Name: SATYAJIT GHANA Registration Number: 17ETCS002159

Laboratory 8

Title of the Laboratory Exercise: Sorting

1. Introduction and Purpose of Experiment

   Students will create assembly code with sorting techniques and nested loops

2. Aim and Objectives

   Aim

   To develop assembly language program to perform sorting using nested loop structures

   Objectives

   At the end of this lab, the student will be able to

   - use nested loops in assembly
   - perform sorting in ascending/ descending order
   - Build complex looping logic in assembly language

3. Experimental Procedure

   1. Write algorithm to solve the given problem
   2. Translate the algorithm to assembly language code
   3. Run the assembly code in GNU assembler
   4. Create laboratory report documenting the work

4. Questions

   Develop an assembly language program to perform the following

   1. Arrange an array of 'n' numbers in ascending order
   2. Arrange an array of 'n' numbers in descending order
   3. Determine the second smallest number in an array.

5. Calculations/Computations/Algorithms

Name: SATYAJIT GHANA Registration Number: 17ETCS002159

```
# Sorting
.section .data
array:
    .int 5, 4, 3, 2, 1

len:
    .int 5  # length of the array

.section .bss

.section .text

.globl _start

# function for system exit code
_ret:
    movq    $60, %rax               # sys_exit
    movq    $0, %rdi                # exit code
    syscall

# driver function
_start:

    movl $0, %ecx           # initialize the two pointers to 0

fast_pointer_reset:
    movl $0, %edx

compare:
    movl array(, %ecx, 4), %eax
    movl array(, %edx, 4), %ebx
    cmp %eax, %ebx
    jg swap

check_conditions:
    # check if we have gone throughout the array
    cmp %ecx, len
    je _end

inner_loop:
    addl $1, %edx

    # check if the fast pointer has reached the end
    cmp %edx, len
    jne compare

    # if the fast pointer has reached the end reset fast pointer
    # and increment slow pointer
    addl $1, %ecx
    movl $0, %edx
    jmp compare

swap:
    movl array(, %ecx, 4), %eax
    movl array(, %edx, 4), %ebx
    movl %eax, array(, %edx, 4)
    movl %ebx, array(, %ecx, 4)
    jmp check_conditions

_end:
    syscall
    call _ret           # exit

```

Name: SATYAJIT GHANA Registration Number: 17ETCS002159

```
# Sorting
.section .data
array:
    .int 1, 2, 3, 4, 5

len:
    .int 5  # length of the array

.section .bss

.section .text

.globl _start

# function for system exit code
_ret:
    movq    $60, %rax               # sys_exit
    movq    $0, %rdi                # exit code
    syscall

# driver function
_start:

    movl $0, %ecx           # initialize the two pointers to 0

fast_pointer_reset:
    movl $0, %edx

compare:
    movl array(, %ecx, 4), %eax
    movl array(, %edx, 4), %ebx
    cmp %eax, %ebx
    jl swap

check_conditions:
    # check if we have gone throughout the array
    cmp %ecx, len
    je _end

inner_loop:
    addl $1, %edx

    # check if the fast pointer has reached the end
    cmp %edx, len
    jne compare

    # if the fast pointer has reached the end reset fast pointer
    # and increment slow pointer
    addl $1, %ecx
    movl $0, %edx
    jmp compare

swap:
    movl array(, %ecx, 4), %eax
    movl array(, %edx, 4), %ebx
    movl %eax, array(, %edx, 4)
    movl %ebx, array(, %ecx, 4)
    jmp check_conditions

_end:
    syscall
    call _ret           # exit

```

Name: SATYAJIT GHANA Registration Number: 17ETCS002159

```
# Second Smallest
.section .data
array:
    .int 5, 4, 3, 2, 1

len:
    .int 5  # length of the array

.section .bss

.section .text

.globl _start

# function for system exit code
_ret:
    movq    $60, %rax               # sys_exit
    movq    $0, %rdi                # exit code
    syscall

# driver function
_start:

    movl $0, %ecx           # initialize the two pointers to 0

fast_pointer_reset:
    movl $0, %edx

compare:
    movl array(, %ecx, 4), %eax
    movl array(, %edx, 4), %ebx
    cmp %eax, %ebx
    jg swap

check_conditions:
    # check if we have gone throughout the array
    cmp %ecx, len
    je _end

inner_loop:
    addl $1, %edx

    # check if the fast pointer has reached the end
    cmp %edx, len
    jne compare

    # if the fast pointer has reached the end reset fast pointer
    # and increment slow pointer
    addl $1, %ecx
    movl $0, %edx
    jmp compare

swap:
    movl array(, %ecx, 4), %eax
    movl array(, %edx, 4), %ebx
    movl %eax, array(, %edx, 4)
    movl %ebx, array(, %ecx, 4)
    jmp check_conditions

_end:
    # move the second smallest to eax
    movl $1, %ebx
    movl array(, %ebx, 4), %eax

    syscall
    call _ret           # exit

```

6. Presentation of Results

```
(gdb) print array@5
$1 = {5, 4, 3, 2, 1}
(gdb) c
Continuing.

Breakpoint 2, _end () at file.s:61
61          syscall
(gdb) print array@5
$2 = {1, 2, 3, 4, 5}
(gdb)
```

Figure 0-1 Sort Array in Ascending Order

```
(gdb) print array@5
$1 = {1, 2, 3, 4, 5}
(gdb) c
Continuing.

Breakpoint 2, _end () at file.s:62
62          call _ret           # exit
(gdb) print array@5
$2 = {5, 4, 3, 2, 1}
(gdb)
```

Figure 0-2 Sort Array in Descending Order

```
(gdb) print array@5
$1 = {5, 4, 3, 2, 1}
(gdb) c
Continuing.

Breakpoint 1, _end () at file.s:65
65          syscall
(gdb) info register eax
eax            0x2      2
(gdb)
```

Figure 0-3 Find Second Smallest Element

7. Analysis and Discussions

The Array here is sorted using various conditional statements by combination of compare and jump instruction, to sort them we are using insertion sort algorithm, but a little unoptimized version of it, we keep on swapping the current element with every element that is smaller than it, we keep on doing this until we reach the end of the array, the time complexity for such a sorting algorithm is $O(n^2)$.

8. Conclusions

Arrays can be sorted by using a combination of jump, compare and labels. These have to be carefully designed as to avoid infinite loops and array index out of bounds errors, which causes segmentation faults.

9. Comments

1. Limitations of Experiments

Complex Sorting Algorithms such as Tim Sort, Radix Sort, Optimized Quick Sort, are very complex to implement in Assembly although they would provide performance benefits.

2. Limitations of Results

The code written for sorting the elements is unoptimized and would perform worse when the data given to it is in huge amount.

3. Learning happened

Elements of an array can be sorted in Assembly, though with quite a lot of written code.

4. Recommendations

The loop statements should be carefully written to avoid infinite loops.

Signature and date Marks