组成原理实验课程第 6 次实验报告

实验名称	单周期 CPU 实现			班级	李涛
学生姓名	胡博浩	学号	2212998	指导老师	董前琨
实验地点	津南实验楼 A308		实验时间	2024.6.6	

1、 实验目的

（1）理解 MIPS 指令结构，理解 MIPS 指令集中常用指令的功能和编码，学会对这些指令进行归纳分类。

（2）了解熟悉 MIPS 体系的处理器结构，如延迟槽，哈佛结构的概念。

（3）熟悉并掌握单周期 CPU 的原理和设计。

（4）进一步加强运用 verilog 语言进行电路设计的能力。

（5）为后续设计多周期 cpu 的实验打下基础。

2、 实验内容说明

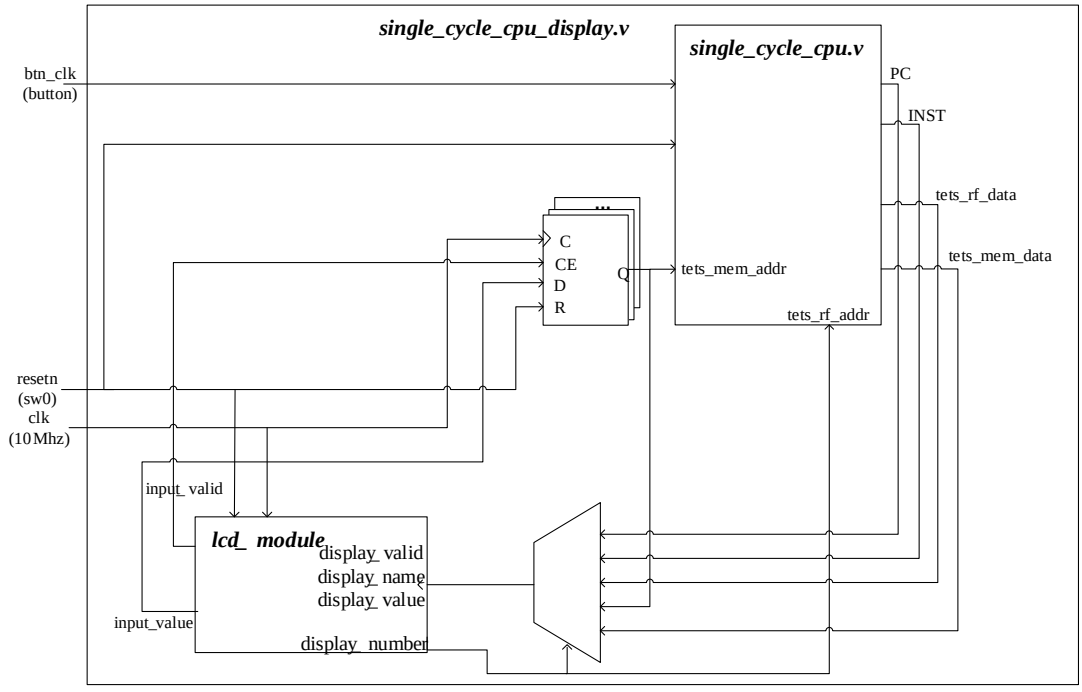
请结合实验指导手册中的实验六（单周期 CPU 实验）完成功能改进，在原有 CPU 基础上，扩充 CPU 可运行的 MIPS 指令，注意以下几点：

1、扩充的指令应为一个时钟周期内能够执行完的指令，要求至少一个 R 型，一个 I 型，另外一个自选。建议在 ALU 实验改进基础上补充。

2、实验报告中原理图为指导手册中的 display 模块图，不用修改，报告中的内容和展示的结果应扩充指令的步骤和实验结果。

3、本次实验报告需要有实验箱上箱验证的照片，同样，针对照片中的数据需要解释说明。若只有仿真波形结果，会适当扣分。

实验原理图



本次实验的改进部分扩充了三个新的指令：**hui---低位加载、nxor---按位同或、blt---有符号大于则置位，均为上次 ALU 实验添加的指令**。其中 hui 是 I 型指令，nxor 和 blt 是 R 型指令。

3、 实验步骤

首先说明一下我在单周期中指令的改变。这次实验我选择添加三条汇编指令，放在跳转指令之前执行。为了方便验证，我选择放在 19~21 的位置，并借助前面赋值好的寄存器。三条汇编指令如下：

---

hui $13,#13

nxor $14 ,$7,$6

blt $15 ,$7,$6

R 型指令的二进制编码格式如下：

op	rs	rt	rd	shamd	funct
000000	5 位	5 位	5 位	00000	6 位

I 型指令的二进制编码格式如下：

op	rs	rt	immediate or address
6 位	5 位	5 位	16 位

根据三条汇编指令，以及我自定义的 funct、op 字段，转为的编码如下：

R 型指令	op	rs	rt	rd	shamd	funct	十六进制编码
nxor $14 ,$7,$6	000000	00111	00110	01110	00000	110001	00E67031
blt $15 ,$7,$6	000000	00111	00110	01111	00000	110011	00E67833

I 型指令	op	rs	rt	immediate or address	十六进制编码
hui $13,#13	110000	00000	01101	0000000000001101	C00D000D

接下来对代码进行修改，本次实验需要修改三个文件，inst_rom.v、alu.v 和 single_cycle_cpu.v 文件。

（1）inst_rom.v

a）修改指令存储器 inst_rom 的位宽，由于添加了三条指令，**改为 23 位位宽**（一开始我漏了。。。花了蛮长时间才检查出来）

```
wire [31:0] inst_rom[22:0];  // 指令存储器，字节地址7'b000_0000~7'b111_1111，改为23位
```

b）在 inst_rom 中添加三条指令,inst_rom[19]、inst_rom[20]、inst_rom[21]，把原来在 inst_rom[19]的跳转指令改为 inst_rom[22]

```
assign inst_rom[19] = 32'hC00D000D; // 4CH: hui   $13,#13 ,    | [R13] = 0000_000DH
assign inst_rom[20] = 32'h00E67031; // 50H: nxor  $14 ,$7,$6   | $14 = FFFF_FFEEH
assign inst_rom[21] = 32'h00E67833; // 54H: blt   $15 ,$7,$6   | $15 = 0000_0001H

assign inst_rom[22] = 32'h08000000; // 58H: j     00H          | 跳转指令00H
```

c）补充对应 inst_rom[20]、inst_rom[21]、inst_rom[22]的输出

```
5'd19: inst <= inst_rom[19];
5'd20: inst <= inst_rom[20];//添加三个输出
5'd21: inst <= inst_rom[21];
5'd22: inst <= inst_rom[22];
default: inst <= 32'd0;
```

（2）alu.v

这个文件的修改仿照之前 alu 实验中的 alu.v 文件

a）修改 alu_control 的位宽，改为 15 位

```
input  [14:0] alu_control,  // ALU控制信号，改为15位
```

b）添加三个独热码

```
wire alu_hui;//低位加载
wire alu_nxor;//按位同或
wire alu_blt;//大于则置位
```

c）添加三个控制信号

```
assign alu_blt  = alu_control[14];//添加控制信号
assign alu_nxor = alu_control[13];
assign alu_hui  = alu_control[12];
```

d）添加三个结果信号，用于存储运算结果

```
wire [31:0] hui_result;//添加结果信号
wire [31:0] nxor_result;
wire [31:0] blt_result;
```

e）编写三个新运算的代码

对于按位同或，直接把异或结果取反就行

```
assign nxor_result= ~xor_result;//同或结果为异或取反
```

对于低位加载，和高位加载相反

```
assign hui_result = {16'd0, alu_src2[15:0]};//立即数装载结果为立即数移位至低半字节
```

对于有符号比较，大于则置位操作，利用真值表、并仿照小于置位操作编写

```
//blt结果
assign blt_result[31:1] = 31'd0;
assign blt_result[0]    = (~alu_src1[31] & alu_src2[31]) | (~(alu_src1[31]^alu_src2[31]) & ~adder_result[31] &(adder_result!=32'b0));
```

f）把运算结果添加到最后的结果集中

```
alu_lui          ? lui_result :
alu_hui          ? hui_result ://低位加载
alu_nxor         ? nxor_result ://按位同或
alu_blt          ? blt_result ://大于则置位
32'd0;
```

3）single_cycle_cpu.v

首先在译码部分进行修改

a）将扩充的三个指令添加到指令列表中

```
// 实现指令列表
wire inst_ADDU, inst_SUBU , inst_SLT, inst_AND;
wire inst_NOR , inst_OR   , inst_XOR, inst_SLL;
wire inst_SRL , inst_ADDIU, inst_BEQ, inst_BNE;
wire inst_LW  , inst_SW   , inst_LUI, inst_J;

wire inst_HUI , inst_NXOR , inst_BLT;//添加三条指令，低位加载、按位同或、大于则置位
```

b）将添加的两个操作赋予操作数

```
assign inst_J    = (op == 6'b000010);                    // 直接跳转

assign inst_HUI  = (op == 6'b110000);//低位加载
assign inst_NXOR = op_zero & sa_zero    & (funct == 6'b110001);//按位同或
assign inst_BLT  = op_zero & sa_zero    & (funct == 6'b110011);//大于则置位
```

c）定义并实现传递到执行模块的 ALU 源操作数和操作码

```
wire inst_hui, inst_nxor,inst_blt;//添加三个

assign inst_hui = inst_HUI;//低位加载
assign inst_nxor= inst_NXOR;//按位同或
assign inst_blt = inst_BLT;//大于则置位
```

d）修改立即数扩展指令，添加 I 型指令低位加载

```
assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LW | inst_SW | inst_HUI;//对I型指令低位加载的立即数进行扩展
```

e）修改 alu_control 的独热编码

首先将 alu_control 改为 15 位（一开始没发现。。。）

```
wire [14:0] alu_control;//改为15位
```

然后添加三个独热编码**（注意要和 alu.v 中定义的 alu_control 指令顺序相同!!!）**

```
assign alu_control = {inst_blt,//添加三个独热编码
                      inst_nxor,
                      inst_hui,
                      inst_add,          // ALU操作码，独热编码
                      inst_sub,
                      inst_slt,
```

最后在写回部分，对应添加三条指令的：I 型指令低位加载写回 rt，R 型指令按位同或、大于则置位写回 rd

```
wire inst_wdest_rt;   // 寄存器堆写入地址为rt的指令
wire inst_wdest_rd;   // 寄存器堆写入地址为rd的指令
assign inst_wdest_rt = inst_ADDIU | inst_LW | inst_LUI | inst_HUI;//I型指令HUI写回rt
assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT | inst_AND | inst_NOR
                     | inst_OR   | inst_XOR  | inst_SLL | inst_SRL | inst_NXOR | inst_BLT;//R型指令NXOR、BLT写回rd
```

4、 实验结果分析

最后经过改进后单周期的指令如下：

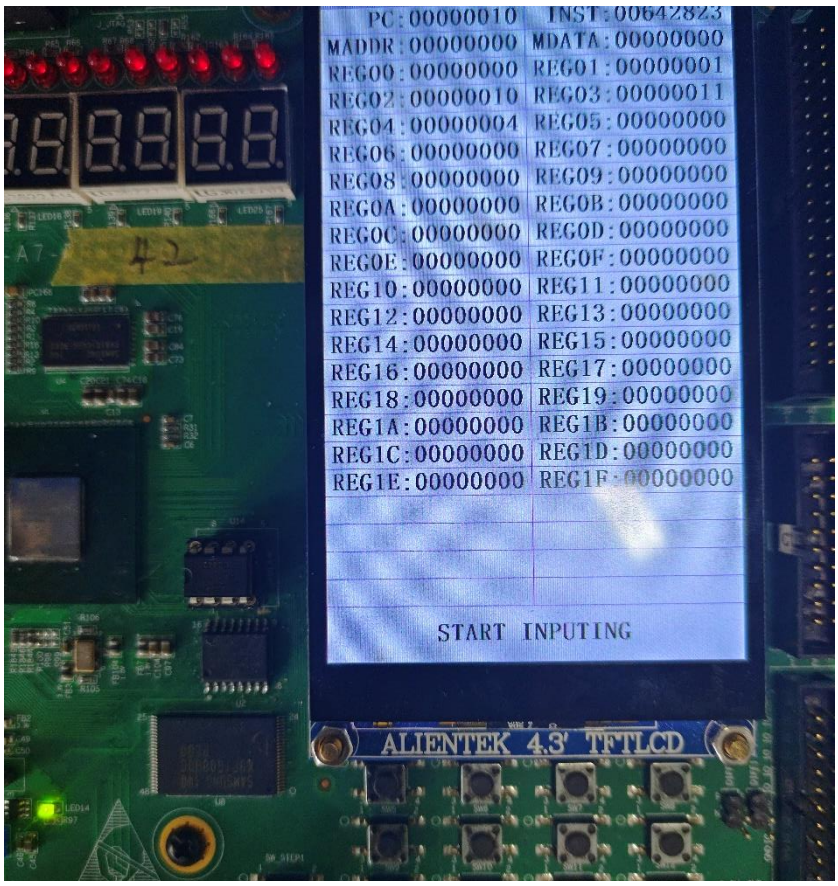
```
//———————— 指令编码 ————————|指令地址|—— 汇编指令 ——|- 指令结果 ——//
assign inst_rom[ 0] = 32'h24010001; // 00H: addiu $1 ,$0,#1    | $1 = 0000_0001H
assign inst_rom[ 1] = 32'h00011100; // 04H: sll   $2 ,$1,#4    | $2 = 0000_0010H
assign inst_rom[ 2] = 32'h00411821; // 08H: addu  $3 ,$2,$1    | $3 = 0000_0011H
assign inst_rom[ 3] = 32'h00022082; // 0CH: srl   $4 ,$2,#2    | $4 = 0000_0004H
assign inst_rom[ 4] = 32'h00642823; // 10H: subu  $5 ,$3,$4    | $5 = 0000_000DH
assign inst_rom[ 5] = 32'hAC250013; // 14H: sw    $5 ,#19($1)  | Mem[0000_0014H] = 0000_000DH
assign inst_rom[ 6] = 32'h00A23027; // 18H: nor   $6 ,$5,$2    | $6 = FFFF_FFE2H
assign inst_rom[ 7] = 32'h00C33825; // 1CH: or    $7 ,$6,$3    | $7 = FFFF_FFF3H
assign inst_rom[ 8] = 32'h00E64026; // 20H: xor   $8 ,$7,$6    | $8 = 0000_0011H
assign inst_rom[ 9] = 32'hAC08001C; // 24H: sw    $8 ,#28($0)  | Mem[0000_001CH] = 0000_0011H
assign inst_rom[10] = 32'h00C7482A; // 28H: slt   $9 ,$6,$7    | $9 = 0000_0001H
assign inst_rom[11] = 32'h11210002; // 2CH: beq   $9 ,$1,#2    | 跳转到指令34H
assign inst_rom[12] = 32'h24010004; // 30H: addiu $1 ,$0,#4    | 不执行
assign inst_rom[13] = 32'h8C2A0013; // 34H: lw    $10,#19($1)  | $10 = 0000_000DH
assign inst_rom[14] = 32'h15450003; // 38H: bne   $10,$5,#3    | 不跳转
assign inst_rom[15] = 32'h00415824; // 3CH: and   $11,$2,$1    | $11 = 0000_0000H
assign inst_rom[16] = 32'hAC0B001C; // 40H: sw    $11,#28($0)  | Men[0000_001CH] = 0000_0000H
assign inst_rom[17] = 32'hAC040010; // 44H: sw    $4 ,#16($0)  | Mem[0000_0010H] = 0000_0004H
assign inst_rom[18] = 32'h3C0C000C; // 48H: lui   $12,#12      | [R12] = 000C_0000H

assign inst_rom[19] = 32'hC00D000D; // 4CH: hui   $13,#13 ,    | [R13] = 0000_000DH
assign inst_rom[20] = 32'h00E67031; // 50H: nxor  $14 ,$7,$6    | $14 = FFFF_FFEEH
assign inst_rom[21] = 32'h00E67833; // 54H: blt   $15 ,$7,$6    | $15 = 0000_0001H

assign inst_rom[22] = 32'h08000000; // 58H: j     00H          | 跳转指令00H
```

上箱实验验证

a）运行至 PC=10，执行了 4 条指令

```
assign inst_rom[ 3] = 32'h00022082; // 0CH: srl    $4 ,$2,#2    | $4 = 0000_0004H
```

按照设定，此时 REG04=0000_0004

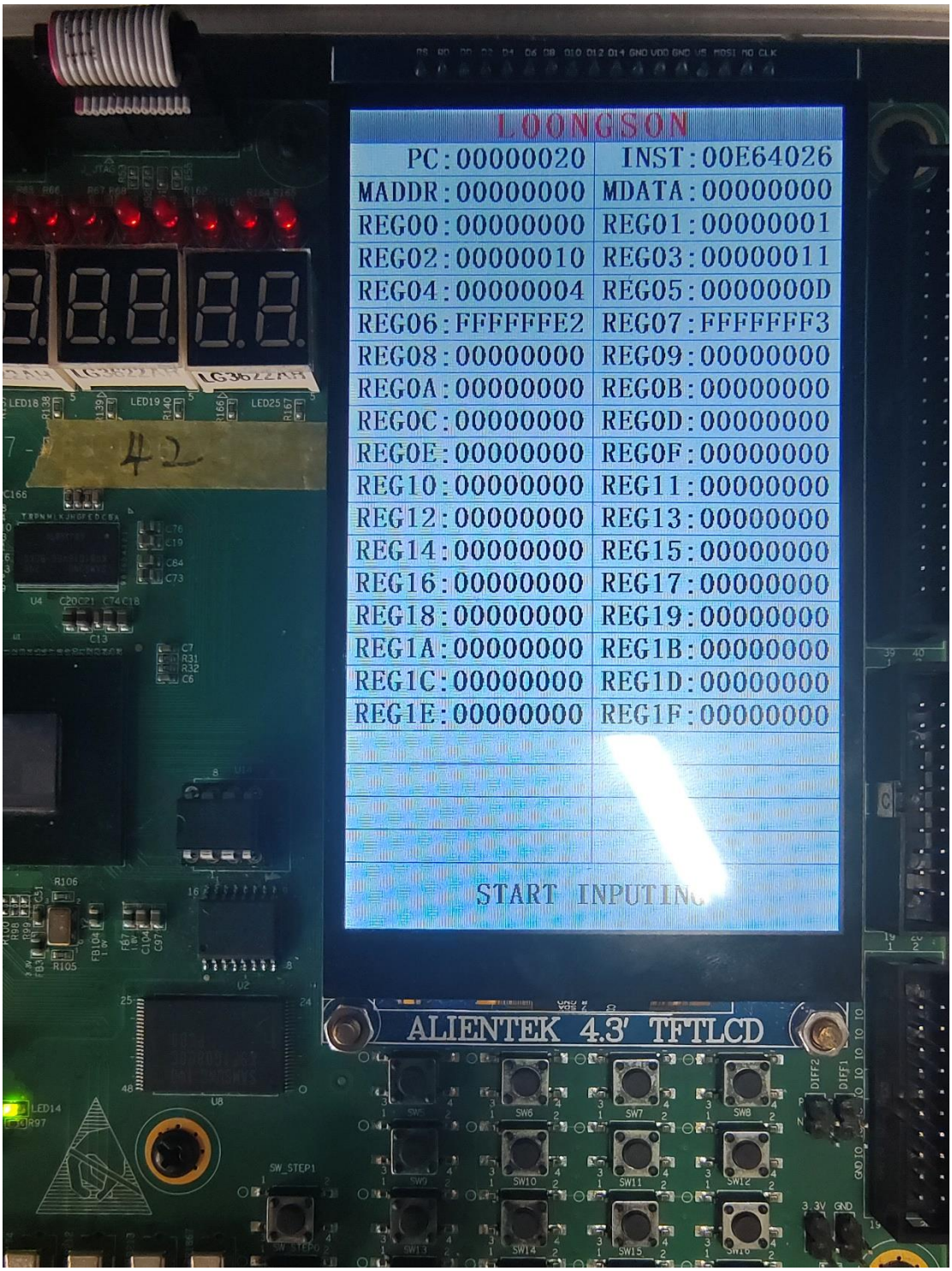


如图所示，结果正确

b）运行至 PC=20，执行了 8 条指令

```
assign inst_rom[ 7] = 32'h00C33825; // 1CH: or     $7 ,$6,$3    | $7 = FFFF FFF3H
```

按照设定，此时 REG07=FFFF_FFF3

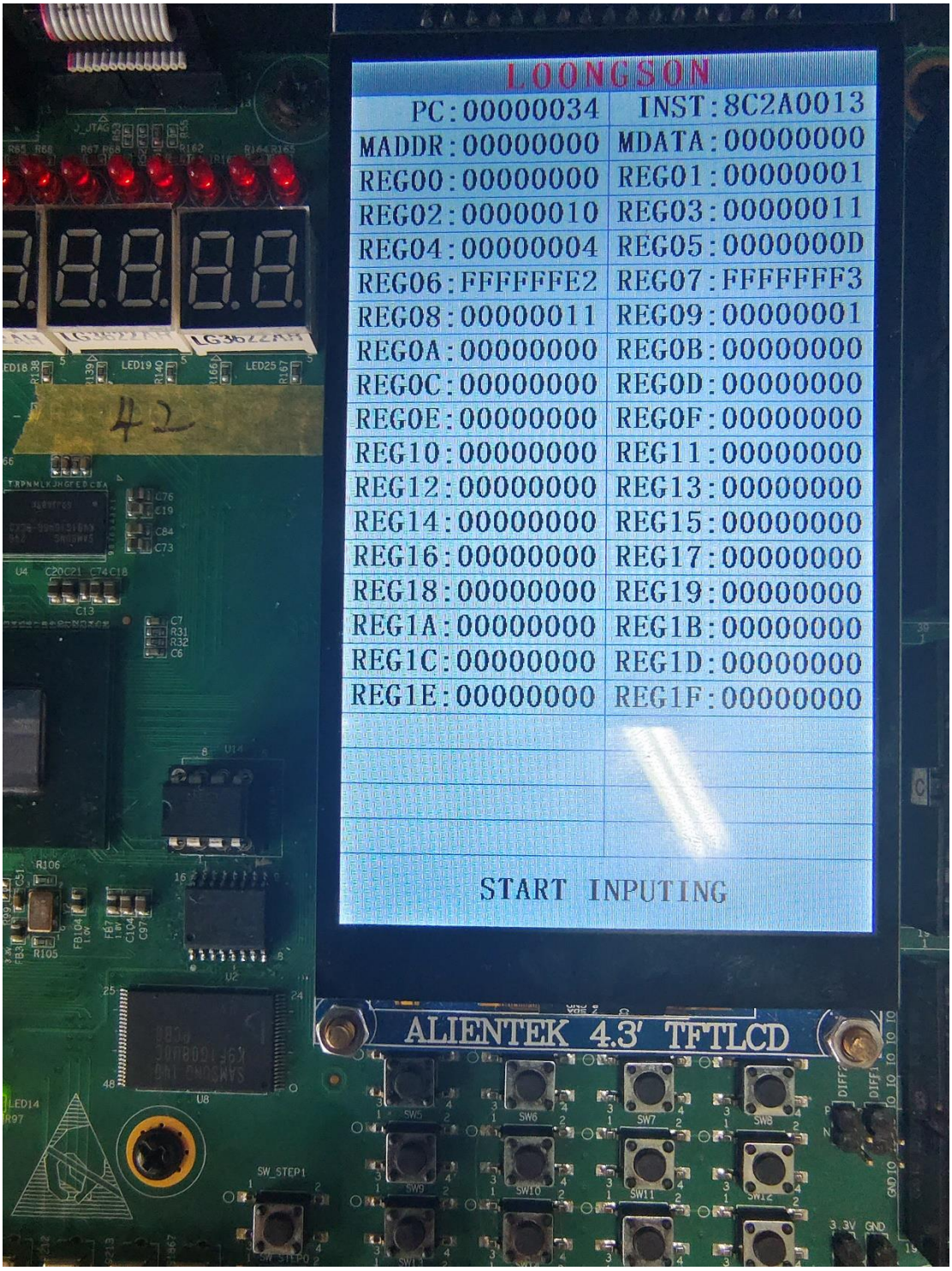


如图所示，结果正确

c）运行至 PC=34

```
assign inst_rom[11] = 32'h11210002; // 2CH: beq   $9 ,$1,#2    | 跳转到指令34H
assign inst_rom[12] = 32'h24010004; // 30H: addiu $1 ,$0,#4    | 不执行
assign inst_rom[13] = 32'h8C2A0013; // 34H: lw    $10,#19($1) | $10 = 0000_000DH
```

按照设定，此时 30H 指令不执行，REG01=0000_0001

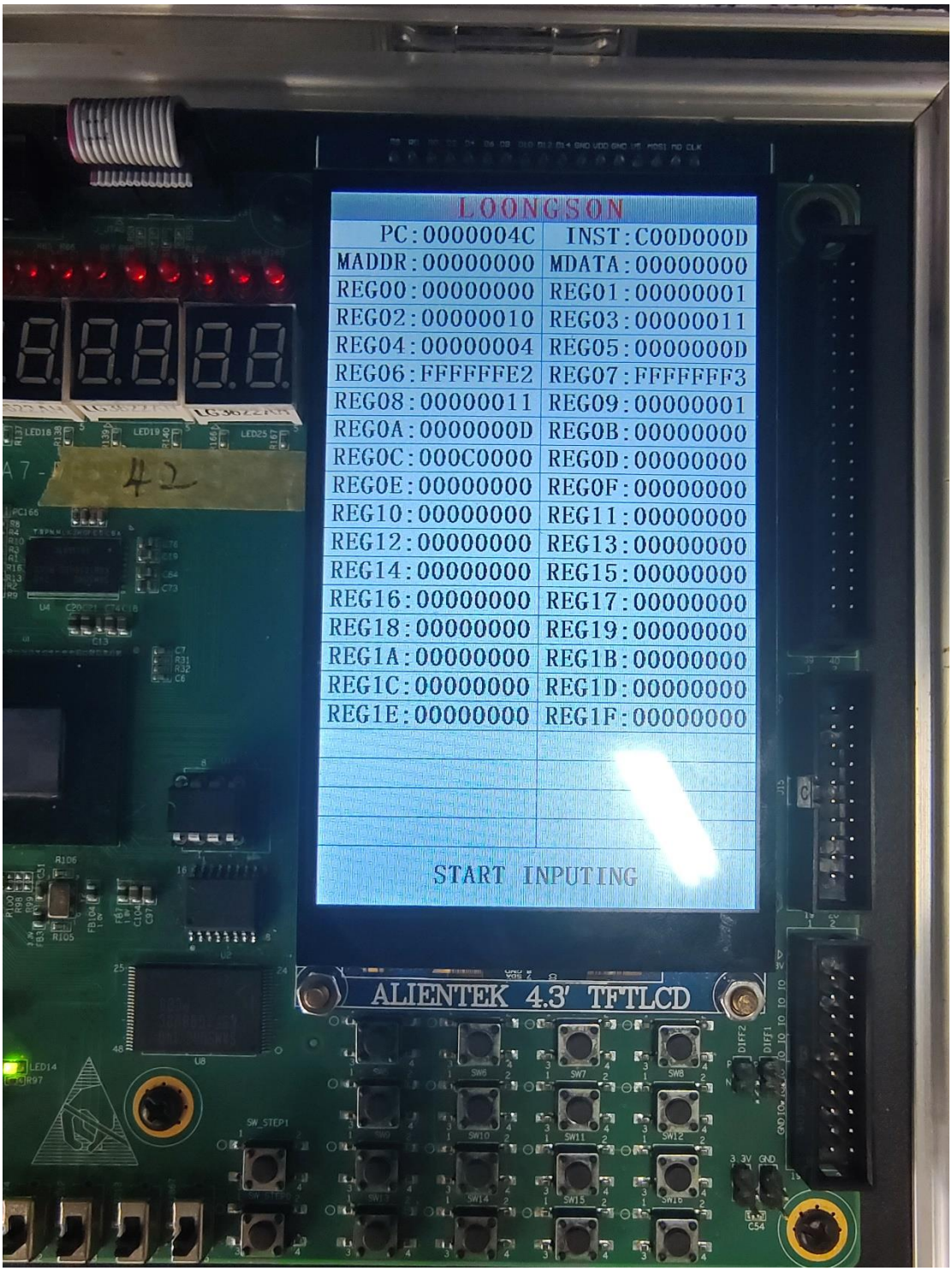


如图所示，结果正确

d）运行至 PC=4C

```
assign inst_rom[18] = 32'h3C0C000C; // 48H: lui   $12,#12      | [R12] = 000C_0000H
```

按照设定，此时 REG0C=000C_0000



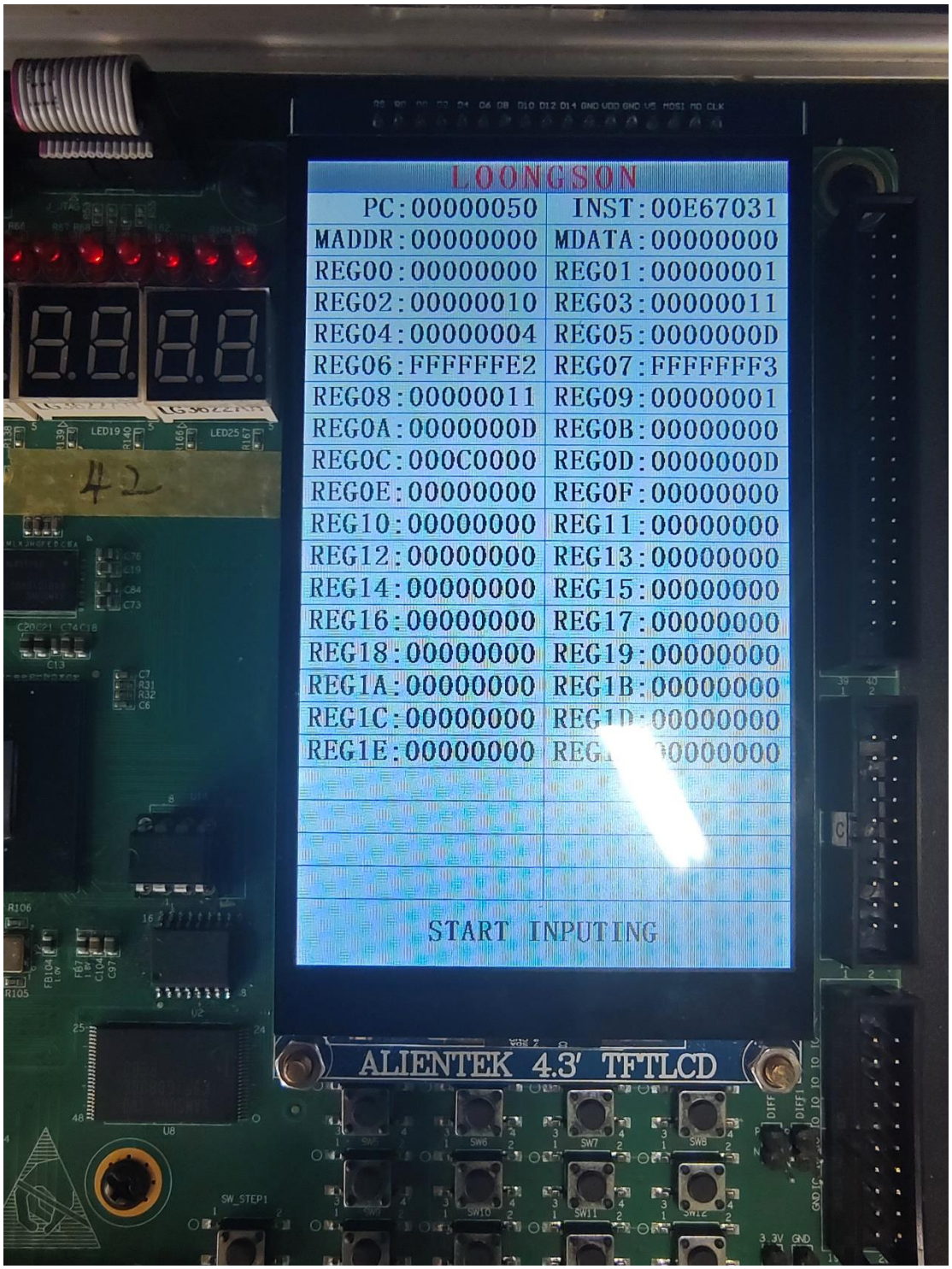
如图所示，结果正确

接下来验证添加的三条指令

1）运行至 PC=50

```
assign inst_rom[19] = 32'hC00D000D; // 4CH: hui   $13,#13 ,    | [R13] = 0000_000DH
```

按照设定，此时 REG0D=0000_000D

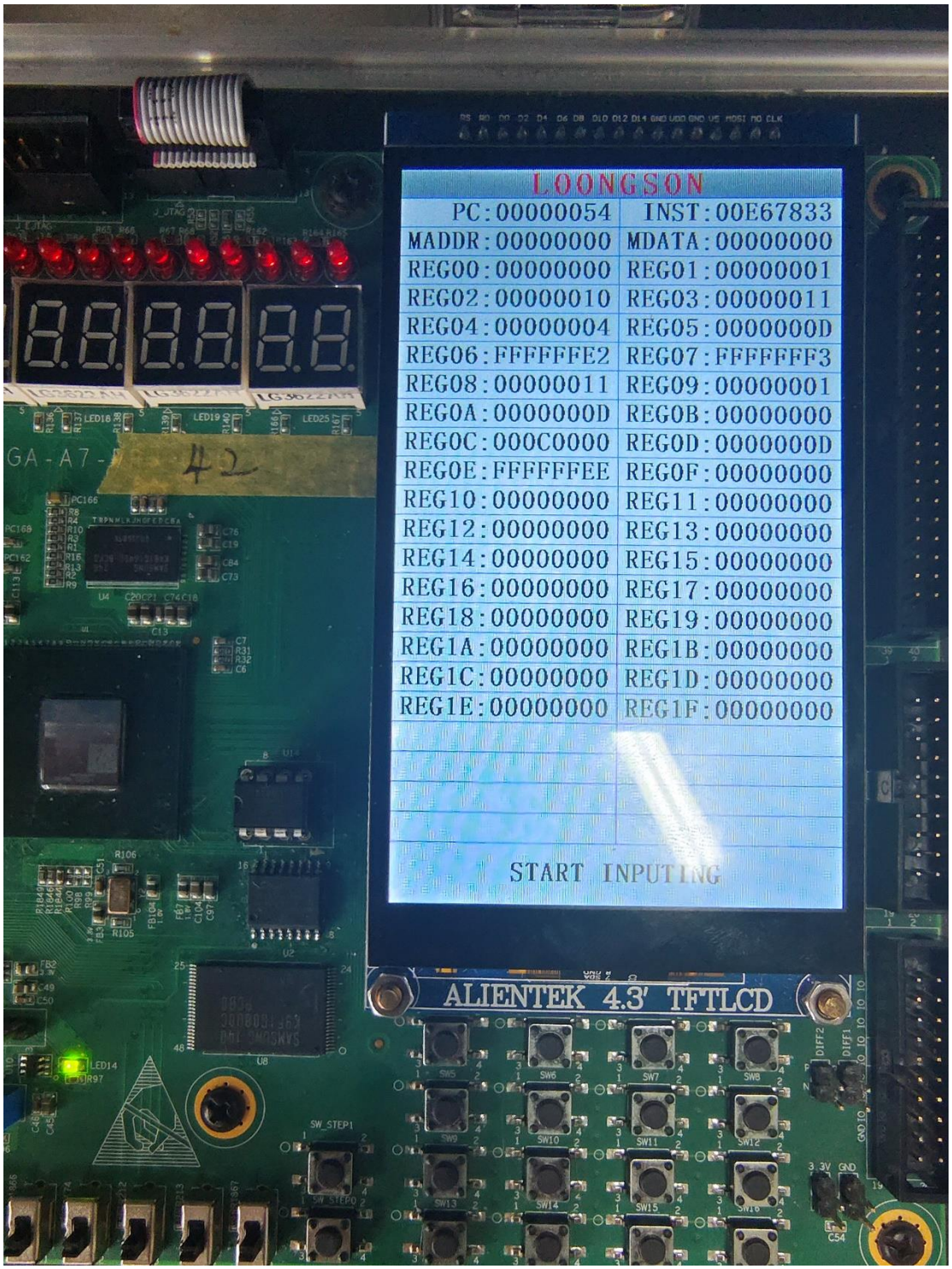


如图所示，结果正确

2）运行至 PC=54

```
assign inst_rom[20] = 32'h00E67031; // 50H: nxor  $14 ,$7,$6  | $14 = FFFF_FFEEH
```

按照设定，此时 REG0E=FFFF_FFEE

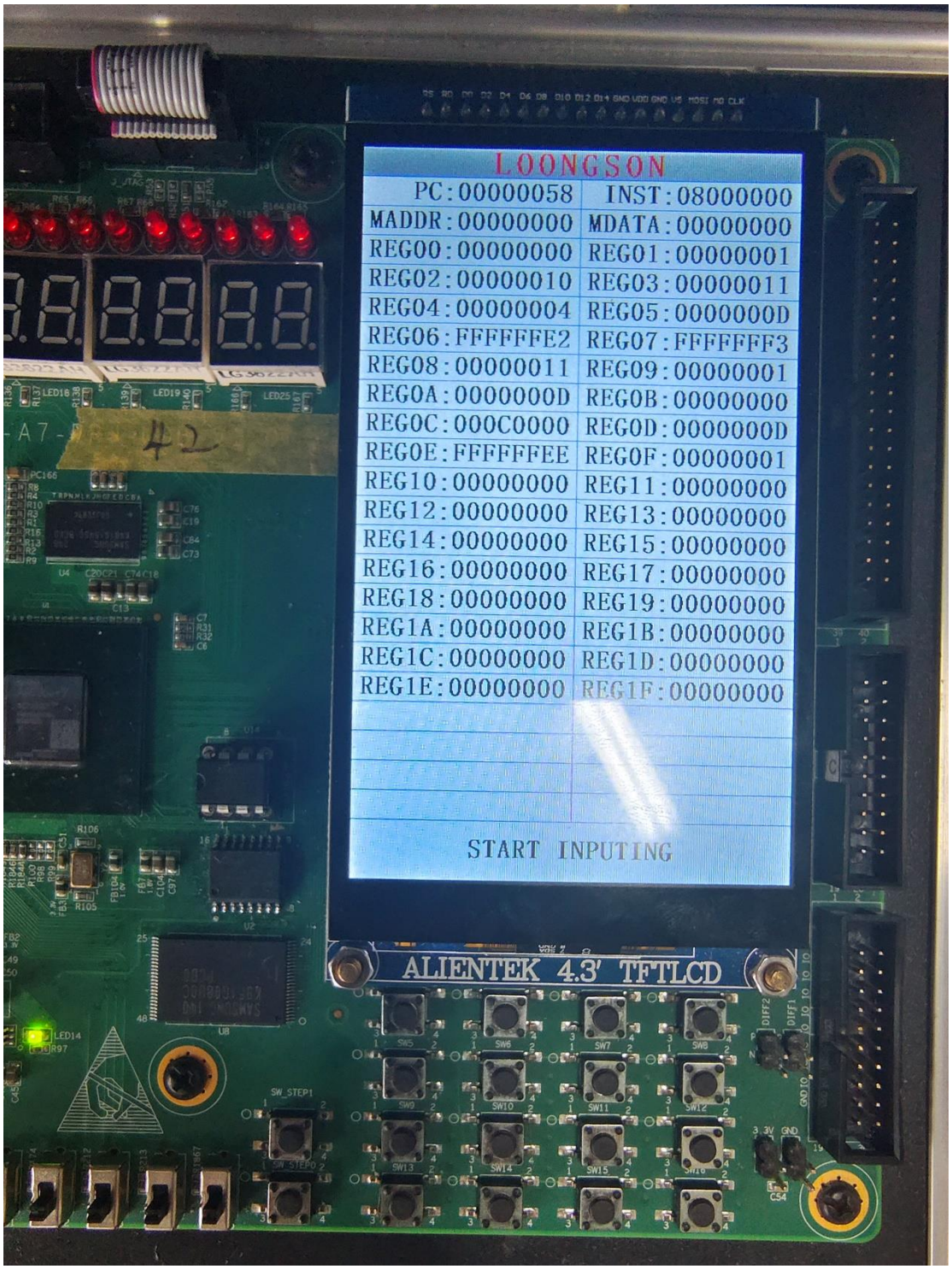


如图所示，结果正确

3）运行至 PC=58

```
assign inst_rom[21] = 32'h00E67833; // 54H: blt   $15 ,$7,$6   | $15 = 0000_0001H
```

按照设定，此时 REG0F=0000_0001

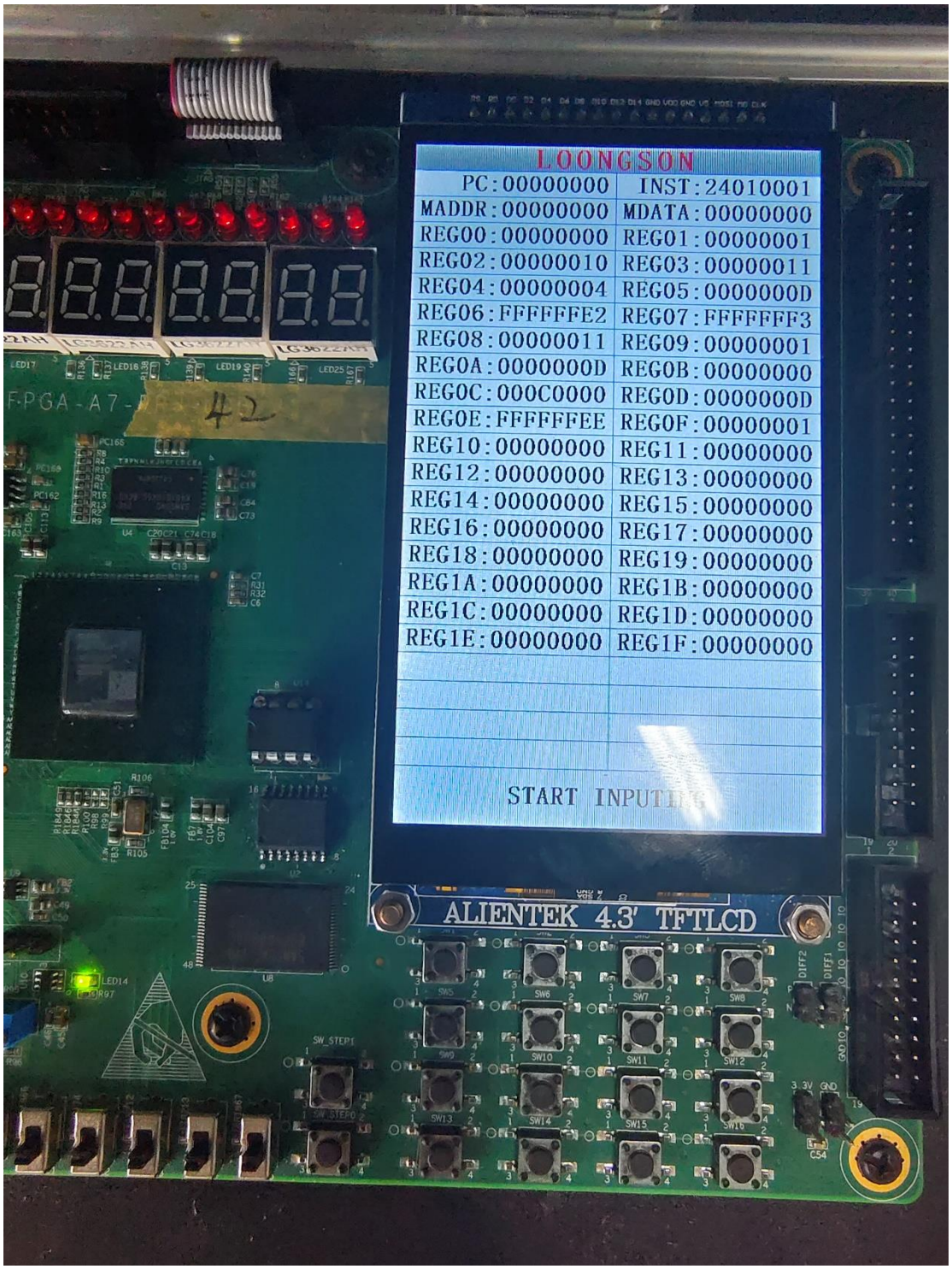


如图所示，结果正确

另外，添加指令后原来的跳转指令也能正常执行

```
assign inst_rom[22] = 32'h08000000; // 58H: j      00H          | 跳转指令00H
```

如下图所示，运行完跳转指令后，PC=0000_0000，表明单周期的指令运行完毕



综上所述，单周期 CPU 的实现和改进正常，实验成功！

5、 总结感想

在本次实验中，我面临了一系列挑战和困难，但通过持续的努力和耐心的调试，我最终成功地完成了单周期 CPU 的设计。

最初，我对于 MIPS 指令结构并不太熟悉，这导致我开始时不太清楚如何进行修改。在遇到一些细节问题时，我有时会忘记修改相关部分，比如 ALU 控制信号的位数调整。在

调试过程中，我不得不不断检查代码，寻找问题所在，并逐步进行修正，这消耗了我相当多的时间。

通过本次实验，我成功地将前期学习的知识和所设计的各个模块有效地整合和拼接，最终实现了单周期 CPU 的设计。在设计过程中，根据实验要求，我对 I 型指令和 R 型指令进行了设计，这使我更深入地理解了 MIPS 指令结构，以及 MIPS 指令集中常用指令的功能、编码和相关分类。

通过本次实验，我不仅加深了对单周期 CPU 原理和设计的理解，还对 MIPS 指令集和计算机组成原理的知识有了更深入的理解。通过动手实现 CPU，我更直观地感受到了 CPU 的执行过程和指令在各个阶段的处理流程，加深了对计算机体系结构的理解。

总的来说，虽然在实验过程中遇到了一些困难，但通过不断的学习和努力，我成功地完成了实验任务，并从中获得了很多收获和经验。这次实验让我更加深入地理解了计算机组成原理的知识，提高了对 Verilog 编程语言的熟练程度，也增强了我解决问题的能力。我期待在未来的学习和实践中，能够继续积累经验，不断提升自己的能力。