5 Suporte a Funções

Material baseado em "A Book on C", Kelley & Pohl, 2a ed, Benjamin/Cummings

Programa 1: Definição de uma função em C.

```
tipo nome_da_funcao( parametros formais ) { // cabecalho
   declaracoes                              // corpo da funcao
   comandos
}
```

Programa 2: Declaração de uma função em C.

```
tipo nome_da_funcao( lista de tipos dos parametros );
```

Programa 3: Exemplos de declarações de funções em C.

```
void f(void);  // sem argumentos e nao retorna valor
int  g(void);  // sem argumentos e retorna valor inteiro
int  h(int, char); // dois argumentos, retorna valor inteiro
int  j(const char *, ...); // numero variavel de argumentos
```

Invocação de função, *call by value*

1. como são avaliados os comandos e expressões em C? Da esquerda para a direita, em avaliação preguiçosa, e com efeitos colaterais. O mesmo vale para os argumentos de uma função;
2. cada expressão na lista de parâmetros é avaliada;
3. se necessários os valores das expressões são convertidos para o tipo do parâmetro formal, e o valor é atribuído ao parâmetro correspondente no corpo da função;
4. o corpo da função é executado;
5. se um comando **`return`** é executado, o controle é devolvido à função que chamou;
6. se o **`return`** inclui uma expressão, seu valor é computado e o tipo convertido para o tipo do valor de retorno da função. Se o **`return`** não contém uma expressão, nenhum valor útil é retornado. Se o corpo da função não inclui um **`return`**, então o controle é devolvido quando a execução do corpo da função chegar ao seu último comando e nenhum valor útil é retornado;
7. todos os argumentos são passados "por valor" (*call by value*), mesmo que o 'valor' seja um endereço (*pointer*).

5.1 Regras de escopo

O valor de identificadores só pode ser acessado nos blocos em que são declarados.

5.2 Classes de armazenagem (*storage classes*)

auto variáveis declaradas dentro de um bloco (variáveis locais);

extern variáveis declaradas fora do corpo de uma função; seu escopo se estende a todas as funções que aparecem após sua declaração. Funções podem ser declaradas como extern;

register indica ao compilador que variável deve, se possível, ser alocada num registrador físico (raramente implementado nos compiladores);

static variáveis declaradas estáticas num bloco retém seus valores entre execuções do bloco;

static (external) variáveis declaradas fora de um bloco mas com escopo restrito ao arquivo em que são declaradas. Funções declaradas estáticas são visíveis apenas no arquivo de declaração.

Variáveis das classes extern e static, se não forem inicializadas pelo programador, são inicializadas pelo compilador em 0.

5.3 Implementação de funções no MIPS32

O processador provê duas instruções para o suporte a funções, *viz.*

JAL, *jump and link*, com dois efeitos: salta para o endereço indicado no argumento e salva o endereço de retorno em r31=ra (*return address*), que é o *link*:

```
jal ender #PC <- ender , ra <- PC+4
```

JR, *jump register*, que salta para o endereço de ligação/retorno, que foi armazenado em ra por JAL.

```
jr ra #PC <- ra
```

A Figura 1 mostra a convenção de uso dos registradores definida na *Application Binary Interface* (ABI) do MIPS32.

$zero	sempre zero (em *hardware*)	r0
at	temporário para o montador	r1
v0-v1	dois registradores para retornar valores;	r2,r3
a0-a3	quatro registradores para passar parâmetros;	r4..r7
s0..s7	registradores 'salvados' entre chamadas de função	r16..r23
t0..t9	registradores 'temporários' não são preservados entre funções	
k0,k1	temporários para o SO (*kernel*)	r26,r27
gp	global pointer (dados estáticos 'pequenos')	r28
sp	apontador de pilha (*stack pointer*)	r29
fp	apontador do registro de ativação (*frame pointer*)	r30
ra	endereço de retorno (*return address*)	r31

Figura 1: Convenção de uso de registradores para chamadas de função

A cada chamada de função encontrada num programa, o compilador deve gerar instruções para efetuar os 6 passos listados abaixo. O Programa 4 mostra o código *assembly* para a implementação do comando `z = int f(int x);` . Os números das linhas indicadas referem-se a este trecho de código.

1. alocar os parâmetros onde o corpo da função possa encontrá-los (linha 1);
2. transferir controle para a função e armazenar *link* (linha 2);
3. o corpo da função deve alocar o espaço necessário na pilha para computar seu resultado (linha 5);
4. executar as instruções do corpo da função (linha 6);
5. colocar o valor computado onde a função que chamou possa encontrá-lo (linha 7);
6. devolver o espaço alocado em pilha (linha 8);
7. retornar controle ao ponto de invocação da função (linha 9).

Programa 4: Protocolo de invocação de função.

```
1          move a0,rx          # prepara argumento
2          jal  f              # salta para a funcao
3          move rz,v0          # valor retornado, endereco de retorno
4          ...
5  f:      add  sp, sp, -40    # aloca espaco na pilha
6          ...                 # computa valor
7          move v0,rw          # prepara valor por retornar
8          add  sp, sp, 40     # devolve espaco da pilha
9          jr   ra             # retorna, usando o link
```

5.3.1 Registro de Ativação

Qual é a estrutura de dados necessária para suportar funções? Por que?

Uma "função folha" é uma função que não invoca outra(s) função(ões). Um registro de ativação (`stack frame`) é alocado para cada função não-folha e para cada função folha que necessita de alocação de variáveis locais. A pilha cresce de endereços mais altos para endereços mais baixos. A Figura 2 mostra o *layout* de um registro de ativação completo.

Um registro de ativação deve conter espaço para:

variáveis locais e temporárias

registradores salvados espaço só é alocado para aqueles registradores que devem ser preservados. Uma função não-folha deve salvar ra. Se qualquer dentre r16-r23 (s0-s7) ou r29-r31 (sp,fp,ra) são alterados no corpo da função, estes devem ser salvados na pilha e restaurados antes do retorno da função. Registradores são empilhados na ordem de número, com registradores de números maiores armazenados em endereços mais altos. A área de salvamento de registradores deve ser alinhada como *doubleword* (8 bytes);

área para argumentos de chamada de função numa função não-folha, o espaço necessário para todos os argumentos que podem ser usados para invocar outras funções deve ser reservado na pilha. No mínimo, quatro palavras devem ser sempre reservadas, mesmo que o número de argumentos passados a qualquer função seja menor que quatro palavras.

alinhamento a convenção (válida para o SO, inclusive) exige que um registro de ativação seja alinhado como *doubleword*. O alinhamento é em *doubleword* porque este é o maior tamanho de "palavra" que pode ser empilhado, que é uma variável de ponto flutuante do tipo `double`.

	registro da função que chamou
	argumentos a mais que 4 (5,6,7...)
fp →	registradores com argumentos 1 a 4, se existem (a0..a3)
	endereço de retorno (ra)
	registradores salvados, se alterados pela função (s0..s7)
	variáveis locais, se existem
sp →	área para construir argumentos a mais que 4, se existem
	registro da próxima função a ser chamada

Figura 2: Registro de ativação no MIPS-32.

Uma função aloca seu registro de ativação ao subtrair do *sp* o tamanho de seu registro, no início de seu código. O ajuste no *sp* deve ocorrer antes que aquele registrador seja usado na função, e antes de qualquer instrução de salto ou desvio. A de-alocação do registro deve ocorrer no último bloco básico da função, que inclui todas as instruções após o último salto ou desvio do código até a instrução de retorno (*jump-register*).

A ordem de armazenagem dos componentes no registro de ativação deve ser respeitada mesmo que o código de uma função não os utilize todos.

A Figura 3 mostra quais recursos devem ser preservados pelo código de uma função. Do ponto de vista da função que chama, nenhum dos recursos do lado esquerdo da tabela pode ser alterado pela função chamada.

Preservados	Não preservados
s0..s7 (registradores salvados)	t0..t9 (temporários)
sp (*stack pointer*)	a0..a4 (argumentos)
ra (*return address*)	v0,v1 (valores de retorno)
	at, k0, k1 (*assembler temporary, kernel*)
pilha acima do sp	pilha abaixo do sp

Figura 3: Preservação de conteúdos entre chamadas de funções.

Ver [http://www.inf.ufpr.br/roberto/ci064/mipsabi30.pdf] páginas 3.11 a 3.21

Como um exemplo, o código da função `int g(int x, int y, int z);` é mostrado no Programa 5. A função `g` declara três variáveis locais em seu corpo, e é uma função não-folha. Seu registro de ativação, mostrado na Figura 4, deve acomodar 3 parâmetros, o registrador com o endereço de retorno, e as três variáveis locais, perfazendo 28 bytes, que alinhado como *doubleword*, resulta em 32 bytes.

Programa 5: Parte do código da função `g(x,y,z)`.

```
1        # w = g(x,y,z);
2        move  a0,rx          # prepara 3 parametros
3        move  a1,ry
4        move  a2,rz
5        jal   g              # salta
6        move  rw,v0          # recebe valor de retorno
7        ...
8  g:    addiu sp,sp,-32      # aloca espaco para 3 param + ra + 3 vars
9        sw    a0,12(sp)      # empilha a0
10       sw    a1,16(sp)      # empilha a1
11       sw    a2,20(sp)      # empilha a2
12       sw    ra,24(sp)      # empilha endereco de retorno
13       ...                  # corpo de g()
```

```
14      move  v0,rw         # valor de retorno
15      lw    ra,24(sp)     # recompoe endereco de retorno
16      addiu sp,sp,32      # de-aloca espaco na pilha
17      jr    ra            # retorna
18      ...
```

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

Figura 4: Registro de ativação do Programa 5.

Exercício: traduzir para assembly a função abaixo:

```
1  int fun(int g, int h, int i, int j){
2      int f;
3      f = (g+h)-(i+j);
4      return (f*4);
5  }
```

Exercício: traduzir para assembly a função abaixo:

```
1  int fat(int n) {
2      int i,j;
3      j=1;
4      if(n>1)
5         for(i=1; i<=n; i++)
6           j = j*i;
7      return(j);
8  }
```