Guião: **G-7r**

Exercício 1. *(Funções)*:

- Inicialmente `%esp=0x800040`, na linha 2 (push) o valor é diminuído 4, `%esp=0x80003C`, e duplicado em `%ebp` na linha 3.
- Nas linhas 6 e 8, as 2 instruções `leal` calculam o valor dos argumentos a passar a `scanf`. Que são empurrados (push) para a pilha pela ordem inversa. Assim, `x` e `y` ficam deslocado, respectivamente, `-4 (0x800038)` e `-8 (0x800034)` células relativamente a `%ebp`.
- Na linha 2 `%esp=0x80003C`, na linha é reduzido 24 e na linha 5 de 4, a que se seguem as 3 instruções de `push` cuo efeito lateral é reduzir o valor de %esp de de 12. Assim no total, desde o inicio (linha 1) %esp foi reduzido de 44 unidades, donde, na linha 11 `%esp=0x800014`.

O desenho da pilha revela a estrutura e conteúdo que segue. De notar que as células de memória 0x800020 e `0x800033` e, ainda, que não são usadas e que não se tomou em consideração o formato *little endian*.

```
                +---------+
0x80003C        | 0x800060 | <-- %ebp
                +----------+
0x800038        |     0x53 | (x)
                +----------+
0x800034        |     0x46 | (y)
                +----------+
0x800030        |          |
                +----------+
0x80002C        |          |
                +----------+
0x800028        |          |
                +----------+
0x800024        |          |
                +----------+
0x800020        |          |
                +----------+
0x80001C        | 0x800038 |
                +----------+
0x800018        | 0x800034 |
                +----------+
0x800014        | 0x300070 | <-- %esp
0x800010        +---------+
```

Exercício 2. (*Vectores*):

Vector	Espaço/elemento	Espaço total	Inicial	Expressão/elemento ***i***
S	2	28	xS	xS + 2 i
T	4	12	xT	xT + 4 i
U	4	24	xU	xU + 4 i
V	12	96	xV	xV + 12 i
W	4	16	xW	xW + 4 i

Exercício 3. *(Estruturas)*: Considerando que o registo %edx foi iniciado com o valor da variável `r` definida de acordo com as declarações que seguem, explique o funcionamento dos fragmentos de código abaixo:

```
struct rec {
int i;
int j;
int a[3];
int *p;} *r;
```

a)

```
1    movl    (%edx),%eax              Get r->i
2    movl    %eax,4(%edx)             Store in r->j
3    leal    8(%edx,%eax,4),%ecx      %ecx = &r->a[r->i]
```

b)

```
r->p = &r->a[r->i + r->j];
```

```
1    movl    4(%edx),%eax             Get r->j
2    addl    (%edx),%eax              Add r->i
3    leal    8(%edx,%eax,4),%eax      Compute &r->a[r->i + r->j]
4    movl    %eax,20(%edx)            Store in r->p
```

Exercício 4. *(Laço for)*:

```
1     movl      8(%ebp),%ebx
2     movl      16(%ebp),%edx
3     xorl      %eax,%eax
4     decl      %edx
5     js        .L4
6     movl      %ebx,%ecx
7     .p2align 4,,7                ; alinha o código na memória para optimizar a cache
8    imull      12(%ebp),%ecx

9 .L6:
10    addl      %ecx,%eax
11    subl      %ebx,%edx
12    jns       .L6

13 .L4:                            ; terminação do laço
```

Pode-se ver que:

- `i` em `%edx`, `i=n` (linha 2) `i=n-1` (linha 4)
- `result=0` em `%eax` (linha 3)
- teste se i >=0 (linhas 5 e 12)
- `i` é decrementada de `x` (linha 11)
- `x*y` em `%ecx`, para o cálculo de `result` linhas 9 e 10
- `result` valor de retorno em `%eax`

```
1 int loop(int x, int y, int n)
2 {
3     int result = 0;
4     int i;
5     for (i = n-1; i >= 0; i = i-x) {
6         result += y * x;
7     }
8     return result;
9 }
```