Arquitetura de Computadores

MIEI – 2021/22

DI-FCT/UNL

Processador Intel

Programação em Assembly: Introdução ao IA-32

Sumário

- Evolução dos processadores Intel
- Programação em Assembly: Introdução ao ISA IA-32

Bibliografia:

Bryant, O'Hallaron Computer Systems: A Programmer's Perspective, 2nd ou 3rd Ed, secções 3.1 a 3.6

João M. Fernandes, Essentials of computing systems, Coleção Educação | Ciências, Engenharia e Tecnologia, Cap 2. secção 4.1, 4.2.1 a 4.2.6 (https://doi.org/10.21814/uminho.ed.33)

Intel

- Fairchild Semiconductor e a Texas Instruments pioneiros nos circuitos integrados (*microchip*)

- Intel- (Integrated Electronics Corporation)
  - Fundada em 1968 por funcionários vindos da Fairchild
  - Micro-processador
    - um único circuito integrado contendo todas as funcionalidades da unidade central de processamento (CPU) do computador
      - Intel 4004 – µ-proc. de 4bits para calculadoras

Nota: Já existiam processadores e computadores!

Processadores Intel

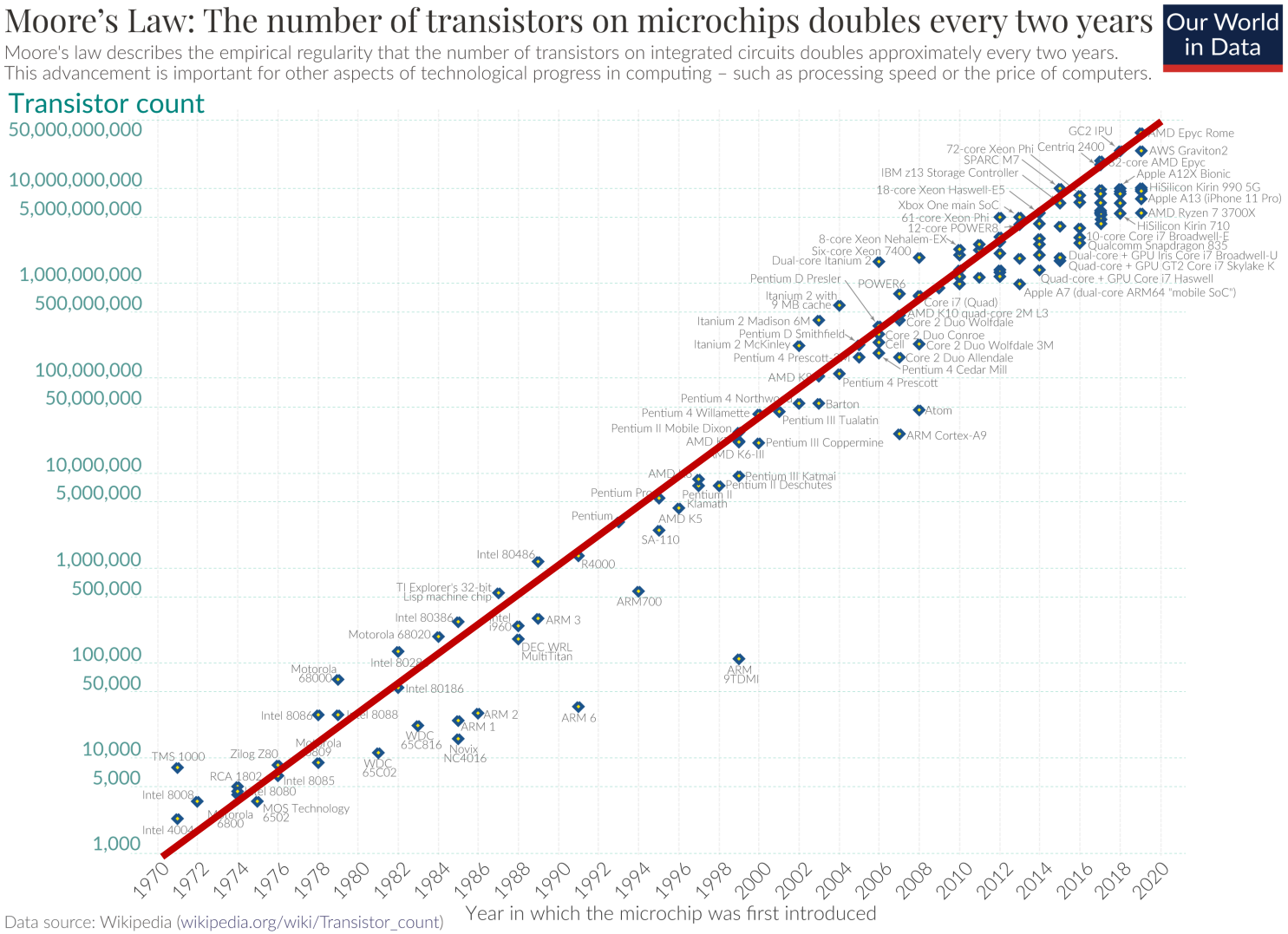
- Domínio quase total do Mercado de servidores, desktops e portáteis
- Desenho evoluiu ao longo do tempo
- Começou em 1978 com 8086
- Mais funcionalidades acrescentadas ao longo do tempo
  - Ainda suporta funcionalidades antigas, já obsoletas
- Complex Instruction Set Computer (CISC)
  - Muitas instruções diferentes com formatos distintos

Observação (“Lei”) de Moore

- A densidade dos componentes nos CI duplica a cada 2 anos

- A mesma “lei” tem sido extrapolada para:
  - Velocidade/poder dos CPU
  - Capacidade das memórias e discos

    (não para as suas velocidades!)



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world’s largest problems.

Impacto da Evolução

- O resultado da longa evolução:
  - Arquiteturas muito complexas e difíceis de compreender e programar
  - Instruções de tamanho variável (de 1 a 18? bytes…)
  - ISA muito complexos, difíceis de interpretar pelo hardware
  - Os programadores (e os compiladores) “refugiam-se” num subconjunto do ISA …
  - Surgem arquiteturas RISC – Reduced Instruction Set Computer
  - Nos CISC, existe uma “tradução” para micro-código, mais simples.
    - Este é executado pelos componentes do CPU
- O desempenho não vem só dos GHz...
  - Pipelines e Paralelismo interno
  - Paralelismo - multi-processadores

Exemplo Intel 80386 / IA-32 (1985)

- CPU com 32 bits de dados e de endereços, com BUS de 32 bits:
  - dimensão do IP, registos de dados, número de linhas no BUS de dados e de endereços: 32
  - endereça até 4G de bytes de memória (de cada vez)
  - transfere e opera até 4bytes de cada vez
- Base para os i486, Pentium/P5 (i586), Pentium Pro/P6 (i686), etc
- Será o nosso objeto de estudo
  - Falaremos de outras arquiteturas mais recentes mais tarde

Nomes dos registos (em *Assembly)*

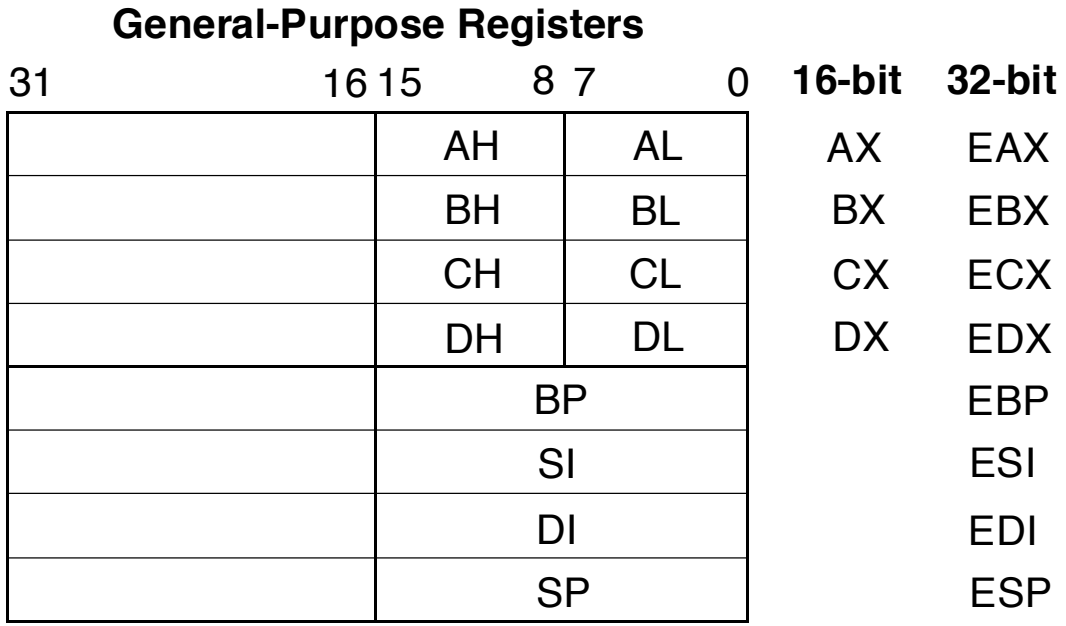


Figure 3-4. Alternate General-Purpose Register Names

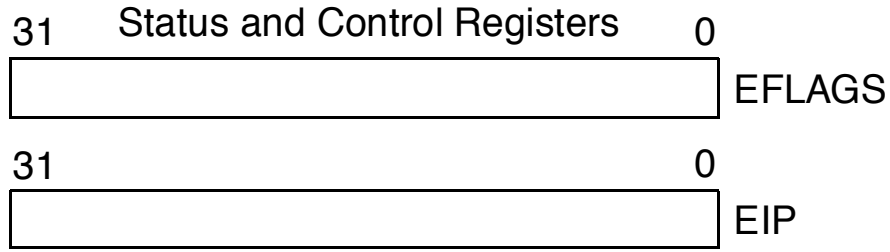


Figure 3-3. Application Programming Registers

Principais “tipos” de dados

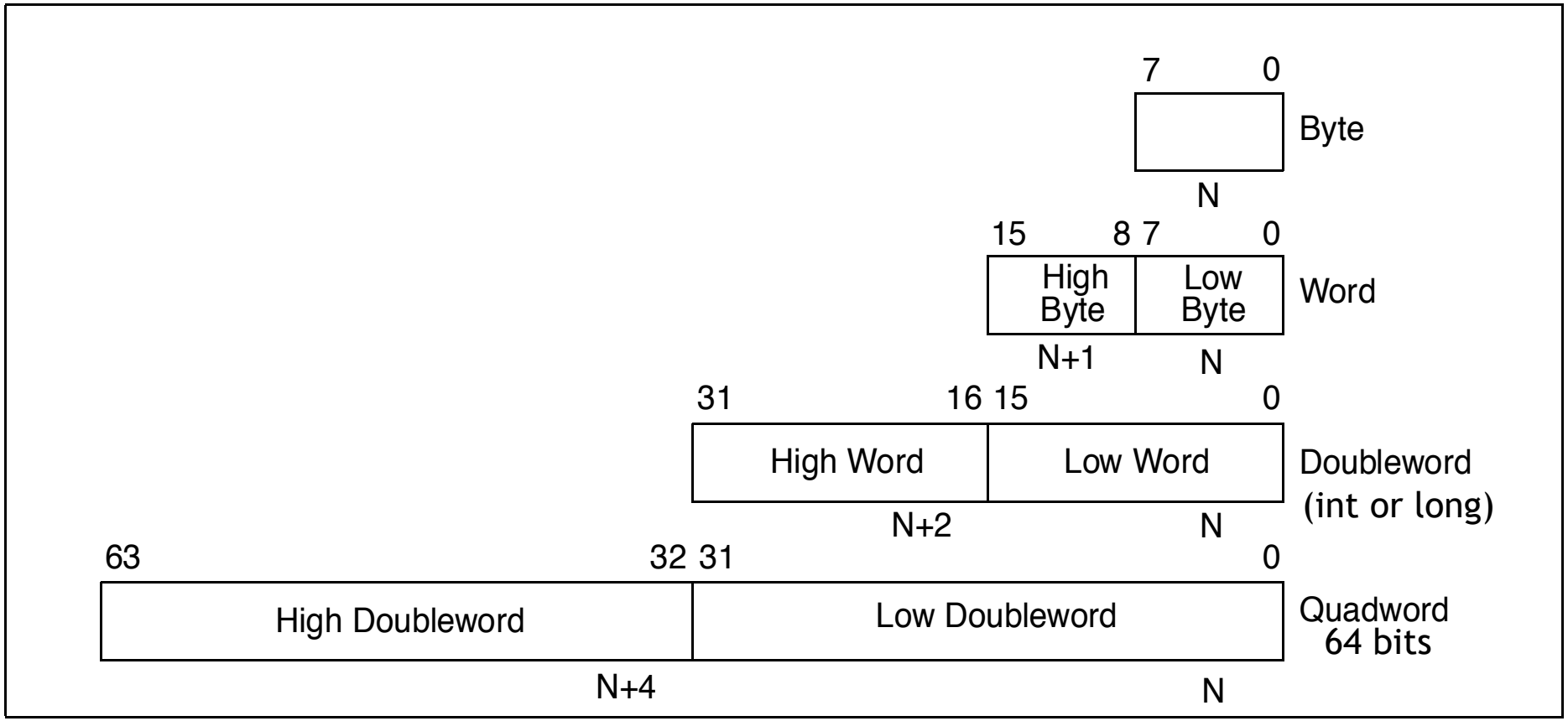
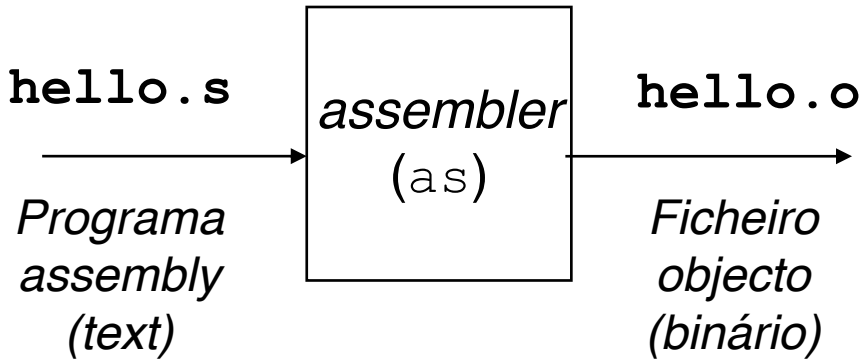


Figure 5-1. Fundamental Data Types

Programação em Assembly

- Ninguém programa diretamente em código máquina
- Programa-se em ***assembly***:
  - Texto, segundo regras próprias, onde se usa mnemónicas para as instruções, números em diversas bases, nomes simbólicos, etc.
- Um programa, o ***assembler***, traduz para código máquina:

```
as –o hello.o  hello.s
```



Assembly Unix para Intel

- Assembly AT&T
- Registos prefixados com % → **`%eax`**, **`%al`**, etc
- Valores imediatos prefixados com $ → **`$20`**, **`$0x1A`**, etc
- Instruções tipicas: ***mnemónica src, dst***
  - Dimensão dos dados da instrução na mnemónica:
    `mov`**`b`** (8bits), `mov`**`w`** (16bits), `mov`**`l`** (32bits)
  - Pode ser subentendido pelos operandos (%eax-32bits; %al-8bits)

```
movl $20, %eax
mov $20, %eax
```

- Existem outros "*assemblys*" para o Intel, como o próprio Assembly Intel
  - Todos geram o mesmo código máquina a partir de sintaxes diferentes

Assembly Unix para Intel

- Existem várias “instruções” que controlam o *assembler* e ajudam o programador mas que não se traduzem diretamente em instruções máquina:
  - Comentários: /* */ ou #
  - Etiquetas (labels) – para referenciar posições de memória
  - Preencher memória (variáveis) com valores: **.ascii**, **.byte**, **.int** ...
  - Definir símbolos para valores (semelhante ao *define* do C)
  - etc

Helloworld em Assembly Unix

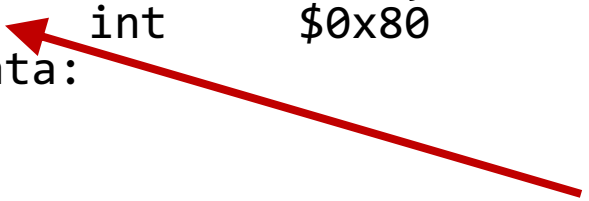
```
        EXIT = 1# usando simbolos para constantes
        WRITE = 4
        LINUX_SYSCALL = 0x80

.data           # secção de dados (variáveis)
msg:    .ascii    "Hello, world!\n"  # um vetor de caracteres
LEN = . - msg  # LEN representa o tamanho do vetor

.text           # secção de código
.global _start # exportar o simbolo _start (inicio do programa)
_start:   mov    $LEN,%edx
          mov    $msg,%ecx
          mov    $1,%ebx
          mov    $WRITE,%eax        # pedir write ao sistema
          int    $LINUX_SYSCALL     # chama o sistema

          mov    $0,%ebx
          mov    $EXIT,%eax         # pedir o exit ao sistema
          int    $LINUX_SYSCALL     # chama o sistema
```

Programação usando o Assembler

- O *assembler* verifica a validade da instrução *assembly*: se existe a instrução máquina tendo em conta os operandos indicados
  - **Exemplo**: `mov %eax, %cx` não existe!
- Resolve as etiquetas obtendo o endereço respetivo
- Também interpreta e faz as conversões de várias bases de numeração
  - Decimal, hexadecimal, binário, caracteres
  - Nomes simbólicos para constantes
- Codifica cada instrução no respectivo código máquina

Vendo o resultado do assembler

```
$ objdump  -D hello.o
hello.o:     file format elf32-i386
Disassembly of section .text:
00000000 <_start>:
0:     ba 0e 00 00 00          movl    $0xe,%edx
5:     b9 00 00 00 00          movl    $0x0,%ecx
a:     bb 01 00 00 00          movl    $0x1,%ebx
f:     b8 04 00 00 00          movl    $0x4,%eax
14:    cd 80                   int     $0x80
16:    bb 00 00 00 00          movl    $0x0,%ebx
1b:    b8 01 00 00 00          movl    $0x1,%eax
20:    cd 80                   int     $0x80
Disassembly of section .data:
00000000 <msg>:
0:     48
1:     65
2:     6c
3:     6c
4:     6f
 . . . etc
```



Representação Hexadecimal do código máquina

Um programa em Unix/Linux

- Um programa, para ser executável num SO tem de seguir algumas regras:
  - O ficheiro contendo o executável tem um formato específico (exemplo: ELF)
  - Contém código e dados em duas secções distintas e claramente identificadas
  - Indica o endereço onde começar a execução (exemplo: _start)
- O linker (ld) produz este tipo de executáveis
  - O código ligado pode ter diversas origens:
    - assembly ou diversas linguagens de programação;
    - compiladas em momentos diferentes (ex: de bibliotecas)

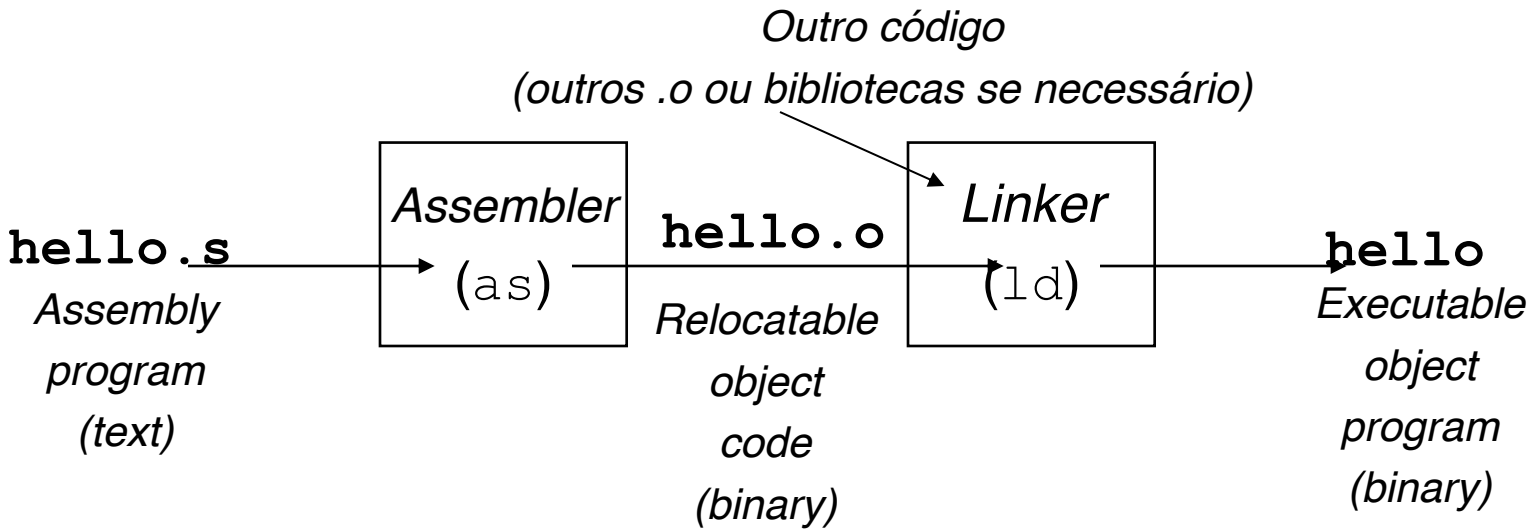
Produzindo o executável em Linux

- O *assember* (as no nosso caso) traduz para código máquina (.o):

```
as –o hello.o hello.s
```

- O *linker* (ld) produz o executável (formato ELF):

```
ld –o hello hello.o
```



MOV (movb, movw, movl)

- Instrução *assembly* de movimento de dados

  `mov  src, dst` *dst* ← *src*

- Corresponde a dezenas de instruções máquina:
  - Dependem do tamanho dos operandos:
    - 8 (b), 16 (w), 32 bits (l)
  - Dependem da localização dos operandos:
    - na instrução, em registo, na memória

Operandos imediatos

- O operando faz parte da própria instrução
  - trazido para o CPU (para o *Instruction Register*), quando este faz *fetch* da instrução
  - Indicado no *assembly* por **$**

Exemplo: **mov $4**, %eax EAX ← 4

  - Serve para indicar constantes na instrução
  - Pode reduzir o tamanho do programa e aumentar a velocidade da sua execução

%eax representa um operando em registo

Operandos imediatos

- Exemplos usando etiquetas ou nomes simbólicos para os valores:

```
Var:   .int 3   # define uma doubleword em memória com o valor 3
CONST = -5      # define o símbolo CONST para o valor -5

mov   $Var, %eax          EAX ← end Var
```

*o **endereço** da variável será colocado no registo EAX*

```
mov   $CONST, %eax        EAX ← -5
```

a representação de -5 será colocada em EAX

Exemplos de mov com operandos imediatos

```
$ objdump  -D hello.o
hello.o:     file format elf32-i386
Disassembly of section .text:
00000000 <_start>:
0:    ba 0e 00 00 00          movl    $0xe,%edx
5:    b9 00 00 00 00          movl    $0x0,%ecx
a:    bb 01 00 00 00          movl    $0x1,%ebx
f:    b8 04 00 00 00          movl    $0x4,%eax
14:   cd 80                   int     $0x80
16:   bb 00 00 00 00          movl    $0x0,%ebx
1b:   b8 01 00 00 00          movl    $0x1,%eax
20:   cd 80                   int     $0x80
 . . .
```

Operandos em registos

- O operando está contido num dos registos do CPU
  - Pressupõe que uma instrução anterior deixou o registador com o valor desejado
  - Indicado no *assembly* por **%**

Exemplo: **mov %ax, %dx** DX ← AX

  - Acesso rápido ao operando
  - Servem como variáveis temporárias (ou de cópias das variáveis em memória)
  - **Limitado pelo número de registos do CPU**

Endereçamento direto da memória

- O endereço do operando está contido na instrução
  - O operando está na memória RAM (p.e. variável do programa)
  - Quando **não tem** $ ou indicado por **( )**

Exemplo: **mov (100)**, %eax EAX ← Memória[100]

- É necessário conhecer essa posição quando se escreve o programa?
  - Pode ser difícil saber e pode obrigar a que o programa seja carregado numa zona específica da memória
  - **O *assembler* ajuda com as etiquetas (labels) que servem como o nome das variáveis**
  - **Pode ser alterado na ligação ou carregamento**

Endereçamento direto da memória

- Exemplo de endereçamento **direto** da memória usando etiqueta em vez do valor do endereço:

```
Var:   .int 3  # define uma doubleword em memória com o valor 3

…
mov   Var, %eax          EAX ← memoria[Var]
```

Equivale a:

```
mov   (Var), %eax
```

ou

```
movl (Var), %eax
```

Exemplos

- Exemplos usando etiquetas e nomes simbólicos:

```
Var:   .int 3   # define uma doubleword em memória com o valor 3
CONST = -5      # define o símbolo CONST para o valor -5

mov   $Var, %eax        EAX ← end Var
      o endereço da variável será colocado no registo EAX
mov   $CONST, %eax              EAX ← -5
      a representação de -5 será colocada em EAX
mov   Var, %eax         EAX ← memoria[Var]
      o valor no endereço de memória Var será colocado no registo EAX
```

Endereçamento indirecto por registo

- Um registo contém o endereço de memória onde está o operando
  - Pressupõe que uma instrução anterior deixou no registo o endereço desejado
  - Indicado no *assembly* por um registo entre **( )**

Exemplo: **mov (%ebx)**, %ax AX ← Memoria[EBX]

  - O registo referencia a memória, como um *pointer*
    - Facilita a programação (p.e. percorrer vetores)

Transferências de dados (MOV)

Table 6-1. Move Instruction Operations

Type of Data Movement	Source → Destination
From memory to a register *(load)*	Memory location → General-purpose register Memory location → Segment register
From a register to memory *(store)*	General-purpose register → Memory location Segment register → Memory location
Between registers	General-purpose register → General-purpose register General-purpose register → Segment register Segment register → General-purpose register General-purpose register → Control register Control register → General-purpose register General-purpose register → Debug register Debug register → General-purpose register
Immediate data to a register	Immediate → General-purpose register
Immediate data to memory	Immediate → Memory location

Não pode copiar de memória para memória!

Resumo - Combinações de operandos

	Origem	, Destino	*Exemplo*	*Correspondência C (aprox)*
mov, **movb**, **movw**, **movl**	*Imm*	*Reg*	`mov $0x4,%eax`	`temp = 0x4;`
		MemDir	`movl $-147,var`	`var = -147;`
		MemInd	`movl $45, (%edx)`	`*p = 45;`
	Reg	*Reg*	`mov %eax,%edx`	`temp2 = temp1;`
		MemDir	`mov %eax,var`	`var = temp;`
		MemInd	`mov %eax,(%edx)`	`*p = temp;`
	MemDir	*Reg*	`mov var,%edx`	`temp = var;`
	MemInd	*Reg*	`mov (%eax),%edx`	`temp = *p;`

É semelhante para as operações aritméticas e lógicas

Codificação de cada instrução máquina

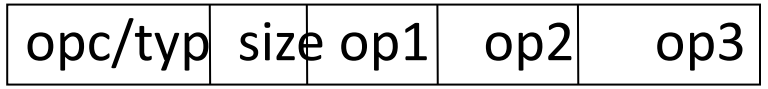
- O formato depende do tipo de instrução
  - O código da operação ou *opcode* (sempre!)
  - A indicação do tamanho dos dados
  - A especificação dos operandos: tipo e localização ou valor

- Exemplo: instrução com 3 operandos:

  op3 ← ***operação****(op1, op2)*

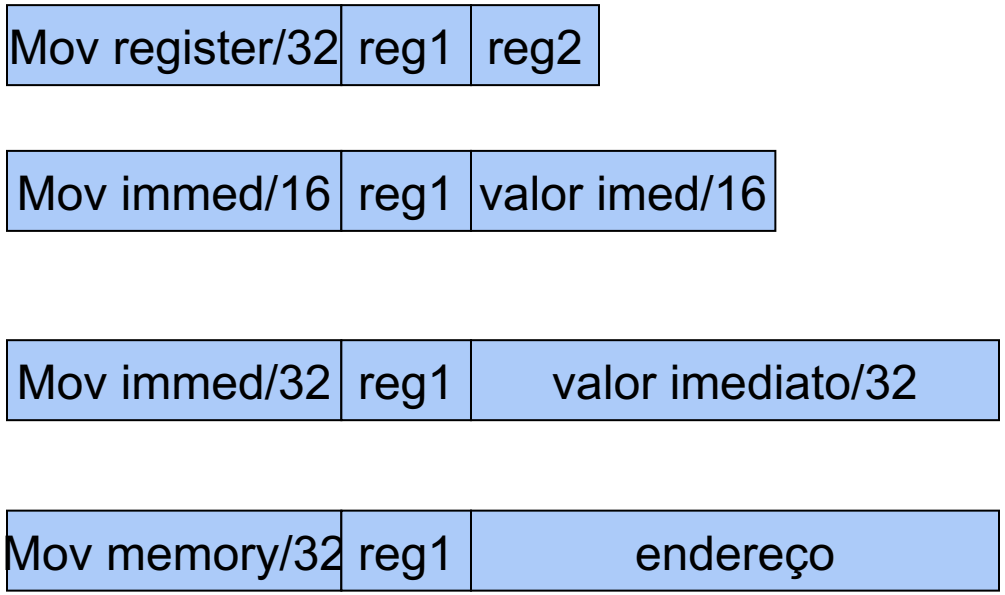
  - Exige codificar em bits: a operação, os tipos de operandos (registos, endereço ou valor imediato) e os valores dos operandos (qual o registo, qual o valor ou qual o endereço de memória).

  - Possível codificação:



Codificação dos operandos nas instruções

- O código da operação indica a dimensão dos dados, o tipo de operandos e seus endereços
- Alguns exemplos de variantes de MOV no Intel



`mov %eax, %edx`
reg1 → reg2

`mov $4, %ax`
valor → reg1
(2 bytes)

`mov $4, %eax`
valor → reg1

`mov (100), %eax`
Mem[endereço] → reg1
(4 bytes)

Vendo o resultado do assembler

```
$ objdump  -D hello.o
hello.o:     file format elf32-i386
Disassembly of section .text:
00000000 < start>:
0:      ba 0e 00 00 00          movl   $0xe,%edx
5:      b9 00 00 00 00          movl   $0x0,%ecx
a:      bb 01 00 00 00          movl   $0x1,%ebx
f:      b8 04 00 00 00          movl   $0x4,%eax
14:     cd 80                   int    $0x80
16:     bb 00 00 00 00          movl   $0x0,%ebx
1b:     b8 01 00 00 00          movl   $0x1,%eax
20:     cd 80                   int    $0x80
Disassembly of section .data:
00000000 <msg>:
0:      48
1:      65
2:      6c
3:      6c
4:      6f
 . . . etc
```

Diferentes arquiteturas podem ter diferentes operandos

- Usar operandos de que tamanho? (8, 16, 32 ?)
- Onde podem estar os operandos?
  - Registos do CPU (indicar número do registo – poucos bits)
  - Na Instrução
  - Na memória (é necessário dar o endereço)
- Qual o número de operandos?
  - op1 ← operação(op1, op2)
  - op ← operação(op)
  - Instruções sem operandos ou que assumem locais pré-definidos para operandos?
- Nos Intel existem de todos os tipos
  - As instruções não são todas do mesmo tamanho
  - Fetch, decode e obter operandos é complicado

Operandos e dimensão da instrução

- Como ter instruções de menor tamanho:
  - Instruções com menos operandos
  - Instruções com operandos em registos do CPU
    - usa menos bits para endereçar os registos
  - Instruções com indireção para os operandos
    - na instrução indica-se o endereço ou o registo onde está o endereço do operando
  - Instruções com operandos implícitos
    - existem registos/endereços pré-definidos para o operando

Intel: Operações aritméticas e lógicas

- Operações:
  - aritméticas: add, sub, inc, dec, cmp, ...
  - lógicas: and, or, xor, not, ...
- Realizadas pela ALU
- Cada operação altera o estado de algumas *flags*

EFLAGS:

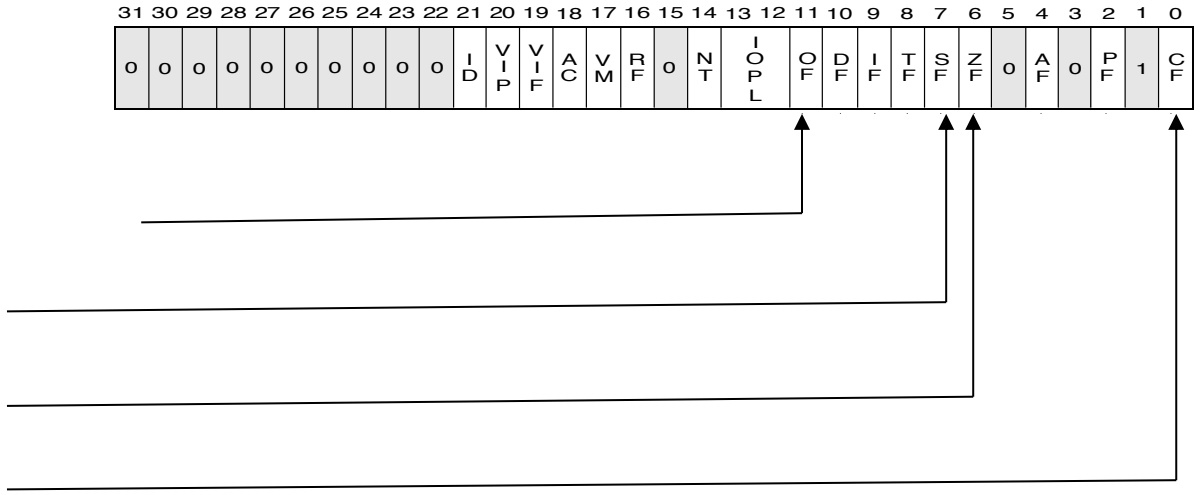
Exemplos:

OF – *Overflow*

SF – *Sign*

ZF – *Zero*

CF – *Carry*



Operações aritmeticas e lógicas (exemplos)

- Exemplos em *assembly*:
  - Aritméticas:

```
add $5, %eax
sub %bx, %cx
inc %al
```

  Lógicas:

```
and %ecx, %eax
or  $0x2F, %ah
not %ax
```

  - Estas correspondem a instruções de tamanhos diferentes, com operadores de tipos diferentes
  - Nestes exemplos a dimensão dos dados pode ser inferida pelo *assembler* a partir dos registos usados
- Estas operações alteram as *flags!* (*Overflow*, *Carry, etc…*)

ADD – Adição de números

add op1, op2

op2←op2+op1

- soma os bits de ambos os operandos
- o resultado é guardado na segunda referência (que não pode ser imediata)
- altera as *flags* de acordo com o resultado
- exemplos:

```
add $5, %eax          -- end. Imediato/reg e 32bits
add %ax, %bx          -- end. Registos e 16bits
add (100), %al        -- end. Direto/reg e 8bits
```

Exemplos ADD e 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

Carry Flag -- CF

- CF a 1: se há transporte do bit mais significativo
- Interpretação a fazer no programa:
  - uma op. aritmética sobre números sem sinal deu um resultado fora do domínio de valores válidos

```
mov $0xFF, %al
add $4, %al
```

```
  1111 1111     FF (255)
  0000 0100     04   ( 4)
 -------------  -----
1 0000 0011   1 03  (259)
```

CF = 1

AL = 0000 0011 (em 8bits, o limite é 255)

Overflow Flag -- OF

- OF a 1: se uma op. aritmética sobre números com sinal, dá resultado fora do domínio de valores válidos

```
mov $4, %al
add $127, %al
```

```
0000 0100    (4)
0111 1111  (127)
-------------
1000  0011  (-125?)
```

→ a soma de dois números positivos não pode dar negativo!

OF = 1 (4+127 > limite de 127)

Nota: CF = 0 → *em números sem sinal: 4+127=131 < 255*

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

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 $2, %ax
sub $2, %ax      → ZF = 1

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

Exemplos ADD

- Exemplos em que o resultado é inválido se em complemento para 2.
  - A OF e CF indicam que o resultado está errado, dependendo da representação a considerar

```
   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
```

SUB – Subtração de números

sub op1, op2

op2←op2-op1

- Subtrai op1 de op2

  o resultado é guardado no segundo operando

  altera as *flags* de acordo com o resultado

- **Exemplos**:

  sub $5, %eax -- Imediato/reg a 32bits

  sub %ax, %bx -- Registos a 16bits

  sub (100), %al -- End. Direto a 8bits

Instruções de controlo

- Saltos (jumps): alteram o valor do EIP - *Instruction Pointer* (ou *Program Counter*)
  - Incondicionais
    - Exemplo: **`jmp` *endereço*** EIP ← endereço
    - a próxima instrução executada é a que se encontra no endereço indicado
  - Condicionais
    - Só alteram EIP se a uma dada condição (**flags**) se verifica.
    - **Exemplos**:

      `jz` *endereço* - só salta se a flag de Zero estiver a 1

      `jnz` *endereço* – só salta se a flag de Zero estiver a 0

Saltos e 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`'

Comparação de números

- **Fazer X-Y, permite comparar os números**
- se *números sem sinal*:
  - CF indica transbordo (*borrow*) na interpretação sem sinal
  - **Se** há *carry* (*borrow*) **então**: X < Y
    Senão: X>=Y
  - **Se** resultado for zero (ZF=1) **então** X=Y

- se *números com sinal*:
  - A relação entre X e Y é mais complicada…

CMP - Compare

- **cmp x, y** faz y-x mas não guarda o resultado, apenas altera as flags
  - põe CF = 1 se houver bit de *borrow*, (significa y<x para números sem sinal)
  - põe ZF = 1 se os operandos forem iguais
  - Põe OF=1 se houver *overflow*,

- Conclusão:
  - **cmp** compara os dois operandos: usa-se em vez do **sub** quando se quer alterar apenas as *flags*

Comparando números com sinal

Sem sinal	*Com sinal*
$1111\ 1111 = 255_{10}$	$1111\ 1111 = -1_{10}$

comparando com zero. **Qual é maior?**

$255 > 0$	$-1 < 0$

***Flag* a testar:**

Sem sinal	*Com sinal*
CF	necessitamos de testar *flags* diferentes!

Exemplos de comparações (c/sinal)

- Usando **cmp** X,Y/**sub** X,Y , quando **Y>X**:

Y	X	Y-X	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**	1-(-7) = -8???
1111(-1)	1001(-7)	0110(6)	0	0	

Exemplos de comparações (c/sinal)

- Usando **cmp** X,Y/**sub** X,Y , quando **Y<X**:

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

Conclusão

- Comparação usando **cmp** x,y /**sub** x,y entre números com sinal:

  Quando ***Y-X>0 → SF=0 e OF = 0 ou SF=1 e OF=1***

  então ***Y>X → (SF=0 e OF=0) ou (SF=1 e OF=1) → SF = OF***

  Quando ***Y-X<0 → SF=1 e OF = 0 ou SF=0 e OF=1***

  então ***Y<X → (SF=1 e OF=0) ou (SF=0 e OF=1) → SF <> OF***

- Concluindo:

  Y > X → SF = OF e ZF = 0

  Y >= X → SF = OF ou ZF = 1

  Y < X → SF <> OF e ZF = 0

  Y <= X → SF <> OF ou ZF = 1

Saltos condicionais (com 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

Exemplo: ciclo **for**

```
            mov $0, %ecx
inicio_for:
            cmp n, %ecx
            jz fim_for
               # codigo_do_ciclo
            inc %ecx
            jmp inicio_for
fim_for:    . . .
```

for (i=*0*; i!=n; i++)
codigo_do_ciclo;

Saltos condicionais (inteiros s/sinal)

- As relações entre inteiros sem sinal, são dadas, após

  cmp** X, Y* (ou ***sub) pelas *flags* ***CF** e **ZF***

- Saltos condicionados por estas *flags*:

JB – Jump if Below (=**JC**)	Y<X
JE – Jump if Equal (=**JZ**)	Y=X
JAE – Jump if Above or Equal (=**JNC**)	Y>=X

E combinações:

JA – Jump if Above (***CF=0*** e ***ZF=0***)	Y>X
JBE – Jump if Below or Equal (***CF=1*** ou ***ZF=1***)	Y<=X

Saltos condicionais (aliases)

- Mais mnemónicas para as mesmas instruções:

  JNBE = JA

  JNB = JAE = JNC

  JNA = JBE

  JNAE = JB = JC

  JNE = JNZ

Saltos condicionais (inteiros c/sinal)

- Saltos condicionais para as relações entre inteiros com sinal:

  após **sub** X, Y ou **cmp** X,Y

salta se			***flags***
Y>X	greater than	JG/JNLE	SF = OF and ZF=0
Y>=X	greater or equal	JGE/JNL	SF = OF
Y<X	less than	JL/JNGE	SF <> OF and ZF=0
Y<=X	less or equal	JLE/JNG	ZF=1 or SF <> OF

Saltos (resumo)

- Instruções de salto de acordo com os valores de “flags”

inst.	Condition *(C expr.)*	Description
`jmp`		Unconditional
`je/jz`	`ZF`	Equal / Zero
`jne/jnz`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (signed)
`jge`	`~(SF^OF)`	Greater or Equal (signed)
`jl`	`(SF^OF)`	Less (signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jae`	`~CF`	Ãbove or Equal(unsigned)
`jb`	`CF`	Below (unsigned)
`jbe`	`CF|ZF`	Below or Equal (unsigned)

Cmp & j*

- Comparação e saltos condicionais são a base para:
  - If
  - For
  - While
  - Etc…

Exemplo: 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 que altere as flags
# (exemplo cmp $5, %eax)
jxx bloco_else
        #  codigo_do_then
        jmp fim_if
bloco_else:
        # codigo_do_else
fim_if:
```

Exemplo: for

- ciclo com base num contador

```
for ( i=0; i<n; i++ )
  codigo_do_ciclo;
```

```
#exemplo:
            mov $0, %ecx
inicio_for:
            cmp $n, %ecx
            je fim_for
            #  codigo_do_ciclo
            inc %ecx
            jmp inicio_for
fim_for:
```

Exemplo: while

- ciclo com teste "à cabeça"

while (*condicao*)
  codigo_do_ciclo;

```
inicio_while:
# código que altere as flags
# (exemplo cmp $5, %eax)
        jxx fim_while
        #  codigo_do_ciclo
        jmp inicio_while
fim_while:
```

Exemplo: do-while

- ciclo com teste "no fim"

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

```
inicio_do:
   # codigo_do_ciclo

   # código que altere as flags
   # (exemplo cmp $5, %eax)
   jxx inicio_do
```

Exemplo: incr. elementos de um vetor

```
.data
vet: .int 0, 3, 5, 10, 2

.text
        . . .
        mov $5, %eax
        mov $vet, %ebx
ciclo: incl (%ebx)   #incr. os ints de vet
        add $4, %ebx
        dec %eax
        jnz ciclo
        . . .
```

Exemplo: for (2)

- ciclo com base num contador (decrementado)

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

```
#exemplo:
            mov $n, %ecx
            cmp $0, %ecx
inicio_for:
            jz fim_for
            #  codigo_do_ciclo
            dec %ecx
            jmp inicio_for
fim_for:
```

Exemplo: for (3)

- Se ciclo é para executa pelo menos uma vez:

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

```
#exemplo:
                mov $n, %ecx
inicio_for:
                #  codigo_do_ciclo
                dec %ecx
                jnz inicio_for
fim_for:
```

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 *(16bits)*

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

```
        mov $n, %cx
inicio_for:
        #  codigo_do_ciclo
        loop inicio_for
```

Nota: para n > 0!

Instruções lógicas

- Tratam as *palavras* como vetores de bits
  - Onde cada bit: 1=true, 0=false
  - as *flags* refletem o resultado

- Exemplos de instruções:

not *X*	**and** *X,Y*	**or** *X,Y*	**xor** *X,Y*
X ← not X	*Y ← X and Y*	*Y ← X or Y*	*Y ← X xor Y*

Em C ou Java:

X=~X	*Y=X&Y*	*Y=X|Y*	*Y=X^Y*

TEST

- test X, Y
  - Igual ao AND mas sem alterar o 2º operando
  - Exemplo:

    test $0b00000100, %dl
  - põe ZF a 1 se o bit 2 de DL está a 0, mas não altera DL

  - Equivale a testar se todos os bits indicados no primeiro operando estão a zero no Segundo

Manipular bits

- Exemplos:
  - **not** %al -- inverte cada bit de AL
  - **and** $0x7FFF, %ax -- põe a 0 o bit 15 de ax
  - **or** $0x8000, %ax -- põe a 1 o bit 15 de ax
  - **xor** $0x8000, %ax -- inverte o bit 15 de ax

  - **test** $0b00010101, %ah -- testa bits 0, 2, 4 de AH: põe ZF a 1 se esses bits em AH forem 0; ZF fica 0 se algum desses bits em AH for 1

Deslocamentos (shifts)

- **shl** n, X (C ou Java: X=X<<n)
  - Desloca os bits em X para a esquerda (é como multiplicar por 2)

    CF <----| <----X------ |<---- 0

Exemplo:

mov 0b00110001, %al

shl $1, %al ou **shl** %al

AL fica com 0b01100010

Deslocamentos (shifts)

- **shr** n, X (C se X unsigned int: X=X**>>**n. Java: X=X**>>>**n.)
  - Desloca os bits em X para a direita (é como dividir por 2)

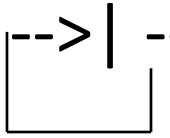
    0 ---->| -----X-----> |----> CF

Exemplo:

```
mov 0b00110001, %al
shr $1, %al      ou      shr %al

AL fica com 0b00011000
```

Deslocamentos (shifts)

- **sar** n, X (se X é int, C e Java: X=X**>>**n)
  - Desloca os bits em X para a direita mas mantém o bit mais significativo (é como dividir por 2 mantendo sinal)



Exemplo:

mov 0b**1**0110001, %al

sar $1, %al ou sar %al

AL fica com 0b**11**011000