


Código de máquina

Entendendo o código de máquina x86-64

O famigerado **código de máquina** (também chamado de **linguagem de máquina**), popularmente conhecido como "zeros e uns", são as instruções que o processador interpreta e executa. São basicamente números onde o processador decodifica esses números afim de executar determinadas operações identificadas pelas instruções.

Acho que boa parte das pessoas da área da computação sabem que processadores de computadores digitais funcionam com sinais elétricos com duas tensões diferentes: Uma alta (lá pelos 3v, mas pode variar de acordo com o processador) e uma baixa (perto de 0v), onde a tensão alta representa o 1 e a tensão baixa representa o 0.

Mas comumente é só isso o que as pessoas sabem sobre código de máquina. O objetivo deste capítulo é dar uma noção aprofundada de como funciona o código de máquina da arquitetura x86-64.

 Cada arquitetura de processador (vulgo ISA, *Instruction Set Architecture*) têm um código de máquina distinto. Portanto as informações aqui são válidas para código de máquina x86 e x86-64. ARM, RISC-V etc. contém código de máquina que funciona de um jeito completamente diferente.

Representação textual

Antes de mais nada um pré-aviso: Sei que é romântico quando se fala de código de máquina meter um monte de zeros e uns (como: `10110100010`). Mas na vida real **ninguém** representa textualmente código de máquina em binário. Isso é normalmente feito em manuais ou ferramentas como *disassemblers* e *debuggers* usando **hexadecimal**.

Então ao pensar em código de máquina não pense nisso `10110100 00001110` mas sim nisso `B4 0E`. Você é humano, pense como tal.

Ferramentas

Comecei a desenvolver uma ferramenta exclusivamente para ser usada como auxílio para esse capítulo. Eu a chamei de **x86-visualizer** e seu intuito é você escrever uma instrução em Assembly e ela lhe exibir o código de máquina dividido em seus campos, assim facilitando o entendimento.

A ferramenta não está concluída então poucas instruções irão funcionar, todavia sugiro seu uso durante a leitura do capítulo afim de facilitar o entendimento da codificação das instruções.

Acesse o repositório dela aqui:

- <https://github.com/Silva97/x86-visualizer> ↗

Também sugiro usar o **ndisasm** afim de fazer experimentações. Ele é um *disassembler* que vem junto com o **nasm** e [já foi utilizado anteriormente no livro](#).

Formato das instruções

O formato das instruções do código de máquina.

CISC

Primeira coisa que a gente precisa saber é que a arquitetura x86-64 é CISC (*Complex Instruction Set Computer*), ou seja uma arquitetura que contém um conjunto complexo de instruções.

O que significa na prática que a arquitetura contém muitas instruções consideradas "complexas", que efetuam muitas operações de uma vez. Por exemplo a instrução `rep movsb` faz um bocado de coisas:

1. Copia o valor em `DS:ESI` para `ES:EDI`.
2. Incrementa o valor de ESI.
3. Incrementa o valor de EDI.
4. Decrementa o valor de ECX.
5. Verifica se o valor de ECX é zero. Se for finaliza o *loop*.

Tudo isso em apenas uma instrução.

Formato

Esse é o formato de uma instrução do código de máquina da arquitetura segundo os manuais da Intel:

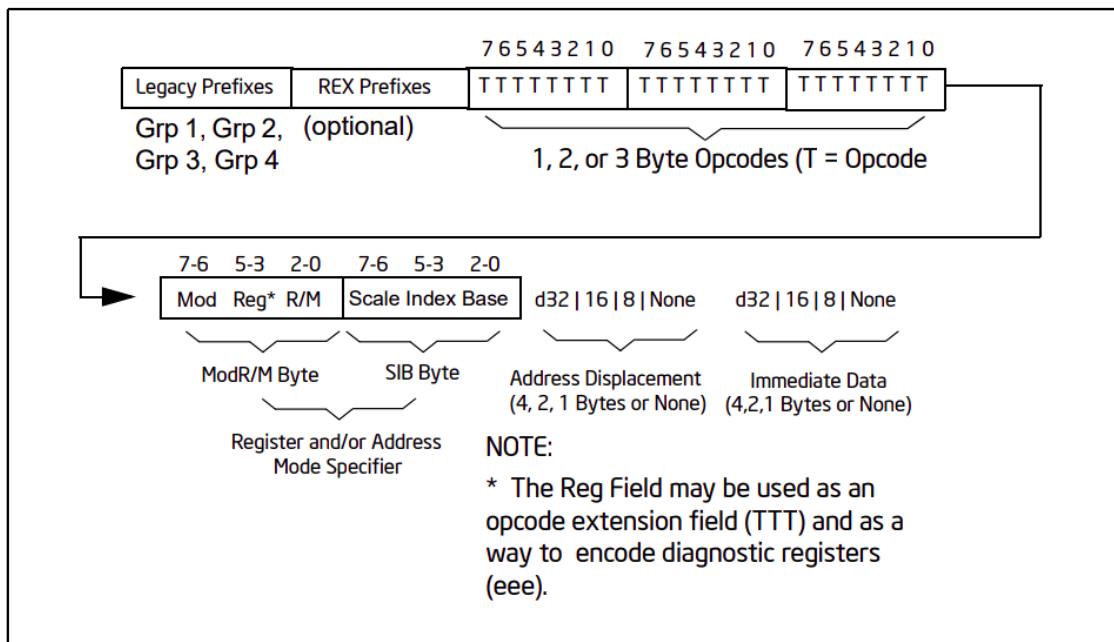


Figure B-1. General Machine Instruction Format

Figura retirada dos [manuais da Intel](#) ↗. Apêndice B do volume 2.

Legacy prefixes: são prefixos que existem desde o x86, alguns até mesmo desde o 8086. Por isso são chamados de "*legacy*" (legados).

REX prefix: é um prefixo novo existente somente no modo de 64-bit e adicionado em processadores x86-64.

Opcode: abreviação para *operation code* (código de operação), é um valor numérico (de 1 a 3 bytes de tamanho) que identifica qual operação o processador deve executar. Desde mover valores, subtrair, somar, calcular a raiz quadrada, modificar o valor de um registrador etc.

ModR/M: é um byte na instrução que não está presente em todas elas. Explico em detalhes depois mas ele serve para definir o modo de endereçamento e/ou qual registrador é usado na operação. Por isso o R/M, que é uma abreviação para *Register/Memory*.

SIB: dependendo do modo de endereçamento definido em **ModR/M**, o byte SIB pode ser usado. Ele define três valores:


- **Scale** (2 bits): determina um fator de "escala" (1, 2, 4 ou 8) que irá multiplicar o valor do **index**.
- **Index** (3 bits): define o registrador que será usado como índice.
- **Base** (3 bits): define o registrador que será usado como base. Na prática o cálculo do endereçamento é feito como na seguinte pseudo-expressão:

```
address = base + index * scale
```

Displacement: é um valor numérico de 1, 2 ou 4 bytes de tamanho que é somado ao endereçamento definido por ModR/M. Nem todo modo de endereçamento definido por ModR/M usa o *displacement*, então nem sempre ele está presente em uma instrução com operando na memória.


Immediate: é um valor numérico de 1, 2 ou 4 bytes de tamanho usado em algumas operações que usam um operando imediato. Por exemplo `mov ah, 0x0E`, onde o número `0x0E` (14 em decimal) é o valor **immediate** na instrução.

Inclusive a instrução `B4 0E` que [mencionei anteriormente](#) é a `mov ah, 0x0E`. Onde `B4` é o **opcode** (de 1 byte) e `0E` o **immediate** (de 1 byte também).

 Uma instrução na arquitetura x86 pode ter de 1 até 15 bytes de tamanho. E caso ainda não tenha ficado claro: sim, instruções na arquitetura x86 têm tamanhos variados.

Atributos e prefixos

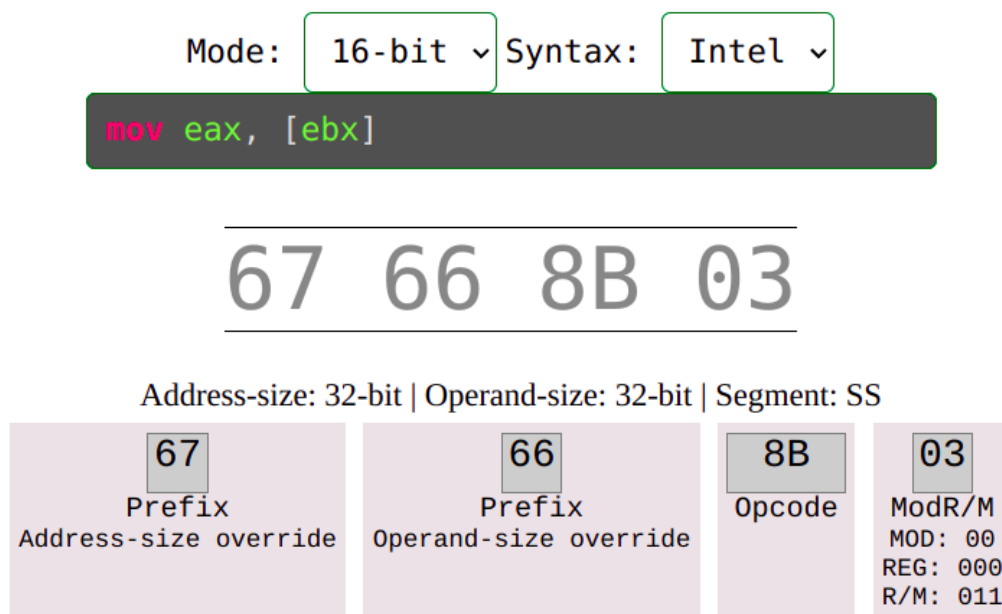
Entendendo os prefixos no código de máquina.

 Os dois tópicos [atributos](#) e [prefixos](#) já explicaram esse assunto antes no livro, mas do ponto de vista do Assembly. Aqui será abordado o assunto mais voltado ao código de máquina e **com mais informações**.

Na arquitetura x86 as instruções contém o que é conhecido como "atributos", onde existe um determinado valor padrão para o atributo e é possível modificá-lo com um prefixo.

Como pode ser observado na ilustração exibida no tópico [Formato das instruções](#), prefixos são bytes que podem (são opcionais na grande maioria das instruções) ser adicionados antes do *opcode* de uma instrução.

Uma instrução pode ter mais de um prefixo (até 4 legados). O prefixo REX existente somente em x86-64 precisa obrigatoriamente vir antes do *opcode* e depois dos demais prefixos. Mas exceto por ele, todos os outros prefixos podem ser adicionados em qualquer ordem que não fará diferença na instrução. Por exemplo a instrução `mov eax, [ebx]` em modo de 16-bit seria compilada como na imagem:



Print do [x86-visualizer](#).

Onde `67 66 8B 03` e `66 67 8B 03` dariam na mesma, o processador executaria as duas instruções de maneira totalmente equivalente.

Atributo address-size

Modos de operação

>

Em modo de 16-bit e modo de 32-bit, desde o processador i386, é possível usar tanto [endereçamento](#) de 16-bit como de 32-bit. No exemplo anterior a instrução `mov eax, [ebx]` foi compilada no modo de 16-bit, porém usando endereçamento e operando de 32-bit.

O atributo **address-size** determina o modo de endereçamento da instrução. Em modo 16-bit o atributo **address-size** por padrão é de 16-bit. E em modo de 32-bit o atributo é por padrão de 32-bit. Já em modo de 64-bit o endereçamento padrão é 64-bit.

O prefixo conhecido como **address-size override**, cujo o byte é `67`, serve para usar o modo de endereçamento não-padrão. Ou seja, ao usar o prefixo se estiver em modo de 16-bit o endereçamento será de 32-bit. E se estiver em modo de 32-bit o endereçamento será de 16-bit. Já se estiver em modo de 64-bit o endereçamento será de 32-bit.

Por isso o prefixo é adicionado em 16-bit para instruções que usam endereçamento de 32-bit. O mesmo também é feito na situação oposta:

Mode: 32-bit Syntax: Intel

```
mov eax, [bx]
```

67 8B 07

Address-size: 16-bit | Operand-size: 32-bit | Segment: DS

67	8B	07
Prefix	Opcode	ModR/M
Address-size override		MOD: 00
		REG: 000
		R/M: 111

[Print do x86-visualizer ↗](#).

Atributo operand-size

Assim como é possível alternar entre endereçamento de 16-bit e 32-bit nos modos de 16-bit (*real mode*) e 32-bit (*protected mode*). Também é possível alternar o tamanho dos operandos usados em operações.

Assim como também foi demonstrado no primeiro exemplo a instrução de 16-bit fez uma operação com um valor de 32-bit (o registrador EAX teve seu valor alterado para os 4 bytes presentes no endereço `[EBX]`).

E para isso foi usado o prefixo **operand-size override**, o byte `66`. E na mesma lógica do `address-size override` ele altera o tamanho do operando para o seu tamanho não-padrão. Onde em modos de 32-bit e 64-bit o tamanho padrão de operando é de 32-bit, e em modo de 16-bit o tamanho padrão é de 16-bit.

- i** Vale citar um erro que eu vi um senhor cometer uma vez: Ele acreditava que em modo de 32-bit era possível usar registradores de 64-bit e endereçamento de 64-bit. Bem, isso está **errado** como você pode notar pela explicação acima.

Em modo de 16-bit é possível usar registradores e endereçamento de 32-bit alterando os atributos **address-size** e **operand-size**. Mas o mesmo não se aplica para 64-bit porque o uso de operandos de 64-bit é feito por meio do prefixo REX, que **só existe em modo de 64-bit**. E em modo de 32-bit só é possível alternar entre endereçamento de 32-bit e 16-bit usando o prefixo `67`.

Atributo segment

Registradores de segmento



Qual segmento de memória será acessado pela instrução é definido em um atributo. O segmento padrão da instrução é definido de acordo com qual registrador foi usado como **base**:

Registrador base	Segmento
RIP	CS
SP/ESP/RSP	SS
BP/EBP/RBP	SS
Qualquer outro registrador	DS

Para alterar o atributo de segmento para um outro segmento de memória é usado um prefixo distinto por segmento:

Segmento	Byte do prefixo
CS	<code>2E</code>
DS	<code>3E</code>
ES	<code>26</code>
FS	<code>64</code>
GS	<code>65</code>

Exemplo:

Mode: 32-bit Syntax: Intel

```
mov eax, es:[ebp]
```

26 8B 45 00

Address-size: 32-bit | Operand-size: 32-bit | Segment: SS

26	8B	45	00
Prefix ES	Opcode	ModR/M MOD: 01 REG: 000 R/M: 101	Displacement 8-bit (0)

Print do [x86-visualizer](#).

Prefixos REP/REPE e REPNE


As instruções de movimentação de dados (`movsb`, `movsw`, `movsd` e `movsq`) bem como outras como `scasb`, `lodsb`, `in`, `out` etc. podem ser executadas em *loop* usando o prefixo REPE ou REPNE.

No caso das instruções `MOVS*` é possível usar o prefixo REPE, que nesse caso também pode ser chamado só de `REP` mas os dois mnemônicos produzem o mesmo byte (`F3`).

Ao usar esse prefixo na instrução, assim como foi [explicado anteriormente](#), ela é executada em *loop* enquanto o valor de ECX não for zero. E a cada iteração do *loop* o valor do registrador é decrementado. Na verdade se CX ou ECX será usado isso é definido pelo atributo **address-size** e pode ser alternado com o prefixo **address-size override**. Por exemplo na sintaxe do NASM ficaria assim:

```
bits 16
; ...
a32 rep movsb
```

Assim ECX seria usado ao invés de CX. Onde `a32` é uma palavra-chave usada no NASM para denotar que o **address-size** daquela instrução deve ser de 32-bit. Se usado em modo de 16-bit ele adiciona o prefixo `67`, mas se estiver em modo de 32-bit então nenhum prefixo será adicionado tendo em vista que o **address-size** padrão já é de 32-bit.

 Sim, também existe `a16` e `a64`. Como também existe `o16`, `o32` e `o64` para denotar o tamanho do **operand-size**. Mas detalhe que `a64` e `o64` denotam o uso do prefixo REX que só existe em modo de 64-bit.

Nas instruções `CMPS*` e `SCAS*` o prefixo `REPE` (ou `REPZ`) repete a instrução enquanto a *zero flag* estiver setada. Já `REPNE` (ou `REPNZ`) repete enquanto a *zero flag* estiver zerada.

Prefixo LOCK

O prefixo LOCK (byte `F0`) é usado para fazer operações de escrita atômica em um determinado endereço de memória. Ou seja o prefixo garante que outros núcleos do processador não escrevam naquele endereço ao mesmo tempo, exigindo que essa operação finalize antes de outra que escreva no mesmo endereço seja executada.

Esse prefixo só pode ser usado nas seguintes instruções: `ADD`, `ADC`, `AND`, `BTC`, `BTR`, `BTS`, `CMPXCHG`, `CMPXCH8B`, `CMPXCHG16B`, `DEC`, `INC`, `NEG`, `NOT`, `OR`, `SBB`, `SUB`, `XOR`, `XADD` e `XCHG`. Isso, obviamente, quando o operando destino (o que está sendo escrito) é um operando na memória.

Na sintaxe do NASM o prefixo pode ser usado simplesmente com a palavra-chave `lock` antes da instrução. Como em:

```
lock add [ebx], 4
```

Prefixos de branch hint

É possível manualmente você instruir para o sistema de **branch prediction** do processador quais saltos condicionais provavelmente irão ocorrer ou não usando dois prefixos:

- `2E` - Instrui para o processador que o pulo provavelmente **não** ocorrerá.
- `3E` - Instrui para o processador que provavelmente o pulo ocorrerá.

Na sintaxe do NASM esses prefixos podem ser adicionados em saltos condicionais com as palavra-chaves `false` e `true` respectivamente. Como em:

```
false jz my_label
```

Todavia esses prefixos são obsoletos e até mesmo ignorados por processadores mais novos, tendo em vista que processadores mais modernos usam um algoritmo para determinar qual salto é mais provável de ser tomado ou não. E também saltos para trás são considerados tomados e saltos para frente como não tomados. Isso por causa da forma como compiladores geram código para *loops* e condicionais.

Em versões mais modernas do NASM ele simplesmente irá ignorar o `false` ou `true` e não adicionará prefixo algum.

Immediate

Campo immediate na instrução do código de máquina.

O campo *immediate* (valor "imediato") pode ter 1, 2, ou 4 bytes de tamanho. Ele é o operando numérico presente em algumas instruções. Exemplo:

```
mov eax, 0x11223344
```

Essa instrução em código de máquina fica: B8 44 33 22 11

Onde B8 é o *opcode* da instrução e 44 33 22 11 o valor imediato (0x11223344). Lembrando que a arquitetura x86 é *little-endian*, portanto o valor imediato fica em *little-endian* na instrução.

O tamanho desse campo é definido pelo atributo **operand-size**, portanto ao usar o prefixo 66 o seu tamanho pode alternar na instrução entre 16-bit e 32-bit. Sobre instruções com operandos de 8-bit, como `mov al, 123`, existem *opcodes* específicos para operandos nesse tamanho portanto o prefixo não é usado nessas instruções. E obrigatoriamente o *immediate* terá 8-bit de tamanho.


Outros dois exemplos seriam `mov ax, 0x1122` e `mov al, 0x11`. Onde o primeiro tem o código de máquina 66 B8 22 11 em modo de 32-bit, e em modo de 16-bit fica igual só que sem o prefixo 66.

Já a segunda instrução terá o código de máquina B0 11 em qualquer modo de operação, já que ela independe do **operand-size**.

Displacement

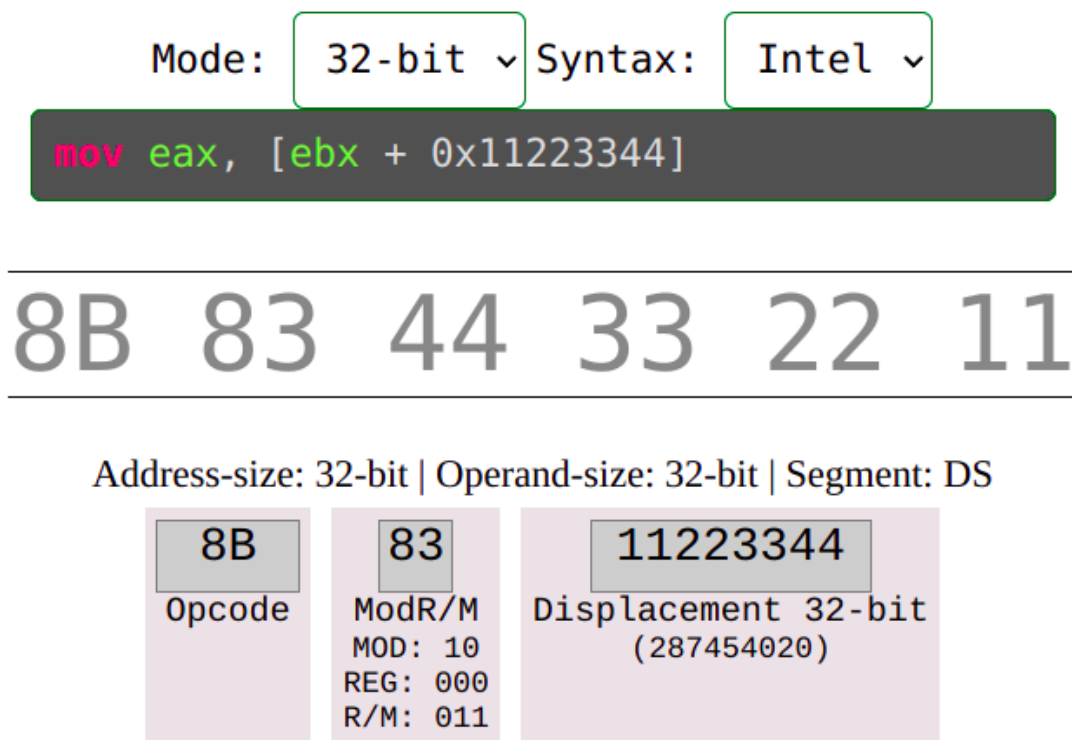
Campo displacement na instrução do código de máquina.

O *displacement* (deslocamento) é um valor numérico de 1, 2 ou 4 bytes de tamanho que também faz parte da instrução assim como o valor imediato.

 Em modo de 32-bit ou 64-bit, o *displacement* pode ser de 1 ou 4 bytes de tamanho. Em modo de 16-bit pode ser de 1 ou 2 bytes de tamanho.

Ele é um valor numérico que é somado ao [endereço](#) definido pelo byte ModR/M. Se esse campo está presente ou não na instrução, bem como seu tamanho, é definido no byte ModR/M.

Exemplo:



Print do [x86-visualizer](#).

Onde o valor `0x11223344` na instrução `mov eax, [ebx + 0x11223344]` é o *displacement* da instrução.

ModR/M e SIB

Entendendo os byte ModR/M e SIB.

Como já foi mencionado anteriormente o byte ModR/M é usado em algumas instruções para especificar o operando na memória ou registrador.

Em Assembly existem dois "tipos" de instruções que recebem dois operandos:

1. As que tem um operando registrador e imediato. Exemplo: `mov eax, 123`
2. As que tem um operando na memória ou dois operandos registradores. Exemplos: `mov [ebx], 123` e `mov eax, ebx`.

O primeiro tipo não precisa do byte ModR/M, pois o registrador destino é especificado nos 3 últimos bits do byte do [opcode](#). Por exemplo o opcode `B8` da instrução `mov eax, 123` é o seguinte em binário: `10111000` Onde o número zero (`000`) é o código para identificar o registrador EAX.

Codificação dos registradores

>

Um jeito mais simples de especificar esse campo no opcode sem precisar lidar com binário é simplesmente somar o opcode "base" (correspondente ao uso de AL/AX/EAX) mais o código do registrador. Por exemplo se a instrução `B8` (`B8 + 0`) corresponde a `mov eax, 123`, então o opcode `BB` (`B8 + 3`) é `mov ebx, 123`. E se eu quiser fazer `mov bx, 123` basta adicionar o prefixo `66` à instrução.


Já as instruções do segundo tipo usam o byte ModR/M para definir o operando destino na memória (no caso de instruções sem o operando registrador) ou os dois operandos. Onde o byte ModR/M consiste nos três campos:

- `MOD` - Os primeiros 2 bits que definem o "modo" do operando R/M.
- `REG` - Os 3 próximos bits que definem o código do operando registrador.
- `R/M` - Os 3 últimos bits que definem o código do operando R/M.

O byte define 2 operandos:

1. Um operando que é sempre um registrador, definido no campo **REG**.
2. Um operando que pode ser um registrador ou operando na memória.

Para que o campo **R/M** defina também o código de um registrador, assim como o **REG**, o valor 3 (11 em binário) deve ser usado no campo **MOD**.

 Um adendo sobre o byte ModR/M é que em algumas instruções o campo **REG** é usado como uma extensão do opcode.

É o caso por exemplo das instruções `inc dword [ebx]` (FF 03) e `dec dword [ebx]` (FF 0B) que contém o mesmo byte de opcode mas fazem operações diferentes.

Repare como o campo R/M é necessário para especificar o operando na memória mas o REG fica "sobrando", por isso os engenheiros da Intel tomaram essa decisão minimamente confusa (vulgo gambiarra), afim de aproveitar dessa peculiaridade em instruções que precisam de um operando na memória mas não precisam de um operando registrador.

Para os demais valores do campo **MOD** os seguintes endereçamentos são feitos de acordo com o valor de **R/M**:

Endereçamento em 16-bit

MOD 00

R/M	Endereçamento
000	[BX+SI]
001	[BX+DI]
010	[BP+SI]
011	[BP+DI]
100	[SI]
101	[DI]
110	displacement 16-bit

MOD 01

R/M	Endereçamento
000	[BX+SI] + displacement 8-bit
001	[BX+DI] + displacement 8-bit
010	[BP+SI] + displacement 8-bit
011	[BP+DI] + displacement 8-bit
100	[SI] + displacement 8-bit
101	[DI] + displacement 8-bit
110	[BP] + displacement 8-bit
111	[BX] + displacement 8-bit

MOD 10

R/M	Endereçamento
000	[BX+SI] + displacement 16-bit
001	[BX+DI] + displacement 16-bit
010	[BP+SI] + displacement 16-bit
011	[BP+DI] + displacement 16-bit
100	[SI] + displacement 16-bit
101	[DI] + displacement 16-bit
110	[BP] + displacement 16-bit
111	[BX] + displacement 16-bit

Endereçamento em 32-bit

MOD 00

R/M	Endereçamento
000	[eax]
001	[ecx]
010	[edx]
011	[ebx]
100	SIB
101	displacement 32-bit
110	[esi]
111	[edi]


MOD 01

R/M	Endereçamento
000	[eax] + displacement 8-bit
001	[ecx] + displacement 8-bit
010	[edx] + displacement 8-bit
011	[ebx] + displacement 8-bit
100	SIB + displacement 8-bit
101	[ebp] + displacement 8-bit
110	[esi] + displacement 8-bit
111	[edi] + displacement 8-bit

MOD 10

R/M	Endereçamento
000	[eax] + displacement 32-bit
001	[ecx] + displacement 32-bit
010	[edx] + displacement 32-bit
011	[ebx] + displacement 32-bit
100	SIB + displacement 32-bit
101	[ebp] + displacement 32-bit
110	[esi] + displacement 32-bit
111	[edi] + displacement 32-bit

Endereçamento em 64-bit

 Devido ao [prefixo REX](#) o campo R/M é estendido em 1 bit no modo de 64-bit.

MOD 00

R/M	Endereçamento
0000	[rax/eax]
0001	[rcx/ecx]
0010	[rdx/edx]
0011	[rbx/ebx]
0100	SIB
0101	[rip/eip] + displacement 32-bit
0110	[rsi/esi]
0111	[rdi/edi]

1000	[r8/r8d]
1001	[r9/r9d]
1010	[r10/r10d]
1011	[r11/r11d]
1100	SIB
1101	[rip/eip] + displacement 32-bit
1110	[r14/r14d]

MOD 01

R/M	Endereçamento
0000	[rax/eax] + displacement 8-bit
0001	[rcx/ecx] + displacement 8-bit
0010	[rdx/edx] + displacement 8-bit
0011	[rbx/ebx] + displacement 8-bit
0100	SIB + displacement 8-bit
0101	[rbp/ebp] + displacement 8-bit
0110	[rsi/esi] + displacement 8-bit
0111	[rdi/edi] + displacement 8-bit
1000	[r8/r8d] + displacement 8-bit
1001	[r9/r9d] + displacement 8-bit
1010	[r10/r10d] + displacement 8-bit
1011	[r11/r11d] + displacement 8-bit
1100	SIB + displacement 8-bit
1101	[r13/r13d] + displacement 8-bit

1110	[r14/r14d] + displacement 8-bit
------	---------------------------------

MOD 10

R/M	Endereçamento
0000	[rax/eax] + displacement 32-bit
0001	[rcx/ecx] + displacement 32-bit
0010	[rdx/edx] + displacement 32-bit
0011	[rbx/ebx] + displacement 32-bit
0100	SIB + displacement 32-bit
0101	[rbp/ebp] + displacement 32-bit
0110	[rsi/esi] + displacement 32-bit
0111	[rdi/edi] + displacement 32-bit
1000	[r8/r8d] + displacement 32-bit
1001	[r9/r9d] + displacement 32-bit
1010	[r10/r10d] + displacement 32-bit
1011	[r11/r11d] + displacement 32-bit
1100	SIB + displacement 32-bit
1101	[r13/r13d] + displacement 32-bit
1110	[r14/r14d] + displacement 32-bit
1111	[r15/r15d] + displacement 32-bit

Byte SIB

Os endereçamentos com R/M **100** (em 32-bit e 64-bit) são os que usam o byte SIB (exceto **MOD 11**), que como já foi explicado anteriormente contém os campos **Scale**, **Index** e **Base** que são calculados de maneira equivalente a expressão:

$$\text{base} + \text{index} * \text{scale}$$

Onde o campo **scale** são os 2 primeiros bits, onde seu valor numérico é equivalente aos seguintes fatores de escala:

- 00 - Não multiplica o **index**
- 01 - Multiplica o **index** por 2
- 10 - Multiplica o **index** por 4
- 11 - Multiplica o **index** por 8

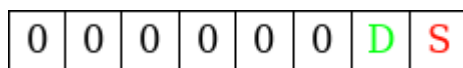
Já os campos **index** e **base** contém 3 bits cada e os mesmos armazenam o [código dos registradores](#) que serão usados. Os bits dos campos no byte seguem a ordem que o próprio nome sugere. Como em: SSIIIBBB.

Opcode

Entendendo o opcode da instrução.

Como já foi dito antes existem opcodes cujo os 3 últimos bits são usados para identificar o registrador usado na instrução. Opcodes nesse estilo de codificação são usados para instruções que só precisam usar um registrador. Por exemplo `mov eax,` `123` cujo o opcode é `B8`.

Já em instruções que usam o [byte ModR/M](#) os dois bits menos significativos do opcode tem um significado especial, que são chamados de bit **D** (*direction bit*) e **S** (*size bit*). Conforme ilustração:



Representação dos bits de um opcode.

BIT D

A função do bit **D** é indicar a direção para onde a operação está sendo executada. Se do REG para o R/M ou vice-versa. Repare nas instruções abaixo e seus respectivos opcodes:

```
mov eax, [ebx] ; 8B03
mov [ebx], eax ; 8903
```

Convertendo os opcodes `8B` e `89` para binário dá para notar um fato interessante:

```
8B -> 10001011
89 -> 10001001
```

A única diferença entre os opcodes é que em um o bit **D** está ligado e no outro não. Quando o bit **D** está ligado o campo REG é usado como operando destino e o campo R/M usado como fonte. E quando ele está desligado é o inverso: o campo R/M é o destino e o REG é o fonte. Obviamente o mesmo também se aplica se o R/M também for um registrador.

Por exemplo a instrução `xor eax, eax` pode ser escrita em código de máquina como `31 C0` ou `33 C0`. Como no campo REG e no campo R/M são os mesmos registradores não faz diferença qual é o fonte e qual é o destino, a operação executada será a mesma. Usando um *disassembler* como o **ndisasm** dá para notar isso:

```
felipe@silva-lenovo:~$ echo -ne "\x31\xC0\x33\xC0" > tst
felipe@silva-lenovo:~$ ndisasm -b32 tst
00000000  31C0                xor eax,eax
00000002  33C0                xor eax,eax
```

BIT S

O bit **S** é usado para definir o tamanho do operando, onde:

- `0` -> Indica que o operando é de 8-bit
- `1` -> Indica que o operando é do tamanho do **operand-size**.

Repare por exemplo a instrução `30 C0`:

```
felipe@silva-lenovo:~$ echo -ne "\x30\xC0" > tst
felipe@silva-lenovo:~$ ndisasm -b32 tst
00000000  30C0                xor al,al
```

Onde `31 C0` (com o bit **S** ligado) usa o operando de 32-bit EAX. Mas `30 C0` usa o operando de 8-bit AL.

Repare também no seguinte caso:

```
felipe@silva-lenovo:~$ echo -ne "\x66\x30\xC0\x66\x31\xC0" > tst
felipe@silva-lenovo:~$ ndisasm -b32 tst
00000000  6630C0              o16 xor al,al
00000003  6631C0              xor ax,ax
```

Veja que ao usar o prefixo `66` (*operand-size override*) em `31 C0` o registrador AX é utilizado. Mas esse prefixo é ignorado em instruções cujo o bit **S** esteja desligado. Por isso o **ndisasm** faz o *disassembly* da instrução ainda como `xor al, al`. Embora ele adicione um `o16` ali para denotar o uso (inútil) do prefixo.

Prefixo REX

Entendendo o prefixo REX no x86-64.

Como eu mencionei antes esse prefixo só existe no modo de 64-bit e ele é necessário para usar operandos de 64-bit. Esse prefixo não é um byte específico mas sim todos os bytes entre `40` e `4F`. Isso porque os últimos 4 bits do prefixo são campos distintos, mas os 4 bits mais significativos do prefixo REX sempre tem o valor fixo de `0100`.

Observe as figuras tiradas dos manuais da Intel:

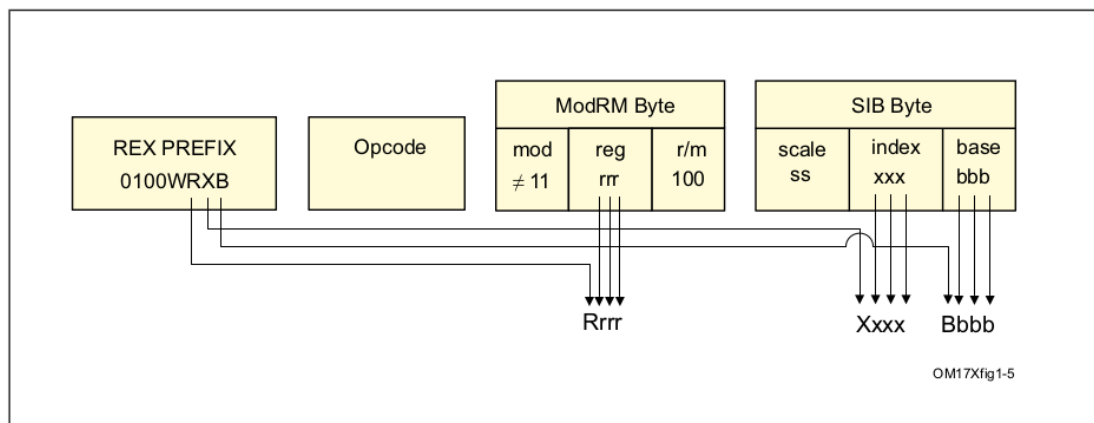


Figure 2-6. Memory Addressing With a SIB Byte

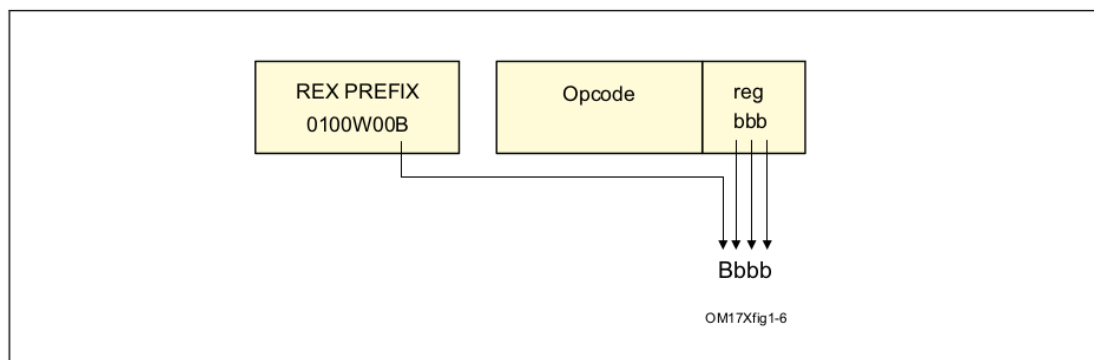


Figure 2-7. Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used

Em modo de 16-bit e 32-bit há 8 registradores de propósito geral, mas em 64-bit há 16 registradores de propósito geral. Como eu mencionei antes os campos que especificam os registradores por códigos contém somente 3 bits de tamanho, daí só é possível especificar 8 registradores distintos.

Mas alguns bits do prefixo REX são usados para estender os tamanhos desses campos em 1 bit, assim permitindo especificar até 16 registradores distintos ou 16 modos de endereçamento distintos. Cada bit do prefixo REX é identificado por uma letra e é comumente referido como no formato `REX.B` que seria o bit `B` (o menos significativo) do prefixo.

REX.B (bit 0)

Em instruções cujo a codificação do registrador faz parte do opcode, ele é usado para estender o campo de registrador. Onde ele se torna o bit mais significativo do valor.

Em instruções com ModR/M (sem SIB) ele estende o campo R/M como o bit mais significativo.

Em instruções com SIB ele estende o campo **Base** como o bit mais significativo.

REX.X (bit 1)

Estende o campo **Index** do SIB como o bit mais significativo.

REX.R (bit 2)

Estende o campo REG do byte ModR/M como o bit mais significativo.

REX.W (bit 3)

Se ligado a instrução usa operandos de 64-bit, onde por padrão os operandos são de 32-bit.

Codificação dos registradores

Entendendo a codificação dos registradores em 16-bit, 32-bit e 64-bit

Em modo de 16-bit e 32-bit cada registrador é identificado usando um número de 3 bits, permitindo assim identificar uma variação de 8 registradores diferentes. Porém vários registradores compartilham do mesmo código, e qual especificamente será usado varia de acordo com a instrução sendo utilizada e o tamanho do operando.

Por exemplo [instruções da FPU](#) irão sempre usar algum registrador ST0~ST7, então o código em uma instrução da FPU será usado para identificar algum deles.

Como por exemplo a instrução `fld st3` que em código de máquina fica `D9 C3`, onde `C3` é o ModR/M:

```
MOD -> 11
REG -> 000
R/M -> 011
```

Repare que essa instrução usa o campo `REG` como extensão do opcode e o R/M é usado para especificar o operando. Não coincidentemente o código 3 (`0b011`) é usado para identificar o registrador ST3.

Já instruções que usam [registradores de propósito geral](#), qual especificamente será usado depende do tamanho do operando na instrução (veja [Atributos e prefixos](#)).

Por exemplo as seguintes instruções compiladas em modo de 64-bit:

<code>mov eax, 0x11223344</code>	<code>b8 44 33 22 11</code>
<code>mov ax, 0x1122</code>	<code>66 b8 22 11</code>
<code>mov al, 0x11</code>	<code>b0 11</code>

Se convertermos esses opcodes em binário teremos o seguinte:

```
B8 -> 10111000
B0 -> 10110000
```

Esses dois opcodes usam os 3 últimos bits para identificar o registrador. Veja que o mesmo código `000` acabou sendo usado para identificar EAX, AX e AL.

Isso porque na primeira instrução o atributo **operand-size** padrão de 32-bit foi usado, então o registrador EAX é usado na instrução. Já na segunda o prefixo **operand-size override** (byte `66`) foi usado, assim o **operand-size** era de 16-bit e portanto o registrador AX é usado.

Já a última instrução é exclusivamente usada para operandos de 8-bit, e portanto o registrador AL é usado.

Tabela de códigos

Como já foi explicado no tópico que fala sobre o [prefixo REX](#), esse prefixo estende os campos usados em ModR/M, SIB e o campo `REG` do opcode em 1 bit. Daí assim o código usado para identificar o registrador, em modo de 64-bit, tem 4 bits de tamanho.

A tabela abaixo lista os códigos usados para identificar os registradores. Lembrando que o bit mais significativo indica um dos bits do REX ligado, ou seja, só é utilizado em modo de 64-bit.

Código	Registrador
<code>0000</code>	AL/AX/EAX/RAX/ST0/MM0/XMM0
<code>0001</code>	CL/CX/ECX/RCX/ST1/MM1/XMM1
<code>0010</code>	DL/DX/EDX/RDX/ST2/MM2/XMM2
<code>0011</code>	BL/BX/EBX/RBX/ST3/MM3/XMM3
<code>0100</code>	AH/SP/ESP/RSP/ST4/MM4/XMM4
<code>0101</code>	CH/BP/EBP/RBP/ST5/MM5/XMM5
<code>0110</code>	DH/SI/ESI/RSI/ST6/MM6/XMM6
<code>0111</code>	BH/DI/EDI/RDI/ST7/MM7/XMM7
<code>1000</code>	R8B/R8W/R8D/R8/ST0/MM0/XMM8

1001	R9B/R9W/R9D/R9/ST1/MM1/XMM9
1010	R10B/R10W/R10D/R10/ST2/MM2/XMM10
1011	R11B/R11W/R11D/R11/ST3/MM3/XMM11
1100	R12B/R12W/R12D/R12/ST4/MM4/XMM12
1101	R13B/R13W/R13D/R13/ST5/MM5/XMM13
1110	R14B/R14W/R14D/R14/ST6/MM6/XMM14
1111	R15B/R15W/R15D/R15/ST7/MM7/XMM15