Aula 5 - Transferência de Dados, Enderecamento e Aritmética
Arquitetura de Computadores I

Prof. MSC. Wagner Guimarães Al-Alam

Universidade Federal do Ceará
Campus de Quixadá

2017-1

Agenda

- Instruções de Transferência de Dados
- Adição e Subtração
- Operadores e diretivas relacionadas aos dados
- Endereçamento indireto
- Instruções JMP e LOOP
- Programação em 64-Bit

Instruções de Transferência de Dados

- Tipos de operandos
- Notação dos operandos de instruções
- Operandos de acesso direto na memória
- Instrução MOV
- Extensão de Zeros & Sinal
- Instrução XCHG
- Instruções Direct-Offset

Tipos de operandos

- Imediato - uma constante inteira (8, 16, ou 32 bits)
  - o valor é codificado na instrução
- Registrador - o nome de um registradort
  - registrador é o nome convertido em um número e codificado com a instrução
- Memória - referência a localização na memória
  - o endereço de memória é condificado na instrução ou um registrador mantém o endereço da localização na memória

Instruction Operand Notation

Operand	Description
reg8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
reg16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
reg32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	Any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
reg/mem8	8-bit operand, which can be an 8-bit general register or memory byte
reg/mem16	16-bit operand, which can be a 16-bit general register or memory word
reg/mem32	32-bit operand, which can be a 32-bit general register or memory doubleword
mem	An 8-, 16-, or 32-bit memory operand

Operandos Diretos na Memória

- Um operando direto na memória é o nome para referenciar armazenamento na memória
- A referencia através de nome (label) é automaticamente desreferenciado pelo montador

```
1   %include "io.inc"
2
3   section .data
4   var1 DB 10h
5
6   section .text
7   global CMAIN
8   CMAIN:
9       mov al, [var1]          ; AL = 10h NASM
10      xor eax, eax
11      ret
```

Instrução MOV

- Move a origem para o destino. Sintaxe:
  MOV destino, origem
- Não é permitido mais de um operando de memória
- CS, EIP, e IP não podem ser o destino
- Não é permitido movimentações de imediatos a segmento

```
1   %include "io.inc"
2
3   section .data
4       count DB 100
5       wVal  DW 2
6
7   section .text
8   global CMAIN
9   CMAIN:
10      mov ebp, esp; for correct debugging
11      mov bl, [count]
12      mov ax, [wVal]
13      mov [count], al
14      mov al,[wVal]     ; erro - altera 1 byte (perde 8 maiores bits de wVal)
15      mov ax,[count]    ; erro - altera somente os 8 menores bits de ax
16      mov eax,[count]   ; erro - altera somente os 8 menores bits de ax
17      xor eax, eax
18      ret
```

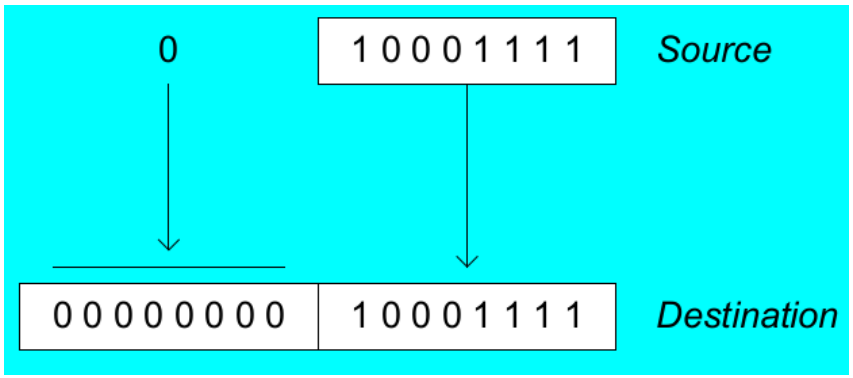
Exemplos

Explique por que cada uma das seguintes instruções MOV é inválida

```
1   %include "io.inc"
2
3   section .data
4   bVal DB    100
5   wVal DW    2
6   dVal DW    5
7
8   section .bss
9   bVal2 RESB 1
10
11  section .text
12  global CMAIN
13  CMAIN:
14  mov ebp, esp; for correct debugging
15
16  mov ds,45                ;nao e permitida movimentacao de valor imediato...
17                           ;...para registrador DS
18  mov esi,[wVal]           ;esi tem 32 bits e wVal 8 bits
19  mov eip,[dVal]           ;eip nao pode ser o destino
20  mov 25,[bVal]            ;valor imediato nao pode ser o destino
21  mov [bVal2],[bVal]       ;nao e permitido mov memoria-memoria
22
23  xor eax, eax
24  ret
```

Extensão de Zeros

Quando você copia um valor menor a um destino maior (dobro), a instrução MOVZX completa a parte mais significativa com zeros.



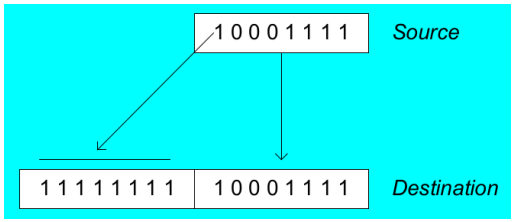

```
1     mov bl,10001111b
2     movzx ax,bl          ; extensao de zeros
```

Observação

O destino deve ser um registrador.

Extensão de Sinal

A instrução MOVSX completa a metade superior do destino com o sinal




```
1    mov bl,10001111b
2    movsx ax,bl          ; extensao com sinal
```

Observação

O destino deve ser um registrador.

Instrução XCHG

XCHG exporta valores entre dois operandos. Ao menos um dos operandos deve ser um registrador e não são permitidos valores imediatos.

```
1     section .data
2         var1 DW   1000h
3         var2 DW   2000h
4     section .text
5         xchg ax, bx           ;exchange 16-bit regs
6         xchg ah, al           ;exchange 8-bit regs
7         xchg [var1], bx       ;exchange mem, reg
8         xchg eax, ebx         ;exchange 32-bit regs
9         xchg [var1], [var2]   ;error: two memory operands
```

Operandos Direct-Offset - 1/2

Um offset constante é adicionado a uma etiqueta de dados para produzir um endereço efetivo. O endereço é desreferenciado para pegar o valor em sua posição de memória.

```
1    section .data
2        arrayB DB 10h,20h,30h,40h
3    section .text
4        mov al,[arrayB+1]
```

Pergunta

Por que **[arrayB+1]** não produz 11h?

Operandos Direct-Offset - 2/2

Um offset constante é adicionado a uma etiqueta de dados para produzir um endereço efetivo. O endereço é desreferenciado para pegar o valor em sua posição de memória.

```
1 section .data
2     arrayW DW 1000h,2000h,3000h
3     arrayD DD 1,2,3,4
4 section .text
5     mov ax,[arrayW+2]   ; AX = 2000h
6     mov ax,[arrayW+4]   ; AX = 3000h
7     mov eax,[arrayD+4]  ; EAX = 00000002h
```

Pergunta

O código das seguintes linhas irão compilar?
mov ax,[arrayW-2] ; ??
mov eax,[arrayD+16] ; ??

O que acontece quando elas executam?

Sua Vez. . .

1. Escreva um programa que rearrange os valores de três doubleword do seguinte array como: 3, 1, 2.
   .data
   arrayD DD 1,2,3
2. Passo1: Copie o primeiro valor em EAX e troque EAX com o valor da segunda posição.
   mov eax,[arrayD]
   xchg eax,[arrayD+4]
3. Passo 2: Troque EAX com o terceiro valor do array e copie o valor de EAX para a primeira posição do array.
   xchg eax,[arrayD+8]
   mov [arrayD],eax

Avalie Isso . . . - 1/2

1. Queremos escrever um programa que soma os 3 seguintes bytes:
   .data
   myBytes DB 80h,66h,0A5h
2. Qual a sua avaliação do codigo que segue?
   mov al,[myBytes]
   add al,[myBytes+1]
   add al,[myBytes+2]
3. Qual a sua avaliação do codigo que segue?
   mov ax,[myBytes]
   add ax,[myBytes+1]
   add ax,[myBytes+2]
4. Alguma outra possibilidade?

Avalie Isso . . . - 2/2

```
1   section .data
2       myBytes DB 80h,66h,0A5h
```

- Sobre o seguinte código:
  Algo está faltando?

```
1   section .text
2       movzx ax, [myBytes]
3       mov bl,[myBytes+1]
4       add ax,bx
5       mov bl,[myBytes+2]
6       add ax,bx
```

Sim: Mover zero para BX antes da instrução MOVZX.

Adição e Subtração

- Instruções INC e DEC
- Instruções ADD e SUB
- Instrução NEG
- Implementando Expressões Aritméticas
- Flags afetadas por aritmética
  - Zero
  - Sign
  - Carry
  - Overflow

Instruções INC e DEC

- Soma 1, subtrai 1 do operando de destino
  - operando deve ser um registrador ou memória
- INC destino
  - Lógica: destino $\longleftarrow$ destino $+ 1$
- DEC destino
  - Logica: destino $\longleftarrow$ destino - 1

Exemplos - INC and DEC

```
1   %include "io.inc"
2
3   section .data
4       myWord dw 1000h
5       myDword dd 10000000h
6
7   section .text
8   global main
9   main:
10      mov ebp, esp                ; for correct debugging
11
12      inc byte [myWord]           ; 1001h
13      dec byte [myWord]           ; 1000h
14      inc dword [myDword]         ; 10000001h
15      mov ax,00FFh
16      inc ax                      ; AX = 0100h
17      mov ax,00FFh
18      inc al                      ; AX = 0000h
19
20      xor eax, eax
21      ret
```

Sua vez...

Mostre o valor do operando destino após cada uma das seguintes instruções executar:

```
1  %include "io.inc"
2
3  section .data
4      myByte DB 0FFh, 0
5
6  section .text
7  global main
8  main:
9      mov ebp, esp                ; for correct debugging
10
11     mov al,[myByte]             ; AL = FFh
12     mov ah,[myByte+1]           ; AH = 00h
13     dec ah                      ; AH = FFh
14     inc al                      ; AL = 00h
15     dec ax                      ; AX = FEFFh
16
17     xor eax, eax
18     ret
```

Instruções ADD e SUB

- ADD destino, origem
  - Logic: destino $\longleftarrow$ destino + origem
- SUB destino, origem
  - Logic: destino $\longleftarrow$ destino - origem
- MEsmas regras de operandos que a instrução MOV

Exemplos de ADD e SUB

```
1  %include "io.inc"
2
3  section .data
4      var1 DD 10000h
5      var2 DD 20000h
6
7  section .text
8  global main
9  main:
10     mov ebp, esp                ; for correct debugging
11                                 ;----EAX----
12     mov eax, [var1]             ;00010000h
13     add eax, [var2]             ;00030000h
14     add ax, 0FFFFh              ;0003FFFFh
15     add eax, 1                  ;00040000h
16     sub ax, 1                   ;0004FFFFh
17
18     xor eax, eax
19     ret
```

Instrução NEG (negar)

Inverte o sinal de um operando. O operando pode ser em um registrador ou em memória.
É necessário definir a quantidade de bits que será negada quando o operando for em memória.

```
1  %include "io.inc"
2
3  section .data
4      valB DB 1
5      valW DW 32767
6
7  section .text
8  global CMAIN
9  CMAIN:
10     mov ebp, esp              ; for correct debugging
11     neg byte [valB]           ; valB = -1
12     mov al,[valB]             ; AL = -1
13     neg al                    ; AL = +1
14     neg dword[valW]           ; valW = -32767
15
16     xor eax, eax
17     ret
```

Suponha que AX contém -32,768 e aplicamos NEG a ele, o resultado será válido?

Instrução NEG e as Flags

- O processador implementa NEG usando a operação interna que segue:
  - SUB 0,operando
- Qualquer operando não zero causa a definição do Carry flag.

```
1  section .data
2  valB DB 1,0
3  valC DB 128
4
5  section .text
6  neg byte [valC]              ; valC = -128
7  neg byte [valB]              ; CF = 1, OF = 0
8  neg byte [valB + 1]          ; CF = 0, OF = 0
9  neg byte [valC]              ; CF = 1, OF = 1
```

Implementando Expressões Aritméticas

- Compiladores HLL (*High-level language*) traduzem expressões matemáticas em linguagem assembly. Você também consegue fazer isso. Por exemplo:
  - Rval = -Xval + (Yval - Zval)

```
1   section .data
2       Xval DD 26
3       Yval DD 30
4       Zval DD 40
5
6   section .bss
7       Rval RESD 1
8
9   section .text
10      mov eax,[Xval]
11      neg eax                 ; EAX = -26
12      mov ebx,[Yval]
13      sub ebx,[Zval]          ; EBX = -10
14      add eax,ebx
15      mov [Rval],eax          ; -36
```

Sua Vez...

- Traduza a seguinte expressão em linguagem assembly.
  Não peremita que Xval, Yval, ou Zval possa ser modificado:
  - Rval = Xval - (-Yval + Zval)
- Assuma que todos os valores são doublewords com sinal.

Sua Vez... Rta.

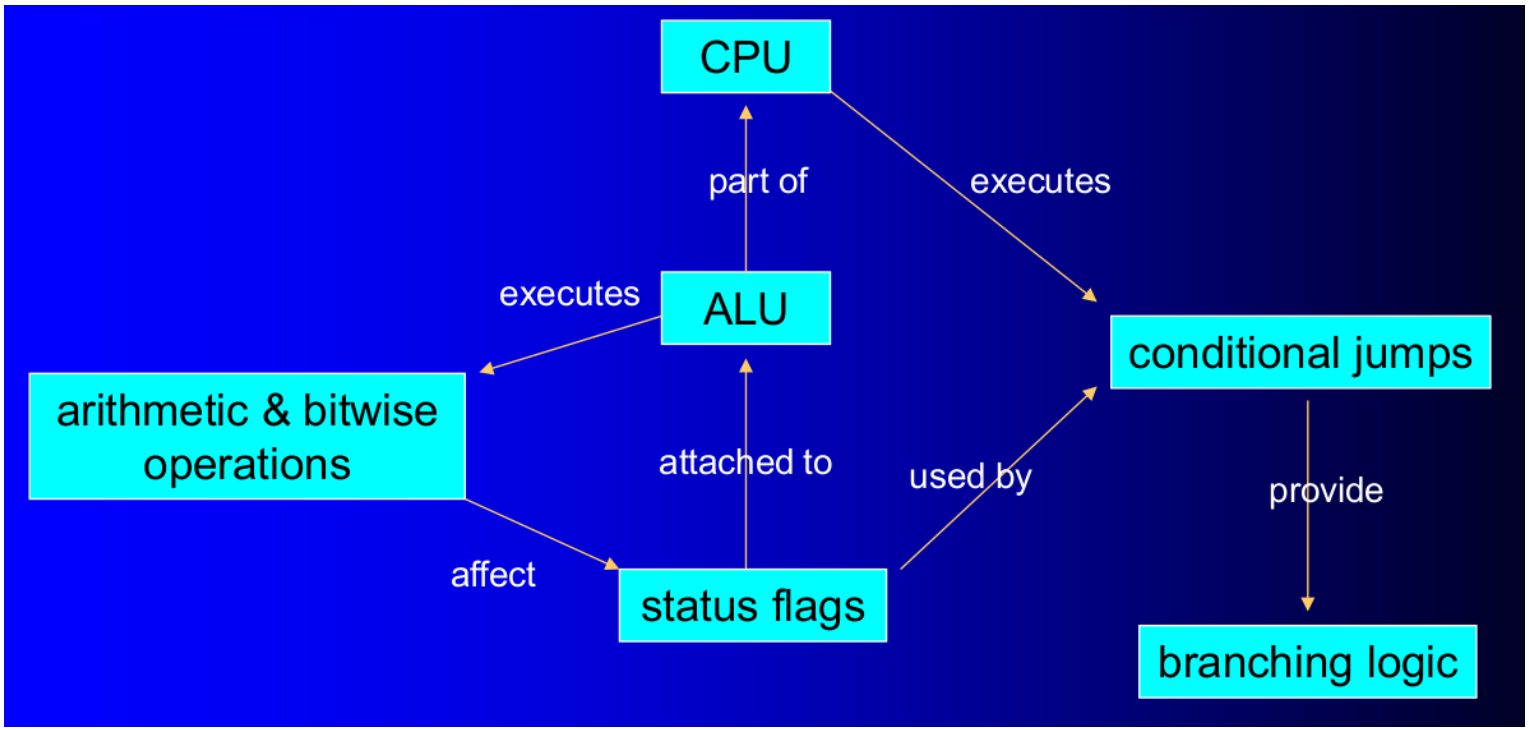
- Traduza a seguinte expressão em linguagem assembly.
  Não peremita que Xval, Yval, ou Zval possa ser modificado:
  - Rval = Xval - (-Yval + Zval)
- Assuma que todos os valores são doublewords com sinal.

```
1    mov ebx,[Yval]
2    neg ebx
3    add ebx,[Zval]
4    mov eax,[Xval]
5    sub eax,ebx
6    mov [Rval],eax
```

Flags Afetadas por Aritmética

- A ULA tem um número de flags que refletem o resultado de uma operação atitmética (bit-a-bit)
  - baseadas no conteúdo do operando de destino
- Flags essenciais:
  - Zero flag - define quando o destino é igual a zero
  - Sign flag - define quando o destino é negativo
  - Carry flag - define quando um valor sem sinal está fora de sua faixa de valores possíveis
  - Overflow flag - define quando um valor com sinal está fora da faixa de valores possíveis
- A instrução MOV jamais afeta as flags.

Mapa Conceitual



Observação

Você pode usar diagramas como esse para expressar os relacionamentos entre os conceitos da linguagem assembly.

Zero Flag (ZF)

- A Zero-flag é definida quando o resultado de uma operação produz zero no operando de destino.

```
1    mov cx,1
2    sub cx,1                ; CX = 0, ZF = 1
3    mov ax,0FFFFh
4    inc ax                  ; AX = 0, ZF = 1
5    inc ax                  ; AX = 1, ZF = 0
```

Lembre

- A flag está definida quando é igual a 1.
- A flag está limpa quando é igual a 0.

Sign Flag (SF)

- A FLAG definina é definida quando o operando de destino é negativo.
- A FLAG é zerada quando o operando destino é positivo.

```
1    mov cx,0
2    sub cx,1              ; CX = -1, SF = 1
3    add cx,2              ; CX = 1, SF = 0
```

- A flag de sinal é uma cópia do bit mais significante do destino:

```
1    mov al,0
2    sub al,1              ; AL = 11111111b, SF = 1
3    add al,2              ; AL = 00000001b, SF = 0
```

Inteiros Signed e Unsigned - Ponto de Vista do Hardware

- Todas isntruções de CPU operam exatamente de mesmo modo em inteiros com ou sem sinal
- A CPU não pode distinguir entre inteiros com ou sem sinal
- VOCÊ, o programador é unicamente responsável por usar o tipo de dado correto para cada tipo de instrução

Flags de Overflow e Carry - Ponto de Vista do Hardware

- Como a instrução ADD afeta as flags OF e CF:
  - CF = (carry out of the MSB)
  - OF = CF XOR MSB
- Como a instrução SUB afeta as flags OF e CF:
  - CF = INVERT (carry out of the MSB)
  - nega a origem e soma no destino
  - OF = CF XOR MSB

M

SB = Most Significant Bit (high-order bit)
XOR = eXclusive-OR operation
NEG = Negate (same as SUB 0,operand)

Carry Flag (CF)

- A flag de Carry é definida quando o resultado de uma operação gera um valor unsigned que está fora da faixa de valores válidos (muito grande ou muito pequeno para o operando de destino).

```
1    mov al,0FFh
2    add al,1                ; CF = 1, AL = 00
3    ; Tente ir abaixo do zero:
4    mov al,0
5    sub al,1                ; CF = 1, AL = FF
```

Sua vez . . .

- Para cada uma das seguintes instruções, mostre os valores do operando de destino e das flags Sign, Zero, e Carry:

```
1   mov ax,00FFh
2   add ax,1            ; AX= 0100h SF= 0 ZF= 0 CF= 0
3   sub ax,1            ; AX= 00FFh SF= 0 ZF= 0 CF= 0
4   add al,1            ; AL= 00h   SF= 0 ZF= 1 CF= 1
5   mov bh,6Ch
6   add bh,95h          ; BH= 01h   SF= 0 ZF= 0 CF= 1
7
8   mov al,2
9   sub al,3            ; AL= FFh   SF= 1 ZF= 0 CF= 1
```

Overflow Flag (OF)

- A flag de Overflow é definida quando o resultado com sinal é inválido ou fora da faixa de valores válidos (out of range).

```
1   ; Exemplo 1
2   mov al,+127
3   add al,1                ; OF = 1, AL = ??
4
5   ; Exemplo 2
6   mov al,7Fh              ; OF = 1, AL = 80h
7   add al,1
```

Observação

Os dois exemplos são idênticos no nível binário, porque 7Fh é igual a +127. Para determinar o valor do operando de destino é mais fácil calcular em hexadecimal.

Regra Prática

- Quando somar dois inteiros, lembre que a flag de Overflow somente é definida quando:
  - Dois operandos positivos são somados e sua soma é negativa
  - Dois operandos negativos são somados e a soma é positiva

Quais serão os valores da flag de Overflow?

```
1    mov al,80h
2    add al,92h          ; OF = 1
3
4    mov al,-2
5    add al,+127         ; OF = 0
```

Sua vez . . .

- Qua serão os valores das flags CF e OF após cada operação?

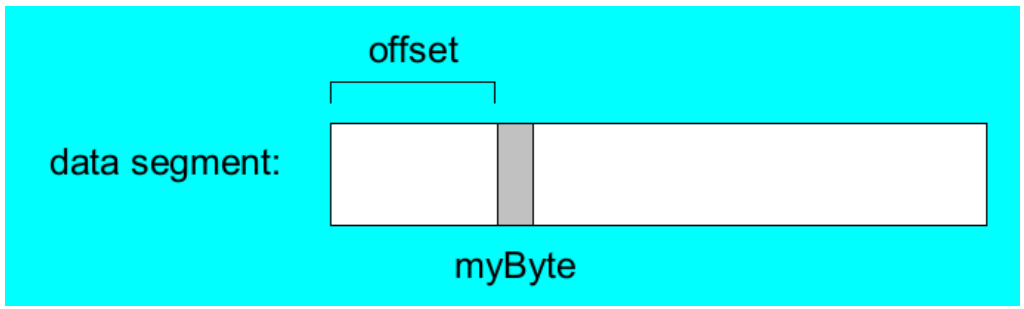
```
1   mov al,-128
2   neg al                  ; CF = 1 OF = 1
3   mov ax,8000h
4   add ax,2                ; CF = 0 OF = 0
5   mov ax,0
6   sub ax,2                ; CF = 1 OF = 0
7   mov al,-5
8   sub al,+125             ; CF = 0 OF = 1
```

Operadores e Diretivas Relacionados aos Dados

- Operador OFFSET
- Operador PTR
- Operador TYPE
- Operador LENGTHOF
- Operador SIZEOF
- Diretiva LABEL

Operador OFFSET

- OFFSET retorna a distância em bytes, de um label desde o início até seu segmento encerrado
  - Modo Protegido: 32 bits
  - Modo Real: 16 bits



Os programas do modo protegido que escrevemos utiliza um simples segmento (modelo de memória flat).

Exemplos de OFFSET

- Vamos assumir que o segmento de dados começa na posição 00404000h:

```
1  section .bss
2      bVal RESB 1
3      wVal RESW 1
4      dVal RESD 1
5      dVal2 RESD 1
6
7  section .text
8      mov esi, bVal      ; ESI = 00404000
9      mov esi, wVal      ; ESI = 00404001
10     mov esi, dVal      ; ESI = 00404003
11     mov esi, dVal2     ; ESI = 00404007
```

Relacionando com C/C++

- O valor retornado por OFFSET é um ponteiro. Compare o seguinte código com ambos C++ e a linguagem assembly:

```
1  // Versao C++ :
2
3  char array[1000];
4  char * p = array;
```

```
1  ; Linguagem Assembly:
2
3  section .bss
4  array RESB 1000
5
6  section .text
7  mov esi, array
```

PTR (MASM) Operator

- Sobrescreve o valor padrão de uma etiqueta (variável). Provê a flexibilidade de acessar parte de uma variável.

```
1   section .data
2   myDouble DD 12345678h
3
4   section .text
5   mov ax,WORD [myDouble]          ; loads 5678h
6   mov WORD [myDouble],4321h       ; saves 4321h
```

- A ordem Little Endian é usada quando armazenamos dados em memória.

Little Endian Order

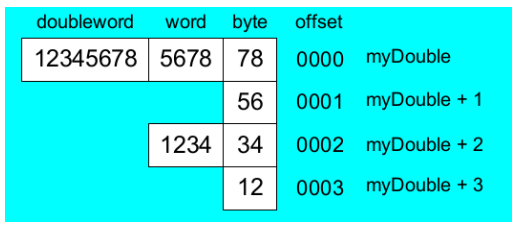
- A Intel utiliza a ordem **Little endian** para armazenar inteiros na memória.
- Inteiros que utilizam múltiplos bytes são armazenados na ordem inversa, com o byte menos significativo armazenado no menor endereço.
- Por exemplo, a **doubleword** 12345678h pode ser armazenada como:

byte	offset
78	0000
56	0001
34	0002
12	0003

Quando inteiros são carregados da memória em registradores, os bytes são automaticamente desinvertidos nas posições corretas.

Exemplo do Operador PTR (MASM) no NASM - 1/2

```
1 .data
2 myDouble DWORD 12345678h
```




```
1     mov al, BYTE [myDouble]       ;AL = 78h
2     mov al, BYTE [myDouble+1]     ;AL = 56h
3     mov al, BYTE [myDouble+2]     ;AL = 34h
4     mov ax, WORD [myDouble]       ;AX = 5678h
5     mov ax, WORD [myDouble+2]     ;AX = 1234h
```

Exemplo do Operador PTR (MASM) no NASM - 2/2

- PTR pode ser usado para combinar elementos de dados de tipos menores e movê-los em operanos maiores. O CPU irá automaticamente inverter os Bytes.

```
1  section .data
2  myBytes DB 12h,34h,56h,78h
3
4  section .text
5  mov ax,WORD [myBytes]      ; AX = 3412h
6  mov ax,WORD [myBytes+2]    ; AX = 7856h
7  mov eax,DWORD [myBytes]      ; EAX = 78563412h
```

Sua Vez . . .

- Escreva o valor de cada operando destino:

```
1   section .data
2       varB DB 65h,31h,02h,05h
3       varW DW 6543h,1202h
4       varD DD 12345678h
5
6   section .text
7       mov ax,WORD [varB+2]     ; a. 0502h
8       mov bl,BYTE [varD]         ; b. 78h
9       mov bl,BYTE [varW+2]     ; c. 02h
10      mov ax,WORD [varD+2]     ; d. 1234h
11      mov eax,DWORD [varW]       ; e. 12026543h
```

Tamanho de uma Variável de Memória

- No NASM não há uma diretiva ou operação para calcular o tamanho de um array ou de um dado, logo precisamos calcular manualmente.
- Símbolos $ e $$
  - $ - Posição corrente
  - $$ - Início do segmento

```
1         section .data
2             array DW 10,20,
3             DW 30,40,
4             DW 50,60
5             sizeOfarray DW ($-array) / 2
6
7         section .text
8             PRINT_UDEC 2, sizeOfarray
```

Dividindo em Múltiplas Linhas(1 of 2)

- Uma declaração de dados em múltiplas linhas se cada linha terminar em vírgula, exceto a última.

```
1    section .data
2    array dw 10,20,
3          dw 30,40,
4          dw 50,60
```

Endereçamento Indireto

- Operandos Indiretos
- Exemplo da Soma de um Vetor
- Operandos indexados
- Ponteiros

Operandos Indiretos

- Um operador indireto armazena o endereço de uma variável, usualmente um array ou string. Ele pode ser dsreferenciado (mesmo modo de um ponteiro).

```
1   section .data
2       val1 DB 10h,20h,30h
3
4   section .text
5       mov esi, val1
6       mov al,[esi] ; dereference ESI (AL = 10h)
7       inc esi
8       mov al,[esi] ; AL = 20h
9       inc esi
10      mov al,[esi] ; AL = 30h
```

Exemplo: Soma de Array

- Operandos indiretos são ideais para percorrer arrays. Note que o registrador entre colchetes deve ser incrementado por valor de acordo com o tipo de dado do array.

```
1    section .data
2    arrayW DW 1000h,2000h,3000h
3     mov esi, arrayW
4     mov ax,[esi]
5     add esi,2
6     add ax,[esi]
7     add esi,2
8     add ax,[esi]                    ;AX = soma do array
```

Tarefa

Modifique esse exemplo para um array de doublewords.

Operandos Indexados

- Um operando indexado soma uma constante a um registro para gerar o endereço efetivo. Existem duas formas de notação:
  - *[label + reg]*

```
1   section .data
2       arrayW DW 1000h,2000h,3000h
3
4   section .text
5       mov esi,0
6       mov ax,[arrayW + esi]              ; AX = 1000h
7       add esi,2
8       add ax,[arrayW + esi]
```

Tarefa

Mosifique esse exemplo para **doublewords**.

Dimensionando a Indexação

- Você pode dimensionar um operando indireto ou indexado para o offset de um elemento de um array. Isso é feito multiplicando o índice pelo tipo do elemento salvo no array:

```
1   section .data
2       arrayB DB 0,1,2,3,4,5
3       arrayW DW 0,1,2,3,4,5
4       arrayD DD 0,1,2,3,4,5
5
6   section .text
7       mov esi,4
8       mov al,[arrayB+(esi*1)]          ;04
9       mov bx,[arrayW+(esi*2)]          ;0004
10      mov edx,[arrayD+(esi*4)]         ;00000004
```

Ponteiros

- Você pode declarar uma variável como ponteiro que contém o offset de outra variável.

```
1   section .data
2       arrayW DW 1000h,2000h,3000h
3       ptrW DD arrayW
4
5   section .text
6       mov esi, [ptrW]
7       mov eax,arrayW
8       mov cx,[esi]            ; AX = 1000h
```

Formato Alternativo

ptrW DWORD OFFSET arrayW

Instruções JMP e LOOP

- Instrução JMP
- Instrução LOOP
- Exemplo de LOOP
- Somando um Array de Inteiros
- Copiando uma String

InstruçÃO JMP

- JMP é um salt incondicional para uma etiqueta (label) que usualmente pertence ao mesmo procedimento (label interna).
- Syntax: JMP alvo
- Lógica: EIP ¡- target
- Exemplo:

```
1    top:
2        .
3        .
4        jmp top
```

Um salto para fora do procedimento atual deve ser de um tpo especial de label, chamado label global

Instrução LOOP

- A instrução LOOP cria um laço contado
- Sintaxe: LOOP alvo
- Logic:
  - $ECX \longleftarrow ECX - 1$
  - if ECX != 0, vá para alvo
- Implementação:
  - O assembler calcula a distância, em bytes, entre o offset da instrução seguinte e o offset do label alvo. Isso é chamado de offset relativo.
  - O offset relativo é somado ao EIP.

Exemplo de LOOP

- O seguinte loop calcula a soma dos inteiros 5 + 4 + 3 +2 + 1:

```
1   OFFSET      MACHINE CODE      SOURCE CODE
2   00000000    66 B8 0000        mov ax,0
3   00000004    B9 00000005       mov ecx,5
4
5   00000009    66 03 C1      L1: add ax,cx
6   0000000C    E2 FB             loop L1
7   0000000E
```

Quando o LOOP é montado, a localização corrente é 0000000E (offset da próxima instrução). –5 (FBh) é somado à posição corrente, causando um salto para a posição 00000009:
$00000009 \longleftarrow 0000000E + FB$

Sua Vez . . .

Se o offset relativo é codificado em um byte com sinal.

(a) Qual o maior salto para trás que pode ser feito?

(b) Qual o maior salto para frente que pode ser feito?

Respostas

(a) -128

(b) +127

Sua Vez . . .

Qual será o valor final de AX?

```
1       mov ax,6
2       mov ecx,4
3   L1:
4       inc ax
5       loop L1
```

Quantas vezes o loop irá executar?

```
1       mov ecx,0
2   X2:
3       inc ax
4       loop X2
```

Sua Vez . . . (Resposta)

Qual será o valor final de AX?
10

```
1     mov ax,6
2     mov ecx,4
3     L1:
4     inc ax
5     loop L1
```

Quantas vezes o loop irá executar?
4,294,967,296

```
1     mov ecx,0
2     X2:
3     inc ax
4     loop X2
```

Loop Aninhado

Se você necessita de um laço com outro laço aninhado, você deve salvar o valor do contador EAX do laço externo. No exemplo que segue, o laço externo executa 100 vezes e o laço interno 20 vezes.

```
1   section .bss
2   count RESD 1
3
4   section .text
5   mov ecx,3               ; define o contador do loop externo
6
7   L1:
8   mov [count],ecx         ; salva o contador do loop externo
9   mov ecx,5               ; define o contador do loop interno
10
11  L2:
12                          ; nao faz nada
13  loop L2                 ; repete o laco interno
14  mov ecx,[count]         ; restaura o contador do laco externo
15  loop L1                 ; repete o laco externo
```

Somando um Array de Inteiros

- O seguinte código calcula a soma de um array de inteiros de 16 bits.

```
1   section .data
2       intarray dw 100h,200h,300h,400h
3       sizeOfIntArray db ($-intarray)/2
4
5   section .text
6       mov edi, intarray              ; endereco do intarray
7       mov ecx, [sizeOfIntArray]      ; contador do laco
8       mov ax,0                       ; zera o acumulador
9   L1:
10      add ax,[edi]                   ; soma um inteiro
11      add edi, 2                     ; aponta para o proximo inteiro
12      loop L1
```

Sua Vez . . .

Atividade

Que mudanças devem ser feitas no programa do slide anterior se você estiver somando um array de **doubleword**?

Copying a String

- The following code copies a string from source to target:

```
1   section .data
2       source DB "This is the source string",0
3       sizeOfSource DB $-source
4
5   section .bss
6       target RESB sizeOfSource-source
7
8   section .text
9       mov esi,0                   ; registrador de indice
10      mov ecx, [sizeOfSource]     ; contador de laco
11  L1:
12      mov al,[source+esi]         ; pega o char na origem
13      mov [target+esi],al         ; armazena o char no destino
14      inc esi                     ; move para o proximo caractere
15      loop L1
```

Sua vez . . .

Tarefa

Reescreva o programa mostrado no slide anterior usando endereçamento indireto ao invés de endereçamento indexado.

Programação em 64-Bit

- Instrução **MOV** em modo 64-bit aceita operandos de 8, 16, 32, ou 64 bits
- Quando você move uma constante de 8, 16, or 32-bit para um registrador de 64-bit, os bits maiores são limpos (0)
- Quando você move um operando de memória em um registrador 64-bit o resultado varia:
  - movimentação 32-bit limpa os bits maiores no destino
  - movimentação 8-bit ou 16-bit não afeta os bits maiores no destino

Mais Programação 64-Bit

- MOVSXD extende o sinal de um valor 32-bit em valor de registrador de destino de 64-bit
- O offset gera endereços de 64-bit
- **LOOP** usa o registrador RCX de 64-bit como contador
- RSI e RDI são os registradores de índice mais comuns de 64-bit para acessar **arrays**
- ADD e SUB afetam as flags da mesma forma que no modo 32-bit
- Você pode dimensionar fatores com operandos indexados