

# 第四章 处理器体系结构

教师：夏 文

计算机科学与技术学院

哈尔滨工业大学（深圳）

# 本章概况

## ■ 背景

- 指令集 (4-1)
- 逻辑设计 (4-2)

## ■ 串行实现

- 简单但是速度较慢的处理器设计 (4-3)

## ■ 流水线

- 使得更多事务同时进行 (4-4)

## ■ 流水线实现

- 实现流水线的思想 (4-4、4-5)

## ■ 高级话题 (4-6)

- 性能分析
- 高性能处理器设计

# 本章的内容

## ■ 我们的方法

- 研究对指定指令集的设计（以Y86-64为例）
  - Y86-64：是一个Intel x86-64的简单版本，这是为了教学专门自己设计的一个指令集。
  - 如果你理解一种，那么你基本就了解所有种类
- 研究微体系结构层
  - 将各个硬件块聚合成一个处理器结构
    - 比如：内存，功能单元等
  - 通过控制逻辑来确保每条指令正确的运行

# 日程安排（分为三个阶段）

## ■ 第一阶段

- 指令集体系结构（4-1）
- 逻辑设计（4-2）

## ■ 第二阶段

- 串行实现（4-3）
- 流水线和初级流水线实现（4-4）

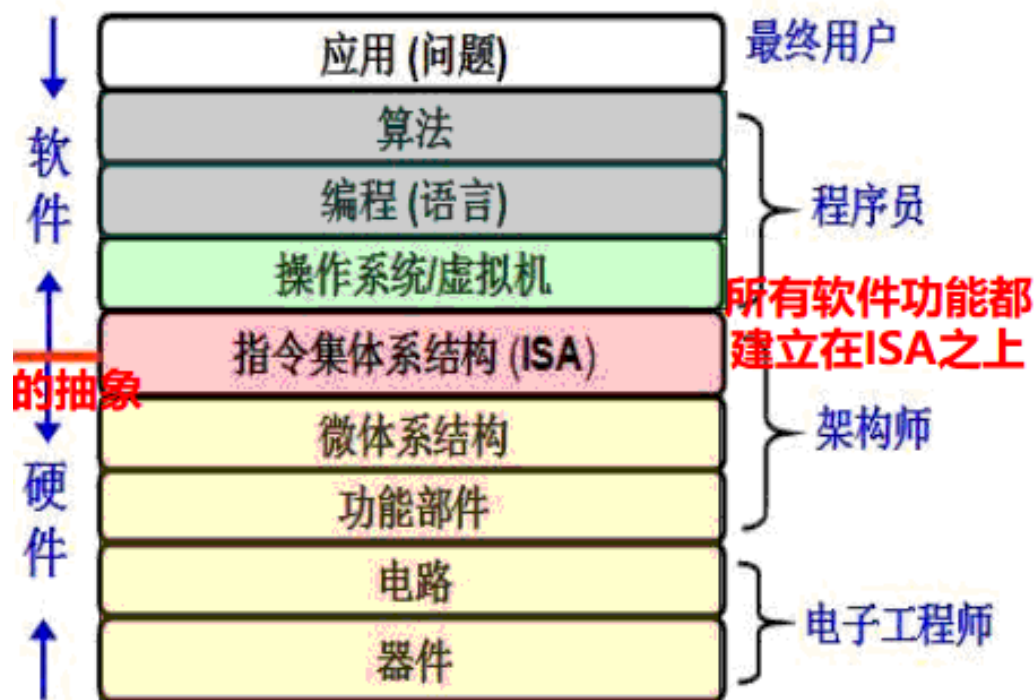
## ■ 第三阶段

- 使流水线运行（4-4、4-5）
- 现代处理器设计（4-6）

# 第四章 处理器体系结构

## 4-1——指令集体系结构/架构

**功能转换：**上层是下层的**抽象**，下层是上层的**实现**  
**底层为上层提供支撑环境！**



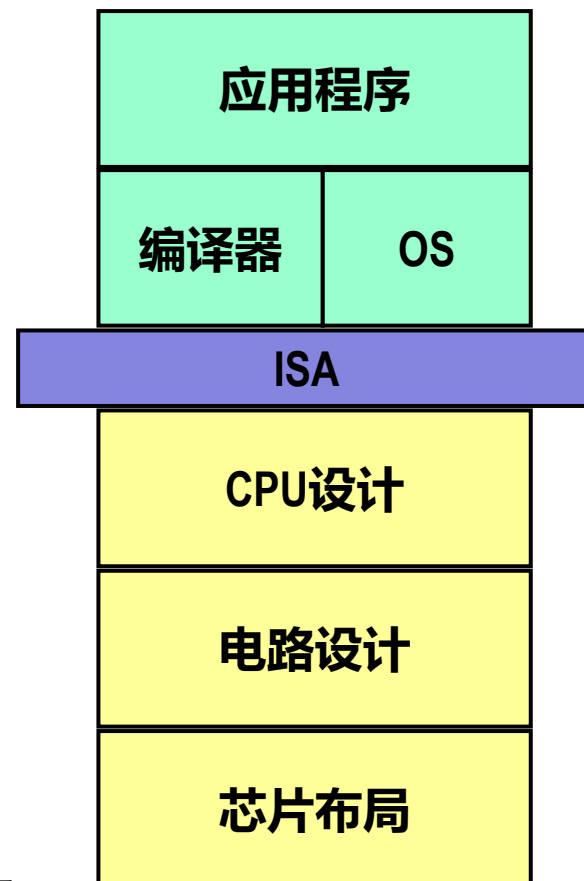
# 指令集架构

## ■ 汇编语言视角

- 处理器状态
  - 寄存器, 内存, ...
- 指令
  - `addq, pushq, ret, ...`
  - 指令是如何编码成字节的

## ■ 抽象层

- 上: 如何对机器进行编程
  - 处理器顺序执行指令
- 下: 我们需要建立什么
  - 用多样的技巧使得机器快速运转
  - 比如 同时处理多条指令



# Y86-64 处理器状态

**RF: 程序寄存器/寄存器文件**  
(Program registers)

%rax	%rsp	%r8	%r12
%rcx	%rbp	%r9	%r13
%rdx	%rsi	%r10	%r14
%rbx	%rdi	%r11	

**CC: 条件码**  
(Condition codes)

**ZF SF OF**

**PC**

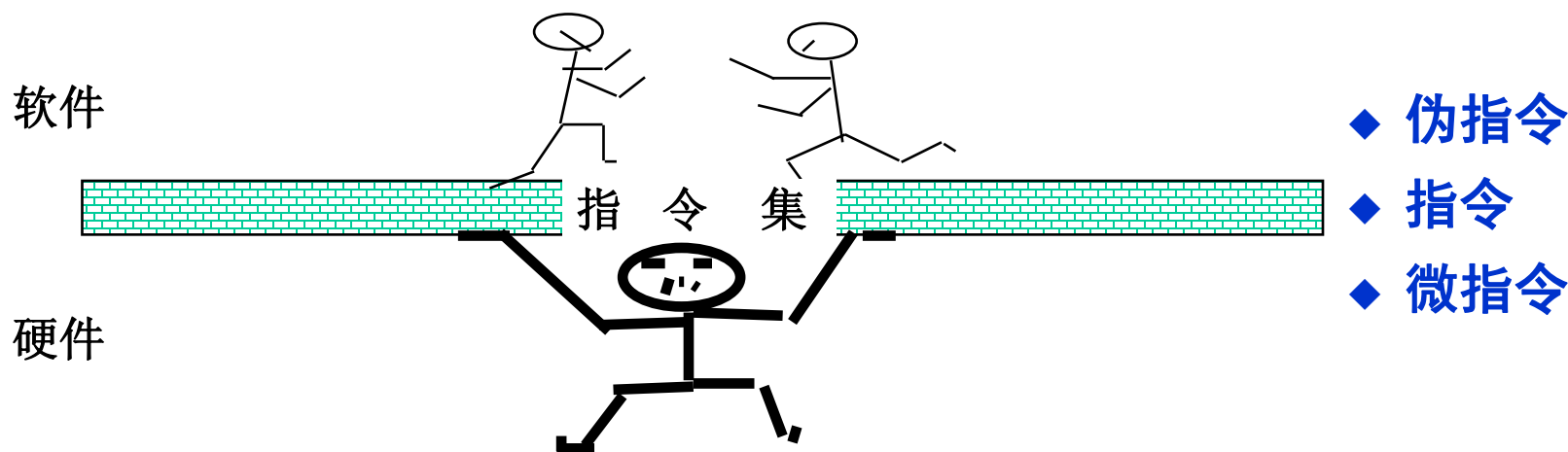
**Stat: 程序状态**  
(Program status)

**DMEM: 内存**  
(Memory)

- **程序寄存器**: 15 个寄存器 (除了 %r15), 每个64位
- **条件码**: 由算术指令或逻辑指令设置的1-bit 标识(flag)
  - ZF: 零(Zero)      SF:负 (Negative)    OF: 溢出(Overflow)
- **程序计数器**: 保存下一条指令的地址
- **程序状态**: 指示是正常操作或一些错误状态
- **内存**: 可字节编址的存储阵列, 小端字节顺序存储的字

# 指令的层次

- 指令系统处在软/硬件交界面，同时被硬件设计者和系统程序员看到
- 硬件设计者角度：指令系统为CPU提供功能需求，要求要易于设计硬件
- 系统程序员角度：通过指令系统来使用硬件，要求易于编写编译器
- 指令系统设计的好坏还决定着：计算机的性能和成本





# CISC 指令集（复杂指令集）

- 复杂指令集计算机
  - IA32是一个例子
- 基于栈的指令集
  - 用栈来传参数，保存程序计数器
  - 明确的入栈、出栈指令
- 算数指令可以访存
  - `addq %rax, 12(%rbx, %rcx, 8)`
    - 需要读写内存
    - 复杂地址计算
- 条件码
  - 设定为算数和逻辑指令的副作用
- 理念
  - 添加指令以便执行“典型”的编程任务

CISC代表：intel的x86（IA32）

# RISC 指令集

RISC代表：天河二号 (MIPS) , ARM系列

- 精简指令集计算机
- IBM的内部项目，后来被 Hennessy (斯坦福大学) 和 Patterson (伯克利大学)推广
- **更少、更简单的指令**
  - 需要用更多的指令来完成任务
  - 可以用小且快速的硬件来执行
- **基于寄存器的指令集**
  - 更多（一般32个)的寄存器
  - 用于存储参数、返回指针和临时存储
- **仅加载 (load) 和存储(store)指令能够访存**
  - 与Y86-64中的 `mrmovq` 和 `rmmovq` 类似
- **没有条件码：测试指令用寄存器返回0或1**

# MIPS 的寄存器（RISC指令集）

\$0	\$0	常数0	\$16	\$s0	被调用者保存暂存单元, 调用过程不可以覆盖写
\$1	\$at	保留 (暂时)	\$17	\$s1	
\$2	\$v0	函数调用返回值	\$18	\$s2	
\$3	\$v1		\$19	\$s3	
\$4	\$a0	过程参数	\$20	\$s4	
\$5	\$a1		\$21	\$s5	调用者保存的暂存单元
\$6	\$a2		\$22	\$s6	
\$7	\$a3		\$23	\$s7	
\$8	\$t0	调用者保存暂存单元, 被调用函数可覆盖写	\$24	\$t8	
\$9	\$t1		\$25	\$t9	操作系统的保留区域
\$10	\$t2		\$26	\$k0	
\$11	\$t3		\$27	\$k1	全局指针
\$12	\$t4		\$28	\$gp	
\$13	\$t5		\$29	\$sp	栈指针
\$14	\$t6		\$30	\$s8	子程序的临时寄存器
\$15	\$t7		\$31	\$ra	返回地址

# CISC vs. RISC

## ■ 起初的辩论

- 强硬的观点
- CISC 支持者---容易编译, 代码的字节数更少
- RISC 支持者---能更好地优化编译器, 用简单的芯片设计能快速运行

## ■ 目前的状态

- 对于台式机处理器来说, 指令集架构的选择不是技术问题
  - 拥有足够的硬件, 可以让任何程序快速运行
  - 代码的兼容性更重要
- x86-64吸纳了很多RISC特性 (内核 RISC, 外围 CISC)
  - 更多的寄存器; 用寄存器传递参数
- 对于嵌入式处理器, RISC 更适合
  - 更小、更便宜、功耗更低
  - 大部分手机用ARM处理器

# ISA指令系统的设计原则

- **RISC还是CISC?**
  - 还是RISC+CISC ?
- **完备性：ISA的指令足够使用**
  - 功能齐全：不能没有加法指令，但可以没有inc指令
- **有效性：程序能够高效率运行**
  - 生成代码小：频率高的操作码和操作数 要短
- **规整性：对称性、匀齐性、一致性**
  - 比如：push rsp / pop rsp 应保证栈顶恢复（要成对出现）
- **兼容性：相同的基本结构、共同的基本指令集**
- **灵活性：**
  - 如操作数的寻址方式：满足基本的数据类型访问
- **可扩充性：操作码字段预留一定的空间**

## ISA指令格式的选择应遵循的几条基本原则

- 尽量短
- 有足够的操作码位数（指令 = 操作码+操作数）
- 指令编码必须有唯一的解释，否则是不合法的指令
- 指令字长是字节的整数倍
- 合理地选择地址字段的个数
- 指令尽量规整

### 与指令集设计相关的重要方面

- 操作码的全部组成：操作码个数 / 种类 / 复杂度

**LD(load)/ST(store)/INC(+1指令)/BRN(无条件跳转)** 四种指令已足够编制任何可计算程序，但程序会很长

- 数据类型：对哪几种数据类型完成操作
- 指令格式：指令长度 / 地址码个数 / 各字段长度
- 通用寄存器：个数 / 功能 / 长度
- 寻址方式：操作数地址的指定方式
- 下条指令的地址如何确定：顺序，PC+1；条件转移；无条件转移： .....

一般通过对操作码进行不同的编码来定义不同的含义，操作码-指令码相同时，再由功能码定义不同的含义！

# 一条指令包含的信息

- 操作码：指定操作类型—指令类型  
(操作码长度：固定 / 可变)  
数据类型bwlq, (短/近/远)跳转, 方向, 寻址方式, 条件码等 (bwlq:Byte, Word, Long Word, Quarter Word, 字节, 字, 双字, 四字。)
- 源操作数参照：一个或多个源操作数所在的地址  
(操作数来源：内存/寄存器/I/O端口/指令本身)
- 结果值参照：产生的结果存放何处（目的操作数）  
(结果地址：内存/寄存器/I/O端口)
- 下一条指令地址：下条指令存放何处  
(下条指令地址：内存)  
(正常情况隐含在PC中, 改变顺序时由指令给出)  
指令的地址可以多个：0、1、2、3、……

# 再次提醒：

根据前面的原则，我们设计出Y86-64。它是一个Intel x86-64的简单版本，这是为了方便教学专门设计的一个指令集。

Y86-64是一个RISC+CISC的指令集。

本章接下来的内容都将以Y86-64指令集为例研究。--如果你理解这一种，那么你基本就了解所有种类的指令集。



# Y86-64 指令集 1

字节	0	1	2	3	4	5	6	7	8	9
halt	0	0	注意这里的数字都是16进制，也就是一个数字占4bit，两个数字是1B。							
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

F指的是R15的编码1111

或者就是16进制的F

F指的是R15的编码1111

或者就是16进制的F

# Y86-64 指令集

## ■ 格式

- 1-10字节的信息，从内存读取
  - 根据第一字节可判断指令长度
  - 比x86-64的指令类型少
  - 比x86-64的编码简单
- 每次存取更改程序状态的一些部分

# Y86-64 指令集 2

字节	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB			V			
rmmovq rA, D(rB)	4	0	rA	rB						D
mrmmovq D(rB), rA	5	0	rA	rB						D
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn								Dest
call Dest	8	0								Dest
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

rrmovq 2 0

cmovle 2 1

cmovl 2 2

cmove 2 3

cmovne 2 4

cmovge 2 5

cmovg 2 6

# Y86-64 指令集 3

字节	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB			V			
rmmovq rA, D(rB)	4	0	rA	rB			D			
mrmmovq D(rB), rA	5	0	rA	rB			D			
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn								
call Dest	8	0								
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

addq 6 0

subq 6 1

andq 6 2

xorq 6 3

# Y86-64 指令集 4

字节	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB						
rmmovq rA, D(rB)	4	0	rA	rB						
mrmmovq D(rB), rA	5	0	rA	rB						
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn								
call Dest	8	0								
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

jmp	7	0
jle	7	1
jl	7	2
je	7	3
jne	7	4
jge	7	5
jg	7	6

# 寄存器编码

- 每个寄存器都有1个4-bit的 ID

<b>%rax</b>	<b>0</b>
<b>%rcx</b>	<b>1</b>
<b>%rdx</b>	<b>2</b>
<b>%rbx</b>	<b>3</b>
<b>%rsp</b>	<b>4</b>
<b>%rbp</b>	<b>5</b>
<b>%rsi</b>	<b>6</b>
<b>%rdi</b>	<b>7</b>

<b>%r8</b>	<b>8</b>
<b>%r9</b>	<b>9</b>
<b>%r10</b>	<b>A</b>
<b>%r11</b>	<b>B</b>
<b>%r12</b>	<b>C</b>
<b>%r13</b>	<b>D</b>
<b>%r14</b>	<b>E</b>
<b>No Register</b>	<b>F</b>

- 和在 x86-64 编码一样
- 寄存器 ID 15 (0xF) 意味着“无寄存器”
  - 会在硬件设计中的许多地方用到

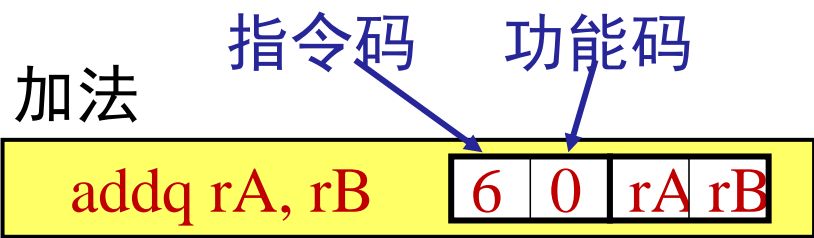
## 指令示例

## ■ 加法指令



- 将rA中的值加到rB中
  - 结果存储到rB中
  - 注意Y86-64仅允许在寄存器数据中应用加法
- 根据结果设置条件码
- e.g., `addq %rax,%rsi`      编码: 60    0    6  
   addq   %rax %rsi
- 两字节编码
  - 第一字节指出指令类型
  - 第二字节指出源和目的寄存器

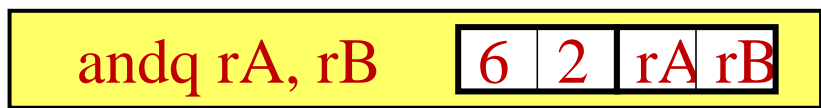
# 算数和逻辑操作



减法(rB-rA)



与操作



异或



- 通常以“OPq”表述
- 编码区别仅在于“功能码”
  - 指令第一个字的低4位
- 设置条件码作为指令执行的副作用



# 传送操作

寄存器 → 寄存器

**rrmovq rA, rB**

2	0
---	---

立即数 → 寄存器

**irmovq V, rB**

3	0	F	rB
---	---	---	----

V

寄存器 → 内存

**rrmmovq rA, D(rB)**

4	0	rA	rB
---	---	----	----

D

内存 → 寄存器

**rmmovq D(rB), rA**

5	0	rA	rB
---	---	----	----

D

- 与x86-64中movq 指令类似
- 内存地址：更简单的格式
- 赋予不同名称，以保持他们的唯一性

# 传送指令示例

## X86-64

**movq \$0xabcd, %rdx**

**编码: 30 82 cd ab 00 00 00 00 00 00**

**movq %rsp, %rbx**

**编码: 20 43**

**movq -12(%rbp),%rcx**

**编码: 50 15 f4 ff ff ff ff ff ff**

**movq %rsi,0x41c(%rsp)**

**编码: 40 64 1c 04 00 00 00 00 00 00**

## Y86-64

**irmovq \$0xabcd, %rdx**

**编码: 30 82 cd ab 00 00 00 00 00 00**

**rrmovq %rsp, %rbx**

**编码: 20 43**

**mrmovq -12(%rbp),%rcx**

**编码: 50 15 f4 ff ff ff ff ff ff**

**rmmovq %rsi,0x41c(%rsp)**

**编码: 40 64 1c 04 00 00 00 00 00 00**

# 条件传送指令

无条件传送

**rrmovq rA, rB**

2	0	rA	rB
---	---	----	----

当小于或等于时传送

**cmovle rA, rB**

2	1	rA	rB
---	---	----	----

当小于时传送

**cmovl rA, rB**

2	2	rA	rB
---	---	----	----

当等于时传送

**cmove rA, rB**

2	3	rA	rB
---	---	----	----

当不等于时传送

**cmovne rA, rB**

2	4	rA	rB
---	---	----	----

当大于或等于时传送

**cmovge rA, rB**

2	5	rA	rB
---	---	----	----

当大于时传送

**cmovg rA, rB**

2	6	rA	rB
---	---	----	----

- 通常表示为“**cmovXX**”
- 指令编码仅在“功能码”上有区别
- 以条件码的值为依据
- **rrmovq** 指令的变体
  - (有条件地)从源寄存器复制到目的寄存器

# 跳转指令

跳转（有条件地）

<b>jXX Dest</b>	<b>7 fn</b>	<b>Dest</b>
-----------------	-------------	-------------

- 通常表示为 “jXX”
- 指令编码仅在“功能码” 上有区别
- 以条件码的值为依据
- 和x86-64中的操作相同
- 对目的地址进行完整编码
  - 和x86-64中的PC相对地址编码不一样

# 跳转指令

无条件跳转

**jmp Dest**   **7** **0** Dest

当小于或等于时跳转

**jle Dest**   **7** **1** Dest

当小于时跳转

**jl Dest**   **7** **2** Dest

当等于时跳转

**je Dest**   **7** **3** Dest

当不等于时跳转

**jne Dest**   **7** **4** Dest

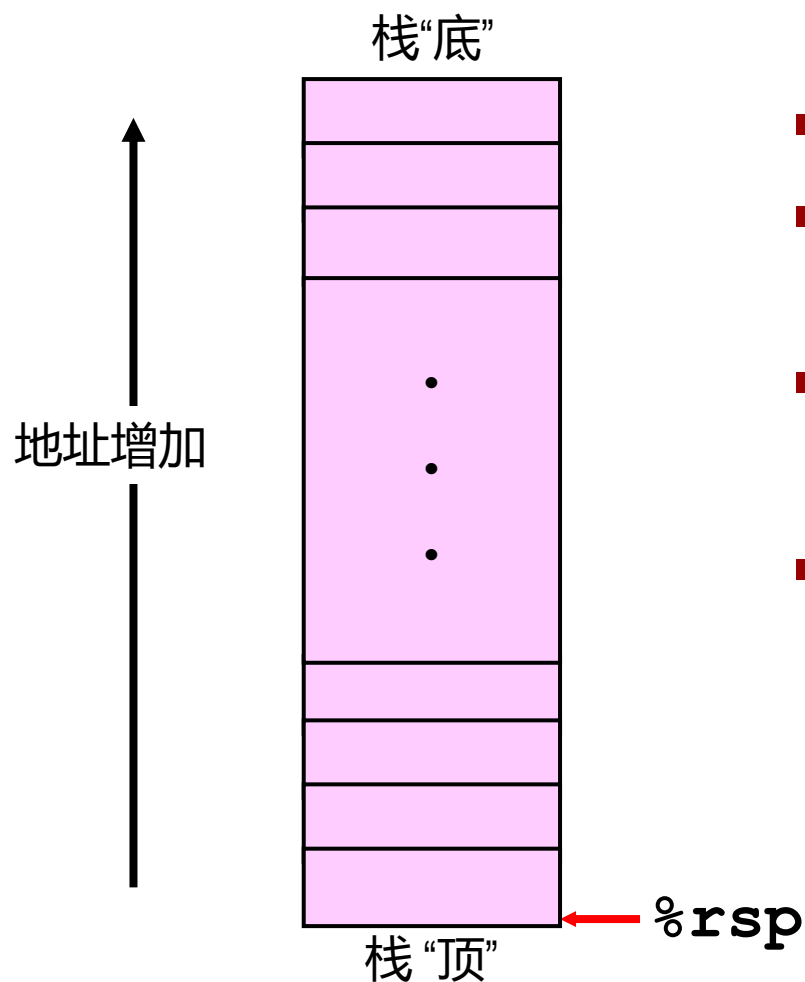
当大于或等于时跳转

**jge Dest**   **7** **5** Dest

当大于时跳转

**jg Dest**   **7** **6** Dest

# Y86-64 程序栈



- 存储程序数据的内存区域
- 在 Y86-64 (和 x86-64) 中用于支持过程调用
- `%rsp` 指向栈顶
  - 栈顶元素的地址
- 栈向低地址的方向增长
  - 入栈操作必须先减小栈指针
  - 出栈操作后，增加栈指针

# 栈操作

**pushq rA**

A	0	rA	F
---	---	----	---

- %rsp 减 8
- 将rA 中保存的字，存储到内存中 %rsp 处
- 与 x86-64 相似

**popq rA**

B	0	rA	F
---	---	----	---

- 从内存中的 %rsp 位置读取字
- 存放到 rA 中
- %rsp 加 8
- 与 x86-64 相似

# 子程序调用和返回

**call Dest**

**8 0 Dest**

- 将下一条指令的地址入栈
- 开始执行位于 Dest 的指令
- 与 x86-64 相似

**ret**

**9 0**

- 从栈中弹出值
- 用作下一条指令的地址
- 与 x86-64 相似



# 其他指令

**nop**

1 0

- 不作任何事

**halt**

0 0

halt: 退出程序。

exit: 退出过程、函数。如果在主程序，则效果和halt一样。

break: 跳出循环。

- 停止执行指令
- x86-64 有类似指令，但是不能在用户模式下执行
- 我们将会用这条指令来终止模拟器
- 编码确保：命中初值为0的内存时，程序会停止

# 状态条件

名字	编码
AOK	1

名字	编码
HLT	2

名字	编码
ADR	3

名字	编码
INS	4

- 正常操作
  - 遇到停止指令halt
  - 遇到错误地址 (指令或者数据)
  - 遇到无效的指令
- 
- 预期行为
    - 如果是AOK，继续执行
    - 否则，停止程序执行

# 编写 Y86-64 代码

## ■ 尽量多用C编译器

- 用 C 编码
- 在 x86-64 中用 `gcc -Og -S` 编译
- 把 X86-64 的 `asm` 直译到 Y86-64 中
- 现代编译器使得这个过程更加复杂

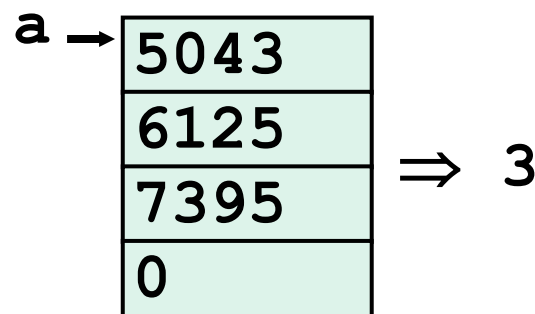
留个奢望：  
做个Y86-64的  
C编译器？

## ■ 编码示例

- 计算以 NULL-0 结尾的列表中元素的个数

```
int len1(int a[]);
```

注意 `int` 数据长度：32-bit (4B)



# Y86-64 代码生成示例

## ■ 首先尝试

- 编写典型的数组代码

```
/* Find number of elements in
   null-terminated list */
long len(long a[])
{
    long len;
    for (len = 0; a[len]; len++)
        ;
    return len;
}
```

- 用 `gcc -Og -S` 编译

## ■ 问题

- 在 Y86-64 上难以做数组的索引
  - 因为没有带比例因子的寻址模式

L3:

```
addq $1,%rax
cmpq $0, (%rdi,%rax,8)
jne L3
```

# Y86-64 代码生成示例 2

## ■ 然后尝试

- 用 C 语言模仿 Y86-64 的代码

```
long len2(long *a){  
    long ip = (long) a;  
    long val = *(long *) ip;  
    long len = 0;  
    while (val) {  
        ip += sizeof(long);  
        len++;  
        val = *(long *) ip;  
    }  
    return len;  
}
```

## ■ 结果

- 编译器生成和之前一模一样的代码
- 编译器将两个版本转换成相同的中间形式

# Y86-64 代码生成示例 3

```

len:
    irmovq $1, %r8          # Constant 1
    irmovq $8, %r9          # Constant 8
    irmovq $0, %rax          # len = 0
    mrmovq (%rdi), %rdx      # val = *a
    andq %rdx, %rdx          # Test val
    je Done                  # If zero,
goto Done
Loop:
    addq %r8, %rax           # len++
    addq %r9, %rdi           # a++
    mrmovq (%rdi), %rdx      # val = *a
    andq %rdx, %rdx          # Test val
    jne Loop                 # If !0, goto
Loop
Done:
    ret

```

寄存器	用途
%rdi	a
%rax	len
%rdx	val
%r8	1
%r9	8

# Y86-64 程序结构 1

```

init:                # Initialization
    ...
    call Main
    halt
    .align 8         # Program data
array:
    ...
Main:                # Main function
    ...
    call len
    ...
len:                 # Length function
    ...
    .pos 0x100       # Placement of stack
Stack:
  
```

- 程序从地址0处开始
- 一定要建立栈
  - 位于哪里
  - 指针的值
  - 确保不要覆盖代码区域
- 一定要初始化数据

# Y86-64 程序结构 2

**init:**

```
# Set up stack pointer
irmovq Stack, %rsp
# Execute main program
call Main
# Terminate
halt
# Array of 4 elements + terminating 0
.align 8
```

**Array:**

```
.quad 0x000d000d000d000d
.quad 0x00c000c000c000c0
.quad 0x0b000b000b000b00
.quad 0xa000a000a000a000
.quad 0
```

- 程序从地址0处开始
- 一定要建立栈
  - 位于哪里
  - 指针的值
  - 确保不要覆盖代码区域
- 一定要初始化数据
- 可以用符号化名字



# Y86-64 程序结构 3

**Main:**

```
irmovq array,%rdi  
# call len(array)  
call len  
ret
```

- **建立对 len 的调用**
  - 遵循 x86-64 的过程约定
  - 将数组地址做为实参

# 汇编 Y86-64 程序

```
unix> yas len.yas
```

- 生成“目标代码”文件 `len.yo`
  - `len.yo`看起来像反汇编程序的输出(机器码与汇编一一对应, x86的生成的机器码不是一一对应的)

```

0x054:                                | len:
0x054: 30f8010000000000000000000000 |   irmovq $1, %r8           # Constant 1
0x05e: 30f9080000000000000000000000 |   irmovq $8, %r9           # Constant 8
0x068: 30f0000000000000000000000000 |   irmovq $0, %rax          # len = 0
0x072: 5027000000000000000000000000 |   mrmovq (%rdi), %rdx       # val = *a
0x07c: 6222                               |   andq %rdx, %rdx          # Test val
0x07e: 73a0000000000000000000000000 |   je Done                  # If zero, goto Done
0x087:                                | Loop:
0x087: 6080                               |   addq %r8, %rax            # len++
0x089: 6097                               |   addq %r9, %rdi            # a++
0x08b: 5027000000000000000000000000 |   mrmovq (%rdi), %rdx       # val = *a
0x095: 6222                               |   andq %rdx, %rdx          # Test val
0x097: 7487000000000000000000000000 |   jne Loop                  # If !0, goto Loop
0x0a0:                                | Done:
0x0a0: 90                               |   ret

```

# 模拟 Y86-64 程序

```
unix> yis len.yo
```

- 指令集模拟器
  - 每条指令在不同处理器状态的计算效果
  - 打印状态的变化

Stopped in 33 steps at PC = 0x13. Status 'HLT', CC Z=1 S=0 O=0

Changes to registers:

%rax:	0x0000000000000000	0x0000000000000004
%rsp:	0x0000000000000000	0x0000000000000100
%rdi:	0x0000000000000000	0x0000000000000038
%r8:	0x0000000000000000	0x0000000000000001
%r9:	0x0000000000000000	0x0000000000000008

Changes to memory:

0x00f0:	0x0000000000000000	0x0000000000000053
0x00f8:	0x0000000000000000	0x0000000000000013

1.在Y86-64 CPU中有15个从0开始编码的通用寄存器，在对指令进行编码时，对于仅使用一个寄存器的指令，简单有效的处理方法是（ ）

A.用特定的指令类型代码

B.用特定的指令功能码

☒ C.用特定编码0xF表示操作数不是寄存器

D.无法实现

2.Y86-64 的指令编码长度是（ ）个字节

☒ A.1~10

B.32

C.64

D.128

# 总结

## ■ Y86-64 指令集架构

- 和x86-64类似的状态、指令
- 更简单的编码
- 介于CISC 和 RISC 之间

## ■ ISA设计有多重要？

- 没有以前更重要
  - 拥有足够的硬件资源，几乎可以让任何程序都能快速运行