Assembly code

registers 寄存器: They are **small memory storage areas built into the processor**.

Instructions 指令: operate on memory (either RAM or registers)，比如加减，mov，jump等

Instruction

MOV:

移动数值到寄存器中。

> would use the **MOV** (move) instruction, which
>
> ```
> MOV RAX, 0x1234
> ```
>
> As you can see, the MOV instruction takes two
>
> MOV is similar to the assignment operator in C
>
> ```
> myVariable = 0x1234;
> ```
>
> You can also move values from one register in
> register RAX:
>
> ```
> MOV RAX, RBX
> ```
>
> In that case, the value of the register RBX isn't
>
> ```
> variable1 = variable2;
> ```

ADD, SUB, AND, OR, NOT, XOR

> The following instruction increments RBX by 4:
>
> ```
> ADD RBX, 4
> ```
>
> RBX +4
>
> The following instruction decrements RBX by 4:
>
> ```
> SUB RBX, 4
> ```
>
> RBX −4
>
> The instructions **AND**, **OR**, **NOT**, and **XOR** are also easy to understand. For example, the following assembly performs a bitwise AND on the RAX register with the 64-bit value `0x0000000000000000`. The result is stored in RAX.
>
> ```
> AND RAX, 0x0000000000000000
> ```

MUL, DIV 乘除

通过移位bit

multiplication (**MUL**), division (**DIV**), shifting bits to the left (SHL) or to the right (SHR) in a register.

MOVZX move with the zero extended.

但因为取1 byte的内容到 4 bytes长的EAX寄存器，就需要补0.

my_arr = rangeforce

C code:	Assembly:
char character = my_array[0];	MOVZX EAX, byte [ESI]
Assuming my_array is a char array with the value "rangeforce", the value of the variable "character" is "r".	The value of EAX is 0x00000072 after the operation.

0x72 = 'r'

ASCII

byte [ESI +1] ，像array [1]取值一样。

比如rangeforce

array的起始位，就是一个字符r，第二个字符就是[ESI +1]

Memory Dereferencing Calculations

move the second character? Or the third? Fortunately, you can perform calculations inside memory dereferencing operations.

	addresses	values	comments
ESI →	0x12345678	0x72616e67	72 61 6e 67 = "rang"
	0x1234567C	0x65666f72	65 66 6f 72 = "efor"
	0x12345680	0x63650000	63 65 = "ce"

C code:	Assembly:
char character = my_array[1];	MOVZX EAX, byte [ESI + 1]
Assuming my_array is a char array with the value "rangeforce", the value of the variable "character" is "a".	The value of EAX is 0x00000061 after the operation.

To get the second character of "**rangeforce**" into EAX, you can do:

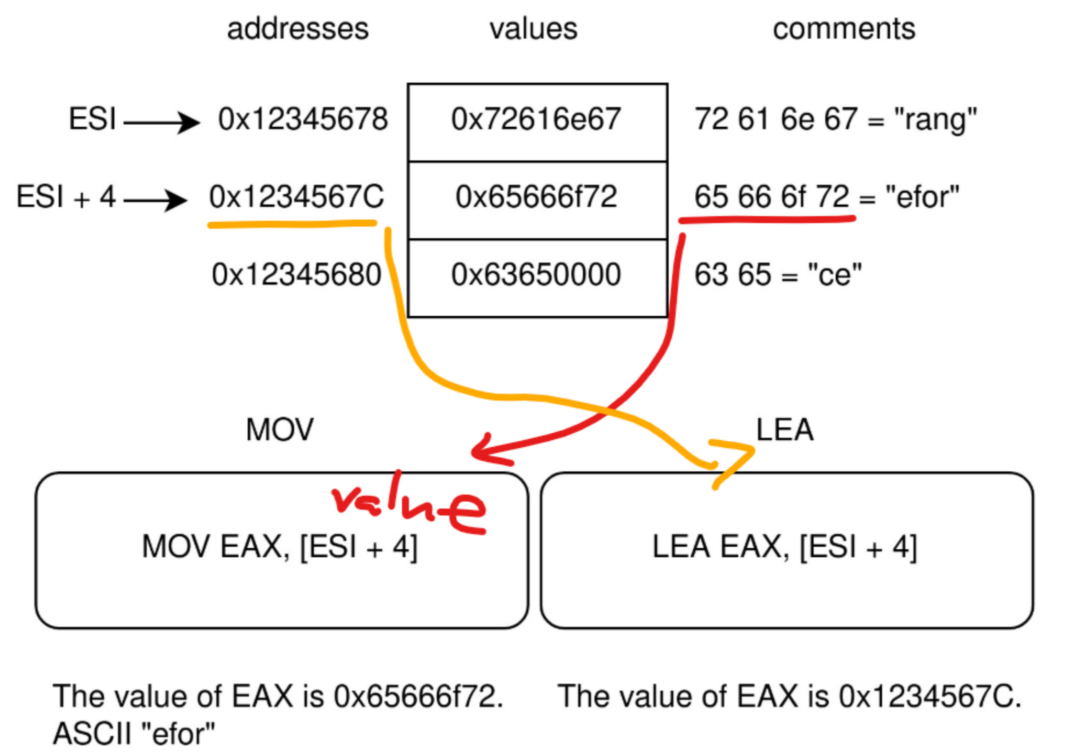
```
1. MOVZX EAX, byte [ESI + 1]
```

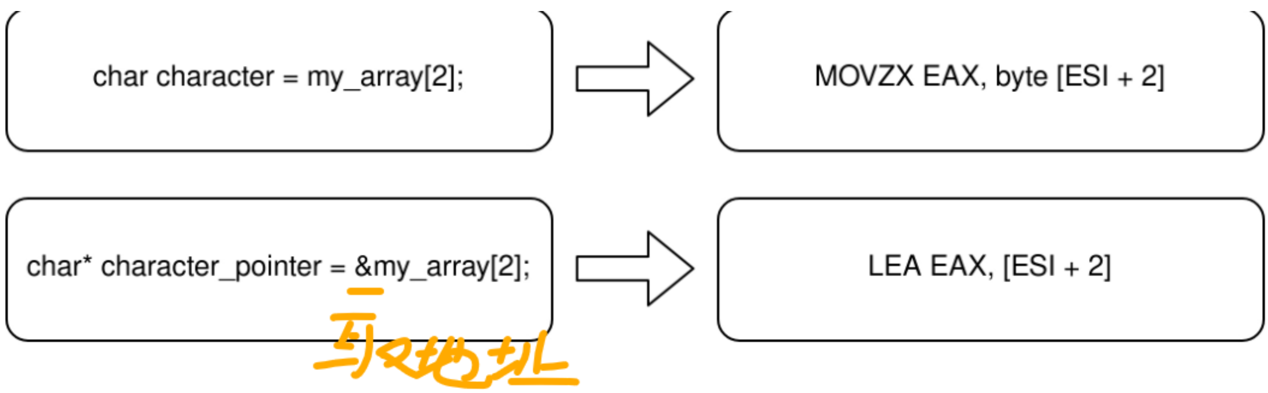
LEA load effective address

获得某value的实际地址

如果你像获得array中第几个字符串的地址，那就使用LEA。

Let's take the previous example with the "**rangeforce**" string again and see how LEA works.





leave

mov ESP, EBP，就是将ebp指向的地址，给esp，这就相当于cleaning这个stack frame

LEAVE

There's one instruction that may cause some confusion if you don't know what it does — **LEAVE**. LEAVE is a high-level instruction, which combines two other instructions into one. Executing the LEAVE instruction is exactly the same as executing the following two instructions:

In 64-bit applications:

```
1. MOV RSP, RBP
2. POP RBP
```

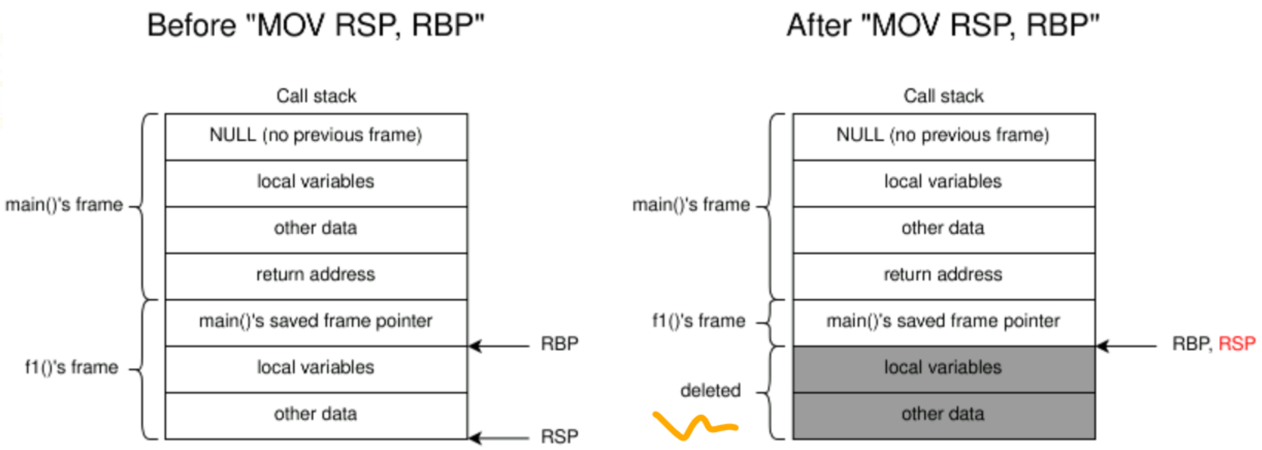
In 32-bit applications:

```
1. MOV ESP, EBP
2. POP EBP
```

结合

离开一个 stack frame

解释



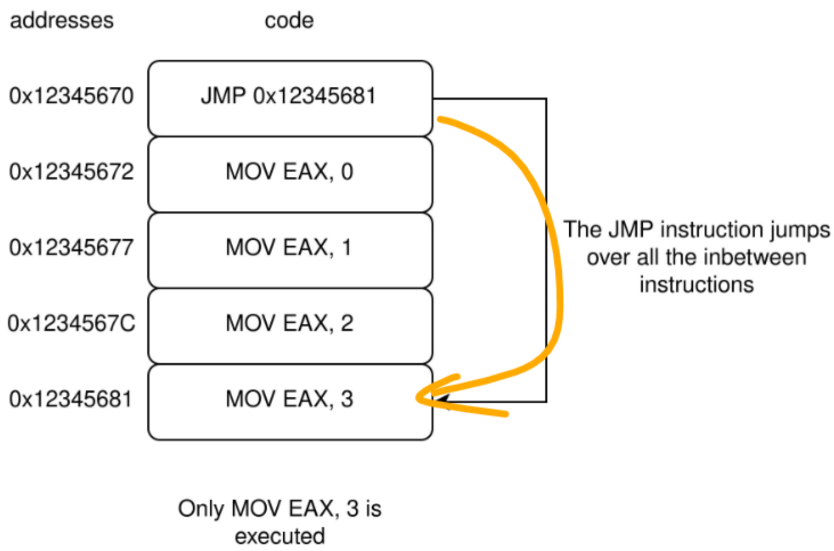
CMP

CMP就是SUB，区别在于不会真的改变其值。

CMP works exactly like **SUB** (subtracts one operand from another), but with one exception — **it doesn't change the value you're comparing**.

jmp, JUMP

The JMP instruction simply changes the **instruction pointer** to point to the specified address. instruction is executed next, the code will "jump" to wherever you point the instruction pointer



Conditional jmp， JNE，JE, JL,JG, JLE, JGE， JZ

通常配合**CMP** compare指令

条件，比如if语句

如果不相同，则jump到指定地址。

In assembly, the conditional jump **JNE (jump if not equal)** is used along with the **CMP** (compare) instruction. Pay attention to how `some code` is not executed when the two variables are not the same:

Addresses	Instructions
0x10000000	CMP EAX, EBX
0x10000002	JNE 0x10000100
...	some code
0x10000100	other code

Explanation: T If EAX **is not equal** to EBX, then the JNE will **jump over** `some code`. `some code` will only get executed if EAX is equal to EBX. The assembly code example assumes that EAX contains the value of the `a` variable, and EBX contains the value of the `b` variable.

L： less than

G: greater than

E: Equal to

Z： zero, JZ, jump if zero

- If the conditional statement was `if (a != b)`, then instead of JNE, the **JE (jump if equals)** instruction would have been used.
- If the statement was `if (a >= b)`, then the **JL (jump if less than)** instruction would have been used.
- If the statement was `if (a <= b)`, then the **JG (jump if greater than)** instruction would have been used.
- If the statement was `if (a > b)`, then the **JLE (jump if less than or equals)** instruction would have been used.
- If the statement was `if (a < b)`, then the **JGE (jump if greater than or equals)** instruction would have been used.

---

while loop 的jump使用

```
while (myvar != 5) {
  // some code
}
```

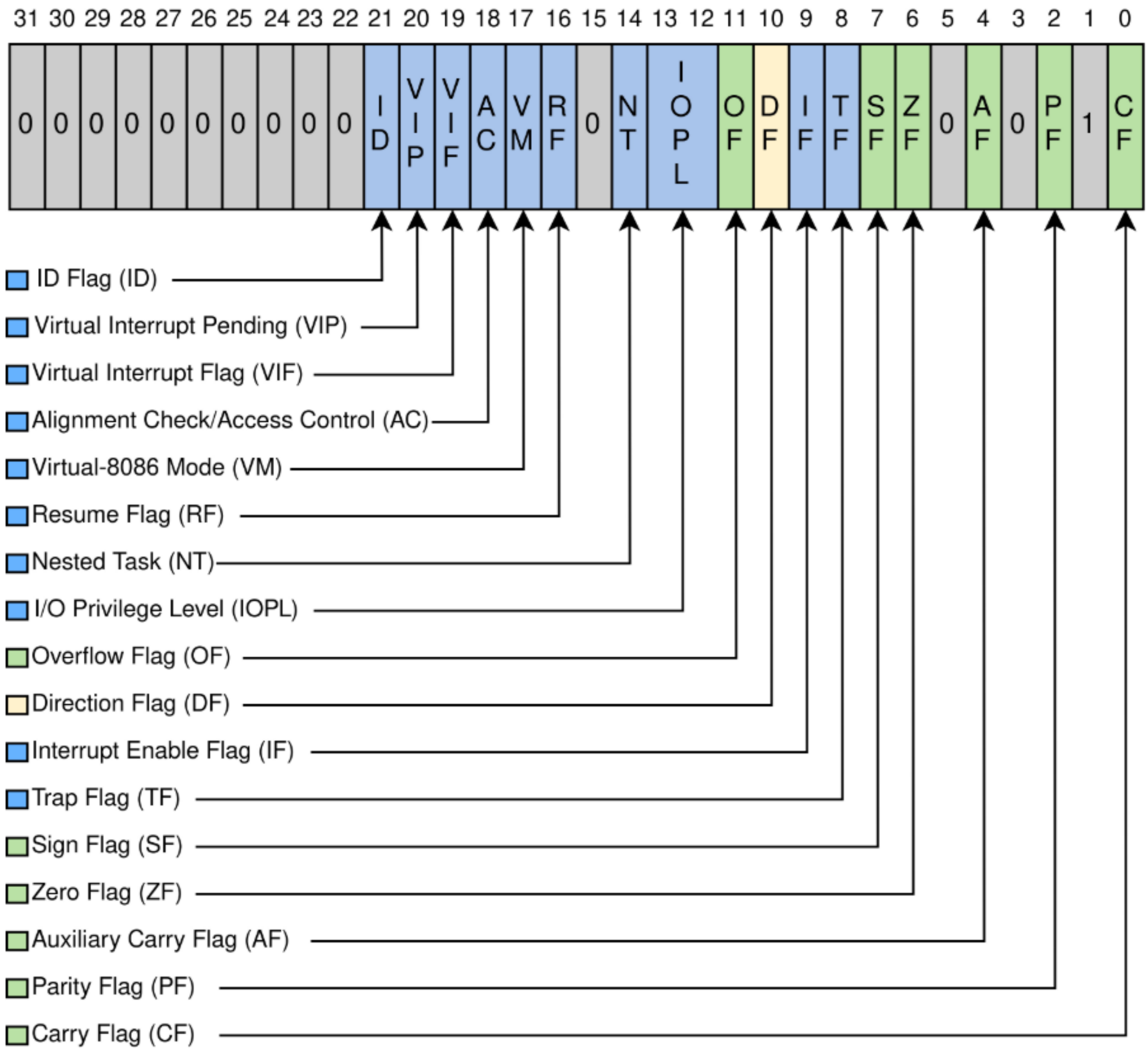
while loop

And this is what it looks like in assembly:

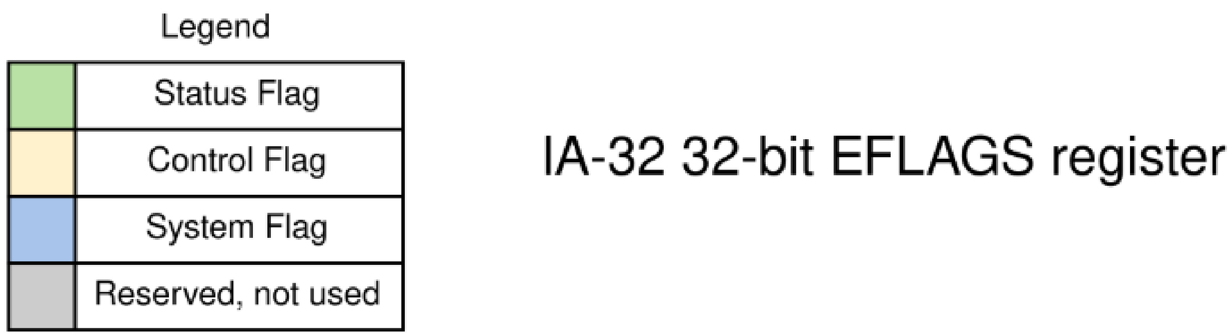
0x10000000	JMP 0x10000050
0x10000002	some code
...	some code
...	some code
0x10000050	CMP [address_of_myvar], 5
0x10000054	JNE 0x10000002

未达条件, jump back

flags



flag的类型



Zero flag

如果两个值相等，则zero flag =1

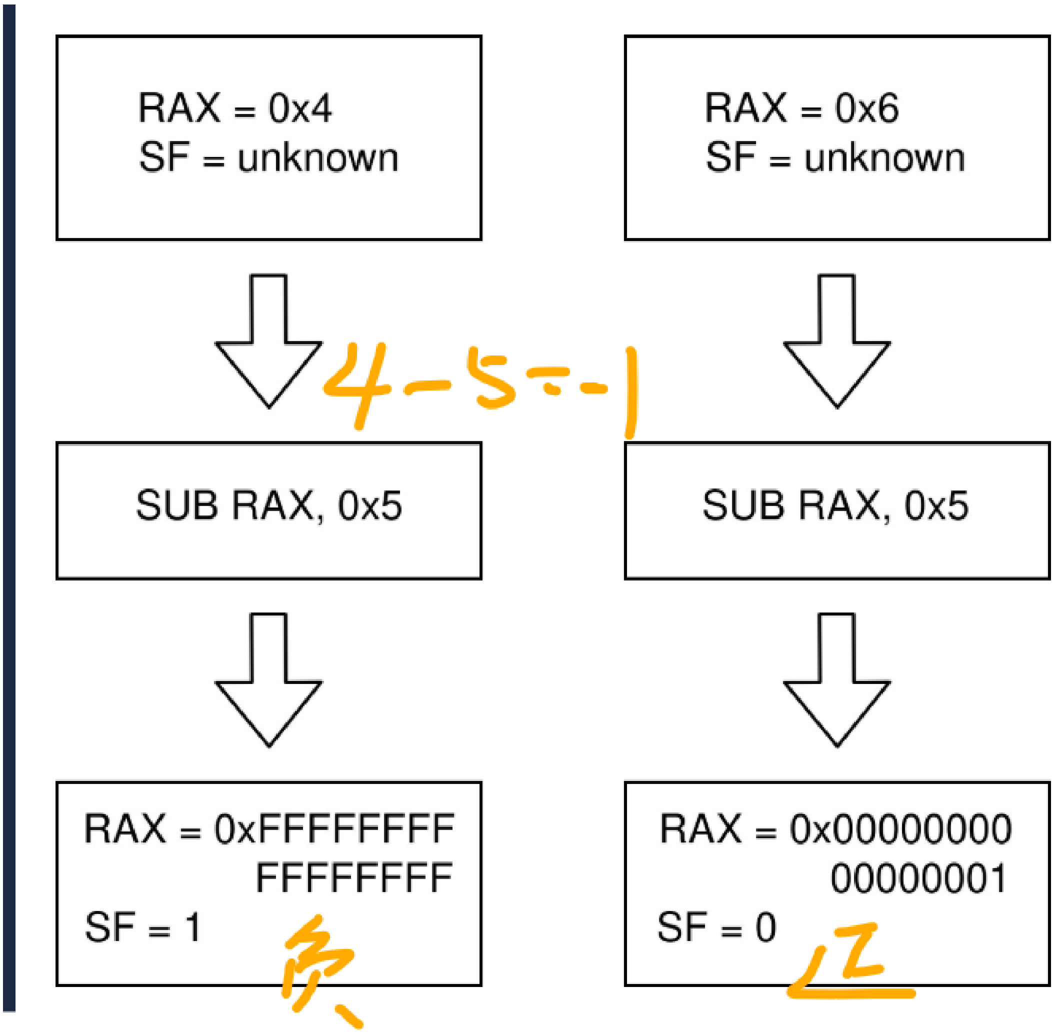
The important takeaway here is that **CMP will set the zero flag to 1 if both operands have the same value**.

通常配合JZ，jump if zero.

Sign flag

如果计算结果是positive, SF= 0，如果是negative, SF=1

The sign flag is used for conditional jumps, such as **JGE** and **JL**. 配合JGE, JL jump

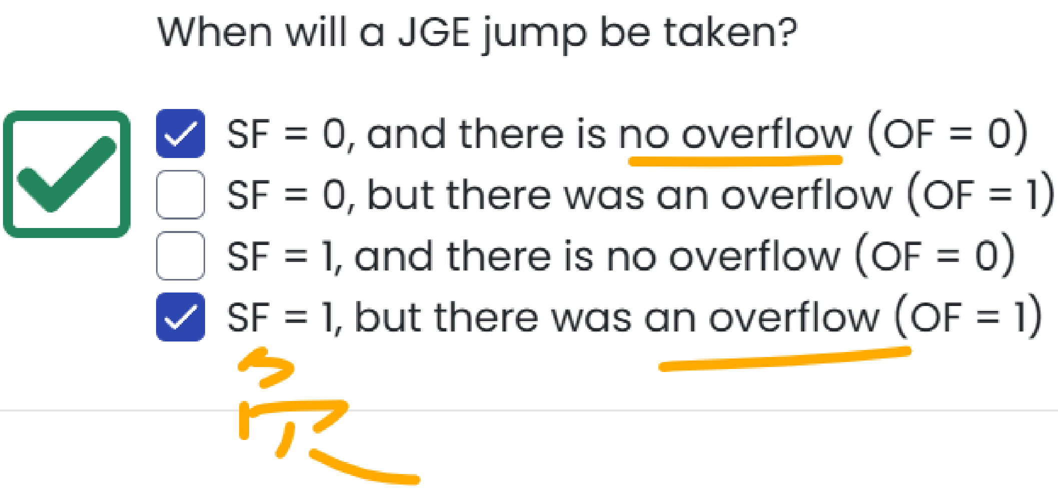


Overflow flag

通常用来检测有符号的结果，是否overflow。

if the overflow flag is set, then that indicates an overflow condition for **signed integers**.

举个例子：if you subtract 1 from 0x00000000 and end up with 0xFFFFFFFF, then that is **not considered an overflow condition** for **signed integers**, but it is an overflow for unsigned integers.



Test

做的是And运算，只不过只改变对应的flags，不会改变其值。

TEST is the same as **AND**, but it only sets flags and doesn't change the value of any operands.

如果RAX是0，那么test rax, rax, 自己和自己比较只能是0，因为AND运算需要 1 & 1=1，只要是0，其结果都是0。因此ZERO flag =1，因为结果为0.

```
TEST RAX, RAX
```

The above instruction sets the zero flag if the value of RAX is 0. That's because if you perform a logical AND on a number with that same number, then the result will be zero only if the number itself was 0 (`0 AND 0`). Otherwise, the result is a nonzero number (the result of `2 AND 2` is 2).

A simpler way of checking whether RAX is 0 is by using the CMP instruction:

```
CMP RAX, 0
```

machine code

汇编只是human readable代码，machine code才是机器能read的

- An **opcode**, which specifies the operation to be performed
- The operands of the instruction

For example, the machine code for `MOV EAX, 0x12345678` is `b878563412`:

- `b8` is the opcode.
- `78563412` is the operand — 0x12345678 in little endian.

1. The leftmost column shows the **address** where the instruction resides in memory.
2. The middle column shows the **machine code** that the computer understands.
3. The rightmost column shows the **human-readable assembly** that corresponds to the opcode.

- Radare2 may add comments next to some instructions. These start with `;`.

```
┌ 27: int main (int argc, char **argv, char **envp);
│          0x5617fdc4b149      f30f1efa       endbr64
│          0x5617fdc4b14d      55             push rbp
│          0x5617fdc4b14e      4889e5         mov rbp, rsp
│          0x5617fdc4b151      488d3dac0e00.  lea rdi, str.Hello_World_ ; 0x5617fdc4c004 ; "Hello World!"
│          0x5617fdc4b158      e8f3feffff     call sym.imp.puts        ; int puts(const char *s)
│          0x5617fdc4b15d      b800000000     mov eax, 0
│          0x5617fdc4b162      5d             pop rbp
└          0x5617fdc4b163      c3             ret
```

address　　machine code　　Assembly

寄存器

64位系统中，有16个寄存器。每一个寄存器是8 bytes(64 bits)

32位系统中，每一个寄存器是4 bytes (32 bits)

RIP 指向下一个要执行的instruction的地址

RSP和RBP来表示一个stack 范围

Register Conventions

There are 16 general-purpose registers plus some special ones (notably the instruction pointer register). In 64-bit systems, each register is 8 bytes (64 bits) long. In 32-bit systems, each register is 4 bytes long.

64-bit general purpose registers 16个

RAX	R8
RBX	R9
RCX	R10
RDX	R11
RSI	R12
RDI	R13
RSP	R14
RBP	R15

Instruction pointer

RIP

Of those 16 + 1 registers, three are special:

- The **instruction pointer** register (RIP) always points to the next instruction that needs to be executed.
- Whenever an instruction is executed, the instruction pointer is automatically changed to point to the next instruction.
- stack The **stack pointer** register (RSP) keeps track of the top of the stack. This register is explained in the **Stack and Heap Basics** module.
- The **stack frame pointer** register (RBP) keeps track of the current stack frame. This register is explained in the **Stack Frames** module.

base pointer

各个Pointer的主要作用。

RCX主要用于loop运算

- RAX (Accumulator register): Stores the return value of a function
- RCX (Counter register): Counter for string and loop operations
- RSI (Source Index register): Source pointer for string operations
- RDI (Destination Index register): Destination pointer for string operations

Keep in mind that these are just conventions. While you **will** often see the registers data operation purposes. The other legacy registers also have names and special conventions for now:

- RBX (Base register)
- RDX (Data register)

RIP, EIP, IP

在16bits system中，ip就是一个16bit的寄存器

32bits，EIP, E for extended IP, 32 bit 寄存器

到64 bit，R 就for 64 bit寄存器

在实际情况中

在一个64bit的软件中，可能会看到32 bit的寄存器使用。

原因是：比如你**只想取RAX内容中的其中32bit内容**，那么就可以使用EAX。其他寄存器也可以以此类推。

之所以这样做是因为原64 bit寄存器需要保存64 bit的return value .

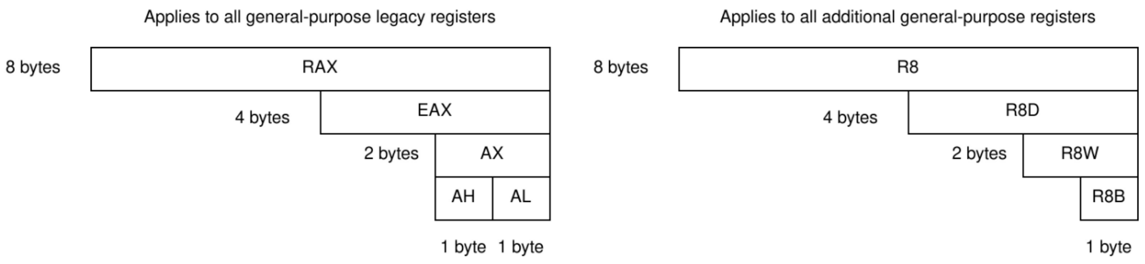
原因

What's going on is that if you only want to access the last 32 bits of the RAX register, then you can do so by using EAX instead of RAX. `MOV EAX, 0` moves the 32-bit value 0x00000000 to the last 32 bits (4 bytes) of the RAX register. In this case, EAX is used instead of RAX because the value it needs to contain is 32 bits long (32-bit **int** data type). Therefore, only the last 4 bytes of RAX need to be used to contain the desired value (0x00000000).

保存原64 bit ret value

The reason why the compiler assigns the value 0 to EAX in this "hello world" example is due to the convention that RAX should be used to store the function's return value. This topic will be covered in more depth in the **x86 Calling Conventions** modules.

Just as you can use EAX to address the last 32 bits of the RAX register, you can also use AX to address the last 16 bits. Furthermore, you can use AH to address the first 8 bits of the AX section, and you can use AL to address the last 8 bits. This naming scheme applies to all legacy registers. For example, to address the last 32 bits of the RBX register, you can use EBX.



R10B

As you can see, the naming for the additional registers is a bit different. R8 addresses the whole 8-byte register (8 bytes is a QUADWORD). **R8D** addresses the last 4 bytes of R8 (4 bytes is a **D**WORD). **R8W** addresses the last 2 bytes, and its length is 1 **W**ORD. R8B addresses the last byte.

汇编语句 Intel, AT&T

Intel vs. AT&T

Intel

```
0000000000001149 <main>:
    1149:       f3 0f 1e fa             endbr64
    114d:       55                      push   rbp
    114e:       48 89 e5                mov    rbp,rsp
    1151:       48 8d 3d ac 0e 00 00    lea    rdi,[rip+0xeac]
    1158:       e8 f3 fe ff ff          call   1050 <puts@plt>
    115d:       b8 00 00 00 00          mov    eax,0x0
    1162:       5d                      pop    rbp
    1163:       c3                      ret
```

And here is the same code in AT&T syntax:

AT&T

```
0000000000001149 <main>:
    1149:       f3 0f 1e fa             endbr64
    114d:       55                      push   %rbp
    114e:       48 89 e5                mov    %rsp,%rbp
    1151:       48 8d 3d ac 0e 00 00    lea    0xeac(%rip),%rdi
    1158:       e8 f3 fe ff ff          callq  1050 <puts@plt>
    115d:       b8 00 00 00 00          mov    $0x0,%eax
    1162:       5d                      pop    %rbp
    1163:       c3                      retq
```

There are many differences but the easy-to-notice ones are that in AT&T syntax:

- There are `%` signs in front of register names.
- The order of the operands is swapped.
- The instructions look a bit different.
- For example, `ret` in Intel syntax might be written as `retq` in AT&T syntax.

Memory dereferencing

一个寄存器，你可以获取它的地址，或取得它包含的value。

[ ]表示取寄存器中的value.

Let's assume ESI contains the memory address of some value. Recapping the information you have learned sc instruction moves the address into the EAX register:

```
MOV EAX, ESI
```

取ESI地址

When the square brackets are used, the value at that address is accessed and moved into the EAX register:

```
MOV EAX, [ESI]
```

取ESI value