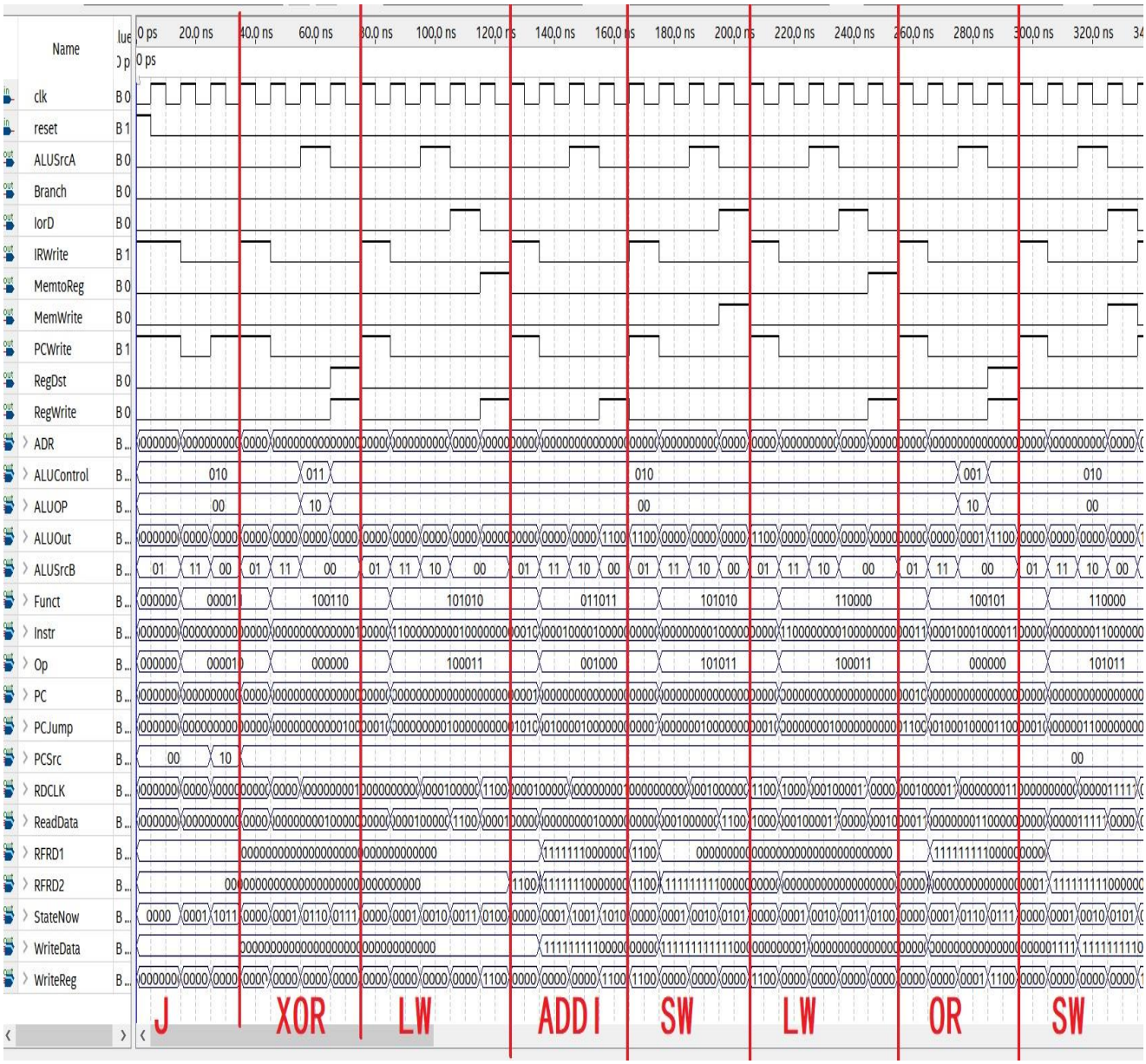
```
mem(20):="00100000";
mem(21):="00100001";
mem(22):=x"00";
mem(23):=x"1B";          --addi $1,$1,1Bh
mem(24):="10101100";
mem(25):="00000001";
mem(26):=x"00";
mem(27):=x"2A";          --sw $1,2Ah($0)
mem(28):="10001100";
mem(29):="00000010";
mem(30):=x"00";
mem(31):=x"30";          --lw $2,30h($0)
mem(32):="00000000";
mem(33):="00100010";
mem(34):="00011000";
mem(35):="00100101";     --or $3,$1,$2
mem(36):="10101100";
mem(37):="00000011";
mem(38):=x"00";
mem(39):=x"30";          --sw $3,30h($0)

mem(42):="11111111";
mem(43):="11111111";     --test data (2Ah)
mem(44):="00000000";
mem(45):="00000000";

mem(48):="00000000";
mem(49):="00000000";
mem(50):="00000000";
mem(51):="11111111";     --test data (30h)
```

该程序首先无条件跳转，之后将 2A 处的数据取出，与立即数 1B 相加，存入 2Ah

再将 30h 的数据取出，与之前的结果进行 OR 操作，存入 30h。

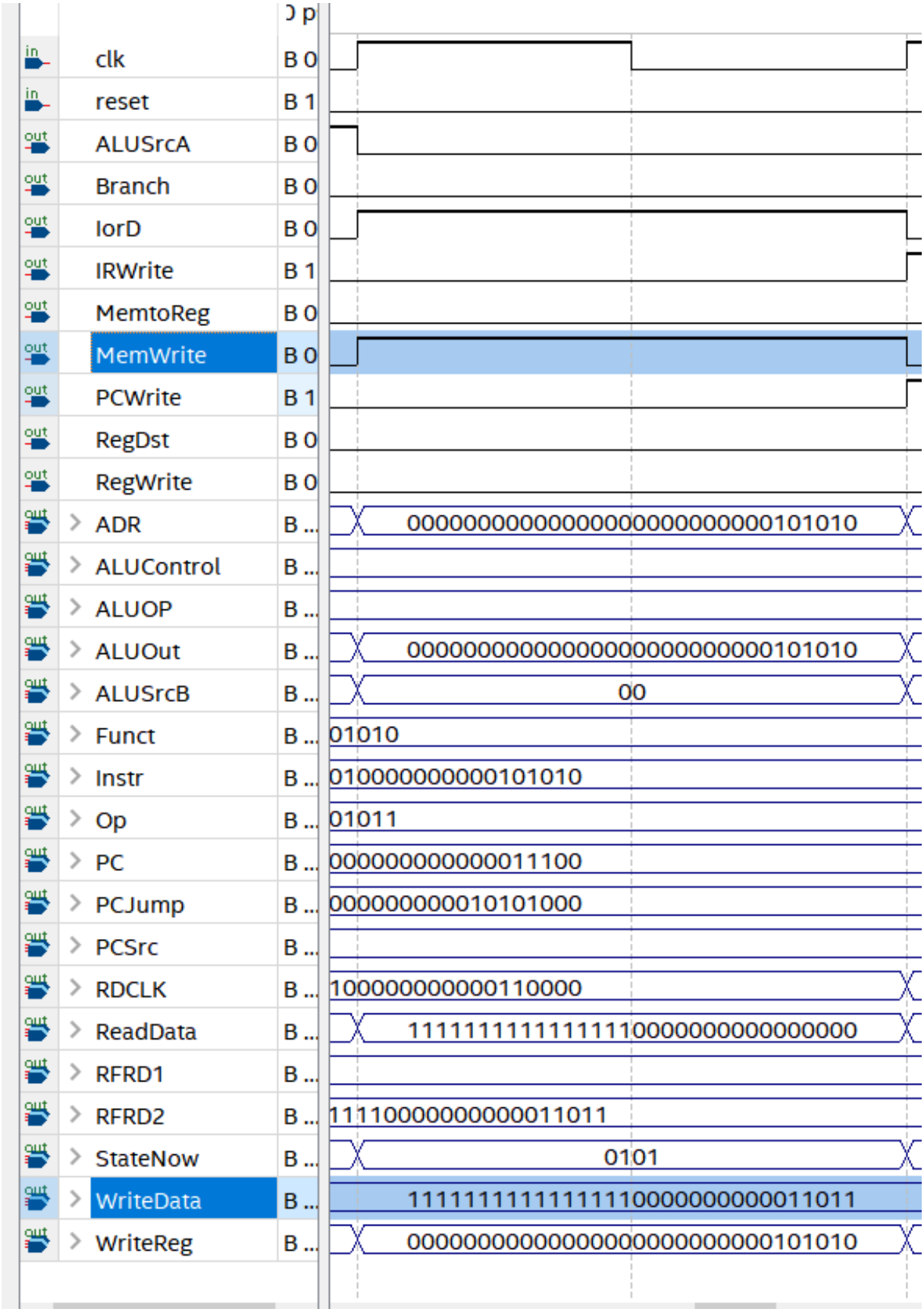
若能正确运行，两次 SW 的结果应该为 FFFF001B 和 FFFF00FF。

仿真结果如下

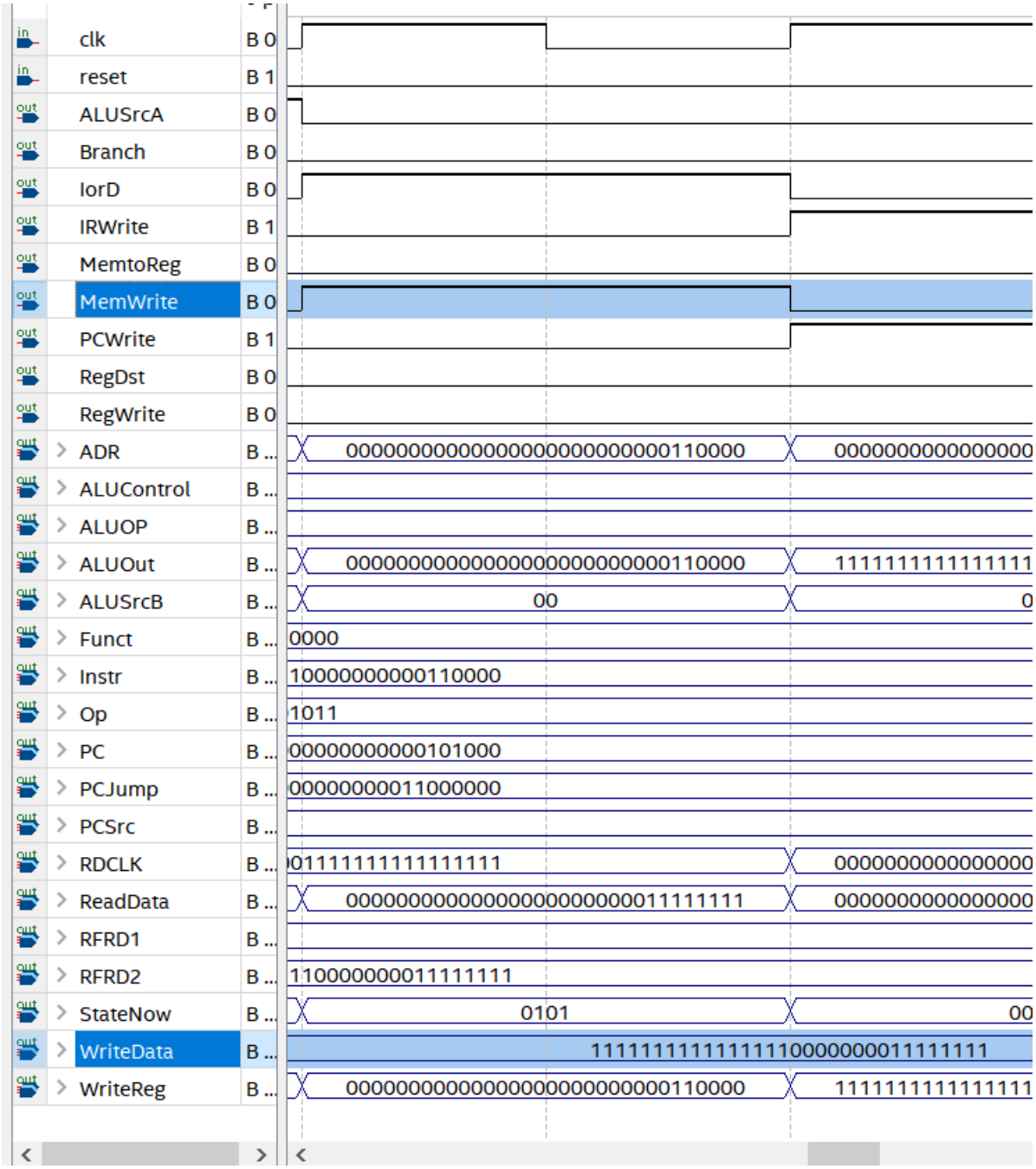


为验证结果，只需要查看两次 SW 的值。

第一次 SW，可以看出存储了 FFFF001B，是 FFFF0000+1B



第二次 SW，可以看出存储了 FFFF001B OR 000000FF



得出结论：结果正确。

第 4 章 系统实现

4.1 系统下载验证

默认程序为读入 0xFFFF0001,0xFFFF0002,0xFFFF0003 的数据至$1,$2,$3，并判断$3,若为 0，则执行加法后存入内存，若为 1，则执行减法后存入内存(此行为会更新数码管的值)，最后跳转回程序开始处。

代码如下

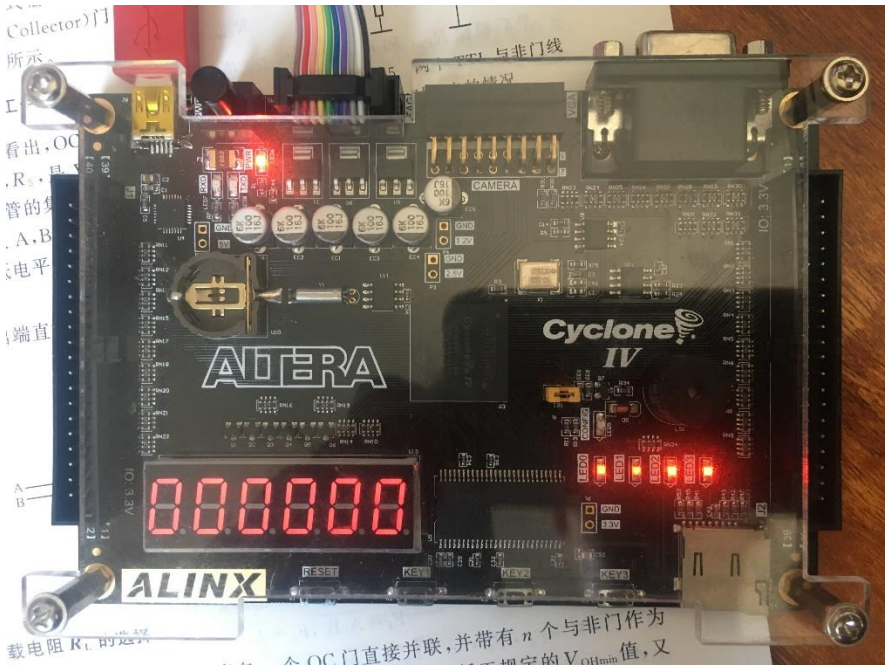
```
mem(0):=        "10001100";--lw $1,60($0)
mem(1):=        "00000001";
mem(2):=        "00000000";
mem(3):=        "00111100";
mem(4):=        "00100000";--addi $2,$1,4
mem(5):=        "00100010";
mem(6):=        "00000000";
mem(7):=        "00000100";
mem(8):=        "00100000";--addi $3,$2,4
mem(9):=        "01000011";
mem(10):=       "00000000";
mem(11):=       "00000100";
mem(12):=       "10001100";--lw $1,0($1)
mem(13):=       "00100001";
mem(14):=       "00000000";
mem(15):=       "00000000";
mem(16):=       "10001100";--lw $2,0($2)
mem(17):=       "01000010";
mem(18):=       "00000000";
mem(19):=       "00000000";
mem(20):=       "10001100";--lw $3,0($3)
mem(21):=       "01100011";
mem(22):=       "00000000";
mem(23):=       "00000000";
mem(24):=       "00100000";--addi $4,$0,0
mem(25):=       "00000100";
mem(26):=       "00000000";
mem(27):=       "00000000";
mem(28):=       "00010000";--beq $4,$3 add_pos
mem(29):=       "10000011";
mem(30):=       "00000000";
mem(31):=       "00000010";
mem(32):=       "00100000";--addi $4,$0,1
mem(33):=       "00000100";
```

```
mem(34):=       "00000000";
mem(35):=       "00000001";
mem(36):=       "00010000";--beq $4,$3 sub_pos
mem(37):=       "10000011";
mem(38):=       "00000000";
mem(39):=       "00000010";
mem(40):=       "00000000";--add $5,$1,$2
mem(41):=       "00100010";
mem(42):=       "00101000";
mem(43):=       "00100000";
mem(44):=       "00010000";--beq $0,$0 sav_pos
mem(45):=       "00000000";
mem(46):=       "00000000";
mem(47):=       "00000001";
mem(48):=       "00000000";--sub $5,$1,$2
mem(49):=       "00100010";
mem(50):=       "00101000";
mem(51):=       "00100010";
mem(52):=       "10101100";--sw $5,64($0)
mem(53):=       "00000101";
mem(54):=       "00000000";
mem(55):=       "01000000";
mem(56):=       "00001000";--j 0
mem(57):=       "00000000";
mem(58):=       "00000000";
mem(59):=       "00000000";
mem(60):=       "11111111";--ffff0001
mem(61):=       "11111111";
mem(62):=       "00000000";
mem(63):=       "00000001";
```

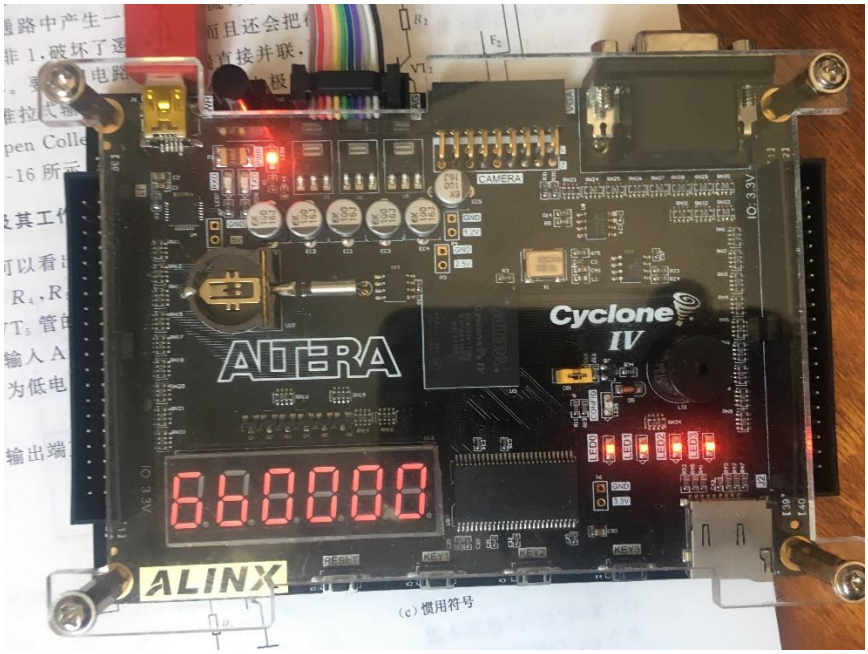
0xFFFF0000~0xFFFFFFFF 地址为串口输入，程序读入的数据为串口输入的数据。所以这个程序可以动态的执行两个数的加法或减法。

下面展示执行 6B4DCF-483ABD 的输入过程。

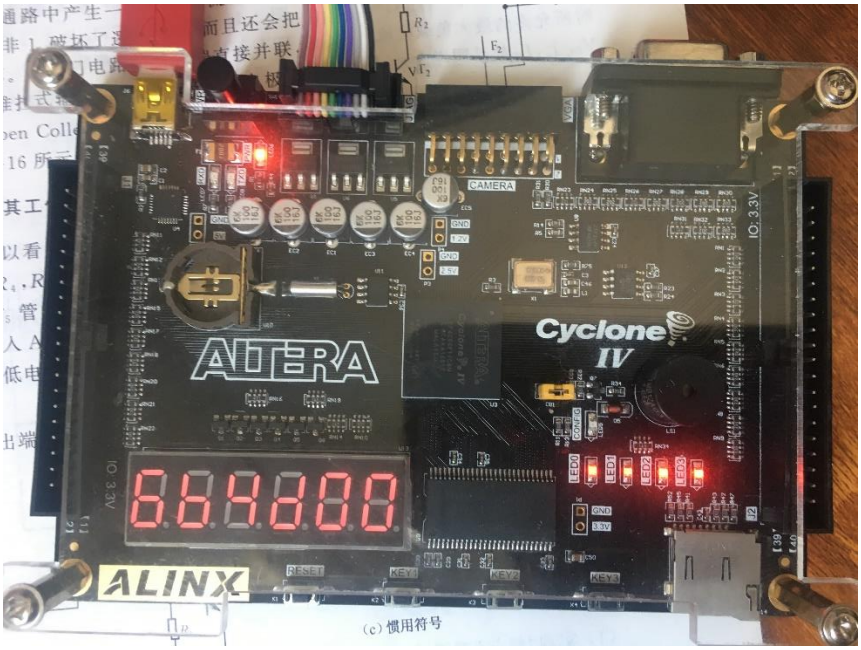
连接开发板后，打开”串口调试小助手”，向开发板发出 0c01,代表执行减法



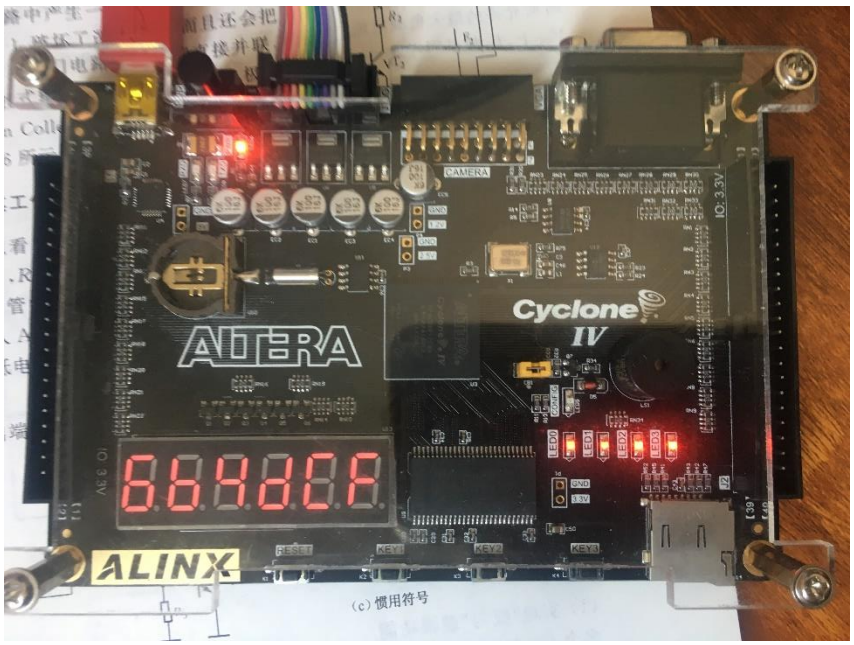
之后编辑第一个数，发送 026B


ALTERA
Cyclone IV
ALINX
CAMERA
RESET
KEY1
KEY2
KEY3
IO 3.3V
(c) 惯用符号

034D

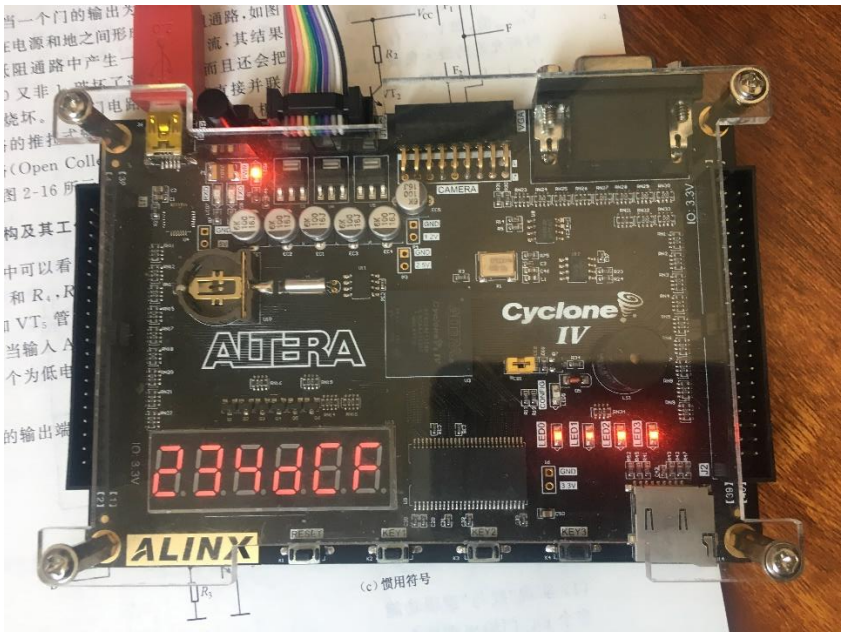

ALTERA
Cyclone IV
ALINX
CAMERA
RESET
KEY1
KEY2
KEY3
IO 3.3V
(c) 惯用符号

04CF

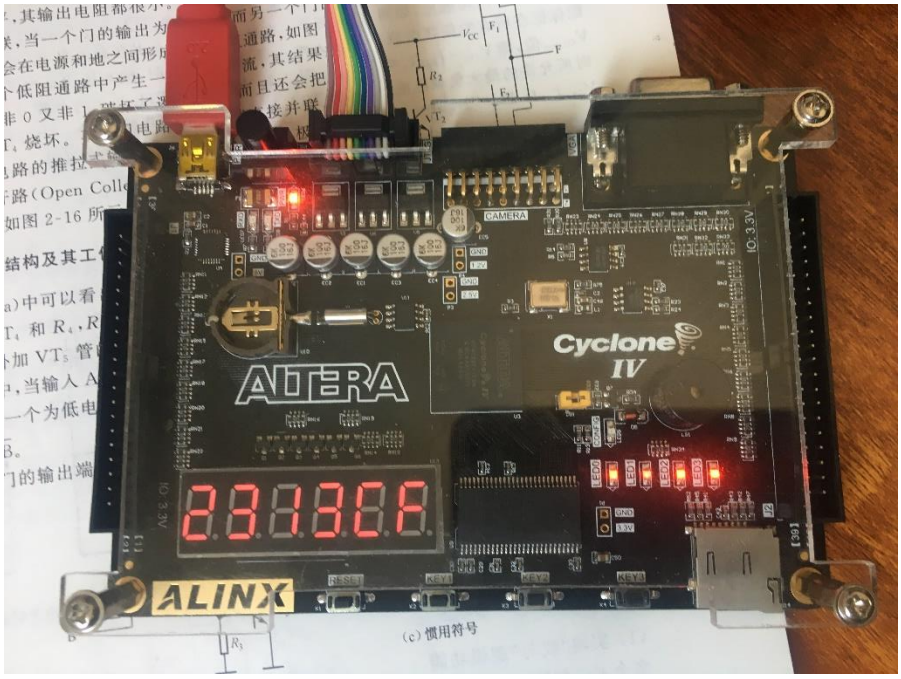


第二个数：

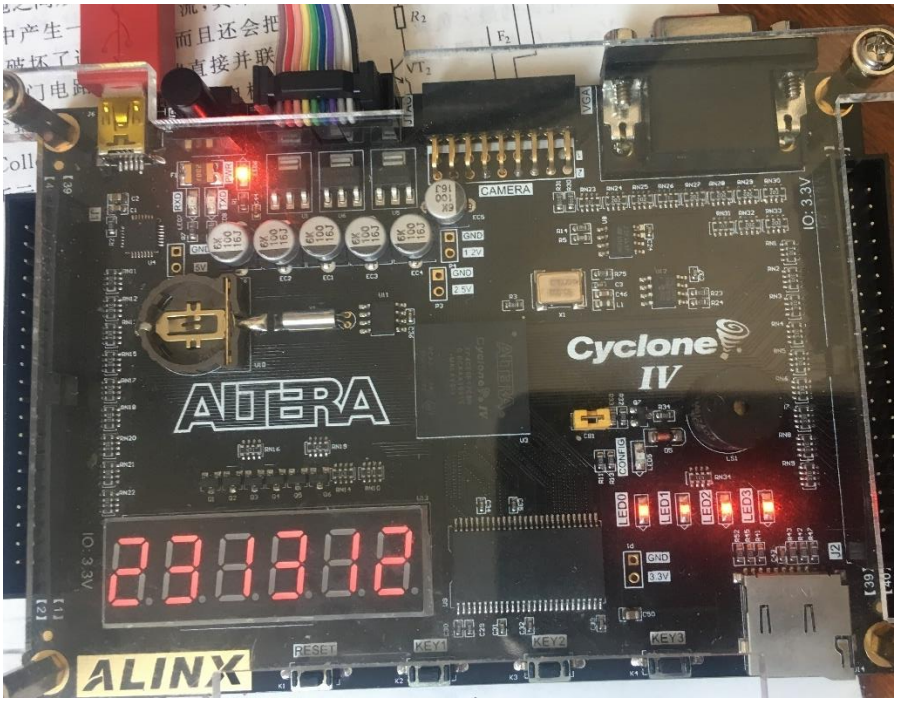
0648



073A



08BD



这样就可以看到结果了

可以心算一下，结果是正确的。