Lic. Eng.ª Informática

1º ano

2009/10

A.J.Proença

Tema

ISA do IA-32

Estrutura do tema ISA do IA-32

1. Desenvolvimento de programas no IA-32 em Linux
2. Acesso a operandos e operações
3. Suporte a estruturas de controlo
4. Suporte à invocação/retorno de funções
5. Acesso e manipulação de dados estruturados
6. Análise comparativa: IA-32 (CISC) e MIPS (RISC)

Nome	*Data*	*Nº transístores*	
8086	1978	29K	

– *processador 16-bits (registos + ALU); base do IBM PC & DOS*
– *espaço de endereçamento limitado a 1MB (DOS apenas vê 640K)*

80286 1982 134K
– *endereço 24-bits e protected-mode; base do IBM PC-AT & Windows*

386 1985 275K → **primeiro IA-32 !!**
– *estendido para 32-bits: registos + op. inteiros + endereçamento*
– *memória segmentada+paginada, capaz de correr Unix*

486 1989 1.9M
– *integração num único chip: 386, co-proc 387, até 16kB cache L1*
– *poucas alterações na arquitectura interna do processador*

Pentium 1993 3.1M *(= **P5**)*
– *arquitectura superescalar, com 2 pipelines de inteiros (de 5 níveis)*

PentiumPro 1995 5.5M *(= **P6** , aka i686)*
– *out-of-order execution, 14 níveis pipeline, 3-issue superscalar*
– *endereço 36-bits, cache L2 on-package*

Pentium/MMX 1997 4.5M
– *SIMD: opera com vectores de 64-bits, tipo `int` de 1, 2, ou 4 bytes*

Pentium II 1997 7.5M *(= Pro + MMX)*

Pentium III 1999 8.2M
– *"Streaming SIMD Ext", SSE: vectores 128-bits, `int/fp` 1/2/4 bytes*

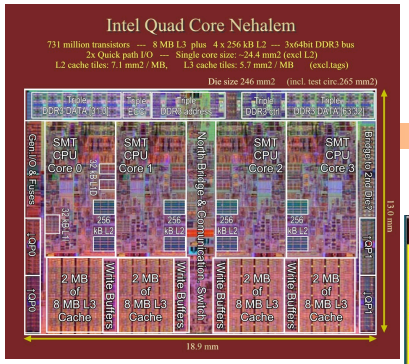
Pentium 4 2000 42M *(= **NetBurst**, aka i786)*
– *trace cache, pipeline muito longo (20 ou 31), suporta multi-threading*
– *SSE2: mais instruções e com dados fp de 8-bytes*

Pentium M 2003 77M *(= **P-M**)*
– *arquitectura mais próxima do Pentium III (eficiência energética)*

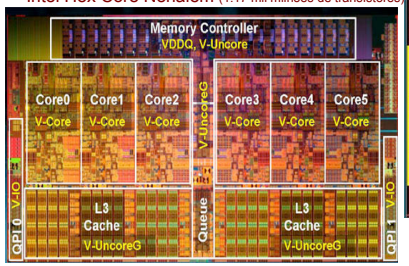
Evolução do IA-32 para Intel 64 (visão do programador)

- IA-32 ou *x86 open architecture* vai crescer para 64-bits
  - *HP e Intel propõem arquitectura incompatível, IA-64 (Itanium CPU)*
  - *AMD anuncia em 1999 extensão do x86: x86-64*
  - *Intel segue AMD: IA-32e (Fev-04) EM64T (Mar-04), ou Intel-64 (2006)*
  - *AMD64 e Intel-64 diferentes; compiladores usam sub-set comum*
- arquitectura Core surge em 2006 *(151M transístores)*
  - *desenvolvida pela mesma equipa que o P-M (Israel)*
  - *14 níveis de pipeline (como P6), mas 4-issue superscalar*
  - *multi-core on-chip (mesmo o Solo!) e virtualização por h/w*
  - *suporta fusão de instruções RISC (μ-ops na terminologia Intel)*
  - *arquitectura Core 2 é integralmente 64-bit (Intel 64)*
- arquitectura Nehalem anunciada em 2008 *(731M transístores)*
  - *inspirada no NetBurst (com multi-threading e maiores clock rates)*
  - *2 a 8 cores por chip, com cache L3 on-chip*
  - *com conexão ponto-a-ponto inter-CPUs*
  - *evolui para Westmere em 2010 (dual-die hex core)*

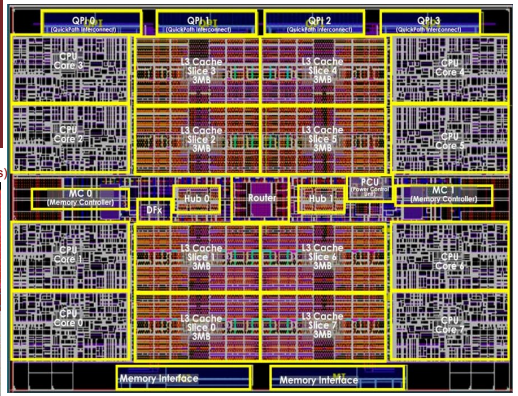
Gerações de Pentium mais recentes



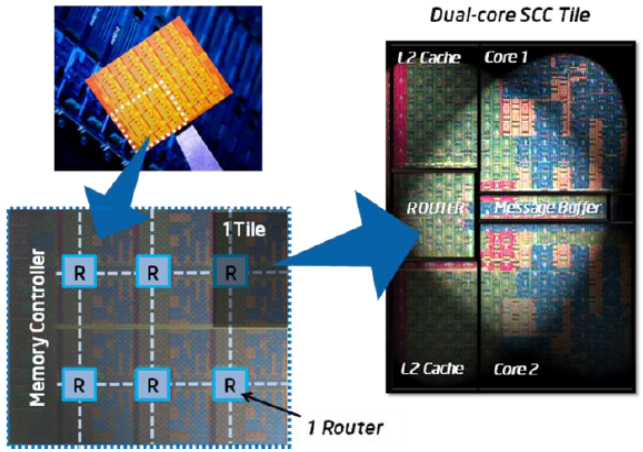
Intel Hex Core Nehalem (1.17 mil milhões de transístores)



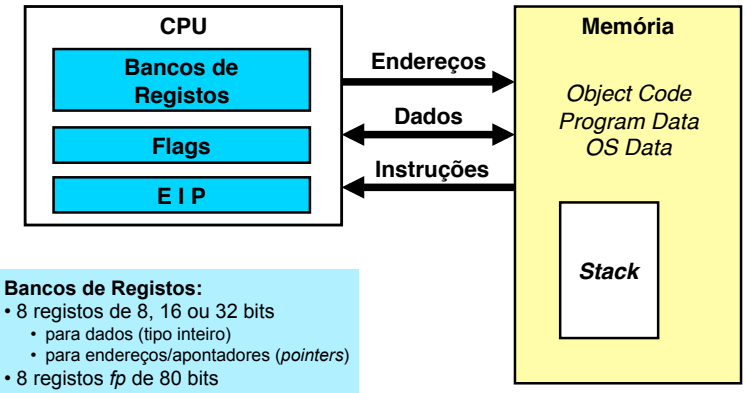
Intel Octal Core Nehalem (2.3 mil milhões de transístores)



Protótipo da Intel em 2010: Single-chip Cloud Computer



O modelo CPU-Mem no IA-32 (visão do programador)



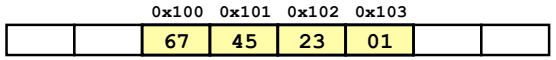
Bancos de Registos:
- 8 registos de 8, 16 ou 32 bits
  - para dados (tipo inteiro)
  - para endereços/apontadores (*pointers*)
- 8 registos *fp* de 80 bits

Flags:
- estado da última op arit/lóg

- Tamanhos de objectos em C (em *bytes*)

Declaração em C	Designação Intel	Tamanho IA-32
char	byte	1
short	word	2
int	double word	4
long int	double word	4
float	single precision	4
double	double precision	8
long double	extended precision	10/12
char * *(ou qq outro apontador)*	double word	4

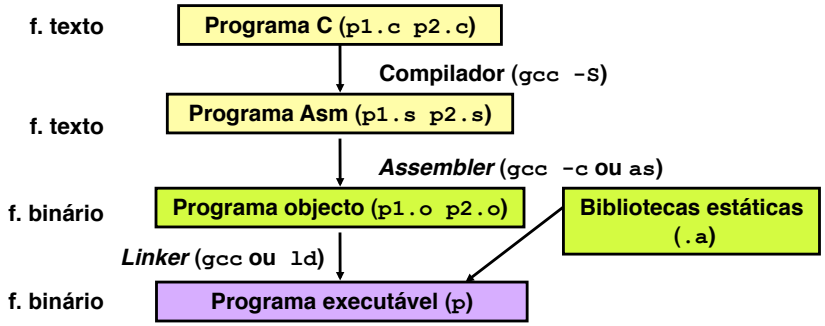
- Ordenação dos *bytes* na memória
  - **O IA-32 é um processador *little endian***
  - **Exemplo:**
    representação de `0x01234567`, cujo endereço dado por `&var` é `0x100`



Operações primitivas:

- Efectuar operações aritméticas/lógicas com dados em registo ou em memória
  - dados do tipo *integer* de 1, 2 ou 4 *bytes*
  - dados em formato *fp* de 4, 8 ou 10 *bytes*
  - só com dados escalares: *arrays* ou *structures* são vistos apenas como *bytes* continuamente alocados em memória
- Transferir dados entre células de memória e um registo
  - carregar (*load*) em registo dados da memória
  - armazenar (*store*) na memória dados em registo
- Transferir o controlo da execução das instruções
  - saltos incondicionais de/para funções/procedimentos
  - saltos ramificados (*branches*) condicionais

- Código C nos ficheiros `p1.c p2.c`
- Comando para a "compilação": `gcc -O2 p1.c p2.c -o p`
  - usa optimizações (`-O2`)
  - coloca binário resultante no ficheiro `p`

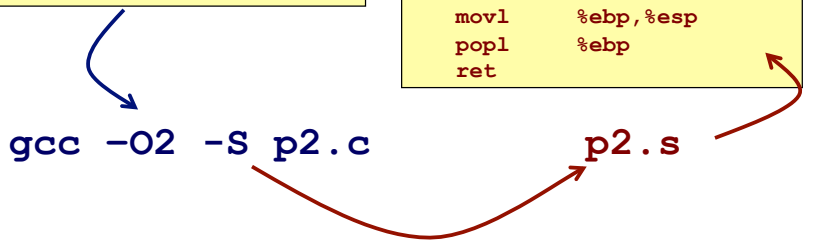


Código C

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

***Assembly* gerado**

```
_sum:
    pushl   %ebp
    movl    %esp,%ebp
    movl    12(%ebp),%eax
    addl    8(%ebp),%eax
    movl    %ebp,%esp
    popl    %ebp
    ret
```



De assembly para binário e executável (exemplo)

Assembly

```
_sum:
    pushl   %ebp
    movl    %esp,%ebp
    movl    12(%ebp),%eax
    addl    8(%ebp),%eax
    movl    %ebp,%esp
    popl    %ebp
    ret
```

p2.s → p2.o

Código binário

```
0x401040 <sum>
:   0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- Começa no endereço 0x401040
- Total 13 bytes
- Cada instrução 1, 2, ou 3 bytes

Papel do *linker*

- **Resolve as referências entre ficheiros**
- **Junta as *static run-time libraries***
  - E.g., código para `malloc`, `printf`
- **Algumas bibliotecas são *dynamically linked***
  - E.g., junção ocorre no início da execução

Desmontagem de código binário executável (exemplo)

```
objdump -d p
```

Código binário desmontado

```
00401040 <_sum>:
   0:    55              push   %ebp
   1:    89 e5           mov    %esp,%ebp
   3:    8b 45 0c        mov    0xc(%ebp),%eax
   6:    03 45 08        add    0x8(%ebp),%eax
   9:    89 ec           mov    %ebp,%esp
   b:    5d              pop    %ebp
   c:    c3              ret
   d:    8d 76 00        lea    0x0(%esi),%esi
```


Método alternativo de análise do código binário executável (exemplo)

Entrar primeiro no depurador `gdb`: **`gdb p`** e…

- examinar apenas alguns bytes: **`x/13b sum`**

```
0x401040<sum>:   0x55 0x89 0xe5 0x8b 0x45 0x0c 0x03 0x45
0x401040<sum+8>: 0x08 0x89 0xec 0x5d 0xc3
```

… ou

- proceder à desmontagem do código: **`disassemble sum`**

```
0x401040   <sum>:      push   %ebp
0x401041   <sum+1>:    mov    %esp,%ebp
0x401043   <sum+3>:    mov    0xc(%ebp),%eax
0x401046   <sum+6>:    add    0x8(%ebp),%eax
0x401049   <sum+9>:    mov    %ebp,%esp
0x40104b   <sum+11>:   pop    %ebp
0x40104c   <sum+12>:   ret
0x40104d   <sum+13>:   lea    0x0(%esi),%esi
```


Que código pode ser desmontado?

Qualquer ficheiro que possa ser interpretado como código executável

- o *disassembler* examina os *bytes* e reconstrói a fonte *assembly*

```
% objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55                      push   %ebp
30001001:  8b ec                   mov    %esp,%ebp
30001003:  6a ff                   push   $0xffffffff
30001005:  68 90 10 00 30          push   $0x30001090
3000100a:  68 91 dc 4c 30          push   $0x304cdc91
```

Estrutura do tema ISA do IA-32

1. Desenvolvimento de programas no IA-32 em Linux
2. Acesso a operandos e operações
3. Suporte a estruturas de controlo
4. Suporte à invocação/retorno de funções
5. Acesso e manipulação de dados estruturados
6. Análise comparativa: IA-32 (CISC) e MIPS (RISC)

Localização de operandos no IA-32

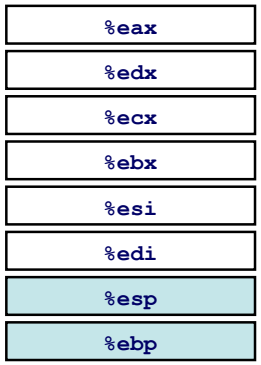
- valores de constantes (ou valores imediatos)
  - incluídos na instrução, i.e., no Reg. Instrução
- variáveis escalares
  - sempre que possível, em registos (inteiros/apont) / *fp* ; se não...
  - na memória
- variáveis estruturadas
  - sempre na memória, em células contíguas

Modos de acesso a operandos no IA-32

- em instruções de transferência de informação
  - instrução mais comum: `movx` , sendo `x` o tamanho (`b`, `w`, `l`)
  - algumas instruções actualizam apontadores (por ex.: `push`, `pop`)
- em operações aritméticas/lógicas

- Transferência simples

  `movl` *`Source,Dest`*
  - move uma *word* de 4 *bytes* ("*long*")
  - instrução mais comum em código de IA-32
- Tipos de operandos
  - imediato: valor constante do tipo inteiro
    - como a constante C, mas com prefixo '`$`'
    - ex.: `$0x400`, `$-533`
    - codificado com 1, 2, ou 4 *bytes*
  - em registo: um de 8 registos inteiros
    - mas… `%esp` and `%ebp` reservados…
    - outros poderão ser usados implicitamente…
  - em memória: 4 *bytes* consecutivos de memória
    - vários modos de especificar o endereço…



	Fonte	Destino		Equivalente em C
`movl`	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`
		Mem	**não é possível no IA32 efectuar transferências memória-memória numa só instrução**	

Modos de endereçamento à memória no IA-32 (1)

- **Indirecto (*normal*) (R) Mem[Reg[R]]**
  - registo R especifica o endereço de memória

```
movl (%ecx),%eax
```

- **Deslocamento D(R) Mem[Reg[R]+D]**
  - registo R especifica início da região de memória
  - deslocamento constante D especifica distância do início

```
movl 8(%ebp),%edx
```


Exemplo de utilização de modos simples de endereçamento à memória no IA-32 (1)

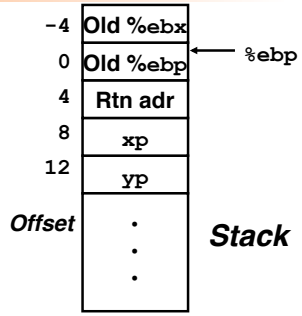
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Arranque — **Corpo** — **Término**

Exemplo de utilização de modos simples de endereçamento à memória no IA-32 (2)

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

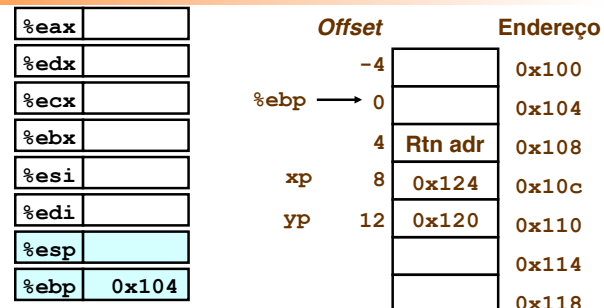


Registo	Variável
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

Corpo

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

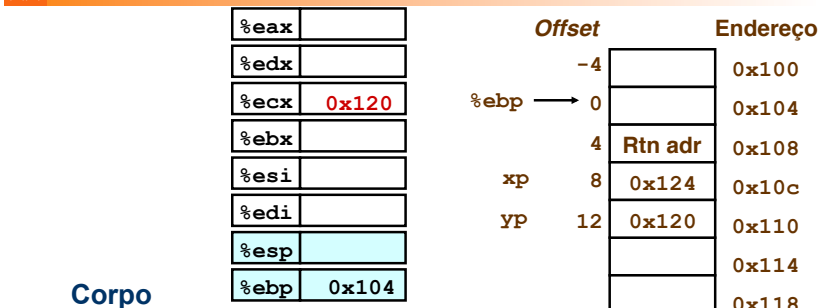

Exemplo de utilização de modos simples de endereçamento à memória no IA-32 (3)



Corpo

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

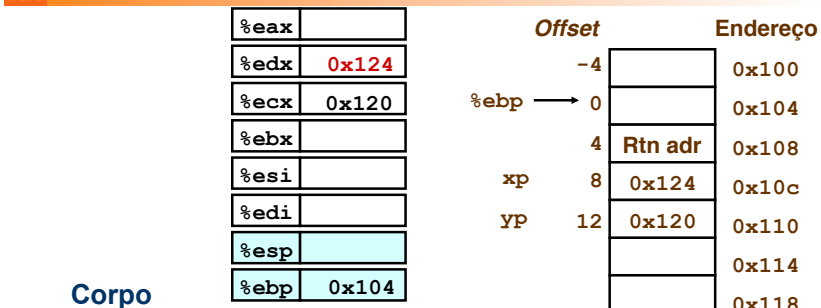
Exemplo de utilização de modos simples de endereçamento à memória no IA-32 (4)



Corpo

```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```

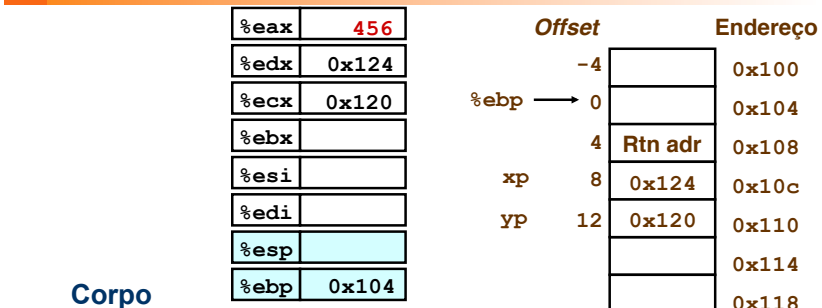

Exemplo de utilização de modos simples de endereçamento à memória no IA-32 (4)



Corpo

```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```

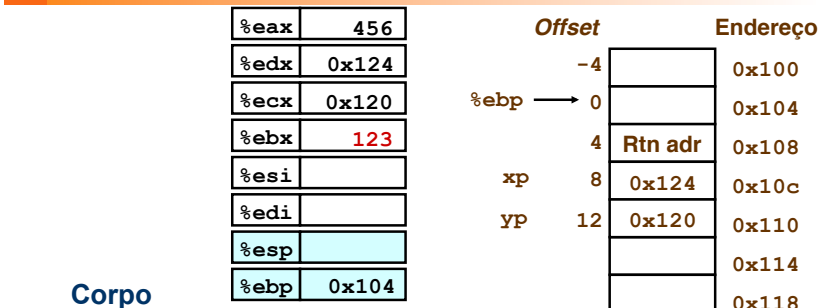

Exemplo de utilização de modos simples de endereçamento à memória no IA-32 (5)



Corpo

```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```

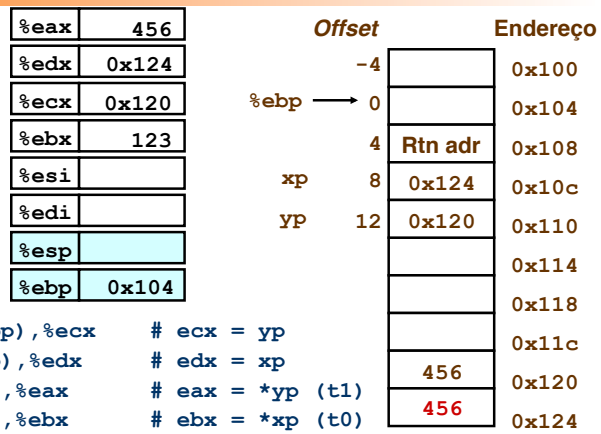

Exemplo de utilização de modos simples de endereçamento à memória no IA-32 (6)



Corpo

```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```

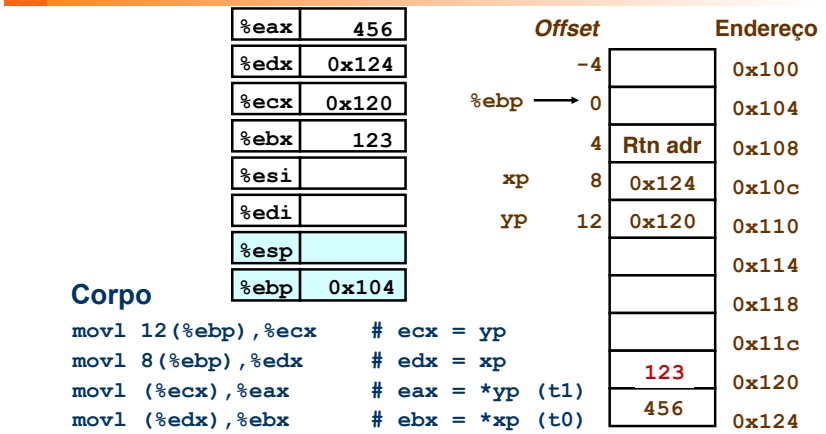
Exemplo de utilização de modos simples de endereçamento à memória no IA-32 (7)



Corpo

```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```


Exemplo de utilização de modos simples de endereçamento à memória no IA-32 (8)



Corpo

```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```


Modos de endereçamento à memória no IA-32 (2)

- Indirecto (R) Mem[Reg[R]] . . .
- Deslocamento D(R) Mem[Reg[R] + D] . . .
- **IndexadoD(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]**

  D: Deslocamento constante de 1, 2, ou 4 *bytes*

  Rb: Registo Base: quaisquer dos 8 Reg Int

  Ri:Registo Indexação: qualquer, excepto `%esp`

  S: Scale: 1, 2, 4, ou 8

  Casos particulares:

(Rb,Ri)	Mem[Reg[Rb] + Reg[Ri]]
D(Rb,Ri)	Mem[Reg[Rb] + Reg[Ri] + D]
(Rb,Ri,S)	Mem[Reg[Rb] + S*Reg[Ri]]

Exemplo de instrução do IA-32 apenas para cálculo do endereço efectivo do operando (1)

`leal` *Src,Dest*

- ***Src*** contém a expressão para cálculo do endereço
- ***Dest*** vai receber o resultado do cálculo da expressão

- **Tipos de utilização desta instrução:**
  - cálculo de um endereço sem acesso à memória
    - Ex.: tradução de `p = &x[i];`
  - cálculo de expressões aritméticas do tipo
    `x + k*y` `para k` = 1, 2, 4, or 8

- **Exemplo ...**

Exemplo de instrução do IA-32 apenas para cálculo do endereço efectivo do operando (2)

`leal` ***Source***`,%eax`

`%edx`	`0xf000`
`%ecx`	`0x100`

Source	Expressão	-> %eax
`0x8(%edx)`	`0xf000 + 0x8`	`0xf008`
`(%edx,%ecx)`	`0xf000 + 0x100`	`0xf100`
`(%edx,%ecx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%edx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Instruções de transferência de informação no IA-32

`movx S,D`	D←S	Move (**b**yte, **w**ord, **l**ong-word)
`movsbl S,D`	D←SignExtend(S)	Move Sign-Extended Byte
`movzbl S,D`	D←ZeroExtend(S)	Move Zero-Extended Byte
`push S`	%esp ← %esp - 4; Mem[%esp] ← S	Push
`pop D`	D←Mem[%esp]; %esp ←%esp+ 4	Pop
`lea S,D`	D← &S	Load Effective Address

D – destino [Reg | Mem] **S** – fonte [Imm | Reg | Mem]
D e **S** não podem ser ambos operandos em memória

Operações aritméticas e lógicas no IA-32

inc D	D← D +1	Increment
dec D	D← D –1	Decrement
neg D	D← -D	Negate
not D	D← ˜D	Complement
add S, D	D← D + S	Add
sub S, D	D← D – S	Subtract
imul S, D	D← D * S	32 bit Multiply
and S, D	D← D & S	And
or S, D	D← D | S	Or
xor S, D	D← D ^ S	Exclusive-Or
shl k, D	D← D << k	Left Shift
sar k, D	D← D >> k	Arithmetic Right Shift
shr k, D	D← D >> k	Logical Right Shift