Manchester Baby

Project information

Word count of main body: 792

Group number: 1

Submission date: 10.12.2021

Compile command under Unix: make exec

Name	Matriculation number
Jack Wiggall	2413924
Elliot Scott	2413916
Ross Coombs	2410466
Heather Currie	2411616
Kai Uerlichs	2421101

Development summary

We started off the development sprint by setting up the GitHub repository and having an initial collaborative coding session using Visual Studio Code Live Share, which allowed us to create the skeleton code for our program together and watch changes as they were entered.

This helped us to understand the basic structure of our Simulator. We then delegated tasks based on the empty method implementations and assigned them on the Issues and Projects pages of GitHub. Going forward, each group member created branches on the repository to separate the work by method and avoid merge conflicts and other Git idiosyncrasies. We followed the same process with the Assembler, where we built the basic code structure together, before delegating tasks and then bringing the completed code together at the next meeting.

After we implemented the core solutions for the Simulator and Assembler, we discussed possible extensions. We had already laid the foundations for some of them, such as the memory expansion, in the previous development phase, which allowed us to implement these features quickly.

We decided that for the Simulator, we wanted to increase the number of addresses in the store, rather than the register width. This is because the register width of the Manchester Baby is already 32-bit by default, which is on par with CPUs released in the last decade. There would be no practical application of a wider register with the minimal instruction set of the Baby. However, increasing the store length would allow for longer programs and more complex calculations, which is why we settled on increasing the addressable locations to up to 4096 lines. We further implemented a graphical display and an extended instruction set.

Updated instruction layout:

$$\underbrace{0000\ 0000\ 0000}_{\text{Operand}}\ 0\underbrace{000\ 0000}_{\text{Opcode}}\ 0000\ 0000\ 0000$$

For the Assembler, we regarded the error handling extension as a crucial part of the program, as it not only made the tool more useful, but also provided important information during development. We further wrote the Assembler in such a way that it was ready to work with the increased address space, and with instruction sets of up to 128 instructions.

Lastly, we developed a few Assembly language programs, which we provided in both Assembly Code and Machine Code form as part of our submission.

Challenges and Solutions

During the coding process, we wanted to implement different colours for text in the display functions. This was done using the "Colors.h" header file, which defined functions for outputting coloured text. However, this only worked with text in the quotation marks, and not for string objects. A workaround for this was to manually colour-code the text with escape codes corresponding to the colour choice.

Since the assembler could not be tested until the code for it was complete and the code was done by multiple people, there was bound to be a lot of errors. This was partially addressed by collaborative coding sessions, which helped to minimise the problems, but was not the final solution. We were able to find problems by adding proper error messages to the code. This allowed us to debug the code and see exactly where the issues were but was also one of the project outcomes outlined in the extensions.

Another hurdle was the binary number handling. While writing the library functions for the conversion and arithmetic of binary numbers was not a big challenge, we realised later that the library would best be implement following big-endian convention, as this closer simulates the Manchester SSEM's way of handling signed integers. This meant that we were required to refactor the whole library and adjust it to handling numbers "the other way around".

Lastly, during this development cycle we noted the importance of developing and implementing strong testing schemes for our code. Due to the large interdependency of code segments, oftentimes a full test of the data structures and algorithms was not possible without linking all the parts together. However, we realise that with the necessary planning, certain test cases and procedures could have been implemented to avoid a large number of errors, which would have sped up the development despite the additional work of writing test code.

Conclusion and Reflection

As a group we came together early and made plans to implement parts of the simulator then the assembler. We actively communicated through messages, but also on longer collaborative development sessions, which we used to debug our results and implement further extensions.

Being able to get a second opinion right away decreased the total time the project took, as code review could take place simultaneously. Since we communicated so often, we could help each other with small errors that would take up large quantities of time if help wasn't available. This project was very helpful for learning to code in C++ and we are happy with the finished product.