Oscar Saharoy, os408@cam.ac.uk, St John’s College, Team 2

IIA Project GF2: Software

Introduction

Over the last few weeks, we have been developing a logic simulation program in Python, learning about the five major phases of the software development life cycle: specification, design, implementation, testing, and maintenance. The process has taught us a lot, from working in a team to technical skills, and has been highly enjoyable along the way.

In this report I cover the project and how we approached it, our successes and mistakes made, and the invaluable lessons I’ve learned over the last few weeks.

The Logic Simulator

We imagined we were asked by a client to implement a logic simulator, receiving the specification and working to a tight deadline to deliver the product. We worked as if part of a software development team in a larger company, and our team was responsible for implementing four modules of the total eight. These included language design and syntax parsing to user interface design, so there were a wide variety of tasks to handle.

A picture containing chart

Description automatically generatedThe logic simulator allows a user to investigate the logic state over time for a given combinatorial or clocked logic circuit. The circuit is defined by a definition file that the user writes in a logic definition language that we designed. The logic network is then constructed by the program and the user can step it forward in time, seeing how the logic state at selected points in the circuit changes over time. We also added features that enable the user to add and remove monitoring points and connections between devices, and toggle the logic state of inputs to the circuit. As an exercise in internationalisation we also added a Chinese language translation.

The software is structured as nine python files, eight of which export a class. These eight are called devices.py, gui.py, monitors.py, names.py, network.py, parse.py, scanner.py and userint.py; each is named after the part of the logic simulation software that it handles. The ninth is logsim.py, which is the entry point for the program. Within logsim.py, an instance of each of the eight other classes is created. Below I have tried to illustrate the program structure and the primary interactions between the classes.

Timeline

Description automatically generated

Other than these files, we have a pytest test suite for each file other than userint.py, gui.py and logsim.py; the first two of these are impractical to test using pytest and the last is simple enough to not require testing. We have a set of example definition files in the ./logsim/tests directory, which are used in the test suite to verify the code is working correctly. There is also the ./logsim/images directory which holds icons used within the graphical user interface code.

Teamwork Approach

Before this project I hadn’t met Eric or Max, so it was a pleasure to get to know them and work together on this project. I think we worked together really well; everyone put in a lot of effort and we helped each other through problems we were facing and with design decisions. We didn’t really have a lot of experience with projects like this but I think we figured things out as we went along and had a good time learning some new skills.

We made sure to split the work evenly, and we each picked the part we thought we would enjoy most. We also didn’t plan too far ahead at a time, making sure we could quickly reprioritise if necessary. We handled the initial interim report well, with Eric and Max designing a very functional and simple logic definition language. This was a critical design decision that made the implementation of the parser much easier later on. Max also created some nice circuits including an adder and binary counter which were excellent for testing throughout the implementation and maintenance phases.

In the maintenance phase, we did feel the effects of the tight deadline. There was an oversight with the handling of comments which led to the program hanging - we would have liked to have spent more time trying to break the program and increasing test coverage. Leaving the submission of my second interim report so late even led to me forgetting to include example definition files; a lesson I am happy to have learned from this project is that being on schedule is being behind schedule, and to be able to deliver on schedule you really have to be ahead of schedule.

I think we did much better with time management leading up to the final report and implemented the newly requested features very well. Max had the NOT gate working within a few minutes, and the Chinese language internationalisation that Eric implemented was something I didn’t even know would be possible. We also improved the feel of the UI significantly and user experience benefitted greatly.

Remote Collaboration

While we would have liked to have been together while working on the project, it wasn’t really feasible and so we used a few online tools to support our development efforts. The first was Facebook Messenger, which allowed us to communicate instantly and discuss features or ask each other questions easily. It was also helpful to have a record of what we had decided to avoid forgetting.

For version control we set up a github repository and got a pytest/flake8 CI pipeline working from the start. This helped us keep testing and code style as a top priority. However, I set up the pipeline and I missed that flake8 does not match pep8 absolutely, one example being that docstrings and comments are limited to 79 characters under flake8 while pep8 recommends 72 characters in these cases. This is a good learning from the project and the type of thing I will make sure to look out for in future. We were very good at working together on the repo; I didn’t have to do a single merge so I think we were working exclusively in parallel.

To manage our tasks, we used Trello which is like a to-do list but with a few extra features that help coordinate work and stay focused. Tools like Trello are especially used within the agile methodology which is one technique we applied some ideas from. I think it was a really good way to keep track of what we still needed to do and see what we were all up to, but it was also very easy to forget to update it.

Our team reports and definition files were written in Google Docs, which gave us a simple way to edit the documents together and quickly iterate on them. We kept the example definition files and reports updated as we made changes which reduced the amount of work to do just before the deadlines.

I think overall our remote collaboration approach worked well but there were definitely times when we could have benefited from a time management tool as well. Maybe setting timelines for each task would have helped us keep on track and reduce the amount of time pressure towards the deadlines.

Names.py

In the names.py module, I implemented three methods. These were Names.lookup, Names.query, and Names.get\_name\_string. They are responsible for assigning a unique id to each name and fetching the ids and names during program execution. After I had written the methods, Max wrote tests for them and made some improvements which I think was a good way to make sure nothing was missed and the code had two people’s creativity and skills to benefit from.

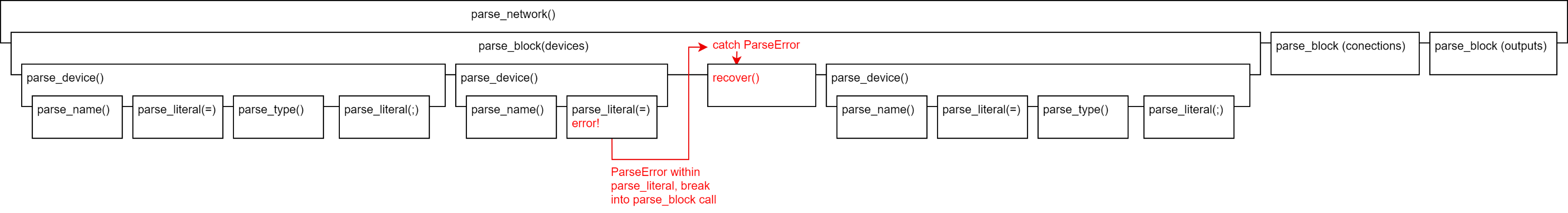
One idea I liked was the below use of a python generator expression and the “next” function to find the matching name string given an id in the name-id mapping dictionary. This is a pythonic and compact expression, but it’s still a bit suboptimal in that it executes in O(n) time whereas the reverse lookup can execute in O(1) time due to the hash map implemented by the python dictionary.

**return** next((name **for** name, name\_id **in** self.name\_map.items()

**if** name\_id == query\_id), None)

Parse.py

The next module I wrote code for was the parse.py module, which was responsible for checking the syntax of the provided definition file and building the logic circuit afterwards. The project handout was very helpful in getting started, and once I had the basic idea the simplicity of our circuit description language made implementation very straightforward. Eric helped me understand the syntax error recovery process, for which we used an exception that broke us out of the recursion and was caught at a higher level to recover and resume parsing from a known state. I wrote tests as I wrote the methods to make sure they were all working ok, although I missed including some vital tests like unterminated comments.



back to known state, proceed as normal

parse\_device()

Gui.py

I also wrote some code for the gui.py module; I started adjusting the layout to fit the elements better and added some dropdown menus to allow selection of the connection start and end points. I also did a few bug fixes on the GUI. However, I found it difficult to get the connection adding and removing logic working so Max helped me finish that off and I directed my time towards writing the report appendices instead. I think it was really positive that we were able to restructure our tasks to fit to the team’s strengths and weaknesses even as we were under a lot of time pressure.

Testing

We used pytest throughout the project as an easy way to verify that no breaking changes were made. I thought it was very helpful to be able to run the tests locally as I edited code to get instant feedback on whether I was doing the right thing. The testing habit we built up was very helpful, especially on the day before the deadline when we decided to make a change to the logic definition syntax. Max made the change, and the tests instructed exactly where to update the code and definition files so that he was able to painlessly implement what could have been a change with deep and unrealised consequences within the codebase.

Text

Description automatically generated

Possible Improvements

There are a number of improvements we would like to have made if we had more time. For example, a graphical representation of the circuit within the software would really help to bridge the gap between the definition file and the GUI controls. This could enable the addition of monitors to be as simple as clicking a point on the circuit diagram, and adding and removing connections could be done visually. I think this would require a lot more experience with the GLCanvas and wxPython, and it is an interesting problem to decide if a good circuit diagram can be constructed from a given definition file. Perhaps the graphical representation of the circuit could replace the definition file completely, but this would require additional work to enable construction of the circuit diagram within the software – this is another improvement that could be made.

The logic simulator could potentially be expanded to recognise groups of outputs as encoding a binary number – this would then enable circuits like adders and multipliers to be built and tested. Another improvement could be to allow previously constructed definition files to be inserted into new ones; this adds a layer of abstraction that could allow the user to work on more complex circuits like ALUs. This would definitely need a lot of extra work, including performance optimisation along with new parsing and UI design.

Finally, the experience of creating a logic definition file was a bit difficult at first, with no syntax highlighting or any of the helpful features that an IDE is able to offer. It might be a nice project to learn how syntax highlighting is done and how intellisense can autocomplete code as it’s being written, and apply them to the simple circuit description language that we created.

Conclusions

We successfully implemented the logic simulator and learned a lot along the way; I was happy to get experience in a lot of different areas. Getting to know Max and Eric and having their support during the project was really nice, and I think we all brought unique skills to the task which meshed together well.

Creating the logic description language was something I had no experience in, and helped me to understand a bit more about the programming languages we have used in the past and the design effort that goes into them. The parser was also an interesting problem, and I will be sure to research grammars beyond LL(1) in future and try to learn more about them. I wondered while building the parser if there was a more declarative way to handle the symbols as well, or a system that leverages the termination characters like semicolons to parse the file with many processes in parallel.

I had never worked on a software project to such tight deadlines before, so the teamwork methodologies we used were a great learning experience. Also, where we went wrong like in time management and missing details about code style and test coverage are invaluable experience to be carried forward to future projects.

And the last thing is to give a big thankyou to Andrew and Tim for hosting such a good project!

Appendix A: Example Definition Files

Example Definition File 1: Full Adder

Shape

Description automatically generated with medium confidence

This is an implementation of the full adder circuit which takes three inputs and provides two outputs representing the 2-bit sum. The circuit comprises 2 XOR gates, 2 AND gates and an OR gate, arranged as shown in the diagram above.

START

DEVICES {

  a = SWITCH(0);

  b = SWITCH(0);

  c = SWITCH(0);

  xor1 = XOR;

  xor2 = XOR;

  and1 = AND(2);

  and2 = AND(2);

  or1 = OR(2);

}

CONNECTIONS {

  a > xor1.I1;

  b > xor1.I2;

  xor1 > xor2.I1;

  c > xor2.I2;

  c > and1.I1;

  xor1 > and1.I2;

  a > and2.I1;

  b > and2.I2;

  and1 > or1.I1;

  and2 > or1.I2;

}

OUTPUTS {

  xor2;

  or1;

}

END

Example Definition File 2: 4-bit Counter

Shape

Description automatically generated with medium confidence

This circuit represents a simple 4 bit counter, as shown in the above diagram. The four D-Type devices are set up with their inverted output connected to their data input, so that they act as toggles. The non-inverting outputs are connected to the clock input of the next gate so that it behaves as a binary counter.

START

DEVICES {

  ff0 = DTYPE;

  ff1 = DTYPE;

  ff2 = DTYPE;

  ff3 = DTYPE;

  clk = CLOCK(1);

  clear = SWITCH(0);

  set = SWITCH(0);

}

CONNECTIONS {

  ff0.QBAR > ff0.DATA;

  ff1.QBAR > ff1.DATA;

  ff2.QBAR > ff2.DATA;

  ff3.QBAR > ff3.DATA;

  clk   > ff0.CLK;

  ff0.Q > ff1.CLK;

  ff1.Q > ff2.CLK;

  ff2.Q > ff3.CLK;

  clear > ff0.CLEAR;

  clear > ff1.CLEAR;

  clear > ff2.CLEAR;

  clear > ff3.CLEAR;

  set > ff0.SET;

  set > ff1.SET;

  set > ff2.SET;

  set > ff3.SET;

}

OUTPUTS {

  ff0.Q;

  ff1.Q;

  ff2.Q;

  ff3.Q;

}

END

Example Definition File 3: NAND and NOR Gates

Shape

Description automatically generated with medium confidence

This is a circuit we made to test the NAND and NOR gates; the top 4 NAND gates are set up to be equivalent to the NOR gate below. The XOR gate should then always output low to show the two branches are functioning identically.

START

DEVICES {

    nand1 = NAND(2);

    nand2 = NAND(2);

    nand3 = NAND(2);

    nand4 = NAND(2);

    nor1 = NOR(2);

    xor1 = XOR;

    a = SWITCH(1);

    b = SWITCH(1);

}

CONNECTIONS {

    a > nand1.I1;

    a > nand1.I2;

    b > nand2.I1;

    b > nand2.I2;

    nand1 > nand3.I1;

    nand2 > nand3.I2;

    nand3 > nand4.I1;

    nand3 > nand4.I2;

    nand4 > xor1.I1;

    a > nor1.I1;

    b > nor1.I2;

    nor1 > xor1.I2;

}

OUTPUTS {

    xor1;

}

END

Appendix B: Logic Description Language

Our logic description language has a simple form that allows the user to define devices in the network (logic gates, clocks, switches and D-types), connections between those devices, and the outputs to initially monitor.

The overall file is of the following structure:

START

DEVICES {

# define devices in the network here #

    nand1 = NAND(2);

}

CONNECTIONS {

# define connections between devices in the network here #

    a > nand1.I1;

}

OUTPUTS {

# define outputs to monitor here #

    xor1;

}

END

The START and END keywords are required at the start and end of the file, but there can be whitespace/newlines before and after them. Throughout the file, whitespace and newlines are ignored.

The DEVICES block must come after START, indicated by the DEVICES keyword and the opening and closing curly brackets. Within this block is a list of device definitions of the following form:

deviceName = DEVICETYPE(5);

The deviceName is the name of the device, used to refer to the device throughout the file. It can be any combination of alphanumeric characters but must not start with a number. The DEVICETYPE is one of CLOCK, SWITCH, AND, OR, NOR, NOT, NAND, XOR, DTYPE. The number in brackets describes something about the device: for the clock it is the half-period, for the switch it is the initial logic state, and for the rest of the gates it is the number of inputs between 1 and 16. For the DTYPE, XOR, and NOT devices the number is omitted and so are the brackets. The semicolon is required.

Next is the CONNECTIONS block, which is indicted by the CONNECTIONS keyword and the curly brackets. Within this block, define connections between two devices like this:

deviceName1.outputPin > deviceName2.inputPin;

deviceName1 refers to the device where the connection starts, and outputPin is the pin on that device that the connection comes from. The outputPin and the dot are omitted in most cases except for the DTYPE, where the outputPin is one of Q or QBAR. deviceName2 refers to the device that the connection leads to, and inputPin is the pin on that device. The inputPin is one of I1, I2, I3 etc for multi-input gates, or SET, CLEAR, CLK or DATA for the DTYPE.

Finally, the OUTPUTS block is created similarly to the other two and has a list of output signals to monitor inside, of the form:

deviceName.outputPin;

In the same way as the connection definition, the deviceName refers to a device and the outputPin to an output pin on it, which can be omitted except for the DTYPE.

Appendix C: User Guide

logsim.py is a logic analyser that can read input files specifying a combinatorial or clocked logic circuit, then simulate the circuit and provide a visualisation of the outputs’ logic levels over time.

Prerequisites

You will require a [python](https://www.python.org/downloads/) installation at least as new as version 3.6 on your system and the [wxPython package](https://www.wxpython.org/pages/downloads/). OpenGL is also required.

Installation

Install the program by downloading the project from [the github repo](https://github.com/OscarSaharoy/gf2) and extracting the contents to a location of your choice. You can also git clone the project.

Usage

Invoke the tool from the command line using: python3 logsim.py input/file/path.txt

You can choose to use the command line interface with: python3 logsim.py -c input/file/path.txt

Errors and warnings will be printed to the console and will help find bugs in your input circuit description file.

You can also run in Chinese using LANG=zh\_CN.utf8 python3 logsim.py input/file/path.txt

GUI

logsim.py has a GUI to make logic analysis easy. If the circuit definition file is parsed correctly, the GUI will open in a new window and present options to run and monitor the circuit as well as visualise the logic state over time.

On the right, the “run” button allows you to startup the circuit and step forward by a number of time steps, and the “continue” button lets you continue stepping forward from the current state. The number of time steps for each button is specified by the number in the text box above.

Beneath, the “Switches” section allows you to toggle the state of any switches in the circuit to examine their effect.

Below that, the “Monitors” section allows you to add and remove monitors at points in the circuit, for which the logic state over time is shown by the traces on the left of the window. Add one by selecting the signal name from the dropdown menu and pressing “Add”.

At the bottom, the “Connections” section allows you to add new connections between devices or remove existing ones. Do this by picking the connection start and end locations from the dropdown menus and pressing “Add” or “Remove”.

A picture containing chart

Description automatically generated

Reset Simulation

Time axis

Logic traces for selected outputs shown here

Set the states of input switches

Add or remove output monitors

Add or remove connections

Run the simulation forward in time

Appendix D: Files in ./final folder

Our ./final folder contains the completed code for the project. Inside are the python files for each of the modules of the program and test files for them. There is also the ./final/images directory which contains some icons for the GUI, and the ./final/tests directory which contains some definition files which are used in the tests for the parser and scanner modules. The ./final/locale folder contains the data for the Chinese language translation of the software.

To run logsim.py from the ./final folder, change the working directory to the final folder and then run logsim.py as normal with python3 logsim.py input/file/path.txt

(.env) PS C:\Uni\gf2> tree final /f

./final

│ devices.py

│ glut32.dll

│ gui.py

│ logsim.py

│ monitors.py

│ names.py

│ network.py

│ parse.py

│ scanner.py

│ test\_devices.py

│ test\_monitors.py

│ test\_names.py

│ test\_network.py

│ test\_parse.py

│ test\_scanner.py

│ translate\_cn.po

│ userint.py

│

├───images

│ floppy-disk.png

│ logout.png

│

├───locale

│ └───zh

│ └───LC\_MESSAGES

│ translate\_cn.mo

│ translate\_cn.po

│

├───tests

│ badconnections.txt

│ baddevices.txt

│ badinputs.txt

│ badoutputs.txt

│ badtypes.txt

│ basic.txt

│ duplicatestatements.txt

│ empty.txt

│ example1.txt

│ ir1\_adder\_ab=1.txt

│ ir2\_adder.txt

│ ir2\_counter.txt

│ ir2\_nandnor.txt

│ nandnornot.txt

│ noend.txt

│ nostart.txt