Arquitetura de Computadores

Uma Introdução

Gabriel P. Silva – José Antonio Borges

Programação em Linguagem de Montagem

Capítulo 7

7.1 DA LINGUAGEM DE ALTO NÍVEL À EXECUÇÃO REAL

O que é Linguagem de Montagem (Assembly Language)?

- Cada processador executa instruções específicas de sua arquitetura
  - Uma instrução é um **código binário** que indica uma **ação a ser realizada pelo processador.**
  - As instruções são representadas como números binários:

    00110000 → operação de somar
- No passado, para realizar as operações desejadas no computador, os programas eram carregados na memória apenas como um conjunto de bits:
  - Isso era um processo altamente trabalhoso e sujeito a erros.

O que é Linguagem de Montagem (Assembly Language)?

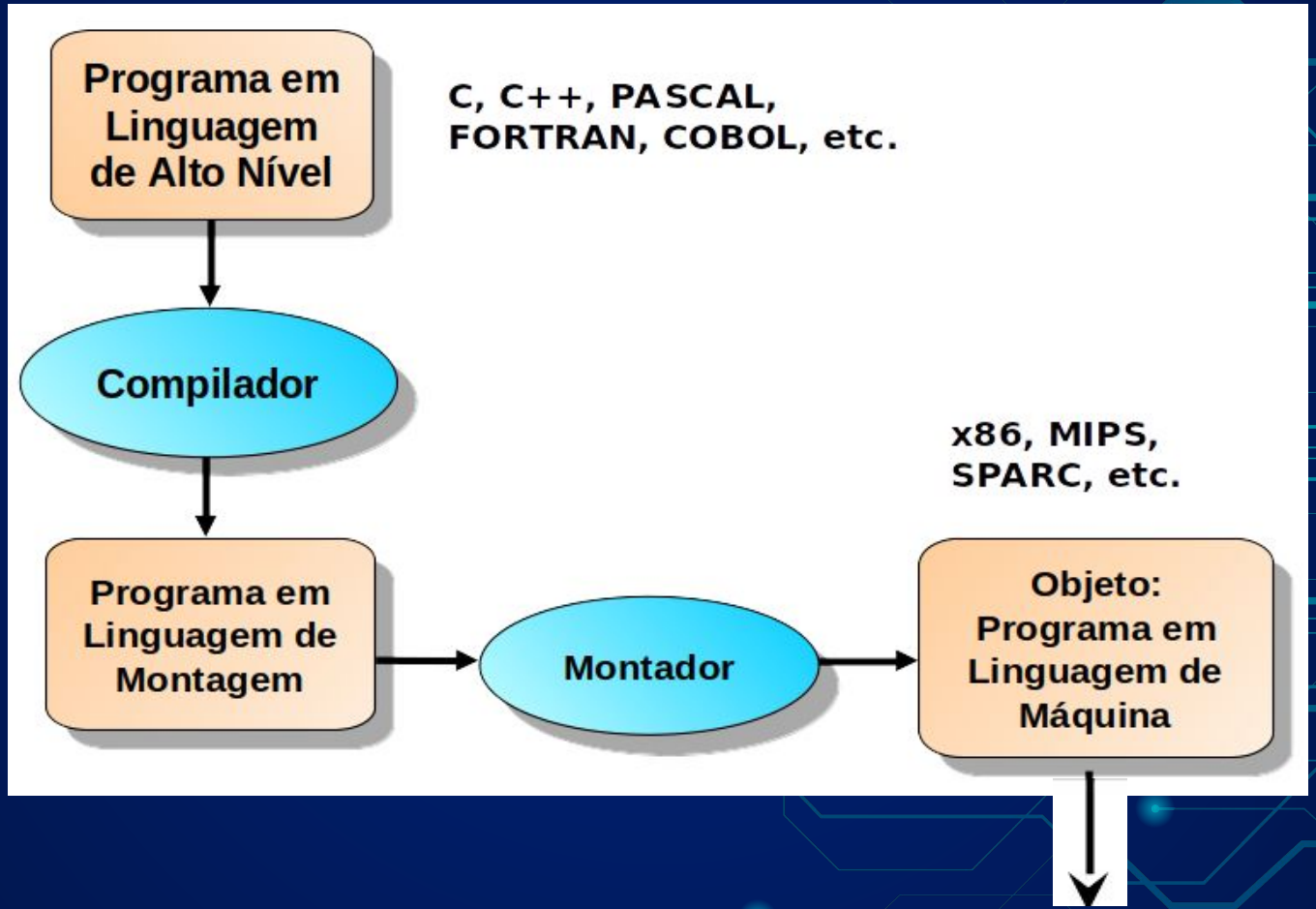
- Para garantir a escrita correta de centenas de instruções, os programadores criavam uma "versão preliminar" utilizando mnemônicos, conhecidos como linguagem de montagem (assembly language), em vez de códigos numéricos diretos.
  - Em vez de **00110000** usa-se um mnemônico **ADD**
- Para simplificar a referência aos dados do programa armazenados na memória, alguns recursos eram utilizados, tais como:
  - Informar em que endereço estavam;
  - Dizer o tamanho dos dados ou associando um valor inicial.

O que é Linguagem de Montagem (Assembly Language)?

```
        ADD   X       ; somar a variável X ao acumulador
...
        ORG   100
X:      DB    5       ; a variável X fica no endereço 100 de
                      ; memória e ocupa 1 byte com valor inicial 5
```

O que é Linguagem de Montagem (Assembly Language)?

- A tradução deste texto criado com mnemônicos para linguagem de máquina é feita por um programa chamado MONTADOR (assembler)
- Não existe um programa montador universal:
  - cada processador será associado a um programa montador **específico** para ele.
  - Um programa escrito em certa linguagem de montagem **só serve** para um processador específico.
    - Por isso, utiliza-se uma linguagem de nível mais alto, que é traduzida para a linguagem de montagem por um COMPILADOR
    - Os compiladores são preparados para gerar um texto em linguagem de montagem para um processador específico.


Programa em Linguagem de Alto Nível
C, C++, PASCAL, FORTRAN, COBOL, etc.
Compilador
x86, MIPS, SPARC, etc.
Programa em Linguagem de Montagem
Montador
Objeto: Programa em Linguagem de Máquina

Programa em Linguagem ‘C’

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
signed char a=3,b=4,c;
    c=a+b;
    if (c==7)
        printf ("A soma deu certo!\n");
    else
        printf ("A soma deu errado!\n");
    exit(0);
}
```

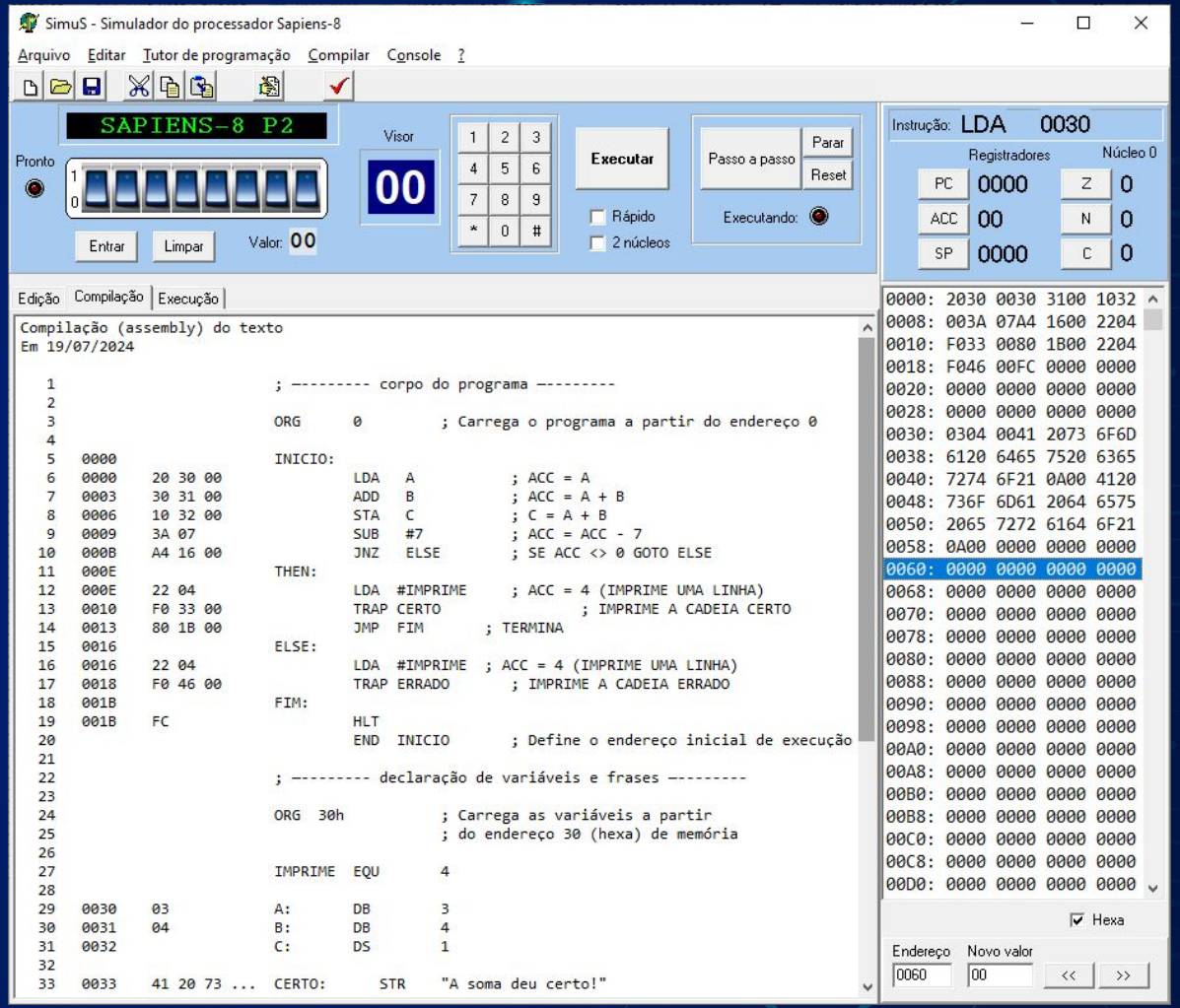
Programa em Linguagem de Montagem

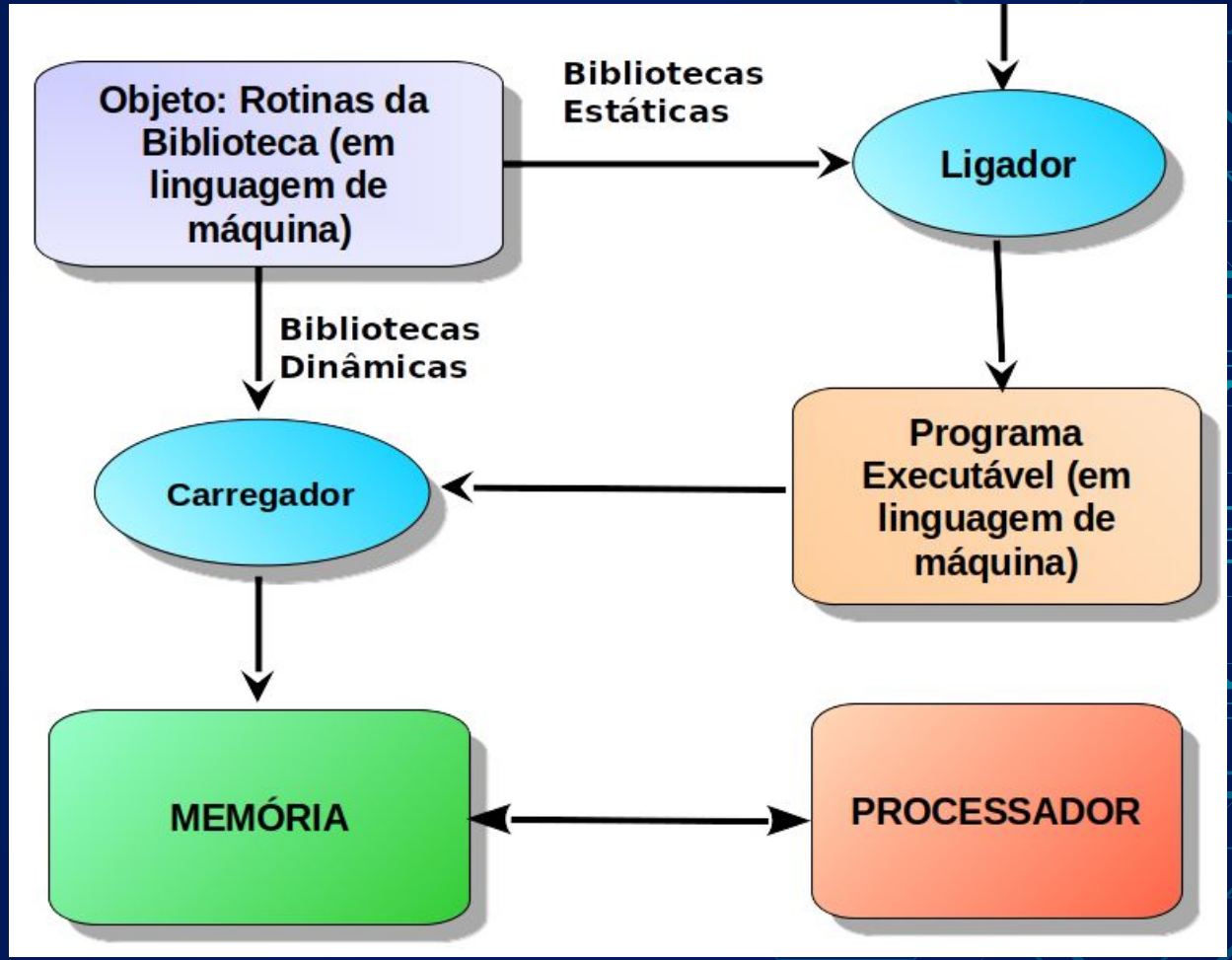
```
; --------- declaração de variáveis e textos ---------

ORG       30h     ; Carrega as variáveis a partir
                  ; do endereço 30 (hexa) de memória
IMPRIME   EQU     4

A:        DB      3         ; Variáveis A, B e C
B:        DB      4
C:        DS      1         ; Sem valor inicial

CERTO:    STR     "A soma deu certo!"      ; Strings codificadas
          DB      0Ah       ; \n
          DB      00h       ; fim de string

ERRADO:   STR     "A soma deu errado!"
          DB      0Ah       ; \n
          DB      00h       ; fim de string
```

```
; --------- corpo do programa ---------

ORG 0     ; Carrega as instruções a  partir
          ; do endereço 0 de memória
INICIO:
          LDA   A          ; ACC = A
          ADD   B          ; ACC = A + B
          STA   C          ; C = A + B
          SUB   #7         ; ACC = ACC - 7
          JNZ   ELSE       ; SE ACC <> 0 GOTO ELSE
THEN:
          LDA  #IMPRIME    ; ACC = 4 (IMPRIME UMA LINHA)
          TRAP CERTO       ; IMPRIME A CADEIA CERTO
          JMP  FIM         ; TERMINA
ELSE:
          LDA  #IMPRIME    ; ACC = 4 (IMPRIME UMA LINHA)
          TRAP ERRADO      ; IMPRIME A CADEIA ERRADO
FIM:
          HLT
          END  INICIO      ; Define o endereço inicial de execução
```

Ambiente de Desenvolvimento

Incluindo
- Editor,
- Compilador
- Simulador




Objeto: Rotinas da Biblioteca (em linguagem de máquina)
Bibliotecas Estáticas
Ligador
Bibliotecas Dinâmicas
Carregador
Programa Executável (em linguagem de máquina)
MEMÓRIA
PROCESSADOR

Listagem da compilação e código gerado

```
  7   0030    03              A:        DB3 ; Variáveis A, B e C
  8   0031    04              B:        DB4
  9   0032                    C:        DS1
 11   0033    41 20 73 ...    CERTO:  STR"A soma deu certo!"; Strings codificadas
 12   0044    0A                        DB0Ah; \n
 13   0045    00                        DB00h; fim de string
 15   0046    41 20 73 ...    ERRADO:   STR"A soma deu errado!"
 16   0058    0A                        DB0Ah; \n
 17   0059    00                        DB00h; fim de string
 23   0000                    INICIO:
 24   0000    20 30 00                  LDA   A    ; ACC = A
 25   0003    30 31 00                  ADD   B    ; ACC = A + B
 26   0006    10 32 00                  STA   C    ; C = A + B
 27   0009    3A 07                     SUB   #7   ; ACC = ACC - 7
 28   000B    A4 16 00                  JNZ   ELSE ; SE ACC <> 0 GOTO ELSE
 29   000E                    THEN:
 30   000E    22 04                     LDA  #IMPRIME; ACC = 4 (IMPRIME UMA LINHA)
 31   0010    F0 33 00                  TRAP CERTO; IMPRIME A CADEIA CERTO
 32   0013    80 1B 00                  JMP  FIM       ; TERMINA
 33   0016                    ELSE:
 34   0016    22 04                     LDA  #IMPRIME  ; ACC = 4 (IMPRIME UMA LINHA)
 35   0018    F0 46 00                  TRAP ERRADO; IMPRIME A CADEIA ERRADO
 36   001B                    FIM:
 37   001B    FC                        HLT
```

```
0030 A          03[3]
0031 B          04[4]
0032 C          00[0]
0033 CERTO      "A soma deu certo!"
0044           0A[10]
0045           00[0]
0046 ERRADO     "A soma deu errado!"
0058           0A[10]
0059           00[0]
```

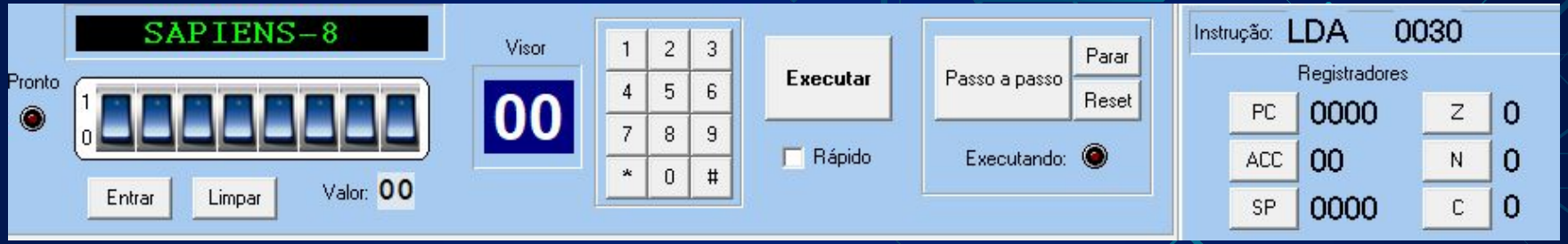
Conteúdo da memória, após compilação

Código em linguagem de máquina (endereço 0) e dados (endereço 30h)

(à esquerda o endereço em hexadecimal)

```
0000: 2030 0030 3100 1032
0008: 003A 07A4 1600 2204
0010: F033 0080 1B00 2204
0018: F046 00FC 0000 0000
0020: 0000 0000 0000 0000
0028: 0000 0000 0000 0000
0030: 0304 0041 2073 6F6D
0038: 6120 6465 7520 6365
0040: 7274 6F21 0A00 4120
0048: 736F 6D61 2064 6575
0050: 2065 7272 6164 6F21
0058: 0A00 0000 0000 0000
```

Visualização do simulador



7.2 LINGUAGEM DE MONTAGEM DO SAPIENS

Introdução

- Criada para facilitar a programação na linguagem de máquina do Sapiens
- Pequeno número de instruções
  - Número reduzido de tipos de operandos
- Fácil de aprender
- Usa uma forma de codificar muito similar à maioria dos montadores do mercado
  - O montador do Sapiens aceita apenas instruções deste processador.

Aspectos gerais

- As linhas do texto podem conter os seguintes elementos:
  - Rótulo - seguido de dois pontos
    - não podem ter acentos nem espaços
    - Não há distinção entre maiúsculas e minúsculas
    - A primeira letra não pode ser numérica
  - Comentários - precedidos por ponto e vírgula
  - Instruções e operandos
  - Pseudo-instruções: são orientações para o montador posicionar o código e as variáveis na memória.

Pseudo instruções

- ORG
- EQU
- DB
- DW
- STR

```
        ORG     100

MAX     EQU     99

X:      DB      3
VETOR:  DB      3, 5, 7, 9

DADOS:  DW      1000

MSG:    STR     "mensagem"
        DB      0
```

Representação de números

- Decimal: -48
- Hexadecimal: 0D0h
- Binário: 11010000b

7.3 ATRIBUIÇÃO DE VARIÁVEIS

O Sapiens é um processador baseado em acumulador (ACC)

- Todas as operações que envolvem uso da memória ou aritmética, são realizadas através do acumulador.
  - Preencher uma posição de memória com certo valor (X = 10)
    - Carregar o valor desejado (p.ex. 10) no acumulador
    - Armazenar o valor do acumulador na memória.
  - Copiar uma variável para outra variável (ex.: X = Y)
    - Carregar o dado na posição Y de memória no acumulador
    - Armazenar o valor do acumulador na posição X de memória.

Atribuição com uma constante

```
ORG    0

CONST: DB     10 ; Variável CONST com valor inicial 10
A:     DS     1  ; Variável A sem valor

INICIO:
       LDA    CONST  ; ACC = 10
       STA    A      ; A = ACC
       HLT           ; Termina a execução
       END    INICIO ; Indica onde a execução inicia
```

Operando imediato (#)

```
ORG     0

A:      DS      1   ; Variável A sem valor

INICIO:
        LDA     #10     ; ACC = 10
        STA     A       ; A = ACC
        HLT             ; Termina a execução
        END     INICIO ; Indica onde a execução inicia
```

Atribuições simples (A ← B)

```
ORG 100
A:      DS    1          ; A no endereço 100
B:      DB    7          ; B no endereço 101 (valor inicial 7)

ORG 0
        LDA   B          ; Acumulador = B
        STA   A          ; A = ACC
        HLT              ; Termina a execução
        END   0          ; onde o programa começa a executar
```

Não há necessidade das variáveis serem declaradas primeiro!

```
ORG 0
        LDA    B        ; Acumulador = B
        STA    A        ; A = ACC
        HLT             ; Termina a execução

ORG 100
A:      DS     1        ; A no endereço 100
B:      DB     7        ; B no endereço 101 (valor inicial 7)

        END    0        ; onde o programa começa a executar
```

7.4 OPERAÇÕES ARITMÉTICAS

Operação com uma variável e uma constante (B ← A + 5)

```
ORG 128
A:      DB    3     ; A com valor inicial 3
B:      DS    1     ; B sem valor inicial, espaço para 1 byte

ORG 0
        LDA   A     ; Acumulador = A
        ADD   #5    ; Acumulador = Acumulador + 5
        STA   B     ; B = Acumulador
        HLT         ; Termina a execução
        END   0
```

Operação com duas variáveis

```
ORG 128
X:      DB      5       ; X com valor inicial 5
Y:      DB      9       ; Y com valor inicial 9
Z:      DS      1       ; Z no endereço 130

ORG 0
        LDA     X       ; Acumulador = X
        ADD     Y       ; Acumulador = Acumulador + Y
        STA     Z       ; Z = Acumulador
        HLT             ; Termina a execução
        END     0
```

Operação com três variáveis

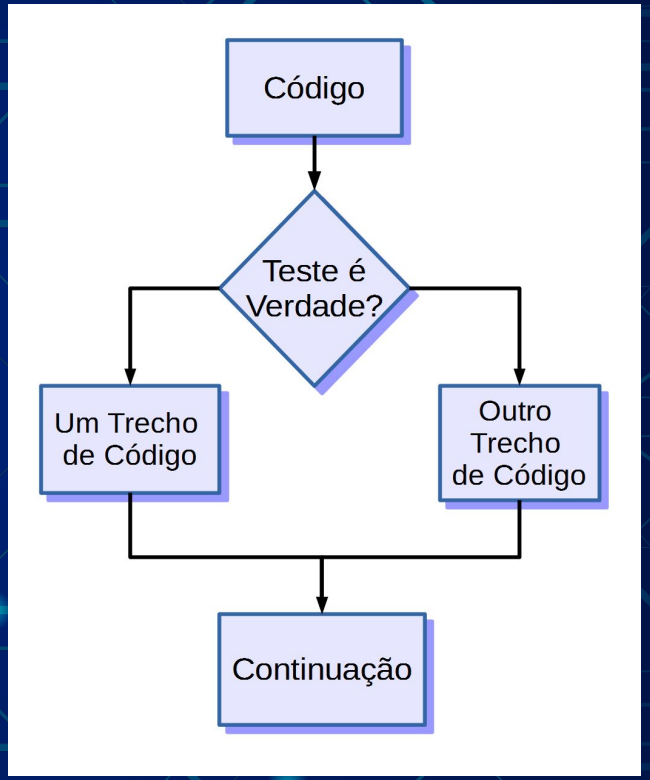
```
ORG 100 ; — dados —
X:      DB  0
Y:      DB  70
Z:      DB  91
```

```
ORG 0   ; — código —
        LDA   Y     ; X = Y
        STA   X
        LDA   Z     ; Soma 1 à variável Z
        ADD   #1
        STA   Z
        LDA   X     ; Subtrai 1 da variável X
        SUB   #1
        STA   X
        LDA   X     ; X = X + Y + Z
        ADD   Y
        ADD   Z
        STA   X
        HLT         ; termina a execução
        END   0
```

7.5 TESTES E DESVIOS

Tomada de decisão

```
...

if (certa condição é verdadeira)
{
      caz algumas coisas;
}
else
{
      Faz outras coisas;
}

... continua o programa ...
```



Flags do processador

- Dentro do Sapiens existem alguns indicadores (também chamados de flags) que indicam o que aconteceu na última **operação aritmética ou lógica**.
- São os seguintes os flags

  Z - deu zero

  N - deu negativo (bit 7 do resultado ligado)

  C - vai um - estourou a capacidade da variável (C - carry)
- Eles são a chave para o processamento das tomadas de decisão, como veremos a seguir.

Desvios

- O Sapiens realiza dois tipos de desvio:
  - Incondicional (vai direto para outro endereço do código)
  - Condicional (só vai para outro endereço caso um dos flags da do processador esteja ligado)
- São as seguintes as instruções de desvio:

`JMP`	desvio incondicional (vai direto para ...).
`JZ`	desvia se o flag zero estiver ativado.
`JNZ`	" " " " zero não estiver ativado.
`JN`	" " " " negativo estiver ativado.
`JP`	" " " " negativo e zero não estiverem ativados.
`JC`	" " " " vai-um estiver ativado.
`JNC`	" " " " vai-um não estiver ativado.

Testando valores diferentes (A <> B)

```
        LDA A
        SUB B
        JNZ DIFERENTES
IGUAIS:
        ; ... Faz coisas para quando A e B forem iguais
        ; ...
        JMP SEGUE

DIFERENTES:
        ; ... Faz coisas para quando diferentes
        ; ...

SEGUE:
        ; ... Continua o programa
```

Testando A > 10

```
        LDA #10
        SUB A
        JN MAIOR
; MENOR_OU_IGUAL: ; (rótulo desnecessário)
        ; ... Faz coisas para quando A menor ou igual a 10
        ; ...
        JMP SEGUE
MAIOR:
        ; ... Faz coisas adequadas a > 10
        ; ...
SEGUE:
        ; ... Continua o programa
```

Testando A <= 10

```
        LDA #10
        SUB A
        JZ IGUAL
        JN MAIOR
; MENOR: ; (rótulo desnecessário)
        ; ... Faz coisas para quando A < 10
        JMP SEGUE
IGUAL:
        ; ... Faz coisas adequadas a = 10
        JMP SEGUE
MAIOR:
        ; ... Faz coisas adequadas a > 10
        ; ...
SEGUE:
        ; ... Continua o programa
```

7.6 REPETIÇÕES

Repetições infinitas (laços simples)

- Neste tipo de situação, algumas instruções são executadas repetidas vezes, sem fim.

```
   …
LACO:
   … faz algumas coisas
   …
       JMP LACO
```

  - *Nota: uma eventual situação de quebra do laço pode ser adicionada no meio do código, se necessário, desviando para o fim do laço usando um desvio condicional*

```
ORG 0
        LDA   #1        ; rep = 1
        STA   REP
REPETE:
        ; ... faz várias coisas aqui
        ; ... eventualmente, irá alterar a variável REP

        LDA   REP       ; testa REP
        JNZ   REPETE    ; para indicar se desvia ou não
         ...
        HLT

REP:    DS    1         ; variável que indica se está repetindo
        END   0
```

Repetições com contador

for (CONTA = 1; i <= 10; i++)
...(faz alguma coisa)

```
        LDA #1
        STA CONTA
REPETE:
        ; ... trecho que será repetido
        ; ...
        LDA CONTA       ; Incrementa o contador
        ADD #1
        STA CONTA
        SUB #10         ; Testa se passou do valor final
        JN REPETE
        ...
        HLT
CONTA: DS  1
```

7.7 SOMA E SUBTRAÇÃO EM 16 BITS

Lidando com 16 ou 32 bits em uma máquina de 8 bits

- No Sapiens, as instruções instruções ADC e SBC, que fazem uso da flag C, são a maneira mais prática de realizar essa separação.
- Através dessas instruções fica fácil encadear somas e subtrações com vários bytes, automaticamente levando em consideração os eventuais casos de “vai-um” e “vem-um” através da flag C.
- Não é necessário o uso de desvios condicionais, o que torna o código mais conciso e fácil de entender.

Soma de 16 bits

```
....
        LDA X       ; Z = X + Y
        ADD Y
        STA Z
        LDA X+1     ; …+1 indica os 8 bits mais significativos
        ADC Y+1     ; das variáveis
        STA Z+1
....

X:      DW  1080H   ; variáveis iniciadas com 16 bits (DW)
Y:      DW  2080H
Z:      DS      2
```

7.8 OPERAÇÕES DE ENTRADA E SAÍDA

Instruções IN e OUT

- Dispositivos de entrada e saída do SimuS:
  - painel de oito chaves binárias
  - visor simples com dois displays de 7 segmentos
  - banner alfanumérico com uma linha com 16 caracteres
  - um teclado numérico com 12 teclas.

Operações de controle básico da E/S

- `IN 0` ACC = valor binário do painel de 8 chaves;
- `IN 1` ACC ← 1 quando houver novos dados no painel de chaves;
- `IN 2` ACC ← valor ASCII da tecla pressionada numa das 12 teclas;
- `IN 3` ACC ← 1 quando uma nova tecla for pressionada

- `OUT 0` visor simples ← ACC, no formato hexadecimal;
- `OUT 2` agrega o valor ASCII do ACC ao fim do banner de 16 letras.
- `OUT 3` o banner de 16 letras é limpo.

Escrevendo no visor o valor 10

```
LDA   #10h ; ACC = 10h
OUT   0    ; Mostra no visor o valor em hexa
HLT        ; Termina a execução
```

Escrevendo no visor o valor da variável X

```
    LDA   X      ; ACC = X
    OUT   0      ; Mostra no visor o valor de X em hexa
    HLT          ; Termina a execução

X:  DB    16     ; Variável X inicialmente com 16 (10 hexa)
```

Lendo chaves, escrevendo no visor

```
ESPERA_CHAVES:
        IN      1       ; Fica em loop de status verificando
        AND     #1      ; se tem valor novo nas chaves (bit 0)
        JZ      ESPERA_CHAVES

        IN      0       ; Faz a leitura das chaves
        OUT     0       ; Coloca o valor no visor (em hexa)
        JMP LACO
```

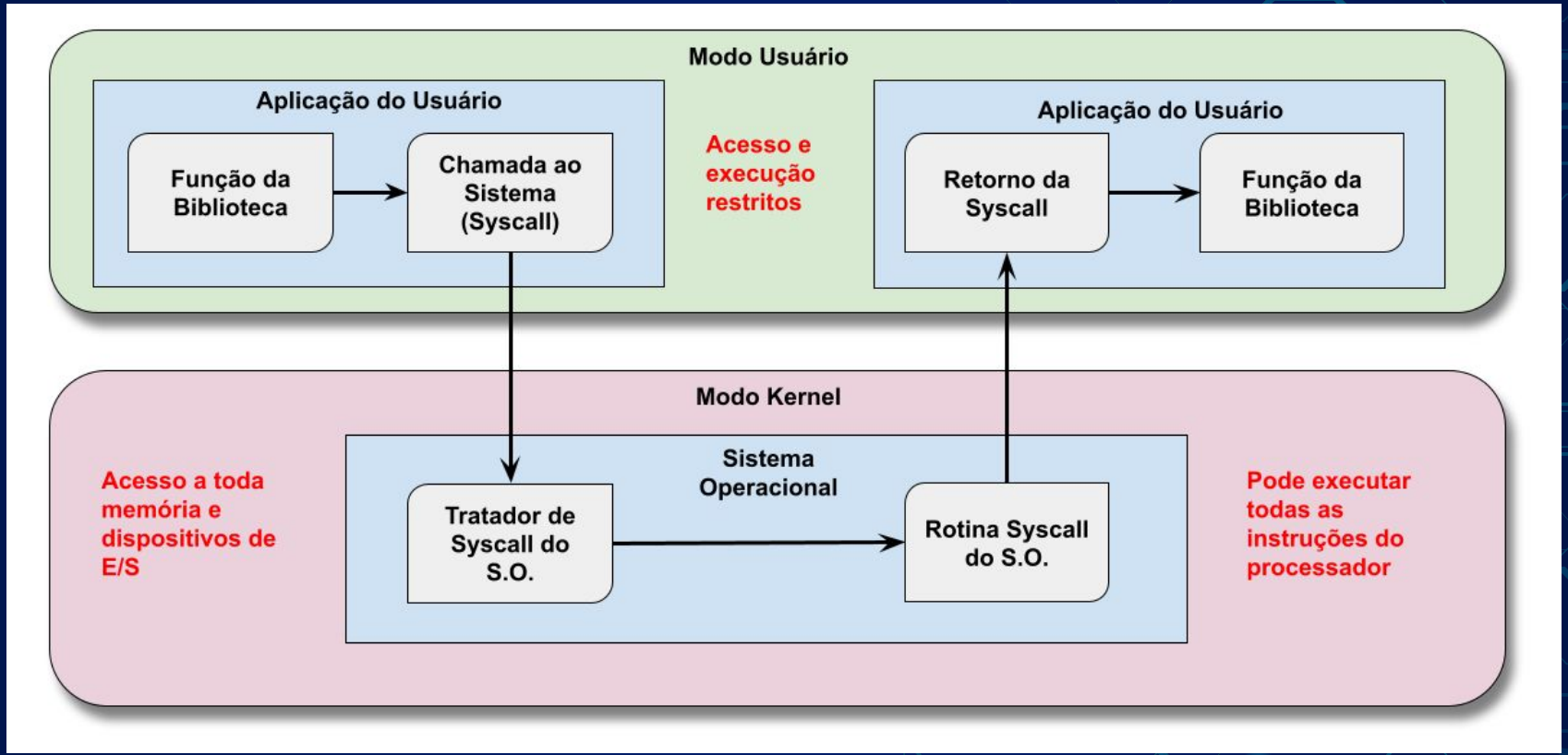
Contador regressivo

```
INICIO:
    IN      1       ; Fica em loop de status verificando
    SUB     #1      ; se tem valor novo nas chaves
    JN      INICIO
    IN      0       ; Faz a leitura das chaves
LOOP:
    OUT     0       ; Coloca o valor no visor (em hexa)
    SUB     #1      ; Subtrai um do valor
    JN      FIM     ; Se já chegou a zero termina
    JMP     LOOP
FIM:                ; Termina o programa
    HLT
    END INICIO
```

Nota: no simulador execute passo-a-passo, pois este código é executado muito rápido no modo normal.

Instrução TRAP

- O processador Sapiens possui uma instrução (TRAP) para realizar operações de E/S mais sofisticada.
- Na realidade são emuladas pelo simulador, de forma similar ao que acontece nos sistemas operacionais reais (às vezes chamado de System Call - SYSCALL).


Modo Usuário
Aplicação do Usuário
Função da Biblioteca
Chamada ao Sistema (Syscall)
Acesso e execução restritos
Aplicação do Usuário
Retorno da Syscall
Função da Biblioteca
Modo Kernel
Acesso a toda memória e dispositivos de E/S
Sistema Operacional
Tratador de Syscall do S.O.
Rotina Syscall do S.O.
Pode executar todas as instruções do processador

Funcionamento geral dos TRAPS

- O acumulador deve conter o código (número) da função desejada.
- A instrução TRAP tem um operando apenas: o endereço de uma área de memória que contém as informações que serão processadas, com frequência terminada em zero.
- Uso típico:

```
        LDA    #4      ; 4 - cód. função “escrever na
console”
        TRAP   info
        …
        …
info:   STR    “oi, teste”
        DB 0
```

TRAPS básicos do Simus (veja detalhes no livro)

1 – ACC ← Lê de um caractere da console (em ASCII).
2 – escreve na console uma letra.
3 – lê uma linha inteira da console.
4 – escreve uma linha inteira na console.
5 – Rotina de temporização em ms.
6 – Toca um tom.
7 – Retorna um número pseudo-aleatório entre 0 e 99 no ACC.
8 – Semente inicial da rotina de número aleatórios.

Ex.: imprime na console letras de A-Z

```
ORG 0
        LDA #41h
        STA LETRA
        LDA #26
        STA CONTA
LOOP:                   ; Fica em loop até conta = 0
        LDA #2
        TRAP LETRA      ; Imprime o caractere na console
;
        LDA LETRA       ; Avança para a letra seguinte
        ADD #1
        STA LETRA
;
        LDA CONTA       ; Atualiza o contador
        SUB #1
        STA CONTA
        JNZ LOOP
;
        HLT
```

```
ORG 100
LETRA: DS  1
CONTA: DB  26
END 0
```

7.9 ACESSANDO UM VETOR

Indexando os elementos de um vetor

- Um vetor (array) é um conjunto de posições de memória, contendo informações, que são associadas a um [índice].
  - Uso comum em linguagens de programação: vet[i]
- O Sapiens permite o acesso indireto a posições de memória, ou seja, através de uma variável que contém um endereço de memória, que chamaremos de ponteiro.
  - Na linguagem de montagem do Sapiens, isso é expresso colocando-se o caractere @ antes do operando da instrução.
  - O uso de acesso indireto facilita a manipulação de vetores (arrays), ajudando a caminhar sobre seus elementos ou mesmo calcular o valor de um índice específico.

Exemplo de uso

- Suponha um vetor de 4 posições armazenado na posição 100h de memória, e com um ponteiro para seu início:

```
ORG 100h
vetor:      DB      10, 30, 50, 80
pont:       DW      vetor+2         ; opcionalmente DW 102h

ORG 0
            LDA     @pont           ;  ACC = @pont
            OUT     0               ; coloca no display o valor 50 (vetor[2])
            HLT                     ;
            END     0
```

Movimentando o ponteiro

- Como ponteiro tem 16 bits, então mover o ponteiro, significa somar 1, o que tem que ser feito em 16 bits.
  - Na ordenação “little endian”, os 8 bits da parte baixa vem antes da parte alta

```
LDA     pont
ADD     #1          ; soma 1 à parte baixa
STA     pont
LDA     pont+1
ADC     #0          ; soma 0 ou 1 (depende do “vai 1” )
STA     pont+1
```

Exibindo um vetor no display

```
; ----- Variáveis ----
ORG 0200h
vetor:  DB   10h, 30h, 25h, 45h, 22h     ; Vetor a exibir
conta:  DB   5            ; Tamanho do vetor
pont:   DW   vetor        ; Ponteiro móvel para o vetor

display EQU 0
```

... continua

```
ORG 0
inicio: LDA  @pont     ; Coloca no ACC um elemento do vetor
        OUT  display   ; Mostra no display
;
        LDA  pont      ; Incrementa o apontador (parte baixa)
        ADD  #1
        STA  pont
        LDA  pont+1    ; Incrementa a parte alta se deu carry
        ADC  #0
        STA  pont+1
;
        LDA  conta     ; decrementa o contador
        SUB  #1
        STA  conta     ;
        JNZ  inicio    ; Se não chegou a zero, repete
;
        HLT            ; Termina a execução
        END  inicio
```

```
; Obs.: por simplicidade, neste programa nós mexemos no ponteiro e
; no contador,  que vêm pré-carregados. Não os reiniciamos depois.
; Assim, se formos executá-lo ; duas vezes, teremos que tomar o
; cuidado de reiniciá-los ou compilar novamente o programa.
; A seguir um código para reiniciar

        LDA  #00h       ; carrega o ponteiro, parte baixa e alta com 0200h
        STA  pont
        LDA  #02h
        STA  pont+1
;
        LDA  #5         ; carrega o contador com 5
        STA  conta
```

Uso mais prático de ponteiros

- O procedimento mostrado anteriormente pode ser muito simplificado, se considerarmos algumas restrições de uso:
  - Limitar os vetores a 256 posições
  - Posicionar vetores sempre em endereços múltiplos de 256 (ou seja, usar ORG com o valor terminado em 00h

```
ORG  0100h     ; ou 0200h, ou 0300h, ou 0400h, etc..
vetor: db      10h, 30h, 25h, 45h, 22h
pont:  dw      vetor
```

- Desta forma, para indexar ou caminhar, basta alterar o primeiro byte dele. Por exemplo, localizar o índice 2 deste vetor.

```
        lda    #2      ; índice 2
        sta    pont    ; alteramos o ponteiro com este índice
        lda    @pont   ; fácil!  pegamos o vetor[2]
```

Versão simplificada deste programa

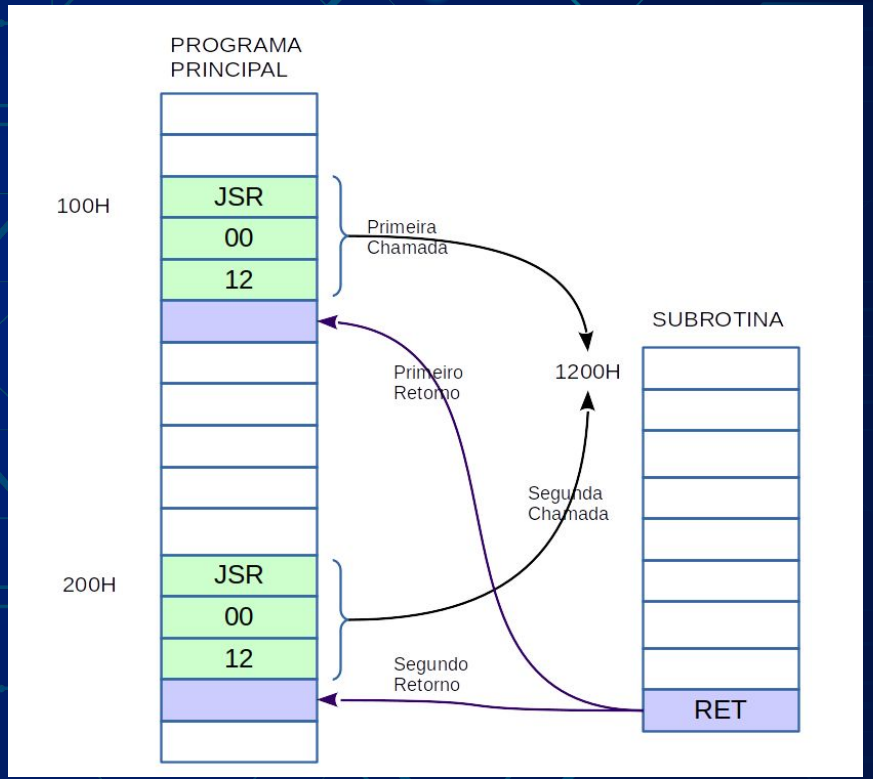
```
        LDA   #0        ; inicia o ponteiro
        STA   pont
        LDA   #5        ; inicia o contador
        STA   conta
inicio:
        LDA   @pont     ; Carrega no ACC um elemento do vetor
        OUT   display   ; Mostra no display
;
        LDA   pont      ; Incrementa o apontador (só parte baixa)
        ADD   #1
        STA   pont
;
        LDA   conta     ; decrementa o contador
        SUB   pont
        JNZ   inicio    ; Se não chegou ao fim, repete
;
        HLT             ; Termina a execução
        END   0
```

7.10 USO DE SUBROTINAS E DA PILHA

Subrotinas

- Sub-rotinas são trechos de código que podem ser chamados de diversos pontos do programa principal.
  - Ao fim deste código, pode-se retornar ao ponto chamador.

- No Sapiens, a instrução JSR (jump to subroutine) faz o desvio de ida, e a instrução RET o desvio de volta.



Pilha (stack) para controle da chamada

- Para produzir o efeito de chamada e retorno, é necessário guardar o endereço de execução atual, que será usado para retornar para o endereço que chamou.
- A maioria dos processadores guarda os endereços de retorno na pilha, para permitir que uma chamada possa ser executada de dentro de outra chamada.
  - Um registrador do processador chamado SP (stack pointer) indica a posição da memória onde está o “topo” da pilha.
  - Este registrador é alterado a cada JSR e a cada RET, inserindo e removendo um elemento da pilha.
  - É possível usar a mesma pilha para transmitir parâmetros para as rotinas e resultados de funções.

Mecânica das chamadas

- Ao executarmos a instrução de chamada de sub-rotina (JSR), o endereço de retorno, que é o endereço da instrução imediatamente após a instrução JSR, é automaticamente salvo na pilha, no endereço indicado pelo apontador de pilha (SP).
- Quando a sub-rotina é concluída, a instrução RET é executada. Ela retira o endereço de retorno da pilha, no endereço indicado pelo apontador de pilha (SP), copiando-o de volta para o apontador de instruções (PC).
    - Numa pilha, o último valor a ser inserido é o primeiro a ser retirado.
    - Por razões históricas, a pilha cresce no sentido oposto da memória, ou seja, dos endereços mais altos para os mais baixos.

7.11 USOS ALTERNATIVOS PARA O APONTADOR DE PILHA

Instruções para manipulação da pilha no Sapiens

- Como vimos, o SP indica a posição do topo da pilha e tem 16 bits.
- A pilha cresce diminuindo os endereços.

LDS	ender	usa o parâmetro para alterar o SP
STS	ender	guarda o SP na memória
JSR	rotina	Coloca o endereço seguinte do programa (PC+3) na pilha, decrementa SP e desvia para a rotina
RET		Armazena no PC o topo da pilha e incrementa SP (ou seja retorna de uma subrotina)
PUSH		armazena o ACC no topo da pilha e decrementa SP
POP		insere um byte no topo da pilha e incrementa SP

Passagem de parâmetros

- As funções PUSH e POP são com frequência usados para salvar o ACC e carregar no ACC valores da pilha, para enviar e receber dados de subrotinas.

Ex.: chamar uma rotina com dois parâmetros, ou seja rotina(p1,p2)

```
...
LDA    valor1     ; coloca na pilha o primeiro parâmetro
PUSH
LDA    valor2     ; coloca na pilha o segundo parâmetro
PUSH
JSR    rotina     ; chama a rotina
...
```

Passagem de parâmetros

- Para receber os parâmetros, precisamos fazer no início da rotina:
  - Como no topo da pilha está o endereço de retorno, devemos salvá-lo em uma outra posição de memória;
  - Depois está o segundo parâmetro, que armazenamos em uma variável local;
  - Depois está o primeiro parâmetro, que também salvamos;
  - Antes de retornar, salvamos novamente na pilha o endereço de retorno.

```
rotina:
    POP                     ; endereço de retorno com 16 bits
    STA    salva1           salva: DW    1
    POP                     ; primeiro parâmetro
    STA    salva1+1         p1:    DS    1
    POP                     ; segundo parâmetro
    STA    p2               p2:    DS    1
    POP
    STA    p1
    LDA    salva1+1
    PUSH
    LDA    salva1
    PUSH
     …   resto da rotina
     …
    RET
```

Usos não convencionais do SP

- O SP é usado geralmente como controle da pilha.
- Porém no Sapiens, é um registrador que pode ser escrito e lido com valores de 16 bits, o que permite seu uso mais geral:
  - Ele pode ajudar a lidar com valores de 16 bits com maior rapidez, por exemplo, na transferência de valores entre variáveis.

```
A:  DW      10
B:  DS      2
    ...
    LDS     A       ; SP = A
    STS     B       ; B = A
```

Não se deve, entretanto, usar este procedimento dentro de rotinas, pois o endereço de retorno seria destruído! (seria necessário salvar o SP antes)

Obrigado !

Arquitetura de Computadores
Uma Introdução

Mais recursos em:
https://simulador-simus.github.io

CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, and infographics & images by Freepik.

Please keep this slide for attribution.


slidesgo