Introdução à Software Básico: Introdução a Assembly IA-32

Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade de Brasília

Sumário

Dispositivos de Hardware

- Vídeo: somente saída (saída padrão)
- Impressora: somente saída
- Teclado: somente entrada (entrada padrão)
- Mouse: somente entrada
- Gravadora CD/DVD: entrada e saída
- Dispositivo de rede: entrada e saída

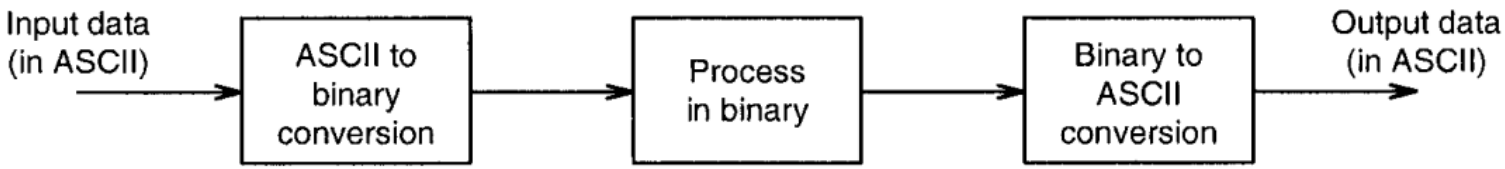
Entrada e Saída de Dados

- E/S é sempre Controlada pelo Sistema Operacional ou pela BIOS
- Em C, cada dispositivo de entrada e saída possui um nome, como um arquivo
  - E/S realizada usando funções disponibilizadas em bibliotecas da linguagem
- Em Assembly, cada dispositivo possui um número identificador.
  - E/S via procedimentos do SO ou da BIOS
  - io.o: biblioteca de macros, facilita E/S

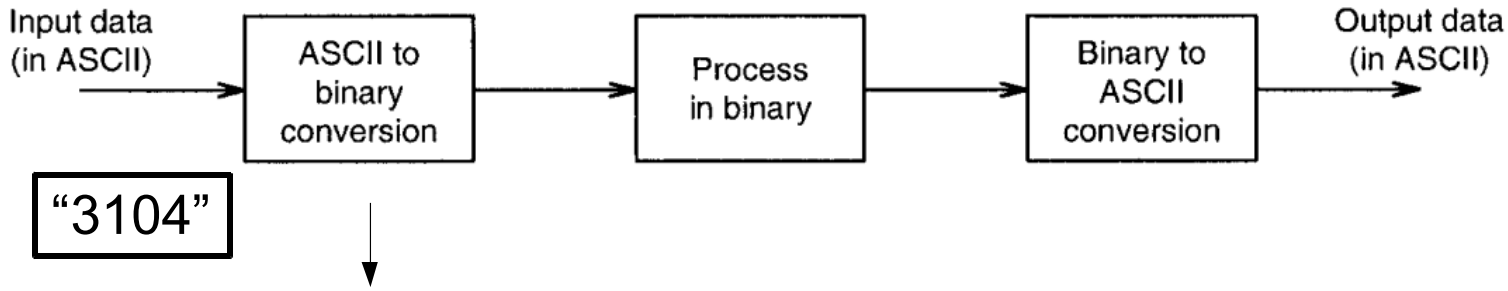
E/S em Assembly

- Leitura e escrita sempre em ASCII
- Chamada a procedimentos do SO (INT 80H) ou da BIOS (INT 14H)
- Biblioteca auxiliar io.o:
  - Macros para entrada e saída para string, caracter, valores de 16 bits com sinal e valores de 32 bits com sinal
  - Conversão para tipos específicos deve ser feito dentro do programa em Assembly

Conversão de tipos: Binário $\longleftrightarrow$ ASCII

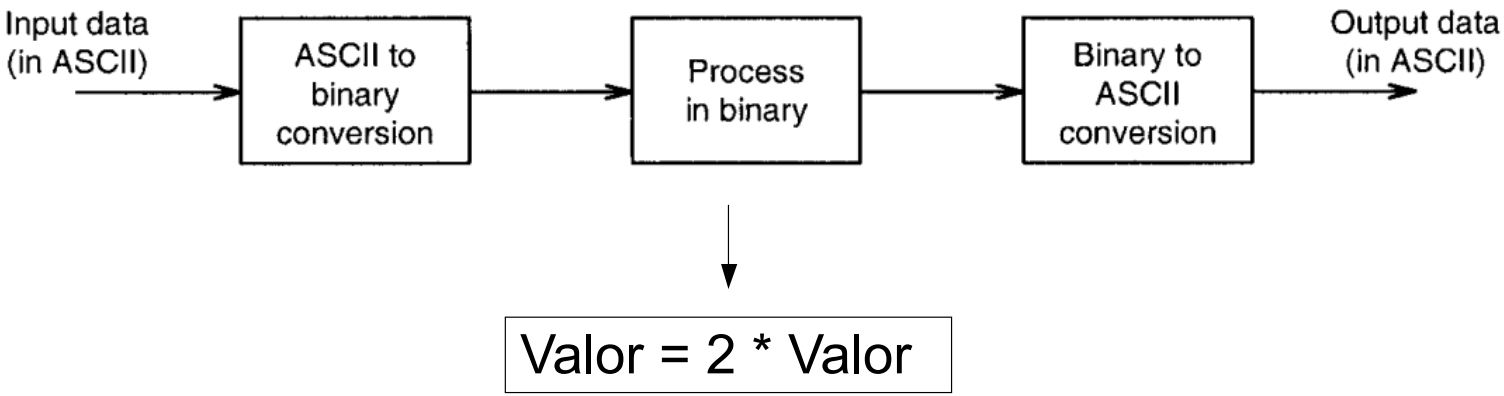


Conversão de tipos: Binário $\longleftrightarrow$ ASCII

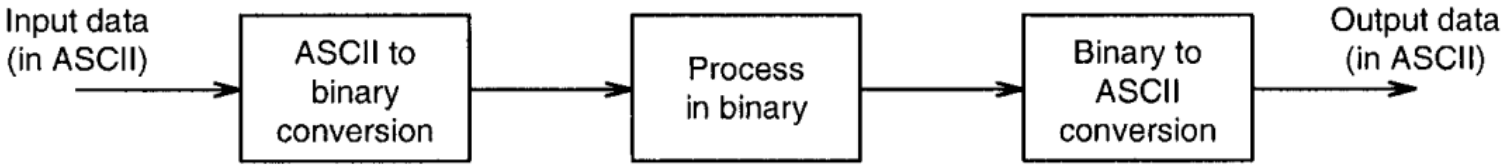



```
Valor = '3' – 0x30
Valor = Valor*10 + '1' – 0x30
Valor = Valor*10 + '0' – 0x30
Valor = Valor*10 + '4' – 0x30
```

Conversão de tipos: Binário $\longleftrightarrow$ ASCII

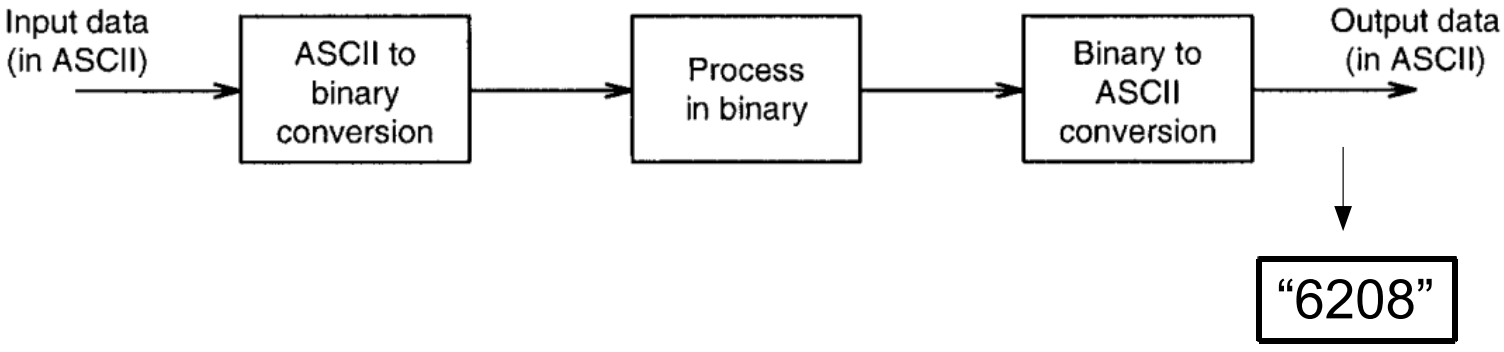


Conversão de tipos: Binário $\longleftrightarrow$ ASCII




```
i=0
do{
  str[i] = (char)((Valor % 10) + 0x30);
  Valor = (int) (Valor / 10);
  If Valor != 0  str[i+1]= str[i]
  i = i+1
} while (Valor != 0)
str[i] = '/0'
```

Conversão de tipos: Binário $\longleftrightarrow$ ASCII



Macros definidas em io.mac

Table 7.1 Summary of I/O routines defined in the `io.mac` file

name	operand(s)	operand location	size	what it does
`PutCh`	`source`	value register memory	8 bits	Displays the character located at `source`
`GetCh`	`dest`	register memory	8 bits	Reads a character into `dest`
`nwln`	none	—	—	Displays a newline
`PutStr`	`source`	memory	variable	Displays the NULL-terminated string at `source`
`GetStr`	`dest[,buf_size]`	memory	variable	Reads a carriage-return-terminated string into `dest` and stores it as a NULL-terminated string. Maximum string length is `buf_size-1`.
`PutInt`	`source`	register memory	16 bits	Displays the signed 16-bit number located at `source`
`GetInt`	`dest`	register memory	16 bits	Reads a signed 16-bit number into `dest`
`PutLInt`	`source`	register memory	32 bits	Displays the signed 32-bit number located at `source`
`GetLInt`	`dest`	register memory	32 bits	Reads a signed 32-bit number into `dest`

Exemplo

```
%include "io.mac"
.DATA
char_prompt    db  "Please input a character: ",0
out_msg1       db  "The ASCII code of '",0
out_msg2       db  "' in hex is ",0
query_msg      db  "Do you want to quit (Y/N): ",0
hex_table      db  "0123456789ABCDEF"    ; translation table: 4-bit binary to hex
.CODE
      .STARTUP
read_char:       PutStr  char_prompt  ; request a char. input
                 GetCh   AL           ; read input character
                 PutStr  out_msg1
                 PutCh   AL
                 PutStr  out_msg2
                 mov     AH,AL        ; save input character in AH
                 mov     EBX,hex_table ; BX := translation table
                 shr     AL,4         ; move upper 4 bits to lower half
                 xlatb                ; replace AL with hex digit
                 PutCh   AL           ; write the first hex digit
                 mov     AL,AH        ; restore input character to AL
                 and     AL,0FH       ; mask off upper 4 bits
                 xlatb
                 PutCh   AL           ; write the second hex digit
                 nwln
                 PutStr  query_msg    ; query user whether to terminate
                 GetCh   AL           ; read response
                 cmp     AL,'Y'       ; if response is not 'Y'
                 jne     read_char    ; read another character
done:     .EXIT
```

Exemplo

```
%include "io.mac"
.DATA
char_prompt    db  "Please input a character: ",0
out_msg1       db  "The ASCII code of '",0
out_msg2       db  "' in hex is ",0
query_msg      db  "Do you want to quit (Y/N): ",0
hex_table      db  "0123456789ABCDEF"    ; translation table: 4-bit binary to hex
.CODE
      .STARTUP
read_char:      PutStr  char_prompt  ; request a char. input
                GetCh   AL           ; read input character
                PutStr  out_msg1
                PutCh   AL
                PutStr  out_msg2
                mov     AH,AL        ; save input character in AH
                mov     EBX,hex_table ; BX := translation table
                shr     AL,4         ; move upper 4 bits to lower half
                xlatb                ; replace AL with hex digit
                PutCh   AL           ; write the first hex digit
                mov     AL,AH        ; restore input character to AL
                and     AL,0FH       ; mask off upper 4 bits
                xlatb
                PutCh   AL           ; write the second hex digit
                nwln
                PutStr  query_msg    ; query user whether to terminate
                GetCh   AL           ; read response
                cmp     AL,'Y'       ; if response is not 'Y'
                jne     read_char    ; read another character
done:     .EXIT
```

Exemplo

XLAT/XLATB - Translate

```
Usage:  XLAT    translation-table
        XLATB   (masm 5.x)
Modifies flags: None

Replaces the byte in AL with byte from a user table addressed by
BX.  The original value of AL is the index into the translate table.
The best way to describe this is MOV AL,[BX+AL]

                          Clocks                  Size
Operands          808x  286   386   486           Bytes

table offset       11    5     5     4             1
```

```
%include "io.mac"
.DATA
                mov     AH,AL       ; save input character in AH
                mov     EBX,hex_table ; BX := translation table
                shr     AL,4        ; move upper 4 bits to lower half
                xlatb               ; replace AL with hex digit
                PutCh   AL          ; write the first hex digit
                mov     AL,AH       ; restore input character to AL
                and     AL,0FH      ; mask off upper 4 bits
                xlatb
                PutCh   AL          ; write the second hex digit
                nwln
                PutStr  query_msg   ; query user whether to terminate
                GetCh   AL          ; read response
                cmp     AL,'Y'      ; if response is not 'Y'
                jne     read_char   ; read another character
done:   .EXIT
```

E/S em Assembly

Sem utilizar io.o

- Em Linux, o acesso a E/S é feito via interrupção de software:
  - Int 0x80
- A interrupção de Software Int 0x80 pode executar várias tarefas de interface com o SO.

int 0x80

Chamando a int 0x80:

- Coloque o valor da system call em EAX;
- Coloque os argumentos da system call em EBX, ECX e assim por diante - caso todos os regs estejam ocupados, EBX guarda o local na memória onde está a lista de argumentos;
- Chame a interrupção int 80h;
- O resultado da chamada geralmente é retornado em EAX.

Abrindo Arquivos

- System call 5 — Open a file
- Entradas:
  - EAX = 5
  - EBX = filename
  - ECX = file access mode
  - EDX = file permissions
- Saídas:
  - EAX = file descriptor
  - Error: EAX = error code

Abrindo Arquivos

- System call 8 — Create and open a file
- Entradas:
  - EAX = 8
  - EBX = filename
  - ECX = file permissions
- Saídas:
  - EAX = file descriptor
  - Error: EAX = error code

Abrindo arquivo para leitura: Exemplo

- mov EAX,5
- mov EBX,in_file_name
- mov ECX,0
- mov EDX,0700
- int 0x80

Criando/Abrindo arquivo para escrita: Exemplo

- mov EAX,8
- mov EBX,out_file_name
- mov ECX,0700
- int 0x80

Permissões de acesso ao Arquivo

Como visto anteriormente, o modo passado durante a criação do arquivos estabelece que tipo de permissões cada usuário do sistema tem sobre o arquivo.

O modo é um inteiro composto por três grupos de três bits.

Olhando o inteiro da esquerda para à direita, temos:

- O primeiro grupo contém as informações relativas às permissões do criador do arquivo.
- O segundo grupo contém as informações relativas às permissões dos usuários do grupo que o usuário pertence.
- O terceiro grupo contém as informações relativas aos usuários que não pertencem ao grupo do criador.

Permissões de acesso ao Arquivo

O primeiro bit de cada um desses grupos, representa que é possível fazer a leitura do arquivo. O segundo bit representa que é possível fazer a escrita no arquivo. O terceiro bit representa que é possível executar o arquivo.

Então, por exemplo, se tomarmos o número octal $0644 = 110100100$ como o modo, temos que o criador do arquivo tem permissão de leitura e escrita no arquivo, os usuários pertencentes ao grupo do criador tem permissão de leitura apenas e os usuários que não pertencem ao grupo do criador tem permissão de leitura apenas.

Modos de Acesso

C Constant	Numeric Value	Description
O_RDONLY	00	Open the file for read-only access.
O_WRONLY	01	Open the file for write-only access.
O_RDWR	02	Open the file for both read and write access.
O_CREAT	0100	Create the file if it does not exist.
O_EXCL	0200	When used with O_CREAT, if the file exists, do not open it.
O_TRUNC	01000	If the file exists and is open in write mode, truncate it to a length of zero.
O_APPEND	02000	Append data to the end of the file.
O_NONBLOCK	04000	Open the file in nonblocking mode.
O_SYNC	010000	Open the file in synchronous mode (allow only one write at a time).
O_ASYNC	020000	Open the file in asynchronous mode (allow multiple writes at a time).

Leitura em Arquivo

- System call 3 — Read from a file
- Entradas:
  - EAX = 3
  - EBX = file descriptor
  - ECX = pointer to input buffer
  - EDX = buffer size (maximum number of bytes to read)
- Saídas:
  - EAX = number of bytes read
  - Error: EAX = error code

Escrita em Arquivo

- System call 4 — Write to a file
- Entradas:
  - EAX = 4
  - EBX = file descriptor
  - ECX = pointer to output buffer
  - EDX = buffer size (number bytes to write)
- Saídas:
  - EAX = number of bytes written
  - Error: EAX = error code

Leitra em Arquivo: Exemplo

- mov EAX,3
- mov EBX,[fd_in]
- mov ECX,in_buf
- mov EDX,BUF_SIZE
- int 0x80

Escrita em arquivo: Exemplo

- mov EDX,EAX
- mov EAX,4
- mov EBX,[fd_out]
- mov ECX,in_buf
- int 0x80

Fechando Arquivos

- System call 6 — Close a file
- Entradas:
  - EAX = 6
  - EBX = file descriptor
- Saídas:
  - EAX = —
  - Error: EAX = error code

Expressões e Operadores em Assembly

- Não é possível criar expressões complexas em uma única linha de código
  - Sequência de instruções forma uma expressão
- Os operadores constituem instruções aritméticas e lógicas própria do processador
  - Adição: add, adc, inc
  - Subtração: sub, sbb, dec, neg, cmp
  - Multiplicação: mul, imul
  - Divisão: div , idiv
  - auxiliares: cbw, cwd, cdq, cwde, movsx, movzx

Relembrado: Soma de Valores sem sinal

$$\begin{array}{r} 00001111B\ (0FH=\ 15D) \\ 11110001B\ (F1H\ =\ 241D) \\ \hline 1\ 00000000B\ (100H\ =\ 256D) \end{array}$$

Relembrado: Soma de Valores sem sinal

```
  00001111B (0FH=  15D)
  11110001B (F1H =  241D)
-----------------------
1 00000000B (100H = 256D)
```

Vai um ou CARRY

Relembrado: Soma de Valores com sinal

Soma de Valores com sinal: -5 + 120

5D = 00000101B → -5 = 11111011+1 = 11111100B

120D = 01111000B

```
  11111100B  (FBH = 251D)
  01111000B  (78H = 120D)
------------------------
1 01110100B (174H = 372D)
```

Relembrado: Soma de Valores com sinal

```
  5D = 00000101B →  -5 = 11111011+1 =
11111100B
  120D = 01111000B

                    11111100B  (FBH = 251D)
                    01111000B  (78H = 120D)
                  ------------------------
                1 01110100B (174H = 372D)
```

Vai um ou CARRY

Relembrado: Soma de Valores com sinal

Soma de Valores com sinal: -5 + 120

5D = 00000101B → -5 = 11111010+1 = 11111011B

120D = 01111000B

Operandos de sinais diferentes, overflow é zero

```
11111011B   (FBH = -5D)
01111000B   (78H = 120D)
--------
01110011B   (73H = 115D)
```

Sinal ou Sem Sinal

- A operação de adição é sempre realizada pelo mesmo circuito lógico:
  - Valores com sinal negativo são representados em complemento 2
- O valor do flag carry é ajustado de acordo com o cálculo sem sinal
- O valor do flag overflow é ajustado de acordo com cálculo com sinal
- O programador deve usar instruções de acordo com o tipo de dado em uso

Incremento e Decremento

- Duas instruções: INC e DEC
  - inc destination
  - dec destination
- Operando pode ser registrador ou memória, 8, 16 ou 32 bits

Incremento e Decremento

Tamanho da Operação Indefinida

```
%include "io.mac"
.DATA
count   DW          0
value   DB          25
.CODE
.STARTUP
        inc         [count]
        dec         [value]
        mov         EBX,count
        inc         [EBX]
        mov         ESI,value
        dec         [ESI]
.EXIT
```

Incremento e Decremento

```
%include "io.mac"
.DATA
count   DW          0
value   DB          25
.CODE
.STARTUP
        inc         WORD  [count]
        dec         BYTE  [value]
        mov         EBX,count
        inc         WORD  [EBX]
        mov         ESI,value
        dec         BYTE  [ESI]
.EXIT
```

Adição

- Instrução ADD
  - add dest,source (dest = dest+source)

Table 9.1 Some examples of the `add` instruction

Instruction	Before add		After add
	source	destination	destination
`add AX,DX`	`DX = AB62H`	`AX = 1052H`	`AX = BBB4H`
`add BL,CH`	`BL = 76H`	`CH = 27H`	`BL = 9DH`
`add value,10H`	—	`value = F0H`	`value = 00H`
`add DX,count`	`count = 3746H`	`DX = C8B9H`	`DX = FFFFH`

Subtração

- Instrução SUB
  - add dest,source (dest = dest-source)

Subtração

- Instrução SUB
  - add dest,source (dest = dest+(-source))
  - O valor negativo é representado em complemento de 2

Table 9.1 Some examples of the `add` instruction

Instruction	Before add		After add
	source	destination	destination
`add AX,DX`	`DX = AB62H`	`AX = 1052H`	`AX = BBB4H`
`add BL,CH`	`BL = 76H`	`CH = 27H`	`BL = 9DH`
`add value,10H`	—	`value = F0H`	`value = 00H`
`add DX,count`	`count = 3746H`	`DX = C8B9H`	`DX = FFFFH`

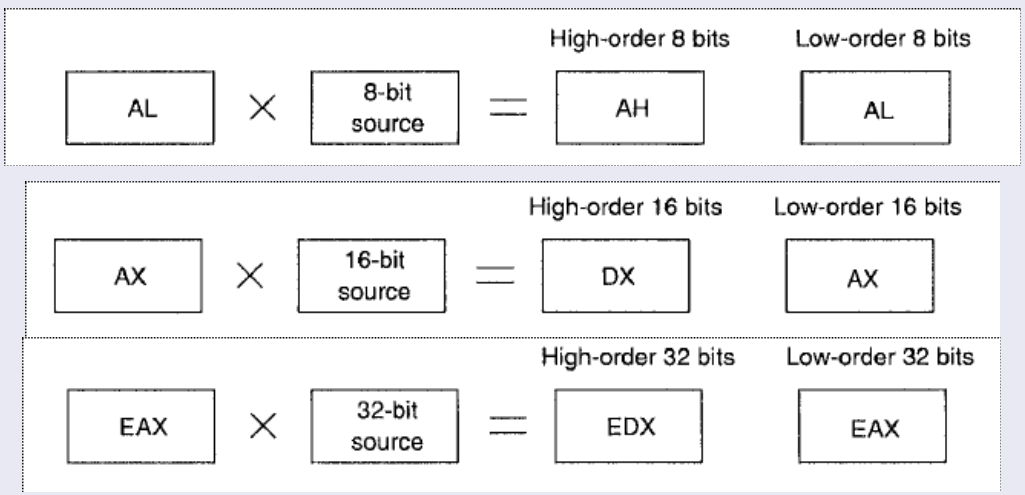
Adição de 64 bits

- Instruções somente aceitam operandos de no máximo 32 bits
- Para somar 64 bits, temos que usar pares de registradores e o carry flag na soma

```
%include "io.mac"
.DATA
Op1     DQ 371026A812579AE7H
Op2     DQ 489BA321FE604213H
.UDATA
Result  RESQ 1
.CODE
.STARTUP
mov  DWORD EBX,[Op1]
mov  DWORD EAX,[Op1+4]
mov  DWORD EDX,[Op2]
mov  DWORD ECX,[Op2+4]
add  EBX,EDX
adc  EAX,ECX
mov  DWORD [Result], EBX
mov  DWORD [Result+4],EAX
.EXIT
```

Multiplicação

- Diferentes instruções para números com e sem sinal
  - mul source
  - Multiplica source pelo conteúdo do acumulador
  - Conteúdo dos Flags indicam se os registradores de parte alta foram utilizados



Multiplicação

Unsigned Multiply	MUL	r/m8	Multiplies unsigned AL by r/m8. Stores result in AX.	AX ← AL * SRC; IF AH=0 THEN EFLAGS.OF, .CF ← 00B; ELSE EFLAGS.OF, .CF ← 11B; *// EFLAGS. .ZF, .AF, .PF., SF are undefined*
		r/m16	Multiplies unsigned AX by r/m16. Stores result in DX:AX.	DX:AX ← AX * SRC; IF DX=0 THEN EFLAGS.OF, .CF ← 00B; ELSE EFLAGS.OF, .CF ← 11B; *// EFLAGS. .ZF, .AF, .PF., SF are undefined*
		r/m32	Multiplies unsigned EAX by r/m32. Stores result in EDX:EAX.	EDX:EAX ← EAX * SRC; IF EDX=0 THEN EFLAGS.OF, .CF ← 00B; ELSE EFLAGS.OF, .CF ← 11B; *// EFLAGS. .ZF, .AF, .PF., SF are undefined*

Multiplicação com sinal (imul)

- imul source
  - Multiplica source pelo conteúdo do acumulador
  - Mesmo formato de mul, mas o significado dos flags overflow e carry são diferentes
  - Flags resetados, parte alta é extensão do sinal

```
mov DL,OFFH ; DL = -1
mov AL,42H ; AL = 66
imul DL
```

1111111110111110 (-66)

```
mov DL,OFFH ; DL = -1
mov AL,0BEH ; AL = -66
imul DL
```

0000000001000010 (+66)

  - Flags setados, resultado maior que operandos (como em mul)

```
mov DL,25 ; DL = 25
mov AL,0F6H ; AL = -10
imul DL
```

→ 1111111100000110 (-250)

Multiplicação

Signed Multiply	IMUL	r/m8	Multiplies signed AL by r/m8. Stores result in AX.	AX ← AL * SRC; IF AX=AL THEN EFLAGS.CF, .OF ← 00B; ELSE EFLAGS.CF, .OF ← 11B; *// EFLAGS. .ZF, .AF, .PF., SF are undefined*
		r/m16	Multiplies signed AX by r/m16. Stores result in DX:AX.	DX:AX ← AX * SRC; IF DX:AX=SignExtend(AX) THEN EFLAGS.CF, .OF ← 00B; ELSE EFLAGS.CF, .OF ← 11B; *// EFLAGS. .ZF, .AF, .PF., SF are undefined*
		r/m32	Multiplies signed EAX by r/m32. Stores result in EDX:EAX.	EDX:EAX ← EDX * SRC; IF EDX:EAX=EAX THEN EFLAGS.CF, .OF ← 00B; ELSE EFLAGS.CF, .OF ← 11B; *// EFLAGS. .ZF, .AF, .PF., SF are undefined*
		r16, r/m16	Multiplies signed word register by r/m16 word.	TMP ← DST * SRC; *// TMP is double DST size* DST ← DST * SRC; IF TMP=DST THEN EFLAGS.CF, .OF ← 00B; ELSE EFLAGS.CF, .OF ← 11B; *// EFLAGS. .ZF, .AF, .PF., SF are undefined*
		r32, r/m32	Multiplies signed dword register by r/m32 dword.	
		r16, imm8	Multiplies signed word register by sign-extend imm8 value.	
		r32, imm8	Multiplies signed dword register by sign-extend imm8 value.	
		r16, imm16	Multiplies signed word register by sign-extend imm8 value.	
		r32, imm32	Multiplies signed dword register by sign-extend imm8 value.	
		r16, r/m16, imm8	Multiplies signed r/m16 word by sign-extend imm8 value. Stores result in word register.	TMP ← SRC1 * SRC2; *// TMP is double SRC1 size* DST ← SRC1 * SRC2 IF TMP=DST THEN EFLAGS.CF, .OF ← 00B; ELSE EFLAGS.CF, .OF ← 11B; *// EFLAGS. .ZF, .AF, .PF., SF are undefined*
		r32, r/m32, imm8	Multiplies signed r/m32 dword by sign-extend imm8 value. Stores result in dword register.	
		r16, r/m16, imm16	Multiplies signed r/m16 word by imm16 value. Stores result in word register.	
		r32, r/m32, imm32	Multiplies signed r/m32 dword by imm16 value. Stores result in dword register.	

Multiplicação

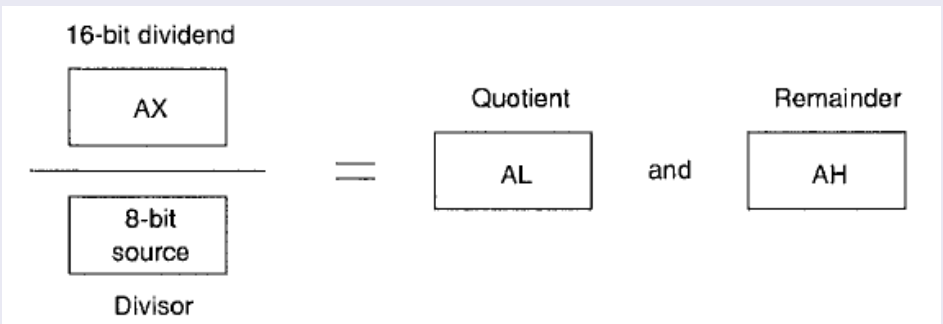
- Instruções de multiplicação são lentas
- Combinação de somas e deslocamentos pode ser mais eficiente

```
add     EAX,EAX     ; EAX = 2y
mov     EBX,EAX     ; EBX = 2y
add     EAX,EAX     ; EAX = 4y
add     EAX,EAX     ; EAX = 8y
add     EAX,EBX     ; EAX = 10y
```

- Somente necessita 5 ciclos de clock enquanto a instrução mul necessitaria de mais de 10 ciclos de clock

Divisão Inteira

- Diferentes instruções para divisões com e sem sinal
- Resultado consiste em dois valores: quociente e resto
  - div source
  - idiv source



- Tamanho do source define tamanho do dividendo, source pode ter 8, 16 ou 32 bits

Divisão Inteira com sinal

- Para executar a divisão inteira com sinal, é necessário fazer uma extensão do sinal no local onde o resultado deve ser armazenado
- Instruções de extensão do sinal:
  - cbw (converte byte para word) AL $\rightarrow$ AH
  - cwd (converte word para doubleword) AX $\rightarrow$ DX
  - cdq (convert doubleword para quadword) EAX $\rightarrow$ EDX

Divisão Inteira com sinal

```
mov AL,-95
cbw          ; AH = FFH
mov CL,12
idiv CL
```

→

AL = - 7 (F9H)
AH = - 11 (F5H)

```
mov AX,-5147
cwd ; DX = FFFFH
mov CX,300
idiv CX
```

→

AX = -17 (FFEFH)
DX = -47 (FFDIH)

Próxima Aula

Continuação IA-32 (Operandos Lógicos, Controles de Fluxo)