

# Aula 1

- Conceitos fundamentais em Arquitetura de Computadores
  - Arquitetura básica de um sistema computacional. Arquitetura básica do CPU
  - O ciclo de execução de uma instrução
  - Níveis de representação. Codificação de instruções
  - *Instruction Set Architecture* (ISA). Classes de instruções
- Princípios básicos de projeto de uma arquitetura
- Aspectos chave da arquitetura MIPS
  - Instruções aritméticas
  - Instruções lógicas e de deslocamento
  - Codificação de instruções no MIPS: formato R

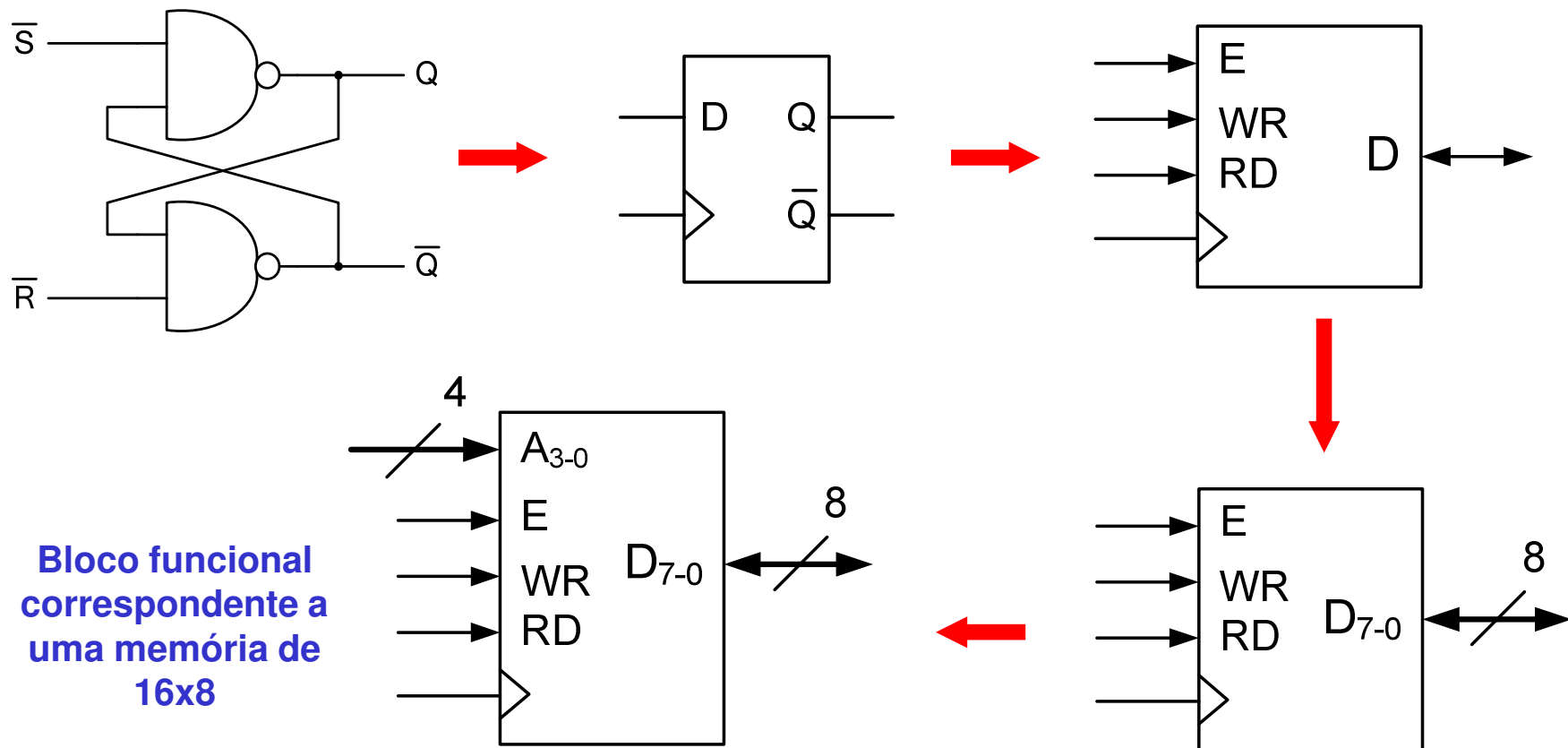
Bernardo Cunha, José Luís Azevedo

# Arquitetura de Computadores e Sistemas Digitais

- Arquitetura de Computadores é uma das áreas de aplicação direta dos conceitos, técnicas e metodologias aprendidas nas duas UCs de Sistemas Digitais
- Em Arquitetura de Computadores, contudo, trabalha-se num nível de abstração diferente
- Recorre-se, na maior parte das vezes, a **blocos funcionais complexos** com cuja síntese, normalmente, não temos que nos preocupar (isso não significa que a sua funcionalidade não tenha que ser totalmente compreendida)

# Exemplo: memória RAM 16x8

- Por exemplo, uma "Memória" (um dispositivo com capacidade para armazenar informação digital binária) pode ser construída à custa de blocos básicos bem conhecidos dos sistemas digitais: **flip-flops**



# A máquina e a sua linguagem

- Princípios básicos dos computadores atuais:
  - As instruções são representadas da mesma forma que os números
  - Os programas são armazenados em memória, para serem lidos e escritos, tal como os números
- Estes princípios formam os fundamentos do conceito da arquitetura **"stored-program"**
  - O conceito "stored-program" implica que na memória possa residir, ao mesmo tempo, informação de natureza tão variada como: o código fonte de um programa em C, um editor de texto, um compilador, e o próprio programa resultante da compilação

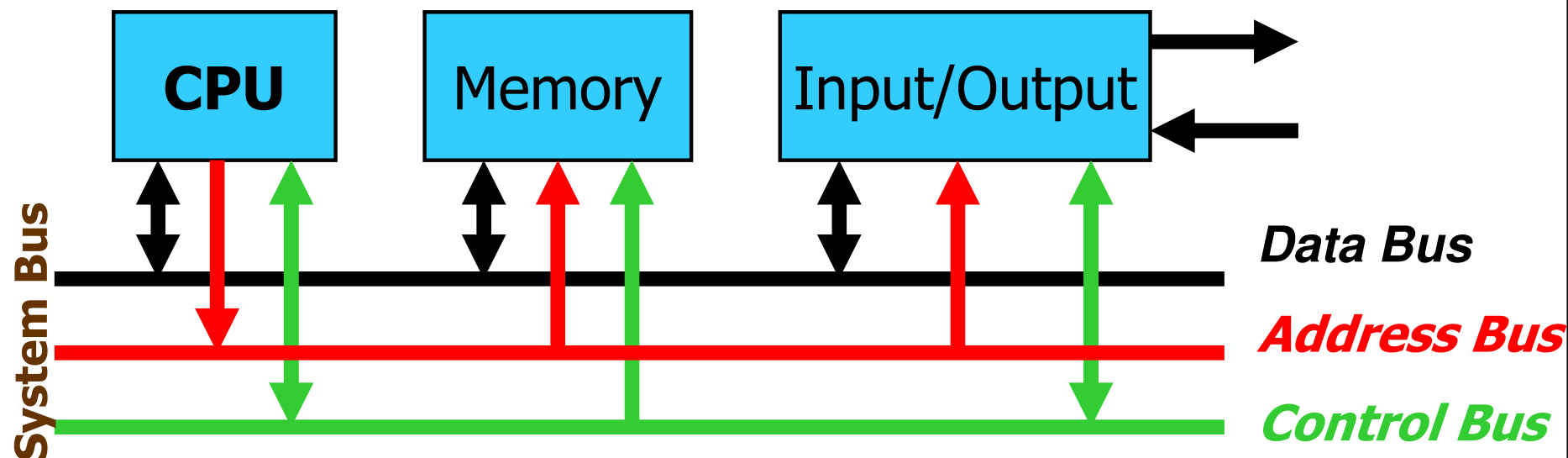
# Arquitetura básica de um sistema computacional

- Unidades fundamentais que constituem um computador
  - **CPU** – responsável pelo processamento da informação através da execução de uma sequência de instruções (programa) armazenadas em memória
  - **Memória** – responsável pelo armazenamento de:
    - Programas
    - Dados para processamento
    - Resultados
  - **Unidades de I/O** – responsáveis pela comunicação com o exterior
    - **Unidades de entrada** – permitem a receção de informação vinda do exterior (dados, programas) e que é armazenada em memória
    - **Unidades de saída** – permitem o envio de resultados para o exterior
- Um computador é um sistema digital complexo

**Cada um destes blocos é um sistema digital!**

# Arquitetura básica de um sistema computacional

- Modelo de von Neumann



- **Data Bus:** barramento de transferência de informação (CPU↔memória, CPU↔Input/Output)
- **Address Bus:** identifica a origem/destino da informação (na memória ou nas unidades de input/output)
- **Control Bus:** sinais de protocolo que especificam o modo como a transferência de informação deve ser feita

# Arquitetura básica de um sistema computacional

- **Endereço** (*address*) – um número (único) que identifica cada registo de memória. Os endereços são contados sequencialmente, começando em 0
  - Exemplo: o conteúdo da posição de memória 0x2000 é 0x32 – (0x2000 é o endereço, 0x32 o valor armazenado)
- **Espaço de endereçamento** (*address space*) – a gama total de endereços que o CPU consegue referenciar (depende da dimensão do barramento de endereços)
  - Exemplo: um CPU com um barramento de endereços de 16 bits pode gerar endereços na gama: 0x0000 a 0xFFFF (i.e., 0 a  $2^{16}-1$ )
  - Qual o espaço de endereçamento de um processador com um barramento de endereços de 32 bits?

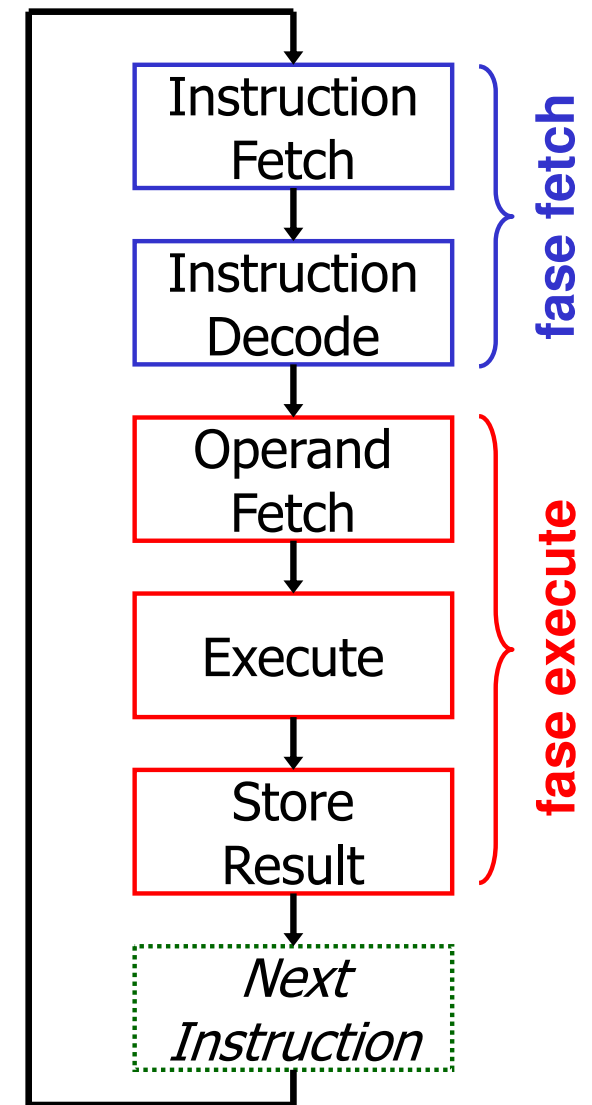
# Arquitetura básica do CPU

- **Secção de dados** (*datapath*) – elementos operativos / funcionais para encaminhamento, processamento e armazenamento de informação
  - Multiplexers
  - Unidade Aritmética e Lógica (ALU) – Add, Sub, And, Or...
  - Registos internos
- **Unidade de controlo** – responsável pela coordenação dos elementos do *datapath*, durante a execução de um programa
  - Gera os sinais de controlo que adequam a operação de cada um dos recursos da secção de dados às necessidades da instrução que estiver a ser executada
  - Dependendo da arquitetura, pode ser uma máquina de estados ou um elemento meramente combinatório
- Independentemente da Unidade de Controlo ser combinatória ou sequencial, **o CPU é sempre uma máquina de estados síncrona**



# Ciclo-base de execução de uma instrução

- **Instruction fetch:** leitura do código máquina da instrução (instrução reside em memória)
- **Instruction decode:** descodificação da instrução pela unidade de controlo
- **Operand fetch:** leitura do(s) operando(s)
- **Execute:** execução da operação especificada pela instrução
- **Store result:** armazenamento do resultado da operação no destino especificado na instrução



# Níveis de Representação

**High-level language**  
program (in C)

```
unsigned char toUpper(unsigned char c)
{
    if(c >= 'a' && c <= 'z')
        return (c - 0x20);
    else
        return c;
}
```

↓  
Compiler

**Assembly language**  
program (for MIPS)

```
toUpper:    addiu $5,$4,-97
            sltiu $1,$5,26
            or    $2,$4,$0
            beq   $1,$0,else
            addiu $2,$4,-32
else:       jr    $31
```

↓  
Assembler

**Binary machine language**  
program (for MIPS)

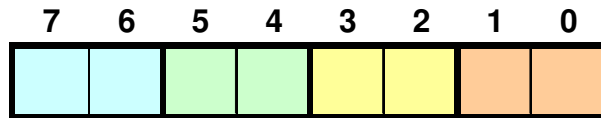
00100100100001001111111110011111	(0x2485ff9f)
00101100101000010000000000011010	(0x2ca1001a)
00000000100000000001000000100101	(0x00801025)
00010000001000000000000000000001	(0x10200001)
00100100100000101111111111000000	(0x2482ffe0)
00000011111000000000000000001000	(0x03e00008)

# Codificação das instruções

- A **codificação de uma instrução**, sob a forma de um número expresso em binário, terá que ter toda a informação de que o CPU necessita para a sua execução
- Qual a operação a realizar?
- Qual a localização dos operandos (se existirem)?
  - podem estar em **registos internos do CPU** ou na **memória externa**. No 1º caso deverá ser especificado o número de um registo; no 2º um endereço de memória
- Onde colocar o resultado?
  - **Registos internos / memória**
- Qual a próxima instrução a executar?
  - em condições normais é a instrução seguinte na sequência e, portanto, não é, normalmente, explicitamente mencionada
  - em instruções que **alteram a sequência de execução** a instrução deverá fornecer o endereço da próxima instrução a ser executada

# Exemplo - CPU hipotético

## Formato de codificação das instruções (8 bits)



Formato 1

**Oper.**   **Rdest**   **Rop1**   **Rop2**



Formato 2

**Oper.**   **Reg.**   **End. Memória**

## Registos Internos do CPU:

00 Reg. 0

01 Reg. 1

10 Reg. 2

11 Reg. 3

## Operações possíveis:

00 Somar o conteúdo de dois registos

01 Ler da memória para um registo interno do CPU (LOAD)

10 Escrever o conteúdo de um registo interno na memória (STORE)

11 Não definida (N.D.)

## Exemplo de programa em código máquina para este processador

**Hex**   **Binary**

0x58   01011000   Ler o conteúdo da posição de memória 8 para o registo interno 1

0x79   01111001   Ler o conteúdo da posição de memória 9 para o registo interno 3

0x15   00010101   Somar o conteúdo do reg. 1 c/ o reg. 1 e depositar o result. no reg. 1

0x07   00000111   Somar o conteúdo do reg. 1 c/ o reg. 3 e depositar o result. no reg. 0

0x8A   10001010   Escrever o conteúdo do reg. 0 na posição de memória 10

Qual é a expressão aritmética implementada neste programa?

# Arquitetura do Conjunto de Instruções (ISA)

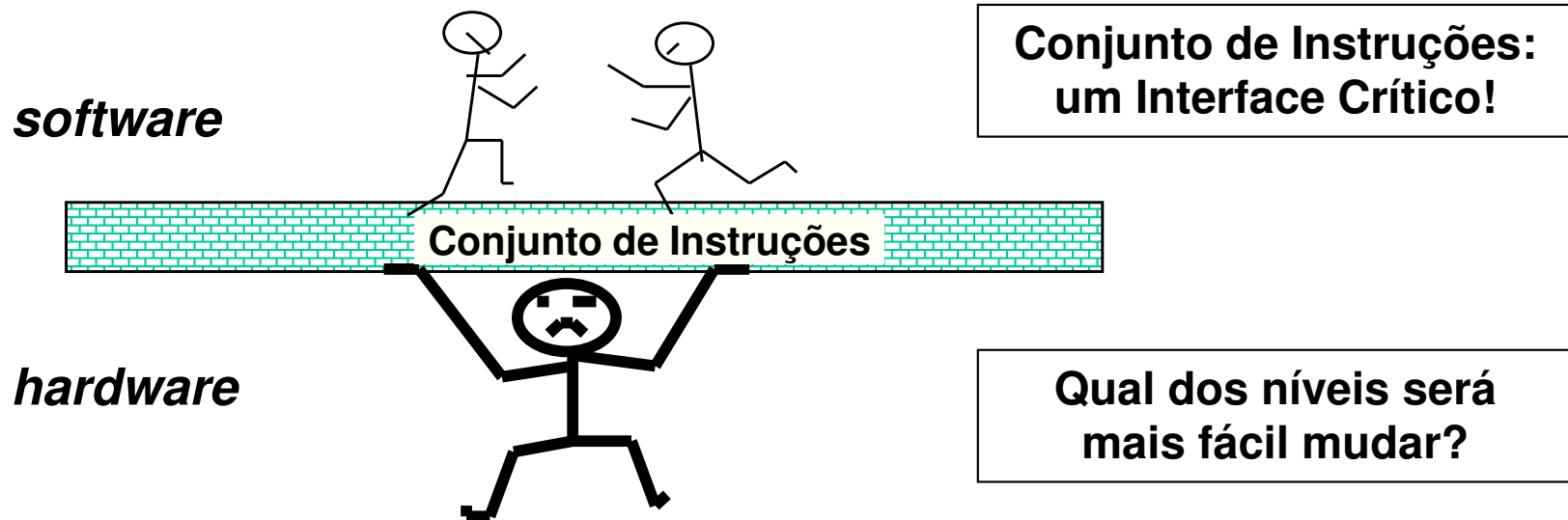
- **Instruction Set Architecture (ISA)**

- Também designada por modelo de programação
- **Atributos do sistema como vistos pelo programador:** estrutura concetual e comportamento funcional (Amdahl, Blaaw & Brooks, 1964)
- Abstração que descreve a **interface entre o nível mais básico de software e o hardware**
- Permite distinguir entre arquitetura (o que o programador vê) e implementação (como é realizado em hardware)
- Exemplo: processadores AMD compatíveis com Intel x86 (mesma ISA, implementações diferentes)

- **Instruction Set:** conjunto completo de operações/instruções que o processador pode realizar (parte integrante da ISA)
- Arquitetura de Computadores = ISA + Organização (como é realizado em hardware)

# Arquitetura do Conjunto de Instruções (ISA)

- Requisitos básicos da Arquitetura do Conjunto de Instruções:
  - Fácil de compreender e programar
  - Que permita o desenvolvimento de compiladores eficientes
  - Implementação simples e eficiente em hardware
  - Com o melhor desempenho possível
  - Eficiente do ponto de vista energético
  - Com o menor custo possível



# Arquitetura do Conjunto de Instruções (ISA)

- Alguns exemplos de ISAs:
  - **MIPS** – muito usado em ensino e em sistemas embebidos
  - **ARM** – presente em dispositivos como Nintendo DS, iPod, câmaras Canon PowerShot, a maioria dos smartphones, tablets e até portáteis recentes
  - **Intel x86** – arquitetura dominante em PCs, servidores e também em muitos Macs até à transição para ARM
  - **PowerPC** – utilizado em consolas como a Nintendo GameCube, Wii e na Xbox 360, além de ter sido usado em computadores Apple antes da transição para Intel
  - **Cell Broadband Engine** – desenvolvido pela Sony, Toshiba e IBM, usado na PlayStation 3
  - **RISC-V** – ISA aberta e em rápido crescimento, com grande adoção em investigação, ensino e novos sistemas embebidos

# Instruções e implementação hardware

- No projeto de um processador a definição do ***instruction set*** exige um delicado compromisso entre múltiplos aspetos, nomeadamente:
  - as facilidades oferecidas aos programadores (por ex. instruções de manipulação de *strings*)
  - a complexidade do hardware envolvido na sua implementação
- Quatro princípios básicos estão subjacentes a um bom design ao nível do hardware:
  - A regularidade favorece a simplicidade
  - Quanto mais pequeno mais rápido
  - O que é mais comum deve ser mais rápido
  - Um bom design implica compromissos adequados



# Instruções e implementação hardware

- **A regularidade favorece a simplicidade**
  - Ex1: todas as instruções do *instruction set* são codificadas com o mesmo número de bits
  - Ex2: instruções aritméticas operam sempre sobre registros internos e depositam o resultado também num registo interno
- **Quanto mais pequeno mais rápido**
- **O que é mais comum deve ser mais rápido**
  - Ex: quando o operando é uma constante esta deve fazer parte da instrução (é vulgar que mais de 50% das instruções que envolvem a ALU num programa utilizem constantes)
- **Um bom *design* implica compromissos adequados**
  - Ex: o compromisso que resulta entre a possibilidade de se poder codificar constantes de maior dimensão nas instruções e a manutenção da dimensão fixa nas instruções

# Classes de instruções

- Uma dada ISA pode incluir centenas de instruções
- É possível, no entanto, considerar a existência de um grupo limitado de **classes de instruções** comuns à generalidade das arquiteturas
- Classes de instruções:
  - **Processamento**
    - Aritméticas e lógicas
  - **Transferência de informação**
    - Cópia entre registos internos e entre registos internos e memória
  - **Controlo de fluxo de execução**
    - Alteração da sequência de execução (estruturas condicionais, ciclos, chamadas a funções,...)

# ISA – formato e codificação das instruções

- Codificação das instruções com um número de bits variável
  - Código mais pequeno
  - Maior flexibilidade
  - *Instruction fetch* em vários passos
- Codificação das instruções com um número de bits fixo
  - *Instruction fetch e decode* mais simples
  - Mais simples de implementar em *pipeline*

# ISA – número de registos internos do CPU

- Vantagens de um número pequeno de registos
  - Menos hardware
  - Acesso mais rápido
  - Menos bits para identificação do registo
  - Mudança de contexto mais rápida
- Vantagens de um número elevado de registos
  - Menos acessos à memória
  - Algumas variáveis dos programas podem residir em registos
  - Certos registos podem ter restrições de utilização

# ISA – localização dos operandos das instruções

- Arquiteturas baseadas em **acumulador**
  - Resultado das operações é armazenado num registo especial designado de acumulador
    - **add a**                      **# acc ← acc + a**
- Arquiteturas baseadas em **Stack**
  - Operandos e resultado armazenados numa *stack* (pilha) de registos
    - **add**                      **# tos ← tos + next**  
                                  **(tos = top of stack)**

# ISA – localização dos operandos das instruções

- Arquiteturas **Register-Memory**

- Operandos das instruções aritméticas e lógicas residem em registos internos do CPU ou em memória

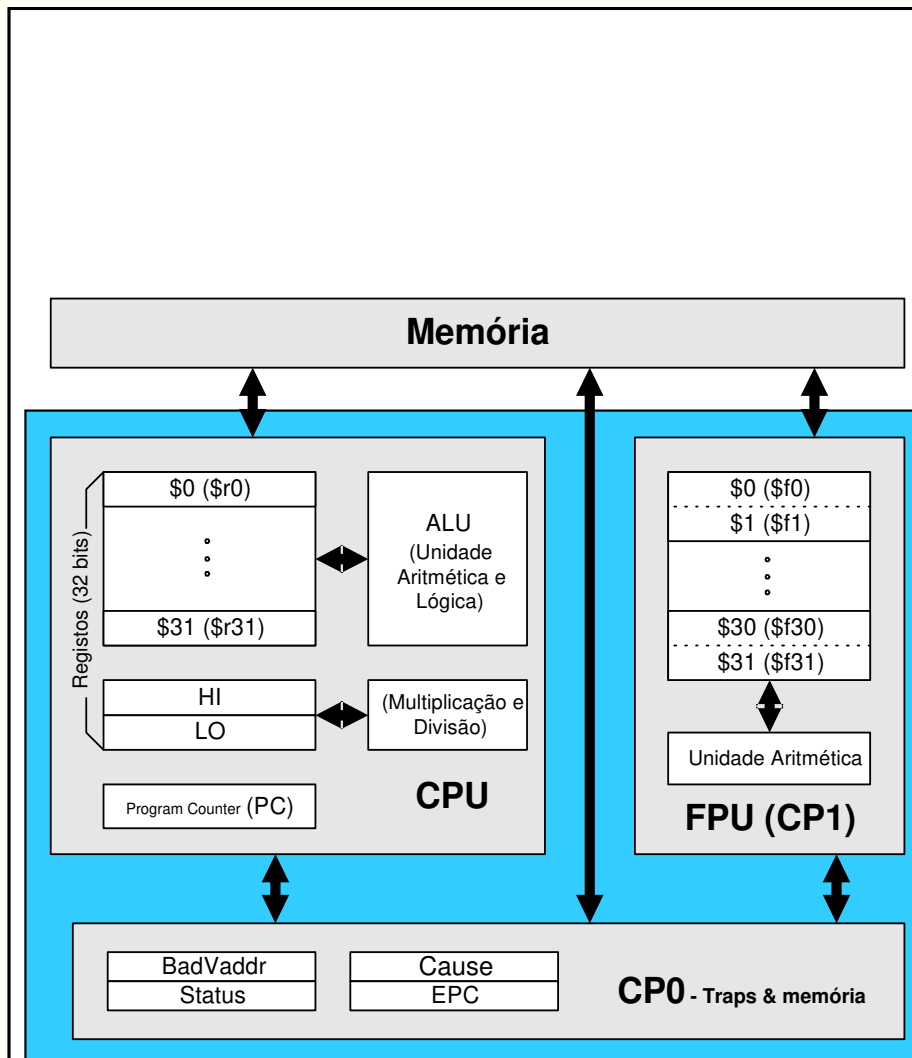
- **load**     **r1**, [a]                    # **r1** ← **mem[a]**
- **add**     **r1**, [b]                    # **r1** ← **r1** + **mem[b]**
- **store**    [c], r1                    # **mem[c]** ← **r1**

- Arquiteturas **Load-store**

- Operandos das instruções aritméticas e lógicas residem em registos internos do CPU de uso geral (mas nunca na memória).

- **load**     r1, [a]                    # **r1** ← **mem[a]**
- **load**     r2, [b]                    # **r2** ← **mem[b]**
- **add**     r3, r1, r2                  # **r3** ← **r1** + **r2**
- **store**    [c], r3                    # **mem[c]** ← **r3**

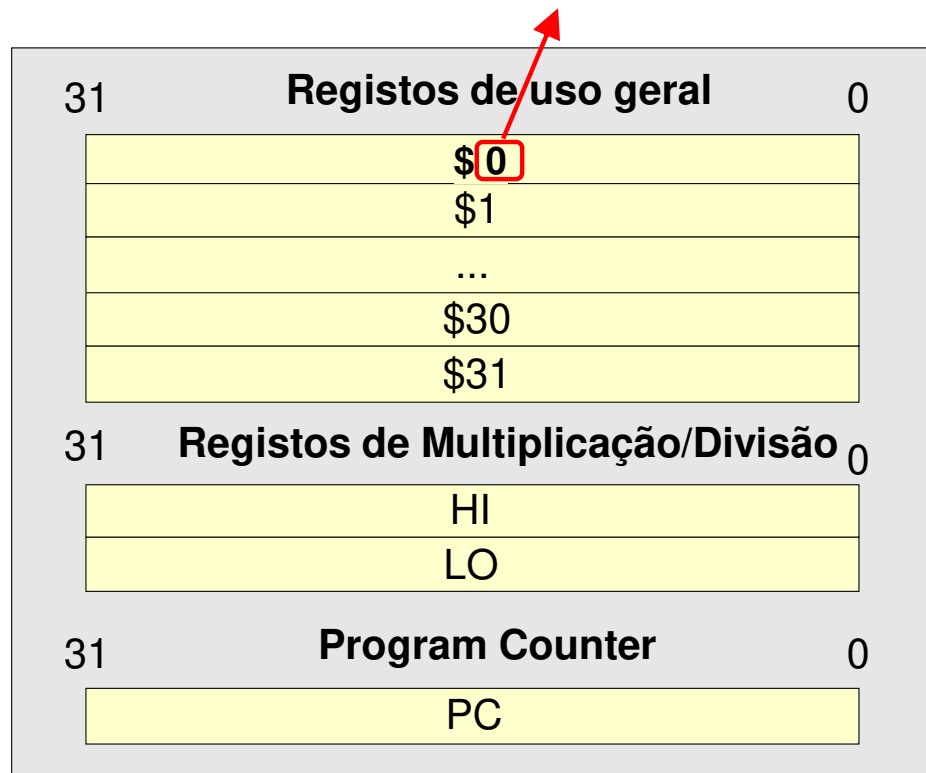
# Aspetos chave da arquitetura MIPS



- 32 Registos de uso geral, de 32 bits cada (1 word  $\Leftrightarrow$  32 bits)
- ISA baseado em **instruções de dimensão fixa** (32 bits)
- Arquitetura **load-store** (*register-register operation*)
- Memória organizada em bytes (memória *byte addressable*)
- Espaço de endereçamento de 32 bits ( $2^{32}$  endereços possíveis, i.e. máximo de 4 GB de memória)
- Barramento de dados externo de 32 bits

# Os registos internos do MIPS

Endereço do registo (0 a 31)



**Program Counter:** registo que contém o endereço de memória onde está armazenado o código da próxima instrução a executar

- Em *assembly* são, normalmente, usados nomes alternativos para os registos (nomes virtuais):

- **\$zero (\$0)**
- **\$at (\$1)**
- **\$v0 e \$v1 (\$2 e \$3)**
- **\$a0 a \$a3**
- **\$t0 a \$t9**
- **\$s0 a \$s7**
- **\$sp (\$29)**
- **\$ra (\$31)**

- Registo **\$0** tem sempre o valor **0x00000000** (apenas pode ser lido)



# Instruções aritméticas - SOMA

Formato da instrução *Assembly* do MIPS:



**add \$3, \$7, \$5** **#** Soma \$7 com \$5 e armazena o  
# resultado em \$3 (\$3 = \$7 + \$5)  
# Se \$3 armazenar a variável **a**, \$7 a  
# variável **b** e \$5 a variável **c**: **a=b+c**

comentário

# Instruções aritméticas - SOMA

Formato da instrução *Assembly* do MIPS:

**add** **Rdst, Rsrc1, Rsrc2** # Rdst = Rsrc1 + Rsrc2

Uma expressão do tipo  **$z = a + b + c + d$** , supondo que a, b, c, d, e z residem em **a: \$17, b: \$18, c: \$19, d: \$20** e **z: \$16**, tem de ser decomposta em:

```
add  $16, $17, $18  # Soma a com b, resultado em z
add  $16, $16, $19  # Soma z com c, resultado em z
add  $16, $16, $20  # Soma z com d, resultado em z
```

# Instruções aritméticas - SUBTRAÇÃO

Formato da instrução *Assembly* do MIPS:

**sub** **Rdst, Rsrc1, Rsrc2** # Rdst = Rsrc1 - Rsrc2

Uma expressão do tipo  **$z = (a + b) - (c + d)$** , supondo que a, b, c, d, e z residem em **a: \$17, b: \$18, c: \$19, d: \$20 e z: \$16**, tem de ser decomposta em:

**add** **\$8, \$17, \$18** # Soma a com b, resultado em x

**add** **\$9, \$19, \$20** # Soma c com d, resultado em y

**sub** **\$16, \$8, \$9** # Subtrai y a x, e coloca o resultado  
# em z

# Exemplo de tradução de C para *Assembly* MIPS

- Programa em C:

```
int a, b, c, d, z;  
z = (a + b) - (c + d);
```

---

```
# a: $17, b: $18, c: $19, d: $20, z: $16
```

```
...
```

```
add    $8, $17, $18    # r1 = a + b;
```

```
add    $9, $19, $20    # r2 = c + d;
```

```
sub    $16, $8, $9     # z = (a + b) - (c + d);
```

```
...
```

- A linguagem C é uma excelente forma de comentar programas em *Assembly* uma vez que permite uma interpretação direta e mais simples do(s) algoritmo(s) implementado(s).

# Instruções lógicas e de deslocamento

- Operadores lógicos bitwise em C:
  - **&** (AND), **|** (OR), **^** (XOR), **~** (NOT)
- Instruções lógicas do MIPS
  - **and Rdst, Rsrc1, Rsrc2** #  $Rdst = Rsrc1 \& Rsrc2$
  - **or Rdst, Rsrc1, Rsrc2** #  $Rdst = Rsrc1 | Rsrc2$
  - **nor Rdst, Rsrc1, Rsrc2** #  $Rdst = \sim(Rsrc1 | Rsrc2)$
  - **xor Rdst, Rsrc1, Rsrc2** #  $Rdst = (Rsrc1 \wedge Rsrc2)$
- Operadores de deslocamento em C:
  - **<<** shift left
  - **>>** shift right, **lógico** ou **aritmético**, dependendo da variável ser do tipo **unsigned** ou **signed**, respetivamente
- Instruções de deslocamento do MIPS
  - **sll Rdst, Rsrc, k** #  $Rdst = Rsrc \ll k$ ; (shift left logical)
  - **srl Rdst, Rsrc, k** #  $Rdst = Rsrc \gg k$ ; (shift right logical)
  - **sra Rdst, Rsrc, k** #  $Rdst = Rsrc \gg k$ ; (shift right arithmetic)

# Instruções lógicas e de deslocamento

- Operadores lógicos bit a bit (*bitwise operators*) em C:
  - **&** (AND), **|** (OR), **^** (XOR), **~** (NOT)
- A operação indicada é realizada bit a bit nos dois operandos, no caso do AND, do OR e do XOR e é feita a negação de todos os bits do operando no caso do NOT.
- Os operadores bit a bit "**&**" e "**|**" não devem ser confundidos com os operadores lógicos relacionais "**&&**" e "**||**".
- **Exercício:** determine os resultados deste programa:

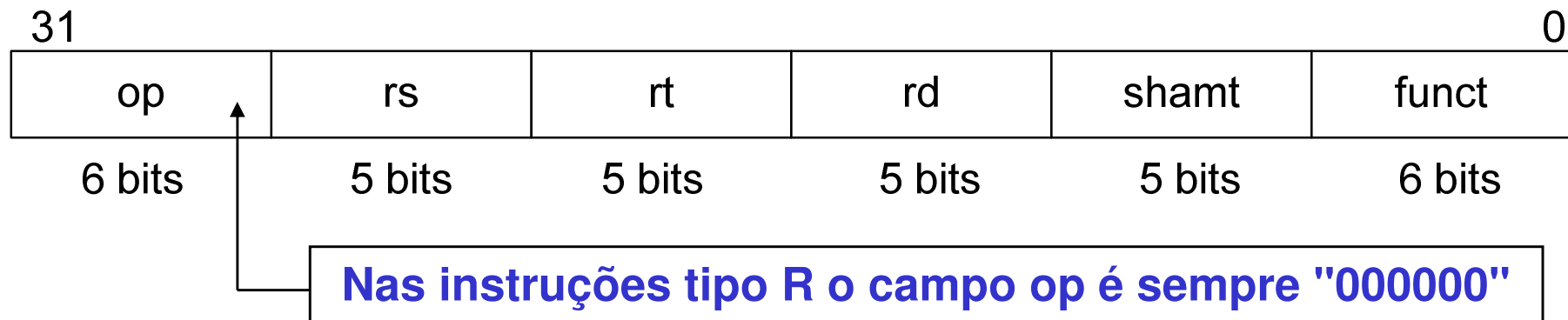
```
void main(void)
{
    int a = 10;
    int b = 9;
    printf("a & b = %d\n", a & b);    // ?
    printf("a && b = %d\n", a && b);  // ?
    printf("a | b = %d\n", a | b);    // ?
    printf("a || b = %d\n", a || b);  // ?
}
```

# Instruções de transferência entre registros internos

- Transferência entre registros internos:  $R_{dst} = R_{src}$
- Registro **\$0** do MIPS tem sempre o valor **0x00000000** (apenas pode ser lido)
- Utilizando o registro **\$0** e a instrução lógica OR é possível realizar uma operação de transferência entre registros internos:
  - **or**     **Rdst, Rsrc, \$0**     #  $R_{dst} = (R_{src} | 0) = R_{src}$
  - Exemplo: **or** **\$t1, \$t2, \$0**   #  $\$t1 = \$t2$
- Para esta operação é habitualmente usada uma **instrução virtual** que melhora a legibilidade dos programas - **"move"**.
- No processo de geração do código máquina, o *assembler* substitui essa instrução pela instrução nativa anterior:
  - **move Rdst, Rsrc**     #  $R_{dst} = R_{src}$
  - Exemplo: **move** **\$t1, \$t2**   #  $\$t1 = \$t2$  (or  $\$t1, \$t2, \$0$ )

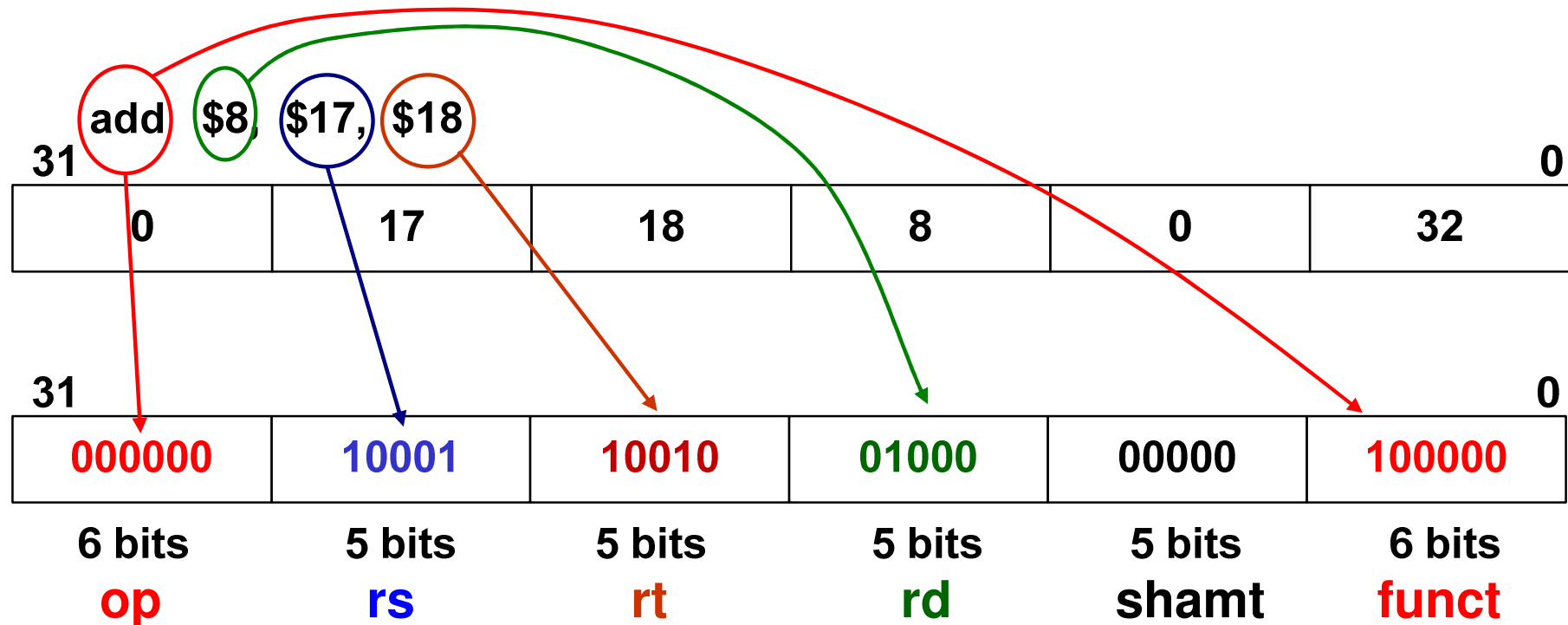
# Codificação de instruções no MIPS – formato R

- O formato R é um dos três formatos de codificação de instruções no MIPS
- Campos da instrução:
  - op:** *opcode* (é sempre zero nas instruções tipo R)
  - rs:** Endereço do registo que contém o 1º operando fonte
  - rt:** Endereço do registo que contém o 2º operando fonte
  - rd:** Endereço do registo onde o resultado vai ser armazenado
  - shamt:** *shift amount* (útil apenas em instruções de deslocamento)
  - funct:** código da operação a realizar





# Codificação de instruções no MIPS – formato R



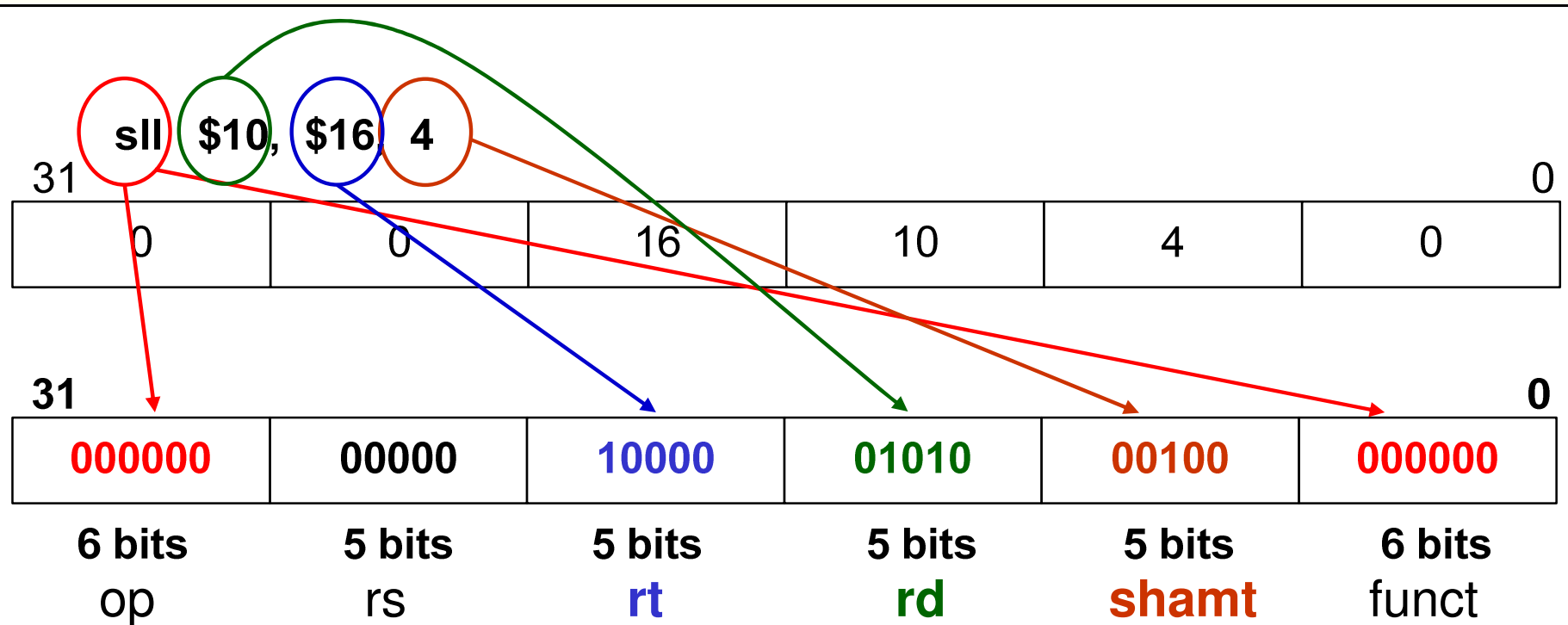
Código máquina  
da instrução:

add rd, rs, rt

000000 10001 10010 01000 00000 100000<sub>2</sub>

0000 0010 0011 0010 0100 0000 0010 0000<sub>2</sub> = 0x02324020

# Codificação de instruções no MIPS – formato R



**sll** **rd**, **rt**, **shamt**

Código máquina

da instrução:  $00000000000100000101000100000000_2 = 0x00105100$

O que faz a instrução cujo código máquina é:  $0x00000000$  ?

# Questões

- O que é um endereço?
- O que é o espaço de endereçamento de um processador?
- Como se organiza internamente um processador? Quais são os blocos fundamentais da secção de dados? Para que serve a unidade de controlo?
- O que é o conceito "stored-program"?
- Como se codifica uma instrução? Que informação fundamental deverá ter o código de uma instrução?
- O que é o ISA?
- Quais são as classes de instruções que agrupam as instruções de uma arquitetura?

# Questões

- O que caracteriza as arquiteturas "register-memory" e "load-store"? De que tipo é a arquitetura MIPS?
- Com quantos bits são codificadas as instruções no MIPS? Quantos registos internos tem o MIPS? O que diferencia o registo `$0` dos restantes? Qual o número do registo interno do MIPS a que corresponde o registo `$ra`?
- Quais os campos em que se divide o formato de codificação **R**? Qual o significado de cada um desses campos? Qual o valor do campo **opCode** nesse formato?
- O que faz a instrução cujo código máquina é: `0x00000000`?
- O símbolo `>>` da linguagem C significa deslocamento à direita e é traduzido por **SRL** ou **SRA** (no caso do MIPS). Quando é que usado **SRL** e quando é que é usado **SRA**?
- Qual a instrução nativa do MIPS em que é traduzida a instrução virtual `"move $4, $15"`?

# Exercícios

- Determine o código máquina das seguintes instruções:  
**xor \$5, \$13, \$24 - sub \$30, \$14, 8 - sll \$3, \$9, 7**  
**sra \$18, \$9, 8**
- Traduza para instruções *assembly* do MIPS a seguinte expressão aritmética, supondo **x** e **y** inteiros e residentes em **\$t2** e **\$t5**, respetivamente (apenas pode usar instruções nativas e não deverá usar a instrução de multiplicação):  
**y = -3 \* x + 5;**
- Traduza para instruções *assembly* do MIPS o seguinte trecho de código:  
**int a, b, c; //a:\$t0, b:\$t1, c:\$t2**  
**unsigned int x, y, z; //x:\$a0, y:\$a1, z:\$a2**  
**z = x >> 2 + y;**  
**c = a >> 5 - 2 \* b;**