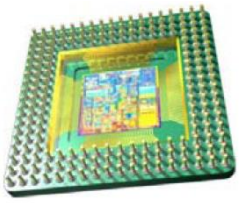


Arquitectura de Computadores

LIC. EM ENGª INFORMÁTICA
FACULDADE DE CIÊNCIA E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Lab 8 – Funções no MIPS e Interligação com o C – Passagem de Parâmetros

Neste trabalho de laboratório pretende-se ver como funciona o mecanismo de chamada de funções e compreender a utilização da pilha (*stack frames*) para passar parâmetros e salvaguardar o contexto, permitindo a correta interligação de funções implementadas em linguagem *assembly* com programas escritos em C.

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

A pilha é uma zona de memória utilizada para armazenar dados temporários, incluindo variáveis locais e passagem de parâmetros para funções. A gestão da pilha é da responsabilidade do programador, que precisa de ter o cuidado de, em cada função, deixar a pilha inalterada. A pilha serve para armazenar temporariamente um conjunto de informações importantes para a execução do programa. Por exemplo, 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.

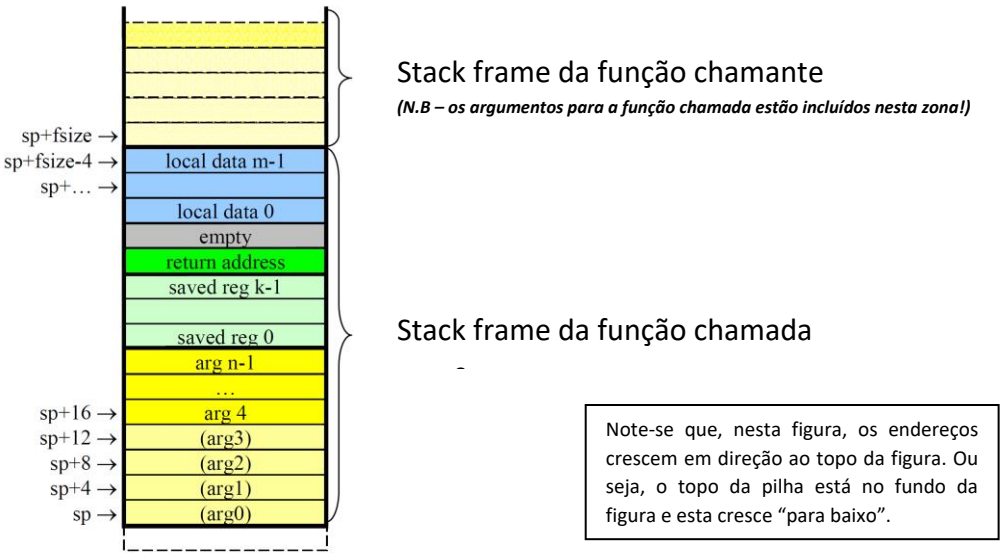
Infelizmente, ao contrário do que acontece no uso de registos, *não existe apenas uma convenção de gestão/uso da pilha* – as convenções usadas dependem de compilador para compilador. Existem, em qualquer caso, sempre pontos em comum entre as convenções efetivamente utilizadas. No caso de uma função típica, 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 considerada:

- 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 figura seguinte[1], 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, apresentada no Trabalho Laboratorial #6). 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, que 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 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`

[1] Figura adaptada de *"MIPS Calling Convention", ECE314 Spring 2006, CSE 410 Computer Systems course, University of Washington*.

para possível uso da função chamada. 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 preenchem 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 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).

Usando esta convenção, é boa prática de programação desenvolver funções que respeitem a seguinte estrutura:

```
# **** 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)
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.

Conselho final:

- Analisar sempre que possível o código *assembly* gerado automaticamente pelo compilador para verificar como se faz tudo o que foi descrito acima.

2. 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.

3. 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. 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:

$$f = 5(x_1 + x_2)(x_3 - 3x_4x_5)$$

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 *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)
```

4. Manipulação de *strings*

Escreva uma função em linguagem Assembly do MIPS (`conta_char()`) que é chamada na função `main()` escrita em linguagem C e cujo código é fornecido. A função `conta_char()` recebe como parâmetros de entrada um ponteiro para a string e o caracter a pesquisar na string, e deverá retornar o número de vezes que o caracter recebido aparece na string.

Nota: Pode encontrar em anexo o ficheiro `main.c`.

5. Ponto 2 Revisitado

Pretende-se novamente implementar as funções *min* e *max* mencionadas no ponto 2, 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.