目录

第 6 章 顺序和分支程序设计 102
6．1 概述 102
6．2 程序的基本结构 104
6.2.1 处理器选择伪指令 104
6.2.2 存储模型说明伪指令 105
6.2.3 段定义及程序结束 106
6．3 转移控制指令 106
6.3.1 转移指令概述 106
6.3.2 简单条件转移指令 107
6.3.3 无符号数条件转移指令 107
6.3.4 有符号数条件转移指令 109
6.3.5 无条件转移指令 110
6．4 简单分支程序设计 111
6.4.1 C 语言的 if 语句与汇编语句的对应 111
6.4.2 分支程序设计示例 113
6.4.3 分支程序设计注意事项 116
6．5 多分支程序设计 117
6.5.1 多分支向无分支的转化 117
6.5.2 switch 语句的编译 120
6．6 条件控制流伪指令 122
习题 6 125
上机实践 6 126

第 6 章 顺序和分支程序设计

在汇编语言程序中，最常见的形式有以下几种：顺序程序、分支程序、循环程序、子程序。这几种程序的设计方法是汇编语言程序设计的基础。本章首先介绍一般的程序设计应注意的问题以及程序的基本结构，重点介绍分支程序设计方法和分支程序设计应注意的问题。所举的实例都是在 x86-32 下 32 位段扁平内存管理模式程序。

6．1 概述

设计一个程序通常从两个方面入手：一是要认真分析任务需求，选择好解决方法；二是要针对选定的算法，选用合适的指令，编写高质量的程序。一个高质量的程序不仅要满足设计的要求，而且还应尽可能实现以下几点：

（1）结构清晰、简明、易读、易调试

（2）执行速度快

（3）占用存储空间少

速度和空间问题有时是矛盾的，这就需要加以权衡，解决矛盾的主要方面。一般情况下，要优先考虑程序的易读性。现在的计算机 CPU 的性能都很高，内存也比较大，因此对于小规模程序，速度和空间都不存在问题。

为了方便阅读和调试，还要写出程序的说明、注释。并不需要在每条语句后面都写注释，例如 mov ax,0；若写注释“将 0 送到 ax 中”，是毫无意义的注释。因为阅读程序的人，是懂得语句的含义的。写注释的主要目的是为了让别人或自己更易读懂程序。

本人在写程序时，常常先写注释然后再写程序。写注释的作用是理清思路，只有逻辑清晰时才容易犯错误。注释的内容包括：一段程序的功能、算法思想、寄存器的功能分配。对于比较简单明了算法可以简写算法思想，没有必要费功夫详细描述，但对于绕弯较多的算法要仔细写出算法思想。

一、汇编语言程序设计的一般步骤

（1）分析问题，选择合适的解题方法，将程序设计成多个模块（即子程序、函数）的形式，每个模块相对独立，各自完成一定的功能；

（2）根据具体问题，确定输入输出数据的格式，分配存储区并给变量命名；

（3）对于每一个模块，分别确定使用的寄存器功能；

（4）绘制程序的流程图，将解题方法和步骤用程序流程图的形式表示出来；

（5）根据流程图编写程序；

（6）静态检查程序是否达到所需要的目标。静态检查就是在上机调试之前，采用人工阅读程序的方法检查程序是否满足设计要求，有无语法或逻辑错误等。程序经过静态检查且修改完善之后再上机调试、运行，将事半功倍。

对初学者来说，特别要注意的是画流程图。流程图由特定的框状、图形符号及简单的文字说明组成，它用来表示数据处理过程的步骤，能形象地描述逻辑控制结构及数据的流程，清晰地表达算法的全貌，具有简洁、明了、直观等特点，便于简化结构、推敲逻辑关系、排除设计错误、完善算法，对设计程序很有帮助。初学时有人不习惯画流程图，总想动手就写指令，这样容易出漏洞，造成逻辑上的混乱，往往在上机调试时出现语法错误或逻辑错误，程序很难顺利通过，不仅浪费时间，而且难以设计出高质量的程序。

流程图的详细程度依问题的需要而定。对于复杂的问题，应首先考虑程序的总体结构，画出各功能模块间的结构图，然后再将各模块的问题细化，画出各模块的流程图。依照细化了的流程图编写程序，工作效率会高得多。另外，在确定输入输出数据格式、分配存储区及寄存器、变量命名时，应依据机器硬件的特性，精确到字节、字或双字等。

在编写程序的过程中，不需要等待程序都写完后，再进行编译。建议写完一部分程序，就立即进行编译，看看有无语法错误。特别是对于初学者在写汇编语言程序时，往往会忘记一些语法规则，尽早发现语法错误，增强了对语法规则的理解，就会在后面的编程中少犯错误。

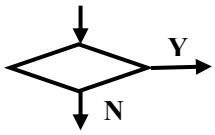
编写完程序，编译通过后，可以先阅读程序，此时可以发现 70%-80%的错误。当然，在阅读时，需要将自己的大脑当成一个 CPU，边读边想语句执行后对寄存器、内存等有何影响。在调试程序时，按模块调试，检查进入模块时的数据是否正确，模块处理结束后加工结果是否正确。需要强调的是，在读、写、上机调试程序的过程中，一定要仔细、认真地逐条推敲所学指令的功能、用途。马马虎虎、似是而非是不可能设计出好程序的。

下面给出汇编语言程序流程图中的常用符号的说明。

二、几种框图符号的说明

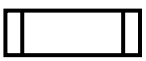
（1）起始、终止框：它用来表示程序的开始和结束。

（2）判断框：它表示程序在这里进行判断以决定程序的流向。判断的条件记入此框中。它具有两个出口，在每个出口处应标明出口的条件（用“Y”表示条件满足，用“N”表示条件不满足）。

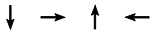


（3）处理说明框：它用来代表一段程序（或一条指令）的功能。其功能应在框内进行简单、明确的说明，尽量采用直观、自然的语言和符号，不要书写指令语句。

（4）子程序或过程调用框：它用来说明要调用的子程序或过程。框中要标明子程序或过程的名字。



（5）流向线：它总是指向下一个要执行的操作，即表示流程的方向。



（6）连接框：框中可标入字母或数字。当框图较复杂或分布在几张纸上时，就用连接框来表示它们之间的关系。相同符号的连接框是互相连接的。

如图 6.1，给出了连接符号使用的示意图。左右两个虚线矩形框表示两张不同的纸。在左边的纸内，连接框 3 的程序是相连的；在左右两张纸中，连接框 1 的程序是前后相连的，连接框 2 的程序也是前后相连的。

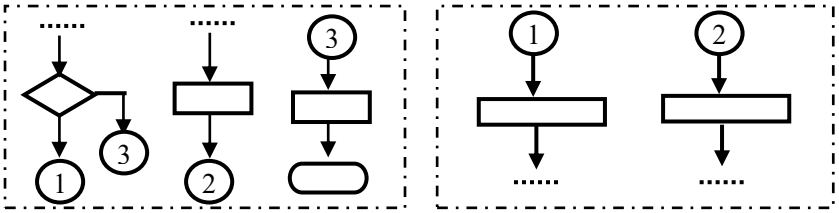


图 6.1　算法流程图连接框的用法示例

在程序调试时，注意测例的充分性，特别是对一些边界条件，检查程序的运行结果是否正确。例如，写一个字符串比较程序，设有一个串为“ABCD”，可以测试它其和“ABCF”、“ABC”、“ABCDE”的比较，看比较的结果是否正确。

6．2 程序的基本结构

在前面的章节中，大家已看到汇编语言程序由多个段组成。除此之外，含有引用的外部函数的说明以及实现这些函数的库的说明和一些伪指令。

x86 宏汇编语言包括如下几类伪指令：

- 处理器选择伪指令
- 存储模型说明伪指令
- 数据定义伪指令
- 符号定义伪指令
- 段定义伪指令
- 过程定义伪指令
- 程序模块的定义与通讯伪指令
- 宏定义伪指令
- 条件汇编伪指令
- 格式控制、列表控制及其它功能伪指令。

有些伪指令已在前面介绍过，有些在后面章节用到的时候进行介绍。

6.2.1 处理器选择伪指令

x86 系列微处理器的功能向下兼容，但每高一档 CPU 都会在前一档 CPU 上增加若干功能更强的新指令，因此，在编写程序时要合理选择所使用的处理器，即告诉汇编程序选择何种 CPU 所支持的指令系统。x86 提供了处理器选择伪指令，其主要伪指令的格式和功能列表如 6.1 所示。

表 6.1 处理器选择伪指令

.8086	接受 8086 指令（缺省方式）
.186	接受 80186 指令
.286	接受除特权指令外的 80286 指令
.286P	接受全部的 80286 指令，包括特权指令

.386	接受除特权指令外的 80386 指令
.386P	接受全部的 80386 指令，包括特权指令
.387	接受 80387 数学协处理器指令
.486	接受除特权指令外的 80486 指令,包括浮点指令
.486P	接受全部 80486 指令,包括浮点指令、特权指令
.586	接受 Pentium 指令，特权指令除外
.586P	接受全部 Pentium 指令
.686	接受 Pentium Pro 指令，特权指令除外
.686P	接受全部 Pentium Pro 指令
.MMX	接受 MMX 指令
.XMM	接受 SSE、SSE2、SSE3 指令

在源程序中，处理器选择伪指令一般放在程序的开始处，表示下面的段均使用该处理器所支持的指令系统。

提示：对一个 C 语言程序，在编译时可让其生成汇编语言程序。设置“项目属性->C/C++->输出文件->汇编程序输出”为“带源代码的程序集（/FAs）”。打开生成的汇编语言程序，可以看到 C 语言程序采用的处理器选择伪指令为“.686P”和“.XMM”。当然，通过分析由 C 语言程序编译生成的汇编语言程序，可以学到不少的知识。

6.2.2 存储模型说明伪指令

格式：.model 存储模型 [,语言类型]

功能：“存储模型”指定内存管理模式，常用的存储模型见表 6.2 所示。对于 Win32 程序来说，由于内存是一个连续的 4GB 段，因此应该选择平坦模式 flat。

表 6.2 常用的存储模型

存储模型	段的大小	代码访问范围	数据访问范围	备注
tiny	16 位	NEAR	NEAR	代码和数据全部放在同一个 64K 段内，常用于生成.COM 程序
small	16 位	NEAR	NEAR	代码和数据在各自的 64K 段内，代码总量和数据总量均不超过 64K
compact	16 位	NEAR	FAR	代码总量不超过 64K，数据总量可以超过 64K
medium	16 位	FAR	NEAR	代码总量可超过 64K，数据总量不超过 64K
large	16 位	FAR	FAR	代码和数据总量均可超过 64K，但单个数组不超过 64K
huge	16 位	FAR	FAR	代码和数据总量均可超过 64K，单个数组可超过 64K
flat	32 位	NEAR	NEAR	代码和数据全部放在同一个 4G 空间内

本书中的例子，所使用的存储模型都是 flat。

“语言类型”指定了函数命名、调用和返回的方法，例如 c、pascal 或 stdcall 等。根据 Win32 中 API 的要求，应选择 stdcall 类型（采用堆栈法传递参数，参数进栈次序为：函数原型描述的参数中最右边的参数最先入栈、最左边的最后入栈；由被调用者在返回时清除参数占用的堆栈空间）。此处省略语言类型时则应在函数说明/定义语句 proto、proc 等中说明。

其他的语言类型还有 stdcall、pascal 等。stdcall 语言类型的参数传递方式与 c 相同，不同之处是在调用函数中清除参数所占的空间，即在函数的实现体中在返回时清除参数占用的堆栈空间，这是 Win32 中 API 的要求的参数传递和清除方式。

该指令必须放在源文件中所有其它段定义伪指令之前且只能使用一次。

6.2.3 段定义及程序结束

汇编语言程序由多个段组成。一个段的开始也是前一个段的结束。

定义数据段伪指令：.data 或 .data?

定义代码段伪指令：.code [段名]

定义堆栈段伪指令：.stack [堆栈字节数（省略时为 1024）]

常数（只读）数据段定义：.const

程序结束伪指令：end [表达式]

该语句为源程序的最后一个语句，用以标志整个程序的结束，即告诉汇编程序，汇编工作到此为止。其中表达式为可选项。如果 end 后面带有表达式，其值必须是一存储器地址。该地址为程序的启动地址，即该程序在计算机上运行时第一条被执行指令的地址。一般情况下，表达式为一个标号，或者是一个子程序（函数）的名字。如果不带表达式，则说明该程序不能单独运行，这时，它往往是作为一个子模块供另外的程序调用。

编程时一定要注意不可将 end 语句错误地安排在程序中间。因为汇编程序在将源程序汇编成目标程序时，是以 end 为汇编工作的结束标记的，这样一来，end 后面的语句就不可能被翻译成目标代码了。

6．3 转移控制指令

转移指令包含条件转移指令和无条件转移指令，其特点是改变程序的执行顺序（即改变指令指针 cs：EIP 的值）。条件转移指令根据条件标志的状态判断是否转移。无条件转移指令则不作任何判断，无条件地转移到指令中指明的目的处执行。转移指令较多，下面分条件转移和无条件转移两种情况介绍这些指令。

除了这些转移指令外，还有一些指令也会改变程序的执行顺序，包括循环指令、子程序调用和返回指令、中断调用和返回指令。这些指令将在后面的章节介绍。

6.3.1 转移指令概述

根据条件转移指令在转移时所依据的条件的特点，可以分成以下四类：

（1）简单条件转移指令。这类指令是根据单个标志的状态决定是否转移，共有 10 条。

（2）无符号数条件转移指令，共有 4 条；

（3）有符号数条件转移指令，共有 4 条；

（4）无条件转移指令，共 6 条，本节只介绍 1 条，其他在子程序设计等章节介绍；

（5）循环转移指令，共有 4 条，将在循环程序设计中介绍。

语句格式：[标号:] 操作符 标号

功能：如果转移条件满足，则(EIP)+ 位移量 → EIP，否则，执行紧跟转移指令之后的那条指令。

在转移指令语句中，操作符后面的标号指向了满足转移条件时程序准备执行的指令语句的位置，但标号必须与转移指令在同一段内。经过汇编程序汇编后，该标号被翻译成紧跟转移指令

后的那条指令的(EIP)到转移目的地址处之间的字节距离（称为位移量）。根据汇编程序计算的结果，位移量将确定成一个 8/16/32 位的有符号数（循环转移指令的位移量只能为 8 位）。当它为正数时，表示往前（下）转移；当它为负数时，表示往回（上）转移。在程序设计中，如果代码段为 32 位段，则位移量可以是 8/32 位数。如果要转移的距离超出上述规定的范围时，汇编程序会报错。如果代码段为 16 位段，则位移量可以是 8 位（位移量在-128～127 之间，也即向前转移时位移量不能超过 127 个字节，往回转移时，不能超过 128 个字节。这时将对应的标号称为短标号）。

6.3.2 简单条件转移指令

在 x86 机器中，标志 CF、ZF、SF、OF、PF 分别为 1 或为 0，可表示 10 种状态，每条简单条件转移指令只简单地关心某一个标志的一种状态，因而设置了 10 条简单的条件转移指令，如表 6.3 所示。

表 6.3　简单条件转移指令

指令名称	助 记 符	转移条件	功能说明
相等/等于 0 转移	je/jz	ZF = 1	前次操作结果是否相等或等于 0
不相等/不等于 0 转	jne/jnz	ZF = 0	前次操作结果是否不相等或不等于 0
为负转移	js	SF = 1	前次操作结果是否为负
为正转移	jns	SF = 0	前次操作结果是否为正
溢出转移	jo	oF = 1	前次操作结果是否溢出
未溢出转移	jno	0F = 0	前次操作结果是否未溢出
进位位为 1 转移	jc	CF = 1	前次操作结果是否有进位或借位
进位位为 0 转移	jnc	CF = 0	前次操作结果是否无进位或借位
偶转移	jp/jpe	PF = 1	前次操作结果中 1 的个数是否为偶数(even)
奇转移	jnp/jpo	PF = 0	前次操作结果中 1 的个数是否为奇数(odd)

前次操作结果是指在本转移指令执行前最后执行的且影响标志位的指令的运算结果。

在这 10 条简单条件转移指令中，有 6 条指令（je/jz、jne/jnz、jc、jnc、jp/jpe 和 jnp/jpo）不受数的符号位的影响，因此，可以将它们划到无符号数条件转移指令的类别中。剩下的 4 条指令（js、jns、jo 和 jno）属于有符号数条件转移指令的类别。

具体应用举例将在分支程序设计中介绍。

6.3.3 无符号数条件转移指令

为了更好的理解无符号数条件转移指令，先看一段 C 语言程序，从宏观层面理解无符号数比较。

```
int flag=0;
unsigned int  ux = -1;
unsigned int  uy = 3;
if (ux > uy)
   flag = 1;
```

这里定义了 2 个无符号整型变量，比较二者的大小，执行后，flag=1。其对应的反汇编窗口的代码如下。

```
int flag = 0;
00A517C8 C7 45 F8 00 00 00 00  mov  dword ptr [flag],0
unsigned int  ux = -1;
00A517CF C7 45 EC FF FF FF FF mov   dword ptr [ux],0FFFFFFFFh
unsigned int  uy = 3;
00A517D6 C7 45 E0 03 00 00 00 mov   dword ptr [uy],3
if (ux > uy)
00A517DD 8B 45 EC          mov      eax,dword ptr [ux]
00A517E0 3B 45 E0          cmp      eax,dword ptr [uy]
00A517E3 76 07             jbe      main+4Ch (0A517ECh)
    flag = 1;
00A517E5 C7 45 F8 01 00 00 00 mov   dword ptr [flag],1
00A517EC ……
```

从反汇编结果可以看到，在 ux 和 uy 进行比较时，未使用“cmp ux,uy”，因为不能两个操作数同时是存储器寻址方式。它首先是将(ux)->eax，然后(eax)和(uy)进行比较。在选用转移指令是，它对条件“>”求反为“<=”,即“<=”时，转移到 if 语句的结束处。jbe 的含义是低于或等于转移（Jump if Below or Equal）。由 ux=0ffffffffh，uy=3,将两者视为无符号数，有 ux>uy，jbe 的条件不成立，故会执行 flag=1。执行该语句后，再执行 00A517EC 处的语句。

从反汇编结果中，还可以看 jbe main+4Ch (0A517ECh) 中给出的地址正是转移的目的地地址。但是其机器指令编码是 76 07。07 就是转移的位移量。当将 jbe 指令取出后，EIP 增加本指令的长度，此时(EIP)= 00A517E5h，即 jbe 之下的一条指令的地址。0A517ECh 与 00A517E5h 之间的间距正好是 7。若 below or equal 的条件成立，（EIP）+位移量->EIP，转移到目标位；若条件不成立，则(EIP)不变，即执行“mov dword ptr [flag],1”。

无符号数条件转移指令往往跟在比较指令之后，根据运算结果设置的条件标志状态确定是否转移。这类指令视比较对象为无符号数。根据不同状态，设置了高于、高于或等于、低于、低于或等于四条指令。下面将从微观层面介绍 CPU 是如何判断无符号比较转移的条件是否成立的。

1）ja/jnbe

ja 即高于转移（Jump if Above），jnbe 即不低于且不等于转移（Jump if Not Below or Equal）。ja/jnbe 是当 CF=0 且 ZF=0 时转移。它用于两个无符号数 a、b 的比较，若 a> b，则条件满足，实现转移。

2）jae/jnb

jae 即高于或等于转移(Jump if Above or Equal),jnb 即不低于转移(Jump if Not Below)。jae/jnb 是当 CF=0 或 ZF=1 时转移。由于 jae/jnb 指令用在比较、减运算之后的语义才是符合实际需要的，而此时 ZF 若为 1，CF 也必然为 0，因此，CPU 在执行该指令时只判断了 CF 标志，也即该指令与 jnc 指令是等价的。

3）jb/jnae

jb 即低于转移（Jump if Below），jnae 即不高于且不等于转移（Jump if Not Above or Equal）。jb/jnae 是当 CF=1 且 ZF=0 时转移。该指令与 jc 指令等价（在比较、减运算后，当 CF=1

时，ZF 就为 0）。

4）jbe/jna

jbe 即低于或等于转移(Jump if Below or Equal)，jna 即不高于转移(Jump if Not Above)。jbe/jna 是当 CF=1 或 ZF=1 时转移。

四条指令的共同特点是根据两个无符号数比较的结果，判断 CF、ZF 的状态是否满足转移条件。当满足条件时转移，否则顺序执行。

对于本节中的示例，(eax)=0FFFFFFFFh, (uy)=3; 执行 cmp eax,[uy]后，ZF=0，CF=0，SF=1，OF=0。由 jbe 转移的条件是 CF=1 或 ZF=1 知，其条件不成立，故不转移，因而 flag=1。

6.3.4 有符号数条件转移指令

在程序设计中，有时把处理对象视为有符号数（负数用补码表示）。当比较判断两个有符号数的大小时，要选用有符号数条件转移指令。有符号数条件转移指令根据条件标志 ZF、SF、OF 的特定组合决定是否转移，共设置了大于、大于或等于、小于、小于或等于四条转移指令，与无符号数转移指令相对应。

1）jg/jnle

jg 即大于转移(Jump if Greater)，jnle 即不小于且不等于转移(Jump if Not Less or Equal)。jg/jnle 是当符号标志 SF 与溢出标志 OF 具有相同状态（即 SF=OF）且 ZF=0 时转移。它用于两个有符号数 a、b 的比较。若 a> b，则条件满足，实现转移。

2）jge/jnl

jge 即大于或等于转移(Jump if greater or equal)，jnl 即不小于转移(Jump if Not Less)。jgE/JNL 是当 SF=OF 或 ZF=1 时转移（当两数相等时，在比较、减运算之后，不仅会使 ZF=1，而且 SF 也会等于 OF；所以，执行 jge/jnl 时，CPU 只判断 SF 是否等于 OF）。

3）jl/jnge

jl 即小于转移，jnge 即不大于且不等于转移。jl/jnge 是当 SF≠OF 且 ZF=0 时转移（与 jge/jnl 类似，CPU 在执行 jl/jnge 时，只需判断 SF≠OF 即可）。

4）jle/jng

jle 即小于或等于转移，JNG 即不大于转移。jle/jng 是当 SF≠OF 或 ZF=1 时转移。

可以看出，有符号条件转移指令判断两个有符号数的大小时利用了 SF 和 OF 标志，而不是单用 SF 标志。这是为了避免当运算结果存在溢出时，SF 表示的语义错误。

对于 6.3.3 节中的例子，将变量 ux 和 uy 前的 unsigned 删除掉，可看到如下的反汇编代码，

```
int flag = 0;
000817C8 C7 45 F8 00 00 00 00  mov   dword ptr [flag],0
int  ux = -1;
000817CF C7 45 EC FF FF FF FF  mov   dword ptr [ux],0FFFFFFFFh
int  uy = 3;
000817D6 C7 45 E0 03 00 00 00  mov   dword ptr [uy],3
if (ux > uy)
000817DD 8B 45 EC              mov    eax,dword ptr [ux]
```

```
000817E0 3B 45 E0             cmp     eax,dword ptr [uy]
000817E3 7E 07                jle     main+4Ch (0817ECh)
    flag = 1;
000817E5 C7 45 F8 01 00 00 00 mov     dword ptr [flag],1
000817EC ……
```

对于本例，(eax)=0FFFFFFFFh,(uy)=3; 执行 cmp eax，[y]后，ZF=0，CF=0，SF=1，OF=0。这与 6.3.3 节的结果是一样的。由 jle 转移的条件是当 SF≠OF 或 ZF=1 知，其条件成立，故转移，因而未执行 flag=1， 因此 flag=0。

总结：

① cmp 指令不区分有符号数、无符号数的比较，只是按结果设置标志位；

② 无符号数比较转移，记住两个单词 Above，Below；由它们很容易想到 4 条转移指令；

③ 有符号数比较转移，记住两个单词 Great，Less；由它们很容易想到 4 条转移指令；

④ CPU 判断转移条件是否成立有其固定的规则，但是编程者完全可以用人类更易理解的方法判断，即将两个数看成无符号数比较大小，或者将两个数看成有符号数比较大小；

⑤ 选择正确的转移指令。

6.3.5 无条件转移指令

无条件转移指令使 CPU 无条件地转移到指令指明的目的地址处执行，包括 jmp、call、ret、int、iret 和 into 指令。本节介绍 jmp 指令。call 和 ret 是子程序的调用和返回（相当于 C 语言的函数），将在子程序设计的章节中介绍。int、iret 是中断处理程序调用和返回指令；into 是溢出中断指令。

jmp 看起来是最简单的，不看任何条件都要转移。然而 jmp 指令却是一条较复杂的指令。它实现转移的基本形式有直接方式和间接方式两种，可通过相应的寻址方式得到要转移的目的地址。也正是因为有可以用间接寻址方式，使得编程可以编写得非常灵活。

直接方式： jmp 标号

间接方式： jmp opd ;(opd)为转移的目的地址

设有如下数据段和程序段：

```
.data
   p  dd  lp
.code
 start:
   jmp   lp          ; 方法 1，直接转移到标号 lp 处
   mov   ebx, p
   jmp   ebx         ; 方法 2，寄存器寻址，转移到（ebx）处，也即 lp 处
   jmp   p           ; 方法 3，直接寻址，转移到（P）处,也即 lp 处
   mov   ebx, offset p
  jmp   dword ptr [ebx] ; 方法 4，寄存器间接寻址，转移到（[ebx]）处
lp: ……
```

上述四种方法的转移结果完全是相同的。正是因为 jmp 可以用间接方式确定转移的目的地址，

使得我们可以发挥自己的想象力，编写出优美的程序。另外，通过使用非直接的跳转方式，会增加程序跟踪时的难度，可以用于程序的反跟踪。

6．4 简单分支程序设计

在C语言中，实现的分支的方法有 if 语句、switch 语句。本节介绍 if 语句的汇编语言实现。通过分析C语言中 if 语句的形式、if 语句的执行流程、与汇编语言语句的对应关系，可以比较轻松的用汇编语言写出分支程序。

6.4.1 C 语言的 if 语句与汇编语句的对应

从程序设计的角度看，不论是C语言程序还是汇编语言程序，算法是一样的，只不过将算法翻译成语句时，对应的语句形式有所差别。

(1) 单分支的 if 语句

设有 C 语言单分支的 if 语句：

```
if （x ==y）{     // 括号中应该写“条件表达式”，
                  // 为了入门，便于用汇编写程序，先给了一个最简单的条件表达式
    statements …… // statements 是一组语句的抽象表示。
}
```

该语句判断条件是否成立，条件成立是执行 then 分支，然后到 if 语句之下；条件不成立，直接跳转到 if 语句之下。

在汇编语言中同样要先判断条件是否成立，但是转移指令都是“某条件成立”转移。为了在原条件成立时，直接执行其下的 then 分支，可以将条件取反。对于转移指令，有一个转移到的目的地，因而在 if 语句结束处要增加一个标号。用汇编表达结果如下。

```
    mov  eax, x    ; 设 x,y 都是双字类型的变量
    cmp  eax, y
    jne  lp        ; 不相等转移到 lp 处
    statements ……
lp:
```

注意，写程序时要记住汇编语言的语法规定，双操作数不能同时是存储器寻址方式，不能直接写 “cmp x,y”。

(2) 双分支的 if-else 语句

设有 C 语言单分支的 if 语句：

```
if （x ==y）{
   statements1 …… // statements1 是一组语句的抽象表示。
}else {
   statements2 …… // statements2 是一组语句的抽象表示。
}
```

该语句判断条件是否成立，条件成立是执行 then 分支，then 分支条件执行完后，无条件转

移到后到 if 语句之下；条件不成立，跳转到 else 分支，else 分支执行完后，也直接到 if 语句之下。因此，整个语句有 2 处转移，在判断条件时有一个条件转移，then 分支结束处有一个无条件转移。有 2 个地方要设置标号，一是 else 分支的开头，一是 if 语句结束处。

同样，在汇编语言程序中，为了保持 then 分支和 else 分支的顺序，将条件取反，即原条件不成立时转移。用汇编表达结果如下。

```
        mov   eax, x     ; 设 x,y 都是双字类型的变量
        cmp   eax, y
        jne   l1         ; 不相等转移到 l1 处
        statements1 ……    ; then 分支语句
        jmp   l2         ; 一定要给 then 分支出口，否则就会继续执行 else 分支的语句
    l1:
        statements2 ……    ; else 分支语句
    l2:
```

(3) 复杂条件表达式的翻译

条件表达式有多个子条件时，要根据子条件之间的逻辑运算关系，判断条件是否成立。

假设条件表达式为：（条件 1 && 条件 2），显然，当条件 1 不成立时，可直接跳转；条件 1 成立时，再判断条件 2，当条件 2 不成立时跳转。如果条件表达式为：（条件 1 || 条件 2），则在条件 1 成立时，直接跳转到 then 分支；否则继续判断条件 2 是否成立,条件 2 不成立，跳转到 else 分支。在 then 分支结束处无条件跳转到 if 的结束（下一条语句开始）处。下面给出了 C 语言程序段及其反汇编后的结果。C 语言程序段如下：

```
int flag ;
int  x = 3;
int  y = -1;
if (x > 0 || y > 0)
    flag = 1;
else
    flag = 0;
```

反汇编结果如下：

```
00241828 C7 45 EC 03 00 00 00 mov   dword ptr [x],3
0024182F C7 45 E0 FF FF FF FF mov   dword ptr [y],0FFFFFFFFh
00241836 83 7D EC 00          cmp   dword ptr [x],0
0024183A 7F 06                jg    main+42h (0241842h)
0024183C 83 7D E0 00          cmp   dword ptr [y],0
00241840 7E 09                jle   main+4Bh (024184Bh)
00241842 C7 45 F8 01 00 00 00 mov   dword ptr [flag],1
00241849 EB 07                jmp   main+52h (0241852h)
0024184B C7 45 F8 00 00 00 00 mov   dword ptr [flag],0
00241852 ……
```

(4) 程序的优化

在写分支语句时，将 if 后的条件表达式求反、then 和 else 分支交换位置，则程序的功能是

不变的。将哪一个分支作为 then 分支，哪一个作为 else 分支更好？从程序的可读性来看，将语句组少的那个分支作为 then 分支更好，这样可以避免头重脚轻，相对容易的找到 else 分支从何处开始。

if 后的条件表达式有多个子条件时，怎样摆放子条件的顺序会更好？假设两个子条件是 and 关系，70%的测例都会使子条件 1 成立，但只有 30%的测例是条件 2 成立。将条件 2 放在前面，70%的例子都不会使其成立，因而也就不会去判断另一个条件，即只有 30%的情况要判断两个条件，否则将条件 1 放在前面就是有 70%的情况要判断两个条件，当然将条件 2 放在前面程序执行起来就更快。其他情况，读者可以自己去琢磨。

6.4.2 分支程序设计示例

本节通过一个示例，介绍如何编写一个分支程序。

【例 6.1】 编制计算下面函数值的程序

$$r=\begin{cases}1 & x\geqslant 0,\ y\geqslant 0\\ -1 & x<0,\ y<0\\ 0 & x、y\ 异号\end{cases}$$

其中 x、y、r 是三个双字类型的有符号整型变量。

分析题目，我们可以给出更细的一个只对一个变量判断的方法。

$$\begin{cases}x\geqslant 0 \begin{cases}y\geqslant 0 & r=1\\ y<0 & r=0\end{cases}\\ x<0 \begin{cases}y\geqslant 0 & r=0\\ y<0 & r=-1\end{cases}\end{cases}$$

在数据段，需要定义变量 x、y、r，并赋予某个初值。程序流程图见图 6.2。

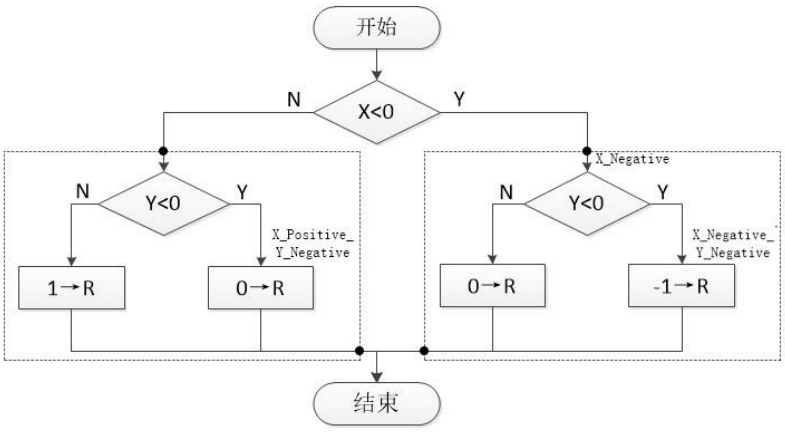


图 6.2 程序流程图

从图 6.2 中可以看到，两个虚框代表了 2 个分支，每个虚框都是单入口、单出口，这样模块化程度会较高。

为了更好的与程序对应，将流程图做了一个变形，如图 6.3 所示，各个框出现的顺序就是程序中语句的排列顺序。

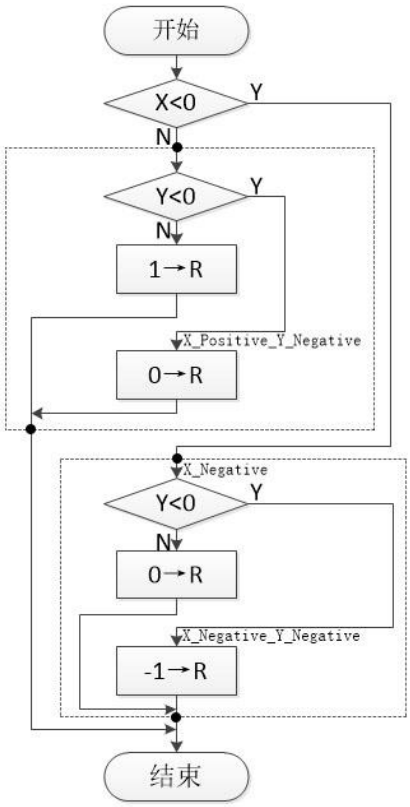


图 6.3 程序流程图

源程序如下：

```
.686P
.model flat, stdcall
  ExitProcess proto :dword
  printf      proto C :ptr sbyte, :vararg
  includelib  kernel32.lib
 includelib   libcmt.lib
 includelib   legacy_stdio_definitions.lib.data
  lpFmtdb  "x=%d y=%d r=%d",0ah, 0dh, 0
  x  sdword  5
  y  sdword  6
  r  sdword  0
.stack 200
.code
 main  proc  c
   cmp  x, 0
   js   X_Negative
        ; 此处向下是  x>=0 的情况
   cmp  y, 0
   js   X_Positive_Y_Negative
```

```
    mov  r, 1
    jmp  exit
X_Positive_Y_Negative:
    mov  r, 0
    jmp  exit
         ; 至此，x>=0 的情况处理结束
         ; 以下是 x<0 的情况
X_Negative:
    cmp y, 0
    js  X_Negative_Y_Negative
    mov r, 0
    jmp exit
X_Negative_Y_Negative:
    mov  r,-1
exit:
    invoke printf,offset lpFmt,x,y,r
    invoke ExitProcess, 0
main  endp
end
```

程序中用“js”判断 x 是否为负，用“jl”判断 y 是否小于零。此处，选用 js 或 jl 都可以，但不能选用 jb。因为 jb 是无符号数条件转移指令。在判断 y 是否大于等于零时选用了有符号数条件转移指令“jge”，如果选用“jae”，则造成运行时的转移混乱。

从程序中，可以看到 “mov r，0”出现了 2 次，可以将其优化，只让其出现一次。对如下段修改一下即可。

```
修改前：  js   X_Negative_Y_Negative
          mov  r, 0
          jmp  exit
修改后：  js   X_Negative_Y_Negative
          jmp  X_Positive_Y_Negative
```

但是从程序的可读性来看，采用先用 x>=0 分 X_Positive 和 X_Negative 两类，再对每一类用 y>=0 再分两类，共四种情况，程序的模块化程序会高一些。

在程序设计中，我们也尽可能使用一些含义比较清楚的名字，如 X_Negative，表示 x<0；X_Negative_Y_Negative 表示 x<0 且 y<0。

在写分支程序时，会出现较多的标号，标号不得同名。但在程序中，有些标号只是给紧靠着它的转移指令中使用，其取名并没有多大的实际意义。在 Microsoft 的汇编语言程序设计中，引入了符号 @@ 作为一个标号，在一个程序中可有多个 @@ 标号。在转移指令中，采用 @F 表示转移到离该语句最近的下一个标号 @@处，采用 @B 表示转移到离该语句最近的上一个标号 @@处。对于例 6.1 中的核心代码可以写成如下形式。

```
          cmp  x, 0
```

```
        js    X_Negative     ; 此处向下是  X>=0 的情况
        cmp   y, 0
        js    @F
        mov   r, 1
        jmp   exit
    @@:
        mov   r, 0
        jmp   exit
          ; 至此，x>=0 的情况处理结束，  以下是 x<0 的情况
X_Negative:
        cmp   y, 0
        js    @F
        mov   r, 0
        jmp   exit
    @@:
        mov   r, -1
exit:
```

该程序编译后的结果与例 6.1 程序是一样的。采用@@、@F、@B 减少了一些标号的命名，也可以有意识的培养不随意跳转、模块化组织程序的习惯。

6.4.3 分支程序设计注意事项

在分支程序设计中，要注意以下问题。

（1）选择合适的转移指令，否则就不能转移到预定的程序分支。

（2）要为每个分支安排出口。如果漏写了出口跳转，程序运行会产生逻辑错误。

（3）应把各分支中的公共部分尽量提到分支前或分支后的公共程序段中，这样可使程序简短、清晰。

例如：编写程序段 |(eax)|+(ebx) ->x，即(eax)的绝对值+(ebx)->x,x 是双字类型的变量。

```
        or    eax，eax
        js    l1            ; 若（eax）为负则转 l1
        add   eax，ebx      ;（eax）为正时，直接与（ebx）相加
        mov   x，eax
        jmp   exit          ; 无条件转 exit
l1:     neg   eax           ;（eax）为负时，转换为绝对值后与（ebx）相加
        add   eax，ebx
        mov   x，eax
exit:  ……
```

其中，指令“add eax，ebx”和“mov x，eax”是两个分支共有的，故程序可简化为：

```
        or    eax，eax
        jns   l1            ; 若（eax）为正则转 l1
        neg   eax
l1:     add   eax，ebx
```

```
        mov  x，eax
    exit:  ……
```

（4）在分支比较多时，流程图中对每个分支判断的先后次序应尽量与问题提出的先后次序一致。而编写程序时也要与流程图中各分支的先后次序一致。这样编写的程序出错机会少，且清晰、易读、易检查。

（5）在调试分支程序时，要假定各种可能的输入数据，沿着每一支路逐一检查，测试程序是否正确。只有所有分支都满足设计要求时，才能保证整个程序满足设计要求。

（6）不要在程序中随意摆放各个分支和随意跳转，最好是将两个分支紧靠在一起，使得整个条件语句像一个小模块，有一个起始点和一个终止点，从起始点到终止点之间的语句全部都是该条件语句的组成部分，而不含有其他语句。

（7）在有两个分支时，一般将语句少的分支（短分支）作为 then 分支，写在前面；将语句多的分支（长分支）作为 else 分支，写在后面，这样写可读性更好。

(8) 一个分支有多个条件组合而成时，将最不容易满足的条件写在前面，优先判断该条件是否成立，可以减少其他指令的运行几率，提高程序运行速度。

6．5 多分支程序设计

6.5.1 多分支向无分支的转化

【例 6.2】 编写一个程序，统计一个全部由小写字母组成的字符串中各个字母出现的次数。

首先，由学习过的 C 语言编程知识可以快速的设计算法和有关的数据结构。

① 定义一个数组 int count[26],来依次存放各个字母的出现次数，count[0]表示‘a’ 出现次数，count[1]表示‘b’ 出现次数，以此类推。

② 用循环来依次处理字符串中的各个字符；

③ 对于字符串的一个字符 c,可以判断其是否为‘a’,是则 count[0]加 1；否则判断其是否为‘b’,是则 count[1]加 1，依次类推。

显然，③ 给出的算法并不好。用该算法写程序，不但程序长，而且执行效率低。利用字母 ASCII 的规律，完全可以避开多分支的比较。

修改后的算法：③ 对于字符串的一个字符 c,直接 count[c-‘a’]增 1。

用汇编语言完成的数据定义和程序片段如下：

```
    count    dd    26 dup(0)
    buf      db    ‘……’,0   ; 小写字母组成的串，串以 0 结束，用于循环控制
; 用 ebx 来指示是字符串中的第几个元素
; 用 AL 来存放读到的字母的 ASCII，进而演变成是字母表的第几个字母
    mov  ebx,  0
l1:
    mov    al, buf[ebx]
    cmp    al, 0
    je     exit
```

```
        sub     al, 'a'
        movzx   eax, al
        inc     count[eax*4]
        inc     ebx
        jmp     l1
    exit: ……
```

例 6.2 给出程序说明多个分支的语句是可以向无分支的语句转换的。

【例 6.3】 编写一个程序，当 x==1 时，显示‘Hello，One’； 当 x==2 时，显示‘Two’；当 x==3 时，显示‘Welcome,Three’,……。即 x 为不同的值，显示不同的串。

显然，可以用多个条件转移指令，完成该程序，例如:

```
        cmp   x,   1
        je    l1
        cmp   x,   2
        je    l2
        cmp   x,  3
        je    l3
    l1: 显示串 1
        jmp   exit
    l2: 显示串 2
        jmp   exit
       ……
```

如果 x 的值很多，也会使程序显的庸长，执行效率低。

换一种思路，显示不同的串，只是一个给 printf(“%s”,串首地址）提供不同的首地址。通过构造一个地址表，即将各个串的首地址依次排列在一个表中，然后从地址表中取出某个地址，作为 printf 的一个参数即可。汇编语言程序如下。

```
.686P
.model flat, stdcall
 ExitProcess proto :dword
 printf      proto c :ptr sbyte, :vararg
 includelib  kernel32.lib
 includelib  libcmt.lib
 includelib  legacy_stdio_definitions.lib
.data
 lpFmt db  "%s",0ah, 0dh, 0
 x     dd  3
 msg1  db  'Hello,One',0
 msg2  db  'Two',0
 msg3  db  'Welcome,Three',0
 vtable  dd  msg1, msg2, msg3
.stack 200
```

```
.code
 main  proc  c
   mov  ebx, x
   dec  ebx        ; ebx 用于存放是第几个串，从 0 开始编号的
   mov  eax, vtable[ebx*4]
   invoke  printf,offset lpFmt, eax
   invoke  ExitProcess, 0
main  endp
end
```

当然，程序中也可以不写 DEC ebx，而直接使用语句：

```
   mov  eax, vtable[ebx*4 -4]
```

该例说明，可以将一些无规律的信息，通过某种组织方式使得它有规律，从而更简单的写出程序。

【例 6.4】 编写一个程序，当 x==1 时，转移到 l1 处执行； 当 x==2 时，转移到 l2 处执行；当 x==3 时，转移到 l3 处执行。

同样，我们可以用指令地址表法，将要转移的目的地址放在一个表中，通过在该表，来查询要转移的目的地址。

```
    .686P
    .model flat, stdcall
      ExitProcess proto stdcall :dword
      printf      proto c :ptr sbyte, :vararg
      includelib  kernel32.lib
      includelib  libcmt.lib
      includelib  legacy_stdio_definitions.lib
    .data
      lpFmt db  "%s",0ah, 0dh, 0
      X     dd  3
      msg1  db  'Hello,One',0
      msg2  db  'Two',0
      msg3  db  'Welcome,Three',0
      ptable  dd  l1, l2, l3
    .stack 200
    .code
    main   proc  c
       mov  ebx, x
       mov  eax, ptable[ebx*4 -4]
       jmp  eax
    l1:
       invoke printf,offset lpFmt, offset msg1
       jmp    exit
```

```
l2:
   invoke printf,offset lpFmt, offset msg2
   jmp    exit
 l3:
   invoke printf,offset lpFmt, offset msg3
exit:
   invoke ExitProcess, 0
main  endp
end
```

在数据段中有“ptable dd l1, l2, l3”，ptable 中的内容为各个标号的地址，代表了程序的某个位置。在指令地址表中可以出现标号，也可以出现子程序的名字（函数名）。在 C 语言程序编译优化中，也经常使用该方案，构造地址列表，从表中取得转移的目的地址。

6.5.2 switch 语句的编译

在 C 语言程序中，经常使用 switch 语句。下面给出了一个例子，通过例子的反汇编，不难理解 switch 语句的处理过程。

```
#include <stdio.h>
int main(int argc, char* argv[])
{
        int  x = 3;
        int  y = -1;
        int  z;
        int  i = 1;
        char c;
        c = getch();
        switch (c) {
            case '+':
            case 'a':     // 用 字符’a’来表示‘+’
                z= x+y;
                break;
            case '-':
            case 's':    // 用 字符’s’来表示‘-’
                z= x-y;
                break;
            default:
                z=0;
       }
    printf(" %d %c %d = %d \n", x,c,y,z);
    return 0;
```

```
}
```

在 case 语句结束处，有 break 语句，它产生了跳转到 switch 语句结束处的 jmp 语句。如果漏写了 break,则会在该分支执行完后，继续执行下面的分支上的语句。

```
 switch (c) {
00C21855 0F BE 45 CB          movsx     eax,byte ptr [c]
00C21859 89 85 00 FF FF FF    mov       dword ptr [ebp-100h],eax
00C2185F 8B 8D 00 FF FF FF    mov       ecx,dword ptr [ebp-100h]
00C21865 83 E9 2B             sub       ecx,2Bh
00C21868 89 8D 00 FF FF FF    mov       dword ptr [ebp-100h],ecx
00C2186E 83 BD 00 FF FF FF 48 cmp       dword ptr [ebp-100h],48h
00C21875 77 2A                ja        $LN5+0Bh (0C218A1h)
00C21877 8B 95 00 FF FF FF    mov       edx,dword ptr [ebp-100h]
00C2187D 0F B6 82 E8 18 C2 00 movzx     eax,byte ptr [edx+0C218E8h]
00C21884 FF 24 85 DC 18 C2 00 jmp       dword ptr [eax*4+0C218DCh]
 case '+':
 case 'a':
     z= x+y;
00C2188B 8B 45 F8             mov       eax,dword ptr [x]
00C2188E 03 45 EC             add       eax,dword ptr [y]
00C21891 89 45 E0             mov       dword ptr [z],eax
     break;
00C21894 EB 12                jmp       $LN5+12h (0C218A8h)
 case '-':
 case 's':
     z= x-y;
00C21896 8B 45 F8             mov       eax,dword ptr [x]
00C21899 2B 45 EC             sub       eax,dword ptr [y]
00C2189C 89 45 E0             mov       dword ptr [z],eax
     break;
00C2189F EB 07                jmp       $LN5+12h (0C218A8h)
 default:
     z=0;
00C218A1 C7 45 E0 00 00 00 00 mov       dword ptr [z],0
 }
00C218A8 ……
```

+2Bh
-2Dh
0x64h
73h
73-2B=48h

内存 1

地址: 0x00C218DC　　列: 自动

```
0x00C218DC  8b 18 c2 00 96 18 c2 00 a1 18 c2 00
0x00C218E8  00 02 01 02 02 02 02 02 02 02 02 02
0x00C218F4  02 02 02 02 02 02 02 02 02 02 02 02
```

图 6.4 switch 语句的调试中的内存窗口

在反汇编程序中，我们没有看到跳转到各分支的语句，但可以看到

```
00C21884 jmp    dword ptr [eax*4+0C218DCh]
```

在“内存”窗口中，观察地址 0C218DCh 中的内容，可以看到内存单元中，依次存放了 3 个分支的入口地址，即 00C2188B、00C21896、00C218A1。(eax)应该是该表中的第几项，通过间接无条件跳转到目的地。 有兴趣的读者可以仔细研究一下编译器对程序进行翻译时采用的技巧。

注意，在 VS2019 等开发环境中，可以在代码生成时设置代码优化的参数，不同的优化参数生成的结果是不同。通过研究代码优化的结果，可以学到不少巧妙编写汇编程序的方法。

6．6 条件控制流伪指令

用汇编语言来写分支程序，要注意选择正确的转移指令、各个分支的出口和汇合、给分支入口和出口语句恰当的标号等问题。在分支中又含有分支时，还要注意分支的摆放顺序，避免随意跳转，用模块化的方法写程序，提高程序的可读性。

与 C 语言程序相比，由于缺乏一些“显眼”的符号，如 if、else、{、}，使得汇编语言编写的分支程序比用 C 语言写的程序的可读性要差。但是从机器语言的角度来看，分支程序的核心指令就是转移指令，包括转入分支的转移指令和从分支跳出的转移指令。C 语言程序要通过编译器将其转换为机器语言程序。为了简化汇编源程序的编写，很多汇编编译器也支持了条件流控制伪指令。当然，对初学者我们并不推荐使用条件控制流伪指令。初学者的重点要放在选择合适的机器指令、安排分支出口等方面。

条件判断伪指令的格式如下。

```
.if   条件表达式 1
      语句序列 1
[[.elseif 条件表达式 2      ；.elseif 语句可以有 0 个多个
      语句序列 2]……
[.else                      ；.else 只能为 0、1 个
      语句序列 3]]
.endif                      ；分支判断结束标志，必须与.if 配对使用
```

上述语句中,中括号“[……]”中的内容是可选的。它可以是如下几种形式。

（1）只有一个条件成立的分支

```
.if   条件表达式 1
      语句序列 1       ；条件表达式 1 为真时执行
.endif
```

（2） 条件成立与不成立的两个分支

```
.if   条件表达式 1
      语句序列 1       ；条件表达式 1 为真时执行
.else
      语句序列 2       ；条件表达式 1 为假时执行
.endif
```

（3） 多分支

```
.if  条件表达式 1
      语句序列 1
.elseif 条件表达式 2
      语句序列 2     ；条件表达式 1 为假且条件表达式 2 为真时执行
.else
      语句序列 3     ；条件表达式 1、2 皆为假时执行
.endif
```

注意，在语句序列中可以含有 IF 语句，即含有“.if …….endif”,形成 if 语句的嵌套。

条件表达式一般由关系运算和逻辑运算组成有意义的式子。关系运算包括：相等“==”、大于“>”、大于等于“>=”、不等“!=”、小于“<”、小于等于“<=”。逻辑运算包括：逻辑非“!”、逻辑与“&&”、逻辑或“||”。除此之外，条件表达式中还可以有如下条件：

位测试 “表达式 & 位号”
进位标志置位“CARRY?”
符号标志置位“SIGN?”
零标志置位“ZERO?”
溢出标志置位“OVERFLOW?”
奇偶标志置位“PARITY?”

对于有多个式子组成的表达式,有一个运算优先级的问题。可以在式子上加小括号“(……)”,其内的式子会被优先计算。

下面用条件流控制伪指令来实现例 6.1 中的分支功能，程序如下。

```
.686P
.model flat, stdcall
  ExitProcess proto stdcall :dword
  printf      proto C :ptr sbyte, :VARARG
  includelib  kernel32.lib
  includelib  libcmt.lib
  includelib  legacy_stdio_definitions.lib
.data
  lpFmt db  "x=%d  y=%d  r=%d",0ah, 0dh, 0
  x  sdword  -5      ;注意，不能用 DD 来定义
  y  sdword  6       ;DD 定义的变量会认为是一个无符号类型的变量
                     ;两种定义会导致下面的 if x>=0 翻译结果不同
  r  dd  0
.stack 200
.code
 main  proc  c
   .if x>=0
       .if  y>=0
           mov  r, 1
```

```
        .else
            mov  r, 0
        .endif
    .else
        .if  y>=0
            mov  r, 0
        .else
            mov  r, -1
        .endif
    .endif
    invoke printf,offset lpFmt,x,y,r
    invoke ExitProcess, 0
main  endp
end
```

对这一程序，还可以进一步优化，改写后的程序的前一部分如下：

```
    mov r, 0
    .if x>=0 && y>=0
            mov  r, 1
    .endif
    .if x<0 && y<0
        mov  r, -1
    .endif
```

条件控制流伪指令的语法格式与 C 语言是很相似的。条件表达式与 C 语言相同。下面给出了上段改写后的程序的反汇编结果。

```
    start:
    00592030  mov         dword ptr [r (0595019h)],0
    0059203A  cmp         dword ptr [x (0595011h)],0
    00592041  jl          _start+26h (0592056h)
    00592043  cmp         dword ptr [y (0595015h)],0
    0059204A  jl          _start+26h (0592056h)
    0059204C  mov         dword ptr [r (0595019h)],1
    @C0001:
    00592056  cmp         dword ptr [x (0595011h)],0
    0059205D  jge         _start+42h (0592072h)
    0059205F  cmp         dword ptr [y (0595015h)],0
    00592066  jge         _start+42h (0592072h)
    00592068  mov         dword ptr [r (0595019h)],0FFFFFFFFh
    @C0004:
    00592072  ……
```

在用汇编语言写程序时，要正确地选择转移指令。例如,对于关系运算 x>0，当 x 是一个有

符号数时，应采用 jg，当 x>0 成立时，jg 的条件成立，发生转移；当 x 是一个无符号数时，应采用 ja，当 x>0 成立时，ja 的条件成立，发生转移。

对于条件控制流伪指令中出现的变量，编译器会根据变量定义的类型选择对应的转移指令。对于 sbyte、sword、sdword、sqword 定义的变量是有符号类型，而用 db、byte、dw、word、dd、dword、dq、qword 定义的变量是无符号类型。在程序中的大小比较就相应的是无符号数之间的比较或者有符号数之间的比较。有兴趣的读者可以比较通过观察反汇编代码比较它们的差别。

习题 6

6.1 简单条件转移指令有哪些？各自的转移条件是什么？

6.2 无符号数条件转移指令有哪些？有符号数条件转移指令又有哪些？

6.3 设(ax)=0D000H, (bx)= 2000H

执行 cmp ax, bx

ja l1

请问程序是否会转移到 l1 处？若将 ja 换成 jg，结果又如何？

若将 ja 换成 js，结果又如何？

6.4 请写出实现下面功能的汇编代码（不要使用条件流控制伪指令）。x、y、z 都是有符号双字类型的变量。

```
if (z>y)
    z=x;
else z=y;
```

6.5 对于 if-else 语句，在 then 分支结束处，应该有什么语句？若漏写了该语句，程序运行结果会如何？

6.6 将下列程序段简化（x、y 为字变量，p1、p2、p3、p4、p5 为标号）：

```
    mov   ax, x
    cmp   ax, y
    jc    p1
    cmp   ax, y
    jo    p2
    cmp   ax, y
    je    p3
    cmp   ax, y
    jns   p4
p3: add   ax, Y
    jc    p5
```

6.7 阅读下列程序段，并指出程序执行到 exit 时 ecx 的内容。

```
    cmp  eax, ebx
    jg   p1
    je   p2
    mov  ecx, -1
    jmp  exit
p1: mov  ecx, 1
    jmp  exit
p2:  mov  ecx, 0
exit:    ……
```

6.8 分支转移指令的机器编码中，是如何表示转移目的地地址的？

上机实践 6

6.1 设在一个缓冲区中存放了 N 个有符号双字类型的数据，编写一个程序，将该缓冲区中所有的负数依次拷贝到另一个缓冲区中。

6.2 编写一个程序，统计一个字符串（包括大、小写字母、数字、其他符号）中各个字母出现的次数。字母不区分大小写。最后显示在字符串中出现过的字母及其出现的次数。

例如：buf db ‘Good’,0

显示 d or D : 1

g or G : 1

o or O : 2

6.3 阅读下面的程序，指出程序的运行结果。

试分析条件表达式 x<y 与 x-y<0 是否等价。

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    short x =  100;
    short y = -32700;
    short z;
    if (x < y)
        printf("condition1:  %d < %d \n", x, y);
    z = x - y;
    if (z < 0)
        printf("condition2:  %d < %d \n", x, y);
    return 0;
}
```

6.4 阅读下面的程序，指出程序的运行结果。

```
#include <stdio.h>
int sum(int a[], unsigned length)
{
     int i;
     int result = 0;
     for (i = 0; i <= length - 1; i++)
         result += a[i];
     return result;
}
int main(int argc, char* argv[])
{
     int  a[5] = { 1, 2, 3, 4, 5 };
     int  z;
     z = sum(a, 0);
     printf("sum : %d \n", z);
     return 0;
}
```

试分析条件表达式 i<length 与 i<=length-1 是否等价；试用所学知识解释程序运行时出现的现象。