Assembly do IA-32 em ambiente Linux

Trabalho para Casa: **TPC3**

Alberto José Proença

Prazos

Entrega da ficha no fim da sessão TP em que for discutida a sua resolução.

Introdução

A lista de exercícios propostos em TPC3 analisa e complementa aspectos relacionados com o nível ISA do IA32, leccionados em aulas teóricas e referente a 2 conjuntos de instruções: **transferência de informação e operações aritméticas/ lógicas** (ver sumários da semana 6 na página da disciplina na Web).

Exercícios

Acesso a operandos

1. **(A)** Considere que os seguintes valores estão armazenados em registos e em endereços de memória:

Endereço	Valor
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

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

Preencha a seguinte tabela mostrando os valores (em hexadecimal) para os operandos indicados (note que a sintaxe do `operando` é a utilizada no *assembly* do Gnu):

Operando	Valor
%eax	
0x104	
$0x108	
(%eax)	
4(%eax)	
9(%eax,%edx)	
260(%ecx,%edx)	
0xFC(,%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 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. **(R)** Suponha que o registo `%eax` contém o valor x e que `%ecx` contém o valor y. 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
`leal 6(%eax), %edx`	$6 + x$
`leal (%eax,%ecx), %edx`	
`leal (%eax,%ecx,4), %edx`	
`leal 7(%eax,%eax,8), %edx`	
`leal 0xA(,%ecx,4), %edx`	
`leal 9(%eax,%ecx,2), %edx`	

Operações aritméticas

4. **(A)** Considere que os seguintes valores estão armazenados em registos e em endereços de memória:

Endereço	Valor
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

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

Preencha a seguinte tabela mostrando os efeitos das instruções seguintes, quer em termos de

localização dos resultados (em registo ou endereço de memória), quer dos respectivos valores:

Instrução	Destino	Valor
`addl %ecx,(%eax)`		
`subl %edx,4(%eax)`		
`imull $16,(%eax,%edx,4)`		
`incl 8(%eax)`		
`decl %ecx`		
`subl %edx,%eax`		

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, respectivamente, 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 fôr verdadeira, e "0" se falsa. Considere `x` como sendo um valor inteiro.

a) Pelo menos um bit de `x` é "1"

b) Pelo menos um bit de `x` é "0"

c) Pelo menos um bit no *byte* menos significativo de `x` é "1"

d) 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. 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 efectua 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, respectivamente, 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 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
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 t4
16  addb    -1(%ebp),%al          Somar t2 a t1
17  addb    -2(%ebp),%al          Somar t3 a t1
18  addb    %bl,%al               Somar t4 a t1
19  addb    -3(%ebp),%al          Somar t5 a t1
20  addb    %dl,%al               Somar 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.

Nº	**Nome:**

Resolução dos exercícios

1. (A) Acesso a operandos

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

2. (R) Transferência de informação em funções

3. (R) *Load effective address*

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

4. (A) Operações aritméticas

Instrução	Destino	Valor
addl %ecx,(%eax)		
subl %edx,4(%eax)		
imull $16,(%eax,%edx,4)		
incl 8(%eax)		
decl %ecx		
subl %edx,%eax		