Name: Isaac Brown
crsID: iwb21
College: Pembroke
Github: Isaac-W-Brown

GF2 Final Report
Isaac Brown

Team 16

7 June 2021

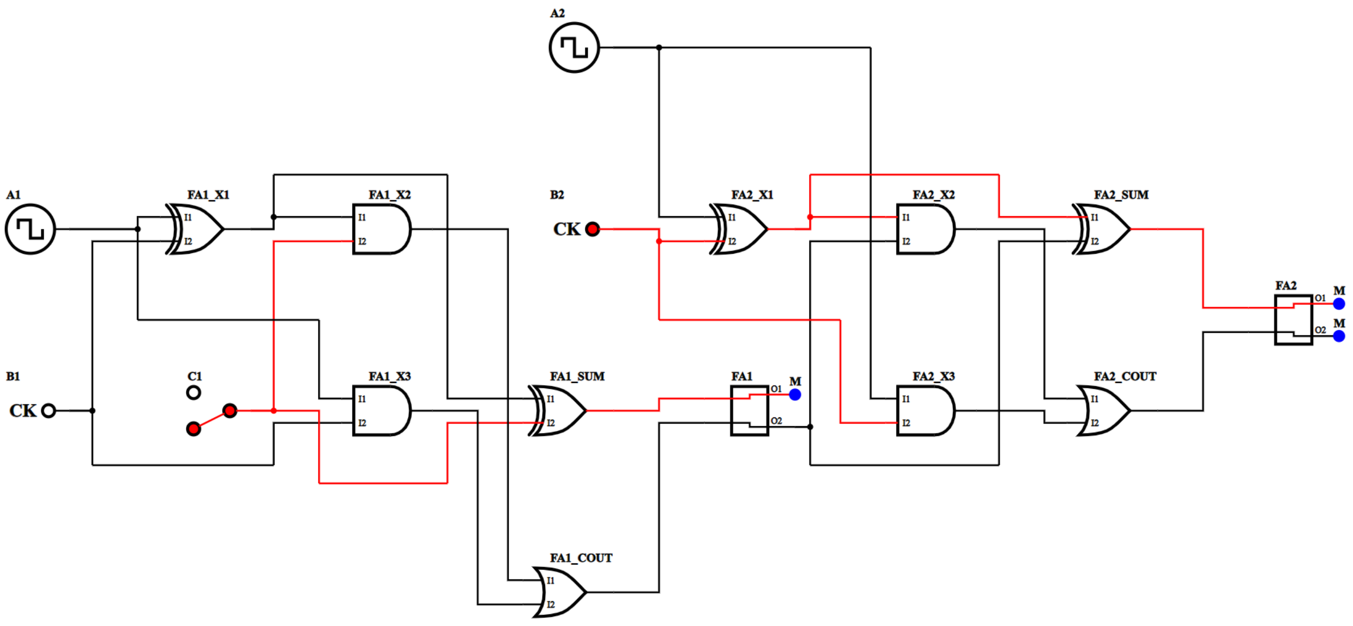


Figure 1: 2-bit adder circuit diagram generated by the Logic Simulator program

Abstract

The motive for this project was to develop a program for a client to simulate a logic circuit, in order to reduce time spent debugging logic hardware. We achieved this by creating an integrated system, from user definition file to a Graphical User Interface (GUI). As a team we collaborated through `git` version control and we maintained clear coding practices and efficient data structures throughout. The result of this project is a fully working logic simulator, alongside a definition grammar which is fast to write. Furthermore, our user interface has features above and beyond the specification, with a live-parsed text editor and an automatically generated logic circuit diagram.

1 Introduction

Logical systems are the backbone of modern digital technology, and ensuring their reliability requires rigorous testing. However, implementing multiple designs and debugging on physical boards is time-consuming and, relative to a software approach, costly. Therefore the purpose of this project is to develop a reliable and fast logic simulator, with an intuitive and attractive user interface.

Approximately one-third of the code was provided by the 'client'. This consisted of classes to handle devices and network execution, as well as skeleton files for the front-end. The logic of the devices was provided, however more devices and logical processes were added during this project. Aside from the supplied code, the most useful previous contributions to this project are the online resources for Python, wxPython and OpenGL, as well as logic circuits which were used to test the program.

The rest of this report will present the features of the end-product of this project, as well as the function of the back-end of the simulator. The processes used to create the software are described, including teamwork and remote-collaboration, made more significant in the context of COVID.

2 Logic simulator function

2.1 'Server'-side

The pipeline from definition file to GUI passes through 3 main stages: scanning, parsing and GUI configuration. Around these, there are helper classes which describe, build and store the logic circuit. These are the `Names`, `Device`, `Devices`, `Network`, `Monitors`, `UserInterface` and `Diagram` classes. All classes are called from the executable file `logsim.py`, which is passed the definition file path [Fig. 2]. From the user input to program output:

Scanning

The scanner recieves the file path and opens the file. It exchanges letters and names for numerical IDs with the `Names` class, which are then used to create symbols using the `Symbol` class. The scanner assigns each symbol a type, for example `self.KEYWORD`. These types are used by the parser in order for it to decide how to call the Devices or Network classes, or indeed return an error. Symbols also have position attributes for returning the positions of Syntax errors.

The scanner also handles white-space. White-space is largely ignored, although it does indicate the termination of a name. New lines have no function other than ending single line comments. Comments are passed by looping through upcoming symbols until # for multi-line comments (which begin #), or a new-line for single line comments (which begin $).

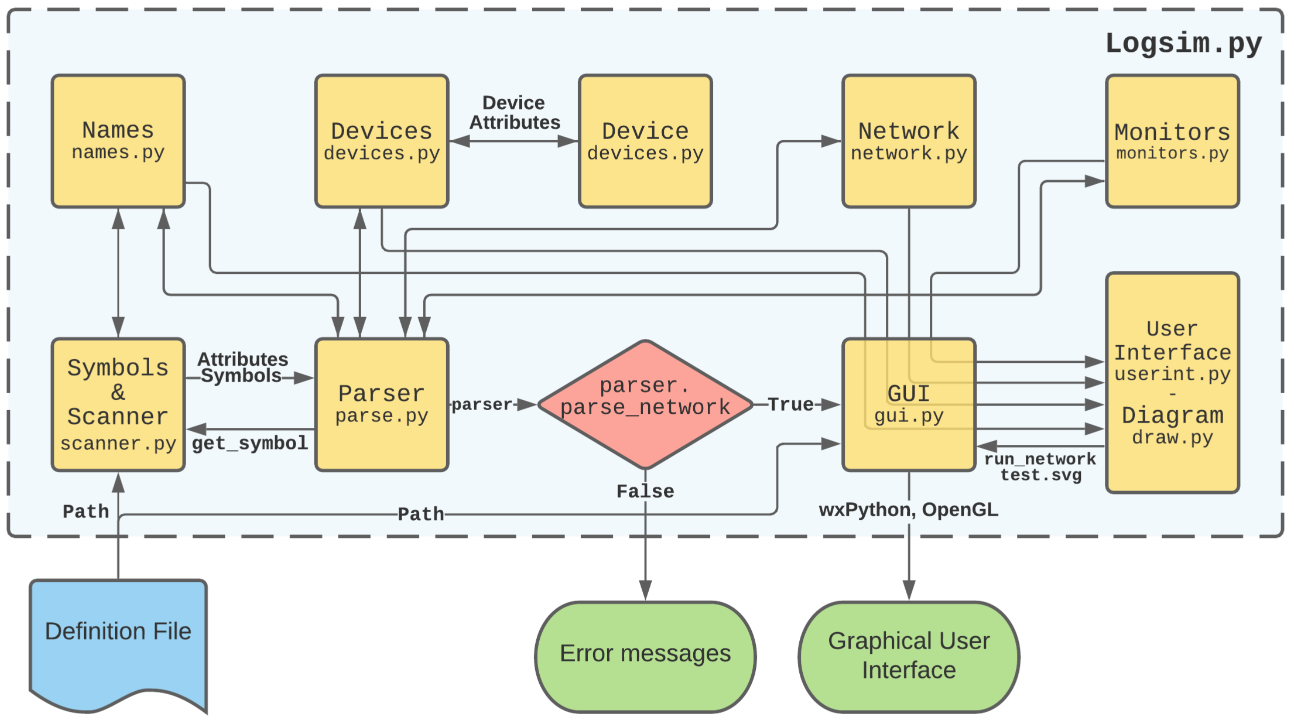


Figure 2: System overview block diagram. Files are shown in blue, classes in yellow, decisions in red and outputs in green. Arrow direction indicates the flow of information

Parsing

The `Parser` class has one public method `parse_network` which builds the circuit and returns `True` or `False` depending on whether there were errors building the circuit. Firstly, `Parser.parse_network` iterates through symbols from the scanner and tests them for syntactic correctness against the EBNF formulated in the first week of this project. The EBNF is unchanged since the first report, with the exception of a `numberlist` which was required for the `SIGGEN` in the maintenance task.

Next, once a line has been confirmed as being free of syntax errors, the arguments accumulated by the scanner are organised and passed to the `Devices`, `Network` and `Monitors` classes, which the `Parser` class is initialised with. These classes build the circuit, or return a 'Semantic' error. If the circuit has been built error free, `parse_network` returns `True`, the wxPython app is started and the `gui.py` module is called.

Error detection and handling works differently for syntax and semantic errors. The `Devices`, `Network` and `Monitors` classes themselves provide some of the semantic error detection, while syntax errors are handled exclusively within the `Parser` class. The parser uses functional stopping symbols to ensure all syntax errors are detected. Furthermore the parser handles functions in the definition file. These features are described in more detail in section 5.

GUI

The `gui.py` module calls `UserInterface` which executes the completed network for a given number of cycles. Once this has been completed successfully, it calls the `Diagram` class which constructs the logic circuit diagram and saves it as an SVG. From then on, the back-end processes are largely complete - the GUI is built and the definition file, error messages, signal traces and logic diagram are loaded in.

The `gui.py` module also handles user interaction with the GUI. This could involve running the circuit for longer, adding monitoring points and changing switches. If the machine the code is running on is French, all messages generated by the GUI are translated. Locale detection takes place in `logsim.py`, however the message which are translated are in `gui.py`. *Internationalization* is described in more detail in section 5.

2.2 Client-side

The logic simulator has 4 main interfaces: a text editor for creating and editing definition files, an output prompt printing messages, a canvas displaying the signal trace and a logic circuit diagram generated from the definition file. Together the text editor and output prompt present errors to the user and, while the signal trace and circuit diagram describe a functioning logic circuit. A more visual description of the simulator and a user-guide is at the end of this report in appendix C.

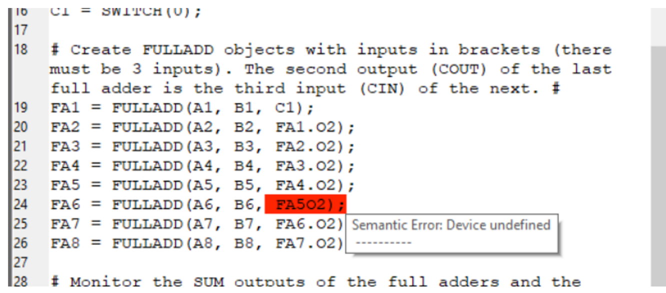
Text Editor

The definition file may be edited in the text editor, which shows live semantic and syntax errors. This file may be passed to the signal trace if there are no errors. Furthermore, file changes can be saved and other definition files may be loaded into the text editor while the simulator is running.

Output prompt

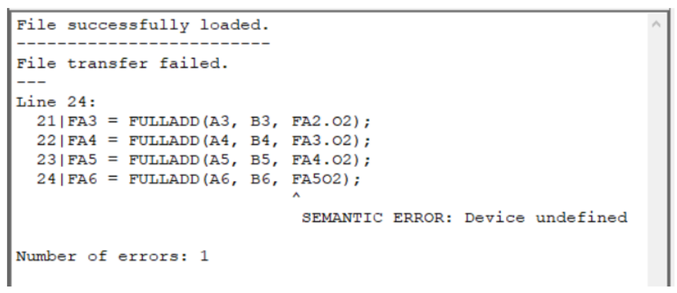
After the initial implementation of the parser, error messages were only printed to the command line. To print them to the GUI an `error_dict` dictionary was added to the parser to record printed error messages. These are then passed to the GUI where they are displayed in the output prompt [Fig. 3(b)]. Alongside this error messages are still printed to the command line, should the client prefer to use it.

Error handling



```
16  C1 = SWITCH(0);
17
18  # Create FULLADD objects with inputs in brackets (there
    must be 3 inputs). The second output (COUT) of the last
    full adder is the third input (CIN) of the next. #
19  FA1 = FULLADD(A1, B1, C1);
20  FA2 = FULLADD(A2, B2, FA1.O2);
21  FA3 = FULLADD(A3, B3, FA2.O2);
22  FA4 = FULLADD(A4, B4, FA3.O2);
23  FA5 = FULLADD(A5, B5, FA4.O2);
24  FA6 = FULLADD(A6, B6, FA5O2);
25  FA7 = FULLADD(A7, B7, FA6.O2)  Semantic Error: Device undefined
26  FA8 = FULLADD(A8, B8, FA7.O2)  ----------
27
28  # Monitor the SUM outputs of the full adders and the
```

(a) Live error detection



```
File successfully loaded.
-------------------------
File transfer failed.
---
Line 24:
  21|FA3 = FULLADD(A3, B3, FA2.O2);
  22|FA4 = FULLADD(A4, B4, FA3.O2);
  23|FA5 = FULLADD(A5, B5, FA4.O2);
  24|FA6 = FULLADD(A6, B6, FA5O2);
                           ^
                            SEMANTIC ERROR: Device undefined

Number of errors: 1
```

(b) Error message in output prompt

Figure 3: Simple error caused by no dot between the device name and input name

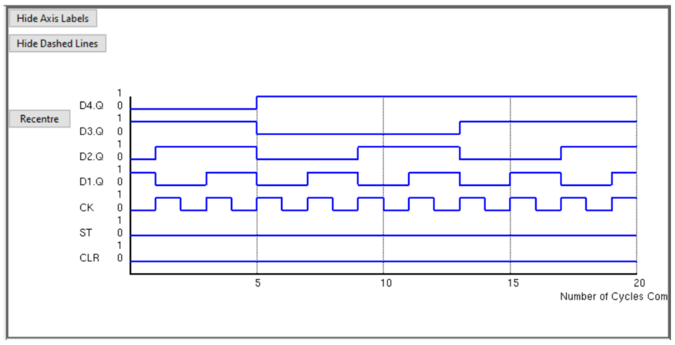
Inspired by commercial Python development environments such as PyCharm, it was decided to implement live error parsing [Fig. 3(a)]. The simulator continually tests the definition file for syntax and semantic errors. The program will return all syntax errors, but only the first semantic error from the Devices and Network classes as these are only called when the network is error-free.

For larger circuits this live error parsing proved sluggish. Therefore a 0.1 second delay since the last typing event was imposed to parse the definition file in the text editor. This improved performance and, in turn, user experience.

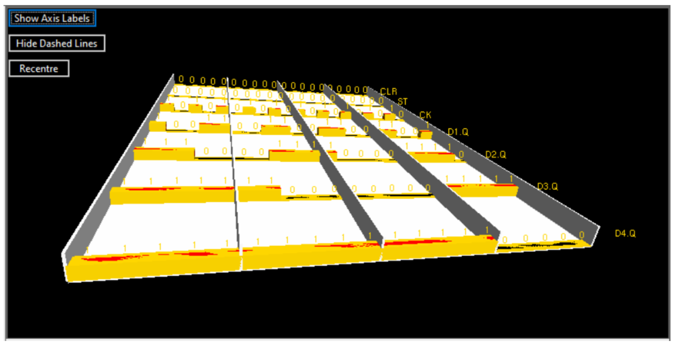
Beside live error parsing is standard error reporting if the user attempts to execute the network [Fig. 3(b)]. The offending line of code is printed, plus the preceding 3 lines for context. The location of the error is denoted with a carat (^).

Signal trace

The signal trace displays the signals of the monitored device outputs, either in 2D or 3D. The trace can be zoomed, stretched, re-centred and fitted. It may be run from current or from the start. The signal trace can also be presented in 3D, with yellow blocks representing a '1' signal.



(a) 2D

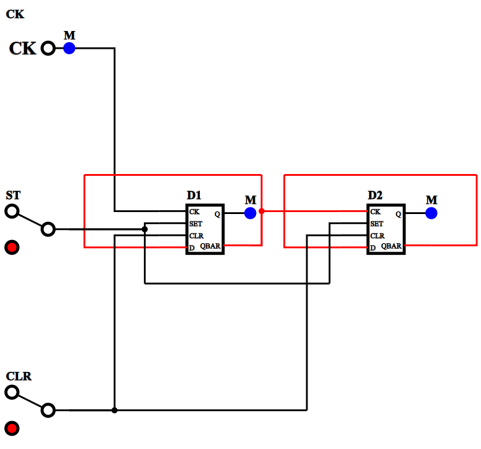


(b) 3D

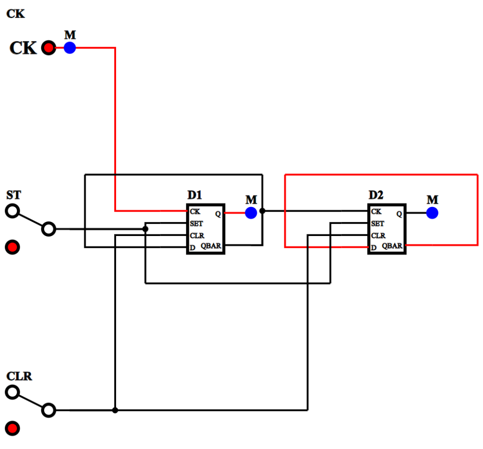
Figure 4: Ripple counter signal trace

Circuit Diagram

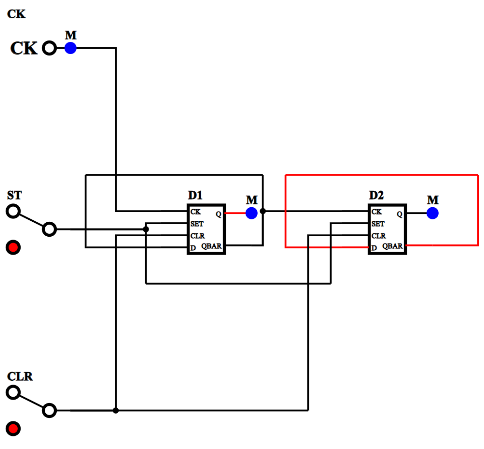
When a network is successfully executed, the program generates a diagram of the circuit. This has been designed to be as clear as possible, with few line crossings and dots to show connected lines. The line colours indicate the signal state of the connection: black for 0, red for 1. Moreover, if the user runs the simulator, the logic circuit updates its signal values, thereby changing the connections' colours [Fig. 5].



(a) Initially D1.Q = D2.Q = 0



(b) Clock rises: D1.Q = D1.DATA = 1



(c) Clock falls: no change

Figure 5: Ripple counter circuit diagram

The principal reason this feature was added is to aid the client's debugging process - it is much easier to visually decode what is wrong with a circuit than by using monitor signal traces. Furthermore the circuit adds some colour and charm to the simulator. Alongside the diagram in the GUI a full size SVG is saved into the logsim folder. Initially the aim was to insert the SVG directly into the GUI, however wxPython could not read the text in the SVG. Therefore it is converted in the `gui.py` module to a PNG, before being sent to the GUI.

A more in depth description of the function and characteristics of this circuit diagram is section 5: 'Personal contribution to the software'.

2.3 Software structure

The software is chiefly coded in a functional design, with one or two classes per module, each with many methods. This was appropriate as this task involves multiple disparate tasks, which are relatively small when compared to other real-world software systems. As a result inheritance was not used for the back-end as not many of the methods and none of the classes shared attributes which would have benefited from it. Furthermore one class per module was how the client provided the code, so it was best to maintain this style. On the other and, the GUI did involve object-oriented programming, as this is the easiest way to handle wxPython.

Data structures

This project uses multiple Pythonic data structures to optimize software performance an readability, notably lists, dictionaries and classes. A key area in which dictionary structures are utilised is within `parse.py` where each `devicename` and `port` symbol is assigned a dictionary with its ID and position for accurate semantic error reporting. Similarly `draw.py` uses these simple data objects to keep track of which lanes in the circuit diagram are busy, and for which co-ordinates they are busy from and till. The use of classes allows attributes to be shared amongst methods, neatening the code, The style of classes made in the project is inspired by the supplied code

Software design patters

The function of the logic simulator lends itself to the *Singleton* Python design pattern. This is because many of the key classes, such as `Devices`, `Network` and `Diagram` are global for each running of the simulator. That is, there should be only one instance of each as they fully define and store the properties of the circuit being created. On the other hand, there are classes which describe the properties of multiple objects, such as `Device`. In this respect, our project is built under the Facade design pattern, as `logsim.py` acts as a facade to multiple instances of objects on the back-end which would otherwise not interface.

3 Teamwork

3.1 Approach

The approach taken to teamwork was to identify three workloads to build the logic simulator which were as independent as possible. This allowed us to concentrate fully on our respective responsibilities and make fast progress, while minimising `git merge` issues. Abhinav had prior experience with OpenGL so he worked mainly on the front-end side of the simulator, while Jasen and myself concentrated on the back-end. Inevitably, however, there was some cross-over and we made sure to notify team members when we encroached on their modules. All team members had a hand in every module due to the inter-connected design of the simulator's back-end.

	Isaac Brown	**Abhinav Heble**	**Jasen Walker**
Principal accountabilities	- `names.py` - `parser.py` - Function parsing - `draw.py`	- `gui.py` - Testing (front-end)	- `scanner.py` - Testing (pytest)
Maintenance	- Internationalization	- 3D GUI	- `SIGGEN`

Table 1: Principal accountabilites of team member

On top of this, testing other team member's code was key to our process. While Jasen wrote most of the pytests, all team members tested the software manually by probing the command line and graphical user interface.

3.2 Real versus expected timeline

The timeline presented in the first report was largely adhered to, as was the members of the team who undertook the tasks. One key difference, however, was the fact that certain tasks such as implementing the parser didn't have an *end* as such, since the changes we made up to the end of the project affected the modules before it. Another difference was that we added more work for ourselves, for instance the automatic circuit diagram. This was possible as the basic scanner and parser were finished within 4 days of the first interim report deadline,

The 'System integration' activity was executed differently to how we expected. Rather than 4 days at the end of the project as planned, this took far less time as we had continually integrated and tested our changes through `git`. Conversely, tasks which took longer than expected were the GUI and Internationalization. We began the GUI earlier than expected and kept working on it for over 2 weeks as there was plenty of scope to add more features, such as the text editor. The completion of internationalization is described in section 5.

4 Remote Collaboration

Our team collaborated well remotely, aided in the main by two technologies: GitHub and online meetings.

4.1 GitHub

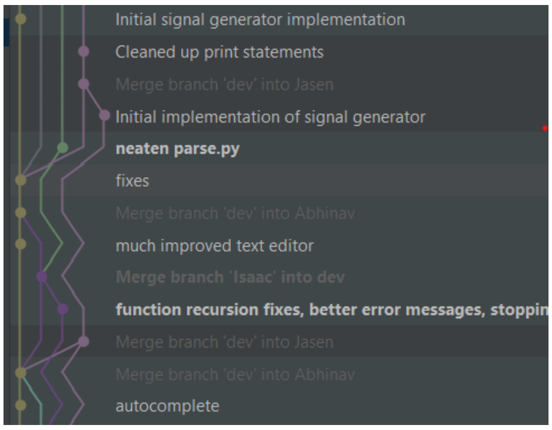
I created the GitHub repository with the supplied code and added the team as collaborators at the start of the project. The team all had Python development environments with GitHub integration, avoiding using the command line to interact with `git`. Our collaborative version control tactic was to have 5 branches to neatly integrate our work:

- 1 **`master`** branch: This branch had the most up to date stable version of the logic simulator on it. Only `dev` merged into `master`, and only after extensive testing. This generally happened before report deadlines and the peer-review. Master was never directly edited by a team member.

- 1 **`dev`** branch: This is the branch which team members merged their changes into on a regular basis, for continuous system integration. After merging our individual feauture branches into `dev` `pytest -v` was run before pushing back to `origin` (GitHub) to ensure any unintended consequences were caught before other members pulled the most up to date version of `dev`.

- 3 team member feature branches, named **`Isaac`**, **`Abhinav`** and **`Jasen`**: Each team member has their own branch to push feature changes to. We aimed to be working on separate modules at a given time to avoid merge conflicts. Once features on these branches had been tested they were merged into `dev`. Our individual branches were no further than 24 hours from the `dev` base, to keep on top of changes. This was achieved through merging in changes or re-basing.

I use the PyCharm IDE, which has GitHub integration and tools for viewing changes to files [Fig. 6]. This made using GitHub and identifying merge errors far easier.



```
Initial signal generator implementation
Cleaned up print statements
Merge branch 'dev' into Jasen
Initial implementation of signal generator
neaten parse.py
fixes
Merge branch 'dev' into Abhinav
much improved text editor
Merge branch 'Isaac' into dev
function recursion fixes, better error messages, stoppin
Merge branch 'dev' into Jasen
Merge branch 'dev' into Abhinav
autocomplete
```

Figure 6: Pycharm Git integration

4.2 Remote meetings

We had virtual meetings approximately every other day, usually before or after a demonstrator meeting. We discussed what we had been doing, any errors we had noticed and planned what we would each do for the day ahead. On top of this we have a group chat to discuss smaller problems. Abhinav and myself live in the same household, so there was some in person collaboration. Nonetheless the vast majority of each of our work was done in isolation. We had 2 in person meetings with all team members present - this blended approach of remote and in person collaboration was perhaps most useful for boosting team chemistry.

5 Personal contribution to the software

We decided early on that my key responsibility would be the parser module, however after implementing a basic version I decided to enrich some of the aspects of the back and front-ends of the logic-simulator.

5.1 `names.py`

I made the first commits to the repository by implementing the full `names.py`, for which a skeleton had been provided. While the module itself was quickly finished, far longer was creating a pytest file to automate testing. This involved a steep learning curve and analysing the code provided in the preliminary exercises. I made `names.py` and `test_names.py` as soon as possible as it was a bottleneck stopping Jasen from beginning work on the scanner module.

```python
# Test lookup by passing lists of names into a used Names instance
@pytest.mark.parametrize("test_names_list, expected_id_list", [
    (["Eve", "Alice", "Bob", "Eve"], [2, 0, 1, 2]),
    (["Eve", "Isaac", "Abhinav", "Jasen", "Alice"], [2, 3, 4, 5, 0]),
    (["Eve", "Isaac", "Abhinav", "Dave", "Jasen", "Eve", "Bob"],
     [2, 3, 4, 5, 6, 2, 1])
])
def test_lookup(used_names, test_names_list, expected_id_list):
    assert used_names.lookup(test_names_list) == expected_id_list
```

Listing 1: `test_names.py` pytest example for testing correct outputs from `names.lookup`

5.2 `parse.py`: Error reporting

Abhinav and myself began the parser module together, before he started work on the GUI. We each completed syntax parsing of incoming symbols from the scanner based on the EBNF presented in our first report. At this point syntax errors led to a funciton which only passed. I then implemented simple error handling, where the parser would skip to the next semicolon when it found a syntax error. However, such an approach is flawed in that it will not detect all syntax errors. For instance, if one device has an invalid name in a device name list, it is desirable for the parser to stop at the next comma to parse other devices rather than skipping to the end of the line.

To achieve this I made use of *stopping symbols*. Each syntax parsing function passes its own stopping symbols to a function it calls. These stopping symbols are any symbols which have yet to be read assuming the definition file is syntactically correct. For example, in Listing 2, calling `__devicenamelist` from `device` on line 512 requires an extra 'equals' stopping symbol as the parser may detect further errors after the equals sign in a device line. Contrastingly, the `__devicename` call on line 526 does not require any further stopping symbols as the equals sign has already been passed successfully. Stopping symbols are not an attribute of the class as it otherwise symbols would have to be removed from the stopping symbols list after the function had been passed through, requiring more lines of code.

```
512     def __device(self, stop_symbols):
513         """Return device attributes.
```

```
518         device_name_list = self.__devicenamelist(stop_symbols +
519                                                  [self.scanner.EQUALS])
520         device_type = None
521         device_param = None
522         if self.symbol.type == self.scanner.EQUALS:
523             self.symbol = self.scanner.get_symbol()
524
525             # Get device type and parameter and try making device
526             device_type = self.__devicename(stop_symbols)
```

Listing 2: `parse.py`. The `__devicenamelist` call is passed an '=' as an additional stopping symbol.

In the event of syntax error, the private method `__syntax_error` is called, with the error message and stopping symbols being inputs. After the message has been handled, symbols are skipped until a stopping symbol, end of file or `END` is reached [Listing 3],

```
370     def __syntax_error(self, reason, stop_symbols):
371         """Print error message and move to next stopping symbol."""
```

```
389         # While not ';', end of file or END, keep skipping symbols
390         while (self.symbol.type not in stop_symbols and not
391                (self.symbol.id == self.scanner.END_ID and
392                 self.symbol.type == self.scanner.KEYWORD)):
393             self.symbol = self.scanner.get_symbol()
```

Listing 3: `parse.py`. Symbols are skipped until a stopping symbol, EoF or `END`, whichever is first

Semantic errors are detected in one of two ways: after being passed to an external class such as `Devices`, or detected by the parser itself. An example of an error detected by the `Devices` class is an invalid port ID, whereas an example of a semantic error detected by the parser is if a device name being defined has a port ID. When the error count is non-zero, the three helper classes are no longer called, preventing any more semantic errors from them being reported.

In order to report the position and nature of semantic errors from the `Devices`, `Network` or `Monitors` classes, I modified them by adding a dictionary mapping error codes to location numbers and messages. The parser then interprets this number to return the position of the symbol with the error. Every `devicename` and `port` consists of a dictionary with keys '`id`' and '`pos`' (position) so that when one is determined to have a semantic error, its location may be printed in the output prompt.

5.3 `parse.py`: Implicit and explicit devices

A key feature of our EBNF presented in the first report was the support of *implicit* devices, as well as *explicit* devices. Impicit devices are of the form:

`device_name = device_type(input_1, input_2, ...  , input_n)`

This allows logic diagrams to be described much faster by the end user. These were implemented by creating two methods within `parse.py`, `make_implicit_device` and `make_explicit_device`. If the parameter is a device name list `make_implicit_device` is called, otherwise `make_explicit_device` (the standard implementation of devices) is called. In `make_implicit_device`, the device type is made with the device name and the inputs are iterated through to create the implied connections.

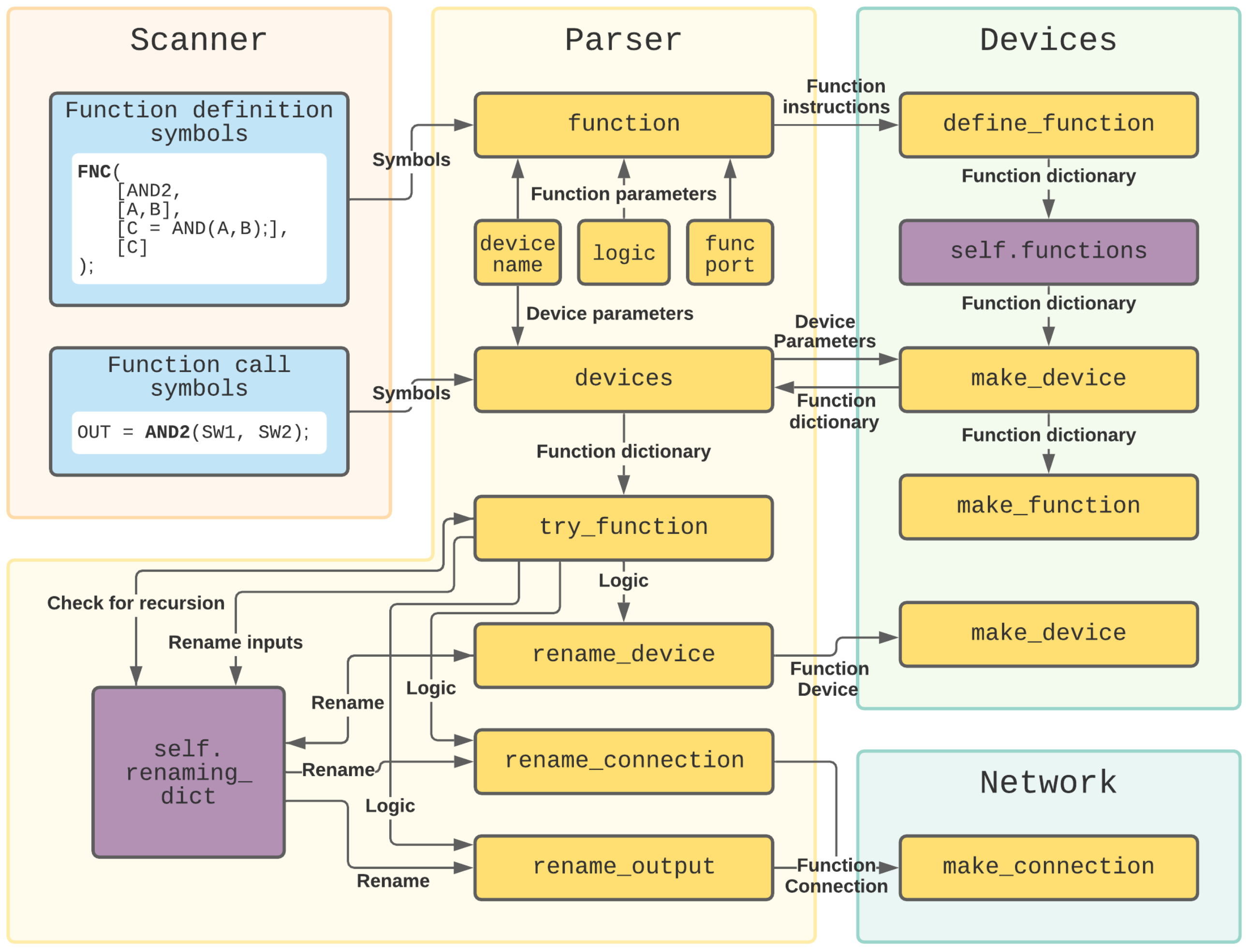
5.4 `parse.py`: Functions

A key feature our first report introduced was the option for the client to describe their logic circuit in a functional way, in order to simplify their definition file. Functions are primarily dealt with by the `Parser` and `Devices` classes, the latter of which had to be modified to include methods for storing function definition files. Broadly, the process for parsing functions has 3 main stages: storing the function's instructions when it is defined, creating renamed devices from the function when it is called, and inserting a dummy device at the device outputs which are defined as the function outputs. Below each of these steps are described:

Storing function instructions

Functions must be defined before they are used, in order to stay within the recursive descent paradigm which simplifies parsing. A function definition will consists of a function name, function inputs, logic, and function outputs. All of the device names described in the definition of the function are *local*, that is they are not callable outside the function and do not represent actual device names. Instead, when a function is called, these local device names are renamed to a unique string which may not be called directly by the rest of the definition file (see next section).

The parser detects the function definition, parses it using helper functions and then sends its instructions to `Devices.define_function`, which stores them in the `Devices` attribute `self.functions`. This is a dictionary which stores defined functions for later use.



```
FNC(
    [AND2,
    [A,B],
    [C = AND(A,B);],
    [C]
);
```

```
OUT = AND2(SW1, SW2);
```

Figure 7: Functions parsing block diagram. Variables are in blue, methods in yellow and attributes in purple. Some methods are left out for visual purposes.

Function call and renaming devices

Later, when a function is called, the parser detects it as a `device` line as it is syntactically indistinguishable from a standard device line. The 'device' parameters are then sent through `Parser.try_device` and `Parser.make_implicit_device`, into `Devices.make_device`. This method detects it as a function by searching its attribute `self.functions` for the `device_kind`. At this stage it makes a dummy function with the same name as the device set equal to this instance of the function - in [Fig. 7] the dummy device would be called '`OUT`'.

The function dictionary containing instructions is sent back to the parser, which then sends it to `Parser.try_function`, the main method for building devices and connections described by an instance of a function. Firstly, this method checks for recursion by searching the `Parser` attribute `self.renaming_dict`. This attribute keeps a record of all the active functions and their respective renaming dictionaries for mapping local names to device names in the logic circuit. If the same function kind is present in the `self.renaming_dict`, the function must call in a loop back to itself, which is recursion and is disallowed by our program. It is disallowed as it is un-physical for a logic circuit and more difficult to program.

Once `Parser.try_function` has checked that the number of inputs given is equal to the number of inputs to the function definition, each of the local function input names are mapped to the real device inputs of the function instance. In [Fig. 7], for example, `A` and `B` are mapped to `SW1` and `SW2` respectively. These mappings are saved in the `renaming_dict` attribute so that future references to the local function inputs `A` and `B` are mapped to `SW1` and `SW2`.

Next, each line of logic in the logic section of the function definition is sent to either `rename_device` or `rename_connection`. These methods get the mappings from local name to circuit name from `self.renaming_dict`. New devices defined in the logic are added to `self.renaming_dict` and are named by concatenating the name of the function instance to the local name of the device, separated by an underscore. For example, in [Fig. 7], the local device '`C`' would be renamed '`OUT_C`'. This ensures devices made by functions are unique as the function instance name must be unique and the local device name must be unique within that function. Furthermore, the underscore ensures the device cannot be called from outside the function as the scanner returns 'invalid character' if it meets an underscore. Once all devices in the logic line have been renamed, these lines are sent to the `try_device` and `try_connection` methods as usual.

Function outputs

Our initial concept involved being able to reference outputs of functions with the syntax `.On` for the nth output. To realise this, a dummy device (outputs = inputs) is created with the same name as the function instance. Its inputs are the devices created by the logic in the function which have been specified as outputs by the function definition. This dummy device is made in the `Devices` class through `make_function`. Similarly to existing devices, if the function has 1 output, it is referenced by only the device name (i.e. not `.O1`).

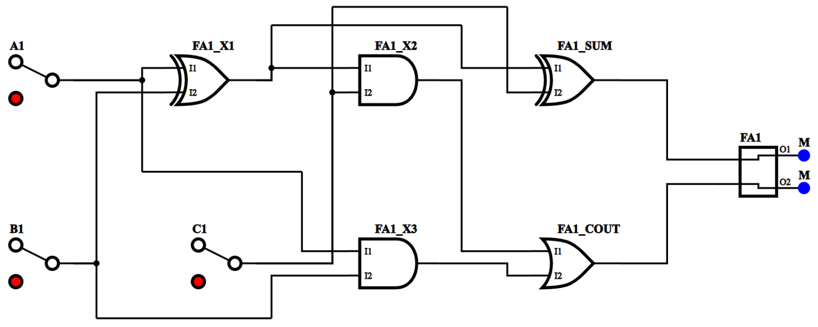


Figure 8: Full adder circuit diagram

Perhaps the clearest way of illustrating the dummy devices is by viewing one of the circuit diagrams generated by `draw.py`. In [Fig. 8], the dummy device `FA1` is present simply to allow the outputs of the function to be referenced `FA1.O1` and `FA1.O2`. A drawback of this 'dummy device' solution is that functions cannot be created explicitly as their inputs cannot be referenced by '`.In`' for the nth input.

5.5 `draw.py`: Circuit diagram

The circuit diagram aims to present the circuit as clearly as possible. This is achieved by laying out the devices in an orderly fashion and minimising crossed connections while maintaining straight connections. The `Diagram` class consists of two public methods, one for making the diagram from scratch and another for updating it with new signal vales. There are 4 key steps taken to draw the circuit diagram:

- Calculate the 'depth' of the devices into the network, i.e. how far right they are, x-value.
- Assign each device a 'height' (y-value) in order to minimise wire crossing; draw the devices.
- Calculate the routes of connections and the positions of any connecting dots, then drawing them.
- The diagrams connections' colours are updated by measuring the output signals after every run in the GUI.

Device depth

The device depth is first calculated from the furthest distance of each device from the inputs. This was the initial implementation [Fig. 9(a)], however for large circuits, such as the 8 bit full adder, it generated a long row of inputs, making it hard to read. Consequently, the second step in determining depth is shifting each device as far rightward as possible, until they reach their closest output device. This results in a much clearer diagram [Fig. 9(b)].

Device height

Device height is calculated from working backwards from the output devices. A devices height is the average height of its output devices. The devices in a column (or level) are then ranked and spaced 1 unit apart, before being snapped to the nearest 0.5 height units. As a result all devices fall on an orderly grid. The devices are drawn on the SVG canvas by inserting pre-defined device shapes in the required position.

Connections

Starting from the left most devices, devices are connected to their inputs.'Lanes' between columns of devices are initialised as lists of busy sections where connections are using them. Two connections with different inputs may not lie on the same lane in the same position. Rules-based calculations are made to handle connections between non-adjacent device columns and connections from the output of one device back to itself.

Early versions of the circuit diagram had no connection dots, making it difficult to know whether crossing connections were supposed to be connected or not [Fig. 9(c)]. Adding dots when lanes encountered two connections of the same input overlapping fixed this issue [Fig. 9(d)]. Connections are plotted on the SVG canvas using SVG `paths`.

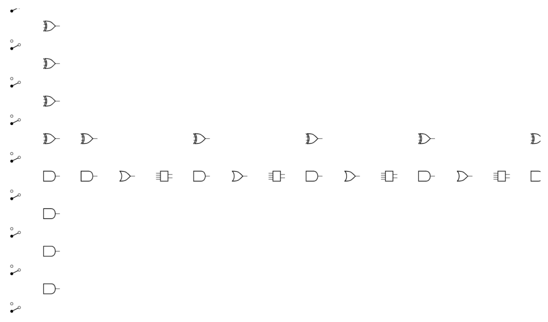
Signal colours

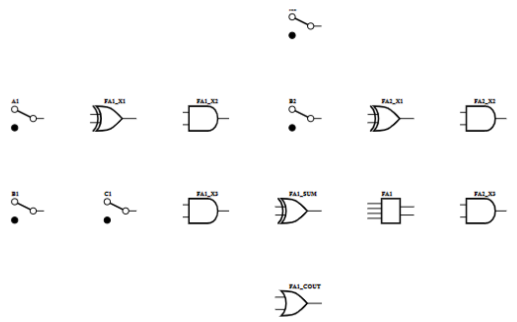
Once the devices and connection positions have been calculated, the signal value of each connection is fetched from the `Devices` class [Fig. 9(e)]. Connections signal '1' are red, connections signal '0' are black. In addition switches move position from on to off. These colours are created when the Diagram is executed for the first time and when the diagram is updated after user actions.

5.6 `logsim.py`, `gui.py`: Internationalization

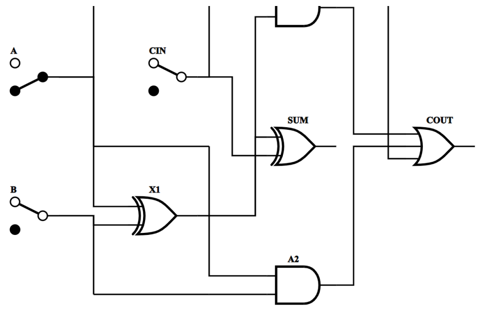
My maintenance task was adding a second language option to the logic simulator. This automatically detects the nationality of the machine and converts the GUI's text to French as appropriate. Firstly, I used `_(<message string>)` to allow `pygettext` to locate the translatable strings and store them in a `.po` file. Next came the actual translation - I translated the text using Google translate and pasted them into the `msgstr` parts of the `.po` file. Some of the French characters were not UTF-8, so I had to convert these translations into UTF-16 unicode before pasting them in. This `.po` file is then converted to a `.mo` (Machine Object) file using the `msgfmt` Python inbuilt tool.

This file was named `gui.mo` and placed in the folder `locale/fr/` so that wxPython's Locale class could locate it. When a machine is set to French, the program now automatically detects this and translates the text. One issue I had was getting the program to run in French when only Logsim was set to French using `LANG=fr_FR.utf8`. No matter which recommended environment variable I used, it did

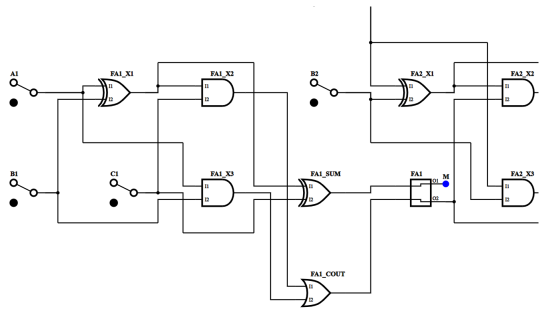

(a) Initial devices depth implementation



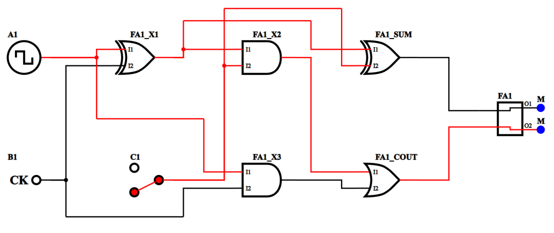
(b) Shifted device depths



(c) Connections, no dots



(d) Connections, dots



(e) Signal colours

Figure 9: Full adder circuit diagram

not recognise the command, plus when I manually added a 'LANG' environment variable it had no effect. Neither of my teammates could get this to work either.

Finally, as required by the maintenance task, the GUI displays non-latin characters in the right pane, with several international translations of the word 'Logic'.

5.7 Code style and layout

I took time and care to ensure the readability of my code. On top of `pycodestyle` and `pydocstyle`, I aimed to keep functions short enough to be viewed without scrolling. Before every section of code (roughly 5 lines) I made sure to describe the upcoming actions with a single line comment. These blocks are spaced out for readability. Furthermore I aspired decrease nesting of loops and conditional statements within functions in order to make most use of the 80 character line limit.

5.8 Other contributions

In all I contributed to all modules, due to the interdependence of the logic simulator. Some examples are:

- **`scanner.py`**: The print statement in `print_line` to preint error messages neatly.
- **`gui.py`**: Functions to call `draw.py` to update the diagram and send this update to the GUI.
- **`test_parse.py`**: I implemented the batch file tests which are asserted to contain a set number of errors.

6 Testing

We took a blended approach to testing, with most of the back-end processes tested automatically using pytest, and the entire program tested manually by the team. Jasen wrote the pytests for my file `parse.py` as it is more effective to test each others work, especially for such a large and complex module. Indeed, he found several bugs with functions once I had implemented them. The automated testing was especially useful for quickly checking changes to the software. Nevertheless, it is difficult to implement automatic testing for the GUI, so this was tested extensively through usage. The objective was to stress the system as much as possible to replicate client-misuse and identify any unforeseen bugs.

7 Conclusions

Overall this was a successful project - we completed the task fully, producing a working logic simulator. In addition, we went beyond the basic requirements, implementing implicit devices, functions, a text editor with live error parsing and an automatic logic circuit diagram. We worked very well as a team, rarely working on conflicting modules and every member contributed well. Our use of `git` was seamless, albeit after a couple of days of getting used to merging without conflicts.

7.1 Recommendations for improvements

Most of the improvements that could be made to this project stem from the short time period we had to complete it.

- More testing: Ideally we would have spent a concerted week testing every feasible input to the simulator to check its functionality. Clearly the timeline of this project prevents that. In particular, we would have wanted to spend more time on `draw.py` - it was only properly tested on small logic circuits.
- Loops in the definition file: While functional programming speeds up the user experience to a degree, our language still takes time to describe systems with many devices, for instance a 256-bit adder. Implementing a syntax for describing loops would remedy this, as well as developing the grammar into a more rounded programming language.
- Function 'wrappers' rather than 'dummy devices': Function outputs are referred to with the syntax `.On`, this is achieved through a 'dummy device'. In order to refer to function inputs with `.In`, a new device with the characteristic of a 'wrapper' would have to be placed over the devices described by a function. This would also allow functions to be described explicitly and connect functions recursively.
- `draw.py` classes and inheritance: If we were to make the `Diagram`, we would employ a more object oriented approach. For example, currently when devices are called to be drawn it leads to a simple function. However each of these devices share attributes such as input numbers and shapes. Therefore this would have been an ideal setting to employ inheritance.

Appendix A: Example Definition Files, Circuit Diagrams, and Test Results

1 Full Adder

Figure 1 and Figure 2 describe a full adder circuit with both explicit and implicit encoding.

```
# Explicit full adder #

A, B, CIN = SWITCH(0);

X1, SUM = XOR();
X2, X3 = AND(2);
COUT = OR(2);

# Set up X1 connections #
CON(A, X1.I1);
CON(B, X1.I2);

# Set up SUM connections #
CON(X1, SUM.I1);
CON(CIN, SUM.I2);

# Set up X2 connections #
CON(X1, X2.I1);
CON(CIN, X2.I2);

# Set up X3 connections #
CON(A, X3.I1);
CON(B, X3.I2);

# Set up COUT connections #
CON(X2, COUT.I1);
CON(X3, COUT.I2);

# Monitor outputs #
MTR(A, B, CIN, SUM, COUT);

END
```

(a) Explicit Full Adder

```
# Implicit full adder #

A, B, CIN = SWITCH(0);

X1 = XOR(A, B);
SUM = XOR(X1, CIN);
X2 = AND(X1, CIN);
X3 = AND(A, B);
COUT = OR(X2, X3);

$ Monitor outputs
MTR(A, B, CIN, SUM, COUT);

END
```

(b) Implicit Full Adder

Figure 1: Two ways to encode the same full adder circuit

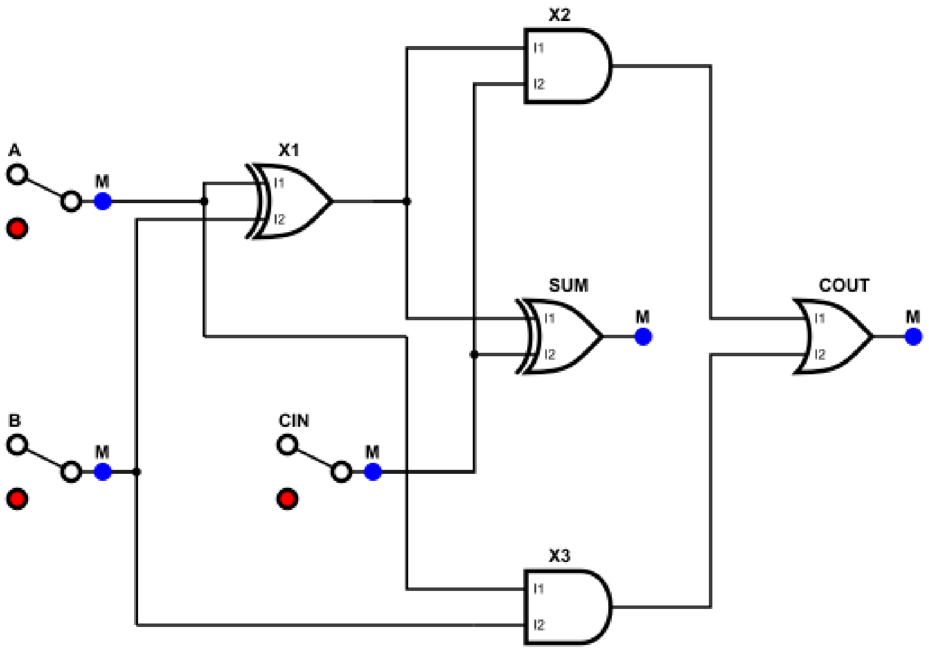
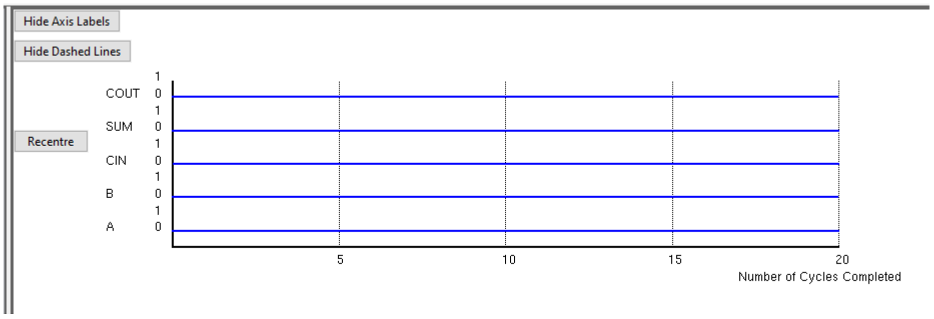
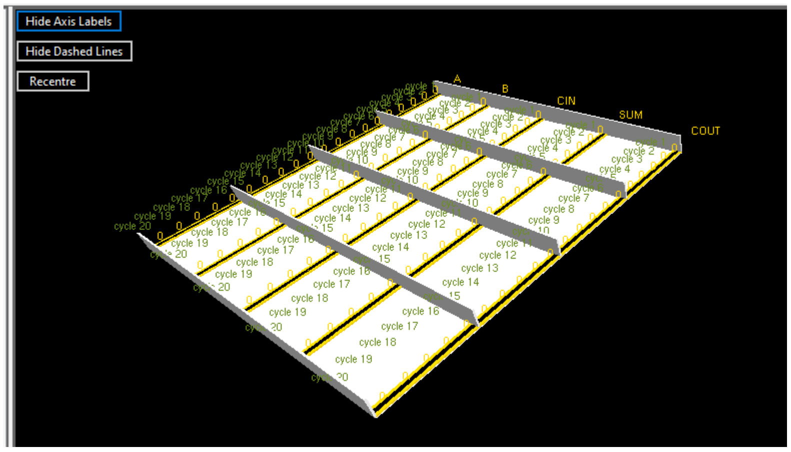


Figure 2: Full Adder Circuit Diagram

Figure 3 shows the full adder circuit, as defined above, run for 20 cycles in both 2D and 3D canvas display modes. The signal value zero is shown by a black line through the rectangle block encoding the signal in the 3D display mode. Since all the signal trace values are zero, the circuit display on simulation end is the same as shown in Figure 2.



(a) 2D signal trace



(b) 3D signal trace

Figure 3: Full adder run for 20 cycles

2 Ripple Counter

Figure 4 and Figure 5 describe a ripple counter circuit with both explicit and implicit encoding.

```
# Explicit Ripple Counter #

ST, CLR = SWITCH(0);
CK = CLOCK(10);

# Configure D-Types #
D1, D2, D3, D4 = DTYPE();

# Configure all D1 inputs. #
CON(D1.QBAR, D1.DATA);
CON(CK, D1.CLK);
CON(ST, D1.SET);
CON(CLR, D1.CLEAR);

# Configure all D2 inputs. #
CON(D2.QBAR, D2.DATA);
CON(D1.QBAR, D2.CLK);
CON(ST, D2.SET);
CON(CLR, D2.CLEAR);

# Configure all D3 inputs. #
CON(D3.QBAR, D3.DATA);
CON(D2.QBAR, D3.CLK);
CON(ST, D3.SET);
CON(CLR, D3.CLEAR);

# Configure all D4 inputs. #
CON(D4.QBAR, D4.DATA);
CON(D3.QBAR, D4.CLK);
CON(ST, D4.SET);
CON(CLR, D4.CLEAR);

MTR(D1.Q, D2.Q, D3.Q, D4.Q);

END
```

(a) Explicit Ripple Counter

```
# Implicit Ripple Counter #

# Set switches ST and CLR to 0. #
ST, CLR = SWITCH(0);

# Initialise a clock with half period 10. #
CK = CLOCK(10);

# Create a DTYPE device.#
D1 = DTYPE(CK, ST, CLR, D1.QBAR);
D2 = DTYPE(D1.QBAR, ST, CLR, D2.QBAR);
D3 = DTYPE(D2.QBAR, ST, CLR, D3.QBAR);
D4 = DTYPE(D3.QBAR, ST, CLR, D4.QBAR);

# Monitor the .Q outputs of the DTYPEs. #
MTR(D1.Q, D2.Q, D3.Q, D4.Q);

END
```

(b) Implicit Ripple Counter

Figure 4: Two ways to encode the same ripple counter circuit

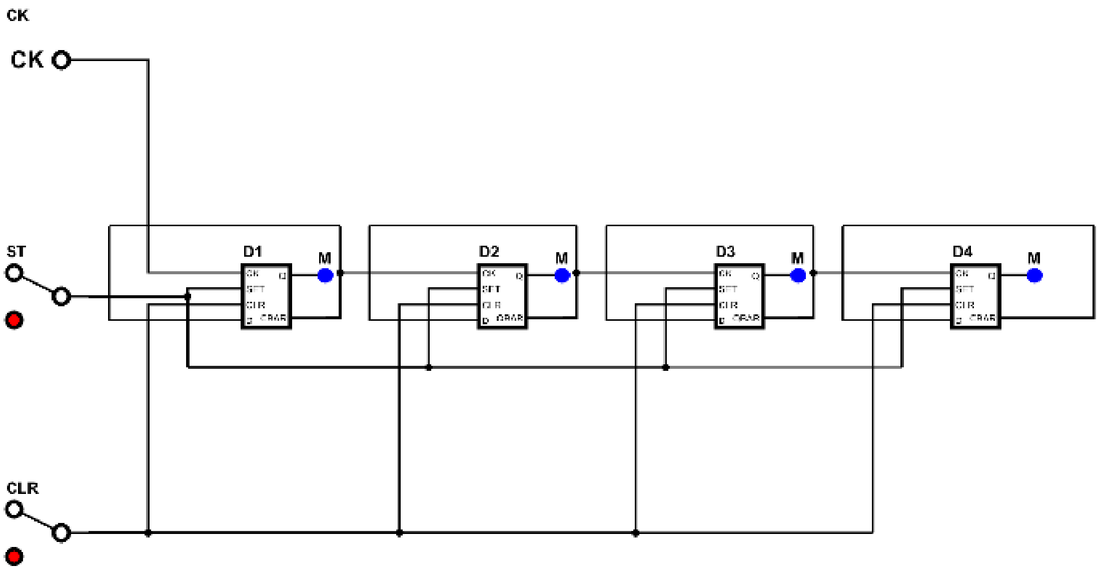
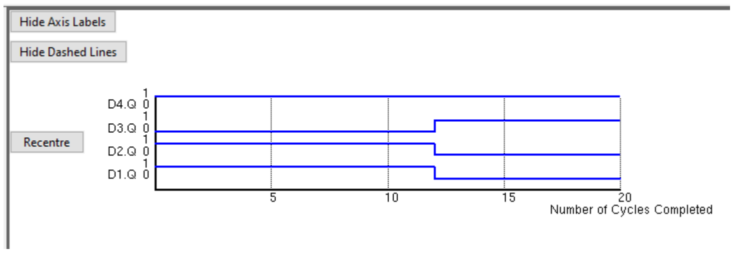
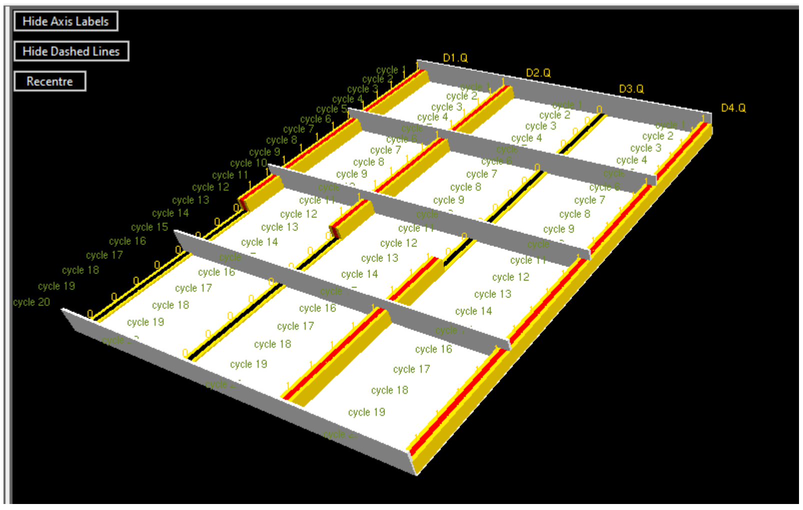


Figure 5: Ripple Counter Circuit Diagram

Figure 6 shows the ripple counter circuit, as defined above, run for 20 cycles in both 2D and 3D canvas display modes. The signal value zero is shown by a black line through the rectangle block encoding the signal, and the signal value one is shown by a red line through the rectangle block encoding the signal in the 3D display mode. Figure 7 displays the signal trace for the final cycle on the circuit diagram, with black for a signal value of zero and red for a signal value of one.



(a) 2D signal trace



(b) 3D signal trace

Figure 6: Ripple counter run for 20 cycles

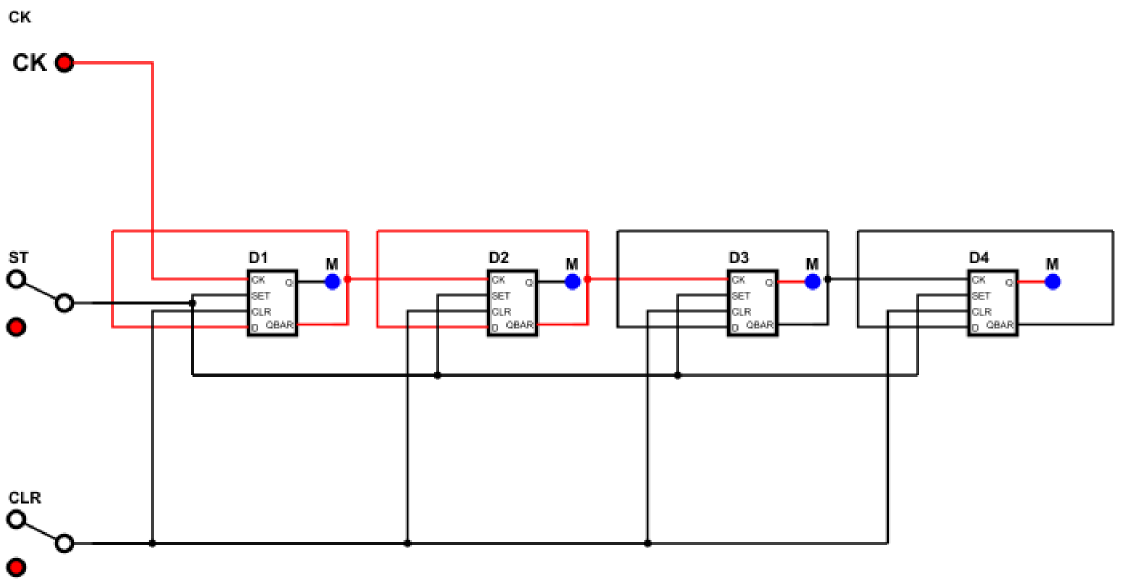


Figure 7: Ripple counter circuit diagram with signal value display

3 Multibit Adder

Figure 8 and Figure 9 describe a multibit adder using functions.

```
# Define a function "FULLADD" which takes in 3 inputs and produces
two outputs: SUM and CARRYOUT.
#
FNC(
        FULLADD,
        [A, B, CIN],
        [X1 = XOR(A, B);
         SUM = XOR(X1, CIN);
         X2 = AND(X1, CIN);
         X3 = AND(A, B);
         COUT = OR(X2, X3);],
        [SUM, COUT]
);

# Initialise input bits which may be switched later. #
A1, A3, A4, A6, A7 = SWITCH(0);

A2, A5, A8 = SWITCH(1);
B1, B2, B3, B4, B5, B8 = SWITCH(0);

B6, B7 = SWITCH(1);
C1 = SWITCH(1);

# Create FULLADD objects with inputs in brackets (there must be 3 inputs).
The second output (COUT) of the last full adder is the third input (CIN)
of the next.
#
FA1 = FULLADD(A1, B1, C1);
FA2 = FULLADD(A2, B2, FA1.O2);
FA3 = FULLADD(A3, B3, FA2.O2);
FA4 = FULLADD(A4, B4, FA3.O2);
FA5 = FULLADD(A5, B5, FA4.O2);
FA6 = FULLADD(A6, B6, FA5.O2);
FA7 = FULLADD(A7, B7, FA6.O2);
FA8 = FULLADD(A8, B8, FA7.O2);

# Monitor the SUM outputs of the full adders and the COUT of the
last one.
#
MTR(FA1.O1, FA2.O1, FA3.O1, FA4.O1, FA5.O1, FA6.O1, FA7.O1, FA8.O1, FA8.O2);

END
```

Figure 8: Multibit adder definition

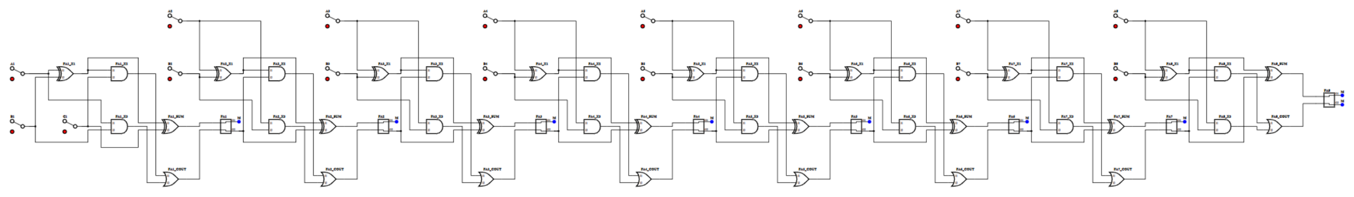
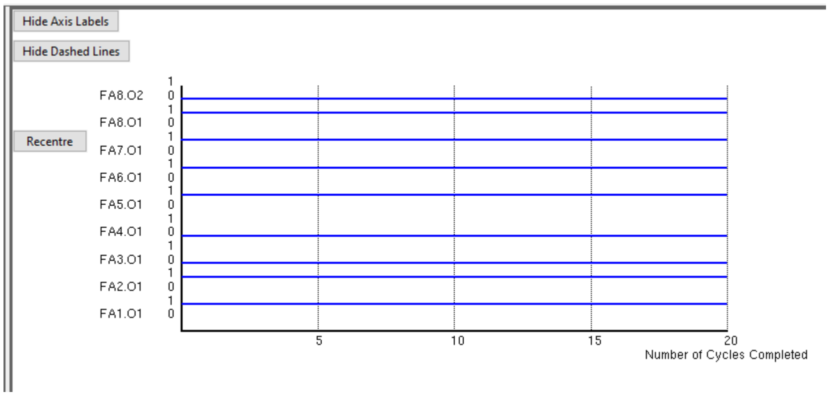


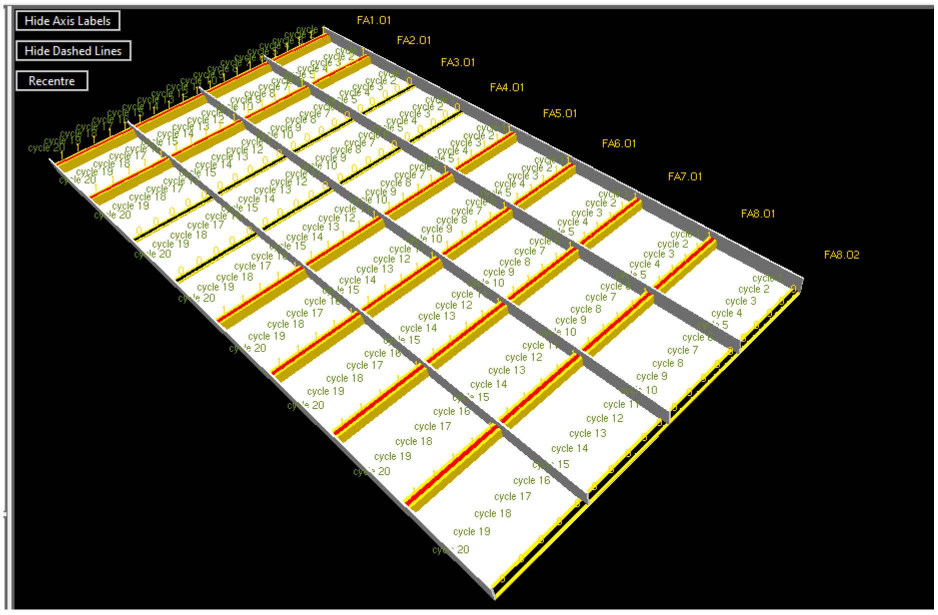
Figure 9: Multibit adder circuit diagram

Figure 10 shows the multibit adder circuit, as defined above, run for 20 cycles in both 2D and 3D canvas display modes. The signal value zero is shown by a black line through the rectangle block encoding the signal, and the signal value one is shown by a red line through the rectangle block encoding the signal in the 3D display mode.

Figure 11 displays the signal trace for the final cycle on the circuit diagram, with black for a signal value of zero and red for a signal value of one.



(a) 2D signal trace



(b) 3D signal trace

Figure 10: Ripple counter run for 20 cycles

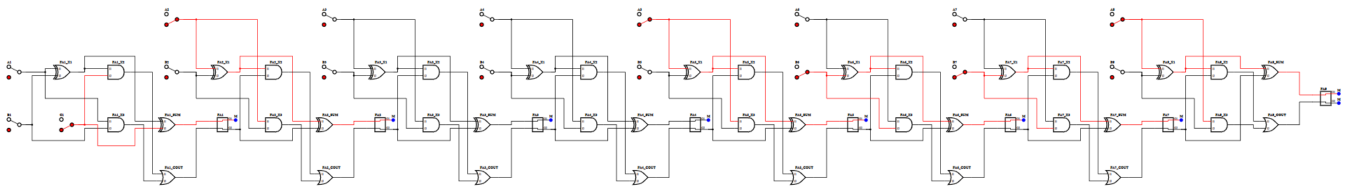


Figure 11: Multibit adder circuit diagram with signal value display

4 Custom Circuit

This circuit is an example of a simple circuit a user may write in the program, including a SIGGEN input. Figure 12 and Figure 13 describe this circuit with both explicit and implicit encoding.

```
# Explicit Custom Circuit #

# Configure Switches. #
S1 = SWITCH(0);

S2 = CLOCK(1);

S3 = SIGGEN([1,2,5,2],0);
S4 = SIGGEN([1,1],1);

# Configure Gates. #
D1 = AND(2);
D2 = OR(2);
D3 = NAND(2);
D4 = NOR(3);

# Configure all D1 inputs. #
CON(S1, D1.I1);
CON(S2, D1.I2);

# Configure all D2 inputs. #
CON(S2, D2.I1);
CON(S3, D2.I2);

# Configure all D3 inputs. #
CON(D1, D3.I1);
CON(D2, D3.I2);

# Configure all D4 inputs. #
CON(D3, D4.I1);
CON(S4, D4.I2);
CON(S3, D4.I3);

# Configure monitors. #
MTR(S2, S3, D1, D2, D3, D4);

END
```

(a) Explicit custom circuit

```
# Implicit Custom Circuit #

# Configure Switches. #
S1 = SWITCH(0);

S2 = CLOCK(1);

S3 = SIGGEN([1,2,5,2],0);
S4 = SIGGEN([1,1],1);

# Configure Gates. #
D1 = AND(S1, S2);
D2 = OR(S2, S3);
D3 = NAND(D1, D2);
D4 = NOR(D3, S3, S4);

# Configure monitors. #
MTR(S2, S3, D1, D2, D3, D4);

END
```

(b) Implicit custom circuit

Figure 12: Two ways to encode the same custom circuit

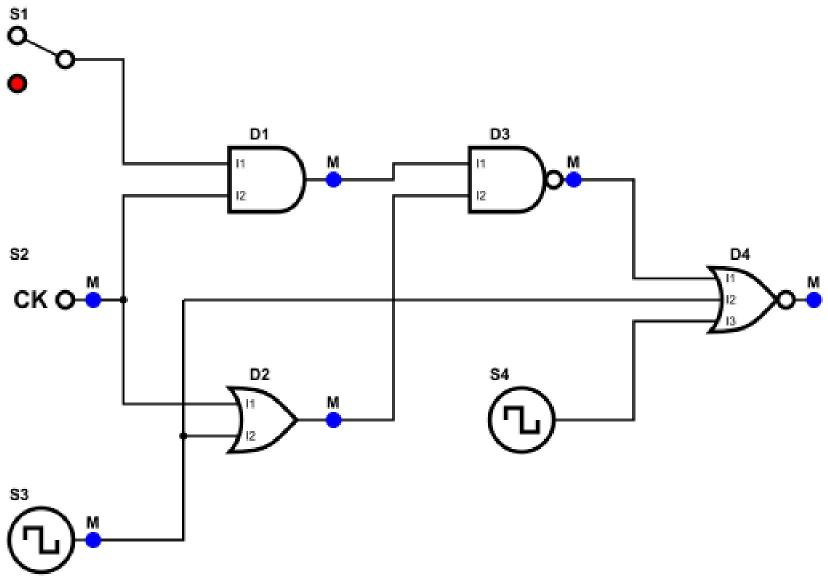
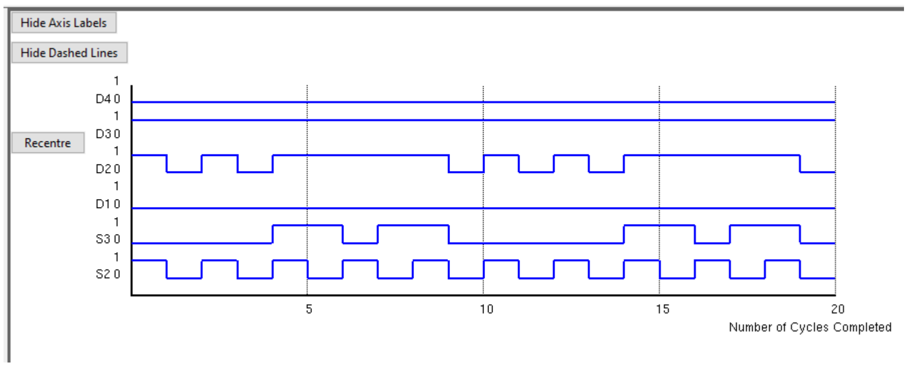


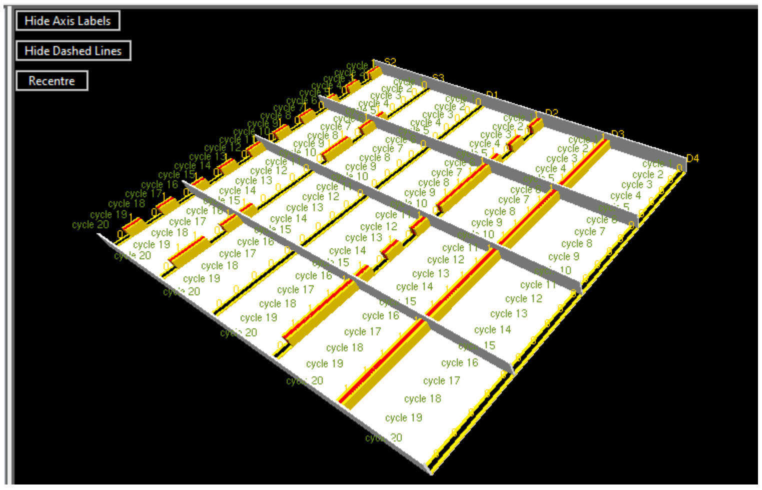
Figure 13: Custom Circuit Diagram

Figure 14 shows the custom circuit, as defined above, run for 20 cycles in both 2D and 3D canvas display modes. The signal value zero is shown by a black line through the rectangle block encoding the signal, and the signal value one is shown by a red line through the rectangle block encoding the signal in the 3D display mode.

Figure 15 displays the signal trace for the final cycle on the circuit diagram, with black for a signal value of zero and red for a signal value of one.



(a) 2D signal trace



(b) 3D signal trace

Figure 14: Custom circuit run for 20 cycles

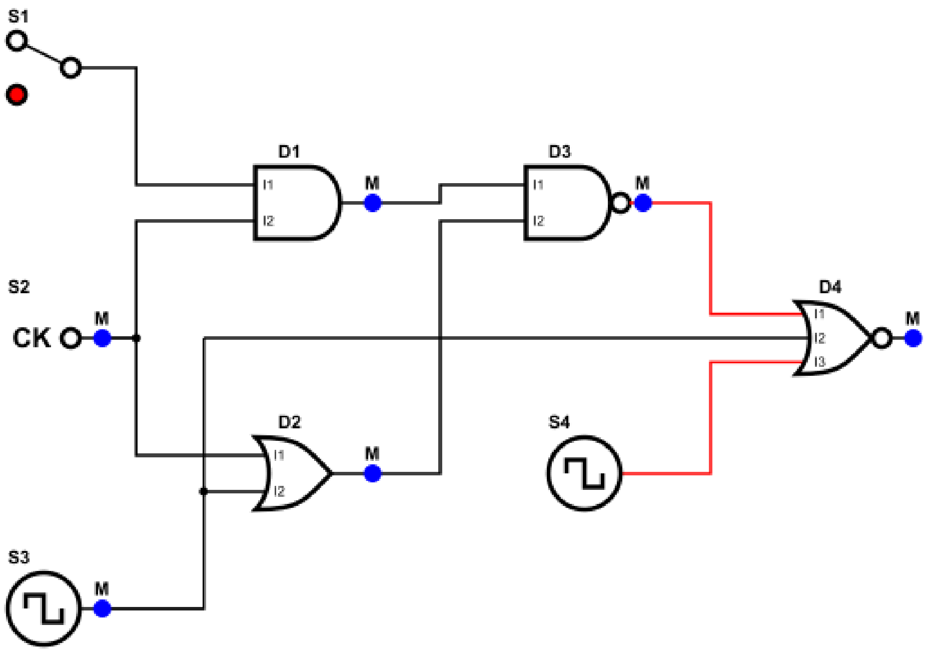


Figure 15: Custom circuit diagram with signal value display

Appendix B: Logic Description Language

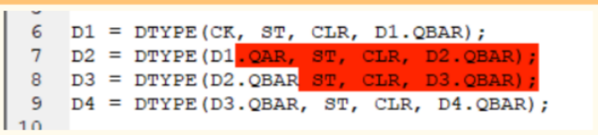
```
# A function has 4 parts: function name, inputs, logic and outputs. #
function = "FNC(", devicename, ",", funcport, ",", logic, ",", funcport, ");";

# A funcport is a list of device inputs or outputs (local to the function). #
funcport = "[", devicenamelist, "]";

# Logic is a series of devices or connections. #
logic = "[", {device | connection}, "]";

# A devicenamelist is a list of devicenames. #
devicenamelist = devicename, {",", devicename};

# A devicename is a string begining with a letter with an optional input or output id. #
devicename = letter, { letter | digit }, [portid];

# A device is a list of names being set to a type of device/gate and a parameter. #
device = devicenamelist, "=", devicename, parameter, ";";

# Connect from the output of one device to the input of another. #
connection = "CON(", devicename, ",", devicename, ");";

# A parameter is a number (e.g number of inputs for a gate, length of half period for a CLOCK, initial
    setting for a SWITCH), a list on inputs to a device, or a numberlist (wave pattern) followed by a number
     (initial state) for a SIGGEN #
parameter = "(", [number | devicenamelist | (numberlist, ",", number)], ")";

# A numberlist is a list of numbers. #
numberlist = "[", number, ",", number, {",", number}, "]";

# eg ".I1" or ".QBAR". #
portid = ".", letter, {letter | digit};

# A number is 1 or more digits. #
number = digit, {digit};

# Monitor a list of device outputs. #
monitor = "MTR(", devicenamelist, ");";

# End the file. #
end = "END"
```

Listing 1: EBNF Grammar.

Also of note, but not specified above, is that single-line comments are also supported, with the comment starting with a `$` symbol, and ending at the end of the line.

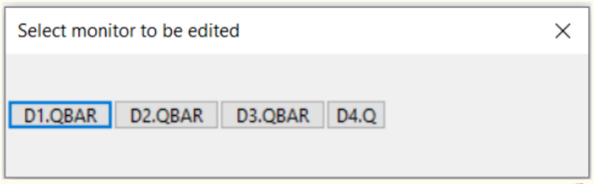
An additional requirement of any logic definition files is that all lines must either be creating a device, creating a monitor, making a connection, defining a function, the `END` symbol, a comment, or a blank line.

Appendix C: Software user guide



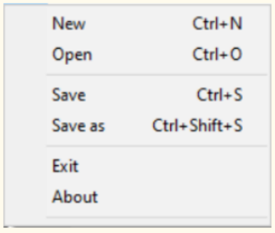
```
D1 = DTYPE(CK, ST, CLR, D1.QBAR);
D2 = DTYPE(D1.QAR, ST, CLR, D2.QBAR);
D3 = DTYPE(D2.QBAR ST, CLR, D3.QBAR);
D4 = DTYPE(D3.QBAR, ST, CLR, D4.QBAR);
```

Text editor: Live error parsing, autocomplete, hover over for info. `SIGGEN` now supported

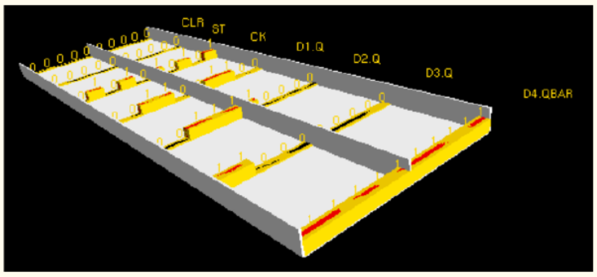


Select monitor to be edited

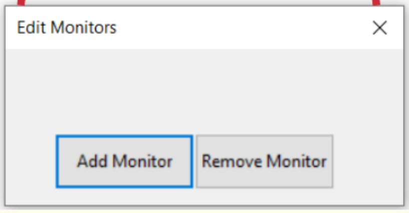
Edit circuit: Switches and monitors may be added



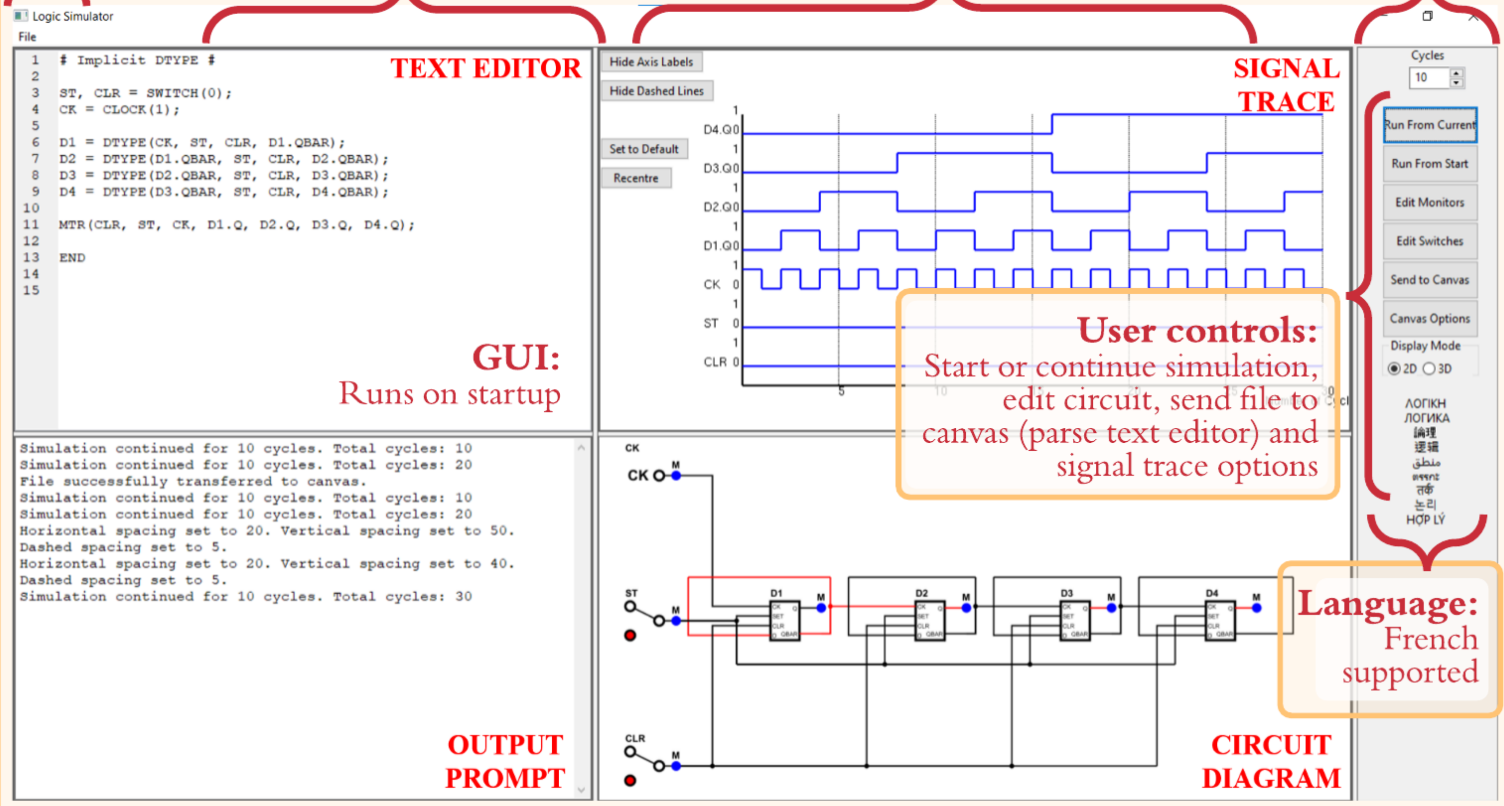
File open: Definition files opened, saved or closed here



3D signal trace: Activated by changing mode to 3D



Dialog: Some user buttons generate further options



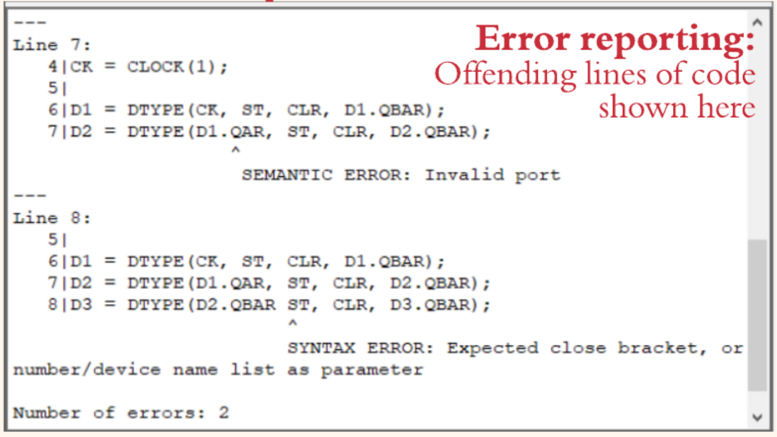
```
# Implicit DTYPE #

ST, CLR = SWITCH(0);
CK = CLOCK(1);

D1 = DTYPE(CK, ST, CLR, D1.QBAR);
D2 = DTYPE(D1.QBAR, ST, CLR, D2.QBAR);
D3 = DTYPE(D2.QBAR, ST, CLR, D3.QBAR);
D4 = DTYPE(D3.QBAR, ST, CLR, D4.QBAR);

MTR(CLR, ST, CK, D1.Q, D2.Q, D3.Q, D4.Q);

END
```

```
Simulation continued for 10 cycles. Total cycles: 10
Simulation continued for 10 cycles. Total cycles: 20
File successfully transferred to canvas.
Simulation continued for 10 cycles. Total cycles: 10
Simulation continued for 10 cycles. Total cycles: 20
Horizontal spacing set to 20. Vertical spacing set to 50.
Dashed spacing set to 5.
Horizontal spacing set to 20. Vertical spacing set to 40.
Dashed spacing set to 5.
Simulation continued for 10 cycles. Total cycles: 30
```

GUI: Runs on startup

User controls: Start or continue simulation, edit circuit, send file to canvas (parse text editor) and signal trace options

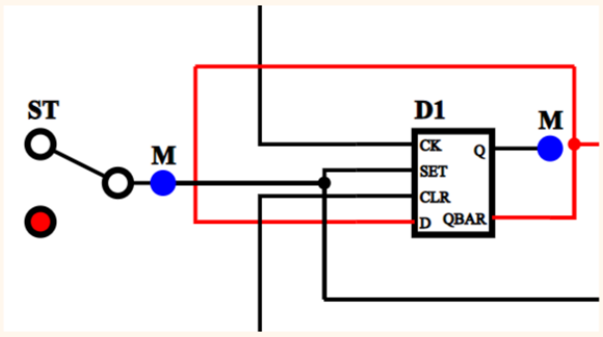
Language: French supported



```
---
Line 7:
    4|CK = CLOCK(1);
    5|
    6|D1 = DTYPE(CK, ST, CLR, D1.QBAR);
    7|D2 = DTYPE(D1.QAR, ST, CLR, D2.QBAR);
                        ^
            SEMANTIC ERROR: Invalid port
---
Line 8:
    5|
    6|D1 = DTYPE(CK, ST, CLR, D1.QBAR);
    7|D2 = DTYPE(D1.QAR, ST, CLR, D2.QBAR);
    8|D3 = DTYPE(D2.QBAR ST, CLR, D3.QBAR);
                          ^
              SYNTAX ERROR: Expected close bracket, or
number/device name list as parameter

Number of errors: 2
```

Error reporting: Offending lines of code shown here

Automatic circuit diagram: Circuit connections colour: 0: Black, 1: Red, Monitors: Blue



Appendix D: Final Team Folder

Python program files

The folder contains the program files, the basic functions of which are listed here:

`devices.py`: Used to make devices and functions, as well as setting switches and starting up clocks and signal generators.

`draw.py`: This file contains a class `Diagram` which draws SVG diagram of logic circuit. Used in the Logic Simulator project to visualise circuits for analysis. This class is imported by `gui.py` and is only used if the network executes correctly.

`gui.py`: Produces the main GUI window, along with additional windows that are opened via buttons, as well as allowing the user to set switches and monitors, and edit, open and save files.

`logsim.py`: The file run to start the logic simulator, which handles the command line arguments and uses them to determine if the siulator should be run on the command line or via the GUI.

`monitors.py`: Used to create, remove, and interact with monitors, which allow the simulator to display the signals from any point in the logic circuit.

`names.py`: Contains the `Names` class which maps name strings to IDs and vice-versa. Also returns unique error codes when requested.

`network.py`: Used to build and execute the network, by making connections between devices and executing device behaviour.

`parse.py`: Contains the `Parser` class which parses symbols from the scanner. The public method `Parser.parse_network` firstly checks these symbols for syntactic correctness, then passes instructions to the `Devices`, `Network` and `Monitors` classes. These classes may return semantic errors to the parser.

`scanner.py`: Used to read in symbols from the file, skip comments, and print or return the current line(s) for error reporting purposes.

`userint.py`: Used to enter commands to control the operation of the logic simulator once it is running.

Python test files

Additionally, the folder contains a number of test files:

`test_devices.py`: Tests the functionality of the `devices` module.

`test_monitors.py`: Tests the functionality of the `monitors` module.

`test_names.py`: Tests the functionality of the `names` module.

`test_network.py`: Tests the functionality of the `network` module.

`test_parse.py`: Tests the functionality of the `parser` module.

`test_scanner.py`: Tests the functionality of the `scanner` module.

Example files

There are two folders of example files:

`examples` (Folder): Folder containing example definition files for parsing. These are used in the `test_parse.py` file for pytests. They are also useful for quickly checking whether changes made to the code run correctly. Some of these examples files parse correctly, some have errors.

- Files which parse: `file_1.txt`, `file_4.txt`, `file_6.txt`, `file_7.txt`, `file_11.txt`.
- Files which shall not parse: `file_2.txt`, `file_3.txt`, `file_5.txt`, `file_8.txt`, `file_9.txt`, `file_10.txt`.

`examples_scan` (folder): Contains a few files used to test the scanner functionality.

Other

There are also the following items:

`requirements.txt`: Contains the package requirements of the logsim simulator. These may be installed using `pip install -r requirements.txt`.

`locale/fr/gui.mo`: A machine object file containing GUI message translations from English to French. This catalogue of translations is automatically added to the wxPython Locale if the machine is set up French.