Overview

This practical tasked us with implementing functions to access a base pointer and return address, as well as print out stack frame data using inline assembly. We were provided with an existing framework containing a recursive factorial function implementation and executeFactorial method. The specifications also outlined the assembler code for specific functions, which we were required to add comments to.

We were expected to familiarise ourselves with x86-64 in AT&T syntax through reading lecture notes as well as architecture documentation and apply our understanding of stacks to the different instructions.

Design

While annotating the "Factorial-Commented.s" file, I ensured that the values specified in each register corresponded to variable names in the function and that x86-64 calling conventions were constantly referenced.

getBasePointer

The getBasePointer function was implemented using inline assembly to access the base pointer of the caller function. Breaking down the assembly code, "movq" was used to copy the value.
The %%rbp value was placed within brackets to dereference the address pointed to by the base pointer rbp, which would return the value of the caller's base pointer (omitting brackets would return the callee's base pointer). The "=r (basePointer)" portion specifies that the value should be written to a general-purpose register and stored in the C variable basePointer.

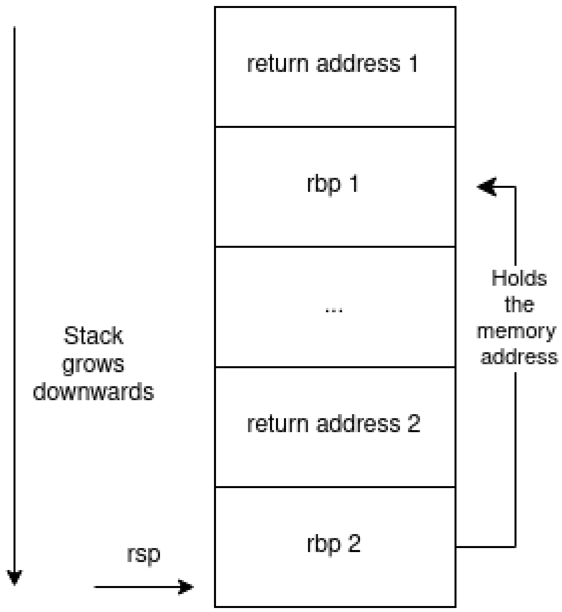

Figure 1: Relationship between saved base pointers and return address

For additional clarity, Figure 1 demonstrates how dereferencing the base pointer allows one to access the base pointer address of the caller function. The rsp register points to “rbp2”, as this is the most recently pushed value. The convention in x86-64 for a function involves pushing the existing base pointer onto the stack and overwriting the rbp register with the value of rsp (i.e. the address of “rbp1”). When the rbp register value is pushed onto the stack during a function call (as “rbp2”), it holds the address of the previous base pointer. Therefore, dereferencing “rbp2” (the address pointed to by the rbp register) allows one to access the memory address of “rbp1”.

getReturnAddress

The getReturnAddress function uses inline assembly in a similar way to getBasePointer. However, it first copies the value to the return register rax before offsetting the address by 8 bytes upwards (as seen in Figure 1) and dereferencing that address (corresponding to return address 1). The address is then stored in the “returnAddress” C variable. I experimented with storing the caller’s base pointer in a variable and then adding the constant BYTES_PER_LINE to the result, before dereferencing the value stored at the memory address pointed to by the updated variable. I found this difficult to read, especially with the casting to pointers, so I decided to implement it all using inline assembly.

printStackFrameData and printStackFrames

The printStackFrameData function prints the contents of memory addresses between two provided base pointers. It iterates over memory addresses within the stack frame using a while loop, starting with the basePointer and ending at the address before the previousBasePointer. I then cast currentAddress to a character pointer, allowing the program to access individual bytes as an array. The address of currentAddress and the value it holds are both printed using the format specifier

016lx for unsigned long hexadecimals, before printing each byte in hexadecimal format. If currentAddress is equal to the base pointer of the stack frame (at the end of the stack frame when printed), a hyphenated separator is displayed to denote the end of the stack frame.

The printStackFrames traverses up the call stack, using getBasePointer to obtain the base pointer of the current stack frame. Within a for loop, it dereferences the value held in basePointer and stores it in previousBasePointer. It loops for "number + 1" times instead of "number" iterations as it includes the current stack frame as well. Then, the printStackFrameData function is called, using the variables basePointer and previousBasePointer as parameters. The base pointer is then set to the value of the previous base pointer, allowing the function to print the next stack frame. As previousBasePointer is the basePointer of the previous stack frame, I have written printStackFrameData in such a way that it stops just before the basePointer of the previous stack frame, printing the basePointer in the next printStackFrameData call and then printing the separator so that the base pointer isn't printed as part of the next stack frame.

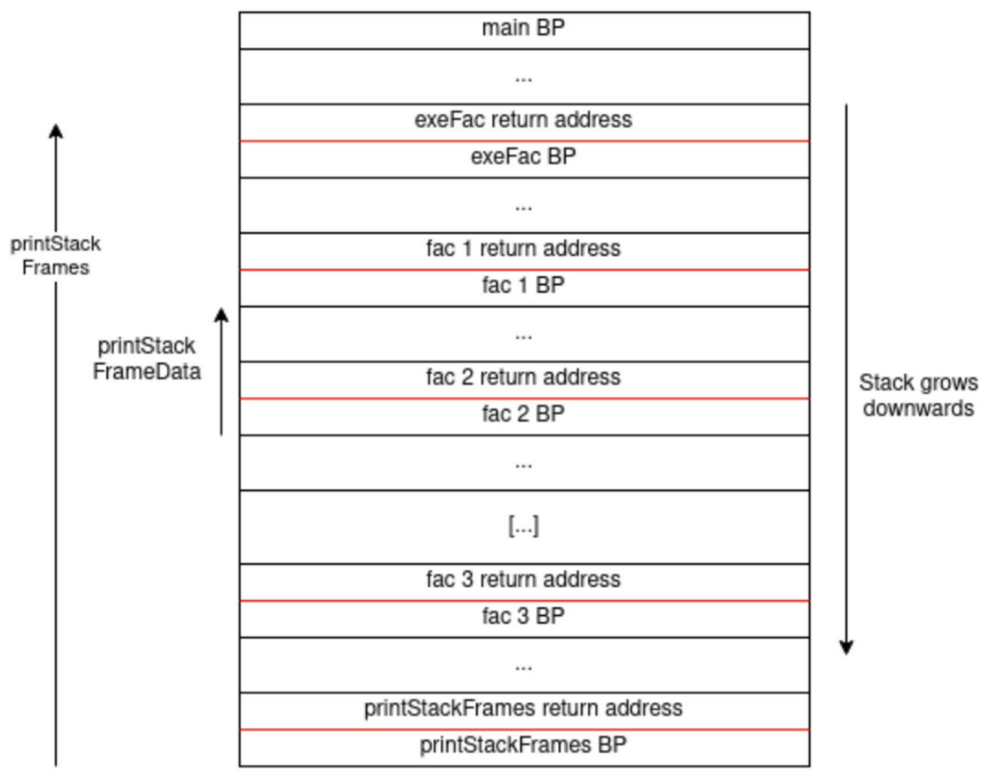

Figure 2: Stack frames during function calls

Figure 2 represents a holistic depiction of the stack during these function calls. Starting from the bottom, the program is traversing each memory address by adding the BYTES_PER_LINE constant and printing its details. The printStackFrameData label is an example of how the function works, printing the base pointer of the lower stack frame before printing the separator (denoted in red) and the other addresses of the stack frame above, stopping before its base pointer. printStackFrames repeats this process across the entire call stack.

Analysis and Testing

The approach to testing was multi-pronged – using unit tests as well as references to objdump, a command-line tool that prints information on object files. The flag "-d" was enabled to display assembler contents of executable sections specifically, as directed by the coursework specifications.

The unit tests are provided below in tabular format:

Name	Purpose	Output
testGetBasePointer	Tests whether the assembly implementation used in the function provides the same result as the C function "__builtin_frame_address()"	Passed
testGetReturnAddress	Tests whether the assembly implementation used in the function provides the same result as the C function "__builtin_return_address()"	Passed
testRelationship	Tests the relationship between the base pointer and return address of callee	Passed

Although "testRelationship" does not test a particular function, it reinforces the concept that the return address of the caller is one 8-byte line above the base pointer on the stack. Further unit tests were not written as I found it significantly challenging to test the printing functions without errors, due to the low-level nature of assembly. I wrote a macro for the assert function that I wished to use for testing. It tests whether an expression is True and prints the results accordingly, using an if-else statement.

```
executeFactorial: basePointer = 7ffc5467d840
executeFactorial: returnAddress = 401154
executeFactorial: about to call factorial which should print the stack

00007ffc5467d740:   00007ffc5467d760   --   60 d7 67 54 fc 7f 00 00
-------------
00007ffc5467d748:   000000000040117c   --   7c 11 40 00 00 00 00 00
00007ffc5467d750:   00000000000002d0   --   d0 02 00 00 00 00 00 00
00007ffc5467d758:   0000000000000001   --   01 00 00 00 00 00 00 00
00007ffc5467d760:   00007ffc5467d780   --   80 d7 67 54 fc 7f 00 00
-------------
00007ffc5467d768:   000000000040119e   --   9e 11 40 00 00 00 00 00
00007ffc5467d770:   0000000000000168   --   68 01 00 00 00 00 00 00
00007ffc5467d778:   0000000000000002   --   02 00 00 00 00 00 00 00
00007ffc5467d780:   00007ffc5467d7a0   --   a0 d7 67 54 fc 7f 00 00
-------------
00007ffc5467d788:   000000000040119e   --   9e 11 40 00 00 00 00 00
00007ffc5467d790:   0000000000000078   --   78 00 00 00 00 00 00 00
00007ffc5467d798:   0000000000000003   --   03 00 00 00 00 00 00 00
00007ffc5467d7a0:   00007ffc5467d7c0   --   c0 d7 67 54 fc 7f 00 00
-------------
00007ffc5467d7a8:   000000000040119e   --   9e 11 40 00 00 00 00 00
00007ffc5467d7b0:   000000000000001e   --   1e 00 00 00 00 00 00 00
00007ffc5467d7b8:   0000000000000004   --   04 00 00 00 00 00 00 00
00007ffc5467d7c0:   00007ffc5467d7e0   --   e0 d7 67 54 fc 7f 00 00
-------------
00007ffc5467d7c8:   000000000040119e   --   9e 11 40 00 00 00 00 00
00007ffc5467d7d0:   0000000000000006   --   06 00 00 00 00 00 00 00
00007ffc5467d7d8:   0000000000000005   --   05 00 00 00 00 00 00 00
00007ffc5467d7e0:   00007ffc5467d800   --   00 d8 67 54 fc 7f 00 00
-------------
00007ffc5467d7e8:   000000000040119e   --   9e 11 40 00 00 00 00 00
00007ffc5467d7f0:   0000000000000001   --   01 00 00 00 00 00 00 00
00007ffc5467d7f8:   0000000000000006   --   06 00 00 00 00 00 00 00
00007ffc5467d800:   00007ffc5467d840   --   40 d8 67 54 fc 7f 00 00
-------------
00007ffc5467d808:   0000000000401225   --   25 12 40 00 00 00 00 00
00007ffc5467d810:   0000000000000040   --   40 00 00 00 00 00 00 00
00007ffc5467d818:   0000000000000001   --   01 00 00 00 00 00 00 00
00007ffc5467d820:   0000000000000006   --   06 00 00 00 00 00 00 00
00007ffc5467d828:   0000000000000000   --   00 00 00 00 00 00 00 00
00007ffc5467d830:   0000000000401154   --   54 11 40 00 00 00 00 00
00007ffc5467d838:   00007ffc5467d840   --   40 d8 67 54 fc 7f 00 00
00007ffc5467d840:   00007ffc5467d850   --   50 d8 67 54 fc 7f 00 00
-------------
00007ffc5467d848:   0000000000401154   --   54 11 40 00 00 00 00 00
executeFactorial: factorial(6) = 720
```

Figure 3: Output of TryStackFrames

As shown in the printed stack frames, the base pointers are pushed on to the stack at the beginning of the stack frame (shown as the bottom, for all except main) and the return addresses are pushed on to the stack at the end of the stack frame (shown as the top). When comparing the output to objdump, one can observe how Figure 3 demonstrates this by deconstructing the return addresses in each stack frame. The objdump output that the bottom-most line corresponds to is:

0114f: e8 4c 00 00 00 callq 4011a0 <executeFactorial>

401154: b8 00 00 00 00 mov $0x0,%eax

These lines are within the main function. Thus, one would expect the return address of executeFactorial to be the instruction after the executeFactorial function call was made in the main function, stored in the rip register. Analysing the return address of executeFactorial printed at the top of Figure 3, one can notice that they are the same as expected, meaning that the getReturnAddress function works as intended. Similarly, the second-last stack frame printed corresponds to the executeFactorial function. Additionally, the getBasePointer function returns the same value as the base pointer of the executeFactorial stack frame, meaning that the function is implemented correctly. As demonstrated in Figure 1, the base pointers of a stack frame hold the address of the base pointer of the previous stack frame, and so on.

In the executeFactorial stack frame, the bottom-most memory address corresponds to the base pointer of that function. Moving upwards in the figure, the next line represents the getBasePointer function call. The line above represents the getReturnAddress function call, which points to the value 401154, the correct return address as established earlier. The subsequent lines correspond to the variables “result”, “number”, and “accumulator”, holding the values they were respectively assigned in the function (0l, 6l, and 1l). The next line represents the stack frame padding of 64 bits (or 8 bytes) as the stack frames are 16-byte aligned, and the line after corresponds to the return address of the next function (401225), which is pushed onto the stack after the factorial function call is made:

401220: e8 36 ff ff ff callq 40115b <factorial>

401225: 48 89 45 e8 mov %rax,-0x18(%rbp)

The following 6 stack frames represent factorial function calls, with the bottom line representing the base pointer, the second-last line representing the “n” variable, the third-last representing the “accumulator” value, and the top line representing the return address for the subsequent function. It is important to note that the return address for 5 of these factorial calls is the same, as it is recursive and returning to the same instruction. The last factorial call has a different return address at the top of its stack (“40117c”) due to reaching the base case, at which point the function printStackFrames is called, with its base pointer on the top-most stack frame.

Evaluation

My approach to the specifications produces a result that corresponds to that provided in the coursework requirements. I have commented the provided Assembly code with references to their functions and x86-64 conventions. I have also provided adequate unit tests to test the getBasePointer and getReturnAddress functions, as well as the necessary Makefile label to compile and link the files.

Conclusion

I found this practical to be very enjoyable as I like the low-level nature of the specifications, having to understand the underlying processes using registers and the stack behind computer systems we use daily. In particular, using the Assembly language was fascinating and allowed me to challenge myself.

Given more time, I would consider using inline assembly while printing the stack frame data to have a finer-tuned control over the memory locations, although I am still extremely satisfied with my results as using inline assembly may result in unexpected output.

Bibliography

[1] https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

[2] https://accu.org/journals/overload/31/173/bendersky/

[3] https://courses.cs.washington.edu/courses/cse351/17au/lectures/11/CSE351-L11-procedures-I_17au-ink.pdf

[4] draw.io

[5] https://web.stanford.edu/class/cs107/lab6/solutions.html

[6] https://stackoverflow.com/questions/42215105/understanding-rip-register-in-intel-assembly

[7] https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp17/cse506/ref/assembly.html

[8] https://cdrdv2.intel.com/v1/dl/getContent/671200