Overview

In this second CS2002 Practical, I was asked to comment some Assembler code, as well as design and implement a StackFrame module that provides the functionality to establish the base pointer and return address in the caller's stack frame and print out stack frame data. I also had to implement test files to make sure that my implementation worked.

I'm proud to say that I've achieved the previously established goals:

- I've commented the Factorial-Commented.s file.
- I've implemented the StackFrame module.
- I've added tests.

Assembler commenting

As required in the Stage 1 of this practical's System Specification, I examined the assembly code in Factorial-Commented.s, and wrote comments for the *factorial* function. The comments are written after each line of assembly code that is not an assembler directive (that starts with a .), as show in *Figure 1*:

src > Factorial-Commented.s

```
4   factorial: # Name for place in program: factorial function
5   .LFB0:
6       .cfi_startproc
7       # Create a new stackframe with pushq, movq and subq
8       pushq   %rbp # Push the base pointer (rbp) to the stack
9       .cfi_def_cfa_offset 16
10      .cfi_offset 6, -16
11      movq    %rsp, %rbp # Set the base pointer (rbp) to the stack pointer (rsp)
12      .cfi_def_cfa_register 6
13      subq    $16, %rsp # Decrement the stack pointer (rsp) by 16, and store the result in the stack pointer (rsp): this moves the stack point
14      # Save the function's parameters
15      movq    %rdi, -8(%rbp) # Save the 1st argument (n, stored in the rdi register) to the stack, at position -8 (relative to the base pointe
16      movq    %rsi, -16(%rbp) # Save the 2nd argument (accumulator, stored in the rsi register) to the stack, at position -16 (relative to the
17      cmpq    $1, -8(%rbp) # Compares n (stored at position -8 relative to the base pointer rbp) with 1
18      ja  .L2 # Will go to section .L2 if 1 is greater than n (unsigned)
19      # If not, the following commands are executed
20      movl    $7, %edi # Store the value 7 to the low 32-bits addressable edi: this will change the value of rdi
21      call    printStackFrames # Transfers control to the printStackFrames function
22      movq    -16(%rbp), %rax # Set the return register (rax) to accumulator (stored at position -16 relative to the base pointer rbp)
23      jmp .L3 # Go to section .L3
24  .L2:
25      # Update the parameters and call the factorial function again
26      movq    -8(%rbp), %rax # Set the return register (rax) to n (stored at position -8 relative to the base pointer rbp)
27      imulq   -16(%rbp), %rax # Multiply accumulator (stored at position -16 relative to the base pointer rbp) by n (stored in the return regi
28      movq    -8(%rbp), %rdx # Set the rdx register to n (stored at position -8 relative to the base pointer rbp)
29      subq    $1, %rdx # Substract 1 to n (stored in the rdx register), and store the reult in the rdx register
30      movq    %rax, %rsi # Replace the value of the 2nd argument (accumulator, stored in the rsi register) with the value in the return regist
31      movq    %rdx, %rdi # Replace the value of the 1st argument (n, stored in the rdi register) with the value in the rdx register (value = n
32      call    factorial # Transfers control to the factorial function, but with updated parameters
33  .L3:
34      leave # Restore the values of the base pointer (rbp) and the stack pointer (rsp) at the end of the function: it "clears" the stackframe
35      .cfi_def_cfa 7, 8
36      ret # Return the value stored in the return register (rax)
37      .cfi_endproc
38  .LFE0:
39      .size   factorial, .-factorial
```

Figure 1: Some comments in Factorial-Commented.s

Note that I also decided to write comments for the *executeFactorial* function. This is not required by the System Specification, but I still decided to do so as it gave me a better understanding of Assembly and x86-64, as well as what *executeFactorial* does.

Design and Implementation

I only had to implement 4 functions in StackFrame.c:

- *getBasePointer*
- *getReturnAddress*
- *printStackFrameData*
- *printStackFrames*

The design of the functions as well as what they do was already implemented and explained in the header file, StackFrame.h. All I had to do was complete those functions.

getBasePointer uses inline assembly to get the base pointer in the stack frame of the function that called *getBasePointer*. The base pointer is stored in x86-64 in the *rbp* register. However, if the function simply accessed the value stored in *rbp*, it would return the base pointer of *getBasePointer*, not the function of *getBasePointer*. This is why I get the value stored in the register at position 0 relative to *rbp*, using inline assembly.

To get the return address, I first had to get the base pointer in the stack frame of the function that called *getReturnAddress*, using the same inline assembly command as in *getBasePointer* and storing it in the variable *base*. The return address being stored in the register with position 8 relative to the base pointer, I used the *movq* assembly operation to copy the value stored in the register with position 8 relative to *base* to a new variable *address*, which I then returned.

The next function that I had to implement was *printStackFrameData*. I started by examining the System Specification, and I managed to identify a few patterns:

1. Firstly, there are $n+2$ stack frames printed out, n being the number used in the *factorial* function. So, for a default value of $n=6$, 8 stack frames are printed out.
2. Secondly, each stack frame follows the same format: one line of data is printed out, followed by a line of "-" signs (13 of them), and then the rest of the data is printed out.
3. Thirdly, the number of lines is equal to the difference between the previous base pointer and the current base pointer, divided by 8 (as we want 8 bytes per line).
4. Fourthly, the first hexadecimal number in the first line of a stack frame is the same as the second hexadecimal number in the first line of the previous stack frame. Therefore, I deduced that in the first line of each stack frame, the first hexadecimal number is the current base pointer, and the second number is the value stored at this address, which here is the previous base pointer.
5. Finally, the 8 individual hexadecimal numbers that are printed after the "--" are identical to the second hexadecimal number in reverse order (and taking each digit in pairs).

The first thing to do in *printStackFrameData* was to compute the size of the stack frame: this is the previous base pointer minus the current base pointer. I then used a *for* loop to repeat the next steps a certain number of times, according to pattern 3. I then created a string of size 16 (17 in c, as it counts the null character) to store the first hexadecimal number: the base pointer. I used the *sprintf* method to format the string accordingly (in hexadecimal, with leading 0 if necessary). I used inline assembly to get the second hexadecimal number (the value stored at the address of the first number), formatted it in the same format as the first hexadecimal number, and then printed both numbers. The next step was to output the 8 individual hexadecimal numbers. To do so, I cycled through the return address in reverse order, two-by-two, and printed out those numbers after a "--". Finally, I had to update the previous

base pointer to the value stored in the current base pointer and increase the current base pointer by 8.

printStackFrames prints out an inputted number of stack frames starting from the caller's stack frame. I started by initialising the base pointer using *getBasePointer*. I then looped a certain number of times, according to the previously explained pattern 1. In this *for* loop, I set the previous base pointer to the previous value of the base pointer, I called the *printStackFrameData* function, and updated the base pointer to the previous base pointer (as explained in pattern 4).

Testing

The testing of my implementation can be divided into different sections:

- Running my code with a default value of $n=6$, using the Makefile and executing the TryStackFrames executable.
- Running my code with different values of n, which I did in the *Tests.c* file.
- Using disassembly with the *objdump -d TryStackFrames | less* command.

I will now explain the different steps of my testing, and why the outputs are valid.

As explained in the README.md file, I compiled and ran my code with a default value of $n=6$ using the Makefile, and this is the obtained output:

```
executeFactorial: basePointer = 7ffef594cd70
executeFactorial: returnAddress = 401164
executeFactorial: about to call factorial which should print the stack

00007ffef594cc70: 00007ffef594cc90  --  90    cc    94    f5    fe    7f    00    00
-------------
00007ffef594cc78: 000000000040118c  --  8c    11    40    00    00    00    00    00
00007ffef594cc80: 00000000000002d0  --  d0    02    00    00    00    00    00    00
00007ffef594cc88: 0000000000000001  --  01    00    00    00    00    00    00    00
00007ffef594cc90: 00007ffef594ccb0  --  b0    cc    94    f5    fe    7f    00    00
-------------
00007ffef594cc98: 00000000004011ae  --  ae    11    40    00    00    00    00    00
00007ffef594cca0: 0000000000000168  --  68    01    00    00    00    00    00    00
00007ffef594cca8: 0000000000000002  --  02    00    00    00    00    00    00    00
00007ffef594ccb0: 00007ffef594ccd0  --  d0    cc    94    f5    fe    7f    00    00
-------------
00007ffef594ccb8: 00000000004011ae  --  ae    11    40    00    00    00    00    00
00007ffef594ccc0: 0000000000000078  --  78    00    00    00    00    00    00    00
00007ffef594ccc8: 0000000000000003  --  03    00    00    00    00    00    00    00
00007ffef594ccd0: 00007ffef594ccf0  --  f0    cc    94    f5    fe    7f    00    00
-------------
00007ffef594ccd8: 00000000004011ae  --  ae    11    40    00    00    00    00    00
00007ffef594cce0: 000000000000001e  --  1e    00    00    00    00    00    00    00
00007ffef594cce8: 0000000000000004  --  04    00    00    00    00    00    00    00
00007ffef594ccf0: 00007ffef594cd10  --  10    cd    94    f5    fe    7f    00    00
-------------
00007ffef594ccf8: 00000000004011ae  --  ae    11    40    00    00    00    00    00
00007ffef594cd00: 0000000000000006  --  06    00    00    00    00    00    00    00
00007ffef594cd08: 0000000000000005  --  05    00    00    00    00    00    00    00
00007ffef594cd10: 00007ffef594cd30  --  30    cd    94    f5    fe    7f    00    00
```

```
-------------
00007ffef594cd18: 00000000004011ae  --  ae    11    40    00    00    00    00    00
00007ffef594cd20: 0000000000000001  --  01    00    00    00    00    00    00    00
00007ffef594cd28: 0000000000000006  --  06    00    00    00    00    00    00    00
00007ffef594cd30: 00007ffef594cd70  --  70    cd    94    f5    fe    7f    00    00
-------------
00007ffef594cd38: 0000000000401235  --  35    12    40    00    00    00    00    00
00007ffef594cd40: 0000000000000040  --  40    00    00    00    00    00    00    00
00007ffef594cd48: 0000000000000001  --  01    00    00    00    00    00    00    00
00007ffef594cd50: 0000000000000006  --  06    00    00    00    00    00    00    00
00007ffef594cd58: 0000000000000000  --  00    00    00    00    00    00    00    00
00007ffef594cd60: 0000000000401164  --  64    11    40    00    00    00    00    00
00007ffef594cd68: 00007ffef594cd70  --  70    cd    94    f5    fe    7f    00    00
00007ffef594cd70: 00007ffef594cd80  --  80    cd    94    f5    fe    7f    00    00
-------------
00007ffef594cd78: 0000000000401164  --  64    11    40    00    00    00    00    00
executeFactorial: factorial(6) = 720
```

Figure 2: ./TryStackFrames output

We can immediately see that the format of the output is identical to the one in the System Specification. Furthermore, all 5 previously stated patterns hold. We can also label some registers and what they contain: in each stack frame, the registers printed on the third and fourth lines are used to store the result of the recursive factorial number ($n \times accumulator$, initialised at 1) and n respectively, as well as what the different stack frames represent. Please see *Figure 3* for a visual analysis of this output.

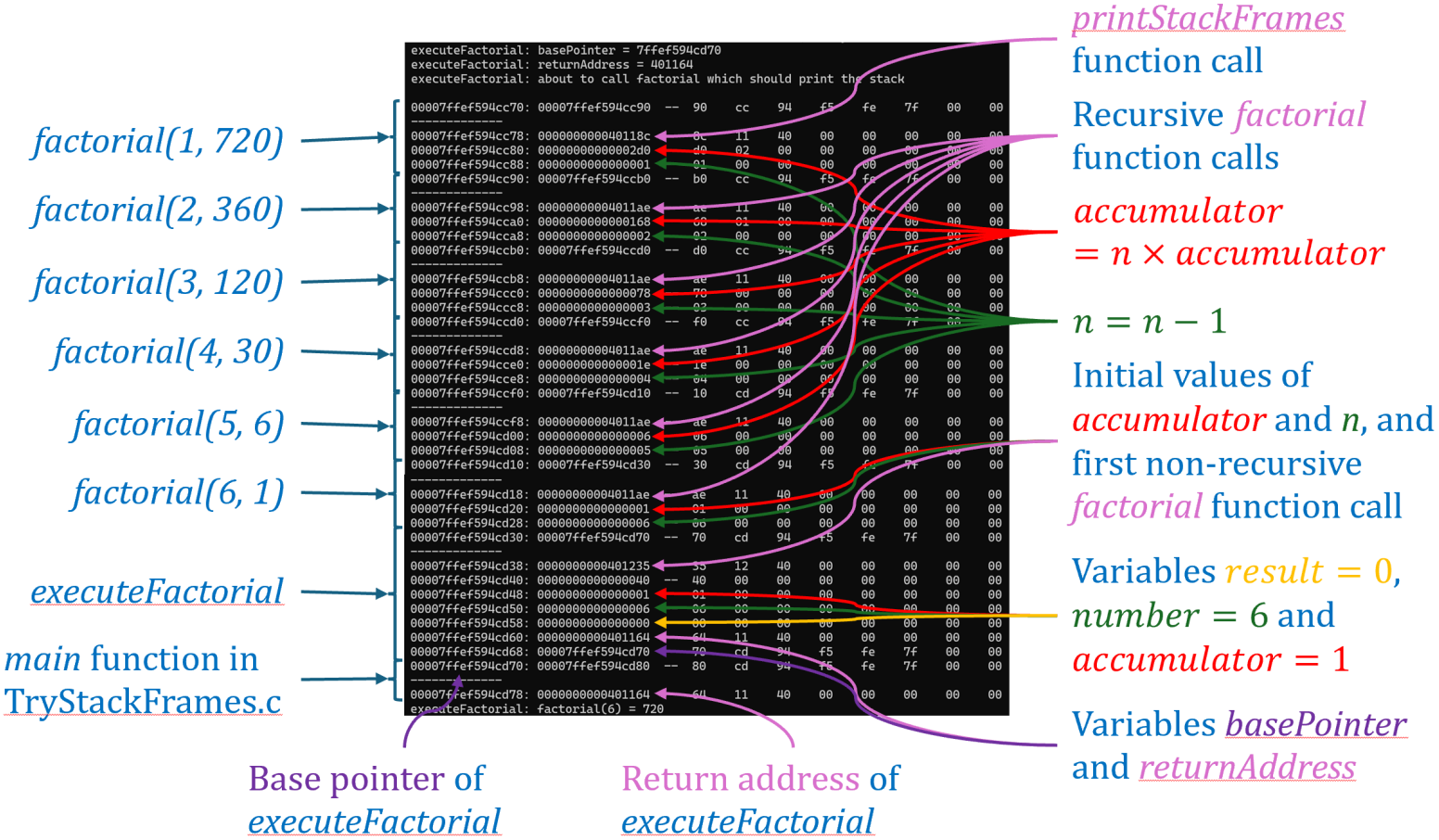

Figure 3: Visual analysis of the ./TryStackFrames output

Note that in each line, the first hexadecimal number corresponds to a register, and the second hexadecimal number is the value that is stored in this

register. Also, the addresses in pink used for the different function calls are the return addresses of those functions.

I also implemented some extra testing in a separate test file called Tests.c: it contains 3 nearly identical copies of the *factorial* and *executeFactorial* functions, except that they don't use *DEFAULT_VALUE_OF_N* but other macro variables defined in the header file Tests.h. This is done to test the stack frames for different values of *n*, not just the default value of $n=6$. I decided to implement three extra tests: one for a larger value of *n* (here $n=10$), one for a smaller value of *n* (here $n=3$), and one for $n=0$. After running it with the Makefile, we obtain the following output:

```
-- TESTING --

1. Testing for larger value of n (n = 10)
executeFactorial: basePointer = 7ffdbb35e1d0
executeFactorial: returnAddress = 401439
executeFactorial: about to call factorial which should print the stack

00007ffdbb35e050: 00007ffdbb35e070  --  70    e0    35    bb    fd    7f    00    00
-------------
00007ffdbb35e058: 0000000000401177  --  77    11    40    00    00    00    00    00
00007ffdbb35e060: 000000000375f00  --  00    5f    37    00    00    00    00    00
00007ffdbb35e068: 0000000000000001  --  01    00    00    00    00    00    00    00
00007ffdbb35e070: 00007ffdbb35e090  --  90    e0    35    bb    fd    7f    00    00
-------------
00007ffdbb35e078: 0000000000401199  --  99    11    40    00    00    00    00    00
00007ffdbb35e080: 00000000001baf80  --  80    af    1b    00    00    00    00    00
00007ffdbb35e088: 0000000000000002  --  02    00    00    00    00    00    00    00
00007ffdbb35e090: 00007ffdbb35e0b0  --  b0    e0    35    bb    fd    7f    00    00
-------------
00007ffdbb35e098: 0000000000401199  --  99    11    40    00    00    00    00    00
00007ffdbb35e0a0: 000000000093a80  --  80    3a    09    00    00    00    00    00
00007ffdbb35e0a8: 0000000000000003  --  03    00    00    00    00    00    00    00
00007ffdbb35e0b0: 00007ffdbb35e0d0  --  d0    e0    35    bb    fd    7f    00    00
-------------
00007ffdbb35e0b8: 0000000000401199  --  99    11    40    00    00    00    00    00
00007ffdbb35e0c0: 000000000024ea0  --  a0    4e    02    00    00    00    00    00
00007ffdbb35e0c8: 0000000000000004  --  04    00    00    00    00    00    00    00
00007ffdbb35e0d0: 00007ffdbb35e0f0  --  f0    e0    35    bb    fd    7f    00    00
-------------
00007ffdbb35e0d8: 0000000000401199  --  99    11    40    00    00    00    00    00
00007ffdbb35e0e0: 000000000007620  --  20    76    00    00    00    00    00    00
00007ffdbb35e0e8: 0000000000000005  --  05    00    00    00    00    00    00    00
00007ffdbb35e0f0: 00007ffdbb35e110  --  10    e1    35    bb    fd    7f    00    00
-------------
00007ffdbb35e0f8: 0000000000401199  --  99    11    40    00    00    00    00    00
00007ffdbb35e100: 00000000000013b0  --  b0    13    00    00    00    00    00    00
00007ffdbb35e108: 0000000000000006  --  06    00    00    00    00    00    00    00
00007ffdbb35e110: 00007ffdbb35e130  --  30    e1    35    bb    fd    7f    00    00
-------------
00007ffdbb35e118: 0000000000401199  --  99    11    40    00    00    00    00    00
00007ffdbb35e120: 00000000000002d0  --  d0    02    00    00    00    00    00    00
00007ffdbb35e128: 0000000000000007  --  07    00    00    00    00    00    00    00
00007ffdbb35e130: 00007ffdbb35e150  --  50    e1    35    bb    fd    7f    00    00
-------------
00007ffdbb35e138: 0000000000401199  --  99    11    40    00    00    00    00    00
00007ffdbb35e140: 000000000000005a  --  5a    00    00    00    00    00    00    00
00007ffdbb35e148: 0000000000000008  --  08    00    00    00    00    00    00    00
00007ffdbb35e150: 00007ffdbb35e170  --  70    e1    35    bb    fd    7f    00    00
-------------
00007ffdbb35e158: 0000000000401199  --  99    11    40    00    00    00    00    00
00007ffdbb35e160: 000000000000000a  --  0a    00    00    00    00    00    00    00
00007ffdbb35e168: 0000000000000009  --  09    00    00    00    00    00    00    00
00007ffdbb35e170: 00007ffdbb35e190  --  90    e1    35    bb    fd    7f    00    00
-------------
00007ffdbb35e178: 0000000000401199  --  99    11    40    00    00    00    00    00
00007ffdbb35e180: 0000000000000001  --  01    00    00    00    00    00    00    00
00007ffdbb35e188: 000000000000000a  --  0a    00    00    00    00    00    00    00
00007ffdbb35e190: 00007ffdbb35e1d0  --  d0    e1    35    bb    fd    7f    00    00
-------------
00007ffdbb35e198: 00000000004012aa  --  aa    12    40    00    00    00    00    00
00007ffdbb35e1a0: 0000000000000000  --  00    00    00    00    00    00    00    00
00007ffdbb35e1a8: 0000000000000001  --  01    00    00    00    00    00    00    00
00007ffdbb35e1b0: 000000000000000a  --  0a    00    00    00    00    00    00    00
00007ffdbb35e1b8: 0000000000000000  --  00    00    00    00    00    00    00    00
00007ffdbb35e1c0: 0000000000401439  --  39    14    40    00    00    00    00    00
00007ffdbb35e1c8: 00007ffdbb35e1d0  --  d0    e1    35    bb    fd    7f    00    00
00007ffdbb35e1d0: 00007ffdbb35e1e0  --  e0    e1    35    bb    fd    7f    00    00
-------------
00007ffdbb35e1d8: 0000000000401439  --  39    14    40    00    00    00    00    00
executeFactorial: factorial(10) = 3628800
```

Figure 4: Extract of ./Tests output (first test for $n=10$)

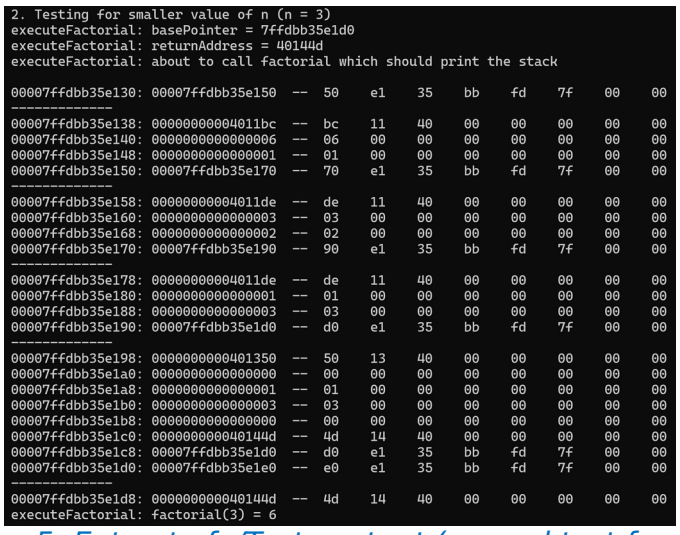

Figure 5: Extract of ./Tests output (second test for $n=3$)

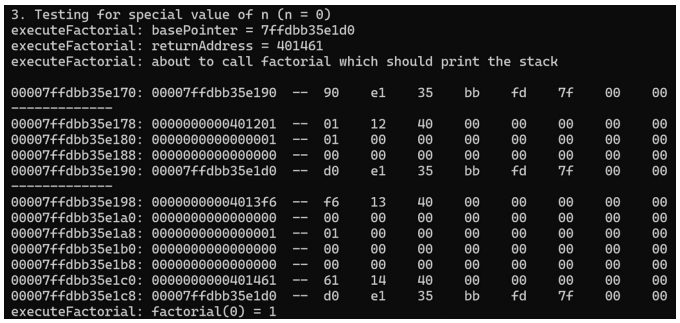

Figure 6: Extract of ./Tests output (third test for $n=0$)

The previously shown outputs follow the same format as the expected one in *Figure 2*. They also follow the same 5 patterns, and a visual analysis similar to the one done in *Figure 3* can be performed to analyse the results obtained here. One thing to notice is that there is no “last line” printed out in *Figure 6*: according to pattern 1, the number of stack frames printed out is equal to $n+2$. Therefore, as we are testing here for $n=0$, only two stack frames are printed out. And as the stack frames are outputted in “reverse order” (the *printStackFrames* call comes first, then the recursive *factorial* calls from 1 to *n*, then *executeFactorial*), the *printStackFrames* and *factorial(0, 1)* function calls (and their corresponding stack frames) will be printed out here.

Finally, I used the *objdump-d TryStackFrames | less* command to verify that my program is printing out the correct values, and this is an extract of the obtained output:

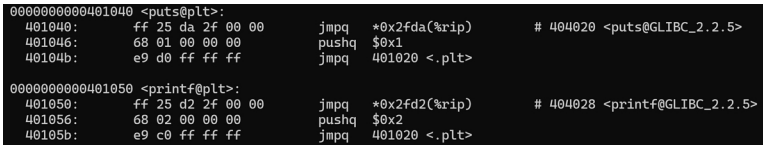

Figure 7: Disassembly output for puts and printf

```
000000000040116b <factorial>:
  40116b:       55                      push   %rbp
  40116c:       48 89 e5                mov    %rsp,%rbp
  40116f:       48 83 ec 10             sub    $0x10,%rsp
  401173:       48 89 7d f8             mov    %rdi,-0x8(%rbp)
  401177:       48 89 75 f0             mov    %rsi,-0x10(%rbp)
  40117b:       48 83 7d f8 01          cmpq   $0x1,-0x8(%rbp)
  401180:       77 10                   ja     401192 <factorial+0x27>
  401182:       bf 07 00 00 00          mov    $0x7,%edi
  401187:       e8 05 02 00 00          callq  401391 <printStackFrames>
  40118c:       48 8b 45 f0             mov    -0x10(%rbp),%rax
  401190:       eb 1c                   jmp    4011ae <factorial+0x43>
  401192:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  401196:       48 0f af 45 f0          imul   -0x10(%rbp),%rax
  40119b:       48 8b 55 f8             mov    -0x8(%rbp),%rdx
  40119f:       48 83 ea 01             sub    $0x1,%rdx
  4011a3:       48 89 c6                mov    %rax,%rsi
  4011a6:       48 89 d7                mov    %rdx,%rdi
  4011a9:       e8 bd ff ff ff          callq  40116b <factorial>
  4011ae:       c9                      leaveq
  4011af:       c3                      retq
```

Figure 8: Disassembly output for factorial

```
00000000004011b0 <executeFactorial>:
  4011b0:       55                      push   %rbp
  4011b1:       48 89 e5                mov    %rsp,%rbp
  4011b4:       48 83 ec 30             sub    $0x30,%rsp
  4011b8:       b8 00 00 00 00          mov    $0x0,%eax
  4011bd:       e8 94 00 00 00          callq  401256 <getBasePointer>
  4011c2:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  4011c6:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  4011ca:       48 89 c6                mov    %rax,%rsi
  4011cd:       bf 10 20 40 00          mov    $0x402010,%edi
  4011d2:       b8 00 00 00 00          mov    $0x0,%eax
  4011d7:       e8 74 fe ff ff          callq  401050 <printf@plt>
  4011dc:       b8 00 00 00 00          mov    $0x0,%eax
  4011e1:       e8 82 00 00 00          callq  401268 <getReturnAddress>
  4011e6:       48 89 45 f0             mov    %rax,-0x10(%rbp)
  4011ea:       48 8b 45 f0             mov    -0x10(%rbp),%rax
  4011ee:       48 89 c6                mov    %rax,%rsi
  4011f1:       bf 38 20 40 00          mov    $0x402038,%edi
  4011f6:       b8 00 00 00 00          mov    $0x0,%eax
  4011fb:       e8 50 fe ff ff          callq  401050 <printf@plt>
  401200:       bf 60 20 40 00          mov    $0x402060,%edi
  401205:       e8 36 fe ff ff          callq  401040 <puts@plt>
  40120a:       48 c7 45 e8 00 00 00    movq   $0x0,-0x18(%rbp)
  401211:       00
  401212:       48 c7 45 e0 06 00 00    movq   $0x6,-0x20(%rbp)
  401219:       00
  40121a:       48 c7 45 d8 01 00 00    movq   $0x1,-0x28(%rbp)
  401221:       00
  401222:       48 8b 55 d8             mov    -0x28(%rbp),%rdx
  401226:       48 8b 45 e0             mov    -0x20(%rbp),%rax
  40122a:       48 89 d6                mov    %rdx,%rsi
  40122d:       48 89 c7                mov    %rax,%rdi
  401230:       e8 36 ff ff ff          callq  40116b <factorial>
  401235:       48 89 45 e8             mov    %rax,-0x18(%rbp)
  401239:       48 8b 55 e8             mov    -0x18(%rbp),%rdx
  40123d:       48 8b 45 e0             mov    -0x20(%rbp),%rax
  401241:       48 89 c6                mov    %rax,%rsi
  401244:       bf a8 20 40 00          mov    $0x4020a8,%edi
  401249:       b8 00 00 00 00          mov    $0x0,%eax
  40124e:       e8 fd fd ff ff          callq  401050 <printf@plt>
```

Figure 9: Disassembly output for executeFactorial

```
0000000000401256 <getBasePointer>:
  401256:       55                      push   %rbp
  401257:       48 89 e5                mov    %rsp,%rbp
  40125a:       48 8b 45 00             mov    0x0(%rbp),%rax
  40125e:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  401262:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  401266:       5d                      pop    %rbp
  401267:       c3                      retq

0000000000401268 <getReturnAddress>:
  401268:       55                      push   %rbp
  401269:       48 89 e5                mov    %rsp,%rbp
  40126c:       48 8b 45 00             mov    0x0(%rbp),%rax
  401270:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  401274:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  401278:       48 8b 40 08             mov    0x8(%rax),%rax
  40127c:       48 89 45 f0             mov    %rax,-0x10(%rbp)
  401280:       48 8b 45 f0             mov    -0x10(%rbp),%rax
  401284:       5d                      pop    %rbp
  401285:       c3                      retq
```

Figure 10: Disassembly output for getBasePointer and getReturnAddress

For each function, the 6-digit hexadecimal number in the first line corresponds to the function's address. Therefore, the machine needs to use this address to call the function. We notice that every time the *callq* operation is used in *executeFactorial* (see *Figure 9*), the address that follows is the same as the first address in the other functions (see *Figures 7, 8* and *Figure 10*). Note that I didn't include screenshots for *printStackFrameData*, *printStackFrames*, and

the *main* function in TryStackFrames.c, but the previously explained statement also holds for these three functions.

Conclusion

All in all, I have found this practical extremely challenging at first, as I had no idea what all those hexadecimal numbers represented. I also had a difficult time understanding what was expected of me in this practical. However, when I finally grasped the meaning of those numbers, and how all parts of the code fitted together, everything became clearer, and the practical became way easier.