

AULA PRÁTICA N.º 3

Objetivos:

- Implementação e codificação de estruturas de controlo de fluxo do tipo **`if..else`**, **`for()`**, **`while()`**, **`do..while()`**.

Conceitos básicos:

- Instruções de alteração de fluxo de execução no MIPS.
- Codificação de lógica relacional.

Guião:

1. O programa seguinte lê do teclado 5 valores inteiros (positivos ou negativos) e apresenta a soma dos valores positivos.

```

void main(void)
{
    int soma, value, i;

    for(i=0, soma=0; i < 5; i++)
    {
        print_string("Introduza um numero: ");
        value = read_int();
        if(value > 0)
            soma = soma + value;
        else
            print_string("Valor ignorado\n");
    }
    print_string("A soma dos positivos e': ");
    print_int10(soma);
}

```

- a) Codifique o programa em *assembly* do MIPS e teste o seu funcionamento no MARS. Para o armazenamento das variáveis do programa utilize registo \$tN, onde N=0, 1, ..., 9.

Tradução incompleta para *assembly*:

```

# Mapa de registos:
# soma:    $t0
# value:   $t1
# i:       $t2
.data
str1: .asciiz    "Introduza um numero: "
str2: ...
str3: ...
.eqv  print_string, 4
.eqv  ...

```

```

        .text
        .globl main
main: li    $t0,0          # soma = 0;
      li    ...
for:   b??   $t2,...,endfor # while(i < 5) {
      (...)           #   print_string("...");
      (...)           #   value=read_int();
if:    b??   $t1,...,else  #   if(value > 0)
      (...)           #     soma += value;
      j    ...
else: (...)             #   else
      #     print_string("...");
endif: addi  $t2,...       #   i++;
      j    for            # }
endfor:
      (...)           #   print_string("...");
      (...)           #   print_int10(soma);
      jr   $ra

```

- b) Observe, no MARS, a decomposição da instrução virtual "ble \$t1,\$0,else" (coluna "Native Instruction"); anote os endereços de memória onde as instruções resultantes da decomposição estão armazenadas.
2. O programa seguinte lê um número introduzido pelo utilizador e apresenta esse mesmo valor representado em binário.

```

void main(void)
{
    unsigned int value, bit, i;

    print_string("Introduza um numero: ");
    value = read_int();
    print_string("\nO valor em binário é: ");
    for(i=0; i < 32; i++)
    {
        bit = value & 0x80000000; // isola bit 31
        if(bit != 0)
            print_char('1');
        else
            print_char('0');
        value = value << 1;      // shift left de 1 bit
    }
}

```

- a) Codifique o programa em *assembly* do MIPS e teste o seu funcionamento no MARS. Para o armazenamento das variáveis do programa utilize registos \$tN, onde N=0, 1, ..., 9.
- Tradução incompleta para *assembly*:

```

# Mapa de registos:
# value: $t0
# bit:   $t1
# i:     $t2
        .data
str1: .asciiz ...
str2: .asciiz ...
        .eqv   print_string,4
        .eqv   ...
        .eqv   ...

```

```

.text
.globl main
main: la    ...          # (instrução virtual)
      li    ...          # (instrução virtual)
      syscall
      (...)

      li    ...
for:   b??   $t2,...,endfor
      li    $t3,0x80000000
      and   $t1,...,...
if:   b??   $t1,...,else
      (...)

      else:
      (...)

      endif:
      j     ...
endfor:
      jr   $ra           # fim do programa

```

- b) Altere o programa anterior de modo a imprimir um espaço entre cada grupo de 4 bits (por exemplo, **1010 0110 0001 ...**). Sugestão para implementar esta alteração:

```

(...)

for(i=0; i < 32; i++)
{
    if((i % 4) == 0)           // resto da divisão inteira
        print_char(' ');
    bit = value & 0x80000000; // isola bit 31
    (...)
```

Nota: em *assembly* do MIPS, o resto da divisão inteira pode ser obtido usando a instrução virtual "**rem**" (remainder), "**rem Rdst, Rsrc, Src**", em que **Src** pode ser um registo ou uma constante (Exemplo: **rem \$t0, \$t1, 4** coloca em **\$t0** o resto da divisão inteira do valor de **\$t1** por 4).

- c) Sabendo que o código ASCII do 0 é **0x30** e do 1 é **0x31**, o programa anterior pode ser simplificado, eliminando a estrutura condicional. Ou seja, a sequência:

```

bit = value & 0x80000000;      // isola bit 31
if(bit != 0)
    print_char('1');
else
    print_char('0');
```

pode ser substituída por:

```

bit = (value & 0x80000000) >> 31;
// ou, em alternativa: bit = (value >> 31) & 0x00000001;
// ou, como value é do tipo unsigned: bit = value >> 31;

print_char(0x30 + bit);       // Ou: print_char('0'+ bit);
```

Inclua esta alteração no programa em C. Traduza as alterações para *assembly* e teste o resultado no MARS.

- d) Altere o programa em C que resultar da alínea anterior, de modo a que não sejam impressos os zeros à esquerda. O programa seguinte apresenta uma possível solução.

```

int flag;
(...)

for(i=0, flag=0; i < 32; i++)
{
    bit = value >> 31;
    if(flag == 1 || bit != 0)
    {
        flag = 1;
        if((i % 4) == 0)
            print_char(' ');
        print_char(0x30 + bit);
    }
    value = value << 1;
}

```

Codifique as alterações em *assembly* e teste novamente o programa.

- e) Reescreva o programa em C que resultou da alínea anterior, de modo a substituir o ciclo "**for**" por uma estrutura do tipo "**do ... while**". Reflita essas alterações no programa *assembly* e teste o código resultante no MARS.

Exercícios adicionais

- O programa seguinte converte uma quantidade, introduzida pelo utilizador, considerada em código Gray para o equivalente em binário natural.

```

void main(void)
{
    unsigned int gray, bin, mask;

    print_string("Introduza um numero: ");
    gray = read_int();

    mask = gray >> 1;
    bin = gray;
    while(mask != 0)
    {
        bin = bin ^ mask;
        mask = mask >> 1;
    }

    print_string("\nValor em código Gray: ");
    print_int16(gray);
    print_string("\nValor em binario: ");
    print_int16(bin);
}

```

- a) Codifique o programa em *assembly* do MIPS e teste o seu funcionamento no MARS. Para o armazenamento das variáveis do programa utilize registo \$tn.

2. A multiplicação de inteiros pode ser realizada recorrendo a sucessivas operações de deslocamentos e somas, tal como esquematizado na figura seguinte para operandos de 4 bits (o algoritmo usado é em tudo semelhante ao que todos aprendemos no ensino básico, aplicado agora a quantidades expressas em binário).

$$\begin{array}{r}
 & \begin{array}{r} 0 & 1 & 0 & 1 \end{array} \\
 \times & \begin{array}{r} 0 & 1 & 1 & 0 \end{array} \\
 \hline
 & \begin{array}{r} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \text{ Res. Inicial} \\
 + & \begin{array}{r} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \text{ 0.mdo.}2^0 \\
 \hline
 & \begin{array}{r} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \text{ 1.mdo.}2^1 \\
 + & \begin{array}{r} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{array} \text{ 1.mdo.}2^2 \\
 \hline
 & \begin{array}{r} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array} \text{ 0.mdo.}2^3 \\
 + & \begin{array}{r} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \\
 \hline
 & \begin{array}{r} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array} \text{ Res. FINAL}
 \end{array}$$

A implementação desse algoritmo, para quantidades de 4 bits, pode ser efetuada do modo que o programa seguinte apresenta.

```

void main(void)
{
    unsigned int res=0, i=0, mdor, mdo;

    print_string("Introduza dois numeros: ");
    mdor = read_int() & 0x0F;
    mdo = read_int() & 0x0F;

    while( (mdor != 0) && (i < 4) )
    {
        if( (mdor & 0x00000001) != 0 )
            res = res + mdo;
        mdo = mdo << 1;
        mdor = mdor >> 1;
        i++;
    }
    print_string("Resultado: ");
    print_int10(res);
}

```

- a) Codifique o programa em *assembly* do MIPS e teste o seu funcionamento no MARS. Para o armazenamento das variáveis do programa utilize registos \$tn.
- b) Execute o programa passo a passo e preencha a tabela seguinte, supondo que os valores iniciais do multiplicador e multiplicando são 11 e 6, respectivamente.

mdor	mdo	i	res	
0x0B	0x06			Valores iniciais
				Fim da 1ª iteração
				Fim da 2ª iteração
				Fim da 3ª iteração
				Fim da 4ª iteração

- c) O programa apresentado apenas manipula valores de 4 bits. Altere o programa em C de modo a poder manipular valores de 16 bits. Codifique as alterações em *assembly* e teste o funcionamento com valores de entrada de 16 bits.