Laboratório 4 Subrotinas

Salto e Retorno

Após uma função ser encerrada, precisamos retornar ao ponto de onde ela foi chamada dentro do programa. Para implementarmos funções em *assembly* (chamadas subrotinas), precisamos retornar ao ponto de chamada, que vai ser diferente a cada vez que a subrotina for usada. A estratégia do MIPS é armazenar este endereço no registrador $ra (*return address*). Veja:

```
main:  jal nada # chama a subrotina; "volta" é guardado em $ra
volta: li $s0,0
       jal nada # chama "nada"; vai armazenar "fim" em $ra
fim:   break    # saída deselegante; use syscall 10 ao invés disso
nada:  li $s0,3
       jr $ra   # pula para o endereço dado por $ra (volta ou fim)
```

A instrução `jal` (*jump-and-link*) é um `j` modificado que guarda automaticamente o valor de PC+4 no $ra (endereço da próxima instrução: é o ponto de retorno da chamada). As subrotinas devem portanto terminar com `jr $ra` para saltar *explicitamente* para o ponto de retorno da chamada.

Parâmetros

Para usarmos parâmetros e retornarmos resultados das funções, usamos os registradores padrão $a0~$a3 para os argumentos, nesta ordem, e $v0 para o resultado (com $v1 se o resultado for de 64 bits).

```
main: li $a0,3
      li $a1,2           # carrega parâmetros (3,2) e chama função
      jal prod
      move $s0,$v0       # copia o resultado de $v0 para $s0
      break
prod: mulo $v0,$a0,$a1   # apenas multiplica os parâmetros
      jr $ra             # retorna; resultado (6) em $v0
```

Note que isso é apenas uma convenção, e apesar de ser uma péssima ideia desobedecê-la, isso é possível. Há um compilador C para o MIPS que gera código sem diferenciar os registradores: usa os disponíveis conforme precisa (embora não possa usar o $0 nem o $31=$ra, além dos $k, usados pelo sistema operacional).

▶ Construa um programa que obtém um valor *válido* de temperatura pelo teclado, o converte de Celsius para Fahrenheit e o imprime, usando no mínimo duas funções além da *main*. *Utilize obrigatoriamente as convenções descritas anteriormente.*

Pilha

A pilha é uma área de memória destinada a guardar dados temporários. É uma estrutura LIFO (*Last In First Out*) que costuma ser usada por processadores para armazenar variáveis locais (*storage class* automática) e endereços de retorno de funções. O endereço do topo da pilha na memória é dado pelo registrador $sp (*Stack Pointer*). As operações são chamadas *push* (guardar dado) e *pop* (recuperar dado), e podem ser vistas na próxima seção .

As funções convencionadas no MIPS assumem pilha aumentando em direção ao endereço 0x0000 0000 e $sp apontando para o último elemento que foi empilhado. (A pilha poderia crescer em direção ao fim da memória e $sp poderia apontar para o próximo espaço a ser utilizado).

Funções que Chamam Outras Funções

Se uma função chama outra, teremos *dois* endereços de retorno a armazenar, um para cada chamada, mas apenas um $ra. A saída é preservar temporariamente o primeiro endereço de

retorno em algum lugar, usualmente a pilha. Veja uma função *func1* que chama outra *func2*:

```
func1: addi $sp,$sp,-4 # reserva espaço na pilha para um dado
       sw $ra,0($sp)   # guarda o $ra original no topo da pilha
       jal func2       # isso vai destruir o valor original de $ra!
       lw $ra,0($sp)   # recupera o valor original da pilha
       addi $sp,$sp,4  # libera o espaço reservado para o $ra
       jr $ra          # retorna para o endereço correto
```

Isso resolve um problema; mas e os outros registradores? Suponha agora que func1 vá precisar de 3 registradores e func2 de outros 4. Como garantir que func2 não vai destruir nada importante? A solução é uma convenção simples: *uma função pode destruir os valores de $t's (temporários) e $a's, mas não pode destruir mais nenhum, em especial os $s's (salvos).*

A regra é: a função chamadora empilha/recupera quaisquer $t's, $a's ou $v's que precisem ser mantidos após a chamada; a função chamada empilha/recupera $ra e os $s's que destruir.

▶	Construa um programa que calcula $y=3x^5+2x^3-6x$ para um valor de x entrado pelo teclado. A *main* deverá pegar o valor, chamar uma função que calcula y e imprimir o resultado. A função que calcula y deverá obrigatoriamente chamar uma função *pow*, que eleva um número $a0 a outro $a1. *Utilize obrigatoriamente as convenções descritas anteriormente.*

Register Spilling e Registradores na Pilha

Temos portanto poucos registradores disponíveis, mas eles são suficientes para o caso comum (pense em quantas variáveis locais costumam ser usadas numa função). Nos casos onde faltam registradores, precisamos salvar valores temporariamente na memória, especificamente na pilha. A isso se chama *register spilling*.

No caso especial de uma função precisar de mais de 4 parâmetros, os demais são empilhados em ordem logo antes da chamada à função, que deverá acessá-los lá dentro da pilha. A convenção da ordem pode ser C/Windows (começa com parâmetros mais à esquerda, progredindo para a direita) ou Pascal (direita para esquerda). Os argumentos devem ser removidos da pilha pela função chamadora (C) ou pela função chamada (Windows/Pascal).

▶	Utilizando uma função hipotética já pronta chamada *cemseno* que calcula o valor de 100*seno($a0), que infelizmente destrói todos os $t e $a no processo, faça uma subrotina que devolve como resultado $a0+cemseno(3*$a1)-cemseno(3*$a0)+$a2*$a1. *Utilize obrigatoriamente as convenções descritas anteriormente.* **Entregar semana que vem.**

▶	Construa e teste uma função que retorna como resultado a soma de oito valores passados a ela como parâmetros. Utilize a convenção do C. *Utilize obrigatoriamente as convenções descritas anteriormente.*

Exercícios (atenção: a correção destes vai ser mais rigorosa)

1. Há um comando chamado *nop* (*no operation*) no *assembly* MIPS, que não faz nada. Ele está presente na maioria dos microprocessadores. Para que ele serve?
2. Note que alguns processadores sempre salvam o endereço de retorno na pilha, mesmo que a subrotina não chame nenhuma outra (seja uma "folha"). Argumente a favor disso.
3. **[ENTREGAR SEMANA QUE VEM!]** Faça e teste uma função *recursiva* que calcula o fatorial de um número, *obedecendo às convenções descritas*. Sugiro tentar entregar antes.
4. O x86 sempre teve poucos registradores de uso geral. Em que circunstâncias isso é desejável, ao invés de termos muitos deles? Por que não podemos ter centenas deles?
5. A família de microcontroladores do 8051 *não possui* registradores explícitos, usando ao invés deles referências à memória RAM interna do processador (128 ou 256 bytes de base). Isso é obviamente razoável (já que ainda se vendem 8051's). Por quê?