Assembly do IA-32 em ambiente Linux

Exercícios de Programação 1

Resolução

Exercício 1.1 *(Acesso a operandos)***:**

Operando	Valor	Comentário
%eax	0x100	Registo
0x104	0xAB	Endereço Absoluto
$0x108	0x108	Imediato
(%eax)	0xFF	Endereço 0x100
4(%eax)	0xAB	Endereço 0x104
9(%eax,%edx)	0x11	Endereço 0x10C
260(%ecx,%edx)	0x13	Endereço 0x108
0xFC(,%ecx,4)	0xFF	Endereço 0x100
(%eax,%edx,4)	0x11	Endereço 0x10C

Exercício 1.2 *(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 acção do compilador de C para determinar que código C teria dado origem a este código *assembly*. A melhor maneira é correr uma "simulação", começando com os valores `x`, `y`, e `z` nas localizações especificadas pelos apontadores `xp`, `yp`, e `zp`, respectivamente. Teríamos o seguinte comportamento:

```
1   movl   8(%ebp),%edi     xp
2   movl   12(%ebp),%ebx    yp
3   movl   16(%ebp),%esi    zp
4   movl   (%edi),%eax      x
5   movl   (%ebx),%edx      y
6   movl   (%esi),%ecx      z
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 tx = *xp;
4    int ty = *yp;
5    int tz = *zp;
6
7    *yp = tx;
8    *zp = ty;
9    *xp = tz;
10 }
```

code/asm/decode1-ans.c

Exercício 1.3 *(Load effective address)***:**

De notar que, embora o acesso aos operandos seja classificado como do tipo "acesso à memória", não existem nestes casos quaisquer acessos à memória.

Instrução	Valor
`leal 6(%eax), %edx`	*6 + x*
`leal (%eax,%ecx), %edx`	*x + y*
`leal (%eax,%ecx,4), %edx`	*x + 4y*
`leal 7(%eax,%eax,8), %edx`	*7 + 9x*
`leal 0xA(,%ecx,4), %edx`	*10 + 4y*
`leal 9(%eax,%ecx,2), %edx`	*9 + x + 2y*
`leal (%eax,%ecx,4), %edx`	*x + 4y*

Exercício 1.4 *(Operações aritméticas)***:**

Instrução	Destino	Valor
`addl %ecx,(%eax)`	`0x100`	`0x100`
`subl %edx,4(%eax)`	`0x104`	`0xA8`
`imull $16,(%eax,%edx,4)`	`0x10C`	`0x110`
`incl 8(%eax)`	`0x108`	`0x14`
`decl %ecx`	`%ecx`	`0x0`
`subl %edx,%eax`	`%eax`	`0xFD`

Exercício 1.5 *(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    sall $2,%eax          x <<= 2
4    sarl %cl,%eax         x >>= n
```

Exercício 1.6 *(Operações aritméticas/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 apenas precisar de 2 *bytes* para a sua codificação, uma vez que não necessita de nenhum *byte* extra para representar a constante 0.

Exercício 1.7 *(Comparações)***:**

Este exercício pretende realçar o facto de que, ao converter o valor de um dos 2 operandos em `unsigned`, a comparação é efectuada 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 }
```

Exercício 1.8 *(Controlo de fluxo)***:**

Este exercício obriga a olhar com atenção a 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 instrução `jbe` tem como endereço-alvo um valor relativo ao PC/IP (depois de ele ter sido incrementado para apontar para a próxima instrução), i.e., `0x8048d1c + 2 + 0xda` . Como o código "desmontado" mostra, esse valor é `0x8048cf8`.

```
8048d1c: 76 da             jbe 0x8048cf8
```

b) De acordo com a notação produzida pelo *disassembler*, o endereço-alvo da instrução `jmp` é o endereço absoluto `0x8048d44`. 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 `0x8048cf0`, tal como confirmado pelo código desmontado.

```
0x8048cee: eb 54           jmp 8048d44
0x8048cf0: c7 45 f8 10 00  mov $0x10,0xfffffff8(%ebp)
```

c) O endereço-alvo está à distância `000000cb` relativo a `0x8048907` (o endereço da instrução `nop`). Adicionando esses valores temos o endereço `0x80489d2`

```
8048902: e9 cb 00 00 00    jmp 0x80489d2
```

d) Um salto indirecto é representado pela instrução codificada em binário `ff 25` . O endereço a partir do qual o endereço-alvo da instrução de salto vai ser lido, encontra-se explicitamente codificado nos 4 *bytes* seguintes. Uma vez que o IA-32 é *litlle endian*, estes valores são apresentados na ordem inversa:

```
80483f0: ff 25 e0 a2 04    jmp *0x804a2e0
80483f5: 08
```