Arquitetura e Organização de Computadores

Cap 2. A Linguagem dos Computadores

AP 1 - Operações, operandos e representações

Prof. Dr. João Fabrício Filho

Universidade Tecnológica Federal do Paraná
Campus Campo Mourão
2024

Quais elementos compõem uma linguagem?

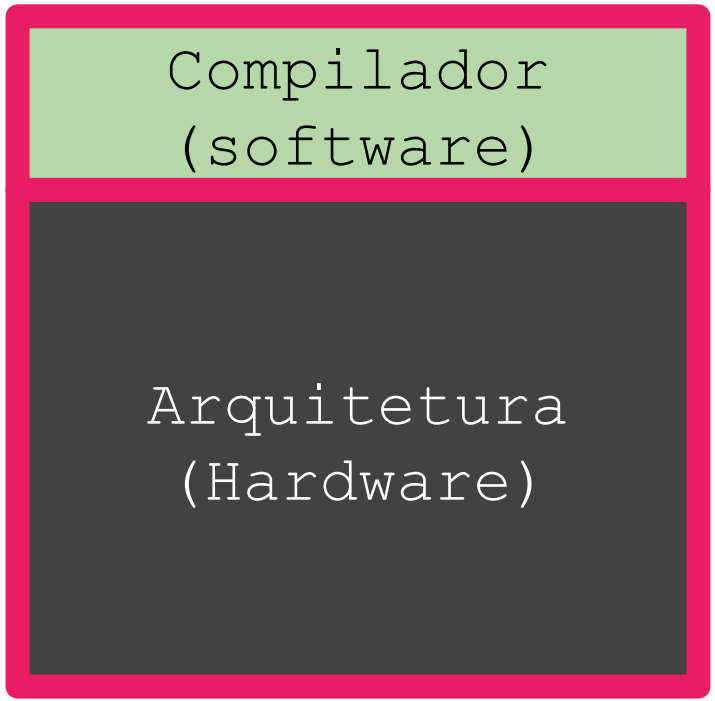
Como fazemos para nos comunicar?

Como nos comunicamos com computadores?

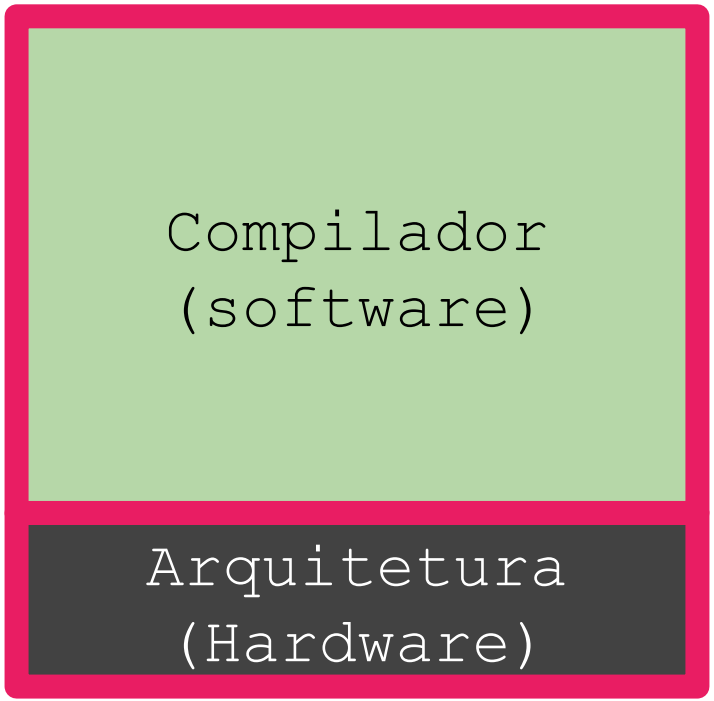
Conjuntos de instruções (ISA)

- O repertório de instruções de um computador
- Diferentes computadores têm diferentes ISAs
  - Podem ter aspectos em comum
- Os primeiros computadores tinham ISAs muito simples
  - Implementação simplificada
- Muitos computadores modernos também possuem ISAs simples

CISC


Compilador
(software)
Arquitetura
(Hardware)

RISC


Compilador
(software)
Arquitetura
(Hardware)

Complex Instruction Set Computer (CISC)

- Instruções complexas
- Qualquer instrução pode referenciar a memória
- Pouco uso do pipeline
- Instruções com formato variável
- Muitas instruções com muitos modos de endereçamento
- A complexidade está no programa
- Poucos registradores

Reduced Instruction Set Computer (RISC)

- Instruções simples
- Referências à memória
  - somente com instruções especiais (load/store)
- Uso intensivo de pipeline
- Instruções de formato fixo
- Poucas instruções com poucos modos de endereçamento
- Compilador complexo
- Vários registradores

Conjunto de instruções MIPS

- Usado como exemplo ao longo do livro
- Stanford MIPS (MIPS Technologies)
- Grande parte do mercado de embarcados
  - Aplicações em eletrônicos de consumo, equipamentos de rede/armazenamento, câmeras, impressoras
- Típico de muitas ISA modernas
  - Veja o MIPS ref card

Operações aritméticas

- Formato estrito
  - Um destino e duas fontes

```
add a, b, c # a recebe b + c

sub d, e, f # d recebe e - f
```

- Princípio de projeto 1: Simplicidade favorece regularidade
  - Regularidade permite implementação mais simples
  - Simplicidade permite maior desempenho e menor custo

Exemplo em C

```
a = b + c + d + e
```

Como podemos implementar em MIPS?

Exemplo em C

```
a = b + c + d + e
```

Como podemos implementar em MIPS?

```
add a, b, c # a recebe b + c

add a, a, d # a recebe b + c + d

add a, a, e # a recebe b + c + d + e
```

Exemplo em C (2)

```
f = (g + h) - (i + j)
```

Como podemos implementar em MIPS?

Exemplo em C (2)

```
f = (g + h) - (i + j)
```

Como podemos implementar em MIPS?

```
add t0, g, h # temp t0 = g + h

add t1, i, j # temp t1 = i + j

sub f, t0, t1 # f = t0 - t1
```

Operando Registradores

- Instruções aritméticas usam registradores como operandos
- O MIPS tem um conjunto com 32 registradores de 32 bits
  - Uso para dados frequentemente acessados
  - Numerados de 0 a 31
  - Dados de 32 bits são chamados de palavra (word)
- Nomes simbólicos dados pelo montador
  - $t0, $t1, …, $t9 para valores temporários
  - $s0, $s1, …, $s7 para mapear variáveis

- Princípio de projeto 2: menor significa mais rápido
  - Memória principal: milhões de localizações

Exemplo em C (2)

```
f = (g + h) - (i + j)
```

Como podemos implementar em MIPS?

```
add $t0, $s1, $s2 # temp t0 = g + h

add $t1, $s3, $s4 # temp t1 = i + j

sub $s0, $t0, $t1 # f = t0 - t1
```

Operandos imediatos

- Dados constantes especificados na instrução

```
addi $s1, $s2, 4
```

- Não existe instrução de subtração imediata
  - Basta usar uma constante negativa

```
addi $s1, $s2, -1
```

- Princípio de projeto 3: Faça o caso comum rápido
  - Constantes pequenas são comuns
  - Operando imediato evita uma instrução *load*

A constante zero

- No MIPS o registrador $zero é a constante 0
  - Não pode ser alterado
- Útil em operações comuns
  - Ex: mover dados entre registradores

```
add $t2, $s1, $zero # $t2 = $s1
```

Inteiros binários sem sinal

- Dado um número de n bits

  $x = (x_{n-1} * 2^{n-1}) + (x_{n-2} * 2^{n-2}) + \ldots + (x_1 * 2^1) + (x_0 * 2^0)$

- Intervalo: 0 até 2^n-1
- Usando 32 bits: 0 até +4.294.967.295
- Exemplo

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$

  $= 0 + \ldots + 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$

  $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

Inteiros binários com sinal - complemento de 2

- Dado um número de n bits

  $x = (-x_{n-1} * 2^{n-1}) + (x_{n-2} * 2^{n-2}) + \ldots + (x_1 * 2^1) + (x_0 * 2^0)$

- Intervalo: 0 até 2^n-1
- Usando 32 bits: -2.147.483.648 até +2.147.483.647
- Exemplo

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$

  $= 0 + \ldots + 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$

  $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

Complemento de 2

$$
\begin{aligned}
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{bin} &= 0_{dec} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{bin} &= 1_{dec} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{bin} &= 2_{dec} \\
\ldots & \ldots \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{bin} &= 2.147.483.645_{dec} \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{bin} &= 2.147.483.646_{dec} \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{bin} &= 2.147.483.647_{dec} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{bin} &= -2.147.483.648_{dec} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{bin} &= -2.147.483.647_{dec} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{bin} &= -2.147.483.646_{dec} \\
\ldots & \ldots \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{bin} &= -3_{dec} \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{bin} &= -2_{dec} \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{bin} &= -1_{dec}
\end{aligned}
$$

Inteiros binários com sinal - complemento de 2

- Bit 31 é o bit de sinal
  - 1 para números negativos
  - 0 para números positivos
- $-(-2^{n-1})$ não pode ser representado
- Números positivos têm a mesma representação sem sinal e em complemento de 2
- Alguns números específicos
  - 0: 0000 0000 0000 0000 0000 0000 0000 0000
  - -1: 1111 1111 1111 1111 1111 1111 1111 1111
  - Menor número: 1000 0000 0000 0000 0000 0000 0000 0000
  - Maior número: 0111 1111 1111 1111 1111 1111 1111 1111

Inteiros com sinal - Complemento de 2

- Atalho para negação
- Complementar e somar 1
  - Complementar significa inverter (de 0 para 1 ou de 1 para 0)

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Exemplo: Negue +2
  - $+2 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2$
  - $-2 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 + 1 = 1111\ \ldots\ 1110_2$

Qual a saída do código abaixo?

```
int a = 2147483647;
a++;
printf("%d", a);
```

Operações com binários sem sinal

- O padrão é lidar com complemento de 2.
- As operações U (*unsigned*) lidam com dados sem sinal.
- Basta usar o similar com o final u

```
addu  $t0, $t1, $t2

subu  $s0, $s3, $s4

addiu $s1, $s2, 3
```

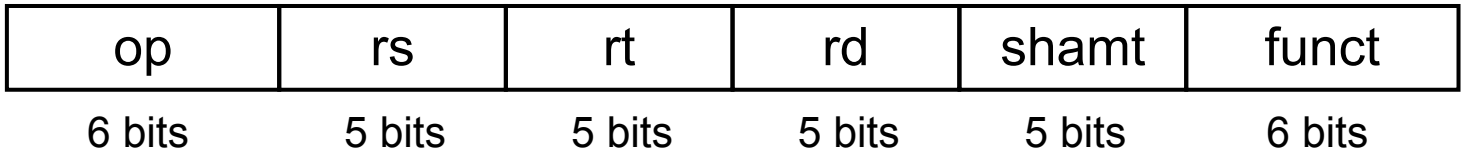
Extensão do sinal

- Representação de um número usando mais bits
  - Preserva o valor numérico
- Na ISA do MIPS
  - addi: estende o valor do imediato
  - lb, lh: estende o byte/halfword carregado
  - beq, bne: estende o deslocamento
- Replica o valor de bit de sinal para a esquerda
  - Valores não sinalizados: estende com zeros
- Exemplos 8 bits para 16 bits
  - +2: **0**000 0010 -> 0000 0000 0000 0010
  - -2: **1**111 1110 -> 1111 1111 1111 1110

Representação das instruções

- Instruções são codificadas em binário
  - Chamadas de código de máquina
- Instruções MIPS
  - Codificadas como palavras de 32 bits
  - Pequeno número de formatos de códigos de operação (opcode), número de registradores
  - Regularidade!
- Número de registradores
  - $t0 - $t7 são os registradores 8 - 15
  - $t8 - $t9 são os registradores 24 - 25
  - $s0 - $s7 são os registradores 16 - 23

Instruções MIPS formato R



- op: código da operação (opcode)
- rs: registrador com primeiro operando origem
- rt: registrador com segundo operando origem
- rd: número do registrador destino
- shamt: *shift amount* (0 por enquanto)
- funct: código de função (estende o opcode)

Formato R - Exemplo

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

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

$0000\ 0010\ 0011\ 0010\ 0100\ 0000\ 0010\ 0000_2 = 0x02324020_{16} = 36847648_{10}$

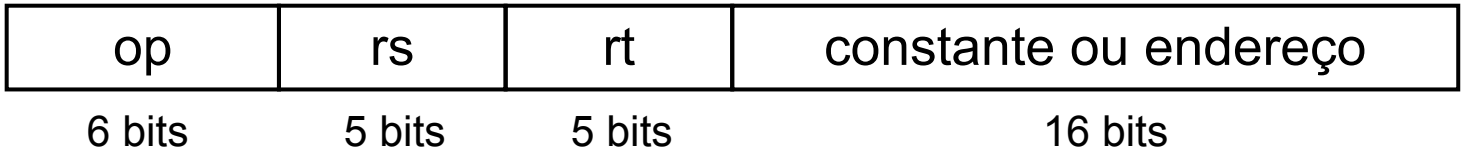
Hexadecimal

- Base 16
  - Representação compacta da cadeia de bits
  - 4 bits por dígito hexadecimal

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Exemplo: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

Instruções MIPS Formato I



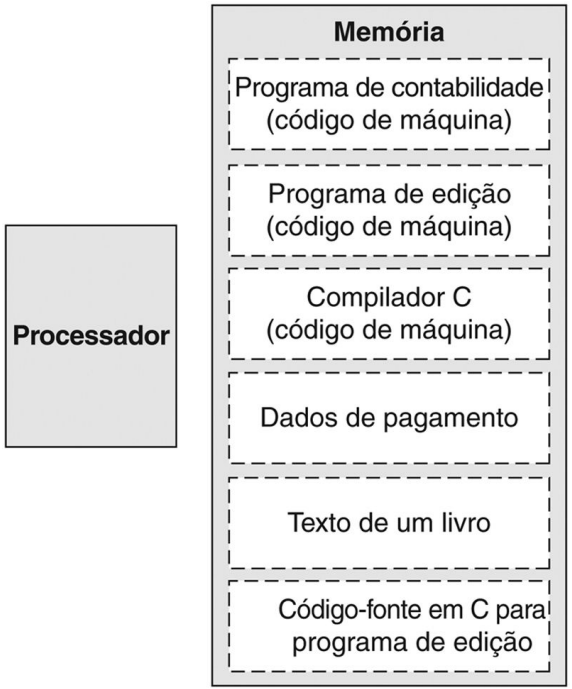
Instruções aritméticas com imediato e load/store

- rt: número do registrador destino ou origem
- Constante: -2^{15} até $+2^{15}-1$
- Endereço: endereço adicionado ao endereço base no registrador rs

- Princípio de projeto 4: Bom projeto exige bons compromissos
  - Manter todas as instruções com o mesmo tamanho
  - Exigindo diferentes tipos de formato para diferentes tipos de instruções

Computadores com programa armazenado

The BIG Picture



- Instruções são representadas em binário, como os dados
- Instruções e dados armazenados na memória
- Programas podem operar sobre programas
  - Ex., compiladores, linkers, …
- Compatibilidade binária permite que programas compilados possam trabalhar em diferentes computadores
  - ISAs padronizados

Operações lógicas

Instruções para manipulação bit a bit

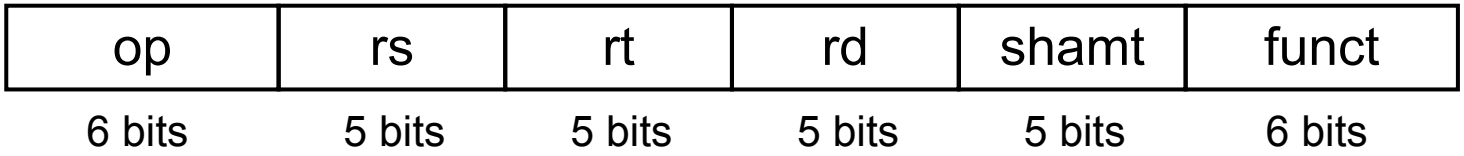
Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

Úteis para extração e inserção de grupos de bits em palavras

“Ao contrário”, continuou Tweedledee, “se foi assim, poderia ser; e se assim fosse, seria; mas como não é, então não é. Isso é lógico.”

Lewis Carroll, Alice no país das maravilhas, 1865

Operações de deslocamento

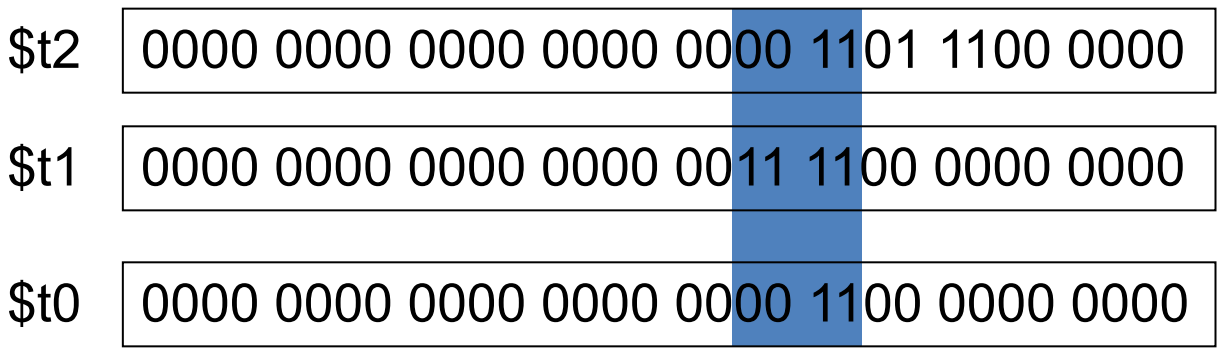


- shamt: quantas posições para o deslocamento
- Shift left lógico
  - Desloca para a esquerda e preenche com bits 0
  - *sll* por i bits multiplica por 2^i
- Shift right lógico
  - Desloca para a direita e preenche com bits 0
  - *srl* por i bits multiplica por 2^i

Operação AND

- Útil para aplicar máscara de bits em uma palavra
  - Seta alguns bits (=1), zera outros

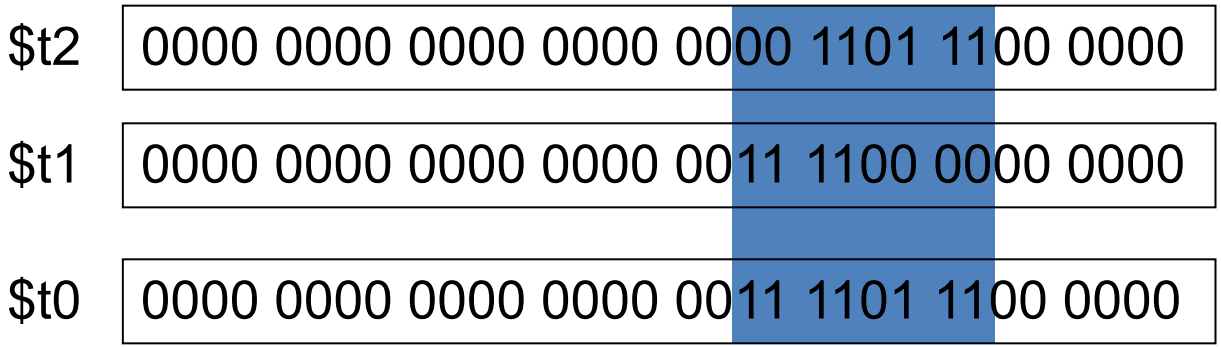
```
and $t0, $t1, $t2
```



Operação OR

- Útil para incluir bits em uma palavra
  - Seleciona alguns bits (=1), deixa os outros inalterados

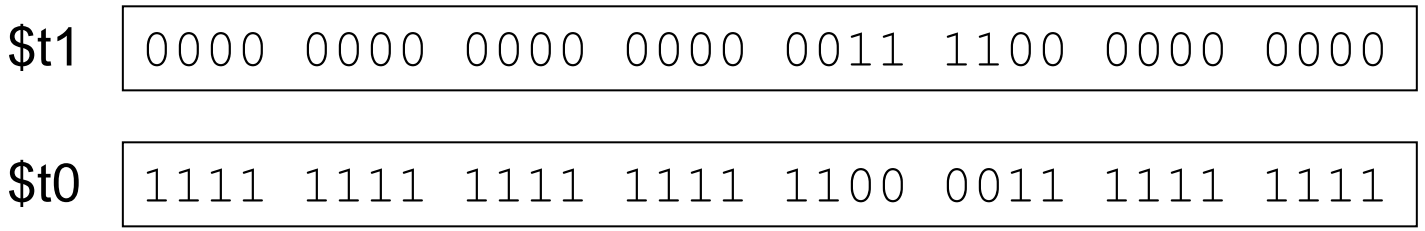
```
or $t0, $t1, $t2
```



Operação NOT

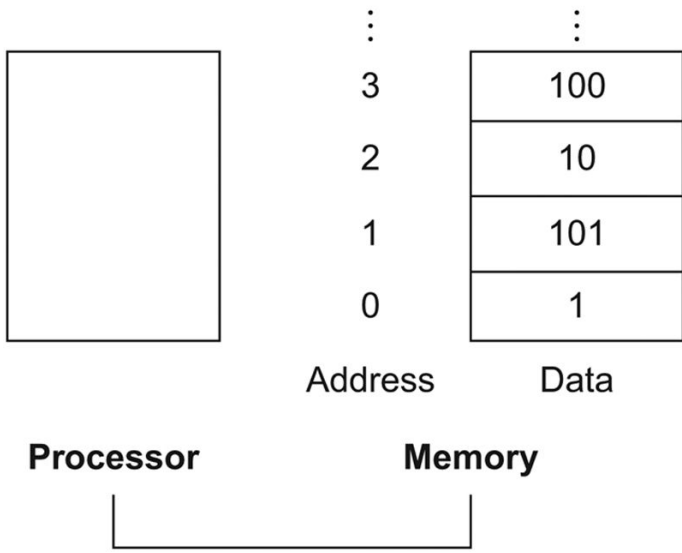
- Útil para inverter bits em uma palavra
  - Muda de 0 para 1 e de 1 para 0
- NOT no MIPS = instrução NOR com 3 operandos

```
nor $t0, $t1, $zero
```



Operandos em memória

- Memória principal usada para compor dados
  - Vetores, estruturas, dados dinâmicos
- Para operações aritméticas:
  - Carregar valores da memória em registradores
  - Armazenar resultados de registradores em memória
- A memória é endereçada por bytes
  - Cada endereço identifica um byte
  - Palavras são alinhadas na memória: endereços são múltiplos de 4
- MIPS é Big Endian
  - O byte mais significativo da palavra (MSB) está no endereço menor



Exemplo de código C com carregamento da memória

```
g = h + A[8]
```

Considere g em $s1, h em $s2 e o endereço base de A em $s3

O endereçamento em MIPS requer um offset de 4 bytes por palavra (32/4 = 8)

Exemplo de código C com carregamento da memória

```
g = h + A[8]
```

Considere g em $s1, h em $s2 e o endereço base de A em $s3

O endereçamento em MIPS requer um offset de 4 bytes por palavra (32/4 = 8)

offset | registrador-base

```
lw $t0, 32($s3) # $t0 = A[8]

add $s1, $s2, $t0 # $s1 = h + A[8]
```

Exemplo de código C com carregamento da memória (2)

```
A[12] = h + A[8]
```

h em $s2 e o endereço base de A em $s3

Exemplo de código C com carregamento da memória (2)

```
A[12] = h + A[8]
```

h em $s2 e o endereço base de A em $s3

índice 8 requer offset de 32

índice 12 requer offset de 48

```
lw $t0, 32($s3) # $t0 = A[8]
add $t1, $s2, $t0 # $t1 = h + A[8]
sw $t1, 48($s3) # A[12] = $t1
```

Exemplo de código C com carregamento da memória (3)

```
A[i] = B[4]
```

i em $s0, o endereço base de A em $s3 e B em $s4

Exemplo de código C com carregamento da memória (3)

```
A[i] = B[4]
```

i em $s0, o endereço base de A em $s3 e B em $s4

```
lw $t0, 16($s4) # $t0 = B[4]

sll $t1, $s0, 2 # $t1 = i*4

add $t1, $t1, $s3 # $t1 = &A[i]

sw $t0, 0($t1) # A[i] = $t0
```

Registradores vs Memória

- Registradores são mais rápidos no acesso do que a memória
- Operação na memória de dados requer *loads* e *stores*
  - Mais instruções a serem executadas
- Compilador deve usar registradores para variáveis quando possível
  - Apenas mapear as variáveis menos usadas na memória
  - Otimização de registradores é importante

Linguagem de máquina do MIPS

Instrução	Formato	op	rs	rt	rd	shamt	funct	endereço
`add`	R	0	reg	reg	reg	0	32_{dec}	n.a.
`sub` (subtract)	R	0	reg	reg	reg	0	34_{dec}	n.a.
`add immediate`	I	8_{dec}	reg	reg	n.a.	n.a.	n.a.	constante
`lw` (load word)	I	35_{dec}	reg	reg	n.a.	n.a.	n.a.	endereço
`sw` (store word)	I	43_{dec}	reg	reg	n.a.	n.a.	n.a.	endereço

Exercícios

Capítulo 2 - A linguagem dos computadores

Implementações disponíveis no moodle

Exercícios sugeridos:

2.1, 2.2, 2.3, 2.4, 2.6, 2.8, 2.9, 2.10, 2.11, 2.12, 2.13, 2.14, 2.16, 2.18, 2.21, 2.22

Referências

Materiais disponibilizados pelos professores Lucas Wanner, Paulo Gonçalves, Ricardo Pannain e Rogério Ap. Gonçalves.

Patterson, David A. Hennessy, John L. Organização e Projeto de Computadores. Disponível em: Minha Biblioteca, (5a. edição). Grupo GEN, 2017.

Slides do livro PATTERSON, David A. e HENNESSY, John L. *Computer Organization and Design Risc-V Edition: The Hardware Software Interface*. Estados Unidos, Ed. Morgan Kauffman, 2020.