Como decompor as instruções virtuais de lógica relacional em instruções nativas, percebendo o porquê.

Comecemos por considerar, para a análise que segue, a seguinte notação:

- **A** -> identifica o conteúdo de um registo do MIPS de uso geral expresso em complemento para 2
- **B** -> identifica o conteúdo de um segundo registo de uso geral expresso em complemento para 2
- **UA** -> identifica o conteúdo de um registo do MIPS de uso geral expresso em binário natural
- **UB** -> identifica o conteúdo de um segundo registo de uso geral expresso em binário natural
- **D** -> idêntica um registo que irá armazenar um resultado de uma operação lógica ou aritmética
- **imm** -> identifica uma constante (valor imediato) em complemento para dois
- **uimm** -> identifica uma constante (valor imediato) em binário natural (zero ou positiva)
- **$AT** -> identifica o registo $1 (reservado pelo Assembler para a decomposição de instruções virtuais em instruções nativas quando se torna necessário recorrer a um registo temporário)

- As instruções virtuais serão apresentadas em castanho (e.g. `bgt A, B, Label`)
- As instruções nativas serão apresentadas em azul (e.g. `beq A, B, Label`)

O Assembler do MIPS inclui um conjunto de instruções virtuais para realizar comparações entre o conteúdo de dois registos ou entre o conteúdo de um registo e uma constante. Mais especificamente, constam dessa lista as seguintes instruções:

1.

```
beqz  A, Label          # Salta para Label se A for igual a zero
bnez  A, Label          # Salta para Label se A for diferente de zero
```

No caso destas duas instruções, a comparação de **A** é feita implicitamente com a constante zero, que é também o conteúdo do registo $0. Logo, as instruções nativas equivalentes serão:

```
beqz  A, Label      ->    beq  A, $0, Label
bnez  A, Label      ->    bne  A, $0, Label
```

2.

```
beq  A, imm, Label          # Salta para Label se A for igual a imm
beq  A, uimm, Label         # Salta para Label se A for igual a uimm
bne  A, imm, Label          # Salta para Label se A for diferente de imm
bne  A, uimm, Label         # Salta para Label se A for diferente de uimm
```

No caso destas quatro instruções, a comparação de **A** é feita com uma constante de 16 bits (valor imediato) que pode ou não ter sinal. Logo, para se poder utilizar as instruções nativas, o valor da constante tem de ser previamente copiado para um registo temporário (**$AT**):

```
beq  A, imm, Label   ->    addi  $AT, $0, imm      # addi ⇒ extensão do sinal de
                           beq   A, $AT, Label     # imm #para os bits 31..17 de $AT

beq  A, uimm, Label  ->    xori  $AT, $0, uimm     # xori ⇒ bits 31..17 de $AT são
                           beq   A, $AT, Label     # mantidos a '0'

bne  A, imm, Label   ->    addi  $AT, $0, imm      # addi ⇒ extensão do sinal de
                           bne   A, $AT, Label     # imm para os bits 31..17 de $AT

bne  A, uimm, Label  ->    xori  $AT, $0, uimm     # xori ⇒ bits 31..17 de $AT são
                           bne   A, $AT, Label     # mantidos a '0'
```

3.

```
blt A,B,Label          # Se A < B então salta para Label
blt A,imm,Label        # Se A < imm então salta para Label
bltu UA,UB,Label       # Se UA < UB então salta para Label
bltu UA,uimm,Label     # Se UA < uimm então salta para Label

bgt A,B,Label          # Se A > B então salta para Label
bgt A,imm,Label        # Se A > imm então salta para Label
bgtu UA,UB,Label       # Se UA > UB então salta para Label
bgtu UA,uimm,Label     # Se IA > uimm então salta para Label

ble A,B,Label          # Se A ≤ B então salta para Label
ble A,imm,Label        # Se A ≤ imm então salta para Label
bleu UA,UB,Label       # Se UA ≤ UB então salta para Label
bleu UA,uimm,Label     # Se UA ≤ uimm então salta para Label

bge A,B,Label          # Se A ≥ B então salta para Label
bge A,imm,Label        # Se A ≥ imm então salta para Label
bgeu UA,UB,Label       # Se UA ≥ UB então salta para Label
bgeu UA,imm,Label      # Se UA ≥ uimm então salta para Label
```

Para perceber como decompor estas instruções virtuais em instruções nativas, temos de perceber quais as limitações que o *Assembly* nativo impõe, quer do ponto de vista das instruções nativas disponíveis para o efeito, quer do ponto de vista das regras sintáticas. As instruções que estão disponíveis para implementar todas as condições relacionais acima indicadas são apenas as seguintes:

Instrução Assembly		Código pseudo-C equivalente

`slt D, A, B`	->	`int A, B, D;`
		`if (A < B){`
		`  D = 1;`
		`} else {`
		`  D = 0;`
		`}`

`sltu D, UA, UB`	->	`unsigned int A, B, D;`
		`if (UA < UB){`
		`  D = 1;`
		`} else {`
		`  D = 0;`
		`}`

`slti D, A, imm`	->	`int A, D;`
		`if (A < imm){`
		`  D = 1;`
		`} else {`
		`  D = 0;`
		`}`

`sltiu D, UA, uimm`	->	`unsigned int UA, D;`
		`if (UA < uimm){`
		`  D = 1;`
		`} else {`
		`  D = 0;`
		`}`

`beq A, $0, Label`	->	`if (A == 0) goto Label;`
`bne A, $0, Label`	->	`if (A != 0) goto Label;`

Tabela 1 – Instruções em *Assembly* nativo que permitem implementar as condições de lógica relacional (>, ≥, < e ≤).

Como se pode perceber da tabela anterior, a principal instrução que nos permite comparar valores faz apenas comparações do tipo (N < I), em que 'N' pode ser **A** ou **UA** e 'I' pode ser **B**, **UB**, **imm** ou **uimm**.

Convém então relembrar as transformações entre as quatro expressões de lógica relacional (>, ≥, < e ≤) que permitem obter, a partir de cada uma dessa expressões, uma outra do tipo (N < I):

Caso	Condição	Valor Booleano	⇒	Condição do tipo (N < I)	Valor Booleano
1	(A > B)	true	⇒	(B < A)	true[1]
2	(A < B)	true	⇒	(A < B)	true
3	(A ≥ B)	true	⇒	(A < B)	false
4	(A ≤ B)	true	⇒	(B < A)	false

Tabela 2 – Transformação entre as quatro condições relacionais e uma condição do tipo (N < I) e respetivo resultado booleano.

Se admitirmos que todas as condições testadas são verdadeiras (porque é nesse caso que o *branch* é *taken*), então, dependendo da transformação que teremos de efetuar para obter uma expressão do tipo (N < I) poderemos ter como resultado booleano um valor verdadeiro (casos 1 e 2) ou falso (casos 3 e 4).

Consideremos, como exemplo, o caso 3. O código C (usando um goto) seria:

```
if (A >= B) goto Label;
```

mas, como não temos disponível uma instrução para determinar a condição ">=", teremos então de rescrever o código da seguinte forma:

```
if (!(A < B)) goto Label; // neste caso o branch é taken se o resultado
                          // de (A < B) for falso
```

Uma outra limitação imposta pelas instruções nativas do MIPS, resulta do facto de que as instruções que usam como operando um imediato (e.g. **`slti $AT, A, imm`**) obrigarem a que esse imediato seja sempre o terceiro operando. Ora, em situações como a do caso **1** da tabela 2, a obtenção de uma condição do tipo (N < I) obriga a trocar a ordem dos operandos. Quando os operandos são registos, essa troca não é problema. Mas nos casos em que um operando é um imediato a troca deixa de ser possível já que o imediato tem de ser, obrigatoriamente, o terceiro operando. Para resolver esse problema vai ser necessário começar por copiar o valor imediato para um registo (iremos usar o $AT). Desta forma passamos a ter os valores a comparar em dois registos e a troca de operandos volta novamente a ser possível.

Antes de construirmos a nossa tabela final, analisemos um exemplo concreto:

```
bgti A, uimm, Label          # caso 1 da tabela 2 usando um imediato
```

Neste caso teríamos de trocar **A** e **uimm** para transformar a condição numa outra da forma (uimm < A). Como não podemos usar um valor imediato como primeiro operando fonte, teremos de usar as seguintes duas instruções:

```
xori $AT, $0, uimm           # $AT passa a ter a constante uimm
slti $AT, $AT, A             # os operandos podem agora ser trocados ($AT < A)
```

se a condição testada pelo "**`slti`**" for verdadeira, o registo $AT ficará com o valor '1'. Logo, para executar o *branch,* falta apenas:

```
bne $AT, $0, Label           # salta para Label se ($AT != 0) ⇒ verdadeiro
```

[1] **Deve ler-se:** Se (A > B) é verdadeiro então (B < A) também é verdadeiro.

A tabela completa com a conversão de todas as instruções de *branch* condicional virtuais em falta será então:

Instrução virtual	Caso (Tabela 2)	Condição p/ branch taken	Instruções nativas	Troca de operandos
`blt A,B,Label`	2	true ⇒ $AT = 1	`slt $AT, A, B` `bne $AT, $0, Label`	Não
`blt A,imm,Label`	2	true ⇒ $AT = 1	`slti $AT, A, imm` `bne $AT, $0, Label`	Não
`bltu UA,UB,Label`	2	true ⇒ $AT = 1	`sltu $AT, UA, UB` `bne $AT, $0, Label`	Não
`bltu UA,uimm,Label`	2	true ⇒ $AT = 1	`sltiu $AT, UA, uimm` `bne $AT, $0, Label`	Não
`bgt A,B,Label`	1	true ⇒ $AT = 1	`slt $AT, B, A` `bne $AT, $0, Label`	Sim

`bgt A,imm,Label`	1	true ⇒ $AT = 1	`addi $AT, $0, imm`
			`slti $AT, $AT, A`
			`bne $AT, $0, Label`
				Sim

`bgtu UA,UB,Label`	1	true ⇒ $AT = 1	`sltu $AT, UB, UA` `bne $AT, $0, Label`	Sim

`bgtu UA,uimm,Label`	1	true ⇒ $AT = 1	`xori $AT, $0, imm`
			`Slti $AT, $AT, UA`
			`bne $AT, $0, Label`
				Sim

`ble A,B,Label`	4	false ⇒ $AT = 0	`slt $AT, B, A` `beq $AT, $0, Label`	Sim

`ble A,imm,Label`	4	false ⇒ $AT = 0	`addi $AT, $0, imm`
			`slt $AT, $AT, A`
			`beq $AT, $0, Label`
				Sim

`bleu UA,UB,Label`	4	false ⇒ $AT = 0	`sltu $AT, UB, UA` `beq $AT, $0, Label`	Sim

`bleu UA,uimm,Label`	4	false ⇒ $AT = 0	`xori $AT, $0, imm`
			`sltu $AT, $AT, UA`
			`beq $AT, $0, Label`
				Sim

`bge A,B,Label`	3	false ⇒ $AT = 0	`slt $AT, A, B` `beq $AT, $0, Label`	Não
`bge A,imm,Label`	3	false ⇒ $AT = 0	`slti $AT, A, imm` `beq $AT, $0, Label`	Não
`bgeu UA,UB,Label`	3	false ⇒ $AT = 0	`sltu $AT, UA, UB` `beq $AT, $0, Label`	Não
`bgeu UA,imm,Label`	3	false ⇒ $AT = 0	`sltiu $AT, UA, imm` `beq $AT, $0, Label`	Não