AULA PRÁTICA N.º 7

Objetivos:

- Implementação de sub-rotinas.
- Utilização da convenção do MIPS para passagem de parâmetros e uso dos registos.
- Implementação e utilização da *stack* no MIPS. Parte 1.

Guião:

Neste guião vão ser implementadas algumas funções para manipulação de *strings*, habitualmente disponíveis nas bibliotecas de suporte à linguagem C.

1. A função **`strlen()`** determina e devolve a dimensão de uma *string* (como já visto anteriormente, em linguagem C uma *string* é terminada com o carater **`'\0'`**). O parâmetro de entrada dessa função é um ponteiro para o início da *string* (i.e., o seu endereço inicial) e o resultado é o número de carateres dessa *string* (excluindo o terminador).

```
int strlen(char *s)
{
   int len=0;
   while(*s++ != '\0')
       len++;
   return len;
}
```

a) Traduza a função anterior para *assembly*, aplicando as regras de utilização de registos e de passagem e devolução de valores do MIPS (veja o resumo das regras no final do guião).
Tradução parcial do código anterior para *assembly*:

```
# O argumento da função é passado em $a0
# O resultado é devolvido em $v0
# Sub-rotina terminal: não devem ser usados registos $sx
strlen:   li      $t1,0           # len = 0;
while:    lb      $t0,...         # while(*s++ != '\0')
          addiu   $a0,$a0,1       #
          b??     $t0,'\0',endw   # {
          addi    $t1,...         #    len++;
          j       ...             # }
endw:     move    $v0,$t1         # return len;
          jr      $ra             #
```

b) Para teste da função **`strlen()`** o programa seguinte imprime o número de carateres de uma *string* definida de forma estática no programa. Traduza esse programa para *assembly* aplicando as regras de utilização de registos e de passagem e devolução de valores. **Não se esqueça que a função `main()` é, em termos de implementação, tratada como qualquer outra função**.

```
int strlen(char *s);

int main(void)
{
   static char str[]="Arquitetura de Computadores I";

   print_int10(strlen(str));
   return 0;
}
```

2. A função **`strrev()`** (*string reverse*) inverte o conteúdo de uma *string*. Tal como no caso da função anterior, o parâmetro de entrada dessa função é um ponteiro para o início da *string*, i.e., o seu endereço inicial. A função retorna o ponteiro com o mesmo valor que foi passado como argumento.

```
void exchange(char *, char *);

char *strrev(char *str)
{
    char *p1 = str;
    char *p2 = str;

    while(*p2 != '\0')
        p2++;
    p2--;
    while( p1 < p2 )
    {
        exchange(p1, p2);
        p1++;
        p2--;
    }
    return str;
}
```

```
void exchange(char *c1, char *c2)
{
    char aux = *c1;

    *c1 = *c2;
    *c2 = aux;
}
```

a) Traduza a função **`strrev()`** para *assembly*, completando o código seguinte:

```
# Mapa de registos:
# str: $a0 -> $s0 (argumento é passado em $a0)
# p1:  $s1  (registo callee-saved)
# p2:  $s2  (registo callee-saved)
#
strrev:   addiu   $sp,$sp,-16     # reserva espaço na stack
          sw      $ra,0($sp)      # guarda endereço de retorno
          sw      $s0,4($sp)      # guarda valor dos registos
          sw      $s1,8($sp)      #  $s0, $s1 e $s2
          sw      $s2,12($sp)     #
          move    $s0,$a0         # registo "callee-saved"
          move    $s1,$a0         # p1 = str
          move    $s2,$a0         # p2 = str
while1:                           # while( *p2 != '\0' ) {
          (...)                   #   p2++;
          j       ...             # }
          (...)                   # p2--;
while2:                           # while(p1 < p2) {
          (...)
          move    $a0,...         #
          move    $a1,...         #
          jal     exchange        #    exchange(p1,p2)
          (...)
          j       ...             # }
          move    $v0,$s0         # return str
          lw      $ra,...         # repõe endereço de retorno
          lw      $s0,...         # repõe o valor dos registos
          lw      $s1,...         #  $s0, $s1 e $s2
          lw      $s2,...         #
          addiu   $sp,...         # liberta espaço da stack
          jr      $ra             # termina a sub-rotina
```

b) O programa seguinte visa o teste da função **strrev()**. Traduza esse programa para *assembly* aplicando as regras de utilização de registos e de passagem e devolução de valores. Teste o resultado no simulador MARS.

```
char *strrev(char *);

int main(void)
{
   static char str[]="ITED - orievA ed edadisrevinU";

   print_string( strrev(str) );
   return 0;
}
```

3. A função **strcpy()** (*string copy*) copia uma *string* residente numa zona de memória para outra zona de memória. A função aceita como argumentos um ponteiro para a *string* de origem (**src**) e um ponteiro para a zona de memória destino (**dst**). A função devolve ainda o ponteiro **dst** com o mesmo valor que foi passado como argumento.

```
char *strcpy(char *dst, char *src)
{
   int i=0;
   do
   {
       dst[i] = src[i];
   } while(src[i++] != '\0');
   return dst;
}
```

a) Traduza a função **strcpy()** para *assembly*.

b) Traduza para *assembly* e teste o funcionamento da função **main()**, apresentada de seguida, para chamada e teste da função **strcpy()**, que usa também as funções **strlen()** e **strrev()** implementadas anteriormente.

```
#define STR_MAX_SIZE 30
char *strcpy(char *dst, char *src);

int main(void)
{
   static char str1[]="I serodatupmoC ed arutetiuqrA";
   static char str2[STR_MAX_SIZE + 1];
   int exit_value;

   if(strlen(str1) <= STR_MAX_SIZE) {
       strcpy(str2, str1);
       print_string(str2);
       print_string("\n");
       print_string(strrev(str2));
       exit_value = 0;
   } else {
       print_string("String too long: ");
       print_int10(strlen(str1));
       exit_value = -1;
   }
   return exit_value;
}
```

c) Reescreva, em C, a função **strcpy()** usando acesso por ponteiro em vez de acesso indexado. Faça as correspondentes alterações no programa *assembly* e teste o resultado.

```
char *strcpy(char *dst, char *src)
{
    char *p=dst;
    do
    {
        *p++ = ...
    } while(*src++ ...
    return dst;
}
```

4. A função **strcat()** (*string concatenate*) permite concatenar duas *strings* – a *string* origem é concatenada no fim (isto é, na posição do terminador) da *string* destino. Tal como na função **strcpy()**, os argumentos de entrada são os ponteiros para a *string* origem (**src**) e para a *string* destino (**dst**). A função devolve ainda o ponteiro **dst** com o mesmo valor que foi passado como argumento. Compete ao programa chamador reservar espaço em memória com dimensão suficiente para armazenar a *string* resultante.

```
char *strcat(char *dst, char *src)
{
    char *p = dst;

    while(*p != '\0')
        p++;
    strcpy(p, src);
    return dst;
}
```

a) Traduza a função **strcat()** para *assembly* (não se esqueça de aplicar corretamente a convenção para a salvaguarda de registos; note ainda que **strcat()** é uma função intermédia).

b) Traduza para *assembly* a função **main()**, apresentada de seguida, para chamada e teste da função **strcat()**.

```
char *strcpy(char *dst, char *src);
char *strcat(char *dst, char *src);

int main(void)
{
    static char str1[]="Arquitetura de ";
    static char str2[50];

    strcpy(str2, str1);
    print_string(str2);
    print_string("\n");
    print_string( strcat(str2, "Computadores I") );
    return 0;
}
```

Exercícios adicionais

1. A função **insert()** insere o valor "**value**" na posição "**pos**" do "**array**" de inteiros de dimensão "**size**". A função **print_array()** imprime os valores de um *array* "**a**" de "**n**" elementos inteiros.

```
int insert(int *array, int value, int pos, int size)
{
    int i;

    if(pos > size)
        return 1;
    else
    {
        for(i = size-1; i >= pos; i--)
            array[i+1] = array[i];
        array[pos] = value;
        return 0;
    }
}

void print_array(int *a, int n)
{
    int *p = a + n;

    for(; a < p; a++)
    {
        print_int10( *a );
        print_string(", ");
    }
}
```

Traduza as duas funções anteriores para *assembly* (não se esqueça de aplicar a convenção de utilização de registos).

2. O programa seguinte permite o teste das funções desenvolvidas no exercício anterior. Traduza esse programa para *assembly* e teste-o no MARS.

```
int insert(int *, int, int, int);
void print_array(int *, int);

int main(void)
{
    static int array[50];
    int i, array_size, insert_value, insert_pos;
    print_string("Size of array : ");
    array_size = read_int();
    for(i=0; i < array_size; i++)
    {
        print_string("array[");
        print_int10(i);
        print_string("] = ");
        array[i] = read_int();
    }
```

```
    print_string("Enter the value to be inserted: ");
    insert_value = read_int();
    print_string("Enter the position: ");
    insert_pos = read_int();

    print_string("\nOriginal array: ");
    print_array(array, array_size);

    insert(array, insert_value, insert_pos, array_size);

    print_string("\nModified array: ");
    print_array(array, array_size + 1);
    return 0;
}
```

Anexo:

Regras para a implementação de sub-rotinas no MIPS

1. A sub-rotina chamadora, antes de chamar:
   - Passa os parâmetros; os 4 primeiros são passados nos registos **`$a0..$a3`** e os restantes na *stack*.
   - Executa a instrução "**`jal`**".
2. A sub-rotina chamada, no início:
   - Salvaguarda na *stack* os registos **`$s0 a $s7`** que pretende utilizar.
   - Salvaguarda o registo **`$ra`** no caso de a rotina também ser chamadora.
3. A sub-rotina chamada, no fim:
   - Coloca o valor de retorno em **`$v0`** (exceto se for tipo **`void`**).
   - Restaura os registos **`$s0`** a **`$s7`** que salvaguardou no início.
   - Restaura o registo **`$ra`** (no caso de ter sido salvaguardado no início).
   - Retorna, executando a instrução "**`jr $ra`**".
4. A sub-rotina chamadora, após regresso:
   - Usa o valor de retorno que está em **`$v0`**.
5. A sub-rotina chamadora não pode assumir em caso algum que qualquer dos registos **`$a0..$a3`**, **`$t0..$t9`**, **`$v0`** e **`$v1`** têm o conteúdo preservado pela rotina chamada.
6. A codificação da sub-rotina "**`main`**" está sujeita às mesmas regras que se aplicam às restantes sub-rotinas.

Considerações práticas sobre a convenção de utilização de registos

1. Sub-rotinas terminais (que não chamam qualquer sub-rotina):
   - Só devem utilizar (preferencialmente) registos que não necessitam de ser salvaguardados (**`$t0..$t9`**, **`$v0..$v1`** e **`$a0..$a3`**).

2. Sub-rotinas intermédias (que chamam outras sub-rotinas):
   - Devem utilizar os registos **`$s0..$s7`** para o armazenamento de valores que se pretenda preservar ao longo da execução de toda a sub-rotina. A utilização destes registos implica a sua prévia salvaguarda na *stack* logo no início da sub-rotina e a respetiva reposição no final.
   - Devem utilizar os registos **`$t0..$t9`**, **`$v0..$v1`** e **`$a0..$a3`** para os restantes valores.

PDF criado em 02/09/2022