

Revisão do NEANDER

Prof. Sérgio L. Cechin

Características

- Largura de dados e endereços: 8 bits
 - Capacidade de endereçamento: 256 bytes
- Dados representados em complemento de 2
 - A “interpretação” dos grupos de bits é a de complemento de 2
 - Esta “interpretação” afeta os cálculos efetuados pela ULA e os códigos de condição gerados

Características

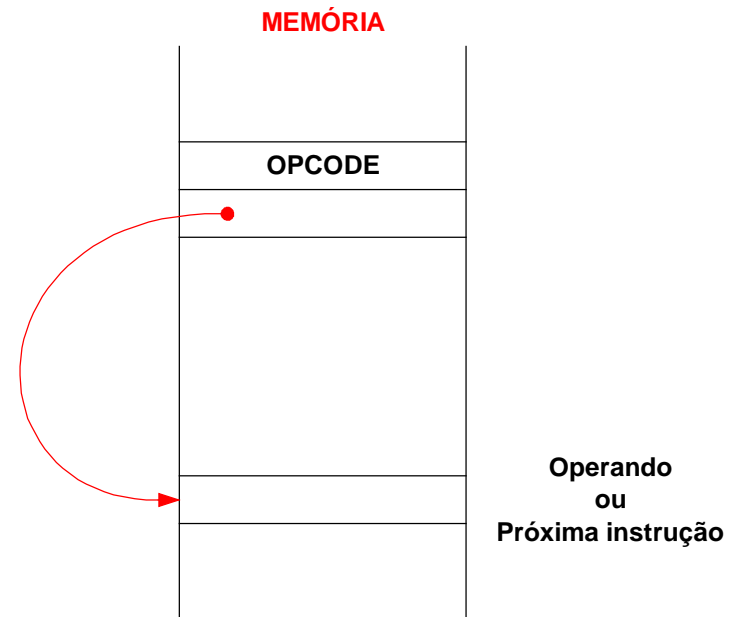
- 4 Registradores
 - Acumulador (AC)
 - Apontador de programa (PC)
 - Program Counter
 - Endereço da próxima instrução a ser buscada (fetch)
 - Registrador de estado
 - Armazena os códigos de condição
 - N: ligado quando o resultado for negativo
 - Z: ligado quando o resultado for zero
 - Registrador de instrução (RI)
 - Usado como memória temporária para a instrução que está sendo executada

Fases de operação

- Ciclo de busca (fetch)
 - Busca o byte no endereço indicado pelo PC
 - $RI \leftarrow MEM(PC)$
 - $PC \leftarrow PC+1$
 - Este byte será interpretado como uma instrução
- Ciclo de execução
 - Decodificação da instrução
 - Execução propriamente dita

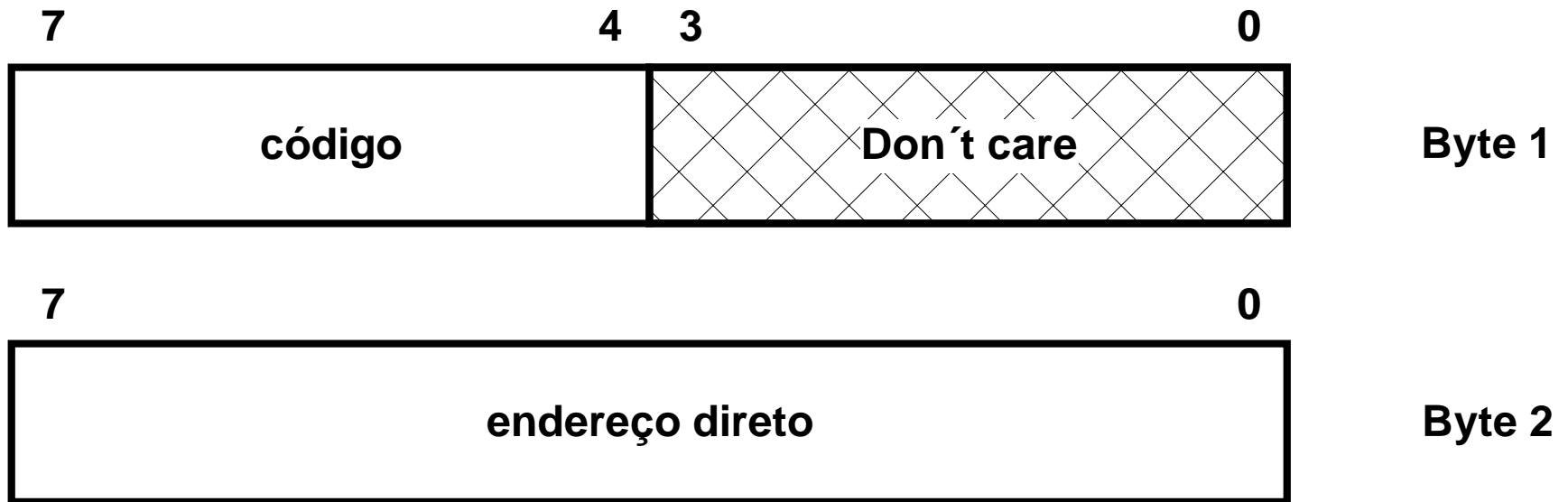
Modos de endereçamento

- Apenas um!
- Modo direto



Codificação das Instruções

- Formadas por 1 ou 2 bytes



Instruções NEANDER

Código	Instrução	Operação	N	Z	Descrição
0000 xxxx (00)	NOP	nenhuma operação			nenhuma operação
0001 xxxx (10)	STA end	$MEM(end) \leftarrow AC$			armazena acumulador - (store)
0010 xxxx (20)	LDA end	$AC \leftarrow MEM(end)$	\updownarrow	\updownarrow	carrega acumulador - (load)
0011 xxxx (30)	ADD end	$AC \leftarrow AC + MEM(end)$	\updownarrow	\updownarrow	soma
0100 xxxx (40)	OR end	$AC \leftarrow AC \mid MEM(end)$	\updownarrow	\updownarrow	“ou” lógico
0101 xxxx (50)	AND end	$AC \leftarrow AC \& MEM(end)$	\updownarrow	\updownarrow	“e” lógico
0110 xxxx (60)	NOT	$AC \leftarrow !AC$	\updownarrow	\updownarrow	inverte (complementa) acumulador
1000 xxxx (80)	JMP dst	$PC \leftarrow dst$			desvio incondicional - (jump)
1001 00xx (90)	JN dst	if N=1 then $PC \leftarrow dst$			desvio condicional - (jump if negative)
1010 00xx (A0)	JZ dst	if Z=1 then $PC \leftarrow dst$			desvio condicional - (jump if zero)
1111 xxxx (F0)	HLT				término de execução - (halt)

Programação

Passos da programação

- Escrever o fluxograma/**algoritmo** a ser implementado
 - Pode-se utilizar linguagem “C” para isso
 - Escrever o **programa simbólico**
 - Listar as **variáveis e constantes** com os seus correspondentes endereços
 - **Codificar** o programa simbólico
-
- Escrever o programa no **simulador**
 - **Testar** o programa

Descrição Algorítmica

- Vamos usar uma “linguagem” para descrever os algoritmos a serem implementados
 - Sugestão: usar a linguagem “C” como referência
- Pode-se usar qualquer linguagem. As vantagens do “C”:
 - A linguagem é simples
 - Vocês conhecem a linguagem
 - É a linguagem de alto nível mais próxima do assembler
 - Pode-se “testar” os algoritmos, compilando e rodando a descrição em “C”

Uso da Linguagem “C”

- Não devem ser usadas funções de biblioteca
 - Ex: funções de entrada (teclado) e saída (tela)
- Serão usados, apenas
 - Declaração de variáveis
 - Operações lógicas e aritméticas
 - Comandos condicionais (if, switch)
 - Comandos de controle de laço (for, while, etc)
- Não é necessário seguir, rigorosamente, a sintaxe do “C”
 - Pois o objetivo é representar o algoritmo (e não rodar o programa em “C”)
 - Se for necessário “rodar” o programa em “C”, pode-se recorrer a implementação de funções.

Enunciado

- Escrever um programa para calcular a diferença entre duas variáveis.
- Cada variável ocupa um byte.
- O resultado deverá ser armazenado em uma terceira posição de memória.
- **Problema:** não existe instrução de diferença!

Solução (passo 1)

Escrever o algoritmo

- $MC = MA - MB$
- MA, MB, e MC são bytes na memória
- Exemplo em “C”

```
unsigned char ma, mb, mc;
```

```
void main (void) {  
    mc = ma + (-mb);  
}
```

Solução (passo 2)

Escrever o programa simbólico

Codificação	Label	Mnemônico	parâmetros	Comentários
		LDA	MB	
		NOT		
		ADD	UM	$mc = ma + (-mb) ;$
		ADD	MA	
		STA	MC	
		HLT		

Solução (passo 3)

Listar variáveis e constantes

- Variáveis

- MA = endereço 080H

- &MA = 0x80

- MB = endereço 081H

- &MB = 0x81

- MC = endereço 082H

- &MC = 0x82

`unsigned char ma, mb, mc;`



- Constantes

- UM = endereço 083H

- &UM = 0x83 (com 0x01)

Solução (passo 4)

Codificar

Codificação	Label	Mnemônico	parâmetros	Comentários
20 81		LDA	MB	
60		NOT		
30 83		ADD	UM	
30 80		ADD	MA	
10 82		STA	MC	
F0		HLT		

Daedalus!

- Vamos usar o montador Daedalus
 - O montador realiza os passos 3 e 4 anteriores
- Entretanto, é necessário informar ao montador onde estão as variáveis
- No exemplo anterior, seria:

	ORG	H80
MA:	DB	0
MB:	DB	0
MC:	DB	0
UM:	DB	H01

Resultado final

```
org      h80

MA:      DB      0           ; unsigned char ma, mb, mc;
MB:      DB      0
MC:      DB      0
UM:      DB      h01 ← Necessária a const. 01!

org      h00           ; void main() {
lda      MB           ;      mc = ma + (-mc);
not
add      UM
add      MA
sta      MC

hlt      ; }
```

Exercício 2

- Faça um programa para escrever 000H em uma área de memória.
- O endereço inicial da área e o seu tamanho são representados por valores com 8 bits.
- O endereço de início está armazenado no endereço 080H.
- O tamanho da área está armazenado no endereço 081H.
- **Problema:** a solução requer acesso à vetores!

Solução (em pseudo “C”)

```
Index = START;  
While (SIZE != 0) {  
    MEM [Index] = 0;  
    Index = Index + 1;  
    SIZE = SIZE - 1;  
}
```

Como se implementa um “while” em assembler????

Removendo o “while”

(e outras coisas mais: “++” e “--”)

```
Index = START;  
While (SIZE != 0) {  
    MEM [Index] = 0;  
    Index = Index + 1;  
    SIZE = SIZE - 1;  
}  
Halt
```

Solução sem goto

```
Index = START;  
Loop:  
If (SIZE == 0) goto FIM  
    MEM [Index] = 0;  
    Index = Index++;  
    SIZE = SIZE--;  
    goto Loop  
Fim:  
Halt
```

Solução com goto

Acesso a vetores

- Acesso da forma “MEM [Index]”
- É necessário usar alteração de código
 - STA *EndereçoDeInstrução*
- Exemplo:
 - $A = \text{MEM} [\text{Index}]$

	LDA	Index
	STA	INST+1
INST:	LDA	0

Solução – Programa Simbólico

```
org      h00

lda      START          ; Index = START;
sta      INST+1

LOOP:    ; LOOP:
lda      SIZE            ; if (SIZE==0) goto FIM;
jz       FIM

lda      ZERO            ; MEM[Index] = 0;
INST:    sta      0

lda      INST+1          ; Index++;
add      UM
sta      INST+1

lda      SIZE            ; SIZE--;
add      MENOS_UM
sta      SIZE

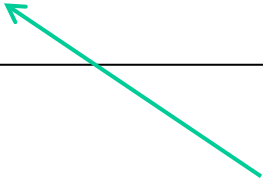
jmp      LOOP            ; goto LOOP

FIM:
hlt
```

Solução – Variáveis.

```
        org      h80
START:   db      0           ; unsigned char start;
SIZE:    db      0           ; unsigned char size;

ZERO:    db      h00
UM:      db      h01
MENOS_UM: db      hff
```

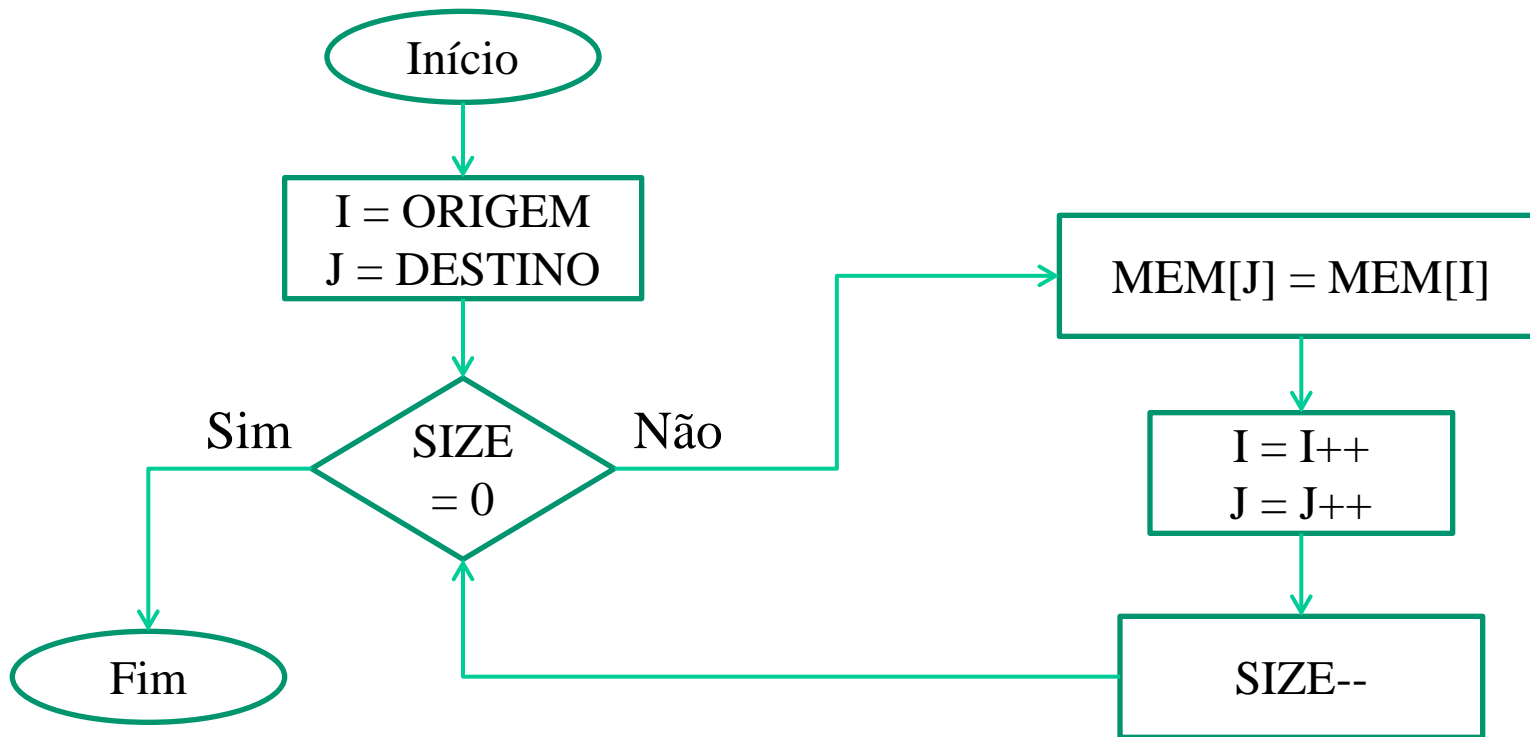


Constantes necessárias
para o assembler!

Exercício 3

- Escrever um programa para mover os dados de uma área da memória para outra
- O endereço de início da área origem está armazenado no endereço 0FDH.
- O endereço de início da área destino está armazenado no endereço 0FEH.
- O número de bytes a serem movidos está armazenado no endereço 0FFH.
- OBS: Não é necessário verificar nenhuma consistência dos endereços (áreas sobrepostas, por exemplo)

Algoritmo (fluxograma)



Codificação em pseudo-“C”

```
unsigned char i;
unsigned char j;
unsigned char mem[256]; // representa toda a memória do RAMSES

void main () {

    i = origem;
    j = destino;

    while (size) {
        mem[j] = mem[i];
        i++;
        j++;
        size--;
    }
}
```

Variáveis

```
                org      hFB
UM:             db       h01      ; constante 1
MENOS_UM:       db       hFF      ; constante -1

ORIGEM:         db       0        ; unsigned char origem;
DESTINO:        db       0        ; unsigned char destino;
SIZE:          db       0        ; unsigned char size;
```

Codificação

```
mem:    org      h00                ; void main () {
;                               ; // representa toda a memória do RAMSES

        lda      ORIGEM            ;      i = origem;
        sta      mi+1

        lda      DESTINO           ;      j = destino;
        sta      mj+1

LOOP:
        lda      SIZE              ;      while (size) {
        jz       FIM

mi:      lda      0                ;                      mem[j] = mem[i];
mj:      sta      0

        lda      mi+1              ;                      i++;
        add      UM
        sta      mi+1
```

Codificação

```
    lda    mj+1      ;          j++;
    add    UM
    sta    mj+1

    lda    size      ;          size--;
    add    MENOS_UM
    sta    size

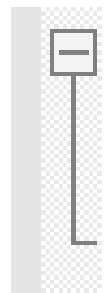
    jmp    LOOP      ;      }

FIM:
    hlt           ; }
```

Exercício para casa!

Exercícios (1)

- Codifique para o NEANDER o seguinte algoritmo
- Considere que:
 - As variáveis “a” e “b” são inteiros, sem sinal, de 8 bits
 - A variável “a” está no endereço H80
 - A variável “b” está no endereço H81



The diagram shows a vertical stack frame with a pointer box on the left. To the right of the pointer is the code: `if (a>b) {` on the first line, `a = a-b;` on the second line, and `}` on the third line. A vertical line connects the pointer box to the opening curly brace of the if statement.

```
if (a>b) {  
    a = a-b;  
}
```


Exercícios (2)

- Codifique para o NEANDER o seguinte algoritmo
- Considere que:
 - A variável “s” é um inteiro, sem sinal, de 16 bits
 - A variável “s” está nos endereços H80 e H81
 - A parte mais significativa está no endereço H80

```
s=0;  
while (s!=10) {  
    s++;  
}
```

Exercícios (3)

- Codifique para o NEANDER o algoritmo abaixo
- Considere que:
 - As variáveis “s” e “i” são inteiros, sem sinal, de 8 bits.
 - A variável “s” está no endereço H80
 - A variável “i” está no endereço H81

```
s=0
for (i=0; i<10; ++i) {
    s=s+i;
}
```

Exercícios (4)

- Codifique para o NEANDER o seguinte algoritmo
- Considere que:
 - As variáveis “i”, “j”, “a” e o vetor “v[]” são inteiros, sem sinal, de 8 bits
 - A variável “i” está no endereço H80
 - A variável “j” está no endereço H81
 - A variável “a” está no endereço H82
 - O vetor “v[]” ocupa os endereços H83 até H8F

```
a=v[i];  
v[i]=v[j];  
v[j]=a;
```

Revisão do NEANDER

Prof. Sérgio L. Cechin