Programação Imperativa

Condicionais

Prof. Edson Alves

Faculdade UnB Gama

2020

Sumário

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 controles (*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
2  ;
3  ;   p(x) = ax^2 + bx + c
4  ;
5  ; Como exemplo, serão utilizados os valores a = 1, b = -5 e c = 6
6  SECTION .data
7  a       dd  1               ; dd = variáveis com 4 bytes
8  b       dd  -5
9  c       dd  6
10 msg     db  '0', 0Ah, 0
11
12 SECTION .text
13 global _start
14
15 _start:
16     mov eax, 4      ; EBX = 4*a*c
17     mov ecx, [a]
18     mul ecx
19     mov ecx, [c]
20     mul ecx
21     mov ebx, eax
22
```

Exemplo de saltos condicionais

```
23      mov eax, [b]     ; b = -5
24      mul eax          ; EAX = b^2
25
26      sub eax, ebx     ; EAX = b^2 - 4*a*c
27
28      jz  one          ; Se EAX = 0 há apenas uma raiz
29      jns two          ; Se EAX > 0 há duas raizes reais
30
31      mov bl, '0'      ; Caso contrário não há raizes reais
32      jmp finish
33
34  one:
35      mov bl, '1'
36      jmp finish
37
38  two:
39      mov bl, '2'
40
41  finish:
42      mov [msg], bl    ; Atualiza a mensagem com o número de raizes
43
```

Exemplo de saltos condicionais

```
44      mov edx, 3          ; msg tem 3 bytes
45      mov ecx, msg        ; msg é a mensagem a ser impressa
46      mov ebx, 1          ; A saída é STDOUT
47      mov eax, 4          ; Optcode de SYS_WRITE
48      int 80h
49
50      mov ebx, 0          ; Encerra com sucesso
51      mov eax, 1
52      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
18     mov edx, 7       ; n é ímpar
19     mov ecx, odd
20     jmp finish
21
```

Exemplo de comparações

```
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
```

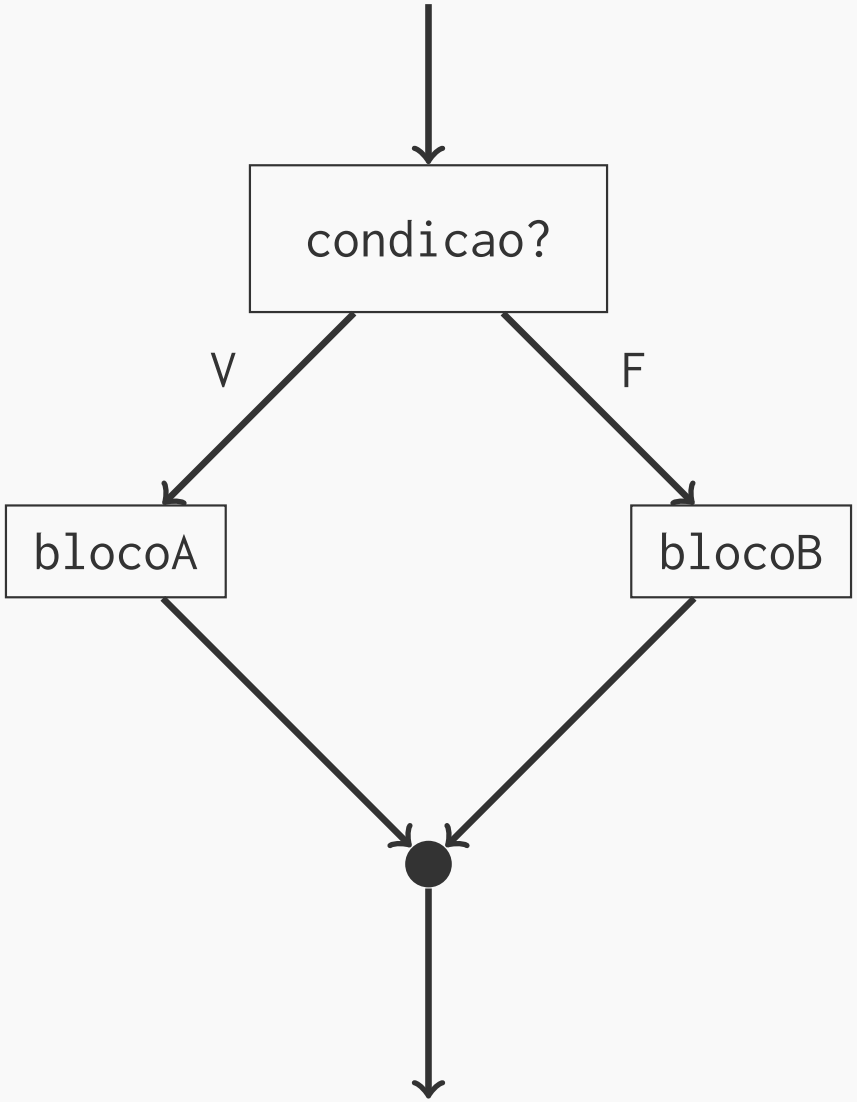
IF-ELSE

- 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

```
1  ; Código correspondente ao construto IF-ELSE
2
3      cmp regA, regB      ; Esta comparação corresponde à condição
4      jnz .blockA         ; Assuma 0 falso, caso contrário verdadeiro
5      jmp .blockB
6
7  .blockA:
8      ; Commandos associados ao blocoA
9      jmp finish
10
11 .blockB:
12     ; Commandos associados ao blocoB
13     jmp finish
14
15 .finish:
16     ; Prossegue com a execução do código
```

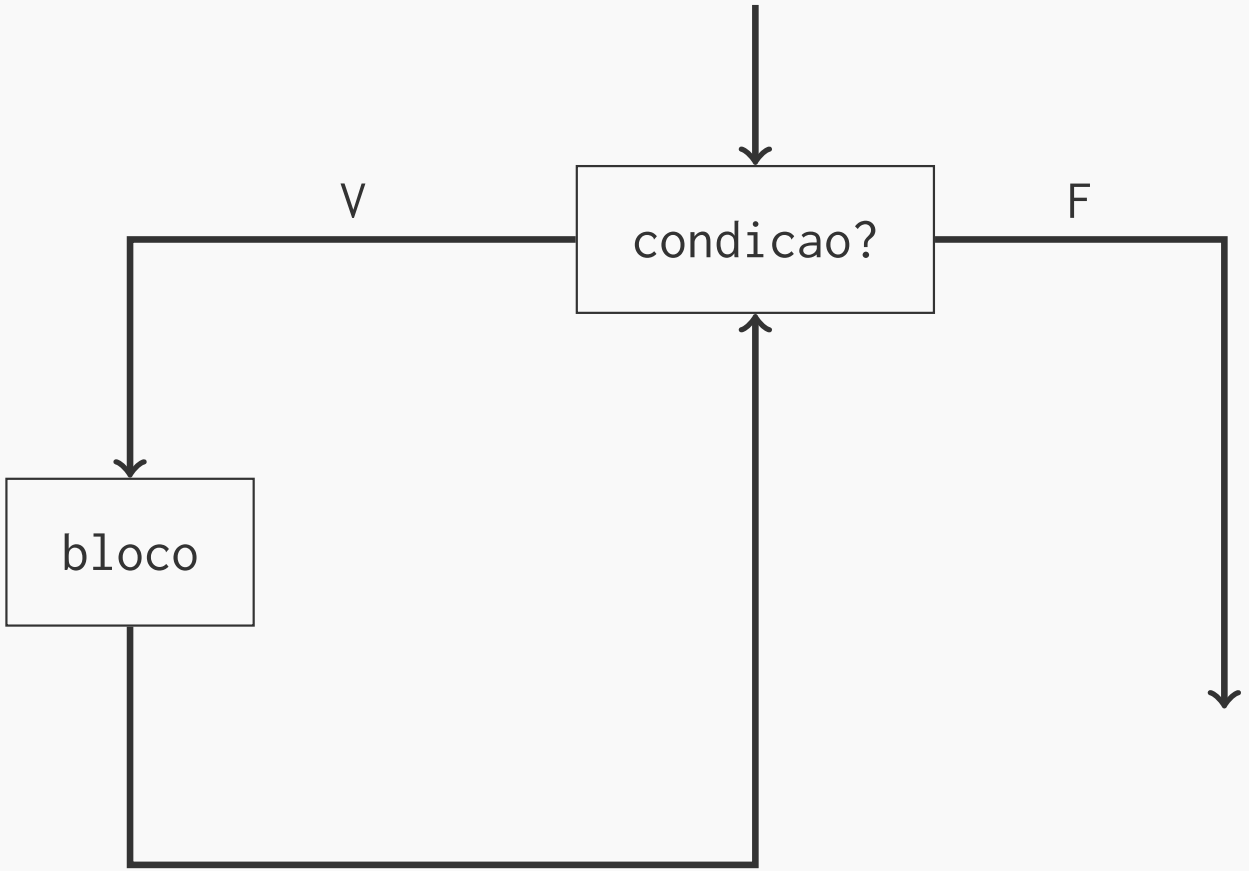
WHILE

- 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'
19
20     dec eax         ; Desconta o '\n'
21
```

Exemplo de laços e condicionais

```
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
38      cmp cl, '0'     ; Se o caractere está fora da faixa [0-9] finaliza
39      jl done
40
41      cmp cl, '9'
42      jg done
43
```

Exemplo de laços e condicionais

```
44      sub ecx, '0'      ; Converte de ASCII para decimal
45
46      mul ebx           ; EAX = 10*EAX + ECX
47      add eax, ecx
48
49      inc esi           ; Avança o ponteiro e continua o laço
50      jmp to_int
51
52  done:
53      mov ebx, eax      ; EBX = n
54
55      ; Verifica se o número é menor que 2
56      mov ecx, 2
57      cmp ebx, ecx
58      jl not_prime
59
60      je is_prime       ; Verifica se é igual a 2
61
62      ; Verifica se é par
63      mov eax, ebx
64      div ecx
65      cmp edx, 0
66      je not_prime
```

Exemplo de laços e condicionais

```
67
68      ; Tenta todos os ímpares menores que a raiz quadrada
69      mov ecx, 3
70
71  next:
72      ; Checa se há ultrapassou a raiz quadrada
73      mov eax, ecx
74      mul eax
75      cmp eax, ebx
76      jg is_prime
77
78      ; Verifica se ECX divide n
79      mov eax, ebx
80      div ecx
81      cmp edx, 0
82      je not_prime
83
84      ; Tenta o próximo ímpar
85      add ecx, 2
86      jmp next
87
```

Exemplo de laços e condicionais

```
88 is_prime:
89     mov edx, 11      ; 'yes' tem 11 bytes
90     mov ecx, yes     ; Escreve o conteúdo de yes
91     jmp finish
92
93 not_prime:
94     mov edx, 15      ; 'no' tem 15 bytes
95     mov ecx, no      ; Escreve o conteúdo de no
96     jmp finish
97
98 finish:
99     mov ebx, 1       ; Escreve em STDOUT
100    mov eax, 4       ; Optcode de sys_write
101    int 80h
102
103    mov ebx, 0       ; Encerra com sucesso
104    mov eax, 1
105    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 não pode ser escrito diretamente

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

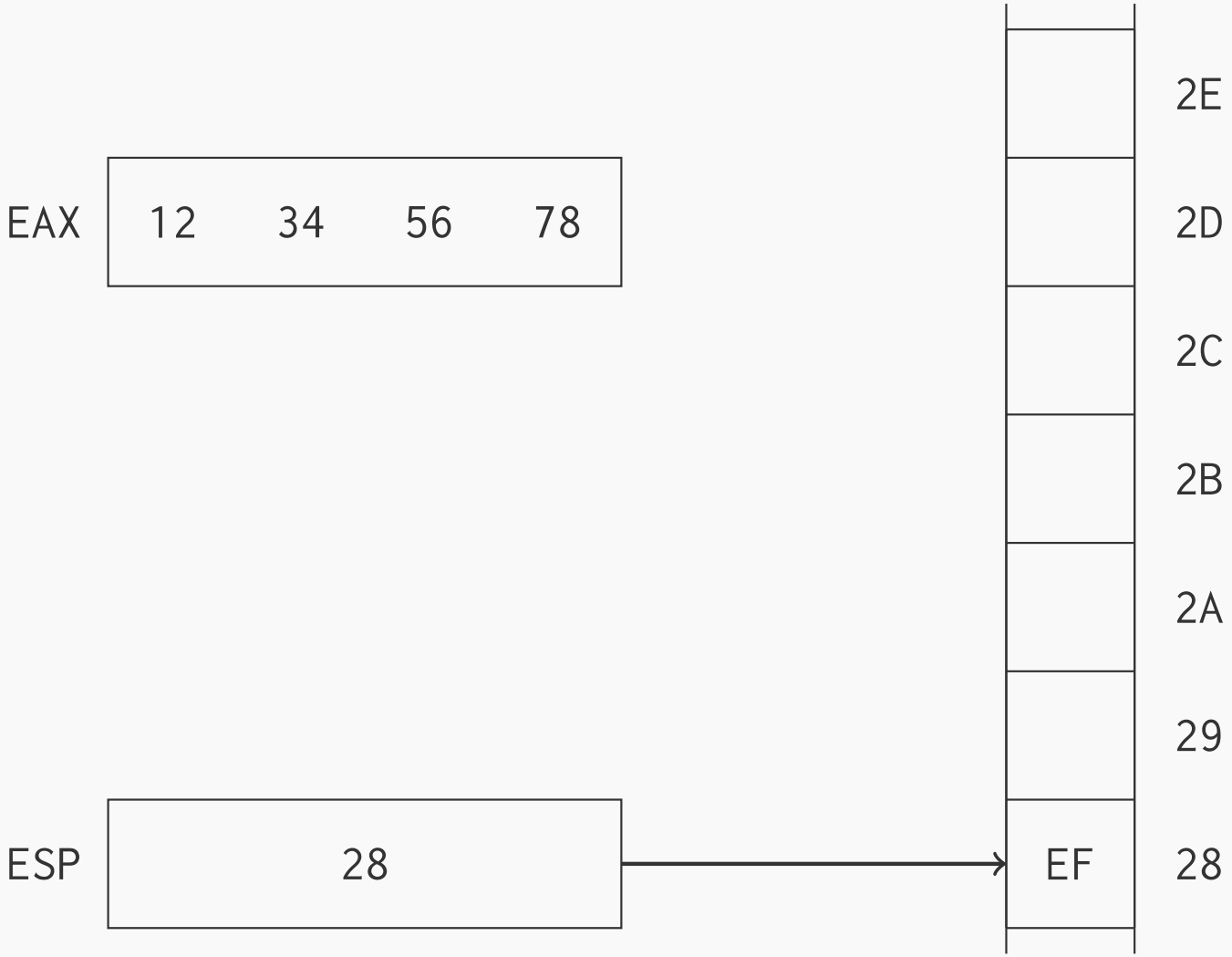


Figura: Estado do programa

Visualização da inserção de elementos na pilha

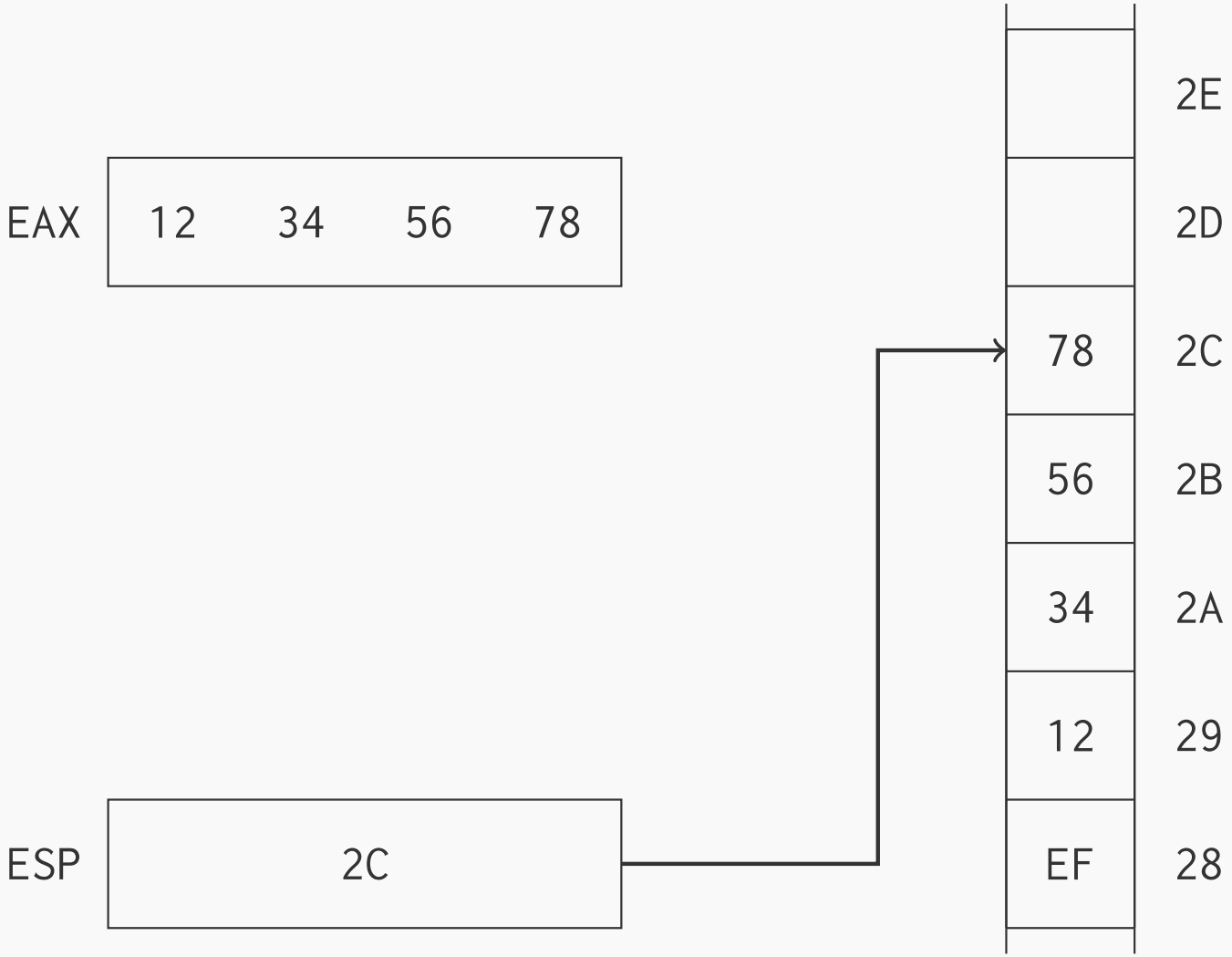


Figura: Estado do programa após PUSH EAX

Visualização da inserção de elementos na pilha

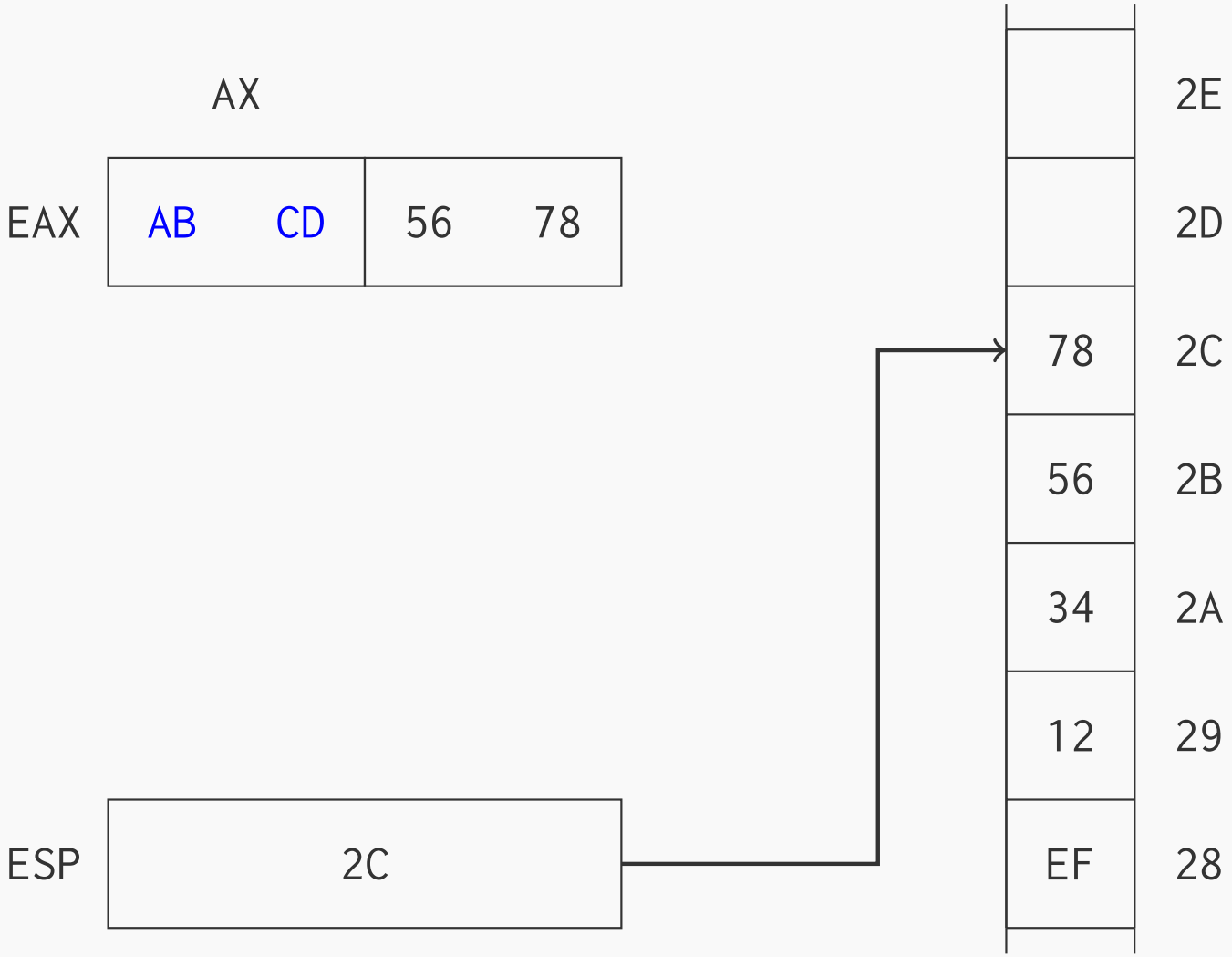


Figura: Atualização do registrador AX

Visualização da inserção de elementos na pilha

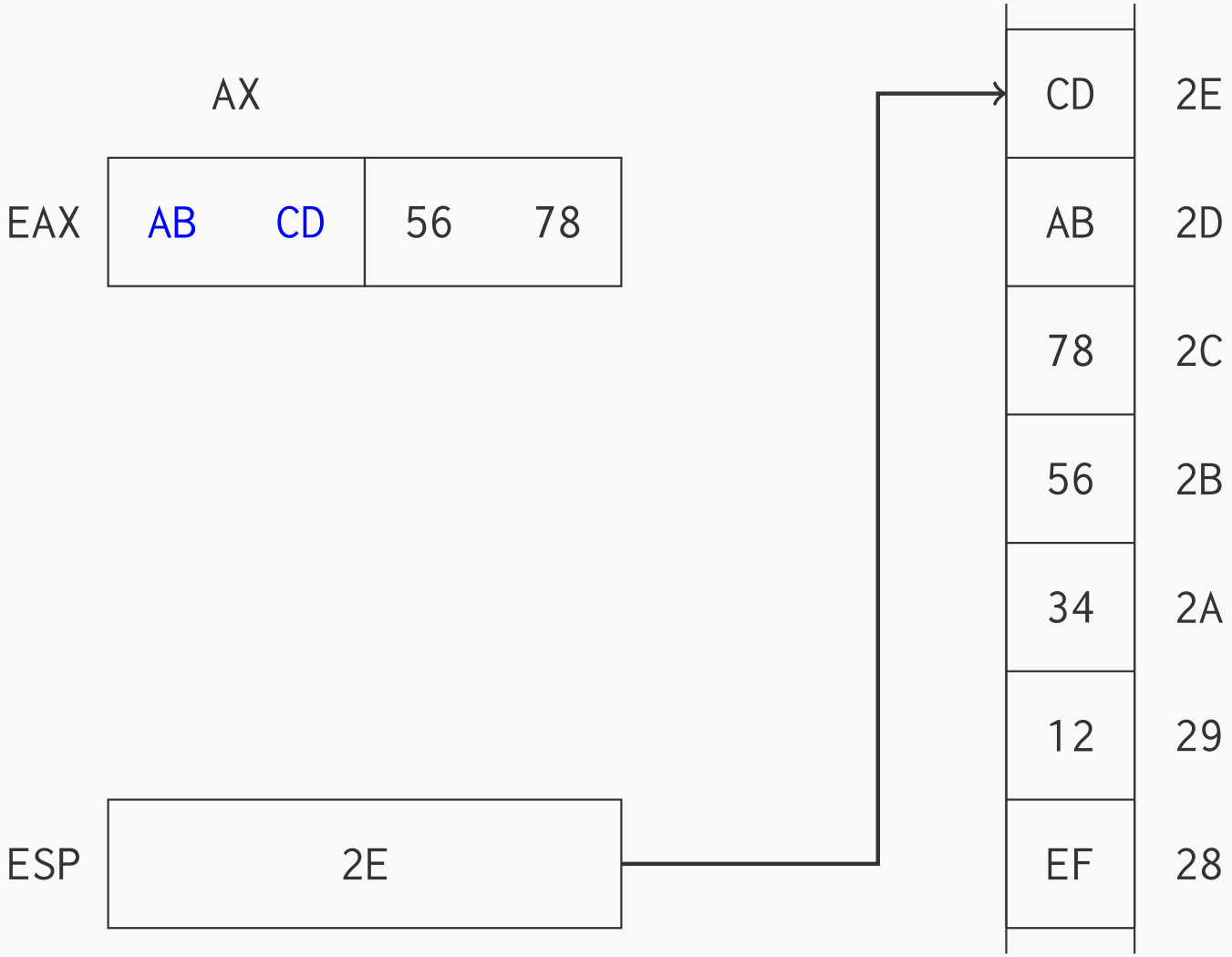


Figura: Estado do programa após PUSH AX

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

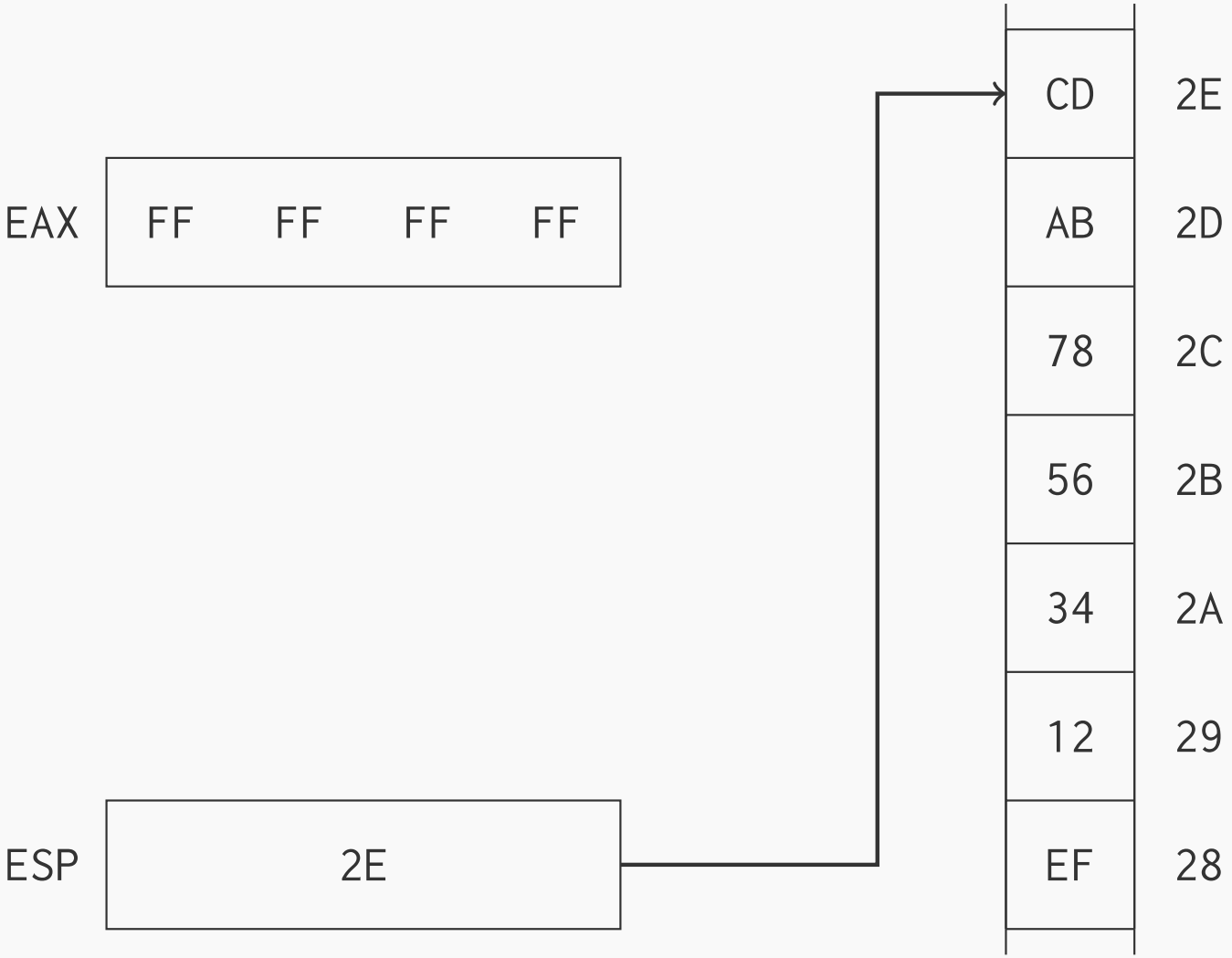


Figura: Estado do programa

Visualização da remoção de elementos na pilha

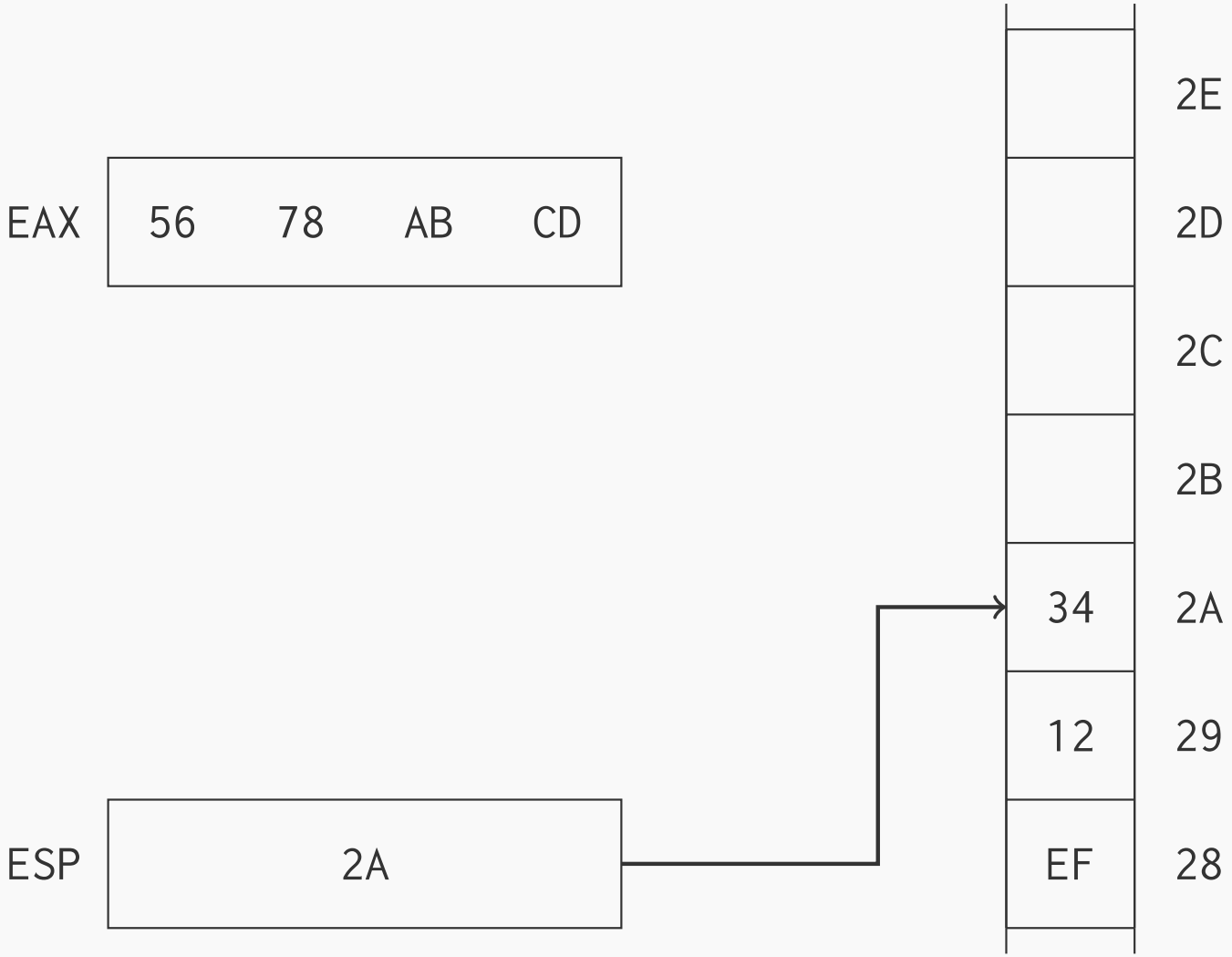


Figura: Estado do programa após a instrução POP EAX

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
19
20     cmp eax, 0       ; Verifica se ainda há dígitos a serem extraídos
21     jne next
22
```

Exemplo de uso da pilha

```
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
37      add esp, 2      ; Remove o topo da pilha
38      dec esi         ; Decrementa o contador
39      jmp print_int
40
41  done:
42      ; Imprime a quebra de linha
43      mov edx, 0Ah
44      push dx
```

Exemplo de uso da pilha

```
45
46      mov edx, 1
47      mov ecx, esp
48      int 80h
49
50      pop dx          ; Remove a quebra de linha da pilha
51
52  exit:
53      ; Encerra o programa com sucesso
54      mov ebx, 0
55      mov eax, 1
56      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
18     cmp ecx, 2          ; Checa se o argumento foi passado
19     jl  .error
20
21     mov eax, [esp + 4]  ; Ignora o nome do programa
22
```

Exemplo do uso de subrotinas: FizzBuzz

```
23      call string_to_int  ; Converte o argumento para inteiro
24
25      mov esi, 1          ; Início da contagem
26      mov edi, eax        ; Fim da contagem
27
28  .fizzbuzz:
29      cmp esi, edi        ; Verifica se a contagem já terminou
30      jg .exit
31
32      mov eax, esi        ; Checa se é múltiplo de 15
33      mov edx, 0
34      mov ebx, 15
35      div ebx
36
37      cmp edx, 0
38      jne .fizz
39
40      mov eax, fizz       ; A saída é FizzBuzz
41      call print_string
42
43      mov eax, buzz
44      call print_string
45
```

Exemplo do uso de subrotinas: FizzBuzz

```
46      jmp .endl
47
48  .fizz:
49      mov eax, esi        ; Checa se é múltiplo de 3
50      mov edx, 0
51      mov ebx, 3
52      div ebx
53
54      cmp edx, 0
55      jne .buzz
56
57      mov eax, fizz       ; A saída é Fizz
58      call print_string
59      jmp .endl
60
61  .buzz:
62      mov eax, esi        ; Checa se é múltiplo de 5
63      mov edx, 0
64      mov ebx, 5
65      div ebx
66
67      cmp edx, 0
68      jne .n
```

Exemplo do uso de subrotinas: FizzBuzz

```
69
70      mov eax, buzz         ; A saída é Buzz
71      call print_string
72      jmp .endl
73
74  .n:
75      mov eax, esi
76      call print_int
77
78  .endl:
79      mov eax, endl
80      call print_string
81
82      inc esi
83      jmp .fizzbuzz
84
85  .error:
86      mov eax, error_msg
87      call print_string
88
```

Exemplo do uso de subrotinas: FizzBuzz

```
89 .exit:
90     mov ebx, 0
91     mov eax, 1
92     int 80h
```

Implementação das subrotinas utilizadas em FizzBuzz

```
1  ; Subrotinas utilizadas no programa FizzBuzz
2
3  ; Calcula o tamanho, em bytes, da string s, cujo endereço
4  ; de memória contido em EAX
5  string_len:
6      push ecx                    ; registrador usado na rotina
7      mov ecx, 0
8
9  .next:
10     cmp byte [eax + ecx], 0     ; 0 é o terminar da string
11     jz  .done
12
13     inc ecx                     ; verifica o próximo caractere
14     jmp .next
15
16 .done:
17     mov eax, ecx                ; Prepara o retorno
18     pop ecx                     ; Restaura o valor original de ECX
19     ret
20
```

Implementação das subrotinas utilizadas em FizzBuzz

```
21 ; Imprime a string s, contida em EAX
22 print_string:
23     push eax            ; Os 4 registradores de dados serão utilizados
24     push ebx
25     push ecx
26     push edx
27
28     mov ecx, eax        ; Guarda o endereço da string
29
30     call string_len     ; Calcula o tamanho da string
31     mov edx, eax
32
33     mov ebx, 1          ; Imprime em SYSOUT
34     mov eax, 4          ; Optcode de SYS_WRITE
35     int 80h             ; Chamada de sistema
36
37     pop edx             ; Restaura os valores originais dos registradores
38     pop ecx
39     pop ebx
40     pop eax
41
42     ret
43
```

Implementação das subrotinas utilizadas em FizzBuzz

```
44 ; Transforma a string s, contida em EAX, em um inteiro n
45 string_to_int:
46     push esi         ; Salva os registradores
47     push ebx
48     push ecx
49
50     mov esi, eax     ; Buffer contendo o número como string
51     mov eax, 0       ; EAX conterá o número convertido
52     mov ebx, 10      ; Base numérica
53     mov ecx, 0       ; Próximo dígito a ser processado
54
55 .next:
56     mov cl, [esi]    ; Obtém o próximo caractere
57
58     cmp cl, '0'      ; Se o caractere está fora da faixa [0-9] finaliza
59     jl .done
60
61     cmp cl, '9'
62     jg .done
63
64     sub ecx, '0'     ; Converte de ASCII para decimal
65
```

Implementação das subrotinas utilizadas em FizzBuzz

```
66      mul ebx             ; EAX = 10*EAX + ECX
67      add eax, ecx
68
69      inc esi             ; Avança o ponteiro e continua o laço
70      jmp .next
71
72  .done:
73      pop ecx             ; Restaura os registradores
74      pop ebx
75      pop esi
76
77      ret
78
```