Conjunto de Instruções

Sumário

- Introdução;
- CISC;
- RISC;
- MIPS;
- Representação de Instruções;
- SPIM;

Sumário

- Operações Lógicas;
- Instruções para Tomada de Decisões;
- Instruções de Desvio;
- Suporte a Procedimentos;
- Bibliografia.

Introdução

- A operação de uma CPU é determinada pelas instruções que ela executa, conhecidas como instruções de máquina ou instruções do computador;
- Cada instrução deve conter toda a informação necessária para que a CPU possa executá-la;
- Nesta(s) aula(s), abordaremos uma arquitetura simples de carregamento-armazenamento de 64 bits chamada MIPS.

CISC

- Antes, porém, vejamos algumas características de arquiteturas bastante comuns:

- CISC – Complex Instruction Set Computer:
  - Programas menores;
  - Simplicidade nos projetos de compiladores
  - Grande número de instruções

- CISC é caracterizada pelo grande números de conjunto de instruções complexas, objetivando o uso mínimo de memória;

CISC

- CISC incorpora complexos modos de endereçamento para os operandos e usa um pequeno número de registradores;
- Máquinas CISC foram, na maioria, construídas como microprogramadas e só agora começou a a implementar micro/macro instruções RISC;
- Exemplos de máquinas CISC:
  - PDP-11, Intel i86, Motorola 68K
  - CISC com micro/macro-instruções RISC: Pentium, AMD Athlon;

RISC

- RISC – Reduced Instruction Set Computer:
  - Simplicidade. Poucas instruções com poucos formatos;
  - Poucos modos de endereçamento;
  - Operações entre registradores;
  - Uma instrução por ciclo.

- RISC é caracterizada por usar um conjunto pequeno e simples de instruções;

- Máquinas RISC geralmente possuem número elevado de registradores;

RISC

- Máquinas RISC têm sido construídas para que suas instruções sejam diretamente executadas pelo hardware (não há microcódigo).

- Exemplos de Máquinas RISC:
  - MIPS,
  - PowerPC;
  - I860;
  - I960;
  - SPARC;
  - ALPAH
  - HP-PA

MIPS

- MIPS fornece um bom modelo arquitetônico para estudo, não apenas devido à popularidade desse tipo de processador, mas também porque é uma arquitetura fácil de entender;

- Nos 15 anos desde que surgiu o primeiro processador MIPS, houve muitas versões de MIPS. Adotaremos um subconjunto daquilo que se denomina agora MIPS64;

MIPS

- O MIPS tem 32 registradores de uso geral (GPRs – General-Purpose Registers) de 64 bits, denominados R1, ..., R31.

- Além disso, existe um conjunto de 32 registradores de ponto flutuante (FPRs – Floating-Point Registers), denominados F0, ..., F31.
  - Esses registradores podem conter 32 valores de precisão simples (de 32 bits) ou 32 valores de precisão dupla (64 bits)
  - Quando o FPR contém um número de precisão simples, a outra metade do FPR não é usada;.

Representação de Instruções

- Os registradores serão manipulados através de instruções. Internamente, cada instrução é representada como uma seqüência de bits.

- O MIPS admite quatro classes gerais de instruções:
  - Operações da ULA;
  - Carga e armazenamento;
  - Desvio e Saltos;
  - Operações de ponto flutuante;

Representação de Instruções

- Formato dos três tipos de instruções mais comuns:

R	op	rs	rt	rd	shamt	Funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
I	op	rs	rt	Imm		
	6 bits	5 bits	5 bits	16 bits		
J	op	Target				
	6 bits	26 bits				

- Instruções do Tipo – R:
  - Simboliza o uso de registradores pela instrução, normalmente, instruções Lógicas e Aritmética, vejamos os significados dos campos:

Representação de Instruções

- Instruções do Tipo – R:

6 bits	*5 bits*	*5 bits*	*5 bits*	*5 bits*	*6 bits*
op	*rs*	*rt*	*rd*	*shamt*	*funct*

  - **op** – código que identifica a instrução. Para R é sempre 0 (excepto rfe=0x10)
  - **rs, rt** – números do dois registos que contêm os operandos;
  - **rd** – número do registo que virá a conter o resultado;
  - **Número de shifts** – apenas para sll, srl e sra (0 nas outras instruções)
  - **shamt** – quantidade de bits a serem deslocados;
  - **funct** – identifica a função (operação)

Representação de Instruções

- Exemplo:

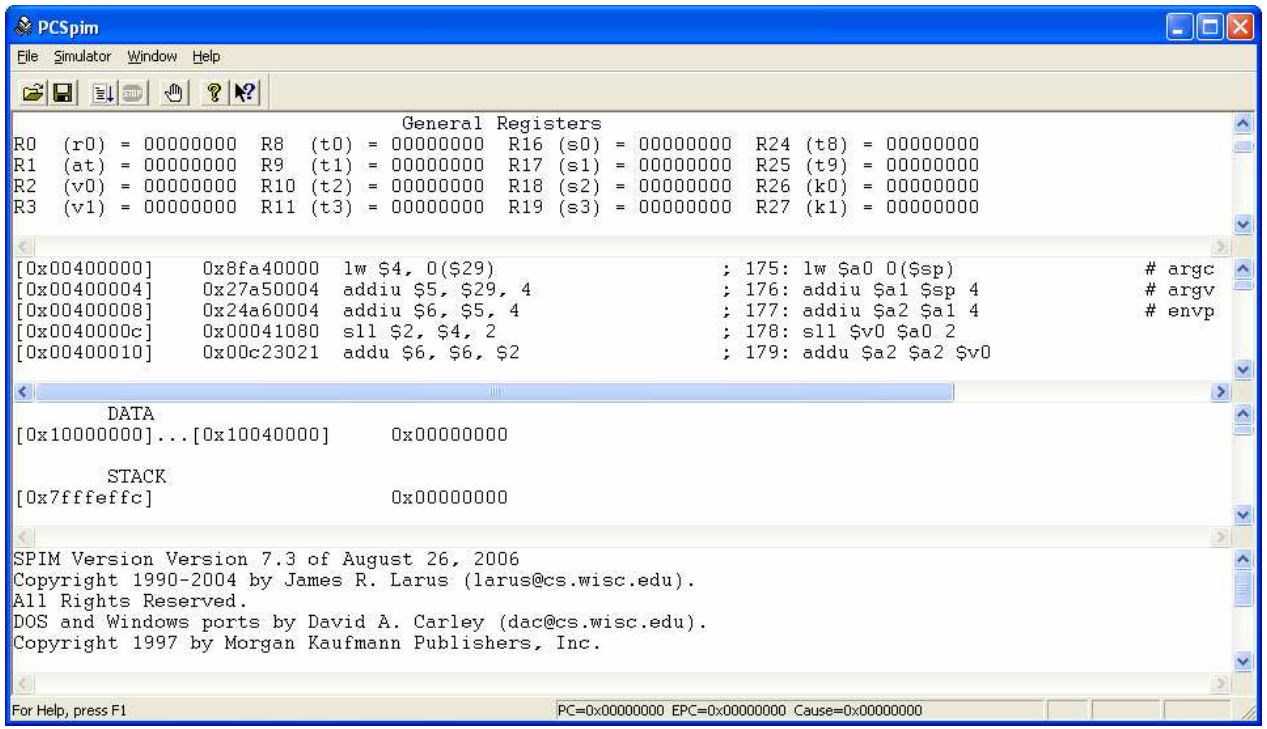
add $t0, $t2, $s0

op = 0x00
funct = 0x20
rs = $t2 = 10
rt = $s0 = 16
rd = $t0 = 8
shamt = 0

op	*rs*	*rt*	*rd*	*shamt*	*funct*
000000	01010	10000	01000	00000	100000

SPIM

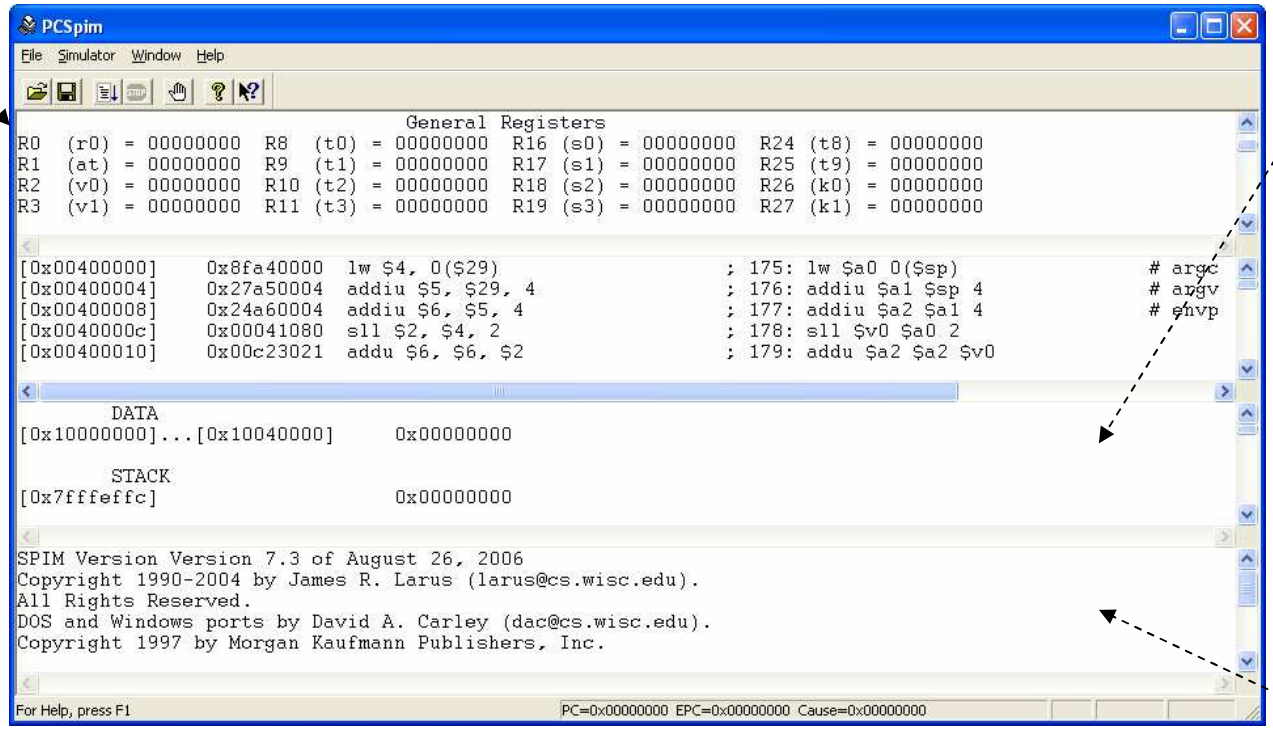
- Antes de prosseguir, vejamos um simulador (SPIM) para a programação em linguagem de máquina baseado no MIPS:



SPIM

Todos os Registradores da CPU MIPS

Display com os dados carregados na memória do programa e os dados sobre a pilha de programa

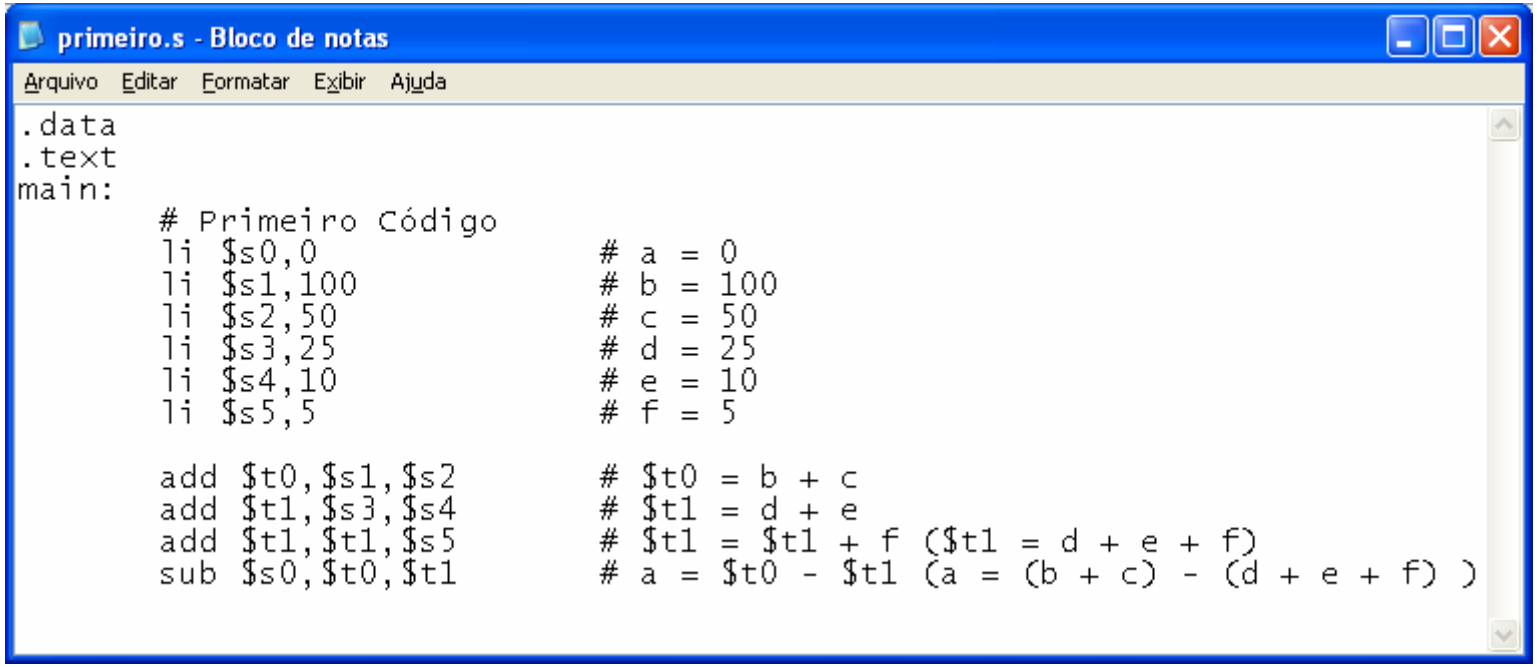


Display com as instruções tanto do programa quanto do sistema carregado automaticamente quando o SPIM roda.

Por fim, a área reservada para escrita de mensagens de erro, por exemplo

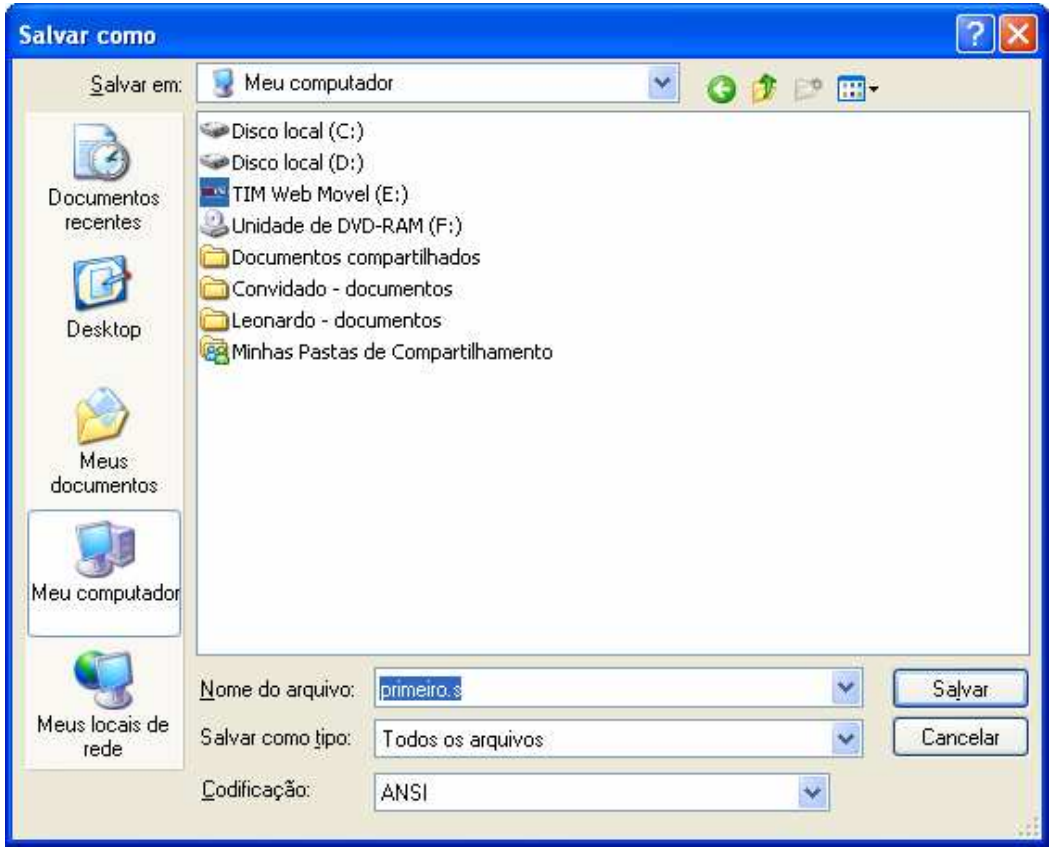
SPIM

- Carregando o primeiro código assembly no SPIM:



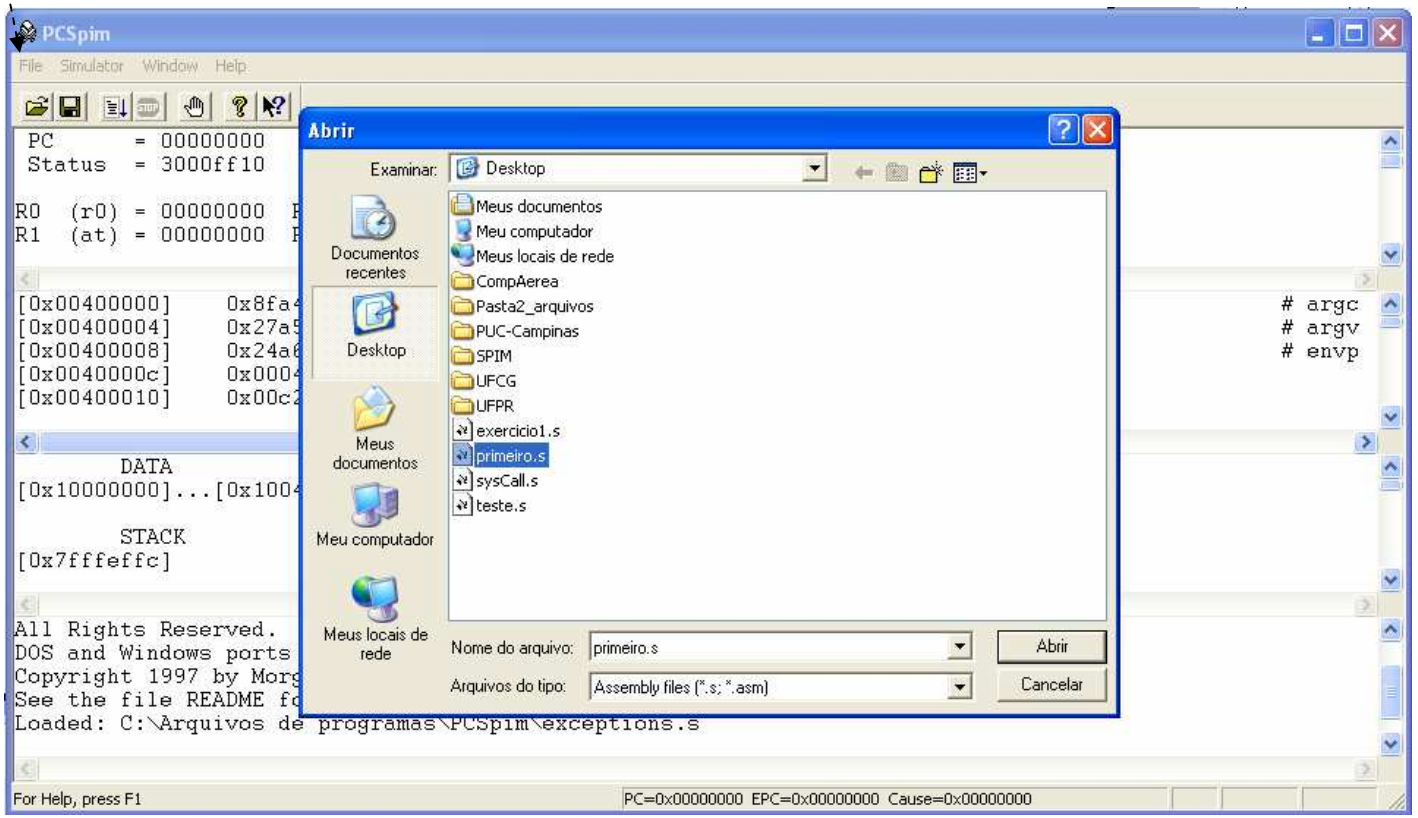
SPIM

- Extensão do arquivo: *.s



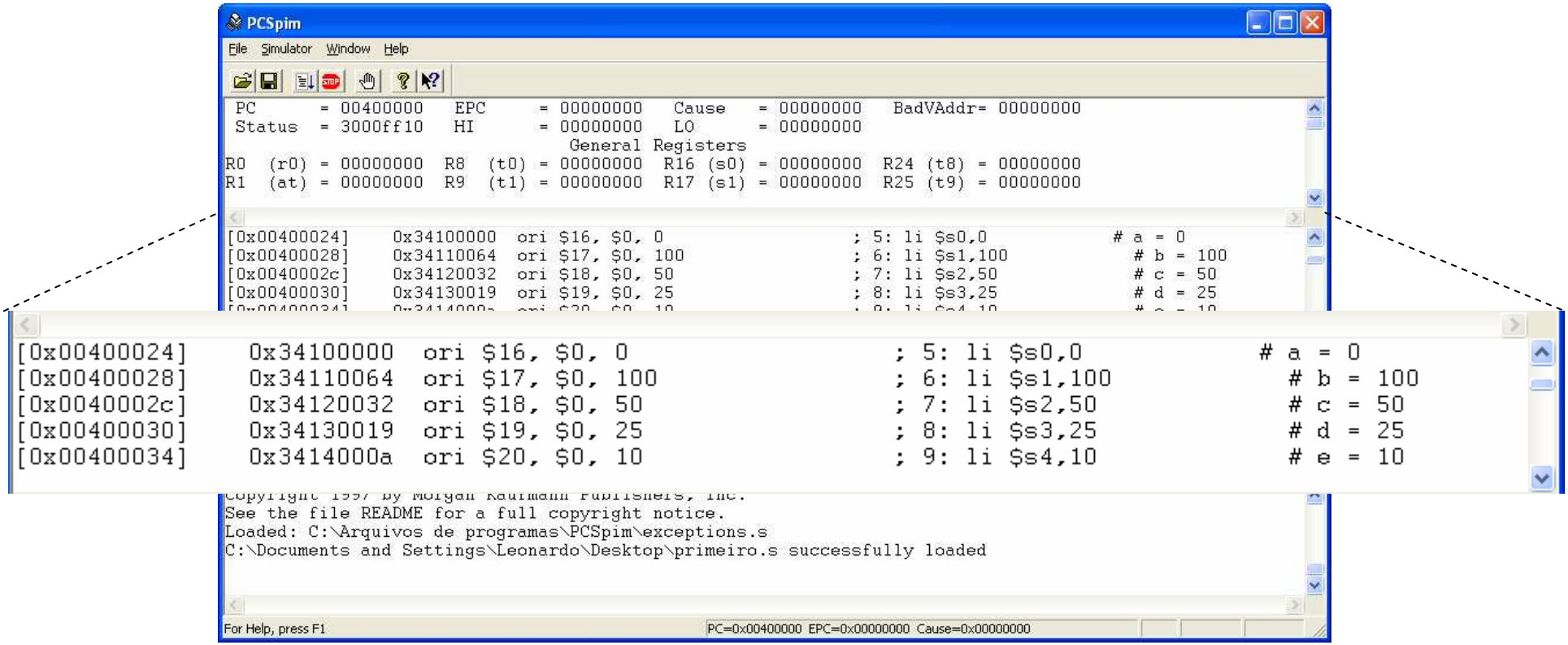
SPIM

- Abrindo o código no simulador:



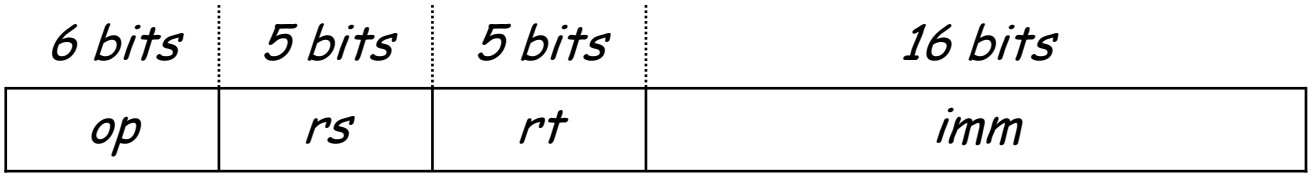
SPIM

- Rodando o primeiro programa:



Representação de Instruções

- Instruções do Tipo – I:
  - Simboliza instruções de transferência de dados, normalmente, instruções carga e armazenamento, vejamos os significados dos campos:



  - **op** – código que identifica a instrução. Para R é sempre 0 (excepto rfe=0x10)
  - **rs, rt** – números do dois registos que contêm os operandos;
  - **imm** – endereço de memória

Representação de Instruções

- Exemplo:

addi $t0, $t2, 256

op = 0x08

rs = $t2 = 10

rt = $t0 = 8

imm = 256

op	*rs*	*rt*	*endereço*
001000	01010	01000	0000 0001 0000 0000

Representação de Instruções

- Exemplo:

lw $t0, 0($t1)

op = 0x23 rt = $t0 = 8

rs = $t1 = 9 imm = 0

op	*rs*	*rt*	*endereço*
100011	01001	01000	0000 0000 0000 0000

Representação de Instruções

- Exemplo: Supondo que o endereço-base de um array A[300] esteja armazenado em $t1. Qual o código de montagem do MIPS para carregar no registrador temporário $t0 o conteúdo da última posição do array A?

3	1000 1111 ... 0010
2	1000 1001 ... 0100
1	1010 0101 ... 1000
0	1000 0000 ... 0000
Endereço	**Dados (32 bits)**

Representação de Instruções

- Exemplo:

Supondo que $t1 guarde o endereço base do array A[300], o deslocamento necessário é 300 x 4 = 1200

```
lw $t0, 1200($t1)
```

op = 0x23

rs = $t1 = 9

rt = $t0 = 8

imm = 1200

op	*rs*	*rt*	*endereço*
100011	01000	01000	0000 0100 1011 0000

Representação de Instruções

- Exemplo:

Armazena o conteúdo do registrador temporário $t0 no endereço de memória 1200 + endereço base;

```
sw $t0, 1200($t1)
```

op = 0x29

rs = $t1 = 9

rt = $t0 = 8

imm = 1200

op	*rs*	*rt*	*endereço*
101011	01001	01000	0000 0100 1011 0000

Exercício

- Vejamos um trecho de código em C:

```
g = h + A[i];
```

  - Suponha que A é um array de 100 elementos cujo endereço-base está no registrador $s3 e que o compilador associa as variáveis g, h e i aos registradores $s1, $s2 e $s4. Qual o código gerado para o MIPS, correspondente ao comando C acima?

Exercício

- Resolução:

```
add $t1, $s4, $s4    # $t1 recebe 2 * i

add $t1, $t1, $t1    # $t1 recebe 4 * i

add $t1, $t1, $s3    # $t1 recebe o endereço de A[i]

lw $t0, 0($t1)       # $t0 recebe A[i]

add $s1, $s2, $t0    # g recebe h + A[i]
```

Operações Lógicas

- Embora os primeiros computadores se concentrassem em words completas, logo ficou claro que era útil atuar sobre campos de bits dentro de uma word ou até mesmo sobre bits individuais;
  - Vejamos alguns operadores lógicos em C e Java e suas instruções MIPS correspondentes;

Operações lógicas	Operadores em C	Instruções MIPS
Shift à esquerda	<<	sll
Shift à direita	>>	srl
AND bit a bit	&	and, andi
OR bit a bit	|	or, ori
NOT bit a bit	~	nor

Operações Lógicas

- Os deslocamentos (*shifts*) movem todos os bits de uma word para a esquerda ou direita, preenchendo os bits que ficaram vazios com 0s;

- Caso o registrador $s0 tivesse o decimal 144, vejamos a aplicação de um srl (*shift right logical*);

srl $t2, $s0, 4

0000 0000 0000 0000 0000 0000 0000 1001 0000 = 144_{dec}
0000 0000 0000 0000 0000 0000 0000 0000 1001 = 9_{dec}

Operações Lógicas

- Vejamos o formato da instrução anterior do tipo R:

srl $t2, $s0, 4

op = 0x00
funct = 0x02

rt = $s0 = 16
rd = $t2 = 10
shamt = 4

op	*rs*	*rt*	*rd*	*shamt*	*funct*
000000	00000	10000	01010	00100	000010

Operações Lógicas

- Vejamos a aplicação de outra instrução lógica:
  - Supondo que o registrador $t2 tenha:

    0000 0000 0000 0000 0000 0000 1101 0000 0000

  - Supondo que o registrador $t1 tenha:

    0000 0000 0000 0000 0000 0011 1100 0000 0000

  - and $t0, $t1, $t2 resultará em:

    0000 0000 0000 0000 0000 0000 1100 0000 0000

Operações Lógicas

- Vejamos a aplicação de outra instrução lógica:
  - Supondo que o registrador $t2 tenha:

    0000 0000 0000 0000 0000 0000 1101 0000 0000

  - Supondo que o registrador $t1 tenha:

    0000 0000 0000 0000 0000 0011 1100 0000 0000

  - or $t0, $t1, $t2 resultará em:

    0000 0000 0000 0000 0000 0011 1101 0000 0000

Operações Lógicas

- Vejamos a aplicação de outra instrução lógica:
  - Supondo que o registrador $t1 tenha:

    0000 0000 0000 0000 0000 0011 1100 0000 0000

  - Supondo que o registrador $t3 tenha:

    0000 0000 0000 0000 0000 0000 0000 0000 0000

  - nor $t0, $t1, $t3 resultará em: NOT (A OR 0)

    1111 1111 1111 1111 1100 0011 1111 1111

Exercício

- Refaça o trecho de código em C usando deslocamentos quando a multiplicação for necessária:

```
g = h + A[i];
```

  - Suponha que A é um array de 100 elementos cujo endereço-base está no registrador $s3 e que o compilador associa as variáveis g, h e i aos registradores $s1, $s2 e $s4. Qual o código gerado para o MIPS, correspondente ao comando C acima?

Instruções para Tomada de Decisões

- As duas instruções mais comuns na linguagem de máquina do MIPS são:

```
beq registrador1, registrador2, L1

bne registrador1, registrador2, L1
```

  - beq (**b**ranch if **eq**ual) desvia se igual
  - bne (**b**ranch if **n**ot **e**qual) desvia se não igual

Instruções para Tomada de Decisões

- Vejamos o segmento de código a seguir, escrito em C:

```
if ( i == j)
    f = f – i;
else
{   f = g – h;
    f = f – i; }
```

  - Supondo que as cinco variáveis de f até j corresponda aos cinco registradores de $s0 a $s4, qual p código MIPS gerado pelo compilador?

Instruções para Tomada de Decisões

- A solução é simples:

```
beq $s3, $s4, L1;          # Desvia para L1 se i = j

add $s0, $s1, $s2          # f = g + h (executa se i ≠ j)

L1: sub $s0, $s0, $s3;     # f = f – i (sempre executado)
```

Instruções de Desvio

- Até o momento vimos o salto que depende de uma condição.
  - Outra importante instrução de desvio é o salto incondicional. Um simples exemplo é:

```
J Exit;
```

  - A instrução desvia para o Label imediatamente (basta ser lida);

- Usando as mesmas variáveis e registradores do exemplo anterior, obtenha o código MIPS gerado para o seguinte trecho de código em C:

```
if ( i == j)
   f = g + h;
else
   f = g – h;
```

Instruções de Desvio

- Vejamos a solução:

```
bne $s3, $s4, Else          # Desvia para Else se i = j

add $s0, $s1, $s2           # f = g + h (executa se i ≠ j)

j Exit                      # Desvia para Exit

Else: sub $s0, $s1, $s2     # f = g – h (sempre executado)

Exit:                       # Label Exit
```

Exercício

- Qual o código assembly MIPS correspondente ao trecho de código em C abaixo?

```
while (mat[i] == k)
    i += 1;
```

  - Supondo que i e k correspondam aos registradores $s3, $s5 e a base do array mat esteja em $s6.

Exercício

- Resposta:

```
Loop: sll $t1, $s3, 2        # Registrador temporário $t1 = 4 * 1
add $t1, $t1, $s6            # $t1 = endereço de mat[i]
lw $t0, 0($t1)               # Registro temporário $t0 = mat[i]
bne $t0, $s5, Exit           # Vá para Exit se mat[i] ≠ k
addi $s4, $s3, 1             # i = i + 1;
J Loop                       # Vá para o label Loop
Exit:                        # Label Exit
```

Arquitetura MIPS até o Momento

MIPS operands

Name	Example	Comments
32 registers	$s0, $s1, ..., $s7 $t0, $t1, ..., $t7, $zero	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0–$s7 map to 16–23 and $t0–$t7 map to 8–15. MIPS register $zero always equals 0.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; data in registers
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory
Conditional branch	branch on equal	beq $s1,$s2,L	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	bne $s1,$s2,L	if ($s1 != $s2) go to L	Not equal test and branch
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; used with beq, bne
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $t1	go to $t1	For *switch* statements

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
beq	I	4	17	18	25			beq $s1,$s2,100
bne	I	5	17	18	25			bne $s1,$s2,100
slt	R	0	18	19	17	0	42	slt $s1,$s2,$s3
j	J	2	2500					j 10000 (see section 3.8)
jr	R	0	9	0	0	0	8	jr $t1
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

FIGURE 3.9 MIPS architecture revealed through section 3.5. Highlighted portions show MIPS structures introduced in section 3.5. The J-format, used for jump instructions, is explained in section 3.8. Section 3.8 also explains the proper values in address fields of branch instructions.

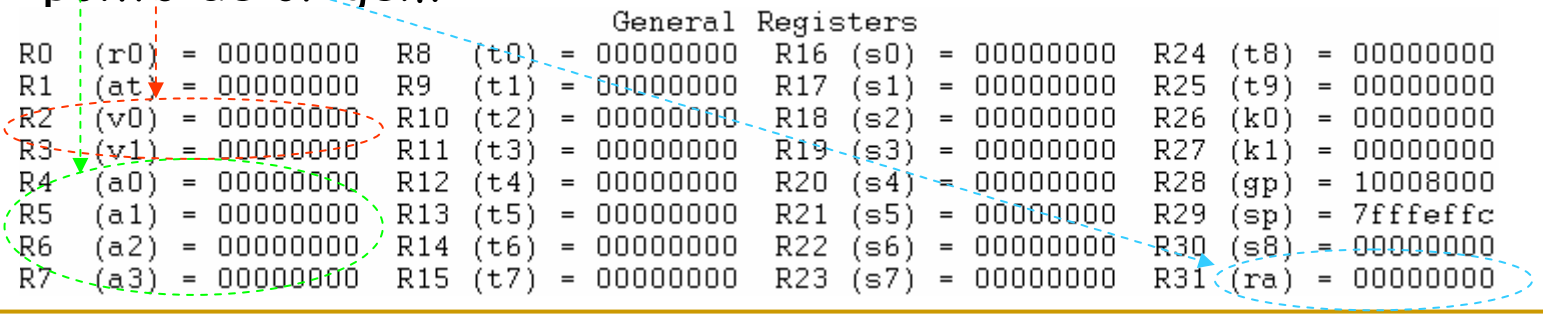
Suporte a Procedimentos

- Um procedimento ou função é uma ferramenta que os programadores utilizam para estruturar programas;

- De modo geral, na execução de um procedimento, o programa precisa seguir seis etapas:
  - **1)** Colocar parâmetros em um lugar onde o procedimento possa acessá-los;
  - **2)** Transferir o controle para o procedimento;
  - **3)** Adquirir os recursos de armazenamento necessários para o procedimento;
  - **4)** Realizar a tarefa desejada;
  - **5)** Colocar o valor do retorno em um lugar onde o programa que o chamou possa acessá-lo;
  - **6)** Retornar o controle para o ponto de origem, pois um procedimento pode ser chamado de vários pontos em um programa;

Suporte a Procedimentos

- O software do MIPS utiliza a seguinte convenção na alocação de seus 32 registradores para a chamada de procedimentos:
  - $a0 - $a3: quatro registradores de argumento, para passar parâmetros;
  - $v0 - $v1: dois registradores de valor, para valores de retorno;
  - $ra: um registrador de endereço de retorno, para retornar ao ponto de origem



Suporte a Procedimentos

- Além de alocar esses registradores, o assembly MIPS inclui uma instrução apenas para os procedimentos:

jal EndereçoProcedimento

  - A instrução jal (*jump-and-link*) desvia para um endereço e simultaneamente salva o endereço da instrução seguinte no registrador $ra.

- É óbvio que um registrador mantém o endereço da instrução atual, nesse caso, o registrador PC (*Program Counter*);

Suporte a Procedimentos

- Na prática, a instrução jal salva o endereço PC + 4 no registrador $ra para o link com a instrução seguinte, a fim de preparar o retorno do procedimento;

- O MIPS utiliza ainda uma instrução de *jump register* (jr), significando um desvio incondicional para o endereço especificado em um registrador:

jr $ra

Suporte a Procedimentos

- Recapitulando:
  - O programa que chama, ou *caller*, um procedimento coloca os valores de parâmetros em $a0 - $a3 e
  - Utiliza jal X para desviar para p procedimento X (*callee*);
  - Então, realiza os cálculos, coloca os resultados em $v0 - $v1 e
  - Retorna o controle para o programa usando jr $ra;
- Vejamos um exemplo, a seguir:

Suporte a Procedimentos

- Converter o procedimento escrito na linguagem de alto nível C para o código assembly MIPS:

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

  - Supondo que as variáveis de parâmetros g, h , i e j correspondem aos registradores $a0, $a1, $a2 e $a3, e f corresponde a $s0.

Suporte a Procedimentos

- Para compilar o programa anterior e preservar a os dados armazenados nos registradores é utilizado uma estrutura de dados do tipo pilha (*LIFO – Last In First Out*);
  - Push: colocar dados na pilha
  - Pop: remover dados da pilha
- A idéia é armazenar na memória os dados contidos nos registradores que serão utilizados pelo procedimento;
- Primeiramente, implementemos o programa fora de qualquer procedimento;

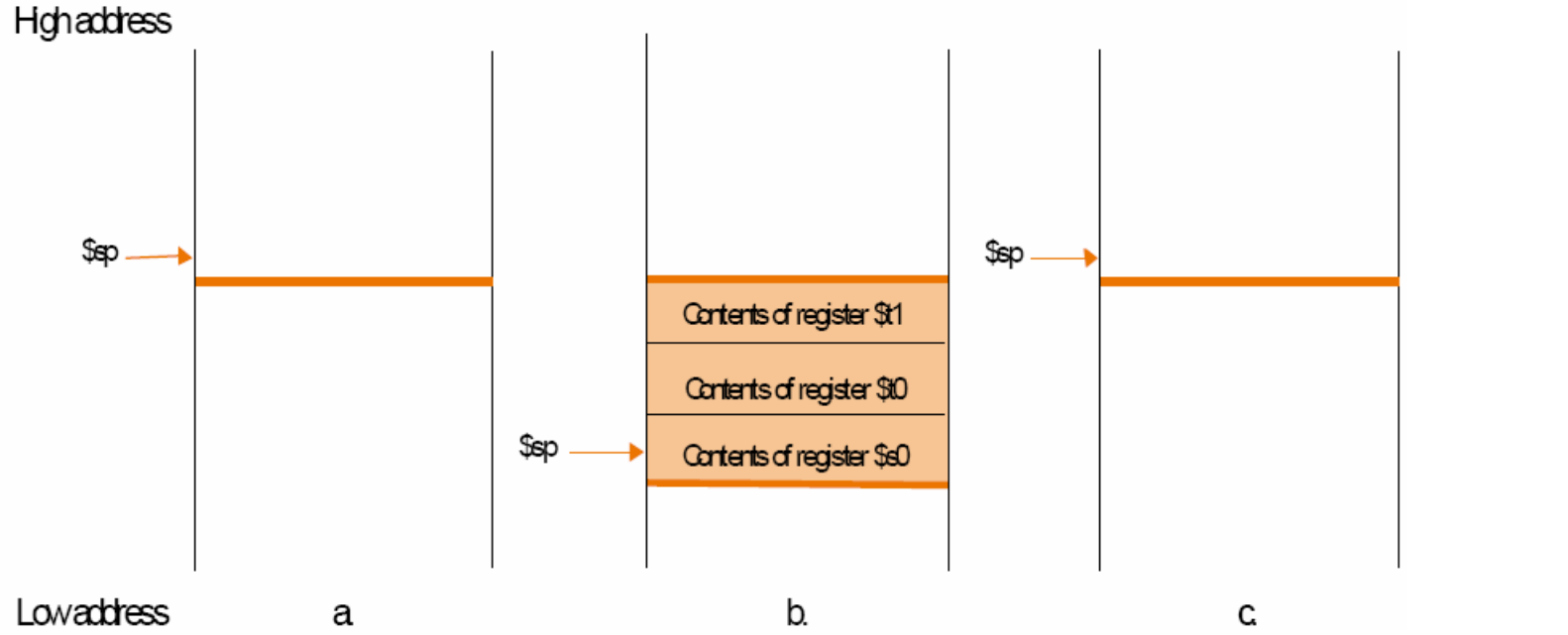
Suporte a Procedimentos

- Resposta:

```
exemplo:                # label exemplo

addi $sp, $sp, -12      # ajuste do sp para empilhar 3 palavras
sw $t1, 8($sp)          # salva $t1 na pilha
sw $t0, 4($sp)          # salva $t0 na pilha
sw $s0, 0($sp)          # salva $s0 na pilha

add $t0, $a0, $a1       # reg. $t0 contém g + h
add $t1, $a2, $a3       # reg. $t1 contém i + j
sub $s0, $t0, $t1       # f = $t0 - $t1, que é (g + h) – (i + j)
add $v0, $s0, $zero     # retorna f ($v0 = $s0 + 0)

lw $s0, 0($sp)          # restaura reg. $s0 para o caller
lw $t0, 4($sp)          # restaura reg. $t0 para o caller
lw $s1, 8($sp)          # restaura reg. $t1 para o caller
addi $sp, $sp, 12       # ajusta pilha para excluir 3 itens
jr $ra                  # desvia de volta à rotina que chamou
```

Suporte a Procedimentos

- Onde está o segredo?
  - No empilhamento dos registradores utilizados no procedimento, coordenado pelo registrador $sp (stack pointer – apontador de pilha);



Procedimentos Aninhados

- Os procedimentos que não chamam outros são denominados procedimentos folha;
  - "Felizmente" nem todos os procedimentos são assim;
- Procedimentos podem chamar outros procedimentos e a si mesmos (recursivos);
- Suponha um procedimento A com um argumento 3 armazenado em $a0 e que chame outro procedimento B com um argumento 7, também colocado em $a0:
  - **CONFLITOS!!!!!** (Nos registradores: $a0 e $ra);

Procedimentos Aninhados

- Uma solução é empilhar todos os outros registradores que precisam ser preservados;
- O caller empilha quaisquer registradores de argumento ($a0 - $a3) ou registradores temporários ($t0 - $t9) que sejam necessários após a chamada.
- Dessa forma, o *stack pointer* ($sp) é ajustado para levar em consideração a quantidade de registradores na pilha.

Procedimentos Aninhados

- Vejamos a aplicação da solução no procedimento recursivo escrito na linguagem C abaixo:

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

  - O parâmetro n corresponde ao registrador de argumentos $a0. O programa compilado começa com o rótulo do procedimento e depois salva dois registradores na pilha e o endereço de retorno e $a0:

Procedimentos Aninhados

- Analisemos a resposta:

```
fatorial:                  # label fatorial
sub $sp,$sp,8              # ajuste da pilha
sw $ra,4($sp)              # salva o endereço de retorno
sw $a0,0(sp)               # salva o argumento n
slt $t0,$a0,1              # teste para n<1
beq $t0,$zero,L1           # se n>=1, vá para L1
add $v0,$zero,1            # retorna 1 se n < 1
add $sp,$sp,8              # pop 2 itens da pilha
jr $ra
```

Procedimentos Aninhados

- Analisemos a resposta:

```
L1:
sub $a0,$a0,1          #n>=1, n-1
jal fatorial           #chamada com n-1
lw $a0,0($sp)          #retorno do jal; restaura n
lw $ra,4($sp)
add $sp,$sp,8
mult $v0,$a0,$v0       #retorna n*fatorial(n-1)
jr $ra
```

Bibliografia

- Stallings, W. *Arquitetura e Organização de Computadores*, Pearson Hall, 5 ed. SP: 2002.