Accelerated Learning Series

(www.ALearnOnline.com – A site dedicated to education)

Modules in Computer Science & Engineering

X64 Assembler Programming

A set of notes detailing fundamental concepts in low level Programming.

Author: SKG.
Dec 2007.

Revision History

Version 1.0 (Dec 2007)

- First version created

Revision History

- Version 1.0: First version created – Dec 2007
- Version 1.1: Adapted for a Linux Bash Environment – Sept 2017

Table of Contents.

Preface

As I started working on this set of notes, I was confounded by many questions related to the most fruitful approach to take. To start with it is fair to question the relevance of Assembler level programming in the midst of the more popular and pervasive high level programming languages. Once one justifies its relevance, there is still the question of which processor to choose to illustrate the concepts in Assembler programming.

Over the last two decades I have observed the number of Assembler programmer positions in the industry dwindle substantially. Today there are probably only two computer industries that exploit the skills of Assembler programmers. The first would be those who write very specialized software that cannot be adequately generated by generic compilers. The other industry would be those who need to understand and resolve difficult problems that hinge on the hardware in which the problems manifest. Besides these two industries however, the average high level computer programmer also benefits, to a lesser extent admittedly, from knowledge of Assembler programming in understanding shortcomings of a compiler and sometimes even revealing faults in their high level code. Even if you find that you will never program a computer at the Assembler level, a preliminary course in Assembler programming primes the student for better grasping constructs in higher level languages.

Since the Personal Computer (PC) has become ubiquitous and since the PC is based on the **Intel** x86 processor, it seemed convenient to use the x86 architecture for illustration here.

The x86 processor operates in 64-bit mode in most modern personal computers. Hence I have adapted an earlier set of notes that used the 16-bit mode into the 64-bit mode. The samples in these notes require the GAS assembler.

1.0 – x64 Architecture

The general purpose computer architecture prepared us with an understanding of the various components that make up a computer. Hence we are familiar with concepts surrounding the CPU, Registers, Memory, data buses, address buses, instruction mnemonics and operands. In this section we delve a little deeper into a specific processor type - namely the x64 processor.

The emphasis in this section however, will be to expose the student to the mechanics of writing assembler code, converting that to machine code and finally executing and debugging the code.

1.1 – x64 Register set

As discussed in the General Purpose Computer Architecture notes, Registers are very fast access memory locations in the CPU.

The x64 processor has the following types of Registers:

64-bit General Purpose Registers – RAX, RBX, RCX, RDX, RBP, RDI, RSP and R8 to R15
Pointer Registers – RIP, RSP
Flags Registers – RFLAGS
Floating Point Registers – FPR0 to FPR7

In addition to the above there are segment registers (not commonly used in x64), Control registers, memory management registers, debug registers, virtualization registers, performance registers etc.

General Registers:

A byte is defined as 8 bits, a word is 16 bits, a double word is 32 bits, a quadword is 64 bits and a double quadword is 128 bits. Intel uses the "little endian" format where lower significant bytes are stored in lower memory addresses.

For the first eight registers, replacing "r" with "e" will allow you to access the double words at the lower significant addresses.

For the RAX, RBX, RCX, and RDX registers removing the "r" will allow you to access the words at the lower significant addresses.

Index Registers:

Some computer instructions operate on contiguous memory locations starting at a particular address and for a certain size. A common example would be an instruction that copies a string (an array of characters that often ends with a NULL character) from one location in memory to another location. This instruction would need the start address of the source string, the length of the source string and the start address of the destination string. The instruction can then **index** with reference to the start address of the source and destination locations to access each subsequent memory location.

The "**RSI**" and "**RDI**" are both 64-bit registers that are commonly used as **source** and **destination index registers**.

Pointer Registers:

"**RSP**" is the 64-bit **Stack Pointer register**.

"**RIP**" is the 64-bit **Instruction Pointer register**.

"**RBP**" is the 64-bit Base Pointer register, that is commonly used by functions to save the "RSP" register before reusing the "RSP" register to allocate memory on the stack of local variables.

Flags Register:

The CPU stores the results of certain operations in the "RFLAGS" register. The following defines the 8 commonly used flag bits in the flags register:

Symbol	Bit	Name	Set if
CF	0	Carry	Operation generated a carry or borrow
PF	2	Parity	Last byte has an even number of 1's, else 0
AF	4	Adjust	Carry or borrow out of the four least significant bits (BCD support)

ZF	6	Zero	Result was 0
SF	7	Sign	Most significant bit of results is 1
IF	9	Interrupt	Interrupt Enable
DF	10	Direction	Direction string instructions operate (increment or decrement)
OF	11	Overflow	Overflow on signed operation

Common uses of registers:

RDI	– Arg 1
RSI	– Arg 2
RDX	– Arg 3
RCX	– Arg 4
R8	– Arg 5
R9	– Arg 6
RAX	– return
RSP	– stack pointer
RIP	– instruction pointer
EFLAGS	- flags

1.2 – Assembler Directives

A computer program is nothing more than a set of instructions. Some of these instructions target the software that converts the program to machine language. The rest of the instructions are meant to be executed by the processor for which the program is written.

The software that coverts an assembly language program to machine language is called an **Assembler**. The instructions directed at the assembler are referred to as **Assembler directives**. Since these directives are Assembler specific, they will not apply to all Assemblers. In the discussion below, I am referring to the GAS Assembler. Let us examine some common GAS Assembler directives.

Data Section Directive:

Most programs will need to use certain constant values throughout the program. It would be useful to refer to these "**constants"** by name and define the constant just once and get the assembler to replace every instance where that name is used with its constant value. This aids the programmer by having to change the constant in only a single location instead of every location where it is used, should the constant value have to change at some point.

The Data section serves this purpose. The following is an example of a data section:

```
.data
        CR              =               13
        LF              =               10
        msgStr:         .ascii          "Hello World!\n"
        msgStrpost:     .byte           CR, LF
        msgStrLen       =               .-msgStr
```

".data" is a keyword known to the Assembler. It tells the Assembler that you intend to define initialized data and constants in this section. "CR" is a coder friendly name for Carriage Return, while "LF" stands for Line Feed. They have ASCII codes of 13 and 10 respectively. "msgStr" is a name to track the starting location of the string we want to use. "msgStrLen" is a constant that defines the length of our string. Note how we add the bytes for CR and LF to the end of our message string. In defining "msgStrLen", the "." indicates to the assembler that you are referring to the current location in memory. Hence ". - msgStr" will give us the number of bytes in the message string.

BSS Section Directive:

Every program will need some reserved memory to keep track of values that change from time to time. For ease of use, we will give these memory locations unique names and will refer to them as "**variables**".

Below is an example of the ".bss" Assembler directive to define memory locations for variables.

```
section .bss

        var1 resb       32
```

Here we are defining a variable called "var1" and reserving 32 bytes for it.

Code Section Directive:

The code segment is defined using the keyword "Segment" with the 'text' identifier as shown below.

```
section .text

        global main

        main:
```

```
        ;write (1, message, 13)
        mov rax, 1
        mov rdi, 1
        mov rsi, message
        mov rdx, msgLen
        syscall

        ;exit(0)
        mov rax, 60
        xor rdi, rdi
        syscall
```

The “Code” segment contains the actual instructions that are intended to be executed by the processor.

The “mov” instructions are instructions known to the x64 processor. They are not assembler directives. “mov rax, 1” is an instruction asking the processor to move “1” to the 64-bit “rax” register.

Operand Syntax:

$	-	Constants start with $
%	-	Registers start with %
()	-	Parenthesis dereference addresses in registers of variables.

1.3 – Your First Computer Program – “Hello World!”

A program is generally written using a text editor. If you don’t have a favorite editor, you can use Notepad++.

Type the following lines of code into your notepad editor, or copy and paste it into the editor.

```
#================================================================
# File:                 Sample1.s
# Assemble:             gcc -c Sample1.s
# Link:                 ld Sample1.o –o Sample1
# Run:                  ./Sample1
#================================================================

# Add the global directive so the symbol "_start" is made available
# in the object code export table.
# If the symbol is not in the export table at link time, the linker
# will not know about it.
# _start is the default entrypoint for an executable and if the linker
# can see it, it will use that as the Entrypoint.
# If you want a different entrypoint, you can use the -e link option.
# Eg. ld -e main Sample1.o
.global _start

# This is the section where the assembler assumes your code is located
.text

__start:
               # write(1, msgStr, msgStrLen)
               # "1" is the sys code for write.
               # Note the "$1" tells the assembler to use the value "1".
        mov     $1,          %rax

               # "1" is the stdout file handle.
        mov     $1,          %rdi

               # Address of string to output.
               # Note here we are passing a variable prefixed by "$".
               # The Assembler will replace $msgStr with the address of msgStr.
        mov     $msgStr,     %rsi


               # Number of bytes in msgStr.
               # Note here we are passing a constant.
               # Invoke operating system to do the write.
        mov     $msgStrLen, %rdx

               # System call 60 is exit.

        syscall

               # exit(0)
               # We want return code 0.
               # Invoke operating system to exit.
        mov     $60, %rax
        xor     %rdi, %rdi
        syscall

# This is the section where the assembler expects initialized data
# Data types recognized include .byte (1 byte), .short (2 bytes), .long (4 bytes),
# .string or .ascii (length based on length of string). Constants are defined
# with the “=” sign.
.data
       CR              =              13
       LF              =              10
       msgStr:         .ascii         "Hello World!\n"
       msgStrpost:     .byte          CR, LF
```

```
        msgStrLen       =               .-msgStr
```

Save the file as “Sample1.asm” and use the following commands to assemble, link and run this code using GAS…

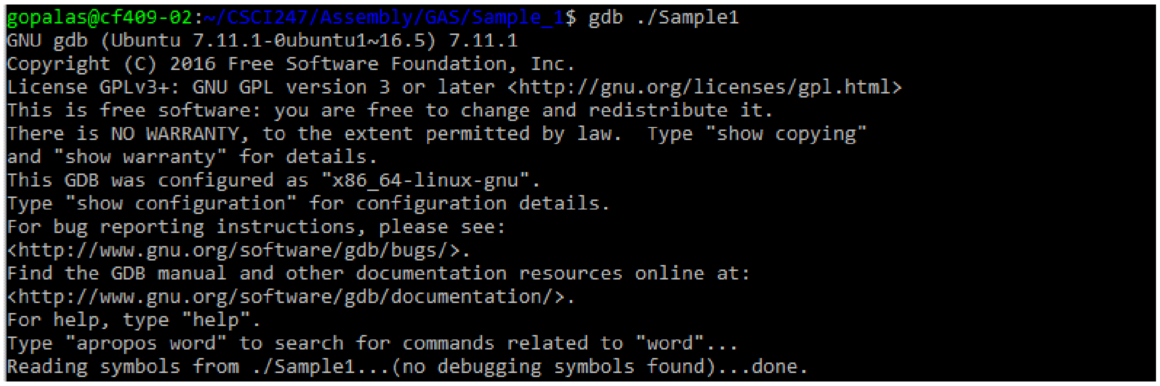
```
        gcc -c Sample1.s
        ld Sample1.o –o Sample1
        ./Sample1
```

As an aside, note that there are two common file formats used for object and executable files – COFF and ELF. COFF stands for Common Object File Format while ELF stands for Executable and Linking Format. Microsoft Visual C++ compilers generate the COFF format while GCC generates the ELF format.

1.4 – Debugging a program

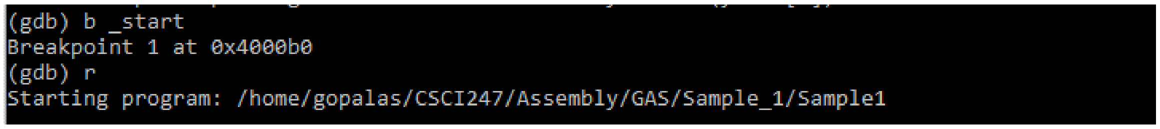
Once you get through the Assembler and linker phase and create an executable file, there is always an urge to run the program and see if it behaves as you expect it to. It is advisable to use a tool called a "**debugger**" to walk through the code to ensure that the logic that is being executed is exactly what was intended. A debugger allows you to walk over (trace) individual instructions and confirm the output of each instruction.

"gdb" is GNU debugger available on Linux. To load "Sample1.exe" in the debugger, type "**gdb <filename>**" in the directory where sample1.exe is located.



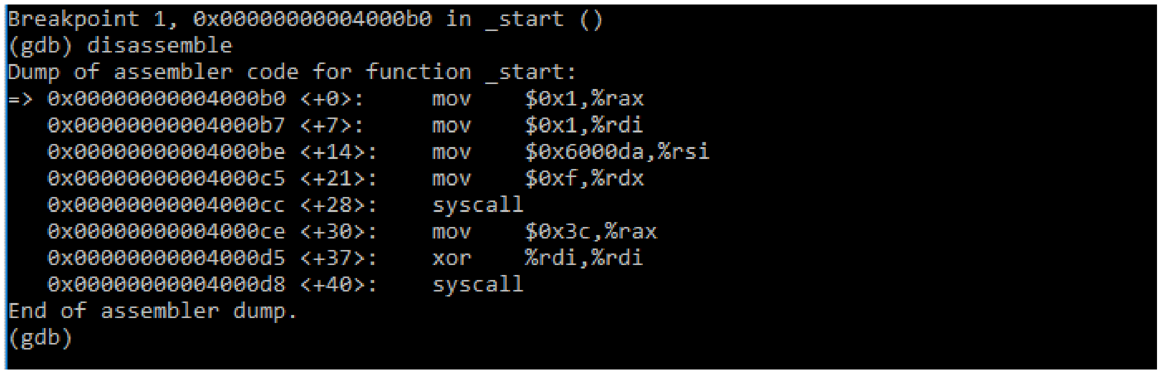
```
gopalas@cf409-02:~/CSCI247/Assembly/GAS/Sample_1$ gdb ./Sample1
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./Sample1...(no debugging symbols found)...done.
```

You can now set a breakpoint at the start of the program ("main") and run to the breakpoint…



```
(gdb) b _start
Breakpoint 1 at 0x4000b0
(gdb) r
Starting program: /home/gopalas/CSCI247/Assembly/GAS/Sample_1/Sample1
```

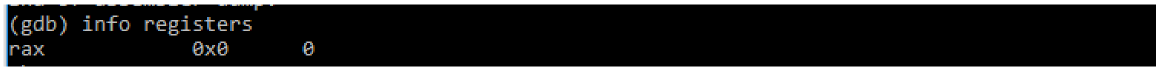
Once you hit the breakpoint, you can disassemble the program and look at how the Assembler assembled your code…



```
Breakpoint 1, 0x00000000004000b0 in _start ()
(gdb) disassemble
Dump of assembler code for function _start:
=> 0x00000000004000b0 <+0>:     mov    $0x1,%rax
   0x00000000004000b7 <+7>:     mov    $0x1,%rdi
   0x00000000004000be <+14>:    mov    $0x6000da,%rsi
   0x00000000004000c5 <+21>:    mov    $0xf,%rdx
   0x00000000004000cc <+28>:    syscall
   0x00000000004000ce <+30>:    mov    $0x3c,%rax
   0x00000000004000d5 <+37>:    xor    %rdi,%rdi
   0x00000000004000d8 <+40>:    syscall
End of assembler dump.
(gdb)
```

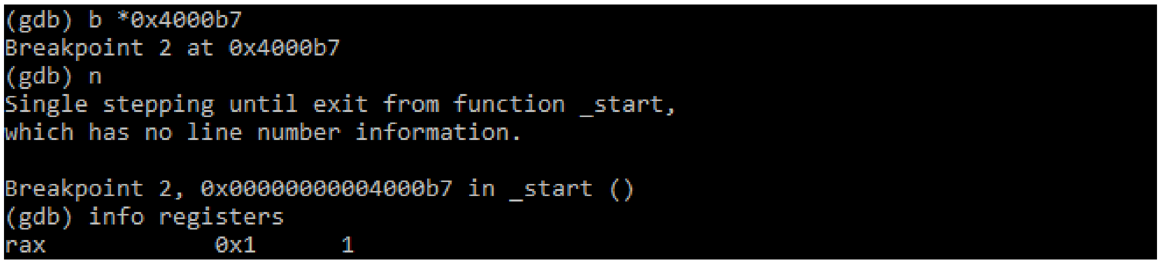
Note the code at offset +14. The Assembler is using the address of "msgStr", whereas in the code in offset +21, it is using the value of the constant "msgStrLen".

Now look at the value of the rax register before you execute your "mov" instruction…



```
(gdb) info registers
rax            0x0      0
```

Now set a breakpoint right after the first move instruction and step over the “mov” instruction and observe the register again…



```
(gdb) b *0x4000b7
Breakpoint 2 at 0x4000b7
(gdb) n
Single stepping until exit from function _start,
which has no line number information.

Breakpoint 2, 0x00000000004000b7 in _start ()
(gdb) info registers
rax            0x1      1
```

Learn some of the other command available in gdb to access your variables and get familiar with using the debugger. It will prove to be the most valuable tool at your disposal. You can find some of the gdb documentation here.

.

1.5 – Interacting with the User

In almost any computer program, there is a need to get input from the user of the program. The program's behavior is often dependent on the input that is supplied by the user. In this section we will write a sample that gets input from the user using the Standard Input.

The most efficient way to master any programming language is to practice writing your own sample programs. Hence I encourage you to study the samples provided in each of these sections and attempt to duplicate their behavior on your own by using the same or similar instructions.

Another very useful programming technique is to do incremental additions. For example, with the sample below, you can first try and put up a prompt to the user. Then try and collect information from the user and then finally try and display the received input. At each of these interim stages, assemble and link your program to confirm that it is behaving as you would expect.

The only new construct that is introduced in the sample below is the use of the READ function code. This code indicates to LINUX that you are asking it to read input from the user. The number of characters read by LINUX is going to be available in the AX register. This is also referred to as the "return value". In the x86 assembler, the return value from any call is usually passed using the AX register.

Once you have studied the following sample, assemble and link the sample using the commands provided in the header of the sample.

You may observe that in the sample below, I have used the "xor ax, ax" instruction when I wanted to zero the contents of a register. An "xor" instruction is an exclusive-or operation. So if you apply that operation to the same register in the source and destination operand fields, you are bound to make the contents of the register to be zero.

You may wonder why I chose an "xor" over a more direct "mov ax, 0" instruction. This has to do with efficiency. In the older processors, a "mov" instruction from a memory to a register would use 4 clock cycles of the CPU, whereas an xor usually cost only 2 clock cycles. Needless to say these sorts of savings are not worth much (if anything at all) with the increased clock speeds and more efficient instructions of modern processors.

```
#================================================================
# File:                 Sample2.s
# Assemble:             gcc -c Sample2.s
# Link:                 ld Sample2.o –o Sample2
# Run:                  ./Sample2
#================================================================

.global _start

# This is the section where the assembler assumes your code is located
.text

_start:

        #write (1, promptMsg, msgLen)
        mov     $1,            %rax
        mov     $1,            %rdi
        mov     $promptMsg,    %rsi
        mov     $promptMsgLen, %rdx
        syscall

       #read (0, inputBuffer, bufLen)
       mov     $0,            %rax
       mov     $0,            %rdi
       lea     (inputBuffer), %rsi
       mov     $bufLen,       %rdx
       syscall
```

```
        #Add and ! and CR to buffer
        lea     (inputBuffer), %rsi
        dec     %rax
        movb    $33,           (%rax, %rsi, 1)
        inc     %rax
        movb    $CR,           (%rax, %rsi, 1)
        inc     %rax
        movb    $LF,           (%rax, %rsi, 1)
        inc     %rax
        mov     %rax,          (nameLen)

       #write (1, Greetings, GreetingLen)
       mov     $1,                            %rax
       mov     $1,                            %rdi
       mov     $Greetings,            %rsi
       mov     $GreetingLen,  %rdx
       syscall

        #write (1, inputBuffer, nameLen)
        mov    $1,                    %rax
        mov    $1,                            %rdi
        mov    $inputBuffer,  %rsi
        mov    nameLen,               %rdx
        syscall

        #exit(0)
        mov    $60,                   %rax
        xor    %rdi,                  %rdi
        syscall

#================================================================

.data
        CR              =                       13
        LF              =                       10
        bufLen          =                       100

        promptMsg:      .ascii          "Enter your name: "
        promptMsgLen    =               .-promptMsg

        CR_LF_1:        .byte           CR, LF
        Greetings:      .ascii          "Hello"
        GreetingLen     =               .-CR_LF_1

#================================================================

# This is the section where the assembler expects uninitialized data
.bss

        .lcomm          inputBuffer,   bufLen
        .lcomm          nameLen,               4
```

2.0 – Jump Instructions

In the previous section we gained the knowledge to write sequential instructions that let us get input from a user and process it. While sequential instructions are the primary mechanism describing a set of steps, it can be very limiting in our ability to reuse our code.

Imagine how difficult it would be if we had to explicitly write the code to read user input every time we needed to get user input within a program. It would be so much easier if we could jump to the code that does the reading of user input every time we need to get user input.

The construct of Jump instructions is designed to do just that. They allow the coder to move around the program without the sequential processing limitation.

In this section we will study the common Jump instructions available in the x86 architecture.

2.1 – Unconditional Jumps (JMP)

An unconditional jump, as the name implies, allows the transfer of execution from one part of your program to another without any conditions.

All jump instructions work by altering the value of the IP register.

Below is a sample that shows the operation of the Jmp mnemonic. This modifies the previous sample to skip the greetings message.

```
#================================================================
# File:                 Sample3.s
# Assemble:             gcc -c Sample3.s
# Link:                 ld Sample3.o –o Sample3
# Run:                  ./Sample3
#================================================================

.global _start

# This is the section where the assembler assumes your code is located
.text

_start:

        #write (1, promptMsg, msgLen)
        mov     $1,            %rax
        mov     $1,            %rdi
        mov     $promptMsg,    %rsi
        mov     $promptMsgLen, %rdx
        syscall

       #read (0, inputBuffer, bufLen)
       mov     $0,            %rax
       mov     $0,            %rdi
       lea     (inputBuffer), %rsi
       mov     $bufLen,       %rdx
       syscall

        #Add an "!" and CR to buffer
        lea     (inputBuffer), %rsi
        dec     %rax
        movb    $33,           (%rax, %rsi, 1)
        inc     %rax
        movb    $CR,           (%rax, %rsi, 1)
        inc     %rax
        movb    $LF,           (%rax, %rsi, 1)
        inc     %rax
        mov     %rax,          (nameLen)

        #Unconditional jump
        jmp            SkipGreetings

       #write (1, Greetings, GreetingLen)
       mov     $1,                           %rax
       mov     $1,                           %rdi
       mov     $Greetings,           %rsi
       mov     $GreetingLen,  %rdx
       syscall

SkipGreetings:

        #write (1, inputBuffer, nameLen)
        mov    $1,                   %rax
        mov    $1,                           %rdi
        mov    $inputBuffer,  %rsi
        mov    nameLen,              %rdx
        syscall
        #exit(0)
        mov    $60,                  %rax
        xor    %rdi,                 %rdi
        syscall

#================================================================

.data
       CR             =                      13
       LF             =                      10
       bufLen         =                      100

       promptMsg:     .ascii         "Enter your name: "
       promptMsgLen   =              .-promptMsg

       CR_LF_1:       .byte          CR, LF
       Greetings:     .ascii         "Hello"
       GreetingLen    =              .-CR_LF_1

#================================================================

# This is the section where the assembler expects uninitialized data
.bss

       .lcomm         inputBuffer,   bufLen
       .lcomm         nameLen,               4
```

2.2 – Compare instruction (CMP)

Recall the flags register…

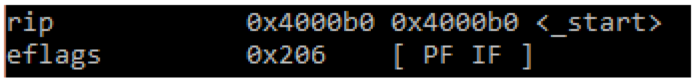
Symbol	Bit	Name	Set if
CF	0	Carry	Operation generated a carry or borrow
PF	2	Parity	Last byte has an even number of 1's, else 0
AF	4	Adjust	Carry or borrow out of the four least significant bits (BCD support)
ZF	6	Zero	Result was 0
SF	7	Sign	Most significant bit of results is 1
IF	9	Interrupt	Interrupt Enable
DF	10	Direction	Direction string instructions operate (increment or decrement)
OF	11	Overflow	Overflow on signed operation

The compare instruction is essentially a subtract instruction that does not alter the value of the operands but impacts the value of the flags registers just like a subtract instruction would. We study the compare instruction because its impact on the flags register is exploited by many Jump instructions.

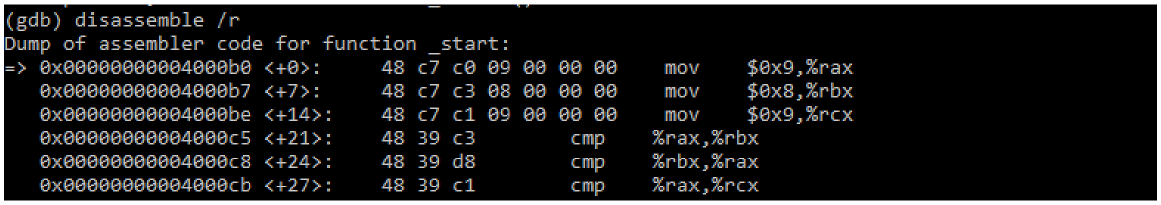
You can type the following instructions into one of your earlier samples and trace each instruction and see how it impacts the flags register.

```
mov             $9, %rax
mov             $8, %rbx
mov             $9, %rcx

cmp             %rax, %rbx

cmp             %rbx, %rax

cmp             %rax, %rcx
```

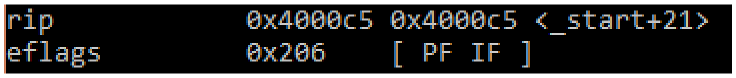
The following debugger output shows that the three "mov" instructions did not impact the flags register.



```
rip            0x4000b0 0x4000b0 <_start>
eflags         0x206    [ PF IF ]
```

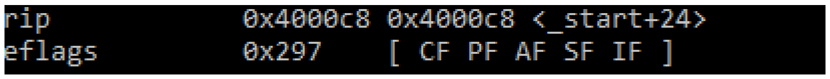


```
(gdb) disassemble /r
Dump of assembler code for function _start:
=> 0x00000000004000b0 <+0>:     48 c7 c0 09 00 00 00    mov    $0x9,%rax
   0x00000000004000b7 <+7>:     48 c7 c3 08 00 00 00    mov    $0x8,%rbx
   0x00000000004000be <+14>:    48 c7 c1 09 00 00 00    mov    $0x9,%rcx
   0x00000000004000c5 <+21>:    48 39 c3        cmp    %rax,%rbx
   0x00000000004000c8 <+24>:    48 39 d8        cmp    %rbx,%rax
   0x00000000004000cb <+27>:    48 39 c1        cmp    %rax,%rcx
```



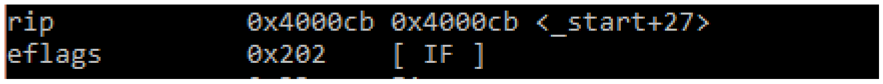
```
rip            0x4000c5 0x4000c5 <_start+21>
eflags         0x206    [ PF IF ]
```

The "cmp %rax, %rbx" involves (%rbx - %rax or "8 – 9"). In signed arithmetic, this leads to "-1". So we expect the "CF", "AF" and "SF" flags to be set…



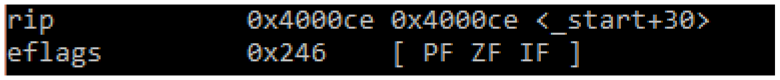
```
rip            0x4000c8 0x4000c8 <_start+24>
eflags         0x297    [ CF PF AF SF IF ]
```

The “cmp %rbx, %rax” involves (%rax - %rbx or “9 - 8”). In both signed and unsigned arithmetic, this leads to a +1. So we expect the previously set flags to be cleared...



```
rip            0x4000cb 0x4000cb <_start+27>
eflags         0x202    [ IF ]
```

The “cmp %rax, %rcx” involves “9 – 9”. This will yield “0” and hence the “ZF” and the “PF” flag is set...



```
rip            0x4000ce 0x4000ce <_start+30>
eflags         0x246    [ PF ZF IF ]
```

2.3 – Zero or Equality Jumps (JZ, JE, JNZ, JNE)

.The Jump Zero (“JZ”) and the Jump Equal (“JE”) instructions do the exact same thing – they both check if the ZERO flag is set and if it is, they jump to the tag provided in the operand.

Similarly the Jump Not Zero (“JNZ”) and the Jump Not Equal (“JNE”), jump to the tag provided in the operand if the ZERO flag is not set.

The sample below demonstrates the use of these instructions. Note that I have used the “JE” and “JNE” instructions. You can replace these with “JZ” and “JNZ” respectively, without altering the behavior.

Instead of moving “8” to the rax register, change the code to move “9” into the ax register and confirm that the jump to “JUMP_ZERO” tag gets executed.

Note that the Jump instructions do not change the value of the flags register and so we can have multiple conditional jumps subsequent to the compare instruction.

```
#===============================================================
# File:                 Sample4.s
# Assemble:             gcc -c Sample4.s
# Link:                 ld Sample4.o -o Sample4
# Run:                  ./Sample4
#===============================================================


.global _start

.text

_start:
               mov            $8, %rax
               mov            $9, %rbx
               cmp            %rax, %rbx
               je             JUMP_ZERO
               jne            JUMP_NOT_ZERO

JUMP_ZERO:

               mov     $1,                  %rax
               mov     $1,                  %rdi
               mov     $zeroMsg,            %rsi
               mov     $zeroMsgLen,   %rdx
               syscall
               jmp            EXIT


JUMP_NOT_ZERO:

               mov     $1,                  %rax
               mov     $1,                         %rdi
               mov     $nonZeroMsg,   %rsi
               mov     $nonZeroMsgLen,      %rdx
               syscall

EXIT:
               #exit(0)
               mov     $60,                 %rax
               xor     %rdi,                %rdi
               syscall

#===============================================================

.data
       CR                     =                     13
       LF                     =                     10
```

```
        zeroMsg:                .ascii          "Zero Message"
        CR_LF_1:                .byte           CR, LF
        zeroMsgLen              =                       .-zeroMsg

        nonZeroMsg:             .ascii          "Non Zero Message"
        CR_LF_2:                .byte           CR, LF
        nonZeroMsgLen  =                        .-nonZeroMsg

#=================================================================
```

2.4 – Unsigned Jumps (JA, JAE, JB, JBE)

The Unsigned jumps use the ZERO and CARRY flags.

Jump if Above ("JA") instruction jumps to the tag provided in the operand if both the ZERO flag and the CARRY flag are not set. If the ZERO flag is set, we know the numbers used in the last compare were equal. If the CARRY flag was set we know the last compare involved subtracting a larger number from a smaller number. If neither of these flags were set, the last compare involved subtracting a smaller number from a larger number. In this case the "JA" instruction will jump to the tag provided in the operand.

Similarly Jump if Above or Equal ("JAE") causes a jump if either the Zero flag is set or if CARRY flag is not set.

Jump if Below ("JB") instruction jumps to the tag provided in the operand if the ZERO flag is not set but the CARRY flag is set.

Jump if Below or Equal ("JBE") causes a jump if either the Zero flag is set or if the CARRY flag is set.

Another way to look at these instructions is to consider a compare between unsigned numbers. If the first number is equal to the second number, the ZERO flag is set. So both the "JAE" and "JBE" will cause a jump in this case.

If the first number is greater than the second number, both the ZERO flag and the CARRY flag are not set. In this case both the "JA" and "JAE" instructions will cause a jump.

If the first number is less than the second number, the ZERO flag is not set, but the CARRY flag is set. In this case both the "JB" and "JBE" will cause a jump.

Hence these jump instructions are designed to compare unsigned numbers.

In the sample below change the values in the ax and bx register before the compare instruction and try and explain why it is not possible to jump to the "JUMP_BELOW_EQUAL" tag irrespective of the values used in ax and bx.

```
#=================================================================
# File:                Sample5.s
# Assemble:            gcc -c Sample5.s
# Link:                ld Sample5.o -o Sample5
# Run:                 ./Sample5
#=================================================================


.global _start

.text

_start:
               mov            $8, %rax
               mov            $9, %rbx
               cmp            %rax, %rbx
               ja             JUMP_ABOVE
               jae            JUMP_ABOVE_EQUAL
               jb             JUMP_BELOW
               jbe            JUMP_BELOW_EQUAL

JUMP_ABOVE:

               mov     $1,                  %rax
               mov     $1,                  %rdi
               mov     $jumpAboveMsg, %rsi
               mov     $JA_MsgLen,    %rdx
               syscall
               jmp            EXIT

JUMP_ABOVE_EQUAL:

               mov     $1,                    %rax
               mov     $1,                           %rdi
               mov     $jumpAboveEqual,%rsi
               mov     $JAE_MsgLen,   %rdx
               syscall
               jmp            EXIT

JUMP_BELOW:

               mov     $1,                    %rax
               mov     $1,                    %rdi
               mov     $jumpBelowMsg, %rsi
               mov     $JB_MsgLen,    %rdx
               syscall
               jmp            EXIT

JUMP_BELOW_EQUAL:

               mov     $1,                    %rax
               mov     $1,                           %rdi
               mov     $jumpBelowEMsg,        %rsi
               mov     $JBE_MsgLen,   %rdx
               syscall

EXIT:
                #exit(0)
                mov    $60,                   %rax
                xor    %rdi,                  %rdi
                syscall

#=================================================================
.data
       CR             =              13
       LF             =              10

       jumpAboveMsg:  .ascii         "Jump Above"
                      .byte          CR, LF
       JA_MsgLen      =              .-jumpAboveMsg

       jumpAboveEqual:.ascii         "Jump Above or Equal"
                                     .byte   CR, LF
       JAE_MsgLen     =              .-jumpAboveEqual

       jumpBelowMsg:  .ascii         "Jump Below"
                      .byte          CR, LF
       JB_MsgLen      =              .-jumpBelowMsg

       jumpBelowEMsg: .ascii         "Jump Below or Equal"
                      .byte          CR, LF
       JBE_MsgLen     =              .-jumpBelowEMsg

#=================================================================
```

2.5 – Signed Jumps (JG, JGE, JL, JLE)

The Signed jumps use the SIGN, ZERO and OVERFLOW flags.

Jump if Greater (“JG”), Jump if Greater or Equal (“JGE”), Jump if Less (“JL”) and Jump if Less or Equal (“JLE”) are very similar to the “JA”, “JAE”, “JB”, “JBE” instructions respectively. However these apply to signed numbers.

The sample below shows the use of signed jumps.

```
#================================================================
# File:                 Sample6.s
# Assemble:             gcc -c Sample6.s
# Link:                 ld Sample6.o -o Sample6
# Run:                  ./Sample6
#================================================================


.global _start

.text

_start:
                       mov            $-4, %rax
                       mov            $-8, %rbx
                       cmp            %rax, %rbx            #%rbx - %rax
                       jg             JUMP_GREATER
                       jge            JUMP_GREATER_EQUAL
                       jl             JUMP_LESS
                       jle            JUMP_LESS_EQUAL

JUMP_GREATER:
                       mov     $1,                   %rax
                       mov     $1,                   %rdi
                       mov     $jumpGreaterMsg,%rsi
                       mov     $JG_MsgLen,    %rdx

                       syscall
                       jmp            EXIT

JUMP_GREATER_EQUAL:
                       mov     $1,     %rax
                       mov     $1,     %rdi
                       mov     $jumpGreaterEq, %rsi
                       mov     $JGE_MsgLen,   %rdx
                       syscall
                       jmp            EXIT

JUMP_LESS:
                       mov     $1,            %rax
                       mov     $1,            %rdi
                       mov     $jumpLessMsg,  %rsi
                       mov     $JL_MsgLen,    %rdx
                       syscall
                       jmp            EXIT

JUMP_LESS_EQUAL:
                       mov     $1, %rax
                       mov     $1, %rdi
                       mov     $jumpLessEMsg, %rsi
                       mov     $JLE_MsgLen,   %rdx
                       syscall

EXIT:
                #exit(0)
                mov    $60,    %rax
                xor    %rdi,   %rdi
                syscall

#================================================================

.data
       CR                             =                      13
       LF                             =                      10

       jumpGreaterMsg:        .ascii          "Jump Greater"
                                      .byte          CR, LF
       JG_MsgLen              =                      .-jumpGreaterMsg

       jumpGreaterEq: .ascii          "Jump Greater Equal"
                                      .byte          CR, LF
       JGE_MsgLen             =                      .-jumpGreaterEq

       jumpLessMsg:   .ascii          "Jump Less"
                                      .byte          CR, LF
       JL_MsgLen              =                      .-jumpLessMsg

       jumpLessEMsg:  .ascii          "Jump Less Equal"
                                      .byte          CR, LF
       JLE_MsgLen             =                      .-jumpLessEMsg

#================================================================
```

2.6 – Other Jumps (JC, JNC, JO, JNO, JS, JNS, JCXZ)

In addition to the Unconditional, Zero, Unsigned and Signed jumps, there are three other jumps that target specific flags.

Jump if Carry ("JC") executes a jump if the CARRY flag is set. Similarly Jump if No Carry ("JNC") jumps if the CARRY flag is not set.

Jump if Overflow ("JO") executes a jump if the Overflow flag is set. Similarly Jump if No Overflow ("JNO") jumps if the Overflow flag is not set.

Jump if Sign ("JS") executes a jump if the Sign flag is set. Similarly Jump if No Sign ("JNS") jumps if the Sign flag is not set.

Yet another jump instruction that can be very useful in implementing a loop with a specific number of iterations is the Jump if RCX Zero ("JRCXZ") instruction.

Below is a sample showing how the "JRCXZ" instruction can be used to implement a loop. Note that I have used three new instructions in this sample – "PUSH", "POP" and "DEC".

The "**PUSH**" and "**POP**" instructions are mechanisms to save data on the stack. In the sample below, the value of the "RCX" register could potentially be overwritten. Hence I saved its value on the stack with the "PUSH" instruction and later restored it with the "POP" instruction.

The "PUSH" instruction effectively translates to the following sub instructions;

```
sub     $8, %rsp
mov      <quad data to be saved>, (%rsp)
```

First we decrement the stack pointer (note the stack grows to lower addresses), then we save the data in the new location of the stack pointer. We will talk more about the indirect addressing mode (the parenthesis around %rsp) later on.

The "POP" instruction is the inverse of the "PUSH". The following sub instructions effectively sum up the "POP instruction.

```
mov     (%rsp), <quad data to be retrieved>
add     %rsp, $8
```

The decrement ("**DEC**") instruction simply subtracts 1 from the operand. Similarly the increment ("**INC**") instruction adds 1 to the operand.

```
#================================================================
# File:                 Sample7.s
# Assemble:             gcc -c Sample7.s
# Link:                 ld Sample7.o -o Sample7
# Run:                  ./Sample7
#================================================================


.global _start

.text

_start:
                        mov     $10, %rcx

START_LOOP:
                        jrcxz   EXIT
                        push    %rcx

                        mov      $1,    %rax
                        mov     $1,            %rdi
                        mov     $loopMsg,      %rsi
                        mov     $loopeMsgLen,  %rdx

                        syscall

                        pop            %rcx
                        dec            %rcx

                        jmp            START_LOOP

EXIT:
                        #exit(0)
                        mov    $60,    %rax
                        xor    %rdi,   %rdi
                        syscall

#=================================================================

.data
        CR                      =                       13
        LF                      =                       10

        loopMsg:                .ascii         "Looping..."
                                .byte          CR, LF
        loopeMsgLen             =              .-loopMsg


#=================================================================
```

3.0 – Loop Instructions

In the previous two sections we learnt two constructs in writing code - sequential processing and the ability to jump to addresses that are not necessarily in sequence. In this section we introduce a third construct in programming called looping.

Loop instructions allow the user to repeatedly execute a set of instructions until one or more conditions are met. In some respects a loop instruction is a special case of a Jump instruction, but the construct is powerful enough to warrant a dedicated section. Besides, most processors including the x86 processor, have specialized instructions for looping.

3.1 – Basic Loop (LOOP)

Our knowledge of the Jump if Not Zero (“JNZ”) and jump if Zero (“JZ”) instructions will allow us to implement a very basic loop.

The sample below is a slight modification of the sample in the previous section. There are two loops in this sample.

In the first loop I have replaced the “JCXZ” instruction with a JNZ. I am exploiting the fact that the “DEC” instruction sets the ZERO flag.

The second loop introduces the “**LOOP**” instruction. The “LOOP” instruction decrements the CX register by one and loops to the operand label if the CX register is not zero. Note that it does not alter the flags register.

```
#================================================================
# File:                 Sample8.s
# Assemble:             gcc -c Sample8.s
# Link:                 ld Sample8.o -o Sample8
# Run:                  ./Sample8
#================================================================


.global _start

.text

_start:
                        mov            $10, %rcx

START_LOOP1:

                        #This loop uses a "dec" and "jnz"
                        push    %rcx

                        mov     $1,                   %rax
                        mov     $1,                   %rdi
                        mov     $loop1Msg,            %rsi
                        mov     $loop1MsgLen,  %rdx

                        syscall

                        pop            %rcx
                        dec            %rcx
                        jnz            START_LOOP1

                        #We have come out of the first loop
                        #Reinitialize %rcx
                        mov            $10, %rcx

START_LOOP2:

                        #This loop replaces the "dec" and "jnz" with a "loop"
                        push    %rcx

                        mov     $1,                   %rax
                        mov     $1,                   %rdi
                        mov     $loop2Msg,            %rsi
                        mov     $loop2MsgLen,  %rdx

                        syscall

                        pop            %rcx
                        loop    START_LOOP2
```

```
EXIT:
                #exit(0)
                mov    $60,                  %rax
                xor    %rdi,                 %rdi
                syscall

#=================================================================

.data
        CR                              =                      13
        LF                              =                      10

        loop1Msg:             .ascii         "Loop 1..."
                                      .byte          CR, LF
        loop1MsgLen           =                      .-loop1Msg

        loop2Msg:             .ascii         "Loop 2..."
                                      .byte          CR, LF
        loop2MsgLen           =                      .-loop2Msg
#=================================================================
```

3.2 – Other Loops (LoopE,LoopZ,LoopNE,LoopNZ)

The Loop if Zero ("**LOOPZ**") or Loop if Equal ("**LOOPE**") instructions are similar to the "LOOP" instruction with one additional condition – they only loop if the ZERO flag is set.

The Loop if Not Zero ("LOOPNZ") or Loop if Not Equal ("LOOPNE") instructions are also similar to the "LOOP" instruction with the additional condition that they only loop if the ZERO flag is **not** set.

These instructions are handy in cases where the number of loops is not always a constant but rather based on a condition.

4.0 – Calling Procedures

In the previous sections we discussed Sequential, Jump and Loop instructions. The main benefits of the latter two programming constructs are that it allows a coder to reuse code that is written for generic purposes (eg. reading user input).

While a Jump instruction allows a coder to jump to any location within the code, it does not provide a mechanism to return to the location from which the Jump occurred once the generic code is executed. This brings us to the fourth programming construct – Procedures.

Procedure calling is designed specifically to remedy the problem of knowing where and in what state to return to, once the generic code is executed.

4.1 – Calling a Procedure and Returning

A Procedure is essentially a set of instructions to do a specific task. For example in most of the samples thus far we had a need to display an output. We would have been more effective to isolate that code to a procedure and call the procedure each time we needed to display an output, rather than duplicate the code each time.

In the sample below, I have written a procedure called “StrOut” to display a string. The caller is expected to pass in the offset to the string to be displayed in the “rax” register and the count of the number of characters in the string in the “rbx” register. These are referred to as the “input” parameters to the procedure. If the procedure was expected to return a value back to the caller, that would be referred to as an “output” parameter. By convention, output parameters are passed in the “rax” register.

The “**CALL**” instruction does 2 things – it saves the RIP register on the stack and jumps to the procedure.

The “**RET**” instruction pops the previously saved IP register.

If the procedure is going to save anything on the stack (and it almost always will), it is very important that it pops everything back before calling the “RET” instruction, else an invalid value will be popped into the IP register.

```
#================================================================
# File:                 Sample9.s
# Assemble:             gcc –g -c Sample9.s
# Link:                 ld Sample9.o -o Sample9
# Run:                  gdb ./Sample9 (use the ‘s’ step command to
#                       walk every line of code and inspect registers)
#================================================================


.global _start
.global _start

.text

_start:

               mov            $msgString1,   %rdi
               mov            $msgStringLen1,       %rsi
               call    StrOut

               mov     $msgString2,  %rsi
               mov     $msgStringLen2,       %rdx
               mov     $1,                   %rax
               mov     $1,                   %rdi
               syscall


EXIT:
                #exit(0)
                mov    $60,                  %rax
                xor    %rdi,                 %rdi
                syscall


 StrOut:

               #Prologue
               push    %rbp                  #save rbp
               mov            %rsp,   %rbp   #save rsp in rbp
               sub            $4,            %rsp    #save area in stack for locals
               push    %rsi                  #save registers we place to use
               push    %rdx
               push    %rax
               push    %rdi
               pushf                                #push flags register


               #Function code
               mov     %rsi,  %rdx
               mov     %rdi,  %rsi
               mov     $1,    %rax
               mov     $1,    %rdi

               syscall

               #Epilogue
               popf                         #pop flags register
               pop     %rdi                 #restore the previously stored registers
               pop     %rax
               pop     %rdx
               pop     %rsi
               mov     %rbp,  %rsp   #remove area for locals
               pop     %rbp                 #restore rbp
               ret

#================================================================

.data
       CR                     =                     13
       LF                     =                     10

       msgString1:            .ascii        "In a function call!"
                              .byte         CR, LF
       msgStringLen1  =       .-msgString1

       msgString2:            .ascii        "Out of function call!"
                              .byte         CR, LF
       msgStringLen2  =       .-msgString2

#================================================================
```

5.0 – Addressing Modes

Accessing data in registers and in memory is an essential part of assembler programming. Every processor allows for different methods to specify source and destination addresses for various instructions.

In this section we will cover the most common addressing modes used by the x64 processor.

5.1 – Register Addressing Mode

This involves accessing data in registers. It is a very common and straight forward technique and we have used it in almost all the samples thus far. The following is an example of Register Addressing mode.

```
mov     %rax, %rbx
```

Here we are moving the contents of the “RAX” register into the “RBX” register.

5.2 – Immediate Addressing Mode

When a data value is a constant, it can be made available as an operand. The following is an example of an Immediate Addressing Mode.

```
mov     $9, %rcx
```

This instruction moves “9” to the “RCX” register.

5.3 – Direct Addressing Mode

If we wanted to access memory at a known address, we could enclose the known address in parenthesis and offer that as our operand. The following is an example of direct addressing.

```
mov     (0x4000b0), %rax
```

Here we move the contents at memory address 0x4000b0 into the RAX register.

5.4 – Register Indirect Addressing Mode

Sometimes a register contains an address of a memory location. In these cases we can access the value at that address by enclosing the register in parenthesis and use that as our operand.

```
mov             $0x4000b0, %rax
mov             (%rax), %rbx
```

Here we move the contents of memory whose address is in the “RAX” register to the “RBX” register.

5.5 – Register Indirect Indexed Addressing Mode

The register indirect addressing mode can be extended to access elements of an array for example, by using the following syntax;

```
mov             $0x4000b0, %rax
mov             0x10(%rax), %rbx
```

This will fetch the contents of memory at the address defined by the “RAX” register plus 0x10 and move it to RBX.

Other variations include the following:

```
mov     (%rax,%rbx), %rcx               ➔ Mem[%rax+%rbx] -> %rcx
mov     (%rax, %rbx, 5), %rcx           ➔ Mem[%rax + 5*%rbx] -> %rcx
mov    0x10(%rax, %rbx, 5), %rcx          ➔ Mem[%rax + 5*%rbx + 40]  -> %rcx
```

The most general form is as follows:

```
Displacement(baseRegister, indexRegister, scale)
```

➔

```
Mem [baseRegister + (Scale * indexRegister) + Displacement]
```

6.0 – More complex Instructions

In this last section on assembler programming, we will study a few more complex instructions available to us in the x64 processor.

6.1 – Bit Operations

<u>OR</u>

The “OR” instruction is used to turn on individual bits in the destination operand based on a mask that is provided as the source operand.

```
or      $0x0f, %rax
```

In the example above, we ensure that the lowest 4 bits in the RAX register are set.

<u>AND</u>

The “AND” instruction is used to turn off individual bits in the destination operand based on a mask that is provided as the source operand.

```
and     $0x0f, %rax
```

In the example above, only the lowest 4 bits of the RAX register are preserved. All other bits are set to zero.

<u>XOR</u>

The Exclusive OR operation inverts all the bits in the destination operand, for which the corresponding mask bits are set.

```
xor     $0x0f, %rax
```

<u>TEST</u>

The “TEST” instruction is very similar to the “AND” operation with the exception that it does not alter the destination operand. Instead it only sets the flags register to indicate if the operation resulted in a zero or non-zero result. This can then be used by the JUMP instructions. The “TEST” is the BOOLEAN equivalent of a “CMP” instruction.

<u>SHL</u>

The Shift Left (“SHL”) instruction shifts the bits in the destination operand to the left. The number of bits to be shifted is provided as the source operand. During each 1 bit left shift, the bit shifted out of the most significant bit is placed in the Carry bit of the flags register and a bit ‘0” is shifted into the least significant bit.

Note that shifting left by 1 bit is the equivalent of multiplying by 0x02.

<u>SHR</u>

The Shift Right (“SHR”) instruction shifts the bits in the destination operand to the right. The number of bits to be shifted is provided as the source operand. During each 1 bit right shift, the bit shifted out of the least significant bit is placed in the Carry bit of the flags register and a bit “0” is shifted into the most significant bit.

Note that shifting right by 1 bit is the equivalent of unsigned division by 0x02.

<u>SAR</u>

The Shift Arithmetic Right (“SAR”) is similar to the “SHR” instruction with the exception that that the most significant bit will always remain unchanged. Since the most significant bit is the sign bit, this instruction can be useful for signed division.

<u>ROL</u>

The Rotate Left ("ROL") instruction rotates the bits in the destination operand in the counter clockwise direction. The number of bit shifts is defined by the source operand. During each shift to the left, the most significant bit defines the value of the Carry flag.

ROR

The Rotate Right ("ROR") instruction rotates the bits in the destination operand in the clockwise direction. The number of bit shifts is defined by the source operand. During each shift to the right, the least significant bit defines the value of the Carry flag.

RCL

The Rotate Left Through Carry ("RCR") is similar to the ROL instruction with the exception that the bits are rotated counter clockwise through the Carry Flag.

RCR

The Rotate Right Through Carry ("RCR") is similar to the ROR instruction with the exception that the bits are rotated clockwise through the Carry Flag.

6.2 – Arithmetic Operations

ADD

The Add instruction adds the source and destination operands and places the result in the destination operand.

ADC

The Add with Carry (“ADC”) instruction adds the source, destination and the “Carry Flag” bit and places the result in the destination operand.

SUB

The Subtract (“SUB”) instruction subtracts the source from the destination and places the result in the destination operand.

SBB

The Subtract with Borrow (“SBB”) subtracts the sum of the source and the “Carry flag” from the destination and places the result in the destination operand.

CBW

The Convert Byte to Word (“CBW”) instruction uses the input BYTE in the AL register and converts it to a WORD in the AX register.

This is a useful instruction for converting both signed and unsigned BYTEs to WORDs. It effectively performs a sign extension by ensuring that the most significant bit in the BYTE is replicated in all the extra bits provided by the WORD.

CWD

The Convert Word to Double (“CWD”) instruction is very similar to the “CBW” instruction. It takes the AX register as the input WORD and converts it to a DWORD in DX and AX registers where DX has the high WORD and AX has the lower WORD.

STC, CLC and CMC

Sometimes you may want to directly control the value of the Carry flag when performing arithmetic operations.

The Set Carry (“STC”) allows you set the Carry flag.

The Clear Carry (“CLC”) allows you to clear the Carry flag.
The Complement Carry (“CMC”) allows you to inverts the current value of the Carry flag.

LEA

The Load Effective Address (“LEA”) instruction allows you to find the address offset of a memory variable. We used this in a number of our previous samples to find the address of our input buffer.

To use the “OFFSET” directive however, the address should be available at assembly time. If the address is only available at runtime, the “LEA” instruction must be used. In the example below, the contents of BX is added to the contents of SI and that result is added to 7 to give the offset that is placed in the AX register. Since the contents of BX and SI are only available at runtime, the “LEA” instruction must be used.

While the “LEA” instruction is generally used for address calculation, there is no reason why it can’t be used for arithmetic operations.

6.3 – Interrupt Operations

STI

The Set Interrupt (“STI”) instruction enables interrupts by setting the Interrupt Flag (IF) bit in the flags register.

CLI

The Clear Interrupt (“CLI”) instruction disables interrupts by clearing the Interrupt Flag (IF) bit in the flags register.

6.4 – String Operations

MOVSB

The Move Single Byte ("MOVSB") instruction is a complex instruction that allows you to move a byte from a source address to a destination address. Effectively it does the following:

```
mov      (%si), %al       ; copySI to al
mov      %al, (%di)       ; move al to DI
inc/dec  %rsi             ; increment or decrement SI based on Direction flag
inc/dec  %rdi             ; increment or decrement DI based on Direction flag
```

As you might guess, before using the "MOVSB" instruction, you need to ensure that ES, SI, DI and the Direction flag is set up correctly.

Note: If the Direction Flag is zero, RDI and RSI are incremented, else they are decremented.

STD and CLD

As noted in the "MOVSB" instruction, the value of the direction flag dictates if SI and DI are incremented or decremented by the "MOVSB" instruction.

The Set Direction ("STD") instruction sets the Direction flag.

The Clear Direction ("CLD") clears the Direction flag.

MOVSW

The "MOVSW" instruction is very similar to the "MOVESB" instruction but works on a WORD instead of a BYTE.

REP MOVSB

The Repeat MOVSB ("REP MOVSB") instruction allows you to repeat the move instruction for a number of iterations defined by the CX register. Think of it as a loop instruction that decrements the CX register in each iteration and jumps out of the loop when CX hits zero.

REP MOVSW

Similar to "REP MOVSB", but works on WORDs instead of BYTEs.

STOSB and **STOSW**

The Store String by Byte ("STOSB") and Store String by Word ("STOSW") allow you to store the value in a register into a memory address. Effectively, these instructions do half the work done by the MOVSB/MOVSW instructions – they move data from register to memory instead of memory to memory.

The STOSB instruction saves the byte in the AL register to the memory address defined by ES:[DI] and then increments or decrements the DI register by **one**, based on the Direction Flag.

The STOSW instruction saves the word in the AX register to the memory address defined by ES:[DI] and then increments or decrements the DI register by **two**, based on the Direction Flag.

REP STOSB and **REP STOSW**

The Repeat ("REP") qualifier can also be used with the STOSB and STOSW instructions to repeat the instructions by a count defined by the RCX register.

LODSB and **LODSW**

The Load String Byte ("LODSB") and Load String Word ("LODSW") allow you to load the value in a memory location into a register. These instructions do the other half of the work done by the MOVSB/MOVSW instructions.

The LODSB instruction loads a byte from DS:[SI] to the AL register and then increments or decrements the DI register by **one**, based on the Direction Flag.

The LODSW instruction loads a word from the DS:[SI] to the AX register and then increments or decrements the DI register by **two**, based on the Direction Flag.

CMPSB

The Compare String Byte ("CMPSB") instruction compares two memory byte locations defined by DS:[SI] and ES:[DI]. It then sets the flags to indicate the result of the comparison and then increments or decrements the SI and DI registers, depending on the Direction Flag.

REPE and REPNE

The Repeat while Equal ("REPE") and Repeat while Not Equal ("REPNE") can be used in conjunction with the "CMPSB" instruction to compare a number of bytes defined by the CX register.

SCASB

The Scan String for Byte ("SCASB"), compares the BYTE in the AL register with the BYTE in ES:[DI] and sets the flags to reflect this comparison. It then increments or decrements the DI register, depending on the Direction Flag.

The REPE and REPNE qualifiers can also be used with "SCASB".

7.0 – Conclusion

Every programming language has a set of constructs and a set of instructions. In the x64 assembler programming language we studied the following five constructs:

- Sequential processing
- Jumps
- Loops
- Procedure calls
- Addressing Modes

Along the way, we also came across several x64 instructions that allow us to do bit manipulations, arithmetic operations and string operations.

The constructs of a programming language are the equivalent of grammar, and instructions are the equivalent of vocabulary, in a spoken language. As with spoken languages, fluency in constructs and instructions come with continued use of the language. Hence it is recommended that the student practice the concepts discussed in this set of notes by writing many more applications.