TPC8 e Guião Laboratorial

Resolução dos exercícios

1 Ciclo *For*

Uma forma de se analisar o código de um ficheiro executável (e para o qual não se tenha acesso ao ficheiro fonte em HLL que lhe deu origem) consiste em **(i)** desmontar o ficheiro binário para a versão *assembly* e depois **(ii)** tentar inverter o processo de compilação e produzir código C que pareça "natural" a um programador de C. Por exemplo, não queremos código com instruções `goto`, uma vez que estas são raramente usadas em C; e muito provavelmente não usaríamos também aqui a instrução `do-while`.
Este exercício obriga-nos a pensar no processo inverso da compilação num dado enquadramento: no modo como os ciclos `for` são traduzidos.

a) Rotina...

b) Ver alínea seguinte…

c) A partir do ficheiro executável que foi disponibilizado, `m_contaN`, é possível desmontá-lo para *assembly*, localizar a parte de código desmontado correspondente à função `contaN` e ainda distinguir as partes de inicialização e término (da função), do corpo da função (a parte pertinente neste exercício).
O código desmontado da função deverá ter um aspecto semelhante ao seguinte (este código inclui já uma anotação introduzida manualmente, bem como as etiquetas):

```
contaN:
0x080483f4 <contaN+0>:  push   %ebp              ; inicializa função:   salvag/ frame pointer anterior
0x080483f5 <contaN+1>:  mov    %esp, %ebp        ;                      cria novo frame pointer (FP)
0x080483f7 <contaN+3>:  push   %esi              ;                      salvaguarda registo %esi
0x080483f8 <contaN+4>:  push   %ebx              ;                      salvaguarda registo %ebx
0x080483f9 <contaN+5>:  mov    0x8(%ebp),%esi    ; %esi =apontador p/ início array cadeia
0x080483fc <contaN+8>:  mov    0xc(%ebp),%ecx    ; inicializa com c a var controlo de ciclo (em %ecx)
0x080483ff <contaN+11>: mov    (%ecx,%esi,1),%dl ; %dl= char, na posição c da cadeia, cadeia[c]
0x08048402 <contaN+14>: xor    %ebx, %ebx        ; inicializa a var local, o somatório result: %ebx=0
0x08048404 <contaN+16>: test   %dl, %dl          ; cadeia[c]=fim-da-string? (caráter "null" em ASCII)
0x08048406 <contaN+18>: je     0x8048420<contaN+34>    ; je equivale a jz: salta p/ fim da função se 0
.L7:                                             ; ciclo
0x08048408 <contaN+20>: lea    -0x30(%edx),%eax  ; %eax= %edx(contém %dl) - 48(ASCII char "0")
0x0804840b <contaN+23>: cmp    $0x9, %al         ; compara %al com 9
0x0804840d <contaN+25>: ja     0x8048416<contaN+34>    ; salta se valor >9, i.e., não é dígito
0x0804840f <contaN+27>: movsb %dl,%eax           ; %eax= char lido estendido para 32b, c/ sinal
0x08048412 <contaN+30>: lea    -0x30(%eax,%ebx),%ebx  ; result= result + dígito na cadeia
.L4:
0x08048416 <contaN+34>: inc    %ecx              ; increm var controlo de ciclo (posição c na cadeia)
0x08048417 <contaN+35>: mov    (%ecx,%esi,1),%al ; %al= char, na posição c da cadeia, cadeia[c]
0x0804841a <contaN+38>: test   %al, %al          ; cadeia[c]=fim-da-string? (caráter "null" em ASCII)
0x0804841c <contaN+40>: mov    %al, %dl          ; %dl= cadeia[c] (não afecta bits de condição)
0x0804841e <contaN+42>: jne     0x8048408<contaN+20>  ; salta p/ início ciclo se não é fim-da-string
.L9:                                             ; fim do ciclo
0x08048420 <contaN+44>: mov    %ebx, %eax        ; %eax= result (∑ a devolver pela função)
0x08048422 <contaN+46>: popl   %ebx              ; término da função: recupera registo %ebx
0x08048423 <contaN+47>: popl   %esi              ;                    recupera registo %esi
0x08048424 <contaN+48>: leave                    ;                    recupera SP e  FP-anterior
0x08048425 <contaN+49>: ret                      ;                    recupera IP e regressa
```

De acordo com as anotações já introduzidas no código, as identificações pedidas na alínea **c)** são agora fáceis de resolver:

- sendo o `result` o somatório de dígitos a devolver, é só procurar a inicialização a zero de um registo e, se este não for o `%eax`, (neste caso é o `%ebx`) então procurar no fim uma instrução que copie o valor em `%ebx` para `%eax`;
- a variável `i` deveria ser inicializado com `c` e incrementada dentro do ciclo `for`, e isso acontece com o registo `%ecx`;
- sabendo que os 1º e 2º argumentos se encontram na *stack* à distância de, respetivamente, 8 e 12 *bytes* do valor apontado pelo *frame pointer* (em `%ebp`), é fácil de ver para que registos foram copiados;
- o ciclo `for` deverá terminar quando o caráter lido for *null* (e também deverá ser garantido que o ciclo não é executado se o caráter inicial para análise da `cadeia` for também *null*); assim o código *assembly* deverá conter 2 expressões de teste semelhantes (estão nas linhas `<contaN+16>` e `<contaN+18>` no teste antes de entrar no ciclo, e `<contaN+38>` e `<contaN+42>` dentro do ciclo `for`);
- atualização da variável `i`: procurar por uma instrução que faça o incremento de `%ecx`;
- consultando uma tabela ASCII fica-se a saber que os algarismos 0 a 9 são codificados por `0x30` a `0x39`, ou seja de 48 a 57 (em decimal);
- a atualização do somatório (na variável `result`) deveria ser feita pela expressão `result = result + ASCII_caráter_lido_estendido_para_32_bits – 48 ;` embora as instruções aritméticas do IA-32 não permitam especificar mais que 2 operandos, a expressividade da instrução `lea` talvez permita... Procurem no código!

Pode-se ainda ver que:

- as expressões de teste do ciclo `for` têm como objetivo verificar se o caráter lido é o fim da cadeia ou não, i.e., se `cadeia[c]=0` ;
- há ainda uma outra expressão de teste associada a uma típica estrutura de `if...then` para decidir quando somar os carateres (quando forem algarismos...); esta expressão tem como objetivo verificar se o caráter lido – o valor de 8 bits que está no registo `%al` – subtraído de 48 é menor ou igual a 9, o que corresponde a verificar se é o código ASCII de um algarismo ou não, i.e., se `cadeia[c] >= '0' && cadeia[c] <= '9'`; este duplo teste é feito de uma só vez ao se usar a instrução `ja` (*jump if above*) em vez de `jg` (*jump if greater*), porque se o caráter estiver na tabela ASCII antes do `'0'`, o resultado da subtração será um número negativo, codificado em complemento para 2, contendo um "1" no bit mais à esquerda, logo estará "acima" (maior do que, mas sem sinal) quando comparado com qualquer número positivo.

d) Com base no código anotado da alínea anterior (e comentários que se seguiram), e sabendo a estrutura habitual do código gerado por um compilador com um nível médio de otimização, é possível chegar-se ao seguinte código original em C:

```
1  int contaN(char *cadeia, int c)
2  {
3   int i;
4   int result=0;
5   for (i = c; cadeia[i]!='\0' ; i++)
      if (cadeia[i] >= '0' && cadeia[i] <= '9')
            result +=(cadeia[i]-'0');
6   return result;
7  }
```