**Universidade Federal de Santa Catarina - Centro Tecnológico - Departamento de Informática e Estatística**
**INE 5411 - Organização de Computadores**
**Roteiro do Laboratório 5 - Estudo de caso de ordenação**

## 1. Objetivo

O objetivo desta aula é revisar a convenção de chamada de procedimentos do MIPS32 e fazer uso prático dela através de um estudo de caso. O estudo de caso proposto é a implementação de um algoritmo de ordenação de números inteiros. O algoritmo escolhido faz uso de dois procedimentos:

- `swap`: troca de posição dois elementos de um arranjo de inteiros;
- `sort`: ordena um arranjo de inteiros no estilo do método "Bubble Sort".

O procedimento `sort` invoca `swap`, que é um procedimento-folha. Como veremos, essa diferença de invocação tem consequências na alocação de registradores e, consequentemente, no uso da convenção de chamada. Para a geração de código de cada um desses procedimentos, precisam ser observar os seguintes passos:

1. Alocar registradores para as variáveis;
2. Produzir o código para o corpo do procedimento;
3. Preservar registradores através da invocação do procedimento.

Os Passos 1 e 3 precisam obedecer à convenção de chamada. A alocação correspondente ao Passo 1 deve levar em conta a declaração de variáveis do código-fonte e o fato de um procedimento ser folha ou não para decidir se uma variável deve ser alocada em registrador temporário (`$t0, $t1, ...`) ou em registrador salvo (`$s0, $s1, ...`). O Passo 3 deve levar em conta se um registrador foi usado no corpo do procedimento (Passo 2) e se ele é temporário ou salvo para decidir se tal registrador precisa ser preservado ou não.

## 2. Revisão da convenção de chamada de procedimentos

### Alocação de valores em registradores

- **Registradores de argumento** (`$a0-$a3`): devem armazenar os quatro primeiros argumentos de um procedimento (os demais são armazenados na pilha).
- **Registrador**(es) **de valor** (`$v0-$v1`): devem armazenar os valores de retorno de funções.
- **Registradores temporários** (`$t0-$t9`): devem armazenar preferencialmente valores temporários com curto tempo de vida, que geralmente não são necessários após uma chamada de procedimento, podendo ser usados no procedimento invocado sem a necessidade de preservação por este último.
- **Registradores salvos** (`$s0-$s7`): devem armazenar valores com longos tempos de vida que geralmente são necessários após uma chamada de procedimento, devendo por isso ser preservados através dela.

### Divisão de responsabilidades

- Procedimento **chamador**: é responsável por preservar os registradores `$a0-$a1` e `$t0-$t9`. Devem ser preservados apenas os registradores cujos valores serão usados no corpo do procedimento chamador depois da chamada.
- Procedimento **chamado:** é responsável por preservar os registradores `$s0-$s7`. Devem ser preservados apenas os registradores cujos valores serão usados no corpo do procedimento chamado. Deve também preservar o registrador `$ra`, exceto se for um procedimento-folha.

### Salvamento e restauração de valores preservados

- Procedimento **chamador**: os conteúdos de registradores `$a0-$a1` ou `$t0-$t9` a serem preservados são normalmente copiados em registradores salvos (`$s0-$s7`), pois estes são preservados através de chamadas (em geral isso é feito apenas para os registradores `$a0-$a1`, pois se houvesse a necessidade de preservar registradores `$t0-$t9` seria mais eficiente ter alocado os conteúdos diretamente em registradores salvos). O salvamento deve ser feito antes de se efetuar a chamada. Alternativamente, tais conteúdos podem ser salvos na pilha.
- Procedimento **chamado**: o conteúdo dos registradores `$s0-$s7` ou `$ra` a serem preservados são armazenados na pilha. O salvamento deve ser feito antes de se iniciar o corpo do procedimento (ou seja, antes dos valores serem alterados). A restauração deve ser feita antes de retornar ao procedimento chamador.

## 3. Procedimento 1: swap

O código abaixo descreve o procedimento, escrito em linguagem C. O primeiro argumento é o endereço-base do arranjo e o segundo é o valor do índice do elemento a ser trocado com seu sucessor.

```
void swap ( int v[], int k )
{
        int temp;
        temp = v[k];
        v[k] = v[k+1];
        v[k+1] = temp;
}
```

## 4. Procedimento 2: sort

O código seguinte descreve o procedimento escrito em linguagem C. O primeiro argumento é o endereço-base do arranjo e o segundo é o seu número total de elementos.

```
void sort(int v[], int n)
{
        int i, j;
        for ( i = 0 ; i < n ; i = i + 1 )
        {
                for ( j = i – 1 ; j >= 0 && v[j] > v[j+1] ; j = j - 1 )
                {
                        swap(v,j);
                }
        }
}
```

O arranjo `v[]` será armazenado nas 10 primeiras palavras da área de dados globais da memória. A décima primeira palavra é o tamanho `n` do arranjo. Aplique o procedimento abaixo para verificar o correto funcionamento de `sort`.

```
# Estímulos: v[] = [9, 8, 7, 6, 5, 4, 3, 2, 1,-1] e n = 10
.data
_v: .word 9,8,7,6,5,4,3,2,1,-1
_n: .word 10
```

**Resultado esperado: v[] = [-1, 1, 2, 3, 4, 5, 6, 7, 8, 9]**

## 5. Experimentos

Os experimentos desta aula prática vão induzir propositadamente alguns **erros típicos** na codificação de procedimentos. Partiremos de um **código que contém erros** e vamos eliminá-los gradativamente até que o resultado esperado seja alcançado. O arquivo fornecido (`experimento1-codigo-base.asm`) contém uma versão inicial da implementação dos procedimentos `swap` e `sort`. Entretanto, essa implementação **não funciona corretamente**: quando o programa é executado, observa-se um resultado diferente do esperado. Recomenda-se fortemente que, antes de abrir o relatório, você faça um estudo comparativo entre o código *assembly* fornecido naquele arquivo e o código-fonte das rotinas `swap` e `sort` mostradas neste roteiro.

O relatório vai propor **6 experimentos** a partir da versão inicial fornecida, solicitando modificações e fazendo perguntas sobre a execução de versões intermediárias que você vai produzir. Por isso, faça **apenas as alterações solicitadas** em cada experimento, interprete os resultados anômalos e responda as perguntas. **Não faça qualquer outra modificação** no código (além das solicitadas no relatório). Cada **nova versão** deverá ser salva com um **nome diferente** (`experimento2.asm` , …, `experimento6.asm`) para que você consiga reverter para uma versão anterior com segurança, caso você precise repetir o experimento anterior (por ter percebido algum engano seu num experimento anterior). Você pode renomear arquivos dentro do MARS, usando a opção “File → Save as ...”.

Em alguns experimentos, você precisará usar *breakpoints* para pausar a execução em um determinado ponto do programa e observar os resultados até aquele ponto. O MARS disponibiliza um recurso para inserir *breakpoints* na primeira coluna da janela “Text Segment”, a qual se abre logo depois que um programa é montado. Basta você selecionar a(s) instrução (ões) onde quer que o programa pare durante a execução. Por favor, familiarize-se com esse recurso antes da aula de laboratório, montado o arquivo fornecido (`experimento1-codigo-base.asm`) e treinando o uso desse recurso.