CS 449 Summer 2019
WA3 – GDB and x86-64 Assembly

x86-64 Assembly Language

Assembly language is a human-readable representation of machine code instructions (generally a one-to-one correspondence). Assembly is machine-specific because the computer architecture and hardware are designed to execute a particular machine code instruction set.

x86-64 is the primary 64-bit instruction set architecture (ISA) used by modern personal computers. It was developed by Intel and AMD and its 32-bit predecessor is called IA32. x86-64 is designed for complex instruction set computing (CISC), generally meaning it contains a larger set of more versatile and more complex instructions.

For this course, we will utilize only a small subset of x86-64's instruction set and omit floating point instructions.

x86-64 Instructions

The subset of x86-64 instructions that we will use in this course take either one or two operands, usually in the form: `instruction operand1, operand2`. There are three options for operands:

- Immediate: constant integer data (*e.g.* `$0x400`, `$-533`) or an address/label (*e.g.* `Loop`, `main`)
- Register: use the data stored in one of the 16 general purpose registers or subsets (*e.g.* `%rax`, `%edi`)
- Memory: use the data at the memory address specified by the addressing mode `D(Rb,Ri,S)`

The operation determines the effect of the operands on the processor state and has a suffix ("`b`" for byte, "`w`" for word, "`l`" for long, "`q`" for quad word) that determines the bit width of the operation. Sometimes the operation size can be inferred from the operands, so the suffix is omitted for brevity.

Control Flow and Condition Codes

Internally, condition codes (Carry, Zero, Sign, Overflow) are set based on the result of the previous operation. The `j*` and `set*` families of instructions use the values of these "flags" to determine their effects. See the table provided on your reference sheet for equivalent conditionals.

An *indirect jump* is specified by adding an asterisk (*) in front of a memory operand and causes your program counter to load the address stored at the computed address. (e.g. `jmp *%rax`) This is useful for switch case statements

Procedure Basics

The instructions `push`, `pop`, `call`, and `ret` move the stack pointer (`%rsp`) automatically.

`%rax` is used for the return value and the first six arguments go in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
("**Di**ane's **Si**lk **D**ress **C**ost **$89**").

x86 instructions	English equivalent
`movq $351, %rax`	Move the number 351 into 8-byte (quad) register "rax"
`addq %rdi, %rsi`	
`movq (%rdi), %r8`	
`leaq (%rax,%rax,8), %rax`	

Exercises:

1. Symbolically, what does the following code return?

```
movl   (%rdi), %eax        #  %rdi  ->  x
leal  (%eax,%eax,2),  %eax  #  %rax  ->  r
addl %eax, %eax
andl  %esi, %eax             # %rsi -> y
subl  %esi, %eax
ret
```

2. Convert the following C function into x86-64 assembly code. You are not being judged on the efficiency of your code – just the correctness.

```
long happy(long *x, long y, long z) {
    if (y > z)
        return z + y;
    else
        return *x;
}
```

3. Write an equivalent C function for the following x86-64 code:

```
mystery:
    testl   %edx, %edx
    js      .L3
    cmpl    %esi, %edx
    jge     .L3
    movslq  %edx, %rdx
    movl    (%rdi,%rdx,4), %eax
    ret
.L3:
    movl    $0, %eax
    ret
```

4. Consider the following x86-64, (partially blank) C code, and memory diagram. Addresses and values are 64-bit. Fill in the C code based on the given assembly.

```
foo:
  movl    $0,      %eax

L1:
  testq   %rdi,    %rdi
  je      L2
  movq    (%rdi),  %rdi
  addl    $1,       %eax
  jmp     L1

L2:
  ret
```

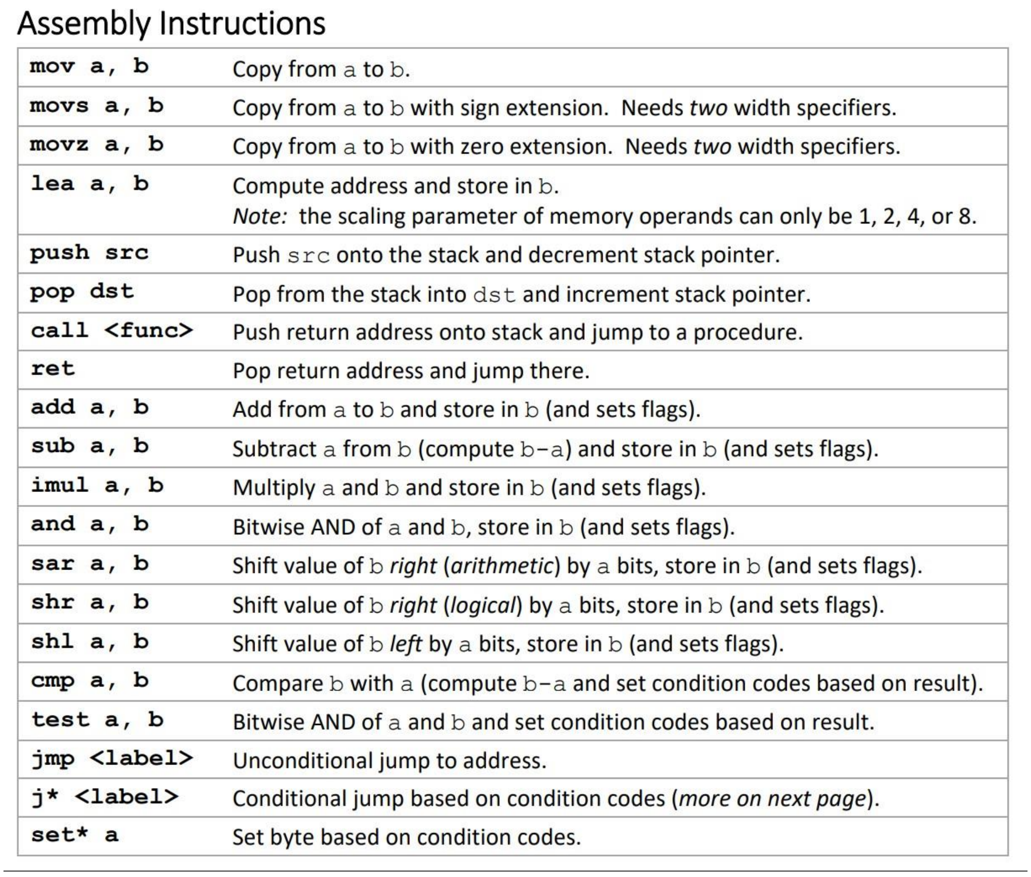
```
int foo(long* p) {
  int result =_____;
  while (_________) {
    p =______________;
    _______=_____________;
  }
  return result;
}
```

Part 2: Follow the execution of foo in assembly, where `0x1000` is passed in to `%rdi`

Write the values of `%rdi` and `%eax` in the columns. If the value doesn't change, you can leave it blank

Instruction	%rdi (hex)	%eax (decimal)
movl	0x1000	0
testq		
je		

Address	Value
0x1000	0x1030
0x1008	0x1020
0x1010	0x1000
0x1018	0x0000
0x1020	0x1030
0x1028	0x1008
0x1030	0x0000
0x1038	0x1038
0x1040	0x1048
0x1048	0x1040



Assembly Instructions

mov a, b	Copy from a to b.
movs a, b	Copy from a to b with sign extension. Needs *two* width specifiers.
movz a, b	Copy from a to b with zero extension. Needs *two* width specifiers.
lea a, b	Compute address and store in b. *Note:* the scaling parameter of memory operands can only be 1, 2, 4, or 8.
push src	Push src onto the stack and decrement stack pointer.
pop dst	Pop from the stack into dst and increment stack pointer.
call <func>	Push return address onto stack and jump to a procedure.
ret	Pop return address and jump there.
add a, b	Add from a to b and store in b (and sets flags).
sub a, b	Subtract a from b (compute b–a) and store in b (and sets flags).
imul a, b	Multiply a and b and store in b (and sets flags).
and a, b	Bitwise AND of a and b, store in b (and sets flags).
sar a, b	Shift value of b *right* (*arithmetic*) by a bits, store in b (and sets flags).
shr a, b	Shift value of b *right* (*logical*) by a bits, store in b (and sets flags).
shl a, b	Shift value of b *left* by a bits, store in b (and sets flags).
cmp a, b	Compare b with a (compute b–a and set condition codes based on result).
test a, b	Bitwise AND of a and b and set condition codes based on result.
jmp <label>	Unconditional jump to address.
j* <label>	Conditional jump based on condition codes (*more on next page*).
set* a	Set byte based on condition codes.

The GNU Debugger (GDB)

The GNU Debugger is a powerful debugging tool that is critical to Lab 3 and is a useful tool to know as a programmer moving forward. There are tutorials and reference sheets available on the course webpage, but the following tutorial should get you started with the basics:

GDB Tutorial:

1) Download `calculator.c` from the class webpage if you didn't already have it from Section 1:

```
> wget https://www.dropbox.com/s/xz03uiospggtgoz/calculator.c
```

2) Compile the file *with debugging symbols* (`-g` flag):

```
> gcc -g -o calculator calculator.c
```

3) Load the binary (executable) into GDB. This will spit out a bunch of information (e.g. version, license).

```
> gdb calculator
```

4) Inside of GDB, use the run command (**run** or just **r**) to execute your program. By default, this will continue until an error or breakpoint is encountered or your program exits.

   a. Command-line arguments can be passed as additional arguments to **run**:

   ```
   (gdb) run 3 4 +
   ```

   b. To step through the program starting at `main()` instead, use the start command (**start** or just **sta**):

   ```
   (gdb) start
   ```

5) To view *source* code while debugging, use the list command (**list** or just **l**).

   a. You can give list a function name ("list <function>") to look at the beginning of a function.

   ```
   (gdb) list main
   ```

   b. You can give list a line number ("list <line>") to look at the lines *around* that line number, or provide a specific range ("list <start>, <end>").

   ```
   (gdb) list 45
   (gdb) list 10, 15
   ```

   c. **"list"** will display the next 10 lines of code *after* whatever was last displayed and "**list –**" will display the previous 10 lines of code before whatever was last displayed.

C

6) To view *assembly* code while debugging, use the disassemble command (**disassemble** or just **disas**).

   a. "**disas**" will display the disassembly of the current function that you are in.

   b. You can also disassemble specific functions.

   ```
   (gdb) disas main
   (gdb) disas print_operation
   ```

x86-64

7) Create breakpoints using the break command (**break** or **b**)

   a. A breakpoint will stop program execution *before* the shown instruction has been executed!

   b. You can create a breakpoint at a function name, source code line number, or assembly instruction address. The following all break at the same place:

   ```
   (gdb) break main
   (gdb) break 34
   (gdb) break *0x4005d5
   ```

   c. Each break point has an associated number. You can view your breakpoints using the info command (**info** or just **i**) and then enable (**enable** or just **en**) or disable (**disable** or just **dis**) specific ones.

   ```
   (gdb) info break
   (gdb) disable 3
   (gdb) enable 3
   ```

8) Navigating source code within GDB is done while program execution is started (**run** or **start**), but halted (e.g. at a breakpoint).

a. Use the next command (**next** or just **n**) to execute the next # of lines of *source* code and then break again. This will complete ("step *over*") any function calls found in the lines of code.

```
(gdb) next
(gdb) next 4
```

b. Use the step command (**step** or just **s**) to execute the next # of lines of *source* code and then break again. This will step *into* any function calls found in the lines of code.

```
(gdb) step
(gdb) step 4
```

C

c. Use the "next instruction" command (**nexti** or just **ni**) to execute the next # of lines of *assembly* code and then break again. This will complete ("step *over*") any function calls.

```
(gdb) nexti
(gdb) nexti 4
```

d. Use the "step instruction" command (**stepi** or just **si**) to execute the next # of lines of *assembly* code and then break again. This will step *into* any function calls.

```
(gdb) stepi
(gdb) stepi 4
```

x86-64

e. Use the finish command (**finish** or just **fin**) to step *out* of the current function call.

f. Use the continue command (**continue** or just **c**) to resume continuous program execution (until next breakpoint is reached oryour program terminates).

9) You can print the current value of variables or expressions using the print command (**print** or just **p**):

a. The print command can take an optional format specifier: /x (hex), /d (decimal), /u (unsigned), /t (binary), /c (char), /f (float)

```
(gdb) print /t argc
(gdb) print /x argv
(gdb) print /d argc*2+5
(gdb) print /x $rax
```

b. The display command (**display** or just **disp**) is similar, but causes the expression to print in the specified format *every time* the programstops.

10) You can terminate the current program run using the kill command (**kill** or just **k**). This will allow you to restart execution (run or start) with your breakpoints intact.

11) You can exit GDB by either typing **Ctrl-D** or using the quit command (**quit** or just **q**)