# 第5章 过　　程

在计算机科学中，汇编语言是一种直接操作计算机硬件的语言，它为我们提供了对计算机底层操作的精细控制。在本章中，我们将深入探讨汇编语言中的关键概念和技术，主要集中在程序链接与链接库、堆栈机制以及过程的定义和使用上。在本章的学习过程中，我们不仅将学习研究汇编过程的技术细节，还将思考其中蕴含的价值观念和社会责任。

首先，我们将研究程序链接与链接库的基本原理。了解如何与外部库进行链接是编写大型程序时必不可少的一环。我们将探讨常见的链接库，以及与外部库链接的概念和技术，为构建模块化、可维护的程序奠定基础。

其次，我们将深入研究堆栈机制。堆栈是计算机内存中一个重要的数据结构，它在程序执行过程中扮演着关键角色。我们将学习运行时栈的概念和基本原理，以及如何使用PUSH和POP指令来操作堆栈，实现数据的存储和检索。

最后，我们将探讨过程的定义和使用。过程是程序中一组相关操作的集合，它们可以被封装和重复利用，增强代码的可读性和可维护性。我们将学习过程声明伪指令PROC、过程调用与返回指令CALL和ERET的使用方法，以及如何绘制流程图来展示程序的执行流程。同时，我们还将讨论寄存器的恢复和保存，以确保程序在调用过程后能够正确地返回原始状态。

通过本章的学习，读者将深入了解汇编语言中程序链接与链接库、堆栈机制以及过程的重要概念和技术，为读者在编写高效、可靠的汇编程序时提供必要的知识和技能。通过学习过程的定义和使用，读者将不仅能够培养良好的编程习惯和技能，还能够认识到个体在社会中的责任和使命，不断提升自我，为社会发展做出更大的贡献。

## 5.1 程序链接与链接库

### 5.1.1 链接库

链接库是一组预先编译的代码集合，通常包含了一系列可复用的函数或程序，目的是为开发者提供完成特定任务的现成工具。这些库可以是动态的(在程序运行时加载)或静态的(在程序编译时直接整合到程序中)，它们提供了一种高效的方式来扩展应用程序的功能，无须从头开始编写所有代码。

在编写汇编语言程序时，链接外部库是一个关键的步骤，它能大幅提高编程效率并扩展程序的功能。通过使用外部库，程序员可以避免重复编写常用代码，并利用专业优化的库函数来提高程序的性能和可靠性。此外，外部库的使用还促进了代码的模块化和抽象。它允许开发者将复杂功能封装在简单的接口后面，从而使汇编程序更加简洁、易于维护，同时也便于跨平台部署和更新。

链接库分为静态链接库和动态链接库两大类。静态库（如 .lib 文件）在程序编译时被复制到最终的可执行文件中，这种方式简化了部署过程，因为最终的程序不依赖于外部的库文件。而动态库（如 .dll 或 .so 文件）则在程序运行时被加载，优势在于可以减小程序的初始大小，共享内存，以及更新库而不需要重新编译整个程序。

在接下来的学习过程中，我们将频繁使用两个特别的链接库：Irvine32.lib 和 Irvine16.lib。这些库为汇编语言的学习和实践提供了极大的便利和支持。这是两个专为教学目的而设计的库，广泛用于学习和教授 x86 汇编语言编程，为初学者提供了一系列简化的汇编语言编程任务的函数。

接下来，我们将通过一个简单的显示字符串函数 WriteString 讲解链接库的使用方法。

首先，在代码中包含 Irvine32.inc 文件：

```
include Irvine32.inc
```

然后，在.data 部分定义想要显示的字符串：

```
.data
message db 'Hello, world!', 0
```

最后，在.code 中调用字符串：

```
mov edx, OFFSET message
call WriteString
```

大家可能发现，我们之前讲解的链接库文件是 Irvine32.lib，但是在代码里导入的是 Irvine32.inc 文件，这是为什么呢？

Irvine32.inc 是一个包含文件，它提供了 Irvine32.lib 库中所有函数的声明和宏定义。在编写汇编程序时，使用 include Irvine32.inc 语句将这个文件的内容包含到源代码中。因此，在编译时，汇编器处理源代码，包括处理 include Irvine32.inc 引入的声明和宏。编译器根据这些信息生成中间的对象文件，这个文件中含有对库中函数的未解决引用。而 Irvine32.lib 是一个库文件，其中包含了实际的可执行代码。程序被编译成对象文件后，在链接阶段，链接器将引用这个库文件来解析程序中那些指向库函数的外部引用。因此在链接器接管后，将查找这些外部引用在 Irvine32.lib 中的对应实现，将它们与对象文件合并，生成最终的可执行文件。

在 MASM32 中，链接器的使用是通过 link 命令实现的：

```
link /subsystem:console example.obj Irvine32.lib kernel32.lib user32.lib /out:
example.exe
```

此时，在控制台运行程序，就会输出 Hello, world!了。

### 5.1.2 常见链接库

在我们的学习中，我们最常用的链接库是 Irvine32.lib 和 Irvine16.lib，接下来我们将讲解这些库中的常用过程。过程是汇编语言中的术语，其功能类似于高级语言的函数，过程是一种方式，通过它可以将代码组织成单独的模块或单元，每个单元执行一个特定的任务。这种组织方式不仅使代码更易于理解和维护，还可以提高代码的重用性。

Irvine32.lib 是为 32 位处理器设计的库，特别是为运行在 32 位 Windows 操作系统上的应用程序。这个库利用 32 位处理器的功能，提供了一系列用于教学的程序接口和功能，Irvine16.lib 则是为 16 位处理器设计的库，但是它依旧使用了 32 位寄存器。接下来我们将介绍它们的常见过程。

#### 1. CloseFile

此过程用于关闭一个打开的文件。文件是以文件句柄标识的，文件句柄是一个用于标识打开的文件的抽象概念。它通常由操作系统提供，用于代表一个特定的文件资源。文件句柄可以被视作一个指向文件的引用或指针，程序通过它可以访问文件的内容、属性或进行其他操作。

此过程在 eax 寄存器中读取文件句柄，如果成功执行，那么 eax 寄存器将返回非零值。同时，此过程仅存在在 Irvine32.lib 中，而不在 Irvine16.lib 中。

```
mov eax, fileHeadle
call CloseFile
```

#### 2. Clrscr

此过程用于清除控制台窗口的内容，通常和 WaitMsg 过程同时使用。WaitMsg 用于暂停程序，防止清除内容前用户错过有用信息。

```
call WaitMsg
call Clrscr
```

#### 3. ReadInt

此过程用于从控制台读取一个整数值。它没有参数，而是直接从标准输入读取数字，并将其作为整数返回在 EAX 寄存器中。这个过程简化了从用户那里获取整数输入的过程，使汇编程序能够轻松集成用户交互功能。

```
call ReadInt
mov ebx, eax
```

#### 4. RandomRange

此过程用于生成一个指定范围内的随机数。调用前，需要在 EAX 寄存器中设置范围的上限。返回的随机数将存储在 EAX 寄存器中。此过程是实现基于随机数逻辑的理想选择。

```
mov eax, 100
call RandomRange
```

```
mov ebx, eax
```

### 5. OpenFile

此过程用于打开一个文件，其文件名地址应放在 EDX 寄存器中。如果操作成功，文件句柄将返回在 EAX 寄存器中，否则返回值为－1。这允许程序动态地访问和修改文件，和 CloseFile 一样，OpenFile 也是进行文件操作时的基础。

```
.data
filename db "data.txt", 0
.code
mov edx, OFFSET filename
call OpenFile
mov fileHandle, eax;
```

### 6. SetTextColor

此过程用于设置控制台窗口中文本的颜色。在调用此过程之前，应将想要设置的颜色代码放入 EAX 寄存器中。这使得程序可以根据需要动态地改变文本颜色，以增强用户界面的视觉效果。

```
mov eax, 2
call SetTextColor
```

### 7. ReadString

此过程用于从标准输入读取一个字符串。在调用之前，需要在 EDX 寄存器中放置字符串的存储地址，并在 ECX 寄存器中设置最大字符数。这个过程简化了从用户输入接收文本的流程，特别是处理用户的交互输入。

```
.data
userInput db 100 dup(0)
.code
lea edx, userInput
mov ecx, 100
call ReadString
```

### 8. Delay

此过程用于在程序中引入延迟，延迟的时间通过 EAX 寄存器以毫秒为单位提供。这个过程在需要暂停程序执行一段指定时间时非常有用，例如在用户交互或动画显示中。

```
mov eax, 5000
call Delay
```

### 9. DumpMem

此过程用于显示内存中特定区域的内容。在调用此过程之前，程序应在 ESI 寄存器中指定内存区域的起始地址，并在 ECX 寄存器中指定要显示的字节数。通过此过程，开发者可以查看和验证内存中的数据状态，这对于调试程序非常有用。

```
.data
startAddress dd offset someData   ; someData 是之前定义的数据
numBytes dd 100
.code
mov esi, startAddress
mov ecx, numBytes
call DumpMem
```

### 10. DumpRegs

此过程用于显示当前所有寄存器的状态，包括通用寄存器、段寄存器、指令指针和标志寄存器。此过程不需要任何参数，调用后会在控制台输出寄存器的当前值。这个过程主要用于调试，帮助理解程序的运行状态和寄存器之间的交互。

```
call DumpRegs
```

### 11. WriteBin

此过程用于将一个寄存器中的值以二进制格式输出到控制台。在调用此过程之前，程序应将要显示的值放在 EAX 寄存器中。这种方式特别适用于需要查看或展示数据的二进制表示的教学或调试场景。

```
mov eax, 0x1234
call WriteBin
```

### 12. WriteChar

此过程用于在控制台输出一个字符。字符应放在 AL 寄存器中。这个过程通常用于输出单个字符或构建更复杂的基于字符的界面。

```
mov al, 'A'
call WriteChar
```

### 13. WriteDec

此过程用于将一个整数以十进制格式输出到控制台。整数值应该放在 EAX 寄存器中。这是在控制台显示整数值时的常用方法，尤其是在需要以用户可读的格式展示数字时。

```
mov eax, 123
call WriteDec
```

### 14. WriteHex

此过程用于将一个整数以十六进制格式输出到控制台。整数值应放在 EAX 寄存器中。使用十六进制格式显示数据可以更方便地理解和分析程序中的地址和其他以位为单位的数据。

```
mov eax, 0x1A3F
call WriteHex
```

### 15. WriteInt

此过程用于在控制台窗口输出一个整数。整数值应先放入 EAX 寄存器中。调用此过

程后，整数将以十进制格式打印到控制台，常用于输出计算结果或状态信息。

```
mov eax, 12345
call WriteInt
```

### 16. WriteString

此过程用于在控制台窗口输出一个字符串。字符串必须以 null 字符结尾，并且其地址应该放在 EDX 寄存器中。调用此过程会将字符串直接打印到控制台窗口，常用于显示消息或程序输出。

```
.data
message db "Hello, world!", 0
.code
mov edx, OFFSET message
call WriteString
```

## 本节习题

(1) 什么是程序链接？它的作用是什么？

(2) 请简要说明静态链接和动态链接的区别，并举例说明各自的优点和缺点。

(3) 外部库链接的概念是什么？它如何帮助程序员提高开发效率？

(4) 简述链接库是如何在程序编译和执行过程中被使用的。

(5) 简述链接器的作用和功能。它是如何将程序与外部库链接起来的？

(6) 列举几种常见的链接库，例如标准 C 库(libc)、数学库(math)、图形库(graphics)等，并说明它们的用途。

(7) 选择一个常见的外部库，例如标准 C 库，简述该库提供了哪些常用的函数和功能。

# 5.2 堆栈机制

堆栈机制在汇编语言编程中是一个核心概念，它对于函数调用、局部变量存储、参数传递以及程序执行状态的保存和恢复都至关重要。堆栈是一种特殊的数据结构，是一种后进先出的结构。在汇编语言中，堆栈主要是指运行时栈，其操作主要通过栈指针寄存器 ESP 进行管理。

## 5.2.1 运行时栈

运行时栈主要有两个基本操作：压栈操作和出栈操作。

### 1. 压栈操作

压栈操作是将数据元素放入栈顶的过程。在汇编语言中，这通常是通过减少栈指针 ESP 的值来完成的，因为栈在内存中是向下增长的。每次执行压栈操作时，栈指针首先向下移动一定数量的字节，然后将数据复制到栈指针指向的新位置上。这样可以确保后进入栈的元素始终位于栈顶，可以被最先访问和修改。

### 2. 出栈操作

出栈操作是从栈中移除元素的过程，通常是指从栈顶开始。在汇编语言中，进行出栈操作时，首先从当前栈指针指向的位置读取数据，然后栈指针会增加相同的字节数，向上移动回到之前的状态。出栈操作后，原先位于栈顶的数据被移除，允许访问在此之前压入栈的下一个元素。

### 3. 堆栈的使用场景

- **函数调用与返回**：在函数调用时，返回地址和函数的参数通常被压入栈中。当函数执行完成后，通过出栈操作来恢复这些返回地址和参数，保证程序能够返回正确的位置继续执行。
- **局部变量的存储**：函数的局部变量也经常存储在栈上。这使得每次函数调用时都有一个清晰的局部作用域，函数执行完毕后，这些局部变量可以简单地通过调整栈指针来丢弃。
- **程序执行状态的保存和恢复**：在进行诸如上下文切换这类操作时，程序的当前状态可以被保存到栈中，以便之后可以恢复到相同的状态继续执行。

## 5.2.2 PUSH、POP 指令

### 1. PUSH 指令

PUSH 指令的操作流程分为两步：①减小 ESP 的值；②将一个 16 位或 32 位的源操作数复制到堆栈上。如果源操作数为 16 位，ESP 的值将减小 2；如果源操作数为 32 位，ESP 的值将减小 4。PUSH 指令具有以下 3 种格式：

```
PUSH r/m16
PUSH r/m32
PUSH imm32
```

如果程序在调用 Irvine32 中的库过程，应该总是会复制 32 位值，否则在库中使用的 Win32 控制台函数将不能正常运行。如果程序调用的过程来自 Irvine16 的库(实地址模式下)，则能够复制 16 位或 32 位数。

在保护模式下的立即数总是 32 位的。在实地址模式下，如果没有使用.386 及更高的处理器伪指令，立即数则默认为是 16 位的。

### 2. POP 指令

POP 指令的操作流程同样分为两步：①将 ESP 所指的堆栈元素复制到 16 位或 32 位的目的操作数中；②增加 ESP 的值。此时，如果操作数为 16 位，则 ESP 值将加 2；如果操作数为 32 位，则 ESP 值将加 4。POP 指令具有以下 2 种格式：

```
POP r/m16
POP r/m32
```

### 3. PUSHFD 和 POPFD 指令

PUSHFD 指令会在堆栈上压入 32 位的 EFLAGS 寄存器的值，POPFD 指令则会从堆

栈顶部弹出一个 32 位的值并将其送至 EFLAGS 寄存器中，它们的指令格式如下：

```
pushfd
popfd
```

实地址模式下的程序使用 PUSHF 指令会在堆栈上压入 16 位的 FLAGS 寄存器的值；使用 POPF 指令则会从堆栈顶部弹出一个 16 位的值并将其送至 FLAGS 寄存器中。

MOV 指令不能将标志寄存器的值复制到变量或寄存器中。因此，使用 PUSHFD 指令就是保存标志寄存器的最佳方式。在某些情况下，进行保存标志的备份以便后续进行恢复和使用是很有必要的。这通常可以用 PUSHFD 和 POPFD 指令将原本的一块指令包围起来：

```
pushfd                              ; 保存标志
;
; 这里是任意语句……
;
popfd                               ; 恢复标志
```

在使用这种类型的标志压栈和标志出栈指令时，程序的执行路径不能跳过 POPFD 指令。随着时间的推移，再修改程序时将很难记清所有的压栈和出栈指令的位置。因此，编写准确的文档是非常关键的。

如果在一些特殊场景下可以允许少量的错误，那么完成同样功能的方法还有将标志保存在变量中：

```
.data
saveFlage DWORD ?
.code
pushfd                              ; 标志入栈
pop saveFlags                       ; 将保存的标志入栈
popfd                               ; 恢复标志
```

### 4. PUSHAD，PUSHA，POPAD 和 POPA 指令

PUSHAD 指令在堆栈上按照以下顺序将数值压入所有的 32 位通用寄存器：EAX，ECX，EDX，EBX，ESP，EBP，ESI 和 EDI。其中，ESP 中是执行 PUSHAD 指令之前的值。POPAD 指令则以相反顺序从堆栈中弹出这些通用寄存器。与此类似，80286 处理器引入的 PUSHA 指令按照以下顺序将数值压入所有的 16 位寄存器：AX，CX，DX，BX，SP，SI 和 DI。POPA 指令则以相反顺序弹出这些寄存器。

如果在一个过程中修改了很多 32 位寄存器，那么可以在过程的开始和结束分别用 PUSHAD 和 POPAD 指令保存和恢复寄存器的值。下面以一个代码片段为例：

```
MySub PROC
    pushad                          ; 保存通用寄存器的值
    .
    .
    mov eax,…
    mov edx,…
    mov ecx,…
```

```
    .
    .
    popad                          ; 恢复通用寄存器的值
    ret
MySub ENDP
```

对于以上这个例子,存在一种特殊情况:当过程通过一个或多个寄存器返回结果时,不应该使用 PUSHA 或 PUSHAD 指令。在下面这个例子中,ReadValue 过程想要通过 EAX 返回一个整数,但对 POPAD 的调用将会覆盖 EAX 中的返回值:

```
ReadValue PROC
    pushad                         ; 保存通用寄存器
    .
    .
    mov eax, return_value
    .
    .
    popad                          ; 覆盖了 EAX!
    ret
ReadValue ENDP
```

**例题 5-1**:反转字符串

RevStr.asm 程序循环遍历字符串并把每个字符都压入堆栈,然后按相反的顺序从堆栈中弹出字符,并保存在原本的字符串变量中。因为堆栈是一个 LIFO(后进先出)的结构,所以字符串中的字符顺序就被反转了:

```
TITLE Reversing a String     (RevStr.asm)
INCLUDE Irivine32.inc
.data
aName BYTE "Happy New Year", 0
nameSize = ($ - aName) - 1
.code
main PROC
; 把 aName 中的每个字符都压入堆栈
    mov ecx, nameSize
    mov esi, 0
L1: movzx eax, aName[esi]          ; 取一个字符
    push eax                       ; 压入堆栈
    inc esi
    loop L1
; 从堆栈中按反序弹出字符
; 并存储在 aName 数组中
    mov ecx, nameSize
    mov esi, 0
L2: pop eax                        ; 取一个字符
    mov aName[esi], al             ; 保持在字符串中
    inc esi
    loop L2
; 显示 aName
    mov edx, OFFSET aName
```

```
    call WriteString
    call Crlf
    exit
main ENDP
End main
```

## 本节习题

(1) 简述堆栈的概念及其在计算机中的作用。

(2) 运行时栈与编译时栈有何区别？它们各自的作用是什么？

(3) 描述堆栈是如何在程序执行过程中被使用的。

(4) 简述堆栈是如何实现后进先出(LIFO)的数据结构的。

(5) 简述 PUSH 指令的概念和语法规则。它是如何将数据压入堆栈的？

(6) 简述 POP 指令的概念和语法规则。它是如何从堆栈中弹出数据的？

(7) 编写一段代码，使用 PUSH 指令将数据 1、2、3 依次压入堆栈，并使用 POP 指令将其弹出并存储到变量中。

(8) 设计一个简单的函数调用过程，包括参数的传递和局部变量的分配，使用 PUSH 和 POP 指令模拟堆栈操作。

# 5.3 过程的定义和使用

读者如果学过其他的高级程序设计语言，就会明白将程序分成子程序的作用。任何复杂的问题能够被理解、实现和有效测试的基础是首先被分解为一系列的任务。在汇编语言中，一般使用术语“过程”(procedure)表示子程序。在其他语言中，子程序会被称为方法或函数。

对于面向对象的程序设计，一个类中的函数或方法可以视为被封装在一个汇编语言模块中的过程和数据的集合。由于汇编语言的发明远早于面向对象的程序设计语言，因此在汇编语言中并没有某些高级语言中的正式结构。如果想要使用类似的结构，必须由程序员来主动定义。

## 5.3.1 过程的概念

过程可以被非正式地定义为以返回语句结束的命名语句块。在汇编语言中，过程是一种用来封装一段代码以便于重复使用的结构。它们允许程序员将大的程序分解成较小的、更容易管理的部分，每个部分都将执行特定的任务。

过程是在汇编代码中定义的，通常以一个标签开始，以 RET 指令结束。过程中可以包含任何正常的指令，包括对其他过程的调用。其他部分的代码可以通过 CALL 指令调用过程。当过程被调用时，程序执行流会跳转到过程的起点，执行其内部代码，然后通过 RET 指令返回到调用它的地方。

当过程被调用时，返回地址会被压入调用者的栈中。过程还会使用栈来保存寄存器的值，这些寄存器可能会在过程中被修改，但调用者希望它们的值在过程调用后保持不变。此