

## 2.1 Lab 8: Basic Data Structures

### Objectives 2

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 RISC-V assembly to traverse arrays
- Write system calls to open, read from, and write to files

### 2.1.1 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: .asciz "Null-terminated string"
```

Figure 2.1: Data Definition

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 `.asciz` 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 2.2. 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.

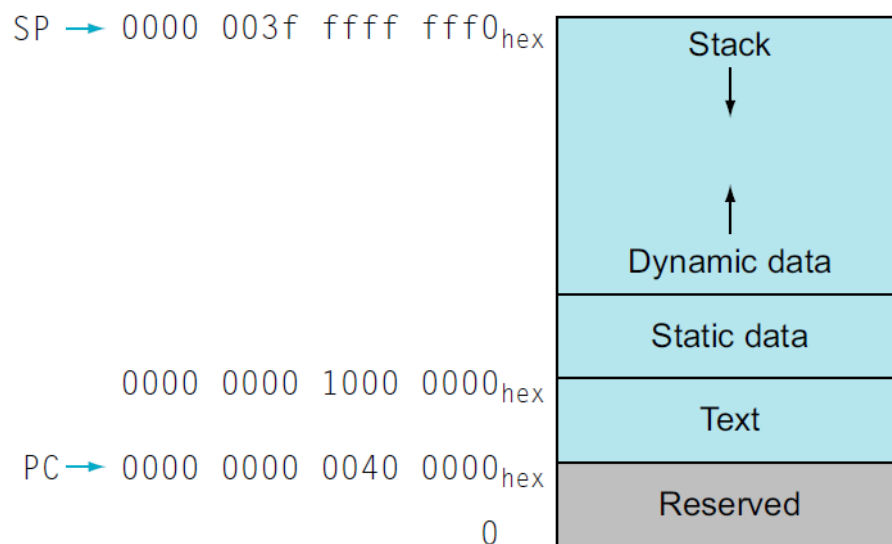


Figure 2.2: Memory Allocation

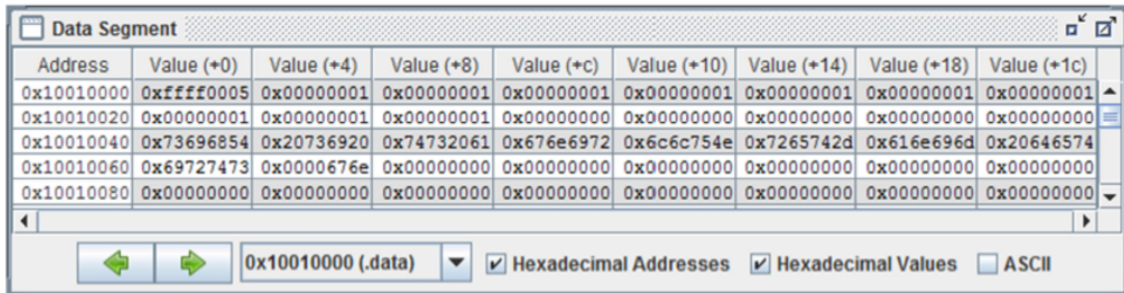
If arrays are allocated in the static area, then one might ask: what is the address of each array? To answer this question, the assembler constructs a symbol table that maps each label to a fixed address. To see this table in the RARS simulator, select “Show Labels Window (symbol table)” from the Settings menu. This will display the Labels window as shown in Figure 2.3. From this figure, one can obtain the address of **arr1** (0x10010000), of **arr2** (0x10010004), of **arr3** (0x1001002c), etc.

Labels	
Label	Address ▲
static_data.asm	
arr1	0x10010000
arr2	0x10010004
arr3	0x1001002c
str1	0x10010040
str2	0x10010050
<input checked="" type="checkbox"/> Data <input type="checkbox"/> Text	

Figure 2.3: Symbol Table

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.

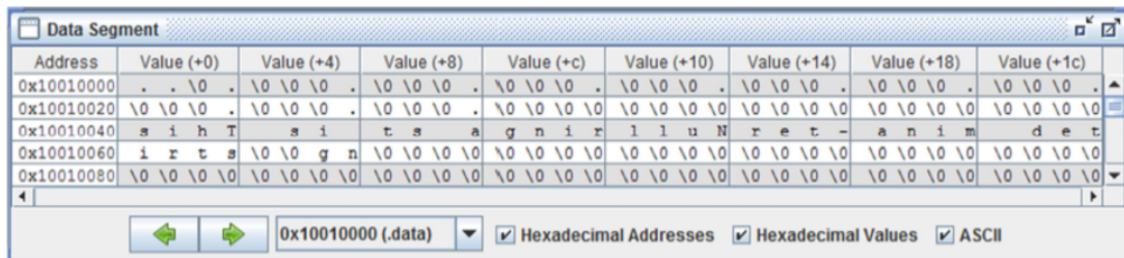
You can see the values in the data segment window as shown in Figure 2.4. To see ASCII characters, click on the ASCII box in the data segment window as shown in Figure 2.5.



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0xffff0005	0x00000001	0x00000001	0x00000001	0x00000001	0x00000001	0x00000001	0x00000001
0x10010020	0x00000001	0x00000001	0x00000001	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x73696854	0x20736920	0x74732061	0x676e6972	0x6c6c754e	0x7265742d	0x616e696d	0x20646574
0x10010060	0x69727473	0x0000676e	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) [X] Hexadecimal Addresses [X] Hexadecimal Values [ ] ASCII

Figure 2.4: Data Segment



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	. . \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .
0x10010020	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010040	s i h T	s i	t s a	g n i r	l l u N	r e t -	a n i m	d e t
0x10010060	i r t s	\0 \0 g n	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

0x10010000 (.data) [X] Hexadecimal Addresses [X] Hexadecimal Values [X] ASCII

Figure 2.5: Data Segment ASCII

## 2.1.2 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 2.2), that can grow dynamically at runtime. The program makes a system call `sbrk`, by loading 9 into `a7` (**li a7, 9**), to allocate memory on the heap, where `a0` should contain the number of bytes to allocate. The system call returns the address of the allocated memory in `a0`. The following program allocates a block on the heap:

```
.macro sbrk(%bytes)
    li a7, 9
    li a0, %bytes
    ecall
.end_macro

sbrk(10)
```

Figure 2.6: Dynamic Data Definition

Note that the **.macro** keyword is used to create a subroutine. We could allocate dynamic memory with the **li** instruction and the `ecall`.

### 2.1.3 Computing the Addresses of Array Elements

All array elements have the same size, then address of array[i], denoted as:

$$\text{array}[i] = \&\text{array} + i \times \text{Element-Size}.$$

In the above example, arr2 is defined as an array of words (.word directive). Since each word is 4 bytes, then:

$$\&\text{arr2}[i] = \&\text{arr2} + i \times 4.$$

The & is the address operator. Since the address of arr2 is given as 0x10010004 in Figure 2.3, then:

$$\&\text{arr2}[i] = 0\text{x}10010004 + i \times 4.$$

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: row<sub>0</sub>, row<sub>1</sub>, row<sub>2</sub>, ... etc. This is known as row-major order. Then, address of matrix[i][j], denoted as &matrix[i][j] becomes:

$$\&\text{matrix}[i][j] = \&\text{matrix} + (i \times \text{Cols} + j) \times \text{ElementSize}$$

If the number of columns is 20 and the element size is 4 bytes (.word), then:

$$\&\text{matrix}[i][j] = \&\text{matrix} + (i \times 20 + j) \times 4.$$

For example, to translate matrix[1][5] = 73 into RISC-V assembly language, one must compute:

$$\&\text{matrix}[1][5] = \&\text{matrix} + (1 \times 20 + 5) \times 4 = \&\text{matrix} + 100.$$

```
.data
matrix: .word 0:200 # 10 by 20 word elements initialized to 0
.text
la t0, matrix      # load address: t0 = &matrix
li t1, 73           # t1 = 73
sw t1, 100(t0)      # matrix[1][5] = 73
```

Figure 2.7: Matrix Definition

Unlike a high-level programming language, address calculation is essential when programming in assembly language. One must calculate the addresses of array elements precisely when processing arrays in assembly language.

### 2.1.4 Writing Loops to Traverse Arrays

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 RISC-V assembly code as follows:

```
.data
arr: .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
.text
la a0, arr
li a1, 10
li a2, 5
mv 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
  slli t3, t1, 2    # t3 = i*4
  add t0, a0, t3    # t0 = &arr + i*4 = &arr[i]
  jal x0, while     # jump to while loop
next:
```

Figure 2.8: Array Traversal

To calculate the address of  $arr[i]$ , the `slli` instruction shifts left  $i$  by 2 bits (computes  $i \times 4$ ) and then the `add` instruction computes  $\&arr[i] = \&arr + i \times 4$ . 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.

```

.data
arr: .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
.text
la a0, arr
li a1, 10
li a2, 5
mv 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
addi t0, t0, 4     # t0 = &arr[i]
j while           # jump to while loop
next:

```

Figure 2.9: Array Traversal with +4

A second example of a 2-dimensional array is shown below:

```

int M[10][5];
int i;
for (i=0; i<10; i++) {
    M[i][3] = i;
}

```

```

.data
M: .word 0:50          # array of 50 words
# &M[i][3] = &M + (i×5+3)×4 = &M + i×20 + 12
.text
la t0, M               # t0 = &M[0][0]
li t1, 0               # t1 = i = 0
li t2, 10              # t2 = 10
li t3, 20              # t3 = 20
for:
mul t5, t3, t1         # t5 = i×20
add t5, t0, t5         # t5 = &M + i×20
sw t1, 12(t5)          # store: M[i][3] = i
addi t1, t1, 1         # i++
bne t1, t2, for        # branch if (i != 10)

```

Figure 2.10: 2D Array Traversal

### 2.1.5 Files

The operating system manages files on the disk storage. It provides system calls to open, read from, and write to files. The RARS tool simulates some of the services of the operating system and provides the following system calls:

Service	a7	Arguments	Result
Open file	1024	a0 = address of null-terminated string containing the file name. a1 = 0 if read only a1 = 1 if write-only with create a1 = 9 if write-only with create and append	a0 = file descriptor a0 = -1 if error
Read from file	63	a0 = file descriptor a1 = address of input buffer a2 = maximum number of characters to read	a0 = number of characters read a0 = 0 if end-of-file a0 = -1 if error
Write to file	64	a0 = file descriptor a1 = address of output buffer a2 = number of characters to write	a0 = number of characters written a0 = -1 if error
Close file	57	a0 = file descriptor	

Table 2.1: Files system calls

Here is a RISC-V program that writes a string to an output file:

```
.data
fout: .asciz "testout.txt" # filename for output
buffer: .asciz "The quick brown fox jumps over the lazy dog."
.text
# Open (for writing) a file that does not exist
li a7, 1024 # system call for open file
la a0, fout # output file name
li a1, 1 # Open for writing (flags are 0: read, 1: write)
ecall # open a file (file descriptor returned in a0)
mv s6, a0 # save the file descriptor
# Write to file just opened
li a7, 64 # system call for write to file
mv a0, s6 # file descriptor
la a1, buffer # address of buffer from which to write
li a2, 44 # hardcoded buffer length
ecall # write to file
# Close the file
li a7, 57 # system call for close file
mv a0, s6 # file descriptor to close
ecall # close file
```

Figure 2.11: Files

### 2.1.6 Lab Assignment

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 for a full mark. Select “Show Labels Window (symbol table)” from the Settings menu in RARS to check the values of your computed addresses.

```

.data
arr1: .word 5:20
arr2: .half 7, -2, 8, -6
arr3: .space 100
str1: .asciz "This is a message"
str2: .asciz "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]?
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]?
4. Write a RISC-V program that defines a one-dimensional array of 10 integers in the static area of the data segment, asks the user to input all 10 array elements, computes, and displays their sum.
5. Write a RISC-V program that allocates an  $n \times n$  array of integers on the heap, where  $n$  is a user input. The program should compute and print the value of each element as follows:

```

for (i=0; i<n; i++)
for (j=0; j<n; j++) {
a[i][j] = i+j;
if (i>0) a[i][j] = a[i][j] + a[i-1][j];
if (j>0) a[i][j] = a[i][j] + a[i][j-1];
print_int(a[i][j]);
print_char(' ');
}
print_char('\n');
}

```

6. Write a RISC-V program to copy an input text file into an output file. The input and output file names should be entered by the user. If the input file cannot be opened, print an error message.