Arquitetura MIPS
Suporte a procedimentos e pilha

Tópicos da aula

- **Suporte a procedimentos**
- **Pilha**

Suporte à procedimentos

- **Por que usar procedimentos?**
  - Facilita o entendimento do programa
  - Possibilita o reuso do código
- **Benefícios ao programador**
  - Ele se concentra em uma única parte do código
  - Os parâmetros funcionam como uma barreira entre o procedimento e o resto do programa e os dados

Suporte à procedimentos

- **Analogia: Espião**
  - O espião recebe as informações e os recursos necessários para a execução de um plano secreto
    - Espião = procedimento
    - Informações = parâmetros
    - Recursos = espaço em memória p/ as variáveis locais
  - Excuta a tarefa sem deixar rastros
    - O espaço ocupado pelas variáveis locais é liberado após a execução do procedimento
  - Retorna com os resultados esperados
    - Resultados = valores de retorno

Suporte à procedimentos

- **Analogia: Espião**
  - Um espião pode contratar um outro espião sem que o cliente saiba da existência do mesmo
  - Ou seja, um procedimento pode chamar outro procedimento sem o conhecimento do "chamador" (função principal)

Suporte à procedimentos

- **Passos para a execução de um procedimento**
  1. Colocar os dados em um lugar acessível ao procedimento (registradores de argumento `$a0-$a3`)
  2. Transferir o controle ao procedimento
  3. Garantir recursos de memória ao procedimento
  4. Realizar a tarefa
  5. Colocar o resultado em um lugar acessível ao "chamador" (registradores de retorno `$v0-$v1`)
  6. Retornar o controle ao ponto de origem (registrador com o endereço de retorno `$ra`)

Suporte à procedimentos

- **Chamada do procedimento**

```
jal      end_do_proc
```

  - Desvia para o endereço do procedimento fazendo
    `$ra ← PC+4` (link para o endereço de retorno)
    `PC ← end_do_proc`

- **Retorno do procedimento**

```
jr       $ra
```

  - Retorna ao chamador fazendo
    `PC ← $ra` (link para o endereço de retorno)

Suporte à procedimentos

- **Resumo**
  - Carrega os argumentos (1 a 4)
    - `$a0 ← arg0` `$a1 ← arg1`
    - `$a2 ← arg2` `$a3 ← arg3`
  - Chama o procedimento
    - `jal  end_do_proc`
  - Realiza o processamento
  - Carrega os valores de retorno (1 a 2)
    - `$v0 ← val0` `$v1 ← val1`
  - Retorna ao procedimento chamador
    - `jr  $ra`

Suporte à procedimentos

- **Problema**
  - Como trabalhar com mais do que 4 argumentos ou 2 valores de retorno se se dispõe apenas de `$a0-$a3` e `$v0-$v1`?
- **Solução**
  - Usar a memória para aumentar "virtualmente" a quantidade de registradores

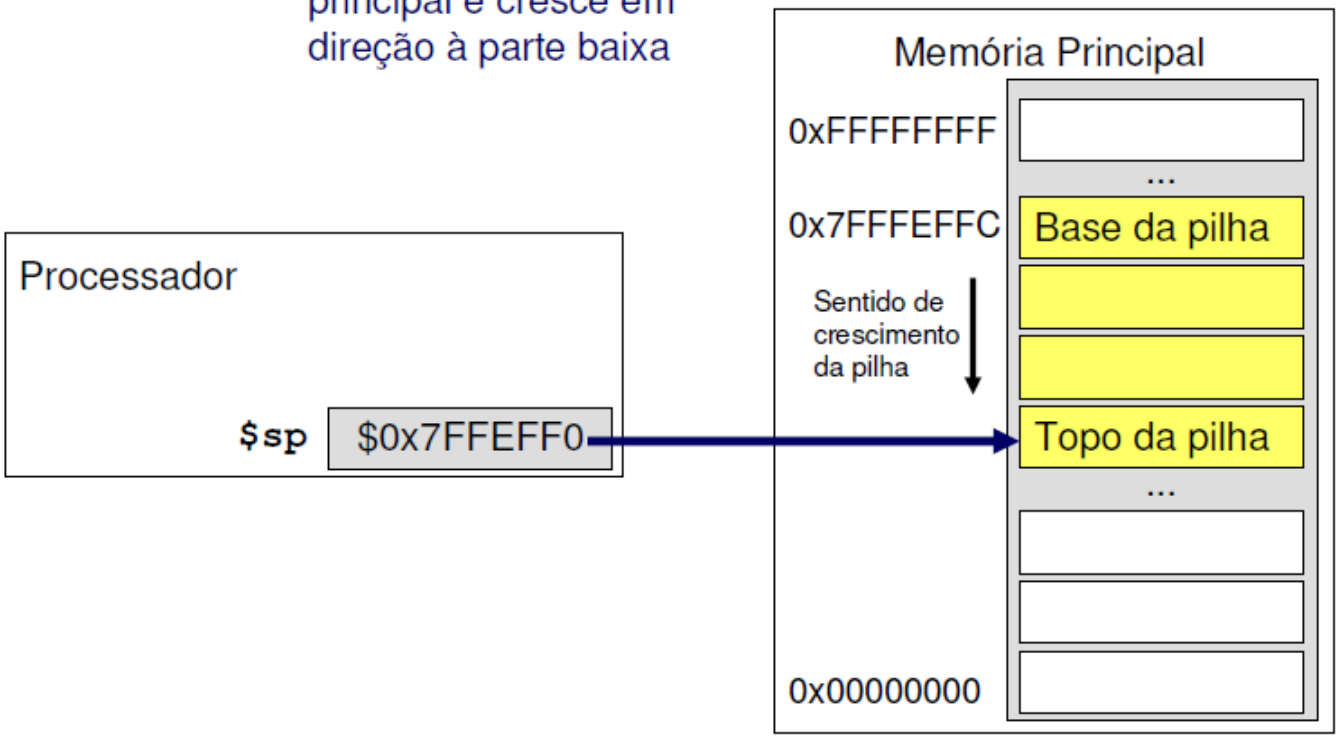
Pilha

- **Definição de pilha**
  - estrutura de dados do tipo LIFO (*Last-In, First-Out*) armazenada na memória principal
  - acessada através de dois tipos básicos de ações
    - **Push**: coloca uma palavra no topo da pilha
    - **Pop**: retira uma palavra do topo da pilha
  - o topo da linha é indicado pelo registrador `$sp` (*stack pointer*)

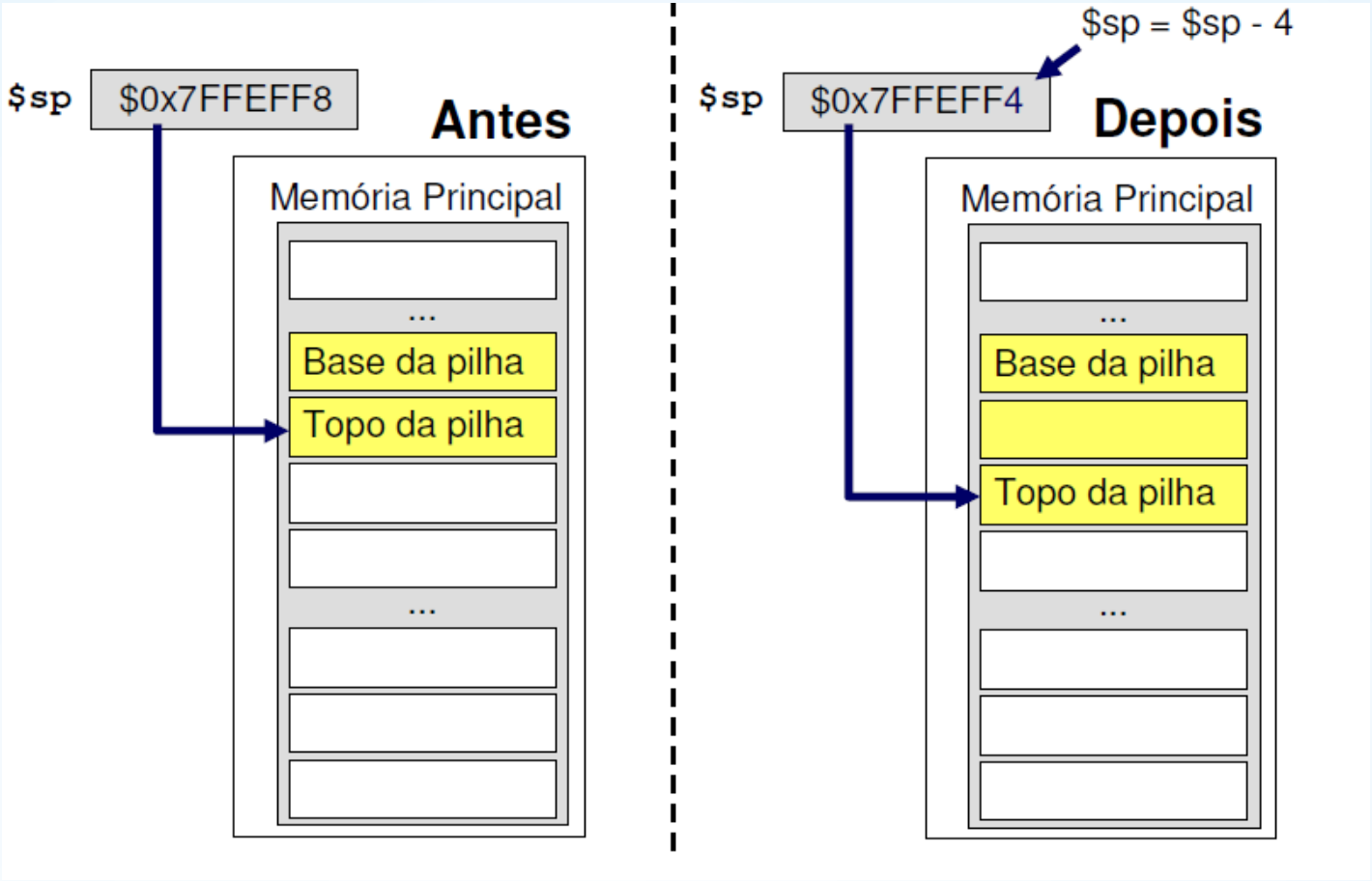
O MIPS não possui instruções push e pop (outros processadores sim). Então ele requer que programador implemente essas ações

Pilha

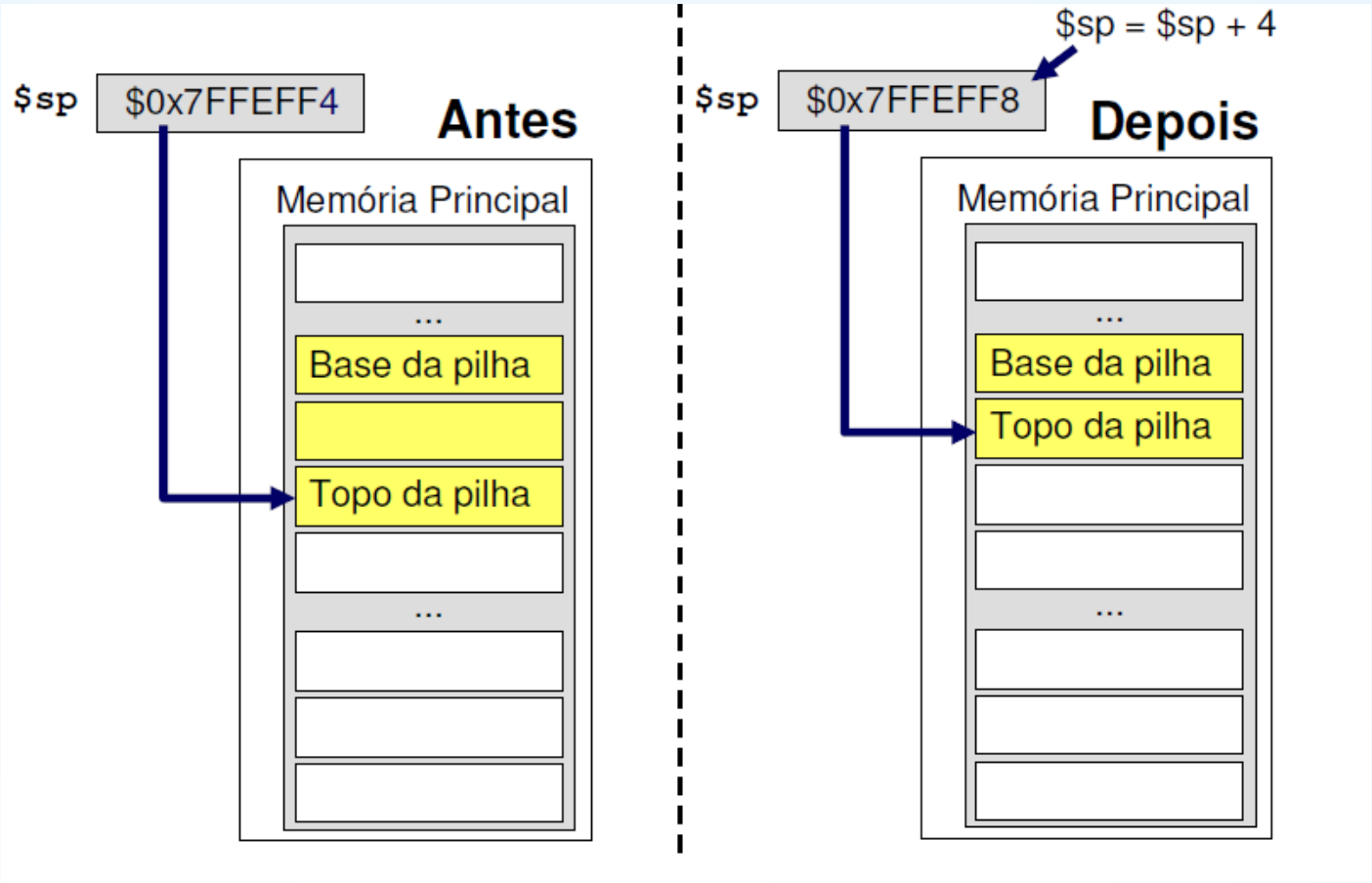
- **A pilha e a memória principal**
  - A pilha é situada em uma região na parte alta da memória principal e cresce em direção à parte baixa



Pilha: operação push (decrementa $sp)



Pilha: operação pop (incrementa $sp)



Pilha: exemplo

- **Compilação de um programa que não chama outro procedimento (procedimento folha)**
  - Código em C

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

Pilha: exemplo

- **Compilação de um programa que não chama outro procedimento (procedimento folha)**
  - Código em ASM

Onde:
g => $a0
h => $a1
i => $a2
j => $a3
f => $s0

```
leaf_example:
```

push

```
addi $sp,$sp,-12   # decrementa $sp em 3
sw   $t1, 8($sp)   # 3 palavras e salva
sw   $t0, 4($sp)   # na pilha os
sw   $s0, 0($sp)   # regs. a serem modificados
```

```
add  $t0,$a0,$a1
add  $t1,$a2,$a3
sub  $s0,$t0,$t1
add  $v0,$s0,$zero # copia $s0 para $v0
```

pop

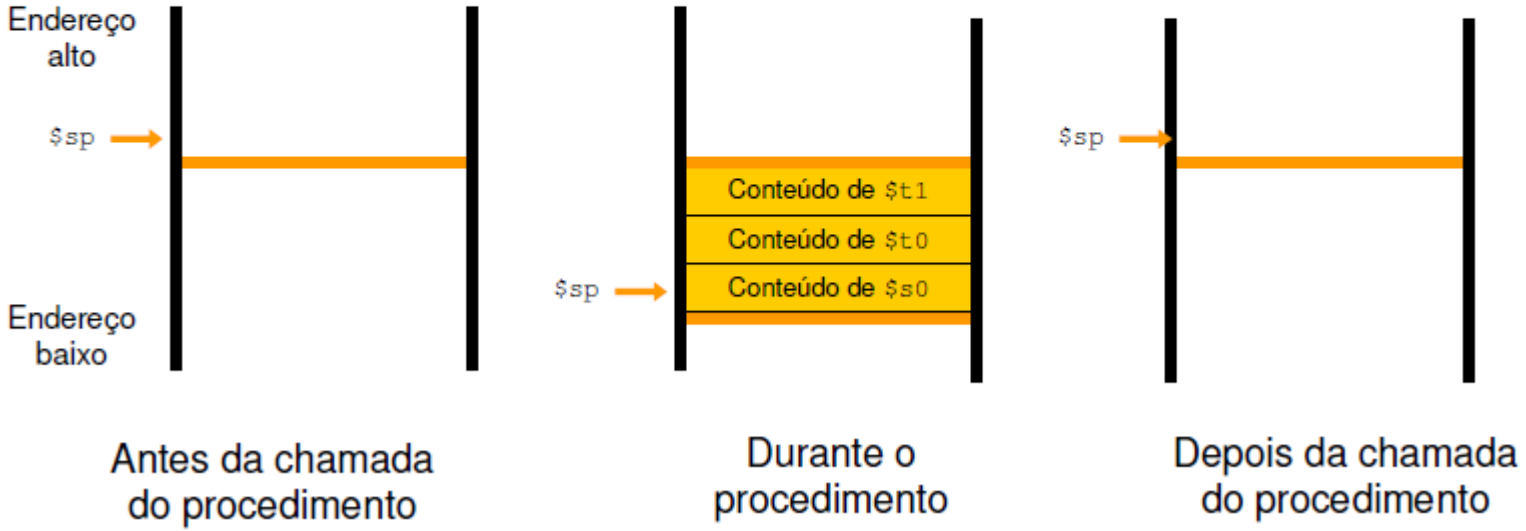
```
lw   $s0, 0($sp)   # restaura, para o chamador,
lw   $t0, 4($sp)   # os registradores armazenados
lw   $t1, 8($sp)   # na pilha ($s0,$t0,$t1) e
addi $sp,$sp,12    # incrementa $sp em 3 palavras
```

```
jr   $ra           # volta para o "chamador"
```

Pilha: exemplo

```
# PUSH
addi $sp,$sp,-12
sw   $t1, 8($sp)
sw   $t0, 4($sp)
sw   $s0, 0($sp)
```

```
# POP
lw   $s0, 0($sp)
lw   $t0, 4($sp)
lw   $t1, 8($sp)
addi $sp,$sp,12
```



Classes de registradores

- **Para evitar gasto com instruções de acesso a pilha, consideram-se duas classes de registradores de uso geral**
  - `$t0-$t9`
    - 10 registradores temporários que não são preservados pelo procedimento chamado quando de uma chamada de procedimento
  - `$s0-$s7`
    - 8 registradores de salvamento que precisam ser preservados quando de uma chamada de procedimento.
    - O procedimento deve salvar e restaurar apenas os registradores modificados.

Ou seja, no exemplo anterior não seria necessário salvar e restaurar os registradores `$t0` e `$t1`.

Procedimentos aninhados

- **Procedimentos não-folha**
  - Procedimentos que chamam outros procedimentos
  - Deve-se ter cuidado redobrado ao chamar procedimentos não-folha
- **Problema**
  - Ao chamar um procedimento, o chamador modifica os registradores de argumento (`$a0-$a3`) e de endereço de retorno (`$ra`)
  - Porém, esse procedimento chama outro procedimento e também modifica os registradores de argumento (`$a0-$a3`) e de endereço de retorno (`$ra`)
  - Isso produz um conflito, pois o argumentos do primeiro procedimentos serão perdidos, assim como o endereço de retorno do chamador

Procedimentos aninhados

- **Solução**
  - Colocar na pilha todos os registradores que precisam ser preservados
  - O procedimento chamador deve colocar na pilha os seus registradores de argumento (`$a0-$a3`) e os temporários por ele utilizados (`$t0-$t9`)
  - O procedimento chamado deve colocar na pilha o registrador do endereço de retorno (`$ra`) e os registradores de salvamento por ele utilizados (`$s0-$s7`)

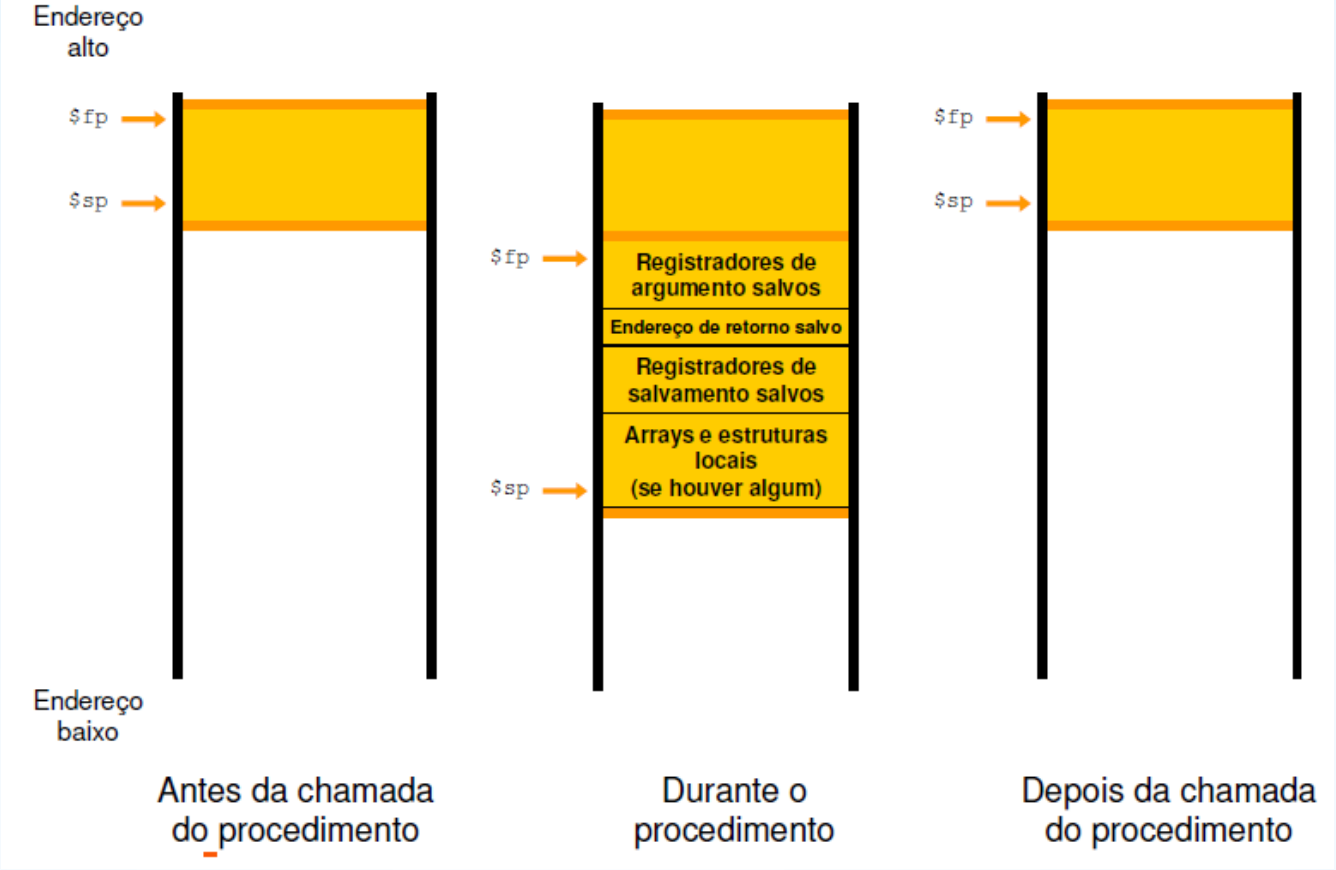
Procedimentos aninhados

- **O que é e o que não é preservado quando de uma chamada a procedimento**
  - Preservados pelo procedimento chamado
    - Registradores de salvamento → `$s0-$s7`
    - Registrador de endereço de retorno → `$ra`
    - Registrador stack pointer → `$sp`
    - Pilha acima do stack pointer
  - Não preservados pelo procedimento chamado
    - Registradores temporário → `$t0-$t9`
    - Registradores de argumento → `$a0-$a3`
    - Registradores de retorno de valores → `$v0-$v1`
    - Pilha abaixo do stack pointer

Alocação de espaço para novos dados

- A pilha também é usada para guardar variáveis locais ao procedimento, que, por serem numerosas, não podem ser armazenadas nos registradores (ex: arrays locais)
- Os registradores e variáveis locais do procedimento são armazenados em um segmento da pilha denominado
  - Quadro do procedimento ou
  - Registro de ativação
- O endereço inicial do quadro de ativação do procedimento é indicado pelo registrador `$fp` (*frame pointer*)

Alocação de espaço para novos dados



Alocação de espaço para novos dados

- **Classes de armazenamento de variáveis em C**
  - **Automática**
    - São locais a um procedimento
    - São descartadas quando o procedimento termina
  - **Estática**
    - Sobrevivem ao procedimento
- **Mais sobre a variáveis estáticas**
  - Variáveis declaradas fora dos procedimentos são, por *default*, estáticas
  - Variáveis também podem ser declaradas como estáticas utilizando-se a palavra-chave *static*
  - Para simplificar o acesso aos dados estáticos, há um registrador denominado $gp (*global pointer*)

FIM MÓDULO 08