Arquitetura de Computadores

Instruction Set Architecture (ISA)

UFMT

Fevereiro de 2025

Agenda

- Introdução.
- Programa Armazenado e Modelo de Von Newman.
- Arquiteturas CISC *versus* RISC.
- MIPS:
    1. Visão geral.
    2. Assembly *vs* Linguagem de Máquina.
    3. Conjunto de Instruções.

Parte 1

    4. Programação.
    5. Modos de Endereçamento.
    6. Compilação, Montagem, Ligação e Carga.
    7. Pseudo-instruções.
    8. Exceções.
    9. Números em Ponto Flutuante.

Parte 2

Caracteres e Código ASCII

Caracteres e o Código ASCII

- Números no intervalo [–128, 127] podem ser armazenados em um único byte em vez de uma palavra inteira.
- Como há menos que 256 caracteres em um teclado, geralmente são eles são representados representados por bytes. A linguagem C usa o tipo char para representar um byte ou caractere.
- Os primeiros computadores não trabalhavam com um padrão para representação de caracteres, por isso a troca de texto entre computadores era difícil.
- Em 1963, a American Standards Association publicou o American Standard Code for Information Interchange (ASCII), que atribui a cada caractere de um valor exclusivo usando um byte.

Alguns Caracteres e o Código ASCII Correspondente

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

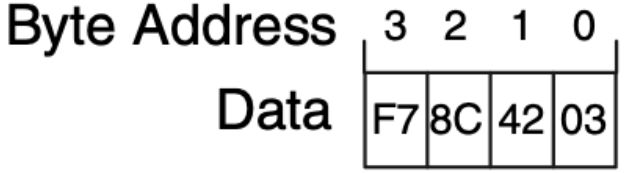
Leitura e escrita de bytes/caracteres

- Já vimos que o MIPS fornece instruções de leitura e armazenamento de bytes:
  - `lb`: lê um byte da memória, o qual é carregado no byte menos significativo do registrador de destino (estendido com o bit de sinal para 32 bits).
  - `sb`: armazenamento do byte menos significativo de um registrador em um byte da memória.
- Precisamos também conhecer a instrução `lbu`: **`lbu rt, imm(rs)`**
  - **Mnemônico:** `lbu` (*load byte unsigned*, leitura de byte sem sinal).
  - **Cálculo do Endereço:** `rs` (registrador base) + `imm` (offset de 16 bits estendido para 32 bits com sinal).
  - **Resultado:** `rt`, recebe em seu byte menos significativo ($rt_{0:7}$) o byte lido da memória ($rt_{[0:7]}$ = `Mem[rs+offset]`). O restante do registrador é preenchido com zeros.

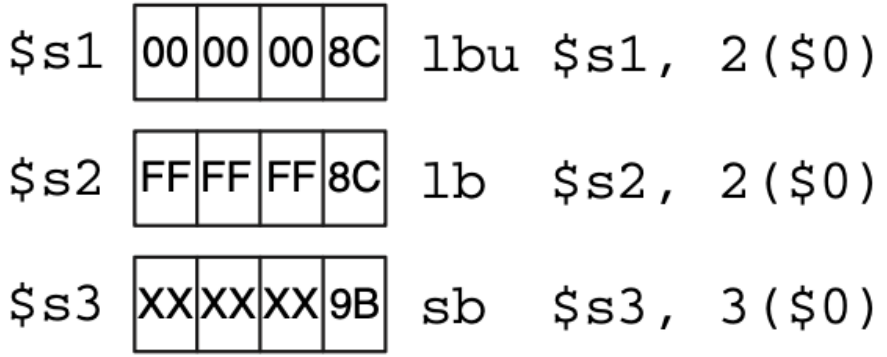
Leitura e escrita de bytes/caracteres

- Exemplo:

Memória Little-Endian



Registradores



Exemplo
Conversão de Para Maiúsculo

```
// Código C
  char array[10];
  int i;
  for (i = 0; i != 10; i = i + 1)
    array[i] = array[i] - 32;
```

```
# Código assembly MIPS
# $s0 = endereço base do array, $s1 = i
```

Exemplo
Conversão de Para Maiúsculo

```
// Código C
  char array[10];
  int i;
  for (i = 0; i != 10; i = i + 1)
    array[i] = array[i] - 32;
```

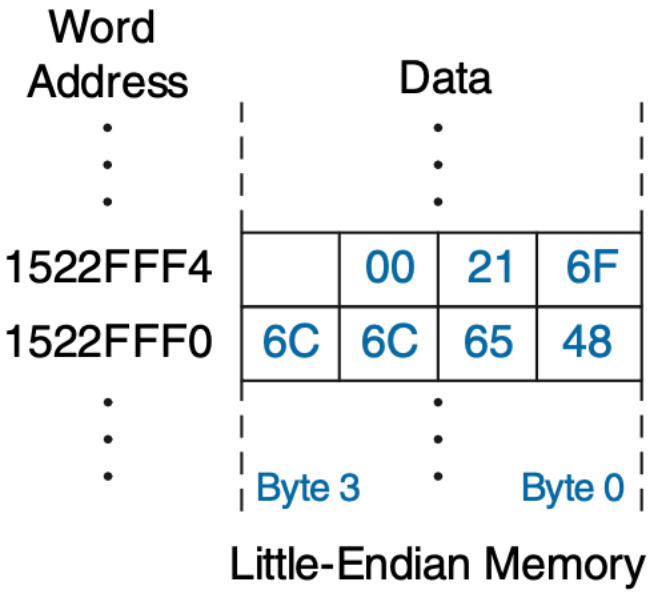
Supondo que os caracteres no array sejam minúsculos (a – z), vamos convertê-los para maiúsculos (A – Z).

```
# Código assembly MIPS
# $s0 = endereço base do array, $s1 = i
        addi $s1, $0, 0
        addi $t0, $0, 10
loop: beq $t0, $s1, done
        add $t1, $s1, $s0
        lbu $t2, 0($t1) # poderia ser lb?
        addi $t2, $t2, -32
        sb $t2, 0($t1)
        addi $s1, $s1, 1
        j loop
done:
```

Strings

- Uma string é sequência de caracteres. As strings têm comprimento variável e, portanto, as linguagens de programação devem fornecer uma maneira de determinar o final da string.
- Em C, o caractere Null (0x00) indica o fim de uma string. A string ao lado tem sete bytes e se estende do endereço 0x1522FFF0 a 0x1522FFF6. O primeiro caractere da string (H = 0x48) é armazenado no endereço de byte mais baixo (0x1522FFF0) devido ao exemplo considerar uma memória Little-Endian.
- Em assembly MIPS uma string pode ser criada com um label e a diretiva `.asciiz:`

```
string: .asciiz "linder"
```



Funções

Introdução às Funções

- Linguagens de alto nível usam funções (também chamadas de procedimentos) para reutilizar código e para tornar os programas mais modulares e legíveis.
  - Uma função é **definida** uma única vez, mas pode ser **chamada** várias vezes.

- As funções têm entradas, chamadas ***argumentos***, e uma saída, chamada ***valor de retorno***.

- As funções devem calcular o valor de retorno sem causar efeitos colaterais indesejados, isto é, depois de retornar a função não deve interferir na execução da função chamadora.

Funções: Caller e Callee

- **Caller (chamador):** a função que chama outra função, neste exemplo `main()`
- **Callee (chamado):** função chamada, neste exemplo, `sum()`

```
//Código C
void main()
{
   int y;
   y = sum(42, 7);
   ...
}

int sum(int a, int b)
{
   return (a + b);
}
```

Quando uma função chama outra função, a função chamadora, caller, e a função chamada, callee, devem concordar sobre onde os argumentos e o valor de retorno devem ser colocados.

MIPS: Convenções para Chamadas de Funções

- **Caller (chamador):**
  1. **Passa argumentos** para a função chamada (callee).
  2. **Faz a chamada**, isto é, desvia para executar as instruções da função chamada.
- **Callee (função chamada):**
  1. **Lê os argumentos passados.**
  2. **Realiza a tarefa e calcula o valor de retorno** (uma sequência de instruções).
  3. **Retorna o resultado** para o chamador (caller).
  4. **Desvia** para a instrução subsequente à sua chamada (volta ao caller).
  5. **Não deve sobrescrever** registradores ou memória usada pelo chamador (caller). Especificamente, os registradores salvos, `$s0`–`$s7`, `$ra`, e a pilha (uma porção da memória usada para variáveis locais).

MIPS: Convenções para Chamadas de Funções

1. Por convenção, o chamador (caller) coloca até quatro argumentos nos registradores `$a0`–`$a3`, antes de fazer a chamada da função. Quando uma função com mais de quatro argumentos é chamada, os argumentos adicionais são colocados na pilha, que discutiremos em breve.
2. O chamador usa a instrução Tipo-J *jump and link* para armazenar o endereço de retorno em `$ra` e desviar para a função chamada: **`jal label`**
   - **Mnemônico:** `jal` (*jump and link*).
   - **Resultado:** `$ra = PC+4`

     PC = JTA

   JTA = = {(PC + $4_{)[31:28]}$, addr << 2}, onde addr é o desvio do label em relação ao início do programa.

MIPS: Convenções para Chamadas de Funções

3. A função chamada (callee) coloca o retorno nos registradores `$v0`–`$v1`, antes de terminar, e retorna usando a instrução Tipo-R *jump register*: **`jr rs`**
   - **Mnemônico:** `jr` (*jump register*).
   - **Resultado:** PC = `$rs`

Exemplo: Chamando a Função mais Simples Possível

```
//Código C
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

```
#Código assembly MIPS
# $s0 = a, $s1 = b, $s2 = c
# main = 0x00400200, simple = 0x00401020
0x00400200 main: jal  simple
0x00400204       add  $s0, $s1, $s2
...

0x00401020 simple: jr $ra
```

jal: desvia para `simple`
e salva o endereço de retorno (`$ra` = PC + 4 = 0x00400204)

jr $ra: desvia para o o ponto de retorno no chamador (`$ra` = 0x00400204)

Funções com Argumento e Retorno

- A função `simple` não é muito útil porque não recebe nenhuma entrada da função chamadora (principal) e não retorna nenhuma saída.
- No exemplo, a seguir a função `diffofsums` é chamada com quatro argumentos e retorna um resultado.
- De acordo com a convenção MIPS, a função chamadora, `main`, deve colocar os argumentos nos registradores $a0–$a3. A função chamada, `diffofsums`, armazena o valor de retorno no registrador de retorno, $v0.
- Uma função que retorna um valor de 64 bits, usa ambos os registradores de retorno, $v0 e $v1. Quando uma função com mais de quatro argumentos é chamada, os argumentos de entrada adicionais são colocados na pilha.

Chamada de Função com Argumentos & Retorno

//C Code

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);
  ...
}

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

#Código assembly MIPS

```
# $s0 = y

main:
  ...
  addi $a0, $0, 2     # argumento 0 = 2
  addi $a1, $0, 3     # argumento 1 = 3
  addi $a2, $0, 4     # argumento 2 = 4
  addi $a3, $0, 5     # argumento 3 = 5
  jal  diffofsums     # chama a função
  add  $s0, $v0, $0   # $s0 = val. retornado
  ...

diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # (f + g) - (h + i)
  add $v0, $s0, $0    # retorno em $v0
  jr  $ra             # retorna para caller
```

Chamada de Função com Argumentos & Retorno

```
#Código assembly MIPS
# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

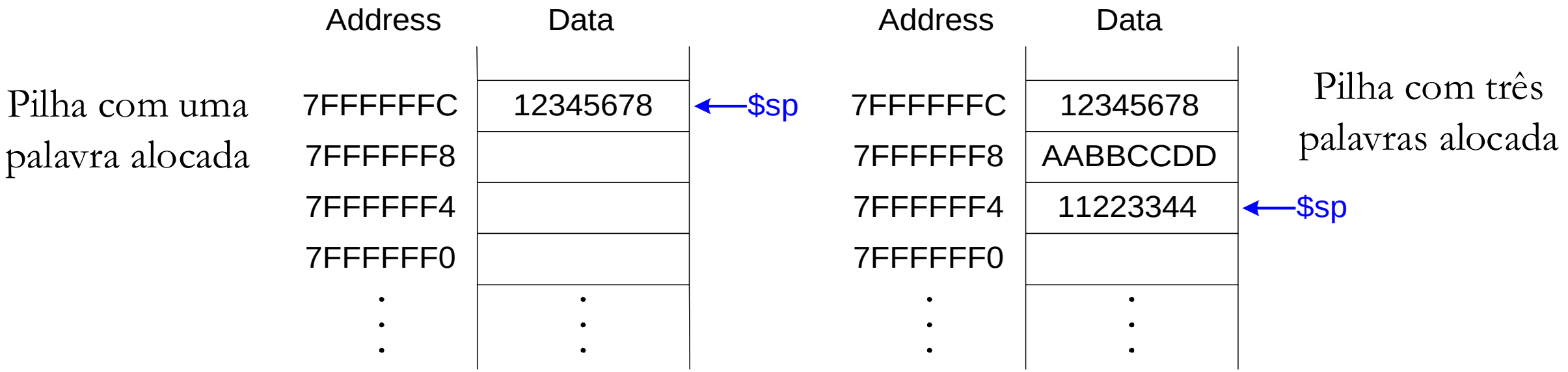
- **Problema**: `diffofsums` sobrescreveu os registradores `$t0`, `$t1` e `$s0`. Isso, potencialmente, poderia comprometer a execução da função chamadora (caller).
- **Solução**: `diffofsums` pode usar uma **pilha** (*stack*) para salvar o valor desses registradores antes de usá-los e, ao final, restaurá-los para os valores originais.

A Pilha (Stack)

- Espaço de memória usado para armazenar temporariamente variáveis locais (de uma função).
- Acessado de acordo com a dinâmica last-in-first-out (LIFO): Análogo ao empilhamento/desempilhamento de pratos.
- Estrutura dinâmica:
  - ***Expande (empilha):*** usa mais memória quando espaço é necessário.
  - ***Contrai (desempilha):*** usa menos memória quando espaço não é necessário.

A Pilha do MIPS

- Cresce dos maiores para os menores endereços de memória.
- O registrador `$sp` é usado como apontador de pilha. Aponta para o topo da pilha, isto é, o menor endereço acessível.



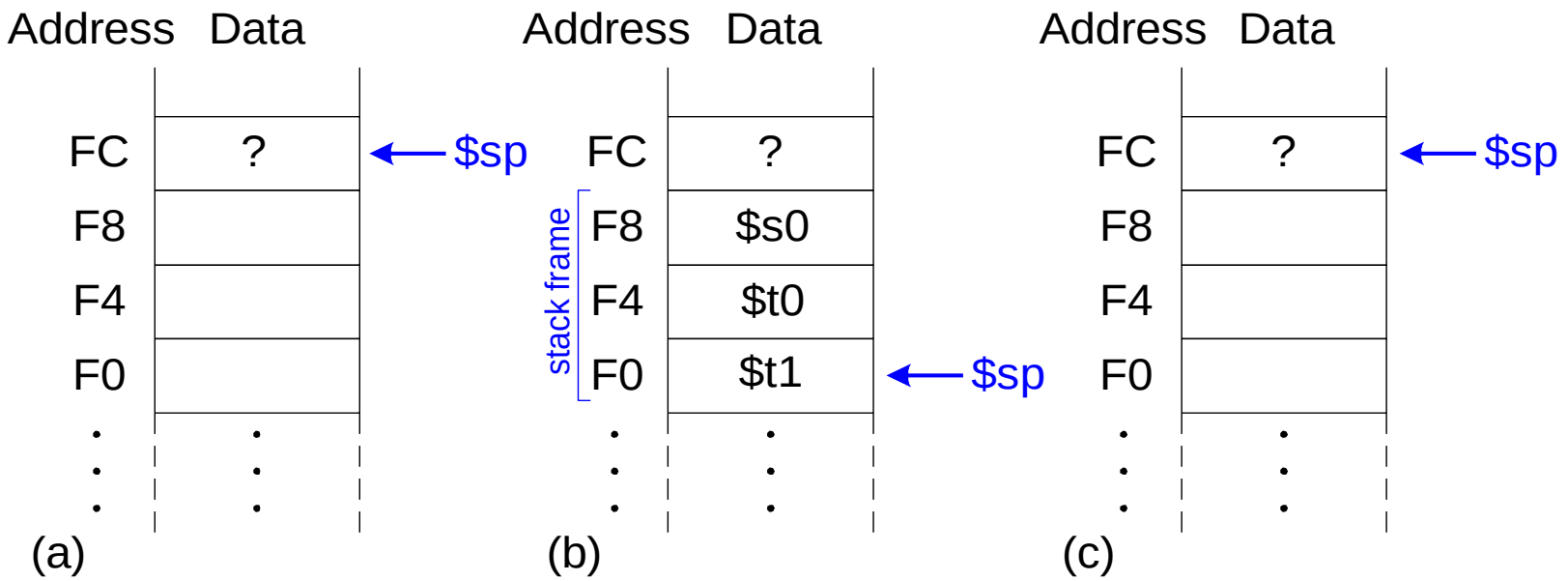
Como as Funções Usam a Pilha?

- A pilha é usada para armazenar variáveis locais e argumentos (quando o número excede 4), bem como salvar valores de registradores.
  1. A função deve alocar na pilha espaço suficiente para armazenar os valores de um ou mais registradores salvos, variáveis locais ou argumentos.
  2. A função deve armazenar os valores desejados na pilha.
  3. A função deve executar sua tarefa, utilizando os registradores que foram salvos.
  4. Antes de terminar, a função deve restaurar os valores originais dos registradores usando os valores salvos na pilha.
  5. A função deve desalocar espaço alocado na pilha, o que naturalmente destrói variáveis locais.

Diffofsums Versão 2: Usando a Pilha

```
diffofsums:
  addi $sp, $sp, -12   # aloca espaço na pilha para 3 registradores
  sw   $s0, 8($sp)     # salva $s0 na pilha
  sw   $t0, 4($sp)     # salva $t0 na pilha
  sw   $t1, 0($sp)     # salva $t1 na pilha
  add  $t0, $a0, $a1   # $t0 = f + g
  add  $t1, $a2, $a3   # $t1 = h + i
  sub  $s0, $t0, $t1   # resultado = (f + g) - (h + i)
  add  $v0, $s0, $0    # valor de retorno em $v0
  lw   $t1, 0($sp)     # restaura $t1 da pilha
  lw   $t0, 4($sp)     # restaura $t0 da pilha
  lw   $s0, 8($sp)     # restaura $s0 da pilha
  addi $sp, $sp, 12    # desaloca espaço na pilha
  jr   $ra             # retorna para o caller
```

A Pilha: Antes, Durante e Depois de `diffofsums`



- O espaço de pilha que uma função aloca para si mesma é chamado de *stack frame* (quadro de pilha). O *stack frame* de `diffofsums` tem três palavras. O princípio da modularidade diz que cada função deve acessar apenas seu próprio *stack frame*, e não os frames pertencentes a outras funções.

Registradores Preservados e Não Preservados por Funções

Preservados *Callee-Saved*	Não preservados *Caller-Saved*
`$s0-$s7`	**`$t0-$t9`**
`$ra`	**`$a0-$a3`**
`$sp`	**`$v0-$v1`**
pilha acima de `$sp`	**pilha abaixo `$sp`**

Para evitar esforço desnecessário, o MIPS dividiu os registradores em duas categorias: preservados (precisam ser restaurados, se alterados) e temporários (podem ser alterados).

Registradores Preservados e Não Preservados (Cont.)

- Quando uma função chama outra, o *callee* deve salvar e restaurar quaisquer registradores preservados que usar, mas pode alterar livremente qualquer um dos registradores não preservados.

- Portanto, se o *caller* (chamador) estiver mantendo dados ativos em um registrador não preservado, ele que precisará salvar esse registrador antes de fazer a chamada de função e depois restaurá-lo após o retorno.

- Por essas razões, os registradores preservados também são chamados de *callee-save*, e os registradores não preservados são chamados de *caller-save*.

diffofsums Versão 3

```
diffofsums:
   addi $sp, $sp, -4   # aloca espaço na pilha  sw  $s0, 0($sp)
   sw $s0, 0($sp)      # salva $s0, mas não $t0 e $t1
   add $t0, $a0, $a1   # $t0 = f + g
   add $t1, $a2, $a3   # $t1 = h + i
   sub $s0, $t0, $t1   # resultado = (f + g) - (h + i)
   add $v0, $s0, $0    # valor de retorno em $v0
   lw  $s0, 0($sp)     # restaura $s0 da pilha
   addi $sp, $sp, 4    # desaloca espaço na pilha
   jr  $ra             # retorna para o chamado
```

Funções Folha e Não Folha

- Uma função que não chama outras funções é chamada de **função folha**; um exemplo é a função `diffofsums`. Uma função que chama outras funções é chamada de **função não-folha**.

- As funções não-folha são um pouco mais complicadas porque podem precisar salvar registradores não preservados na pilha antes de chamar outra função e, em seguida, restaurar esses registradores.

- Especificamente, o chamador salva quaisquer registradores não preservados ($t0–$t9 e $a0–$a3) necessários após o retorno da função chamada. A função chamada (callee) salva qualquer um dos registradores preservados ($s0–$s7 e $ra) que pretende modificar.

Exemplo
Função Não Folha

```
proc1:
   addi $sp, $sp, -4    # aloca espaço na pilha
   sw   $ra, 0($sp)     # salva $ra na pilha
   jal  proc2           # chama proc2
   ...
   lw   $ra, 0($sp)     # restaura $ra
   addi $sp, $sp, 4     # desaloca espaço na pilha
   jr  $ra              # return to caller
```

Funções Recursivas

- Uma função **recursiva** é uma função não-folha que chama a si mesma. Essas funções trazem algumas complicações que discutiremos tomando como exemplo a função fatorial.

- Lembre-se de que fatorial(n) = n × (n –1) × (n –2) × ... × 2 × 1.

- Portanto, a função fatorial pode ser reescrita recursivamente como fatorial(n) = n × fatorial(n – 1) . O fatorial de 1 é simplesmente 1.

//Código C

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

Código assembly MIPS

Chamada de Função Recursiva: Fatorial

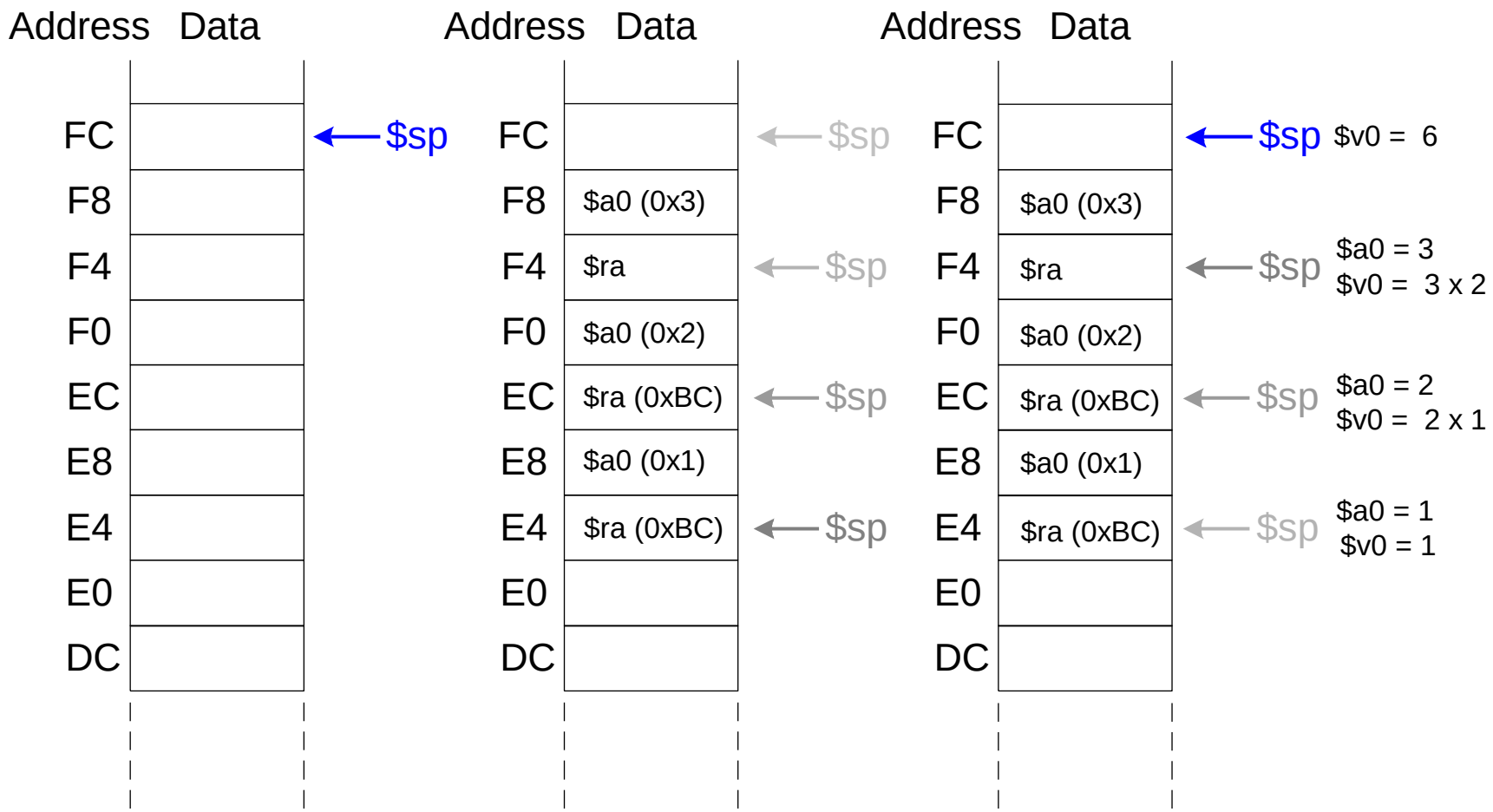
//Código C

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

Código assembly MIPS

```
0x90 factorial: addi $sp, $sp, -8  # aloca espaço
0x94            sw   $a0, 4($sp)   # armazena $a0
0x98            sw   $ra, 0($sp)   # armazena $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # no: vai para else
0xA8            addi $v0, $0, 1    # yes: return 1
0xAC            addi $sp, $sp, 8   # restore $sp
0xB0            jr   $ra           # return
0xB4       else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # recursive call
0xBC            lw   $ra, 0($sp)   # restore $ra
0xC0            lw   $a0, 4($sp)   # restore $a0
0xC4            addi $sp, $sp, 8   # restore $sp
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # return
```

Chamada de Função Recursiva: Fatorial

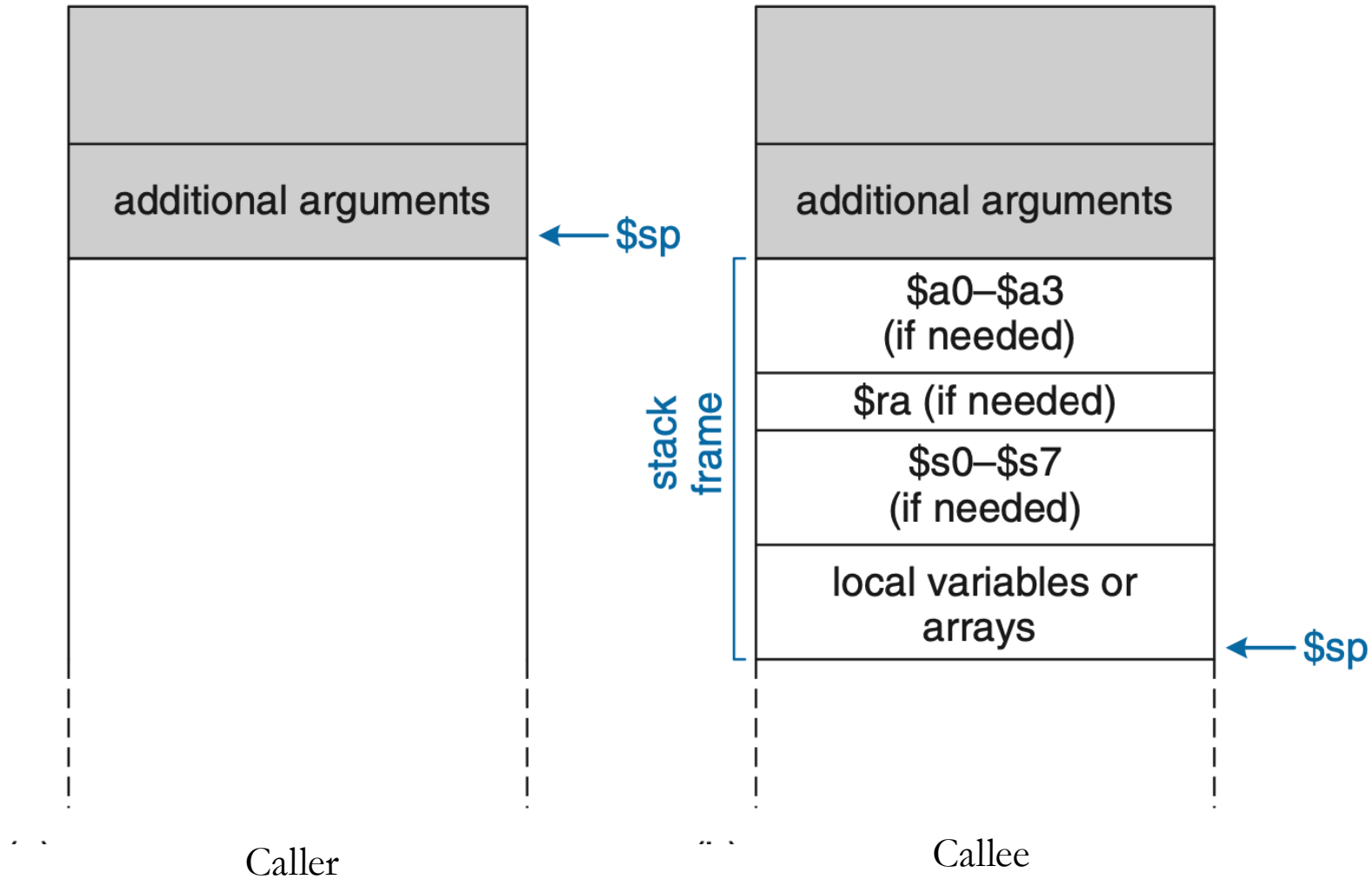


Pilha durante a chamada da função fatorial para n = 3

Argumentos Adicionais e Variáveis Locais

- As funções podem ter mais de quatro argumentos de entrada, bem como variáveis locais. A pilha é usada para armazenar esses valores temporários.
- Pela convenção MIPS, se uma função tiver mais de quatro argumentos, os primeiros quatro serão passados nos registradores $a0-$a3.
- Argumentos adicionais são passados na pilha, logo acima de $sp. O chamador deve expandir sua pilha para liberar espaço para argumentos adicionais.
- Uma função também pode declarar variáveis locais ou arrays locais. Variáveis locais são declaradas dentro de uma função e podem ser acessadas somente dentro dessa função. Variáveis locais são armazenadas em $s0–$s7, mas se houver muitas elas podem ser armazenadas no stack frame da função.

Argumentos Adicionais e Variáveis Locais



Sumário Sobre Chamada de Funções

- **Caller**
  - Põe argumentos em $a0-$a3 (talvez na pilha)
  - Salva na pilha qualquer registrador necessário ($ra, talvez $t0-t9)
  - Chama a função (`jal` callee)
  - Restaura registradores
  - Busca resultado em $v0 (talvez em $v1)

- **Callee**
  - Salva registradores que possam estar em uso pelo chamador ($s0-$s7)
  - Realiza a tarefa
  - Coloca o resultado em $v0 (talvez $v1)
  - Restaura registradores
  - Volta ao ponto após sua chamada `jr $ra`