Datawhale

Datawhale 开源社区

Datawhale open source community

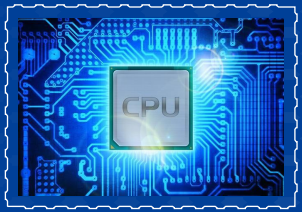
深入理解计算机系统（5）

Computer Systems A Programmer's Perspective CSAPP

李岳昆、易远哲

realyurk@gmail.com、yuanzhe.yi@outlook.com

2021 年 9 月 25 日



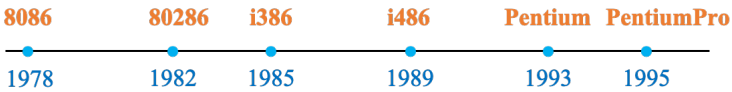
第 I 部分

程序的机器级表示-I

程序编码　　访问信息

我们先来简单的了解一下 Intel 处理器的发展历史。

1978 年 Intel 发布了第一款微处理器-8086，在接下来的四十多年里，Intel 不断地推出新的处理器，从最早的 16 位扩展到 32 位，近些年又扩展到 64 位。



演示程序

Datawhale

main.c

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

mstore.c

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

演示程序

Datawhale

示例包含两个源文件，一个是 main.c，另外一个是 mstore.c。通过以下命令进行编译：

- 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

以源文件 mstore.c 为例，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

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

生成汇编文件

Datawhale

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

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

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

生成汇编文件

Datawhale

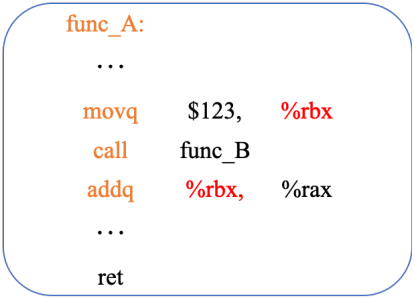
在 Intel x86-64 的处理器中包含了 16 个通用目的的寄存器，这些寄存器用来存放整数数据和指针。

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

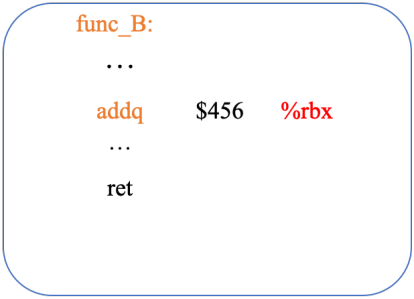
图中显示的这 16 个寄存器，它们的名字都是以%r 开头的，在详细介绍寄存器的功能之前，我们首先需要搞清楚两个概念：**调用者保存寄存器**和**被调用者保存寄存器。**

两种保存器

函数 A 中调用了函数 B，因此，函数 A 称为调用者，函数 B 称为被调用者。



Caller



Callee

两种保存器

Datawhale

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
```

对于具体使用哪一种策略，不同的寄存器被定义成不同的策略，具体如图所示

Callee saved:	%rbx,	%rbp,	%r12,	%r13,	%r14,	%15
Caller saved:	%r10,	%r11				
	%rax					
	%rdi,	%rsi,	%rdx,	%rcx,	%r8,	%r9

汇编拆解

Datawhale

- 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 mulstore(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
```

汇编拆解

Datawhale

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。

```
long mult2(long,long);

        //x->%rdi y->%rsi dest->%rdx
void mulstore(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
```

渊源与缩写

Datawhale

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。

汇编详解

- 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 mulstore(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

Datawhale

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
- 执行这条命合，即可生产 mstore.c 对应的机器代码文件 mstore.o。由于该文件是二进制格式的，所以无法直接查看。这里我们需要借助一个反汇编工具一objdump。汇编器将汇编代码翻译成二进制的机器代码，那么反汇编器就是机器代码翻译成汇编代码。

通过以下命合，我们可以查看 mstore.o 中的相关信息。

- linux> objdump -d mstore.o

```
0000000000000000<multstore>:
   0：53                    push   %rbx
   1：48 89 d3              mov    %rdx, %rbx
   4：e8 00 00 00 00        callq  9 <multstore+0x9>
   9：48 89 03              mov    %rax, (%rbx)
   c：5b                    pop    %rbx
   d：c3                    retq
```

生成机器代码文件

Datawhale

通过对比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。

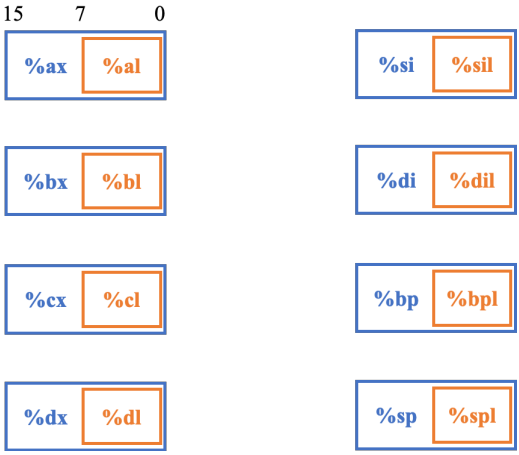
0000000000000000<multstore>:			multstore:	
0：53	push	%rbx	pushq	%rbx
1：48 89 d3	mov	%rdx, %rbx	movq	%rdx, %rbx
4：e8 00 00 00 00	callq	9 <multstore+0x9>	call	mult2
9：48 89 03	mov	%rax, (%rbx)	movq	%rax，(%rbx)
c：5b	pop	%rbx	popq	%rbx
d：c3	retq		ret	

反汇编代码省略了很多指令的后缀的“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

Datawhale

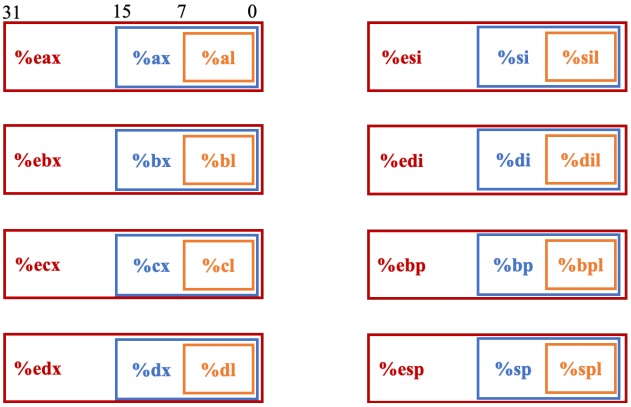
- 最早 8086 的处理器中，包含 8 个 16 位的通用寄存器。



寄存器发展史

Datawhale

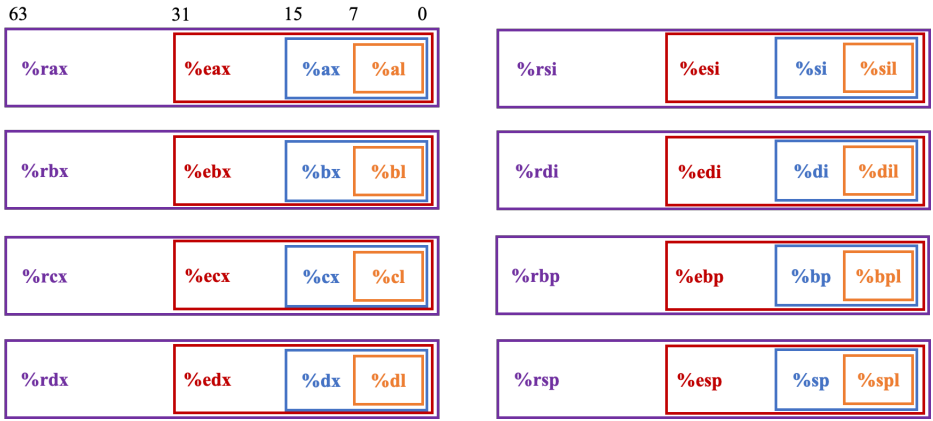
- 每个寄存器都有特殊的功能，它们的名字就反映了不同的用途，当处理器从 16 位扩展到 32 位时，寄存器的位数也随之扩展到了 32 位。



寄存器发展史

Datawhale

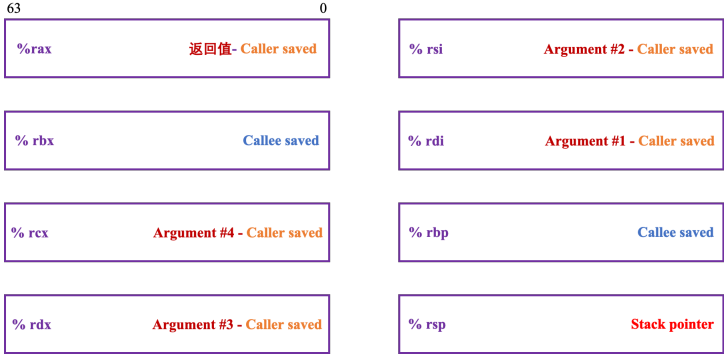
- 直到今天，原来 8 个 16 位寄存器已经扩展成了 64 位。此外还增加了 8 个新的寄存器。



寄存器现状

Datawhale

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。



寄存器现状

Datawhale

63 0

% r8	Argument #5 - Caller saved
% r9	Argument #6 - Caller saved
% r10	Caller saved
% r11	Caller saved
% r12	Callee saved
% r13	Callee saved
% r14	Callee saved
% r15	Callee saved

指令相关

Datawhale

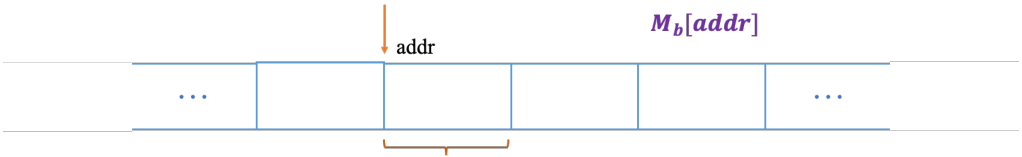
大多数指令包含两部分：操作码和操作数。例如图中的这几条指令 movq、addq、subq 这部分被定义为操作码，它决定了 CPU 执行操作的类型；操作码之后的这部分是操作数，大多数指令具有一个或者多个操作数。不过像 ret 返回指令，是没有操作数的。

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

操作数指示符

Datawhale

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。



操作数指示符

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。

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

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。

操作数指示符

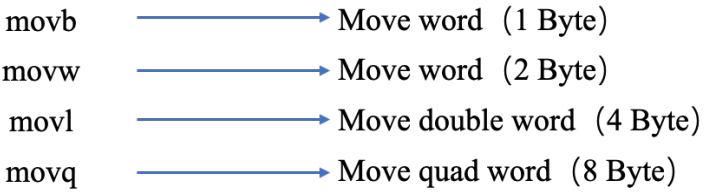
Datawhale

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	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i \cdot s]]$	Scaled indexed
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base+displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$(.r_i, s)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(, r_i, s)$	$M[R[r_i] \cdot s]$	Scale indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scale indexed
Memory	$Imm(, r_i, s)$	$M[R[r_b] + R[r_i] \cdot s]$	Scale indexed

mov 指令

- 对于 mov 类指令，含有两个操作数，个称为源操作数，另外一个称为目的操作数。
- 对于源操作数，可以是一个立即数、一个寄存器，或者是内存引用。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，注意目的操作数不能是一个立即数。



mov 指令

Datawhale

- 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。

MOV	源操作数	目的操作数
	立即数 寄存器 内存	寄存器 内存

mov 指令

Datawhale

下图中的指令给出了不同类型的源操作数和目的操作数的组合，第一个是源操作数，第二个是目的操作数。

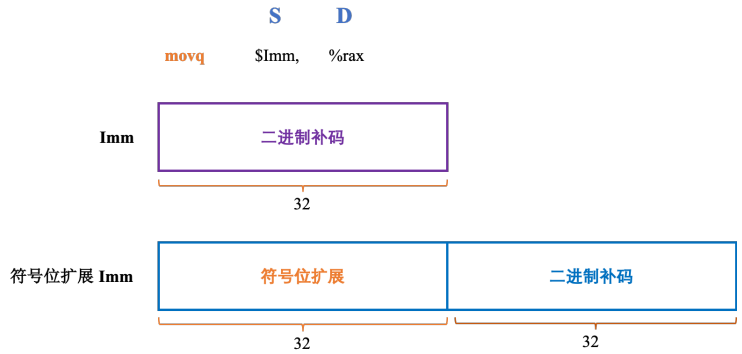
Memory	Memory：
	mov memory, register
	mov register, memory

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

Datawhale

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?

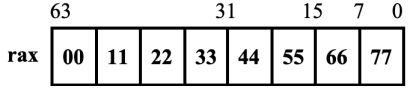
这里引入一个新的指令 movabsq，该指令的源操作数可以是任意的 64 位立即数，需要注意的是目的操作数只能是寄存器。

	S	D
movq	$Imm	%rax
movabsq	$Imm	Register

接下来，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的。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
```

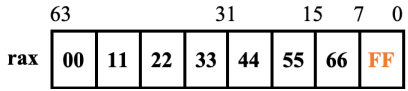
此时，寄存器 rax 内保存的数值如图所示。



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
```

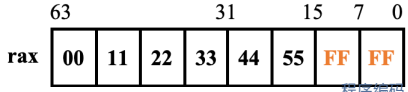
此时寄存器 rax 的低 8 位发生了改变。



第三条指令 movw 是将立即数-1 复制到寄存器 ax。

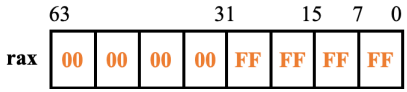
```
movw     $-1    %ax
```

此时寄存器 rax 的低 16 位发生了改变。



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
```



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[1]。

[1]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

Datawhale

- 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。

指令	影响	描述
MOVZ S, R	$R \leftarrow$ZeroExtend(S)	Move with zero extension
movzbw		Move Zero-extended byte to word
movzbl		Move Zero-extended byte to Double word
movzwl		Move Zero-extended word to Double word
movzbq		Move Zero-extended byte to Quad word
movzwq		Move Zero-extended word to Quad word

符号位扩展传送指令

Datawhale

- 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。

指令	影响	描述
MOVZ S, R	$R \leftarrow$SignExtend(S)	Move with sign extension
movsbw		Move Sign-extended byte to word
movsbl		Move Sign-extended byte to Double word
movswl		Move Sign-extended word to Double word
movsbq		Move Sign-extended byte to Quad word
movswq		Move Sign-extended word to Quad word
movslq		Move Sign-extended Double word to quad word

一些补充

Datawhale

- 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。

最后，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 rax。

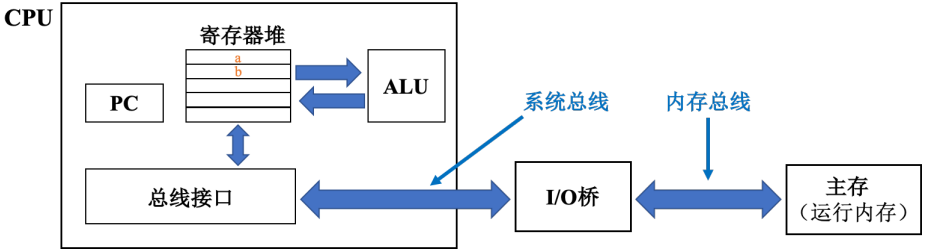
cltq 指令效果与图中这条指令的效果一致，只不过编码更紧凑一些。

```
cltq        movslq    %eax,    %rax
```

数据传送指令

Datawhale

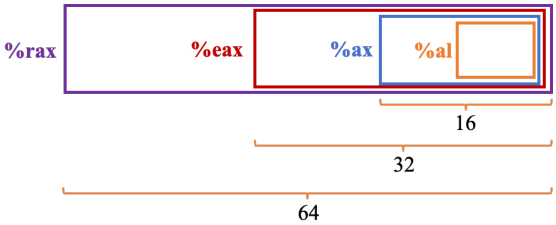
实际上，在一些程序的执行过程中，需要在 CPU 和内存之间进行频繁的数据存取。例如 CPU 执行一个简单的加法操作 c=a +b。那么首先通过 CPU 执行数据传送指令将 a 和 b 的值从内存读到寄存器内，寄存器就是 CPU 内的一种数据存储部件，只不过是容量比较小。



数据传送指令

Datawhale

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；



数据传送指令

- 对于寄存器 rax，如果使用全部的 64 位，用符号%rax 来表示；如果是只用到低 32 位，可以用符号%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。
- 正是因为数据传送在计算机系统中是一个非常频繁的操作，所以了解一下数据传输指令对理解计算机系统会有很大的帮助。

数据传送示例

Datawhale

代码示例

```
1  int main(){
2      long a = 4;
3      long b = exchange(&a, 3);
4      printf("a = %ld, b = %ld\n", a, b);
5      return 0;
6  }
7  long exchange(long *xp, long y){
8      long x = *xp;
9      *xp = y;
10     return x;
11 }
```

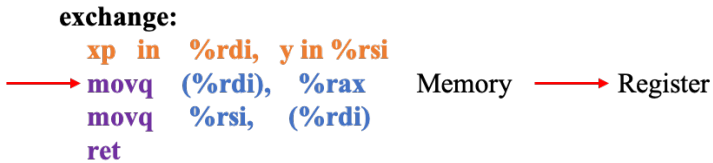
变量 a 的值会替换成 3，变量 b 将保存变量 a 原来的值 4。重点看函数 exchange 所对应的汇编指令：

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。根据寄存器的使用惯例，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。这段汇编代码中并没有显式的将这部分表示出来，需要注意一下。

```
long exchange(long *xp, long y){
    long x = *xp;
    *xp = y;
    return x;
}
```

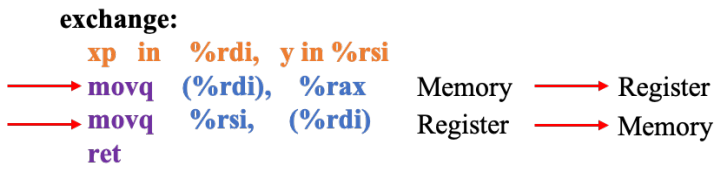
```
exchange:
    xp  in   %rdi,  y in %rsi
    movq   (%rdi),   %rax
    movq    %rsi,    (%rdi)
    ret
```

- 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。



Datawhale

- 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;

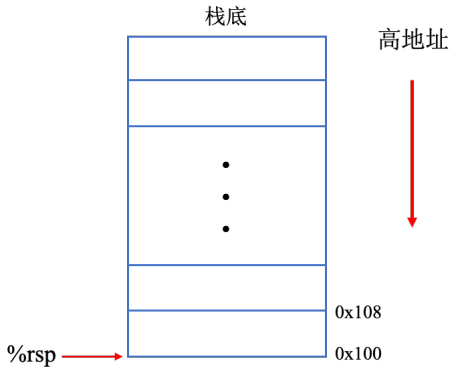


通过这个例子，我们可以看到 C 语言中所谓的指针其实就是地址。

此外，还有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。栈的增长方向是从高地址向低地址，因此，栈顶的元素是所有栈中元素地址中最低的。根据惯例，栈是倒过来画的，栈顶在图的底部，栈底在顶部。

例如我们们需要保存寄存器 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：

❶ 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。



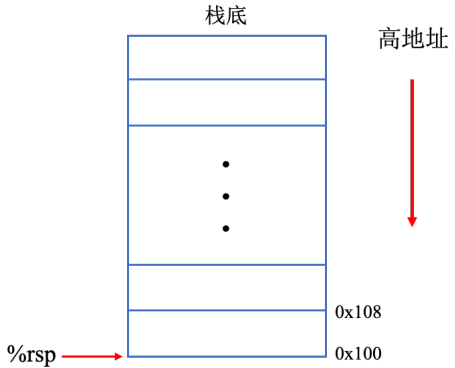
❷ 然后将需要保存的数据复制到新的栈顶地址，此时，内存地址 0x100 处将保存寄存器 rax 内存储的数据 0x123。实际上 pushq 的指令等效于图中 subq 和 movq 这两条指令。它们之间的区别是在于 pushq 这一条指令只需要一个字节，而 subq 和 movq 这两条指令需要 8 个字节。

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

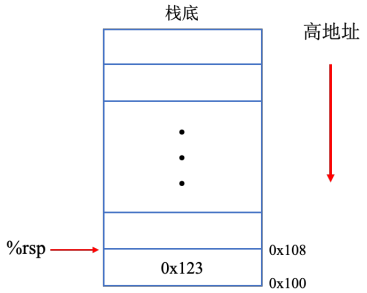
说到底，push 指令的本质还是将数据写入到内存中，那么与之对应的 pop 指令就是从内存中读取数据，并且修改栈顶指针。例如图中这条 popq 指令就是将栈顶保存的数据复制到寄存器 rbx 中。

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。



❷ 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.