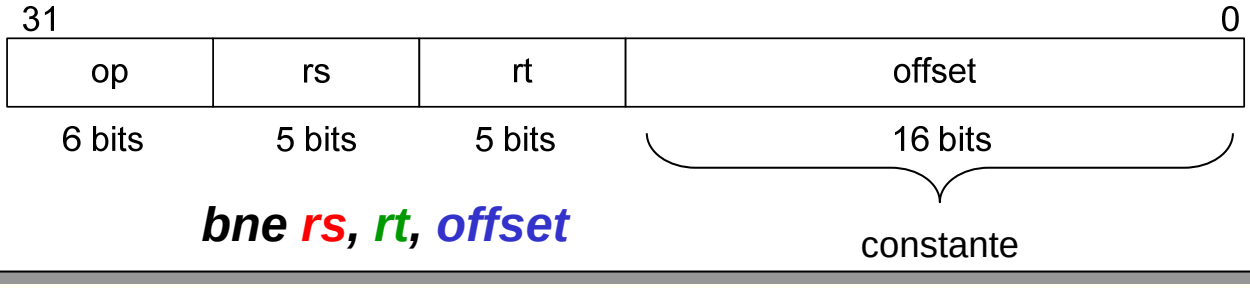
Aula 5

- Métodos de endereçamento em “saltos” condicionais e incondicionais
- Codificação das instruções de salto condicional e incondicional no MIPS
- Formatos de instrução **I** e **J**
- Modos de endereçamento do MIPS:
  - Modo imediato e uso de constantes

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira, Tomás Oliveira e Silva

- **Codificação das instruções de salto condicional no MIPS:**
  - As instruções aritméticas e lógicas no MIPS são codificadas no **formato R**

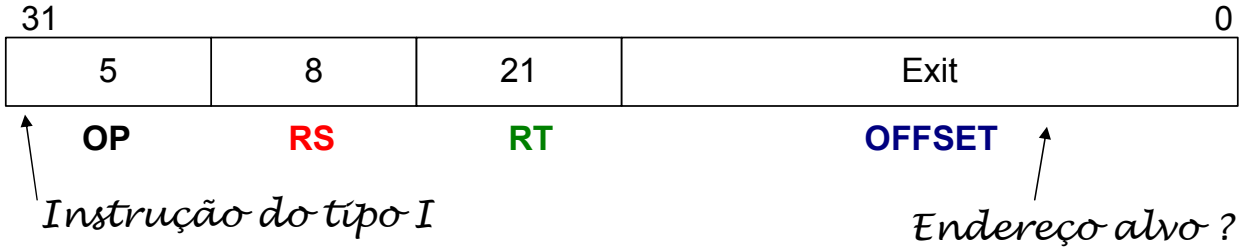
  - A necessidade de codificação do **endereço-alvo** da instrução de salto obriga à definição de um novo formato de codificação, o **formato I**



bne rs, rt, offset

Codificação de *branches* em código máquina.

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



bne rs, rt, offset

Mas se o **endereço alvo** é codificado com 16 bits, isso significa que o programa não pode ter uma dimensão superior a 2^{16} (64K). **Será essa uma opção realista?**

- Uma alternativa seria 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})
- **Mas qual o registo a usar?** Comecemos por observar que:
  - A maioria das instruções de salto condicional fazem-se para a vizinhança da própria instrução (exemplo: estatisticamente verifica-se 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, ...)

Solução: Utilizar o registo PC (Program Counter)

- Usando **endereçamento relativo** – o valor do endereço alvo é calculado somando algebricamente o *offset* de 16 bits codificado na instrução ao valor corrente do PC
- No MIPS o valor do PC corresponde ao endereço da instrução seguinte, uma vez que esse registo é incrementado antes da fase *execute* da instrução
- Assim, o endereço-alvo (novo PC) passaria a ser:

  Novo **PC = PC** actual **+ offset**

Note-se que 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

Tomemos 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

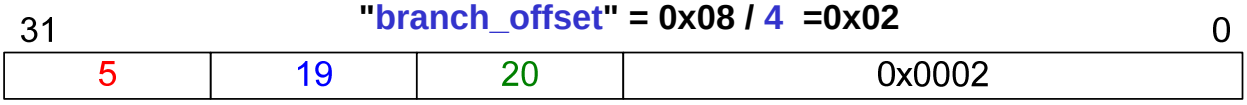
O "branch_offset" seria portanto:

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 "branch_offset" é, na realidade, calculado em instruções. Logo:

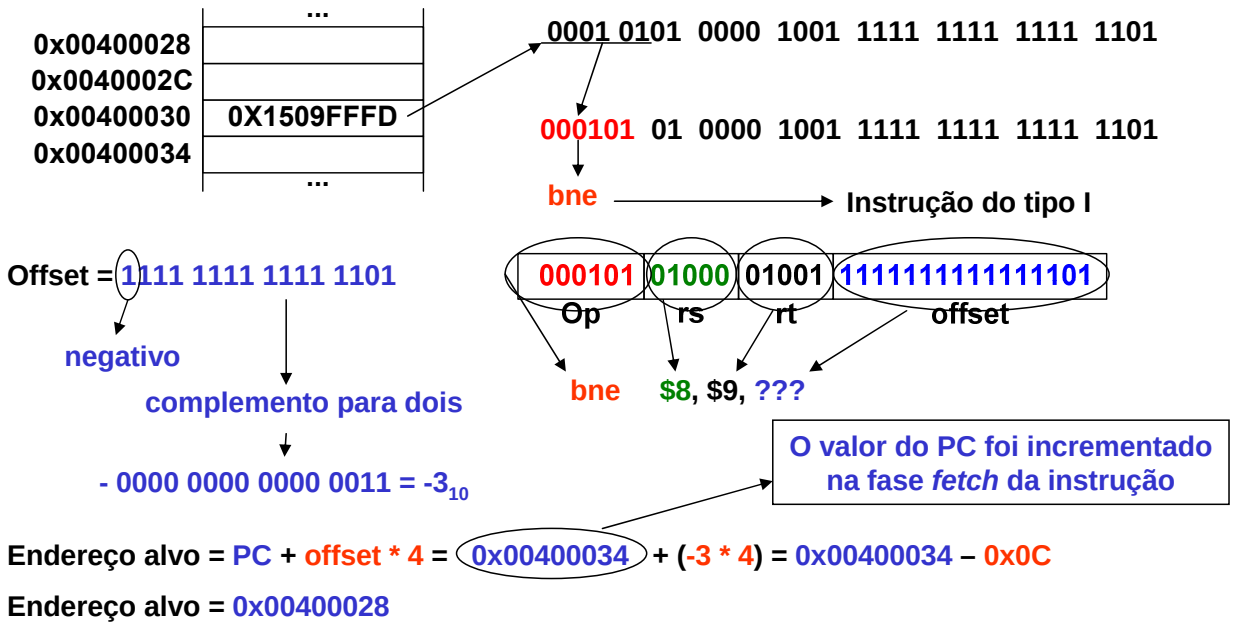
"branch_offset" = 0x08 / 4 =0x02



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

Código máquina: **00010110011101000000000000000010 = 0x16740002**

Exemplo! Como interpreta o CPU uma instrução?



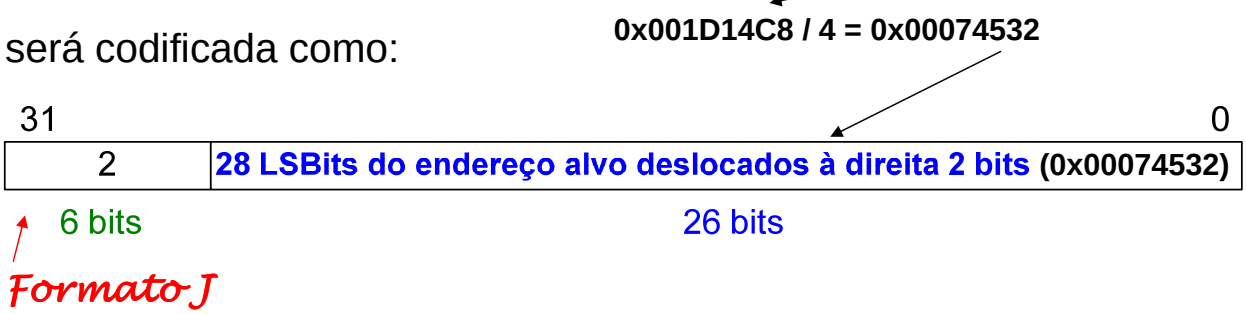
Instrução descodificada: **bne $8, $9, 0x00400028**

E no caso do **salto incondicional** (instrução j)?

Nesse caso estamos perante uma situação de **endereçamento directo**, i.e. o **código máquina** da instrução **codifica directamente o endereço alvo** (endereço do qual será lida a próxima instrução).

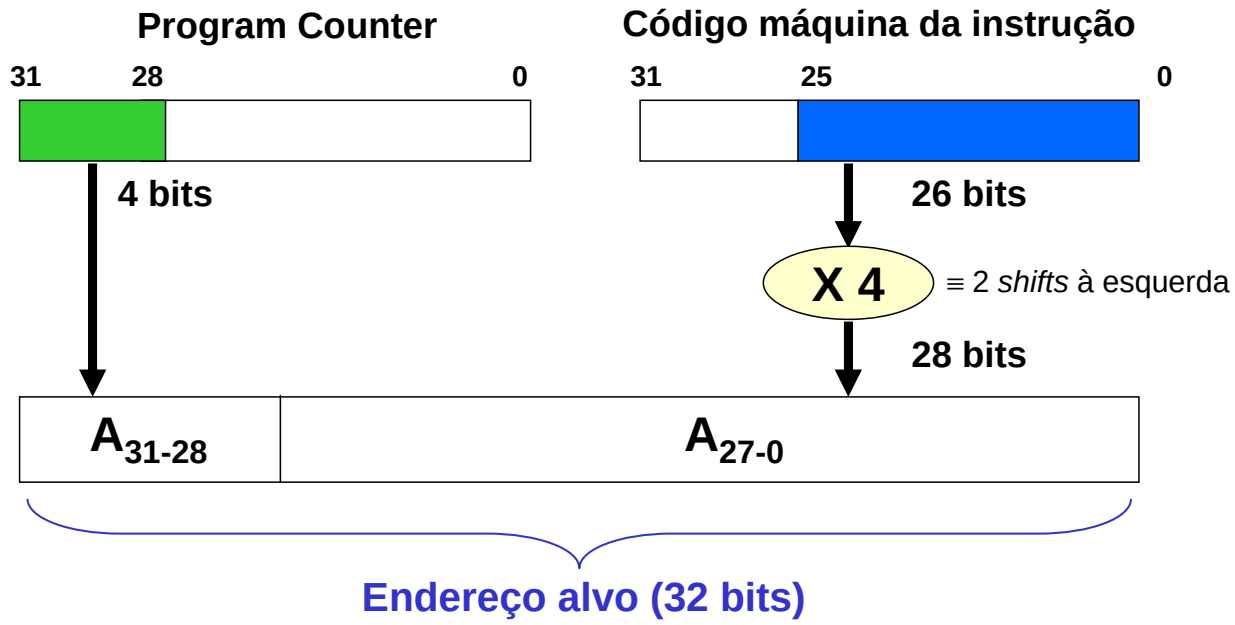
Exemplo: A instrução `j Label #se Label=0x001D14C8`

será codificada como:

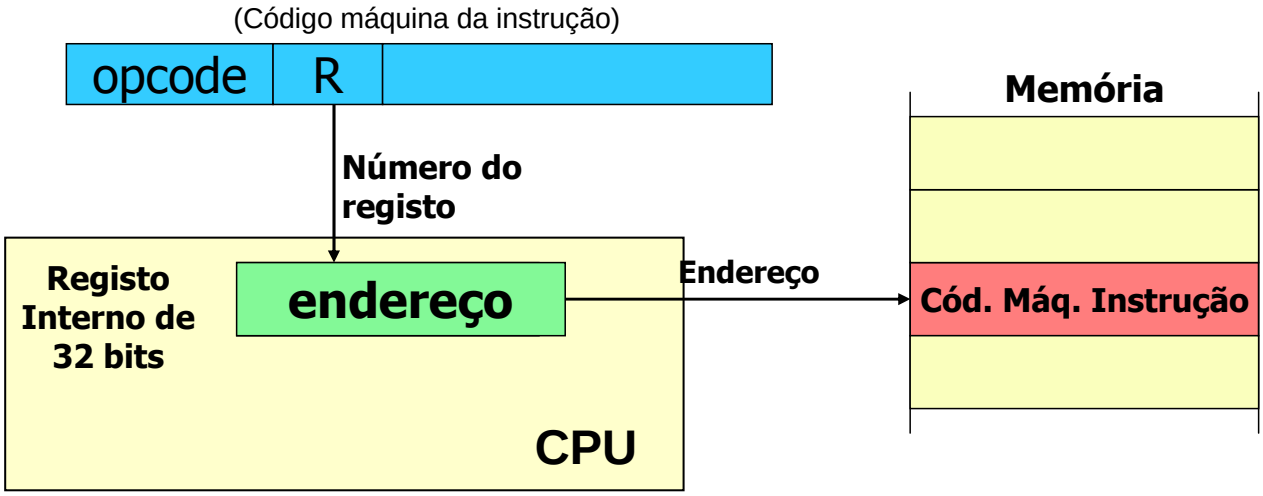


Código Máquina: $000010000000011101000101001100 10_2$ = **0x08074532**

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



- **Haverá maneira de especificar um endereço para um *salto*, usando para isso os 32 bits?**
- Há! Utiliza-se **endereçamento indirecto por registo**. Ou seja, um registo interno (de 32 bits) armazena o endereço alvo da instrução de salto



- **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 desta instrução é o formato R:

JR ***RS***

Modos de endereçamento no MIPS:

- O **método** usado pela arquitectura para **identificar o elemento que contém a informação** que irá ser processada por uma dada instrução é genericamente designada por "**Modo de Endereçamento**"
- Já tivemos oportunidade de observar, nos slides anteriores, mais do que um modo de endereçamento usado pelo MIPS:
  - Nas instruções aritméticas, os endereços dos registos internos envolvidos na operação são especificados directamente na própria instrução, em campos de 5 bits (***rs***, ***rt*** e ***rd***) – **endereçamento tipo registo**
  - 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 indirecto por registo**

Para além dos modos de endereçamento que já conhecemos, o MIPS suporta ainda um outro tipo de endereçamento, designado por "**endereçamento imediato**".

Relembremos os quatro princípios básicos no design de uma arquitectura

1. A simplicidade favorece a regularidade
2. Quanto mais pequeno mais rápido
3. Um bom design implica compromissos adequados
4. **O que é mais comum deve ser mais rápido**

O ponto 4. determina que a capacidade de tornar mais rápida a execução das operações que ocorrem mais vezes, resulta num aumento global do desempenho!

- Pode-se verificar, 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.
- A constante "zero", por exemplo, é tão usada, que o MIPS tem um registo permanentemente com esse valor ($0).
- A constante "um", por outro lado, também é muito utilizada em operações de incremento ou decremento de variáveis de contagem usadas dentro de estruturas em ciclo fechado.

Chamamos 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

- **Se a constante fosse armazenada na memória externa**, a sua utilização implicaria sempre o recurso a duas instruções: uma para ler o valor da constante para um registo interno, e outra para operar com base nessa constante.

Ex.:
```
#Constante na mem. externa (endereço em $6)
lw    $5, 0($6)     #Ler constante p/ o registo $5
add   $8, $7, $5    #Somar $7 com a constante
```

- Para aumentar a eficiência, as arquitecturas 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 do tipo **imediato** são identificadas **pelo sufixo "i"**, como nas seguintes:

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

- Mas, 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 recorrendo ao **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...

- 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
slti  $12, $10, 0xFFFF
```

ADD Immediate (signed)

A gama de representação da constante será: [**-32768, +32767**]

- Existem também instruções em que a constante deve ser entendida como uma quantidade inteira sem sinal. Por exemplo todas as instruções lógicas:

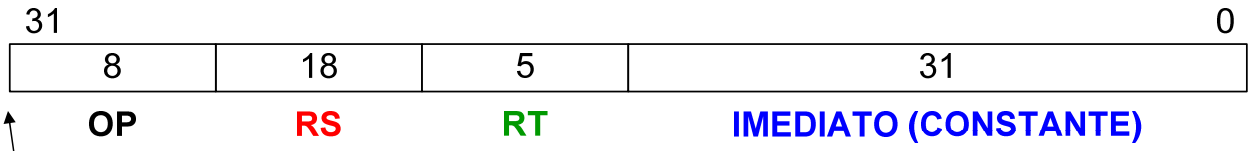
```
andi  $3,  $5, 0xFFFF
```

AND Immediate

A gama de representação da constante será: [**0, 65535**]

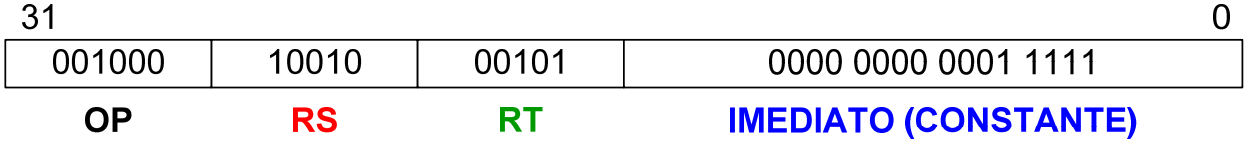
O formato de codificação das instruções que usam constantes é então:

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



Instrução do tipo I

addi rt, rs, immediate



Cod. Máquina: 00100010010000101000000000011111 = 0x2245001F

- Pode ser necessário referenciar 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?

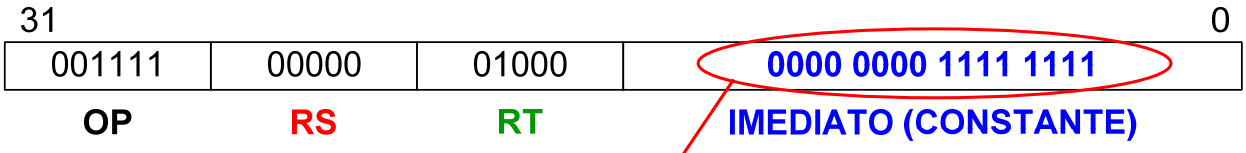
- Para facilitar a manipulação de **imediatos** com **mais de 16 bits**, o *set* de instruções do MIPS inclui a seguinte instrução:

```
lui   $reg, immediate
```

lui – "Load upper immediate"

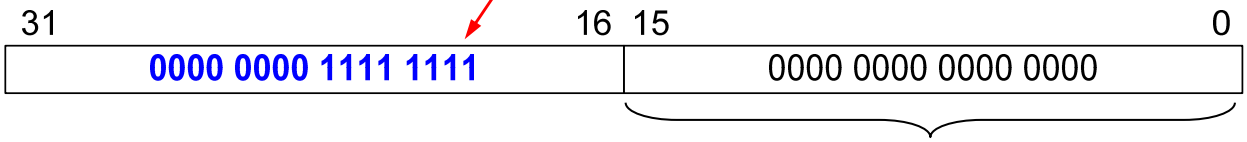
coloca a constante "immediate"nos 16 bits mais significativos do registo destino (também é uma instrução do tipo I)

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

Desta forma, a **instrução virtual** "**load address**"

```
la   $16, MyData    # Ex. MyData = 0x10010034
```

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

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

Notas:

- O **registo $1** 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"**

	0x 1001 0000	$1
+	**0x 0000 0034**	Imediato
	0x 1001 0034	$16