COMP2100 Workshop Week 9
C and Assembly

Tutorial Questions

Machine Level Programming

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

1. Suppose that the function `whatisit` is called as follows:

```
char *p;
long status;
…
status = whatisit(p);
```

Here is some of the code from inside whatisit…

```
        cmpb    $0x63,0x2(%rdi)
        jne     .L1
        cmpb    $0x64,(%rdi)
        jne     .L1
        cmpb    $0x65,0x23(%rdi)
        je      .L2
        ...
.L1:
        call    failed_the_test
        ...
.L2:
        movl    $0,%eax
        retq
```

Answer the following questions

a. What is the C type of the value passed in %rdi? It is a pointer to__________

b. Convert each of the following memory address expressions to a pseudo-C array reference expression relative to the pointer rdi. The first one has been done for you. Note: You may need to convert hexadecimal to decimal.

Assembly Expression	**C equivalent**
`(%rdi)`	`rdi[0]`
`0x2(%rdi)`	
`0x23(%rdi)`	

c. What is the C type of each of the expressions `rdi[0]` etc that are listed in part b?

d. Convert each of the following hexadecimal constants to an ASCII character.

```
0x63                    lower case c
0x64                    lower case d
0x65                    lower case e
```

e. You don't want the program to go to label .L1 but you do want it to go to label .L2. What ASCII character should be in each of the positions rdi[0], etc?. Note that some positions are unknown – one of them has been filled out below.

Position	**ASCII**
`0:`	
`1:`	`Unknown`
`2:`	
`3:`	
`4:`	
`...` *fill out any other positions that you know*	

2. Convert the following for loop to a goto representation.

```
for (j = 0; j < 8; j++)
    a[j]++;
```

Practical Exercises

1. Multiplication by a constant in machine code

To begin this practical exercise, copy the tar file /home/unit/group/comp2100/mulk.tar to your directory and extract the contents, then cd into the directory mulk that is extracted from the tar file.

There is a Makefile in the directory mulk. It compiles the program mulk.c and prints out the assembly code. You must specify a value for K on the command line to make. E.g.

```
$ make k=5
$ make k=7
etc.
```

Here is the program mulk.c. The value of K is defined on the command line.

```
int mulk(int a) {
    return a * K;
}
```

The make command prints out the assembly code. You need to read the machine instructions after the mulk: label and before the "ret" return instruction. For example, here is the output for K=17. The important lines are in bold face. Most of the lines are actually assembly directives for the linker.

```
$ make k=17
        .file   "mulk.c"
        .text
        .globl  mulk
        .type   mulk, @function
mulk:
.LFB0:
        .cfi_startproc
        movq    %rdi, %rax
        salq    $4, %rax
        addq    %rdi, %rax
        ret
        .cfi_endproc
.LFE0:
        .size   mulk, .-mulk
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
        .section        .note.GNU-stack,"",@progbits
```

Using make as shown, explore different values of K and fill out the following table, showing which instructions are executed inside mulk (excluding ret), for each value of K. Write a brief explanation of each instruction sequence. Some cases have been filled out for you, and some of the code examples are listed to save you time.

> Hint: You can do this quite quickly if you know how to use the command line history and command line editing.

K	Instructions	
1	`movq    %rdi, %rax .`	The result of multiplication by 1 is the input value passed in %rdi so move it to %rax
2	`leaq    (%rdi,%rdi), %rax`	Multiply by 2 using leaq to add %rdi to itself and store result in %rax.
3	`leaq    (%rdi,%rdi,2), %rax`	Use leaq to add %rdi to %rdi multiplied by 2 and store in %rax
4	`leaq 0(,%rdi,4),%rax`	
5	`leaq (%rdi,%rdi,4),%rax`	
6	`leaq    (%rdi,%rdi,2), %rax` `addq    %rax, %rax`	Multiply %rdi by 3 into %rax using leaq, then double %rax by adding it to itself
7	`leaq    0(,%rdi,8), %rax` `subq    %rdi, %rax`	
8		
9		
10	`leaq    (%rdi,%rdi,4), %rax` `addq    %rax, %rax`	Compute %rdi*5 using leaq and then double %rax by adding it to itself
11		
12	`leaq    (%rdi,%rdi,2), %rax` `salq    $2, %rax`	

13	`leaq (%rdi,%rdi,2), %rax` `leaq (%rdi,%rax,4), %rax`	

14	`leaq 0(,%rdi,8), %rax`
	`subq %rdi, %rax`
	`addq %rax, %rax`	

15	`movq %rdi, %rax`
	`salq $4, %rax`
	`subq %rdi, %rax`	

16	`movq %rdi, %rax` `salq $4, %rax`	Move %rdi into %rax then shift left by 4 bits to multiply by 16
32	`movq %rdi, %rax` `salq $5, %rax`	
46	`imulq $46, %rdi, %rax`	Integer multiply three-operand instruction. Multiplies immediate constant 46 by register %rdi and stores result in %rax

46 is the smallest positive integer multiplier for which the compiler generates an instruction that uses the general purpose integer multiplier (imulq instruction). What does this tell you about the performance of the integer multiplier compared to other instruction sequences?

You might find it interesting to consider other instruction alternatives where the compiler has made particular choices such as choosing between shift, add, or leaq as the way of doing a particular basic multiplication. What factors seem to influence the compiler?

Challenge question: What is the next small integer for which the compiler uses imulq? In what way is that number similar to 46?