ARM11 Final Report - Group 13

Oliver FEDERICO, Chloe LAU, Karan OBHRAI, Leo UNOKI

June 19, 2020

Note: This is a trimmed version of our CO120.3C Final Report, mainly focusing on the extension CLogic.

1 Part IV Extension

1.1 CLogic

As part of our *CO112 Introduction to Computer Systems* course, we have built strong logic and circuit knowledge with a variety of software. In order to enhance the learning experience, we decided to build an extension to our C Project by building a logic gate combinatorial circuit design emulator, `CLogic`, with easy instructions for students to use within and beyond the course.

1.2 Usage

The program has two input forms, which take the format `<NAME> INPUT` or `<NAME> OUTPUT? <FUNCTION> <INPUTS...>` where `?` wildcard indicates optional. Requirements for each input file is to have at least one `output`. Detailed description of the requirement and usage can be referred to the README.md file in `CLogic/README.md`.

1.3 Methodology

The `CLogic` program takes in instructions to build a logic gate circuit through a parser, then the function finds the value of each output by a recursive trees approach. The program allows easier control of multiple `(n >= 2)` inputs being processed to one operation, with an efficient handling method and possibly faster for theoretical circuits designs.

1.4 Implementation

We have separated the entire function into its structure, the main function that parses the input and outputs, and an executor that executes the recursive functions for the outputs.

`CLogic.c` The file contains the main runnable file for the extension CLogic, it checks the input and processes the function with decode and execution functions.

`extension_lib` The folder contains the helper functions for `CLogic.c`, which `ext_decode.c` decodes and parses the input file and `ext_execute.c` executes the functions.

`extension_test` tests the functions with sample inputs and outputs. `ext_test_execute.c` tests the execute functions, `testsuite_runner.sh` This folder contains the tests for the entire function including the execute function.

1.4.1 Implementation of `ext_structs.h`

The file contains macros, enumerations and type definitions used for the Extension. It could be further expanded for more gate implementations. We decided that, in order to fully optimise our solution so that we could simulate the circuit as we change the inputs as fast as possible, we would build a recursive structure.

Therefore, we identified that we would need 3 main types of gate, `INPUT`, `OUTPUT` and `GATE`. `OUTPUT` and `GATE` are very similar except that the value of a gate with the optional `OUTPUT` tag will be displayed on compilation. All these types are wrapped into the typedef structure (`logic_node`), however the type specified

in this structure tells the execute function which field to look at within the structure. This allowed us to build the trie model as each `logic_node` structure is composed of pointers to other `logic_node` structures.

1.4.2 Implementation of `ext_decode.h` and `ext_decode.c`

The file contains helper functions to parse the input file by splitting the entire file into lines then tokenizing it by splitting it by spaces and end-line characters. This will then build the correct structure and return an array of all the nodes which the user has specified as outputs.

These can then be iterated through as each node contains pointers to subnodes and the cumulative action to be performed once each of their values are determined.

1.4.3 Implementation of `ext_execute.h` and `ext_execute.c`

The execution of the program has a main call from the main `CLogic.c` file that calls for a `bool execute(logic_node *node)` function, which allocates `input` to directly process user input values, `gates` and `outputs` recursively through helper functions and a print function to allow a trie based approach and which all functions return a boolean which is the value of the logic gate at any point.

Each enumeration type of operation has its own helper function that allows easy debugging and it is being passed through a print function to check whether it is an output that requires a result to be printed.

The implementation of this file was test driven, which allowed a rather swift method to ensure the functions are working as they are supposed to. Initially planning to use `CUnit`, we moved to assert and prints as we did not have enough time to unit test specifications to ensure the usage is absolutely right.

1.4.4 Implementation of `CLogic.c`

The file contains the main runnable file for the extension CLogic. It checks the input and processes the function with decode and execution functions.

It checks arguments and passes the file to `ext_decode.c` for parsing, then passes to the execute function to execute the tokenized instructions.

1.5 Group Organisation

The extension was worked on mainly by Karan and Chloe, while Leo and Oliver spent most of their efforts completing the Assembler. Karan focused on the structure and the parsing and Chloe on the executor.

1.6 Challenges within Implementation

The execute helper functions have been optimised to our best extent. Our original approach involved having `&&` for `and`, and `||` for `or`, and the loop would iterate through all the terms and produce a similar product of `(...((a[0]) && a[1]) && ...)`, which would be a very simple loop but would be way less efficient than our current approach which breaks if any value are 0 for `and`. We have implemented similar approaches for our other operators.

Parsing data from the files turned out be to quite challenging, especially due to the fact that strings are primarily stored as pointers meaning that we needed to deal with pointers extensively in order to accurately read the string. The `<string.h>` library was extremely useful, as it contained functions such as `strtok()` which allowed us to parse a string and separate it into tokens based on delimiters. Through this, we then were able to recursively read each line and form trie structures from them.

Another challenge we faced was how to successfully get all possible combinations of gates for an unknown number of inputs. A solution was found by imagining the inputs as binary sequences (e.g., for 4 inputs, all the combinations would be found by iterating from binary numbers 0000 to 1111 where each bit would symbolise the value of a gate). This was actually done using the LSL operator as the number of iterations was $2^{\text{number of inputs}}$. Then, by applying a mask to extract each bit, we would alter the input values.

1.7 Considering the Effectiveness of the Implementation

The extension employed methods and helper functions that allowed a more efficient solution than conventional logical operands, but for special operators like `XAND` and `XOR`, these required a solution that is minimum `O(n)` and has to check all elements. Also, with respect to the user interface side, we could have implemented a simple GUI had we had more time with the extension, which would make real life usage easier than a CLI or a .txt file based input format.

1.8 Testing

Testing for the extension CLogic is based on test suites that tests the functionalities of the execution and decoding, alongside a test pack for the entire function. We cross tested each other's code to ensure it worked. Have we have had more time to finish the project, the test suite should be populated with more edge cases to ensure the recursive function calls are right and no errors are shown.

2 Overall Group Organisation

We decided on Week 7 (Project started at Week 4) that two of the teammates would move onto the extension, and two would finish up the assembler. Karan and Chloe took up the extension implementation, and Leo and Oliver finished the rest. Midway through Week 8, we finished both sections and we all moved onto the LaTeX report, documentation and README files, as well as cleaning up and refining the code. We then proceeded onto the presentation.

The decision on the topic of the extension was a difficult one involving trade-offs on the length of the project, achievability, and the use of the program. We had come up with numerous ideas that we wanted to pursue but, due to time constraints, we were unable to give further consideration to any of the more ambitious ones. In the end, we believe that we made the right choice as our extension was finished on time with time left for the development of a test suite.

3 Reflections

3.1 Group Reflections

The lack of face-to-face and in-person communications made it more difficult to explain certain ideas, discuss implementations, and assist with bug fixing. Our communication as a team has been good despite members being in different time zones. We have utilised software tools such as *Notion* and *Google Drive* for progress checking, resources and ideas, alongside having *Messenger*, *Microsoft Teams* and *Zoom* for daily contacts and meetings. We also utilised *Overleaf*'s cloud based collaboration to enable us to work on the LaTeX files together without significant merge conflicts and to visualise the layout while we may be working on different branches on the projects. Despite this being an entirely remote project, we have remained in contact virtually and have kept up-to-date with the project at all times by making use of modern technologies.

A good recommendation that we have all agreed on is to set more internal deadlines during future projects, and make sure each milestone is reached within the time frame. This will allow the project to progress better and leave more time for contingency in case, for example, the project stops working suddenly after a merge.

3.2 Individual Reflections

Note: This section is the group members' personal opinions on the entire ARM11 Assembler and Emulator Project, alongside the extension project CLogic.

3.2.1 Oliver FEDERICO

At the beginning of the project I had a relatively strong understanding of the emulator spec and a good idea of the task ahead and how our team would approach it. As a result, I helped to consult and design the structures we would need. However, due to the way we allocated work for the emulator, my code was

dependent on the rest of the team completing their implementation before I could finalise mine. In addition, as someone with no prior knowledge of C, I struggled initially to balance learning C and working on the project. As a result, I felt that I was not contributing as much code compared to the rest of the team. On reflection, I should have consulted the group and agreed an area for me to focus on that allowed me to contribute earlier. I did not have a very strong understanding of Git and version control other than what was required for previous programming assignments and tests. I feel I am more confident now and have a better idea of how projects should be structured and how to use Git. My approach to implementing the assembler was not, with hindsight, as efficient as it could have been. I should have implemented one instruction type fully at a time, testing each instruction once it was implemented and then refactoring the code to reduce repeated code. However, I think the structure of the assembler was well thought out and makes clear what functions and files are used for.

3.2.2 Chloe LAU

Personally I have had trouble to understand the specification initially, causing a huge doubt and delay in starting coding functions, however, it then worked well and allocation of work seemed pretty good. The decision for two to finish up the assembler and two to start the extension was a good idea as otherwise we would have struggled to finish within the time frame for the entire project. However, we would still work on a lot of methods of software engineering that would boost our abilities to finish the project in a more efficient and tidy manner, and the ability to use a bit of TDD did help the overall project in terms of development efficiency. The project is overall very stimulating and challenging, and as one of our first group project in software engineering that is fully remote, I think we did well.

3.2.3 Karan OBHRAI

I found the C Project difficult overall. The necessity to implement certain parts of code earlier, rather than later, whilst learning it meant a lot of time was needed to research ahead of lectures in order to achieve an effective structure. As the specification was not always sufficiently specific, I found the Test-Driven-Design method extremely effective as both a debugging method and as a way to implement the code. By physically writing down a binary instruction, for example, we were able to follow a specific decision tree as specified by our interpretation of the specification, compare this to the expected output, and therefore readjust our decision tree accordingly. I found the extension a lot more engaging as the lack of rigidity in the specification meant we were able to increase the scope of our project as we liked and actually create a very useful and interesting program.

3.2.4 Leo UNOKI

Overall, I think we did well as a team, finishing the code and passing all the mandatory test cases. Although I initially found the specifications hard to understand, I managed to wrap my head around it after a number of discussions with my teammates. Throughout the entire project my strengths have been in writing test cases to ensure the correctness of the behaviour of the program. For the emulator, I worked on writing the test cases as well as helper functions that could be useful to debug the program later on when we encounter issues. For the assembler, I worked on the first half of the implementation which involves reading instructions from the file to tokenizing them and generating a symbol table. I think Oliver and I did a great job in designating tasks and working on them separately. This was a good approach as we could focus on specific task, writing good code and unit tests for our functions. As I worked on both emulator and assembler, one issue I noticed is that we had not planned out the design thoroughly at the beginning, focusing too much on the emulator. If I had to redo the project, I would redesign the data structure in a way that could be reused in both emulator and assembler. My weakness in this project was the lack of knowledge of Git and LaTeX. The project has given me the opportunity to learn both of them and appreciate how they streamline team collaboration.