Programação Imperativa

Condicionais

Prof. Edson Alves

Faculdade UnB Gama

Sumário

1. **Saltos**

2. **Pilha**

3. **Funções**

Saltos

- As atribuições simulam as instruções de escrever ou apagar um traço nas fitas das máquinas de Turing
- Assim, as linguagens imperativas precisam também de mecanismos que permitam que o controle decida o próximo estado a ser avaliado de forma condicional, de acordo com o estado atual
- Nas linguagens Assembly isto é feito por meio de saltos condicionais
- Os saltos são instruções que desviam o controle para pontos específicos, identificados por rótulos, de acordo com o estado dos registradores de controle (*flags*)
- Como podem existir diversas combinações das (*flags*) a serem consideradas, há várias instruções de salto distintas
- Também existe uma instrução para saltos incondicionais

Salto incondicional

- A instrução `JMP` corresponde a um salto incondicional
- A sintaxe desta instrução é

```
JMP label
```

- *label* corresponde a um rótulo, e a execução do programa seguirá para a primeira instrução que segue o rótulo
- Como o salto é incondicional, a depender do posicionamento da instrução de salto e do rótulo o programa pode ficar preso em um laço infinito, jamais encerrando sua execução
- Ainda assim, saltos incondicionais são úteis, principalmente para sair de laços aninhados ou para encerrar o programa a partir de qualquer ponto

Saltos condicionais

- Um salto condicional avalia uma ou mais *flags* e, a depender dos estados delas (0 ou 1), realiza o salto ou não
- A instrução `JZ` salta para *label* se a *flag* zero for igual a 1
- As instruções `ADD` e `SUB` modificam esta *flag*, tornando igual a 1 se o resultado da operação é igual a zero, ou 0, caso contrário
- A instrução `JNZ` salta para *label* se a *flag* zero for igual a 0
- Outra *flag* que é modificada pelas instruções `ADD` e `SUB` é a de sinal, que se torna um se o resultado da operação é negativo, ou zero, caso contrário
- As instruções `JS` e `JNS` são semelhantes às instruções `JZ` e `JNZ`, porém elas avaliam a *flag* de sinal

Exemplo de saltos condicionais

```
1  ; Computa o número de raizes reais do polinômio  p(x) = ax^2 + bx + c
2  ; Como exemplo, serão utilizados os valores a = 1, b = -5 e c = 6
3  SECTION .data
4  a       dd  1               ; dd = variáveis com 4 bytes
5  b       dd  -5
6  c       dd  6
7  msg     db  '0', 0Ah, 0
8
9  SECTION .text
10 global _start
11
12 _start:
13     mov eax, 4      ; EBX = 4*a*c
14     mov ecx, [a]
15     mul ecx
16     mov ecx, [c]
17     mul ecx
18     mov ebx, eax
```

Exemplo de saltos condicionais

```
20      mov eax, [b]    ; b = -5
21      mul eax         ; EAX = b^2
22
23      sub eax, ebx    ; EAX = b^2 - 4*a*c
24
25      jz  one         ; Se EAX = 0 há apenas uma raiz
26      jns two         ; Se EAX > 0 há duas raizes reais
27
28      mov bl, '0'     ; Caso contrário não há raizes reais
29      jmp finish
30
31  one:
32      mov bl, '1'
33      jmp finish
34
35  two:
36      mov bl, '2'
37
```

Exemplo de saltos condicionais

```
38 finish:
39     mov [msg], bl   ; Atualiza a mensagem com o número de raizes
40
41     mov edx, 3      ; msg tem 3 bytes
42     mov ecx, msg    ; msg é a mensagem a ser impressa
43     mov ebx, 1      ; A saída é STDOUT
44     mov eax, 4      ; Optcode de SYS_WRITE
45     int 80h
46
47     mov ebx, 0      ; Encerra com sucesso
48     mov eax, 1
49     int 80h
```

Comparações

- Outra *flag* que é modificada pelas operações aritméticas é a de *overflow*
- As diferentes combinações das *flags* de sinal, de *overflow* e zero permite simular os operadores relacionais das linguagens de programação de alto nível
- A instrução `CMP`, cuja sintaxe é

```
CMP x, y
```

  compara o conteúdo dos registradores x e y, modificando as *flags* apropriadas
- Após esta instrução, é possível utilizar um dos comandos de saltos listados na tabela a seguir
- Observe que há um conjunto de instruções para inteiros sinalizados e outro para inteiros não sinalizados
- Veja que, se após a instrução `CMP` e antes do salto, for executada alguma instrução que modifique alguma das *flags*, o salto pode não ter o efeito esperado

Saltos associados aos operadores relacionais

Sinalizados	Não sinalizados	Efeito
JL	JB	Salta se $x < y$
JLE	JBE	Salta se $x \leq y$
JG	JA	Salta se $x > y$
JGE	JAE	Salta se $x \geq y$
JE	JE	Salta se $x = y$
JNL	JNB	Salta se $x \not< y$
JNLE	JNBE	Salta se $x \not\leq y$
JNG	JNA	Salta se $x \not> y$
JNGE	JNAE	Salta se $x \not\geq y$
JNE	JNE	Salta se $x \neq y$

Observação: L = *less*, G = *greater*, A = *above*, B = *below*, J = *jump*, E = *equal*, N = *not*

Exemplo de comparações

```
1  ; Determina se o número n é par ou ímpar
2  SECTION .data
3  n       dd  5
4  even    db  'Par', 0Ah, 0
5  odd     db  'Impar', 0Ah, 0
6
7  SECTION .text
8  global _start
9
10 _start:
11     mov eax, [n]     ; a = n
12     mov ebx, 2       ; b = 2
13     div ebx          ; r = edx, q = eax
14
15     cmp edx, 0       ; Testa se n é par (r == 0)
16     je  par
17
```

Exemplo de comparações

```
18      mov edx, 7      ; n é ímpar
19      mov ecx, odd
20      jmp finish
21
22  par:
23      mov edx, 5      ; even tem 5 bytes
24      mov ecx, even
25      jmp finish
26
27  finish:
28      mov ebx, 1      ; A saída é STDOUT
29      mov eax, 4      ; Optcode de sys_write
30      int 80h
31
32      mov ebx, 0      ; Encerra com sucesso
33      mov eax, 1
34      int 80h
```

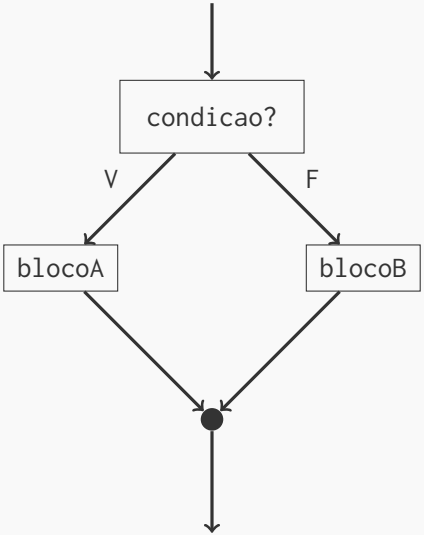
IF-ELSE

- Os saltos condicionais permitem simular estruturas de controle presentes em outras linguagens
- O construto IF-ELSE tem a seguinte sintaxe:

```
if condicao then
    blocoA
else
    blocoB
```

- Se a condicao for avaliada como verdadeira, são executados os comandos associados ao blocoA; caso contrário, são executados os comandos do blocoB
- A cláusula ELSE é opcional
- Este construto desvia a execução do programa, que a partir da condição passa a ter dois caminhos possíveis, e estes caminhos são mutuamente exclusivos
- Após o término do bloco escolhido, a execução continua do ponto que segue o último comando associado a blocoB

Visualização do construto IF-ELSE



Codificação do construto IF-ELSE em Assembly

```
; Código correspondente ao construto IF-ELSE

    cmp regA, regB      ; Esta comparação corresponde à condição
    jnz .blockA         ; Assuma 0 falso, caso contrário verdadeiro
    jmp .blockB

.blockA:
    ; Commandos associados ao blocoA
    jmp finish

.blockB:
    ; Commandos associados ao blocoB
    jmp finish

.finish:
    ; Prossegue com a execução do código
```

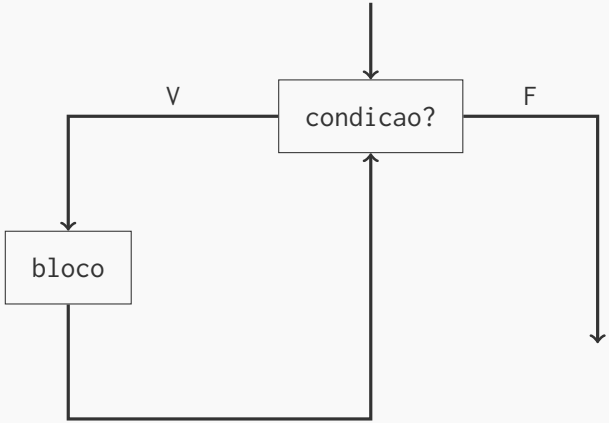
WHILE

- Outro construto que pode ser simulado com saltos condicionais é o laço `WHILE`
- A sintaxe do laço `WHILE` é

```
while condicao do
    bloco
```

- Os comandos associados ao `bloco` serão executados sempre que a `condicao` for verdadeira
- Deste modo, caso os comandos do bloco não alterem o estado do programa de modo a permitir que a condição se torne falsa, o laço se repetirá indefinidamente
- A presença de saltos dentre os comandos do bloco permite a saída prematura do bloco

Visualização do construto WHILE



Codificação do construto WHILE em Assembly

```
1  ; Código correspondente ao construto WHILE
2
3  .while:
4      cmp regA, regB      ; Esta comparação corresponde à condição
5      jz  finish          ; Assuma 0 falso, caso contrário verdadeiro
6
7      ; Commandos associados ao bloco
8
9      jmp .while          ; Retorna ao início, reavaliando a condição
10
11 .finish:
12     ; Prossegue com a execução do código
```

Exemplo de laços e condicionais

```
1  ; Determina se o número n é primo ou não
2  SECTION .data
3  yes     db  ' eh primo', 0Ah, 0
4  no      db  ' nao eh primo', 0Ah, 0
5
6  SECTION .bss
7  bf:     resb 256    ; Buffer de leitura
8
9  SECTION .text
10 global _start
11
12 _start:
13     ; Lê um número do console
14     mov edx, 256    ; Lê, no máximo, 256 caracteres
15     mov ecx, bf     ; Grava a leitura em bf
16     mov ebx, 0      ; Lê de STDIN
17     mov eax, 3      ; Optcode de SYS_READ
18     int 80h         ; EAX = digitos lidos + '\n'
```

Exemplo de laços e condicionais

```
20      dec eax          ; Desconta o '\n'
21
22      ; Imprime o número lido
23      mov edx, eax     ; Número de caracteres lidos
24      mov ecx, bf      ; Buffer de leitura
25      mov ebx, 1       ; Escreve em STDOUT
26      mov eax, 4       ; Optcode de SYS_WRITE
27      int 80h
28
29      ; Converte a string para inteiro
30      mov eax, 0       ; EAX conterá o número convertido
31      mov esi, bf      ; Buffer contendo o número como string
32      mov ebx, 10      ; Base numérica
33      mov ecx, 0       ; Próximo dígito a ser processado
34
35  to_int:
36      mov cl, [esi]
37      cmp cl, '0'      ; Se o caractere está fora da faixa [0-9] finaliza
38      jl done
```

Exemplo de laços e condicionais

```
40      cmp cl, '9'
41      jg done
42
43      sub ecx, '0'    ; Converte de ASCII para decimal
44
45      mul ebx         ; EAX = 10*EAX + ECX
46      add eax, ecx
47
48      inc esi         ; Avança o ponteiro e continua o laço
49      jmp to_int
50
51  done:
52      mov ebx, eax    ; EBX = n
53
54      ; Verifica se o número é menor que 2
55      mov ecx, 2
56      cmp ebx, ecx
57      jl not_prime
```

Exemplo de laços e condicionais

```
59      je is_prime      ; Verifica se é igual a 2
60
61      ; Verifica se é par
62      mov eax, ebx
63      div ecx
64      cmp edx, 0
65      je not_prime
66
67      ; Tenta todos os ímpares menores que a raiz quadrada
68      mov ecx, 3
69
70  next:
71      ; Checa se há ultrapassou a raiz quadrada
72      mov eax, ecx
73      mul eax
74      cmp eax, ebx
75      jg is_prime
```

Exemplo de laços e condicionais

```
77      ; Verifica se ECX divide n
78      mov eax, ebx
79      div ecx
80      cmp edx, 0
81      je not_prime
82
83      ; Tenta o próximo ímpar
84      add ecx, 2
85      jmp next
86
87  is_prime:
88      mov edx, 11      ; 'yes' tem 11 bytes
89      mov ecx, yes     ; Escreve o conteúdo de yes
90      jmp finish
91
92  not_prime:
93      mov edx, 15      ; 'no' tem 15 bytes
94      mov ecx, no      ; Escreve o conteúdo de no
95      jmp finish
```

Exemplo de laços e condicionais

```
97 finish:
98     mov ebx, 1      ; Escreve em STDOUT
99     mov eax, 4      ; Optcode de sys_write
100    int 80h
101
102    mov ebx, 0      ; Encerra com sucesso
103    mov eax, 1
104    int 80h
```

Pilha

- A pilha (*stack*) é uma porção de memória compartilhada entre o programa e o sistema operacional
- Ela pode ser usada para a comunicação entre subprogramas, armazenamento temporário e por chamadas de sistema
- Ela é uma estrutura LIFO (*last in, first out*), isto é, o último elemento que foi inserido é o primeiro a ser removido
- Como é uma área compartilhada, ela deve ser usada com bastante cuidado, sob o risco de quebra do programa
- O registrador ESP (SP – *stack pointer*) contém o endereço de memória do elemento do topo da pilha
- Este registrador pode ser lido a qualquer momento, mas deve ser manipulado com cautela

Inserção de elementos na pilha

- A instrução PUSH insere um novo elemento no topo da pilha, atualizando apropriadamente o valor de ESP

```
PUSH reg16
PUSH reg32
```

- O tamanho da memória acrescida à pilha depende do tamanho do registrador passado como parâmetro: 2 *bytes* para um registrador de 16 *bits*, 4 *bytes* para um registrador de 32 *bits*
- Esta instrução não suporta registradores de 8 *bits*
- A pilha cresce "para baixo", isto é, um PUSH com um registrador de 32 *bits* (por exemplo, EAX) equivale às instruções

```
sub esp, 4
mov [esp], eax
```

Visualização da inserção de elementos na pilha

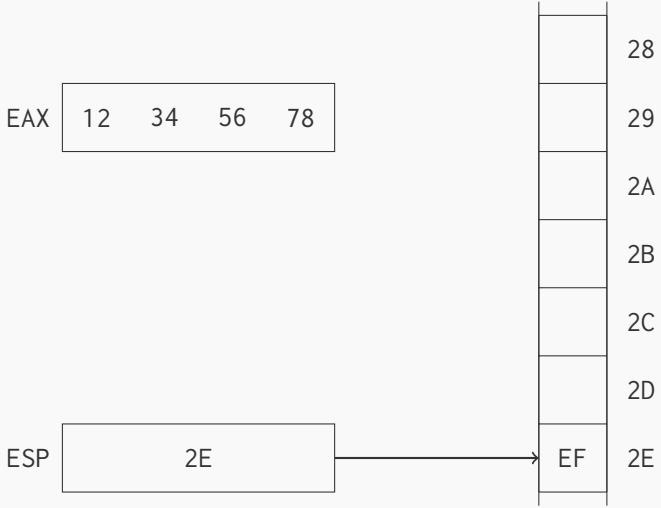


Figura: Estado do programa

Visualização da inserção de elementos na pilha

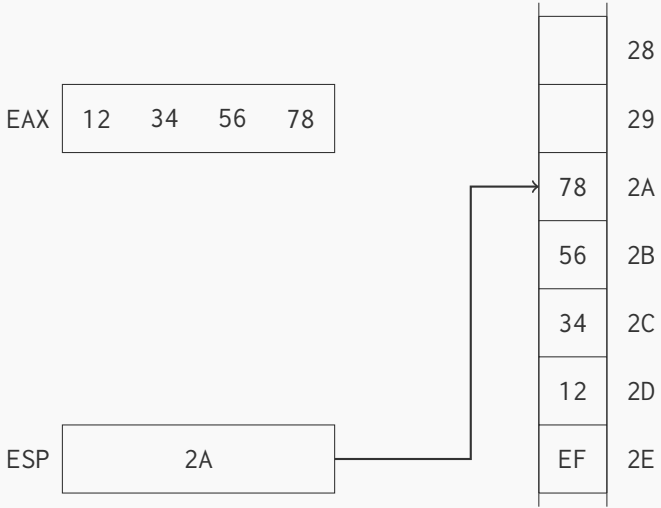


Figura: Estado do programa após PUSH EAX

Visualização da inserção de elementos na pilha

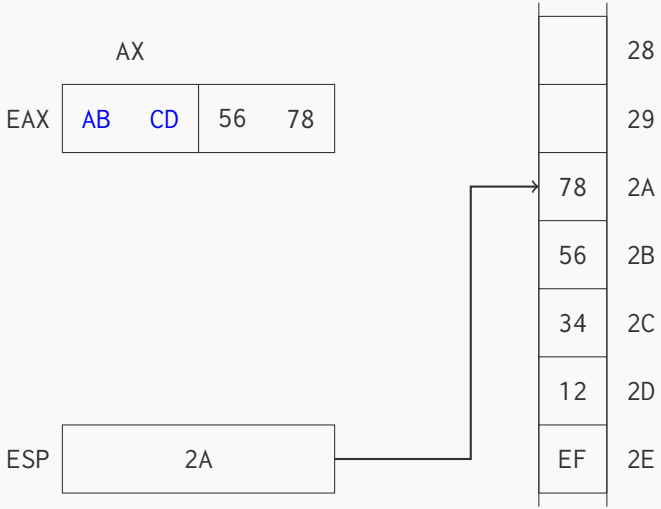


Figura: Atualização do registrador AX

Visualização da inserção de elementos na pilha

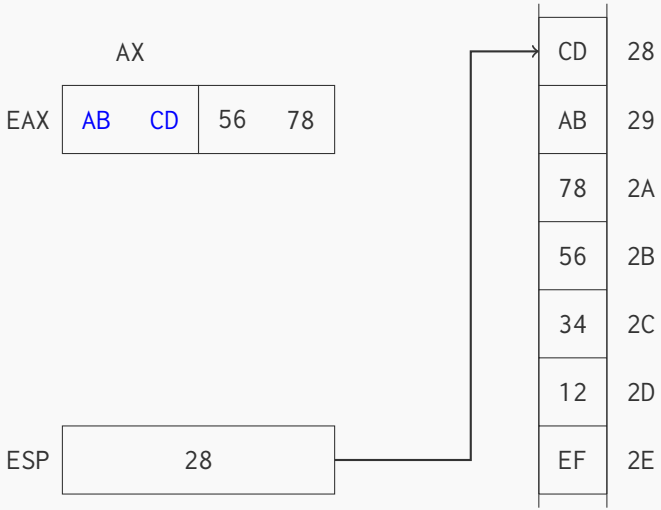


Figura: Estado do programa após PUSH AX

Remoção de elementos da pilha

- A instrução POP remove o elemento que está no topo da pilha, atualizando o valor de ESP para o novo topo

```
POP reg16
POP reg32
```

- A quantidade de *bytes* que serão efetivamente removidos depende do tamanho do registrador passado como parâmetros: 2 *bytes* para um registrador de 16 *bits*, 4 *bytes* para um registrador de 32 *bits*
- De forma semelhante à instrução PUSH, não há suporte para registradores de 8 *bits*
- Como a pilha cresce "para baixo", remover um elemento de b *bytes* equivale a somar b *bytes* em ESP
- Assim, a instrução POP AX equivale às instruções

```
mov ax, [esp]
add esp, 2
```

Visualização da remoção de elementos na pilha

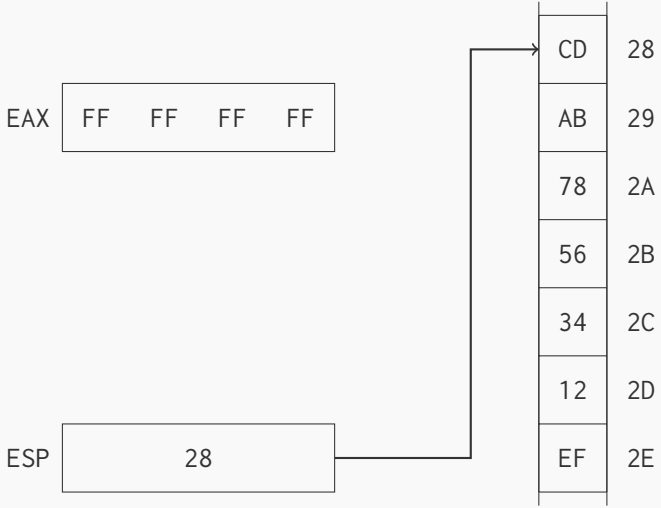


Figura: Estado do programa

Visualização da remoção de elementos na pilha

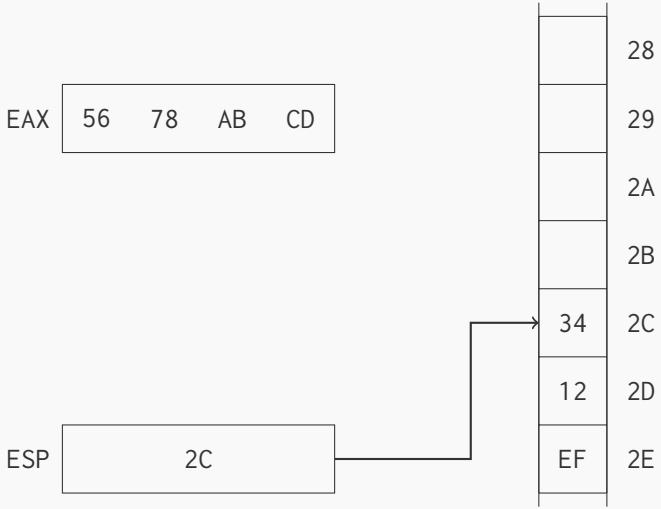


Figura: Estado do programa após a instrução POP EAX

Exemplo de uso da pilha

```
1  ; Imprime o número inteiro n, contido no registrador EAX. A rotina
2  ; que converte de inteiro para string usa a pilha para armazenar
3  ; os dígitos
4  SECTION .text
5  global _start
6
7  _start:
8      mov eax, 5291    ; n = EAX
9      mov ecx, 0       ; ECX = número de dígitos
10
11 next:
12     mov edx, 0       ; Prepara a divisão: a = EDX:EAX
13     mov ebx, 10      ; b = 10
14     div ebx          ; q = eax, r = edx
15
16     add edx, '0'     ; converte r para ASCII
17     push dx          ; insere na pilha
18     inc ecx          ; atualiza a contagem de dígitos
```

Exemplo de uso da pilha

```
20      cmp eax, 0      ; Verifica se ainda há dígitos a serem extraídos
21      jne next
22
23      ; Prepara a rotina de impressão
24      mov esi, ecx    ; ESI conterá o número de dígitos
25      mov edx, 1      ; Imprime um dígito por vez
26      mov ebx, 1      ; Imprime em STDOUT
27      mov eax, 4      ; Optocode de SYS_WRITE
28
29  print_int:
30      cmp esi, 0      ; Verifica se há dígitos a serem impressos
31      je done
32
33      mov ecx, esp    ; Aponta para o topo da pilha
34      int 80h         ; Imprime o caractere do topo
35      mov eax, 4      ; Restaura o valor de eax
36
```

Exemplo de uso da pilha

```
37      add esp, 2      ; Remove o topo da pilha
38      dec esi         ; Decrementa o contador
39      jmp print_int
40
41  done:
42      mov edx, 0Ah    ; Imprime a quebra de linha
43      push dx
44
45      mov edx, 1
46      mov ecx, esp
47      int 80h
48
49      pop dx          ; Remove a quebra de linha da pilha
50
51  exit:
52      mov ebx, 0
53      mov eax, 1
54      int 80h
```

Subrotinas

- A possibilidade de alterar qualquer registrador ou memória disponível a qualquer momento, assim como saltar arbitrariamente para qualquer posição válida, dificulta a escrita de trechos de código reutilizáveis
- Construtos de linguagens de alto nível, como subrotinas ou funções, podem ser implementados em Assembly, com o uso da pilha
- O principal uso da pilha é o de manter os valores dos registradores antes da chamada da subrotina
- Ela também pode ser utilizada tanto para armazenar os parâmetros da subrotina quanto para armazenar o valor de retorno, no caso de funções
- Além disso, ela deve armazenar o endereço de memória para o qual a subrotina deve seguir após o seu retorno

Subrotinas

- A ordem de passagem dos parâmetros e da localização do valor de retorno (pilha ou registrador) depende da convenção de cada plataforma
- Na implementação GNU da linguagem C, o retorno é feito no registrador `EAX`, e os parâmetros são passados da direita para a esquerda (assim, o primeiro parâmetro será inserido por último na pilha)
- Também a responsabilidade de remover os parâmetros da subrotina inseridos na pilha faz parte da convenção
- No caso da linguagem C, a responsabilidade é da rotina que invocou a subrotina
- As subrotinas podem ser implementadas em um arquivo separado, o qual pode ser incluído no programa por meio da diretiva `include`, precedida pelo caractere '%'

Chamada e retorno

- A instrução `CALL` é usada para invocar um subrotina

```
CALL label
```

- Esta instrução difere da instrução `JUMP` por armazenar, na pilha, o endereço de memória que marca o ponto de retorno da execução, após a conclusão da subrotina
- Para seguir este endereço ao término da subrotina, é utilizada a instrução `RET`, a qual não recebe parâmetros
- Esta instrução salta para o endereço de memória apropriado na rotina que invocou a subrotina, e remove este endereço da pilha
- As instruções de salto não devem ser utilizadas para implementar as subrotinas, pois há o risco de que fique lixo na pilha ou que esta seja corrompida por uma remoção não associada a uma inserção prévia

Exemplo do uso de subrotinas: FizzBuzz

```
1  ; Implementa o FizzBuzz em Assembly. Os parâmetros da chamada do
2  ; executável são armazenados na pilha: o topo contém o número de
3  ; argumentos passados (o nome do programa é o argumento 0)
4  SECTION .data
5  error_msg   db  'Usage: fizzbuzz n', 0Ah, 0
6  fizz        db  'Fizz', 0
7  buzz        db  'Buzz', 0
8  endl        db  0Ah, 0
9
10 %include 'subroutines.s'
11
12 SECTION .text
13 global _start
14
15 _start:
16     pop ecx             ; ECX = número de argumentos passados
17     cmp ecx, 2          ; Checa se o argumento foi passado
18     jl  .error
```

Exemplo do uso de subrotinas: FizzBuzz

```
20      mov eax, [esp + 4]  ; Ignora o nome do programa
21
22      call string_to_int  ; Converte o argumento para inteiro
23
24      mov esi, 1          ; Início da contagem
25      mov edi, eax        ; Fim da contagem
26
27  .fizzbuzz:
28      cmp esi, edi        ; Verifica se a contagem já terminou
29      jg .exit
30
31      mov eax, esi        ; Checa se é múltiplo de 15
32      mov edx, 0
33      mov ebx, 15
34      div ebx
35
36      cmp edx, 0
37      jne .fizz
```

Exemplo do uso de subrotinas: FizzBuzz

```
39      mov eax, fizz       ; A saída é FizzBuzz
40      call print_string
41
42      mov eax, buzz
43      call print_string
44
45      jmp .endl
46
47  .fizz:
48      mov eax, esi        ; Checa se é múltiplo de 3
49      mov edx, 0
50      mov ebx, 3
51      div ebx
52
53      cmp edx, 0
54      jne .buzz
55
```

Exemplo do uso de subrotinas: FizzBuzz

```
56      mov eax, fizz       ; A saída é Fizz
57      call print_string
58      jmp .endl
59
60  .buzz:
61      mov eax, esi        ; Checa se é múltiplo de 5
62      mov edx, 0
63      mov ebx, 5
64      div ebx
65
66      cmp edx, 0
67      jne .n
68
69      mov eax, buzz       ; A saída é Buzz
70      call print_string
71      jmp .endl
72
```

Exemplo do uso de subrotinas: FizzBuzz

```
73 .n:
74     mov eax, esi
75     call print_int
76
77 .endl:
78     mov eax, endl
79     call print_string
80     inc esi
81     jmp .fizzbuzz
82
83 .error:
84     mov eax, error_msg
85     call print_string
86
87 .exit:
88     mov ebx, 0
89     mov eax, 1
90     int 80h
```

Implementação das subrotinas utilizadas em FizzBuzz

```
1  ; Subrotinas utilizadas no programa FizzBuzz
2  ; Calcula o tamanho, em bytes, da string s, cujo endereço
3  ; de memória contido em EAX
4  string_len:
5      push ecx                    ; registrador usado na rotina
6      mov ecx, 0
7
8  .next:
9      cmp byte [eax + ecx], 0     ; 0 é o terminar da string
10     jz  .done
11
12     inc ecx                     ; verifica o próximo caractere
13     jmp .next
14
15 .done:
16     mov eax, ecx                ; Prepara o retorno
17     pop ecx                     ; Restaura o valor original de ECX
18     ret
```

Implementação das subrotinas utilizadas em FizzBuzz

```
20 ; Imprime a string s, contida em EAX
21 print_string:
22     push eax            ; Os 4 registradores de dados serão utilizados
23     push ebx
24     push ecx
25     push edx
26
27     mov ecx, eax        ; Guarda o endereço da string
28
29     call string_len     ; Calcula o tamanho da string
30     mov edx, eax
31
32     mov ebx, 1          ; Imprime em SYSOUT
33     mov eax, 4          ; Optcode de SYS_WRITE
34     int 80h             ; Chamada de sistema
35
```

Implementação das subrotinas utilizadas em FizzBuzz

```
36      pop edx             ; Restaura os valores originais dos registradores
37      pop ecx
38      pop ebx
39      pop eax
40
41      ret
42
43  ; Transforma a string s, contida em EAX, em um inteiro n
44  string_to_int:
45      push esi        ; Salva os registradores
46      push ebx
47      push ecx
48
49      mov esi, eax    ; Buffer contendo o número como string
50      mov eax, 0      ; EAX conterá o número convertido
51      mov ebx, 10     ; Base numérica
52      mov ecx, 0      ; Próximo dígito a ser processado
53
```

Implementação das subrotinas utilizadas em FizzBuzz

```
54 .next:
55     mov cl, [esi]   ; Obtém o próximo caractere
56
57     cmp cl, '0'     ; Se o caractere está fora da faixa [0-9] finaliza
58     jl .done
59
60     cmp cl, '9'
61     jg .done
62
63     sub ecx, '0'    ; Converte de ASCII para decimal
64
65     mul ebx         ; EAX = 10*EAX + ECX
66     add eax, ecx
67
68     inc esi         ; Avança o ponteiro e continua o laço
69     jmp .next
70
```

Implementação das subrotinas utilizadas em FizzBuzz

```
71 .done:
72     pop ecx          ; Restaura os registradores
73     pop ebx
74     pop esi
75
76     ret
77
78 ; Imprime o inteiro n, contido em EAX
79 print_int:
80     push eax         ; Salva os registradores
81     push ebx
82     push ecx
83     push edx
84
85     mov ecx, 0       ; ECX = número de dígitos
86
```

Implementação das subrotinas utilizadas em FizzBuzz

```
87 .next:
88     mov edx, 0       ; Prepara a divisão: a = EDX:EAX
89     mov ebx, 10      ; b = 10
90     div ebx          ; q = eax, r = edx
91
92     add edx, '0'     ; converte r para ASCII
93     push dx          ; insere na pilha
94     inc ecx          ; atualiza a contagem de dígitos
95
96     cmp eax, 0       ; Verifica se ainda há dígitos a serem extraídos
97     jne .next
98
99 .print:
100    cmp ecx, 0       ; Verifica se há dígitos a serem impressos
101    je .done
102
103    mov eax, esp     ; Imprime o topo da pilha
104    call print_string
```

Implementação das subrotinas utilizadas em FizzBuzz

```
105
106     add esp, 2      ; Remove o topo da pilha
107     dec ecx         ; Decrementa o contador
108     jmp .print
109
110 .done:
111     pop edx         ; Restaura os registradores
112     pop ecx
113     pop ebx
114     pop eax
115
116     ret
```

Referências

1. asmtutor.com. Learn Assembly Language, acesso em 16/01/2020.
2. NASM. Site Oficial, acesso em 16/01/2020.
3. **NEVELN**, Bob. *Linux Assembly Language Programming*, Open Source Technology Series, Prentice-Hall, 2000.
4. **SHALOM**, Elad. *A Review of Programming Paradigms Througout the History – With a Suggestion Toward a Future Approach*, Amazon, 2019.