Programação em Sistemas Computacionais PSC

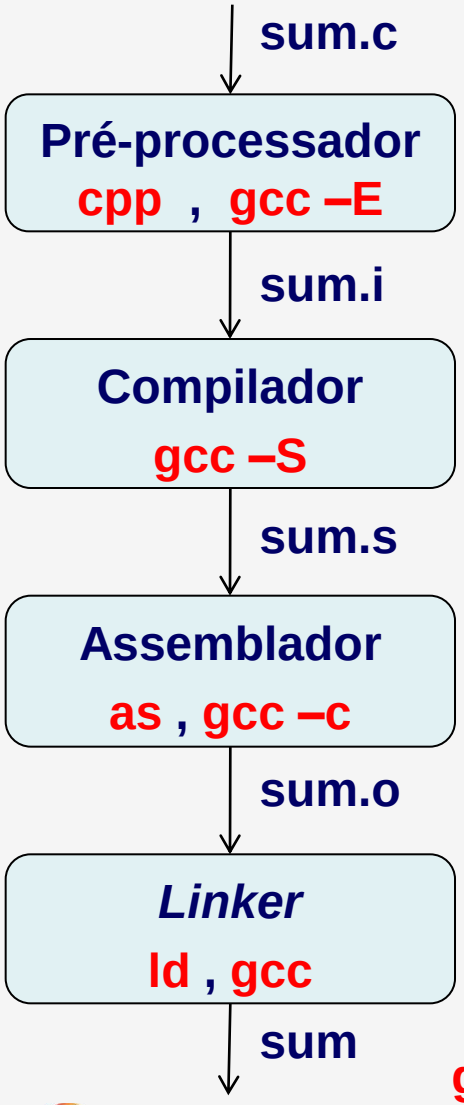
Assembly

Arquitectura IA32

Centro de Cálculo
Instituto Superior de Engenharia de Lisboa

Pedro Pereira palex@cc.isel.ipl.pt

Fases da compilação



- O pré-processador interpreta as linhas iniciadas com '#' no ficheiro fonte (texto) e gera outro ficheiro fonte
- O compilador gera o ficheiro em assembly (texto) com as instruções a executar
- O assemblador gera o ficheiro objecto (binário) com código de máquina das instruções
- O Linker gera o ficheiro executável (binário) depois de juntar com o código dos outros módulos

```
gcc –O2 –masm=intel –save-temps sum.c –o sum
```

Pré-processador

cpp sum.c > sum.i

gcc –E sum.c > sum.i

sum.c

```
#include <stdio.h>
#define VALUE 10

int accum;

int sum(int x, int y) {
  int t;
  t = x+y;
  accum += t;
  return t;
}

int main() {
  sum(VALUE,VALUE);
  printf("accum=%d\n",accum);
}
```

sum.i

```
…
int printf(const char *s,…);
…

int accum;

int sum(int x, int y) {
  int t;
  t = x+y;
  accum += t;
  return t;
}

int main() {
  sum(10,10);
  printf("accum=%d\n",accum);
}
```

Compilador

gcc –O2 –S –masm=intel sum.i

sum.i

```
…
int printf(const char *s,…);
…

int accum;

int sum(int x, int y) {
  int t;
  t = x+y;
  accum += t;
  return t;
}

int main() {
  sum(10,10);
  printf("accum=%d\n",accum);
}
```

sum.s

```
  .intel_syntax noprefix
  .comm accum,4
  …
  .text
sum:
  push  ebp
  mov   ebp, esp
  mov   eax, [ebp+12]
  add   eax, [ebp+8]
  add   accum, eax
  pop   ebp
  ret
  .globl main
main:
  …
  ret
```

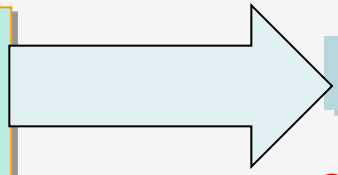
Assemblador

as sum.s –o sum.o

gcc –c sum.s –o sum.o

sum.s

```
  .intel_syntax noprefix
   .comm accum,4
   …
   .text
sum:
   push  ebp
   mov   ebp, esp
   mov   eax, [ebp+12]
   add   eax, [ebp+8]
   add   accum, eax
   pop   ebp
   ret
.globl main
main:
   …
   ret
```



sum.o

objdump -d -M intel sum.o > sum.od

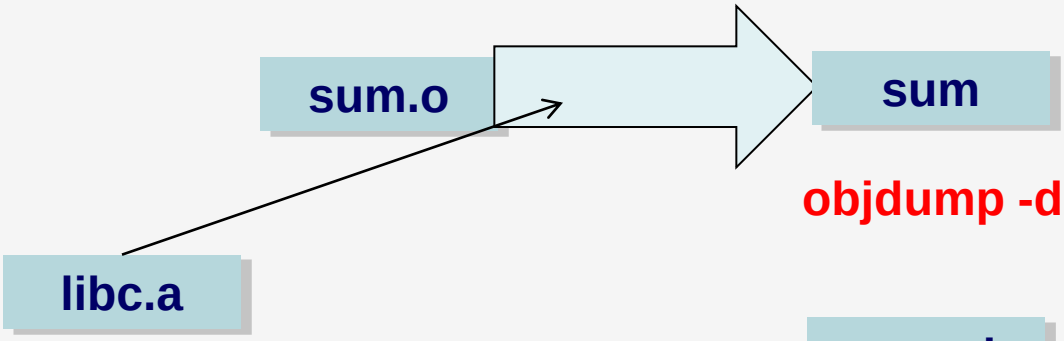
sum.od

```
…
Disassembly of section .text:
00000000 <sum>:
 0: 55                      push ebp
 1: 89 e5                   mov  ebp,esp
 3: 8b 45 0c                mov  eax,[ebp+0xc]
 6: 03 45 08                add  eax,[ebp+0x8]
 9: 01 05 00 00 00 00       add  ds:0x0,eax
 f: 5d                      pop  ebp
10: c3                      ret
…
```

Linker

ld sum.o –o sum –lc …

gcc sum.o –o sum



objdump -d -M intel sum > sum.d

sum.d

```
…
080483d0 <sum>:
 80483d0: 55                    push ebp
 80483d1: 89 e5                 mov  ebp,esp
 80483d3: 8b 45 0c              mov  eax, [ebp+0xc]
 80483d6: 03 45 08              add  eax, [ebp+0x8]
 80483d9: 01 05 1c a0 04 08     add  ds:0x804a01c,eax
 80483df: 5d                    pop  ebp
 80483e0: c3                    ret
…
```

- **Arquitectura Intel a 32 bits**
  - Registos
    - eax, esi…
  - Instruções
    - mov, inc, cmp, jne, ret, xor…
- **Por quê saber IA32?**
  - Debug (demo)
  - Optimizar código (mesmo que escrito em C)
  - Melhor compreensão do C
    - Ponteiros;
    - Passagem de parâmetros;
    - etc…

História dos processadores Intel IA32 (família x86)

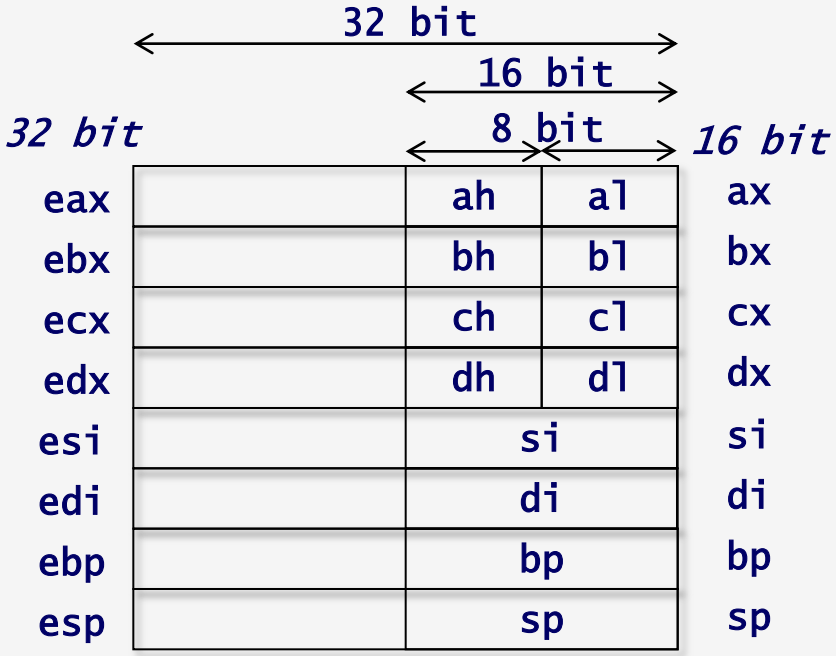
- **1978: 8086 (8MHz) 24KT**
  - 16 bits
  - Address 2^{20} bytes (1MB)
- **1982: 80286 (12.5MHz) 134KT**
  - Address 2^{24} bytes (16MB)
  - Segmentação
- **1985: i386 (20MHz) 275KT**
  - 32 bits
  - Address 2^{32} bytes (4GB)
  - Paginação
- **1989: i486 (25MHz) 1.9MT**
  - Organização em pipeline
  - L1 cache
- **1992: Pentium (60MHz) 3.1MT**
  - Organização super-escalar
- **1995: PentiumPro (200MHz) 6.5MT**
  - L2 cache
  - Espaço endereçamento (64GB)
- **1997: Pentium II (266MHz) 7MT**
  - L3 cache
- **1999: Pentium III (500MHz) 24MT**
- **2000: Pentium 4 (1.5GHz) 42MT**
  - 8 byte Integer
- **2006: Core 2 (3.33GHz) 400MT**
  - 64 bits (IA64)
  - Dual core
- **2008: Core i7 (3.33GHz) 800MT**
  - Quad core

IA64

Registos

- **8 registos de 32 *bits* de uso (quase) genérico**
- **Alguns com acesso a 8 e 16 *bits***
  - esp stack pointer
  - ebp base pointer
  - esi, edi (em strings)
  - ecx (em contagens)
- **Outros registos**
  - eip Instruction Pointer
  - eflags (para flags)

    cf-carry; pf-parity; **zf**-zero; **sf**-sign; **of**-overflow; df-direction
  - cs, ds, ss, es, fs, gs selectores de segmento (16 bits)



8 bits - byte
16 bits - word
32 bits - double word

Operandos das instruções

- **Instruções com 0, 1 ou 2 operandos**
  - `hlt`
  - `inc ebx`
  - `mov ax, cx` (o 1º operando é o destino)
- **Tipos de operandos**
  - Imediato (valor constante): 577 0x1F
  - Registo (valor em registo): eax bl ch si
  - Memória (valor em memória): [esi] [edx+9] [eax+edx]

Formas de endereçamento

- **Forma genérica**
  - endereço = registo + scale*registo + imediato
    - registo – qualquer registo a 32 *bits*
    - scale – 1, 2, 4 ou 8
    - imediato – definido a 8, 16 ou 32 bits
- **Não necessita de ter todas as componentes**
  - Endereço = imediato (directo)
  - Endereço = registo (indirecto)
  - Endereço = registo + imediato (baseado)
  - Endereço = registo + registo (indexado)
  - Endereço = registo + scale*registo (indexado escalado)
- **Exemplo**

```
mov eax, [esi+4*edi+15]
```

Exemplos de operandos

- **Dados os seguintes valores nos registos e em memória**

eax	0x100
ecx	0x1
edx	0x3

0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

- **Qual o valor que fica em bl com cada uma das instruções:**

`mov bl, ah`	bl 0x01	`bl=Mb[ah];`
`mov bl, [0x104]`	bl 0xAB	`bl=Mb[0x104];`
`mov bl, 104`	bl 104	`bl=104;`
`mov bl, [eax]`	bl 255	`bl=Mb[eax];`
`mov bl, [eax+4]`	bl 0xAB	`bl=Mb[eax+4];`
`mov bl, [eax+edx+9]`	bl 0x11	`bl=Mb[eax+edx+9];`
`mov bl, [ecx*4+0xFC]`	bl 0xFF	`bl=Mb[ecx*4+0xFC];`
`mov bl, [eax+edx*4]`	bl 0x11	`bl=Mb[eax+edx*4];`

Instruções

- **Com 2 operandos (o primeiro especifica o destino)**
  - Registo, Registo
  - Registo, Imediato
  - Registo, Memória
  - Memória, Registo
  - Memória, Imediato
  - Memória, Memória

```
mov   eax, ecx
add   ebx, 10
and   ecx, [ebx]
mov   [ebx], eax
sub   [esi], 3
mov   [ebx],[eax]
```

- **Instruções para:**
  - Transferência de dados: mov, push, pop, pusha, popa, …
  - Aritméticas: add, sub, mul, div, inc, neg, …
  - Lógicas: cmp, and, or, …
  - Deslocamento e rotação de bits: shr, shl, ror, rol, rcr, rcl, …
  - Transferência de controle: jmp, call, ret, iret, int, j**, …
  - Operações sobre strings: movs, cmps, scas, lods, stos, …
  - Controle das flags: stc, clc, std, cld, pushf, popf, …
  - ...

Transferência de dados

- **mov D, F**

  Move o valor de F para D.

  Faz LOAD ou STORE.

  Normalmente, a dimensão dos operandos é inferida pelo nome do registo envolvido.

```
mov   ebp, esp          ebp=esp;
mov   eax, [ebp+2]      eax=Mw[ebp+2];
mov   [ebp+4], al       Mb[ebp+4]=al;
```

- **push F**

  Coloca o valor de F no topo do stack (registo esp)

```
push  ebp               esp-=4;
                        Mw[esp]=ebp;
```

```
sub esp, 4
mov [esp], ebp
```

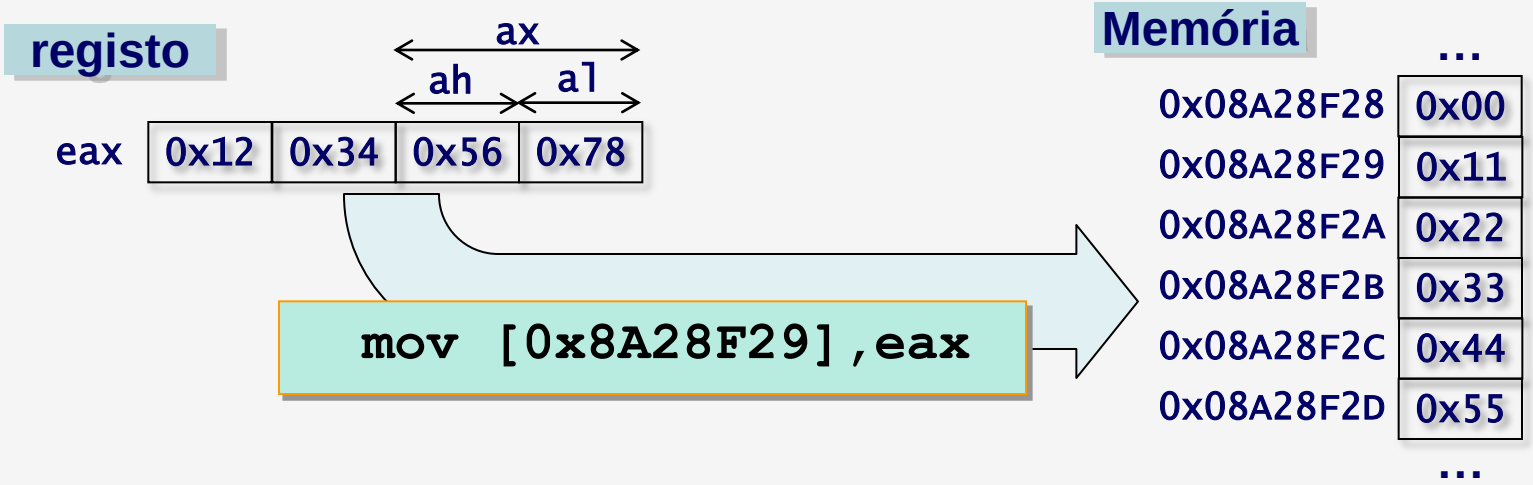
- **pop D**

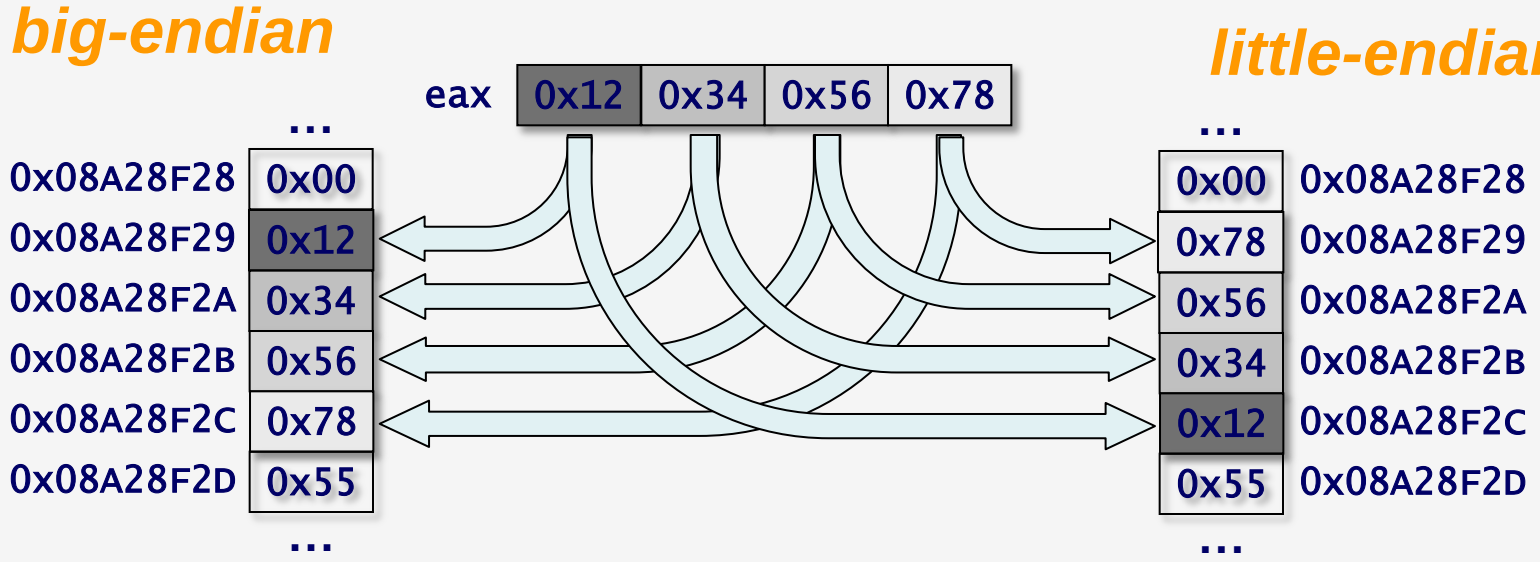
  Repõe o valor que está no topo do stack em D

```
pop   eax               eax=Mw[esp];
                        esp+=4;
```

```
mov eax, [esp]
add esp, 4
```

big-endian e little-endian





big-endian e little-endian

bl_endian.c

```
#include <stdio.h>

int main() {
  int x = 0x12345678;
  char *p = (char*)&x;
  int i;
  for(i=sizeof(x); i ; --i,++p )
    printf("%X\n",*p);
  return 0;
}
```

big-endian 32 bits

```
12
34
56
78
```

little-endian 32 bits

```
78
56
34
12
```

Exemplo (mov)

exchange.c

```
…
#include <stdio.h>

int exchange(int *xp, int y) {
  int x = *xp;
  *xp = y;
  return x;
}

int main() {
  int a=4;
  int b=3;
  b = exchange(&a,b);
  printf("a=%d, b=%d\n",a,b);
}
```

Sabendo que:

- **Os valores do primeiro e do segundo parâmetros ficam em ebp+8 e ebp+12**
- **O valor a retornar fica em eax**

```
…
mov   edx, [ebp+8]
mov   ecx, [ebp+12]
mov   eax, [edx]
mov   [edx], ecx
…
```

```
/* eax ⇔ x */
edx <- xp
ecx <- y
x = *xp;
*xp = y;
```

Operações aritméticas e lógicas

- **Operações aritméticas**
  - inc D dec D
  - neg D
  - add D, S sub D, S imul D, S

  actualizam: cf, zf, sf, of

  inc e dec não alteram: cf

```
dec cx
neg ecx
add cx,[eax]
```

```
D++   D--
D=-D
D+=S  D-=S  D*=S
```

- **Load effective address**
  - lea D, S

  não altera flags

```
lea edx,[eax+*8+ecx+7]
```

```
D=&S
```

- **Operações lógicas**
  - not D
  - xor D, S or D, S and D,S

  actualizam: zf, sf reset: cf, of

```
not cx
xor ax,ax
```

```
D=~D
D^=S  D|=S  D&=S
```

- **Shift**
  - sal D, k sar D, k
  - shl D, k shr D, k

  bit out: cf reset: of

```
sal cx,3
shr eax,cl
```

```
D<<=k  D>>=k   (signed)
D<<=k  D>>=k (unsigned)
```

Exemplos de operações aritméticas

- **Dados os seguintes valores nos registos e em memória**

eax	0x100
ecx	0x1
edx	0x3

0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

- **Qual o local alterado e qual o valor que lá fica?**

`add [eax], ecx`	0x100	0x200	`Mw[eax]+=ecx;`
`sub [eax+4], edx`	0x104	0xA8	`Mw[eax+4]-=edx;`
`imul [eax+edx*4],0x10`	0x10C	0x110	`Mw[eax+edx*4]*=16;`
`inc [eax+8]`	0x108	0x14	`Mw[eax+8]++;`
`dec ecx`	ecx	0	`ecx--;`
`sub eax,edx`	eax	0xFD	`eax-=edx;`
`shl [eax+ecx*8],4`	0x108	0x130	`Mw[eax+ecx*8]<<=4;`

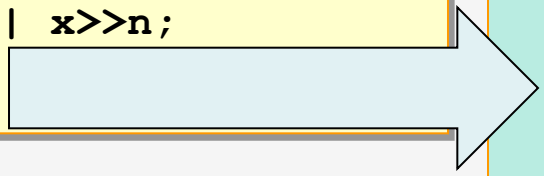
Exemplo (operações lógicas)

Sabendo que:

- **Os valores do primeiro e do segundo parâmetros ficam em ebp+8 e ebp+12**
- **O valor a retornar fica em eax**

shift.c

```
…
int shift_l2_rn(int x, int n) {
  return x<<2 | x>>n;
}
```



```
…
mov   edx, [ebp+8]
mov   ecx, [ebp+12]
mov   eax, edx
sar   eax, cl
sal   edx, 2
or    eax, edx
…
```

```
edx <- x
ecx <- n

eax = x>>n;
edx = x<<2;
return eax|edx;
```

Operações aritméticas especiais

- **Multiplicação**
  - imul S
  - mul S

```
imul ecx
mul 1024
```

```
(edx:eax)=S*eax   (signed)
...               (unsigned)
```

- **Divisão**
  - idiv S
  - div S

```
idiv ecx
```

```
div  ebx
```

```
edx=(edx:eax)%S   (signed)
eax=(edx:eax)/S
...               (unsigned)
```

- **Conversões**
  - cltd
  - movsx D(word), S(byte)
  - movzx D(word), S(byte)

```
cltd
movsx eax,dh
movzx eax,dh
```

```
(edx:eax)=eax   (signed)
 D=S            (signed)
 ...            (unsigned)
```

Instruções com Flags

- **Comparações**
  - cmp A , B

    Actualiza flags com o resultado de A – B
    sem alterar A e B

```
cmp eax, 10
```

```
zf =  eax==10;
sf =  eax<10;
...
```

  - test A , B

    Actualiza flags com o resultado de A & B
    sem alterar A e B

```
test eax, eax
```

```
zf =  eax==0;
sf =  eax<0;
...
```

- **Obter valor das flags** D – registo a 8 bits fica com 0 ou 1

set(e|z) D	`D = zf;`	setn(e|z) D	`D = ~zf;`
sets D	`D = sf;`	setns D	`D = ~sf;`
set(g|nle) D	`D = ~(sf^of)&~zf;`	set(ge|nl) D	`D = ~(sf^of);`
set(l|nge) D	`D = sf^of;`	set(le|ng) D	`D = (sf^of)|zf;`
set(a|nbe) D	`D = ~cf&~zf;`	set(ae|nb) D	`D = ~cf;`
set(b|nae) D	`D = cf;`	set(be|na) D	`D = cf|zf;`

Jumps

- **Jump incondicional**
  - Directo: jmp L

```
jmp .L5
```

```
goto L5;
```

A execução continua na instrução do endereço indicado

  - Indirecto: jmp S

A execução continua na instrução do endereço armazenado no registo

```
jmp eax
```

```
goto eax;
```

ou posição de memória indicada

```
jmp [ebx+10]
```

```
goto Mw[ebx+10]
```

- **Jumps condicionados aos valor das flags**

– j(e|z) L	`if (zf) goto L;`	jn(e|z) L	`if (~zf) goto L;`
– js L	`if (sf) goto L;`	jns L	`if (~sf) …`
– j(g|nle) L	`if (~(sf^of)&~zf)`	j(ge|nl) L	`if (~(sf^of))`
– j(l|nge) L	`if (sf^of)`	j(le|ng) L	`if ((sf^of)|zf)`
– j(a|nbe) L	`if (~cf&~zf)`	j(ae|nb) L	`if (~cf)`
– j(b|nae) L	`if (cf)`	j(be|na) L	`if (cf|zf)`