COMP2100 Workshop Ten

Machine Level Programming – Procedure calls and arrays

The first priority in tutorials is **your questions**. Come with relevant questions to help you better understand the lectures and readings.

Questions not from the text book

The X64 Assembly and ASCII Reference Sheet will be helpful with the following questions.

1. Calling conventions.
   The X64 Assembly and ASCII Reference Sheet summarises the calling conventions recently discussed in lectures. The following assembly code does not follow the calling conventions. Some comments have been added to explain the intent of the code – you should assume that any inconsistency between the comments and the assembly is a mistake. Identify the mistakes in the code and explain each mistake. Hint: You should be able to find at least three distinct important mistakes. In fact, there are at least six distinct mistakes.

```
func:                                  # long func(long x, long y)
        pushq   %rdx                   # Save rdx on the stack
        movq    %rdi, %rdx             # Save x to register rdx
        movq    %rsi, %rbx             # Save y to register rbx
        sarq    %rdi                   # Compute x>>1 as parameter
        # Call doit with y as first parameter and x>>1 as
        # second parameter.
        call    doit                   # rcx = doit(y, x>>1)
        andq    %rdx, %rcx             # rcx &= x
        addq    %rbx, %rcx             # rcx += y
        popq    %rdx                   # Restore rdx from the stack
        ret                            # return rcx
```

2. Consider the following disassembled C function `compute`:

```
1. Dump of assembler code for function compute:
2. 0x00400626 <+0>:     mov    %rdi,%rax
3. 0x00400629 <+3>:     test   %rdi,%rdi
4. 0x0040062c <+6>:     jns    0x40063e <compute+24>
5. 0x0040062e <+8>:     sub    $0x8,%rsp
6. 0x00400632 <+12>:    neg    %rdi
7. 0x00400635 <+15>:    callq  0x400626 <compute>
8. 0x0040063a <+20>:    add    $0x8,%rsp
9. 0x0040063e <+24>:    repz retq
```

   a. Write pseudo-C corresponding to each line of the function. Note: Use brief English descriptions for lines that have no C equivalent such as test of a register, and manipulating the stack pointer.

   b. Trace the execution of `compute(-3)`. Write the line number of each line that is being executed and write the values of the registers *after* execution of that line. Show what you know about condition codes and whether branches are taken or not. Take particular note of any procedure calls and returns. When a call is taken, trace the recursive call until it returns, then continue tracing the outer instance. Here is a start on your answer:

Line	rdi	rax	condition codes/notes
1.	-3	?	(This is the entry values)
2.	-3	-3	

   c. What is the final return value of `compute(-3)`?

   d. What is `compute` actually computing?

3. Consider the following snippet of C code.

```
long process (char *cp, short *sp, long *lp, long j) {
...
}
```

Each of the following assembly addressing expressions is found inside the implementation of the function `process`. Interpret each expression as a C array reference. Use the appropriate parameter pointer name in your C expression. For example, **`3(%rdi)`** is equivalent to the C array expression **cp[3]**.
The last three examples are unusual and more difficult.

```
a. 8(%rdi)
b. 0x13(%rdi)
c. (%rdi)
d. 8(%rsi)
e. 8(%rdx)
f. (%rdi,%rcx)
g. (%rsi,%rcx,2)
h. 3(%rdi,%rcx)
i. 8(%rdx,%rcx,8)
j. (%rdi,%rcx,4)
```

Practical Exercise

4. Write a C function long stringlen(char *p) that computes the length of a string p without calling strlen().
   a. Use a while loop in your function. Compile it to assembler using gcc -S. Identify the code lines that correspond to the components of a while loop – the branch over the loop and the branch that iterates the loop.
   b. Use a for loop. Compile it to assembler. Are there any differences from the code used to implement the while loop?