Arquitectura de Computadores I 2007/08

1º Semestre de 2007/2008

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Arquitectura de Computadores I 2007/08

Aula 7

Modos de endereçamento do MIPS:

- Modo imediato e uso de constantes
- Resumo dos quatro modos base

Estruturas de controlo de fluxo de execução:

- **if()...then...else**
- Ciclos **for()** e **while()**
- Ciclo **do...while()**

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:
  - No caso das instrução de "*load*" e "*store*", por exemplo, o endereço do registo da memória externa envolvido na transferência é obtido da soma de uma constante com o conteúdo de um registo interno.
  - Já 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***).

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

Relembremos os quatro princípios básicos no design de uma arquitectura

1. A simplicidade favorece a 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 da performance!

- Acontece que se pode verificar, estatisticamente, que um número muito significativo de instruções que operam aritmeticamente usam uma **constante** como um dos seus parâmetros. Na realidade, 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 operação 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.

Exp:
```
#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, os arquitectos do MIPS conceberam 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.

Arquitectura de Computadores I 2007/08

As instruções do tipo **imediato** são identificadas, em *Assembly* do MIPS, **pelo sufixo "i"**, como no caso das instruções:

```
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"?**

A resposta é: **16 bits**.
Este espaço é geralmente suficiente para armazenar as constantes mais frequentemente utilizadas (geralmente valores pequenos).

Arquitectura de Computadores I 2007/08

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
ori    $12, $10, 0xFF
```

Neste caso a gama de representação será: [-32768, +32767]

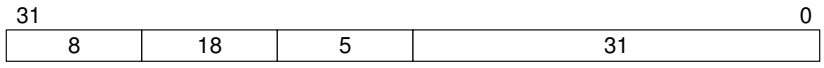
Existem no entanto instruções que explicitamente determinam que a constante deve ser entendida como uma quantidade inteira sem sinal. Por exemplo:

```
addiu  $3, $5, 0x4F
```

Situação em que a gama de representação será: [0, 65535]

E qual é, então, o formato das instruções que usam constantes?

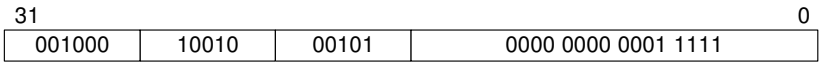
Exemplo: **addi $5, $18, 31**



OP **RS** **RT** **IMEDIATO (CONSTANTE)**

Instrução do tipo I

addi rt, rs, immediate



OP **RS** **RT** **IMEDIATO (CONSTANTE)**

0010 0010 0100 0101 0000 0000 0001 1111 = 0x2245001F

Pode, no entanto, acontecer que seja necessário referenciar constantes que necessitem de um espaço de armazenamento com mais do que 16 bits (como seja a referência explícita a um endereço). Como lidar com esses casos?

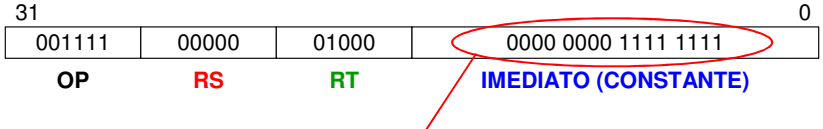
Para facilitar a manipulação de **imediatos** com mais de 16 bits, o *set* de instruções do MIPS inclui a seguinte instrução:

```
lui   $reg, immediate
```

lui – "Load upper immediate"

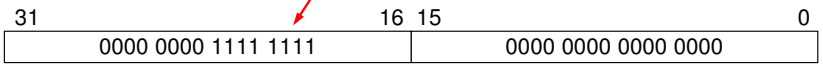
coloca a constante nos 16 bits mais significativos do registo destino (também é uma instrução do tipo I)

Exemplo: `lui $8, 255`



lui rt, immediate

Conteúdo do registo $8 após a execução da instrução:



$8 = 0x00FF0000

Desta forma, a instrução (virtual) "load address"

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

será executada no MIPS pela sequência de instruções:

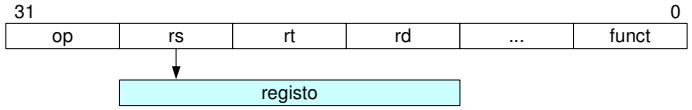
```
lui $1,0x1001      # $1 = 0x10010000
ori $16,$1,0x0034 #$16 = 0x10010000 | 0x00000034
```

Notas:
- O **registo $1** é reservado para o *Assembler*, para permitir este tipo de decomposição de instruções!
- **A instrução li (*load immediate*)** é executada no MIPS pela mesma sequência de instruções

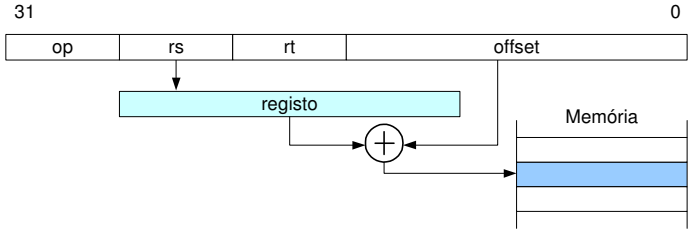
0x 1001 0000		$1
+	**0x 0000 0034**	Imediato
0x 1001 0034		$16

Resumindo. Os modos de endereçamento suportados pelo MIPS são:

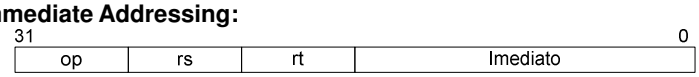
- **Register Addressing:**



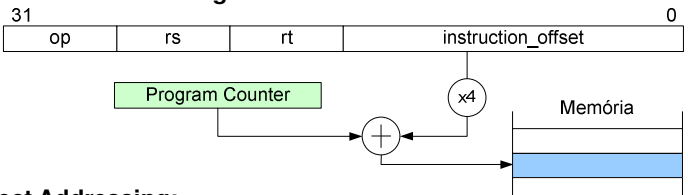
- **Indirecto por registo com deslocamento (base addressing):**



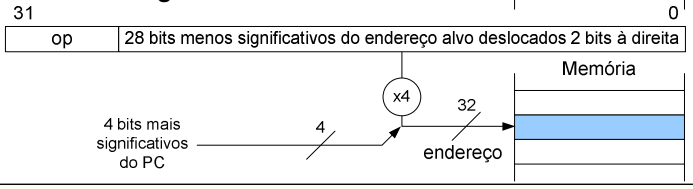
- **Immediate Addressing:**



- **PC-relative Addressing:**



- **Direct Addressing:**



Estruturas de controlo de fluxo em *Assembly* do MIPS

Consideremos os seguintes trechos de código:

```
if (a >= n) {
    b = c;
} else {
    b = d;
}...
```

```
for (n = 0; n < 100; n++)
{
    a = a + b[n];
}
...
```

```
n = 0;
do
{
    a = a + b[n];
    n++;
}while (n < 100);
...
```

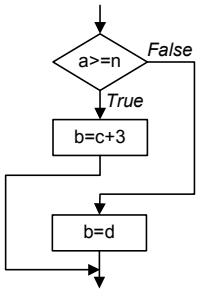
```
n = 0;
while (n < 100)
{
    a = a + b[n];
    n++;
}
...
```


1º Exemplo:

```
if (a >= n) {
    b = c + 3;
} else {
    b = d;
}
```

$s1 ↔ a
$s2 ↔ n
$s3 ↔ c
$t0 ↔ b
$s4 ↔ d

Transformando o código apresentado no fluxograma equivalente:



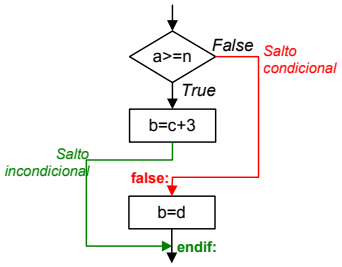
É possível identificar facilmente a ocorrência de 1 salto condicional e de um salto incondicional.

1º Exemplo:

```
if (a >= n) {
    b = c + 3;
} else {
    b = d;
}
```

$s1 ↔ a
$s2 ↔ n
$s3 ↔ c
$t0 ↔ b
$s4 ↔ d

Transformando o código apresentado no fluxograma equivalente:



É possível identificar facilmente a ocorrência de 1 salto condicional e de um salto incondicional.

E adaptar o salto condicional para que este salte quando a condição for verdadeira (tal como nos *branches*).

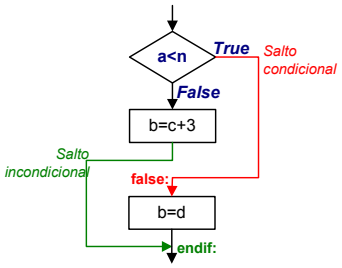
Arquitectura de Computadores I 2007/08

1º Exemplo:

```
if (a >= n) {
    b = c + 3;
} else {
    b = d;
}
```

$s1 ↔ a
$s2 ↔ n
$s3 ↔ c
$t0 ↔ b
$s4 ↔ d

Transformando o código apresentado no fluxograma equivalente:



É possível identificar facilmente a ocorrência de 1 salto condicional e de um salto incondicional.

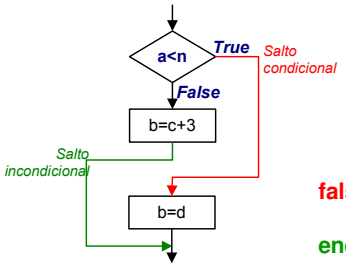
E adaptar o salto condicional para que este salte quando a condição for verdadeira (tal como nos *branches*).

1º Exemplo:

```
if (a >= n) {
    b = c + 3;
} else {
    b = d;
}
```

$s1 ↔ a
$s2 ↔ n
$s3 ↔ c
$t0 ↔ b
$s4 ↔ d

Transformando o código apresentado no fluxograma equivalente:



A tradução torna-se directa:

```
        blt     $s1, $s2, false  # if (a >= n) {
        addi    $t0, $s3, 3      #   b = c + 3;
        j       endif            # }
false:                           # else {
        move    $t0, $s4         #   b = d;
endif:  ...                      # }
```


2º Exemplo:

```
for (n = 0; n < 100; n++)
{
    a = a + b[n];
}
...
```

```
n = 0;
while (n < 100)
{
    a = a + b[n];
    n++;
}
...
```

Estes dois exemplos são funcionalmente equivalentes!

Operações a executar antes e fora do corpo do ciclo (inicializações)

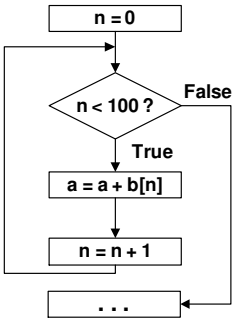
Condição de continuação da execução do ciclo

Operações a realizar no fim (dentro) do corpo do ciclo

2º Exemplo:

```
for (n = 0; n < 100; n++)
{
    a = a + b[n];
}
...
```

```
n = 0;
while (n < 100)
{
    a = a + b[n];
    n++;
}
...
```



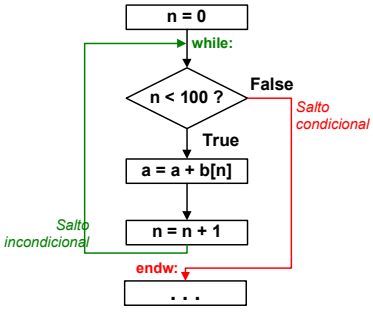
É possível identificar a ocorrência de 1 salto condicional e de um salto incondicional.

Arquitectura de Computadores I 2007/08

2º Exemplo:

```
for (n = 0; n < 100; n++)
{
    a = a + b[n];
}
...
```

```
n = 0;
while (n < 100)
{
    a = a + b[n];
    n++;
}
...
```



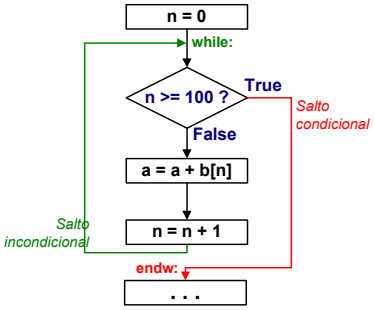
É possível identificar a ocorrência de 1 salto condicional e de um salto incondicional.

O salto condicional necessita de ser modificado de forma a saltar quando a condição for verdadeira.

2º Exemplo:

```
for (n = 0; n < 100; n++)
{
    a = a + b[n];
}
...
```

```
n = 0;
while (n < 100)
{
    a = a + b[n];
    n++;
}
...
```



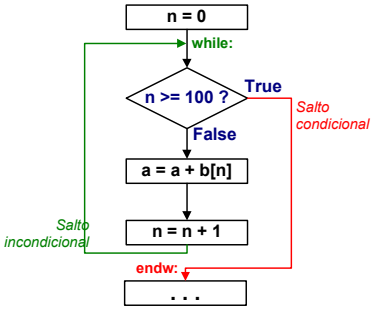
É possível identificar a ocorrência de 1 salto condicional e de um salto incondicional.

O salto condicional necessita de ser modificado de forma a saltar quando a condição for verdadeira.

2º Exemplo:

```
for (n = 0; n < 100; n++)
{
    a = a + b[n];
}
...
```

```
n = 0;
while (n < 100)
{
    a = a + b[n];
    n++;
}
...
```



Traduzindo a estrutura de controlo:

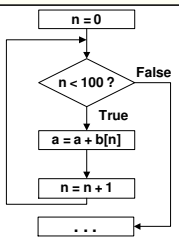
```
        li    n, 0
while:
        bge   n, 100, endw
        .
        .
        addi  n, n, 1
        j     while
endw:
```

2º Exemplo:

Note-se o uso do complemento lógico

```
for (n = 0; n < 100; n++)
{
    a = a + b[n];
}
...
```

```
n = 0;
while (n < 100)
{
    a = a + b[n];
    n++;
}
...
```



O código equivalente em *Assembly* MIPS:
($s1 ↔ n, $s2 ↔ b, $s3 ↔ a)

```
        li    $s1, 0          # n = 0;
while:  bge   $s1, 100, endw  # while(n < 100) {
        mul   $t0, $s1, 4     #     temp = 4 * n;
        add   $t0, $s2, $t0   #     temp = temp + b;
        lw    $t0, 0($t0)     #     temp = *temp;
        add   $s3, $s3, $t0   #     a = a + temp;
        addi  $s1, $s1, 1     #     n++;
        j     while           # }
endw:   ...
```

Teste condicional efectuado "à cabeça"

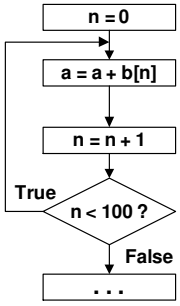
3º Exemplo:

Ao contrário do **for()** e do **while()**, o corpo do ciclo **do...while()** é executado incondicionalmente pelo menos uma vez!

```
n = 0;
do
{
    a = a + b[n];
    n++;
}while (n < 100);
...
```

do:

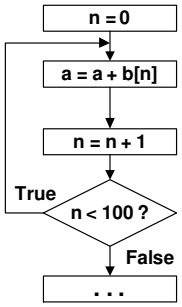
blt n, 100, do



3º Exemplo:

$s1 ↔ n
$s2 ↔ b
$s3 ↔ a

```
n = 0;
do
{
    a = a + b[n];
    n++;
}while (n < 100);
...
```



O código equivalente em *Assembly* do MIPS:

```
        li      $s1,0           # n = 0;
do:                             # do {
        mul     $t0, $s1, 4     #     temp = 4 * n;
        add     $t0, $s2, $t0   #     temp = temp + b;
        lw      $t0, 0($t0)     #     temp = *temp;
        add     $s3, $s3, $t0   #     a = a + temp;
        addi    $s1, $s1, 1     #     n = n + 1;
        blt     $s1, 100, do    # } while(n < 100);
        ...
```

Desta feita, o teste condicional é efectuado "no fim do ciclo

Resumindo:

- As estruturas do tipo **ciclo** incluem, geralmente, uma ou mais instruções de inicialização de variáveis, executadas antes e fora do mesmo
- No caso do `for()` e do `while()` o teste condicional é executado no início do ciclo
- No caso do `do...while()` o teste condicional é efectuado no fim do ciclo
- Na tradução de um `for()` para *Assembly*, o terceiro campo é incluído no fim do corpo do ciclo.