

# 高性能计算导论 | 计算机组装原理与程序运行模型

(Introduction to HPC - Training Camp | Computer Composition and Program Model)

李岳昆

Public Safety Institution of Big Data  
Taiyuan University of Technology

2022 年 1 月 20 日



太原理工大学  
TAIYUAN UNIVERSITY OF TECHNOLOGY



PUBLIC SAFETY  
INSTITUTION OF BIG DATA

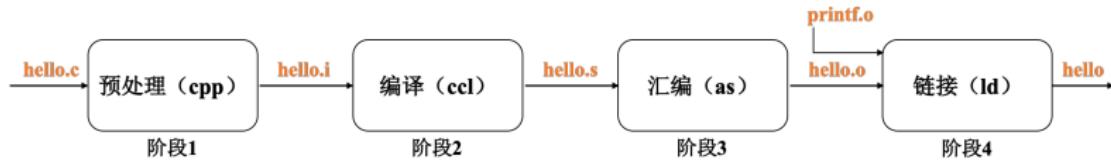
- Hello World 程序执行过程中发生了什么?
- 计算机硬件架构
- 从指令角度观察程序运行
- CPU: 计算机中的加工厂
- 内存: 原料仓库
- 高速缓存策略与映射
- 初识汇编语言

# Hello World 程序执行过程中发生了什么？

## Hello World

```
#include<stdio.h>
int main(){
    printf("hello,world");
    return 0;
}
```

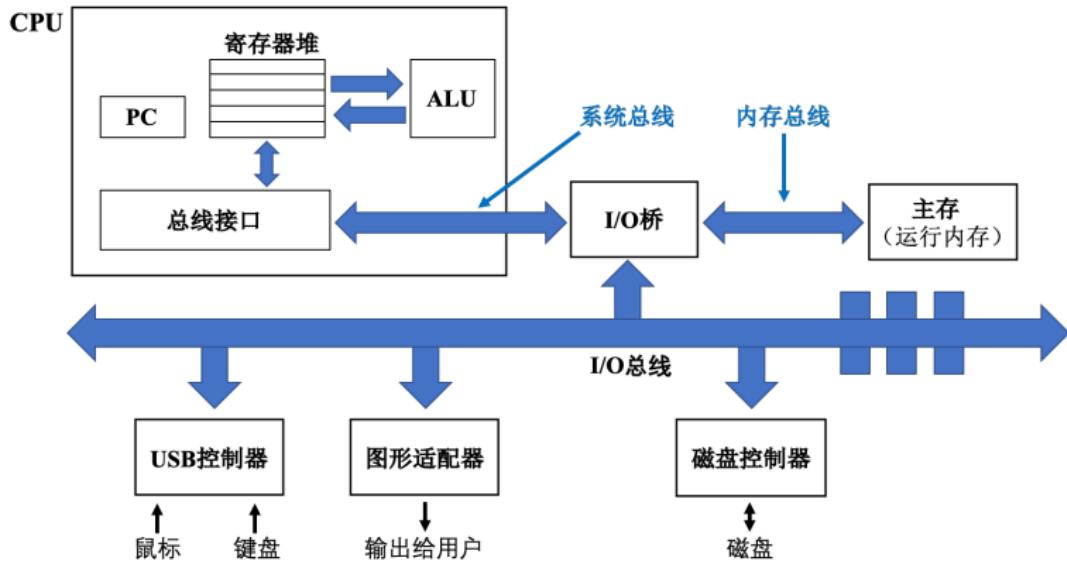
- 在 Linux 中，生成可执行程序 hello：
  - `gcc -o hello hello.c`
- 过程虽然是通过一条命令完成的，然而实际上编译系统的处理过程却是非常复杂的，大致可以分为四个阶段，分别为预处理、编译、汇编以及链接。



- 预处理：预处理器以 # 开头的代码修改原始程序。例如 hello 程序中引入了头文件 stdio.h，预处理器会读取该头文件中的内容，将其中的内容直接插入到源程序中，得到文本文件 hello.i。
- 编译：编译器将 hello.i 文件翻译成 hello.s 文件，这一阶段包括词法分析、语法分析、语义分析、中间代码生成以及优化等等一系列的中间操作。

- 汇编：汇编器根据指令集将汇编程序 hello.s 翻译成机器指令，并按规则打包，得到可重定位二进制目标文件 hello.o。
- 链接：链接器 (ld) 负责把 hello.o 和 printf.o 按照一定规则进行合并，得到可执行目标文件 hello。

# 计算机硬件架构



- hello.c 经过编译系统得到可执行目标文件 hello，此时可执行目标文件 hello 已经存放在系统的磁盘上。
- 在 linux 系统上运行可执行程序：打开一个 shell 程序，然后在 shell 中输入相应可执行程序的文件名：
- *linux>./hello*

shell 是一个命令解释程序，如果命令行的第一个单词不是内置的 shell 命令，shell 就会对这个文件进行加载并运行。此处，shell 加载并且运行 hello 程序，屏幕上显示 hello,world 内容，hello 程序运行结束并退出，shell 继续等待下一个命令的输入。

## 执行过程

- ① 首先我们通过键盘输入"./hello" 的字符串，shell 程序会将输入的字符逐一读入寄存器，处理器会把 hello 这个字符串放入内存中。
- ② 当我们完成输入，按下回车键时，shell 程序就知道我们已经完成了命令的输入，然后执行一系列的指令来加载可执行文件 hello。
- ③ 这些指令将 hello 中的数据和代码从磁盘复制到内存。数据就是我们要显示输出的"hello , world\n"，这个复制的过程将利用 DMA(Direct Memory Access) 技术，数据可以不经过处理器，从磁盘直接到达内存。
- ④ 当可执行文件 hello 中的代码和数据被加载到内存中，处理器就开始执行 main 函数中的代码，main 函数非常简单，只有一个打印功能。

# 了解编译系统如何工作是大有益处的

## ① 理解编译系统可以优化程序的性能。

- 现代编译器是非常成熟的工具，通常可以生成很好的代码，作为一个程序员，我们没有必要为了写出高效的代码，而去研究编译器的内部是如何工作的，但是，我们还是需要对机器执行的代码有一个基本的了解，这样我们就知道编译器把不同的 C 代码转换成的机器代码是什么。
- 我们在写代码的时候可能会有这样的困惑，或者面试中会被问到以下类似的问题：
- 例如：一个 switch 语句是不是要比一连串的 if-else 要高效的多？一个函数调用的开销有多大？while 循环比 for 循环更高效么？

## ② 理解编译系统可以帮助我们理解链接过程中出现的错误。

- 如果所有的程序都像 helloworld 一样简单，那的确没有必要去理解编译系统，但是当你试图去构建大型程序的时候，往往涉及到各种函数库的调用，根据以往的经验，一些奇奇怪怪的错误往往都是与链接器有关的。
- 例如：静态变量和全局变量的区别是什么？静态库和动态库的区别是什么？
- 更严重的是，还有一些链接错误直到程序运行的时候才会出现。

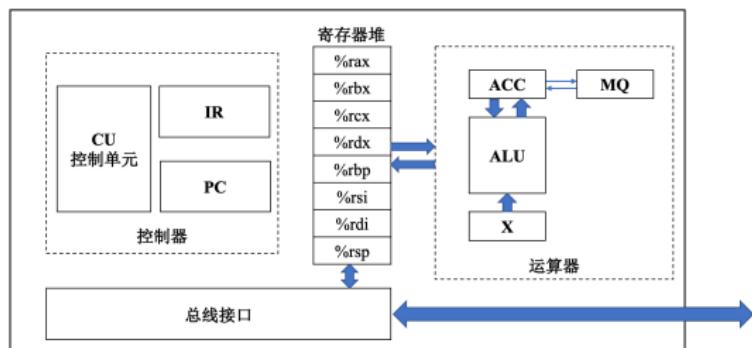
## ③ 避免安全漏洞。

- 多年以来，缓冲区溢出 (buffer overflow) 是导致互联网安全漏洞的主要原因，如何避免写出的代码存在安全漏洞，第一步就是要理解数据和控制信息在程序栈上是如何存储的，了解不严谨不规范的书写方式会引起什么样的后果。

## CPU

中央处理单元 (Central Processing Unit , CPU), 也称处理器, 包含 PC ( 程序计数器: Program Count )、寄存器堆 (Register file)、ALU(算数/逻辑计算单元: Arithmetic/logic Unit) 三个部分。

## CPU



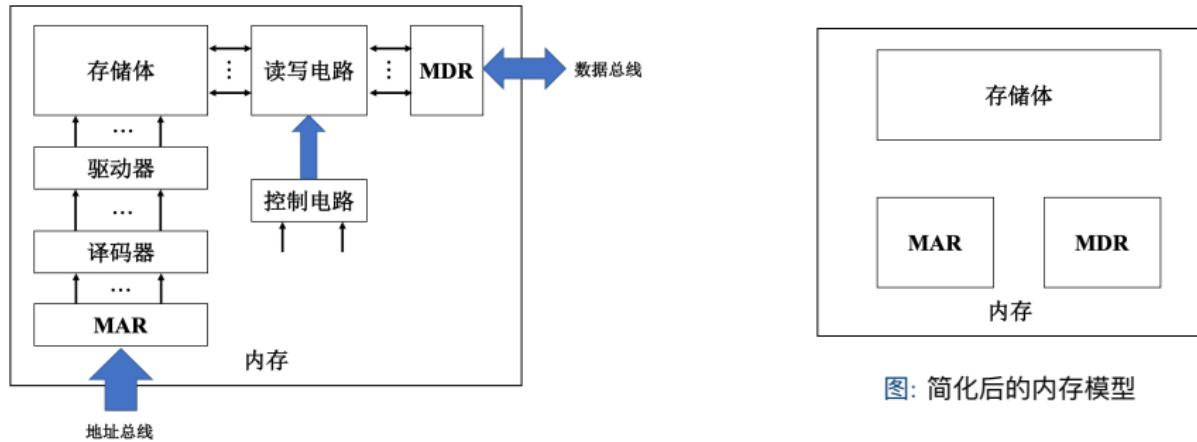
就 CPU 而言, 其控制器内部构成较为复杂, 包含:

- ① PC: 程序计数器, 负责取指令;
- ② IR: 指令寄存器, 存放当前执行的指令, 负责分析指令;
- ③ CU: 控制单元, 负责分析指令, 给出控制信号, 负责执行指令。

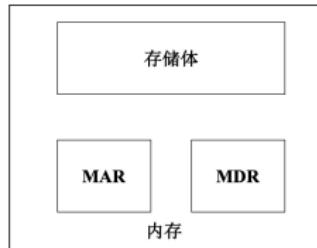
- 运算器中也不仅仅为 ALU(算数/逻辑运算单元), 更有: ACC: 累加器, 用于存放操作数, 或运算结果。MQ: 乘商寄存器, 在乘、除运算时, 用于存放操作数或运算结果。X: 通用的操作数寄存器, 用于存放操作数。

## 内存

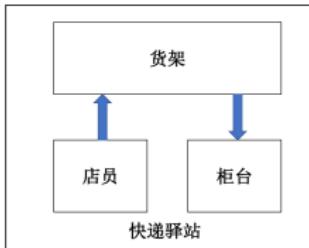
**主存** (Main Memory), 也称为内存、运行内存, 处理器在执行程序时, 内存主要存放程序指令以及数据。从物理上讲, 内存是由随机动态存储器芯片组成; 从逻辑上讲, 内存可以看成一个从零开始的大数组, 每个字节都有相应地址。



- ① MAR 全称为存储地址寄存器 (Memory Address Register)
- ② MDR 全称为存储数据寄存器 (Memory Data Register)



图：内存模型



图：快递驿站



图：存储体架构

- 我们可以将存储体看成是有着  $n$  层高的货架，每一层都可以存储若干个包裹，即每个存储单元都由若干个二进制码组成。每一层存储的包裹为一个存储字 (word)，而每一层能够存储的包裹数量为存储字长。

## 一个不太恰当的比喻

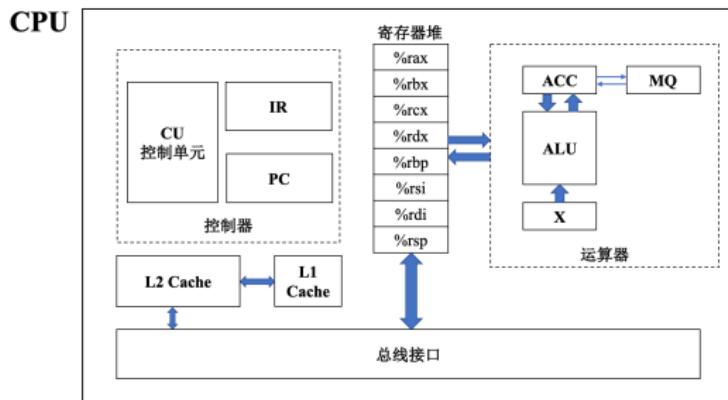
主存的架构有点类似于取快递的过程：当收到取件码后，我们应该到快递驿站向店员出示取件号，店员在货架的对应位置上取走我们的包裹并放置柜台上，我们在柜台取走对应的包裹。

## Cache 至关重要

对于处理器而言，从磁盘上读取一个字所花费的时间开销比从内存中读取的开销大 1000 万倍。寄存器文件的只能存储几百个字节的信息，而内存的可以存放几十亿的字节信息 (GB 级)，从寄存器文件读取数据比从内存读取差不多要快 100 倍。随着半导体技术的发展，处理器与内存之间的差距还在持续增大，针对处理器和内存之间的差异，系统设计人员在寄存器文件和内存之间引入了高速缓存 (Cache)。

一般有三级高速缓存，分别为 L1 cache, L2 cache 以及 L3 cache。

- ① L1 cache 的访问速度与访问寄存器文件几乎一样快，容量大小为数万字节 (KB 级别)；
  - ② L2 cache 的访问速度是 L1 cache 的五分之一，容量大小为数十万到数百万字节之间；
  - ③ L3 cache 的容量更大，同样访问速度与 L2 cache 相比也更慢。



## 总线与输入输出设备

- 内存和处理器之间通过**总线**来进行数据传递。实际上，总线贯穿了整个计算机系统，它负责将信息从一个部件传递到另外一个部件。通常总线被设计成传送固定长度的字节块，也就是字(word)，至于这个字到底是多少个字节，各个系统中是不一样的，32位的机器，一个字长是4个字节；而64位的机器，一个字长是8个字节。
- 除了处理器，内存以及总线，计算机系统还包含了各种输入输出设备，例如键盘、鼠标、显示器以及磁盘等等。每一个输入输出设备都通过一个**控制器**或者**适配器**与IO总线相连<sup>a</sup>。

<sup>a</sup>控制器与适配器主要区别是在于它们的封装方式，无论是控制器还是适配器，它们的功能都是在IO设备与IO总线之间传递数据。

- 总线结构看似对计算机体系架构无关紧要，但实际上却是决定芯片体积大小的重要因素之一。在小小的芯片上布置超大规模总线结构(电子晶体管)，是超大规模集成电路(VLSI)主要探讨的问题，也是现代工业界着力解决的问题之一。

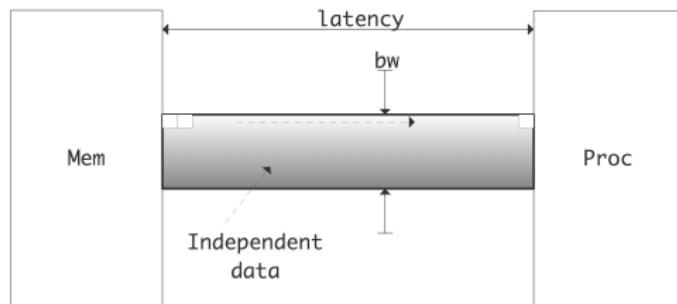
## Little 定理

为了充分利用带宽，在任何时候都必须有相当于带宽乘以延迟的数据量在运行。由于这些数据必须是独立的，我们得到了

$$\text{并发度} = \text{带宽} \times \text{延迟}$$

维护并发性的问题并不是程序没有这种并发性，而是程序要让编译器和运行时系统识别它。例如，如果一个循环遍历了一个长的数组，编译器就不会发出大量的内存请求。预取机制会提前发出一些内存请求，但通常不够。因此，为了使用可用的带宽，多个数据流需要同时进行。因此，我们也可以将 Little 定理表述为

$$\text{有效吞吐量} = \text{并发性}/\text{延迟}$$



## 延迟与带宽

- 延迟：从发送内存请求到实际完成数据移动的时间
- 带宽：单位时间数据移动量

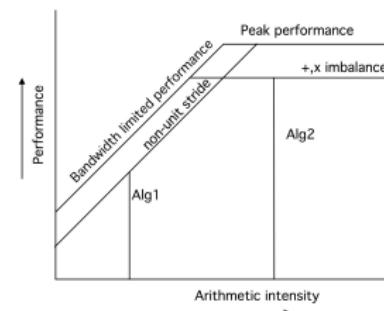
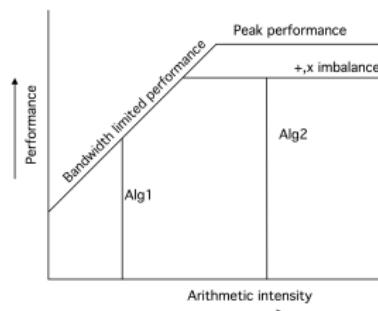
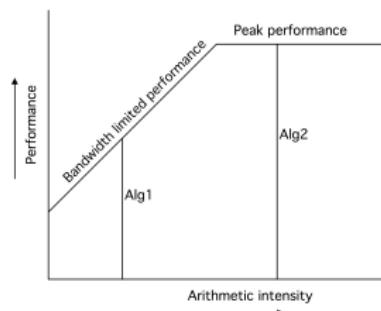
图: Little 定律规定了运行中需要多少独立的数据

计算密度与约束

如果  $n$  是一个算法所操作的数据项的数量, 而  $f(n)$  是它所需要的操作的数量, 则计算密度为

$$f(n)/n$$

对于一个给定的计算密度，其性能是由其垂直线与 Roofline 相交的位置决定的。如果这是在水平部分，那么该计算被称为受计算约束 (compute-bound)：性能由处理器的特性决定，而带宽不是问题。另一方面，如果这条垂直线与屋顶的倾斜部分相交，那么计算被称为受带宽约束 (bandwidth-bound)：性能由内存子系统决定，处理器的全部能力没有被使用。



## 缓存映射——直接映射

缓存映射基本策略有三种：直接映射、全关联映射与 k 路关联映射。最简单的缓存映射策略是直接映射 (direct mapping)。假设内存地址是 32 位长，可以寻址 4G 字节；进一步假设缓存有 8K 个字，也就是 64K 字节，需要 16 位来寻址。直接映射从每个内存地址取最后（“最低有效”）16 位，并使用这些作为缓存中的数据项的地址。

- 直接映射的效率非常高，因为它的地址计算速度非常快，导致了较低的延迟，但在实际应用中存在一个问题。如果两个被 8K 字分隔的条目被寻址，它们将被映射到相同的缓存位置，这将使某些计算效率低下。例如：

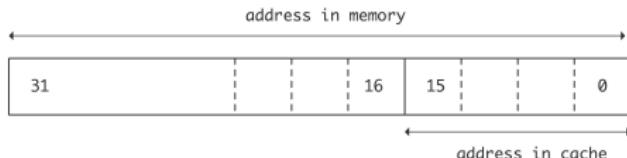
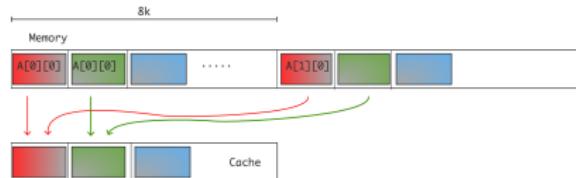


图: 32 位地址直接映射到 64K 高速缓存中

## 缓存映射

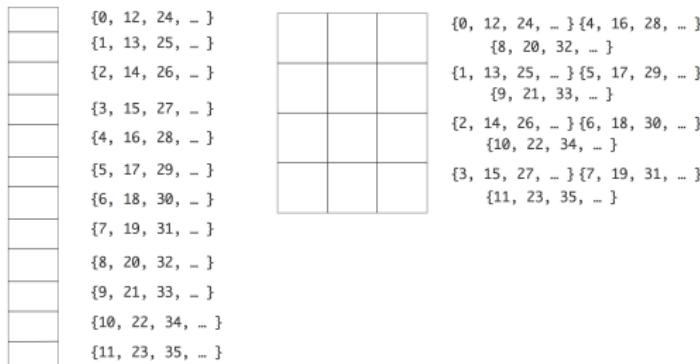
```
double A[3][8192];
for ( i=0; i<512; i++)
    a[2][i] = (a[0][i]+a[1][i])/2;
```



图：直接映射缓存中的映射冲突

# 缓存映射——全关联映射与 k 路关联映射

- 每一个缓存项都与内存地址中的任意位置相关联，这样的缓存被称为**全关联映射** (fully associative mapping)，虽然这是最好的办法，但它的构建成本很高，而且在使用中比直接映射的缓存慢得多。
- 正因如此，最常见的解决方案是建立一个 **k-路关联映射** ( $k$ -way associative mapping)，将缓存分为若干组，每组进行全关联映射。
- 一个直接映射的和一个三向关联的缓冲区的内存地址与缓冲区位置的映射情况。两个缓冲区都有 12 个元素，但是它们的使用方式不同。直接映射的高速缓存 (左边) 在内存地址 0 和 12 之间会有冲突，但是在三向关联高速缓存中，这两个地址可以被映射到三个元素中的任何一个。



## Cache 替换策略

Cache 的替换策略有四种：

- ① 随机替换算法 (RAND);
- ② 先进先出算法 (FIFO);
- ③ 最近最少使用 (LRU);
- ④ 最近不经常使用 (LFU)。

图：两个 12 个元素的缓存：直接映射 (左) 和 3 路关联 (右)



## 缓存一致性

有两种实现缓存一致性的基本机制：监听 (Snooping) 和标签目录 (Tag Directory)。

- ① 在监听机制中，任何对数据的请求都会被发送到所有的缓冲区，如果数据存，就会被返回；否则就会从内存中检索。
- ② 标签目录即一个中央目录，它包含了一些缓存中存在的数据的信息，以及具体在哪个缓存中。对于拥有大量内核的处理器 (如英特尔 Xeon Phi)，该目录可以分布在各个内核上。

```
local_results = new double[num_threads];
#pragma omp parallel{
    int thread_num = omp_get_thread_num();
    for (int i=my_lo; i<my_hi; i++)
        local_results[thread_num] = ... f(i) ...
}
global_result = g(local_results)
```

- 缓存线就会在核心之间不断移动。这就是所谓的**伪共享** (false sharing)。
- 例如左侧代码中，所有线程都在更新自己在部分结果数组中的位置。虽然没有实际的竞争条件，但这段代码的性能会很低，因为带有 *local\_result* 数组的缓存线会不断被废止。

## 性能不佳的代码

```
for (loop=0; loop<10; loop++) {  
    for (i=0; i<N; i++) {  
        ... = ... x[i] ...  
    }  
}
```

## 改进后的代码

```
for (i=0; i<N; i++) {  
    for (loop=0; loop<10; loop++) {  
        ... = ... x[i] ...  
    }  
}
```

- 由于从缓存中读取数据的时间开销要小于从内存中读取，我们希望以这种方式进行编码，进而使缓存中的数据最大程度上得到复用。
- 时间局部性是最容易解释的：即数据使用一次后短时间内再次被使用。由于大多数缓存使用 LRU 替换策略，如果在两次引用之间被引用的数据少于缓存的大小，那么该元素仍然会存在缓存之中，进而实现快速访问。
- $x$  的每个元素将被使用 10 次，但是如果向量（加上其他被访问的数据）超过了缓存的大小，每个元素将在下一次使用前被刷新。因此， $x[i]$  的使用并没有表现出时间局部性：再次使用时的时间间隔太远，使得数据无法停留在缓存中。
- 改进后的代码中  $x$  的元素被反复使用，因此更有可能留在缓存中。

## 空间局部性

空间局部性的概念要稍微复杂一些。如果一个程序引用的内存与它已经引用过的内存“接近”，那么这个程序就被认为具有空间局部性。经典的冯·诺依曼架构中只有一个处理器和内存，此时空间局部性并不突出，因为内存中的一个地址可以像其他地址一样被快速检索。然而，在一个有缓存的现代CPU中，情况就不同了。

- 由于数据是以缓存线而不是单独的字或字节为单位移动的，因此使缓存线所有的元素都得到应用是有所裨益的，在下列循环中

```
for ( i=0; i<N*s; i+=s){  
    ... x[i] ...  
}
```

空间局部性体现为函数所进行的跨步递减  $s$ 。

- 设  $S$  为缓存线的大小，那么当  $s$  的范围从  $1 \dots S$ ，每个缓存线使用的元素数就会从  $S$  下降到 1。相对来说，这增加了循环中的内存流量花销：如果  $s = 1$ ，我们为每个元素加载  $1/S$  的缓存线；如果  $s = S$ ，我们为每个元素加载一个缓存线。

## TLB 也有空间局部性

如果一个程序引用的元素距离很近，它们很可能在同一个内存页上，通过 TLB 的地址转换会很迅速；反之则会造成较大时间开销。

## 单层 AMAT(Average Memory Access Time) 模型

假设只有一级缓存，AMAT 模型预测的平均访存时间为：

$$\begin{aligned} \text{AMAT} &= (1 - r)T_{\$} + r(T_{\$} + T_M) \\ &= T_{\$} + rT_M \end{aligned}$$

其中  $T_{\$}$  为缓存访问时间 (Hit time)， $T_M$  为内存访问时间 (Miss penalty)， $r$  为缓存失效率 (Miss rate)。

假如某平台 CPU 主频为 1GHz，即 CPU 时钟周期是 1ns，且

- 缓存访问开销为 2 拍 (cycle)，即  $T_{\$} = 2\text{ns}$ ；
- 缓存失效损失为 300 拍 (cycle)，即  $T_m = 300\text{ns}$ ；
- 缓存命中率为 90%，即  $r = 0.1$ ；则

$$\text{AMAT} = 2\text{ns} + 0.1 \times 300\text{ns} = 32\text{ns}.$$

## 多层 AMAT(Average Memory Access Time) 模型

假设有两级/三级缓存， $T_1, T_2, T_3$  分别为  $L_1, L_2, L_3$  缓存访问时间， $T_M$  为内存访问时间， $r_1, r_2, r_3$  分别为  $L_1, L_2, L_3$  缓存的局部失效率 (Local miss rate)，则两级/三级 AMAT 模型预测的平均访存时间为：

$$\begin{aligned}\text{AMAT}_2 &= T_1 + r_1(T_2 + r_2 T_M) \\ &= T_1 + R_1 T_2 + R_2 T_M,\end{aligned}$$

$$\begin{aligned}\text{AMAT}_3 &= T_1 + r_1[T_2 + r_2(T_3 + r_3 T_M)] \\ &= T_1 + R_1 T_2 + R_2 T_3 + R_3 T_M.\end{aligned}$$

其中  $R_1 = r_1$ ,  $R_2 = r_1 r_2$ ,  $R_3 = r_1 r_2 r_3$  分别为  $L_1, L_2, L_3$  缓存的整体失效率 (Global miss rate)

注：局部失效率指该层次缓存失效的概率，整体失效率表示该层次缓存及其上层所有缓存同时失效的概率。

# 考虑总线后内存中一个现实的问题

- 前面我们以 Hello World 为例探讨了计算机程序的一般过程。下面我们将以一个简单程序探索指令执行的具体情况。
- 将上述内容在 Linux 中编译后装入主存，可以得到如下过程：

内存地址	指令		注释
	操作码	地址码	
0	000001	00000000101	取数 a 至 ACC
1	000100	00000000110	乘 b 得 ab, 存于 ACC 中
2	000011	00000000111	加 c 得 ab+c, 存于 ACC 中
3	000010	00000001000	将 ab+c, 存于内存
4	000110	00000000000	停机
5	0000000000000010		原始数据 a=2
6	0000000000000011		原始数据 b=3
7	0000000000000001		原始数据 c=1
8	0000000000000000		原始数据 y=0

图：对应程序的指令码

## Instructions.c

```
int a=2, b=3, c=1, y=0;
void main(){
    y=a*b+c;
}
```

- (PC)=0, 整个程序将指向第一条指令的存储地址；
- PC 指向 MAR, 令 (MAR)=0, 发现对应位置存储的指令为 000001 0000000101
- MDR 将数据摆出, CPU 中的 IR 取走指令, 此时 (IR)=000001 0000000101
- IR 分析指令, 将前 6 位操作码送到 CU 控制单元处, 执行“取数”指令, 后 10 位送至 MAR 中;
- MAR 对 0000000101 进行翻译, 得到数字 5 并在内存中寻址;
- 内存地址为 5 的位置上存放着 a = 2, 于是 MAR 将 a = 2 的二进制码放到 MDR 上;
- 于是 ACC 将 2 取回, PC 自加 1。



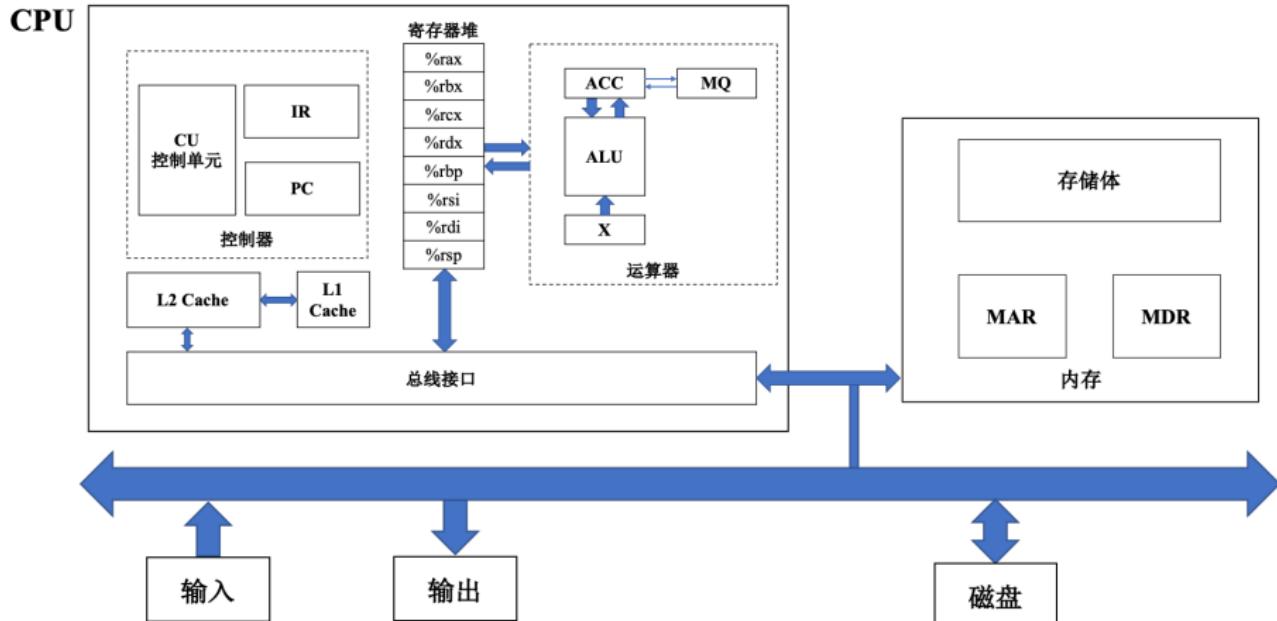
- ① PC 指向 MAR，向其索取下一条指令，此时 MAR=1，MDR 上放置了 000100 0000000110
- ② IR 取回指令，将前 6 位送至 CU 控制单元处，并将后 10 位送至 MAR 中寻址；
- ③ 0000000110 对应存储体上内存地址为 6 的指令，含义为  $b = 3$ ；
- ④ 于是，X 保存 ACC 中的 2 元素， $(MQ)=3$ ，执行  $(MQ) * (X) \rightarrow ACC$
- ⑤ 由 ALU 实现乘法运算，此时  $(ACC)=6$ ；值得注意的是，如果此时乘法过大，则需要 MQ 辅助存储，这样，第二条指令就顺利完成！

## 更多微指令过程

- 请你阐述第三条指令发生的过程。值得注意的是，此时 MAR 将 '0000000000000001' 放置 MDR 上以后，该二进制码将由 X 保存，即  $(X)=1$ ，此时发生的过程为： $(ACC)+(X) \rightarrow ACC$ ，仍由 ALU 进行运算。
- 请你阐述第四条指令发生的过程<sup>a</sup>。

<sup>a</sup>注：当指令为存于内存时，IR 将不是向 MAR 寻址，而是将 ACC 的值保存到对应的 MDR 中

## 计算机硬件架构



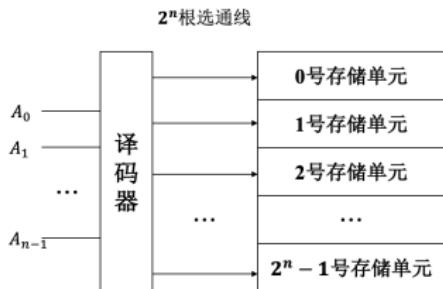
程序运行过程中指令的执行过程

- 计算机硬件需要考虑实际架构情况，CPU 开始对 MAR 进行访问时，首先译码器开始启动工作，并将 MAR 中指向的位置告诉驱动器，令驱动器在存储体中找到对应位置。



内存模型中的问题

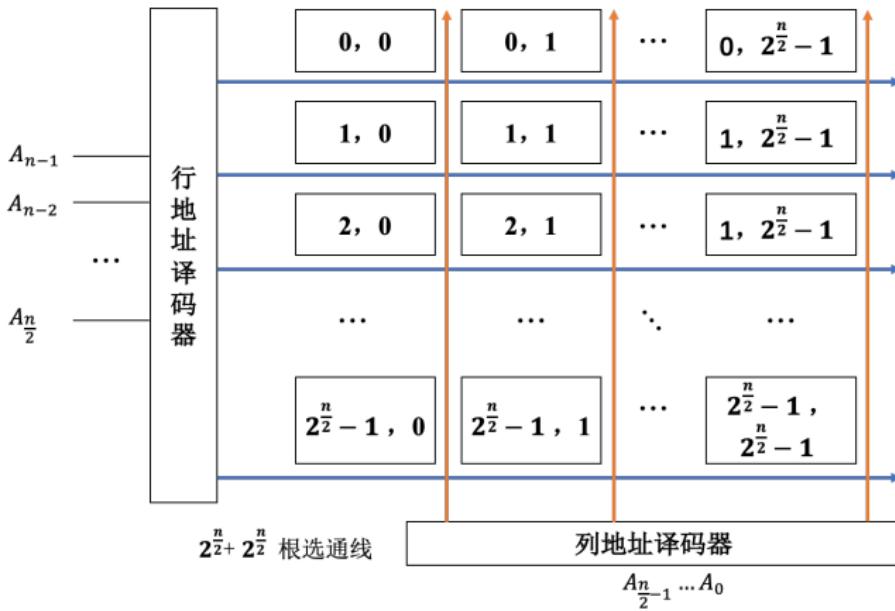
在刚开始的内存模型中，我们用快递驿站的货架来比喻各层级的存储单元，但这有一个坏处：如果每个存储单元都和译码器之间有一条连线，那么  $n$  根地址线就有  $2^n$  种不同的情况，每种情况和存储单元构成一一映射，我们共需要  $2^n$  根选通线，这显然是不可接受的！



- 如何改进这种弊端？一种策略是将一维的存储单元展开，变成矩阵形式，我们引入行地址译码器和列地址译码器，于是横向我们只需要  $2^{\frac{n}{2}}$  根选通线，而纵向也同样需要  $2^{n/2}$  根选通线即可完成等价情况。



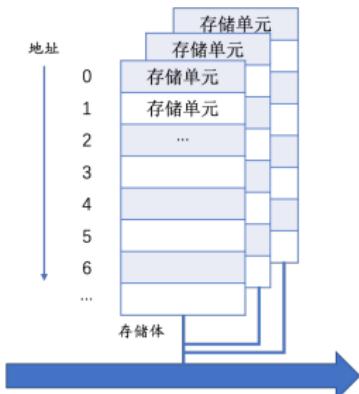
内存中的改进



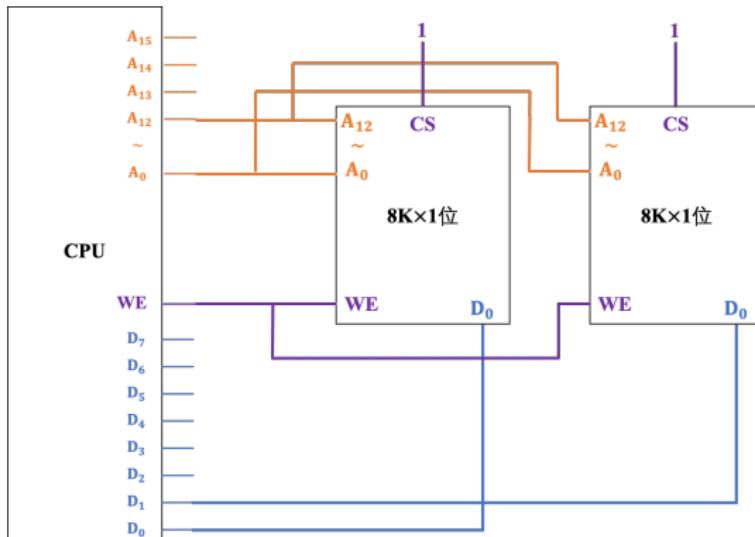
两种扩展

在简易内存模型中，我们将 MAR、MDR 画在主存储器内部，而现在的计算机 MAR、MDR 通常集成在 CPU 内部。前面已经说过，考虑到总线的存在，很多模型就会复杂起来：如果数据总线宽度大于存储芯片字长该怎么办？若想扩展主存字数该怎么办？这两个问题分别对应着位扩展与字扩展。

- 若存储单元的存储字长足够，但每个字节内的位数不够，则需要将若干个存储芯片并联；



图：位扩展：并联

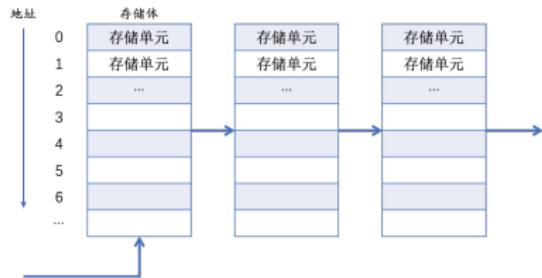


图：位扩展接线图

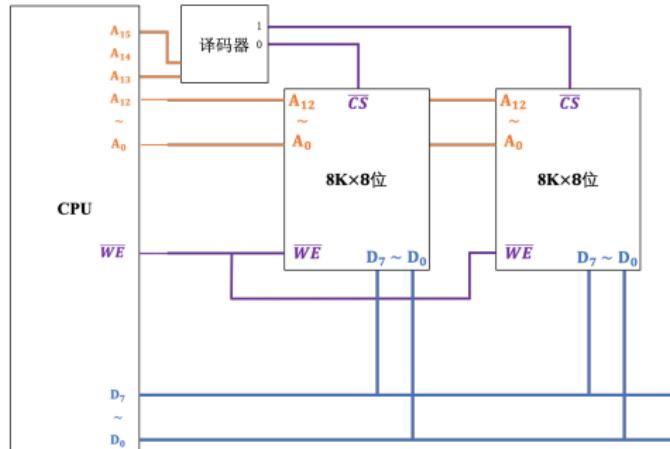


## 位扩展与字扩展

- 若存储单元的存储字长不够，但每个字节内的位数足够，则需要将若干个存储芯片串联；



图：字扩展：串联



图：字扩展接线图

## 扩展后的地址问题

上图中的字扩展省略了后面若干块存储芯片。假设共有 4 块存储芯片进行字扩展，则第 0 块芯片的地址范围为：00 0...0 ~ 00 1...1，最后一块芯片地址范围为：11 0...0 ~ 11 1...1。



## main.c

```
#include<stdio.h>
void mulstore(long ,long ,long *);
int main() {
    long d;mulstore(2, 3, &d) ;
    printf("2* 3 —>%ld \n", d) ;
    return 0;
}
```

## mstore.c

```
long mult2(long a, long b){
    long s = a* b;
    return s;
}
```

- *linux> gcc -Og -o prog main.c mstore.c*
- 编译选项-Og 是用来告诉编译器生成符合原始 C 代码整体结构的机器代码。在实际项目中，为了获得更高的性能，会使用-O1 或者-O2，甚至更高的编译优化选项。但是使用高级别的优化产生的代码会严重变形，导致产生的机器代码与最初的源代码之间的关系难以理解，这里为了理解方便，因此选择-Og 这个优化选项。
- -o 后面跟的参数 prog 表示生成可执行文件的文件名。

# 生成汇编文件

使用以下命令生成汇编文件 mstore.s 并用 vim 打开：

- linux> gcc -Og -S mstore.c

## mstore.c

```
.file "mstore.c"
.text
.globl mulstore
.type mulstore, @function
mulstore:
.LFBO:
.cfi_startproc
pushq %rbx
movq %rdx,%rbx
call %mult2
movq %rax,(%rbx)
popq %rbx
ret
...
```

- 其中以. 开头的行都是指导汇编器和链接器工作的伪指令，可忽略。删除后，剩余汇编代码与源文件中代码是相关的。

## 删除冗余后的汇编指令

### multstore:

```
pushq %rbx
movq %rdx,%rbx
call %mult2
movq %rax,(%rbx)
popq %rbx
ret
```

- ② pushq 指令的意思是将寄存器 rbx 的值压入程序栈进行保存。
- ② 为什么程序一开始要保存寄存器 rbx 的内容？



# 生成汇编文件

在 Intel x86-64 的处理器中包含了 16 个通用目的的寄存器，这些寄存器用来存放整数数据和指针。

```
%rax    %rbx    %rcx    %rdx  
%rsi    %rdi    %rbp    %rsp  
%r8     %r9     %r10    %r11  
%r12    %r13    %r14    %r15
```

上面的 16 个寄存器名字都是以%r 开头的，在详细介绍寄存器的功能之前，我们首先需要搞清楚两个概念：**调用者保存寄存器**和**被调用者保存寄存器**。

func\_A:

```
...  
  
movq $123, %rbx  
call func_B  
addq %rbx, %rax  
  
...  
  
ret
```

Caller

func\_B:

```
...  
  
addq $456 %rbx  
...  
  
ret
```

Callee



# 两种寄存器

## 调用者保存寄存器与被调用者保存寄存器

由于调用了函数 B，寄存器 rbx 在函数 B 中被修改，逻辑上寄存器 rbx 的内容在调用函数 B 的前后应该保持一致，解决这个问题有两个策略：

- ① 函数 A 在调用函数 B 之前，提前保存寄存器 rbx 的内容，执行完函数 B 之后，再恢复寄存器 rbx 原来存储的内容，这种策略就称之为调用者保存；
- ② 函数 B 在使用寄存器 rbx 之前，先保存寄存器 rbx 的值，在函数 B 返回之前，先恢复寄存器 rbx 原来存储的内容，这种策略被称之为被调用者保存。

func\_A:

...

movq \$123, %rbx

保存寄存器rbx的内容

call func\_B

恢复寄存器rbx原来存储的内容

addq %rbx, %rax

...

ret

func\_B:

...

保存寄存器rbx的内容

addq \$456 %rbx

恢复寄存器rbx原来存储的内容

...

ret



太原理工大学

TAIYUAN UNIVERSITY OF TECHNOLOGY

对于具体使用哪一种策略，不同的寄存器定义成不同

**Callee saved:** %rbx, %rbp, %r12, %r13, %r14, %15

**Caller saved:** %r10, %r11

%rax

%rdi, %rsi, %rdx, %rcx, %r8, %r9

- 寄存器 rbx 被定义为被调用者保存寄存器 (callee-saved register)，因此，pushq 就是用来保存寄存器 rbx 的内容。
- 在函数返回之前，使用了 pop 指令，恢复寄存器 rbx 的内容。
- 第二行汇编代码的含义是将寄存器 rdx 的内容复制到寄存器 rbx。

```
long mult2(long, long);
void multstore(long x, long y, long *dest){
    long t = mult2(x, y);
    *dest = t;
}
```

**multstore :**

pushq	%rbx
movq	%rdx,%rbx
call	%mult2
movq	%rax,(%rbx)
popq	%rbx
ret	



根据寄存器用法的定义，函数 multstore 的三个参数分别保存在寄存器 rdi、rsi 和 rdx 中，这条指令执行结束后，寄存器 rbx 与寄存器 rdx 的内容一致，都是 dest 指针所指向的内存地址。movq 指令的后缀“q”表示数据的大小。

## 渊源与缩写

早期的机器是 16 位，后才扩展到 32 位。Intel 用字 (word) 来表示 16 位的数据类型，因此，32 位数据类型称为双字，64 位称为四字。下表给出了 C 语言的基本类型对应的汇编后缀表示，movq 的“q”表示四字。

C 声明	Intel 数据类型	汇编码后缀	大小 (bytes)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
long	Quad word	q	8
char *	Quad word	q	8
float	Single precision	s	4
double	Double precision	l	8

- GCC 数据传送指令四个变种，分别为：movb、movw、movl 以及 movq。
- 其中，movb 是 move byte 的缩写，表示传送字节；以此类推。

- call 指令对应于 C 代码中的函数调用，这一行代码比较容易理解，该函数的返回值会保存到寄存器 `rax` 中，因此，寄存器 `rax` 中保存了 `x` 和 `y` 的乘积结果。
- 下一条指令将寄存器 `rax` 的值送到内存中，内存的地址就存放在寄存器 `rbx` 中。
- 最后一条指令 `ret` 就是函数返回。

```
long mult2(long, long);
```

```
//x -> %rdi, y -> %rsi, dest -> %rdx
void multstore(long x, long y, long *dest){
    long t = mult2(x, y); //x * y -> %rax
    *dest = t;
}
```

```
multstore:
```

pushq	%rbx
movq	%rdx,%rbx
call	%mult2
movq	%rax,(%rbx)
popq	%rbx
ret	

# 生成机器代码文件

① 我们只需要将编译选项-S 替换成-c，就可将上述文件翻译成机器代码。

- linux> gcc -Og -c mstore.c

- 执行这条命令，即可生成 mstore.c 对应的机器代码文件 mstore.o。由于该文件是二进制格式无法直接查看。这里我们需要借助一个反汇编工具— objdump。汇编器将汇编代码翻译成二进制的机器代码，反汇编器就是机器代码翻译成汇编代码。

② 通过以下命令，我们可以查看 mstore.o 中的相关信息。

- linux> objdump -d mstore.o

③ 通过对反汇编得到的汇编代码与编译器直接产生的汇编代码，可以发现二者存在细微的差异

0000000000000000<multstore>:

0 : 53	push	%rbx	pushq	%rbx
1 : 48 89 d3	mov	%rdx, %rbx	movq	%rdx, %rbx
4 : e8 00 00 00 00	callq	9 <multstore+0x9>	call	mult2
9 : 48 89 03	mov	%rax, (%rbx)	movq	%rax, (%rbx)
c : 5b	pop	%rbx	popq	%rbx
d : c3	retq		ret	

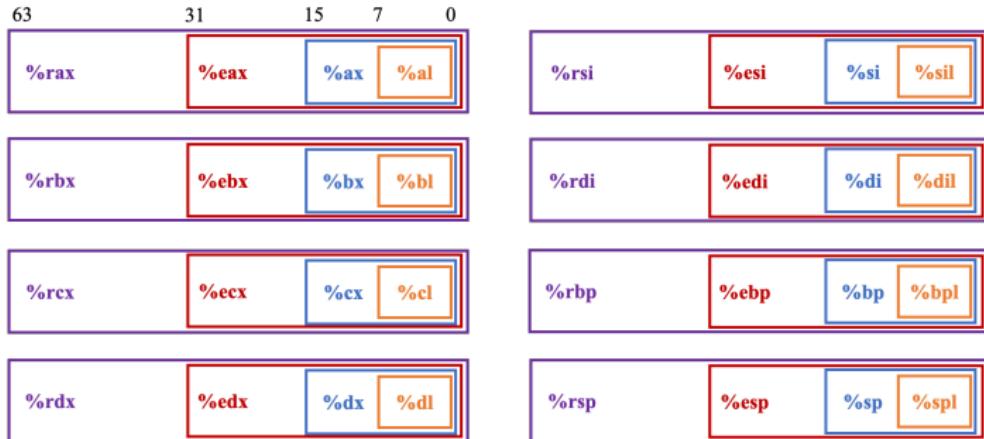
multstore:

反汇编代码省略了很多指令的后缀“q”，但在 call 和 ret 指令添加后缀‘q’，由于 q 只是表示大小指示符，大多数情况下是可以省略的。



## 寄存器发展史

- 最早 8086 的处理器中，包含 8 个 16 位的通用寄存器。每个寄存器都有特殊的功能，它们的名字就反映了不同的用途，当处理器从 16 位扩展到 32 位时，寄存器的位数也随之扩展到了 32 位。直到今天，原来 8 个 16 位寄存器已经扩展成了 64 位。此外还增加了 8 个新的寄存器。



# 寄存器现状

63	%rax	返回值 - Caller saved	0
	%rsi	Argument #2 - Caller saved	
	%rbx	Callee saved	
	%rdi	Argument #1 - Caller saved	
	%rcx	Argument #4 - Caller saved	
	%rbp	Callee saved	
	%rdx	Argument #3 - Caller saved	
	%rsp	Stack pointer	

63	%r8	Argument #5 - Caller saved	0
	%r9	Argument #6 - Caller saved	
	%r10	Caller saved	
	%r11	Caller saved	

## 当代寄存器

在一般的程序中，不同的寄存器扮演着不同的角色，相应的编程规范规定了如何使用这些寄存器。例如寄存器 rax 目来保存函数的返回值，寄存器 rsp 用来保存程序栈的结束位置，除此之外，还有 6 个寄存器可以用来传递函数参数。

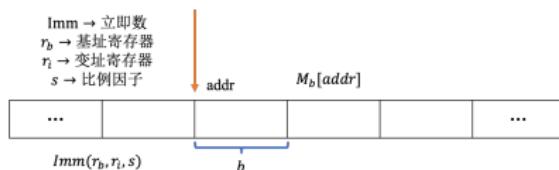
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved



## 操作码和操作数

大多数指令包含两部分：**操作码**和**操作数**。例如指令 movq、addq、subq 这部分被定义为操作码，它决定了 CPU 执行操作的类型；操作码之后的部分是操作数，大多数指令具有一个或者多个操作数。不过像 ret 返回指令，是没有操作数的。

操作码	操作数
movq	(%rdi),%rax
addq	\$8,%rsx
subq	%rdi,%rax
xorq	%rsi,%rdi
ret	



- ① 在 AT&T 格式的汇编中，立即数以 \$ 符号开头，后跟一个 C 定义的整数。
  - ② 操作数是寄存器的情况，即使在 64 位的处理器上，不仅 64 位的寄存器可以作为操作数，32 位、16 位甚至 8 位的寄存器都可以作为操作数。
  - ③ 寄存器带小括号表示内存引用。我们通常将内存抽象成一个字节数组，当需要从内存中存取数据时，需要获得目的数据的起始地址  $addr$ ，以及数据长度  $b$ 。为了简便，通常会省略下标  $b$ 。

- 有效地址是通过立即数与基址寄存器的值相加，再加上变址寄存器与比例因子的乘积。

$$\text{Imm}(r_b, r_i, s) \rightarrow \text{Imm} + R[r_b] + R[r_i] \cdot s$$

- 比例因子 s 的取值必须是 1、2、4 或者 8。实际上比例因子的取值是与源代码中定义的数组类型的是相关的，编译器会根据数组的类型来确定比例因子的数值，例如定义 char 类型的数组，比例因子就是 1，int 类型，比例因子就是 4，至于 double 类型比例因子就是 8。
- 其他的形式的内存引用都是这种普通形式的变种，省略了其中的某些部分，图中列出了内存引用的其他形式，需要特别注意的两种写法是：不带 \$ 符号的立即数和带了括号的寄存器。

Type	Form	Operand value	Name
Memory	$\text{Imm}(r_b, r_i, s)$	$M[\text{Imm} + R[r_b] + R[r_i \cdot s]]$	Scaled indexed
Memory	$\text{Imm}$	$M[\text{Imm}]$	Absolute
Memory	$(r_a)$	$M[R[r_a]]$	Indirect
Memory	$\text{Imm}(r_b)$	$M[\text{Imm} + R[r_b]]$	Base+displacement
Memory	$(r_b, r_i)$	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$(.r_i, s)$	$M[\text{Imm} + R[r_b] + R[r_i]]$	Indexed
Memory	$\text{Imm}(, r_i, s)$	$M[R[r_i] \cdot s]$	Scale indexed
Memory	$\text{Imm}(, r_i, s)$	$M[\text{Imm} + R[r_i] \cdot s]$	Scale indexed
Memory	$\text{Imm}(, r_i, s)$	$M[R[r_b] + R[r_i] \cdot s]$	Scale indexed

## mov 指令含义

mov 指令包含一个源操作数和目的操作数。源操作数可以是一个立即数、一个寄存器或是内存引用。由于目的操作数是用来存放源操作数的内容，所以目的操作数要么是一个寄存器，要么是一个内存引用，而不能是一个立即数。除此之外，x86-64 处理器有一条限制，就是 mov 指令的源操作数和目的操作数不能都是内存的地址，那么当需要将一个数从内存的一个位置复制到另一个位置时，需要两条 mov 指令来完成：第一条指令将内存源位置的数值加载到寄存器；第二条指令再将该寄存器的值写入内存的目的位置。

movb	Move byte (1 Byte)
movw	Move word (2 Bytes)
movl	Move double word (4 Bytes)
movq	Move quad word (8 Bytes)

源操作数	目的操作数
立即数	寄存器
寄存器	内存
内存	

- mov 指令的后缀与寄存器的大小一定得是匹配的，例如寄存器 eax 是 32 位，与双字“l”对应。

## mov 指令

- mov 指令还有几种特殊情况，当 movq 指令的源操作数是立即数时，该立即数只能是 32 位的补码表示，对该数符号位扩展后，将得到的 64 位数传送到目的位置。
  - 这个限制会带来一个问题，当立即数是 64 位时应该如何处理？

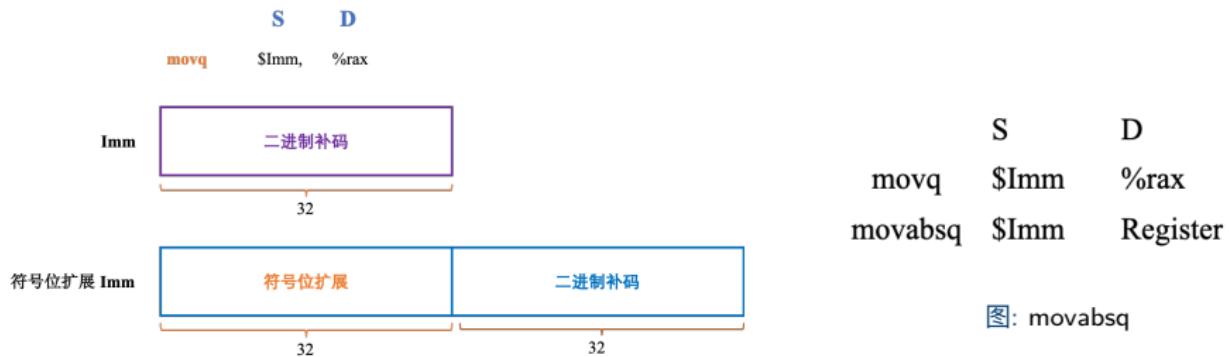


图: 64 位时该如何处理?

- 我们引入一个新的指令 `movabsq`, 该指令的源操作数可以是任意的 64 位立即数, 而目的操作数只能是寄存器。

# mov 指令

- ① 我们通过一个例子来看一下使用 mov 指令进行数据传送时，对目的寄存器的修改结果是怎样。首先使用 movabsq 指令将一个 64 位的立即数复制到寄存器 rax。

movabsq, \$0x0011223344556677, %rax

- ② 此时，寄存器 rax 内保存的数值如图所示。

63	31	15	7	0
00	11	22	33	44 55 66 77

- ③ 接下来，使用 movb 指令将立即数-1 复制到寄存器 al，寄存器 al 的长度为 8，与 movb 指令所操作的数据大小一致。

movb, \$ - 1 %al

- ④ 此时寄存器 rax 的低 8 位发生了改变。

63	31	15	7	0
00	11	22	33	44 55 66 FF

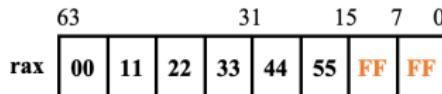
- ⑤ 第三条指令 movw 是将立即数-1 复制到寄存器 ax。

movw \$ - 1 %ax



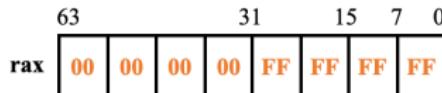
# mov 指令

- ① 此时寄存器 `rax` 的低 16 位发生了改变。



- ② 当指令 `movl` 将立即数 -1 复制到寄存器 `eax` 时，此时寄存器 `rax` 不仅仅是低 32 位发生了变化，而且高 32 位也发了变化。

`movl $-1 %eax`



- ③ 当 `movl` 的目的操作数是寄存器时，它会把该寄存器的高 4 字节设置为 0，这是 x86-64 处理器的一个规定，即任何位寄存器生成 32 位值的指令都会把该寄存器的高位部分置为 0<sup>1</sup>。

<sup>1</sup>以上介绍的都是源操作数与目的操作数的大小一致的情况。

## 源操作数的数位小于目的操作数

当源操作数的数位小于目的操作数时，需要对目的操作数剩余的字节进行零扩展或者符号位扩展。

- 零扩展数据传送指令有 5 条，其中字符 z 是 zero 的缩写。指令最后两个字符都是大小指示符，第一个字母表示源操作数的大小，第二个字母表示目的操作数的大小。
- 符号位扩展传送指令有 6 条，其中字符 s 是 sign 的缩写，同样指令最后的两个字符也是大小指示符。符号位扩展还有一条没有操作数的特殊指令 cltq，该指令的源操作数总是寄存器 eax，目的操作数总是寄存器是 rxz。

指令	影响	描述
MOVS(Z) S, R	$R \leftarrow \text{Sign}(\text{Zero})\text{Extend}(S)$	Move with sign(zero) extension
movsbw		Move Sign(Zero)-extended byte to word
movsbl		Move Sign(Zero)-extended byte to Double word
movswl		Move Sign(Zero)-extended word to Double word
movsbq		Move Sign(Zero)-extended byte to Quad word
movswq		Move Sign(Zero)-extended word to Quad word
movslq		Move Sign-extended Double word to quad word

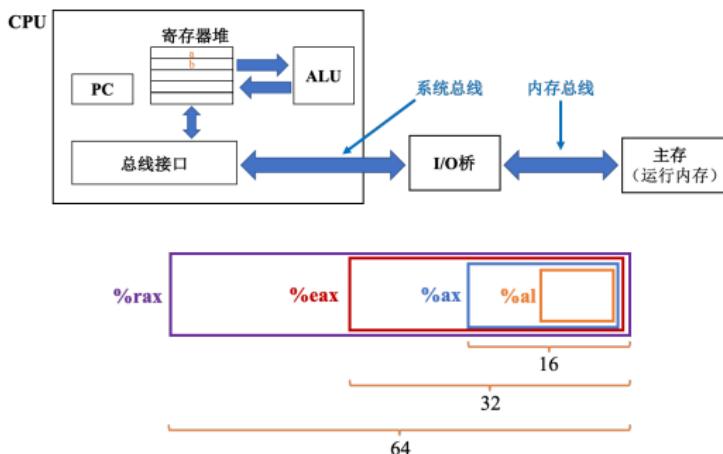
- 对比零扩展和符号扩展，我们可以发现符号扩展比零扩展多一条 4 字节到 8 字节的扩展指令，为什么零扩展没有 movzlq 的指令呢？是因为这种情况的数据传送可以使用 movl 指令来实现。

cltq      movslq    %eax,    %rax



## 数据传送指令过程

程序的执行过程中需要在 CPU 和内存之间进行频繁的数据存取。例如执行加法操作  $c=a+b$ 。首先通过 CPU 执行数据传送指令将  $a$  和  $b$  的值从内存读到寄存器内。以 x86-64 处理器为例，寄存器  $rax$  的大小是 64 个比特位（8 个字节），如果变量  $a$  是 long 类型，需要占用 8 个字节，因此，寄存器  $rax$  全部的数据位都用来保存变量  $a$ ；如果变量  $a$  是 int 类型，那么只需要用 4 个字节来存储该变量，那么只需要用到寄存器的低 32 位就够了；如果变量  $a$  是 short 类型，则只需要用到寄存器的低 16 位。



- 如果使用寄存器  $rax$  全部的 64 位，用符号  $%rax$  来表示；如果是只用到低 32 位，可用符号  $%eax$  来表示；对于低 16 位和低 8 位的，分别用  $%ax$  和  $%al$  来表示。
- 虽然用了不同的表示符号，但实际上只是针对同一寄存器的不同数位进行操作，处理器完成加法运算之后，再通过一条数据传送指令将计算结果保存到内存。
- 数据传送在计算机系统中非常频繁，因此了解数据传输指令过程对理解计算机系统大有帮助。

## 代码示例

```
int main(){
    long a = 4;
    long b = exchange(&a, 3);
    printf("a = %ld, b = %ld\n", a, b);
    return 0;
}
long exchange(long *xp, long y){
    long x = *xp;
    *xp = y;
    return x;
}
```

- 变量 a 的值会替换成 3，变量 b 将保存变量 a 原来的值 4。重点看函数 exchange 所对应的汇编指令；

exchange:

```
xp in %rdi, y in %rsi
movq (%rdi), %rax
movq %rsi, (%rdi)
ret
```

- 函数 exchange 由三条指令实现，包括两条数据传送指令和一条返回指令。寄存器 rdi 和 rsi 分别用来保存函数传递的第一个参数和第二个参数，因此，寄存器 rdi 中保存了 xp 的值，寄存器 rsi 保存了变量 y 的值。

# exchange 实现过程

- 第一条 mov 指令从内存中读取数值到寄存器，内存地址保存在寄存器 rdi 中，目的操作数是寄存器 rax，这条指令对应于代码的 `long x = *xp;` 由于最后函数 exchange 需要返回变量 x 的值，所以这里直接将变量 x 放到寄存器 rax 中。
- 第二条 mov 指令将变量 y 的值写到内存里，变量 y 存储在寄存器 rsi 中，内存地址保存在寄存器 rdi 中，也就是 xp 指向的内存位置。这条指令对应函数 exchange 中的 `*xp = y;`

```
exchange:  
    xp in %rdi, y in %rsi  
    movq (%rdi), %rax      Memory—>Register  
    movq %rsi, (%rdi)      Register—>Memory  
    ret
```

通过这个例子，我们可以看到 C 语言中所谓的指针其实就是地址。

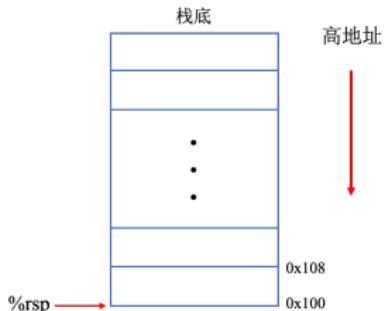
## 程序栈

两个数据传送指令需要借助程序栈，程序栈本质上是内存中的一个区域。例如我们需要保存寄存器 rax 内存储的数据 0x123，可以使用 pushq 指令把数据压入栈内。该指令执行的过程可以分解为两步：



## exchange 实现过程

- ① 首先指向栈顶的寄存器的 `rsp` 进行一个减法操作，例如压栈之前，栈顶指针 `rsp` 指向栈顶的位置，此处的内存地址 `0x108`；压栈的第一步就是寄存器 `rsp` 的值减 8，此时指向的内存地址是 `0x100`。



- ② 然后将需要保存的数据复制到新的栈顶地址，此时，内存地址 0x100 处将保存寄存器 `rax` 内存储的数据 0x123。实际上 `pushq` 的指令等效于图中 `subq` 和 `movq` 这两条指令。它们之间的区别是在于 `pushq` 这一条指令只需要一个字节，而 `subq` 和 `movq` 这两条指令需要 8 个字节。

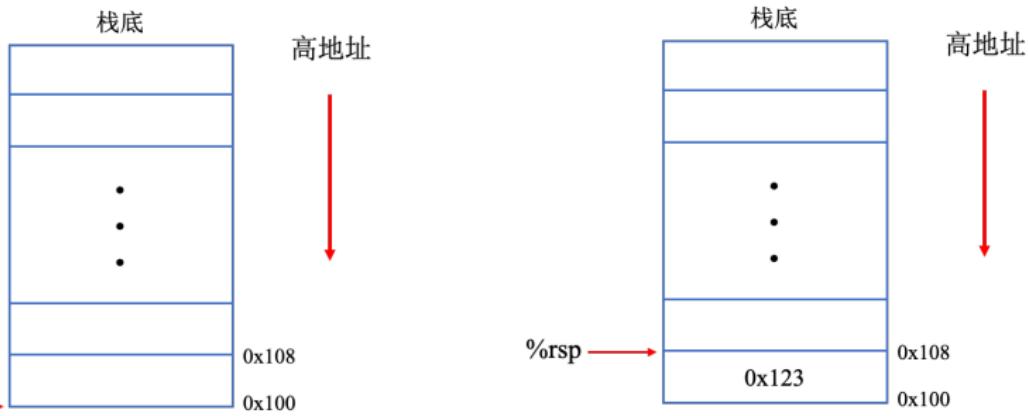
```
%rax  
pushq %rax  
subq $8, %rsp  
movq %rax, (%rsp)
```

## push 与 pop

说到底，push 指令的本质还是将数据写入到内存中，那么与之对应的 pop 指令就是从内存中读取数据，并且修改栈顶指针。例如图中这条 popq 指令就是将栈顶保存的数据复制到寄存器 rbx 中。pop 指令的操作也可以分解为两步：



- ① 首先从栈顶的位置读出数据，复制到寄存器 rbx。此时，栈顶指针 rsp 指向的内存地址是 0x100。
- ② 然后将栈顶指针加 8，pop 后栈顶指针 rsp 指向的内存地址是 0x108。



- 因此 `pop` 操作也可以等效 `movq` 和 `addq` 这两条指令。实际上 `pop` 指令是通过修改栈顶指针所指向的内存地址来实现数据删除的，此时，内存地址 **0x100** 内所保存的数据 **0x123** 仍然存在，直到下次 `push` 操作，此处保存的数值才会被覆盖。

## 加载有效地址 leaq

leaq 为加载有效地址指令， q 表示地址的长度是四个字，由于 x86-64 位处理器上，地址长度都是 64 位，因此不存在 leab、leaw 这类有关大小的变种。格式为：

leaq S, D → Load Effective Address

- 例如下列指令是将有效地址复制到寄存器 `rax` 中

`leaq 7(%rdx,%rdx,4),%rax`

- 源操作数看上去与内存引用的格式类似，有效地址的计算方式与之前讲到的内存地址的计算方式一致，可以通过下列中的公式计算得到

$$\text{Imm}(r_b, r_i, s) \rightarrow \text{Imm} + R[r_b] + R[r_i] \cdot s$$

- 假设寄存器 `rdx` 内保存的数值为  $x$ ，那么有效地址的值为  $7 + \%rdx + \%rdx * 4 = 7 + 5x$ 。注意，对于 `leaq` 指令所执行的操作并不是去内存地址  $(5x+7)$  处读取数据，而是将有效地址  $(5x+7)$  这个值直接写入到目的寄存器 `rax`。
- 除了加载有效地址的功能，`leaq` 指令还可以用来表示加法和有限的乘法运算：

# leaq 实现算术运算

leaq.c

```
long scale(long x, long y, long z){  
    long t = x + 4*y + 12*z;  
    return t;  
}
```

- 经过编译后，这段代码是通过三条 leaq 指令来实现：

scale:

leaq	(%rdi, %rsi, 4),	%rax
leaq	(%rdx, %rdx, 2),	%rdx
leaq	(%rax, %rdx, 4),	%rax
ret		

- $x, y, z$  分别保存在  $rdi$ 、 $rsi$  以及  $rdx$  中，第一条指令由 leaq 将数值保存到寄存器  $rax$  中。
- $z*12$  的乘法运算会有一些复杂，需要分成两步：计算  $3*z$  的数值，第二条的 leaq 指令执行完毕，此时寄存器  $rdx$  中保存的值是  $3z$ ；再把  $3z$  作为一个整体乘以 4。

scale:

x in %rdi, y in %rsi,	z in %rdx	
leaq (%rdi, %rsi, 4),	%rax	$\rightarrow \%rdi+4*\%rsi=x+4*y$
leaq (%rdx, %rdx, 2),	%rdx	$\rightarrow \%rdx+2*\%rdx=z+2*z$
leaq (%rax, %rdx, 4),	%rax	$\rightarrow \%rax+4*\%rdx$
ret		$=(x+4*y)$

- 之所以不使用下列这条指令直接得出结果，是因为比例因子取值只能是 1,2,4,8 这四个数中的一个。

leaq (%rax,%rdx,12), %rax  $\rightarrow \%rax + 12 * \%rdx = (x + 4 * y) + 12 * z$



## 一元和二元操作指令

## 两种操作指令对比

一元操作指令只有一个操作数，因此该操作数既是源操作数也是目的操作数，操作数可以是寄存器，也可以是内存地址。而二元操作指令包含两个操作数，第一个操作数是源操作数，这个操作数可以是立即数、寄存器或者内存地址；第二个操作数既是源操作数也是目的操作数，这个操作数可以是寄存器或者内存地址，但不能是立即数。

表：一元操作指令

指令	影响	描述
$INC D$	$D \leftarrow D + 1$	加 1
$DEC D$	$D \leftarrow D - 1$	减 1
$NEG D$	$D \leftarrow -D$	取负
$NOT D$	$D \leftarrow \sim D$	取补

表·二元操作指令

指令	影响	描述
$ADD S, D$	$D \leftarrow D+S$	加
$SUB S, D$	$D \leftarrow D-S$	减
$IMUL S, D$	$D \leftarrow D*S$	乘
$XOR S, D$	$D \leftarrow D \wedge S$	异或
$OR S, D$	$D \leftarrow D S$	或
$AND S, D$	$D \leftarrow D\&S$	与



## 内存与寄存器中保存数据的方式

0x100		0x108		0x110		0x118	
...	0xFF	0xAB	0x13	0x11	0x3	...	...
	%rax $\leftarrow$ 0x100		%rcx $\leftarrow$ 0x1		%rdx $\leftarrow$ 0x3		

addq	%rcx, (%rax) → Mem[0x100] = Mem[0x100] + R[%rcx]
subq	%rdx, 8(%rax) → Mem[0x108] = Mem[0x108] - R[%rdx]
incq	16(%rax), → Mem[0x110] = Mem[0x110] + 1
subq	%rdx, %rax → R[%rax] = R[%rax] - R[%rdx]

图：内存以及寄存器中所保存的数据

- ④ 加法指令 addq 是将内存地址 0x100 内的数据与寄存器 rcx 相加，二者之和再存储到内存地址 0x100 处，该指令执行完毕后，内存地址 0x100 处所存储的数据由 0xFF 变成 0x100。

	0x100	0x108	0x110	0x118	
...	0x100	0xAB	0x13	0x11	...
	%rax $\leftarrow$ 0x100	%rcx $\leftarrow$ 0x1	%rdx $\leftarrow$ 0x3		



# 内存与寄存器中保存数据的方式

- ② 减法指令 subq 是将内存地址 0x108 内的数据减去寄存器 rdx 内的数据，二者之差在存储到内存地址 0x108 处，该指令执行完毕后，内存地址 0x108 处所存储的数据由 0xAB 变成 0xA8。

	0x100	0x108	0x110	0x118	
...	0x100	0xA8	0x13	0x11	...
	%rax ← 0x100	%rcx ← 0x1	%rdx ← 0x3		

- ③ 对于加一指令 incq，就是将内存地址 0x110 内存储的数据加 1，结果是内存地址 0x110 处所存储的数据由 0x13 变成 0x14。

	0x100	0x108	0x110	0x118	
...	0x100	0xA8	0x14	0x11	...
	%rax ← 0x100	%rcx ← 0x1	%rdx ← 0x3		

- ④ 最后一条加法指令是将寄存器 rax 内的值减去寄存器 rdx 内的值，最终寄存器 rax 的值由 0x100 变成 0xFD。

...	0x100	0xA8	0x13	0x11	...
	%rax ← 0xFD	%rcx ← 0x1	%rdx ← 0x3		



## 左移与右移

左移指令有两个：SAL 和 SHL，二者的效果相同，都是在右边填零；右移指令不同，分为算术右移和逻辑右移，算术右移需要填符号位，逻辑右移需要填零。

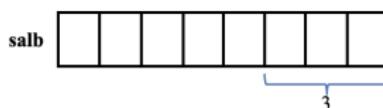
指令	影响	描述
SAL k, D	$D \leftarrow D \ll k$	左移
SHL k, D	$D \leftarrow D \ll k$	左移，等同于 SAL
SAR k, D	$D \leftarrow D >>_A k$	算数右移
SHR k, D	$D \leftarrow D >>_L k$	逻辑右移

- 对于移位量  $k$ ，可以是一个立即数，或者是放在寄存器 cl 中的数，对于移位指令只允许以特定的寄存器 cl 作为操作数，其他寄存器不行，这里需要特别注意一下。

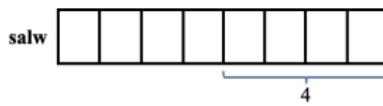


以此类推，双字对应的是低 5 位，四字对应的是低 6 位。

- 由于寄存器 cl 的长度为 8，原则上移位量的编码范围可达  $2^8 - 1$ (255)，实际上，对于  $w$  位的操作数进行移位操作，移位量是由寄存器 cl 的低  $m$  位来决定，也就是说，对于指令 salb，当目的操作数是 8 位，移位量由寄存器 cl 的低 3 位来决定。



- 对于指令 salw，移位量则是由寄存器 cl 的低 4 位来决定。



## 移位操作应用

```
long arith(long x, long y, long z){  
    long t1 = x ^ y;  
    long t2 = z * 48;  
    long t3 = t1 & 0xF0F0F0F;  
    long t4 = t2 - t3;  
    return t4;  
}
```

- 我们重点看一下  $z * 48$  这行代码所对应的汇编指令，整个过程被分为两步：

```
long t2 = z * 48;  
leaq (%rdx,%rdx,2), %rax  
salq $4, %rax
```

- 首先通过 leaq 指令计算  $3 * z$ ，结果保存到寄存器 rax 中

leaq (%rdx,%rdx,2), %rax  
 $R[\%rdx] + R[\%rdx]*2 = 3*z \rightarrow \%rax$

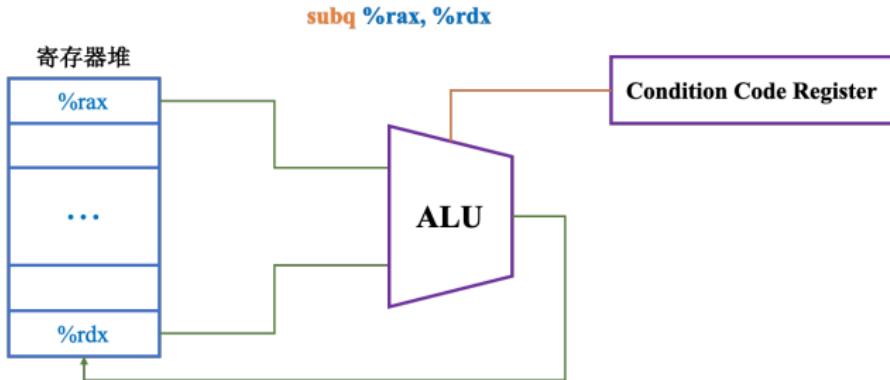
- 第二步，将寄存器 rax 进行左移 4 位，左移 4 位的操作是等效于乘以 2 的四次方，也就是乘以 16。

salq \$4, %rax  
 $2^4 * R[\%rax] = 16 * (3 * z) = 48 * z$

- 为什么编译器不直接使用乘法指令来实现这个运算呢？主要是因为乘法指令的执行需要更长的时间，因此编译器在生成汇编指令时，会优先考虑更高效的方式。

## 条件码

ALU 除了执行算术和逻辑运算指令外，还会根据该运算的结果去设置条件码寄存器。条件码寄存器它是由 CPU 来维护的，长度是单个比特位，它描述了最近执行操作的属性。



- 假如 ALU 执行两条连续的算术指令：



t1: `addq %rax, %rbx`  
t2: `subq %rcx, %rdx`

# 条件码

- t1 和 t2 表示时刻, t1 时刻条件码寄存器中保存的是指令 1 的执行结果的属性, t2 时刻, 条件码寄存器的内容被下一条指令所覆盖。

CF	ZF	...	SF	OF
----	----	-----	----	----

CF —— Carry Flag (进位标志)

ZF —— Zero Flag (零标志)

SF —— Sign Flag (符号标志)

OF —— Overflow Flag (溢出标志)

- 进位标志, 当 CPU 最近执行的一条指令最高位产生了进位时, 进位标志 (CF) 会被置为 1, 它可以用来检查无符号数操作的溢出。
- 零标志, 当最近操作的结果等于零时, 零标志 (ZF) 会被置 1。
- 符号标志, 当最近的操作结果小于零时, 符号标志 (SF) 会被置 1
- 溢出标志, 针对有符号数, 最近的操作导致正溢出或者负溢出时溢出标志 (OF) 会被置 1。

- 条件码寄存器的值是由 ALU 在执行算术和运算指令时写入的, 下图中的这些算术和逻辑运算指令都会改变条件码寄存器的内容。

Unary Operations	Binary Operations	Shift Operations
<i>INC D</i>	<i>ADD S, D</i>	<i>SAL k, D</i>
<i>DEC D</i>	<i>SUB S, D</i>	<i>SHL k, D</i>
<i>NEG D</i>	<i>IMUL S, D</i>	<i>SAR k, D</i>
	<i>OR S, D</i>	<i>SHR k, D</i>
	<i>XOR S, D</i>	
	<i>AND S, D</i>	



## 条件码

对于不同的指令也定义了相应的规则来设置条件码寄存器。例如逻辑操作指令 xor，进位标志 (CF) 和溢出标志 (OF) 会置 0；对于加一指令和减一指令会设置溢出标志 (OF) 和零标志 (ZF)，但不会改变进位标志 (CF)。

除此之外，还有两类指令可以设置条件码寄存器：cmp 指令和 test 指令。

- cmp 指令是根据两个操作数的差来设置条件码寄存器。cmp 指令和减法指令 (sub) 类似，也是根据两个操作数的差来设置条件码，二者不同的是 cmp 指令只是设置条件码寄存器，并不会更新目的寄存器的值。
- test 指令和 and 指令类似，同样 test 指令只是设置条件码寄存器，而不改变目的寄存器的值。

```
int comp(long a, long b){  
    return (a == b);  
}
```



comp:  
comq %rsi, %rdi  
sete %al  
movzbl %al, %eax  
ret

```
int comp(long a, long b){  
    return (a==b);  
}
```

comp:  
a in %rdi, b in %rsi  
comq %rsi, %rdi  
sete %al  
movzbl %al, %eax  
ret



## sete

通常情况下，我们并不会直接去读条件码寄存器，而是根据条件码的某种组合，通过 set 类指令，将一个字节设置为 0 或者 1。在这个例子中，指令 sete 根据需标志 (ZF) 的值对寄存器 al 进行赋值，后缀 e 是 equal 的缩写。如果零标志等于 1，指令 sete 将寄存器 al 置为 1；如果零标志等于 0，指令 sete 将寄存器 al 置为 0。

comp:

a in %rdi, b in %rsi

comq %rsi, %rdi → a-b

sete %al

if ZF=1, %al=1

if ZF=0, %al=0

movzbl %al, %eax

ret

- 下面看了另一段代码及其汇编：

```
int comp(char a, char b){  
    return (a < b);  
}
```

comp:

comq %sil, %rdil

sete %al

movzbl %al, %eax

ret

- 然后 mov 指令对寄存器 al 进行零扩展，最后返回判断结果

- 对比前面相等的情况，可以发现指令有些不同：`sete` 变成了指令 `setl`，指令 `setl` 的含义是如果  $a < b$ ，将寄存器 `al` 设置为 1，其中后缀 `l` 是 `less` 的缩写，表示“在小于时设置”，而非表示大小 `long word`。
- 相对于相等的情况，判断小于的情况要稍微复杂一点。需要根据符号标志 (SF) 和溢出标志 (OF) 的异或结果来判定。
- 两个有符号数相减，当没有发生溢出时，如果  $a < b$ ，结果为负数，那么符号标志 (SF) 被置为 1；如果  $a > b$ ，结果为正数，那么符号标志 (SF) 就不会被置 1。

Case 3 :  $a < b$   
 $t < 0 \quad a = -2 \quad b = 127$   
 $t = 127 > 0, \quad SF = 0 \quad OF = 1 \quad SF \wedge OF = 1$

- 溢出后，符号标志 SF 不会置一，但溢出标志 OF 会置一。因此仅仅通过符号标志无法判断  $a$  是否小于  $b$ 。
- 当  $a=1, b=-128$ ，由于发生了正溢出，结果  $t=-127$ ，虽然  $a > b$ ，但是由于溢出导致了结果  $t$  小于 0，此时符号标志 (SF) 和溢出标志 (OF) 都会被置为 1。

Case 4 :  $a > b$   
 $a = 1 \quad b = -128$   
 $t = -127 < 0, \quad SF = 1 \quad OF = 1 \quad SF \wedge OF = 0$

## 符号标志和溢出标志

根据符号标志 (SF) 和溢出标志 (OF) 的异或结果，可以对 a 小于 b 是否为真做出判断。对于无符号数的比较情况，需要注意指令 cmp 会设置进位标志，因而针对无符号数的比较，采用的是进位标志和零标志的组合，具体的条件码组合如下表所示：

指令	影响	描述
setg D	$D \leftarrow \sim(SF \wedge OF) \& \sim ZF$	Greater(signed >)
setge D	$D \leftarrow \sim(SF \wedge OF)$	Greater or equal(signed >=)
setl D	$D \leftarrow SF \wedge OF$	Less(signed <)
setle D	$D \leftarrow (SF \wedge OF)   ZF$	Less or equal(signed <=)

指令	影响	描述
seta D	$D \leftarrow \sim CF \& \sim ZF$	Above(unsigned >)
setae D	$D \leftarrow \sim CF$	Above or equal(unsigned >=)
setb D	$D \leftarrow CF$	Below(unsigned <)
setbe D	$D \leftarrow CF   ZF$	Below or equal(unsigned <=)

- 这些条件码的组合并不需要去记住，了解条件语句的底层实现，这对我们深入理解整个计算机系统会有一定的帮助。

## 跳转指令

```
long absdiff_se(long x, long y){  
    long result;  
    if(x < y){ result = y - x;}  
    else{ result = x - y;}  
    return result;  
}
```

absdiff\_se:

cmpq	%rsi, %rdi
jl .L4	
movq	%rdi, %rax
subq	%rsi, %rax
ret	

- 条件语句  $x$  小于  $y$  由指令 cmp 来实现，指令 cmp 会根据  $(x-y)$  的结果来设置符号标志 (SF) 和溢出标志 (OF)。跳转指令 jl 根据符号标志 (SF) 和溢出标志 (OF) 的异或结果来判断究竟是顺序执行，还是跳转到 L4 处执行。当  $x$  大于  $y$  时，指令顺序执行后返回执行结果，L4 处的指令不会被执行；当  $x$  小于  $y$  时，程序跳转到 L4 处执行，然后返回执行结果，跳转指令会根据条件寄存器的某种组合来决定是否进行跳转。

指令	跳转条件	描述
jg Label	$\sim(SF \wedge OF) \& \sim ZF$	Greater(signed $>$ )
jge Label	$\sim(SF \wedge OF)$	Greater or equal(signed $\geq$ )
jl Label	$SF \wedge OF$	Less(signed $<$ )
jle Label	$(SF \wedge OF) \mid ZF$	Less or equal(signed $\leq$ )



- if-else 语句在当代处理器上表现出的性能并不理想，一种替代策略是：使用数据的条件转移代替控制的条件转移。还是针对两个数差的绝对值问题，给出了另外一种实现方式：

## cmovdiff\_se.c

```
long cmovdiff_se(long x, long y){  
    long rval = y - x;  
    long eval = x - y;  
    long ntest = x >= y;  
    if(ntest){rval = eval;}  
    return rval;  
}
```

## cmovdiff\_se:

movq	%rsi, %rdx
sub	%rdi, %rdx
movq	%rdi, %rax
subq	%rsi, %rax
cmpq	%rsi, %rdi
cmovge	%rdx, %rax
ret	

- 我们既要计算  $y-x$  的值，也要计算  $x-y$  的值，分别用两个变量来记录结果，然后再判断  $x$  与  $y$  的大小，根据测试情况来判断是否更新返回值。这两种写法看上去差别不大，但第二种效率更高。第二种代码的汇编指令如下

前面这几条指令都是普通的数据传送和减法操作。cmovge 是根据条件码的某种组合来进行有条件的传送数据，当满足规定的条件时，将寄存器 rdx 内的数据复制到寄存器 rax 内。在这个例子中，只有当 x 大于等于 y 时，才会执行这一条指令。

cmpq %rsi, %rdi → compare x:y

cmovge %rdx, %rax → ~ (SF^OF)

表：其他传送指令

指令	移动条件	描述
cmovg S, R	~ (SF^OF) & ~ ZF	Greater(signed >)
cmovge S, R	~ (SF^OF)	Greater or equal(signed >=)
cmovl S, R	SF^OF	Less(signed <)
cmovle S, R	(SF^OF) ZF	Less or equal(signed <=)

- 为什么基于条件传送的代码会比基于跳转指令的代码效率高？这里涉及到现代处理器的指令流水线。当遇到条件跳转时，处理器会根据分支预测器来猜测每条跳转指令是否执行，当发生错误预测时，会浪费大量的时间，导致程序性能严重下降。

C 语言中提供了三种循环结构，即 do-while、while 以及 for 语句，汇编语言中没有定义专内的指令来实现循环结构，循环语句是通过条件测试与跳转的结合来实现的。下面，我们分别用这三种循环结构来实现 N 的阶乘：

## do-while 循环

```
long fact_do(long n){  
    long result = 1;  
    do{  
        result *= n;  
        n = n - 1;  
    } while(n>1)  
    result result;  
}
```

fact\_do:

movl	n in %rdi
	\$1, %eax
.L2	
imulq	%rdi, %rax
subq	\$1, %rdi
cmpq	\$1, %rdi
jg.L2	

- 我们可以发现指令 cmp 与跳转指令的组合实现了循环操作：当 n 大于 1 时，跳转到 L2 处执行循环，直到 n 的值算减术和小逻辑到操作 1，循环控制结束。

# 循环

对比 do-while 循环和 while 循环的实现方式，我们可以发现这两种循环的差别在于，N 大于 1 这个循环测试的位置不同。

```
long fact_do(long n){  
    long result = 1;  
    do{  
        result *= n;  
        n = n - 1;  
    }while (n > 1);  
    return result;  
}
```

```
long fact_while(long n){  
    long result = 1;  
    while (n > 1){  
        result *= n;  
        n = n - 1;  
    }  
    return result;  
}
```

图: do-while 与 while 循环

```
long fact_for(long n){  
    long i ;  
    long result = 1 ;  
    for (i=2; i<=n; i++) {  
        result *= i;  
    }  
    return result;  
}  
  
long fact_for_while(long n){  
    long i = 2 ;  
    long result = 1 ;  
    while (i<=n) {  
        result *= i;  
        i++;  
    }  
    return result;  
}
```

图: while 与 for 循环

do-while 循环是先执行循环体的内容，然后再进行循环测试，while 循环则是先进行循环测试，根据测试结果是否执行循环体内容。我们将这个 for 循环转换成 while 循环。



# for 循环

对比 for 循环和 while 循环产生的汇编代码，可以发现除了这一句跳转指令不同，其他部分都是一致的。

fact\_for:

    movl \$1, %eax

    movl \$2, %edx

    jmp .L2

.L3

    imulq %rdx, %rax

    addq \$1, %rdx

.L2

    cmpq %rdi, %rdx

    jle .L3

    rep ret

fact\_for\_while:

    movl \$1, %eax

    movl \$2, %edx

    jmp .L2

.L3

    imulq %rdx, %rax

    addq \$1, %rdx

.L2

    cmpq %rdi, %rdx

    jl .L3

    rep ret

- 这两个汇编代码是采用-Og 选项产生的。
- 综上所述，三种形式的循环语句都是通过条件测试和跳转指令来实现。

# switch 语句

## switch

```
void switch_eg(long x, long n, long *dest){  
    long val = x;  
    switch(n){  
        case 0: val *= 13; break;  
        case 2: val += 10; break;  
        case 3: val += 11; break;  
        case 4:  
        case 6: val += 11; break;  
        default: val = 0;  
    }  
    *dest = val;  
}
```

## switch\_eg:

n in %rsi

cmpq \$6, %rsi compare n:6

ja .L8

leaq .L4(%rip), %rcx

movslq (%rcx, %rsi, 4), %rax

addq %rcx %rax

jmp \*%rax



太原理工大学

# switch 语句

- ① 指令 cmp 判断参数 n 与立即数 6 的大小，如果 n 大于 6，程序跳转到 default 对应的 L8 程序段。case0 case6 的情况，可以通过跳转表来访问不同分支。代码将跳转表声明为一个长度为 7 的数组，每个元素都是一个指向代码位置的指针。
- ② 数组的长度为 7，是因为需要覆盖 Case0 Case6 的情况，对重复的情况 case4 和 case6，使用相同的标号。
- ③ 对于缺失的 case1 和 case5 的情况，使用默认情况的标号。

.L4:

.long: .L3-.L4 → Case 0  
.long: .L8-.L4 → Case 1  
.long: .L5-.L4 → Case 2  
.long: .L6-.L4 → Case 3  
.long: .L7-.L4 → Case 4  
.long: .L8-.L4 → Case 5  
.long: .L7-.L4 → Case 6

## switch 总结

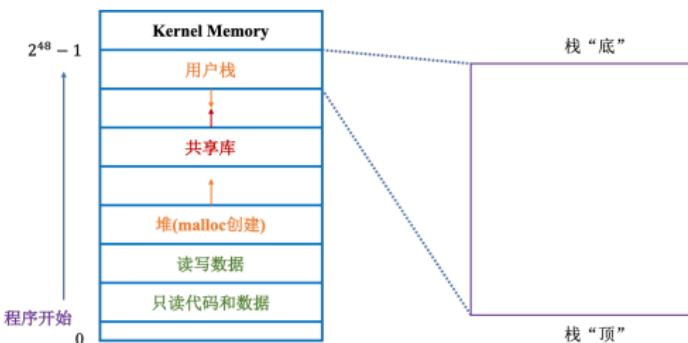
在这个例子中，程序使用跳转表来处理多重分支，甚至当 switch 有上百种情况时，虽然跳转表的长度会增加，但是程序的执行只需要一次跳转也能处理复杂分支的情况，与使用一组很长的 if-else 相比，使用跳转表的优点是执行 switch 语句的时间与 case 的数量是无关的。因此在处理多重分支时，与一组很长的 if-else 相比，switch 的执行效率要高。



过程是什么？

在大型软件的构建过程中，需要对复杂功能进行切分。过程提供了一种封装代码的方式，它可以隐藏某个行为的具体实现，同时提供清晰简洁的接口定义。在不同的编程语言中，过程的具体实现又是多种多样的。例如 C 语言中的函数，Java 语言中的方法等。我们以 C 语言中的函数调用为例，介绍一下过程的机制，为了方便讨论，假设函数 P 调用函数 Q，函数 O 执行完返回函数 P，这一系列操作包括图中的一个或者多个机制。

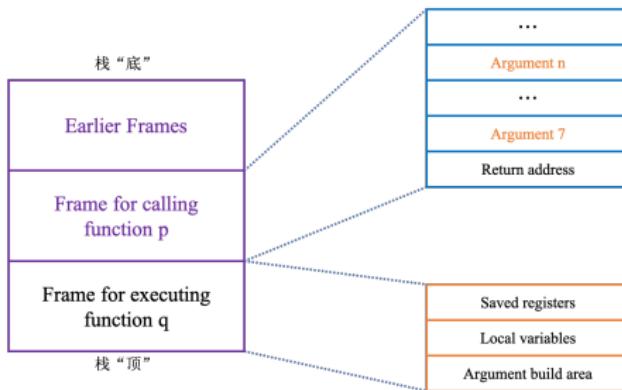
```
long P0    •传递控制  
{  
    ...    •传递数据  
}  
  
long Q0    •分配和释放内存  
{  
    ...  
}
```



图：栈为函数调用提供了后进先出的内存管理机制。在函数 P 调用函数 Q 的例子中，当函数 Q 正在执行时，函数 P 以及相关调用链上的函数都会被暂时挂起。

## 栈帧

当函数执行所需要的存储空间超出寄存器能够存放的大小时，就会借助栈上的存储空间，我们把这部分存储空间称为函数的栈帧。对于函数 P 调用函数 Q 的例子，包括较早的帧、调用函数 P 的帧，还有正在执行函数 Q 的帧。



- 函数 P 调用函数 Q 时，会把返回地址压入栈中，该地址指明了当函数 Q 执行结束返回时要从函数 P 的哪个位置继续执行。这个返回地址的压栈操作并不是由指令 push 来执行的，而是由函数调用 call 来实现的。

- 我们以 main 函数调用 multstore 函数为例来解释一下指令 call 和指令 ret 的执行情况

## multstore

```
#include<stdio.h>
void multstore(long, long, long *);
int main(){
    long d;
    multstore(2, 3, &d);
    printf("2 * 3 -> %d\n", d);
    return 0;
}
long mult2(long a, long b){
    long s = a * b;
    return s;
}
```

## mult2

```
long mult2(long, long);
void multstore(long x, long y, long z){
    long t = mult2(x, y);
    *dest = t;
}
```

由于涉及地址的操作，我们需要查看这两个函数的反汇编代码。

- linux> gcc -Og -o prog main.c mstore.c
- linux> objdump -d proc

# 转移控制

- 右侧为相关部分的反汇编代码：
- 这一条 call 指令对应 multstore 函数的调用。

6fb: e8 41 00 00 00 callq 741 <multstore>

0000000000000006da<main>:

...  
6fb: e8 41 00 00 00 callq 741 <multstore>

700: 48 8b 14 24 mov (%rsp), %rdx

000000000000000741<multstore>:

...  
741: 53 push %rbx

742: 48 89 d3 mov %rdx, %rbx

## 执行过程

指令 call 不仅要将函数 multstore 的第一条指令的地址写入到程序指令寄存器 rip 中，以此实现函数调用；同时还要将返回地址压入栈中。

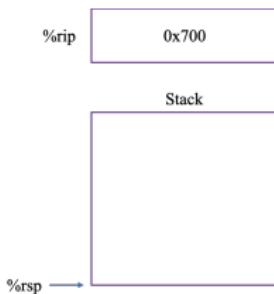


# 转移控制

返回地址就是函数 multstore 调用执行完毕后，下一条指令的地址。

```
000000000000006da<main>:  
...  
6fb: e8 41 00 00 00  callq    741 <multstore>  
→ 700: 48 8b 14 24     mov      (%rsp), %rdx
```

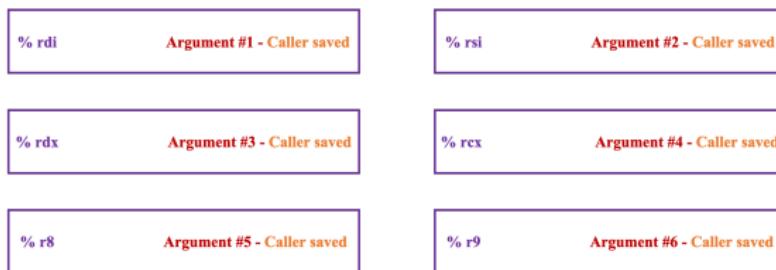
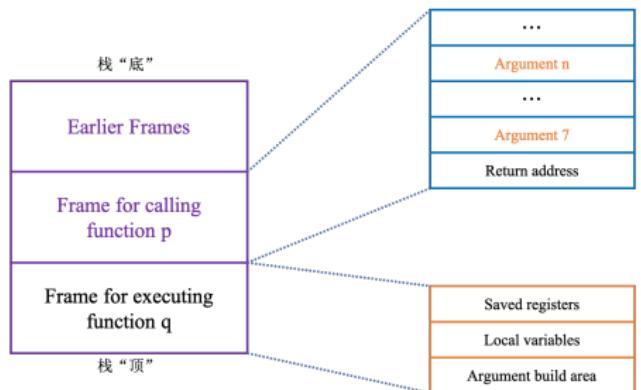
当函数 multstore 执行完毕，指令 ret 从栈中将返回地址弹出，写入到程序指令寄存器 rip 中。



函数返回，继续执行 main 函数中相关的操作。以上整个过程就是函数调用与返回所涉及的操作。

# 参数传递

- 下面我们讨论参数传递，如果一个函数的参数数量大于 6，超出的部分就要通过栈来传递。假设函数 P 有 n 个整型参数，当 n 的值大于 6 时，参数 7 及其后的参数需要用到栈来传递。参数 1 及参数 6 的传递可以使用对应的寄存器。



## 参数传递 demo

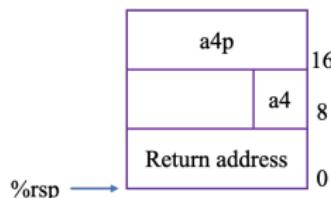
```
void proc(long a1, long *a1p,  
         int a2, long *a2p,  
         short a3, long *a3p,  
         char a4, long *a4p){  
    a1p += a1;  
    a2p += a2;  
    a3p += a3;  
    a4p += a4;  
}
```

- 代码中函数有 8 个参数，包括字节数不同的整数以及不同类型的指针，参数 1 到参数 6 是通过寄存器来传递，参数 7 和参数 8 是通过栈来传递。

a1 → %rdi	a1p → %rsi
a2 → %edx	a2p → %rcx
a3 → %r8w	a3p → %r9
a4 → %rsp + 8	
a4p → %rsp + 16	

# 两点需要注意

- ① 通过栈来传递参数时，所有数据的大小都是向 8 的倍数对齐，虽然变量 a4 只占一个字节，但是仍然为其分配了 8 个字节的存储空间。由于返回地址占用了栈顶的位置，所以这两个参数距离栈顶指针的距离分别为 8 和 16。



- ② 使用寄存器进行参数传递时，寄存器的使用是有特殊顺序规定的，此外，寄存器名字的使用取决于传递参数的大小。如果第一个参数大小是 4 字节，需要用寄存器 edi 来保存。

Operand size(bits)	Argument number					
	1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

# 栈上的局部存储

- 当代码中对一个局部变量使用地址运算符时，我们需要在栈上为这个局部变量开辟相应的存储空间，接下来我们看一个与地址运算符相关的例子。
- 函数 caller 定义了两个局部变量 arg1 和 arg2，函数 swap 的功能是交换这两个变量的值，最后返回二者之和。

## long caller

```
long caller(){  
    long arg1 = 534;  
    long arg2 = 1057;  
    long sum = swap(&arg1, &arg2);  
    long diff = arg1 - arg2;  
    return sum * diff;  
}  
  
long swap(long *xp, long *yp){  
    long x = *xp;  
    long y = *yp;  
    *xp = y;  
    *yp = x;  
    return x + y;  
}
```

## caller:

```
subq $16, %rsp  
movq $534, (%rsp)  
movq $1057, 8(%rsp)  
leaq 8(%rsp), %rsi  
movq %rsp, %rdi  
call swap  
movq (%rsp), %rdx  
subq 8(%rsp), %rdx  
imulq %rdx, %rax  
addq $16, %rsp  
ret
```



## 栈上的局部存储

- 第一条减法指令将栈顶指针减去 16，它表示的含义是在栈上分配 16 个字节的空间。



图：栈顶指针加上 16

图：栈上分配 16 个字节

- 根据接着的两条 mov 指令，可以推断出变量 arg1 和 arg2 存储在函数 caller 的栈帧上，接下来，分别计算变量 arg1 和 arg2 存储的地址，参数准备完毕，执行 call 指令调用 swap 函数。最后函数 caller 返回之前，通过栈顶指针加上 16 的操作来释放栈帧。

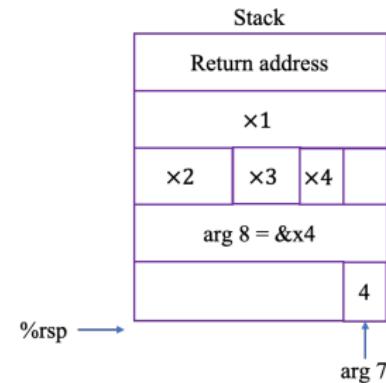


## 栈上的局部存储

我们再看一个稍微复杂的例子：

call proc

```
long call_proc(){
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1 + x2) * (x3 - x4);
}
```



- 函数的栈帧如图。根据变量的类型可知  $x_1$  占 8 个字节,  $x_2$  占 4 个字节,  $x_3$  占两个字节,  $x_4$  占一个字节。由于函数 proc 需要 8 个参数, 因此参数 7 和参数 8 需要通过栈帧来传递。注意, 传递的参数需要 8 个字节对齐, 而局部变量是不需要对齐的。



# 寄存器中的局部存储空间

对于 16 个通用寄存器，除了寄存器 `rsp` 之外，其他 15 个寄存器分别被定义为调用者保存和被调用者保存

## Caller-saved Register (调用者保存寄存器)

%rdi    %rsi    %rdx  
%rcx    %r8    %r9  
%rax    %r10    %r11

## Callee-saved Register (被调用者保存寄存器)

%rbx    %rbp    %r12  
%r13    %r14    %r15

当函数运行需要局部存储空间时，栈提供了内存分配与回收的机制。在程序执行的过程中，寄存器是被所有函数共享的一种资源，为了避免寄存器的使用过程中出现数据覆盖的问题，处理器规定了寄存器使用的惯例，所有的函数调用都必须遵守这个惯例。

## 寄存器中的局部存储空间

P:

x in %rdi, y in %rsi

```
void P(long x, long y){  
    long u = Q(y);  
    long v = Q(x);  
    return u + v;  
}
```

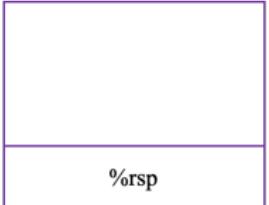
pushq	%rbp
pushq	%rbx
subq	\$8, %rsp
movq	%rdi, %rbp → Save x
movq	%rsi, %rdi
call	Q
...	
popq	%rbx
popq	%rbp
ret	

- ① 由于函数 Q 需要使用寄存器 rdi 来传递参数，因此，函数 P 需要保存寄存器 rdi 中的参数 x；保存参数 x 使用了寄存器 rbp，根据寄存器使用规则，寄存器 rbp 被定义为被调用者保存寄存器，所以便有了开头的这条指令 pushq %rbp，至于 pushq %rbx 也是类似的道理。
  - ② 在函数 P 返回之前，使用 pop 指令恢复寄存器 rbp 和 rbx 的值。由于栈的规则是后进先出，所以弹出的顺序与压入的顺序相反。



```
long rfact(long n){  
    long result;  
    if(n <= 1){  
        result = 1;  
    } else {  
        result = n* rfact( n - 1 );  
    }  
    return result ;  
}
```

rfact:

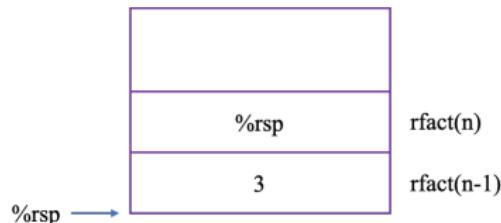
<code>pushq %rbx</code>	<code>pushq %rbx → Save %rbx</code>
<code>movq %rdi, %rbx</code>	
<code>movl \$1, %eax</code>	
<code>cmpq \$1, %rdi</code>	
<code>jle .L35</code>	<code>%rsp →</code>
<code>leaq -1(%rdi), %rdi</code>	
<code>call rfact</code>	<code>%rsp</code>
<code>imulq %rbx, %rax</code>	
<code>.L35</code>	<code>rfact(n)</code>
<code>popq %rbx</code>	
<code>ret</code>	

- ① 这段代码是关于 N 的阶乘的递归实现，我们假设 n=3 时，看一些汇编代码的执行情况。由于使用寄存器 rbx 来保存 n 的值，根据寄存器使用惯例，首先保存寄存器 rbx 的值。

- ② 由于 n=3，所以跳转指令 jle 不会跳转到 L35 处执行。

- ③ 指令 leaq 是用来计算 n-1，然后再次调用该函数。

- 注意，此时寄存器 rbx 内保存的值是 3，指令 pushq 执行完毕后，栈的状态如图所示。



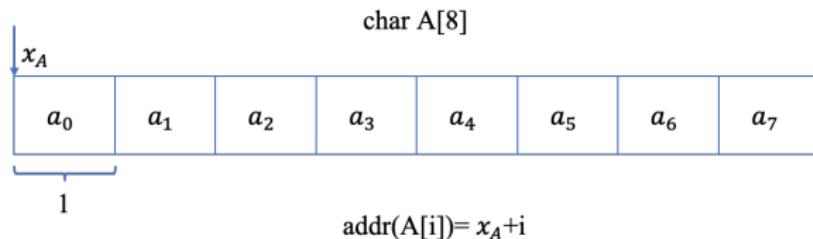
- 继续执行，直到  $n=1$  时，程序跳转到 L35 处，执行 pop 操作。

## 递归总结

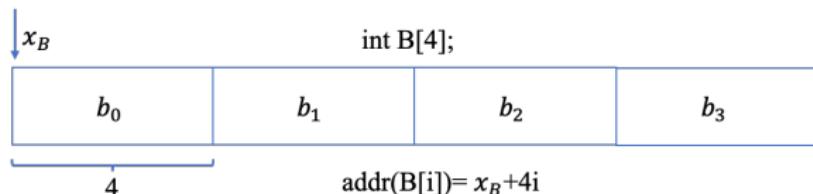
可以看出，递归调用一个函数本身与调用其他函数是一样的，每次函数调用都有它自己私有的状态信息。栈分配与释放的规则与函数调用返回的顺序也是匹配的，当 N 的值非常大时，并不建议使用递归调用。

# 数组分配和访问

- ① 数组 A 是由 8 个 char 类型的元素组成，每个元素的大小是一个字节。假设数组 A 的起始地址是  $X_A$ ，那么数组元素  $A[i]$  的地址就是  $X_A + i$ 。

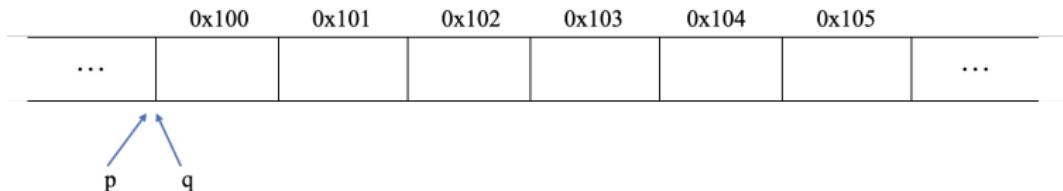


- ② 再来看一个 int 类型的数组：数组 B 是由 4 个整数组成，每个元素占 4 个字节，因此数组 B 总的大小为 16 个字节。假设数组 B 的起始地址是  $X_B$ ，那么数组元素  $B[i]$  的地址就是  $X_B + 4i$ 。

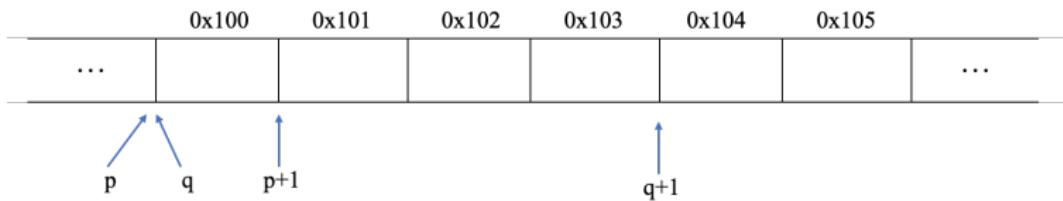


## 指针运算

- C 语言允许对指针进行运算。例如，我们声明一个指向 char 类型的指针 p，和一个指向 int 类型的指针 q。为了方便理解，我们还是把内存抽象成一个大的数组。假设指针 p 和指针 q 都指向 0x100（内存地址）处。



- 现在分别对指针 p 和指针 q 进行加一的操作，指针 p 加 1 指向 0x101 处，而指针 q 加 1 后指向 0x104 处。



- 虽然都是对指针进行加一的运算，但是得到的结果却不同。这是因为对指针进行运算时，计算结果会根据该指针引用的数据类型进行相应的伸缩。

# 指针运算

- ① 接下来，我们定义一个数组 E：假设这个数组存放在内存中，对于数组的每一个元素都有两个属性：一个属性是它存储的内容，另外一个属性是它的存储地址。元素的存储地址可以通过取地址运算符来获得。

int E[6];							
$E[i] \left\{ \begin{array}{l} e_i \\ address \ E[i] \end{array} \right.$							
	0x100	0x104	0x108	0x10C	0x110	0x114	
...	$e_0$	$e_1$	$e_2$	$e_3$	$e_4$	$e_5$	...

- ② 通常我们习惯使用数组引用的方式来访问数组中的元素。

	0x100	0x104	0x108	0x10C	0x110	0x114	
...	$e_0$	$e_1$	$e_2$	$e_3$	$e_4$	$e_5$	...

E[2]

- ③ 除此之外，还有如图另一种方式<sup>2</sup>。表达式  $E + 2$  表示数组第二个元素的存储地址，大写字母 E 表示数组的起始地址（第 0 个元素），此处加 2 的操作与指针加 2 的运算类似，也是与数据类型相关<sup>3</sup>。

	0x100	0x104	0x108	0x10C	0x110	0x114	
...	$e_0$	$e_1$	$e_2$	$e_3$	$e_4$	$e_5$	...

E  
 $(E+2)$   
E[2] \* (E + 2)

<sup>2</sup>指针运算符 \* 可以理解成从该地址处取数据。

<sup>3</sup>理解了内存地址的概念后，可以发现指针其实就是地址的抽象表述。

# 嵌套的数组

- 嵌套数组也被称为二维数组，图中我们声明了一个数组 A，数组 A 可以被看成 5 行 3 列的二维数组，这种理解方式与矩阵的排列类似。在计算机系统中，我们通常把内存抽象为一个巨大的数组，对于二维数组在内存中是按照“行优先”的顺序进行存储的，基于这个规则，我们可以画出数组 A 在内存中的存储情况。

	A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]		A[4][0]	A[4][1]	A[4][2]	
...	$a_{00}$	$a_{01}$	$a_{02}$	$a_{10}$	$a_{11}$	$a_{12}$	...	$a_{40}$	$a_{41}$	$a_{42}$	...

- 关于数组的理解，还有一种方式，就是可以把数组 A 看成一个有 5 个元素的数组，其中每个元素都是一个长度为 3 的数组，这便是嵌套数组的理解方式。

	A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]		A[4][0]	A[4][1]	A[4][2]	
...	$a_{00}$	$a_{01}$	$a_{02}$	$a_{10}$	$a_{11}$	$a_{12}$	...	$a_{40}$	$a_{41}$	$a_{42}$	...

$\underbrace{\hspace{12em}}$  A[0]       $\underbrace{\hspace{12em}}$  A[1]       $\underbrace{\hspace{12em}}$  A[4]

# 嵌套的数组

- 无论用何种方式来理解，数组元素在内存中的存储位置都是一样的。下面我们来看一下数组元素的地址是如何计算的，对于数组 D 任意一个元素可以通过图中的计算公式来计算地址。

$$\begin{aligned} & T \text{ } D[R][C]; \\ & \&D[i][j] = x_D + L ( C \cdot i + j ) \end{aligned}$$

- 其中， $x_D$  表示数组的起始地址； $L$  表示数据类型  $T$  的大小，如果  $T$  是 int 类型， $L$  就等于 4， $T$  是 char 类型， $L$  就等于 1；在具体的示例中， $C$ 、 $i$ 、 $j$  都是常数。
- 根据图中的计算公式，对于  $5 \times 3$  的数组 A，其任意元素的地址可以  $x_A + 4 * (3i + j)$  来计算。

$$\begin{aligned} & \text{int } A[5][3]; \\ & \&A[i][j] = x_A + 4 ( 3 \cdot i + j ) \end{aligned}$$

- 假设数组起始地址  $x_A$  在寄存器 rdi 中，索引值  $i$  和  $j$  分别在寄存器 rsi 和 rdx 中，我们可以用图中的汇编代码将  $A[i][i]$  的值复制到寄存器 eax 中，具体如图所示。

$x_A$  in %rdi,  $i$  in %rsi,  $j$  in %rdx

leaq	(%rsi, %rsi, 2)	%rax	compute $3*i$
leaq	(%rdi, %rax, 4)	%rax	compute $x_A + 12 * i$
movl	(%rax, %rdx, 4)	%eax	Read from M[ $x_A + 12i + 4j$ ]



接下来，我们看一下编译器对定长多维数组的优化。首先使用以下方式将数据类型 fix\_matrix 声明为 16\*16 的整型数组。

- #define N 16
- typedef int fix\_matrix[N][N];

通过 define 声明将 N 与常数 16 关联到一起，之后的代码中就可以使用 N 来代替常数 16，当需要修改数组的长度时，只需要简单的修改 define 声明即可。

## 定长数组

```
#define N 16
typedef int fix_matrix[N][N]
int matrix(fix_matrix A, fix_matrix B,
           long j;
           int result = 0;
           for(j = 0; j < N; j++){
               result += A[i][j] * B[j][k];
           }
           return result;
}
```

- 接下来，我们看一下如何使用汇编代码访问数组元素。由于编译器对相关的操作进行了优化，因此，这段汇编代码有些晦涩难懂。

matrix:

```
salq    $6, %rdx  
addq    %rdx, %rdi  
leaq    (%rsi, %rcx, 4), %rcx  
leaq    1024(%rcx), %rsi  
movl    $0, %eax
```

.L7:

...

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \\ a_{i1} & a_{i2} & \cdots & a_{in} \\ \vdots & & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1k} \cdots b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2k} \cdots b_{2n} \\ \vdots & & & \\ b_{n1} & b_{n2} & \cdots & b_{nk} \cdots b_{nn} \end{bmatrix}$$

Bptr



太原理工大学  
TAIYUAN UNIVERSITY OF TECHNOLOGY

- 在进行循环操作之前，前四行代码是用来计算三个数组元素的地址，一个是数组 A 第 i 行首个元素的地址，另外两个分别是数组 B 第 k 列的第一个元素和最后一个元素的地址，然后将这三个地址分别存放到不同的寄存器中，具体如图所示。

为了方便表述，这里我们引入三个指针来记录这三个地址，接下来，我们介绍一下循环的实现。

```
.L7:  
    movl  (%rdi), %edx  
    imull %rcx, %edx  
    addl  %edx, %eax  
    addq  $4, %rdi  
    addq  $64, %rcx  
    cmpq  (%rsi), %rcx  
    jne .L7  
    rep; ret
```

- ① 首先读取指针 Aptr 指向元素的数据，然后将指针 Aptr 指向的元素与指针 Bptr 指向的元素相乘，最后将乘积结果进行累加，结果保存到寄存器 eax 中。
- ② 计算完成之后，分别移动指针 Aptr 和 Bptr 指向下一个元素，由于 int 类型占 4 个字节，对寄存器 rdi 加 4 的这个操作，对应于移动指针 Aptr 指向数组 A 的下一个元素。由于数组 B 一行元素的数量为 16 个，每个元素占 4 个字节，因此相邻列元素的地址相差为 64 个字节，对寄存器 rcx 进行加 64 的操作对应移动指针 Bptr 指向数组 B 的下一个元素。

判断循环结束的条件是：指针 Bptr 指针与指针 Bend 是否指向同一个内存地址，如果二者不相等，继续跳转到 L7 处执行，如果二者相等，循环结束。

$$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1k} \cdots b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2k} \cdots b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nk} \cdots b_{nn} \end{bmatrix}$$

64

- 通过这段汇编代码，我们可以发现，编译器使用了很巧妙的方式来计算数组元素的地址，这些优化方法显著的提升了程序的执行效率。
- 在 C89 的标准中，程序员在使用变长数组时需要使用 malloc 这类函数，为数组动态分配存储空间。在 ISO C99 的标准中，引入了变长数组的概念，因此，我们可以通过下列代码的方式来声明一个变长数组。

## 变长数组

```
int A[expr1][expr2];
long var_ele(long n, int A[n][n], long i, long k){
    return A[i][j];
}
```



- 它可以作为一个局部变量，也可以作为函数的参数，当变长数组作为函数参数时，参数 n 必须在数组 A 之前。
- 变长数组元素的地址计算与定长数组类似，不同点在于新增了参数 n，需要使用乘法指令来计算 n 乘以 i。

var\_ele:

imulq	%rdx, %rdi	compute $n * i$
leaq	(%rsi, %rdi, 4), %rax	compute $x_A + 4(n * i)$
movl	(%rax, %rcx, 4), %eax	Read from M[ $x_A + 4(n * i) + 4j$ ]
ret		

还是矩阵 A 和矩阵 B 内积的例子，如果采用变长数组来存储矩阵 A 和矩阵 B，与定长数组相比 C 代码的实现几乎没有差别。

# 结构体

vra\_mat

```
int var_mat(long n, int A[n][n], int B[n][n], long i, long k){  
    long j;  
    int result = 0;  
    for (j=0; j < N; j++){  
        result += A[i][j] * B[j][k];  
    }  
    return result;  
}
```

不过对比二者的汇编代码，可以发现编译器采用了不同的优化方法。

.L7:(fixed)

```
    movl  (%rdi), %edx  
    imull (%rcx), %edx  
    addl  %edx, %eax  
    addq  $4, %rdi  
    addq  $64, %rcx  
    cmpq  (%rsi), %rcx  
    jne .L7  
    rep; ret
```

.L24:(variable)

```
    movl  (%rsi, %rdx, 4), %r8x  
    imull (%rcx), %r8d  
    addl  %r8d, %eax  
    addq  $1, %rdx  
    addq  %r9, %rcx  
    cmpq  %rdi, %rdx  
    jne .L24
```



struct

```
struct rec{  
    int i;  
    int j;  
    int a[2];  
    int *p;  
}
```

假设 r 存放在寄存器 rdi 中，可以使用下图的汇编指令将字段 i 的值复制到字段 j 中。

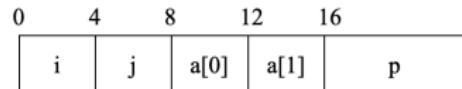
r in %rdi

```
movl (%rdi), %eax Get r->i  
movl %eax, 4(%rdi) Store in r->j
```

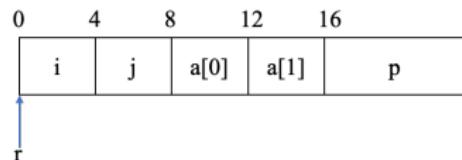
- 首先读取字段 i 的值，由于字段 i 相对于结构体起始地址的偏移量为 0，所以字段 i 的地址就是 r 的值，而字段 j 的偏移量为 4，因此需要将 r 加上偏移量 4。
- 其中结构体指针 r 存放在寄存器 rdi 中，数组元素的索引值 i 存放在寄存器 rsi 中，最后地址的计算结果，存放在寄存器 rax 中。

综上所述，无论是单个变量还是数组元素，都是通过起始地址加偏移量的方式来访问。

- 这个结构体包含四个字段：两个 int 类型的变量，一个 int 类型的数组和一个 int 类型的指针。我们可以画出各个字段相对于结构体起始地址处的字节偏移。



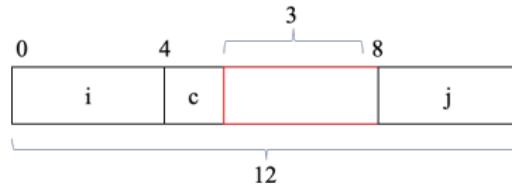
- 可以看出数组 a 的元素是嵌入到结构体中的。接下来，我们看一下如何访问结构体中的字段。
- 例如，我们声明一个结构体类型指针变量 r，它指向结构体的起始地址。



对于图中的结构体，它包含两个 int 类型的变量和一个 char 类型的变量。

```
struct S1{  
    int i;  
    char c;  
    int j;  
}
```

- 根据前面的知识，我们会直观的认为该结构体占用 9 个字节的存储空间，但是当使用 `sizeof` 函数对该结构体的大小进行求值时，得到的结果却是 12 个字节。原因是为了提高内存系统的性能，系统对于数据存储的合法地址做出了一些限制。
  - 例如变量 `j` 是 `int` 类型，占 4 个字节，它的起始地址必须是 4 的倍数，因此，编译器会在变量 `c` 和变量 `j` 之间插入一个 3 字节的间隙，这样变量 `j` 相对于起始地址的偏移量就为 8，整个结构体的大小就变成了 12 个字节。

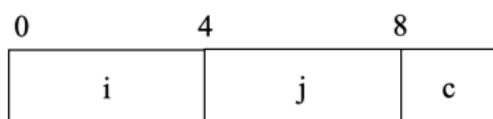


对于不同的数据类型，地址对齐的原则是任何 K 字节的基本对象的地址必须是 K 的倍数。也就是说对于 short 类型，起始地址必须是 2 的倍数；对于占 8 个字节的数据类型，起始地址必须是 8 的倍数。

表: 不同数据类型的长度

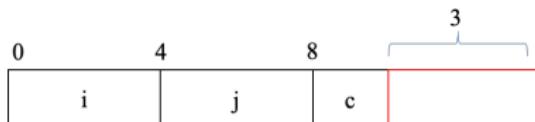
K	Types
1	char
2	short
4	int, float
8	long, double, char*

```
struct S1{  
    int i;  
    char c;  
    int j;  
};  
struct S2{  
    int i;  
    int j;  
    char c;  
};
```



- 基于上表的规则，编译器可能需要在字段的地址空间分配时插入间隙，以此保证每个结构体的元素都满足对齐的要求。
- 除此之外，结构体的末尾可能需要填充间隙，还是刚才的这个结构体，可以通过调整字段 j 和字段 c 的排列顺序，使得所有的字段都满足了数据对齐的要求。

但是当我们声明一个结构体数组时，分配 9 个字节的存储空间，是无法满足所有数组元素的对齐要求，因此，编译器会在结构体的末端增加 3 个字节的填充，这样一来，所有的对齐限制都满足了。



根据上述对齐原则，我们看一个复杂的示例。

```
struct {  
    char *a;  
    short b;  
    double c;  
    char d;  
    float e;  
    char f;  
    long g;  
    int h;  
}rec;
```

- ① 变量 a 是一个指针变量，占 8 个字节。
- ② 变量 b 是 short 类型，占两个字节，它起始地址的字节偏移量是 8，满足对齐规则的 2 的倍数。
- ③ 由于变量 c 是 double 类型，占 8 个字节，因此，该变量起始地址的偏移量需要是 8 的倍数，所以需要在变量 b 之后插入 6 个字节的间隙。
- ④ 对于变量 d 只占一个字节，顺序排列即可。
- ⑤ 由于变量 e 占 4 个字节，它的偏移量需要是 4 的倍数，因此，需要在变量 d 之后插入 3 个字节的间隙。
- ⑥ 同样变量 f 是 char 类型，顺序排列即可。
- ⑦ 由于变量 g 占 8 个字节，因此需要在变量 f 之后插入 7 个字节的间隙。
- ⑧ 最后一个变量 h 占 4 个字节，此时结构体的大小为 52 个字节，为了保证每个元素都满足对齐要求，还需要在结构体的尾端填充 4 个字节的间隙。

# 数据对齐

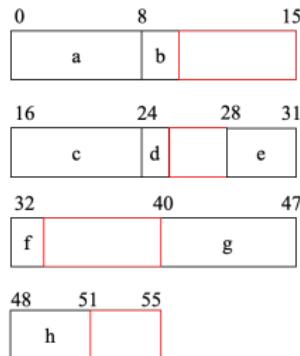


图: 对齐后的结构体

```
struct S3{  
    char   c;  
    int    i[2];  
    double v;  
};
```

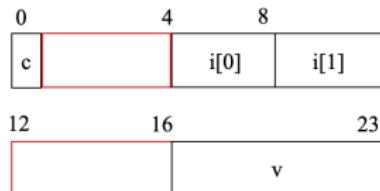
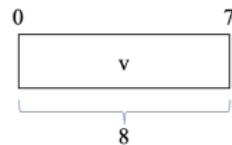


图: 结构体与联合体对比

```
union S3{  
    char   c;  
    int    i[2];  
    double v;  
};
```



- ① 此外，关于更多情况的数据对齐情况，还需要针对不同型号的处理器以及编译系统进行具体分析。
- ② 与结构体不同，联合体中的所有字段共享同一存储区域，因此联合体的大小取决于它最大字段的大小。
- ③ 变量 v 和数组 i 的大小都是 8 个字节，因此，这个联合体的占 8 个字节的存储空间。

- 联合体的一种应用情况是：我们事先知道两个不同字段的使用是互斥的，那么我们可以将这两个字段声明为一个联合体。原理就是不会让不可能有数据的字段白白浪费内存。
- 例如，我们定义一个二叉树的数据结构，这个二叉树分为内部节点和叶子节点，其中每个内部节点不含数据，都有指向两个孩子节点的指针，每个叶子节点都有两个 double 类型的数据值。

我们可以用结构体来定义该二叉树的节点：

## node

```
struct node_s{  
    struct node_s *left;  
    struct node_s *right;  
    double data[2];  
};
```

- 那么每个节点需要 32 个字节，由于该二叉树的特殊性，我们事先知道该二叉树的任意一个节点不是内部节点就是叶子节点，因此，我们可以用联合体来定义节点。

## node

```
union node_u{  
    struct {  
        union node_s *left;  
        union node_s *right;  
    } internal;  
    double data[2];  
};
```

- 这样每个节点只需要 16 个字节的存储空间，相对于结构体的定义方式可以节省一半的空间。不过这种编码方式存在一个问题：就是没有办法确定一个节点到底是叶子节点还是内部节点，通常的解决方法是引入一个枚举类型，然后创建一个结构体，它包含一个标签和一个联合体。



## node

```
typedef enum{N_LEAF, N_INTERNAL} nodetype_t;
struct node_t{
    nodetype_t type;
    union{
        struct{
            union node_s *left;
            union node_s *right;
        }internal;
        double data[2];
    }info;
};
```

- 其中 type 占 4 个字节，联合体占 16 个字节，type 和联合体之间需要加入 4 个间隙，因此，整个结构体的大小为 24 个字节。

- 虽然使用联合体可以节省存储空间，但是相对于给代码编写造成的麻烦，这样的节省意义不大。因此，对于有较多字段的情况，使用联合体带来的空间节省才会更吸引人。
- 除此之外，联合体还可以用来访问不同数据类型的位模式，当我们使用简单的强制类型转换，将 double 类型的数据转换成 unsigned long 类型时，除了 d 等于 0 的情况，二者的二进制位表示差别很大，这时我们可以将这两种类型的变量声明为一个联合体，这样就是可以以一种类型来存储，以另外一种类型来访问，变量 u 和 d 就具有相同的位表示。

## union

```
unsigned long u = (unsigned long) d;
unsigned long double2bits(double d){
    union{
        double d;
        unsigned long u;
    }temp;
    temp.d = d;
    return temp.u;
};
```



echo 函数声明了一个长度为 8 的字符数组。gets 函数是 C 语言标准库中定义的函数，它的功能是从标准输入读入一行字符串，在遇到回车或者某个错误的情况下停止，gets 函数将这个字符串复制到参数 buf 指明的位置，并在字符串结束的位置加上 null 字符。注意 gets 函数会有一个问题，就是它无法确定是否有足够大的空间来保存整个字符串，长一些字符串可能会导致栈上的其他信息被覆盖，通过汇编代码，我们可以发现实际上栈上分配了 24 个字节的存储空间。

## union

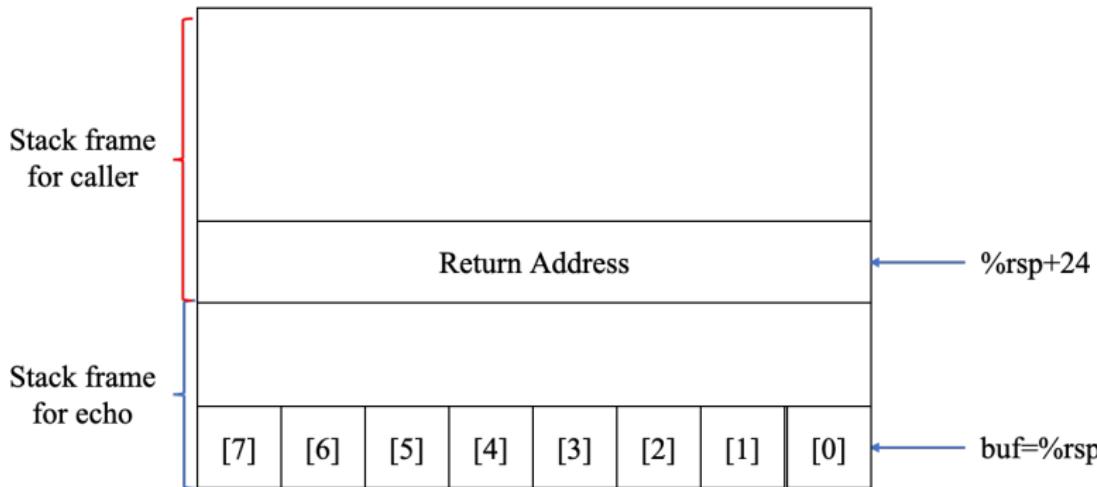
```
void echo(){
    char buf[8];
    gets(buf);
    puts(buf);
}
```

echo:

void echo()	subq	\$24, %rsp
{	movq	%rsp, %rdi
char buf[8];	call	gets
gets(buf);	movq	%rsp, %rdi
puts(buf);	call	puts
}	addq	\$24, %rsp
	ret	



为了方便表述，我们将栈的数据分布画了出来，其中字符数组位于栈顶的位置。



- ① 实际上当输入字符串的长度不超过 23 时，不会发生严重的后果，超过以后，返回地址以及更多的状态信息会被破坏，那么返回指令会导致程序跳转到一个完全意想不到的地方。
  - ② 历史上许多计算机病毒就是利用缓冲区溢出的方式对计算机系统进行攻击的，针对缓冲区溢出的攻击，现在编译器和操作系统实现了很多机制，来限制入侵者通过这种攻击方式来获得系统控制权。
  - ③ 例如栈随机化、栈破坏检测以及限制可执行代码区域等。

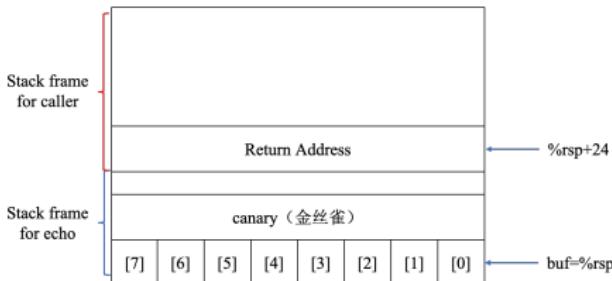


## union

```
int main(){
    long local;
    printf("local at %p\n", &local);
    return 0;
}
```

- 在过去，程序的栈地址非常容易预测，如果一个攻击者可以确定一个 web 服务器所使用的栈空间，那就可以设计一个病毒程序来攻击多台机器，栈随机化的思想是栈的位置在程序每次运行时都有变化，上面这段代码只是简单的打印 main 函数中局部变量 local 的地址，每次运行打印结果都可能不同。

- 在 64 位 linux 系统上，地址的范围：0x7fff0001b698~0x7fffffaa4a8。因此，采用了栈随机化的机制，即使许多机器都运行相同的代码，它们的栈地址也是不同的。
- 在 linux 系统中，栈随机化已经成为标准行为，它属于地址空间布局随机化的一种，简称 ASLR，采用 ASLR，每次运行时程序的不同部分都会被加载到内存的不同区域，这类技术的应用增加了系统的安全性，降低了病毒的传播速度。



## 金丝雀

编译器会在产生的汇编代码中加入一种栈保护者的机制来检测缓冲区越界，就是在缓冲区与栈保存的状态值之间存储一个特殊值，这个特殊值被称作金丝雀值，之所以叫这个名字，是因为从前煤矿工人会根据金丝雀的叫声来判断煤矿中有毒气体的含量。

# 栈破坏检测

- 金丝雀值是每次程序运行时随机产生的，因此攻击者想要知道这个金丝雀值具体是什么并不容易，在函数返回之前，检测金丝雀值是否被修改来判断是否遭受攻击。接下来，我们通过汇编代码看一下编译器是如何避免栈溢出攻击的。

echo:

```
subq    $24,    %rsp  
movq    %fs:40 , %rax  
movq    %rax , 8(%rsp)  
xorl    %eax,   %eax  
movq    %rsp,   %rdi  
call    gets  
movq    %rsp,   %rdi  
call    puts
```

echo:

```
movq    8(%rsp), %rax  
xorq    %fs:40 , %rax  
je     .L9  
call    __ stack_chk_fail  
.L9:  
addq    $24 ,  %rsp  
ret
```

echo:

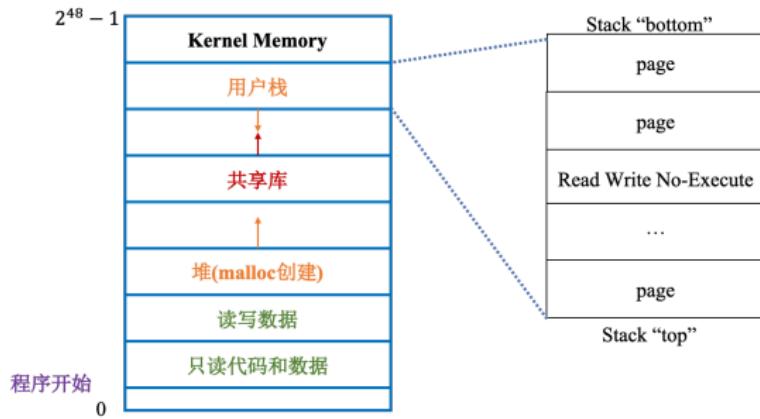
```
subq    $24,    %rsp  
movq    %fs:40 , %rax  
movq    %rax , 8(%rsp)  
xorl    %eax,   %eax  
movq    %rsp,   %rdi  
call    gets  
movq    %rsp,   %rdi  
call    puts
```

```
movq    8(%rsp), %rax  
xorq    %fs:40 , %rax  
je     .L9  
call    __ stack_chk_fail  
.L9:  
addq    $24 ,  %rsp  
ret
```

- 图中这两行代码是从内存中读取一个数值，然后将该数值放到栈上，其中这个数值就是刚才提到的金丝雀值，存放的位置与程序中定义的缓冲区是相邻的。其中指令源操作数%fs:40 可以简单的理解为一个内存地址，这个内存地址属于特殊的段，被操作系统标记为“只读”，因此，攻击者是无法修改金丝雀值的。
- 函数返回之前，我们通过指令 xor 来检查金丝雀值是否被更改。如果金丝雀值被更改，那么程序就会调用一个错误处理例程，如果没有被更改，程序就正常执行。

## 限制可执行代码区域

- 最后一种机制是消除攻击者向系统中插入可执行代码的能力，其中一种方法是限制哪些内存区域能够存放可执行代码。
    - ① 以前，x86 的处理器将可读和可执行的访问控制合并成一位标志，所以可读的内存页也都是可执行的，由于栈上的数据需要被读写，因此栈上的数据也是可执行的。
    - ② 虽然实现了一些机制能够限制一些页可读且不可执行，但是这些机制通常会带来严重的性能损失，后来，处理器的内存保护引入了不可执行位，将读和可执行访问模式分开了。有了这个特性，栈可以被标记为可读和可写，但是不可执行。检查页是否可执行由硬件来完成，效率上没有损失。



- 以上这三种机制，都不需要程序员做任何额外的工作，都是通过编译器和操作系统来实现的，单独每一种机制都能降低漏洞的等级，组合起来使用会更加有效。
  - 不幸的是，仍然有方法能够对计算机进行攻击。

- ① Randal E. Bryant, David R. O'Hallaron, et. al. 著, 龚奕利等译. 深入理解计算机系统, 机械工业出版社, 2016
- ② 唐朔飞, 计算机组成原理, 高等教育出版社, 2008
- ③ 王道论坛, 计算机组装原理考研复习指导, 电子工业出版社, 2021