

## Aula 2

- 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, Arnaldo Oliveira

# Classes de instruções

- Um dada arquitetura pode ter um ISA com 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,...)

# 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 registos 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

# 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

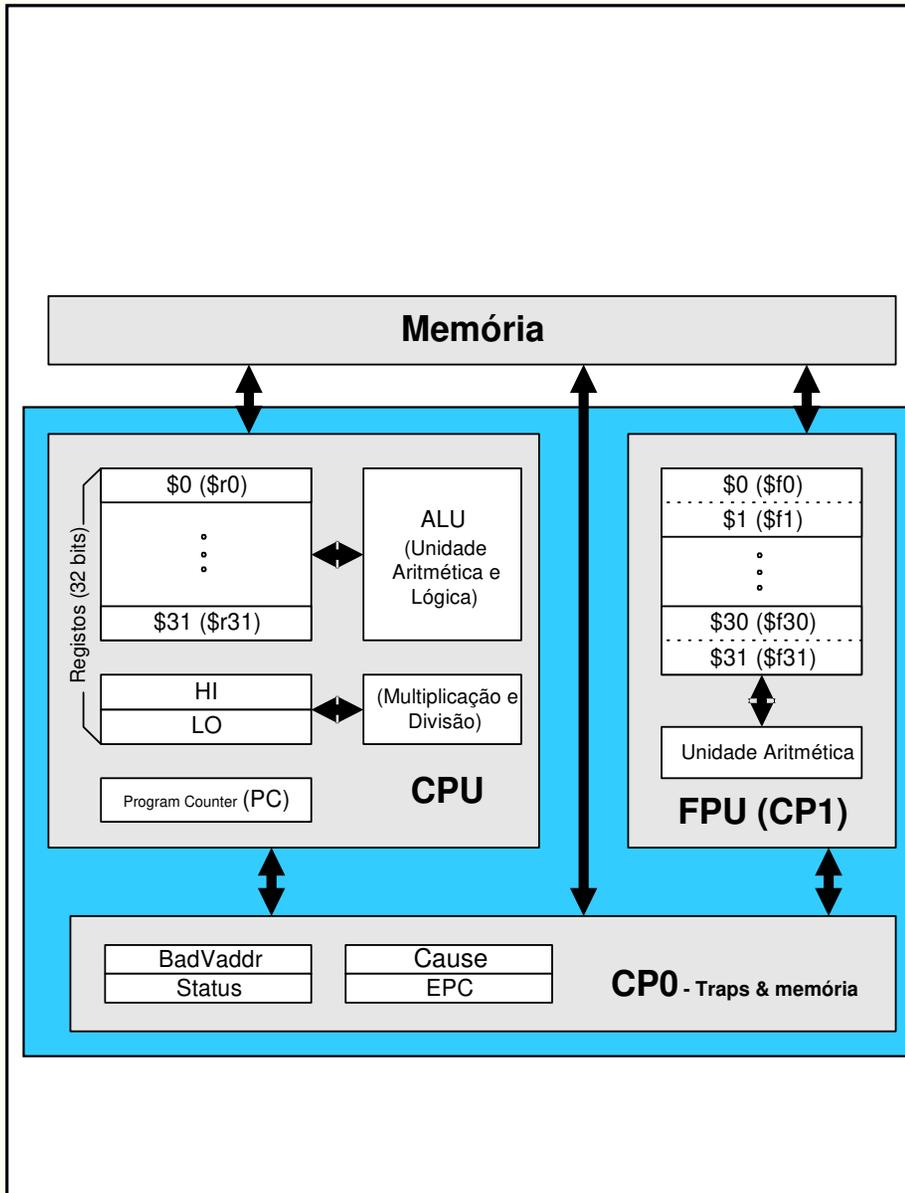
- Arquiteturas **Load-store**

- 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

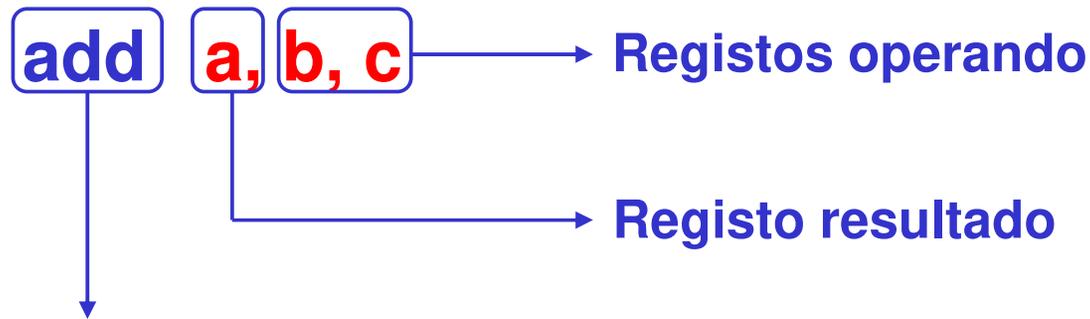
# Aspetos chave da arquitetura MIPS

- 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



# Instruções aritméticas - SOMA

Formato da instrução *Assembly* do MIPS:



**Mnemónica da instrução**  
(palavra-chave que identifica a instrução)

**add a, b, c** **#** Soma **b** com **c** e armazena o resultado  
# em **a** ( $a = b + c$ ) **comentário**

# Instruções aritméticas - SOMA

Formato da instrução *Assembly* do MIPS:

**add a, b, c** # Soma **b** com **c** e armazena o resultado  
# em **a** ( $a = b + c$ )

Uma expressão do tipo

**$z = a + b + c + d$**

Tem de ser decomposta em:

**add z, a, b** # Soma **a** com **b**, resultado em **z**

**add z, z, c** # Soma **z** com **c**, resultado em **z**

**add z, z, d** # Soma **z** com **d**, resultado em **z**

# Instruções aritméticas - SUBTRAÇÃO

Formato da instrução Assembly do Mips:

**sub** **a**, **b**, **c** # Subtrai **c** a **b** e armazena o resultado  
# em **a** ( $a = b - c$ )

**Exemplo:** A expressão  $z = (a + b) - (c + d)$   
tem de ser decomposta em:

**add** **x**, **a**, **b** # Soma **a** com **b**, resultado em **x**

**add** **y**, **c**, **d** # Soma **c** com **d**, resultado em **y**

**sub** **z**, **x**, **y** # Subtrai **y** a **x**, e coloca o resultado em **z**

# 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)

# Exemplo de tradução de C para *Assembly* MIPS

- Programa em C:

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

- Em *assembly* (supondo que a, b, c, d, z residem em a: \$17, b: \$18, c: \$19, d: \$20 e z: \$16):

```
add  $8, $17, $18 # Soma $17 com $18 e armazena o  
      # resultado em $8  
add  $9, $19, $20 # Soma $19 com $20 e armazena o  
      # resultado em $9  
sub  $16, $8, $9  # Subtrai $9 a $8 e armazena o  
      # resultado em $16
```

# 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;
```

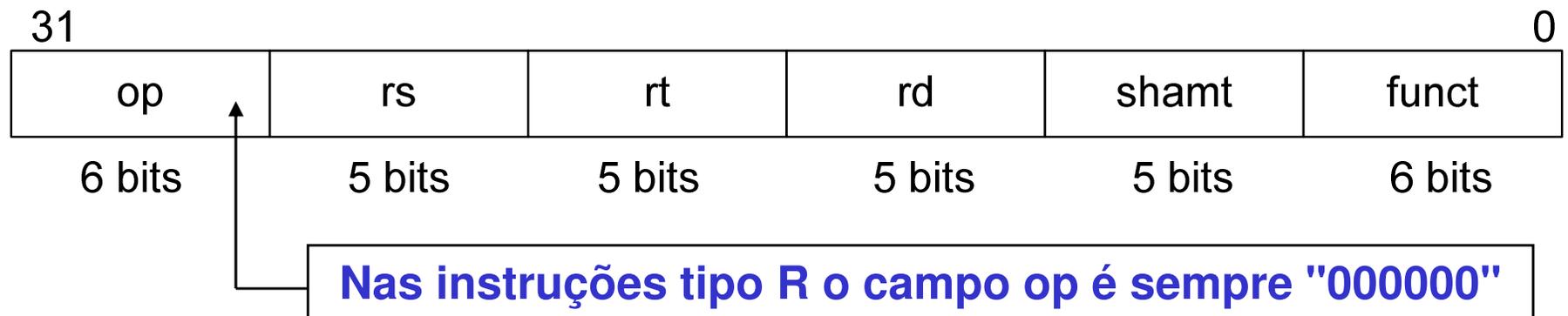
```
sub    $16, $8, $9     # z = (a + b) - (c + d);
```

```
...
```

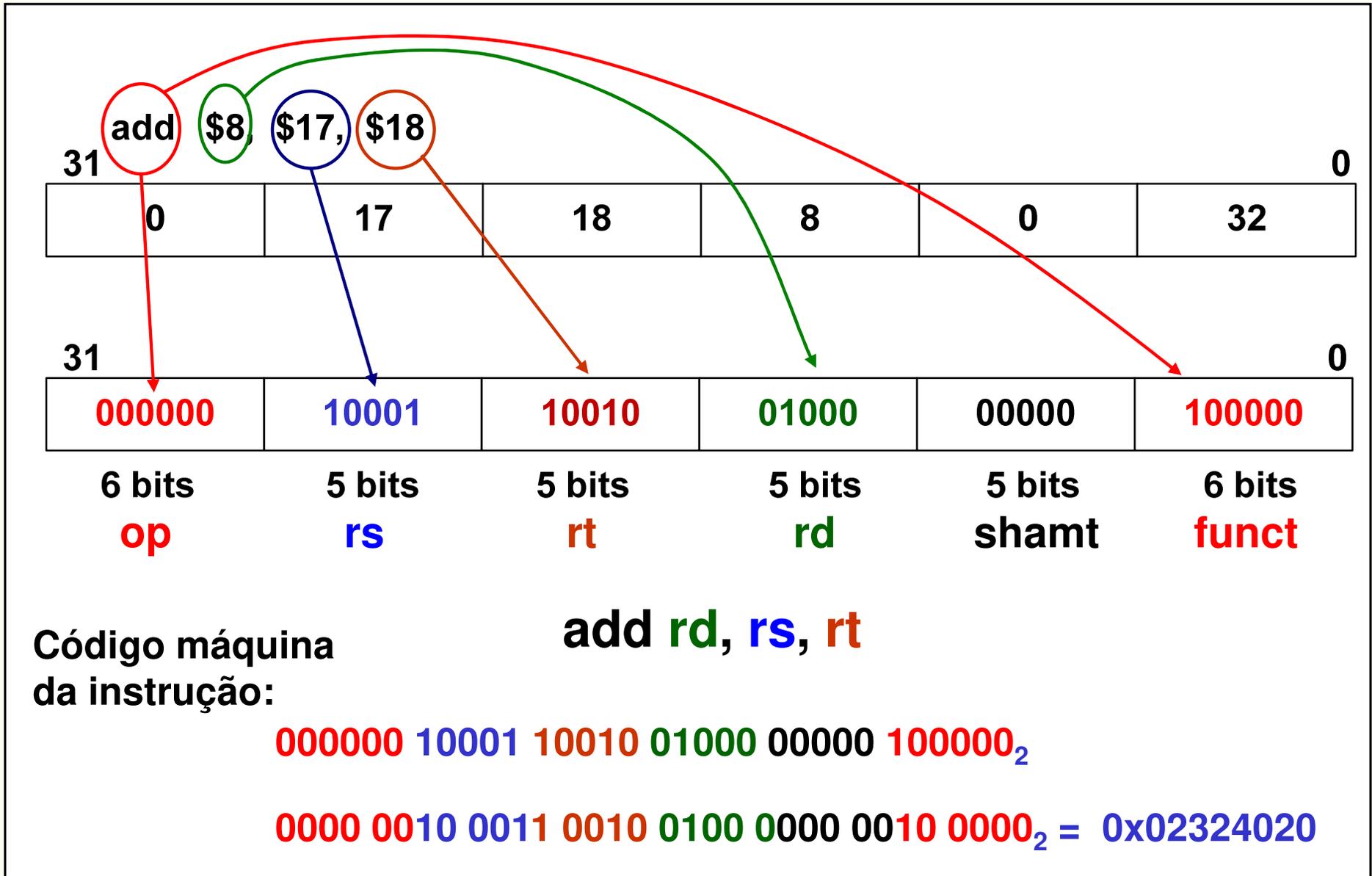
- A linguagem C é uma excelente forma de comentar programas em *Assembly* uma vez que permite uma interpretação direta e mais simples do(s) algoritmo(s) implementado(s).

# 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



# Instruções lógicas e de deslocamento

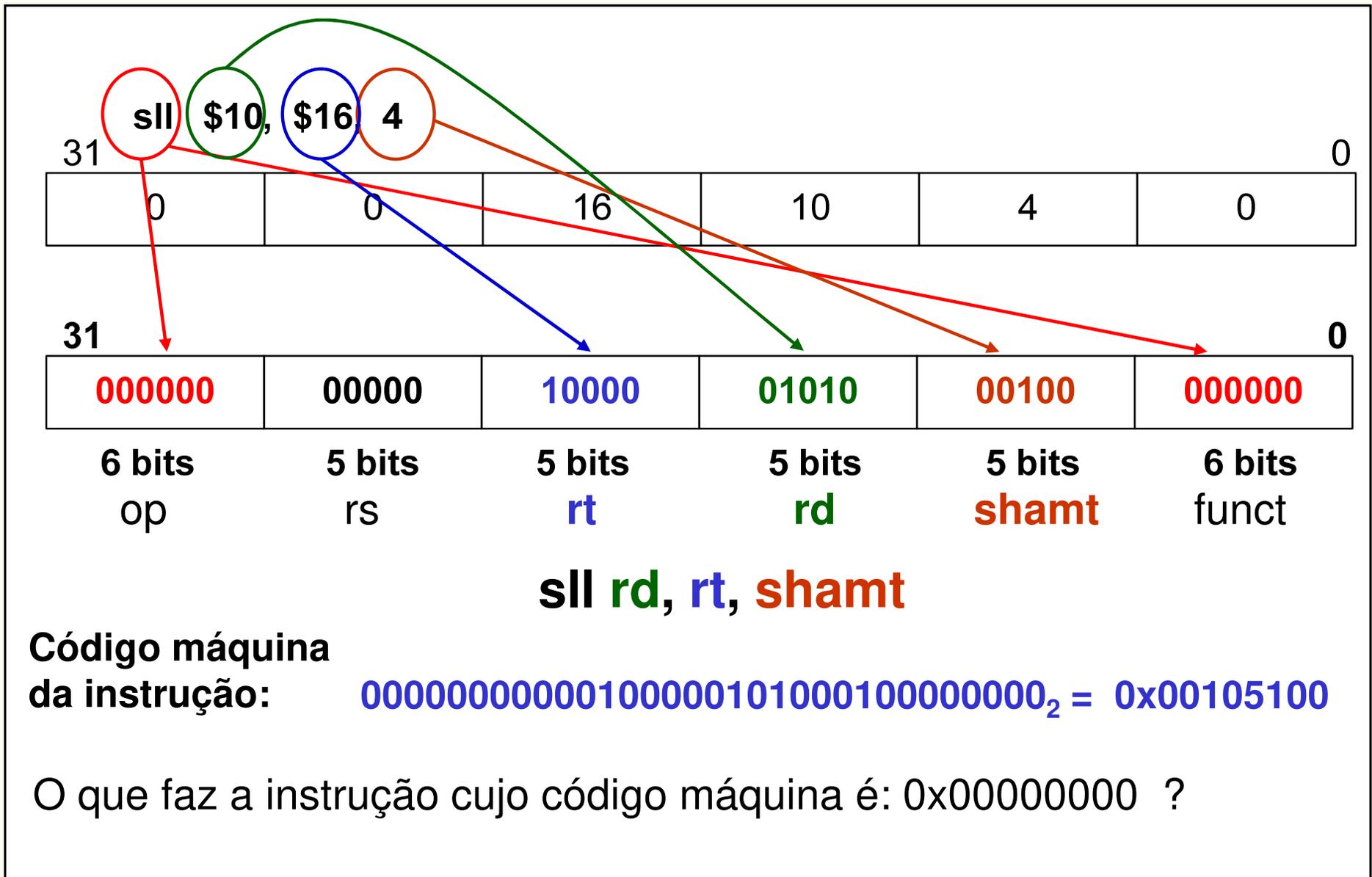
- Operadores lógicos bit a bit (*bitwise operators*) em C:
  - `&` (AND), `|` (OR), `^` (XOR), `~` (NOT)
- A operação indicada é realizada bit a bit nos dois operandos, no caso do AND, do OR e do XOR e é feita a negação de todos os bits do operando no caso do NOT.
- Os operadores bit a bit "`&`" e "`|`" 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 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 = ~(Rsrc1 | Rsrc2)
  - **xor Rdst, Rsrc1, Rsrc2** # Rdst = (Rsrc1 ^ 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 << k; (shift left logical)
  - **srl Rdst, Rsrc, k** # Rdst = Rsrc >> k; (shift right logical)
  - **sra Rdst, Rsrc, k** # Rdst = Rsrc >> k; (shift right arithmetic)

# Codificação de instruções no MIPS – formato R



## Instruções de transferência entre registos internos

- Transferência entre registos internos:  $Rdst = Rsrc$
- Registo **\$0** do MIPS tem sempre o valor **0x00000000** (apenas pode ser lido)
- Utilizando o registo **\$0** e a instrução lógica OR é possível realizar uma operação de transferência entre registos internos:
  - **or Rdst, Rsrc, \$0**      #  $Rdst = (Rsrc | 0) = Rsrc$
  - 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**      #  $Rdst = Rsrc$
  - Exemplo: **move \$t1, \$t2**      #  $\$t1 = \$t2$  (or \$t1, \$t2, \$0)

# 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`, respectivamente (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;
```