Linguagens de Montagem

Instruções Básicas Aritméticas e Lógicas

Aula 05

Edmar André Bellorini

Introdução

- Aula01: Montar, linkar e executar

```
$: nasm -f elf64 nome.asm
$: ld nome.o -o nome.x
$: ./nome.x
```

- Aula02: Estrutura dos programas e dados inicializados

```
section .data
```

- Aula03: Registradores e instrução de movimentação

```
MOV reg64, r/m64
```

- Aula04: Dados não inicializados e estrutura de programas

```
section .bss
```

Instruções Aritméticas e Lógicas

- Aritméticas
  - ADD
  - SUB
  - MUL e IMUL
  - DIV e IDIV
- Lógicas
  - OR
  - AND
  - XOR
  - NOT
  - NEG
- Deslocamento
  - SHL e SHR
  - SAL e SAR
  - outros deslocamentos

Soma

- Instrução `ADD`
  - Acumula em Destino o valor de Fonte

    `ADD Destino, Fonte`

    $Destino = Destino + Fonte$
- Sintaxe

```
ADD reg, r/m ; 8, 16, 32 e 64 bits
ADD mem, reg ; 8, 16, 32 e 64 bits
ADD r/m, imm ; 8, 16, 32 bits
```

Subtração

- Instrução SUB
  - Reduz o Fonte do Destino

    `SUB Destino, Fonte`

    Destino = *Destino* − *Fonte*
- Sintaxe

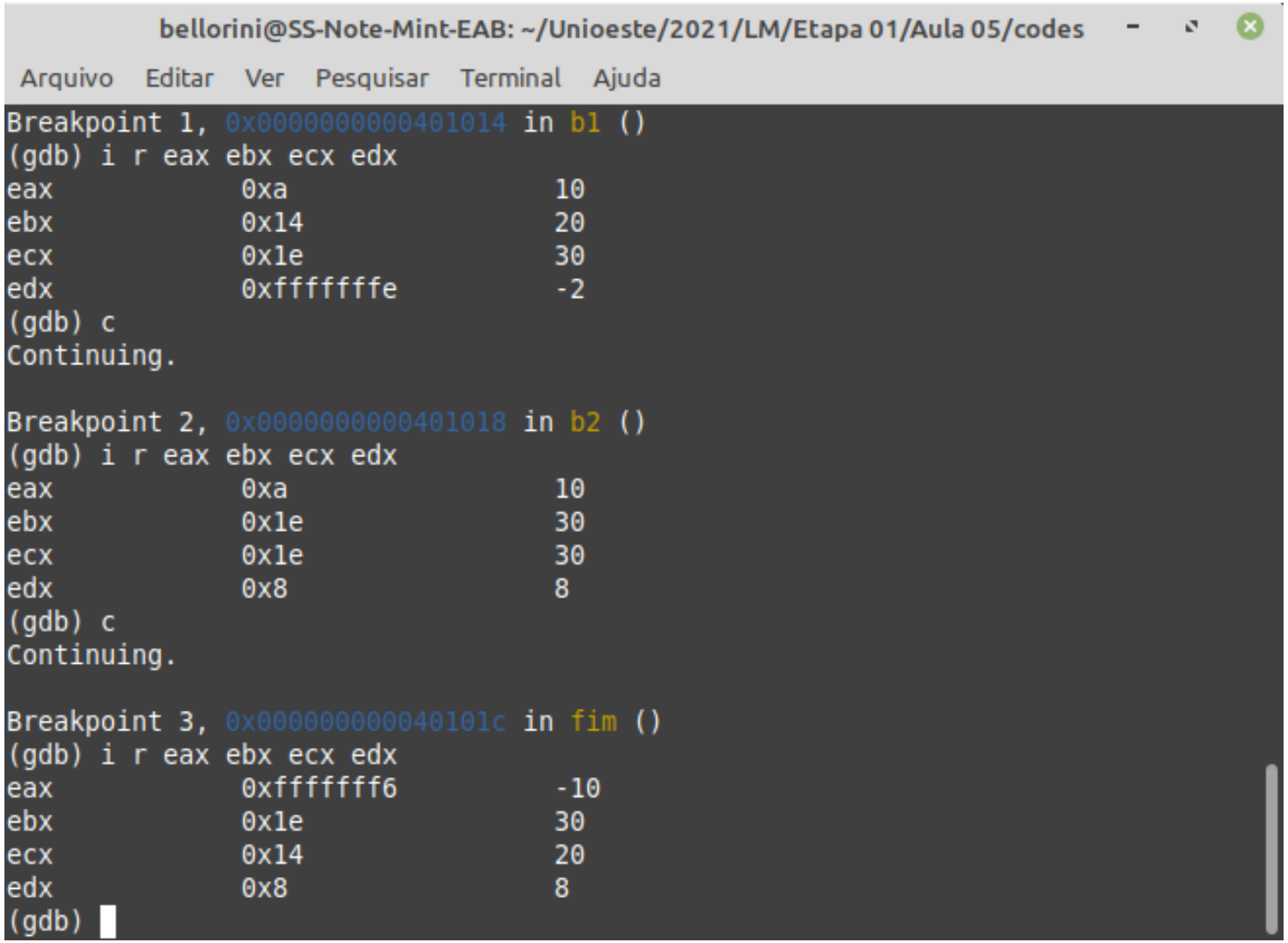
```
SUB reg, r/m ; 8, 16, 32 e 64 bits
SUB mem, reg ; 8, 16, 32 e 64 bits
SUB r/m, imm ; 8, 16, 32 bits
```

- Observação
  - Tanto ADD quanto SUB são operações sinalizadas
    Geram sinais de overflow (*flag* `OF`)
    Registrador `EFLAGS` ...

Exemplo a05e01.asm

```
1  section .text
2      global _start
3
4  _start:
5
6      mov eax, 10
7      mov ebx, 20
8      mov ecx, 30
9      mov edx, -2
10 b1:
11     add ebx, eax
12     add edx, eax
13 b2:
14     sub ecx, eax
15     sub eax, ecx
16
17 fim:
18     mov rax, 60
19     mov rdi, 0
20     syscall
```

Debugger


bellorini@SS-Note-Mint-EAB: ~/Unioeste/2021/LM/Etapa 01/Aula 05/codes
Arquivo Editar Ver Pesquisar Terminal Ajuda
Breakpoint 1, 0x0000000000401014 in b1 ()
(gdb) i r eax ebx ecx edx
eax 0xa 10
ebx 0x14 20
ecx 0x1e 30
edx 0xfffffffe -2
(gdb) c
Continuing.
Breakpoint 2, 0x0000000000401018 in b2 ()
(gdb) i r eax ebx ecx edx
eax 0xa 10
ebx 0x1e 30
ecx 0x1e 30
edx 0x8 8
(gdb) c
Continuing.
Breakpoint 3, 0x000000000040101c in fim ()
(gdb) i r eax ebx ecx edx
eax 0xfffffff6 -10
ebx 0x1e 30
ecx 0x14 20
edx 0x8 8
(gdb)

Registrador FLAGS

- É um registrador especial de estado do processador
- É alterado indiretamente por instruções aritméticas e lógicas
- FLAG
  - Registrador de estado de 16 bits
  - Somente para máquinas de 16 bits (antigas)
- EFLAG
  - Registrador de estado de 32 bits
  - Mantém retrocompatibilidade

Duas `Flags` de interesse

- Overflow
  - OF $= 1 \rightarrow$ operação aritmética gerou overflow
  - Importante para operações sinalizadas
- Carry
  - CF $= 1 \rightarrow$ operação aritmética gerou bit *carry*
  - Importante para operações não sinalizadas
- GDB

  `(gdb) info register eflags`

  ou

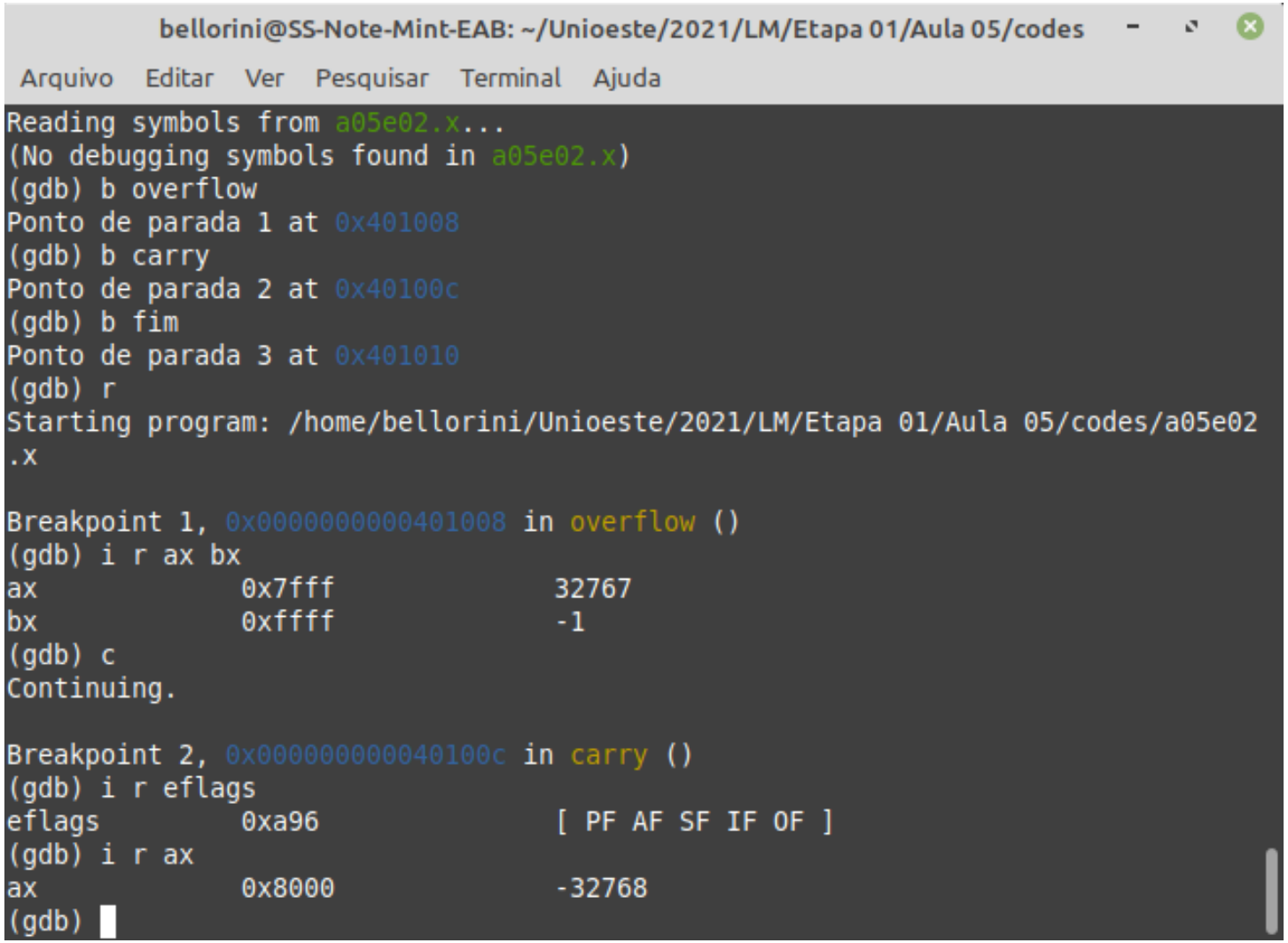
  `(gdb) i r eflags`

  - Será mostrado as flags ativas (valor = '1')

Exemplo a05e02.asm

```
1  section .text
2      global _start
3
4  _start:
5      mov ax, 0x7fff ; 32767
6      mov bx, 0xffff ; 65535
7
8  overflow:
9      add ax, 1 ; -32768
10 carry:
11     add bx, 1 ; 0
12
13 fim:
14     mov rax, 60
15     mov rdi, 0
16     syscall
```

Debugger


bellorini@SS-Note-Mint-EAB: ~/Unioeste/2021/LM/Etapa 01/Aula 05/codes
Arquivo Editar Ver Pesquisar Terminal Ajuda
Reading symbols from a05e02.x...
(No debugging symbols found in a05e02.x)
(gdb) b overflow
Ponto de parada 1 at 0x401008
(gdb) b carry
Ponto de parada 2 at 0x40100c
(gdb) b fim
Ponto de parada 3 at 0x401010
(gdb) r
Starting program: /home/bellorini/Unioeste/2021/LM/Etapa 01/Aula 05/codes/a05e02
.x
Breakpoint 1, 0x0000000000401008 in overflow ()
(gdb) i r ax bx
ax 0x7fff 32767
bx 0xffff -1
(gdb) c
Continuing.
Breakpoint 2, 0x000000000040100c in carry ()
(gdb) i r eflags
eflags 0xa96 [PF AF SF IF OF]
(gdb) i r ax
ax 0x8000 -32768
(gdb)

Instruções INC e DEC

- INC
  - Incrementa *Destino* em 1

    `INC Destino`

    $Destino = Destino + +$
- DEC
  - Decrementa *Destino* em 1

    `DEC Destino`

    $Destino = Destino - -$

AND, OR, e XOR

- Instruções
  - AND
    - Operação lógica AND bit-a-bit

      ```
      AND reg , r/m/i
      AND mem , r/i
      ```
  - OR
    - Operação lógica OR bit-a-bit

      ```
      OR reg , r/m/i
      OR mem , r/i
      ```
  - XOR
    - Operação lógica XOR bit-a-bit

      ```
      XOR reg , r/m/i
      XOR mem , r/i
      ```

NOT vs NEG

- Instruções
  - NOT
    - Operação lógica NOT bit-a-bit (**não** carrega EFLAGS)

      `NOT r/m`
  - NEG
    - Operação lógica de Negação em Complemento de 2

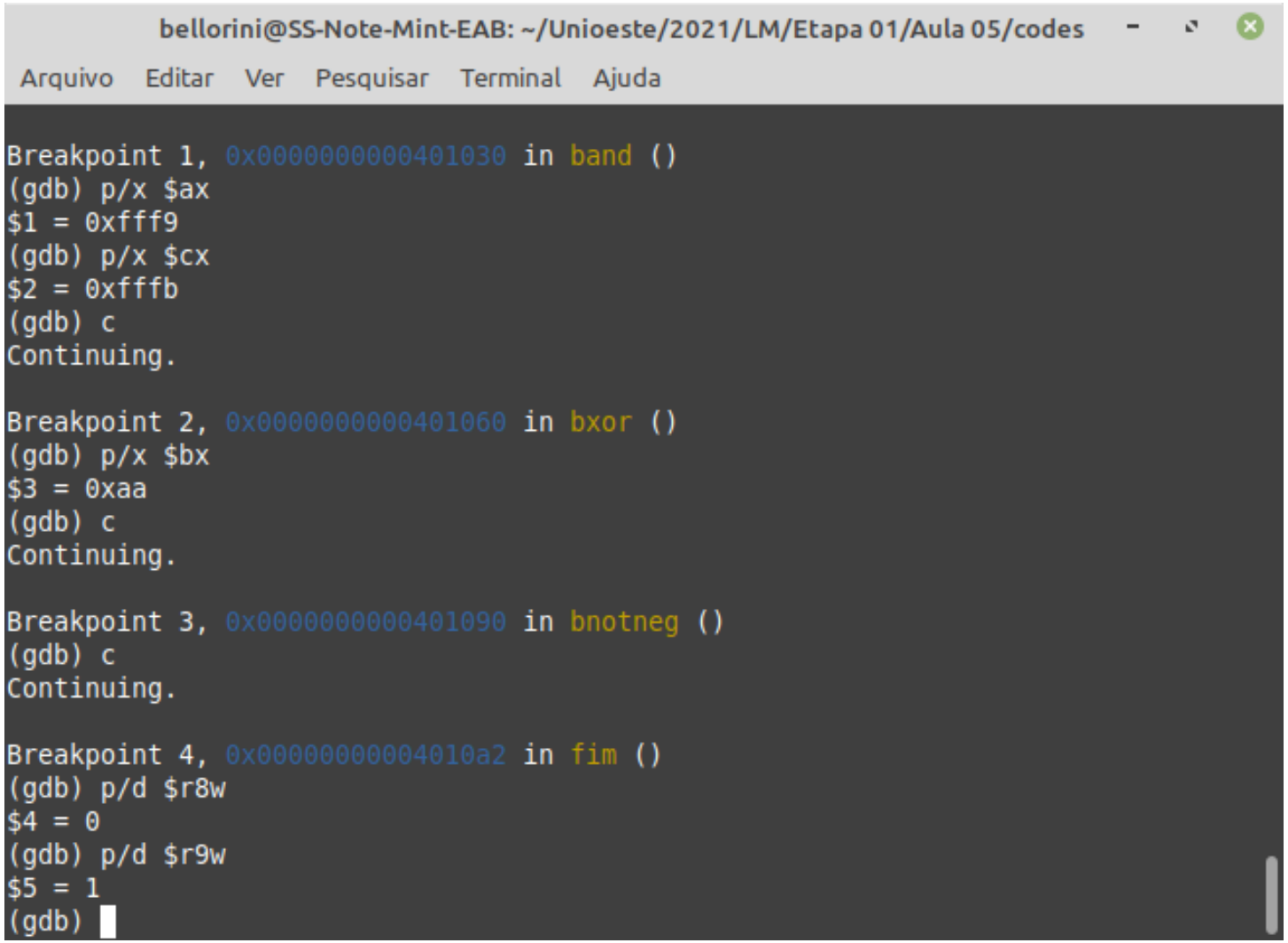
      `NEG r/m`

Exemplo - a05e03.asm

```
14    ...
15 section .text
16    global _start
17
18 _start:
19    mov ax, [v1]
20    mov bx, [v2]
21    mov cx, [v2]
22
23    ; or reg x reg
24    or cx, ax
25    ...
```

```
49    ...
50 bxor:
51    ; xor
52    mov ax, [v1]
53    mov bx, [v2]
54    mov cx, [v2]
55
56    ; or reg x reg
57    xor cx, ax
58
59    ; or mem x reg
60    mov [v1xorv2], ax
61    xor [v1xorv2], bx
62    ...
```

Debugger


bellorini@SS-Note-Mint-EAB: ~/Unioeste/2021/LM/Etapa 01/Aula 05/codes
Arquivo Editar Ver Pesquisar Terminal Ajuda
Breakpoint 1, 0x0000000000401030 in band ()
(gdb) p/x $ax
$1 = 0xfff9
(gdb) p/x $cx
$2 = 0xfffb
(gdb) c
Continuing.
Breakpoint 2, 0x0000000000401060 in bxor ()
(gdb) p/x $bx
$3 = 0xaa
(gdb) c
Continuing.
Breakpoint 3, 0x0000000000401090 in bnotneg ()
(gdb) c
Continuing.
Breakpoint 4, 0x00000000004010a2 in fim ()
(gdb) p/d $r8w
$4 = 0
(gdb) p/d $r9w
$5 = 1
(gdb)

Operações Lógicas de Deslocamento

- Deslocamento Lógico

```
SHL r/m, i
SHR r/m, i
```

- Deslocamento Aritmético

```
SAL r/m, i
SAR r/m, i
```

Deslocamento Lógico à Esquerda

- SHL → *Logical Left Shift*
  - Aplica deslocamento lógico em *dest*, *src* bits à esquerda

```
    _start:
    mov ax, 0xEE01      ;      1110 1110 0000 0001
desloc1:
    shl ax, 1 ; 0xDC02 ; 1 | 1101 1100 0000 0010
desloc2:
    shl ax, 4 ; 0xC020 ; 1 | 1100 0000 0010 0000
```

  - CF = bit expurgado
  - bit de entrada à direita é '0'
    usado em aritmética **não** sinalizada

Deslocamento Lógico à Direita

- SHR → *Logical Right Shift*
  - Aplica deslocamento lógico em *dest*, *src* bits à direita

```
   _start:
   mov bx, 0xEE01      ; 1110 1110 0000 0001
desloc1:
   shr bx, 1 ; 0x7700 ; 0111 0111 0000 0000 | 1
desloc2:
   shr bx, 4 ; 0x0770 ; 0000 0111 0111 0000 | 0
```

  - CF = bit expurgado
  - bit de entrada à esquerda é '0'
    usado em aritmética **não** sinalizada

Exemplo: a05e04.asm (Desl. Lógico)

```
    ...
section .text
    global _start

_start:
    mov al, 0x81 ; 1000 0001
toshl:
    shl al, 1
    ; CF = 1

toshr:
    ; al = 0000 0010
    shr al, 1
    ; al = 0000 0001
    ; CF = 0
    ...
```

Deslocamento Aritmético à Esquerda

- SAL → *Arithmetic Left Shift*
  - Aplica deslocamento aritmético em *dest*, *src* bits à esquerda

```
    _start:
    mov bx, 0xEE01      ;      1110 1110 0000 0001
desloc1:
    sal bx, 1 ; 0xDC02 ; 1 | 1101 1100 0000 0010
desloc2:
    sal bx, 4 ; 0xC020 ; 1 | 1100 0000 0010 0000
```

  - CF = bit expurgado
  - bit de entrada à direita é '0'
    bit menos significativo não altera sinal do número
    SAL é sinônimo de SHL
    usado em aritmética **sinalizada**

Deslocamento Aritmético à Direita

- SAR → *Arithmetic Right Shift*
  - Aplica deslocamento aritmético em *dest*, *src* bits à direita

```
    _start:
    mov bx, 0xEE01      ; 1110 1110 0000 0001
desloc1:
    sar bx, 1 ; 0xF700 ; 1111 0111 0000 0000 | 1
desloc2:
    sar bx, 4 ; 0xFF70 ; 1111 1111 0111 0000 | 0
```

  - CF = bit expurgado
  - bit de entrada à esquerda é mantido
    bit mais significativo **altera** sinal do número
    usado em aritmética **sinalizada**

Exemplo: a05e05.asm (Desl. Aritmético)

```
    ...
section .text
    global _start

_start:
    mov al, 0x81 ; 1000 0001
toshl:
    sal al, 1
    ; CF = 1

toshr:
    ; al = 0000 0010
    sar al, 1
    ; al = 0000 0001
    ; CF = 0
    ...
```

Outros Deslocamentos - ROL|ROR, RCL|RCR, variantes X

- ROL|ROR $\rightarrow$ *Rotate Left |Right*
  - Aplica deslocamento rotacional em *dest*, *src* bits na direção
  - CF = bit expurgado
  - bit de entrada à direita é bit expurgado
- RCL|RCR $\rightarrow$ *Rotate Through Carry Left |Right*
  - Aplica deslocamento rotacional em *dest*, *src* bits na direção
  - CF = bit expurgado
  - bit de entrada à direita é CF anterior
- SARX |SHLX |SHRX $\rightarrow$ *Deslocamento*
  - Aplica deslocamento em *dest*, *src* bits e não afeta FLAGS

Operações Aritméticas de Inteiros

- Multiplicação

```
MUL  r/m
IMUL r/m
IMUL reg, r/m     ; 16, 32 ou 64 apenas
```

- Divisão

```
DIV  r/m
IDIV r/m
```

Multiplicação de Inteiros

- MUL → Multiplicação de inteiros **não** sinalizados
- IMUL → Multiplicação de inteiros **sinalizados**

```
 MUL r/m
IMUL r/m
```

- Operandos
  - Implícito: RAX (EAX |AX)
  - Explícito: r/m (64, 32 ou 16 bits)
  - Resultado: RDX:RAX (EDX:EAX |DX:AX)
- Operação:
  RDX:RAX = RAX * $r64/m64$ (para 64 bits)

Multiplicação de Inteiros - Slide 2

- IMUL → Multiplicação de inteiros **sinalizados**

```
IMUL r, r/m
```

- Operandos
  - Explícito: r
  - Explícito: r/m (64, 32 ou 16 bits)
  - Resultado: r
- Operação:
  $r = r^*\ r/m$

Divisão de Inteiros

- DIV → Divisão de inteiros **não** sinalizados
- IDIV → Divisão de inteiros **sinalizados**

```
 DIV r/m
IDIV r/m
```

- Operandos
  - Implícito: RDX:RAX (EDX:EAX |DX:AX)
  - Explícito: r/m (64, 32 ou 16 bits)
  - Resultado: RAX (Quociente) e RDX (Resto)
- Operação:
  RAX = RDX:RAX $\div$ $r64/m64$
  - com Resto armazenado em RDX

Divisão de Inteiros

- Como cuidar de #DX?
- Instruções especiais

```
CWD - Convert Word to Doubleword
DX:AX = sign-extend AX
```

```
CDQ - Convert Doubleword to Quadword
EDX:EAX = sign-extend EAX
```

```
CQO - Convert Quadword to Octaword
RDX:RAX = sign-extend RAX
```

Exemplo: a05e06.asm (Mul e Div de Inteiros un/signed)

```
    ...
multiplicacao0:
    ; EAX <- multiplicando1 = 50
    ; EDX:EAX <- EAX * multiplicador
    mov eax,  [multiplicando1]
    mul dword [multiplicador1] ; pode ser memoria
    ...
```

```
    ...
divisao0:
    ; EAX <- dividendo1 = 100
    ; EDX:EAX <- EDX:EAX / divisor
    mov eax,  [dividendo1] ; bytes baixos
    cdq
    div dword [divisor1]  ; pode ser memoria
    ...
```

Atividade a05at01

- a05at01 - Escreva um código funcional que:
  - Contenha 2 variáveis inicializadas:

```
maiuscula: db 'A'
minuscula: db 'b'
```

  - Contenha 2 variáveis não inicializadas:

```
lowercase: resb 1
uppercase: resb 1
```

  - Usando apenas as instruções `ADD` e/ou `SUB` converta:
    - *maiuscula* para minuscula e grave em *lowercase*
      $lowercase = tolowercase(maiuscula)$
    - *minuscula* para maiúscula e grave em *uppercase*
      $uppsercase = touppercase(minuscula)$
  - A tabela `ASCII` é sua amiga e extremamente necessária para completude deste exercício

Atividade a05at02

- a05at02 - A atividade a05at01 contém um *bug*
  - O que acontece se os caracteres inicializados forem colocados de maneira incorreta? Por exemplo:

```
maiuscula: db 'a'
minuscula: db 'B'
```

  - Usando somente as operações lógicas `OR`, `XOR` e/ou `AND`, reescreva o código a05at01 de modo a eliminar o *bug*

Atividade a05at03

- a05at03 - Somatório de um vetor de Inteiros.
  - Considere as seguintes seções:

```
section .data
    triangularNum : dd 0, 1, 3, 6, 10, 15, 21, 28

section .bss
    somatorio : resd 1
```

  - Elabore um programa em assembly que armazene em *somatorio* o somatório de todos os elementos de *triangularNum*.
    - Observações:
      Ainda não temos laços de repetição, então, deverá ser somado um-a-um
      Para calcular cada endereço, use LEA + indexador * tamanhoDoInteiro
      O indexador pode ser incrementado a cada operação de calculo de endereço

Atividade a05at04

- a05at04 - Calculo do n-éssimo número triangular
  - Considere as seguintes seções:

```
section .data
    n : dd 40

section .bss
    enessimotrinumero : resd 1
```

  - Elabore um programa em assembly que calcule e armazene em *enessimotrinumero* o número tringular na posição *n*.
    - Para calcular o n-essimo número use: $T_n = \frac{n*(n+1)}{2}$

Fim do Documento

Dúvidas?

Próxima aula:

- Aula 06: Chamadas de Sistemas
  - Escrever no terminal
  - Ler do teclado
  - Encerrar o programa
  - Arquivos