TPC7 e Guião Laboratorial

Resolução dos exercícios

Neste guião pretende-se analisar um código *assembly* e obter um código C equivalente. Um objetivo deste TPC é conseguir perceber um código *assembly* com alguma complexidade. O código C incompleto ajuda a perceber os vários blocos do código *assembly*. Nomeadamente: i) a estrutura de controlo é composta por um ciclo *for*, com um *if* dentro do ciclo; ii) existem 4 variáveis: 2 argumentos da função (*s* e *c*) e duas variáveis locais (*result* e *i*).

```
int contaN(char *s, int c) {
  int result=???;
  for (int i=??? ; s[i]!= ??? ; i???)
    if (s[i] >= '0' && ??? )
        result += ???;
  return result;
}
```

1. Obter o código *assemby*:

   sugere-se a utilização de `objdump -d m_contaN`, mas também é possível utilizar o gdb.

```
080483c0 <contaN>:
 80483c0:     55                      push   %ebp
 80483c1:     89 e5                   mov    %esp,%ebp
 80483c3:     56                      push   %esi
 80483c4:     53                      push   %ebx
 80483c5:     8b 75 08                mov    0x8(%ebp),%esi
 80483c8:     8b 4d 0c                mov    0xc(%ebp),%ecx
 80483cb:     8a 14 31                mov    (%ecx,%esi,1),%dl
 80483ce:     31 db                   xor    %ebx,%ebx
 80483d0:     84 d2                   test   %dl,%dl
 80483d2:     74 18                   je     80483ec <contaN+0x2c>
 80483d4:     8d 42 d0                lea    0xffffffd0(%edx),%eax
 80483d7:     3c 09                   cmp    $0x9,%al
 80483d9:     77 07                   ja     80483e2 <contaN+0x22>
 80483db:     0f be c2                movsbl %dl,%eax
 80483de:     8d 5c 18 d0             lea    0xffffffd0(%eax,%ebx,1),%ebx
 80483e2:     41                      inc    %ecx
 80483e3:     8a 04 31                mov    (%ecx,%esi,1),%al
 80483e6:     84 c0                   test   %al,%al
 80483e8:     88 c2                   mov    %al,%dl
 80483ea:     75 e8                   jne    80483d4 <contaN+0x14>
 80483ec:     89 d8                   mov    %ebx,%eax
 80483ee:     5b                      pop    %ebx
 80483ef:     5e                      pop    %esi
 80483f0:     c9                      leave
 80483f1:     c3                      ret
```

2. O preenchimento da tabela seguinte permite:

   a) Identificar os registos utilizados para cada uma das variáveis (instruções a vermelho)

   b) Identificar os limites do corpo do ciclo e a condição de paragem (instruções a azul)

   c) Identificar, dentro do ciclo *for*, a atualização do valor de `i` e de `result` (instruções a verde)

Questão	Registo atribuído pelo compilador	Instrução *Assembly*
registo com cópia do argumento `char *s`	`%esi`	`mov 0x8(%ebp),%esi`
registo com cópia do argumento `int c`	`%ecx`	`mov 0xc(%ebp),%ecx`
registo atribuído à variável local `result` e valor inicial	`%ebx`	**`mov %ebx,%eax`** **`( return(result) )`** **`xor %ebx,%ebx`** **`result`**`=0`
registo atribuído à variável local `i` e valor inicial	`%ecx?`	`mov 0xc(%ebp),%ecx` `i = c?` (? - confirmado depois ao preencher mais linhas da tabela)
condição de teste/salto do ciclo `for` e limites do corpo do ciclo (endereços)	`Repete se %al!=0`	**`test %al,%al`** ~~`mov %al,%dl`~~ **`80483ea: jne 80483d4`** `80483d4_ a 80483ea__`
Atualização do valor de `result` dentro do ciclo	`%ebx`	**`lea -48(%eax,%ebx,1),%ebx`** `ebx = eax + ebx*1 - 0x30`
Atualização do valor de `i` dentro do ciclo	`%ecx`	**`inc %ecx`**
condição de paragem do ciclo	`%al = s[i]` `%al !=0` (nota char 8 bits)	`mov (%ecx, %esi,1),%al` **`test %al,%al`** **`jne 80483d4`**

3. Código C da função `contaN` com as respostas anteriores:

```
int contaN(char *s, int c) {
  int result=0;
  for (int i=c ; s[i]!= 0 ; i++)
    if (s[i] >= '0' && ??? )
        result += s[i]-0x30;
  return result;
}
```

Falta completar a condição do if, para isso terá que ser analisado o código que implementa o if:

```
80483d4:    8d 42 d0          lea    0xffffffd0(%edx),%eax
80483d7:    3c 09             cmp    $0x9,%al
80483d9:    77 07             ja     80483e2 <contaN+0x22>
80483db:    0f be c2          movsbl %dl,%eax
80483de:    8d 5c 18 d0       lea    0xffffffd0(%eax,%ebx,1),%ebx
80483e2:    41                inc    %ecx
```

Isto corresponde a: `if (s[i]=>'0' && s[i]<='9')` (ver explicação em anexo).

Anexo 1: Explicação sobre a utilização de JA (JUMP ABOVE)

Para testar se um dado valor está numa gama predefinida de valores (neste caso entre 0 e 9) o normal será realizar duas comparações/saltos: 1ª) se o valor for menor do que 0 não executa o corpo do ***if***, saltando para o fim do ***if***; 2ª) caso o valor seja maior do que 0, testa se este valor é maior do que 9, saltando para o fim do ***if***; nesse caso. Esta implementação origina duas comparações e dois saltos (4 instruções, coluna "gcc 13.1 -O0" da tabela).

Exemplo simples:

Código C	**gcc 13.1 -O0 (simplificado)**	**gcc 13.1 -02**

`void XXX(int a)`
`{`
`  if ( a>=0 && a<=9 )`
`  ... /* if_body */`
`}`
	`(1)  cmpl  $0, 8(%ebp)`
	`(2)  jl  .fif  # se a<0`
	`(3)  cmpl  $9, 8(%ebp)`
	`(4)  jg  .fif # se a>9`
	`  ... /* if_body */`
	`.fif:`
		`(1)  cmpl  $9, 8(%ebp)`
		`(2)  ja  .fif`
		`  ... /* if_body */`
		`.fif:`

Os compiladores utilizam uma otimização para reduzir o número de instruções executadas, substituindo as duas comparações/saltos por apenas uma comparação/salto (coluna "gcc 13.1 -O2" da tabela). A ideia é "saltar" se o valor for "*above*" 9; ou seja, se o padrão de bits, interpretado como um valor sem sinal, for maior do que 9. Ilustrando com 8 bits:

0000 0000 (0)

0000 1001 (9)

0000 1010 (10) -Já é ABOVE $9

...

1000 0000 (128, ou seja, "above $9" se for interpretado como um valor "unsigned")

...

1111 1111 (255, também é "above $9")

Analisando em mais detalhe:

"JA" salta se o resultado não originar *Carry* e se não for *Zero* (!CF && !ZF).

Exemplos em binário:

0..8 + (-9) - não há *carry*	**9 + (-9) - *carry* e *zero***	**qualquer padrão >9 - há sempre *carry***	
0000 1000 (8) 1111 0111 (-9) --- 1111 1111	0000 1001 (9) 1111 0111 (-9) --- **1** 0000 0000	0000 1010 (10) 1111 0111 (-9) --- **1** 0000 0001	1000 0000 (-127 / 128) 1111 0111 (-9) --- **1** 0111 0111

Esta estratégia só funciona se o limite inferior da comparação for 0, mas pode ser pode ser aplicada a qualquer caso semelhante, subtraindo uma constante aos limites:

`if (a>=x1 && a<=x2)  <=>  if (a-x1)>=0 && (a-x1)<=(x2-x1)`

No exemplo deste TPC x1 é 48 (0x30 ou ASCII "0") e x2 é 48+9.