

# 第八章：目标代码生成

# 目标代码

## 实际目标代码

**绝对地址机器代码：**绝对机器语言代码即是能够立即执行的机器语言代码，代码中所有地址均已定位，编译后可直接执行。

**可重定位的机器代码：**当程序执行时，必须由连接装入程序把它们和一些运行时子程序连接起来，转换成可立即执行的机器语言代码。

**汇编代码：**必须将汇编语言代码转换成可执行的机器语言代码才可以执行。

**虚拟目标代码：**虚拟机上的目标程序。

在本地机器上具备虚拟机的解释器。

# 虚拟目标机的指令系统

目标机指令系统的指令分成如下的形式：

- ◆ 存取指令：LD ST
- ◆ 输入输出：IN OUT
- ◆ 运算型指令：ADD SUB MUL DIV GT GE...
- ◆ 转移型指令：JMP JMPT JMPF
- ◆ 地址运算指令和块传递指令：LEA MOVEB

# 虚拟目标机的指令系统

| 指令名称 | 指令形式      | 指令名称  | 指令形式            |
|------|-----------|-------|-----------------|
| 读    | IN R      | 除     | DIV R, A        |
| 写    | OUT R     | 条件真转移 | JMPT R, A       |
| 取值   | LD R, A   | 条件假转移 | JMPF R, A       |
| 存值   | ST A, R   | 无条件转移 | JMP A           |
| 加    | ADD R, A  | 取址    | LEA R, A        |
| 减    | SUB R, A  | 块传送   | MOVEB A1, A2, S |
| 乘    | MULT R, A |       |                 |

# 虚拟目标机的寄存器

- ◆ 寄存器是CPU内部的元件，寄存器拥有非常高的读写速度，它们可用来暂存指令、数据和地址。
- ◆ 寄存器种类：累加器、变址器、通用寄存器等等
- ◆ 在虚拟目标机中，取出几个寄存器作为地址计算专用的寄存器分别为SP、TOP、SP0；其他寄存器用R1、R2…表示

# 四元式转化为目标指令

四元式等价的转换成目标指令需要用到两个栈：

- ◆ 标号定位栈L1：定位性标号是为了某些转移提供地址的，需要把暂时没用到的标号存在栈中，例如while四元式可以对应一个嵌套的循环，在定位产生时还没产生跳转指令，它的地址还没用到，为了让后面能用到需要用栈把标号保存下来
- ◆ 目标指令地址栈L2：在有些产生跳转指令的时候，转移地址暂时无法确定，例如do四元式，不知道后面的转移地址，则把当前目标指令地址存到栈里，在知道转移地址以后，回填这个指令地址。回填地址是编译中的一项非常重要的技术

# 四元式转化为目标指令

运算型四元式 ( $Op, X, Y, T$ ) : 对应生成3条指令

LD R X; 把X取到R中

Op R Y; 根据运算符对应的运算生成指令, 结果存入R

ST T R; 将R中的内容存入T中

两点重要的解释:

- ❖ 目标指令中X, Y, T对应的一个具体的地址, 如  $sp+off_x$

- ❖ 如果X、Y为间接变量, 这里的寻址方式为间接, 如X为间接变量, 用\*X表示他是一个间接寻址方式。

# 四元式转化为目标指令

赋值型的四元式 ( $=, x, -, y$ ): 生成的目标指令是两条

LD R X; 把x取到寄存器中

ST Y R; 把寄存器中的内容存到y中

注意的问题和前面是一样的。



# 四元式转化为目标指令

- ◆ 输入语句四元式的一般形式为：

(READ, \_ , \_ , A )

表示将一个外部值读入到变量A中，则生成的代码如下：

IN R ; ST A R

- ◆ 输出语句的四元式形式一般为：

(WRITE, A , \_ , \_ )

表示将变量A的值输出，它生成的目标代码为：

LD R A; OUT R

## 四元式转化为目标指令

- ◆ 条件语句的四元式的翻译 `then else`  
`endif` :

(then, x, -, -) 实际产生一条半指令

[n] LD R X; 一条

**[n+1]JMPF R \* (地址) 半条**

## 同时把转移指令地址n+1压入L2栈

# 四元式转化为目标指令

(ELSE, -, -, -), else在起到定位和转移的作用, 因此要产生跳转指令和地址的回填。

[m-1].....

地址回填, 从L2栈中取出指令, 将m+1作为地址加入该指令: [n+1] JMPF R m+1

[m] JMP \*

将m压入L2栈

[m+1].....

(ENDIF, \_\_, \_\_, \_\_), 定位作用, 不产生目标指令, 只回填地址

# 四元式转化为目标指令

- ◆ while循环语句的翻译： WHILE DO  
ENDWHILE
- ◆ (WHILE, -, -, -) 起到一个定位作用，  
不产生指令，  
[n]... ..  
[n+1]记录当前目标指令地址n+1，压入L1栈

# 四元式转化为目标指令

## ◆ 循环语句的翻译while:

(DO, X, -, -) 生成一条半指令

[m] LD X R;

[m+1] JMPF R \*

m+1压入L2栈, 后面遇到再回填

# 四元式转化为目标指令

## ◆ 循环语句的翻译while:

(endwhile, -, -, -) 生成一条指令

[k] JMP L1(top); L1栈顶出栈

将地址k+1回填给L2栈顶代码: JMPF R k+1

L2栈顶出栈

[k+1] .....

例子P183-5 试写出下述程序段的目标代码，其中变量均为非形参实型变量。

```
while (x < y)
{
    y = y + 1;
    if (y > 0)    y = y-x;
    else while (y < 0)
        y = y + x;
}
```

中间代码结构

```
(WHILE,-,-,-)
(LT , x, y, t1)
(DO,t1,-,-)
(ADDF,y,1,t2)
(ASSIG,t2,-,y)
(GT , y, 0, t3)
(THEN, t3,-,-)
(SUBF,y,x,t4)
(ASSIG,t4,-,y)
```

```
(ELSE,-,-,-)
(WHILE,-,-,-)
(LT , y, 0, t5)
(DO,t5,-,-)
(ADDF,y,x,t6)
(ASSIG,t6,-,y)
(ENDWHILE , - , - , - )
(ENDIF,-,-,-)
(ENDWHILE , -, - , - )
```

## 目标代码

1. **L1(top)=1**

LD R,x

2. LT R,y

3. ST t<sub>1</sub>, R

4. LD R,t<sub>1</sub>

5. JMPF R, \* (34)

**L2(top)=5**

6. LD R,y

7. ADD R,1

8. ST t<sub>2</sub>, R

9. LD R,t<sub>2</sub>

10. ST y, R

11. LD R,y

12. GT R,0

13. ST t<sub>3</sub>, R

14. LD R,t<sub>3</sub>

15. JMPF R,\* (22)

**L2(top)=15**

16. LD R,y

17. SUB R,x

18. ST t<sub>4</sub>, R

19. LD R,t<sub>4</sub>

20. ST y, R

21. JMP \*(33)

回填22至L2(top)

**L2(top)=21**

22. **L1(top)=22** LD R,y

23. LT R,0

24. ST t<sub>5</sub>, R

25. LD R,t<sub>5</sub>

26. JMPF R,\* (33)

**L2(top)=26**

27. LD R,y

28. ADD R,x

29. ST t<sub>6</sub>, R

30. LD R,t<sub>6</sub>

31. ST y, R

32. JMP 22:L1(top)

回填33至L2(top)

33. 回填33至L2(top)

JMP 1:L1(top)

回填34至L2(top)

```
while (x < y)
{
    y = y + 1;
    if (y > 0)    y = y-x;
    else while (y < 0)
        y = y + x;
}
```

L1:

L2:

1



# 四元式转化为目标指令

- ◆ **标号语句**：  $(lable, -, - L)$  ，将当前目标代码地址存入L对应的标号表单元。
- ◆ **goto语句**：  $(goto, -, -, L)$  产生一个无条件转移指令 **JMP**  $*$   $(L)$  ，对应的地址暂时为一个指向标号表L的指针。
- ◆ 目标代码全部生成以后，对所有由  $(goto, -, -, L)$  生成的目标指令，依据标号表中对应的目标代码地址信息，进行地址回填。

# 四元式转化为目标指令

- ◆ 过程函数声明部分的四元式处理：
- ◆ (ENTRY, Q, -, -)

| Name | TypePtr | Kind | Level | off | Parm | Class | <u>Code</u> | Size | Forward |
|------|---------|------|-------|-----|------|-------|-------------|------|---------|
|------|---------|------|-------|-----|------|-------|-------------|------|---------|

入口四元式不产生任何指令，而是把当前指令地址填入函数信息表中。

# 四元式转化为目标指令

## ◆ 过程函数声明部分的四元式处理:

(ENDFUNC, -, -, -)

- 1: 生成一组读取命令, 即恢复寄存器的现场信息
- 2: 作废当前的活动记录, 由两个指令完成, 把当前的sp  
存给top, 把动态链指针存给sp

ST top, sp ; LD sp, \*0(top)

- 3: 产生一条返回指令, 根据返回地址生成一个跳转指令。  
。 jmp \*1(top)

如果是返回语句, 要增加一个操作, 将返回结果存储到活动记录\*2(top)中。

\*设过程活动记录首单元存动态链指针、第二个单元存返回地址、第三个单元存返回值。

# 四元式转化为目标指令

## ◆ 函数调用四元式的处理:

值参四元式( ValACT , t , Offset , size )

若t为间接变量, 则生成的目标代码为:

LD R , \* t ; ST Offset(top) , R

b. 若t 为直接变量, 则生成的目标代码为:

LD R , t ; ST Offset(top) , R

c. 若t 为数组, 则生成成组传送的目标代码:

MOVEB t , Offset(top) , size

# 四元式转化为目标指令

## ◆ 函数调用四元式的处理:

变参四元式( VarACT , t , Offset , size )

a. 若t为直接变量, 则生成的目标代码为:

LEA R , t

ST offset(top) , R

b. 若t为间接变量, 则生成的目标代码为:

LD R , t

ST offset(top) , R

# 四元式转化为目标指令

- ◆ 函数调用四元式的处理：  
( CALL , f, true , [ Result ] )
- 1. 生成填写变量访问环境指令
- 2. 把机器状态(寄存器内容)保存到活动记录的寄存器状态区中, 一般应生成一组存的指令
- 3. 要填写管理信息. 首先填写过程层数. 从过程f的语义信息中取其层数, 填入到3(top)中, 生成指令为  
LD R , sem[f].level  
ST 3(top), R

|        |
|--------|
| 临时变量区  |
| 局部变量区  |
| 形参区    |
| 变量访问环境 |
| 活动记录大小 |
| 寄存器状态  |
| 过程层数   |
| 返回值    |
| 返回地址   |
| 动态链指针  |

# 四元式转化为目标指令

## 4. 填写动态链指针

ST 0(top), sp

## 5. 填写返回地址

[A] LD R, A+5

[A+1] ST 1(top), R

## 6. 生成过程活动记录

[A+2] ST sp, top

[A+3] ST top, top + sem[f].size

## 7. 生成转向过程f入口的指令

[A+4] JMP sem[f].code

## 8. 如果是函数调用，则把函数值读到寄存器中（调用返回时，返回值存值时需要，存地址时不需要）

[A+5] LD R, 2(top)

[A+6] ST result, R

# 多寄存器的分配

为了产生更高效的目标代码，合理利用寄存器资源，需要构造一个寄存器状态表：

| 名字 | 状态  | 变量名 |
|----|-----|-----|
| R1 | 0/1 | X   |

当需要用到一个寄存器时，可以用一个函数查找寄存器表，检查是否存在空闲寄存器，若存在，则将空闲的寄存器分配，然后按存取计算操作来操作，同时把相应的寄存器的状态和占用者记录在寄存器状态表



# 多寄存器的分配

- ◆ 当所有寄存器都被占用时，涉及到寄存器的剥夺问题。剥夺算法优劣会影响目标程序的执行效率。
- ◆ 对寄存器的剥夺是把寄存器中的内容存入内存中。
- ◆ 理论上的最优是最远使用点方法，实现困难
- ◆ 局部最优是基本块内最远使用点方法

# 对目标程序的评价

- ◆ 对内存的访问次数总和决定了目标程序的执行效率。
- ◆ 指令的执行代价：访问内存一次的代价为1，例如间接寻址指令的代价较高
- ◆ 多利用寄存器中的资源可以达到减少访问内存的效果。

# 总结

目标程序的生成要考虑的问题：

- ◆ 目标机的具体形式
- ◆ 如何把四元式翻译成目标程序
- ◆ 优化，如何高效的生成目标代码。

◆ 要注意的问题：

符号表一直存在到目标代码生成，目标代码生成之后没有符号表，运行的只是目标程序。