

第15章 TinyC 后端

至此我们的 TinyC 前端已经完成，可以将 TinyC 源程序编译成中间代码 Pcode，且可以用 Pcode 模拟器来运行 TinyC 前端生成的 Pcode。接下来编写 TinyC 后端，将中间代码编译、链接成可执行源程序。我们将针对所有 Pcode 命令编写同名的 NASM 宏将 Pcode 翻译成 x86 (32位) 汇编指令，然后利用 Nasm 汇编成二进制目标程序，最后用链接器 ld 链接成 Linux 下的 32 位可执行程序。

15.1 NASM 简介

NASM 全称 The Netwide Assembler，是一款基于 x86 平台的汇编语言编译程序，其设计初衷是为了实现编译器程序跨平台和模块化的特性。NASM 支持大量的文件格式，包括 Linux，BSD，a.out，ELF，COFF，Mach-0，Microsoft 16-bit OBJ，Win32 以及 Win64，同时也支持简单的二进制文件生成。它的语法被设计的简单易懂，相较 Intel 的语法更为简单，支持目前已知的所有 x86 架构之上的扩展语法，同时也拥有对宏命令的良好支持。

用 NASM 编写 Linux 下的 hello world 示例程序 `hello.nasm` 如下：

```
GLOBAL _start

[SECTION .TEXT]
_start:
    MOV EAX, 4           ; write
    MOV EBX, 1           ; stdout
    MOV ECX, msg
    MOV EDX, len
    INT 0x80             ; write(stdout, msg, len)

    MOV EAX, 1           ; exit
    MOV EBX, 0
    INT 0x80             ; exit(0)

[SECTION .DATA]
    msg: DB "Hello, world!", 10
    len: EQU $-msg
```

编译和运行的命令如下（Debian-8.4-amd64 环境下）：

```
$ nasm -f elf32 -o hello.o hello.nasm
$ ld -m elf_i386 -o hello hello.o
$ ./hello
Hello, world!
```

Linux 32位可执行程序中，用“INT 0x80”指令来执行一个系统调用，用“EAX”指定系统调用编号，用“EBX, ECX, EDX”来传递系统调用需要的参数。上面这段汇编代码中，首先执行了编号为 4 的系统调用 (write)，向 stdout 写了一个长为 len 的字符串 (msg)，之后，执行编号为 1 的系统调用 (exit)。

NASM 拥有对宏命令的良好支持，可以简化很多重复代码的编写。对于上面这个程序，可以编写两个名为 print 和 exit 的宏用来重复使用。新建一个 `macro.inc` 文件，内容如下：

```
%MACRO print 1
    [SECTION .DATA]
        %%STRING:    DB %1, 10
        %%LEN:       EQU $-%%STRING
    [SECTION .TEXT]
        MOV EAX, 4          ; write
        MOV EBX, 1          ; stdout
        MOV ECX, %%STRING
        MOV EDX, %%LEN
        INT 0x80            ; write(stdout, %%STRING, %%LEN)
%ENDMACRO

%MACRO exit 1
    MOV EAX, 1
    MOV EBX, %1
    INT 0x80
%ENDMACRO

GLOBAL _start

[SECTION .TEXT]
_start:
```

新的 `hello.nasm` 如下：

```
%include "macro.inc"

print "Hello world!"
```

```
print "Hello again!"
exit 0
```

后面这段代码够简洁吧。

上面这段代码中的 `%include` 命令和 C 语言中的 `#include` 的作用是一样的，就是把 `%include` 后面的文件名对应的文件的内容原样的拷贝进来。

下面再来解释一下 NASM 宏的使用。首先看简单一点的 `exit` 宏。NASM 中：`%MACRO` 是宏定义的开始；`%MACRO` 后面接宏的名称；此处是 “`exit`”；宏名后面是宏的参数数量，此处是 “1”，表示该宏带有一个参数，宏内部中可以用 “`%1, %2, %3, ...`” 来引用宏的第 1、2、3、... 个参数；`%ENDMACRO` 是宏定义的结束。

宏定义好后，若后面的代码中遇到这个宏，则会用宏定义中的内容来替换这个宏。如 `hello.nasm` 中的第 5 行 “`exit 0`”，会被替换成：

```
MOV EAX, 1
MOV EBX, 0
INT 0x80
```

注意宏定义中的 `%1` 将被替换为 `exit` 后面的参数 `0`。

`print` 宏定义稍微复杂一点，多了 `%%STRING` 和 `%%LEN`，它们可以看成是宏定义中的局部名称，在每个 `print` 宏被展开的时候，NASM 会为这种类型的名称生成一个唯一的标志符。我们可以用 `nasm -e hello.nasm` 来查看 `hello.nasm` 文件经过预处理后的代码，如下（以下代码经过的适当的缩进和注释处理）：

```
[global _start]

[SECTION .TEXT]
_start:

; print "Hello world!"
[SECTION .DATA]
    ..@1.STRING: DB "Hello world!", 10
    ..@1.LEN: EQU $-..@1.STRING
[SECTION .TEXT]
    MOV EAX, 4
    MOV EBX, 1
    MOV ECX, ..@1.STRING
```

```

MOV EDX, ..@1.LEN
INT 0x80

; print "Hello again"
[SECTION .DATA]
    ..@2.STRING: DB "Hello again!", 10
    ..@2.LEN: EQU $-..@2.STRING
[SECTION .TEXT]
    MOV EAX, 4
    MOV EBX, 1
    MOV ECX, ..@2.STRING
    MOV EDX, ..@2.LEN
    INT 0x80

; exit 0
MOV EAX, 1
MOV EBX, 0
INT 0x80

```

可以看到，在 ‘print “Hello world!”’ 宏中， %%STRING 被展开为 ..@1.STRING，而在 ‘print “Hello again!”’ 宏中， %%STRING 被展开为 ..@2.STRING。

15.2 用 NASM 宏将 Pcode 命令翻译成 x86 指令（print 命令）

上面简单介绍了 NASM 以及它的强大的宏命令，可以实现复杂多样的宏展开。从本节开始，将编写一系列的宏定义，将中间代码 Pcode 命令展开为 x86 汇编指令。建议读者先回顾一下第 3、4 章的内容，再来看本章接下来的内容。

在开始编写宏定义之前，首先说明一下两个约定：（1）所有 Pcode 命令以及相应的宏名称都小写（FUNC / ENDFUNC 命令除外），而所有 x86 汇编指令都大写；（2）本书中的 x86 汇编指令，均用 NASM 语法书写。

首先翻译 Pcode 命令中的 print 命令。上一节中定义的 print 宏已经和 Pcode 中的 print 命令在使用的格式上是一模一样的了，但是它还不能实现 %d 格式化输出。我们最终的 print 宏需要使下面这段代码（[print.nasm](#)）的输出为 “a = 1, b = 2, c = 3”：

```
PUSH DWORD 1
PUSH DWORD 2
PUSH DWORD 3
print "a = %d, b = %d, c = %d"

exit 0
```

直接看代码吧。宏文件 `macro.inc` :

```
%MACRO print 1
    [SECTION .DATA]
        %%STRING:  DB %1, 10, 0
    [SECTION .TEXT]
        PUSH DWORD %%STRING
        CALL PRINT
        SHL EAX, 2
        ADD ESP, EAX
%ENDMACRO

%MACRO exit 1
    MOV EAX, 1
    MOV EBX, %1
    INT 0x80
%ENDMACRO

EXTERN PRINT
GLOBAL _start

[SECTION .TEXT]
_start:
```

从宏文件可以看出， `print` 宏将会被展开为一个 `PUSH` 命令，一个函数调用命令（`CALL PRINT`），以及清栈的命令。具体的输出工作将由 `PRINT` 函数来处理，同时 `PRINT` 函数还需要返回字符串中含有的 `%d` 的个数，这样函数调用完毕后可以由返回值（保存在 `EAX` 中）来进行清栈（这就是“`SHL EAX, 2`”和“`ADD ESP, EAX`”的作用）。

`PRINT` 函数可以用 C 语言来编写，然后编译成库文件，最后和目标文件一起链接成可执行文件。`PRINT` 函数源代码如下（ `tio.c` ）：

```
void SYS_PRINT(char *string, int len);

#define BUFLen 1024
```

```

int PRINT(char *fmt, ...)
{
    int *args = (int*)&fmt;
    char buf[BUFLen];
    char *p1 = fmt, *p2 = buf + BUFLen;
    int len = -1, argc = 1;

    while (*p1++) ;

    do {
        p1--;
        if (*p1 == '%' && *(p1+1) == 'd') {
            p2++; len--; argc++;
            int num = *(++args), negative = 0;

            if (num < 0) {
                negative = 1;
                num = -num;
            }

            do {
                *(--p2) = num % 10 + '0'; len++;
                num /= 10;
            } while (num);

            if (negative) {
                *(--p2) = '-'; len++;
            }
        } else {
            *(--p2) = *p1; len++;
        }
    } while (p1 != fmt);

    SYS_PRINT(p2, len);

    return argc;
}

```

```

void SYS_PRINT(char *string, int len)
{
    __asm__(
        ".intel_syntax noprefix\n\
        PUSH EAX\n\
        PUSH EBX\n\
        PUSH ECX\n\
        PUSH EDX\n\
        \n\
        MOV EAX, 4\n\

```

```
        MOV EBX, 1\n\
        MOV ECX, [EBP+4*2]\n\
        MOV EDX, [EBP+4*3]\n\
        INT 0X80\n\
        \n\
        POP EDX\n\
        POP ECX\n\
        POP EBX\n\
        POP EAX\n\
        .att_syntax"
    );
}
```

用以下命令将 `tio.c` 编译成库文件 `libtio.a` 。

```
gcc -m32 -c -o tio.o tio.c
ar -crv libtio.a tio.o
```

再将 `print.nasm` 汇编成目标文件 `print.o` 。

```
nasm -f elf32 -P"macro.inc" -o print.o print.nasm
```

最后将 `print.o` 链接为可执行文件 `print`，链接时指定 `tio` 库（库文件为 `libtio.a`），命令如下：

```
ld -m elf_i386 -o print print.o -L. -ltio
```

运行 `print` 将输出 `"a = 1, b = 2, c = 3"` 。

以上文件中，`print.nasm` 中的 `CALL PRINT`（由 `print` 宏展开得到）将调用定义在 `tio.c` 中的 `PRINT` 函数。在 Linux（32位）的汇编编程中，如果一个文件需要调用由外部文件定义的函数，那么需要遵循以下约定：

- (1) 本文件中需有 **EXTERN funcname**，表示需要引用外部函数（函数名为 `funcname`）；
- (2) 函数的参数通过栈传递，且按从右向左的顺序入栈，函数的第一个参数要最后一个入栈；
- (3) 函数开头的汇编指令为 `"PUSH EBP; MOV EBP, ESP"`，函数结尾的汇编指令为 `"MOV ESP, EBP; POP EBP; RET"`，因

此，在函数体内，第一个参数保存在 EBP+8 处，第二个参数保存在 EBP+12，第三个参数保存在 EBP+16，以此类推，...。

(4) 入栈的参数由调用者负责出栈。

下面结合这四个约定来详细的说明一下 print 宏是如何模拟出 Pcode 中 print 命令的效果的：

(1) 首先，在 macro.inc 中定义了 print 宏，因此 print.nasm 中的代码：

```
PUSH DWORD 1
PUSH DWORD 2
PUSH DWORD 3
print "a = %d, b = %d, c = %d"
```

将会被展开为下面的形式：

```
[SECTION .TEXT]
    PUSH DWORD 1
    PUSH DWORD 2
    PUSH DWORD 3
    PUSH DWORD %%STRING
    CALL PRINT
    SHL EAX, 2
    ADD ESP, EAX

[SECTION .DATA]
    %%STRING:  DB "a = %d, b = %d, c = %d", 10, 0
```

(2) 以上代码中的“CALL PRINT”将调用定义在 tio.c 中的 PRINT 函数，该函数原型为 int PRINT(char *fmt, ...)，其中第一个参数 fmt 就是最后一个入栈的参数，也就是字符串“a = %d, b = %d, c = %d\n\0”的起始地址。

(3) PRINT 函数中的第一行 int *args = (int*)&fmt 得到 fmt 的地址（注意：不是 fmt 的值），因此，args+1 就是倒数第二个入栈的参数的地址，*(args+1) 就是该参数的值（此处为 3），*(args+2) 就是倒数第三个入栈的参数的值（此处为 2），...。

(4) PRINT 函数首先找到字符串 `fmt` 的结尾，然后从结尾一直向前扫描该字符串，如果扫描到普通字符，则直接拷贝到 `buf` 数组中，如果扫描到一个 `"%d"`，则执行 `num = *(++args)` 得到相应的参数的数值，然后将此数值转换为字符串并拷贝到 `buf` 数组中，按此原则一直扫描到字符串的开头，最后将 `buf` 数组中的内容打印到终端。

(5) 打印完毕后，PRINT 函数返回 `fmt` 中含有的 `"%d"` 的个数（保存在 `EAX` 中），因此，`"CALL PRINT"` 后面的 `"SHL EAX, 2"` 和 `"ADD ESP, EAX"` 会将所有的入栈的参数都出栈。

15.3 翻译 Pcode 中的 `readint` 命令

`readint` 命令的翻译和 `print` 命令的翻译方法差不多，也需要利用 C 语言编写库函数。以下为相关的代码：

测试代码 `test.nasm`：

```
readint "Please input an number: "  
print "Your input is: %d"  
exit 0
```

`readint` 宏，在 `macro.inc` 文件中：

```
%MACRO readint 1  
    [SECTION .DATA]  
        %%STRING: DB %1, 0  
    [SECTION .TEXT]  
        PUSH DWORD %%STRING  
        CALL READINT  
        MOV [ESP], EAX  
%ENDMACRO  
EXTERN PRINT, READINT
```

`READINT` 库函数，在 `tio.c` 文件中：

```
int STRLEN(char *s);  
int SYS_READ(char *buf, int len);  
  
int READINT(char *prompt) {  
    char buf[BUFLen], *p = buf, *p_end;  
    SYS_PRINT(prompt, STRLEN(prompt));
```

```

int len = SYS_READ(buf, BUFLen-1), value = 0, negative = 0;

p_end = buf + len + 1;

while (p ≠ p_end) {
    if (*p = ' ' || *p = '\t') {
        p++;
    } else {
        break;
    }
}

if (p ≠ p_end && *p = '-') {
    negative = 1;
    p++;
}

while (p ≠ p_end) {
    if (*p ≤ '9' && *p ≥ '0') {
        value = value * 10 + *p - '0';
        *p++;
    } else {
        break;
    }
}

if (negative) {
    value = -value;
}

return value;
}

int STRLEN(char *s) {
    int i = 0;
    while(*s++) i++;
    return i;
}

int SYS_READ(char *buf, int len) {
    __asm__(
        ".intel_syntax noprefix\n\
        PUSH EBX\n\
        PUSH ECX\n\
        PUSH EDX\n\
        \n\
        MOV EAX, 3\n\
        MOV EBX, 2\n\

```

```
        MOV ECX, [EBP+4*2]\n\  
        MOV EDX, [EBP+4*3]\n\  
        INT 0X80\n\  
        \n\  
        POP EDX\n\  
        POP ECX\n\  
        POP EBX\n\  
    .att_syntax"  
);  
}
```

makefile 文件:

```
test: test.o libtio.a  
    ld -m elf_i386 -o test test.o -L. -ltio  
  
run: test  
    ./test  
  
test.o: test.nasm macro.inc  
    nasm -f elf32 -P"macro.inc" -o test.o test.nasm  
  
libtio.a: tio.c  
    gcc -m32 -c -o tio.o tio.c  
    ar -crv libtio.a tio.o  
  
clean:  
    rm test.o test tio.o libtio.a
```

将以上四个文件下载下来放到一个目录，输入 `make run` 即可编译并运行测试代码。运行过程如下：

```
$ make run  
...  
./test  
Please input an number: 15  
Your input is: 15
```

15.4 翻译 Pcode 中的算术命令、 push/pop 命令以及 jmp/jz 命令

算术命令 (add / sub / mul / div / mod / cmpeq / cmpne / cmpgt / cmplt / cmpge / cmple / and / or / not / neg 命令)、push/pop 命令以及 jmp/jz 命令的操作很简单, 因此将其翻译成 x86 指令也很简单, 结合第 3 、 4 章中介绍的这些命令对栈的操作步骤, 典型的宏定义如下:

```
%MACRO add 0
    POP EAX
    ADD DWORD [ESP], EAX
%ENDMACRO
```

```
%MACRO sub 0
    POP EAX
    SUB DWORD [ESP], EAX
%ENDMACRO
```

```
%MACRO cmpeq 0
    MOV EAX, [ESP+4]
    CMP EAX, [ESP]
    PUSHF
    POP EAX
    SHR EAX, 6
    AND EAX, 0X1
    ADD ESP, 4
    MOV [ESP], EAX
%ENDMACRO
```

```
%MACRO not 0
    MOV EAX, [ESP]
    OR EAX, EAX
    PUSHF
    POP EAX
    SHR EAX, 6
    AND EAX, 0X1
    MOV [ESP], EAX
%ENDMACRO
```

```
%MACRO jz 1
    POP EAX
    OR EAX, EAX
    JZ %1
%ENDMACRO
```

```
%MACRO jmp 1
    JMP %1
%ENDMACRO
```

```

%MACRO push 1
    PUSH DWORD %1
%ENDMACRO

%MACRO pop 0-1
    %IFIDN %0, 0
        ADD ESP, 4
    %ELSE
        POP DWORD %1
    %ENDIF
%ENDMACRO

```

所有宏定义见 `macro.inc` 。

测试文件 `test.nasm` :

```

MOV EBP, ESP
SUB ESP, 8
%define a [EBP-4]
%define b [EBP-8]

; a = readint("Please input an number `a`: ")
readint "Please input an number `a`: "
pop a      ; ==> POP DWORD [EBP-4]

; b = readint("Please input another number `b`: ")
readint "Please input another number `b`: "
pop b      ; ==> POP DWORD [EBP-8]

; print("a = %d", a)
push a     ; ==> PUSH DWORD [EBP-4]
print "a = %d"

; print("b = %d", b)
push b     ; ==> PUSH DWORD [EBP-8]
print "b = %d"

; print("a - b = %d", a - b)
push a
push b
sub
print "a - b = %d"

; if (a > b) { print("a > b"); } else { print("a ≤ b") }
push a

```

```
push b
cmpgt
jz _LESSEQUAL
print "a > b"
jmp _EXIT
_LESSEQUAL:
    print "a ≤ b"
_EXIT:
    exit 0
```

库函数文件 (`tio.c`) 和 `makefile` 文件, 和上一节是一样的。

将以上四个文件下载下来放到一个目录, 输入 `make run` 即可编译并运行测试代码。运行过程如下:

```
$ make run
...
./test
Please input an number `a`: 46
Please input another number `b`: 48
a = 46
b = 48
a - b = -2
a ≤ b
```

`test.nasm` 的开头在栈上分配了 8 个字节的空间 (也就是 2 个 `int`), 并定义了 `a` 和 `b` 代表这储存在这两个 `int` 型空间中的数值:

```
MOV EBP, ESP
SUB ESP, 8
%define a [EBP-4]
%define b [EBP-8]
```

在 `macro.inc` 中定义了 `pop` 和 `push` 宏, `test.nasm` 中的 “`pop a`” 和 “`push a`” 将会被展开为: “`POP DWORD [EBP-4]`” 和 “`PUSH DWORD [EBP-8]`”。

15.5 翻译 Pcode 中的自定义函数命令和变量声明命令

上一节我们已经实现了将简单 TinyC 语句手工翻译成 Pcode , 然后编写了 NASM 宏将这些 Pcode 翻译成 x86 指令, 最后汇编、链接成可执行程序。我们已经编写了大部分 Pcode 命令所对应的宏, 本节将编写 NASM 宏来翻译 Pcode 中最复杂、也是最难翻译的命令-自定义函数命令和变量声明命令 (FUNC / ENDFUNC / arg / ret / \$func_name / var) 。

具体来说, 我们需要将以下 Pcode 翻译成 x86 指令:

```
; int main() {
FUNC @main:
    ; int a;
    var a

    ; a = 3;
    push 3
    pop a

    ; print("sum = %d", sum(4, a));
    push 4
    push a
    $sum
    print "sum = %d"

    ; return 0;
    ret 0
; }
ENDFUNC

; int sum(int a, int b) {
FUNC @sum:
    arg a, b

    ; int c;
    var c

    ; c = a + b;
    push a
    push b
    add
    pop c

    ; return c;
    ret c
; }
ENDFUNC
```

最难的部分在于如何避免不同函数中的同名变量的冲突。NASM 的宏虽然强大，但无法满足如此复杂的需要。为降低翻译的难度，将以上 Pcode 稍微改写一下，保存为 `test.pcode`：

```
; int main() {
FUNC @main:
    ; int a;
    main.var a

    ; a = 3;
    push 3
    pop a

    ; print("sum = %d", sum(4, a));
    push 4
    push a
    $sum
    print "sum = %d"

    ; return 0;
    ret 0
; }
ENDFUNC@main

; int sum(int a, int b) {
FUNC @sum:
    sum.arg a, b

    ; int c;
    sum.var c

    ; c = a + b;
    push a
    push b
    add
    pop c

    ; return c;
    ret c
; }
ENDFUNC@sum
```

首先编写 FUNC 和 ret 宏，放到 `macro.inc` 文件中，同时在该文件的最后增加调用 @main 函数及退出的指令：

```
%MACRO FUNC 1
    %1
    PUSH EBP
    MOV EBP, ESP
%ENDMACRO

%MACRO ret 0-1
    %IFIDN %0, 1
        %IFIDN %1, ~
            MOV EAX, [ESP]
        %ELSE
            MOV EAX, %1
        %ENDIF
    %ENDIF
    LEAVE
    RET
%ENDMACRO
```

```
EXTERN PRINT, READINT
GLOBAL _start
```

```
[SECTION .TEXT]
_start:
    CALL @main
    PUSH EAX
    exit [ESP]
```

然后, 编写 `main.var`, `ENDFUNC@main`, `$sum`, `sum.arg`, `sum.var` 和 `ENDFUNC@sum` 宏, 保存为 `test.funcmacro` :

```
; ==== begin function `main` ====
#define main.varc 1

%MACRO main.var main.varc
    %define a [EBP - 4*1]
    SUB ESP, 4*main.varc
%ENDMACRO

%MACRO ENDFUNC@main 0
    LEAVE
    RET
    %undef a
%ENDMACRO
; ==== end function `main` ====

; ==== begin function `sum` ====
```

```

#define sum argc 2
#define sum varc 1

%MACRO $sum 0
    CALL @sum
    ADD ESP, 4*sum argc
    PUSH EAX
%ENDMACRO

%MACRO sum arg sum argc
    %define a [EBP + 8 + 4*sum argc - 4*1]
    %define b [EBP + 8 + 4*sum argc - 4*2]
%ENDMACRO

%MACRO sum var sum varc
    %define c [EBP - 4*1]
    SUB ESP, 4*sum varc
%ENDMACRO

%MACRO ENDFUNC@sum 0
    LEAVE
    RET
    %undef a
    %undef b
    %undef c
%ENDMACRO

; ==== end function `sum` ====

```

这些宏会将 @sum 函数展开为如下形式:

```

@sum:
    PUSH EBP
    MOV EBP, ESP
    SUB ESP, 4*1


    PUSH DWORD [EBP + 12]    ; push a
    PUSH DWORD [EBP + 8]    ; push b
    add
    POP DWORD [EBP - 4*1]    ; pop c

    MOV EAX, [EBP - 4*1]     ; MOV EAX, c
    LEAVE
    RET

```

最后, 改写 `makefile` 文件中的以下两行:

```
test.o: test.pcode test.funcmacro macro.inc
        nasm -f elf32 -P"macro.inc" -P"test.funcmacro" -o test.o tes
```



将以上四个文件以及库函数文件 `tio.c` 放到用一目录，输入 `make run` 即可编译并运行测试代码。

至此所有 Pcode 命令对应的 NASM 宏编写完毕。

第 15 章完