

# Assembly do IA-32 em ambiente Linux

Trabalho para Casa: TPC5

Alberto José Proença

## Objectivo

A lista de exercícios propostos em TPC5 – para resolução antes e durante a próxima sessão TP – analisa e complementa os seguintes aspectos relacionados com o nível ISA do IA-32: **transferência de informação, operações aritméticas/ lógicas e instruções de salto**. Os exercícios para serem resolvidos e entregues antes da aula TP estão na última folha.

## Exercícios

### Acesso a operandos

1. <sup>(A)</sup> Considere que os seguintes valores estão guardados em registos e em células de memória:

Endereço	Valor
0x100	0xFF
0x101 a 103	0
0x104	0xAB
0x105 a 107	0
0x108	0x13
0x109 a 10B	0
0x10C	0x11

Registo	Valor
%eax	0x100
%ecx	0x1
%edx	0x3

Preencha a seguinte tabela mostrando os valores (em hex) para os operandos indicados. Notas: (i) o operando é um valor de 32 bits e a sua sintaxe é a utilizada no *assembly* do GNU; (ii) no comentário indique se o valor é imediato (uma  $c^{te}$ ), ou se encontra em Reg ou em Mem[<endereço>].

Operando	Valor	Comentário
%eax		
0x104		
\$0x108		
(%eax)		
4(%eax)		
9(%eax,%edx)		
260(%ecx,%edx)		
0xFC(,%ecx,4)		
(%eax,%edx,4)		

### Transferência de informação em funções

2. <sup>(R)</sup> Considere que a seguinte função, cuja assinatura (*prototype*) vem dada por

```
void decode1(int *xp, int *yp, int *zp);
```

é compilada para o nível do *assembly*. O corpo da função fica assim codificado:

```
1    movl    8(%ebp),%edi
2    movl    12(%ebp),%ebx
3    movl    16(%ebp),%esi
4    movl    (%edi),%eax
5    movl    (%ebx),%edx
6    movl    (%esi),%ecx
7    movl    %eax,(%ebx)
8    movl    %edx,(%esi)
9    movl    %ecx,(%edi)
```

Os parâmetros *xp*, *yp*, e *zp* estão armazenados nas posições de memória com um deslocamento de 8, 12, e 16, respectivamente, relativo ao endereço no registo `%ebp`.

Escreva código C para `decode1` que tenha um efeito equivalente ao programa em *assembly* apresentado em cima. Verifique a sua proposta compilando com o `switch -S`. O compilador que usar poderá eventualmente gerar código com uma utilização diferente dos registos ou de ordenação das referências à memória, mas deverá ser funcionalmente equivalente.

### Load effective address

3. <sup>(R)</sup> Considere que o registo `%eax` contém o valor de *x*, `%ecx` o valor de *y* e `%edx` foi alocado à variável *z*. Preencha a tabela seguinte, com expressões (fórmulas) que indiquem o valor que será armazenado no registo `%edx` para cada uma das seguintes instruções em *assembly*:

Instrução	Valor
<code>leal 6(%eax), %edx</code>	$z = 6 + x$
<code>leal (%eax,%ecx), %edx</code>	
<code>leal (%eax,%ecx,4), %edx</code>	
<code>leal 7(%eax,%eax,8), %edx</code>	
<code>leal 0xA(,%ecx,4), %edx</code>	
<code>leal 9(%eax,%ecx,2), %edx</code>	

### Operações aritméticas

4. <sup>(A)</sup> Considere que os seguintes valores estão guardados em registos e em células de memória:

Endereço	Valor
0x100	0xFF
0x101 a 103	0
0x104	0xAB
0x105 a 107	0
0x108	0x13
0x109 a 10B	0
0x10C	0x11

Registo	Valor
<code>%eax</code>	0x100
<code>%ecx</code>	0x1
<code>%edx</code>	0x3

Preencha a seguinte tabela, mostrando os efeitos das instruções seguintes em termos de localização dos resultados (em registo ou endereço de memória), e dos respectivos valores:

Instrução	Destino	Valor
<code>addl %ecx, (%eax)</code>		
<code>subl %edx, 4(%eax)</code>		
<code>imull \$16, (%eax, %edx, 4)</code>		
<code>incl 8(%eax)</code>		
<code>decl %ecx</code>		
<code>subl %edx, %eax</code>		

### Operações lógicas e de manipulação de bits

A linguagem C disponibiliza um conjunto de operações Booleanas - `|` para OR, `&` para AND, `~` para NOT - as quais admitem como operandos qualquer tipo de dados “integral”, i.e., declarados como `char` ou `int`, com ou sem qualificadores (`short`, `long`, `unsigned`). Estas operações aplicam-se sobre cada um dos bits dos operandos (mais detalhe em 2.1.8 de CSAPP).

Adicionalmente, a linguagem C disponibiliza ainda um conjunto de operadores lógicos, `||`, `&&`, e `!`, os quais correspondem às operações OR, AND e NOT da lógica proposicional. As operações lógicas consideram qualquer argumento distinto de zero como sendo `True`, e o argumento 0 representando `False`; devolvem o valor 1 ou 0, indicando, respetivamente, um resultado de `True` ou `False`.

5. <sup>(B)</sup> Usando apenas estas operações, escreva código em C contendo expressões que produzam o resultado “1” se a condição descrita for verdadeira, e “0” se falsa. Considere `x` como sendo um valor inteiro.
- Pelo menos um bit de `x` é “1”
  - Pelo menos um bit de `x` é “0”
  - Pelo menos um bit no *byte* menos significativo de `x` é “1”
  - Pelo menos um bit no *byte* menos significativo de `x` é “0”

6. <sup>(R)</sup> Na compilação do seguinte ciclo:

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

encontrou-se a seguinte linha de código *assembly*:

```
xorl %edx, %edx
```

Explique a presença desta instrução, sabendo que não há operadores de XOR no código C. Que operação do programa, em C, conduz à implementação desta instrução em *assembly*?

### Operações de deslocamento

7. <sup>(R)</sup> Suponha que se pretende gerar código *assembly* para a seguinte função C:

```
int shift_left2_rightn(int x, int n)
{
    x <<= 2;
    x >>= n;
    return x;
}
```

Apresenta-se de seguida uma porção do código *assembly* que efetua as operações de deslocamento e deixa o valor final em `%eax`. Duas instruções chave foram retiradas. O parâmetros `x` e `n` estão armazenados nas posições de memória com um deslocamento relativo ao endereço no registo `%ebp` de, respetivamente, 8 e 12 células.

```

1   movl    8(%ebp),%eax      Get x
2   movl    12(%ebp),%ecx    Get n
3   _____             x <=<= 2
4   _____             x >>= n

```

Complete o programa com as instruções em falta, de acordo com os comentários à direita. O *right shift* deverá ser realizado aritmeticamente.

### Operações de comparação

8. <sup>(R)</sup>No código C a seguir, substituiu-se alguns dos operadores de comparação por “`__`” e retiraram-se os tipos de dados nas conversões de tipo (*cast*).

```

1 char ctest(int a, int b, int c)
2 {
3     char t1 = a __ b;
4     char t2 = b __ ( ) a;
5     char t3 = ( ) c __ ( ) a;
6     char t4 = ( ) a __ ( ) c;
7     char t5 = c __ b;
8     char t6 = a __ 0;
9     return t1 + t2 + t3 + t4 + t5 + t6;
10 }

```

A partir do código original em C, o GCC gera o seguinte código *assembly*:

```

1   movl    8(%ebp),%ecx      Buscar argumento a
2   movl    12(%ebp),%esi    Buscar argumento b
3   cmpl   %esi,%ecx        Comparar a:b
4   setl   %al              Calcular t1
5   cmpl   %ecx,%esi        Comparar b:a
6   setb   -1(%ebp)         Calcular t2
7   cmpw   %cx,16(%ebp)     Comparar c:a
8   setge  -2(%ebp)         Calcular t3
9   movb   %cl,%dl          Comparar a:c
10  cmpb   16(%ebp),%dl      Calcular t4
11  setne  %bl              Comparar c:b
12  cmpl   %esi,16(%ebp)    Calcular t5
13  setg   -3(%ebp)         Testar a
14  testl  %ecx,%ecx        Calcular t4
15  setg   %dl              Somar t2 a t1
16  addb   -1(%ebp),%al     Somar t3 a t1
17  addb   -2(%ebp),%al     Somar t4 a t1
18  addb   %bl,%al          Somar t5 a t1
19  addb   -3(%ebp),%al     Somar t6 a t1
20  addb   %dl,%al          Converter a soma de char para int
21  movsbl %al,%eax

```

Baseado neste programa em *assembly*, preencha as partes em falta (as comparações e as conversões de tipo) no código C.

## Controlo do fluxo de execução de instruções

9. Nos seguintes excertos de programas desmontados do binário (*disassembled binary*), alguns itens de informação foram substituídos por X's.

**Notas:**

- (i) No *assembly* da GNU, a especificação de um endereço em modo absoluto em hexadecimal contém o prefixo `*0x`, enquanto a especificação em modo relativo se faz em hexadecimal sem qualquer prefixo;
- (ii) Não esquecer que o IA-32 é *little endian*.

Responda às seguintes questões.

- a) <sup>(A)</sup> Qual o endereço destino especificado na instrução `jge`?

```
8048d1c: 7d 9e                jge XXXXXXX
8048d1e: eb 24                jmp 8048d44
```

- b) <sup>(A)</sup> Qual o endereço em que se encontra o início da instrução `jmp`?

```
XXXXXXXX: eb 54                jmp 8047c42
XXXXXXXX: c7 45 f8 10          mov $0x10,0xffffffff8(%ebp)
```

- c) <sup>(R)</sup> Nesta alínea, o endereço da instrução de salto é especificado no modo relativo ao IP/PC, em 4 *bytes*, codificado em complemento para 2.

Qual o endereço especificado na instrução `jmp`?

```
8048902: e9 c2 10 00 00      jmp XXXXXXX
8048907: 90                  nop
```

- d) <sup>(R)</sup> Nesta alínea o código contém várias referências a endereços em instruções de salto, cujos valores se encontram na gama  $8043xxx_{16}$ . Contudo, a sua codificação em binário segue regras distintas (absoluto/relativo, 1 ou 4 *bytes*, ...). Calcule os endereços em falta para cada um dos 3 casos, e explicita a respetiva regra de codificação.

```
8043563: e9 XX XX XX XX      jmp 80436c1
8043568: 89 c2                mov %eax,%edx
804356a: 83 fa ff            cmp $0xffffffff,%edx
804356d: 74 XX                je 8043548
804356f: 89 d3                mov %edx,%ebx
8043571: ff 24 XX XX XX XX   jmp *0x8043580
```

