组成原理实验课程第_六_次实验报告

实验名称	单周期 CPU 实现			班级	李涛老师
学生姓名	王昱	学号	2212046	指导老师	董前琨
实验地点	A306		实验时间	2024.6.6	

一、实验目的

- 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。
- 了解熟悉 MIPS 体系的处理器结构，如延迟槽，哈佛结构的概念。
- 熟悉并掌握单周期 CPU 的原理和设计。
- 进一步加强运用 verilog 语言进行电路设计的能力。
- 为后续设计多周期 cpu 的实验打下基础。

二、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。

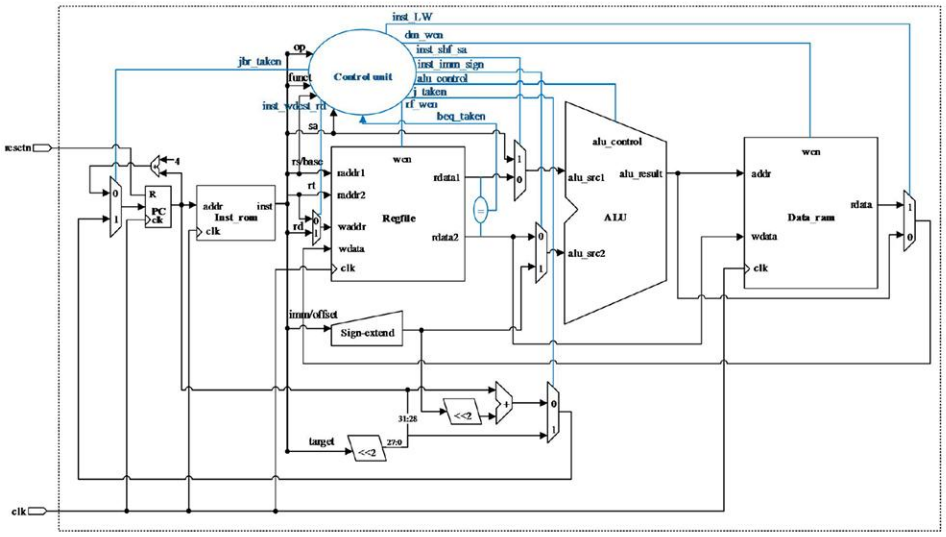
2、实验报告中原理图为指导手册中的 display 模块图，不用修改，报告中的内容和展示的结果应扩充指令的步骤和实验结果。

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。

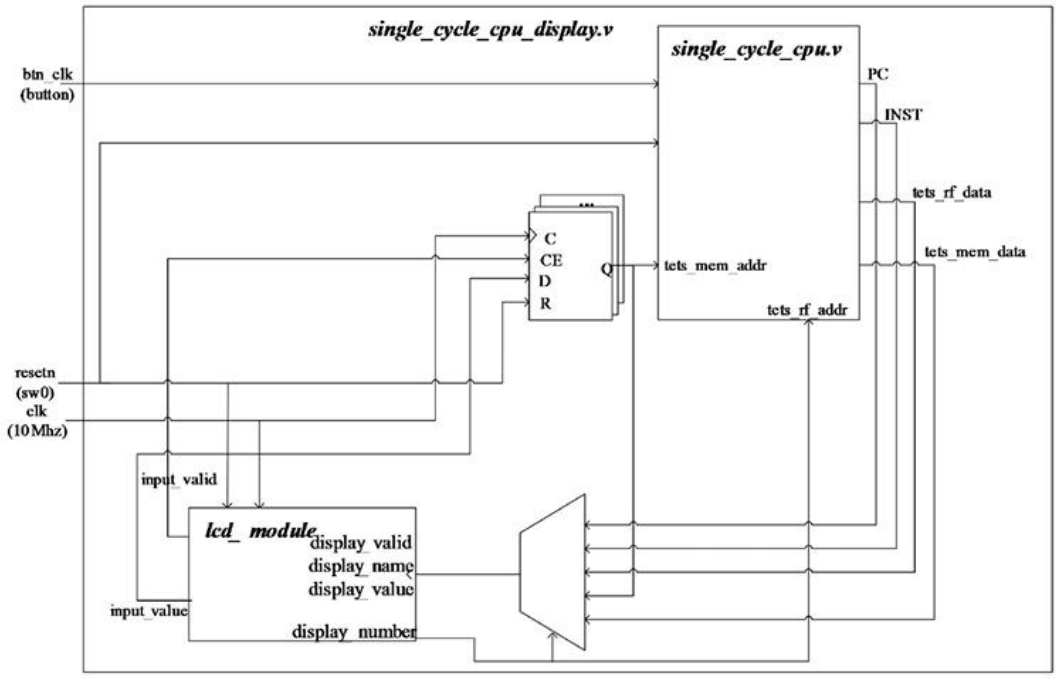
4、实验报告模板参考百度云盘文件，注意提交截至时间为 6 月 14 日下午 18：00。

三、实验原理图

单周期 CPU 的实现框图：



实验顶层模块实现图：



四、单周期 CPU 指令介绍及添加指令

单周期 CPU 能够执行 MIPS 指令集系统的一个子集，共 16 条指令，包括存储访问指令、运算指令、跳转指令。 根据拥有的字段类型不同，我们将指令分为 R 型指令 、I 型指令和 J 型指令。

- R 型指令

op	rs	rt	rd	shamd	funct

1.指令介绍

- op 段（6b）：恒为 0b000000；
- rs（5b）、 rt（5b） ：两个源操作数所在的寄存器号； rd（5b）：目的操作数所在的寄存器号；
- shamt（5 位）：位移量，移位指令的移位位数； func（6b）：决定 R 型指令的具体功能。

2.本次实验我添加的 R 型指令为 sgt,是依据小于则置位 slt 设计的大于则置位：

指令类型	汇编指令	指令码	操作数 1	操作数 2	操作数 3	目的寄存器	功能描述
R 型指令	sgt rd,rs,rt	000000|rs|rt|rd|00000|101110	[rd]	[rs]	[rt]	[rd]	大于则置位，如果操作数[rs]大于[rt]则把 rd 寄存器的值置位 1，与 slt 相反。

3.R 型指令执行过程

- 从指令存储器中取指令，更新 PC。
- ALU 根据 funct 字段确定 ALU 的功能。
- 从寄存器堆中读出寄存器 rs 和 rt。
- ALU 根据 2 中确定的功能，对从寄存器堆读出的数据进行操作。
- 将运算结果写入到 rd 字段对应的目标寄存器。

- I 型指令

op	rs	rt	constant or address

1.指令介绍

- op 段（6b）：决定 I 型指令类型；
- rs（5b）：是第一个源操作数所在的寄存器号；
- rt （5b）：是第二个源操作数所在的寄存器号 或 目的操作数所在的寄存器编号。
- constant or address（16b）：立即数或地址

2.本次实验我添加的 I 型指令

lli 指令依据 lui 指令，lui 加载到了高位，我设计了一个加载到低位的 lli 指令;

blt 指令依据 slt 指令，slt 是小于则跳转，我设计了大于则跳转，方便了一些需要大于条件判断的操作；

指令类型	汇编指令	指令码	操作数 1	操作数 2	目的寄存器	功能描述
I 型指令	lli rs,imm (立即数)	000111|00000|rt|imm	[rd]	imm	[rd]	对立即数进行扩展，将其加载到低位 [rd]={16'd0,imm}
I 型指令	blt rs,rt,offset	100100|rs|rt|offset	[rs]	[rt]		若 rs 寄存器的值大于 rt，就跳转到目标地址，执行如下指令 PC=PC+(offset)<<2

3.执行过程

存取指令：

- 从指令存储器中取指令，更新 PC 。
- ALU 根据 op 字段确定 ALU 的功能。
- 从寄存器堆中读出寄存器 rs 的值，并将其与符号扩展后的指令低 16 位立即数的值相加。
- 若为存储指令，则将 rt 寄存器中的值存到上步相加得到的存储器地址；

  若为取数指令，则将上步所得存储器地址里所存的数据放到 rt 目标寄存器中。

分支指令：

- 从指令存储器中取指令，更新 PC 。
- 从寄存器堆中读出寄存器 rs 和 rt 的值。
- 将所读寄存器的两值相减。
- 根据上步的结果是否为 0，将 PC+4 的值或 address 字段所对应地址存入 PC 中。

- J 型指令

op	address

1.指令介绍

- op 段（6b）：决定 J 型指令类型；
- op 段（6b）：决定 J 型指令类型；
- constant or address（26b）：转移地址

2.执行过程

- 从指令存储器中取指令，更新 PC 。
- 取出 address 字段，作为目标跳转地址。
- 将目标跳转地址存入PC 中。

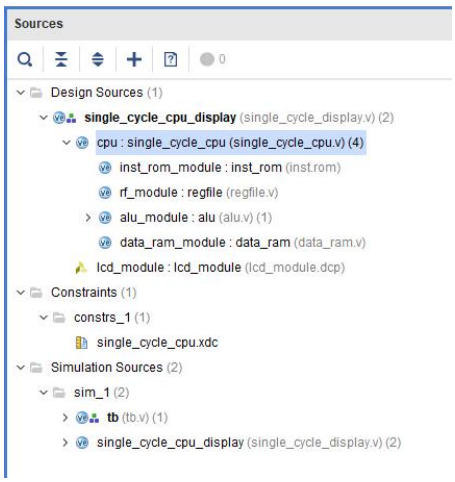
五、实验步骤

5.1 复现原实验

复现实验首先要把如下源文件中的代码复制到工程中：

名称	修改日期	类型	大小
adder.v	2016/4/29 12:58	V 文件	1 KB
alu.v	2016/4/30 11:03	V 文件	9 KB
data_ram.v	2016/4/30 11:03	V 文件	5 KB
inst_rom.v	2016/5/7 14:14	V 文件	4 KB
regfile.v	2016/4/30 11:02	V 文件	5 KB
single_cycle_cpu.v	2016/5/9 19:53	V 文件	10 KB
single_cycle_cpu.xdc	2017/2/9 11:43	XDC 文件	4 KB
single_cycle_cpu_display.v	2016/11/17 23:11	V 文件	6 KB
tb.v	2016/4/30 11:06	V 文件	2 KB

导入后所建文件如图所示：



随后上箱验证即可

5.2 改进实验，新增三条指令

选择 sgt 作为添加的 R 型指令，选择 lli 作为添加的 I 型指令,还添加了一条 blt 指令。下

面介绍具体修改之处：

1. 修改 single_cycle_cpu.v 文件

将添加的两个操作添加到指令列表中 ：

- wire inst_LLI;（立即数加载低位）
- wire inst_SGT;（大于则置位）
- wire inst_BLT;（大于则跳转）

```
 wire inst_LLI,wire inst_SGT,inst_BLT;
```

- 然后要将添加的这三个操作赋予操作数：

```
assign inst_LLI   = (op == 6'b000111);
assign inst_BLT   = (op == 6'b100100);
assign inst_SGT=op_zero & sa_zero & (funct == 6'b101110);
```

对于 I 型指令来说，只需要判断 op 字段即可对应到具体的操作，对于 R 型指令，需要判断它的 funct 字段，才可以知道具体要做哪种操作，通过这个部分就可以实现对于读到的指令进行译码，就可以知道进行具体的操作。

- 分支跳转指令 blt 实现：

```
  //分支跳转
 wire        beq_taken;
 wire        bne_taken;
 wire        blt_taken;
 wire [31:0] br_target;
 assign beq_taken = (rs_value == rt_value);        // BEQ 跳转条件：
GPR[rs]=GPR[rt]
 assign bne_taken = ~beq_taken;                    // BNE 跳转条件：
GPR[rs]≠GPR[rt]
 assign blt_taken = ~alu_result[31];
 assign br_target[31:2] = pc[31:2] + {{14{offset[15]}}, offset};
 assign br_target[1:0]  = pc[1:0];     // 分支跳转目标地址：
PC=PC+offset<<2

assign jbr_taken = j_taken               // 指令跳转：无条件跳转 或 满足分
支跳转条件
                  | (inst_BEQ & beq_taken)
                  | (inst_BNE & bne_taken)
                  | (inst_BLT & blt_taken);
 assign jbr_target = j_taken ? j_target : br_target;
```

这里是对 blt 跳转指令的译码，blt 指令需要对两个操作数进行减法操作，如果大于的话对应的最高位符号位就是 0，对其取反，就是确定要进行 blt 跳转的信号，否则就不需

要进行 blt 跳转，将进行 bit 跳转的信号参与到下面的跳转指令信号的构成中，进行或操作，这样就可以实现 blt 跳转的效果，译码完成后，下面可以进入 alu 模块进行计算 。

- 接下来需要定义并实现传递到执行模块的 ALU 源操作数和操作码：

```
// 传递到执行模块的 ALU 源操作数和操作码
 wire inst_lli,inst_sgt,inst_blt;
 assign inst_lli = inst_LLI;
 assign inst_sub = inst_SUBU  | inst_BLT;
 assign inst_sgt = inst_SGT;
```

- 然后修改立即数拓展指令：

```
assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LLI  | inst_LW |
inst_SW| inst_LLI ;
```

- 修改独热编码：

```
// ALU 操作码，独热编码
assign alu_control = {inst_sgt,
                      inst_lli,
                      inst_add,
                      inst_sub,
                      inst_slt,
                      inst_sltu,
                      inst_and,
                      inst_nor,
                      inst_or,
                      inst_xor,
                      inst_sll,
                      inst_srl,
                      inst_sra,
                      inst_lui};
```

- 修改寄存器的写回值

```
assign inst_wdest_rt = inst_ADDIU | inst_LW | inst_LUI| inst_LLI ;
assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT | inst_AND |
inst_NOR| inst_OR   | inst_XOR | inst_SLL | inst_SRL | inst_SGT
```

2.接下来我们修改 alu.v 文件

需要在 alu 中添加的操作为 lli、sgt，blt 只需要 alu 来进行减法

- 拓展 alu 位宽：

```
input  [13:0] alu_control,  // ALU 控制信号
```

- 然后添加独热编码：

```
// ALU 控制信号，独热码
wire alu_add;    //加法操作
```

```
wire alu_sub;    //减法操作
wire alu_slt;    //有符号比较，小于置位，复用加法器做减法
wire alu_sgt;    //有符号比较，大于置位，复用加法器做减法
wire alu_sltu;   //无符号比较，小于置位，复用加法器做减法
wire alu_and;    //按位与
wire alu_nor;    //按位或非
wire alu_or;     //按位或
wire alu_xor;    //按位异或
wire alu_sll;    //逻辑左移
wire alu_srl;    //逻辑右移
wire alu_sra;    //算术右移
wire alu_lui;    //高位加载
wire alu_lli;    //低位加载
```

- 再添加控制信号：

```
assign alu_sgt  = alu_control[13];
assign alu_lli  = alu_control[12];
```

- 添加结果信号以及运算逻辑：

```
wire [31:0] lli_result;
wire [31:0] sgt_result;
assign lli_result = {16'd0, alu_src2[31:16]};
assign sgt_result[31:1] = 31'd0;
assign sgt_result[0]=(~slt_result[0])&(adder_result!=32'b0);
```

- 添加选择结果输出：

```
assign alu_result = (alu_add|alu_sub) ? add_sub_result[31:0] :
                    alu_slt           ? slt_result :
                    alu_sgt           ? sgt_result :
                    alu_sltu          ? sltu_result :
                    alu_and           ? and_result :
                    alu_nor           ? nor_result :
                    alu_or            ? or_result  :
                    alu_xor           ? xor_result :
                    alu_sll           ? sll_result :
                    alu_srl           ? srl_result :
                    alu_sra           ? sra_result :
                    alu_lui           ? lui_result :
                    alu_lli           ? lli_result :
                    32'd0;
endmodule
```

3.接下来我们修改 inst_rom.v 文件：

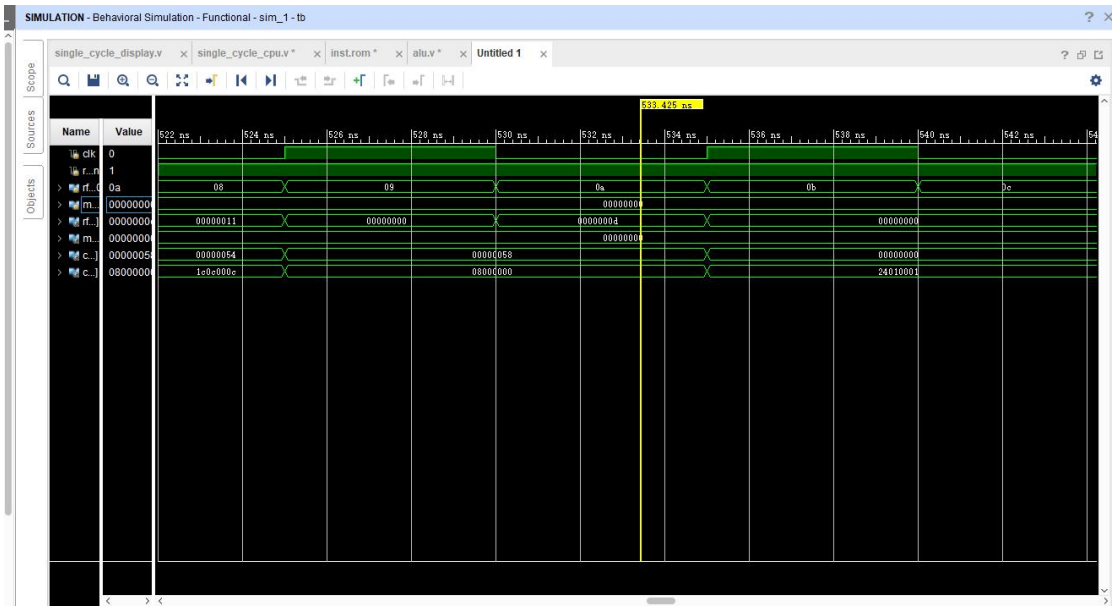
添加三条指令对我们新添加的操作结果进行验证：

```
//-------------- 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'h00C7482E; // 4CH:sgt  $9, $6, $7
 assign inst_rom[20] = 32'h91210002; // 50H:blt $9,$1,#2
 assign inst_rom[21] = 32'h1C0C000C; // 54H:lli $12,#12
 assign inst_rom[22] = 32'h08000000; // 58H: j     00H          | 跳转指令 00H
```

六、实验验证：

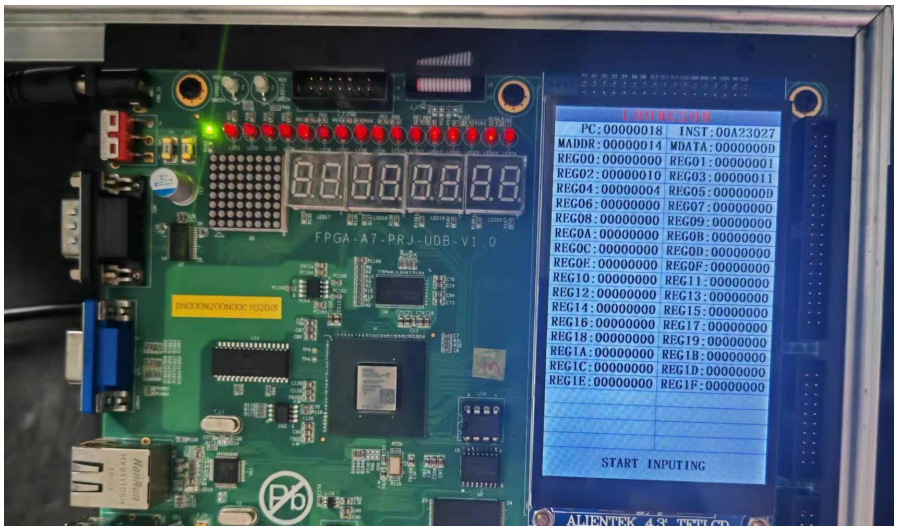
首先在 VIVADO 上进行仿真验证，发现所有指令都可以顺利执行，包括跳转指令：



（一）实验复现

接下来是对一些操作实验指令的复现：

- PC=0x18：

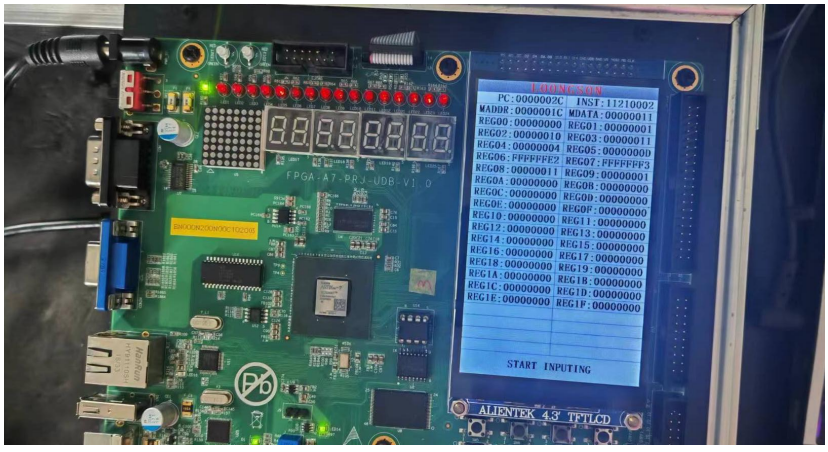


此时，已经运行了以下指令：

```
 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
```

对以上指令执行后，所得到的值与图中相同，可知结果正确。

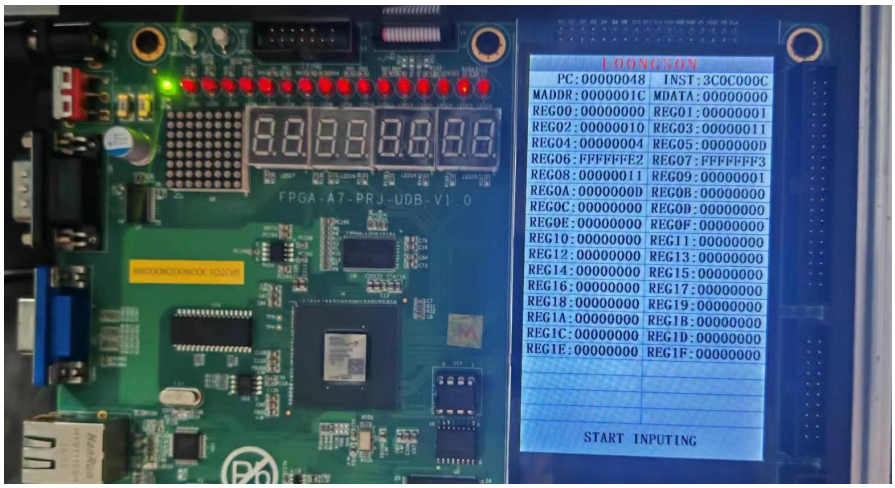
- PC = 0x2C



```
 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
```

执行以上指令后，进行验算，发现结果相同，说明正确。

- PC=0x34

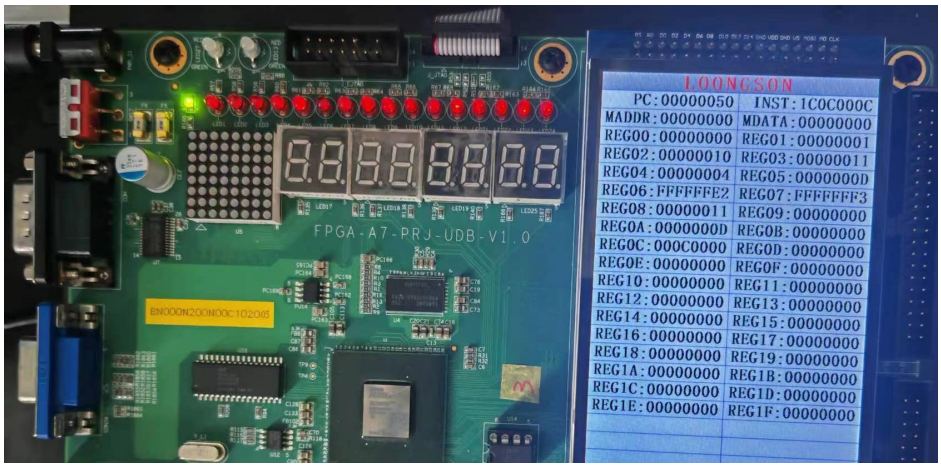



```
 assign inst_rom[11] = 32'h11210002; // 2CH: beq   $9 ,$1,#2    | 跳转
到指令 34H
 assign inst_rom[12] = 32'h24010004; // 30H: addiu $1 ,$0,#4    | 不执
行
```

其中 beq 指令得到执行，跳转到 34H，其后面的 addiu 指令不执行，因此当前 PC 为 0x34。

执行以上指令后，进行验算，发现结果相同，说明正确。

- PC = 0x48

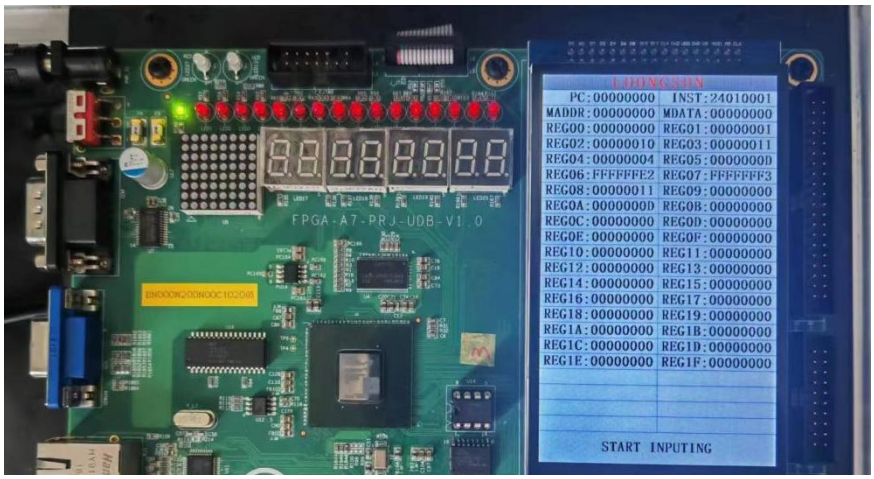


```
    assign inst_rom[13] = 32'h8C2A0013; // 34H: lw    $10,#19($1) | $10
= 0000_000DH
  assign inst_rom[14] = 32'h15450003; // 38H: bne   $10,$5,#3   | 不跳
转
    assign inst_rom[15] = 32'h00415824; // 3CH: and   $11,$2,$1   | $11
= 0000_0000H
    assign inst_rom[16] = 32'hAC0B001C; // 40H: sw    $11,#28($0) |
Men[0000_001CH] = 0000_0000H
    assign inst_rom[17] = 32'hAC040010; // 44H: sw    $4 ,#16($0) |
Mem[0000_0010H] = 0000_0004H
```

进行验算后，同样可见各个寄存器中的值均为正确结果。

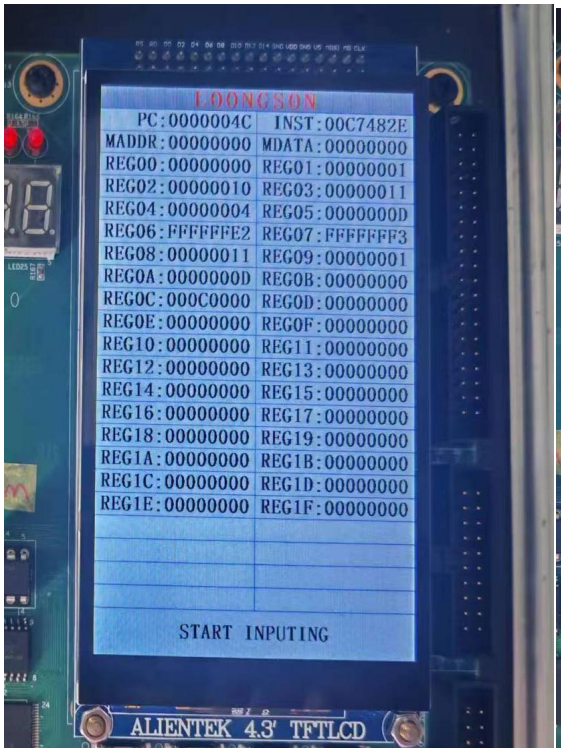
- PC=0

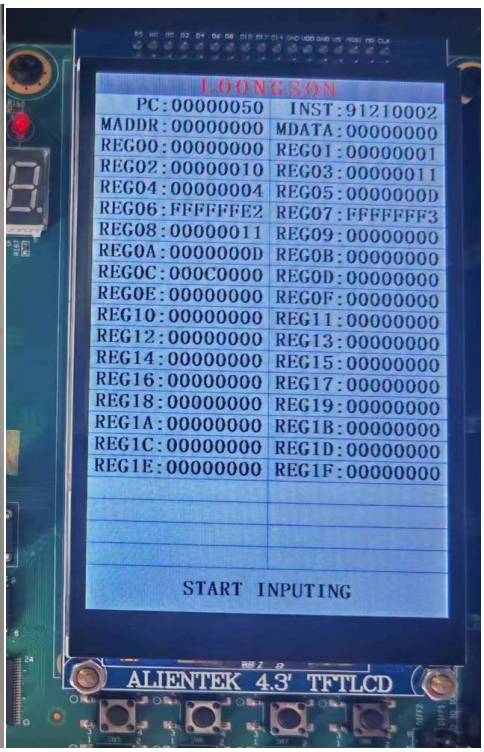
执行最后一条指令后跳转到地址为 0 的地方，如下图所示：



（二）改进后的实验验证

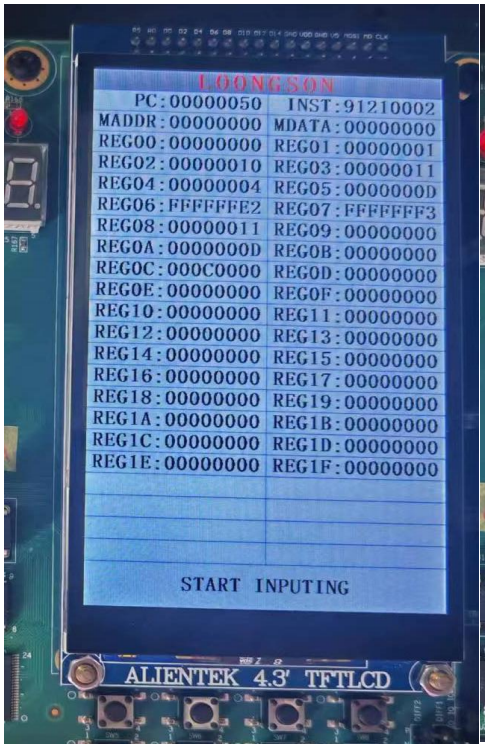
- 0x4c:

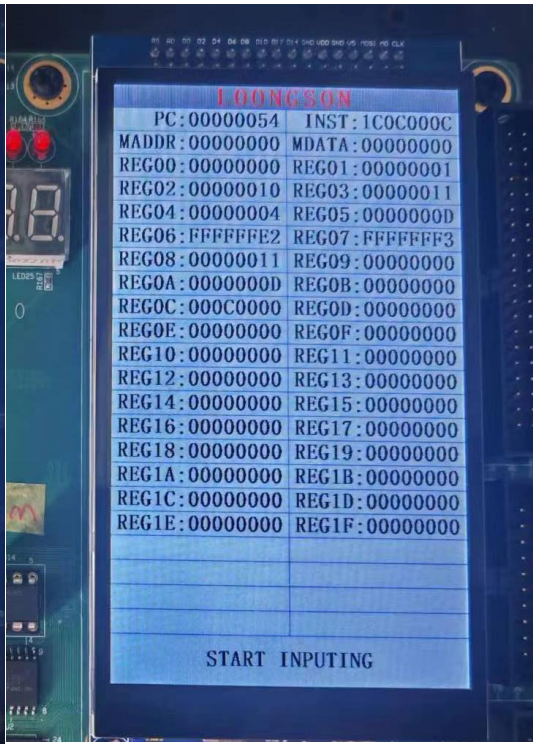




如图，执行了 sgt $9,$6,$7 大于则置位指令后，由于六号寄存器的值小于七号寄存器的值，所以置九号寄存器的值为 0。

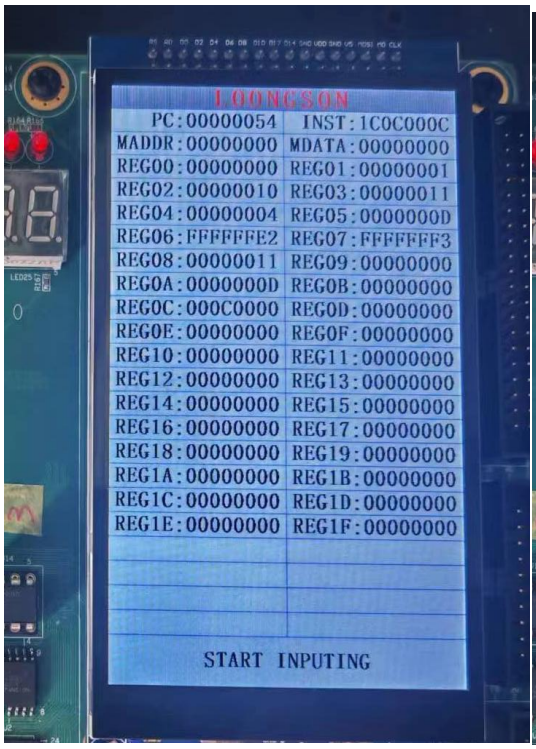
- 0x50：

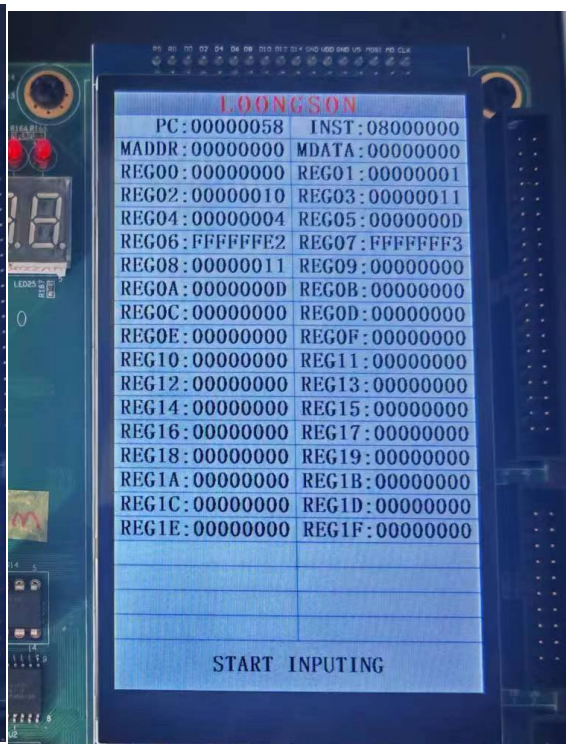




执行 blt $9,$1,#2，由于此时九号寄存器与一号寄存器的值相等（都为 0）,所以不跳转。

- 0x54:





如图，执行了 54H:lli $12,#12 后，立即数 0X000C 扩展后高位为 0，加载到 12 号寄存器的低半字上，用 0 填充高位，所以十二号 REG0C 寄存器的值变为 0，结果正确。

以上就完成了本学期计组实验的全部内容！

七.总结感想

1.本次实验可以看作是前几次实验的一个融合，具有一定的难度，通过本次实验我深刻理解了单周期 CPU 的原理和实现，MIPS 指令集和单周期 CPU 的原理和设计，提高了对 MIPS 体系处理器结构的理解，通过代码改进，我对指令的理解有所提高。

2.本学期计组实验通过老师的悉心指导以及自己课后的学习，我对 verilog 语言的理解与掌握更进一步，同时对 vivado 软件和实验箱的使用更加得心应手，对计组理论知识的应用理解也有所加深，收获很大！