第 5 章 汇编语言中的控制结构

控制结构是程序中的基本构件，用于控制程序执行的流程。顺序结构按顺序执行指令，选择结构根据条件选择执行不同的代码块，循环结构则重复执行某段代码直至满足特定条件。这些结构共同决定了程序的逻辑和运行效率。本章将介绍关于顺序结构、选择结构、循环结构的相关内容。

5.1 顺序结构

顺序结构是程序设计的基本结构之一，它表示程序中指令的线性执行顺序。在顺序结构中，指令会按照书写顺序逐一执行，确保了有序性并简化了控制流程。作为默认结构，顺序结构因其简单明了而易于理解和实现，减少了控制复杂性，使程序逻辑清晰，便于调试和维护。这一基础结构为实现更复杂的控制结构奠定了可靠基础，具体表现为逐条执行代码，如图 5-1 所示。

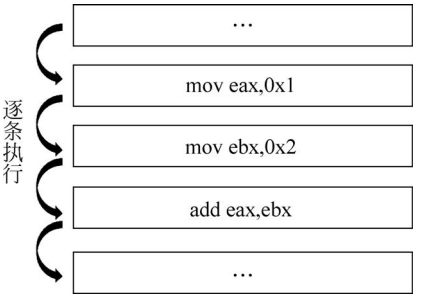

图 5-1 顺序结构代码执行的基本原理

在汇编语言中，每条指令都有特定地址，程序从起始地址开始，依次读取和执行指令，直到程序执行结束。接下来，本书将以在终端窗口中输出"Hello Hacker!"字符串程序为例来阐述顺序结构，代码如下：

```
//ch05/hellohacker.asm
1  section .data
2      msg db 'Hello Hacker!', 0xA

3  global _start
4  section .text
5  _start:
6      mov eax, 4
7      mov ebx, 1
8      mov ecx, msg
```

```
9       mov edx, 13
10      int 0x80
11      mov eax, 1
12      xor ebx, ebx
13      int 0x80
```

第 1 行和第 2 行代码定义数据段，用于存放程序的数据，它包括一个字符串 msg，其内容为"Hello Hacker!"和一个换行符 0xA。第 3 行代码的 global _start 将_start 标签声明为全局符号，告诉链接器程序的入口点。第 4 行代码定义代码段，用于存放程序的指令。第 5 行代码的_start:是程序的入口点，程序将从这里开始执行。第 6～9 行代码将使用 sys_write 的系统调用向终端窗口中输出"Hello Hacker!"字符串和一个换行符。第 11～13 行代码将执行 sys_exit 系统调用来结束程序。值得注意的是，第 12 行代码使用异或逻辑运算来置零寄存器 EBX。如果在终端窗口中成功地执行了 hellohacker 程序，则会输出"Hello Hacker!"字符串信息，如图 5-2 所示。

图 5-2　成功执行 hellohacker 程序

虽然直接执行程序可以查看运行结果，但无法深入剖析程序内部的流程控制结构，因此笔者经常使用 gdp 调试器逐条查看并分析程序的运行原理。通过 gdb 工具加载 hellohacker 程序，执行 disassemble 命令可以实现对程序的反编译，并输出相应的汇编代码，如图 5-3 所示。

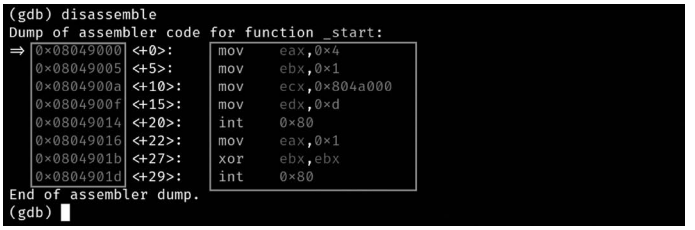

图 5-3　使用 gdb 查看 hellohacker 程序的汇编代码

细心的读者可能会注意到在结果信息的左侧会输出汇编代码对应的内容地址，例如，代码 mov eax,0x4 相应的内存地址为 0x0804900。最终，hellohacker 程序会根据地址的大小来逐条执行对应的汇编代码，如图 5-4 所示。

在顺序结构中，代码按照从上到下的顺序逐行执行。若出现结束或跳转指令，执行顺序则会被打破，导致程序结束或重新开始执行。接下来，本书将深入探讨汇编语言中构建选择结构的结束、比较和跳转指令。

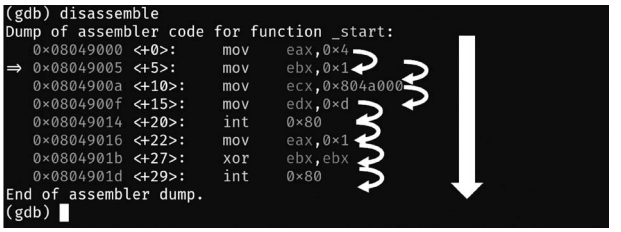

图 5-4　按照顺序结构逐条执行汇编代码

5.2　选择结构

计算机中的选择结构是一种控制程序执行流程的机制，它根据特定条件决定执行不同的代码路径，其主要作用是引导程序流向，使其能根据不同的输入或状态执行相应的逻辑。例如，在选择结构中，当条件判断为真时会执行操作 1 对应的代码；如果判断为假，则执行操作 2 对应的代码，如图 5-5 所示。

当然，选择结构可以嵌套其他选择结构，以实现更复杂的判断逻辑。例如，在操作 2 中嵌套额外的判断条件，当条件为真时执行操作 2，而当条件为假时则执行操作 3，如图 5-6 所示。

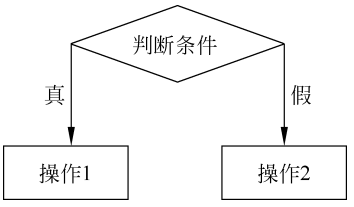

图 5-5　选择结构的基本原理

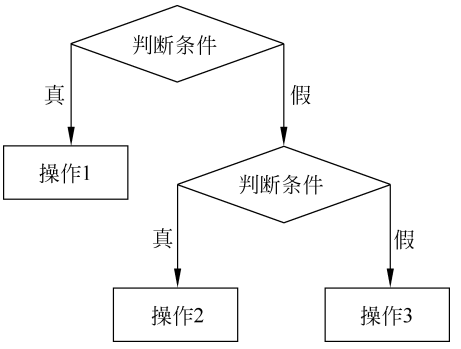

图 5-6　通过嵌套条件判断实现复杂逻辑

注意：操作 1、2、3 中的代码将按顺序逐条执行。如果这些操作中包含结束或跳转指令，则程序将终止或跳转到指定位置继续执行。

5.2.1　结束指令

系统调用提供了一种访问系统资源和服务的方式，例如文件操作、进程管理和网络通信。本质上，系统调用是 Linux 内核提供的一组函数，这些函数只能在内核空间中运行，并实现特定功能。用户程序可以通过 int 0x80 中断指令来执行这些系统调用。

当用户程序执行系统调用时，首先将系统调用编号存入寄存器 EAX，并将参数值传递到其他寄存器。随后，程序通过中断机制切换到内核空间，内核根据寄存器 EAX 中的值识

别出系统调用的编号，从而找到相应的系统调用函数并执行。当系统调用成功完成后，程序会切换回用户空间，并将返回值传递给用户程序。例如，在 Linux 系统中执行 sys_exit 系统调用的基本原理如图 5-7 所示。

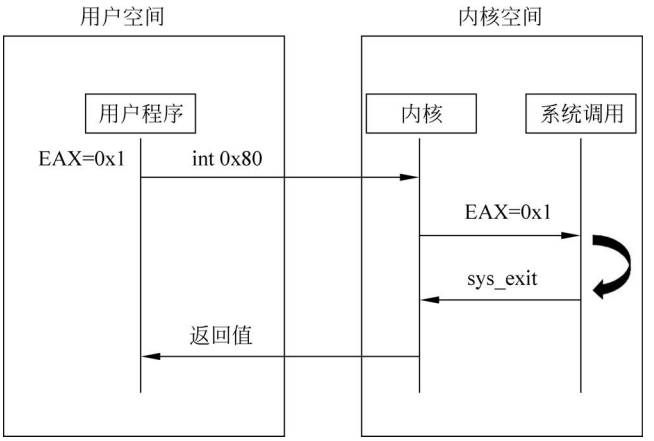

图 5-7　Linux 执行 sys_exit 系统调用的基本原理

在 Linux x86 汇编语言中，结束程序的常用指令是 mov 和 int 指令组合，通常使用 sys_exit 系统调用来优雅地终止程序，代码如下：

```
1  mov eax, 1
2  xor ebx, ebx
3  int 0x80
```

第 1 行代码将数值 1 传送到寄存器 EAX 中，在 Linux x86 操作系统中，系统调用号 1 表示 sys_exit，用于终止程序执行。第 2 行代码通过异或运算将寄存器 EBX 的值设置为 0，表示程序的返回值。第 3 行代码调用中断机制以触发系统调用。Linux 操作系统的内核提供了多种功能的系统调用，每个系统调用都有唯一的编号。在 x86 架构的系统中，系统调用号存储在/usr/include/x86_64-linux-gnu/asm/unistd_32.h 文件中。该文件是一个文本文件，可以使用文本编辑器或 Linux 内置的命令查看其内容，例如，使用 Linux 内置的 cat 命令来查看 unistd_32.h 头文件的内容，如图 5-8 所示。

```
└─$ cat /usr/include/x86_64-linux-gnu/asm/unistd_32.h
#ifndef _ASM_UNISTD_32_H
#define _ASM_UNISTD_32_H

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
```

图 5-8　查看系统调用文件 unistd_32.h 的内容

在不同架构的 Linux 操作系统中，尽管它们共享相同的内核，但系统调用号会被保存到不同的文件中，例如，x64 架构的系统调用号保存在 unistd_64.h 文件中。此外，不同版本的

Linux 内核也可能具有不同的系统调用号，因此在执行系统调用时，必须查看相应的系统调用号以确保正确地使用了这些调用。

虽然在 unistd_32.h 文件中保存着系统调用号，但是它并没有关于该系统调用的使用方法，因此笔者通常会从 unistd_32.h 文件中查看系统调用的名称，再使用 man 命令来查阅系统调用的帮助手册，从而正确地使用它们。例如，使用 man 命令组合参数 2 来查看 exit 系统调用的帮助信息，如图 5-9 所示。

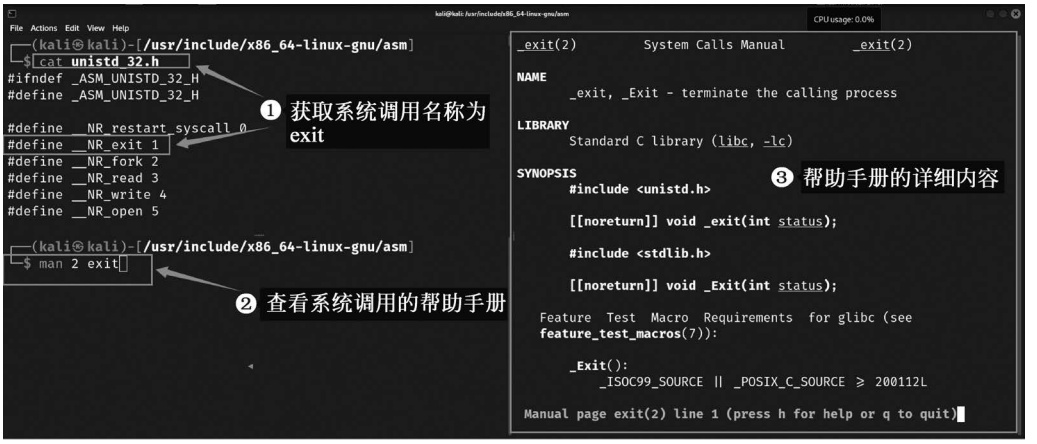

图 5-9　查看系统调用 exit 的帮助手册

除了上述方法，还可以尝试使用 Shellnoob 等工具来自动获取系统调用号。Shellnoob 是一个基于 Python 开发的工具，旨在简化编写 shellcode 的过程。例如，使用 Shellnoob 工具查看 exit 的系统调用号，如图 5-10 所示。

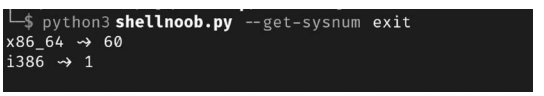

图 5-10　使用 Shellnoob 工具查看系统调用号

如果在 Kali Linux 的终端中成功地运行了 Shellnoob 工具，则可以查看 x86 架构的 exit 系统调用号为 1，而 x64 架构的 exit 系统调用号为 60。

注意：x86_64 表示 64 位操作系统，而 i386 表示 32 位操作系统。

5.2.2　比较指令

CMP 指令是汇编语言中的一种比较指令，主要用于比较两个操作数。它通过执行减法来比较两个操作数，仅影响标志寄存器 EFLAGS 的标志位，但并不实际存储结果。如果两个操作数相等，则寄存器 EFLAGS 的标志位 ZF 被设置为 1。如果发生借位，即第 1 个操作数小于第 2 个操作数，则寄存器 EFLAGS 的标志位 CF 被设置为 1。如果第 1 个数大于第 2 个数，则寄存器 EFLAGS 的标志位 ZF 和 CF 都不会被设置。这样，CMP 指令可以用于后续的条件跳转，帮助控制程序执行路径。下面以 cmp.asm 文件为例阐述 CMP 指令的使用

方法，代码如下：

```
//ch05/cmp.asm
1 global _start
2 section .text
3 _start:
4     mov eax, 5
5     mov ebx, 5
6     cmp eax, ebx
7     mov eax, 3
8     cmp eax, ebx
9     cmp ebx, eax
10    mov eax, 1
11    xor ebx, ebx
12    int 0x80
```

第 4 行和第 5 行代码实现了将十进制数 5 传送给寄存器 EAX 和 EBX。第 6 行代码使用 CMP 指令比较两个寄存器保存的值。如果使用 gdp 调试器加载 cmp 可执行程序并执行到第 6 行代码，则可使用 info registers eflags 命令来查看标志寄存器中的标志位。在输出的结果信息中，仅会包含设置的标志位名称，例如，如果 ZF 被设置为 1，则只会输出 ZF 的名称，如图 5-11 所示。

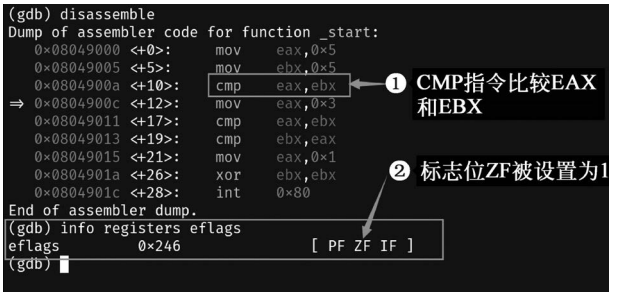

图 5-11　标志寄存器 EFLAGS 的 ZF 标志位被设置为 1

第 7 行代码会将十进制数 3 传送到寄存器 EAX 中，此时 EBX 寄存器仍然保存着十进制数 5。第 8 行代码中的 CMP 指令用于比较寄存器 EAX 和 EBX 的值。如果成功地执行了第 8 行代码，则可以通过 gdp 调试器查看标志寄存器 EFLAGS 的 CF 标志位被设置为 1，如图 5-12 所示。

第 9 行代码用于比较寄存器 EBX 和 EAX 的值，此时 EBX 的值为 5，而 EAX 的值为 3。由此可见，EBX 的值大于 EAX 的值。如果执行 cmp ebx，eax 命令，则不会设置 ZF 和 CF 标志位，如图 5-13 所示。

第 10～12 行代码实现了执行系统调用 exit 来正常退出程序。CMP 指令用于比较两个操作数，并设置相应的标志位，而跳转指令则根据这些标志位的状态进行条件跳转，从而实现控制流的灵活性。接下来，本书将介绍关于跳转指令的相关内容。

图 5-12　标志寄存器 EFLAGS 的 CF 标志位被设置为 1

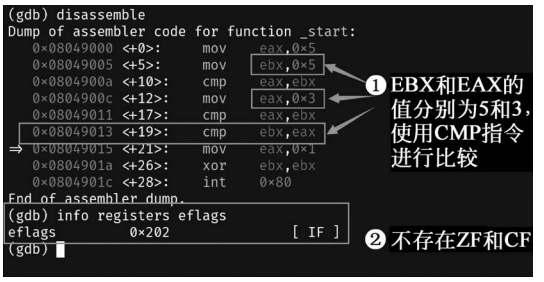

图 5-13　标志寄存器 EFLAGS 未设置 ZF 和 CF 标志位

5.2.3　跳转指令

在 Linux x86 汇编语言中，跳转指令用于控制程序的执行流程，主要分为无条件跳转和条件跳转两类。无条件跳转指令 JMP 能够直接跳转到指定的标签，标签本质上是代码中的一个标识符，用于指向特定位置。使用 JMP 指令，程序可以无条件地转移到所需的位置，从而实现代码的重用或跳过不必要的部分。

条件跳转指令可以据特定条件和标志位的状态，决定是否跳转。常见的条件跳转指令及其触发条件如表 5-1 所示。

表 5-1　常见的条件跳转指令及其触发条件

条件跳转指令	触 发 条 件
JE/JZ	相等跳转指令，当 ZF=1 时跳转
JNE/JNZ	不相等跳转指令，当 ZF=0 时跳转
JG	大于跳转指令，当 ZF = 0 且 SF = OF 时跳转
JL	小于跳转指令，当 SF ≠ OF 时跳转
JGE	大于或等于跳转指令，当 SF = OF 时跳转
JLE	小于或等于跳转指令，当 ZF = 1 或 SF ≠ OF 时跳转

在使用汇编语言时，确实不需要每次都关注 EFLAGS 的具体标志位，而是可以通过指令的条件含义来判断，例如，JNE 表示不等于就跳转，用户只需关注逻辑关系，而不必深入标志位的状态。这种方式能简化思维过程，更专注于程序逻辑的实现。接下来，本书将以判断用户输入的数字是否为 1 为例来阐述跳转指令，代码如下：

```
//ch05/jmp.asm
1  section .data
2      msg1 db 'Input is 1', 0
3      msg2 db 'Input is not 1', 0
4      prompt db 'Enter a number: ', 0

5  section .bss
6      input resb 1

7  global _start
8  section .text
9  _start:
10   mov eax, 4
11   mov ebx, 1
12   mov ecx, prompt
13   mov edx, 16
14   int 0x80

15   mov eax, 3
16   mov ebx, 0
17   mov ecx, input
18   mov edx, 1
19   int 0x80

20   sub byte [input], '0'
21   cmp byte [input], 1
22   jne not_one

23 is_one:
24   mov eax, 4
25   mov ebx, 1
26   mov ecx, msg1
27   mov edx, 15
28   int 0x80
29   jmp end_program

30 not_one:
31   mov eax, 4
32   mov ebx, 1
33   mov ecx, msg2
34   mov edx, 16
35   int 0x80
36   jmp end_program

37 end_program:
38   mov eax, 1
39   xor ebx, ebx
40   int 0x80
```

第 1～4 行代码在数据段中声明了变量 msg1、msg2、prompt，并对它们分别赋值相应字符串，同时以零结尾。第 5 行和第 6 行代码在 BSS 段声明了一个 input 变量，并为其保留一字节的空间，用于存储用户输入。第 7～9 行代码中声明了_start 作为程序入口点，并定义代码段。第 10～14 行代码通过执行 sys_write 的系统调用来向终端中输出 prompt 变量保存的字符串提示信息。如果使用 gdp 调试器将程序运行到第 10 行代码，则会在终端窗口中输出 Enter a number:的提示信息，如图 5-14 所示。

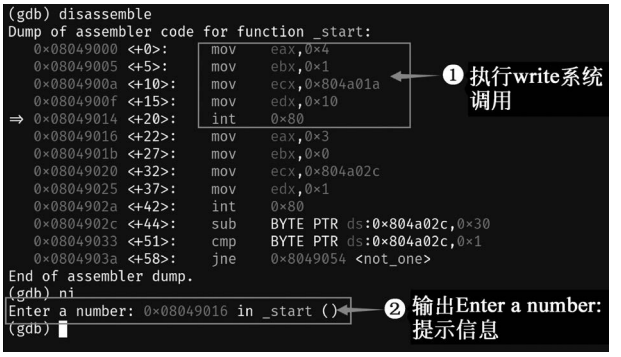

图 5-14　执行 write 系统调用

第 15～19 行代码会运行 sys_read 系统调用来等待用户输入的数据，并将其保存到 input 变量中。由于默认输入的值都是 ASCII 码，因此必须将其转换为对应的整数才能进行比较。如果使用 gdp 调试器将程序运行到第 20 行代码，则会等待用户输入数据，如图 5-15 所示。

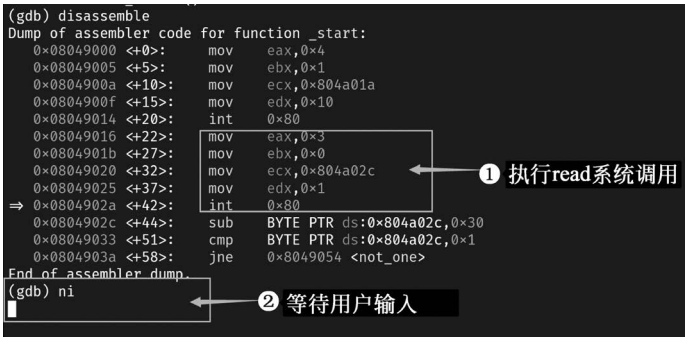

图 5-15　执行 read 系统调用

接下来，输入的数据会被保存到 0x804a02c 内存地址对应的空间中，例如，在 gdp 调试器中输入数值 1，并按 Enter 键来确认输入的数据。最后，通过执行 x/1db 命令来查看该地址空间的内容，如图 5-16 所示。

细心的读者可能会注意到内存地址 0x804a02c 保存的值为 49，它是字符‘1’对应的 ASCII 码值。在汇编语言中，字符的 ASCII 值与它们对应的整数值之间存在一定的差距，例如，字符‘0’的 ASCII 值是 48，而字符‘1’的 ASCII 值是 49，以此类推，因此如果想将一个

```
(gdb) disassemble
Dump of assembler code for function _start:
   0×08049000 <+0>:     mov    eax,0×4
   0×08049005 <+5>:     mov    ebx,0×1
   0×0804900a <+10>:    mov    ecx,0×804a01a
   0×0804900f <+15>:    mov    edx,0×10
   0×08049014 <+20>:    int    0×80
   0×08049016 <+22>:    mov    eax,0×3
   0×0804901b <+27>:    mov    ebx,0×0
   0×08049020 <+32>:    mov    ecx,0×804a02c
   0×08049025 <+37>:    mov    edx,0×1
   0×0804902a <+42>:    int    0×80
⇒ 0×0804902c <+44>:    sub    BYTE PTR ds:0×804a02c,0×30
   0×08049033 <+51>:    cmp    BYTE PTR ds:0×804a02c,0×1
   0×0804903a <+58>:    jne    0×8049054 <not_one>
End of assembler dump.
(gdb) x/1db 0×804a02c
0×804a02c:      49
(gdb)
```

图 5-16　使用 gdb 查看内存地址中的值

字符形式的数字转换为它的整数形式，则需要减去‘0’的 ASCII 值。通过 SUB 指令对输入的 ASCII 值减去 48，就可以得到该字符对应的整数。字符‘0’对应的 ASCII 码值为 48，与此值相应的十六进制数是 0x30。

第 20 行代码通过 SUB 指令对 input 变量执行减‘0’操作，从而实现了将输入的 ASCII 值转换为整数。如果使用 gdp 调试器执行这行代码，则可以使用 x/1db 命令来查看该内存地址保存的数值，如图 5-17 所示。

```
(gdb) disassemble
Dump of assembler code for function _start:
   0×08049000 <+0>:     mov    eax,0×4
   0×08049005 <+5>:     mov    ebx,0×1
   0×0804900a <+10>:    mov    ecx,0×804a01a
   0×0804900f <+15>:    mov    edx,0×10
   0×08049014 <+20>:    int    0×80
   0×08049016 <+22>:    mov    eax,0×3
   0×0804901b <+27>:    mov    ebx,0×0
   0×08049020 <+32>:    mov    ecx,0×804a02c
   0×08049025 <+37>:    mov    edx,0×1
   0×0804902a <+42>:    int    0×80
   0×0804902c <+44>:    sub    BYTE PTR ds:0×804a02c,0×30
⇒ 0×08049033 <+51>:    cmp    BYTE PTR ds:0×804a02c,0×1
   0×0804903a <+58>:    jne    0×8049054 <not_one>
End of assembler dump.
(gdb) x/1db 0×804a02c
0×804a02c:      1
(gdb)
```

❶ 执行SUB指令代码

❷ 查看内存0x804a02c的值为1

图 5-17　使用 gdb 查看内存保存的值

第 21 行和第 22 行代码会组合 CMP 和 JNE 指令来实现判断输入的数值是否为 1。如果输入的数值是 1，则会跳转到标签 is_one 的位置，如图 5-18 所示。

```
⇒ 0×0804903a <+58>:    jne    0×8049054 <not_one>
End of assembler dump.
(gdb) ni
0×0804903c in is_one ()
(gdb) disassemble
Dump of assembler code for function is_one:
⇒ 0×0804903c <+0>:     mov    eax,0×4
   0×08049041 <+5>:     mov    ebx,0×1
   0×08049046 <+10>:    mov    ecx,0×804a000
   0×0804904b <+15>:    mov    edx,0×f
   0×08049050 <+20>:    int    0×80
   0×08049052 <+22>:    jmp    0×804906c <end_program>
End of assembler dump.
(gdb)
```

❶ 跳转到is_one标签

图 5-18　输入数值 1 跳转到 is_one 标签

如果输入的数值不为 1，则会执行 not_one 标签对应的代码，如图 5-19 所示。

图 5-19　输入数值 2 跳转到 not_one 标签

第 23～29 行代码为 is_one 标签的代码块，它会向终端输出 msg1 变量的值，并通过执行 JMP 指令跳转至标签为 end_program 的位置，如图 5-20 所示。

图 5-20　输出 Input is 1 的提示信息

第 30～36 行代码是 not_one 标签的代码块，它能够实现向终端输出 msg2 变量的值，并最终跳转到标签 end_program 的地址，如图 5-21 所示。

图 5-21　输出 Input is not 1 的提示信息

最终，程序会执行第 37 至 40 行代码以实现正常退出。感兴趣的读者可以尝试使用其他跳转指令来创建更复杂的程序。此外，跳转指令也可用于实现代码的重复执行。接下来，本书将探讨如何使用跳转指令构建循环结构，并介绍如何利用 LOOP 指令来简化循环的实现。

5.3　循环结构

循环结构是程序设计中的一种控制结构，用于重复执行代码，直至满足特定条件。它允许程序根据输入或状态动态地决定是否继续执行某部分代码。循环结构可分为计数循环、

条件循环和无限循环 3 种类型。尽管它们适用于不同的场景，但都具有相似的组成部分，包括循环计数器、初始值、计数器更新和终止条件。

5.3.1　计数循环

计数循环是一种根据指定次数重复执行代码的结构，通常依赖一个循环计数器来跟踪当前的迭代次数。循环计数器通常使用变量来存储当前的计数值，以控制循环的执行次数。在循环开始前，计数器被设置为初始值，表示将执行的次数。在每次执行循环体时，计数器会更新。当计数器达到特定值时，跳出循环，否则继续进入循环，如图 5-22 所示。

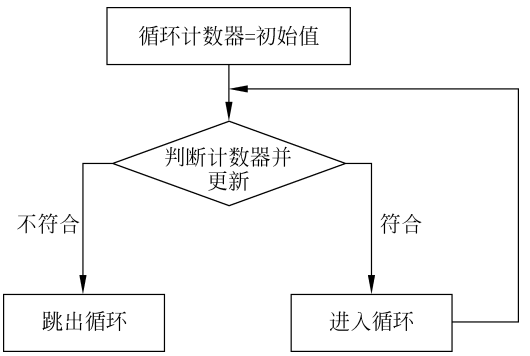

图 5-22　计数循环结构的基本原理

计数循环通过简单的计数器管理循环次数，它适合于需要执行固定次数的场景，例如，在终端窗口中输出 5 次"Hello Hacker!"提示信息，代码如下：

```
//ch05/loop1.asm
1  section .data
2     message db 'Hello Hacker!', 0xA
3     msg_length equ $ - message

4  global _start
5  section .text
6  _start:
7     mov ecx, 5

8  .loop_start:
9     push ecx
10    mov eax, 4
11    mov ebx, 1
12    mov ecx, message
13    mov edx, msg_length
14    int 0x80
15    pop ecx

16    dec ecx
```

```
17    jnz .loop_start

18    mov eax, 1
19    xor ebx, ebx
20    int 0x80
```

第 1～3 行代码表示先在数据段中定义变量 message 并初始化为"Hello Hacker!"，然后使用 0xA 换行符作为结尾。同时，定义常量 msg_length，用来保存 message 变量值的长度。在汇编语言中，符号“$”通常表示当前地址。如果使用符号“$”减去 message 变量名，则表示变量 message 保存的字符串长度，如图 5-23 所示。

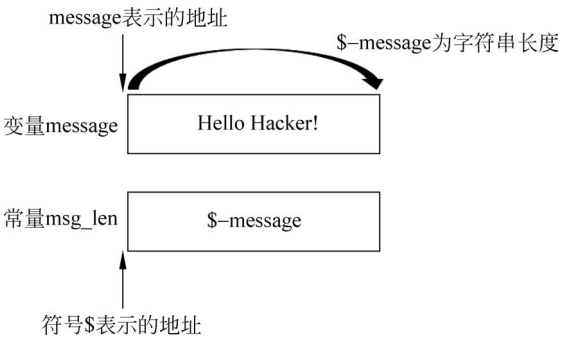

图 5-23　通过 $ －message 获取字符串长度的原理

第 4～7 行代码将_start 声明为全局符号，使其可以被链接器识别为程序的入口点。同时，在代码段的_start 入口标签中，将数字 5 传送给寄存器 EAX 作为循环计数器的初始值。如果使用 gdp 调试器将程序执行到第 8 行代码，则可以通过运行 info registers ecx 或 print $ecx 命令来查看寄存器 ECX 的值，如图 5-24 所示。

图 5-24　查看寄存器 ECX 的初始值

第 8～15 行代码定义了一个标签 loop_start，通过执行 sys_write 系统调用将"Hello Hacker!"字符串输出到终端中。由于寄存器 ECX 同时用作循环变量和 sys_write 系统调用的第 3 个参数，为了避免冲突，可以使用 PUSH 指令保存其值，随后用 POP 指令恢复。这样可以确保在调用过程中不丢失寄存器 ECX 的原有值。PUSH 和 POP 指令通过对栈空间的操作来压入和弹出数据，从而实现保存和恢复数据的功能。在后续章节中会对 PUSH 和 POP 指令进行详细介绍。在循环开始之前，使用 push ecx 将寄存器 ECX 的值保存到栈空间中。在完成循环后，执行 pop ecx 命令将栈空间中的值传送给寄存器 ECX，从而恢复它的原始值。

在循环结构中，mov eax,4 用于设定系统调用 sys_write 的编号，mov ebx,1 将文件描述符设置为标准输出，mov ecx,message 将 message 的地址加载到寄存器 ecx 中，mov edx,msg_length 指定输出数据的长度。最终，通过 int 0x80 指令来触发 sys_write 系统调用，并向终端中输出"Hello Hacker!"字符串。如果使用 gdp 调试器将程序暂停到第 14 行代码，则执行 1 次循环结构中的代码，并输出 1 行字符串信息，如图 5-25 所示。

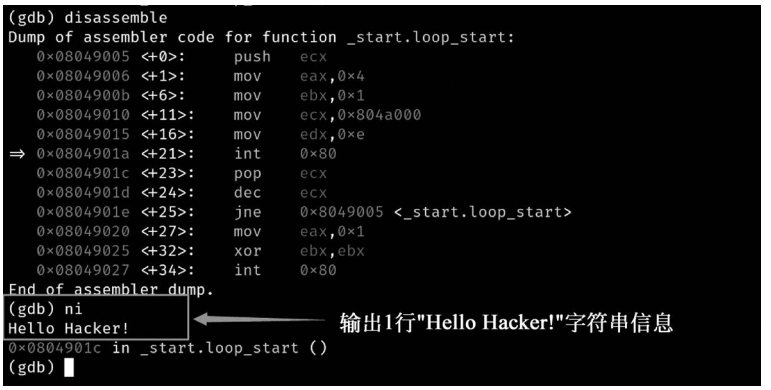

图 5-25　使用 gdb 调试程序并输出 1 行字符串信息

第 16 行和第 17 行代码执行 DEC 指令以将寄存器 ECX 的值减 1，并通过 JNZ 指令判断其值是否为 0。如果 ECX 的值不为 0，则跳转到标签. loop_start，继续执行循环，如图 5-26 所示。

细心的读者可能会注意到，gdp 调试器将 loop1 程序中的 JNZ 指令识别为 JNE 指令。尽管 JNZ 和 JNE 都是条件跳转指令，但它们的使用场景有所不同。JNZ 通常用于 DEC 指令之后，以判断寄存器的值是否非零，而 JNE 则常用于 CMP 指令之后，判断两个值是否不相等，以决定是否跳转到不等于的条件。由此可见，JNZ 更加专注于检查值是否为 0，而 JNE 则基于比较结果。虽然 gdp 调试器将 JNZ 反编译为 JNE，但是并不会影响程序的正常执行。

如果使用 gdp 调试器执行 5 次循环结构，则寄存器 ECX 的值会被设置为 0，并跳出循环，如图 5-27 所示。

第 18～20 行代码表示执行 sys_exit 系统调用并正常退出程序。虽然组合 DEC 和 JNZ 指令能够实现计数循环，但是汇编语言为了简化程序代码提供了 LOOP 指令，此指令能够替代这种组合方式来实现循环，代码如下：

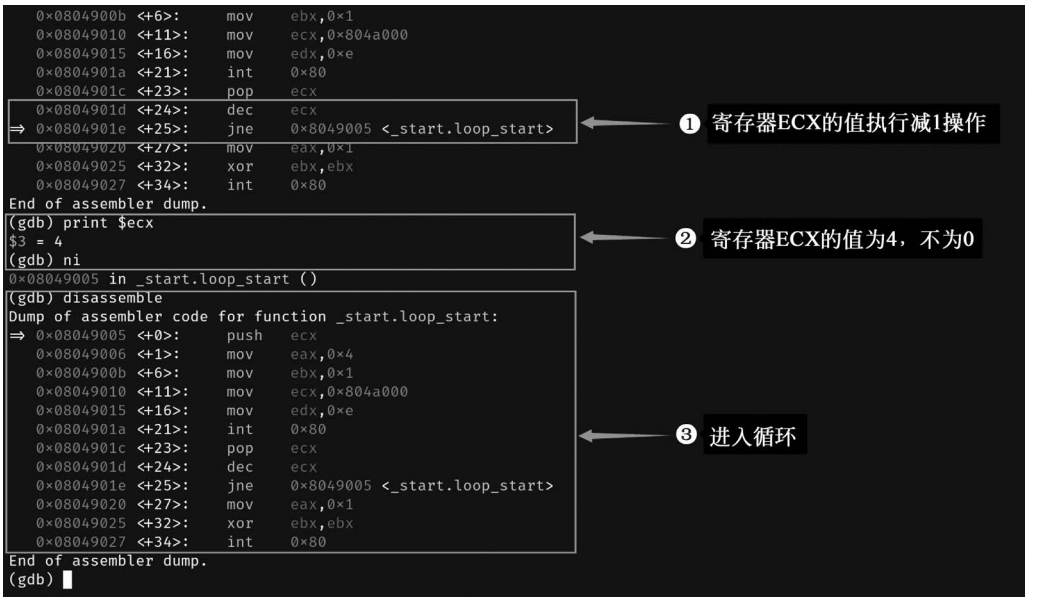

图 5-26　使用 gdb 调试程序并进入循环

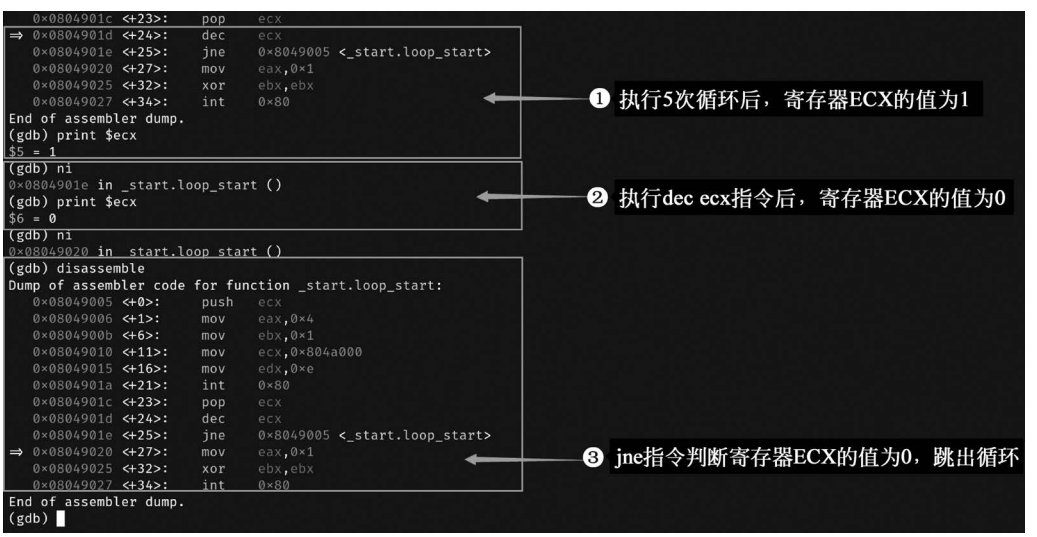

图 5-27　执行 5 次循环后，跳出循环

```
//ch05/loop2.asm
1  section .data
2      message db 'Hello Hacker!', 0xA
3      msg_length equ $ - message

4  global _start
```

```
5  section .text
6  _start:
7      mov ecx, 5

8   .loop_start:
9      push ecx
10     mov eax, 4
11     mov ebx, 1
12     mov ecx, message
13     mov edx, msg_length
14     int 0x80
15     pop ecx
16     loop .loop_start
17     mov eax, 1
18     xor ebx, ebx
19     int 0x80
```

第16行代码表示对寄存器ECX的值执行减1操作并判断该值是否为0。如果ECX不为0，则会跳转到.loop_start标签位置，进入循环，如图5-28所示。

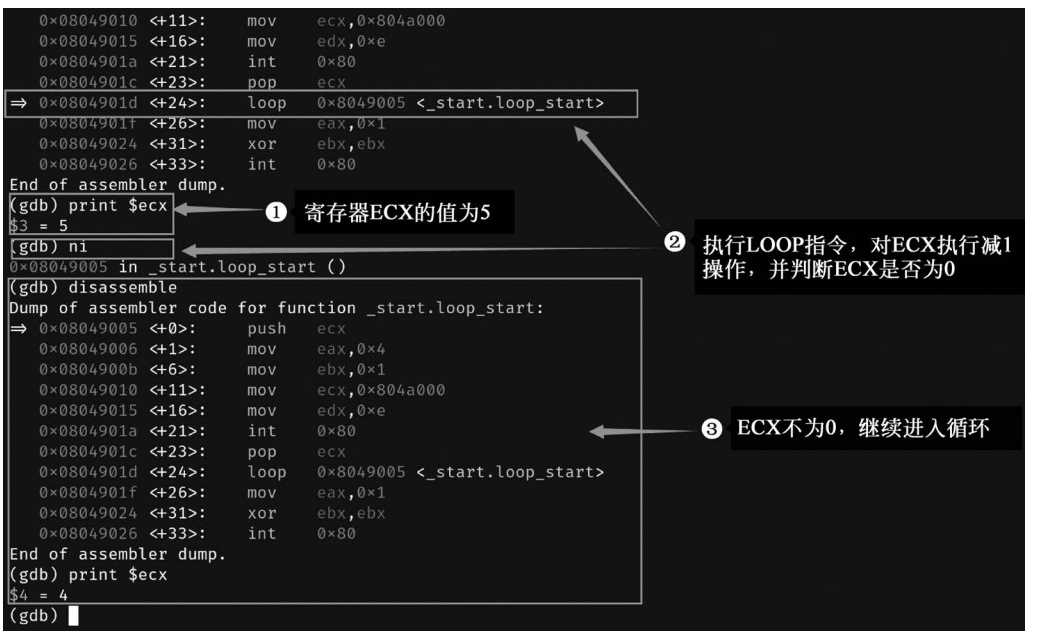

图5-28　使用gdb调试程序并进入循环

如果寄存器ECX的值为0，则LOOP指令会判断ECX为0并跳出循环，如图5-29所示。

显然，使用LOOP指令可以替代DEC和JNZ的组合来实现计数循环。尽管计数循环能够执行指定次数的代码，但固定的循环次数在处理动态数据时不够灵活，无法适应不同的输入或条件，因此采用条件循环能够更好地满足在不同条件下的循环需求。

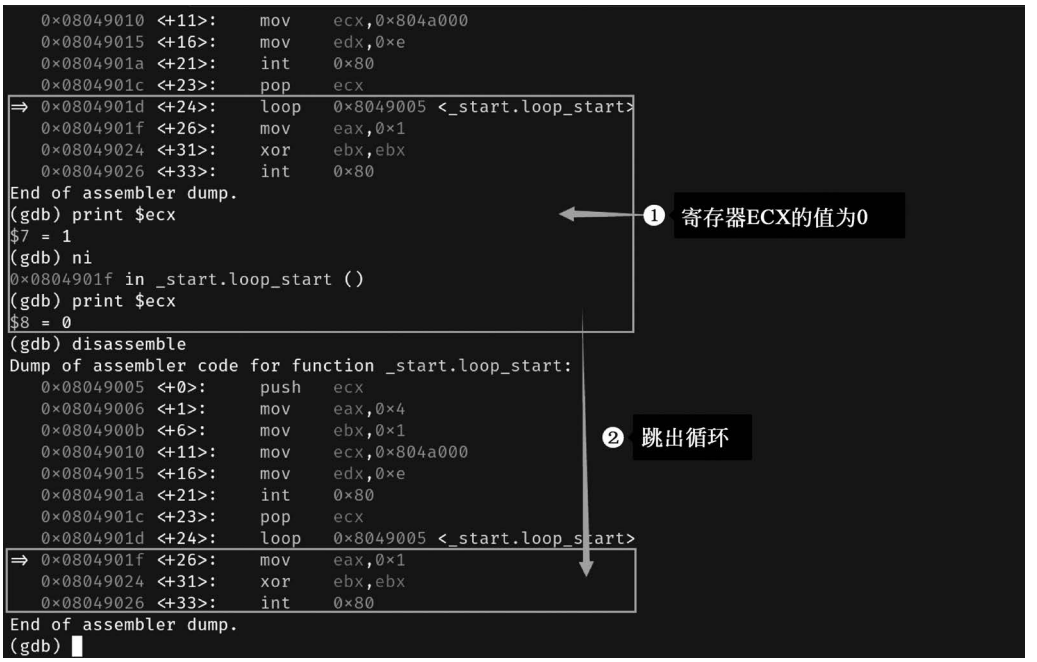

图 5-29　寄存器 ECX 的值为 0，跳出循环

5.3.2　条件循环

条件循环是一种根据特定条件执行代码块的循环结构。与计数循环不同，条件循环在每次迭代时都会检查条件，以决定是否继续执行。笔者认为条件循环的本质是组合使用条件结构与循环结构，从而能够使用自定义条件判断来执行循环代码。如果符合判断条件，则进入循环，否则跳出循环，如图 5-30 所示。

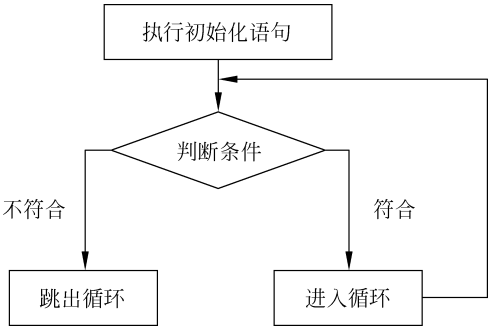

图 5-30　条件循环结构的基本原理

通过条件循环，程序能够更灵活地应对各种情况，适应动态变化的需求。例如，使用条件循环结构实现计算 1～10 的奇数和，代码如下：

```
//ch05/loop3.asm
1  section .data
2     sum db 0

3  global start
4  section .text
5  start:
6     mov ecx, 1
7     mov eax, 0

8  .loop:
9     cmp ecx, 11
10    jge .done
11    test ecx, 1
12    jz .skip
13    add eax, ecx
14 .skip:
15    inc ecx
16    jmp .loop
17 .done:
18    mov ebx, eax
19    mov eax, 1
20    int 0x80
```

第1行和第2行代码表示在数据段中定义变量sum，并将其初始化为数值0。变量sum的功能是用于保存计算结果。在gdp调试器中，使用info variables命令能够查看定义的变量，通过x命令可以查看变量对应的值，如图5-31所示。

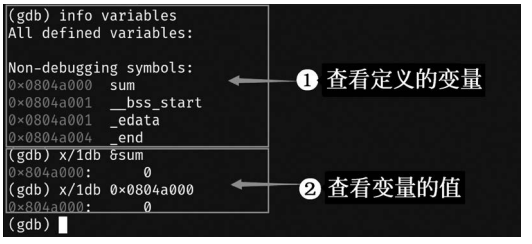

图5-31　查看变量sum的值

第3～7行代码用于将_start声明为全局标签，使其成为程序的入口点。在代码段的_start标签下，将数字1和0分别传送到寄存器ECX和EAX。ECX用于初始化计数器，从1开始，而EAX被初始化为0，用于在后续循环中累加奇数和。如果gdp调试器成功地执行了第3～7行代码，则可以使用print命令来查看寄存器中保存的值，如图5-32所示。

第8行代码用于声明标签.loop，表示循环的主要部分。第9行和第10行代码使用CMP指令比较寄存器ECX与数值11，并通过JGE指令判断结果。如果ECX的值小于11，则程序将不进行跳转，而执行第11行代码test ecx,1，如图5-33所示。

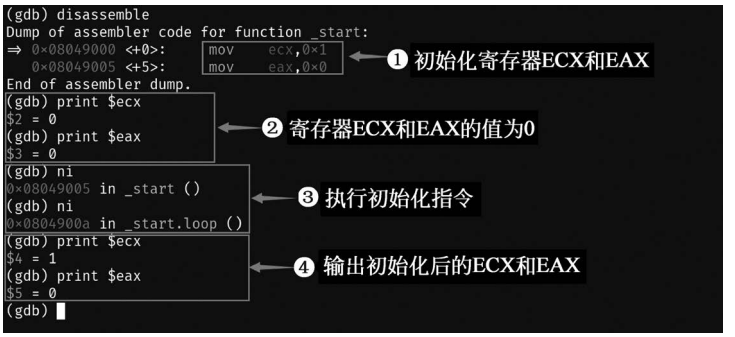

图 5-32　执行初始化 ECX 和 EAX 的指令

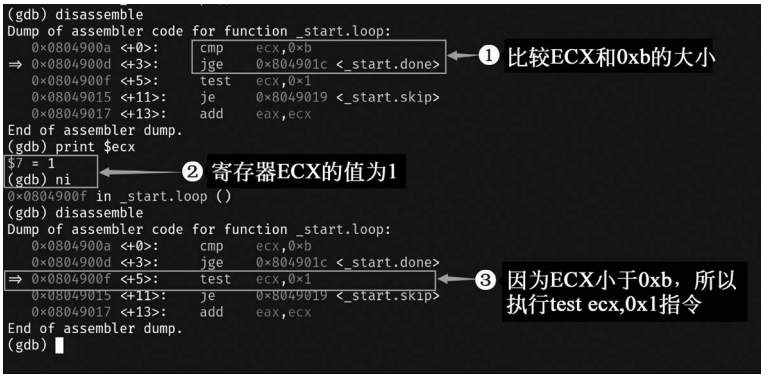

图 5-33　寄存器 ECX 的值小于 11 的情况

第 11～13 行代码用于判断 ECX 是否为奇数。如果是奇数，则程序将执行 add ecx，eax 指令，将计数累加到寄存器 EAX 中。例如，寄存器 ECX 的值为 1，程序执行的流程如图 5-34 所示。

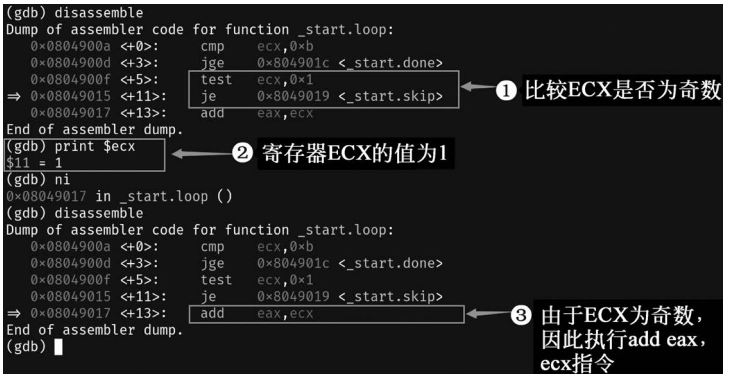

图 5-34　ECX 为奇数时，程序的执行流程

注意：test 指令用于执行按位与操作，但不会改变操作数的值。它的语法是 test 操作数 1，操作数 2 会计算操作数 1 和操作数 2 的按位与，并根据结果设置标志寄存器。如果 ECX 是奇数，最低位为 1，test ecx，1 的结果非零，则 ZF 为 0。如果 ECX 是偶数，最低位为

0，test ecx，1 的结果为 0，则 ZF 被置为 1。JZ 指令会在标志位 ZF 等于 1 时，执行跳转操作，否则不执行跳转。

如果寄存器 ECX 保存的值为偶数，则会执行 JZ 指令跳转到标签.skip。例如，ECX 的值为 2，程序的执行流程如图 5-35 所示。

图 5-35　ECX 为偶数时，程序的执行流程

第 14～16 行代码实现了标签为.skip 的代码块，负责对寄存器 ECX 进行加 1 操作，并跳转到标签.loop。该过程将持续进行，直到 ECX 的值达到或超过 11，此时程序将跳转至标签.done，以正常退出程序。最终，寄存器 EAX 保存 1～10 的奇数和。使用 gdp 调试器的 info register eax 命令能够查看该寄存器的值，如图 5-36 所示。

图 5-36　查看寄存器 EAX 保存的 1～10 的奇数和

条件循环可以精确地控制循环的每个步骤，包括循环的初始化、条件检查和迭代，可以实现复杂的循环逻辑，因此它也是最常用的循环结构。

5.3.3　无限循环

汇编语言中的无限循环是一种程序结构，它会持续执行某段代码，直到外部条件强制停止，通常被称为死循环。其基本结构通常通过跳转指令实现，例如，使用 JMP 指令不断地跳转到同一标签，从而形成循环。这种结构常用于等待外部事件或处理持续任务，例如，实现无限向终端输出"Hello Hacker!"字符串信息的程序，代码如下：

```
//ch05/loop4.asm
1  section .data
2      msg db 'Hello Hacker', 0xA

3  global _start
```

```
4  section .text
5  _start:
6      jmp write_msg

7  write_msg:
8    mov eax, 4
9    mov ebx, 1
10   mov ecx, msg
11   mov edx, 13
12   int 0x80
13   jmp write_msg
14   mov eax, 1
15   xor ebx, ebx
16   int 0x80
```

第 1 行和第 2 行代码在数据段中定义了变量 msg,并为其赋值字符串"Hello Hacker",最后通过 0xA 添加换行符,标志字符串的结束。第 3～6 行代码声明了全局标签_start 作为程序的入口点,并在程序开始时通过 jmp 指令跳转到 write_msg 标签的代码块。第 7～12 行代码通过 sys_write 系统调用将变量 msg 的值输出到终端。第 13 行使用 jmp 指令无条件跳转到标签 write_msg 相应的代码块,导致程序陷入无限循环,持续输出"Hello Hacker"。由于无限循环的存在,第 14 至第 16 行代码中表示正常退出的部分永远不会被执行。如果在终端中运行 loop4 可执行程序,则会不停地输出"Hello Hacker",如图 5-37 所示。

```
┌──(kali㉿kali)-[~/Desktop/asm/ch05]
└─$ ./loop4
Hello Hacker
Hello Hacker
Hello Hacker
Hello Hacker
Hello Hacker
Hello Hacker
Hello Hacker
Hello Hacker
```

图 5-37　执行 loop4 可执行程序

如果要停止这种无限循环程序,则只能通过快捷键 Ctrl+C 来强制终止。无限循环是常用的循环类型之一,常用于需要实时刷新和监控的场景中,例如,服务进程、设备驱动、事件监听器等。无限循环通常会依赖外部事件或特定的退出条件来结束运行。