

Overview

For this second assessed practical of the semester, we were asked to 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. To do this, we had to use inline assembly in C using x86-64 in AT&T syntax. My submission completes all required parts of the practical.

Design

As this was my first time reading and using Assembly language, looking at the `.s` file for the first time was very disconcerting. While it was challenging at first, after extensive research on Assembly x86-64, I gained the ability to understand and read assembly language. As required by the specification, I commented each instruction for the *factorial* function succinctly explaining the purpose of the instruction in its context. To get a better understanding of the output that our program was required to produce, I also commented most instructions for the *executeFactorial* function in the `.s` file.

To implement the function inside of the `StackFrame` file, I have designed some MACRO definitions inside of the header file which are used throughout my implementation. This was mainly for good practice and to avoid magic numbers.

For the design of the *getBasePointer* function, I first decided to use the inline assembler example provided on the specification replacing the register *rax* by *rbp*. But I quickly realized that I was getting the wrong addresses. After looking at some online documentation on x86-64 assembly [1], I understood that I was fetching *getBasePointer's %rbp* and not its caller, *executeFactorial*. Hence, I choose to change the *movq* instruction's source register to `0(%rbp)`, which holds the previous value of *%rbp* (i.e. the function calling *getBasePointer*). This works since *rbp* is callee saved, hence the value held at the base pointer memory address is the previous frame's (the caller's) base pointer.

Once again, MIT's documentation on x86-64 assembly [1] was particularly useful when implementing the *getReturnAddress* function. It turns out that the return address of a frame is stored 8 bytes above its base pointer. While this confused me at first, it made a lot more sense after doing some research on the "call" assembly instruction. When a function is called in assembly language, the address of the next instruction (i.e., the return address) after the "callq" instruction is pushed onto the stack. This allows the program to return to the correct location once the called function completes its execution. So, the stack manipulation caused by "callq" occurs in the caller's stack frame, not the callee's which explains how the program output is and why the return address of a frame is always stored 8 bytes above (and not below) its base pointer.

Hence, I decided to use RIP Relative Memory Addressing mode to obtain the return address. This choice of addressing mode was perfect for this task since the code runs no matter where it is loaded in memory (i.e., the function will always return the return address of its caller). But first, my function would need to obtain the caller's base pointer address to obtain the value at an offset of plus 8 bytes. Unfortunately, I could not simply call *getBasePointer* to do this as this would have given me the base pointer for *getReturnAddress* and not *executeFactorial* (or whatever function calls *getReturnAddress*) which is the function we are interested in. Thus, instead of defining another unsigned long variable to store the caller's base pointer in an analogous manner as in *getBasePointer* and thus, wasting memory. I choose to use the register `%rax` as a clobbered operand in the inline assembly instructions making my program more efficient. The first instruction loads the caller's base pointer into the register `%rax`. Then the second instruction uses RIP Relative Addressing Mode on the base pointer in the `%rax` register to obtain the returned address stored at the memory address 8 bytes above the base pointer.

Furthermore, I designed the function that print a stack frame's data, with two for loops and a conditional statement. To design and implement the function, I used my understanding of the stack's organization. Global and local variables are stored on the stack, a region of memory that is typically addressed by offsets from the registers `%rbp` and `%rsp`. Each procedure call results in the creation of a stack frame where the procedure can store local variable and temporary intermediate values for that invocation. The stack is organized as follow:

Position	Contents	Frame
$8n+16(\%rbp)$	argument n	Previous
\dots $16(\%rbp)$	\dots argument 7	
$8(\%rbp)$	return address	Current
$0(\%rbp)$	previous <code>%rbp</code> value	
\dots $0(\%rsp)$	Locals and temps	

It immediately becomes obvious that the stack frames for the *factorial* function calls would require less lines be printed out since they have less local and temporary variables compared to the *executeFactorial* function which has many more variable such as its own base pointer and return addresses stored in unsigned long C variable.

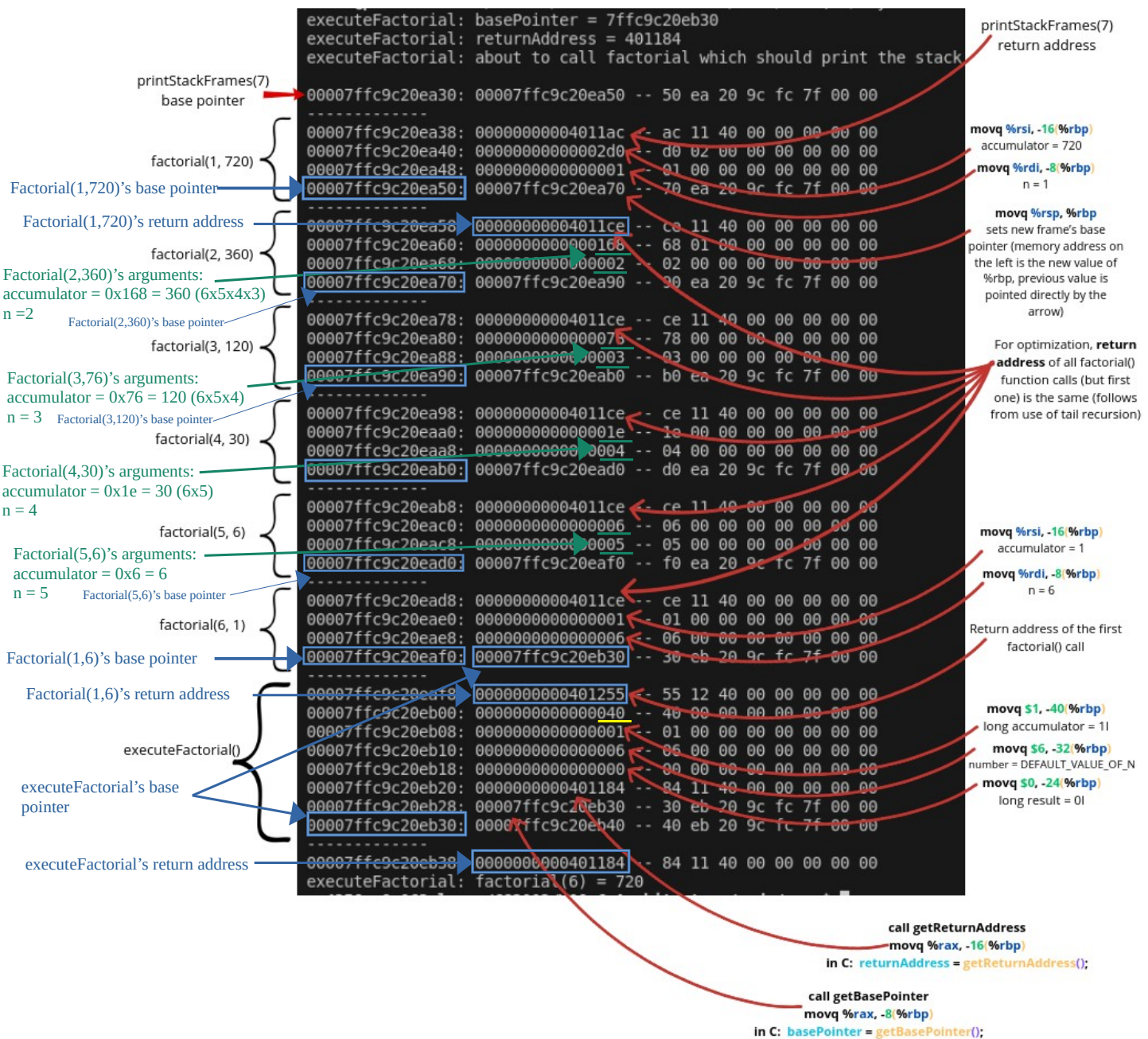
Therefore, the challenge was to design my *printStackFrameData* function such that it would not be specification specific but could adapt to output the content

of any stack frame, independently of its size. To do this, it first calculates current frame's size given by the difference of the highest memory address (previous frame's base pointer) minus the lowest memory address (current frame's base pointer). The function then loops through the frame's memory addresses delimited by the two base pointers starting at the current base pointer (lowest memory address) and going up by an offset of 8 bytes until the frame's size is reached.

To print a line, I used a single x86-64 assembly instruction which obtains the 64bit number in decimal stored at the memory address of the line. The `"movq 0(%1), %0;"` instruction moves the 64-bit (quadword) value at the memory address `"0 + basePointer + offset"` to the C variable `*ptr_numberInDec` which, is a pointer to an unsigned long variable. Then, using `sprintf` I convert and stored this value in hexadecimal in a character array (a C string). Another for loop then iterates through the string to print the 64-bit number's 8 individual bytes as separate hexadecimal numbers. Any variable used inside of the loop is dynamically allocated and memory is freed at the end of each operation.

So, each `printStackFrameData` call starts by printing the line with the current base pointer as memory address followed by its 64-bits number value (which should be previous base pointer) and the 64-bits number's 8 individual bytes as separate hexadecimal numbers. Then, it prints a dashed indicating it's printing the stack frame's body. The next line prints the memory address where is held the return address of the base pointer's frame (since it is 8 bytes above the base pointer). If any other lines are printed out, then these are should be local and temps of the previous base pointer's frame. There is a line for every memory address at 8 bytes multiples offsets from the base pointer and between the two base pointers given to the function is printed out.

Moreover, I have designed and commented a typical output of my program as required by the specification. It goes in detail over every important bit that must be explained. Anything repetitive is not describe on the image (for clarity and visibility reasons) but is explained in the paragraphs below the image containing the commented output. I have explained below, in details, what each line of the output corresponds to. On an interesting note: the 64-bit value 40 highlighted in yellow on the image below is actually compiler specific. I tried changing gcc for clang in the makefile and at the same memory address I obtained a value of 10 instead of 40.



The very first line “00007ffc9c20ea30 00007ffc9c20ea50 50 ea 20 9c ff 7c” being printed out has *printStackFrame*’s base pointer as memory address and has the very last factorial call’s base pointer as 64-bit number. In fact, we can see from the above output that *rbp* is callee saved as in every stack frame, the 64-bit number held at the base pointer memory address is the previous frame’s base pointer (which is why 0(%rbp) returns the base pointer of the previous frame).

Each stack frame is separated by dashed line. The line at the top of each stack (i.e. right under each dashed line) gives the memory address 8 bytes above the below frame’s base pointer which is where the return address of the next stack

frame is stored as the 64-bit number value. The bottom line of each stack (right above every dashed line) is the frame's base pointer as memory address followed by the previous frame's base pointer as 64-bit number. Anything in-between are local variables to the functions. For each factorial call the stack frame has 4 lines: one for where the return address of the next stack frame is stored, one for the value of accumulator (just below the line with the return address), the next is for the value of n , and finally the line with the base pointer of the frame as memory address (which holds the previous base pointer address as value).

From this output we can easily spot the value of n decreasing after each factorial call (6, 5, 4, 3, 2, 1) stored 8 bytes below the factorial calls base pointers. Meanwhile, the observed 64bit number values in hexadecimal of *accumulator* are, as expected, increasing after each factorial call (1, 6, 1e, 78, 168, 2d0) giving respectively in decimal (1, 6, 30, 120, 360, 720) which are the values one would expect for 6 factorial. These are stored 16 bytes below the factorial calls base pointers.

Indeed, the program starts printing the stack frames starting from `printStackFrame`'s caller which is `factorial(1, 720)`. It then prints the stack frames data for all the factorial call up to `factorial(6, 1)`. Then, the stack frame data for `executeFactorial` is printed out. The latter has more lines as it holds more C variables (such as its own base pointer and return address). The first line printed out is the memory address where the return address of `factorial(6,1)` is stored. The next line is where the value 40 is stored which is a decision made by the compiler. The next is where the "accumulator" variable is stored which has value 1. Then, the following line is where the value of n is stored which is 6. The next line is where the value of "result" is stored which is 0. Finally, the two next lines being printed out are the values returned by `getReturnAddress` and `getBasePointer`.

The last line (right above "`executeFactorial: factorial(6) = 720`") is the return address of `executeFactorial` which is held at the memory address 8 bytes above `executeFactorial`'s base pointer.

Since *factorial* is tail recursive, we notice that all *factorial* calls (but the first one with return address 40125) all have the same return address 4011ce. Then 4011ac is the return address of `printStackFrameData` and 401184 is `executeFactorial`'s. The return addresses can be confusing as the callee's return address is printed out as in its caller's stack frame (as a return address of a frame is always stored 8 bytes above its base pointer).

Testing

Firstly, to ensure that my program outputs the expected memory addresses and values, I have used *objdump*. Having a view of the disassemble program

with the memory layout of the program code helped me identify whether the output from my program in places matches sensible return addresses. When a function is called in assembly, the return address, which is the address of the instruction immediately following the “call” instruction, is pushed onto the stack.

```
40117f: e8 4c 00 00 00    callq 4011d0 <executeFactorial>
401184: b8 00 00 00 00    mov $0x0,%eax
```

Figure 1: Screenshot taken from disassembled program view of the main function : *executeFactorial* has return address 401184

In the above case, the “callq” instruction at “40117f” pushes the return address onto the stack and then jumps to “executeFactorial” function. When the “executeFactorial” function completes, it will then execute a “ret” instruction, which will pop the return address from the stack and jump back to that address. So, the return address will be whatever instruction comes after the “callq” instruction, in this case it is “401184”.

```
401250: e8 36 ff ff ff    callq 40118b <factorial>
401255: 48 89 45 e8       mov %rax,-0x18(%rbp)
```

Figure 2: Screenshot taken from disassembled program view of the *executeFactorial* function : *factorial(6, 1)* has return address 401255

```
4011c9: e8 bd ff ff ff    callq 40118b <factorial>
4011ce: c9               leaveq
```

Figure 3: Screenshot taken from disassembled program view of the *factorial* function calls (but first one) : return address is 4011ce.

Similarly, by looking at figures 2 and 3 I concluded that the return address of the first factorial call is 4011c9 and that the return address of all the following factorial calls is 4011ce. My submission outputs all of these return addresses correctly, as expected.

Secondly, I have designed test executables which can be generated by running the command “make Tests”. This will automatically run all the test files I have designed. I wanted to make sure that my implementation for the *StackFrame* interface can adapt to *executeFactorial* for any value of n and not only 6. Hence, I tested with n equal to 0, 1, 7 and 10 and 20 as well as 21. Running “make Tests” should print out the stack frames outputted for performing factorial 0, 1, 7, 10 and 20 instead of factorial 6. My implementation of *StackFrame.h* still outputs a similar table to the specification (with 11 stack frames this time for factorial 10, 8 stack frames for factorial 7, 2 stack frames for factorial 1, and so on). This test shows that my program is not specification specific and can adapt to print out the stack for executing the other factorial than factorial 6.

Zero and one are edge cases which should both result in calling *factorial* only once. You will find below the results I obtained.

```
./TestFactorial0
Program Output For Factorial 0:

executeFactorial: basePointer = 7ffef3c5c350
executeFactorial: returnAddress = 401283
executeFactorial: about to call factorial which should print the stack

00007ffef3c5c2f0: 00007ffef3c5c310 -- 10 c3 c5 f3 fe 7f 00 00
-----
00007ffef3c5c2f8: 0000000000401197 -- 97 11 40 00 00 00 00 00
00007ffef3c5c300: 0000000000000001 -- 01 00 00 00 00 00 00 00
00007ffef3c5c308: 0000000000000000 -- 00 00 00 00 00 00 00 00
00007ffef3c5c310: 00007ffef3c5c350 -- 50 c3 c5 f3 fe 7f 00 00
-----
00007ffef3c5c318: 0000000000401240 -- 40 12 40 00 00 00 00 00
00007ffef3c5c320: 0000000000000000 -- 00 00 00 00 00 00 00 00
00007ffef3c5c328: 0000000000000001 -- 01 00 00 00 00 00 00 00
00007ffef3c5c330: 0000000000000000 -- 00 00 00 00 00 00 00 00
00007ffef3c5c338: 0000000000000000 -- 00 00 00 00 00 00 00 00
00007ffef3c5c340: 0000000000401283 -- 83 12 40 00 00 00 00 00
00007ffef3c5c348: 00007ffef3c5c350 -- 50 c3 c5 f3 fe 7f 00 00
00007ffef3c5c350: 00007ffef3c5c360 -- 60 c3 c5 f3 fe 7f 00 00
-----
00007ffef3c5c358: 0000000000401283 -- 83 12 40 00 00 00 00 00
executeFactorial: factorial(0) = 1
```

Figure 4: Program output for *executeFactorial0* where *printStackFrame* has number equal to $(0+2)$ instead of $(0+1)$.

```
#####
./TestFactorial1
Program Output For Factorial 1:

executeFactorial: basePointer = 7ffc4998cc30
executeFactorial: returnAddress = 401283
executeFactorial: about to call factorial which should print the stack

00007ffc4998cbd0: 00007ffc4998cbf0 -- f0 cb 98 49 fc 7f 00 00
-----
00007ffc4998cbd8: 0000000000401197 -- 97 11 40 00 00 00 00 00
00007ffc4998cbe0: 0000000000000001 -- 01 00 00 00 00 00 00 00
00007ffc4998cbe8: 0000000000000000 -- 00 00 00 00 00 00 00 00
00007ffc4998cbf0: 00007ffc4998cc30 -- 30 cc 98 49 fc 7f 00 00
-----
00007ffc4998cbf8: 0000000000401240 -- 40 12 40 00 00 00 00 00
00007ffc4998cc00: 0000000000000040 -- 40 00 00 00 00 00 00 00
00007ffc4998cc08: 0000000000000001 -- 01 00 00 00 00 00 00 00
00007ffc4998cc10: 0000000000000001 -- 01 00 00 00 00 00 00 00
00007ffc4998cc18: 0000000000000000 -- 00 00 00 00 00 00 00 00
00007ffc4998cc20: 0000000000401283 -- 83 12 40 00 00 00 00 00
00007ffc4998cc28: 00007ffc4998cc30 -- 30 cc 98 49 fc 7f 00 00
00007ffc4998cc30: 00007ffc4998cc40 -- 40 cc 98 49 fc 7f 00 00
-----
00007ffc4998cc38: 0000000000401283 -- 83 12 40 00 00 00 00 00
executeFactorial: factorial(1) = 1
```

Figure 5: Program output for *executeFactorial1*

Both returned the expected stack frames and results. We can see that the returned outputs are in the same format as the one provided in the specification, with one stack frames corresponding to *executeFactorial* and the other to the single *factorial* call. This shows that my program can print the stack frames in the required format for edge cases such as 0 or 1.

To ensure that my program prints can print an odd number of stack frames (above 1) in the required format, I also tried to output the stack frames for executing factorial 7.

```
#####
./TestFactorial7
Program Output For Factorial 7:

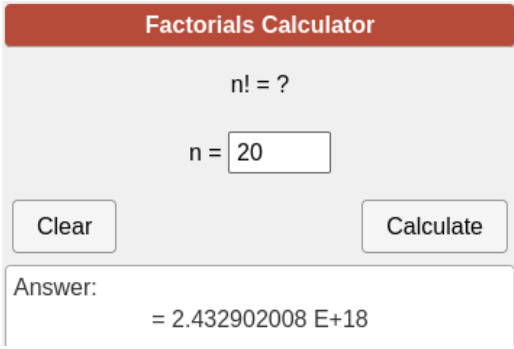
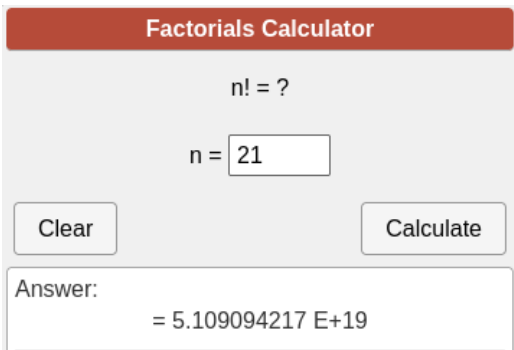
executeFactorial: basePointer = 7ffd168f0f90
executeFactorial: returnAddress = 401283
executeFactorial: about to call factorial which should print the stack

00007ffd168f0e70: 00007ffd168f0e90 -- 90 0e 8f 16 fd 7f 00 00
-----
00007ffd168f0e78: 0000000000401197 -- 97 11 40 00 00 00 00 00
00007ffd168f0e80: 00000000000013b0 -- b0 13 00 00 00 00 00 00
00007ffd168f0e88: 0000000000000001 -- 01 00 00 00 00 00 00 00
00007ffd168f0e90: 00007ffd168f0eb0 -- b0 0e 8f 16 fd 7f 00 00
-----
00007ffd168f0e98: 00000000004011b9 -- b9 11 40 00 00 00 00 00
00007ffd168f0ea0: 00000000000009d8 -- d8 09 00 00 00 00 00 00
00007ffd168f0ea8: 0000000000000002 -- 02 00 00 00 00 00 00 00
00007ffd168f0eb0: 00007ffd168f0ed0 -- d0 0e 8f 16 fd 7f 00 00
-----
00007ffd168f0eb8: 00000000004011b9 -- b9 11 40 00 00 00 00 00
00007ffd168f0ec0: 0000000000000348 -- 48 03 00 00 00 00 00 00
00007ffd168f0ec8: 0000000000000003 -- 03 00 00 00 00 00 00 00
00007ffd168f0ed0: 00007ffd168f0ef0 -- f0 0e 8f 16 fd 7f 00 00
-----
00007ffd168f0ed8: 00000000004011b9 -- b9 11 40 00 00 00 00 00
00007ffd168f0ee0: 00000000000000d2 -- d2 00 00 00 00 00 00 00
00007ffd168f0ee8: 0000000000000004 -- 04 00 00 00 00 00 00 00
00007ffd168f0ef0: 00007ffd168f0f10 -- 10 0f 8f 16 fd 7f 00 00
-----
00007ffd168f0ef8: 00000000004011b9 -- b9 11 40 00 00 00 00 00
00007ffd168f0f00: 000000000000002a -- 2a 00 00 00 00 00 00 00
00007ffd168f0f08: 0000000000000005 -- 05 00 00 00 00 00 00 00
00007ffd168f0f10: 00007ffd168f0f30 -- 30 0f 8f 16 fd 7f 00 00
-----
00007ffd168f0f18: 00000000004011b9 -- b9 11 40 00 00 00 00 00
00007ffd168f0f20: 0000000000000007 -- 07 00 00 00 00 00 00 00
00007ffd168f0f28: 0000000000000000 -- 00 00 00 00 00 00 00 00
00007ffd168f0f30: 00007ffd168f0f50 -- 50 0f 8f 16 fd 7f 00 00
-----
00007ffd168f0f38: 00000000004011b9 -- b9 11 40 00 00 00 00 00
00007ffd168f0f40: 0000000000000001 -- 01 00 00 00 00 00 00 00
00007ffd168f0f48: 0000000000000007 -- 07 00 00 00 00 00 00 00
00007ffd168f0f50: 00007ffd168f0f90 -- 90 0f 8f 16 fd 7f 00 00
-----
00007ffd168f0f58: 0000000000401240 -- 40 12 40 00 00 00 00 00
00007ffd168f0f60: 0000000000000040 -- 40 00 00 00 00 00 00 00
00007ffd168f0f68: 0000000000000001 -- 01 00 00 00 00 00 00 00
00007ffd168f0f70: 0000000000000007 -- 07 00 00 00 00 00 00 00
00007ffd168f0f78: 0000000000000000 -- 00 00 00 00 00 00 00 00
00007ffd168f0f80: 0000000000401283 -- 83 12 40 00 00 00 00 00
00007ffd168f0f88: 00007ffd168f0f90 -- 90 0f 8f 16 fd 7f 00 00
00007ffd168f0f90: 00007ffd168f0fa0 -- a0 0f 8f 16 fd 7f 00 00
-----
00007ffd168f0f98: 0000000000401283 -- 83 12 40 00 00 00 00 00
executeFactorial: factorial(7) = 5040
```

Figure 6: Program output for *executeFactorial7*

The obtained results (figure 3) ensure that my implementation can print the stack frames as expected for odd factorial number (other than 1).

I also wanted to test for value of n larger than 6, hence resulting in more stack frames being printed out. Therefore, I have designed a test for executing factorial 10 which printed out the expected 11 stack frames starting from the caller's stack frame. I also implemented a test for executing factorial 20, which is the largest factorial number that fits in a 64-bits number. Executing any factorial number above 20 will either results in an output of zero (integer overflow) or a wrong result.

Obtained	Expected
<pre>executeFactorial: factorial(20) = 2432902008176640000</pre> <p>Correct result!</p>	
<pre>executeFactorial: factorial(21) = 14197454024290336768</pre> <p>Wrong result!</p>	

This is a major flaw of the program which is caused by specification limits. Still, my program works for any value between 0 and 20 included

Evaluation

My submission implements all required parts of the specification: it implements the functions defined in the module interface *StackFrame.h* to produce the expected output. Each stack frame is printed with one line per 8 bytes, with each line giving the memory address, followed by the 4bit number given in hexadecimal and then the 8 individual bytes as separate hexadecimal numbers, giving an output similar to the specification's one. I have also commented each assembly language instruction for the function *factorial* as

required and went further by also commenting the function *executeFactorial*'s instructions. Moreover, my report contains a detailed explanation of the produced output. These are all specification requirements that my submission achieved. Furthermore, my testing shows my program's adaptability. It can print out the stack frames in the expected format when performing other factorial calculation than factorial 6.

Although my submission is not perfect, and the specification has its limitations; the way factorial and *executeFactorial* were implemented does not allow for calculating factorial n when n is above 20. This is a major flaw of the program which has a significant impact on the output produced by the functions I implemented in *StackFrame*. In fact, for values of n bigger than 20 the stack frames outputted start printing out wrong outputs or outputs of zero for the values of the accumulator and the result when the 64-bit number held by the accumulator is getting bigger than 20 factorial (20!).

Conclusion

I enjoyed this practical; I managed implement my first assembly language instructions. Although commenting the assembly file was challenging at first, understanding how programming languages are converted to lower-level instructions was highly satisfying. This practical gave me a broader understanding of programming languages, through the task of outputting functions' stack frames and a program's stack. If I had more time, I would have explored x86-64 Assembly a bit further. I wished we had more programming to perform in x86-64 rather than a limited use of inline assembly in C. Reading and writing Assembly language was surprisingly not as hard as I thought it would be and is not as scary anymore!

References

[1] <http://6.s081.scripts.mit.edu/sp18/x86-64-architecture-guide.html>