LAB #05 Array

Defining Arrays in 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, but we will focus on statical memory allocation only.

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 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 5.1. 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.

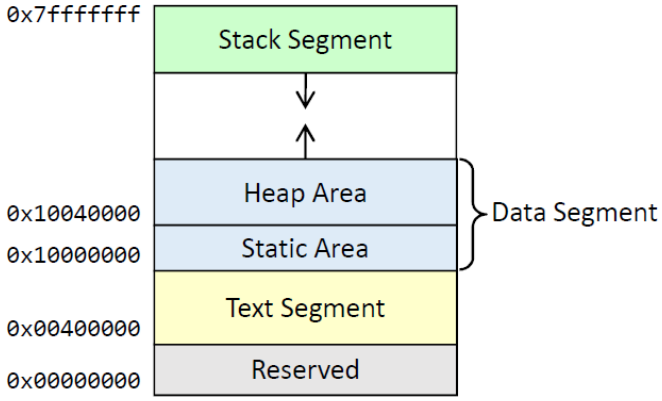

Fig. 1 The text, data, and stack segments of a program

How do we know the address of each array?

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 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.

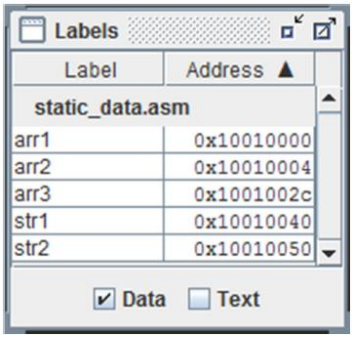

Fig. 2 Labels (symbol table) window in MARS

You can watch the values in the data segment window as shown in Figure 3. To watch ASCII characters, click on the ASCII box in the data segment window.
Notice that characters appear in reverse order. 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.
FYI: Classic MIPS was big-endian. Later MIPS support selectable endianness.

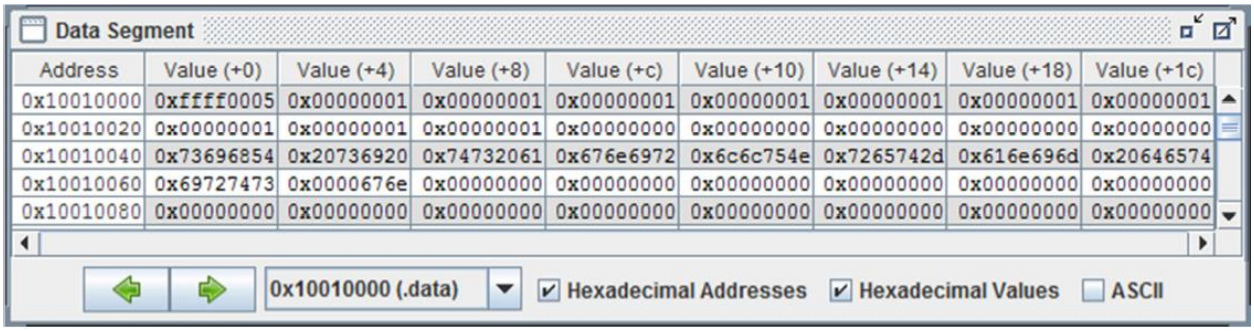

Fig. 3 Data segment window in MARS

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 Fig 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. 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
```

In most programming languages, a two-dimensional array is stored row-wise in memory, e.g., 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, to translate `matrix[1][5] = 73` into MIPS assembly language, one must compute:

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

Accordingly,

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

Task to do

1. Given the following data definition statements, compute the addresses of arr2, arr3, str1, and str2, given that the address of arr1 is 0x10010000. (Show your steps of calculation.) Select “Show Labels Window (symbol table)” from the Settings menu in MARS to check the values of your computed addresses.

```
.data
arr1: .word 5:20
arr2: .half 7, -2, 8, -6
arr3: .space 100
str1: .asciiz "This is a message"
str2: .asciiz "Another important string"
```

2. In problem 1, given that arr1 is a one-dimensional array of integers, what are the addresses of `arr1[5]` and `arr1[17]`? (Show your steps of calculation and check it in MARS.)

3. In problem 1, given that arr3 is a two-dimensional array of bytes with 20 rows and 5 columns, what are the addresses of `arr3[7][2]`, `arr3[11][4]`, and `arr3[19][3]`? (Show your steps of calculation and check it in MARS.)

NOTE: Write a report according to the LAB questions and submit a word (or pdf) file.