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. **Install and set up the tools:** If you haven't already, follow the instructions to set up the tools needed to do the projects.

2. **Login to the server:** Open a terminal window to get a shell prompt, and then connect to the class server…

   ```
   $ ssh yourusername@systems.cs.amherst.edu
   ```

3. **Login to GitLab:** Before you download the code to get started, create a *repository* on our GitLab server, which will server as a backup and history of your work. (It is also how you will submit your completed assignment.)

   From your browser, login to `gitlab.amherst.edu`

4. **Start a new project:** In the upper-left of the GitLab page, you will see a little drop-down menu marked by a plus-sign. Click that, and then select from the menu, `New project/repository`.

   A new window will appear (*Create new project*), giving you options of what to create. Select `Create blank project`.

5. **Name and create the project:** Set the *Project name* to be `sysproj-1`, and leave the other default values. Click on the `Create project` button at the bottom.

6. **Clone the repository onto the course server:** Next, *clone* the repository you just made on the GitLab site onto the course server, and then *change directories* into the newly created directory on the server. In your terminal window, connected to `systems.cs`, It should look like this:

   ```
   $ git clone git@gitlab.amherst.edu:yourusername/sysproj-1.git
   Cloning into 'sysproj-1'...
   remote: Enumerating objects: 3, done.
   remote: Counting objects: 100% (3/3), done.
   remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
   Receiving objects: 100% (3/3), done.
   $ cd sysproj-1
   ```

7. **Fix your identity for the repository:** `git` allows users to specify their identity (name and email), and it wants you to confirm the automatically configured values by GitLab (lest it give you lots of warning messages). To do so, do the following:

   ```
   $ git commit --amend --reset-author -m "Fixing identity."
   ```

This command will be followed by the very warning that it is intended to quash; it should be the one and only time that you see it.

8. **Download the source code:** The `wget` command will copy two source code files into your directory. You can list the directory to see them after you perform the download:

```
$ wget -nv -i https://bit.ly/cosc-171-24s-p1
$ ls -l
total 9.0K
-rw------- 1 you you 1.4K Aug 14 13:58 countdown.asm
-rw------- 1 you you 1.1K Aug 14 13:58 hello.asm
-rw------- 1 you you  194 Aug 14 13:58 project-1-files.txt
```

9. **Add the source code to the repository:** Tell git that the two assembly code files (`.asm` files) are now part of the repository:

```
$ git add *.asm
```

Notice that **no news is good news**. This command generates no output when everything goes correctly.

10. **Commit and push the updated repository:** First, *commit* the changes—the addition of the `.asm` files. Then, *push* this update to GitLab.

```
$ git commit -m "Starting code."
[master (root-commit) c812cdc] Starting code.
 2 files changed, 70 insertions(+)
 create mode 100644 countdown.asm
 create mode 100644 hello.asm
$ git push
Enter passphrase for key '/home/yourusername/.ssh/id_rsa':
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 1.15 KiB | 1.15 MiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To gitlab.amherst.edu:yourusername/sysproj-1.git
 * [new branch]      master -> master
```

If you like, you can go back to your browser and refresh the GitLab page. You should see the source code files appear as part of the repository there.

11. **Open our first assembly code program:** At your shell prompt, open the `hello.asm` assembly source code file into a text editor. Specifically, I recommend using *Emacs* or *Vim*, both of which are made for editing code. If you have never used either, a simple web search for *emacs tutorial* or *vim tutorial* will show you a number of useful beginner's guides to learning these editors.[1] To open the file with *Emacs*, for example, do the following, and then move onto the next section…

```
$ 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/terminal (henceforth, *standard output*, or *stdout*). Here is what you should do with it:

- **Read it:** This program sets up and performs two *system calls*—special function calls to an underlying, basic operating system kernel. 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`].[2]

```
$ nasm -felf64 -gdwarf 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:

[1] As an alternative, it is possible to set up *VSCode* with the *ssh extension* plug-in to connect to the server and do your editing; the TA's can help with this setup if you want to try it.

[2] *Object code* files are an intermediate form between assembly and an executable program. For bigger programs, multiple object files, which contain *machine code* and *statics* produced by the assembler, and be combined by a *linker* to make a single executable program file.

```
$ gdb hello
(gdb) disassemble _start
Dump of assembler code for function _start:
Dump of assembler code for function _start:
  0x0000000000401000 <+0>:     mov    $0x1,%eax
  0x0000000000401005 <+5>:     mov    $0x1,%edi
  0x000000000040100a <+10>:    movabs $0x402000,%rsi
  0x0000000000401014 <+20>:    mov    $0xd,%edx
  0x0000000000401019 <+25>:    syscall
  0x000000000040101b <+27>:    mov    $0x3c,%eax
  0x0000000000401020 <+32>:    sub    %rdi,%rdi
  0x0000000000401023 <+35>:    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 sometimes 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 0x401000
(gdb) run
Starting program: /home/yourusername/sysproj-1/hello

Breakpoint 1, _start () at hello.asm:18
18  mov rax, 1  ; rax gets the system call code for "write".
(gdb)
```

Now we can go through our program, one instruction at a time, seeing the registers change and things happen:[3]

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

Breakpoint 1, _start () at hello.asm:18
18  mov  rax, 1
(gdb) si
19  mov  rdi, 1
(gdb) p $rax
$1 = 1
(gdb) si
20  mov rsi, greetings
(gdb) p $rdi
$2 = 1
(gdb) si
21  mov  rdx, 13
(gdb) p/x $rsi
$3 = 0x402000
(gdb) si
22  syscall
(gdb) p $rdx
$4 = 13
(gdb) si
Hello, World
24  mov  rax, 60
(gdb) si
25  sub  rdi, rdi
(gdb) si
26  syscall
(gdb) p $rdi
$5 = 0
```

[3]I have remove the code comments from the end of the lines of code shown here, since they wouldn't reasonably fit on the page.

```
(gdb) si
[Inferior 1 (process 19462) 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. Then, commit/push your updated code to Gitlab:

  ```
  $ git add hello.asm
  $ git commit -m "Hello, with a modified message."
  $ git push
  ```

  Get into the habit of performing a commit/push of files that you modify. Each modification is added to a history of updated that GitLab keeps, providing a backup and the ability to jump back to previous versions if we need to do so.

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
```

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 that we have discussed (or will discuss!) in class: `sub`, `cmp`, and `je/jne`.

5 How to submit your work

First, be sure that the most recent versions of your work have been *committed* and *pushed* to the GitLab server.

Then, go to GitLab with your browser. Navigate to the `sysproj-1` project. On the left side of the page, click on `Settings` to reveal a drop-down menu, from which you should select `Members`.

In this *Project members* window, under *Invite member*:

1. Under *Select members to invite*, enter `sfkaplan`. You will see me appear (*Prof. Scott Kaplan*) as a user; select me. Also select
2. Under *Choose a role permission*, click the drop-down menu and select `Developer`.
3. Below, click the `Add to project` button.

This assignment is due on Sunday, Feb-11, 11:59 pm.