Aula 2

- Instruções e classes de instruções
- Princípios básicos de projeto de uma Arquitetura
- Aspetos chave da arquitetura MIPS
- Instruções aritméticas
- Instruções lógicas e de deslocamento
- Codificação de instruções no MIPS: formato R

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador

Introdução: 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

ISA: Instruções e classes de instruções

"It is easy to see by formal-logical methods that there exist certain instruction sets that are in abstract adequate to control and cause the execution of any sequence of operations...

The really decisive considerations from the present point of view, in selecting an instruction set, are more of a practical nature: simplicity of the equipment demanded by the instruction set, and the clarity of its application to the actually important problems together with the speed of its handling of those problems"

Burks, Goldstine and von Neumann, 1947

Instruções e classes de instruções

Será, portanto, possível considerar a existência de um grupo limitado de classes de instruções que possam ser consideradas comuns à generalidade das arquiteturas?

> *There must certainly be instructions for performing the fundamental arithmetic operations*
>
> *Burks, Goldstine and von Neumann, 1947*

- **Classes de instruções:**
  - Processamento (aritméticas e lógicas)
  - Transferência de informação
  - Controlo de fluxo de execução

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

Arquitetura do set de instruções (ISA). Relembrando...

- Formato e codificação das instruções
  - como são descodificadas?
- Operandos das instruções e resultados
  - onde podem residir?
  - quantos operandos explícitos?
  - como localizar?
  - quais podem residir na memória externa?
- Tipo e dimensão dos dados
- Operações
  - quais são suportadas?
- Instruções auxiliares
  - jumps, conditions, branches

ISA – formato e codificação das instruções

- Tamanho variável
  - Código mais pequeno
  - Maior flexibilidade
  - *Instruction fetch* em vários passos
- Tamanho fixo
  - *Instruction fetch* e *decode* mais simples
  - Mais simples de implementar em *pipeline*

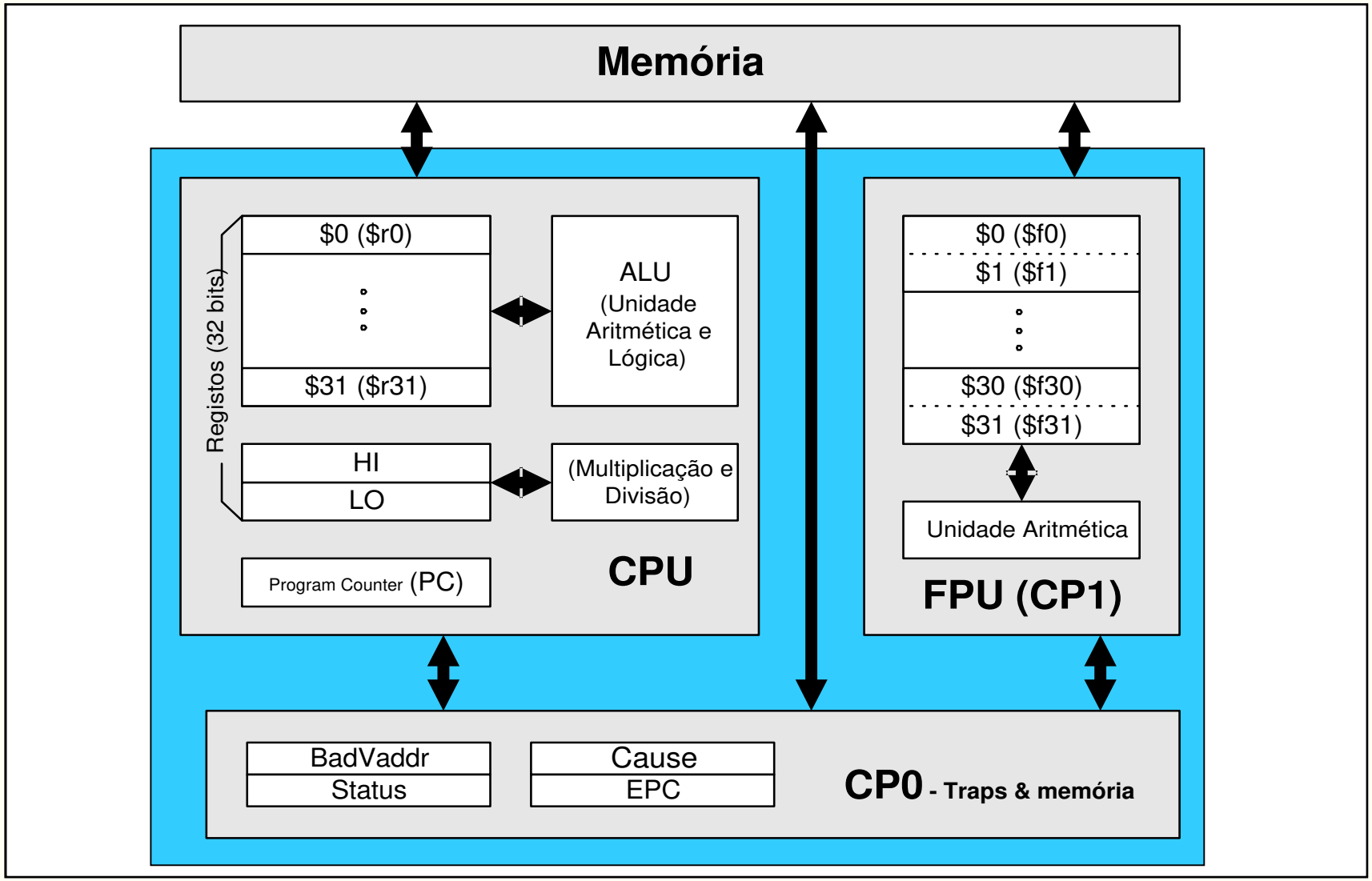
ISA – número de registos internos do CPU

- 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
  - Variáveis em registos
  - Certos registos podem ter restrições de utilização

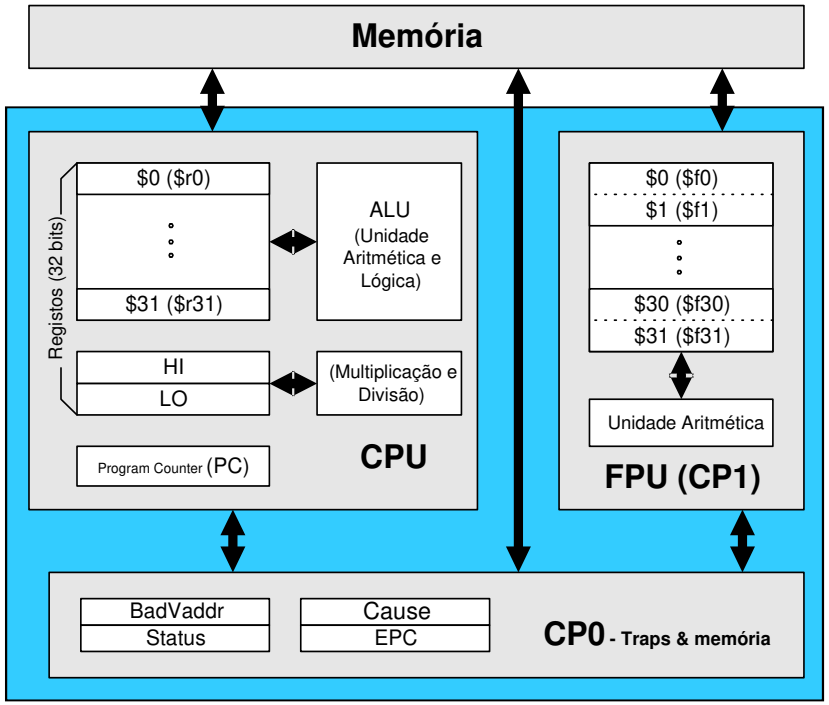
ISA – localização dos operandos das instruções

- Arquiteturas baseadas em **acumulador**
  - Resultado das operações é armazenado num registo especial designado de acumulador
- Arquiteturas baseadas em **Stack**
  - Operandos e resultado armazenados numa *stack* (pilha) de registos
- Arquiteturas **Register-Memory**
  - Operandos residem em registos internos do CPU ou em memória
- Arquiteturas **Load-store**
  - Operandos das instruções residem em registos internos do CPU de uso geral (mas nunca na memória).

Arquitetura MIPS



Aspetos chave da arquitetura MIPS



- 32 Registos de uso geral, de 32 bits cada (1 word ⇔ 32 bits)
- ISA baseado em **instruções de dimensão fixa** (32 bits)
- 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 data bus externo que conecta a memoria
- Arquitetura ***load-store*** *(register-register operation)*

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 adição do tipo

z = a + b + c + d

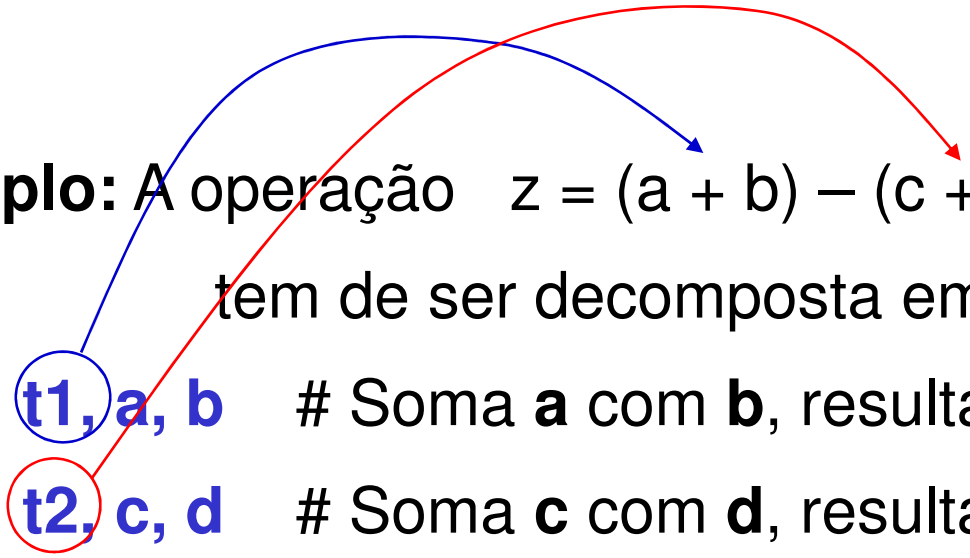
Tem de ser decomposta em:

```
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)
```

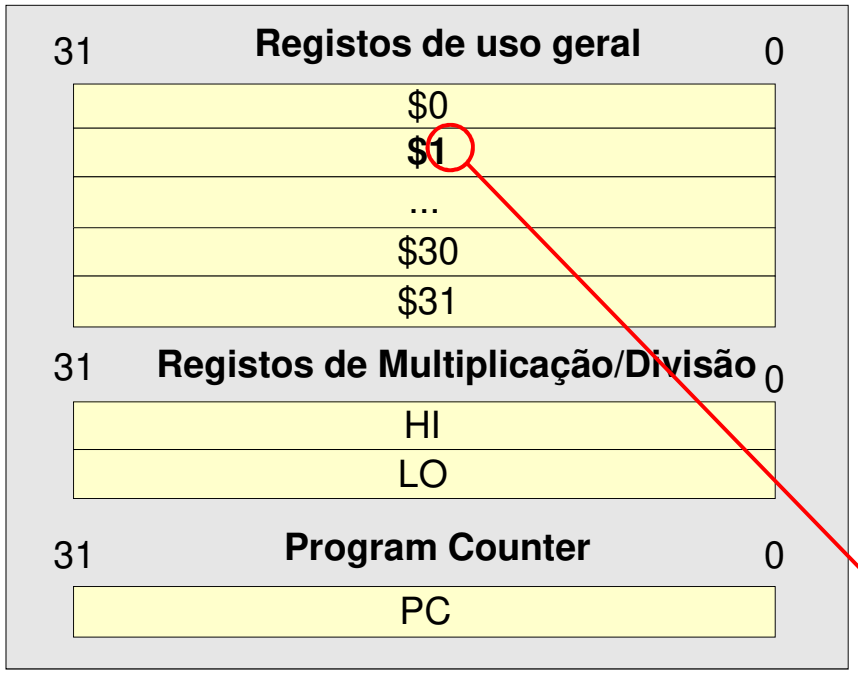


Exemplo: A operação z = (a + b) – (c + d)

tem de ser decomposta em:

```
add  t1, a, b   # Soma a com b, resultado em t1
add  t2, c, d   # Soma c com d, resultado em t2
sub  z, t1, t2  # Subtrai t2 a t1, resultado em z
```

Os registos internos do MIPS



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)**

Porquê 32 registos ?

Endereço do registo (0 a 31)

na instrucao, so tens 5 bits para identificar o registo destino/base

Exemplo de tradução de C para *Assembly* MIPS

- Programa em C:

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

- Em assembly (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
```

Como comentar um programa *Assembly*

```
# 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**)

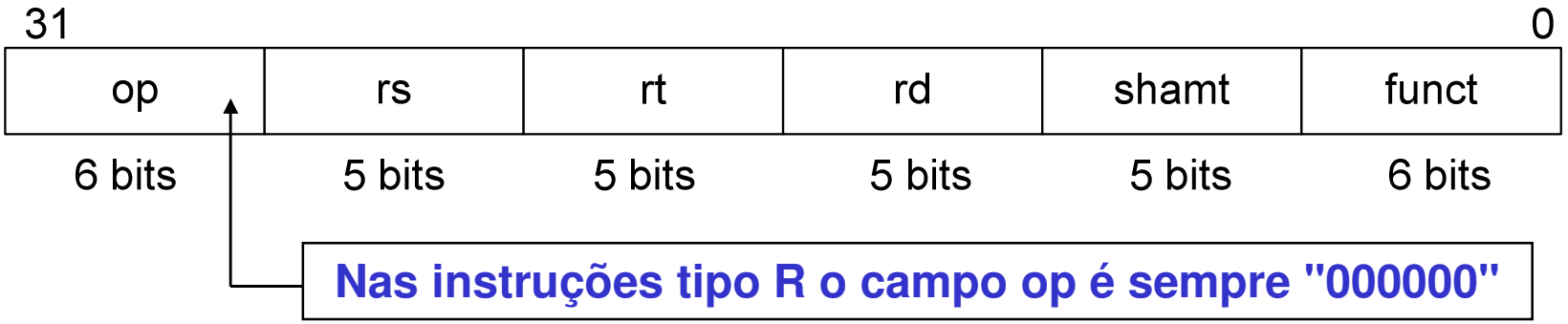
  rs: Endereço do registo que contém o 1º operando fonte

  rt: Endereço do registo que contém o 2º operando fonte

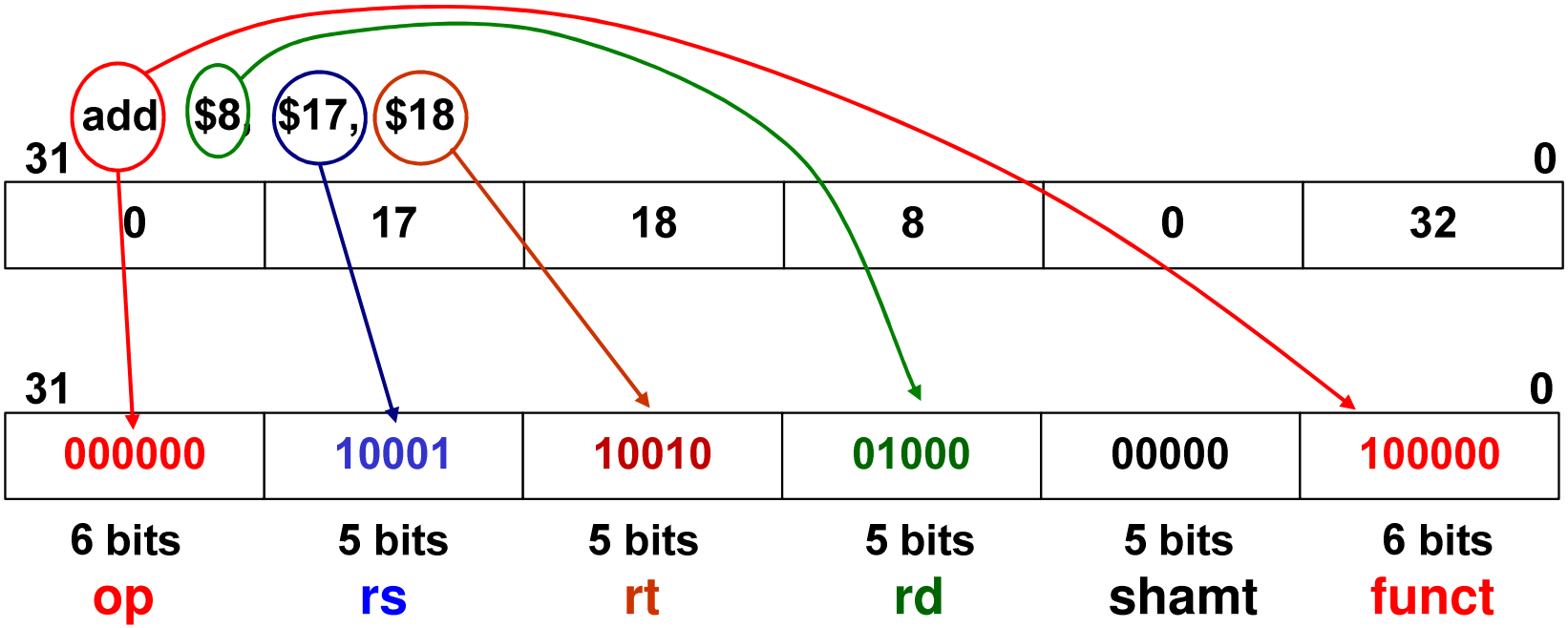
  rd: Endereço do registo onde o resultado vai ser armazenado

  shamt: ***shift amount*** (útil apenas em instruções de deslocamento)

  funct: código da operação a realizar



Codificação de instruções no MIPS – formato R



add rd, rs, rt

Código máquina da instrução:

$000000\,10001\,10010\,01000\,00000\,100000_2$

$0000\ 0010\ 0011\ 0010\ 0100\ 0000\ 0010\ 0000_2 = 0x02324020$

Instruções lógicas e de deslocamento

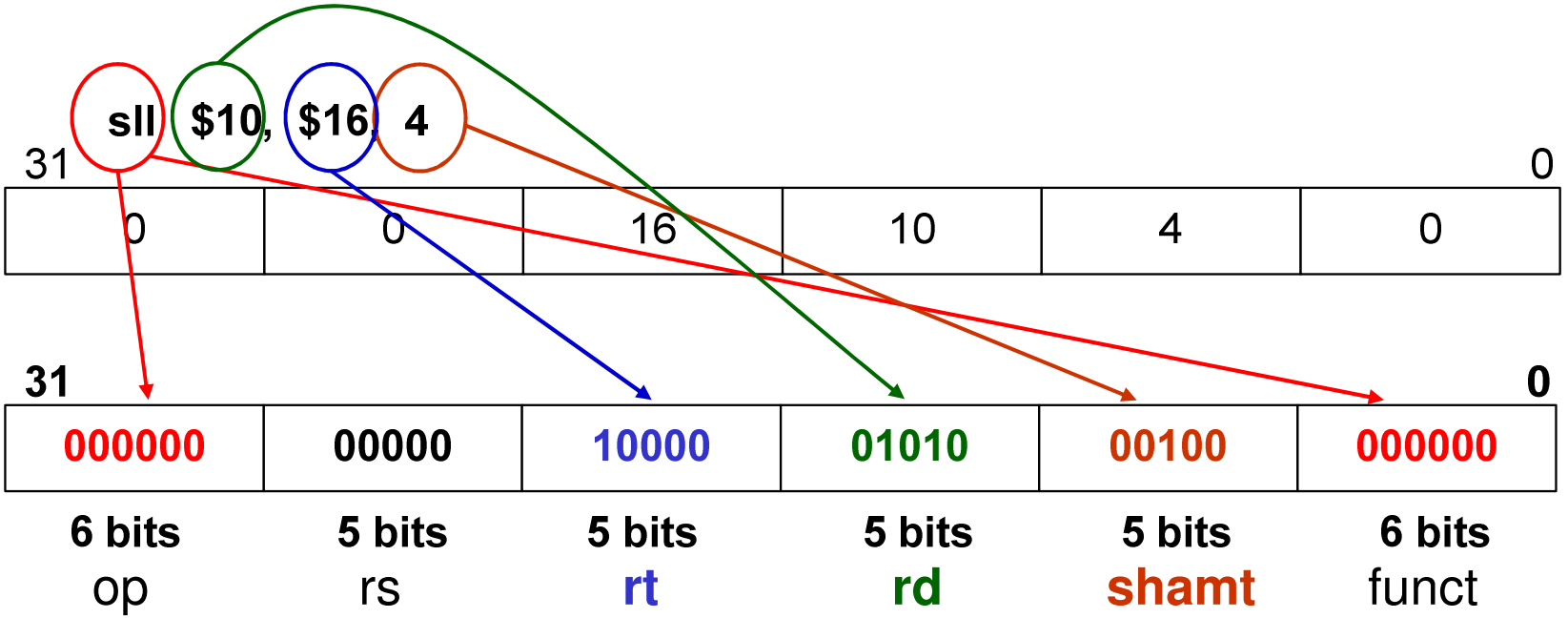
- Operadores lógicos bitwise em C:
  - **&** (AND), **|** (OR), **^** (XOR), **~** (NOT)
- Instruções lógicas do MIPS
  - **and Rdst, Rsrc1, Rsrc2** # Rdst = Rsrc1 & Rsrc2
  - **or Rdst, Rsrc1, Rsrc2** # Rdst = Rsrc1 | Rsrc2
  - **nor Rdst, Rsrc1, Rsrc2** # Rdst = ~(Rsrc1 | Rsrc2)
  - **xor Rdst, Rsrc1, 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)

x2, 4, 8, 16, 32, 64 etc…

/2, 4, 8, 16, 32, 64, 128….

fill registo do numero de sina, por exemplo

Codificação de instruções no MIPS – formato R



sll rd, rt, shamt

Código máquina da instrução: $00000000001000001010001000000000_2$ = **0x00105100**

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

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, $0, Rsrc** # Rdst = (0 | Rsrc) = Rsrc
  - Exemplo: **or $t1, $0, $t2** # $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, $0, $t2)