

Aula 2

- 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

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 aspectos, 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

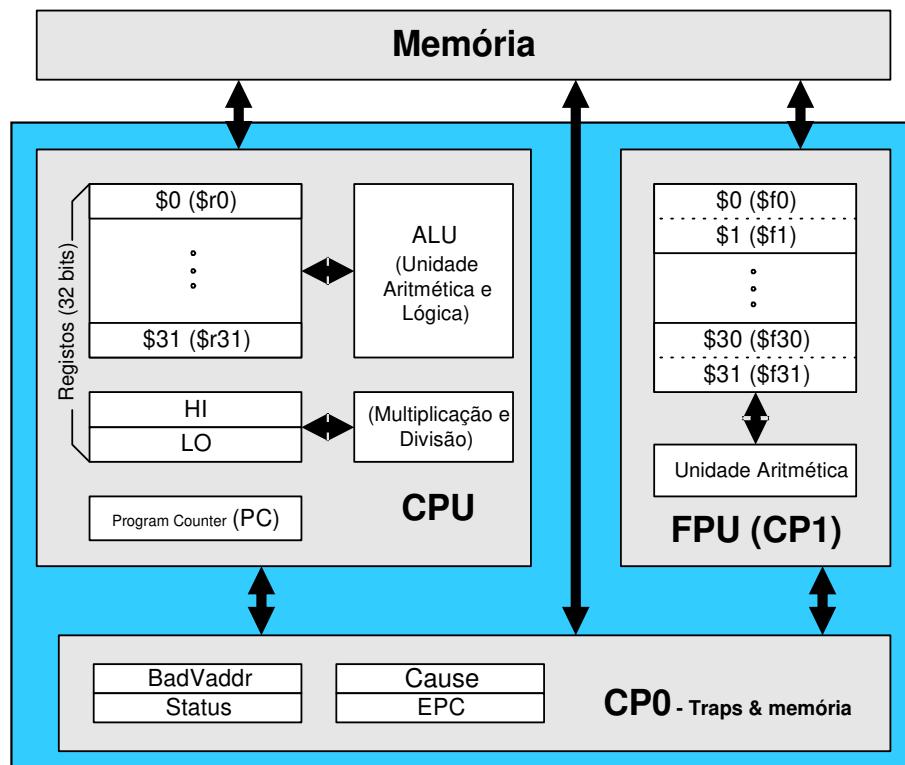
- **load r1, [a]** # $r1 \leftarrow \text{mem}[a]$
- **add r1, [b]** # $r1 \leftarrow r1 + \text{mem}[b]$
- **store [c], r1** # $\text{mem}[c] \leftarrow r1$

- 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]** # $r1 \leftarrow \text{mem}[a]$
- **load r2, [b]** # $r2 \leftarrow \text{mem}[b]$
- **add r3, r1, r2** # $r3 \leftarrow r1 + r2$
- **store [c], r3** # $\text{mem}[c] \leftarrow r3$

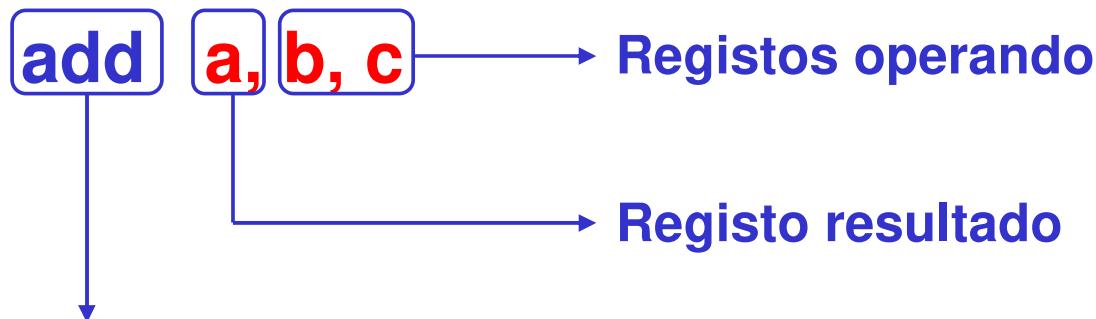
Aspetos chave da arquitetura MIPS



- 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)

A green arrow points from the '#' symbol in the assembly code to the word 'comentário' (comment) in green text above it.

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:

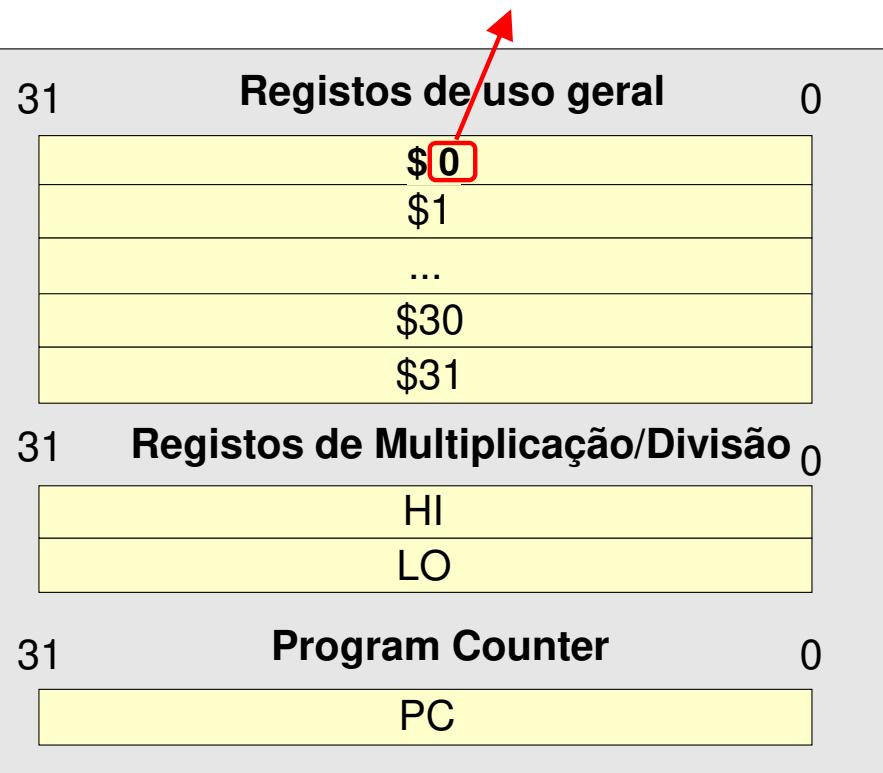
add x, a, b # Soma **a** com **b**, resultado em **x**

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)

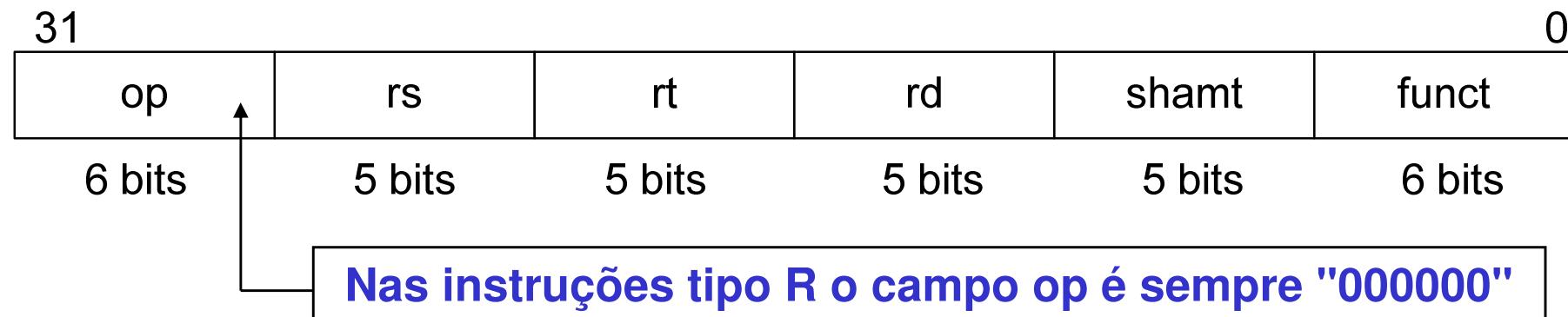
rs: Endereço do registo que contém o 1º operando fonte

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

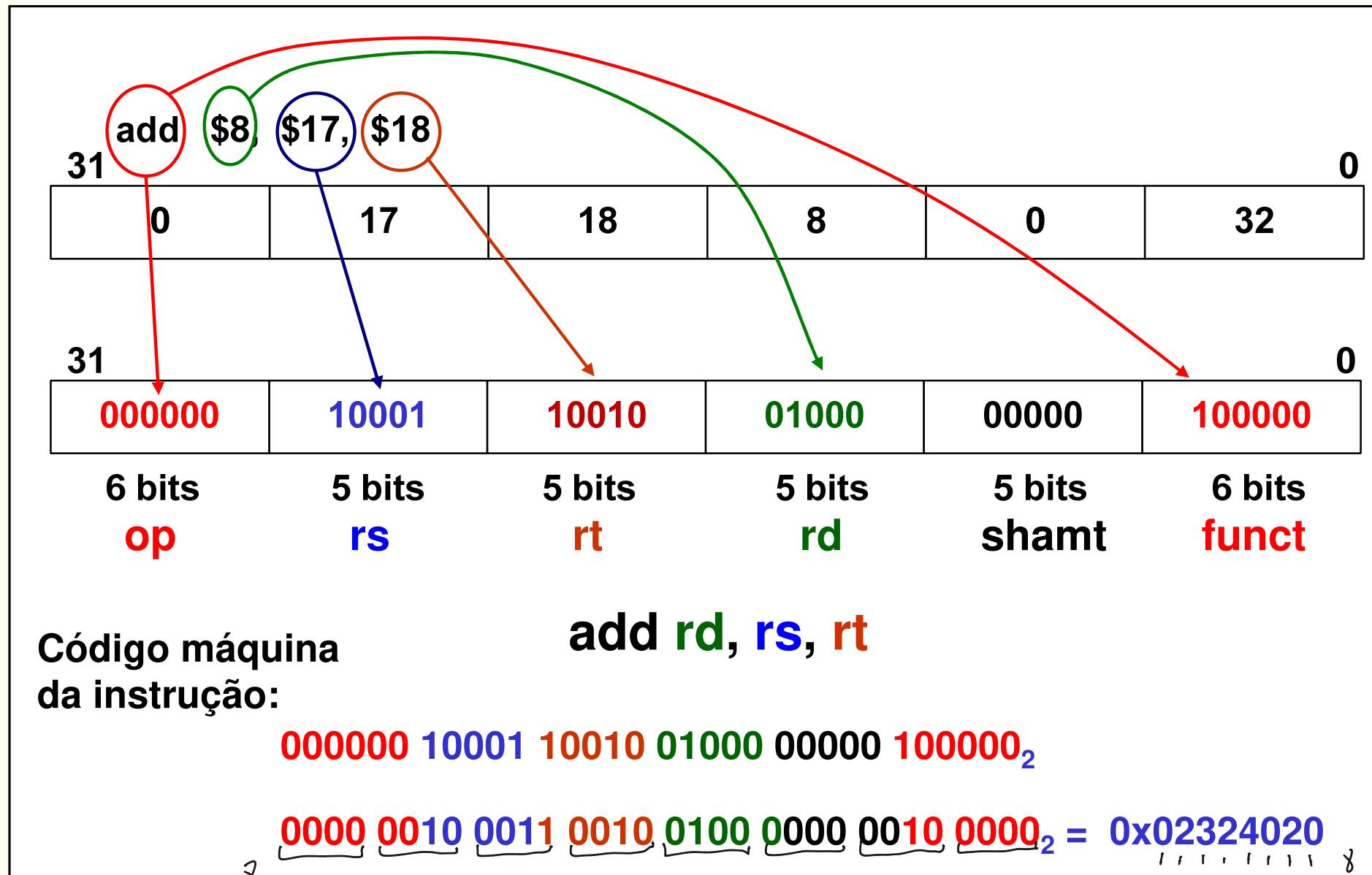
rd: Endereço do registo onde o resultado vai ser armazenado

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

funct: código da operação a realizar



Codificação de instruções no MIPS – formato R



$\&\&$ → Todos ≠ 0 // → Pelo menos 1 ≠ 0

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:

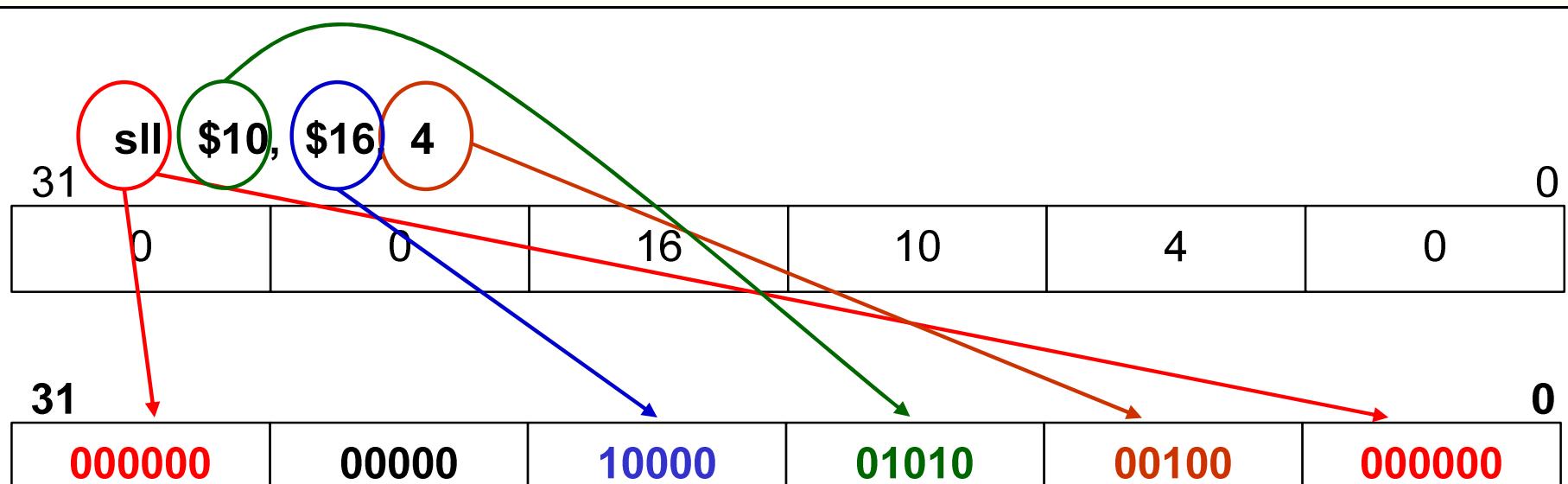
```
void main(void)
{
    int a = 10;
    int b = 9;
    printf("a & b = %d\n", a & b);      // ? 8
    printf("a && b = %d\n", a && b);    // ? 1
    printf("a | b = %d\n", a | b);       // ? 11
    printf("a || b = %d\n", a || b);     // ? 1
}
```

$$\begin{array}{r} 1010 \\ 1001 \\ \hline 1000 \end{array} \quad \begin{array}{l} 10 \neq 0 \\ 9 \neq 0 \end{array} \left. \right\} \text{true} \quad \begin{array}{r} 1010 \\ 1001 \\ \hline 1011 \end{array}$$

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** # $Rdst = (Rsrc1 ^ Rsrc2)$
- 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



sll rd, rt, shamt

Código máquina
da instrução:

$0000|0000|0001|0000|0101|0001|0000|0000_2 = 0x00105100$

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

NADA

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** # $Rdst = (Rsrc \mid 0) = Rsrc$
 - 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

- ① • O que caracteriza as arquiteturas "register-memory" e "load-store"? De que tipo é a arquitetura MIPS?
- ② • 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**?
- ③ • 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?
- ④ • O que faz a instrução cujo código máquina é: **0x00000000**?
- ⑤ • 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**?
- ⑥ • Qual a instrução nativa do MIPS em que é traduzida a instrução virtual "**move \$4, \$15**"?

Exercícios

(7)

- Determine o código máquina das seguintes instruções:

xor \$5,\$13,\$24 - sub \$30,\$14,8 - sll \$3,\$9,7

sra \$18,\$9,8

(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):

$$y = -3 * x + 5;$$

(10)

- 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;
```

1) No register-memory os operandos das instruções aritméticas e lógicas residem nos registo internos do CPU ou em memória.

No load-store os operandos das instruções aritméticas e lógicas residem em registo internos do CPU, mas nunca na memória.

0 MIPS usa a arquitetura load-store.

2) As instruções do MIPS são codificadas em 32 bits

0 MIPS tem 32 registo internos.

\$0 é um valor que alem pode ser lido, e contém o valor 0.

\$ra = \$31

3) Os campos em que se divide o formato de codificação são:

- op → 6 bits opcode (sempre 0 nas instruções tipo R)
- rs → 5 reg. 1º operando
- rt → 5 reg. 2º operando
- rd → 5 registo final
- shamt → 5 shift amount
- funct → 6 bits código de op. a realizar

4) Não faz nada

0 tip rsl é usado quando o registo é do tipo unsigned e sra é usado quando o registo é do tipo signed.

⑥

A integer move today we have or \$4, \$0, \$15

⑦

$xor \$5, \$13, \$24$ shant funct
6 5 5 5 5 6

0000 0001 0111 1000 0010 1000 0010 0110

0x01782826

1010 - A
1011 - B
1100 - C
1101 - D
1110 - E
1111 - F

$sub \$30, \$14, \$8$ shant funct
6 5 5 5 6

0000 0001 1100 1000 1111 0000 0010 0010

Instrução	funct (Hexadecimal)	funct (Decimal)
add	0x20	32
addu	0x21	33
sub	0x22	34
subu	0x23	35
and	0x24	36
or	0x25	37
xor	0x26	38
nor	0x27	39
sll	0x00	0
srl	0x02	2
sra	0x03	3
slt	0x2A	42
sltu	0x2B	43
jr	0x08	8
jalr	0x09	9
mfhi	0x10	16
mfhi	0x11	17
mflo	0x12	18
mtlo	0x13	19
mult	0x18	24
multu	0x19	25
div	0x1A	26
divu	0x1B	27

0x01C8F022

	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰
2	1	1	0	0	0	1
32	0	1	0	1	1	
13	0	0	1	0	1	
5	1	1	0	1	0	
26	1	1	0	1	0	
38	1	0	0	1	1	0
34	1	0	0	0	1	0
30	1	1	1	1	1	0

$sll \$3, \$9, 7$

0000b0 09b00 01001 100011 00111 09b00
OP rn rt rd shant

0000 0000 0000 0011 0100 1001 1100 0000

0x000919C0

$sra \$18, \$9, 8$

0000b0 09b00 01001 1001b 010b0 09b011
OP rn rt rd shant funct

0x00099203

8

$$y = \$t5 \quad x = \$t4$$

add \$ts, \$t4, \$t4 # $2^* x = y$

add \$ts, \$ts, \$t4 # $3^* x = y$

sub \$ts, \$0, \$ts # $-3^* x = y$

add \$ts, \$ts, 5 # $y = -3^* x + 5$

9

a: \$t0 b: \$t1 c: \$t2

x: \$c0 y: \$c1 z: \$c2

srsl \$c2, \$c0, 2

add \$c2, \$c2, \$c1

add \$t1, \$t1, \$t1 # $2^* b$

sub \$t1, \$0, \$t1 # $-2^* b$

srsl \$t2, \$t0, 5

add \$t2, \$t2, \$t1 # $c = c \gg 5 - 2^* b$