

Programação Imperativa

Conceitos Elementares

Prof. Edson Alves

Faculdade UnB Gama

Sumário

1. **Conceitos Elementares**
2. **Atribuições e Operadores**
3. **Variáveis**

Programação Imperativa

- ▶ É uma abstração de computadores reais, os quais são baseados em máquinas de Turing e na arquitetura de Von Neumann, com registradores e memória
- ▶ O conceito fundamental é o de estados modificáveis
- ▶ Variáveis e atribuições são os construtos de programação análogos aos registradores dos ábacos e dos quadrados das máquinas de Turing
- ▶ Linguagens imperativas fornecem uma série de comandos que permitem a manipulação do estado da máquina

Estados

- ▶ **Nomes** podem ser associados a um valor e depois serem associados a um outro valor distinto
- ▶ O conjunto de nomes, de valores associados a estes nomes e a localização do ponto de controle do programa constituem o **estado** do programa
- ▶ Um programa em execução gera uma sequência de estados
- ▶ As **transições** entre os estados é feita por meio de atribuições e comandos de sequenciamento
- ▶ A menos que sejam cuidadosamente escritos, programas imperativos só podem ser entendidos em termos de seu comportamento de execução
- ▶ Isto porque, durante a execução, qualquer variável pode ser referenciada, o controle pode se mover para um ponto arbitrário e qualquer valor pode ser modificado

Valores, variáveis e nomes

- ▶ Os padrões binários que o hardware reconhece são considerados, nas linguagens de programação, **valores**
- ▶ Uma unidade de armazenamento em hardware equivale a uma **variável**, no ponto de vista do programa
- ▶ O endereço da unidade de armazenamento é interpretado como um **nome** no contexto de programação
- ▶ Assim, um nome é associado tanto ao endereço (localização) de uma unidade de armazenamento quanto ao valor armazenado nesta unidade
- ▶ A localização é denominada *l-value*, e o valor em si, *r-value*
- ▶ Por exemplo, na expressão:

$$x = x + 2;$$

o x à esquerda da expressão é um *l-value* (localização), enquanto o x da direita é um *r-value* (valor)

Atribuições

- ▶ Atribuições mudam os valores de uma dada localização
- ▶ Em Assembly, atribuições são feitas por meio da instrução **MOV**:

```
MOV reg, reg
```

```
MOV reg, imm
```

- ▶ A sintaxe Assembly para instruções com dois parâmetros é a seguinte:

```
CMD dest, orig
```

onde `dest` é a localização onde será escrito o valor contido em `orig`

- ▶ Na segunda forma da instrução **MOV**, `imm` refere-se a um valor **imediato**
- ▶ Este valor corresponde a um número inteiro, em notação decimal ou hexadecimal (por meio dos sufixos `H` ou `h`)

Exemplo de atribuição

```
1 ; Exemplo de atribuição em Assembly
2
3 ; O resultado da execução deste programa pode ser visto no terminal
4 ; por meio do comando
5 ;
6 ;  $ echo $?
7 ;
8 SECTION .text
9 global _start
10
11 _start:
12     mov ecx, 14H      ; O número 20 (em forma hexadecimal) para ECX
13     mov ebx, ecx      ; Copia o valor de ECX em EBX
14     mov eax, 1        ; Move o código de SYS_EXIT (opcode 1) para EAX
15     int 80h           ; Encerra o programa com erro (código 20)
```

Adição e subtração

- ▶ O valor a ser atribuído pode ser o resultado de uma das quatro operações aritméticas
- ▶ A **adição** e a **subtração** tem a mesma sintaxe:

```
ADD regA, regB      ; regA = regA + regB  
ADD reg, imm        ; reg = reg + imm
```

```
SUB regA, regB      ; regA = regA - regB  
SUB reg, imm        ; reg = reg - imm
```

- ▶ Na primeira forma, o valor armazenado em orig é adicionado/subtraído do valor contido em dest, e o resultado é armazenado em dest
- ▶ Na segunda forma, o valor do registrador é atualizado, através da adição/subtração do valor imediato
- ▶ Ao contrário da matemática, nenhum dos dois comandos é comutativo

Exemplo de aplicação da adição e da subtração

```
1 ; Computa o número de vértices de um poliedro por meio da
2 ; fórmula de Euler:
3 ;
4 ;    $V - E + F = 2$ 
5 ;
6 SECTION text
7 global _start
8
9 _start:
10     mov edx, 6      ; Número de arestas (E) em EDX
11     mov ecx, 4      ; Número de faces (F) em ECX
12
13     mov ebx, 2      ; O número de vértices (V) ficará armazenado em EBX
14     add ebx, edx
15     sub ebx, ecx
16
17     mov eax, 1      ; Move o código de SYS_EXIT (opcode 1) para EAX
18     int 80h        ; Encerra o programa com erro V = 4 (tetraedro)
```

Multiplicação

- ▶ A multiplicação não compartilha da mesma sintaxe da adição e da subtração
- ▶ Isto porque, ao multiplicar dois números de b -bits, o resultado será um número de $2b$ -bits
- ▶ Assim, a sintaxe da multiplicação é

`MUL reg`

- ▶ Se `reg` é um registrador de 8-bits, este será multiplicado por `AL` e o produto será armazenado em `AX`
- ▶ Por exemplo,

`MUL BH`

equivale a $AX = AL \cdot BH$

Multiplicação

- ▶ Se `reg` é um registrador de 16-*bits*, este será multiplicado por `AX` e o produto será armazenado no par de registradores `DX:AX`
- ▶ `AX` conterà os *bits* menos significativos do resultado, e os mais significativos serão escritos em `DX`
- ▶ No caso de um registrador de 32-*bits*, serão necessários dois registradores de igual tamanho para armazenar o resultado
- ▶ Por exemplo,

`MUL EBX`

equivale a $\text{EDX:EAX} = \text{EAX} \cdot \text{EBX}$

- ▶ A instrução `MUL` não tem suporte para valores imediatos

Exemplo de uso da multiplicação

```
1 ; Calcula a área de um retângulo de dimensões B e H
2
3 SECTION .text
4 global _start
5
6 _start:
7     mov eax, 25      ; A base B é armazenada em EAX
8     mov ebx, 8       ; A altura H é armazenada em EBX
9     mul ebx         ; O resultado está no par EDX:EAX
10
11     mov ebx, eax     ; Copia o resultado em EBX
12     mov eax, 1       ; Código de SYS_EXIT (opcode 1)
13     int 80h         ; Encerra com erro igual a área do retângulo
```

Divisão

- ▶ A sintaxe da instrução **DIV** é semelhante à da instrução **MUL**

DIV reg

- ▶ Pela Divisão de Euclides, dados dois inteiros a, b , existem únicos q, r tais que

$$a = bq + r, \quad 0 \leq r < |b|$$

- ▶ a é o **dividendo**, b o **divisor**, q o **quociente** e r o **resto** da divisão
- ▶ A divisão por zero leva a uma exceção
- ▶ Se o quociente for maior do que o registrador reservado para armazená-lo, ocorrerá um erro de *overflow*

Divisor (reg)	Dividendo	Quociente	Resto
8-bits	AX	AL	AH
16-bits	DX:AX	AX	DX
32-bits	EDX:EAX	EAX	EDX

Exemplo de divisão

```
1 ; Calcula o índice de massa corporal: IMC = m/h^2. A unidade de medida padrão é kg/m^2,
2 ; mas no código abaixo a massa será dada em gramas e a altura em centímetros
3 SECTION .text
4 global _start
5
6 _start:
7     mov ebx, 179      ; A altura h é armazenada em EBX
8     mov eax, ebx      ; EBX = h^2
9     mul ebx
10    mov ebx, eax
11    mov eax, 805000    ; A massa m é armazenada em EAX. O zero extra é
12                      ; oriundo da conversão das unidades de medida
13
14    div ebx            ; eax = IMC
15    mov ebx, eax       ; O resultado será o código de retorno do programa
16
17    mov eax, 1         ; Move o código de SYS_EXIT (opcode 1) para EAX
18    int 80h           ; Encerra o programa com erro IMC
```

Variáveis e Assembly

- ▶ Como a programação imperativa consiste em uma série de comandos a serem executados, é preciso manter registro de tudo que já foi computado
- ▶ As variáveis mantêm o estado do programa
- ▶ Elas permitem que o controle decida o próximo estado a ser atingido, após a modificação do estado atual
- ▶ Em Assembly as variáveis correspondem aos registradores e aos endereços de memória acessíveis
- ▶ Os nomes dos registradores já estão pré-definidos
- ▶ Já as áreas de memória acessíveis são identificadores por **rótulos**, e cada *byte* desta memória deve ser acessado por meio de seu endereço
- ▶ Cada rótulo corresponde à área de memória onde está a instrução que o sucede

Estrutura dos programas em Assembly

- ▶ Embora cada arquitetura tenha uma linguagem Assembly distinta, a estrutura dos programas escritos em Assembly tende a ser a mesma
- ▶ Esta estrutura é denominada **formato de quatro campos**

```
label1:
    mn1 op1          ; comment 1
    mn2 op21,op22    ; comment 2
    mn3              ; comment 3
```

- ▶ A primeira coluna contém os **rótulos** do programa: cada rótulo é seguido de dois pontos
- ▶ A segunda coluna contém os **mnemônicos** que correspondem às instruções a serem executadas
- ▶ A terceira coluna contém os **operandos** de cada instrução, se houverem
- ▶ A quarta coluna contém os **comentários** pertinentes

Seções

- ▶ Além das quatro colunas identificadas, os códigos Assembly também são divididos em **seções**
- ▶ A seção `.text` contém as instruções do programa
- ▶ A seção `.data` contém as variáveis inicializadas
- ▶ Cada variável da seção `.data` é associada a um nome, acessível em `.text`
- ▶ As pseudo-instruções **DB**, **DW** e **DD** (*define byte*, *define word* e *define double word*) são utilizadas para declarar variáveis cuja unidade de medida é 1, 2, ou 4 *bytes*, respectivamente
- ▶ Um vetor de variáveis pode ser inicializado por meio de vírgulas

```
SECTION .data
primes    dd    2, 3, 5, 7        ; 4 primeiros primos
```

- ▶ Um vetor com 25 zeros pode ser inicializado com a sintaxe

```
SECTION .data
zeros     TIMES    25    dd    0
```

Seções

- ▶ A seção `.bss` (*Block Started by Symbol*) é utilizada para reservar memória para variáveis não inicializadas
- ▶ A pseudo-instrução **RESB** (*reserve byte*), cuja sintaxe é

```
RESB imm
```

reserva `imm` bytes de memória

- ▶ As instruções **RESW** e **RESD** tem mesma sintaxe, mas as unidades de medida correspondem às aquelas apresentadas para as instruções **DW** e **DD**, respectivamente
- ▶ A palavra-chave **global** declara os símbolos que serão exportados pelo código-objeto
- ▶ O símbolo `_start` será ponto de partida da execução de programas escritos em Assembly e montados com o NASM

Tipos de dados em Assembly

- ▶ As linguagens Assembly não mantêm registro dos tipos das variáveis do programa
- ▶ O significado dos padrões binários armazenados nos registradores ou na memória fica a cargo do programador, que deve manter a coerência dos mesmos ao longo do programa
- ▶ Por exemplo, as instruções de adição não fazem distinção se os números são sinalizados ou não
- ▶ O *assembler* utilizado pode fornecer macros para facilitar a inicialização de variáveis, como no caso dos valores imediatos e de strings
- ▶ Mesmo com estas facilidades, não há nenhuma verificação, nem em tempo de compilação, nem em tempo de execução, de tipos de dado, limites, violações, etc

Hello World!

```
1 ; A impressão no terminal é feita por meio da chamada SYS_WRITE (opcode 4):
2 ;     EDX     Tamanho, em bytes, da string a ser escrita
3 ;     ECX     Endereço da string a ser impressa
4 ;     EBX     Arquivo onde a string será impressa (STDOUT = 1)
5 SECTION .data
6 msg      db  'Hello World!', 0Ah, 0  ; msg + '\n' + zero terminador
7
8 SECTION .text
9 global _start
10
11 _start:
12     mov edx, 14      ; msg tem um total de 14 bytes
13     mov ecx, msg     ; msg contém o endereço da mensagem
14     mov ebx, 1       ; A saída é o console
15     mov eax, 4       ; Optcode de SYS_WRITE
16     int 80h         ; Realiza a chamada via interrupção
17     mov ebx, 0       ; Encerra o programa com sucesso
18     mov eax, 1
19     int 80h
```

Hello World com entrada de usuário

```
1 ; A entrada é feita por meio da chamada de sistema SYS_READ (opcode 3):
2 ;     EDX     Tamanho máximo, em bytes, a ser preenchido
3 ;     ECX     Endereço da variável a ser preenchida
4 ;     EBX     Arquivo do qual a string será carregada (STDIN = 0)
5 SECTION .data
6 msg     db  'Hello, ', 0 ; mensagem + zero terminador
7
8 SECTION .bss
9 name:   resb 255
10
11 SECTION .text
12 global _start
13
14 _start:
15     mov edx, 255      ; Tamanho máximo do buffer
16     mov ecx, name     ; Endereço de memória do buffer
17     mov ebx, 0        ; Lê a string do console
18     mov eax, 3        ; Opcode de SYS_READ
19     int 80h           ; Lê o nome do usuário
```

Hello World com entrada de usuário

```
21  mov edx, 8      ; msg tem um total de 8 bytes
22  mov ecx, msg    ; msg contém o endereço da mensagem
23  mov ebx, 1      ; A saída é o console
24  mov eax, 4      ; Optcode de SYS_WRITE
25  int 80h         ; Imprime msg, sem quebra de linha
26
27  mov edx, 255    ; Imprime todo o buffer
28  mov ecx, name   ; name contém o endereço do buffer
29  mov ebx, 1      ; A saída é o console
30  mov eax, 4      ; Optcode de SYS_WRITE
31  int 80h         ; Imprime name, conforme foi lido
32
33  mov ebx, 0      ; Encerra o programa com sucesso
34  mov eax, 1
35  int 80h
```

Variáveis e memória

- ▶ É possível manipular a memória por meio de atribuições
- ▶ Para tal, é preciso carregar a informação para um registrador, realizar o processamento desejado e depois mover o resultado para a memória
- ▶ A sintaxe para indicar um endereço de memória é $[X]$, onde X é um rótulo ou um registrador
- ▶ É possível realizar aritmética de ponteiros, através da sintaxe
$$[\text{reg} + \text{scale} * \text{reg} + \text{offset}]$$
- ▶ Os valores possíveis para o fator de escala scale são: 1, 2, 4 ou 8
- ▶ O primeiro registrador é denominado **registrador base**
- ▶ O registrador multiplicado pelo fator de escala é denominado **registrador índice**
- ▶ Tanto o registrador índice como o deslocamento (offset) podem ser omitidos

Exemplo de manipulação de variável em memória

```
1 SECTION .data
2 msg      db  'abc', 0Ah, 0
3
4 SECTION .text
5 global _start
6
7 _start:
8     mov bl,[msg+1] ; Torna maiúscula a segunda letra de msg
9     sub bl, 20h    ; bl tem 1 byte de tamanho
10    mov [msg+1], bl
11    mov edx, 5      ; msg tem um total de 5 bytes
12    mov ecx, msg    ; msg contém o endereço da mensagem
13    mov ebx, 1      ; A saída é o console
14    mov eax, 4      ; Optcode de sys_write
15    int 80h        ; Realiza a chamada via interrupção
16    mov ebx, 0      ; Encerra o programa com sucesso
17    mov eax, 1
18    int 80h
```


Referências

1. asmtutor.com. [Learn Assembly Language](#), acesso em 16/01/2020.
2. Chromium OS Docs. [Linux System Call Table](#), acesso em 30/03/2021.
3. NASM. [Site Oficial](#), acesso em 16/01/2020.
4. NASM Manual. [Pseudo-Instructions](#), acesso em 21/01/2020.
5. **NEVELN**, Bob. *Linux Assembly Language Programming*, Open Source Technology Series, Prentice-Hall, 2000.
6. **SHALOM**, Elad. *A Review of Programming Paradigms Throughtout the History – With a Suggestion Toward a Future Approach*, Amazon, 2019.
7. The Geometry Junkyard. [Twenty Proofs of Euler's Formula: \$V - E + F = 2\$](#) , acesso em 21/01/2020.