

嵌入式计算机系统与实验 | 作业 3

Shuiyuan@Noroshi

1 ADC0809 与 MCS51 的对接

1.1 ADC0809 概述

ADC0809 是一种久经考验的, 简单耐用的模拟信号源与数字信号的转换元件. 其转换时间相对较低, 供电要求简单且不要求校准, 功耗低, 分辨率为 8 位, 在严寒与高温环境下都能正常工作

1.1.1 ADC0809 的接口

其关键的部分接口如下

- IN0~IN7: 8 路模拟信号输入端
- D0~D7: 8 路数字信号输出端
- ADDA, ADDB, ADDC: 模拟通道选择器
- ALE: 地址锁存信号
- START: 启动信号, 开始接收 CPU 的信号输入
- EOC: 结束信号, 高电平表明一次转换完成
- OE: 数据输出允许信号, 位于高电平时将数据从 D 口输出

ADC 接口的简单工作流程:

1. 处理器先向 ADC 送出所需要的模拟信号的地址(ADDA, ADDB, ADDC), 并锁存(ALE=1)
2. 启动 ADC(START=1)
3. ADC 开始工作, 此时 EOC 进入低电平
4. EOC 拉高, 送出 OE 信号
5. 处理器读走数据

值得一提的是, ADC 并没有内置时钟电路, 需要外部提供时钟信号

1.1.2

一种方法是把 ADC0809 视作 51 单片机的外部 RAM, 简单来说就是 P0 和 P2 联合向 ADC 提供例如地址, 通道以及其他控制信号, 然后 P0 同时复用为数据总线来接受其最终得到的八位数字信号, 实际上, 我们采用的指令也是与外部读写 RAM 同样的 [MOVX](#) 指令 不过需要注意的是, ADC 的转换时间大约要 100ms, 也就是 100 个机器周期了, 需要在程序里特别注意, 不要还没转换好就忙着读取输出了

1.1.3

另一种方法是把 ADC0809 视作一种中断, 即把完成信号输出端口接在中断端口上, 这样对主机的影响较小, 主机在 ADC 转换期间也可以完成其他的一些程序. 相对而言会更加高效一点

1.1.4 51 单片机的中断机制

虽然还没有学, 但这里有必要对 51 单片机的机制有一点大概的了解. 51 单片机的中断机制多样, 包括但不限于外部手动发送中断, 计时器溢出中断, 以及串口中断等. 当然我们这里还是先着眼于最简单的外部发送中断. 有两个接口都可以接受外部中断的信号, 即 P3.2 和 P3.3, 当其满足中断条件时(可能是高电位或者其他情况), 就会跳转到特定的地址(P3.2 触发的外部中断 0 是跳转至 00003H, P3.3 触发的外部中断 1 是跳转至 0013H), 然后系统会执行这部分的代码, 随后结束中断, 返回主程序

1.2 ADC 与单片机连接的实际尝试

1.2.1 中断连接-LED 灯泡输出

参考 PPT, 这部分的代码如下

```

ORG 0000H
    LJMP START
ORG 0003H
    LJMP EINTO

ORG 0100H
START:
    MOV R0, #30H
    MOV R2, #08H
    SETB EA
    SETB IT0
    SETB EX0

LOOP:
    MOV DPTR, #000H
    MOVX @DPTR, A
    LCALL DELAY
    SJMP START

DELAY:
    MOV R5, #050H
DELAY_2:
    DJNZ R5, DELAY_2
    RET

ORG 0200H
EINTO:
    MOVX A, @DPTR
    MOV @R0, A
    MOV P1, A
    INC DPTR
    INC R0
    DJNZ R2, NEXT
    CLR EA
    CLR EX0
    RETI

NEXT:
    MOVX @DPTR, A
    RETI

END

```

那我们还是一部分一部分地看这段代码的内容: 程序开始运行, 跳转到 0100H 地址, 其先将 R0, R2 分别赋值 30H, 8H, 这两个在之后还有用; 而随后将 EA, IT0, EX0 设定为高位, 这部分主要是设定了中断的设置: EA 拉高使单片机可以接受中断信号, EX0 拉高使单片机可以更具体地可以接受来自外部中断 0 的中断信号, 而 IT0 拉高的作用使设置中断信号触发为脉冲模式, 即在发生改变的时候中断, 而非位于高/低电位时中断。接下来进入 LOOP 段, 首先我们给数据指针所指向的位置设置为 000H(初始化), 随后我们把 A 的值赋值给数据指针所指向的位置(即 000H), 之后主程序部分就进入了等待和死循环的环节, 纳闷我们再看看中断处理程序:

当触发外部中断 0 时, 按照单片机的运行机制, 其会立刻停止自己的动作, 然后跳转至 0003H 处, 那么在这段程序里, 也就是先跳到 0003H 之后会自动跳转到(位于 0200H)处的 EINTO 程序段, 在这段里, 我们先将

DPTR 指针所指向的指针(即模拟信号的转换后输出处), 然后我们将 R0 中存储的地址(其实也就是把数值存到 30H 地址), 然后我们把这个信号传递给 P1 即可以完成输出工作. 之后我们将数据指针后移, 避免例如数据被覆盖等问题.

接下来是电路部分:

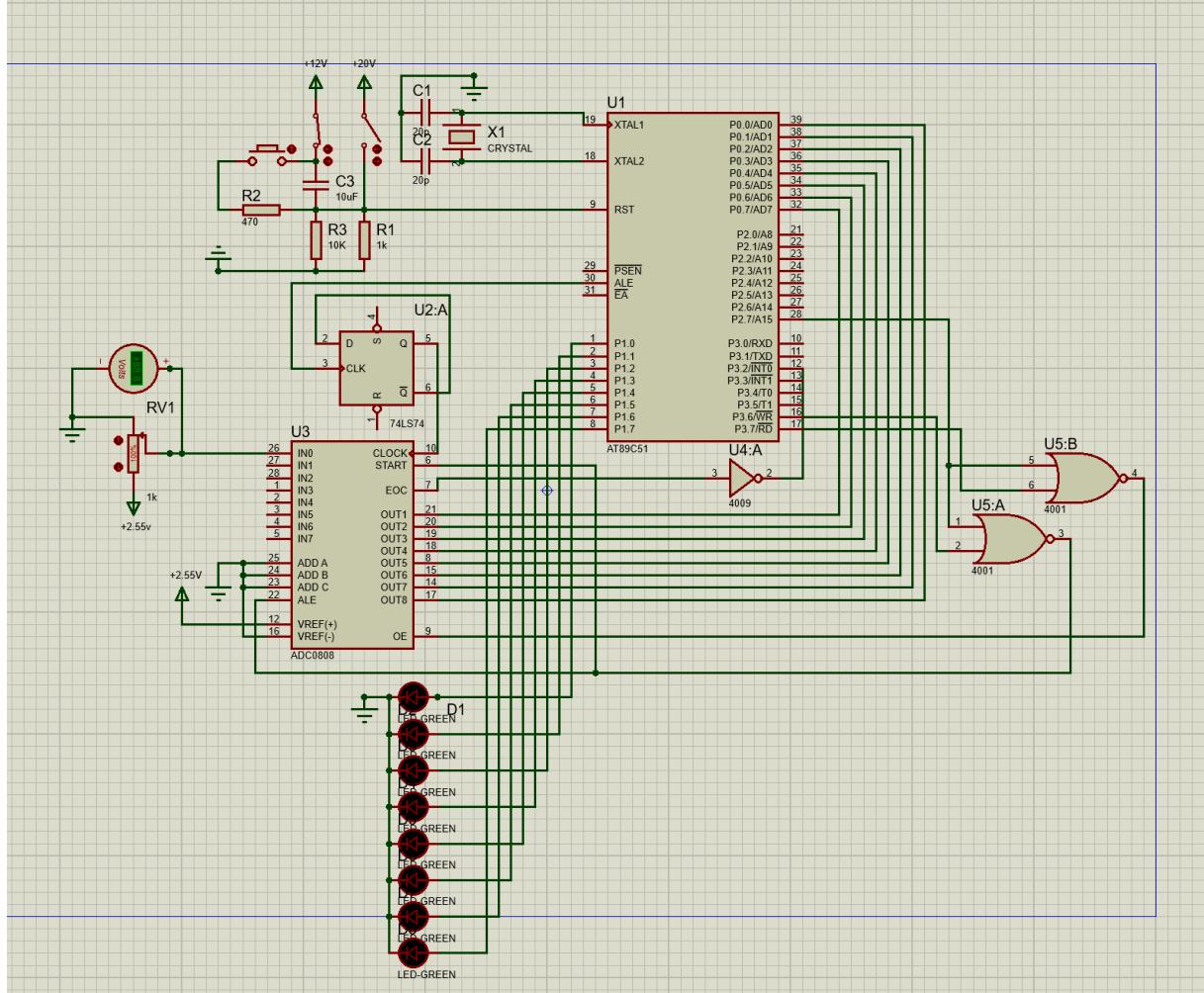


Figure 1: ADC 读取滑动变阻器数值并转换为 LED 亮灯状态的电路示意

左上部分的应用于单片机主机的复位和时钟电路我们就略过了. 中间的 74LS74 元件在这里就是一种时钟电路, 其接收 ALE 的周期性脉冲信号, 并作为时钟信号传给 ADC; 再看 ADC 部分, 左侧的滑动变阻器的电压就是我们需要测的模拟信号, ADC 左下角为电路部分和指明模拟器接收位. 那么 OUT 的八位线与主机 P0 连接, 设计类似于数据总线, 而 P1 端口即为输出信号.

接下来我们来看看右下角这里这个 4009 的非门. 我们知道 EOC 的逻辑是当其 ADC 进入转换的时候会置零, 而在完成转换之后会拉高; 在非门之后的对接 INT0 接口的逻辑就是, 在转换时处在高电位, 而在转换完成后会变为低电位, 同时又由于我们设置的是边沿触发, 也就是不是由电位本身触发而是由电位的变化触发的. 所以也就是这里的目的就是在转换完成后触发中断信号.

那么我们来试着运行一下

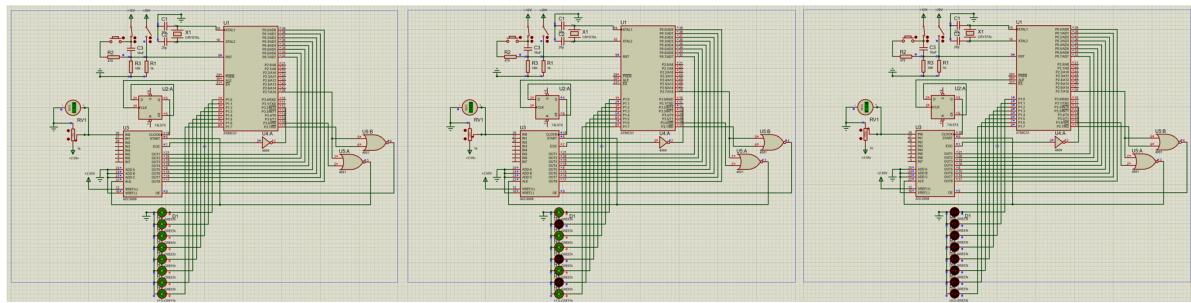


Figure 2: 可以观察其能在不同电压下 LED 显示排列不同

1.2.2 查询连接-四位 LED 数码管输出

首先我们先介绍一下这个实验中新追加的元件,也就是 7SEG-MPX4-CA, 7 段数码管,采用了共阳极的驱动方式,其原理其实和单个 7 段差不多,是轮流向 4 个 7 段管发送其所需要显示的信号,利用类似视觉暂留的形式让人观感上是 4 位数字同时在显示

接下来让我们看看这部分的代码,不过由于代码太长了让我们还是一部分一部分地看吧

```

ORG 0000H
LJMP START

ORG 0040H
START:
MOV DPTR, #7FFBH
MOVX @DPTR, A
LCALL DELAY
MOVX A, @DPTR
MOV 30H, A
JMP NEXT

DELAY:
MOV R5, #050
DELAYY:
DJNZ R5, DELAYY
RET

```

这部分包含两个程序片段,也就是 START 和 DELAY,后者显然就是一个等待的操作,我们就不多解释了.前者则包含了我们初始化和读取数字信号的部分,其实就是整片程序的核心部分了,后面的一长串的主要目的只是把我们读到的数据打到数码管上.这部分 START 程序先给 DPTR 赋值,此处我们可以看到这里 DPTR 的最低三位为 000,这也就告诉 ADC 我们读取的就是 IN0 的信号,然后我们把 A(这里应该就是默认 0)赋给 ADC 中存储数字信号的地址,那么实际上没什么用.然后我们就等,等到数据转换完成之后再把他读回 A 里,然后存放在数据存储器的某个一般地址上

```

NEXT:
MOV A, 30H
MOV B, #10
DIV AB
PUSH B
MOV B, #10
DIV AB
PUSH B
MOV B, #10
DIV AB
MOV A, B

```

这部分开始我们先将存在一般地址里的数据再放回寄存器内,因为我们要开始把我们测到的数值的每一位拆出来,拆的方法就是连续除以 10,每次的余数分别代表个十百位.比如假设 A 的数值是 255,那么这里

我们第一次除以 10, 根据 DIV 的逻辑那么商存入 A, 余数存入 B, 这里的 B 就是个位数值了, 将其压入栈中, B 重新设为 10, 重复一次 B 又得到了十位数值, 再压入栈中, 再次重复就能得到百位数值了.

N30:

```
CJNE A, #0, N31
MOV R2, #01000000B
JMP N20
```

...

这部分其实就是一个匹配数字的功能, 将数字转换为 7 段数码管显示对应数字时的信号(个, 十, 百分别存到 R4, R3, R2), 不过注意到这里我们有 8 位, 但是数码管只有 7 位, 这是因为最高位是小数点位, 若为 0 则点亮自身后面的小数点, 原例程这里点亮的是第二位数后面的小数点, 但似乎还是最高位后的小数点亮更符合实际情况, 这里改一下就行

```
L000P:
MOV P2, #00000010B
MOV P1, R2
MOV P1, #11111111B
MOV P2, #00000100B
MOV P1, R3
MOV P1, #11111111B
MOV P2, #00001000B
MOV P1, R4
MOV P1, #11111111B
LJMP START
```

最后这部分就是输出电路了, 按照电路设计, P2 连接的是指示当前数码管是哪位亮的接口, 而 P1 口连接的是指示对应位数码管亮的数字的接口, 这里就是先通过 P2 接口告诉数码管我们接下来要显示第几位, 然后 P1 口告诉数码管这位的数值是几.

那我们看看实际电路:

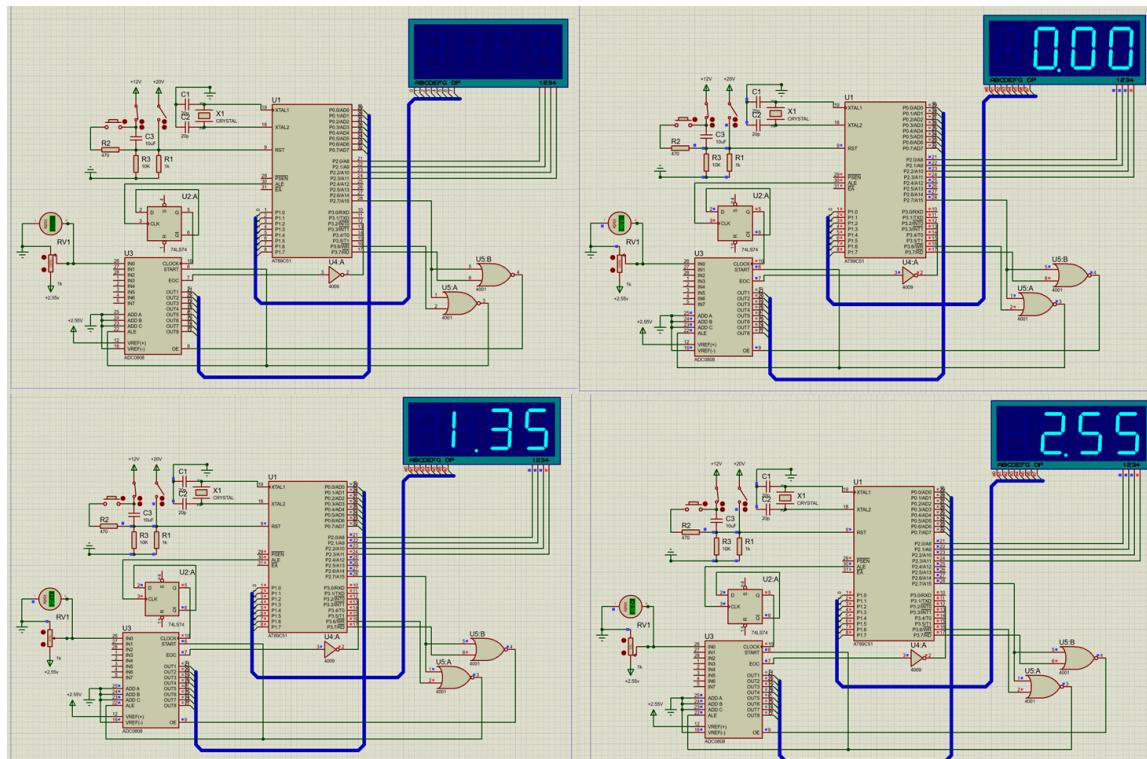


Figure 3: 电路图与运行情况, 可以发现其数字 LCD 显示与左下角的实际电压值是相同的.

不过这部分电路感觉和之前的连线方式的整体差别不大,正如上面我们所提到的,4位7段数码管总共有2组输入口,一组用来说明正在输出哪一位,另一组说明输出的那个数.除此之外没有本质的区别.但这里我采用了总线的方式连接几组端口(蓝色线),蓝色总线就类似于把几根绿色线绑在一起,在总线的输出端,我们可以用连线标号来表明总线里的分线,例如这里 OUT1 和 P0.7 都有 OUT1 标签,也就是这两个实际上是连在一起的.

2 外扩存储时总线上的读写时序

TODO

3 数据输入输出探究

研究下面这段代码:

```

MOV 23H, #30H
MOV 12H, #34H
MOV R0, #23H
MOV R7, 12H
MOV R1, #12H
MOV A, @R0
MOV 34H, @R1
MOV 45H, 34H
MOV DPTR, #6712H
MOV 12H, DPH
MOV R0, DPL
MOV A, @R0

```

我们可以预测运行结果

| 指令 | DPH | DPL | 45H | 34H | 23H | 12H | R7 | R1 | R0 | A |
|---------------|------|-----|------|------|------|------|------|------|------|------|
| MOV 00H | 00H | 00H | 00H | 00H | *30H | 00H | 00H | 00H | 00H | 00H |
| 23H, #030H | | | | | | | | | | |
| MOV 00H | 00H | 00H | 00H | 00H | 30H | *34H | 00H | 00H | 00H | 00H |
| 12H, #034H | | | | | | | | | | |
| MOV 00H | 00H | 00H | 00H | 00H | 30H | 34H | 00H | 00H | *23H | 00H |
| R0, #023H | | | | | | | | | | |
| MOV 00H | 00H | 00H | 00H | 00H | 30H | 34H | *34H | 00H | 23H | 00H |
| R7, | | | | | | | | | | |
| 012H | | | | | | | | | | |
| MOV 00H | 00H | 00H | 00H | 00H | 30H | 34H | 34H | *12H | 23H | 00H |
| R1, #012H | | | | | | | | | | |
| MOV 00H | 00H | 00H | 00H | 00H | 30H | 34H | 34H | 12H | 23H | *30H |
| A, | | | | | | | | | | |
| @R0 | | | | | | | | | | |
| MOV 00H | 00H | 00H | 00H | *34H | 30H | 34H | 34H | 12H | 23H | 30H |
| 34H, | | | | | | | | | | |
| @R1 | | | | | | | | | | |
| MOV 00H | 00H | 00H | *34H | 34H | 30H | 34H | 34H | 12H | 23H | 30H |
| 45H, | | | | | | | | | | |
| 34H | | | | | | | | | | |
| MOV DPTR,*67H | *12H | 34H | 34H | 30H | 34H | 34H | 12H | 23H | 30H | |
| #6712H | | | | | | | | | | |
| MOV 67H | 12H | 34H | 34H | 30H | *67H | 34H | 12H | 23H | 30H | |
| 12H, | | | | | | | | | | |
| DPH | | | | | | | | | | |

```

MOV      67H    12H    34H    34H    30H    67H    34H    12H    *12H   30H
R0,          DPL
MOV      67H    12H    34H    34H    30H    67H    34H    12H    12H    *67H
A,          @R0
  
```



Figure 4: 运行分步结果, 似乎是没有问题的

4 例程探究

4.1 十六位加减法

下面这段代码的目的是完成 $(1234)_{16}$ 与 $(0FA3)_{16}$ 的减法运算, 由于单片机中一个地址只能存一个字节共8位数据, 所以这里我们需要使用两个地址分别存高八位和低八位, 类似于DPTR/P0, P2口做总线时的想法。首先我们手动计算一下, $(1234)_{16} - (0FA3)_{16} = (4660)_{10} - (4003)_{10} = (657)_{10} = (291)_{16}$

```

ORG 0000H
LJMP START
; 1234H - 0FA3H
ORG 0040H
START:
  
```

```

CLR C
MOV A, #34H
SUBB A, #0A3H
MOV 50H, A
MOV A, #12H
SUBB A, #0FH
MOV 51H, A
END

```

我们运行一下可以发现这里低八位发生了进位，也就是 C 被设为了 1，然后带着这个 1 进行高八位的运算，最后的结果也是符合正确答案的。

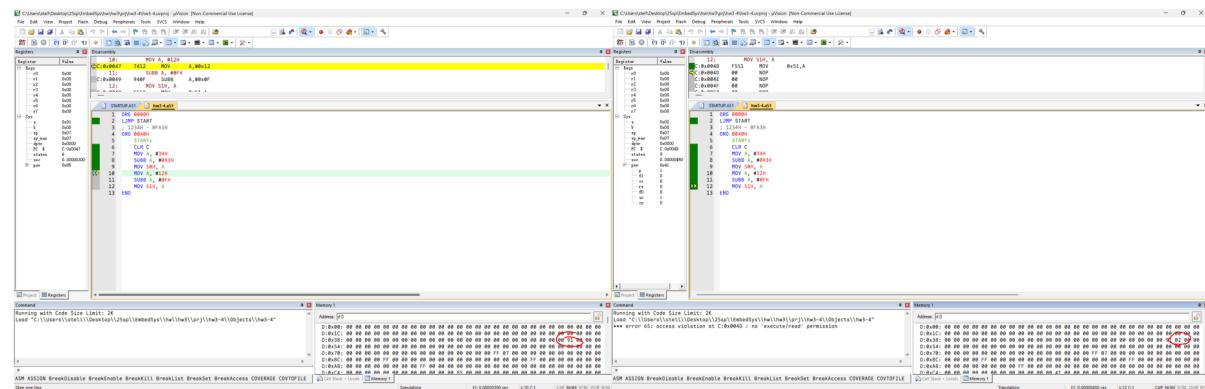


Figure 5: 答案应该是正确的

4.2 按键状态的读取

利用下片代码，我们可以通过按钮控制 P3.2, P3.3 的电位，继而控制 P0 的电位(控制的 P0 是因为我在把代码从作业文件抄到 keil 里的时候打错了)

```

ORG 0000H
LJMP START

ORG 0100H
START:
    MOV P0, #0FFH
    MOV P3, #000H
    LJMP PART1
PART1:
    JNB P3.2, PART2
    JNB P3.3, PART3
    LJMP PART1
PART2:
    MOV P0, #000H
    LJMP PART1
PART3:
    MOV P0, #0FFH
    LJMP PART1
END

```

电路如图，注意 P0 口输出高电位的时候需要上拉电阻

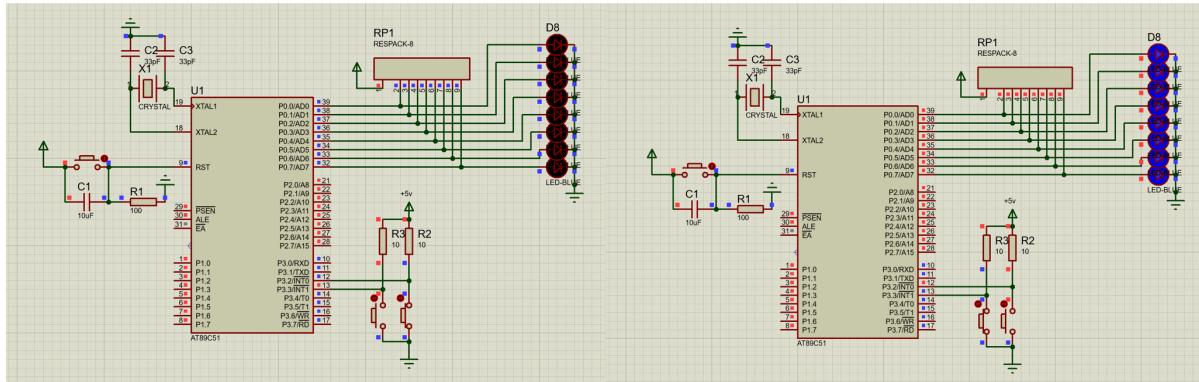


Figure 6: 按下右侧按钮后 P3.2 为低电位, 使 P0 低电位, LED 灯灭, 按下左侧按钮后 P3.3 为低电位, 使 P0 高电位, LED 灯亮