

Lab 9

Funções, *Stack Frames* e Recursividade – Parte 1

Objetivo: Neste trabalho de laboratório pretende-se estudar o 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.

1. Introdução – *stack frames* e as convenções de chamada a funções em *assembly*

De forma resumida, aprendemos que a pilha é uma zona de memória utilizada para armazenar dados temporários, incluindo variáveis locais e passagem de parâmetros para funções. O programador precisa de ter o cuidado de, em cada função, deixar a pilha inalterada. Além disso, quando uma função chama outra função necessita de guardar o seu próprio endereço de retorno ($\$ra$) de forma a poder regressar ao código que a chamou.

A passagem de parâmetros faz-se através dos registos $\$a0$ a $\$a3$ ($\$4$ a $\$7$) e a devolução será feita pelos registos $\$v0$ e $\$v1$ ($\$2$ e $\$3$ – este último só é usado para resultados que exijam 64 *bits*).

Neste trabalho, iremos estudar mais a fundo como se faz a gestão da pilha em MIPS ao longo do decurso da execução de funções, e como é que se conjuga esta gestão com o uso de registos e passagem de parâmetros entre funções.

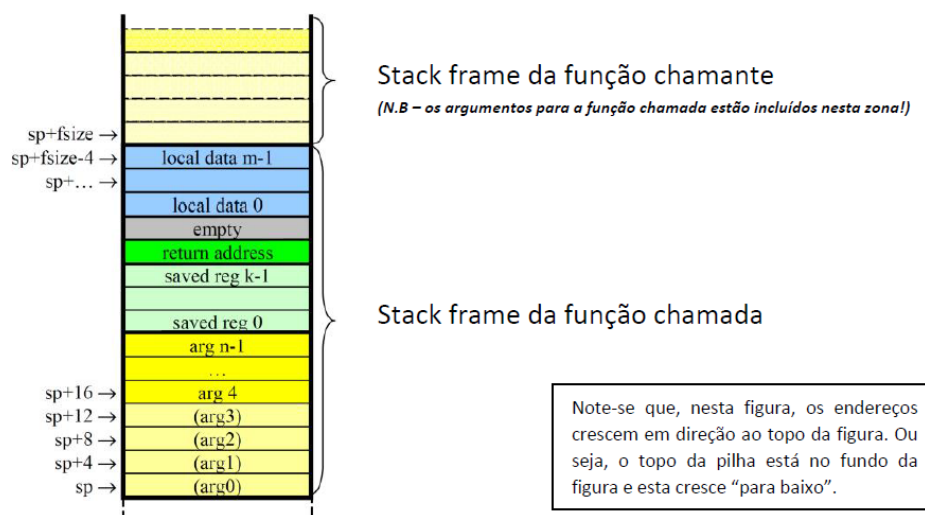
Infelizmente, ao contrário do que acontece no uso de registos, *não existe apenas uma convenção de gestão/uso da pilha – as convenções usadas dependem de compilador para compilador*. Existem, em qualquer caso, sempre pontos em comum entre as convenções efetivamente utilizadas. No caso de uma função típica, medianamente complexa (por exemplo, que utilize variáveis locais e chame outras funções com mais de 4 parâmetros), verificam-se os seguintes factos, qualquer que seja a convenção em consideração:

- De cada vez que uma função é chamada, um *stack frame* específico é criado para cada instância dessa função (ou seja, se a função for recursiva, existirá um *stack frame* criado por cada chamada à função).
- A função reserva espaço no *stack frame* para armazenar argumentos que precisa de passar a funções que eventualmente venha a chamar.
- A função reserva espaço no *stack frame* para salvaguardar os *saved registers* ($\$s0$ a $\$s7$) que venha a usar logo no início da sua execução (obrigação da função como

função que foi chamada por outra). Nota importante: se o frame pointer ($\$fp$) for usado, deve ser considerado como *saved register* (ver adiante).

- A função reserva espaço no *stack frame* para salvaguardar o endereço de retorno (contido no registo $\$ra$) no início da sua execução, se vier eventualmente a chamar outra função.
- A função reserva espaço no *stack frame* para armazenamento local.
- A organização do *stack frame* é importante, na medida em que (1) define a organização do armazenamento local (e por isso temporário) da função em causa; (2) forma um contrato entre uma função chamante e uma função chamada, que ambas precisam de ter em conta para poderem comunicar uma com a outra como parte daquilo a que se dá o nome de passagem de parâmetros.

Na figura¹ seguinte, ilustra-se a convenção de organização da pilha utilizada pelo compilador usado nas aulas práticas (exercício: confrontar esta convenção com aquela que se encontra descrita no livro). Note-se que, nesta convenção, o programador é obrigado a reservar espaço na pilha de forma a alinhá-la ao *double word* (ou seja, **usando um múltiplo de 8!**).



Como se pode ver na figura, em geral esta organização da pilha divide o stack frame em 5 partes distintas:

1. A secção de argumentos serve de interface (juntamente com os registos de passagem de argumentos) entre a função chamante e as suas funções chamadas. As primeiras quatro words desta secção não necessitam de ser usadas pela função chamante, e em código desenvolvido durante as aulas não vai ser necessário mais do que reservar o espaço correspondente – **no entanto, este procedimento de reserva de espaço é obrigatório, qualquer que seja o número de argumentos passados para a função chamada**. Em qualquer caso, quando gera código automaticamente o `gcc` efetua cópias dos registos de argumentos $\$a0$ a $\$a3$ para possível uso da função chamada.

¹ Figura adaptada de "MIPS Calling Convention", ECE314 Spring 2006, CSE 410 Computer Systems course, University of Washington, também anexo ao trabalho.

Quando há mais de quatro argumentos, reserva-se a quantidade de espaço necessária para reservar cada um deles começando a partir de $\$sp+16$, em conformidade com a convenção típica de passagem de parâmetros.

2. A secção de *saved registers* contém o espaço para salvaguardar os valores dos registos $\$s0$ a $\$s7$ que venham a ser utilizados ao longo da execução da função (responsabilidade da função quando é chamada).
3. A secção de *return address*.
4. Uma secção de “enchimento” que não armazena nada de útil, servindo apenas para acomodar o caso de se terem dados que não preencham o alinhamento ao *double word* por completo (**no máximo**, será necessário um espaço vazio, dado que 8 é um múltiplo de 4).
5. Uma secção de armazenamento local, que servirá para armazenar estruturas de dados que correspondem à noção de variáveis locais em C (sob forma de valores simples, *arrays*, etc.) e registos temporários $\$t0$ a $\$t9$ que venham a ser utilizados e que a função chamante queira ver preservados sempre que chame uma sub-função (responsabilidade da função quando é função chamante).

É, ainda, de notar que a construção de um stack frame para uma função é sempre desenhada pelo programador em função das necessidades da função a construir. Alguns exemplos das escolhas que o programador deve considerar incluem:

- As funções designam-se por funções **Leaf**, ou folhas, se forem funções que não chamam outras funções. Nesse caso, as suas stack frames não necessitam obrigatoriamente de:
 - ter espaço para o return address $\$ra$, porque este nunca vai sofrer alterações no decorrer da execução desta função
 - ter espaço para uma secção de argumentos, porque como nunca vão chamar outras funções, este espaço nunca será usado
- Ainda em relação às funções **Leaf**, estas funções mantêm a obrigatoriedade de respeitar a convenção dos registos, pelo que no caso de usarem os registos *saved*, necessitam de os salvaguardar primeiro da stack frame e, antes de terminarem, necessitam de os recuperar.
- Assim, se uma função **Leaf** não alterar os registos *saved* e não tiver variáveis locais (necessariamente guardadas no stack frame), então estas funções não necessitam sequer de criar um stack frame e, conseqüentemente, de ter um Prólogo e um Epílogo.
- As funções designam-se por funções **Non-Leaf**, ou seja, não-folhas, se forem funções que chamam outras funções. Neste caso, o desenho do stack frame é obrigatório podendo, ainda assim, só ter as secções de que a função necessite.

Usando esta convenção, é prática de boa programação desenvolver funções que respeitem a seguinte estrutura, para funções **Non-Leaf**, isto é, funções que chamam outras funções:

```
# **** Secção de Argumentos ****

# **** guarda na stack frame da função chamante o(s) argumento(s) de entrada da
# função - só necessário se a função quiser guardar os seus argumentos antes de
# chamar outras funções, para qualquer número de argumentos ****

sw $a0, 0($sp)      # copia $a0 para o stack frame da caller (1st arg)
sw $a1, 4($sp)      # copia $a1 para o stack frame da caller (2nd arg)
...
sw $a3, 12($sp)     # copia $a3 para o stack frame da caller (4th arg)

# **** Prólogo ****
# (contém tudo o que é obrigação de uma função chamada, quando é
# chamada)

addiu $sp, $sp, -8x  # reserva de stack frame salvaguarda na pilha do
...                 # que for necessário ($s0-$s7 e $fp se for usado,
# $ra). Nota: 8x significa um múltiplo de 8 bytes

move $fp, $sp       # opcional (só necessário se $sp mudar)!

# **** Corpo do código ****
...

# ** chamada a função **
# -- passagem de parâmetros (4 primeiros por $a0-$a3, restantes
# por pilha)
...

# -- chamada propriamente dita
jal ...

# -- recuperação de valores retornados ($v0-$v1)
...
# (código continua)
...

# **** Epílogo ****
# (contém tudo o que é obrigação de uma função chamada,
# quando retorna - "espelho invertido" do prólogo)

move $sp, $fp       # opcional (ver acima)!
...                 # restauro da pilha do que for necessário
# ($s0-$s7 e $fp se for usado, $ra)
addiu $sp, $sp, 8x   # libertação de stack frame

# **** Retorno ****
jr $ra
```

Note-se que o registo `$fp` (*frame pointer*), que serve para apontar de forma estável a fronteira do *stack frame*, só necessita de ser usado se se modificar o `$sp` ao longo do corpo do código da função. Obviamente, se `$sp` for modificado, todas as posições relativas dos valores armazenados na pilha mudariam face a `$sp`, o que seria inconveniente para o programador – usando `$fp` como ponteiro auxiliar, este inconveniente é contornado.

Nota: Sugere-se a consulta do ficheiro anexo a este enunciado com o título *MIPSCallingConventionsSummary.pdf*, que também pode ser encontrado neste link: <https://courses.cs.washington.edu/courses/cse410/09sp/examples/MIPSCallingConventions>

[Summary.pdf](#). Neste ficheiro pode estudar as diversas convenções de chamada no MIPS na versão que é usada pelo compilador que usamos nas aulas laboratoriais – `gcc`.

Conselho final:

- Analisar sempre que possível o código *assembly* gerado automaticamente pelo compilador para aprender como se faz tudo o que foi descrito acima (no Trabalho Laboratorial #8, no segundo problema, há uma oportunidade excelente para este exercício de engenharia inversa).

Exemplo: anexo a este enunciado, está disponível um exemplo de implementação de uma função em C que chama uma função **Non-Leaf** em Assembly do MIPS (`inside_circle`) que, por sua vez, chama outra função **Leaf** em Assembly (`squared`). O objetivo é testar se um ponto do plano XY está dentro ou fora de um círculo.

1. Chamada de funções assembly em C

Neste primeiro exercício pretende-se implementar duas funções escritas em *assembly* e que permitam devolver o máximo e o mínimo valor de entre quatro valores passados como parâmetros. Estas funções deverão ser chamadas a partir de um programa em C que peça ao utilizador para introduzir um conjunto de 4 valores e que escreva no ecrã o valor máximo e o valor mínimo introduzido. Utilize os ficheiros fornecidos com o enunciado do trabalho como ponto de partida para implementar estas funções. Encontre os valores mínimos e máximos para as funções `min` e `max`, respetivamente, recorrendo a comparações sucessivas.

2. Passagem de referências da memória

Pretende-se novamente implementar as funções `min` e `max` mencionadas no ponto anterior, mas desta vez assumindo que os parâmetros de entrada de cada uma destas funções passam a ser uma tabela de inteiros e o número de elementos que essa tabela contém. Algo do género: `int max(int *tabela, int numValores)`.

Tal como nos exemplos anteriores, assuma a existência de um programa escrito em C onde a tabela é inicializada e que chama as duas funções, apresentando depois o resultado. Utilize os ficheiros fornecidos com o enunciado do trabalho como ponto de partida para implementar estas funções.

3. Funções Non-Leaf

Voltando a implementar as funções `min` e `max`, encontre os valores mínimos e máximos para as funções `min` e `max`, respetivamente, recorrendo a duas funções adicionais que deve escrever: `min2` e `max2`, que calculam o mínimo ou o máximo entre apenas dois valores de entrada. Repare que esta implementação obriga a usar as convenções de chamada das funções no *assembly* do MIPS, quer do lado do callee, quer do lado do caller.