Aula 4

- Acesso sequencial aos elementos de um *array*:
  - Acesso indexado; acesso com ponteiro
  - Tradução para *assembly* do MIPS
- Métodos de endereçamento em saltos condicionais e incondicionais no MIPS
- Codificação das instruções de salto:
  - Condicional (formato I)
  - Incondicional (formato J)

Bernardo Cunha, José Luís Azevedo

Acesso sequencial a elementos de um *array*

- Em C, o acesso sequencial a elementos de um *array* apoia-se em uma de duas estratégias:
  1. Acesso indexado, isto é, endereçamento a partir do nome do *array* e de um índice que identifica o elemento a que se pretende aceder:

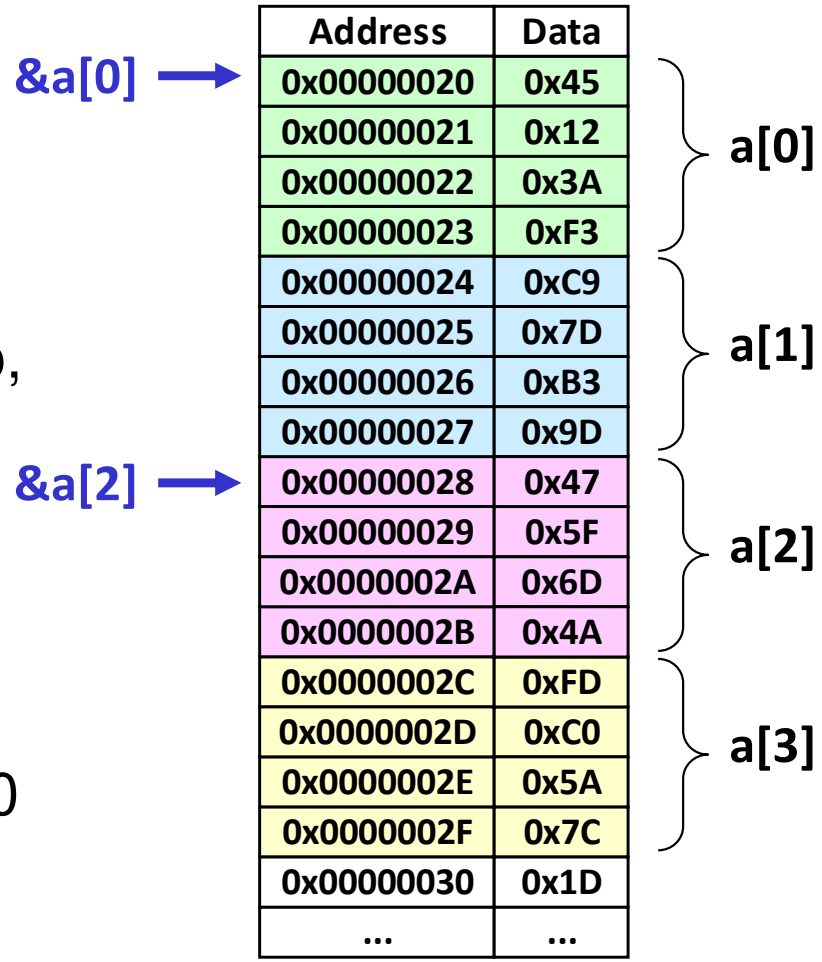
     v = a[i];
  2. Utilização de um ponteiro (endereço armazenado num registo) que identifica em cada instante o endereço do elemento a que se pretende aceder:

     v = *p; // com p = endereço de a[i] (i.e. **p = &a[i]**)

- Estas 2 formas de acesso traduzem-se em **implementações distintas** em *assembly*

Acesso sequencial a elementos de um *array*

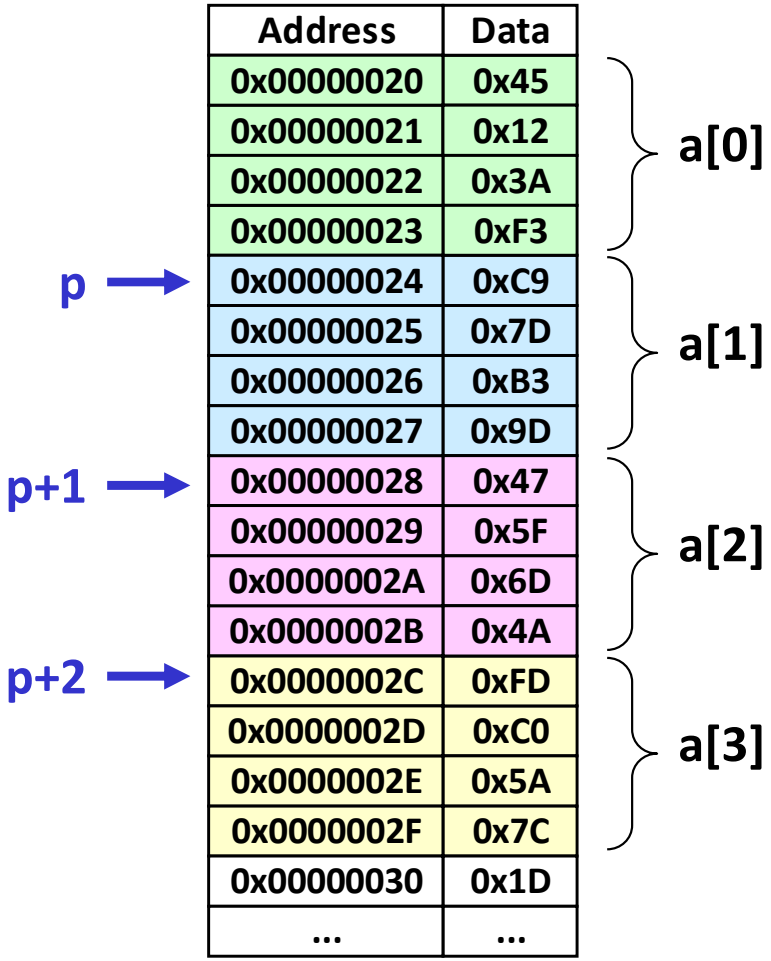
- **Acesso indexado**
  - **v = a[i];** // Com i ≥ 0
  - Para aceder ao elemento **"i"** do *array* **"a"**, o programa começa por calcular o respetivo endereço, **a partir do endereço inicial do *array***
  - Por exemplo se se tratar de um *array* de inteiros, o endereço do elemento 2 está 8 endereços à frente do endereço do elemento 0



endereço do elemento a aceder = endereço inicial do *array* + (índice * dimensão em *bytes* de cada elemento do *array*)

Acesso sequencial a elementos de um *array*

- **Acesso por ponteiro**
  - **v = *p;**
  - O endereço do elemento a aceder está armazenado num registo



endereço do elemento seguinte = endereço actual +

dimensão em *bytes* de cada elemento do *array*

Exemplos de acesso sequencial a *arrays*

```
// Exemplo 1
int i;
static int array[SIZE];

for(i = 0; i < SIZE; i++){
    array[i] = 0;
}
```

Acesso indexado

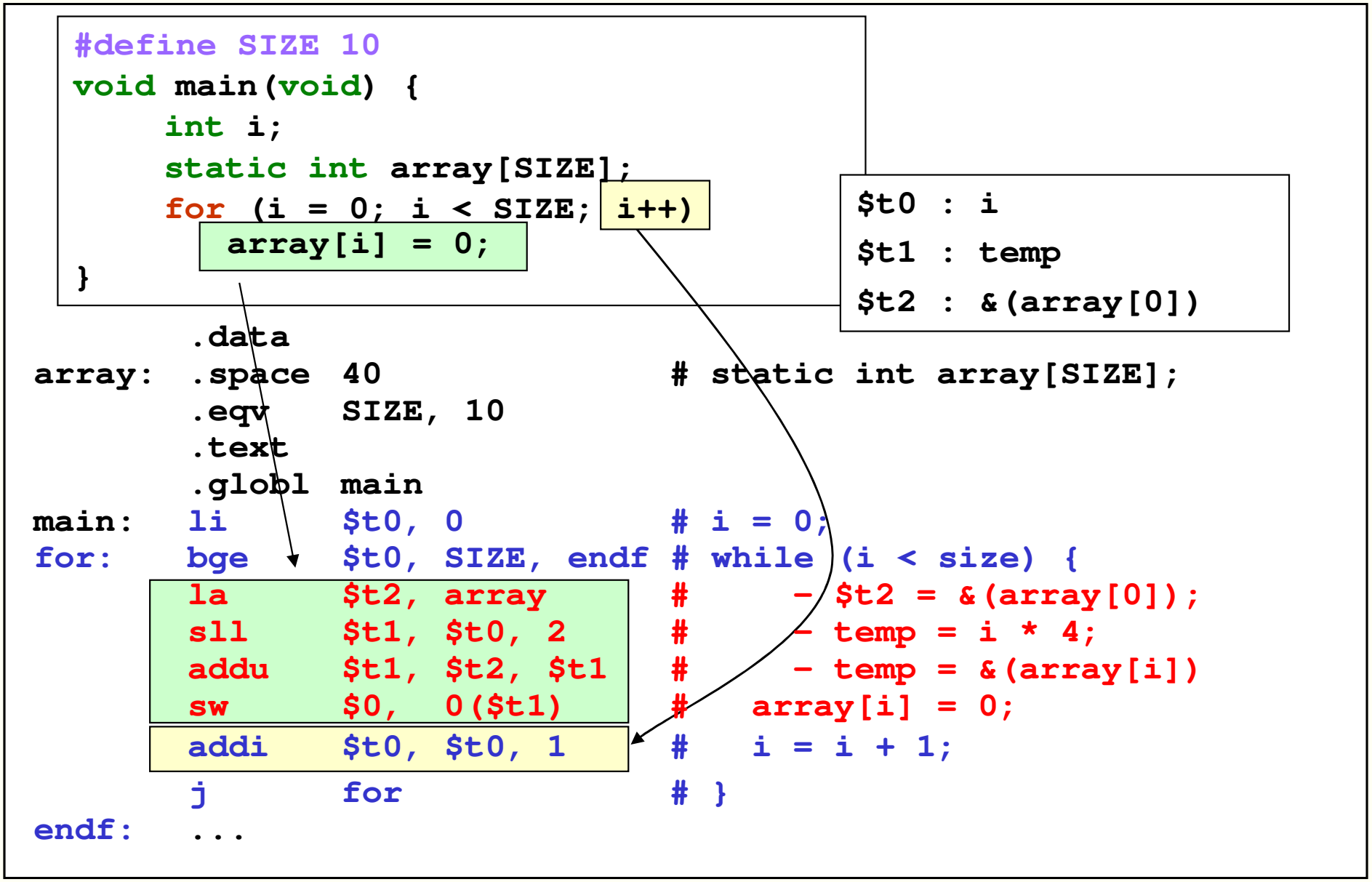
```
// Exemplo 2
int *p;
static int array[SIZE];

for(p=&array[0];p < &array[SIZE];p++)
{
    *p = 0;
}
```

Acesso por ponteiro

Também pode ser escrito como: `for(p=array; p < array+SIZE; p++)`

Acesso sequencial a *arrays* – exemplo 1



Acesso sequencial a *arrays* – exemplo 2

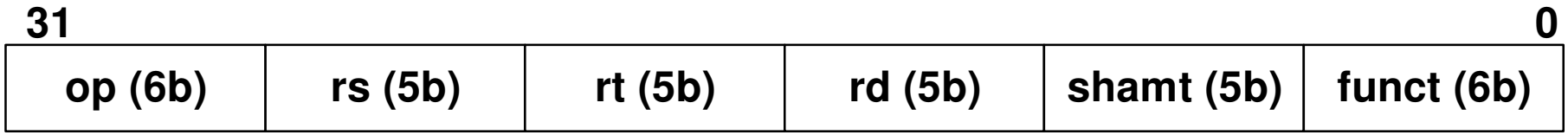
```
#define SIZE 10
void main(void){
    int *p;
    static int array[size];
    for (p=&array[0]; p < &array[size]; p++){
      *p = 0;
}
```

```
$t0 : p
$t1 : &(array[size])
$a0 : size
```

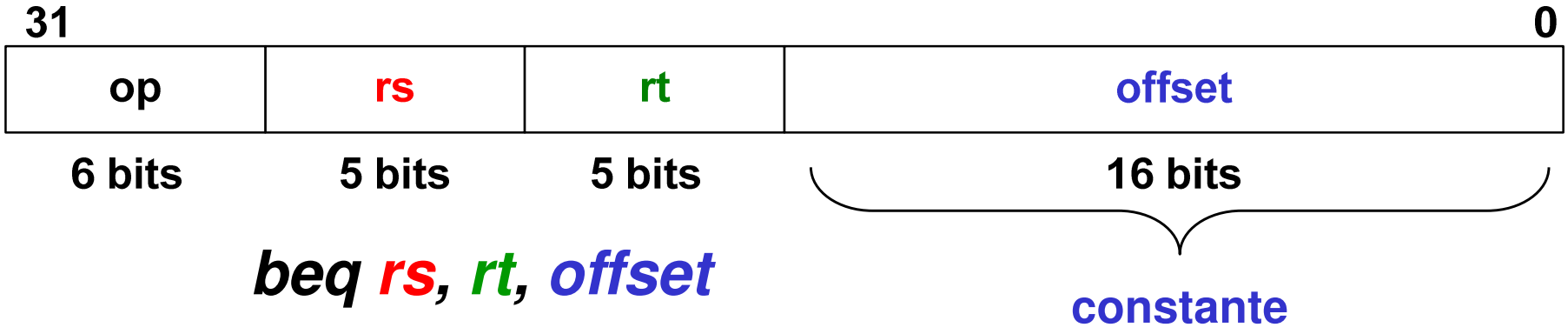
```
        .data
array:  .space  40              # static int array[SIZE];
        .eqv    SIZE, 10
        .text
        .globl  main
main:   la      $t0, array      # $t0 = &(array[0])
        li      $a0, SIZE       # $a0 = SIZE
        sll     $t1, $a0, 2     # $t1 = size * 4
        addu    $t1, $t1, $t0   # $t1 = &(array[size]);
for:    bgeu    $t0, $t1, endf  # while (p < &array[size]) {
        sw      $0,  0($t0)     #     *p = 0;
        addiu   $t0, $t0, 4     #      p = p + 1;
        j       for             # }
endf:   ...
```

Codificação de *branches* no MIPS

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

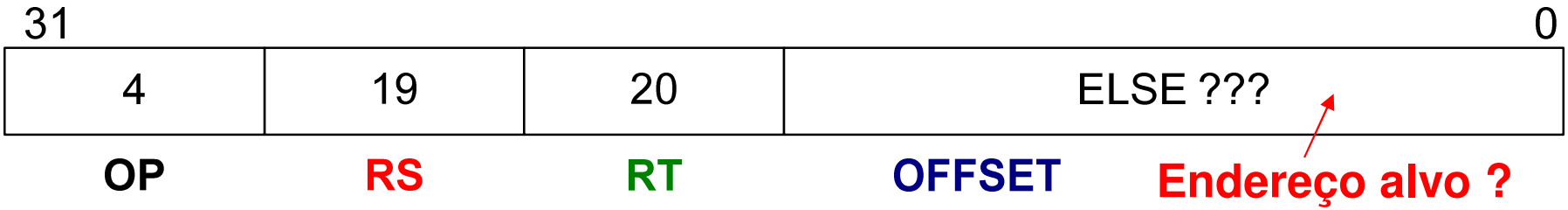


- A necessidade de codificação do **endereço-alvo** das instruções de salto condicional obriga a que estas instruções sejam codificadas recorrendo ao **formato I**



Codificação de *branches* no MIPS

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*
- 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
- 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**
- Endereçamento relativo ao PC (**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 o cálculo do *offset* é o endereço da instrução seguinte**
- As instruções estão armazenadas em memória em endereços múltiplos de 4 (e.g., **0x00400004**, **0x00400008**,…) 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

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

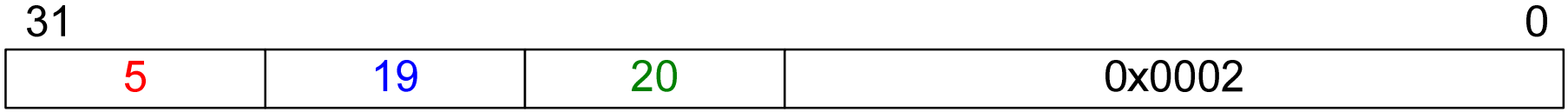
O "offset" seria portanto:

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!!!)



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.

Execução de uma instrução de *branch* no MIPS

- O campo *offset* do código máquina da instrução de *branch* é então usado para codificar a **diferença** entre o valor do endereço-alvo e o valor do endereço seguinte ao da instrução de *branch,* **dividida por 4**
- Durante a execução da instrução, o processador calcula o endereço-alvo como:

Endereço_alvo = PC_atual + (offset * 4)

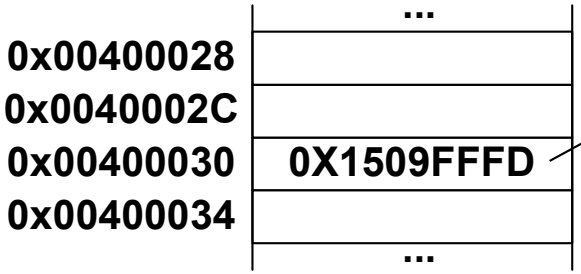
ou:

Endereço_alvo = PC_atual + (offset << 2)

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

Interpretação de uma instrução de *branch* no MIPS

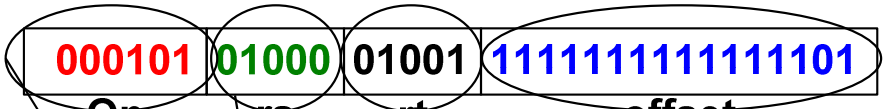
Exemplo



0001 0101 0000 1001 1111 1111 1111 1101

000101 01 0000 1001 1111 1111 1111 1101

bne → Instrução do tipo I



bne $8, $9, ???

Offset = 1111 1111 1111 1101

↓

1111 1111 1111 1111 1111 1111 1111 1101

↓

1111 1111 1111 1111 1111 1111 1111 0100

(0xFFFFFFF4)

O valor do PC foi incrementado na fase *fetch* da instrução

Endereço alvo = PC + (offset << 2) = 0x00400034 + 0xFFFFFFF4 = 0x00400028

Instrução descodificada: bne $8, $9, 0x00400028

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**

- Formato J:

- Endereço alvo da instrução "j" é sempre múltiplo de 4 (2 bits menos significativos são sempre 0)

Codificação da instrução de salto incondicional

- **Exemplo:** **j** **Label** **# com Label = 0x001D14C8**

0x001D14C8: 0000 **0000 0001 1101 0001 0100 1100 1000**

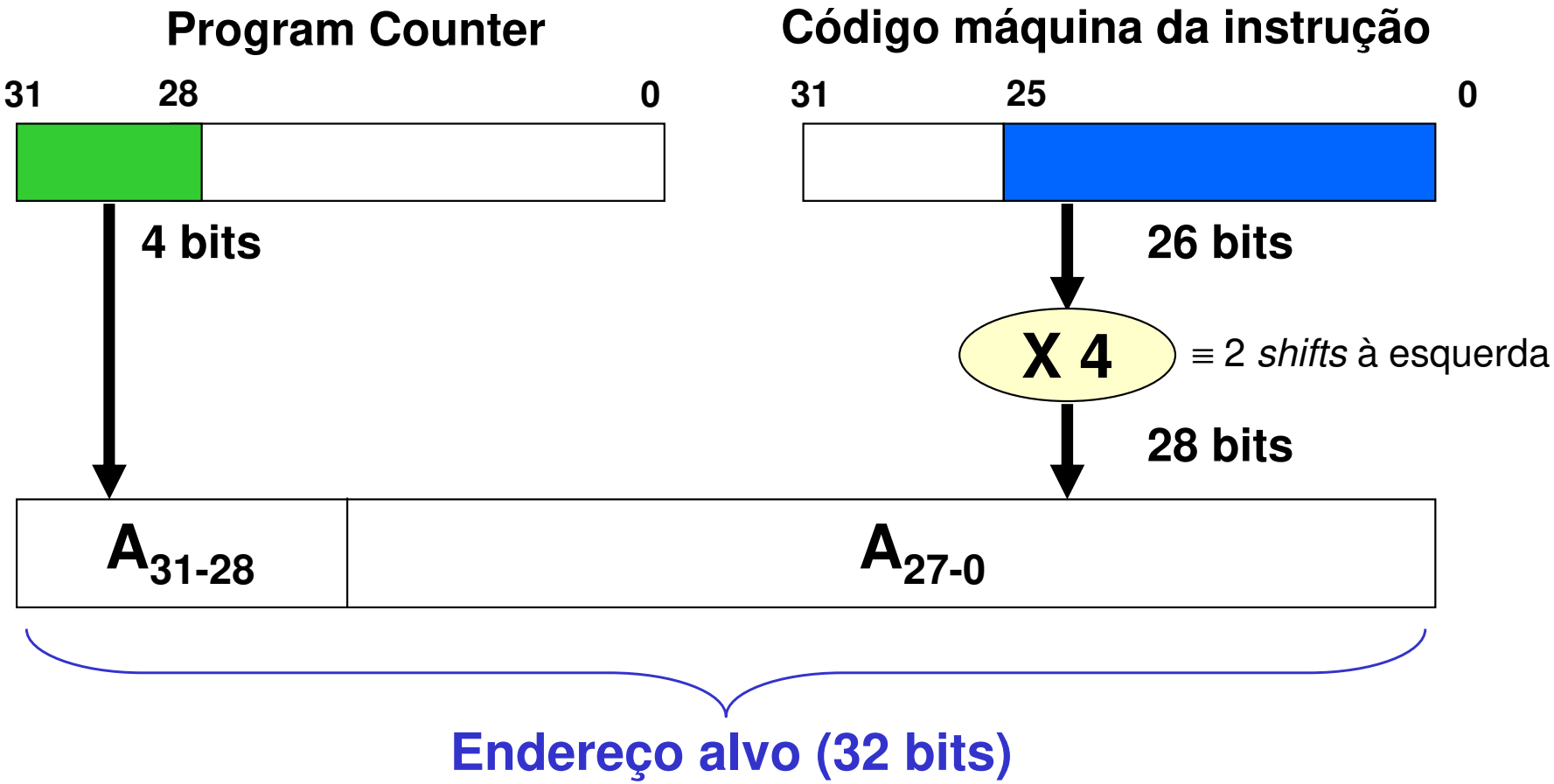
(26 bits) **00 0000 0111 0100 0101 0011 0010**

- Código máquina (opcode do "j" é 0x02):

0000 1000 0000 0111 0100 0101 0011 0010 = 0x08074532

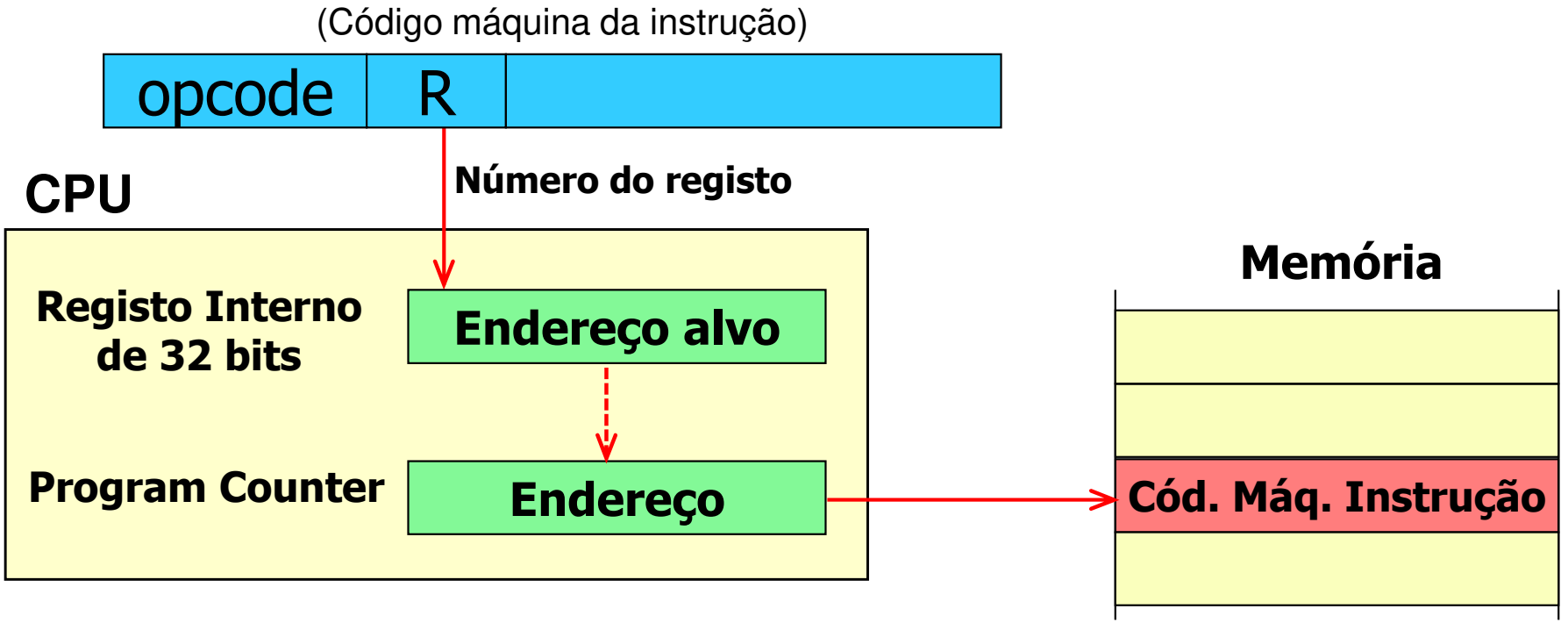
Cálculo do endereço-alvo de uma instrução J

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 que realize um 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** - Jump register)



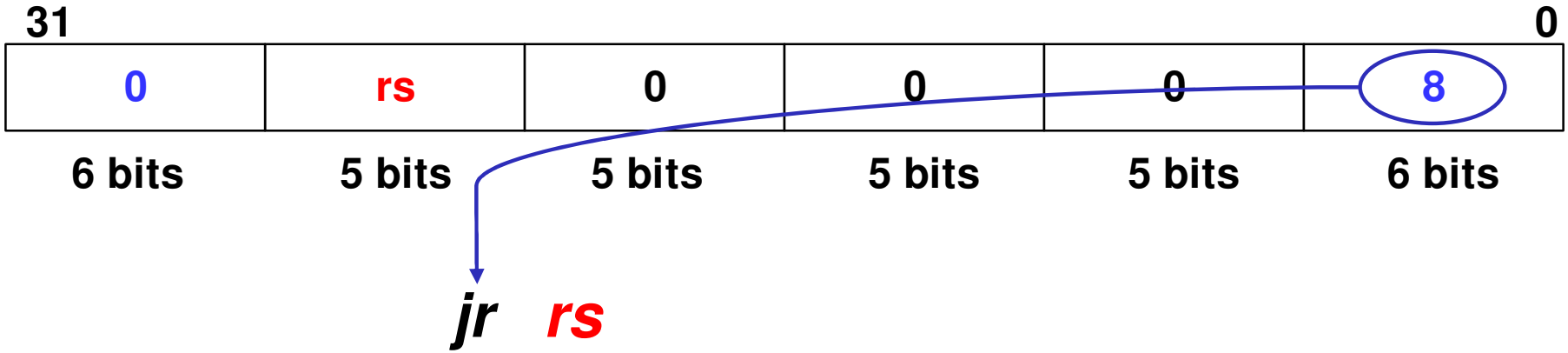
Instrução JR (jump on register)

```
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:

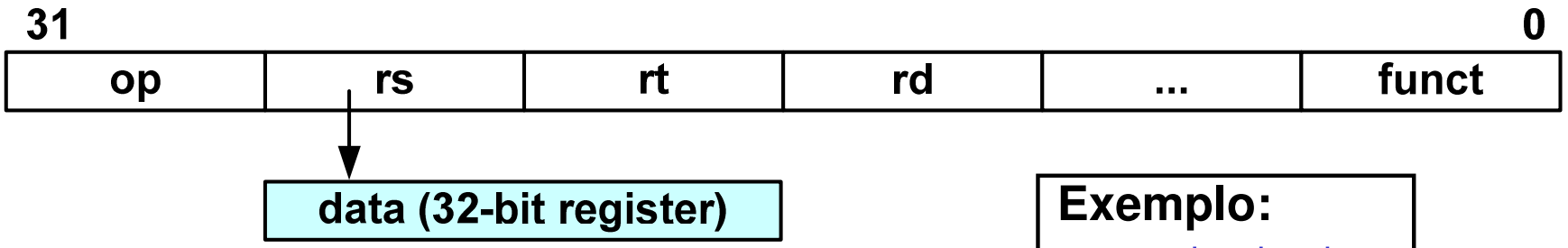


Modos de endereçamento no MIPS (resumo)

- 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**")

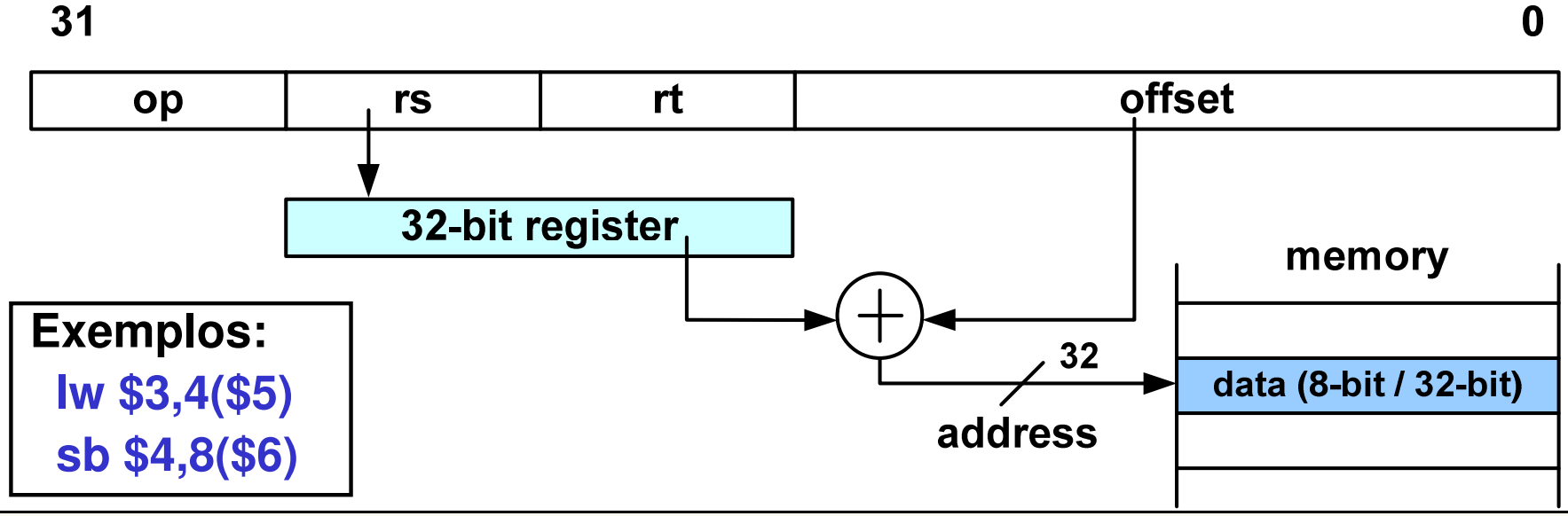
Modos de endereçamento do MIPS (resumo)

- **Register Addressing (endereçamento tipo registo):**



Exemplo:
add $3,$4,$5

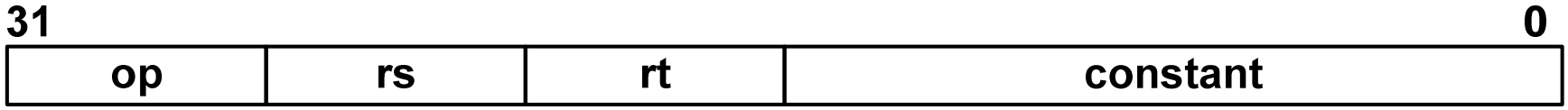
- **Base addressing (indireto por registo com deslocamento):**



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

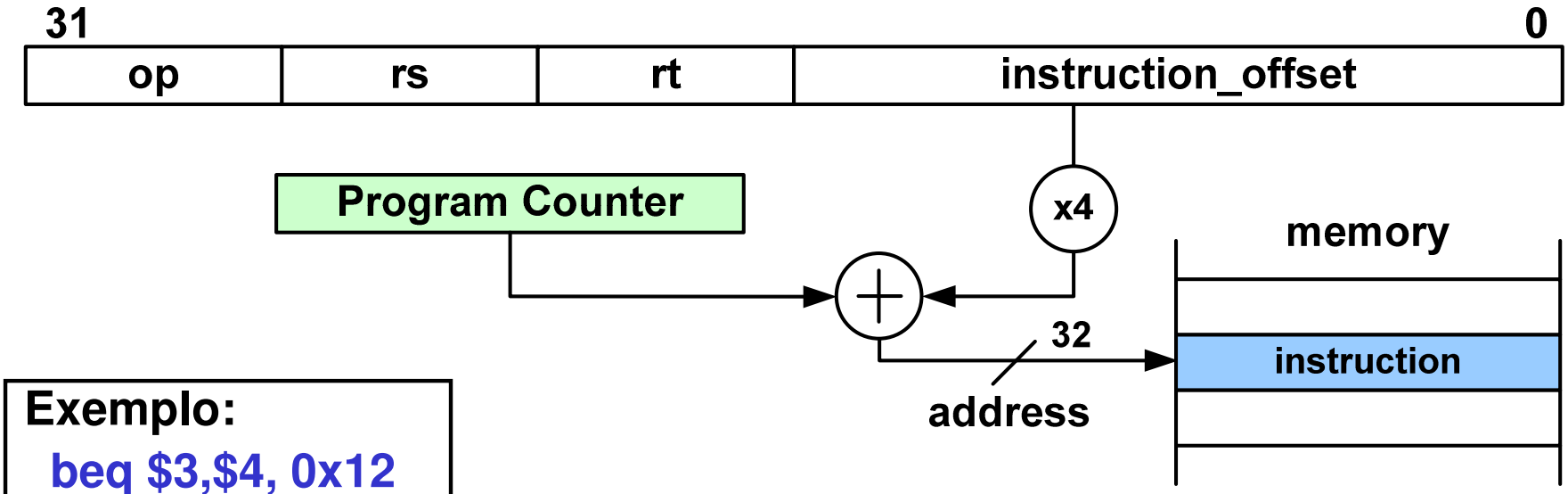
Modos de endereçamento do MIPS (resumo)

- **Immediate Addressing (endereçamento imediato):**



Exemplo:
addi $3,$4,0x3F

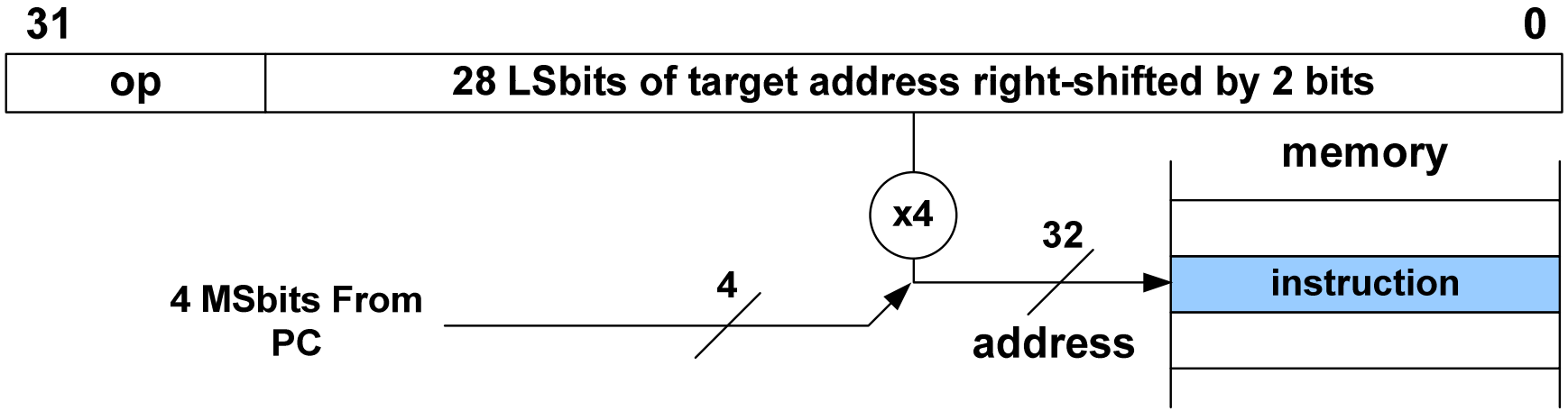
- **PC-relative Addressing (endereçamento relativo ao PC):**



Exemplo:
beq $3,$4, 0x12

Modos de endereçamento do MIPS (resumo)

- **Pseudo-direct Addressing (endereçamento 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
```

Questões

- O que significa a declaração "**`int *ac;`**"? Qual a diferença entre essa declaração e "**`int ac`**"?
  O que significa a declaração "**`char *ac;`**"?
- A partir das declarações de "**`a`**" e "**`b`**":

```
int  a;
int *b;
```

  identifique quais das seguintes atribuições são válidas:

```
a=b;     b=*a;       b=&(a+1);  a=&b;       b=&a;
b=*a+1;  b=*(a+1);   a=*b;      a=*(b+1);   a=*b+1;
```

- Identifique as operações, e respetiva sequência, realizadas nas seguintes instruções C:

```
a=*b++;  a=*(b)++;  a=*(++b);
```

- Suponha que "**`p`**" está declarado como "**`int *p;`**". Supondo que a organização da memória é do tipo "*byte-addressable*", qual o incremento no endereço que é obtido pela operação "**`p=p+2;`**" ?

Questões

- Suponha que "**b**" é um *array* declarado como "**int b[25]**;". Como é obtido o endereço inicial do *array*, i.e., o endereço da sua primeira posição? Supondo uma memória "*byte-addressable*", como é obtido o endereço do elemento "**b[6]**"?
- Dada a seguinte sequência de declarações:

```
int b[25];
int a;
int *p = b;
```

Identifique qual ou quais das seguintes atribuições permitem aceder ao elemento de índice 5 do *array* "**b**":

```
a = b[5];       a = *p + 5;
a = *(p + 5);   a = *(p + 20);
```

Exercício

- Pretende-se escrever uma função para a troca do conteúdo de duas variáveis (**troca(a, b);**). Isto é, se, antes da chamada à função, a=2 e b=5, então, após a chamada à função, os valores de a e b devem ser: a=5 e b= 2

  Uma solução incorreta para o problema é a seguinte:

```
void troca(int x, int y)
{
    int aux;

    aux = x;
    x = y;
    y = aux;
}
```

- Identifique o erro presente no trecho de código e faça as necessárias correções para que a função tenha o comportamento pretendido