

# 6.035 2002年秋

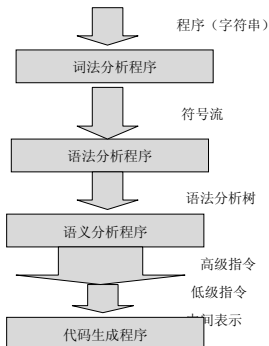
讲座9: 未优化代码生成

从中间表达到其码

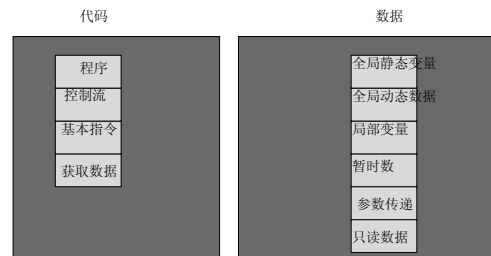
## 第四阶段路线图

- 检查点
  - 10月22日星期二
  - 将已完成的打包为一个文件提交
  - 使用代码生成器生成的代码无效
  - 如果你最后还有很多问题, 而且在检查点前并没有做足够的工作, 我们会对你非常严厉的
- 完成时间10月31日
- 论文讨论
  - Amarasinghe 教授                      下周一 (17日)
  - Rinard教授                                下周五 (21日)

## 计算机解析原理



## 高级语言的组成

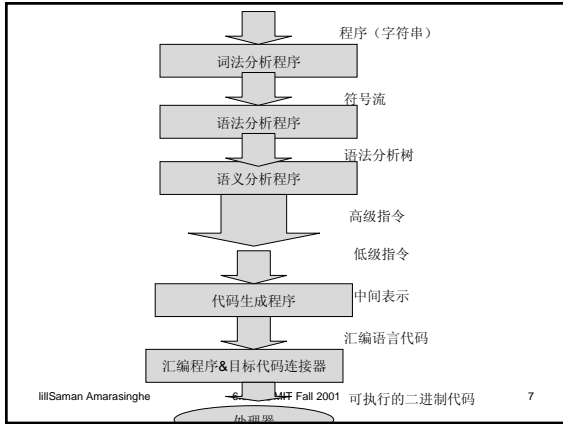


## 机器代码生成器功能

- 将中间表达的所有指令翻译成汇编语言
- 为变量、数组等定位存储位置
- 坚持调用原则
- 创建必要的符号信息

## 机器所能理解的

位置	数据	编译指令
0046	8B45FC	movl -4(%ebp), %eax
0049	48C3F0	movlq %eax,%eax
004c	8B45FC	movl -4(%ebp), %eax
004f	48C3D0	movlq %eax,%edx
0052	8B45FC	movl -4(%ebp), %eax
0055	4999	cltq
0057	8B44500	movl B(,%eax,4), %eax
005e	000000	movl 0, %eax
0065	8B45FC	movl -4(%ebp), %eax
0067	8B45FC	movl -4(%ebp), %eax
006a	4999	cltq
006c	8B45FC	movl -4(%ebp), %eax
006e	8B45FC	movl -4(%ebp), %eax
0075	8B45FC	movl -4(%ebp), %eax
0078	48C3C8	movlq %eax,%edx
007b	8B45FC	movl -4(%ebp), %eax
007e	4999	cltq
0080	8B44500	movl B(,%eax,4), %eax



## 汇编语言

- 优势
  - 由于符号指令和符号名的使用, 简化了代码生成
  - 逻辑抽象层
  - 用一个汇编语言代码可以描述多种结构
    - ⇒ 可修改编译器的执行过程
    - 汇编语言宏指令
- 劣势
  - 增加了汇编和连接的处理过程
  - 汇编程序本身增加了系统开销

MIT Fall 2001 6.035 @MIT Fall 2001 8

## 汇编语言

- 可重新定位的机器语言 (目标模块)
  - 所有定位 (地址) 都由符号表示
  - 在链接和载入时间上和存储器地址成映射关系
  - 可独立编辑的弹性
- 绝对机器语言
  - 地址为硬代码
  - 简单和直向前的实现方式
  - 无弹性——重新载入生成的代码非常困难
  - 适用于中断操作和设备驱动

MIT Fall 2001 6.035 @MIT Fall 2001 9

## 汇编程序示例

```

0000 6572726F7200 .LC0: .section .rodata
                                .string "error"
                                .text
                                .globl fact
                                fact:
0000 55                pushq %rbp
0001 4889E5            movq %rbp, %rbp
0004 4883EC10          subq $16, %rsp
0008 897DFC            movl %edi, -4(%rbp)
000b 837DFC00          cmpl $0, -4(%rbp)
000f 7911              jns .L2
0011 E900000000      movl $, .LC0, %edi
0016 E900000000      movl $0, %eax
001b E900000000      call printf
0020 EB22              jmp .L3
0022 837DFC00          .L2: cmpl $0, -4(%rbp)
0026 7509              jne .L4
0028 C745F801000000 movl $1, -8(%rbp)
002f EB13              jmp .L3
0031 8B7DFC            .L4: movl -4(%rbp), %edi
0034 FFCF            decl %edi
0036 E900000000      call fact
003b 02A945FC        mulb -4(%rbp), %eax
003f 8945F8            movl %eax, -8(%rbp)
0042 EB00              jmp .L1
0044 8B45F8            .L3: movl -8(%rbp), %eax
0047 C9              leave
0048 C3              ret

```

MIT Fall 2001 6.035 @MIT Fall 2001 10

## 目标文件的构成

- 我们使用ELF文件格式
- 目标文件有:
  - 多节
  - 符号信息
  - 重新定位信息
- 每节:
  - 全局偏移表
  - 过程连接表
  - 文本 (代码)
  - 数据
  - 只读数据

```

.file "test.c" .rodata
.LC0: .section "error %d"
      .section .text
      .globl fact
      fact:
      pushq %rbp
      movq %rbp, %rbp
      subq $16, %rsp
      movl -8(%rbp), %eax
      leave
      ret
      .comm bar,4,1
      .comm a,1,1
      .comm b,1,1
      .section eh_frame,"a",@progbits
      .long 0x0
      .byte 0x1
      .string ""
      .uleb128 0x1

```

MIT Fall 2001 6.035 @MIT Fall 2001 11

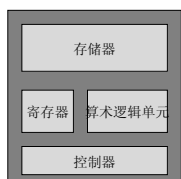
## 现代处理器的总体结构

- 算术逻辑单元
- 控制器
- 存储器
- 寄存器

MIT Fall 2001 6.035 @MIT Fall 2001 12

## 算术逻辑单元

- 执行大多数的数据操作
- 形式为  
`OP<oprnd1>,<oprnd2>`  
`--<oprnd2>=<oprnd1>OP<oprnd2>`  
 或者  
`OP<oprnd1>`
- 操作数有:
  - 立即数        \$25
  - 寄存器值     %rax
  - 存储器值     4 (%rbp)
- 操作行为有:
  - 算法操作 (add,sub,mul)
  - 逻辑操作 (and,sal)
  - 独立数操作 (inc,dec)



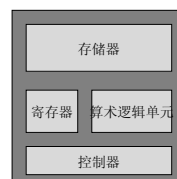
lllSaman Amarasinghe

6.035 @MIT Fall 2001

13

## 算术逻辑单元

- 许多算法可能导致出错
  - 上溢出或下溢出
- 可以操作不同的数据类型
  - `addb` 8bits
  - `addw` 16bits
  - `addl` 32bits
  - `addq` 64bits(Decaf全是64位的)
  - 有符号数和无符号数的计算
  - 浮点数的计算 (独立的ALU)



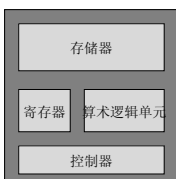
lllSaman Amarasinghe

6.035 @MIT Fall 2001

14

## 控制器

- 处理指令的先后顺序
- 执行指令
  - 所有的指令都存储在存储器中
  - 取出指针所指向的指令并执行指令
  - 对于一般指令来说, 增加指针的值时期指向存储器中的下一个位置



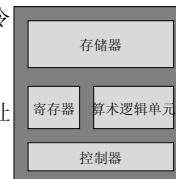
lllSaman Amarasinghe

6.035 @MIT Fall 2001

15

## 控制器

- 无条件分支 (unconditional branches)
  - 从另外一个位置取出下一条指令
  - 无条件跳转到指定地址  
`jmp .L32`
  - 无条件跳转到某一寄存器的地址  
`jmp %rax`
  - 响应处理器的调用  
`call fact   call%r11`



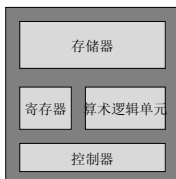
lllSaman Amarasinghe

6.035 @MIT Fall 2001

16

## 控制器

- 所有的算术操作都会升级条件代码 (rFLAGS)
- 比较操作清除的设置了rFLASS
  - `Cmp $0,%rax`
- 通过rFLAGS有条件跳转
  - `Jxx .L32 Jxx 4(%rbp)`
  - 示例
    - `JO` 溢出跳转
    - `JC` 有进位则跳转
    - `JAE` 大于或等于则跳转
    - `JZ` 等于零则跳转
    - `JNE` 不相等则跳转



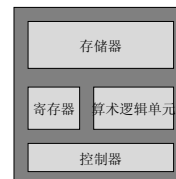
lllSaman Amarasinghe

6.035 @MIT Fall 2001

17

## 控制器

- 只有在特殊情况下才转移控制器 (很少情况)
  - 陷入和异常
  - 行为
    - 保存下一个 (或当前) 指令位置
    - 找到跳转的地址 (从异常向量中)
    - 跳转到那个地址



lllSaman Amarasinghe

6.035 @MIT Fall 2001

18

## 何时使用

- 请给出可以使用以下分支指令的例子
  1. jmp L0
  2. call L1
  3. jmp %rax
  4. jz -4(%rbp)
  5. jne L1

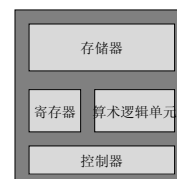
lllSaman Amarasinghe

6.035 @MIT Fall 2001

19

## 存储器

- 单层地址空间
  - 由字组成
  - 可按比特进行地址分配
- 要存储的数据类型
  - 程序
  - 局部变量
  - 全局变量和数据
  - 栈
  - 堆

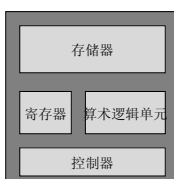
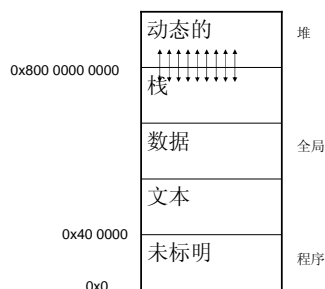


lllSaman Amarasinghe

6.035 @MIT Fall 2001

20

## 存储器



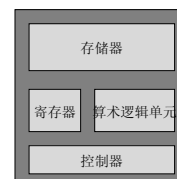
lllSaman Amarasinghe

6.035 @MIT Fall 2001

21

## 寄存器

- 指令只允许有限的存储器操作类型
  - `add -4(%rbp),8(%rbp)`
  - `mov -4(%rbp),%r10`
  - `add %r10,-8(%rbp)`
- 在性能表现方面有重要作用  
数量有限
- 特殊寄存器
  - %rbp     base pointer
  - %rsp     stack pointer



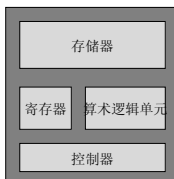
lllSaman Amarasinghe

6.035 @MIT Fall 2001

22

## 其它交互行为

- 其它操作
  - 输入/输出
  - 权限/安全操作
  - 操控特殊硬件
    - TLBs, 缓存等
- 大多通过系统调用
  - 汇编中hand-coded编码技术
  - 编译器将其视为一般函数的调用



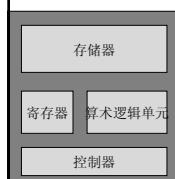
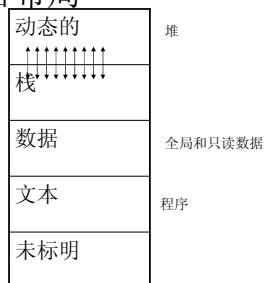
lllSaman Amarasinghe

6.035 @MIT Fall 2001

23

## 存储器布局

- 堆管理
  - 自由清单
- 从文本部分开始定位



6.035 @MIT Fall 2001

24

## 分配只读数据

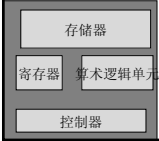
- 所有只读数据都存储在文本段
- 整数
  - 使用立即读取指令
- 字符串
  - 使用.string宏定义

```
.section .text
.globl main
main:
    enter    $0, $0
    movq    $5, x(%rip)
    push   x(%rip)
    push   $msg
    call   printf@GOTPLT
    add    $16, %rsp
    leave
    ret

.msg:
    .string "Five: %d\n"
```

6.035 @MIT Fall 2001

25



## 全局变量

- 分配: 使用编译器的.comm指令
- 使用电脑的间接寻址
  - %rip是当前指令的地址
  - X(%rip)会把从当前指令的位置到x在数据段中所占空间的偏移量加到%rip中
  - 创建容易重新定位的二进制数

```
section text
.globl main
main:
    enter    $0, $0
    movq    $5, x(%rip)
    push   x(%rip)
    call   printf@GOTPLT
    add    $16, %rsp
    leave
    ret

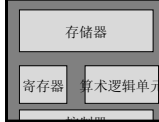
.comm    x, 8
```

.comm 命名、大小和边界基准

.comm指令在数据段分配存储空间。标志符用来指代这个存储空间。大小以比特

6.035 @MIT Fall 2001

26



## 过程抽象

- 需要全系统合作
  - 在存储器布局、保护、资源分配调用顺序&错误解决等方面要有广泛的一致
  - 必须包含结构 (ISA, Industry Standard Architecture 工业标准结构), 操作系统, &编译器
- 提供系统范围内的共享
  - 存储管理, 控制流, 中断
  - 输入/输出设备接口, 保护设备, 计时器、同步标记, 计数器.....
- 建立私有环境需求
  - 对每一个过程调用创建私有存储空间。
  - 压缩控制流&数据抽象信息

过程抽象是一个社会和约 (罗素)

lilSaman Amarasinghe

6.035 @MIT Fall 2001

27

## 过程抽象

- 实际上导致了:
  - 多过程
  - 库调用
  - 由多个编译器编译, 由不同的编程语言编写, 手写汇编
- 对于项目, 我们需要关注的是:
  - 参数传递
  - 寄存器
  - 堆
  - 调用协定

lilSaman Amarasinghe

6.035 @MIT Fall 2001

28

## 参数传递规则

- 多种传递方式
  - 参数调用
  - 值调用
  - 结果值调用

lilSaman Amarasinghe

6.035 @MIT Fall 2001

29

## 参数传递规则

```
Program {
    int A;
    foo(int B) {
        B = B + 1
        B = B + A
    }
    Main() {
        A = 10;
        foo(A);
    }
}
```

- 参数调用      A是???
- 值调用        A是???
- 结果值调用    A是???

lilSaman Amarasinghe

6.035 @MIT Fall 2001

30

## 参数传递规则

- 多种传递方法
  - 参数调用
  - 值调用
  - 结果值调用
- 如何传递参数
  - 通过堆栈
  - 通过寄存器
  - 或者两者混合
- 在Decaf语言调用协定中，所有的参数传递通过堆栈传递

## 寄存器

- 在程序调用中，如何处理活跃的寄存器？
  - 调用者保护
  - 被调用者保护

## 问题

- 下述两者的优势/劣势是：
  - 调用者保护寄存器？
  - 被调用者保护寄存器？
- 如果一半是调用者保护另一半是被调用保护，在调用和被调用过程中应该使用哪些寄存器？

## 寄存器

- 在过程调用中，如何处理活跃的寄存器？
  - 调用者保护
  - 被调用者保护
- 在这个部分，仅将寄存器作为短期存储数据的容器
  - 不能在过程间调用时仍然有效
  - 为了在第五部分的性能表现，将会开始保存数据

## 栈

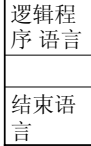
$8*n+16(\%rbp)$	参数n
	.....
$16(\%rbp)$	参数0
$8(\%rbp)$	返回地址
$0(\%rbp)$	前一个%rbp
$-8(\%rbp)$	本地0
	.....
$-8*m-8(\%rbp)$	本地m
$0(\%rsp)$	变量大小

## 问题

- 为什么使用栈？为什么不使用堆或者提前在数据段分配的空间？

# 过程链接

## • 标准过程链接

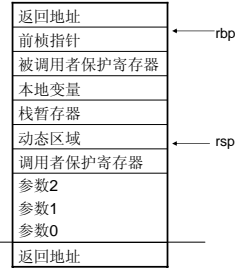


过程包括：  
 ·标准逻辑程序语言  
 ·标准结束语言  
 每部分都要包括  
 ·前调用序列  
 ·后返回序列

# 栈

## • 调用：调用者

- 假使%rcx为活跃的并且时调用保护的
- 调用foo(A,B,C)
  - A在-8(%rbp)
  - B在-16(%rbp)
  - C在-24(%rbp)



```
push    %rcx
push    -24(%rbp)
push    -16(%rbp)
push    -8(%rbp)
call    foo
add     $0, %rsp
```

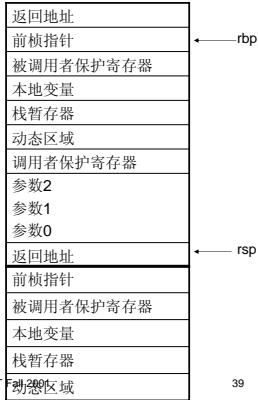
# 栈

## • 调用：被调用者

- 假使%rbx在函数中使用并且时被调用保护的
- 假使局部需要40比特空间

```
foo:
enter   $48, $0
```

```
mov    %rbx, -8(%rbp)
```



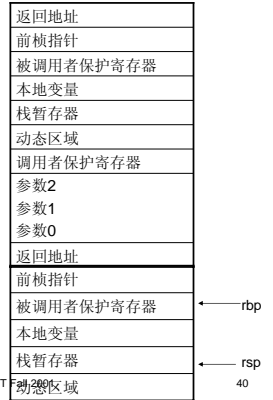
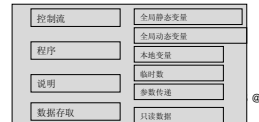
# 栈

## • 实参

## • 调用foo(A,B,C)

- 在调用前压栈
- 使用16+xx (%rbp)进行访问

```
mov    16(%rbp), %rax
mov    24(%rbp), %r10
```



# 栈

## • 局部和临时变量

- 计算大小并且在栈中分配空间
- 使用-8-xx(%rbx)进行访问

```
sub    $48, %rsp
or     enter    $48, 0
```

```
mov    -28(%rbx), %r10
mov    %r11, -20(%rbx)
```

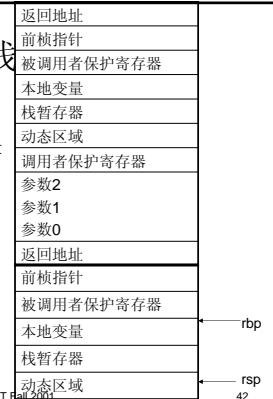


# 栈

## • 被调用返回

- 假使返回时第一个临时变量
- 回复调用者保护寄存器
- 将返回值保存到%rax
- 消除调用栈

```
mov    -8(%rbp), %rbx
mov    -16(%rbp), %rax
ret
```



## 栈

- 调用者返回
  - 假使返回值存入第一个临时变量
  - 恢复栈以重新定义实参空间
  - 恢复调用者保护寄存器

返回地址	
前帧指针	← rbp
被调用者保护寄存器	
本地变量	
栈暂存器	
动态区域	
调用者保护寄存器	← rsp
参数2	
参数1	
参数0	
返回地址	

```

call    foo
add     $24, %rsp
pop     %rax
mov     %rax, 8(%rbp)
...

```

lllSaman Amarasinghe      6.035 @MIT Fall 2001      43

## 问题

- 需要\$rbp么?
- 使用\$rbp的优缺点各是什么?

lllSaman Amarasinghe      6.035 @MIT Fall 2001      44

## 程序示例

```

program{
  int sum3d(int ax, int ay, int az)
  {
    int dx,dy,dz;
    if(ax>ay)
      dx=ax-bx;
    else
      dx=bx-ax;
    return dx+dy+dz;
  }
main(){
  int px,py,pz;
  px=10; py=20; pz=30;
  sum3d(px,py,pz);
}

```

返回地址	
前帧指针	← rbp
本地变量px (10)	
本地变量py (20)	
本地变量pz (30)	← rsp

lllSaman Amarasinghe      6.035 @MIT Fall 2001      45

## 代码生成的指导方针

- 缓慢的降低抽象层次
  - 只做很少的事情（或者只有一件事情），进行传递
    - 更易于分解项目，生成代码和调试
- 保持抽象层次一致
  - 中间表达需要在任何时候都有“正确”的语义
    - 至少你自己需要明确语义
  - 在传递间可能运行一些优化
- 灵活使用声明
  - 使用一个声明来检查假设

lllSaman Amarasinghe      6.035 @MIT Fall 2001      46

## 代码生成的指导方针

- 做最简单但最没有智能的事情
  - 产生如 $0+1*x+0*y$ 这样的代码是可以的
  - 代码也许看上去很别扭，但是会有助于优化
- 请明确希望可以变成现实在
  - 编译器的编译时间
  - 使用生成代码的运行时间

lllSaman Amarasinghe      6.035 @MIT Fall 2001      47

## 代码生成的指导方针

- 记住后面就是优化
  - 让优化器做优化工作
  - 请思考需要哪种优化器并且相应的规划代码结构
  - 例如：寄存器分配、代数简化、常数复制
- 建立一个好的测试基础
  - 衰退测试
    - 如果输入程序产生了一个缺陷，则把这个缺陷作为衰退测试用
  - 学习好的缺陷追踪过程
    - 例如：二进制搜索

lllSaman Amarasinghe      6.035 @MIT Fall 2001      48