Controlo de fluxo
e
Variáveis estruturadas

Guião: **G-6r**

Exercício 1 *(Laço Do-While)*:

a)

```
1     movl      8(%ebp),%esi          ;x em %esi
2     movl      12(%ebp),%ebx         ;y em %ebx
3     movl      16(%ebp),%ecx         ;n em %ecx
4     .p2align 4,,7                 /* alinha o código na memória para optimizar a cache */
5  .L6:                             ;laço:
6     imull     %ecx,%ebx             ;y *= n
7     addl      %ecx,%esi             ;x += n
8     decl      %ecx                  ;n--
9     testl     %ecx,%ecx             ;Testa n  (n /\ n)  operação /\ bit a bit
10    setg      %al                   ;o valor lógico  (n > 0) em %al
11    cmpl      %ecx,%ebx             ;Compara y:n (y - n)
12    setl      %dl                   ;o valor lógico (y < n) em %dl (8 bits)
13    andl      %edx,%eax             ;(n > 0) /| (y < n) operação /\ bit a bit
14    testb     $1,%al                ;determina o valor do bit menos significativo em %al
15    jne       .L6                   ;Se != 0, vá para inicio de laço
```

Nota 1: os registos `%al` e `%dl` correspondem à parte *(8 bits)* menos significativa dos registos `%eax` e `%edx`, respectivamente.
Nota 2: o compilador usa uma forma pouco usual de avaliar a expressão de teste. Com efeito, no pressuposto que as duas condições de saída `(n > 0)` e `(y < n)` apenas podem tomar os valores de 0 ou 1, basta averiguar o valor (0/1) do bit menos significativo do resultado do /\. Em alternativa poderia ter sido usada a instrução `testb` para efectuar a operação `/\`.

b)

Registo	Variável	Atribuição inicial
%esi	x	x
%ebx	Y	Y
%ecx	n	n
%al	"temp1"	(n > 0)
%dl	"temp2"	(y < n)

c) O corpo do laço do-While encontra-se nas linhas 4 a 6 no código C e nas linhas 6 a 8 do código de montagem. A expressão de teste está na linha 7 do código C, a que corresponde, no código de montagem, as instruções nas linhas 9 a 14 e a condição de salto na linha 15.

Exercício 2 *(Laço While)*:

a)

```
1     movl      8(%ebp),%eax          ;a em %eax
2     movl      12(%ebp),%ebx         ;b em %ebx
3     xorl      %ecx,%ecx             ;i = 0
4     movl      %eax,%edx             ;result = a em %edx
5     .p2align 4,,7
6  .L5:                             ;laço:
7     addl      %eax,%edx             ;result += a
8     subl      %ebx,%eax             ;a -= b
9     addl      %ebx,%ecx             ;i += b
10    cmpl      $255,%ecx             ;Compara i:255 (255-i)
11    jle       .L5                   ;Se o i <= 255 salta para laço
12    movl      %edx,%eax             ; prepara retorno
```

b)

Registo	Variável	Atribuição inicial
%eax	a	a
%ebx	b	b
%ecx	i	0
%edx	result	a

c) A expressão de teste aparece na linha 5 do código C e no código de montagem na linha 10 e na condição de salto na linha 11. O corpo do laço while está nas linhas 6 a 8 do código C a que correspondem as linhas 7 a 9 no código de montagem. O compilador detectou que o teste inicial do laço while é sempre verdadeiro, uma vez que sendo i iniciado a 0 (zero) o seu valor é sempre inferior a 256. Nesta assunção, o teste inicial normalmente associado a um while pode ser evitado. Eis o código equivalente com goto:

```
1 int loop_while_goto(int a, int b)
2 {
    int i = 0;
    int result = a;
    loop:
    result += a;
    a -= b;
    i += b;
    if (i <= 255)
      goto loop;
    return result;}
```

d) Optimizações:

- Utilização de registos (`%eax, %ebx, %ecx e %edx`) para guardar as variáveis (`a,b,i result`) por forma a evitar acessos desnecessários à memória.
- Transformação do `while` num "`do while`". O compilador detectou que o teste inicial do ciclo seria executado pelo menos uma vez já que `i = 0` é obviamente inferior a 256.
- Uso da instrução `xorl %ecx, %ecx` em vez de `movl $0,%ecx` que é mais eficiente porque a instrução não necessita de octetos extras (valor imediato) para representar a constante 0. Esta instrução permite pôr o registo `%ecx` a `0`, através da operação lógica (ou exclusivo) : e.g. `%ecx^%ecx -> 000000...00` qualquer que seja o valor de '`x`', no código C corresponde à expressão `x = 0`; a versão xorl (em arquitetura IA32) requer 2 octetos (`e.g xorl 31 c0`) enquanto a versão com `movl $0, %ecx` requer 5 octetos (`b8 00 00 00 00`)

Exercício 3 *(Apontadores)***:**

Considerando que Xs corresponde ao valor do apontador para a estrutura temos:

Expressão	Tipo de Dados	Valor	Instrução
S+1	short *	xS + 2	**leal** 2(%edx),%eax
S[3]	short	Mem[xS + 6]	**movw** 6(%edx),%ax
&S[i]	short *	xS + 2 i	**leal** (%edx,%ecx,2),%eax
S[4*i+1]	short	Mem[xS + 8 i + 2]	**movw** 2(%edx,%ecx,8),%ax
S+i-5	short *	xS + 2 i – 10	**leal** -10(%edx,%ecx,2),%eax

Nota : O compilador pode substituir instruções como `movw 6(%edx),%ax` (`e.g S[3]`) pela instrução `movswl 6(%edx),%eax`. No exemplo, faz a extensão de sinal dos 2 octetos originais em memória para os 4 octetos do registo de destino `%eax`.

Exercício 4 (*Estruturas*):

a) 2 * 4 (inteiro) + 2 *4 (apontador) = 16 octetos.

b) A organização da memória correspondente à estrutura *prob* é:

Deslocamento	0	4	8	12
Conteúdo	p	s.x	s.y	next

O código da função em linguagem de montagem após a compilação:

c) O código de montagem:

```
1     movl      8(%ebp),%eax                ;sp em %eax (endereço da estrutura na memória)
2     movl      8(%eax),%edx                ;Obter sp->s.y (sp + 8)
3     movl      %edx,4(%eax)                ;Copiar para sp->s.x (sp +4)
4     leal      4(%eax),%edx                ;Obter &(sp->s.x)
5     movl      %edx,(%eax)                 ;Copiar para sp->p (sp + 0)
6     movl      %eax,12(%eax)               ;sp->next = sp (sp + 12)
```

```
void sp_init(struct prob *sp)
{
sp->s.x = sp->s.y ;
sp->p = &(sp->s.x) ;
sp->next = sp ;}
```