COMPUTER SYSTEMS
PROJECT 1
Working with simple assembly/machine code

1 *x86* assembly code

We have discussed, during lecture, the basics of *assembly code* and how it is transformed into *machine code*. For this first project, you are going to get a little hands-on experience with both forms of code and how they are really used.

For our assignments, we will be working on *Linux* systems that run on processors that implement the *x86-64 instruction set architecture (ISA)*. While we will later work with the *C* programming language, for this assignment, we will use the *nasm* assembler to translate our assembly code to machine code, and the *GNU debugger (gdb)* to help us run and debug the code. I will note here that the materials available for all of these—*Linux*, *x86-64*, *nasm*, and *gdb*—is an embarrassment of riches. There is extensive documentation, tutorials, code samples, and discussions on their uses, targeted at audiences ephebic to expert. In short, when you run into difficulty or are unsure of what to do, **first, use The Google**. In this context, it is the right thing to do to find answers and understand more.

Our foray into this type of assembly programming is going to require an understanding of the following capabilities and concepts:

- *Sections:* The division of the code into *instructions (text)* and *data*.
- *Labels:* The marking of specific instructions and data with names.
- *Instructions:* The sequence of steps, each defined by an *opcode* and *operands*, that make up the program.
- *Registers:* The small set of fast memory elements to hold data.
- *Main memory:* The addressable storage of all of the instructions and data and its layout.
- *System calls:* How to call into the functions of the *operating system kernel*, passing it arguments.

2 Getting started

1. You must login: to begin with. There is no doubt whatever about that. Use *Remote Desktop* to connect to either of the college's Linux servers, `remus.amherst.edu` or `romulus.amherst.edu` (henceforth, `remus/romulus`). Login using your usual college username and password.

2. Open a terminal window.

3. Make a new directory for this class and project, and change into it:

```
$ mkdir -p systems/project-1
$ cd systems/project-1
```

4. Download the project's source code:

```
$ wget -nv -i https://bit.ly/COSC-171-project-1-source
```

5. Open our first assembly code program:

```
$ emacs hello.asm &
```

3 An already-written program

You should now have an *Emacs* window open, showing you a simple program that writes a message to the console (henceforth, *standard output*, or *stdout*). Here is what you should do with it:

- **Read it:** This program sets up and performs two *system calls*. The first prints a message by calling on the kernel to WRITE a string to the *stdout*; the second calls the kernel to EXIT, thus ending the program. See how various registers are set to appropriate values to carry the desired operation and arguments to each system call.

- **Assemble it:** Translate this "human readable" assembly code [`hello.asm`] into machine code (specifically, *object code*) [`hello.o`]:

```
$ nasm -felf64 -g hello.asm
```

- **Link it:** Wrap the object code [`hello.o`] in a special layout that the kernel will interpret as a runnable program, known as an *executable file* [`hello`]:

```
$ ld -o hello hello.o
```

- **Debug/test it:** Load the executable file into the *debugger*, where we can run it in a very controlled fashion and see the result of each step. Once loaded, first *disassemble* the program, making *gdb* turn the machine code back into assembly code:

```
$ gdb hello
(gdb) disassemble _start
Dump of assembler code for function _start:
 0x00000000004000b0 <+0>:     movabs $0x1,%rax
 0x00000000004000ba <+10>:    movabs $0x1,%rdi
 0x00000000004000c4 <+20>:    movabs $0x6000ec,%rsi
 0x00000000004000ce <+30>:    movabs $0xd,%rdx
 0x00000000004000d8 <+40>:    syscall
 0x00000000004000da <+42>:    movabs $0x3c,%rax
 0x00000000004000e4 <+52>:    sub    %rdi,%rdi
 0x00000000004000e7 <+55>:    syscall
End of assembler dump.
```

There are a number of things worth noting in this disassembly:

- The first column shown is the *main memory address* at which the program's machine-code instructions have been loaded. The addresses are shown in *hexadecimal*, or *base 16*, which is denoted by the prefix `0x` on each address. The starting address of each instruction to shown.
- The second column, in angle-brackets, is the *address offset* of each instruction. That is, it is the number of bytes from the beginning of the code to the given instruction. For some strange reason, the offsets are given in decimal.
- The third column provides the *opcode* of each instruction. Notice that the assembler may have changed the opcode to be slightly different from the one written in the source assembly code. For example, the `movabs` opcode appears in place of the `mov` opcode originally written. These changes are, for our purposes, not important; do a web search for `movabs` if you want to learn what the deal is. What matters is that you not be surprised or distressed by these changes.
- What remains are the *operands*, and they are shown in a form that is clearly different. Here, constants are shown with a `$` prefix, and are in hexadecimal. Additionally, register names are prefixed with the `%` symbol. These are merely changes in assembly convention that, again, are not important for our purposes, and merely need to be seen as normal. If you are curious about the difference, you can read about the difference between difference between *AT&T* and *Intel assembly syntaxes*.

Now let's set a *breakpoint*, telling *gdb* where in the program to pause when it reaches that point, and then run the program to reach that point:

```
(gdb) break _start
Breakpoint 1 at 0x4000b0
(gdb) run
Starting program: /home/staff/sfkaplan/systems/project-1/hello

Breakpoint 1, _start () at hello.asm:13
```

```
13      _start:   mov       rax, 1
(gdb)
```

Now we can go through our program, one instruction at a time, seeing the registers change and things happen:

```
Starting program: /home/staff/sfkaplan/systems/project-1/hello

Breakpoint 1, _start () at hello.asm:13
13      _start:   mov       rax, 1
(gdb) si
14                mov       rdi, 1
(gdb) p $rax
$1 = 1
(gdb) si
15                mov       rsi, message
(gdb) p $rdi
$2 = 1
(gdb) si
16                mov       rdx, 13
(gdb) p/x $rsi
$3 = 0x6000ec
(gdb) si
17                syscall
(gdb) p $rdx
$4 = 13
(gdb) si
Hello, World
18                mov       rax, 60
(gdb) si
19                sub       rdi, rdi
(gdb) si
20                syscall
(gdb) p $rdi
$6 = 0
(gdb) si
Program exited normally.
(gdb) quit
```

Note the following commands:

- `si`: Step forward one instruction. That is, run the next instruction and then pause again.
- `p $reg`: Print, in decimal, the value of a given register, which name must be (anomalously) prefixed with the `$` character.

  - `p/x $reg`: Print, in hexadecimal, the value of the given register.
  - `run`: Although we used it above to get to the breakpoint, you could issue this command at any point in the middle of the program, causing it to move forward though the instructions without pausing until it reaches another breakpoint or the process ends.

- **Run it:** Now that we see what's happening inside, let's just run it normally:

```
$ ./hello
Hello, World
```

  Notice that, on the command line, you must use the prefix `./` on the executable file name. That indicates to the shell that the program to be run is in *this directory, right here*; the `hello` file in this directory should be loaded. Without that prefix, the shell will look through a list of pre-set directories—the `PATH` environment variable—for an executable file named `hello`; when it doesn't find it will report `Command not found`.

- **Change it:** Open the `hello.asm` code in *Emacs* again. **Change the message**, modestly, to something a little more lengthy and personal. *"Working in hexadecimal is cruel"*, or whatever feels right to you.

  Having changed it, go back and **assembly, link, debug, and run** the newly modified version. Make sure it works.

4 Countdown

It is time to write (or, at least, complete) a slightly more interesting program. In your terminal, open up the other file that you downloaded earlier:

```
$ emacs countdown.asm &
```

You will see here a skeleton of a program. As its comment header explains the program is supposed to do the following if it works properly:

```
$ ./countdown
9
8
7
6
5
4
3
2
1
0
```

The start to the program, the system call to EXIT the program, and the *data* section containing the string for the first line of output are all provided.

Your assignment is to write the loop that counts down from 9 to 0, generating the output along the way, as shown above. You should use all of the tools that you used on the `hello.asm` program in order to assemble, debug, and ultimately run a correctly running program. You will likely need to use the following opcodes discussed in class: `sub`, `cmp`, and `je/jne`.

5 How to submit your work

Submit your `hello.asm` and `countdown.asm` files. You may use either of the following two methods to use the CS submission system:

- **Web-based:** Visit the submission system web page.
- **Command-line based:** Use the `cssubmit` command at the shell prompt on `remus/romulus`.

This assignment is due on Wednesday, Sep-12, 11:59 pm.