CS 367 - Recitation #4 **Assembly Programming**

You will need access to Zeus for this activity.

Name_____ **G#**_____

Adilet Kuroda g01253384

Group Member Name: _____

Nicholas Thor Page

Group Member Name: _____

Leighton Le

Today's Goals: We want to get comfortable with addressing modes (x86-64), basic Assembly code, and using GDB with Assembly.

Work in groups of 2-3 students. Every group will turn in what they've got to Blackboard.

Grading is based on participation. Get as much done as you can. You will also be given feedback in the form of a 'score' (1-3) and possibly some comments. This doesn't affect your grade – it is solely for feedback. A score of 3 means everything looks great. A score of two indicates some minor problems. And a score of one indicates that there were some major issues. If you get a 1, don't panic - go see your prof or a GTA to get more extensive feedback.

This is similar to Exercise 3.1 in the Textbook.

Register	Initial Value
%rax	**0x100**
%rsi	**0x8**
%rdi	**0x2**

Memory Location	Initial Value
M[0x100]	**0xabcd1234**
M[0x104]	**0x11223344**
M[0x108]	**0x55667788**
M[0x10C]	**0x12345678**
M[0x110]	**0x9abcdef0**

Using the initial values at the top, fill in the associated 32-bit values in the following table:

Operand	Value
%rax	0x100
(%rax)	0xabcd1234
8(%rax)	0x55667788
(%rax, %rsi)	0x55667788
8(%rax, %rdi, 2)	0x12345678
0x100(%rsi, %rdi, 4)	0x9abcdef
0xF8(,%rdi,8)	0x55667788

Reading Assembly Code

Before we write assembly, we will practice reading assembly code.

For the given functions below, what do you think each function is computing? **Read the code and figure out**: given each of the following initial register contents, what is the final value of `%rax` for each function? **Fill in `%rax` in the chart as your answer.**

After manually reading the code and filling the charts, you can run the functions printed below to double check your answer! Download code from blackboard and:

- Copy the tarfile to zeus and unpack it (`tar xvf rec4.tar`).
- Recompile and run the main program on any of the functions

```
zeus-prompt$ make
zeus-prompt$ ./rec4 func1 10        # func1 has one int argument (%rdi)
90
zeus-prompt$ ./rec4 func2 2 4 6 8   # func2 has four int arguments (%rdi, %rsi, %rdx, %rcx)
22
zeus-prompt$ ./rec4 func3 5 6 7     # func3 has three int arguments (*%rdi, *rsi, rdx)
27
```

- Note that you can change the arguments to initialize the listed registers/memory locations with different values. We will explain the mapping soon in class.

Func 1

Register	Value
%rdi	8
%rax	72

```
func1:
      movq %rdi, %rax
      shlq $3, %rax
      addq %rdi, %rax
      ret
```

Func 1

Register	Value
%rdi	50
%rax	450

Func 2

Register	Value
%rdi	10
%rsi	8
%rdx	3
%rcx	5
%rax	88

```
func2:
      imulq %rsi, %rdi
      addq %rdx, %rcx
      movq %rcx, %rax
      addq %rdi, %rax
      ret
```

Func 2

Register	Value
%rdi	1000
%rsi	10
%rdx	100
%rcx	1
%rax	10101

Func 3

Register	Value
%rdi	0x100
%rsi	0x108
%rdx	2
%rax	0x42

Func 3

Register	Value
%rdi	0x108
%rsi	0x100
%rdx	8
%rax	0x110

Memory Location	Initial Value
M[0x100]	0x10
M[0x108]	0x99

```
func3:
      movq (%rdi), %rcx
      movq (%rsi), %rax
      addq $1, %rax
      movq %rax, (%rdi)
      imulq $4,  %rcx
      addq %rdx, %rcx
      movq %rcx, %rax
      movq %rax, (%rsi)
      ret

# -Note: When run func3 w/ provided program,
# arguments are used to set the values stored in
# memory locations pointed by %rdi and %rsi.
# -How do the contents of the two memory locations
# change?
```

Instructions Crib Sheet

Instructions usually need to know the width of data to operate (how many bytes).

`b`	byte	1 byte	
`w`	"word" size	2 bytes	(thanks to old systems!)
`l`	long	4 bytes	(common for 32-bit systems)
`q`	quadword	8 bytes	(common for now)

below, *`S`* means "source", *D* means "destination", `R` means "register".

arguments can be:

- immediate (S's only): `$3`, `$0xA5`, `$-5`, etc.
- registers: `%rax`, `%rdi`, `%r8`, etc.
- addresses: `offset(Rb, Ri, scale)`
  scale: `1`, `2`, `4`, `8` only!
  •

`movq S, D` overwrite `D` with `S`.
`leaq S, D` overwrite `D` with `S`.

bitwise operators

`andq S, D` AND `D` with `S` and store result in `D`
`orq S, D` OR `D` with `S` and store result in `D`

shifting:
`shrq R` logical right shift by `R`
`sarq R` arithmetic right shift by `R`
`shlq R` left shift by `R`

increment/decrement:
`incq D` `decq D`

`addq S, D` D = D + S
`subq S, D` D = D – S (backwards…)
`imulq S, D` D = D * S

`ret` leave. Assume answer in `%rax`

Running GDB

In the remaining time, we would like you to get some baseline experience using GDB (or refreshing your memory, if you've used it before).

GDB is a debugger program. As such, you can do the usual things: set breakpoints, step into/over/out, inspect memory, even modify current memory. You fire up gdb and get dumped into an interactive session, where you can set things like breakpoints, and then begin running code and debug it.

The class Piazza page has a nice GDB reference that you can use as a reference.
http://csapp.cs.cmu.edu/3e/docs/gdbnotes-x86-64.pdf

You should explore this more interactively, but here are some commands you should try and see what they do.

`zeus$ gdb rec4`	start up gdb; only indicate the executable, don't feed it arguments just yet.
`(gdb) break func2`	set a break point before we begin running code
`(gdb) run func2 13 17 19 23`	feed all command-line args, and begin running.
`(gdb) x/7i $rip`	examine 7 instructions, starting at %rip
`(gdb) printf "rdi=%d dsi=%d\n", $rdi, $rsi` `(gdb) printf "rdx=%d rcx=%d\n", $rdx, $rcx`	good old printf is ready for us (minus parentheses)
`(gdb) info registers`	print a lot of details about ***all*** registers
`(gdb) display/5i $rip`	every gdb operation, please print this stuff
`(gdb) si`	step-into one assembly-instruction. Pairs up nicely with x/#i style display.

There are of course many more instructions! Look at the linked guide for more ideas of what to explore. This is absolutely the practice you need to get comfortable for next week's recitation (writing assembly by hand) and for project 2 (reading assembly with gdb as basically your only assistance).

Try running func3 with different inputs and see that different instructions are executed depending on whether the first or second value is the largest.