Introdução ao Assembly x86-64

Andrei de Araújo Formiga

12 de dezembro de 2024

Os computadores que usamos são muito versáteis e utilizados em um grande número de aplicações, em vários domínios. A chave para essa versatilidade é que os computadores são *programáveis*, ou seja, ao invés de seguir apenas um conjunto pré-determinado de funções (determinada pelo projetista do hardware), os computadores podem ser programados para realizarem diferentes tarefas, todas tendo como base um conjunto relativamente simples de operações realizadas pelos processadores.

O conjunto de operações simples que podem ser executadas diretamente pelo processador é chamada de *linguagem de máquina* do processador. Um programa em linguagem de máquina é representado como uma sequência de bytes na memória, com cada operação sendo associada a um código. Por exemplo, a instrução ADD pode (dependendo da situação) ser codificada com o código 80 em hexadecimal (128 em decimal). Dizemos que 80 é o código da operação ou *opcode* da instrução ADD. Assim, quando o processador está executando código e encontra um byte de valor 80, ele efetua uma operação de adição. O que deve ser adicionado (os operandos) são codificados após o *opcode*.

Para escrever um programa em linguagem de máquina contendo uma operação ADD, o programador deve consultar alguma referência para determinar o *opcode* da operação e também como codificar os seus operandos.

Obviamente, esta é uma forma muito primitiva de programar, embora tenha sido usada em alguns momentos. Rapidamente os programadores pensaram em criar uma forma mais amigável de criar programas, automatizando parte do processo. Dessa ideia surgiram as linguagens de montagem ou *assembly*.

1 Linguagem Assembly

Uma linguagem *assembly* (ou linguagem de montagem) define uma representação textual para um programa em código de máquina. Ao invés de escrever os bytes 0x80 0x10 0x00 para uma operação de soma, o programador pode escrever:

```
ADD $16, %AL
```

Depois de escrever seu programa em um arquivo de texto, o programador vai usar um *assembler* (ou montador) para montar o programa em linguagem de máquina, que pode ser executado.

A sintaxe da linguagem assembly utilizada vai variar de acordo com o conjunto de instruções (*instruction set*) do processador e também de acordo com o montador específico utilizado. Por exemplo, a instrução ADD vista acima está de acordo com a sintaxe padrão do GNU Assembler (gas) para a arquitetura x86-64. Se fôssemos usar o NASM ou MASM a mesma instrução teria uma sintaxe ligeiramente diferente:

```
ADD AL, 16
```

2 Modelo de Execução

Um processador executa diretamente um conjunto de instruções simples, mas nada funciona sem a existência de alguma memória para guardar dados (de entrada, de saída, ou intermediários) e mesmo o código do programa que deve ser executado. (O fato do processador poder executar programas arbitrários que estejam na memória é o que o torna programável).

De forma simplificada, o processador tem acesso a dois tipos de memória: os *registradores*, que são uma memória interna ao processador, e a memória externa. Os registradores possuem espaço limitado mas são muito mais rápidos de acessar que a memória externa (por exemplo, somar dois números em registradores pode ser centenas de vezes mais rápido que somar dois números na memória). A memória externa leva mais tempo para acessar mas tem muito mais capacidade. Outro aspecto é que, como os registradores estão conectados aos componentes da CPU, é muitas vezes necessário usar pelo menos um registrador na maioria das instruções. Por exemplo, não é possível, na maioria das arquiteturas, somar dois números na memória diretamente; pelo menos um deles deve ser copiado para um registrador.

3 Registradores

Os registradores são divididos em registradores de uso geral e registradores especiais. Na arquitetura x86-64 existem 16 registradores de propósito geral, separado em alguns grupos:

- RAX, RBX, RCX e RDX
- RSI e RDI
- RBP e RSP
- Registradores numerados de R8 até R15

A nomenclatura dos registradores é inconsistente por uma questão histórica. Também por questões históricas, algumas instruções podem restringir quais registradores podem ser utilizados, ou mesmo utilizar alguns registradores fixos de forma implícita (veremos exemplos disso nas instruções de multiplicação e divisão com sinal).

Entre os registradores especiais, apenas dois são importantes para nossos objetivos agora: RFLAGS agrupa uma série de *flags* binárias que refletem o resultado da última operação aritmética ou lógica efetuada pelo processador. Essas *flags* são usadas para implementar condicionais.

O outro registrador especial é o RIP, o *Instruction Pointer*. O conteúdo deste registrador é o endereço da próxima instrução que deve ser executada pelo processador. Normalmente não acessamos esse registrador diretamente, apenas através das instruções de salto.

4 Instruções

Os quatro tipos mais importantes de instruções executadas pelo processador são:

1. Operações aritméticas e lógicas 2. Cópia de dados 3. Verificação de condições 4. Saltos (mudanças na sequência de execução)

5 Soma (ADD)

Os dois formatos que vamos usar inicialmente são soma de dois registradores e soma de um valor imediato com um registrador.

```
add reg, reg
add imm, reg
```

Nas duas formas, o valor do primeiro operando é somado ao valor do segundo operando e armazenado no registrador do segundo operando. O operando imediato está limitado a constantes de até 32 bits.

6 Subtração (SUB)

As instruções de subtração funcionam de maneira similar às instruções de adição. Os dois formatos que usaremos são os mesmos do ADD:

```
sub reg, reg
sub imm, reg
```

Nas duas formas, a operação realizada é calcular o segundo operando menos o primeiro, e armazenar o resultado no segundo. O operando imediato está limitado a constantes de até 32 bits.

7 Multiplicação (MUL, IMUL)

Vamos usar apenas uma forma das operações MUL e IMUL, cujo operando deve ser um registrador de 64-bits.

```
mul reg
imul reg
```

MUL e IMUL realizam a multiplicação de RAX pelo operando, colocando o resultado em RDX:RAX (os 64 bits mais altos em RDX, e os 64 bits mais baixos em RAX).

A diferença entre MUL e IMUL é que MUL realiza uma multiplicação sem sinal, enquanto IMUL realiza uma multiplicação que leva em consideração o sinal dos operandos.

8 Divisão (DIV, IDIV)

DIV e IDIV funcionam de forma similar a MUL e IMUL. Também usaremos apenas uma forma de cada instrução, que tem um registrador como operando.

```
div reg
idiv reg
```

DIV e IDIV dividem o número de 128 bits em RDX:RAX pelo operando, armazenando o quociente inteiro da divisão em RAX e o resto em RDX. DIV faz uma operação sem sinal, enquanto IDIV divide levando em consideração o sinal.

9 Seções

Um arquivo objeto no formato ELF usado pelo Linux é dividido em seções. Três seções são obrigatórias em todo arquivo objeto:

- TEXT é a seção para código
- DATA é a seção para dados estáticos inicializados
- BSS é a seção para dados estáticos não-inicializados

Todo *assembler* aceita um conjunto de diretivas que não geram código, apenas estabelecem alguma informação sobre o arquivo objeto. Um exemplo são as diretivas para especificar seções. No gas, a diretiva é `.section`.

10 Hello, World

```
.section .data
hello:
.ascii "Hello, World!\n"

.section .text
.globl _start

_start:
mov $1, %rax # sys_write
mov $1, %rdi # stdout
mov $hello, %rsi  # endereco do buffer
mov $14, %rdx           # numero de bytes
syscall

mov $60, %rax  # sys_exit
xor %rdi, %rdi  # codigo de saida
syscall
```

O código *assembly* nesse exemplo define duas seções: DATA e TEXT (a seção BSS será necessariamente gerada no arquivo objeto, mas não conterá nada). Na seção de dados inicializados, colocamos a string que será impressa. O rótulo `hello` apontará para o começo dessa sequência de bytes (similar a um *array* em C).

Depois é definida a seção TEXT que vai conter o código. O rótulo `_start` (o ponto de entrada do programa) é declarado como um nome global usando a diretiva `.globl`. Em seguida, o rótulo `_start` em si é definido para o início do código do programa.

Este código apenas utiliza a chamada de sistema `write` do kernel Linux para imprimir a mensagem na tela. O arquivo que receberá a escrita é a saída padrão (stdout). Após preparar os valores adequados nos registradores, a chamada de sistema é realizada com a instrução `syscall`.

Após isso, a chamada de sistema `exit` é usada para instruir o sistema operacional a terminar o programa.