嵌入式计算机系统与实验 | 作业 4

Shuiyuan@Noroshi

1 程序设计基础例程

1.1 例一：顺序结构程序

变量存在内部 RAM 的 20H 单元中，其取值范围：0 5，编程，查表法求其平方值，结果放在地址为 21H 的 RAM 中。 写出代码：

```
ORG 0000H
LJMP START

ORG 0050H
  START:
  MOV 20H, #03
  MOV DPTR, #TABLE
  MOV A, 20H
  MOVC A, @A+DPTR
  MOV 21H, A
  SJMP $
  TABLE:
  DB 0, 1, 4, 9, 16, 25
END
```

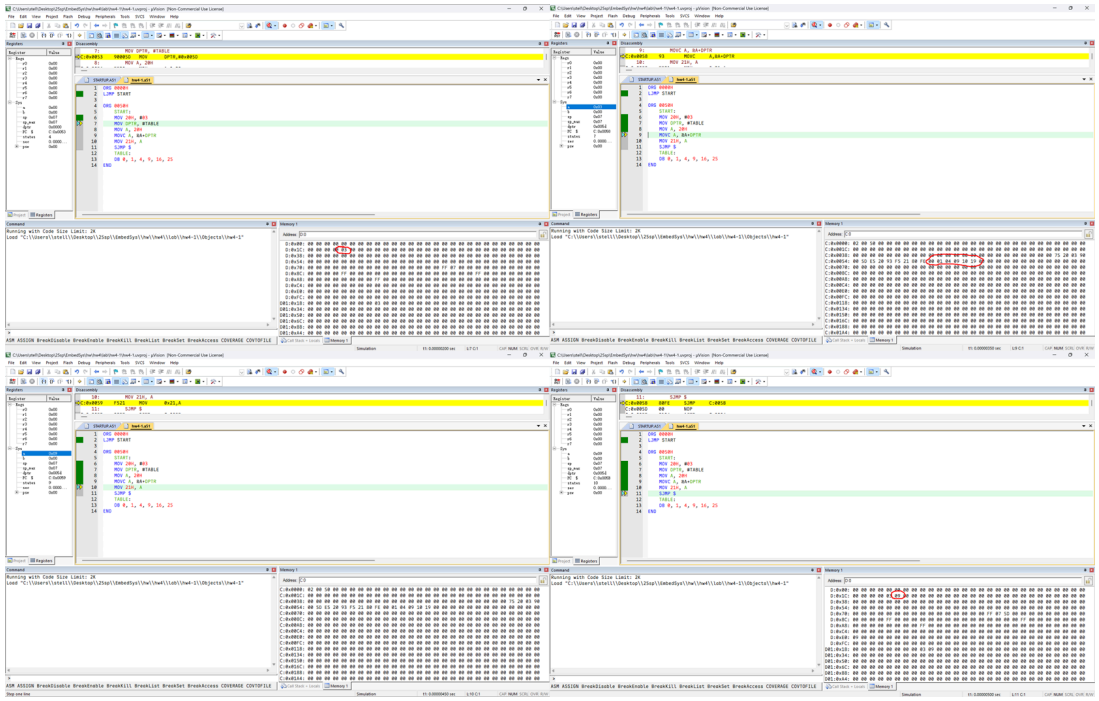


Figure 1: 运行时截图

那我们来看看运行效果，从下面的运行效果来看，左上图可见我们 MOV 20H, #03 指令将 RAM 的 20H 处成功赋值，此时 21H 为 0，然后右上图我们可见在程序本身编译后的机器码之后紧跟着的就是我们的 TABLE 平方查找表，这是最后伪指令 DB 0, 1, 4, 9, 16, 25 带来的，右下图则可见在执行了 MOVC A, @A+DPTR 后，A 已经被重新赋值成了 DPTR 所指向的，也就是 TABLE 表的第 A 位，也就是 3^2 = 9，最后右下角图我们可以看见 21H 被成功赋值为了 9

1.2 例二：多分支结构

设变量 x 以补码形式存放在片内 RAM 30H 单元中，变量 y 与 x 的关系是：

$$y=\begin{cases} x, & x>0 \\ 20H, & x=0 \\ x+5, & x<0 \end{cases}$$

写出代码:

```
ORG 0000H
LJMP START

ORG 0050H
  START:
  MOV 30H, #00000000 ;0
  MOV A, 30H
  JZ EQUAL_ZERO
  ANL A, #10000000B
  JZ PLUS_ZERO
  MOV A, #05H
  ADD A, 30H
  MOV 30H, A
  LJMP ED
  PLUS_ZERO:
  LJMP ED
  EQUAL_ZERO:
  MOV 30H, #20H
  LJMP ED
  ED:
  SJMP $
END
```

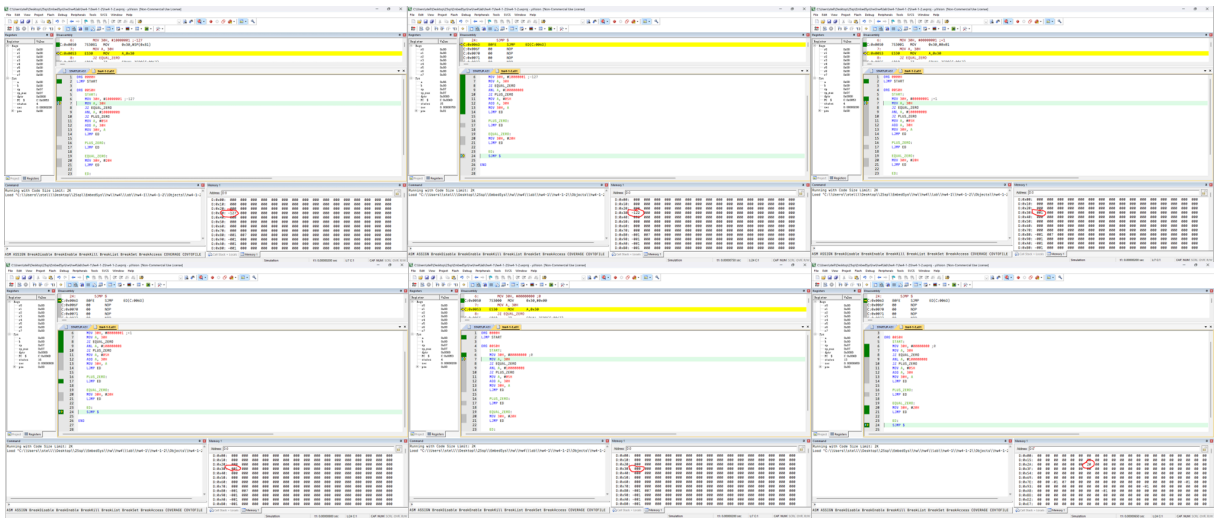


Figure 3: 运行时截图

这段代码首先将内部 RAM 的 30H 地址（存放变量 x，以补码形式） 赋值。然后根据 x 的值计算 y 并存回 30H：若 x > 0，则 y = x；若 x = 0，则 y = 20H；若 x < 0，则 y = x + 5。最后程序在 ED 标号处进入无限循环. 观察下方的运行截图，我们可以发现其完成的分段的目标(为了直观展示，前五个图的 RAM 中数值为 10 进制显示，最后一个图为 16 进制显示)

1.3 例三：循环结构(一)

50ms 延时子程序。设晶振频率为 12MHz，即机器周期为 1us。

```
ORG 0000H
LJMP START

ORG 0050H
  START:
  DEL:
  MOV R7, #200  ;1MC
  DEL1:
  MOV R6, #123  ;1MC
  NOP           ;1MC
  DJNZ R6, $    ;2MC
  DJNZ R7, DEL1 ;2MC
  LJMP START    ;2MC

END
```

那么这段的延时时间大概是：t=1+200[(1+1+2*123)+2]+2≈50000us=50ms

那么我们稍微修改一下这里的部分代码，让其可以输出方波信号，具体就是删除了 NOP 段，然后加了一个同样占用 1MC 的 CPL P1.0，然后我们按照惯例将其烧入 51 单片机中观察效果：

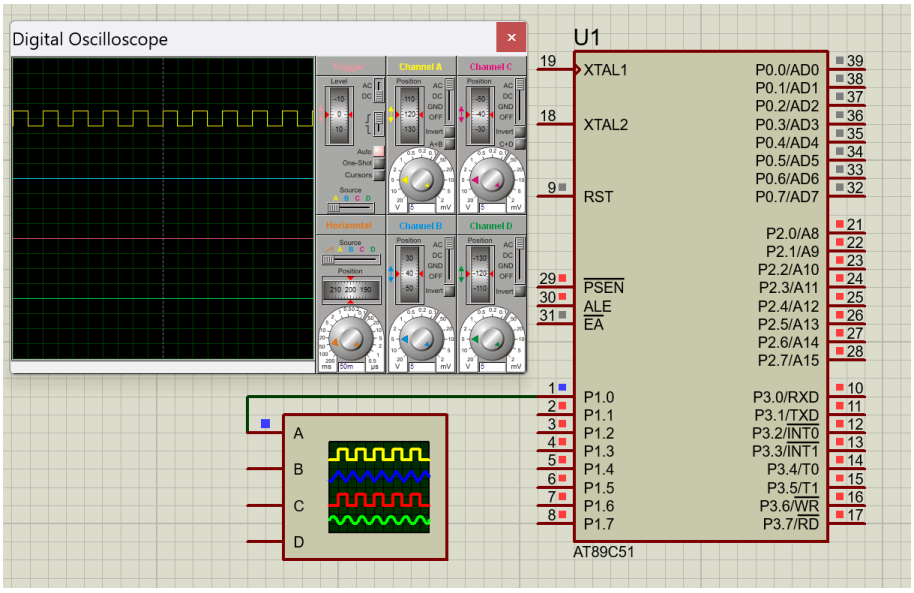


Figure 4: 电路运行截图，示波器被设置为一个小格 50ms，我们可以看见 Channel A 上确实出现了一个 100ms 为周期的方波

1.4 例四：循环结构(二)

将内部 RAM 中起始地址为 data 的数据发送到外部 RAM 中起始地址为 buffer 的存储区域中，直到发现 ‘$’ 字符，传送停止，循环次数事先不知道先判断，后执行。

写出代码:

```
ORG 0000H
LJMP START
```

```
ORG 0050H
  START:
  MOV 30H, #12H
  MOV 31H, #34H
  MOV 32H, #56H
  MOV 33H, #24H ; '$'


  MOV R0, #0030H
  MOV DPTR, #2000H
  LOOP1:
  MOV A, @R0
  CJNE A, #24H, LOOP2
  SJMP ENDEND
  LOOP2:
  MOVX @DPTR, A
  PUSH ACC
  MOV A, #00H
  MOVX A, @DPTR
  MOV R1, A
  POP ACC
  INC R0
  INC DPTR
  SJMP LOOP1
  ENDEND:
  SJMP $

END
```

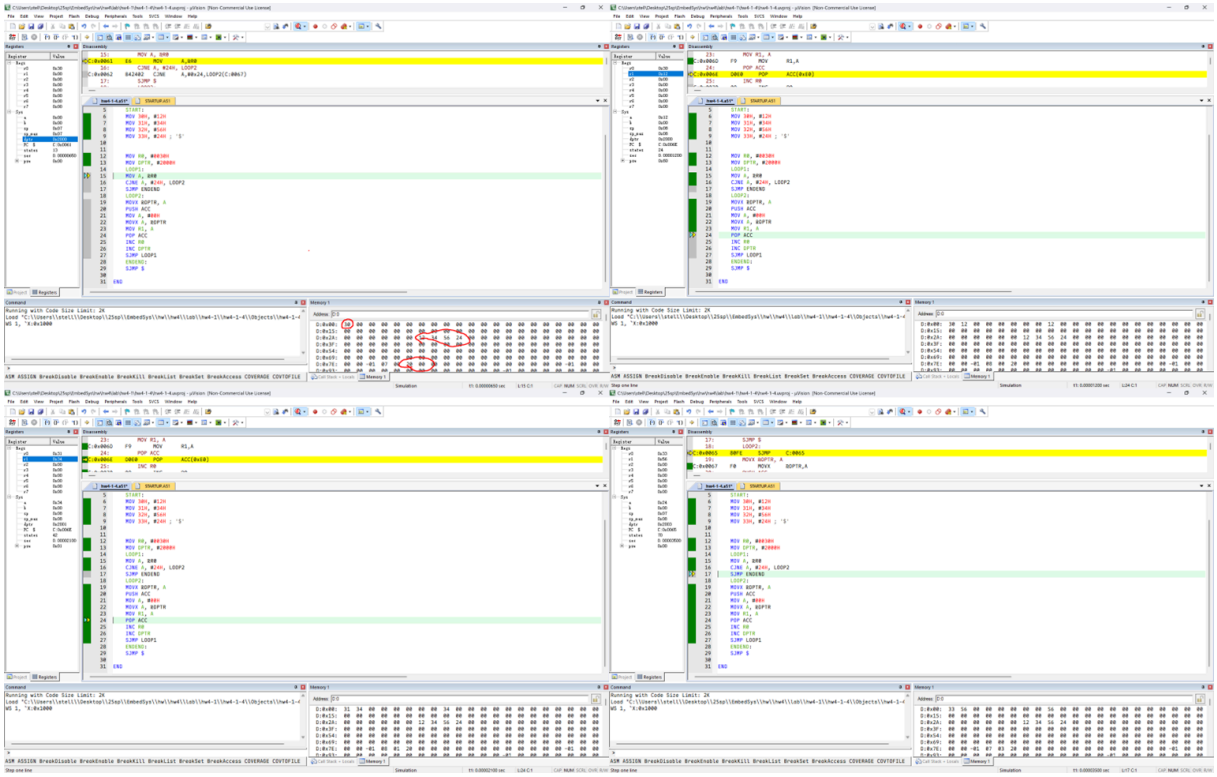


Figure 5: 运行时截图

由于我没想出来该怎么给地址直接赋值赋字符，所以就只能先赋 ‘$’ 对应的 ASCII 码数值，以及，我研究了很久都没有找到 KEIL 在调试状态下如何找到外部 RAM 而非内部 RAM，所以我们只能通过之后又把 DPTR 指向的地址的值再重新赋给 A 然后转接给 R1 来看数据是不是成功赋值到了对应的位置上😯

这段代码的作用是我们取了片内 RAM 的数据段，并把地址赋给 R0，然后我们通过间接寻址的方式逐个读取并判断是否为 ‘$’，如果不是就赋值到 DPTR 对应的地址，并将 DPTR 和 R0 两个指针同时右移，直到复制到 ‘$’ 时停止，结果也确实如此，由上面的运行图我们可以看见，在 ‘$’之前的值都被成功赋值到了 DPTR 段，而在检测到 ‘$’之后就终止了复制流程.

1.5 例五：子程序调用

利用查表法求平方和，设 a、b、c 分别存于内部 RAM 的 DA、DB、DC 三个单元中。

```
ORG 0000H
ADDR_A EQU 30H
ADDR_B EQU 40H
ADDR_C EQU 50H
LJMP START
ORG 0050H
  START:
  MOV ADDR_A, #03
  MOV ADDR_B, #04
  MOV A, ADDR_A
  ACALL SQR
  MOV R0, A
  MOV A, ADDR_B
  ACALL SQR
  ADD A, R0
  MOV ADDR_C, A
  SQR:
  MOV DPTR, #TABLE
  MOVC A, @A+DPTR
  RET
  TABLE:
  DB 0, 1, 4, 9, 16, 25, 36, 49
END
```

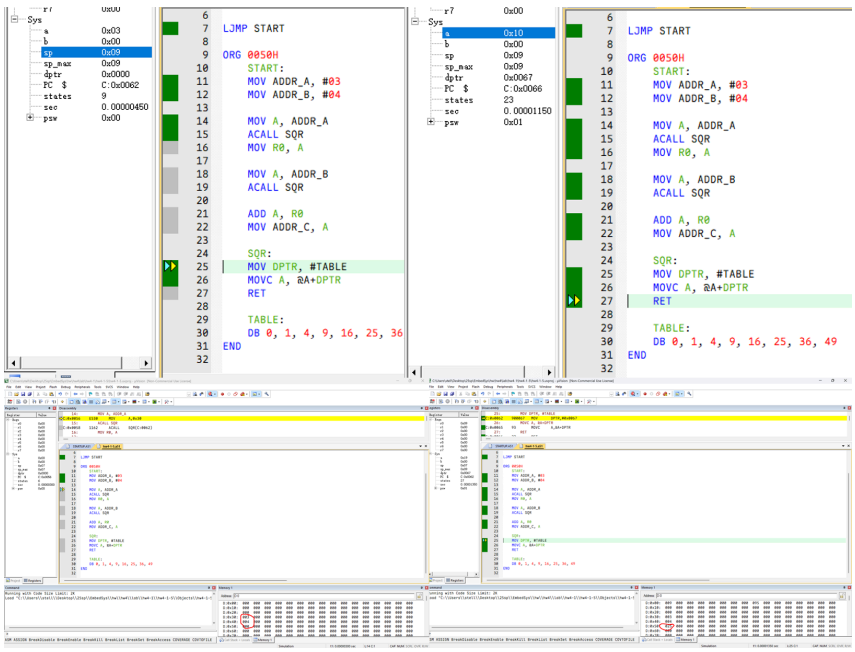


Figure 6: 运行时截图

由图可见，我们最先通过 EQU 伪指令和赋值为 ADDR_A， ADDR_B 赋初值，然后我们 A 经过两次子程序调用之后分别为 9 和 16(16 进制 10)，最后相加后写入了目标地址.

2 C/ASM 混合编程

2.1 C 语言主程序

此处我们将尝试使用 c 语言编写主程序，同时使用汇编和 c 语言实现一个同样的子程序

```c
// main.c

#include <reg51.h>

extern int sqr_C(int a, int b);
extern int sqr_ASM(int a, int b);

int main() {
  int a = 3;
  int b = 4;
  int c = 0;
  c = sqr_C(a, b);
  c = 0;
  c = sqr_ASM(a, b);
  return 0;
}
```

a51 子程序

```
; sqr_ASM.a51
?PR?FUNCTION SEGMENT CODE

RSEG ?PR?FUNCTION

PUBLIC _sqr_ASM

  _sqr_ASM:
  MOV A, R7
  MOV B, R7
  MUL AB
  MOV R7, A
  MOV R6, B
  MOV A, R5
  MOV B, R5
  MUL AB
  MOV R5, A
  MOV R4, B
  MOV A, R7
  ADD A, R5
  MOV R7, A
  MOV A, R6
  ADDC A, R4
  MOV R6, A
  RET

END
```

c 语言子程序

```c
int sqr_C(int a, int b) {
  return a*a + b*b;
}
```

c 语言的部分我们不用过多解释了， 不过这里的汇编部分有一些新的内容， 比如开头的这部分 `?PR?<PROGRAM>` 是一段伪指令，用以在多文件状态下告诉连接器在某段(?PR?)是程序段定义一个<PROGRAM>的段，`RSEG ?PR?FUNCION` 表示这个程序可以放在任何地方，RSEG 表示程序属性.

这段语言我们先给 a，b 赋值了 3，4. 然后我们先后调用了使用 C 语言写的平方和程序和汇编写的平方和程序，我们可以在下图观察结果

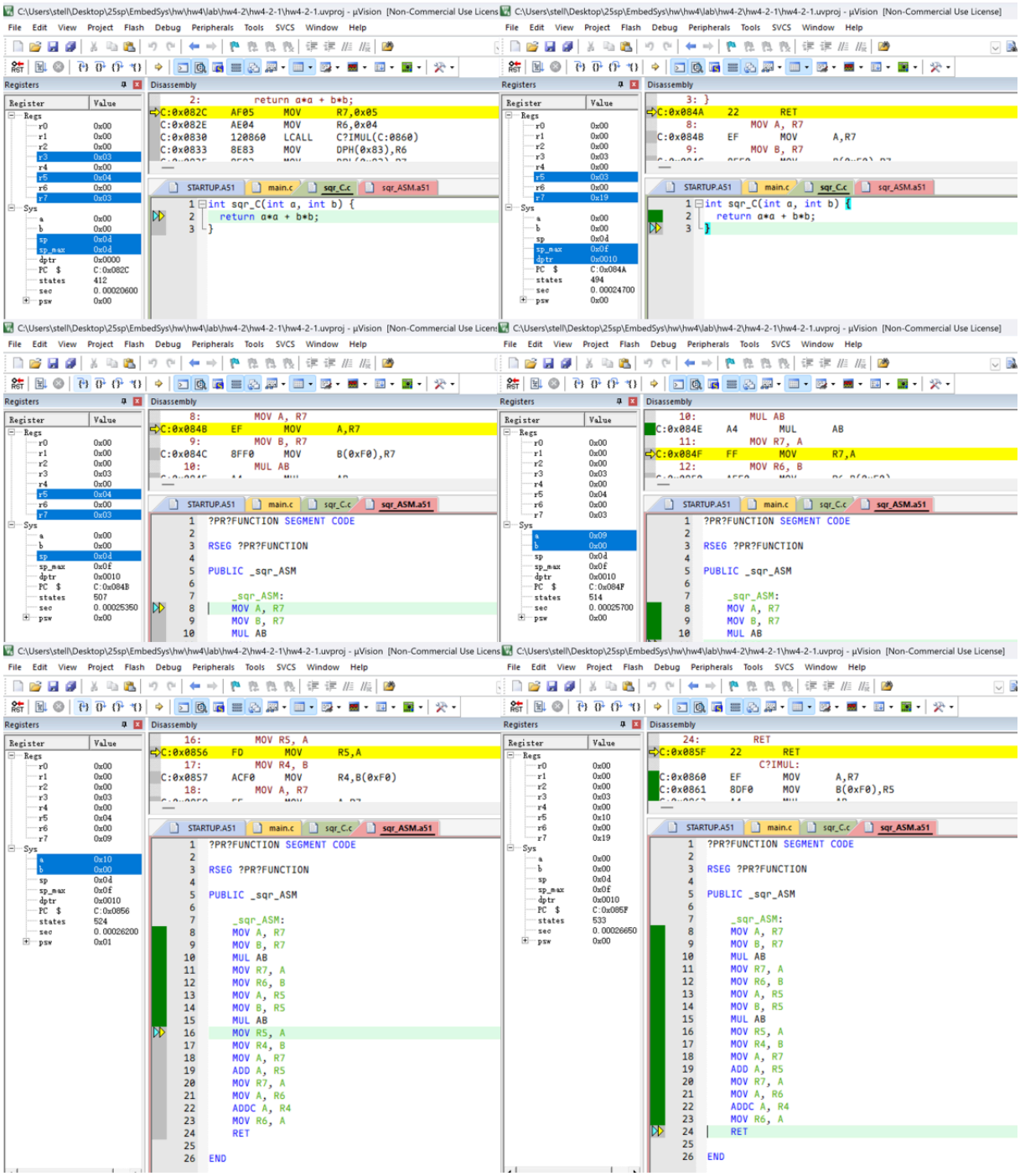


Figure 7: 运行时截图

我们可以观察在进入 c 语言调用时，可见我们传入的 3 和 4 的参数分别在 R5 与 R7，而在最后在完成平方和运算后的返回值存在了 R6(高六位，所以是 0x00)和，R7(第六位所以是 0x19，即 10 进制的 25)，而在汇编部分也是类似的传递方式，由 R5 和 R7 传入参数，最后返回时提供给 R6(高位，同样为 0x00)和 R7

2.2 a51 主程序

此处我们将尝试使用汇编来完成主程序和子程序

```
ORG 0000H
LJMP MAIN


?PR?MAIN SEGMENT CODE
  RSEG  ?PR?MAIN
EXTRN CODE (SQR)

ORG 0100H
  MAIN:
  MOV R5, #03
  MOV R7, #04
  LCALL SQR

END
```

子程序

```
?PR?SQR SEGMENT CODE
  RSEG  ?PR?SQR

ORG 1000H

  PUBLIC SQR

  SQR:
  MOV A, R5
  MOV B, A
  MUL AB
  MOV R0, A
  MOV R1, B
  MOV A, R7
  MOV B, A
  MUL AB
  MOV R2, B
  MOV B, R0
  ADD A, B
  MOV R7, A
  MOV A, R1
  MOV B, R2
  ADDC A, B
  MOV R6, A
  RET

END
```

3 中断程序的实例

接下来我们将设计一个电路，其 P1 口分别连到 8 盏灯上，初始状态下 P1.0 口的灯亮起，现在，我们需要在按下按钮时，亮灯的灯泡下移一个. 在这个任务中，由于我们一方面需要记住当前是哪个灯泡在亮，与此同时，这个按钮被按下的时间并不是固定的，因此，最简单的方式便是采用中断的方式，因此写出代码与电路如下:

```
ORG 0000H
LJMP MAIN

ORG 0013H
```

```
LJMP INT_HANDLE

ORG 0050H
  MAIN:
  SETB EA
  SETB EX1
  SETB IT1
  MOV A, #00000001B
  MOV P1, A
  SJMP $

  INT_HANDLE:
  RL A
  MOV P1, A
  RETI

END
```

这部分代码倒是挺简单的，三个 SETB 就是打开总中断，INT1 中断开关，以及将中断模式设置为边沿触发抓状态，然后我们就给 A 赋值，主程序本身就是这个结构.

当触发中断时，其会先跳转到预留的中断位，也就是这里的 0013H，然后这里 13H 就只有个转接指令转接到了我们自己写的中断处理程序片段，这里我们将 A 换移了一位，实现的下一盏灯亮的效果，然后我们就完成中断，使用 RETI 返回

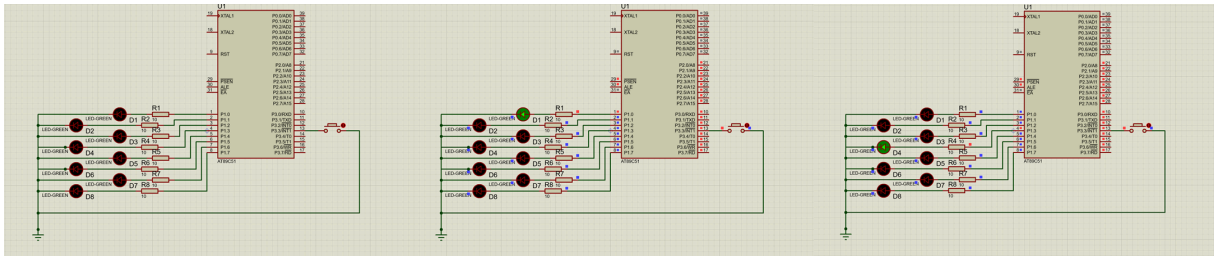


Figure 8: 电路图与运行时截图，当按钮按下时下一盏灯会亮起，符合我们的预期

3.1 改进与探索

1. 改为 INT0 触发:

   这比较简单，代码部分，需要就修改一下初始的中断位部分即可

   ```
   + SETB EX0
   + SETB IT0
   ;------------
   - SETB EX1
   - SETB IT1
   ```

   电路部分就是把右边这根本来接到 INT1 的线改接到 INT0 上

2. 中断管脚使用外部上拉电路，中断触发方式改为低电平触发

   代码也没什么好说的，把将 IT1 置一的那一行删掉即可

   ```
   - SETB IT1
   ```

   电路部分也是只需要对右边做出一些微调即可

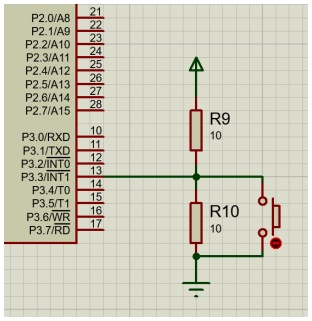
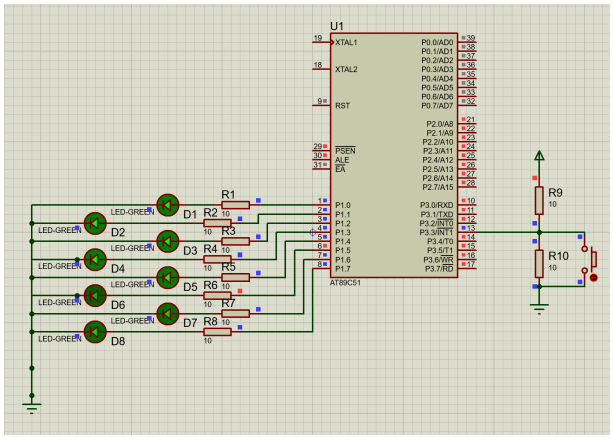


Figure 9: 外部上拉电路

不过在实际运行时出现了一点点小问题，可能按钮的设计问题，但短按产生的低电平时间也远长于数个机械周期，导致其按一次就执行了相当多次数的高频的中断触发，模拟出来的效果如下图



1. 低电平时 LED 灯亮:

   我觉得每个 LED 灯前装一个非门应该是可以的，不过我用了一个更加奇怪的办法，也就是我把接地端改为了接一个电源，并将 LED 灯全部反向，似乎从设计角度来看，当输出高电平时，电流由单片机流向电源，此时 LED 灯断路，而当输出低电平时，电流由外部电流流向电源，LED 灯短路

   我不清楚最后运行起来是不是真的是这个原理，但似乎效果符合我们的预期，那就行.

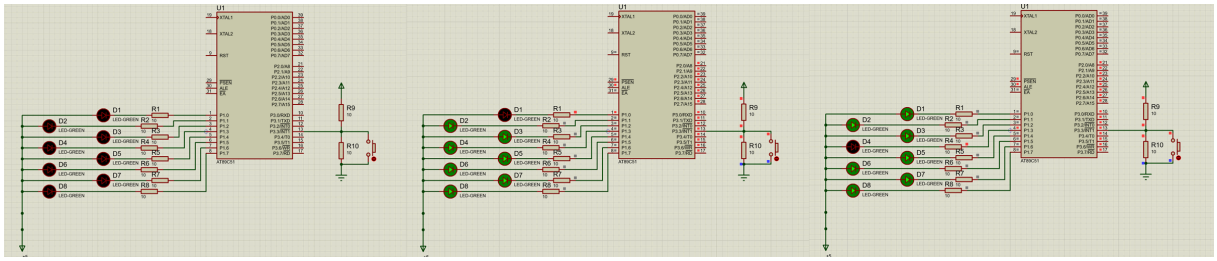


Figure 11: 电路图与运行示意