Introdução à Arquitetura de Computadores

Aula18

Assembly 3: *Arrays* e Funções

Arrays - Acesso a elementos

- Uso de índices
  - Array de inteiros; Instruções lw/sw
- Código ASCII ; Bytes e caracteres
- Instruções lb, lbu e sb
  - Extensão de *byte* para 32bits
  - Array de bytes

Funções

- Invocação e Retorno (*Caller e Callee*)
  - Instruções jal e jr
- Convenções para uso de registos:
  - Passagem de argumentos ($a0-$a3)
  - Retorno de valor ($v0)

1 - Array (1) - Uso e Caraterísticas

Array (*estrutura de dados*)

- Usada para armazenar grandes quantidades de dados, constituídos por elementos do mesmo tipo (e.g., inteiro, caractere, etc).
- Os elementos do *array* ocupam posições de memória contíguas.

Caraterísticas

- **Tamanho** (*Size:* N): número de elementos
- **Índice** (0..N-1): para aceder a cada elemento*

* Índice = Número de ordem do elemento no *array*

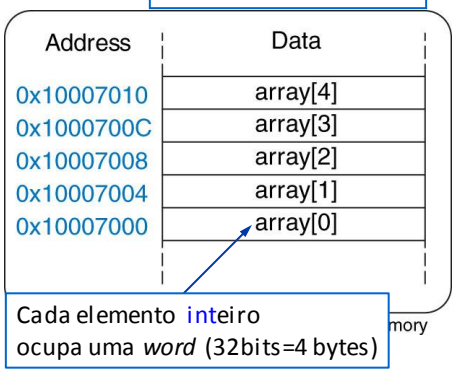
1 - Array (2) - Elementos tipo Inteiro - Exemplo

Array com 5 elementos (tipo inteiro)

```
int array[5];// C Code
```

Size =5; Índice: 0..4

Endereço-Base = 0x10007000
(endereço do primeiro elemento)



Cada elemento inteiro ocupa uma *word* (32bits=4 bytes)

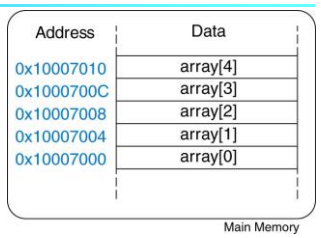
Acesso aos elementos

Primeiro passo:

Carregar o **Endereço-Base** do *array* num registo.

1 - Array (3) - Exemplo em C e Acesso em ASM (Proced.)

```
// C Code
int array[5] ={-2, 4, 5, 123, -324};
      array[0] = array[0] * 8;
      array[1] = array[1] * 8;

# MIPS assembly code
# $s0 = array base address
```



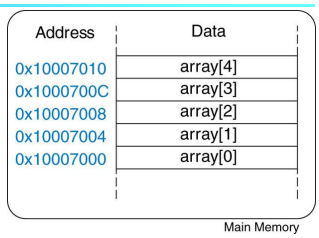
Procedimento de acesso (leitura/escrita)

1. Carregar o endereço do *array* num registo, $s0
2. Usar a instrução **lw** para carregar o valor do elemento array[0] noutro registo $t1; [**lw** $t1, 0($s0)]
3. Multiplicar o valor de $t1 por 8; [**sll** $t1, $t1, 3]
4. Usar a instrução **sw** para armazenar o novo valor de $t1 na mesma posição de memória; [**sw** $t1, 0($s0)]

Repetir os passos de 2 a 4 para o elemento array[1], ajustando o offset de 0 para 4.

1 - Array (4) - Exemplo em ASM

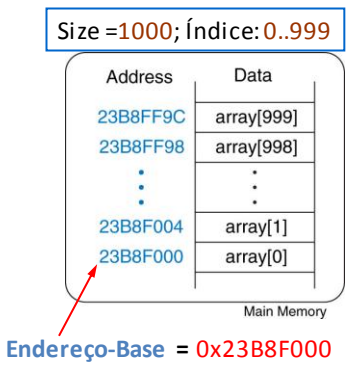
```
// C Code
int array[5] ={-2, 4, 5, 123, -324};
       array[0] = array[0] * 8;
       array[1] = array[1] * 8;

 # MIPS assembly code
 # $s0 = array base address
   lui  $s0, 0x1000          # 0x1000 in upper half of $s0
   ori  $s0, $s0, 0x7000     # 0x7000 in lower half of $s0
 # array[0]
   lw   $t1, 0($s0)          # $t1 = array[0]
   sll  $t1, $t1, 3          # $t1 = $t1<<3 = $t1 * 8
   sw   $t1, 0($s0)          # array[0] = $t1
 # array[1]: byte offset = 4!
   lw   $t1, 4($s0)          # $t1 = array[1]
   sll  $t1, $t1, 3          # $t1 = $t1<<3 = $t1 * 8
   sw   $t1, 4($s0)          # array[1] = $t1
```



1 - Array (5) - Ciclo For em C

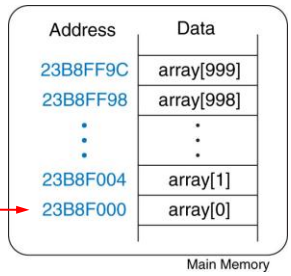
- A codificação anterior é ineficiente para *arrays* longos.
- Preferencial/, usam-se ciclos iterativos *for*, *while*, etc.
- Exemplo: array longo usando um ciclo *for*.

```
// C Code
   int array[1000]; // words
   int i;

   for (i=0; i < 1000; i = i + 1)
       array[i] = array[i] * 8;

# MIPS assembly code
# $s0 = base address, $s1 = i
```



1 - Array (6) - Ciclo For em ASM

```
# $s0 = base address, $s1 = i
# for (i=0; i < 1000; i = i + 1)
# initialization code
  lui  $s0, 0x23B8          # $s0 = 0x23B80000
  ori  $s0, $s0, 0xF000     # $s0 = 0x23B8F000
  addi $s1, $0, 0           # i = 0
  addi $t2, $0, 1000        # $t2 = 1000
loop:  # for loop
  slt  $t0, $s1, $t2        # i < 1000?
  beq  $t0, $0, done        # if not then done
  sll  $t0, $s1, 2          # $t0 = i * 4 (byte offset)
  add  $t0, $t0, $s0        # address of array[i]; $t0 = array + 4*i
  lw   $t1, 0($t0)          # $t1 = array[i]
  sll  $t1, $t1, 3          # $t1 = array[i] * 8
  sw   $t1, 0($t0)          # array[i] = array[i] * 8
# next element
  addi $s1, $s1, 1          # i = i + 1
  j    loop                 # repeat
done:
```




```
int array[1000]; // words
int i;
for (i=0; i < 1000; i = i + 1)
  array[i] = array[i] * 8;
```


2 - Bytes e caracteres (1) - Código ASCII

- *American Standard Code for Information Interchange*
- Cada caractere (de texto) é representado pelo valor (único) de um byte
  - Exemplo, 'S' = 0x53, 'a' = 0x61, 'A' = 0x41
    - A diferença entre as minúsculas ('a') e as maiúsculas ('A') é igual a $0x20_{16}$ (32)

O standard ASCII (1963) veio uniformizar o mapeamento entre bytes e caracteres do alfabeto Inglês, para facilitar a transmissão de texto entre computadores.

2 - Bytes e caracteres (2) - Tabela ASCII

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

*Todos os bytes ASCII têm o bit de sinal igual a zero.

2 - Instruções Load/Store Byte* (1)

Para carregar, da memória, um registo de 32-bits com um byte, existem duas instruções: lbu e lb.

lbu - **load byte unsigned**

- faz a extensão (do byte) para 32-bits com zeros

lb - **load byte**

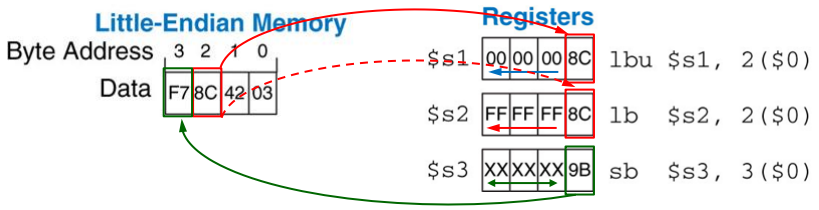
- faz a extensão para 32-bits com o bit de sinal do byte.

sb - **store byte**

- armazena o byte menos significativo do registo (LSB) no endereço (de byte) da memória e ignora os restantes bytes do registo.

*Também existem instruções para aceder a *half-words* (16-bits): lhu, lh e sth .

2 - Instruções Load/Store Byte (2)



lbu $s1, 2 ($0) - carrega o byte 0x8C do endereço 2 no LSB do registo $s1 e preenche com zeros os restantes 3 bytes.

lb $s2, 2 ($0) - carrega o byte 0x8C do endereço 2 no LSB do registo $s2 e preenche com 1's (sinal) os restantes 3 bytes.

sb $s3, 3 ($0) - armazena o byte (LSB) 0x9B do registo $s3 no endereço 3 da memória, substituindo o byte 0xF7, que lá se encontrava, e ignorando os restantes bytes (XX).

*Não é a memória que é *Little-Endian*, mas sim o CPU ☹...

2 - Exemplo lb/sb - Array de Bytes (1)

O seguinte código C converte um *array* com 10 caracteres de minúsculas para maiúsculas, subtraindo 32 (0x20) a cada elemento.

```
char chararr[10]; // bytes
int i;
for (i=0; i != 10; i = i + 1)
  chararr[i] = chararr[i] - 32;
```

Exemplo (ver tabela ASCII):
'a' (0x61) --> 'A' (0x41)
'A' = 'a' - 0x20 = 'a' - 32

Traduzir para Assembly.

Ter em consideração que a diferença entre endereços de memória de dois elementos consecutivos do *array* é agora de um só byte e não de 4 bytes (caso do *array* de inteiros).

Supor que o registo $s0 já está inicializado com o endereço do *array* **chararr**.

2 - Exemplo lb/sb - Array de Bytes (2)

```
char chararr[10]; // bytes (com bit de sinal = 0)
int i;
for (i=0; i != 10; i = i + 1)
  chararr[i] = chararr[i] - 32;
```

```
# $s0 = array base address = chararr = &chararr[0]
# $s1 = i
  addi $s1, $0, 0          # i = 0
  addi $t0, $0, 10         # $t0 = 10
# for loop
loop:
  beq  $s1, $t0, done      # if (i==10) done
  # $t1 = chararr + i = &chararr[i]
  # não é necesário multiplicar o valor de i ($s1), porquê?
  add  $t1, $s1, $s0       # $t1 = address of chararr[i]
  lb   $t2, 0($t1)         # $t2 = chararr[i]
  addi $t2, $t2, -32       # conv_to_upcase: $t2 = $t2 - 32
  sb   $t2, 0($t1)         # chararr[i] = chararr[i]-32
  addi $s1, $s1, 1         # i = i + 1
  j    loop                # repeat
done:
```

Note-se a utilização das instruções lb e sb em vez de lw e sw.

3 - Funções (1) - Introdução

Definição

- As Linguagens de Alto-Nível usam *funções* (*procedures* ou *subrotinas*) para estruturar um programa em módulos reutilizáveis e ainda para aumentar a clareza do código.

Argumentos e Retorno

- As funções possuem entradas, os **argumentos**, e uma saída, o valor de **retorno**.

Caller e Callee

- Quando uma função (*caller*) invoca outra (*callee*) é necessária uma convenção (conjunto de regras) para a passagem dos argumentos e para a recolha do valor retorno devolvido pela função.

3 - Funções (2) - Caller e Callee através de Exemplo

A função main invoca a função sum para calcular a soma de **a** + **b**.

main (*caller*) passa os argumentos **a** e **b** e recebe o resultado devolvido pela função sum (*callee*).

```
int sum(int a, int b);

int main(){
  int y;
    y = sum(42, 7);
    ...// y = 49
  return 0;
}

int sum(int a, int b){
    return (a + b);
}
```

- ***Caller***: função *Invocadora* (main)
- ***Callee***: função *Invocada* (sum)

3 - Funções (3) - Procedimento de Invocação e Retorno

- ***Caller*** (Invocadora):
  - Passa os **argumentos** à *Callee*
  - 'Salta' para o código da *Callee*
- ***Callee*** (Invocada) :
  - Usa os argumentos para **Executar** o código da função
  - **Devolve o** resultado à *Caller*
  - **Regressa** ao código donde foi chamada
  - **Não deve alterar** registos ou memória necessários à *Caller*.

3 - Funções (4) - Instruções e Convenções MIPS

- **Instruções**
  - **Invocar** uma função: jump and link (**jal**)
    (*Caller:* executa jal <*Callee*>)
  - **Retornar** duma função: jump register (**jr**)
    (*Callee:* executa jr $ra)
- **Convenções**
  - **Argumentos**: $a0 - $a3
    (*Caller passa* $a0..$a3 à *Callee*)
  - **Valor de Retorno**: $v0
    (*Callee* devolve $v0 à *Caller*)

3 - Funções (5) - Instruções jal e jr

```
int main() {
  simple();
  a = b + c;
return 0;
}

void simple() {
  return;
}
```

```
0x00400200 main:  jal  simple
0x00400204        add  $s0, $s1, $s2
...
0x00401020 simple:jr $ra
```

jal simple : 'salta' para **simple** e 'liga'
$ra = PC + 4 = 0x00400204

jr $ra : 'salta' para o endereço contido em **$ra** (0x00400204)

(i.e., regressa ao ponto após a jal)

$ra - return address

void - significa que a função 'simple' não devolve qualquer valor.

3 - Funções (6) - Argumentos e Retorno - Código C

A função **diffofsums** é invocada com quatro argumentos e devolve o resultado em $v0.

A função *caller* coloca os argumentos nos registos $a0-$a3.
A função *callee* devolve o resultado no registo $v0 .

```
int main(){
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i){
  int result;
  result = (f + g) - (h + i);
  return result;                // return value
}
```


3 - Funções (7) - Argumentos e Retorno - Código ASM

main coloca os argumentos nos registos $a0-$a3.
diffofsums devolve o resultado no registo $v0 .

```
# $s0 = y
main:
  ...
  addi $a0, $0, 2     # arg0 (f) = 2
  addi $a1, $0, 3     # arg1 (g) = 3
  addi $a2, $0, 4     # arg2 (h) = 4
  addi $a3, $0, 5     # arg3 (i) = 5
→ jal  diffofsums     # call Function
  add  $s0, $v0, $0   # y = returned value
  ...

# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
→ jr  $ra             # return to caller
```

3 - Funções (8) - Argumentos e Retorno - Código ASM_2

main coloca os argumentos nos registos $a0-$a3.
diffofsums devolve o resultado no registo $v0.

```
# $s0 = y
main:
  ...
  addi $a0, $0, 2     # arg0 (f) = 2
  addi $a1, $0, 3     # arg1 (g) = 3
  addi $a2, $0, 4     # arg2 (h) = 4
  addi $a3, $0, 5     # arg3 (i) = 5
  jal  diffofsums     # call Function
  add  $s0, $v0, $0   # y = returned value
  ...

# $s0 = result; isto não é necessário!
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $v0, $t0, $t1   # $v0 = (f + g) - (h + i)
  #add $v0, $s0, $0   # put return value in $v0
  jr  $ra             # return to caller
```

O código de *diffofsums* podia ser simplificado, mas esse não o ponto, por agora ☺.

3 - Funções (9) - Salvaguarda de Registos - O Problema

```
# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

diffofsums

Alterou 3 registos: $t0, $t1 e $s0

P: E se a função main também usar esses registos?

R: Podem (main e ou `diffofsums`) usar o ***stack*** (?) para salvaguardar temporária/ o conteúdo dos registos.