Assignment II

Connor Taffe

January 29, 2015

The following is my report for Assignment II.

Q. 1

First I copied the file `foo-bar.c` into my directory for Assigment II.

```
$ cd courses/3380/ass2-2015/
$ ls
$ cp /tmp/foo-bar.c  .
$ ls
foo-bar.c
```

Then, I compiled it using gcc with the `-S` option to produce only assembly code output.

```
$ gcc -S foo-bar.c
$ ls
foo-bar.c  foo-bar.s
```

Q. 2

I then used the `cat` command to list the contents of `foo-bar.s`, which is the file containing the generated assembly code. For brevity I have omitted many lines of output.

```
$ cat foo-bar.s
.file "foo-bar.c"
.text
.globl main
.type main, @function
main:
leal 4(%esp), %ecx
andl $-16, %esp
pushl -4(%ecx)
pushl %ebp
... (many lines omitted)
```

Q. 3

Following are the answers for the a, b, and c subquestions.

Sub Q. A

Variable a is at an offset of -12 from `%ebp`, and variable b is at an offset of -8 from `%ebp`. I found them via the `mov` instruction that set their values from the constants 3 and 4. Since a was assigned 3, and b, 4, I was able to find their positions.

Sub Q. B

Variable c is stored at a -4 offset. These function calls are using the System V ABI, as the caller is responsible for stack cleanup. When `foo` calls `bar`, `bar`'s return value is stored in `%eax`, this is moved to a -4 offset from `%ebp`.

Sub Q. C

Variable d is at an offset of -4 from `%ebp`. The variable d is set frop the squaring of x, so I looked for a multiplication of the same reference from the stack by itself, this was saved to a -4 offset of `%ebp`, so that must be d.

Q. 4

I compiled the code as follows:

```
$ gcc -g -o foo-bar foo-bar.c
$ ls
foo-bar  foo-bar.c  foo-bar.s
```

Q. 5

I then used gdb in emacs to trace the execution of the program at the assembly code level as follows:

```
$ emacs -nw
(emacs fills the terminal with its new window)
```

After emacs loads, I used the meta-x key combo to bring up a M-x prompt.

```
M-x gdb
Run gdb (like this): gdb foo-bar
(terminal refreshes)
(gdb)
```

Following are the answers to subquestions a through q.

Sub Q. A

I then listed the main function and set a breakpoint at `w -= foo(a, b);`.

```
(gdb) list
1       int w;
2       int foo(int, int);
3       int bar(int);
4
5       int main()
6       {
7         int a = 3;
8         int b = 4;
9         w = foo(a, b);
10      }
(gdb) break 9
Breakpoint 1 at 0x8048393: file foo-bar.c, line 9.
```

Sub Q. B

I then ran the program until it stopped at a the set breakpoint.

```
(gdb) r
Starting program: /home/cptaffe/courses/3380/ass2-2015/foo-bar

Breakpoint 1, main () at foo-bar.c:9
```

Following is the accompanying output from the second screen.

```
int w;
int foo(int, int);
int bar(int);

int main()
{
  int a = 3;
  int b = 4;
=>w = foo(a, b);
}
```

Sub Q. C

I then used `disas` to disassemble the code in `main()`.

```
(gdb) disas
... (scrolled away)
0x0804837e <main+10>:   push   %ebp
```

```
0x0804837f <main+11>:   mov    %esp,%ebp
0x08048381 <main+13>:   push   %ecx
0x08048382 <main+14>:   sub    $0x24,%esp
0x08048385 <main+17>:   movl   $0x3,-0xc(%ebp)
0x0804838c <main+24>:   movl   $0x4,-0x8(%ebp)
0x08048393 <main+31>:   mov    -0x8(%ebp),%eax
0x08048396 <main+34>:   mov    %eax,0x4(%esp)
0x0804839a <main+38>:   mov    -0xc(%ebp),%eax
0x0804839d <main+41>:   mov    %eax,(%esp)
0x080483a0 <main+44>:   call   0x80483b3 <foo>
0x080483a5 <main+49>:   mov    %eax,0x8049634
0x080483aa <main+54>:   add    $0x24,%esp
0x080483ad <main+57>:   pop    %ecx
0x080483ae <main+58>:   pop    %ebp
0x080483af <main+59>:   lea    -0x4(%ecx),%esp
0x080483b2 <main+62>:   ret
End of assembler dump.
```

Sub Q. D

Then I printed the contents of `%eip`.

```
(gdb) print $eip
$2 = (void (*)()) 0x8048393 <main+31>
```

Sub Q. E

I then ran the next few instructions up to `call foo`.

```
(gdb) stepi
(gdb) stepi
(gdb) print $eip
$1 = (void (*)()) 0x8048396 <main+34>
(gdb) stepi
(gdb) stepi
(gdb) print $eip
$2 = (void (*)()) 0x804839d <main+41>
(gdb) stepi
(gdb) print $eip
$3 = (void (*)()) 0x80483a0 <main+44>
```

Sub Q. F

Following is the current stack diagram:

variable b as passed to `foo` ⟵ `esp`
variable a as passed to `foo`
... (20 bytes)
local variable b
local variable a
register `ecx`
register `ebp` ⟵ `ebp`

Sub Q. G

I then stepped into the `foo` function and used `disas` to disassemble the code.

```
(gdb) step
(could be more ouput here, but it was scrolled away)
foo (x=3, y=4) at foo-bar.c:15
(gdb) disas
Dump of assembler code for function foo:
0x080483b3 <foo+0>:     push   %ebp
0x080483b4 <foo+1>:     mov    %esp,%ebp
0x080483b6 <foo+3>:     sub    $0x18,%esp
0x080483b9 <foo+6>:     mov    0x8(%ebp),%eax
0x080483bc <foo+9>:     mov    %eax,(%esp)
0x080483bf <foo+12>:    call   0x80483d1 <bar>
0x080483c4 <foo+17>:    mov    %eax,-0x4(%ebp)
0x080483c7 <foo+20>:    mov    0xc(%ebp),%edx
0x080483ca <foo+23>:    mov    -0x4(%ebp),%eax
0x080483cd <foo+26>:    sub    %edx,%eax
0x080483cf <foo+28>:    leave
0x080483d0 <foo+29>:    ret
End of assembler dump.
```

Sub Q. H

Then I ran instructions up to the `call bar` instruction.

```
(gdb) print $eip
$7 = (void (*)()) 0x80483b9 <foo+6>
(gdb) stepi
(gdb) stepi
(gdb) print $eip
$8 = (void (*)()) 0x80483bf <foo+12>
```

Sub Q. I

The following is the current stack in diagram form.

⟵ esp
variable x, passed to `bar`
register `%ebp` ⟵ `ebp`
passed variable x
passed variable y
(main)

Sub Q. J

I then stepped into the `bar` function, dand used the `disas` command to show the disassembly.

```
(gdb) step
bar (z=3) at foo-bar.c:22
(gdb) disas
Dump of assembler code for function bar:
0x080483d1 <bar+0>:     push   %ebp
0x080483d2 <bar+1>:     mov    %esp,%ebp
0x080483d4 <bar+3>:     sub    $0x10,%esp
0x080483d7 <bar+6>:     mov    0x8(%ebp),%eax
0x080483da <bar+9>:     imul   0x8(%ebp),%eax
0x080483de <bar+13>:    mov    %eax,-0x4(%ebp)
0x080483e1 <bar+16>:    mov    -0x4(%ebp),%eax
0x080483e4 <bar+19>:    leave
0x080483e5 <bar+20>:    ret
End of assembler dump.
```

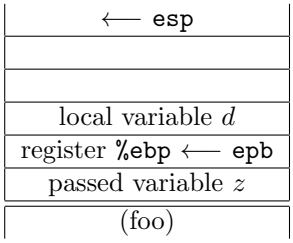
Sub Q. K

I then ran the instructions in `bar` up to the `leave` instruction.

```
(gdb) print $eip
$9 = (void (*)()) 0x80483d7 <bar+6>
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) print $eip
$10 = (void (*)()) 0x80483e4 <bar+19>
```

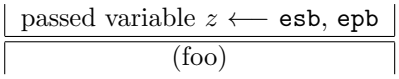
Sub Q. L

The following is the stack diagram at this point.



Sub Q. M, N, O

I then executed the `leave` instruction, the following is the stack at this point.



I then executed the `ret` instruction.

Sub Q. P

I then ran the `disas` command and viewed the disassembly.

```
(gdb) disas
Dump of assembler code for function foo:
0x080483b3 <foo+0>:     push   %ebp
0x080483b4 <foo+1>:     mov    %esp,%ebp
0x080483b6 <foo+3>:     sub    $0x18,%esp
0x080483b9 <foo+6>:     mov    0x8(%ebp),%eax
0x080483bc <foo+9>:     mov    %eax,(%esp)
0x080483bf <foo+12>:    call   0x80483d1 <bar>
0x080483c4 <foo+17>:    mov    %eax,-0x4(%ebp)
0x080483c7 <foo+20>:    mov    0xc(%ebp),%edx
0x080483ca <foo+23>:    mov    -0x4(%ebp),%eax
0x080483cd <foo+26>:    sub    %edx,%eax
0x080483cf <foo+28>:    leave
0x080483d0 <foo+29>:    ret
End of assembler dump.
```

Sub Q. Q

i. The current function is `foo`.

ii. `0x080483c4 <foo+17>:    mov    %eax,-0x4(%ebp)`

iii. The `ret` instruction sets the instruction pointer to the value dictated by the stack, or the value pushed there by `call`.