

Introdução à Arquitetura e Organização de Computadores

Professor: Diego de Assis Monteiro Fernandes

Aula 5 - Instruções: Parte 2

1 Chamada de Procedimentos

Um **procedimento**, também chamado de subrotina ou função, é um modo de permitir que o programador estruture seus programas, tanto para torná-lo mais fácil de compreender como para permitir seu reuso. Procedimentos permitem que o programador isole uma parte da tarefa e tente resolvê-la separadamente. A interação entre um procedimento e o restante do programa é realizada através da passagem de parâmetros, que atuam como uma barreira, permitindo a passagem de valores e o retorno de resultados.

Invocar um procedimento em C++ ou Pascal é, em geral, uma tarefa simples. Entretanto, esse código, quando transformado em linguagem de máquina, desempenha algumas tarefas que ficam ocultas ao programador. Essas tarefas são importantes pois controlam a execução o fluxo de execução, ou seja, direcionam qual as instruções a serem executadas. Além disso, também reservam os recursos necessários para a execução do procedimento. O programa, para executar um procedimento, deve seguir os seguintes passos:

1. Armazenar os parâmetros em algum dispositivo de armazenamento onde o procedimento possa acessá-los.
2. Transferir o controle para o procedimento, ou seja, desviar o fluxo de execução para as instruções referentes ao procedimento.
3. Obter os recursos de armazenamento necessários para o procedimento.
4. Realizar a tarefa desejada.
5. Armazenar os resultados em algum dispositivo de armazenamento onde o programa possa acessá-los.
6. Retornar o controle para o ponto de origem.

A arquitetura MIPS reserva os alguns registradores para a chamada de procedimentos. Esse registradores são utilizados para a passagem de parâmetros, para armazenar os resultados produzidos e para indicar para qual parte do programa retornar ao fim do procedimento. Os registradores são os seguintes::

- **\$a0 - \$a3** : quatro registradores de argumento para a passagem de parâmetros.
- **\$v0 - \$v1** : dois registradores de valor para o retorno dos resultados.
- **\$ra** : um registrador de endereço de retorno para retornar ao ponto de origem.

Além dos registradores, a linguagem de montagem MIPS inclui uma instrução reservada para a chamada de procedimentos. Essa instrução realiza as seguintes operações: desvia a execução do programa para outro endereço e simultaneamente salva, no registrador \$ra, o endereço instrução seguinte que seria executada caso não houvesse o desvio. Esse endereço, armazenado no registrador \$ra é denominado **endereço de retorno**. A instrução descrita é representada pela abreviação `jal` (*jump-and-link*) e é invocada da seguinte forma:

jal EndereçoDoProcedimento

Quando um determinado programa está executando, existe um registrador chamado **PC** (*program counter*) que armazena o endereço da instrução correntemente em execução. É através do endereço contido nesse registrador que as instruções são buscadas na memória para serem executadas. Normalmente (sem desvios de fluxo), a próxima instrução a ser executada é a que está contígua na memória com a instrução corrente. Para buscá-la, o registrador PC deve então ser incrementado em 4 (tamanho em bytes de qualquer instrução MIPS, 32 bits).

Quando um procedimento é chamado, a arquitetura deve ser capaz de, ao seu término, retornar a execução para seu fluxo anterior. Para isso, a instrução **jal** armazena o valor $PC + 4$ no registrador **\$ra**, permitindo que o fluxo possa ser retomado corretamente ao fim do procedimento. Desse modo, a última instrução dentro do procedimento deve ter como função a retomada do fluxo original do programa. Essa instrução é a **jr** (*jump register*) que é chamada da seguinte forma:

jr \$ra

O código em linguagem de montagem a seguir exemplifica um caso onde um procedimento, que simplesmente retorna o argumento multiplicado por 4, é chamado:

```
main:                #função principal
...
    add $a0, $s0, $zero    #armazena o valor de $s0 em $a0 (registrador de argumento)
    jal funct             #invoca o procedimento
    add $s0, $v0, $zero    #armazena o resultado do procedimento em $s0
...

funct:              #início do procedimento
    add $v0, $a0, $a0      #faz $v0 = 2*$a0
    add $v0, $v0, $v0      #faz $v0 = 4*$a0
    j $ra                 #retorna ao fluxo de execução anterior
```

1.1 Utilizando mais registradores

Suponha que um compilador necessite de mais registradores para a execução de um procedimento do que os quatro de argumento e os dois de retorno. Uma vez que a execução do programa vai ser retomada, todos os registradores utilizados durante o procedimento devem ter seu valor restaurado. Nesse caso, os valores dos registradores são armazenados na memória, esse processo de transferência é chamado *spill*. A estrutura de dados ideal para esse tipo de operação é uma **pilha**. Sendo assim uma pilha é utilizada na arquitetura MIPS e, para acessá-la, outro de seus registradores é reservado: o registrador **\$sp** (*stack pointer*), que armazena o endereço do topo da pilha. Por razões históricas, um pilha aumenta de tamanho na memória do endereço mais alto para o mais baixo, ou seja, para inserir um elemento na pilha o registrador **\$sp** deve ter seu valor subtraído e para remover um elemento da pilha **\$sp** deve ter seu valor adicionado. O exemplo a seguir transforma um procedimento escrito em C para a linguagem de montagem MIPS.

Exemplo

Qual é o código MIPS para:

```
int leaf_example(int g, int h, int i, int j)
{
    int f;
    f = (g+h) - (i+j);
    return f;
}
```

Assuma que é possível utilizar constantes nas instruções aritméticas. Os parâmetros g, h, i e j correspondem respectivamente aos registradores \$a0, \$a1, \$a2, e \$a3, e f corresponde à \$s0. O procedimento inicia com seu rótulo:

```
leaf_example:
```

O próximo passo é salvar na pilha os registradores usados pelo procedimento (\$s0, \$t0 e \$t1).

```
sub $sp,$sp,12 #ajusta a pilha para acrescentar 3 elementos
sw $t1, 8($sp) #salva $t1
sw $t0, 4($sp) #salva $t0
sw $s0, 0($sp) #salva $s0
```

As próximas instruções correspondem ao corpo do procedimento:

```
add $t0,$a0,$a1
add $t1,$a2,$a3
sub $s0,$t0,$t1
```

Para retornar o valor de f, basta copiá-lo para um dos registradores de valor:

```
add $v0,$s0,$zero
```

Antes de retornar, é preciso restaurar o valor dos registradores que foram salvos e ajustar o topo da pilha para sua posição original:

```
lw $s0, 0($sp)
lw $t0, 4($sp)
lw $t1, 8($sp)
add $sp,$sp,12
```

O procedimento termina com o desvio para o fluxo original.

```
jr $ra
```

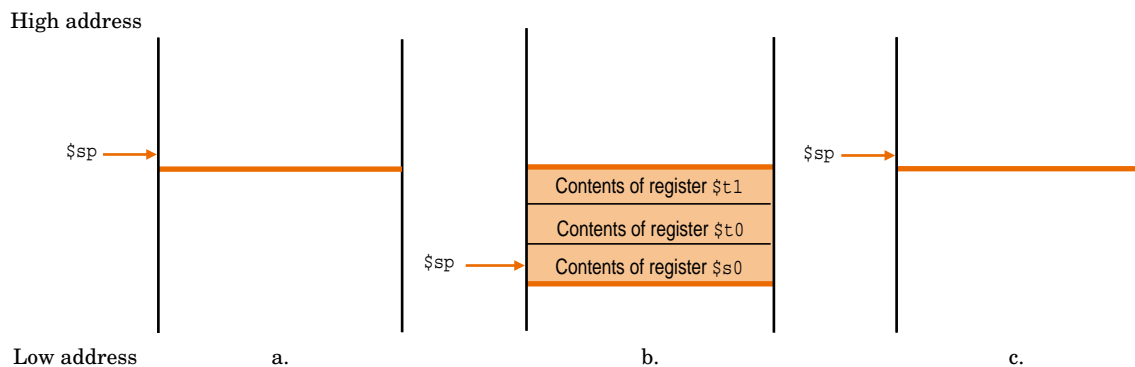


Figura 1: Estado da pilha antes, durante e depois da chamada do procedimento leaf example.

A Figura 1 mostra a pilha antes, durante e depois da chamada do procedimento. No exemplo, foram utilizados registradores temporários (os registradores \$t0 e \$t1) e assumido que seus valores devem ser salvos e restaurados. Para evitar salvar e restaurar registradores cujos valores podem não estar sendo usados, como os registradores temporários, MIPS oferece duas classes de registradores:

- \$t0 - \$t9 : 10 registradores temporários que não são preservados em uma chamada de procedimento.
- \$s0 - \$s7 : 8 registradores que devem ser preservados em uma chamada de procedimento, ou seja, registradores que devem ser salvos na pilha no início de um procedimento.

Essa é uma convenção criada pelos projetistas da arquitetura com o intuito de evitar operações desnecessárias. Retornando ao exemplo, com essa convenção, somente o registrador \$s0 deveria ser salvo, dispensando os registradores \$t0 e \$t1.

1.2 Procedimentos encadeados

Um procedimento que não invoca outros procedimentos em seu corpo é chamado **procedimento folha**. Porém, nem todos os procedimentos são folhas, uma vez que um procedimento pode invocar outros procedimentos, ou, no caso de recursão, pode invocar um “clone” seu. Nesses casos, os registradores de passagem de parâmetros e retorno de resultado também podem ser sobrescritos, causando conflito na execução do programa. Uma solução é empilhar todos os registradores que devem ser preservados. O exemplo a seguir converte um procedimento recursivo que calcula o fatorial de um número para a linguagem de montagem onde torna-se necessário armazenar os registradores descritos na pilha.

Exemplo:

Qual é o código MIPS para o seguinte procedimento recursivo:

```
int fact (int n)
{
    if(n < 1)
        return 1;
    else
        return (n * fact(n-1));
}
```

O parâmetro `n` corresponde ao registrador de argumento `$a0`. O procedimento inicia com o rótulo e então salva os dois registradores na pilha, o endereço de retorno e `$a0`:

```
fact:
    sub $sp,$sp,8
    sw $ra, 4($sp)
    sw $a0, 0($sp)
```

As próximas instruções testam se `n` é menor do que 1, indo para `L1` se $n \geq 1$:

```
slt $t0, $a0, 1 #testa se n é menor do que 1
beq $t0, $zero,L1 #se n é maior do que 1, então vai para L1
```

Se `n` é menor do que 1, `fact` retorna 1 (inserindo 1 no registrador de valor). A seguir, o topo da pilha é ajustado (os valores não foram recuperados porque não foram alterados) e a execução é desviada para o endereço de retorno:

```
add $v0,$zero,1
add $sp,$sp,8
jr $ra
```

Se `n` não é menor do que 1, o argumento `n` é decrementado e então `fact` é invocada novamente:

```
L1:
    sub $a0,$a0,1
    jal fact
```

A próxima instrução é onde `fact` retorna. O endereço de retorno e o argumento são restaurados, e o topo da pilha é ajustado:

```
lw $a0, 0($sp)
lw $ra, 4($sp)
add $sp,$sp,8
```

O resultado é então multiplicado (a instrução de multiplicação será detalhada posteriormente):

```
mult $v0,$a0,$v0 #return n * fact(n-1)
```

Finalmente, fact desvia o fluxo para o endereço de retorno:

```
jr $ra
```

A Tabela 1 resume, entre os registradores apresentados, quais devem ou não serem preservados em uma chamada de procedimento.

Preservado	Não preservado
registradores salvos \$s0 - \$s7	registradores temporários \$t0 - \$t9
registrador da pilha \$sp	registradores de argumento \$a0 - \$a3
registrador do endereço de retorno \$ra	registrador do valores de retorno \$v0 - \$v1
pilha além do \$sp	pilha abaixo do \$sp

Tabela 1: O que é e o que não é preservado depois de uma chamada de procedimento.

1.3 Alocando espaço para novos dados

A complexidade final é que a pilha é também utilizada para armazenar variáveis que são locais a um procedimento e não podem ser arranjadas nos registradores disponíveis, tais como vetores ou estruturas. O segmento da pilha contendo os registradores salvos e as variáveis locais é chamado de **quadro do procedimento** (*frame procedure*) ou **registro de ativação**.

Alguns softwares MIPS utilizam o registrador \$fp (frame pointer) para apontar para a primeira palavra do quadro de um procedimento. Sua necessidade é justificada pelo fato do registrador \$sp poder ser alterado durante a execução do procedimento, dificultando a referência ao quadro. Já o \$fp oferece uma base estável para referências locais à memória. A Figura 2 ilustra a alocação da pilha antes, durante e depois da chamada de um procedimento, mostrando os registradores \$sp e \$fp.

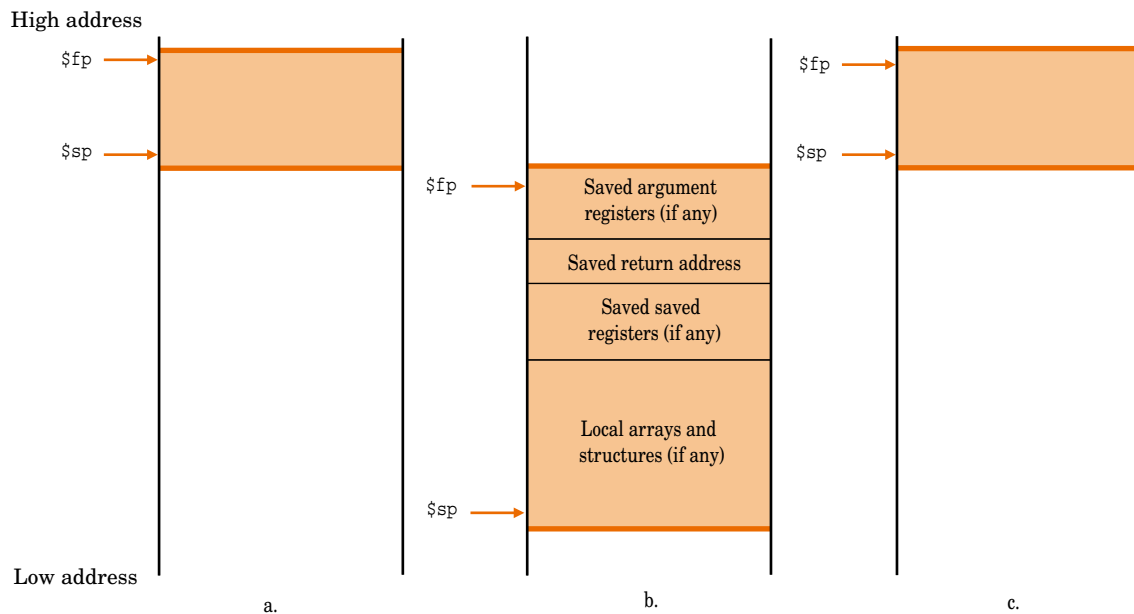


Figura 2: Estado da pilha antes, durante e depois da chamada de um procedimento.

Além das variáveis locais a um procedimento, algumas variáveis utilizadas em programas, denominadas variáveis globais, são “visíveis” em toda a execução, incluindo a parte em que os procedimentos estão executando. Para facilitar o acesso a essas variáveis, o registrador `$gp` (*global pointer*) é utilizado. A Tabela 2 descreve os 32 registradores presentes na arquitetura MIPS.

Nome	Número	Uso	Presevado em uma chamada?
<code>\$zero</code>	0	contém a constante 0	não é alocável
<code>\$v0-\$v1</code>	2-3	valores de resultados e expressões	não
<code>\$a0-\$a3</code>	4-7	argumentos	não
<code>\$t0-\$t7</code>	8-15	temporários	não
<code>\$s0-\$s7</code>	16-23	salvos	sim
<code>\$t8-\$t9</code>	24-25	mais temporários	não
<code>\$gp</code>	28	global pointer	sim
<code>\$sp</code>	29	stack pointer	sim
<code>\$fp</code>	30	frame pointer	sim
<code>\$ra</code>	31	return address	sim

Tabela 2: Convenção dos registradores MIPS. O registrador 1, chamado `$at`, é reservado para o montador, e os registradores 26 e 27, chamados `$k0-$k1`, são reservados para os sistema operacional.

2 Além de Números

Computadores foram criados para manipular inicialmente números. Entretanto, logo que se tornaram comercialmente viáveis, foram utilizados também na manipulação de textos. Grande parte dos computadores atuais utilizam 8-bits (1 byte) para representar caracteres através da codificação ASCII (*American Standard Code for Information Interchange*).

Para manipular esse tipo de dado, assim como uma palavra, MIPS provê instruções para transferência de bytes da memória para um registrador e vice-versa. A instrução `lb` (*load byte* lê um *byte* da memória e escreve nos 8-bits mais à direita de um registrador. Já a instrução `sb` lê os 8-bits mais à direita de um registrador e armazena-os na memória. Uso das instruções está ilustrado a seguir.

```
lb $t0,0($sp)
sb $t0,0($gp)
```

Uma *string* é um conjunto de caracteres e pode ser representada na memória de três modos distintos:

- A primeira posição da string é reservada para armazenar seu tamanho.
- Uma variável contém o tamanho da string (como em uma estrutura).
- A última posição da string é indicada por um caracter utilizado para sinalizar o final de string (é o esquema utilizado em C, sendo o byte 0 o fim de string).

Utilizando a última forma descrita para representar uma cadeia de caracteres, o exemplo a seguir ilustra a utilização das instruções `lb` e `sb`.

Exemplo:

A função `strcpy` copia a string `y` para a string `x` utilizando o byte 0 (byte nulo) como convenção para o fim da string.

```
void strcpy (char x[], char y[])
{
    int i;
    i = 0;
```

```

while ( (x[i] = y[i]) != 0 )
    i = i + 1;
}

```

Qual é o código MIPS, assumindo que o endereço base para os vetores x e y estão em \$a0 e \$a1, enquanto i está em \$s0?

Inicialmente, strcpy ajusta \$sp e então salva o registrador \$s0 na pilha:

```

strcpy:
    sub $sp,$sp,4
    sw $s0, 0($sp)

```

Inicializa i com 0:

```

add $s0,$zero,$zero

```

Inicia o loop. O endereço de y[i] é formado pela adição de i com y[] (observe que não é necessário multiplicar i por 4, uma vez que o vetor é de bytes):

```

L1:
    add $t1,$a1,$s0

```

Carrega o caracter y[i], armazenando-o em \$t2:

```

lb $t2, 0($t1)

```

Calcula o endereço de x[i] e armazena em \$t3, e então o caracter em \$t2 é gravado no endereço:

```

add $t3,$a0,$s0
sb $t2,0($t3)

```

Incrementa o valor de i e volta ao início do loop se o caracter não for 0 (fim da string):

```

add $s0,$s0,1
bne $t2,$zero,L1

```

Se for o último caractere, \$s0 e \$sp são restaurados e então o retorno é chamado:

```

lw $s0, 0($sp)
add $sp,$sp,4
jr $ra

```

Uma vez que o procedimento strcpy é um procedimento folha, o compilador poderia alocar i em um registrador temporário, evitando assim as operações realizadas para salvar e restaurar \$s0.