	...	sp + 28
	a2	sp + 24
	a1	sp + 20
	a0	sp + 16
	ra	sp + 12
	var loc1	sp + 8
	var loc2	sp + 4
sp →	var loc3	sp + 0

Figura 11.10: Registro de ativação do Programa 11.10.

Exemplo 11.12 Vejamos a tradução para *assembly* de uma função simples, empregando as convenções de programação do MIPS.

```
int fun(int g, int h, int i, int j) {
   int f = 0;

   f = (g+h)-(i+j);
   return (f*4);
}
```

Os quatro argumentos são armazenados, da esquerda para a direita nos registradores `a0` a `a3`. O valor da função é retornado em `v0`. A função `f` é uma função folha e portanto é desnecessário empilhar o endereço de retorno, assim como os registradores com os argumentos. O trecho inicial de código mostra a preparação dos argumentos – as variáveis são copiadas para os registradores, e o valor da função é copiado para a variável `k` após o retorno.

Espaço na pilha é alocado para a variável local `f`, num registro de ativação alinhado como *doubleword*. O registro de ativação contém somente a variável local, que é alocada no endereço apontado por `sp`. As operações intermediárias salvam seus resultados em registradores temporários (`t0` a `t3`). O valor intermediário é salvado na variável local, recuperado e então multiplicado por quatro com um deslocamento para a esquerda. O espaço na pilha é desalocado antes do retorno da função.

```
main: ...
      move a0, rg          # quatro argumentos
      move a1, rh
      move a2, ri
      move a3, rj
      jal  fun             # salta para fun()
      move rk, v0          # valor de retorno
      ...
fun: addiu sp, sp, -8      # aloca f na pilha, alinhado
     sw    r0, 0(sp)       # f <- 0
     add   t0, a0, a1      # t0 <- g + h
     add   t1, a2, a3      # t1 <- i + j
     sub   t2, t0, t1
     sw    t2, 0(sp)       # f <- (g+h)-(i+j);
     lw    t3, 0(sp)
     sll   v0, t3, 2       # v0 <- f*4
     addiu sp, sp, 8       # desaloca espaço na pilha
     jr    ra              # retorna
```

Esta função está codificada em 10 instruções, sendo três delas acessos à memória, que são operações deveras custosas. Este estilo de código é o produzido por um compilador, ao compilar sem nenhuma otimização, tal como com `gcc -O0`. ◁

Exemplo 11.13 Vejamos algumas otimizações para reduzir o tamanho e complexidade no código do Exemplo 11.12.

O corpo da função é tão simples, que a variável local pode ser mantida num registrador e portanto não é necessário salvar nada na pilha e o registro de ativação da função é vazio. Economia: duas instruções para manipular a pilha, inicialização de `f`, salvamento e leitura desta variável – todos os acessos à pilha foram eliminados porque desnecessários.

```
fun: add   a0, a0, a1     # a0 <- g + h
     add   a2, a2, a3     # a2 <- i + j
     sub   a2, a0, a2     # a2 <- (g+h)-(i+j);
     sll   v0, a2, 2      # v0 <- f*4
     jr    ra             # retorna
```

Os registradores temporários também são desnecessários porque o corpo da função é simples – tão simples que uma macro seria suficiente. Os valores intermediários são computados nos registradores de argumentos.

A versão otimizada tem cinco instruções e nenhum acesso à memória, além dos acessos inevitáveis para buscar as instruções da função. Esta economia é obtida quando o compilador otimiza o código, por exemplo, com `gcc -O2`. ◁

Exercícios

Ex. 11.3 Traduza para *assembly* a função abaixo. Seu código *assembly* deve empregar as convenções de programação do MIPS. Todas as variáveis estão declaradas e tem os tipos e tamanhos adequados.

```
int fun(int a, int b, int c, int d, int e, int f);
   ...
   int a, p, q, z, w, v[N];
   ...
   x = fun(16*a, z*w, gun(p,q,r,s), v[3], v[z], z-2);
   ...
```

Ex. 11.4 Traduza para *assembly* a função abaixo. Seu código *assembly* deve empregar as convenções de programação do MIPS.

```
int fati(int n) {
   int i,j;
   j=1;
   if(n > 1)
      for(i = 1; i <= n; i++)
        j = j*i;
   return(j);
}
```