# Lab 6: Arrays in MIPS Assembly Language

**Objectives:**

After completing this lab, you will:

- Define and initialize arrays statically in the data segment
- Allocate memory dynamically on the heap
- Compute the memory addresses of array elements
- Write loops in MIPS assembly to traverse arrays

## 1. Defining and Initializing Arrays Statically in the Data Segment

Unlike high-level programming languages, assembly language has no special notion for an array. An array is just a block of memory. In fact, all data structures and objects that exist in a high-level programming language are simply blocks of memory. The block of memory can be allocated statically or dynamically, as will be explained shortly.

An array is a homogeneous data structure. It has the following properties:

1. All array elements must be of the same type and size.
2. Once an array is allocated, its size becomes fixed and cannot be changed.
3. The base address of an array is the address of the first element in the array.
4. The address of an element can be computed from the base address and the element index.

An array can be allocated and initialized statically in the data segment. This requires:

1. A **`label:`** for the array name.
2. A **`.type`** directive for the type and size of each array element.
3. A list of initial values, or a count of the number of elements

A data definition statement allocates memory in the data segment. It has the following syntax:

```
label: .type value [, value] . . .
```

Examples of data definition statements are shown below:

```
.data
arr1: .half 5,-1  # array of 2 half words initialized to 5, -1
arr2: .word 1:10   # array of 10 words, all initialized to 1
arr3: .space 20    # array of 20 bytes, uninitialized
str1: .ascii "This is a string"
str2: .asciiz "Null-terminated string"
```

In the above example, **`arr1`** is an array of 2 half words, as indicated by the **`.half`** directive,
initialized to the values **`5`** and **`-1`**. **`arr2`** is an array of 10 words, as indicated by the **`.word`** directive, all initialized to **`1`**. The **`1:10`** notation indicates that the value **`1`** is repeated **`10`** times. **`arr3`** is an array of **`20`** bytes. The **`.space`** directive allocates bytes without initializing them in memory. The **`.ascii`** directive allocates memory

for a string, which is an array of bytes. The **`.asciiz`** directive does the same thing, but adds a NULL byte at the end of the string. In addition to the above, the **`.byte`** directive is used to define bytes, the **`.float`** and **`.double`** directives are used to define floating-point numbers.

Every program has three segments when it is loaded into memory by the operating system, as shown in Figure 1.1. There is the **text segment** where the machine language instructions are stored, the **data segment** where constants, variables and arrays are stored, and the **stack segment** that provides an area that can be allocated and freed by functions. Every segment starts at a specific address in memory. The data segment is divided into a **static area** and a **dynamic area.** The dynamic area of the data segment is called the **heap**. Data definition statements allocate space for variables and arrays in the static area.

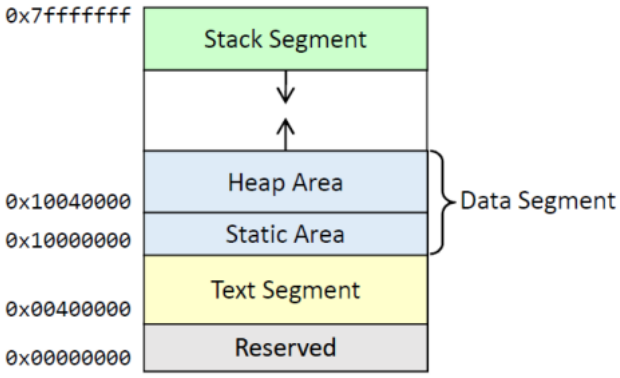



Figure 1.1 The text, data, and stack segments of a program

If arrays are allocated in the static area, then one might ask: what is the address of each array? To answer this question, the assembler constructs a **symbol table** that maps each **`label`** to a fixed address. To see this table in the MARS simulator, select "Show Labels Window (symbol table)" from the Settings menu. This will display the Labels window as shown in Figure 1.2. From this figure, one can obtain the address of **`arr1`** (**`0x10010000`**), of **`arr2`** (**`0x10010004`**), of **`arr3`** (**`0x1001002c`**), etc.

The **`la`** pseudo-instruction loads the address of a label into a register. For example, **`la $t0, arr3`** loads the address of **`arr3`** (**`0x1001002c`**) into register **`$t0`**. This is essential because the programmer needs the address of an array to process its elements in memory.

You can watch the values in the data segment window as shown in Figure 1.3. To watch ASCII characters, click on the ASCII box in the data segment window as shown in Figure 1.4. Notice that characters appear in reverse order within a word in Figure 1.4. If the **`lw`** instruction is used to load four ASCII characters into a register, then the first character is loaded into the least significant byte of a register. This is known as **little-endian** byte ordering. On the other hand, **big-endian** orders the bytes within a word from the most-significant to the least-significant byte.

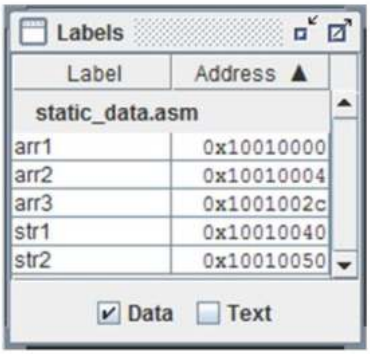



Figure 1.2: Labels (symbol table) window under MARS

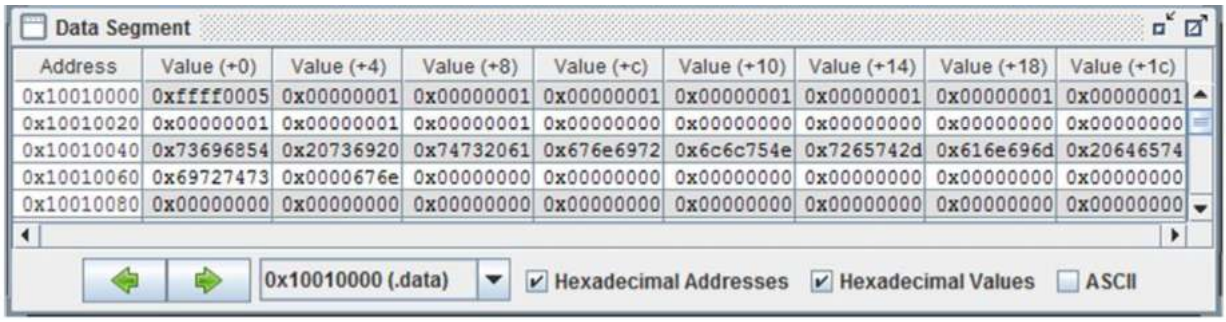



Figure 1.3: Watching Values in hexadecimal in the Data Segment

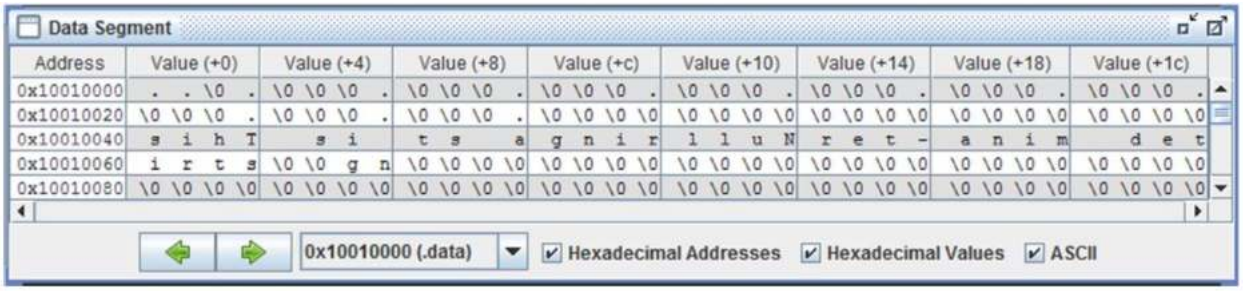



Figure 1.4: Watching ASCII Characters in the Data Segment

## 2. Allocating Memory Dynamically on the Heap

Defining data in the static area of the data segment might not be convenient. Sometimes, a program wants to allocate memory dynamically at runtime. One of the functions of the operating system is to manage memory. During runtime, a program can make requests to the operating system to allocate additional memory dynamically on the heap. The heap area is a part of the data segment (Figure 1.1), that can grow dynamically at runtime. The program makes a system call (**`$v0 = 9`**) to allocate memory on the heap, where **`$a0`** is the number of bytes to allocate. The system call returns the address of the allocated memory in **`$v0`**.

**Exercise1:** The following program allocates two blocks on the heap:

Edit | Execute

exe2_lab6.asm*

```
1  .text
2  . . .
3  li $a0, 100            # $a0 = number of bytes to allocate
4  li $v0, 9              # system call 9
5  syscall               # allocate 100 bytes on the heap
6
7  move $t0, $v0          # $t0 = address of first block
8  li $s0, 50
9  sw $s0, 0($t0)
10
11 li $a0, 200         # $a0 = number of bytes to allocate
12 li $v0, 9           # system call 9
13 syscall             # allocate 200 bytes on the heap
14 move $t1, $v0       # $t1 = address of second block
15 li $s1, 30
16 sw $s1, 0($t1)
17
18 li $v0, 10
19  syscall
20
```

Edit | Execute

Text Segment

| Bkpt | Address | Code | Basic | Source |
|---|---|---|---|---|
| | 4194304 | 0x24040064 | addiu $4,$0,100 | 3: li $a0, 100 # $a0 = number of bytes to allocate |
| | 4194308 | 0x24020009 | addiu $2,$0,9 | 4: li $v0, 9 # system call 9 |
| | 4194312 | 0x0000000c | syscall | 5: syscall # allocate 100 bytes on the heap |
| | 4194316 | 0x00024021 | addu $8,$0,$2 | 7: move $t0, $v0 # $t0 = address of first block |
| | 4194320 | 0x24100032 | addiu $16,$0,50 | 8: li $s0, 50 |
| | 4194324 | 0xad100000 | sw $16,0($8) | 9: sw $s0, 0($t0) |
| | 4194328 | 0x240400c8 | addiu $4,$0,200 | 11: li $a0, 200 # $a0 = number of bytes to allocate |
| | 4194332 | 0x24020009 | addiu $2,$0,9 | 12: li $v0, 9 # system call 9 |
| | 4194336 | 0x0000000c | syscall | 13: syscall # allocate 200 bytes on the heap |
| | 4194340 | 0x00024821 | addu $9,$0,$2 | 14: move $t1, $v0 # $t1 = address of second block |
| | 4194344 | 0x2411001e | addiu $17,$0,30 | 15: li $s1, 30 |
| | 4194348 | 0xad310000 | sw $17,0($9) | 16: sw $s1, 0($t1) |
| | 4194352 | 0x2402000a | addiu $2,$0,10 | 18: li $v0, 10 |
| | 4194356 | 0x0000000c | syscall | 19: syscall |

Data Segment

| Address | Value (+0) | Value (+4) | Value (+8) | Value (+12) | Value (+16 |
|---|---|---|---|---|---|
| 268697600 | 50 | 0 | 0 | 0 | |
| 268697632 | 0 | 0 | 0 | 0 | |
| 268697664 | 0 | 0 | 0 | 0 | |
| 268697696 | 0 | 30 | 0 | 0 | |
| 268697728 | 0 | 0 | 0 | 0 | |
| 268697760 | 0 | 0 | 0 | 0 | |
| 268697792 | 0 | 0 | 0 | 0 | |
| 268697824 | 0 | 0 | 0 | 0 | |
| 268697856 | 0 | 0 | 0 | 0 | |
| 268697888 | 0 | 0 | 0 | 0 | |
| 268697920 | 0 | 0 | 0 | 0 | |
| 268697952 | 0 | 0 | 0 | 0 | |
| 268697984 | 0 | 0 | 0 | 0 | |
| 268698016 | 0 | 0 | 0 | 0 | |

x10040000 (heap) | Hexadecimal Addresses | Hexad

Registers | Coproc 1 | Coproc 0

| Name | Number | Value |
|---|---|---|
| $zero | 0 | 0 |
| $at | 1 | 0 |
| $v0 | 2 | 10 |
| $v1 | 3 | 0 |
| $a0 | 4 | 200 |
| $a1 | 5 | 0 |
| $a2 | 6 | 0 |
| $a3 | 7 | 0 |
| $t0 | 8 | 268697600 |
| $t1 | 9 | 268697700 |
| $t2 | 10 | 0 |
| $t3 | 11 | 0 |
| $t4 | 12 | 0 |
| $t5 | 13 | 0 |
| $t6 | 14 | 0 |
| $t7 | 15 | 0 |
| $s0 | 16 | 50 |
| $s1 | 17 | 30 |
| $s2 | 18 | 0 |
| $s3 | 19 | 0 |
| $s4 | 20 | 0 |
| $s5 | 21 | 0 |
| $s6 | 22 | 0 |
| $s7 | 23 | 0 |
| $t8 | 24 | 0 |
| $t9 | 25 | 0 |
| $k0 | 26 | 0 |
| $k1 | 27 | 0 |
| $gp | 28 | 268468224 |
| $sp | 29 | 2147479548 |
| $fp | 30 | 0 |
| $ra | 31 | 0 |
| pc | | 4194360 |
| hi | | 0 |
| lo | | 0 |

### 3. Computing the Addresses of Array Elements

In a high-level programming language, arrays are indexed. Typically, **`array[0]`** is the first element in the array, and **`array[i]`** is the element at index **`i`**. Because all

array elements have the same size, then **address** of **array[i]**, denoted as **&array[i] = &array + i × element_size**.

In the above example, **arr2** is defined as an array of words (**.word** directive). Since each word is **4** bytes, then **&arr2[i] = &arr2 + i×4**. The **&** is the address operator. Since the address of **arr2** is given as **0x10010004** in Figure 1.2, then: **&arr2[i] = 0x10010004 + i×4**.

A two-dimensional array is stored linearly in memory, similar to a one-dimensional array. To define **matrix[Rows][Cols]**, one must allocate **Rows × Cols** elements in memory.

In most programming languages, a two-dimensional array is stored row-wise in memory: row 0, row 1, row 2, … etc. This is known as **row-major order**. Then, **address of matrix[i][j]**, denoted as **&matrix[i][j]** becomes:
**& matrix[i][j] = &matrix + (i×Cols + j) × element_size**

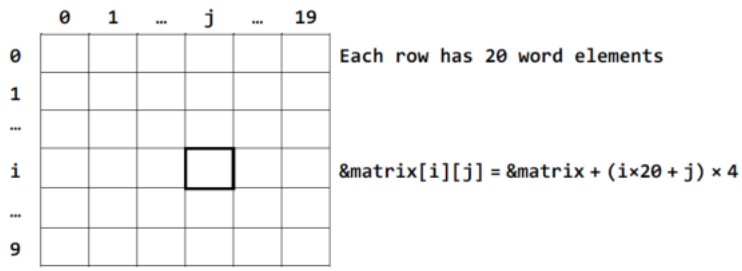
If the number of columns is **20** and the element size is **4** bytes (**.word**), then:
**&matrix[i][j] = &matrix + (i×20 + j) × 4**

For example, one can define a matrix of 10 rows × 20 columns word elements, all initialized to zero, as follows:
```
matrix: .word 0:200 # 10 by 20 word elements initialized to 0
```

To translate **matrix[1][5] = 73** into MIPS assembly language, one must compute:

**&matrix[1][5] = &matrix + (1×20+5)×4 = &matrix + 100**.

```
la $t0, matrix       # load address: $t0 = &matrix
li $t1, 73           # $t1 = 73
sw $t1, 100($t0)     # matrix[1][5] = 73
```




## 4. Writing Loops to Traverse Arrays

## Exercise2:

The following **while** loop searches an array of **n** integers linearly for a given **target** value:

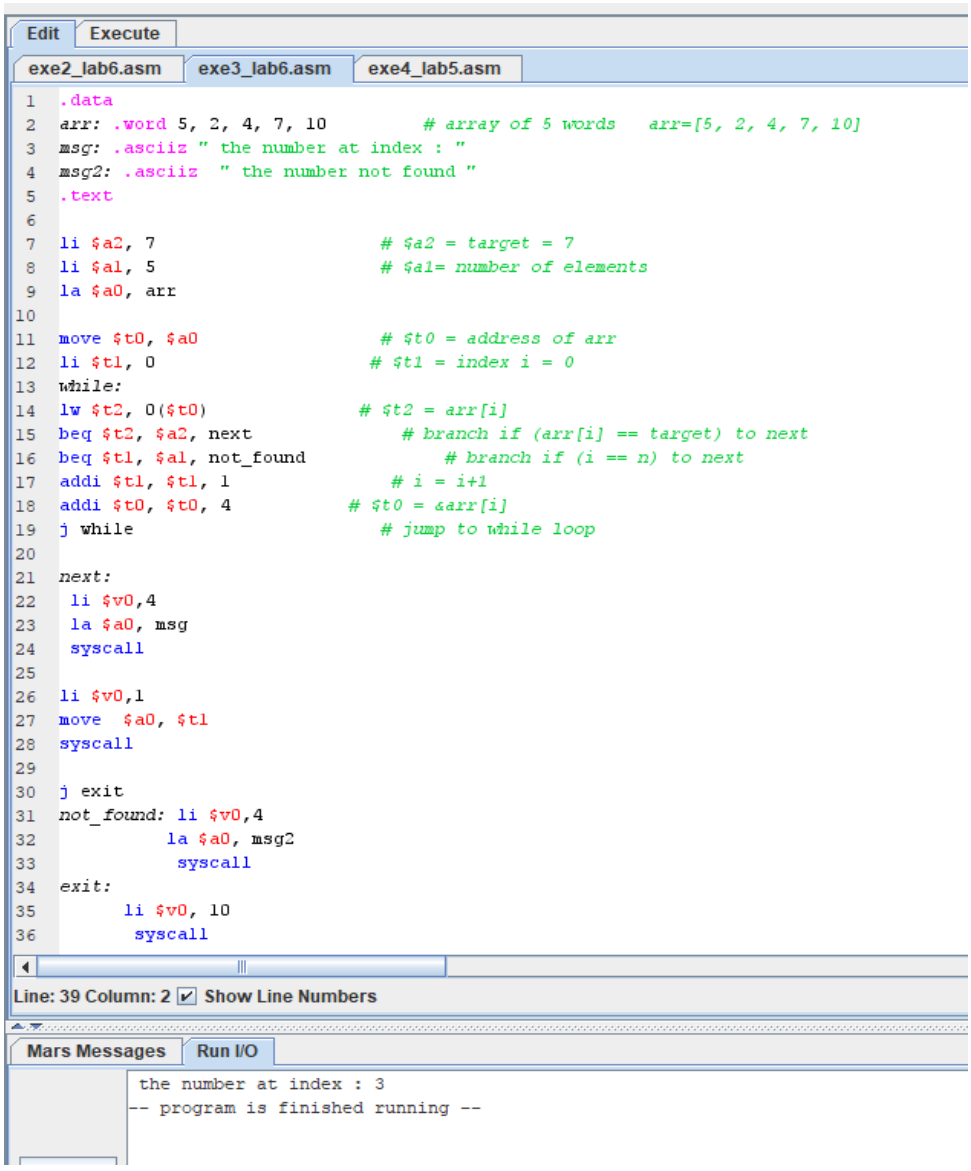
**int i=0;**
**while (arr[i] != target && i<n) i = i+1;**

Given that **$a0 = &arr** (address of **arr**), **$a1 = n**, and **$a2 = target**, the above loop is translated into MIPS assembly code as follows:

```
move $t0, $a0                # $t0 = address of arr
li $t1, 0                    # $t1 = index i = 0
while:
lw $t2, 0($t0)               # $t2 = arr[i]
beq $t2, $a2, next           # branch if (arr[i] == target) to next
beq $t1, $a1, next           # branch if (i == n) to next
addi $t1, $t1, 1             # i = i+1
sll $t3, $t1, 2              # $t3 = i×4
add $t0, $a0, $t3            # $t0 = &arr + i×4 = &arr[i]
j while                      # jump to while loop
next:
. . .
```

To calculate the address of **arr[i]**, the **sll** instruction shifts left **i** by **2** bits (computes **i×4**) and then the **add** instruction computes **&arr[i] = &arr + i×4**. However, one can also point to the next array element by incrementing the address in **$t0** by **4**, as shown below. Using a pointer to traverse an array sequentially is generally faster than computing the address from the index.

```
while:
lw $t2, 0($t0)               # $t2 = arr[i]
beq $t2, $a2, next           # branch if (arr[i] == target) to next
beq $t1, $a1, next           # branch if (i == n) to next
addi $t1, $t1, 1             # i = i+1
addi $t0, $t0, 4             # $t0 = &arr[i]
j while                      # jump to while loop
next:
. . .
```




## Exercise3:

Write a MIPS program that defines a static byte array, then counts the frequency of a specific number k in this array. Then displays it. Suppose Array= [4, 2, 4, 5, 9, 4, 4, 6, 4, 7].

**For( i=1; i<n; i++) {**

**IF (array[i] = k) { Count++; } }**

Edit Execute

count_element_lab6.asm*

```
1   .data
2   array:  .byte  4, 2, 4, 5, 9, 4, 4, 6, 4, 7
3   msg1: .asciiz " The frequency of 4 is :  "
4
5   .text
6
7   la $a0, array
8   li $t0, 10     # number of elements
9   li $s0, 0      # counter =0
10  li $t1, 0      # i=0
11  li $s1, 4      # k=4 ... specific number
12
13  Loop: lb $t2, 0($a0)
14        beq  $t1, $t0, Exit
15        beq $t2, $s1, Increase
16        j  Continue
17  Increase:
18        addi $s0, $s0, 1
19  Continue:
20        addi $t1, $t1, 1
21        addi $a0,$a0, 1
22        j Loop
23   Exit:
24       li $v0, 4
25       la $a0, msg1
26       syscall
27
28       move $a0, $s0
29       li $v0, 1
30      syscall
31
32      li $v0, 10
33      syscall
```

Line: 35 Column: 6 Show Line Numbers

Mars Messages Run I/O

```
The frequency of 4 is :  5
-- program is finished running --
```