



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

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

2021 年 12 月 14 日



## 第 I 部分

## 程序的机器级表示-I

## 程序编码

## 访问信息

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。



```
1 #include<stdio.h>
2 void multstore(long,long,long *);
3 int main() {
4     long d;multstore(2, 3, &d) ;
5     printf("2* 3 —>%ld \n", d) ;return 0;
6 }
```

```
1 long mult2(long a, long b){
2     png s = a* b;return s;
3 }
```

示例包含两个源文件，一个是 `main.c`，另外一个为 `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。

- `linux> gcc -Og -S mstore.c`

- `linux> gcc -Og -S mstore.c`

multistore:

**.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，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?

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**。



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。

func\_A:

• • •

```
movq    $123,    %rbx
```

call func B

```
addq    %rbx, %rax
```

• • •

ret

## Caller

func\_B:

• • •

```
addq    $456    %rbx
```

...

ret

## Callee

- 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。

恢复寄存器rbx原来存储的内容

恢复寄存器rbx原来存储的内容

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

<b>Callee saved:</b>	%rbx,	%rbp,	%r12,	%r13,	%r14,	%r15
<b>Caller saved:</b>	%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·mulstore(long·x, long·y, long·*dest){  
    ... long·t = mult2(x, y);  
    ... *dest = t;  
}
```

## multistore:

```
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。

7

 $\} \neg$ 

ret

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。

- ```
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;  
}  
·
```

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

- `linux> gcc -Og -c mstore.c`
- 执行这条命令，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—`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。

- `linux> objdump -d mstore.o`

|                    |       |                   |
|--------------------|-------|-------------------|
| 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  | 程序编码              |



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。

000000000000000000&lt;multstore&gt;:

```
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
```

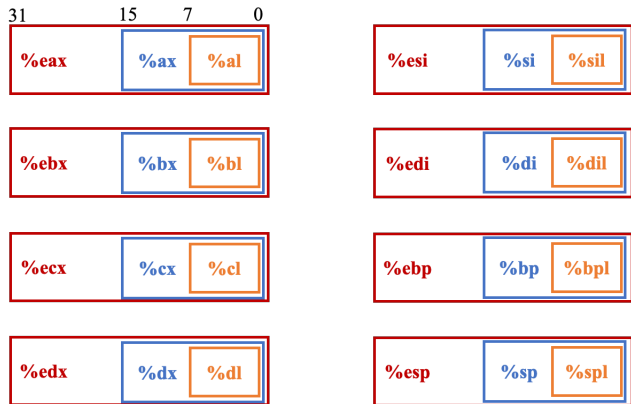
multistore:

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

- | 15                                         | 7 | 0 |                                             |  |  |
|--------------------------------------------|---|---|---------------------------------------------|--|--|
| <div> <div>%ax</div> <div>%al</div> </div> |   |   | <div> <div>%si</div> <div>%sil</div> </div> |  |  |
| <div> <div>%bx</div> <div>%bl</div> </div> |   |   | <div> <div>%di</div> <div>%dil</div> </div> |  |  |
| <div> <div>%cx</div> <div>%cl</div> </div> |   |   | <div> <div>%bp</div> <div>%bpl</div> </div> |  |  |
| <div> <div>%dx</div> <div>%dl</div> </div> |   |   | <div> <div>%sp</div> <div>%spl</div> </div> |  |  |

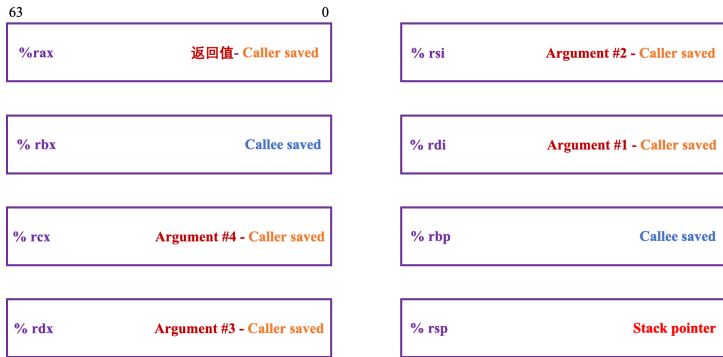
- 每个寄存器都有特殊的功能，它们的名字就反映了不同的用途，当处理器从 16 位扩展到 32 位时，寄存器的位数也随之扩展到了 32 位。



- 
- Diagram illustrating the bit fields of x86-64 registers:
- %rax**: 64-bit register. Bit fields: **%eax** (32-bit), **%ax** (16-bit), **%al** (8-bit).
  - %rbx**: 64-bit register. Bit fields: **%ebx** (32-bit), **%bx** (16-bit), **%bl** (8-bit).
  - %rcx**: 64-bit register. Bit fields: **%ecx** (32-bit), **%cx** (16-bit), **%cl** (8-bit).
  - %rdx**: 64-bit register. Bit fields: **%edx** (32-bit), **%dx** (16-bit), **%dl** (8-bit).
- Bit positions 63, 31, 15, 7, and 0 are indicated at the top of the diagram.



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` 用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。



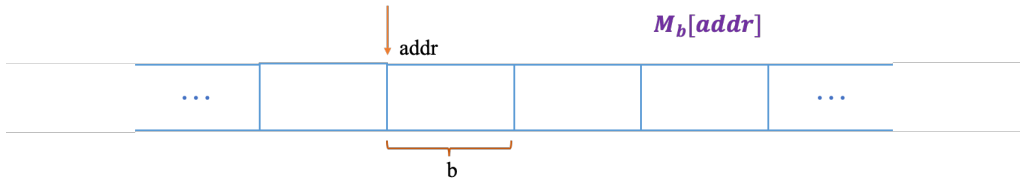
## 寄存器现状



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  |               |

- 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  $b$ 。



$Imm(r_b, r_i, s)$

$Imm$  → 立即数

$r_b$  → 基址寄存器

$r_i$  → 变址寄存器

$s$  → 比例因子



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

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。

其他的形式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     |

- |      |   |                           |
|------|---|---------------------------|
| movb | → | Move word (1 Byte)        |
| movw | → | Move word (2 Byte)        |
| movl | → | Move double word (4 Byte) |
| movq | → | Move quad word (8 Byte)   |

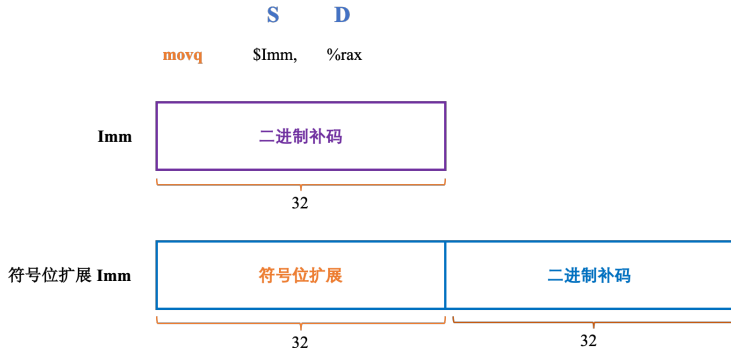
- | MOV | 源操作数 | 目的操作数 |
|-----|------|-------|
|     | 立即数  |       |
|     | 寄存器  | 寄存器   |
|     | 内存   | 内存    |

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。

### Memory :

mov register, memory

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？

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`。

此时，寄存器 `rax` 内保存的数值如图所示。



```
movb,    $-1    %al
```

|            |    |    |    |    |    |    |    |    |
|------------|----|----|----|----|----|----|----|----|
|            | 63 | 31 |    |    | 15 |    | 7  | 0  |
| <b>rax</b> | 00 | 11 | 22 | 33 | 44 | 55 | 66 | FF |

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

|     |    |    |    |    |    |    |    |    |   |
|-----|----|----|----|----|----|----|----|----|---|
|     | 63 |    | 31 |    |    | 15 |    | 7  | 0 |
| rax | 00 | 11 | 22 | 33 | 44 | 55 | FF | FF |   |



```
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<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。

- | 指令          | 影响                                  | 描述                                     |
|-------------|-------------------------------------|----------------------------------------|
| MOVZ $S, R$ | $R \leftarrow \text{ZeroExtend}(S)$ | 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 $S, R$ | $R \leftarrow \text{SignExtend}(S)$ | 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 |

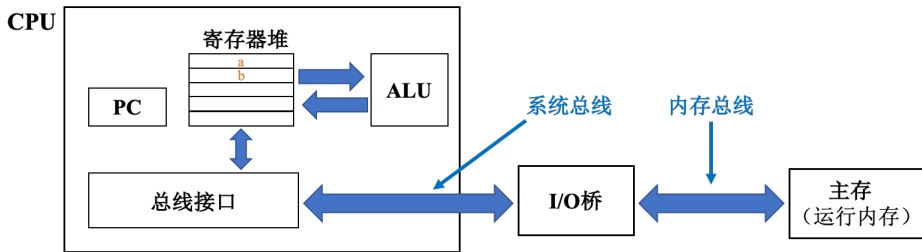
- 对比零扩展和符号扩展，我们可以发现符号扩展比零扩展多一条 4 字节到 8 字节的扩展指令，为什么零扩展没有 `movzbl` 的指令呢？是因为这种情况的数据传送可以使用 `movl` 指令来实现。

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`。

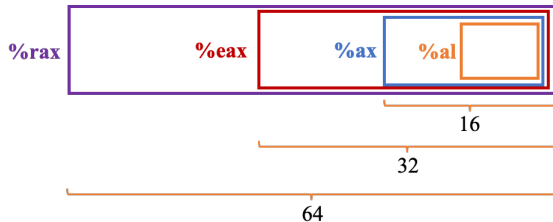
`cltq` 指令效果与图中这条指令的效果一致，只不过编码更紧凑一些。

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

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。



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；



- 程序编码 访问信息

## 代码示例

```
1 int main(){
2     long a = 4;
3     long b = exchange(&a, 3);
4     printf("a = %1d, b = %1d\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。

**exchange:**

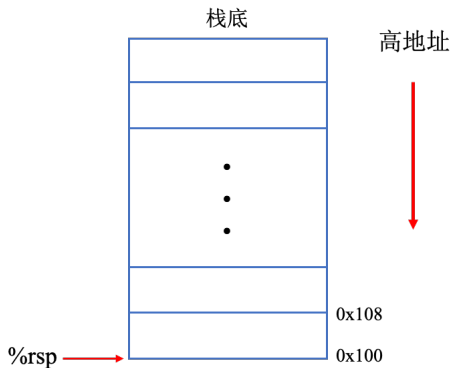
```
      xp in %rdi, y in %rsi  
→ movq (%rdi), %rax      Memory → Register  
  movq %rsi, (%rdi)  
  ret
```



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：

- 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`。



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

程序编码 访问信息

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`。

