Laboratório 4 Subrotinas

Funções em *assembly* são usualmente chamadas de subrotinas e possuem várias convenções para interoperabilidade, já que estamos mexendo diretamente com os detalhes do processador neste nível.

Salto e Retorno

Quando uma função é encerrada, ele deve voltar ao ponto em que foi chamada. Mas este ponto pode variar, já que a função pode ser chamada diversas vezes, de diversos lugares do programa.

Como saber qual o endereço de retorno? 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 $t0,3
       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 instrução seguinte ao `jal`: 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). Veja a subrotina "prod", que tem como resultado o produto de seus dois parâmetros.

```
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: mul $v0,$a0,$a1    # apenas multiplica os parâmetros
      jr $ra             # retorna; resultado (6) em $v0
```

Note que isso é apenas uma convenção[1], 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 auxiliares. *Utilize obrigatoriamente as convenções descritas anteriormente.* Use números inteiros.

1 Estas convenções fazem parte de uma ABI – *Application Binary Interface*. Uma ABI define várias coisas, entre elas o comportamento padrão de funções, para que os *linkers* (linkeditores) consigam ligar módulos de diferentes linguagens num mesmo programa-alvo. Poderíamos, por exemplo, usar um módulo em C, outro em *assembly* e um terceiro em Java, compilando tudo para um único executável.

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 (variáveis com *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 que a pilha aumenta em direção ao endereço 0x0000 0000 (ou seja, $sp vai diminuindo) e $sp aponta para o último elemento que foi empilhado. Em outras convenções, a pilha pode crescer em direção ao fim da memória e $sp poder apontar para o próximo espaço a ser utilizado, por exemplo.

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, convencionalmente na pilha. Veja uma função *func1* que apenas chama corretamente outra função *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 valor original guardado na 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 nenhum outro, 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).

Resumo das Convenções

Seguem novamente em formato de lista, para clareza.

- Uma função pode usar apenas os **registradores $t0~$t9**, além de $a0~$a3 e $v0
- Se a função precisar utilizar os outros registradores, em especial os **$s0~$s9**, eles deverão ser **salvos na pilha** no começo da função e ter seus valores recuperados da pilha ao final da função
- A função só pode usar valores passados a ela como **parâmetros** pelos registradores **$a0 a $a3**, nesta ordem. Ou seja, uma função *volume(altura, largura, profundidade)* não pode passar a altura por $a3, largura por $a1 e deixar a profundidade em $t0: deverá usar $a0, $a1 e $a2. Se houver mais de quatro parâmetros, eles são passados através da pilha.
- Se a função produz um **resultado** a retornar, deverá obrigatoriamente usar **$v0** para devolvê-lo à chamadora.
- Se a função chama outra função, deverá guardar o valor de **$ra na pilha** no seu início e recuperá-lo da pilha ao final. Se não chama outra função (função folha), isso é desnecessário.
- Se a função chama outra função, deverá, antes da chamada, **guardar na pilha** o valor de **todos os registradores que utilizar** e recuperá-los logo depois, pois teoricamente a função chamada pode destruir os valores. *Isso inclui os $t's, $a's e $v's!*
  Por exemplo: se a função *matriz* usa $t0, $t2 e $a0 e chama a função *varredura*, os valores de $ra, $t0, $t2 e $a0 deverão ser empilhado antes do *jal varredura* e desempilhados depois.

▶	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.*

▶	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.*

Números em Ponto Flutuante[2]

Os números em **ponto flutuante** são representados em binário e quebrados em sinal, mantissa e expoente. Por exemplo, π em binário é: $3{,}141593_{10} \simeq 11{,}001001000_2$ pois $2^1+2^0+2^{-3}+2^{-6} = 2+1+0{,}125+0{,}015625 = 3{,}140625_{10}$ (perceba a aproximação provocada pelo erro de precisão).

Vamos representar π em *float* usando, como exemplo, 16 bits: $\pi = 1{,}1001001000_2*2^1$ (notação científica binária, uma ideia similar à do formato decimal, parecido com representar a velocidade da luz com $2{,}9979*10^8$ m/s). O primeiro 1 (o MSB) sempre fica implícito, pois ele sempre existe (certo?)[3], então os dados são 1001001000. Como o expoente é 1, temos:

sinal	expoente	mantissa
0 (positivo)	00001	1001001000

Outro exemplo: os dígitos binários *1*11110*0010001000* (sinal 1, expoente 11110, mantissa em itálico 0010001000) significam, neste *float* de 16 bits, $-1{,}0010001000_2*2^{-2} = -0{,}010010001000_2 = 0{,}25+0{,}03125+0{,}001953125 = 0{,}283203125_{10}$.

Para conversão de decimal para ponto flutuante, o método mais simples é o seguinte: primeiro convertemos a parte inteira, depois vamos multiplicando a parte fracionária por dois, tomando o primeiro dígito. Por exemplo, para π=3.141593, temos 3_2=11, inicialmente. Então:

2 O livro-texto possui explicações detalhadas deste tópico.

3 Neste padrão, o *expoente zero* com mantissa zero é usado para representar o número 0.0. É um caso especial.

0.141592*2=**0.**283184 retiramos o 0
0.283184*2=**0.**566368 retiramos o 0
0.566368*2=**1.**132736 retiramos o 1
0.132736*2=**0.**265472 retiramos o 0
0.265472*2=**0.**530944 retiramos o 0
0.530944*2=**1.**061888 retiramos o 1
0.061888*2=**0.**123776 retiramos o 0 e assim por diante.

Montando em ordem, finalmente temos $\pi=11.0010010_2$. Alternativamente, podemos ir calculando na mão o peso de cada bit e testar a adição (ex.: o primeiro bit vale $2^{-1}=0.5$, então $11.1_2=3.5$ que é maior que π; portanto este bit deve ser zerado, e assim por diante).

A respeito deste formato especificado acima:

1. Converta 123,456 e -0,000456 para o formato e 010100010001 para representação decimal.
2. Quais os valores máximo e mínimo representáveis? Quantos dígitos decimais temos de precisão?
3. Descreva um circuito somador e um subtrator para números em ponto flutuante.
4. Descreva um circuito multiplicador em ponto flutuante. Estime o número de clocks gastos.
5. Se tivermos x=123456000 e y=0,000987, na representação de 12 bits apresentada, qual será o resultado do cálculo não otimizado x+y-x? (O ponto é: *operações com floats não são comutativas!*)

O formato padrão *de facto* é o IEEE 754 que especifica o *binary16* ou *half precision,* muito similar ao descrito acima. Há poucas diferenças: valores especiais (infinito e NAN – *Not a Number*), valores não normalizados e, em especial, a representação do expoente em excesso *(bias* de 15 no *float),* para facilitar comparações.

Exercícios

1. Faça e teste uma função *recursiva* que calcula o fatorial de um número, *obedecendo às convenções descritas*. Sugiro me perguntar se está correta.
2. 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?
3. 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.
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?

Resposta Detalhada do Exercício 4 do Laboratório #3

No 8051, é o programador que decide se o número de 8 bits é sinalizado ou não. Dada a conta: $11001110_2+11000111_2$ com resultado num registrador de 8 bits, quais são os valores decimais envolvidos? Houve estouro da capacidade de 8 bits ou não?

Chamemos o valor 11001110_2 de x e o valor 11000111_2 de y. Temos quatro possibilidades interpretativas, usando notação da linguagem C:

tipo	x	y
signed de 8 bits	-50	-57
unsigned de 8 bits	206	199

O resultado final da soma será 10010101_2, independentemente de haver sinal ou não, pois a ULA faz as somas sem saber se o número é binário puro ou complemento de 2, já que a operação

matemática é a mesma em ambos os casos. Este valor tanto pode ser 149, caso o resultado seja interpretado como *unsigned*, como pode ser -107, se for *signed*.

Perceba que o resultado está correto em todas as possibilidades, exceto quando tanto x quanto y são *unsigned:*

1. -50-57=-107
2. -50+199=149
3. 206-57=149
4. 206+199=405, e 405=256+149 (o 256 é o *carry*)

Isso ocorre porque as contas de 8 bits são feitas em *módulo 256*.

Observe que não há *overflow* e há *carry* nesta conta $11001110_2+11000111_2$. Ademais, o *overflow* só vale para o caso 1, sinalizado, e o *carry* só deve ser usado no caso 4, não sinalizado. Os casos 2 e 3 são "perigosos" para o programador, misturando representações sinalizada e não sinalizada, mas dão o resultado certo.

Portanto só há estouro da capacidade de 8 bits se estivermos usando números não sinalizados; caso contrário, não há estouro.