

Introdução
Conteúdo
Como usar este livro
A base
Novação geral da arquitetura
Modos de operação
Sintaxe
Registradores de propósito geral
Endereçamento
Pilha
Saltos
Procedimentos
Seções e símbolos
Instruções assembly x86
Instruções do NASM
Pré-processador do NASM
Syscall no Linux
Olá mundo no Linux
Revisão

Aprofundando em Assembly >

Desenvolvendo sua própria C

Powered By GitBook

Instruções assembly x86

Entendendo algumas instruções do Assembly x86

Até agora já foram explicados alguns dos conceitos principais da linguagem Assembly da arquitetura x86, agora que já entendemos como a base funciona precisamos nos munir de algumas instruções para poder fazer códigos mais complexos. Pensando nisso vou listar aqui algumas instruções e uma explicação bem básica de como utilizá-las.

Formato da instrução

Já expliquei a sintaxe de uma instrução no NASM mas não expliquei o formato em si da instrução no código de máquina. Para simplificar uma instrução pode ter os seguintes operandos:

- Um operando registrador
- Um operando registrador OU operando na memória
- Um operando imediato, que é um valor numérico que faz parte da instrução.

Basicamente são três tipos de operandos: Um registrador, valor na memória e um valor imediato. Um exemplo de cada um para ilustrar sendo mostrado como o segundo operando de MOV:

```
mov eax, ebx      ; EBX = Registrador
mov eax, [ebx]    ; [EBX] = Memória
mov eax, 65       ; 65 = Valor imediato
mov eax, "A"       ; "A" = Valor imediato, mesmo que 65
```

Como demonstrado na linha 4 strings podem ser passadas como um operando imediato. O assembler irá converter a string em sua respectiva representação em bytes, só que é necessário ter atenção em relação ao tamanho da string que não pode ser maior do que o operando destino.

São três operandos diferentes e cada um deles é opcional, isto é, pode ou não ser utilizado pela instrução (opcional para a instrução e não para nós).

Repare que somente um dos operandos pode ser um valor na memória ou registrador, enquanto o outro é especificamente um registrador. É devido a isso que há a limitação de haver apenas um operando na memória, enquanto que o uso de dois operandos registradores é permitido.

Notação

① Irei utilizar uma explicação simplificada aqui que irá deixar muita informação importante de fora.

As seguintes nomenclaturas serão utilizadas:

Nomenclatura	Significado
reg	Um operando registrador
r/m	Um operando registrador ou na memória
imm	Um operando imediato
addr	Denota um endereço, geralmente se usa um rótulo. Na prática é um valor imediato assim como o operando imediato.

Em alguns casos eu posso colocar um número junto a essa nomenclatura para especificar o tamanho do operando em bits. Por exemplo $\text{r}/\text{m}16$ indica um operando registrador/memória de 16 bits.

Em cada instrução irei apresentar a notação demonstrando cada combinação diferente de operandos que é possível utilizar. Lembrando que o **operando destino** é o mais à esquerda, enquanto que o **operando fonte** é o operando mais à direita.

① Cada nome de instrução em Assembly é um mnemônico, que é basicamente uma abreviatura feita para fácil memorização. Pensando nisso leia cada instrução com seu nome extenso equivalente para lembrar o que ela faz. No título de cada instrução irei deixar após um “|” o nome extenso da instrução para facilitar nessa tarefa.

MOV | Move

```
mov reg, r/m
mov reg, imm
mov r/m, reg
mov r/m, imm
```

Copia o valor do operando fonte para o operando destino.

```
pseudo.c
destiny = source;
```

ADD

```
add reg, r/m
add reg, imm
add r/m, reg
add r/m, imm
```

Soma o valor do operando destino com o valor do operando fonte, armazenando o resultado no próprio operando destino.

ON THIS PAGE

Formato da instrução

Notação
MOV Move
ADD
SUB Subtract
INC Increment
DEC Decrement
MUL Multiply
DIV Divide
LEA Load Effective Address
AND
OR
XOR Exclusive OR
XCHG Exchange
XADD Exchange and Add
SHL Shift Left
SHR Shift Right
CMP Compare
SETcc Set byte if condition
CMOVcc Conditional Move
NEG Negate
NOT
MOVSB/MOVSW/MOVSD/MOV...
CMPSB/CMPSW/CMPSD/CMP...
LODSB/LODSW/LODSD/LODSQ ...
SCASB/SCASW/SCASD/SCASQ ...

```
pseudo.c  
destiny = destiny + source;
```

SUB | Subtract

```
sub reg, r/m  
sub reg, imm  
sub r/m, reg  
sub r/m, imm
```

Subtrai o valor do operando destino com o valor do operando fonte.

```
pseudo.c  
destiny = destiny - source;
```

INC | Increment

```
inc r/m
```

Incrementa o valor do operando destino em 1.

```
pseudo.c  
destiny++;
```

DEC | Decrement

```
dec r/m
```

Decrementa o valor do operando destino em 1.

```
pseudo.c  
destiny--;
```

MUL | Multiplicate

```
mul r/m
```

Multiplica uma parte do mapeamento de RAX pelo operando fonte passado. Com base no tamanho do operando uma parte diferente de RAX será multiplicada e o resultado armazenado em um registrador diferente.

Operando 1	Operando 2	Destino
AL	r/m8	AX
AX	r/m16	DX:AX
EAX	r/m32	EDX:EAX
RAX	r/m64	RDX:RAX

No caso por exemplo de DX:AX, os registradores de 16 bits são usados em conjunto para representar um valor de 32 bits. Onde DX armazena os 2 bytes mais significativos do valor e AX os 2 bytes menos significativos.

```
pseudo.c  
// Se operando de 8 bits  
AX = AL * operand;  
  
// Se operando de 16 bits  
aux = AX * operand;  
DX = (aux & 0xffff0000) >> 16;  
AX = aux & 0x0000ffff;
```

DIV | Divide

```
div r/m
```

Seguindo uma premissa inversa de MUL, essa instrução faz a divisão de um valor pelo operando fonte passado e armazena o quociente e a sobra dessa divisão.

Operando 1	Operando 2	Destino quociente	Destino sobra
AX	r/m8	AL	AH
DX:AX	r/m16	AX	DX
EDX:EAX	r/m32	EAX	EDX
RDX:RAX	r/m64	RAX	RDX

```
pseudo.c  
// Se operando de 8 bits  
AL = AX / operand;  
AH = AX % operand;
```

LEA | Load Effective Address

```
lea reg, mem
```

Calcula o endereço efetivo do operando fonte e armazena o resultado do cálculo no registrador destino. Ou seja, ao invés de ler o valor no endereço do operando na memória o próprio endereço resultante do cálculo de endereço será armazenado no registrador. Exemplo:

O endereço será armazenado no registrador. Exemplo:

```
    mov rbx, 5
    lea rax, [rbx + 7]
    ; Aqui RAX teria o valor 12
```

AND

```
and reg, r/m
and reg, imm
and r/m, reg
and r/m, imm
```

Faz uma operação *E* bit a bit nos operandos e armazena o resultado no operando destino.

```
pseudo.c
destiny = destiny & source;
```

OR

```
or reg, r/m
or reg, imm
or r/m, reg
or r/m, imm
```

Faz uma operação *OU* bit a bit nos operandos e armazena o resultado no operando destino.

```
pseudo.c
destiny = destiny | source;
```

XOR | Exclusive OR

```
xor reg, r/m
xor reg, imm
xor r/m, reg
xor r/m, imm
```

Faz uma operação *OU Exclusivo* bit a bit nos operandos e armazena o resultado no operando destino.

```
pseudo.c
destiny = destiny ^ source;
```

XCHG | Exchange

```
xchg reg, r/m
xchg r/m, reg
```

O operando **2** recebe o valor do operando **1** e o operando **1** recebe o valor anterior do operando **2**. Fazendo assim uma troca nos valores dos dois operandos. Repare que diferente das instruções anteriores essa modifica também o valor do segundo operando.

```
auxiliary = destiny;
destiny   = source;
source   = auxiliary;
```

XADD | Exchange and Add

```
xadd r/m, reg
```

O operando **2** recebe o valor do operando **1** e, em seguida, o operando **1** é somado com o valor anterior do operando **2**. Basicamente preserva o valor anterior do operando **1** no operando **2** ao mesmo tempo que faz um ADD nele.

```
pseudo.c
auxiliary = source;
source   = destiny;
destiny   = destiny + auxiliary;
```

Essa instrução é equivalente a seguinte sequência de instruções:

```
xchg rax, rbx
add rax, rbx
```

SHL | Shift Left

```
shl r/m
shl r/m, imm
shl r/m, CL
```

Faz o deslocamento de bits do operando destino para a esquerda com base no número especificado no operando fonte. Se o operando fonte não é especificado então faz o *shift left* apenas 1 vez.

```
pseudo.c
destiny = destiny << 1;      // Se: shl r/m
destiny = destiny << source; // Nos outros casos
```

SHR | Shift Right

```
shr r/m
shr r/m, imm
shr r/m, CL
```

Mesmo caso que SHL porém faz o deslocamento de bits para a direita.

```
pseudo.c
destiny = destiny >> 1;      // Se: shl r/m
destiny = destiny >> source; // Nos outros casos
```

CMP | Compare

```
cmp r/m, imm
cmp r/m, reg
cmp reg, r/m
```

Compara o valor dos dois operando e define RFLAGS de acordo.

```
pseudo.c
RFLAGS = compare(operand1, operand2);
```

SETcc | Set byte if condition

```
SETcc r/m8
```

Define o valor do operando de 8 bits para 1 ou 0 dependendo se a condição for atendida (1) ou não (0). Assim como no caso dos *jumps* condicionais, o 'cc' aqui denota uma sigla para uma condição. Cuja a condição pode ser uma das mesmas utilizadas nos *jumps*. Exemplo:

```
sete al
; Se RFLAGS indica um valor igual: AL = 1. Se não: AL = 0
```

```
pseudo.c
if (verify_rflags(condition) == true)
{
    destiny = 1;
}
else
{
    destiny = 0;
}
```

CMOVcc | Conditional Move

```
CMOVcc reg, r/m
```

Basicamente uma instrução MOV condicional. Só irá definir o valor do operando destino caso a condição seja atendida.

```
pseudo.c
if (verify_rflags(condition) == true)
{
    destiny = source;
}
```

NEG | Negate

```
neg r/m
```

Inverte o sinal do valor numérico do operando.

```
pseudo.c
destiny = -destiny;
```

NOT

```
not r/m
```

Faz uma operação *NÃO* bit a bit no operando.

```
pseudo.c
destiny = ~destiny;
```

MOVSB/MOVSW/MOVSD/MOVSQ | Move String

```
movsb ; byte      (1 byte)
movsw ; word       (2 bytes)
movsd ; double word (4 bytes)
movsq ; quad word  (8 bytes)
```

Copia um valor do tamanho de um byte, word, double word ou quad word a partir do endereço apontado por RSI (*Source Index*) para o endereço apontado por RDI (*Destiny Index*). Depois disso incrementa o valor dos dois registradores com o tamanho em bytes do dado que foi movido.

```
pseudo.c
// Se MOVSW
word [RDI] = word [RSI];
RDI      = RDI + 2;
RSI      = RSI + 2;
```

CMPSB/CMPSW/CMPSD/CMPSQ | Compare String

```
cmpsb ; byte      (1 byte)
cmpsw ; word       (2 bytes)
cmpsd ; double word (4 bytes)
```

```
cmpsq ; quad word (8 bytes)
```

Compara os valores na memória apontados por RDI e RSI, depois incrementa os registradores com o tamanho em bytes do dado.

```
pseudo.c
```

```
// CMPSW  
RFLAGS = compare(word [RDI], word [RSI]);  
RDI = RDI + 2;  
RSI = RSI + 2;
```

LODSB/LODSW/LOSDS/LODSQ | Load String

```
lodsb ; byte (1 byte)  
lodsw ; word (2 bytes)  
lodsd ; double word (4 bytes)  
lodsq ; quad word (8 bytes)
```

Copia o valor na memória apontado por RSI para uma parte do mapeamento de RAX equivalente ao tamanho do dado, e depois incrementa RSI com o tamanho do valor.

```
pseudo.c
```

```
// LODSW  
AX = word [RSI];  
RSI = RSI + 2;
```

SCASB/SCASW/SCASD/SCASQ | Scan String

```
scasb ; byte (1 byte)  
scasw ; word (2 bytes)  
scasd ; double word (4 bytes)  
scasq ; quad word (8 bytes)
```

Compara o valor em uma parte mapeada de RAX com o valor na memória apontado por RDI e depois incrementa RDI de acordo.

```
pseudo.c
```

```
// SCASW  
RFLAGS = compare(AX, word [RDI]);  
RDI = RDI + 2;
```

STOSB/STOSW/STOSD/STODQ | Store String

```
stosb ; byte (1 byte)  
stosw ; word (2 bytes)  
stosd ; double word (4 bytes)  
stosq ; quad word (8 bytes)
```

Copia o valor de uma parte mapeada de RAX e armazena na memória apontada por RDI, depois incrementa RDI de acordo.

```
pseudo.c
```

```
// STOSW  
word [RDI] = AX;  
RDI = RDI + 2;
```

LOOP/LOOPE/LOOPNE

```
pseudo.c
```

```
loop addr8  
loope addr8  
loopne addr8
```

Essas instruções são utilizadas para gerar procedimentos de laço (*loop*) usando o registrador RCX como contador. Elas primeiro decrementam o valor de RCX e comparam o mesmo com o valor zero. Se RCX for diferente de zero a instrução faz um salto para o endereço passado como operando, senão o fluxo de código continua normalmente.

No caso de *loope* e *loopne* os sufixos indicam a condição de *Igual* e *não Igual* respectivamente. Ou seja, além da comparação do valor de RCX elas também verificam o valor de RFLAGS como uma condição extra.

```
pseudo.c
```

```
// loop  
RCX = RCX - 1;  
if(RCX != 0)  
{  
    goto operand;  
}  
  
// loope  
RCX = RCX - 1;  
if(RCX != 0 && verify_rflags(EQUAL) == true)  
{  
    goto operand;  
}  
  
// loopne  
RCX = RCX - 1;  
if(RCX != 0 && verify_rflags(EQUAL) == false)  
{  
    goto operand;  
}
```

NOP | No Operation

```
nop
```

Não faz nenhuma operação... Sério, não faz nada. Essa instrução normalmente é utilizada apenas como um "preenchimento" por compiladores afim de alinhar o endereço do código por motivos de otimização.

```
pseudo.c  
EAX = EAX;
```

ⓘ Não cabe a esse livro explicar porque esse alinhamento melhora a performance do código mas se estiver curioso estude à respeito do cache do processador e *cache line*. Para simplificar um desvio de código para um endereço que esteja próximo ao início de uma linha de cache é mais performático.

Se você é um escovador de bits sugiro ler à respeito no [manual de otimização da Intel](#) no tópico [3.4.1.4 Code Alignment](#).



Previous
Seções e símbolos

Next
Instruções do NASM



Last modified 2yr ago

WAS THIS PAGE HELPFUL?