中山大學
SUN YAT-SEN UNIVERSITY

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

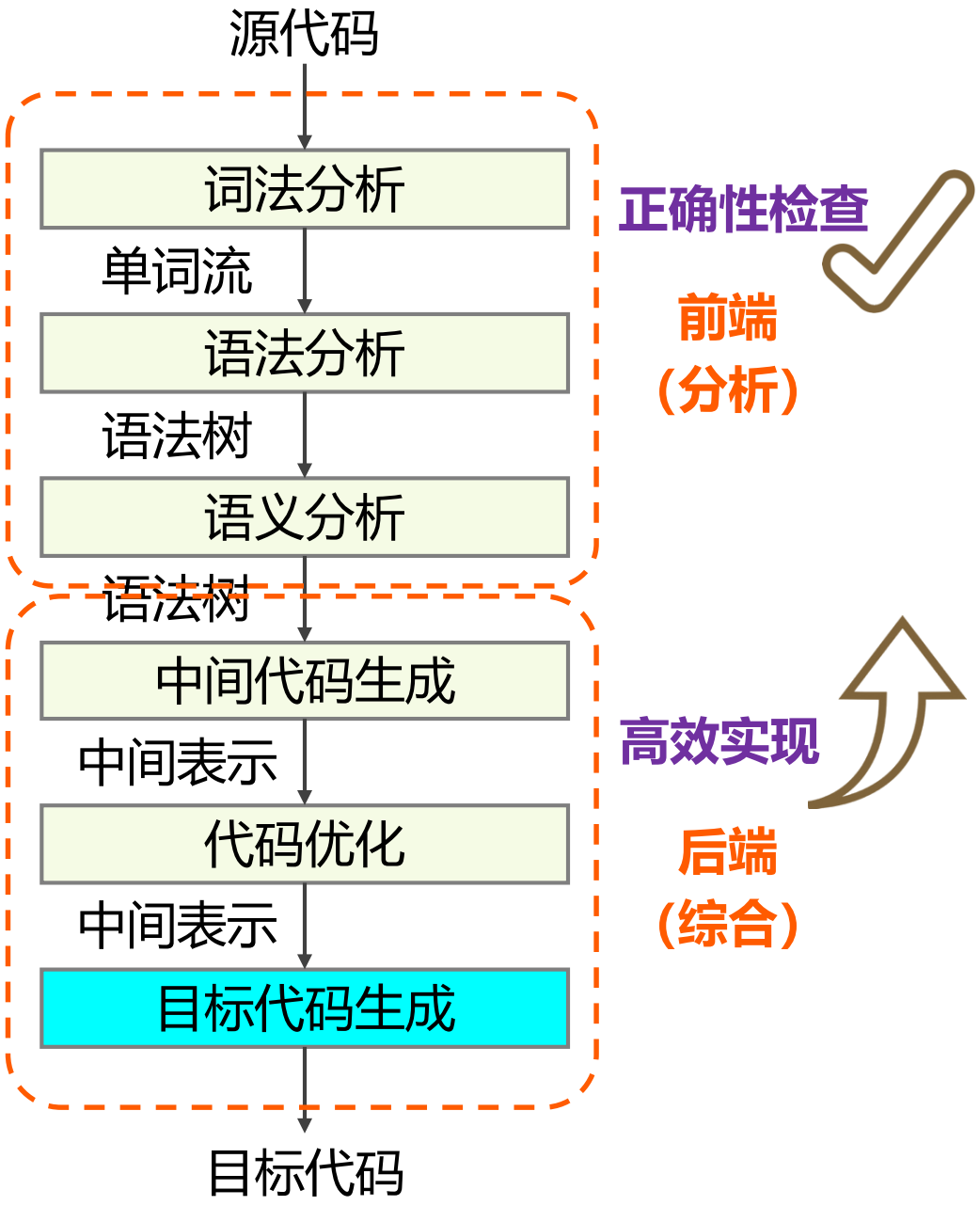
DCS290

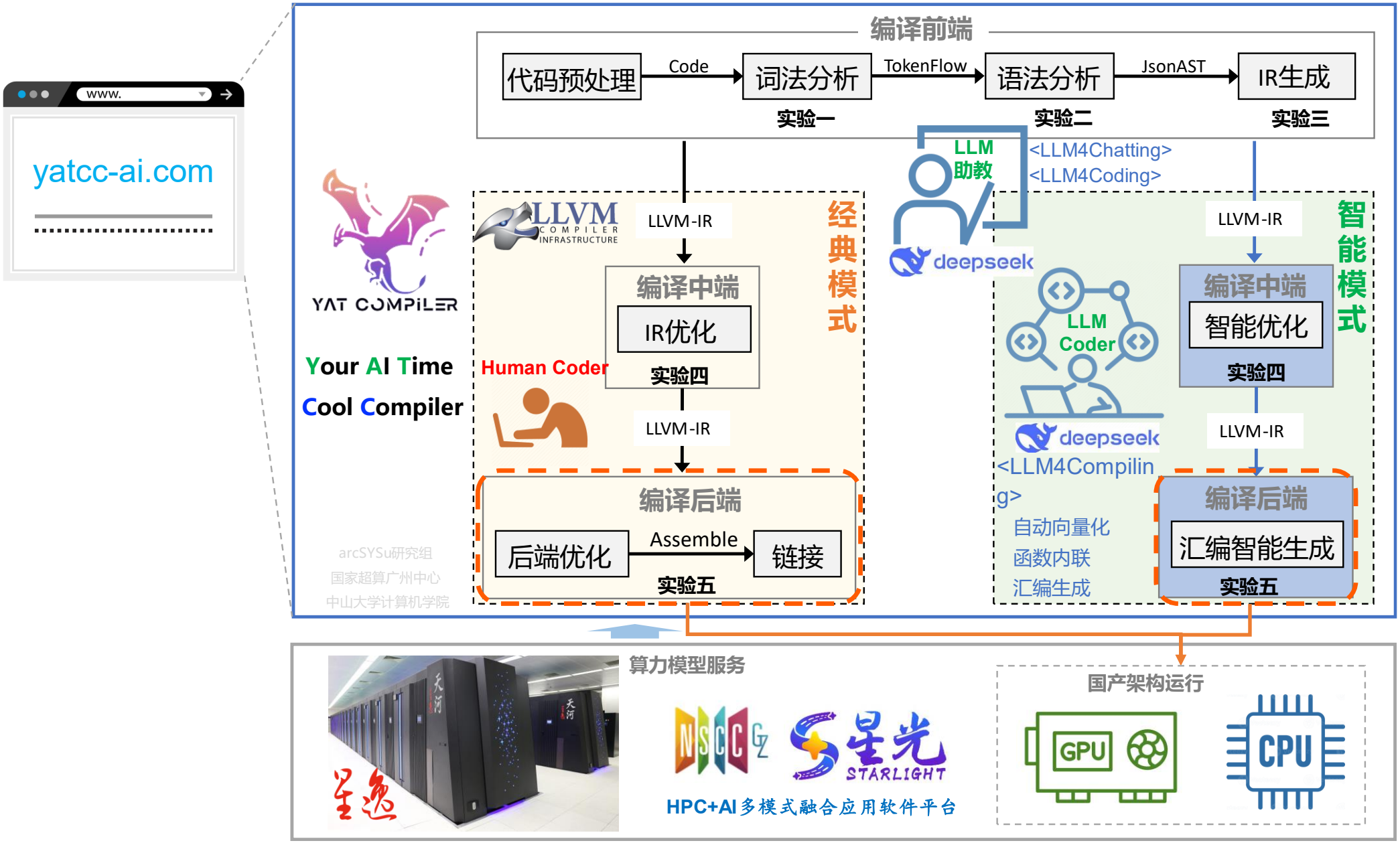
Compilation Principle

编 译 原 理

第八章 目标代码生成（1）

郑馥丹
zhengfd5@mail.sysu.edu.cn


源代码
词法分析
单词流
语法分析
语法树
语义分析
语法树
中间代码生成
中间表示
代码优化
中间表示
目标代码生成
目标代码
正确性检查
前端
（分析）
高效实现
后端
（综合）


www.
yatcc-ai.com
编译前端
代码预处理
Code
词法分析
TokenFlow
语法分析
JsonAST
IR生成
实验一
实验二
实验三
LLM
助教
<LLM4Chatting>
<LLM4Coding>
YAT COMPILER
Your AI Time
Cool Compiler
LLVM
COMPILER
INFRASTRUCTURE
LLVM-IR
经典模式
编译中端
IR优化
实验四
Human Coder
LLVM-IR
编译后端
后端优化
Assemble
链接
实验五
deepseek
LLM
Coder
deepseek
LLVM-IR
智能模式
编译中端
智能优化
实验四
LLVM-IR
<LLM4Compiling>
自动向量化
函数内联
汇编生成
编译后端
汇编智能生成
实验五
arcSYSu研究组
国家超算广州中心
中山大学计算机学院
算力模型服务
NSCCGZ
星光
STARLIGHT
HPC+AI多模式融合应用软件平台
国产架构运行
GPU
CPU

CONTENTS

目 录

0. 代码生成

- **为特定机器产生目标代码**（e.g., 汇编）
  - 输入：(优化的)IR，输出：目标代码
  - 寄存器分配：放置频繁访问数据
  - 指令选取：确定机器指令实现IR操作
  - 进一步的机器有关优化
    - ✓ 如：寄存器及访存优化

```
void main()
{
    int arr[10], i, x = 1;

    for (i = 0; i < 10; i++)
        arr[i] = x * 5;
}
```

```
14:   8b 55 f8          mov    -0x8(%rbp),%edx          // edx = x
17:   89 d0             mov    %edx,%eax                // eax = x
19:   c1 e0 02          shl    $0x2,%eax                // eax  = (x << 2)
1c:   01 c2             add    %eax,%edx                // edx = (x << 2) + x
1e:   8b 45 fc          mov    -0x4(%rbp),%eax          // eax = i
21:   48 98             cltq
23:   89 54 85 d0       mov    %edx,-0x30(%rbp,%rax,4)  // arr[i] = 5x
27:   83 45 fc 01       addl   $0x1,-0x4(%rbp)          // i++
2b:   83 7d fc 09       cmpl   $0x9,-0x4(%rbp)          // i <= 9
2f:   7e e3             jle    14 <main+0x14>           // loop end?
```

0. 代码生成

- 代码生成的任务
  - 把中间代码（经过优化或未经过优化）作为输入，将其转换成特定机器的机器语言或汇编语言作为输出，这样的转换程序称为**代码生成器**(Code Generator)。
- 目标代码生成需要考虑的基本问题：
  - 如何使生成的目标代码较短**（占空间小）**
  - 如何充分利用计算机的寄存器，减少目标代码访问存储单元的次数**（速度快，时间短）**

0. 代码生成

- 讨论目标代码生成的一些共同的问题，而不讨论某个特定机器的代码生成问题
  - 寄存器分配算法
    - ✓ 目标代码的执行效率很大程度依赖于**寄存器**的使用**（寄存器速度比存储器快很多）**
  - 基本块的代码生成算法
    - ✓ 寄存器分配算法仅限定在一个**基本块**的范围内，以四元式的**中间代码**作为输入，以**汇编语言**作为输出

0. 代码生成

- 代码生成器主要关注
  - **指令选择[Instruction selection]**
    - ✓选择合适的目标机器指令来实现中间表示（IR）语句
  - **寄存器分配与指派[Register allocation and assignment]**
    - ✓充分利用寄存器（最快的计算单元）
  - **指令排序[Instruction ordering]**
    - ✓决定指令执行的顺序
    - ✓也称为指令调度**[Instruction scheduling]**

1. 指令选择

- 生成的代码的速度和大小成本都需要考虑——trade off
- 例1：a=a+1
  - 方案1（3条指令）：
    - ✓ LD R0, a
    - ✓ ADD R0, R0, #1
    - ✓ ST a, R0

    更通用，但代码较长，内存访问开销大，效率较低
  - 方案2（1条指令）：
    - ✓ INC a　　//直接对内存中的a执行自增

    代码简洁，但某些架构可能不支持内存直接操作（依赖硬件支持）

1. 指令选择

- 生成的代码的速度和大小成本都需要考虑——trade off
- 例2：将寄存器R0设置为0（寄存器清零）
  - 方案一：LD R0, #0
    - ✓ 将立即数0加载到R0
    - ✓ 无需内存访问，简单直接
    - ✓ 须编码操作码LD、目标寄存器R0和立即数0
  - 方案二：XOR R0, R0
    - ✓ 通过异或操作（R0 ^ R0）清零
    - ✓ 无需内存访问，且代码体积更小（只需编码操作码和寄存器）
- 选择依据：利用**硬件特性**、根据**目标机器的指令集**选择最优实现

2. 寄存器分配与指派

- 目的：高效利用寄存器资源，减少内存访问开销
- 包含两个子问题
  - 寄存器分配（Register Allocation）
    - ✓ 选择在程序的每个点上哪些变量将驻留在寄存器中
  - 寄存器指派（Register Assignment）
    - ✓ 为变量分配具体的寄存器
- **这是代码生成中的一个难题**
  - 寄存器数量有限
  - 最优分配算法复杂
  - 不同架构的寄存器约定（如调用约定、保留寄存器）不同

3. 指令调度

- 对现代流水线处理器至关重要
- 指令调度的核心目标
  - 通过重新排列指令顺序，最大化利用CPU流水线，减少因依赖关系导致的停顿（Bubble），提升指令级并行性（ILP）
- 数据冲突（流水线中的停顿）：尝试在数据尚未就绪于寄存器前使用它
  - 写入寄存器的值在其后几个时钟周期内不可用
  - 后续周期应被不依赖该值的指令占用
  - 否则这些周期将被浪费

- 优化方法：将无依赖关系的指令聚类执行，同时保持原始语义

CONTENTS

目 录

1. 指令集[Instruction Set]

- RISC[Reduced Instruction Set Computer，精简指令集计算机]架构支持以下指令：
  - LD dst, addr　　// 加载（从内存addr读取数据到目标寄存器dst）
  - ST x, Ri　　// 存储（将寄存器数据写入内存）
  - OP dst, src1, src2 // 二元运算（如加、减等）
  - OP dst, src1　　// 一元运算
  - BR L　　// 无条件跳转（跳转到标签L）
  - Bcond r, L　　// 条件跳转（根据寄存器r的值决定是否跳转到标签L）

2. 寻址模式[Addressing Modes]

- 指令中的寻址方式：
  - #C // 立即数常量，开销=1
  - x // 绝对地址，开销=1
  - *x // 间接内存寻址，开销=1
  - R // 直接寄存器寻址，开销=0
  - *R // 间接寄存器寻址，开销=0

2. 寻址模式[Addressing Modes]

- 指令中的寻址方式：
  - a(Ri) // 直接变址寻址，开销=1
    - ✓ a为变量或常量
    - ✓ 示例：LD R1, a(R2)

      含义：R1 = contents(a + contents(R2))

      contents(x)表示x所代表的寄存器或内存位置中存放的内容
    - ✓ 示例：LD R1, 100(R2)

      含义：寄存器R2中的值加上100得到的和所指向的位置中的内容加载到R1中

2. 寻址模式[Addressing Modes]

- 指令中的寻址方式：
  - *a(Ri) // 间接变址寻址，开销=1
    - ✓ Ri中的值加上a的和所代表的位置上的内容所代表的位置中的值
    - ✓ 示例：LD R1, *100(R2)

      含义：寄存器R2中的值加上100得到的和所指向的位置中的内容所代表的位置中的值加载到R1中

2. 寻址模式[Addressing Modes]

- 例:
  - x=y-z
    - ✓ LD R1, y　　// R1=y
    - ✓ LD R2, z　　// R2=z
    - ✓ SUB R1, R1, R2　　// R1=R1-R2
    - ✓ ST x, R1　　// x=R1
  - b=a[i]　// 假设a是一个元素为8字节的数组，下标从0开始
    - ✓ LD R1, i　　// R1=i
    - ✓ MUL R1, R1, 8　　// R1=R1*8
    - ✓ LD R2, a(R1)　　// R2=contents(a+contents(R1))
    - ✓ ST b, R2　　// b=R2

2. 寻址模式[Addressing Modes]

- 例:
  - a[j]=c
    - ✓ LD R1, c
    - ✓ LD R2, j
    - ✓ MUL R2, R2, 8
    - ✓ ST a(R2), R1 // contents(a+contents(R2))=R1
  - x=*p
    - ✓ LD R1, p
    - ✓ LD R2, 0(R1) // R2=contents(0+contents(R1))
    - ✓ ST x, R2

2. 寻址模式[Addressing Modes]

- 例:
  - *p=y
    - ✓ LD R1, p
    - ✓ LD R2, y
    - ✓ ST 0(R1), R2 // contents(0+contents(R1))=R2
  - if x<y goto L
    - ✓ LD R1, x
    - ✓ LD R2, y
    - ✓ SUB R1, R1, R2
    - ✓ BLTZ R1, M // if R1<0 jump to M, M是标号为L的三地址码的第一个指令的标号

随堂练习（1）

(1) x=b*c, y=a+x

(2) 假设a和b是元素为4字节值的数组：x=a[i], y=b[j], z=x*y

(3) y=*q, q=q+4, *p=y, p=p+4

(4) if x<y goto L1

z=0

goto L2

L1: z=1

随堂练习（1）

(1) x=b*c, y=a+x

```
LD R1, b          // 将b的值加载到R1
LD R2, c          // 将c的值加载到R2
MUL R1, R1, R2    // R1 = b * c
ST x, R1          // 将结果存储到x
LD R2, a          // 将a的值加载到R2
ADD R2, R2, R1    // R2 = a + x
ST y, R2          // 将结果存储到y
```

随堂练习（1）

(2) 假设a和b是元素为4字节值的数组：x=a[i], y=b[j], z=x*y

```
LD R1, i            // 将i加载到R1
MUL R1, R1, 4       // 乘以元素大小(4字节)
LD R2, a(R1)        // 加载a[i]的值
ST x, R2            // 存储到x
LD R1, j                        LD R1, x
MUL R1, R1, 4                   LD R2, y
LD R2, b(R1)                    MUL R1, R1, R2
ST y, R2                        ST z, R1
```

随堂练习（1）

(3) y=*q, q=q+4, *p=y, p=p+4

```
LD R1, q          // 加载q指针值
LD R2, 0(R1)      // 引用q，加载*q到R2
ST y, R2          // 存储到y
ADD R1, R1, 4     // 指针加4
ST q, R1          // 存储更新后的q
LD R1, p          // 加载p指针值
LD R2, y          // 加载y的值
ST 0(R1), R2      // 将y存储到*p
ADD R1, R1, 4     // 指针加4
ST p, R1          // 存储更新后的p
```

随堂练习（1）

(4) if x<y goto L1

z=0

goto L2

L1: z=1

```
        LD R1, x          // 加载x
        LD R2, y          // 加载y
        CMP R1, R2        // 比较x和y
        BLT L1            // 如果x < y，跳转到L1
        LD R1, 0          // R1 = 0
        ST z, R1          // z = 0
        BR L2             // 无条件跳转到L2
L1:     LD R1, 1          // R1 = 1
        ST z, R1          // z = 1
L2:     ...
```

3. 程序和指令的代价[Cost]

- 一条指令的代价 = 1 + 操作数寻址代价
- 一个程序的代价 = 所有指令代价的总和

- 示例
  - LD R0, R1　　// 代价= 1
  - LD R0, x　　// 代价= 2
  - LD R1, *100(R2)　　// 代价= 2

随堂练习（2）

- 确定下列指令序列的代价

(1)

LD R0, y	2
LD R1, z	2
ADD R0, R0, R1	1
ST x, R0	2

总代价为7

(2)

LD R0, i	2
MUL R0, R0, 8	2
LD R1, a(R0)	2
ST b, R1	2

总代价为8

CONTENTS

目 录

1. 静态分配[Static Allocation]

- call *callee*的实现
  - call表示过程调用
  - *callee*是被调用的过程/函数
  - ST *callee*.staticArea, #here + 20　　// 将返回地址保存到*callee*的活动记录的开始处*callee*.staticArea；// #here：当前指令的地址// +20：假设调用序列需占用20字节
  - BR *callee*.codeArea　　// 跳转到被调用过程*callee*的目标代码上
- return的实现
  - BR **callee*.staticArea　　//跳转到保存*callee*的活动记录开始位置的地址上

1. 静态分配[Static Allocation]

- 例:
  - // c的代码
  - action1
  - call p
  - action2
  - halt
  - // p的代码
  - action3
  - return

假定:

- c的代码从地址100开始
- p的代码从地址200开始
- 每个action伪指令占用20个字节
- 这些过程的活动记录以静态方式分配，地址分别是300和364

1. 静态分配[Static Allocation]

- 静态分配的目标代码

```
// c的代码
action1
call p
action2
halt

// p的代码
action3
return
```

```
// c的代码      100: action1
                120: ST 364, #140   // 在位置364上存放返回地址140
                132: BR 200         // 调用p
                140: action2
                160: HALT           // 返回操作系统
                ...
// p的代码      200: action3
                220: BR *364        // 返回在位置364保存的地址处
                ....
// 300-363存放c的活动记录
                300:
                ....
// 364-451存放p的活动记录
                364: [140]
                ....
```

2. 栈分配[Stack Allocation]

- 初始化
  - LD SP, #stackStart　　// 初始化栈指针
    // 在寄存器SP中存放栈指针
  - …　　// main()函数的代码
  - HALT　　// 程序终止
- call *callee*的实现
  - ADD SP, SP, #caller.recordSize　// 扩展栈空间
  - ST 0(SP), #here + 16　　// 保存返回地址
  - BR callee.codeArea　　// 跳转到被调用程序
  - SUB SP, SP, #caller.recordSize // 恢复栈指针
- return的实现
  - BR *0(SP)　　// 返回到调用者

2. 栈分配[Stack Allocation]

- 例:

```
// m的代码
action1
call q
action2
halt
// p的代码
action3
return
```

```
// q的代码
action4
call p
action5
call q
action6
call q
return
```

2. 栈分配[Stack Allocation]

- 栈分配的目标代码

```
// m的代码
action1
call q
action2
halt
// p的代码
action3
return
```

```
100 LD SP, #600
108 action1
128 ADD SP, SP, #msize
136 ST *SP, #152
144 BR 300
152 SUB SP, SP, #msize
160 action2
180 HALT
...
200 action3
220 BR *0(SP)
....
300 action4
320 ADD SP, SP, #qsize
328 ST *SP, #344
336 BR 200
344 SUB SP, SP, #qsize
352 action5
372 ADD SP, SP, #qsize
380 ST *SP, #396
388 BR 300
396 SUB SP, SP, #qsize
404 action6
424 ADD SP, SP, #qsize
432 ST *SP, #448
440 BR 300
448 SUB SP, SP, #qsize
456 BR *0(SP)
...
600
```

```
// q的代码
action4
call p
action5
call q
action6
call q
return
```