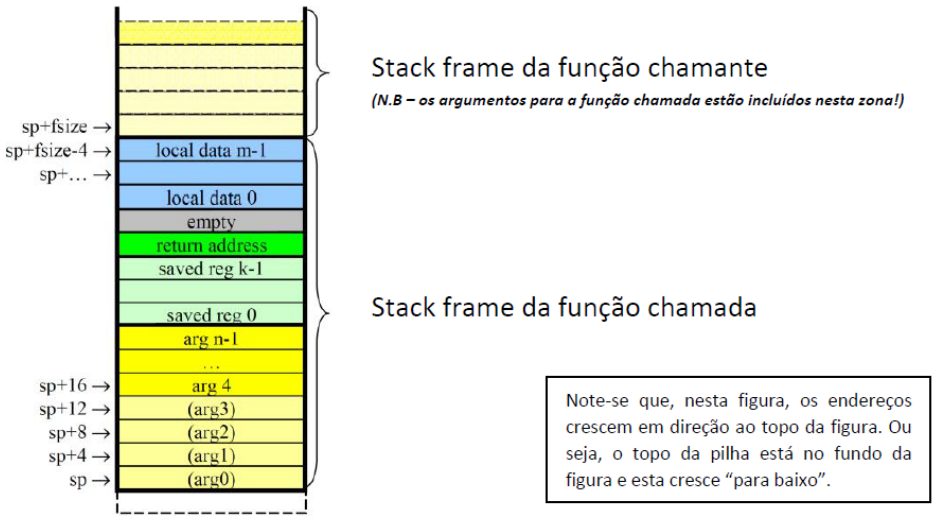
Lab 10
Funções, *Stack Frames* e Recursividade – Parte 2

***Objetivo*: Neste trabalho de laboratório pretende-se reforçar o estudo do funcionamento do mecanismo de chamada de funções e a utilização da pilha (*stack frames*), permitindo a correta interligação de funções em linguagem *assembly* e em C, bem como a instanciação independente de cada chamada a uma função que permite a ocorrência de recursividade.**

Sugere-se a releitura atenta da introdução do enunciado do Lab #9.

1. Passagem de mais do que quatro parâmetros

A passagem de parâmetros para uma função pode ter mais do que os 4 parâmetros passados através dos registos `$a0` a `$a3`. Quando tal acontece, os restantes parâmetros têm de ser passados pela pilha na ordem inversa à do cabeçalho (veja a figura seguinte).



Assim, ao chamar uma função que tem mais do que 4 parâmetros de entrada é necessário alocar espaço na pilha para os parâmetros a mais, chamar a função e, quando esta retornar, é necessário limpar a pilha dos parâmetros adicionais.

Pretende-se então implementar um programa que permita determinar o valor dado pela expressão:

$$f = 5(x_1 + x_2)(x_3 - 2x_4x_5)$$

Escreva um programa em linguagem *assembly* que leia uma série de parâmetros, x_1 a x_5, de um vetor armazenado em memória, e chame uma função também em *assembly*, `PolyCalc()`. A passagem dos parâmetros para esta última função, cujo propósito é calcular e devolver o valor de *f*, é feita, por sua vez, ***por valor***. Como anteriormente, e porque desenvolver um programa inteiramente em *assembly* pode ser trabalhoso, invoque o programa a partir da função `main()` de um programa em C; a função `main()`, que deverá incluir a declaração do vetor, propriamente dito, após retornar da função em *assembly* terá também de imprimir no ecrã o valor de *f* e os valores dos elementos desse vetor.

Em resumo, as funções e os seus protótipos devem ser os seguintes (os nomes dos parâmetros foram omitidos):

```
main() >> int programa(int*) >> int PolyCalc(int,int,int,int,int)
```

2. Recursividade

Este trabalho pretende reproduzir as funções de recursividade implementadas no trabalho laboratorial Lab #4, onde foram estudadas e testadas as diferenças entre funções cíclicas ou iterativas e funções recursivas. O que se pretende neste trabalho é fazer as mesmas implementações em Assembly do MIPS, respeitando as convenções de chamada de funções e de registos.

2.1 Implemente uma função principal em C, no ficheiro lab10_2.c que calcule os valores da sequência de Fibonacci, através de uma função `fibonacci_cycle()`, implementada em Assembly do MIPS que usa um ciclo para obter a solução. O valor de n deve ser introduzido na linha de comandos. Para converter a string com o número indicado pelo utilizador num inteiro pode recorrer à função `atoi()`. Caso o utilizador não indique o número na linha de comando deve indicar como deve ser utilizado o programa fibonacci.

Repare que a função `fibonacci_cycle()` não chama nenhuma outra função, pelo que é uma função Leaf.

Exemplos de utilização:

```
aluno@mips:~$ ./fibonacci

usage: fibonacci numero

aluno@mips:~$ ./fibonacci 7
fibo(0) = 0
fibo(1) = 1
fibo(2) = 1
fibo(3) = 2
fibo(4) = 3
fibo(5) = 5
fibo(6) = 8
fibo(7) = 13

aluno@mips:~$
```

Teste a implementação das funções.

2.2 Implemente agora em Assembly do MIPS, no ficheiro **fibo_recursive.s**, uma implementação recursiva da sequência de Fibonacci em que a função `fibonacci_recursive()` se chama a si própria. Teste o programa em C com a chamada às duas implementações da função de Fibonacci, comparando os resultados.

Repare que a função `fibonacci_recursive()` chama-se a si própria, por isso é uma função Non-Leaf, ao mesmo tempo caller e callee, sendo necessário ter especial cuidado com a secção de argumentos, a criação do prólogo e do epílogo e a chamada da função.