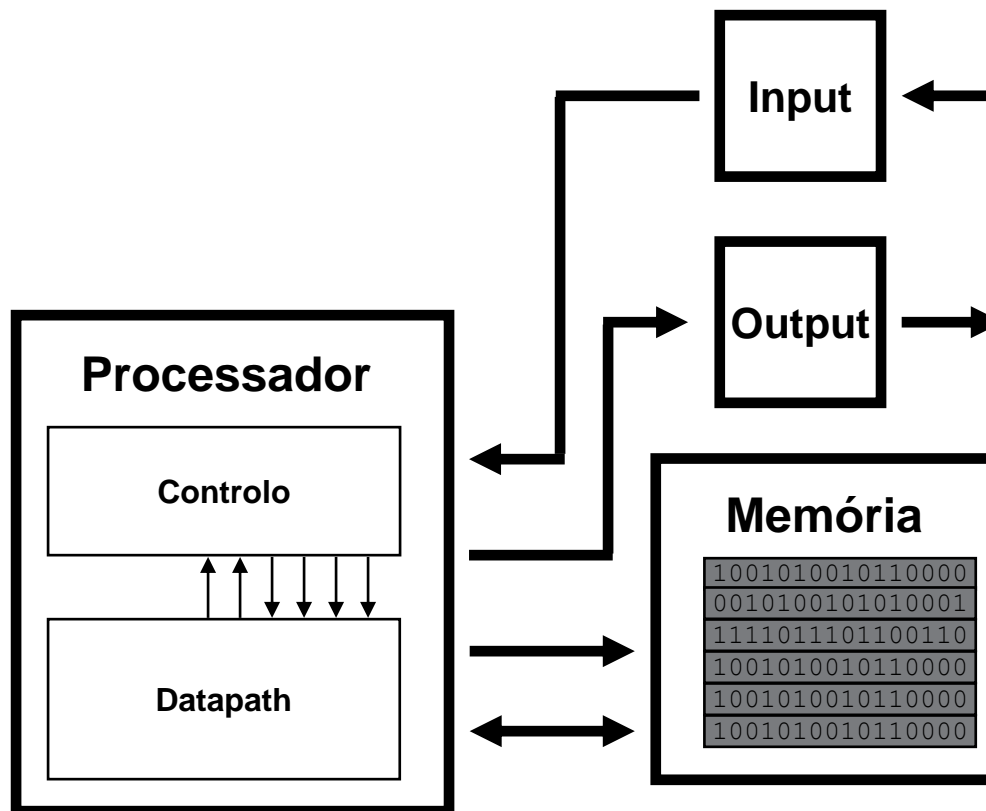


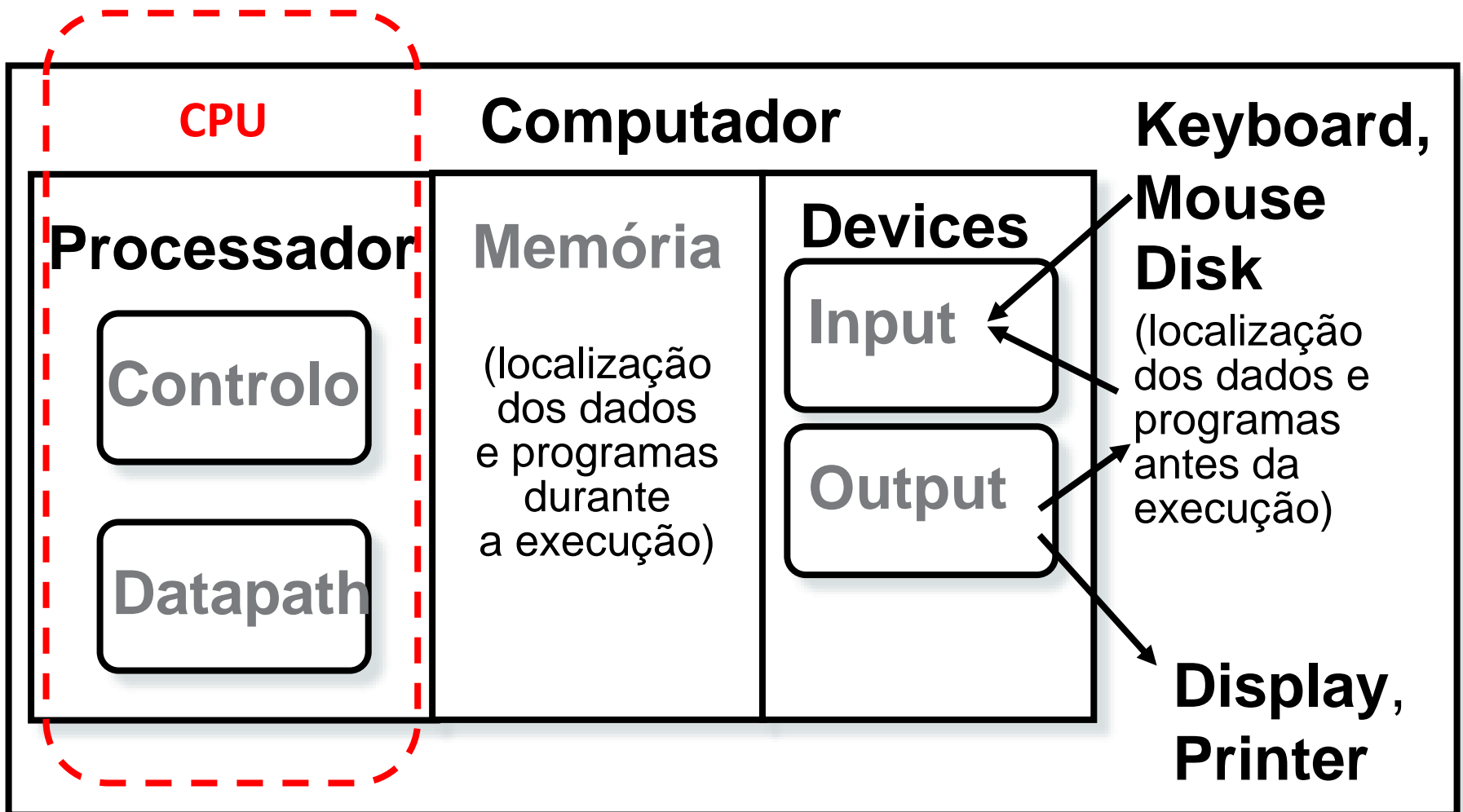
# **Introdução à Arquitetura de Computadores - Conjunto de Instruções -**

**Arquitetura de Computadores 2023/2024**

# Os 5 componentes clássicos de um computador



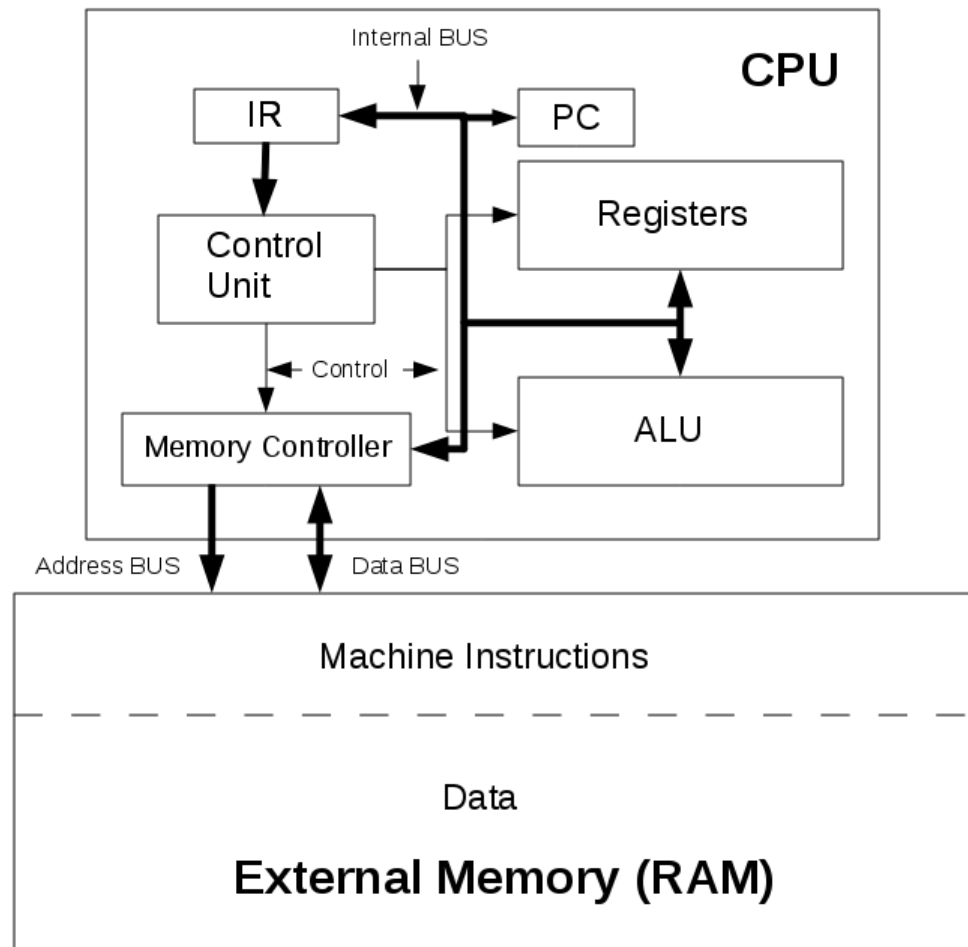
# Os 5 componentes fundamentais



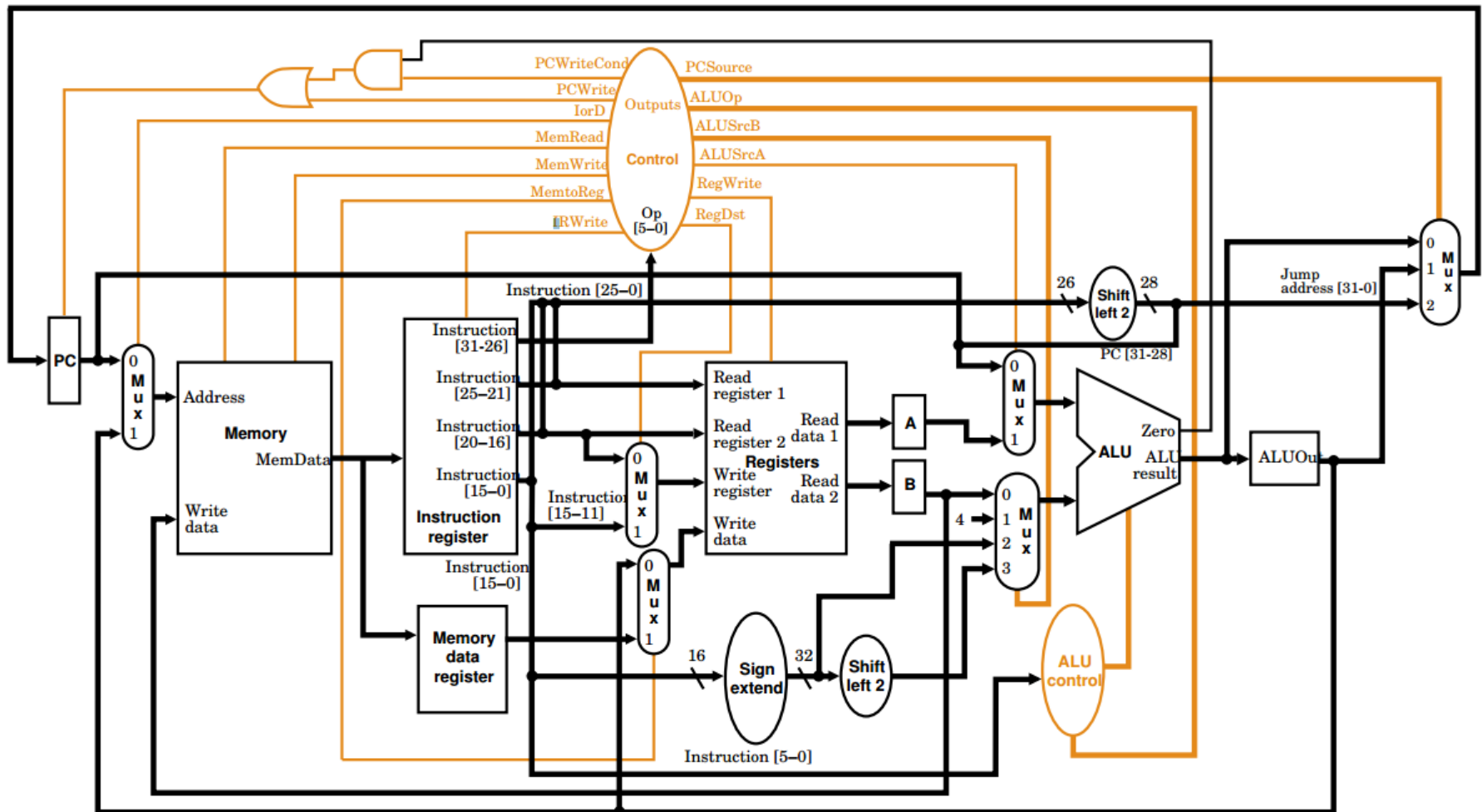
# O CPU

- **Processador (CPU):** a parte activa do computador que faz o trabalho (manipulação de dados e tomada de decisões)
- **Datapath:** parte do processador que contém o hardware necessário ao desempenho de operações (*the brawn / o músculo*)
- **Control:** parte do processador (também em hardware) que diz ao datapath o que é preciso ser feito (*the brain / o cérebro*)

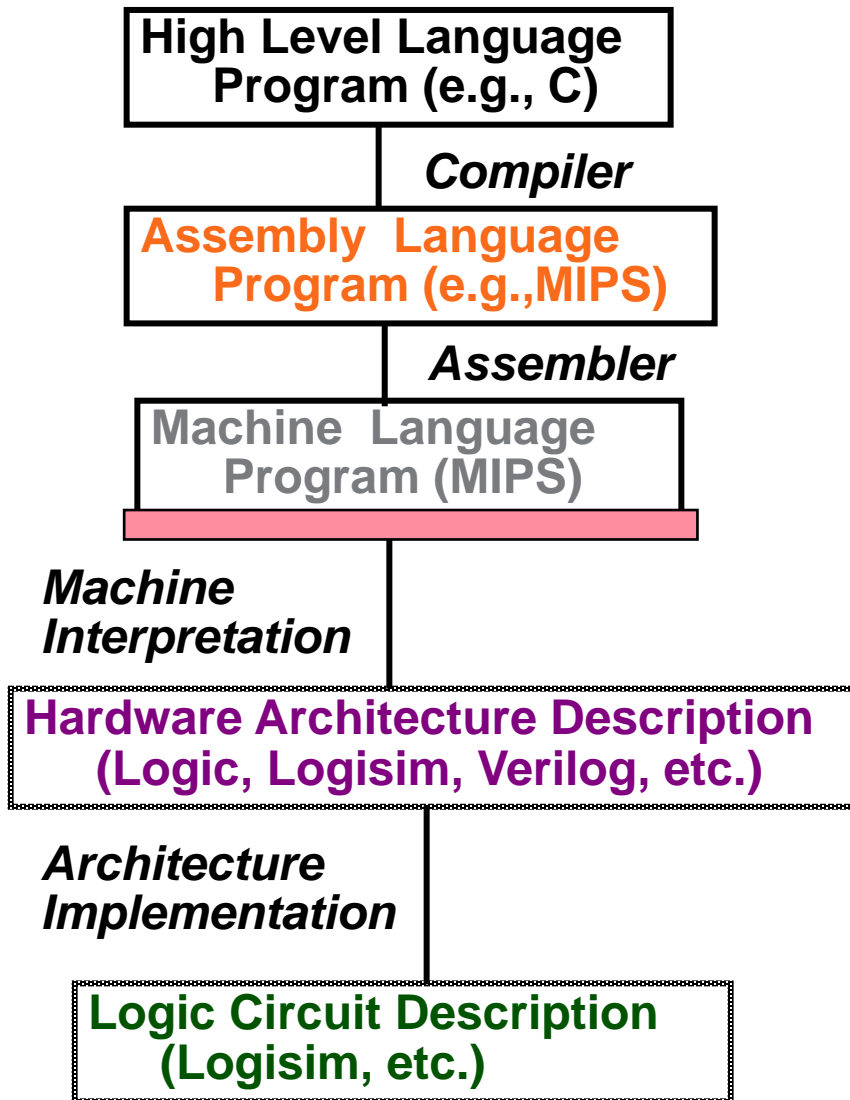
# Arquitectura Interna de uma CPU



# Arquitetura Interna de uma CPU



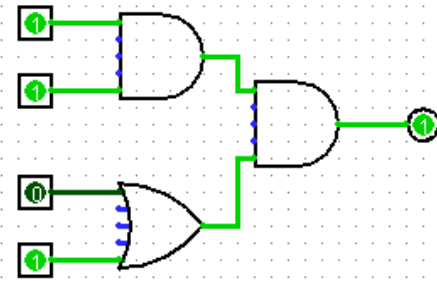
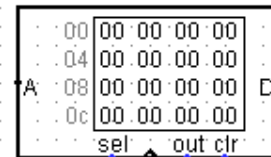
# Recordando...



```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



# Linguagem Assembly

- Tarefa principal da CPU: Executar muitas *instruções*.
- As instruções definem as ações/operações básicas que a CPU é capaz de levar a cabo.
- Diferentes CPUs implementam diferentes conjuntos de instruções. O conjunto de instruções implementado por uma determinada CPU designa-se por *Instruction Set Architecture (ISA)*.
  - Exemplos: Intel 80x86 (Intel Core i7, Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS, Intel IA64, ...



# Instruction Set Architectures

- Inicialmente a filosofia de desenvolvimento consistia em adicionar mais instruções aos novos processadores para realizar tarefas cada vez mais complexas
  - A arquitetura VAX tinha instruções para a multiplicação de polinómios!
  - Estes eram os processadores **CISC (Complex Instruction Set Computer)**
- A partir da década de 80 a filosofia **RISC - Reduced Instruction Set Computer** - começou a impor-se
  - Manter um "instruction set" pequeno e simples facilita o desenho de hardware mais rápido (*smaller is faster*).
  - As operações complicadas são feitas pelo software através da composição de várias instruções simples.

# Arquitetura do MIPS



- MIPS – companhia de semicondutores que construiu uma das primeiras arquiteturas comerciais RISC.
- Nesta disciplina iremos estudar a arquitetura do MIPS em detalhe.
- Porquê o MIPS e não o Intel 80x86?
  - MIPS é simples e elegante. O design da Intel é mais complexo e tortuoso devido à necessidade de manter compatibilidade com versões anteriores (*legacy issues*).
  - MIPS é mais usado que Intel em aplicações embebidas e há mais computadores embebidos que PCs.

Most HP LaserJet workgroup printers are driven by MIPS-based™ 64-bit processors.

# Instruções: Introdução

- Linguagem da máquina
- Mais primitivas do que as das linguagens de alto nível, por exemplo, não existem instruções mais complexas como instruções de control de fluxo como os ciclos «*while*» ou «*for*»
- Conjunto de instruções bastante reduzido
  - Por exemplo a arquitectura MIPS (RISC versus CISC)
- Iremos trabalhar precisamente com o MIPS
  - inspirada na maior parte das arquitecturas desenvolvidas na década de 80
  - Utilizada pela NEC, Nintendo, Silicon Graphics, Sony
  - O nome não quer dizer *millions of instructions per second* !
  - Mas sim: **m**icrocomputer without **i**nterlocked **p**ipeline **s**tages !
- Objectivos de Design: *maximizar o desempenho minimizando os custos e reduzindo o tempo de projecto*

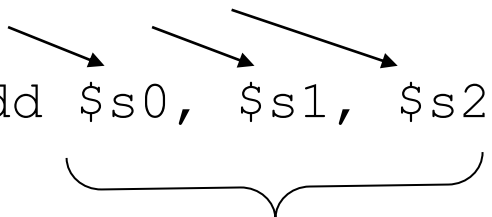
# Instruções MIPS

Registos  
↓  
memória interna do  
microprocessador

- Todas as instruções do MIPS têm 3 operandos
- A ordem dos operandos é fixa (i.e., destino primeiro)
- *Exemplo:*

Linguagem de Alto Nível:  $A = B + C$

Código MIPS equivalente: `add $s0, $s1, $s2`



O trabalho do compilador é  
associar variáveis a registos

# Instruções MIPS

- *Os operandos devem estar nos registos – apenas 32 registos disponíveis (o que requer 5 bits de endereço para seleccionar cada registo). Razões para tão poucos registos:*
- Princípio de Design: *mais pequeno é mais rápido. Porque?*
  - *Os sinais eléctricos têm que viajar mais em chips maiores, o que incrementa o ciclo de relógio necessário.*
  - *Mais pequeno é mais barato!*

# Organização da Memória

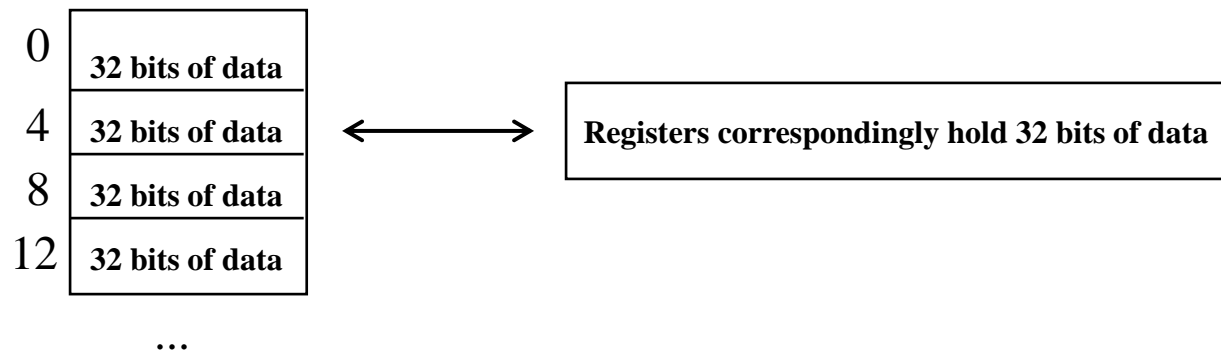
- Vista como uma grande tabela unidimensional com o acesso a ser referenciado por um endereço ou índice
- Um endereço de memória é basicamente um índice na tabela de memória
- O endereçamento é tipicamente ao nível de um *byte*, o que significa que o índice aponta para um *byte* de memória e que a unidade de memória acedida por um carregamento/armazenamento é um *byte*

|   |                |
|---|----------------|
| 0 | 8 bits of data |
| 1 | 8 bits of data |
| 2 | 8 bits of data |
| 3 | 8 bits of data |
| 4 | 8 bits of data |
| 5 | 8 bits of data |
| 6 | 8 bits of data |

...

# Organização da Memória

- Os Bytes são a unidades mínima de carga/armazenamento, mas a maioria dos processadores hoje em dia usa representações de dados utilizando palavras maiores
- No MIPS, uma palavra tem 32 *bits* ou 4 *bytes*.



- $2^{32}$  bytes com endereços entre 0 e  $2^{32}-1$
- $2^{30}$  palavras com endereços entre 0, 4, 8, ...  $2^{32}-4$ 
  - i.e., as palavras estão alinhadas
  - *quais são os 2 bits menos significativos de um endereço de palavra?*

# Introdução ao MIPS

- Variáveis em *Assembly* e Operações Aritméticas -

Arquitetura de Computadores 2023/2024



# "Variáveis" em *Assembly*: Registos (1/3)

- Ao contrário de Linguagens de Alto Nível, como o C e o Python, o assembly não pode usar variáveis
  - Porque não? "*Keep the hardware simple*"
- Os operandos em *assembly* são os registos
  - Pequeno número de locais de armazenamento construídos diretamente em hardware
  - As operações só podem ser realizadas sobre registos!
- Benefício: Como os registos são construídos diretamente em hardware, são muito rápidos (uma mudança num registo é feita em menos de um nano-segundo)

## "Variáveis" em *Assembly*: Registos (2/3)

- Desvantagem: Como os registos são construídos em hardware, existe um número pré-determinado que não pode ser aumentado.
  - Solução: O código do MIPS tem de ser concebido com cuidado de modo a usar eficientemente os recursos disponíveis.
- O MIPS tem 32 registos ... e o x86 ainda tem menos!
  - Porquê 32? **Smaller is faster**
- Os registos no MIPS têm todos 32 bits
  - Os grupos de 32 bits designam-se por word na arquitetura do MIPS
  - Atenção que a dimensão de uma word varia entre diferentes arquiteturas!

# "Variáveis" em *Assembly*: Registos (3/3)

- Os registos estão numerados de 0 a 31
- Os registos tanto podem ser referenciados por um número como por um nome:
  - Referência por número :  
\$0, \$1, \$2, ... \$30, \$31
  - Referência por nome :
    - Semelhante às variáveis em C  
\$16 - \$23 → \$s0 - \$s7
    - Variáveis temporárias  
\$8 - \$15 → \$t0 - \$t7
  - Mais à frente falaremos dos nomes dos 16 registos que faltam.
- Utilize preferencialmente nomes para tornar o seu código mais legível

# Registos MIPS

| Name      | Register number | Usage                                        |
|-----------|-----------------|----------------------------------------------|
| \$zero    | 0               | the constant value 0                         |
| \$at      | 1               | reserved for assembler                       |
| \$v0-\$v1 | 2-3             | values for results and expression evaluation |
| \$a0-\$a3 | 4-7             | arguments                                    |
| \$t0-\$t7 | 8-15            | temporary registers                          |
| \$s0-\$s7 | 16-23           | saved registers                              |
| \$t8-\$t9 | 24-25           | more temporary registers                     |
| \$k0-\$k1 | 26-27           | reserved for Operating System kernel         |
| \$gp      | 28              | global pointer                               |
| \$sp      | 29              | stack pointer                                |
| \$fp      | 30              | frame pointer                                |
| \$ra      | 31              | return address                               |

# QUIZ

Para Pensar:

- Quais serão os programas compilados que ocuparão mais espaço em memória? Os programas para uma arquitetura CISC ou RISC?
- Em que medida o aumento no tamanho das memórias disponíveis terá ajudado à mudança de CISC para RISC?

# C, Java variáveis vs. registos

- Nas linguagens de alto nível como o C, as variáveis têm de ser previamente declaradas como pertencendo a um determinado tipo
  - Exemplo:

```
int fahr, celsius;  
char a, b, c, d, e;
```
- Uma variável só pode representar um valor do tipo declarado (e.g. não podemos misturar e comparar variáveis do tipo `int` e `char`).
- Em assembly os registos não têm um tipo pré-definido. As operações sobre os registos é que vão definir implicitamente o tipo dos dados.

# Comentários em *Assembly*

- Utilizar comentários também ajuda a tornar o código mais legível!
- Em MIPS para comentar uma linha utilize o símbolo cardinal (#)  
`# comentário em assembly`
- Nota: Diferente do C
  - Os comentários em C têm a forma  
`/* comentário em c */`  
e podem conter múltiplas linhas

# Instruções em Assembly

- Em *assembly*, cada linha de código (designada por Instrução), executa uma, e uma só, ação de uma lista de comandos simples pré-estabelecidos
- Ao contrário do que acontece no C, cada linha contém no máximo uma instrução para o processador.
- As instruções em *assembly* são equivalentes às operações (=, +, -, \*, /) em C ou Java.



# Adição e Subtracção no MIPS (1/4)

- Sintaxe:

1 2, 3, 4

Onde :

1) nome da operação

2) operando que recebe o resultado ("*destination*")

3) 1º operando ("*source1*")

4) 2º operando ("*source2*")

- A sintaxe é rígida:

–1 operador + 3 operandos

–Porquê? Regularidade para manter o hardware simples



# Adição e Subtracção no MIPS (2/4)

- Adição em *assembly*

—Exemplo: `add $s0, $s1, $s2` (MIPS)

Equivalente a:  $a = b + c$  (C)

onde os registos do MIPS `$s0`, `$s1`, `$s2` estão associados com as variáveis do C `a`, `b`, `c`

- Subtração em *assembly*

—Exemplo: `sub $s3, $s4, $s5` (MIPS)

Equivalente a:  $d = e - f$  (C)

onde os registos do MIPS `$s3`, `$s4`, `$s5` estão associados com as variáveis do C `d`, `e`, `f`

# Adição e Subtracção no MIPS (3/4)

- Qual é o equivalente à seguinte instrução em C?

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

a → \$s0  
b → \$s1  
c → \$s2  
d → \$s3  
e → \$s4

- Dividir em múltiplas instruções

`add $t0, $s1, $s2 # temp = b + c`

`add $t0, $t0, $s3 # temp = temp + d`

`sub $s0, $t0, $s4 # a = temp - e`

- Nota: Uma única linha em C pode dar origem a várias linhas em assembly do MIPS.
- Nota: Tudo aquilo que estiver depois do cardinal é ignorado (comentários)

# Adição e Subtracção no MIPS (4/4)

- Qual é o equivalente da seguinte instrução?

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

- Temos que utilizar registos temporários

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

# Valores Imediatos (1/2)

- As constantes numéricas designam-se por “*immediatos*”.
- Os “*immediatos*” aparecem frequentemente no código. Sempre que aparecem valores constantes temos que usar instruções específicas (Porquê?)
- Adição com immediatos:  
`addi $s0,$s1,10` (MIPS)  
 $f = g + 10$  (C)  
Onde os registos `$s0`, `$s1` estão associados às variáveis do C `f`, `g`
- Sintaxe semelhante à instrução `add`, excepto no facto de o último argumento ser uma constante em vez de um registo

## Valores Imediatos (2/2)

- Não existe uma instrução no MIPS para subtração com imediatos: Porquê?
- O conjunto de instruções elementares deve ter a menor dimensão possível de forma a simplificar o hardware.
  - Se uma operação pode ser decomposta em instruções mais simples, então não faz sentido incluí-la no "*instruction set*"

– `addi ..., -X` é o mesmo que `subi ..., X` (portanto não há `subi`)

- `addi $s0, $s1, -10` (MIPS)   
 `f = g - 10` (C)   
 onde os registos `$s0`, `$s1` estão associados com as variáveis do C `f`, `g`

codificada numa palavra de 32 bits

# Registo Zero

- O número zero (0) é um "imediato" que aparece muito frequentemente no código.
- Definimos um registo zero (\$0 ou \$zero) para termos o valor 0 sempre à mão; e.g.

add \$s0, \$s1, \$zero (MIPS) <sup>convertida em</sup>  $\Rightarrow$  move \$s0, \$s1  
f = g (C)

onde os registos do MIPS \$s0, \$s1 estão associados com as variáveis do C f, g

- O registo \$zero <sup>sempre 0, imutável</sup> está definido no hardware, e a instrução  
add \$zero, \$zero, \$s0 não faz nada

# Concluindo ...

- Na linguagem assembly do MIPS:
  - Os registos substituem as variáveis em C
  - Existe uma instrução elementar por linha
  - "*Simpler is Better*"
  - "*Smaller is Faster*"

} Paradigma do microprocessador
- Novas instruções que aprendemos:  
add, addi, sub
- Novos registos:  
Variáveis género C: \$s0 - \$s7  
Variáveis temporárias: \$t0 - \$t9  
Zero: \$zero





# Introdução ao MIPS

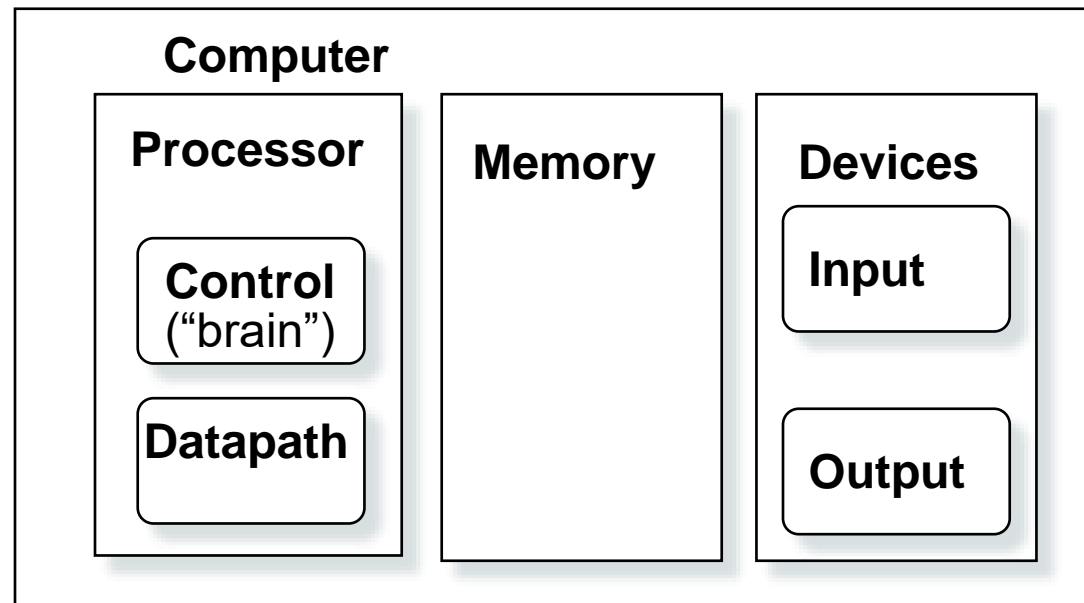
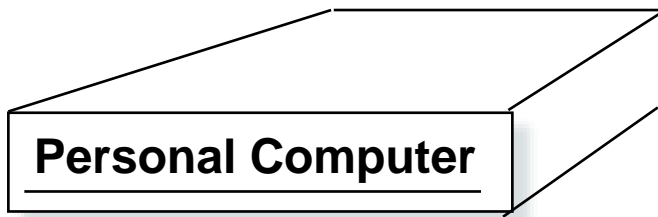
## - Load & Store -

Arquitetura de Computadores 2023/2024

# A Memória

- Até aqui mapeámos as variáveis do C em registos do processador; o que fazer com estruturas de dados de maiores dimensões como as tabelas/arrays?
- As estruturas de dados são guardadas em memória, que é 1 dos 5 componentes fundamentais do computador
- As instruções aritméticas do MIPS só operam sobre registos, e nunca sobre a memória.
- As instruções de transferência de dados permitem transferir dados entre os registos e a memória:
  - Da memória para um registo
  - De um registo para a memória

# Anatomia: os 5 componentes de um Computador

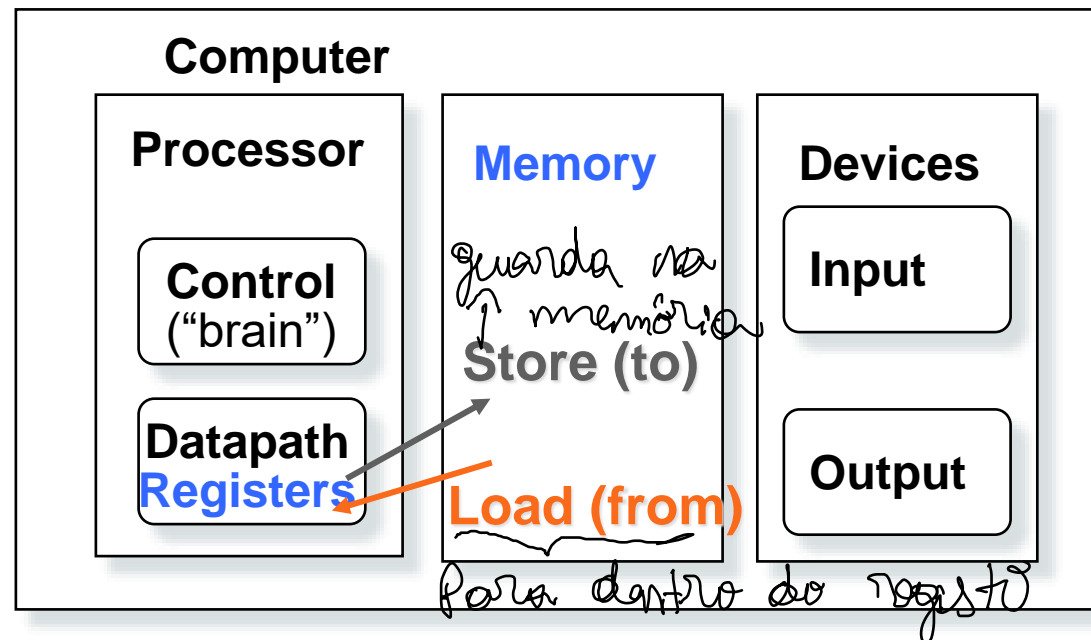


# Anatomia: os 5 componentes de um Computador

- Os registos estão no "*datapath*" do processador.
- Se os operandos estiverem em memória, então:

perspetiva  
do processador

1. Os dados são transferidos para os registos
2. A ação é realizada
3. O resultado é colocado de volta na memória



Estas são as instruções para "data transfer" ...

# ***Data Transfer: Memória para Reg. (1/4)***

- Para transferir uma "word" de dados precisamos de especificar duas coisas:
  - **Registo**: especifica-se usando o # de referência (\$0 - \$31) ou o nome simbólico (\$s0,..., \$t0, ...)
  - **Endereço de memória**: mais difícil
    - Pense na memória como sendo uma grande tabela unidimensional. Cada elemento dessa tabela é referenciado por um ponteiro que corresponde ao endereço de uma célula do array (`char=1 byte`) .
    - Muitas vezes iremos querer incrementar esse ponteiro/endereço
- Lembre-se:
  - **“Load FROM memory”**

# Data Transfer: Memória para Reg. (2/4)

DATA

Dados do programa → Vai para a memória

(A: word 32)

- Para especificar um endereço de memória de onde quer copiar precisa de duas coisas:

Temos então 2 Zonas de

TEXT – Um registo contendo um ponteiro para memória

Um deslocamento (offset) numérico (sempre bytes pois em assembly não existem tipos)

Código

memória

- O endereço de memória pretendido é a soma destes dois elementos.

LA \$s0, A

- Exemplo: LW \$s1, 8(\$t0)

– Especifica o endereço de memória apontado pelo valor no registo \$t0, mais 8 bytes

offset

0x100100



# *Data Transfer: Memória para Reg. (3/4)*

- Sintaxe da instrução Load :

1 2, 3 (4)

Em que

1) nome da operação

2) registo que recebe o valor

3) deslocamento em bytes (offset)

4) registo contendo o endereço base (ponteiro) para a memória

- Nome da Operação:

–  $lw$  (que significa Load Word, ou seja transferir 32 bits (1 word) de cada vez)

# *Data Transfer: Memória para Reg. (4/4)*



- Exemplo: `lw $t0, 12($s0)`

Esta instrução agarra no valor que está no registo `$s0` (ponteiro base), adiciona-lhe um deslocamento de 12 bytes para obter o endereço de memória, e transfere para `$t0` o conteúdo das 4 células de memória apontadas por esse endereço.

- Notas:
  - `$s0` é chamado o registo base
  - 12 é chamado o offset
  - O offset é geralmente usado para aceder aos elementos de um array ou estrutura: o registo base aponta para o início desse array ou estrutura (nota: o offset é sempre uma constante).



# *Data Transfer: Registo para Memória*

- Queremos agora transferir do registo para a memória
  - A instrução store tem uma sintaxe semelhante ao load

- MIPS Instruction Name:

**sw** (significa Store Word, ou seja, transferir 32 bits (1 word) de cada vez)



- Exemplo: **sw \$t0, 12(\$s0)**

Esta instrução agarra no ponteiro em `$s0`, adiciona-lhe 10 bytes, e depois guarda o valor do registo `$t0` no endereço de memória assim calculado

- Lembre-se: “Store INTO memory”

# Endereçamento: Byte vs. word

- Todas as *words* em memória têm um endereço.
- Os primeiros computadores referenciavam as words da mesma forma que o C numera elementos num array:

– `Memory[0], Memory[1], Memory[2], ...`

← ↑ →  
“**endereço**” de uma word

- ◆ No entanto os computadores precisam de referenciar simultaneamente bytes e words (4 bytes/word)
- ◆ Hoje em dia todas as arquiteturas endereçam a memória em bytes (i.e., “**Byte Addressed**”). Assim para aceder a words de 32-bits os endereços têm de dar saltos de 4 bytes
  - `Memory[0], Memory[4], Memory[8], ...`

# Compilação de Acessos à Memória

- Qual o offset que devemos usar com `lw` para aceder a `A[5]`, sendo `A` uma tabela de `int` em C?

- ❖ Para seleccionar `A[5]` temos que  $4 \times 5 = 20$ : byte vs. word

- Desafio: Compile a instrução à mão usando registos:

- ❖ `g = h + A[5]` com `g`: `$s1`, `h`: `$s2`, endereço base de `A`: `$s3`

- ❖ Transfira da memória para o registo:

```
lw $t0, 20($s3)      # $t0 gets A[5]
```

- Adicione 20 a `$s3` para seleccionar `A[5]` e coloque em `$t0`

- ❖ Adicione o resultado a `h` e coloque em `g`

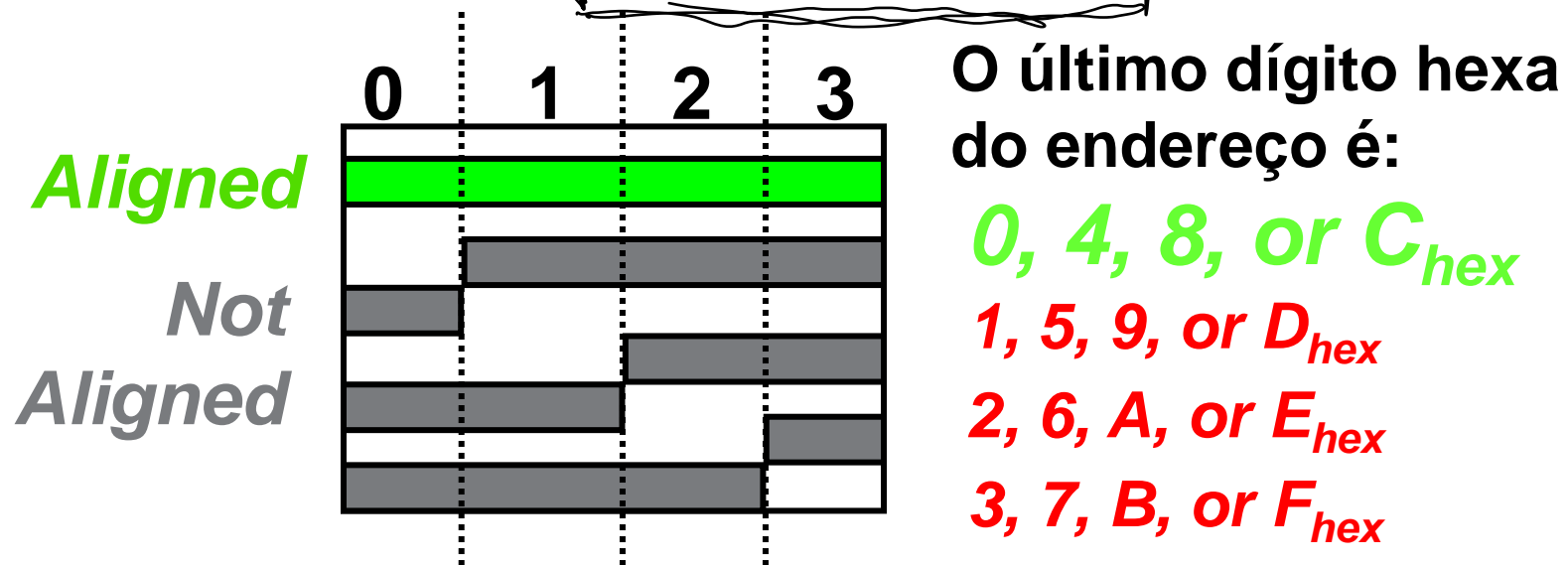
```
add $s1, $s2, $t0      # $s1 = h+A[5]
```

# Notas sobre a memória

- Erro Frequente: Esquecermo-nos que os endereços de words sucessivas numa máquina com “Byte Addressing” diferem em mais do que 1.
  - Muitos programadores de *assembly* cometem erros por assumirem que o endereço da próxima word pode ser obtido incrementando o registo em 1 unidade em vez de adicionarem o número de bytes da word (diferente do C).
  - Ao contrário do que acontece no C, em *assembly* não existe a noção de tipo, e é impossível o computador saber o tamanho de uma word fazendo o ajuste implícito do incremento dos ponteiros.
  - Lembre-se também que no `lw` e `sw`, a soma do endereço de base com o offset deve ser sempre um múltiplo de 4 ( **word aligned memory** )

# Alinhamento de Memória

- No MIPS as words e objetos são guardados em memória em bytes cujo endereço é sempre múltiplo de 4.



- ◆ **Alinhamento de Memória:** os objetos começam sempre em endereços que são múltiplos do seu tamanho

■ Pode gerar um "Bus Error"!  $\Rightarrow$  processador gera uma exceção

# Registos vs Memória

- O que acontece se houver mais variáveis do que registos?
  - O compilador tenta manter as variáveis mais utilizadas nos registos
  - As variáveis menos usadas são armazenadas em memória: spilling
  - Consulte o comando `register` do C
- Porque não manter todas as variáveis em memória?
  - Smaller is faster: os registos são mais rápidos do que a memória
  - Os registos são mais versáteis:
    - Cada instrução aritmética do MIPS pode ler 2 registos, fazer uma operação sobre os dados e escrever o resultado num registo
    - Uma instrução de transferência de dados só pode ler ou escrever 1 operando.

# Leitura e escrita de bytes (1/2)

- Para além da transferência de “words” (4 bytes usando `lw` e `sw`), o MIPS permite também a transferência de bytes:
  - load byte: `lb`
  - store byte: `sb`

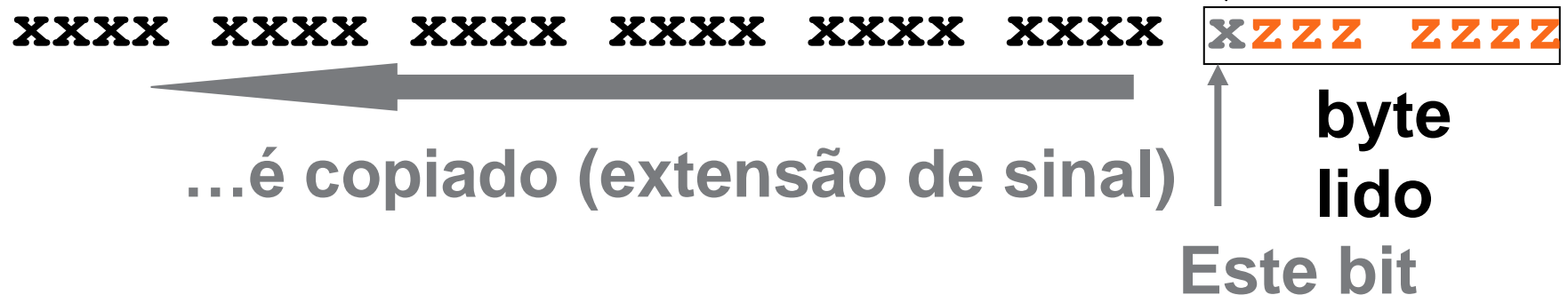
- O formato das instruções é semelhante ao `lw`, `sw`

E.g., `lb $s0, (3)($s1)` → não há alinhamento de memória (é 1 byte)  
 o byte de memória com endereço = “3” + “conteúdo do registo `s1`” é copiado para o byte menos significativo do registo `s0`.

# Leitura e escrita de bytes (2/2)

- O que é que acontece com os outros 24 bits do registo de 32 bits?

— **lb**: extensão de sinal para preencher os 24 bits mais significativos (relembrar que a representação em complementos de 2 assume um número fixo de bits)



- No caso de leitura de “chars” nós não queremos que haja extensão de sinal!

- Neste caso devemos usar a seguinte instrução

load byte unsigned: **lbu**

→ Usar chars / strings → Não há extensão de sinal



# Leitura e escrita de half-words (16 bits)

```
lh $s0, 10($s1)
```

Carrega dois bytes do endereço dado por (\$s1+10) para o registo \$s0. O endereço tem que ser múltiplo de 2 para evitar problemas de alinhamento.

O sinal é estendido como no caso do `lb`!



Existe também uma versão sem sinal:

```
lhu $s0, 10($s1)
```

Neste caso não há lugar a extensão do sinal.

# Inicialização de Endereços

- Para inicializar o conteúdo dos registos com o valor de um endereço, utilizamos a instrução **la \$rd,address**:

```
.data
```

```
year: .word 2021
```

```
.text
```

```
la $t1, year
```

```
lw $t2, 0($t1)
```

## E concluindo ...

- A memória é endereçada em **bytes**, mas as instruções `lw` e `sw` acedem a uma **word (4 bytes)** de cada vez.
- Um ponteiro (usado em `lw` e `sw`) é só um endereço de memórias. Podemos adicionar ou subtrair valores ao endereço base (usando o *offset*).
- Novas instruções que vimos:  
`lw, sw`