Chamadas de Função

Yuri Kaszubowski Lopes

UDESC

Anotações

Funções

O que é uma função?

- Um trecho contendo instruções, que recebe alguns parâmetros, e pode ou não retornar uma resposta ao "chamador"
  - Função **chamada** e função **chamadora**
- Analogia de Patterson e Henessy
  - Uma função é um espião que sai com um plano secreto
  - Adquire recursos, realiza a tarefa, cobre seus rastros e retorna ao ponto de origem com o resultado solicitado
  - Nada mais é perturbado depois da "missão" terminar
  - Um espião opera apenas com o que ele precisa saber
    - Não sabe nada sobre quem o chamou

Anotações

Funções

- Como podemos imaginar uma função com as instruções do MIPS??
  - Podemos imaginar como um grupo de instruções que realiza uma tarefa
  - Saltamos para esse grupo
    - Identificado por um rótulo (label)
  - No final, temos que elaborar alguma forma para inserir no contador de programa (PC) o endereço da instrução posterior à instrução que saltou para a função
    - Retornar ao "chamador"

Anotações

Etapas para chamar a função

- Vamos criar uma função que faz o seguinte (exemplo de Patterson, Henessy):

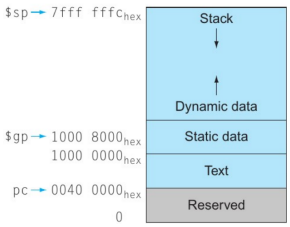
```
1 int leaf_example(int g, int h, int i, int j){
2     int f;
3     f = (g+h) - (i+j);
4     return f;
5 }
```

- Etapas para chamar a função:
  1. Colocar os parâmetros em algum lugar que a função possa acessar
     - Onde?
  2. Transferir o controle para a função
     - Como?
  3. Adquirir os recursos de armazenamento necessários
  4. Realizar a tarefa
  5. Colocar o valor de retorno em um lugar visível ao chamador: `return`
  6. Liberar os recursos e limpar os rastros
  7. Retornamos o controle ao chamador

Anotações

1. Passagem de parâmetros

- Colocar os parâmetros em algum lugar que a função possa acessar.
  - No MIPS, temos os registradores de argumento `$a0`, `$a1`, `$a2` e `$a3`
  - E se precisarmos de mais argumentos?
    - Salvamos na pilha (memória)



- Arquiteturas diferentes possuem formas diferentes para se passar os parâmetros
  - Em x86-64: No modo 64 bits, os primeiros 6 parâmetros vão em `rdi`, `rsi`, `rdx`, `rcx`, `r8`, e `r9`.
    - Pode diferir ainda dependendo do sistema operaciona
  - Em x86: No modo 32 bits, todos parâmetros são passados via pilha

Anotações

1. Passagem de parâmetros

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

```
1     .text
2     .globl main
3 main:
4     ori $a0, $zero, 1  # argumento g
5     ori $a1, $zero, 2  # argumento h
6     ori $a2, $zero, 3  # argumento i
7     ori $a3, $zero, 4  # argumento j
8 end:
9     li $v0, 10
10    syscall
11 leaf_example:
12    #vamos escrever nossa função aqui
```


Anotações

2. Transferir o controle para a função

- Como Transferir o controle para a função?
- Poderíamos fazer um jump simples, certo?
  - O problema é que não saberemos o endereço para retornar posteriormente!
- O MIPS inclui uma instrução especial chamada jump-and-link (`jal`)
- Salva o endereço da próxima instrução no registrador `$ra` (return address) e só então salta para o endereço especificado

```
1     .text
2     .globl main
3 main:
4     ori $a0, $zero, 1  # argumento g
5     ori $a1, $zero, 2  # argumento h
6     ori $a2, $zero, 3  # argumento i
7     ori $a3, $zero, 4  # argumento j
8     jal leaf_example
9 end:
10    li $v0, 10
11    syscall
12 leaf_example:
13    #vamos escrever nossa função aqui
```


Anotações

2. Transferir o controle para a função

- Qual o endereço em `$ra` e em `$pc` quando o `jal` é executado?

```
1                 .text
2                 .globl main
3             main:
4 0x00400000      ori $a0, $zero, 1  # argumento g
5 0x00400004      ori $a1, $zero, 2  # argumento h
6 0x00400008      ori $a2, $zero, 3  # argumento i
7 0x0040000C      ori $a3, $zero, 4  # argumento j
8 0x00400010      jal leaf_example
9 0x00400014      # Outras instruções
10            end:
11 0x00400200     li $v0, 10
12 0x00400204     syscall
13            leaf_example:
14 0x00400208     #vamos escrever nossa função aqui
```

- `$ra` em `0x00400014`
- `$pc` em `0x00400208`

Anotações

3. Adquirir os recursos de armazenamento

- Vamos assumir que no nosso exemplo, os registradores `$s0` e `$s1` serão utilizados para realizar a operação da função
- Qual o problema?
  - Esses registradores podem conter valores que estão sendo utilizados pelo chamador
    - "O espião não pode assumir nada quanto ao seu contratante"
    - "O espião deve limpar seus rastros após a missão"
    - Retornar tudo para a forma que estava anteriormente
  - Como podemos fazer isso?
    - Vamos salvar `$s0` e `$s1` na pilha
    - Registrador `$sp` contém o endereço atual do topo da pilha
    - SP → Stack Pointer (Ponteiro de Pilha)

```
1 leaf_example:
2     add $s0, $a0, $a1
3     add $s1, $a2, $a3
4     sub $v0, $s0, $s1
```

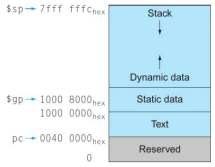
```
1 int leaf_example(int g, int h, int i, int j){
2     int f;
3     f = (g+h) - (i+j);
4     return f;
5 }
```


Anotações

3. Adquirir os recursos de armazenamento

```
leaf_example:
    addi $sp, $sp, -8  # deslocando o topo da pilha 8 bytes
    sw $s0, 0($sp)  # armazenando $s0 nos últimos 4 bytes da pilha
    sw $s1, 4($sp)  # armazenando $s1 nos 4 bytes após $s0
    add $s0, $a0, $a1
    add $s1, $a2, $a3
    sub $v0, $s0, $s1
```

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



Anotações

4. Realizar a tarefa e 5. return

```
leaf_example:
    addi $sp, $sp, -8  # deslocando o topo da pilha 8 bytes
    sw $s0, 0($sp)  # armazenando $s0 nos últimos 4 bytes da pilha
    sw $s1, 4($sp)  # armazenando $s1 nos 4 bytes após $s0
    add $s0, $a0, $a1
    add $s1, $a2, $a3
    sub $v0, $s0, $s1
```

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

- Colocar o valor de retorno em um lugar visível ao chamador: `return`
  - No MIPS temos os registradores `$v0` e `$v1` para valores de retorno
  - Se precisarmos de mais valores de retorno, podemos mais uma vez utilizar a pilha
  - Mais uma vez, pode depender da arquitetura e do S.O.

Anotações

6. Liberar os recursos e limpar os rastros

- Restauramos os valores salvos para os registradores
- Ajustamos a pilha

```
leaf_example:
    addi $sp, $sp, -8  # deslocando o topo da pilha 8 bytes
    sw $s0, 0($sp)  # armazenando $s0 nos últimos 4 bytes da pilha
    sw $s1, 4($sp)  # armazenando $s1 nos 4 bytes após $s0
    add $s0, $a0, $a1
    add $s1, $a2, $a3
    sub $v0, $s0, $s1
    lw $s0, 0($sp)  # restaurando o valor de $s0
    lw $s1, 4($sp)  # restaurando o valor de $s1
    addi $sp, $sp, 8  # ajustando topo da pilha para "excluir" os itens
```


Anotações

7. Retornamos o controle ao chamador

- Retornamos o controle ao chamador
- Instrução especial jump register (`jr`)
  - Salta para o endereço armazenado no registrador `$ra`

```
leaf_example:
    addi $sp, $sp, -8
    sw $s0, 0($sp)
    sw $s1, 4($sp)
    add $s0, $a0, $a1
    add $s1, $a2, $a3
    sub $v0, $s0, $s1
    lw $s0, 0($sp)
    lw $s1, 4($sp)
    addi $sp, $sp, 8
    jr $ra  # saltando para o endereço armazenado em $ra
```


Anotações

Código completo

```
    .text
    .globl main
main:
    ori $a0, $zero, 1  # argumento g
    ori $a1, $zero, 2  # argumento h
    ori $a2, $zero, 3  # argumento i
    ori $a3, $zero, 4  # argumento j
    jal leaf_example
end:
    li $v0, 10
    syscall
leaf_example:
    addi $sp, $sp, -8
    sw $s0, 0($sp)
    sw $s1, 4($sp)
    add $s0, $a0, $a1
    add $s1, $a2, $a3
    sub $v0, $s0, $s1
    lw $s0, 0($sp)
    lw $s1, 4($sp)
    addi $sp, $sp, 8
    jr $ra  # saltando para o endereço armazenado em $ra
```


Anotações

Equivalente em C

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

int main() {
    leaf_example(1, 2, 3, 4);
    return 0;
}
```


Anotações

Salvar ou não salvar

- No exemplo salvamos os registradores `$s_` para recuperá-los posteriormente
  - Isso não é necessário (por convenção) para todos os registradores
  - Veja na tabela o que sua função deve ou não preservar
  - Note que o processador não salva sozinho. Você é quem deve garantir que os conteúdos são salvos
    - Quem chamar sua função espera que `$s0` — `$s7` sejam devolvidos intactos ao término da função
    - O conteúdo de `$t0` — `$t9` não é garantido de se manter intacto após a chamada

Preservado	Não preservado
`$s0` — `$s7`	`$t0` — `$t9`
`$sp`	`$a0` — `$a3`
`$ra`	`$v0` — `$v1`
Pilha acima de `$sp`	Pilha abaixo de `$sp`

Anotações

Exercícios - em assembly do MIPS32

1. Execute o programa de exemplo no MARS passo a passo, verificando os conteúdos dos registradores sendo modificados e os conteúdos da memória.
2. Faça uma função que receba **dois** números e retorne o maior deles
   - Faça também a função chamadora `main` que deve solicitar os argumentos e imprimir os resultados para o usuário
3. Faça uma função que receba **três** números e retorne o maior deles em *v*0 e o menor em *v*1
   - Faça também a função chamadora `main` que deve solicitar os argumentos e imprimir os resultados para o usuário
4. Crie uma função que recebe um valor inteiro N, e retorne quantos dígitos N possui
   - Exemplo: 12345 possui 5 dígitos
   - Dica: utilize sucessivas divisões por 10 para obter o valor
   - Faça também a função chamadora `main` que deve solicitar os argumentos e imprimir os resultados para o usuário
5. Descreva pelo menos duas vantagens e duas desvantagens de se programar utilizando assembly quando comparado com linguagens compiladas (e.g., C).

Anotações

Exercícios

6. Traduza a função em C com 6 argumentos (parâmetros) abaixo para assembly do MIPS
   - Você deverá utilizar a pilha para passar pelo menos dois destes argumentos
   - Faça também a função chamadora `main` conforme código abaixo

```c
int leaf_example(int g, int h, int i, int j, int k, int z){
    int f;
    f = (g+h) - (i+j) + 2 * (k+z);
    return f;
}

int main() {
    leaf_example(1, 2, 3, 4, 5, 6);
    return 0;
}
```

7. Qual o problema ocorre se fizermos uma chamada de função dentro de outra chamada de função? Como isso pode ser corrigido?

Anotações