Group 12 GF2 First interim report

Ali Ahmed, Jakob Bull, Charlie Marcus

May 2021

1 Introduction

In this task we will design a logic simulator in Python 3. To this end, we will first design a logic description language, then design and later implement parser and scanner modules in order to simulate the logic defined by our language. Additionally we will design a graphical user interface for this simulator using wxPython. Our team is composed of Ali Ahmed, Jakob Bull and Charlie Marcus, three Trinity College engineering students.

2 Teamwork planning

Our teamwork is organised as outlined in the Gantt chart in Figure 1. Deadlines for design of code are rather generous to allow for sufficient thought. However, we expect that team members should be able to use these to get ahead of schedule. Given the short time scale of the project our progress will be measured only using the provided report deadlines. However, we will also meet every two days as a team and will use this to make sure we are on track/ answer any questions/ help struggling team members. Within the Gantt chart deadlines are scheduled such that the deadline lies on the last full day of work available before the deadline, ie. we expect to finish work indicated on the chart at the latest midnight before the deadline. Additionally, we have scheduled the full day before the deadline to be ”free”, in order to allow for unexpected bugs/ problems in the code.

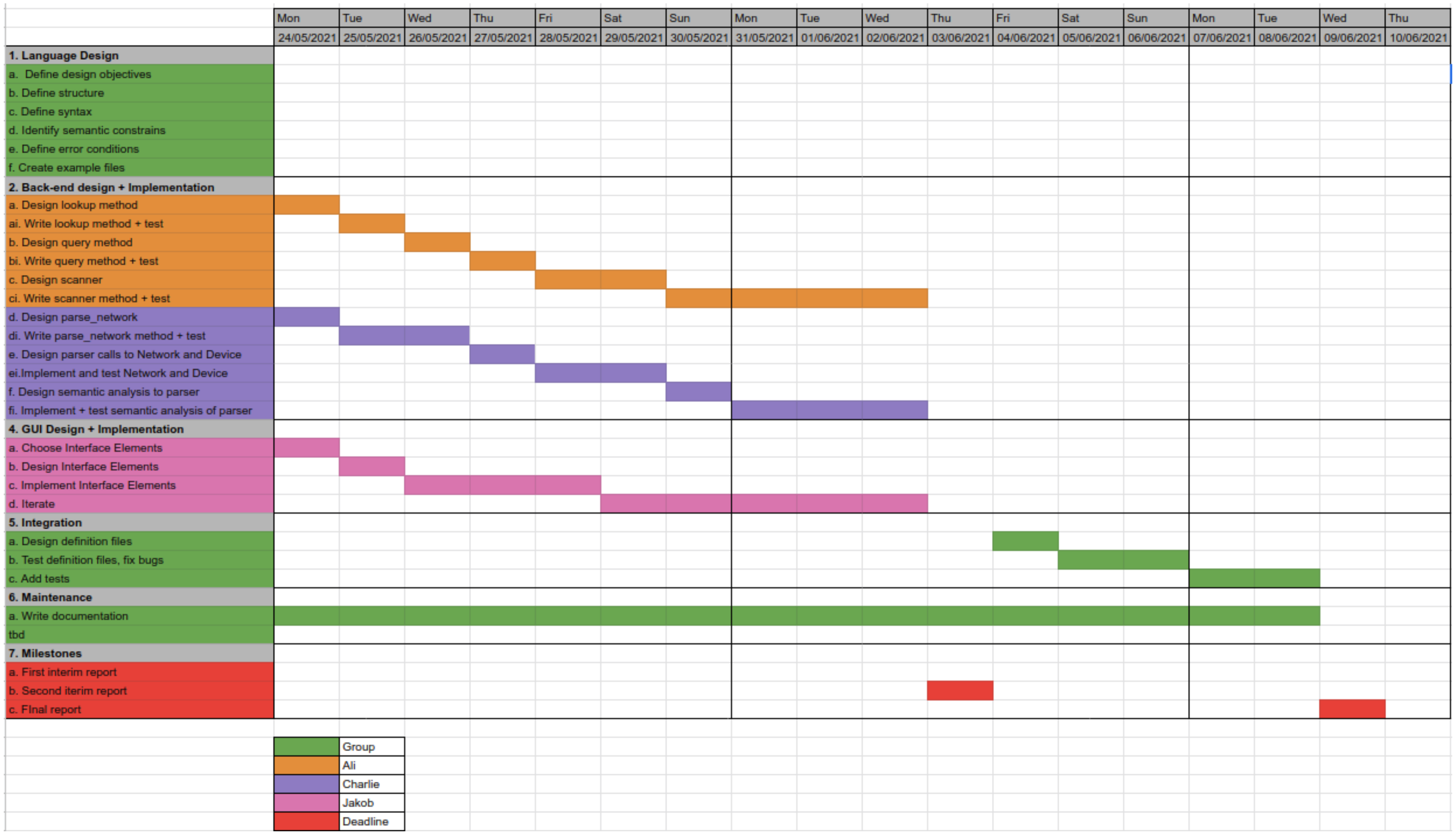

Figure 1: Project Gantt chart

3 Introduction to our logic language

Key design paradigms:

- LL1 grammar
- Adequate complexity
- Readability
- Modularity
- Ease of writing
- Redundancy
- Low capacity for semantic errors

The list above outlines the key design paradigms that we aimed to fulfil, whilst designing our language (in approximate order of importance). A definition of the terms and explanation of our approach towards them follows. The explanation of an LL1 grammar is omitted for brevity, as it bears no importance on design choices other than that the grammar simply must always be LL1.

Adequate complexity:

Our grammar must be able to express logic networks of arbitrary complexity (limited by memory only). We thus chose a strict logical hierarchy where definitions (call this D1 here) are made up of some syntax and further definitions (call one of these D2 here), where D2 is by necessity less expressive and more closely defined than D1. Ultimately at the lowest level of our language D2 is a well defined component of the network, with simple, reliable and repeatable performance that is considered known. All the D1 in contrast are considered unknown and their performance is determined at runtime. Additionally, just below the network level (ie. the entire definition file) we chose four sub-levels according to their use. First, the Devicelist defines the components of the network. Then the Connectionlist defines the connections within the network. Finally, the Signallist defines input signals and the Monitorlist defines output signals to monitor. We chose to go to this structure as it made sure all components/ signals were defined before being connected/measured, as well as because this structure made sense from how they are used.

Readability and ease of writing:

This is of critical importance. Any language is fundamentally a tool for communication and will only be used if it is not only expressive but also easy to use. Whilst ease of writing is important our main concern was readability. This was such that written files could easily be inspected, both to understand the logic network defined and check for typos. In order to achieve this we chose syntax that closely resembles the english language. Moreover, critically in the Connectionlist we define connections ordered by component (ie. for components A,B we defined all the inputs and outputs of A and then all those of B). This design was chosen as it follows the structure by which a human would inspect the network. Additionally it adds redundancy, as most connections are defined twice, that can be used for error checking.

Modularity:

The key design choices here were separating devices, connections, input signals and monitored signals. This makes it easy to re-use for instance the old devices and connections with new signals. Additionally, defining connections ordered by device, allows for easy substitution of network components.

Redundancy:
This was mentioned above already. Moreover, defining the number of inputs/outputs of a device as well as later defining those in- and outputs adds more redundancy.

Low capacity for semantic errors:
This was particuarly challenging. At first we aimed for a very constrained grammar. However, after much delibaration we decided to instead go for one that was very precise. This means that whilst the syntax allows for semantic errors we will be able to write simple functions to reject those.

4 EBNF for syntax

network =”NETWORK”, ”{”, devicelist, connectionlist, signallist, monitorlist, ”}”;
devicelist = ”DEVICES”, ”{”, {device}, ”}”;
device = name, ”is type”, devicetype, ”with”, number, ”inputs and“ | ”input and”, number, “outputs” | ”output”, ”;”;
devicetype = ”CLOCK” | ”SWITCH” | ”AND” | ”NAND” | ”OR” | ”NOR” | ”DTYPE” | ”XOR”;
connectionlist = ”CONNECTIONS”, ”{”, {connection}, ”}”;
connection = name, ”{”, ”INPUTS”, ”{”, {input}, ”}”, ”OUTPUTS”, ”{”, {output}, ”}”, ”}”;
input = ”input”, name, ”.”, port, ”is connected to”, name, ”.”, port, [{”and, name, ”.”, port}], ”;”;
output = ”output”, name, ”.”, port, ”is connected to”, name, ”.”, port, [{”and, name, ”.”, port}], ”;”;
signallist =”SIGNALS”, ”{” [{setsignal}] | [{setclock}] | [{setoutput}], ”}”;
monitorlist = ”MONITOR”, ”{”, {monitor}, ”}”;
setsignal = ”set”, name, ”.”, port, ”to be”, binarynumber, ”at time”, number, ”;”;
setoutput = ”set”, name, ”.”, port, ”to be”, binarynumber, ”;”;
setclock = ”initialize”, name, ”.”, port, ”to be”, binarynumber, ”at time”, number, ”with period”, number, ”changing first after”, number, ”;”;
binarynumber = ”0” | ”1” ;
monitor = ”monitor”, name, ”.”, port, ”;”;

5 Semantic constraints

device:
name given to a new device must not have been used before
either the number prior to ”input|inputs” or the number prior to ”output|outputs” have to be greater than 0
the number prior to ”input|inputs” and the number prior to ”output|outputs” must not be less than 0
if devicetype has a fixed number of inputs and outputs this must match the numbers prior to ”input|inputs” and ”output|outputs” respectively

connection:
name must have been previously defined
number of input|inputs and number of output|outputs must match the device definitions

input:
port to the left of ”is connected to” must be a valid input port for the devicetype of the name that comes prior to it
port to the right of ”is connected to” must be a valid output port for the devicetype of the name that comes prior to it

output:
port to the right of ”is connected to” must be a valid output port for the devicetype of the name that comes prior to it
port to the left of ”is connected to” must be a valid input port for the devicetype of the name that comes prior to it

setsignal:
name.port must be a defined input port, not connected to an output port
devicetype of name must be SWITCH
number must not be less than 0

setoutput:
name.port must be a defined port

setclock:
name.port must be defined
devicetype of name must be CLOCK
number must not be less than 0 for both number values
the second number must not be greater than the first number

monitor:
name.port must be defined

6 Error handling

During parsing of the file our compiler will first check for syntax and then for semantic error. The aim is to show exactly to the word, where an error has occured. Given the specificty of our semantic constraint this should usually be possible for semantic errors. However, if the format of the input completely deviates from out EBNF, it may be hard to pinpoint syntax errors. Nonetheless, due to the structure of our language, even then we should be able to at least highlight the line on which the error has occured. In general after an error is thrown we plan on resuming parsing after the next semicolon. Should this approach fail however, we also plan on using our strict bracketing system to aid parsing (ie. resume once encountering the expected number of closed brackets).

Syntax Errors
These occur during parsing when the syntax deviates from our grammar. We will write a function that throws an error as soon as the parser encounters a Syntax error and returns the location of that error, so that we can pinpoint its location. Parsing can then be resumed after the next semicolon.

Semantic Errors
First, it is worth noting that we have missed some possible semantic errors that could occur in this report. However, we expect that we will encounter many of these during testing such that we can add constraints for them. We expect that the structure of our language is modular enough to deal with any such issues that arise. To test for semantic errors, we will write methods for each of the semantic constraints. During parsing, if a semantic error is encountered the method will return the error, along with the location of the error and the violated constraint. This will require us to adjust the symbol class so as to include the line number and position on the line where the symbol occurs. By adding the location where errors occur in the error message, user experience will be improved when it comes to debugging. Again, parsing can be resumed after the next semicolon.

7 Example 1

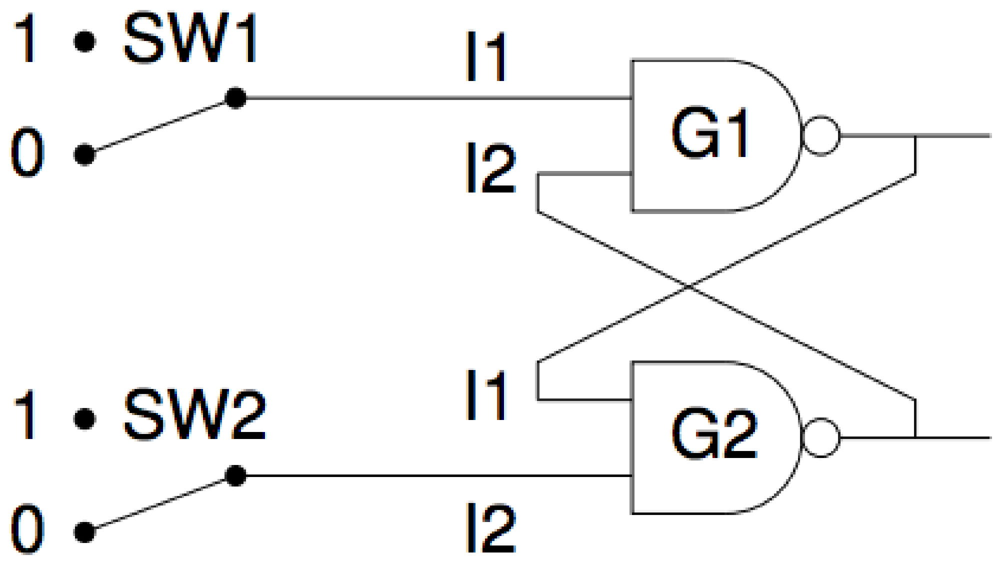

Figure 2: Circuit 1

```
NETWORK{
    DEVICES{
        SW1 is type SWITCH with 0 inputs and 1 output;
        SW2 is type SWITCH with 0 inputs and 1 output;
        G1 is type NAND with 2 inputs and 1 output;
        G2 is type NAND with 2 inputs and 1 output;
    }
    CONNECTIONS{
        SW1 {
            INPUTS{}
            OUTPUTS{
                output SW1.O1 is connected to G1.I1;
                }
            }
        SW2 {
                INPUTS{}
                OUTPUTS{
                    output SW2.O1 is connected to G2.I2;
                }
            }
        G1  {
                INPUTS{
                    input G1.I1 is connected to SW1.O1;
                    input G1.I2 is connected to G2.O1;
                }
                OUTPUTS{
                    output G1.O1 is connected to G2.I1;
                }
            }
        G2  {
                INPUTS{
                    input G2.I1 is connected to G1.O1;
                    input G2.I2 is connected to SW2.O1;
                }
                OUTPUTS{
                    output G2.O1 is connected to G1.I2;
                }
            }
    }
    SIGNALS{
        set SW1.O1 to be 1 at time 0;
        set SW1.O2 to be 1 at time 0;

        //initializing outputs
        initialize G1.O1 to be 1;
        initialize G2.O1 to be 1;
    }
    MONITOR{
        monitor G1.O1;
        monitor G2.O2;
    }

}
```

8 Example 2

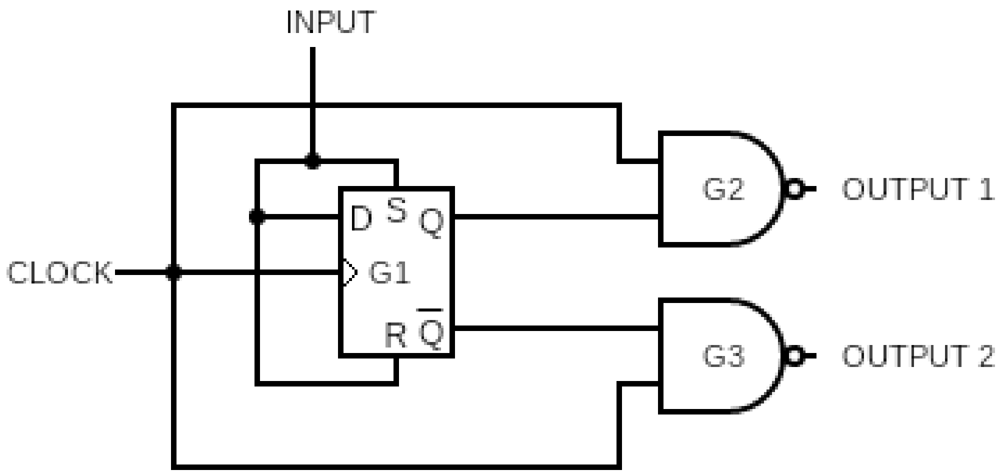

Figure 3: Circuit 2

```
NETWORK{
    DEVICES{
        CLK is type CLOCK with 0  inputs and 1 output;
        G1 is type DTYPE with 4 inputs and 2 outputs;
        G2 is type NAND with 2 inputs and 1 output;
        G3 is type NAND with 2 inputs and 1 output;
        }
    CONNECTIONS{
        CLK {
                INPUTS{}
                OUTPUTS{
                    output CLK.O1 is connected to G1.CLK and G2.I1 and G3.I2;
                }
            }
        G1  {
                INPUTS{
                    input G1.SET is connected to G1.DATA and G1.CLEAR;
                    input G1.DATA is connected to G1.SET and G1.CLEAR;
                    input G1.CLEAR is connected to G1.SET and G1.DATA;
                    input G1.CLK is connected to CLK.O1;
                }
                OUTPUTS{
                    output G1.Q is connected to G2.I2;
                    output G1.QBAR is connected to G3.I1;
                }
            }
        G2  {
                INPUTS{
                    input G2.I1 is connected to CLK.O1;
                    input G2.I2 is connected to G1.Q;
                }
                OUTPUTS{}
            }
        G3  {
                INPUTS{
                    input G3.I1 is connected to G1.QBAR;
                    input G3.I2 is connected to CLK.O1;
                }
                OUTPUTS{}
            }
    }
    SIGNALS{
        set G1.SET to be 1 at time 0;
        set CLK.O1 to be 1 at time 0 with period 10 changing first after 4;

        //initializing outputs
        initialize G2.O1 to be 0;
        initialize G2.O1 to be 0;
    }
    MONITOR{
        monitor G2.O1;
        monitor G3.O2;
    }
}
```