

Aula 6

- 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

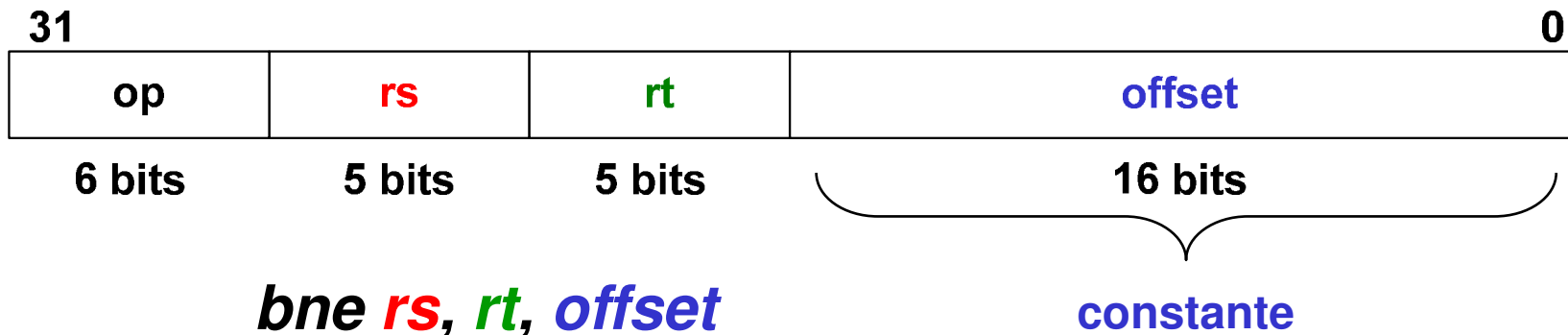
José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador

Codificação das instruções de salto condicional no MIPS

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



- Por outro lado, a necessidade de codificação do **endereço-alvo** nas instruções de salto condicional obriga a que estas instruções sejam codificadas recorrendo ao **formato I**



Codificação de *branches* no MIPS

Exemplo: **bne** \$8, \$21, Exit



↑ **OP**

RS

RT

OFFSET

Instrução do tipo I

Endereço alvo ?

bne *rs*, *rt*, *offset*

- Se o endereço alvo for codificado nos 16 bits menos significativos da instrução, isso significa que o programa não pode ter uma dimensão superior a 2^{16} (64K). Será essa uma opção realista?
- Uma alternativa mais interessante poderia passar por especificar um registo interno cujo conteúdo pudesse ser somado à constante codificada na instrução (*offset*), de tal modo que **PC = Reg + Offset**
- Desta forma a dimensão máxima do programa já poderia ser 2^{32}

Codificação de *branches* no MIPS

Note-se contudo que:

- A maioria das instruções de salto condicional realizam esse salto para a vizinhança da própria instrução
- Como exemplo verifica-se, estatisticamente, que no gcc (GNU C Compiler) quase metade de todos os saltos condicionais são para endereços correspondentes a uma gama de ± 16 instruções)
- Com 16 bits é possível endereçar 2^{16} endereços distintos (64K endereços)
- **No MIPS todas as instruções são armazenadas em endereços múltiplos de 4** (e.g. 0x00400000, 0x00400004, 0x00400008, ...)
- Qual deverá ser então o registo-base a usar?

Codificação de *branches* no MIPS

- **Solução:**

- Utilizar o registo PC (Program Counter)
- Usar **endereço relativo**: o valor do endereço alvo é calculado somando algebricamente o *offset* de 16 bits, codificado na instrução, ao valor corrente do PC (o valor do *offset* é estendido com sinal para 32 bits)
- No MIPS o valor do PC, na fase de execução de um "branch", corresponde ao endereço da instrução seguinte, uma vez que esse registo é incrementado na fase "*fetch*" da instrução
- Assim, o endereço-alvo (novo PC) é calculado como:

$$\text{Novo_PC} = \text{PC_atual} + \text{offset}$$

O *offset* de 16 bits é interpretado como um **valor em complemento para dois**, permitindo o salto para **endereços anteriores (offset negativo) ou posteriores (offset positivo)** ao PC

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 "branch_offset" seria portanto:
$$\text{ELSE} - [\text{PC}] =$$
$$0x0040000C - 0x00400004 = 0x08$$

No entanto, como **cada instrução ocupa sempre 4 bytes** na memória (a partir de um endereço múltiplo de 4), o "branch_offset" é **calculado em instruções**. Logo:

$$\text{"branch_offset"} = 0x08 / 4 = 0x02$$

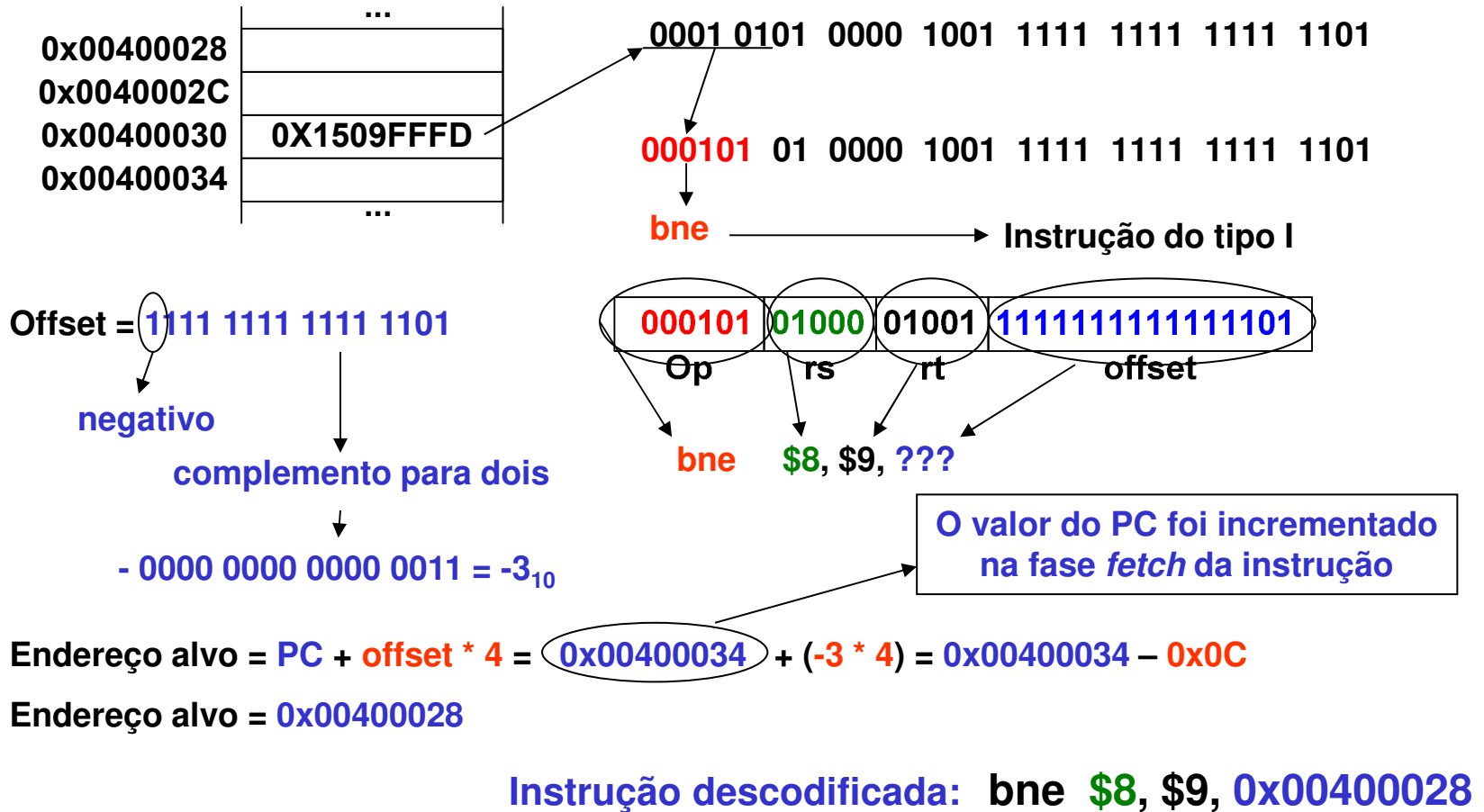
31				0
5	19	20	0x0002	

Código máquina: **0001011001110100000000000000010** = **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* pelo CPU

Exemplo



Codificação da instrução de salto incondicional no MIPS

- 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** (endereço do qual será lida a próxima instrução)

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

será codificada como:

$$0x001D14C8 / 4 = 0x00074532$$



↑ 6 bits

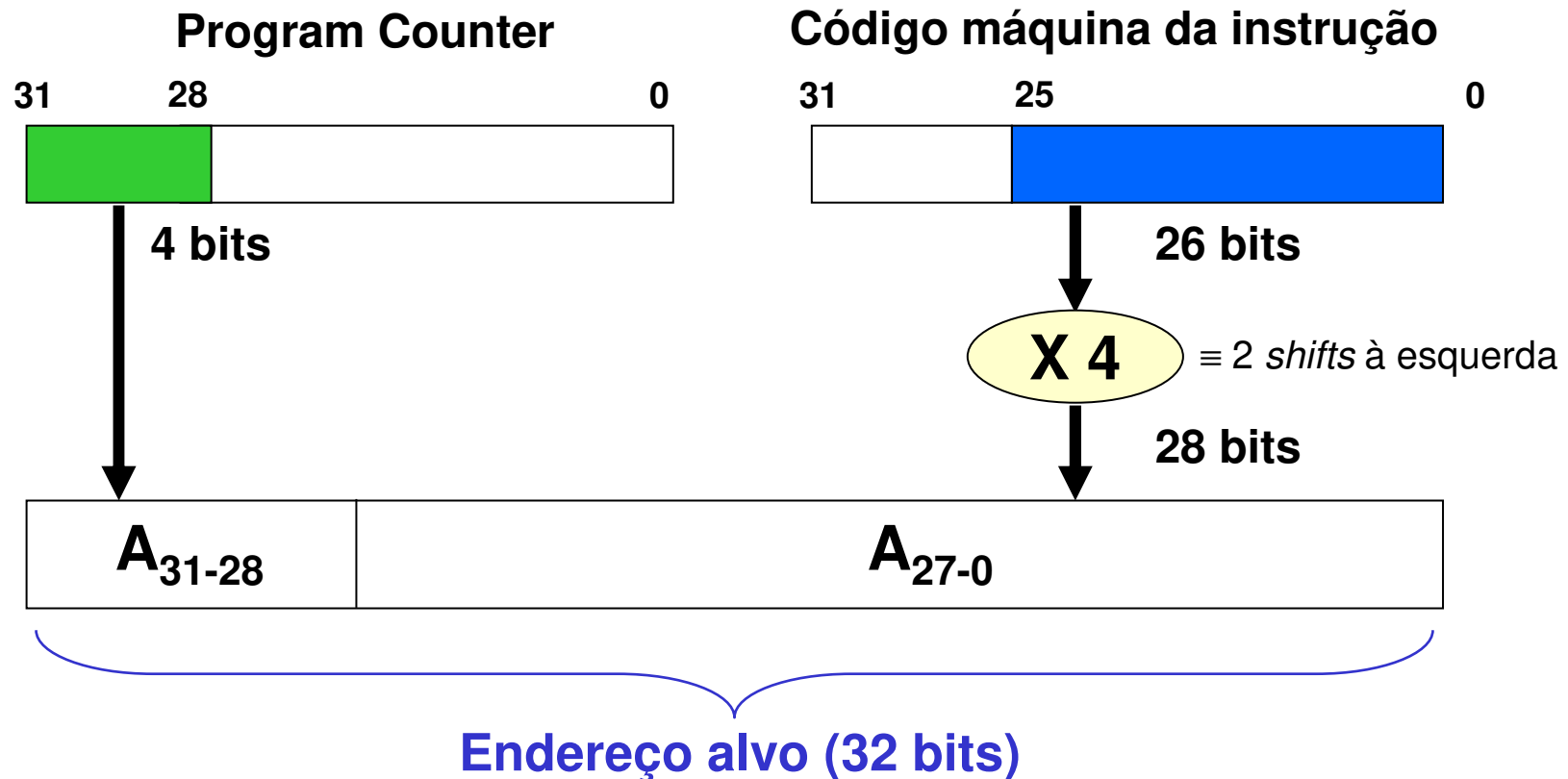
26 bits

Formato J

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

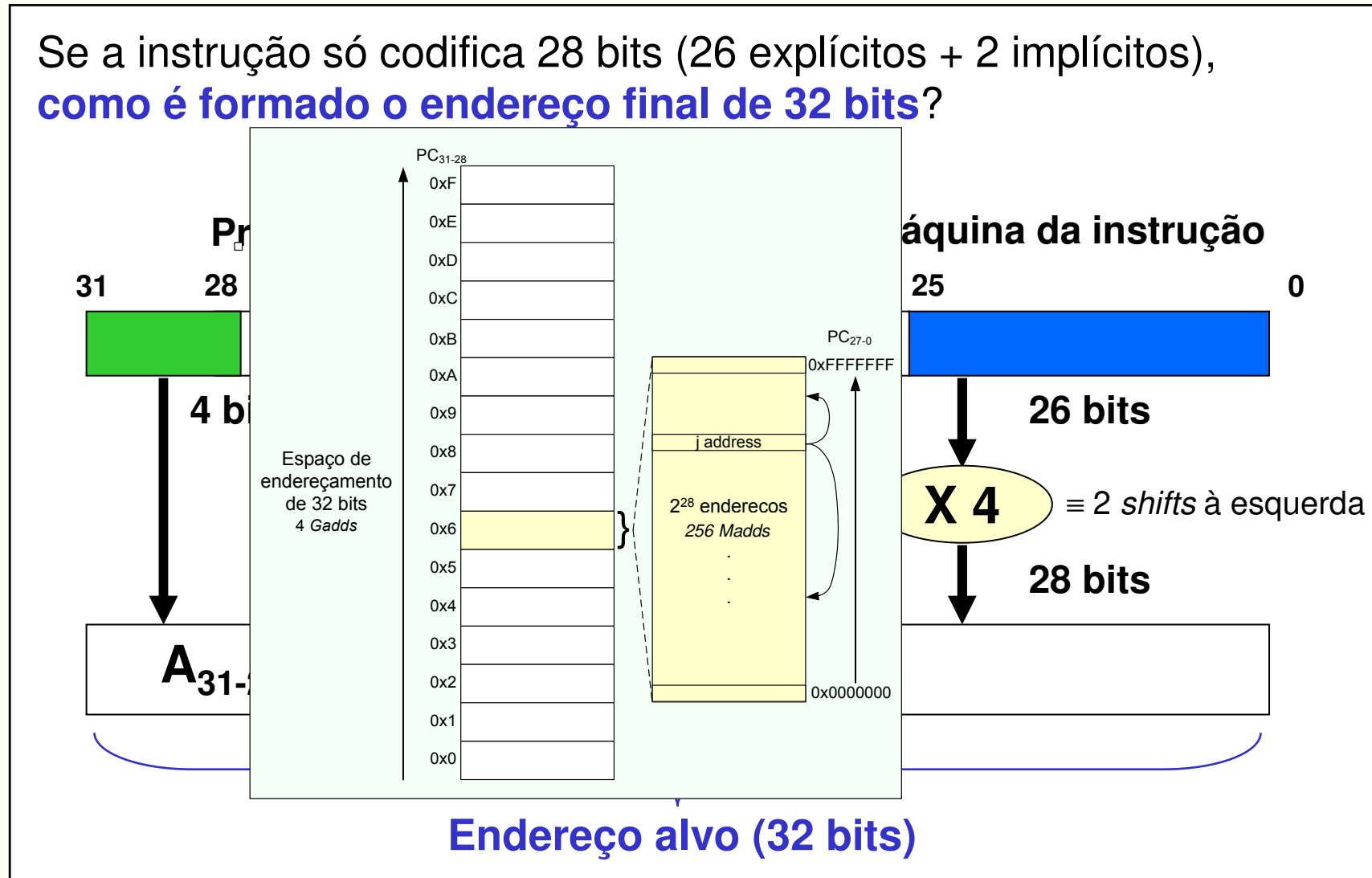
Cálculo, no CPU, do endereço-alvo de uma instrução J

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



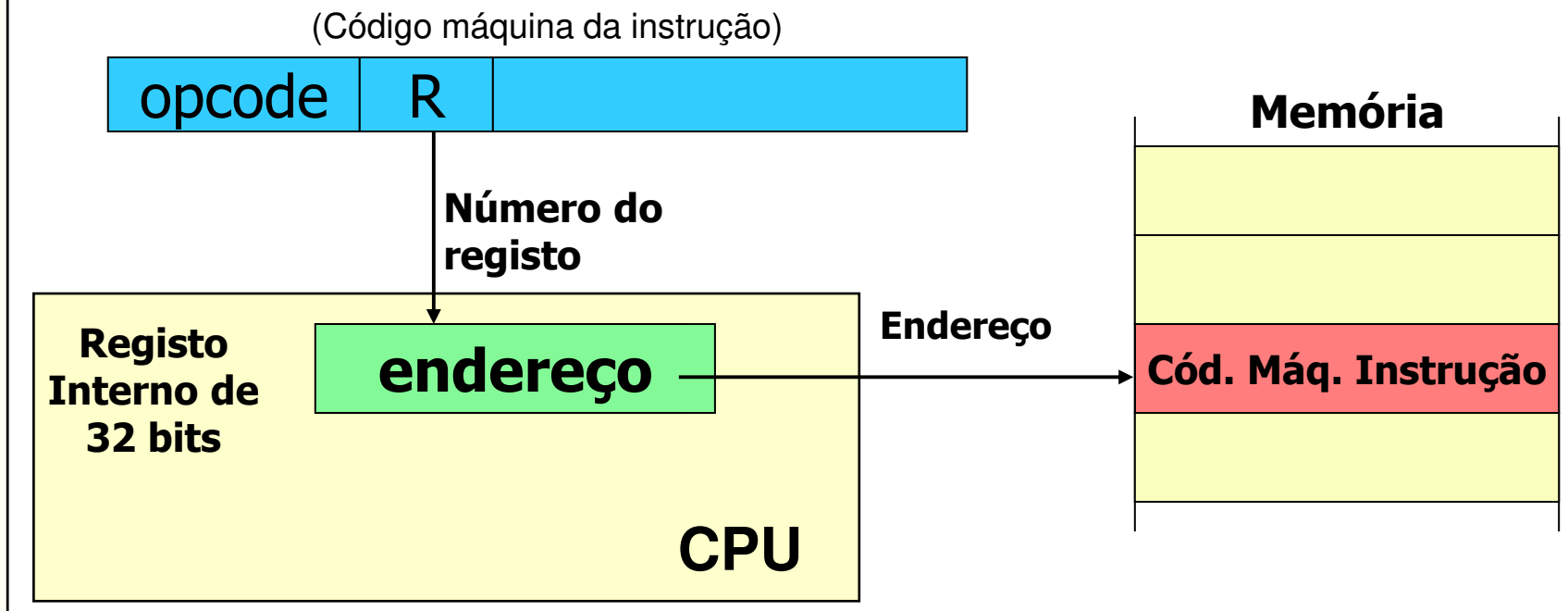
Cálculo, no CPU, do endereço-alvo de uma instrução J

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**)



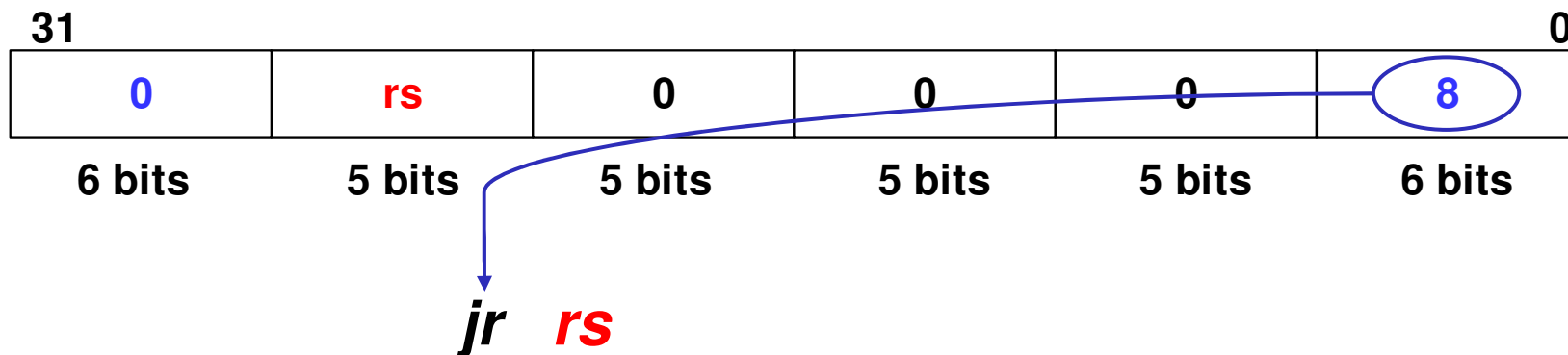
Instrução JR (jump on register)

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:



Modos de endereçamento no MIPS

- Instruções aritméticas e lógicas: **endereçamento tipo registo**
- Instruções de acesso à memória : **endereçamento indireto por registo com deslocamento**
- Instruções de salto condicional: **endereçamento relativo ao PC**
- Na instrução de salto incondicional através de um registo (instrução **JR**) é usado um registo interno do processador para armazenar o endereço-alvo da instrução de salto: **endereçamento indireto por registo**
- Na instrução de salto incondicional (**J**) é usado **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 no MIPS

- Para além dos modos de endereçamento registo e indireto por registo, o MIPS suporta ainda um outro tipo de endereçamento, designado por “**endereçamento imediato**”.
- Relembrando os quatro princípios básicos subjacentes ao design de uma arquitetura
 - A simplicidade favorece a regularidade
 - Quanto mais pequeno mais rápido
 - Um bom design implica compromissos adequados
 - **O que é mais comum deve ser mais rápido**
- O último ponto determina que a capacidade de tornar mais rápida a execução das operações que ocorrem mais vezes, resulta num aumento global do desempenho!

Endereçamento imediato

- Pode verificar-se, estatisticamente, que um número muito significativo de instruções em que está envolvida uma operação aritmética usa uma **constante** como um dos seus operandos
- É vulgar que este número seja superior a 50% do total das instruções que envolvem a ALU num determinado programa.
 - Chama-se **constante** a um valor determinado com antecedência (na altura em que o programa é escrito) e que não se pretende que seja ou possa ser mudado durante a execução do programa.

Endereçamento imediato

- A constante “zero” é tão usada, que o MIPS tem um registo interno onde esse valor está permanentemente disponível (\$0)
- A constante “um”, por outro lado, também é muito utilizada em operações de incremento ou decremento de variáveis de contagem usadas em ciclos
- 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

Endereçamento imediato

- Para aumentar a eficiência, as arquiteturas disponibilizam, habitualmente, 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
- No caso do MIPS as instruções aritméticas e lógicas 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>

Endereçamento imediato – gama de representação

- Se todas as instruções do MIPS ocupam um espaço de armazenamento de 32 bits, **quantos desses 32 bits são dedicados a armazenar o “valor imediato”**?
- Estas instruções são codificados usando o **formato I**. Logo a resposta é **16 bits**
- 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...

Endereçamento imediato – gama de representação

- 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
```

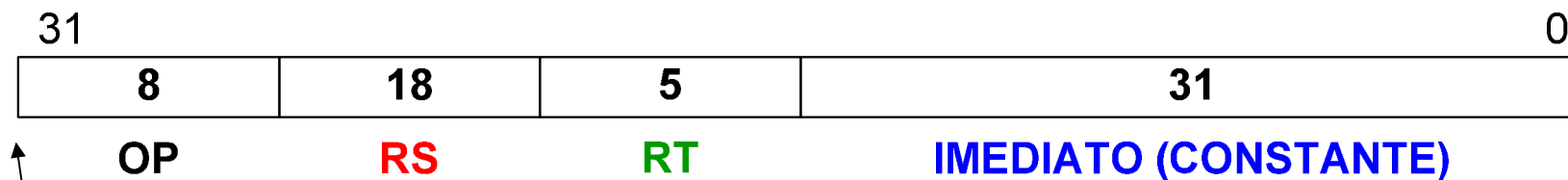
- Gama de representação da constante: **[-32768, +32767]**
 - A constante de 16 bits é extendida para 32 bits, preservando o sinal (para a constante -4, o valor do operando é **0xFFFFF**)
- Existem também instruções em que a constante deve ser entendida como uma quantidade inteira sem sinal. Estão neste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

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

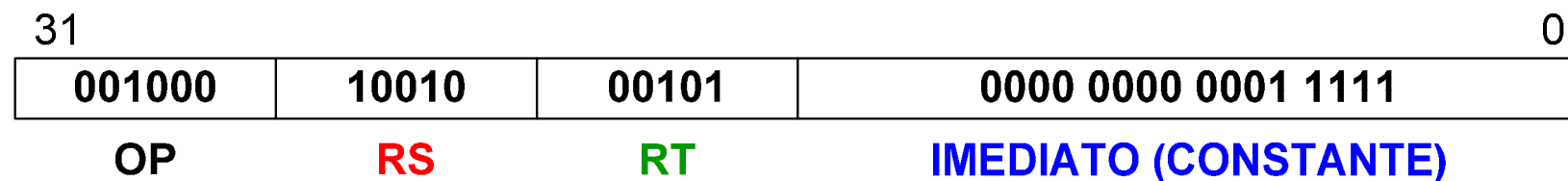
- Gama de representação da constante: **[0, 65535]**
- A constante de 16 bits é extendida para 32 bits, sendo os 16 mais significativos **0x0000** (para o exemplo: **0x0000FFFF**)

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

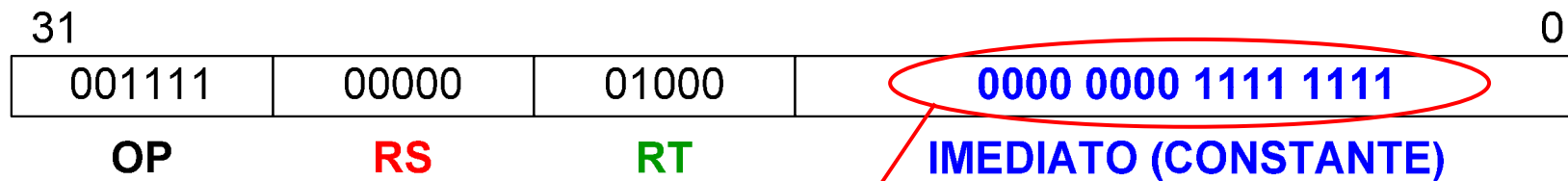
- Em alguns casos pode ser necessário manipular constantes que necessitem de um espaço de armazenamento com mais do que 16 bits (como, por exemplo, a referência explícita a um endereço). Como lidar com esses casos?
- Para facilitar a manipulação de imediatos com mais de 16 bits, o ISA do MIPS inclui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

lui \$reg, immediate

- A instrução **lui** ("Load Upper Immediate"), coloca a constante "immediate" nos **16 bits mais significativos do registo destino** (também é uma instrução do tipo I)
- Os 16 bits menos significativos ficam com **0x0000**

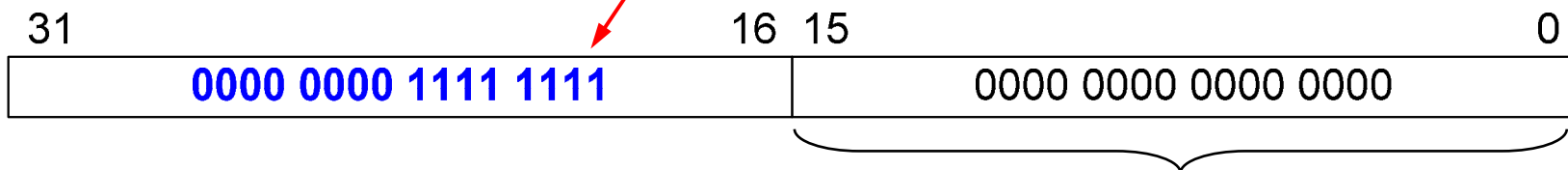
Manipulação de constantes de 32 bits – LUI

Exemplo: `lui $8, 255` # $255_{10} = 0x00FF$



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"

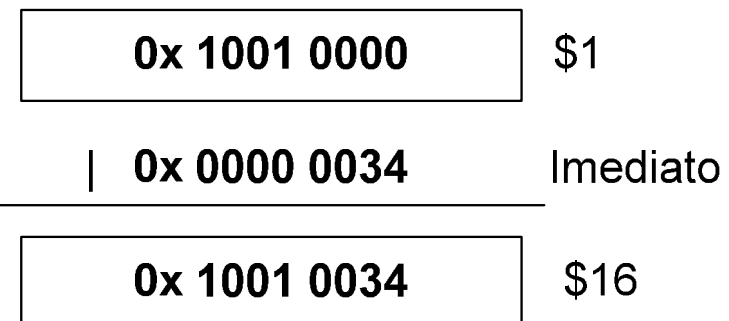
```
la    $16, MyData # Ex. MyData = 0x10010034
      # Segmento de dados em 0x1001000
```

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

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

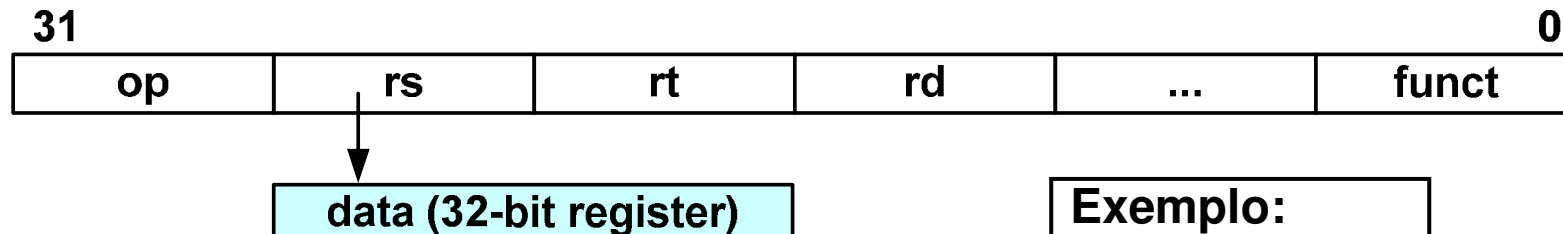
Notas:

- 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”



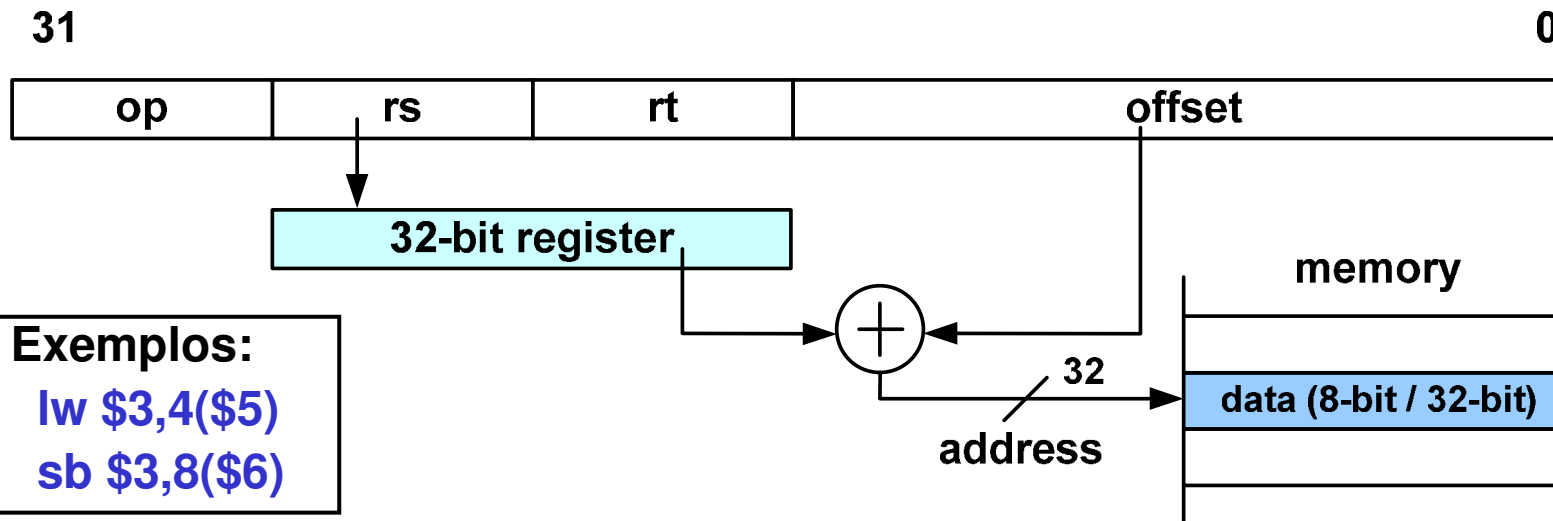
Modos de endereçamento do MIPS (resumo)

- 1 - Register Addressing (endereçamento tipo registo):



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

- 2 - Base addressing (indireto por registo com deslocamento):



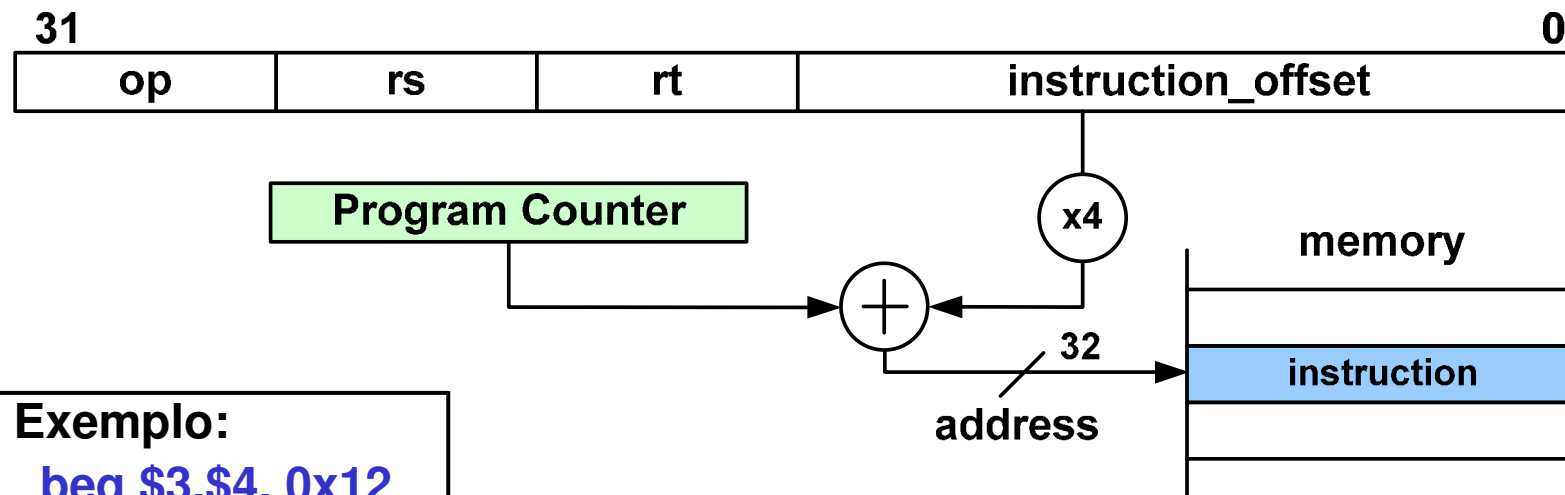
Modos de endereçamento do MIPS (resumo)

- 3 - Immediate Addressing (endereço imediato):



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

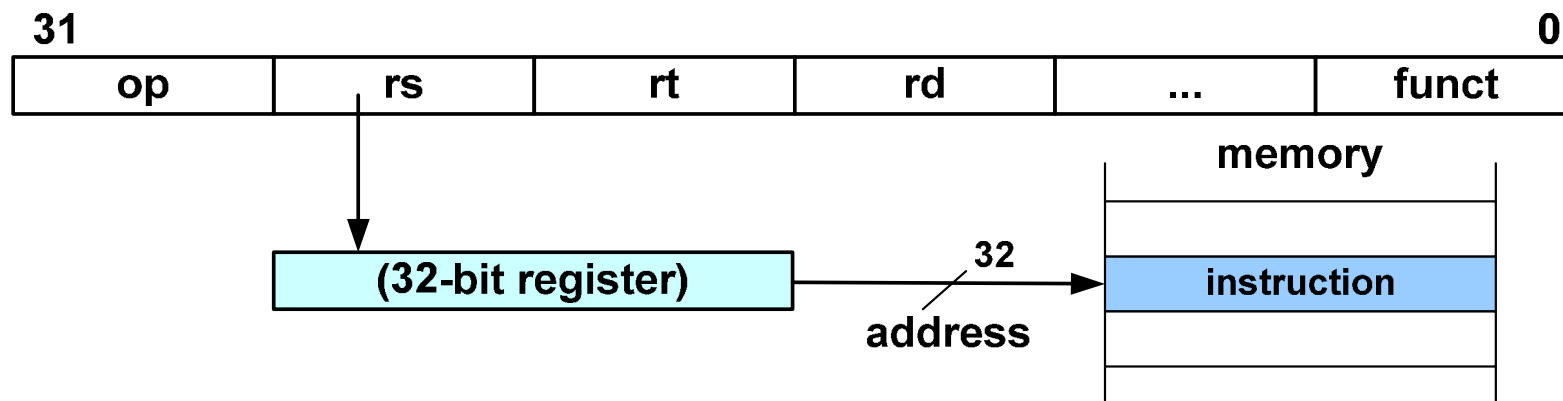
- 4 - PC-relative Addressing (endereço relativo ao PC):



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

Modos de endereçamento do MIPS (resumo)

- 5 – Indirect Register Addressing (endereçamento indireto por registro):



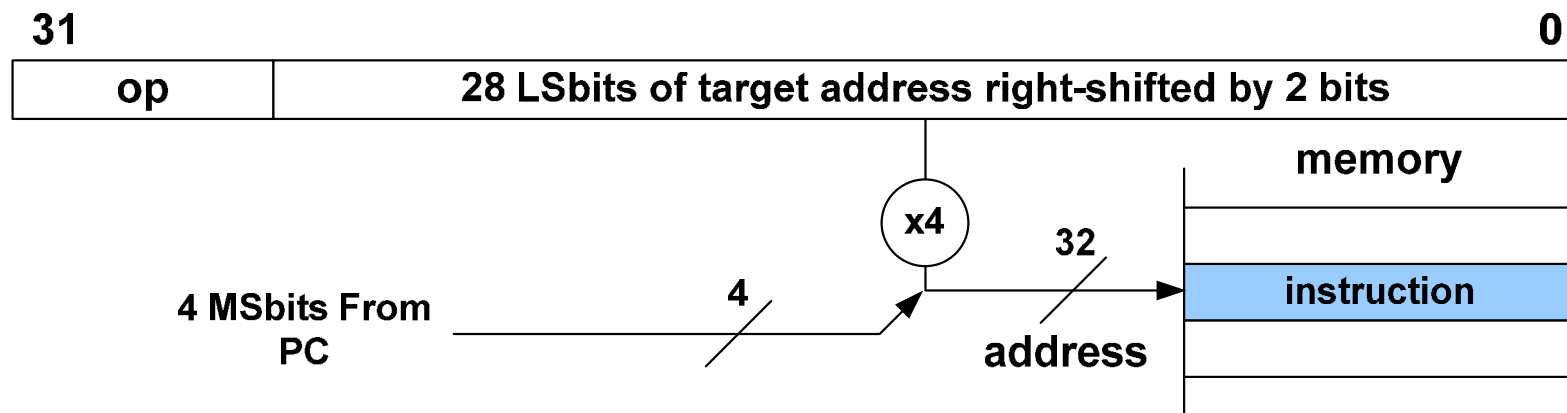
Exemplo:

jr \$ra # target address is [\$ra]



Modos de endereçamento do MIPS (resumo)

- 6 - Pseudo-direct Addressing (endereçamento pseudo-direto):



Exemplos:

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

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