1º Semestre de 2007/2008

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Aula 8

Acesso sequencial a elementos de um *array* residente em memória:

- *Arrays versus* ponteiros

Estruturas de controlo de fluxo de execução do tipo ***switch()***

Procedimentos: Invocação e retorno

Directivas do *Assembler*:

`.ASCIIZ` ***str***	Reserva espaço e armazena a string *str* em sucessivas posições de memória; acrescenta o terminador `'\0'` **(NULL)**
`.SPACE` ***n***	Reserva *n* posições de memória (sem inicialização)
`.BYTE` $b_1,b_2,...,b_n$	Reserva espaço e armazena os *bytes* $b_1,b_2,...,b_n$ em sucessivas posições de memória
`.WORD` $w_1,w_2,...,w_n$	Reserva espaço e armazena as *words* $w_1,w_2,...,w_n$ em sucessivas posições de memória (cada *word* em 4 posições)
`.ALIGN` ***n***	Alinha o próximo item num endereço múltiplo de 2^n

Directivas do *Assembler* - Exemplo:

```
      .DATA   # 0x10010000
STR1:.ASCIIZ  "AULA8"
      .ALIGN  2
ARR1:.WORD    0x1234, MAIN
VAR1:.BYTE    0x12
ARR2:.SPACE   2
STR2:.ASCIIZ  ":)"
      .TEXT   # 0x00400000
      .GLOBL  MAIN
MAIN:
```

	0x10010015	'\0' (0x00)
	0x10010014	')' (0x29)
STR2	0x10010013	':' (0x3A)
	0x10010012	????????
ARR2	0x10010011	????????
VAR1	0x10010010	0x12
	0x1001000F	0x00
	0x1001000E	0x40
	0x1001000D	0x00
	0x1001000C	0x00
	0x1001000B	0x00
	0x1001000A	0x00
	0x10010009	0x12
ARR1	0x10010008	0x34
	0x10010007	????????
	0x10010006	????????
	0x10010005	'\0' (0x00)
	0x10010004	'8' (0x38)
	0x10010003	'A' (0x41)
	0x10010002	'L' (0x4C)
	0x10010001	'U' (0x55)
STR1	0x10010000	'A' (0x41)

Linguagem C

Ponteiros e endereços – o operador &

- Um ponteiro é uma variável que contém o endereço de outra variável. O acesso à variável faz-se, assim, indirectamente através do ponteiro
- Exemplo:
  - "**x**" é uma variável (por ex. um inteiro) e "**px**" é um ponteiro. O endereço da variável "**x**" pode ser obtido através do operador "**&**", do seguinte modo:

```
px = &x;    // Atribui o endereço de "x" a "px"
```

  - Diz-se que "**px**" é um ponteiro que aponta para "**x**"
- O operador "**&**" apenas pode ser utilizado com variáveis e elementos de arrays. Exemplos de utilizações incorrectas:

`&5;` `&(x+1);` (sendo x uma variável do tipo inteiro)

Ponteiros e endereços – o operador *

- O operador "*****" trata o seu operando como um endereço. Permite o acesso ao endereço para obter o respectivo conteúdo.
- Exemplo:

```
y = *px;  // Atribui o conteúdo do endereço
          // apontado por "px" a "y"
```

- A sequência:

```
px = &x;
y = *px;
```

Atribui a "**y**" o mesmo valor que:

```
y = x;
```

Ponteiros e endereços – declaração de variáveis

- As variáveis envolvidas têm que ser declaradas. Para o exemplo anterior, supondo que se tratava de variáveis inteiras:

```
int  x, y;// x, y:variáveis do tipo inteiro
int  *px; // ou int* px;
```

- A declaração do ponteiro (`int *px;`) deve ser entendida como uma mnemónica e significa que `px` é um ponteiro e que o conjunto `*px` é do tipo inteiro.
- Exemplos de declarações de ponteiros

```
char  *p; // p é um ponteiro para caracter
double *v;// v é um ponteiro para double
```


Ponteiros – manipulação em expressões

- Exemplo: supondo que `px` aponta para `x` (`px = &x;`), a expressão `y = *px + 1;` atribui a `y` o valor de `x` acrescido de 1
- Os ponteiros podem igualmente ser utilizados na parte esquerda de uma expressão. Por exemplo, (supondo que `px = &x;`)

```
*px = 0; // equivalente a x=0
```

ou

```
*px = *px + 1; // equiv. a x = x + 1
*px += 1;
(*px)++;
```

Ponteiros como argumentos de funções

- Em C os argumentos das funções são passados por valor (cópia do conteúdo da variável original). Isso significa que não existe uma forma directa de uma função chamada alterar uma variável da função chamadora
- Se tal for necessário, a solução reside na utilização de ponteiros
- Suponhamos que se pretende implementar uma função para a troca do conteúdo de duas variáveis (**troca(a, b);**)

```
troca(a, b);
…
void troca(int x, int y)
{
  int aux;
  aux = x;
  x = y;
  y = aux;
}
```

```
troca(&a, &b);
…
void troca(int *x, int *y)
{
  int aux;
  aux = *x;
  *x = *y;
  *y = aux;
}
```

apenas o segundo trecho de código produz o efeito desejado

Ponteiros e arrays

- Sejam as declarações

```
int a[10];   // array de inteiros "a" com
             // 10 elementos
int *pa;     // ponteiro para um inteiro
int v;       // variável do tipo inteiro
```

- A expressão `pa = &a[0];` atribui a "`pa`" o endereço do 1º elemento do array. Então, a expressão `v = *pa;` atribui a "`v`" o valor de `a[0]`
- Se "`pa`" aponta para um dado elemento do array, `pa+1` aponta para o seguinte. Então, em geral, `*(pa+i)` refere-se ao conteúdo do elemento "`i`" do array
- A expressão `pa = &a[0];` pode também ser escrita como `pa = a;`

Aritmética de Ponteiros

- Se `pa` é um ponteiro, então a expressão `pa++;` incrementa `pa` de modo a apontar para o elemento seguinte (seja qual for o tipo de variável para o qual `pa` aponta)
- Do mesmo modo `pa = pa + i;` incrementa `pa` para apontar para `i` elementos à frente do elemento actual
- A tradução das expressões anteriores para Assembly tem que ter em conta o tipo de variável para o qual o ponteiro aponta
- Por exemplo, se um inteiro for definido com 4 bytes (32 bits), então a expressão `pa++;` implica adicionar 4 ao valor actual de `pa`

Acesso sequencial a elementos de um *array*.

O acesso sequencial a elementos de um *array* apoia-se em uma de duas estratégias:

- Endereçamento a partir do nome do *array* e de um índice que identifica o elemento a que se pretende aceder:
  - **f = a[i];**
- Utilização de um ponteiro (endereço armazenado num registo) que identifica em cada instante o endereço do elemento a que se pretende aceder:
  - **f = *pt;** /* com **pt** = endereço de **a[i]** (i.e. pt=&a[i]) */

- **f = a[i];** // Com i ≥ 0

Para aceder ao elemento **i** do array **a**, o programa começa por calcular o respectivo endereço, a partir do endereço inicial do array:

endereço do elemento a aceder = endereço inicial do *array* +
(índice * dimensão em *bytes* de cada posição do *array*)

- **f = *pt;**

O endereço do elemento a aceder está armazenado num registo.

endereço do elemento seguinte = endereço actual +
dimensão em *bytes* de cada posição do *array*

Dois exemplos de acesso sequencial a *arrays*:

```
//Exemplo1

int i, array[size];
for (i = 0; i < size; i++)
{
    array[i] = 0;
}
```

Acesso indexado

```
//Exemplo2

int *p, array[size];
for (p=&array[0]; p < &array[size]; p++)
{
    *p = 0;
}
```

Acesso por ponteiro

```
//Exemplo1
int i, array[size];
for (i = 0; i < size; i++)
{
   array[i] = 0;
}
```

```
$t0 ← i
$t1 ← temp
$t2 ← &(array[0])
$a0 ← size
```

```
        .DATA
array:  .SPACE  size * 4
        .TEXT
(...)
        la      $t2, array      # $t2 = &(array[0]);
        li      $t0, 0          # i = 0;
loop:   bge     $t0, $a0, endf  # while (i < size) {
        mul     $t1, $t0, 4     #   temp = i * 4;
        addu    $t1, $t2, $t1   #   temp = &(array[i])
        sw      $0, 0($t1)      #   array[i] = 0;
        addi    $t0, $t0, 1     #   i = i + 1;
        j       loop            # }
endf:   ...
```



```
//Exemplo2
int *p, array[size];
for (p=&array[0]; p < &array[size]; p++)
{
   *p = 0;
}
```

```
$t0 ← p
$t1 ← &(array[size])
$t7 ← size
```

```
        .DATA
array:  .SPACE  size * 4
        .TEXT
(...)
        la      $t0, array      # $t0 = &(array[0]);
        mul     $t1, $t7, 4     # $t1 = size * 4;
        add     $t1, $t1, $t0   # $t1 = &(array[size]);
loop:   bge     $t0, $t1, endf  # while (p < &array[size]) {
        sw      $0, 0($t0)      #     *p = 0;
        addi    $t0, $t0, 4     #     p = p + 1;
        j       loop            # }
endf:   ...
```

Implementação de uma estrutura do tipo "*switch()*"

Consideremos o seguinte exemplo:

```
switch (k) {
       case 0:  f = i + j; break;
       case 1:  f = g + h; break;
       case 2:  f = g – h; break;
       case 3:  f = i – j; break;
}
```

- Este exemplo pode, evidentemente, ser codificado na forma de uma cadeia de "*if()...then...else*".
- No entanto, quando ocorre que a variável de controlo (neste caso ***k***) varie entre um conjunto de valores contíguos e normalmente separados pela unidade, é possível conceber uma solução de codificação em *Assembly* que resulte num código mais eficiente!


```
switch (k) {
   case 0:
        f = i + j;
        break;
   case 1:
        f = g + h;
        break;
   case 2:
        f = g – h;
        break;
   case 3:
        f = i – j;
        break;
}
```

Codificação do corpo do "switch":

```
CASE0:   add  $s0, $s3, $s4
         j    EXIT

CASE1:   add  $s0, $s1, $s2
         j    EXIT

CASE2:   sub  $s0, $s1, $s2
         j    EXIT

CASE3:   sub  $s0, $s3, $s4
EXIT:    ...
```

Tabela

0x00400054	0
0x0040005C	1
0x00400064	2
0x0040006C	3

Endereço do "case" a executar = Tabela[k]

(c/ k ∈ {0, 1, 2, 3})

```
[0x400054] CASE0:  add   $s0, $s3, $s4
[0x400058]         j     EXIT
[0x40005C] CASE1:  add   $s0, $s1, $s2
[0x400060]         j     EXIT
[0x400064] CASE2:  sub   $s0, $s1, $s2
[0x400068]         j     EXIT
[0x40006C] CASE3:  sub   $s0, $s3, $s4
[0x400070] EXIT:   ...
```

	0x10010010	
	0x1001000F	0x00
	0x1001000E	0x40
	0x1001000D	0x00
	0x1001000C	0x6C
	0x1001000B	0x00
	0x1001000A	0x40
	0x10010009	0x00
	0x10010008	0x64
	0x10010007	0x00
	0x10010006	0x40
	0x10010005	0x00
	0x10010004	0x5C
	0x10010003	0x00
	0x10010002	0x40
	0x10010001	0x00
Tabela	0x10010000	0x54


```
                .DATA
          TABELA: .WORD   CASE0,CASE1,CASE2,CASE3
                .TEXT
                ...
```

variável k reside no registo $s8

```
[........] SWITCH: la     $t2, TABELA     # $t2 = &(TABELA[0]);
[........]         sll    $t0, $s8, 2     # temp = 4 * k
[........]         addu   $t0, $t0, $t2   # $t0=&(TABELA[k]);
[........]         lw     $t1, 0($t0)     # $t1 = TABELA[k];
[........]         jr     $t1             # goto [$t1]

[0x400054] CASE0:  add    $s0, $s3, $s4  # f = i + j;
[0x400058]         j      EXIT           # break;
[0x40005C] CASE1:  add    $s0, $s1, $s2  # f = g + h;
[0x400060]         j      EXIT           # break;
[0x400064] CASE2:  sub    $s0, $s1, $s2  # f = g - h;
[0x400068]         j      EXIT           # break;
[0x40006C] CASE3:  sub    $s0, $s3, $s4  # f = i - j;
[0x400070] EXIT:   ...
```

Que tipo de eficiência se consegue com esta codificação?

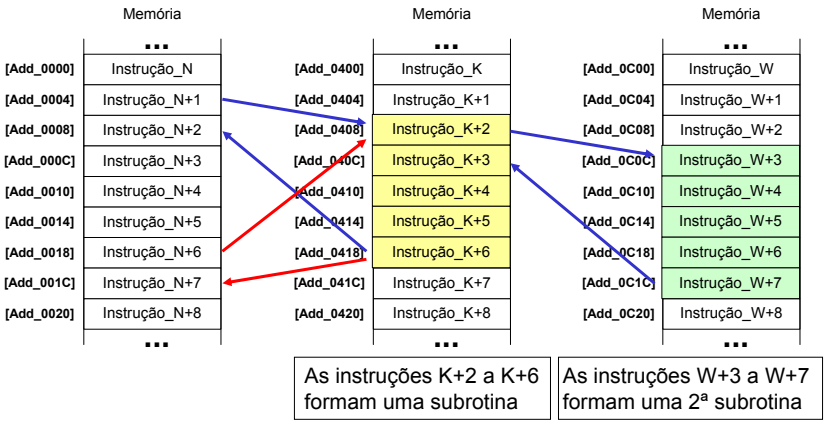
Procedimentos

Há três razões principais que justificam a existência de procedimentos (ou funções)*:

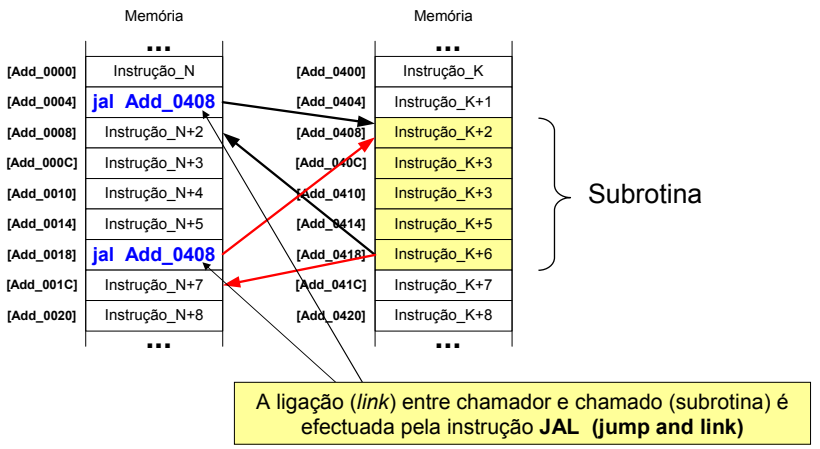
- A reutilização no contexto de um determinado programa - aumento da eficiência na dimensão do código, substituindo a repetição de um mesmo trecho de código por um único trecho evocável de múltiplos pontos do programa;
- A reutilização no contexto de um conjunto de programas, permitindo que o mesmo código possa ser reaproveitado (bibliotecas de funções);
- A organização e estruturação do código

(*) No contexto da linguagem *Assembly*, as funções e os procedimentos são genericamente conhecidas por **subrotinas**!

Subrotinas: Exemplo prático

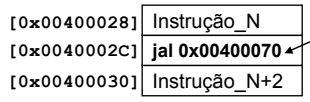


Subrotinas: Exemplo prático (MIPS)



A ligação (*link*) entre chamador e chamado (subrotina) é efectuada pela instrução **JAL (jump and link)**

Ciclo de execução da instrução "jump and link" - **jal *address***



Endereço da 1ª instrução da subrotina a executar

Endereço de retorno

(PC = 0x0040002C)

Fase *fetch*:
- `IR = MEM[PC]` → `IR = jal 0x00400070`
- `PC = PC + 4` → `PC = 0x00400030`

Fase *execute*:
- `$ra = PC` → `$ra = 0x00400030`
- `PC = address` → `PC = 0x00400070`

O registo **$ra** ($31) armazena o endereço de retorno

A próxima instrução a ser executada está no endereço 0x00400070

E como regressar à instrução que sucede à instrução "jal" ?

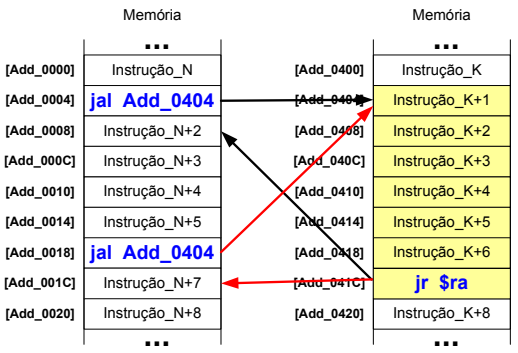
Resposta: aproveitando o endereço de retorno armazenado em **$ra** durante a execução da instrução "**jal**"

(PC = 0x00400094)

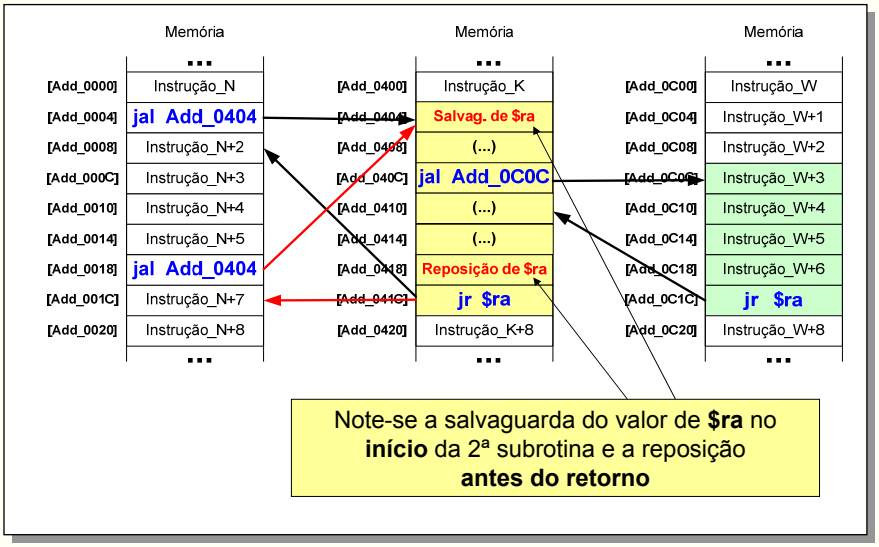
Fase *fetch*:
- IR = MEM[PC] → IR = jr $ra
- PC = PC + 4 → PC = 0x00400098

Fase *execute*:
- PC = $ra → PC = 0x00400030

A próxima instrução a ser executada está no endereço 0x00400030



No caso em que a subrotina chama uma 2ª subrotina, o valor do registo **$ra** é alterado (pela instrução "**jal**"), perdendo-se a ligação para o primeiro chamador. Como resolver este problema?


Memória
[Add_0000] Instrução_N
[Add_0004] jal Add_0404
[Add_0008] Instrução_N+2
[Add_000C] Instrução_N+3
[Add_0010] Instrução_N+4
[Add_0014] Instrução_N+5
[Add_0018] jal Add_0404
[Add_001C] Instrução_N+7
[Add_0020] Instrução_N+8
Memória
[Add_0400] Instrução_K
[Add_0404] Salvag. de $ra
[Add_0408] (...)
[Add_040C] jal Add_0C0C
[Add_0410] (...)
[Add_0414] (...)
[Add_0418] Reposição de $ra
[Add_041C] jr $ra
[Add_0420] Instrução_K+8
Memória
[Add_0C00] Instrução_W
[Add_0C04] Instrução_W+1
[Add_0C08] Instrução_W+2
[Add_0C0C] Instrução_W+3
[Add_0C10] Instrução_W+4
[Add_0C14] Instrução_W+5
[Add_0C18] Instrução_W+6
[Add_0C1C] jr $ra
[Add_0C20] Instrução_W+8
Note-se a salvaguarda do valor de $ra no início da 2ª subrotina e a reposição antes do retorno