



**Universidade do Estado do Rio de Janeiro**  
**Instituto de Matemática e Estatística**  
**Departamento de Informática e Ciência da Computação**

# Arquitetura de Computadores I

Linguagem de Máquina  
por  
Jacques Alves da Silva

# Introdução

- Para controlar o hardware de um computador, é preciso falar a sua linguagem
- As palavras da linguagem de um computador são chamadas **instruções**
- O **conjunto de instruções** formam o vocabulário dos comandos entendidos por uma determinada arquitetura

# Introdução

- As linguagens de computador são muito semelhantes, visto que:
  - Todos os computadores são construídos a partir de tecnologias de hardware baseadas em princípios básicos semelhantes
  - Existem algumas operações básicas que todos os computadores precisam fornecer

# Introdução

- Além do mais, os projetistas de computador possuem um objetivo comum: encontrar uma linguagem que facilite o projeto do hardware e do compilador enquanto maximiza o desempenho e minimiza o custo

# Introdução

- *É fácil ver, por métodos lógicos formais, que existem certos [conjuntos de instruções] que são adequados para controlar e causar a execução de qualquer sequência de operações... As considerações realmente decisivas, do ponto de vista atual, na seleção de um [conjunto de instruções], são mais de natureza prática: a **simplicidade** do equipamento exigido pelo [conjunto de instruções] e a **clareza** de sua aplicação para os problemas realmente importantes, junto com a velocidade com que tratam esses problemas*
  - *Burks, Goldstine e von Neumann, 1947*

# Introdução

- Neste capítulo será mostrado o conjunto de instruções de um computador real, tanto na forma escrita pelos humanos quanto na forma lida pelo computador
- Apresentação das instruções em uma abordagem top-down

# Introdução

- O conjunto de instruções escolhido vem do MIPS
  - Conjunto de instruções criado a partir da década de 1980
  - Quase 100 milhões de processadores fabricados em 2002
  - Encontrados em produtos da Cisco, NEC, Nintendo, Silicon Graphics, Sony e Toshiba, entre outros

# Operações do Hardware do Computador

- Todo computador é capaz de realizar operações aritméticas
- A notação em assembly do MIPS

add a, b, c

Instrui um computador a somar as duas variáveis **b** e **c** para colocar o resultado da soma em **a**



# Operações do Hardware do Computador

- Cada instrução aritmética do MIPS realiza apenas uma operação e sempre precisa ter exatamente três variáveis
- Para a instrução abaixo em C, qual é a sequência de instruções em assembly para a soma das quatro variáveis?

$a = b + c + d + e$

# Operações do Hardware do Computador

- A sequência de instruções a seguir soma as quatro variáveis:

add a, b, c     # A soma  $b + c$  está em a

add a, a, d     # A soma  $b + c + d$  está em a

add a, a, e     # A soma  $b + c + d + e$  está em a

- As palavras à direita do símbolo # são comentários e são ignorados pelo computador

# Operações do Hardware do Computador

- Cada linha dessa linguagem pode conter apenas uma instrução
- O número de operandos de operações aritméticas é exatamente 3: 2 operandos fonte e 1 destino. Essa regra simplifica o hardware

# Operações do Hardware do Computador

- O hardware para um número variável de operandos é mais complicado do que o hardware para um número fixo

Princípio de Projeto 1: simplicidade favorece a regularidade

# Operações do Hardware do Computador

- Considere o seguinte trecho de código de um programa em C:

`a = b + c;`

`d = a - e;`

- As duas instruções são compiladas diretamente nessas duas instruções em assembly do MIPS

`add a, b, c`

`sub d, a, e`

# Operações do Hardware do Computador

- Considere uma instrução mais complexa com cinco variáveis **f**, **g**, **h**, **i** e **j**

$$f = (g + h) - (i + j);$$

O que um compilador C poderia produzir?

# Operações do Hardware do Computador

- O compilador precisa desmembrar essa instrução em várias instruções assembly, pois somente uma operação é realizada por instrução MIPS
- Além disso, o compilador precisa criar variáveis temporárias para armazenar os resultados parciais

# Operações do Hardware do Computador

- Para a instrução em C:

$f = (g + h) - (i + j);$

- O compilador poderia produzir o seguinte código em assembly do MIPS:

```
add t0, g, h  # variável temporária t0 recebe g + h
add t1, i, j   # variável temporária t1 recebe i + j
sub f, t0, t1  # f recebe t0 - t1 = (g + h) - (i + j)
```



# Operandos do Hardware do Computador

- Os operandos das instruções aritméticas são os **registradores**
- Os registradores são primitivas usadas no projeto do hardware que também são visíveis ao programador
- Os registradores são usados para armazenar informações internamente no processador

# Operandos do Hardware do Computador

- O tamanho de um registrador na arquitetura MIPS é de 32 bits
- Os grupos de 32 bits ocorrem com tanta frequência na arquitetura MIPS, que são denominados de **word** (palavra)

# Operandos do Hardware do Computador

- Uma diferença entre as variáveis de uma linguagem de programação e os registradores é o número limitado de registradores
- O banco de registradores da arquitetura MIPS é formado de 32 registradores

# Operandos do Hardware do Computador

- Existe a restrição de que cada um dos três operandos das instruções aritméticas do MIPS precisa ser escolhido a partir de um dos 32 registradores de 32 bits
- O limite dos 32 registradores está relacionado com:

Princípio de Projeto 2: menor significa mais rápido

# Operandos do Hardware do Computador

- Uma grande quantidade de registradores pode aumentar o tempo do ciclo de clock simplesmente porque os sinais eletrônicos levam mais tempo quando precisam atravessar uma distância maior
- O projetista de computador precisa equilibrar o desejo dos programas por mais registradores com o desejo do projetista de manter o ciclo de clock rápido

# Operandos do Hardware do Computador

- Outro motivo para não se utilizar mais de 32 registradores é o número de bits que seria necessário no formato da instrução
- Pela convenção do MIPS, os registradores são identificados com um sinal de cifrão seguido por dois caracteres

# Operandos do Hardware do Computador

- O compilador tem a tarefa de associar as variáveis do programa aos registradores
- Considere a instrução de atribuição do exemplo anterior:  
$$f = (g + h) - (i + j);$$
- As variáveis **f**, **g**, **h**, **i** e **j** são associadas aos registradores \$s0, \$s1, \$s2, \$s3 e \$s4, respectivamente
- Qual seria o código MIPS compilado?

# Operandos do Hardware do Computador

- O programa compilado é semelhante ao anterior, exceto por:
  - Substituição das variáveis pelos nomes dos registradores
  - Dois registradores temporários, \$t0 e \$t1, que correspondem às variáveis temporárias

```
add $t0, $s1, $s2    # registrador $t0 recebe g + h
add $t1, $s3, $s4    # registrador $t1 recebe i + j
sub $s0, $t0, $t1    # f recebe $t0 - $t1
```



# Operandos em Memória

- Estruturas de dados complexas podem conter mais elementos de dados do que a quantidade de registradores disponíveis no processador
- Entretanto, a memória do computador pode conter milhões de elementos de dados. Logo, as estruturas de dados, tais como arrays, são mantidas na memória

# Operandos em Memória

- Nas instruções MIPS, as operações aritméticas são executadas sobre registradores
- Assim, o MIPS precisa incluir instruções que transferem dados entre a memória e os registradores. Essas instruções são denominadas **instruções de transferência de dados**

# Operandos em Memória

- Para acessar uma palavra (ou *word*) na memória, é necessário que a instrução forneça o **endereço** de memória da palavra
- A memória pode ser imaginada como um *array* unidimensional (vetor), e o endereço de memória de uma palavra como o índice deste *array*

# Operandos em Memória

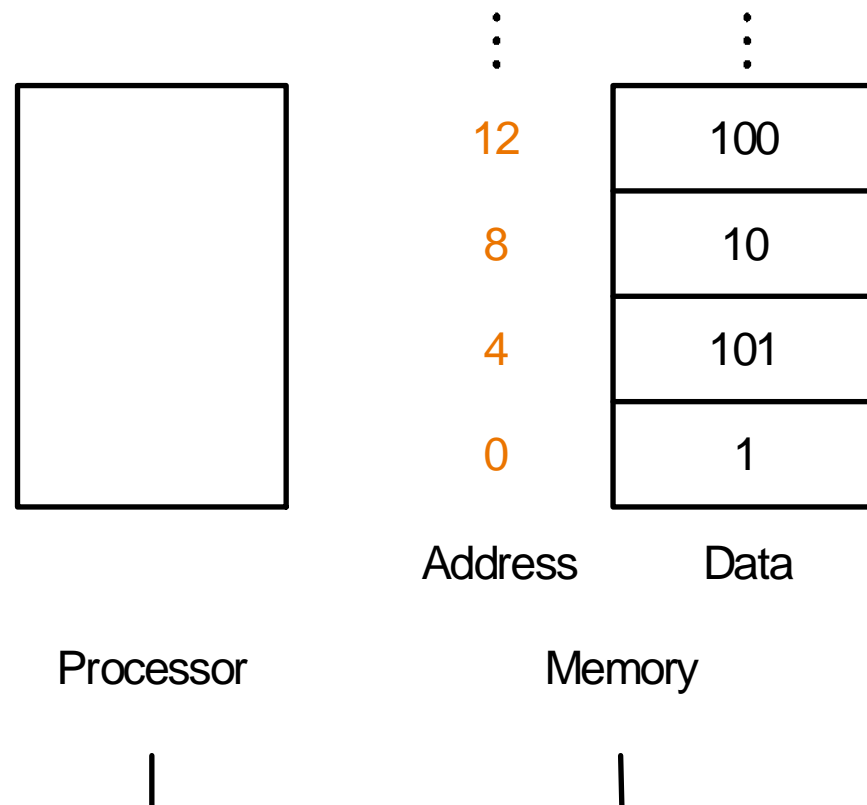
- Além de associar variáveis a registradores, o compilador aloca estrutura de dados em locais na memória
- Desta maneira, o compilador pode colocar o endereço inicial apropriado nas instruções de transferência de dados

# Operandos em Memória

- A arquitetura MIPS endereça bytes e não palavras. Portanto, o endereço de uma palavra na memória combina os endereços dos 4 bytes dentro da palavra
- Os endereços sequenciais das palavras diferem em 4 vezes, ou seja, o índice  $i$  da palavra tem um endereço igual a  $i \times 4$

# Operandos em Memória

- Endereços de memória reais do MIPS e conteúdo da memória para essas palavras



# Operandos em Memória

- Instruções de transferência de dados:
  - *load word* (lw): copia uma palavra da memória para um registrador
  - *store word* (sw): copia uma palavra de um registrador para a memória

# Operandos em Memória

- O formato da instrução lw e sw:
  - Nome da operação, seguido pelo registrador a ser carregado (lw) ou a ser armazenado (sw), depois o *offset* (deslocamento) para selecionar o elemento do *array* e por último o registrador base (endereço inicial do *array*)
- O endereço MIPS é especificado em parte por uma constante e em parte pelo conteúdo de um registrador



# Operandos em Memória

- Suponha que *A* seja uma sequência de 100 palavras e que o compilador tenha associado a variável *h* ao registrador \$s2, e o endereço base do *array A* esteja em \$s3. Qual é o código assembly do MIPS para a instrução em C a seguir?

$A[12] = h + A[8];$

# Operandos em Memória

- Instrução em C:

$A[12] = h + A[8];$

- Código MIPS:

```
lw    $t0, 32($s3)    # $t0 recebe A[8]
add   $t0, $s2, $t0    # $t0 recebe h + A[8]
sw    $t0, 48($s3)    # Armazena h + A[8] em A[12]
```

# Operandos em Memória

- Sabe-se que a maioria dos programas possuem um número de variáveis maior do que o número de registradores disponíveis. O compilador tenta manter as variáveis mais utilizadas nos registradores e armazena as restantes na memória. Esse processo é conhecido como *spilling registers*

# Constantes ou Operandos Imediatos

- Muitas vezes, um programa usará uma constante em uma operação
  - Exemplo: incremento de índice
- Usando apenas as instruções vistas até o momento, seria necessário ler uma constante da memória para utilizá-la
  - Utilização da instrução *load*

# Constantes ou Operandos Imediatos

- Uma alternativa é oferecer versões das instruções aritméticas em que o operando seja uma constante
- A instrução add rápida, com uma constante no lugar do operando, é denominada **add imediato**, ou **addi**

`addi $s3, $s3, 4`     $\# \text{ \$s3} = \text{\$s3} + 4$

# Constantes ou Operandos Imediatos

- Os operandos constantes ocorrem com frequência. As instruções aritméticas se tornam muito mais rápidas com a inclusão das constantes dentro dela do que se as constantes fossem lidas da memória
- As instruções imediatas ilustram:

Princípio de Projeto 3: agilize os casos mais comuns

# Representando Instruções no Computador

- Nós somos ensinados a pensar na base 10, mas os números podem ser representados em qualquer base
  - Por exemplo  $123_{10} = 1111011_2$
- No hardware do computador, os números são mantidos como uma série de sinais eletrônicos altos e baixos e, por isso, são considerados números na base 2

# Representando Instruções no Computador

- As instruções podem ser representadas como números
  - Cada parte da instrução pode ser considerada um número individual (campos da instrução)
  - A colocação desses números lado a lado forma a instrução



# Representando Instruções no Computador

- Os registradores fazem parte de quase todas as instruções do MIPS. Assim, se torna necessária uma convenção para mapear nomes de registrador em números
  - Os registradores \$s0 a \$s7 são mapeados nos registradores de 16 a 23
  - Os registradores \$t0 a \$t7 são mapeados nos registradores de 8 a 15

# Representando Instruções no Computador

- Considere a instrução MIPS:  
**add \$t0, \$s1, \$s2**
- A representação decimal é:

|   |    |    |   |   |    |
|---|----|----|---|---|----|
| 0 | 17 | 18 | 8 | 0 | 32 |
|---|----|----|---|---|----|

- O primeiro e o último campos (0 e 32) indicam que essa instrução realiza soma
- O segundo e terceiro campos (17 e 18) são os operandos fontes
- O quarto campo é o número do registrador que receberá a soma

# Representando Instruções no Computador

- Essa instrução representada com seus campos em números binários

|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
| 000000 | 10001  | 10010  | 01000  | 00000  | 100000 |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

- Todas as instruções MIPS possuem 32 bits, que é o mesmo tamanho de uma palavra

# Representando Instruções no Computador

- O *layout* da instrução é denominado **formato de instrução**
- A versão binária das instruções é denominada de **linguagem de máquina**, e a sequência dessas instruções é o código de máquina

# Representando Instruções no Computador

- Os campos da instrução MIPS recebem os seguintes nomes:

|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
| op     | rs     | rt     | rd     | shamt  | funct  |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

- op: operação (**opcode**)
- rs: registrador do primeiro operando fonte
- rt: registrador do segundo operando fonte
- rd: registrador do operando destino
- shamt: *shift amount*
- funct: função

# Representando Instruções no Computador

- Existe um problema quando uma instrução requer campos maiores
  - Por exemplo, a instrução *load word* requer dois registradores e uma constante. Se a constante tivesse que usar um dos campos de 5 bits, ela seria limitada a apenas  $2^5$ , ou 32. Esse campo de 5 bits é muito pequeno para selecionar o elemento do *array* (deslocamento)

# Representando Instruções no Computador

- Existe um conflito entre manter todas as instruções com o mesmo tamanho e ter um formato de instrução único

Princípio de Projeto 4: um bom projeto exige bons compromissos

# Representando Instruções no Computador

- O compromisso escolhido pelos projetistas do MIPS é manter todas as instruções com o mesmo tamanho, sendo exigidos diferentes tipos de formatos para diferentes tipos de instruções
- O formato anterior é denominado de **tipo R** (de registrador) ou **formato R**



# Representando Instruções no Computador

- Um segundo tipo de formato de instrução é denominado de **tipo I** (de imediato) ou **formato I**, que é utilizado pelas instruções imediatas e de transferência de dados
- Os campos do formato I são:

|        |        |        |                       |
|--------|--------|--------|-----------------------|
| op     | rs     | rt     | constante ou endereço |
| 6 bits | 5 bits | 5 bits | 16 bits               |

# Representando Instruções no Computador

- Os três primeiros campos nos formatos de tipo R e tipo I possuem o mesmo tamanho e têm os mesmos nomes
- O quarto campo do tipo I é igual ao tamanho dos três últimos campos do tipo R
- As instruções com formato do tipo R e tipo I são diferenciadas pelo campo **op** (opcode)

# Representando Instruções no Computador

- Suponha a instrução em C:

**$A[300] = h + A[300];$**

- Se \$t1 possui a base do *array* A e \$s2 corresponde a h, então o código assembly:

**lw     \$t0, 1200(\$t1) # \$t0 recebe A[300]**

**add    \$t0, \$s2, \$t0    # \$t0 recebe h + A[300]**

**sw     \$t0, 1200(\$t1) # Armazena h + A[300] em A[300]**

# Representando Instruções no Computador

- O código em linguagem de máquina MIPS:
  - Representação decimal

|           |                         | op        | rs       | rt       | endereço    |
|-----------|-------------------------|-----------|----------|----------|-------------|
| <b>lw</b> | <b>\$t0, 1200(\$t1)</b> | <b>35</b> | <b>9</b> | <b>8</b> | <b>1200</b> |

|            |                         | op       | rs        | rt       | rd       | shamt    | func      |
|------------|-------------------------|----------|-----------|----------|----------|----------|-----------|
| <b>add</b> | <b>\$t0, \$s2, \$t0</b> | <b>0</b> | <b>18</b> | <b>8</b> | <b>8</b> | <b>0</b> | <b>32</b> |

|           |                         | op        | rs       | rt       | endereço    |
|-----------|-------------------------|-----------|----------|----------|-------------|
| <b>sw</b> | <b>\$t0, 1200(\$t1)</b> | <b>43</b> | <b>9</b> | <b>8</b> | <b>1200</b> |

# Representando Instruções no Computador

- O código em linguagem de máquina MIPS:
  - Representação binária

|        |       |       |                     |       |        |
|--------|-------|-------|---------------------|-------|--------|
| 100011 | 01001 | 01000 | 0000 0100 1011 0000 |       |        |
| 000000 | 10010 | 01000 | 01000               | 00000 | 100000 |
| 101011 | 01001 | 01000 | 0000 0100 1011 0000 |       |        |

- A única diferença entre a primeira e a última instrução está no terceiro bit da esquerda para a direita

# Operações Lógicas

- Embora os primeiros computadores se concentrassem em palavras completas, logo se notou a utilidade de atuar sobre campos de bits dentro de uma palavra ou até mesmo sobre bits individuais
  - Por exemplo, examinar caracteres dentro de uma palavra, cada um dos quais ocupam 8 bits
- Essas instruções são denominadas de **operações lógicas**

# Operações Lógicas

- Operadores lógicos em C e Java e suas instruções MIPS correspondentes

| Operações Lógicas | Operadores C | Operadores Java | 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

- A primeira classe dessas operações é denominada de *shifts* (deslocamentos)
  - Movem todos os bits de uma palavra para a esquerda ou direita, preenchendo os bits que ficaram vazios com zeros



# Operações Lógicas

- Por exemplo, se o registrador \$s0 contém  
0000 0000 0000 0000 0000 0000 0000 1001<sub>2</sub> = 9<sub>10</sub>
- A instrução para deslocar 4 bits à esquerda gera o seguinte novo valor:  
0000 0000 0000 0000 0000 0000 1001 0000<sub>2</sub> = 144<sub>10</sub>
- O complemento de um *shift* à esquerda é um *shift* à direita. Os nomes reais das duas instruções no MIPS são *shift left logical* (**sll**) e *shift right logical* (**srl**)

# Operações Lógicas

- A instrução a seguir realiza essa operação, supondo o resultado em \$t2

**sl**    **\$t2, \$s0, 4**    # \$t2 = \$s0 << 4 bits

- O campo **shamt** do formato R significa *shift amount* (quantidade de deslocamento) e é usado nas instruções de deslocamento

# Operações Lógicas

- A versão em linguagem de máquina da instrução em representação decimal é:

| op | rs | rt | rd | shamt | funct |
|----|----|----|----|-------|-------|
| 0  | 0  | 16 | 10 | 4     | 0     |

- A codificação de **sll** é 0 nos campos op e funct, rd contém \$t2, rt contém \$s0 e shamt contém 4. O campo rs não é utilizado e é definido como 0

# Operações Lógicas

- O deslocamento à esquerda de  $i$  bits gera o mesmo resultado que multiplicar por  $2^i$ 
  - No exemplo anterior,  $9 \times 2^4 = 9 \times 16 = 144$
- Outra operação lógica é o AND bit a bit, que gera um 1 no resultado somente se os dois bits dos operandos forem 1

# Operações Lógicas

- Por exemplo, se o registrador \$t2 contém  
0000 0000 0000 0000 0000 1101 0000 0000<sub>2</sub>  
e o registrador \$t1 contém  
0000 0000 0000 0000 0011 1100 0000 0000<sub>2</sub>  
depois de executar a instrução MIPS  
**and \$t0, \$t1, \$t2** # \$t0 = \$t1 & \$t2  
o valor do registrador \$t0 seria  
0000 0000 0000 0000 0000 1100 0000 0000<sub>2</sub>

# Operações Lógicas

- A versão em linguagem de máquina da instrução em representação decimal é:

| op | rs | rt | rd | shamt | funct |
|----|----|----|----|-------|-------|
| 0  | 9  | 10 | 8  | 0     | 36    |

- A codificação de **and** é 0 no campo op e 36 no campo funct, rs contém \$t1, rt contém \$t2, rd contém \$t0. O campo shamt não é utilizado e é definido como 0

# Operações Lógicas

- A instrução AND pode aplicar um padrão de bits a um conjunto de bits para forçar zeros onde houver um 0 no padrão de bits
  - Esse padrão de bits, em conjunto com o AND, é denominado de máscara, pois “oculta” alguns bits

# Operações Lógicas

- Existe o complemento da instrução AND, que é denominado OR. Essa é uma operação bit a bit, que produz um 1 no resultado se um ou os dois bits dos operandos forem 1



# Operações Lógicas

- Por exemplo, se o registrador \$t2 contém  
0000 0000 0000 0000 0000 1101 0000 0000<sub>2</sub>  
e o registrador \$t1 contém  
0000 0000 0000 0000 0011 1100 0000 0000<sub>2</sub>  
depois de executar a instrução MIPS  
**or \$t0, \$t1, \$t2** # \$t0 = \$t1 | \$t2  
o valor do registrador \$t0 seria  
0000 0000 0000 0000 0011 1101 0000 0000<sub>2</sub>

# Operações Lógicas

- A versão em linguagem de máquina da instrução em representação decimal é:

| op | rs | rt | rd | shamt | funct |
|----|----|----|----|-------|-------|
| 0  | 9  | 10 | 8  | 0     | 37    |

- A codificação de **or** é 0 no campo op e 37 no campo funct, rs contém \$t1, rt contém \$t2, rd contém \$t0. O campo shamt não é utilizado e é definido como 0

# Operações Lógicas

- A última operação lógica é a negação. O NOT recebe um operando e produz um 1 no resultado se um bit do operando for 0, e vice-versa
- Seguindo o formato de dois operandos, os projetistas do MIPS incluíram a instrução NOR (NOT OR) no lugar de NOT. Se um operando for zero, então equivale a NOT
  - Exemplo:  $A \text{ NOR } 0 = \text{NOT}(A \text{ OR } 0) = \text{NOT}(A)$

# Operações Lógicas

- Por exemplo, se o registrador \$t1 contém

0000 0000 0000 0000 0011 1100 0000 0000<sub>2</sub>

e o registrador \$t3 contém

0000 0000 0000 0000 0000 0000 0000 0000<sub>2</sub>

depois de executar a instrução MIPS

**nor \$t0, \$t1, \$t3**    #  $\$t0 = \sim (\$t1 \mid \$t3)$

o valor do registrador \$t0 seria

1111 1111 1111 1111 1100 0011 1111 1111<sub>2</sub>

# Operações Lógicas

- A versão em linguagem de máquina da instrução em representação decimal é:

| op | rs | rt | rd | shamt | funct |
|----|----|----|----|-------|-------|
| 0  | 9  | 11 | 8  | 0     | 39    |

- A codificação de **nor** é 0 no campo op e 39 no campo funct, rs contém \$t1, rt contém \$t3, rd contém \$t0. O campo shamt não é utilizado e é definido como 0

# Operações Lógicas

- As constantes são úteis nas operações lógicas AND e OR, de modo que o MIPS oferece as instruções **and imediato (andi)** e **or imediato (ori)**
- Visto que o principal objetivo da operação NOR é inverter os bits de um único operando, o hardware não possui uma versão imediata para esta instrução

# Instruções de Decisão

- Com base em dados de entrada e valores gerados durante o processamento, diferentes instruções são executadas
- O assembly do MIPS inclui duas instruções para tomadas de decisões, que são parecidas com uma instrução *if* combinada com uma instrução *go to*

# Instruções de Decisão

- A primeira instrução é:  
**beq registrador1, registrador2, L1**
- Esta instrução desvia a execução para a instrução rotulada **L1**, se o valor no **registrador1** for igual ao do **registrador2**
- O mnemônico **beq** significa *branch if equal* (desvia se for igual)



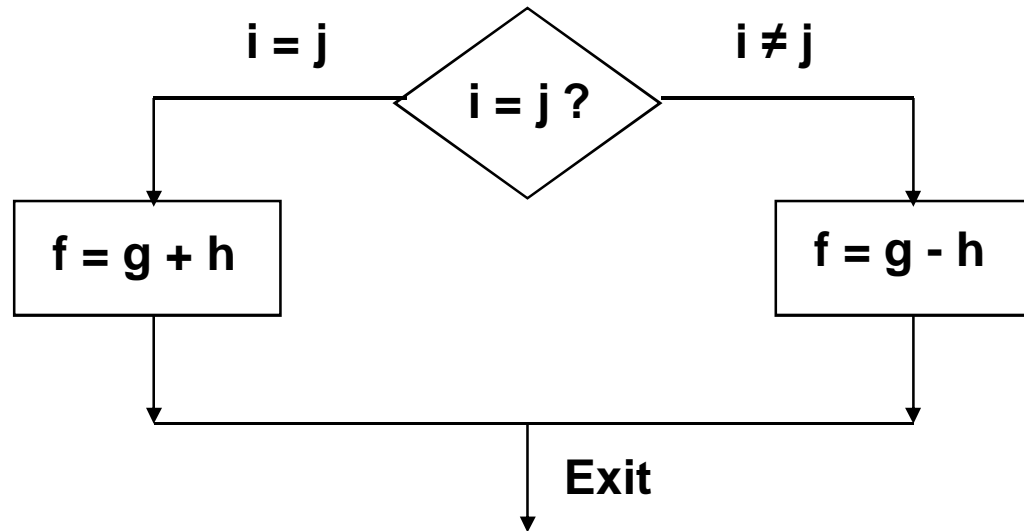
# Instruções de Decisão

- A segunda instrução é:  
**bne registrador1, registrador2, L1**
- Esta instrução desvia a execução para a instrução rotulada **L1**, se o valor no **registrador1** não for igual ao do **registrador2**
- O mnemônico **bne** significa *branch if not equal* (desvia se não for igual)

# Instruções de Decisão

- Estas duas instruções são denominadas **desvios condicionais**
- Considere o seguinte trecho de código em C, onde as variáveis de f a j correspondem aos cinco registradores de \$s0 a \$s4  
**if (i == j) f = g + h; else f = g - h;**
- Qual é o código MIPS compilado para esta instrução *if* em C?

# Instruções de Decisão



- Código assembly MIPS

```
    bne $s3, $s4, Else
    add $s0, $s1, $s2
    j    Exit
Else: sub $s0, $s1, $s2
Exit:
```

# Instruções de Decisão

- O exemplo anterior apresenta outro tipo de desvio denominado **desvio incondicional**
  - O processador sempre deverá seguir o desvio
- O nome MIPS para esta instrução é **jump**, representada por **j**
- Os campos do **formato J** ou **tipo J** são:

|        |          |
|--------|----------|
| op     | endereço |
| 6 bits | 26 bits  |

# Loops

- Considere o seguinte *loop* em C:

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

onde as variáveis *i* e *k* correspondem aos registradores \$s3 e \$s5 e a base do *array* esteja em \$s6

- Qual é o código assembly MIPS para este segmento em C?

# Loops

- Código assembly MIPS

```
Loop:    sll    $t1, $s3, 2           # $t1 = 4 * i  
          add    $t1, $t1, $s6  
          lw     $t0, 0($t1)  
          bne    $t0, $s5, Exit  
          addi   $s3, $s3, 1  
          j      Loop
```

**Exit:**

# Instruções de Decisão

- Comando de teste **A < B**
- A instrução **slt** (*set on less than* – atribuir se menor que): compara o conteúdo de dois registradores fontes
  - RegDest = 1 se **A < B**
  - RegDest = 0 se **A ≥ B**

# Instruções de Decisão

- Código para testar se **A** (\$s0) é menor que **B** (\$s1) e desviar para **Less** se a condição é verdadeira
- MIPS possui registrador com conteúdo fixo igual a 0 (\$zero)
- Código assembly MIPS

```
slt    $t0, $s0, $s1
bne    $t0, $zero, Less
```



# Instruções de Decisão

- Comando switch/case
- Seleção de uma entre várias alternativas
- Seleção depende do valor de uma variável
- O modo mais simples de implementar um *switch* é por meio de uma sequência de testes condicionais
  - Cadeia de instruções *if-then-else*

# Instruções de Decisão

- Outro modo de implementar um *switch* é por meio de uma **tabela de endereços de desvio**
  - *Array* de palavras contendo endereços
- O programa carrega a entrada apropriada da tabela de desvios para um registrador, e depois desvia para o endereço apropriado
  - Instrução *jump register (jr)*: Desvio incondicional para endereço especificado em um registrador

# Instruções de Decisão

- Considere o seguinte trecho de código em C, onde as variáveis de **f** a **k** correspondem aos seis registradores de \$s0 a \$s5, \$t2 = 4 e \$t4 possui o endereço base da tabela de desvios

```
switch (k) {  
    case 0: f = i + j; break;  
    case 1: f = g + h; break;  
    case 2: f = g - h; break;  
    case 3: f = i - j;  
}
```

# Instruções de Decisão

- Seleção de uma entre quatro alternativas dependendo do valor de **k**
  - **k = {0, 1, 2, 3}**

# Instruções de Decisão

```
slt    $t3, $s5, $zero
bne    $t3, $zero, Exit
slt    $t3, $s5, $t2
beq    $t3, $zero, Exit
sll    $t1, $s5, 2
add    $t1, $t1, $t4
lw     $t0, 0($t1)
jr     $t0
L0:    add    $s0, $s3, $s4
        j     Exit
L1:    add    $s0, $s1, $s2
        j     Exit
L2:    sub    $s0, $s1, $s2
        j     Exit
L3:    sub    $s0, $s3, $s4
Exit:
```

# Suporte a Chamada de Procedimentos

- A utilização de um procedimento ou função tem a finalidade de estruturar os programas
  - Facilidade de entendimento
  - Reutilização do código

# Suporte a Chamada de Procedimentos

- Os seis passos a serem realizados na execução de um procedimento:
  - Armazenar os parâmetros para acesso
  - Transferir controle para o procedimento
  - Garantir recursos de memória para execução
  - Realizar tarefa
  - Colocar resultado de modo acessível
  - Retornar programa ao ponto de origem

# Suporte a Chamada de Procedimentos

- O processador MIPS utiliza registradores especiais para chamada de procedimentos:
  - \$a0 - \$a3: quatro registradores para parâmetros
  - \$v0 - \$v1: dois registradores para valores de retorno
  - \$ra: registrador que contém o endereço de retorno



# Suporte a Chamada de Procedimentos

- O assembly do MIPS inclui uma instrução apenas para os procedimentos:
  - A instrução denominada *jump-and-link* (**jal**)
    - **jal EndereçoProcedimento**
- Esta instrução desvia para um endereço e simultaneamente salva o endereço da instrução seguinte no registrador \$ra
- O valor armazenado no registrador \$ra é chamado de **endereço de retorno**

# Suporte a Chamada de Procedimentos

- O registrador que mantém o endereço da instrução sendo executada é denominado **contador de programa**, abreviado como **PC** (*Program Counter*)
- A instrução **jal** armazena o valor  $PC + 4$  no registrador \$ra para estabelecer o *link* de retorno, e a instrução que implementa o retorno é a instrução *jump register*

# Suporte a Chamada de Procedimentos

- Suponha que o compilador precise de mais registradores para um procedimento do que os quatro para argumentos e os dois para valores de retorno
  - Após a execução do procedimento **chamado** (*callee*), quaisquer registradores necessários ao procedimento **chamador** (*caller*) deverão ser restaurados aos valores que possuíam antes do procedimento ser chamado
  - Utilizar a memória

# Suporte a Chamada de Procedimentos

- A estrutura de dados usada para armazenar os valores em memória é a **pilha**
  - Comportamento **LIFO** (*Last In First Out* – Último a entrar, primeiro a sair)
  - O registrador **\$sp** (*stack pointer*) armazena o endereço do topo da pilha
  - Operações:
    - *Push* – Inserir dados na pilha
    - *Pop* – Retirar dados da pilha
  - Por convenção os endereços são dos maiores para os menores

# Suporte a Chamada de Procedimentos

- Considere o seguinte procedimento em C:

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

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

- Qual é o código assembly do MIPS?

# Suporte a Chamada de Procedimentos

- As variáveis de parâmetro **g**, **h**, **i** e **j** correspondem aos registradores de argumento \$a0, \$a1, \$a2, \$a3
- Suponha que a variável **f** corresponde ao registrador \$s0
- A instrução de atribuição usa dois registradores temporários

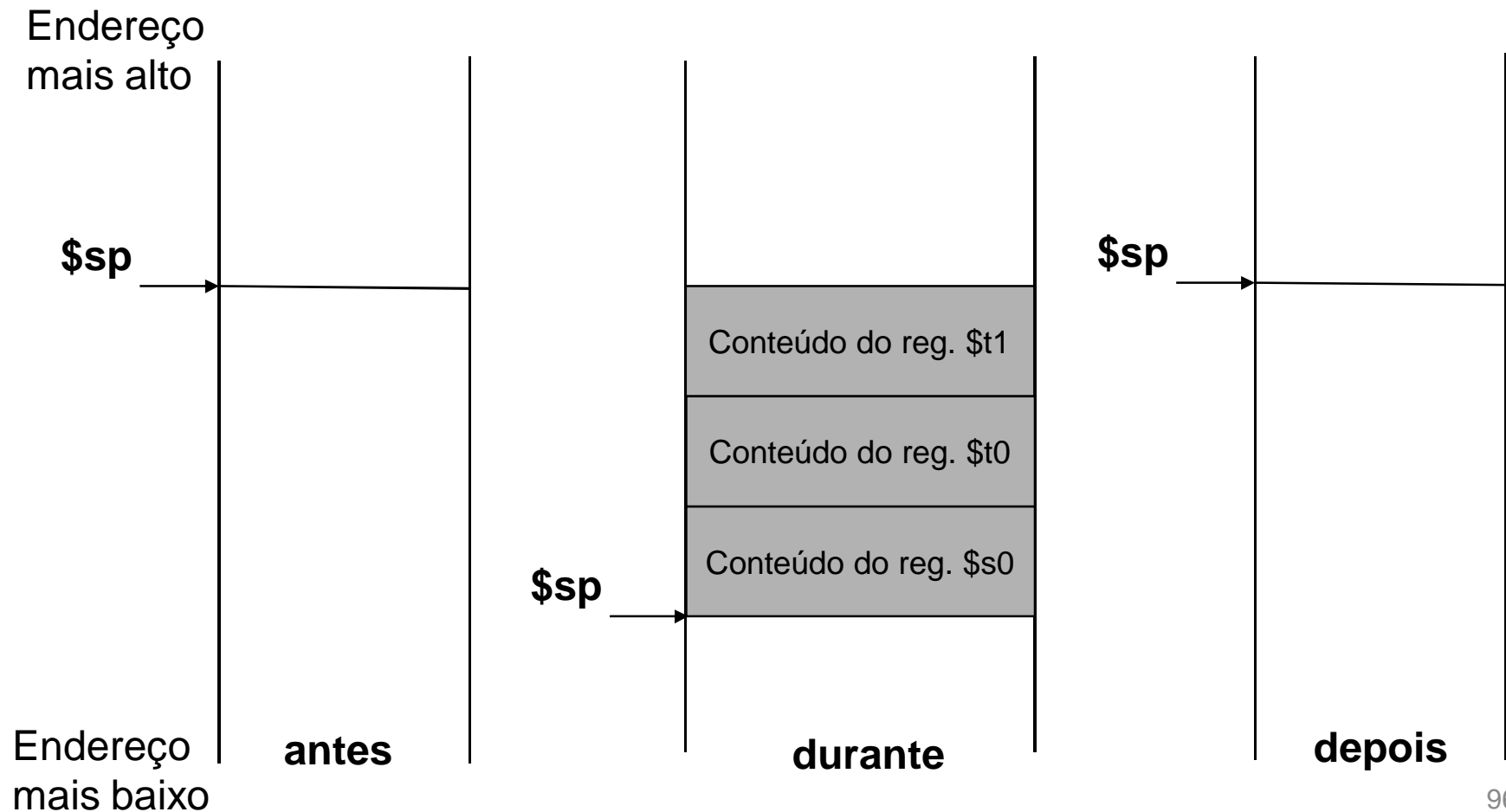
# Suporte a Chamada de Procedimentos

**exemplo\_folha:**

```
addi $sp, $sp, -12  
sw   $t1, 8($sp)  
sw   $t0, 4($sp)  
sw   $s0, 0($sp)  
add  $t0, $a0, $a1  
add  $t1, $a2, $a3  
sub  $s0, $t0, $t1  
add  $v0, $s0, $zero  
lw   $s0, 0($sp)  
lw   $t0, 4($sp)  
lw   $t1, 8($sp)  
add  $sp, $sp, 12  
jr   $ra
```

# Suporte a Chamada de Procedimentos

- Os valores do *stack pointer* e a pilha





# Suporte a Chamada de Procedimentos

- No exemplo anterior, dois registradores temporários são usados e seus valores antigos foram salvos e restaurados
- É necessário evitar salvar e restaurar um registrador cujo valor nunca é utilizado
  - Pode acontecer com registradores temporários

# Suporte a Chamada de Procedimentos

- O MIPS possui uma convenção:
  - \$t0-\$t9 são 10 registradores temporários que não precisam ser preservados
  - \$s0-\$s7 são 8 registradores de salvamento que precisam ser preservados. Se forem usados, o procedimento chamado os salva e restaura
- No exemplo anterior:
  - Os registradores \$t0 e \$t1 não precisavam ser preservados, mas somente o registrador \$s0

# Procedimentos Aninhados

- Os procedimentos aninhados são procedimentos que chamam outro procedimento
  - O procedimento chamador empilha os registradores de argumento (\$a0-\$a3) e os registradores temporários (\$t0-\$t9) que sejam necessários após a chamada
  - O procedimento chamado empilha o registrador do endereço de retorno \$ra e quaisquer registradores de salvamento (\$s0-\$s7) usados por ele

# Procedimentos Recursivos

- Considere o procedimento recursivo que calcula o fatorial:

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

- Qual é o código assembly do MIPS?

# Procedimentos Recursivos

- O parâmetro **n** corresponde ao registrador de argumento \$a0
- O programa compilado começa com o rótulo do procedimento
- Salva os registradores \$ra e \$a0 na pilha

**fatorial:**

```
addi $sp, $sp, -8  
sw   $ra, 4($sp)  
sw   $a0, 0($sp)
```

# Procedimentos Recursivos

- Na primeira vez que o procedimento **fatorial** é chamado, **sw** salva o endereço do programa que chamou **fatorial**
- As próximas duas instruções testam se **n** é menor do que 1, indo para L1 se  $n \geq 1$

```
slti    $t0, $a0, 1  
beq     $t0, $zero, L1
```

# Procedimentos Recursivos

- Se  $n < 1$ , **fatorial** retorna o valor 1
  - Coloca 1 no registrador de valor. Soma 1 a 0 e atribui essa soma em \$v0
  - Retira os dois valores salvos da pilha
  - Desvia para o endereço de retorno

```
addi $v0, $zero, 1
addi $sp, $sp, 8
jr    $ra
```

# Procedimentos Recursivos

- Os registradores `$a0` e `$ra` não precisaram ser restaurados, pois não mudam quando `n` é menor do que 1



# Procedimentos Recursivos

- Se  $n \geq 1$ 
  - O argumento **n** é decrementado
  - O procedimento **fatorial** é chamado novamente com o valor decrementado

```
L1: addi $a0, $a0, -1  
    jal  fatorial
```

# Procedimentos Recursivos

- A próxima instrução é onde **fatorial** retorna
- O endereço de retorno antigo e o argumento antigo são restaurados
- O *stack pointer* (\$sp) é reajustado

```
lw    $a0, 0($sp)
lw    $ra, 4($sp)
addi  $sp, $sp, 8
```

# Procedimentos Recursivos

- Em seguida, o registrador de valor \$v0 recebe o produto do argumento antigo \$a0 e o valor atual do registrador de valor, ou seja,  $\$v0 = n * \text{fatorial}(n-1)$
- Considere a existência da instrução de multiplicação (**mult**)

**mult \$v0, \$a0, \$v0**

# Procedimentos Recursivos

- Por fim, o procedimento **fatorial** salta novamente para o endereço de retorno  
**jr \$ra**

# Suporte a Chamada de Procedimentos

- A linguagem C possui duas classes de armazenamento:
  - Automática
  - Estática
- As variáveis automáticas são locais a um procedimento e são descartadas quando o procedimento termina

# Suporte a Chamada de Procedimentos

- As variáveis estáticas permanecem durante entradas e saídas de procedimentos
  - Variáveis C declaradas fora de todos os procedimentos
  - Variáveis declaradas por meio da palavra reservada **static**
- Para simplificar o acesso aos dados estáticos, o MIPS reserva um registrador denominado **ponteiro global** ou **\$gp**

# Suporte a Chamada de Procedimentos

- A pilha também é utilizada para armazenar variáveis que são locais ao procedimento, tais como *arrays* e estruturas locais
  - Se as variáveis forem em maior número do que a quantidade de registradores disponíveis
- O segmento da pilha que contém os registradores com os conteúdos salvos do procedimento e suas variáveis locais é denominado de **registro de ativação**

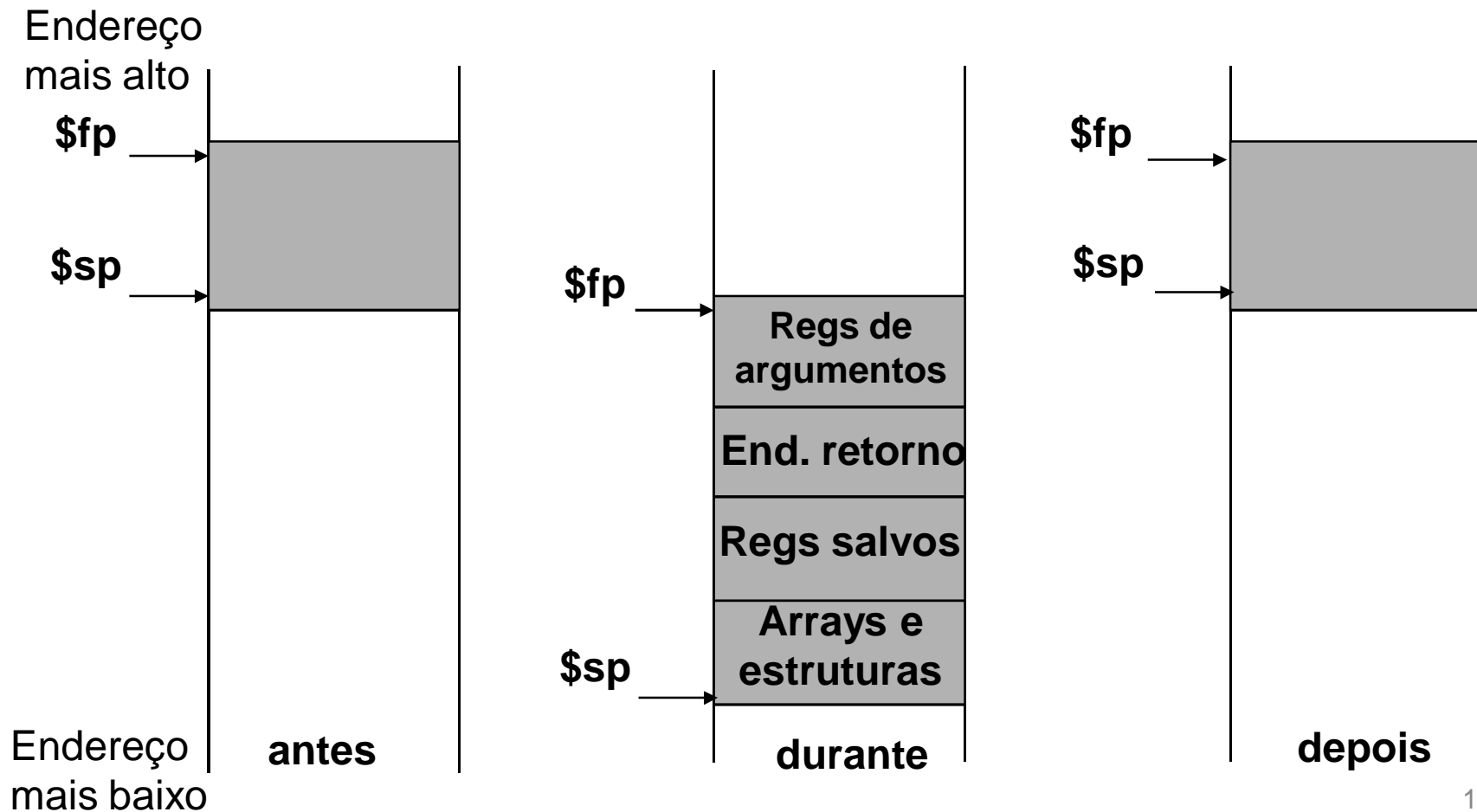
# Suporte a Chamada de Procedimentos

- Alguns softwares do processador MIPS usam um registrador denominado *frame pointer* (**\$fp**) para apontar para o registro de ativação de um procedimento
  - Atua como um registrador base estável para as variáveis locais a um procedimento referenciarem a memória



# Suporte a Chamada de Procedimentos

- Pilha



# Convenção de Registradores MIPS

| Nome             | Número do Registrador | Utilização                                | Preservado?   |
|------------------|-----------------------|---|---------------|
| <i>\$zero</i>    | 0                     | Valor da constante 0                      | Não se aplica |
| <i>\$at</i>      | 1                     | Reservado para o montador                 | Não se aplica |
| <i>\$v0-\$v1</i> | 2-3                   | Armazenar resultados e avaliar expressões | Não           |
| <i>\$a0-\$a3</i> | 4-7                   | Argumentos                                | Sim           |
| <i>\$t0-\$t7</i> | 8-15                  | Temporários                               | Não           |
| <i>\$s0-\$s7</i> | 16-23                 | Salvos                                    | Sim           |
| <i>\$t8-\$t9</i> | 24-25                 | Temporários                               | Não           |
| <i>\$k0-\$k1</i> | 26-27                 | Reservado para o sistema operacional      | Não se aplica |
| <i>\$gp</i>      | 28                    | <i>Global pointer</i>                     | Sim           |
| <i>\$sp</i>      | 29                    | <i>Stack pointer</i>                      | Sim           |
| <i>\$fp</i>      | 30                    | <i>Frame pointer</i>                      | Sim           |
| <i>\$ra</i>      | 31                    | Endereço de retorno de procedimento       | Sim           |

# Operandos Imediatos de 32 bits

- O conjunto de instruções MIPS inclui a instrução *load upper immediate* (**lui**)
  - Atribui os 16 bits mais altos de uma constante a um registrador
  - Permitindo outra instrução atribuir os 16 bits mais baixos da constante

# Operandos Imediatos de 32 bits

- A instrução **lui** transfere o valor do campo de constante imediata de 16 bits para os 16 bits mais à esquerda do registrador e preenche os 16 bits de menor ordem (direita) com **zeros**

# Operandos Imediatos de 32 bits

- Versão em linguagem de máquina da instrução **lui \$t0, 255**

|        |       |       |                     |
|--------|-------|-------|---------------------|
| 001111 | 00000 | 01000 | 0000 0000 1111 1111 |
| 31-26  | 25-21 | 20-16 | 15-0                |

- Conteúdo de **\$t0** após executar **lui \$t0, 255**

|                     |                     |
|---------------------|---------------------|
| 0000 0000 1111 1111 | 0000 0000 0000 0000 |
| 31-16               | 15-0                |

# Operandos Imediatos de 32 bits

- Qual é o código assembly do MIPS para carregar esta constante de 32 bits no registrador \$s0?

0000 0000 0011 1101 0000 1001 0000 0000

# Operandos Imediatos de 32 bits

- Carregar os 16 bits mais altos

**lui \$s0, 61**                      #  $61_{10} = 0000\ 0000\ 0011\ 1101_2$

- Valor do registrador \$s0

0000 0000 0011 1101 0000 0000 0000 0000

- Acrescentar os 16 bits inferiores

**ori \$s0, \$s0, 2034**    #  $2034_{10} = 0000\ 1001\ 0000\ 0000_2$

- Valor final do registrador \$s0

0000 0000 0011 1101 0000 1001 0000 0000

# Operandos Imediatos de 32 bits

- A instrução **ori** (or imediato) carrega **zeros** nos 16 bits superiores e, portanto, é usada pelo montador em conjunto com a instrução **lui** para criar constantes de 32 bits



# Endereçamento em desvios condicionais e *jumps*

- A instrução de desvio condicional precisa especificar dois operandos além do endereço de desvio
- Apenas 16 bits para o endereço de desvio
- Exemplo:

**bne \$s0, \$s1, Exit**    # Vai para Exit, se \$s0 ≠ \$s1

|        |        |        |         |
|--------|--------|--------|---------|
| 5      | 16     | 17     | Exit    |
| 6 bits | 5 bits | 5 bits | 16 bits |

# Endereçamento em desvios condicionais e *jumps*

- Se os endereços dos programas tivessem de caber nesse campo de 16 bits, nenhum programa poderia ser maior do que  $2^{16}$
- Os desvios condicionais são encontrados em *loops* e em instruções *if*, de modo que costumam desviar para uma instrução próxima

# Endereçamento em desvios condicionais e *jumps*

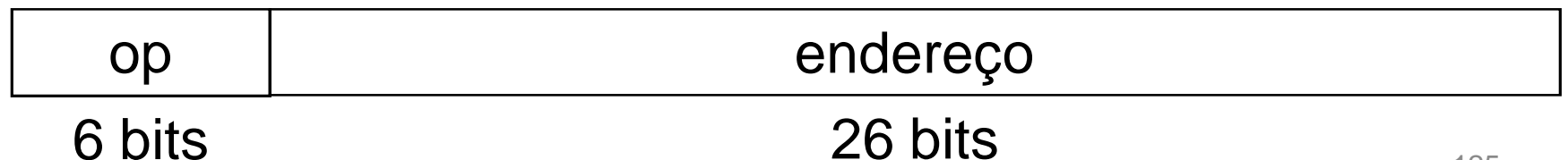
- O contador de programa (PC) contém o endereço da instrução atual e pode ser somado ao endereço de desvio
- Essa forma de endereçamento de desvio é denominada **endereçamento relativo ao PC**
  - Na realidade, o endereço MIPS é relativo ao endereço da instrução seguinte (PC+4)

# Endereçamento em desvios condicionais e *jumps*

- O MIPS aumenta a distância do desvio fazendo com que o endereço relativo ao PC se refira ao número de *words* até a próxima instrução, em vez do número de bytes
  - O campo de 16 bits pode se desviar para uma distância 4 vezes maior
  - Desvios dentro de  $\pm 2^{15}$  *words* da instrução atual
  - Quase todos os *loops* e instruções *if* possuem desvios muito menores

# Endereçamento em desvios condicionais e *jumps*

- As instruções de *jump-and-link* chamam procedimentos que não têm motivo para estarem próximos à chamada
- A arquitetura MIPS oferece endereços longos para chamadas de procedimento
- Formato do tipo J usado para instruções de *jump* e *jump-and-link*



# Endereçamento em desvios condicionais e *jumps*

- O campo de 26 bits nas instruções de *jump* é um endereço de *word*
  - Representa um endereço de 28 bits
- O PC utiliza 32 bits. A instrução de *jump* do MIPS substitui apenas os 28 bits menos significativos do PC, deixando os 4 bits mais significativos inalterados

# Desvio em Linguagem de Máquina

- Exemplo em um *loop while*

**while (save[i] == k)**

**i += 1; // i em \$s3, k em \$s5 e save em \$s6**

|       |     |       |    |   |   |    |
|-------|-----|-------|----|---|---|----|
| 80000 | 0   | 0     | 19 | 9 | 2 | 0  |
| 80004 | 0   | 9     | 22 | 9 | 0 | 32 |
| 80008 | 35  | 9     | 8  | 0 |   |    |
| 80012 | 5   | 8     | 21 | 2 |   |    |
| 80016 | 8   | 19    | 19 | 1 |   |    |
| 80020 | 2   | 20000 |    |   |   |    |
| 80024 | ... |       |    |   |   |    |

**Loop:**    **sll**        **\$t1, \$s3, 2**  
             **add**        **\$t1, \$t1, \$s6**  
             **lw**         **\$t0, 0(\$t1)**  
             **bne**        **\$t0, \$s5, Exit**  
             **addi**       **\$s3, \$s3, 1**  
             **j**            **Loop**

**Exit:**

# Desviando para um Lugar mais Distante

- Considere um desvio onde o registrador \$s0 é igual ao registrador \$s1

**beq \$s0, \$s1, L1**

- Pode-se substituir esta instrução por um par de instruções de forma a oferecer uma distância de desvio muito maior

**bne \$s0, \$s1, L2**

**j L1**

**L2:**



# Resumo dos modos de endereçamento no MIPS

- Endereçamento em registrador:
  - Operando é um registrador
- Endereçamento de base ou deslocamento:
  - Operando em memória, cujo endereço é a soma de um registrador com uma constante na instrução
- Endereçamento imediato:
  - Operando é uma constante dentro da instrução

# Resumo dos modos de endereçamento no MIPS

- Endereçamento relativo ao PC:
  - Endereçamento é a soma do PC com uma constante na instrução
- Endereçamento pseudodireto:
  - Endereço de *jump* são os 26 bits da instrução concatenados com os bits mais altos do PC

# Tradução de um Programa

- Em geral, os programas são desenvolvidos em uma linguagem de alto nível como Fortran, Pascal ou C
- O compilador traduz o programa de alto nível em uma sequência de **instruções de processador**
  - Programa em **linguagem de montagem** ou **linguagem *assembly***
  - Representa textualmente as instruções

# Tradução de um Programa

- O programa em linguagem de montagem é convertido para um programa em **código objeto** pelo **montador** (*assembler*)
- O montador traduz diretamente uma instrução da forma textual para a forma de código binário (**linguagem de máquina**)

# Tradução de um Programa

- Sob a forma binária que a instrução é carregada na memória e interpretada pelo processador
- Programas complexos são normalmente estruturados em módulos. Cada módulo é compilado separadamente e submetido ao montador, gerando diversos módulos em código objeto

# Tradução de um Programa

- Estes módulos e as rotinas de biblioteca (em linguagem de máquina) são reunidos pelo **ligador** (*linker*), resultando no programa executável
- O programa executável é carregado na memória pelo **carregador** (*loader*)

# Bibliografia

- [1] Maria Clicia S de Castro. "Capítulo 4: O Conjunto de Instruções do Processador.", Apostila disponível em <http://www.ime.uerj.br/professores/Mariaclicia/Oc1/oc1.htm>
- [2] D. A. Patterson e J. L. Hennessy, "Organização e projeto de computadores: a interface hardware/software", Ed. Campus, 2005.

**Parte do material cedido pela Profa. Clicia (UERJ)**