TPC7 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 código *assembly* que o comando `gcc -S -O2` irá gerar na máquina virtual:

```
while_loop:
        pushl   %ebp
        movl    %esp, %ebp
        movl    16(%ebp), %edx
        testl   %edx, %edx
        pushl   %ebx
        movl    12(%ebp), %eax
        movl    8(%ebp), %ebx
        jle     .L3
        movl    %edx, %ecx
        sall    $4, %ecx
        cmpl    %ecx, %eax
        jge     .L3
        .p2align 2,,3
.L6:
        addl    %edx, %ebx
        imull   %edx, %eax
        decl    %edx
        subl    $16, %ecx
        testl   %edx, %edx
        jle     .L3
        cmpl    %ecx, %eax
        jl      .L6
.L3:
        movl    %ebx, %eax
        popl    %ebx
        leave
        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.
O valor de `n*16`, utilizado numa das condições do ciclo, é pré-calculado e guardado em `%ecx`; em cada iteração do ciclo este valor é atualizado subtraindo 16, uma vez que `n` é decrementado de 1 unidade.

Utilização dos Registos		
Registo	Variável	Atribuição inicial
%ebx	x	valor recebido do 1º arg (x)
%eax	y	valor recebido do 2º arg (y)
%edx	n	valor recebido do 3º arg (n)
%ecx	temp (=n*16)	16x o valor recebido do 3º arg (n)

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

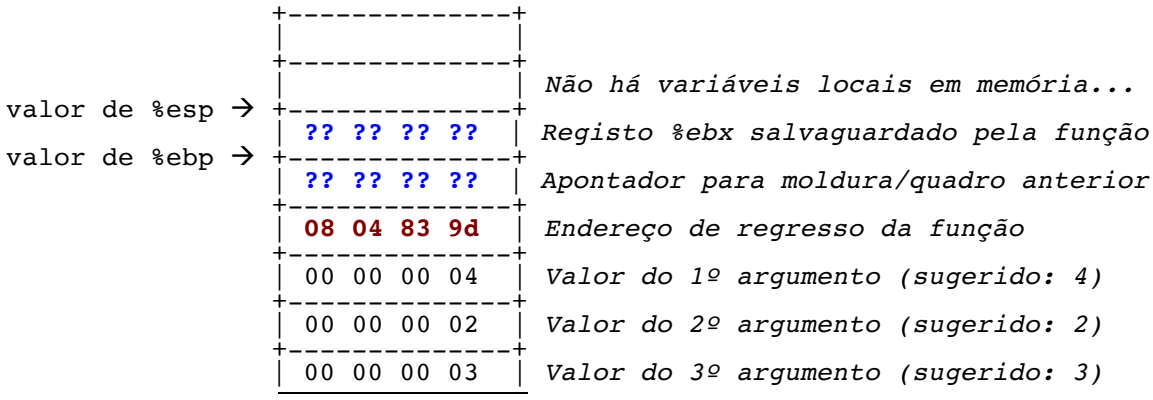
b) **(A)** (Feito na aula; valores sugeridos para inicializar `x`, `y` e `n`: `4, 2, 3`).

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, no `gdb` é possível visualizar o código da função `main` usando o comando `disas main`:

```
0x08048388 <main+0>:    push   %ebp
0x08048389 <main+1>:    mov    %esp,%ebp
0x0804838b <main+3>:    sub    $0x8,%esp
0x0804838e <main+6>:    and    $0xfffffff0,%esp
0x08048391 <main+9>:    push   %eax
0x08048392 <main+10>:   push   $0x3
0x08048394 <main+12>:   push   $0x2
0x08048396 <main+14>:   push   $0x4
0x08048398 <main+16>:   call   0x8048354 <while_loop>
0x0804839d <main+21>:   leave
0x0804839e <main+22>:   ret
```

Neste caso, 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`), ou seja `0x0804839d` (note que os endereços podem variar ligeiramente no seu caso, dependendo da forma como o código `C` está escrito).



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 24 posições de memória na *stack*, coloque então outro *breakpoint* após a salvaguarda do registo `%ebx`; quando o programa parar aí pode então usar um dos comandos para examinar dados do *debugger* e visualizar o conteúdo das 24 células com início no topo da pilha (valor em `%esp`), quer *byte* a *byte* (dá para ver o funcionamento *little endian*), quer em 6 blocos de 4 *bytes*.

Passo a passo:

(i) Escrevendo `disas while_loop` tem aceso ao código da função:

```
0x08048354 <while_loop+0>:      push   %ebp
0x08048355 <while_loop+1>:      mov    %esp,%ebp
0x08048357 <while_loop+3>:      mov    0x10(%ebp),%edx
0x0804835a <while_loop+6>:      test   %edx,%edx
0x0804835c <while_loop+8>:      push   %ebx
0x0804835d <while_loop+9>:      mov    0xc(%ebp),%eax
0x08048360 <while_loop+12>:     mov    0x8(%ebp),%ebx
0x08048363 <while_loop+15>:     jle    0x8048381 <while_loop+45>
0x08048365 <while_loop+17>:     mov    %edx,%ecx
0x08048367 <while_loop+19>:     shl    $0x4,%ecx
0x0804836a <while_loop+22>:     cmp    %ecx,%eax
0x0804836c <while_loop+24>:     jge    0x8048381 <while_loop+45>
0x0804836e <while_loop+26>:     xchg   %ax,%ax
0x08048370 <while_loop+28>:     add    %edx,%ebx
0x08048372 <while_loop+30>:     imul   %edx,%eax
0x08048375 <while_loop+33>:     dec    %edx
0x08048376 <while_loop+34>:     sub    $0x10,%ecx
0x08048379 <while_loop+37>:     test   %edx,%edx
0x0804837b <while_loop+39>:     jle    0x8048381 <while_loop+45>
0x0804837d <while_loop+41>:     cmp    %ecx,%eax
0x0804837f <while_loop+43>:     jl     0x8048370 <while_loop+28>
0x08048381 <while_loop+45>:     mov    %ebx,%eax
0x08048383 <while_loop+47>:     pop    %ebx
0x08048384 <while_loop+48>:     leave
0x08048385 <while_loop+49>:     ret
```

(ii) Coloque um *breakpoint* após `push %ebx`:

```
break *0x804835d
```

(iii) execute o programa (`run`) e ele parará no *breakpoint*; visualize o conteúdo dos registos escrevendo `info reg`

(iv) Examine 6 "palavras" na memória a partir do endereço em `%esp`:

```
x/6wx $esp
   0xbfffe874:     0x007d3ff4      0xbfffe898    0x0804839d    0x00000004
   0xbfffe884:     0x00000002      0x00000003
```

e) **(A/R)** A expressão de teste é implementada em dois blocos. O primeiro bloco verifica se o ciclo deve ser executado na 1ª iteração:

```
testl   %edx, %edx      ; testa se n é 0; n <AND> n só é zero se n for zero
…
jle     .L3             ; salta para fim do ciclo se n <= 0
movl    %edx, %ecx      ; copia n para %ecx
sall    $4, %ecx        ; %ecx = n*16
cmpl    %ecx, %eax      ; compara y com n*16
jge     .L3             ; salta para fim do ciclo se y >= 16
```

O segundo bloco, no fim do corpo do ciclo, verifica se este deve iterar:

```
decl    %edx            ; n--
subl    $16, %ecx       ; %ecx = %ecx-16 (=n*16)
testl   %edx, %edx      ; testa se n é 0
jle     .L3             ; salta para o fim do ciclo se n <= 0
cmpl    %ecx, %eax      ; compara y com n * 16
jl      .L6             ; salta para o fim do ciclo se y < n*16
```

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*16))) goto done;
4     loop:
5        x += n;
6        y *= n;
7        n--;
8        if ((n > 0) && (y < n*16)) goto loop;
9     done:
10    return x;
11 }
```