**Overview**

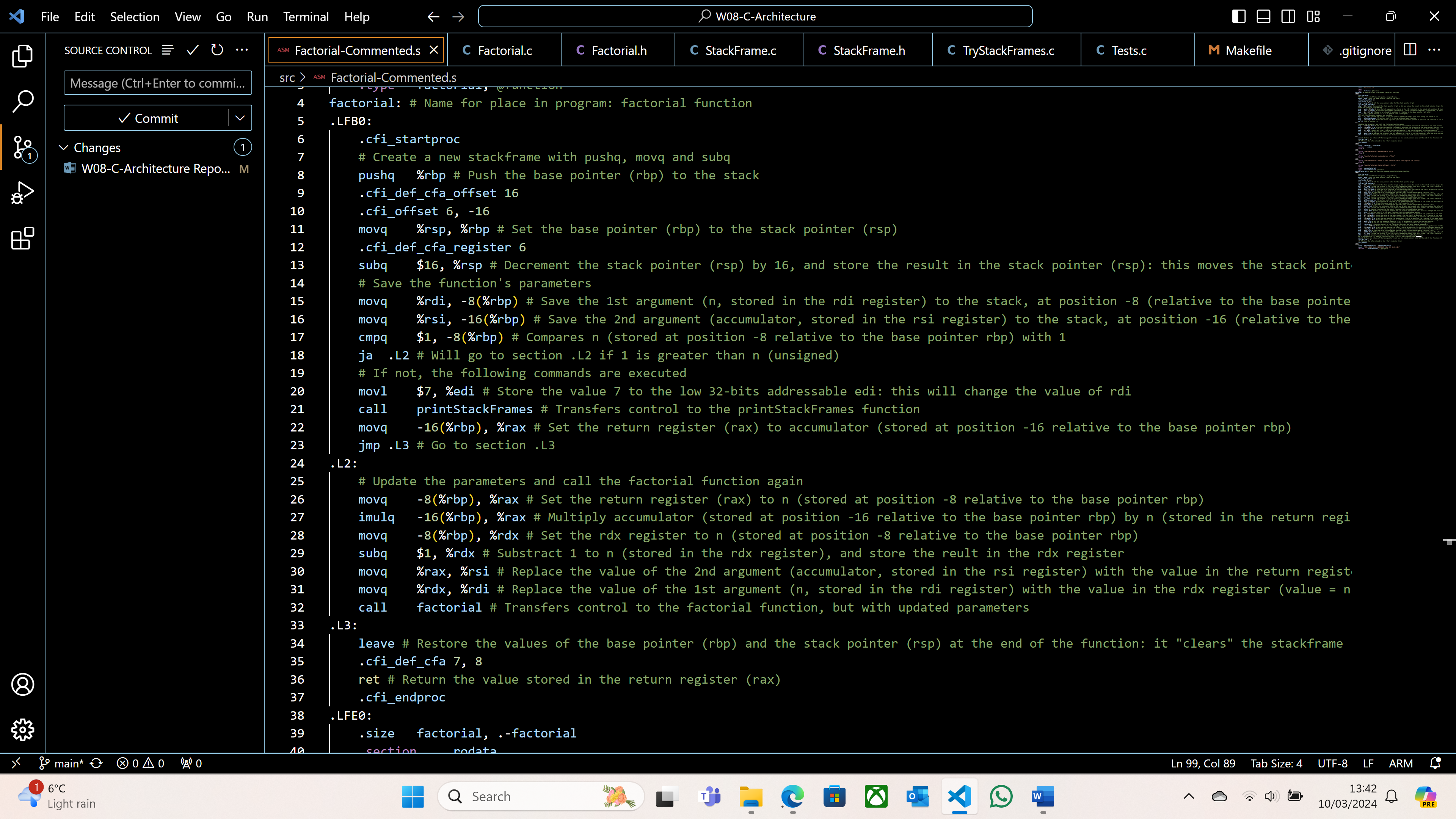
In this second CS2002 Practical, I was asked to comment some Assembler code, as well as design and implement a StackFrame module that provides the functionality to establish the base pointer and return address in the caller’s stack frame and print out stack frame data. I also had to implement test files 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](https://studres.cs.st-andrews.ac.uk/CS2002/Coursework/W08-C-Architecture/W08-C-Architecture.pdf), 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*:



*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](https://studres.cs.st-andrews.ac.uk/CS2002/Coursework/W08-C-Architecture/W08-C-Architecture.pdf), 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](https://studres.cs.st-andrews.ac.uk/CS2002/Coursework/W08-C-Architecture/W08-C-Architecture.pdf), and I managed to identify a few patterns:

1. Firstly, there are stack frames printed out, being the number used in the *factorial* function. So, for a default value of , 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.
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 , using the Makefile and executing the TryStackFrames executable.
* Running my code with different values of , 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 using the Makefile, and this is the obtained output:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

*Figure 2: ./TryStackFrames output*

We can immediately see that the format of the output is identical to the one in the [System Specification](https://studres.cs.st-andrews.ac.uk/CS2002/Coursework/W08-C-Architecture/W08-C-Architecture.pdf). 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 (, initialised at 1) and respectively, as well as what the different stack frames represent. Please see *Figure 3* for a visual analysis of this output.

A screenshot of a computer

Description automatically generated

*Figure 3: Analysis of the ./TryStackFrames output*

I also implemented some extra testing in a separate test file called Tests.c: it contains identical copies of the *factorial* and *executeFactorial* functions, except that they take an extra argument, *default­\_value*, as parameter. This is done to test the stack frames for different values of *n*, not just . I decided to implement three extra tests: one for a larger value of *n* (here ), one for a smaller value of *n* (here ), and one for . After running it with the Makefile, we obtain the following output:

A screenshot of a computer

Description automatically generated

*Figure 4: Extract of ./Tests output*

We clearly see that this output is similar to the previous output, except that there is an extra line in each stack frame: this corresponds to the extra variable that we introduced here, *default\_value*. Furthermore, the 5 patterns still hold here, as well as the previously made observations in *Figure 3*.

**Conclusion**

All in all, I have found this practical quite challenging, as this was the first time (excluding the W02-Exercise) coding in C. I was not used to pointers and memory allocations, and I struggled with formatting my truth table. However, after implementing unit tests and reviewing the examples give in lectures, I managed to implement the required truth table generator.

If I had extra time, I would try to implement a more memory-efficient stack implementation, as mine always has a maximum capacity of 1000, regardless of the size of the inputted formula. Also, for 26 Boolean variables and the formula “*abcdefghijklmnopqrstuvwxyz&&&&&&&&&&&&&&&&&&&&&&&&&*”, my implementation takes approximately 12 minutes to run, which I believe isn’t the most efficient implementation in terms of speed.