UNIVALI

Laboratório de Sistemas Embarcados e Distribuídos

Conjunto de Instruções

Parte IV - Programação de Procedimentos

Histórico de revisões

Revisão	Data	Responsável	Descrição
0.1	-x-	Prof. Cesar Zeferino	Primeira versão
0.2	03/2017	Felipe Viel	Revisão do modelo
0.3	09/2022	Thiago Felski Pereira	Atualização de arquitetura
0.4	04/2023	Thiago Felski Pereira	RISC-V64 para RISC-V32

Observação: Este material foi produzido por pesquisadores do Laboratório de Sistemas Embarcados e Distribuídos (LEDS – Laboratory of Embedded and Distributed Systems) da Universidade do Vale do Itajaí e é destinado para uso em aulas ministradas por seus pesquisadores.

Introdução

- **Objetivo**
  - Saber implementar procedimentos e usar a pilha no RISC-V
- **Conteúdo**
  - Suporte a procedimentos
  - Pilha

Introdução

- **Bibliografia**
  - PATTERSON, David A.; HENNESSY, John L. Abstrações e tecnologias computacionais. *In*: _____. **Organização e projeto de computadores**: a interface hardware/software. 4. ed. Rio de Janeiro: Campus, 2014. cap. 2. Disponível em: <http://www.sciencedirect.com/science/book/9788535235852>. Acesso em: 13 mar. 2017.
  - Edições anteriores
    - Patterson e Hennessy (2005, cap. 3)
    - Patterson e Hennessy (2000, cap. 3)
  - NECESSITA ATUALIZAÇÃO

1 Suporte a 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

1 Suporte a 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

1 Suporte a procedimentos

- **Analogia: Espião**

- **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)

1 Suporte a procedimentos

Passos para a execução de um procedimento

1. Colocar os dados em um lugar acessível ao procedimento (registradores de argumento `a0-a7`)
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 `a0-a1`)
6. Retornar o controle ao ponto de origem (registrador com o endereço de retorno `ra`)

1 Suporte a 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

```
jalr     ra, 0
```

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

1 Suporte a procedimentos

Resumo

1. Carrega os argumentos (1 a 4)
   - `a0 ← arg0  a1 ← arg1  a2 ← arg2  a3 ← arg3`
   - `a4 ← arg4  a5 ← arg5  a6 ← arg4  a7 ← arg5`
2. Chama o procedimento
   - `jal  end_do_proc`
3. Realiza o processamento
4. Carrega os valores de retorno (1 a 2)
   - `a0 ← val0  a1 ← val1`
5. Retorna ao procedimento chamador
   - `jalr  ra, 0`

1 Suporte a procedimentos

- **Problema**
  - Como trabalhar com mais do que 8 argumentos ou 2 valores de retorno se se dispõe apenas de `a0-a7` e `a0-a1`?

- **Solução**
  - Usar a memória para aumentar “virtualmente” a quantidade de registradores

2 Pilha

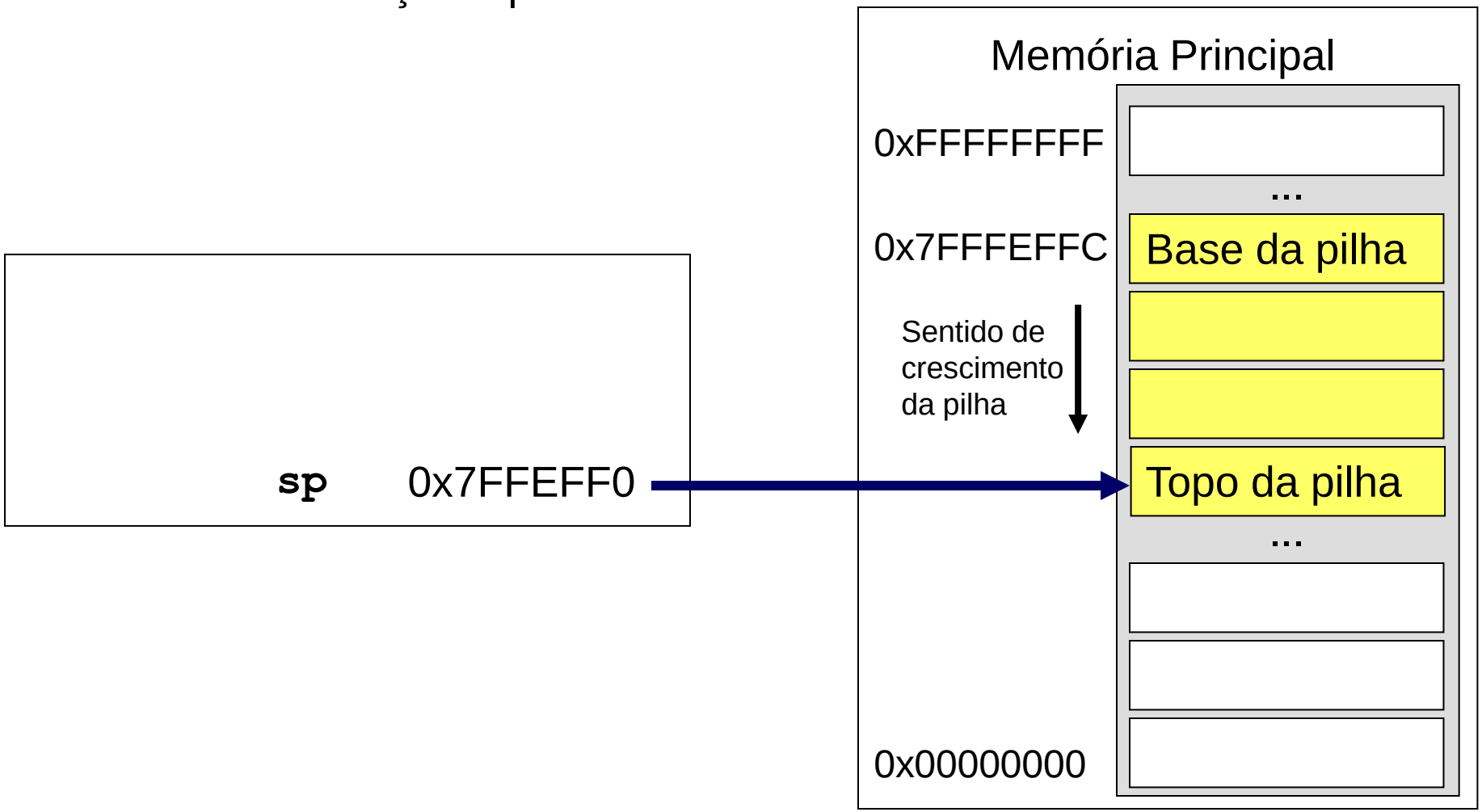
- **Definição de pilha**
  - estrutura de dados do tipo LIFO (*Last-In, First-Out*) armazenada na memória principal
  - acessada através de dois tipos básicos de ações
    - **Push**: coloca uma palavra no topo da pilha
    - **Pop**: retira uma palavra do topo da pilha
  - o topo da pilha é indicado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

Há arquiteturas (ex: x86) que possuem instruções push e pop

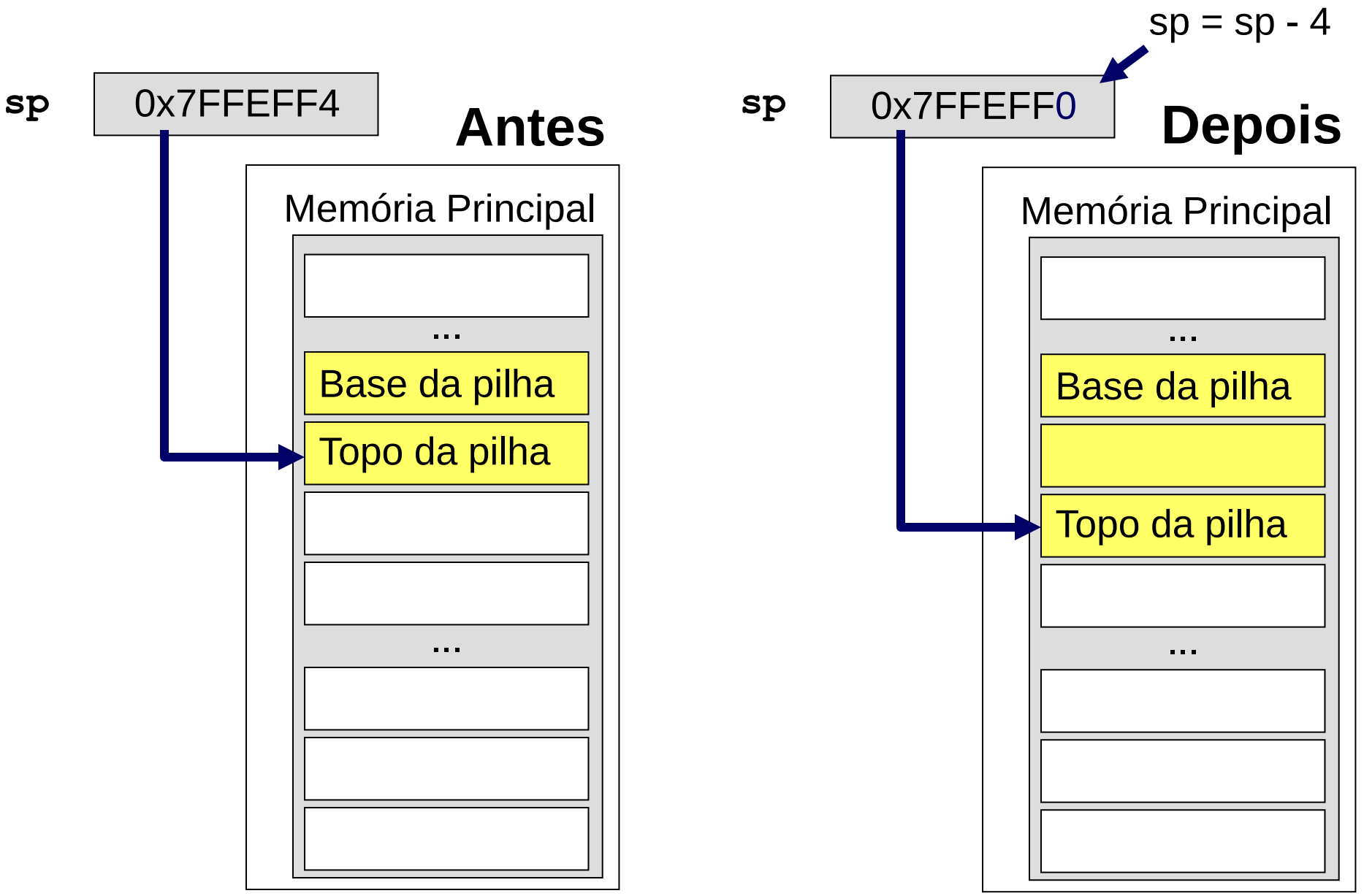
O RISC-V não possui instruções push e pop (outros processadores sim). Então ele requer que programador implemente essas ações

2 Pilha

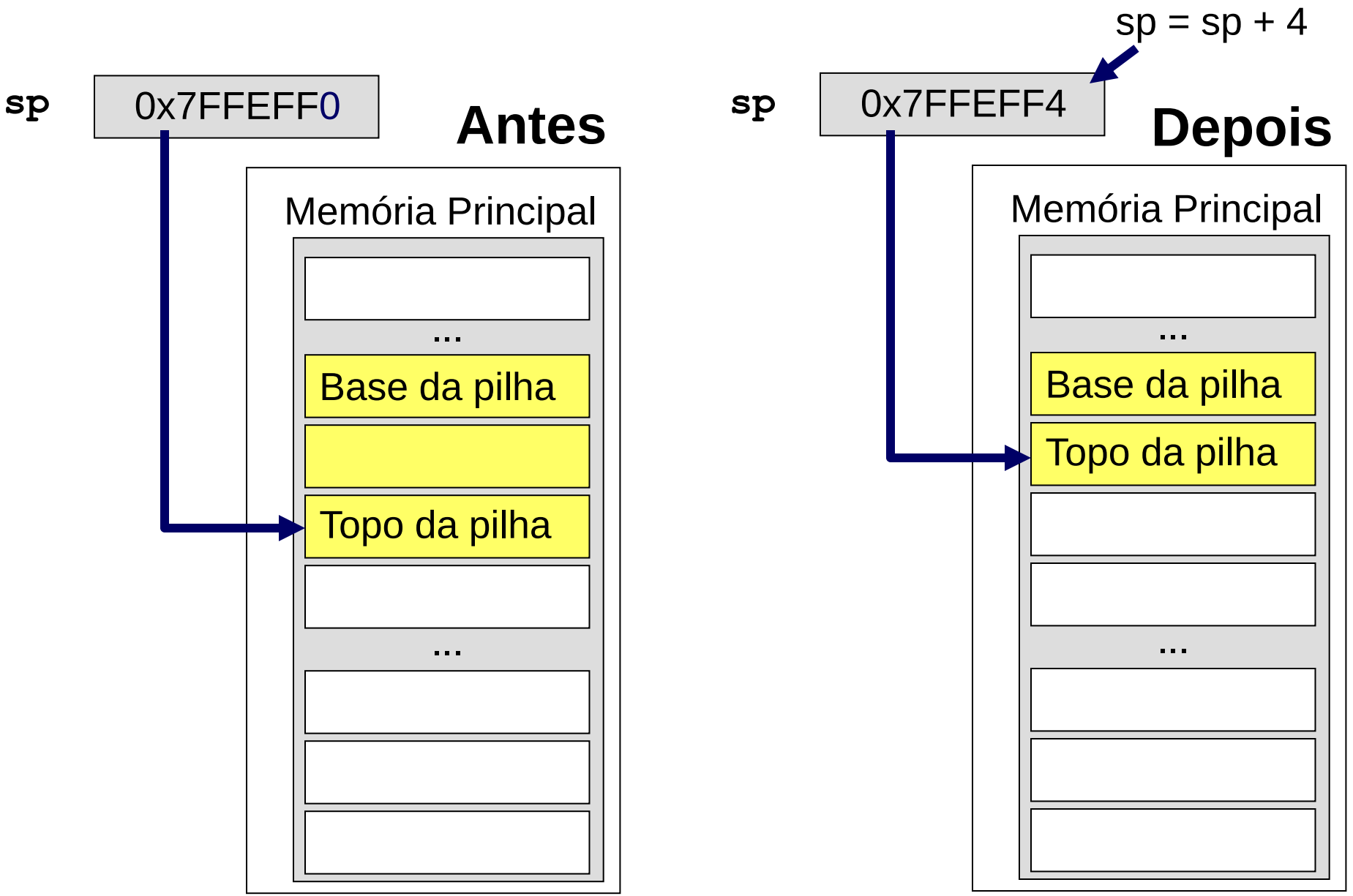
- **A pilha e a memória principal**
  - A pilha é situada em uma região na parte alta da memória principal e cresce em direção à parte baixa



2 Pilha



2 Pilha



2 Pilha

- **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;
}
```

2 Pilha

- **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  a0, s0, zero   # copia s0 para a0
```

pop

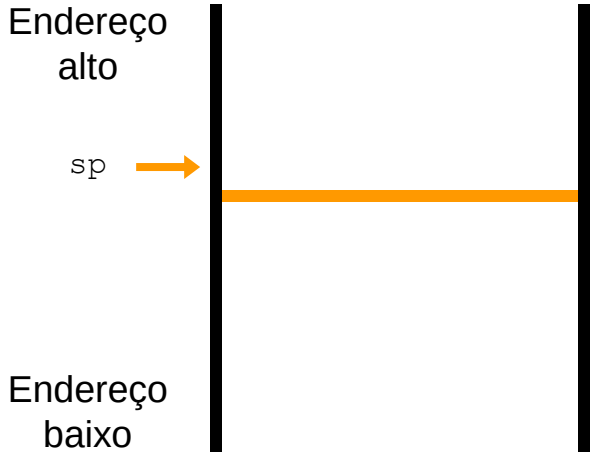
```
lw   s0,  0(sp)     # restaura, para o chamador,
lw   t0,  4(sp)     # os registradores armazenados
lw   t1, 8(sp)      # na pilha (s0,t0,t1) e
addi sp, sp, 12     # incrementa sp em 3 palavras duplas
```

```
jalr   ra, 0        # volta para o "chamador"
```

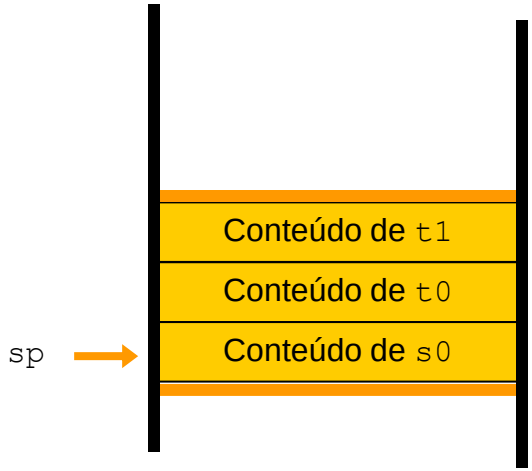
2 Pilha


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

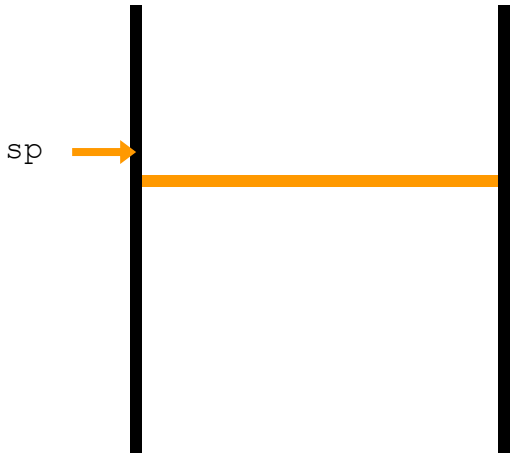
```
# POP
ld   s0,  0(sp)
ld   t0,  4(sp)
ld   t1,  8(sp)
addi sp, sp, 12
```



Antes da chamada do procedimento



Durante o procedimento



Depois da chamada do procedimento

A instrução `sub` não lida com constantes e no RISCV não existe a instrução `subi`. Por isso deve ser usado o `addi` com uma constante negativa. Contudo, destaca-se que o montador do PCSpim converteria `sub sp,sp,12` em `addi sp,sp,-12`.

3 Classe de registradores

- **Para evitar gasto com instruções de acesso a pilha, consideram-se duas classes de registradores de uso geral**
  - `t0-t6`
    - 07 registradores temporários que não são preservados pelo procedimento chamado quando de uma chamada de procedimento
  - `s0-s11`
    - 12 registradores de salvamento que precisam ser preservados quando de uma chamada de procedimento
    - O procedimento deve salvar e restaurar apenas os registradores modificados
- **Quantos menos instruções são executadas, menos energia é gasta**

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

4 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-a7`**) e de endereço de retorno (**`ra`**)
- Porém, esse procedimento chama outro procedimento e também modifica os registradores de argumento (**`a0-a7`**) 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

4 Procedimentos aninhados

Solução

- Colocar na pilha todos os registradores que precisam ser preservados
- Procedimento principal (MAIN)
  - Seus registradores de argumento (**a0-a7**) e os temporários por ele utilizados (**t0-t6**)
- Procedimento chamado do tipo NÃO-FOLHA
  - Registradores de salvamento por ele utilizados (**s0-s11**) e antes de chamar outro procedimento,
    - os seus registradores de argumento (**a0-a7**)
    - os temporários por ele utilizados (**t0-t6**)
    - o registrador do endereço de retorno (**ra**)
- O procedimento chamado do tipo FOLHA deve colocar na pilha os registradores de salvamento por ele utilizados (**s0-s11**)

4 Procedimentos aninhados: Exemplo

- **Compilação de um procedimento recursivo mostrando a ligação entre procedimentos aninhados (não-folha)**
  - Código em C do fatorial

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

4 Procedimentos aninhados: Exemplo - Fatorial

- Código em ASM

Salva ra e a2 na pilha

Testa se n<1

Se n<1, retorna 1 e elimina 2 itens da pilha

Se n>=1, chama fact(n-1)

Restaura $ra e a2 da pilha

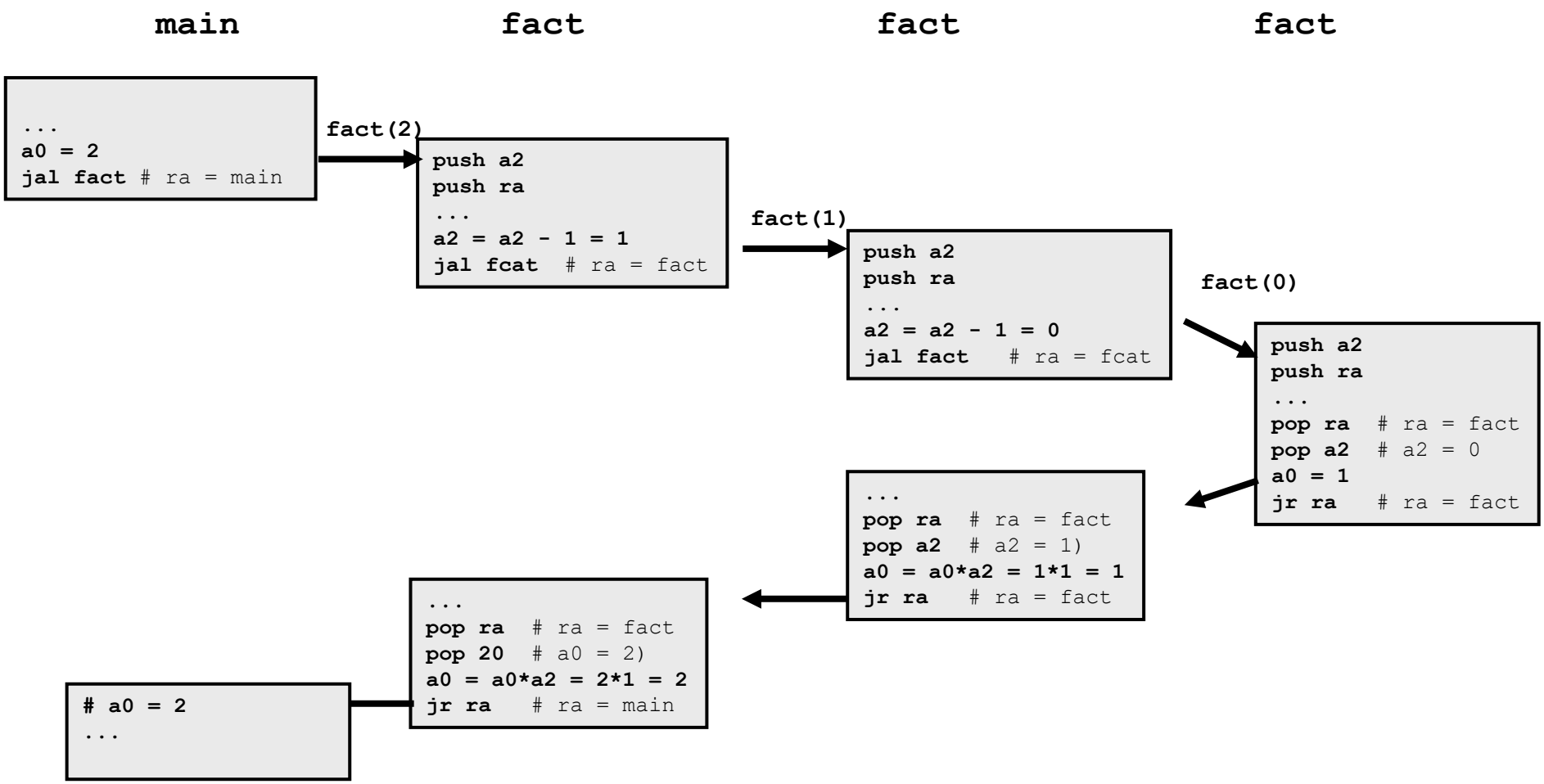
Retorna n*fact(n-1)

```
fat_rec:
        addi  sp,sp,-8          # atualiza sp (push 2 itens)
        sw    ra,0(sp)          # - push ra
        sw    a0,4(sp)          # - push a0
        addi  t0,zero,1         # testa se n<1
        bgt   a0,t0,ret_n1      # se n>1, desvia para ret_n1
        addi  a1,zero,1         # retorna o valor 1
        jalr  ra, 0             # retorna para depois de jal

ret_n1:
        addi  a0,a0,-1          # n>=1, a0 = n-1
        jal   fat_rec           # chama fat_rec(n-1)
        lw    a0,4(sp)          # - pop a2  (ou seja, n)
        lw    ra,0(sp)          # - pop ra
        addi  sp,sp,8           # atualiza sp (pop 2 itens)
        mul   a1,a1,a0          # retorna n*fact(n-1)
        jalr    ra, 0           # retorna para o chamador
```

4 Procedimentos aninhados: Exemplo - Fatorial

- Exemplo de execução do fatorial: fact(2)



4 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-s11`
    - Registrador de endereço de retorno `ra`
    - Registrador stack pointer `sp`
    - Pilha acima do stack pointer
  - Não preservados pelo procedimento chamado
    - Registradores temporário `t0-t6`
    - Registradores de argumento `a0-a7`
    - Registradores de retorno de valores `a0-a1`
    - Pilha abaixo do stack pointer

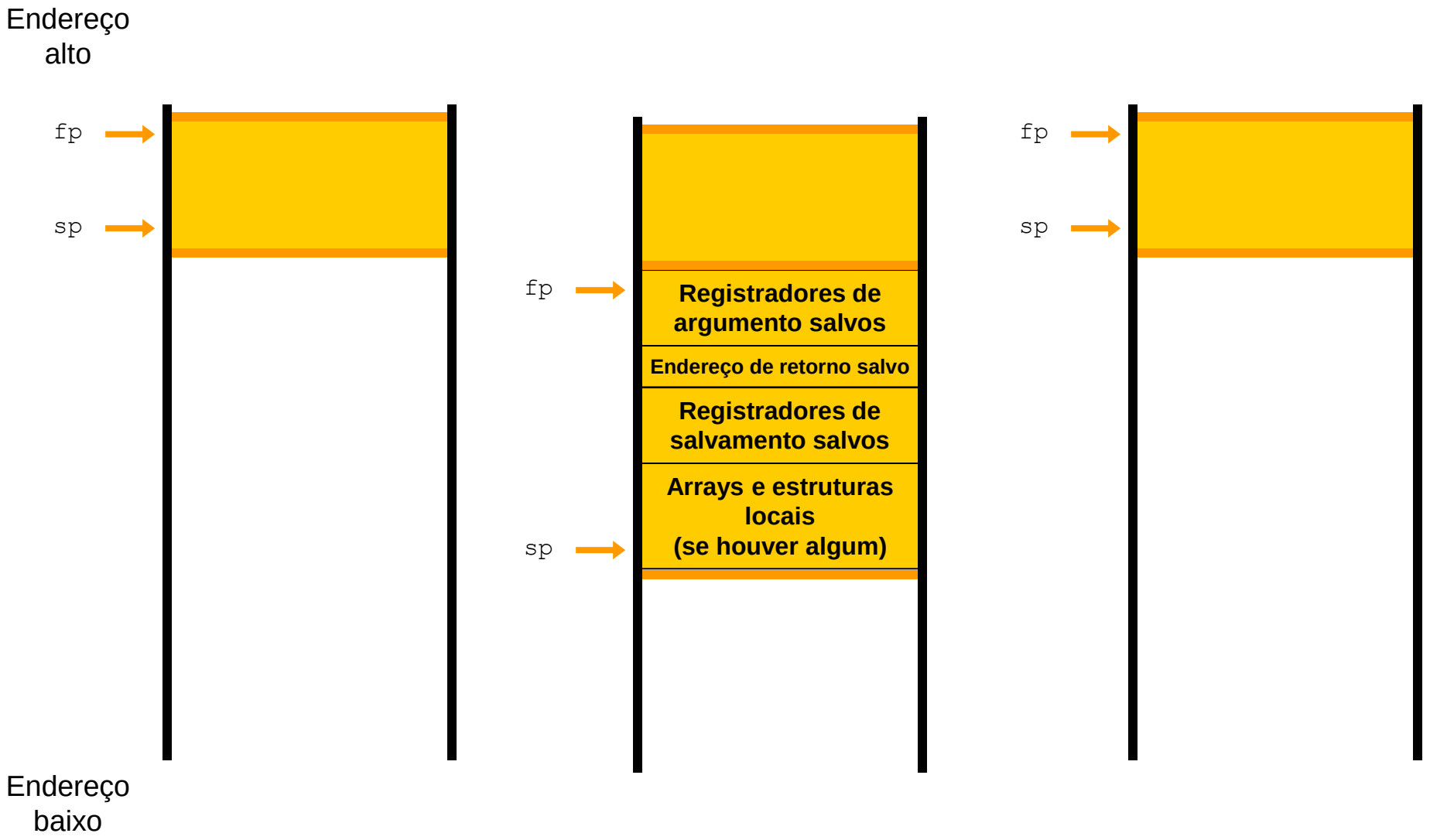
5 Alocação de espaço para novos dados

- **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*)**

5 Alocação de espaço para novos dados



Antes da chamada do procedimento

Durante o procedimento

Depois da chamada do procedimento

Resumo

Arquitetura estudada do RISC-V

- Registradores

`s0-s11`	salvamento
`t0-t6`	temporários
`zero`	constante 0
`a0-a7`	argumentos
`a0-a1`	valores de retorno
`sp`	stack pointer
`ra`	endereço de retorno
`fp`	frame pointer
`gp`	global pointer

- Memória: 2^{32} palavras de 32 bits
- Formatos de instrução: R, I, S, B, U e J
- Modos de endereçamento: via registrador, via registrador-base, relativo ao PC, pseudodireto e imediato
- Instruções: `add`, `sub`, `addi`, `ld`, `sd`, `sll`, `srl` `and`, `andi`, `or`, `ori`, `xor`, `xori` `nor`, `not`, `beq`, `bne`, `bge`, `blt`, `j`, `slt`, **`slti`**, **`mult`**, **`jal`** e **`jr`**

Resumo

Endereços dos registradores

End	Nome	End	Nome	End	Nome	End	Nome
0	zero	8	s0	16	a6	24	s8
1	ra	9	s1	17	a7	25	s9
2	sp	10	a0	18	s2	26	s10
3	gp	11	a1	19	s3	27	s11
4	tp	12	a2	20	s4	28	t3
5	t0	13	a3	21	s5	29	t4
6	t1	14	a4	22	s6	30	t5
7	t2	15	a5	23	s7	31	t6

Exercício

- **Implemente um procedimento que determine o máximo divisor comum entre dois números**

- **Requisitos**
  - O procedimento deve ser chamado por uma função principal que solicite ao usuário a entrada dos dois números, faça a chamada ao procedimento e imprima no console o valor retornado pelo procedimento.

  - A entrada dos dois números deve ser feita meio das chamadas de sistema (ecall).

Exercício

- **Abaixo é apresentado um exemplo de procedimento em linguagem C que determina o máximo divisor comum entre dois números**

```
int proc_mdc(int x, int y) {
  while (x != y) {
    if (x < y)
      y = y - x;
    else
      x = x - y;
  }
  return x;
}
```