CMSC 131 Introduction to Computer Organization and Machine-Level Programming
Handout 5: Implementing Iterative Statements

Learning Outcomes

At the end of this session, the student should be able to:

1. understand the implementation of iterative statements in Assembly; and
2. develop Assembly programs implementing iterative statements

Content

I.	Review on Labels and Conditional Control Instructions
II.	Implementing iterative statements using conditional control instructions
III.	Implementing iterative statements using the `loop` instruction

Labels

A program label is a target, or a location to jump to, for control statements. Generally, a label starts with a letter, followed by letters, numbers or symbols (limited to "_"), terminated with a colon (":"). A label must be defined exactly once.

The general form of a label is:

```
<labelName>:
     NEXT SET OF INSTRUCTIONS
```

Conditional Control Instructions

The general form of the compare instruction is:

```
cmp <operand1>, <operand2>
```

NOTES:

- Operand1 and operand2 must be of the same size.
- Operands cannot be both from memory.
- Operand1 cannot be an immediate value, but operand2 may be an immediate value.

The general forms of the conditional jump instruction are:

SIGNED JUMP INSTRUCTION		**UNSIGNED JUMP INSTRUCTION**	
jl <label>	**jg** <label>	**jb** <label>	**ja** <label>
jle <label>	**jge** <label>	**jbe** <label>	**jae** <label>
SIGNED & UNSIGNED JUMP INSTRUCTION			
je <label>		**jne** <label>	

CMSC 131 Introduction to Computer Organization and Machine-Level Programming
Handout 5: Implementing Iterative Statements

Iteration

Iteration is a mechanism that allows the repetition of a sequence of statements provided that a certain condition is true. It only passes the control to the next statement once the test or condition evaluates to false. In assembly, a basic loop can be implemented into two major ways:

1. Using conditional control instructions
2. Using the **`loop`** instruction

Iteration using Conditional Control Instructions

The implementation of a basic loop in assembly consists of a *counter* which can be checked at either the bottom or start of a loop, a `cmp` *instruction*, and a *conditional jump*.

EXAMPLE 1:

```
global _start

section .data
      SYS_EXIT equ 60
      lpCnt dq 5
      sum dq 0

section .text
_start:
      mov rcx, qword[lpCnt]      ; loop counter
      mov rax, 1                 ; odd integer counter

sum_loop:
      add qword[sum], rax         ; sum += current odd integer
      add rax, 2                  ; set next odd integer
      dec rcx                     ; decrement loop counter
      cmp rcx, 0
      jne sum_loop

exit_here:
      mov rax, SYS_EXIT
      xor rdi, rdi
      syscall
```

In Example 1, `rax` is used to hold the value of the current odd integer to be added. On the other hand, `rcx` is used as the program's loop counter.

EXAMPLE 2:

```
global _start

section .data
      SYS_EXIT equ 60
      lpCnt dq 5
      sum dq 0

section .text
_start:
      mov rcx, qword[lpCnt]      ; loop counter
      mov rax, 1                 ; odd integer counter

sum_loop:
      cmp rcx, 0
      je exit_here
      add qword[sum], rax         ; sum current odd integer
      add rax, 2                  ; set next odd integer
      dec rcx                     ; decrement loop counter
      jmp sum_loop

exit_here:
      mov rax, SYS_EXIT
      xor rdi, rdi
      syscall
```

Iteration using the `loop` Directive

The process that is shown in Example 1 can be outlined as follows:

1. Execute a sequence of instructions included in the loop mechanism
2. Decrement the `rcx` register
3. Compare `rcx` with `0`.
4. Jump to the specified label if `rcx != 0`

In assembly, the `loop` instruction provides the same functionality as steps 2 to 4 combined (provided that the label was only declared once).

CODE SET 1	**CODE SET 2**

`loop <label>`	`dec rcx`
	`cmp rcx, 0`
	`jne <label>`

Let us rewrite our code from Example 1.

EXAMPLE 3:

```
global _start

section .data
      SYS_EXIT equ 60
      lpCnt dq 5
      sum dq 0

section .text
_start:
      mov rcx, qword[lpCnt]      ; loop counter
      mov rax, 1                 ; odd integer counter

sum_loop:
      add qword[sum], rax        ; sum current odd integer
      add rax, 2                 ; set next odd integer
      loop sum_loop

exit_here:
      mov rax, SYS_EXIT
      xor rdi, rdi
      syscall
```

NOTE:
The `rcx` register is always decremented and checked in the `loop` instruction. It is therefore imperative to always set the value of the `rcx` register before your iterative statements. Forgetting to set your `rcx` could affect the number of times that your program may repeat which can also lead to execution and debugging errors.

The `loop` instruction is also *limited to the* `rcx` *register and to counting down*. Nested loops may require additional steps for saving and restoring the value of `rcx` to avoid conflicts between the inner and the outer loop.

References

Jorgensen, Ed. 2019. x86-64 Assembly Language Programming with Ubuntu. Version 1.1.40.