TPC6 e Guião Laboratorial

Resolução dos exercícios

1 Ciclo *While*

O código gerado na compilação de ciclos pode ser complicado de analisar, devido aos diferentes tipos de otimização do código do ciclo que o compilador poderá optar, para além da dificuldade em mapear variáveis do programa a registos do CPU. Para adquirimos alguma técnica, nada como começar com um ciclo relativamente simples.

Eis o ficheiro em *assembly* que o comando `gcc –S –O2` irá gerar na máquina virtual:

```
07    pushl    %ebp
08    movl     %esp, %ebp
09    movl     16(%ebp), %ecx
10    pushl    %esi
11    testl    %ecx, %ecx
12    pushl    %ebx
13    movl     12(%ebp), %ebx
14    setg     %dl
15    cmpl     %ecx, %ebx
16    setl     %al
17    andl     %edx, %eax
18    andl     $1, %eax
19    movl     8(%ebp), %esi
20    je       .L7
21    .p2align 2,,3
22 .L5:
23    imull    %ecx, %ebx
24    addl     %ecx, %esi
25    decl     %ecx
26    testl    %ecx, %ecx
27    setg     %dl
28    cmpl     %ecx, %ebx
29    setl     %al
30    andl     %edx, %eax
31    andl     $1, %eax
32    jne      .L5
33 .L7:
34    popl     %ebx
35    movl     %esi, %eax
36    popl     %esi
37    leave
38    ret
```

a) (A) A análise do modo como os argumentos são recuperados no código da função dá-nos uma boa pista de como o `gcc` usa os registos no cálculo de expressões de teste.
Algumas versões do `gcc` geram código com diferenças significativas, como por ex., usarem os registos `%eax` e `%edx` em substituição de `%al` e `%dl` (o que requer colocação prévia a 0, uma vez que as instruções de `set` apenas alteram registos de 8-bits).

Utilização dos Registos		
Registo	Variável	Atribuição inicial
%esi	x	valor recebido do 1º arg (x)
%ebx	y	valor recebido do 2º arg (y)
%ecx	n	valor recebido do 3º arg (n)
%dl	temp1 (=n>0)	valor booleano da expressão (n > 0)
%al	temp2 (=y<n)	valor booleano da expressão (y < n)

Para confirmar esta utilização dos registos, proceda conforme sugerido na alínea **b)**.

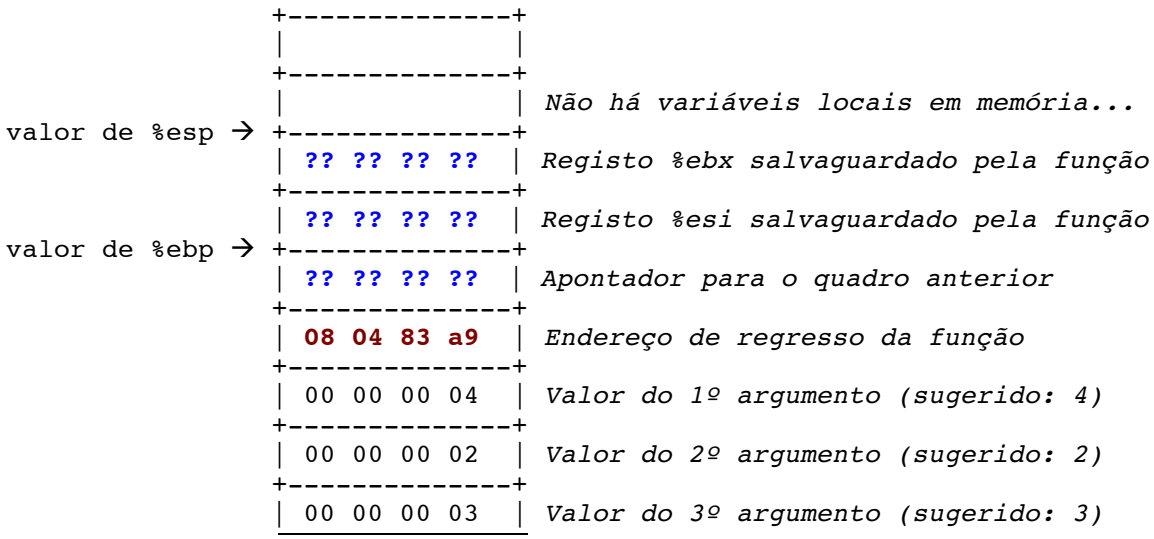
b) **(A)** (Feito na aula; valores sugeridos para inicializar `x`, `y` e `n`: 4, 2, 3).

c) **(R/B)** (Feito na aula com os grupos que melhor se prepararam antes da sessão laboratorial).

d) **(A/R)** Pretende-se neste exercício que se preencham 3 tipos de informação: **(i)** à esquerda do desenho da *stack*, os endereços do início de algumas "caixas"; **(ii)** no interior das "caixas" o valor numérico que lá deveria estar (pode ser em hexadecimal); **(iii)** à direita das "caixas", uma explicação do valor que se encontra na respectiva "caixa". Cada "caixa" não é mais que um bloco de 32 bits armazenado em 4 células de 1 *byte* cada, em que o conteúdo da célula com menor endereço é o *byte* mais à direita de um valor de 32 bits (*little endian*).
A resolução completa implica uma análise detalhada do código gerado pelo `gcc`, do conteúdo de alguns registos e da localização em que deverá ser executado no PC, entre outros aspetos. Por exemplo, se no *object dump* com desmontagem do código se observa o seguinte bloco com a `main`

```
08048394 <main>:
 8048394:   55                  push   %ebp
 8048395:   89 e5               mov    %esp,%ebp
 8048397:   83 ec 08            sub    $0x8,%esp
 804839a:   83 e4 f0            and    $0xfffffff0,%esp
 804839d:   50                  push   %eax
 804839e:   6a 03               push   $0x3
 80483a0:   6a 02               push   $0x2
 80483a2:   6a 04               push   $0x4
 80483a4:   e8 ab ff ff ff      call   8048354 <while_loop>
 80483a9:   b8 01 00 00 00      mov    $0x1,%eax
 80483ae:   c9                  leave
 80483af:   c3                  ret
```

então o valor do endereço de regresso que deverá estar na *stack* deverá ser o endereço da instrução na `main` imediatamente a seguir à invocação da função (após a instrução `call`).



O valor dos registos salvaguardados (incluindo o apontador para o quadro da `main` na *stack*, i.e., tudo o que está com ?? na figura) pode ser obtido no `gdb`, parando a execução do código logo na 1ª instrução da função (coloque aí um *breakpoint*), e analisando o conteúdo desses registos (que ainda não foram colocados na *stack*). Para confirmar os conteúdos destas 28 posições de memória na *stack*, coloque então outro *breakpoint* após a salvaguarda do último registo (que foi o `%ebx`) e quando o programa parar aí pode então usar um dos comandos para examinar dados do *debugger* e visualizar o conteúdos das 28 células com início no topo da pilha (valor em `%esp`), quer *byte* a *byte* (dá para ver o funcionamento *little endian*), quer em 7 blocos de 32 bits (e então vê como na figura atrás).

e) **(A/R)** A expressão de teste aparece na linha 3 do código C. No código *assembly*, é implementado pelas instruções nas linhas 09 a 19 (excluindo a salvaguarda de dois registos nas linhas 10 e 12; este é o cálculo inicial da expressão de teste, fora do ciclo), bem como nas instruções de 26 a 31 (cálculo dentro do ciclo).
O bloco *body-statement* encontra-se nas linhas 4 a 6 no código C, e nas linhas 23 a 25 no código *assembly*. Neste caso, o compilador não fez nenhuma otimização de destacar. Contudo, se substituir na expressão de teste o operador booleano `&` pelo operador de lógica proposicional `&&` (operações descritas no enunciado do TPC anterior), iria provavelmente encontrar uma otimização conforme descrito antes.

Eis um exemplo de código *assembly* devidamente anotado (apenas do corpo da função):

```
// Inicialmente x, y, e n estão, respetivamente, à distância de 8, 12, e 16 células do valor em %ebp
09    movl    16(%ebp), %ecx     ; coloca o 3º argumento (n) em %ecx
10    pushl   %esi               ; salvaguarda de %esi (normalmente antes do corpo)
11    testl   %ecx, %ecx         ; testa n (=n&n), i.e., afeta essencial/ os bits ZF e SF
12    pushl   %ebx               ; salvaguarda de %ebx (normalmente antes do corpo)
13    movl    12(%ebp), %ebx     ; coloca 2º arg (y) em %ebx (não afeta nenhuma flag)
14    setg    %dl                ; coloca em %dl o valor lógico de (n>0) (é 0 ou 1)
15    cmpl    %ecx, %ebx         ; calcula y-n (afetando as flags ou bits de condição)
16    setl    %al                ; coloca em %al o valor lógico de (y-n<0) ou (y<n)
17    andl    %edx, %eax         ; coloca em %eax o valor lógico de "(n>0) & (y<n)"
18    andl    $1, %eax           ; limpa %eax (e %al) exceto o bit menos significativo
19    movl    8(%ebp), %esi      ; coloca 1º arg (x) em %esi (não afeta nenhuma flag)
20    je      .L7                ; se expr_teste é F (ZF=0) vai para fim_do_ciclo
22 .L5:                        ;ciclo
23    imull   %ecx, %ebx         ; y *= n
24    addl    %ecx, %esi         ; x += n
25    decl    %ecx               ; n--
26    testl   %ecx, %ecx         ; testa n (=n&n), i.e., afeta essencial/ os bits ZF e SF
27    setg    %dl                ; coloca em %dl o valor lógico de (n>0)
28    cmpl    %ecx, %ebx         ; calcula y-n (afetando os bits de condição)
29    setl    %al                ; coloca em %dl o valor lógico de (y<n)
30    andl    %edx, %eax         ; coloca em %al o resultado de "(n>0) & (y<n)"
31    andl    $1, %eax           ; limpa %eax (e %al) exceto o bit menos significativo
32    jne     .L5                ; se expr_teste != 0, vai para ciclo
33 .L7:                        ;fim_do_ciclo
34    popl    %ebx               ; recupera %esi (normalmente depois do corpo)
35    movl    %esi, %eax         ; coloca em %eax o valor a devolver (x)
```

Note a forma de calcular a expressão de teste: o compilador sabe que as duas condições de teste – `(n>0)` e `(y<n)` – apenas podem tomar os valores de 0 ou 1, e daí apenas precisa de testar o bit menos significativo do resultado do `&`. O compilador poderia ter usado apenas a instrução `testb %dl,%al`, para efetuar a operação `&` (em vez das instruções 17 e 18).

f) **(R)** Versão do tipo `goto` (em C) da função, com uma estrutura semelhante ao do código *assembly* (tal como foi feito para a série Fibonacci):

```
1  int while_loop_goto(int x, int y, int n)
2  {
3     if (!((n > 0) & (y < n))) goto done;
4     loop:
5        x += n;
6        y *= n;
7        n--;
8        if ((n > 0) & (y < n)) goto loop;
9     done:
10    return x;
11 }
```