

Aulas 3 e 4

- Instruções de controlo de fluxo de execução
- Estruturas de controlo de fluxo de execução:
 - if()...then...else
 - Ciclos “for()”, “while()” e “do...while()”
- Tradução das estruturas de controlo de fluxo de execução para *Assembly* do MIPS

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Instruções de controlo de fluxo de execução

- Durante a execução de um programa há necessidade de tomar decisões com base em valores que só são conhecidos durante a execução do mesmo
- As instruções que permitem a tomada de decisões pertencem à classe "controlo de fluxo de execução"
- No MIPS as instruções básicas de controlo de fluxo de execução são:

beq **Rsrc1**, **Rsrc2**, **Label** # branch **if equal**

bne **Rsrc1**, **Rsrc2**, **Label** # branch **if not equal**

e são conhecidas como "**branches**" (saltos / *jumps*)
condicionais

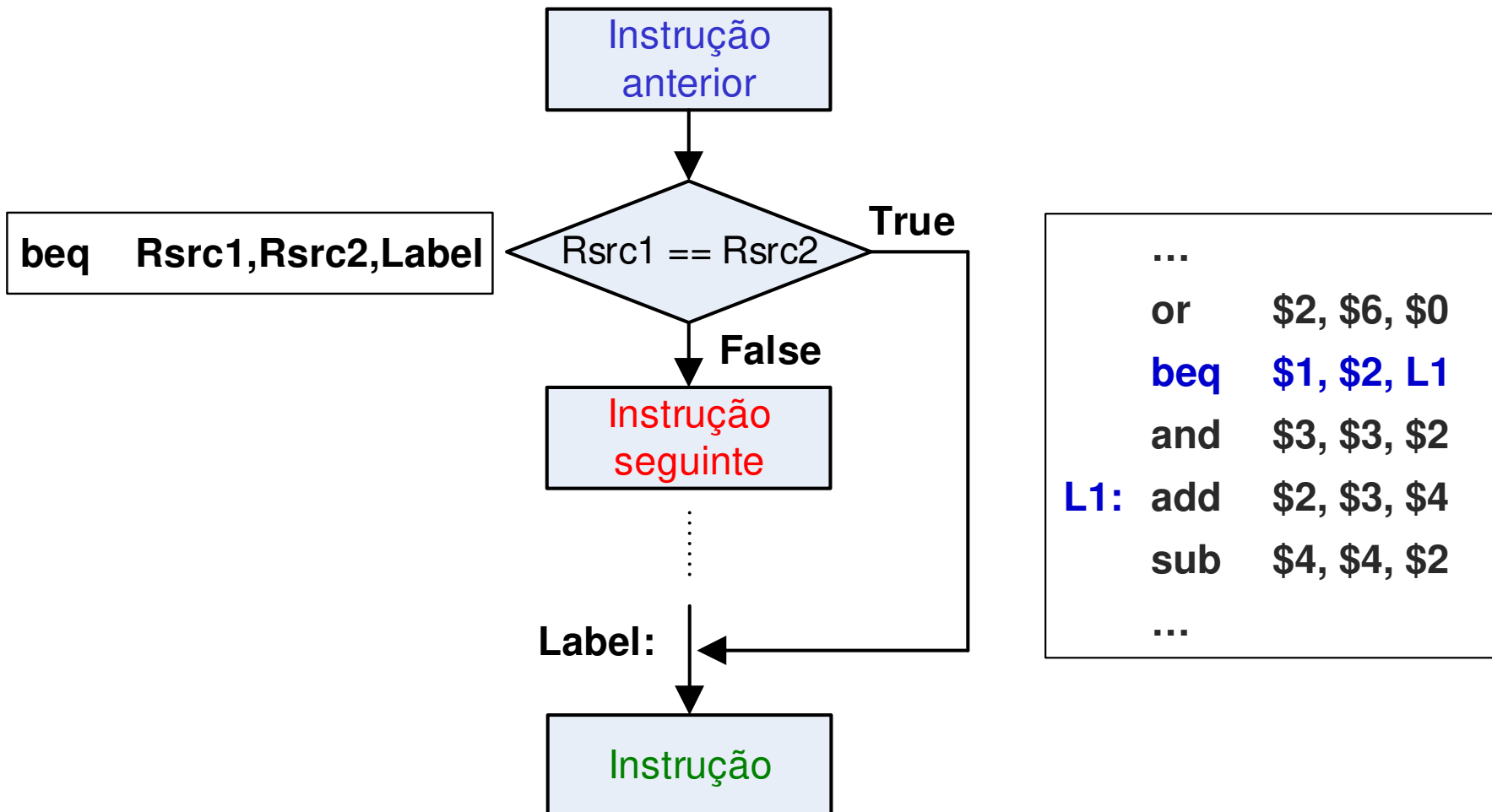
Instruções de controlo de fluxo de execução – BEQ

beq **Rsrc1**, **Rsrc2**, **Label** # branch if equal

- Se os conteúdos dos registos **Rsrc1** e **Rsrc2** forem iguais é realizado um salto, i.e., a execução continua na **instrução situada no endereço representado por "Label"** (*branch taken*)
- A execução continua na **instrução seguinte** se os conteúdos dos 2 registos forem diferentes (*branch not taken*)
- O endereço para onde o salto é efetuado (no caso de a condição ser verdadeira) designa-se por **endereço-alvo** da instrução de *branch* (*branch target address*)

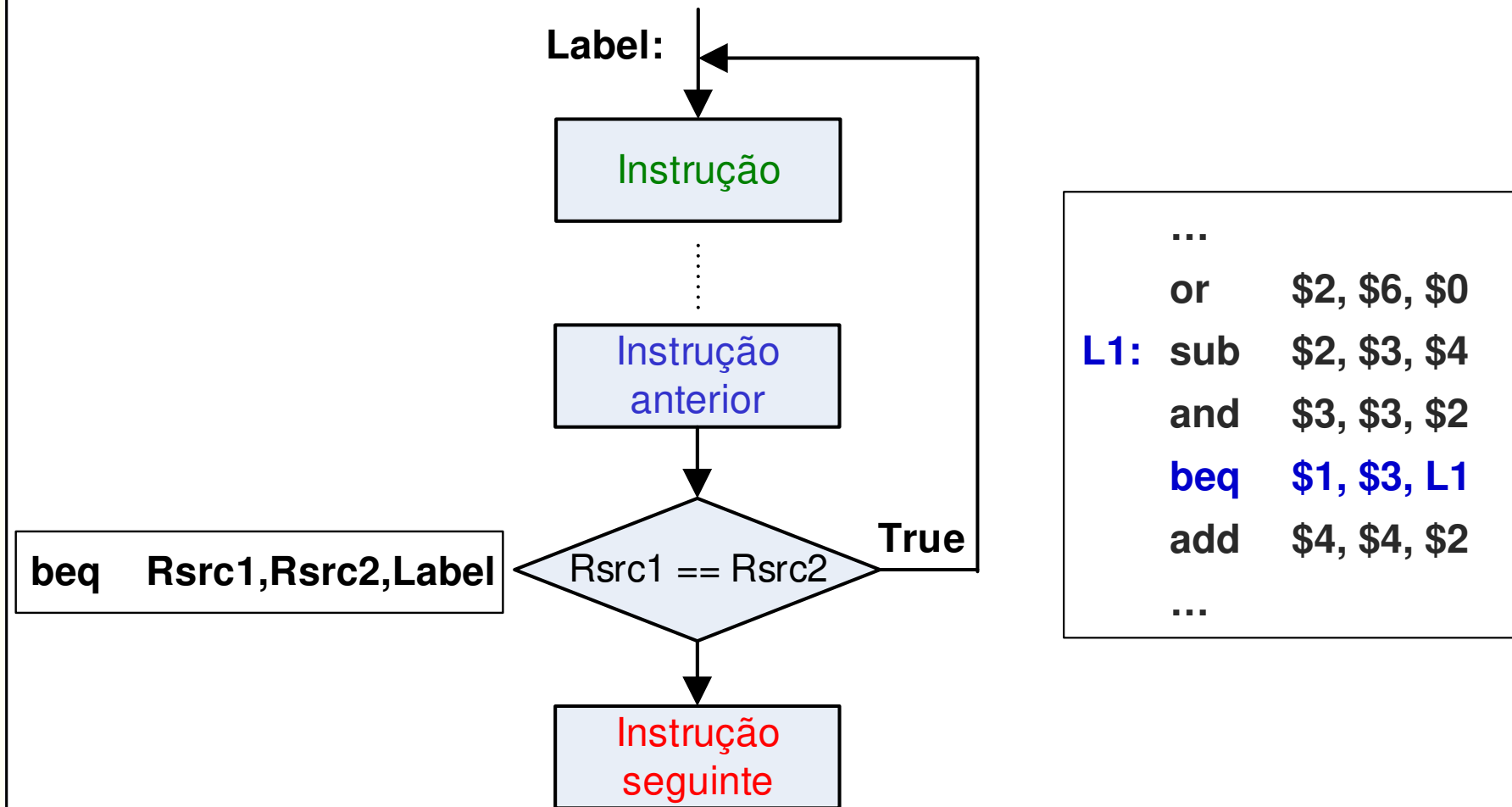
Instruções de *branch* – como funcionam?

- **beq** **Rsrc1**, **Rsrc2**, **Label** # branch if equal



Instruções de *branch* – como funcionam?

- **beq** **Rsrc1**, **Rsrc2**, **Label** # branch if equal



Instruções de *branch* – como funcionam?

- Se a **condição** testada na instrução **for verdadeira** (no caso do "beq" **Rsrc1=Rsrc2**, isto é **Rsrc1 – Rsrc2 = 0**), o valor corrente do PC (**Program Counter**) é substituído pelo endereço a que corresponde "Label" (endereço-alvo)
 - A instrução que é executada de seguida é a que se situa no endereço-alvo
- Se a **condição for falsa**, a sequência de execução não é alterada
 - A instrução que é executada de seguida é a que se situa imediatamente a seguir à instrução de *branch*

Instruções de controlo de fluxo de execução – BNE

bne **Rsrc1**, **Rsrc2**, **Label** # branch if not equal

- Se os conteúdos dos registos **Rsrc1** e **Rsrc2** forem diferentes é realizado um salto, i.e., a execução continua na **instrução situada no endereço representado por "Label"** (*branch taken*)
- A execução continua na instrução seguinte se os conteúdos dos 2 registos forem iguais (*branch not taken*)
- Exemplo:

```
or      $2, $6, $0
bne     $1, $2, L1  # se "branch taken" (i.e. $1 != $2)
                        # a próxima instrução a executar
                        # está em L1 (add $2,$3,$4)
and     $3, $3, $2  # se "branch not taken" a sequência
                        # de execução não é alterada
L1:    add     $2, $3, $4 #
        sub     $4, $4, $2 #
```

Outras instruções de *branch* do MIPS

- O ISA do MIPS suporta ainda um conjunto de instruções que **comparam diretamente com zero**:
 - **bltz** **Rsrc**, **Label** **# Branch if Rsrc < 0**
 - **blez** **Rsrc**, **Label** **# Branch if Rsrc ≤ 0**
 - **bgtz** **Rsrc**, **Label** **# Branch if Rsrc > 0**
 - **bgez** **Rsrc**, **Label** **# Branch if Rsrc ≥ 0**
- Nestas instruções **o registo \$0 está implícito** como o segundo registo a comparar
- Exemplos:
 - **blez \$1, L2 # if \$1 <= 0 then goto L2**
 - **bgtz \$2, L3 # if \$2 > 0 then goto L3**

Instrução SLT

Para além das instruções de salto com base no critério de igualdade e desigualdade, o MIPS suporta ainda a instrução:

```
slt Rdst, Rsrc1, Rsrc2    # slt  $\equiv$  "set if less than"  
                                # set Rdst if Rsrc1 < Rsrc2
```

Descrição: O registo "Rdst" toma o valor "1" se o conteúdo do registo "Rsrc1" for inferior ao do registo "Rsrc2". Caso contrário toma o valor "0".

```
slti Rdst, Rsrc1, Imm    # slt  $\equiv$  "set if less than"  
                                # set Rdst if Rsrc1 < Imm
```

A utilização das instruções "**bne**", "**beq**", "**slt**" e "**slti**", em conjunto com o registo **\$0**, permite a implementação de todas as condições de comparação entre dois registos e também entre um registo e uma constante: (**A = B**), (**A \neq B**), (**A > B**), (**A \geq B**), (**A < B**), (**A \leq B**)

Instruções virtuais de *branch* do MIPS

- Nos programas *Assembly*, podem ser utilizadas instruções de salto não diretamente suportadas pelo MIPS (**instruções virtuais**), mas que são **decompostas pelo assembler em instruções nativas**. Essas instruções são:

- **blt** **Rsrc1**, **Rsrc2**, **Label** **# Branch if Rsrc1 < Rsrc2**
- **ble** **Rsrc1**, **Rsrc2**, **Label** **# Branch if Rsrc ≤ Rsrc2**
- **bgt** **Rsrc1**, **Rsrc2**, **Label** **# Branch if Rsrc > Rsrc2**
- **bge** **Rsrc1**, **Rsrc2**, **Label** **# Branch if Rsrc ≥ Rsrc2**

- Nestas instruções **Rsrc2** pode ser substituído por uma **constante**.

- Exemplos:

- **blt** \$1, \$2, L2 **# if \$1 < \$2 goto L2**
- **bgt** \$1, 100, L3 **# if \$1 > 100 goto L3**

Como são decompostas estas instruções?

Decomposição das instruções virtuais BGT e BGE

A instrução virtual "**bge**" (*branch if greater or equal than*):

```
bge    $4, $7, exit    # if $4 ≥ $7 goto exit
                        # (i.e. goto exit if !($4 < $7))
```

É decomposta nas **instruções nativas**:

```
slt    $1, $4, $7      # $1 = 1 if $4 < $7 ($1=0 if $4 ≥ $7)
beq    $1, $0, exit    # if $1 = 0 goto exit
```

De modo similar, a instrução virtual "**bgt**" (*branch if greater than*):

```
bgt    $4, $7, exit    # if $4 > $7 goto exit
                        # (i.e. goto exit if $7 < $4)
```

É decomposta nas **instruções nativas**:

```
slt    $1, $7, $4      # $1 = 1 if $7 < $4 ($1=1 if $4 > $7)
bne    $1, $0, exit    # if $1 ≠ 0 goto exit
```

Decomposição das instruções virtuais BGT e BGE

Sobre este tema, pode encontrar informação complementar e mais detalhada no documento:

Como decompor instruções de logica relacional em instruções nativas

disponível na secção “Documentos de apoio às aulas teóricas e práticas” do moodle da UC.

Instrução de salto incondicional

- As arquiteturas disponibilizam também instruções de **salto incondicional**
- Numa instrução de **salto incondicional** não existe o teste de qualquer condição: **o salto é sempre realizado**
- No ISA do MIPS a instrução de salto incondicional tem a mnemónica "**j**", que significa *jump*

j label

- O fluxo de execução é desviado, de forma incondicional, para "label"

Estruturas de controlo de fluxo em C

- Exemplos

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

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

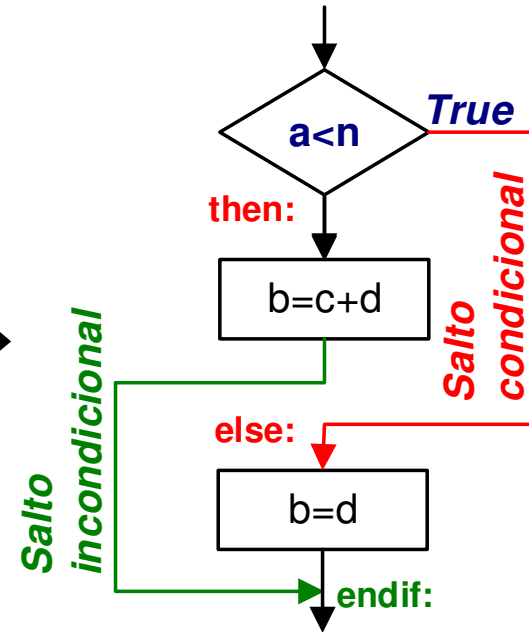
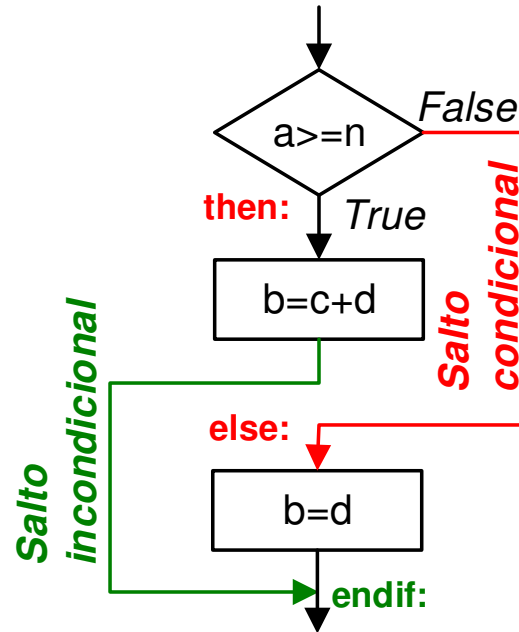
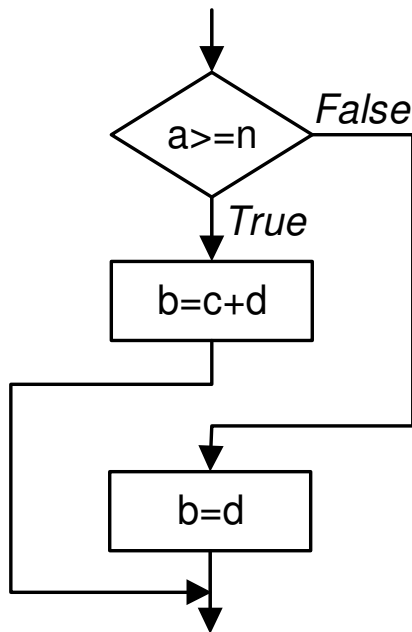
```
n = 0;  
do {  
    a = a + b[n];  
    n++;  
} while (n < 100);  
...
```

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

Tradução para *Assembly* do MIPS (`if()`... `then`... `else`)

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

- Transformando o código apresentado no fluxograma equivalente, é possível identificar a ocorrência de um **salto condicional** e de um **salto incondicional**
- E adaptar o salto condicional (usando a condição "complemento lógico") para que este se efetue quando a condição for verdadeira (tal como nos *branches*).

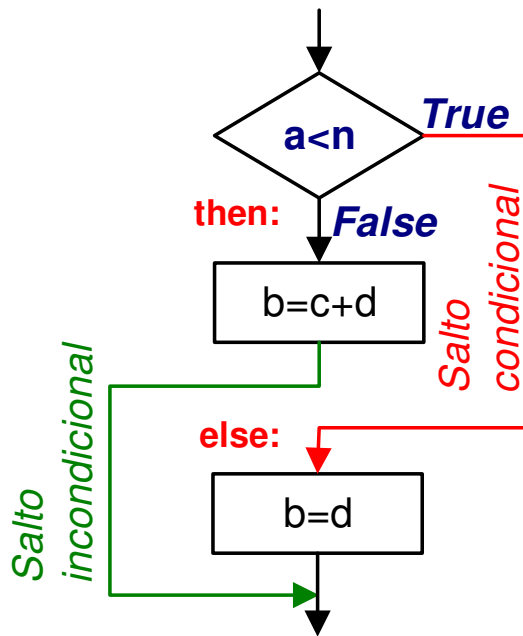


Tradução para *Assembly* do MIPS – *if()... then... else*

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

a: \$t0
n: \$t1
c: \$t2
b: \$t3
d: \$t4

Supondo que as variáveis residem nos registos \$t0 a \$t4, a tradução para *Assembly* fica:



```
blt    $t0, $t1, else # if (a >= n) {  
add    $t3, $t2, $t4 #   b = c + d;  
j      endif         # }  
else:  # else {  
or     $t3, $t4, $0  #   b = d;  
endif: ...          # }
```

j significa *jump* e representa um **salto incondicional** para o "label" indicado

Tradução para *Assembly* do MIPS - ciclos *for()* e *while()*

```
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 do corpo do ciclo** (inicializações)

Condição de continuação da execução do ciclo

Operações a realizar no **fim do corpo do ciclo**

Os 3 campos do ciclo "**for**" são opcionais. Exemplo:

```
for (; ; i++) {  
    ...  
}
```

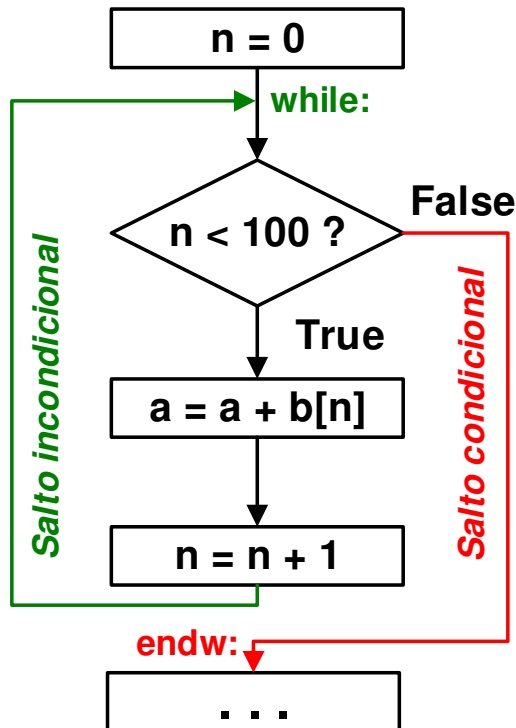
Qual o resultado deste código?

Tradução para *Assembly* do MIPS - ciclos *for()* e *while()*

```
int n;  
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 um **salto condicional** e de um **salto incondicional**
- O salto condicional necessita de ser modificado de forma a ser efetuado quando a condição for verdadeira
- Para isso usa-se o **complemento lógico** da condição presente no código original (para o exemplo, "**<**" passa a ser "**>=**")

Tradução para *Assembly* do MIPS - ciclos *for()* e *while()*

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

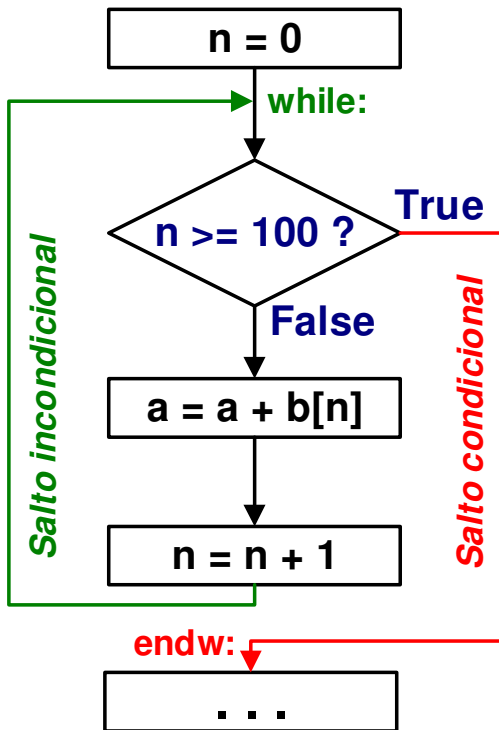
```
n = 0;  
while (n < 100) {  
    a = a + b[n];  
    n++;  
} ...
```



Complemento lógico:
"<" passa a ">="

Teste condicional feito
no início do ciclo

A tradução da estrutura de controlo, com "n"
a residir em \$t1, fica:

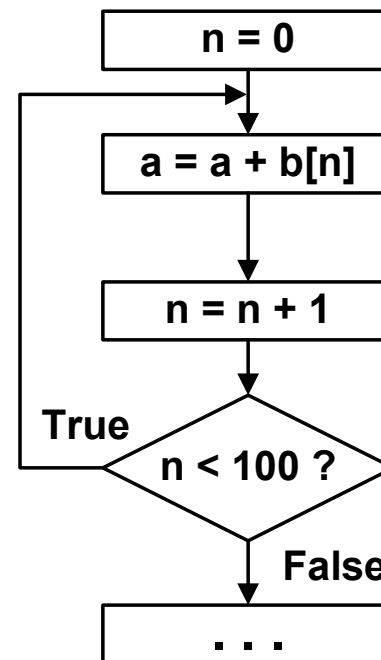


```
ori    $t1,$0,0           # n=0  
while: bge    $t1,100,endw # while (n<100)  
# {  
#     ...  
#     ...  
addi   $t1,$t1,1         # n++;  
j      while             # }  
endw:
```

Tradução para *Assembly* do MIPS - ciclo **do ... while()**

- Ao contrário do **for()** e do **while()**, o corpo do ciclo **do...while()** é executado incondicionalmente pelo menos uma vez!
- O teste da condição é efetuado no fim do ciclo

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

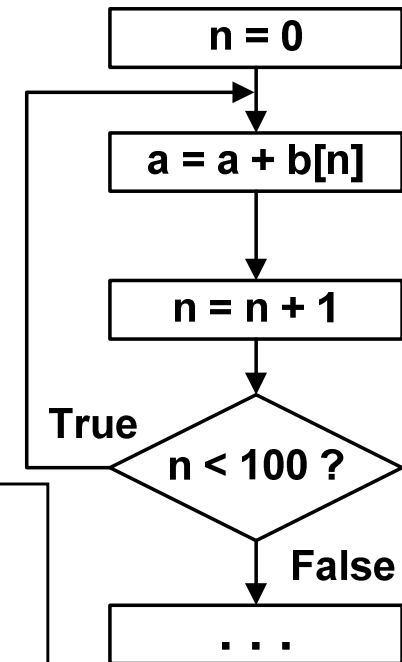


Tradução para *Assembly* do MIPS - ciclo *do ... while()*

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

A tradução para *Assembly* do MIPS fica (com "n" a residir em \$t1):

```
ori    $t1,$0,0      # n = 0;
do:    # do {
....    # a = a + b[n]
addi   $t1,$t1,1     # n = n + 1;
blt    $t1,100,do    # } while(n < 100);
....
```



Teste condicional feito no fim do ciclo. Mesma condição do código original.

Conclusão

- As estruturas do tipo ciclo incluem, geralmente, uma ou mais instruções de inicialização de variáveis, executadas antes e fora do mesmo
- No caso dos ciclos **for()** e do **while()** o teste condicional é executado no início do ciclo
- No caso do **do...while()** o teste condicional é efetuado no fim do ciclo, o que significa que o corpo do ciclo é executado pelo menos uma vez
- Na tradução de um ciclo **for()** para *Assembly*, o terceiro campo é codificado no fim do corpo do ciclo, ou seja, é a última instrução do ciclo antes do **j** que fecha o ciclo.

Questões / Exercícios

- Qual a função da instrução `"slt"`?
- Qual o valor armazenado no registo `$1` na execução da instrução `"slt $1, $3, $7"`, admitindo que: a) `$3=5` e `$7=23` e b) `$3=0xFE` e `$7=0x913D45FC`
- Com que registo comparam as instruções `"bltz"`, `"blez"`, `"bgtz"` e `"bgez"`?
- Decomponha em instruções nativas do MIPS as seguintes instruções virtuais:
 - `blt $15, $3, exit`
 - `ble $6, $9, exit`
 - `bgt $5, 0xA3, exit`
 - `bge $10, 0x57, exit`
 - `blt $19, 0x39, exit`
 - `ble $23, 0x16, exit`

Exercícios

- Traduza para *assembly* do MIPS os seguintes trechos de código de linguagem C (admita que **a**, **b** e **c** residem nos registos \$4, \$7 e \$13, respetivamente):

```
1)   if (a > b && b != 0)
      c = b << 2;
      else
      c = (a & b) ^ (a | b);
```

```
2)   if (a > 3 || b <= c)
      c = c - (a + b);
      else
      c = c + (a - 5);
```

- Na tradução para *assembly*, que diferenças encontra entre um ciclo do tipo "**while** (...) {...}" e um do tipo "**do**{...}**while** (...);"

Exercícios

- Traduza para *assembly* do MIPS os seguintes trechos de código de linguagem C (atribua registos internos para o armazenamento das variáveis **i** e **k**) :

```
1)   int i, k;
      for(i=5, k=0; i < 20; i++, k+=5);
```

```
2)   int i=100, k=0;
      for( ; i >= 0; )
      {
          i--;
          k -= 2;
      }
```

```
3)   unsigned int k=0;
      for( ; ; )
      {
          k += 10;
      }
```

```
4)   int k=0, i=100;
      do
      {
          k += 5;
      } while(--i >= 0);
```