第3章

单片机的程序结构与编程

本章导读：

编程不仅要熟悉数据传输、计算等指令，还需要熟悉编译、程序结构等指令。程序结构是指程序的组成方式，主要包括顺序、分支、循环等。对程序结构的理解有助于提高程序架构能力。结构化程序是指程序组成结构化、功能模块化、运行流程化。程序需要采用结构化设计，以方便调试、移植和开发。在进行 MCS-51 系列单片机结构化程序设计时，按照总体规划和总体设计，由若干软件设计人员分别设计编程各功能模块，再依据软件结构和程序流程，将若干功能模块组成结构化程序，实现应用程序整体功能，解决复杂的实际问题。按照结构化程序设计要求编程，有助于提高软件设计人员的编程效率及编程水平。

本章围绕宏汇编指令和程序结构展开，主要介绍单片机程序的组成结构、各个部分的设计要求和注意事项以及实现结构化程序的步骤，并给出结构化程序实现案例。

本章主要内容：

51 系列单片机的宏汇编程序设计、基本伪指令、程序结构与转移指令、子程序设计及调用、C 语言程序结构设计。通过本章学习应该达到以下目的：熟悉 C 语言与汇编语言及两者区别、熟练掌握 MCS-51 系列单片机的宏汇编语言与 C 语言的程序结构与程序设计方法。

3.1 宏汇编程序设计

3.1.1 伪指令

在利用汇编语言进行编写程序时，除了使用指令系统规定的指令外，还要用到一些伪指令。伪指令与普通指令不同，它并不生成可执行的目标代码，只是对汇编过程进行某种控制或提供某些汇编信息。MCS-51 汇编语言中常用的伪指令有以下几种。

1. 定位伪指令 ORG

格式：

```
标号: ORG  地址表达式;//地址表达式表示首指令存放的起始地址
```

功能：规定程序块或数据块存放的起始位置。

例如：

```
ORG 1000H
MAIN: MOV A, R1
```

规定首指令从地址 1000H 单元开始存放，标号 MAIN 的值为 1000H。

如果需要在一个程序中规定不同程序段或数据段存放的起始位置，可以多次使用 ORG。

2. 定义字节数据伪指令 DB

格式：

```
标号: DB  字节数据表
```

功能：给数据表中的数据分配存储单元。标号所在位置为存放的起始位置，其中字节数据表可以是一个或多个字节数据、字符串或表达式，它表示将字节数据表中的数据按从左到右的顺序依次存放在指定的存储单元中，一个数据占用一个存储单元。

例如：

```
ORG 1000H
TAB1: DB 50H,30H,'ABC'
```

指令定义的数据 50H、30H、ABC 从指定的 1000H 单元开始依次存放，一个数据占用一个存储单元，字符形式数据按照 ASCII 码存放。定义后存放结果为(1000H)50H、(1001H)30H、(1002H)41H、(1003H)42H、(1004H)43H。

3. 定义字数据伪指令 DW

格式：

```
标号: DW  字数据表
```

该伪指令功能与 DB 伪指令类似，DW 伪指令是给数据表中的数据分配存储单元。不同的是，DB 伪指令定义的数据为字节，而 DW 伪指令定义的数据为字，即两个字节。一个字数据占用两个连续的存储单元，先将高 8 位数据存入低地址单元，后将低 8 位数据存入高地址单元。

例如：

```
ORG 1000H
TAB1: DW 2347H,2CH
```

可以注意到数据 2CH 高位为 0，因此在传输数据时需要将高位补齐后再传输。定义后存放结果为 (1000H)23H、(1001H)47H、(1002H)00H、(1003H)2CH。

4. 符号定义伪指令 EQU

格式：

```
NAME  EQU  EXPRESSION
```

这条伪指令表示将表达式的值或特定的某个汇编符号定义为一个指定的符号名。由EQU伪指令定义过的符号名可以在本程序段的任意位置引用。

例 3.1　使用指令 EQU 将值 30H 赋给符号 A，并对 A 进行运算。

解：

```
ABC  EQU  30H          ;将 ABC 定义为 30H
MOV  A, #20H           ;将立即数 20H 赋值给 A,A 内存数为 20H
MOV  30H, A            ;将 A 的值(20H)存入地址为 30H 的存储单元
MOV  A, #ABC           ;将数 30H 赋值给 A 赋值
MOV  A, ABC            ;将地址为 30H 的存储单元的内容赋值给 A
```

5. 汇编结束伪指令 END

格式：

```
标号: END          //标号可省略
```

END 伪指令是汇编语言源程序的结束标志。在每一个源程序的末尾都要写上，汇编程序运行到 END 时，自动结束对本程序的处理，END 后面的语句将不再编译。

6. 位地址符号定义伪指令 BIT

格式：

```
符号名      BIT      地址表达式
```

功能：将位地址赋给所定义的符号名。

例如：

```
SIG  BIT  P2.1
```

上述指令表示将 P2.1 的位地址赋给符号名 SIG，赋值后编程中可以用 SIG 代替 P2.1 使用。

3.1.2　宏汇编语言格式

宏汇编语言格式与其他汇编语言格式类似，通常由标号、操作码、操作数和注释 4 项内容组成，其表示格式为：

```
标号: 操作码  操作数 ;注释
```

1. 标号

MCS-51 汇编语言中的标号是程序地址的标记，可以由其他指令引用此标号，实现控制程序的转移、寻址与程序调用。在汇编过程中，标号将直接编译为标号所在语句的实际地址。

标号使用规则：

(1) 由 1～8 个字符组成，且第一个字符必须是字母。

(2) 标号中允许使用的字符是英文字母、阿拉伯数字和下画线。如 WS、x_y、PQ 都为合法的标号。

(3) 标号是可供选择的项,语句中可以不加。

(4) 标号后面必须加上冒号。

(5) 标号不能使用在汇编语言中已经定义过的符号名,如指令助记符、寄存器名。

(6) 标号只在本程序中有效,并且在同一个程序中不能使用相同的标号。

2. 操作码

用于规定该语句执行的操作功能,由指令助记符或伪指令表示。

3. 操作数

操作数给出参与操作的数据或地址。它可以是 3 个、2 个、1 个或没有,当有 2 个及以上的操作数时,它们之间用逗号分隔或空格符分隔。

4. 注释

注释项用于对指令功能或程序段功能做简要标记和说明。该项内容计算机不予处理,仅是为了便于阅读程序,用分号与操作数分隔。

3.2 单片机程序结构设计

在程序设计中,不论程序如何复杂,整体上均由顺序结构、分支结构和循环结构 3 种基本结构组合而成。

3.2.1 顺序结构

顺序结构程序是指一种无分支的直线程序,程序的执行是按程序计数器自动加 1 的顺序执行,它主要包括数据传送指令和数据运算类指令。

例 3.2 将内部 RAM 51H、52H 两个单元中的无符号数相加,存入 R0(高位)及 R1(低位)。

考虑到两个单元单字节数相加之和可能超过一个字节,因此要按双字节来处理。

程序流程图如图 3-1 所示。

源程序如下:

```
MOV     A,51H           ;取 51H 单元值给 A
ADD     A,52H           ;把 A + (52H)的值给 A,并影响 CY
MOV     R1,A            ;暂存于 R1 中
CLR     A               ;A 清 0
ADDC    A,#00H          ;CY 送入高位
MOV     R0,A            ;高位存入 R0
```

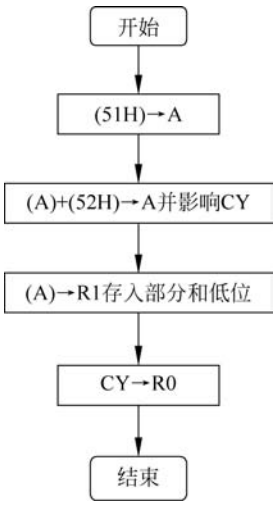

图 3-1 单字节加法流程图

3.2.2 分支结构

所谓分支程序,就是利用条件转移指令,使程序执行某一指令后,根据条件是否满足来改变程序执行的顺序。在 MCS-51 指令系统中直接用于判断分支条件的指令有累加器判零条件转移指令 JZ(JNZ),比较条件转移指令 CJNE,位条件转移

指令 JC(JNC)、JB(JNB)、JBC 等。

例 3.3　设变量 X 存于内部 RAM 50H 单元，函数值 Y 存于 51H 单元，试根据下式对 Y 赋值。

$$Y=\begin{cases}2, & X>0\\0, & X=0\\X, & X<0\end{cases}$$

在程序中，要根据 X 的值给 Y 赋值。先把变量 X 从内存中取出来并用符号 A 代替，然后判断是否等于 0。若是 0，则 A 无须处理，若不为 0，再判断是否大于 0。若大于 0，则令 A＝2，反之 A 无须处理。判断完成后再将 A 中的值传入 Y 中并保存。相应的流程图如图 3-2 所示。

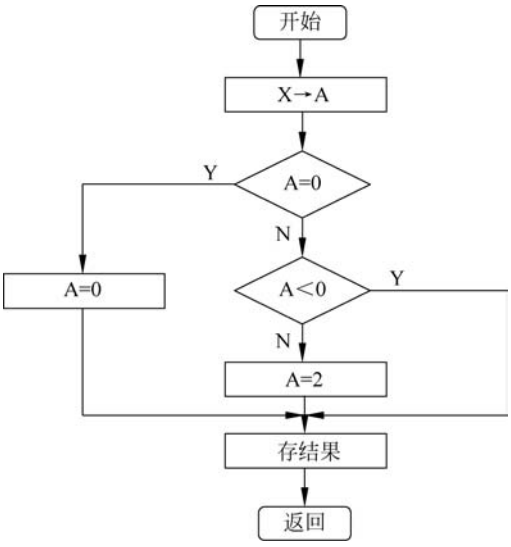

图 3-2　分支结构流程图

源程序如下：

```
MOV     A, 50H              ;取数
JZ      ZERO                ;(A)为 0,则转至 ZERO
JB      ACC.7, STORE        ;(A)为负数,则转至 STORE
MOV     A, #02H             ;(A)为正数,则赋值 2
SJMP    STORE
ZERO:   MOV A, #0
STORE:  MOV 51H, A
```

3.2.3　循环结构

在程序中，往往要求某一段程序重复执行多次，这时就可以利用循环程序结构完成，在很大程度上简化程序结构，减少程序占用的存储单元数。一个循环结构由以下 3 部分组成。

1. 循环体

重复执行的程序段部分，又分为循环工作部分和循环控制部分。循环控制部分每循环一次检查循环结束条件，当满足条件时就跳出循环，顺序执行其他程序；当不满足条件时，顺序执行循环工作部分。

2. 循环结束条件

在循环程序中必须给出循环结束条件，否则程序就会进入死循环。单片机中最常见的循环是计数循环，当循环到指定次数后就结束循环。在循环初始部分将循环次数置入计数器中，每循环一次计数器减 1，当计数器内容减到 0 时循环结束。除此之外，有些循环程序中事先无法知道循环次数，而只知道循环有关的条件，此时只能根据给定的条件判断循环是否继续，一般可参照分支程序设计方法中的条件判别指令实现。

3. 循环初态

循环初态是用于循环过程的工作单元，在循环开始时大多要置以初态即赋初值，此步骤也是循环程序中的重要环节。循环初态可以分成两部分：循环工作部分初态与结束条件的初态，例如设置地址指针，使某些寄存器清 0，或设某些标志等。

在循环程序中，控制循环的方法因问题的条件不同而有若干种。

1）计数器控制循环

例 3.4 从 32H 单元开始存放一个数据块，其长度存放在 25H 单元，编写一个数据块求和程序，要求将和存入 26H 单元，设和不超过 255。

解：程序初始化部分，定义和的符号为 A，将数据取出并保存。由于数据依次保存在不同的单元，需要编写循环结构依次取出，且每取出一个数据将其与 A 求和。数据全部取出后将 A 输出即可。

程序流程图如图 3-3 所示。

源程序如下：

```
    CLR     A
    MOV     R2, 25H             ;取字节数
    MOV     R1, 32H             ;取地址
LOOP: ADD   A, @R1              ;求和
    INC     R1                  ;地址加 1
    DJNZ    R2, LOOP            ;控制循环是否结束
    MOV     26H, A              ;存入结果
```

2）条件控制循环

例 3.5 设字符串放在内部 RAM 地址为 30H 开始的单元中，以“$”作结束标志，现要求计算该字符串长度，并将计算结果放在 25H 单元中。

程序流程图如图 3-4 所示。

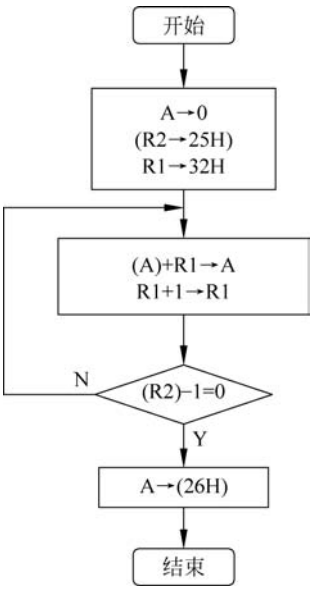

图 3-3 求和程序流程图

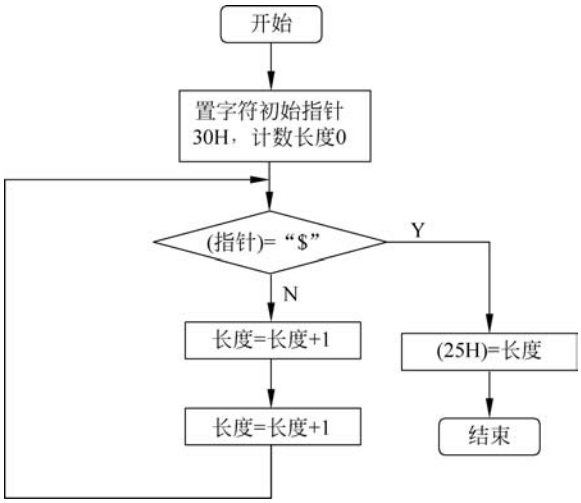

图 3-4 求字符串长度程序流程图

源程序如下：

```
    CLR     A
    MOV     R0, #30H            ;取数
LOOP: CJNE  @R0, #24H, NEXT     ;与"$"(ASCII 值为十六进制 24)比较
```

```
        SJMP    COMP            ;找到"$"结束
NEXT:   INC     A               ;若不为"$",则计数器加1
        INC     R0              ;修改地址指针
        SJMP    LOOP
COMP:   MOV     25H,A           ;存结果
```

3.2.4　子程序设计及调用

子程序是程序设计中经常使用的程序结构，通过将一些固定的或经常使用的功能做成子程序的形式，使源程序及目标程序大大缩短，从而提高程序设计的效率和可靠性。

对于一个子程序，应注意它的入口参数和出口参数。入口参数由主程序传给子程序，而出口参数是子程序运算完传给主程序的结果。另外，子程序所使用的寄存器和存储单元往往需要保护，防止返回后影响主程序的运行。

主程序可在不同的位置通过 ACALL 或 LCALL 指令多次调用子程序，通过子程序中的最后一条指令返回指令 RET，重新返回到主程序中的断点地址，继续执行主程序。所谓断点地址，是子程序调用指令的下一条指令的地址，取决于调用指令的字节数，可以是 PC+2（对应 ACALL 指令）或 PC+3（对应 LCALL 指令），这里的 PC 是指调用指令所在的地址。

主程序在调用子程序时，一方面初始数据要传给子程序，另一方面子程序运行结果要传给主程序，因此，主程序和子程序之间的参数传递是非常重要的。

参数传递一般有 3 种方法实现：

- 利用寄存器传递，此方法将所需传递的参数直接放在主程序的寄存器中传递给子程序，此方法也是最常见的方法。
- 利用存储单元传递，此方法将所需传递的参数直接放在子程序调用指令代码之后。
- 利用堆栈传递，此方法将参数压入堆栈，在子程序运行时从堆栈中取参数。

在调用子程序时应注意现场保护问题。在子程序运行时，可能要使用累加器和某些寄存器，而在主程序调用子程序前，这些寄存器可能存放有主程序的中间结果，它们在子程序返回后仍需使用。这就使得在进入子程序时，将累加器和某些工作寄存器的内容转移到安全区域保存起来就变得非常必要了，即保护现场操作。当子程序执行完成返回主程序之前，再将这些内容取出，送回到累加器和原来的工作寄存器中，即恢复现场操作。

保护现场和恢复现场通常使用堆栈操作，即在进入子程序时，将需要保护的数据压入堆栈加以保护，在返回主程序之前将压入的数据弹出到原来的工作单元中，恢复其原来的状态。由于堆栈操作是遵循“先进后出”原则，因此先压入堆栈的参数应该后弹出，才能保证恢复原来正确的状态。

下面通过实例说明子程序设计及参数传递方法。

例 3.6　在 MAA 单元有两个十六进制数，将它们分别转换成 ASCII 码存入 ASC 及 ASC+1 单元。例如，MAA 单元存数为 31H，分别转换成 ASCII 码为（ASC）33H 和（ASC+1）31H。

解：因为要进行两次转换，所以可用子程序来完成，参数传递用堆栈来完成。

```
STAT:   PUSH    MAA             ;将十六进制参数压入堆栈
        ACALL   CNV             ;调用转换子程序
        POP     ASC             ;返回参数送 ASC 单元
```

```
        MOV         A,MAA       ;MAA 单元内容送 A
        SWAP        A           ;高、低 4 位交换
        PUSH        ACC         ;将第 2 个十六进制数压入堆栈
        ACALL       CNV         ;再次调用子程序
        POP         ASC + 1     ;存第 2 个 ASCII 码
WAIT:
        SJMP        WAIP        ;暂停
                                ;通过堆栈传递数据子程序
CNV:
        POP     ACC             ;弹出参数到 A
        ANL     A,#0FH          ;屏蔽高 4 位
        CJNE    A,#0AH,NEXT     ;十六进制数转换成 ASCII 码
        AJMP    L1
NEXT:
       ADD      A,#37H
       SJMP     L2
L1:    ADD      A,#30H
L2:    PUSH     ACC             ;参数压入堆栈
       RET
```

主程序是通过堆栈将要转换的十六进制数传到子程序，并将子程序转换的结果再通过堆栈送回到主程序。采用该方式，只需要在调用前将入口参数压入堆栈，在调用后将返回参数弹出堆栈即可。

3.2.5 程序的控制转移指令

MCS-51 有较丰富的控制转移指令，除了按布尔变量控制程序转移的指令外，控制程序转移类指令共有 17 条，包括全存储空间的长调用与长转移指令、按 2KB 分块的程序空间内的绝对调用与绝对转移指令、全空间的长相对转移与一页范围的短相对转移、条件转移指令。

1. 无条件转移类指令

无条件转移类指令是相对于条件转移类指令而言的，主要包括下面 3 条指令。

短转移类指令：AJMP addr11

长转移类指令：LJMP addr16

相对转移指令：SJMP rel

上面 3 条指令的使用方法和区别如下：JMP 标号，即跳转到一个标号处，三者区别在于跳转的范围不一样。LJMP 至多跳转 64KB，AJMP 至多跳转 2KB，SJMP 则至多跳转 256B。原则上，所有用 SJMP 或 AJMP 的地方都可以用 LJMP 来替代，因此在初学时，需要跳转时可以全用 LJMP。在存储空间受限时，LJMP、AJMP 和 SJMP 不能混用。AJMP 是一条双字节指令，该指令占用存储器(ROM)的两个单元。而 LJMP 则是三字节指令，即指令占用存储器的 3 个单元。

例 3.7 分析以下指令在每一步运行前后数据与地址的变化。

```
ORG     0800H
WR:     AJMP addr11
        SJMP rel
        LJMP 4200H
```

分析：第一条指令执行时，先将 PC 的内容加 2，PC 值为 0802H（二进制为 0000 1000 0000 0010B），然后 PC 值的高 5 位（00001）与指令中 addr11 代表的 a10～a0 的 11 位地址组合成 16 位绝对地址（PC15～11，a10～a0）。

使用此指令时需注意，转移到的地址必须和 PC 内容加 2 后的地址处在同一个 2KB 区域。假如 addr11＝001 0010 0000B，标号 WR 地址是 1030H（二进制为 0001 0000 0011 0000B），则执行该指令后，程序转移到 1120H（二进制为 0001 0001 0010 0000B）。addr11 仍为 001 0010 0000B，当 WR 为 3030H（二进制为 0011 0000 0011 0000B），则执行该指令后，程序转移到 3120H（二进制为 0011 0001 0010 0000B）。

第二条指令转移的目的地址＝源地址＋2＋rel。此处 rel 是一个 8 位带符号数，因此可向前或向后转移，转移的范围为－128～127。例如，在 0100H 单元有一条 SJMP 指令，若相对地址为 18H，则将转移到 0102H＋18H＝011AH 地址上。由于该指令给出的是相对转移地址，因此在修改程序时，只要相对地址不变就不需要改动。

最后一条指令是长跳转指令，它无条件地转向指定地址。在 64KB 程序存储器空间的任何地方都可以作为转移的目标地址，该指令对标志位无影响。

2. *间接转移指令*

格式：

```
JMP @A + DPTR
```

该指令的用途也为跳转，相当于 C 语言的 switch-case 语句，这里以实例帮助理解。

例 3.8　采用 JMP 语句，R0 * 2 的值做相对地址，跳转到对应程序入口，并执行该程序。

示例程序：

```
    MOV   DPTR, #TAB          ;将 TAB 所代表的地址送入 DPTR
    MOV   A, R0               ;从 R0 中取数(详见下面的分析)
    MOV   B, #2
    MUL   A, B                ;A 中的值乘 2(详见下面的分析)
    JMP   A, @A + DPTR        ;跳转
TAB: AJMP S1                  ;跳转表格
    AJMP S2
    AJMP S3
```

分析：在单片机开发中，经常要用到键盘，如图 3-5 所示的 9 个按键的键盘，要求当按下功能键 A～G 时实现不同的功能，即按下不同的键执行不同的程序段。键盘程序将在后续章节详细分析，这里仅需了解不同按键对应不同且唯一的子程序即可。

A	B	C
D	E	F
G	H	I

图 3-5　一种常用键盘示意图

如图 3-6 所示，前面的程序读入的是按键的值，如按下 A 键后获得的键值是 0，按下 B 键后获得的值是 1 等，然后根据不同的值进行跳转，若键值为 0 则转到 S1 执行，若为 1 则转到 S2 执行，以此类推。

先从程序后半部分看，这若干个 AJMP 语句最后在存储器中存放示意图如图 3-7 所示，即每个 AJMP 语句均占用两个存储器的空间，并且连续存放。而 AJMP S1 存放的地址是 TAB，而我们不需要知道 TAB 等于多少。

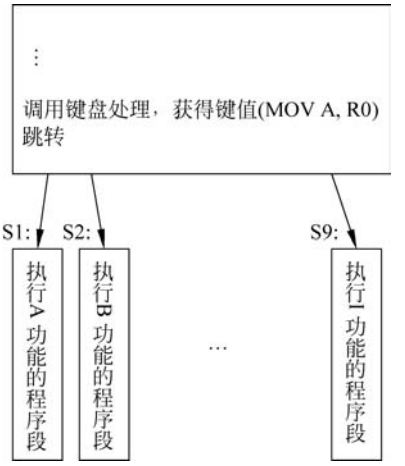

图 3-6　程序代码说明图

ROM	地址
.	.
.	.
.	.
AJMP　S4	TAB+6
	TAB+5
AJMP　S3	TAB+4
	TAB+3
AJMP　S2	TAB+2
	TAB+1
AJMP　S1	TAB

图 3-7　程序存放地址示意图

执行过程如下，第 1 条指令“MOV DPTR，＃TAB”执行完成后，DPTR 中的值为 TAB。第 2 条指令“MOV A，R0”，假设 R0 是由按键处理程序获得的键值，如按下 A 键，R0 中的值是 0。现在假设按下的是 B 键，在执行完第 2 条指令后，A 中的值就是 1，应当执行 S2 这段程序。第 3 条与第 4 条指令是将 A 中的值乘 2，即执行完第 4 条指令后 A 中的值是 2。下面就执行 “JMP @A＋DPTR”指令，现在 DPTR 中的值是 TAB，而 A＋DPTR 后就是 TAB＋2，因此执行此语句后，将会跳到 TAB＋2 地址处继续执行，在 TAB＋2 地址中是 AJMP S2 指令，立即执行 AJMP S2 指令，程序将跳到 S2 处往下执行，这与我们的要求相符合。

这样用指令“JMP @A＋DPTR”就实现了按下一键跳到相应的程序段去执行的要求。这里需要注意，由于 AJMP 是双字节指令，因此取得键值之后要乘 2。

3. *条件转移指令*

条件转移指令是指在满足一定条件时进行相对转移的部分指令集合。

1）判 A 内容是否为 0 转移指令

JZ rel：若累加器 A 的内容为 0，则跳转到偏移量所表示的目的地址；若累加器 A 的内容不为 0，则顺序执行。

JNZ rel：若累加器 A 的内容非 0，则跳转到偏移量所表示的目的地址；若累加器 A 的内容为 0，则顺序执行。

JC rel：若进位标志 CY 为 1，则跳转到偏移量所表示的目的地址；否则顺序执行。

JNC rel：若进位标志 CY 为 0，则跳转到偏移量所表示的目的地址；否则顺序执行。

JB bit，rel：三字节指令，第一个字节为操作码，第二个字节为位地址，第三个字节是相对地址。若位地址所指定的位为 1，则跳转到相对地址所指定的单元；否则顺序执行指令。该操作对标志位无影响。

JNB bit，rel：若位地址所指定的位为 0，就跳转到相对地址所指定的单元；否则顺序执行指令。该操作对标志位无影响。

JBC bit，rel：若位地址所指定的位为1，就跳转到相对地址所指定的单元，并将该位清0。

2）比较转移指令

比较转移指令的基本格式如下：

```
CJNE A, #data, rel
```

指令的功能是将A中的值和立即数data比较，如果两者相等则顺序执行；反之转移。利用这条指令可以判断两数是否相等。在程序中，可以将rel直接写成程序标号，编译程序会自动计算对应的rel，即CJNE A，#data。本指令相当于先计算A-#data，并改变状态标志位，但不保留结果，所以能通过进位标志CY判断比较的两数大小，若前面的数大，则CY=0，反之CY=1，因此在程序转移后再次利用CY就可判断出A中的数比data大还是小了。

例3.9　若R0≥#10H，跳转到程序段L3，否则跳转到L2，请编写程序完成以上功能。

分析：如果A=10H，则顺序执行即R1=0FFH，反之则转到L1处继续执行。在L1处，再次进行判断，如果A>10H，则CY=1，顺序执行，即执行MOV R1，#0AAH指令。如果A<10H，则将转移到L2处执行，即执行MOV R1，#0FFH指令。因此本程序执行前，如果R0=10H，则R1=0FFH。如果R0>10H，则R1=0AAH。如果R0<10H，则R1=0FFH。

参考程序：

```
    MOV A, R0
    CJNE A, #10H, L1
    MOV R1, #0FFH
    AJMP L3
L1: JC L2
    MOV R1, #0AAH
    AJMP L3
L2: MOV R1, #0FFH
L3: SJMP L3
```

注：指令JC这条指令的原型是JC rel，作用与JZ类似，通过判断CY的值决定转移与否。若CY=1，则转移到JC后面的标号处执行；反之，则顺序执行。

其他的比较转移指令：

```
CJNE A, direct, rel       ;将A当中的值和直接地址中的值比较
CJNE Rn, #data, rel       ;将直接地址中的值和立即数比较
CJNE @Ri, #data,rel       ;将寄存器间接寻址得到的数和立即数比较
```

例3.10　分析以下3条指令所执行的功能。

```
CJNE A, 10H
CJNE 10H, #35H
CJNE @R0, #35H
```

解：第1条指令为将A中的值和10H中的值比较，注意指令中为10H，与#10H不同，

比较时该指令为与地址为10H中的值进行比较。同理,第2条指令为将10H中的值与35H进行比较。第3条指令为将R0中的值作为地址,从此地址中取数并与35H比较。

3) 循环转移指令

循环转移指令基本格式:

```
DJNZ Rn, rel          DJNZ direct,rel
```

第1条指令的操作是:(PC)=(PC)+2,(Rn)=(Rn)-1,如果(Rn)≠0,则(PC)=(PC)+rel,转移到规定的地址单元;如果(Rn)=0,则顺序执行。第2条指令的操作是:(PC)=(PC)+3,(direct)=(direct)-1,如果(direct)≠0,则(PC)=(PC)+rel,转移到规定的地址单元;如果(direct)=0,则顺序执行。这组指令区别是第1条指令为2字节指令,第2条指令为三字节指令。

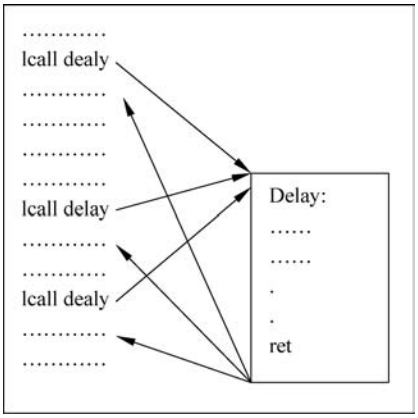

图3-8 主程序调用子程序过程示意图

(1) 调用与返回指令。

以生活中实例分析:数学老师布置了10道算术题,经观察每一道题中都包含一个(47-9)×3的运算,可以有两种方案供选择。第一种选择,每做一道题,都把这个算式算一遍;第二种选择,可以先把这个结果(即57)算出来,放在一边,当需要用到该算式时将57代入。在这种情况下第二种选择更适合,设计程序时也是如此,有时一个功能会在程序的不同地方反复使用,就可以把这个功能做成一段程序,每次需要用到这个功能时就调用一下。

调用及返回过程:主程序调用了子程序,子程序执行完之后必须再回到主程序继续执行,回到调用子程序的下面一条指令继续执行,如图3-8所示。

(2) 调用指令。

调用指令包括长调用指令:

```
LCALL addr16
```

短调用指令:

```
ACALL addr11
```

第1条指令用于无条件地调用位于指定地址的子程序,该指令的转移范围与LJMP指令的转移范围相同。第2条指令调用首地址由a10～a0所指出的子程序,指令的转移范围与AJMP指令的转移范围相同,它所调用的子程序的起始地址必须与ACALL之后指令的第一个字节在同一个2KB区域的程序存储器中,下面用例子来加以说明。

例3.11 设(SP)=50H,标号STRT值为0213H,子程序ADD1位于0234H。分析以下语句执行前后地址与数据的变化。

语句一,STRT: ACALL ADD1

语句二,STRT: LCALL ADD1

解:语句一的执行过程为:(PC)=(PC)+2

(SP)=(SP)+1，((SP))=(PC7～0)
(SP)=(SP)+1，((SP))=(PC15～8)
(PC10～0)=addr10～0
(PC15～11)不变

结果为(SP)=52H，内部 RAM 中堆栈区内(51H)=15H，(52H)=02H，(PC)=0234H。

设(SP)=50H，标号 STRT 值为 0213H，标号 ADD1 值为 0234H。

语句二的执行过程为：(PC)=(PC)+3
(SP)=(SP)+1，((SP))=(PC7～0)
(SP)=(SP)+1，((SP))=(PC15～8)
(PC)=addr15～0

结果(SP)=52H，堆栈区内(51H)=16H，(52H)=02H，(PC)=0234H。

(3) 返回指令。

ret：从堆栈中取出断点，将值传给计数器 PC，SP 值减 2，使程序从断点处执行。

(4) 空操作指令。

nop：空操作，PC 值增 1，停一个周期，一般用作短时间的延时。

3.2.6　查表程序及相应指令

在单片机应用系统中，查表是一种常用程序，它广泛用于打印机打印、计算、转换等各种场合，而且它的编程效率有时也比编写运算程序要高得多。

查表通常包括 3 步操作，若常用 DPTR 作基址寄存器，首先要将表格的首地址送入 DPTR，其次将访问项的偏移值装入累加器 A 中，最后执行“MOVC A，@A+DPTR”，查表结果送回累加器 A 中。如果表格长度不超过 256B，则 DPTR 值固定为表格的首地址即可。若表格数据长度超过 256B，则需变更 DPTR 值。现举例说明查表程序的用法。

例 3.12　设某仪表的键盘扫描程序中，不同按键对应不同且唯一的入口地址。设键值在 40H 单元中，转换后入口地址存放在 41H 及 42H，对应关系如下：

键值：　0　　1　　2　　3　　4　　5　　…　　9

入口：0100　0110　0220　0330　0440　0550　…　0990

试编写程序完成以上功能。

解：参考源程序如下：

```
    MOV    DPTR, #TABL          ;指向表首
    MOV    A, 40H               ;取得键值
    RL     A                    ;乘 2 作表偏移量
    MOV    40H, A               ;存偏移量
    MOVC   A, @A + DPTR         ;查表取得高 8 位地址
    MOV    41H, A               ;存高 8 位地址
    INC    DPTR                 ;指向入口低 8 位
    MOV    A, 40H               ;取偏移量
    MOV    A, @A + DPTR         ;查表得低 8 位地址
    MOV    42H, A               ;存低 8 位地址
    RET
TABL: DB 01H
```

```
DB 00H                    0 键入口
DB 01H
DB 10H                    ;1 键入口
DB 02H
DB 20H                    ;2 键入口
…
DB 09H
DB 90H                    ;9 键入口
```

当数据指针 DPTR 另有它用、所查表格为位于源程序邻近的较小表格(偏移量不超过256)时,可采用 PC 内容为基地址的查表指令。使用步骤为先将所查表中访问项的偏移值装入累加器 A 中,然后使用“ADD A, #DATA”指令进行地址调整,最后执行“MOVC A, @A+PC”指令进行查表,结果送回 A 中。其中 DATA 是查表指令“MOVC A, @A+PC”执行以后的地址到所查表的首地址之间的距离,即算出这两个地址之间其他指令所占的字节数,将这个值作为 DATA 值。

3.3 C51 程序结构

3.3.1 文件包含与编译

1. 包含文件

文件包含命令的功能是将指定的文件插入该命令行位置,从而使指定文件和当前程序文件连成一个源文件。文件包含命令将一个大的程序分为多个模块,由多个程序员分别编程。将一些重要的公用的符号常量或宏定义组成一个文件,在其他文件的开头使用文件包含命令包含该文件,可以提高编程效率和可靠性。

文件包含命令行的形式为:

```
#include  "文件名"
```

例如:

```
#include  "stdio.h"
#include  "math.h"
```

文件包含的要求:

- 包含命令中的文件名可以用双引号括起来,也可以用尖括号括起来。使用尖括号如 #include<math.h>,表示在由用户在设置环境时设置的包含文件目录中去查找,而不在源文件目录去查找。使用双引号,如 #include "stdio.h",则表示先在当前的源文件目录中查找,若未找到才到包含目录中去查找。
- 一个 include 命令只能指定一个被包含文件,文件包含允许嵌套,即在一个被包含的文件中又可以包含另一个文件。

C51 常用的库文件及作用如表 3-1 所示。

表 3-1　C51 常用的库文件及作用

名　　称	作　　用
reg51.h	定义 MCS-51 系列单片机的特殊功能寄存器和位寄存器
reg52.h	定义 52 单片机的特殊功能寄存器和位寄存器
math.h	定义常用数学运算
ctype.h	定义常用字符函数
stdlib.h	定义系统标准输入/输出函数
absacc.h	定义常用访问绝对地址的宏

2. 条件编译

预处理程序提供了条件编译功能，可按不同的条件编译不同的程序部分，产生不同的目标代码文件，有利于程序的移植和调试。条件编译有 3 种形式。

1）
```
#ifdef 标识符
    程序段 1
#else
    程序段 2
#endif
```

功能：如果标识符已被#define 命令定义过，则对程序段 1 进行编译；反之，则对程序段 2 进行编译。如果没有程序段 2(它为空)，本格式中的#else 可以没有，即可以写为：

```
#ifdef 标识符
程序段
#endif
```

2）
```
#ifndef 标识符
程序段 1
#else
    程序段 2
#endif
```

或

```
#ifndef 标识符
程序段
#endif
```

功能：如果标识符未被#define 命令定义过，则对程序段 1 进行编译；反之，则对程序段 2 进行编译。与第一种形式的功能正相反。

3）
```
#if 常量表达式
    程序段 1
#else
    程序段 2
#endif
```

功能：如常量表达式的值为真(非 0)，则对程序段 1 进行编译；反之，则对程序段 2 进行编译，因此可以使程序在不同条件下完成不同的功能。

3.3.2 C语句

C语句可分为5类：表达式语句、函数调用语句、控制语句、复合语句和空语句。

1. 表达式语句

表达式语句由表达式加上分号(;)组成。其一般形式为：

```
表达式;
```

执行表达式语句就是计算表达式的值。例如，“x=y+z;”是赋值语句。“y+z;”是加法运算语句，但计算结果不能保留，无实际意义。“i++;”是自增1语句，执行该语句后i值增1。

2. 函数调用语句

函数调用语句由函数名和实际参数加上分号(;)组成。

其一般形式为：

```
函数名(实际参数表);.
```

执行函数语句就是调用函数体并将实际参数赋予函数定义中的形式参数，然后执行被调用函数体中的语句，求取函数值(将在3.4节详细介绍)。例如，“printf("C Program");”调用库函数，输出字符串。

3. 控制语句

控制语句用于控制程序的流程，以实现程序的各种结构方式。

它们由特定的语句定义符组成，C语言有9种控制语句，可分成以下3类：

- 条件判断语句：if语句、switch语句。
- 循环执行语句：do while语句、while语句、for语句。
- 转向语句：break语句、goto语句、continue语句、return语句。

4. 复合语句

把多个语句用花括号{}括起来组成的一个语句称复合语句。在程序中应把复合语句看成是单条语句，而不是多条语句，例如：

```
{    x=y+z;
     a=b+c;
}
```

复合语句内的各条语句都必须以分号(;)结尾，在花括号(})外不能加分号。

5. 空语句

只有分号(;)组成的语句称为空语句，该语句不执行任何操作，在程序中空语句可用来作空循环体。

例如：

```
while(P1_1!=0);
```

本语句的功能是，若P1.1口不为0，则在此处等待。这里的循环体为空语句。

3.3.3 分支结构

源程序可以分为3种基本结构：顺序结构、分支结构和循环结构。C语言提供了多种

语句来实现这些程序结构。

程序的分支都是通过关系运算符来判断程序的走向，所谓分支即在何种条件下程序该如何执行的问题。

1. if 语句

if 语句是分支程序常用的语句，其基本格式分为两种。

(1) if 形式：

```
If(条件)
        语句 1;
```

if-else 形式：

```
if(条件)
    语句 1;
else
    语句 2;
    …
```

(2) if-else-if 形式：

```
if(表达式 1)
    语句 1;
else if(表达式 2)
    语句 2;
    …
else if(表达式 m)
    语句 m;
else
    语句 m + 1
```

例 3.13　编写程序，判断 a 与 b 的大小并将最大值赋予 max，要求采用条件语句。

解：

```
void main()
{
    int a, b,max;
    max = a;
    if(max<b)max = b;
    printf("max = %d", &max);
}
```

执行结果为若 if 条件成立，执行赋值语句 max＝b；反之，则 max 仍为 a，处理完成后输出。

2. switch 语句

switch 语句是一种用于多分支选择的语句，其一般形式为：

```
switch(表达式)
{
case 常量表达式 1: 语句 1;
case 常量表达式 2: 语句 2;
…
```

```
case 常量表达式 n: 语句 n;
default : 语句 n + 1;
}
```

其语义是：计算表达式的值，并逐个与其后的常量表达式值相比较，当表达式的值与某个常量表达式的值相等时，即执行其后的语句，如表达式的值与所有 case 后的常量表达式均不相同时，则执行 default 后的语句。

例 3.14 采用 switch-case 语句，根据数值 a 决定对 a 进行何种运算。若 a 为 0，则 a＝a＋1。若 a 为 1，则 a＝a－1。若 a 为 2，则 a＝a＊2。若 a 不为 0、1、2 中的任何数，则将 a 赋值为 0。

解：此题可以使用前面的 if 语句形式编程，由于需要对 a 进行多次判断，因此采用 if 语句并不是最佳选择，依照题目要求使用 switch-case 语句可以更方便地完成题目要求，且程序更为简洁。

```
void main()
{
    int a;
    switch (a)
    {
        case 0: a += 1;
        case 1: a -= 1
        case 2: a * = a;
        default : a = 0;
    }
}
```

在使用 switch 语句时还应注意以下 4 点。

- 在 case 后的各常量表达式的值不能相同，否则会出现错误。
- 在 case 后允许有多个语句，可以不用{}括起来。
- 各 case 和 default 子句的先后顺序可以变动，而不会影响程序执行结果。
- default 子句可以省略不用。

3. return 语句

该语句的作用是从子程序中返回，格式为：

```
return(语句)
```

其中括号里的可省略，表示不需要返回值。

例 3.15 分析程序中函数 function1 与 function2 中返回值的不同之处。

```
void function1(unsigned char a)
    {
        unsigned char x;
        x = a + 1;
        return;
    }
void function2(unsigned char a)
    {
        unsigned char x;
```

```
        x = a + 1;
        return x;
    }
void main(void)
    {
        unsigned char m = 10;
        function1(m);
        p = function2(m);
    }
```

分析：function1 函数并没有返回值，void 表示当函数返回时不返回任何数据类型的值。而 function2 函数返回值为 x，在函数中处理完成后将返回值传输给 p。

3.3.4　循环结构

在给定条件成立时，反复执行某程序段，直到条件不成立为止的语句称为循环语句。给定的条件称为循环条件，反复执行的程序段称为循环体。

1. while 循环语句

while 语句的形式为：

```
while(表达式)语句;
```

其中，表达式为循环条件，语句为循环体。

while 语句的作用是：计算表达式的值，当值为真(非 0)时，执行循环体语句。

例如：

```
while(P1_1 ! = 0) ;
    while(P1_1 ! = 1) ;
```

上面两个 while 语句都是空语句，作用是等待 P1_1 脚上升沿跳变。

2. do-while 循环语句

do-while 语句的形式为：

```
do
    语句;
while(表达式);
```

其中，语句为循环体，表达式为循环条件。

do-while 语句的作用是先执行循环体语句一次，再判别表达式的值，若为真(非 0)，则继续循环；反之，则终止循环。

do-while 语句和 while 语句的区别在于 do-while 是先执行后判断，因此 do-while 至少要执行一次循环体。while 是先判断后执行，如果条件不满足，则一次循环体语句也不执行。while 语句和 do-while 语句一般都可以相互改写。

例 3.16　试编写程序产生一个只有 8 个周期的方波。

解：每次循环体执行的时候就对 P1_1 口求反，从而实现方波的输出。

方波程序如下：

```
    unsigned counter;
```

```
    …
    do
    {
        P1_1 = !P1_1;
        delay(1000); /* 延时 */
        counter++;
    }
while(counter<= 8);
```

注：循环中需要有语句来控制循环变量，如示例中的 counter＋＋语句，否则将陷入死循环。

3. for 循环语句

for 语句是 C 语言广泛使用的一种循环语句。其形式为：

```
for(表达式 1;表达式 2;表达 3)
    语句;
```

(1) 表达式 1：给循环变量赋初值，一般是赋值表达式。

(2) 表达式 2：循环条件，一般为关系表达式或逻辑表达式。

(3) 表达式 3：用来修改循环变量的值，一般是赋值语句。

3 个表达式都是任选项，都可以省略，但分号间隔符不能少。

for 语句的作用是：

(1) 计算表达式 1 的值。

(2) 计算表达式 2 的值，若值为真(非 0)则执行循环体一次，否则跳出循环。

(3) 计算表达式 3 的值，转回第(2)步重复执行。

例 3.17 试编写程序，计算 s＝1＋2＋3＋…＋99＋100。

解：

```
int n, s = 0;
for(n = 1; n<= 100; n++)
s = s + n;
```

若省去循环中的表达式 1，则等价为：

```
int n = 0, s = 0;
for( ; n<= 100; n++)
s = s + n;
```

注意，省去表达式 1 后，需要在循环语句前对数据进行初始化。

若省去循环中的表达式 3，还可以写为：

```
int n = 0,s = 0;
for( ; n<= 100;)
{
    s = s + n;
    n++;
}
```

注意，在套用编写时要将层次关系分清楚，不同的层次采用不同的缩进量，从而达到区

分内外层，整个程序结构也显得更清晰，便于阅读。

4. goto 循环语句

goto 语句称为无条件转移语句，其格式如下：

```
goto 语句标号;
```

其中，语句标号是按标识符规定书写的符号，放在某一语句行的前面，标号后加冒号(:)。语句标号起标识语句的作用，与 goto 语句配合使用。

在结构化程序设计中一般不建议使用 goto 语句，以免造成程序流程的混乱，导致理解和调试程序带来困难。

3.3.5　break 和 continue 语句

1. break 语句

break 语句只能在 switch 语句或循环语句中使用，作用是跳出 switch 语句或跳出本层循环。由于 break 语句的转移方向明确，不需要语句标号与之配合。

break 语句的形式为：

```
break;
```

2. continue 语句

continue 语句只能用在循环体中，其格式是：

```
continue;
```

其语义是结束本次循环，转入下一次循环条件的判断与执行。应当注意，本语句只结束本层本次的循环，并不跳出循环。

例 3.18　编写程序，给定 a 值为 0，将 a 累加到 50，要求使用 break 与 continue 语句。

解：

```
void main()
{
    int a;
    for (a = 0; 1; a++)
    {   if (a == 50) break;
        else continue;
    }
}
```

在上面例子中，for 语句中的循环中止条件用 break 语句来完成，当 a＝50 时，则跳出循环。continue 语句的作用是结束当前的本次循环，继续下次循环。

C 语言的语句结构如表 3-2 所示。

表 3-2　C 语言语句结构综合表

名　称	一 般 形 式	名　称	一 般 形 式
简单语句	表达式语句表达式；	开关语句	switch(表达式) { case 常量表达式：语句 … default:语句；}
空语句	；		
复合语句	{语句 }		

续表

名　称	一 般 形 式	名　称	一 般 形 式
条件语句	(1) if(表达式)语句； (2) if(表达式)语句 1； else　语句 2； (3) if(表达式 1)语句 1； else if(表达式 2)语句 2 … else 语句 n；	循环语句	while 语句：while(表达式)语句； for 语句：for(表达式 1；表达式 2；表达式 3)语句； break 语句：break； goto 语句：goto； continue 语句：continue； return 语句：return(表达式)；

3.2.6　结构体与联合体

1. 结构体

结构体是数据项的集合，每个结构体由其类型和名称制定。在 C 语言中，使用结构体 struct 来存放一组不同类型的数据。结构体的定义形式为：

```
struct 结构体名{
结构体所包含的变量或数组;
};
```

结构体是一种集合，它里面包含了很多变量，它们的类型可以相同，也可以不同，每个变量都称为结构体的成员。下面给出一个例子：

```
struct student{
char *name;
int age;
int num;
float score;
};
```

其中 student 是结构体名，它包含了 4 个成员，分别是 name、age、num、score。结构体成员定义的方式与变量和数组的定义方式相同，但是结构体成员不能初始化。

结构体也是一种数据类型，它由程序员自己来定义，可以包含多个其他类型的数据。

既然结构体是一种数据类型，那么就可以用它来定义变量。如：

```
struct  student s1,s2,s3;
```

定义了 3 个变量 s1、s2 和 s3，它们都是 student 类型，都由 name、age、num、score 4 个成员组成。

还可以在定义结构体的同时定义结构体变量。如下面的例子所示：

```
struct student{
char *name;
int  age;
int  num;
float  score;
}s1,s2,s3;
```

2. 联合体

在C语言中，有一种与结构体非常类似的构造类型（或者说复杂类型），即共用体，又叫作联合体（union）。它的定义格式为：

```
union 联合体名{
成员列表;
};
```

联合体也是一种自定义类型，可以创建变量，例如：

```
union data{
int l;
char s;
float m;
double n;
};
union data a, b, c;
```

这是先定义联合体，再创建变量，也可以在定义联合体的同时创建变量。如：

```
union  data{
int l;
char s;
float m;
double n;
} a, b, c;
```

联合体与结构体的区别在于：结构体的各成员会占用不同内存，互相之间没有影响；而联合体的各成员共用一段内存，修改一个成员会影响其他所有成员。

结构体占用内存大于或等于各成员所占用的内存之和，而联合体占用内存等于最长成员所占用的内存。联合体使用的是内存覆盖技术，同一时刻只能保存一个成员的值，所以对新的成员赋值时，就会把原来成员的值覆盖掉。

3.4　函数

C源程序是由函数组成的。虽然在前面各章的程序中都只有一个主函数main()，但实际程序往往由多个函数组成。函数是C源程序的基本模块，通过对函数模块的调用实现特定的功能。可以说C程序的全部工作都是由各式各样的函数完成的，因此也把C语言称为函数式语言。由于采用了函数模块式的结构，C语言易于实现结构化程序设计。使程序的层次结构清晰，便于程序的编写、阅读、调试。

3.4.1　函数的分类

1. 从函数定义的角度分

1）库函数

由C系统提供，用户无须定义，也不必在程序中作类型说明，只需在程序前包含有该函

数原型的头文件即可在程序中直接调用。在前面各章的例题中反复用到的 printf、scanf、getchar、putchar、gets、puts、strcat 等函数均属此类。

2）用户定义函数

由用户按需要写的函数。对于用户自定义函数，不仅要在程序中定义函数本身，而且在主调函数模块中还必须对该被调函数进行类型说明，然后才能使用。

2. 从函数和过程功能角度分

1）有返回值函数

此类函数被调用执行完后将向调用者返回一个执行结果，称为函数返回值，如数学函数即属于此类函数。由用户定义的带返回函数值的函数，必须在函数定义和函数说明中明确返回值的类型。

例如：

```
unsigned char fun1(unsigned char a)
{
    return (a);
}
```

2）无返回值函数

此类函数用于完成某项特定的处理任务，执行完成后不向调用者返回函数值。这类函数类似于其他语言的过程。由于函数无须返回值，所以用户在定义此类函数时可指定它的返回为“空类型”，空类型的说明符为 void。

```
void fun1(unsigned char a)
{
    return ;
}
```

3. 从主调函数和被调函数之间数据传送的角度分

1）无参函数

函数定义、函数说明及函数调用中均不带参数，主调函数和被调函数之间不进行参数传送。此类函数通常用来完成一组指定的功能，可以返回或不返回函数值。

2）有参函数

有参函数也称为带参函数。在函数定义及函数说明时都有参数，该参数称为形式参数（简称为形参）。在函数调用时也必须给出参数，该参数称为实际参数（简称为实参）。进行函数调用时，主调函数将把实际参数的值传送给形式参数，供被调函数使用。

3.4.2 函数的定义

1. 无参函数的一般形式

无参函数的一般形式如下：

```
类型说明符 函数名()
{
类型说明
语句
}
```

其中，类型说明符和函数名称为函数头，它说明函数的类型，函数的类型实际上是函数返回值的类型，该类型说明符与之前介绍的各种说明符相同。函数名是由用户定义的标识符，函数名后有一个内容为空的括号，其中无参数，但括号是必需的。{}中的内容称为函数体。在函数体中也有类型说明，这是对函数体内部所用到的变量的类型说明。在很多情况下都不要求无参函数有返回值，此时函数类型符可以写为 void。

例 3.19　编写固定延时子程序。

解：

```
void delay_fix(void)
{
    unsigned int a;
    for(a = 0; a<2000; a++)
    {;}
}
```

该程序延时时长固定，根据每条指令的运行时间可以计算出总延时时长。

2. *有参函数的一般形式*

有参函数的一般形式如下：

```
类型说明符 函数名(形式参数表)
形式参数类型说明
{
类型说明
语句
}
```

有参函数比无参函数多了两个内容：其一是形式参数表，其二是形式参数类型说明。在形式参数表中给出的参数称为形式参数，它们可以是各种类型的变量，各参数之间用逗号间隔。在进行函数调用时，主调函数将赋予这些形式参数实际的值。

例 3.20　编写可控延时子程序：

解：

```
void delay(unsigned int b)
{
    unsigned int a;
    for(a = 0; a<b; a++)
    {;}
}
```

该程序可以根据 b 的值决定延时时长。

3.4.3　函数的调用

在程序中是通过对函数的调用来执行函数体的，其过程与其他语言的子程序调用相似。函数调用的一般形式为：

```
函数名(实际参数表)
```

对无参函数调用时则无实际参数表，实际参数表中的参数可以是常数、变量或其他构造

类型数据及表达式，各实际参数之间用逗号分隔。

例如：

```
int max(int a, int b)
{    If(a>b) return a;
     else return b;
}
```

上面的子程序是一个比较大小的子程序，有一个返回值。

通过其他程序调用它的方法如下。

1. 函数表达式

函数是表达式中的一项，以函数返回值参与表达式的运算，这种方式要求函数是有返回值的。例如，z＝max（x，y）为一个赋值表达式，把 max 的返回值赋予变量 z。

2. 函数语句

函数调用的一般形式加上分号即构成函数语句。

例如：

```
delay (max (a, b));
```

以函数语句的方式调用函数。

3. 函数实际参数

函数作为另一个函数调用的实际参数出现，这种情况是把该函数的返回值作为实际参数进行传送，因此要求该函数必须是有返回值的。

对于上面函数的调用都是通过主调函数将值（实际参数）传给被调函数的变量（形式参数），这两个参数在内存中占据不同的单元，因此这种方法调用函数是不能改变主调函数中实际参数的值的，因为是单向赋值的过程，但是通过数组或指针的调用却能改变。

例 3.21 分析以下程序中形式参数与实际参数的传递。

```
int max (int a, int b)
    {    if (a>b)
            return a;
         else
            return b;
    }
main ()
    {    int c, d;
         max (c, d)
    }
```

分析：将 c 的值赋给临时变量 a，d 的值赋给临时变量 b。a、b、c、d 在内存中没有共同的地址，并且形式参数 a 与 b 的变化并不会影响实际参数 c 与 d 的变化。

3.4.4 函数值

函数的值是指函数被调用之后，执行函数体中的程序段所取得的并返回给主调函数的值。

函数的值只能通过 return 语句返回主调函数。return 语句的一般形式为：

```
return (表达式);
```

该语句的功能是计算表达式的值，并返回给主调函数。在函数中允许有多个 return 语句，但每次调用只能有一个 return 语句被执行，因此只能返回一个函数值。

函数值的类型和函数定义中函数的类型应保持一致。如果两者不一致，则以函数类型为准，自动进行类型转换。若函数值为整型，则在函数定义时可以省去类型说明。

不返回函数值的函数，可以明确定义为“空类型”，类型说明符为 void。

3.4.5　函数的递归调用

一个函数在它的函数体内调用自身称为递归调用，这种函数称为递归函数。C 语言允许函数的递归调用。在递归调用中，主调函数又是被调函数。执行递归函数将反复调用其自身。每调用一次就进入新的一层。例如，有函数 f 如下：

```
int f (int x)
    {
    int y;
    z = f(y);
    return z;
    }
```

这个函数是一个递归函数，但是运行该函数将无休止地调用其自身，这当然是不正确的。为了防止递归调用无终止地进行，必须在函数内有终止递归调用的手段，常用的办法是加条件判断，满足某种条件后就不再作递归调用，然后逐层返回。

在单片机 C51 语言中，称可递归调用的程序为可重入程序(reentrance)，因为递归调用要在内存中开辟一段存储区，由于 51 系列 RAM 内存较小，所以递归调用是受到限制的。

3.5　单片机软件延时原理

3.5.1　单片机时序与指令周期

CPU 在执行指令时所控制信号的时间顺序称为时序。时序通过定时单位来描述，MCS-51 系列单片机的时序单位有 4 个，分别是时钟周期(节拍)、状态、机器周期和指令周期。

1. 时钟周期

时钟周期又称为振荡周期，用节拍 P 表示，即为单片机提供时钟信号的振荡源(oscillator，一般简写为 osc)的周期，是时序中的最小单位。

2. 状态

时钟周期经过二分频后即得到整个单片机工作系统的状态，用 S 表示。一个状态有两个节拍，前半周期对应的节拍定义为 P1，后半周期对应的节拍定义为 P2。

3. 机器周期

完成一个基本操作所需的时间称为机器周期。MCS-51 中规定一个机器周期包含 12 个时钟周期，即有 6 个状态，分别表示为 S1～S6。例如，若晶振为 6MHz，则机器周期为 2μs；若晶振为 12MHz，则机器周期为 1μs。

4. 指令周期

执行一条指令所需要的时间称为指令周期。一个指令周期通常含有 1～4 个机器周期。

运算速度是由指令所包含的机器周期数决定的，机器周期数越少，其执行速度越快。

以机器周期为单位，指令可分为单周期指令、双周期指令和四周期指令 3 种。

能完成一定功能的指令的集合称为程序，执行程序的过程就是执行指令，单片机执行任何一条指令时都可以分为取指和执行。取指的功能是 CPU 从程序存储器中取出指令操作码，送到指令寄存器，再经指令译码器译码，执行的功能是根据指令译码产生一系列控制信号，完成本指令规定的操作。

单周期和双周期指令的取指时序图如图 3-9 所示。

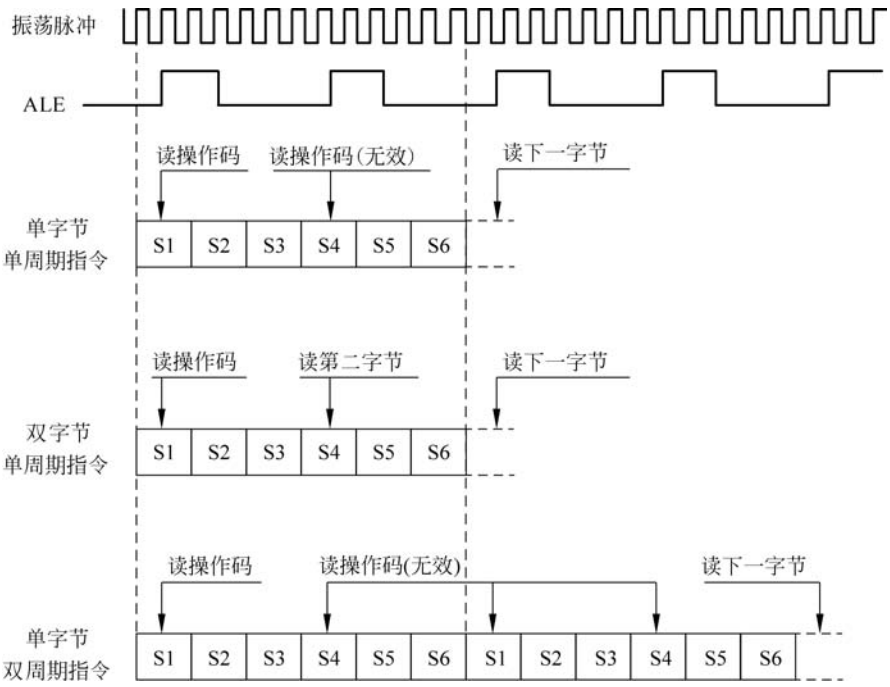

图 3-9　单周期和双周期指令的取指时序图

ALE 信号是用于锁存低 8 位地址的选通信号，每出现一次 ALE 信号，单片机进行一次读指令操作。当指令为多字节或多周期指令时，只有第一个 ALE 信号进行读指令操作，其余的 ALE 信号为无效操作，或读操作数操作。

3.5.2　软件延时函数的编写

在单片机的控制应用中，控制过程中常有延时的需要。延时功能除了可以使用单片机的定时/计数器实现之外，还可以通过使用软件程序完成。

软件延时程序是典型的循环程序，它是通过执行一个具有固定延时时间的循环体来实现的。在单片机中，机器周期相当于一个固定延时时间，它只由单片机的晶振频率影响。下面先通过一段汇编延时程序加以说明。

```
DELAY30MS: MOV  R7, #100;
DELAY:MOV  R6, #150;
    DJNZ  R6, $
    DJNZ  R7, DELAY
    RET
```

上面是一个 30ms 的延时程序，单片机使用的晶振频率为 12MHz，查单片机指令表可知，“DJNZ Rn，rel”指令执行时间为 2 个机器指令时间，即执行一次 DJNZ 指令需要 12/12M×2＝2μs，整段语句的执行时间为 2μs×150×100＝30ms。

例 3.22　C 语言的软件延时程序。

```
void Delay_us(unsigned int times)
{
unsigned int i;
for (i = 0; i<times; i++)
     {     _nop_(); // 调用 nop,延时 1μs
           _nop_();
           _nop_();
           _nop_();
                }
}
```

分析：上面程序中的_nop_()是空指令，延时一个指令周期，即延时 1μs。该程序根据给定值 times 可以决定延时时长，每次循环延时 4μs。

3.5.3　红绿灯应用实验

1. 实验内容

如图 3-10 所示，将数码管第 7 位、第 3 位和第 1 位分别视为红灯、黄灯和绿灯，显示红灯 5s 后，显示黄灯 1s，再显示绿灯 3s，依次循环显示，编写对应程序，程序要求延时部分使用子程序。其中，灯亮视作对应位显示 0。

2. 程序分析及设计

1）程序结构设计

依照程序设计要求，可画出程序结构示意图，如图 3-11 所示。

2）位置控制

方法一：直接置位。

语句：　CLR 端口　　//清 0

　　SETB 端口　　//置 1

方法二：传送数据。

语句：MOV P2, #data

3）数字控制

语句：

立即寻址 MOV P1, #data

直接寻址 MOV 地址, #data

　　MOV P1, 地址

4）延时程序

例 3.23　延时 5s 程序设计。

解：

```
5000000μs/2μs = 2500000 = 250 × 100 × 100 次
DELAY5S:     MOV R7, #250
```

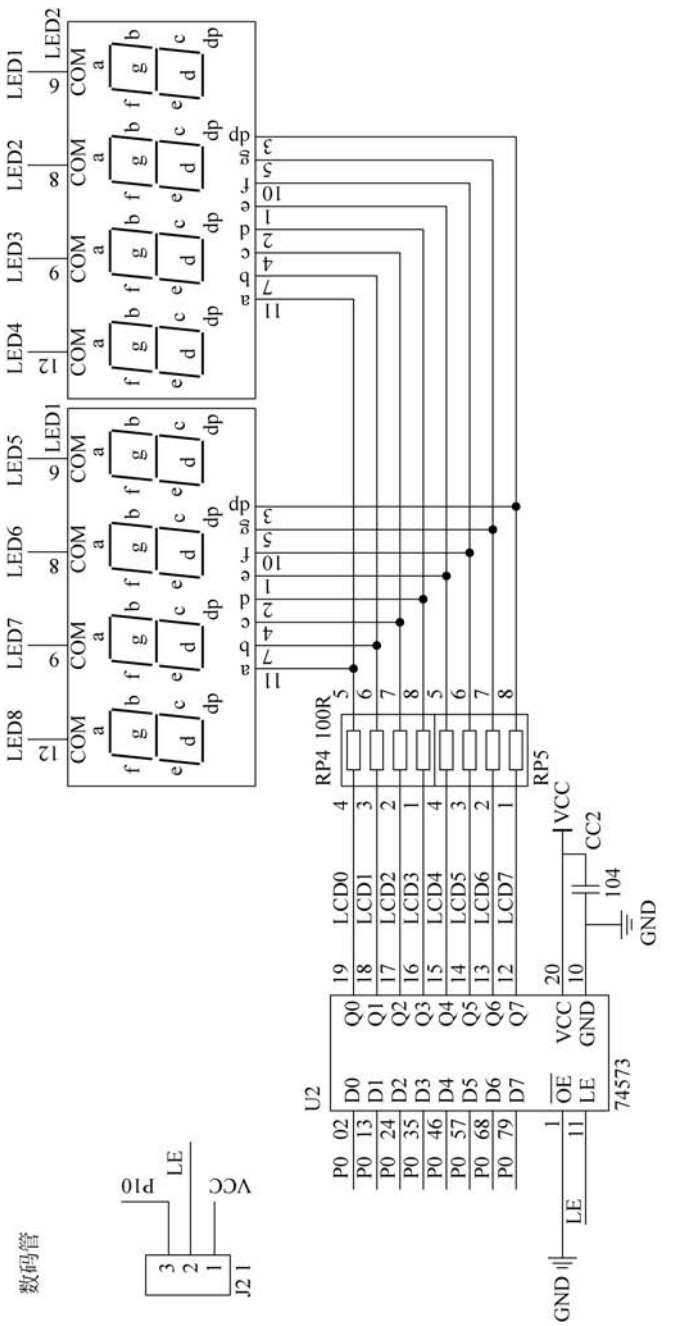

图 3-10　红绿灯应用实验

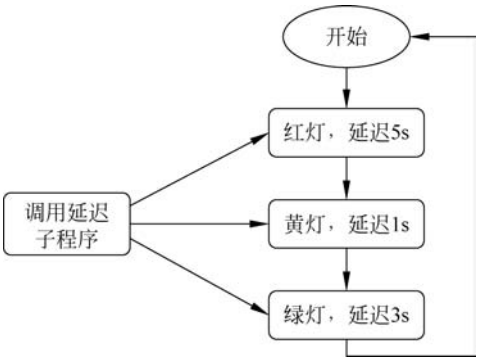

图 3-11　红绿灯实验程序结构

```
DELAY1:     MOV R6, #100
DELAY2:     MOV R5, #100
            DJNZ R6, $
            DJNZ R6, DELAY2
            DJNZ R7, DELAY1
            RET
```

使用时采用语句 LCALL 调用即可实现延时 5s。

其他时长延时可以依照该示例相应更改程序。

5）程序清单

C 语言程序清单	汇编语言程序清单

#include<reg52.h>
#define GPIO_DIG P0
sbit LSA = P2 ^ 2;
sbit LSB = P2 ^ 3;
sbit LSC = P2 ^ 4;
unsigned char code DIG_CODE[10] = {0x3f, 0x06,
0x5b, 0x4f, 0x66, 0x6d,
0x7d, 0x07, 0x7f, 0x6f };
void Delay10ms(unsigned int c)
{
unsigned char a, b;
for (; c>0; c--)
for (b = 38; b>0; b--)
for (a = 130; a>0; a--);
}
void main()
{
GPIO_DIG = DIG_CODE[0];
while(1)
{
LSA = 0;
LSB = 1;
LSC = 1;
Delay10ms(500);　　//红灯 5s;
LSA = 0;
	ORG　00H
	LJMP　INIT
	ORG　30H
	LSA　EQU　P2.2
	LSB　EQU　P2.3
	LSC　EQU　P2.4
	DAT0　EQU　30H
	GPIO_DIG　EQU　P0
	INIT:
	MOV　DAT0,　#03FH
	MOV　R6,　#00H
	MOV　R7,　#00H
	MOV　R8,　#00H
	MAIN :
	CLR　LSA
	SETB　LSB
	SETB　LSC
	MOV　GPIO_DIG,　DAT0
	ACALL　DELAY5S
	MOV　GPIO_DIG,　#00H
	CLR　LSA
	SETB　LSB
	CLR　LSC
	MOV　GPIO_DIG,　DAT0
	ACALL　DELAY1S

C语言程序清单	汇编语言程序清单

LSB = 1;
LSC = 0;
Delay10ms(100); //黄灯亮 1s;
LSA = 0;
LSB = 0;
LSC = 0;
Delay10ms(300); //绿灯亮 3s;
}
}
	MOV GPIO_DIG, #00H
	CLR LSA
	CLR LSB
	CLR LSC
	MOV GPIO_DIG, DAT0
	ACALL DELAY3S
	MOV GPIO_DIG, #00H
	LJMP MAIN
	DELAY5S: MOV R8 , #250
	DEL1: MOV R7, #100
	DEL2: MOV R6,#100
	DJNZ R6, $
	DJNZ R7, DEL2
	DJNZ R8, DEL1
	RET
	DELAY3S: MOV R8, #150
	DEL1: MOV R7, #100
	DEL2: MOV R6, #100
	DJNZ R6, $
	DJNZ R7, DEL2
	DJNZ R8, DEL1
	RET
	DELAY1S: MOV R8, #100
	DEL1: MOV R7, #100
	DEL2: MOV R6, #100
	DJNZ R6, $
	DJNZ R7, DEL2
	DJNZ R8, DEL1
	RET
	END

3. 实验思考

(1) 什么是静态显示和动态显示?

(2) 为什么在动态显示中,每显示一个数要延时若干时间?试修改延时程序的立即数观察运行效果,并做记录。分析延时长一些好还是延时短一些好,说明延时量对人眼的视觉影响。

3.6 习题

1. 布尔操作指令中采用了什么寻址方式?

2. 比较两个无符号数 a、b 的大小,根据 a>b、a=b、a<b 这 3 种不同情况分别转向标号为 DAYU、DENGYU、XIAOYU 的处理程序段,请编写相应的程序。

3. 编写一个程序,将外部存储器 1000H~1100H 中的所有数从大到小排列。

4. 编写一个脉冲输出程序,从 P1.0 输出占空比为 0.5 的方波。