Assignment 3: Functions and Loops in Assembly

Due: Friday, February 3, 10pm

Starter code: See Assignment 3 on Canvas for the Github link.

Submission: Submit the contents of your repository via Gradescope. See Deliverables below for what to submit.

This is an *individual* assignment.

This assignment will take you through 2 tasks implementing and using functions in assembly. You will be also asked to write some C (pseudo-)code as a blueprint for your assembly. The C code does not need to compile or run, but needs to reflect the functionality. Feel free to rely on your Java knowledge here. If in doubt, leave comments.

This assignment will also introduce *separate compilation*. For one of the tasks you will write a function called by an already implemented `main` function. For the other, your task will be to write a `main` function for an already implemented function. Read the instruction carefully to make sure you understand what is being asked in each of the tasks.

For each of the required functions, you need to follow the Assembly Design Recipe from class. You can optimize your code once you've written it – if, for example, there is some repetition that can be trivially removed.

Task 0: Set up your VM

Follow the instructions at https://github.khoury.northeastern.edu/cs3650/lab01/raw/main/ExternalResources/XenOrchestra-Access-Sp23.pdf to set up a XOA VM. Note, if the network interface `bond0` isn't available, use `STT-VMs`. Make sure you correctly add your *public key*, which you will pass to SSH when connecting to the VM. Some SSH implementations expect the public key after the `-i` switch (with the `.pub`) extension, but some want the private key file (without the `.pub` extension). Try both if you are still being asked for a password for the VM user. The only password you should have to enter is the passkey that unlocks your key.

You do not need to submit anything to show you have the VM, but from the next assignment, we will start pretending `login.khoury.northeastern.edu` does not exist anymore.

Task 1: Compare

We have provided an implementation of a simple function in C: `long compare(long, long)`. This function takes two signed long integers and returns -1, 0, or 1, based on whether the first argument is *less* than the second, they are *equal*, or the first argument is *greater* than the second.

Your task is to complete the `compare` program by implementing the `main` function in `compare-main.s`, which needs to call the provided `compare` function. This is also called a "driver". When compiled, the program should have the following features and behaviour:

1. Accept exactly two arguments. You can assume that, when provided, these arguments are valid signed long integers.
2. The provided compare function should be called with these two numbers, converting them from strings as necessary.
3. Based on the result from the function, the program should print one of the following strings (with a newline at the end) and exit with an exit status of `0`:
   - `less`, if the first command-line argument is less than the second
   - `equal`, if they are equal
   - `greater`, if the first argument is greater than the second.
4. If fewer or more than 2 arguments are provided, the program should print "`Two arguments required.`", followed by a newline, and exit with a status of `1`.

First, write the `main` function in C and save it as `compare-main.c`, where we provided a "stub" for you. The program does not have to compile, but it should be a fairly accurate high-level representation of the assembly program. Do not spend too much time on this, but give it your best shot. Try to use your knowledge of Java and provide comments if you are struggling with C.

Second, implement an assembly version of the `main` function in `compare-main.s`. **Your program must compile without any modification to the provided `compare.c` file, using the provided Makefile.** The command `make compare` will combine `compare.c` (provided by us) and `compare-main.s` (written by you) into the executable `compare`.

Sample interactions with `compare`:

```
$ ./compare 1 2
less
$ ./compare 2 1
greater
$ ./compare -12 -12
equal
$ ./compare 1
Two arguments required.
```

Task 2: The Maximum of an Array

The second task is to write the function:

```c
unsigned long array_max(unsigned long n, unsigned long *items)
```

This function will return the maximum value of an array of long integers ≥ 0. The first argument provided is the number of elements, the second argument is the address of the first element.

You do not need to write a C version of this function, but we recommend doing so.

We have provided the driver program in `array-max-main.c`, which processes the command line arguments and calls the `array_max` function. The Makefile will compile both `array-max-main.c` and your implementation of the `array_max` function to produce the executable `array-max`. Once compiled, the interactions with `array-max` should look as follows:

```
$ ./array-max 1 2
2
$ ./array-max 42 1
42
$ ./array-max 3 1 5 8 2 4 8 20 1
20
```

Your program must compile without any modification to the provided `array-max-main.c` file, using the provided Makefile. The command `make array-max` will combine `array-max-main.c` (provided by us) and `array-max.s` (written by you) into the executable `array-max`.

Using the Makefile

We have provided a Makefile for you. You can use it as follows on the command line:

- `make`: compile all programs (`compare`, `array-max`)
- `make clean`: basic cleanup, remove binaries
- `make compare`: compile the `compare` program using `compare-main.s` and `compare.c`
- `make array-max`: compile the `array-max` program using `array-max.s` and `array-max-main.c`

Deliverables

Task 1 Modify the files `compare-main.s` (implementation) and `compare-main.c` (blueprint) and commit them to your repository.

Figure 1: `compare`

Task 2 Modify the file `array-max.s` and commit it to your repository.

- Do not include any executables, object files, or any other binary, intermediate or hidden files.

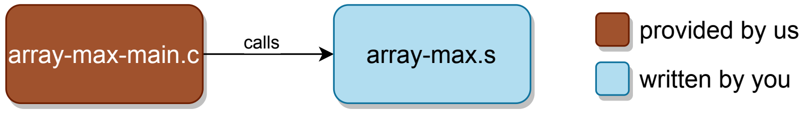

Figure 2: `array-max`

- Finally, go to our Gradescope and submit a ZIP archive of your repository (which can be downloaded from Github).

Note

While inspecting C compiler output to learn about assembly is a good approach, you are not, under any circumstances, allowed to submit compiler (or any other machine-generated) output as your `.s` files. Doing so will result in an automatic 0 for the assignment. The code must be clearly written by you, following the Assembly Design Recipe

Hints and Tips

- Read the assignment carefully and make sure you understand what is being asked in each Task.
- Start early. This doesn't mean you start writing code right away, but you should at least read the description, clone the assignment repository, and look for "missing links".
- Make sure you understand the provided starter code. Ask questions early if not.
- Each argument to a program is a *string*, that is, the value is actually a memory address pointing to the first character of the string in memory.
- The above is reflected in the signature of `main`, which is `int main(int argc, char *argv[])` – `argc` is the argument count, and `argv` is an array of strings. Each string is an array of characters, ending with a *null byte/character* (i.e., a character whose numeric ASCII value is 0). An array is represented as the address of (= a pointer to) its first element.
- The first element of `argv` is always the path and name of the executable. This means the actual arguments start at `argv[1]`.
- The return value of `main` is the exit code or exit status of the program. We return 0 by default to signal success.
- You can use standard C functions like `atol` where needed.
- Pay close attention to assembly calling conventions and the use of registers when calling C functions, or when writing functions.
- Use examples from the lectures and the labs to help you get unstuck and ask questions.
- Learn more about Makefiles: https://makefiletutorial.com/