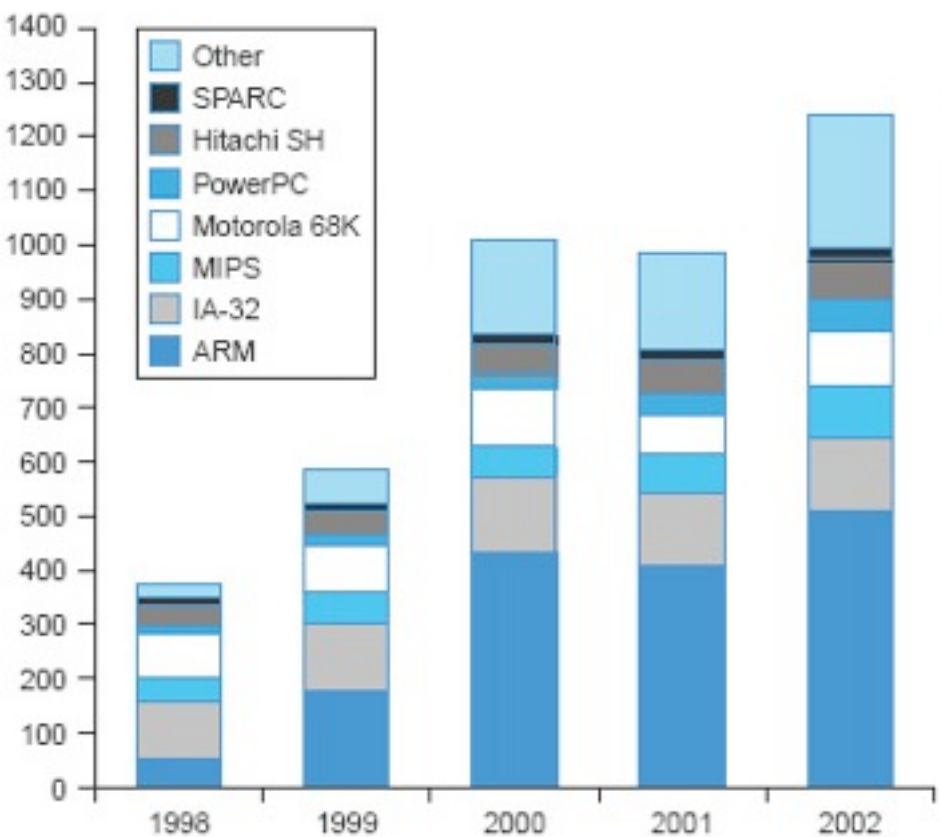
Universidade de Brasília
Departamento de Ciência da Computação

Arquitetura MIPS

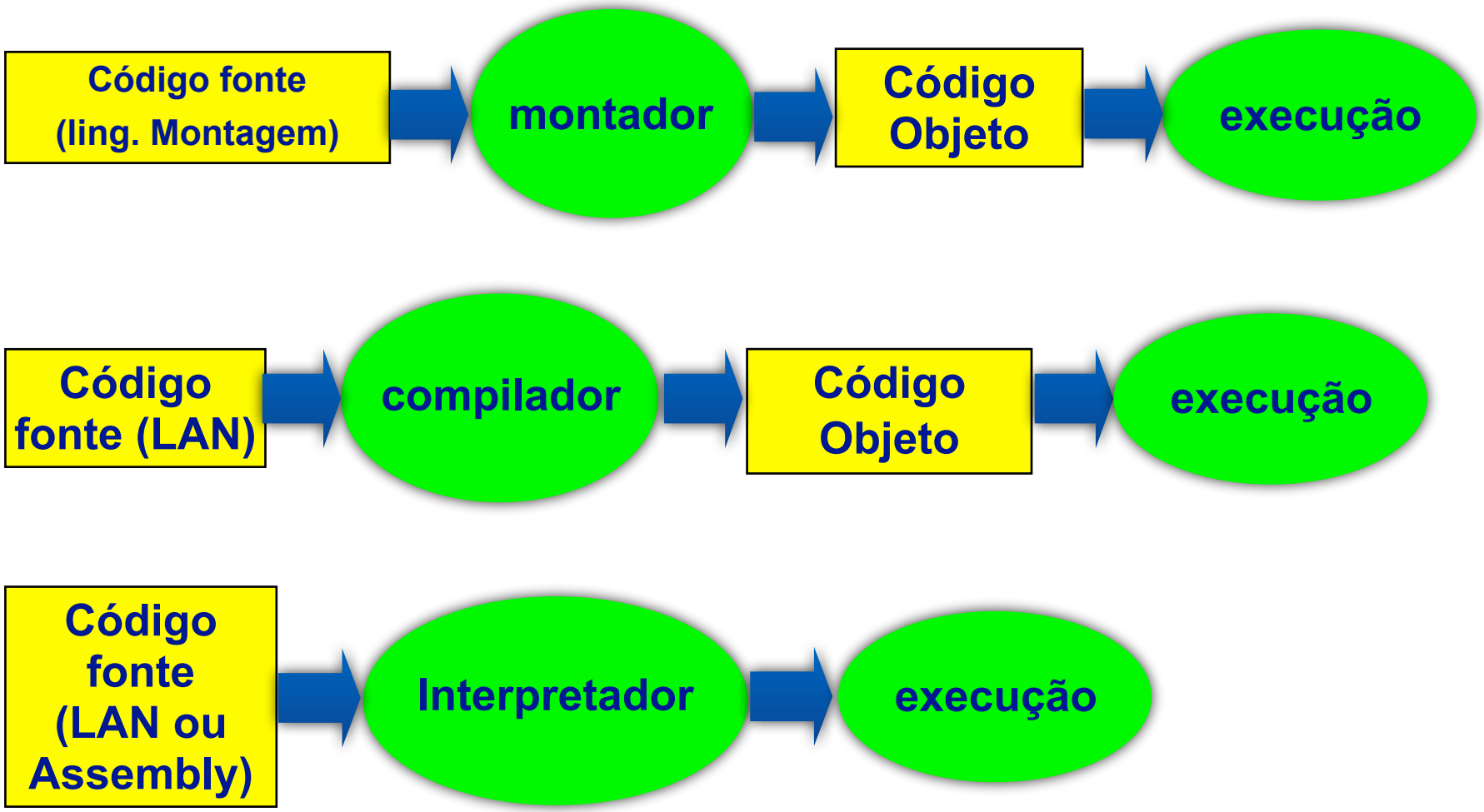
ISA MIPS (*Instruction Set Architecture*)

- Vamos trabalhar com a arquitetura do conjunto de instruções MIPS
  - Semelhante a outras arquiteturas desenvolvidas desde a década de 1980
  - Mais de 350 milhões de processadores MIPS fabricados em 2009
  - Usada pela NEC, Nintendo, Cisco, **Silicon Graphics**, Sony...
  - MIPS 32 e 64 bits incrementados (3D,DSP,etc)!



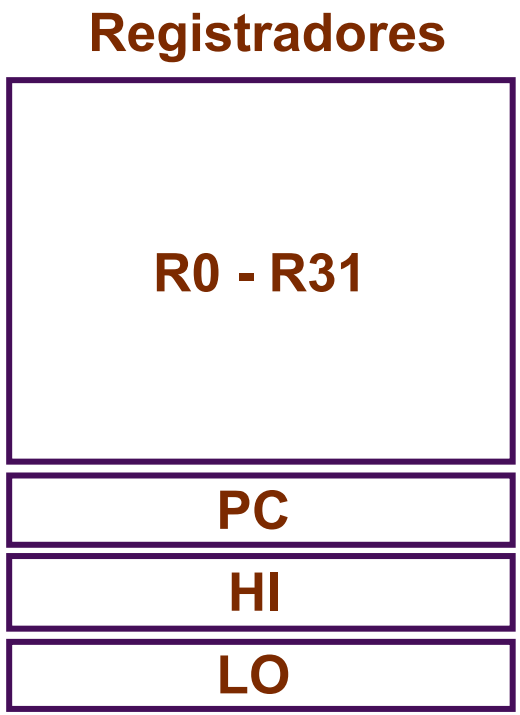
Tradutores

- Montadores, Compiladores e Interpretadores



ISA do MIPS (simplificada)

- Categorias de Instruções:
  - Load/Store
  - Computação
  - Jump e Desvio
  - Ponto Flutuante
  - Gerenciamento de Memória
  - Especial
- 3 Formatos de Instrução: 32 bits



Operações do Hardware

- Todo computador deve ser capaz de realizar operações aritméticas.

ex..: ***add a,b,c***  a = b + c

- Instruções aritméticas no MIPS têm formato fixo, realizando somente uma operação e tendo três “variáveis”

Operações do Hardware

- Todo computador deve ser capaz de realizar operações aritméticas.

ex..: ***add a,b,c***  a = b + c

- Instruções aritméticas no MIPS têm formato fixo, realizando somente uma operação e tendo três “variáveis”

ex: a = b + c + d + e

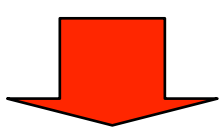
Operações do Hardware

- Todo computador deve ser capaz de realizar operações aritméticas.

ex..: ***add a,b,c*** ⟺ a = b + c

- Instruções aritméticas no MIPS têm formato fixo, realizando somente uma operação e tendo três “variáveis”

ex: a = b + c + d + e



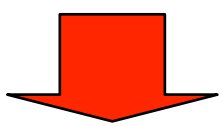
Operações do Hardware

- Todo computador deve ser capaz de realizar operações aritméticas.

ex..: ***add a,b,c*** ⟺ a = b + c

- Instruções aritméticas no MIPS têm formato fixo, realizando somente uma operação e tendo três “variáveis”

ex: a = b + c + d + e



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

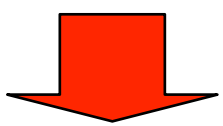
Operações do Hardware

- Todo computador deve ser capaz de realizar operações aritméticas.

ex..: ***add a,b,c*** ⟺ a = b + c

- Instruções aritméticas no MIPS têm formato fixo, realizando somente uma operação e tendo três "variáveis"

ex: a = b + c + d + e



Somente uma instrução por linha

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

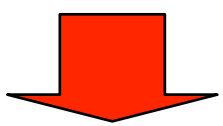
Operações do Hardware

- Todo computador deve ser capaz de realizar operações aritméticas.

ex..: ***add a,b,c*** ⟺ a = b + c

- Instruções aritméticas no MIPS têm formato fixo, realizando somente uma operação e tendo três "variáveis"

ex: a = b + c + d + e



Comentários

Somente uma instrução por linha

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

Operações do Hardware

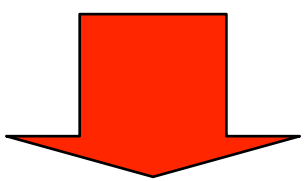
- Exigir que toda instrução tenha exatamente três operandos condiz com a filosofia de manter o hardware simples: hardware para número variável de parâmetros é mais complexo que para número fixo.

Princípio #1 para projetos: Simplicidade favorece a regularidade

Exemplo 1

- Qual o código gerado por um compilador C para o seguinte trecho?

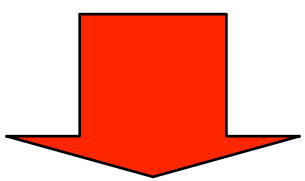
```
a = b + c;
d = a – e;
```



Exemplo 1

- Qual o código gerado por um compilador C para o seguinte trecho?

```
a = b + c;
d = a – e;
```



```
add a, b, c        # a = b + c
sub d, a, e        # d = a - e
```

Exemplo 2

- Qual o código gerado por um compilador C para o seguinte trecho?

```
f = (g + h) – (i + j);
```

Exemplo 2

- Qual o código gerado por um compilador C para o seguinte trecho?

```
f = (g + h) – (i + j);
```

Somente uma operação é feita por instrução: necessidade de variáveis temporárias.

Exemplo 2

- Qual o código gerado por um compilador C para o seguinte trecho?

```
f = (g + h) – (i + j);
```

Somente uma operação é feita por instrução: necessidade de variáveis temporárias.

```
add t0, g, h        # temporário t0 = g + h
add t1, i, j        # temporário t1 = i + j
sub f, t0, t1       #f = (g + h) – (I + j)
```

Operandos e Registradores

- Registradores do MIPS são de 32 bits;
- no MIPS, blocos de 32 bits são chamados de palavra;
- Número de registradores é limitado: MIPS → 32 registradores, numerados de 0 a 31
  - acesso mais rápido, interno ao *chip*
  - fácil acesso

- **Princípio #2 para projetos: menor é mais rápido**
  - Um número muito grande de registradores aumentaria o período de clock.

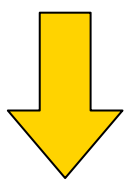
Operandos e Registradores

- No MIPS existe uma convenção para nomear registradores na forma x_i:
  - $s0, $s1, $s2, … para registradores que correspondam a variáveis em C
  - $t0, $t1, $t2, … para registradores temporários necessários para compilar o programa em instruções MIPS

Exemplo 2...

- Considerando a convenção adotada, podemos associar:
  - f => $s0
  - g => $s1
  - h => $s2
  - i => $s3
  - j => $s4

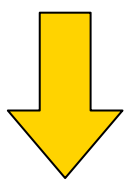
f = (g + h) – (i + j)



Exemplo 2...

- Considerando a convenção adotada, podemos associar:
  - f => $s0
  - g => $s1
  - h => $s2
  - i => $s3
  - j => $s4

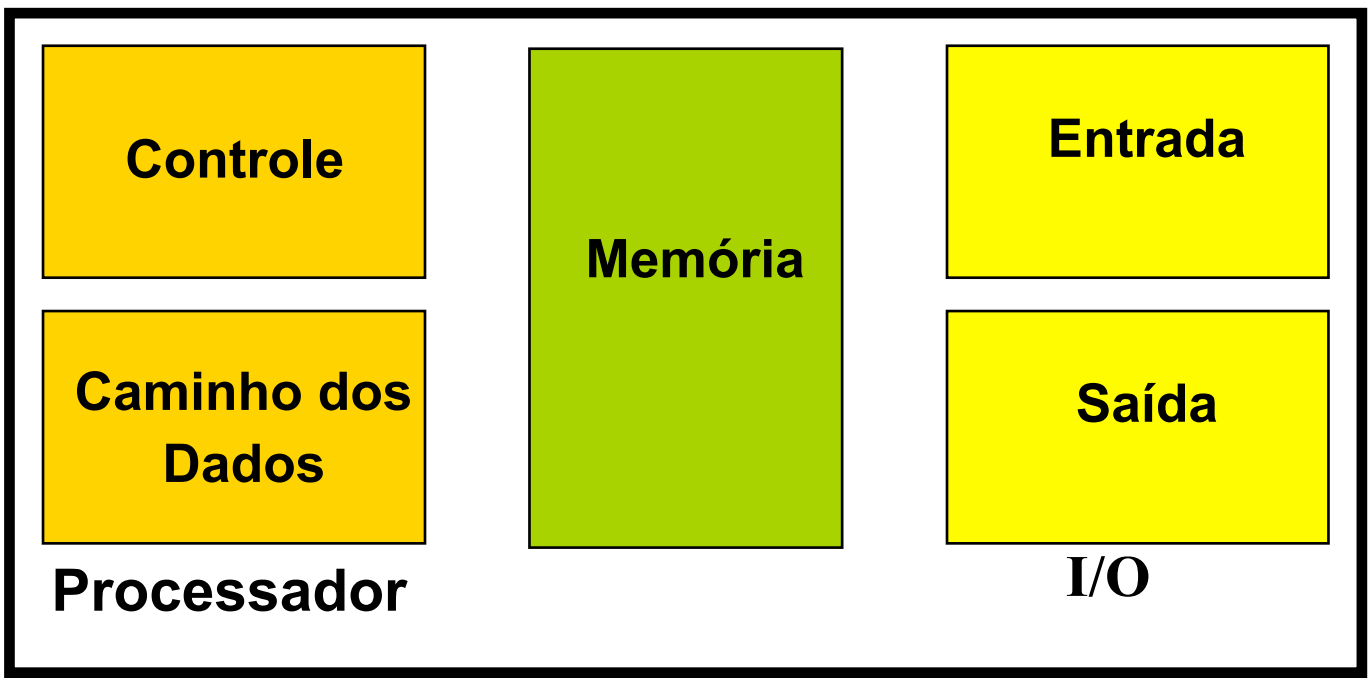
f = (g + h) – (i + j)



```
add   $t0, $s1, $s2    # temporário t0 = g + h
add   $t1, $s3, $s4    # temporário t1 = i + j
sub   $s0, $t0, $t1    # f = (g + h) – (I + j)
```

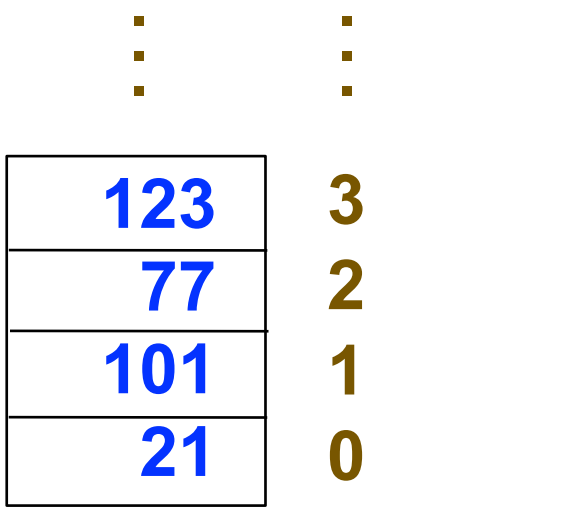
Registradores vs. Memória

- Operandos de instruções aritméticas devem ser registradores (32 registradores disponíveis).
- Compilador associa variáveis a registradores.
- E programas com várias variáveis?



Organização da Memória

- Vista como um grande *array* unidimensional, com endereços seqüenciais, começando em 0
- Um **endereço** de memória é um **índice** no *array*.
- "*Byte addressing*" significa que o índice aponta para um *byte* na memória.



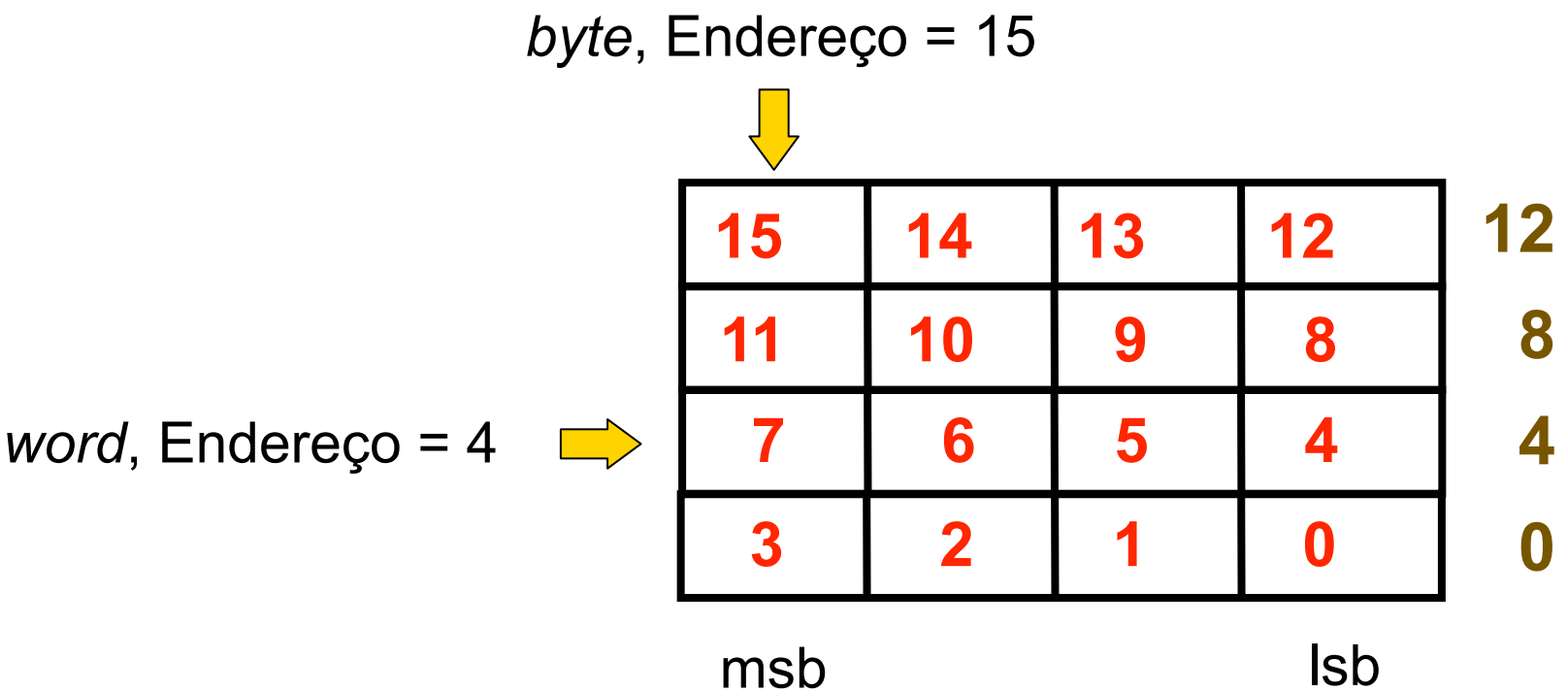
Barramento

Dados Endereços

Processador **Memória**

Organização da Memória

- As palavras de 32 bits são divididas em 4 bytes
- O MIPS pode endereçar um byte ou uma palavra inteira



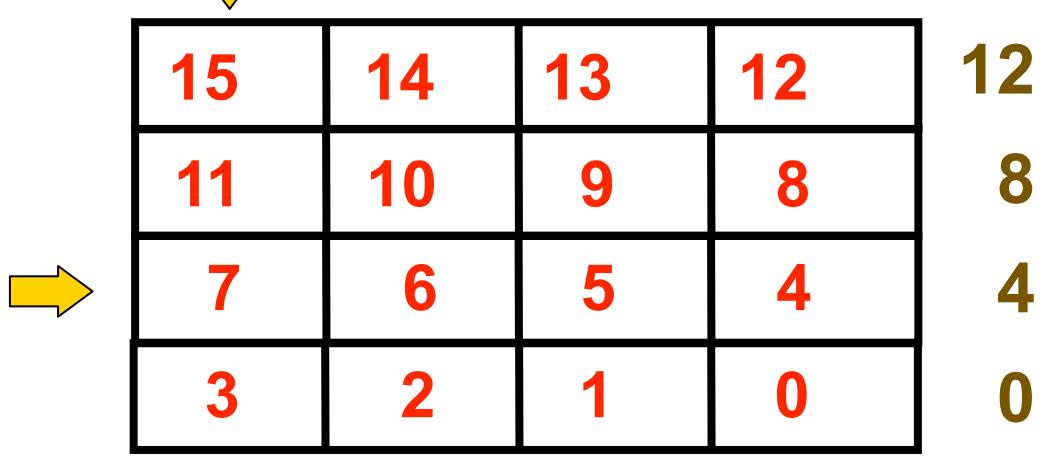
Organização da Memória

- As palavras de 32 bits são divididas em 4 bytes
- O MIPS pode endereçar um byte ou uma palavra inteira

byte, Endereço = 15

Instrução define se é *byte* ou *word*

word, Endereço = 4

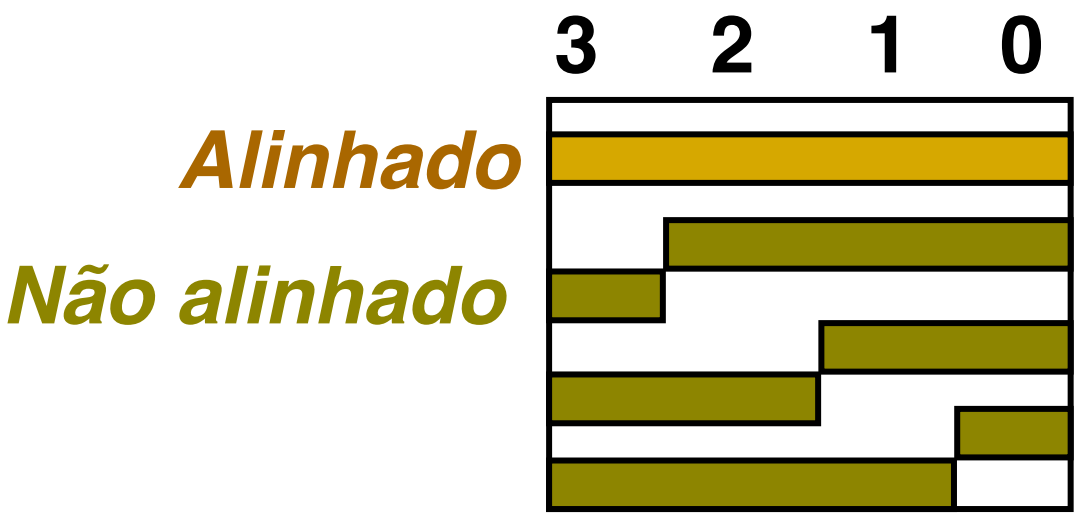


Organização da Memória

2^{32} *bytes* com endereços de *byte* de 0, 1, 2, 3, ... 2^{32}-1

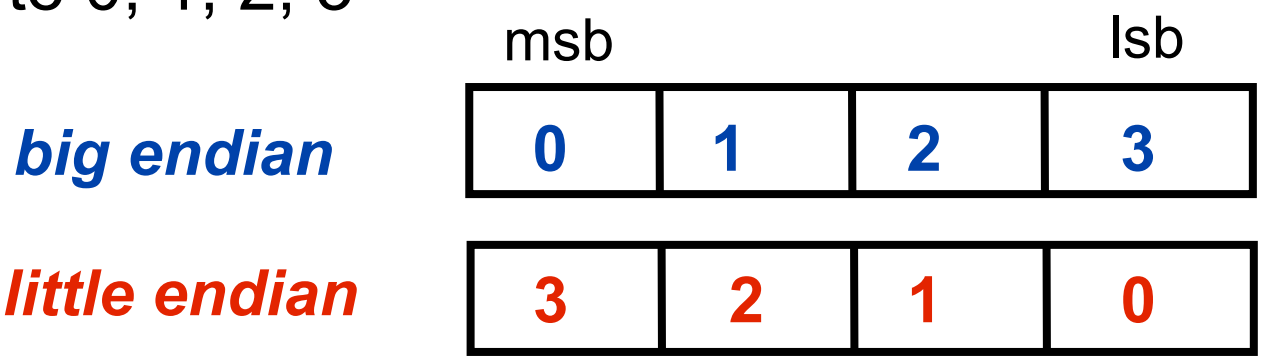
2^{30} *words* com endereços de *byte* de 0, 4, 8, ... 2^{32}-4

Words são **alinhadas**, i.e., quais são os valores dos 2 bits menos significativos do endereço de uma *word*?



Ordenamento dos *Bytes*

- Processadores podem numerar bytes dentro de uma palavra, de tal forma que o byte com o menor número é o mais a esquerda ou o mais a direita. Isto é chamado de byte order.
- Ex: .byte 0, 1, 2, 3



- Big endian: IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: Intel 80x86, MIPS, DEC Vax, DEC Alpha

Transferindo dados da memória

- A instrução de transferência de dados da memória para o registrador é chamada de load.

No MIPS, o nome da instrução é: ***lw*** *(load word)*

- Formato:

lw *registrador destino, constante (registrador base)*

- Ex:

Transferindo dados da memória

- A instrução de transferência de dados da memória para o registrador é chamada de load.

No MIPS, o nome da instrução é: ***lw*** *(load word)*

- Formato:

lw *registrador destino, constante (registrador base)*

- Ex: **g = h + *a;**

Transferindo dados da memória

- A instrução de transferência de dados da memória para o registrador é chamada de load.

No MIPS, o nome da instrução é: ***lw*** *(load word)*

- Formato:

lw *registrador destino, constante (registrador base)*

- Ex: **g = h + *a;**

a => s3, g => s1, h => s2

Transferindo dados da memória

- A instrução de transferência de dados da memória para o registrador é chamada de load.

No MIPS, o nome da instrução é: ***lw*** *(load word)*

- Formato:

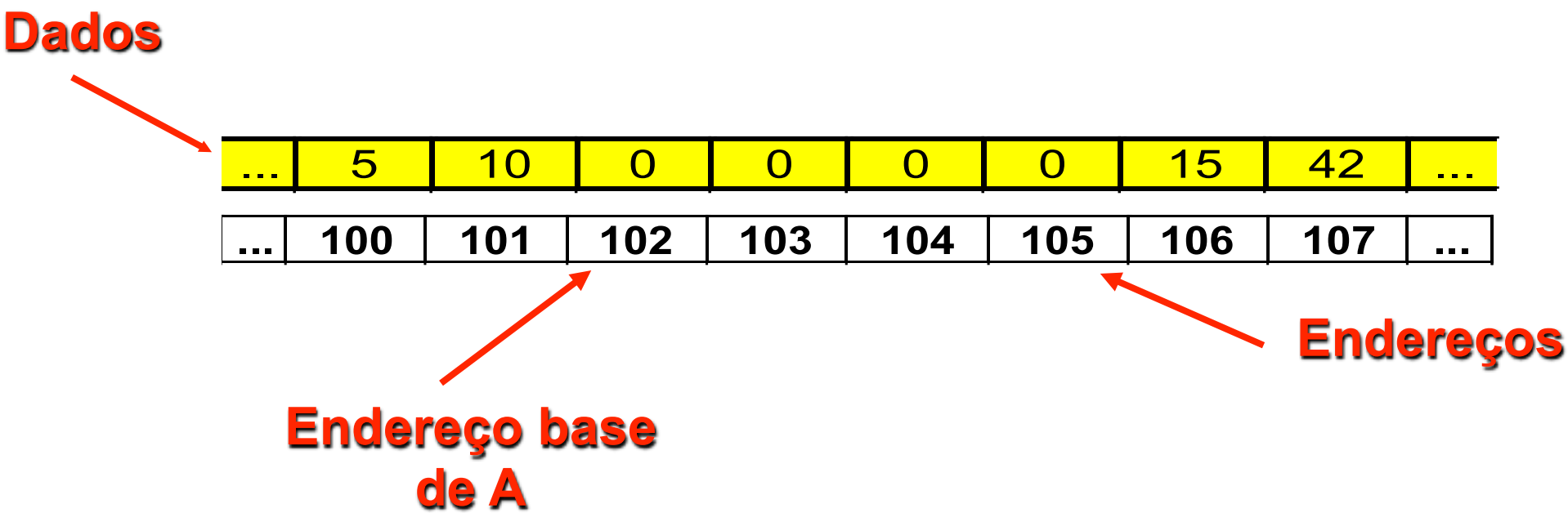
lw *registrador destino, constante (registrador base)*

- Ex: **g = h + *a;**

a => s3, g => s1, h => s2

```
lw      $t0,  0($s3)      # temporário t0 = *a
add     $s1, $s2, $t0     # g = h + *a
```

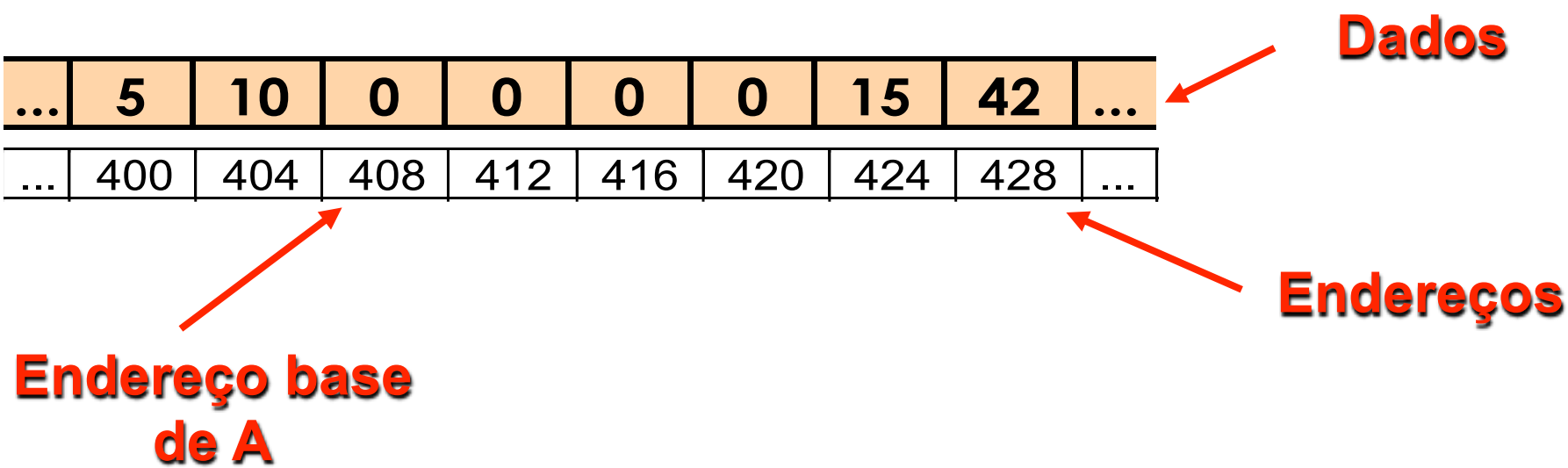
Vetor na Memória



- Vetor A = [0,0,0,0,15], com 5 posições, começando no endereço de memória 102. Este endereço é chamado de endereço base do vetor. Assim, 102 é o endereço de A[0],103 o de A[1], ...,106 o de A[4].

Vetor de *words*

- Cada posição do vetor (de inteiros) é uma palavra, e portanto ocupa 4 bytes



- Vetor A = [0,0,0,0,15], com 5 posições, começando no endereço de memória 408. Assim, 408 é o endereço de A[0],412 o de A[1], 416 o de A[2], 420 o de A[3] e 424 o de A[4].

Exemplo

- Suponha que o vetor A tenha 100 posições, e que o compilador associou a variável *h* ao registrador $s2. Temos ainda que o endereço base do vetor A é dado em $s3. Qual o código para:

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

- A nona posição do vetor A, A[8], está no offset 8 x 4 = 32

```
lw $t0,  32($s3)      # temporário t0 = A[8]
add $t0, $s2, $t0     # temporário t0 = h + A[8]
```

- A décima-terceira posição do vetor A, A[12], está no offset 12 x 4 = 48

```
sw $t0, 48($s3)       # carrega A[12]  em $t0
```

Transferindo dados para a memória

- A instrução de transferência de dados de um registrador para a memória é chamada de store.

No MIPS, o nome da instrução é:
sw (store word)

- Formato:

sw registrador fonte, constante (registrador base)

- Endereço de memória acessado é dado pela soma da constante (*offset*) com o conteúdo do registrador base

Exercício

- Temos ainda que o endereço base do vetor A é dado em $s2, e que as variáveis i e g são dadas em $s0 e $s1, respectivamente. Qual o código para

A[i+g] = g + A[i] – A[0] ?

Organização dos processadores MIPS

Convenção do uso dos Registradores

O registrador **$0** contém sempre o valor **0** (*hardwired*).

Os registradores **$1** (**$at**), **$26** (**$k0**) e **$27** (**$k1**) são reservados para uso do montador e sistema operacional.

Os registradores **$2** e **$3** (**$v0**, **$v1**) são utilizados para retornar valores de funções.

Organização dos processadores MIPS

Os registradores **$4** ... **$7** (**$a0** ... **$a3**) são utilizados para passagem dos primeiros quatro argumentos para sub-programas (os argumentos restantes são passados através da pilha).

Os registradores **$8**...**$15**, **$24**, **$25** (**$t0...$t9**) são *caller-saved* para dados temporários que não necessitam ser preservados durante as chamadas.

Os registradores **$16**...**$23** (**$s0**...**$s7**) são *callee-saved* para dados que necessitam ser preservados durante as chamadas

Organização dos processadores MIPS

O registrador **$28 ($gp)** é um ponteiro global que aponta para o meio de um bloco de memória de 64K, no segmento de dados estáticos.

O registrador **$29** (**$sp**) é o ponteiro de pilha, apontando sempre para o primeiro elemento da pilha.

Organização dos processadores MIPS

O registrador **$30** (**$fp**) é o ponteiro de *frame*. Pode ser utilizado como registrador *callee-saved* **$s8**.

O registrador **$31** (**$ra**) armazena o endereço de retorno quando é executada a instrução **jal**.

Nome	Número	Uso (sugerido)
zero	0	Constante 0
at	1	Reservado para o montador
v0	2	Avaliação de expressão e resultado de função
v1	3	Avaliação de expressão e resultado de função
a0	4	Argumento 1
a1	5	Argumento 2
a2	6	Argumento 3
a3	7	Argumento 4
t0	8	Temporário (não preservado após *call*)
t1	9	Temporário (não preservado após *call*)
t2	10	Temporário (não preservado após *call*)
t3	11	Temporário (não preservado após *call*)
t4	12	Temporário (não preservado após *call*)
t5	13	Temporário (não preservado após *call*)
t6	14	Temporário (não preservado após *call*)
t7	15	Temporário (não preservado após *call*)

Nome	Número	Uso (sugerido)
s0	16	Temporário salvo (preservado após *call*)
s1	17	Temporário salvo (preservado após *call*)
s2	18	Temporário salvo (preservado após *call*)
s3	19	Temporário salvo (preservado após *call*)
s4	20	Temporário salvo (preservado após *call*)
s5	21	Temporário salvo (preservado após *call*)
s6	22	Temporário salvo (preservado após *call*)
s7	23	Temporário salvo (preservado após *call*)
t8	24	Temporário (não preservado após *call*
t9	25	Temporário (não preservado após *call*
k0	26	Reservado para o núcleo do sistema operacional
k1	27	Reservado para o núcleo do sistema operacional
gp	28	Ponteiro para variáveis global
sp	29	Ponteiro de pilha
fp	30	Ponteiro de *frame*
ra	31	Endereço de retorno (utilizado por chamada de sub-programa)