Controle de fluxo de execução (i)

Fluxo sequencial de execução:
  próxima instrução é aquela em PC+4

Instruções para efetuar Desvios if( ){ } while( ){ }

```
beq r1, r2, ender  # branchEqual desvia se r1 == r2
bne r1, r2, ender  # branchNotEq desvia se r1 != r2
```

se desvia PC ← ender ; senão PC ←PC+4

Instruções para efetuar Saltos goto

```
j ender            # jump (salto incondicional)
jr rt              # jump register ; rt = ender destino
```

PC ← ender

desvios são condicionais ; saltos são incondicionais

Controle de fluxo de execução (ii)

Instruções para efetuar Desvios if( ){ } while( ){ }

```
beq r1, r2, ender    # Branch Equal  desvia se r1 == r2
bne r1, r2, ender    # Branch Not Eq desvia se r1 != r2
```

Instruções para comparação de magnitude

```
slt rd, r1, r2       Set on Less Than
rd ← 1 se r1 < r2    em C: rd = ((r1 < r2) ? 1 : 0);

slt rd, r1, r2       # rd ← 1 se ( r1 < r2 )
slti rd, r1, const   # rd ← 1 se ( r2 < ext(const) )
sltu rd, r1, r2      # subtração não gera exceção
sltiu rd, r1, const  # subtração não gera exceção
```

Controle de fluxo de execução (iii)

```
beq r1, r2, ender      # branchEqual desvia se r1 == r2
bne r1, r2, ender      # branchNotEq desvia se r1 != r2
slt rd, r1, r2         # rd ← 1 se r1 < r2, senão 0
```

em C, 1=TRUE e 0=FALSE

sequência equivalente a `blt` (branch on less than)

```
slt r1, r2, r3         # r1 ← (r2 < r3)     T ou F
bne r1, $zero, ender   # salta se TRUE ≠ FALSE
```

sequência equivalente a bge (branch if greater or equal)

```
slt r1, r2, r3         # r1 ← (r2 < r3)     T ou F
beq r1, $zero, ender   # salta se FALSE = FALSE
```

Desvios: endereço de destino

`beq r1, r2, desloc`

Quando o desvio for tomado, o endereço de destino é:

$(\text{PC} + 4) + \mathit{extSinal}(\texttt{desloc}) \times 4$

`desloc` é o número de instruções por saltar (c.r.a PC+4)

```
     beq r1, r2, desloc   # salta para L1
     add ...
     sub ...
     xor ...
L1:  sw ...               # qual o valor em desloc?
```

Atenção: é o montador que traduz o endereço do label para `desloc`

Desvios: endereço de destino (cont)

Qual o efeito das sequências abaixo? nop = no-operation

```
L1:  nop
L2:  beq r1, r1, -1
L3:  nop

L4:  nop
L5:  beq r1, r1, 0
L6:  nop

L7:  nop
L8:  beq r1, r1, +1
L9:  nop
```

Desvios e Saltos (i)

```
if (i == j) goto L1;          beq ri, rj, L1
  f = g + h;                  add rf, rg, rh
L1:                     L1:   sub rf, rf, ri
f = f - i;
```

```
if (i == j)                   bne ri, rj, Else
  f = g + h;                  add rf, rg, rh
else                          j Exit    # salta else
  f = g - h;            Else: sub rf, rg, rh
                        Exit: nop
```

Desvios e Saltos (ii)

```
while (save[i] == k)
  i = i + j;

# i,j,k <-> r19,r20,r21
       la r7, save          # r7 ← &(save[0])
Loop:  sll r9, r19, 2       # r9 ← i*4   (i<<2)
       add r9, r7, r9       # r9 ← (i*4 + save)
       lw r8, 0(r9)         # r8 ← M[ i*4 + save ]
       bne r8, r21, Exit    # termina se save[i] ≠ k
       add r19, r19, r20    # i ← i + j
       j Loop               # repete
Exit:  nop
```

Estruturas de Dados em C – strings

Strings são sequências de caracteres terminados por `'\0'` = 0x00

Uma *string* é um 'vetor' do tipo `char`, de tamanho "indefinido":
⇝ o fim da *string* armazenada num vetor é a posição do `'\0'`
no código fonte, *strings* são representadas entre aspas duplas,
e caracteres representados entre aspas simples.

No código fonte: `"palavra"` ⇝ o `'\0'` não é mostrado
Em memória, com a *string* alocada no endereço 30:

ender	30	31	32	33	34	35	36	37
	'p'	'a'	'l'	'a'	'v'	'r'	'a'	'\0'

Quando se computa o tamanho da *string*, o `'\0'` é contado

Código ASCII

Caracteres tipicamente são codificados em ASCII:
cada caractere é representado em 7 bits e armazenado num byte

O dígito 0 é representado pelo caractere `'0'` que é 0x30,
1 é 0x31, 2 é 0x32, ... 9 é 0x39

maiúsculas: `A=0x41, B=0x42, ..., Z=0x5A`
minúsculas: `a=0x61, b=0x62, ..., z=0x7A`

espaço é 0x20, exclamação é 0x21, aspas duplas é 0x22, ...

`man ascii` para a tabela completa

o código ASCII foi projetado para representar o idioma Inglês e portanto não provê acentuação;
`man iso_8859-1` para codificação com acentos (Latin-1) em 8 bits

Manipulação de *strings* em *assembly* (i)

Trecho de código que copia uma cadeia para um vetor de characteres:

```
char fte[16]="abcd-efgh-ijkl-";   // 16 contando '\0'
char dst[32];
int i;

i = 0;
while ( fte[i] != '\0' ) {   // terminou?
   dst[i] = fte[i];
   i = i + 1;
}
dst[i] = '\0';   // laço não copia o '\0'
```

Manipulação de *strings* em *assembly* (ii)

```
       la r8, fte            # r8 ← fte
       la r9, dst            # r9 ← dst
       addi r4,$zero,$zero   # i = 0;
                             # while (fte[i] != '\0') {
lasso: add r18,r8,r4         # r18 ← fte+i
       lbu r5, 0(r18)        # r5 ← fte[i], extZero
       beq r5, $zero, fim    # (r5 = '\0') ? terminou
       add r19,r9,r4         # r19 ← dst+i
       sb r5, 0(r19);        # dst[i] ← fte[i]
       addi r4,r4,1          # i = i + 1;
       j lasso               # }
fim:   sb $zero, 0(r9)       # dst[i] = '\0';
```

Exercícios

Traduza de C para *assembly* do MIPS:

```
char fte[NN]; char dst[NN];
int i, num;

i = 0;
while (fte[i] != '\0') {
   i = i + 1;
}
num = i - 1;
i = 0;
while (num >= 0) {          // reverte a cadeia
   dst[i] = fte[num];
   num = num - 1; i = i+1;
}
dst[i] = '\0';
```