

## TPC5

### Resultados dos exercícios propostos

#### 1. Acesso a operandos

Operando	Valor	Comentário
%eax	0x400	no registo eax
0x404	0xcb	em Mem[0x404] a Mem[0x407]
\$0x408	0x408	c <sup>te</sup> (4 bytes, incluídos na instrução)
(%eax)	0xdd	em Mem[0x400] a Mem[0x403]
4(%eax)	0xcb	em Mem[0x404] a Mem[0x407]
9(%eax,%edx)	0x10	em Mem[0x40c] a Mem[0x40f]
1028(%ecx,%edx)	0x14	em Mem[0x408] a Mem[0x40b]
0x3fc(,%ecx,4)	0xdd	em Mem[0x400] a Mem[0x403]
(%eax,%edx,4)	0x10	em Mem[0x40c] a Mem[0x40f]

#### 2. Transferência de informação em funções

*Reverse engineering* é um bom método para compreender o funcionamento de sistemas. Neste caso, pretende-se recuperar o efeito da ação do compilador de C para determinar que código C teria dado origem a este código *assembly*. A melhor maneira é correr uma “simulação”, com os valores *x*, *y*, e *z* nas localizações especificadas pelos apontadores *xp*, *yp*, e *zp*, respetivamente. Teríamos o seguinte comportamento:

```

1  movl    8(%ebp),%edi    ;xp (1ª arg, apontador para var x) → %edi
2  movl    12(%ebp),%ebx   ;yp (2ª arg, apontador para var y) → %ebx
3  movl    16(%ebp),%esi   ;zp (3ª arg, apontador para var z) → %esi
4  movl    (%edi),%eax     ;*xp (x, valor apontado por xp) → %eax
5  movl    (%ebx),%edx     ;*yp (y, valor apontado por yp) → %edx
6  movl    (%esi),%ecx     ;*zp (z, valor apontado por zp) → %ecx
7  movl    %eax,%ebx      ;*yp = x
8  movl    %edx,%esi      ;*zp = y
9  movl    %ecx,%edi      ;*xp = z

```

A partir daqui podemos gerar o seguinte código C:

[code/asm/decode1-ans.c](#)

```

1 void decode1(int *xp, int *yp, int *zp)
2 {
3     int x = *xp;
4     int y = *yp;
5     int z = *zp;
6
7     *yp = x;
8     *zp = y;
9     *xp = z;
10 }

```

[code/asm/decode1-ans.c](#)

### 3. Load effective address

Embora a especificação do operando-fonte use a sintaxe reservada para indicar um endereço de memória, notar que não existe qualquer acesso à memória. Esta instrução é bastante usada para cálculo de expressões aritméticas, pois permite especificar mais de 2 operandos.

Instrução	Valor
<code>leal (%eax,%ecx), %edx</code>	$z = x + y$
<code>leal (%eax,%ecx,8), %edx</code>	$z = x + 8y$
<code>leal 7(%eax,%eax,4), %edx</code>	$z = 7 + 5x$
<code>leal 0xc(,%ecx,4), %edx</code>	$z = 12 + 4y$
<code>leal 6(%eax,%ecx,4), %edx</code>	$z = 6 + x + 4y$

Se quiser fazer um pouco mais de *reverse engineering*, pode tentar escrever código em C que, após compilado, gere código *assembly* contendo as expressões da tabela acima apresentada. Para que tal aconteça (i.e., para que o compilador gere expressões com `leal` em vez de `addl` e `sall`), algumas regras deverão ser seguidas:

- as variáveis `x` e `y` deverão estar alocadas a registos (i.e. deverão ser variáveis locais duma função; sugestão: modificar o código do exercício anterior, onde já se viu que as variáveis `x` e `y` eram alocadas a registos);
- os resultados dessas expressões (na coluna "Valor") deverão ser usadas em instruções nas linhas seguintes (caso contrário o compilador deteta que não é necessário gerar código e não o faz; sugestão: usar `printf` (e apenas) para cada uma das expressões); e
- deverá ser evitada a replicação de expressões, senão o compilador introduz otimizações e poderá gerar código distinto (por ex., neste exercício aparece 2 vezes `4y`; sugestão: substituir na segunda expressão por `4x`).

Se cumprir estas regras, então o código em *assembly* que o compilador geraria a partir do código C deveria conter as seguintes linhas de código (várias linhas de código foram retiradas, e os comentários foram acrescentados posteriormente):

```
.file "lea.c"

/* Corpo da função sem o código associado ao printf */
movl 8(%ebp), %eax
movl (%eax), %ebx          /* x em %ebx          */
movl 12(%ebp), %eax
movl (%eax), %esi          /* y em %esi          */
leal 6(%ebx), %eax          /* calcula 6+x;       */
leal (%esi,%ebx), %eax      /* calcula x+y        */
leal (%ebx,%esi,8), %eax    /* calcula x+8*y      */
leal 6(%ebx,%esi,4), %esi   /* calcula 6+x+4*y    */
leal 7(%ebx,%ebx,4), %eax   /* calcula 7+5*x      */
leal 12(,%ebx,4), %eax      /* calcula 12+4*x     */
```

#### 4. Operações aritméticas

Instrução	Destino	Valor
<code>addl %ecx, (%eax)</code>	Mem[0x400] a Mem[0x403]	0xde
<code>subl %edx, 4(%eax)</code>	Mem[0x404] a Mem[0x407]	0xc8
<code>imull \$16, (%eax, %edx, 4)</code>	Mem[0x40c] a Mem[0x40f]	0x100
<code>incl 8(%eax)</code>	Mem[0x408] a Mem[0x40b]	0x15
<code>decl %ecx</code>	Registo %ecx	0x0
<code>subl %edx, %eax</code>	Registo %eax	0x3fd

#### 5. Operações lógicas e de manipulação de bits

Estes exercícios pedem uma reflexão sobre a operação lógica ! de uma maneira não convencional. De um modo geral associa-se a estas operações a negação lógica; no entanto, muitas vezes elas são usadas como um modo de detetar se existe algum bit diferente de zero numa dada palavra. Indica-se em baixo as expressões que produzem o resultado "1" (se a afirmação for verdadeira) ou "0" (se falsa).

- |  |               |
|--|---------------|
| a) Pelo menos um bit de x é "1"                      | !!x           |
| b) Pelo menos um bit de x é "0"                      | !!~x          |
| c) ... no <i>byte</i> menos significativo de x é "1" | !!(x & 0xff)  |
| d) ... no <i>byte</i> menos significativo de x é "0" | !!(~x & 0xff) |

#### 6. Operações lógicas

Esta instrução é usada para colocar o valor 0 no registo %edx, usando a propriedade  $x^x = 0$ , para qualquer x. Corresponde à atribuição `i=0` em C.

Isto é um exemplo de um "idioma" na linguagem *assembly* – um fragmento de código por vezes gerado com determinado fim, neste caso o de maior eficiência, pelo facto desta instrução não necessitar de nenhum *byte* extra para representar a constante 0.

#### 7. Operações de deslocamento

Com este exercício têm a oportunidade de analisar um pouco de código *assembly* gerado pelo GCC. Uma vez carregado o parâmetro n no registo %ecx, pode-se então usar o *byte* menos significativo desse registo (%cl) para especificar a quantidade de bits a deslocar na instrução `sarl`.

3	<code>sall \$2, %eax</code>	$x \ll= 2$
4	<code>sarl %cl, %eax</code>	$x \gg= n$

#### 8. Operações de comparação

Este exercício pretende realçar o facto de que, ao converter o valor de um dos 2 operandos em *unsigned*, a comparação é efetuada como se ambos os operandos não tivessem sinal (*unsigned*), devido à forma implícita de conversão entre tipos (*casting*).

```

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

## 9. Controlo do fluxo de execução de instruções

Este exercício obriga a olhar com atenção o código “desmontado” e a pensar nos modos de codificação dos endereços-alvo nas instruções de salto. Vai obrigar também a alguma aritmética com valores hexadecimais...

- a) A 2ª instrução neste exercício tem apenas uma função didática: ilustra e sugere a resolução. Como o endereço-alvo destino é representado em binário com apenas 1 *byte*, sugere que esse *byte* deverá ser a indicação de um deslocamento (o valor 0x24) em relação ao valor do PC/IP. Isto pode ser confirmado se adicionarmos esse valor ao que deverá estar no PC/IP (depois de ter sido incrementado para apontar para a próxima instrução, i.e.,  $0x24 + 0x8048d1e + 2$ ) e verificarmos que o resultado é o valor que aparece diante de `jmp`. Assim, a instrução `jge` tem, em binário, o valor do deslocamento (com sinal) relativo ao PC/IP; como o valor é negativo (o *byte* com 0xf8), o endereço-alvo pode ser calculado de 2 modos:  $(0x8048d1c + 2) - 0x8$  (complemento para 2 de 0xf8) **ou**  $(0x8048d1c + 2) + 0xfffffffff8$  (extensão do *byte* 0xf8 para 32 bits, e desprezando o bit de *carry* no resultado da adição, já que a adição de dois valores de sinais opostos nunca dá *overflow*). Como pode confirmar no código “desmontado”, esse valor é 0x8048d16

```
8048d1c: 7d f8                jge 8048d16
```

- b) De acordo com a notação produzida pelo *disassembler*, o endereço-alvo da instrução `jmp` é o endereço absoluto 0x8047c42. De acordo com a codificação binária, este endereço deverá ser o valor relativo ao PC/IP que se encontra 0x54 *bytes* adiante da instrução `mov`. Subtraindo estes valores, chegamos ao endereço 0x8047bee, tal como pode ser confirmado pelo código desmontado.

```
0x8047bec: eb 54                jmp 8047c42
0x8047bee: c7 45 f8 10          mov $0x10,0xfffffffff8(%ebp)
```

- c) O endereço-alvo está à distância 0x10c2 (que necessita de mais que 1 *byte* para ser representado, e em *little endian*) relativo ao valor do PC/IP (0x8048907). Adicionando esses valores temos o endereço 0x80499c9

```
8048902: e9 c2 10 00 00       jmp 80499c9
```

- d) Há 3 instruções de salto a completar neste exercício:

(i) um `jmp` para um endereço especificado em modo direto, que irá ser codificado em binário com um valor relativo ao PC/IP (em *little endian*); cálculo a fazer: subtrair ao endereço destino 0x80436c1 o endereço da instrução seguinte, 0x8043568 (dá um valor positivo, maior que 0xff ou 127; logo o compilador opta por representá-lo com 4 *bytes* e não 2: 0x159):

```
8043563: e9 59 01 00 00       jmp 80436c1
```

(ii) um salto condicional, `jge`, também para um endereço especificado em modo direto e relativo ao PC/IP, ocupando neste caso apenas 1 *byte*; cálculo a fazer: o mesmo, i.e., subtrair ao endereço destino 0x8043548 o endereço da instrução seguinte, 0x804356f (dá um valor negativo, já em complemento para 2, mas representável com apenas 1 *byte*: d9):

```
804356d: 74 d9                jge 8043548
```

(iii) um `jmp` para um endereço especificado em modo indireto; i.e., a localização na memória onde se encontra o endereço-alvo da instrução de salto, vem especificada como se fosse um operando em memória (duma instrução de `mov` ou duma operação aritmética/lógica) e, neste caso, encontra-se explicitamente codificado nos últimos 4 *bytes* da instrução (na ordem inversa, por ser *little endian*):

```
8043571: ff 24 80 35 04 08     jmp *0x8043580
```