**Overview**

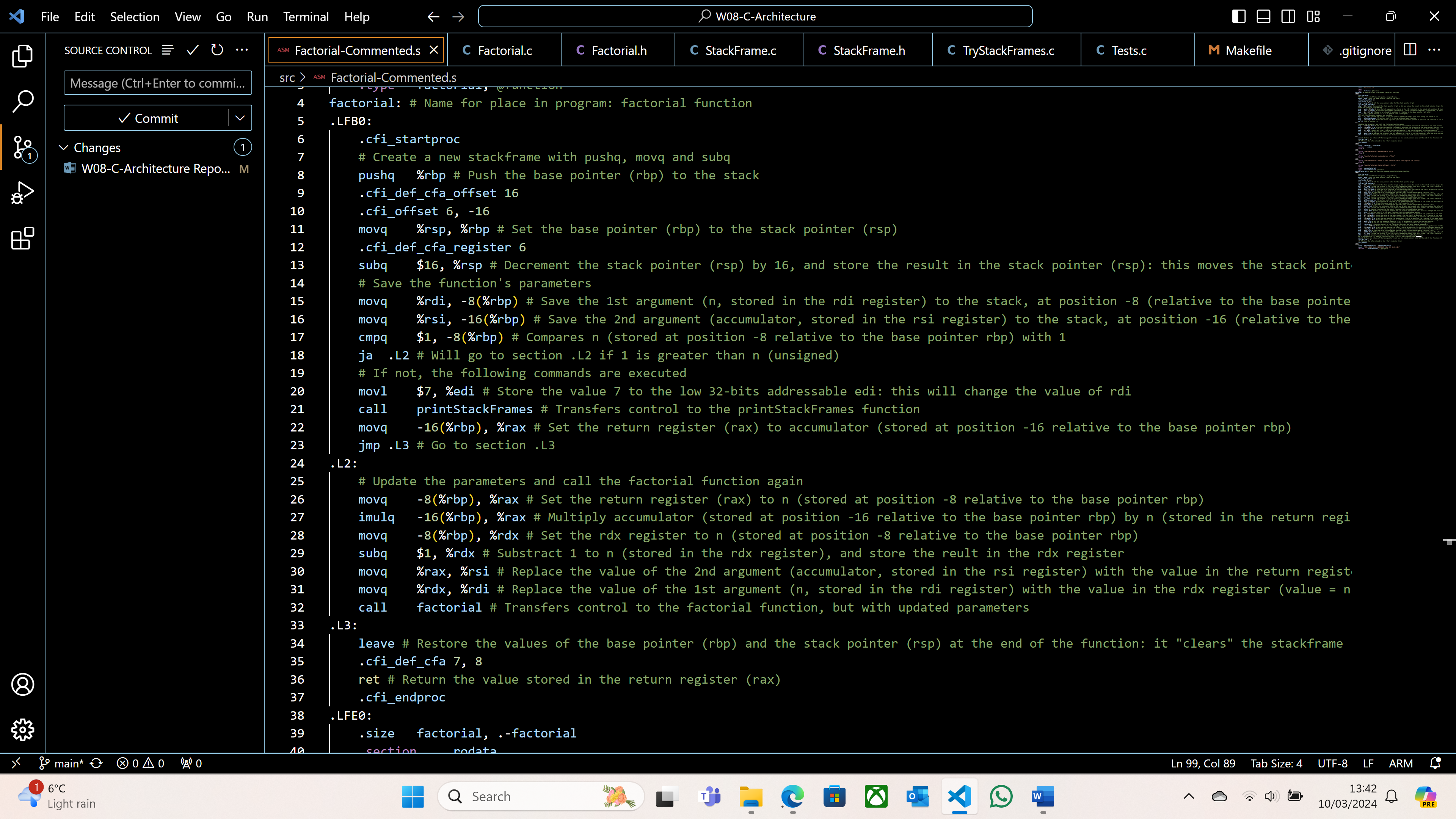
In this second CS2002 Practical, I was asked to comment some Assembler code, as well as design and implement a StackFrame module that provides the functionality to establish the base pointer and return address in the caller’s stack frame and print out stack frame data. I also had to implement test files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

* *test\_invalid.c*, which tests that my implementation deals with handling errors.
* *test\_stack.c*, which tests my implementation of a stack.
* *test\_print.c*, which tests the functions in *print\_table.c*.

The format of my testing is identical in each of the three testing files. Each test calls the method that it must test and compares the obtained output to an expected value. If they differ, the test fails, the function is considered invalid, both submitted and expected outputs are printed, and the test method returns 1. Otherwise, the test is considered successful, and the test method returns 0.

The detail of the test is printed out when ran: it explains which file is being tested, which method in the file, and in which scenario/for which inputs. To run the tests, simply check the README file for compilation and execution instructions. If my implementation is valid, all 8 stacscheck tests and my unit tests should pass.

**Part 2 Logic Questions**

Question 1:

To prove De Morgan’s law, I converted the formula in RPN and ran my *ttable* executable file with it. If the law is valid, then the inputted formula should be a tautology. I converted De Morgan’s law to “ab|cd||-a-b-&c-d-&&=” and executed *ttable* with 4 variables and this formula. The truth table obtained is the following:

A screen shot of a computer

Description automatically generated

*Figure 4: Truth table for De Morgan’s law*

All result entries are 1, therefore the inputted formula is a tautology, and De Morgan’s law holds.

Question 2:

The question can be interpreted in two different ways. On one hand, let’s suppose that there must be one and only one winner. If so, then we have two possible scenarios:

1. If the coin lands on heads, Chris wins
2. If the coin lands on tails, Ian loses, therefore Chris wins

In this case, there is only one outcome to the game: Chris wins.

On the other hand, let’s suppose that there can be multiple winners (or even no winners). If so, we have 4 Boolean variables:

1. a = “The coin lands on head”
2. b = “The coin lands on tails”
3. c = “Chris wins”
4. d = “Ian wins”

We can also translate the statements in the question into 3 compound formulas:

1. “The coin is either heads or tails (but not both)” = “”
2. “If the coin lands on heads, Chris wins” = “”
3. “If the coin lands on tails, Ian loses” = “”

By combining all three compound formulas into one, and converting them to RPN, we obtain the following: “ab#ac>&bd->&”. If we execute the *./ttable 4 “ab#ac>&bd->&”* command, we obtain the following truth table:

A screen shot of a computer screen

Description automatically generated

*Figure 5: Truth table for question 2*

This formula returns 1 when:

1. a = 0, b = 1, c = 0, d = 0: The coin lands on tails, so Ian loses, and Chris loses.
2. a = 0, b = 1, c = 1, d = 0: The coin lands on tails, so Ian loses, and Chris wins.
3. a = 1, b = 0, c = 1, d = 0: The coin lands on heads, so Chris wins, and Ian loses.
4. a = 1, b = 0, c = 1, d = 1: The coin lands on tails, so Chris wins, and Ian wins.

Question 3:

We have 5 Boolean variables:

1. a = “Ann attended the dinner”
2. b = “Barbara attended the dinner”
3. c = “Charles attended the dinner”
4. d = “Deborah attended the dinner”
5. e = “Elanor attended the dinner”

We can also translate the facts in the question into 5 compound formulas:

1. “Either Deborah or Charles or both attended the dinner” = “”
2. “Either Barbara or Eleanor but not both attended” = “”
3. “If Ann attended, then so did Barbara” = “”
4. “Eleanor attended if and only if Deborah attended” = “”
5. “If Charles attended, then both Ann and Deborah attended” = “”

By combining all five compound formulas into one, and converting them to RPN, we obtain the following: “dc|be#&ab>&ed=&cad&>&”. If we execute the *./ttable 5 “*dc|be#&ab>&ed=&cad&>&*”* command, we obtain the following truth table:

A screenshot of a computer screen

Description automatically generated

*Figure 6: Truth table for question 3*

This formula returns 1 only when a = 0, b = 0, c = 0, d = 1 and e = 1: only Deborah and Elanor attended the dinner.

Question 4:

We have 9 Boolean variables:

* a = “Box 1 contains the red card”
* b = “Box 1 contains the black card”
* c = “Box 1 contains the prize”
* d = “Box 2 contains the red card”
* e = “Box 2 contains the black card”
* f = “Box 2 contains the prize”
* g = “Box 3 contains the red card”
* h = “Box 3 contains the black card”
* i = “Box 3 contains the prize”

We can also translate the statements in the question into 8 compound formulas:

* “One box contains either a red card, a black card, or a prize but no cards” = “” = “” = “”
* “One box contains a red card, one a black card, and the other contains a prize but no cards” = “” = “” = “”
* “Box 1: This box contains a prize” is True = “”
* “Box 1: This box contains a prize” is False = “”
* “Box 2: The sentence on Box 1 is true” is True = “”
* “Box 2: The sentence on Box 1 is true” is False = “”
* “Box 3: Box 2 contains a black card” is True = “”
* “Box 3: Box 2 contains a black card” is False = “”

By combining all eight compound formulas into one, and converting them to RPN, we obtain the following: “ab#c#ab&c&#de#f#de&f&#&gh#i#gh&i&#&ad#g#ad&g&#be#h#be&h&#&cf#i#cf&i&#&&ac>&bc->&dc>&ec->&ge>&he->&”. If we execute the *./ttable 9 “ab#c#ab&c&#de#f#de&f&#&gh#i#gh&i&#&ad#g#ad&g&#be#h#be&h&#&cf#i#cf&i&#&&ac>&bc->&dc>&ec->&ge>&he->&”* command, we obtain a truth table, and here’s the header and an extract:





*Figures 7 and 8: Header and extract of the truth table for question 4*

This formula returns 1 only for c = 1, d = 1, h = 1, and the rest of the variables are 0: Box 1 has the prize, Box 2 has the red card, and Box 3 has the black card.

**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.