COEN311 Lab 0

Ted Obuchowicz

January 8, 2021

1 Objectives

- Learn how to connect to a ENCS Linux server.
- Learn the rudiments of Linux (create a directory, `cd` into a directory, edit a text file with `nano`).
- Become familiar with Zoom videoconferencing software.

2 Procedure

2.1 Connect to an ENCS Linux Server

2.1.1 Linux and macOS

The ssh command is used to connect your home computer to a university Linux server. There are several such as:

- `login.encs.concordia.ca`
- `computation.encs.concordia.ca`
- A few other which may be restricted access

The server called `login` is meant for short duration sessions, the `computation` server is meant to be used for long sessions which require significant amount of compute time. It is recommended that students login to the server `computation.encs.concordia.ca` when performing online labs as this will allow for automatic load balancing. The following is the command to issue from your terminal prompt :

```
[ted@localhost ~]$    ssh -Y login.encs.concordia.ca -l ted
```

There are several things to note here:

- `[ted@localhost  ]$`
- This is my local prompt, yours may appear different depending on how your account has been setup by our system administrators. It is NOT part of the command. In Linux, commands are entered after the prompt (and then pressing the Enter key).
- `computation.encs.concordia.ca`

  This address specifies the fully quantified domain name of the remote host you wish to connect to
- `-l user_name`

  This parameter is used to specify the login name on the remote server. Otherwise, the default is the login name on your home computer (which may be differrent from your ENCS/GCS login name).

After pressing the Enter key, you will see be prompted to enter your ENCS password:

```
ted@computation.encs.concordia.ca's password: (enter your password)
```

The following will be displayed:

```
Last login: Thu Jun  4 11:59:16 2020 from some_ip_address
================================================================================
Gina Cody School of Engineering and Computer Science, Concordia University

             Unauthorized access is strictly forbidden.

For assistance: e-mail: servicedesk@encs.concordia.ca
For information:   web: https://www.concordia.ca/ginacody/

================================================================================
   ------------------------------------------------------------------------
   Wed May 20 2020

   We will soon be updating the ssh server software on our systems
   in order to keep up with security fixes, and as a result the host
   keys will change.

   Your ssh client software may advise you that the system you're
   connecting to isn't the same as the system you meant to connect to.
   In that case, you'll need to accept the new key presented by our
   system before the software will continue connecting.  You will
   need to do this for each of the systems that you connect to, though
   there is no deadline for doing it for any system.  It will simply
   need to be done the next time you connect to a system after its
   host key has changed.

   If you encounter issues, or have any questions or concerns about
   this upcoming change, please send an email to help@concordia.ca
   and we will do our best to assist.
   ------------------------------------------------------------------------


ted@poise ~ 5:40pm >
```

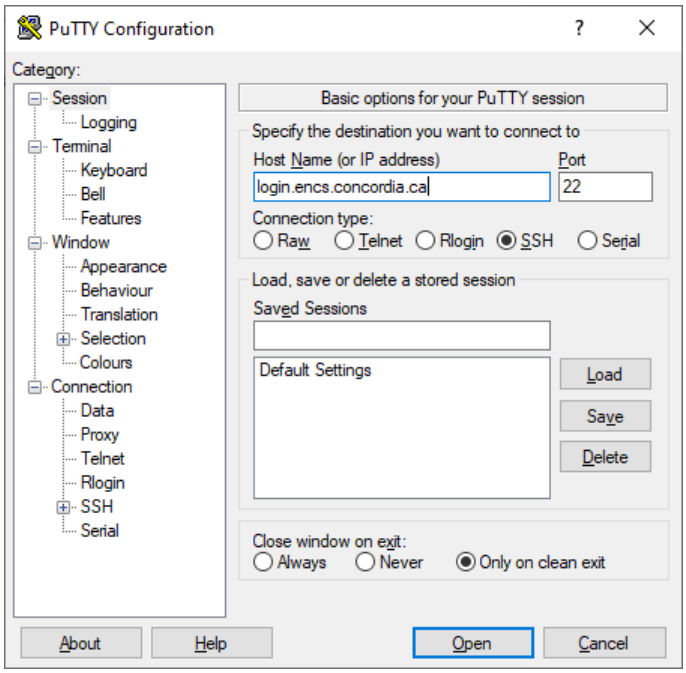
You are now logged into the ENCS Linux server and can issue commands and run programs from the command line.

2.2 Windows

The Windows program puTTY is a free SSH and Telnet client program, available at `https://putty.org` . It lets one remotely connect to another computer, run

applications on the remote computer, and have the results displayed on your local (home) Windows computer. To use Putty to establish a simple command line terminal session, download and install puTTY. Once launched, you will be presented with the configuration window.

Specify the destination you want to connect to as `computation.encs.concordia.ca`, the connection type as **SSH** and the port as **22**.



Once connecting, you may receive a message stating: "The server's host key is not cached in the registry. You will have ..."

Simply select Yes to this standard security alert message. A terminal window will appear and you will be prompted to enter your ENCS login name, followed by your password. You are now logged into the ENCS Linux computer with your ENCS username.

3 In the Terminal

Once your connection has been estatblished, the environment for the lab must be prepared. It is strongly recommended that you create a directory called COEN311 to hold all your COEN 311 related files, and a directory called NASM within your COEN 311 directory to hold all your assembly language programs. The following Linux commands are used to do this:

```
mkdir COEN311
cd COEN311
```

```
mkdir NASM
cd NASM
```

4 Using a Linux Command Line Text Editor

For those unfamiliar with Linux and the available text editors, here is a very short guide to using the `nano` Linux text editor.

Any Linux command line text editor can be used such as `vi`, `emacs`, `nano` or `pico`. This example gives the basics of using the `nano` text editor as it is easy and user-friendly:

```
ted@willpower NASM 6:04pm >nano junk.txt
```

After you press Enter, you will see on your terminal:

```
  GNU nano 2.3.1               File: junk.txt
```

Start to type your code here. blah blah blah.. you can use the *up*, *down*, *left* and *right* arrows to move around in the file and use the *Backspace* and *Delete* keys.

```
                                  [ New File ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

To save the file, press the *Ctrl* and *O* keys simultaneously. The name of the file (`junk.txt`) will be the default file name, you can specify another name if your wish and the contents of the file will be saved to a file residing in the directory from where you invoked the nano text editor from. It is wise to save often, or else you may accidentally lose what you typed in.

To exit from the text editor (after you have saved your file), press the *Ctrl* and *X* keys simultaneously (this keystroke combination is shortened to *Ctrl-X*). Save the file with a name of your choice with a `.asm` filename extension. For example: `add_2_mumbers.asm` or `lab1.asm`.

5 Use a Linux Text Editor to Create an ASCII Text File

Once comfortable with using `nano`, the next step is to create a text file in the `NASM` directory containing:

```
; Ted Obuchowicz  (replace with your name, this is a comment line)
; June 3, 2020
; sample program to add two numbers

section .data

; put your data in this section using
; db , dw, dd directions
; this program has nothing in the .data section

section .bss

; put UNINITIALIZED data here using
; this program has nothing in the .bss section


section .text
        global _start

_start:
        mov ax,5  ; store 5 into the ax register
keith:  mov bx,2  ; store 2 into the bx register
        add ax,bx ; ax = ax + bx
                  ; contents of register bx is added to the
                  ; original contents of register ax and the
                  ; result is stored in register ax (overwriting
                  ; the original content


        mov eax,1             ; The system call for exit (sys_exit)
        mov ebx,0             ; Exit with return code of 0 (no error)
        int 80h
; end of program
```

Save the file with a name of your choice with a `.asm` filename extension. For example: `add_2_mumbers.asm` or `lab1.asm`.

COEN311 Lab 1

Ted Obuchowicz

January 8, 2021

1 Objectives

- To review the method of connecting to an ENCS Linux server.
- Review of basic Linux commands and text editing with `nano`.

  The above were covered in Lab 0, the steps are included in this lab for completeness and convenience
- To gain familiarity with using the `nasm` assembler and the `ld` loader to generate an executable program from Intel x86 assembly language source. code and
- To learn to single-step through the program using the `gdb` command line debugger.

2 Connect to an ENCS Linux Server

2.1 Linux and macOS

The ssh command is used to connect your home computer to a university Linux server. There are several such as

- `login.encs.concordia.ca`
- `computation.encs.concordia.ca`
- A few other which may be restricted access

The server called `login` is meant for short duration sessions, the `computation` server is meant to be used for long sessions which require significant amount of compute time. The following is the command to issue from your terminal prompt :

```
[ted@localhost ~]$    ssh -Y computation.encs.concordia.ca -l ted
```

There are several things to note here:

- `[ted@localhost  ]$`
- This is my local prompt, yours may appear different depending on how your account has been setup by our system administrators. It is NOT part of the command. In Linux, commands are entered after the prompt (and then pressing the Enter key).
- `computation.encs.concordia.ca`

  This address specifies the fully quantified domain name of the remote host you wish to connect to

- `-l user_name`

  This parameter is used to specify the login name on the remote server. Otherwise, the default is the login name on your home computer (which may be different from your ENCS/GCS login name).

After pressing the Enter key, you will see be prompted to enter your ENCS password:

```
ted@computation.encs.concordia.ca's password: (enter your password)
```

The following will be displayed:

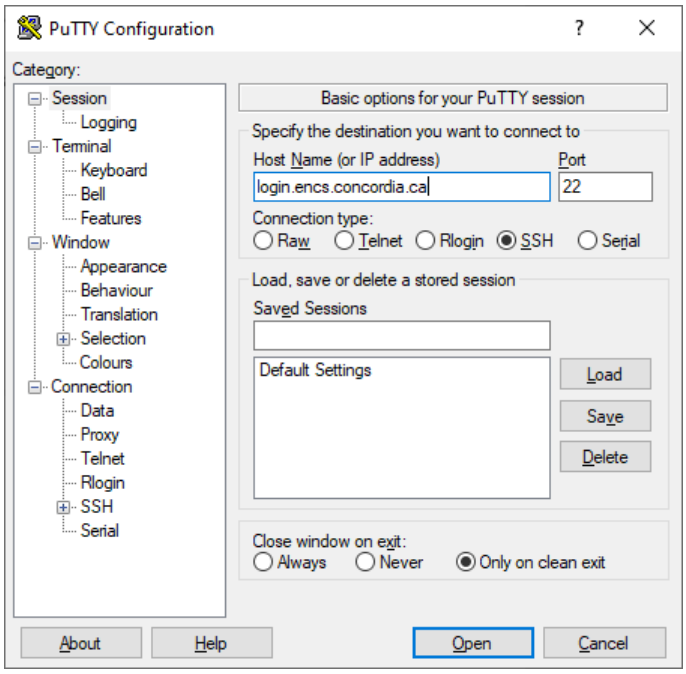
```
Last login: Thu Jun  4 11:59:16 2020 from some_ip_address
================================================================================
Gina Cody School of Engineering and Computer Science, Concordia University

             Unauthorized access is strictly forbidden.

For assistance: e-mail: servicedesk@encs.concordia.ca
For information:   web: https://www.concordia.ca/ginacody/

================================================================================
   ------------------------------------------------------------------
   Wed May 20 2020

   We will soon be updating the ssh server software on our systems
   in order to keep up with security fixes, and as a result the host
   keys will change.

   Your ssh client software may advise you that the system you're
   connecting to isn't the same as the system you meant to connect to.
   In that case, you'll need to accept the new key presented by our
   system before the software will continue connecting.  You will
   need to do this for each of the systems that you connect to, though
   there is no deadline for doing it for any system.  It will simply
   need to be done the next time you connect to a system after its
   host key has changed.

   If you encounter issues, or have any questions or concerns about
   this upcoming change, please send an email to help@concordia.ca
   and we will do our best to assist.
   ------------------------------------------------------------------


ted@poise ~ 5:40pm >
```

You are now logged into the ENCS Linux server and can issue commands and run programs from the command line.

2.2 Windows

The Windows program puTTY is a free SSH and Telnet client program, available at `https://putty.org` . It lets one remotely connect to another computer, run applications on the remote computer, and have the results displayed on your local (home) Windows computer. To use Putty to establish a simple command line terminal session, download and install puTTY. Once launched, you will be presented with the configuration window.

Specify the location you want to connect to as `login.encs.concordia.ca`, the connection type as **SSH** and the port as **22**.



Once connecting, you may receive a message stating: "The server's host key is not cached in the registry. You will have ..."

Simply select Yes to this standard security alert message. A terminal window will appear and you will be prompted to enter your ENCS login name, followed by your password. You are now logged into the ENCS Linux computer with your ENCS username.

3 In the Terminal

Once your connection has been estatblished, the environment for the lab must be made. It is strongly recommended that you create a directory called COEN311 to hold all your COEN 311 related files, and a directory called NASM within your COEN 311 directory to hold all your assembly language programs. The following Linux commands are used to do this:

```
mkdir COEN311
cd COEN311
mkdir NASM
cd NASM
```

4 Using a Linux Command Line Text Editor

For those unfamiliar with Linux and the available text editors, here is a very short guide to using the `nano` Linux text editor.

Any Linux text command line text editor can be used such as `vi`, `emacs`, `nano` or `pico`. This example gives the basics of using the `nano` text editor as it is easy and user-friendly:

```
ted@willpower NASM 6:04pm >nano junk.txt
```

After you press Enter, you will see on your terminal:

```
  GNU nano 2.3.1               File: junk.txt
```

Start to type your code here. blah blah blah.. you can use the *up*, *down*, *left* and *right* arrows to move around in the file and use the *Backspace* and *Delete* keys.

```
                                  [ New File ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

To save the file, press the *Ctrl* and *O* keys simultaneously. The name of the file (`junk.txt`) will be the default file name, you can specify another name if your wish and the contents of the file will be saved a file residing in the directory from where you invoked the nano text editor from. It is wise to save often, or else you may accidentally lose what you typed in.

To exit from the text editor (after you have saved your file), press the *Ctrl* and *X* keys simultaneously (this keystroke combination is shortened to *Ctrl-X*). Save the file with a name of your choice with a `.asm` filename extension. For example: `add_2_mumbers.asm` or `lab1.asm`.

5 Use a Linux Text Editor to Create an ASCII Text File

Once comfortable with using `nano`, the next step is to create a text file in the `NASM` directory containing:

```
; Ted Obuchowicz  (replace with your name, this is a comment line)
; June 3, 2020
; sample program to add two numbers

section .data

; put your data in this section using
; db , dw, dd directions
; this program has nothing in the .data section

section .bss

; put UNINITIALIZED data here using
; this program has nothing in the .bss section


section .text
        global _start

_start:
        mov ax,5  ; store 5 into the ax register
keith:  mov bx,2  ; store 2 into the bx register
        add ax,bx ; ax = ax + bx
                  ; contents of register bx is added to the
                  ; original contents of register ax and the
                  ; result is stored in register ax (overwriting
                  ; the original content


        mov eax,1             ; The system call for exit (sys_exit)
        mov ebx,0             ; Exit with return code of 0 (no error)
        int 80h
; end of program
```

Save the file with a name of your choice with a `.asm` filename extension. For example: `add_2_mumbers.asm` or `lab1.asm`.

6 Assemble the File

Assemble the file using the `nasm` assembler. In the examples which follow, the portion:

```
ted@willpower NASM 11:57am >
```

Will be referred to as the (Linux) command prompt. Your command prompt may appear slightly different depending on how your account has been set up by our system administrators.

6.1 Verify That the Assembler Is Installed

Before starting, it must be verified that the assembler is installed on the system which you have logged into by using the `which` command:

```
ted@willpower NASM 11:58am >which nasm
/encs/bin/nasm
ted@willpower NASM 12:01pm >
```

You will note that the system responds by displaying the path to where the nasm program resides, and you are then returned to the command prompt.

6.2 Verify the Version Number of NASM

This is optional but one can verify the version number of `nasm` with the following command:

```
ted@willpower NASM 12:01pm >nasm -version
NASM version 2.14.02 compiled on May 29 2019
ted@willpower NASM 12:03pm >
```

6.3 Assemble the source code using the nasm assembler

```
ted@willpower NASM 12:03pm >nasm -f elf add_2_numbers.asm -l add_2_numbers.lis

ted@willpower 12:03pm >
```

The `-f elf` option to the assembler is used to specify that the output file format (the default output file name is obtained by prepending the input file name with a `.o` filename extension, in this case the output filename would be `add_2_numbers.o`) should be a Linux "executable and linkable format". Many other output file formats exist.

The `-l` (the letter 'l') option is used to specify that a listing file with specified name is to be produded. A listing file is an ASCII text file containing the machine code of the program on the left hand side, and the original source code on the right hand side.

If your assembly language source code contains no errors, the `.o` and `.lis` files will be written in the current directory. If your source code contains syntax errors, they will be shown in the terminal window:

```
ted@willpower NASM 12:22pm >nasm -f elf bad.asm -l bad.lis
bad.asm:21: error: parser: instruction expected
```

The number 21 refers to the line number which caused the error to be generated. No listing and no output file is generated if the program does not assemble. If your assembly language source code contains errors, it is necessary to edit the file, correct the errors and then reassemble the source code.

6.4 Using Linux Commands to List Directory Contents

We will now present some useful Linux commands. The `ls` command is used to list the contents of a directory (what Windows calls a folder). Type the following from your command prompt:

```
ted@willpower NASM 12:25pm >ls -al add*
-rw------- 1 ted ted  855 Jun  3 14:36 add_2_numbers.asm
-rw------- 1 ted ted 2175 Jun  5 12:20 add_2_numbers.lis
-rw------- 1 ted ted  576 Jun  5 12:20 add_2_numbers.o
```

The `*` character at the end of the string `add` is the Linux wildcard character. Its use in the given command will match any filename which resides in the directory starting with the letters `add`. Contrast this with the the :

```
ted@willpower NASM 12:29pm >ls -al
total 52
drwx------  2 ted ted  4096 Jun  5 12:25 .
drwx------ 19 ted ted  4096 Jun  3 14:17 ..
-rw-------  1 ted ted   149 Jun  5 11:51 80.txt
-rw-------  1 ted ted   855 Jun  3 14:36 add_2_numbers.asm
-rw-------  1 ted ted  2175 Jun  5 12:20 add_2_numbers.lis
-rw-------  1 ted ted   576 Jun  5 12:20 add_2_numbers.o
-rw-------  1 ted ted   856 Jun  5 12:21 bad.asm
-rw-------  1 ted ted  3702 Jun  5 12:25 lab1.txt
-rw-------  1 ted ted 16384 Jun  5 12:32 .lab1.txt.swp
-rw-------  1 ted ted   602 Jun  3 14:31 template.asm
```

Your listing may appear different than the provided one depending on the files in the directory.

The Linux `file` command is used to determine the type of the specified file:

```
ted@willpower NASM 12:32pm >file add_2_numbers.asm
add_2_numbers.asm: ASCII text


ted@willpower NASM 12:33pm >file add_2_numbers.o
add_2_numbers.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV),
not stripped
```

The Linux 'more' command is used to view the contents of an ASCII text file on the screen, one screenful at a time (and press the space bar to display the next screenful). Use the `more` command to view the contents of your `.lis` file:

```
ted@willpower NASM 12:34pm >more add_2_numbers.lis
     1                                  ; Ted Obuchowicz
     2                                  ; June 3, 2020
     3                                  ; sample program to add two numbers
     4
     5                                  section .data
     6
     7                                  ; put your data in this section using
     8                                  ; db , dw, dd directions
     9                                  ; this program has nothing in the .data
section
    10
    11                                  section .bss
    12
    13                                  ; put UNINITIALIZED data here using
    14                                  ; this program has nothing in the .bss s
ection
    15
 15
    16
    17                                  section .text
    18                                          global _start
    19
    20                                  _start:
    21 00000000 66B80500                        mov ax,5  ; store 5 into the ax
register
    22 00000004 66BB0200                keith:  mov bx,2  ; store 2 into the bx
register
    23 00000008 6601D8                          add ax,bx ; ax = ax + bx

(rest of the .lis file is intentionally not shown)
```

7 Create An Executable Program

The next step is to create an executable program from the `.o` (object file) with the `ld` command:

The `ld` command "combines a number of object and archive files, relocates their data and ties up symbol references. Usually the last in compiling a program is to run `ld`".

If you are wondering how this was found, try the following from the command prompt:

```
ted@willpower NASM 12:44pm >man ld
```

Linux contains a command called `man` (short for manual) which displays the instructions for any Linux command. Try listing (and reading) the man pages for the `ls` and `more` commands.

The command to run the linker is:

```
ted@willpower NASM 12:44pm >ld  -melf_i386  -o add_2_numbers add_2_numbers.o
```

Let us examine each part of the command line and its options and input files and output files:

- -melf_i386

  This option specifies the emulation linker to be used. For our environment, the elf_i386 is the required version.

- -o add_2_numbers

  The -o option is used to specifiy the name of the executable output file. In the absence of specifying an output file name, the default output file name is `a.out` (which stands for 'assembler output')

- add_2_numbers.o

  The input file to the `ld` command. Recall that the `.o` file was created by assembling your `.asm` file.

Let's run the `file` command on the `add_2_numbers` file created by the `ld` command:

```
ted@willpower NASM 12:52pm >file add_2_numbers
add_2_numbers: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
statically linked, not stripped
```

It is a complete executable program ready to be run. In short, all the above steps may be summarized in two lines:

```
ted@willpower NASM 12:58pm >nasm -f elf add_2_numbers.asm -l add_2_numbers.lis
ted@willpower NASM 12:58pm >ld -melf_i386 -o add_2_numbers add_2_numbers.o
```

8 Running The program

We can run any executable program by typing its name on the Linux command prompt. The program will run and when it has reached the end of its exectiion, control returns to the user's command prompt.

```
ted@willpower NASM 12:58pm >add_2_numbers
ted@willpower NASM DING! >
```

You may be wondering "Where is the output of this program? Why did the program not do anything?". The assembly language program produced no output since it did not contain any specific instructions to display anything on the screen. The program merely loaded two CPU registers with some constant values (5 and 2), and then added these two registers and saved the result in one of the registers.

When working in assembly language, one typically makes use of a debugger. A *debugger* is a program which runs an executable program one instruction at a time, and allows the user to display the contents of CPU registers, memory locations and other information instruction by instruction. This is known as single-stepping through a program. The man page for `gdb` succintly summarizes what `gdb` is:

```
DESCRIPTION
       The purpose of a debugger such as GDB is to allow you to see what is
       going on "inside" another program while it executes -- or what another
       program was doing at the moment it crashed.
       GDB can do four main kinds of things (plus other things in support of
       these) to help you catch bugs in the act:

       ·   Start your program, specifying anything that might affect its
           behavior.

       ·   Make your program stop on specified conditions.

       ·   Examine what has happened, when your program has stopped.

       ·   Change things in your program, so you can experiment with
           correcting the effects of one bug and go on to learn about another.

       You can use GDB to debug programs written in C, C++, Fortran and
       Modula-2.

       GDB is invoked with the shell command "gdb".  Once started, it reads
       commands from the terminal until you tell it to exit with the GDB
       command "quit".  You can get online help from GDB itself by using the
       command "help"."
```

The `gdb` debugger utility will be used to single step through the executable program:

```
ted@willpower NASM 1:09pm >gdb add_2_numbers
GNU gdb (GDB) 7.7
Copyright (C) 2014 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-unknown-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 add_2_numbers...(no debugging symbols found)...done.
(gdb) break keith
Breakpoint 1 at 0x8048064
```

The `break` command is used to set a breakpoint in the program. In this case, we set a breakpoint on the **second** line of code in the assembly language program specifying the label `keith` in the `break` command. Recall that we can given any line in our assembly language program a label. Specifying a breakpoint within a debugging environment is one use of an instruction label.

`gdb` has the quirk that a program must be run before any register contents can be displayed. Thus, we set a breakpoint at the second instruction and use the `run` command to run the program. Program execution will stop at the specified breakpoint. We can then use `gdb` to examine registers and continue execution of the next instruction.

```
(gdb) run
Starting program: /nfs/home/t/ted/COEN311/NASM/add_2_numbers

Breakpoint 1, 0x08048064 in keith ()
```

The program has ran and stopped at the instruction stored in main memory address `0x08048064` (hex value). Let us examine the contents of the CPU registers with the `info registers` command:

```
(gdb) info registers
eax            0x5      5
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xffffd100       0xffffd100
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048064        0x8048064 <keith>
eflags         0x202    [ IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
```

```
es             0x2b     43
fs             0x0      0
gs             0x0      0
```

Note that the `EAX` register contains the number 5, and that also the `EIP` (Extended Instruction Pointer, also commonly called the *Program Counter*) contains the number `0x8048064`. The `EIP` contains the address of the next instruction to be executed.

`gdb` can disassemble the program it is running. Disassembly is the process of converting binary machine code into human readable assembly language instructions. The gdb command to disassemble is `disassemble`. Addresses in hex are specified on the left-hand side and instructions are listed on the right:

```
(gdb) disassemble
Dump of assembler code for function keith:
=> 0x08048064 <+0>:     mov    $0x2,%bx
   0x08048068 <+4>:     add    %bx,%ax
   0x0804806b <+7>:     mov    $0x1,%eax
   0x08048070 <+12>:    mov    $0x0,%ebx
   0x08048075 <+17>:    int    $0x80
End of assembler dump.
(gdb)
```

`disassemble`, by default, disassembles the next few instructions starting with the instruction pointed to by the instruction pointer register. The more astute reader will note that there are differences between the disassembled instructions genereated by `gdb` and the assembly language instructions originally written in the `.asm` file. For example, we typed in:

```
keith:  mov bx,2  ; store 2 into the bx register
        add ax,bx ; ax = ax + bx
```

But `gdb` disassembled these as:

```
0x08048064 <+0>:     mov    $0x2,%bx
0x08048068 <+4>:     add    %bx,%ax
```

We can use the `set_disassembly_flavor_intel` instruction to tell `gdb` to use *Intel x86* style syntax when disassembling rather than the non-Intel default. This command must be issued before the `run` command in order to have any effect. In what follows, `gdb` was quit from, then restarted and issued the following commands:

```
(gdb) set disassembly-flavor intel
(gdb) break keith
(gdb) run
(gdb) disassemble
Dump of assembler code for function keith:
```

```
=> 0x08048064 <+0>:     mov    bx,0x2
   0x08048068 <+4>:     add    ax,bx
   0x0804806b <+7>:     mov    eax,0x1
   0x08048070 <+12>:    mov    ebx,0x0
   0x08048075 <+17>:    int    0x80
End of assembler dump.
```

Let us use the `nexti` (can be also entered as `ni`) instruction to single step and execute the instruction pointed to by `EIP` at address `0x08048064`. Before doing so, we use the `print` command to display the contents of register `BX` before the `mov bx, 2` instruction has been executed:

```
(gdb) print/x $bx
$1 = 0x0
```

In `gdb`, `$register_name` refers to the contents of the specified register. The /X is a format specifier, indicating hexadecimal output is to be used when displaying the value. We note that register `BX` has a value of `0`. Following this, execute the next instruction (using `ni`), disassemble and then display the contents of register `BX`:

```
(gdb) ni
0x08048068 in keith ()
(gdb) disassemble
Dump of assembler code for function keith:
   0x08048064 <+0>:     mov    bx,0x2
=> 0x08048068 <+4>:     add    ax,bx
   0x0804806b <+7>:     mov    eax,0x1
   0x08048070 <+12>:    mov    ebx,0x0
   0x08048075 <+17>:    int    0x80
End of assembler dump.
(gdb) print/x $bx
$2 = 0x2
(gdb)
```

We can continue in this manner to single step through the remaining instructions of the program, examining registers and performing other operations.

```
(gdb) ni
0x0804806b in keith ()
(gdb) disassemble
Dump of assembler code for function keith:
   0x08048064 <+0>:     mov    bx,0x2
   0x08048068 <+4>:     add    ax,bx
=> 0x0804806b <+7>:     mov    eax,0x1
   0x08048070 <+12>:    mov    ebx,0x0
   0x08048075 <+17>:    int    0x80
End of assembler dump.
```

```
(gdb) print $ax
$4 = 7
```

Generally, one does not need to single step through the last three instructions of an Intel x86 program:

```
mov    eax,0x1
mov    ebx,0x0
int    0x80
```

Since they are meant to be executed when a program is ran from a UNIX-style command prompt (such as a Linux shell) and not when single-stepping with `gdb`. The `quit` command is used to quit from `gdb` and return to the Linux command prompt:

```
(gdb) quit
A debugging session is active.

        Inferior 1 [process 21396] will be killed.

Quit anyway? (y or n) y
ted@willpower NASM 2:12pm >
```

9 Questions

1. Explain the differences between an assembly language source code file, the listing file produced by `nasm`, the object file (`.o`) and the final executable program.

10 Required Deliverables

Submit together with your lab report the following:

- The `.asm` text file containing the Intel x86 assembly language program which adds the contents of two registers.
- The corresponding listing file
- A screenshot (or ASCII copy and paste from your terminal window) of your `gdb` debugging session with the contents of the destination register after the `add` instruction has been executed.
- The answers to the questions.

COEN311 Lab 2

Ted Obuchowicz

January 8, 2021

1 Objectives

- To explore addressing modes.
- Gain additional familiarity with Linux utilities.
- Gain additional familiarity with `nasm` and `gdb`.

2 Introduction

This lab introduces several Intel x86 addressing modes - which are the means that a particular assembly language instruction can access the data operands. More features of the `nasm` assembler, `gdb` debugger and other Linux utilities are explored.

The first lab used only what is referred to as "immediate mode". Recall the two instructions from Lab 1:

```
        mov ax,5  ; store 5 into the ax register
keith:  mov bx,2  ; store 2 into the bx register
```

In Intel x86 assembly language, instructions are of the general format:

```
label: opcode destination_operand, source_operand
```

A label is optional and is used to generally refer to a particular instruction. In the two instructions, the numeric constants 5 and 2 are called immediate data. The term immediate is used to refer to the fact that the data is stored as part of the instruction. Hence, once the instruction has been fetched from main memory and is stored inside the processor, the CPU does not need to access main memory in order to execute the instruction. All the data required by the instruction is already within the processor. The information in the listing file more clearly emphasizes this point:

```
00000000 66B80500            mov ax,5
00000004 66BB0200    keith:  mov bx,2
```

The first set of numbers in the left most columns refer to addresses in main memory where the machine code for the program is stored. Actually, at this point in time, they refer to offsets from some starting address in main memory which will contain the machine code of the program since it is not yet known at assemble time where in main memory the program will be loaded into.

The second set of numbers is the actual machine code given in hexadecimal format. Since two hex digits corresponds to one byte, we can draw the layout of the first two instructions in main memory as:

Address	Value	Comments
00000000	66	This is the first instruction
00000001	B8	66BB is the `mov ax` part of the instruction
00000002	05	The 0500 is the data part of the instruction
00000003	00	Intel stores multi-byte numbers "backwards"
00000004	66	This is the second instruction.
00000005	BB	66BB is the `mov bx` part of the instruction
00000006	02	The 0200 is the data part of the instruction
00000007	00	Intel stores multi-byte numbers "backwards"

A few comments worth noting:
In the original assembly language source file (`.asm`), the programmer wrote:

```
mov ax, 5
```

But note how the assembler stored the constant `5` as a two byte number: `00 05`. The assembler was able to determine that two bytes are required (as opposed to 1 byte or 4 bytes) since the destination operand was specified to be the `AX` register and the `AX` register is 16 bits, or 2 bytes wide.

Intel assembly language supports another fundamental addressing mode called *direct* (also known as *absolute*). Direct addressing is useful when the data is stored in some main memory location (as variables in a C++ program are stored).

The following program makes use of direct addressing and also introduces the `dw` (define word) assembler directive:

```
; Ted Obuchowicz
; June 9, 2020
; sample program to add two numbers which
; are stored somewhere in memory

section .data

; put your data in this section using
; db , dw, dd directions

mick  dw 2  ; define one word (2 bytes)  of data
keith dw 3  ; define another word of data with value 3

section .bss

; put UNINITIALIZED data here using
; this program has nothing in the .bss section
```

```
section .text
        global _start

_start:
        mov ax,[mick]    ; store contents of memory word at
                         ; location mick into  into the ax register
ron:    mov bx,[keith]   ; store contents of memory word  at
                         ; location keith into the bx register
        add ax,bx        ; ax = ax + bx
                         ; contents of register bx is added to the
                         ; original contents of register ax and the
                         ; result is stored in register ax (overwriting
                         ; the original content


        mov eax,1             ; The system call for exit (sys_exit)
        mov ebx,0             ; Exit with return code of 0 (no error)
        int 80h
```

In Intel assembly language, putting `[ ]` around a label means the "address of the label", or "the data is found in main memory associated with the label". Let us take a look at the partial listing file corresponding to the `.data` section portion:

```
00000000 0200                   mick  dw 2
00000002 0300                   keith dw 3
```

The first set of numbers again correspond to memory addresses (acually offsets from what is called the *data segment* which is a portion of main memory used to hold program data. The data segment is a distinct section of memory from the *text segment* which is used to hold the machine code of the program.

The number `0002` is stored in at the beginning of the data segment. This is why its *offset* from the beginning of the data segment is `00000000`. It occupies 2 bytes of main memory.

The number `0003` is stored in the next two consecutive bytes at offsets `00000002` and `00000003`. Pictorially, the twp words of data are stored in memory as:

Offset	Data	Comments
00000000	02	Low byte of first number
00000001	00	High byte of first number
00000002	03	Low byte of second number
00000003	00	High byte of second number

Again, note how the Intel x86 architecture stores numbers in a "backwards" format. This manner of storing multi-byte data is called *little-endian*, as opposed to the opposite fashion referred to as *big-endian*.

The nasm assembler allows for the following data definition directives when running in 32-bit mode:

- db

  Define 1 byte

- dw

  Define word (1 word = 2 bytes)

- dd

  Define double word (2 words = 4 bytes)

Examining the first two lines of the listing corresponding to the first two instructions of the program, we find:

```
00000000 66A1[00000000]                 mov ax,[mick]
00000006 668B1D[01000000]       ron:    mov bx,[keith]
```

Note how the nasm assembler has put [ ] around the respective offsets of labels mick and keith. This is meant to be interpreted that these numbers are offsets and not immediate data. The 66A1 portion of the first instruction encodes that the instruction is a move, and the destination operand is the AX register and the source operand is specified using direct mode and the [00000000] is the offset from the beginning of the data segment which contains the data. There is quite a lot of information here.

Similarly, the 668B1D part of the second instruction is the machine code encoding for stating that this instruction is mov bx with the source operand specified using direct addressing. The [01000000] specifies that the offset of the memory operand is 1. Note how even offsets are stored in little-endian (from low byte to high order byte) format.

3 Exercise

The goal of this lab exercise is to assemble the program with nasm, create an executable with ld, and single-step through the program with gdb. While doing so, verify the contents of the AX and BX registers after each instruction.

3.1 Step 1

The first step is to assemble the program with nasm to generate the .o and the .lst files from the .asm file. This can be done with the following command:

```
ted@cmos NASM 8:10pm >nasm -f elf -o add_2_numbers_mem.o -l
add_2_numbers_mem.lst add_2_numbers_mem.asm
```

3.2 Step 2

The next step is to create an executable with `ld` from the object file. This can be done with the following command:

```
ted@cmos NASM 8:10pm >ld -melf_i386 -o add_2_numbers_mem add_2_numbers_mem.o
```

3.3 Step 3

Once the program has been linked, it can be executed and single-stepped with `gdb`. During this step, observe memory contents and register values throughout the execution of the program.

```
ted@cmos NASM 8:11pm >gdb add_2_numbers_mem
GNU gdb (GDB) 7.7
Copyright (C) 2014 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-unknown-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 add_2_numbers_mem...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
```

Now that `gdb` is running and the preferred disassembly flavor is set, set a breakpoint at the label `_start`, which is the first instruction.

```
(gdb) break _start
Breakpoint 1 at 0x8048080
(gdb) run
Starting program: /nfs/home/t/ted/COEN311/NASM/add_2_numbers_mem

Breakpoint 1, 0x08048080 in _start ()
```

Now that the breakpoint has been reached and execution has been paused, disassemble the program:

```
(gdb) disassemble
Dump of assembler code for function _start:
=> 0x08048080 <+0>:     mov    ax,ds:0x804909c
End of assembler dump.
```

Is there any observable change in the disassembled output compared to the machine code found in the `.lst` file?

Instead of specifying offset values such as in the `.lst`, the disassembled machine code now gives an actual memory address for what was originally `[mick]` in the `.asm` file. The reason for this is that the loader program decided upon some location in memory for the program to be stored in and the offsets were replaced with actual addresses. Based on this, it can be concluded that `mick` is associated with address `0x804909c`. Single-step the next instruction and disasemble:

```
(gdb) ni
0x08048086 in ron ()
(gdb) disassemble
Dump of assembler code for function ron:
=> 0x08048086 <+0>:     mov    bx,WORD PTR ds:0x804909e
   0x0804808d <+7>:     add    ax,bx
   0x08048090 <+10>:    mov    eax,0x1
   0x08048095 <+15>:    mov    ebx,0x0
   0x0804809a <+20>:    int    0x80
End of assembler dump.
```

Can you determine the memory address associated with the label `keith` from the disassembled code?

Answer: `keith` is associated with the address `0x804909e`.

Now, use the `x` command in `gdb` to examine the contents of memory:

```
(gdb) x/1xb &mick
0x804909c:      0x02
```

In the `x/1xb` command, the following parameters are passed:

- `1`

  Size specifier, dictates the number of items to show

- `x`

  Format specifier, dictates the representation used for the data to be examined. (x for hexadecimal, d for decimal, t for binary, etc.)

- `b`

  Data Size Specifier (b for byte, h for half word or 2 bytes, w for word or 4 bytes)

Given `x/1xb`, this implies show the contents of memory with 1 item of size byte in hexadecimal. We can examine the two bytes stored beginning at `mick` with:

```
(gdb) x/2xb &mick
0x804909c:      0x02    0x00
```

Similarly, we can examine the two bytes beginning at `keith` with:

```
(gdb) x/2xb &keith
0x804909e:      0x03    0x00
```

We can also examine the four bytes beginning at `mick` with:

```
(gdb) x/4xb &mick
0x804909c:      0x02    0x00    0x03    0x00
```

One can even specify an address instead of `&some_label`:

```
(gdb) x/4xb 0x804909c
0x804909c:      0x02    0x00    0x03    0x00
```

Be careful when using the `h` or `w` size specifier. For example, to examine the half word (2 bytes) in hex at `mick`:

```
(gdb) x/1xh &mick
0x804909c:      0x0002
```

Note how the `x` command in `gdb` has reversed the bytes so that the two byte number appears in the "normal" or *big-endian manner*. This is just a convenience for the user.

COEN311 Lab 3

Ted Obuchowicz

January 8, 2021

1 Objectives

- To obtain a strong understanding of the available Intel x86 addressing modes.
- To learn the basic arithmetic operations.
- To apply one's knowledge of addressing modes and arithmetic operations to write an assembly language program which accesses the elements of a two dimensional array of integers which are stored in memory.
- To gain an appreciation of the task performed by a high level programming language compiler.

2 Prelab

Prior to attending the lab session, the student is expected :

- Have read the the "Addressing Modes" and the "Instruction Set summary" with emphasis on the `mul` instruction sections in the "Introduction to Introduction to x86 Assembly" document. It is also strongly suggested to read the preliminary sections as well as a review of fundamental material.
- Have read the posted `more_intel_addressing_modes.asm` Intel assembly language program which explains variations on the register indirect addressing mode (which will be helpful for the purposes of this lab).
- Have read the *entire* contents of this document.

3 Introduction

Consider the C++ declaration of a two-dimensional array of integers:

```
int my_array[4][3] = {1, 2, 3,
                      4, 5, 6,
                      7, 8, 9,
                      10,11,12};
```

In a high-level programming language such as C++, one makes use of array notation to access a particular array element. For example,

```
int my_element ;

my_element = my_array[3][2] // assign array element with value 12
                            //  to integer variable my_element
```

The use of the `[ ][ ]` array index operators represents a level of abstraction, which is also known as "let the compiler do the dirty work of figuring out where in main memory the particular array element is stored at". Recall, that a high-level programming language first converts the code into assembly language then invokes the native assembler which converts to machine code. Ultimately, all we have to work with is the CPU instruction set.

Most introductory programming concepts introduce the concept of array address translation formulas. The elements of a (two-dimensional or higher) array are stored row by row in main memory. Main memory is abstracted as a one-dimensional array of locations. The two-dimensional array address translation formula is:

$$\text{Location of } \texttt{a[i][j]} = x + (((i \times \#\text{ of columns}) + j) \times \texttt{sizeof(data\_type)})$$

where x is the starting address in main memory where the first array element is stored

In C++, the name of an array is synonymous with the starting address in main memory where the first element of the array is stored. Thus, the above formula can be succinctly stated as `a[i][j]` is found at address:

$$\texttt{a} + (((i \times \#\text{ of columns}) + j) \times \texttt{sizeof(data\_type)})$$

Refer to the Appendix called "How 2-Dimensional Arrays Are Stored in Memory" for a pictorial explanation of the 2-D array address translation formula.

4 Procedure

Write an Intel x86 assembly language which makes use of the 2-D array translation formula to access the array elements, making use of the register indirect addressing mode (or more specifically "based-indexed addressing" which is just a variation of good ol' register indirect mode).

Use the following `.data` section in your program:

```
section .data

; put your data in this section using
array db 3,2,4,1,5,6

; we will assume that these 6 numbers represent
; a 2-d array of 3 rows and 2 columns
;
; 3 rows and 2 columns
; array =  3 2
;          4 1
;          5 6
```

Use one register to hold the starting address of the array, and use another register which holds the *offset* of the element. You may hardcode two registers to hold the row and column indices of the desired array element. For example:

```
mov al, 1       ; al holds row index
mov bl, 1       ; bl holds column index
```

Load the desired array element into a register. Test your program using `gdb` for several typical values of row and column indices to ensure your program is working correctly. For the purposes of this lab, it is suffcient to merely edit your source code and put new values into the row and column indices, re-assemble and re-run with `gdb`.

5 Appendix: How 2-Dimensional Arrays Are Stored in Memory

Assume the following declaration of a two-dimensional array of integers, of size 4 rows, with each row consisting of 3 columns:

```
int a[4][3] = {1,  2, 3
               4,  5, 6,
               7,  8, 9,
               10, 11, 12};
```

Pictorally, this array is represented as a two-dimensional matrix (a typical one from a linear algebra course):

$$\begin{array}{c} \\ \text{Row 0} \\ \text{Row 1} \\ \text{Row 2} \\ \text{Row 3} \end{array} \begin{array}{c} \begin{array}{ccc} \text{Col 0} & \text{Col 1} & \text{Col 2} \end{array} \\ \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \end{array}$$

Two nested `for` loops may be used to print out this array:

```
for (int row = 0; row < 4; row++)
{
    for (int col = 0; col < 3; col++)
    {
        cout << a[row][col] << " ";
    }

    // Print a new line for the next row
    cout << endl;
    // Continue with row outer loop
}
```

5.1 Useful Exercise

A useful exercise to perform on your own is to determine how to modify the above code such that the *transpose* of the array is printed.

Hint: In a matrix transpose, the rows become the columns and the columns become the rows. The transpose of the given array is:

	Col 0	Col 1	Col 2	Col 3
Row 0	1	4	7	10
Row 1	2	5	8	11
Row 2	3	6	9	12

5.2 How 2-Dimensional Arrays Are Stored In Memory

Consider our original 2-D array of size 4 rows by 3 columns:

1	2	3	Row 0
4	5	6	Row 1
7	8	9	Row 2
10	11	12	Row 3

This 2-D array is stored in main memory in a *row-by-row* manner. This is formerly known as *row-major* form. For ease of identification, the individual rows of the array have been indicated by a unique background colour.

Array Element	Main Memory	Address
	...	
`a[0][0]`	1	1000
`a[0][1]`	2	1004
`a[0][2]`	3	1008
`a[1][0]`	4	1012
`a[1][1]`	5	1016
`a[1][2]`	6	1020
`a[2][0]`	7	1024
`a[2][1]`	8	1028
`a[2][2]`	9	1032
`a[3][0]`	10	1036
`a[3][1]`	11	1040
`a[3][2]`	12	1044
	...	

Each element of the array is 4 bytes in size since each element of the array is an integer. There is a 2-D array address translation formula that the compiler uses when it encounters array notation in the program code:

$$\texttt{a[i][j]} = \texttt{a} + ((i \times 3 + j) \times \texttt{sizeof(int)})$$

where 3 is the number of columns in every row

Let us use this formula to show how the compiler translates `a[1][2]` into an address which contains the first byte of the integer element stored in position `a[1][2]` of the array:

$$\begin{aligned} \texttt{a[1][2]} &= \texttt{a} + ((1 \times 3 + 2) \times 4) \\ &= 1000 + (3 + 2) \times 4 \\ &= 1020 \end{aligned}$$

Convince yourselves that the 2-D array address translation formula properly converts from good ol' `a[i][j]` array notation into the starting address of main memory in which element `a[i][j]` of the array is stored in main memory. Since the compiler only needs to know:

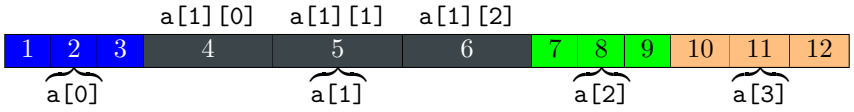
1. The number of columns in every row so it knows what number to use for the translation formula
2. The data type of the array so that it knows what to put in the `sizeof()` expression to compute the size in bytes of each element of the array.

It is for the above reasons that when we pass a 2-D array to a function, we **specify the number of columns but not the number of rows**. As an example:

```
void some_func (int some_array[][3])
{
    // whenever in the function we use array notation
    // similar to some_array[i][j], the compiler translates this
    // to address:
    // some_array + (((i *3)+j) * sizeof(int))
}
```

Since the number of rows in the array is not required in the address formula, it is not necessary to specify the number of rows when passing a 2-D array to a function. Recall that just as in the 1-D case, the argument name `some_array` simply contains the starting address of the passed array.

Consider an alternate view. In this view, a two-dimensional array may be considered a 1-D array of 1-D arrays. With this view, the 2-D array can be shown as:



The first dimension in the array definition `a[4][3]` selects one of these *colored sets of blocks*. Each block itself is another array where the second dimension is the individual block in the set. Each block is an `int`.

COEN311 Lab 4

Ted Obuchowicz

January 8, 2021

1 Objectives

- To gain an understanding of the flow control assembly language statements such as unconditional, conditional jumps and compare instructions.
- To gain an understanding of the string display `int 80h` interrupt routine.
- To apply these concepts in order to write a complete Intel x86 assembly language program which converts a string of characters into all upper case.

2 Prelab

Prior to attending the lab session, the student is expected to:

- Have read the *Flow Control* section (section 7.4) in the *Introduction to x86 Assembly* document.
- Have read the *entire* contents of this lab handout.

3 Procedure

3.1 Displaying Characters On The Screen in Assembly Language

The `int 80h` family of interrupts contains a useful routine used to display a string of ASCII characters on the screen. Its use is straightforward and documented in the following program:

```
; Ted Obuchowicz
; Nov. 10, 2020
; write_string_to_display.asm

section .data
 hello   db 'Hello world!',0x0a  ; 'Hello world!' plus a linefeed character

section .text
        global _start

_start:
        mov eax,4            ; The system call for write (sys_write)
        mov ebx,1            ; File descriptor 1 - standard output
        mov ecx,hello        ; Put the  address of the string  in ecx
        mov edx,13           ; Put number of characters to write in edx
                             ;  string contains 13 characters in total
        int 80h              ; Call the  interrupt routine which
                             ; performs the display of the characters
```

```
                                ; on the screen

        mov eax,1               ; The system call for exit (sys_exit)
        mov ebx,0               ; Exit with return code of 0 (no error)
        int 80h
```

By loading 4 into the `EAX` register, 1 into `EBX`, the main memory address of some character string stored in main memory into the `ECX` register, and by specifying the number of characters to be printed in the `EDX` register, invoking the `int 80h` interrupt will display the specified number of characters on the screen. For example, the output produced by the above program, when run directly from the Linux prompt (no need to single step with `gdb`) is:

```
[ted@localhost NASM]$ ./write_string_to_display
Hello world!
```

The string display routine provided above will be used in this lab.

3.2 The Task At Hand

Consider the task of converting an array of ASCII characters from lower case into their upper-case equivalents. Any non-alphabetic characters are to remain as is. If the character is already in upper-case form, then it should remain as is. The last character in the array is the NULL character (ASCII code = 0).

The lowercase ASCII characters `a`, `b`, `c`, ... `z` are represented with the ASCII codes 97 through to 122. To convert a lower-case ASCII alphabetic character into its upper-case equivalent, we simply subtract decimal 32 from the lower case ASCII value. For example, For example, `A` = `a` - 32 (65 = 97 - 32).

The following C++ program illustrates the lower-case to upper-case conversion process. It makes use of the infamous `goto` statement which has caused considerable debate among computer scientists who quibble about the existence of the `goto` statement in high-level programming languages. Its use in this example is however justified as there is a direct link between the C++ program and its assembly language equivalent.

```
// Author: Ted Obuchowicz
// lower-case to upper-case ASCII conversion


#include <iostream>
using namespace std;

int main()
{

char array[] = {'j','A','c','K',' ','f', 'L', 'a', 'S', 'h','#', '1', '\0'};
```

```
// print out the array before converting to all UPPER case

cout << array << endl;

int i = 0;

// figure out how to implement a while in assembly language
// hint... do a MOVE of the first element of the array
// and branch to somewhere if it's zero
// if it's not equal to 0, then perform the body of the loop


while ( array[i] != '\0')   // while we did not yet reach the end of
                                 // the string
    {
          if ( array[i] <  'a' ) // figure out how to do an if in assembly
          {
            goto next;  // skip over any non-lowercase letters, figure
                        // out how to do a goto in assembly
          }

         if ( array[i] > 'z' ) // figure out how to do another if in assembly
         {
           goto next;  // skip over any non-lowercase letters
         }

    // if we got to here, we know the letter is a lower case one
    // so convert it to its uppercase version by subtracting 32
    // from its ASCII value.

        array[i] = array[i] - 32 ;

        next: i = i + 1; // point to the next letter

    }


// print it out to see if we did this correctly...

cout << array << endl;

return 0;
}
```

When ran, the program will output the following string: `JACK FLASH#1`
Here is another version which makes greater use of the goto statement which

leads to “spaghetti” code but illustrates in a clear manner the assembly language version:

```
// Author: Ted Obuchowicz
// lower-case to upper-case ASCII conversion
// uses the controversial goto statment

#include <iostream>
using namespace std;

int main()
{

char array[] = {'j','A','c','K',' ','f', 'L', 'a', 'S', 'h','#', '1', '\0'};

// print out the array before converting to all UPPER case

cout << array << endl;

int i = 0;

// figure out how to implement a while in assembly language
// hint... do a MOVE of the first element of the array
// and branch to somewhere if it's zero
// if it's not equal to 0, then perform the body of the loop

start: if ( array[i] == '\0')
       {
         goto the_end;
       }
       else
       {
        if ( array[i] <  'a' ) // figure out how to do an if in assembly
        {
          goto next;  // skip over any non-lowercase letters, figure
                      // out how to do a goto in assembly
        }

        if ( array[i] > 'z' ) // figure out how to do another if in assembly
        {
         goto next;  // skip over any non-lowercase letters
        }

// if we got to here, we know the letter is a lower case one
// so convert it to its uppercase version by subtracting 32
// from its ASCII value.
```

```
        array[i] = array[i] - 32 ;

        next: i = i + 1; // point to the next letter

        }

        goto start;


// print it out to see if we did this correctly...

the_end: cout << array << endl;


return 0;
}


// yes, it works as this is the output

//JACK FLASH#1
```

You are to write an Intel x86 assembly language program which makes use of a loop to access the individual elements of the array containing the ASCII characters. You are to initialize the array with the following `db` directive:

```
message db 'juMping JAck flaSh #1',10, 0
```

The ASCII character represented by decimal value 10 (`0x0A`) is the *line feed* character (similar to `endl` in C++). The byte with all zeros (the so called `NULL` character in ASCII) is used to represent the end of the string.

Before entering the loop, which performs the case conversion, the program is to display on the screen the (original) contents of the string. Within the loop, the program is to determine whether the current character represents a lower case character. If the character is lower case, it is to be converted into its upper case version. Non-alphabetic characters are to remain as is. Upon converting all the lower case characters, the program is to display the string on the screen. Here is an example of a correct program execution:

```
ted@cmos NASM 2:25pm >to_upper_with_display
juMping JAck flaSh #1
JUMPING JACK FLASH #1
ted@cmos NASM 2:25pm >
```

Test your program by running the executable from the Linux command prompt. If it works properly, you are done. If it does not work properly, debug your assembly language program using `gdb`.

4 Deliverables

In one `.zip` file, submit:

- The `.asm` file.
- The listing file.
- Your written lab report in PDF format.