

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.**

Responda manualmente às questões na folha fornecida para o efeito e entregue a resolução até à hora de início da sessão PL seguinte, com a presença do estudante durante a sessão PL para que o TPC seja contabilizado na avaliação por participação. Não serão aceites trabalhos entregues depois deste prazo.

Exercícios

Acesso a operandos

1. ^(A) Considere a execução da instrução `movl <operando_fonte>, %ebx`; antes da execução dessa instrução os seguintes valores estão guardados em células de memória e em registos:

Endereço	Valor
0x200 a 0x203	0xdd
0x204 a 0x207	0xcb
0x208 a 0x20b	0x14
0x20c a 0x20f	0x10

Registo	Valor
%eax	0x200
%ecx	0x1
%edx	0x3

Preencha a seguinte tabela mostrando os valores (em hex) colocados em %ebx.

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 uma constante, ou se encontra em registo (indique qual) ou se encontra em memória (especifique assim a localização da 1ª célula: Mem[<endereço>]).

<operando_fonte>	%ebx	Comentário
%eax		
0x204		
\$0x208		
(%eax)		
4(%eax)		
9(%eax,%edx)		
516(%ecx,%edx)		
0x1fc(,%ecx,4)		
(%eax,%edx,4)		

Transferência de informação em funções

2. ^(R) 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 argumentos *xp*, *yp*, e *zp* estão armazenados nas posições de memória com um deslocamento de 8, 12, e 16 células, respetivamente, 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` (use o servidor remoto). 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. ^(R) 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 (%ecx,%eax), %edx</code>	
<code>leal (%eax,%ecx,8), %edx</code>	
<code>leal 7(%eax,%eax,4), %edx</code>	
<code>leal 0xc(,%ecx,4), %edx</code>	
<code>leal 6(%eax,%ecx,4), %edx</code>	

Operações aritméticas

4. ^(A) Considere que os seguintes valores estão guardados em células de memória e em registos:

Endereço	Valor
0x200 a 0x203	0xDD
0x204 a 0x207	0xCB
0x208 a 0x20b	0x14
0x20c a 0x20f	0x10

Registo	Valor
<code>%eax</code>	0x200
<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. ^(B) 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. ^(R) 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.

Sugestão: construa a tabela de verdade da operação lógica “ou-exclusivo”.

Qual o resultado da operação quando os 2 operandos são iguais?

Pense agora: porque razão o compilador escolheria esta operação em vez de `movl 0, %edx`?

Que operação do programa, em C, conduz à implementação desta instrução em *assembly*?

Operações de deslocamento

7. ^(R) 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. Os 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. (R) 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 este código *assembly* (anotado manualmente):

```

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            #
10   cmpb   16(%ebp),%dl        # comparar a:c
11   setne  %bl                # calcular t4
12   cmpl   %esi,16(%ebp)      # comparar c:b
13   setg   -3(%ebp)           # calcular t5
14   testl  %ecx,%ecx          # testar a
15   setg   %dl                # calcular t6
16   addb   -1(%ebp),%al        # adicionar t2 a t1
17   addb   -2(%ebp),%al        # adicionar t3 a t1
18   addb   %bl,%al            # adicionar t4 a t1
19   addb   -3(%ebp),%al        # adicionar t5 a t1
20   addb   %dl,%al            # adicionar t6 a t1
21   movsbl %al,%eax           # converter a soma de char para int

```

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) (A) Qual o endereço destino especificado na instrução `jge`?

```
8048d1c: 7d f8                jge XXXXXXXX
8048d1e: eb 24                jmp 8048d44
```

Sugestão: estude como foi implementada a instrução de salto incondicional (`jmp`)...

- b) (A) 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)
```

Sugestão: veja como foi codificada a instrução de salto incondicional `jmp`: especificando o endereço destino do salto de modo relativo com apenas 1 *byte*, (o valor `0x54`) este vai ser adicionado ao conteúdo do IP (que já está a apontar para a instrução `mov`) e o resultado dessa adição vai ser o destino do salto, que no código em *assembly* diz que é `0x8047c42`

- c) (R) Qual o endereço especificado na instrução `jmp`, sabendo-se que o endereço da instrução de salto é especificado no modo relativo ao IP/PC, em 4 *bytes*, codificado em complemento para 2?

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

- d) (R) Este pedaço de código contém várias referências a endereços em instruções de salto, cujos valores se encontram na gama `8043xxx16`. 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
```

Sugestão: leia as Notas (i) e (ii) em cima...

