

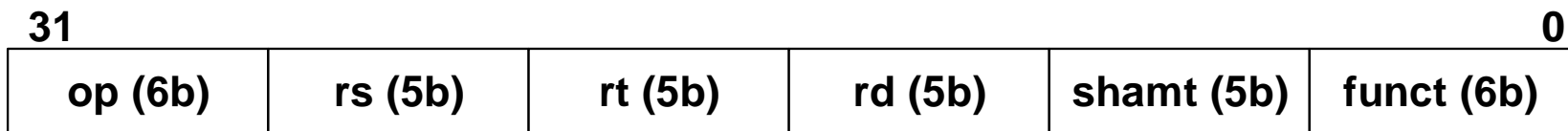
Aula 8

- Métodos de endereçamento em saltos condicionais e incondicionais
- Codificação das instruções de salto condicional no MIPS
- Codificação das instruções de salto incondicional no MIPS: o formato J
- Endereçamento imediato e uso de constantes
- Resumo dos modos de endereçamento do MIPS

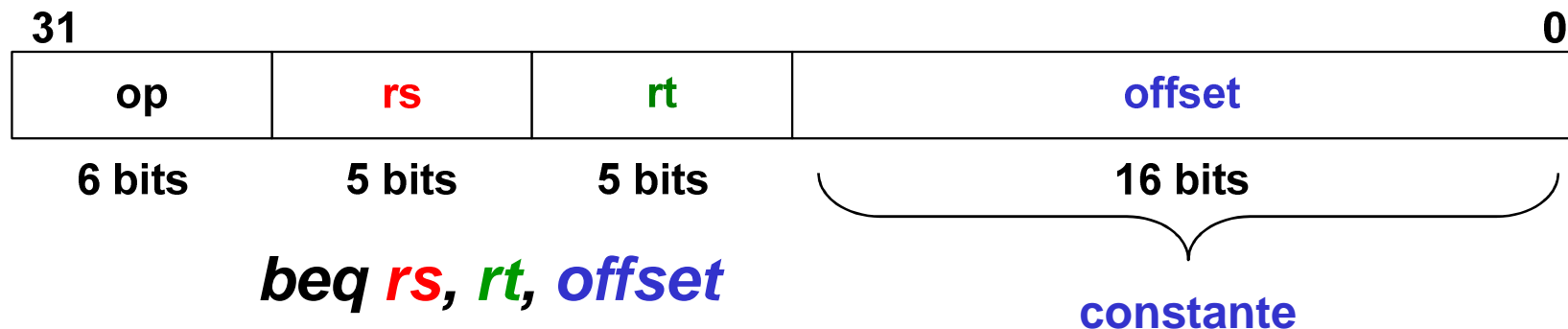
Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Codificação de *branches* no MIPS

- As instruções aritméticas e lógicas no MIPS são codificadas no **formato R**



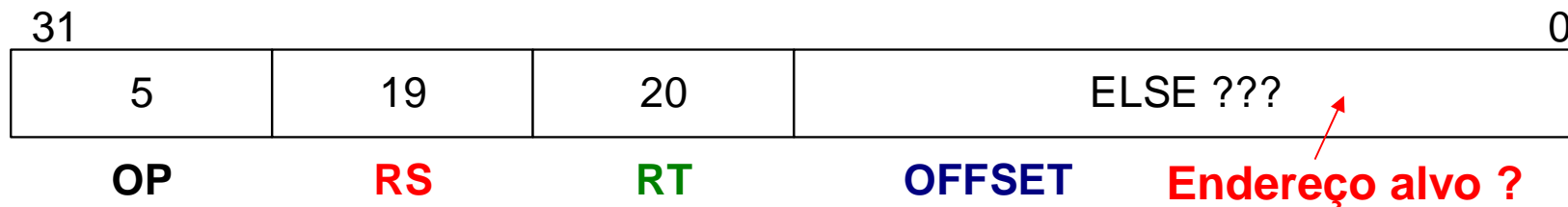
- A necessidade de codificação do **endereço-alvo** da instrução de salto obriga a que estas instruções sejam codificadas recorrendo ao **formato I**



No MIPS todas as instruções são armazenadas em endereços múltiplos de 4 (e.g. 0x00400000, 0x00400004, 0x00400008, ...)

Codificação de *branches*

Exemplo: **beq \$19, \$20, ELSE** # "ELSE" representa o endereço-alvo



- Se o endereço alvo fosse codificado diretamente nos 16 bits menos significativos da instrução, isso significaria que o programa não poderia ter uma dimensão superior a 2^{16} (64K)...
- Em vez de um endereço absoluto, o campo *offset* pode ser usado para codificar a diferença entre o valor do endereço-alvo e o endereço onde está armazenada a instrução de *branch*
- Durante a execução da instrução de *branch* o seu endereço está disponível no registo PC, pelo que o processador pode calcular o endereço-alvo como: **Endereço-alvo = PC + offset**
- O *offset* é interpretado como um valor em complemento para dois, permitindo o salto para endereços anteriores (*offset* negativo) ou posteriores (*offset* positivo) ao PC
- Endereçamento relativo (PC-relative addressing)

Codificação de *branches* no MIPS

- No MIPS, na fase de execução de um *branch*, o PC corresponde ao endereço da instrução seguinte (o PC é incrementado na fase “*fetch*” da instrução)
- Por essa razão, na codificação de uma instrução de *branch*, a referência para cálculo do *offset* é o endereço da instrução seguinte
- As instruções estão armazenadas em memória em endereços múltiplos de 4, pelo que o *offset* é também um valor múltiplo de 4 (2 bits menos significativos são sempre 0)
- De modo a otimizar o espaço disponível para o *offset* na instrução, os dois bits menos significativos não são representados
- Assim, o endereço-alvo (novo PC) é calculado como:

$$\text{Novo_PC} = \text{PC_atual} + (\text{offset} \ll 2)$$

(o *offset* de 16 bits é estendido com sinal para 32 bits antes do *shift*)

Codificação de *branches* no MIPS

Considere-se o seguinte exemplo:

```
0x00400000    bne    $19, $20, ELSE
0x00400004    add    $16, $17, $18
0x00400008    j      END_IF
0x0040000C    ELSE:  sub    $16, $16, $19
0x00400010    END_IF:
```

Durante o *instruction fetch*
o PC é incrementado
(i.e. PC=0x00400004)

O endereço correspondente ao
label ELSE é 0x0040000C

O "offset" seria portanto:
 $ELSE - [PC] =$
 $0x0040000C - 0x00400004 = 0x08$

No entanto, como **cada instrução ocupa sempre 4 bytes** na memória (a partir de um endereço múltiplo de 4), o "offset" é também múltiplo de 4 Logo:

"offset" = $0x08 / 4 = 0x02$ (**offset em número de instruções!!!**)

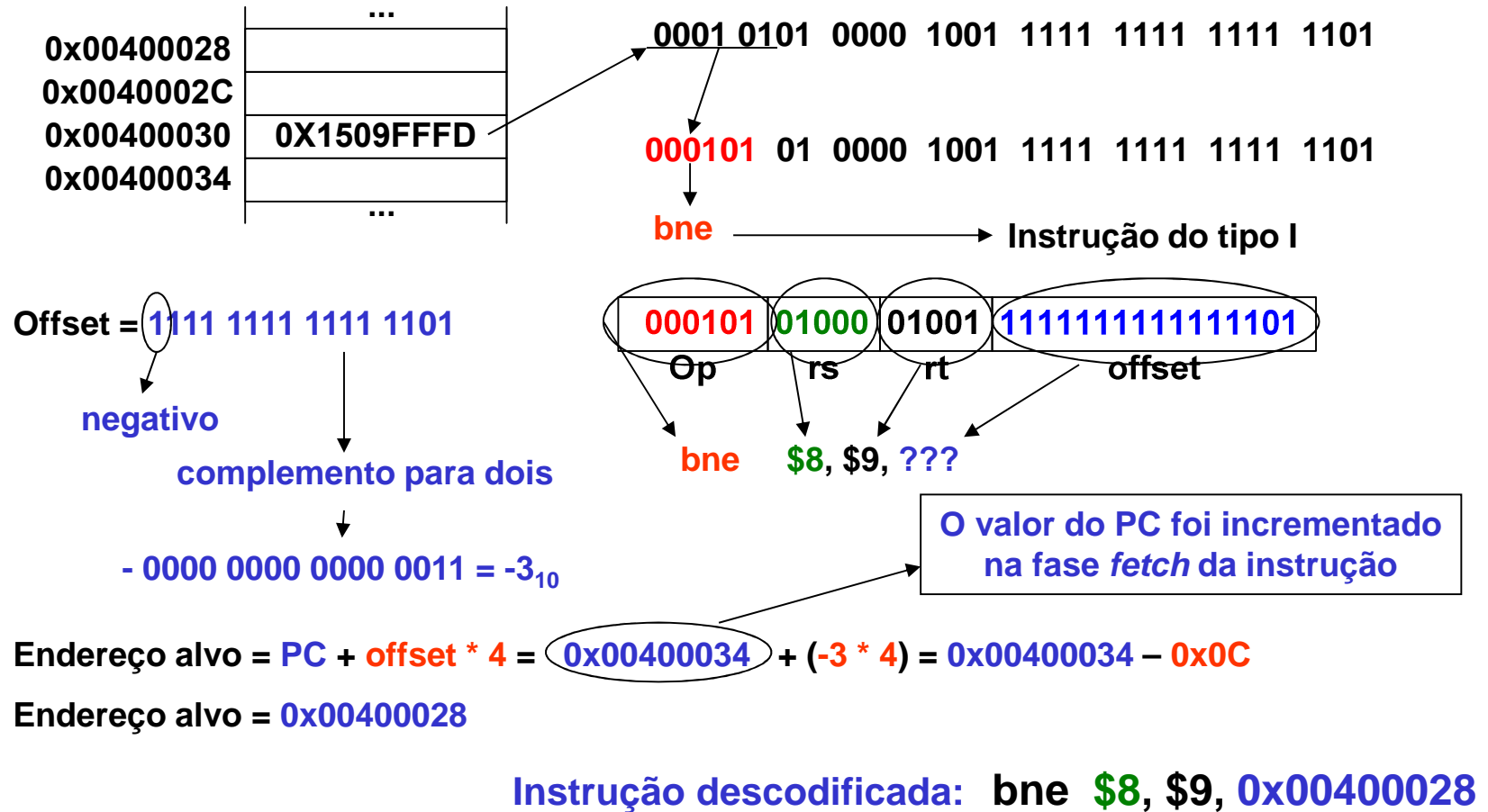
31					0
5	19	20		0x0002	

Código máquina: **00010110011101000000000000000010** = **0x16740002**

Uma instrução de salto condicional pode referenciar qualquer endereço de uma outra instrução que se situe até **32K instruções** antes ou depois dela própria.

Interpretação de uma instrução de *branch* no MIPS

Exemplo



Codificação da instrução de salto incondicional

- No caso da instrução de salto incondicional (" j "), é usado **endereçamento pseudo-direto**, i.e. o **código máquina** da instrução **codifica diretamente parte do endereço alvo** (múltiplo de 4)

- Exemplo:** a instrução `j Label #se Label=0x001D14C8`

será codificada como:

$$0x001D14C8 / 4 = 0x00074532$$



↑ 6 bits

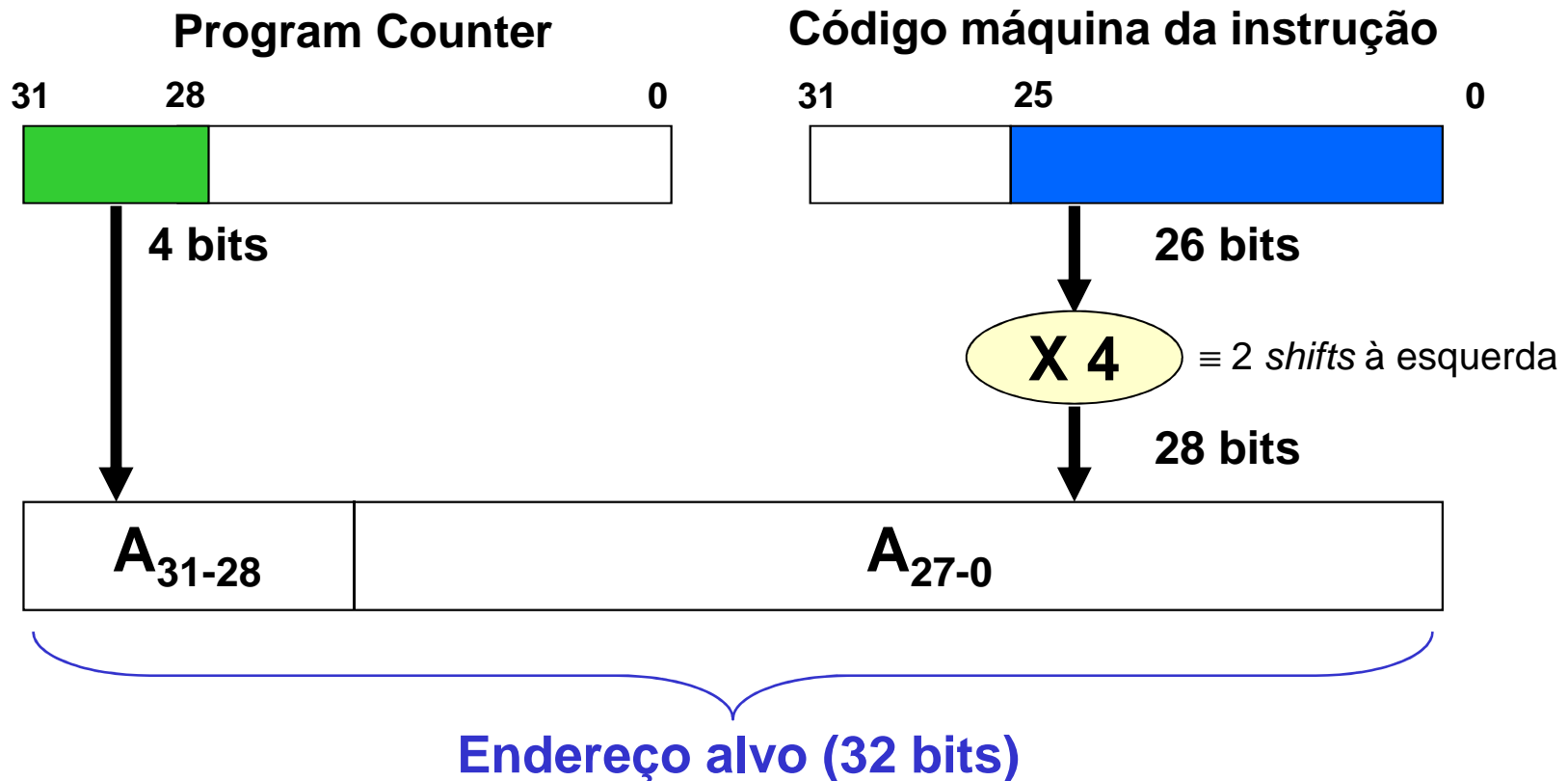
26 bits

Formato J

Código Máquina: **00001000000001110100010100110010**₂ = 0x08074532

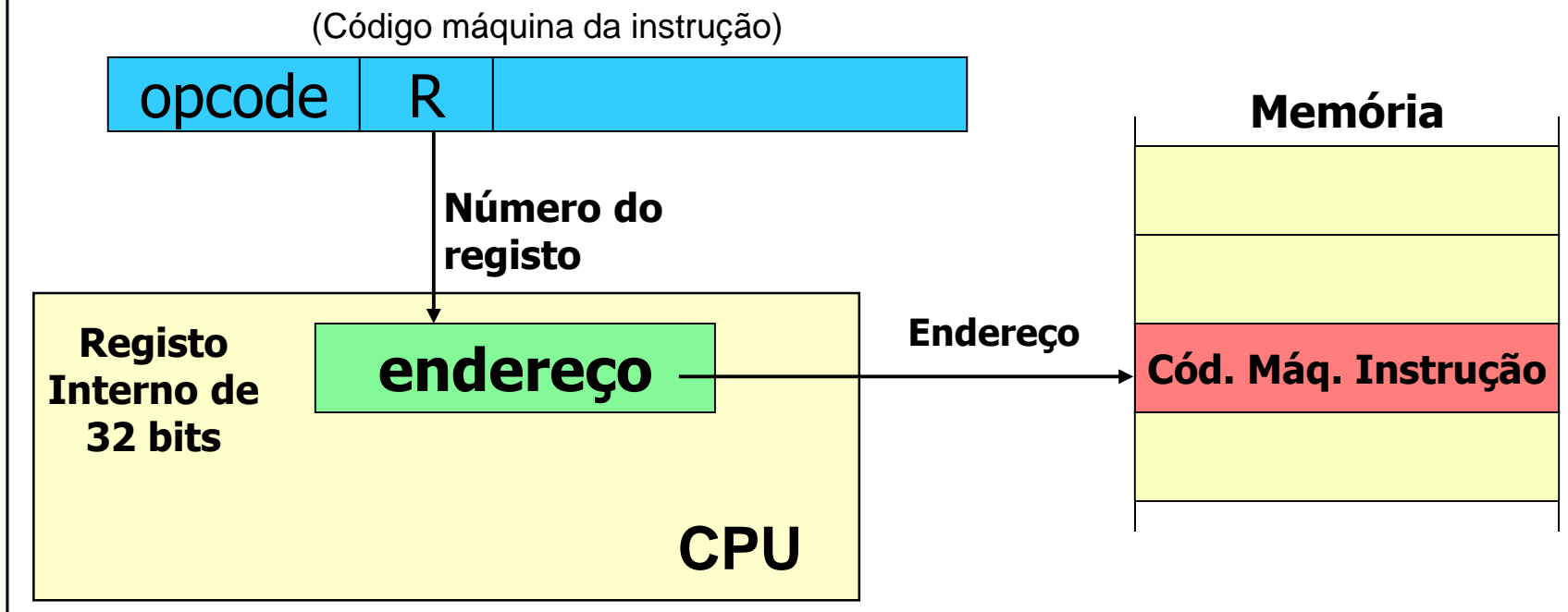
Cálculo do endereço-alvo de uma instrução J

Se a instrução só codifica 28 bits (26 explícitos + 2 implícitos),
como é formado o endereço final de 32 bits?



Salto incondicional – endereçamento indireto por registo

- Haverá maneira de especificar, numa instrução de salto incondicional, um endereço-alvo de 32 bits?
- Há! Utiliza-se **endereçamento indireto por registo**. Ou seja, um registo interno (de 32 bits) armazena o endereço alvo da instrução de salto (**instrução JR**)



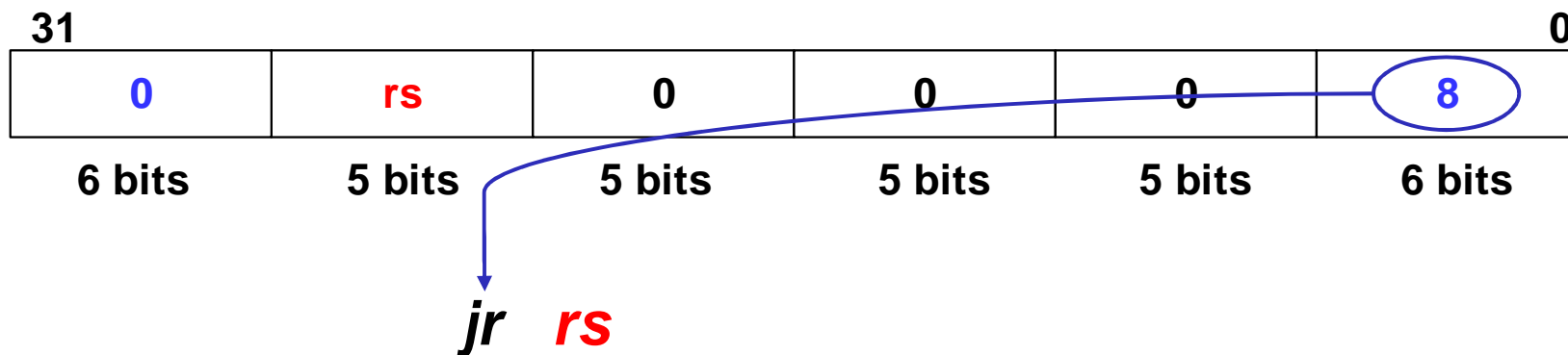
Instrução JR (jump on register)

jr **Rsrc** # salta para o endereço que
 # se encontra armazenado no registo Rsrc

Exemplo:

jr **\$ra** # Salta para o endereço que está
 # armazenado no registo \$ra

O formato de codificação da instrução JR é o formato R:



Manipulação de constantes

- Constante é um valor determinado com antecedência (quando o programa é escrito) e que não se pretende que seja ou possa ser mudado durante a execução do programa
- As constantes poderiam ser armazenadas na memória externa. Nesse caso, a sua utilização implicaria sempre o recurso a duas instruções:
 - leitura do valor residente em memória para um registo interno
 - operação com essa constante
- Para aumentar a eficiência, as arquiteturas disponibilizam um conjunto de instruções em que as **constantes se encontram armazenadas na própria instrução**
- Desta forma o acesso à constante é “**imediato**”, sem necessidade de recorrer a uma operação prévia de leitura da memória: “**endereço imediato**”

Manipulação de constantes no MIPS

- As instruções aritméticas e lógicas que manipulam constantes (do tipo imediato) são identificadas pelo sufixo “i”:

<code>addi \$3,\$5,4</code>	<code># \$3 = \$5 + 0x0004</code>
<code>andi \$17,\$18,0x3AF5</code>	<code># \$17 = \$18 & 0x3AF5</code>
<code>ori \$12,\$10,0x0FA2</code>	<code># \$12 = \$10 0x0FA2</code>
<code>slti \$2,\$12,16</code>	<code># \$2 = 1 se \$12 < 16</code>
	<code># (\$2 = 0 se \$12 ≥ 16)</code>

- Estas instruções são codificados usando o **formato I**. Logo apenas **16 bits** podem ser usados para codificar a constante
- Este espaço é geralmente suficiente para armazenar as constantes mais frequentemente utilizadas (geralmente valores pequenos)
- Se há apenas 16 bits dedicados ao armazenamento da constante, qual será a **gama de representação** dessa constante?
 - Depende da instrução...

Manipulação de constantes no MIPS

- No caso mais geral, a constante representa uma quantidade inteira, positiva ou negativa, codificada em **complemento para dois**. É o caso das instruções:

```
addi $3, $5, -4      # equivalente a 0xFFFC
addi $4, $2, 0x15    # 2110
slti $6, $7, 0xFFFF # -110
```

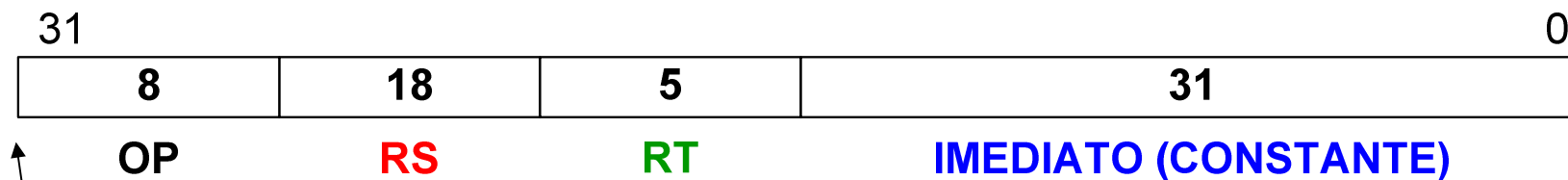
- Gama de representação da constante: **[-32768, +32767]**
 - A constante de 16 bits é extendida para 32 bits, preservando o sinal (para a constante -4, o valor do operando é **0xFFFFF**)
- Existem também instruções em que a constante deve ser entendida como uma quantidade inteira sem sinal. Estão neste grupo todas as instruções lógicas:

```
andi $3, $5, 0xFFFF
```

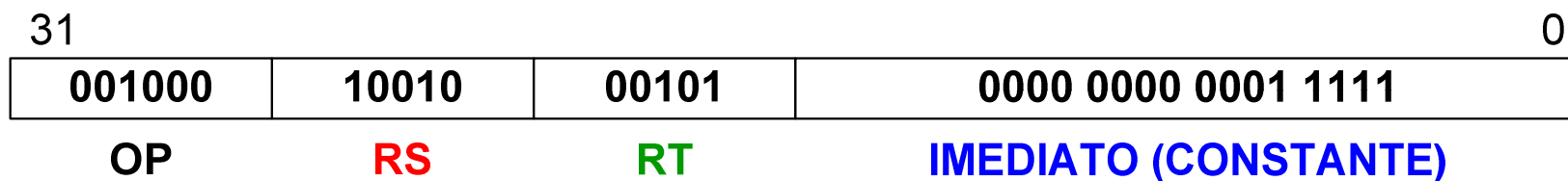
- Gama de representação da constante: **[0, 65535]**
- A constante de 16 bits é extendida para 32 bits, sendo os 16 mais significativos **0x0000** (para o exemplo: **0x0000FFFF**)

Codificação das instruções que usam constantes

Exemplo: **addi \$5, \$18, 31**



addi *rt*, *rs*, *immediate*



Cod. Máquina: 00100010010001010000000000011111 = 0x2245001F

Manipulação de constantes de 32 bits – LUI

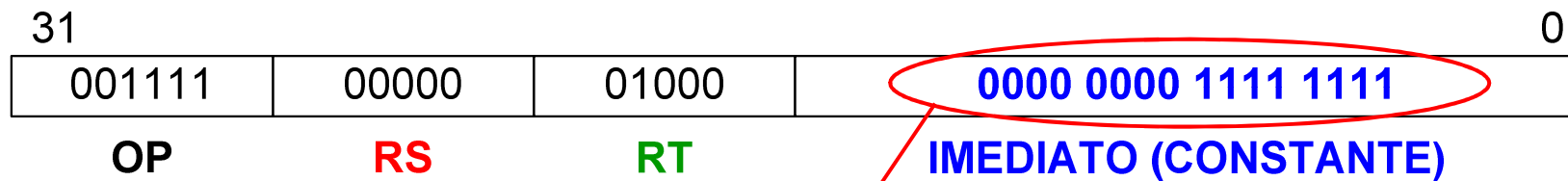
- Em alguns casos pode ser necessário manipular constantes que necessitem de um espaço de armazenamento com mais do que 16 bits (e.g., a referência explícita a um endereço)
- Como lidar com esses casos?
- Para facilitar a manipulação de constantes com mais de 16 bits, o ISA do MIPS inclui a seguinte instrução, também codificada com o formato I:

lui \$reg, immediate

- A instrução **lui** ("Load Upper Immediate"), coloca a constante "immediate" nos **16 bits mais significativos do registo destino** (\$reg)
- Os 16 bits menos significativos ficam com **0x0000**

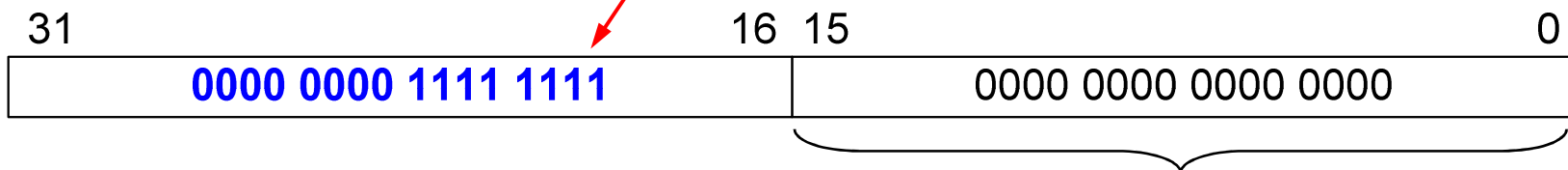
Manipulação de constantes de 32 bits – LUI

Exemplo: `lui $8, 255` # $255_{10} = 0xFF$



lui *rt, immediate*

Conteúdo do registo \$8 após a execução da instrução:



Valor que fica armazenado
em \$8 = $0x00FF0000$

Os 16 bits menos significativos
ficam com o valor 0

Manipulação de constantes de 32 bits – LA / LI

A instrução virtual "load address"

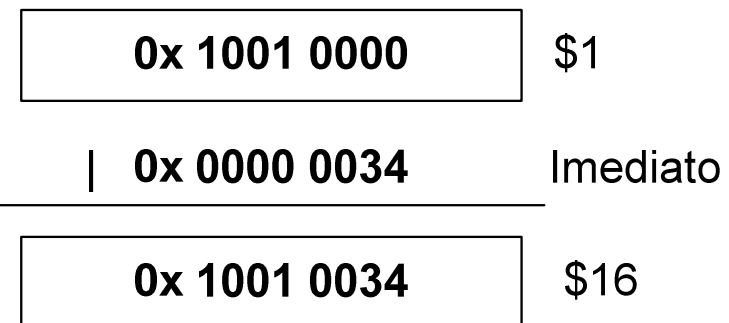
```
la    $16, MyData # Ex. MyData = 0x10010034
                        # Segmento de dados em 0x1001000
```

é executada no MIPS pela sequência de **instruções nativas**:

```
lui   $1, 0x1001      # $1 = 0x10010000
ori   $16, $1, 0x0034 # $16 = 0x10010000 | 0x00000034
```

Notas:

- O **registro \$1 (\$at)** é reservado para o *Assembler*, para permitir este tipo de decomposição de **instruções virtuais** em **instruções nativas**.
- A instrução “li” (*load immediate*) é decomposta em instruções nativas de forma análoga à instrução “la”

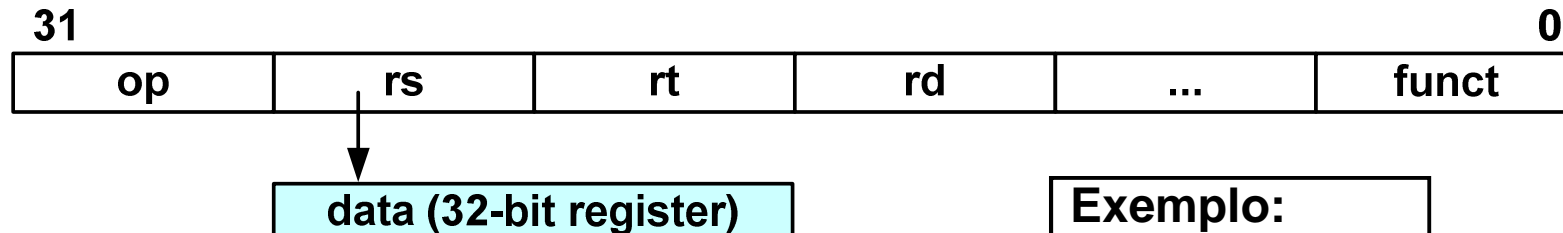


Modos de endereçamento no MIPS (resumo)

- Instruções aritméticas e lógicas: **endereçamento tipo registo**
- Instruções aritméticas e lógicas com constantes: **endereçamento imediato**
- Instruções de acesso à memória: **endereçamento indireto por registo com deslocamento**
- Instruções de salto condicional (*branches*): **endereçamento relativo ao PC**
- Instrução de salto incondicional através de um registo (instrução **JR**): **endereçamento indireto por registo**
- Instrução de salto incondicional (**J**): **endereçamento direto** (uma vez que o endereço não é especificado na totalidade, esse tipo de endereçamento é normalmente designado por "**pseudo-direto**")

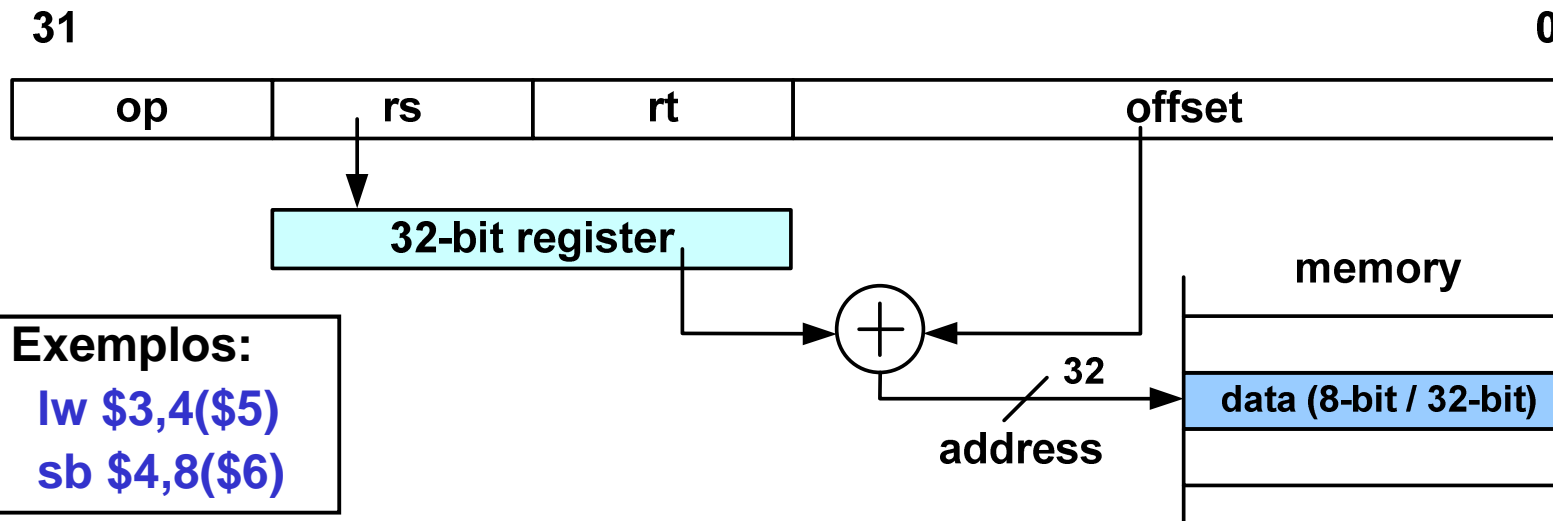
Modos de endereçamento do MIPS (resumo)

- Register Addressing (endereçamento tipo registro):



Exemplo:
add \$3,\$4,\$5

- Base addressing (indireto por registro com deslocamento):



Exemplos:
lw \$3,4(\$5)
sb \$4,8(\$6)

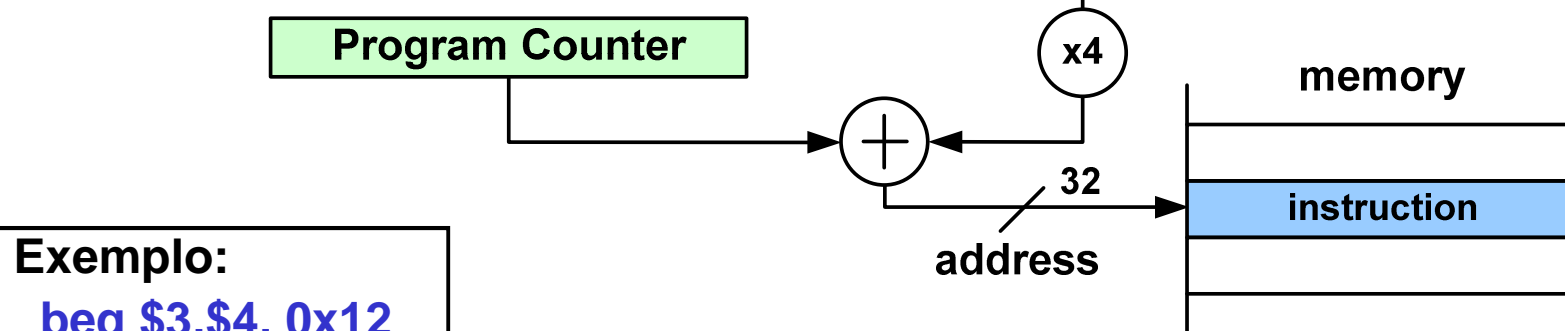
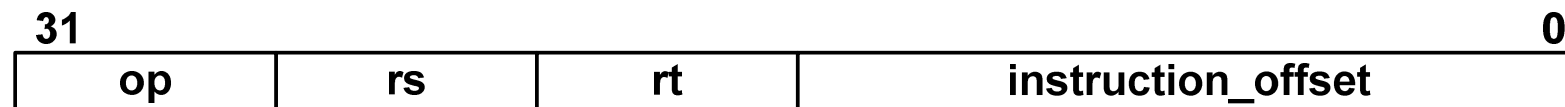
Modos de endereçamento do MIPS (resumo)

- Immediate Addressing (endereçamento imediato):



Exemplo:
`addi $3,$4,0x3F`

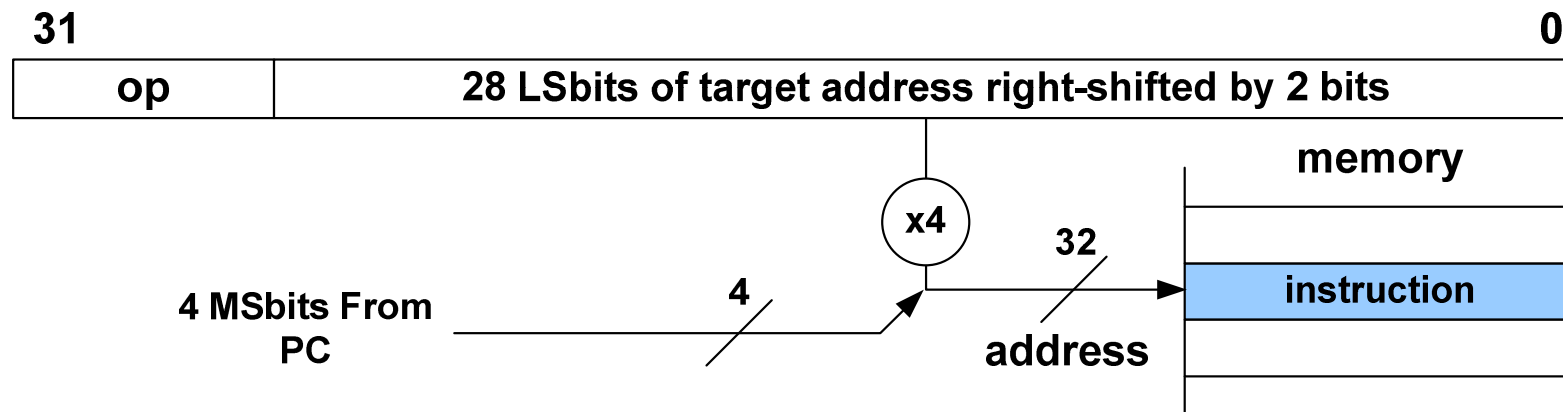
- PC-relative Addressing (endereçamento relativo ao PC):



Exemplo:
`beq $3,$4, 0x12`

Modos de endereçamento do MIPS (resumo)

- Pseudo-direct Addressing (endereço pseudo-direto):



Exemplos:

j 0x0010000B # target address is 0x0040002C
jal 0x0010048E # target address is 0x00401238

(target calculado supondo que PC = 0x0...)

Questões / exercícios

- Qual o formato de codificação de cada uma das seguintes instruções: "**beq/bne**", "**j**", "**jr**"?
- O que é codificado no campo *offset* do código máquina das instruções "**beq/bne**" ?
- A partir do código máquina de uma instrução "**beq/bne**", como é formado o endereço-alvo (*Branch Target Address*)?
- A partir do código máquina de uma instrução "**j**", como é formado o endereço-alvo (*Jump Target Address*)?
- Na instrução "**jr \$ra**", como é obtido o endereço-alvo?
- Qual o endereço mínimo e máximo para onde uma instrução "**j**", residente no endereço de memória **0x5A18F34C**, pode saltar?
- Qual o endereço mínimo e máximo para onde uma instrução "**beq**", residente no endereço de memória **0x5A18F34C**, pode saltar?
- Qual o endereço mínimo e máximo para onde uma instrução "**jr**", residente no endereço de memória **0x5A18F34C** pode saltar?

Questões / exercícios

- Qual a gama de representação da constante nas instruções aritméticas imediatas?
- Qual a gama de representação da constante nas instruções lógicas imediatas?
- Porque razão não existe no ISA do MIPS uma instrução que permita manipular diretamente uma constante de 32 bits?
- Como é que no MIPS se podem manipular constantes de 32 bits?
- Apresente a decomposição em instruções nativas das seguintes instruções virtuais:

```
li      $6, 0x8B47BE0F
xori    $3, $4, 0x12345678
addi    $5, $2, 0xF345AB17
beq     $7, 100, L1
blt     $3, 0x123456, L2
```