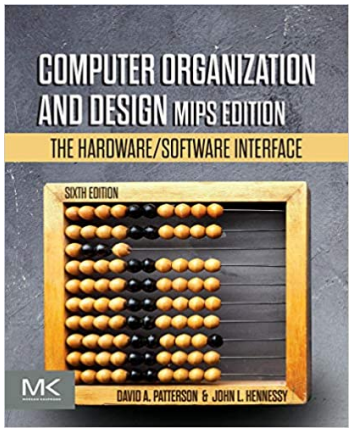
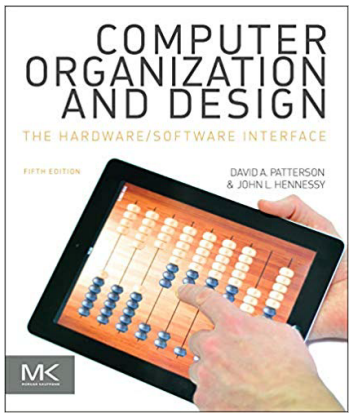
# Aula 04 – Linguagem Assembly

Prof. Dr. Clodoaldo A. de Moraes Lima

Material baseado no livro "Patterson, David A., Hennessy, J. L. - Computer Organization And Design: The Hardware/Software Interface"





# Conjunto de Instruções

## Introdução

- Um computador precisa receber ordens para que possa executar as ações desejadas por seu operador.
- Operador precisa conseguir "falar"em uma lingua que o computador entenda

## Instruções

"Palavras"da linguagem compreendida pelo computador

## Conjunto de Instruções

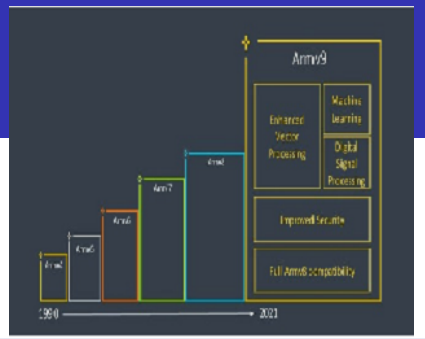
Vocabulário da linguagem compreendida pelo computador

# Conjunto de Instruções

## Introdução

- Semelhante a uma linguagem de programação restrita (poucos comandos).
- Projetistas de processadores objetivam:
  - Criar uma linguagem de hardware que seja de fácil implementação e tradução para compiladores de outras linguagens.
- Conjunto de instruções deve ser escolhido de forma a simplificar o projeto dos circuitos internos do processador.
- Conhecer o conjunto de instruções permite compreender o funcionamento do processador e as decisões de melhoria do desempenho.
- Consideraremos o conjunto de instruções dos processadores MIPS.

# MIPS




- É um exemplo elegante do conjunto de instruções projetado desde 1980
- ARMv7 é similar para o MIPS. Mais que 9 bilhões de chips com processadores ARM foram fabricados em 2011, tornando-o conjunto de instruções mais popular no mundo
- O segundo exemplo é o Intel x86, que alimenta tando o PC e a nuvem da Era Pós PC
- O terceiro exemplo é o ARMv8, que estende o tamanho do endereço do ARMv7 de 32 bits para 64 bits. Este conjunto de instruções (lançado em 2013) é mais perto do MIPS que é para ARMv7.

## Operações do Hardware do Computador

- O conjunto básico de instruções de um processador são as operações aritméticas
- O seguinte código: add a, b, c
  - Executa a soma das variáveis b e c, e coloca o valor resultante dentro da variável a.
- Qual o equivalente em MIPS para o seguinte código em linguagem de alto nível: a = b+c+d+e?

```
add a, b, c
add a, a, d
add a, a, e
```

# Conjunto de Instruções

## Operações do Hardware do Computador

- Na arquitetura MIPS as instruções aritméticas executam sempre sobre 3 operandos.

## Regra: Simplicidade favorece regularidade

Regularidade implica em um projeto de processador mais simples.

## Operações do Hardware do Computador

- Operandos da linguagem MIPS

**MIPS operands**

| Name | Example | Comments |
|---|---|---|
| 32 registers | `$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at` | Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants. |
| $2^{30}$ memory words | Memory[0], Memory[4], . . . , Memory[4294967292] | Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers. |

## Operações do Hardware do Computador

- Linguagem Assembly MIPS

**MIPS assembly language**

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | `add $s1,$s2,$s3` | $s1 = $s2 + $s3 | Three register operands |
| | subtract | `sub $s1,$s2,$s3` | $s1 = $s2 – $s3 | Three register operands |
| | add immediate | `addi $s1,$s2,20` | $s1 = $s2 + 20 | Used to add constants |
| Data transfer | load word | `lw $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Word from memory to register |
| | store word | `sw $s1,20($s2)` | Memory[$s2 + 20] = $s1 | Word from register to memory |
| | load half | `lh $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Halfword memory to register |
| | load half unsigned | `lhu $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Halfword memory to register |
| | store half | `sh $s1,20($s2)` | Memory[$s2 + 20] = $s1 | Halfword register to memory |
| | load byte | `lb $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Byte from memory to register |
| | load byte unsigned | `lbu $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Byte from memory to register |
| | store byte | `sb $s1,20($s2)` | Memory[$s2 + 20] = $s1 | Byte from register to memory |
| | load linked word | `ll $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Load word as 1st half of atomic swap |
| | store condition. word | `sc $s1,20($s2)` | Memory[$s2+20]=$s1;$s1=0 or 1 | Store word as 2nd half of atomic swap |
| | load upper immed. | `lui $s1,20` | $s1 = 20 * $2^{16}$ | Loads constant in upper 16 bits |

## Operações do Hardware do Computador

- Linguagem Assembly MIPS

| | | | | |
|---|---|---|---|---|
| Logical | and | `and $s1,$s2,$s3` | $s1 = $s2 & $s3 | Three reg. operands; bit-by-bit AND |
| | or | `or $s1,$s2,$s3` | $s1 = $s2 \| $s3 | Three reg. operands; bit-by-bit OR |
| | nor | `nor $s1,$s2,$s3` | $s1 = ~ ($s2 \| $s3) | Three reg. operands; bit-by-bit NOR |
| | and immediate | `andi $s1,$s2,20` | $s1 = $s2 & 20 | Bit-by-bit AND reg with constant |
| | or immediate | `ori $s1,$s2,20` | $s1 = $s2 \| 20 | Bit-by-bit OR reg with constant |
| | shift left logical | `sll $s1,$s2,10` | $s1 = $s2 << 10 | Shift left by constant |
| | shift right logical | `srl $s1,$s2,10` | $s1 = $s2 >> 10 | Shift right by constant |
| Conditional branch | branch on equal | `beq $s1,$s2,25` | if ($s1 == $s2) go to PC + 4 + 100 | Equal test; PC-relative branch |
| | branch on not equal | `bne $s1,$s2,25` | if ($s1!= $s2) go to PC + 4 + 100 | Not equal test; PC-relative |
| | set on less than | `slt $s1,$s2,$s3` | if ($s2 < $s3) $s1 = 1; else $s1 = 0 | Compare less than; for `beq`, `bne` |
| | set on less than unsigned | `sltu $s1,$s2,$s3` | if ($s2 < $s3) $s1 = 1; else $s1 = 0 | Compare less than unsigned |
| | set less than immediate | `slti $s1,$s2,20` | if ($s2 < 20) $s1 = 1; else $s1 = 0 | Compare less than constant |
| | set less than immediate unsigned | `sltiu $s1,$s2,20` | if ($s2 < 20) $s1 = 1; else $s1 = 0 | Compare less than constant unsigned |

## Operações do Hardware do Computador

- Linguagem Assembly MIPS

| | | | | |
|---|---|---|---|---|
| Unconditional jump | jump | `j 2500` | go to 10000 | Jump to target address |
| | jump register | `jr $ra` | go to `$ra` | For switch, procedure return |
| | jump and link | `jal 2500` | `$ra` = PC + 4; go to 10000 | For procedure call |

## Operações do Hadware do Computador

- Exemplo 1:
  - Transformar o seguinte segmento de código em C para a linguagem assembly MIPS

$$a = b + c$$
$$d = a - c$$

- Resposta

```
add a, b, c
sub d, a, c
```

## Operações do Hadware do Computador

- Exemplo 2
  - Transformar o seguinte segmento de código em C para a linguagem assemby MIPS: $f = (g + h) - (i + j)$
- Resposta
  - O compilador separa a atribuição em diversas instruções
  - Passo 1 - Cálculo de g+h;
  - Passo 2 - Cálculo de i+j;
  - Cálculo da subtração;

```
add t0, g, h # variável temporária g+h
add t1, i, j # variável temporária i+j
sub f, t0, t1 # f pega t0 - t1, que é (g+h)-(i+j)
```

# Conjunto de Instruções

## Operandos do Hardware

- Diferentemente de programas em alto-nível (C/C++, Java, Python, etc) operandos de instruções aritméticas são restritos.
- Operandos precisam vir de locais específicos chamados "registradores".
- "registradores" são componentes em hardware também visíveis aos desenvolvedores.
- O tamanho de um registrador na linguagem MIPS é 32bits.
- Grupos de 32bits são conhecidos como palavra (word).

# Conjunto de Instruções

## Operandos do Hardware

- Diferentemente de programas em alto-nível (C/C++, Java, Python, etc) operandos de instruções aritméticas são em quantidades limitadas.
- Operandos precisam vir de locais específicos chamados "registradores" - pequenos espaços de memória dentro do processador
- Banco de registradores: Regiões contínuas de memória dentro do processador.
- "registradores" são componentes em hardware também visíveis aos desenvolvedores.

# Conjunto de Instruções

## Operandos do Hardware

- O tamanho de um registrador na linguagem MIPS é 32bits.
- Grupos de 32bits são conhecidos como palavra (word).
- Na arquitetura MIPS o banco de registradores armazena no máximo 32 registradores.
  - No momento da compilação apenas 32 variáveis podem estar em uso simultaneamente.
  - "Smaller is faster": Um número maior de registrador pode alterar o tempo de ciclo de clock, pois os sinais precisariam percorrer um caminho maior.

## Operandos do Hardware

- A nomenclatura dos registradores do MIPS faz uso do caractere '$' no início de cada um: $s1, $s2, $t0, $t1, etc.
- Exemplo 3
  - Conhecendo as regras de nomenclatura, alterar o código do exemplo anterior para a linguagem MIPS: $f = (g + h) - (i + j)$
- Resposta:

```
add $t0,$s1,$s2 # registrador t0 contém g+h
add $t1,$s3,$s4 # registrador t1 contém i+j
sub $t0,$s1,$s2 # f pegar $t0 - $t1, que é (g+h)-(i+j)
```

## Operandos do Hardware

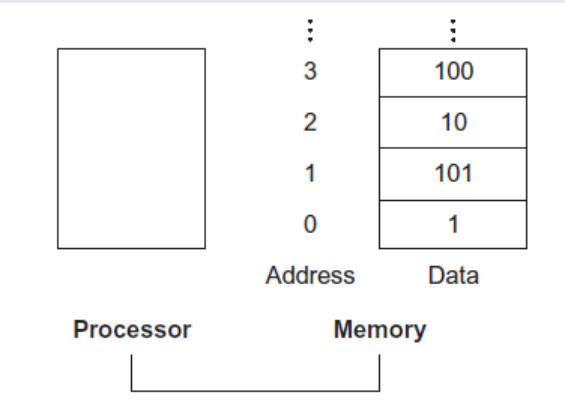
- Operandos em memória
  - Variáveis simples são armazenadas em registradores. Mas e quanto aos arrays e estruturas complexas?
  - Um array simples pode conter mais elementos que o número de registradores disponíveis. Como armazená-los?
  - Memória do computador pode conter bilhões de elementos, assim...
  - Estruturas de dados complexas são mantidas em memória.

## Operandos do Hardware

- Operandos em memória
  - Se as operações ocorrem somente nos registradores, como então fazer uso da memória?
  - Linguagem MIPS provê recursos para que seja possível transferir dados entre a memória e os registradores.
  - Instruções de transferência de dados (data transfer instructions).
  - Para acessar uma palavra na memória, o endereço dessa palavra precisa ser fornecido.

## Operandos do Hardware

- Operandos em memória
  - Nesse caso, a memória nada mais é que um simples array unidimensional com o endereço atuando como índice.



## Operandos do Hardware

- Operandos em memória
  - Carregando dados da memória para registradores (load word - lw)
  - Carregando dados dos registradores para memória (save word - sw)
- Exemplo 4:
  - A é um array de 100 palavras e o compilador associou às variáveis g e h os registradores $s1 e $s2. Ainda, o endereço inicial do array (base address) está localizado no registrador $s3. Compile o seguinte trecho de código: $g = h + A[8]$
- Resposta

```
lw $t0,8($S3) # registrador temporario, pega A[8]
add $s1, $S2, $t0 # g = h + A[8]
```

## Operandos do Hardware

- Operandos em memória
  - No MIPS, devido ao tamanho dos registradores (32 bits), o endereço de memória também usa 32 bits (4 bytes). Por isso, cada acesso a memória tem o ajuste de 4.
  - Endereçamento de palavras sempre é dado em múltiplos de 4.
  - Em relação ao modo de endereçamento, computadores se dividem em dois grupos: big endian e little endian.
  - MIPS é big endian.

## Operandos do Hardware

- Operandos em memória
  - Para os dados: 0x90AB12CD → 0x90, 0xAB, 0x12, 0xCD
  - big endian: Byte mais significativo no menor endereço de memória.

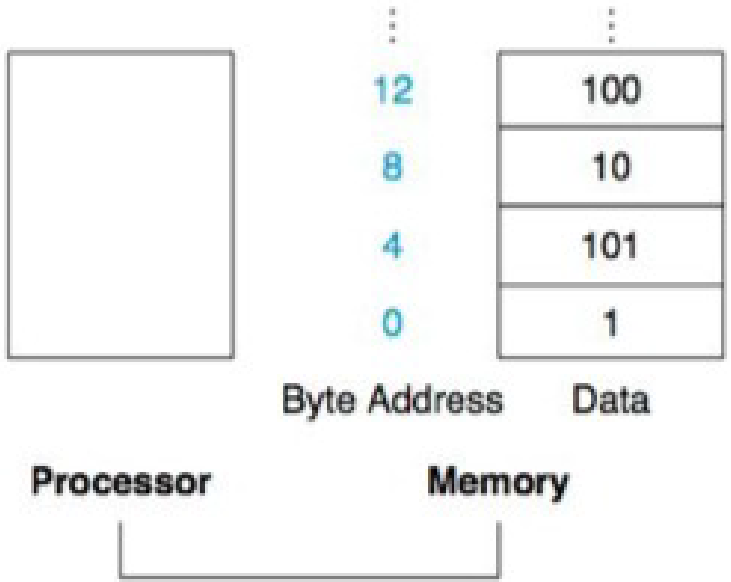
| Endereço | Valor |
|---|---|
| 1003 | CD |
| 1002 | 12 |
| 1001 | AB |
| 1000 | 90 |

  - little endian: Byte menos significativo no menor endereço de memória.

| Endereço | Valor |
|---|---|
| 1003 | 90 |
| 1002 | AB |
| 1001 | 12 |
| 1000 | CD |

## Operandos do Hardware

- Operandos em memória
  - Considerando o endereçamento de 32 bits, o endereço utilizado no Exemplo 4 necessita ser reescrito conforme o layout a seguir.
  - Offset necessita ser multiplicado pelo tamanho de cada endereço. $8x4 = 32$.



## Operandos do Hardware

- Operandos em memória
- Exemplo 5:
  - Assumindo que a variável h está associada com o registrador $s2 e o endereço base do array A está em $s3, qual o código em assembly MIPS relacionado ao seguinte código C? $A[12] = h + A[8]$;
  - Resposta

```
lw $t0, 32($S3) # registrador temporário $t0 pega A[8]
add $t0, $S2, $t0 # registrador temporário $t0 pega h+ A[8]
sw $t0, 48$s3 # armazena h+ A[8] de volta em A[12]
```

## Operandos do Hardware

- Operandos em memória
  - Registradores são mais rápidos que memória.
  - Acesso aos registradores requer um menor gasto energético.
  - Registradores precisam ser utilizados de maneira eficiente.
  - Programas com maior quantidade de variáveis que registradores: Compilador tenta manter nos registradores as variáveis mais frequentemente utilizadas.

## Operandos do Hardware

- Operandos em memória
  - Algumas vezes um programa irá fazer uso de valores constantes em determinadas operações (ex: incremento de uma var. em um laço).
  - Até agora, somar 4 a um registrador $s3:
    lw $t0, AddrConstant4($s1) # $t0 = constant 4
    add $s3, $s3, $t0 # $s3 = $s3 + $t0 ($t0 == 4)
  - $s1 + AddrConstant4 → é o endereço de memória da constante 4
  - Problema?
    - Carregar dados da memória tem um alto custo.

## Operandos do Hardware

- Operandos em memória
- Solução
  - Versões das instruções aritméticas onde um dos operandos é uma constante.
  - Evita-se uso das funções de carga de dados da memória (lw).
  - Regra: Operações mais comuns devem ser as mais rápidas.
  - Para adicionar 4 ao registrador $s3:
    addi $s3, $s3, 4 # $s3 = $s3 +4

## Operandos do Hardware

- Exercicios
  - Para cada trecho de código, converta para assembly MIPS usando a menor quantidade possível de registradores:
  - $f = g + (h + 5)$
  - $f = f + f + i$
  - $f = f + g + h + i + j + 2$
  - $f = g + h + B[4]$
  - $f = g - A[B[4]]$

# Instruções Para Salto Condicional/Incondicional

## O que difere um computador de uma calculadora?

- Habilidade em tomar diferentes decisões.
- Baseando-se nos dados inseridos e nos valores obtidos durante a computação, diferentes instruções podem ser executadas.
- Tomada de decisão é comumente representada em linguagens de programação com a palavra reservada if (combinado ou não com o goto).

# Instruções Para Salto Condicional/Incondicional

## O que difere um computador de uma calculadora?

- Em alguns casos se faz necessário simplesmente alterar o fluxo do programa:

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>

using namespace std;

int main() {
   int i = 1;

   if (i == 1) {
       printf("i == 1\n");
       goto finish;
   }
   printf("i != 1\n");
   finish:
   printf("FIM\n");
   return 0;
}
```

# Instruções Para Salto Condicional/Incondicional

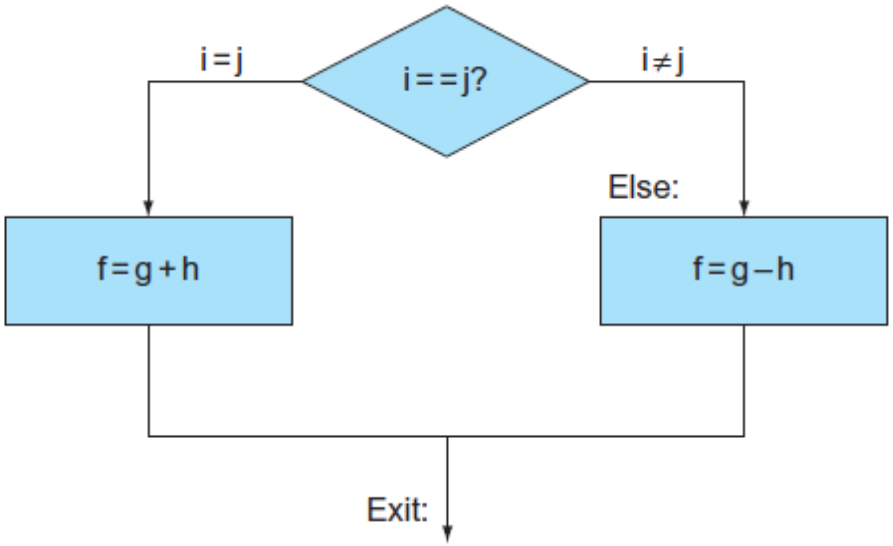
## Salto Condicional

- Em MIPS existem duas instruções para controle de decisões.
- beq avalia se os argumentos possuem valores iguais:
  beq register1, register2, L1
  - L1 será tomado se os valores em register1 e register2 forem iguais.
- bne avalia se os argumentos possuem valores diferentes:
  bne register1, register2, L1
  - L1 será tomado se os valores em register1 e register2 forem diferentes.

# Instruções Para Salto Condicional/Incondicional

## Salto Condicional/Incondicional

- Exemplo 8:
  - No segmento de código abaixo f,g,h,i e j são variáveis. As mesmas estão associadas aos registradores $s0 a $s4. Qual é o código equivalente em MIPS para:
    if (i==j) f = g + h; else f = g - h;
- Resposta



# Instruções Para Salto Condicional/Incondicional

## Salto Condicional/Incondicional

- Resposta:
  - Primeira expressão compara se os elementos são iguais, logo:
    bne $s3, $s4, Else # go to Else if i $\neq$ j
  - Se o caminho Else acima não é tomado, então:
    add $s0, $s1, $s2 # f = g + h (skipped if i $\neq$ j)
  - Finalizando:
    j Exit # go to Exit
  - Tomando o fluxo do label Else
    Else: sub $s0, $s1, $s2 # f = g - h (skipped if i = j)
    Exit:

# Instruções Para Salto Condicional/Incondicional

## Loops

- Praticamente impossível se escrever uma aplicação funcional em linguagem de alto nível sem o uso de loops.
- Denotados através dos termos for e while.
- Com MIPS assembly também é possível se construir estruturas de loop com os saltos condicionais/incondicionais

# Instruções Para Salto Condicional/Incondicional

## Loops

- Exemplo 9
  - Consideremos o seguinte loop em linguagem C:
    while (save[i] == k
    i+=1;
  - Assumindo que as variáveis i e k são os registradores $s3 e $s5 e o endereço base de save[] está no registrador $s6. Qual é o código equivalente em assembly MIPS?
- Resposta
  - Passo 1: Carregar o conteúdo de save[i] em um registrador temporario
    - Passo 1.1: Descobrir o endereço correto de save[i]

# Instruções Para Salto Condicional/Incondicional

## Loops

- Resposta
  - Passo 1: Carregar o conteúdo de save[i] em um registrador temporário.
    - Passo 1.1: Cálculo do offset "i" através do uso da instrução "shift left logical" (sll).
      Loop: sll $t1, $s3, 2 # Temp reg $t1 = i * 4
    - Passo 1.2: Descrobrir o endereço correto de save[i].
      add $t1, $t1, $s6 # $t1 = address of save[i]
    - Passo 1.3: Utilizar o endereço para carregar o valor em um registrador temporário.
      lw $t0,0 ($t1) # Temp reg $t0 = save [i]

# Instruções Para Salto Condicional/Incondicional

## Loops

- Resposta:
  - Passo 2: Implementar o teste do Loop
    bne $t0, $s5, Exit # go to Exit if save[i] $\neq$ k
  - Passo 3: Incremetar o teste do loop
    addi $s3, $s3, 1# i = i +1
  - Passo 4: Finalizando o Loop
    j Loop # go to Loop
    Exit:

# Instruções Para Salto Condicional/Incondicional

## Loops

- Normalmente a ocorrência de saltos condicionais/incondicionais é muito comum em um programa.
- Blocos de instruções sem que não contenham instruções de branch são chamados de basic blocks (bloco básico).
- Primeiras fases da compilação: Separar o programa em basic blocks.
- Por afetarem o fluxo do programa, a tomada de decisão também afeta o desempenho final.

# Instruções Para Salto Condicional/Incondicional

## Loops

- As instruções que visam testar a igualdade/desigualdade de dois valores são muito populares na programação.
- Exemplo: Um loop for testa a cada rodada se uma dada variável é igual a 0 (zero).
- Tais construções são tão comuns que levaram ao suporte de uma instrução para o test e set.
- Em MIPS, existe a instrução set on less than (slt) que compara dois registradores e atribui a um terceiro o valor:
  - 1 se o primeiro registrador é menor que o segundo; ou
  - 0 se o primeiro registrador é maior ou igual ao segundo.

# Instruções Para Salto Condicional/Incondicional

## Loops

- Exemplo 10:
  slt $t0,$s3,$s4 # $t0 = 1 if $s3 < $s4
- $t0 é igual a 1 se o valor em $s3 é menor que o valor em $s4.
- Caso contrário, $t0 é igual a 0
- Versão com operador constante: Verifica se o registrador $s2 é menor que 10
  slti $t0, $s2, 10 # $t0 = 1 if $s2 < 10

# Instruções Para Salto Condicional/Incondicional

## Loops

- Exemplo 11:
  - Considerando que i foi previamente carregado no registrador $s0, qual o equivalente em MIPS para o seguinte trecho de código?
    if (i $<$ 4)
- Resposta:
  slti $t0, $s0, 4
  beq $t0, $zero, L1
- Importante: MIPS faz uso das instruções slt, slti, beq, bne e do valor 0 ($zero) para criar todas as possíveis condições:
  $==, !=, <, <=, >, >=$.

# Instruções Para Salto Condicional/Incondicional

## Loops

- Exercício
  - Faça a tradução do seguinte trecho de código em linguagem C, para a linguagem assembly MIPS:

```
int i,j,a,b;
int soma[10]; //base address = $s4

a = 0; //$s0
b = 0; //$s1
i = 1; //$s2
j = 0; //$s3

if (i <= 1) {
    while (j <= 10) {
        j++;
        soma[j] = j;
    }
} else {
    while (j <= 10) {
        soma[j] = b+1;
        a = soma[j];
        b++;
        a++;
    }
}
```

# Instruções Para Salto Condicional/Incondicional

## Loops

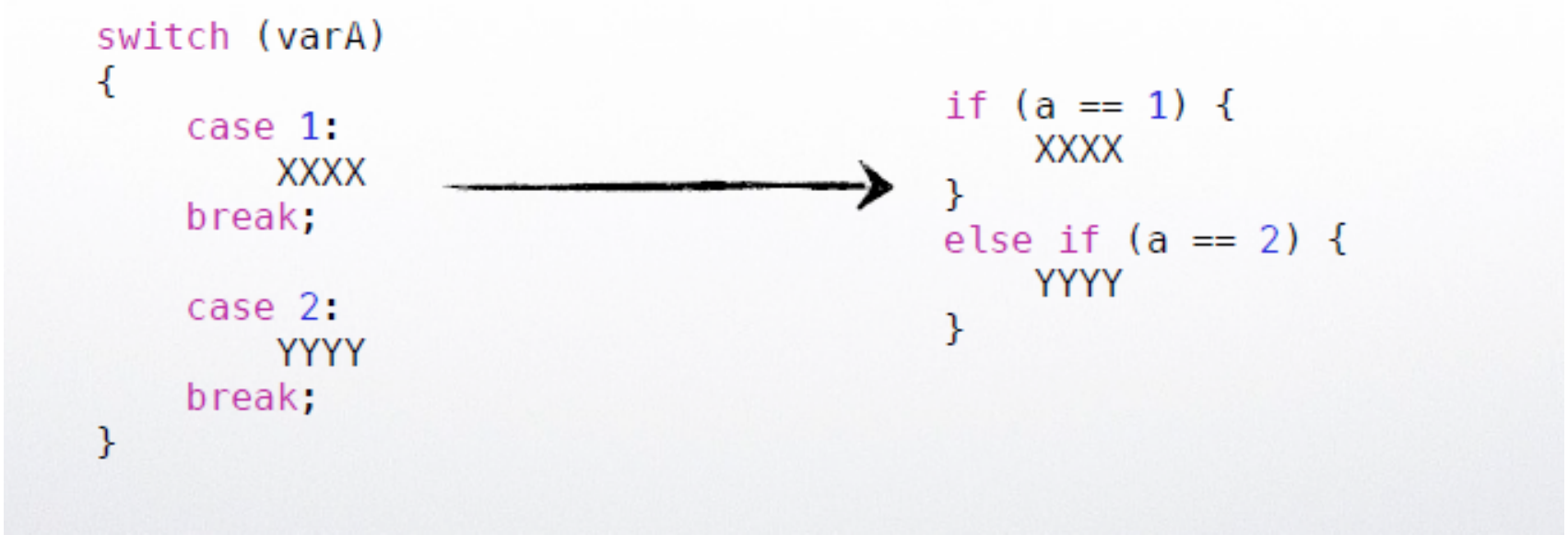
- Resposta:

```
        slti $t0,$s2,2
        beq $t0,$zero,ELSE
LOOP:   slti $t0,$s3,11
        beq $t0,$zero,FIM
        addi $s3,$s3,1
        sll $t1,$s3,2
        add $t1,$t1,$s4
        sw $s3,0($t1)
        j LOOP
ELSE:
LOOP2:  slti $t0,$s3,11
        beq $t0,$zero,FIM
        sll $t1,$s3,2
        add $t1,$t1,$s4
        addi $t2,$s1,1
        sw $t2,0($t1)
        lw $s0,0($t1)
        addi $s1,$s1,1
        addi $s0,$s0,1
        j LOOP2
FIM:
```

# Instruções Para Salto Condicional/Incondicional

## Switch/Case

- Alguns compiladores traduzem o bloco switch/case como uma sequência de if-then-else:



# Instruções Para Salto Condicional/Incondicional

## Switch/Case

- Em alguns casos, a instrução jr - jump register - pode ser usada.
- jr : Salto incondicional cujo endereço se encontra dentro de um registrador.
- Neste caso, o compilador cria um label para cada "case"e coloca na memória.
- Em seguida, seleciona a linha de memória alvo para um registrador e executa a instrução jr.

# Suporte a Procedimentos e Funções

## Introdução

- Procedimentos ou funções são ferramentas utilizadas por programadores para estruturar programas.
- Principais Vantagens:
  - Auxiliam no aumento da legibilidade do programa através de uma melhor organização.
  - Permitem o reuso de código.
  - Facilitam a implementação: Somente pensar em um problema por vez.

# Suporte a Procedimentos e Funções

## Introdução

- Parâmetros agem como uma interface entre a função/ procedimento e o restante do programa e dados.
- Analogia com um espião:
  - Inicia com um plano;
  - Adquire recursos;
  - Executa as tarefas;
  - Cobre os rastros; e
  - Retorna ao seu ponto de origem.

# Suporte a Procedimentos e Funções

## Introdução

- Execução de uma função/procedimento - 6 passos:
  - 1) Alocação dos parâmetros em um local onde o procedimento possa acessá-los;
  - 2) Transferência do controle para o procedimento;
  - 3) Alocação dos recursos armazenados que serão utilizados pelo procedimento;
  - 4) Execução da tarefa desejada;
  - 5) Colocar o valor do resultado em um local onde o programa que chamou possa acessá-lo;
  - 6) Retorno do controle para o ponto de origem, uma vez que o procedimento pode ser chamado à partir de diversos pontos em um programa.

# Suporte a Procedimentos e Funções

## Introdução

- MIPS convenciona o uso de alguns registradores:
- $a0 - $a3: Registradores de argumentos utilizados para a passagem de parâmetros.
- $v0 - $v1: Registradores para retorno de valores.
- $ra: Registrador do endereço de retorno para garantir a volta ao ponto de origem.

# Suporte a Procedimentos e Funções

## Introdução

- MIPS possui instruções de suporte ao uso de procedimentos:
  - Executa o jump para um determinado endereço e simultaneamente salva o endereço da próxima instrução em $ra.
  - A instrução jump and link (jal):
    jal ProcedureAddress
  - link: Um endereço (link) é criado e aponta para a posição que chamou a função de forma a facilitar o retorno.
  - Importante: Faz uso do PC (program counter). Em MIPS conhecido como instruction address register (iar).

# Suporte a Procedimentos e Funções

## Introdução

- MIPS possui instruções de suporte ao uso de procedimentos:
  - Quando o link é feito, $ra = PC + 4$.
  - Função/procedimento é chamado e executado.
  - Volta ao endereço anterior deve ser feita através da chamada à instrução jump register (jr):
    jr $ra
  - Registradores $a0-$a3 são usados para argumentos.

# Suporte a Procedimentos e Funções

## Uso de Registradores em Função/Proc.

- Suponha uma função/procedimento que necessite de quatro parâmetros.
- Ao final da execução os dados utilizados por tais registradores necessitam ser restaurados aos originais.
  - Analogia do espião: cobertura dos rastros.
- A estrutura ideal para o gerenciamento dos registradores é uma pilha (stack).
- Uma pilha mantém sempre um apontador o endereço mais recentemente alocado.

# Suporte a Procedimentos e Funções

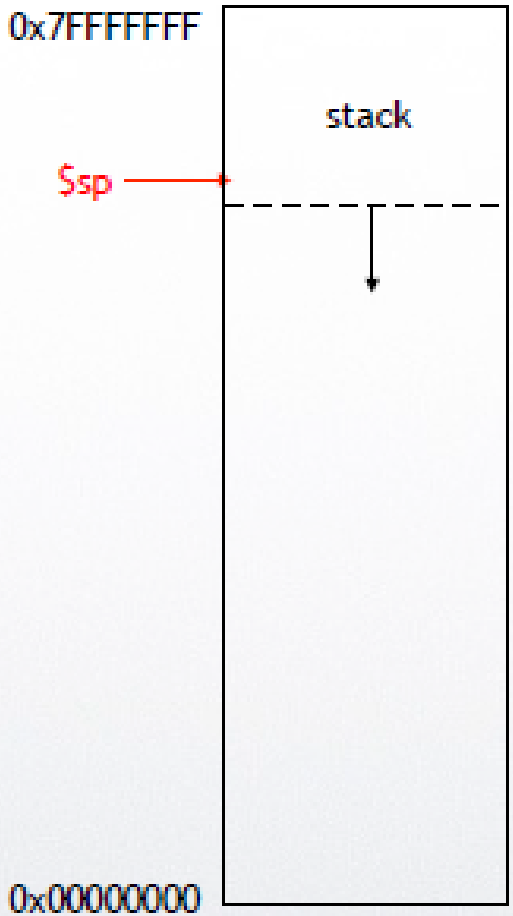
## Uso de Registradores em Função/Proc.

- O apontador de pilha (stack pointer - $sp) é ajustado para cada palavra do registrador colocado ou retirado da pilha.
- Nomenclatura própria:
  - push: Adição de um novo item a pilha;
  - pop: Remoção de um item da pilha.
- Por convenção: Stack cresce de endereços de memória mais altos para os mais baixos.
  - Adição de items se dá pela subtração do $sp.

# Suporte a Procedimentos e Funções

## Uso de Registradores em Função/Proc.

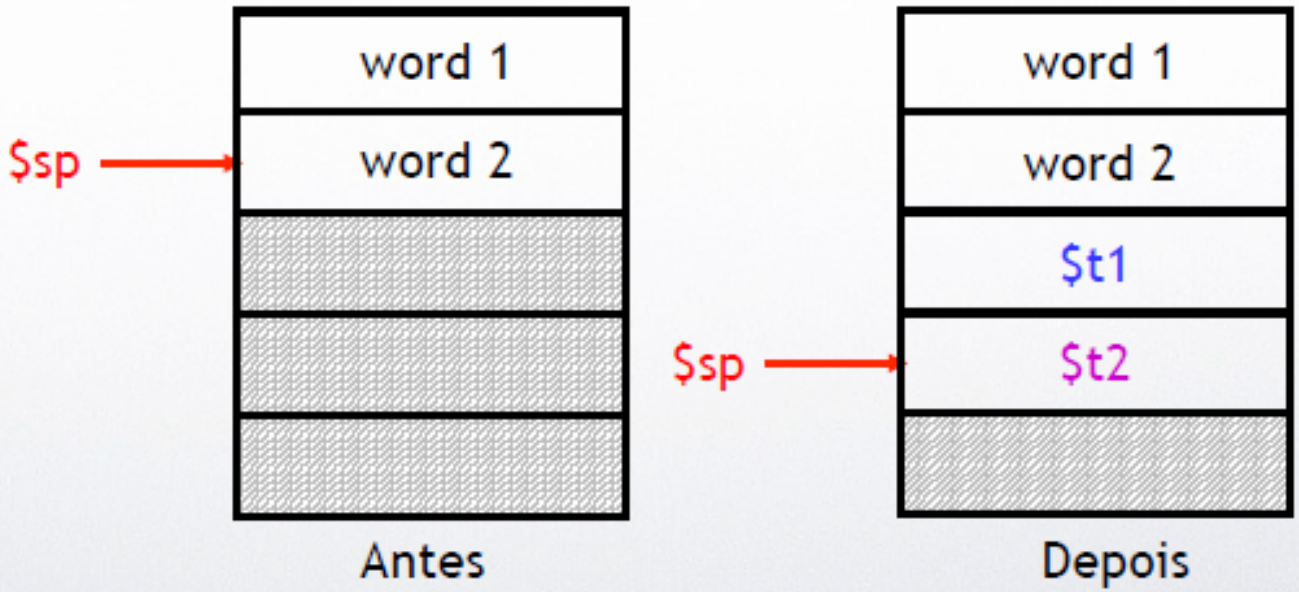
- push e pop deve ser feito pelo programador.



## Uso de Registradores em Função/Proc.

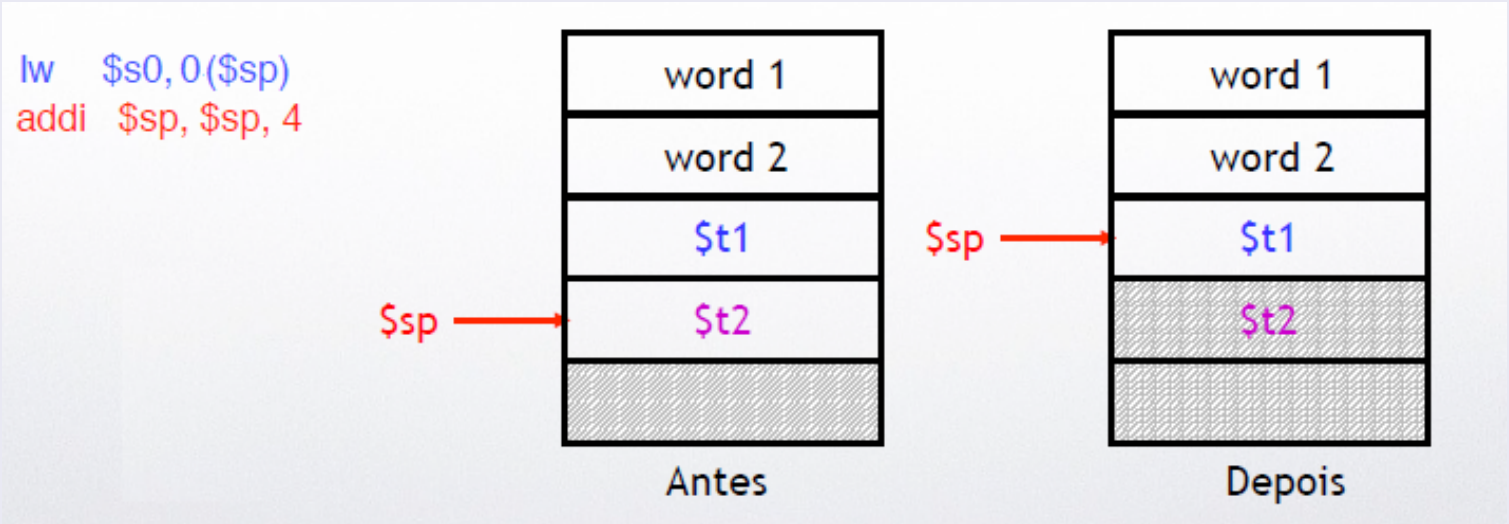
- Para colocar elementos na pilha (push)
  - Mover o stack pointer ($sp) para baixo
- Exemplo

```
subi  $sp, $sp, 8
sw    $t1, 4($sp)
sw    $t2, 0($sp)

   ou:

sw   $t1, -4($sp)
sw   $t2, -8($sp)
sub  $sp, $sp, 8
```



## Uso de Registradores em Função/Proc.

- Para colocar elementos na pilha (push)
  - Mover o stack pointer ($sp) para cima
- Exemplo



# Suporte a Procedimentos e Funções

## Uso de Registradores em Função/Proc.

- Exemplo 12
  - Traduzir a seguinte função em C para assemby MIPS (sem chamada):

```
int leaf_example(int g, int h, int i, int j) {
    int f;

    f = (g + h) - (i + j);
    return f;
}
```

- As variáveis g,h, i e j correspondem respectivamente aos registradores $a0, $a1, $a2 e $a3.
- A variável f está associada ao registrador $s0

## Uso de Registradores em Função/Proc.

- Resposta
  - Passo 1: Salvamento dos registradores utilizados pelo procedimento.
    - Instrução interna ao procedimento:
      $f = (g + h) - (i + j)$;
      add t0, g, h # variável temporária t0 contendo g+h
      add t1, i, j # variável temporária t0 contendo i+j
      sub f, t0, t1 # pega t0 - t1, que é (g+h) -(i+j)
    - Uso de dois registradores temporários ($t0 e $t1).
    - "Apagar os rastros": Valores dos temporários podem já estar sendo utilizados.

## Uso de Registradores em Função/Proc.

- Resposta
  - Passo 1: Salvamento dos registradores utilizados pelo procedimento.
    - Necessário guardar valores anteriores dos registradores a serem utilizados.
    - Guardar valores = Colocá-los na pilha para posterior recuperação.
    - Entretanto, previamente é necessário alocar espaço para 3 registradores (a serem utilizados) na pilha.

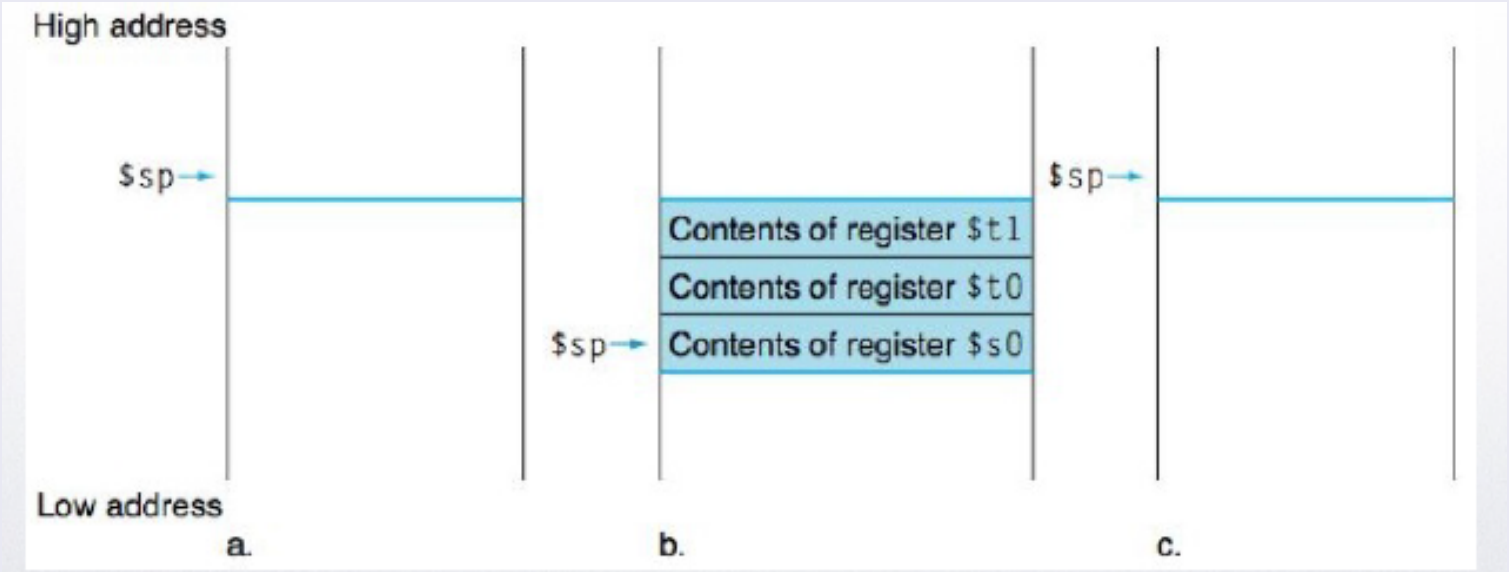
## Uso de Registradores em Função/Proc.

- Resposta
  - Passo 1: Salvamento dos registradores utilizados pelo procedimento.
    - Criando o espaço necessário para três registradores na pilha (12 bytes):

```
addi $sp, $sp, -12 #Ajuste o ponteiro para abrir espaço para 3 itens.
sw $t1, 8($sp) # salva o registrador $t1
sw $t0, 4($sp) # salva o registrador $t0
sw $s0, 0($sp) # salva o registrador $s0
```

## Uso de Registradores em Função/Proc.

- Resposta
  - Passo 1: Salvamento dos registradores utilizados pelo procedimento.



## Uso de Registradores em Função/Proc.

- Resposta
  - Passo 2: Fazer o cálculo da soma expressão:
    add $t0, $a0, $a1 # register $t0 contains g + h
    add $t1, $a2, $a3 # register $t1 contains i + j
    sub $s0, $t0, $t1 # f = $t0 - $t1, que é (g + h) - (i + j)
  - Passo 3: Copiar o valor de retorno para o registrador de retorno (return register)
    add $v0, $s0, $zero # retorno f ($v0 = $s0 + 0)

## Uso de Registradores em Função/Proc.

- Resposta
  - Passo 4: Antes de retornar, fazer a restauração dos três valores previamente salvos na pilha (pop):
    lw $s0,0($sp) # recupera registrador $s0 para quem chamou
    lw $t0,4($sp) # recupera registrador $t0 para quem chamou
    lw $t1,8($sp) # recupera registrador $t1 para quem chamou
    addi $sp, $sp, 12) # ajusta o ponteiro para deletar 3 items
  - Passo 5: Retornar o fluxo de execução para o endereço de retorno:
    jr $ra # jump back to calling routine

# Suporte a Procedimentos e Funções

## Uso de Registradores em Função/Proc.

- Argumentos são passados nos registradores $a0 a $a3.
- Limite de 4 argumentos por função.
- O que fazer se uma função possuir $> 4$ argumentos? Como fazer uso de mais registradores?
  - Inicialmente faz-se necessário um melhor entendimento do funcionamento da pilha.