Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue 10:00-11:50 @UTS 203, Labs: Tue 12:00-13:50 & Wed 12:00-13:50 @FAB 88-10

Lab 6: X86-64 Assembly Language Programming

(Adapted from Prof. Mark Jones' earlier version)

In this lab, you'll practice x86-64 assembly language programming by implementing a set of examples. If you're not familiar with x86-64 assembly language programming, or even if you're just a bit rusty, then these examples should help to get you started, and also to give you some ideas and programming tips.

The purpose of these examples is to build familiarity and confidence with the basics of programming in x86-64 assembly language. Clarity and correctness are our primary goals. In particular, we're not looking to demonstrate mastery of the full instruction set and we're not going to worry about performance or code size.

The Examples

In each of these examples, the goal is to write a fragment of x86-64 code that takes the address of an array of (32 bit/4 byte) integers as its input in the `rdi` register and returns a result in the `eax` register. In each case, we will assume that the elements of the array are stored in successive locations in memory, each 4 bytes wide, with the first entry at the address that is provided in `rdi`. We will also assume that the array is terminated with a zero and that it contains at least one element before that terminating zero is reached.

- *Example 1* — Return the length of the input array in `eax`.
- *Example 2* — Return the largest number from the array in `eax`.
- *Example 3* — Return the position of the largest number in the array in `eax`.
- *Example 4* — Return the average value of the numbers in the array using integer division and ignoring any remainder.
- *Example 5* — Reverse the order of the elements in the array, without using any additional storage.
- *Example 6* — Sort the elements in the array into increasing numerical order, without using any additional storage. (No algorithmic sophistication should be expected here!)

X86-64 Register Use Convention

There are 16 general purpose registers that can be used in x86-64 code. We adopt the conventions of the *"System V Application Binary Interface (ABI)"* for how these registers can be used in the code sequences that we write:

- The input parameter is in `rdi`.
- `rsp` holds the stack pointer and should not be used for other purposes.
- If the code uses `rbx`, `rbp`, `r12`, `r13`, `r14`, or `r15`, then it *must* restore them to their original values before it ends. (One way to accomplish this is to `pushq` the register value on to the stack at the beginning of the code and then `popq` it off at the end. Fortunately, there are plenty of registers for simple tasks, so we won't need to do this too often; indeed, we won't need it at all for these examples.)
- The code can freely use the following registers:

```
rax                     (return result)
rsi, rdx, rcx, r8, r9   (arguments 2-6)
r10, r11                (caller saved)
```

Also remember that each of the registers named above holds a 64 bit value, which is appropriate when we're working with pointers or addresses or 64 bit integers. For much of the work that we'll do here, however, we'll be using C integers, which are 32 bits wide. Fortunately, it is easy to use each of the general purpose registers (`rax`, `rsi`, `rdx`, `rcx`, etc.) as if they were 32 bit registers instead (`eax`, `esi`, `edx`, `ecx`, etc.). In other words, we replace the `"r"` prefix on each 64 bit register name with an `"e"` to get the name of the corresponding 32 bit register. Remember also that instructions producing 32 bit results typically add an `"l"` (for `"long"`) suffix, while those producing 64 bit results typically add a `"q"` suffix. Thus, you will see instructions like `"movl %eax,%edx"` to copy the value from register `eax` into register `edx`, but an instruction like `"movq %rdi,%rdx"` is needed to copy the value from `rdi` into `rdx`.

Testing

For testing purposes, a `main.c` program is provided that runs the assembly language code on three arrays and also shows the contents of the array before and after the call. The latter is particularly useful for Examples 5 and 6 that modify the array but don't return a useful result.

You can compile and run an example program using the following lines (replacing `N` with the appropriate example number):

```
linux> gcc -o exampleN main.c exampleN.s
./exampleN
```

Alternatively, you can use the provide `Makefile`:

```
linux> make exampleN
./exampleN
```