Arquitetura de Computadores
ACH2055
Aula 03 – Assembly MIPS (Parte I)

Norton Trevisan Roman
(norton@usp.br)

23 de agosto de 2019

A Arquitetura MIPS

Definição

- *Microprocessor Without Interlocked Pipeline Stages*
  - Desenvolvida pela MIPS Technologies (1984)

A Arquitetura MIPS

Definição

- *Microprocessor Without Interlocked Pipeline Stages*
  - Desenvolvida pela MIPS Technologies (1984)
- Trata-se de uma arquitetura de conjunto de instruções
  - *Instruction Set Architecture – ISA*
  - No caso do MIPS, RISC (veremos mais adiante)

A Arquitetura MIPS

Definição

- *Microprocessor Without Interlocked Pipeline Stages*
  - Desenvolvida pela MIPS Technologies (1984)
- Trata-se de uma arquitetura de conjunto de instruções
  - *Instruction Set Architecture – ISA*
  - No caso do MIPS, RISC (veremos mais adiante)
- Presentes em diversos sistemas
  - Videogames, roteadores etc

A Arquitetura MIPS

Vantagens

- Instruções têm tamanho fixo
  - 32 e 64 bits
  - X86 não tem tamanho fixo
  - São mais fáceis de se manipular, especialmente por uma pipeline (veremos mais adiante)

A Arquitetura MIPS

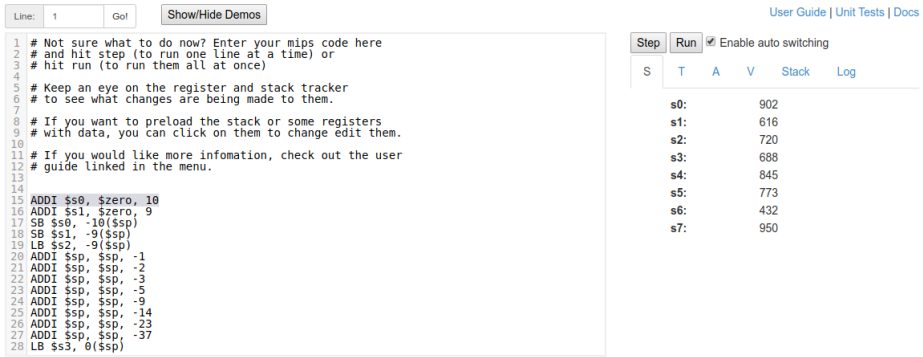
Vantagens

- Instruções têm tamanho fixo
  - 32 e 64 bits
  - X86 não tem tamanho fixo
  - São mais fáceis de se manipular, especialmente por uma pipeline (veremos mais adiante)
- As instruções seguem um formato geral fixo
  - OPERAÇÃO <lista de operandos>
  - Ex: ADD $s1, $s2, $s3

Programando em MIPS

Simuladores

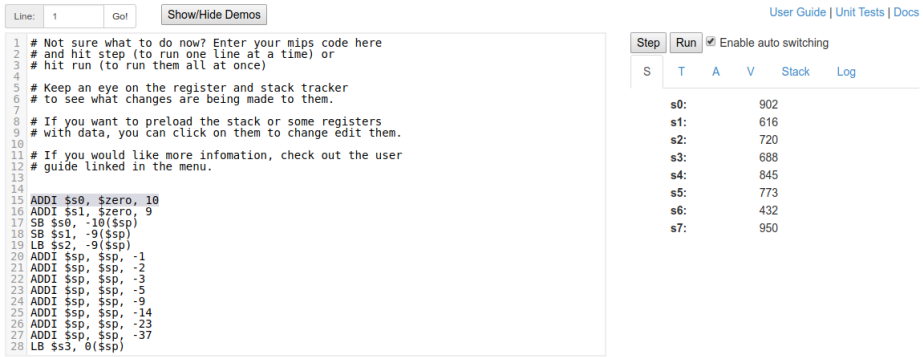
- Wemips (online)
  - `https://rivoire.cs.sonoma.edu/cs351/wemips/`

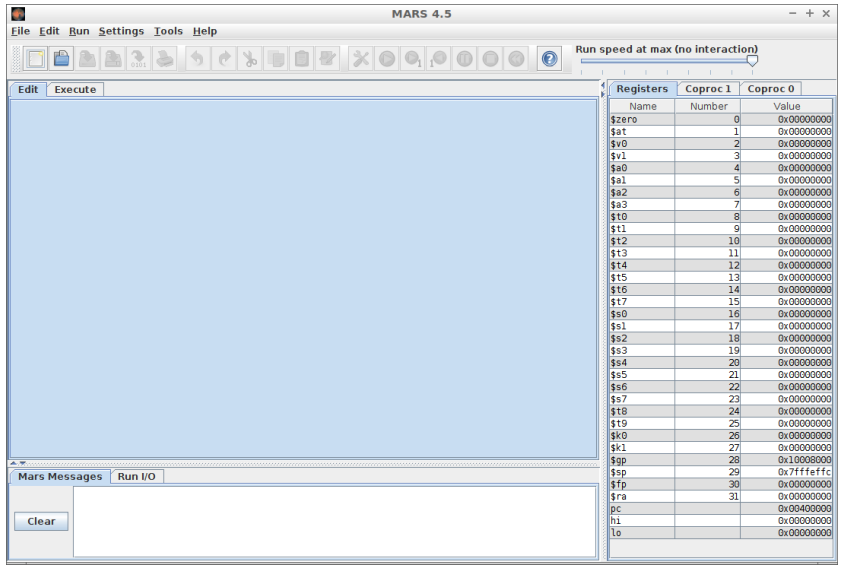


Programando em MIPS

Simuladores

- Wemips (online)
  - `https://rivoire.cs.sonoma.edu/cs351/wemips/`
- Mars (.jar)
  - `http://courses.missouristate.edu/kenvollmar/mars/`





Programando em MIPS

Registradores

- MIPS possui 32 registradores de uso geral
  - Numerados de 0 a 31
  - Em geral referenciados pelo seu número ($0 a $31) ou nome (ex: $t1)

Programando em MIPS

Registradores

- MIPS possui 32 registradores de uso geral
  - Numerados de 0 a 31
  - Em geral referenciados pelo seu número ($0 a $31) ou nome (ex: $t1)
- Embora possam ser usados livremente, há uma convenção em seu uso
  - Seguida por programadores e compiladores
  - Seguir essa convenção permite que um código possa funcionar com outros em MIPS

Programando em MIPS

Registradores – Convenção

Nome	*Número*	*Uso*
`zero`	0	Constante zero
`at`	1	Reservado para o montador
`v0 – v1`	2 – 3	Registradores de resultado
`a0 – a3`	4 – 7	Registradores de argumentos
`t0 – t9`	8 – 15, 24 – 25	Registradores temporários
`s0 – s7`	16 – 23	Registradores salvos
`k0 – k1`	26 – 27	Registradores do Kernel
`gp`	28	Global Pointer
`sp`	29	Stack Pointer
`fp`	30	Frame Pointer
`ra`	31	Return Address

Programando em MIPS

Registradores – Convenção

Nome	*Número*	*Uso*
`zero`	0	Constante zero
`at`	1	Reservado para o montador
`v0 – v1`	2 – 3	Registradores de resultado
`a0 – a3`	4 – 7	Registradores de argumentos
`t0 – t9`	8 – 15, 24 – 25	Registradores temporários
`s0 – s7`	16 – 23	Registradores salvos
`k0 – k1`	26 – 27	Registradores do Kernel
`gp`	28	Global Pointer
`sp`	29	Stack Pointer
`fp`	30	Frame Pointer
`ra`	31	Return Address

`zero` sempre contém o valor 0 (*hard-wired*)

Programando em MIPS

Registradores – Convenção

Nome	*Número*	*Uso*
`zero`	0	Constante zero
`at`	1	Reservado para o montador
`v0 – v1`	2 – 3	Registradores de resultado
`a0 – a3`	4 – 7	Registradores de argumentos
`t0 – t9`	8 – 15, 24 – 25	Registradores temporários
`s0 – s7`	16 – 23	Registradores salvos
`k0 – k1`	26 – 27	Registradores do Kernel
`gp`	28	Global Pointer
`sp`	29	Stack Pointer
`fp`	30	Frame Pointer
`ra`	31	Return Address

Iniciados em `v` guardam a avaliação de uma expressão ou os resultados de uma função

Programando em MIPS

Registradores – Convenção

Nome	*Número*	*Uso*
`zero`	0	Constante zero
`at`	1	Reservado para o montador
`v0 – v1`	2 – 3	Registradores de resultado
`a0 – a3`	4 – 7	Registradores de argumentos
`t0 – t9`	8 – 15, 24 – 25	Registradores temporários
`s0 – s7`	16 – 23	Registradores salvos
`k0 – k1`	26 – 27	Registradores do Kernel
`gp`	28	Global Pointer
`sp`	29	Stack Pointer
`fp`	30	Frame Pointer
`ra`	31	Return Address

Iniciados em a guardam argumentos para as rotinas

Programando em MIPS

Registradores – Convenção

Nome	*Número*	*Uso*
`zero`	0	Constante zero
`at`	1	Reservado para o montador
`v0` – `v1`	2 – 3	Registradores de resultado
`a0` – `a3`	4 – 7	Registradores de argumentos
`t0` – `t9`	8 – 15, 24 – 25	Registradores temporários
`s0` – `s7`	16 – 23	Registradores salvos
`k0` – `k1`	26 – 27	Registradores do Kernel
`gp`	28	Global Pointer
`sp`	29	Stack Pointer
`fp`	30	Frame Pointer
`ra`	31	Return Address

Iniciados em `t` guardam valores que não precisam ser preservados entre chamadas

Programando em MIPS

Registradores – Convenção

Nome	*Número*	*Uso*
`zero`	0	Constante zero
`at`	1	Reservado para o montador
`v0 – v1`	2 – 3	Registradores de resultado
`a0 – a3`	4 – 7	Registradores de argumentos
`t0 – t9`	8 – 15, 24 – 25	Registradores temporários
`s0 – s7`	16 – 23	Registradores salvos
`k0 – k1`	26 – 27	Registradores do Kernel
`gp`	28	Global Pointer
`sp`	29	Stack Pointer
`fp`	30	Frame Pointer
`ra`	31	Return Address

Iniciados em s guardam valores que precisam ser preservados entre chamadas

Programando em MIPS

Registradores – Convenção

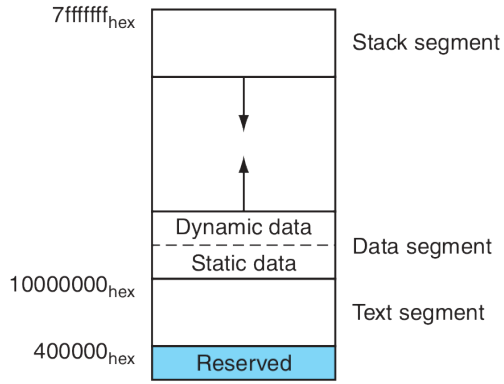
Nome	*Número*	*Uso*
`zero`	0	Constante zero
`at`	1	Reservado para o montador
`v0 – v1`	2 – 3	Registradores de resultado
`a0 – a3`	4 – 7	Registradores de argumentos
`t0 – t9`	8 – 15, 24 – 25	Registradores temporários
`s0 – s7`	16 – 23	Registradores salvos
`k0 – k1`	26 – 27	Registradores do Kernel
`gp`	28	Global Pointer
`sp`	29	Stack Pointer
`fp`	30	Frame Pointer
`ra`	31	Return Address

Iniciados em `k` são reservados para o kernel do S.O.

Programando em MIPS

Registradores – Convenção

Nome	*Número*	*Uso*
`zero`	0	Constante zero
`at`	1	Reservado para o montador
`v0 – v1`	2 – 3	Registradores de resultado
`a0 – a3`	4 – 7	Registradores de argumentos
`t0 – t9`	8 – 15, 24 – 25	Registradores temporários
`s0 – s7`	16 – 23	Registradores salvos
`k0 – k1`	26 – 27	Registradores do Kernel
`gp`	28	Global Pointer
`sp`	29	Stack Pointer
`fp`	30	Frame Pointer
`ra`	31	Return Address



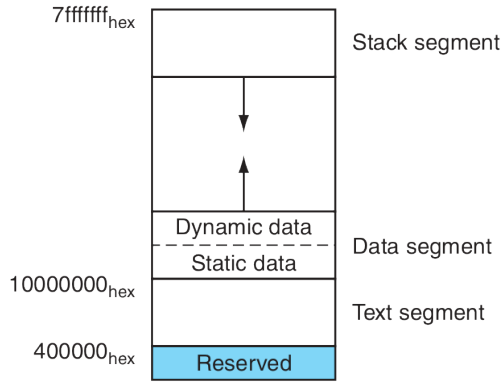
Fonte: [1]

gp aponta para o endereço do fim dos primeiros 32K no segmento de dados estáticos

Programando em MIPS

Registradores – Convenção

Nome	*Número*	*Uso*
`zero`	0	Constante zero
`at`	1	Reservado para o montador
`v0 – v1`	2 – 3	Registradores de resultado
`a0 – a3`	4 – 7	Registradores de argumentos
`t0 – t9`	8 – 15, 24 – 25	Registradores temporários
`s0 – s7`	16 – 23	Registradores salvos
`k0 – k1`	26 – 27	Registradores do Kernel
`gp`	28	Global Pointer
`sp`	29	Stack Pointer
`fp`	30	Frame Pointer
`ra`	31	Return Address



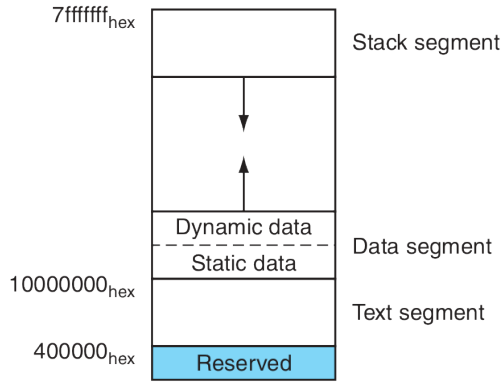
Fonte: [1]

Permitindo assim acesso rápido aos primeiros 64K desse segmento

Programando em MIPS

Registradores – Convenção

Nome	*Número*	*Uso*
`zero`	0	Constante zero
`at`	1	Reservado para o montador
`v0 – v1`	2 – 3	Registradores de resultado
`a0 – a3`	4 – 7	Registradores de argumentos
`t0 – t9`	8 – 15, 24 – 25	Registradores temporários
`s0 – s7`	16 – 23	Registradores salvos
`k0 – k1`	26 – 27	Registradores do Kernel
`gp`	28	Global Pointer
`sp`	29	Stack Pointer
`fp`	30	Frame Pointer
`ra`	31	Return Address



Fonte: [1]

sp guarda o endereço do último local da pilha

Programando em MIPS

Registradores – Convenção

Nome	*Número*	*Uso*
`zero`	0	Constante zero
`at`	1	Reservado para o montador
`v0` – `v1`	2 – 3	Registradores de resultado
`a0` – `a3`	4 – 7	Registradores de argumentos
`t0` – `t9`	8 – 15, 24 – 25	Registradores temporários
`s0` – `s7`	16 – 23	Registradores salvos
`k0` – `k1`	26 – 27	Registradores do Kernel
`gp`	28	Global Pointer
`sp`	29	Stack Pointer
`fp`	30	Frame Pointer
`ra`	31	Return Address

`fp` guarda o endereço da primeira palavra da moldura de pilha do procedimento

Programando em MIPS

Registradores – Convenção

Nome	*Número*	*Uso*
`zero`	0	Constante zero
`at`	1	Reservado para o montador
`v0 – v1`	2 – 3	Registradores de resultado
`a0 – a3`	4 – 7	Registradores de argumentos
`t0 – t9`	8 – 15, 24 – 25	Registradores temporários
`s0 – s7`	16 – 23	Registradores salvos
`k0 – k1`	26 – 27	Registradores do Kernel
`gp`	28	Global Pointer
`sp`	29	Stack Pointer
`fp`	30	Frame Pointer
`ra`	31	Return Address

`ra` guarda o endereço de retorno de uma chamada de procedimento

Programando em MIPS

Instruções Aritméticas

- Cada instrução executa uma única operação, possuindo 3 campos
  - 2 operandos e um local para armazenamento do resultado
  - Cada um correspondendo a um dos 32 registradores disponíveis

Programando em MIPS

Instruções Aritméticas

- Cada instrução executa uma única operação, possuindo 3 campos
  - 2 operandos e um local para armazenamento do resultado
  - Cada um correspondendo a um dos 32 registradores disponíveis
- Formato:
  - operação \$resultado, \$operando$_1$, \$operando$_2$
  - Ex: `add $t0, $s2, $s3`

Programando em MIPS

Principais Instruções Aritméticas

- Adição
  - `add $t0, $s2, $s3`
- Subtração
  - `sub $t0, $s2, $s3`
- Multiplicação
  - `mult $t0, $s2, $s3`
- Divisão
  - `div $t0, $s2, $s3`

Programando em MIPS

Principais Instruções Aritméticas

- Adição
  - `add $t0, $s2, $s3`
- Subtração
  - `sub $t0, $s2, $s3`
- Multiplicação
  - `mult $t0, $s2, $s3`
- Divisão
  - `div $t0, $s2, $s3`
- Todas com versões para inteiros sem sinal
  - `addu`, `subu`, `multu`, `divu`

Programando em MIPS

Principais Instruções Aritméticas

- Adição
  - `add $t0, $s2, $s3`
- Subtração
  - `sub $t0, $s2, $s3`
- Multiplicação
  - `mult $t0, $s2, $s3`
- Divisão
  - `div $t0, $s2, $s3`
- Todas com versões para inteiros sem sinal
  - `addu`, `subu`, `multu`, `divu`
- Para ponto flutuante etc
  - `add.d`, `sub.d`, `mult.d`, `div.d`

Programando em MIPS

Constantes (valores imediatos)

- Essas instruções, contudo, trabalham com registradores pré-carregados
  - Como carregar esses registradores?

Programando em MIPS

Constantes (valores imediatos)

- Essas instruções, contudo, trabalham com registradores pré-carregados
  - Como carregar esses registradores?
- Com operadores imediatos
  - `li $s2, 10` (*load immediate* – a constante 10 é armazenada no registrador `$s2`)

Programando em MIPS

Constantes (valores imediatos)

- Essas instruções, contudo, trabalham com registradores pré-carregados
  - Como carregar esses registradores?
- Com operadores imediatos
  - `li $s2, 10` (*load immediate* – a constante 10 é armazenada no registrador `$s2`)
- Instruções aritméticas também têm versões assim:
  - Ex: `addi $t0, $s2, 4`
  - Presentes apenas para soma e subtração

Programando em MIPS

Constantes (valores imediatos)

- De fato, a instrução `li` não existe
  - Trata-se de uma pseudo-instrução

Programando em MIPS

Constantes (valores imediatos)

- De fato, a instrução `li` não existe
  - Trata-se de uma pseudo-instrução
- Em geral traduzida pelo montador como
  - `addiu destino, 0, valor`
  - `addiu` – *add immediate unsigned*
  - Ex: `li $s4, 10` ⇒ `addiu $s4, $zero, 10`

Programando em MIPS

Estrutura de um Programa MARS

- Dividido em 2 partes: declaração e código

Programando em MIPS

Estrutura de um Programa MARS

- Dividido em 2 partes: declaração e código
- Declaração
  - Usa a diretiva `.data`
    - Itens subsequentes serão armazenados no segmento de dados, no próximo endereço disponível
    - Se for fornecido um endereço (`.data end` $\rightarrow$ opcional), serão armazenados a partir desse endereço

Programando em MIPS

Estrutura de um Programa MARS

- Dividido em 2 partes: declaração e código
- Declaração
  - Usa a diretiva `.data`
    - Itens subsequentes serão armazenados no segmento de dados, no próximo endereço disponível
    - Se for fornecido um endereço (`.data end` → opcional), serão armazenados a partir desse endereço
  - Formato dos itens:
    - nome: .tipo_de_armazenagem valor
    - Ex: `var1:  .word 10`

Programando em MIPS

Estrutura de um Programa MARS

- Dividido em 2 partes: declaração e código
- Declaração
  - Usa a diretiva `.data`
    - Itens subsequentes serão armazenados no segmento de dados, no próximo endereço disponível
    - Se for fornecido um endereço (`.data end` → opcional), serão armazenados a partir desse endereço
  - Formato dos itens:
    - nome: .tipo_de_armazenagem valor
    - Ex: `var1:  .word 10`

Palavras seguidas por ':' (ex: `var1`) são rótulos, marcando o endereço de memória a seguir (onde 10 está na RAM)

Programando em MIPS

Estrutura de um Programa

- Código
  - Usa a diretiva `.text`
    - Itens subsequentes serão armazenados no segmento de texto
    - Se for fornecido um endereço (`.text end` $\rightarrow$ opcional), serão armazenados a partir desse endereço
  - Parte do programa em que as instruções a serem executadas são escritas

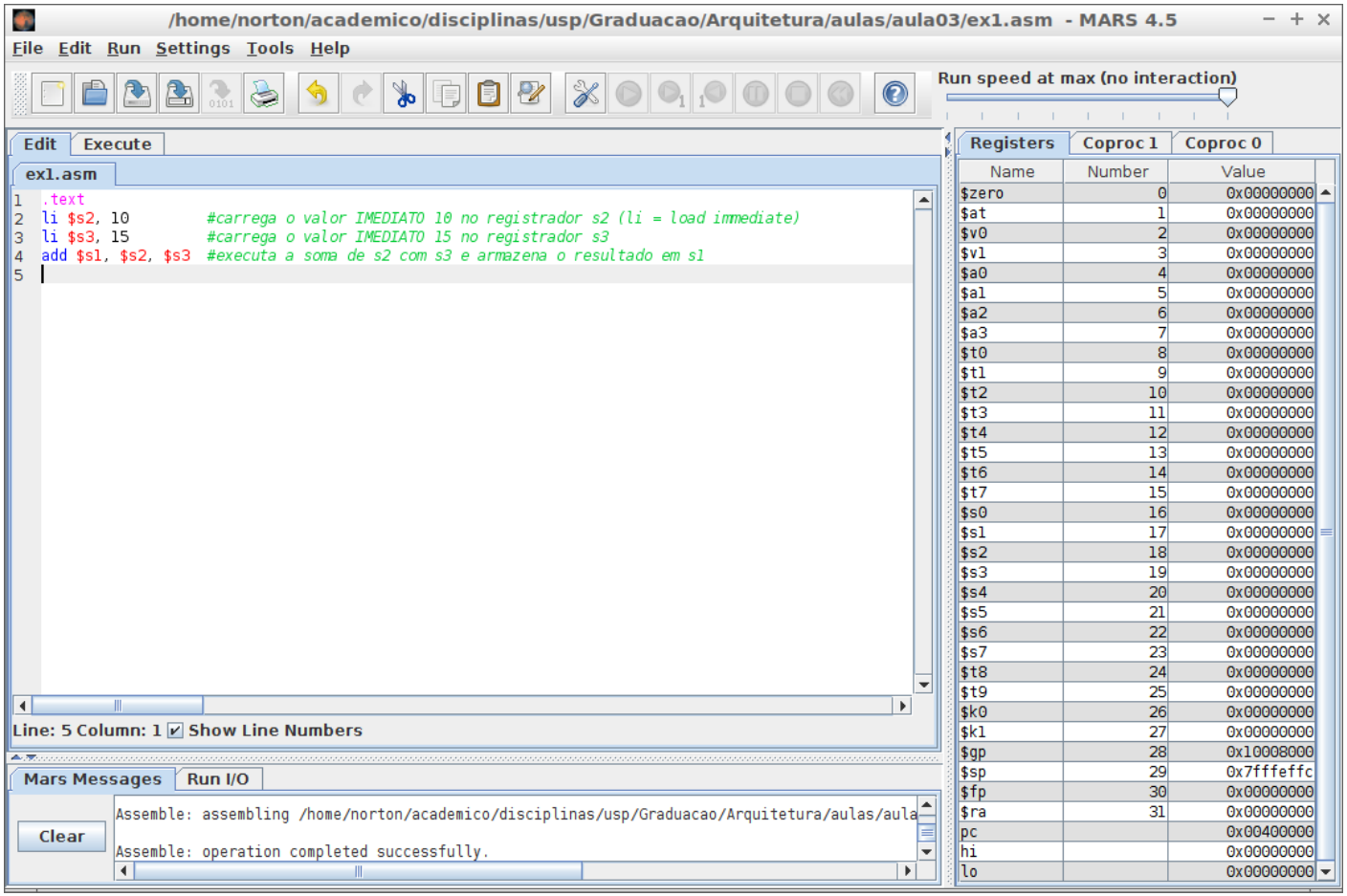
Programando em MIPS

Estrutura de um Programa

- Código
  - Usa a diretiva `.text`
    - Itens subsequentes serão armazenados no segmento de texto
    - Se for fornecido um endereço (`.text end` → opcional), serão armazenados a partir desse endereço
  - Parte do programa em que as instruções a serem executadas são escritas
- De início, focaremos apenas nessa parte

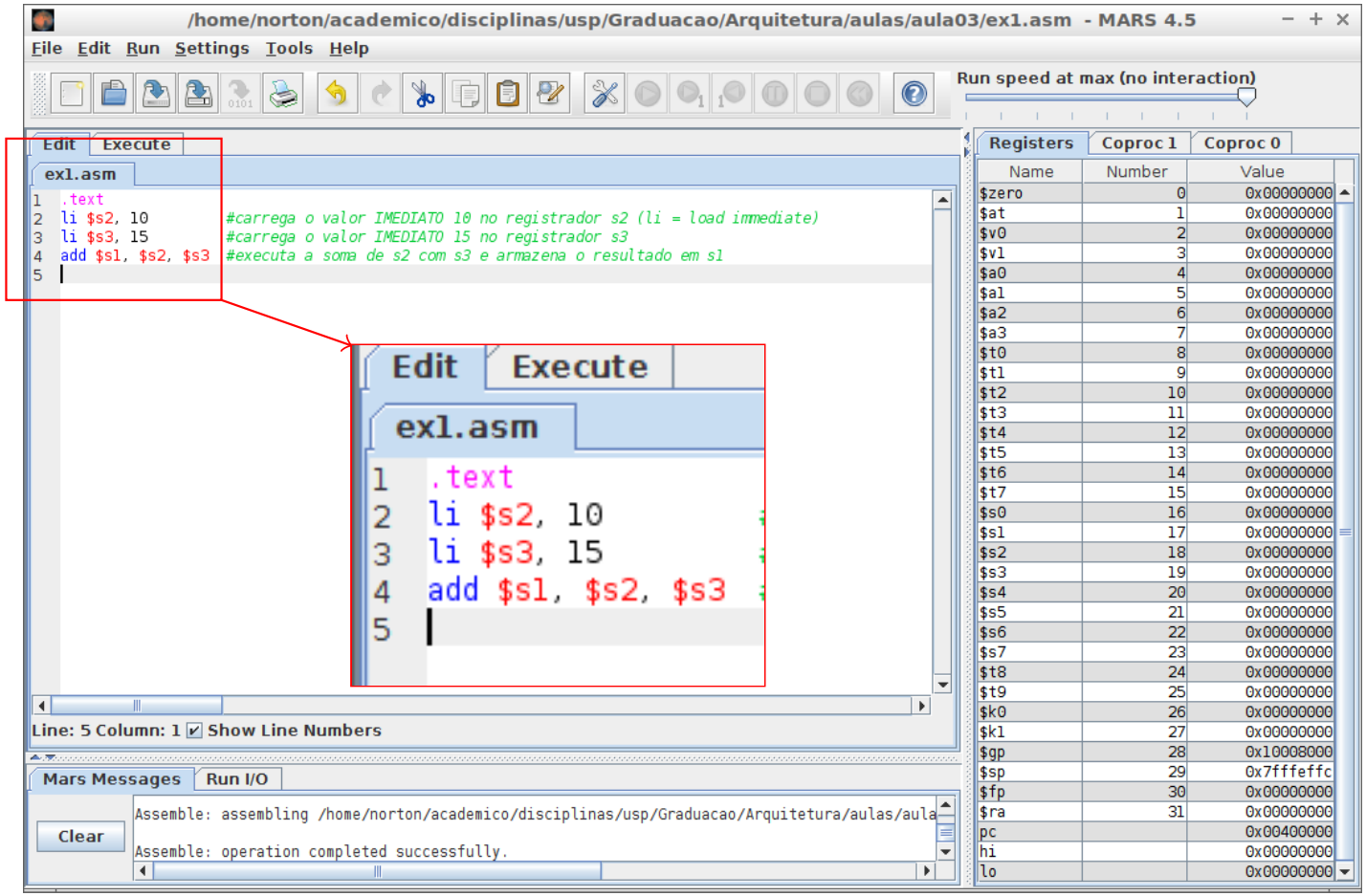
Programando em MIPS

Um Primeiro Programa



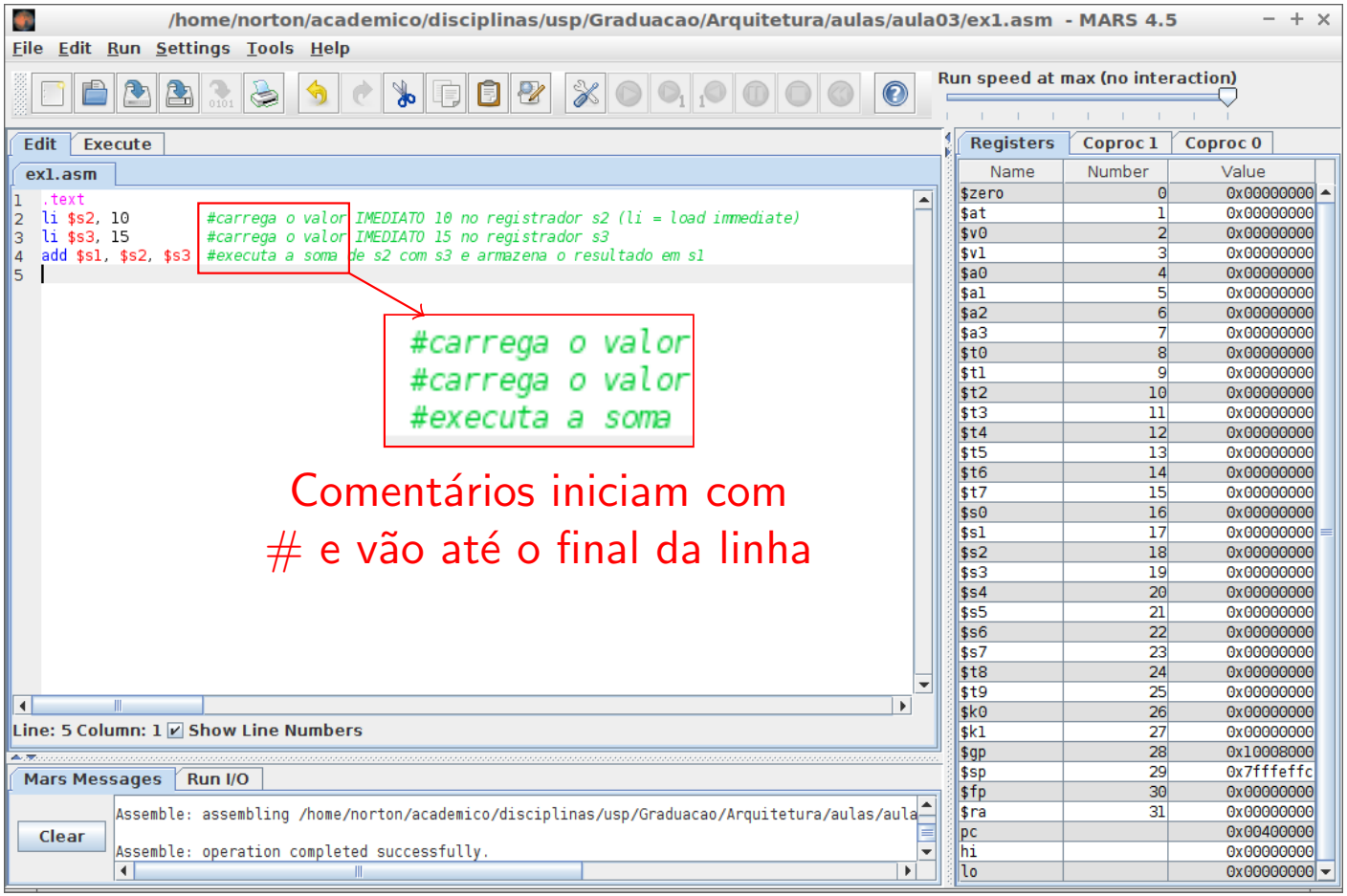
Programando em MIPS

Um Primeiro Programa



Programando em MIPS

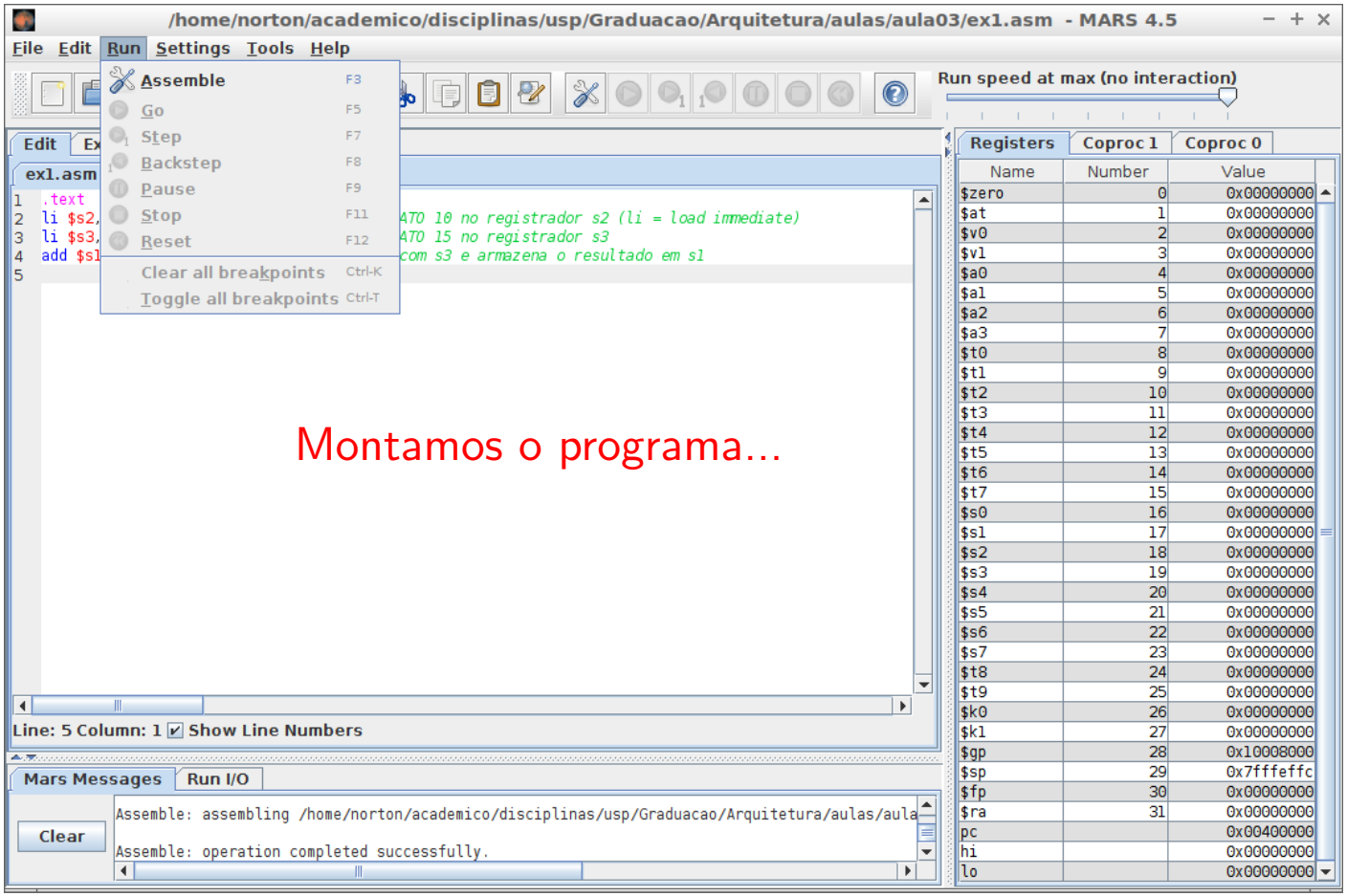
Um Primeiro Programa



Comentários iniciam com
e vão até o final da linha

Programando em MIPS

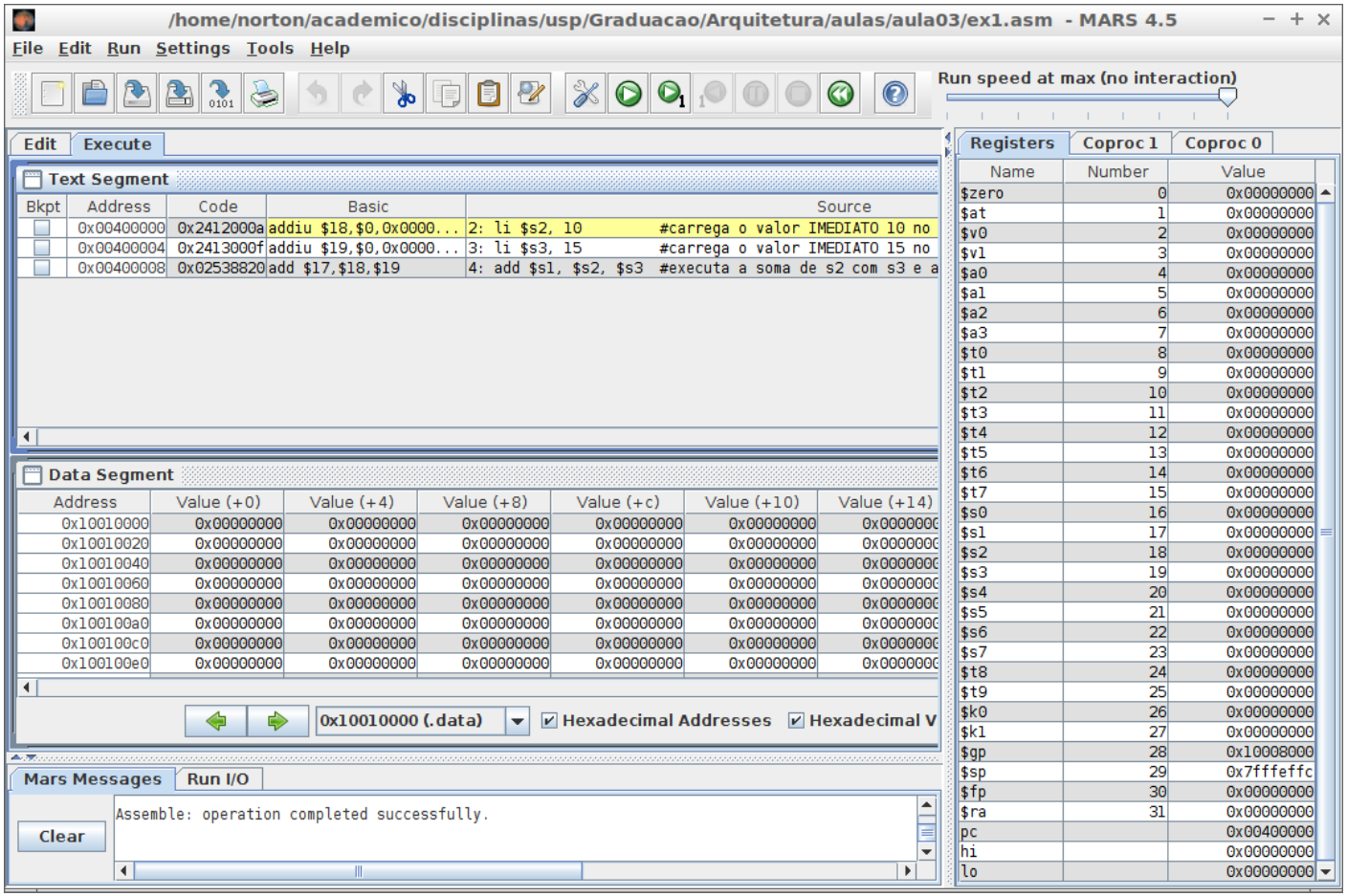
Um Primeiro Programa



Montamos o programa...

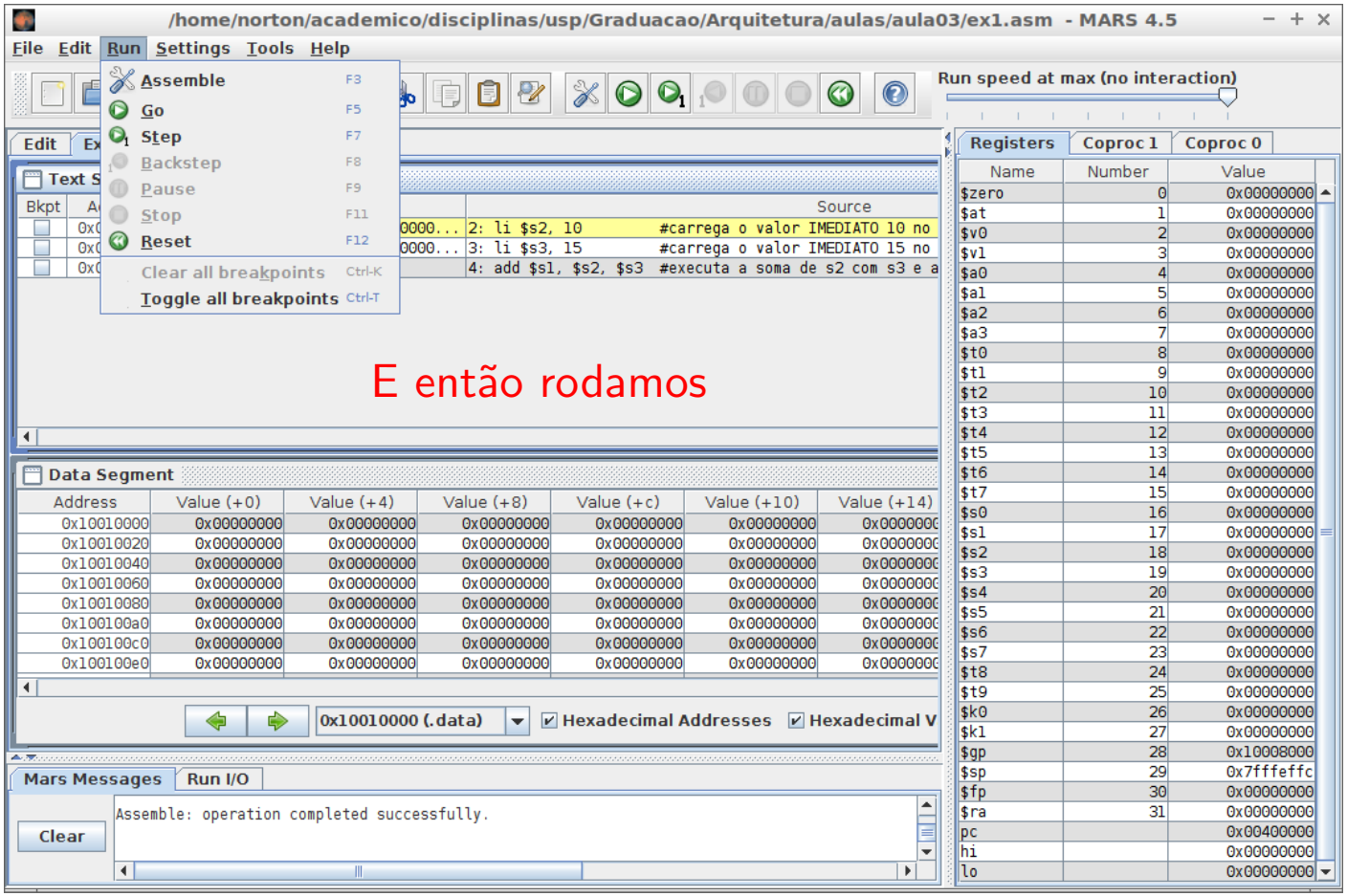
Programando em MIPS

Um Primeiro Programa



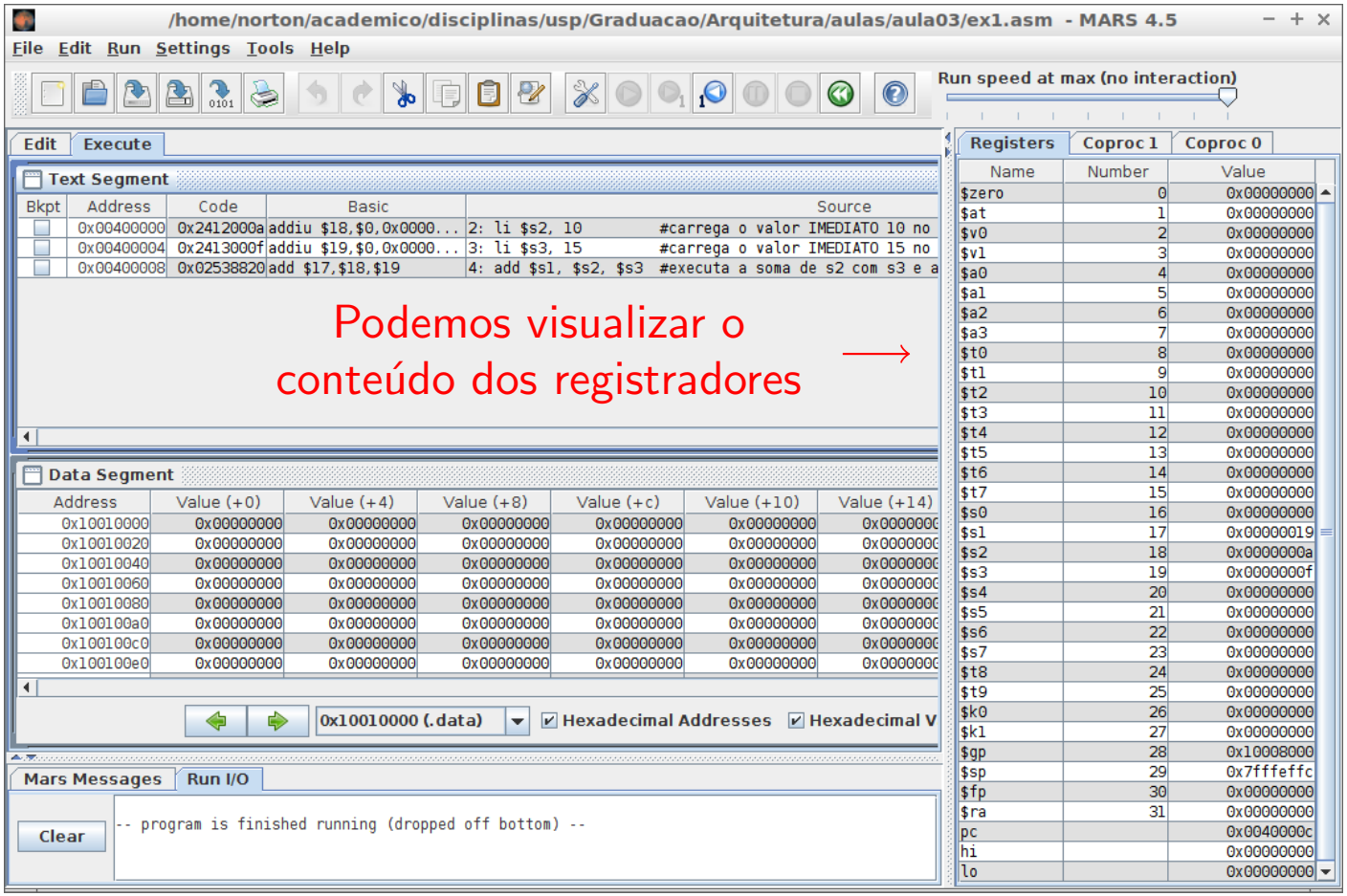
Programando em MIPS

Um Primeiro Programa



Programando em MIPS

Um Primeiro Programa



Programando em MIPS

Instruções Relacionais

- Possuem o mesmo formato das aritméticas
  - operação \$resultado, \$operando$_1$, \$operando$_2$
  - Ex: `slt $t0, $s2, $s3`

Programando em MIPS

Instruções Relacionais

- Possuem o mesmo formato das aritméticas
  - operação \$resultado, \$operando$_1$, \$operando$_2$
  - Ex: `slt $t0, $s2, $s3`
- Existem, contudo, apenas 2 instruções
  - SLT e SLTI
  - De fato, há mais duas, relacionadas a desvios, que veremos mais adiante

Programando em MIPS

Instruções Relacionais

- SLT – *Set on Less Than*
  - `slt $t0, $s1, $s2`
  - `$t0` será 1 se `$s1` $<$ `$s2`. Do contrário, `$t0` será 0
  - Equivale a `$st0` $\leftarrow$ `$s1` $<$ `$s2`

Programando em MIPS

Instruções Relacionais

- SLT – *Set on Less Than*
  - `slt $t0, $s1, $s2`
  - `$t0` será 1 se `$s1` $<$ `$s2`. Do contrário, `$t0` será 0
  - Equivale a `$st0` $\leftarrow$ `$s1` $<$ `$s2`
- SLTI – *Set on Less Than Immediate* (com sinal)
  - `slti $t0, $s1, 10`
  - Mesmo comportamento de `slt`
  - Exite também a versão para *unsigned int*: `sltiu`

Programando em MIPS

Instruções Lógicas

- São 7 as instruções lógicas em MIPS
  - Deslocamento à esquerda (SLL)
  - Deslocamento à direita (SRL)
  - *And* bit a bit (AND, ANDI)
  - *Or* bit a bit (OR, ORI)
  - *Nor* bit a bit (NOR)

Programando em MIPS

Instruções Lógicas

- SLL – *Shift Left Logical*
  - `sll` resultado, origem, deslocamento
  - `sll $t2, $s0, 4`
    - Desloque o conteúdo de `$s0` 4 bits para a esquerda e armazene o resultado em `$t2`

Programando em MIPS

Instruções Lógicas

- SLL – *Shift Left Logical*
  - `sll` resultado, origem, deslocamento
  - `sll $t2, $s0, 4`
    - Desloque o conteúdo de `$s0` 4 bits para a esquerda e armazene o resultado em `$t2`
- SRL – *Shift Right Logical*
  - `srl` resultado, origem, deslocamento
  - `srl $t2, $s0, 2`
    - Desloque o conteúdo de `$s0` 2 bits para a direita e armazene o resultado em `$t2`

Programando em MIPS

Instruções Lógicas

- SLL – *Shift Left Logical*
  - `sll` resultado, origem, deslocamento
  - `sll $t2, $s0, 4`
    - Desloque o conteúdo de `$s0` 4 bits para a esquerda e armazene o resultado em `$t2`
- SRL – *Shift Right Logical*
  - `srl` resultado, origem, deslocamento
  - `srl $t2, $s0, 2`
    - Desloque o conteúdo de `$s0` 2 bits para a direita e armazene o resultado em `$t2`

São chamadas lógicas por não preservarem sinal nem distinguirem expoente de mantissa

Programando em MIPS

Instruções Lógicas

- SLL – *Shift Left Logical*
  - `sll` resultado, origem, deslocamento
  - `sll $t2, $s0, 4`
    - Desloque o conteúdo de `$s0` 4 bits para a esquerda e armazene o resultado em `$t2`

Os bits são simplesmente movidos sem preocupação com seu significado

- SRL – *Shift Right Logical*
  - `srl` resultado, origem, deslocamento
  - `srl $t2, $s0, 2`
    - Desloque o conteúdo de `$s0` 2 bits para a direita e armazene o resultado em `$t2`

Programando em MIPS

Instruções Lógicas

- AND
  - `and resultado, origem`$_1$`, origem`$_2$
  - `and $t0, $s1, $s2`
    - $t0 recebe o resultado de um *AND* bit a bit entre $s1 e $s2

Programando em MIPS

Instruções Lógicas

- AND
  - `and resultado, origem`$_1$`, origem`$_2$
  - `and $t0, $s1, $s2`
    - $t0 recebe o resultado de um *AND* bit a bit entre $s1 e $s2
- ANDI – *AND Immediate*
  - `andi resultado, origem`$_1$`, valor`
  - `andi $t0, $s2, 10`
    - $t0 recebe o resultado de um *AND* bit a bit entre $s2 e 10

Programando em MIPS

Instruções Lógicas

- OR
  - `or resultado, origem`$_1$`, origem`$_2$
  - `or $t0, $s1, $s2`
    - $t0 recebe o resultado de um *OR* bit a bit entre $s1 e $s2

Programando em MIPS

Instruções Lógicas

- OR
  - `or resultado, origem`$_1$`, origem`$_2$
  - `or $t0, $s1, $s2`
    - $t0 recebe o resultado de um *OR* bit a bit entre $s1 e $s2
- ORI – *OR Immediate*
  - `ori resultado, origem`$_1$`, valor`
  - `ori $t0, $s2, 10`
    - $t0 recebe o resultado de um *OR* bit a bit entre $s2 e 10

Programando em MIPS

Instruções Lógicas

- NOR
  - Implementado assim para seguir o padrão de 2 operandos
  - Podemos, contudo, simular o NOT usando NOR:

A NOR 0
NOT (A OR 0)
NOT A

Programando em MIPS

Instruções Lógicas

- NOR
  - Implementado assim para seguir o padrão de 2 operandos
  - Podemos, contudo, simular o NOT usando NOR:

    A NOR 0
    NOT (A OR 0)
    NOT A

  - `nor resultado, origem`$_1$`, origem`$_2$
  - `nor $t0, $s1, $zero`
    - $t0 recebe o resultado de um *NOT* bit a bit em $s1
    - Lembre-se que todos os 32 (ou 64) bits são invertidos

Programando em MIPS

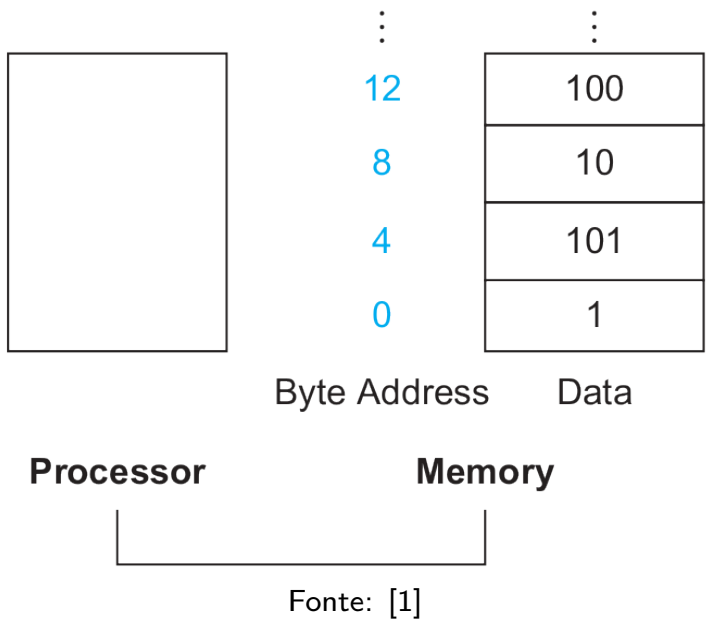
Endereçamento de Memória

- MIPS endereça cada byte de memória
  - Com 32 bits, endereços de palavras adjacentes diferem em 4
  - Temos assim $2^{32} \div 4 = 2^{30}$ palavras de memória

Programando em MIPS

Endereçamento de Memória

- MIPS endereça cada byte de memória
  - Com 32 bits, endereços de palavras adjacentes diferem em 4
  - Temos assim $2^{32} \div 4 = 2^{30}$ palavras de memória
- A memória então é vista como um arranjo unidimensional
  - Com o endereço funcionando como índice

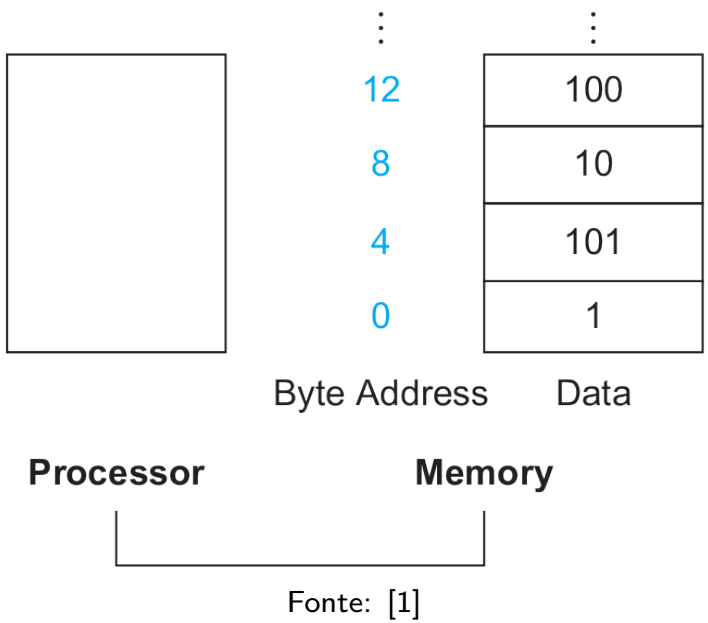


Fonte: [1]

Programando em MIPS

Endereçamento de Memória

- Mas e como os bytes estão arranjados dentro da palavra?
  - O mais significativo (*big-endian*) ou o menos significativo (*little-endian*) no endereço mais baixo?



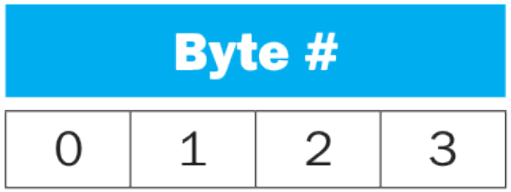
Fonte: [1]

Programando em MIPS

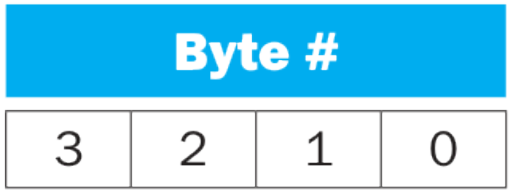
Endereçamento de Memória

- Mas e como os bytes estão arranjados dentro da palavra?
  - O mais significativo (*big-endian*) ou o menos significativo (*little-endian*) no endereço mais baixo?

Big-endian



Little-endian



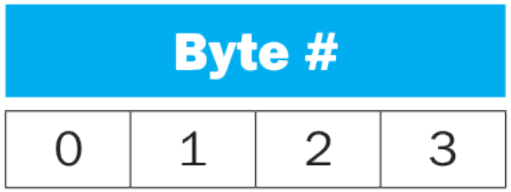
Fonte: [1]

Programando em MIPS

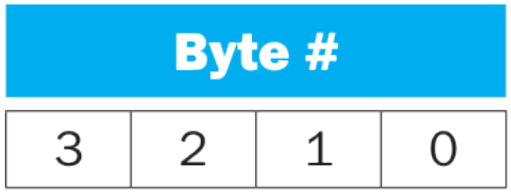
Endereçamento de Memória

- Mas e como os bytes estão arranjados dentro da palavra?
  - O mais significativo (*big-endian*) ou o menos significativo (*little-endian*) no endereço mais baixo?
- MIPS é do tipo *big-endian*

Big-endian



Little-endian



Fonte: [1]

Programando em MIPS

Instruções de Acesso à Memória

- MIPS possui diversas instruções de acesso à memória
  - As principais sendo `lw` e `sw`

Programando em MIPS

Instruções de Acesso à Memória

- MIPS possui diversas instruções de acesso à memória
  - As principais sendo `lw` e `sw`
- LW – *Load Word*
  - Transfere uma palavra da RAM para um registrador
  - `lw registrador, deslocamento(registrador_base)`
  - `lw $t0, 30($s0)`
    - Carregue em $t0 a palavra no endereço correspondente a $s0 + 30
    - O deslocamento (aqui 30) pode ser negativo

Programando em MIPS

Instruções de Acesso à Memória

- SW – *Store Word*
  - Transfere uma palavra de um registrador para a RAM
  - `sw registrador, deslocamento(registrador_base)`
  - `sw $t0, 30($s0)`
    - Armazene o conteúdo de $t0 no endereço correspondente a $s0 + 30
    - Novamente, o deslocamento pode ser negativo

Programando em MIPS

Instruções de Acesso à Memória

- E como sabemos o endereço de memória?
  - Ou especificamos, carregando o registrador com `li` (cuidado!!!)
  - Ou deixamos que o montador tome conta disso

Programando em MIPS

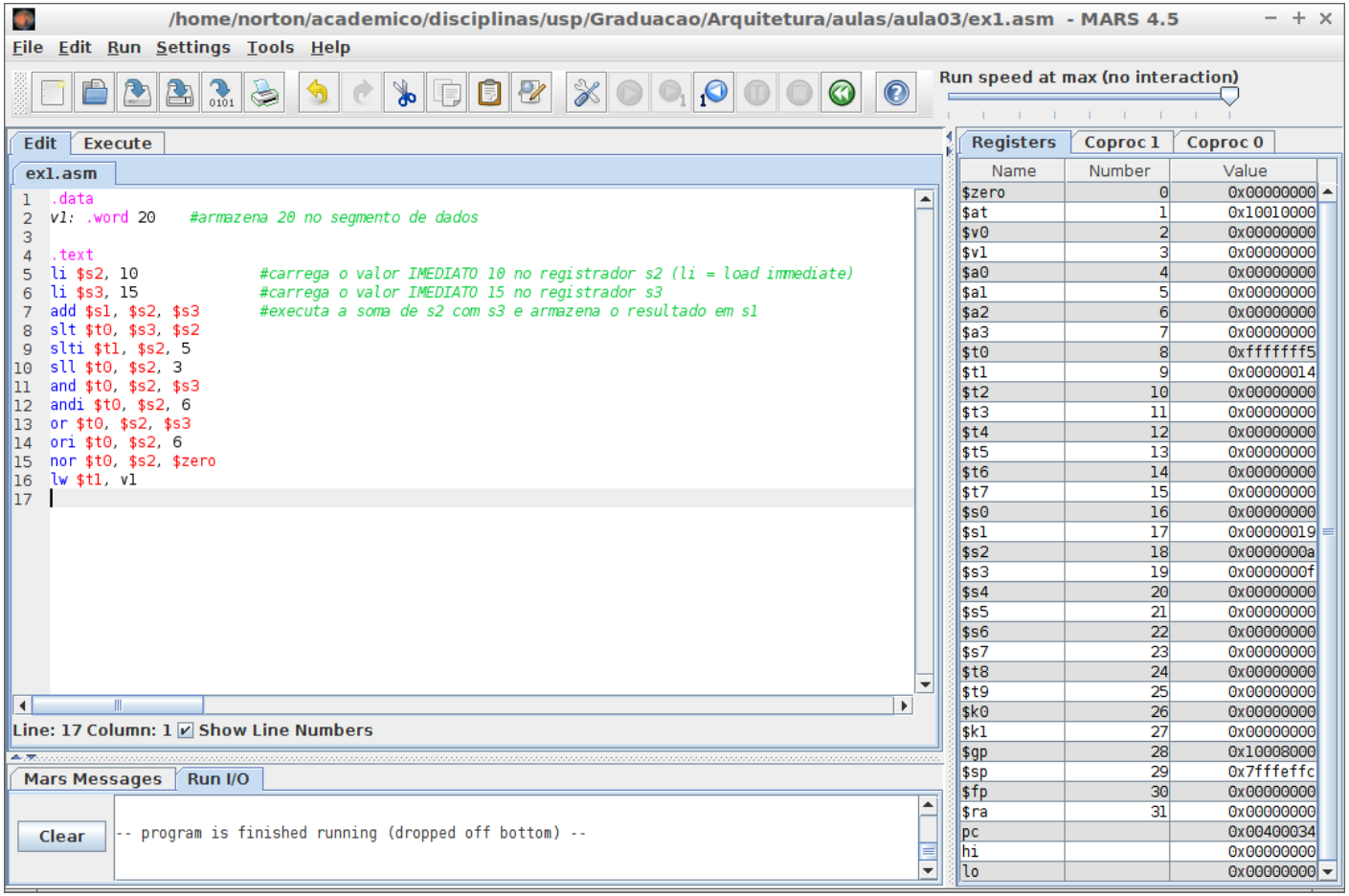
Instruções de Acesso à Memória

- E como sabemos o endereço de memória?
  - Ou especificamos, carregando o registrador com `li` (cuidado!!!)
  - Ou deixamos que o montador tome conta disso
- Como?
  - Usando as diretivas de montador MIPS
  - Ex:

```
.data
v1:   .word 20
```

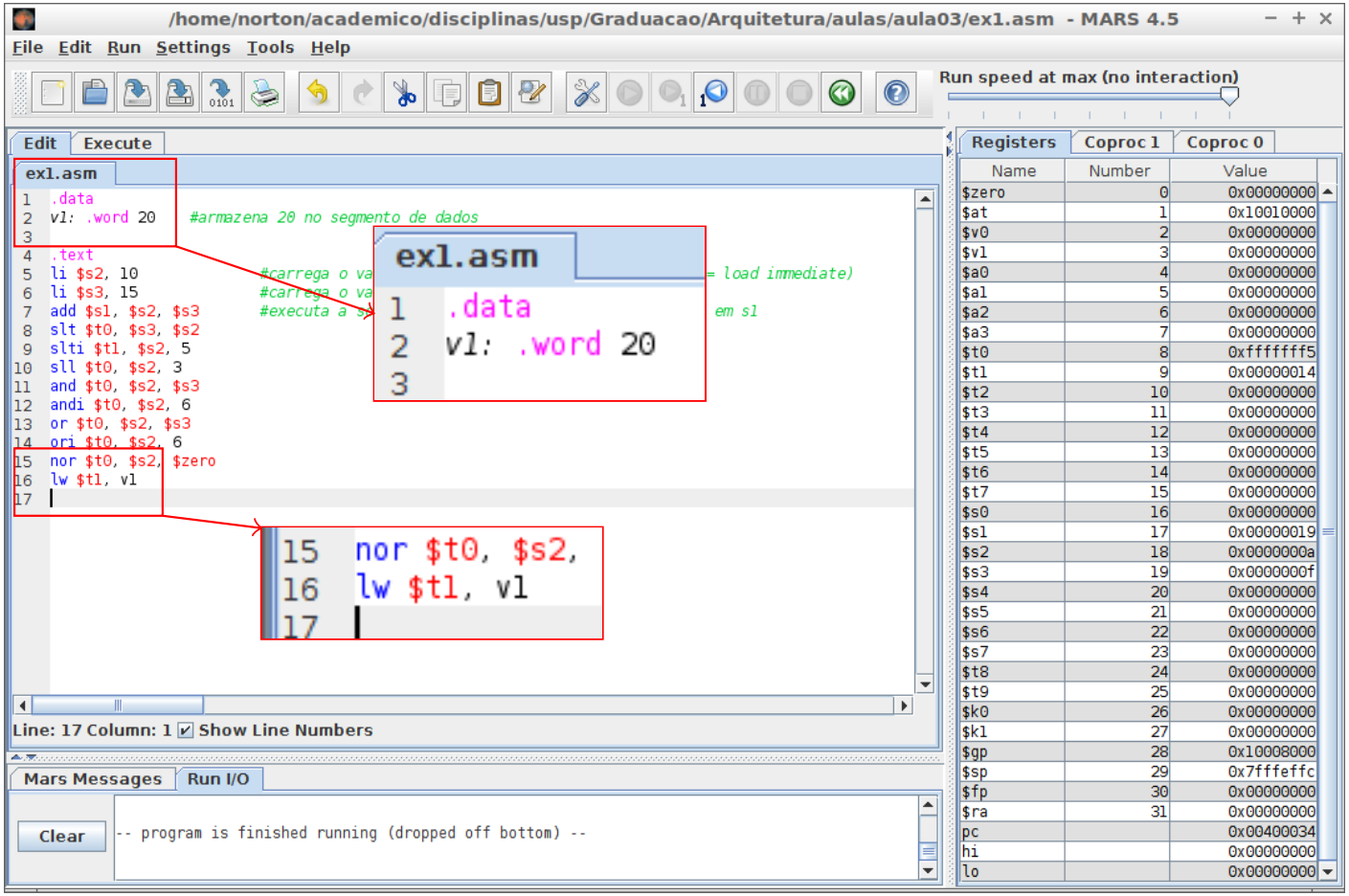
Programando em MIPS

Instruções de Acesso à Memória



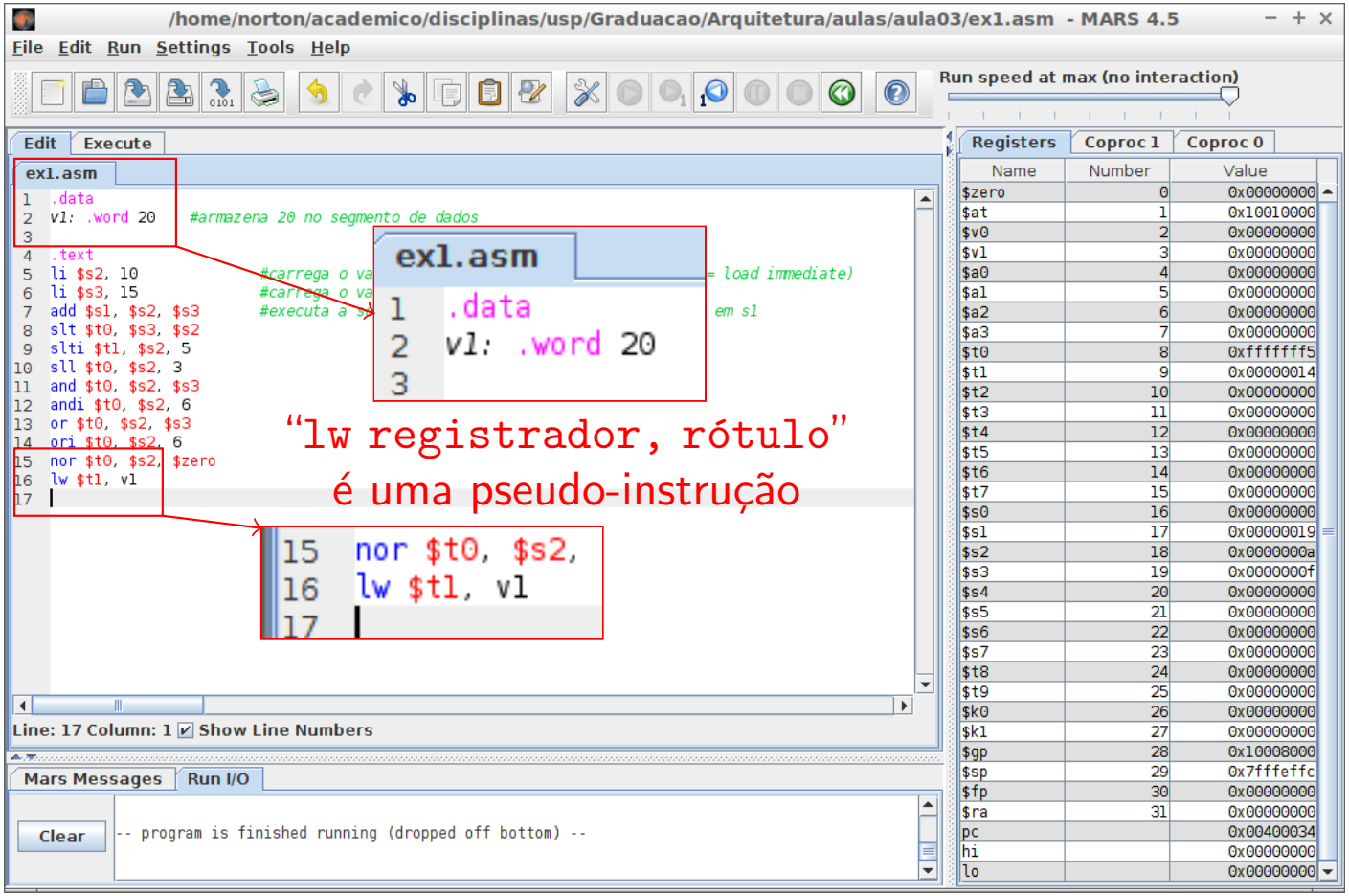
Programando em MIPS

Instruções de Acesso à Memória



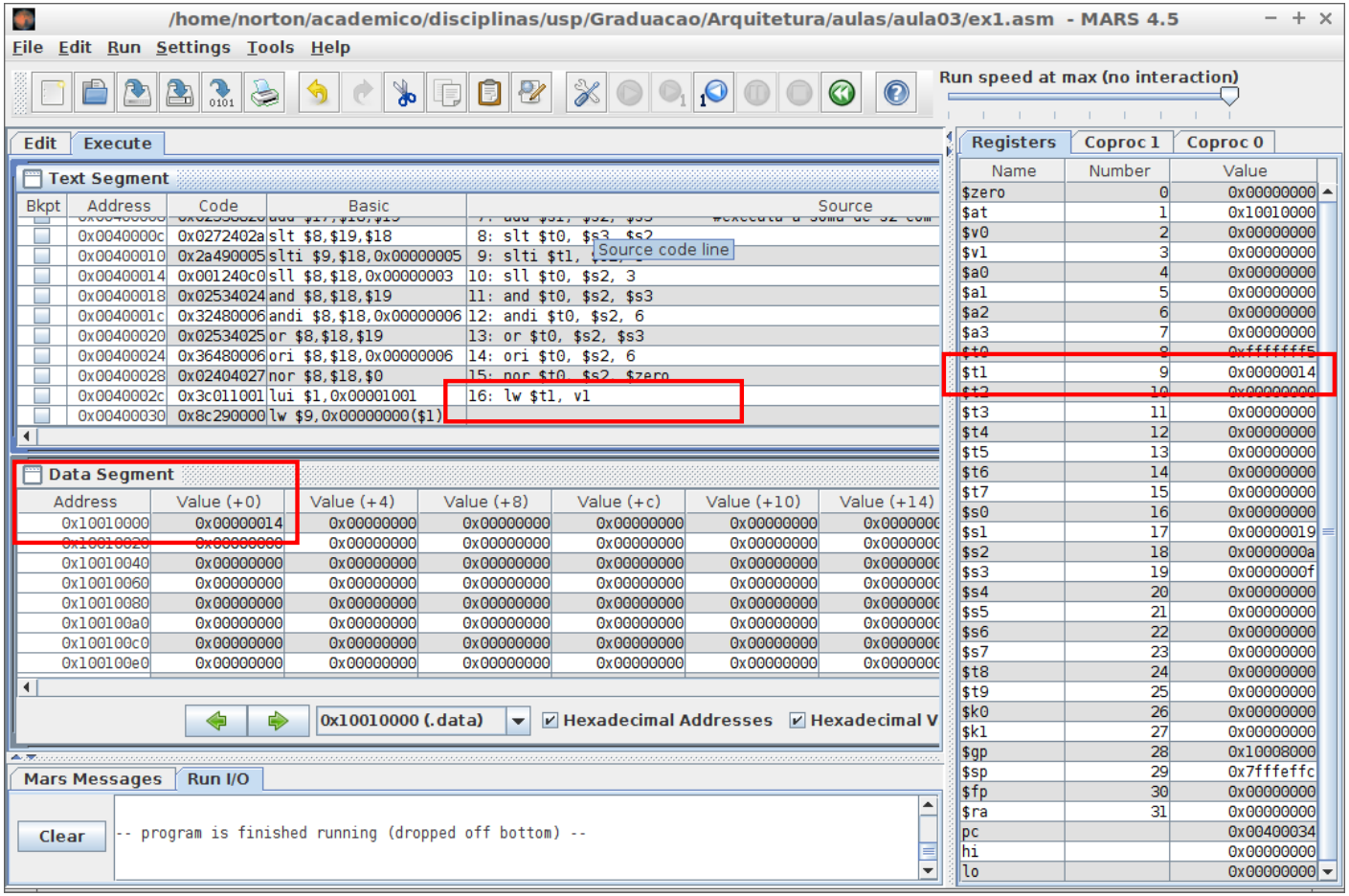
Programando em MIPS

Instruções de Acesso à Memória



Programando em MIPS

Instruções de Acesso à Memória



Programando em MIPS

Arranjos

- Suponha que temos um arranjo em memória

```
.data
arr:   .word 2, 0, -1, 5, -3, 2, 2, 10, 9, -2
```

Programando em MIPS

Arranjos

- Suponha que temos um arranjo em memória

```
.data
arr:   .word 2, 0, -1, 5, -3, 2, 2, 10, 9, -2
```

- Como acessar o elemento `arr[3]`?
  - Precisamos armazenar o endereço-base do arranjo em um registrador, para calcular o deslocamento
  - `la $t1, arr`
    - Pseudo-instrução que carrega em $t1 o endereço representado por `arr`

Programando em MIPS

Arranjos

- Como acessar o elemento `arr[3]` (cont.)?
  - `lw $t2, 3($t1)`?

Programando em MIPS

Arranjos

- Como acessar o elemento `arr[3]` (cont.)?
  - `lw $t2, 3($t1)`?
    - Mas armazenamos palavras, não bytes
    - O índice precisa ser multiplicado pelo tamanho da palavra

Programando em MIPS

Arranjos

- Como acessar o elemento `arr[3]` (cont.)?
  - `lw $t2, 3($t1)`?
    - Mas armazenamos palavras, não bytes
    - O índice precisa ser multiplicado pelo tamanho da palavra
  - Então `lw $t2, 12($t1)`

```
.data
arr:  .word 2, 0, -1, 5, -3, 2, 2, 10, 9, -2
.text
la $t1, arr
lw $t2, 12($t1)
```

Programando em MIPS

Codificação das Instruções

- MIPS possui 3 famílias de instruções:
  - Instruções do Tipo R (*Register*)
    - `add $s1, $s2, $s3`
    - `sll $t2, $s0, 4`

Programando em MIPS

Codificação das Instruções

- MIPS possui 3 famílias de instruções:
  - Instruções do Tipo R (*Register*)
    - `add $s1, $s2, $s3`
    - `sll $t2, $s0, 4`
  - Instruções do Tipo I (*Immediate*)
    - `addi $t0, $s2, 4`
    - `lw $t2, 12($t1)`

Programando em MIPS

Codificação das Instruções

- MIPS possui 3 famílias de instruções:
  - Instruções do Tipo R (*Register*)
    - `add $s1, $s2, $s3`
    - `sll $t2, $s0, 4`
  - Instruções do Tipo I (*Immediate*)
    - `addi $t0, $s2, 4`
    - `lw $t2, 12($t1)`
  - Instruções do Tipo J (*Jump*)
    - Usadas em desvios (mais adiante)

Programando em MIPS

Codificação das Instruções

- Instruções Tipo R (*Register*)

6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits

Fonte: [1]

Codificação das Instruções

- Instruções Tipo R (*Register*)

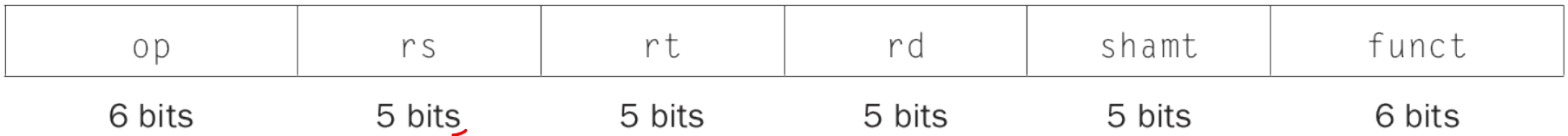
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Fonte: [1]

Código da operação (*opcode*)

Codificação das Instruções

- Instruções Tipo R (*Register*)



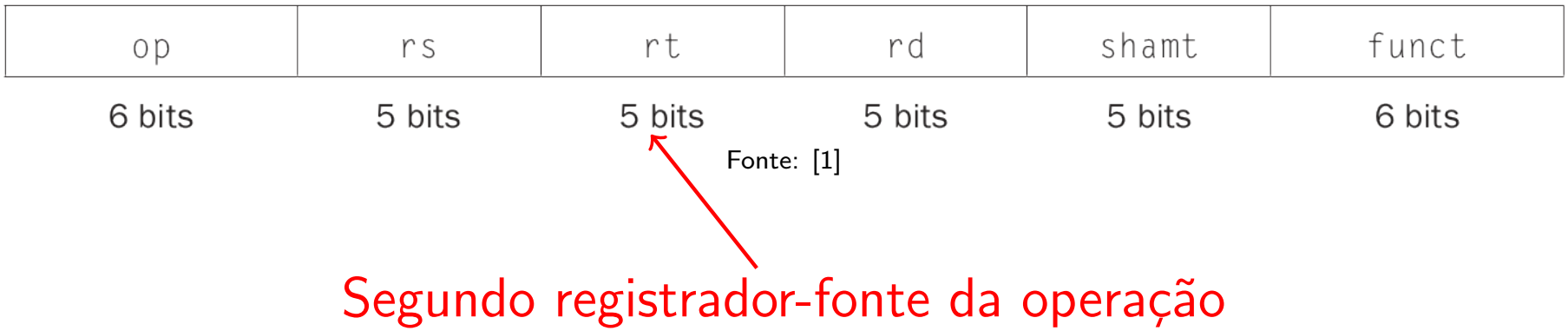
Fonte: [1]

Primeiro registrador-fonte da operação

Programando em MIPS

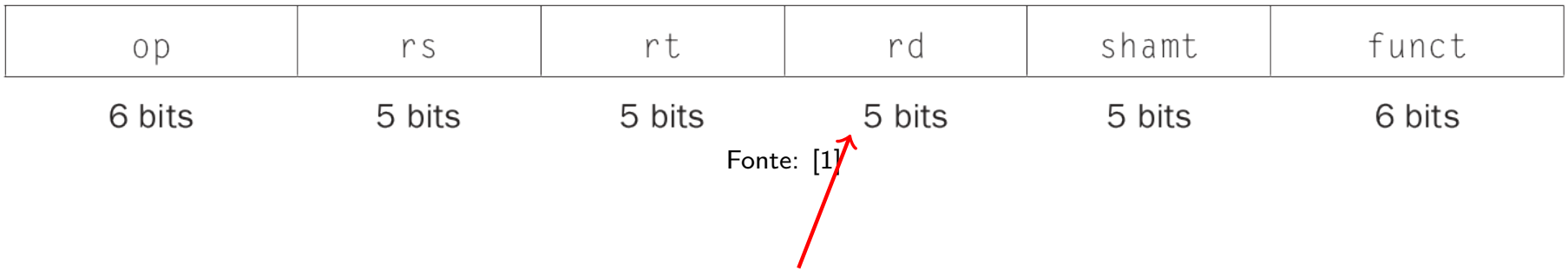
Codificação das Instruções

- Instruções Tipo R (*Register*)



Codificação das Instruções

- Instruções Tipo R (*Register*)



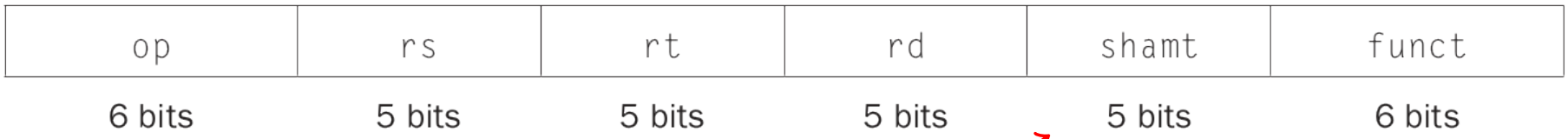
Fonte: [1]

Registrador-destino (recebe o resultado da operação)

Programando em MIPS

Codificação das Instruções

- Instruções Tipo R (*Register*)



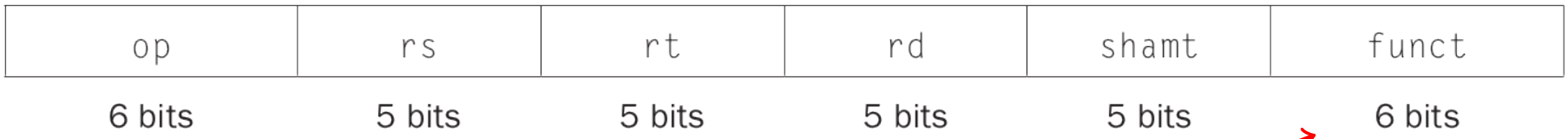
Fonte: [1]

Shift amount – deslocamento (usado em instruções como `sll` e `srl`)

Programando em MIPS

Codificação das Instruções

- Instruções Tipo R (*Register*)



Fonte: [1]

Código de função – seleciona a variante específica da operação no campo *op*

Programando em MIPS

Codificação das Instruções

- Instruções Tipo R (*Register*)

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Fonte: [1]

Campos não usados são carregados com zero

Codificação das Instruções

- Instruções Tipo I (*Immediate*)

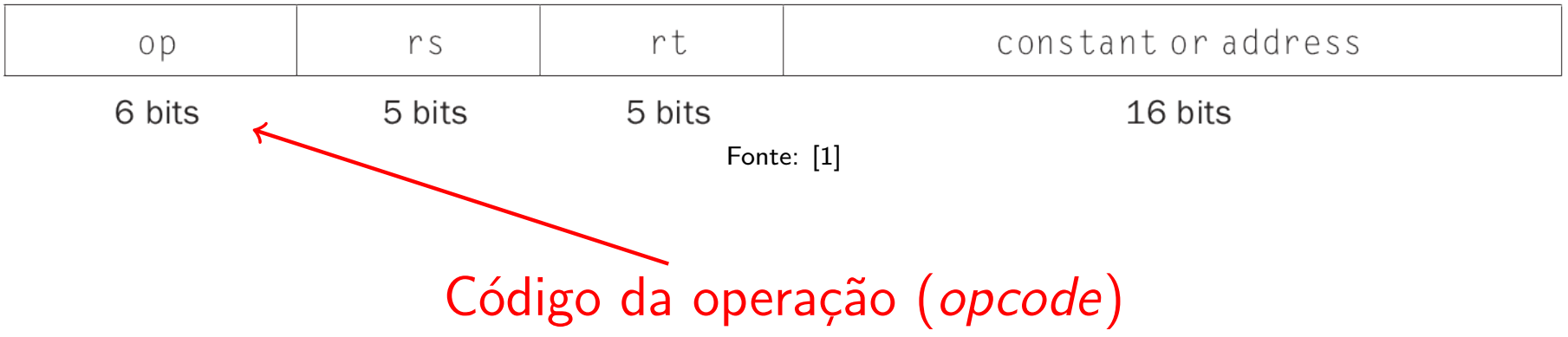
6 bits | 5 bits | 5 bits | 16 bits

Fonte: [1]

Programando em MIPS

Codificação das Instruções

- Instruções Tipo I (*Immediate*)

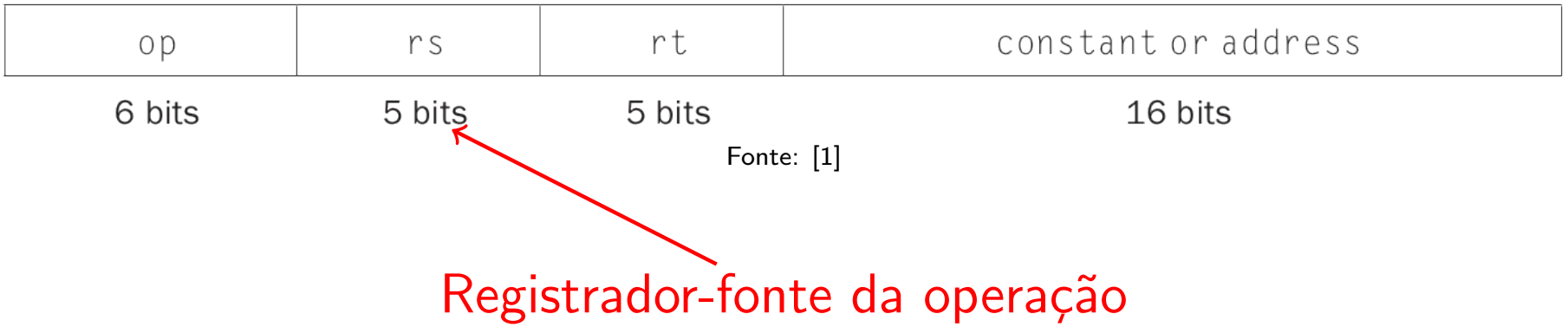


Fonte: [1]

Programando em MIPS

Codificação das Instruções

- Instruções Tipo I (*Immediate*)

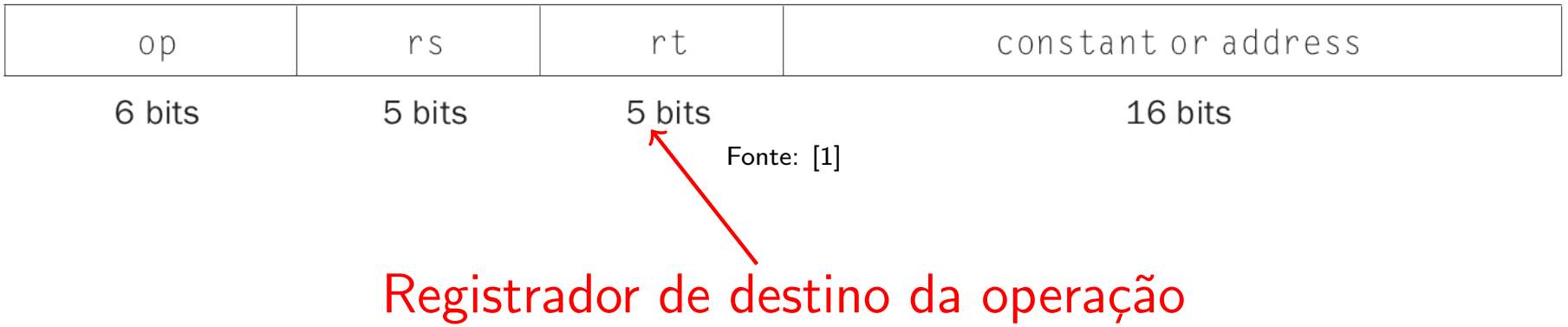


Fonte: [1]

Programando em MIPS

Codificação das Instruções

- Instruções Tipo I (*Immediate*)

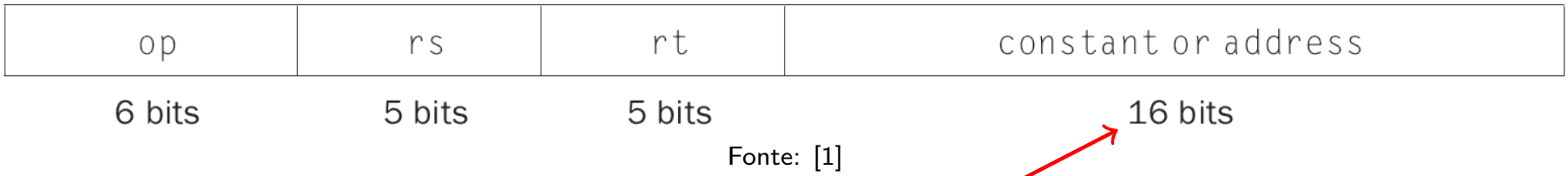


Fonte: [1]

Programando em MIPS

Codificação das Instruções

- Instruções Tipo I (*Immediate*)



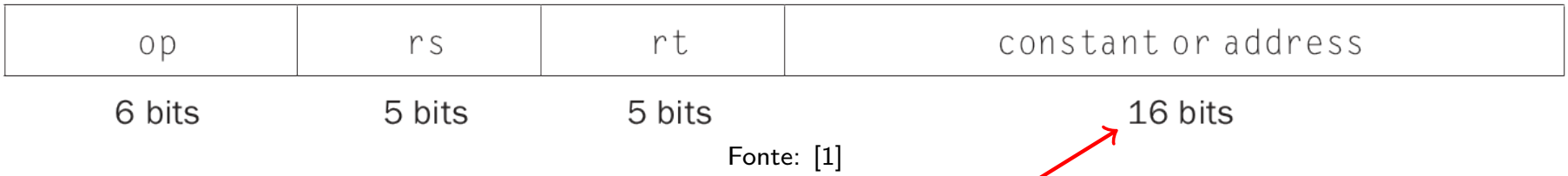
Fonte: [1]

Valor (constante ou endereço) usado na operação

Programando em MIPS

Codificação das Instruções

- Instruções Tipo I (*Immediate*)



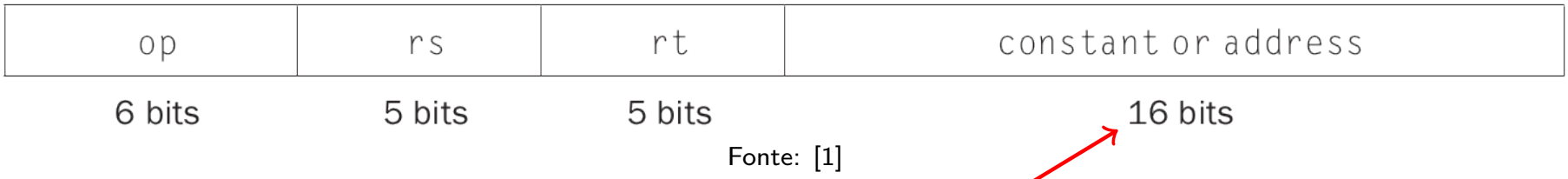
Fonte: [1]

Note que isso nos dá um limite de $\pm 2^{15}$ bytes de deslocamento (há o bit de sinal) em `lw $reg1, desl($reg2)`

Programando em MIPS

Codificação das Instruções

- Instruções Tipo I (*Immediate*)



Fonte: [1]

Por isso, ao apontar para o byte 32K no segmento de dados, $gp dá acesso aos primeiros 64K desse segmento

Programando em MIPS

Codificação das Instruções

- Instruções Tipo J (*Jump*)

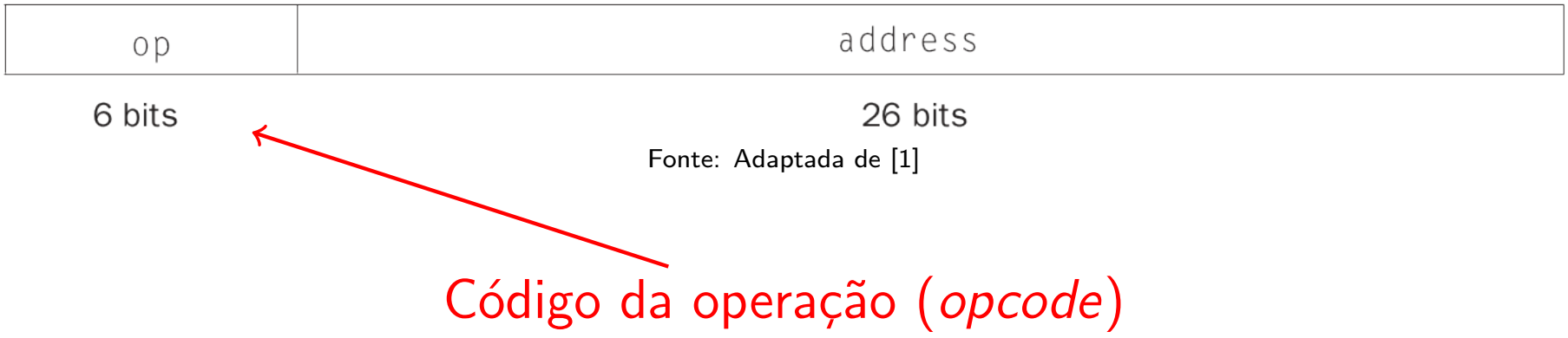
6 bits 26 bits

Fonte: Adaptada de [1]

Programando em MIPS

Codificação das Instruções

- Instruções Tipo J (*Jump*)

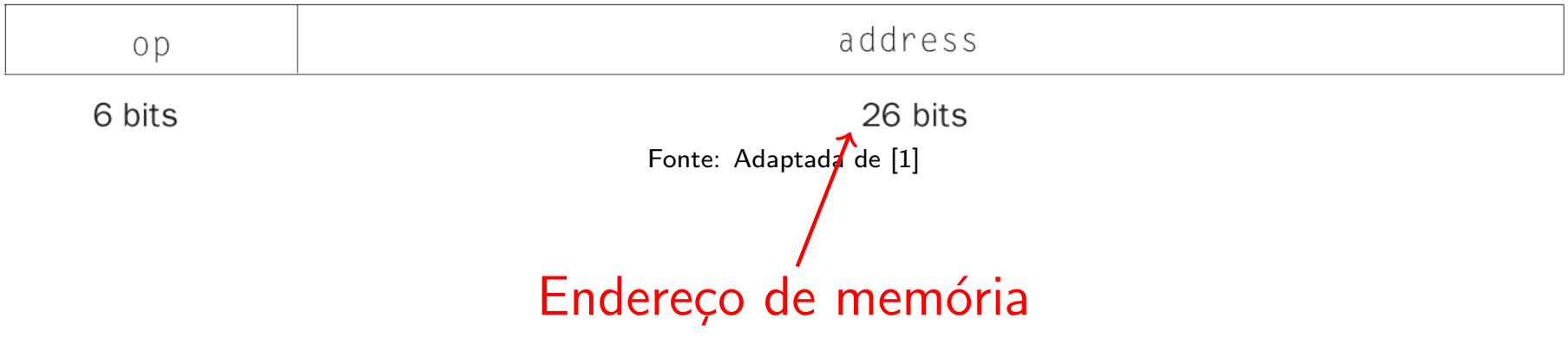


Fonte: Adaptada de [1]

Programando em MIPS

Codificação das Instruções

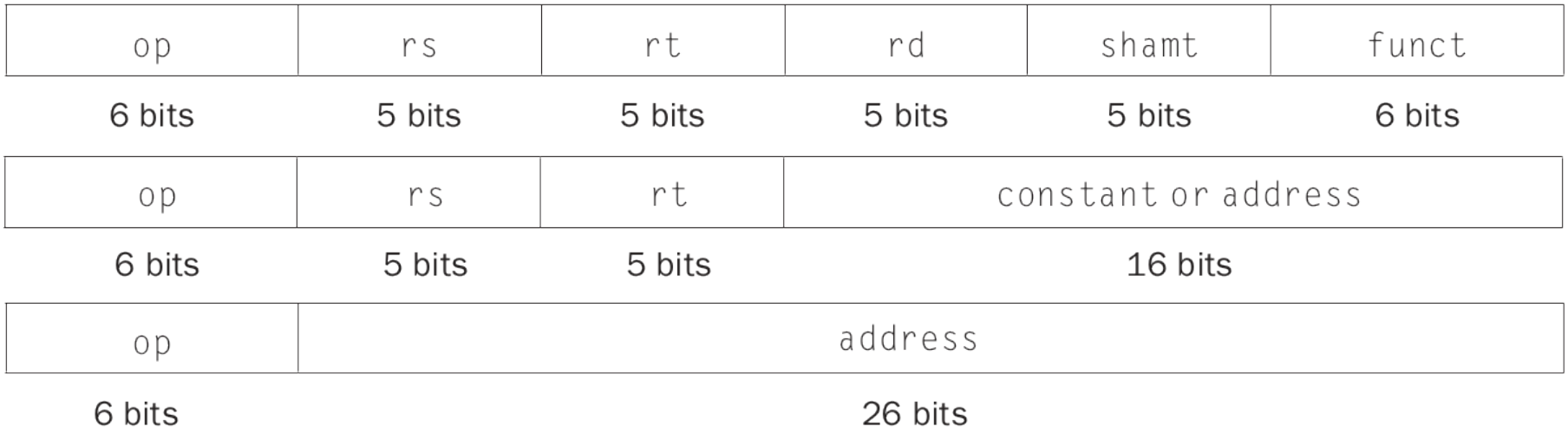
- Instruções Tipo J (*Jump*)



Fonte: Adaptada de [1]

Programando em MIPS

Codificação das Instruções



Fonte: Adaptada de [1]

- Os formatos são diferenciados no circuito pelo valor no campo *op*

Programando em MIPS

Pseudo-instruções

- Instruções adicionadas pelo montador, para simplificar algumas operações
  - Tratadas por ele como se fossem instruções legítimas
  - Traduzidas para instruções reais

Programando em MIPS

Pseudo-instruções

- Instruções adicionadas pelo montador, para simplificar algumas operações
  - Tratadas por ele como se fossem instruções legítimas
  - Traduzidas para instruções reais
- Exemplos:

Pseudo-instrução	*Expansão*
`li $s4, 10`	`addiu $s4, $zero, 10`
`move $t0, $t1`	`add $t0, $zero, $t1`

Referências

1. Patterson, D.A.; Hennessy, J.L. (2013): Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann. 5ª ed.
   - Para detalhes sobre MIPS consulte também o Apêndice A
2. `https://www.embarcados.com.br/arquitetura-de-conjunto-de-instrucoes-mips/`
   - Curso bastante completo, em português
3. `https://sweetcode.io/building-first-simple-program-mips-assembly-language/`
   - Tutorial básico passo a passo
4. `https://sites.cs.ucsb.edu/~franklin/64/lectures/mipsassemblytutorial.pdf`
   - Tutorial mais profundo

Referências

1. WEMIPS
   - `https://rivoire.cs.sonoma.edu/cs351/wemips/`
   - Simulador MIPS online
2. MARS (*MIPS Assembler and Runtime Simulator*)
   - `http://courses.missouristate.edu/kenvollmar/mars/`
   - Simulador MIPS (.jar, multiplataforma)
3. `https://en.wikipedia.org/wiki/Logical_shift`
   - Diferença entre deslocamento lógico e aritmético