Sistemas Operacionais – Prof. Rafael R. Obelheiro

Fundamentos de Programação Assembly x86-32

Prof. Rafael R. Obelheiro

Este documento foi traduzido e adaptado de (Nayuki, 2016) e (Carbonneaux, 2014), com material adicional escrito por Rafael Obelheiro.

A arquitetura x86 está no coração das CPUs de computadores pessoais e servidores Internet há mais de duas décadas. Conseguir ler e escrever código de baixo nível na linguagem Assembly é uma habilidade poderosa. Isso permite escrever código mais eficiente, usar recursos do processador que não estão disponíveis em C, e fazer engenharia reversa de código compilado.

Mas o início pode ser intimidador. Os manuais da documentação oficial da Intel têm milhares de páginas. 20 anos de evolução contínua sem perder a compatibilidade com versões anteriores levaram a um cenário com princípios de projeto conflitantes de diferentes eras, características obsoletas que ficam ocupando espaço, camadas sobre camadas de chaveamentos de modo, e uma exceção para cada regra.

Neste tutorial, irei ajudá-lo a obter um entendimento sólido da arquitetura x86 a partir de princípios básicos. O foco será mais em *construir um modelo mental claro* do que está acontecendo, em vez de esquadrinhar cada detalhe de forma precisa (o que seria uma leitura longa e tediosa). Se você quiser usar esse conhecimento, sugiro começar testando e modificando pequenos trechos de código para entender o funcionamento das instruções; um depurador é muito útil para isso (vide Seção 1). Ter uma lista das instruções da CPU à mão para referência também ajuda. Meu tutorial irá começar em território familiar e gradativamente aumentar a complexidade em pequenos passos – diferente de outros documentos que tendem a expor a informação toda de uma vez.

Os pré-requisitos para a leitura deste tutorial são saber trabalhar com números binários, experiência moderada com uma linguagem imperativa (C/C++/Java/Python/etc.), e o conceito de ponteiros de memória (C/C++). Você não precisa saber como uma CPU funciona internamente, nem ter conhecimento prévio de linguagem Assembly.

Este tutorial usa a sintaxe da AT&T em vez da sintaxe da Intel para a linguagem Assembly. Os conceitos subjacentes são os mesmos em ambos os casos, mas a notação é um pouco diferente. É possível traduzir mecanicamente de uma sintaxe para a outra, então não é preciso se preocupar muito com isso.

Sumário

1 Ferramentas e Teste **2**

2 Ambiente Básico de Execução da CPU **3**

3 Instrução de Transferência de Dados **5**

4 Instruções Aritméticas Básicas **6**

5 O Registrador de Flags e Comparações **10**

6 Endereçamento de Memória, Leitura, Escrita **11**

7 Desvios, Rótulos e Código de Máquina **14**

8 A Pilha **19**

9 Sub-rotinas **20**

10 Entrada e Saída **24**

1 Ferramentas e Teste

Ao ler este tutorial, é útil escrever e testar seus próprios programas em linguagem Assembly. Isso é bem simples de fazer no Linux (mais difícil mas ainda possível no Windows). O código abaixo é um programa Assembly que usa a biblioteca C; o nome da função (linhas 1 e 2) deve sempre ser `main`, e as duas instruções nas linhas 4 e 5 fazem o equivalente a `return 0` em C (a linha 3 é um comentário).

```
1 .globl main
2 main:
3     # return 0;
4     movl $0, %eax
5     ret
```

Para reaproveitar este código como base para seus próprios programas, basta incluir instruções entre as linhas 2 e 3.

Supondo que o programa acima esteja em um arquivo chamado `my-asm.s`, é possível gerar um executável com o comando

```
gcc -m32 -g -o my-asm my-asm.s
```

A opção `-g` indica ao GCC para incluir símbolos de depuração, e é fortemente recomendada. O programa pode ser executado com `./my-asm`.

Para experimentar com linguagem Assembly, o ideal é trabalhar com um depurador, como o GDB (*GNU Debugger*) ou uma de suas interfaces gráficas, como o DDD (*Data Display Debugger*). Essas ferramentas permitem executar o código passo a passo (instrução por instrução), examinar e modificar o conteúdo da memória, e inspecionar os registradores da CPU.

Dicas para o DDD

Dicas de depuração usando o DDD:

1. Defina um *breakpoint* (ponto de parada) no rótulo `main` (clique na linha que contém `main` e depois no ícone Break na parte superior da tela);
2. Abra uma janela com os registradores (Status → Registers);
3. Abra um *display* para observação da pilha
   - clique em Data → Memory;
   - ajuste o primeiro campo para 10;
   - escolha decimal no segundo campo;
   - escolha words(4) no terceiro campo;
   - escreva `$esp` no quarto campo;
   - clique em Display.
4. Execute o código passo a passo clicando em Run e depois em Step;
5. Para examinar uma variável, clique nela com o botão direito e selecione Print ou Display.

Dicas para o GDB

Dicas de depuração usando o GDB com a interface modo texto (TUI, *text user interface*):

1. Invoque o depurador com `gdb -tui ./prog`, onde `./prog` é o programa a ser depurado;
2. Defina um *breakpoint* (ponto de parada) no rótulo `main`, usando o comando `break main`;
3. Abra uma janela com o código fonte e outra janela com os registradores, usando os comandos `layout src` e `layout regs` (nessa ordem);
4. Abra um *display* para observação da pilha, usando o comando `display/10dw $esp`. Com isso, as 10 posições a partir do topo da pilha serão atualizadas e exibidas a cada vez que alguma parte do código for executada.
5. Inicie a execução do código usando o comando `run`. Se você definiu um *breakpoint* (por exemplo, `break main`), o código irá executar até esse ponto;
6. Para executar o código:
   - Para execução passo a passo, use o comando `step` ou `s`. Se a linha a ser executada for uma chamada de função, o GDB irá entrar na função e começar a executar o seu código, uma linha de cada vez. Para executar a função até o final e retornar ao código chamador, use o comando `finish` ou `fin`.
   - Se você não quiser entrar na função, use o comando `adv+1`. (O comando normalmente usado para executar funções sem entrar nelas é `next` ou `n`, mas ele não funciona com código Assembly.)
   - Para continuar a execução, use o comando `continue` ou `c`. O GDB irá executar o código até o final, ou até atingir o próximo *breakpoint*.
7. Para examinar uma variável `foo`, use o comando `print foo`.

Outros Recursos

Dicas adicionais de como usar GDB/DDD para depuração de código Assembly podem ser encontradas em:

- (Matloff, 2017), Seções 3.9 e 3.10
- `https://www.cs.swarthmore.edu/~newhall/cs31/resources/ia32_gdb.php`
- `http://dbp-consulting.com/tutorials/debugging/basicAsmDebuggingGDB.html`

2 Ambiente Básico de Execução da CPU

Uma CPU x86 tem oito registradores de 32 bits de propósito geral. Por razões históricas, os registradores são chamados {`eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, `edi`}. (Em outras arquiteturas eles seriam chamados simplesmente `r0`, `r1`, ..., `r7`.) Cada registrador pode armazenar qualquer valor inteiro de 32 bits. A arquitetura x86 na verdade tem mais de cem registradores, mas só cobriremos registradores específicos quando necessário.

Em uma primeira aproximação, a CPU executa uma lista de instruções sequencialmente, uma a uma, na ordem em que elas aparecem no código fonte (Figura 1). Mais tarde, nós veremos como o código pode ser executado de forma não linear, cobrindo conceitos como estruturas de decisão (if/then), laços, e chamadas de função.

Existem na realidade oito registradores de 16 bits e oito registradores de 8 bits que são subpartes dos oito registradores de 32 bits de propósito geral (Figura 2). Isso é uma herança da era das CPUs x86 de 16 bits (como os processadores 8086 e 8088), mas ainda tem algum uso ocasional no modo

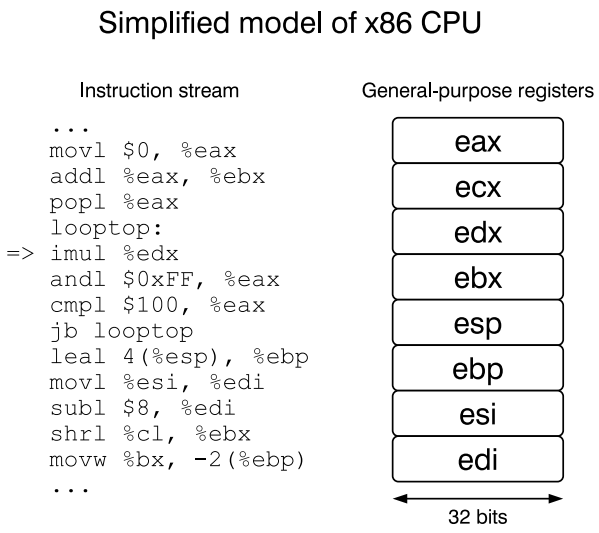

Figura 1: Modelo simplificado de uma CPU x86

de 32 bits. Os registradores de 16 bits são chamados {`ax`, `cx`, `dx`, `bx`, `sp`, `bp`, `si`, `di`}, e representam os 16 bits menos significativos dos registradores de 32 bits correspondentes {`eax`, `ecx`, `edx`, ..., `edi`} (o prefixo "e" significa "estendido"). Os registradores de 8 bits são chamados {`al`, `bl`, `cl`, `dl`, `ah`, `bh`, `ch`, `dh`}, e representam os 8 bits menos significativos (`?l`) e os 8 bits mais significativos (`?h`) dos registradores {`ax`, `cx`, `dx`, `bx`}. Sempre que o valor de um registrador de 8 ou 16 bits for modificado, os bits superiores do registrador completo de 32 bits permanecem inalterados.

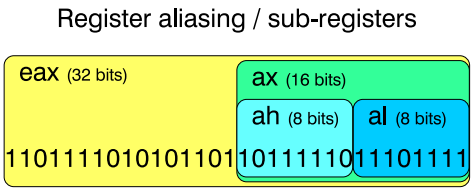

Figura 2: Sub-registradores

Alguns exemplos:

- Se `eax` = 0x12345678, `ax` = 0x5678, `ah` = 0x56 e `al` = 0x78. Em decimal, teríamos:
  - `eax` = 305.419.896
  - `ax` = 0x5678 = 22.136 ($305.419.896 \mod 2^{16} = 22.136$)
  - `ah` = 0x56 = 86 ($22.136 \div 2^8 = 86$)
  - `al` = 0x78 = 120 ($22.136 \mod 2^8 = 120$)
- Se `ebx` = 0x01020304 e `bx` recebesse 0x3456, o valor de `ebx` passaria a ser 0x01023456.

2.1 Exercícios

1. Se `ecx` = 150.000, quais os valores de `cx`, `ch` e `cl`?
2. Suponha que `edx` = 200.000 e o valor de `dh` é incrementado de 2. Qual o novo valor de `edx`?
3. Suponha que inicialmente `eax` = 250.000 e `ebx` = 700.000. Na sequência, o valor de `bx` é copiado para `ax`. Quais os valores de `eax` e `ebx` depois da cópia?

3 Instrução de Transferência de Dados

A principal instrução de transferência de dados no x86 é `mov`, que pode ser usada para transferir dados de um registrador para outro registrador, de um registrador para a memória (e vice-versa), e para carregar uma constante em um registrador ou posição de memória. (O uso da memória será visto mais adiante na Seção 6.)

A instrução `mov` *`fonte`*`,` *`destino`* copia o valor de *`fonte`* para *`destino`*. Note que, apesar do que sugere seu nome, a instrução realiza uma cópia (ela não "move" no sentido estrito da palavra), mesmo que o operando fonte não seja uma constante. Um registrador deve ser prefixado com `%` (`%eax`), e uma constante com `$` (`$255` em decimal, `$0xff` em hexadecimal). Alguns exemplos:

- `mov %eax, %ebx` copia o valor de `eax` para `ebx`;
- `mov $10, %ecx` carrega a constante 10 em `ecx`;
- `mov $0xab, %dl` carrega a constante hexa 0xAB (171 em decimal) para `dl` (isso afeta `dx` e `edx`).

3.1 Referência

Neste documento, cada seção primeiro discute as instruções de maneira mais informal, e depois é fornecida uma referência mais completa das principais instruções abordadas na seção. Estas seções de referência são mais para uso posterior, e podem ser puladas em uma primeira leitura. Elas não devem ser consideradas uma lista exaustiva de instruções x86, mas apenas um subconjunto útil. Para uma lista completa, veja uma referência do conjunto de instruções da Intel, como (Cloutier, 2019).

A seguinte notação será adotada (endereços de memória serão discutidos na Seção 6):

`reg32`	qualquer registrador de 32 bits (`%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%esp`, `%ebp`)
`reg16`	qualquer registrador de 16 bits (`%ax`, `%bx`, `%cx`, `%dx`)
`reg8`	qualquer registrador de 8 bits (`%ah`, `%bh`, `%ch`, `%dh`, `%al`, `%bl`, `%cl`, `%dl`)
`reg`	qualquer registrador (de qualquer tamanho)
`mem`	um endereço de memória (p.ex., `(%eax)`, `4+var(,1)`, `(%eax,%ebx,1)`)
`con32`	qualquer constante de 32 bits
`con16`	qualquer constante de 16 bits
`con8`	qualquer constante de 8 bits
`con`	qualquer constante (de 8, 16 ou 32 bits)

`mov` – move

A instrução `mov` copia o item de dados referenciado pelo primeiro operando (isto é, o conteúdo de um registrador, um conteúdo da memória ou um valor constante) para a posição referenciada pelo segundo operando (isto é, um registrador ou a memória). Movimentações de registrador para registrador são permitidas, mas movimentações da memória para a memória não. Nos casos em que transferências em memória são desejadas, o conteúdo da memória de origem deve ser primeiro carregado em um registrador, e daí transferido para o endereço de destino na memória.

Sintaxe

```
mov <reg>, <reg>
mov <reg>, <mem>
mov <mem>, <reg>
mov <con>, <reg>
mov <con>, <mem>
```

Exemplos

`mov %ebx, %eax`	copia o valor de `ebx` em `eax`
`movb $5, var(,1)`	armazena o valor 5 no byte na posição de memória `var`

3.2 Exercícios

4. Suponha que `eax` = 0x00001000, `ebx` = 0x20000000, `ecx` = 0x00000003 e `edx` = 0x40004000. Determine quais desses 4 registradores são afetados pelas instruções abaixo, e qual valor eles assumem após a instrução ser executada:

   (a) `mov %eax, %ebx`

   (b) `mov %dx, %cx`

   (c) `mov $0x12, %ah`

4 Instruções Aritméticas Básicas

A instruções aritméticas mais básicas do x86 operam em dois registradores de 32 bits. O primeiro operando é usando como fonte, e o segundo operando funciona tanto como fonte quanto como destino. Por exemplo: `add %ecx, %eax` – usando a notação do C, isso significa: `eax = eax + ecx`, onde `eax` e `ecx` são do tipo `uint32_t`. Muitas instruções se encaixam nesse esquema – por exemplo:

- `xor %esi, %ebp` corresponde a `ebp = ebp ^ esi;`
- `sub %edx, %ebx` corresponde a `ebx = ebx - edx;`
- `and %esp, %eax` corresponde a `eax = eax & esp`.

Algumas instruções aritméticas têm como argumento apenas um registrador, por exemplo:

- `not %eax` corresponde a `eax = ~eax;`
- `inc %ecx` corresponde a `ecx = ecx + 1`.

As instruções de deslocamento e rotação de bits usam um registrador de 32 bits para o valor a ser deslocado, e o registrador de 8 bits `cl` para o número de posições. Por exemplo, `shl %cl, %ebx` corresponde a `ebx = ebx << cl`.

Muitas instruções aritméticas admitem um *valor imediato* (uma constante) como primeiro operando. O valor imediato é fixo (não variável), e codificado diretamente na própria instrução. Valores imediatos são prefixados com `$`. Por exemplo:

- `mov $0xFF, %esi` corresponde a `esi = 0xFF;`
- `add $-2, %edi` corresponde a `edi = edi + (-2);`
- `shr $3, %edx` corresponde a `edx = edx >> 3`.

Observação

Este é um bom momento para mencionar um princípio da programação Assembly: nem toda operação desejada pode ser expressada diretamente em uma única instrução. Nas linguagens de programação típicas usadas pela maioria das pessoas, muitas construções podem ser compostas e adaptáveis a diferentes situações, e a aritmética pode ser aninhada. Em Assembly, porém, só é possível escrever o que o conjunto de instruções permite. Para ilustrar com alguns exemplos:

- Não é possível somar duas constantes imediatas, embora isso seja permitido em C. Em Assembly seria necessário calcular o valor durante a montagem, ou expressar essa soma como uma sequência de instruções.
- É possível somar dois registradores de 32 bits com uma instrução, mas não se pode somar três registradores – é preciso dividir o processo em duas instruções.

- Não é possível somar um registrador de 16 bits com um registrador de 32 bits. Seria necessário usar uma instrução para realizar uma conversão de 16 para 32 bits, e outra para efetuar a soma.
- Ao realizar um deslocamento de bits, o número de posições deve ser um valor imediato fixo ou o registrador `cl`. Não é possível usar outro registrador. Se o número de posições estiver em outro registrador, ele deve ser copiado antes para `cl`.

O resumo da ópera é que você não deve tentar adivinhar ou inventar sintaxes que não existem (como `addl %eax, %ebx, %ecx`), e também que se você não encontrar a instrução desejada na lista das instruções disponíveis, terá que implementá-la manualmente como uma sequência de instruções (e possivelmente alocar alguns registradores temporários para armazenar valores intermediários).

4.1 Referência

`add` – soma de inteiros

A instrução `add` soma seus dois operandos, armazenando o resultado no segundo operando. Observe que, embora ambos operandos possam ser registradores, no máximo um operando pode ser uma posição de memória.

Sintaxe

```
add <reg>, <reg>
add <reg>, <mem>
add <mem>, <reg>
add <con>, <reg>
add <con>, <mem>
```

Exemplos

`add $10, %eax`	adiciona 10 a `eax`
`addb $10, (%eax)`	adiciona 10 ao byte no endereço de memória armazenado em `eax`

`sub` – subtração de inteiros

A instrução `sub` calcula a diferença entre os operandos (o segundo menos o primeiro), armazenando o resultado no segundo operando. Observe que, assim como em `add`, embora ambos operandos possam ser registradores, no máximo um operando pode ser uma posição de memória.

Sintaxe

```
sub <reg>, <reg>
sub <reg>, <mem>
sub <mem>, <reg>
sub <con>, <reg>
sub <con>, <mem>
```

Exemplos

`sub %ah, %al`	`al` $\leftarrow$ `al` $-$ `ah`
`sub $216, %eax`	`eax` $\leftarrow$ `eax` $-$ 216

`inc/dec` – incremento/decremento

A instrução `inc` incrementa o seu operando em 1, e a instrução `dec` decrementa o seu operando de 1.

Sintaxe

```
inc <reg>
inc <mem>
dec <reg>
dec <mem>
```

Exemplos

`dec %eax`	decrementa `eax`
`incl var(,1)`	incrementa o inteiro de 32 bits no endereço `var`

`imul` – multiplicação de inteiros

A instrução `imul` tem dois formatos básicos: com dois operandos (primeiras duas formas mostradas abaixo) e com três operandos (últimas duas formas abaixo).

O formato de dois operandos multiplica seus dois operandos e armazena o resultado no segundo operando. O operando de resultado (ou seja, o segundo) deve ser um registrador.

O formato de três operandos multiplica o primeiro e o segundo operandos, e armazena o resultado no terceiro operando. Novamente, o operando de resultado deve ser um registrador. Além disso, o primeiro operando deve ser uma constante.

Sintaxe

```
imul <reg32>, <reg32>
imul <mem>, <reg32>
imul <con>, <reg32>, <reg32>
imul <con>, <mem>, <reg32>
```

Exemplos

`imul (%ebx), %eax`	multiplica `eax` pelo valor de 32 bits no endereço `ebx` e armazena o resultado em `eax`
`imul $25, %edi, %esi`	`esi` $\leftarrow$ `edi` $\times$ 25

`idiv` – divisão de inteiros

A instrução `idiv` divide o inteiro de 64 bits formado por `edx`:`eax` (isto é, `edx` contém os 4 bytes mais significativos e `eax` os 4 bytes menos significativos do dividendo) pelo operando especificado. O quociente é armazenado em `eax`, e o resto da divisão em `edx`. Caso o quociente não caiba em 32 bits, será gerada uma exceção.

Sintaxe

```
idiv <reg32>
idiv <mem>
```

Exemplos

`idiv %ebx`	divide `edx`:`eax` por `ebx`, armazenando o quociente em `eax` e o resto em `edx`
`idivl (%ebx)`	divide `edx`:`eax` pelo inteiro de 32 bits no endereço `ebx`, armazenando o quociente em `eax` e o resto em `edx`

`and`, `or`, `xor` – operações lógicas bit a bit

Essas instruções realizam as operações lógicas bit a bit especificadas (AND, OR, XOR) com os seus operandos, armazenando o resultado no segundo operando.

Sintaxe

```
and <reg>, <reg>
and <mem>, <reg>
and <reg>, <mem>
and <con>, <reg>
and <con>, <mem>

or <reg>, <reg>
or <mem>, <reg>
or <reg>, <mem>
```

```
or <con>, <reg>
or <con>, <mem>

xor <reg>, <reg>
xor <mem>, <reg>
xor <reg>, <mem>
xor <con>, <reg>
xor <con>, <mem>
```

Exemplos

`and $0x0f, %eax` zera todos os bits de `eax` exceto os 4 últimos
`xor %edx, %edx` zera `edx`

`not` – NOT lógico bit a bit

Inverte (nega) todos os bits do operando.

Sintaxe

```
not <reg>
not <mem>
```

Exemplo

`not %eax` inverte todos os bits de `eax`

`neg` – inverte sinal

Realiza a negação em complemento a dois (isto é, inverte o sinal) do operando.

Sintaxe

```
neg <reg>
neg <mem>
```

Exemplo

`neg %eax` `eax` $\leftarrow -$ `eax`

`shl/shr` – deslocamento à esquerda/direita

Essas instruções deslocam os bits do segundo operando à esquerda ou direita, preenchendo os bits vazios com zeros. O segundo operando pode ser deslocado até 31 posições. O número de bits do deslocamento é especificado pelo primeiro operando, que pode ser uma constante de 8 bits ou o registrador `cl`. Em ambos os casos, deslocamentos de mais de 31 posições são efetuados módulo 32.

Sintaxe

```
shl <con8>, <reg>
shl <con8>, <mem>
shl %cl, <reg>
shl %cl, <mem>

shr <con8>, <reg>
shr <con8>, <mem>
shr %cl, <reg>
shr %cl, <mem>
```

Exemplos

`shl $1, %eax` multiplica `eax` por 2 (se o bit mais significativo for 0)
`shr %cl, %ebx` armazena em `ebx` a parte inteira da divisão de `ebx` por 2^n, onde n é o valor em `cl`

4.2 Exercícios

5. Escreva instruções Assembly que realizem as operações abaixo (algumas operações podem exigir mais de uma instrução):

   (a) `eax` $\leftarrow$ `eax` $+$ `ebx`

   (b) `ebx` $\leftarrow$ `ebx` $- 7$

   (c) `eax` $\leftarrow$ `ecx` $\times 3$

   (d) `ebx` $\leftarrow$ `eax` $\times$ `ebx`

   (e) `edx` $\leftarrow$ `eax` $\times$ `ebx`

   (f) `eax` $\leftarrow$ `ebx` $+ 4 \times$ `ecx`

   (g) `eax` $\leftarrow$ `ebx` $\div 4$

6. Suponha que `eax` $= 10$, `ebx` $= 20$, `ecx` $= 30$, `edx` $= 40$, `esi` $=$ 0x12341234 e `edi` $=$ 0x0000ffff. Determine o resultado das seguintes instruções (sempre considerando os registradores com os valores acima):

   (a) `add %ebx, %edx`

   (b) `sub %eax, %ecx`

   (c) `imul $10, %eax, %ebx`

   (d) `imul %edx, %eax`

   (e) `idiv %edi`

   (f) `and %esi, %edi`

   (g) `or %edi, %esi`

   (h) `xor $0xffffffff, %edi`

   (i) `not %edi`

5 O Registrador de Flags e Comparações

Existe um registrador de 32 bits chamado `eflags` que é implicitamente lido ou escrito em muitas instruções (Figura 3). Em outras palavras, seu valor desempenha um papel na execução de instruções, mas o registrador não é mencionado explicitamente no código Assembly.

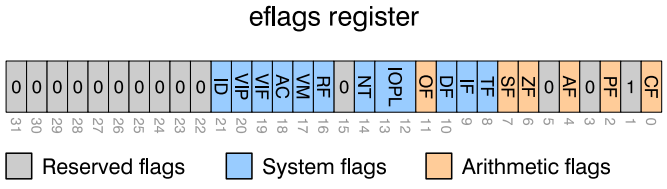

Figura 3: O registrador `eflags`

Instruções aritméticas como `addl` geralmente atualizam `eflags` com base no resultado obtido. A instruções pode ligar ou desligar *flags* como vai um (CF, *carry flag*), *overflow* (OF), sinal (SF), paridade (PF), zero (ZF), etc. Algumas instruções leem as *flags* – por exemplo, `adc` soma dois números e usa a *flag* de vai um como um terceiro operando: `adc %ebx, %eax` corresponde a `eax = eax + ebx + CF`. Algumas instruções alteram um registrador dependendo de uma *flag* – por exemplo, `setz %al` coloca o registrador de 8 bits `al` em 0 se ZF estiver desligada, e em 1 se ZF estiver ligada. Algumas instruções afetam diretamente um bit das *flags*, como `cld`, que desliga a *flag* de direção (DF).

Instruções de comparação afetam `eflags` sem modificar nenhum registrador de propósito geral. Por exemplo, `cmp %eax, %ebx` compara o valor dos dois registradores realizando uma subtração em

um local temporário e alterando as *flags* de acordo com o resultado, de modo que seja possível saber se `eax` $<$ `ebx` ou `eax == ebx` ou `eax` $>$ `ebx`, considerando números com ou sem sinal. De forma parecida, `test %eax, %ebx` computa `eax & ebx` em um local temporário e ajusta as *flags* de acordo com o resultado. Na maioria das vezes, a instrução após uma comparação é um desvio condicional (discutido na Seção 7).

Até aqui, vimos que algumas *flags* estão relacionadas a operações aritméticas. Outras *flags* dizem respeito ao comportamento da CPU – se interrupções de *hardware* devem ou não ser recebidas, se o modo 8086 virtual está ativo, e outras questões de gerenciamento do sistema que são de interesse do sistema operacional, e não de aplicações. Na maior parte, o registrador `eflags` pode ser praticamente ignorado. As *flags* de sistema podem ser ignoradas, e as *flags* aritméticas podem ser esquecidas exceto para comparações e operações aritméticas que podem resultar em *overflow*.

6 Endereçamento de Memória, Leitura, Escrita

A CPU por si só não é um computador muito útil. Ter apenas 8 registradores de dados limita bastante as computações que podem ser realizadas porque não é possível armazenar muita informação. A memória RAM é usada para estender a CPU com uma grande memória de sistema. Basicamente, a RAM é um enorme vetor de bytes – por exemplo, 128 MiB de RAM são 134.217.728 bytes que podem ser usados para armazenar dados quaisquer (Figura 4).

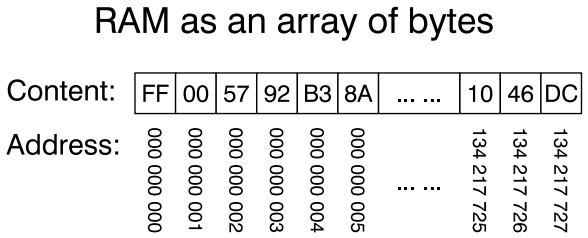

Figura 4: Memória RAM como um vetor de bytes

Quando o dado armazenado é maior do que um byte, ele é representado na notação *little endian*. Por exemplo, se um registrador de 32 bits contiver o valor 0xDEADBEEF e esse registrador for armazenado na memória a partir do endereço 10, então o byte 0xEF vai no endereço 10, 0xBE vai no endereço 11, 0xAD vai no endereço 12 e 0xDE vai no endereço 13 (Figura 5). Quando dados são lidos da memória, a mesma regra é aplicada – os bytes nos endereços mais baixos de memória são carregados nas partes mais baixas de um registrador.

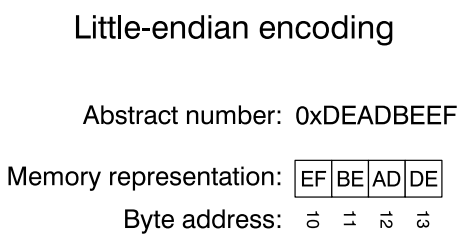

Figura 5: Ordenação *little endian*

É desnecessário dizer que a CPU tem instruções para ler e escrever da memória. Especificamente, é possível ler ou escrever um ou mais bytes de/em qualquer endereço de memória. A operação mais simples envolvendo a memória é ler ou escrever um único byte:

- `mov (%ecx), %al` corresponde a `al = *ecx` (o byte no endereço de memória `ecx` é carregado no registrador de 8 bits `al`);
- `mov %bl, (%edx)` corresponde a `*edx = bl` (o registrador `bl` é armazenado no byte no endereço de memória `edx`);
- O código C equivalente considera que `al` e `bl` são do tipo `uint8_t`, e `ecx` e `edx` estão sendo convertidos de `uint32_t` para `uint8_t *`.

Muitas operações aritméticas aceitam um operando em memória (nunca dois). Por exemplo:

- `add (%ecx), %eax` corresponde a `eax = eax + (*ecx)` (32 bits são lidos da memória);
- `add %ebx, (%edx)` corresponde a `*edx = (*edx) + ebx` (32 bits são lidos e escritos na memória).

6.1 Usando Variáveis em Memória

Assim como em linguagens de programação de alto nível, em Assembly podemos ter variáveis globais e locais. Você pode declarar regiões de dados estáticos (análogas a variáveis globais) no Assembly x86 usando diretivas especiais de montagem para esse fim. Declarações de dados devem ser precedidas pela diretiva `.data`. Após essa diretiva, as diretivas `.byte`, `.short` e `.long` podem ser usadas para declarar posições de dados com 1, 2 e 4 bytes, respectivamente. As posições declaradas podem ser rotuladas com nomes para referência posterior – isso é similar a declarar variáveis por nome, mas obedece a alguma regras de mais baixo nível. Por exemplo, posições declaradas em sequência serão localizadas lado a lado na memória.

Exemplos de declaração:

```
.data
var:
  .byte 64 /* declara um byte, rotulado como 'var', contendo o valor 64 */
  .byte 10 /* declara um byte não rotulado contendo 10. Seu endereço será var+1 */
x:
  .short 42 /* declara um valor de 2 bytes, chamado 'x', contendo 42 */
y:
  .long 30000 /* declara um valor de 4 bytes, chamado 'y', contendo 30000 */
```

Diferente de linguagens de alto nível em que vetores podem ter várias dimensões e são acessados por índices, vetores em Assembly x86 são apenas um número de células contíguas na memória. Um vetor pode ser declarado apenas listando os valores, como no primeiro exemplo abaixo. Vetores de bytes também podem ser declarados como *strings* usando a diretiva `.string` (neste caso, o byte zero de terminação da *string* é automaticamente incluído pelo montador). A diretiva `.zero` pode ser usada para preencher uma região de memória com zeros.

Alguns exemplos:

```
s:
  .long 1, 2, 3 /* um vetor de valores de 4 bytes contendo {1, 2, 3}.
barr:
  .zero 10 /* uma sequencia de 10 bytes contendo zero */
str:
  .string "hello" /* uma sequencia de 6 bytes contendo "hello" terminada por zero */
```

Na primeira declaração, o valor 1 ocupa a posição `s` na memória, o valor 2 a posição `s+4`, e o valor 3 a posição `s+8`.

6.2 Modos de Endereçamento

Quando se escreve código que contém laços, geralmente um registrador armazena o endereço base de um vetor e outro registrador armazena o índice corrente que está sendo processado. Embora

seja possível calcular manualmente o endereço do elemento sendo processado, a arquitetura x86 oferece uma solução mais elegante – existem modos de endereçamento de memória que permitem que certos registradores sejam somados e multiplicados para obter o endereço. Provavelmente é mais fácil dar exemplos do que explicar:

- `mov (%eax,%ecx), %bh` corresponde a `bh = *(eax + ecx);`
- `mov -10(%eax,%ecx,4), %bh` corresponde a `bh = *(eax + (ecx * 4) - 10)`.

O formato de um endereço é `desloc(base, indice, escala)`, onde `desloc` é uma constante inteira (que pode ser positiva, negativa ou zero), `base` e `indice` são registradores de 32 bits (com algumas combinações proibidas), e `escala` pode ser 1, 2, 4 ou 8. Por exemplo, se um vetor armazena inteiros de 64 bits, a `escala` seria 8 porque cada elemento ocupa 8 bytes.

Os modos de endereçamento de memória são válidos em qualquer lugar que um operando em memória puder ser usado. Assim, se é possível escrever `subl %eax, (%eax)`, é igualmente possível escrever `subl %eax, (%eax,%eax,2)` se a indexação for necessária. Note também que o endereço sendo calculado é um valor temporário que não é salvo em nenhum registrador. Isso é bom porque se você quisesse calcular o endereço explicitamente, você teria que alocar um registrador para isso, e com 8 registradores de propósito geral a coisa fica um pouco apertada quando você precisa armazenar outras variáveis.

Existe uma instrução especial que usa endereçamento de memória mas que não acessa realmente a memória. A instrução `lea` (*load effective address*) calcula o endereço final de memória de acordo com o modo de endereçamento, e armazena o resultado em um registrador. Por exemplo, `lea 5(%eax,%ebx,8), %ecx` corresponde a `ecx = eax + ebx*8 + 5`. Note que essa é uma operação aritmética e não envolve um acesso ao endereço de memória calculado.

6.3 Sufixos de Operação

Em geral, o tamanho de um item de dados em um determinado endereço de memória pode ser inferido da instrução Assembly na qual ele é referenciado. Por exemplo, em todas as instruções acima, o tamanho das regiões de memória pode ser inferido do tamanho do registrador usado como operando. Quando estávamos carregando um registrador de 32 bits, o montador podia inferir que a região de memória à qual nos referíamos possuía 4 bytes. Quando estávamos armazenando o valor de um registrador de 1 byte na memória, o montador podia inferir que queríamos que o endereço referenciasse um único byte na memória.

Entretanto, em alguns casos o tamanho da região de memória referenciada é ambíguo. Considere a instrução `mov $2, (%ebx)`. Essa instrução deve escrever o valor 2 no byte com endereço `ebx`? Talvez ela deva escrever 2 representado como um valor de 32 bits nos 4 bytes que iniciam no endereço `ebx`. Como ambas as interpretações são possíveis, o montador deve ser informado explicitamente sobre qual delas é a correta. Os sufixos de tamanho `b`, `w` e `l` servem para isso, indicando tamanhos de 1, 2 e 4 bytes, respectivamente. Por exemplo:

```
movb $2, (%ebx) /* escreve 2 (0x02) no byte no endereço ebx */
movw $2, (%ebx) /* escreve 2 (0x0002) nos 2 bytes que começam no endereço ebx */
movl $2, (%ebx) /* escreve 2 (0x00000002) nos 4 bytes que começam no endereço ebx */
```

6.4 Exercícios

7. Suponha que `eax` $= 100$, `ebx` $= 200$, `ecx` $= 300$, `edx` $= 400$, `esi` $= 500$ e `edi` $= 600$. Suponha também que o conteúdo dos endereços de memória de 0 a 1999 seja a sequência de valores [1000, 2999], ou seja, *mem[addr]=addr+1000* (para $addr \in [0, 1999]$). Determine o resultado das seguintes instruções (sempre considerando os registradores com os valores acima):
   (a) `mov (%ebx), %edx`
   (b) `add 40(%eax), %ecx`

(c) `mov %eax, -12(%ebx, %esi)`

(d) `mov 4(%eax, %ebx, 8), %edi`

8. Considere que a seção de dados de um programa Assembly tenha as seguintes definições:

```
.section .data
x:   .long 10
y:   .long 20
z:   .long 30
```

(a) Determine o resultado das seguintes instruções:

i. `mov x, %esi`

ii. `mov $y, %edi`

iii. `mov %edx, z`

iv. `mov %ecx, $x`

(b) Escreva instruções Assembly para:

i. copiar `eax` para `x`;

ii. copiar `y` para `ebx`;

iii. copiar o endereço de `z` para `ecx`.

7 Desvios, Rótulos e Código de Máquina

Cada instrução em linguagem Assembly pode ser prefixada por zero ou mais rótulos. Esses rótulos são úteis quando precisamos desviar para uma determinada instrução. Exemplos:

```
foo:             /* um rótulo */
  negl %eax      /* possui um rótulo (foo) */
  addl %eax, %eax /* zero rótulos */
bar:
baz:
  sbbl %eax, %eax /* possui dois rótulos (bar e baz) */
```

A instrução `jmp` ordena à CPU que a próxima instrução a ser executada será uma instrução rotulada, e não a instrução seguinte. Aqui está um simples laço infinito:

```
top: incl %ecx
    jmp top
```

Embora `jmp` seja incondicional, existem diversas instruções que observam o estado de `eflags` e desviam para o rótulo se a condição for satisfeita (caso contrário, a execução continua na instrução seguinte). Instruções de desvio condicional incluem: `ja` (*jump if above*, desvie se acima), `jle` (*jump if less than or equal to*, desvie se menor ou igual), `jo` (*jump if overflow*, desvie se *overflow*), `jnz` (*jump if not zero*, desvie se diferente de zero), etc. Ao todo existem 16 instruções, e algumas têm sinônimos – por exemplo, `jz` (*jump if zero*, desvie se zero) é o mesmo que `je` (*jump if equal*), `ja` (*jump if above*) é o mesmo que `jnbe` (*jump if not below or equal*). Um exemplo de uso de desvio condicional:

```
  jc skip /* se a flag de carry estiver ligada, desvie para skip */
  /* CF está desligada, então execute isto */
  notl %eax
  /* segue implicitamente para a próxima instrução */
skip:
  adcl %eax, %eax
```

Os endereços dos rótulos são fixados no código quando este é compilado mas também é possível desviar para um endereço arbitrário de memória calculado em tempo de execução. Em particular, é possível desviar para o valor de um registrador: `jmp %ecx` corresponde essencialmente a copiar o valor de `ecx` para `eip`, o registrador do ponteiro de instrução.

Agora é o momento perfeito para discutir um conceito sobre instruções e execução que foi omitido na Seção 2. Cada instrução em Assembly é traduzida em 1 a 15 bytes de código de máquina e essas instruções de máquina são combinadas para criar um arquivo executável (Figura 6). A CPU tem um registrador de 32 bits chamado `eip` (*extended instruction pointer*), que, durante a execução de um programa, armazena o endereço de memória da instrução sendo executada. Note que existem poucas formas de ler ou escrever em `eip`, e por isso ele se comporta de forma bem diferente dos 8 registradores de propósito geral principais. Sempre que uma instrução é executada, a CPU sabe quantos bytes de comprimento ela possui, e incrementa `eip` com essa quantidade para apontar para a próxima instrução.

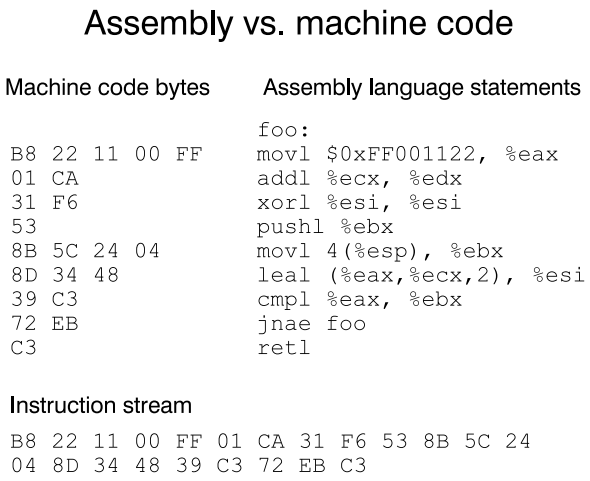

Figura 6: Código Assembly e linguagem de máquina

Enquanto estamos nesta observação sobre código de máquina, a linguagem Assembly não é efetivamente o nível mais baixo a que um programador pode chegar; o nível mais baixo é código de máquina binário puro. (Funcionários da Intel têm acesso a níveis ainda mais baixos, como microcódigo e depuração de *pipeline* – mas programadores comuns não conseguem chegar lá.) Escrever código de máquina à mão é muito desagradável (na real, o Assembly já é suficientemente desagradável), mas oferece algumas possibilidades menos importantes adicionais. Escrevendo código de máquina, é possível codificar algumas instruções de formas alternativas (por exemplo, uma sequência mais longa de bytes que produz o mesmo efeito quando executada), ou também gerar deliberadamente instruções inválidas para testar o comportamento da CPU (nem todas as CPUs lidam com erros da mesma maneira).

7.1 Referência

`jmp` – desvio (*jump*) incondicional

> Transfere o fluxo de controle do programa para a instrução no endereço indicado pelo operando.
>
> *Sintaxe*
> `jmp <rótulo>`
>
> *Exemplo*
> `jmp begin` desvie para a instrução rotulada `begin`

`j`*`condição`* – desvio condicional

Essas instruções são desvios condicionais com base no estado de um conjunto de códigos de condição armazenados no registrador `eflags`, que representa a *palavra de estado do programa* (*program status word*, PSW). O conteúdo da PSW inclui informações sobre a última operação aritmética executada. Por exemplo, um bit dessa palavra indica se o último resultado foi zero. Outro indica se o último resultado foi negativo. Com base nesses códigos de condição, vários desvios condicionais podem ser executados. Por exemplo, a instrução `jz` executa um desvio para o operando especificado se o resultado da última operação aritmética foi zero. Caso contrário, a execução prossegue na instrução seguinte.

Vários dos desvios condicionais recebem nomes que são intuitivos quando se toma por base que a última instrução executada foi uma instrução especial de comparação, `cmp` (ver abaixo). Por exemplo, desvios condicionais como `jle` e `jne` são baseados em uma comparação prévia entre os operandos desejados.

No caso de comparações em que a ordem dos operandos é relevante (por exemplo, $a > b$), os operandos devem ser trocados na instrução de comparação (`cmp b, a`) para que o desvio condicional seja interpretado corretamente. Por exemplo, se a comparação for escrita `cmp %eax, %ebx`, o desvio `jl` será verdadeiro quando `ebx` for menor que `eax`.

Outro ponto importante é as comparações do tipo *greater/less* devem ser usadas com números com sinal. Se os números forem sem sinal, devem ser usadas as comparações do tipo *above/below*.

Sintaxe

`je <rótulo>`	*Jump if Equal* – desvie se igual
`jne <rótulo>`	*Jump if Not Equal* – desvie se diferente
`jz <rótulo>`	*Jump if Zero* – desvie se o último resultado foi zero
`jg <rótulo>`	*Jump if Greater than* – desvie se maior (com sinal)
`jge <rótulo>`	*Jump if Greater than or Equal to* – desvie se maior ou igual (com sinal)
`jl <rótulo>`	*Jump if Less Than* – desvie se menor (com sinal)
`jle <rótulo>`	*Jump if Less than or Equal to* – desvie se menor ou igual (com sinal)
`ja <rótulo>`	*Jump if Above* – desvie se maior (sem sinal)
`jae <rótulo>`	*Jump if Above or Equal to* – desvie se maior ou igual (sem sinal)
`jb <rótulo>`	*Jump if Below* – desvie se menor (sem sinal)
`jbe <rótulo>`	*Jump if Below or Equal to* – desvie se menor ou igual (sem sinal)

Exemplo

```
cmp %ebx, %eax
jle done          se eax ≤ ebx, desvie para done, senão siga na próxima instrução
```

`cmp` – compare

Compara o valor dos dois operandos especificados, ajustando os códigos de condição em `eflags` de acordo com o resultado. Essa instrução é equivalente a `sub`, mas o resultado da subtração é descartado e o segundo operando não é afetado.

Sintaxe

```
cmp <reg>, <reg>
cmp <mem>, <reg>
cmp <reg>, <mem>
cmp <con>, <reg>
```

Exemplo

```
cmpl $10, (%ebx)
je loop            se os 4 bytes no endereço ebx forem iguais a 10, desvie para loop
```

7.2 Implementando Estruturas de Controle do C em Assembly

Esta seção mostra como estruturas de controle do C (`if`, `while`, `do/while` e `for`) podem ser expressas em Assembly usando desvios condicionais.

7.2.1 `if`

Código C:

```
if (x == y) {
   <codigo then>
} else {
   <codigo else>
}
<sequencia do codigo>
```

Código Assembly equivalente (assume `eax` = `x` e `ebx` = `y`):

```
   cmpl %eax, %ebx
   je cod_then

cod_else:
   <codigo else>
   jmp seq_cod

cod_then:
   <codigo then>

seq_cod:
   <sequencia do codigo>
```

Se a condição for verdadeira (`eax == ebx`), ocorre um desvio para `cod_then`; caso contrário, a execução prossegue na instrução seguinte, rotulada como `cod_else`. Observa-se que a ordem de <`codigo then`> e <`codigo else`> fica invertida em relação ao código C.

Algumas variações sao possíveis para inverter a ordem de `cod_then` e `cod_else` no código (i.e., mantê-los na mesma ordem do código C), como negar logicamente a condição (p.ex., usar `jne` para desviar para `cod_else` em vez de `je`) ou acrescentar um desvio incondicional quando a condição do `then` é falsa:

```
  cmpl %eax, %ebx
  je cod_then
  jmp cod_else

cod_then:
  <codigo then>
  jmp seq_cod

code_else:
  <codigo else>

seq_cod:
   <sequencia do codigo>
```

7.2.2 `while`

Código C:

```
while (x < y) {
   <codigo while>
}
<sequencia do codigo>
```

Código Assembly equivalente (assume `eax` = `x` e `ebx` = `y`):

```
teste_while:
   cmpl %eax, %ebx
   jle fim_while  # !(x < y) equivale a y <= x -- sinaliza fim do laco

   <codigo while>
   jmp teste_while # retorna para proxima iteracao

fim_while:
   <sequencia do codigo>
```

7.2.3 do/while

Código C:

```
do {
   <codigo do>
} while (x < y);
<sequencia do codigo>
```

Código Assembly equivalente (assume eax = x e ebx = y):

```
cod_do_wh:
   <codigo do>

   cmpl %eax, %ebx
   jg cod_do_wh   # y > x equivale a x < y -- sinaliza proxima iteracao

fim_do_wh:
   <sequencia do codigo>
```

7.2.4 for

Na linguagem C, o comando `for` é na verdade uma representação mais compacta de `while`. Examinaremos aqui o caso de laços simples em que um grupo de comandos é repetido um dado número de vezes, como no código C abaixo:

```
for (i=0; i < x; i++) {
   <codigo for>
}
<sequencia do codigo>
```

Código Assembly equivalente (assume eax = x e ecx = i):

```
   movl $0, %ecx  # inicializacao

teste_for:
   cmpl %eax, %ecx # se i >= x, termina o laco
   jge fim_for

   <codigo for>

   incl %ecx     # incrementa i e volta para o teste
   jmp teste_for

fim_for:
   <sequencia do codigo>
```

Se `i` for usado apenas como contador (ou seja, seu valor não for usado no interior do laço), pode-se usar a instrução `loop` para contar de `x` até zero:

```
  movl x(,1), %ecx # inicializacao

cod_for:
  <codigo for>
  loop cod_for    # decrementa %ecx e desvia se %ecx > 0

fim_for:
  <sequencia do codigo>
```

8 A Pilha

Conceitualmente, a pilha é uma região de memória endereçada pelo registrador `esp`. A arquitetura x86 tem diversas instruções para manipular a pilha. Embora essa funcionalidade possa ser implementada com `movl`, `addl`, etc., e com outros registradores que não `esp`, usar as instruções de pilha é mais idiomático e conciso.

No x86, a pilha cresce para baixo, dos endereços mais altos para os endereços mais baixos na memória. Por exemplo, empilhar um valor de 32 bits usando a instrução `push` significa primeiro decrementar `esp` de 4 e depois armazenar o valor a partir do endereço `esp`. Desempilhar um valor usando a instrução `pop` realiza a operação inversa – o valor no endereço `esp` é carregado (podendo ser copiado para um registrador ou descartado), e `esp` é incrementado de 4. Exemplos:

- `push $51` empilha a constante 51;
- `push %eax` empilha o conteúdo de `eax`;
- `push var` empilha o conteúdo da posição de memória `var`;
- `pop %eax` move o valor do topo da pilha para `eax`;
- `pop (%edi)` move o valor do topo da pilha para os 4 bytes no endereço `edi`.

A pilha é importante para chamadas de função. A instrução `call` é como `jmp`, exceto que o endereço da próxima instrução a ser executada é empilhado antes do desvio. Assim, é possível retornar executando a instrução `ret`, que desempilha o endereço e o carrega em `eip`. Além disso, a convenção de chamada C padrão coloca alguns ou todos os argumentos da função na pilha, como será visto na Seção 9.

Note que a memória da pilha pode ser usada para ler e escrever no registrador `eflags`, e para ler o registrador `eip`. Acessar esses dois registradores é um pouco estranho porque eles não podem ser usados em instruções `mov` ou aritméticas típicas.

8.1 Referência

`push` – empilha

> A instrução `push` insere seu operando no topo da pilha do processador na memória. Mais especificamente, `push` decrementa `esp` de 4, e a seguir copia seu operando para a posição de memória de 32 bits no endereço `(%esp)`. `esp` (o ponteiro de pilha) é decrementado por `push` porque no x86 a pilha cresce para baixo – ou seja, a pilha cresce dos endereços mais altos para os endereços mais baixos.
>
> *Sintaxe*
> `push <reg32>`
> `push <mem>`
> `push <con32>`

Exemplos

`push %eax`	coloca `eax` na pilha
`push var`	empilha os 4 bytes no endereço `var`

`pop` – desempilha

A instrução `pop` remove o item de dados de 4 bytes do topo da pilha do processador e o coloca no operando especificado (um registrador ou posição de memória). A instrução primeiro copia os 4 bytes na posição `(%esp)` para o registrador ou posição de memória especificado, e a seguir incrementa `esp` de 4.

Sintaxe

```
pop <reg32>
pop <mem>
```

Exemplos

`pop %edi`	move o elemento do topo da pilha para `edi`
`pop (%ebx)`	move o elemento do topo da pilha para os 4 bytes no endereço `ebx`

8.2 Exercícios

9. Escreva um trecho de código que troque os valores dos registradores `esi` e `edi` usando apenas a pilha (sem usar `mov`).

9 Sub-rotinas

Quando código C é compilado, ele é traduzido em código Assembly e depois em código de máquina. Uma convenção de chamada define como funções em C recebem argumentos e retornam um valor, usando a pilha e/ou registradores. A convenção de chamada aplica-se a uma função C que invoca outra função C, a código Assembly que invoca uma função C, e a uma função C que invoca uma função Assembly. (Ela não se aplica a código Assembly que invoca um trecho arbitrário de código Assembly; neste caso não há restrições.)

No x86 de 32 bits no Linux, a convenção de chamada é conhecida como convenção de chamada do C, ou *cdecl*[1]. A convenção *cdecl* utiliza a pilha do processador. Ela é baseada nas instruções `push`, `pop`, `call` e `ret`. Parâmetros de sub-rotinas são passados na pilha. Os registradores são salvos na pilha, e as variáveis locais usadas pelas sub-rotinas são alocadas na pilha. A imensa maioria das linguagens procedurais de alto nível usa convenções de chamada similares.

A convenção de chamada é dividida em dois conjuntos de regras. O primeiro conjunto de regras é usado pelo chamador (*caller*) da sub-rotina, e o segundo conjunto de regras é observado pela própria sub-rotina (*callee*). Deve-se enfatizar que erros na observância dessas regras resultam rapidamente em erros fatais de execução, uma vez que a pilha ficará em um estado inconsistente. Assim, é necessário ter bastante cuidado ao implementar a convenção de chamada em suas próprias sub-rotinas.

Uma boa maneira de visualizar o funcionamento da convenção de chamada é representar o conteúdo da pilha durante a execução de uma sub-rotina. Por exemplo, seja a função C abaixo:

```
int func(int p1, int p2, int p3) {
  int loc1, loc2, loc3;
  loc1 = 2*p1;
  loc2 = -2*p2;
  loc3 = 4*p3;
  return (loc1 + loc2 + loc3);
}
```

[1] `https://en.wikipedia.org/wiki/X86_calling_conventions#cdecl`

A Figura 7 mostra o conteúdo da pilha durante a execução de `func()`. As células mostradas na pilha são posições de memória de 32 bits, e portanto os seus endereços são espaçados de 4 em 4 bytes. O primeiro parâmetro (`p1`) fica a 8 bytes de distância do ponteiro de base (`ebp`). Acima dos parâmetros na pilha (e abaixo do ponteiro de base), a instrução `call` colocou o endereço de retorno, levando a um deslocamento adicional de 4 bytes do ponteiro de base para o primeiro parâmetro. Quando a instrução `ret` for usada para retornar da sub-rotina, ela irá desviar para o endereço de retorno armazenado na pilha.

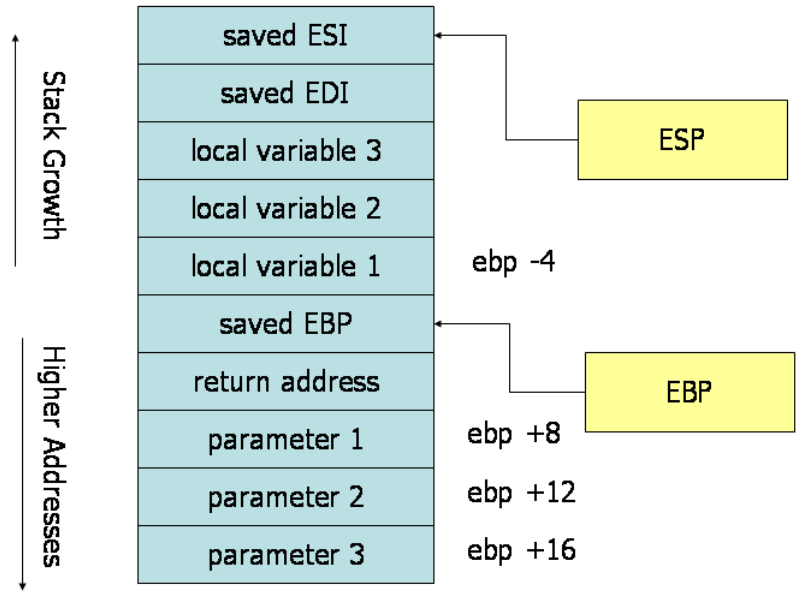

Figura 7: Pilha durante uma chamada de sub-rotina

Na convenção *cdecl*, o chamador deve:

1. Salvar os registradores `eax`, `ecx` e `edx` na pilha, se o seu conteúdo tiver de ser preservado. Esses registradores podem ser modificados livremente pela sub-rotina.
2. Empilhar os parâmetros para a sub-rotina, em ordem inversa: o último parâmetro é empilhado primeiro, e o primeiro parâmetro é empilhado por último.
3. Chamar a sub-rotina usando `call`.
4. Após o retorno da sub-rotina, recuperar o valor de retorno em `eax`.
5. Remover os parâmetros da pilha, usando `pop` ou incrementando `esp`.
6. Restaurar `eax`, `ecx` e/ou `edx` se eles foram salvos antes da chamada.

A sub-rotina é responsável por:

1. Empilhar `ebp` e copiar `esp` para `ebp` usando as instruções

```
push $ebp
mov $esp, $ebp
```

   Esta ação inicial preserva o ponteiro de base, `ebp`. O ponteiro de base é usado por convenção como um ponto de referência para encontrar parâmetros e variáveis locais na pilha. Quando uma sub-rotina está sendo executada, o ponteiro de base contém uma cópia do ponteiro de pilha de quando a sub-rotina começou a executar. Parâmetros e variáveis locais sempre estarão localizados em deslocamentos conhecidos do valor do ponteiro de base. O ponteiro de base

antigo é empilhado no início da sub-rotina para que possa ser restaurado ao final. Lembre que o chamador não espera que sub-rotina mude o valor do ponteiro de base. O ponteiro de pilha é salvo em `ebp` para obter o ponto de referência para acessar parâmetros e variáveis locais.

2. A seguir, alocar variáveis locais criando espaço para elas na pilha. Lembre-se que a pilha cresce para baixo, então, para criar espaço na pilha, o ponteiro de pilha deve ser decrementado. O valor do decremento depende do número e tamanho das variáveis locais necessárias. Por exemplo, se fossem necessários 3 inteiros (de 4 bytes) locais, o ponteiro de pilha deveria ser decrementado de 12 (`subl $12, %esp`) para criar o espaço correspondente. Do mesmo modo que os parâmetros, as variáveis locais ficam a distâncias conhecidas do ponteiro de base.

3. A seguir, os registradores `ebx`, `esi` e `edi` deverão ser salvos na pilha, caso eles venham a ser modificados na sub-rotina.

Depois que essas três ações forem realizadas, o corpo da sub-rotina pode iniciar. O encerramento da sub-rotina requer os seguintes passos:

4. Colocar o valor de retorno em `eax`.

5. Restaurar os valores antigos dos registradores salvos (`ebx`, `esi`, `edi`) que tenham sido modificados. O conteúdo dos registradores é restaurado desempilhando-os na sequência inversa à que foram empilhados.

6. Desalocar variáveis locais. A maneira óbvia de fazer isso é adicionar o valor apropriado ao ponteiro de pilha (já que o espaço foi alocado decrementando esse ponteiro). Na prática, uma forma de desalocar as variáveis que é menos suscetível a erros consiste em copiar o valor do ponteiro de base para o ponteiro de pilha (`mov %ebp, %esp`). Isso funciona porque o ponteiro de base sempre contém o valor que o ponteiro de pilha possuía imediatamente antes da alocação das variáveis locais.

7. Imediatamente antes de retornar, restaurar o ponteiro de base do chamador desempilhando `ebp`. Lembre que a primeira coisa que fizemos ao entrar na sub-rotina foi salvar o valor antigo de `ebp` na pilha.

8. Finalmente, retornar ao chamador executando a instrução `ret`. Essa instrução irá encontrar e remover o endereço de retorno apropriado da pilha.

Observe que as regras do chamado dividem-se em duas metades praticamente espelhadas. A primeira metade das regras é aplicada no início da função, e é comumente chamada de *prólogo* da função. A segunda metade é aplicada ao final da função, sendo chamada de *epílogo* da função.

O código abaixo mostra um exemplo de chamada de `x = func(10, 20, 30)`:

```
1   # salva registradores
2   pushl %eax
3   pushl %ecx
4   pushl %edx
5
6   # empilha parametros em ordem inversa
7   pushl $30
8   pushl $20
9   pushl $10
10
11  # invoca funcao
12  call func
13
14  # copia para x o valor de retorno que está em %eax
15  movl %eax, $x
16
17  # remove parametros da pilha
```

```
18      addl $12, %esp
19
20      # restaura registradores em ordem inversa
21      popl %edx
22      popl %ecx
23      popl %eax
```

A listagem abaixo, por sua vez, mostra como `func()` poderia ser implementada:

```
1   func:
2     # prólogo da função
3     push %ebp
4     mov %esp, %ebp
5
6     # aloca variáveis locais - 3 variáveis * 4 bytes = 12
7     subl $12, %esp
8
9     # não é necessário salvar registradores, função não usa %ebx, %esi nem %edi
10
11    # corpo da função
12    # copia p1 para %eax, multiplica por 2 e armazena em loc1
13    movl 8(%ebp), %eax
14    shl $1, %eax
15    movl %eax, -4(%ebp)
16
17    # copia p2 para %eax, multiplica por 2, inverte o sinal, e
18    # armazena em loc2
19    movl 12(%ebp), %eax
20    shl $1, %eax
21    neg %eax
22    movl %eax, -8(%ebp)
23
24    # copia p3 para %eax, multiplica por 4 e armazena em loc3
25    movl 16(%ebp), %eax
26    shl $2, %eax
27    movl %eax, -12(%ebp)
28
29    # loc3 já está em %eax, soma loc1 e loc2
30    addl 8(%ebp), %eax
31    addl 12(%ebp), %eax
32
33    # valor de retorno está em %eax, podemos encerrar a função
34    # não é necessário restaurar %ebx, %esi, %edi (não foram usados)
35    # epílogo da função -- libera espaço das variáveis locais e restaura %ebp
36    mov %ebp, %esp
37    pop %ebp
38
39    ret  # retorna
```

9.1 Exercícios

10. Considere o trecho de código C abaixo:

```
int f2(char *s, int x) {
  int aux;
  ...
}
```

(a) Dentro de `f2()`, quais os endereços (relativos a `ebp`) dos parâmetros `s` e `x`?

(b) Qual o endereço (relativo a `ebp`) da variável local `aux`?

(c) Quais instruções seriam usadas para fazer `aux=x` no corpo da função?

(d) Supondo que `str` seja uma *string* e `num` seja um inteiro, ambos na seção `.data`, quais instruções deveriam ser usadas para invocar `f2(str, num)`?

10 Entrada e Saída

O Assembly x86 não possui instruções para realizar operações de entrada e saída, como ler dados do teclado ou imprimir algo no vídeo. Para realizar essas tarefas, as duas opções viáveis são (i) usar as chamadas de sistema oferecidas pelo SO e (ii) usar alguma biblioteca. Uma boa saída é usar a própria biblioteca C.

O código abaixo mostra como usar as invocações de sub-rotinas introduzidas na Seção 9 para usar `printf()` e `scanf()` em um programa Assembly. A principal diferença é que as *strings* de formato são declaradas como variáveis na seção `.data`.

```
1  .data
2  x:     .long 0
3  pr1:   .string "Entre um numero: "
4  pr2:   .string "%d^2=%d\n"
5  sc:    .string "%d"
6
7  .text
8         .global main
9
10 main:
11        # printf(pr1);
12        pushl $pr1
13        call printf
14        addl $4, %esp
15
16        # scanf(sc, &x);
17        pushl $x
18        pushl $sc
19        call scanf
20        addl $8, %esp
21
22        # %eax = x*x;
23        movl x, %eax
24        imul %eax, %eax
25
26        # printf(pr2, x, %eax);
27        pushl %eax
28        pushl x
29        pushl $pr2
30        call printf
31        addl $12, %esp
32
33        # return 0;
34        movl $0, %eax
35        ret
```

Referências

Carbonneaux, Q. (2014). x86 assembly guide. `http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html`.

Cloutier, F. (2019). x86 and amd64 instruction reference. https://www.felixcloutier.com/x86/.

Matloff, N. (2017). Below C level: An introduction to computer systems. http://heather.cs.ucdavis.edu/~matloff/50/PLN/CompSystsBook.pdf.

Nayuki (2016). A fundamental introduction to x86 assembly programming. https://www.nayuki.io/page/a-fundamental-introduction-to-x86-assembly-programming.

A Respostas dos Exercícios

1. `ecx` = 150.000 = 0x000249f0
   `cx` = 0x49f0 = 18.928 `ch` = 0x49 = 73 `cl` = 0xf0 = 240

2. `edx` = 200.000 = 0x00030d40
   `dh` = 0x0d + 2 = 0x0f ⇒ `edx` = 0x00030f40 = 200.512

3. `eax` = 250.000 = 0x0003d090 (`ax` = 0xd090)
   `ebx` = 700.000 = 0x000aae60 (`bx` = 0xae60)
   Após mover `bx` para `ax`: `eax` = 0x0003ae60 = 241.248
   `ebx` = 0x000aae60 = 700.000

4.

	instrução	registrador afetado
(a)	`mov %eax, %ebx`	`ebx` = 0x00001000
(b)	`mov %dx, %cx`	`ecx` = 0x00004000
(c)	`mov $0x12, %ah`	`eax` = 0x00001200

5.

	operação	código
(a)	`eax` ← `eax` + `ebx`	`add %ebx, %eax`
(b)	`ebx` ← `ebx` − 7	`sub $7, %ebx`
(c)	`eax` ← `ecx` × 3	`imul $3, %ecx, %eax`
(d)	`ebx` ← `eax` × `ebx`	`imul %eax, %ebx`
(e)	`edx` ← `eax` × `ebx`	`imul %eax, %ebx` `mov %ebx, %edx`
(f)	`eax` ← `ebx` + 4×`ecx`	`imul $4, %ecx, %eax` `add %ebx, %eax`
(g)	`eax` ← `ebx` ÷ 4	`mov %ebx, %eax` `shr $2, %eax`

6.

	instrução	registradores afetados
(a)	`add %ebx, %edx`	`edx` = 60
(b)	`sub %eax, %ecx`	`ecx` = 20
(c)	`imul $10, %eax, %ebx`	`ebx` = 100
(d)	`imul %edx, %eax`	`eax` = 400
(e)	`idiv %edi`	`eax` = 0x00280028 = 2.621.480 `edx` = 50
(f)	`and %esi, %edi`	`edi` = 0x00001234
(g)	`or %edi, %esi`	`esi` = 0x1234ffff
(h)	`xor $0xffffffff, %edi`	`edi` = 0xffff0000
(i)	`not %edi`	`edi` = 0xffff0000

7. (a) `edx` = *mem*[200] = 1200
   (b) `ecx` = `ecx` + *mem*[140] = 300 + 1140 = 1440
   (c) *mem*[200 + 500 − 12] = `eax` ⇒ *mem*[688] = 100

(d) `edi` $= mem[100 + 8 \times 200 + 4] = mem[1704] = 2704$

8. (a) i. `esi` $= 10$
   ii. `edi` $=$ `&y` $=$ `(&x)`$+4$
   iii. `z` $=$ `edx`
   iv. $mem[10] =$ `eax`

   (b) i. `mov %eax, x`
   ii. `mov y, %ebx`
   iii. `mov $z, %ecx`

9.
```
push %esi
push %edi
pop  %esi
pop  %edi
```

10. (a) s: `ebp`$+8$ x: `ebp`$+12$

    (b) aux: `ebp`-4

    (c)
```
movl 12(%ebp), %eax
movl %eax, -4(%ebp)
```

    (d) Desconsiderando o salvamento e restauração dos registradores `eax`, `ecx` e `edx`:
```
push $num       # empilha parametros na ordem inversa
push $str
call f2
addl $8, %esp   # desempilha parametros
```