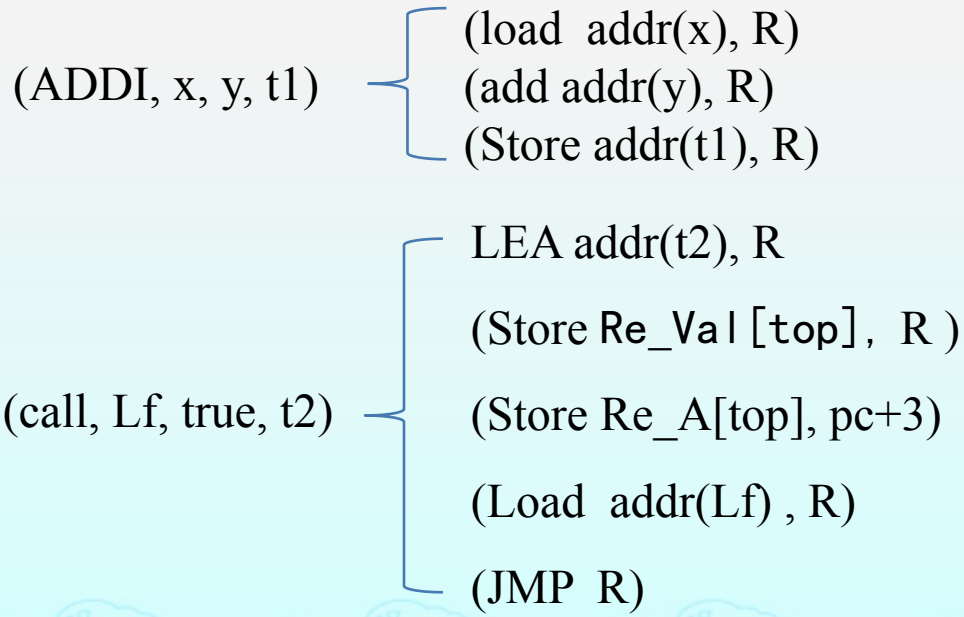
第10章 目标代码生成

- 10.1 目标代码生成的任务
- 10.2 一种虚拟机语言
- 10.3 四元式到虚拟机语言的翻译
- 10.4 目标代码生成时要考虑的问题

10.1 目标代码生成的任务

- 任务：
  - 给变量分配实际内存地址，为目标指令分配寄存器，生成四元式对应的目标指令和管理AR的指令



10.1 目标代码生成的任务

- 过程：
  - 顺次扫描四元式,生成对应的目标指令序列
- 质量标准：
  - 在保证语义相等的情况下，生成的目标指令的条数越少越好，目标指令的运行速度越快越好
- 目标代码的种类：
  - 绝对的机器代码: 执行速度快，缺少灵活性
  - 可重定位机器代码: 支持分模块编译，运行前需连接(link)和装入(load)
  - 汇编代码:需经过汇编后才可运行
  - 虚拟机代码:增强了代码的可移植性，但需要开发虚拟机的解释器，且代码需解释后才可执行

10.2 一种虚拟机语言

- 目标机的指令格式：
  - Op #C, R (立即-----寄存器)
  - Op d[R1], R2 (存储器-----寄存器)
  - Op R1, R2 (寄存器-----寄存器)
  - Op &A, R (绝对地址 ----- 寄存器)
  - Op *R1, R2 (间接寄存器 ----- 寄存器)
  - Op *d[R1], R2 (间接变址 ----- 寄存器)

10.2 一种虚拟机语言

- 抽象地址到绝对地址的映射

10.2 一种虚拟机语言

- 基本指令
  - Load Source, R ------从Source 读出送入R
  - Store Target, R ------R的内容送入Target
  - Add Source, R ------R+Source结果送入R
  - Jmp(0/1), label ------- 跳转到label对应的地址
  - IN R ------- 输入值到R
  - OUT R -------输出R的值
  - LEA A, R ------- A的地址送给R

10.3 四元式到虚拟机语言的翻译

- 四元式种类:
  - 运算型: (op, A, B , T)
  - 赋值: (ASSIGN, A, size, B)
  - 跳转和标号(JUMP(0/1), -,-, L)和(LABEL, -,-, L)
  - 函数相关
    - (ENTRY, Lf, size, level)
    - (ENDFUNC, _, _,_)
    - (VALACT/VARACT,FUNCACT, X.arg, offset, size)
    - (CALL, Lf , true, t)
    - (RETURN, -, -, t)

10.3.1 运算型四元式目标代码的生成

- (op,A, B, t)
  - 确定 A, B 和 T 的目标地址, 记为A.addr, B.addr和T.addr;
  - 找到一个可用的寄存器R,
  - 生成如下目标代码

Load A.addr, R
Op B.addr, R
Store T.addr, R

10.3.1 运算型四元式目标代码的生成

- 例：(+, a, b, t), 其中 a:(0, 2, dir); b:(1, 2, indir), t: (-1, 5, dir)

a的目标地址: 2[gp]
b的目标地址: *(2+InitOff)[sp]
t的目标地址: (5+InitOff)[sp]

```
Load 2[gp], R
Add *(2+InitOff)[sp], R
Store   (5+InitOff)[sp],   R
```

10.3.2 赋值四元式目标代码的生成

- (ASSIGN, A, size, B)
  - 求A和B的地址;
  - 申请寄存器R_A, 生成(Load A.addr, R_A)指令;
  - 生成指令(Store B.addr, R_A);

10.3.3 标号和跳转四元式代码生成

- (LABEL, -, -, L)
  - 不生成目标代码
  - 将当前指令的下一条指令的地址存储起来，放入“标号表”中;
- (JUMP,-,-, L)
  - 到“标号表”中找到L对应的指令地址；若无，则需回填；
  - 生成（Jmp, Pc）

10.3.4 函数相关四元式代码生成

- 相关四元式：
  - (VALACT , a, off, size)
  - (VARACT , a, off, size)
  - (FUNCACT, p, offset, size)
  - (CALL, Lf, true/false, t)
  - (ENTRY, Lf, size, Level)
  - (RETURN, -,-, t)
  - (ENDFUNC, -,-,-)
- 翻译：
  - 函数调用时：申请新的AR空间，参数传递，保存寄存器状态信息，转向相应过程体
  - 函数返回时：释放AR的空间，恢复寄存器状态信息，传递返回值，按照返回地址返回

值参传递

- (VALACT, c, offset,1):c是常量

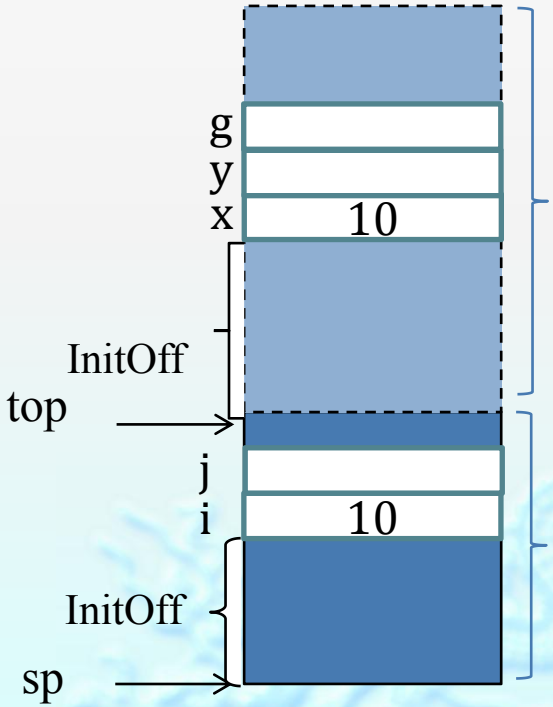
```
Load #c, (offset+InitOff)[top]
```

- (VALACT, a, offset,1):a是变量

```
Load  a.addr, R

Store  (offset+InitOff)[top], R
```

```
int f(int x, float* y, char* g() )   {.....}
char* h()  {.....}
void main()
{
      int i=10; float j;
      f(20, &j, h);       //(VALACT, 20,1, 1)
      f(i, &j, h);        //(VALACT, i, 1, 1)
}
```

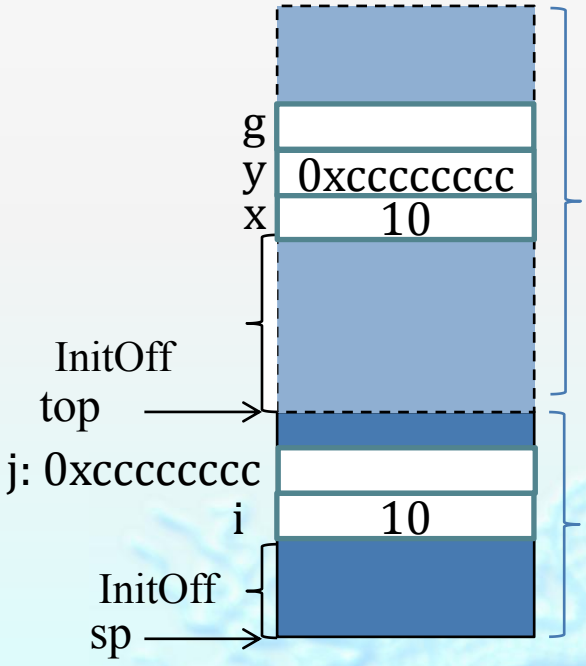


变参传递

- (VARACT, a, offset,1)

```
LEA   a.addr, R
Store   (offset+InitOff)[top], R
```

```
int f(int x, float* y, char* g() )
{.....}
char* h()
{.....}
void main()
{
      int i=10; float j;
      f(10, &j, h);      //(VARACT, 10,1, 1)
 }
```

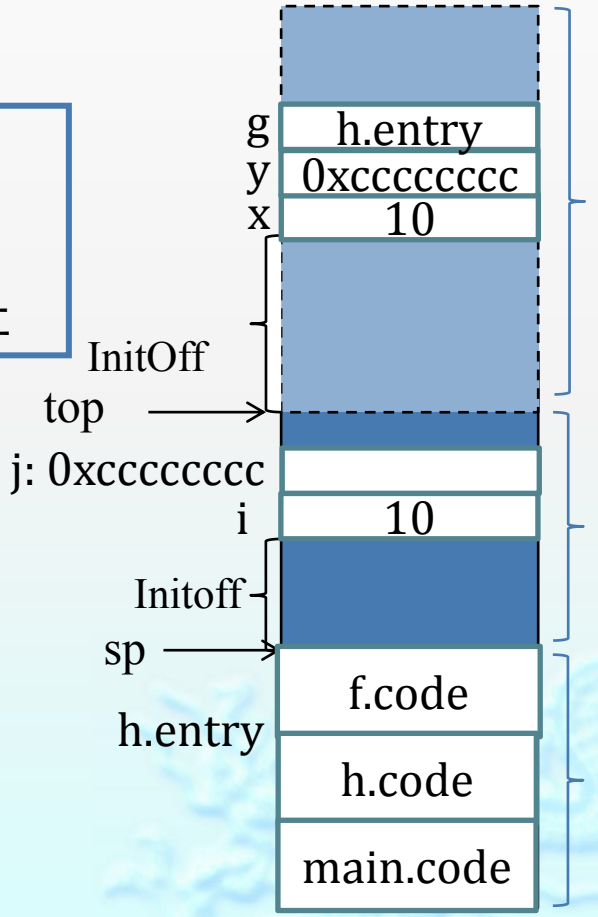


函数参数传递

- (FUNCACT, p, offset, 1)

```
Load Entry(p), R
Store (offset+InitOff)[top], R
        //其中Entry(p)代表的是函数p的入口地址
```

```
int f(int x, float* y, char* g() )
{.....}
char* h()
{.....}
void main()
{
    int i=10; float j;
    f(10, &j, h);      //(FUNCACT, h.entry,1, 1)
}
```



函数调用

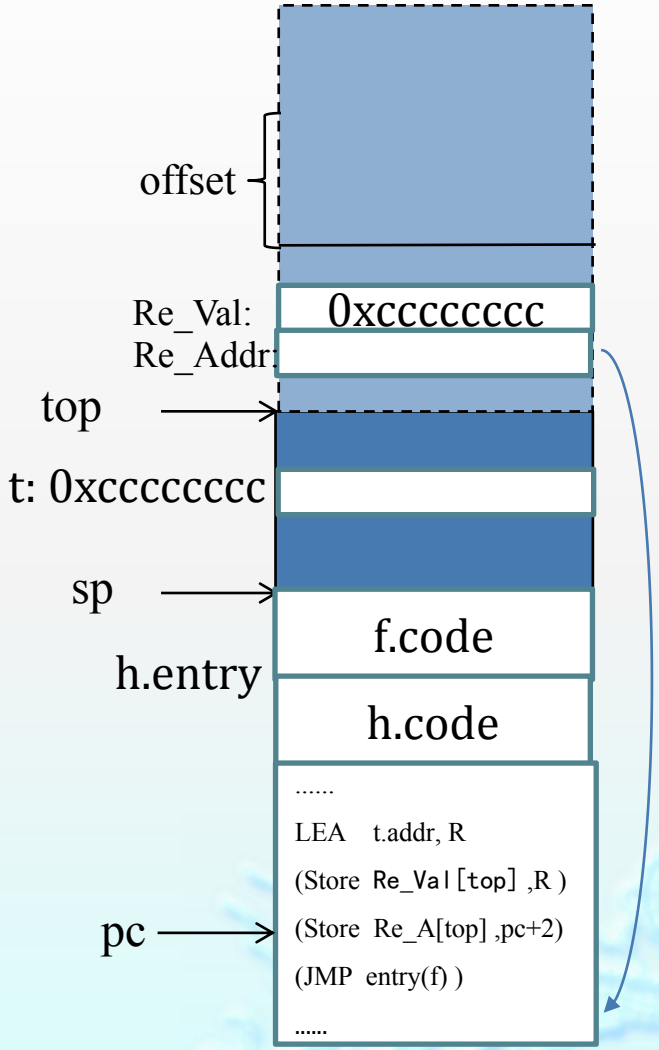
- (call, Lf, true, t)

```
LEA    t.addr, R
(Store  Re_Val[top] ,R )
(Store  Re_A[top] ,pc+2)
(JMP  entry(f) )
```

pc →

pc+2 →

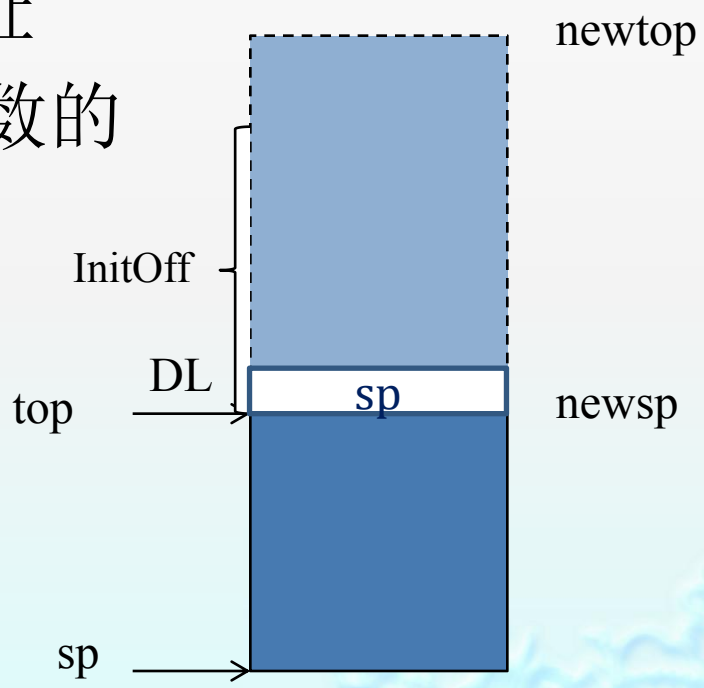
```
int f(int x, float* y, char* g() )
{.....}
char* h()
{.....}
void main()
{
      int i=10; float j;
      f(10, &j, h);      //(call, Lf, true, t)
 }
```



函数入口

- (Entry, Lf , size, Level)
  - 保存调用者的AR的起始地址
  - 调整sp和top指向被调用函数的AR的开始和结束

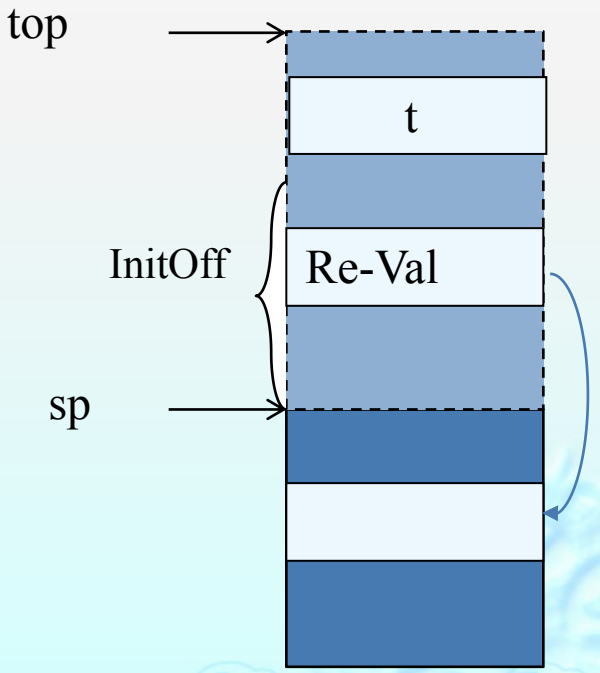
```
(Store  DL[top], sp )
(Load  top, sp)
(ADD  size, top)
```



函数返回

- (RETURN, - , -, t)
  - 将返回值写进Re_Val记录的地址单元
  - 恢复之前保留的各个寄存器的状态

```
Load  Re_Val[sp], R1
Load    t.addr, R2
Store    *R1, R2
```



函数出口

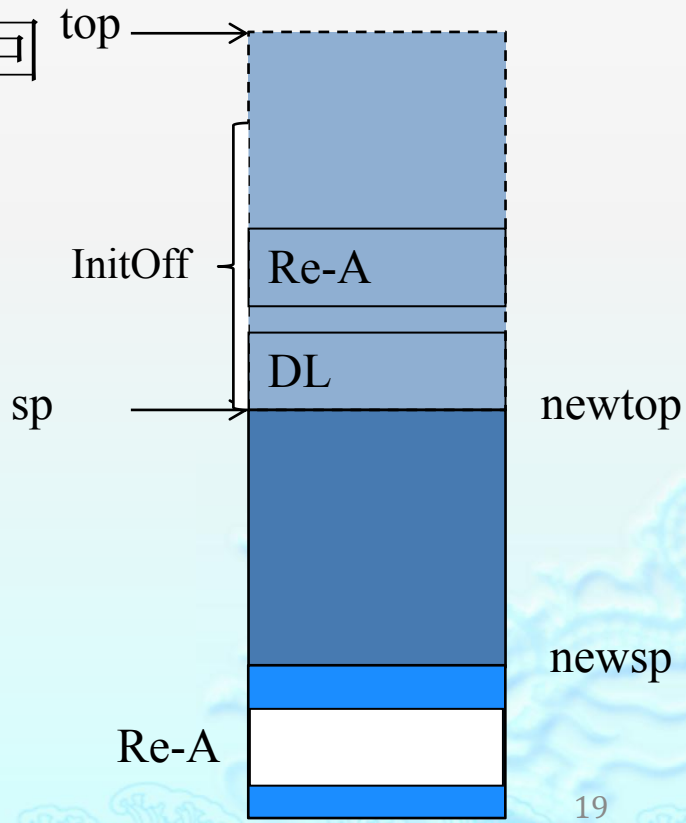
- (ENDFUNC, - , -, -)
  - 调整sp和top指针
  - 按照先前保存的返回地址返回

(Load sp, top)

(Load DL[sp],sp)

(Load Re-A[top], R)

(Jmp R)



一个简单的目标代码生成的例子

```
#define  n 2
int  sum = 0;
int fac(int i)
{ if (i==0) return 1;
  if (i<0) return -1;
  return (i*fac(i-1));
 }

 void main ()
 {
   sum = fac(n);
 }
```

(ASSIGN, 0, _, sum)
(ENTRY, L1, 6 , 0)
(EQ, i, 0, t2)
(JUMP0, t2,_, L2)
(RETURN, _, _, 1)
(LABEL, _, _, L2)

(LT, i, 0, t3)
(JUMP0, t3,_, L3)
(RETURN, _, _, -1)
(LABEL, _, _, L3)

(SUBI, i, 1, t4)
(VALACT, t4,0, 1)
(CALL, L_{fac}, true, t5)
(MULTI, i, t5, t6)
(RETURN, _, _, t6)

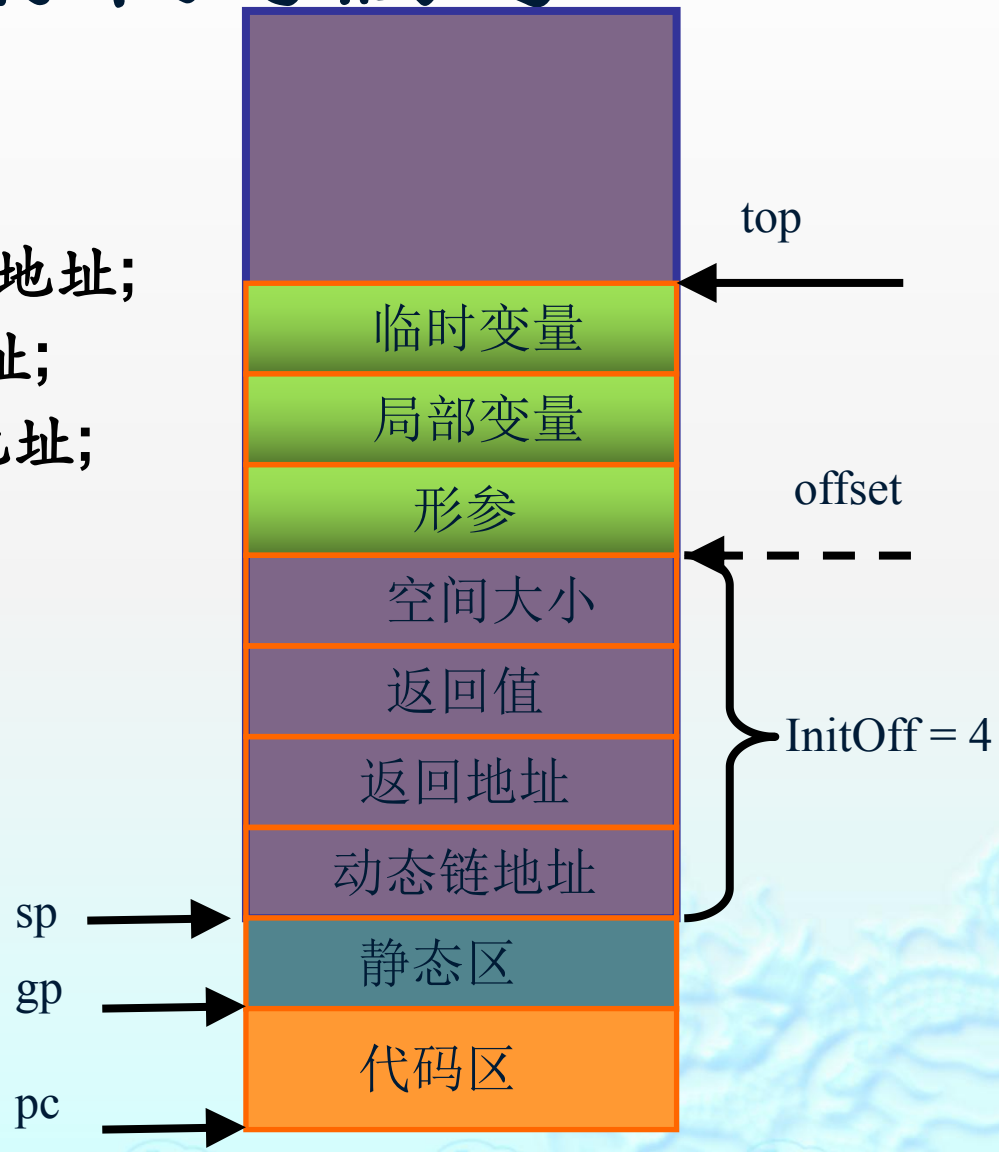
(ENDFUNC, _, _ , _)

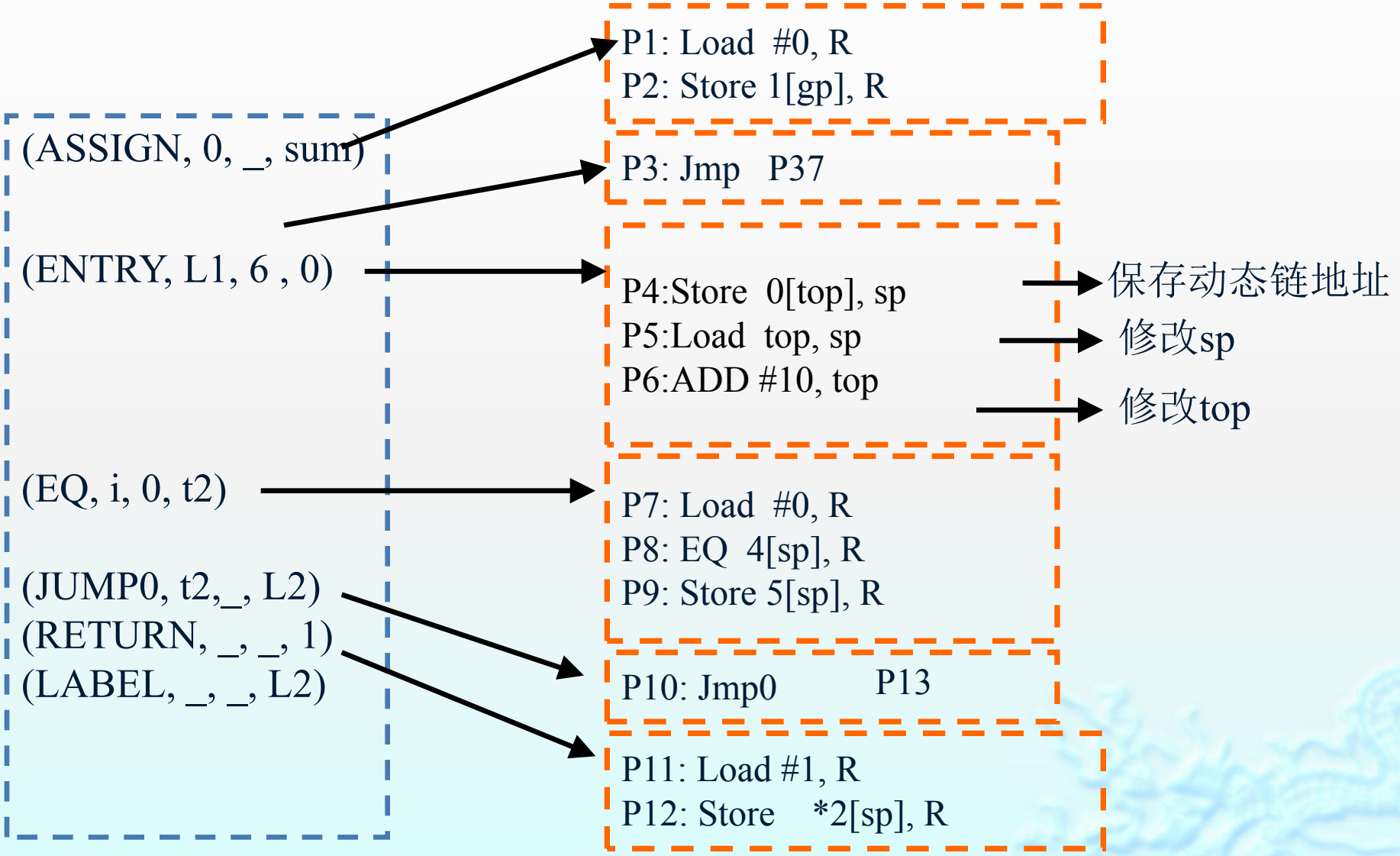
(ENTRY, L4, 1 , 0)
(VALACT,n,0, 1)
(CALL, L_{fac}, true, t1)
(ASSIGN, t1, _, sum)

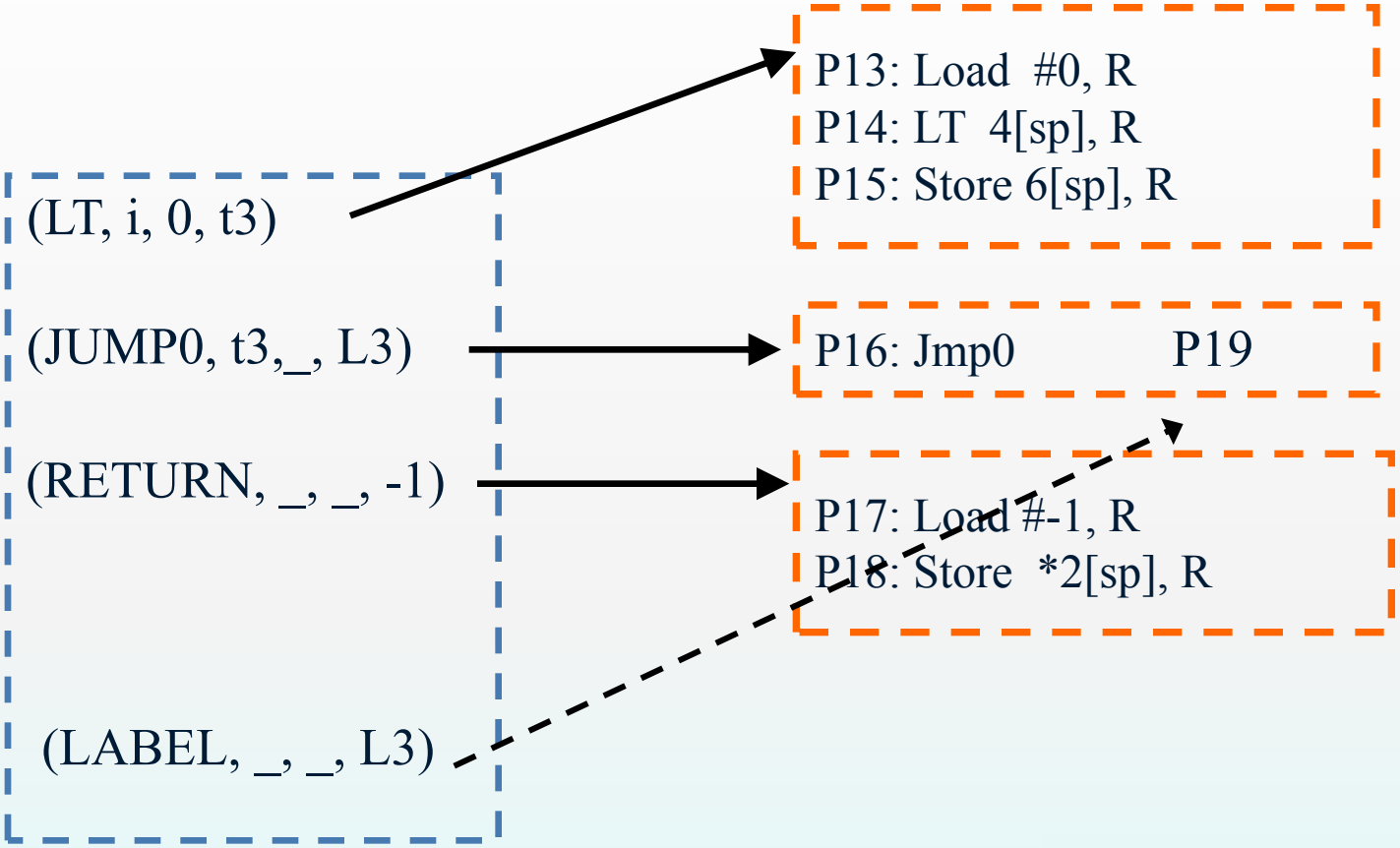
(ENDFUNC, _, _ , _)

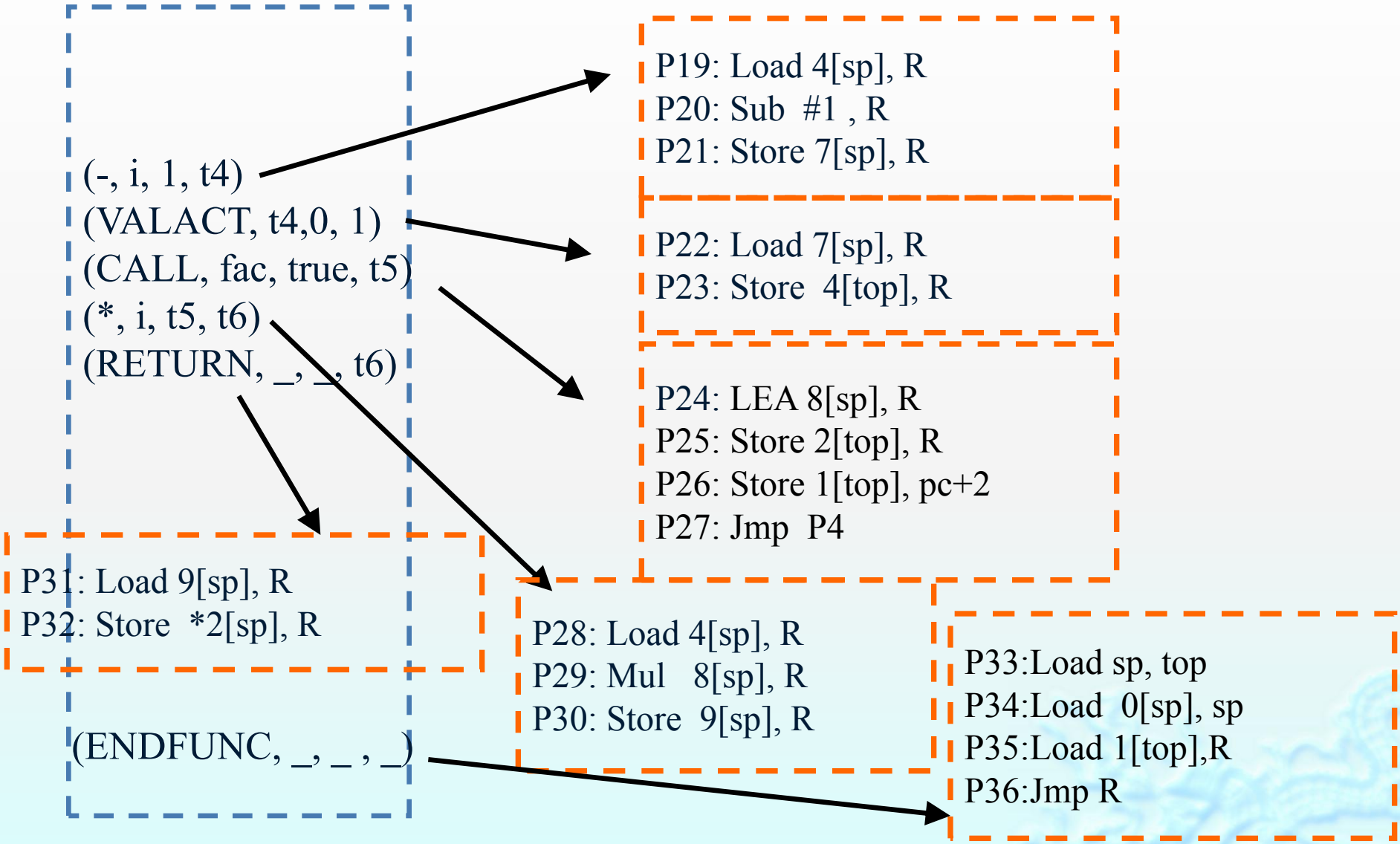
关于目标机的假定

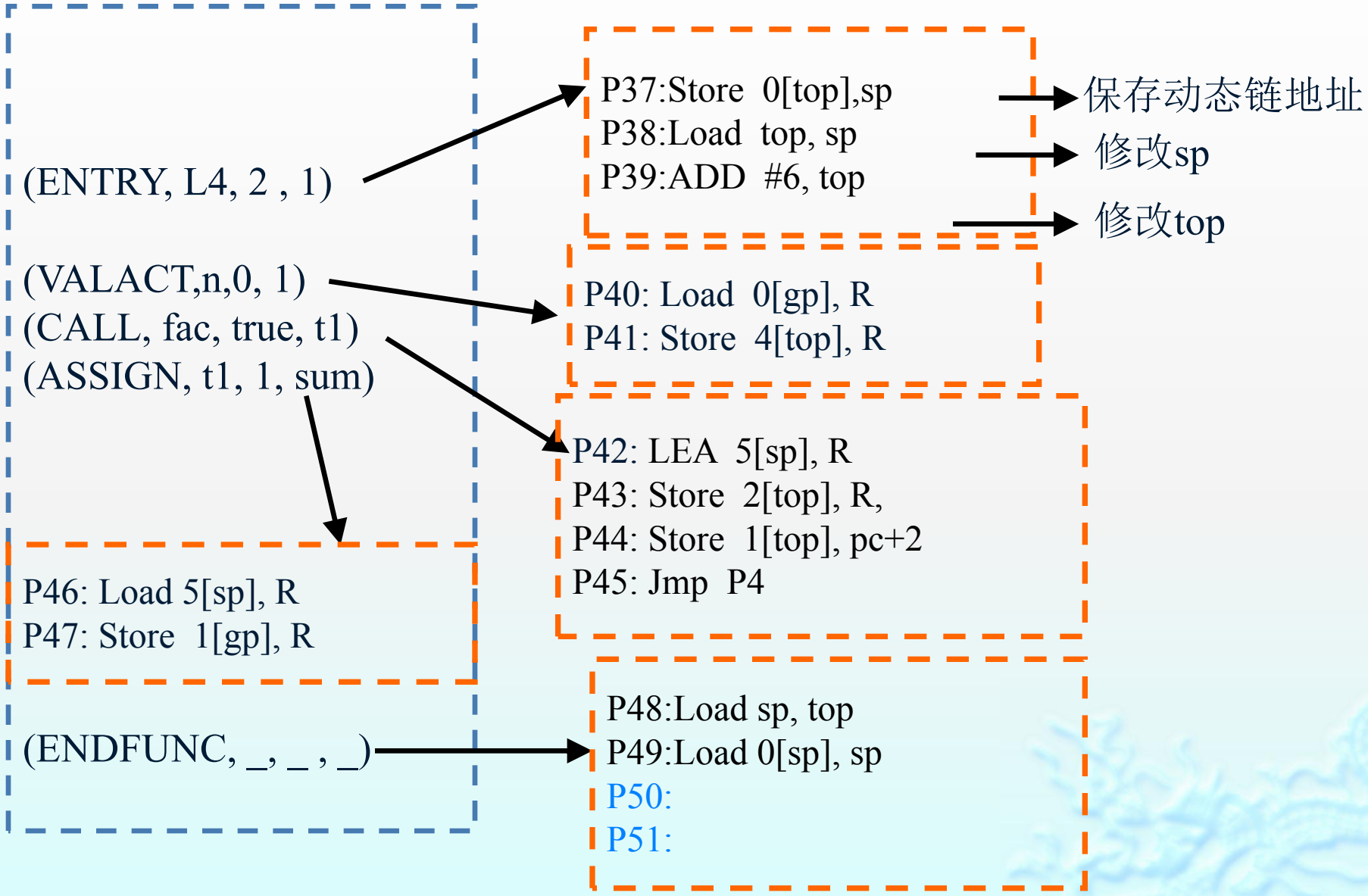
- 四个寄存器:
  - **sp :保存当前AR的首地址;**
  - **top:保存当前栈顶地址;**
  - **gp:保存静态区的首地址;**
  - **pc: 程序计数器;**
- AR的结构:




(ASSIGN, 0, _, sum)
(ENTRY, L1, 6 , 0)
(EQ, i, 0, t2)
(JUMP0, t2,_, L2)
(RETURN, _, _, 1)
(LABEL, _, _, L2)
P1: Load #0, R
P2: Store 1[gp], R
P3: Jmp P37
P4:Store 0[top], sp
P5:Load top, sp
P6:ADD #10, top
保存动态链地址
修改sp
修改top
P7: Load #0, R
P8: EQ 4[sp], R
P9: Store 5[sp], R
P10: Jmp0 P13
P11: Load #1, R
P12: Store *2[sp], R


(LT, i, 0, t3)
(JUMP0, t3,_, L3)
(RETURN, _, _, -1)
(LABEL, _, _, L3)
P13: Load #0, R
P14: LT 4[sp], R
P15: Store 6[sp], R
P16: Jmp0 P19
P17: Load #-1, R
P18: Store *2[sp], R


(-, i, 1, t4)
(VALACT, t4,0, 1)
(CALL, fac, true, t5)
(*, i, t5, t6)
(RETURN, _, _, t6)
(ENDFUNC, _, _ , _)
P19: Load 4[sp], R
P20: Sub #1 , R
P21: Store 7[sp], R
P22: Load 7[sp], R
P23: Store 4[top], R
P24: LEA 8[sp], R
P25: Store 2[top], R
P26: Store 1[top], pc+2
P27: Jmp P4
P28: Load 4[sp], R
P29: Mul 8[sp], R
P30: Store 9[sp], R
P31: Load 9[sp], R
P32: Store *2[sp], R
P33:Load sp, top
P34:Load 0[sp], sp
P35:Load 1[top],R
P36:Jmp R


(ENTRY, L4, 2 , 1)
(VALACT,n,0, 1)
(CALL, fac, true, t1)
(ASSIGN, t1, 1, sum)
P46: Load 5[sp], R
P47: Store 1[gp], R
(ENDFUNC, _, _ , _)
P37:Store 0[top],sp
P38:Load top, sp
P39:ADD #6, top
保存动态链地址
修改sp
修改top
P40: Load 0[gp], R
P41: Store 4[top], R
P42: LEA 5[sp], R
P43: Store 2[top], R,
P44: Store 1[top], pc+2
P45: Jmp P4
P48:Load sp, top
P49:Load 0[sp], sp
P50:
P51:

10.4 目标代码生成时要考虑的问题

- 寄存器的分配
  - 寄存器优先原则
  - 寄存器多载原则
  - 寄存器活跃原则
- 临时变量地址的共享
  - 临时变量数量多，寿命短
  - 中间代码不优化，则临时变量只定义一次，使用一次

考虑寄存器分配的代码生成

(+, A, B, T)	B不在寄存器 且不活跃	B不在寄存器 且活跃	B在寄存器R_B
A不在寄存器	Load B, R_B Add A, R_B	Load A, R_A Load B, R_B Add R_B, R_A	Load A, R_A Add R_B, R_A
A在寄存器R_A	Add B, R_A	Load B, R_B Add R_B, R_A	Add R_B, R_A