PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
São Gabriel
Curso de Engenharia de computação

Disciplinas	Turno	Período
Arq. e Org. de Computadores II e Conceitos de Ling. de Programação	Manhã	4°

Professores
Romanelli Lodron Zuim e Hugo de Paula

Trabalho Integrado
Recursividade de cauda

Grupos de até 4 pessoas
Data de entrega: 11/11

1 Informações Gerais

1. Semestre 2/2014
2. Disciplinas Envolvidas:
   - Arquitetura e Organização de Computadores II
   - Laboratório de Arquitetura e Organização de Computadores II
   - Conceitos e Paradigmas de Linguagens de Programação
3. Objetivos Gerais (comuns às diversas disciplinas envolvidas):
   - Contribuir para a aplicação dos conhecimentos obtidos nas disciplinas.
   - Perceber a interação dos conteúdos programáticos das diferentes disciplinas e sua importância para a formação de um engenheiro de computação.
4. Objetivos Específicos (para cada disciplina):
   - **Arquitetura e Organização de Computadores II**: Implementar algoritmos em MIPS e analisar instruções.
   - **Conceitos e Paradigmas de Linguagens de Programação**: Comparar a implementação de algoritmos recursivos na resolução de problemas utilizando diferentes paradigmas de programação.

2 Introdução

A recursividade de cauda é uma técnica de recursão que faz menos uso de memória durante o processo de empilhamento de chamadas, o que a torna mais rápida que a recursão comum. No processo de recursão clássico, a cada chamada recursiva realizada, é necessário guardar a posição do código onde foi feita a chamada, para que a chamada continue dali assim que receber o resultado. Na recursividade de cauda não é necessário guardar a posição onde foi feita a chamada, pois esta é a última operação realizada pela função. O código a seguir apresenta a função fatorial sem recursividade de cauda, escrita na linguagem Scheme:

```
(define fatorial
   (lambda (n)
      (if (= n 1)
         1
         (* n (fatorial (- n 1)))
      )
   )
)

>(fatorial 6)
720
```

A seguir temos a versão da função fatorial com recursividade de cauda.

```
(define fatorial
   (lambda (n resultado)
      (if (= n 1)
         resultado
         (fatorial (- n 1) (* n resultado))
      )
   )
)

>(fatorial 6 1)
720
```

Alguns compiladores possuem otimização especial para funções com recursividade de cauda, transformando-as em um loop. A seguir segue o exemplo do código gerado pela otimização de recursividade de cauda da linguagem Scheme:

```
(define fatorial
   (lambda (n resultado)
      (begin
         (while (> n 1)
            (set! resultado (* n resultado))
            (set! n (- n 1))
         )
         resultado
      )
   )
)

>(fatorial 6 1)
720
```

2.1 Problema a ser resolvido

O trabalho irá abordar a implementação de um algoritmo com recursividade de cauda implementado em diversas linguagens de programação. O aluno irá experimentar a implementação em uma linguagem do paradigma funcional, em uma linguagem de programação imperativa clássica e em uma linguagem de máquina. O objetivo é que o aluno aprenda a analisar as diferenças de implementação e o código em linguagem de montagem gerado pelos respectivos compiladores.

3 Especificação

Suponha uma função chamada ATOI, com semântica semelhante à função ATOI da linguagem C, que converte um vetor de char contendo a representação textual de um número inteiro na base 10 em um valor inteiro armazenado na memória. A assinatura da função atoi pode ser escrita em C da seguinte forma:

```
int atoi(char* str)
```

Em Haskell, a declaração da função seria:

```
atoi :: [Char] -> Int
```

Essa função pode ser implementada recursivamente com, ou sem recursividade de cauda.

3.1 Otimização de código pelo compilador

Em alguns casos, o compilador é capaz de detectar um código que está utilizando recursividade de cauda, e alterar o código objeto resultante, evitando o estouro da pilha de chamada de funções. O GCC é um exemplo de compilador que permite a otimização de código, gerando códigos em linguagem Assembly diferentes, caso a otimização seja executada. Considere o seguinte código em linguagem C que implementa a função fatorial.

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

Ao ser compilado sem nenhuma opção de otimização (GCC - S FATORIAL.C), a função fatorial gerou o seguinte código em linguagem Assembly:

```
        .file   "fatorial.c"
        .text
        .globl _fatorial
        .def    _fatorial;      .scl    2;      .type   32;     .endef
        _fatorial:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        cmpl    $1, 8(%ebp)
        jg      L2
        movl    $1, %eax
        jmp     L3
        L2:
        movl    8(%ebp), %eax
        decl    %eax
        movl    %eax, (%esp)
        call    _fatorial
        imull   8(%ebp), %eax
        L3:
        leave
        ret
```

Em seguida, utilizou-se otimização de código no processo de compilação para retirar a recursividade da função fatorial. O código gerado com a opçãod e otimização (GCC -O2 -S FATORIAL.C) foi:

```
        .file   "fatorial.c"
        .text
        .p2align 2,,3
        .globl _fatorial
        .def    _fatorial;      .scl    2;      .type   32;     .endef
        _fatorial:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %edx
        cmpl    $1, %edx
        jle     L4
        movl    $1, %eax
        jmp     L3
        .p2align 2,,3
        L5:
        movl    %ecx, %edx
        L3:
        leal    -1(%edx), %ecx
        imull   %edx, %eax
        cmpl    $1, %ecx
        jne     L5
        leave
        ret
        L4:
        movl    $1, %eax
        leave
        ret
```

Note que no novo código não há mais a execução do comando CALL _FATORIAL, responsável pela chamada recursiva.

3.2 Atividades

Neste trabalho, cada grupo deverá implementar a função ATOI nas seguintes versões:

1. Versão em Haskell sem recursividade de cauda.
2. Versão em Haskell com recursividade de cauda.
3. Versão em C sem recursividade de cauda.
4. Versão em C com recursividade de cauda.
5. Versão em C utilizando loops.
6. Versão em linguagem de máquina para o microprocessador MIPS.

O grupo deverá compilar os códigos utilizando o GHC para a linguagem Haskell, o GCC com e sem a opção -O2 para a linguagem C. Esses compiladores irão produzir código que deverá ser analisado pelo grupo. Os alunos deverão analisar os tipos de instrução gerados,

Para a análise no MIPS, será necessário construir uma tabela comparativa entre as duas implementações. Para tal, calcule um valor aproximado do número total de instruções considerando que teremos 10 chamadas recursivas e a porcentagem de cada tipo de instrução (ALU, Desvios, Mem, Outras).

Vamos considerar os seguintes CPI´s para cada tipo de instrução:

ALU = 4 ciclos de máquina

Desvios = 2 ciclos de máquina

Mem = 5 ciclos de máquina

Outras = 6 ciclos de máquina

Calcule:

a) O CPI médio para cada programa

b) O *Speedup* de uma implementação sobre a outra

c) O tempo de execução de cada programa considerando que a máquina irá operar em 100 MHz.

Finalmente, o grupo deverá implementar um mecanismo de medição de tempo de CPU para os programas desenvolvidos em Haskell e C. Deverão ser executados testes com todas as funções implementadas. Para permitir validade estatística, cada programa deverá ser executado 10 vezes, e os resultados deverão ser apresentados em uma tabela contendo os tempos mínimo, máximo e médio de cada algoritmo.

4 Resultados Esperados

Para todas as disciplinas, deverá ser entregue os seguintes artefatos:

1. Código de todas as implementações funcionando.
    - Indentação e organização
    - Obedecendo à assinatura proposta
    - Gerando resultados corretos.
2. Relatório com as informações:
    - Descrição dos programas implementados.
    - Apresentação e descrição do código Assembly gerado pelas compilações e na implementação para o MIPS.
    - Análise dos algoritmos segundo os parâmetros estabelecidos na sessão anterior.
    - Uma conclusão comparando as soluções com e sem recursividade de cauda, e apontando as vantagens e desvantagens no que se refere a tempo de processamento e consumo de memória.
3. Apresentação do Trabalho (Todos os alunos do grupo)

Para os alunos que cursam a disciplina **Conceitos de Linguagens de Programação**, deverá ser incluído no relatório:

- Modelagem dos valores e tipos utilizando a notação matemática;
- Cálculo da quantidade de memória teórica e da cardinalidade de cada tipo.

Para os alunos que cursam **Arquitetura e Organização de Computadores II**, deverá ser entregue um arquivo compactado e submetido através do SGA em uma pasta já destinada para este trabalho (pasta: Interd. LP), com os resultados a seguir:

- O código MIPS dos programas (arquivo .asm) capaz de ser executado no PCSPIM ou MARS.
- Um relatório com os cálculos acima solicitados

5 Como o trabalho será avaliado

Para a disciplina de **Arquitetura e Organização de Computadores II**, teórica, esse trabalho irá substituir a terceira lista de exercícios, valendo 5 pontos. Para os alunos que não cursam a disciplina de Conceitos e Paradigmas de LP, apenas a implementação em MIPS é necessária.

Para a disciplina de **Laboratório de AOC2**, conforme o plano de ensino e a distribuição de pontos no SGA, já temos um valor destinado a uma atividade interdisciplinar, este valor será de 5 pontos para este trabalho. O restante dos pontos é para outro trabalho interdisciplinar com a Teoria de AOC2 envolvendo microcontroladores PIC, já anteriormente previsto.

Para a disciplina de **Conceitos e Paradigmas de Linguagens de Programação**, o TI é previsto no plano de ensino valendo 15 pontos. Caso o aluno não curse AOC2, a implementação do MIPS será dispensada. As demais implementações são necessárias.