Dilbert, por Scott Adams.

Revisão Chamadas de Função

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

Funções

Segundo Patterson e Hennessy, uma função:

- Adquire recursos, realiza a tarefa, cobre seus rastros e retorna ao ponto de origem com o resultado solicitado.
- Nada mais é perturbado depois da chamada da função.
- Opera apenas com os dados que precisa saber (parâmetros).
- Não sabe nada sobre quem a chamou.

Funções

Uma função é um grupo de instruções que realiza uma tarefa.

Saltamos (jump) para esse grupo.

No final, temos que elaborar alguma forma para inserir no contador de programa o endereço da instrução posterior à instrução que saltou para a função.

Retornar ao “chamador”.

Exemplo

Exemplo em C, de Patterson e Hennessy (2020).

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

Etapa 1

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

Antes de chamar a função, precisamos carregar os parâmetros `g`, `h`, `i` e `j` para algum lugar que a função possa ver.

A convenção do RISC-V é passar os parâmetros nos registradores x10 a x17.

Se precisarmos de mais parâmetros, usar a pilha (memória principal).

Etapa 1

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

Antes de chamar a função, precisamos carregar os parâmetros `g`, `h`, `i` e `j` para algum lugar que a função possa ver.

A convenção do RISC-V é passar os parâmetros nos registradores x10 a x17.

Se precisarmos de mais parâmetros, usar a pilha (memória principal).

Suponha que vamos chamar a função passando `g=1`,`h=2`,`i=3`,`j=4`. Como fica o trecho em Assembly RISC-V?

Etapa 1

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

```
main:
    ori x10, x0, 1
    ori x11, x0, 2
    ori x12, x0, 3
    ori x13, x0, 4

end:
    ori x10, x0, 17
    ori x11, x0, 0
    ecall
```

Etapa 2 - Transferir o Controle

```
main:
    ori x10, x0, 1
    ori x11, x0, 2
    ori x12, x0, 3
    ori x13, x0, 4
    jal x1, leaf_example

end:
    ori a0, x0, 17
    ori a1, x0, 0
    ecall

leaf_example:
    #vamos escrever nossa função aqui
```

Etapa 2 - Transferir o Controle

```
main:
    ori x10, x0, 1
    ori x11, x0, 2
    ori x12, x0, 3
    ori x13, x0, 4
    jal x1, leaf_example

end:
    ori a0, x0, 17
    ori a1, x0, 0
    ecall

leaf_example:
    #vamos escrever nossa função aqui
```

`jal` (Jump and Link) vai salvar o endereço da próxima instrução que seria executada (`ori a0, x0, 17`), e saltar para o rótulo.

Etapa 3 - Salvar

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

Para implementar as operações da função, vamos usar os registradores x5, x6 e x20.

Observação: poderíamos ser mais eficientes, utilizando menos registradores, mas vamos usar esses para exercitar.

Problema:

E se quem chamou a função tivesse salvo valores úteis nesses registradores? Vamos sobrescrever?

Salvos e não salvos

A convenção do RISC-V sobre a responsabilidade de salvar os valores dos registradores é a da tabela.

Registrador	Descrição	Quem Salva?
x1 (ra)	Endereço de Retorno.	Chamado.
x2 (sp)	Stack Pointer.	Chamado.
x5-x7	Temporários (para realizar operações).	Quem chama.
x8-x9	Salvos (para realizar operações).	Chamado.
x10-x17	Argumentos e resultados.	Quem chama.
x18-x27	Salvos (para realizar operações).	Chamado.
x28-x31	Temporários (para realizar operações).	Quem chama.

Etapa 3 - Salvar

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

Para implementar as operações da função, vamos usar os registradores x5, x6 e x20.

Vamos precisar salvar os valores na Pilha, para restaurar depois.

Mapa da Memória

Reservado: por exemplo, para fazer I/O em memória.

Texto: onde instruções do programa ficam.

Dados Estáticos: variáveis estáticas alocadas (ex.: variáveis globais).

Dados Dinâmicos: variáveis alocadas dinamicamente (heap).

Pilha: variáveis locais.

Isso é apenas uma convenção, e não uma restrição de Hardware. No Venus, por exemplo, o Stack Pointer aponta para 0x7FFFFFDC no início do programa.

Pilha

O registrador x2 é utilizado como ponteiro de pilha (stack pointer – sp).

Armazena o endereço atual da pilha.

A pilha é invertida!

Começa no último endereço válido de memória, e cresce em direção ao primeiro.

Até colidir com a heap. Stack Overflow!

Etapa 3 - Salvar

Vamos precisar salvar 3 registradores na pilha.

Primeiro, abrir espaço.

```
     …

leaf_example:
     addi sp, sp, -12
```

Etapa 3 - Salvar

Salvar os registradores

```
     …

leaf_example:
     addi sp, sp, -12
     sw x5, 8(sp)
     sw x6, 4(sp)
     sw x20, 0(sp)
```

Etapa 4 - Implementar

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

```
    …

leaf_example:
    addi sp, sp, -12
    sw x5, 8(sp)
    sw x6, 4(sp)
    sw x20, 0(sp)

    add x5, x10, x11 #g + h
    add x6, x12, x13 #i + j
    sub x20, x5, x6  #(g + h) - (i + j)
```

Etapa 5 - Retorno

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

Deixar o retorno em algum lugar visível.

- Registradores x10 e x11.
- …

```
leaf_example:
     addi sp, sp, -12
     sw x5, 8(sp)
     sw x6, 4(sp)
     sw x20, 0(sp)

     add x5, x10, x11 #g + h
     add x6, x12, x13 #i + j
     sub x20, x5, x6  #(g + h) - (i + j)
     or x10, x0, x20
```

Etapa 6 - Restaurar

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

Restaurar o estado dos registradores salvos, e do stack pointer.

…

```
leaf_example:
     addi sp, sp, -12
     sw x5, 8(sp)
     sw x6, 4(sp)
     sw x20, 0(sp)

     add x5, x10, x11 #g + h
     add x6, x12, x13 #i + j
     sub x20, x5, x6  #(g + h) - (i + j)
     or x10, x0, x20

     lw x20, 0(sp)
     lw x6, 4(sp)
     lw x5, 8(sp)
     addi sp, sp, 12
```

Etapa 7 - Retornar

Instrução jalr reg1, reg2, Imediato

Salta para o endereço armazenado em reg2 somado com o imediato.

O endereço da próxima instrução é salvo em reg1.

Etapa 7 - Retornar

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

Restaurar o estado dos registradores salvos, e do stack pointer.

…

```
leaf_example:
     addi sp, sp, -12
     sw x5, 8(sp)
     sw x6, 4(sp)
     sw x20, 0(sp)

     add x5, x10, x11 #g + h
     add x6, x12, x13 #i + j
     sub x20, x5, x6  #(g + h) - (i + j)
     or x10, x0, x20

     lw x20, 0(sp)
     lw x6, 4(sp)
     lw x5, 8(sp)
     addi sp, sp, 12

     jalr x0, 0(x1)
```

Programa

```
main:
      ori x10, x0, 1
      ori x11, x0, 2
      ori x12, x0, 3
      ori x13, x0, 4
      jal x1, leaf_example
      or x11, x0, x10
      ori x10, x0, 1 # Print int
      ecall

end:
      ori a0, x0, 17
      ori a1, x0, 0
      ecall

leaf_example:
      addi sp, sp, -12
      sw x5, 8(sp)
      sw x6, 4(sp)
      sw x20, 0(sp)

      add x5, x10, x11 #g + h
      add x6, x12, x13 #i + j
      sub x20, x5, x6  #(g + h) - (i + j)
      or x10, x0, x20

      lw x20, 0(sp)
      lw x6, 4(sp)
      lw x5, 8(sp)
      addi sp, sp, 12

      jalr x0, x1, 0
```

Funções não folha

O exemplo anterior era uma função folha.

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

Uma função que realiza sua tarefa e retorna **sem chamar outra função**.

Seríamos mais felizes se toda função fosse folha.

Mas nem tudo são flores.

Funções não folha

Uma função que chama outra internamente para resolver o problema é uma **função não folha**.

A função pode **chamar outra função**, ou um **clone de si mesma (recursão)**.

Os problemas são os mesmos em ambos os casos.

Exemplo

Considere a função que calcula o fatorial recursivamente.

Que problemas criamos agora a nível de linguagem de montagem que não existiam em uma função folha?

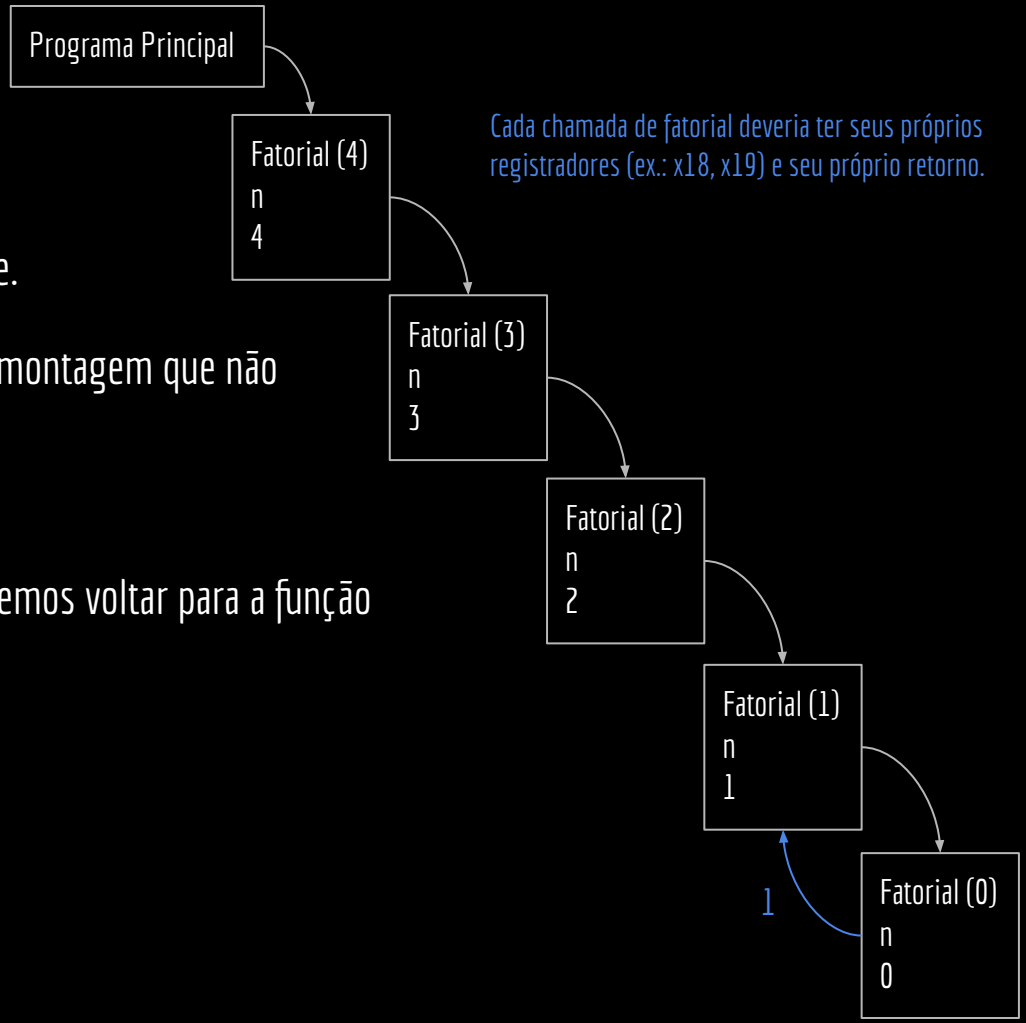
```
int fatorial(int n){
     if(n < 1)
          return 1;
     return n * fatorial(n-1);
}
```

Exemplo

Considere a função que calcula o fatorial recursivamente.

Que problemas criamos agora a nível de linguagem de montagem que não existiam em uma função folha?

- Os valores dos registradores podem se perder.
- O endereço de retorno vai se perder, e não saberemos voltar para a função original que chamou `fatorial`.




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

Um abacaxi

Cada chamada de fatorial deveria ter seus próprios registradores (ex.: x18, x19) e seu próprio retorno.

Como podemos resolver?

Um abacaxi

Cada chamada de fatorial deveria ter seus próprios registradores (ex.: x18, x19) e seu próprio retorno.

Como podemos resolver?

> **Empilhar** os valores que precisam ser salvos.

Fatorial Recursivo

```
fatorial_rec:
     bge x0,x10,fatorial_0 #Caso base se chegamos em 0
```

```
fatorial_0:                #caso base
     ori x10,x0,1
     jalr x0, x1, 0
```

Fatorial Recursivo

Antes de chamar recursivamente, salvar na pilha o valor de x10 e o endereço de retorno.

```
fatorial_rec:
     bge x0,x10,fatorial_0 #Caso base se chegamos em 0
     addi sp, sp, -8       #ajusta a pilha para 2 itens
     sw x10, 0(sp)         #guarda x10 para depois
     sw x1, 4(sp)          #guarda o endereço de retorno
```

```
fatorial_0:                #caso base
     ori x10,x0,1
     jalr x0, x1, 0
```

Fatorial Recursivo

Ajustar o parâmetro e fazer a chamada recursiva.

```
fatorial_rec:
     bge x0,x10,fatorial_0 #Caso base se chegamos em 0
     addi sp, sp, -8       #ajusta a pilha para 2 itens
     sw x10, 0(sp)         #guarda x10 para depois
     sw x1, 4(sp)          #guarda o endereço de retorno
     addi x10, x10, -1
     jal x1, fatorial_rec  #chama fatorial para n-1
```

```
fatorial_0:                #caso base
     ori x10,x0,1
     jalr x0, x1, 0
```

Fatorial Recursivo

A função retorna o resultado de fat(n-1) em x10.

Restaurar o parâmetro original (n) e o endereço de retorno a partir da memória.

Restaurar a pilha para a posição original.

```
fatorial_rec:
     bge x0,x10,fatorial_0 #Caso base se chegamos em 0
     addi sp, sp, -8       #ajusta a pilha para 2 itens
     sw x10, 0(sp)         #guarda x10 para depois
     sw x1, 4(sp)          #guarda o endereço de retorno
     addi x10, x10, -1
     jal x1, fatorial_rec  #chama fatorial para n-1
     lw x5, 0(sp)          #carrega o valor antigo de x10 para x5
     lw x1, 4(sp)          #carrega o endereço de retorno
     addi sp, sp, 8        #restaura posição da pilha


fatorial_0:                #caso base
     ori x10,x0,1
     jalr x0, x1, 0
```

Fatorial Recursivo

Realizar a operação n*fat(n-1).

Retornar o resultado para quem chamou.

```
fatorial_rec:
     bge x0,x10,fatorial_0 #Caso base se chegamos em 0
     addi sp, sp, -8       #ajusta a pilha para 2 itens
     sw x10, 0(sp)         #guarda x10 para depois
     sw x1, 4(sp)          #guarda o endereço de retorno
     addi x10, x10, -1
     jal x1, fatorial_rec  #chama fatorial para n-1
     lw x5, 0(sp)          #carrega o valor antigo de x10 para x5
     lw x1, 4(sp)          #carrega o endereço de retorno
     addi sp, sp, 8        #restaura posição da pilha
     mul x10, x10, x5      #calcula n*fat(n-1)
     jalr x0, x1, 0
fatorial_0:                #caso base
     ori x10,x0,1
     jalr x0, x1, 0
```

Solução

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

```
     .text
main:
     ori x10, x0, 4 #argumento
     jal x1, fatorial_rec
     or x11, x0, x10
     ori x10, x0, 1 # Print int
     ecall
end:
     ori a0, x0, 17
     ori a1, x0, 0
     ecall
fatorial_rec:
     bge x0,x10,fatorial_0 #Caso base se chegamos em 0
     addi sp, sp, -8       #ajusta a pilha para 2 itens
     sw x10, 0(sp)         #guarda x10 para depois
     sw x1, 4(sp)          #guarda o endereço de retorno
     addi x10, x10, -1
     jal x1, fatorial_rec  #chama fatorial para n-1
     lw x5, 0(sp)          #carrega o valor antigo de x10 para x5
     lw x1, 4(sp)          #carrega o endereço de retorno
     addi sp, sp, 8        #restaura posição da pilha
     mul x10, x10, x5      #calcula n*fat(n-1)
     jalr x0, x1, 0
fatorial_0:                #caso base
     ori x10,x0,1
     jalr x0, x1, 0
```

Pontos a Considerar

Muitos problemas possuem soluções mais simples quando utilizado o conceito de recursão.

Exemplo: navegar em uma árvore binária.

Pontos a Considerar

A recursão (ou mesmo chamada de procedimentos não folha) custa caro para a máquina.

Por quê?

Pontos a Considerar

A recursão (ou mesmo chamada de procedimentos não folha) custa caro para a máquina.

Cada chamada exige comunicação com a memória para empilhar os dados.

- Ocupa espaço na pilha.
- Ocupa a CPU transferindo dados.

Invalida a memória cache (veremos na disciplina).

Sendo assim, uma solução iterativa é preferível a nível de linguagem de máquina.

- Compiladores modernos fazem o que podem para tentar eliminar recursão.

Exercícios - Sem Recursão

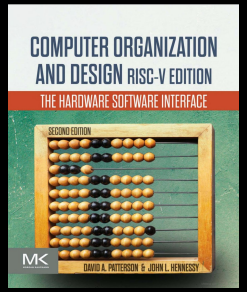
1. Execute os programas da aula no Venus passo a passo, verificando os conteúdos dos registradores sendo modificados e os conteúdos da memória.
2. Crie uma função que retorna o enésimo número da sequência de Fibonacci. Considere que n é passado como parâmetro. Utilize uma implementação iterativa para Fibonacci.
3. Crie uma função que recebe um valor inteiro N, e retorna quantos dígitos N possui. Por exemplo, o número 12345 possui 5 dígitos.
4. Considere os polinômios de terceiro grau, que são na forma $ax^3 + bx^2 + cx + d$. Crie uma função que recebe como parâmetro os coeficientes a, b, c e d, e também um ponto x, e devolve o valor de x no ponto especificado. Considere que todos os valores são inteiros.
5. Descreva pelo menos duas vantagens e duas desvantagens de se programar utilizando assembly quando comparado com linguagens compiladas (ex.: C).

Exercícios - Com Recursão

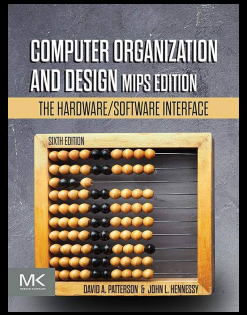
6. Execute o fatorial recursivo no Venus passo a passo, analise e entenda as mudanças ocorridas em cada um dos registradores e nos endereços de memória.
7. Crie uma função recursiva para calcular o enésimo número da sequência de Fibonacci.
8. Escreva uma função recursiva que determina quantas vezes um dígito K ocorre em um número natural N. Por exemplo, o dígito 2 ocorre 3 vezes em 762021192.
9. Escreva uma função recursiva que calcula a soma dos dígitos de um valor inteiro qualquer passado como parâmetro. Exemplo: Para a entrada 36, a resposta é 3 + 6 = 9.
10. Escreva uma função recursiva que recebe como parâmetro um inteiro positivo qualquer, e retorna (em um inteiro) o seu equivalente em binário. Exemplo: Para 66_{10}, a resposta deve ser 1000010_{2}.

Referências

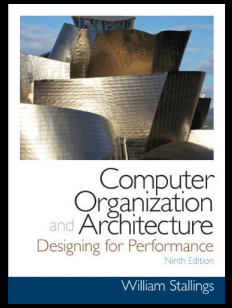
Patterson, Hennessy. Computer Organization and Design RISC-V Edition: The Hardware Software Interface. 2020.



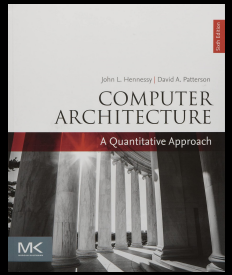
Patterson, Hennessy. Computer Organization and Design MIPS Edition: The Hardware/Software Interface. 2020.



Stallings, W. Organização de Arquitetura de Computadores. 10a Ed. 2016.



Hennessy, Patterson. Arquitetura de Computadores: uma abordagem quantitativa. 2019.



Licença