Assembly NASM/IA-32

Introdução

- Que assembly para o IA-32?
- Estrutura de um programa NASM
- Assembler e Linker
- Modos de endereçamento
- Instruções
  - ✓ Movimento de dados
  - ✓ Instruções aritméticas e lógicas
  - ✓ Saltos

Que Assembly para o IA-32?

Formato Intel/Microsoft

```
lea   eax, [ecx+ecx*2]
sub   esp, 8
cmp   dword [ebp-8], 0
mov   eax, dword [eax*4+0x100]
```

Formato Gnu

```
leal  (%ecx,%ecx,2), %eax
subl  $8, %esp
cmpl  $0, -8(%ebp)
movl  $0x100(,%eax,4), %eax
```

- Operandos em ordem inversa

  `mov destino, origem` `movl origem, destino`

- Constantes precedidas por '$' no formato Gnu (0x denota que a constante está em hexadecimal, o formato Intel também aceita 100h)

  `0x100` `$0x100`

- Gnu: o tamanho dos operandos é explícito na mnemónica

  `sub` `subl`

- A sintaxe para os modos de endereçamento diferem

  `[eax*4+100h]` `$0x100(,%eax,4)`

Que Assembly para o IA-32?

- Vamos utilizar o formato Intel, mais propriamente o assembly NASM (Netwide Assembler)
  - ✓ A sintaxe do NASM é, na nossa opinião, a mais fácil de ler
  - ✓ O NASM está disponível para Linux e Windows
    - ✓ Os programas apenas diferem na interacção com o SO
  - ✓ Mais tarde iremos utilizar o TASM (Borland Turbo Assembler) sobre o DOS (o sistema operativo da Microsoft anterior ao Windows)
    - ✓ Ambos os assemblies seguem o formato Intel e são (em quase tudo) iguais
    - ✓ Excepções são a declaração de variáveis não inicializadas e algumas directivas ao assembler
  - ✓ O livro das teóricas usa o MASM/TASM nos capítulos, tendo um apêndice dedicado ao NASM; o das práticas usa o NASM

Um Programa em Assembly NASM

- Um programa NASM é composto por várias secções
  - ✓ `.data` → declarar variáveis inicializadas
  - ✓ `.bss` → reservar espaço para variáveis não inicializadas
  - ✓ `.text` → programa

- A secção de texto deve começar pela etiqueta `_start` declarada como global
- Símbolos são declarados como globais (públicos) através da directiva `global`
- Comentários são denotados por ; e vão até ao fim da linha

Um Exemplo Inicial

```
global _start
```

Definir o símbolo _start como público para outros ficheiros

```
section .data
  A: dd 24
  B: dd 1
  C: dd 0
```

Secção de dados

```
section .text
_start: mov eax, [A]
        add eax, [B]
        mov [C], eax

        mov eax, 1
        mov ebx, 0
        int 0x80
```

Secção de código

Chamada ao sistema Linux para terminar o programa

Etiquetas (*labels*)

- Em vez de usar endereços, no *assembly* etiqueta-se as posições de memória
  - ✓ Exemplo:

```
ciclo:  add eax, ebx
        jmp ciclo
```

  - ✓ `ciclo` → representa o endereço de memória onde fica a instrução '`add eax, ebx`'

- Usa-te também para etiquetar posições de memória que contém dados: variáveis
  - ✓ Exemplo, carregar para eax o conteúdo de Var

```
mov eax, [Var]
```

Secção .data

- Declaração de variáveis inicializadas:

```
[etiqueta]  directiva_define   valor [, valor]
```

Entre parênteses → opcional

- Directivas
  - ✓ **DB** → Define Byte – Reserva 1 byte
  - ✓ **DW** → Define Word - Reserva 2 bytes
  - ✓ **DD** → Define Doubleword - Reserva 4 bytes
  - ✓ **DQ** → Define Quadword - Reserva 8 bytes
  - ✓ **DT** → Define Ten Bytes - Reserva 10 bytes
- Exemplos:

```
A: DD    10     ; inteiro a 4 bytes com valor 10
C: DD    3.2    ; real a 4 bytes com valor 3.2
C: DB    'a'    ; inteiro a 1 byte com o código
                ; ASCII de 'a'
```

Secção .data

- Declaração de variáveis múltiplos valores podem ser abreviadas
  - ✓ Exemplo:

```
Message:    DB    'h'
            DB    'e'
            DB    'l'
            DB    'l'
            DB    'o'
            DB    0      ; terminador
```

  - ✓ Pode ser abreviado para:

```
Message:    DB    'h', 'e', 'l', 'l', 'o', 0
```

  - ✓ ou

```
Message:    DB    "hello", 0
```

Secção .text

- Código do programa

- Sequência de linhas do tipo:

  `[etiqueta]  mnemónica  [operandos]`

- Exemplo:

```
_start: mov eax, [A]
        add eax, [B]
        mov [C], eax

        mov eax, 1
        mov ebx, 0
        int 0x80
```

Secção .bss

```
global _start

section .data
  A: dd 24
  B: dd 1
section .bss
  C: resd 1

section .text
_start: mov eax, [A]
        add eax, [B]
        mov [C], eax
        mov eax, 1
        mov ebx, 0
        int OX80
```

Secção de dados não inicializados. Reserva de uma (1) double word - 32 bits

Secção .bss

- BSS → Block Started by Symbol
- Declaração de variáveis não inicializadas:

```
[etiqueta] directiva_reserve número_de_elementos
```

- Directivas
  - ✓ **RESB** → Reserve Byte – Reserva 1 byte
  - ✓ **RESW** → Reserve Word - Reserva 2 bytes
  - ✓ **RESD** → Reserve Doubleword - Reserva 4 bytes
  - ✓ **RESQ** → Reserve Quadword - Reserva 8 bytes
  - ✓ **REST** → Reserve Ten Bytes - Reserva 10 bytes
- Exemplos:

```
A: RESB  1      ; reserva espaço para 1 byte
B: RESD  10     ; reserva espaço para 10*4 bytes
```

Substituição de Texto

```
SYS_EXIT equ 1
LINUX_SYSCALL equ 0x80
global _start
section .data
  A: dd 24
  B: dd 1
section .bss
  C: resd 1
section .text
_start: mov eax, [A]
        add eax, [B]
        mov [C], eax
        mov eax, SYS_EXIT
        mov ebx, 0
        int LINUX_SYSCALL
```

Constantes simbólicas

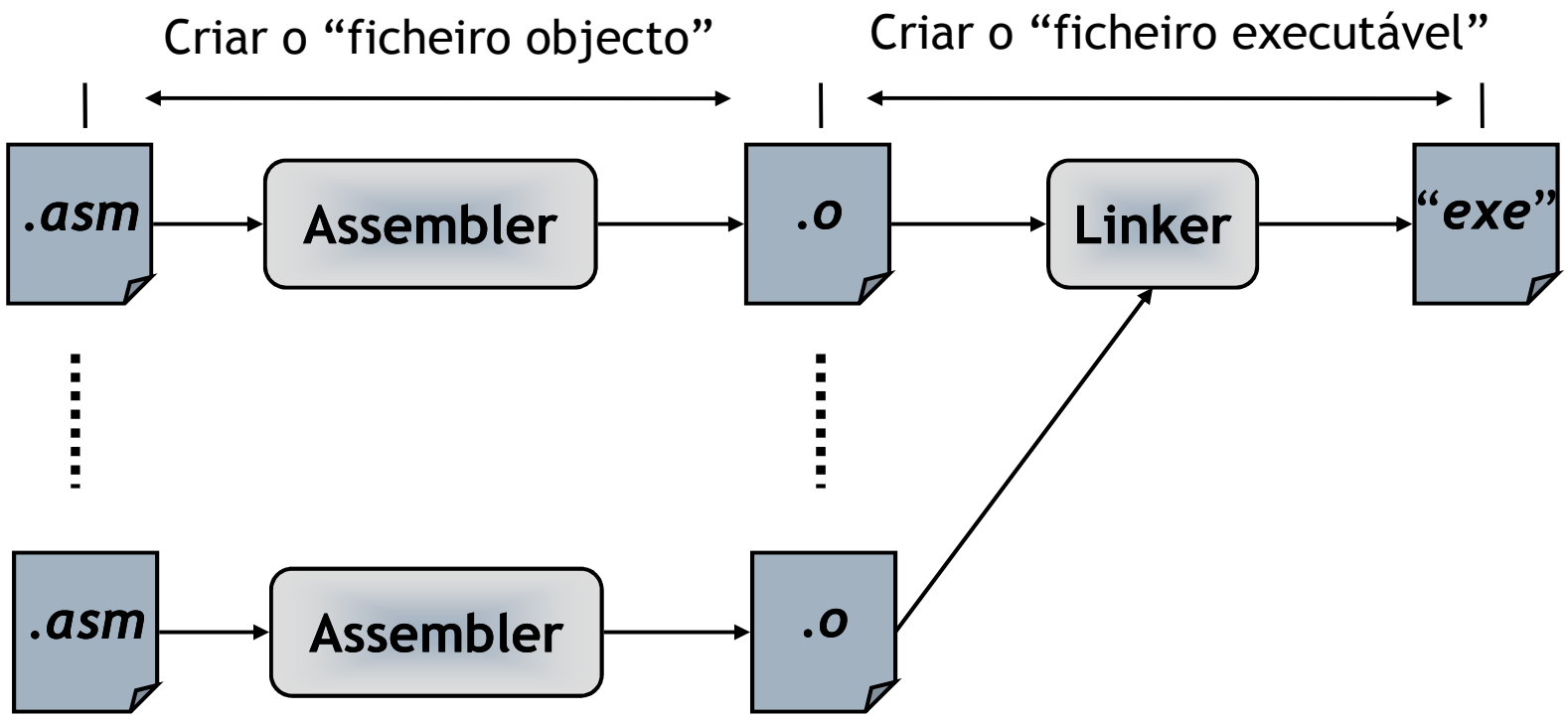
Chamada ao sistema Linux para terminar o programa

Substituição de Texto

- Constantes definidas com a pseudo-instrução EQU
  - ✓ O assembler substitui as ocorrências da etiqueta pelo seu valor

- Torna o código mais legível

Assemblar e Linkar um Programa



Podem ser vários ficheiros fonte

Um Exemplo em Assembly

- Considere uma subrotina que converte um valor de graus para radianos, equivalente à seguinte função C:

```
float graus_para_rad (int x) {
 return x*3.1415/180;
}
```

  - ✓ Não se preocupe em perceber o código nesta altura

```
graus_para_rad:
   push    ebp
   mov     ebp, esp
   sub     esp, 4
   fild    dword [ebp+8] ; argumento x
   fld     qword [0x10]  ; carregar 3.1415 que está no end 0x10
   fmulp   st1, st       ; calcular x*3.1415
   fld     qword [0x18]  ; carregar 180 que está no end. 0x18
   fdivrp st1, st        ; calcular x*3.1415/180
   fstp    dword [ebp-4]
   fld     dword [ebp-4]
   ret
```

Assembler

- O assembler tem o papel de traduzir as mnemónicas (linguagem assembly) no respectivo código máquina do processador.
- Podemos escrever (em assembly) instruções que não existem no ISA (o assembler verifica isso, e dará o respectivo erro)
- Existem outras instruções que são destinadas ao próprio programa assembler (p.ex. macro, endm)
- Para um ISA podem existir vários “assemblies”
  - ✓ Podemos definir um nós próprios, desde que implementemos o seu assembler

Código Máquina em Hexadecimal

- Código da função graus_para_rad

Endereços	Código máquina
00:	55
01:	89 e5
03:	83 ec 04
06:	db 45 08
09:	dd 05 10 00 00 00
0f:	de c9
11:	dd 05 18 00 00 00
17:	de f9
19:	d9 5d fc
1f:	d9 45 fc
20:	c3

Constante 0x10
em little endian

Opcode

Código máquina de
fld qword [0x10]
Carregar valor 3.1415
que está no endereço
0x10
O que é que está no
endereço 0x10?

Nota: **qword** são 8 bytes
no assembly que usámos

Código Máquina em Hexadecimal

- Código da função graus_para_rad

Endereços	Código máquina
00:	55
01:	89 e5
03:	83 ec 04
06:	db 45 08
09:	dd **05 10 00 00 00**
0f:	**de c9**
11:	**dd** 05 18 00 00 00
17:	de f9
19:	d9 5d fc
1f:	d9 45 fc
20:	c3

Código? Isto não é o valor 3.1415!

Na realidade nesta altura do processo de compilação ambos os dados e o código começam na posição 0.

Assim sendo, a instrução refere-se ao endereço 0x10 na secção de dados

Linker (ou Ligador)

- Liga o código todo da aplicação e faz a translação dos endereços
- Só neste passo é que é ligado (linkado) o código de toda o programa

Endereços	Código máquina
0x080483c4:	55
0x080483c5:	89 e5
0x080483c7:	83 ec 04
0x080483ca:	db 45 08
0x080483cd:	dd 05 **10 85 04 08**
0x080483d4:	de c9
0x080483d6:	dd 05 18 85 04 08
0x080483dc:	de f9
0x080483de:	d9 5d fc
0x080483e1:	d9 45 fc
0x080483e4:	c3

O valor mudou?

Na realidade este código máquina refere-se ao assembly

```
fld qword 0x08048510
```

Com a ligação a secção de dados começa no endereço `0x08048500`, enquanto que o código começa em `0x080483c4`

Ficheiros Objecto

- Os ficheiros objecto também são estruturados, seguem um dado formato

- Cada sistema operativo utiliza um formato em particular
  - ✓ É preciso saber para que formato assemblar
  - ✓ Nas aulas práticas vamos trabalhar em ambiente Linux
    - ✓ Este sistema operativo utiliza o formato ELF (Executable and Linkable Format)
  - ✓ O Windows utiliza o formato PE (Portable Executable)
  - ✓ O MacOS utiliza o formato Mach-O (Mach Object)

Assemblagem e Linkagem no Linux

- O NASM assembla para vários formatos de ficheiro objecto, exemplos são:
  - ✓ ELF 32 bits → elf32
  - ✓ ELF 64 bits → elf64
  - ✓ Windows 32 bits → win32
  - ✓ Windows 64 bits → win64
  - ✓ MachOS → macho
- Exemplo para Linux sobre IA-32

```
nasm –f elf32 ficheiro1.asm
…
nasm –f elf32 ficheiron.asm
ld –o prog ficheiro1.o … ficheiron.o
```

Modos de Endereçamento

- Existem vários modos de endereçar (identificar onde estão) os operandos de uma instrução
- Modos que endereçam posições de memória calculam um endereço denominado ***endereço lógico ou efectivo***
- Os ISA e assemblies normalmente não oferecem todos os modos existentes
  - ✓ Abordagem CISC
    - ✓ Facilidade de programação
      - ✓ Formas variadas de referir os operandos facilitam a vida ao programador
    - ✓ Menor número de instruções
      - ✓ Instruções com maior expressividade fazem com que os programas tenham menos instruções (embora mais longas)

Modos de Endereçamento

- Abordagem RISC:
  - Instruções mais curtas
    - Limitando o número de operandos (ou obrigando a que sejam registos)
    - Mais rápido, mas mais instruções no programa
  - Mais bits para constantes
    - k bits permitem representar 2k valores
    - logo mais bits é melhor, mas alonga as instruções
  - Fases de "fetch" e descodificação mais rápidas
    - Quanto menos operandos mais rápida e simples é a fase de fetch e obtenção dos operandos
    - Instruções simples simplificam a unidade de controlo e permite mais optimizações

Endereçamento Imediato

- O operando faz parte da própria instrução
  - ✓ Trazido para o CPU (para o *Instruction Register*), quando este faz *fetch* da instrução
  - ✓ Sintaxe NASM (constante em decimal)

```
mov eax, 5
```

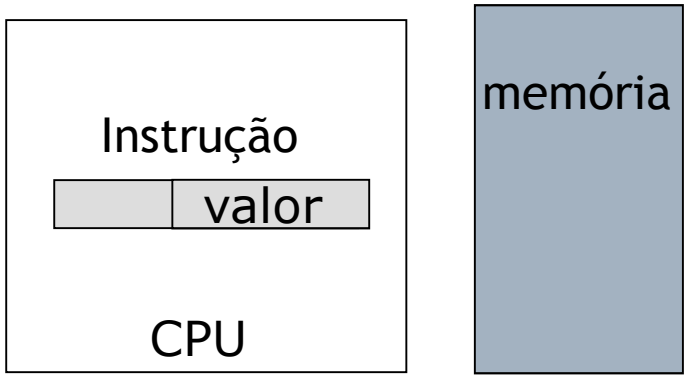
```
eax ← 5
```

  - ✓ Constante em hexadecimal

```
mov eax, 5h
```

  - ✓ Constante em binário

```
mov eax, 101b
```



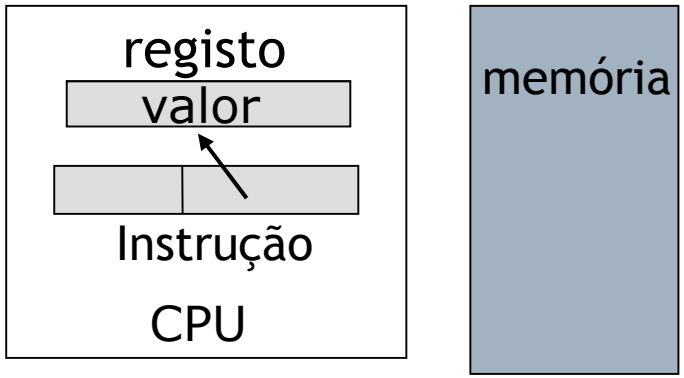
- Serve para especificar constantes na instrução
- Não é absolutamente necessário, mas pode reduzir o tamanho do programa e aumentar a rapidez da sua execução

Endereçamento por Registo

- O operando está contido num dos registos do CPU
  - ✓ Pressupõe que uma instrução anterior deixou o registo com o valor desejado
  - ✓ Sintaxe NASM:

```
mov eax, edx
```

eax ← edx



- Acesso muito rápido ao operando

- Limitado pelo número de registos do CPU

Endereçamento Directo a Memória

- O endereço efectivo está contido na instrução
  - ✓ O endereço na instrução indica a posição de memória onde está o operando
  - ✓ Sintaxe NASM:

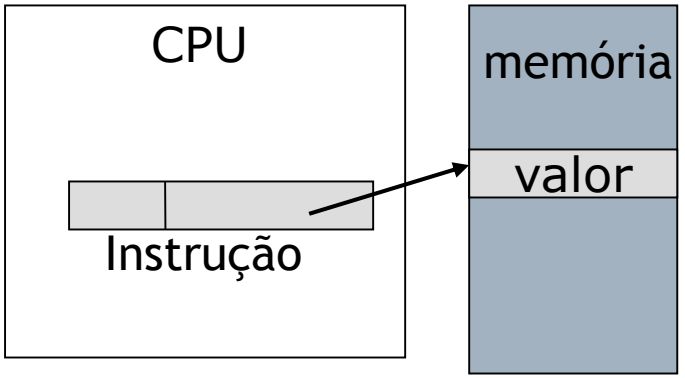
```
mov eax, [100]
```

eax ← Mem[100]

```
mov eax, [x]
```

eax ← Mem[x]

o assembler resolve o símbolo x

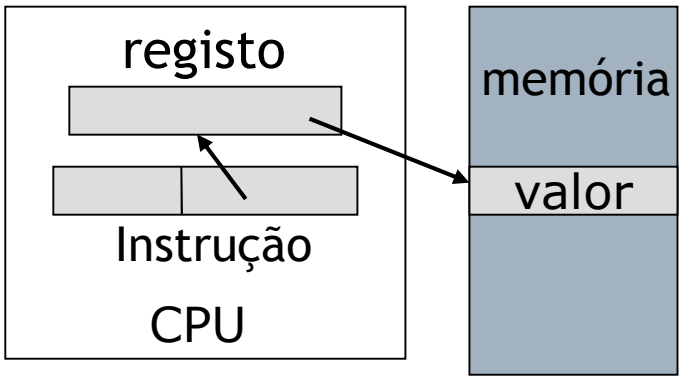


- É obrigatório saber que posição de quer aceder quando se escreve o programa
- A instrução contém o endereço, com o número de bits suficiente para alcançar toda a memória real

Endereçamento Indirecto por Registo

- Um registo contém o endereço de memória onde está o operando
  - ✓ Pressupõe que uma instrução anterior deixou o registador com o endereço desejado
  - ✓ Sintaxe NASM:

```
mov eax, [ebx]
    eax ← Mem[ebx]
mov eax, [bx]
    é válido no IA-32? Não
```



- O registo referencia a memória (apontador ou *pointer*)
  - ✓ Facilita a programação (por exemplo para percorrer vectores)

Segmentos por Omissão

- Segmentos utilizados por omissão
  - ✓ Fetch
    - ✓ Base → CS
  - ✓ Operações sobre a pilha: push e pop
    - ✓ Base → SS
  - ✓ Operações sobre dados
    - ✓ Base → DS

- Não usar o registo de segmento por omissão
  - ✓ Notação NASM: Base:Deslocamento
    - ✓ Exemplo: `mov [ES:EBX], 10`

Movimento de Dados

mov dest, orig

dest ← orig

- Copia o valor em *orig* para a posição dada por *dest*
- Copia e não move, o valor de origem não é destruído
- Exemplos:

```
mov eax, 5   ; coloca 5 em eax
mov ax, bx   ; coloca o conteúdo de bx em ax
```

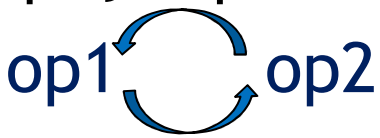
- *Para load ou store usa-se sempre a mnemónica* **mov**
  - ✓ leitura (*load*): mov ebx, [276534]

    ebx ← Mem[276534..276537] (ebx → 4 Bytes)
  - ✓ escrita (*store*): mov [82763], cx

    Mem[82763.. 82764] ← cx (cx → 2 Bytes)

Movimento de Dados

xchg op1, op2



- Troca os conteúdos de op1 e op2
- Exemplos:

```
xchg eax, ebx ; troca os conteúdos de eax e ebx
xchg [x], bx  ; coloca o conteúdo de x pelo
              ; conteúdo de bx
```

- Só um dos operandos pode ser uma posição de memória
- Caso ambos sejam registos têm de ter o mesmo tamanho

Operações Lógicas e Aritméticas

- Operações:
  - ✓ aritméticas: **`add, sub, inc, dec, mul, cmp, …`**
  - ✓ lógicas: **`and, or, xor, not, …`**
- Realizadas pela ALU (Arithmetic Logic Unit)
- Cada operação altera o estado das *flags*
- *Flags principais*:

  CF – Carry Flag – resultado da última operação tem carry

  OF – Overflow Flag - resultado da última operação tem resultou num overflow

  ZF – Zero Flag - resultado da última operação é 0

  SF – Sign Flag - resultado da última operação é negativo

  PF – Parity Flag – paridade do byte menos significativo do resultado

ADD – Adição de Números

add op1, op2

op1 ← op1 + op2

- Soma os bits de ambos os operandos
- O resultado é guardado na primeira referência (que não pode ser imediata)
- Altera as *flags* de acordo
- Exemplos:

```
add eax, 5    ; 2º op - endereçamento imediato a 32bits
add ax, bx    ; 2º op - endereçamento por registo a 16bits
add al, [100] ; 2º op - endereçamento directo a 8bits
```

SUB – Subtracção de Números

sub op1, op2

op1 ← op1 - op2

- Subtrai os bits de op2 a op1
- O resultado é guardado na primeira referência (que não pode ser imediata)
- Altera as *flags* de acordo
- Exemplos:

```
sub eax, 5  ; 2º op - endereçamento imediato a 32bits
sub ax, bx  ; 2º op - endereçamento por registo a 16bits
sub al, [100]  ; 2º op - endereçamento directo e 8bits
```

Carry Flag - CF

- CF a 1: se há transporte do bit mais significativo

- Interpretação do programa:
  - ✓ Uma operação aritmética sobre números sem sinal deu um resultado fora do domínio de valores válidos

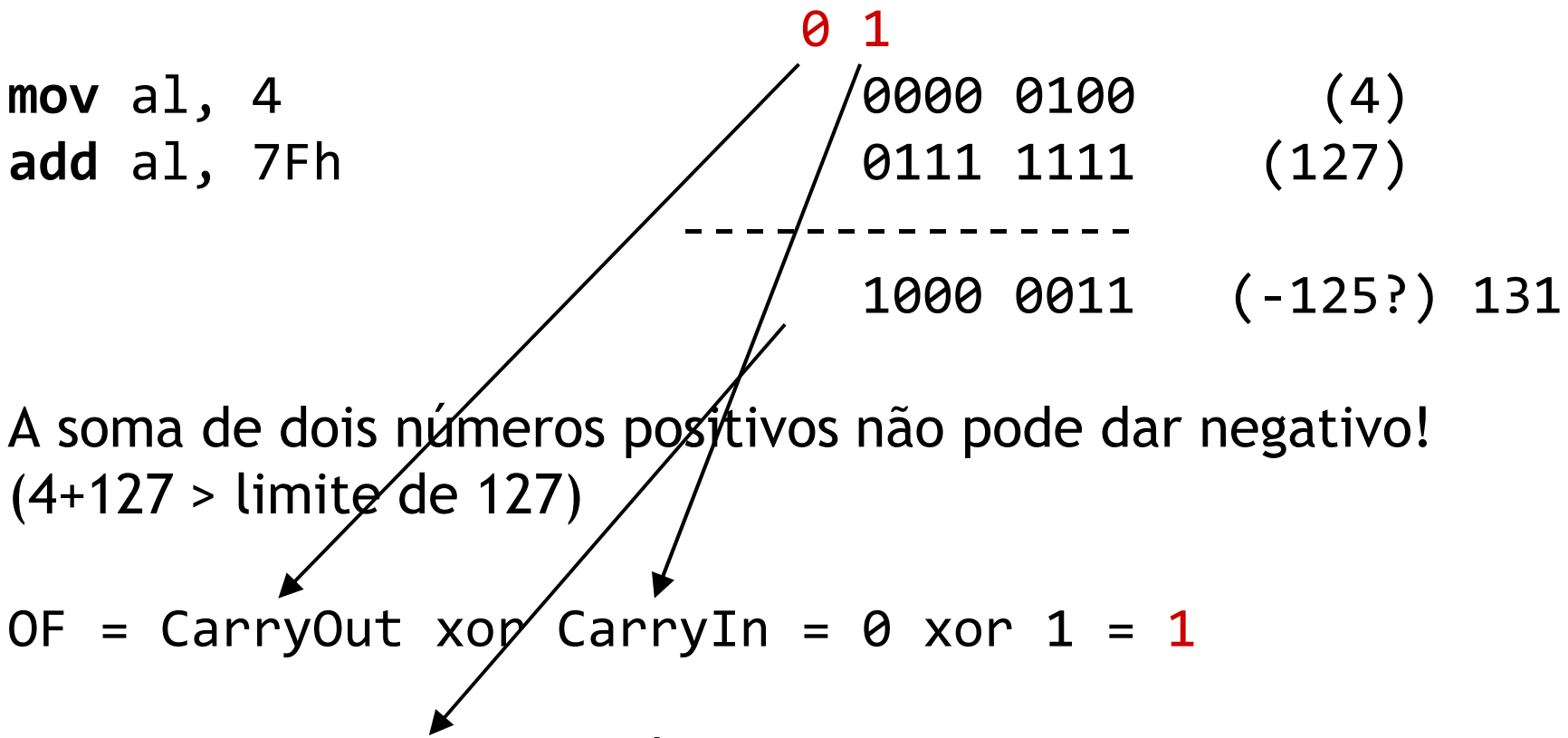
```
                    1 1111 100
mov al, FFh           1111 1111          FFh  (255)
add al, 4             0000 0100          04    ( 4)
                  ---------------      -----
                    1 0000 0011      1 03h  (259)
```

CF = 1

al = 0000 0011 (o limite válido é 255)

Overflow Flag - OF

- OF a 1: uma operação aritmética sobre números com sinal dá um resultado fora do domínio de valores válidos

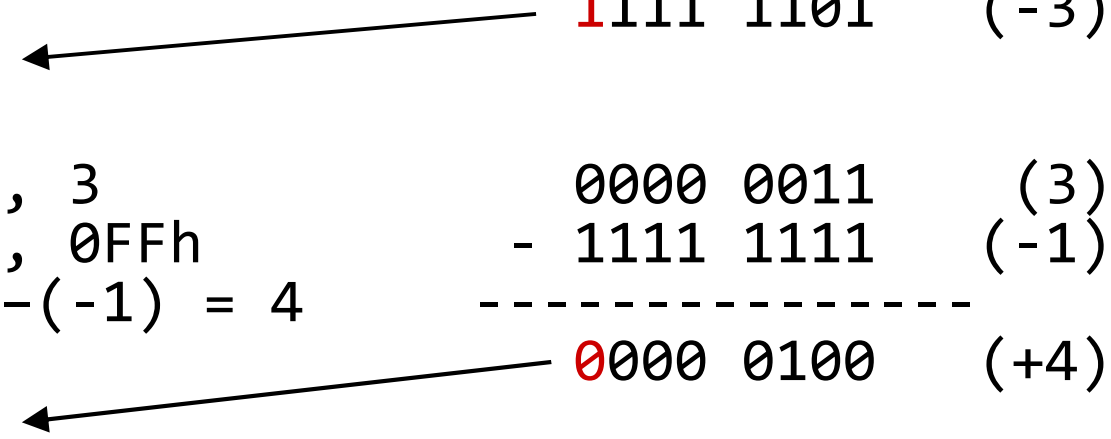


Sign Flag - SF

- SF = ao valor do bit mais significativo do resultado, que é o bit de sinal na representação em complemento a 2
  - ✓ SF = 0 se positivo,
  - ✓ SF = 1 se negativo

```
mov al, 2                0000 0010    (2)
sub al, 5              - 0000 0101    (5)
al = 2-5 = -3          ---------------
                         1111 1101   (-3)
SF = 1

mov al, 3                0000 0011    (3)
sub al, 0FFh           - 1111 1111   (-1)
al = 3-(-1) = 4        ---------------
                         0000 0100   (+4)
SF = 0
```



Zero Flag - ZF

- ZF = 1 se resultado da última operação efectuada pela ALU for zero

- ZF = 0 se o resultado for diferente de zero

```
mov ax, 2
sub ax, 2       → ZF = 1

mov ax, 3
sub ax, 2       → ZF = 0
```

ADD e as Flags

```
    1110   14   -2
+   1010   10   -6
--------   ----  ----
 1  1000    8?   -8
CF=1  ZF=0  OF=0  SF=1
```

Interpretação sem e com sinal

```
    0110    6   +6
+   1101   13   -3
 --------  ----  ----
 1  0011    3?    3
CF=1  ZF=0  OF=0  SF=0
```

```
    1001    9    -7
+   0100    4     4
 --------  ----   ----
    1101   13    -3
CF=0  ZF=0  OF=0  SF=1
```

- Adição de 2 números com sinais diferentes nunca produz *overflow*
- Adição de 2 números com sinais iguais só dá *overflow* se o sinal do resultado for diferente

ADD e as Flags

- Exemplos em que o resultado é inválido quando se interpretam os operandos como tendo sinal (em complemento para 2)
  - ✓ A OF e CF indicam que o resultado está errado, dependendo interpretação com ou sem sinal

```
   1101   13   -3          0101   5    +5
+  1010   10   -6        +  0110   6    +6
------- ----- ----       -------- ---- -----
 1 0111    7?  +7?           1011  11    -5?
CF=1 ZF=0 OF=1 SF=0      CF=0 ZF=0 OF=1 SF=1

   1000    8  -8          0111    7    +7
+  1000    8  -8        +  0111    7    +7
------------------       -------- ----- -----
 1 0000    0?  0?           1110   14    -2?
CF=1 ZF=1 OF=1 SF=0      CF=0 ZF=0 OF=1 SF=1
```

ADD com Carry

adc op1, op2

op1 ← op1 + op2 + CF

- Soma os bits de ambos os operandos mais o bit da Carry Flag

- Útil para somar valores a 64 bits no IA-32
  - ✓ IA-32 têm registos de dados com capacidades de 8, 16 e 32 bits
  - ✓ No entanto pode-se suportar aritmética a 64 bits concatenando registos
  - ✓ Exemplo: EDX:EAX

ADD com Carry

- Como realizar a soma de dois valores a 64bits, suponhamos EBX:EAX e EDX:ECX
  - ✓ EBX:EAX = EBX:EAX + EDX:ECX

```
add     eax, ecx
adc     ebx, edx
```

- O carry é propagado para a segunda soma

Incremento e decremento

inc op

op ← op + 1

dec op

op ← op - 1

- Incrementa ou decrementa o operando (que não pode ser imediato)
- Por opção estas instruções não afectam a CF
  - ✓ Usam-se outras flags
  - ✓ Ex: no **inc** pode-se testar a ZF

```
  1111 1111      255
+ 0000 0001    +   1
------------   -----
  0000 0000        0
```

- Exemplos:

```
inc eax        ; endereçamento por registo a 32bits
dec dh         ; endereçamento por registo a 8bits
```

Carry Flag na Subtracção

```
mov al, 2                          0000 0010    (2)   (2)
sub al, 0FFh                     - 1111 1111  (255) (-1)
                      (Borrow)  1 1111 111
                                ---------------
                                   0000 0011  (???)  (3)
                                                Sem   Com
                                                sinal sinal
```

al = 0000 0011

CF = 1 (2-255 em números sem sinal não é possível)

OF = 0 2-(-1) = 3 (resultado válido em números com sinal)

Carry Flag na Subtracção

- X-Y é equivalente a X+(-Y) , com -Y obtido por complemento para 2:

```
  + 4     0100              0100   4+(-3)
- + 3   - 0011            +  1101  (comp. 2)
---------------           ----------
  + 1     0001             1  0001
```

Note: no CPU não aparece CF neste caso!

```
  + 3     0011              0011  3+(-4)
- + 4   - 0100              1100  (comp. 2)
--------------------        ----------
  - 1   1 1111                0 1111
```

Neste caso tem de aparecer CF = 1

borrow

SUB com Borrow

sbb op1, op2

op1 ← op1 - op2 - CF

- Equivalente ao ADD com Carry
- Útil para subtrair números a 64 bits no IA-32

Subtracção Como Meio de Comparação

- X-Y equivale a X+(-Y)
- Ao fazer X-Y, se números sem sinal:
  - ✓ CF é usada para indicar transbordo (borrow) na interpretação como números SEM SINAL
    - ✓ É a negação do carry natural da soma
  - ✓ Se a carry flag está activada houve borrow então: X < Y
  - ✓ Senão: X>=Y
- Ao fazer X-Y, se números com sinal:
  - ✓ Se há overflow então a relação entre X e Y depende dos respectivos sinais…
- Permite comparar 2 números → **cmp**

CMP - Compare

cmp op1, op2

- Faz op1-op2 mas não guarda o resultado, apenas altera as flags
  - ✓ Põe CF = 1 se houver bit de *borrow*
    - ✓ op1 < op2 para números sem sinal
  - ✓ Põe OF = 1 se houver *overflow*
    - ✓ *O caso de números com sinal mais à frente…*
  - ✓ Põe ZF = 1 se os operandos forem iguais

- Conclusão: **cmp** compara os dois operandos
  - ✓ Utiliza-se em vez do **sub** quando se pretende alterar apenas as *flags*
- Exemplo de utilização: antes dos saltos condicionais

Multiplicação e Divisão

- IA-32 suporta multiplicação e divisão
- Sem sinal (32 bits)
  - ✓ **`mul`** *op* — `edx:eax ← eax * op`
    (edx e eax são sempre usados implicitamente)
  - ✓ **`div`** *op* — `eax ← edx:eax / op`
    EDX fica com o resto da divisão
- Com sinal (32 bits)
  - ✓ **`imul`** *op* — `edx:eax ← eax * op`
    (edx e eax são sempre usados implicitamente)
  - ✓ **`idiv`** *op* — `eax ← edx:eax / op`
    EDX fica com o resto da divisão
- edx e eax são sempre usados implicitamente
- Também existem as operações em 16 e 8 bits

Exemplos

```
mul ebx   ; endereçamento de registo a 32 bits
mul bx    ; endereçamento de registo a 16 bits
mul 6     ; endereçamento imediato a quantos bits?
mul [Var] ; endereçamento directo a quantos bits?
mul [ebx] ; endereçamento indirecto por registo a
          ; quantos bits?
```

- Não se consegue inferir o número de bits a transmitir a partir dos operandos.
  - ✓ Resultado de assemblar no NASM:

```
error: operation size not specified
```

Especificação do Tamanho do Operando

- É necessário especificar o tamanho do operando
  - ✓ BYTE → 8 bits
  - ✓ WORD → 16 bits
  - ✓ DWORD → 32 bits
  - ✓ QWORD → 64 bits
  - ✓ TWORD → 80 bits

- Exemplos:

```
mul dword 6     ; endereçamento imediato a 32 bits
mul word [Var]  ; endereçamento directo a 16 bits
                ; O resultado é colocado em dx:ax
mul byte [ebx]  ; endereçamento indirecto por
                ; registo a 8 bits
                ; O resultado é colocado em ax
```

Instruções lógicas

- Tratam os *bytes* como vectores de bits
  - ✓Onde cada bit: 1 → ***true*** ; 0 → ***false***
  - ✓As *flags* reflectem o resultado
- Exemplos de instruções (para cada bit dos operandos):

not *op*

op ← not *op*

and op1, op2

op1 ← *op1* and *op2*

or op1, op2

op1 ← *op1* or *op2*

xor op1, op2

op1 ← *op1* xor *op2*

"Ligar" e “Desligar" Bits

- **and** op1, op2
  - ✓ Põe bits individuais do 1º operando a 0, conforme indicado pelos zeros no 2º operando
  - ✓ Exemplo: **and** ah, 00011100b
    - ✓ Põe bits 0,1,5,6 e 7 de ah a 0 e mantém os restantes
    - ✓ ah = 000xxx00 onde xxx ficaram inalterados

- **or** op1, op2
  - ✓ Põe bits individuais do 1º operando a 1, conforme indicado pelos uns no 2º operando,
  - ✓ Exemplo: **or** ah, 11000000b
    - ✓ Põe bits 6 e 7 de ah a 1 e mantém os restantes
    - ✓ ah = 11xxxxxx onde xxxxxx ficam inalterados

Mais Operações

- **not** op
  - ✓ Complemento bit a bit dos bits do único operando
- **xor** op1, op2
  - ✓ Troca os bits individuais do 1º operando, conforme indicado pelos uns no 2º operando
  - ✓ Exemplo: **xor** ah, 00000011b
    - ✓ Complementa os bits 0 e 1 de ah e mantém os restantes
    - ✓ ah = xxxxxxCC
- **test** op1, op2
  - ✓ equivale ao **and** sem alterar o 1º operando
  - ✓ Exemplo: **test** ah, 01010101b
    - ✓ põe ZF a 1 se os bits 0,2,4 e 6 de ah estão a 0, mas não altera ah

Manipular Bits Individualmente

- Exemplos:
  - ✓ **`and`** `ax, 7FFFh`
    - ✓ Põe a 0 o bit 15 de ax
  - ✓ **`or`** `ax, 8000h`
    - ✓ Põe a 1 o bit 15 de ax
  - ✓ **`xor`** `ax, 8000h`
    - ✓ Complementa o bit 15 de ax
  - ✓ **`test`** `dl, 00000100b`
    - ✓ ZF = 1 se o bit 2 de dl for 0
    - ✓ ZF = 0 se o bit 2 de dl for 1
    - ✓ Equivale a fazer: **`and`** `dl, 00000100b` mas só afecta as *flags*. Não altera o registo dl

NEG – Negar (complemento a 2)

- **neg** op
  - op ← -op (complemento para 2 de op)
- Equivale a:
  - **not** op
  - **inc** op
- O que é que acontece quando se obtém o complementar a 2 do maior número negativo (em valor absoluto) ?
  - ✓ Exemplo a 4 bits: –(-8) = +8 → inválido → OF = 1

```
1000 (-8) →     0111
                +  1
               ------
                1000   (-8)
```

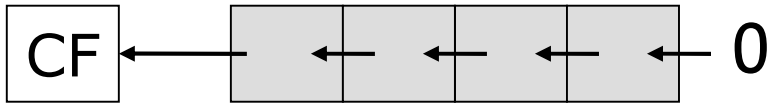
Shifts - desvio/deslocamento de bits

- Desviam os bits de um operando, uma ou mais posições, para a esquerda ou para a direita
- O último bit que sai fica na Carry Flag
- Pode operar sobre registos e sobre bytes em memória
- Servem para facilitar a selecção e operação sobre grupos de bits, parte do valor original

Desvios Lógicos

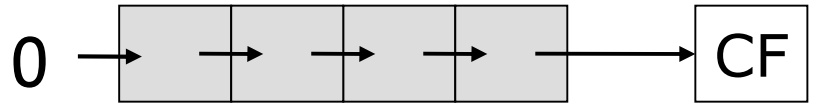
- Interpretam números sem signal

- **`shl`** op,n op ← *shiftLeft*(op, n)



equivale a multiplicar por 2^n

- **`shr`** op,n op ← *shiftRight*(op, n)



equivale a dividir por 2^n

Desvios Aritméticos

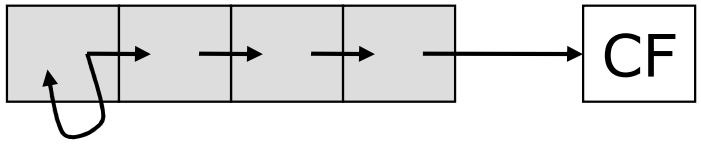
- Se manipulamos um número com sinal, ao desviar para a direita, devemos manter o sinal

- **`sal`** op,n é igual a **`shl`** op,n

  op ← *shiftAriLeft*(op,n)

- **`sar`** op,n

  op ← *shiftAriRight*(op,n)



mantém o sinal

Desvios Aritméticos

- Interpretam números com sinal:
  - ✓ Left: equivale a multiplicação*2
    - ✓ OF indica transbordo
  - ✓ Right: equivale a divisão/2
    - ✓ CF indica resto
    - ✓ O bit que entra é igual ao mais significativo → mantém o sinal

Desvios Lógicos vs Aritméticos

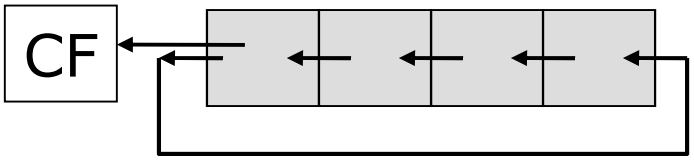
- Consideremos, a título de exemplo, números a 4 bits sem sinal:
  - ✓ Um desvio lógico de 1 posição sobre:
    - ✓ $0011 = 3_{(10)}$ tem como resultado $0110 = 6_{(10)}$ e CF = 0
    - ✓ $0110 = 6_{(10)}$ tem como resultado $1100 = 12_{(10)}$ e CF = 0
    - ✓ $1100 = 12_{(10)}$ tem como resultado $1000 = 8_{(10)}$ e CF = 1
      - ✓ Estamos na presença de um erro, indicado pela Carry Flag

- Consideremos agora números a 4 bits com sinal:
  - ✓ Um desvio aritmético de 1 posição sobre:
    - ✓ $0011 = 3_{(10)}$ tem como resultado $0110 = 6_{(10)}$ e CF = 0
    - ✓ $0110 = 6_{(10)}$ tem como resultado $1100 = -4_{(10)}$ e CF = 0
      - ✓ No entanto estamos na presença de um overflow, logo OF = 1
      - ✓ Estamos na presença de um erro, indicado pela Overflow Flag

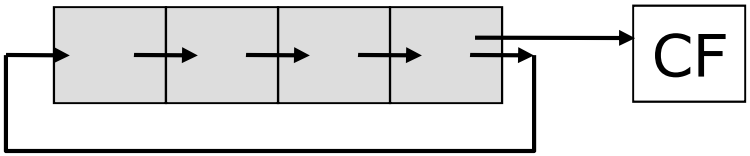
Rotações

- Não se perdem bits
  - ✓ O que sai entra do outro lado

- **`rol`** `op,n` op ← *rotLeft(op,n)*



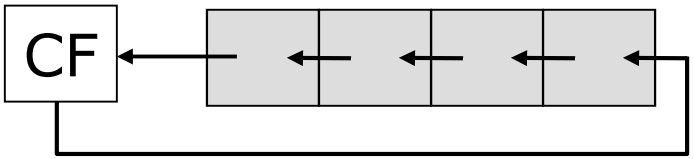
- **`ror`** `op,n` op ← *rotRight*(op,n)



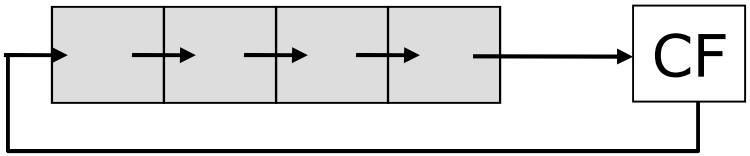
Rotações com Carry

- A Carry Flag é considerada na rotação

- **`rcl`** op,n op ← *rotLeftCarry(op,n)*



- **`rcr`** op,n op ← *rotRightCarry(op,n)*

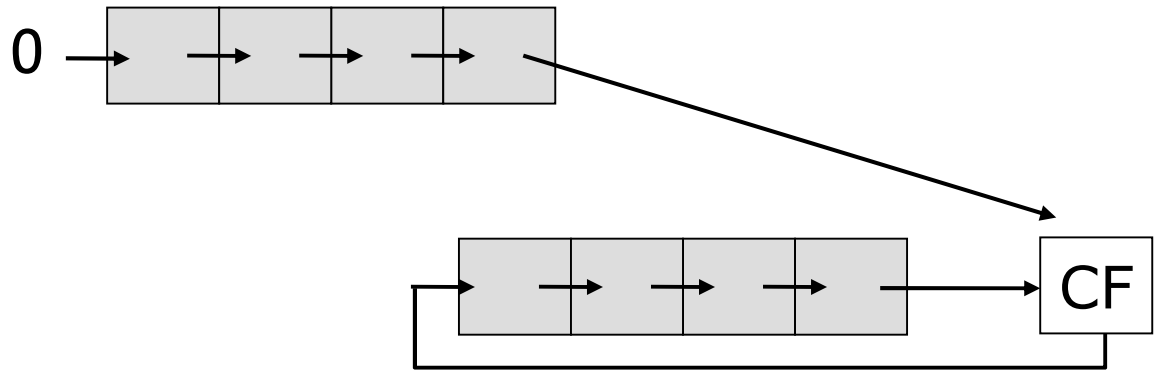


Rotações com Carry

- Útil na implementação de deslocamentos de números a 64 bits
  - ✓ Exemplo: EDX:EAX

```
shr     edx,1
rcr     eax,1
```

  - ✓ O bit que sai para a Carry Flag aquando do deslocamento sobre o EDX entra no lado mais significativo do EAX



Instruções de controlo

- Saltos (jumps): alteram o valor do EIP - Instruction Pointer (ou Program Counter)
  - ✓ Incondicionais

    `jmp etiqueta`

    eip ← endereço etiquetado
    - ✓ A próxima instrução executada é a que se encontra no endereço etiquetado
  - ✓ Condicionais
    - ✓ Primeiro testam códigos de condição (flags)
    - ✓ Exemplos:
      - ✓ `jz etiqueta` → só salta se a flag Zero estiver a 1
      - ✓ `jnz etiqueta` → só salta se a flag Zero estiver a 0
    - ✓ Outras instruções testam outras flags ou conjuntos de flags

Saltos Incondicionais

- Saltos directos
  - ✓ O alvo está codificado na própria instrução e é um deslocamento relativo à posição actual
  - ✓ Exemplo:

```
jmp etiqueta
```

- Salto indirecto
  - ✓ O alvo está codificado num registo e é um endereço absoluto
  - ✓ Exemplo:

```
jmp ecx
```

  - ✓ Úteis para implementar tabelas de saltos, necessárias, por exemplo, para codificar as instrução **switch** do C

Saltos Condicionais (sobre uma flag)

- Os que testam directamente uma *flag*

- Salta se:

Carry	CF=1	JC – Jump if Carry
	CF=0	JNC – Jump if Not Carry
Overflow	OF=1	JO – Jump if Overflow
	OF=0	JNO – Jump if Not Overflow
Signal	SF=1	JS – Jump if Sign
	SF=0	JNS – Jump if Not Sign
Zero	ZF=1	JZ – Jump if Zero
	ZF=0	JNZ – Jump if Not Zero

Saltos Condicionais (inteiros s/sinal)

- As relações entre inteiros sem sinal, são dadas, após **cmp** (ou **sub**) pelas *flags*:
  - ✓ após **sub** X,Y ou **cmp** X,Y
    - se X<Y → CF=1 *(borrow)*
    - se X=Y → ZF=1
    - se X>Y → CF=0 e ZF=0
  - ✓ Saltos condicionados por estas *flags*:
    - JB – Jump if Below (=JC)
    - JE – Jump if Equal (=JZ)
    - JA – Jump if Above (CF=0 e ZF=0)
    - JBE – Jump if Below or Equal (CF=1 ou ZF=1)
    - JAE – Jump if Above or Equal (=JNC)

Saltos Condicionais (aliases)

- Mais mnemónicas para as mesmas instruções:
  - ✓ JNBE = JA
  - ✓ JNB = JAE = JNC
  - ✓ JNA = JBE
  - ✓ JNAE = JB = JC
  - ✓ JNE = JNZ

Comparar Números com Sinal

- Sem sinal

1111 1111 = 255_{10}

0000 0000 = 0_{10}

Qual é maior?

255 > 0

Flag a testar?

Carry (CF), usando **sub/cmp**

- Com sinal

1111 1111 = -1_{10}

0000 0000 = 0_{10}

Qual é maior?

-1 < 0

Flag a testar?

necessitamos de testar flags diferentes

Exemplos de Comparações

- Usando cmp/sub para X>Y:

X	Y	X-Y	OF	SF
0111 (7)	0001 (1)	0110 (6)	0	0
0001 (1)	1110 (-2)	0011 (3)	0	0
0001 (1)	1001 (-7)	*1000 (-8?)*	1	1
1111 (-1)	1001 (-7)	0110 (6)	0	0

Exemplos de Comparações

- Usando cmp/sub para X<Y:

X	Y	X-Y	OF	SF
0001 (1)	0111 (7)	1010 (-6)	0	1
1000 (-8)	0001 (1)	*0111 (3?)*	1	0
1001 (-7)	0001 (1)	1000 (-8)	0	1
1001 (-7)	1111 (-1)	1010 (-6)	0	1

Conclusão

- Comparação usando **`cmp/sub`** entre números com sinal:

- *Quando X-Y>=0 (SF=0) então X>=Y, excepto se houver Overflow (OF=1)*
  - ✓Após **`sub`** `X,Y` ou **`cmp`** `X,Y`

Se X > Y	→ *Greater than:*	SF = OF and ZF = 0
Se X >= Y	→ *Greater or Equal:*	SF = OF
Se X < Y	→ *Less than:*	SF ≠ OF and ZF = 0
Se X <= Y	→ *Less or Equal:*	SF ≠ OF or ZF = 1

Saltos Condicionais (inteiros c/sinal)

- Saltos condicionais para as relações entre inteiros com sinal:

  JG – Jump if greater than (=JNLE)

  JGE – Jump if greater or equal (=JNL)

  JL – Jump if less than (=JNGE)

  JLE – Jump if less or equal (=JNG)

Comparando Números (sub/cmp)

- Sem sinal

1111 1111 = 255_{10}

0000 0000 = 0_{10}

Qual é maior?

255 > 0

- Com sinal

1111 1111 = -1_{10}

0000 0000 = 0_{10}

Qual é maior?

-1 < 0

Necessitamos de instruções que comparem flags diferentes. Exemplos:

Flag a testar:

Carry (CF)

ja /**jb** / etc

Flags a testar:

Sign (SF) e *Overflow* (OF)

jg / **jl** / *etc*

Codificação de instruções de alto-nível: if

- Como executar código diferente dependendo de uma condição?
  - ✓ condição → estado das flags
  - ✓ salto condicional (**jxx**)

```
if  (condicao)
  codigo_do_then;
else
  codigo_do_else;
```

```
    ; código da condição de modo
    ; a que altere as flags
    ; (exemplo cmp eax, 5)
    jxx bloco_else
    ; codigo_do_then
    jmp fim_if
bloco_else:
    ; codigo_do_else
fim_if:
```

Codificação de instruções de alto-nível: if

- Exemplo

```
if  (x < 5)
  codigo_do_then;
else
  codigo_do_else;
```

```
    mov eax, [x]
    cmp eax, 5
    jge bloco_else
    ; codigo_do_then (eax < 5)
    jmp fim_if
bloco_else:
    ; codigo_do_else (eax ≥ 5)
fim_if:
```

Codificação de instruções de alto-nível
while

- Ciclo com teste "à cabeça“
- Versão a 32 bits do código gerado pelo Turbo C (16 bits)

```
while (condicao)
  codigo_do_ciclo;
```

```
    jmp while_cond
while_body:
    ; codigo_do_ciclo
while_cond:
    ; código da condição de modo
    ; a que altere as flags
    ; (exemplo cmp eax, 5)
    jxx while_body
end_while:
```

Codificação de instruções de alto-nível
while

- Exemplo
- Versão a 32 bits do código gerado pelo Turbo C (16 bits)

```
while (x < 5)
  codigo_do_ciclo;
```

```
    jmp while_cond
while_body:
    ; codigo_do_ciclo
while_cond:
    mov eax, [x]
    cmp eax, 5
    jl while_body
end_while:
```

do-while

- Ciclo com teste "no fim“
- Versão a 32 bits do código gerado pelo Turbo C (16 bits)

```
do
   codigo_do_ciclo;
while (condicao);
```

```
do_body:
    ; codigo_do_ciclo
cond_test:
    ; código da condição de modo
    ; a que altere as flags
    ; (exemplo cmp eax, 5)
    jxx do_body
end_do_while:
```

do-while

- Exemplo
- Versão a 32 bits do código gerado pelo Turbo C (16 bits)

```
do
   codigo_do_ciclo;
while (x < 5);
```

```
do_body:
   ; codigo_do_ciclo
cond_test:
   mov eax, [x]
   cmp eax, 5
   jl do_body
end_while:
```

Codificação de instruções de alto-nível
for

- Ciclo com base num contador decrescente
- Versão a 32 bits do código gerado pelo Turbo C (16 bits)

```
for ( i=n; i>0; i-- )
  codigo_do_ciclo;
```

```
        mov esi, [n]
        jmp for_cond
for_body:
        ; codigo_do_ciclo
        dec esi
for_cond:
        or esi, esi
           ; o or garante a
           ; passagem pela ALU
        jge for_body
```

LOOP

- Salto com base num contador, para ciclos que executam pelo menos uma vez
  - ✓ Equivale a ‘dec cx’ seguido de ‘jnz’
- Usa implicitamente o registo CX ou ECX

```
for ( i=n; i>0; i-- )
  codigo_do_ciclo;
```

```
      mov ecx, [n]
inicio_for:
      ; codigo_do_ciclo
      loop inicio_for
```

Nota: tem de n > 0!